Extending the Hierarchical Deep Reinforcement Learning framework

Stefano Gatto B.Sc. in Computer Science

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science Graphic and Vision Technologies

Supervisor: Mads Haahr

September 2018

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Stefano Gatto

September 7, 2018

Permission to Lend and/or Copy

Acknowledgments

Firstly, I want to thank my supervisor, Dr Mads Haahr, for his guidance and advice throughout the dissertation. Dr Haahr's support was an invaluable asset to shape and develop this research, and his insights helped me progress when I was in difficulty.
I would like to express my sincerest gratitude to Kristian Hartikainen, co-author of the baseline of our research, for helpin me understand and use his code.
I want to thank my mother, my father, and my brother, for their eternal support and company.
I want to thank all my classmates, for making this year so interesting and fun. Special thanks go to Chuka, Adwi and Anant, for their dodginess. Lastly I want to thank my flatmates, for their company and the great food.

Stefano Gatto

University of Dublin, Trinity College
September 2018

Extending the Hierarchical Deep Reinforcement Learning framework

Stefano Gatto, Master of Science in Computer Science

University of Dublin, Trinity College, 2018

Supervisor: Mads Haahr

Deep Reinforcement Learning (DRL) allows to train an agent to perform a task in an environment. The generality of this concept makes DRL a very powerful tool, applicable to a multitude of fields. Self-driving cars, robot locomotion, drone control and video games AI are only some of the fields which currently use and research DRL. To train an agent to perform a task, a DRL algorithm learns a policy that maximizes the future expected reward, thus ensuring that the agent will act optimally with respect to a reward signal. When this reward is very sparse or the task to learn is very complex, it is hard for a DRL algorithm to learn to perform it. When faced with a complex task to perform, the human brain decomposes the actions to take into simpler ones, developing a hierarchical understanding of the problem. Hierarchical Deep Reinforcement Learning (HDRL) brings this type of understanding to DRL models, allowing them to tackle also very complex and sparse reward tasks.

SAC-LSP is a HDRL algorithm that allows to grow hierarchies of policies in bottom-up layer-wise fashion. Each layer of the hierarchy is trained to perform the main task, and even if the lower layers are not able to completely solve it, they "make the job easier" for the higher layers. SAC-LSP yielded state-of-the-art results on a series of simulated locomotion and control environments. In this research, we propose four different optimizations to SAC-LSP, and evaluate their performance. The main contribution of this work is Distributed SAC-LSP, an optimization that outperforms the baseline by 40%.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation explores the efficiency of Hierarchical Deep Reinforcement Learning (HDRL) in a simulated locomotion environment. The main contribution is a novel algorithm, in which a hierarchical framework takes advantage of popular techniques like prioritized experience replay and distributed reinforcement learning. We first test each technique singularly on the baseline, and then combine the best ones to create our final algorithm. This chapter introduces the research by first explaining why Deep Reinforcement Learning (DRL) is relevant and what motivates this work. It then lists the main objectives of our research, and finally it explains the structure of the rest of the dissertation.

1.1 Motivation

Artificial Intelligence (AI) is an immensely broad field with lots of challenging problems and application in many different areas. In the collective imagination though, the first and main goal of artificial intelligence is the achievement of Artificial General Intelligence (AGI) i.e. the creation of an AI able to carry out any function that a human being can. Recent progress in machine learning seems to have narrowed the gap between current technology and AGI, with deep neural networks that can perform human-like tasks like understand the content of an image by looking directly at the raw pixels, understand and translate the context of a text and even listen to human vocal requests and answer coherently. Nevertheless supervised deep neural networks are bound to certain constraints: huge amounts of labeled independent and identically distributed (i.i.d.) data are needed for the training phase, networks trained for a specific task are usually unable to perform any different task, and the whole learning process is essentially off-line. This kind of learning is similar to that of a student who reads a book over and over until he has mastered its contents. The most common way in which the human brain learns, though, is essentially on-line and interactive, storing information about what is happening around it and reacting consequently, learning at the same time from the mistakes made during the process.

Reinforcement Learning (RL) is the area of machine learning that deals with creating agents able to take the best actions in an environment with respect to a certain goal. The paradigm of RL is based on a trial-and-error approach, in which learning depends on negative and positive rewards obtained by performing actions in an environment – the actions, or series of actions, that led to a positive reward will be repeated more often, whereas those that led to a negative reward will be avoided in the future. In recent years, DRL has allowed to tackle challenging problems like locomotion, classic games, video games, manipulation and control, yielding previously unreachable results. The main difference to classic RL, is that deep neural networks are used as function approximators during the learning process. The introduction of deep neural networks in the paradigm allows to learn directly from the “real” raw inputs, like a camera view or the pixels of a screen. The high instability of reinforcement learning when approximated with a nonlinear function like a neural network had in the past caused a halt in the study of DRL. Since the seminal papers by *Mnih et al.* [7, 1], which analyzed the causes of such instabilities and proposed effective methods to stabilize the learning process, very significant results have been achieved [8]. DRL is currently evolving at very fast pace, with papers that every few months are able to surpass the results of the previous state-of-the-art by up to an order of magnitude.

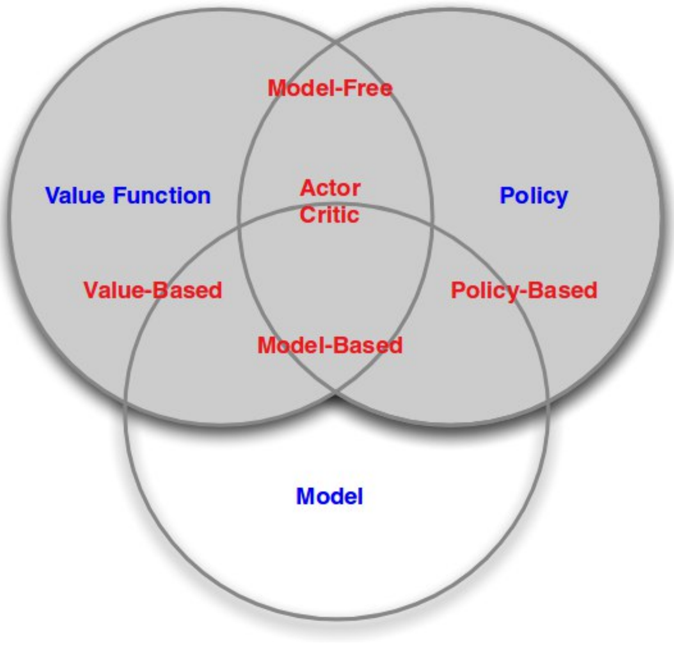

Figure 1.1: Basic taxonomy of reinforcement learning methods. Sourced from (http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html)

RL can be naturally divided in a taxonomy of sub-categories depending on some core characteristics which will be briefly explored in Section 2.1. Between those, HDRL is often considered to be one of the categories with more potential. This intuition comes from the way the human brain is able to learn primitive simple information and abstract them to reason at more complicated levels. For example, to learn how to move an object from one position to another, we first need to learn how to move our arms, and how to use them to move an object in some direction. Once those two low-level concepts are learnt, it is easy for the human brain to use them to learn how to move an object, which is a way more complex action. Furthermore, it has been found that when deep networks are applied to computer vision tasks, hierarchical structures tend to emerge [9]. A DRL algorithm that is similarly able to automatically "grow" and learn hierarchies, would also be able to obtain better results in sparse rewards environments. As aforementioned, the learning process depends strictly on the rewards obtained. If such rewards are very sparse, though, it is hard for an algorithm to figure out what actions – out of the long sequence of actions it has taken – helped it to effectively obtain that reward. For example, imagine a setting in which an agent has to navigate a maze of rooms to search for a key and use it to escape from the maze. If a human explores until finding the key the first time, it would be easy for him to understand what actions led him to obtaining the key and repeat them the next time, even if rewards are obtained only on collection and usage of the key. On the contrary, for an algorithm it would be very hard to understand out of all the actions it took, which actions made him obtain the key. If the algorithm is not able to back-propagate the reward to the right actions, it will probably end up just trying to repeat the behavior he showed right before obtaining the key. A hierarchical understanding of the problem, could help the algorithm to generate sub-goals to the task it must complete, like that in order to get to the key it first needs to get to the room where the key is, and that in order to do that it must move in a specific direction. Such sub-goals could be used for the learning of the lower hierarchies of the algorithm, whilst the higher hierarchy keeps trying to solve the main goal, i.e. to get to the key and use it. Nonetheless, it is still not clear how to to make such hierarchies arise in the reinforcement learning paradigm. While in some cases the researcher is able to define task-specific hierarchies and subgoals, the automation and generalization of how to "grow" task-agnostic hierarchies is still an open problem. Such reasons led us to choose Soft Actor-Critic with Latent Space Policies (SAC-LSP) [2] as the baseline for this research. SAC-LSP is a HDRL algorithm that can construct hierarchical representations in bottom-up layer-wise fashion. Each layer is augmented with latent space variables, and since only invertible mappings from latent variables to actions are used, all the layers remain unconstrained in their behavior, contrarily to other HDRL algorithms which cripple or limit the expressiveness of some layers in order to grow a hierarchy [10, 11, 12, 13]. Furthermore, since the higher layers of the hierarchy use the latent variables of the policy layers below it as action space, the hierarchies can be grown both automatically (using the same reward for all layers) or using handcrafted lower-layers shaping rewards.

Although the different types of RL methods can differ substantially from an architectural point of view, many improvements in the field were obtained through the introduction of techniques that are agnostic to the underlying architecture. Furthermore it has been recently proved that many of such optimizations are orthogonal to each other and hence can be combined [14, 5]. It is our expectation that some of these optimizations will become more and more common in most DRL algorithms, and it is thus important to study their effectiveness on a still not deeply explored field like HDRL. For such reason we choose four promising techniques and try to adapt them to SAC-LSP. The first technique we try is categorical RL, which in recent works [3, 15, 5, 14] yielded state-of-the-art results on a set of Atari 2600 games. While normally DRL algorithms try to learn the expected future reward, categorical RL tries to learn the approximate distribution of rewards. This has different advantages, like stabilizing the learning process and preserving multimodality in value distributions. More generally, we can simply imagine that since we learn a whole distribution, our algorithm produces at every step a set of auxiliary predictions. This means that even if the main prediction the algorithm does is wrong, it is still able to learn thanks to the auxiliary ones.

The second technique we try is probably one of the most famous. Prioritized experience replay [16] modifies the way mini-batches are sampled during the learning phase, so that the most rare and unexpected experiences are sampled more often. This allows for much better data efficiency, and the positive performance impact of utilizing such technique has been demonstrated in different cases and on multiple algorihtms [16, 14, 5].

The third technique is similar to the previous one in that it goes to modify the experience replay mechanism. Hindsight experience replay [17] modifies the normal reward system by adding auxiliary goals to achieve. These goals are derived from states that the agent will reach in the future, and help the algorithm to learn even when the reward is sparse or the state space dimensionality is very high. The intuition behind it is that humans learn from failures as much as they learn from successes. By augmenting our environment with auxiliary rewards, hindsight experience replay tries to give RL that same ability to learn from failures. Furthermore, this technique can become particularly interesting in the case of HDRL, like recently proved by [18].

The final technique we test is distributed RL. A lot of the recent advances in deep learning have been possible thanks to the high parallelization that modern hardware allows. Making good use of graphic cards and distributing the computation over more machines allows to yield much better results in a fraction of the wall-clock time. Applying similar techniques to DRL has been one of the main objectives of research, although results have often been limited respect to the one obtained by deep learning. This is due to the sequential nature of reinforcement learning, that needs to act and collect experience in order to learn from it. Ape-x [4] is a recent approach that was able to greatly outperform all previous state-of-the-art methods. With this method, a number of worker processes are created, each of which acts independently in its own copy of the environment. By decoupling the classic RL learning loop into two different components, the ape-x framework is able to efficiently parallelize the collection of experience across multiple machines. This method allows for much greater data efficiency, as well as it allows to take greater advantage of graphic cards and multi-processing. Furthermore, since each process collects data independently from the others, this allows for greater exploration as also previously suggested in [19].

The choice of an adequately challenging and useful environment is essential to test DRL algorithms. Since the aim is to create algorithms able to solve complicated tasks while at the same time generalizing over a different number of problems, the environments must offer a vast number of tasks of varying difficulty with similar interfaces. Although many are available, currently Atari Learning Environment (ALE) [20] and MuJoCo [21] are the two main choices as learning environments for DRL. ALE is a suit of Atari 2600 games, each of which has a RAM version (since all the logic about an Atari 2600 game is loaded in the RAM during execution) and a raw-pixel version. The latter is particularly interesting, because it forces the DRL algorithm to learn directly from the game frames, which means that the algorithm has the same inputs a human would, i.e., the screen and the controller buttons. Another important trait of ALE is that games have discreet action space, since by pressing a button on a controller, a specific action is performed. MuJoCo instead is a physics engine tailored to model-based control. It allows for efficient and optimized forward and inverse dynamics, as well as precise and realistic physics simulations. As such, it is commonly used in DRL for robotic simulation and locomotion. In the first case, the usual environment is a robotic arm that has to complete certain tasks like moving or stockpiling objects. In the latter, a series of models with an increasing number of degrees of freedom are put in a environment in which they have to learn to walk or even navigate a maze. The main difference respect to ALE is that in MuJoCo the action space is continue and much larger, as at each frame we must specify how much and in which direction we are moving each joint of the model.

1.2 Objectives

SAC-LSP is an actor-critic off-policy algorithm, and its original implementation is designed to be tested on a series of locomotion tasks based on the MuJoCo environments. Such environments offer complex tasks to learn in a continuous action space environment. Our aim in this research is to try to extend SAC-LSP with previously successful techniques. In further details our objectives are:

- To extend the algorithm to the categorical RL framework and test its performance.
- To extend the algorithm to use prioritized experience replay and test its performance.
- To extend the algorithm to use hindsight experience replay and test its performance.
- To extend the algorithm to have a distributed computation similar to that of the ape-x framework, and test its performance.

The details of such techniques and their advantages will be explained in Section 2.1

1.3 Roadmap

The second chapter of this dissertation is dedicated to a thorough review of the relevant literature. As DRL is a recent and complex field, particular attention will be dedicated in the first section of such chapter to explaining how the basics of DRL works, with respect to some of the most successful DRL algorithms. The second part of this chapter will instead talk about the state-of-the-art, although it will concentrate mainly on the related work to not confuse the reader diverting his attention from the technologies significant to our research.

Chapter 3 presents in greater details the design and functionalities of the chosen baseline, and presents the modification proposed to extend its functionalities.

Chapter 4 dives into the details of the actual model implementation, presenting the problems and limitations that we had to address during our research.

In chapter 5 we present an evaluation of the model extensions through a series of experiments, first comparing each single extension to the baseline, and then summarizing the results in a general performance comparison of all the models.

Finally, chapter 6 takes stock of the research presented, summarizing the contribution made and the most interesting results, as well as suggesting interesting directions for future work.

Chapter 2

State of the Art

In this chapter we will review the state-of-the-art in Deep Reinforcement Learning. As DRL is a wide field in full expansion, choices must be made regarding what technologies and recent advances to mention in this chapter. To give the reader a good understanding of the research presented, we will divide this chapter in two main sections. The first section will explain how reinforcement learning works, and present some of the main successes of DRL. The second section will instead present the related work, i.e. the papers that inspired and influenced the most this research.

