Towards Autonomous Reinforcement Learning: Automatic Setting of Hyper-parameters using Bayesian Optimization

Juan Cruz Barsce

Dpto. de Ing. en Sistemas de Información,
Fac. Reg. Villa María - UTN,
Villa María, Argentina
jbarsce@frvm.utn.edu.ar

and

Jorge A. Palombarini

Dpto. de Ing. en Sistemas de Información,
GISIQ, Fac. Reg. Villa María - UTN,
CIT Villa María - CONICET-UNVM,
Villa María, Argentina
jpalombarini@frvm.utn.edu.ar

and

Ernesto C. Martínez

Instituto de Desarrollo y Diseño
CONICET-UTN,
Santa Fe, Argentina
ecmarti@santafe-conicet.gob.ar

Abstract

With the increase of machine learning usage by industries and scientific communities in a variety of tasks such as text mining, image recognition and self-driving cars, automatic setting of hyper-parameter in learning algorithms is a key factor for achieving satisfactory performance regardless of user expertise in the inner workings of the techniques and methodologies. In particular, for a reinforcement learning algorithm, the efficiency of an agent learning a control policy in an uncertain environment is heavily dependent on the hyper-parameters used to balance exploration with exploitation. In this work, an autonomous learning framework that integrates Bayesian optimization with Gaussian process regression to optimize the hyper-parameters of a reinforcement learning algorithm, is proposed. Also, a bandits-based approach to achieve a balance between computational costs and decreasing uncertainty about the Q-values, is presented. A gridworld example is used to highlight how hyper-parameter configurations of a learning algorithm (SARSA) are iteratively improved based on two performance functions.

1   Introduction

In recent years, with the notable increase of computational power in terms of floating point operations, a vast number of different applications of machine learning algorithms have been attempted, yet optimization of hyper-parameters is needed in order to obtain higher levels of performance. Such is the case of supervised learning algorithms like random forests, support vector machines and neural networks, where each one of them has its own set of hyper-parameters exerting influence on attributes such as model complexity or the learning rate of the algorithm, which are key issues in order to reap important benefits such as a better extrapolation to unseen situations, a reduction in computational times, a decrease in the model complexity, among others. Such hyper-parameters are often manually tuned and their correct setting makes the difference between mediocre and near-optimal performance of algorithms [1]. Several approaches have been proposed in order to optimize the hyper-parameters to reduce the error generated by a bad configuration of the model (e.g. [2, 3, 4, 5, 6]), with methods such as random search, gradient search, Bayesian optimization, among others.

In the particular case of reinforcement learning (RL) [7], as opposed to supervised learning, there are not correct examples of desired behavior in the form of datasets containing a label for each vector of features, but instead a learning agent should seek for trade-off between exploitation with exploration so as to find a way of behaving using external hints. Given a state s of the agent's environment, the RL problem is about obtaining the optimal policy (a function which, given sensed states, defines the action that should be chosen) in order to maximize the amount of numerical scores (rewards) provided by the environment. For the most commonly studied tasks that can be solved by using RL, where the agent may take a long sequence of actions receiving insignificant or no reward until finally arriving to a state for which a high reward is received (see [8]), the fact of having a delayed reward signal means larger execution times, so optimizing hyper-parameters using optimization strategies like Grid Search, Random Search, Monte Carlo or Gradient-based methods is not suitable for efficient autonomous learning. Alternatively, the Bayesian optimization strategy provides an approach designed to maximize the effectiveness of an expensive-to-evaluate function since it is derivative-free and less prone to be caught in a local minimum [9].

Among RL algorithms, such as Q-Learning [10] or SARSA [11], there are several parameters to be configured prior to the execution of the algorithm, e.g. the learning rate α or the discount factor γ. In trivial environments such as a gridworld, the impact of a change in a parameter like the exploration rate ϵ of an ϵ-greedy policy in an agent run is often easy to understand, but in the case of more complex environments and parameters such as the number of N planning iterations in a Dyna algorithm [12, 13], the cut-off time of a given episode, or the computational temperature τ in a *Softmax* policy, the impact that changes to these parameters may have in the agent's learning curve to solve a given task is normally unclear prior to its execution. Furthermore, when comparing the learning of different agents using different combinations of hyper-parameters, it is often unclear which one outperforms the other because it has superior learning rule, or simply because it has a better parameter setting. Moreover, if the environment is complex enough such that each learning episode is computationally expensive, having a bad parameter configuration could result in higher execution times that can eventually give rise to mediocre performance with a limited budget for experimentation[3].

