PoS PROCEEDINGS OF SCIENCE

Reinforce-lib: A Reinforcement Learning Library for Scientific Research

Luca Anzalone[a,*] **and Daniele Bonacorsi**[a]

[a] *Department of Physics and Astronomy (DIFA), University of Bologna, 40127, Bologna, Italy*

E-mail: luca.anzalone2@unibo.it

Reinforcement Learning (RL) has already achieved several breakthroughs on complex, high-dimensional, and even multi-agent tasks, gaining increasingly interest from not only the research community. Although very powerful in principle, its applicability is still limited to solving games and control problems, leaving plenty opportunities to apply and develop RL algorithms for (but not limited to) scientific domains like physics, and biology. Apart from the domain of interest, the applicability of RL is also limited by numerous difficulties encountered while training agents, like training instabilities and sensitivity to hyperparameters. For such reasons, we propose a *modern*, *modular*, *simple* and *understandable* Python RL library called `reinforce-lib`. Our main aim is to enable newcomers, practitioners, and researchers to easily employ RL to solve new scientific problems. Our library is available at https://github.com/Luca96/reinforce-lib.

The paper is organized as follows: in section 1 we introduce and motivate our contribution, in sections 2 and 3 we provide a short introduction to the reinforcement learning paradigm, as well as describing three popular families of algorithms, then, in the following section 4 we present our library, finally section 5 concludes our discussion.

International Symposium on Grids & Clouds 2022 (ISGC 2022)
21 - 25 March, 2022
Online, Academia Sinica Computing Centre (ASGC), Taipei, Taiwan

*Speaker

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

PoS(ISGC2022)018

1. Introduction

Since 2013, thanks to the breakthrough achieved by the *DQN* agent [1] on the *Atari learning environment* [2] - a benchmark that was thought to be feasible only for humans - Reinforcement Learning [3] (RL) and, especially, its combination with Deep Learning [4] (DL), called *Deep Reinforcement Learning* (DRL), have both attracted attention since then, likewise AlexNet [5] (thanks to the astounding improvement achieved on the ILSVRC [6], compared to the best classical computer vision algorithm at that time) had started the deep learning era.

After few years, we have now powerful actor-critic distributed agents [7–9] that are able to solve complex control [10] and robotic tasks [11], and surprising even able to handle exponentially large search spaces like the ones found in the board games of Chess and Go [12–14], as well as high-dimensional continuous state-spaces of multi-agent environments [15–17]. All these successes have common roots: powerful neural-networks function approximators borrowed from DL, and distributed training.

Although DRL seems to be incredibly powerful, in practice training successful agents is notoriously *difficult*, *time consuming*, *resource intensive*, *costly*, and *error-prone* [18, 19]: mainly due to very sensitive hyperparameters, beyond problem complexity itself. Such difficulties may arise from a still very limited understanding of the underlying mechanisms that power both RL and DRL, effectively preventing us to derive simpler (i.e. with way less moving parts, and hyperparameters) and thus more effective, sample-efficient, RL algorithms.

As happened in DL, having widely-used tools like Keras [20] that simplify the building and training of neural networks is essential for speeding-up and improving research in that particular and related field(s). With such in mind, our aim is to provide a tool to ease the workflow of *defining*, *building*, *training*, *evaluating* and *debugging* DRL agents. Our Python library `reinforce-lib` provides simple code interfaces to a variety of implement agents and environments. We adopt a *modular design* that allows users to replace components like the agent's *networks*, its *policy*, and even *memory buffer* with other components made available by the library itself, or new modules designed by the users themselves: this should enable the practitioner or researcher to easily prototype either novel research ideas, improvements to existing algorithms and components, or to adapt the agent to novel, previously unsolved, research problems.

The reinforce-lib library other than being designed to be easy to use and extend, it will be also complete in the long-term, encompassing the thee main paradigms found in reinforcement learning, namely: *model-free*, *model-based* [21], and *inverse* RL [22]. This will allow users to solve a broader variety of problems according to their prior problem setting. For example, if the problem we want to solve naturally allow us to define a reward function, we will choose a model-free agent to solve it. If, instead, is easy for the researcher to provide a model of the environment (task or problem), model-based agents will do the job. Lastly, if we have many data coming from optimal sources (like domain experts, precise but expensive numerical simulations, exact algorithms, etc) we can leverage inverse RL algorithms to first learn a reward function, and then use the learned reward to power the learning of model-free or hybrid agents. Moreover, the design principles of reinforce-lib, namely *usability*, *extensibilty*, and *completeness*, should make the library distinguish itself from currently available RL libraries which are often based on old or legacy code, resulting in unfriendly code interfaces that are difficult to use and/or to extend.

PoS(ISGC2022)018

2. Reinforcement Learning

Reinforcement Learning (RL) [3] is a learning paradigm to tackle decision-making problems, that provides a formalism for modeling behavior, in which a software or physical *agent* learns how to take optimal *actions* within an *environment* (i.e. a real or simulated world) by trial and error, guided only by positives or negatives scalar *reward* signals (also called *reinforcements*).

Compared to (un)supervised learning, reinforcement learning is different in many aspects:

- The supervision is *weak*: optimal actions (i.e. the supervisory signal) are not known, and so the agent have to learn them by trial and error, guided only by a reward signal. This also implies that the agent has to try diverse enough actions, while learning their goodness.
- The feedback can be *delayed*: rewards can be sparse (i.e. zero or one), noisy, mostly meaningless, and even only provided late in time (e.g. at the end of a match).
- The data distribution is *sequential* not i.i.d.: the agent learns from data coming by interacting with the environment, which is sequential. This implies the data to be correlated along time, and that, also, the actions derived by the agent will affect the future distribution of the data.
- The **reward hypothesis**: assumes that the task the agent has to solve can be described by the maximization of rewards through the definition of some reward function. If we are unable to provide a reward function, basically we cannot apply reinforcement learning to a given problem.

