# Reinforcement Learning for hyperparameters optimization

H. Cherfi, A. Louahadj, A. Langlade, K. Gaume, C. Mouliets and L. Gonçalves Braz

***Abstract*—Technologies in the field of Artificial Intelligence (AI) have blown up in the last decade and have been involved in more and more research surroundings, in which their use allows the automation of a variety of processes. Recently, AI has been empowering new breakthroughs which have helped computer science research immensely. In this article, we focus on a particular approach : Reinforcement Learning (RL). This way of working computer science issues enables for instance a machine to learn from the mistakes of thousands of experiences (steps) while it tries to reach its goal in a defined environment. Video games can be used to develop and train RL models and many studies have been conducted using simple games, for example the well-known Atari game Pong. The agent implemented has learned to play the game so well that it cannot lose anymore. This performance is made possible by the agent's parameters (hyper-parameters), which have been adjusted for these games specifically through a long tuning process. More generally, any Neural Network (NN) approach for reinforcement, supervised or unsupervised learning, highly rely on these adjustments. This process is typically led by an expert in Computer Science using personal experience. Therefore, when their use is required from researchers who do not have the required skills, solving this problem can be challenging. Moreover, this process needs to be repeated from scratch each time the type of NN, the dataset or the objective is modified. In this study, we describe many concepts useful to enable a RL agent to train a NN in order to automatize this process. We believe this approach to be promising and able to outperform the current optimization algorithms used. We therefore focus on well-known RL methods and methods to optimize hyper-parameters, aiming to use the first to perform the latter in a more generic way.**

## I. Introduction

The use of a Neural Network is one of many ways in which we can teach a model, or a program, how to behave facing a specific problem. The process to achieve this result, as we know, is to run a large scale of experiments with thousands or millions of data samples to show it how to react in theses situations. This approach has been successful in many different fields, including very recently in 2020 with DeepMind's team's AlphaFold project, claiming to have resolved a 50 years old chemistry fundamental problem [1]. However, this process requires enormous amounts of data, as well as time, and knowledge from many different fields experts to understand if the agent is performing well or if it can still be improved. More specifically, in the field of medicine, it requires medics or biologists to understand and annotate the samples as they have the expertise needed to make the right decision observing the situation presented in the data samples. However, the decisions made by the agent are fundamentally ruled by mathematical equations and computer science logic and are in the field of Computer Science. As in psychology, when you want to teach someone what is good to do or not, a reward signal is sent to indicate if what was done was a good or a bad decision. By knowing this, the agent then tries to optimize its decision-making process by itself through the interactions with the data samples. However, as we can expect, it is not as simple as it seems, and involves the knowledge of an expert in Machine Learning to observe how the agent behave and tune it manually if we want to achieve satisfying performance after its training.

In the field of Reinforcement Learning (RL), there is a large application test bench available, and the most used is composed of video games, due to their notably basic environments. As they are easy to simulate, a whole dedicated library, GYM, is presented in section VII. The only point of a game is, of course, to win, and so it's easier to set up the reward system as a function of the score obtained by performing a sequence of actions. For example, Cartpole is one of the most used and basic test games in RL. Applications in the field of robotics have been developed during the last years, as we will discuss, and are becoming part of the assessment protocol of RL agent's performance as they are complex, realistic tasks to achieve but are easy to build nowadays in simulations. Many algorithms have already been studied and provide a large fan of approaches which we will introduce, though not exhaustively, as Deep Q-Learning or Actor Critic in sections III and IV respectively.

From this point forward, our goal is to get rid of the human, RL expert, intervention during the training process to provide to the medical or biological experts an automatic and simple way to build an efficient NN. Following the last years regained interest in RL, this idea is thus to develop a RL agent that can train an other agent. No algorithm has been already developed in this way, we will therefore in this article, present all the knowledge needed to achieve this objective. First we need to figure out which approach of RL to focus on, therefore we present the most commonly used. Regarding the behavior of these agents, hyper-parameters are properties that govern the entire training process and need to be initialized before training a model, as they greatly impact the quality of training (for example, the number of hidden layers or the learning rate). We noticed that hyper-parameter optimizations are specific approaches in RL and we will present two of

them to become familiar with the ideas. Finally, reusable holdout appeared to be a key point to solve the problem while avoiding over fitting to the data. By giving less information to the RL agent, it becomes more generalized as it will need to train NN aimed at solving various supervised learning problems.

## II. Key concepts

### A. Markovian Decision Process (MDP)

In RL, only sequential decision problems are considered. To do so, in this sub-section, we suppose that the environment is fully observable, that is to say the agent is always able to localize itself. At each time step $t$, in the state $s_t$, every possible $ACTIONS(s)$ or $A(s)$ are known from the agent. However, sometimes they are unreliable and the environment studied then becomes stochastic, with a transition model $P(s'|s,a)$. This model describes the probability to reach state $s'$, from state $s$, performing the action $a$. In RL, it is also usually assumed the transitions are Markovian : $P(s_{t+1}) = P(s_{t+1}|s_t, a_t)$. In other words, $s_{t+1}$ only depends on the action $a_t$ realized in its previous state $s_t$ and not on earlier ones. As the agent has to choose an action at each time step, we can set a simple utility function, to determine its performance, depending on the sequence of visited states, the environment history. At each time step, it will also receive a reward, $R(s)$, that can be either positive or negative. Therefor, the utility of the agent can be evaluated straightforwardly by the sum of the rewards received, from the history.

To sum up, as written in the book [2], *a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a Markov decision process, or MDP*, and is composed, as seen before, of *set of states (with an initial state $s_0$); a set $ACTIONS(s)$; a transition model $P(s'|s,a)$; and a reward function $R(s)$*. To represent the behavior of the agent, a policy denoted $\pi$ is used, with $\pi(s)$ the action recommended by the policy in the current state $s$. If the agent has a complete policy, then no matter in which state we get after performing action $a$, it will always know what to do next. In the case of a stochastic environment, an optimal policy, denoted $\pi^*$, yields the highest expected utility (i.e. the highest expected performance as measured by the utility function). As a result, the choice between two trajectories will change depending on the value of $R(s)$ for the non-terminal states, thus the optimal policy greatly depends on the reward function's value. *The careful balancing of risk and reward is a characteristic of MDPs that does not arise in deterministic search problems* as we are never sure where we will get from $s_t$ doing $a_t$ and *is characteristic of many real-world decision problems*.

### B. Utility

As explained in the chapter 17 of [2], the choice of the performance measure, written $U([s_0, s_1, ..., s_n])$, is not arbitrary and will also largely influence the choice of optimal policy. To choose an appropriate measure, the first question to answer is whether there is a finite horizon or an infinite horizon for decision making. A finite horizon means that there is a fixed time $N$ after which there is no interest in continuing, therefor we stop to play : $U([s_0, s_1, ..., s_{N+k}]) = U([s_0, s_1, ..., s_N])$, for $k > 0$. In this case, the optimal action in a given state can change over time, it is non-stationary. For example, if we are close to the end of the game, the agent will tend to take actions more risky to rush to the goal instead of taking a safer path taking more time steps. On the contrary, with no fixed time limit and in the same state at the different time, there is no reason for the agent to take a different action, as it has all the time it wants, the optimal policy is stationary.

Considering the second option, one simple and important assumption we can make is that the agent's preferences between two state sequences are stationary. If two state sequences $[s_0, s_1, s_2, ...]$ and $[s'_0, s'_1, s'_2, ...]$ begin with the same state (i.e., $s_0 = s'_0$), then the two sequences should be preference-ordered similarly to the sequences $[s_1, s_2, ...]$ and $[s'_1, s'_2, ...]$. Under this condition, assigning utilities to sequences can be done in two ways :

- Additive rewards :
  $\hat{U}([s_0, s_1, s_2, ...]) = R(s_0) + R(s_1) + R(s_2) + ...$
- Discounted rewards :
  $\hat{U}([s_0, s_1, s_2, ...]) = R(s_0) + \gamma * R(s_1) + \gamma^2 * R(s_2) + ...$
  where $\gamma \in [0; 1]$ (discount factor, preference of current over future reward)

In RL, discounted rewards are used for two main reasons :

- If $\gamma < 1$, the utility of an infinite sequence can be bounded as : $\hat{U}([s_0, s_1, s_2, ...]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq R_{max}/(1-\gamma)$
- Influence the agent to reach the terminal state, usually with a big positive reward, instead of staying infinitely in a state giving a little reward. As future rewards have less importance than the next ones, the agent will see a higher utility in a sequence reaching the terminal state compared to others. If we know the policy will necessarily get to a terminal state, it is a proper policy

### C. Partially Observable MDP (POMDP)

To summarize the idea presented in chapter 17 of [2], an environment that is not fully but partially observable has a transition model $P(s'|s,a)$, actions $A(s)$, rewards $R(s)$ and also a **sensor model** $P(e|s)$ that is the probability of receiving the evidence $e$ knowing we are actually in the state $s$. It can be seen as an additive noise on the sensor that might make the agent believe it is in a certain state, while it is actually in another one. But this does not really matter, since the optimal action the agent should choose depends only on the agent's belief state, and not on it's real one. If we consider $b$ as the belief state, having the higher probability to be the actual state $s$, $b$ will be the state maximizing the expected utility. This is

helpful and will simply let us write the decision cycle of a POMPD agent as :

- given the belief state b, execute the action $a = \pi^*(b)$
- receive percept $e$
- refresh the belief state thanks to $b$, $e$, $a$ and repeat

### D. A practical explanation

#### 1) Environment

Notations :

| | |
|---|---|
| $\mathbf{S}$ | the state space |
| $s \in \mathbf{S}$ | a state of the environment |
| $\mathbf{A}$ | the set of actions an agent can take in the environment |
| $a \in \mathbf{A}(\mathbf{s})$ | the action taken by the agent in state $s$ |
| $\delta : \mathbf{S} \times \mathbf{A}(\mathbf{s}) \to \mathbf{S}$ $(s, a) \mapsto \delta(s, a) = s'$ | the new state reached by taking action $a$ when in state $s$ |
| $r : \mathbf{S} \times \mathbf{A}(\mathbf{s}) \to \mathbf{R}$ $(s, a) \mapsto r(s, a)$ | the reward given by the environment to the agent for taking action $a$ when in state $s$ |

One of the key concepts in RL is the environment. The environment is what the agent is going to interact with in order to learn and to become better at doing the task it is supposed to do. To interact with the environment, the agent will perform one action, then receive two pieces of information: the reward and the state.
An example to illustrate this, is the *freeway* environment created by *GYM*.

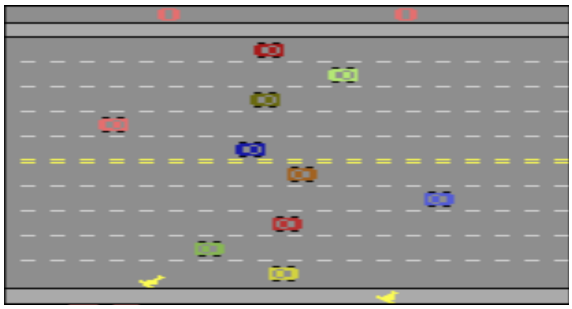

Fig. 1. Freeway environment [3]

In this environment, the two kangaroos at the bottom of Figure 1 are the agents that are going to interact with the environment (in this case called a multi-agent environment). The goal for the agents is to cross the road without being hit by a vehicle. An agent can either go forward or backward. This is called a finite discrete bounded action space : $\mathbf{A} = \{$*forward , backward*$\}$. In other environments, action spaces could be infinite discrete unbounded (i.e. $\mathbb{N}$); infinite continuous bounded (i.e. $[0; 1]$); or infinite continuous unbounded (i.e. $\mathbb{R}^+$).