2.1 Background

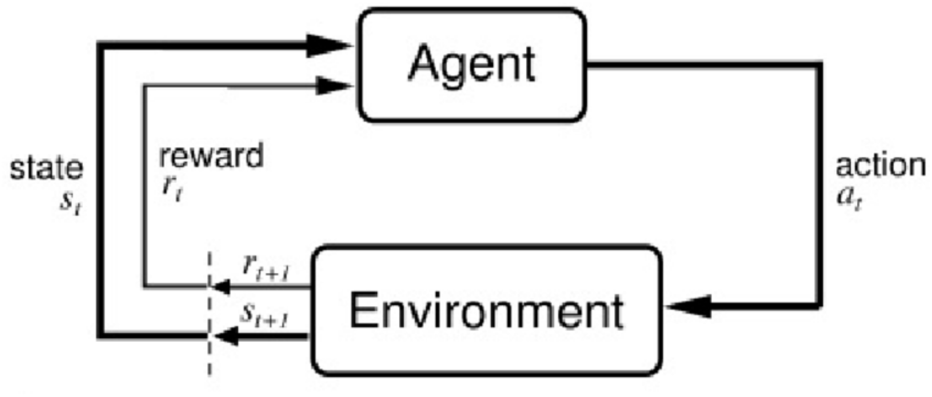

Figure 2.1: A scheme illustrating the basic functioning of the RL loop. Sourced from (http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html)

Before diving into the details of specific DRL algorithms, it is important to draw the general picture about the reinforcement learning framework and define some basic terminology. As briefly aforementioned, RL algorithms try to learn to perform a certain task in an environment through a trial-and-error process. The basic idea is simple: an agent is placed in an environment, where it will perform a random action and, depending on the type of reward it will obtain from such action, it will keep or avoid doing it in the future, so to try to maximize the total reward over time.

2.1.1 Terminology

Thus far, we have already informally introduced different main terms of reinforcement learning. We will now list the more important ones, which will be from now on used in the text with this specific meaning unless differently stated.

- Agent: The agent is the component that takes actions in the environment. In the case of a video game it would be the controllable character, in a locomotion scenario it is likely to be a real or simulated robot. In our tests, it will be a simulated ant-shaped robot.

- Environment: The environment is the “world” in which the agent acts, like the real world for a drone or the game logic for a video game. Nonetheless, when we refer to the environment we usually refer to the representation of such environment given to the agent. The environment is therefore usually a function or class that takes the agents current state and action as input, and returns as output the agents reward and next state.

- Action: Unless stated differently, we will refer to sets with capital letters and elements of such sets with the corresponding lowercase. A will then be the set of all possible actions the agent can make in its current environment, and a will be one of such actions. It is important to note that in many environments a no-op – i.e. not doing anything for a timestep – is a valid action.

- Reward: An agent taking an action a at time t will receive a reward r at time $t + 1$. It is important though to understand that, depending on the environment and the action, it is probable that the reward received is not the fruit of just the last action taken, but of a set of actions executed in the past. In some cases, the last action will not even have any relation with the reward obtained. Imagine for example the classic Atari game Breakout, where we move a base to make a ball bounce on some bricks. The rewards will be obtained when the ball hits the bricks, although the actions just before that moment would normally just be no-ops, and the only actions that were really imporant were the ones taken to hit the ball with the base.

- State: A state s is the representation of the environment at a specific timestep. It therefore represents the environment - comprised of the agent, what action it is taking and what effect it has on the environment - at a specific moment and place. An observation is the agents representation of the current state of the environment. Although semantically they are slightly different, this two terms will be used interchangeably.

- Policy: The policy π is the strategy that the agent employs to decide the next action to take based on the current state. Thus, it is a function $\pi(s) = a$ that maps states to actions.

- Value: The value function $V_\pi(s)$ returns the sum of the expected cumulative reward obtained from following a policy π from a state s onwards. To limit the error introduced by using this prediction during learning, a discount factor γ is applied to future state rewards.

- Q-value or action-value: The Q-value function $Q_\pi(s,a)$ returns the expected cumulative reward obtained from taking an action a in a state s, and following a policy π from the next state s' onwards. In other words

  $$Q_\pi(s,a) = r(s,a) + V_\pi(s')$$

  where $r(s,a)$ is the reward obtained for taking action a at state s.

- Experience or transition: A single experience is a tuple (s,a,s',r) which tells us what action was taken in which state, what state we reached by performing that action and what reward we received from it.

2.1.2 Reinforcement Learning

The goal of RL is to find the optimal policy π^*, i.e. the policy that given a state returns the action that maximizes the future cumulative reward. This is achievable if we know the optimal Q-value function

$$Q^*(s,a) = max_\pi(Q_\pi(s,a)) \tag{2.1}$$

which allows us to write the optimal policy simply as

$$\pi^*(s) = argmax_a(Q^*(s,a)) \tag{2.2}$$

Reinforcement learning can be formalized as a Markov Decision Process (MDP) or, since often the environment in not completely observable – i.e. the observations do not capture all the information about the current state of the environment – as a Partially Observable Markov Decision Process (POMDP). An important feature of MDPs is the Markov property, which states that the probability distribution of future states depends only on the current state and not on the previous ones. This is equivalent to saying that the current state contains all the “relevant information” about previous states that could influence future ones, and the probability distribution of future states is therefore independent from all states except the current one. Being able to formalize RL problems as MDPs allows to use the Bellman equation [22] to write the Q-value function recursively as

$$Q_{\pi}(s,a) = \mathbb{E}\left[r(s,a) + \gamma Q_{\pi}(s', \pi(s'))\right] \tag{2.3}$$

Since we still don’t have policy at this point, we can indirectly derive it by acting greedily on the Q-value. Choosing the highest Q-value across all possible actions at each iteration gives us the following formula, called Q-value iteration

$$Q(s,a) = \mathbb{E}\left[r(s,a) + \gamma\, max_{a'} Q(s',a')\right] \tag{2.4}$$

that is the basis for Q-learning [23]

$$Q(s,a) = Q(s,a) + \alpha\big(r(s,a) + \gamma\, max_{a'} Q(s',a') - Q(s,a)\big) \tag{2.5}$$

which is proven to converge to Q^*. Q-learning is a model-free method, i.e., a method that does not try to learn explicit models of the environment state transition and reward functions, but derives an optimal policy. Model-free methods are used in the cases where the MDP model is unknown or it is too big to use. If for example we are building an RL method to control a drone, the model is well known since the drone follows the laws of physics and kinematics, but building the model of all the transitions dynamics would be simply infeasible. These particularly difficult problems are also the ones usually tackled by DRL, and in fact most DRL methods are model-free.

2.1.3 DQN

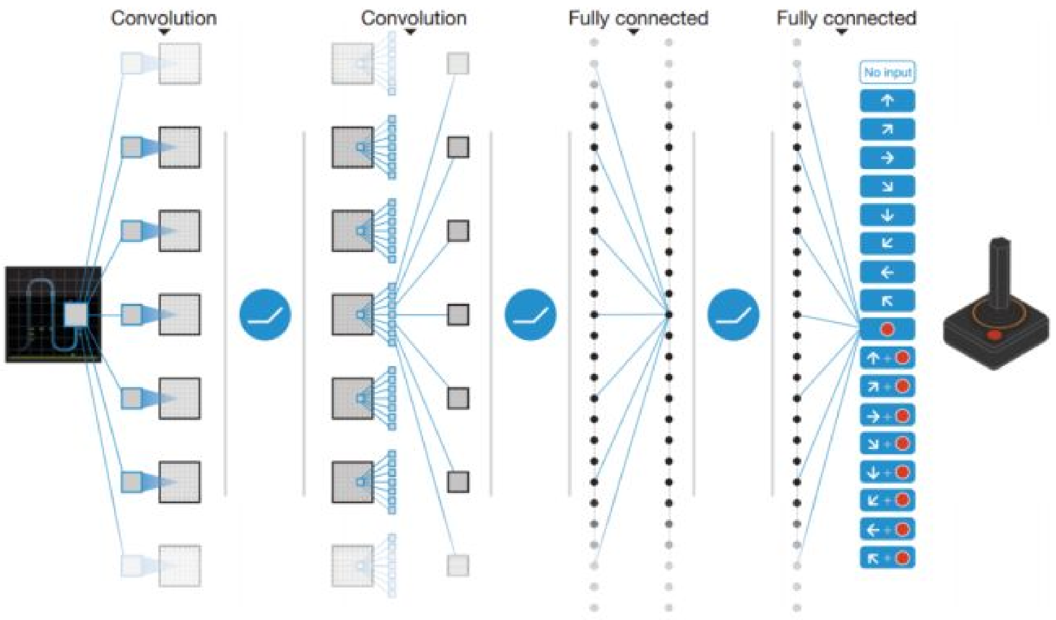

Figure 2.2: DQN's CNN is composed of some convolutional layers to understand the content of the image and then some fully connected layers, with the last one having the number of outputs equal to the number of possible actions. Sourced from [1]

We finally have the basics to introduce DQN [7, 1], the first algorithm able to learn complex tasks like playing Atari 2600 games directly from raw pixels as input, and even match or surpass the average human-expert score on most of the games. *Mnih et al.* were not the first ones to try to use neural networks as functions approximators in the RL field [24, 25] but research showed that Q-learning can hardly converge, and sometimes even diverge, when a non-linear approximator like a neural network is used [26]. *Mnih et al.* wanted to exploit the recent advances of deep learning and use a Convolutional Neural Network (CNN) to learn directly from raw pixels and build an algorithm able to generalize over different tasks. To do so, they introduced two main modifications to Q-learning that essentially try to make RL "look like" deep learning.

The differences between RL and DL are numerous. Deep learning takes large batches of labeled i.i.d. data, and takes advantage of the hardware advances in GPUs and CPUs to aggressively parallelize the gradients computation during learning. Reinforcement learning instead has to work with data that are extremely correlated due to their sequential nature. Furthermore the data labeling depends on the target network – i.e. the maxed term in Equation 2.5 – and the data distribution depends on the policy of the model. As such, both labels and data distribution change over time, leading to learning instability and eventually to the inability to converge to an optimal policy. To limit the instability introduced by the target network, DQN keeps the weights of the target network fixed, copying them from the learned network at regular intervals. This allows for both labels and policy to change more slowly without introducing extra computation. Arguably the most important modification they introduced, though, was the utilization of experience replay [27]. The underlying idea behind is very simple: every time the algorithm takes a step, it stores the transition in a ring buffer in the form of a (s, a, s', r, t) tuple, where t is simply a boolean saying whether the transition is the last of an episode or not. Then every time it has to execute a gradient update, a random batch is sampled from the experience buffer. This breaks the correlation of the training samples reducing the variance of the updates, as well as allowing for much greater data efficiency since the same transition is potentially used in different weight updates. As aforementioned, a CNN is used to approximate the Q-value function. To use it, the loss function is written in the form of

$$L(\theta) = \mathbb{E}\Big[\big(\mathbb{E}\big[r(s,a) + \gamma\, max_{a'} Q(s',a';\theta^-)\big] - Q(s,a;\theta)\big)^2\Big] \tag{2.6}$$

where θ represents the network parameters and θ^- indicates the fixed weights of the target network. Differentiating the loss function with respect to the weights, we finally derive the gradient used in the final algorithm:

$$\nabla_\theta L(\theta) = \mathbb{E}\Big[\big(r(s,a) + \gamma\, max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta)\big)\, \nabla_\theta Q(s,a;\theta)\Big] \tag{2.7}$$

2.1.4 Actor-critic methods

The main ideas of DQN have been used in most of the DRL algorithms that followed it. A part of the research concentrated on improving the performance of DQN, introducing modifications to the function approximator that would reduce the biases introduced by the Q-value function approximation [28, 29]. DQN (and its evolutions) falls into the category of value-based methods, i.e. methods that do not try to learn a full model nor they try to learn directly a policy, but instead learn a value-function from which it is possible to indirectly derive a greedy policy. This implies some drawbacks, like the biases introduced by the Q-value function approximation or the difficulty to apply such methods to a continuous action environment due to the curse of dimensionality. Thus another part of the research focused on adapting the main ideas of DQN to other RL methods, like policy-based and, primarily, actor-critic methods.

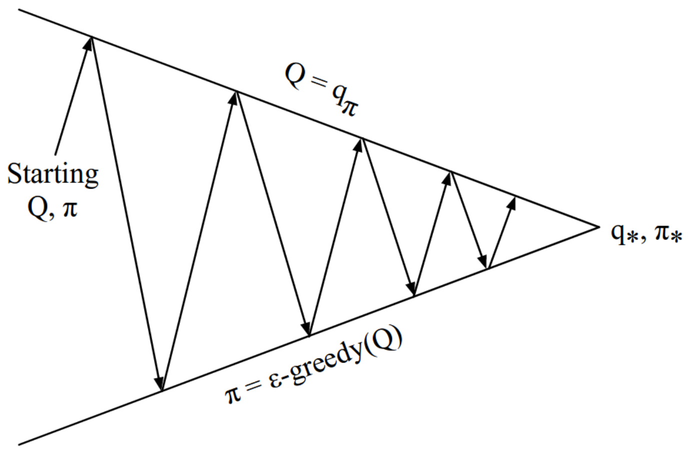

Figure 2.3: Policy iteration converges to both the optimal policy and the optimal Q-value function. Sourced from (http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html)

Policy-based methods directly learn a policy. Having an explicit policy allows to utilize the Q-value iterative function as in Equation 2.3, which means we eliminate the non-linear max operator that was limiting us to discrete spaces. We can then use the Q-value function we just estimated to improve our policy, simply doing

$$\pi'(s) = argmax_a Q(s, a) \tag{2.8}$$

This brings us to the policy-iteration algorithm [30], which alternates policy evaluation (Equation 2.3) and policy improvement (Equation 2.8). The policy improvement step, though, uses an argmax, which is just as problematic as the max operator of Equation

2.4. To obviate such problems, policy gradient methods perform direct policy search. To be able to directly improve our policy in policy space, we need to define a measure of the "goodness" of a policy. We can do that defining a score function as the expected cumulative reward of following a policy from the first state. To extend the score function to continuous environments, we take the average per-time step of the expected reward, ending up with the following formula

$$J(\phi) = \mathbb{E}[V_\pi(s)] = \sum_s d(s) \sum_a \pi(a|s\,;\phi)\; r(s,a) \tag{2.9}$$

where ϕ are the parameters of the parametrized policy, $d(s)$ is the probability of being on state s given policy π, and $\pi(a|s\,;\phi)$ is the probability of taking action a by following policy π in state s. Using the policy gradient theorem [31], we are now able to write the policy gradient as

$$\nabla_\phi J(\phi) = \mathbb{E}[\; \nabla_\phi\, log\, \pi(a|s\,;\phi)\; r_t\;] \tag{2.10}$$

where with r_t we indicate the actual cumulative return of an episode. This approach is also known as Monte Carlo policy gradient or REINFORCE [32]. As aforementioned, policy-based methods have different advantages. They work well in continuous action environments and with very large action spaces. Furthermore, while a value-based method is optimizing the value (or Q-value) function, a policy-based method is optimizing the "right" problem, i.e., the expected cumulative reward of a policy, which helps to converge faster. Nonetheless, being the gradients dependent on the sampled cumulative return of an episode, they suffer from high variance. Furthermore, policy gradients methods tend to get stuck on a local optimum.

