A Hierarchical Two-tier Approach to Hyper-parameter Optimization in Reinforcement Learning

Juan Cruz Barsce[1], Jorge A. Palombarini[1,2,3], and Ernesto Martínez[4]

[1] Dpto. de Ingeniería en Sistemas de Información, Facultad Regional Villa María, UTN, Argentina
[2] GISIQ, Facultad Regional Villa María, UTN, Argentina
[3] CIT Villa María - CONICET-UNVM, Argentina
[4] Instituto de Desarrollo y Diseño CONICET-UTN, Argentina,
ecmarti@santafe-conicet.gob.ar

Abstract. Optimization of hyper-parameters in real-world applications of reinforcement learning (RL) is a key issue, because their settings determine how fast the agent will learn its policy by interacting with its environment due to the information content of data gathered. In this work, an approach that uses Bayesian optimization to perform an autonomous two-tier optimization of both representation decisions and algorithm hyper-parameters is proposed: first, categorical / structural RL hyper-parameters are taken as binary variables and optimized with an acquisition function tailored for such type of variables. Then, at a lower level of abstraction, solution-level hyper-parameters are optimized by resorting to the expected improvement acquisition function, whereas the categorical hyper-parameters found in the optimization at the upper-level of abstraction are fixed. This two-tier approach is validated with a tabular and neural network setting of the value function, in a classic simulated control task. Results obtained are promising and open the way for more user-independent applications of reinforcement learning.

Keywords: reinforcement learning, hyper-parameter optimization, Bayesian optimization, Bayesian optimization of combinatorial structures (BOCS)

1 Introduction

Generalizing from data in supervised learning involves a training process, where an algorithm is used to learn the model structure (representation) and parameters that best fit the available data. Training, in turn, depends on prior design decisions (representation and algorithms) that defines the hyper-parameters that are the constraints for data-driven learning of a functional map. Setting properly these hyper-parameters is crucial to the learning process, and can make the difference between mediocre and state-of-art model induction and generalization [13].

In machine learning, the process of finding the optimum set of hyper-parameters ψ can be addressed as a black-box optimization problem, stated as finding

$$\psi^* = \arg\max_{\psi} f(\psi \mid D) \tag{1}$$

being f an objective function that takes an input ψ given data D, and maximizes some metric, such as the prediction accuracy in supervised learning, or the expected average reward in reinforcement learning.

As a sub-area of machine learning, reinforcement learning (RL) [29] algorithms have achieved very significant milestones recently (specially in games),a after its performance against professional players in Poker [7] and against grandmaster players in StarCraft II [31]. However, successfully applying reinforcement learning algorithms involves pinpointing a set of hyper-parameters ψ such that the learning agent can fast converge to an optimal behavior policy using highly informative data. This is difficult in RL because data is not provided *a priori* with correct examples of N feature-labeled pairs $D = \{(X, y)_i\}_{i=1}^{N}$ as in supervised learning; instead, examples are sequentially generated through on-going interactions between a learning agent and its environment, obtaining data in the form of tuples of N observed transitions $D = \{(s, a, s', r)_i\}_{i=1}^{N}$, where s is an state, a is an action that has been applied in s, s' is the new state where the agent has arrived and r is the corresponding reward obtained.

As data in RL is sequentially generated based on previously seen state transitions, actions taken and received rewards, hyper-parameter setting is the most influential decision regarding the information content of the generated tuples. Moreover, interactions of the learning agent with its environment requires a proper balance between exploiting current knowledge by selecting the apparently best actions, and exploring seemingly sub-optimal actions to discover better, hopefully optimal, actions. In other words, agents have to experience informative transitions due to sub-optimal actions in order to learn from them and then converge towards an optimal policy. Furthermore, only tuples of transitions which are actually seen determine the values of parameters θ that the agent will use to define its policy and estimate the values $Q(s, a)$ of applying an action a in a given state s, which also influence the next set of data generated, and so on and so forth. This is the main difference with supervised learning algorithms, which learn their θ parameters by being trained with annotated examples from a fixed data set. In particular, when dealing with complex representation architectures such as deep neural networks, there are often thousands of parameters θ that must be learned and, added to the inherent variance of the RL learning process, the absence of an autonomous hyper-parameter optimization methodology make it very difficult to evaluate or replicate the efficacy of alternative architectures [11] [14].

Such comparison is difficult, as hyper-parameters in representations and algorithms used are usually manually tuned, which can be very ineffective [13], or optimized by resorting to very inefficient methods such as grid search, random search [5] or plain Bayesian optimization (BO) [19] [27]. As a commonly-used strategy in machine learning, random search performs a random sample

of the value of hyper-parameters in a local hyper-sphere surrounding the current maximum. On the other hand, Bayesian optimization performs a black-box optimization of f, resorting both to a prior distribution $f \sim P(y)$ and to the available points from previous queries (X, y), in order to compute the mean and variance for unseen inputs. This is typically predicted by Gaussian process (GP) regression [23], which is used to maximize an *acquisition function* that is cheap to optimize globally. The issue with random search is that the method uses very limited information about previous queries of f. On the other hand, while Bayesian optimization uses information regarding past queries, it also has two limitations similarly to random search: 1) it involves no assumption about the influence of hyper-parameters on the information content, and 2) it is inefficient at optimizing categorical hyper-parameters such as the RL algorithm selected and representation used for states and actions. Finally, notice that such existing methods for hyper-parameter optimization are all based on the underlying assumption that they are optimizing hyper-parameters for a supervised learning task, where cross-validation errors can be accurately estimated.

In this work, an autonomous hyper-parameter optimization algorithm named *RLOpt two-tier* is proposed. The algorithm assumes a hierarchical relationship between RL hyper-parameters, in a way that there is a set of structural hyper-parameters that are most influential in comparison to others, real-valued ones, so that must be optimized iteratively at two abstraction levels. With such an assumption, hyper-parameters that are assumed as crucial are often either discrete (such as the number of hidden layers in a neural network), categorical, integer or binary (such as an hyper-parameter that determines if the exploration rate is to be decreased over time or not) or composed of numerical values (such as the batch size). In other words, they represent a black-box combinatorial structure that is very expensive to sample.

Taking the above considerations into account, in this approach the structural hyper-parameters are optimized first by using a variant of Bayesian optimization to handle categorical hyper-parameters. Then, traditional Bayesian optimization is used to tune the real-valued hyper-parameters of the learning algorithm that depends of the first structural decision, starting with the best structure and trained parameters previously found.

Compared with traditional Bayesian optimization, the proposed algorithm has two main advantages:

1. It focus the optimization in one subset of hyper-parameters at a time, allowing to handle different levels of complexity at each optimization step.
2. By dividing the optimization in two tiers, the best models or hyper-parameters of one tier can be reused when optimizing the other. For instance, optimization of lower hierarchy hyper-parameters can start with the pre-trained parameters θ from the best model of higher-level hierarchy.

The proposed algorithm is validated against random search and standard Bayesian optimization in the classical Cart-pole and Pendulum control environments. This work is an extended version of [3], and is organized as follows: Section

2 describes the RL problem and Section 3 describes the Bayesian optimization algorithm and the BOCS algorithm, that are used to define the two-tier approach for RL hyper-parameter optimization. On the other hand, in Section 4, hyper-parameters in RL and the hierarchical assumption among them are detailed, whereas in Section 5 is outlined the proposed two-tier optimization approach, where Section 6 details the computational experiments made that validates the proposed two-tier approach. Finally, Section 7 compares this work with other similar approaches, and in Section 8 some concluding remarks are made and promising avenues of our current research work are stated.