At the beginning, the agent has no clue on the strategy to adopt in order to cross the road. To make a decision and take an action, the agent needs to know the state it is in. In other words, it needs to *consider* the environment. In the aforementioned example, the agent can observe that the screen is made of 210 x 160 pixels; that a pixel contains three color components (RGB) each coded on 8 bits ; and that, consequently, a single frame totals up to 806400 bits.

Although this makes up for a tremendous number of possible arrangements, the environment is finite. Also, because it can be considered as **fully observable**, the same behavior can be applied as with an MDP by using the observations as belief states, even if the environment actually is a POMDP.

Last but not least, the environment is **deterministic** because each action in a specific state will always trigger the same reaction. To go further, it is easy to understand that with a single frame as the one seen on picture 1, the agent knows where each car is, but does not know at what speed it is moving : the agent could use two frames, make the difference between them, and then figure out the speed of each car.

Let us assume the agent moves forward. When doing so, the environment will give feedback to the agent : the observation, as well as a **reward**. In the case of the Freeway environment, if nothing happen, the environment will give a reward of 0. If the kangaroo, which represents the agent, is hit by a car, the game is over, and the agent obtains a reward of -1. On the other end, if the agent reaches the other side of the road, it is given a reward of +1. This will help the agent to make better choices to reach it's goal, as the choice of the rewards is greatly impacting the optimal policy. An evident example leading to a wrong optimal policy can easily be found with Freeway. If a reward of 0.5 was granted when nothing happens, the agent may tend to stay forever in the middle of the road while avoiding the cars.

For now, we can simply define the **horizon** as the time the agent will be able to spend in the environment before ending the game : if the horizon is infinite, the agent will make sure to maximize the reward, no matter how long it takes to reach it. On the other hand, if the horizon is finite (and in particular short), the agent will hurry to reach the goal, but this might lead to taking some risks in the case of stochastic action spaces [2].

#### 2) Policy

We have discussed about the key concepts of RL, but what we are missing is how the agent will evolve regarding the rewards it obtains from the environment. To go further, we are going to define some functions that are going to lead us to the goal of making the agent intelligent. To illustrate this, we will carry on our explanations on the environment Figure 1.

A policy is a function that takes the state $s$ as it's input, and outputs the action to take. In the case of the freeway, the input is a vector of size 210 x 160 x 3, each component being an integer in $[0; 255]$. The output is an action of the action space. The whole point of reinforcement learning is finding the policy working the best, i.e. that when taking a

sequence of actions following the policy, we maximize the utility. In this case, it simply means finding the policy that will make our agent reach the other side of the road. This is only possible by receiving a reward from the environment, at each step, and then update the policy to maximize the expectation iteratively.

*3) Exploration and exploitation*

We will conclude the Freeway example by explaining the exploration/exploitation trade-off. When an agent is learning, it can stick to the policy, or in other terms, **exploit** what it thinks works best, but sometimes it can be interesting to take an action that is not the output of the policy at that time : this is called **exploration**. One way to choose between exploration and exploitation is via a method called $\epsilon - greedy$ : the agent will randomly choose an action, that is different than the one the policy outputs, with a probability $\epsilon$ at each step. There are other approaches to exploration, different of picking an action randomly, such as using confidence bounds of the value-action. In fact, whatever is the exploration policy, it has been demonstrated it is necessary to train correctly the agent. The reason is, if it always follows its policy, it can stick to a local optimum in the environment, for example always trying to cross the road without stopping. A better and less risky solution would be to change lane by moving forward or backward only when a car is coming, to a lane where there is not, in order to reach the other side iteratively. But the agent cannot know that at the beginning of its training and has to try random actions sometimes to hopefully learn to prefer the second option.

## E. Naive approaches

In this section, we present two naive approaches representing two categories of agents : model-based or model-free. In RL, most of agents are model-free, as the environment are too complex to model, but we present both principles to understand the two possibilities.

*1) Adaptive dynamic programming*

Thanks to all of these concepts, the chapter 21 of [2] follows the explanation. The idea is then to take advantage of the percepts of the environment to compute the transition model. We will consider the environment observable as it almost works the same way for POMDPs and for MDPs as we saw. The objective is to estimate the value function for every state of the environment, using the Bellman equations $U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s'))$ for the optimal policy and in particular $U(s) = R(s) + \gamma \max_{a=\pi(s)} \sum_{s'} P(s'|s,a)U(s'))$ for the policy $\pi$. To do this, it tries to approximate $P(s'|s,a)$ just by sampled states from $\pi$.

**function** PASSIVE-ADP-AGENT($percept$) **returns** an action
  **inputs**: $percept$, a percept indicating the current state $s'$ and reward signal $r'$
  **persistent**: $\pi$, a fixed policy
    $mdp$, an MDP with model $P$, rewards $R$, discount $\gamma$
    $U$, a table of utilities, initially empty
    $N_{sa}$, a table of frequencies for state–action pairs, initially zero
    $N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
    $s$, $a$, the previous state and action, initially null

  **if** $s'$ is new **then** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
  **if** $s$ is not null **then**
    increment $N_{sa}[s,a]$ and $N_{s'|sa}[s',s,a]$
    **for each** $t$ such that $N_{s'|sa}[t,s,a]$ is nonzero **do**
      $P(t \mid s,a) \leftarrow N_{s'|sa}[t,s,a] \,/\, N_{sa}[s,a]$
  $U \leftarrow$ POLICY-EVALUATION($\pi, U, mdp$)
  **if** $s'$.TERMINAL? **then** $s,a \leftarrow$ null **else** $s,a \leftarrow s', \pi[s']$
  **return** $a$

Fig. 2. Passive RL agent based on ADP. The POLICY-EVALUATION function computes $U(s)$ using the Bellman equation [2]

As we can see in the Figure 2, it is really simple to compute $P(s'|s,a)$ as it is the number of times we got to $s'$ from $s$ relatively to the number of times we have been in $s$ doing the action $a$. However, in real-life problems, where we can have millions of potential states, it is easy to understand this solution is intractable as it would need a running value for each combination of $s', s, a$ and $s, a$.

*2) Temporal-difference (TD)*

Temporal-difference is another way, using the Bellman update equation, to estimate the value function as $U(s) = R(s) + \gamma U(s')$. It uses difference in utilities functions between successive states (when an agent moves from $s$ to $s'$) without having to compute the transition model :

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$$

with $\alpha$ the learning rate, and the term between parenthesis the estimated value error.

**function** PASSIVE-TD-AGENT($percept$) **returns** an action
  **inputs**: $percept$, a percept indicating the current state $s'$ and reward signal $r'$
  **persistent**: $\pi$, a fixed policy
    $U$, a table of utilities, initially empty
    $N_s$, a table of frequencies for states, initially zero
    $s$, $a$, $r$, the previous state, action, and reward, initially null

  **if** $s'$ is new **then** $U[s'] \leftarrow r'$
  **if** $s$ is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma\, U[s'] - U[s])$
  **if** $s'$.TERMINAL? **then** $s,a,r \leftarrow$ null **else** $s,a,r \leftarrow s', \pi[s'], r'$
  **return** $a$

Fig. 3. Passive RL agent based on TD [2]

The advantages compared to dynamic programming are firstly that the update only involves the only next state $s'$, instead of all states $s \in \mathbf{S}$. Secondly, it is much simpler to implement and requires less computation per observation. Although, it takes longer to converge than ADP, and only makes one adjustment per observed transition.

## F. Generalization

Finally, from the chapter 17 and 21 of [2], we have explained the notions at the core of RL but for the moment

we did not talk about any NN. Knowing how to react facing a situation we have already encountered is one thing but the agent needs to know how to react even in situations it has not encountered yet. It is easy to notice we will be rapidly limited by the capacity of the computer if we just remember how to react to a previous situation each time we see it. It is impossible to implement this on games with millions of possible states such as chess or backgammon. Moreover, just visiting more than $10^{20}$ possibilities is absurd, so we need to handle this using an approximation function for utility, or Q-function:

$$U_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + ... + \theta_n f_n(s) \quad (1)$$

Here $U_\theta(s)$ approximates the true utility function with respect to its parameters $\theta_i$ to compress $10^{20}$ states to, for example, $n = 20$ parameters. By varying the value of theses parameters, the agent will try to find a good approximation making it perform great during the game for example. Taking a larger hypothesis space will increase the likelihood that a good approximation can be found, but also means that convergence will likely be delayed. For reinforcement learning, it makes sense to use an online learning algorithm that updates every parameter $\theta_i$ after each trial. For this matter, we compute them with a Widrow-Hoff rule (or the delta rule) that adds every step an error defined as the squared difference of the predicted total reward $U_\theta(s)$ and the actual total reward observed $u_j(s)$ from the state $s$ onward, $E_j(s) = (U_\theta(s) - u_j(s))^2/2$, as:

$$\theta_i \leftarrow \theta_i + \alpha \frac{\partial E_j(s)}{\partial \theta_i} \quad (2)$$

By selecting this approximation, we allow a reinforcement learner to generalize from its experiences. We can apply these ideas just as well to temporal-difference (TD) learners. The only thing to do it to adjust the parameters that would reduce the temporal difference between successive states.

### G. Types of agents

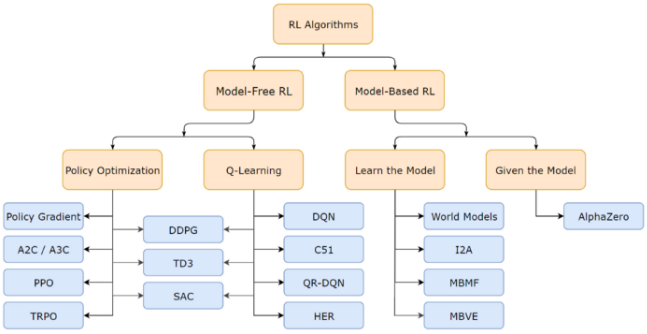


Fig. 4. Types of agents [4]

## III. Q-Learning

As we know, the terms model-free or model-based simply refer to whether the agent uses predictions of the environment response during learning or acting and builds transition model : $P[s_{t+1}|s_t, a_t]$. In the model-free family we can find Q-learning methods where the algorithms learn an approximation $Q_\theta(s, a)$ of the optimal action-value function $Q^*(s, a)$ such as $U^*(s) = \max_a Q^*(s, a)$ by using deep convolutional filters [5].

### A. Deep Q Network (DQN)

To apply RL to real-life situations, agents must derive efficient representations of the environment from high-dimensional sensory inputs and use these to generalize past experiences to new situations. RL agents have shown great results in some domains with fully observed and low-dimensional state spaces. However, DQN agents can learn policies directly from high-dimensional sensory inputs and take actions to never seen before situations by identifying them to previous ones and act accordingly.

Deep Q-Learning is a general-purpose algorithm combining Q-Learning with deep neural networks, allowing the agent to work with high-dimensional environments. Instead of a Q Table of values to look up and update, we have a model which we use to make predictions and train. This regression model gives a continuous float output value of the Q-function for each of the possible actions possible. Knowing which action corresponds to each output, the agent then selects the action giving the output maximizing the cumulative future reward : $\arg\max_{a_t} Q_\theta(a_t, s_t)$. Unfortunately, divergence or instability caused by the correlation and non-stationary distributions of input data samples may arise. To counter this and take in account that small changes to Q may completely change the policy, the DQN algorithm uses an experience replay mechanism, randomly sampling previous transitions from an experience buffer. An iterative update uses these to adjust the action-values towards target values as :

$$Q^*(s, a) = \max_\pi \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi] \quad (3)$$

$Q^*(s, a)$ following the Bellman equation and giving the maximum expected return achievable. Following some strategy after a sequence of states $s$ and then taking an action $a$, is a policy mapping sequences to actions.

