UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Master Degree in Physics of Data

Final Dissertation

Deep Reinforcement Learning methods for StarCraft II Learning Environment

Thesis supervisor

Prof. Alberto Testolin

Thesis co-supervisor

Prof. Alexander Ilin

Thesis advisor

MSc. Rinu Boney

Candidate

Nicola Dainese

Academic Year 2019/2020

Contents

Abstract

Reinforcement Learning (RL) is a Machine Learning framework in which an agent learns to solve a task by trial-and-error interaction with the surrounding environment. The recent adoption of artificial neural networks in this field pushed forward the boundaries of the tasks that Reinforcement Learning algorithms are able to solve, but also introduced great challenges in terms of algorithmic stability and sample efficiency.

Game environments are often used as proxies for real environments to test new algorithms, since they provide tasks that are typically challenging for humans and let the RL agents make experience much faster and at a cheaper price than if they were to make it in the real world.

In this thesis state-of-the-art Deep Reinforcement Learning methods are presented and applied to solve four mini-games of the StarCraft II learning environment. StarCraft II is a real-time strategy game with large action and state space, which requires learning complex long-term strategies in order to be solved; StarCraft mini-games are auxiliary tasks of increasing difficulty that test an agent's ability to learn different dynamics of the game.

A first algorithm, the Advantage Actor-Critic (A2C), is studied in depth in the CartPole environment and in a simple setting of the StarCraft environment, then is trained on four out of seven StarCraft mini-games. A second algorithm, the Importance Weighted Actor-Learner Architecture (IMPALA), is introduced and trained on the same mini-games, resulting approximately 16 times faster than the A2C and achieving far better scores on the two hardest mini-games, lower score on one and equal score on the easiest one. Both agents were trained for 5 runs for each mini-game, making use of 20 CPU cores and a GPU and up to 72 hours of computation time.

The best scores from the 5 runs of IMPALA are compared with the results obtained by the DeepMind team author of the paper StarCraft II: A New Challenge for Reinforcement Learning (Vinyals et al. 2017), which report the best scores out of 100 runs that used approximately two orders of magnitude more of training steps than our runs. Our IMPALA agent surpasses the performance of the DeepMind agent in two out of the four mini-games considered and obtains slightly lower scores on the other two.

Preface

This thesis was done under the supervision of Prof. Alberto Testolin and I would like to thank him for his support and trust.

I wish to thank my co-supervisor Prof. Alexander Ilin for his invaluable teachings and insights.

I also would like to thank my advisor Rinu Boney for his precious help and useful feedback.

I would like to acknowledge the Helsinki Doctoral Education Network in Information and Communications Technology (HICT) for funding this thesis and CSC – IT Center for Science (Finland) for providing computational resources.

Finally I want to thank from the bottom of my heart my parents for their immense support throughout all my years of studies.

Padova, 16 October 2020

Nicola Dainese

1 Introduction to Reinforcement Learning

Reinforcement Learning (RL) is a field of Machine Learning concerned with learning how to act in an environment so as to solve a specific task. The acting entity - the agent - has to observe the surrounding environment and, based on its observation, decide what to do next in order to achieve its goal. A numerical feedback is given after every action, from which the agent can evaluate its own actions and learn how to perform better.

The term "reinforcement" comes from the work of Pavlov on conditioned reflexes, where he studied how a response to a stimulus could be reinforced in animals by being followed by a reward (reinforcer) or discouraged by being followed by a punishment. Since the reward and punishment are designed so as to cause pleasure and pain in the agent, its objective is always to maximize pleasure and minimize pain. To do that, it has to learn how to react to different stimuli, associating them with their long-term consequences. The reinforced behavior persists even after the reinforcer is withdrawn, thus it can be considered a form of learning from experience and association between stimuli.

The main elements that compose a RL problem are:

- States: description of the surrounding environment. It is important to notice that in Reinforcement Learning anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of its environment.
- Observations: what the agent perceives of the environment state. If state and observation do not coincide we say that the environment is partially observable.
- Actions: what the agent decides to do.
- Reward signal: numerical signal that completely defines the RL task. In general it will be a stochastic function of the current state and the agent's last action. The goal of the agent is to maximize the total reward received during the task.
- Policy: behavioral strategy of the agent, mapping the observed states of the environment to actions to be taken when in those states. It can be a deterministic or stochastic mapping.
- Model of the environment: implements the dynamics of the environment. In general, given a state and an action, predicts (or, in case of a stochastic dynamics, samples) the next state and the reward for the last action.
- Value function: defines how valuable a state is, i.e. what is the total amount of reward that an agent can expect to accumulate over the future, starting from that state.

There are some important remarks to point out:

1. In general a task's reward signal can be designed in many equivalent ways, but two incompatible reward signals will define two different tasks. For example if we want to teach an agent to play chess, our only objective would be for it to win every game, thus we can use a reward function that yields a reward of 0 for all game steps except the last, where it will give $+1$ for victory or -1 for loss. A final reward of $+10$ or -10 would obviously yield equivalent results, since it would just be a change in the measurement units of the reward (i.e. the agent's task is scale-invariant in the reward). However it's clear that for a program to learn which actions in the game were good or bad without any feedback during the middle-game can be extremely challenging, thus we could be tempted to design a more informative reward function, that assigns to each piece of the board a value (there are many scales of values used by professional chess players) and gives a reward of plus or minus the value of a piece if the agent respectively captures or loses that piece. This fixes a certain reward scale, say in the order between 1 and 10. Hence we would have to be careful to pick a sensible value for the

reward of winning and losing the game. How can we trade-off between the importance of capturing a piece, possibly gaining an advantage over the opponent, and the invaluable goal of making a checkmate? The problem with modifying the reward function is that the agent does not have an explicit understanding that the final objective is always to win the game and that the capture of the opponent's pieces is subordinate to that. This means that it will end up maximizing exactly the provided reward function, with unforeseeable consequences on the optimal strategy, e.g. it might avoid a checkmate to capture the opponent's queen. Hence engineering a reward function will in general greatly reduce the experience required to learn a task, but could also prevent the agent to adopt an optimal strategy (with respect to the ideal task) by inadvertently make it sub-optimal. This is why many researchers nowadays prefer the conservative approach of giving only the binary reward of +1 or -1 at the end of the game and a reward of zero throughout the rest of the game, which can be seen as "learning from scratch".

2. The agent's goal is to choose the action that will maximize the total reward accumulated during the entire game and not just the reward that will be received immediately after the action. This difference is crucial, since in general selecting a greedy action, which maximizes the immediate reward, might lead to a sub-optimal final outcome due to the shortsightedness of that action. For instance capturing an opponent's piece in chess might lead to a counter-move that leads the opponent to victory ("strategic sacrifice"), which of course is the opposite outcome than the one we are maximizing for. This is a key point to understand: the agent needs to seek states with high value and not with high immediate reward. But how can we predict how valuable a state is? And how do we act on that knowledge in order to translate that potential reward in actual reward? These are some of the central questions that RL addresses.

3. Rewards are primary, whereas values, as predictions of rewards, are secondary. Without rewards there could be no values and the only purpose of estimating the values is to achieve more reward. In other words values can be used as a proxy for the policy, by rephrasing the question "Which action is the best?" in "Which action will yield the most reward?", but it's by no means necessary to learn the value function in order to solve a RL task.

4. The value of a state does not exist on its own, because it depends on the future actions decided by the agent. Ultimately it's the expected reward *following* a certain policy, thus we will always talk about the value of a state accordingly to a certain policy.

5. Many RL problems could in theory be exactly solved by a brute-force search approach, however for any problem worth the name, brute-force search will be too inefficient for practical use. For instance it's estimated that the number of possible states in chess is of order 10^{46}, which is totally out of reach as order of computations required, especially since this kind of search should be performed for every action that needs to be selected during the game. Even more stunningly, in the board game of Go the number of possible states is around 10^{170} and currently only RL-based algorithms (e.g. Silver, Huang, et al. 2016) are capable of defeating human players in this game. RL algorithms are in fact designed to work efficiently even in presence of extremely high-dimensional search spaces at the price of finding only an approximate solution. Nonetheless cutting-edge problems in RL require huge computational power and massive hardware to be solved in an acceptable amount of time.

RL as a discipline received contributions from two separate threads: from behavioral psychology it took the ideas about the reward, the reinforcement of reactions to stimuli and about learning through interaction with the environment (Minsky 1961); from control theory it derived the theoretical framework - Markov Decision Processes (MDPs) - for its algorithms, the idea of the policy and its iterative improvement as well as the Bellman equations, recursive equations to estimate the future value of a state (Howard 1960). In the next section we will introduce Markov Decision Processes and review all the concepts that we introduced so far in a formal way.

2 Markov Decision Processes

Finite Markov Decision Processes (MDPs) (Bellman 1957, Howard 1960) provide a suitable theoretical framework to model the interactions between an agent and an environment.

The main assumptions of finite MDPs are:

1. There is a finite number of states and of actions.
2. Time is a discrete quantity. This assumption could be relaxed[1], but we will keep it for its simplicity throughout all this work.
3. The environment dynamics is Markovian, i.e. all transition probabilities are independent from the time variable, and completely determined by the current state (or state-action pair) of the system.
4. The interaction between agent and environment works as follows: the agent receives a state and a reward from the environment, decides an action to take and submits it back to the environment, receiving a new state and a reward for its last action and so on (see Figure 1).

We will denote the random variables for the state, action and reward at a certain time-step t respectively as S_t, A_t and R_t. Their sets of values, or spaces, will be $\mathcal{S}$, $\mathcal{A}(S_t)$ and $\mathcal{R} \subset \mathbb{R}$. Notice that in general the action space (the set of actions available) will depend on the current state, but many times we will assume that it will always be the same and write just $\mathcal{A}$, and that the reward is always a real number (scalar).

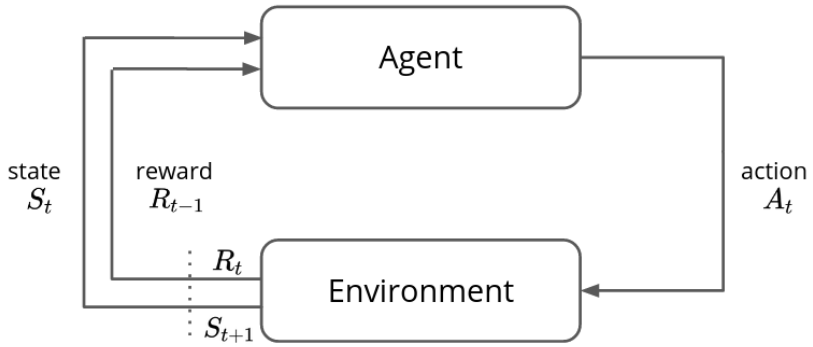


Figure 1: Agent-environment interaction: at each time step the agent receives the current state S_t and the reward for the last action R_{t-1}; based on S_t decides an action A_t that submits to the environment, which in turn will output the new state S_{t+1} and the reward R_t for the action A_t.

The interaction between the agent and the environment gives rise to a sequence of states, actions and rewards like the following:

$$S_0, A_0, R_0, S_1, A_1, R_1 \dots \tag{1}$$

This is also called a trajectory. In this work we adopt the convention in which the reward for action A_t taken in state S_t is R_t (and not R_{t+1}), since it's received as a consequence of action S_t.

Furthermore we will indicate the values assumed by the random variables with the same lowercase letter, i.e. s_t, a_t and r_t. Given a state s_t and an action a_t, the new state s_{t+1} and the reward r_t for the last action will be distributed accordingly to the joint distribution:

$$p(s_{t+1}, r_t | s_t, a_t) \equiv Pr(S_{t+1} = s_{t+1}, R_t = r_t | S_t = s_t, A_t = a_t) \tag{2}$$

[1] In fact in the early history of Control Theory are treated equivalently both the continuous and the discrete time cases, see for example Bellman and Kalaba 1965.

Sometimes we will use the symbols p_s and p_r for the marginal distributions of the next state or the reward. The marginal distributions are obtained as:

$$p_s(s_{t+1}|s_t, a_t) = \sum_{r \in \mathcal{R}} p(s_{t+1}, r|s_t, a_t) \tag{3}$$

$$p_r(r_t|s_t, a_t) = \sum_{s \in \mathcal{S}} p(s, r_t|s_t, a_t) \tag{4}$$

Finally we will use the symbol π for the agent's policy, which specifies the probability of choosing an action conditioned by the agent's state:

$$\pi(a|s) = Pr(\text{choose } A = a|S = s) \tag{5}$$

As said in Section 1, the goal of the agent is to maximize the total future reward, also called the *return*, G. However, since in principle a task could continue indefinitely in the future, the return in general could be infinite. A common workaround infinite values is discounting future rewards: basically we consider a reward $\gamma^{t'-t}$ times less valuable if it's received at time $t' > t$, where $\gamma \in (0, 1)$ that is called *discount factor*. In this way the return of a trajectory is:

$$G = \sum_{t=0}^{\infty} \gamma^t R_t \tag{6}$$

This implies that at each step t the agent must act to maximize the future expected reward, or *reward-to-go*, denoted with G_t:

$$G_t = \sum_{u=t}^{\infty} \gamma^{u-t} R_u \tag{7}$$

Under this notation, the value $v_\pi(s_t)$ of a state s_t under a policy π is defined as:

$$v_\pi(s_t) = \mathop{\mathbb{E}}_{\pi} \left[\sum_{u=t}^{\infty} \gamma^{u-t} R_u \middle| S_t = s_t \right] \tag{8}$$

where the expected value is considering both the environment dynamics' distribution $p(S_{t+1}, R_t|s_t, a_t)$ and the policy distribution $\pi(A_t|s_t)$.

Another kind of value is used throughout Reinforcement Learning and that is the state-action value, or Q-value, $q_\pi(s, a)$. This is defined as the return expected from taking action a in state s and following the policy π afterwards:

$$q_\pi(s_t, a_t) = \mathop{\mathbb{E}}_{\pi} \left[\sum_{u=t}^{\infty} \gamma^{u-t} R_u \middle| S_t = s_t, A_t = a_t \right] \tag{9}$$

This is particularly useful because is encoding an actionable knowledge, since it gives us the ability to directly compare which action is best to take in a certain state given a certain policy. On the other hand, the state value alone is not helpful in deciding which action to take next.

2.1 Finite trajectories and bootstrapping truncated episodes

We have seen in the previous section that an agent in general has to maximize its expected return over an infinite number of steps. There are some tasks (e.g. solving a maze), called *episodic*, that admit a terminal state (e.g. finding the exit of the maze), which once reached ends the agent-environment interaction. In these cases we can set $\gamma = 1$ and, calling the terminal step T, we will set all rewards subsequent to T to be zero.

However many tasks cannot be reduced to episodic tasks, yet in practice we can't wait for them to finish either, thus we restrict their trajectories to a maximum number of time-steps T after which are truncated to avoid never-ending episodes. This means that we have access only to the

partial sequence of the rewards $r_0, \ldots, r_{T-1}$ and not to the infinite series that we have to maximize for our objective.

One example of an infinite-horizon tasks would be the following: an agent is indicated a point on a map and has to reach it as fast as possible; once reached the point, a new one appears, the agent has to reach it again and so on. The objective is to reach as much points as possible in a given amount of time T. This is an example of a single episodic task (reaching an indicated point) turned into a task that could in theory proceed forever and then for convenience is truncated at an arbitrary step T. Another example is the balancing of a cartpole (see Figure 2), which is an inverted pole put on top of a cart that can move along some tracks; the agent has to move the cart left or right at each step in order to balance the pole. The environment is designed so that an optimal agent can keep the pole balanced forever, thus the environment terminates the episode when the pole falls or T steps are passed.

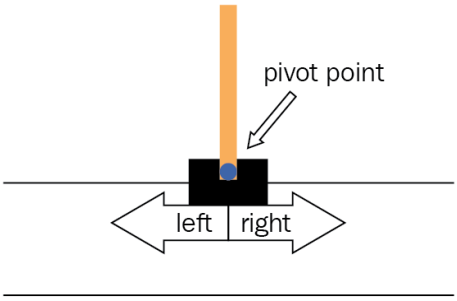


Figure 2: Representation of the CartPole environment. An inverted pendulum attached to a cart must be balanced by the agent moving the cart either right or left at each timestep.

Let's focus on this last example and assume for simplicity that the state space of the system is composed of 5 states[2]: when the pole is almost vertical but left or right pending (2 states), when it's too inclined to be balanced again, on the left or on the right, (other 2 states) and when it's fallen (terminal state). We also assume that the optimal policy is making the pole swing from the almost vertical but left-pending position to almost vertical but right-pending one and receives a reward of $+1$ for each step in which the pole does not fall.

Now, the problem with the time truncation is that if we re-define the task objective to be the maximization of the truncated episode return $G = \sum_{t=0}^{T-1} \gamma^t r_t$, the value of a state is going to depend on the time step at which we encounter the state, and not just on the characteristic of the state, since the expected reward under the optimal policy π_* (for cartpole task) is always:

$$V_*(s_t) = \sum_{u=t}^{T-1} r_u \Big|_{r_u=1} = \frac{1-\gamma^{T-t}}{1-\gamma} \tag{10}$$

In a certain sense the dynamic of the system is no more time-homogeneous, because at step $t = T$ there is a transition to a state that can be considered terminal. This in general is creating a problem in estimating the value of a state, because we never consider the time variable in doing so. However if we consider again the same problem with infinite-horizon, i.e. without time truncation, we remove the time dependence from the state value. For example in the case of the optimal policy for cartpole we would have a finite discounted value for the two states between which the system

[2]A sensible way of describing the system would be to provide the cart position and velocity, together with the pole angle and angular velocity. However many possible representations of the problem are possible, for instance we can group similar states together using our physical understanding of the problem in order to make it simpler.

swings[3]:

$$V_*(s_t) = \sum_{u=t}^{\infty} \gamma^{u-t} r_u \Big|_{r_u=1} = \frac{1}{1-\gamma} \equiv V_* < \infty \tag{11}$$

The final step to solve this issue is to be able to estimate V_* by experiencing only the steps up to T. We can do that with a practice that is called *bootstrapping*, in which we decompose the total expected reward as the reward obtained in the first $T-t$ steps plus the value of the state s_T:

$$\begin{aligned} V(s_t) &= \sum_{u=t}^{\infty} \gamma^{u-t} r_u \\ &= \sum_{u=t}^{T-1} \gamma^{u-t} r_u + \gamma^{T-t} \sum_{u=T}^{\infty} \gamma^{u-T} r_u \\ &= \sum_{u=t}^{T-1} \gamma^{u-t} r_u + \gamma^{T-t} V(s_T) \end{aligned} \tag{12}$$

This expression is self-consistent with the analytical value of V_* that we have derived:

$$V_*(s_t) = \sum_{u=t}^{T-1} \gamma^{u-t} r_u \Big|_{r_u=1} + \gamma^{T-t} V_*(s_T) = \frac{1-\gamma^{T-t}}{1-\gamma} + \frac{\gamma^{T-t}}{1-\gamma} = \frac{1}{1-\gamma} = V_* \tag{13}$$

Ultimately, to get rid of the time-dependence of the state value we re-extended the episode to an infinite-horizon task and estimated the task discounted return as the empirical return until truncation plus the discounted value of the last state encountered. The key point to understand is that we are using the fact that the extended task is time-homogeneous to predict how the trajectory and the reward are going to be in the future.

This technique can be considered as part of the RL task's set of rules designed to enable the agent's learning process and can be applied to all algorithms where a value estimation is present.

[3] All the expressions making use of V_* in this section are only valid for the cartpole environment as we described it and in the case of the optimal policy. We use the optimal policy of cartpole as an analytical example because of its simplicity, but the bootstrapping technique can be used for every policy and every environment.

3 Tabular methods

In the case of finite MDPs we have a finite number of possible states and actions combinations, thus we can solve any RL task by simply memorizing what to do in every specific situation. We refer to this class of methods as tabular methods, since we treat the policy π and the values v as tables, whose entries are the independent parameters to be learned.

3.1 Bellman equation, optimal policies and optimal value functions

The value $v_\pi(s)$ of a state s under a policy π is by definition the expected return from being in that state and following policy π from there. If we look one step forward, from s we will choose an action a with probability $\pi(a|s)$ and then access to a state s' and get a reward r with joint probability $p(s', r|s, a)$; we can then express the expected value of state s as a weighted sum over the pairs of next rewards and state values:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{14}$$

This equation is called *Bellman equation* and captures the consistency condition that holds between the value of s and the value of its possible successor states. The value function v_π is the unique solution of Eq. 14 given a certain policy π.

Now let's imagine that we are provided with a table in which there is an entry for every state $s \in \mathcal{S}$, whose value is the optimal value $v_*(s)$, i.e. the maximum possible reward achievable on average from any policy starting from that state. Because it is the optimal value function, however, v_*'s consistency condition can be written in a special form without reference to any specific policy:

$$v_*(s) \equiv \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_*(s')) \tag{15}$$

Eq. 15 is also called *Bellman optimality equation*.

This means that any policy π_* for which v_* is a solution of Eq. 14 (i.e. any optimal policy) must choose an action greedily, in order to maximize the right hand side of Eq. 15:

$$\pi_*(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_*(s')) \tag{16}$$

In other words the knowledge of the optimal values v_* together with the knowledge of the environment dynamics $p(s', r|s, a)$ let us derive an optimal policy through a one-step-ahead greedy search instead of having to look explicitly to long-term returns.

The Bellman optimality equation can in theory be solved as a non-linear system[4] of n equations (one for every state s) in n unknowns (the n state values), where n is the number of states in the system. However this in practice is rarely feasible, since to solve the system at least three assumptions need to hold:

1. The environment dynamics $p(s', r|s, a)$ is perfectly known.
2. We have enough computational power to solve an n x n non-linear system.
3. The system is Markovian.

Property 1 and 3 might hold in many cases, but the state space for most interesting tasks is too large to let us solve the system exactly.

A more practical approach is to find an approximate solution of the Bellman equations by means of iterative algorithms. To do that the general idea is to turn the consistency condition of

[4]The equations are non-linear because of the max operation. On the contrary non-optimal Bellman equations are linear but depend on a generic policy, thus they cannot be used to find an exact optimal solution.

the Bellman optimality equation in an update rule, where we iteratively update the left hand side of the equation to match the right hand side.

For instance in the Value Iteration algorithm (see Sutton and Barto 1998, chapter 4) at each iteration we sweep the entire space state $\mathcal{S}$ and update the state values with the following rule:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_k(s')) \tag{17}$$

For $k \to \infty$ the value can be shown to converge to v_* and the optimal policy will be given by Eq. 16.

As a final remark notice that if instead of learning the optimal state values v_*, we were to learn the optimal Q-values q_*, it would be straightforward to derive the optimal policy from them, even without having any knowledge about the environment dynamics:

$$\pi_*(s) = \arg\max_a q_*(s, a) \tag{18}$$

This is because Q-values already encode the one-step-ahead search that we have to perform in the case of the state values. In fact for a generic policy it holds:

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{19}$$

from which we can obtain Eq. 18 by plugging this result in Eq. 16 .

3.2 Q-learning

We have just seen that if we don't know $p(s', r|s, a)$, a sufficient way to solve a RL task is to learn its optimal Q-values, but how can we learn them?

An answer to that question is provided by the so-called Q-learning algorithm (C. J. C. H. Watkins 1989). In the case of a finite MDP, the Q-values are a matrix whose entries can be accessed by state-action pairs (s, a).

The workflow of the algorithm is the following:

1. Initialize all Q-values with arbitrary values, except the entries corresponding to terminal states, whose values are set to 0. We refer to them as $q_0(s, a)$.
2. Use as initial policy the greedy policy $\pi_0(s) = \max_a q_0(s, a)$.
3. Given a state s, select the action $a = \pi_0(s)$ and submit it to the environment. The new state s' and the reward r for taking a are received.
4. Update the entry (s, a) of the Q-values table as $q_1(s, a) \leftarrow (1-\alpha_0)q_0(s, a)+\alpha_0[r+\gamma q_0(s', \pi_0(s'))]$, where α_0 is called *learning rate* and is a real number in $(0, 1)$.
5. Repeat from point 3 with π_k, q_k and α_k instead of π_0, q_0 and α_0 until a satisfying convergence is achieved, e.g. when the difference between $q_{k+1}(s, a)$ and $q_k(s, a)$ is negligible for all (s, a) pairs.

The algorithm has been shown to converge (C. J. Watkins and Dayan 1992) with probability 1 to the optimal Q-values q_*, provided that all state-action pairs are sampled infinitely-many times in the asymptotic limit and the sequence of learning rates $(\alpha_0, \alpha_1, \ldots)$ satisfies[5]:

$$\sum_{i=0}^{\infty} \alpha_i = \infty, \quad \sum_{i=0}^{\infty} \alpha_i^2 < \infty \tag{20}$$

[5]The actual requirement is that for each sub-sequence $\alpha_{n^i(s,a)}$ the two constraints are satisfied, where $n^i(s, a)$ is the i-th time that the state-action pair (s, a) has been encountered and n is the number of updates done so far. Then for $n \to \infty$, the Q-values converge to the optimal Q-values with probability 1 for all state-action pairs.