2 Reinforcement learning

Reinforcement learning [29] is a sub-area of machine learning that has recent success in tasks such as StarCraft II [31], Go [28], Atari, Chess [25] and robotics [20]. It involves an autonomous agent that must learn to control an external environment while learning a control policy (a way of behaving) that maximize the cumulative or average of rewards received from such an environment over time. Formally, it can be stated as a Markov Decision Process (MDP), $(S, A, R(.), P(.), \gamma)$, where

- S is a set of environmental states.
- A is a set of actions available to the agent.
- $R(s)$ is an external function that assigns the agent a reward to state transition caused by the agent action taken at any state $s \in S$.
- $P(s' \mid s, a)$ is a function that determines the probability that the agent transitions from a state $s \in S$ to a state $s' \in S$ when an action $a \in A$ is taken.
- Finally, $\gamma \in [0, 1)$ is a real number that assigns a discount to values of future rewards, such that the return at a given time t equals $G_t = \gamma^0 r_t + \gamma^2 r_{t+2} + \ldots$.

The control policy is defined as a function $\pi(a \mid s)$, and represents the probability of taking a given action a when the environment is in a certain state s. With $\pi(.)$, the agent aims to maximize the *value function* $V_\pi(.)$ for every state it may be in, defined as the expected (cumulative or average) return starting from a given state at time-step t and following a given policy π thereafter. Formally such function must satisfy the Bellman equation [29], defined in Eqn. 2.

$$\begin{aligned} V_\pi(s) &= \mathbb{E}(R_t \mid s_t = s) \\ &= \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a)(r(s, a, s') + \gamma V_\pi(s')) \end{aligned} \tag{2}$$

The Bellman equation can also be described in terms of the expected value of a state s given that an action a is chosen and future actions will be selected using a given policy $\pi(s)$. In such terms, Eqn. 2 would be expressed as the function $Q(s, a)$, which represents the expected value of s and choosing an action a and

following $\pi(s)$ afterwards. Such distinction allows to differentiate the expected return of following a policy from the value of each individual action available at each state.

To solve a Markov Decision Process, a RL algorithm is used, that consists of an iterative process focused on improving the value function. Among basic algorithms, the most commonly used are Q-Learning [32] and *SARSA* [24]. Both algorithms compute the action-value function $Q(s, a)$ according to a temporal difference between the discounted value of $Q(s', a')$ of the next state and action, and the previous Q-value for the current state and chosen action. The learning rule based on temporal differences in Q-learning is given in Eqn. 3.

$$Q_{new}(s, a) = Q_{prev}(s, a) + \alpha(r + \gamma \arg\max_{a'} Q(s', a') - Q_{prev}(s, a)) \quad (3)$$

The difference between Q-Learning and *SARSA* is how they choose the next action a', where the latter selects the action based on the policy π, and thus it is an *on-policy* algorithm, whereas the former selects the best estimated action a' for the resulting state s', therefore it is considered as *off-policy.* Algorithms may also update the $Q(s, a)$ values of past states and actions that were responsible for reaching the current state, using a mechanism known as *eligibility traces* [29].

A crucial aspect in RL is the trade-off between exploration and exploitation, in which the agent has to choose between taking actions that are considered to be the best according to the current estimation of the optimal policy learned so far, or taking actions that are deemed as sub-optimal but makes room for the agent to discover better actions to exploit in the future.

To achieve this trade-off, a commonly used exploration policy is the ϵ-greedy policy, where the estimated best action a' is chosen with an $1 - \epsilon$ probability, and the other alternative actions are chosen at random with a low probability ϵ. Alternatively, the *Softmax* policy is also a common choice, where each action is selected based on the equation $\pi(a \mid s) = e^{Q(s,a)/\tau} / \sum_{a'} e^{Q(s,a')/\tau}$, where τ is an hyper-parameter that establishes the influence of the $Q(s, a)$ values in defining the action selection probabilities.

Each of the RL algorithms and policies have their own set of hyper-parameters that must be defined before the agent learning process begins. Common hyper-parameters include a learning rate that determines the speed of the convergence of the agent $\alpha \in (0, 1)$, an exploration rate $\epsilon \in (0, 1)$ if the policy is ϵ-*greedy*, and a discount factor $\gamma \in (0, 1)$ for future rewards. If the policy used is ϵ-greedy, an additional hyper-parameter known ϵ-decay rate can be used to reduce the value of exploration parameter ϵ after an episode, in order to increasingly lower the exploration rate of the agent after a given number of episodes has been experienced.

Due to the difficulties of performing calculations in big and possibly non discrete state spaces is that in most applications, the value function v is normally not computed directly in such cases but instead approximated through an estimation of the value function $V_\pi(s) \approx \tilde{v}_\pi(s \mid \theta)$ parametrized with learned weights $\theta = \theta_0, \theta_1, \ldots, \theta_p$. Such θ can comprise trivial models such as a simple

linear regression, or complex non-linear models such as deep neural networks. Outputs of such models include the Q-values for each action availabe at a given state, as it happens in [18], or the probability $\pi(s, a)$ of selecting an action a as it happens in policy gradient algorithms such as [26], among others.

In recent years, a particular architecture called deep Q network (DQN) [18] kick-started the adoption of deep learning architectures for encoding the policy learned by reinforcement learning algorithms (called Deep RL). Such architecture extends the classical Q-learning algorithm in different ways:

1. Prediction of the action-value function of a state s given by $Q(\phi(s), a \mid \theta)$, where $\phi(s)$ is a function that applies pre-processing on states (e.g. if s is an image, such pre-processing can consist in converting its RGB values to grayscale, resizing of the image, etc.). Having the estimation of Q, action selection is performed with an ϵ-greedy policy.
2. A different set of weights θ^- is used to predict the target y, in order to stabilize the learning process by preventing that both the prediction $Q(\phi(s), a \mid \theta)$ and its target y (defined below) are changed simultaneously at each gradient step. Therefore, target weights θ^- are not updated by gradient descent, but are replaced with weights θ after C steps.
3. Storing tuples of experience (s, a, r, s') in a replay memory. Each tuple contains key information of the observed transitions from a given state s to another state s' after applying an action a, with its corresponding reward r. Once stored, tuples are uniformly sampled in mini-batches, and are used to estimate the target value y, given by

$$y = \begin{cases} r \text{ if episode is terminated} \\ (r_j + \gamma max_{a'} \hat{Q}(\phi(s'), a' \mid \theta^-)) \text{ otherwise} \end{cases} \tag{4}$$

   Then, gradient descent is applied on $(y - Q(\phi(s), a \mid \theta))^2$ with respect to the weights θ.
   In order to separate tuples from the policy that generates it, an off-policy estimation is used. This allows learning from tuples of experience generated by different policies.
4. In order to avoid a high variance due to constant oscillation of network weights, rewards are clipped in the interval $[-1, 1]$ for rewards which are larger or smaller than both extremes of the interval.

Given its importance and success, the DQN architecture was included in this work to validate the proposed approach.

3 Bayesian optimization

Bayesian optimization [19] [27] is a sequential method for finding the point X that maximizes a costly black-box function $f : X^d \to \mathbb{R}$. This is performed in a

way that gives $f(X)$ a Bayesian treatment, by assuming it follows a prior distribution, and then by computing the posterior distribution based in the Bayes' rule and the previously sampled data points $D = (X_1, y_1), (X_2, y_2) \dots (X_n, y_n)$ (being each $y_i = f(X_i) + \varepsilon$). As $f(X_1), \dots, f(X_n)$ are assumed to follow a multivariate Gaussian distribution, then the statistical meta-model of the black-box is a Gaussian process [23]. This allows computing the mean μ and variance σ^2 of f for any point X_{n+1} over the input domain given the observations D using a closed-form functional approximation given by Eqn. 5

$$\begin{aligned} \mu_{n+1}(X) &= \mu_0(X) + k(X)^T K^{-1}(Y - \mu_0) \\ \sigma^2_{n+1}(X) &= k(X, X) - k(X)^T K^{-1} k(X) \end{aligned} \tag{5}$$

As the meta-model function f is expensive to query, next sampling points are selected by maximizing a surrogate function called *acquisition function*, being the *expected improvement* (EI) function the most common variant. Such function is used to sample candidate points X_{test} by calculating the predicted mean and variance of each of them and returning as output the expected improvement over an incumbent (that normally equals to the point $\tau = X^+$ that currently maximizes f). The EI function is given by Eqn. 6.