2.1 Markov Decision Processes

Formally, an environment is a *Markov Decision Process* (MDP) usually represented by a 5-element tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma \rangle$, in which:

- $\mathcal{S}$ is the *state space*, defining what the agent could experience while interacting with the environment. States (denoted by s) can be discrete or continuous, scalar or multi-dimensional. In general, even if enumerable, all the possible states are not known in advance by the agent.
- $\mathcal{A}$ is the *action space*, defining which actions $a \in \mathcal{A}$ the agent is allowed to draw. Actions can be discrete or continuous.
- $\mathcal{P}(s' \mid s, a)$ is the *transition model* (also called the *environment dynamics*) that specifies the probability of a state transition: i.e. a change of state given the application of an action by the environment. If the transition model is not known nor provided or used, we are in the model-free setting of RL in which the agent learns by *association*. Instead, if $\mathcal{P}$ is either given or learned, the *model-based* agent can use the environment's model to *plan* future actions without actually interacting with it.
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the *reward function*. At each timestep t of interaction the agent will receive a scalar *immediate* reward, $r_t = r(s_t, a_t)$, yield by the environment as a feedback that evaluates the "immediate" goodness of applying the predicted action a_t on the current state s_t of the environment. Sometimes specifying a reward function is either difficult, imprecise

PoS(ISGC2022)018

or not possible. In such cases is possible to resort to *inverse reinforcement learning* [22] to learn a reward function from *expert* data (i.e. data thought to be optimal or sufficiently good for the domain of interest).

- $\gamma \in [0, 1]$ is the *discount factor*, that "discounts" the value of rewards received late in time. Discounting is also mandatory for continuing, non-terminating environments, in order to prevent the sum of rewards to diverge. Moreover, rewards obtained after $1/(1-\gamma)$ timesteps tend to be approximately zero; so, it's like discarding them.

To accomplish the task the agent is requested to solve, it has to interact multiple (many) times with the MDP. In jargon, one-step of interaction is called a *transition* (s, a, s', r) in which: s is the current state, a the predicted action, s' the next state, and r the immediate reward; while a collection of single interactions (i.e. transitions) starting from the beginning ($t = 0$) until termination ($t = T - 1$, where T is the maximum number of timesteps that can be potentially unbounded) is called a *trajectory*, denoted as follows: $\tau = \{(s_t, a_t, s_{t+1}, r_t)\}_{t=0}^{T-1}$. Also a trajectory represents a learning *episode*, part of the interaction or learning loop between the agent and the environment.

The RL *learning loop* is made of multiple episodes each with maximum length T: likewise in ML or DL an epoch is made of multiple (update) steps. At the start of each episode (when the timestep is zero) the environment is reset, providing the agent an *initial state* $s_0 \in \mathcal{S}$ sampled from the *initial state distribution* $\rho(s)$, which basically tells which states of the space $\mathcal{S}$ can be initial states, and also with which probability. From such initial state, the agent then predicts an action a_0 which is then applied by the environment resulting in a new state, s_1, according to its transition model $s_1 \sim \mathcal{P}(s_1 \mid s_0, a_0)$. The change of state is then evaluated by the environment, resulting in the reward r_1 the agent will use as feedback, to learn if the choice of the action a_0 was good for the state s_0. In practice, the environment also provides a Boolean *done* flag, d, that denotes whether or not the next state s_1 is a *terminal state* or not. If it is, then the interaction terminates here and the environment must be reset to start a new episode of interaction. All of this is just a single interaction, resulting in transition (s_0, a_0, s_1, r_1, d). If d is false, then the interaction continues and so we increment t by one. If the agent never encounters a terminal state, the episode is still terminated (although artificially) when the maximum timestep is exceeded, i.e. when $t > T - 1$.

An important property of MDPs is that states are *Markov*, implying that a state transition at time t, $\mathcal{P}(s_{t+1} \mid s_t, a_t)$, does only depend on the previous *state-action pair* (s_t, a_t) and not on all the previous pairs $a_{t-1}, s_{t-1}, a_{t-2}, s_{t-2}, \dots, s_0$, also called the *history* H_t, from which it is *independent*. This is what enables agents to derive actions from only the current state, and not all the previous ones. Indeed, the Markov property only holds if the states are *fully-observable*, if not the states are now called *observations* to highlight the fact that they are only *partially observable*. Partial observability means that some aspects of the state are not provided to the agent, and so there exist latent variables that explain the underlying fully-observable state: e.g. image observations only represent an observable portion of the full, internal state of the environment at a given time t. In such cases, the agent has to consider the full or truncated history in order to learn effectively.

2.2 Policy, Value Functions, and the Performance Objective

The way the agent derives an action given a state is by means of a *policy*, and the way it learns the "value" (or goodness) of actions and states is through the aid of *value functions*. These components

PoS(ISGC2022)018

are fundamental to enable the agent to act and learn.

The **policy** can be *stochastic* or *deterministic*. A stochastic policy, $\pi(a \mid s)$, is a probability distribution over actions given states; such an object must support the *sampling* of actions, as well as computing their (log-)probability. Common choices for the underlying distribution are:

- *Bernoulli*: when the action space $\mathcal{A}$ is discrete, and made only of two *binary* actions (either zero or one).
- *Categorical*: for discrete action spaces, with enumerable and finite actions.
- *Gaussian*: for continuous and *unbounded* action spaces. If there are more then one action, a *diagonal* multivariate Gaussian can be also used.
- *Beta* [23]: for continuous and *bounded* action spaces. The Beta distribution has support in $[0, 1]$, so sampled actions must be scaled to a range suitable for the environment. Alternatives to the Beta distribution are the *truncated* Gaussian, or the *squashed* Gaussian (bounded in $[-1, +1]$ by means of the tanh operation).