To approximate the action-value function, a neural network $Q_\theta(s, a) = Q^*(s, a)$ is used with the parameters $\theta$, in other words weights, of the Q-network. As mentioned, to train a it, updates have to be applied on samples of experience. The Q-learning update follows a loss function $L_i(\theta_i)$ that needs to be minimized to train the Q-network at each iteration i.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a, \theta_i))^2] \quad (4)$$

With $\gamma$ the discount factor, $\theta_i$ the parameters of the Q-network at iteration i used to compute the target at iteration i.

The DQN agent has proven its worth through various applications. The most famous is probably the application on the Atari 2600 games [6]. By using pixels and the game score as inputs, the agent has surpassed all previous algorithms and

achieved performance equal or superior to the best human players.

Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
**For** episode = 1, $M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  **For** $t$ = 1,T **do**
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  **End For**
**End For**

Fig. 5. DQN algorithm [6]

Noted in the algorithm on Figure 5, the action is selected thanks to an $\epsilon$-greedy policy to balance between exploration and exploitation. Also, the state considered here is not directly the inputs (raw pixels and score) as it does not give enough information on the true state of the environment (POMDP problem). Rather, a function of the history is given to the network (difference of multiple frames) to augment the quality of information received. Finally, $N$ random samples are selected from the history $\mathcal{D}$ to perform a minibatch update.

### *B. Dueling DQN*

The dueling architecture has the same conventional layers as the single-stream network, but the latter consists of two streams that represent the value and advantage functions while sharing a common convolutional feature learning module. This approach allows the network to be combined with both existing and future algorithms for reinforcement learning.

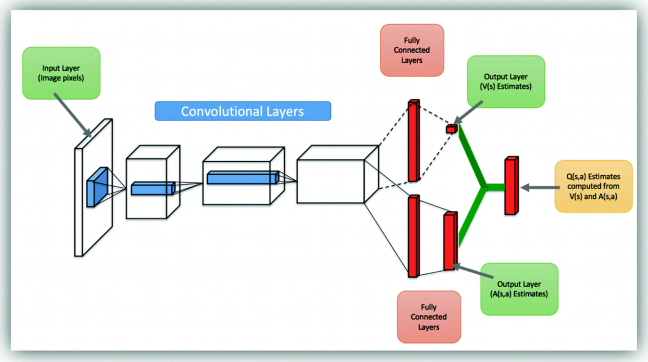


Fig. 6. Schematic Dueling Q network [7]

The value stream has a single output neuron whereas the advantage stream has a number of output neurons equal to the number of actions available to the agent. The two streams are combined by a special aggregating layer to produce an estimate of the state-action value function Q. The advantage function is an important value corresponding to the benefit of doing a specific action $a$ relatively to the expected reward-to-go for a specific state $a$. The advantage function is computed as:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{5}$$

From the advantage and the value function, it is easy to compute the Q-function, however in our case, we only have estimates of these quantities. Moreover, it is, of course, not possible to recover A(s, a) or V(s) only from the value of Q(s, a) as one compensates the other. It is then possible to have a good approximation of Q from terrible approximations of $A$ and $V$ that compensate each other. This leads to poor performance using this equation directly as the network is not able to compute these values separately on the output of the two streams.

To counter this issue, it is possible to force the advantage function to zero for the best action. Thanks to this constraint, it is easy to understand we will be sure to obtain the good value for the value function in this case. It incites the network to compute the real estimations of $A$ and $V$ and as a result the equation becomes identifiable. Thus, in the last layer, the parameterized estimate of the true Q-function for a given state-action combination $Q(s,a;\theta,\alpha,\beta)$ is the approximate value of that state $V(s;\theta,\beta)$, summed up with the approximate advantage value $A(s;\theta,\alpha)$ minus the maximum of advantage values. Here, $\theta$ represents the parameters of the convolutional layers while $\alpha$ and $\beta$ are the parameters of the two streams of fully-connected layers [8].

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \max_{a' \in |\mathcal{A}|} A(s,a';\theta,\alpha)) \tag{6}$$

However, the first experiences highlighted the difficulty for the advantage function to follow the rapid changes due to its maximum value. Alternatively to using a softmax function, this version was replaced with a mean advantage value as it was simpler to implement and gives similar results :

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)) \tag{7}$$

The forward step of the dueling network architecture doesn't provide an explicit benefit in comparison to the single stream Deep-Q network. If we assume that both represent the optimal Q function, the benefit of the dueling network architecture reveals itself during training. The reason is that the Q-error in deep Q networks is based on the temporal difference error. This means that we know how much only one of the output Q values should be changed because we don't know how much reward the agent can get while performing other actions in the given state. Therefore, only one output neuron is contributing

to backpropagation per training sample. Moreover, the dueling architecture can learn which states are valuable without having to learn the effect of each action for each state.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha y \tag{8}$$

With $y = r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)$ the temporal difference error.

As the deep Q network architecture has $n$ output neurons for $n$ different actions which an agent can perform in an environment, the increasing number of actions results in the training of the neuron network becoming more sparse and biased in the last layers. The separation into two streams of state value and advantage value for each action allows learning of the state value function efficiently with every update of the Q values. And the separate advantage stream makes the neural network robust to reordering of actions with the highest Q value due to small amounts of noise in the neural network updates.

To train a Dueling Q-network, Q-learning updates have to be applied on samples of experience. The Q-learning update follows a loss function $L_i(\theta_i)$ that needs to be minimized to train the Q-network at each iteration $i$ just like with the simple DQN.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)}[(y_i^{DDQN} - Q(s, a; \theta_i))^2] \tag{9}$$

With :

$$y_i^{DDQN} = r + \gamma Q(s', arg_{a'} max Q(s', a'; \theta_i); \theta^-) \tag{10}$$

Where $\theta^-$ represents the parameters of a fixed and separate target network.

In [9], an exhaustive comparison between several improvements to the deep Q network algorithm demonstrated that the performance of the dueling network architecture, gets a higher score and exhibits faster learning in Atari games than the classic single stream deep Q network.

**input** : $\mathcal{D}$ – empty replay buffer; $\theta$ – initial network parameters, $\theta^-$ – copy of $\theta$
**input** : $N_r$ – replay buffer maximum size; $N_b$ – training batch size; $N^-$ – target network replacement freq.
**for** *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  **for** $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame $x^t$ from environment $\mathcal{E}$ given $(s, a)$ and receive reward $r$, and append $x^t$ to $\mathbf{x}$
    **if** $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end**
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple $(s, a, r, s')$ to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of $N_b$ tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the $N_b$ tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every $N^-$ steps
  **end**
**end**

Fig. 7. Dueling DQN algorithm [9]

### C. Categorical DQN

Usually, in RL, the agent tries to compute the Q-value, which is an expectation. In general, it has to maximize its value and to do so, the Bellman's equation is used to approximate it. Instead, in the Categorical DQN, the agent predicts directly the distribution of this value.

$$Z(x, a) =^D R(x, a) + \gamma Z(x', a') \tag{11}$$

This distributional Bellman equation depends on the interaction between the reward R of the next state action-pair $(x', a')$ and its random returned distribution $Z(x', a')$. It appears this approach makes RL significantly better-behaved as it is learning the distribution of the reward-to-go and not only a single value, its expectation : $Z^\pi(x, a) = \sum_{t=}^{\infty} \gamma^t R(x_t, a_t)$ with $x_t \sim P(.|x_{t-}, a_{t-})$ and $a_t \sim \pi(.|x_t), x_0 = x, a_0 = a$.

We recall that the interaction between the agent and its environment is modeled by a time-homogeneous MDP. For a given state space $X$ and action space $A$, the agent chooses an action depending on its current state and receives a reward coming from its environment which then takes the next state depending on the selected action.

Categorical DQN has an architecture based on DQN but the output layer predicts the distribution of the returns for each action instead of its mean. For this reason, each action is represented by $N$ output neurons encoding the support of the discrete distribution of returns. If the return is bounded between two borders then the distribution can be represented by $N$ discrete bars or "atoms" as they are referred to in the article [10]. This discrete approximation of the distribution is thus computationally friendly and highly expressive.

From their experiments, the authors found the most efficient number of atoms to be 51. An atom is noted as $z_i = V_{min} + i\Delta z : 0 \leq i < N$ with $\Delta z = \frac{V_{max} - V_{min}}{N-1}$. The probability that the return of a state-action pair $(s, a)$ lies within the bar associated to the atom $z_i$ is $p_i(s, a)$, given by a parametric model $\theta : \mathcal{X}.\mathcal{A} \rightarrow \mathbb{R}^N$.

$$p_i(s, a; \theta) = \frac{e^{\theta_i(s,a)}}{\sum_{j=1}^{N} e^{\theta_j(s,a)}} \tag{12}$$

Now the model is approximating $p_i(s, a; \theta)$, the agent possess a more complex and detailed point of view to select which action is the better. As the network outputs the atom's probabilities for each action, it only needs to take the one yielding the highest reward-to-go. Therefor, similarly to the aforementioned approaches, it searches for $\arg\max_a Q_\theta(s, a)$ computed following :

$$Q_\theta(s, a) = \mathbb{E}[Z_\theta(s, a)] = \sum_{i=1}^{N} z_i p_i(s, a; \theta) \tag{13}$$

Alternatively to using a typical loss function $(r + \gamma Q(x', \pi(x')) - Q(x, a))^2$, the categorical algorithm uses a cross-entropy loss, which is better suited in this case. Specifically, though we will not describe these theories here, the

KL divergence is a common way to statistically measure the distance between two distributions $p$ and $q$ as :

$$D_{KL}(p||q) = \mathbb{E}_{x\sim p}[\log \frac{p(x)}{q(x)}] \tag{14}$$
$$= \int p(x) \log \frac{p(x)}{q(x)} dx$$

In a supervised learning case, the objective would be to minimize $D_{KL}(p(x), q(x))$ for $p(x)$ distribution of a labeled sample of data and $q(x)$ the network's prediction. Reaching the value 0, means we have the same distribution and the network is perfectly predicting how the data is distributed. In this situation, the goal is to minimize the divergence between the projected update (not detailed in the article) and its current value $D_{KL}(\Phi\hat{\mathcal{T}}Z_\theta(x,a)||Z_\theta(x,a))$ using gradient descent. The categorical algorithm returns this value, used to update the network, and is computed as follows :

**input** A transition $x_t, a_t, r_t, x_{t+1}, \gamma_t \in [0,1]$
  $Q(x_{t+1}, a) := \sum_i z_i p_i(x_{t+1}, a)$
  $a^* \leftarrow \arg\max_a Q(x_{t+1}, a)$
  $m_i = 0, \quad i \in 0, \ldots, N-1$
  **for** $j \in 0, \ldots, N-1$ **do**
    # Compute the projection of $\hat{\mathcal{T}} z_j$ onto the support $\{z_i\}$
    $\hat{\mathcal{T}} z_j \leftarrow [r_t + \gamma_t z_j]_{V_{\text{MIN}}}^{V_{\text{MAX}}}$
    $b_j \leftarrow (\hat{\mathcal{T}} z_j - V_{\text{MIN}})/\Delta z \quad \# \, b_j \in [0, N-1]$
    $l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$
    # Distribute probability of $\hat{\mathcal{T}} z_j$
    $m_l \leftarrow m_l + p_j(x_{t+1}, a^*)(u - b_j)$
    $m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$
  **end for**
**output** $-\sum_i m_i \log p_i(x_t, a_t)$ # Cross-entropy loss

Fig. 8. Categorical DQN algorithm [10]

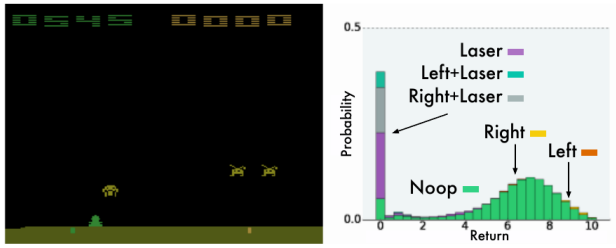


Fig. 9. Value distribution during an episode of SPACE INVADERS with a different color for each action [10]

Drawn from the paper [10], Figure 9 is an example of learned value distribution. Each bar represents the probability $p_i(s,a)$ to received the value $z_i$ for different actions, in different colors. On this frame we can see the actions involving the laser have a high probability to result in a null reward, in other words lose the game. Deciding to fire now is a risky decision as the enemy is moving fast and it will have the time to pass over the agent before it can shoot again. Alternatively, we can observe doing nothing, or just moving in a direction, seems to be a good option and is in fact what many players would do, waiting for the enemy to be almost above to shoot. Therefore, similarly to a human player, the network will choose between one of these two options and proves it understood how to game works.