$$\begin{aligned} \alpha_{EI}(X) &= \mathbb{E}[f(X) - \tau] P[f(X) > \tau] \\ &= (\mu_n(X) - \tau)\Phi(Z) + \sigma_n(X)\phi(Z) \end{aligned} \tag{6}$$

where $Z = \frac{\mu_n(X) - f(X^+)}{\sigma_n(X)}$. Considering that, the Bayesian optimization procedure is defined in Algorithm 1

Algorithm 1: Bayesian Optimization

Input : Gaussian process hyper-parameters, acquisition function α, unknown black-box function f

1 **for** *evaluation* $= 1$ *to* N *evaluations* **do**
2 Obtain X_n by optimizing acquisition function $\alpha(X^d) \to \mathbb{R}$ using predicted μ_{n+1} and σ_{n+1} from a statistical model (e.g. Gaussian Process)
3 Query the objective function f at the point X_{n+1}
4 Add the result $f(X_{n+1})$ to D
5 Update the statistical model (e.g. Gaussian process)
6 **end**

Output: $\arg\max_X f(X)$

Despite of their applicability in optimizing hyper-parameters, Algorithm 1 cannot efficiently handle discrete or categorical hyper-parameters that make choices that are structural or over categories, such as choosing an exploration algorithm or different representations of states and actions. A recent variant of

BO helps addressing such shortcoming, where a modified version of Algorithm 1 and an acquisition function that uses semidefinite programming are employed to take such values as input in a way that the algorithm can be applied with scarce data [2]. This variant consists in modeling the objective function as a second-order model, given by Eqn. 7.

$$f(X \mid \alpha) = \alpha_0 + \sum_j \alpha_j X_j + \sum_{i,j>i} \alpha_{i,j} X_i X_j - \lambda \|X\|_2^2 \tag{7}$$

where $X \in \{0,1\}^d$, λ is a regularization parameter, and α is a vector of learned coefficients that follow a Bayesian treatment, assumed to follow a horseshoe prior [8] and a Gaussian posterior distribution. The acquisition function employed is based on the well-known Thompson sampling idea and optimizes a quadratic form of Eqn. 7. As such function is very inefficient to optimize in that form, it is relaxed from quadratic to a vector mathematical program, where binary variables are replaced by a variable vector of $d+1$ dimension in a $d+1$ dimensional unit hyper-sphere. To approximate it in polynomial time, this vector program is rewritten as a semidefinite (SDP) program, then converted back to a vector where a randomized rounding method is used to obtain the solution.

Taking into account such acquisition function, the BOCS algorithm consists of a BO loop that performs Gibbs sampling of a vector of coefficients α from a posterior distribution, updated iteratively upon new data. The procedure is presented in Algorithm 2.

Algorithm 2: Bayesian Optimization of Combinatorial Structures (BOCS)

Input : Posterior distribution $P(.)$, unknown black-box binary function f, λ

1 **for** *evaluation* $= 1$ *to* N *evaluations* **do**
2   Sample $\alpha_1, \ldots, \alpha_p$ from the posterior $P(\alpha \mid D)$
3   Obtain X_n by finding $\arg\max_X f_\alpha(X) - \lambda \|X\|_2^2$
4   Query the objective function f at the point X_{n+1}
5   Add the result $f(X_{n+1})$ to D
6   Update the statistical model with the posterior $P(\alpha \mid D)$
7 **end**

Output: $\arg\max_X f(X)$

4 Hyper-parameters in reinforcement learning

Reinforcement learning is mainly about transforming data from on-going interactions between an agent and its environment into knowledge in the form of an optimal control policy or agent behavior. Given a task, a reward function — which is externally provided and beyond the agent control— is used to provide goal-related hints to assess the goodness or badness of actions taken at different

environmental states visited. Any systematic procedure to learn an optimal policy in such a way can be named an RL algorithm. Mathematically speaking, any reinforcement learning algorithm is based on a number of hyper-parameters that affect its efficiency and effectiveness in the learning process. Hyper-parameters may include those directly related to balance exploration with exploitation such as ϵ for an ϵ-greedy control policy or the "temperature" for Softmax exploration based on Q-value estimation. There are also hyper-parameters that influence the speed of the learning process itself such as the discount rate γ for future rewards, eligibility traces λ and the learning rate α. They also may include a more abstract or categorical type of hyper-parameters such as the learning rule (e. g., temporal difference) and the type and corresponding parameters (e.g., structure of a neural network or the covariance function in a Gaussian process) used to represent the policy being learned, including deep representations of state features. Moreover, there may exist hyper-parameters used to define abstract representations through state aggregation and define options or extended courses of actions in temporal or relational abstractions. Sub-optimal or poor setting of some or many of these hyper-parameters significantly affect the learning process experienced by the agent using environmental reinforcements and the resulting approximation to the optimal policy. Bearing in mind the above considerations, and using as starting point the notion of abstractions specifics to reinforcement learning based on the PIAGeT principle [21], a rather simple hierarchy of abstraction for different hyper-parameters required to fully specify an RL agent is proposed. This is done in order to tackle the complex issue of autonomous learning of key decisions, and to lay the foundations for the two-tier autonomous reinforcement learning setting in Section 5. Given a sequential decision problem under uncertainty (e.g., goal-directed learning control problem) formulated as a Markov Decision Process (MDP), where learning an optimal policy for achieving a desired goal is the main focus of attention, four different layers of abstraction are proposed in order to specify the hyper-parameters in the detailed design of an RL agent. Each layer is specifically related to a given type of hyper-parameter (see Fig. 1). In the Sections below, the rationale behind this hierarchy and the corresponding hyper-parameters are discussed.

4.1 RL Problem statement

Without any loss of generality, at the uppermost level, the RL problem is stated here as a "flat" Markov Decision Process [21], where all environmental states are defined alongside all possible actions the agent can choose in each one of those states. At this level of abstraction, hyper-parameters may be related to state features and granularity for actions as well as all the parameters in any generative model, if any. Also, in the upper level of the hierarchy other aspects of the RL problem are defined, such as whether the task is addressed as episodic or non-episodic, whether time is represented as a discrete variable or if it is continuum as in Semi-Markov Decision Processes, if there exist hidden states such as in a Partially Observable MDP, or if state transitions due to actions taken are assumed as deterministic or probabilistic. With the possible exception of some rather

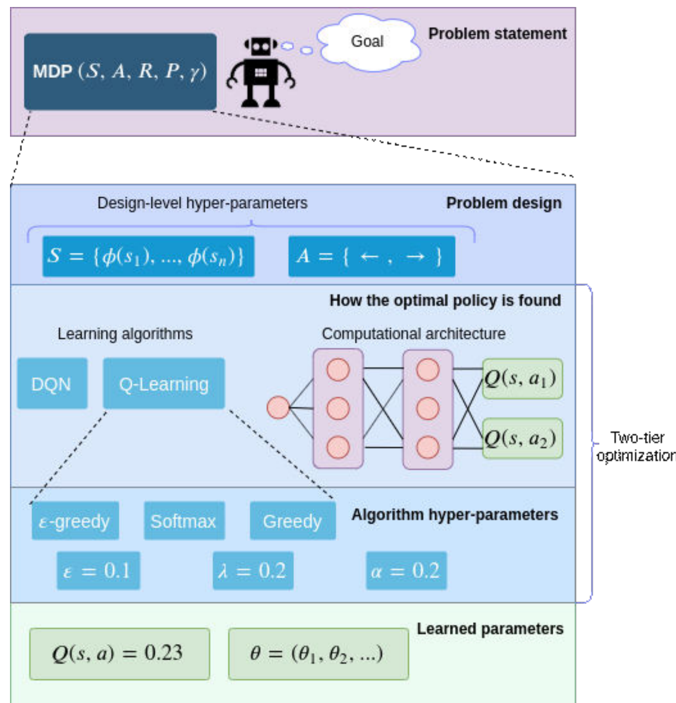

Fig. 1. Reinforcement learning hyper-parameter hierarchy.

simple tasks such as small grid-worlds, proper setting of all the involved hyper-parameters for solving the RL problem is a challenge which demands expertise in the inner workings of an RL algorithm and its computational implementation. Therefore, in the quest for autonomous setting of hyper-parameters in RL using Bayesian optimization, the proposed hierarchy in Fig. 1 is only a starting point for abstraction in which the two-tier optimization algorithm presented in Section 5 is based upon.