In this work, a novel approach that employs Bayesian optimization to find a good set of hyper-parameters that improves the commonly used policies for a RL agent is proposed. In order to learn a near-optimal set of hyper-parameters using previous agent-environment simulated interactions, a Bayesian optimization framework is presented. In such an approach, a Gaussian process regression model is trained by using two functions that measure the performance of the agent in the simulated environment for unseen hyper-parameter settings. For meta-learning, a bandit-based algorithm is used to achieve a balance between the reduction of uncertainty about Q-values and the computational costs of active exploration of untried action-state pairs. Finally, it is discussed an example that shows how the agent immersed in a gridworld iteratively converges to a near-optimal policy by querying the objective function for meta-learning of hyper-parameters which gives rise to learning episodes with increasingly improved sets of hyper-parameters. Results obtained demonstrate that autonomous reinforcement learning clearly outperforms a default configuration of the parameters in a RL algorithm.

This work is an extended version of [14] [1], where a Bandits-based meta-learning layer has been added aiming to significantly reduce the computational expense involved in autonomously minimizing the variance of the different sequences of actions of a reinforcement learning agent - each with a different setting of hyper-parameters - to advantage. The proposed variance reduction is performed bearing in mind the fact that, in [14], most of resampling effort was invested in trying non-promising combinations of hyper-parameters (e.g. points whose metric is much lower than the average for alternative combinations), so performing additional

---

[1]Source code is available on-line at `https://github.com/jbarsce/rloptvb`.

simulations in those bad configurations is a waste of computational resources that can instead be directed at doing more simulations with promising combinations (e.g. at points where the acquisition function is maximized, as explained in section 2.3). This is the main purpose that motivates the addition of another meta-learning component to decide, after evaluating the objective function at each point, if the outcome is good enough compared to others in order to resample the objective function. To this aim, new experiments have been performed, showing that a very significant reduction of computational times can be achieved with different meta-learning bandits-based approaches. However, for some bandit algorithms computational gains are not so important as one can expect, which is shown in Section 4.5. Finally, in order to compare how the proposed approach performs against another common method for hyper-parameter optimization, a comparison between the convergence towards the optimum of the proposed algorithm against random search has been added, as it is shown in Section 4.4.

The article is organized as follows. In Section 2 a revision of the different methods to be used later is presented, including reinforcement learning, Bayesian optimization, Gaussian processes, meta-learning algorithms and bandit-based meta-learning; such methods are discussed in the context of autonomous setting of a learning algorithm hyper-parameters. This is followed by a section devoted to the proposed framework of autonomous reinforcement learning, Section 3, RLOpt, which aims to implement fast convergence to near-optimal agent policies while minimizing the number of queries in the objective function at the meta-learning level. This is followed by Section 4, which presents the different computational experiments that were made in order to validate the proposed RLOpt approach. Finally, the work is summarized and the future research directions are presented in section 5.

2 Background

2.1 Related Work

The task of optimizing the hyper-parameters of machine learning algorithms is very challenging. A correct setting of hyper-parameters makes a significant impact on the effectiveness for the task being addressed, affecting the generalization errors in ways that are not easy to anticipate. Morever, only a small fraction of hyper-parameters are relevant to guarantee fast learning with few examples [2]. As the gradients are normally not available, most methods for setting hyper-parameters are based on optimizing them using pointwise function evaluations; this is often called black-box or derivative-free optimization. Alternatively, a common approach is manual tuning, which consists of setting the hyper-parameters arbitrarily and evaluating the generalization errors until the algorithm reaches some desirable performance for a given task. This is often used because it is fast (no algorithmic implementation is required) and sometimes the user can gain some empirical insights about the influence of hyper-parameters on learning method efficacy. However, hand-tuning of hyper-parameters is costly in terms of time and money, hinders the reproducibility of results obtained and limits machine learning to expert users, among other drawbacks [3, 2]. As an alternative to hand-tuning, another common approach is grid search, which consists of training a model over every combination of a subset of the hyper-parameter space Θ, in order to find a configuration $\theta \in \Theta$ that maximizes some metric. One of the problems of grid search is that it requires an amount of computational resources that increases exponentially with the number of hyper-parameters. As a result, a lot of poor hyper-parameter combinations are tried unecessarily [2]. In this regard, random search, an algorithm that consists on a random, independent sampling of the hyper-parameter space instead on a fixed sampling, is proved to outperform grid search [2]. As an alternative to gradient-free optimization in neural networks, a gradient-based hyper-parameter optimization method was proposed in [15], that resorts to obtaining gradients by performing an exact reverse of stochastic gradient descent with momentum.