The requirement of sampling all state-action pairs with non-zero probability throughout the whole learning process is a strong one and is violated by the simple algorithm outlined above. This is because the greedy policy $\pi(s) = \max_a q(s,a)$ is deterministic, thus is assigning all the probability to a single action, preventing a full exploration of the possible strategies. Using a greedy policy can fail to discover the best strategies for many reasons:

- The resulting policy will depend on the initial conditions: if we initialize all Q-values in a conservatory way, e.g. with a value much lower than the real optimal value, the greedy policy will never select them, hence it will remain based on totally biased Q-values. On the other hand an optimistic initialization can boost exploration and then the agent will naturally re-evaluate the initial values until their value becomes lower than the highest optimal Q-value for that state. This should grant us good estimates of the highest Q-values for each state, but will probably yield poor estimates of the lower ones.
- During run-time the algorithm is estimating the Q-values based on other Q-values, hence they will be biased for a transient time before (possibly) converging to the true values. This would happen even with other policies, but for deterministic policies this can lead to rule out forever good strategies based on a biased estimate.
- Another source of bias is given by a poor statistics for estimating the expected Q-values with sample averages. In general the reward is a random variable distributed accordingly to $p_r(r_t|s_t,a_t)$, thus many samples are needed to converge to the true value $\mathbb{E}[R_t|s_t,a_t]$.

An easy fix would be to adopt instead a stochastic policy $\pi(a|s)$ that still reflects a preference for the action with the highest Q-learning and that has a parameter controlling its entropy (how stochastic it is), so that we can use its deterministic version when we need optimal answers. There are two main policies of the sort that are commonly used in the RL literature: the ϵ-greedy and the softmax (or Boltzmann) policies.

The ϵ-greedy policy is:

$$\pi_\epsilon(a|s;q) = \begin{cases} 1-\epsilon+\frac{\epsilon}{|\mathcal{A}|} & if\ a = \arg\max_{a'} q(s,a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{21}$$

whereas the softmax policy:

$$\pi_{SM(\tau)}(a|s;q) = \frac{\exp(\frac{q(s,a)}{\tau})}{\sum_{a'\in\mathcal{A}}\exp(\frac{q(s,a')}{\tau})} \tag{22}$$

The main difference is that the ϵ-greedy policy shares the probability of an exploration move equally between all actions, whereas the softmax is taking into account the Q-values of all the actions to assign them their probabilities, thus is more selective as an exploration strategy. In both cases, in the limit where the parameters ϵ and τ, controlling the entropy of the policies, are going to 0, we re-obtain the greedy policy based on the Q-values.

Finally to obtain the Q-learning update we have to substitute the term $q(s',\pi(s'))$ with $q(s',\arg\max_{a'}\ q(s',a'))$, since the policy π used to sample the trajectories has been modified and we want to evaluate the next state like if we were following the best policy available given our current knowledge, i.e. the greedy policy. Thus the final update for Q-learning is:

$$q_{k+1}(s,a) = (1-\alpha_k)q_k + \alpha_k(r + \gamma q_k(s', \arg\max_{a'}\ q_k(s',a'))) \tag{23}$$

where is intended that the only entry updated is the one corresponding to the state-action pair (s,a) that has been experienced. A pseudo-code version of the algorithm is reported in box Algorithm 1.

To sum up, one of the requirements for the convergence of the Q-learning algorithm is a continuous exploration of the state-action space during the learning process. This is because even though

most of the time trying a random action (or one that is sub-optimal from the perspective of our current knowledge) is going to be a waste of time, it might be the case that we discover a new viable strategy, possibly better than the optimal one so far. Hence all RL algorithms must find a balance between exploiting the currently knowledge and exploring all the other possible strategies, with exploration that usually is somehow decreased along the training process because it is usually subject to diminishing marginal returns and in the last phase of learning the agent should focus mainly on refining the best strategies discovered so far.

Algorithm 1 Q-learning

procedure
  $q_0(s, a) \leftarrow$ constant Q $>$ max(episode return)
  $q_0(s_T, a) \leftarrow 0$ ▷ for terminal states s_T
  $\pi_k(a|s) \leftarrow \pi_\epsilon(a|s; q_k)$ (or $\pi_{SM(\tau)}(a|s; q_k)$)
  γ, α and ϵ (or τ)

  for n=1, ..., N **do**
    $s_0 \sim \rho_0(s_0)$ ▷ sample initial state
    for t=0, ..., T-1 **do**
      $k \leftarrow (n-1)T + t$
      $a_t \sim \pi_k(a_t|s_t)$ ▷ sample action
      $s_{t+1}, r_t \sim p(s_{t+1}, r_t|s_t, a_t)$ ▷ get reward and new state
      $q_{k+1}(s_t, a_t) \leftarrow (1-\alpha)q_k(s_t, a_t) + \alpha(r_t + \gamma max_a q_k(s_{t+1}, a))$ ▷ update only the entry of (s_t, a_t)
    end for
    $\alpha \leftarrow C_\alpha \alpha$, with $C_\alpha \in (0, 1)$ ▷ decrease learning rate
    $\epsilon \leftarrow C_\epsilon \epsilon$, with $C_\epsilon \in (0, 1)$ (or τ instead of ϵ if using softmax policy) ▷ decrease exploration
  end for
  return q_N
end procedure

3.3 On-policy and off-policy learning

In the Q-learning algorithm we have seen that to estimate correctly the optimal Q-values, it is necessary to introduce a stochastic policy to ensure that all state-action pairs will be explored continuously throughout learning. However, when it was time to estimate the value $q(s, a)$ we assumed that once arrived in state s' we would have chosen the next action a_{eval} accordingly to the greedy policy, i.e. $a_{eval} = \arg\max_a q(s', a)$, even though the episode continued by sampling another action a' from the stochastic policy, thus in principle $a_{eval} \neq a'$.

This method of using separate policies for sampling actions and evaluating values is called *off-policy learning*. The sampling policy is usually referred to as *behavioral policy* μ, whereas the evaluation policy (which is also the one that we want to be optimal at the end of the learning) is called *target policy* π.

In Q-learning we are learning the Q-values accordingly to the target policy π, even though all experience comes from the behavioral policy μ. This is possible because during the evaluation of the Q-values we can remove every reference from μ since:

1. Action a, that has been chosen by μ, is considered as given, thus it does not matter if the probability to select it was $\mu(a|s)$ instead of $\pi(a|s)$.

2. The next state s' and the reward r are sampled from the probability distribution $p(s', r|s, a)$, that does not depend on the policy that chose a.

3. The return expected from being in state s', i.e. the state value $v(s')$, depends on which policy we are following and, if we were to approximate it with the experimental return G, we would have estimated the state value accordingly to policy μ. Instead, by using $v(s') = \max_a q(s', a)$, we are using an estimate from the greedy policy, because that's the one that would always choose the best action accordingly to the current estimates of the Q-values.

In a certain sense, this update is estimating the Q-values accordingly to π because:

- it's making use of a virtual greedy action a_{eval}, that would be selected if from that point we switched from following μ to following π.
- it's estimating only from past experience (i.e. other Q-values) how much reward we would have obtained if the virtual action would have been played.

There is an alternative formulation to learn the Q-values in which the behavioral policy μ coincides with the target policy π. In this case we don't need to use virtual actions to access the Q-values' table, but we can use the action that has been actually chosen, since it's from the same policy that we are evaluating. Notice that we are estimating the value of a state-action pair as the expected reward r immediately obtained plus the expected state value $v(s')$ of the new state, so that when the algorithm will converge it will hold:

$$\begin{aligned} q_\pi(s,a) &= \sum_r p_r(r|s,a)r + \gamma \sum_{s'} p_s(s'|s,a) v_\pi(s') \\ &= \sum_r p_r(r|s,a)r + \gamma \sum_{s'} p_s(s'|s,a) \sum_{a'} \pi(a'|s') q_\pi(a', s') \\ &= \mathop{\mathbb{E}}_{r \sim p_r(\cdot|s,a)}[r] + \gamma \mathop{\mathbb{E}}_{s' \sim p_s(\cdot|s,a)} \left[\mathop{\mathbb{E}}_{a' \sim \pi}[q_\pi(s', a')] \right] \end{aligned} \tag{24}$$

In the case of Q-learning specifically, the expected value under policy π is a direct evaluation taking the maximum, i.e. $\mathop{\mathbb{E}}_{a' \sim \pi}[q_\pi(s', a')] = q_\pi(s', \arg\max_a q_\pi(s', a))$. However, in the case of a stochastic policy, the expected value would be $\mathop{\mathbb{E}}_{a' \sim \pi}[q_\pi(s', a')] = \sum_a \pi(a|s') q(s', a)$ and we can have an approximate estimate of it by considering a single action a, as long as it's sampled accordingly to $\pi(a|s')$. After sampling a transition $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, we can update the Q-values as follows:

$$q_{k+1}(s_t, a_t) = (1 - \alpha_k) q_k(s_t, a_t) + \alpha_k (r_t + \gamma q_k(s_{t+1}, a_{t+1})) \tag{25}$$

This algorithm, which uses the target policy equal to the behavioral policy and only evaluation actions that have actually been played, is called SARSA (Rummery and Niranjan 1994) (from the transition used in the update) and is part of the so-called *on-policy* algorithms' class.

A drawback of exploration in the on-policy algorithms is that sometimes we will evaluate a promising state-action pair (s_t, a_t) by sampling an exploratory action a_{t+1} in state s_{t+1}; since exploratory actions are mostly sub-optimal, this is going to reflect on the estimate of the Q-value $q(s_t, a_t)$. This is why particularly on-policy algorithms have to be cautious on how much they explore and use a slightly smaller learning rate than their off-policy counterpart in order to reduce the impact of exploration noise on learning. One consequence of this is that on-policy algorithms tend to be more conservative while learning than off-policy ones, sometimes having better performances during training and converging to more risky but rewarding policies later than off-policy algorithms.

3.4 Model-based and model-free algorithms

In Section 3.1 we have shown how to derive a policy from the state value if the environment dynamics' distribution $p(s_{t+1}, r_t | s_t, a_t)$ is known. Algorithms that make use of that knowledge are called *model-based*, whereas algorithms that estimate those distributions implicitly with sample

averages are called *model-free*. All the algorithms that we discuss in this work are model-free, thus, before going on, we want to briefly present the arguments in favour and against these kinds of models.

First of all, model-based algorithms require access to the environment dynamics, or, if not directly available, have to learn how to approximate it. In the tabular settings, given enough experience, we can estimate the transition probability $p_s(s_{t+1}|s_t, a_t)$ with the relative frequency of each transition and the expected reward $r_t(s_t, a_t)$ with a running average or if the reward is discrete, estimating the full distribution $p_r(r_{t+1}|s_t, a_t)$ in a similar way of p_s. However this becomes more complicated if we want to transition from a tabular setting to a functional approximation of the environment distribution; of course it is still possible to learn a model of the environment, for example using a Gaussian Process (Deisenroth, Neumann, and Peters 2013, Boedecker et al. 2014 and Ko and Fox 2009), a Gaussian Mixture Model (Khansari-Zadeh and Billard 2014) or a Deep Neural Network (Nagabandi et al. 2018), but this is adding a layer of complexity to the problem.

Now let's assume that we have a model of the environment dynamics. What are the advantages of knowing it? In other words how can we use it to solve the RL task? One way is by estimating the state values in an iterative way as done in the Value Iteration algorithm Eq. 17. Although it looks very efficient because once the model is known it requires no interaction at all with the environment, the amount of computation required to update all states scales quadratically with the number of states (because we need to consider all the combinations of s and s'), which in turn scale exponentially with the number of variables that compose the state as we will see in the next section. Thus this algorithm is practical only for the simplest tabular methods. Then if we consider the functional approximation case, it's even less clear how to generalize it in an efficient way (since we are assuming a continuous space we should basically perform some kind of integration to estimate $q(s, a)$, but then we should also choose which states s to update at each iteration, since in principle there would be an infinite amount of them).

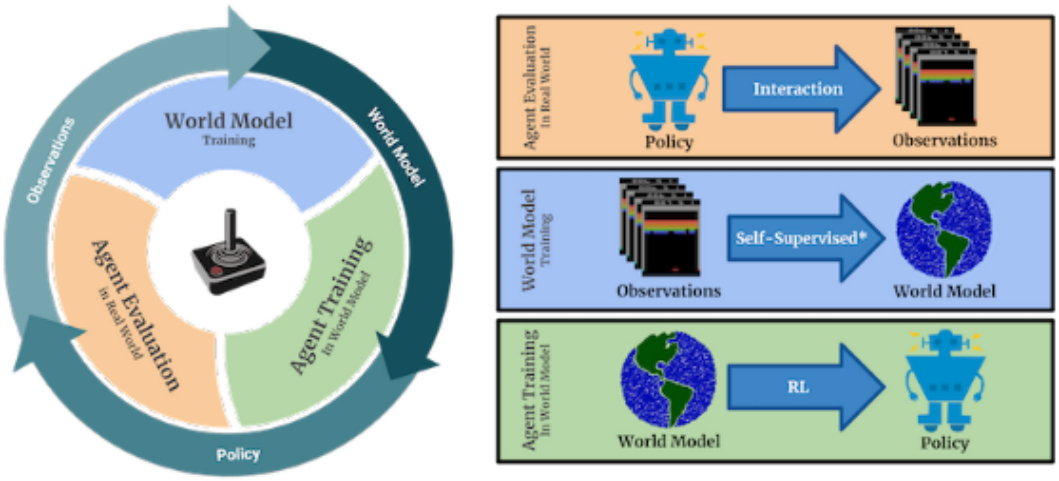
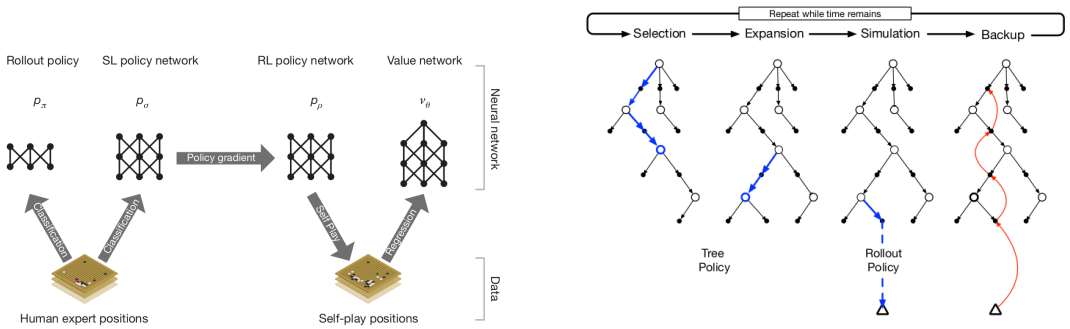


Figure 3: Main loop of SimPLe: 1) the agent is evaluated on the real environment and produces experience in the meanwhile; 2) the world model is trained on the experience produced by the agent during phase 1; 3) the agent is trained making experience on the simulated environment provided by the world model. Source: Model Based Reinforcement Learning for Atari, Kaiser et al. 2019.

So the option of taking into account the full range of possible transitions one step ahead turns out computationally infeasible. The next option is to simulate experience by interacting with the model of the environment just as we would do with the real one. The good thing about this is that we decrease the number of times in which we interact with the real environment, which can

be a bottleneck in some cases, especially when the environment is a physical one (e.g. robotics applications). A notable example in this category is SimPLe (Kaiser et al. 2019, see Figure 3 for the training loop), where the authors use a deep neural network to learn a model of the environment for many Atari games from the Arcade Learning Environment (Bellemare et al. 2013) and train a policy with a model-free algorithm (PPO, Schulman, Wolski, et al. 2017) mostly on experience simulated by the learned model. This resulted in a sample-efficient algorithm, which could obtain better results in the low-data regime (e.g. 100K transitions instead of 1M or more) than the same PPO algorithm trained on the real environment. However this sample-efficiency is only partial, since it is not counting the 15M simulated transitions generated by the neural network model of the environment on which the agent was trained; this together with the fact that to simulate a transition their model needed 32 ms, against the 0.4 ms used by the original environment, probably implied that the training time of the model-based agent was approximately 1200 times higher than the one of the model-free agent. The problems with this approach are that the experience produced could be biased if the model of the environment is not perfect and that the gains in using a model of the environment strongly depend on how fast and accessible is the real environment. For instance all the environments that we are going to use, which are based on virtual engines, the access to real experience is unlimited and it will probably be faster than the one we would get from our learned model; however looking forward to real-world applications it will probably be the case that working in a low-data regime will be a hard constraint, thus making the speed issue a secondary one. To sum up the big issues with this approach are that we are doing the same things we do in model-free algorithms, but with an intermediate layer that adds complexity and may introduce noise, without assuring gains in speed.



(a) (b)

Figure 4: (a) Summary of the neural networks used by AlphaGo and the methods and data with which they were trained. (b) Monte Carlo Tree Search. Every time the agent encounters a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration is composed of the four phases: selection of the leaf node, (eventual) expansion of the tree, simulation until the end of the episode, and backup to update all previous Q-values based on the result of the simulated trajectory. Sources: (a) Mastering the game of Go with deep neural networks and tree search, Silver, Huang, et al. 2016; (b)Reinforcement Learning: An Introduction, Sutton and Barto 1998.

One of the biggest breakthroughs in RL was when DeepMind's AlphaGo (Silver, Huang, et al. 2016) algorithm defeated the 18-times world champion of Go in 2016. AlphaGo used a policy network that outputs action probabilities, and a value network that outputs state values. The policy network was trained initially by supervised learning to accurately predict human expert moves, and was subsequently refined by policy-gradient reinforcement learning. The value network was trained to predict the winner of games played by the policy network against itself. Once trained, these networks were combined with a Monte-Carlo Tree Search (MCTS) to provide a forward search, using the policy network to focus the search on the most promising ramifications, and

using the value network (in conjunction with Monte-Carlo trajectories using a fast rollout policy) to evaluate positions in the tree. MCTS can be seen as form of model-based policy improvement at decision-time that gave AlphaGo an edge over previous Go programs. However MCTS in itself is not a learning algorithm, since it is not trying to learn a global policy or global values, but only deciding what to do in a given state starting from a naive policy and improving it by means of an efficient forward search.

Its next version was AlphaGoZero (Silver, Schrittwieser, et al. 2017) can be regarded as a more fundamental way of learning through model-based reinforcement learning: in fact it did not use any human data (so no supervised learning) and was trained through self-play, training its policy and value networks to respectively predict the MCTS outcome and the probability of winning the game. In this way MCTS was in the learning loop and actively helping to train the policy.

Overall we can say that model-free algorithms are nowadays preferred over model-based ones for their simplicity, speed and performance even though issues like data-efficiency, stability and generalization to different tasks might require breakthroughs in model-based algorithms in order to be solved.

3.5 Shortcomings of tabular methods

Tabular methods are a simple and intuitive way of solving finite Markov Decision Processes by estimating one by one all possible state (or state-action) values. Moreover the great majority of theoretical results on Reinforcement Learning algorithms (e.g. assurance of convergence and sample complexity of an algorithm to converge) are demonstrated only for this class of algorithms. However tabular methods also have strong limitations, namely:

1. **Curse of dimensionality**: the number of possible values that a state composed by n independent discrete variables can assume is exponential in n. Let $\overline{S} = (S_1, \dots, S_i, \dots, S_n)$ be a vectorial state and let's assume for simplicity that all S_i can assume exactly m values, i.e. $S_i \in \{s_{i1}, \dots, s_{ij}, \dots, s_{im}\}$, $\forall i \in n$, then the number of possible values of $\overline{S}$ is m^n. If this looks like a severe limitation in case of some high-level variables, in the case of a state composed by L rectangular layers of W x H pixels (like an RGB image) with M possible values each, the number of possible combinations would be M^{WHL}, that even for a binary image ($M = 2$) of a single layer ($L = 1$) of dimension 16x16, would result in $2^{256} \approx 10^{77}$ possible values.

2. **Lack of generalization capabilities**: a tabular function $F : \mathcal{S} \to \mathbb{R}$ is basically an arbitrary mapping between inputs and outputs, which is equivalent to an entire set of $|\mathcal{S}|$ independent parameters associated to $|\mathcal{S}|$ different inputs. The problem with that is that the knowledge of the property of a state s_i tells us nothing about the same property of any other state s_j, thus we are not learning to recognize similarities between inputs and to associate them to similar properties, but we are just memorizing all inputs independently of each other.

These two facts together imply that the memory required to solve a RL task grows exponentially with the dimension of states and so does the minimal experience required to estimate all their values. Furthermore from an epistemological point of view it's arguable that this class of algorithms is learning in the weakest possible way, i.e. memorizing facts and using them following a set of predefined rules to solve a task.

All of this motivates the substitution of tabular methods with functional approximation methods, where the state values, state-action values and policies instead of being look-up tables are functions parametrized by a number of parameters much smaller than the number of possible states. This transition comes with its own set of challenges, ranging from stability issues, to expressive power issues, over-fitting and poor theoretical guarantees. A popular way[6] to approximate a function,

[6]Part of the Neural Networks popularity also is due to the fact that they have become very efficient to train since when NVIDIA's Graphics Processing Unit have been adapted to perform Neural Networks' computations in parallel

although not the only possible one, is to use Neural Networks (NNs), since they have the property of being universal function approximators (Cybenko 1989). Moreover different kinds of layers have been proposed to optimize neural networks for different tasks, like processing images (with Convolutional Layers) or memory processing (with Recurrent Layers).

and many state-of-the-art programming frameworks are completely open-source (e.g. TensorFlow from Google and PyTorch from Facebook).

4 Policy Gradient and Actor-Critic Methods

The objective of a RL task is always to learn a policy $\pi : S \longrightarrow P(A)$, that will maximize the expected return over the whole episode. In the case of value-based methods (e.g. Q-learning and SARSA), we first learn the Q-values $Q : S \longrightarrow \mathbb{R}^{|\mathcal{A}|}$ and then a policy on the Q-values $\pi : \mathbb{R}^{|\mathcal{A}|} \longrightarrow P(A)$, but we can also devise an algorithm that just learns π.

Learning directly a policy instead of learning the Q-values has three main advantages:

1. It's a straightforward process, because it's learning directly what we need to learn. In the Q-learning case instead, in addition to learning the Q-values, we have to specify a policy based on them to solve the RL task.

2. By modelling a policy we are always able to learn stochastic policies[7], whereas value-based method might not be equipped to do that. For example a greedy policy that selects the action with greater Q-value might be very efficient but is deterministic and not stochastic.

3. An arbitrarily small change in the estimated value of an action can cause it to be, or not be, selected. This does not happen with policy learning, since a small change in its outputs is by definition a small change in the probability of selecting the various actions.

The last point might be mitigated by using a stochastic policy, but most of the times they are used in an almost deterministic way and they don't have any tuning mechanism to reproduce optimal stochastic policies. For instance the ϵ-greedy policy can be seen as a perturbation of the greedy policy with very restrictive constraint on the distributions that can represent (all but one action are constrained to have the same probability of being chosen, which is totally unrealistic for a generic policy). Instead the softmax case is more interesting, because by varying τ we can model a whole class of policies in which the probability of each action in being chosen is proportional to its relative exponential weight, with τ that is compressing the exponential weights range for $\tau >> \max Q$ or increasing it for $\tau << \max Q$. Even in this case we are assuming that a softmax mapping exists between the Q-values and the optimal policy, whereas in general that is not the case, e.g. the optimal policy might cut-off the probability to all but the top n actions and use a value of τ which is incompatible with that cut-off. Interestingly, it can be shown that Q-learning methods are equivalent to a certain class of policy methods if the chosen policy is the softmax policy with an additional entropy regularization term (Schulman, Abbeel, and Chen 2017).

Additionally, the susceptibility of value-based policies to small changes in action's values is exacerbated when a complex function approximation of the Q-values is used, since all the Q-values become interdependent and adjusting one Q-value can radically change the policy in different regions of the state-action space. In the next sections we will derive a class of algorithms, called Policy Gradient methods, that learns directly a function approximation of the policy, enabling us to learn general stochastic policies even in the case of MDPs that are intractable with tabular methods.

4.1 Learning a parametric policy with policy gradient methods

In the following discussion we will assume that our policy π_θ will be parametrized by some function differentiable in its parameters θ, may it be a Neural Network or even a simple linear function. Furthermore for simplicity in notation we derive all algorithms for episodic tasks, but for infinite-horizon tasks the derivation can be straightforwardly adapted by taking $T = \infty$ in all estimates

[7] Stochastic policies can be the only optimal policy in a game theoretic situation where we have an opponent and the only Nash equilibrium is given by a mixed strategy (e.g. one-stage repeated games like n rounds of rock-paper-scissors) (see Tadelis 2013, chapter 6). Instead in the single-agent setting, if the environment is partially observable or inherently stochastic, adopting a stochastic policy can reduce the variance of the outcomes and speed up the convergence of the algorithm, which can later converge to an optimal deterministic policy. The key point is that an optimal deterministic policy can in principle be estimated if the probability distribution of the current state and the outcomes is known (or is implicitly learned by the policy/values), but at the beginning of the training it may be unknown, thus a stochastic policy is able to take into account this uncertainty and model the decision process in accord to it.

of the episode return and re-introducing the discount factor where missing.[8] The objective of the learning process is to maximize the expected return J under the policy π_θ:

$$\theta^* = \arg\max_\theta J^{\pi_\theta} = \arg\max_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \tag{26}$$

where

- $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_T, a_T, r_T, s_{T+1})$ is a state-action trajectory.
- $\tau \sim \pi$ means that the every action in the trajectory τ has been sampled following policy π, i.e.
  $a_i \sim \pi(\cdot|s_i)$, $\forall i \in [0, T]$.
- $R(\tau) = \sum_{t=0}^{T} r_t$ is the episode return (i.e. the sum of all rewards received during the episode).

Most of the times we will actually use discounted rewards, so the return will be $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$.

To learn how to approximate the optimal policy we would like to estimate the gradient $\nabla_\theta J^{\pi_\theta}|_\theta$ and then update the parameters with a simple gradient ascent rule (or a more complicated momentum-based optimizer, like Adam):

$$\theta_{k+1} \longleftarrow \theta_k + \alpha \nabla_\theta J^{\pi_\theta}\Big|_{\theta_k} \tag{27}$$

where α is called *learning rate* and is a real number in (0,1). This is used to compensate for the fact that our estimate of the gradient contains noise because it's based only on few trajectories, thus we should move more slowly towards the optimal policy in order to assure the convergence of the optimization process.