Policies can also be deterministic, usually denoted by $a = \mu(s)$, in which μ is a deterministic function of states. Since sampling from a deterministic policy is not possible, to ensure enough diversity (and also to prevent premature local minimum), *noise* is commonly added while learning. For example, the action $a = \mu(s) + \epsilon$ is obtained by adding noise, $\epsilon \sim \mathcal{N}(0, \sigma)$, sampled from a Normal distribution. In such case, the amount of injected noise can be easily controlled by varying the σ parameter, corresponding to the standard deviation of the distribution.

The policy provides a way of acting for the agent, whose role is to learn the distribution parameters such that better actions are sampled (or predicted) more and more frequently. The way policies are evaluated is by means of **value functions**, which turn out to be also useful to later improve them. There are *two* kind of value functions: the *state-value function* $V^\pi(s)$, and the *action-value function* $Q^\pi(s)$ (also called the *Q-function*). The former is responsible to compute the value of states, while the latter provides the value of an action considering the current state. Here the term "value" indicates how much *expected* (discounted) reward is gained from that state and onward. Therefore value functions are not greedy (just accumulating discounted rewards), but they try to evaluate the future in the long and "expected" term, thus effectively averaging over all possible situations. A value function, basically, computes the *expected return* according to the current way of acting, i.e. given the current policy π. The *return*, G_t, is just a collection of discounted rewards from a certain timestep, t, and onward until termination (assuming a horizon of length T):

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k} \tag{1}$$

Having defined the return, we can now define the two value functions in terms of it:

$$\begin{aligned}
V^\pi(s) &= \mathbb{E}_\pi\left[G_t \mid s_t = s\right] && (2)\\
&= \mathbb{E}_\pi\left[r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s\right] && (3)\\
Q^\pi(s, a) &= \mathbb{E}_\pi\left[G_t \mid s_t = s, a_t = a\right] && (4)\\
&= \mathbb{E}_\pi\left[r_t + \gamma \mathbb{E}_{a' \sim \pi}[Q^\pi(s_{t+1}, a_{t+1}) \mid a_{t+1} = a'] \mid s_t = s\right] && (5)
\end{aligned}$$

PoS(ISGC2022)018

As we can notice, both value functions have a *recursive* definition in terms of the immediate reward r_t, and themselves, but evaluated in the future: i.e. on the next state s_{t+1} (according to the environment dynamics), or on the next state-action pair (s_{t+1}, a_{t+1}) (according to both environment dynamics and policy). Also, in the definition of the Q-function (Eq. 5), we can observe how the inner expectation (over possible actions) actually corresponds to the state-value function of the next state, i.e.: $V^\pi(s') = \mathbb{E}_{a' \sim \pi}\left[Q^\pi(s', a')\right]$. This is an useful relation that can avoid the need to learn two separate value functions, since we can recover V^π from just Q^π.

At this point, we can say that the actual aim of the agent is to find a policy $\pi^\star$ such that the sum of (discounted) rewards is maximal, averaged (expected) over all possible trajectories τ. This is what is called the **performance objective** $J(\pi)$, defined as follows:

$$J(\pi) = \mathbb{E}_{(s_t, a_t) \sim p(\cdot \mid \pi)}\left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t)\right] \tag{6}$$

$$\pi^\star = \arg\max_\pi J(\pi) \tag{7}$$

Solving Eq. 7 entails a search over the space of possible policies, which is not trivial. Moreover, we can notice that the agent wants to maximize the return, but expected over all possible trajectories τ (or state-action pairs) induced by $p(\tau \mid \pi)$, which is the *induced trajectory probability*:

$$p(\tau \mid \pi) = \rho(s_0) \prod_{t=0}^{T-1} \mathcal{P}(s_{t+1} \mid s_t, a_t) \pi(a_t \mid s_t), \tag{8}$$

describing which states will be visited, and which actions will be yield, both according the transition model of the environment and the actual policy π of the agent.

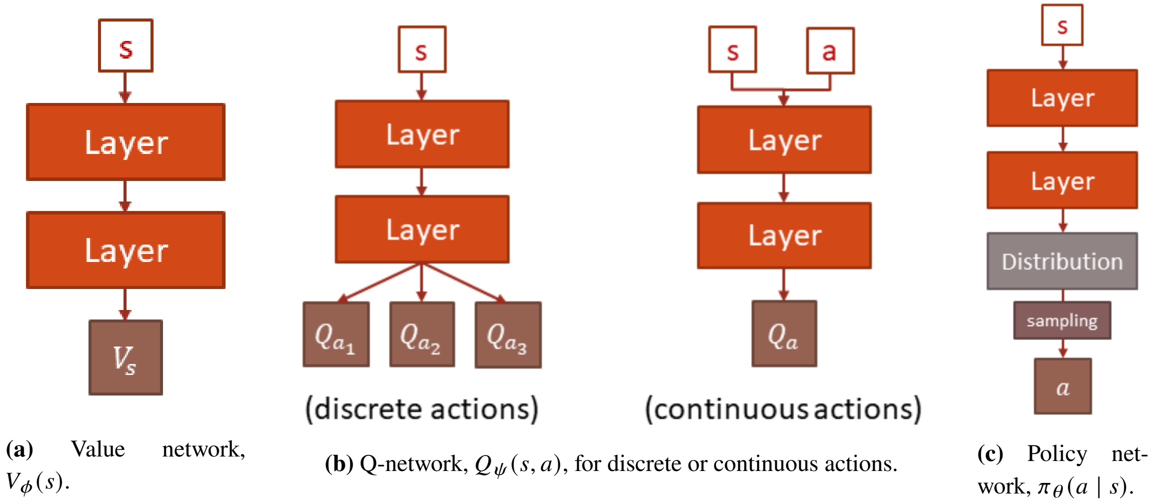
3. Deep Reinforcement Learning

One classical approach to solve the reinforcement learning problem (i.e. solving Eq. 7, given an MDP), described in the previous section, is by first representing both the policy and a value function as *tables*, and then iteratively learn the value function from which a better policy is devised. The process is then repeated until convergence. Having a function represented by a table introduces difficulties when states and/or actions are continuous, since we cannot use them to *index* the table, e.g. to retrieve $v_s = V_{\text{table}}[s]$. In addition, tables suffer from the *curse of dimensionality*, even when the state and/or action spaces are discrete. This means that the number of entries of the table will be exponentially large in the number of dimensions of the states and/or actions.