The main advantage of exhaustive search methods is that they are easy to run in parallel to easy the computation burden. However, they are based on samples that are independent from each other, and therefore they do not use the information of previous evaluations, so the convergence towards the optimum is slowed down. This is very significant when the budget of computational resources is scarce. As an alternative method aimed for expensive to evaluate functions, the sequential model-based optimization (SMBO) (a common algorithm is Bayesian optimization) consists of resorting to a probabilistic model in order to determine, according to the previous observations, what is the next point that yields, depending on the probabilistic model, the highest probability of optimizing an expensive performance function [9, 16, 17]. Empirical evaluations highlight that SMBO performs better than random search on hyper-parameter tuning in some applications (see e.g. [4]).

Regarding hyper-parameter setting for reinforcement learning algorithms, common hyper-parameters are the step-size parameter α, that regulates the speed of learning; the discount factor γ, that determines how future rewards are discounted (explained in section 2.2), policy related parameters such as ϵ or τ, that determines how the agent select its actions seeking to trade-off exploitation with exploration. In some

algorithms, the trace parameter λ, that determines the decay rate of credit-assignment to previously visited states may be difficult to set. Each one of these hyper-parameters has been studied separately (for example, see [18], [19] and [20] for the tuning of α, [21] for optimizing γ and [18] and [22] for adapting λ). A meta-learning approach has been proposed in [23] to tune the hyper-parameters of a reinforcement learning for a problem related to animal behavior. The proposed approach is rather limited as the information to tune the hyper-parameters is obtained from dopamine neuron firing in animals. For high-level hyper-parameter setting in a reinforcement learning task, a relevant problem recently studied in this area is the problem of algorithm selection, where the hyper-parameter can be the learning algorithm itself (for example, [24]). Considering the scope and limitations of previously references in the field, our main motivation is to propose a general framework to autonomously tune all the hyper-parameters in a reinforcement learning algorithm so as to make hyper-paraneter setting part of the learning problem related to Q-values.

2.2 Reinforcement Learning

As a method that can be combined with a large set of algorithms such as deep learning representations, as presented in [8], and employed in applications such as self-driving cars [25] and in games such as Go [26, 27], reinforcement learning (RL) is a form of computational learning where an agent is immersed in an environment E and its goal is to converge to an optimal policy by performing sequences of actions in different environmental states and receiving reinforcement signals after each action is taken. At any discrete time t, the environment is in a state s that the agent can sense. Each agent action a is applied to the environment, making it to transition from state s to a new state s'. A frequent setting is one that uses delayed rewards, i.e. by returning a non-zero reward whenever a goal state is reached and 0 otherwise. In such a setting, the agent could not normally determine whether an action is good or bad until a reinforcement is received, so the agent must try different actions in different states without a priori knowing the final outcome. For example, consider an agent learning to play Chess, where a check-mate situation would lead to a reinforcement of 1 or -1 depending on if it was a victory or a loss, whilst being in every other state would lead to a reinforcement of of 0.

For the cases where the RL problem has a finite set of actions and states, and the sequences of $(state, action)$ satisfies the Markov property, then the problem can be formally defined as a Markov Decision Process (MDP) where $S = \{s_1, s_2, ..., s_n\}$ is the set of states, $A = \{a_1, a_2, ..., a_m\}$ is the set of actions and $P_a(s, s')$ is the probability that the agent, being in state s, transitions to state s' after taking an action a and receive a reward r. An episode is defined by the finite sequence $s_0, a_0, r_1, s_1, a_1, \ldots, a_{n-1}, r_n, s_n$. At each time step t, the agent senses its environmental state s_t and selects the action $a_t \in A$ to take next using the (estimated) optimal policy. The latter summed up the experience gathered in previous interactions with the environment and tends to maximize future rewards. To balance the search for short-term versus long-term rewards, future rewards are discounted by a factor $\gamma \in [0, 1]$, such that the total reward R_t at time t is given by $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ... + \gamma^n r_{t+n}$.