4.2 RL Problem design

At the highest-level of abstraction in the hierarchy of Fig. 1, hyper-parameters are related to the design aspects of an RL agent. Decisions at this stage determines how the representation of a learning task is structured as an MDP and constrained so as to facilitate policy learning using informative data generated from agent-environment interactions. At this level of abstraction, decisions made and conceptual models used aim giving the learning agent a short of *scaffolding* which is defined by key hyper-parameters whose values heavily influence what can be learned from data generated in on-going agent-environment interactions.

For instance, given a perception s from the environment, there is a design decision involving how such input will be internally represented by the learning

agent. If the environment has a discrete number of states and is simple enough such as a grid-world, the state where the agent performs its learning could directly match the perception s. However, the issue of representation type, e.g. propositional vs relational, must be addressed earlier on at this abstraction level. Moreover, in complex RL problems where the number of environmental states is significantly large or even a continuum, it is often better to represent the state using features in $\phi(s)$ of the perceived state s, where $\phi(s) = (\phi_1, \ldots, \phi_d)$ is the vector of features of the state s. For example, in [18], the actual agent perception s is a snapshot image of the screen of an Atari game, and $\phi(s)$ is a pre-processed sequence that stacks the last four frames of the game which is, hopefully, more informative for learning from agent-interaction tuples than mere individual frame.

Another important aspect is the relationship between state representation and perception abstractions and how to model the actions available to the agent. Accordingly, a design step involves how to define the granularity level of available actions depending on how perceptions are transformed into states and also on the type of representation used based on state features. The actions can be considered the same as the raw actions available, or they can be converted into composite actions; for example, in [18], a single action is taken four times before sensing the effect on the environment after being selected. In a relational model of environmental states, actions are also defined in a logic format using deitic representation, which makes room for abstracting a large number of low-level micro-actions into a single action.

Shaping internally the learning process may also demand properly setting some hyper-parameters. Often, the agent must achieve the goal (possibly, in a sub-optimal way) of making enough room for the credit-assignment process to provide a meaningful estimation of value functions. In reward design, given a task, additional *internal* rewards can be defined at some environmental states in order to incorporate bias into the agent behavior a priori knowledge that may speed up the learning process. For example, intrinsic rewards can be added as hints or guidelines to avoid certain risky states or motivate certain type of behaviors that help achieving the goal more efficiently. Another approach is adding an internal reward to drive exploration of given actions in non or sparsely visited states [22].

Regarding modeling the environment using a generative model, assumptions made about transition probabilities demand setting some hyper-parameters to trade off a priori knowledge or causal models with inductive modeling as the agent accumulates experience. Optimal setting of related hyper-parameters for predicting the environment may accelerate the learning process towards the optimal policy. For example, in [9], a (possibly inaccurate) model of the transition probabilities of the MDP may be incorporated in order to motivate the agent to purposefully explore a subspace of environmental states.

Finally, the real-valued hyper-parameter γ is another key hyper-parameter that determines the credit-assignment process for actions taken in the sequence of actions and state transitions towards the goal. Optimally setting its value

depends significantly on the task at hand. Hence, autonomous setting in an RL algorithm may help circumventing the drawbacks associated with neglecting the inter-dependency between the discount rate with other hyper-parameter values. Lacking a proper understanding of the importance of these hyper-parameters during the learning process often favors using constant default values. An example is choosing $\gamma = 0.9$, which could unnecessarily slow down the learning process experienced by the agent.

4.3 Finding the optimal policy

Going down in the hierarchy of Fig. 1, decision making revolves around choosing the algorithm and its underlying computational structure for accumulating knowledge so that the agent may converge to the optimal policy, and thus solving the RL problem for an externally provided reward function. Algorithm selection involves considering alternatives such as one-step Q-Learning or $SARSA$ to update state-action values, using gradient-based methods to learn the policy parameters directly using actor-critic methods or multi-step algorithms. On the other hand, the selection of the computational architecture in which the algorithm will store its knowledge may involve saving individual predictions of Q-values in tabular structures, or instead using a convolutional neural network architecture, a recurrent neural network, or a long-short term memory structure (LSTM), among others. Once the algorithm to accumulate knowledge resulting from agent-environment interactions as well as the structure to store and make predictions from it has been decided, the learning agent is equipped with the architecture required to focus exploring different available actions at the states encountered, to model state transitions and corresponding returns. Eventually, earmarked by all the above decisions, the agent is ready to learn the optimal policy for solving the task and (hopefully safely) generalize the policy to unseen state-action pairs.

4.4 Algorithm hyper-parameters

Once the learning algorithm and architecture have been chosen, on a third level of abstraction, the hyper-parameters that bias and constraint data gathering must be set. For example, in the Q-Learning algorithm, a hyper-parameter is used to balance exploration with exploitation (e.g. ϵ for choosing a non-greedy action or the profile for the annealing temperature τ in Softmax exploration) or to make value updates. If an algorithm with eligibility traces is used, then the hyper-parameter λ will determine the "memory" of the credit-assignment process in the sequence of seen tuples. The information content of the generated data depends significantly on the setting of these algorithm hyper-parameters. Moreover, both the rate of convergence of the learning process and the final approximation to the optimal policy are heavily dependent on the values chosen for these real-valued parameters. Another crucial hyper-parameter in this level of abstraction is the learning rate α, which determines the size of the update of the value or action-value function and, depending on the architecture, may

update the value on a table or determine the magnitude of the loss that is applied through gradient descent in a neural network, to name but a few examples. The algorithm hyper-parameters are not necessarily fixed during learning and may be aptly changed to advantage. For example, the amount of exploration or the learning rate can be steadily lowered as the number of episodes increases. This also gives rise to new hyper-parameters that also must be optimized at this lower level of the hierarchy.

5 Two-tier hierarchical Bayesian optimization of RL hyper-parameters

Considering the hierarchical taxonomy of hyper-parameters described in the foregoing section, in this work the proposed *RLOpt two-tier* autonomous tuning algorithm rest upon two separate Bayesian optimization strategies to perform a two-tier optimization of both structural ψ_a and solution-level hyper-parameters ψ_b of an RL agent. The latter ones correspond to the lower layers of abstraction in Fig. 1.

In our proposal, the higher level of optimization tunes the structural hyper-parameters located in the second layer of abstraction in the first place, and then the lower-level algorithm hyper-parameters of the third layer, kick-starting from the best categorical and structural decisions ψ_a and learned parameters θ. After a first-round optimization of categorical and representation hyper-parameters, they are kept frozen and the best combination of real-valued hyper-parameters found is used as a prior for the next round of optimization in the external loop, so it starts from a better initial point in optimizing categorical hyper-parameters. The two-tier optimization process is depicted in Fig. 2, where it is shown how both tiers are interrelated by an external and an internal loops for iterative improvement of the two sets of hyper-parameters.

To measure the performance of the optimization, it is proposed an objective function $f : \psi_a \cup \psi_b \to \mathbb{R}$ that maps a tuple (ψ_a, ψ_b) of both structural and solution level hyper-parameters of the algorithm to a real number that measures the overall performance of the learning agent, assuming an episodic task.

In order to calculate the value of $f(\psi_a \cup \psi_b)$, an RL agent is instantiated in a certain environment with hyper-parameters (ψ_a, ψ_b), and set to run for a certain number of episodes, in order to learn a policy to behave in such a way to maximize its received cumulative (or average) reward. Whenever the learning agent is given a new setting for its hyper-parameters, it resets all its prior knowledge about the policy previously learned in order to make a fresh start, unbiased by previous prior hyper-parameter values used. The new instance of the RL agent which run for a certain number of episodes under the same hyper-parameter setting is referred to here as a *meta-episode*.