Those issues had limited the applicability of RL to larger and complex problems, like continuous control. But by representing the functions of interest with *deep neural networks* [4] (as shown in figure 1) instead of tables, we can easily overcome such problems: unfortunately without introducing new issues, like the lost of *convergence guarantees* ensured by tables, which may result in unstable and sensitive training.

Since deep RL deals with training of one or more neural networks (representing the agent), the performance objective (Eq. 7) is now about finding the optimal set of *weights*, $\theta^\star$, such that agent performance are maximal. So, the optimization problem is now: $\theta^\star = \arg\max_\theta J(\theta)$, involving a search in parameter space (instead of policy space). Such search is conducted by means of stochastic

PoS(ISGC2022)018



(a) Value network, $V_\phi(s)$.

(b) Q-network, $Q_\psi(s, a)$, for discrete or continuous actions.

(c) Policy network, $\pi_\theta(a \mid s)$.

Figure 1: Schematic representation of value (a), action-value (b), and policy (c) functions (described in section 2.2) implemented as neural networks. Each network has its own set of learnable weights. Moreover, the agent can use and learn one or more of them. In case of discrete actions, the action-value network (b: left) will output an estimate of the Q-value for each action in the space, otherwise (b: right) we just combine (e.g. concatenate) the action with the input state, to predict the value of continuous actions. Lastly, the policy network (c) has to include an instance of a suitable probability distribution, whose parameters are yield by the last hidden layer. In addition, to enable learning (i.e. avoid disconnected gradients) the probability distribution has to be correctly *reparametrized*: an operation that is ensured by using libraries like `tfp` [24].

gradient descent (SGD; usually Adam [25]) and back-propagation [26] but, only if a differentiable learning objective is provided. Unfortunately, gradients cannot be computed from the performance objective $J(\theta)$ directly, since there is no dependence between the rewards and the parameters θ, as the former are yield by the environment. Although, it is possible to setup a Monte-Carlo *score estimator* to estimate the gradient of $J(\theta)$ by sampling, it would be tremendously high-variance, and so unstable. In this perspective, deep reinforcement learning algorithms can be interpreted as being clever gradient estimators of non-differentiable objectives. Perhaps, the most famous is the REINFORCE [27] gradient estimator, that belongs to the family of policy gradient algorithms.

In practice, the performance objective cannot be even computed exactly as it involves an expectation over all possible trajectories, which would mean for the agent to first explore the *whole* space of possible states. Hence, an approximation of the performance objective is employed but only to evaluate the performance of the agent:

$$J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \gamma^t r(s_t^{(i)}, a_t^{(i)}) \tag{9}$$

Such equation is now feasible to compute, as it only requires to average the total reward gained on N trajectories (or episodes of interaction).

3.1 Deep Q-Learning

The DQN [1] agent (or algorithm) has made deep reinforcement to be both famous and effectively applicable to complex problems, such as solving computer games in the arcade learning environment

PoS(ISGC2022)018

(ALE) [2]. The agent features *only* a deep Q-network (like the ones depicted in figure 1b, left), from which it borrows its name, that receives a stack of four recent image frames: this the case of ALE, indeed, that has image observations; moreover, the frame stacking is an effective yet simple technique to deal with partial observability. There is no explicitly policy network, and, in fact, the policy is indirectly derived from the learned Q-network: basically, this means that an improvement of the action-value estimates results in an improved policy. Given a Q-network, Q_ϕ, the policy π is obtained as follows:

$$\pi(a \mid s) = \begin{cases} 1 & \text{if } a^\star = \arg\max_{a \in \mathcal{A}} Q_\phi(s, a) \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

This is the *greedy* policy, that always samples the action $a^\star$ with the highest action-value. We say that the greedy policy is a policy that always *exploits*, meaning that it always selects only the best actions, discarding sub-optimal ones. A policy that always exploits is not favorable while learning, since it can trap the agent into a local minimum, as enough *exploration* of actions is not contemplated. A way to pretend exploration is to "sometimes" pick an action that is not the greedy action (i.e. the action with best Q-value). Such a policy is called *epsilon-greedy*, since the ratio of exploration can be easily controlled by setting the ϵ hyperparameter ($\epsilon \in [0, 1]$):

$$\pi_\epsilon(a \mid s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^\star = \arg\max_{a \in \mathcal{A}} Q_\phi(s, a) \\ \epsilon/m & \text{otherwise} \end{cases} \tag{11}$$

The ϵ-greedy policy will select a random action with probability ϵ and, since actions are sampled uniformly, each action a has a chance of ϵ/m (where $m = |\mathcal{A}|$ is the number of discrete actions) to be drawn. Furthermore, the random action can be still the greedy action: this implies that the actual probability of the greedy action to be chosen is $\epsilon/m + 1 - \epsilon$: not just $1 - \epsilon$.