## IV. Actor-Critic

The way in which DeepQ works by assigning a score to each possible action for the agent in the environment to guide it to the best result possible. This method is very efficient, but unfortunately bound to finite action spaces, as it would be impossible to assign an infinite number of scores in a continuous action space. To work in these complex environments, we need a very different type of algorithm : this is when actor-critic methods come into play.

At the core of every actor-critic method are two models that work together on the same problem. One of them, the actor, takes the environment as its input, computes the agent's policy and returns the probability for each action to be executed in a continuous space. The other model, the critic, takes this action and the environment as inputs, computes the value function and returns a score representing an evaluation of the decision, guiding the actor to make better and better decisions as it learns itself to judge them more accurately each step.

As a tool to understand this method, we can use the well-known and simple analogy of a small child in a park. At first, the child does not know what he can or cannot do and relies entirely on his parents for guidance. The actions he could choose are infinite : crawl on the floor, play on a swing, eat rocks or mud, run in the grass, etc. As the child takes an action, his parents' role is to observe its consequences and give feedback so that their child does not get hurt or cause havoc in the park. While positive actions will be reinforced with positive feedback, negative actions will be discouraged with negative feedback. This feedback is equivalent to the score attributed to the actor by the critic. By moving away from actions that generate negative feedback and towards actions that result in the more positive feedback, the child will learn to interact properly with his environment. This analogy is basic and therefore lacks the nuance that will be provided in the rest of this section, but it can be a useful tool to grasp a basic of actor-critic methods.

As in other methods, we will need derivatives to compute the parameters' updates. With two networks connected to each other, the chain rule will be very useful, we can simply express it as follows:

$$f'(x) = g(h(x))' = g'(h(x))h'(x) \tag{15}$$

Now that we are trying to maximize the scores calculated by the critic, actor-critic methods will typically use gradient

ascent instead of gradient descent, aiming at a global maximum instead of a minimum. The direction is given by the chain rule, which will be used to compute gradients and the weights in the network, which will be updated at each time step and not at the end of each episode as opposed to policy gradients. Tuning the actor's parameters will change the Q-value and so the output of this model is considered as the intermediary between the actor and the critic. We will now have to work with two cost functions, one for updating the critic and another one for updating the actor. One of the great strengths of this type of model is that they typically converge in very few iterations and need little training time compared to other methods. They have been proven to be able to work in extremely complex environment and tested thoroughly especially in games, whether it be 2D or 3D video games, where they performed better than most other models. The reasons behind these impressive performances will be discussed by focusing on some of the most well-known and more efficient algorithms to implement actor-critic methods. [11]

### *A. A3C*

Developed by DeepMind in 2016, A3C (Asynchronous Advantage Actor-Critic) is one of the most recent in the field of deep reinforcement learning and its creation quickly rendered several other algorithms such as policy gradient and DQN obsolete. This algorithm is composed of multiple actor-critic agents that all work with different parameters and independently from each other under the supervision of a global network. Since the algorithm is asynchronous, the weights of the global network are updated by all the other agents at the end of their own episode. When the global parameters are updated, all agents update their own parameters to match them, then continue learning independently until the next update, which means that all agents are connected not only to the global network but also to each other since they serve to update the parameters of the whole network. This architecture allows the agents to learn with very different training data from one another, contributing to bringing new and valuable information to the global network.

To understand the importance of this way of functioning, using an analogy can be very useful [12]: each agent is seen as a human whose experience contributes to the global knowledge. Contributing with experience that is very different from those of others is much more valuable than if we were to all get a similar experience of life (similar training data). This diversification allows the algorithm to perform better than standard techniques. Executing these multiple agents in parallel decorrelates the data, since at a given time they will all be training on different data and therefore experiencing different states. It is this asynchronicity that enables the use of on-policy algorithms like A3C to work in reinforcement learning with deep neural networks ; with only one network, the samples would be too correlated for an on-policy algorithm. Using multiple networks also allows the algorithm to explore a bigger portion of space in less time than other algorithms [13]. Asynchronous methods for deep reinforcement learning A3C can be described with the following diagram :

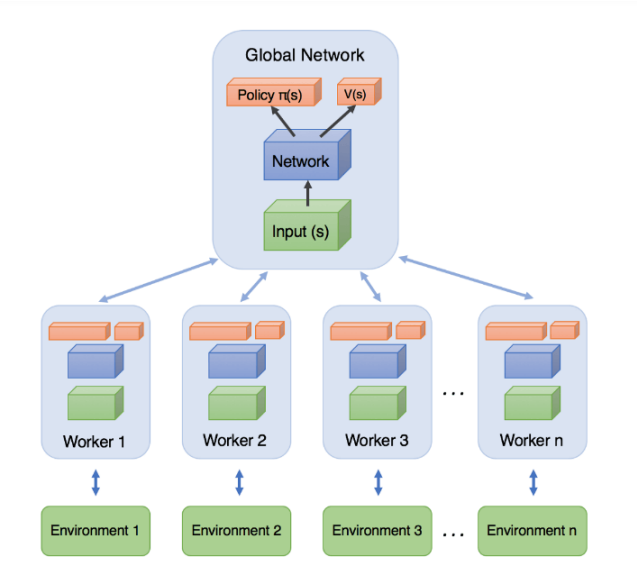


Fig. 10. Architecture of A3C [14]

The following pseudocode for the algorithm describes the functioning of each network in the system :

**Algorithm S3** Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

*// Assume global shared parameter vectors* $\theta$ *and* $\theta_v$ *and global shared counter* $T = 0$
*// Assume thread-specific parameter vectors* $\theta'$ *and* $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
**repeat**
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state $s_t$
  **repeat**
    Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
    Receive reward $r_t$ and new state $s_{t+1}$
    $t \leftarrow t+1$
    $T \leftarrow T+1$
  **until** terminal $s_t$ **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  **for** $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
  **end for**
  Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
**until** $T > T_{max}$

Fig. 11. A3C algorithm [13]

Moreover, the algorithm works by predicting both the value function and the optimal policy function. The value function, computed by the critic, $\hat{q}(s, a, w)$ is used to update the optimal policy function, computed by the actor, $\pi(s, a, \theta)$. The actor and the critic train separately, using gradient ascent so as to find a global maximum and reach the best action with the highest probability. That way, the actor learns the best policy while the critic learns to evaluate the actor better until reaching an optimum.

The "advantage" part of the algorithm concerns the implementation of policy gradient. Instead of using the value of

discounted returns to tell the agent which action is rewarded and which is penalized, it uses the value of advantage, meaning that the agent knows what to expect from an action and learns how much better or worse than expected each action turns out to be. The advantage is computed with the following equation [12]:

$$Advantage = Q(s,a) - V(s) \tag{16}$$

Unlike in the classical actor-critic algorithm, an update of the parameters in A3C updates parameters asynchronously with accumulated gradients. The process functions in forward view : the agent's exploration policy selects actions until a maximum number ($t_{max}$ fixed) or until a terminal state is reached. When this process is over, the agent receives $t_{max}$ rewards for its actions and the policy and the value function are both updated.

The training of several actors in parallel using this method helps improve stability. The policy is typically computed with a softmax function whereas the value function is computed with a linear output. These two outputs share all the previous layers, therefore the network is working to compute both functions at the same time. William and Peng also found in 1991 that adding the entropy of the policy to the objective function improved exploration by avoiding local optima. A3C is widely considered to be one of the best reinforcement learning agent to this day. Its functioning allows it to use less resource than other approaches and to perform in both continuous and discrete action spaces while still surpassing these older algorithms. This graph shows a comparison between A3C and a few other deep learning algorithms on five Atari games:

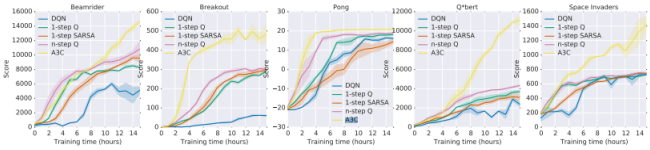


Fig. 12. Comparison of performance in various video games [13]

It is very clear here that A3C constantly surpasses the other tested methods in this example. This specific algorithm is very efficient for applications similar to video games and has the advantage of being adaptable to both discrete and continuous action spaces.

### B. A2C

A2C (Advantage Actor-Critic) is simply a synchronous version of A3C created as an improvement of the latter, it delivers similar performance while being more efficient.

Q-values can be separated in value and advantage functions: $Q(s,a) = V(s) + A(s,a)$ with the advantage $A(s,a) = Q(s,a)–V(s)$, which can be changed for $A(s,a) = r + \gamma V(\hat{s}) - V(s)$. This last equation allows the action value and the state value to be computed by one network, this way we only need one network to compute the advantage.

As in A3C, advantage is used to estimate not only how good an action is (that is the role of the value function) but how much better it is compared to other possible actions in the present state. In A2C, the critic works by learning the advantage values instead of the Q-values. This specificity stabilizes the model and reduces the variance. The problem with the asynchronous nature of A3C is that, since the agents are working independently and interacting with the global parameters, the different agents could have different policy versions at the same time, which would result in a sub-optimal update of the parameters.

Since A2C is a synchronous algorithm, it waits for each network to finish its episode before updating the global network using an average of the outputs from the other networks. All agents can then start a new iteration with the same policy, which helps the algorithm converge. This way of functioning gets rid of the instability of A3C, uses CPU more efficiently and allows training with bigger batch sizes to be more effective, which produces an algorithm that is sometimes considered to be equivalent or even more effective than its asynchronous counterpart [15]. The difference between the two is shown in the following diagram :

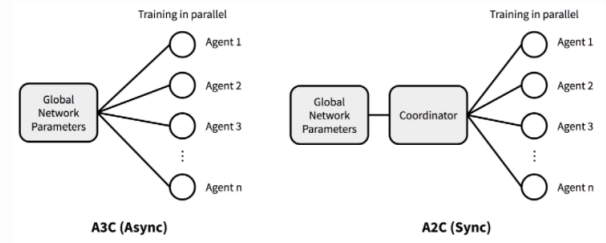


Fig. 13. Visualization of A3C and A2C algorithms [16]

Like A3C, A2C is an on-policy algorithm, the network plays by learning the value function for the same policy that it is following. That is the reason why the updates seem to be working in a very similar way in both methods.

### C. ACER

The goal of ACER (Actor-Critic with Experience Replay) is to work on both discrete and continuous environments like A3C does, but also to do so with better sample efficiency. Experience replay results in a reduce in non-stationarity and a decorrelation of updates, but unfortunately, this algorithm limits the methods to off-policy algorithm and uses more memory and computation than the algorithms previously discussed in this section [17].

ACER can be seen as a sample efficient version of A3C, therefore most of the features present in A3C are also present in ACER, including parallel computation with an efficient use of CPU power. A single network is used to compute the value function and the policy for which most parameters are shared by the unique network.