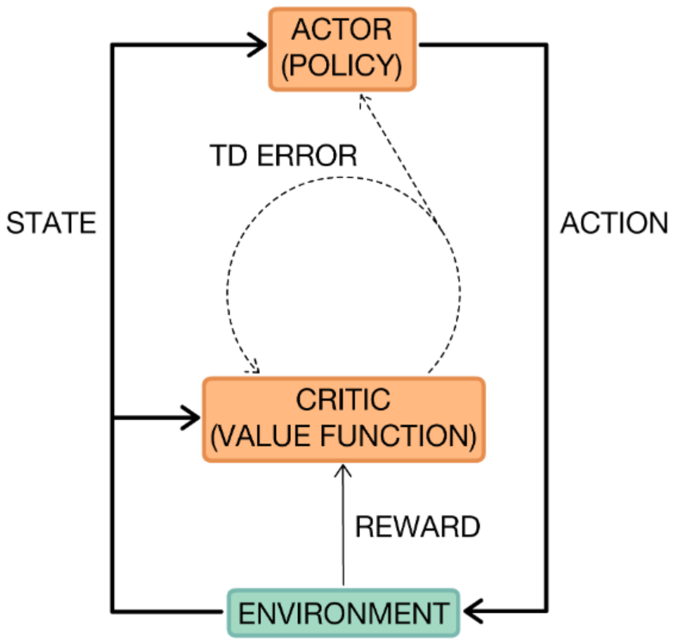

Figure 2.4: Actor-critic learning loop. The actor (policy) receives the current state from the environment and chooses what action to perform. Simultaneously, the critic (value function) receives the state and reward of the transition and uses the error calculated to perform the updates. Sourced from (http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html)

Actor-critic methods combine value functions and explicit policy representations, therefore trading off between the advantages of the two methods. As the name suggests, actor-critic methods are composed of two main components: an actor, represented by a policy network that decides how to act, and a critic, represented by a Q-network that decides how good an action was. One of the most important families of actor-critic methods derives from Deterministic Policy Gradient (DPG). In [33] *Silver et al.* are in fact able to demonstrate that the policy gradient can be obtained from the expected gradient of the Q-value function using the DPG theorem:

$$\nabla_\phi J(\phi) = \mathbb{E}[\ \nabla_\phi \pi(s;\phi)\ \nabla_\theta Q_\pi(a|s\,;\theta)\ ] \tag{2.11}$$

This allows for DPGs to be estimated much more efficiently than the usual stochastic policy gradients. Furthermore, since they don't need to integrate over the action space but only over the state space, they tend to require fewer samples in large action spaces. This work was followed a couples of year later by Deep Deterministic Policy Gradients (DDPG) [34], which applies the main features of DQN to DPG. First of all, the policy and Q-value functions where implemented with a CNN to be able to work directly on raw pixels as input. Since DPG is an off-policy method, it is also straightforward to implement an experience replay buffer similarly to DQN. In order to obtain a good learning stability and avoid divergence, the network parameters updates where slightly changed respect to DQN. Apart from fixing the weights of the Q-value network that produces the Q-value targets, also the policy network that calculates the policy targets had to be fixed. Furthermore, instead of performing the normal hard update and just copying the newer network parameters into the target networks, a soft update was introduced in the form of

$$\theta' = \tau\theta + (1-\tau)\theta' \tag{2.12}$$

where θ' are the weights of a target network, and $\tau \ll 1$, so that the values updates are ensured to change slowly. Although this may slow down the convergence, it stabilizes notably the learning process and makes RL looks even more like DL.

This section was not intended to lecture about DRL in general, but rather to give the reader the knowledge necessary to understand the learning process of a generic DRL algorithm and the techniques that will be introduced in the next section. We consider the two algorithms described above, together with the RL jargon and theoretical basics introduced, to be enough to comprehend the rest of this work. This is why other very important DRL algorithms like A3C [19] or TRPO [35], that are roughly contemporary to DQN and DDPG and also had a similar impact on the future development of the field, will not be treated here. We refer the interested readers to the references for further information.

2.2 Related Work

In this section we will introduce some state-of-the-art DRL algorithms. The focus will be on the papers that inspired this research, therefore we will describe the algorithms that we used as a baseline and the ones that introduced the techniques experimented in this work.

2.2.1 Maximum entropy framework

The main baseline for this work is SAC-LSP [2] a hierarchical deep reinforcement learning algorithm that directly evolved from previous works by the same authors, namely Soft Q-Learning [36] and the Soft Actor-Critic (SAC) [37]. These algorithms fall in the so called maximum entropy framework, which augments the standard maximum reward reinforcement learning objective with an entropy maximization term. This means that while the network will be trying to maximize the expected reward, it will also be trying to maximize its entropy. In this way, the agent tries solve the task proposed while acting as randomly as possible. Maximum entropy framework algorithms have been shown to have an improved exploration and to be able to capture multiple modes of near-optimal strategies, the latter of which entails also better results in transfer learning. The standard objective of a RL algorithm is to learn the optimal policy of Equation 2.2, which in the most general form can be written as

$$\pi^* = argmax_\pi \sum_t \mathbb{E}[\, r(s_t, a_t)\,] \tag{2.13}$$

The entropy augmented optimal policy will then be defined as

$$\pi^*_{MaxEnt} = argmax_\pi \sum_t \mathbb{E}[\, r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s))\,] \tag{2.14}$$

where α is a temperature hyperparameter that weights the relative importance of entropy and reward. Using a deep energy-based policy representation of the form

$$\pi(a|s) \propto exp(\, -\mathcal{E}(s, a)\,) \tag{2.15}$$

where $\mathcal{E}$ is an energy function represented by a universal function approximator like a deep neural network, the policy is ensured the freedom to be able to represent any distribution $\pi(a|s)$. Once the soft Q-value function is defined as

$$Q^*_{soft}(s,a) = \mathbb{E}\Big[\sum_l \gamma^l\big(r(s_{t+l},a_{t+l}) + \alpha\mathcal{H}(\pi(\cdot|s_{t+l}))\big)\Big] \tag{2.16}$$

and the soft value function is defined as

$$V^*_{soft}(s) = \alpha\, log \int_A exp\Big(\frac{1}{\alpha}\, Q^*_{soft}(s,a')\Big)\, da' \Big] \tag{2.17}$$

it is demonstrated [36] that the optimal maximum entropy policy is given by

$$\pi^*_{MaxEnt}(a|s) = exp\Big(\frac{1}{\alpha}(Q^*_{soft}(s,a) - V^*_{soft}(s))\Big) \tag{2.18}$$

Using such formulation, it is possible to write the soft Q-function of a stochastic policy π as

$$Q_\pi(s,a) = r(s_t,a_t) + \gamma\,\mathbb{E}[\,V_\pi(s')\,] \tag{2.19}$$

and the soft value function as

$$V_\pi(s) = \mathbb{E}\Big[\,Q_\pi(s,a) - log\,\pi(a|s)\Big] \tag{2.20}$$

and prove that they still satisfy the soft Bellman equation. This allows us to the define the soft Bellman backup operator, which is the core of Soft Q-learning, as $Q \leftarrow \mathcal{T}_\pi Q$, with

$$\mathcal{T}_\pi Q = r(s_t,a_t) + \gamma\,\mathbb{E}[\,Q_\pi(s',a') - log\,\pi(a'|s')\,] \tag{2.21}$$

In SAC [37] the maximum entropy framework is adapted to an actor-critic setting. Using policy iteration (remember Figure 2.3) as base to enable theoretical analysis and convergence guarantees, the soft Bellman backup operator (Equation 2.21) becomes the policy evaluation step and the policy improvement step is defined as

$$\pi_{new} = argmin_{\pi'\in\Pi}\, D_{KL}\big(\,\pi'(\cdot|s)\,||\,(Q_{\pi_{old}}(s,\cdot) - log\,Z_{\pi_{old}}(a|s))\big) \tag{2.22}$$

where the restriction $\pi' \in \Pi$ is used to ensure the policies are part of a set of tractable parametrized parameterized family of distributions like Gaussians, and $D_{KL}(x||y)$ is the Kullback-Leibler divergence, which projects the new policy in the set of desired policies. This can now be adapted to a DRL setting using a parametrized function approximator for the value function $V(s\,;\,\psi)$, the Q-value function $Q(s,a\,;\,\theta)$ and the policy $\pi(a|s\,;\,\phi)$. It is in theory not necessary to have a different network for the value function since it could be derived from the Q-value function, but in practice this increases the stability of the training [37]. With this, it is possible to derive the new goals and gradients for all the three function approximators. The soft value function is trained to minimize the squared residual error

$$J_V(\psi) = \mathbb{E}\left[\frac{1}{2}(V(s\,;\,\psi) - \mathbb{E}[\,(Q(s,a\,;\,\theta) - log\,\pi(a|s\,;\,\phi)\,])^2\right] \tag{2.23}$$

and its gradient takes the form

$$\nabla_\psi J_V(\psi) = \nabla_\psi V(s\,;\,\psi)\big(\,V(s\,;\,\psi) - Q(s,a\,;\,\theta) + log\,\pi(a|s\,;\,\phi)\,\big) \tag{2.24}$$

The Q-value function is trained to minimize the soft Bellman residual

$$J_Q(\theta) = \mathbb{E}\left[\frac{1}{2}(Q(s,a\,;\,\theta) - (r(s,a) + \gamma\,\mathbb{E}[\,V(s\,;\,\psi)]))^2\right] \tag{2.25}$$

and its gradient takes the form

$$\nabla_\theta J_Q(\theta) = \nabla_\theta Q(s,a\,;\,\theta)\big(\,Q(s,a\,;\,\theta) - r(s,a) - \gamma\,V(s\,;\,\psi)\,\big) \tag{2.26}$$

Finally, the policy parameters are learned by minimizing the KL divergence of Equation 2.22, and its gradient can be estimated using a likelihood ratio gradient estimator in the form of

$$\nabla_\phi J_\pi(\phi) = \nabla_\phi log\,\pi(a|s\,;\,\phi)\big(\,log\,\pi(a|s\,;\,\phi) - Q(s,a\,;\,\theta) - \,V(s\,;\,\psi)\,\big) \tag{2.27}$$

The learning loop of SAC alternates between collecting experience from the environment with the current policy and updating the function approximators. The updates are performed with the stochastic gradients listed above on batches sampled from a replay buffer, resulting in an off-policy method. Furthermore the algorithm is agnostic to the parameterization of the policy, as long as it can be evaluated for any arbitrary state-action tuple. The suggested parametrization is based on Gaussian mixtures models, for which we refer the reader interested on further details to the original paper [37]. Using Mujoco as testing environment, SAC substantially outperformed DDPG in both sample efficiency and final performance, and it has been seen to learn much faster than TRPO [35]. Overall, SAC yielded state-of-the-art results and an increased stability respect to methods like DDPG and TRPO.

SAC-LSP [2] extends the soft actor-critic to HDRL by using latent space policies. We already said that SAC is agnostic to the policy parametrization, and by using latent variables to determine how the policy maps states into actions, it is possible to build a hierarchy of policies in bottom-up layer-wise fashion. In this hierarchy, the latent variables of each layer act as the action space for the layer above, creating a hierarchy in which each layer can try to solve the main task and, even if it is not fully able to do so, it still makes the simplifies the task for higher layers. Furthermore, since the transformation from latent variables to actions is ensured to be fully invertible, all the layers of the hierarchy maintain complete expressiveness. Imagine for example that we want to make a humanoid robot learn how to navigate a maze. A possible tactic could be to train the lowest layer of the hierarchy on a simpler task in which the algorithm has to learn locomotion, and then train a second layer on the real maze-navigation task. It is possible in this case for the lowest layer policy to learn to maximize the speed, so that the agent will always try to move as fast as possible. This behavior might not be desirable in the maze-navigation task, since it might result almost impossible for the simulated humanoid to turn without falling when it is running at max speed. Normally, in a bottom-up layer-wise HDRL algorithm – in which first the lowest layer is trained and then is frozen in order to train higher layer – this behavior might prevent our algorithm to learn to perform the desired task. Due to the invertibility of the transformations from latent variables to actions, hough, the second layer will be able fix the behavior of the lower policy. This, united with the advantages of the maximum entropy framework, makes SAC-LSP a stable and expressive algorithm, capable of achieving state-of-the-art results in a multitude of continuous control benchmarks [2]. For further information on the design of this algorithm, we refer the reader to the next chapter.

2.2.2 Prioritized experience replay

As it should be clear by now, in DRL one of the main features for stabilizing the learning process and making the algorithm data efficient, is experience replay. The two main advantages that this technology brings are the possibility to break the temporal correlations between the experiences thanks to the uniform random sampling of past transitions for the updates, and the fact that rare and valuable experience will be used for more than one update, thus improving the data efficiency. The second advantage hints at the question: wouldn't it be better if instead of sampling uniformly, we sampled "rare and valuable" experience more often? Prioritized experience replay [16] does exactly that, modifying the experience buffer to sample more often the transitions with high expected learning progress. The ideal measure for the importance of a transition would be the expected learning progress, which is an information that we do not posses while learning. A good estimate of such value can be the magnitude of the Temporal Difference (TD) error of that transition. The TD error has been already introduced informally using Q-values estimates in Equation 2.5, as it is the difference in expected return before and after taking one step, more formally

$$TD(0) = (r(s, a) + \gamma V(s')) - V(s) \tag{2.28}$$

This tells us how far the expected return of a transition is from its next-step bootstrap estimate, or in other words how much our estimate of the future reward changes after taking a step. Using the TD error as a priority factor, we are able to sample "surprising" experience more often, and since the TD error changes during the learning process, as long as we update the TD error of a transition, we will always sample useful transitions to learn from. Since a normal experience buffer usually contains around one million transitions, though, it is not feasible to update all the TD errors at each update, thus only the sampled ones get updated. Sampling greedily only the high-priority transition would bring a number of disadvantages, like the fact that the initial TD error estimates will strongly influence the learning, sampling too often transitions with an overestimated TD error and never replaying rare experiences with an initially underestimated error. To atone for this, prioritized experience replay uses a stochastic sampling method that interpolates between greedy and uniform sampling. The probability of sampling a transition then becomes

$$P(i) = \frac{p_i^\alpha}{\sum_j p_j^\alpha} \tag{2.29}$$

where α is a hyperparameter defining how much prioritization to use. The priority can be proportional to the TD error

$$p_i = |TD_i(0)| + \epsilon \tag{2.30}$$

where ϵ is a small constant to avoid transitions to have TD error equal to 0, or rank-based

$$p_i = \frac{1}{rank(i)} \tag{2.31}$$

where $rank(i)$ is the position of transition i when the buffer's ordering is based on the TD error. In order to converge to the right solution when performing the estimation of the expected return with stochastic updates, we need the distribution of the transitions visited and the distribution of the transition sampled for the updates to be the same. Hence in order to compensate the bias introduced by prioritized sampling, weighted Importance Sampling (IS) is used in the updates in the form of

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \tag{2.32}$$

where N is the replay buffer size and β is an hyperparameter that we anneal during the training towards one, since at $\beta = 1$ the IS completely compensates for the bias introduction, and the influence of this bias is much stronger in the late stages of training. Prioritized experience replay can boost the data efficiency of a DRL algorithm considerably, and in the original paper prioritized DQN outperformed normal DQN in 41 out of 49 games. Furthermore, more recently an ablation study on Rainbow [14] – a state-of-the-art algorithm that combines many different optimizations – has proven prioritized experience replay to be the most influential of the optimizations proposed.