The DQN algorithm is an instance of the classical Q-Learning [28], but with neural networks instead of tables. For this reason, the authors developed a couple of neat tricks to make DQN more stable during learning. The first trick is to use a *replay memory* to store transitions experienced by the agent. The replay buffer has the role of decorrelating the experience tuples, since these are sampled uniformly to then estimate the gradient of Q_ϕ. Correlated transitions is a notorious problem of RL, and so random sampling makes them to look more i.i.d. Another problem is related to the computation of the *targets* y towards which the action-values are regressed. As we will see, computing the targets involves the use of Q_ϕ, which is also what is being learned. So, as the parameters ϕ got updated, the estimates of Q_ϕ will change and then the targets y will "move": this, in fact, is the problem of the *moving targets*. To deal with this, we just introduce a *copy* of Q_ϕ, called the *target network*, $Q_{\phi'}$ (we just copy the weights), that is only used for the computation of the targets y, and then the weights ϕ' got slowly updated towards ϕ: e.g., after several gradient updates. An overview of the Deep Q-Learning algorithm is provided in listing 1.

3.2 Policy Gradient

The Policy Gradient (PG) family of model-free algorithms, compared to Q-learning, is more theoretically grounded, since the *gradient expression* derived from the performance objective $J(\theta)$ (Eq. 9) is used to directly learn a policy π_θ. Such formula is called the (sampled) *policy gradient*,

PoS(ISGC2022)018

Algorithm 1: Deep Q-Learning

Allocate replay memory $\mathcal{D}$ with capacity N
Initialize Q_ϕ with random weights, and make a copy $\phi' = \phi$
Assume discount factor γ, learning rate α, epsilon ϵ, and target update frequency K
Instantiate the environment, *env*

```
1  k = 0                                                         // init counter for target network
2  for episode, e = 1 to E do
3  |  s_0 = env.reset()                                          // get initial observation
4  |  for timestep, t = 0 to T − 1 do
5  |  |  k ← k + 1
6  |  |  a_t ~ π_ϵ(· | s_t)                                      // sample action from ϵ-greedy policy
7  |  |  s_{t+1}, r_t, d_t = env.step(a_t)                       // environment step, which applies a_t
8  |  |  Store transition (s_t, a_t, r_t, s_{t+1}, d_t) in D     // d_t tells whether s_{t+1} is terminal
9  |  |  Sample mini-batch B = {(s_i, a_i, r_i, s'_i, d_i)}_i from D
10 |  |  Compute targets: y_i = r_i + d_i · γ max_{a'∈A} Q_{ϕ'}(s'_i, a') for B
11 |  |  Update step by SGD: ϕ ← ϕ − α · ∇_ϕ 1/|B| Σ_{i∈B} (Q_ϕ(s_i, a_i) − y_i)^2
12 |  |  if k mod K == 0 then
13 |  |  |  Update target network: ϕ' ← ϕ
14 |  |  if done flag d_t is True then
15 |  |  |  Terminate episode
```

or the REINFORCE [27] estimator:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\Bigg(\underbrace{\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\big(a_t^{(i)} \mid s_t^{(i)}\big)}_{\text{likelihood}}\Bigg)\Bigg(\underbrace{\sum_{k=t}^{T-1}\gamma^{k-t} r\big(s_k^{(i)}, a_k^{(i)}\big)}_{\text{rewards-to-go}}\Bigg) \tag{12}$$

The policy gradient intuitively formalizes the notion of *trial-and-error* learning, since to discover which actions are the best, we have to first try some of them, and then evaluate their goodness. The evaluation of actions is provided by the second term in Eq. 12, while the first term basically computes their likelihood: simply put, the policy gradient increases the likelihood of good actions while decreasing the likelihood of sampling worse actions.

Apart from being more theoretically grounded, compared to Q-Learning, PG-based algorithms can be applied to both discrete and continuous action spaces, without any modification of the underling learning objective: just embed a suitable probability distribution in the policy network. An overview of the REINFORCE algorithm is showed in listing 2.

3.3 Actor-Critic

Unfortunately, the policy gradient described by Eq. 12 still suffers from *high-variance*. This is due to the fact that the gradient is obtained through a sample average of N trajectories, as well as due to the variance introduced by the rewards-to-go (or returns, G). An effective variance reduction

PoS(ISGC2022)018

Algorithm 2: REINFORCE

```
    Initialize policy π_θ with random weights
    Assume discount factor γ, learning rate α, and number of trajectories N
    Instantiate the environment, env
1   for k = 1 to K do
        /* Collect N trajectories (sequentially or in parallel)                 */
2       Initialize buffer: D = {}
3       for i = 0 to N − 1 do
4           Sample trajectory τ_i from π_θ, by interacting with env until termination, or t = T
5           Compute rewards-to-go: R_i = [ Σ_t γ^t r_t^(i) | ∀t = 0, 1, ..., |τ_i| ]    // is a vector
6           Store τ_i and R_i in D
        /* Update the policy parameters by gradient ascent                       */
7       θ ← θ + α · 1/|D| Σ_{τ_i,R_i ∈ D} Σ_{t=0}^{|τ_i|−1} ∇_θ log π_θ(a_t^(i) | s_t^(i)) R_{i,t}
8       Discard or clean D
```

technique for the PG is to subtract a *baseline function*, b. If we let b be the value-function, $V(s)$, by subtracting it to the returns (also corresponding to an empirical estimate of the action-value function, i.e. $Q(s,a)$), we obtained the so-called *advantage function*, $A(s,a)$:

$$A(s,a) = Q(s,a) - V(s), \tag{13}$$

which, intuitively, says how much better (or how much worse) action a, for state s, is compared to the *average action*, represented by the numerical value of $V(s)$: since the value function can be derived by averaging over action-values.