4.2 Policy Gradient Theorem

To estimate the expected return, discounted or not, we will need to know the probability of a trajectory under a certain policy and to take its derivative with respect to the policy's parameters. Assuming that there is a distribution ρ_0 over the possible initial states s_0 and that from the current state s_t we get to a new state s_{t+1} in a stochastic way that depends also on the action a_t chosen at that step, i.e. $s_{t+1} \sim p_s(\cdot|s_t, a_t)$, the probability of a trajectory under a policy π will be:

$$P(\tau|\theta) = \rho_0(s_0) \prod_{t=0}^{T} p_s(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{28}$$

Computing the gradient of J^{π_θ} we have:

$$\begin{aligned}
\nabla_\theta J^{\pi_\theta} &= \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \sum_{\{\tau\}} (P(\tau|\theta)R(\tau)) \\
&= \sum_{\{\tau\}} \nabla_\theta \left(P(\tau|\theta)R(\tau)\right) \\
&= \sum_{\{\tau\}} [\nabla_\theta P(\tau|\theta)R(\tau) + \cancel{P(\tau|\theta)\nabla_\theta R(\tau)}]
\end{aligned} \tag{29}$$

[8] Ultimately we can still use truncated trajectories to optimize infinite-horizon tasks if we are able to estimate the infinite series of discounted rewards through bootstrapping (as seen in Section 2.1). In Monte Carlo based methods (e.g. the REINFORCE algorithm) this is not possible and a truncation error is introduced, whereas in Temporal Difference methods is possible because we can use the critic for bootstrapping.

where in the last equality we used the fact that the return $R(\tau)$ obtained from a trajectory depends only on the trajectory itself and not on the policy parametrization. At this point we first make use of the so-called "log-derivative trick" to rewrite the gradient of the trajectory's probability:

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta) \tag{30}$$

which comes from the composite derivative of the logarithm of a function. The next step is to simplify the derivative of the logarithm of the probability using Eq. 28 and the independence of the environment dynamics' distribution from the policy's parameters:

$$\nabla_\theta \log P(\tau|\theta) = \cancel{\nabla_\theta \log \rho_0(s_0)} + \cancel{\sum_{t=0}^{T} \nabla_\theta p_s(s_{t+1}|s_t, a_t)} + \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{31}$$

Finally we obtain the expression for the gradient of J^{π_θ}

$$\begin{aligned} \nabla_\theta J^{\pi_\theta} &= \sum_{\{\tau\}} P(\tau|\theta) \left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right) \\ &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right] \end{aligned} \tag{32}$$

This result goes under the name of Policy Gradient Theorem (Sutton, McAllester, et al. 1999). Notice that the role of the log-derivative trick in this derivation was double: first we were able to cancel the probability terms that depended only on the environment dynamics and then to transform the sum of $\nabla_\theta P(\tau|\theta)$ over all possible trajectories following π into an expected value under that policy, which can be estimated by an average over a finite number of sampled trajectories.

4.3 REINFORCE algorithm

Although Eq. 32 is a good result, it's still not totally satisfying the fact that the gradient is changing the log-probabilities of the actions a_t proportionally to the sum of the full episode return, because an action taken at time t should be evaluated only on future rewards - *reward-to-go* G_t - and not also on past ones. In other words we would like to use the following expression to compute the gradient:

$$\nabla_\theta J^{\pi_\theta} = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{u=t}^{T} r_u \right] \tag{33}$$

To do that we will first demonstrate a lemma on the expected value of the gradient of the log-probabilities (shortened in EGLP lemma) and then use it to show the equivalence between the two formulations. The EGLP lemma states (Achiam 2018):

$$\mathop{\mathbb{E}}_{x\sim P_\theta} [\nabla_\theta \log P_\theta(x)] = 0 \tag{34}$$

where P_θ is a discrete probability distribution parametrized by parameters θ.

Starting from the normalization condition we have:

$$\sum_{\{x\}} P_\theta(x) = 1 \longrightarrow \sum_{\{x\}} \nabla_\theta P_\theta(x) = 0 \tag{35}$$

Then using the log-derivative trick

$$0 = \sum_{\{x\}} \nabla_\theta P_\theta(x) = \sum_{\{x\}} P_\theta(x) \nabla_\theta \log P_\theta(x) = \mathop{\mathbb{E}}_{x\sim P_\theta} [\nabla_\theta \log P_\theta(x)] \tag{36}$$

as claimed.

Now considering again Eq. 32 and writing explicitly the expectation operator we have

$$
\begin{aligned}
\nabla_\theta J^{\pi_\theta} &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right] \\
&= \sum_{t=0}^{T}\mathop{\mathbb{E}}_{a_t\sim\pi_\theta(\cdot|s_t)}\left[\mathop{\mathbb{E}}_{s_{t+1},r_t\sim P(s_t,a_t)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{u<t} r_u + \nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{u\geq t} r_u\right]\right] \\
&= \sum_{t=0}^{T}\left(\sum_{u<t} r_u\right)\cancel{\mathop{\mathbb{E}}_{a_t\sim\pi_\theta(\cdot|s_t)}\left[\mathop{\mathbb{E}}_{s_{t+1},r_t\sim P(s_t,a_t)}[\nabla_\theta \log \pi_\theta(a_t|s_t)]\right]} \\
&+ \sum_{t=0}^{T}\mathop{\mathbb{E}}_{a_t\sim\pi_\theta(\cdot|s_t)}\left[\mathop{\mathbb{E}}_{s_{t+1},r_t\sim P(s_t,a_t)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{u\geq t} r_u\right]\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{u=t}^{T} r_u\right]
\end{aligned}
\tag{37}
$$

where in the third equality we used the EGLP lemma to simplify the first term of the right hand side.

The algorithm that implements the Policy Gradient as in Eq. 33 is called REINFORCE (Williams 1992). See box Algorithm 2 for a pseudo-code version. For an actual implementation on a deep learning framework like PyTorch, where the policy is approximated with a neural network we don't need to manually compute $\nabla_\theta J^{\pi_\theta}$, but just an unbiased estimate of J^{π_θ} through sample average. This is because we can use PyTorch's automatic differentiation functionality to compute the gradient of J with respect to all parameters θ of our neural network; in addition typically we will compute $-J$ because all the optimizers work on loss minimization through gradient descent, instead of operating with gradient ascent.

A final, important note on the REINFORCE algorithm is that the reward-to-go is always computed after the action a_t has been decided, hence its expected value is going to be by definition the Q-value $q^{\pi_\theta}(s_t, a_t)$. In this algorithm we estimate it every time by sampling all the rewards until the end of the trajectory, but if we had access to the function $q^{\pi_\theta}(s, a)$ we could use that one instead.

Algorithm 2 REINFORCE

procedure
  $\theta_1 \leftarrow$ randomly initialized
  Set γ and α
  for n=1, ..., N **do**
    $s_0 \sim \rho_0(s_0)$ ▷ sample initial state
    Produce trajectory $(s_0, a_0, r_0, \ldots, a_{T-1}, r_{T-1}, s_T)$ following π_{θ_n}
    Save $(\log \pi_{\theta_n}(a_0|s_0), \ldots, \log \pi_{\theta_n}(a_{T-1}|s_{T-1}))$
    Compute $G_t = \sum_{u=t}^{T-1} \gamma^{u-t} r_u$ ▷ compute *reward-to-go*
    Compute $g_\theta = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_{\theta_n}(a_t|s_t) G_t$ ▷ estimate gradient
    $\theta_{n+1} \leftarrow \theta_n + \alpha g_\theta$ ▷ update policy parameters
    $\alpha \leftarrow C_\alpha \alpha$, with $C_\alpha \in (0, 1)$ ▷ decrease learning rate
  end for
  return π_{θ_N}
end procedure

4.4 REINFORCE with baseline

Another consequence of the EGLP lemma is that in Eq. 33 we can subtract to the reward-to-go any function $b(s_t)$, called baseline, that depends on the current state s_t, since

$$\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)b(s_t)\right] = \sum_{t=0}^{T}\mathop{\mathbb{E}}_{s_t\sim p_s(s_{t-1},a_{t-1})}\left[b(s_t)\underbrace{\mathop{\mathbb{E}}_{a_t\sim\pi_\theta(\cdot|s_t)}[\nabla_\theta \log \pi_\theta(a_t|s_t)]}\right] = 0 \quad (38)$$

A natural choice for the baseline is the on-policy value v^π, since is defined as the expected reward-to-go from state s_t:

$$V^\pi(s_t) = \mathop{\mathbb{E}}_{\tau_{[t:T]}\sim\pi_\theta}\left[\sum_{u=t}^{T} r_t\right] \quad (39)$$

In this way if after taking action a_t the return from t to the end of the episode is equal to the estimated value $v^\pi(s_t)$, we will not increase nor decrease the probability of choosing that action again. It is observed empirically (see Sutton and Barto 1998, chapter 13) that subtracting a baseline reduces the variance of the estimates, thus speeding up the convergence to the optimal policy. To implement this with a neural network we can train the baseline network V_ϕ[9] to minimize the mean squared error between the network's value estimates for each visited state $V_\phi(s_t)$ and the returns obtained in the episode after having visited those states $G_t = \sum_{u=t}^{T} r_u$:

$$\text{Loss}(\phi) = \frac{1}{T}\sum_{t=0}^{T}(V_\phi(s_t) - G_t)^2 \quad (40)$$

Sometimes a factor $\frac{1}{2}$ is multiplied, so that the analytical gradient does not have a factor 2 in front, but it can be absorbed inside the learning rate factor and it's really just a different convention. In this case the policy gradient formula would be:

$$\nabla_\theta J^{\pi_\theta} = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - V_\phi(s_t))\right] \quad (41)$$

4.5 Actor-Critic methods

So far in the policy gradient methods we always relied on the computation of the full trajectory return to make our updates. This method of estimating the expected return is also called a Monte Carlo estimate, because it's an estimate based on sampling from probability distributions. However there are some drawbacks with this method:

1. Some tasks are open-ended and do not have a finite duration; it is clear that we cannot wait until the end of the task in order to estimate how good our actions were, but we would like to use some kind of incremental learning.

2. Even with finite tasks we might incur into a time truncation. In these cases the final state shouldn't be considered of value zero, since if it wasn't for the time truncation it would have yielded an expected reward equal to its state value. This has already been discussed in more detail in Section 2.1.

3. Using long trajectories within large state and action spaces to estimate the current state value will result in high-variance estimates.

Additionally, as we will see in future sections, all the most advanced algorithms use a fixed number of time steps to make their updates, so this would be similar to a self-imposed time truncation, in which for sure the state will not be terminal and it will definitely become necessary to estimate how much reward we would have obtained had the game continued until the end.

[9] From here on we will use V and Q instead of v and q to indicate that we are working with functional approximations, e.g. neural network outputs, of the state and state-action values.

Methods that estimate the episode return used in the policy gradient update with a function are called Actor-Critic methods, where the actor is the policy function that has to decide which actions to take and the critic is the action-value function that has to say how good the actions of the critic were. As we will see in Section 4.6, there is a substantial difference between using a function for the baseline as in Eq. 41 and using it to approximate the state-action value, because in the former case the ELPG lemma says that the baseline won't change the expected value of the gradient, whereas in the latter case a biased critic will bias such estimate too.

As we mentioned in Section 4.3, we can write the policy gradient update using the Q-values:

$$\nabla_\theta J^{\pi_\theta} = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)Q^{\pi_\theta}(s_t,a_t)\right] \tag{42}$$

What we would like to do as a form of incremental learning is to use just n steps to estimate part of $Q^{\pi_\theta}(s_t,a_t)$ and use past experience to estimate how much reward we can gain from the state s_{t+n} onward:

$$\begin{aligned} Q^{\pi_\theta}(s_t,a_t) &\equiv \mathop{\mathbb{E}}_{\tau_{[t:T]}\sim\pi_\theta}\left[\sum_{m=0}^{T-t}\gamma^m r_{t+m}\middle| s_t,a_t\right] \\ &= \mathop{\mathbb{E}}_{\tau_{[t:t+n-1]}\sim\pi_\theta}\left[\sum_{m=0}^{n-1}\gamma^m r_{t+m}\middle| s_t,a_t\right] + \mathop{\mathbb{E}}_{s_{t+n}}\left[\mathop{\mathbb{E}}_{\tau_{[t+n:T]}\sim\pi_\theta}\left[\sum_{m=n}^{T-t}\gamma^m r_{t+m}\middle| s_{t+n}\right]\right] \\ &= \mathop{\mathbb{E}}_{\tau_{[t:t+n-1]}\sim\pi_\theta}\left[\sum_{m=0}^{n-1}\gamma^m r_{t+m}\middle| s_t,a_t\right] + \gamma^n \mathop{\mathbb{E}}_{s_{t+n}}\left[V^{\pi_\theta}(s_{t+n})\right] \end{aligned} \tag{43}$$

We can say that this is an n-step look-ahead estimate of the Q-value $Q(s_t,a_t)$. What we are saying by that is that, if we can estimate V^{π_θ}, then, to compute the policy gradient, we just need n steps (with $n \geq 1$) instead of $T-t$ steps (where t is the current step and T the terminal step). However in the last section we were training our value network V using $G_t = \sum_{u=t}^{T} r_u$, which presents the same problem of using $T-t$ steps. But similarly to what we have just done in Eq. 43 we can estimate the rewards of the trajectory's tail with the value function of the state s_{t+n}:

$$V^{\pi_\theta}(s_t) = \mathop{\mathbb{E}}_{\tau_{[t:T]}\sim\pi_\theta}\left[\sum_{m=0}^{T-t}\gamma^m r_{t+m}\middle| s_t\right] = \mathop{\mathbb{E}}_{\tau_{[t:t+n-1]}\sim\pi_\theta}\left[\sum_{m=0}^{n-1}\gamma^m r_{t+m}\middle| s_t\right] + \gamma^n \mathop{\mathbb{E}}_{s_{t+n}}\left[V^{\pi_\theta}(s_{t+n})\right] \tag{44}$$

When it comes to estimating with n-step look-ahead the targets that the Q and V networks have to reproduce, we basically replace the expected values with sample averages, obtaining:

$$V^{\pi_\theta}(s_t) \approx \sum_{m=0}^{n-1}\gamma^m r_{t+m} + \gamma^n V(s_{t+n}) \tag{45}$$

and

$$Q^{\pi_\theta}(s_t,a_t) \approx \sum_{m=0}^{n-1}\gamma^m r_{t+m} + \gamma^n V(s_{t+n}) \tag{46}$$

where the only difference between the two formulas is that in the first case r_t is the expected value conditioned to s_t but not a_t, whereas in the second case is also conditioned to a_t. More concretely, since in practice every reward is obtained after an action has been chosen, the difference is that in the case of the state value V, all trajectories passing through s_t contribute to the estimate of $V(s_t)$, whereas only those that passed through s_t and selected a_t will contribute to the estimates of $Q(s_t,a_t)$.

Since both Q and V can be estimated by V and the rewards, we will train a single network for V and use it also to estimate Q when needed. This works because Q is only needed during

the update of the policy and we always require state-action values only of actions that have been sampled during that trajectory. If that wasn't the case it wouldn't be possible to estimate Q from V, because in general the reward $r(s_t, a_t)$ would be conditioned to an action a_t different than the one required. The loss function to be minimized at each step will be:

$$\text{Loss}(\phi) = \frac{1}{T}\sum_{t=0}^{T}\left[V_\phi(s_t) - \left(\sum_{m=0}^{n-1}\gamma^m r_{t+m} + \gamma^n \hat{V}_\phi(s_{t+n})\right)\right]^2 \tag{47}$$

where $\hat{V}_\phi$ is signalling that the gradient is not going to be back-propagated through that variable. This because empirically has been found that bootstrapping neural networks is more stable if the target is considered independent from the network's parameters (Mnih, Kavukcuoglu, et al. 2013). This method of training the actor-critic is also called n-steps temporal difference (TD) learning, because we train the critic to minimize the difference between two estimates of the same quantity done at different times (t and $t+n$).

Finally we can use the ELPG lemma and subtract the baseline value to the Q-value, obtaining what are called state-action advantages:

$$A^\pi(s_t, a_t) \equiv Q^\pi(s_t, a_t) - V^\pi(s_t) \tag{48}$$

The advantages are called in this way because they estimate how much more reward the agent is going to receive by choosing one action with respect to the reward that it would receive on average by following the current policy. The algorithms that use the advantage function in the policy gradient update are called Advantage Actor-Critics (A2C) and their update is:

$$\nabla_\theta J^{\pi_\theta} = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t) A^\pi(s_t, a_t)\right] \tag{49}$$

The actual formula to estimate the advantage with an n-step look-ahead method while using only a value network is

$$A^{\pi_\theta}(s_t, a_t) \approx \sum_{m=0}^{n-1}\gamma^m r_{t+m} + \gamma^n V^{\pi_\theta}(s_{t+n}) - V^{\pi_\theta}(s_t) \tag{50}$$

and the value (or critic) network is trained minimizing Eq. 47. In the box Algorithm 3 is reported a pseudo-code version of the A2C.

Algorithm 3 Single-environment A2C

procedure
  $\theta_1, \phi_1 \leftarrow$ randomly initialized
  Set γ and α
  for n=1, ..., N **do**
    $s_0 \sim \rho_0(s_0)$ ▷ sample initial state
    Produce trajectory $(s_0, a_0, r_0, \ldots, a_{T-1}, r_{T-1}, s_T)$ following π_{θ_n}
    Save $(\log \pi_{\theta_n}(a_0|s_0), \ldots, \log \pi_{\theta_n}(a_{T-1}|s_{T-1}))$
    Compute $A^{\pi_{\theta_n}}$ with Eq. 48
    Compute $g_\theta = \sum_{t=0}^{T-1}\nabla_\theta \log \pi_{\theta_n}(a_t|s_t) A^{\pi_{\theta_n}}(s_t, a_t)$ ▷ estimate gradient
    $\theta_{n+1} \leftarrow \theta_n + \alpha g_\theta$ ▷ update policy's parameters
    Compute Loss(ϕ) with Eq. 47
    $\phi_{n+1} \leftarrow \phi_n - \alpha\nabla_\phi \text{Loss}(\phi)$ ▷ update critic's parameters
    $\alpha \leftarrow C_\alpha \alpha$, with $C_\alpha \in (0, 1)$ ▷ decrease learning rate
  end for
  return $\pi_{\theta_N}, V_{\phi_N}$
end procedure

4.6 Variance and bias in Actor-Critic methods

We have seen that the main difference between the the Actor-Critic algorithm and REINFORCE algorithm (with or without baseline) is the use of bootstrapping to estimate the episode return, that will be used to compute the advantages (or the Q-values) and the target values for the critic.

This has two main consequences:

- Temporal difference methods have smaller variance in the target values' estimates than Monte Carlo estimates; intuitively this is because the value $V(s_t)$ is estimating the expected return of state s_t, whereas an MC estimate is based on a single sample from the ensemble of trajectories used to compute the expected return. $n-$steps TD method it's in the middle between the two extremes, with the estimates' variance that increases with n.
- TD methods also introduce a bias in the target values (and consequently on the critic's estimates), since V_ϕ^π is a parametric approximation of the on-policy value V^π. This bias is not present in MC method, because it's estimating through sample average the episode return. Also it's important to notice that the bias in the value will not affect the result as long as it's used as a baseline, because of the EGLP lemma (see Eq. 34); this is why the REINFORCE with baseline is not considered an Actor-Critic by Stutton & Barto (see Sutton and Barto 1998, chapter 13).

How to trade-off between variance and bias in Actor-Critic methods is still an active field of research. One interesting analysis of this topic can be found in Schulman, Moritz, et al. 2015, where the authors also present a novel method, called Generalized Advantage Estimation, to trade-off between variance and bias through an exponential average of the n-steps advantages of an action-state pair for all possible n-steps. In this way they can avoid to choose a specific n-step, which can be highly dependent on the time scale of the task considered.

4.7 Actor-Critic instability

When using a neural network to parametrize a function, the learning process can be unstable, due to the fact that the representation mechanism is not local, but global: a weight change induced by an update in a certain part of the state space might influence the values in arbitrary other regions - and therefore destroy the effort done so far in those regions. Moreover, in contrast to typical NN training cycle, where the dataset is stationary, in RL tasks the data we use to train the agent depends on the current learned policy, thus it will change in time as the agent learns, putting more emphasis on different state space regions at different times. In the extreme case of updating the agent after a single trajectory, we can imagine that the new policy will be optimized especially for that trajectory, risking to over-fit on it and consequently having worse generalization properties on the other possible trajectories[10]. This leads to typically very long learning times or even to the final failure of learning. On the other hand, we are forced to use approximation methods because we need to generalize our predictions to previously unseen situations, given the fact that using a tabular setting in which all state values are independent from each other would require an amount of experience impossible to gather for the most complex tasks. Therefore the question is: how can we constrain the negative influence of a new update of the value function in a neural network?

In the literature three kind of solutions have been found: the first applies only to off-policy algorithms and goes under the name of "experience replay" (Lin 1993), where updates on new trajectories are interspersed with updates on past experience, in order to increase the retain of strategies learned in the past (see Algorithm 4 for an implementation of Experience Replay in Deep Q-Learning algorithm from Mnih, Kavukcuoglu, et al. 2013). The second one is based on algorithmic improvements, in order to reduce the instability of the actor and/or the critic by introducing safety measures in their updates. The last one can be applied both to on-policy and

[10] In fact all neural networks are trained on some version of the Stochastic Gradient Descent, which is based on the fact that all samples are independently and identically distributed. The independence of samples in on-policy reinforcement learning is strongly violated, since subsequent states are obviously correlated.

off-policy learning and is based on learning in parallel from multiple copies of the environment (Mnih, Badia, et al. 2016), hence increasing the portion of the state space on which the NN is being fitted at every update and reducing the overfitting problem.

Experience replay cannot be applied to actor-critic methods as they are because they not only require the knowledge of the state-action-reward trajectory, but also of the probability with which each action has been selected by the policy used during that trajectory. Since past experience refers to past policies, optimizing with the actor-critic equations on past experience would be like trying to update the parameters of the current policy as if they were that of a past policy, which is totally incorrect. It is possible to make the actor-critic algorithm an off-policy one, by means for example of an importance sampling correction, but for standard AC methods experience replay does not work.

Algorithm 4 Deep Q-Learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
$\theta \leftarrow$ randomly initialized
for episode=1, ..., M **do**
  $s_0 \sim \rho_0(s_0)$
  for t=1, ..., T **do**
    $a_t \sim \pi_\epsilon(a_t|s_t; Q_\theta)$
    $s_{t+1}, r_t \sim p(s_{t+1}, r_t|s_t, a_t)$
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of B transitions $\{(s_j, a_j, r_j, s_{j+1})\}_{j=1,...,B}$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_a Q_\theta(s_{j+1}, a) & \text{for non-terminal } s_{j+1} \end{cases}$
    Compute $\text{Loss}(\theta) = \frac{1}{B} \sum_{j=1}^{B} (y_j - Q_\theta(s_j, a_j))^2$
    $\theta \leftarrow \theta - \alpha \nabla_\theta \text{Loss}(\theta)$
  end for
end for
return Q_θ

Some algorithmic improvements on the critic update are:

- The introduction of a copy of the critic's network to compute its target values in the formula Eq. 47; this network is then slowly updated, e.g. $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$, where θ' are the critic's target parameters, θ the critic's parameters and $\tau \in (0, 1)$ regulates how fast the target network is updated, so that the target network is always "conservative" in its estimates. This is found to have a stabilizing effect on the learning (Lillicrap et al. 2015).
- Taking the minimum value between a pair of critics to limit over-estimation of the values. Again using a conservative estimate (now in value instead than in time) helps to increase the stability of the actor-critic (Fujimoto, Van Hoof, and Meger 2018).

On the actor side other notable improvements were:

- Constraining the policy updates to remain inside a trusted region, formalized by a constraint on the Kullback-Leibler divergence between the old and the new policy (Schulman, Levine, et al. 2015). This insures that small changes in the parameter space are not translated in huge and risky changes in the policy distribution.
- Using "soft" Q-values, which include a bonus for increasing the policy entropy, in order to introduce an inherent exploration incentive in the actor target. Exploration greatly helps the actor stability, avoiding the convergence to sub-optimal deterministic policies (Haarnoja et al. 2018).

Some other proposals were even targeted to reduce the instability that arises from the interaction between the actor and the critic, for example reducing the magnitude of actor's updates when the critic's predictions contain big errors (Parisi et al. 2019).

In general algorithmic improvements can be stacked on top of both the experience replay idea (when the algorithms are off-policy, like Lillicrap et al. 2015 and Haarnoja et al. 2018) and the parallel learning.

Regarding parallel learning, this is similar as an idea to waiting for multiple episodes to finish before updating the policy, since in both cases we are increasing the variety of states and trajectories on which we are optimizing. However using parallel environments is superior for at least three reasons on using multiple episodes from a single environment:

- Parallel environments as the name says can be run in parallel, exploiting multiple CPU's cores at the same time and being many times more efficient than a single environment in producing experience.
- Neural networks are optimized to work on "batched" data, i.e. performing vectorial operations, especially when running on a GPU. Thus deciding which action to take on N different states all at once is more efficient than deciding the same thing one state at the time.
- In a single environment, to gather experience from multiple episodes, we have to interact with them sequentially, finishing at least all but the last before updating. This can be very inconvenient for long episodes, since we don't have any control over the total number of steps that we have to wait before updating. In the case of parallel environments we can update after every environment has produced T steps, without waiting for the episodes to finish, and then use a temporal difference update with all its advantages.
- Subsequent states of a trajectory are strongly correlated, making neural networks training unstable. Using samples from different episodes reduces their correlation and yields a less noisy gradient estimate.