2.2.3 Categorical RL

Categorical RL [3, 15, 38] is a framework that augments the signal learned by the value functions. While classic DRL algorithms learn the expected return, categorical RL learns the distribution of the random return received by the agent. This brings different advantages, since it preserves multimodality in value distributions and alleviates the effects of learning from nonstationary policies. Furthermore if the value functions return a distribution rather than a single value, each probability of the distribution will naturally work as an auxiliary prediction. To put it in simpler terms, we are basically giving our algorithm the ability to keep multiple options in mind when predicting the return. This ultimately stabilizes the learning and increases the performance of the algorithm. We can write the distributional Bellman equation as

$$Z(s,a) \overset{\mathrm{D}}{=} R(s,a) + \gamma Z(S', A') \tag{2.33}$$

where Z is the distribution whose expectation is the Q-value, the capital letters are used to indicate random variables, and $\overset{\mathrm{D}}{=}$ means that the two distribution are equivalent. To represent the value distribution Z, a discrete distribution parameterized by the number of supports is proposed in [3]. More specifically, we define a set of atoms

$$\{z_i = V_{min} + i\Delta z : 0 \le i \le N\},\ \Delta z = \frac{V_{MAX} - V_{MIN}}{N-1} \tag{2.34}$$

where each atom is a potential return, and the probability of each atom is given by a parametric model θ such that

$$Z(s,a;\theta) = z_i \quad with\, probability \quad p(s,a) = \frac{e^{\theta_i(s,a)}}{\sum_j e^{\theta_j(s,a)}} \tag{2.35}$$

Using a discrete distribution like this has different advantages, since it is easy to compute, highly expressive and allows us to use cross-entropy to calculate the difference between two distributions. In order to use cross-entropy though, we need to ensure that both our target distribution and our current distribution have the same set of supports. Hence, we project the target distribution obtained by applying Equation 2.33 onto the supports of our current distribution, and only after that we calculate the difference between the two distributions. We refer the reader interested in the exact nature of the projection function or on further theoretical details about the categorical RL framework to the references. Categorical RL was originally applied to DQN, but was later expanded to DDPG-style actor-critic methods in [5] by defining the policy gradient as

$$\nabla J_\phi = \mathbb{E}\left[\nabla_\phi \pi(s;\phi)\, \mathbb{E}[\nabla_a Z(s,a;\theta)]\right] \tag{2.36}$$

C51 [3] Rainbow [14] and D4PG [5] are a few examples of algorithms that where able to yield state-of-the-art results using categorical RL.

2.2.4 Hindsight experience replay

Hindsight Experience Replay (HER) [17] is a promising technology that allows to augment the reward signal and data efficiency of the algorithm. As briefly mentioned in Section 1.1, the inspiration comes from the ability of humans to learn even from our failures. Imagine a simple setting in which we need to throw a ball at a target. If the target gets hit one point is given, otherwise zero. If an algorithm manages to throw the ball, but misses the target by some margin to the right, it will just infer that the sequence of actions it just performed led to a failure. This makes the algorithm learn nothing from this experience, whereas a human could have easily inferred that if the target would have been more to the right, the throw would have been a success. To try to achieve this kind of reasoning, HER augments the reward signal introducing goals. Taking inspiration form the Universal Value Function Approximators (UVFA) framework [39], HER trains policies and value functions which take as input not only a state but also a goal. To define such goals, it is assumed that there is a predicate $f_g : S \to \{0, 1\}$ that maps states into 0 or 1, and the goal of the agent becomes to reach a state such s such that $f_g(s) = 1$. This slight modifications allow to train universal policies by giving a reward of -1 when $f_g(s) = 0$ and 0 otherwise. This method results to not be very efficient in reality due to the sparsity and little information given by the reward function. To solve this problem, HER implements a simple strategy to make better use of off-policy learning. The agent will start acting in the environment, adding each transition to the experience buffer like usual. In addition to that though, every time an episode finishes, we sample a set of additional goals for every transition of the episode and add all the variants to the experience buffer as well. Multiple strategies to sample the new goals are suggested, but the best performing one consist in deriving the new goal from a random state reached by the agent in that episode, but in a transition successive to the one we are currently considering. HER is is able to learn complex behaviors even in sparse and binary rewards, where normally techniques like DDPG would fail. Furthermore it has much space for further experimentation, since it is orthogonal to other techniques like prioritized experience replay, distributional RL and entropy-regularized RL, which makes interesting testing its effects in conjunction with such techniques [40].

2.2.5 Distributed DRL

We have already mentioned more than once that one of the more common trends in DRL is to try to make it as similar as possible to deep learning. This is because we currently have a much better understanding of how to optimize and stabilize deep learning than we have with DRL. One of the main causes of success of deep learning is that it is possible to highly parallelize the computation, exploiting not only the advantages of modern graphic cards to the fullest, but also the computational power that distributed systems with many processors can guarantee. Usually the main way to take advantage of distributed systems for training neural networks is to parallelize the computation of gradients, so that the network parameters can be optimized more quickly. Although some successes in applying distributed asynchronous parameter updates where achieved also in DRL [41, 42], most of the research has concentrated on asynchronous parameter updates and parallel data generation within a single machine [19, 43, 44, 45]. This is partially due also to the necessity of ensuring low latency communication, which is essential in a context like DRL, where gradients becomes quickly outdated and the data collected depends heavily on the network parameters.

The ape-x framework [4] takes a different approach, distributing data collection instead of gradient calculation. To do this, it decomposes the standard DRL computation in two components, an actor and a learner. This is a common decomposition in distributional DRL, although the exact nature of this components slightly changes between different methods [4, 41, 44]. To avoid confusion with the actor of actor-critic methods, which has no connection to the actor of the ape-x framework, from now on we will refer to the latter as executor. In ape-x, the executor is a process with its own copy of the environment. The executor takes a step in the environment using the policy network to choose the best action to take, and then stores the transition in a shared replay buffer. The learner retrieves batches of data from the replay buffer and updates the network parameters. Another interesting feature of this framework, is that it was built to extend prioritized experience replay. To allow it to be highly scalable, the way priorities for new transitions are initialized is modified. In vanilla prioritized experience replay, priorities are initialized to the highest priority seen so far, and updated only when they are sampled for the first time. Due to the amount of experience that ape-x allows to collect, this strategy does not scale well as it would introduce a bias towards the most recent data collected. The executors though already calculate the loss while evaluating their local copies of the policy. Using it to calculate online estimates of the priorities allows to obtain more realistic estimates without extra computation. The ape-x framework is agnostic to the DRL algorithm being used, as long as it is off-policy. Using half of the wall-clock time, 376 cores and 1 GPU, ape-x applied to DQN improves the median results over ALE of more than five times, doubles the results of a state-of-the-art algorithm like Rainbow, and collects a number of transitions that is two order of magnitude higher than all previous methods.

Chapter 3

Design

This chapter describes in detail the design of our HDRL algorithm. Since this research expands the SAC-LSP algorithm with four different techniques, we will first explain in further details how our baseline works, and then explain how the optimizations we have chosen have been applied to it.

3.1 Baseline

Our baseline is an off-policy, model-free, maximum entropy, actor-critic, HDRL algorithm. SAC adapts the maximum entropy framework to a DDPG-style actor-critic. With SAC-LSP the authors are interested in building hierarchies of policies, which means that using Gaussian Mixture Models to represent stochastic policies like SAC does would be limiting. As a matter of fact, while Gaussian Mixture Models are sufficiently expressive in medium-dimensional action spaces, as a sub-policy they can only provide a limited number of behaviors for the higher layers, namely the number of mixture elements. This limits the expressiveness of higher layers of the hierarchy, potentially preventing them from solving the task.

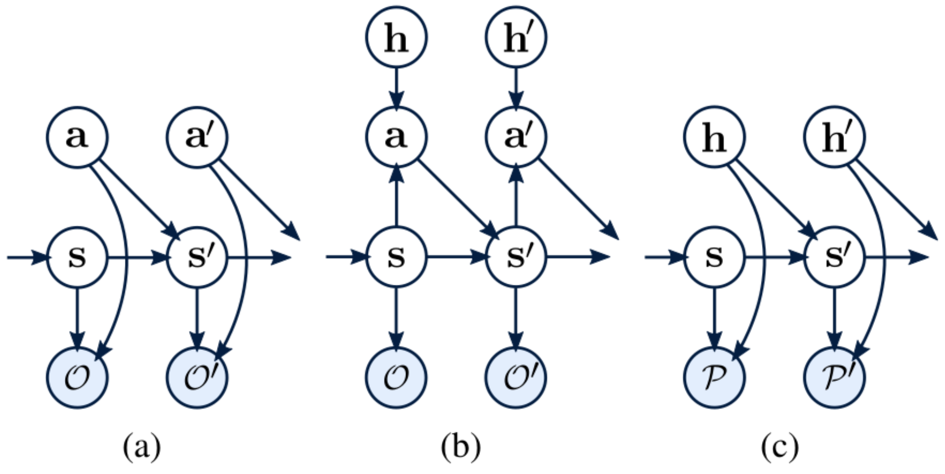

Figure 3.1: Probabilistic graphical model representing how to grow hierarchies of policies. Sourced from [2]

To obtain a framework able to grow hierarchies in bottom-up layer-wise fashion, SAC-LSP casts the optimal control problem of learning a policy that optimizes the future reward, to an inference problem, as can be seen in Figure 3.1a. Each state and action is conditioned on a binary optimality variable $\mathcal{O}$, which says whether that state-action couple was optimal or not with respect to a certain reward function. Incorporating the reward function by setting $p(\mathcal{O}|s, a) = exp(r(s, a))$ allows to re-derive the maximum entropy objective of Equation 2.14, giving at the same time a probabilistic graphical model on which we can easily define a hierarchy. A stochastic base policy is defined as a latent variable model composed by two factors, namely a conditional action distribution $\pi(a|s, h)$ with h latent random variable, and a prior $p(h)$. In this setting, to sample an action we first need to sample h from the prior and then sample an action conditioned on h, bringing us to the graphical model of Figure 3.1b. Finally, by marginalizing out the actions and conditioning the model on a new optimality variable $\mathcal{P}$ – which can represent either the same task or a different higher level task – we obtain the graphical model of Figure 3.1c, that is structurally identical to the one of Figure 3.1a with the exception of h taking the role of the actions a. This new model incorporates the base policy into the transition model of the MDP, exposing at the same time a higher level set of actions h. This allows to train a new, higher level policy by conditioning it on our new optimality variable $\mathcal{P}$. Furthermore, this process can be repeated arbitrarily, integrating at each new hierarchy layer the lower level policy into the MDP's transition dynamics, and learning a new higher level policy that uses the latent space of the lower level policy as action space.

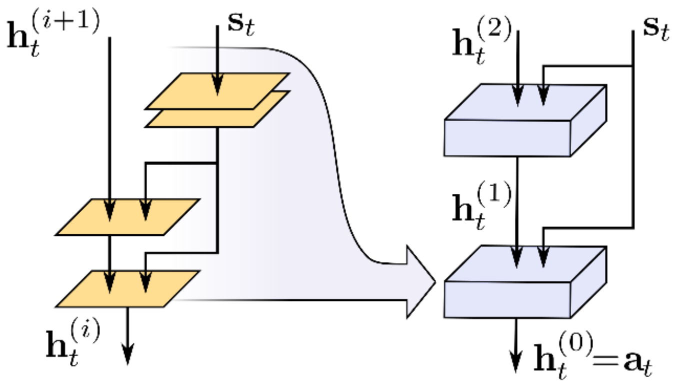

Figure 3.2: The hierarchy used in SAC-LSP. Sourced from [2]

The only thing left is to choose the policy representation, which should be tractable in order to be able to maximize its log-likelihoood, expressive enough to not reduce the freedom of higher levels of the hierarchy, and deterministic since the higher levels will view each conditional factor as part of the environment. To ensure these three characteristics, SAC-LSP represents policies as bijective transformations from latent variables to actions. Such transformations are represented with real-valued non-volume preserving (real-NVP) neural networks [46], an unsupervised learning techniques that allows to learn bijective transformations in a computationally effective method. The specifics of real-NVPs are out of the scope of this research, hence we refer the interested reader to the references. In Figure 3.2 we can finally see the actual hierarchy proposed in SAC-LSP. The hierarchy is composed of two levels, although as we said we can build an arbitrary number of layers if desired. Each level takes in input the current observation and a latent vector from the level above, and produces in output a latent vector, which in the case of the lowest layer is equivalent to the action chosen.

3.2 Prioritized SAC-LSP

Prioritized experience replay is one of the easiest techniques to adapt to new algorithms. This technique increases the data efficiency by modifying the sampling strategy of the experience buffer. Hence, it can be easily applied to most off-policy algorithms, as long as they use an experience buffer. We follow the design proposed in the original paper [16], opting to define the priority of a trainsition in its proportional form (Equation 2.30). Prioritized experience replay has three main operations to take into account: insertion of new experience, sampling of new experience and priority update. In order to be able to execute such operations efficiently, we represent our experience buffer as a sum-tree. We still introduce a bit of latency, but the performance increase strongly justifies it. Finally, we multiply the TD error of the Q-function updates utilizing the weighted IS as calculated in Equation 2.32.

3.3 Categorical SAC-LSP

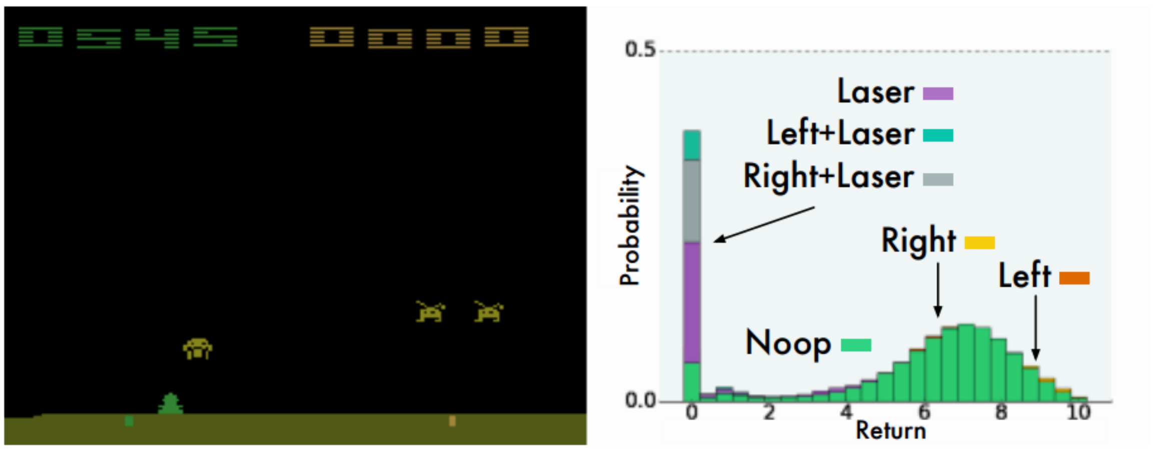

Figure 3.3: Learned value distribution during a game of Space Invaders. In the graph, different colors represent different actions. Sourced from [3]

We want to extend SAC-LSP to the categorical RL framework. Augmenting the signals received by our policies, value functions or reward functions is a common trend in DRL research, since it helps to boost the performance of our algorithm even in the hardest sparse-reward environment. Our baseline falls in the maximum entropy framework, which as we saw in Section 2.2 augments the optimal policies with a policy entropy signal that we aim to maximize, yielding better exploration and performance. The categorical RL framework instead augments the return of the value functions, which instead of returning the expected future reward, return the value distribution. In Section 2.2 we have already shown the modifications that each of those frameworks brings, offering for both frameworks an overview based on their application to a DDPG-style actor critic, in order to highlight the similarities between the methods and make the discussion more readable. Nonetheless, when a new DRL framework is created, loss functions and gradient updates must be re-derived from scratch in order to obtain a method that is both tractable and proven to converge to an optimal solution. Unfortunately this is a long process, which exceeds the time constraints and scope of this research. Therefore, we opt for a more naive approach, substituting the value functions in the SAC-LSP updates with their distributional versions. More precisely, we modify Equation 2.25 so that the new Bellman error is

$$J_Q(\theta) = \mathbb{E}\left[\frac{1}{2}(Z(s,a\,;\,\theta) - (r(s,a) + \gamma\,\mathbb{E}[\,V_d(s\,;\,\psi)]))^2\right] \tag{3.1}$$

where Z is the distributional Q-value function and V_d is the distributional value function. Moreover we modify Equation 2.23 to be

$$J_V(\psi) = \mathbb{E}\left[\frac{1}{2}(V_d(s\,;\,\psi) - \mathbb{E}[\,(Z(s,a\,;\,\theta) - log\,\pi(a|s\,;\,\phi)\,])^2\right] \tag{3.2}$$