By switching the rewards-to-go for the advantage function, the overall interpretation of the policy gradient is still more intuitive, also having way lower variance:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)}) A(s_t^{(i)}, a_t^{(i)}) \tag{14}$$

Indeed, in order to plug the advantage in the gradient estimate we have to learn it first. Since we already have the returns G, we can use them to estimate $Q(s,a)$, and then learn a value network, V_ϕ, to approximate the value function, $V(s)$. In this way, we can recover an estimate for the advantage function. The additional value network is now called the *critic*, that has the general role of evaluating states, and if we rename the policy to be the *actor*, we derive a new family of agents that rely on two neural networks: the actor (policy), and the critic (state- or action-value function). Of course, when the input states are complex and demanding (like images), the two networks can share a common (e.g. convolutional) *backbone*, devoted to *feature extraction*, or, more generally, to *representation learning*. In this case, the way we learn the critic network is though regression towards sampled returns, as detailed in the code listing 3.

PoS(ISGC2022)018

Algorithm 3: Actor-Critic

Initialize actor π_θ, and critic V_ϕ with random weights
Assume discount factor γ, learning rates α and β, and number of trajectories N
Instantiate the environment, *env*

```
1  for k = 1 to K do
       /* Collect N trajectories (sequentially or in parallel)                        */
2      Initialize buffer: D = {}
3      for i = 0 to N − 1 do
4          Sample trajectory τ_i from π_θ, by interacting with env until termination
5          Compute the returns: G_i = [ Σ_t γ^t r_t^(i) | ∀t = 0, 1, ..., |τ_i| ]   // is a vector
6          Estimate advantages: Â_i = G_i − V_φ(s_i)        // s_i is a vector of states from τ_i
7          Store τ_i, G_i, and Â_t in D
       /* Update the critic first                                                     */
8      φ ← φ − β · 1/(2|D|) Σ_{τ_i,G_i∈D} ∇_φ ||V_φ(s_i) − G_i||_2^2
       /* Then, update the policy parameters by gradient ascent                       */
9      θ ← θ + α · 1/|D| Σ_{τ_i,Â_i∈D} Σ_{t=0}^{|τ_i|−1} ∇_θ log π_θ(a_t^(i) | s_t^(i)) Â_{i,t}
10     Discard or clean D
```

Line 5: $G_i = \left[\sum_t \gamma^t r_t^{(i)} \mid \forall t = 0, 1, \dots, |\tau_i|\right]$

Line 6: $\hat{A}_i = G_i - V_\phi(\mathbf{s}_i)$

Line 8: $\phi \leftarrow \phi - \beta \cdot \frac{1}{2|\mathcal{D}|} \sum_{\tau_i, G_i \in \mathcal{D}} \nabla_\phi \|V_\phi(\mathbf{s}_i) - G_i\|_2^2$

Line 9: $\theta \leftarrow \theta + \alpha \cdot \frac{1}{|\mathcal{D}|} \sum_{\tau_i, \hat{A}_i \in \mathcal{D}} \sum_{t=0}^{|\tau_i|-1} \nabla_\theta \log \pi_\theta\left(a_t^{(i)} \mid s_t^{(i)}\right) \hat{A}_{i,t}$

4. Reinforce-lib

In this section we present our *modern*, *modular*, *usable*, *understandable*, and *extensible* reinforcement learning library: see figure 2. Reinforce-lib is designed around simplicity, to allow faster learning of its APIs (Applicaion Programming Interfaces), but also to easily understand its source code. Moreover, it's modern relying on recent TensorFlow 2.8 [29] and `tensorflow-probability` [24], the latter for reliable implementations of probability distributions. In addition, the library aims to be generally applicable to a variety of problems: this is achieved by following the widely established `gym.Env` interface of the environment, from the OpenAI's gym [30] library. We also opted for a modular design, to allow for ease code (module) reuse, switching, and even custom extensions provided by the users themselves. The last design principle is the *completeness*: currently the library contains a variety of popular model-free agents, but in the long-term we aim to complement it with algorithms from both the model-based, and inverse sub-fields of reinforcement learning, since these will allow the practitioners to face additional classes of problems.

As anticipated, reinforce-lib already implements widely used model-free algorithms, such as: DQN [1] and extensions [31–33], DDPG [34], TD3 [35], SAC [36], A2C (synchronous variant of A3C [7]), PPO [37] with GAE [38], Rainbow [39], and IQN [40]. With also included a variety of *code-level optimizations* [41, 42], which are often crucial to achieve good or SOTA performance, although usually not described by authors, such as: reward/observation/return/advantage normalization and clipping, decaying schedules for hyperparameters (e.g. learning rate, epsilon, entropy), gradient clipping, and more.

PoS(ISGC2022)018

Figure 2: Overview of the principles around which reinforce-lib has been designed: (1) *usability*, allows for rapid prototyping and development of research ideas, (2) *extensibility*, enables the users to apply and/or adapt the provided algorithms to a variety of problems, finally (3) *completeness*, will permit to tackle a larger variety of class of problems.

PoS(ISGC2022)018

4.1 The `rl` Namespace

The library API is available through the `rl` namespace, and is mostly divided into:

- `rl.agents`: contains the `Agent` and `ParallelAgent` interfaces, as well as the single implementations of the model-free agents. Both `A2C` and `PPO` agents inherits from `ParallelAgent`, meaning that both support *environment-level parallelism* (i.e. multiple copies of the environment in parallel) though the Python `multiprocessing` module.
- `rl.networks`: it's where the `Network` interface resides, which is the base class for policies (either deterministic or stochastic), and value networks.
- `rl.memories`: defines the `Memory` and `TransitionSpec` interfaces. The former defines the abstract notion of a memory buffer, while the latter is used to describe the structure of a transition, i.e. what to store into the agent's memory. Having a transition specification allows memories to adapt automatically to what both the agent and environment request to be saved.
- `rl.layers`: some custom layers and wrappers to distributions used by `Network` for various purposes, like preprocessing.
- `rl.parameters`: contains useful decaying schedules (i.e. instances of `DynamicParameter`) that can be applied to various hyperparameters, so not only the learning rate.