At each time step, the algorithm chooses an action $a$ by observing the state vector, following its policy $p_i(a, x_i)$. The environment receiving the action produces a reward that can be continuous or discrete. The goal of the agent is of course to maximize the expected return, and therefore the parameters of the policy are updated with the discounted approximation to the policy gradient (Sutton et al., 2000) detailed here [18] :

$$g = E_{x_{0:\infty}, a_{0:\infty}} \left[ \sum_{t \geq 0} A^{\pi}(x_t, a_t) \nabla_{\theta} log \pi_{\theta}(a_t \mid x_t) \right]. \quad (17)$$

**Algorithm 3** ACER for Continuous Actions

Reset gradients $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
Initialize parameters $\theta' \leftarrow \theta$ and $\theta'_v \leftarrow \theta_v$.
Sample the trajectory $\{x_0, a_0, r_0, \mu(\cdot|x_0), \cdots, x_k, a_k, r_k, \mu(\cdot|x_k)\}$ from the replay memory.
**for** $i \in \{0, \cdots, k\}$ **do**
  Compute $f(\cdot|\phi_{\theta'}(x_i))$, $V_{\theta'_v}(x_i)$, $\widetilde{Q}_{\theta'_v}(x_i, a_i)$, and $f(\cdot|\phi_{\theta_a}(x_i))$.
  Sample $a'_i \sim f(\cdot|\phi_{\theta'}(x_i))$
  $\rho_i \leftarrow \frac{f(a_i|\phi_{\theta'}(x_i))}{\mu(a_i|x_i)}$ and $\rho'_i \leftarrow \frac{f(a'_i|\phi_{\theta'}(x_i))}{\mu(a'_i|x_i)}$
  $c_i \leftarrow \min\left\{1, (\rho_i)^{\frac{1}{d}}\right\}$.
**end for**
$Q^{ret} \leftarrow \begin{cases} 0 & \text{for terminal } x_k \\ V_{\theta'_v}(x_k) & \text{otherwise} \end{cases}$
$Q^{opc} \leftarrow Q^{ret}$
**for** $i \in \{k-1, \cdots, 0\}$ **do**
  $Q^{ret} \leftarrow r_i + \gamma Q^{ret}$
  $Q^{opc} \leftarrow r_i + \gamma Q^{opc}$
  Computing quantities needed for trust region updating:

$$\begin{aligned} g &\leftarrow \min\{c, \rho_i\} \nabla_{\phi_{\theta'}(x_i)} \log f(a_i|\phi_{\theta'}(x_i)) \left(Q^{opc}(x_i, a_i) - V_{\theta'_v}(x_i)\right) \\ &+ \left[1 - \frac{c}{\rho'_i}\right]_+ (\widetilde{Q}_{\theta'_v}(x_i, a'_i) - V_{\theta'_v}(x_i)) \nabla_{\phi_{\theta'}(x_i)} \log f(a'_i|\phi_{\theta'}(x_i)) \\ k &\leftarrow \nabla_{\phi_{\theta'}(x_i)} D_{KL}\left[f(\cdot|\phi_{\theta_a}(x_i) \| f(\cdot|\phi_{\theta'}(x_i)\right] \end{aligned}$$

  Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial \phi_{\theta'}(x_i)}{\partial \theta'} \left(g - \max\left\{0, \frac{k^T g - \delta}{\|k\|_2^2}\right\} k\right)$
  Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + (Q^{ret} - \widetilde{Q}_{\theta'_v}(x_i, a_i)) \nabla_{\theta'_v} \widetilde{Q}_{\theta'_v}(x_i, a_i)$
    $d\theta_v \leftarrow d\theta_v + \min\{1, \rho_i\} \left(Q^{ret}(x_t, a_i) - \widetilde{Q}_{\theta'_v}(x_t, a_i)\right) \nabla_{\theta'_v} V_{\theta'_v}(x_i)$
  Update Retrace target: $Q^{ret} \leftarrow c_i \left(Q^{ret} - \widetilde{Q}_{\theta'_v}(x_i, a_i)\right) + V_{\theta'_v}(x_i)$
  Update Retrace target: $Q^{opc} \leftarrow \left(Q^{opc} - \widetilde{Q}_{\theta'_v}(x_i, a_i)\right) + V_{\theta'_v}(x_i)$
**end for**
Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
Updating the average policy network: $\theta_a \leftarrow \alpha\theta_a + (1-\alpha)\theta$

Fig. 14. ACER for continuous action [18]

This algorithm is considered to be a hybrid between on and off-policy gradients, using the off-policy Retrace algorithm instead of Q-learning to train the critic. Instead of simply using the TD error to update the values, Retrace generalizes eligibility trace in the forward view. Eligibility trace allows us to take into consideration $\delta_{t'}$, the TD error of future transitions as well as the current one, with lambda assuring stability. $\Delta Q^{\pi}(s_t, a_t) = \alpha \sum_{t'=t}^{T} (\gamma\lambda)^{t'-t} \delta_{t'}$

Using a parameter $c_s$ for each time step between t and t', Retrace generalizes this algorithm resulting in the formula: $\Delta Q^{\pi}(s_t, a_t) = \alpha \sum_{t'=t}^{T} (\gamma)^{t'-t} \left(\prod_{s=t+1}^{t'} c_s\right) \delta_{t'}$.

Adaptable according to the value of $c_s$, Retrace can be used in many reinforcement learning methods, including actor-critic algorithms, where it shows interesting results in terms of speed of learning and optimality compared to DQN. The way ACER works is that it learns on-policy (current policy) from a sample of a trajectory, stores it in its replay buffer, then sample a set number (n) of trajectories from the buffer and apply its off-policy on each of the samples. The parameters of the actor are updated through a Trust Region Policy Optimization algorithm. To do this, a target actor network $\theta'$ is used to track the actor $\theta : \theta' \leftarrow \alpha\theta' + (1-\alpha)\theta$.

The goal here is to update the actor slowly by reducing the gap between the new policy and its past values (defined by the target actor) and make the objective function for the optimization of the actor's parameter quadratic with a linear constraint, leading to a simple optimization problem, and therefore a simpler solution. Similarly, a target critic network $\phi$ can be used to compute the value of the states for variance reduction [19].

Like most algorithms discussed here, ACER can also be modified to suit different applications. For example, it is possible to augment ACER with stochastic activation, resulting in a new algorithm called SACER, or to randomize the value and policy network, which results in NN-ACER. These algorithms were tested in the application of CarRacing, a driving simulator, to compare their performances. The input for this test is composed of pixels that constitute the screen seen by the player and the action space is discretized. The results of this experiment show a significative improvement in performance [20] :

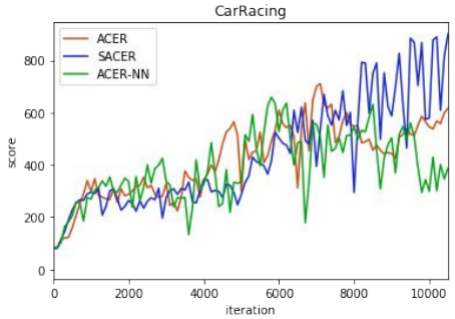


Fig. 15. Improvement of ACER's performance [20]

Uncertainty is crucial in the functioning of reinforcement learning, therefore it would make sense to try and introduce parametric noise in our algorithms, since this technique was successful for improving other types of algorithms. This technique is at the core of Stochastic Neural Networks and helps them improve exploration and reduce uncertainty during the training phase of the algorithm. However, parametric noise is not so successful when it comes to actor-critic methods. Stochastic activation is the actor-critic equivalent of parametric noise but can also be applied to many other actor-critic algorithms, including on-policy methods such as the previously discussed A3C (SA3C), resulting in a general improve of performance and a faster convergence [21].

### *D. DDPG*

Deep Deterministic Policy Gradient is also an Actor-Critic neural network. As detailed in [22] and presented here, the main idea behind DDPG is to resolve one of the major problem with DQN methods: it cannot be applied to continuous action spaces. In the examples we discussed, you have noticed that each output of the network is dedicated to

one action and give its Q-function. In the case of continuous action space, you could not have an infinite number of outputs, thus in order to select the best action, you would have to determine which one maximizes the Q-function. This would require solving an optimization problem at each iteration and would be too computationally expensive. Of course, a discretization of the action space has too many limitations to be a good solution in most situations.

Here is presented a model-free, off-policy actor-critic algorithm. However, the ideas behind DQN are kept and the reader will find here familiar topics from the DQN-based algorithms. The policy here maps states to a probability distribution over the actions $\pi : S \rightarrow \mathcal{P}(A)$. The reward is: $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$, and the goal is still to maximize $J = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1]$ as in this case the policy is probabilistic. Thus, as in DQN, the Q-function is computed as the expectation of the reward to go :

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_{i>t}, s_{i>t} \sim E, a_{i>t} \sim \pi}[R_t | s_t, a_t] \tag{18}$$

Using the Bellman equation, we get to :

$$\begin{aligned} Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) \\ + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q^{\pi}(s_{t+1}, a_{t+1})]] \end{aligned} \tag{19}$$

The key point here, is to consider the target policy as deterministic as $\mu : S \rightarrow A$. This way, the inner expectation is suppressed :

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, a_{t+1})] \tag{20}$$

The reader can notice the Q-function of the target policy only depends now on the environment. It means it is possible to learn $Q^{\mu}$ from transitions generated by an other policy than the optimal one. This point, really important as we will see, will allow to use a second network and its experience to improve the first one.

The loss function, as before, is a quadratic difference between the value predicted by the network for a state $s_t$ and the reward at state $s_t$ plus the value predicted for the state $s_{t+1}$ using the Bellman equation, the TD error :

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^{\beta}, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t | \theta^Q) - y_t)^2] \tag{21}$$

where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}) | \theta^Q) \tag{22}$$

As we know, this difference must be as low as possible as the Bellman equation needs to be respected and thus, it is a way to improve the network. The actor however, to find the best action to perform, searches for the actor's parameters maximizing the Q-function, will be updated with respect to its parameters $\theta^{\mu}$ thanks to :

$$\nabla_{\theta^{\mu}} J \approx \mathbb{E}_{s_t \sim \rho^{\beta}}[\nabla_{\theta^{\mu}} Q(s, a | \theta^Q) | s = s_t, a = \mu(s_t | \theta^{\mu})] \tag{23}$$

$$\begin{aligned} = \mathbb{E}_{s_t \sim \rho^{\beta}}[\nabla_a Q(s, a | \theta^Q) \nabla_{\theta_{\mu}} \mu(s | \theta^{\mu}) \\ | s = s_t, a = \mu(s_t)] \end{aligned} \tag{24}$$

As in DQN, an experience replay buffer is used to learn from mini-batches and samples are drawn randomly from the memory. Directly implementing this method led in the first experiments to an unstable network and a soft update of the parameters was needed as the optimal policy was never obtained and it kept oscillating around it. To solve this problem, a copy of the actor and the critic networks, $\mu'(s|\theta^{\mu'})$ and $Q'(s, a|\theta^{Q'}$, was created in order to update their weights more slowly at each step : $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ with $\tau \ll 1$.

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma$, $\beta$
**Output:** $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Fig. 16. Batch Normalizing Transform, applied to activation $x$ over a mini-batch [23]

The observations, from the different environments, may have really different ranges of values or physical units depending on the problem to be solved. This fact can lead the network to behave very differently from one problem to another if we keep the same hyper-parameters and eventually result in divergence. The authors therefore decided to scale the features using batch normalization [23] that is a standard method, not only to scales NN inputs, described in the Figure 16.