In order to solve a task using RL, the learning agent employs a *policy* π that defines, being in the environmental state s, what action must be taken in each given time t. With a given policy, the agent aims, for all states, to learn to maximize its *action-value function* $Q_\pi(s, a)$, which represents the expected reward that would be obtained when starting from state s by taking action a and following policy π afterwards. Given s, a and π the action-value function satisfies the *Bellman equation*, expressed as

$$
\begin{aligned}
Q_\pi(s, a) &= \mathbb{E}(R_t \mid s_t = s, a_t = a) \\
&= \sum_{s'} P(s' \mid s, a)(r(s, a, s') + \gamma V_\pi(s'))
\end{aligned}
\tag{1}
$$

where $P(s' \mid s, a)$ is the probability that the agent transitions from state s to state s' after taking action a, $r(s, a, s')$ is the reward obtained by applying the action a in state s and transitioning to state s' and $V_\pi(s')$ is the expected reward to obtain starting from state s and by following policy π. Solving Eq. (1) by applying iterations is what the different algorithm implementations of RL attempt to achieve. One of the most common algorithms is Q-Learning [10], which uses temporal difference at each time-step t in order to update $Q(s, a)$. Q-Learning is an *off-policy* algorithm, in the sense that each $Q(s, a)$ is updated considering the best value of Q in the next time-step, regardless of the policy the agent actually uses. The update step is given by

$$
\begin{aligned}
Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \\
&\quad \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))
\end{aligned}
\tag{2}
$$

Other well-known reinforcement learning algorithms includes *on-policy* algorithms such as SARSA [11], *model-based* algorithms such as Dyna versions of Q-learning and algorithms that incorporate *eligibility traces* such as $Q(\lambda)$ [28], among many others. All of them have a fixed set of hyper-parameters such as the step-size α, which impacts on the temporal difference update of $Q(s,a)$, or n, the number of times the model is used to simulate experience after each step as in Dyna-Q.

Regarding the agent's action-selection policy, a common one is ϵ-*greedy*, where the agent *explores* actions that are a priori suboptimal, by taking a random action with probability ϵ, and *exploits* its current knowledge by taking the action that is believed to be the optimal one with probability $1 - \epsilon$. Another common policy that the agent can follow in its learning policy is Softmax, where the agent takes an action with a probability that depends on its $Q(s,a)$ value. The most common implementation of this algorithm is the Boltzmann Softmax, which defines the probability of taking the action a in state s as

$$\pi(a \mid s) = \frac{\exp\{Q(s,a)/\tau\}}{\sum_b \exp\{Q(s,b)/\tau\}} \tag{3}$$

where τ is the computational temperature. By using this policy, the agent starts by employing an exploratory policy at the beginning with a"high temperature", switching over time to a more greedy policy as the value of the hyper-parameter τ is increasingly reduced, i.e. exploration "cools down".

2.3 Bayesian Optimization

As an approach typically used to maximize an expensive to evaluate function such as the one measuring the performance of a reinforcement learning agent in an uncertain environment, Bayesian optimization [29] seeks to optimize an unknown objective function $f(X)$, where $X \in \mathbb{R}^n$, by treating it as a black-box function and defining a probabilistic model, a prior distribution, over its nature. Then, the Bayes' theorem is used to update the posterior belief about its distribution after observing the data $D_n = \{(X_1, f(X_1)), (X_2, f(X_2)), ..., (X_n, f(X_n))\}$ of new f queries (the algorithm for BO is stated in Algorithm 1 [9]). In order to maximize the efficiency under a limited amount of possible queries i.e. a budget of queries for sampling f, Bayesian optimization (BO) resorts to a cheap probabilistic surrogate model in order to determine the next point to query. This point is selected by maximizing a function called *acquisition function*. To begin with, sampling methods such as Latin hypercube [30] are used. The acquisition function determines how much the agent exploits the current knowledge and selects points with the highest probability of being a new maximum of f versus how much the agent explores a priori sub-optimal regions aiming to discover a reduced, yet feasible region of interest where the function maximum may be located with high probability. Emphasizing exploitation accelerates convergence, but the agent may often fall in a local optimum. On the other hand, emphasizing exploration ensures a more comprehensive sampling, but can significantly increase the time and cost needed to learn Q-values. An acquisition function commonly chosen is the *expected improvement function* [31, 32], because it provides a sensible trade off between exploration and exploitation by weighting the amount of improvement of a given point X with regards to the current maximum X^+ by the probability that such point X will improve over the current (known) maximum. The expected improvement acquisition function is given by