In general all Deep RL algorithms can benefit from using batched experience, which can be easily generated in parallel, thus virtually all these algorithms use parallel environments for learning. Parallel learning is also incredibly scalable: the most massive experiment in RL we know of was performed by OpenAI to solve the video game of Dota2 and made use of 51.200 CPUs and 1024 GPUs to process as much information as possible in the unit of time (Berner et al. 2019).

In our work we focused on parallel learning because is the only solution that increases not only the stability and performance of the Actor-Critic, but also its speed of training, which is another main limiting factor in RL applications.

4.8 Parallel Actor-Critics

There are two implementations of the Advantage Actor-Critic with parallel environments: the asynchronous version (A3C) and the synchronous one (A2C).

The A3C (Mnih, Badia, et al. 2016) is the first one that has been proposed and the idea is to have multiple copies both of the agent (also called workers) and the environment, each pair running on a CPU's core. Each worker interacts with the environment to produce a trajectory of states, actions and rewards. The decision making (forward operation in a neural network) is executed on a single state each time and runs on the CPU's core. After the trajectory (or a certain number of steps T) is obtained, the gradients of the parameters are computed, then communicated to a global copy of the network and the optimization step is applied there (stochastic gradient descent in the easiest case). Finally the workers fetch back the updated parameters and repeat the process.

Two important details have to be highlighted: first, the gradients are computed on the local parameters that might not be up-to-date with the global network, but are applied to that

network. Second, the updates are executed asynchronously by the different workers. These two characteristics introduce some noise in the updates, which is claimed to have a stabilizing effect.

Overall, the main points in favour to the A3C are that workers have a small waiting time (just the time of communicating the gradients, applying them and receiving the new parameters) and no specialized hardware (GPU) is needed, whereas the problem is that with big neural networks the backward operation to compute the gradients is extremely inefficient on a CPU. In fact an implementation of a hybrid CPU/GPU Asynchronous Advantage Actor Critic called GA3C (Babaeizadeh et al. 2016) showed how using a GPU both in forward and backward operations resulted in speed-up factor of x6 in case of a small network and up to x45 in case of a large one.

The synchronous version of the Advantage Actor-Critic has a single agent model running (possibly) on a GPU and many copies of the environment on the various CPU's cores. At each interaction of the agent with the environments, these send the current states and the rewards obtained from the previous step to the agent; the states are batched together and the agent decides, based on them, a batch of actions, that are then dispatched one for each environment, where are executed and a new interaction can begin. After T interactions have been made, the trajectories and the log-probs of all the actions taken are batched together and a unique update is made on the global model.

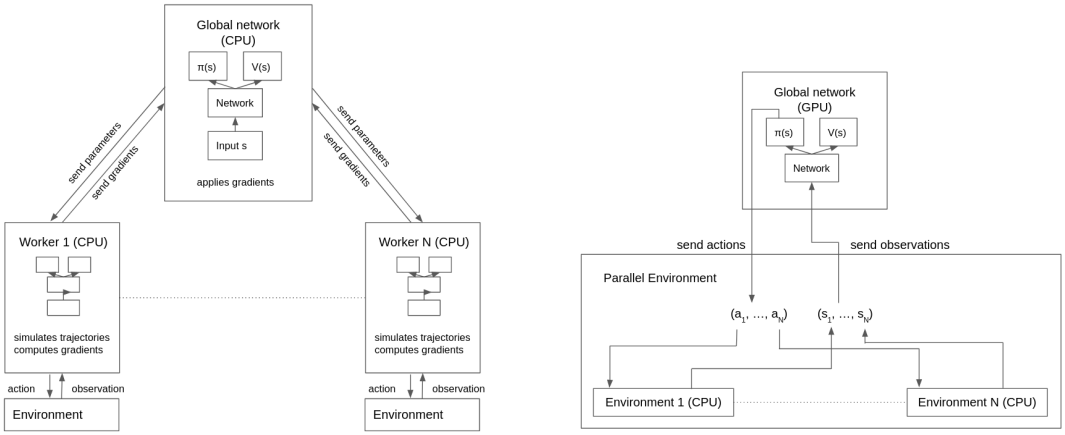
It's important to notice that both the forward and the backward operations are executed in a batched fashion and (possibly) on a GPU, potentially being many times faster than the ones executed by the A3C. However a factor that slows down the A2C significantly is that when the agent is gathering the states, different environments may require different times to send the states. This is because before sending the new state they have to compute it from the previous state and action and the rendering times might vary significantly from case to case in certain environments. Also sometimes episodes can end at different times and the environment has to be reset to initial condition before proceeding with a new episode; this of course requires additional time.

A comparison between the two algorithms has been done by a team at OpenAI, a company leading the research efforts in the RL field. Their conclusion was:

"Our synchronous A2C implementation performs better than our asynchronous implementations — we have not seen any evidence that the noise introduced by asynchrony provides any performance benefit. This A2C implementation is more cost-effective than A3C when using single-GPU machines, and is faster than a CPU-only A3C implementation when using larger policies." - OpenAI blog, Wu et al. 2017

In our work we implemented both algorithms[11] and even though no thorough comparison was made, it was clear that the A2C implementation was faster when having at disposal the same hardware, which included a GPU device. Consequently that's the version that we used in the StarCraft II environments' suite. Both architectures' schemes are reported in Figure 5.

[11] A3C algorithm was implemented only on a simple grid world environment, whereas A2C was implemented there and also adapted for the StarCraft II Learning Environment. Our observations are based on the comparison on these grid world tests.



(a) A3C architecture.

(b) A2C architecture.

Figure 5: Architecture schemes for the A3C (a) and A2C (b). A3C distributes the model on multiple cores and updates a global network asynchronously, whereas A2C works on a single device (possibly a GPU) and interacts with parallel environments in a vectorial fashion.

4.9 Exploration in Policy Gradient methods

As we mentioned in Section 3.2 to solve a RL task is fundamental to find a balance between the exploitation of working strategies and exploration of new ones. Although in Policy Gradient and Actor-Critic methods we parameterize a stochastic policy, which is highly random at its initialization, hence granting an initial exploration, it is often the case that simple strategies that represent local optima are reinforced before the more complicated ones can even be tried, effectively ending the exploration too early. A solution with which researchers came up with is to use the so-called *entropy regularization*. The idea is to subtract to the actor loss $-J$ the average entropy of the policy at each step of the trajectory, weighted by a constant h. In this way, when we minimize our objective, we will favour policies with higher entropy, hence we will slightly favour exploration for the whole duration of the training process. This technique was originally proposed by Williams and Peng 1991.

This regularization technique works for all algorithms that are derived from the basic Policy Gradient algorithm (Eq. 33). In particular for the A2C the regularized actor gradient is:

$$\nabla_\theta J^{\pi_\theta} = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)A^\pi(s_t,a_t) + h\frac{1}{T}\sum_{t=0}^{T}\nabla_\theta H[\pi_\theta(\cdot|s_t)]\right] \tag{51}$$

where $H[\pi_\theta(\cdot|s_t)]$ is the entropy of the policy in state s_t:

$$H[\pi_\theta(\cdot|s_t)] = \mathop{\mathbb{E}}_{a\sim\pi_\theta(\cdot|s_t)}[-\log \pi_\theta(a|s_t)] = \sum_{a\in A} -\pi_\theta(a|s_t)\log \pi_\theta(a|s_t) \tag{52}$$

There is also a more refined theory called Maximum Entropy (MaxEnt) RL (Levine 2018 and Schulman, Abbeel, and Chen 2017), in which the objective of the agent is to maximize the expected *entropy-augmented return*:

$$G_t = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t(r_t - h\mathrm{KL}_t)\right] \tag{53}$$

where KL_t is the Kullback-Leibler divergence between the current policy $\pi(\cdot|s_t)$ and a reference policy (that can be for example a random policy) $\overline{\pi}(\cdot|s_t)$. The main difference is that while in the entropy regularization at each step t we are maximizing the discounted return G_t together with a fraction of the current entropy H_t, in the MaxEnt RL framework we take into account also the

entropy of the policy in the future states; this makes sense in the optic in which, if we can decide between two states with the almost the same value, we should favour the one that gives us more choices in the future, since more stochastic policies are less prone to get stuck in local optima.

Although this last theoretical framework is ultimately useful to draw a connection between value-based methods like Q-learning and the actor-critic methods, its increase in complexity is not matched by an increase in performance, thus in our implementations we used the simpler entropy regularization.

5 IMPALA

In this section we are going to introduce a more advanced RL algorithm, called Importance Weighted Actor Learner Architecture (IMPALA, Espeholt et al. 2018), that stems from the family of the Actor-Critic methods.

5.1 Factors that influence throughput

Current Reinforcement Learning algorithms require huge amounts of experience in order to solve complex tasks. When we are on a time budget but have at disposal unlimited interactions with the environment during that time (e.g. when the environment is virtual and fast to interact with), to reduce the time it takes to solve a task we can either make the algorithms more sample-efficient, i.e. trainable with less experience, or make them faster, so that they can produce and consume more experience in the same amount of time. A measure of how fast an algorithm is is the throughput, that is the number of agent-environment interactions that it can process in the unit of time.

The goal of IMPALA is to achieve the highest throughput possible during the agent's training. This in general will depend on many factors, most notably:

1. **Number of parameters of the agent's neural network**: even though the speed of computing which action to take and the state's value might not be predicted just by knowing the number of parameters, it is true that if we keep the same architecture (number and type of layers), but increase their hidden units (or convolutional channels) more computations will need to be done and so the time for the forward and backward operations is going to increase.

2. **Speed of the environment step**: the environment engine has to read and apply the action chosen by the agent, compute the new state and render it, for example in raw pixels or feature planes. This step in most cases is executed on a CPU core and there isn't much that we can do to speed it up.

3. **Device in which forward and backward operations are computed**: usually the backward operation is the most expensive operation and speed-up factors in executing it on a GPU instead than on a CPU core are of order 10. For forward operation the speed-up factor might be less because the overhead of loading variables on the GPU is greater in proportion when the operation is faster, but there is always a gain to be made. Ideally any high-performance algorithm should execute at least the backward operation on a GPU.

4. **Waiting times**: in general we have 3 computational blocks, the environment step and the agent's forward and backward operations. In a single environment scenario we repeat for T times the environment step and the agent's forward and then we execute the backward operation once. We can call this whole cycle an epoch. What we can observe is that the environment step and the agent's forward have to be executed sequentially (even though not necessarily on the same device, e.g. environment step on CPU and agent's forward on GPU). Furthermore the backward operation is dependent on the results of T sequential interactions between the environment and the agent, so that will be its waiting time. This differs from the typical supervised learning training process, in which every forward operation is executed on a batch of samples and immediately followed by a backward operation. When we use multiple environments, an additional factor is added, and that is the fact that at certain points we might want to batch together some results before processing them. Thus we will need to wait for all the processes to finish in order to gather the results: this is always source of delays, that will be higher if there is high variability in the processing time of the various processes and also it will increase with the number of processes, since the probability of getting at least one slow outlier will increase with them.

5. **Period T of the updates**: in this context also called *unroll length*, is the number of agent-environment steps after which we execute an optimization step (backward operation to compute the gradients of the network's parameters, followed by the step of the optimizer

to change the network's parameters accordingly to the gradients). Since this operation is the most expensive, higher unroll lengths imply lower update frequencies and result in a higher throughput. However updates with higher frequency could be more sample-efficient, i.e. use less samples to achieve the same performance on the long run. Also longer update periods result in higher RAM requirements, which can be problematic if we consider big network architectures and batch sizes (usually equal or proportional to the number of parallel environments that we are using).

5.2 Actor Learner Architecture

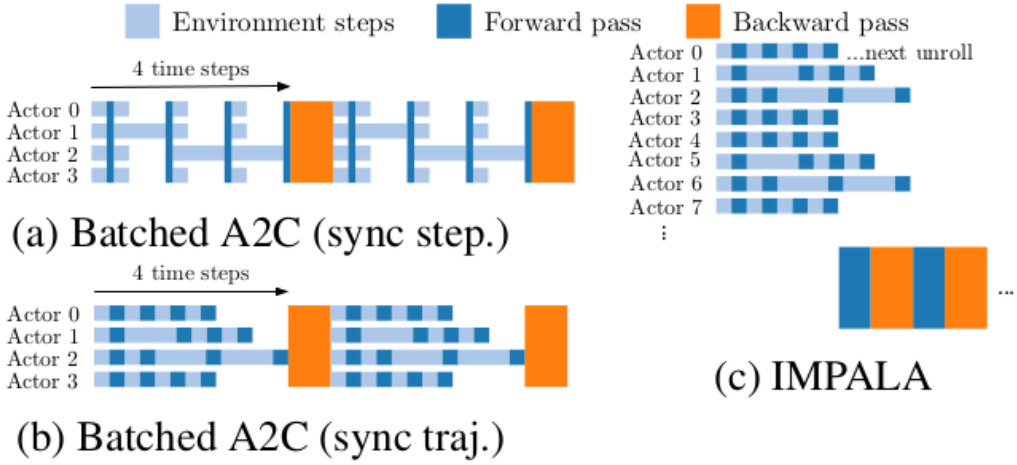


Figure 6: Timeline for one unroll with 4 steps using different architectures. Strategies shown in (a) and (b) can lead to low GPU utilisation due to rendering time variance within a batch. In (a), the actors are synchronised after every step. In (b) after every n steps. IMPALA (c) decouples acting from learning. Source: IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures, Espeholt et al. 2018

The idea behind IMPALA is to decouple the forward and the backward operations, as well as decoupling all parallel environment processes, while still executing the backward operation on a GPU device. This will reduce to zero two waiting times present in the A2C algorithm (see Figure 6): the time parallel processes have to wait each other when we synchronize their results and the time of the backward operation, during which usually neither the forward operation nor the environment step can be executed. To do this we introduce two kind of agents: actors and learners (we will use just one of them).

Actors have to interact with a local copy of the environment to produce trajectories of states, actions and rewards (plus the usual additional signals, like the end of episode one); they will also store at each step the probability of choosing the action that has been sampled and the entropy of the policy. When they finish producing these variables, they write them in an buffer at a specific index (can be thought as a simple memory address) that is shared across processes so that the learner can access it. Once done that, they can continue producing another trajectory, without waiting neither the other processes to finish nor the learner to make the update. Typically the actors' processes run on CPU's cores.

The learner instead will wait until a certain number of buffer's indexes will be full and then it will extract the trajectories inside them, batch them together and perform an optimization step; after that it will send the updated parameters to the actors. Here is important to notice that there will be part of the new trajectories that will be produced following the old policy and another part that will follow the updated one. Also notice that the learner has to wait for all processes to write a trajectory in the buffer before computing the update, but no other process has to wait for the

learner, hence as long as the optimization step is faster than T agent-environment interactions, it will not slow down the training process at all.

Comparing this architecture with A3C instead we can notice that:

1. The agent-environment interactions are produced in similar ways: in both algorithms there are parallel processes running on CPU cores, each with its own copy of the actor, producing experience independently and communicating with a central model (global network in the case of A3C, learner in the case of IMPALA) asynchronously.

2. In A3C the actors communicate gradients (computed on local trajectories) to the central model; in IMPALA they communicate trajectories of experience, which then will be batched by the central learner and used for an update.

3. Actor processes in A3C are slowed down by the computations of the backward operation on the CPU core, whereas in IMPALA they just have to produce experience; as a result the latter are more efficient in doing that.

4. In A3C there is no theoretical correction to take into account the lag between the local policy and the global one, whereas in IMPALA the V-trace correction does that, providing stability to the learning process.

A bridge between the two architectures is the GA3C model (Babaeizadeh et al. 2016, already mentioned in Section 4.8), which used a dynamic queuing system similar to IMPALA to run both forward and backward operations on a single model loaded on the GPU (thus similarly to A2C). Its main components are:

- **Agent**: local agent that interacts with its copy of the environment on a CPU core; queues policy requests in a Prediction Queue before each action, which then it's executed on the environment, in order to create a trajectory. Periodically submits a batch of input and reward experiences to a Training Queue.

- **Predictor**: dequeues as many prediction requests as are immediately available, batches them into a single inference query to the model loaded on the GPU and then returns back the selected actions along with their probabilities to the each agent. Many predictors can be run concurrently to hide latency.

- **Trainer**: dequeues training batches submitted by agents, then submits them to the GPU in order to compute the update. Many trainers can be run concurrently to hide latency.

While this model runs also inference (forward operation) on the GPU on a batched fashion, it fails to take into account the lag that exists between the model used to predict the actions and the one using the experience produced by those actions to compute the Advantage Actor-Critic update (since one trainer might execute other updates in the meanwhile, changing the model parameters). IMPALA was developed after GA3C and solved the issue with the V-trace algorithm; furthermore it can also use the GA3C setting with the Prediction Queue and a single central model to make use of the GPU in the inference mode too[12].

5.3 Off-policy correction: V-trace and Importance Sampling

So far we have seen only the improvements of IMPALA over the A2C's training cycle, but we also mentioned that the learner will receive trajectories that, at least in part, are produced by policies with different parameters of its own ones, which are more up to date in general. We have also seen that A2C is an on-policy RL algorithm, meaning that we cannot use data produced by old (or different) policies in the updates of the current one, thus for IMPALA to use off-policy trajectories we need to define an off-policy correction that goes under the name of V-trace.

Adopting the notation from Espeholt et al. 2018, we will denote:

[12]The IMPALA implementation from which we adapted our StarCraft agent had a single-machine version which didn't implement the GPU inference and a more complicated multi-machine version which did it. For time constraints we only used the first one and described that version of the IMPALA architecture in the text.

- π: learner's policy.
- μ: actors' policy.
- $(x_t, a_t, r_t)_{t=s}^{s+n}$: trajectory sampled following policy μ.
- $V(x_s)$: value of state x_s following policy π (so this needs an off-policy target to be trained).
- v_s : off-policy target for $V(x_s)$ computed through V-trace (see below).

We then define the n-step V-trace target for $V(x_s)$ as:

$$v_s \equiv V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \left(\prod_{i=s}^{t-1} c_i \right) \delta_t V \tag{54}$$

where

$$\delta_t V \equiv \rho_t (r_t + \gamma V(x_{t+1}) - V(x_t)) \tag{55}$$

$$\rho_t \equiv \min \left(\overline{\rho}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)} \right) \quad , \quad \overline{\rho} \in \mathbb{R}_+ \tag{56}$$

$$c_t \equiv \min \left(\overline{c}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)} \right) \quad , \quad \overline{c} \in \mathbb{R}_+ \tag{57}$$

So basically $\delta_t V$ is an estimate of the expected 1-step value difference according to policy π (if we forget for an instant about the possible truncation $\overline{\rho}$): $V(x_t)$ refers to the value of state x_t according to π, but we sample a_t that determines r_t according to μ, so we correct the expectation value with the factor ρ. In other words:

$$\begin{aligned} \mathbb{E}_\pi[\delta_t V] &= \sum_{a_t \in A} \pi(a_t|x_t) \sum_{x_{t+1}, r_t} p(x_{t+1}, r_t|x_t, a_t)(r_t + \gamma V(x_{t+1}) - V(x_t)) \\ &= \sum_{a_t \in A} \mu(a_t|x_t) \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)} \sum_{x_{t+1}, r_t} p(x_{t+1}, r_t|x_t, a_t)(r_t + \gamma V(x_{t+1}) - V(x_t)) \\ &= \mathbb{E}_\mu[\rho_t (R_t + \gamma V(X_{t+1}) - V(x_t))] \end{aligned} \tag{58}$$

The last expression is actually what we estimate by sampling, since all our trajectories are produced following policy μ. Looking at Eq. 54, the other term that we have to understand is the product of the c_i: that is a truncated importance sampling (IS) factor, telling us what is the likelihood ratio in choosing the sequence of actions $(a_s, \ldots, a_{t-1})$ that lead to state x_t from policy π instead that from policy μ.

To sum up we have these two off-policy corrections in the V-trace formula, one to account for the fact that $\delta_t V$ is using the rewards sampled following μ to estimate the future value of the state following policy π; the other to account that the trajectories sampled following μ would have been more or less probable if we had chosen them according to π. In both cases, if we neglect the truncation factors $\overline{\rho}$ and $\overline{c}$, the IS factors allow us to correctly compute expectations with respect to policy π by sampling with policy μ.

One can give precise theoretical explanations of how $\overline{\rho}$ and $\overline{c}$ affect the results of the learning process (Espeholt et al. 2018), but in practice we can simply consider them as a variance-reduction technique, because they are reducing the impact that rare choices from policy μ have on the estimates of policy π. In practice during the experiments that we performed both $\overline{\rho}$ and $\overline{c}$ are set to 1. This is done also in Retrace (Munos et al. 2016), which is the algorithm that first introduced the truncated importance sampling[13].

[13]The name V-trace was in fact derived from it, since the only difference is that requires learning state value functions V instead of action-state value functions Q in order to make the off-policy correction.

A useful manipulation of Eq. 54 yields a recursive formula to estimate v_s, which we can use in practice to reduce the number of computations needed to compute all v_s.

$$\begin{aligned} v_s &\equiv V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} \left(\prod_{i=s}^{t-1} c_i \right) \delta_t V \\ &= V(x_s) + \gamma^0 \cdot 1 \cdot \delta_s V + \sum_{t=s+1}^{s+n-1} \gamma^{t-s} \left(\prod_{i=s}^{t-1} c_i \right) \delta_t V \\ &= V(x_s) + \delta_s V + \gamma c_s \sum_{t=s+1}^{s+n-1} \gamma^{t-(s+1)} \left(\prod_{i=s+1}^{t-1} c_i \right) \delta_t V \\ &= V(x_s) + \delta_s V + \gamma c_s (v_{s+1} - V(x_{s+1})) \end{aligned} \tag{59}$$

This formula is saying that the target value of state x_s under the learner's policy π is equal to the critic's estimate of the state value $V(x_s)$ plus the temporal difference correction $\delta_s V$ for state x_t and the error committed by the critic on state x_{t+1}, discounted by γ and weighted with the IS factor c_s. One small detail in the derivation is the fact that the substitution in the last passage is using an $n-1$-steps V-trace definition, instead of an n-steps. In practice though, if we take n as large as possible, there will be a truncation point after which we just make use of the same bootstrapping in both cases, so there won't be any difference between $n-1$-steps and n-steps.

5.4 Off-policy advantage and policy gradient

The state-action advantage is usually defined as the difference between the expected return taking a certain action in a certain state and the expected return for just being in that state:

$$\begin{aligned} A^\pi(x_t, a_t) &= \mathbb{E}_\pi \left[\sum_{s=t}^{T} \gamma^{s-t} r_s | X_t = x_t, A_t = a_t \right] - \mathbb{E}_\pi \left[\sum_{s=t}^{T} \gamma^{s-t} r_s | X_t = x_t \right] \\ &= \mathop{\mathbb{E}}_{r_t \sim p_r(\cdot|x_t, a_t)} [r_t | X_t = x_t, A_t = a_t] + \gamma \mathop{\mathbb{E}}_{x_{t+1} \sim p_x(\cdot|x_t, a_t)} \left[\mathbb{E}_\pi \left[\sum_{s=t+1}^{T} \gamma^{s-t} r_s | X_{t+1} = x_{t+1} \right] \right] \\ &- \mathbb{E}_\pi \left[\sum_{s=t}^{T} \gamma^{s-t} r_t | X_t = x_t \right] \\ &= \mathop{\mathbb{E}}_{r_t \sim p_r(\cdot|x_t, a_t)} [r_t | X_t = x_t, A_t = a_t] + \gamma \mathop{\mathbb{E}}_{x_{t+1} \sim p_x(\cdot|x_t, a_t)} [V^\pi(x_{t+1})] - V^\pi(x_t) \end{aligned} \tag{60}$$

Taking away the expectation values from the last line (because they do not depend on the policy, but only on the environment dynamics), we can estimate the advantage function like this[14]:

$$A^\pi(x_t, a_t) = r_t + \gamma V^\pi(x_{t+1}) - V^\pi(x_t) \tag{61}$$

Furthermore any estimate of $V^\pi(x_{t+1})$ can be used; a common one is the target value for state x_{t+1}, that is also used to train the critic network. In the off-policy case we can use v_{t+1} instead of $V(x_{t+1})$ and in the appendix of the IMPALA paper they show that experimentally this leads to better results, so they use this variant of the advantage also in their code implementation. From here we usually estimate the policy gradient loss, that differentiated with respect to the policy's parameters will yield minus the policy gradient. The expected loss in the on-policy case is the following:

$$\text{Loss} = -\sum_{t=0}^{T} \mathbb{E}_\pi [\log \pi(a_t | x_t) A^\pi(x_t, a_t)] \tag{62}$$

[14] As long as the sampling of x_{t+1} is conditioned on (x_t, a_t)

We can compute this in the off-policy case using the importance sampling factors ρ_t computed in Eq. 56:

$$\text{Loss} = -\sum_{t=0}^{T} \mathbb{E}_{\mu}[\rho_t \log \pi(a_t|x_t) A^{\pi}(x_t, a_t)] \tag{63}$$

5.5 IMPALA implementation details

Our implementation of IMPALA algorithm for the StarCraft II environment is an adaptation[15] of a PyTorch version of IMPALA for Atari games developed by Facebook Research team (Kuttler et al. 2019).

The workflow of the algorithm is the following:

1. Set input parameters `n_buffers`, `batch_size` and `n_actors`, with `n_buffers` $\geq$ `batch_size` and `n_buffers` $\geq$ `n_actors`, e.g. 24, 12, 12.

2. Create `n_buffers` buffers. Each buffer is a dictionary, whose keys are the names of the variables needed for the learner update. Every value consists of a list of `n_buffers` arrays of shape (T+1, `variable_shape`), where T is called unroll length and is the number of time-steps simulated by the actor and the environment used in the update.