At the upper-level hierarchy of considered hyper-parameters, Bayesian optimization for categorical structures (BOCS) is used to optimize the categorical, integer and representation hyper-parameters, taking them as binary variables. In such variables, each individual hyper-parameter $\psi_a^i \in \psi_a$ may represent the

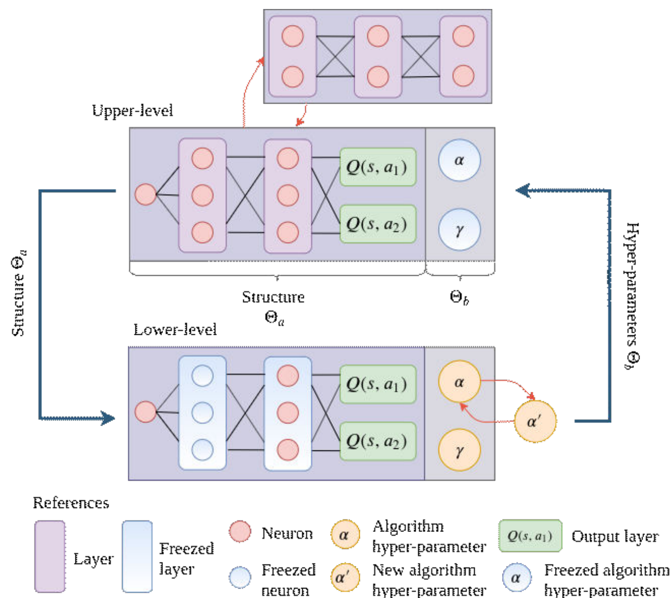

Fig. 2. Two-tier RL hyper-parameter optimization. In the upper-layer, algorithm hyper-parameters such as α are frozen. After the upper-layer optimization finishes, the structural hyper-parameters such as the number of neural network layers and a predefined set of its learned parameters are now fixed, while the rest of the hyper-parameters is fine-tuned in the lower loop.

presence or not of a certain feature (taking a value of $\psi_a^i = 1$ or $\psi_a^i = 0$, respectively). This feature can be categorical, for example to indicate whether if a dueling DQN architecture will be used, or can correspond to an integer value, e.g. to represent the number of hidden layers. While optimizing categorical hyper-parameters, the algorithm hyper-parameters that depend on them are kept frozen (i.e. they are not modified by the BOCS algorithm), starting from an initial vector of preset algorithm hyper-parameters ψ_b. Such initial vector is determined either from prior knowledge, default values or by the latest setting from the lower-level optimization loop as the external loop iteration proceeds.

Regarding the hyper-parameter optimization of the lower-level hierarchy that are real-valued, a standard Bayesian optimization approach is used once the upper-tier optimization has been made in the outer loop. Such an approach involves Gaussian assumptions and a Gaussian process statistical model to optimize the real-valued hyper-parameter that depends on those of at the higher level in the hierarchy, such as the learning rate α. In order to start the training from a solid basis, and considering that the upper-tier optimization has already performed some optimization improvement to the architecture and representation, the meta-episodes in the lower hierarchy are initialized with the best categorical hyper-parameters θ from the upper-layer optimization loop. In other

words, this means that whenever the agent is restarted while optimizing the lower-tier real-valued hyper-parameters, it is restarted to the parameters θ from the last setting from the upper level. This is used both to kick-start the agent with a promissory pre-trained model with weights θ, and to give the statistical model a reference of f, so BO starts with a performance benchmark of the best value of f obtained so far. In that sense, the lower-level hierarchy acts as a *fine-tuning* of the upper-level hierarchy model, as it starts optimizing a portion of a pre-trained neural-network. Bearing this in mind, if the underlying structural model is a neural network-based architecture, the lower-level optimization can also work with a reduced subset of the latest hidden layers, having the first hidden layers been kept fixed in terms of their links and weights. This allows a more stable and precise optimization of the remaining hyper-parameters, reducing the cost of training the whole architecture and representation, especially when such architecture involves many convolutional layers of filters.

After the lower-level hierarchy training is finished, a full evaluation cycle ends. Then, a new evaluation starts with a new set of N structural evaluations. The best vector of algorithm hyper-parameters ψ_b found in the latest evaluation is used as the initial set of the new structural cycle, a new iteration of the outer-loop optimization begins.

The full optimization algorithm is stated in Algorithm 3.

6 Computational experiments

6.1 Initial validation of the two-tier optimization

In order to obtain an initial validation, the proposed approach is run in a discretized version of the classic Cart-pole control environment, which consists of an environment with a cart that may be pushed either left or right, and it is holding a pole that can swing in both directions. The objective for the learning agent is to keep the pole balanced (i.e. by not letting it in a position where it will inevitably fall), while maintaining itself within certain limits. Each episode is terminated whenever the pole position is above or below 12 degrees from the vertical position, when the cart moves beyond a distance of 2.4 units from the center, or when 200 time-steps have elapsed. A reward of +1 is given after every time-step when the pole is still maintained upright, and a reward of -200 is assigned to the agent whenever the pole has fallen. The learning task is considered "solved" if the average reward reaches a threshold of +195, therefore an agent that follows an optimal policy would yield at least such reward, on average. The implementation used for the environment was the OpenAI Gym implementation [6].

The proposed Algorithm 3 is compared against two of the most commonly used methods for hyper-parameter in machine learning: random search and Bayesian optimization. To optimize RL hyper-parameters using BO, the RLOpt framework [4] is used. A total number of 30 meta-episodes were run for the three approaches, where the average reward was used to compute f on each

Algorithm 3: Bayesian optimization applied to optimize structural, architectural (e.g. neural network) and algorithm hyper-parameters

Input : RL-agent function f, prior set of algorithm hyper-parameters ψ_b, number of evaluations (N, E, M), Gaussian process hyper-parameters, acquisition function α, posterior distribution P, λ

1 Set algorithm hyper-parameters as the prior set ψ_b for the first N structural evaluations
2 **for** *full-evaluation* $= 1$ *to* E **do**
3   **for** *structural evaluation* $= 1$ *to* N **do**
4   Obtain ψ_a structural hyper-parameters that optimizes $\alpha_{BOCS}(.) \to \mathbb{R}$
5   Query the objective function f at the point (ψ_a, ψ_b)
6   Add the new triplet (ψ_a, ψ_b, f) to D_1,
7   Update the BOCS posterior $P(\alpha \mid D_1)$
8   **end**
9   Initialize D_2 with the triplet (ψ_a, ψ_b, f) that yielded the highest f
10   Set structural hyper-parameters as the best ψ_a structure and set the initial learned parameters as θ for the next M hyper-parameters evaluation
11   **for** *Algorithm hyper-parameters evaluation* $= 1$ *to* M **do**
12   Obtain ψ_b algorithm hyper-parameters that optimizes $\alpha_{EI}(.) \to \mathbb{R}$
13   Query the objective function f at the point (ψ_a, ψ_b)
14   Add the new triplet (ψ_a, ψ_b, f) to D_2
15   Update Gaussian process posterior functions $\mu_{n+1}(X)$ and $\sigma^2_{n+1}(X)$
16   **end**
17   Set prior algorithm hyper-parameters as the best ψ_b for the next N structural evaluations
18 **end**

Output: $(\arg\max_{(\psi_a,\psi_b)} f, \max f)$

meta-episode. In each iteration of the proposed algorithm, 10 meta-episodes were run to optimize the discrete hyper-parameters and 20 meta-episodes were used to optimize the real-valued hyper-parameters once the structural hyper-parameters were fixed. The structural hyper-parameters optimized were *algorithm* $\in$ {*Q-learning*, *SARSA*}, *eligibility-traces* $\in \{true, false\}$, *policy* $\in$ {ϵ*-greedy*, *Softmax*} and ϵ*-decay* $\in$ {*true*, *false*} (it only applies when the ϵ-greedy policy is selected). On the other hand, the lower-level algorithm hyper-parameters subjected to optimization were $\alpha \in (0, 1)$, $\epsilon \in (0, 1)$, $\gamma \in (0, 1)$, the number of bins that divides the cart position and speed, *n-bins* $\in (5, 20)$, and the number of bins used to discretize the pole angle position and its angular speed, *n-bins-angle* $\in (5, 20)$.