$$\begin{aligned} \alpha_{EI} &= \mathbb{E}(f(X) - f(X^+))P(f(X) > f(X^+)) \\ &= \Phi(Z)(\mu_n(X) - f(X^+)) + \phi(Z)\sigma_n(X) \end{aligned} \tag{4}$$

where X^+ is the point corresponding to the highest observed value of f, whereas Φ and ϕ are the cumulative distribution function and probabilistic density function of the standard Normal distribution, respectively, and $Z = \frac{\mu(X) - f(X^+)}{\sigma(X)}$.

In the Algorithm 1, it can be appreciated how, given an acquisition function, the objective function is queried in a controlled manner. The core sentence of the algorithm can be observed in line 2, where the acquisition function is optimized aiming to obtain the query point X_t that, influenced by the chosen prior distribution, is the point whith higher probability of being the maximum of the objective function.

2.4 Gaussian Processes

In a supervised learning task, the agent is given examples in the form of correct feature-response pairs $D_n = \{(X_i, f(X_i))\}$, $i = 1, \ldots, n$ which are used to learn inductively how to make predictions for unseen X. Features or inputs are normally $X \in \mathbb{R}^D$, whereas responses can be either real valued, in such case the agent is learning an input-output *regression* map, or categorically valued, i.e. $f(X) \in \{C_1, C_2, ..., C_n\}$. In the

---

Algorithm 1: Bayesian Optimization

Input : α_{model} - the acquisition function

1 for $t = 1, 2, ...$ **do**

2 $X_t = \arg\max_X(\alpha_{model}(X \mid D))$

3   Sample the objective function $y_t = f(X_t) + \epsilon$

4 $D_{t+1} = add(D_n, (X_{t+1}, y_{t+1}))$

5   Update the statistical model

6 end

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

---

latter case the agent is learning a *classification* task. Such feature/response pairs are fitted to training data, in such a way that the agent minimizes a loss function $L(f(X), \hat{f}(X))$ that measures the cost of predicting using the model $\hat{f}(X)$ instead of the actual (unknown) function, $f(X)$. A common loss function is the mean-squared error, given by

$$MSE = n^{-1} \sum_i^n (f(X_i) - \hat{f}(X_i))^2 \tag{5}$$

Since the function $f(X)$ is unknown and expensive to query, a common approach is to assume that it follows a multivariate Gaussian distribution defined by a prior mean function $\mu_0(X) \to \mathbb{R}$ and a covariance function $k(X_i, X_j) \to \mathbb{R}$. Based on this assumption and by incorporating the pairs D_n obtained from querying the objective function n times, we are using a *Gaussian process* [33] as the surrogate model. A Gaussian process (GP) places a prior over the data set D_n, and, by applying the *kernel trick* [33], it constructs a Bayesian regression model as follows: given D_n, the mean and variance of a new point X is given by

$$\mu_{n+1}(X) = \mu_0(X) + k(X)^T K^{-1}(Y - \mu_0) \tag{6}$$

$$\sigma_{n+1}^2(X) = k(X, X) - k(X)^T K^{-1} k(X) \tag{7}$$

where K is the covariance matrix, whose element $K_{ij} = k(X_i, X_j)$ and $k(X) = (k(X_1, X), ..., k(X_n, X))$ is a covariance vector composed of the covariance between X and each of the points X_i for $i = 1, 2, ..., n$. On the other hand, the covariance function k defines the smoothness properties of the samples drawn from the GP. The most common choice is the squared exponential function [33], given by

$$k_{se}(X_i, X_j) \;\; = \;\; \sigma_f^2 \exp\{-\frac{1}{2}(X_i - X_j)^T \tag{8}$$
$$diag(l)^{-2}(X_i - X_j)\} + \sigma_n^2 \delta_{ij}$$