3. Two queues are shared between all actors' processes and the learner thread: both can contain values ranging between 0 and n_buffers - 1, but from one we can write and read the indexes of the free buffers, whereas from the other one we can write and read the indexes of the full buffers.

4. At the beginning of the training all buffers' indexes are inserted in the free queue, then each of the actors' processes fetches one of the free indexes (removing it from the free queue), simulates a trajectory with their local copy of the environment and writes all variables for all T time-steps in the buffer whose index they received from the free queue. After that, they insert the buffer's index into the full queue.

5. The learner thread retrieves from the full queue the indexes of n_buffers, reads the buffers at those addresses and batches together each variable. After retrieving the batch, the learner thread immediately frees the indexes that it just read by inserting them in the free queue (they have already been removed from the full queue when the learner read them).

6. Once obtained a batch, the V-trace algorithm is used to compute the off-policy update for the learner actor-critic (the main point being weighting the actors' experience based on how much their policy reflects the learner's current policy, e.g. improbable actions do not carry the same weight of more "shareable" actions).

7. Finally the policy gradient, critic and entropy losses are computed, an optimization step is taken on the learner's parameters and then those parameters are copied on the actor's model (which is a single model with shared memory for the parameters).

A pseudo-code version of the actor and learner's processes can be found in Algorithm 5. Notice that to compute the advantages Eq. 61 has been modified to use the target values v_{j+1} instead of the critic's predictions $V_\theta(x_{j+1})$, as mentioned in that section, and that to the usual actor loss is subtracted the average entropy of the policy weighted by a factor h (usually taken in the range $(10^{-5}, 10^{-2})$). The last detail to which has to be paid attention is the fact that we are assuming that the actor and the critic networks partially share the same parameters, thus are optimized together.

[15] We maintained all the core structures (e.g. actor and learner threads, buffers, and V-trace algorithm), but we had to moderately modify each of the functions to adapt the algorithm for the StarCraft II Learning Environment. In particular actor and learner architectures and functionalities are coded completely from scratch, V-trace had to be slightly modified to implement our bootstrapping procedure and a wrapper of the StarCraft environment was coded from scratch too. Finally the buffers were modified to communicate efficiently the compound actions of StarCraft with their arguments. Code can be found at: https://github.com/nicoladainese96/SC2-RL.

However the gradient of the weights is not computed with respect to the advantages present in Eq. 65, because in the original derivation is assumed that the critic's estimate are independent from the policy's parameters. Both implementation choices were introduced in Mnih, Badia, et al. 2016 and became widely adopted in Actor-Critic methods.

5.6 Bootstrapping with V-trace

We have seen in Section 2.1 how to virtually extend the horizon of a task which is normally time-truncated.

In practice we use bootstrapping to compute the target value for the critic in the on-policy setting as follows:

$$\begin{aligned} V(x_t) &= r_t + \gamma \neg done_{t+1} V(x_{t+1}^{trg}) + \gamma bootstrap_{t+1} V(x_{t+1}^{trg}) \\ &= r_t + \gamma(\neg done_{t+1} + bootstrap_{t+1}) V(x_{t+1}^{trg}) \end{aligned} \tag{66}$$

where `done_t` is a boolean signal saying (if true) that the episode was ended at step t and x_t is the initial state of a new episode, which is totally unrelated to the previous state x_{t-1}, whereas `bootstrap_t` is another boolean signal saying (if true) that the episode was ended at step t by time truncation, thus we should bootstrap. So basically we are using an OR logical operation between `not done` and `bootstrap`. Another thing that we need to pay attention to is which state to use for bootstrapping: in fact, if the bootstrap signal at time t is true, then we first reached a state x_{t+1}^{trg} and afterwards the environment was reset and we obtained x_{t+1}. All of this is used to compute the $\delta_t V$ in equation Eq. 55 in the correct way:

$$\delta_t V = \rho_t [r_t + \gamma(\neg done_{t+1} + bootstrap_{t+1}) V(x_{t+1}^{trg}) - V(x_t)] \tag{67}$$

Using IMPALA without bootstrapping, a problem emerged with StarCraft mini-games: there were periodic peaks in the actor and critic's losses, whose period was around the same length of the unroll length T (see Figure 7a).

Looking more into depth at the problem we realised that this phenomenon was connected with the `done` signal given by the environment, which wasn't differentiating between natural end on an episode and time truncation. The problem with time truncation is that, if we consider the last state of the trajectory as terminal but we don't provide time as a variable in the state representation, is virtually impossible to discriminate between a terminal state and the same state at a different time (say $t = 0$). So the `done` signal is saying that the value of the last state is just equal to the expected reward obtained in that step, but we know that the next state x_{t+1}^{trg} will not be terminal in itself, but just because of time truncation, so in general it's value will be different from zero.

Implementing IMPALA with the bootstrapping solved completely the problem, as can be seen in Figure 7b.

Algorithm 5 IMPALA

procedure ACT
  Receive initialized actor network μ_θ
  $x_1 \sim \rho_0(x_1)$
  Get actor output $(a_1, \log \mu_\theta(a_1|x_1))$
  while True **do**
    Get buffer `index` from `free_queue`
    for t=1, ..., T **do**
      $x_{t+1}, r_t \sim p(x_{t+1}, r_t|x_t, a_t)$
      Get actor output $(a_{t+1}, \log \mu_\theta(a_{t+1}|x_{t+1}))$
      Write $(x_{t+1}, r_t, a_{t+1}, \log \mu_\theta(a_{t+1}|x_{t+1})$ at position t of buffer `index`
    end for
    Put `index` in the `full_queue`
    $x_1, a_1 \leftarrow x_T, a_T$ ▷ use last state and action to start the new trajectory
  end while
end procedure

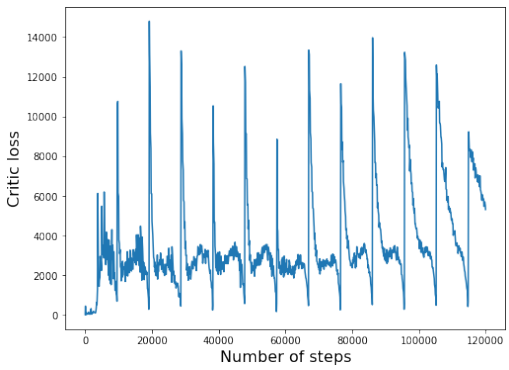
procedure LEARN
  Receive initialized learner network (π_θ, V_θ) ▷ partially shared parameters
  while `steps` < `total_steps` **do**
    Get B `indexes` $\mathcal{I} = \{i_1, \ldots, i_B\}$ from `full_queue`
    Get batch $\{(x^i_j, r^i_{j-1}, a^i_j, \log \mu_{\theta'}(a^i_j|x^i_j)\}_{j=1,\ldots,B}$ ▷ θ' can be out-of-date w.r.t. θ
    Put `indexes` $\mathcal{I}$ in `free_queue` ▷ actors' processes can proceed from now
    Compute learner off-policy predictions $(V_\theta(x^i_j), \pi_\theta(a^i_j|x^i_j))$, $i \in \mathcal{I}$, $j \in [1, T]$
    Compute critic's target values v^i_j with Eq. 59
    Compute advantages $A^{\pi_\theta}(x^i_j, a^i_j) = r^i_j + \gamma v^i_{j+1} - V_\theta(x^i_j)$
    Compute critic loss

$$L(\theta) = \frac{1}{2BT} \sum_{i=1}^{B} \sum_{j=1}^{T} (V_\theta(x^i_j) - v^i_j)^2 \tag{64}$$

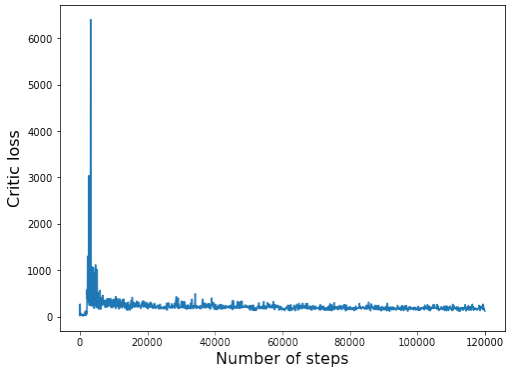
    Compute actor loss ▷ add entropy regularization

$$-J(\theta) = -\frac{1}{B} \sum_{i=1}^{B} \sum_{j=1}^{T} \rho^i_j \log \pi_\theta(a^i_j|x^i_j) A^{\pi_\theta}(x^i_j, a^i_j) - \frac{h}{BT} \sum_{i=1}^{B} \sum_{j=1}^{T} H[\pi_\theta(\cdot|x^i_j)] \tag{65}$$

    Update parameters $\theta \leftarrow \theta - \alpha \nabla_\theta (L(\theta) - J(\theta))$ ▷ Learner's actor and critic optimized together
    Send θ to all actors' networks
    `steps` $\leftarrow$ `steps` + T
  end while
end procedure



(a) Critic loss for IMPALA agent without bootstrapping.



(b) Critic loss for IMPALA agent with bootstrapping.

Figure 7: Critic loss for IMPALA agent when no bootstrapping was implemented (a) and when it was introduced (b). The interval between the peaks in (a) was on average equal to the unroll length T.

6 StarCraft II Learning Environment

StarCraft II Learning Environment (SC2LE, Vinyals et al. 2017) is a reinforcement learning environment based on the game of StarCraft II, which is a Real Time Strategy (RTS) game. As many other RTS games, a typical match between two players requires to gather resources, construct buildings, create units, develop technology and engage in combat the opponent, with the goal of destroying all of its units and buildings (see Figure 8).

Figure 8: A frame from StarCraft II game showing the base of one player: on the center there is the screen representation, on the bottom left there is the minimap, on the bottom center are shown some information about the unit currently selected and on the bottom right some of the available actions. To get to this point one player has to use its units to gather resources (minerals and gas), construct buildings and use them to create new kinds of units and to research new technologies.

StarCraft II is a game with imperfect information due to the partially observed map, has large state and action spaces and requires the adoption of long-term strategies. Moreover, given the fact that a player can only win or lose, the reward function for the full game will be 0 at all time steps except the last one, where it will be +1 or -1, depending on the result; this creates a huge challenge in assigning the credit for the various actions taken that brought to the outcome of the match.

All of this motivated the introduction of a suite of minigames, which cover different aspects of the game, testing the ability of an agent to learn specific tasks under controlled initial conditions and with a more informative reward function. There are in total seven minigames, with different difficulties and computation requirements.

A short description of each of them is provided in the paper StarCraft II: A New Challenge for Reinforcement Learning (Vinyals et al. 2017):

- MoveToBeacon: The agent has a single marine that gets +1 each time it reaches a beacon. This map is a unit test with a trivial greedy strategy.
- CollectMineralShards: The agent starts with two marines and must select and move them to pick up mineral shards spread around the map. The more efficiently it moves the units, the higher the score.

(a) MoveToBeacon

(b) CollectMineralShards

(c) FindAndDefeatZerglings

(d) DefeatZerglingsAndBanelings

Figure 9: Screenshots of the 4 StarCraft mini-games considered: (a) MoveToBeacon, (b) CollectMineralShards, (c) FindAndDefeatZerglings, (d) DefeatZerglingsAndBanelings. Of these, MoveToBeacon is really basic, CollectMineralShards and FindAndDefeatZerglings are somehow intermediate, whereas DefeatZerglingsAndBanelings is already challenging for a human who never played StarCraft II.

- FindAndDefeatZerglings: The agent starts with 3 marines and must explore a map to find and defeat individual Zerglings. This requires moving the camera and efficient exploration.
- DefeatRoaches: The agent starts with 9 marines and must defeat 4 roaches. Every time it defeats all of the roaches it gets 5 more marines as reinforcements and 4 new roaches spawn. The reward is +10 per roach killed and -1 per marine killed. The more marines it can keep alive, the more roaches it can defeat.
- DefeatZerglingsAndBanelings: The same as DefeatRoaches, except the opponent has Zerglings and Banelings, which give +5 reward each when killed. This requires a different strategy because the enemy units have different abilities.
- CollectMineralsAndGas: The agent starts with a limited base and is rewarded for the total resources collected in a limited time. A successful agent must build more workers and expand to increase its resource collection rate.
- BuildMarines: The agent starts with a limited base and is rewarded for building marines. It must build workers, collect resources, build Supply Depots, build Barracks, and then train marines. The action space is limited to the minimum action set needed to accomplish this goal.

For constraints in terms of time and computational resources, we focused just on the four minigames that showed the less complexity: MoveToBeacon, CollectMineralShards, FindAndDefeatZerglings and DefeatZerglingsAndBanelings. See Figure 9 for some screenshots of these mini-games.

6.1 State space

The observations of the environment are composed by three main variables:

1. `screen_state`: a representation of the game's screen in raw pixels feature planes, i.e. a 3D array with shape (`screen_channels`, `screen_width`, `screen_height`), following the convention used in Computer Vision, where images are seen as a certain number of matrices stacked along a "channel" dimension. An example of a feature plane would be a matrix containing ones in the pixels where friendly units are present and zeros in all others. For a list of all channels please refer to Figure 10.

2. `minimap_state`: a representation of the game's minimap, again in raw pixels feature planes, but with a different number of channels and in general with different resolution in pixels. So the `minimap_state` will have shape (`minimap_channels`, `minimap_width`, `minimap_height`). An example of a minimap's feature plane would be a matrix containing ones in the pixels that are also shown in the screen and zeros in all the others.

3. `player_state`: a 1D array of miscellaneous scalar quantities that are relevant to the game; we will call its length `player_features`. Example of features are the quantity of various resources stored (minerals and gas) or the number of units in the army.

In practice, for simplicity, we will restrict ourselves to the case in which screen and minimap have the same resolution and fix that resolution to 32 by 32 pixels unless specified otherwise. This is just high enough to resolve the smallest units, while keeping at a minimum the computational requirements for processing the state and the pixels on which is possible to click. In this way we are also reducing the action space by diminishing the possible values that spatial arguments can assume.

Regarding the number of channels, there are 27 screen channels and 11 minimap channels, but most of them are not useful in the restricted context of the minigames, thus after a first filtering we remain respectively with 11 and 7 channels.

6.2 Action space

The full action space of the game is huge, because it's composed by a total of 573 actions; furthermore, most actions require one or more parameters to be specified. For example an action can be to move a unit and it will take as arguments the point on the screen where to move (*spatial* argument) and a boolean flag (*categorical* or *non spatial* argument) saying whether to execute the action immediately or queue it to the one that is currently being executed.

Since we are working with a resolution of 32 by 32, there are 1024 possible values for every spatial argument of every spatial action. However not all actions are always available, since most actions are unlocked only in specific situations; the list of the available actions at a certain time step is received together with the current state from the environment. Another way in which the action space was reduced was by handpicking a small (but not minimal) set of actions that was enough to optimally solve all minigames. This resulted in a list of 20 main actions, that are reported in Figure 11.

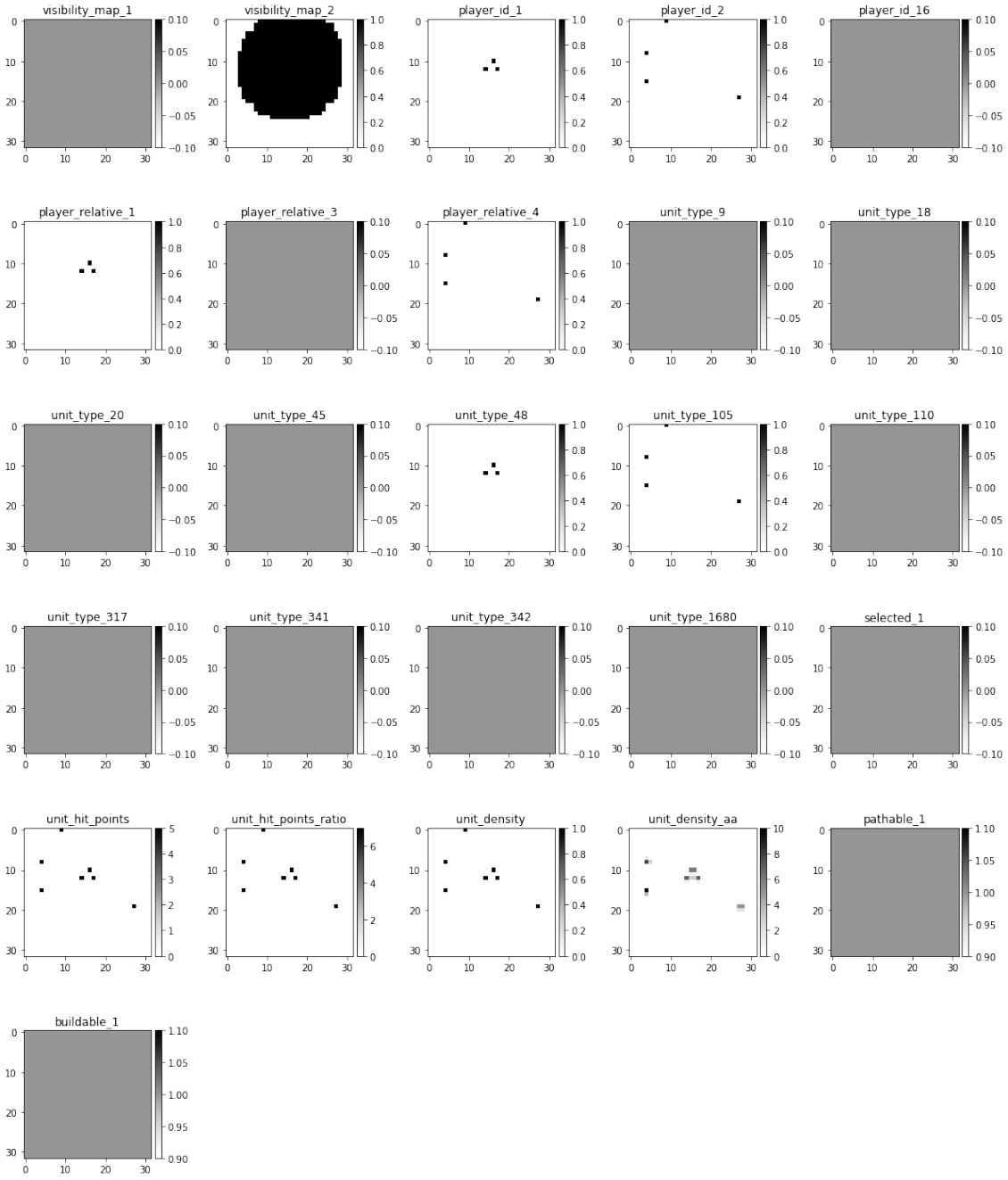


Figure 10: This figure shows how our agent actually sees the screen after an initial pre-processing discussed more in detail in Appendix Section C. All layers have a resolution of 32x32 and are stacked together to form a multi-channel image that we call `screen_state`. The `minimap_state` is represented in a similar way, whereas the `player_state` is just a standard array.

```
1 move_camera (1/minimap [32, 32])
2 select_point (6/select_point_act [4]; 0/screen [32, 32])
3 select_rect (7/select_add [2]; 0/screen [32, 32]; 2/screen2 [32, 32])
4 6 select_idle_worker (7/select_add [2])
5 7 select_army (7/select_add [2])
6 12 Attack_screen (3/queued [2]; 0/screen [32, 32])
7 13 Attack_minimap (3/queued [2]; 1/minimap [32, 32])
8 42 Build_Barracks_screen (3/queued [2]; 0/screen [32, 32])
9 44 Build_CommandCenter_screen (3/queued [2]; 0/screen [32, 32])
10 79 Build_Refinery_screen (3/queued [2]; 0/screen [32, 32])
11 91 Build_SupplyDepot_screen (3/queued [2]; 0/screen [32, 32])
12 268 Harvest_Gather_SCV_screen (3/queued [2]; 0/screen [32, 32])
13 273 Harvest_Return_SCV_quick (3/queued [2])
14 274 HoldPosition_quick (3/queued [2])
15 331 Move_screen (3/queued [2]; 0/screen [32, 32])
16 332 Move_minimap (3/queued [2]; 1/minimap [32, 32])
17 343 Rally_Workers_screen (3/queued [2]; 0/screen [32, 32])
18 344 Rally_Workers_minimap (3/queued [2]; 1/minimap [32, 32])
19 477 Train_Marine_quick (3/queued [2])
20 490 Train_SCV_quick (3/queued [2])
```

Figure 11: Actions selected to constitute the action space for all 7 minigames. First is reported the action ID, then the name of the action and between parenthesis the list of all the arguments that the action requires, in the format argument_ID/argument_name [range_of_possible_values]. The *spatial* actions are the ones that require a `minimap` or `screen` argument; all the others are called *non-spatial* or *categorical* actions. Many actions have been introduced for the last 2 minigames - CollectMineralsAndGas and BuildMarines - thus they either are not available or not useful in the other minigames. These are actions 6, 42, 44, 79, 91, 238, 273, 343, 344, 477, 490. Furthermore some actions have been included to create a redundancy in the action space, like `Attack_screen` and `Move_screen`, which implement basically the same action (move selected unit to the point in the screen indicated by argument `screen`), but with the difference that using `Attack_screen` the units engage in combat while reaching the indicated point. Also there are multiple selection actions, some of which are categorical (hence simple to use), while others are spatial hence more complicated. This means that there are different strategies that may or may not be equivalent and may have different learning curves.

6.3 Advantage Actor-Critic for SC2LE

The first agent that has been used to solve some minigames is an Advantage Actor-Critic (A2C), which uses multiple copies of the environment to produce a batch of trajectories in parallel and then uses those trajectories to perform the update. For an A2C update we need to know the state s_t, the reward r_t, the done signal d_t and the probability $\pi(a_t|s_t)$ of choosing the action that has been played at time t for all time steps. Regarding the probability of selecting an action with its arguments, we can consider it as a compound event and, calling α_i the i-th main action identifier and β_{jk} the k-th value of the j-th argument, we can factor the full probability as follows:

$$\pi(a_{ik_1\dots k_n}|s) = \pi(\alpha_i|s)\prod_{j=1}^{n}\pi(\beta_{jk_j}|s,\alpha_i) \tag{68}$$

where $a = (\alpha_i, \beta_{1k_1}, \dots, \beta_{nk_n})$ and assuming that all parameters are sampled independently from each other, only conditioned to the main action identifier.

A design problem that we face now is the fact that different actions might require the same argument (e.g. which point of the screen to click on), so we have two main options:

1. Use a single set of parameters (i.e. a single network) to sample a certain argument no matter which action is requiring it; In this case an explicit conditioning of the network on the main

action that has been sampled is necessary, because for different actions the optimal pixel on which to click might be different.

2. Use a different set of parameters to sample a certain argument for different actions. All sets of parameters are independent, so this actually equals to considering different the same parameters if they belong to different actions.

In our implementation we opted for the second option because it's simpler and heuristically yielded better results.

In conclusion we can factorize the probability of the compound action as:

$$\pi(a_{ik_1\dots k_n}|s) = \pi(\alpha_i|s)\prod_{j=1}^{n}\pi_{ij}(\beta_{k_j}|s) \tag{69}$$

where π_{ij} represents the network computing the probabilities of the $j-th$ argument of the $i-th$ action. More often we will compute directly the logarithm of that probability, because it's more stable numerically:

$$\log\pi(a_{ik_1\dots k_n}|s) = \log\pi(\alpha_i|s) + \sum_{j=1}^{n}\log\pi_{ij}(\beta_{k_j}|s) \tag{70}$$

6.4 Fully Convolutional Network for SC2LE

The following architecture, described in Vinyals et al. 2017, is the main architecture that we used for both the A2C and the IMPALA agents. Its structure is based on two important observations:

1. The spatial arguments, with 1024 values each, are the most difficult and important to learn. Their sampling is done by producing a matrix of probabilities in output, sampling from them and selecting the coordinates x and y of the selected pixel.

2. It is easier to learn how to produce a "good" probability matrix in output if all the computations involve only matrices and we never lose the spatial dimension flattening the input into an array.

Convolutional neural networks are the perfect tool for extracting patterns in multi-channel "images" and the characteristic of this architecture is that the probability matrices of the spatial arguments are obtained only through operations of convolution that maintain the spatial resolution. Recalling that the parameters that determine the output resolution of a convolution layer are the *kernel size*, the *stride* and the *padding*, the general formula to compute the output's resolution is:

$$out\_res = (in\_res - kernel\_size + 2 * padding)//stride + 1 \tag{71}$$

thus usually a simple way to keep the resolution unvaried is to use an odd kernel size, a stride of 1 and a padding given by $\frac{kernel\ size-1}{2}$ (Dumoulin and Visin 2016).