Results obtained are shown in Fig. 3, where the different maximum obtained for the f function (in this case, the maximum average reward) based on the current and all previous meta-episodes in the learning curve are shown. In the curves, the thick lines and their nearby curves correspond to the average maximum reached and the 95% confidence interval for ten simulations with different random seeds. As can be seen, the proposed method is consistently better in finding an optimized set of hyper-parameters that reach the maximum compared to

the other two methods that does not optimize the structural hyper-parameters. The average execution time for 30 meta-episodes with a single random seed was 7, 8 and 12 minutes for the random search, RLOpt, and for the RLOpt two-tier optimizer, respectively, resulting in 3.5 hours to run all experiments in a computer that consisted of four Intel i5-3230M CPU at 2.60GHz cores, and 12 GB of RAM.

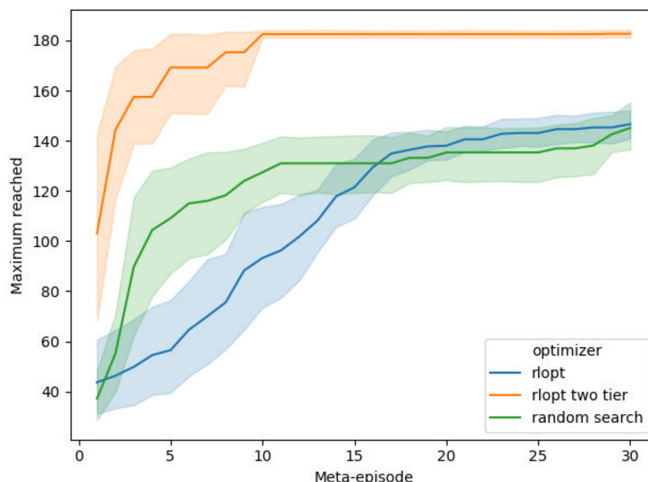

Fig. 3. Average maximum reached by each optimizer, per each meta-episode

6.2 Optimizing DQN hyper-parameters in neural networks

The following and successive sub-sections involves the validation of the two-tier approach using a neural network architecture that runs over the Cart-Pole environment. In this implementation, all restrictions regarding the discretization of the state and the intrinsic reward used in the precedent Section are removed, thereby taking as input the raw state tuple instead of applying a discretization, and no penalization is applied when the pole fells.

The underlying architecture is a deep-Q network that made updates to its target network weights every 10 steps, and consisted of the following structure: an input layer with four inputs that made up the perceived state s from the Cart-pole environment followed by up to 4 hidden layers with up to 32 neurons each. The output layer is composed of two outputs corresponding to the estimated Q values for the left and right actions, given the input state. Each of the hyper-parameter optimizers is set to run for 30 meta-episodes, where each consists of an agent running 100 episodes optimizing both structural and algorithm hyper-parameters. As can be seen, results of the experiments highlight

that the RLOpt two-tier approach for hyper-parameter optimization of alternative neural network architectures can achieve a consistent convergence towards the optimum hyper-parameters, and that the convergence rate is improved when the best model is used within several full evaluation loops. On the other hand, results also show that optimizing the hyper-parameters of categorical decisions for defining such an elaborated representation have less performance than resorting to a tabular and discretized version of value functions and states, which showcase that more complex representation architectures does not necessarily imply a better performance over tabular, simpler representations of states.

Regarding the RLOpt two-tier optimizer, its structural hyper-parameters were optimized using the BOCS algorithm, where 6 bits formed the combination of options, given by Table 1.

Table 1. The 6-bits representation used for structural hyper-parameters.

Bit position	Bit value = 0	Bit value = 1
6	no change in layers	+2 hidden layers
5	1 hidden layer	2 hidden layers
4	no change in neurons	+16 neurons per layer
3	8 neurons per layer	16 neurons per layer
2	no change in batch size	+32 batch size
1	batch size of 16	batch size of 32

Regarding the vector of real-valued algorithm hyper-parameters which is made up here of $(\alpha, \epsilon, \epsilon\text{-decay}, \gamma)$, whose ranges of values are as follows: $\alpha \in (1e-6, 1e-2)$, $\epsilon \in (0.01, 0.3)$, ϵ-decay $\in (0.001, 0.1)$ and $\gamma \in (0.01, 0.99)$. Finally, it is noticed that the optimization of the neural network parameters was performed by the Adam algorithm [16].

Regarding random search and the (monolithic) Bayesian optimization, they optimized all the structural and algorithm hyper-parameters at the same time. As both work with real-valued numbers, structural hyper-parameters are selected by taking the integer part of the hyper-parameter value.

In the first simulation, random search and Bayesian optimization were compared against the RLOpt two-tier approach for optimizing the neural network representation. The proposed approach resorts to 30 meta-episodes in 1 full-evaluation loop, that consisted of 10 structural evaluations and 20 algorithm hyper-parameter evaluations. Results are depicted in Fig. 4, which showcases the fact that the RLOpt two-tier approach exhibits a performance that surpasses the other two approaches, being ahead of both random search and RLOpt in the first episodes and being close to RLOpt, the second best approach, towards the end and with less variance. Average running times for this simulation times were 12, 13 and 14 minutes for RLOpt two-tier, standard Bayesian optimization and random search, respectively.

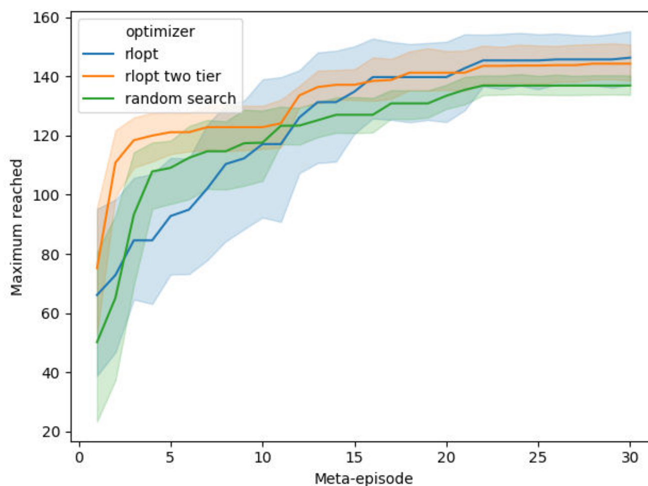

Fig. 4. Average maximum reached by each optimizer when tuning neural network hyper-parameters.

6.3 Optimizing DQN hyper-parameters sharing the best model

To validate the RLOpt two-tier capacity of optimizing pre-trained neural networks, this section shows how the algorithm performs when all the meta-episodes that occurs in the lower-level hierarchy optimization are started with the best configuration and hyper-parameters found in all the previous upper-tier meta-episodes. Results are shown in Fig. 5, where as can be seen that sharing the best representation has a positive impact in the average reward that can be obtained at the end of the learning curve. Average execution times for each random seed were 12 and 16 minutes for the RLOpt two-tier without and with sharing the best model, respectively.

6.4 Optimizing DQN in several full evaluation loops

Finally, an additional experiment was made to validate the performance of the proposed two-tier approach, by comparing the performance of several full-evaluation loops in a shared model execution. Each of the executions were divided as:

– Execution 1 (blue curve) used 1 full evaluation loop and its the same evaluation as the prior subsections, with 10 structural evaluations followed by 20 algorithm optimization meta-episodes.
– Execution 2 (orange curve) used 2 full evaluation loops that consisted on 2 sequences of 7 structural hyper-parameter optimization meta-episodes followed by 8 algorithm hyper-parameter optimization meta-episodes each.