where σ_f^2 is the variance of the function f, σ_n^2 is the noise signal, l is a vector of positive values that defines the magnitude of the covariance and δ_{ij} is the Kronecker delta such that $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. This function is infinitely differentiable and it is often considered as unrealistically smooth for several optimization problems [4]; an alternative is the Matérn covariance function. Assuming no prior knowledge about the hyper-parameters $\theta_{GP} = (\sigma_f^2, \sigma_n^2, l)$, a common approach is to maximize

$$\log P(y \mid X, \theta_{GP}) = \;\; - \;\; \frac{1}{2}(y - \mu_0)^T K^{-1}(y - \mu_0)$$
$$- \;\; \frac{1}{2}\log|K| - \frac{n}{2}\log 2\pi \tag{9}$$

where y is the vector containing the observations $f(X_1), f(X_2), ..., f(X_n)$ and μ_0 is the vector containing the prior means. Regarding θ_{GP}, approaches such as Latin Hypercube Sampling or the Nelder-Mead optimization method can be used in order to obtain values that maximizes the likelihood of the dataset given the GP hyper-parameters.

2.5 Meta-learning

The integration of algorithms for Bayesian optimization of parameters in a RL algorithm defines a more abstract layer of learning known as *meta-learning* [34, 35]. Meta-learning seeks to imitate the human

capacity of fast generalizing concepts after seeing just a few examples, and is also referred to as *learning to learn*. Meta-learning is closely related to transfer learning where an agent learns to perform better in a task by resorting to knowledge gain in successfully solving similar tasks [36], even learning the learning algorithm itself [37]. Meta-learning is aimed at learning the specific bias of the task the agent is trying to do, i.e. the set of constraining assumptions that influence the choice of hypothesis for explaining the data. These biases allow the agent to gain prior knowledge that increase the efficiency in solving similar tasks [37]. Meta-learning involves at least two learning systems: a low level system that generally learns fast and an abstract learning layers that normally learns slower and whose purpose is to improve the learning efficiency at the low level system [36]. A well-known meta-learning example in machine learning is the boosting method [38], that consists on a set of "weak" learners that are trained sequentially and their predictions are weighted and combined to form a "strong" classifier that enjoys of a sensible higher prediction capability for class separation.

Regarding meta-learning approaches more specifically geared towards reinforcement learning, recent works include the incorporation of a general purpose RL agent as a meta-learning agent, used to train a recurrent neural network [39, 36]. Also, a model-agnostic meta-learning layer that learns to generalize by performing a low number of gradient steps [40], which can also be used in supervised and unsupervised learning has been proposed. Other meta-learning approaches have shown the effectivity of using Bayesian optimization at a more abstract learning layers, particularly in reinforcement learning for robotic applications (e.g. [41, 42]), where such layer is used to learn a policy for a damaged robot, updating the model after each iteration. This allows, for example, that the robot be able to walk by adapting its policy to a loss in motor functionality using knowledge acquired in previous faulty conditions.

2.6 Bandit algorithms

The multi-armed bandit or N-Armed bandit problem is a decision problem in which a gambler must choose among N arms or slot machines (also referred as "one-armed bandits"), where each arm has a latent probability distribution. The gambler seeks to maximize the value received from the slot machines, having a limited budget of pulls to be tried. The decision problem arises from the gambler having to decide between pulling those arms which rewarded a high payoff (exploitation) and pulling apparently less profitable arms where the value received was a lower value or unexplored arms that have not yet been pulled (exploration). This is a somewhat simplified reinforcement learning setting but without state related association of rewards received (it can be formalized as a one-state Markov decision process). This problem has several variants, but a common formulation consists of $N \in \mathbb{N}^+$ arms where $D = \{\Phi_1(\mu_1, \sigma_1), ..., \Phi_N(\mu_N, \sigma_N)\}$ is the set of latent probabilistic payoff distributions corresponding to each $i = 1, \ldots, N$. At every discrete time-step $t = 0, 1, \ldots$, the gambler has to choose which arm $a_i(t) \in \{a_1(t), \ldots, a_N(t)\}$ to pull, receiving in turn a reward $r \sim \Phi_i(\mu_i, \sigma_i)$, from which she may calculate the sample mean and the sample standard deviation, $\hat{\mu}_i$ and $\hat{\sigma}$, respectively. The problem of maximizing the sum of received rewards, $\sum r$, is what bandit algorithms attempt to solve. In this section there are listed some common ones (based on [43]), which are also those used in this work.