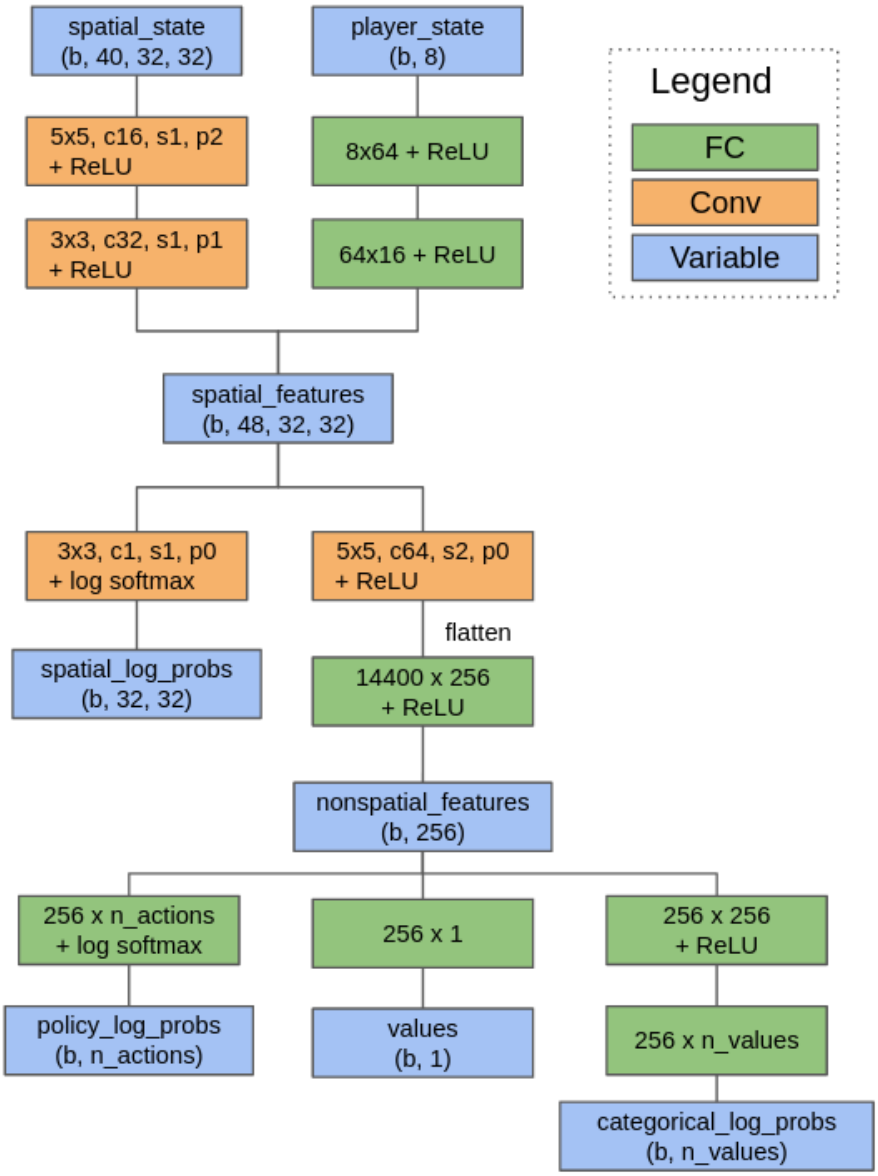


Figure 12: Fully Convolutional architecture schematics. Input variables are on top and are processed downwards. Under each variable is reported their shape, with b that is the batch size. Convolutional layer cells report information about the kernel size, the number of output channels, the stride and padding; Fully Connected layer (FC) cells report input and output dimension, whose product roughly approximates the number of parameters of the layer (neglecting the bias).

In the following part of the section we delve into the Fully Convolutional architecture's details; for a schematics of the architecture see Figure 12.

Pre-processing: the first thing to do in the pre-processing of the state's observation is to select only the features in the `screen_state`, the `minimap_state` and the `player_state` that are useful in at least one of the minigames. After that, if a feature can assume a high range of values (e.g. the player's resources or the hit points of the units), a logarithm in base 2 is applied; if a feature is encoded in categorical values (e.g. the type of unit in a certain cell) a one hot encoding is applied. The one hot encoding is a mapping of an integer n in range $1, \ldots, N$ to a 1D vector full of zeros in all entries except the $n - th$, where there will be a one. In case of

the one hot encoding of a feature plane, each pixel is mapped to a 1D vector, so from a single plane we obtain N planes. All other features are left untouched. In this case a feature may be a feature plane in case of the spatial variables, or a scalar feature in the case of the `player_state`. Finally the `screen_state` and the `minimap_state` are merged by concatenating them along the channel dimension in a `spatial_state` variable. Technically speaking, the pre-processing can be considered part of the environment, because it's a fixed process (not subject to learning) that is influencing the observation of the environment's state.

Spatial features network: this is the core of the Fully Convolutional architecture, because it's processing the pre-processed input into higher-level spatial features and merging them with the additional information extracted from the `player_state`.

The `spatial_state` is fed to two convolutional layers (Dumoulin and Visin 2016), the first one with a kernel size of 5, stride of 1, padding of 2 and 16 channels in output, the second one with a kernel size of 3, stride of 1, padding of 1 and 32 channels in output. Rectified Linear Units (ReLUs)[16] are applied after both layers.

The `player_state` instead is fed to two fully connected (FC) layers (see Goodfellow, Bengio, and Courville 2016, Chapter 6), respectively with 64 and 16 output units and both followed by ReLU activation functions.

At this moment we have the processed `spatial_state` that is a 32-channel image with linear dimensions of 32 by 32 and the processed `player_state` that is an array of 16 features. We merge them by concatenating the processed `player_state` along the channel dimension of every single pixel of the processed `spatial_state`; in this way we obtain the `spatial_features` of shape (48, 32, 32).

Non spatial features network: this network is extracting high-level scalar features from the `spatial_features`; its output will be used to compute the action probabilities (policy output), the value (critic output) and categorical parameters' probabilities.

First of all we feed the `spatial_features` to a convolutional layer of kernel 3 and stride 2 in order to decrease its resolution; this has 64 output channels and a ReLU is applied afterwards. The output is completely flattened, from a shape of (64, 15, 15) to a single array with 14400 entries, than is passed to a FC layer with 256 units and another ReLU is applied. We call the resulting vector with 256 features `nonspatial_features`.

Policy and value network: the policy head of the network applies a FC layer with a number of output units equal to the number of possible actions `n_actions`; in this way from the `nonspatial_features` we obtain the policy logits[17]. Subsequently we mask the actions that are not available, substituting their logits with $-\infty$ and then apply a log-softmax to get the log-probabilities. From there it's straightforward to sample the action.

To compute the critic's value we just apply a FC layer with one output unit to the `nonspatial_features`.

Categorical parameters: here we compute the categorical parameters' logits using a 2 FC layers, the first one with 256 units and the second one with as many units as the possible values of the parameter that we have to sample. Between the two layers is applied a ReLU and from the logits we obtain the log-probabilities by means of a log-softmax.

Spatial parameters: as anticipated before, to sample the spatial parameters we start from the `spatial_features` and apply to it a convolutional layer with kernel size of 3, stride and padding of 1 and a single output channel. The output will be a matrix of logits and applying a log-softmax to them we obtain the log-probabilities.

This architecture is closely following the FullyConv architecture proposed in the StarCraft paper, except for the non-spatial features network, where the original architecture was completely flattening the spatial features in an array of $48 \times 32 \times 32 = 49152$ and then passing it through a single FC layer with 256 output units and a ReLU activation. Our choice has been made considering the following facts:

[16] Given an input vector $\mathbf{x} = (x_1, \ldots, x_n)$, the Rectified Linear Unit applies element-wise the function $ReLU(x_i) = max(0, x_i)$

[17] By *logits* we commonly refer to an array of real numbers $(x_1, \ldots, x_N)$ that has to be mapped to probabilities by means of a soft-max operation $p_i = SM(x_i) \equiv exp(x_i) / \sum_{j=1}^{N} exp(x_j)$

1. Fully connected layers with N inputs neurons and M outputs neurons have NM parameters (neglecting the M biases added to the outputs before applying the activation functions); having 256 output units, we get 10.7 millions of parameters with the flattened input size of 49152 computed above.

2. Convolutional layers have $C_{in}C_{out}K^2$ parameters, because we are using a number of 3D kernels (kernel width K x kernel height K x input channels C_{in}) equal to the number of output channels C_{out} of the layer. In our case we have $C_{in} = 48$ and $C_{out} = 64$ and the kernel size of 3, so the number of parameters will be 27658 parameters, almost 400 times less. From there we can flatten the results, obtaining 14400 and with the same output size of 256 we managed to process the input with 2 layers instead of just one and at the same time using roughly 3.4 times less parameters.

3. It is generally recognised that heuristically using convolutions with stride bigger than 1 is a better alternative both to max pooling and fully connected layers to compress spatial information in a meaningful way. Our belief is that this simple change made our architecture more sample-efficient by reducing the number of parameters that we need to learn and by compressing the information in a more natural way for images.

Notice that this architecture is sharing part of the network between the actor and the critic, which from now on we consider parametrized by the same parameters θ and no more by two different sets of parameters θ and ϕ. Coherently we will optimize them together by minimizing the sum of their losses. The same principles apply both for A2C and IMPALA agents and will be used also in the next architecture that we will present.

6.5 Deep Architecture for SC2LE

The following architecture is presented in the paper Relational Deep Reinforcement Learning (Zambaldi et al. 2018) by another team of DeepMind. In this second paper on StarCraft's minigames the authors introduced two architectures, which both improved on the results obtained with the Fully Convolutional architecture introduced in the first paper. The focus of this paper was to demonstrate that using the self-attention mechanism introduced in the Transformers architecture (Vaswani et al. 2017), they could model relations between entities (e.g. vectors of features) and improve on a control architecture that wasn't using that mechanism. Looking at the results reported in Figure 13 we noticed that the gap between these two architectures was much smaller than the one from the previous paper; in other words both architectures outperformed the Fully Convolutional architecture[18], not only improving strategies learned by the latter, but learning qualitatively superior ones. From the same results seems improbable that the relational agent learned different strategies from the control agent and it's more likely that it just optimized them slightly better (which is still an impressive result, but does not fully justifies the added complexity).

[18]Here we are reporting a version of that architecture with an LSTM layer for memory processing, but it achieved better results of the FullyConv architecture on 3 minigames, worse results on other 3 and the same performance on the easiest one.

Agent	Mini-game						
	①	②	③	④	⑤	⑥	⑦
DeepMind Human Player [15]	26	133	46	41	729	6880	138
StarCraft Grandmaster [15]	28	177	61	215	727	7566	133
Random Policy [15]	1	17	4	1	23	12	< 1
FullyConv LSTM [15]	26	104	44	98	96	3351	6
PBT-A3C [33]	–	101	50	132	125	3345	0
Relational agent	**27**	**196** ↑	**62** ↑	**303** ↑	**736** ↑	4906	**123**
Control agent	**27**	187 ↑	61	295 ↑	602	**5055**	120

Figure 13: Mean scores (in the best run) achieved in the StarCraft II mini-games using full action set. ↑ denotes a score that is higher than a StarCraft Grandmaster. Mini-games: (1) Move To Beacon, (2) Collect Mineral Shards, (3) Find And Defeat Zerglings, (4) Defeat Roaches, (5) Defeat Zerglings And Banelings, (6) Collect Minerals And Gas, (7) Build Marines. Source: Relational Deep Reinforcement Learning, Zambaldi et al. 2018

For these reasons and for limited time and resources, we decided to implement only the control architecture, that for simplicity we will call Deep Architecture, whereas in this context the previous named Fully Convolutional will also be called Shallow Architecture.

The Deep Architecture improves the Shallow one on various aspects, namely:

1. More detailed state representation, giving in input the ID of the last played action and some layers saying if that action was applied to the screen, the minimap or neither of them. This information can be used to build more coherent strategies by "remembering" the previous action, even though, as all pre-processing operations, it is not learned but hard-coded.

2. Screen and minimap layers are processed separately by some convolutional layers before being merged; this in general could be avoided only if the two sets of layers have the same resolution, which is a particular case. A possible reason why this has been implemented even with the same resolution is to avoid convolutional kernels acting on layers with the same number of pixels but representing different scales: the minimap is representing an area much broader than the screen with the same number of pixels, so the idea of jointly processing the two sets of layers with the same convolutional kernel could create very confusing features. Two objection to this are that the agent should learn on his own to construct kernels that apply only on one set of features at the time even though it can access both and that even if we process them separately for a while it will come a point in which we will have to merge them again, so the agent will still have the possibility of confusing itself, just on higher level features.

3. A layer capable of memory processing has been added, so we can expect that the agent will learn easily long-term optimal strategies.

4. A block of 12 residual convolutional layers has been used for the main spatial processing (after the memory processing).

5. Given the increase in layers, to speed up the computations the input layers are reduced in resolution by 4 times per each linear dimension before entering the core of the architecture

and then the spatial matrices of logits are up-sampled by transposed convolutional layers to recover the original spatial resolution.

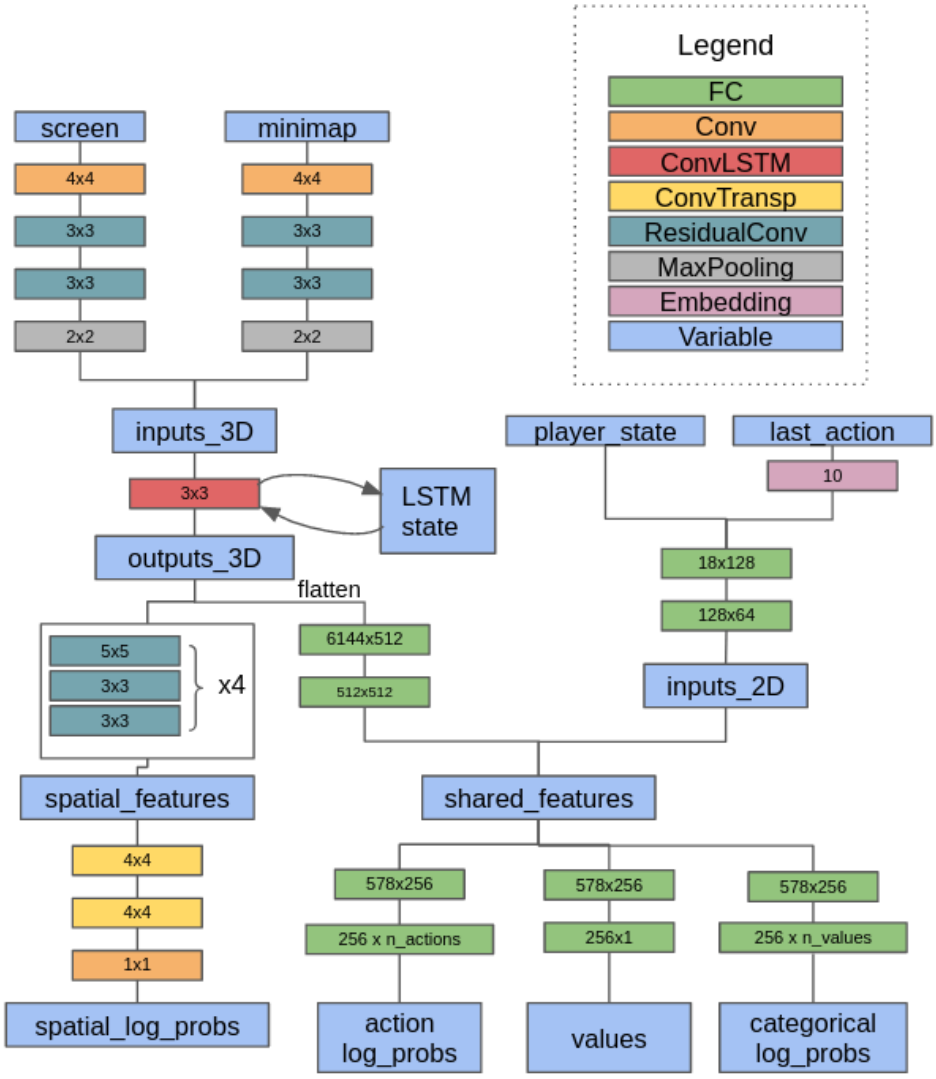


Figure 14: Deep Architecture schematics. Input variables are on top and are processed downwards. For space issues we did not report as much details as in Figure 12; for the dimensions of the variables, the activation functions and the parameters of the convolutional layers please refer to the details in Section 6.5. Convolution-related layer cells report only the kernel size while Fully Connected layer (FC) cells report input and output dimension, whose product roughly approximates the number of parameters of the layer (neglecting the bias).

We will now look into the details of the Deep Architecture, for a schematics of the architecture see Figure 14.

Pre-processing: the state pre-processing is similar to the one in the Shallow Architecture, with the difference that we also have the ID of the last action (just of the main action and not of its parameters) and we also add a channel both to the screen and the minimap layers. The additional channel to the screen (minimap) will contain all ones if the last action was applied to the screen

(minimap), otherwise it will contain all zeros. This channels are also called binary masks, since they are 0-1 matrices.

Screen and minimap initial processing: the screen and the minimap layers are passed through two identical but independent networks.

- First a convolutional layer with kernel size of 4, stride of 2, padding of 1 and 32 output channels; this halves the spatial resolution from 32×32 to 16×16.
- 2 residual convolutional layers are applied, keeping 32 channels both in input and output. A residual convolutional layer applies a ReLU activation on the input *before* applying a convolutional layer to it; the convolution has stride 1 and padding selected to balance the kernel size, so that the output will have the same resolution of the input. Finally the input is added to the result[19].
- Finally a max-pooling operation is used to halve again the resolution of the output, obtaining an 8×8, 32 channels image.

At this point the processed screen and minimap states are concatenated along the channel dimension, obtaining what we call `inputs_3D` of shape (b, 64, 8, 8), where b is the batch size.

Player state and last action initial processing: first of all we embed the last action in a 10-dimensional space through an embedding layer. This works as a look-up table, where for each integer value of the input (last action) we obtain as output a 10-dimensional vector, whose entries are learnable parameters of the embedding layer. Notice that the embedding layer has a number of independent parameter equal to the dictionary size (number of possible input values) times the embedding dimension (10). This kind of layer is usually used in machine learning to learn how to encode a series of symbols (or categorical values, where the value itself is just a label and has no inherent meaning) in a dense vectorial space. The embedded last action is concatenated with the player state and then passed through 2 FC layers with 128 and 64 output units respectively and ReLU activation functions; we call the output `inputs_2D`.

Memory processing: in this passage we process `inputs_3D` with a 2D-Convolutional Long Short-Term Memory (LSTM) layer (Xingjian et al. 2015). This layer differs from a standard LSTM (Hochreiter and Schmidhuber 1997) in the fact that instead of processing vectors with a single dimension with FC layers is processing multi-channel images with convolutional operations. We can expect each output pixel to depend mainly on the history of the local pixels; in fact it was introduced as a layer to help weather forecasting, where for example we could be interested in predicting where a cloud will be, based on where it was in the past, and of course local information is going to be much more important than information from far away. We call the output `outputs_3D`, that has shape (b, `out_channels`, 8, 8), where `out_channels` is the number of hidden channels of the LSTM and is set to 96 in our case.

Spatial processing: this part of the architecture processes the `outputs_3D` while keeping their shape (both resolution and number of channels); we call the output of this network `spatial features` and it will be used to sample the spatial arguments. The spatial processing consists of 4 residual convolutional blocks of 3 convolutional layers each. Every block works applies a 5×5 residual convolutional layer followed by 2 3×3 residual convolutional layers. Despite being quite a deep and computationally heavy architecture, since every layer is a residual layer the gradient is able to back-propagate easily through all layers, because there exist a pathway of identity mappings from the input to the output of the network.

Non-spatial processing: this part of the architecture processes the `outputs_3D` with the objective of obtaining a non-spatial vector that will be integrated to the information coming from the player state and the last action; this information will be then used to compute the policy logits, the state value and the categorical parameters of the actions. First we flatten the `outputs_3D`,

[19] This implements an identity mapping and forces the convolutional layer to learn just the residual of the transformation that we want to parametrize, e.g. $Res(x) = F(x) - x$, which should be easier to learn and more importantly helps in training deep networks by back-propagation because the gradient can easily propagate through the identity mappings without vanishing. For more details see He et al. 2016.

obtaining a tensor of shape (b, $96 \times 8 \times 8$) = (b, 6144), then we feed it to 2 FC layers with 512 output units each and ReLU activations to produce what we call `non-spatial features`.

Policy, value and categorical parameters: `inputs_2D` and `non-spatial features` are concatenated to produce what we call `shared features`. Shared features used to produce policy's log-probabilities, the critic's state value V and the categorical parameters' log-probabilities. To each of these outputs corresponds a different network "head", composed by a first FC layer with with 256 units and ReLU activation, followed by a second FC layer which will have a number of output units equal to:

- The number of possible actions, in the case of the policy's logits; a log-softmax is applied to the result to get the log-probabilities, from which we can sample the action for the current time step.
- A single unit, in the case of the critic's output; no activation function is applied to this result, so that the state-value will be able to virtually assume every possible real value.
- The number of categorical arguments multiplied by the maximum number of possible values that an argument can assume; this is to compute in parallel the log-probabilities of all possible values of all categorical arguments, as explained in the appendix Section B, and then sample them also in parallel.

Spatial parameters: the `spatial features` are fed to 2 Transposed Convolutional layers (Dumoulin and Visin 2016) with kernel size of 4, stride of 2 and padding of 1, which implement some sort of inverse operation of what a Convolutional layer with the same parameters would do (so basically they double the spatial resolution instead of halving it, in this case). We use 16 output channels with both Transposed Convolutional layers, thus the output will have shape (b, 16, 32, 32). After this we feed the result to a final convolutional layer with kernel size of 3, stride and padding of 1 and a number of channels equal to the number of spatial arguments, so that again we can compute in parallel the log-probabilities of all possible values of every spatial argument and sample them also in parallel.

7 Results

In the following sections we present various experiments made on the CartPole environment and the StarCraft II learning environment.

7.1 Preliminary studies on CartPole environment

In this section we present some preliminary studies of the Advantage Actor-Critic on a single CartPole environment (see Figure 2), comparing it with the REINFORCE algorithm and looking at different n-steps Temporal Difference updates.

CartPole is a RL environment in the open-source suite of OpenAI's gym library (Brockman et al. 2016). CartPole, known also as an Inverted Pendulum, is a pendulum with a center of gravity above its pivot point. It's unstable, but can be controlled by moving the pivot point under the center of mass. The goal is to keep the cartpole balanced by applying appropriate forces to a pivot point.

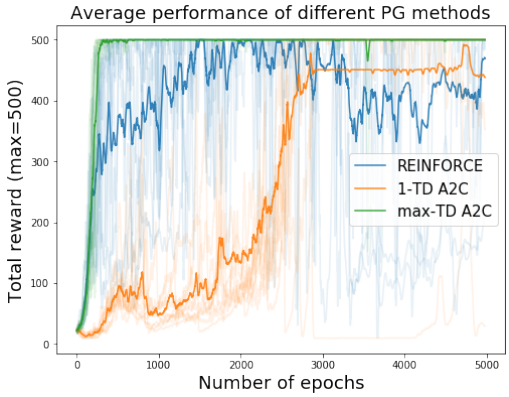
More specifically, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. Additionally a time truncation is performed if the agent is able to stabilize the pole for 500 consecutive time-steps, hence 500 is the highest reward achievable.

The observation provided at each time step to the agent is composed by the cart position, the cart velocity, the pole angle and the pole angular velocity, thus it can be regarded as a high-level state representation that is already making use of a physical understanding of the problem to best describe it. This setup in which we can just decide in which direction to push the cart is very simple and we use it to perform some preliminary studies on the family of policy gradient algorithms.

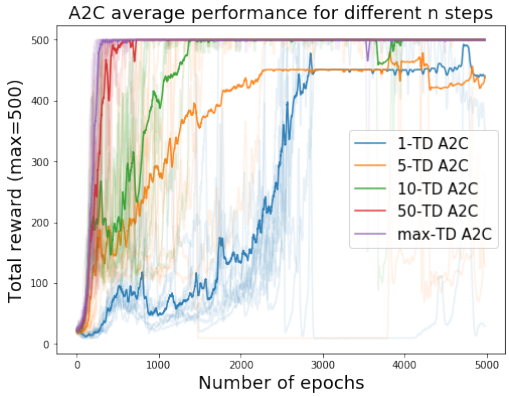
The first experiment in Figure 15a is comparing the performances of the REINFORCE algorithm (Eq. 33) without baseline, the 1-step Temporal Difference (1-TD) A2C and the max-steps Temporal Difference (max-TD) A2C. These last two models are represented by the same equation for the advantages (Eq. 50) and the critic training (Eq. 47), since the max-TD version is an n-step TD method with $n = \infty$ in normal episodes and $n_t = T - t$ when a time truncation happens. Both algorithms have been modified to use bootstrapping in the case in which an episode ends for time truncation and not for the agent fault (as explained in Section 2.1). In case of bootstrapping, the discounted value of the last state is added to the episode return (the power at which the discount factor γ is elevated is $T - t$, where T is the step at which the truncation happens and t the current step).

We can see that max-TD A2C is superior to the other two methods (since is the only one in which 10 out of 10 runs converge to the optimal policy in the given training time) and REINFORCE, while achieving fast initial improvement, it's the most unstable.

In Figure 15b we study how the n-steps parameter used in the Temporal Difference update of the A2C influences its convergence speed and its probability to converge. Our results show that the higher is n, the faster is the convergence and the higher is the probability of each run to converge to the optimal policy. This result might depend strongly on the task used to train and evaluate the agent, since CartPole task has no characteristic length and has a low dimensional state space, hence we expect that the ability to directly take into account the distant future (high n steps) in the update will be beneficial, also because the variability in the estimates won't be as high as it would in a high dimensional state space, where exponentially more trajectories can happen.



(a) Study of different policy gradient algorithms.



(b) Study of n-steps parameter in A2C.

Figure 15: (a) Comparison of the REINFORCE, the 1-step Temporal Difference (1-TD) A2C and the max-steps Temporal Difference (max-TD) A2C algorithms. All solid curves are the average of 10 runs, with the number of episodes on the x axis and the moving average over 20 episodes of the episode reward on the y axis; 500 is the maximum reward obtainable, because after that number of steps the episode is truncated automatically. We can see that max-TD A2C is superior to the other 2 methods and REINFORCE, while achieving fast initial improvement, it's the most unstable. (b) Study of the influence of the n-steps parameter on speed and probability of convergence of the n-TD A2C. All solid curves are the average of 10 runs, x and y axis represent the same variables of (a). In this plot there is a clear trend according to which the greater the n-steps parameter, the faster is the convergence. Also notice how the two agents with lower n-steps parameter (1 and 5) failed to converge in one run each, lowering the average asymptotic performance to approximately 450 (instead of 500 if all 10 runs would have converged).