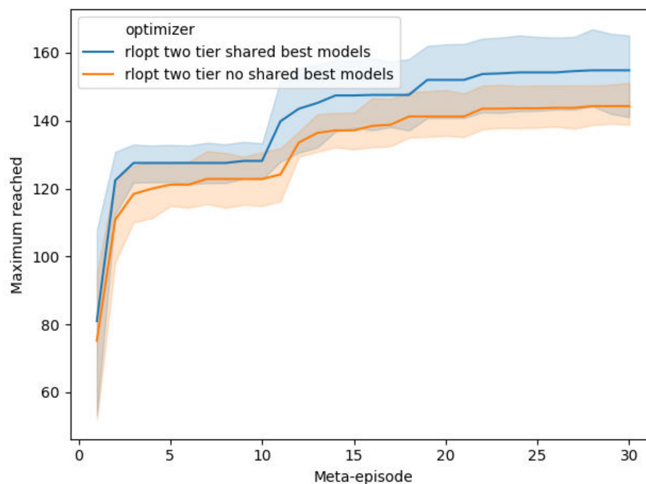

Fig. 5. Comparison between the average maximum reached when using the best structure and parameters in the lower-tier optimization.

– Execution 3 (green curve) used 3 full evaluation loops that were comprised of 3 sequences of 5 structural optimization meta-episodes followed by 5 algorithm optimization meta-episodes each.

Regarding the results, Fig. 6 depicts how the approach behaves when having 1, 2 and 3 full-evaluations, where a noticeable improvement can be seen in the maximum reached following the first fine-tuning pass of hyper-parameters. In particular, the optimization with 2 full evaluations is highlighted against the optimization with 1 full evaluation, because it starts behind in the first meta-episodes, and then catches up when it changes from upper to lower-tier BO optimization for the first time, maintaining high-level performance thereafter and onwards. The changes from the external loop to the internal loop can been appreciated by observing how the learning curves evolve in such meta-episodes, being the 2-full evaluation loops the alternative that gives rise to greater changes. The average running time for each random seed were 16, 17 and 14 minutes for the RLOpt two-tier with 1, 2 and 3 full evaluations, respectively.

6.5 Optimizing Soft actor-critic in continuous action space

The proposed approach is additionally validated with a more challenging environment with continuous action space, the Pendulum environment (using the OpenAI implementation of the environment), which consists on an agent that applies a real valued torque $\in [-2, 2]$ to a pendulum in order to swing it up in a way that it always remain upright. As DQN cannot handle a continuous action space, an algorithm that can take continuos actions was used for this

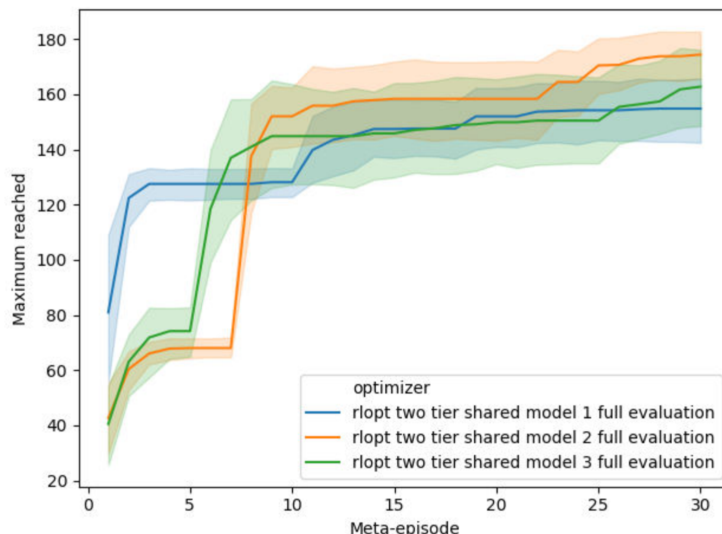

Fig. 6. Comparison of the average of the maximum reached by RLOpt two-tier executions with 1, 2 and 3 full loops.

environment: the Soft-actor critic (SAC) [10] algorithm. SAC is an off-policy RL algorithm that uses experience replay such as DQN, and has the following main differences:

1. It uses two Q-value networks instead of one in order to minimize bias.
2. It uses a policy network in order to sample the action. As the action space is continuous, actions are sampled from a normal distribution with Gaussian noise.
3. It adds a bonus reward based on the entropy of the policy distribution in order to promote exploration. A hyper-parameter $\alpha_{entropy}$ is used to control the magnitude of the bonus.
4. It uses an experience buffer, but the next action is sampled from current policy instead of using a buffered action. As a result, it trains a stochastic policy instead of a deterministic.

Learning experiments in the Pendulum environment are based on a similar setting as the previous subsections. They consisted of 30 meta episodes, made up of 100 episodes each. Structural hyper-parameters were the same as in the DQN network, defining a vector of 6 bits. Alternative settings for structural hyper-parameters in such vectors are as follows: 1) the number of hidden layers $\in \{2, 3\}$, 2) the number of neurons per layer $\in [32, 96]$ and 3) the size of the batch $\in [30, 110]$. On the other hand, the algorithm hyper-parameters to be optimized were: 1) learning rate $\alpha_{lr} \in (1e-3, 1e-2)$, 2) discount factor $\gamma \in (0.85, 0.99)$, 3) policy hyper-parameter $\rho \in (0.99, 0.999)$, that determines the magnitude of the

update of the network parameters theta, and 4) entropy coefficient $\alpha_{entropy} \in (0.01, 0.99)$, that determines the magnitude of the bonus entropy reward.

Results obtained are shown in Fig. 7, where the maximum of all the executions are summarized. It can be appreciated that sharing the best model has a huge effect in the performance of the agents, given that the top-3 performing executions involved sharing the best model found in the upper tier. A noticeable aspect about the best performing models was that they performed best when an increasing number of full evaluations were used. On the other hand, another aspect worth mentioning is that not sharing the best model had a negative effect in the performance of the RLOpt two-tier optimizer, which gives rise to a lower maximum compared with others. Running times were the following:

- RLOpt two-tier without sharing the best model: 1 hour 17 minutes on average, total of 10 executions: 12h57m.
- RLOpt two-tier sharing best model with 1 full evaluation: 1h07m on average (11h15m total)
- RLOpt two-tier sharing best models with 2 full evaluations: 1h14m on average (12h27m total)
- RLOpt two-tier sharing best models with 3 full evaluations: 1h14m on average (12h21m total)
- Random search: 1h01m on average (10h19m total)
- RLOpt with standard Bayesian optimization: 1h07m on average (11h17m total)

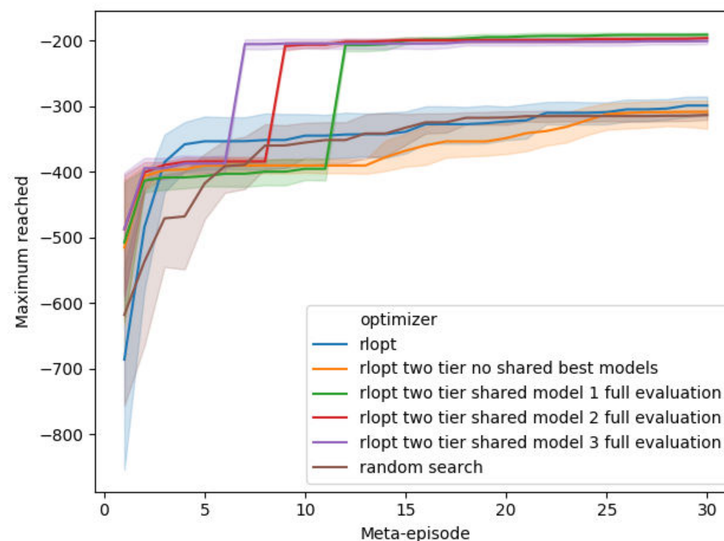

Fig. 7. Comparison of the different maximums reached by all executions.

7 Related work

Hyper-parameter optimization is a challenging problem in machine learning. It has been approached in different ways in the literature with some success in supervisory learning. In RL, the problem has received less attention and success is more complicated due to role of hyper-parameters in the information content of generated data. As the gradients of the hyper-parameters are not normally available, the most common methods used are "expert" manual tuning, random search [5], Bayesian optimization and *grid search*. In the latter case, hyper-parameter setting must perform an exhaustive search all over the hyper-parameter space which is prohibitively expense in time and money. The main drawback with grid search is that, as it does not use information of local or global optima in the search over hyper-parameter space, it may sample combinations of hyper-parameters that are non-informative yet there is no knowledge nor guidelines to follow.