3.4 HER SAC-LSP

To extend SAC-LSP with hindsight experience replay, we need to modify the reward to be goal-based. Following the UVFA framework, we assume there is a predicate $f_g : S \to \{0,1\}$ which is well defined for every state $s \in S$. This is not limiting since rewards are always based on reaching a state in which certain conditions are satisfied. We therefore need to define a mapping $m : S \to G$ such that $f_{m(s)}(s) = 1$. If we want our agent to simply reach a certain state, we can define m as an identity mapping, since in this case the predicate simplifies to $f_g(s) = [s == g]$. Since the state in a Mujoco environment is a list of data relative to the simulated robot like its position, speed and the angle of its joints, reaching a specific state might result to hard. We therefore define m as a function that returns the position of the agent, and the goal to be a position that the agent must reach. Moreover, we also experiment with m as a function that returns the speed of the agent, in which case the goal is to reach that speed. Similarly to [17] we test multiple possible rewards. The first one is the standard goal utilized by HER, in which $r_g(s, a) = -1$ ifthe goal is reached, and 0 otherwise. This reward makes sense for environments with binary and sparse reward, which are the ones at which HER is mainly aimed at. As already explained in Section 2.2.4, the underlying idea of HER is to instill a hindsight thinking mechanism into an RL agent by sampling extra sub-goals based on future states for each transition. Often though, the reward signal of an environment can contain useful information that we do not want to loose. We therefore also test two further alternatives in which instead of substituting the standard environment reward with our goal-based one, we augment the environment reward with the goal-based reward. The first augmentation tactic is the sum of the two rewards, i.e.

$$r(s, a) = r_{env}(s, a) + r_g(s, a) \tag{3.3}$$

If the scale of the environment reward and the goal-based reward do not match or the environment reward is very dense, the final reward will likely be instable or dominated by one of the two. Thus, we also test a proportional reward, more formally

$$r(s, a) = r_{env}(s, a) + (r_g(s, a) * 0.1 * r_{env}(s, a)) \tag{3.4}$$

Finally, we modify our value function, Q-value function and policy to take also the goal as input.

3.5 Distributed SAC-LSP

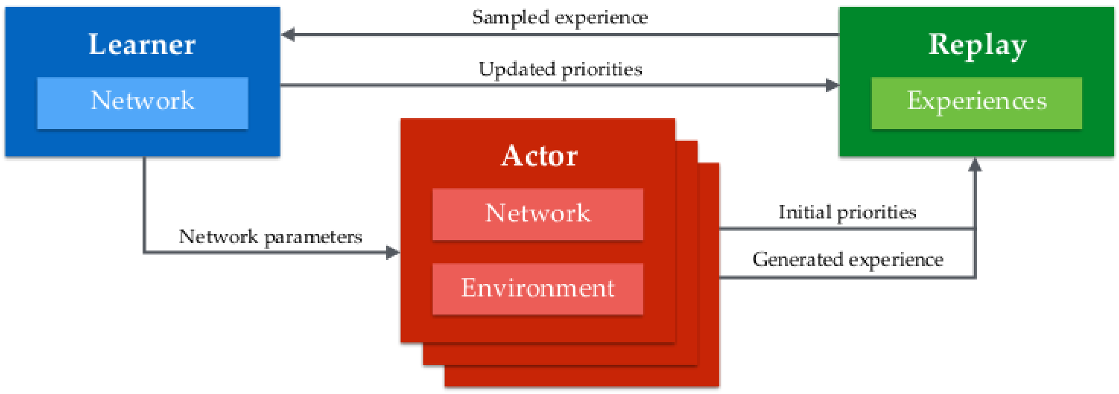

Figure 3.4: The ape-x framework. Multiple executors (actors) generate experience, and a central learner executes gradient descent and updates the network parameters. Sourced from [4]

Our baseline has many good qualities like stability, good exploration and the possibility to build hierarchies of policies in bottom-up fashion without the need of hand-designed task-specific features. Nonetheless, it runs on a single processor, which limits its scalability. We want want therefore to expand it to a distributed framework, so that it is able to make better use not only of distributed systems, but also of classic multi-core computers. We choose to use the ape-x framework [4] for multiple reasons. First, this method yielded results that completely outperformed previous state-of-the-art algorithms, showing that this method can scale up well even with distributed systems composed of hundreds of machines. Furthermore, since it distributes the experience collection, it brings enhanced data efficiency – a key factor for every successful DRL method – with minimum computational overhead. This is a feature in which we are particularly interested, since we aim to test this method on a single multi-core machine. Finally, this method is designed to take further advantage of prioritized experience replay, and the improvements that yield combining this two methods have already been documented [4]. This allows us to build this extension directly on top of Prioritized SAC-LSP. The classic learning loop of an off-policy DRL algorithm consist of an agent taking an action in the environment, adding the experience just gained to the experience buffer, then sampling a mini-batch of experience and using it calculate the gradients and update the network parameters. The ape-x framework decouples this process into two independent components, the executor and the learner. Figure 3.4 represents the separation of concerns introduced by the ape-x framework. Each executor is a process with an independent copy of the environment and access to the current policy. Using the policy, the executor takes actions in the environment and adds the experience gathered in a shared experience buffer. Multiple executors asynchronously act in their own environment, all using the same policy and adding the experience to the same experience buffer. The learner is instead a single process, which takes batches of experience from the shared buffer and uses it to update the policy and value functions networks. All the processes in the original ape-x framework run asynchronously, which ensures that no single process will act as bottleneck of the framework. While this is advantageous in a distributed system, in a single multi-core machine this can significantly slow down the learning process. This is due to the fact that the many executor processes tends to overcome the learner, which ends up being executed rarely. To fix this, we introduce a synchronization mechanism that ensures that each executor can take no more than one step per each learner step. This significantly reduces the latency introduced by the ape-x framework, although also reducing the data collection. To further boost the exploration, originally every executor should have a different exploration policy. We find this to not be strictly needed in our case, thanks to the stochastic nature of SAC-LSP policies. Finally, as aforementioned in Section 2.2.5, we take advantage of the errors calculated by each executor when evaluating its policy to insert experience in the buffer with a good estimate of its priority, without requiring extra computation. This ensures once again that the agent is able to scale well also to very high numbers of executors.

Chapter 4

Implementation

In this chapter we enter into the details of how our baseline and its extensions have been implemented. All our programs are written in Python 3.5 and all our models are developed in Tensorflow [47, 48], an open source software library to develop machine learning applications. Tensorflow uses multidimensional data arrays called tensors to store variables, and data flow graphs to represent computations. In these graphs each node is a mathematical operation, and the edges represent the path that the tensors will follow during the computation. This architecture makes Tensorflow a powerful and flexible tool, able to efficiently run computation on CPU, GPU and distributed systems. Furthermore, it is currently the most used library for DRL applications, therefore using it allows us to take inspiration from the multiple collections of open source implementations of DRL algorithms available [49, 50, 51]. We use the rllab [50] framework to develop and evaluate our DRL algorithms. This framework offers a set of continuous control tasks, interfaces to develop and extends DRL algorithms, and full compatibility with the other major DRL framework, OpenAI gym [52]. This allows us to have access to a wide range of both discrete and continuous tasks – many of which based on the ALE and the Mujoco environment – with the same interface.

4.1 SAC-LSP

The code of SAC-LSP was published online by the authors for reproducibility[1]. It is built in Python and Tensorflow using the rllab framework. Due to the reliability of the implementation and the extensibility that using rllab entails, our research uses such implementation as baseline. The networks that represent the value functions are implemented as a Multilayer Perceptron (MLP), feed-forward neural networks composed entirely of fully connected layers. MLPs are often considered less powerful than CNNs, mostly because their fully connected nature prevents them to scale to very large inputs. When a DRL algorithm has to learn directly from pixels, like in the case of a task based on ALE, CNNs are imperative to represent the policy and the value functions. Since the observations in a Mujoco environment are composed of data relative to the agent's locomotion, though, we can use a MLP, which also results much faster to train. Both the value function network and the Q-value function network are composed of two layers of 128 neurons each, plus a final layer with just one neuron. A ReLU non-linearity is applied to the hidden layers, whereas a tanh non-linearity is applied to the output layer. In the case of the Q-value function, which takes both an observation and an action as input, the corresponding tensors are concatenated before being fed to the first layer of the network. As illustrated in Figure 3.2, the observation embedding on which the policy is conditioned is composed of an MLP with 2 fully connected layers of 128 neurons. The output layer has size equal to the degrees of freedom of the simulated robot, e.g. 16 for the the ant model. The policy itself is represented with a real-NVP bijector composed of two coupling layers. The Adam optimization algorithm is used to execute the network parameters updates, with a learning rate of 3e-4. When calculating the TD error, we utilize a discount factor of 0.99, and the soft network updates (recall Equation 2.12) have τ = 1e-2.

4.2 Prioritized SAC

The standard replay-buffer is composed of five numpy arrays of length 1M, one for each element of the tuple (*state*, *action*, *reward*, *next state*, *terminal*). Therefore, retrieving experience i is as simple as creating the tuple ($state[i]$, $action[i]$, $reward[i]$,

[1]https://github.com/haarnoja/sac

next observation[i], *terminal*[i]). The arrays are used like ring-buffers, which means that once the array is full, each new transition will overwrite the oldest one. Implementing prioritized experience replay with this data structure, although theoretically possible, would entail a lot of latency. This is because for each batch sampled we would need to search for the highest priorities in the buffer. Since we are using the proportional priority defined in Equation 2.30, we can take advantage of a data structure similar to a binary heap, the sum-tree. In a sum-tree, the internal nodes store the sum of its children. We store the transition priorities in the leaf nodes. In this way, the root node contains the sum of all the priorities, which must be always accessible to calculate the probability to sample a transition (Equation 2.29). To sample a batch of k transitions, we divide the range $[0, p_{total}]$ – where p_{total} is the sum of all the priorities – into k equal sub-ranges. We then uniformly sample one priority from each sub-range, and using the index of those priorities we can retrieve the corresponding experiences from the actual experience buffer. In this way we do not change the buffer storage tactic (which is still based on numpy arrays that act as ring buffers), and by using a sum-tree to hold the priorities and linking them to the corresponding stored transitions, we can obtain $O(logN)$ complexity for updates and sampling, and $O(1)$ for insertion. We will use an annealing parameter of $\beta = 0.4$, which will be annealed uniformly towards one at each new epoch. Finally, we use a prioritization exponent of $\alpha = 0.5$.

4.3 Categorical SAC-LSP

Scalar Value
Q
Categorical
ω

Figure 4.1: Left, output layer of the standard value functions. Right, output layer of the categorical value functions. Sourced from [5]

Categorical RL's value functions must return parametrized distributions. Since the value functions are represented by MLPs, we modify the output layer to have a number of outputs equal to the number of atoms of our parametrized distribution. Moreover, we change the activation function of the output layer to be a softmax non-linearity. The softmax function assigns decimal probabilities to each class in a multi-class problem, therefore using it as activation function of our last layer outputs the probabilities of each discrete support of our value function Z. Finally, we make sure to project the target value functions on the supports of the current value functions, and modify the gradient updates as shown in Equation 3.2 and Equation 3.1. Following the successful results of [3, 5], we implement a distribution with 51 atoms. To set sensible values for V_{MIN} and V_{MAX}, we first observe what values the value function outputs during the training, and consequently chose $V_{MIN} = -100$ and $V_{MAX} = 5000.$, which are the two extremes observed.

4.4 HER SAC-LSP

To extend SAC-LSP to utilize HER, we need to modify our value function, Q-value function and policy to take a goal as additional input. Our neural network already supports multiple inputs as explained in Section 4.1, therefore we just need to concatenate the goals with the observations (and actions where appropriate) before feeding the input to the first layer of the networks. To introduce the hindsight thinking mechanism, we need to be able to sample new goals from future states. In order to do that, we create an episode buffer – a list populated with (*observation*, *action*, *reward*, *next observation*, *terminal*, *goal*) tuples throughout each episode, and cleared at its end. This allows us, at the end of each episode, to re-visit each transition and sample new sub-goals from a random state the agent will reach in the future. As mentioned in Section 3.4, the observations in Mujoco are basically a list of information relative to the position and speed for each of the agent's joints. After inspecting the Mujoco environment class, it is easy to derive the position or speed of the model from it. We can then define a new goal as a random position or speed (using the same format used in the Mujoco environment model) and sample new goals from future states easily. To ensure that the goals that we create at the beginning of each episode assume reasonable results, we first inspect the values that such parameters assume when simulating a previously trained policy, and then sample only values inside the range of values observed. We can now define the distance between the current state and our goal as the L2 norm of the two vectors, i.e., the Euclidean distance between the two positions or the two speeds. A goal is then considered to be reached if the L2 norm is lower than a certain threshold. We sample $k = 4$ new goals for each transition, and use a threshold of 3.0 when using a speed-based goal and a threshold of 10.0 when using a position-based goal.

4.5 Distributed SAC-LSP

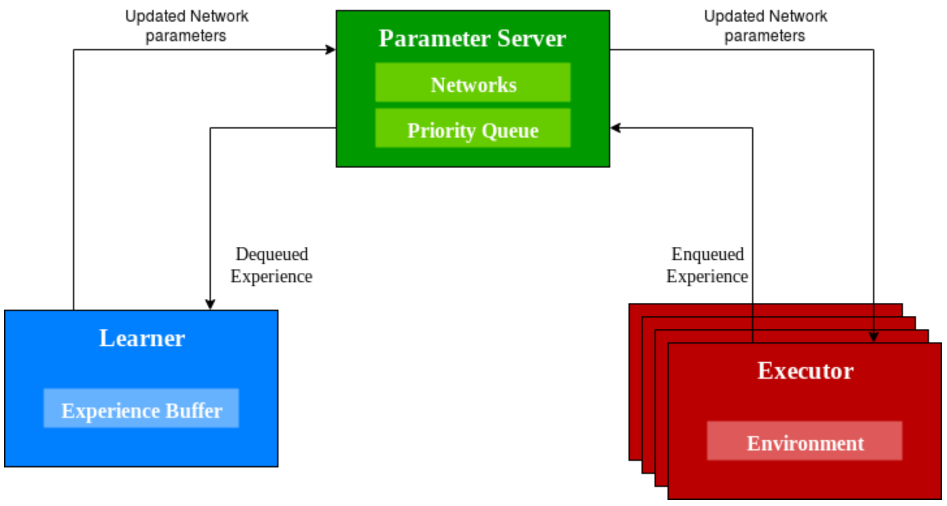

Figure 4.2: Ape-x framework's implementation in Tensorflow.