7.2 A2C convergence and speed for different batch sizes

The next experiment that we performed had the objective of studying the influence of the batch size on the convergence and training time of the A2C. The basic ideas, introduced in Section 4.7, are that:

1. A2C's updates made on multiple trajectories at once improve the convergence and stability of the algorithm, because the agent is trained at the same time on more heterogeneous experience.

2. Producing the multiple trajectories in parallel is many times faster than producing them sequentially.

However there are many ways in which an improvement in an algorithm's stability can manifest itself, for example

- Transition from failure to success in solving a task.
- Not experiencing drops in performance once the optimal solution is reached.
- Converging faster to the solution, thanks to less noise in the neural network's updates. This might be translated for example in a better and faster credit assignment to positive actions.

To study the first two cases directly, we would have needed a task complex enough to experience failure and/or drops in performance in some cases, but typically this happens with very demanding tasks, in which each run might take days to be learned in the setting with high batch sizes (hence it would need up to 32 times more to give a fair chance to the setting with batch size of 1 to process the same experience). For these reasons we focused on the simplest task in the StarCraft's minigame

suite, called MoveToBeacon, which can always be solved, but with different times depending on the batch size used. The way we performed the experiments is the following: first we trained the A2C for 10 runs, a batch size of 1 and a convenient number of steps in order to be sure of reaching convergence[20]. After that we trained again the A2C for 10 runs, doubling each time the batch size and halving the training steps (this was done just to save resources). We then identified for each run the approximate step in which convergence was reached and compared the convergence for the different batch sizes. The key point is that the number of steps is not multiplied by the batch size, thus for example what are reported as 60 steps in the plot with a batch size of 32 are 60 steps for each of the 32 environments, which correspond to a single update in our setting (where we fixed the trajectory length of an update to 60, independently of the batch size).

The average of the runs together with their average convergence step are reported in Figure 16, where we can see how using more experience from parallel environments in an update is beneficial to learning. The magnitude of the increase in sample-efficiency depends on the regime in which we find ourselves: the lower the initial batch size, the more we have to gain in increasing it; conversely we can see that for higher batch sizes this increase is saturating. For instance doubling the batch size from 1 to 2 reduces the number of steps to convergence of 46%, whereas changing it from 16 to 32 reduced them only by 27%. Looking at the data we can say that the number of steps to convergence always decreased significantly increasing the batch size, but the dependence is sub-linear [21].

Regarding the training time, we can measure it in many ways, but probably the most important measures in practice are the number of steps and the wall-clock time to reach convergence. In order to have a more natural way to interpret these numbers, we report how many times the algorithm improves by increasing the batch size from one to the number reported in the x-axis. For instance if we had found a linear relation in these plots it would have meant that taking a batch size of n would have decreased by n times the steps to convergence (except for a small constant overhead) and the training time. In practice we found that these quantities grow sub-linearly with the batch size, but never saturate in the tested range. If instead of being constrained by time, one was constrained by the cost of resources, if it wasn't for the CPU, the most efficient thing would be to use a batch size of 4, but in the computational cluster that we used a GPU is 60 times more expensive per unit of time than a CPU, thus the highest number of batch size available would be preferable. In terms of their billing units (BU), doing 123 thousand steps (approximately the ones required to converge for a batch size of 1) with a single CPU's core and a GPU would cost 310 BU, doing 29 thousand with 4 CPU's cores and a GPU would cost 93 BU and doing 10 thousands steps with 32 cores and a GPU would cost 80 BU.

Finally we can see from Figure 18 the CPU and GPU efficiency depending on the batch size used. In particular it is possible to see in Figure 18a the drop in efficiency of the CPU due to the synchronization issues discussed in Section 4.8, which is more severe than one could expect at priori.

[20] We selected 720 thousands of steps, but it's an arbitrary quantity and it's not the point of the experiment

[21] Looking at the function with logarithmic scale in x we can see that is growing more than $\log(x)$ (because it's convex, whereas $y = a + b\log(x)$ should be linear), but using a log-log scale we can see that is growing less than x^α (because it's concave, whereas $\log(y) = \alpha\log(x) + \beta$ should be linear). Since it's not in the scope of this work to evaluate this functional dependence we stopped here this analysis.

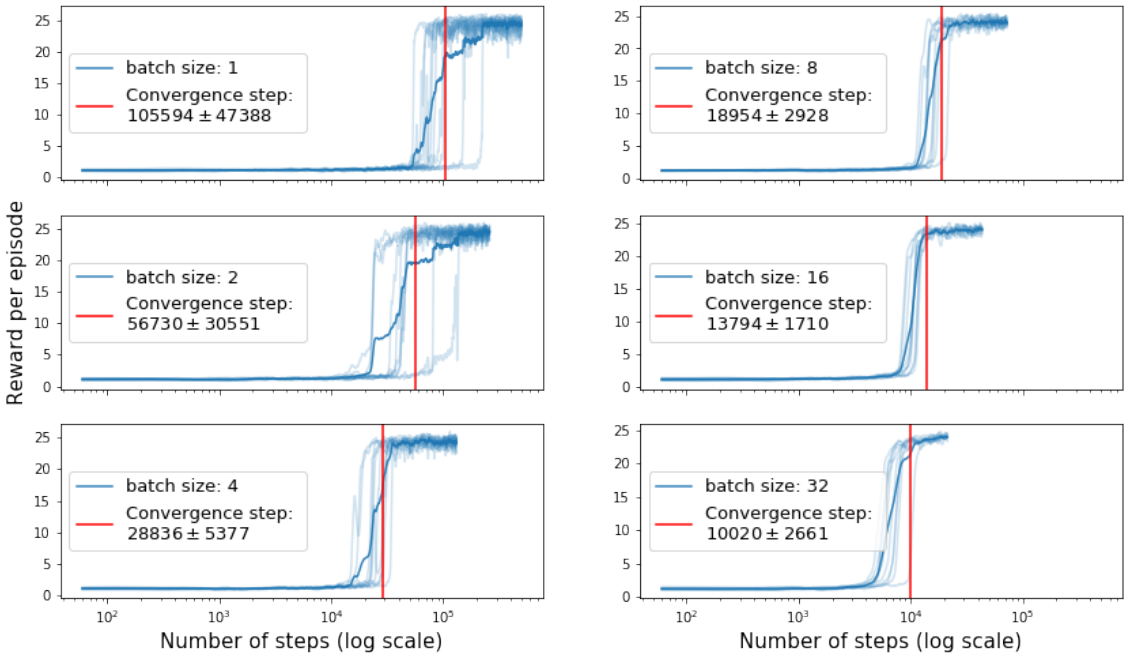
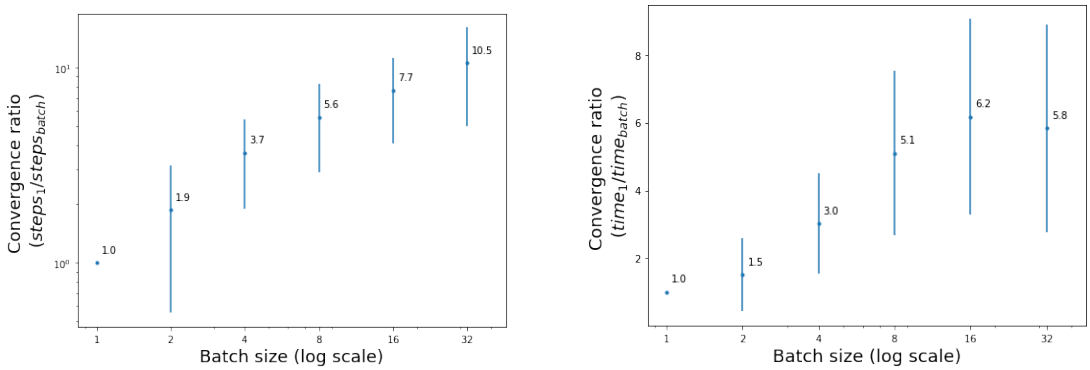
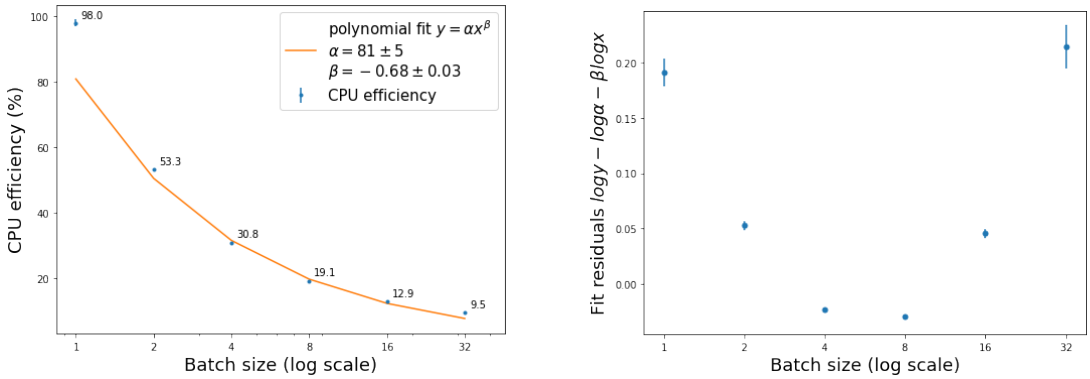


Figure 16: In the plots are reported the results of 10 runs of the A2C algorithm as a function of the number of steps for different batch sizes. The solid line represents the average over the runs. The x-axis is in logarithmic scale and each curve is intersected by a red line, showing the average convergence time (its value along with one standard deviation is reported in the legend). The key point is that the number of steps is not multiplied by the batch size, thus for example what are reported as 60 steps in the plot with a batch size of 32 are 60 steps for each of the 32 environments, which correspond to a single update in our setting (where we fixed the trajectory length of an update to 60, independently of the batch size). These plots demonstrate how using more experience from parallel environments in an update is beneficial to learning, since increasing the batch size always reduces the number of steps needed to converge.



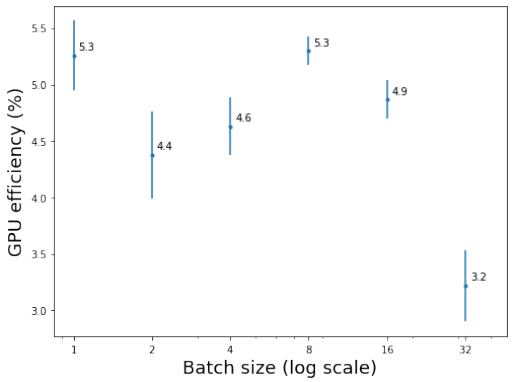
(a) (b)

Figure 17: (a) Gain in increasing the batch size in terms of convergence steps, measured as the ratio between the average steps to convergence using a batch size of 1 and the same quantity for the batch size indicated in the x-axis (represented in log-scale); for instance it's saying that with a batch size of two we are 1.9 times faster (in terms of steps required) in reaching convergence than with a batch size of 1. It's possible to see that the gain becomes strictly sub-linear for batch sizes greater than 4 (remember that the x-axis is in log-scale). (b) Gain in increasing the batch size in terms of wall-clock time. This measurement is also taking into account imperfect parallelization due to the time wasted in synchronization and higher computing time for the operations on the GPU. Notice how this issue decreases the potential gains due to multiple environments because the same number of steps requires more time to be executed on n environments each instead of just on one. For instance after a batch size of 16 it's no more efficient to increase the batch size, because the time gained in reducing the steps is out-weighted by the one spent due to decreased throughput.



(a)

(b)



(c)

Figure 18: (a) Average CPU efficiency during the runs with different batch sizes. The plot shows how severe the synchronization issue is for the efficiency of the A2C, at least in our implementation. The data has been fitted by a polynomial function, but the analysis of the residuals (b) shows that the efficiency actually decreases slightly less than polinomially. (c) Average GPU efficiency during the runs with different batch sizes. It can be seen that the efficiency is very low on average (even though the peak performance is between 30% and 42%) and is not clearly affected by the changes in the batch size, with the only exception of the data for the batch size of 32, where there is a substantial drop in efficiency. It is not clear whether it is due to the batch size or to an anomaly in the cluster performance while the runs were being performed, since other slow-downs were also observed elsewhere.

7.3 Minigames results

In this section we present the main results of this thesis, comparing the performances of the A2C and IMPALA agents on the four minigames considered (see Figure 9) from the StarCraft II learning environment between them and with the FullyConv agent trained in Vinyals et al. 2017.

Before training for a long session is necessary to estimate for how many steps we can train in a given amount of time, e.g. 3 days, which is the maximum training time on a GPU node of the CSC cluster in a single session. To do that we ran two test jobs, one for each algorithm, on the minigame MoveToBeacon for a small number of steps (e.g. 100 thousands), using the same parameters that we would use in the real training. In Table 1 are reported some measures of efficiency for the A2C and IMPALA algorithms, together with their throughput measured in frames per second (FPS). In this specific case the tests are executed with 20 parallel environments, but the throughput is just measuring how many frames per second are consumed from each environment by the agent,

thus the total experience unfolding the batch size would be 20 times more. The speed of both algorithms is affected also by the resolution at which we are running the game: both algorithms are tested at 32 x 32, but since A2C is much slower than IMPALA, it has also been tested at 16 x 16, because this is the resolution at which we decided to train it in the end.

Algorithm	CPU efficiency	GPU mean load	GPU std dev load	GPU max load	FPS
A2C (32)	12.6%	1.5 %	6.4%	56%	4.4
A2C (16)	18.7%	2.5 %	6.1%	38%	11.8
IMPALA (32)	86%	15%	26%	94%	72.5

Table 1: GPU and CPU utilization in A2C and IMPALA for a single run on the CSC cluster. Both algorithms are tested on MoveToBeacon using 20 CPU cores and 1 GPU. IMPALA is far more performing than A2C, with a throughput of 72.5 frames per second (FPS) for each parallel environment compared to only 4.4 FPS for the A2C, thus gaining a factor 16.5 in speed.

Coming to the real training, both algorithms have been trained using the Fully Convolutional architecture described in Section 6.4. The summary of the parameters used in A2C and IMPALA are reported respectively in Tables 2 and 3.

Parameter	Value
Resolution	16x16
Optimizer	Adam
Learning rate	$3 \cdot 10^{-4}$
Entropy cost	$7 \cdot 10^{-2}$
Unroll length	60
n-steps	20

Table 2: Parameters used to train A2C agent on the StarCraft minigames. The parameter "unroll length" is the length of the trajectory considered to make the update, which is different from the entire trajectory length (which typically in the 4 minigames is 240, if the episode does not end before time truncation).

Parameter	Value
Resolution	32x32
Optimizer	RMSprop
Learning rate	$3 \cdot 10^{-4}$
Entropy cost	$5 \cdot 10^{-4}$
Unroll length	60

Table 3: Parameters used to train IMPALA agent on the StarCraft minigames.

For each minigame considered we trained both agents for five runs in order to accumulate some statistics. Except for MoveToBeacon, which is the simplest minigame (hence required less training than the others) the A2C agent has been trained for less steps than the IMPALA agent to keep down the cost in terms of time and billing units.

In Figure 19 are reported all the runs of the A2C, whereas the results can be found in Table 4. Plots of the IMPALA runs and the table with its results can be found in Figure 20 and Table 3. A2C results match the results from DeepMind agent (Vinyals et al. 2017) for two out of four minigames (MoveToBeacon and CollectMineralShards), whereas achieve significantly lower scores on the other two minigames, which might be considered more complicated than the previous two.

IMPALA significantly improves on A2C on 2 out of 4 minigames (the harder 2) and achieves slightly better results (although without reaching statistical significance) also on MoveToBeacon. In CollectMineralShards however the A2C agent performs better both on average and on the best run, even though IMPALA's results are also good (they are qualitatively similar, although not as optimized as in the A2C case). Probably the result which is best showcasing the difference between the two algorithms is the best run of IMPALA on Figure 20d, where after 4 million steps there is a sharp improvement in the agent's performance, leading it to outperform also the best DeepMind agent. This shows how being able to train for longer times is necessary to obtain better results. All the comparisons are reported in Table 6.

Before comparing our results to the ones from DeepMind, it is worth noting that their agents were trained for 600 millions steps with the equivalent of a batch size of 64, which is around 2 orders of magnitude more than the amount of training that we used. Moreover they reported only the average results achieved by the best out of 100 runs[22]: this implies that their results should be compared only with our best run average score, but even then the different number of runs rig the comparison in favour of DeepMind score.

Looking at the comparison between our results with IMPALA and the DeepMind FullyConv agent we notice that:

- The performance of MoveToBeacon is slightly lower, a gap which could be filled with a tuning in the hyper-parameters (mainly the learning rate and entropy cost).
- The performance of CollectMineralShards is significantly lower, on average because one run failed to learn the same strategy of the others, but also because probably the other 4 runs were using the right strategy, but without being able to optimize it properly. It might have been the case that a longer training time could have yielded better results.
- On FindAndDefeatZerglings all runs score fairly similar and the average performance is compatible with the DeepMind best mean.
- Finally on DefeatZerglingsAndBanelings there is a lot of variance in the IMPALA's results, with three runs that scored less than DeepMind best mean, one that was compatible with it and the best one that significantly outperformed it after 6 millions of steps, probably learning a qualitatively superior strategy.

If then we consider the advantage given by approximately two orders of magnitude more of training and by taking the maximum over 100 runs instead than over just 5, it is fair to expect that the IMPALA agent under the same conditions would match or surpass DeepMind best agent performance on all 4 minigames.

[22]This is not a critique to how they presented the results, since also the average would have presented problems, vastly underestimating the achievements of the agent. This is because they were sampling the learning rate in the interval $(10^{-5}, 10^{-3})$, instead of being partially tuned as in our case. It is however very likely that taking the maximum over 100 runs with randomly sampled learning rates with our agent would have yielded better results than taking the maximum over only 5 runs, even though using a partially tuned learning rate.

Minigame	A2C Best Run Average	A2C 5 Runs Average	DeepMind Best Run Average	Steps to train
MoveToBeacon	25.5 (0.2)	25.0 (0.3)	**26**	0.6 M
CollectMineralShards	**106.3 (0.9)**	101 (7)	103	2.8 M
FindAndDefeatZerglings	41 (1)	37 (6)	**45**	2.8 M
DefeatZerglingsAndBanelings	53 (2)	37 (17)	**62**	2.5 M

Table 4: Results for the 4 minigames we trained the A2C agent on using the Fully Convolutional architecture. Best Run Average is showing the mean result of the best run over the last 100 updates (5 test episodes are executed every 10 updates) for a total of 50 test episodes, with standard deviation of the mean included. 5 Runs Average is reporting the average over the mean results of the 5 runs, each of them computed as in Best Run Average; the standard deviation is computed over the set of 5 measures. DeepMind Average is reporting the Best Run Average results achieved by the Fully Convolutional architecture trained with an A3C agent on 64 parallel environments for 600M steps and 100 runs with different learning rates each.

Minigame	IMPALA Best Run Average	IMPALA 5 Runs Average	DeepMind Best Run Average	Steps to train
MoveToBeacon	25.6 (0.2)	25.2 (0.3)	**26**	0.6 M
CollectMineralShards	98.5 (0.8)	90 (14)	**103**	6M
FindAndDefeatZerglings	**46.4 (0.6)**	44 (1)	45	6M
DefeatZerglingsAndBanelings	**76 (3)**	61 (8)	62	11 - 16 M

Table 5: Results for the 4 minigames we trained the IMPALA agent on using the Fully Convolutional architecture. Best Run Average is showing the mean result of the best run over 100 test episodes, with standard deviation of the mean included. 5 Runs Average is reporting the average over the mean results of the 5 runs, each of them computed as in Best Run Average; the standard deviation is computed over the set of 5 measures. DeepMind Average is reporting the Best Run Average results achieved by the Fully Convolutional architecture trained with an A3C agent on 64 parallel environments for 600M steps and 100 runs with different learning rates each. Our agent was trained on DefeatZerglingsAndBanelings for 72 hours, but the number of steps varies from run to run because being a combat game there are many restarts that ultimately depend on the ability of the agent to fight and frequent restarts slow down the game rendering.

Minigame	A2C Best Run Average	IMPALA Best Run Average	A2C 5 Runs Average	IMPALA 5 Runs Average
MoveToBeacon	25.5 (0.2)	**25.6 (0.2)**	25.0 (0.3)	**25.2 (0.3)**
CollectMineralShards	**106.3 (0.9)**	98.5 (0.8)	**101 (7)**	90 (14)
FindAndDefeatZerglings	41 (1)	**46.4 (0.6)**	37 (6)	**44 (1)**
DefeatZerglingsAndBanelings	53 (2)	**76 (3)**	37 (17)	**61 (8)**

Table 6: Comparison between A2C and IMPALA. IMPALA significantly improves on A2C on 2 out of 4 minigames and achieves slightly better results (although without reaching statistical significance) also on MoveToBeacon. In CollectMineralShards however the A2C agent performs better both on average and on the best run.

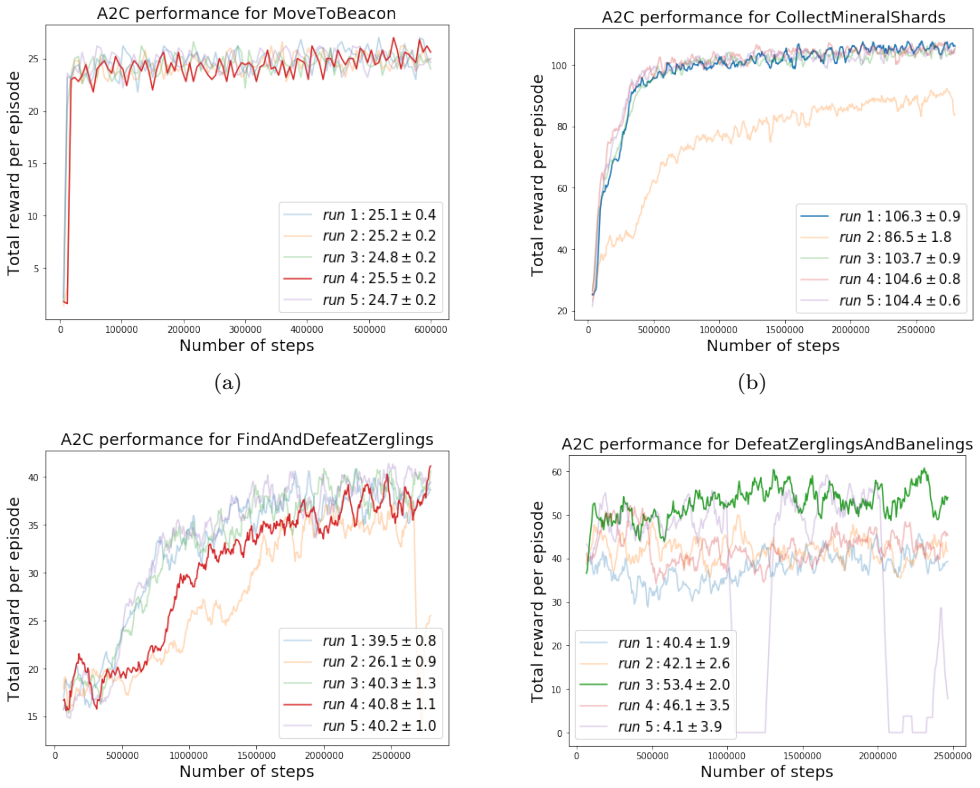


Figure 19: Performance of the A2C agent trained on 4 different StarCraft II minigames. Plots are showing the episode return (total reward) as a function of the number of steps (agent-environment steps) for 5 different runs of each task; the best run is represented without transparency. Steps are not multiplied for the batch size used (20) and the agent is trained on a resolution of 16x16 both for the screen and the minimap. In the plots' legends are reported the asymptotic average returns with one standard deviation of the average. Minigame (a) shows a fast convergence to the optimal result, (b) and (c) have slower convergences in the order of 1-2 million of steps, with one run of (b) that scored much less than the others and one run of (c) whose performance dropped shortly after 2.5 millions of steps. In minigame (d) the agent converges to a range of sub-optimal strategies in the first hundred thousands steps and fails to improve subsequently.