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^{\mu})$ with weights $\theta^Q$ and $\theta^{\mu}$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^{\mu}$
Initialize replay buffer $R$
**for** episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  **for** t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^{\mu}) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1 - \tau)\theta^{\mu'}$$

  **end for**
**end for**

Fig. 17. DDPG algorithm [22]

As it is explained in the algorithm, Figure 17, the critic

is not only updated using the Bellman equation on its own and the actor's predictions but is also using the predictions of the target critic and the target actor, which are evolving more slowly. The reason is we want to reduce the difference between the predictions of these networks, as we want both of them to converge to the optimal policy. When this goal will be reached, it will mean the targets are a good representation of the environment. To balance between exploitation and exploration, the policy is constructed adding noise to it during the training : $\mu'(s_t) = \mu(s_t|\theta_t^\mu)+\mathcal{N}$ with $\mathcal{N}$ chosen to suit the environment.

To synthesize, at each iteration, an action occurs between the agent an the environment, N samples are drawn randomly from experience and are used to update the actor and the critic. Then, a soft update of the target critic and target actor is done in the "direction" of the critic and actor's weights.

To conclude with this algorithm, all the problems solved by DDPG in the experiments of the authors took less than 2.5 million steps of experience, which is a factor 20 fewer steps than DQN and makes this agent one of the most efficient network available to this date.

### *E. SAC*

One of the main reasons why on-policy learning algorithms are sample inefficient is that they require new samples for every update of the policy such as A3C. Off-policy ones, as we saw in the last examples, use experience replay in the update process instead of fresh data. This one also uses it and is part of a kind of policy we did not talk about for the moment : maximum entropy policies. They are robust to errors and improve exploration by learning different behaviors in different situations.

This soft actor-critic, detailed in [24], uses 3 key concepts: it has a separate policy and value function network, it uses experience replay and aims at entropy maximization to encourage stability and exploration. As we discussed above, DDPG uses a Q-function estimator and a deterministic policy and this is specifically what makes it extremely difficult to configure. Even with the soft update, the choices of the hyper-parameters and the pre-treatment can negatively impact the agent's behaviour. As a consequence, its lack of stability makes it difficult to use in complex, high-dimensional tasks.

To begin with this part, we need some notations : $S$ is the state space, $A$ the action space, $p : SxSxA \rightarrow [0, \infty)$ is the probability density of $s_{t+1} \in S$ given $s_t \in S$ and $a_t \in A$. Very usual for now, but our objective is not exactly the same as before as we are dealing with maximum entropy policy :

$$\pi^* = \arg\max_\pi \sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t, a_t) + \alpha\mathcal{H}(\pi(.|s_t))] \quad (25)$$

The role of the optimal policy is not only to maximize the expectation of the sum of the rewards but also the entropy at each state. Here $\alpha$ is a parameter called the temperature that controls the importance of the entropy versus the reward at each state. The reader can easily understand that this hyper-parameter has a crucial importance as it determines how much the policy will be incited to explore the environment. Let's consider it a fixed hyper-parameter for the moment.

Repeatedly using the Bellman backup operator $\mathcal{T}^\pi$ as we already did for the value function of the Q-function, we get :

$$\mathcal{T}^\pi Q(s_t, a_t) = r(s_t, a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}(V(s_{t+1})) \quad (26)$$

where

$$V(s_t) = \mathbb{E}_{a_t\sim\pi}[Q(s_t, a_t) - \alpha log\pi(a_t|s_t)] \quad (27)$$

Using a different value-function than previously, the reward we can hope to get from the state $s_t$ depends on the expectation of the Q-function and on how much we want to consider if the policy is confident about this action or not (its entropy). If the probability is low, the logarithm will be negative and the entropy positive. This will increase the expectation and consequently encourage the policy to look in this direction. As we will see in this algorithm, it alternates between policy improvements and policy evaluation, and will probably converge, as detailed by the authors, if we restrain the policy to a set of policies will not discuss here.

Let's now consider the networks with the parameterized Q-function $Q_\theta(s_t, a_t)$ and policy $\pi_\phi(a_t|s_t)$ respectively with parameters $\theta$ and $\phi$. As in DDPG, the network can be improved by minimizing a similar loss function :

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim D}[\frac{1}{2}(Q(s_t, a_t|\theta^Q) - y_t)^2] \quad (28)$$

where

$$y_t = r(s_t, a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\theta}}(s_{t+1})] \quad (29)$$

The update makes use of a target soft Q-function with parameters $\bar{\theta}$ which has been shown to stabilize training. To improve the actor, the objective is to minimize the KL-divergence, also used for Categorical DQN in section III in a different fashion, we do not detail but results in :

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim D}[\mathbb{E}_{a_t\sim\pi_\phi}[\alpha log(\pi_\phi(a_t|s_t)) - Q_\theta(s_t, a_t)]] \quad (30)$$

After a reparametrization trick $a_t = f_\phi(\epsilon_t; s_t)$ where $\epsilon_t$ is a noise vector, we get the gradient:

$$\begin{aligned}&\nabla_\phi J_\pi(\phi) = \nabla_\phi\alpha\log(\pi_\phi(a_t|s_t)) \quad (31)\\ &+ (\nabla_{a_t}\alpha\log(\pi_\phi(a_t|s_t)) - \nabla_{a_t}Q(s_t, a_t))\nabla_\phi f_\phi(\epsilon_t; s_t)\end{aligned}$$

The reader will notice the last part of the gradient being similar to the gradient in the DDPG as we still want to improve the Q-function but also the entropy.

We considered the temperature constant but we also need to optimize it automatically as it is a parameter too difficult to settle for complex tasks. It needs to be tuned for each task and is non-trivial to find but crucial to build an efficient agent. The reason is, in regions where the entropy is high because of the

uncertainty of the action, it should let it free to explore. On contrary, it should be more deterministic when we are sure what is a good or a bad action. Resolving a maximization problem with minimum Lagrangian constraint, it is possible to compute a value for $\alpha$ at each iteration we will not detail here.

**Input:** $\theta_1, \theta_2, \phi$
  $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$
  $\mathcal{D} \leftarrow \emptyset$
  **for** each iteration **do**
    **for** each environment step **do**
      $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
      $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
    **end for**
    **for** each gradient step **do**
      $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
      $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
      $\bar{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1, 2\}$
    **end for**
  **end for**
**Output:** $\theta_1, \theta_2, \phi$

Fig. 18. SAC algorithm [24]

As aforementioned, two Q-functions are used to counter-balance bias and seemed, during experiments, to significantly improve speed on difficult tasks. They are trained separately and the minimum value of them is used to compute the gradients. Now that we defined the principles, the algorithm is uncomplicated. To synthesize, for $N$ environment steps, the agent will interact and store observations, actions and reward in its experience dataset. Then, for $M$ gradient steps, the networks $\theta_i$, $\phi_i$ and $\alpha$ will be updated thanks to gradient descent and the target networks thanks to soft update, as it was done for DDPG.

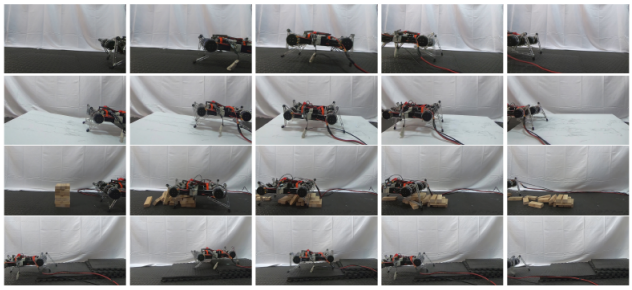

Fig. 19. Training of the Minitaur robot to walk on flat terrain (first row) and generalization to unseen situations (other rows) [24]

Finally, and more importantly, SAC was tested on OpenAI gym environments such as Hopper-v2, Walker2d-v2 or Humanoid(rllab) which are challenging continuous control tasks. It performed significantly better than DDPG, TD3 [25] or PPO [26] both with fixed or learned temperature in every environment. Even on quadrupedal locomotion in the real world, it was trained on flat terrain and yet was able to generalize, without additional learning for slopes, obstacles or stairs, with no difficulty. It took approximately 400 episodes of 500 steps, corresponding to two hours of training time in real-world, a short learning time for such a complex task.

## V. Hyper-parameter optimization

### A. Implicit Differentiation

At this point of our review, we did not say much about hyper-parameters. In a NN approach, they define how the agent or how the process of the optimization behaves and the choice of there values are crucial to generalize from the learned policy. In this part, we will detail two approaches and there limits thanks to [27].

Random and grid search allow the optimization of hyper-parameters and lead to good results in finding the best ones. However these approaches are very generic and face two limitations:

- these algorithms test various values hoping to find a local, or even a global optimum. Thus, they do not "understand" how, nor how much, changing the value in one direction could improve the results: this is called a black-box problem
- as long as there are between one and five dimensions, these approaches perform appropriately, however, when optimizing more than five, this performance goes down. As an alternative, Bayesian optimization leads to good improvements but also goes down for high-dimensional hyper-parameters optimization (>100 dimensions)

The reason, as we mentioned, comes from the incapacity of these algorithms to understand how to improve the agent.

A solution suggested in many papers is to perform a gradient-based optimization. In Figure 20, you can observe two phases: the training and the validation. During training, the objective is to minimize the loss function with respect to the weights of the network thanks to the various architectures we have seen so far for example. When we reach the validation, we can consider to have reached optimal weights and concentrate our work on minimizing the loss function with respect to the hyper-parameters.

What we described is a nested or bi-level optimization in which we search for the optimal hyper-parameters $\lambda^*$ and optimal weights $w^*(\lambda)$ as :

$$\lambda^* := \arg\min_\lambda \mathcal{L}_V^*(\lambda) \text{ where} \tag{32}$$

$$\mathcal{L}_V^*(\lambda) := \mathcal{L}_V(\lambda, w^*(\lambda)) \text{ and } w^*(\lambda) := \arg\min_w \mathcal{L}_T(\lambda, w) \tag{33}$$

What we can expect to do then is to use the gradient of the loss-function with respect to $\lambda$. However, in many configurations, the improvement not directly depends on $\lambda$ and is equal to zero as it falls into a saddle point. Hopefully it still depends indirectly on the weights of the network $w$. Therefore the gradient is extended to include this possibility as in Figure 21.