The ape-x framework is composed of many executor processes and one learner process. No communication is needed between executors, whereas each executor must send the experience collected to the shared experience buffer, and the learner must communicate the parameters update to each executor. Figure 3.4 gives a good idea of the main components of the framework and the main communications between such components. Nonetheless, the actual implementation of the framework takes a slightly different form, as illustrated in Figure 4.2. In Tensorflow, a "cluster" is a set of "tasks" that have access to the same execution graph. Each task is a different process, and tasks can be grouped by "jobs". Our framework has two jobs, the worker and the parameter server. The parameter server stores and updates the network parameters, furthermore all processes communications pass through it. The workers are of two types, executor and learner. Each executor has its own copy of the environment, and adds the experience it collects to a priority queue held by the parameter server. The queue has fixed size, and an add operation to a full queue will block the executor until there is enough space to execute the insertion. The learner holds the experience buffer, which it populates by retrieving minibatches from the priority queue. Since the learner is responsible for computing the parameter updates, we want it to be able to run as frequently as possible. Hence, since the dequeue operation is blocking when the priority queue is empty, we always check if the queue has enough elements before trying to retrieve data from it. Workers and learner do not need to program specific operations for updating their local copies of the networks, as all network operations will automatically be synchronized with the parameter server under the rug. The "global step" is a shared variable that gets incremented every time there is a network update performed by the learner. Using that variable, we are able to synchronize the executors so that they do no perform more than one environment step per each network update. We utilize a total of 10 processes: one parameter server, one learner, and eight executors.

Chapter 5

Evaluation

In this chapter, we present the experiments run to evaluate the four extensions we tried on SAC-LSP. We first present some general information regarding the training of the algorithm and the tasks that it has to solve. Then, we show the results of each extension in comparison to our baseline, and present an interpretation explaining the reasons behind the successes and the failures.

5.1 Training specifications

We run all our experiments on a single machine equipped with an Intel i7-7700K CPU at 4.20GHz, 16 GB of DDRAM and a GPU nVidia GTX 1080. As explained in Section 4.1, we only utilize small networks in our algorithm. We tested running the baseline on both CPU and GPU, and due to the small sizes of our networks, running all computation on CPU resulted faster. Therefore, all the experiments presented in this chapter have been run on the CPU unless explicitly stated otherwise. To compare the performances of the different extensions, all of them will have to learn the same task. Specifically, we choose the Mujoco ant environment. The ant agent is composed of 4 legs attached to the main torso, and each leg is composed by two joints.

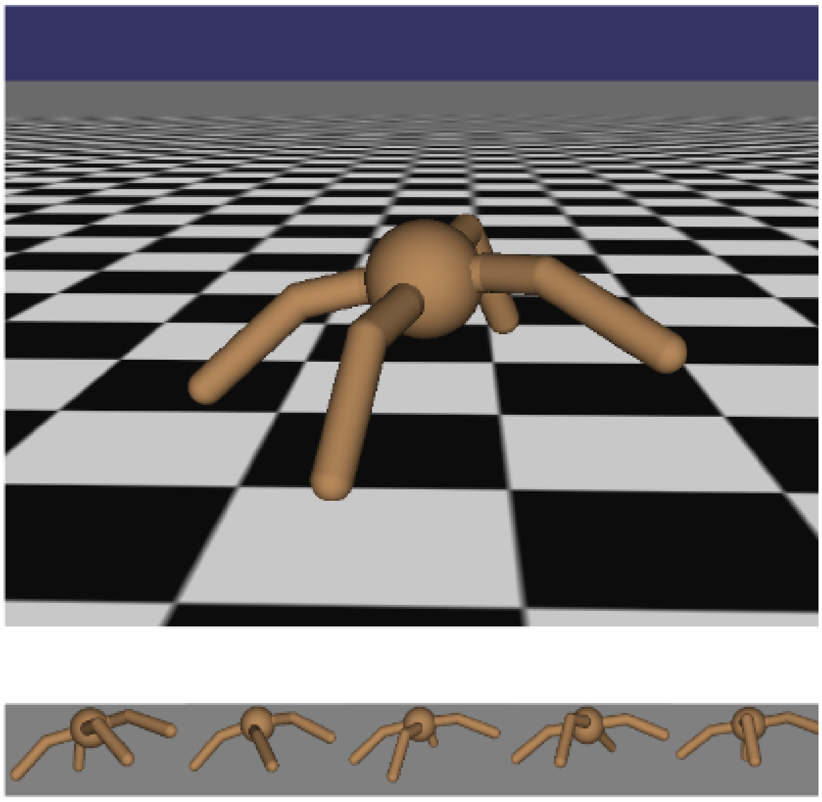

Figure 5.1: Top: The ant agent in its environment. Bottom: Frames of the ant performing locomotion. Sourced from [6]

The goal of the task is to learn locomotion, therefore rewards are given proportionally to how fast the ant will run and how far it will reach. This environment has been chosen because it offers a good trade-off between the task difficulty and acceptable training times for a single machine with our hardware. As a matter of fact, while the environment is challenging due to the complexity of the simulated robot – which has 8 joints and 16 degrees of freedom –, it is still possible with our configuration to train vanilla SAC-LSP for 10,000 epochs in roughly 12 hours. Each epoch is composed of 1,000 learning steps. During a learning step, the algorithm takes an action in the environment, adds the current transition to the experience buffer, samples a batch of transitions from it and executes the networks updates. We sample batches of 128 transitions and update the target networks at each learning step. This does not influence the stability of the program since we perform soft network updates (Equation 2.12).

After each epoch, we evaluate our agent by running one episode with the current policy. Furthermore, we save our model on disk each 1,000 epochs. These models can later be used both for further evaluation and to train the second layer of the hierarchy. Following the example of [2], we train the second layer of the hierarchy on a policy trained for 6,000 epochs, and train it for further 4,000 epochs. This allows us to see the improvement that the two layer hierarchy yields over the single layer hierarchy, while maintaining the total number of epochs at 10,000.

5.2 Prioritized SAC-LSP

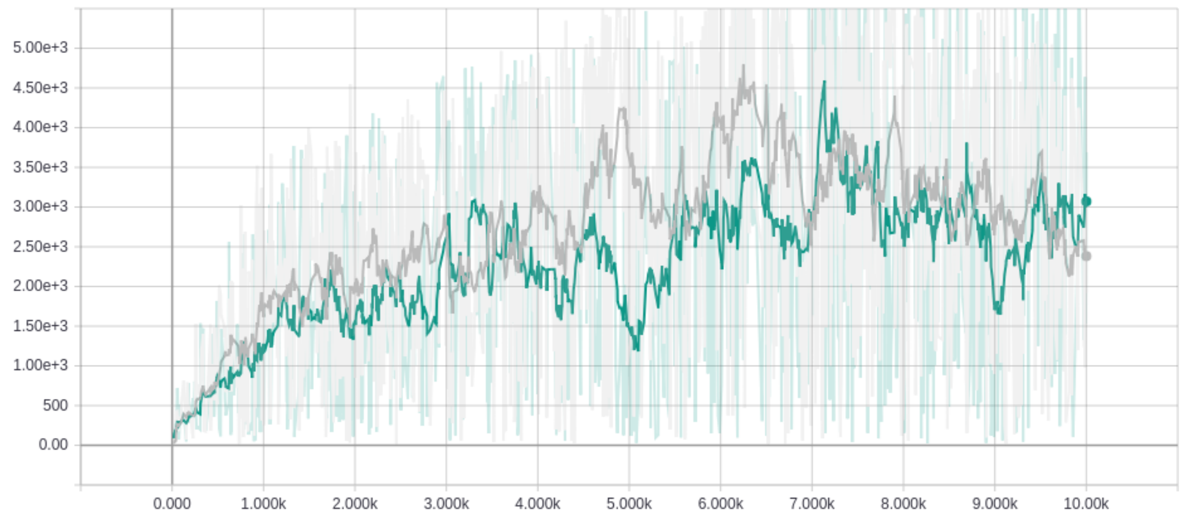

Figure 5.2: Graph of the total return of an episode (y axis) against the number of epochs passed (x axis). The baseline is in gray, while prioritized SAC-LSP is in green.

Figure 5.2 plots the online evaluation produced during learning. This shows the improvement in performance of the algorithm with respect to the amount of training performed. The online evaluation of the algorithm is subject to extremely strong oscillations, as can bee seen in the light-colored lines of the graph, which plot the real learning curve of the algorithm. This is due to the stochastic nature of our policies, and the fact that at each evaluation step we only run one episode. To have a more precise evaluation of the learning curve, we should average the cumulative return of multiple episodes at each evaluation step. However, this would introduce further latency in our algorithm, and due to the time-constraints of this research we prefer to avoid that. Instead, we use tensorboard's inbuilt smoothing utility to have more readable learning curves, and the results are the full-colored lines of Figure 5.2. We can notice how prioritized SAC-LSP is comparable to the baseline, which tells us that the optimization is able to effectively learn the task and that further evaluation will probably yield similar results. Due to the promising results of the single-layer prioritized SAC-LSP shown in Figure 5.2, we also train train a second layer of the policy.

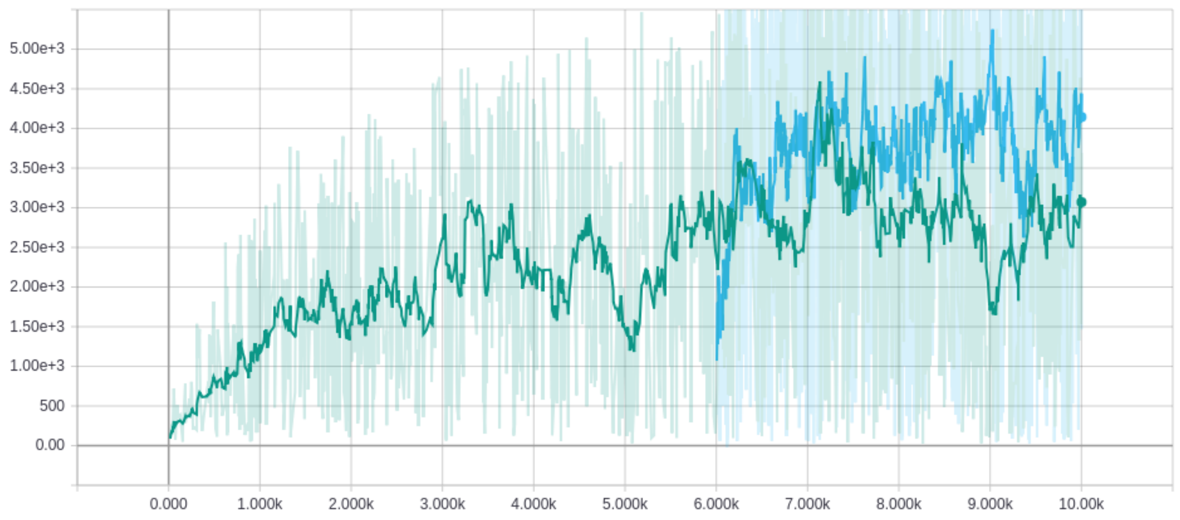

Figure 5.3: Graph of the total return of an episode (y axis) against the number of epochs passed (x axis). Single layer prioritized SAC-LSP is in green, while the second layer of the hierarchy is in light blue.

In Figure 5.3 we can see that training the second level of the hierarchy yields moderate improvements. The light blue curve starts at 6,00 epochs since it uses the policy trained by prioritized SAC-LSP as lower level policy. This helps to visualize how much a simple two-level hierarchy – in which the lower-level policy is trained for 6,000 epochs and the higher-level policy is trained for 4,000 epochs – improves the performances of a single-level hierarchy trained for 10,000 epochs. In other words, the improvements that we achieve by training the two-level hierarchy comes at no extra time compared to training a single-level hierarchy for the same total amount of epochs. To achieve a more accurate estimation of the performance of our algorithm, we evaluate the final models for 200 episodes and average their cumulative rewards. Table 5.1 shows the results of these tests. Prioritized SAC-LSP with a single layer is able to improve the performance of the corresponding baseline by 35%, and the 2-level hierarchy boosts the performances even further.

Table 5.1: Averaged cumulative reward of SAC-LSP with single and 2-layer policies, and the respective prioritized variants. Below, the relative improvement in comparison to the baseline

	SAC-LSP	Prioritized SAC-LSP	2L SAC-LSP	2L Prioritized SAC-LSP
Avg. reward	3157.40	4278.88	4255.05	4821.40
Rel. impr.	1	1.36	1.35	1.53

5.3 Categorical SAC-LSP

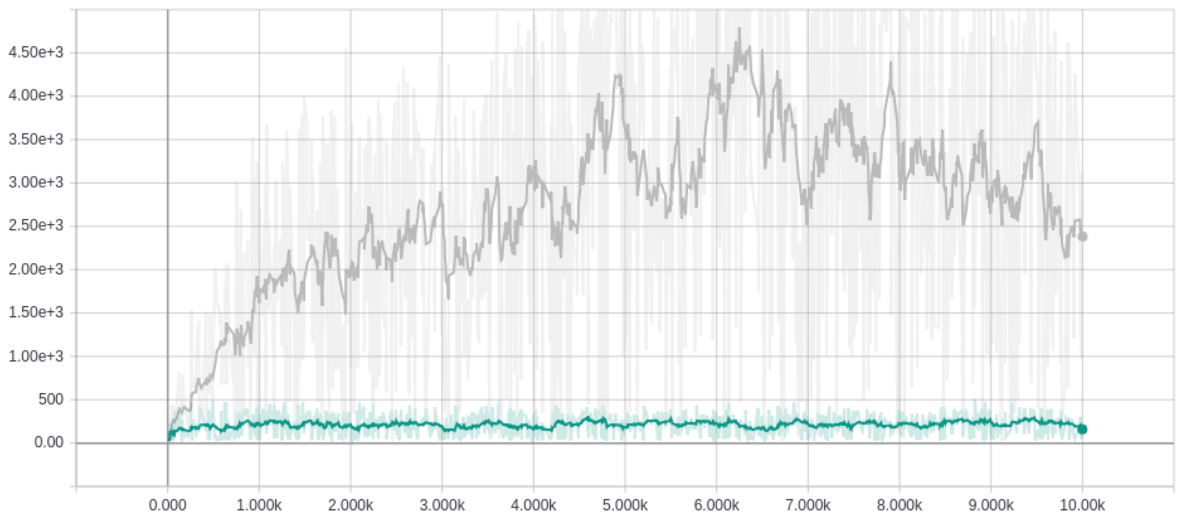

Figure 5.4: Graph of the total return of an episode (y axis) against the number of epochs passed (x axis). The baseline is in gray, while categorical SAC-LSP is in green.

Categorical SAC-LSP, as clear from Figure 5.4, fails to learn the task. Throughout the training, the algorithm's performance always remains in the range (0, 500). This is the same range that a randomly initialized policy would achieve, and we can notice from the graph that the performance remains stable throughout the 10,000 epochs of learning. It is clear that our naive implementation of categorical SAC-LSP is not able to solve the task. To try to understand why, we take a look at the values that the Q-value function assumes during training.

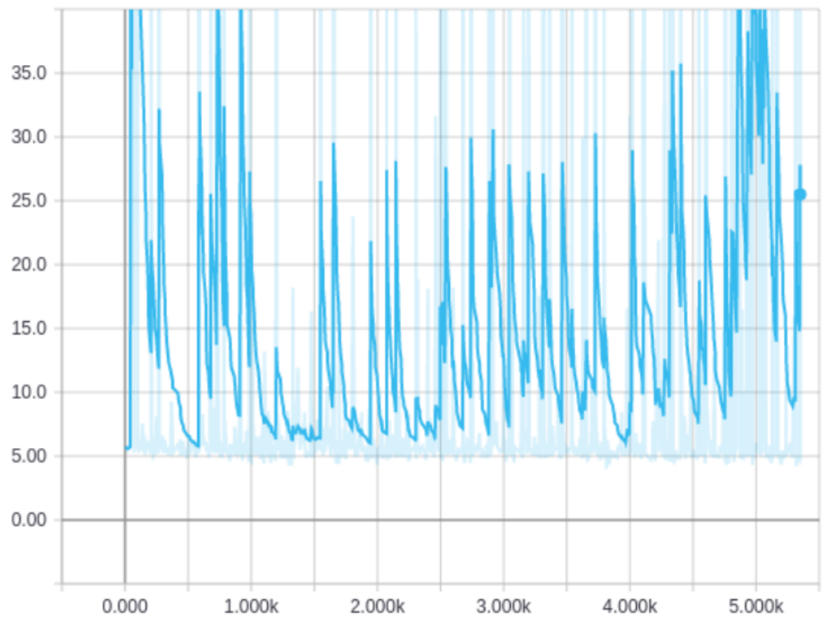

Figure 5.5: Standard deviation of Q-value function against the number of epochs of categorical SAC-LSP.