In general, an agent have to define a transition specification that is used to instantiate and allocate its memory buffer, also having one or more networks (usually, a policy and a critic) used to interact with and learn from the environment.

4.2 API and Code Examples

An overview of the reinforce-lib APIs is summarized in list 1. Also, a couple of example code are provided in figure 3 and 4. Furthermore, our library is also debug-oriented (refer to figure 5), meaning that monitoring of useful quantities (e.g. the loss, norms of gradients, cumulative rewards, etc), as well as hyperparameters, is done through the `TensorBoard` visualization tool.

```
# Defines the structure of
  transitions
  transition_spec()
# Uses π to predict action
  act(state)
# Learning loop
  learn(...)
# Logic to update the
  parameters of the agent
  update(batch)
# Loading and saving
  load(path)
  save(path)
```

```
# Applies the network
  call(inputs)
# Defines the (default)
  architecture
  structure(...)
# Defines the output layer,
  used in structure()
  output_layer()
# Defines the loss function
  (e.g. policy gradient)
  subject to optimization
  objective(batch)
```

```
# Tells whether the buffer is
  full or not
  is_full()
# Stores an experience tuple
  store(transition)
# Returns a batch of
  transitions
  get_batch(batch_size)
# Retrieves all the
  transitions at once
  get_data(...)
# Logic to conclude a
  trajectory
  end_trajectory()
```

List 1: API overview of the code interfaces, respectively of: `Agent`, `Network`, and `Memory`.

```python
from rl import utils
from rl.agents import DQN
from rl.parameters import StepDecay
from rl.layers.preprocessing import MinMaxScaling
from rl.presets import Preset


# Fix the random seed; for local reproducibility
utils.set_random_seed(42)

# Preprocess states by min-max scaling, to scale them in [-1, 1]
min_max_scaler = MinMaxScaling(min_value=Preset.CARTPOLE_MIN,
                               max_value=Preset.CARTPOLE_MAX)

# Define a DQN agent for the CartPole environment
agent = DQN(env='CartPole-v1', name='dqn-cartpole', batch_size=128,
            policy='e-greedy', lr=1e-3, update_target_network=500,
            # `epsilon` is a DynamicParameter that is halved each 100 episodes
            epsilon=StepDecay(0.2, steps=100, rate=0.5),
            memory_size=50_000, gamma=0.99, seed=utils.GLOBAL_SEED,
            # customize the default, 2-layers network architecture
            network=dict(units=64, preprocess=dict(state=min_max_scaler)))

# Train the agent: evaluation occurs each 10 episodes, with checkpointed saving;
# also the first `512` steps are exploratory (i.e. random)
agent.learn(episodes=200, timesteps=500, save=True, render=False,
            evaluation=dict(episodes=20, freq=10), exploration_steps=512)
```

Figure 3: Example definition and training of a DQN agent on the CartPole environment.

PoS(ISGC2022)018

```python
# Load pretrained agent
agent.load(path='your-agent-path')

# Record the agent interacting with the environment
agent.record(timesteps=500, folder='video')
```

Figure 4: After having trained (and saved) an agent using the `learn` method, it's possible to `load` it to finally act in the environment, and also to `record` its interactions.

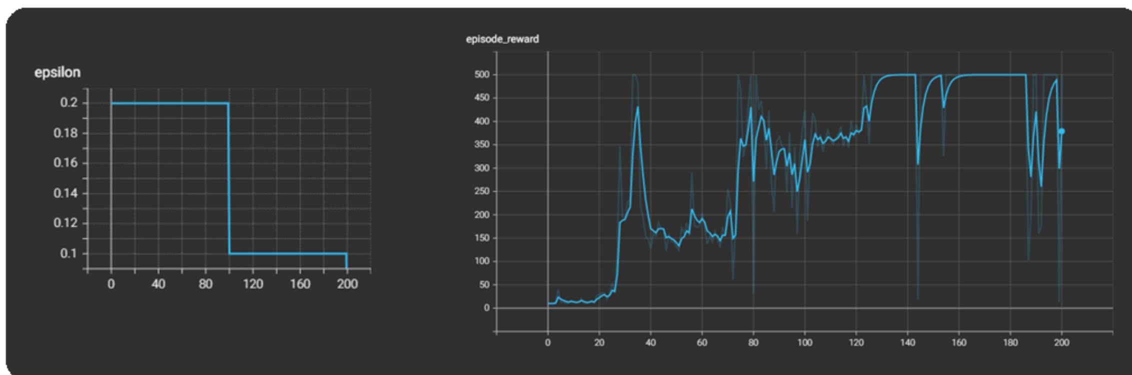


Figure 5: An example of two `TensorBoard` summaries. The plot on the left depicts the DQN's *epsilon* hyperparameter, as defined in figure 3. Whereas, the plot on the right depicts the learning curve (i.e. sum of rewards) of the same DQN agent.

5. Conclusions

We propose a modern, modular, and easy reinforcement learning library, called `reinforce-lib`. Although being in its initial development state, the library already features a whole set of popular model-free algorithms. In the long term, the library will provide support for recurrent policies, and improved exploration strategies, as well as, accomplishing completeness by implementing both model-based and inverse RL algorithms. Our main aim is to provide a comprehensive tool for convenient and rapid application and development of reinforcement learning ideas, to eventually novel research problems, especially targeting those scientific domains in which RL is not yet taken into account, or rarely applied in practice: thus, making the reinforcement learning paradigm to be more accessible to newcomers.

We believe reinforcement learning to have many applications in scientific fields, thus further improving over classical or deep learning baselines, being even able to provide an answer to previously infeasible problems like the amazing breakthrough AlphaFold [43] accomplished about *protein folding*. Today RL is mostly used in games, and control systems (often in simulation). Having the right tools, like the library we propose, could help reinforcement learning to find applications in many more scientific scenarios.