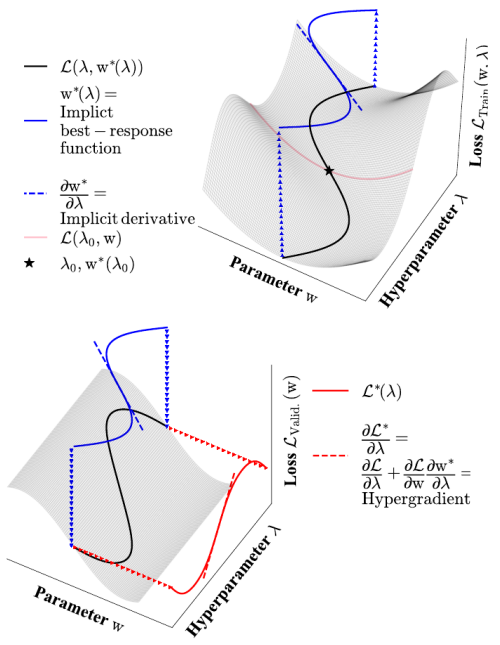


Fig. 20. Overview of the gradient-based hyper-parameter optimization [27]

$$\underbrace{\frac{\partial \mathcal{L}_V^*(\boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}}}_{\text{hypergradient}} = \left(\frac{\partial \mathcal{L}_V}{\partial \boldsymbol{\lambda}} + \frac{\partial \mathcal{L}_V}{\partial \mathbf{w}}\frac{\partial \mathbf{w}^*}{\partial \boldsymbol{\lambda}}\right)\Bigg|_{\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda})} = \tag{3}$$

$$\underbrace{\frac{\partial \mathcal{L}_V(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda}))}{\partial \boldsymbol{\lambda}}}_{\text{hyperparam direct grad.}} + \overbrace{\underbrace{\frac{\partial \mathcal{L}_V(\boldsymbol{\lambda}, \mathbf{w}^*(\boldsymbol{\lambda}))}{\partial \mathbf{w}^*(\boldsymbol{\lambda})}}_{\text{parameter direct grad.}} \times \underbrace{\frac{\partial \mathbf{w}^*(\boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}}}_{\text{best-response Jacobian}}}^{\text{hyperparam indirect grad.}}$$

Fig. 21. Hyper-gradient development [27]

The difficulty is to compute the best-response Jacobian or in other words to understand how the optimal weights change with respect to the hyper-parameters. To solve this problem, we can decompose it thanks to Cauchy's Implicit Function Theorem as in Figure 22, under some conditions which will not be detailed. As inverting the training Hessian is intractable

$$\frac{\partial \mathbf{w}^*}{\partial \boldsymbol{\lambda}}\Big|_{\boldsymbol{\lambda}'} = -\Big[\underbrace{\frac{\partial^2 \mathcal{L}_T}{\partial \mathbf{w} \partial \mathbf{w}}}_{\text{training Hessian}}\Big]^{-1} \times \underbrace{\frac{\partial^2 \mathcal{L}_T}{\partial \mathbf{w} \partial \boldsymbol{\lambda}}}_{\text{training mixed partials}}\Big|_{\boldsymbol{\lambda}', \mathbf{w}^*(\boldsymbol{\lambda}')}$$

Fig. 22. Best response Jacobian development [27]

in modern neural networks due to their size, the objective of the various gradient-based hyper-parameters optimizations is to approximate it. In the referenced paper, the Neumann series is used as an efficient approximation in memory usage as the authors decided to compute it. Their approach almost perfectly over-fits the validation data for a Recurrent Neural Network (RNN) with two layers of 650 units (13 280 400 weights). Consequently, it shows a satisfying performance at optimizing hyper-parameters with large networks.

### B. Self-tuning networks

An alternative approach was developed by the same team, months before, as presented in [28], that is based on a NN to estimate the best-response function. From the same objective of bi-level optimization, they consider this time $\lambda$ as a random variable and its distribution probability $p(\lambda)$ fixed. From this point, if we consider $w^* \approx \hat{w}_\phi$ as a good enough approximation, a gradient descent with respect to $\phi$ minimizes:

$$\mathbb{E}_{\lambda \sim p(\lambda)}[\mathcal{L}_T(\lambda, \hat{w}_\phi(\lambda))] \tag{34}$$

Then, if our hyper-network is sufficiently flexible regarding the support of $p(\lambda)$, we can consider the optimization of the hyper-parameters as a single-level problem :

$$\min_{\lambda \in \mathbb{R}^n} \mathcal{L}_V(\lambda, \hat{w}_\phi(\lambda)) \tag{35}$$

However, of course, this is not the case in practice. For this reason, the idea is to approximate $w^*$ in a neighborhood around the current value of $\lambda$. Adding some noise, for example Gaussian, with distribution $p(\epsilon|\sigma)$ in which $\sigma \in \mathbb{R}_+^n$ is a fixed scale parameter, $\phi$ can be found by minimizing :

$$\mathbb{E}_{\epsilon \sim p(\epsilon|\sigma)}[\mathcal{L}_T(\lambda + \epsilon, \hat{w}_\phi(\lambda + \epsilon))] \tag{36}$$

Even if it is unclear, if this approach can work on large scale problems, the ideas behind it are interesting and can be used as inspiration. A side effect of the local optimization is the choice of the variance of the noise will directly impact the capacity of generalization. Therefore, the correlation between the true best-response and its approximation around $\lambda$ will be degraded. For example, if the neighborhood is too small, the approximation will only match the true value for $\lambda$. On the contrary, if it is too wide, it will not be flexible enough to be a good approximation as we can see in Figure 23.

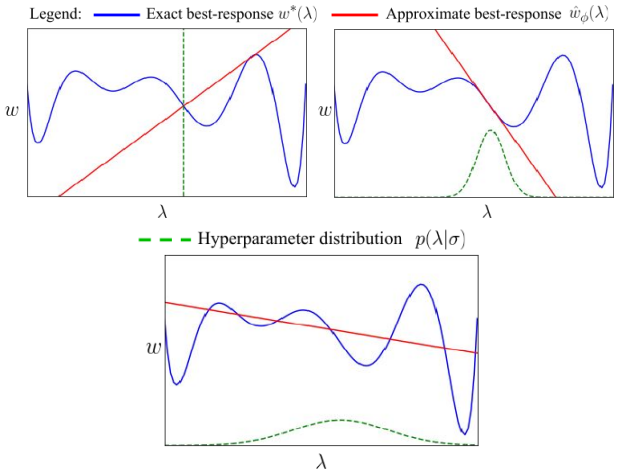


Fig. 23. Effect of the sampled neighborhood [28]

To answer the problem, an entropy term was added to the objective. Weighted by $\tau \in \mathbb{R}_+$, it is used to force the network to enlarge $\epsilon$. As expressed in equation 37, it will try

to minimize the loss-function while having a noise probability sufficiently large to avoid a heavy entropy penalty.

$$\mathbb{E}_{\epsilon\sim p(\epsilon|\sigma)}[\mathcal{L}_V(\lambda+\epsilon,\hat{w}_\phi(\lambda+\epsilon))]-\tau\mathbb{H}[p(\epsilon|\sigma)] \tag{37}$$

Finally, using every thing we saw to this point, the hyper-network can be trained as follows in Figure 24.

As for the first approach, on the experiences (language

**Initialize:** Best-response approximation parameters $\boldsymbol{\phi}$, hyperparameters $\boldsymbol{\lambda}$, learning rates $\{\alpha_i\}_{i=1}^3$
**while** not converged **do**
  **for** $t=1,\dots,T_{train}$ **do**
    $\boldsymbol{\epsilon}\sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$
    $\boldsymbol{\phi}\leftarrow\boldsymbol{\phi}-\alpha_1\frac{\partial}{\partial\boldsymbol{\phi}}f(\boldsymbol{\lambda}+\boldsymbol{\epsilon},\hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda}+\boldsymbol{\epsilon}))$
  **for** $t=1,\dots,T_{valid}$ **do**
    $\boldsymbol{\epsilon}\sim p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})$
    $\boldsymbol{\lambda}\leftarrow\boldsymbol{\lambda}-\alpha_2\frac{\partial}{\partial\boldsymbol{\lambda}}\left(F(\boldsymbol{\lambda}+\boldsymbol{\epsilon},\hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda}+\boldsymbol{\epsilon}))-\tau\mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})]\right)$
    $\boldsymbol{\sigma}\leftarrow\boldsymbol{\sigma}-\alpha_3\frac{\partial}{\partial\boldsymbol{\sigma}}\left(F(\boldsymbol{\lambda}+\boldsymbol{\epsilon},\hat{\mathbf{w}}_{\boldsymbol{\phi}}(\boldsymbol{\lambda}+\boldsymbol{\epsilon}))-\tau\mathbb{H}[p(\boldsymbol{\epsilon}|\boldsymbol{\sigma})]\right)$

Fig. 24. STN training algorithm where $F$ refers to $\mathcal{L}_V$ and $f$ refers to $\mathcal{L}_T$ [28]

modeling and image classification), this one was able to over-fit the validation set and outperform grid search, random search and Bayesian optimization.

## VI. Reusable holdout

Agents are able to fit to the training set, hyper-parameter optimizers are able to lead the agent to fit to the validation set but did we build the desired agent ? Probably not, and this is not a reinforcement learning related problem but a more general machine learning problem. As explained in the introduction, we want our algorithm the have the best reaction possible when facing a problem. However, we cannot train it on an infinite set of problems of the same type in order for it to be able to solve one. Thus, we train it on a finite, often small, set of examples and try to make it generalize so it is prepared to be evaluated on fresh data.

This is where our agent will fail as it is trained to overfit both the training and validation dataset and will yield a poor generalization capacity when faced to fresh data. In this section, backed by the theoretical aspects detailed in [29], we will present a way to train an agent under strong confidence intervals of not overfitting. Consider a function $\phi:\mathcal{X}\to[0,1]$ on distribution $\mathcal{P}$. Its expectation is denoted as $\mathcal{P}[\phi]=\mathbb{E}_{x\sim\mathcal{P}}\phi(x)$ and a natural estimator, given a dataset of $n$ random and independent samples $S=(x_1,...,x_n)$, is the empirical average $\mathcal{E}_S[\phi]=\frac{1}{n}\sum_{i=1}^n\phi(x_i)$. From the Hoeffding bound, it is possible to bound the probability for the estimator to have an error greater than $\tau$ as :

$$\mathbb{P}(|\mathcal{E}_S[\phi]-\mathcal{P}[\phi]|>\tau)\leq 2e^{-2\tau^2 n} \tag{38}$$

### A. Max information

First, we want to determine exactly the amount of information that was used by the agent, in other words, to quantify the amount of information learned from the data : that is maximum information. If we consider jointly distributed random variables $(X,Y)$, the max information bound the logarithm of factor by which uncertainty about $X$ is reduced given $Y$ :

$$I_\infty(X,Y)=k \text{ such that } \mathbb{P}[X=x|Y=y]\leq 2^k\mathbb{P}[X=x] \tag{39}$$

We can now consider our problem, overfitting to the dataset, denoted $S\in R(\Phi)$ and bound the probability for this to happen as :

$$I_\infty(S,\Phi)=k \text{ such that } \mathbb{P}[S\in R(\Phi)]\leq 2^k\max_\phi\mathbb{P}[S\in R(\phi)] \tag{40}$$

More precisely, if we consider that $\phi$ overfits the data if the empirical average (on the new data) has an error greater than $\tau$, we can define the dataset on which it happens as :

$$R_\tau(\phi)=\{S\in\mathcal{X}^n:\mathcal{E}_S[\phi]-\mathcal{P}[\phi]>\tau\} \tag{41}$$

Therefore, if :

$$I_\infty(S,\Phi)\leq\log_2 e^{\tau^2 n} \text{ then } \mathbb{P}[S\in R_\tau(\Phi)]\leq e^{-\tau^2 n} \tag{42}$$

### B. Differential Privacy

Now that we have been able to bound the probability of overfiting to $S$ under some conditions, we need a way to communicate with the agent without giving it too much information : a differentially private algorithm. Two datasets $x,y$ are considered adjacent if they differ by a single element. A randomized algorithm $\mathcal{M}:\mathcal{X}^n\to S$ is $(\epsilon,\delta)$-differentially private if for all adjacent datasets $x,y\in\mathcal{X}^n$ :

$$\mathbb{P}[\mathcal{M}(x)\in S]\leq e^\epsilon\mathbb{P}[\mathcal{M}(y)\in S]+\delta \tag{43}$$

Although we will not detail it in this paper, we could guarantee that the probability of any possible function $\phi$ from $\Phi$ is less or equal to $6e^{-\tau^2 n}$, has a budget of $\tau^2$. This means the agent will be able to interact with the data for a maximum of $\tau^2$ iterations through an $(\epsilon,0)$-differential private algorithm for $\epsilon\leq\tau$. If we want to guarantee an overfitting probability less or equal to $\beta=e^{-2r^2 n}$ through $(\tau/4,(\beta/8)^{4/\tau})$-differential privacy, we need to set a budget $\frac{\tau^5 n^2}{215\ln(8/\beta)}$.