Informal-tuning on the other hand is commonly used in many papers of the literature [13]. Hand-tuning involves optimizing the hyper-parameters without a solid methodology that commonly consists of tuning by trial-and-error and using common hyper-parameters or "expert rules". This is a sub-optimal strategy, as most experienced tuples are hardly informative, making difficult to reproduce both baselines and iterative meta-learning approaches, making it hard to transfer the "rules of thumb" used to another domains or algorithms.

As each evaluation of hyper-parameter is expensive, Bayesian optimization have been a common denominator for several frameworks that have been proposed in recent years. For instance, SMAC [12] [17] combined Bayesian optimization with random forests, Auto-Weka [30] combined Bayesian optimization with Gaussian process and Optuna [1], used Bayesian optimization and a pruning algorithm for automatically cutting unpromising trials.

Regarding neural network hyper-parameters, a method for finding optimized neural architectures that were recently proposed is neural architecture search, which automatically alter neural network structures in the midst of the training process, for example by changing activation functions, connections and neurons, and had promissory results in domains such as image classification [33]. Another approach for neural optimization is the use of population-based strategies [15], which consists of finding the optimal hyper-parameters by resorting to a population of models that are trained in parallel, and sharing solutions among them to increase the performance of individual so-called 'workers', namely optimizers. Both approaches involve mechanisms that are outside the scope of this work, however they can be good candidates for future additions to RLOpt so as to integrate them in the upper-level of the hierarchy when optimizing more complex structures that includes convolutional neural networks.

8 Concluding remarks and future directions

In this work, a novel approach that addresses the optimization of both categorical and real-valued RL hyper-parameters, assuming a hierarchical relationship be-

tween them was presented for optimizing RL agents for task involving either both a tabular (discretized) and approximated representation of states and actions. The validation in the classic Cart-pole and Pendulum environments highlights that the proposed RLOpt two-tier approach performs consistently better than the monolithic optimization of the real-valued hyper-parameters alone. It was also shown that this two-tier hierarchy can also be used as a way to fine-tune a pre-trained neural network structure, starting from a pre-defined structure and heuristically tuned parameters. That was the distinctive feature that made the two-tier approach better than the monolithic approach, because as each meta-episode consists in searching in a portion of the total hyper-parameter space (as it was divided between meta-episodes that optimize hyper-parameters of the structure and meta-episodes that optimize hyper-parameters at the algorithm level), sharing the best model allowed to kick start the optimization from a better initial condition, something that was not possible when optimizing over the whole search space as in monolithic Bayesian optimization.

In this line, our current research efforts are focused on:

- Extending the concept of a hierarchical relationship considering the hyper-parameters in architectures that includes convolutional neural networks.
- Use the two-tier approach to optimize hyper-parameters in policy gradient algorithms.
- Employ the fine-tuning to efficiently transfer the performed learning to a different reinforcement learning task.
- Incorporate mechanism to prune unpromising trials and to integrate parallelism into the two-tier optimization, so as to maximize resource efficiency.

References

1. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A Next-generation Hyperparameter Optimization Framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining - KDD '19. pp. 2623–2631. ACM Press, Anchorage, AK, USA (2019)
2. Baptista, R., Poloczek, M.: Bayesian Optimization of Combinatorial Structures. arXiv:1806.08838 [cs, math, stat] (Jun 2018)
3. Barsce, J.C., Palombarini, J.A., Martínez, E.: A Hierarchical Two-tier Approach to Hyper-parameter Optimization in Reinforcement Learning. In: Anales Del Simposio Argentino de Inteligencia Artificial (ASAI) 2019. Sociedad Argentina de Informática, Salta, Argentina (Sep 2019)
4. Barsce, J.C., Palombarini, J.A., Martínez, E.C.: Towards Autonomous Reinforcement Learning: Automatic Setting of Hyper-parameters using Bayesian Optimization. CLEI Electronic Journal 21(2), 1:1–1:22 (Aug 2018)
5. Bergstra, J., Bengio, Y.: Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research 13(Feb), 281–305 (2012)
6. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym (2016)
7. Brown, N., Sandholm, T.: Superhuman AI for multiplayer poker. Science p. eaay2400 (Jul 2019)

8. Carvalho, C.M., Polson, N.G., Scott, J.G.: The horseshoe estimator for sparse signals. Biometrika 97(2), 465–480 (Jun 2010)
9. Epshteyn, A., Vogel, A., Dejong, G.: Active reinforcement learning. In: Proceedings of the 25th International Conference on Machine Learning. pp. 296–303 (2008)
10. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv:1801.01290 [cs, stat] (Aug 2018)
11. Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., Meger, D.: Deep Reinforcement Learning that Matters. arXiv:1709.06560 [cs, stat] (Sep 2017)
12. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential Model-Based Optimization for General Algorithm Configuration. In: Learning and Intelligent Optimization. pp. 507–523. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (Jan 2011)
13. Hutter, F., Lücke, J., Schmidt-Thieme, L.: Beyond Manual Tuning of Hyperparameters. KI - Künstliche Intelligenz 29(4), 329–337 (Nov 2015)
14. Islam, R., Henderson, P., Gomrokchi, M., Precup, D.: Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control. arXiv:1708.04133 [cs] (Aug 2017)
15. Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W.M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., Kavukcuoglu, K.: Population Based Training of Neural Networks. arXiv:1711.09846 [cs] (Nov 2017)
16. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs] (Jan 2017)
17. Lindauer, M., Eggensperger, K., Feurer, M., Falkner, S., Biedenkapp, A., Hutter, F.: SMAC v3: Algorithm Configuration in Python. GitHub (2017)
18. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature 518(7540), 529–533 (Feb 2015)
19. Močkus, J., Tiesis, V., Zilinskas, A.: The application of Bayesian methods for seeking the extremum. In: Dixon, L., Szego, G. (eds.) Towards Global Optimisation 2, pp. 117–129. North-Holand (1978)
20. Nagabandi, A., Konoglie, K., Levine, S., Kumar, V.: Deep Dynamics Models for Learning Dexterous Manipulation. arXiv:1909.11652 [cs] (Sep 2019)
21. van Otterlo, M.: The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for Adaptive Sequential Decision Making under Uncertainty in First-Order and Relational Domains. No. 192 in Frontiers in Artificial Intelligence and Applications, IOS Press, Amsterdam (2009), oCLC: 313654110
22. Pathak, D., Agrawal, P., Efros, A.A., Darrell, T.: Curiosity-driven Exploration by Self-supervised Prediction. In: International Conference on Machine Learning. pp. 2778–2787 (Jul 2017)
23. Rasmussen, C.E., Williams, C.K.I.: Gaussian Processes for Machine Learning. Adaptive Computation and Machine Learning, MIT Press, Cambridge, Mass., 3. print edn. (2008)
24. Rummery, G.A., Niranjan, M.: On-Line Q-Learning Using Connectionist Systems. CUED/F-INFENG/TR 166, Cambridge University Engineering Department (1994)

25. Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., Silver, D.: Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. arXiv:1911.08265 [cs, stat] (Nov 2019)
26. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs] (Jul 2017)
27. Shahriari, B., Swersky, K., Wang, Z., Adams, R., de Freitas, N.: Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the IEEE 104(1), 148–175 (Jan 2016)
28. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. Nature 529(7587), 484–489 (Jan 2016)
29. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. Adaptive Computation and Machine Learning, MIT Press, Cambridge, Mass, 2nd edn. (2018)
30. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 847–855. KDD '13, ACM, New York, NY, USA (2013)
31. Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J.P., Jaderberg, M., Vezhnevets, A.S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T.L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., Silver, D.: Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature pp. 1–5 (Oct 2019)
32. Watkins, C.J.C.H., Dayan, P.: Q-learning. Machine Learning 8(3-4), 279–292 (May 1992)
33. Zoph, B., Le, Q.V.: Neural Architecture Search with Reinforcement Learning. arXiv:1611.01578 [cs] (Nov 2016)