PoS(ISGC2022)018

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare et al., *Human-level control through deep reinforcement learning*, *nature* **518** (2015) 529.

[2] M.G. Bellemare, Y. Naddaf, J. Veness and M. Bowling, *The arcade learning environment: An evaluation platform for general agents*, *Journal of Artificial Intelligence Research* **47** (2013) 253.

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

[4] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning*, MIT press (2016).

[5] A. Krizhevsky, I. Sutskever and G.E. Hinton, *Imagenet classification with deep convolutional neural networks*, *Advances in neural information processing systems* **25** (2012) .

[6] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma et al., *Imagenet large scale visual recognition challenge*, *International journal of computer vision* **115** (2015) 211.

[7] V. Mnih, A.P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley et al., *Asynchronous methods for deep reinforcement learning*, in *International conference on machine learning*, pp. 1928–1937, PMLR, 2016.

[8] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward et al., *Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures*, in *International Conference on Machine Learning*, pp. 1407–1416, PMLR, 2018.

[9] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. Van Hasselt et al., *Distributed prioritized experience replay*, *arXiv preprint arXiv:1803.00933* (2018) .

[10] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen et al., *Learning to drive in a day*, in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 8248–8254, IEEE, 2019.

[11] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron et al., *Solving rubik's cube with a robot hand*, *arXiv preprint arXiv:1910.07113* (2019) .

[12] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche et al., *Mastering the game of go with deep neural networks and tree search*, *nature* **529** (2016) 484.

[13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez et al., *Mastering the game of go without human knowledge*, *nature* **550** (2017) 354.

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez et al., *A general reinforcement learning algorithm that masters chess, shogi, and go through self-play*, *Science* **362** (2018) 1140.

[15] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison et al., *Dota 2 with large scale deep reinforcement learning*, *arXiv preprint arXiv:1912.06680* (2019) .

PoS(ISGC2022)018

[16] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A.S. Vezhnevets, M. Yeo et al., *Starcraft ii: A new challenge for reinforcement learning*, *arXiv preprint arXiv:1708.04782* (2017) .

[17] O. Vinyals, I. Babuschkin, W.M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung et al., *Grandmaster level in starcraft ii using multi-agent reinforcement learning*, *Nature* **575** (2019) 350.

[18] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup and D. Meger, *Deep reinforcement learning that matters*, in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[19] G. Dulac-Arnold, D. Mankowitz and T. Hester, *Challenges of real-world reinforcement learning*, *arXiv preprint arXiv:1904.12901* (2019) .

[20] F. Chollet et al., *keras*, 2015.

[21] D. Ha and J. Schmidhuber, *World models*, *arXiv preprint arXiv:1803.10122* (2018) .

[22] P. Abbeel and A.Y. Ng, *Apprenticeship learning via inverse reinforcement learning*, in *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, C.E. Brodley, ed., vol. 69 of *ACM International Conference Proceeding Series*, ACM, 2004, DOI.

[23] P.-W. Chou, D. Maturana and S. Scherer, *Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution*, in *International conference on machine learning*, pp. 834–843, PMLR, 2017.

[24] J.V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore et al., *Tensorflow distributions*, *CoRR* **abs/1711.10604** (2017) [1711.10604].

[25] D.P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, *arXiv preprint arXiv:1412.6980* (2014) .

[26] D.E. Rumelhart, G.E. Hinton and R.J. Williams, *Learning representations by back-propagating errors*, *nature* **323** (1986) 533.

[27] R.J. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, *Machine learning* **8** (1992) 229.

[28] C.J. Watkins and P. Dayan, *Q-learning*, *Machine learning* **8** (1992) 279.

[29] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean et al., *Tensorflow: A system for large-scale machine learning*, in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, eds., pp. 265–283, USENIX Association, 2016, https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[30] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang et al., *Openai gym*, *CoRR* **abs/1606.01540** (2016) [1606.01540].

PoS(ISGC2022)018

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

[32] H. Van Hasselt, A. Guez and D. Silver, *Deep reinforcement learning with double q-learning*, in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[33] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot and N. Freitas, *Dueling network architectures for deep reinforcement learning*, in *International conference on machine learning*, pp. 1995–2003, PMLR, 2016.

[34] T.P. Lillicrap, J.J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa et al., *Continuous control with deep reinforcement learning*, *arXiv preprint arXiv:1509.02971* (2015) .

[35] S. Fujimoto, H. Hoof and D. Meger, *Addressing function approximation error in actor-critic methods*, in *International conference on machine learning*, pp. 1587–1596, PMLR, 2018.

[36] T. Haarnoja, A. Zhou, P. Abbeel and S. Levine, *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*, in *International conference on machine learning*, pp. 1861–1870, PMLR, 2018.

[37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, *Proximal policy optimization algorithms*, *arXiv preprint arXiv:1707.06347* (2017) .

[38] J. Schulman, P. Moritz, S. Levine, M. Jordan and P. Abbeel, *High-dimensional continuous control using generalized advantage estimation*, *arXiv preprint arXiv:1506.02438* (2015) .

[39] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney et al., *Rainbow: Combining improvements in deep reinforcement learning*, in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[40] W. Dabney, G. Ostrovski, D. Silver and R. Munos, *Implicit quantile networks for distributional reinforcement learning*, in *International conference on machine learning*, pp. 1096–1105, PMLR, 2018.

[41] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph et al., *Implementation matters in deep policy gradients: A case study on ppo and trpo*, *arXiv preprint arXiv:2005.12729* (2020) .

[42] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier et al., *What matters in on-policy reinforcement learning? a large-scale empirical study*, *arXiv preprint arXiv:2006.05990* (2020) .

[43] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger et al., *Highly accurate protein structure prediction with alphafold*, *Nature* **596** (2021) 583.

PoS(ISGC2022)018