The first budget will guarantee a low probability of overfitting, but with a constant number of access to the data. On the contrary, the second has a similar probability but the number of interaction is proportional to the squared number of samples in the dataset. Finally, if we respect all of these constraints, it is possible to build a simple algorithm to interact with our agent as presented on Figure 25.

In this algorithm, Lap is a Laplacian function and the threshold is the difference limit for which we consider the

**Input:** Training set $S_t$, holdout set $S_h$, threshold $T$, tolerance $\tau$, budget $B$
**Query step:** Set $\hat{T} \leftarrow T + \gamma$ for $\gamma \sim \mathrm{Lap}(4 \cdot \tau)$. Given a function $\phi \colon \mathcal{X} \to [-1, 1]$, do:

1. If $B < 1$ output "$\perp$"
2. Else sample $\xi \sim \mathrm{Lap}(2 \cdot \tau)$, $\gamma \sim \mathrm{Lap}(4 \cdot \tau)$, and $\eta \sim \mathrm{Lap}(8 \cdot \tau)$
   (a) If $|\mathcal{E}_{S_h}[\phi] - \mathcal{E}_{S_t}[\phi]| > \hat{T} + \eta$, output $\mathcal{E}_{S_h}[\phi] + \xi$ and set $B \leftarrow B - 1$ and $\hat{T} \leftarrow T + \gamma$.
   (b) Otherwise, output $\mathcal{E}_{S_t}[\phi]$.

Fig. 25. Thresholdout algorithm [29]

agent to be overfitting. What the reader can understand from it is that the agent is given an unlimited access to the training dataset. If you have interacted with the validation set more than the budget allows it however, the algorithm will not let you have any more information about it. On the other hand, if you overfit the validation set with respect to the threshold you have set, it outputs the performance on the validation set with additional noise.

This way, the agent is never allowed to learn exactly what samples are in the validation set. It never knows if it is overfitting exactly to the threshold $T$ and does not even know its exact performance. Even though it is unclear, for the moment, if this approach is improving the generalization capacity on complex data, it held a strong correlation in their experiment of the performance on the validation set and with fresh data. Moreover, the demonstration in the paper is a strong mathematical guarantee for the performance of any differential algorithm and therefore can be adapted to suit any kind of overfitting problem.

## VII. Libraries

Python is not more adapted to implement reinforcement learning algorithms than other programming languages, but its easy and concise syntax make it an interesting language to go with. Also, there is already a good amount of tools developed to help researchers implement an experimentation. In this section, we present four different libraries that can be used in Python to implement and test a reinforcement learning project. These are pfrl, rlpyt, autonomous-learning-library and Deep-Reinforcement-Learning-Algorithms-with-PyTorch. They all have an available repository on github. At the end of this section, a table summarizes which libraries implement or not the algorithms presented in this paper.

Before starting, we need to introduce OpenAI Gym. Reinforcement learning studies how an agent can learn to achieve goals in an environment. To do so, we need to implement both the agent and the environment. The main goal of the following libraries is to implement the agents, not the environments. In fact, they use the OpenAI Gym library to offer some famous environments but we can apply an agent provided by one of these library on our own environment if it supports the subset of OpenAI Gym's interface. Gym was developed to try to fix the lack of standardization of environments used in publications. Its strength is the Atari 2600 games implemented. There are also environments to simulate movements for robotic hands, arms or 3D environments for one-legged, two-legged and four-legged robots. Moreover, there are famous environments such as CartPole, Acrobot, MountainCar and dozens more created by third-party.

### A. PFRL

Pfrl was developed by the company Preferred Networks (PFN), which previously developed Chainer, a framework for neural networks, and ChainerRL, a reinforcement learning library. It implements various state-of-the-art deep reinforcement algorithms using PyTorch, an open source machine learning framework. For the environments, the ones that support the subset of OpenAI Gym's interface can be used. Pfrl provides agents, each of which implements a deep reinforcement learning algorithm. The full list of pfrl agents currently includes: A2C, A3C, ACER, Advantage Learning (AL), CategoricalDQN, CategoricalDoubleDQN, DDPG, DQN, DoubleDQN, Persistent Advantage Learning (PAL), DoublePAL, PPO, REINFORCE, SAC, TRPO, TD3, IQN, and Dynamic Policy Programming (DPP). Pfrl furnish a good quickstart guide and a complete documentation.

### B. RLPYT

Rlpyt was developed by the Berkeley Artificial Intelligence Research (BAIR) Lab. A multitude of new algorithms have flourished with the enthusiasm of using deep reinforcement learning in domains like games or simulated robotic control. Observing that most of these algorithms can be categorized into three groups (deep Q-learning, policy gradients and Q-value policy gradients) that have been developed along separate lines of research, BAIR created the rlpyt library to gather the implementations in one repository to avoid that RL researchers must invest time reimplementing algorithms. Rlpyt is compatible with the OpenAI Gym interface and provides access to many existing learning environments. It contains modular implementations for many common deep RL algorithms in Python using Pytorch : A2C, PPO, DQN (+ Double, Dueling, Categorical, Recurrent), DDPG, TD3, SAC. Rlpyt provides a complete documentation and some examples.

### C. Autonomous-learning-library

This library was built in the Autonomous Learning Laboratory (ALL) at the University of Massachusetts, Amherst. It was written and is currently maintained by Chris Nota. Observing that building an effective intelligent agent requires setting a mass of hyperparameters to shape the environment, establish the rewards and so on, a group of researchers has developed the autonomous-learning-library to simplify the process. This PyTorch-based library is designed for the quick implementation of novel reinforcement learning agents. One of the stated core philosophies of the initiative is that the reinforcement learning should be agent-based, meaning the models simply accept a state and a reward and then return an action. Once again, environments are linked to OpenAI Gym library. It contains A2C, Categorical DQN(C51), DDPG, DQN+extensions, PPO, Rainbow, SAC. The documentation is complete, there is not a lot of examples but a detailed user guide.

### D. Deep-Reinforcement-Learning-Algorithms-with-PyTorch

This library was developed by Petros Christodoulou, a machine learning scientist at Amazon. There are some environments implemented but we can implement ours with OpenAI Gym. It implements many agents : DQN, DDQN, Dueling DDQN, REINFORCE, DDPG, TD3, SAC, A3C, A2C, PPO and some others but there is no documentation, only few tests and examples.

### E. Summary of libraries implementations

This table summarizes whether the libraries introduced in this section implement the algorithms previously studied in this paper.

| | pfrl | rlpyt | a-l-l | D-R-L-A-w-P |
|---|---|---|---|---|
| A3C | ✓ | | | ✓ |
| A2C | ✓ | ✓ | ✓ | ✓ |
| ACER | ✓ | | | |
| DDPG | ✓ | ✓ | ✓ | ✓ |
| SAC | ✓ | ✓ | ✓ | ✓ |
| DQN | ✓ | ✓ | ✓ | ✓ |
| Dueling DQN | | ✓ | ✓ | ✓ |
| Categorical DQN | ✓ | ✓ | ✓ | |

## VIII. Conclusion

In this paper, we presented the different topics necessary to understand how to perform efficient reinforcement learning in various ways. We set some basic knowledge and then discussed about DQN and Actor-Critic algorithms using countless principles to perform on the training set: experience replay, entropy, stochastic noise, soft update, etc. After we explained these principles, we have been through hyper-parameter optimization as gradient-based or STN. We discussed an approach to counter-balance the side effect of these optimizations, that is overfitting, and presented four useful libraries to implement all the concepts developed in this article. Thanks to all of these aspects, we presented the main topics necessary to build a hyper-parameters optimizer reinforcement learning agent that will be presented in a next article.

## References


[1] "AlphaFold: a solution to a 50-year-old grand challenge in biology."

[2] S. J. Russell and P. Norvig, *Artificial intelligence : a modern approach*. Boston: Pearson, 2016.

[3] OpenAI, "Gym: A toolkit for developing and comparing reinforcement learning algorithms."

[4] "Part 2: Kinds of RL Algorithms — Spinning Up documentation."

[5] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[7] M. Sewak, "Deep Q Network (DQN), Double DQN, and Dueling DQN," in *Deep Reinforcement Learning: Frontiers of Artificial Intelligence* (M. Sewak, ed.), pp. 95–108, Singapore: Springer, 2019.

[8] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," *arXiv:1511.06581 [cs]*, Apr. 2016. arXiv: 1511.06581.

[9] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning," *arXiv:1710.02298 [cs]*, Oct. 2017. arXiv: 1710.02298.

[10] M. G. Bellemare, W. Dabney, and R. Munos, "A Distributional Perspective on Reinforcement Learning," *arXiv:1707.06887 [cs, stat]*, July 2017. arXiv: 1707.06887.

[11] "An intro to Advantage Actor Critic methods: let's play Sonic the Hedgehog!."

[12] "Asynchronous Advantage Actor Critic (A3C) algorithm," June 2019. Section: Machine Learning.

[13] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *arXiv:1602.01783 [cs]*, June 2016. arXiv: 1602.01783.

[14] A. Sarkar, "A Brandom-ian view of Reinforcement Learning towards strong-AI," *arXiv:1803.02912 [cs]*, Mar. 2018. arXiv: 1803.02912 version: 1.

[15] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation," *arXiv:1708.05144 [cs]*, Aug. 2017. arXiv: 1708.05144.

[16] "Policy Gradient Algorithms," Apr. 2018.

[17] A. Gruslys, W. Dabney, M. G. Azar, B. Piot, M. Bellemare, and R. Munos, "The Reactor: A fast and sample-efficient Actor-Critic agent for Reinforcement Learning," *arXiv:1704.04651 [cs]*, June 2018. arXiv: 1704.04651.

[18] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample Efficient Actor-Critic with Experience Replay," *arXiv:1611.01224 [cs]*, July 2017. arXiv: 1611.01224.

[19] "Deep Reinforcement Learning."

[20] W. Shang, D. van der Wal, H. van Hoof, and M. Welling, "Stochastic Activation Actor Critic Methods," in *Machine Learning and Knowledge Discovery in Databases* (U. Brefeld, E. Fromont, A. Hotho, A. Knobbe, M. Maathuis, and C. Robardet, eds.), vol. 11908, pp. 103–117, Cham: Springer International Publishing, 2020. Series Title: Lecture Notes in Computer Science.

[21] W. Shang, "Learn to Adapt Uncertainty with Stochastic Activation Actor-Critic Methods," p. 17.

[22] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv:1509.02971 [cs, stat]*, July 2019. arXiv: 1509.02971.

[23] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *arXiv:1502.03167 [cs]*, Mar. 2015. arXiv: 1502.03167.

[24] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft Actor-Critic Algorithms and Applications," *arXiv:1812.05905 [cs, stat]*, Jan. 2019. arXiv: 1812.05905.

[25] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," *arXiv:1802.09477 [cs, stat]*, Oct. 2018. arXiv: 1802.09477 version: 3.

[26] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv:1707.06347 [cs]*, Aug. 2017. arXiv: 1707.06347.

[27] J. Lorraine, P. Vicol, and D. Duvenaud, "Optimizing Millions of Hyperparameters by Implicit Differentiation," *arXiv:1911.02590 [cs, stat]*, Nov. 2019. arXiv: 1911.02590.

[28] M. MacKay, P. Vicol, J. Lorraine, D. Duvenaud, and R. Grosse, "Self-Tuning Networks: Bilevel Optimization of Hyperparameters using Structured Best-Response Functions," *arXiv:1903.03088 [cs, stat]*, Mar. 2019. arXiv: 1903.03088.

[29] C. Dwork, V. Feldman, M. Hardt, T. Pitassi, O. Reingold, and A. Roth, "The reusable holdout: Preserving validity in adaptive data analysis," *Science*, vol. 349, pp. 636–638, Aug. 2015. Publisher: American Association for the Advancement of Science Section: Report.