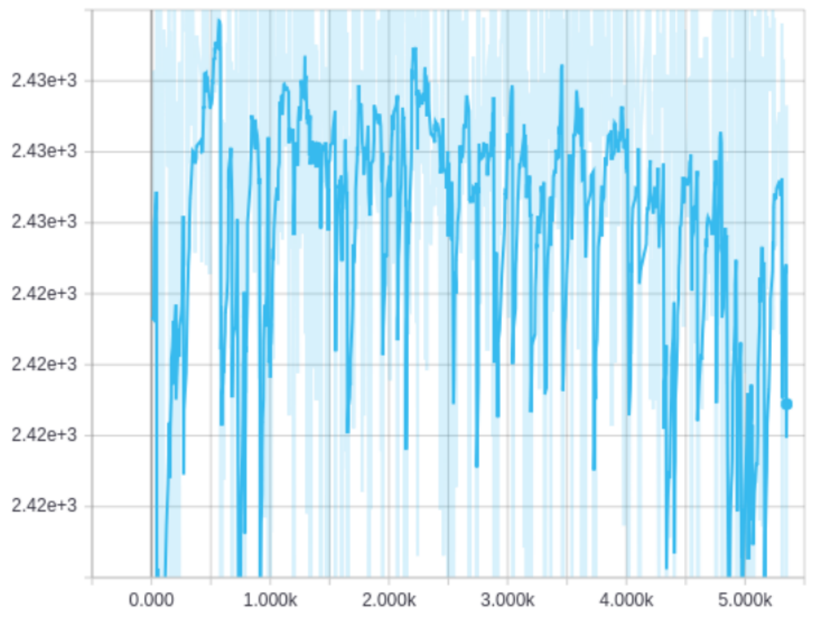

Figure 5.6: Average of Q-value function against the number of epochs of categorical SAC-LSP.

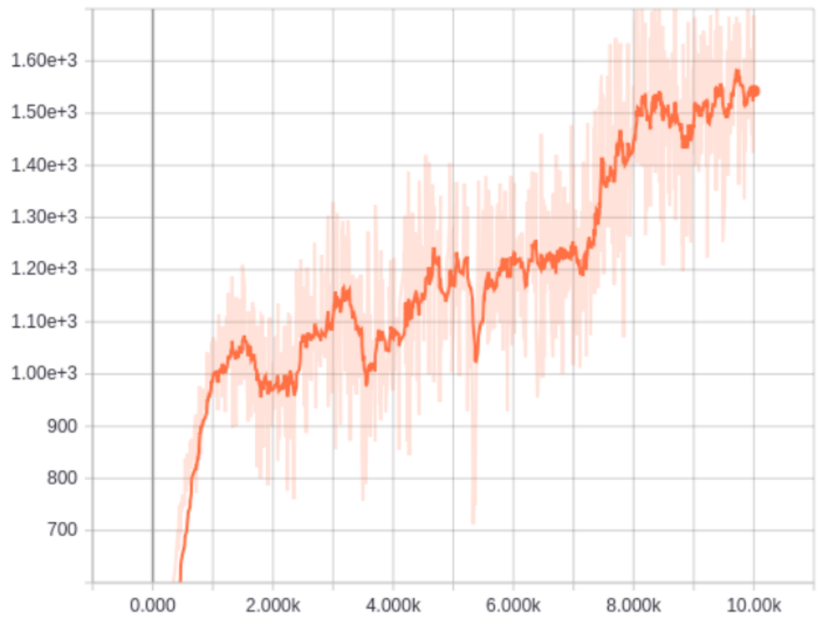

Figure 5.7: Standard deviation of Q-value function against the number of epochs of categorical SAC-LSP.

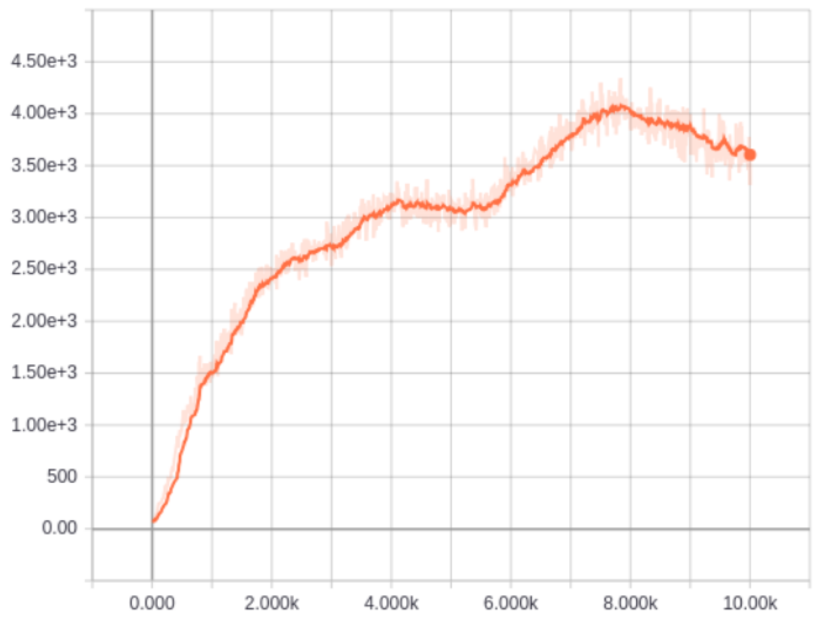

Figure 5.8: Average of Q-value function against the number of epochs of SAC-LSP.

Normally, both the standard deviation and the average of the Q-value function have a stable growth, assuming values in the same order of magnitude, as visible in Figure 5.7 and Figure 5.8. This is expectable, since the Q-value function outputs the expected return from taking a specific action and the following the current policy from the next state onwards. This means that while the policy improves, the Q-value function will predict higher and higher values, since following a better policy entails obtaining a higher cumulative reward. Nevertheless, with categorical SAC-LSP such values suffer extreme fluctuations, and differ by up to three orders of magnitude as can be seen in Figure 5.5 and Figure 5.6. If the algorithm was simply not able to learn because of the complexity of the problem, the predicted expected returns would have remained in a range similar to that of the average cumulative rewards. Instead, the values that the Q-value function returns tells us that the loss formulated in Equation 3.1 is wrong, i.e., the Q-value function updates do not converge towards the optimal Q-value function. Although we present the results of the Q-value function, inspecting the value function yields the same results. It is important to clarify that this results do not preclude maximum entropy frameworks and categorical frameworks to be effectively combined. Deriving the soft bellman updates from scratch, taking into consideration that we are modeling categorical value functions and that we can express their difference in terms of KL divergence, is a potential approach to solve the problems present in our design.

5.4 HER SAC-LSP

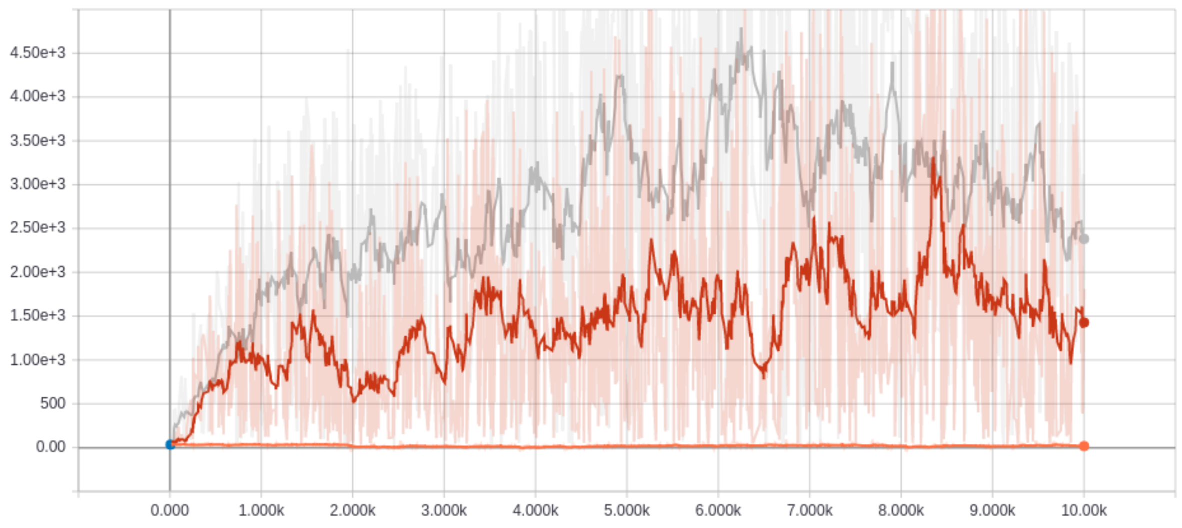

Figure 5.9: Graph of the total return of an episode (y axis) against the number of epochs passed (x axis). The baseline is in gray, binary speed-based HER SAC-LSP is in orange, and proportional speed-based HER SAC-LSP is in red. For readability, only the learning curves of speed-based goals models are plotted. The learning curves of position-based goals models are comparable to their speed-based counterparts, and their performance is reported in Table 5.2

HER is the technique for which we experiment the most. As a matter of fact we try two types of goal. The first is a position-based goal, which means that the goal of the agent is to reach a specific position. The second goal is speed-based, which means that the goal of the agent is to move at a certain speed. Furthermore for each goal type we test a binary reward as expected by the UVFA framework, a sum-augmented reward as defined in Equation 3.3 and a proportionally-augmented reward as defined in Equation 3.4. Although HER modifies the reward signal of the task, during the evaluation we use the standard environment reward in order to be able to compare the performances of HER SAC-LSP with our baseline.

With both goal types, we find the algorithm to be unable to learn locomotion when using a binary reward. Simulating the policy learned using the position-based reward, we notice that at each episode the agent simply takes one step and then stalls. From this behavior we can infer that the combination of the complexity of the task and the sparsity of the reward introduce a bias towards the first states that the agent visits in the environment. Due to the complexity of the task, and considering that an episode terminates if the agent falls, the first episodes last a very small amount of time. Moreover due to the sparsity of the reward, learning how to take the first steps is even harder, which means the amount of short episodes we will experience will be higher. If to these two factors we add the hindsight thinking mechanism, which re-samples extra goals from future states, this causes the algorithm to receive rewards signals almost only from the earliest states, thus introducing a bias. In the case of the goal-based reward then, the algorithm will learn to reach an very close position and stay there, as already mentioned. In the case of the speed-based reward, it will receive rewards signals almost only from speeds close to zero, so the agent will not learn to maximize speed and ultimately learns to just stand still. It is interesting to note how these results are very different from those of categorical SAC-LSP. As a matter of fact, while categorical SAC-LSP has a wrong formulation which prevents the algorithm from learning an effective policy, in this case the algorithm is able to actually learn but gets stuck on a local minimum at the earliest stages. It is possible that further fine-tuning of the threshold hyper-parameter and the goal generation strategy might help to solve the problem, but we argue that this partially defies one of the main objectives of this method, which is to be able to learn in difficult environments without the need to to engineer complicated task-specific reward functions.

The rewards we define augmenting the environment reward are able to learn locomotion to a certain degree, but yield considerably lower performances compared to vanilla SAC-LSP. We argue that this is because the reward augmentation introduced is not expressive enough to justify the increase in complexity that having to use the goal implies. As a matter of fact, utilizing a goal increases the dimension of the input of all the networks, but still gives very sparse rewards. In an environment like Mujoco ant, where the standard reward is already dense and expressive, the addition of the goal based framework increases the complexity of the task without bringing enough extra information to justify it. In conclusion then, we see that HER only deteriorates the performance of our algorithm in the environment tested. Nonetheless, this does not preclude HER SAC-LSP to yield better results in a different environment, especially if it is characterized by binary and sparse reward. The averaged cumulative return of an episode for each of the variations proposed can be seen in Table 5.2

Table 5.2: Averaged cumulative reward of HER SAC-LSP variants

HER	Binary	Sum-augmented	Proportionally-augmented
Position-based	25.05	1020.32	1345.33
Speed-based	32.85	1146.52	1595.57

5.5 Distributed SAC-LSP

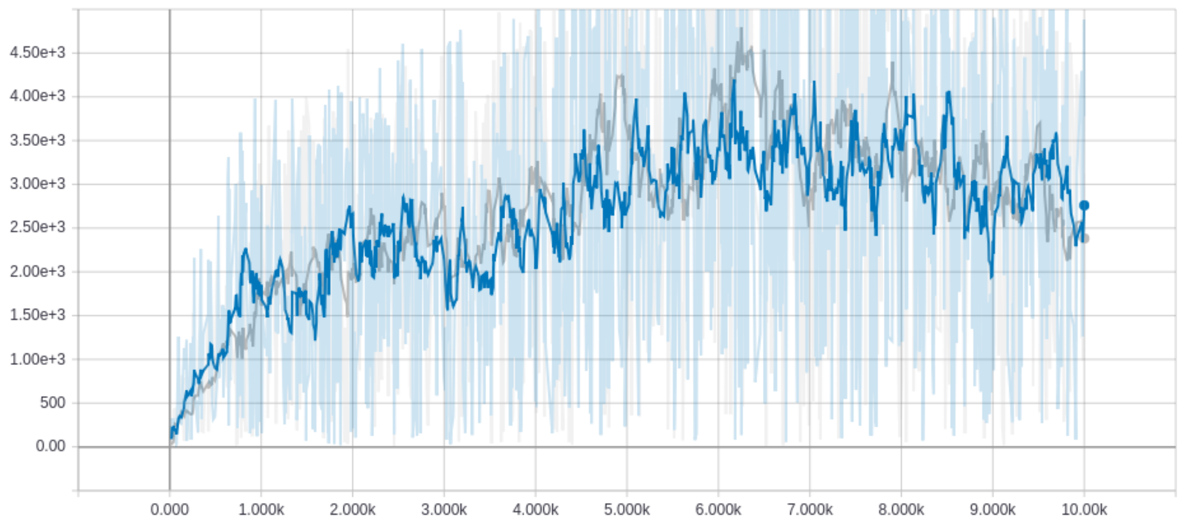

Figure 5.10: Graph of the total return of an episode (y axis) against the number of epochs passed (x axis). The baseline is in gray, while distributed SAC-LSP is in blue.

The last optimization we test is distributed SAC-LSP. We can see from Figure 5.6 that its learning curve is very close to that of the baseline, which tells us that the algorithm is able to learn and achieve results comparable to that of the baseline. We therefore choose to test distributed SAC-LSP on our standard 2-level hierarchical policy, as already done in Section 5.2.