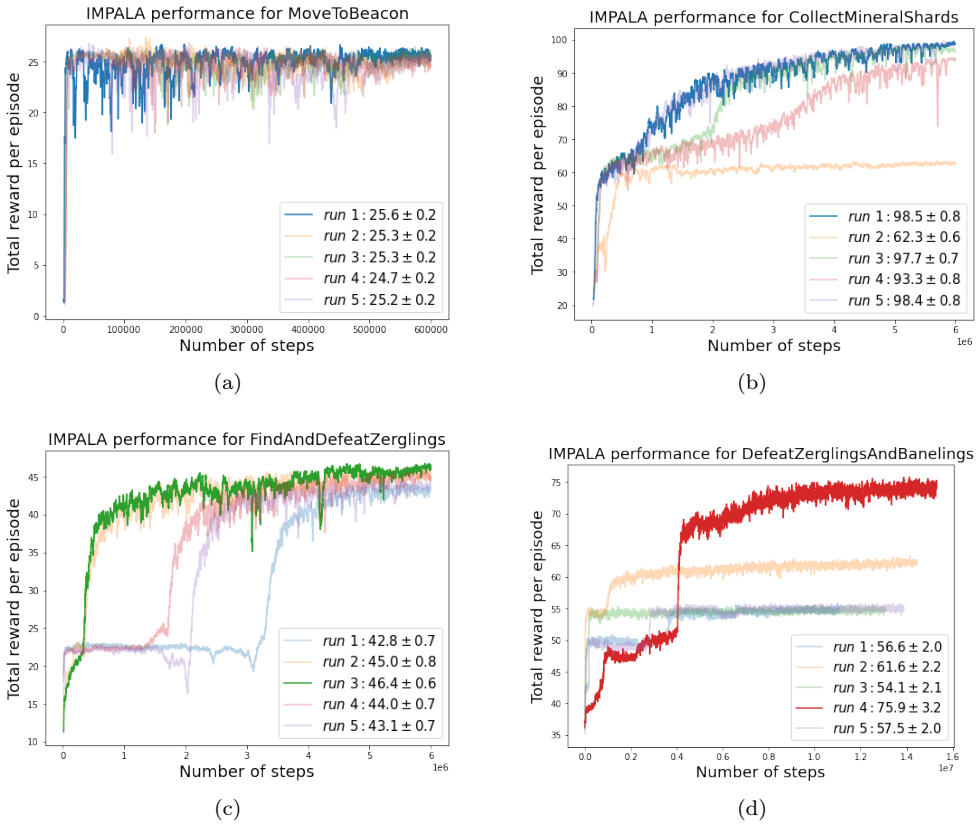


Figure 20: Performance of the IMPALA agent trained on 4 different StarCraft II minigames. Plots are showing the episode return (total reward) as a function of the number of steps (agent-environment steps) for 5 different runs of each task; the best run is represented without transparency. Steps are not multiplied for the batch size used (20) and the agent is trained on a resolution of 32x32 both for the screen and the minimap. In the plots' legends are reported the asymptotic average returns with one standard deviation of the average. Minigame (a) shows a fast convergence to the optimal result, (b) and (c) have slower convergences in the order of 1-4 million of steps, with one run of (b) that scored much less than the others. In minigame (d) 3 runs scored less than DeepMind average, one was compatible with that and the best one significantly outperformed it after 6 millions of steps, probably learning a qualitatively superior strategy.

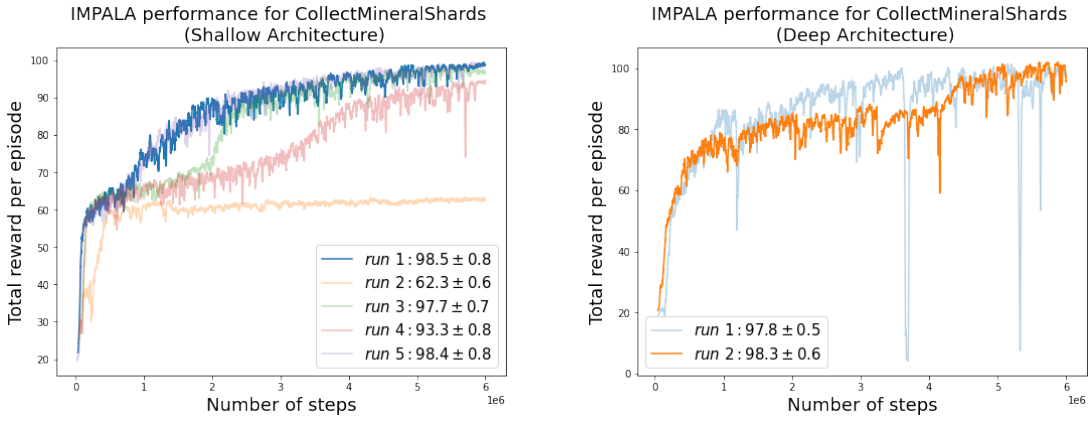
7.4 Additional experiments with the Deep Architecture

As a final experiment, we made some preliminary attempts in training the Deep Architecture discussed in Section 6.5.

The main issues in training it are that the increase in computations performed at each step slows down the throughput from approximately 72.5 FPS to 46.5 FPS (which is almost 2/3 of the initial throughput) and that the increased number of trainable parameters increases the experience needed to train the network. We focused our attempts on two different minigames in order to look for improvements in different areas, also taking into account the results achieved by DeepMind in Zambaldi et al. 2018 with the control architecture.

The first minigame was CollectMineralShards and we selected it because all previous agents converged to a local optimum in terms of strategies, learning how to control two units as a single group and moving them together to collect the mineral shards distributed across the map; however the global optimum would be to learn to control the two units separately and at the same time, moving them in two different areas of the map, potentially doubling the episode reward. In fact the Fully Conv agent in Vinyals et al. 2017 achieves an average score of 103, whereas the control agent in Zambaldi et al. 2018 achieves 187 on average and the relational agent from the same paper achieves 196 (see Figure 13).

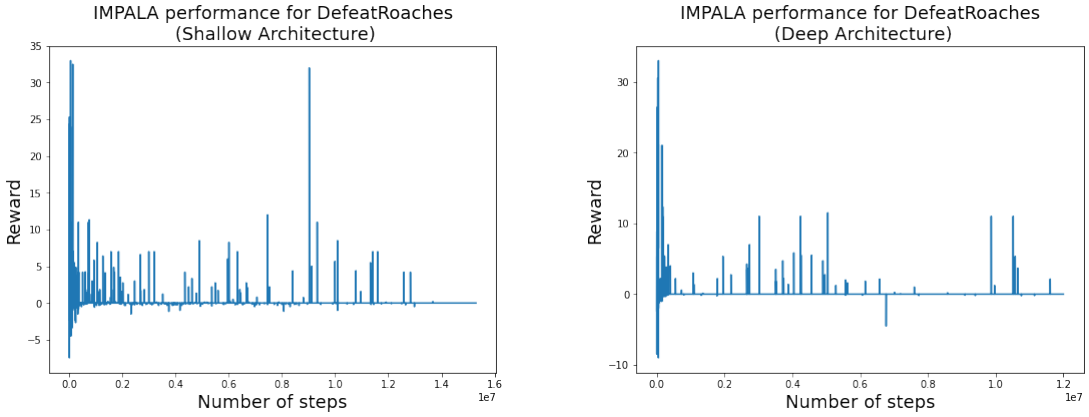
We trained the agent for 2 runs of 6 millions steps, obtaining results comparable to the ones of the Fully Convolutional (Shallow) architecture. It is probable that by training for longer time and trying out more runs we could have obtained at least one agent learning to split the units and outperform the previous agent, but the authors of Zambaldi et al. 2018 trained their agents for 10 billions of steps in order to achieve their results, thus there was no guarantee to replicate their results with our resources.



(a) (b)

Figure 21: Comparison between the Shallow (a) and Deep (b) Architectures for the IMPALA agent. We did only 2 runs of (b) because it was a preliminary study. The results suggest that given the same amount of experience the two architectures achieved similar results. It is probable that both architectures would have benefited from more training steps.

The second minigame that we attempted was DefeatRoaches, a combat game which resulted much harder than DefeatZerglingsAndBanelings to solve, since both A2C and IMPALA agents failed to learn anything at all, with the performances always collapsing to 0 after a certain point. We trained the IMPALA agent with the Deep Architecture for 72 hours (approximately 12 million steps) and a single run, but the performance was similar to the previous attempts with the other algorithms, i.e. never managed to learn a viable strategy, as can be seen in Figure 22.



(a) (b)

Figure 22: Plots showing a single run of training for the IMPALA agent using the Shallow (a) and the Deep (b) Architectures. The results in both cases are the complete failure of training, with an asymptotic reward of 0, which happens mainly when the agent learns not to engage in combat. Using the Deep architecture did not improve the IMPALA performance in this case.

Overall with these preliminary runs we were able to train the Deep Architecture so to match the performance of the Shallow one, but not to improve on that given similar amounts of training time and resources.

8 Conclusions and discussion

In this thesis we derived and implemented two Reinforcement Learning algorithms, the synchronous Advantage Actor-Critic (A2C) and the Importance Weighted Actor-Learner Architecture (IMPALA), and trained them on four out of seven minigames of the StarCraft II learning environment (SC2LE). On these environments we were able to obtain results comparable to the ones in the paper StarCraft II: A New Challenge for Reinforcement Learning (Vinyals et al. 2017) by DeepMind with approximately two orders of magnitude less of training, while using the same network architecture. Overall our IMPALA implementation is more than 16 times faster than our A2C implementation and improved the A2C performance on three out of four minigames.

As side studies, we showed in the CartPole environment how using an n-steps Temporal Difference (TD) update for the A2C agent is superior to a single-step TD update, although the optimal n will depend on the characteristic of the task in general. We then showed how we can decrease the training time of the Advantage Actor-Critic using multiple environments with the synchronous A2C algorithm and finally how decoupling the actors from the learner sped up of multiple times the training speed. Finally we implemented and partially trained the control architecture of the paper Relational Deep Reinforcement Learning (Zambaldi et al. 2018) by DeepMind, which is a deep neural network containing various kinds of layers used to process the input efficiently both in the spatial and temporal dimension.

In this work we showed how it's possible to tackle complex decision-making problems with Deep Reinforcement Learning and how to design algorithms that make an efficient use of modern hardware infrastructure to reduce the training time required for cutting-edge tasks by orders of magnitude. We also contribute to lower the entry barrier for other researchers that want to use the SC2LE as a RL benchmark by releasing all the project's code as open-source[23].

Given the limited time and resources, we couldn't train our agents on the remaining 3 minigames, which were far more demanding than the other 4, nor systematically train the deeper architecture in an attempt to match the current state-of-the-art for StarCraft II mini-games. One possible development of this thesis would be to scale up all the training variables while using a version of IMPALA that runs both inference and updates on multiple GPUs in order to reproduce the latest DeepMind's results on SC2LE in a reasonable amount of time.

More original developments could address the many challenges that prevent the application of Deep RL solutions to real-life problems: the most pressing ones probably are increasing the algorithms' sample efficiency and learning to apply the knowledge acquired in previous tasks to unseen ones. For instance in StarCraft any task more complicated than a simple navigation problem requires at least millions of iterations (which translates to entire days of playing) in order to achieve human-level abilities and the algorithm has to be trained again from scratch to solve a different but related scenario.

Humans instead can learn much more efficiently and they can effectively adapt their knowledge to new challenges. In fact, having a semantic understanding of the environment's objects, features, possible actions and goals makes it intuitive for us to play a game even without knowing exactly its rules. We use previous knowledge of semantic, hierarchical and causal relations between variables to project strategies and models learned in the past to new ones. Then we refine them in a process of trial-and-error, where trials (ideally) are thoughtfully planned thanks to the causal understanding of the problem, so as to efficiently explore the most promising strategies and come up with the best one in very few iterations.

Given the richness of scenarios that SC2LE provides, a future challenge could be to solve all mini-games sequentially, showing capabilities of transferring knowledge between tasks and achieving a sample efficiency similar to humans.

[23]The main project is hosted at https://github.com/nicoladainese96/SC2-RL.

References

Achiam, Joshua (2018). *Spinning Up in Deep Reinforcement Learning*. URL: `https://spinningup.openai.com/en/latest/user/introduction.html`.

Babaeizadeh, Mohammad, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz (2016). "GA3C: GPU-based A3C for deep reinforcement learning". In: *arXiv preprint arXiv:1611.06256*.

Bellemare, Marc G, Yavar Naddaf, Joel Veness, and Michael Bowling (2013). "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47, pp. 253–279.

Bellman, Richard (1957). "A Markovian decision process". In: *Journal of mathematics and mechanics*, pp. 679–684.

Bellman, Richard and Robert E Kalaba (1965). *Dynamic programming and modern control theory*. Vol. 81. Citeseer.

Berner, Christopher, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. (2019). "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680*.

Boedecker, Joschka, Jost Tobias Springenberg, Jan Wülfing, and Martin Riedmiller (2014). "Approximate real-time optimal control based on sparse gaussian process models". In: *2014 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE, pp. 1–8.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba (2016). "OpenAI Gym". In: *arXiv preprint arXiv:1606.01540*.

Cybenko, George (1989). "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4, pp. 303–314.

Deisenroth, Marc Peter, Gerhard Neumann, and Jan Peters (2013). *A survey on policy search for robotics*. now publishers.

Dumoulin, Vincent and Francesco Visin (2016). "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285*.

Espeholt, Lasse, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu (2018). "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *arXiv preprint arXiv:1802.01561*.

Fujimoto, Scott, Herke Van Hoof, and David Meger (2018). "Addressing function approximation error in actor-critic methods". In: *arXiv preprint arXiv:1802.09477*.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine (2018). "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *arXiv preprint arXiv:1801.01290*.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016). "Identity mappings in deep residual networks". In: *European conference on computer vision*. Springer, pp. 630–645.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Howard, Ronald A (1960). "Dynamic programming and markov processes." In:

Kaiser, Lukasz, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. (2019). "Model-based reinforcement learning for atari". In: *arXiv preprint arXiv:1903.00374*.

Khansari-Zadeh, S Mohammad and Aude Billard (2014). "Learning control Lyapunov function to ensure stability of dynamical system-based robot reaching motions". In: *Robotics and Autonomous Systems* 62.6, pp. 752–765.

Ko, Jonathan and Dieter Fox (2009). "GP-BayesFilters: Bayesian filtering using Gaussian process prediction and observation models". In: *Autonomous Robots* 27.1, pp. 75–90.

Kuttler, Heinrich, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktaschel, and Edward Grefenstette (2019). "TorchBeast: A PyTorch Platform for Dis-

tributed RL". In: *arXiv preprint arXiv:1910.03552*. URL: https://github.com/facebookresearch/torchbeast.

Levine, Sergey (2018). "Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review". In: *arXiv preprint arXiv:1805.00909*.

Lillicrap, Timothy P, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra (2015). "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971*.

Lin, Long-Ji (1993). *Reinforcement learning for robots using neural networks*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

Minsky, Marvin (1961). "Steps toward artificial intelligence". In: *Proceedings of the IRE* 49.1, pp. 8–30.

Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu (2016). "Asynchronous Methods for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1602.01783*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller (2013). "Playing Atari with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1312.5602*.

Munos, Rémi, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare (2016). "Safe and efficient off-policy reinforcement learning". In: *Advances in Neural Information Processing Systems*, pp. 1054–1062.

Nagabandi, Anusha, Gregory Kahn, Ronald S Fearing, and Sergey Levine (2018). "Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 7559–7566.

Parisi, Simone, Voot Tangkaratt, Jan Peters, and Mohammad Emtiyaz Khan (2019). "TD-regularized actor-critic methods". In: *Machine Learning* 108.8-9, pp. 1467–1501.

Rummery, G. and Mahesan Niranjan (1994). "On-Line Q-Learning Using Connectionist Systems". In: *Technical Report CUED/F-INFENG/TR 166*.

Schulman, John, Pieter Abbeel, and Xi Chen (2017). "Equivalence Between Policy Gradients and Soft Q-Learning". In: *arXiv preprint arXiv:1704.06440*.

Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz (2015). "Trust region policy optimization". In: *International conference on machine learning*, pp. 1889–1897.

Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel (2015). "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438*.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov (2017). "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347*.

seungeunrho (2020). *minimalRL*. URL: https://github.com/seungeunrho/minimalRL.

Silver, David, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587, pp. 484–489.

Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. (2017). "Mastering the game of go without human knowledge". In: *nature* 550.7676, pp. 354–359.

Sutton, Richard S. and Andrew G. Barto (1998). *Reinforcement Learning: An Introduction*. The MIT Press.

Sutton, Richard S., David McAllester, Satinder Singh, and Yishay Mansour (1999). "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Proceedings of the 12th International Conference on Neural Information Processing Systems*. NIPS'99. Denver, CO: MIT Press, pp. 1057–1063.

Tadelis, Steven (2013). *Game theory: an introduction*. Princeton University Press.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is all you need". In: *Advances in neural information processing systems*, pp. 5998–6008.

Vinyals, Oriol, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John P. Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing (2017). "StarCraft II: A New Challenge for Reinforcement Learning". In: *arXiv preprint arXiv:1708.04782*.

Watkins, Christopher JCH and Peter Dayan (1992). "Q-learning". In: *Machine learning* 8.3-4, pp. 279–292.

Watkins, Christopher John Cornish Hellaby (1989). "Learning from delayed rewards". In:

Williams, Ronald J. (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning*. DOI: https://doi.org/10.1007/BF00992696.

Williams, Ronald J. and Jing Peng (1991). "Function Optimization using Connectionist Reinforcement Learning Algorithms". In: *Connection Science* 3.3, pp. 241–268. DOI: 10.1080/09540099108946587.

Wu, Yuhuai, Elman Mansimov, Shun Liao, Alec Radford, and John Schulman (2017). "OpenAI Baselines: ACKTR & A2C". In: URL: https://openai.com/blog/baselines-acktr-a2c/.

Xingjian, SHI, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo (2015). "Convolutional LSTM network: A machine learning approach for precipitation nowcasting". In: *Advances in neural information processing systems*, pp. 802–810.

Zambaldi, Vinicius Flores, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David P. Reichert, Timothy P. Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals, and Peter W. Battaglia (2018). "Relational Deep Reinforcement Learning". In: *arXiv preprint arXiv:1806.01830*.

Appendix

A A3C and A2C pseudo-code

Our implementations were based on the open-source code available on GitHub (seungeunrho 2020) and modified accordingly to our necessities. Both the A2C and the A3C implementations make use of a PyTorch library for multiprocessing that works only for a single machine (or computing node in case of a cluster).

A3C:

- Launch as many processes as CPU cores available.
- Initialize a global model in the main process and share its memory across processes. Memory sharing implemented in this way is another part of the program that works only on a single machine.
- For each process initialize a local model and a local environment.
- Each local agent interacts with its copy of the environment to create a state-action-reward trajectory.
- At the end of a trajectory (or after a certain number of steps) the actor-critic loss is computed and then the gradients of the loss with respect to the network's parameters are obtained through back-propagation (backward operation called on the loss).
- The gradients of the parameters of the local model are copied on the global model.
- The optimizer applies the gradients to the global model (with the optimizer's step operation).
- The new parameters are copied back to the local model.

A2C:

- Launch as many processes as CPU cores available; for each process initialize a local environment.
- Initialize a single agent model, possibly loaded on a GPU.
- Initialize a parallel environment class, that handles the communication between the agent and all the local environments of actions in one direction and states and rewards on the other.
- At each interaction the parallel environment collects the current state and reward from all local environments, synchronizing them. All the states are batched together and fed to the agent.
- The agent outputs a batch of actions, each corresponding to its decision in the corresponding state of the batch. The actions are submitted to the parallel environment and from there dispatched each to its own environment.
- After T interactions have been made, the trajectories and the log-probs of all the actions taken are batched and a unique update is made on the global model.

All code is available on GitHub at https://github.com/nicoladainese96/RelationalDeepRL.

B StarCraft A2C various versions

In this section we discuss different implementations and design choices of the Advantage Actor-Critic for the StarCraft environment.

The first design choice was about the richness of the state representation. The amount of information provided by the StarCraft environment is huge, as described in Section 6.1. In particular before pre-processing we have 27 screen layers, 11 minimap layers and 11 player features. However some layers are useless in all minigames and would be of some use only on the full StarCraft game; moreover some minigames require much less information than others, e.g. they do not require to look at the minimap nor at the player information. As a consequence we can tailor the pre-processed state representation to a specific minigame injecting some knowledge of the task in form of information filtering, making the task easier to learn, or we can use the most general state representation in order to make it work in all scenarios (possibly also in previously unseen ones). What we did was to build and test different states representations, which from the simplest to the most general and complex used:

1. 3 to 6 layers of the screen, but no information from the minimap nor the player features;
2. A custom number of screen and minimap layers, but no player information.
3. All possibly useful layers of screen and minimap and all possibly useful player features.

We noticed that the sample efficiency of the algorithm tested on the simplest minigame (MoveToBeacon) wasn't much affected by the increase in complexity of the state representation, but the time required to perform the forward and backward operations increased (because the parameters to process the state representation were increased too).

A second design choice was about the different possible ways of dealing with arguments' networks:

1. Creating one network (a single set of parameters) for each argument, that would be used by all the actions that required that argument. The network output can be conditioned with the ID of the action that is requiring the argument (e.g. we might want to encode if the point on the screen that we have to click on is going to select a unit present there or move an already selected unit in that direction, so that we can aim to a proper region of interest), for example by embedding the action in a vector and using the resulting vector as additional information to be concatenated to the spatial and non-spatial information that are fed as input to the argument's network.
2. Creating many independent networks for the same argument, one for each action that is requiring it. No action conditioning is needed, because the networks are already dedicated to a single argument, hence no additional context is needed.

We tried out both possibilities and found out that the second choice worked much better in terms of sample-efficiency and final performance. Tests once again were performed on MoveToBeacon minigame.

Finally, from a computational point of view there was another design decision to take: in A2C at each time step we receive a batch of states and have to output a batch of actions, each with their parameters. We would like to do this in a vectorial fashion, so that when the computation is executed in the GPU it will be completely parallelized. The problem with that is that different actions require not only different parameters, but a different number of them; also the number of parameters required is just a small fraction of the total number of them (the specific value depends on the action space considered, but is around $\frac{1}{10}$.

The solutions we came up with were two. Both of them start by sampling a batch of main actions, then the first option, that we can call sequential, would cycle on the main actions, to see which parameters they required and to sample them. In the second option instead we modified the spatial and categorical networks, so that they computed the log probs of all the possible values

of all possible arguments in a single step, then for each index in the batch, we would sample all possible arguments and finally mask out all those that are not required by the main action with that index.

To exemplify this with the spatial arguments, let's suppose that there are 20 main actions and only 10 of them require a spatial argument each; let's also suppose that we are working with a batch size of 16 (number of parallel environments). The network for the spatial parameters takes as input the spatial features of shape (16, 48, 32, 32) and, if it had to sample a single argument, would output a matrix of logits of shape (1, 32, 32) from which it could sample the argument; then it would have to repeat this 16 times (and on average only 8 actions would require a spatial argument at all). In this case instead we output a matrix of shape (16, 10, 32, 32) and then sample in parallel 10 values, one for each channel, for each index in the batch. So we obtain a batch of values of shape (16, 10), where the values are mapped to the interval [1, 1024] and from which we can retrieve their corresponding x and y, and then use a binary mask that selects a single value among the 10, based on the ID of the action. For instance if the action $a^{(i)}$ has id 2, then it will require the second of the 10 possible values and the $i-th$ row of the binary mask will be $(0, 1, 0, \ldots, 0)$. The resulting spatial parameters will be 16 (x,y) pairs corresponding to the 16 pixels that have been sampled from the correct arguments. This method can in theory be faster than the previous one, even though is computing 10 times more arguments, because is avoiding 2 loops, one on the batch size and one on the various arguments. In practice we observed that is only slightly faster, but on the same order of magnitude of the first method. Of course the performances of the two methods depend on the batch size, the action space and the parameter space.

C StarCraft pre-processing details

In this section we report in full detail the choices made to pre-process the state space.

The screen layers were divided into 4 categories based on the transformation that was applied to them: one hot encoding (ohe), logarithm of 2 (log), casting to floating point format (float) and discarded (unknown, since we discarded the variables which we didn't know what they did and never changed values throughout the minigames).

ohe: since some of these variables assumed just few integer values (e.g. 1, 2, 16) in a much higher range (e.g. [1,16]), instead of encoding each of them in a vector of dimension equal to the range of possible values (e.g. 16), they were encoded in a vector of dimension equal to the actual number of values that were assumed during the minigames. Since this information wasn't provided a priori anywhere, we had first to infer the values by a few test games for each minigame and then to keep looking if new values ever appeared during training (since they depend on the scenario, on the hardest minigames new units and buildings appear only if specific strategies are implemented), in order to take note of them and update the pre-processing information in an iterative way. For completeness we report here all the layers that we one hot encoded in the format ID/`name` (number_of_values, [possible_values]), since at our knowledge no such a list is publicly available.

1. 1/visibility_map (2, [1,2])
2. 4/player_id (3, [1,2,16])
3. 5/player_relative (3, [1,3,4]
4. 6/unit_type (11, [9, 18, 20, 45, 48, 105, 110, 317, 341, 342, 1680])
5. 7/selected (1, [1])
6. 24/pathable (1, [1])
7. 25/buildable (1, [1])

Similarly for the minimap layers that we one hot encoded the list is:

1. 1/visibility_map (2, [1,2])

2. 3/camera(1, [1])
3. 4/player_id (3, [1,2,16])
4. 5/player_relative (3, [1,3,4]
5. 6/selected (1, [1])
6. 7/unit_type (11, [9, 18, 20, 45, 48, 105, 110, 317, 341, 342, 1680])
7. 9/pathable (1, [1])
8. 10/buildable (1, [1])

However to reduce the number of layers we discarded the unit_type layer from the minimap.

log: the logarithm in base 2 was applied to a couple of screen layers containing numerical values spanning more than 2 orders of magnitude: unit_hit_points (ID 8) and unit_hit_points_ratio (ID 9).

float: the screen layers unit_density and unit_density_aa were numerical variables with values respectively in [0,2] and [0,16] (the upper values are the maximum observed). These were simply casted to floating points variables. No such variables were present in the minimap layers.

unknown: the following layers were not used in the state representation because they are never used in the minigames scenarios.

For the screen the complete list is:

1. 0/height_map
2. 2/creep
3. 3/power
4. 10/unit_energy
5. 11/unit_energy_ratio
6. 12/unit_shields
7. 13/unit_shields_ratio
8. 17/hallucinations
9. 18/cloaked
10. 19/blip
11. 20/buffs
12. 21/buff_duration
13. 22/active
14. 23/build_progress
15. 26/placeholder

For the minimap the list is:

1. 0/height_map
2. 2/creep
3. 8/alerts

To this list was also added the layer unit_type as mentioned before.

Finally of the player_features, 3 were discarded, 2 were processed with the log_2 and the rest were numerical variables that were simply casted to float.

discarded:

- 0/player_id
- 9/warp_gate_count
- 10/larva_count

log:

- 1/minerals
- 2/vespene

float:

- 3/food_used
- 4/food_cap
- 5/food_army
- 6/food_workers
- 7/idle_worker_count
- 8/army_count