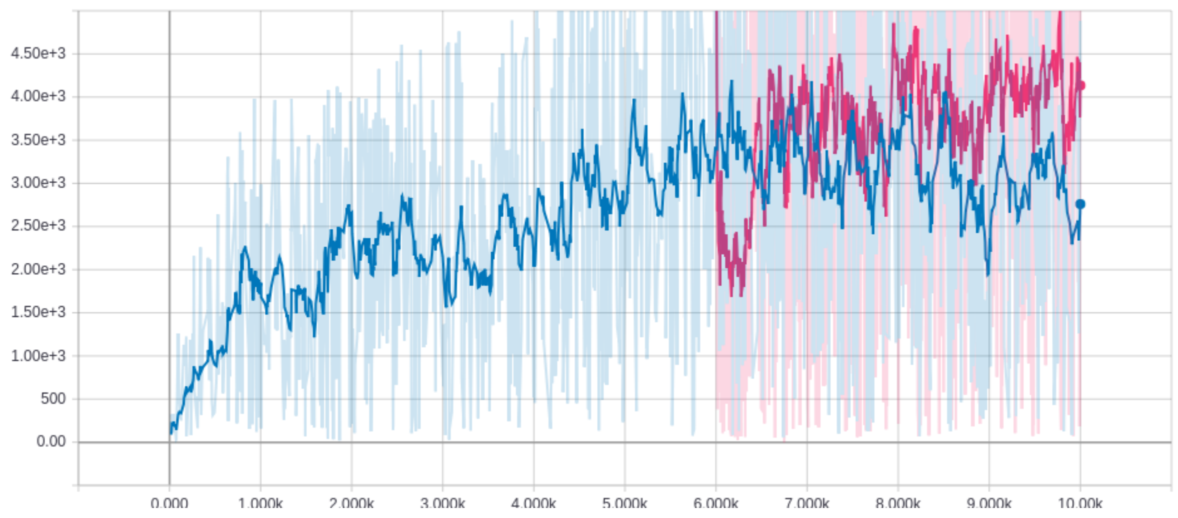

Figure 5.11: Graph of the total return of an episode (y axis) against the number of epochs passed (x axis). Single-layer distributed SAC-LSP is in blue, while the 2-layer hierarchy distributed SAC-LSP is in purple.

The results of training a higher-level policy starting from the 6,000th epoch can be seen in Figure 5.11. Once again, the hierarchy introduction helps to further boost the performances of the algorithm. As mentioned before, due to the type of online evaluation performed, the learning curve plotted above is mainly an indication of whether the algorithm is able to learn and a rough estimate of how well. To evaluate the actual performance of the algorithm then, we simulate the trained policies and average the cumulative reward of 200 episodes.

Table 5.3: Averaged cumulative reward of SAC-LSP with single and 2-layers policies, and the respective distributed variants. Below, the relative improvement in comparison to the baseline

	SAC-LSP	Distributed SAC-LSP	2L SAC-LSP	2L Distributed SAC-LSP
Avg. reward	3157.4	4425.76	4255.0	5173.40
Rel. improv.	1	1.40	1.35	1.64

Distributed SAC-LSP is able to further boost the results obtained by prioritized SAC-LSP, albeit just slightly. This comes at the cost of an increased latency of the training, which takes roughly three times the wall-clock time required to train the baseline. Nonetheless, this is still a very good result especially if we consider that instead of running the algorithm asynchronously with hundreds of executors on a distributed system, we are running 8 executors synchronously on a single machine. This means that the simple collection of 8 times the normal amount of experience yields an improvement between 3% and 7% respect to prioritized SAC-LSP.

5.6 Comparison

We evaluated 4 different extensions against the baseline. Of those four extensions, one failed to learn the desired task (categorical SAC-LSP), one yielded poor results (HER SAC-LSP) and two where able to improve the results over the baseline. Our research has to cope with two main limitations. The first one is the hardware, which being limited to a single machine prevents us from training more than one model at a time, and in the case of distributed SAC-LSP introduces a considerable latency. The second limitation comes from the time-constraints of this research. Researching machine learning requires time for experimentation and parameter tuning. Although the latter is slightly less important in DRL since models are expected to be able to generalize to multiple tasks without the need of fine-tuning, a certain level of parameter research is still needed. All the models, baseline included, suffered from extreme oscillations during the training phase. This is unexpected, since SAC-LSP is presented as a "surprisingly stable" ([2]) method. Furthermore, the performance of a model can be strongly influenced by the seed used to randomly initialize the network parameters. In [2], the results reported are the mean of the performance of the model trained with 5 different random seeds. Performing such thorough analysis was infeasible for this research, therefore we only train each model once. Furthermore, to avoid parameter research, for each hyperparameter we either use the most common value in the literature, or we use simple heuristics to choose sensible values. An example of the latter method is how we chose the the values of V_{MIN} and V_{MAX} in Section 4.3. In Table 5.4, we summarize the performance of each model we trained.

Table 5.4: List of all the trained models performance and relative improvement in comparison to the baseline. In the HER SAC-LSP entries, the first letter indicates if the model uses a position-based goal (P.) or a speed-based goal (S.), whereas the second letter indicates if the model used a binary (B.), sum-augmented (S.) or proportionally-augmented (P.) reward.

	Average reward	Rel. improvement
SAC-LSP	3157.4	1
2L SAC-LSP	4255.05	1.35
Prioritized SAC-LSP	4278.88	1.36
2L Prioritized SAC-LSP	4821.40	1.53
Categorical SAC-LSP	237.02	0.08
P.B. HER SAC-LSP	25.05	0.008
P.S. HER SAC-LSP	1020.32	0.32
P.P. HER SAC-LSP	1345.33	0.42
S.B. HER SAC-LSP	32.85	0.01
S.S. HER SAC-LSP	1146.52	0.36
S.P. HER SAC-LSP	1595.57	0.51
Distributed SAC-LSP	4425.76	1.40
2L Distributed SAC-LSP	5173.40	1.64

Our final model is distributed SAC-LSP, which combines prioritized experience replay with a distributed experience collection mechanism. The single-layer distributed SAC-LSP model outperforms the single-layer baseline by 40%, while two-layer distributed SAC-LSP outperforms the two-layer baseline by 22%.

Chapter 6

Conclusion

This chapter concludes the dissertation by summarizing the main contributions presented in this research. We then discuss possible future works, outlining what could be done to continue the research and further improve the model we created.

6.1 Main Contributions

In this research we studied the importance of deep reinforcement learning, explaining the advantages that studying HDRL can bring to the field. We then chose a flexible and powerful HDRL algorithm, which can create hierarchies of policies in bottom-up layer-wise fashion, composed of an arbitrary number of layers. We use SAC-LSP, the HDRL algorithm just mentioned, as a baseline for four different extensions. The extension we tested are all orthogonal to each other, allowing to potentially combine them all in a unique algorithm. Furthermore, to our knowledge such techniques have had little to no application to HDRL algorithms so far. Our main contribution is distributed SAC-LSP, an algorithm that extends the baseline with prioritized experience replay and distributed experience collection. This algorithm is obtained by applying a slightly modified version of the ape-x framework, where the number of executors running is very limited, there is a synchronization mechanism that avoids the executors to take more than one environment step per learning step, and the algorithm runs on a single multi-core machine. Although our implementation is limited respect to the original ape-x framework, distributed SAC-LSP is able to outperform vanilla SAC-LSP by up to 40% in our tests. The other two optimization tested, namely categorical DRL and HER, do not make it into our final model since they fail to improve over the baseline's performance. Nonetheless, our analysis of the results leaves margin for further study of how to improve on those two optimizations, as we will discuss in further details in Section 6.2. In particular, our study of HER tried to apply it to an environment with a very different reward signal respect to the one this kind of technique was originally meant for. We prove that HER if applied to very complex tasks and with little fine-tuning can introduce a bias towards the first states visited that ultimately makes it get stuck on a local minimum. Furthermore, trying to expand HER to be a auxiliary reward mechanism, we propose two different reward augmentation tactics. HER SAC-LSP with environment reward augmentation are able to surpass the local minimum on which binary HER SAC-LSP got stuck, but ultimately even the best performing variation – namely HER SAC-LSP with speed-based goals and proportionally-augmented reward – yields results 49% lower to that of vanilla SAC-LSP.

6.2 Future Work

Distributed SAC-LSP is our main contribution, and although it is able to increase the performance of our baseline, most of the improvement is due to the use of prioritized experience replay as it is clear by comparing Table 5.1 with Table 5.3. As matter of fact the difference between prioritized SAC-LSP and distributed SAC-LSP is between 3% and 7%. We speculate that this is due to the modifications we applied to the ape-x framework, which effectively limit the full potentiality of the method. Although such modifications resulted necessary to train the algorithm on our configuration and within our time-constraints, they can easily be lifted when using a distributed system. Therefore, one of the easiest yet more interesting directions to pursue in the future would be to eliminate such limitation and test the full potentiality of distributed SAC-LSP on a distributed system.

In this research we tried a naive application of the categorical DRL framework to SAC-LSP, which ultimately led to a wrong formulation of the loss functions. Nevertheless, categorical DRL remains a very promising technique and trying to combine it with the maximum entropy framework on which SAC-LSP is based remains a very interesting problem. A possible approach to do so would be to follow the formulation of the maximum entropy framework as presented in [36, 37], but taking into account from the beginning that the value functions we want to learn are categorical and that the distance between two distributions can be defined as the cross-entropy or the KL divergence between the distributions.

Although HER is the technique with which we experiment the most, it still leaves margin for further study. Firstly, it would be interesting to test HER SAC-LSP on more suited environments, like the robot-arm tasks used in [17] and the maze navigation task used in [2]. Furthermore, HER is particularly interesting if applied to HDRL methods, as proven in HAC[18], a recent HDRL algorithm that applies the hindsight thinking mechanism of HER not only to goals, but also to actions generated by a higher-level policy. Applying a similar idea to SAC-LSP is certainly a promising direction to pursue in the future.

Finally, it would be interesting to test the performance of both our baseline and distributed SAC-LSP on an environment like ALE. Atari environments in fact are substantially different from Mujoco environments, since their action space is discrete and their observation space are the game frames. Moreover, games like Montezuma's Revenge are still a challenge to DRL algorithms, due to the sparsity of the reward and the amount of exploration that the algorithm must perform. As mentioned in Section 1.1, HDRL algorithms are expected to improve the performances in such environments, and both stochastic policies and distributed DRL improve the action-state exploration. This makes distributed SAC-LSP a particularly interesting algorithm to test on this kind of environments.

Bibliography

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

[2] T. Haarnoja, K. Hartikainen, P. Abbeel, and S. Levine, “Latent space policies for hierarchical reinforcement learning,” *arXiv preprint arXiv:1804.02808*, 2018.

[3] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos, “Distributional reinforcement learning with quantile regression,” *CoRR*, vol. abs/1710.10044, 2017.

[4] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. Van Hasselt, and D. Silver, “Distributed prioritized experience replay,” *arXiv preprint arXiv:1803.00933*, 2018.

[5] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, A. Muldal, N. Heess, and T. Lillicrap, “Distributed distributional deterministic policy gradients,” *arXiv preprint arXiv:1804.08617*, 2018.

[6] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” *CoRR*, vol. abs/1506.02438, 2015.

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

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

[9] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[10] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture.," in *AAAI*, pp. 1726–1734, 2017.

[11] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine, "Diversity is all you need: Learning skills without a reward function," *CoRR*, vol. abs/1802.06070, 2018.

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

[13] C. Florensa, Y. Duan, and P. Abbeel, "Stochastic neural networks for hierarchical reinforcement learning," *CoRR*, vol. abs/1704.03012, 2017.

[14] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *arXiv preprint arXiv:1710.02298*, 2017.

[15] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," *CoRR*, vol. abs/1707.06887, 2017.

[16] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[17] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba, "Hindsight experience replay," in *Advances in Neural Information Processing Systems*, pp. 5048–5058, 2017.

[18] A. Levy, R. P. Jr., and K. Saenko, "Hierarchical actor-critic," *CoRR*, vol. abs/1712.00948, 2017.

[19] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *International conference on machine learning*, pp. 1928–1937, 2016.

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

[21] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, Oct 2012.

[22] R. Bellman, “The theory of dynamic programming,” tech. rep., RAND Corp Santa Monica CA, 1954.

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

[24] M. Riedmiller, “Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method,” in *European Conference on Machine Learning*, pp. 317–328, Springer, 2005.

[25] G. Tesauro, “Temporal difference learning and td-gammon,” *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[26] J. N. Tsitsiklis and B. Van Roy, “Analysis of temporal-diffference learning with function approximation,” in *Advances in neural information processing systems*, pp. 1075–1081, 1997.

[27] L.-J. Lin, “Reinforcement learning for robots using neural networks,” tech. rep., Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[28] H. Van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning.,” in *AAAI*, vol. 2, p. 5, Phoenix, AZ, 2016.

[29] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, “Dueling network architectures for deep reinforcement learning,” *arXiv preprint arXiv:1511.06581*, 2015.

[30] R. A. Howard, "Dynamic programming," *Management Science*, vol. 12, no. 5, pp. 317–348, 1966.

[31] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, pp. 1057–1063, 2000.

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

[33] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[34] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

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

[36] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine, "Reinforcement learning with deep energy-based policies," *arXiv preprint arXiv:1702.08165*, 2017.

[37] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[38] M. Rowland, M. G. Bellemare, W. Dabney, R. Munos, and Y. W. Teh, "An analysis of categorical distributional reinforcement learning," *arXiv preprint arXiv:1802.08163*, 2018.

[39] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators," in *International Conference on Machine Learning*, pp. 1312–1320, 2015.

[40] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba,

"Multi-goal reinforcement learning: Challenging robotics environments and request for research," *CoRR*, vol. abs/1802.09464, 2018.

[41] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, *et al.*, "Massively parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1507.04296*, 2015.

[42] N. Heess, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, A. Eslami, M. Riedmiller, *et al.*, "Emergence of locomotion behaviours in rich environments," *arXiv preprint arXiv:1707.02286*, 2017.

[43] A. V. Clemente, H. N. Castejón, and A. Chandra, "Efficient parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1705.04862*, 2017.

[44] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "Ga3c: Gpu-based a3c for deep reinforcement learning," *CoRR abs/1611.06256*, 2016.

[45] K. Shirahata, Y. Coppens, T. Fukagai, Y. Tomita, and A. Ike, "Gunreal: Gpu-accelerated unsupervised reinforcement and auxiliary learning," *International Journal of Networking and Computing*, vol. 8, no. 2, pp. 408–423, 2018.

[46] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real NVP," *CoRR*, vol. abs/1605.08803, 2016.

[47] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: a system for large-scale machine learning.," in *OSDI*, vol. 16, pp. 265–283, 2016.

[48] S. S. Girija, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016.

[49] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Openai baselines." https://github.com/openai/baselines, 2017.

[50] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," *CoRR*, vol. abs/1604.06778, 2016.

[51] M. Lapan, *Deep Reinforcement Learning Hands-On: Apply Modern RL Methods, with Deep Q-Networks, Value Iteration, Policy Gradients, TRPO, AlphaGo Zero and More.* Packt Publishing, Limited, 2018.

[52] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

Appendix

The original code of SAC-LSP can be found at this link:
https://github.com/haarnoja/sac

The implementation of our research, can be found at the following link:
https://github.com/Stefa-no/Extending-SAC-LSP

Acronyms

AGI	Artificial General Intelligence.
AI	Artificial Intelligence.
ALE	Atari Learning Environment.
CNN	Convolutional Neural Network.
DDPG	Deep Deterministic Policy Gradients.
DPG	Deterministic Policy Gradient.
DRL	Deep Reinforcement Learning.
HDRL	Hierarchical Deep Reinforcement Learning.
HER	Hindsight Experience Replay.
i.i.d.	independent and identically distributed.
IS	Importance Sampling.
MDP	Markov Decision Process.
MLP	Multilayer Perceptron.
POMDP	Partially Observable Markov Decision Process.
real-NVP	real-valued non-volume preserving.
RL	Reinforcement Learning.
SAC	Soft Actor-Critic.
SAC-LSP	Soft Actor-Critic with Latent Space Policies.
TD	Temporal Difference.
UVFA	Universal Value Function Approximators.