Two common policies, as in the reinforcement learning problem, are ϵ-greedy and Softmax. In the case of ϵ-greedy, the policy for choosing an arm follows

$$a_i(t) = \begin{cases} \arg\max_{a_{i=1,\ldots,N}} \hat{\mu}_i(t) & \text{with probability } 1 - \epsilon \\ a_i & \text{with probability } \epsilon/N \end{cases} \tag{10}$$

This means that the best arm will be selected with a $1 - \epsilon$ probability, and with ϵ probability a random arm is selected. A common variant of ϵ-greedy is the greedy policy, which is the ϵ-greedy policy with $\epsilon = 0$ i.e. a policy which always selects the arm whose sampled mean is higher. On the other hand, the Softmax policy (as seen on Equation 3) states that each arm has a probability of being selected distributed in proportion to the sample mean, and given by

$$P(a_i(t)) = \frac{\exp\{\tau\hat{\mu}_i(t)\}}{\sum_b \exp\{\tau\hat{\mu}_i(t)\}} \tag{11}$$

Other common family of policies are the Upper Confidence Bound algorithms [44], which consist in algorithms that maximizes the upper bounds of a confidence interval. Two algorithms used in this work are the UCB1 and the UCB1Tuned algorithms. In the UCB1 algorithm, the arm to be selected is determined in proportion of the sampled mean of that arm and a factor between an amount proportional to the amount of times each

arm has been pulled and the current discrete time t. Analytically this is given by

$$a_i(t) = \arg\max_{a_{i=1,\ldots,N}} \left[\hat{\mu}_i(t) + \sqrt{\frac{2\log t}{n_i}} \right] \tag{12}$$

where n_i is the amount of times the i-th has been pulled. The UCB1Tuned algorithm is similar to UCB1, but it is extended by taking the variance into account by weighting the factor of UCB1 by an amount proportional to the sampled variance. Therefore, UCB1Tuned is given by:

$$a_i(t) = \arg\max_{a_{i=1,\ldots,N}} \left[\hat{\mu}_i + \sqrt{\frac{\log t}{n_i} \min\left[\frac{1}{4}, V_i(n_i) \right]} \right] \tag{13}$$

where $V_i(n_i) = \hat{\sigma}_i^2 + \sqrt{\frac{2\log t}{n_i}}$.

In this work, a meta-learning layer that combines a bandit learning algorithm and Bayesian optimization to control the performance, querying and selection of the hyper-parameters of a RL learning agent at a lower layer is proposed.

3 RLOpt Framework

The autonomous reinforcement learning framework proposed in this work, *RLOpt*, consists of the integration of Bayesian optimization as a meta-learning layer in a reinforcement learning algorithm, in order to take full advantage of the agent's past experience with its environment given a different hyper-parameter configuration. The distinctive feature of RLOpt is the autonomous seeking for a good set of hyper-parameters that maximizes the efficiency of the agent learning an optimal policy without requiring any difficult-to-understand inputs from the user. In order to be able to optimize the outcome of each learning episode of an RL agent A, its interaction with a simulated environment given a set of hyper-parameters is treated as a supervised learning task at the meta-level of learning. In such a task, the set of hyper-parameters θ constitutes the input to a random function $f_A(\theta) \to \mathbb{R}$ which measures the influence of hyper-parameters on the learning curve. The output of such function is the performance metric in learning to solving the task. In other words, the hyper-parameters of the RL algorithm are taken as predictors of a real-valued performance metric for optimal policy learned so far, as in supervised learning; however, the distinctive aspect of the proposed framework is that it uses a supervised learning approach for hyper-parameters in a higher level of abstraction to improve to speed up the learning curve experienced by the agent while seeking to find in the best policy through interactions with the environment. Therefore, the objective of the RLOpt framework is resorting to data gathered $D_n = \{(\theta_i, f_A(\theta_i))\}$ as evaluative feedback to train a regression model, and then use Bayesian optimization to learn good algorithm configurations for A with a minimum number of queries, i.e. learning episodes. On the other hand, RLOpt has its own set of hyper-parameters Θ at the highest level of abstraction which includes the number of queries that A will perform and the parameters of the GP model such as the covariance function.

The framework consists of six components, as it is depicted in Fig. 1, separated by RL learning and a meta-learning layers. The components are described below:
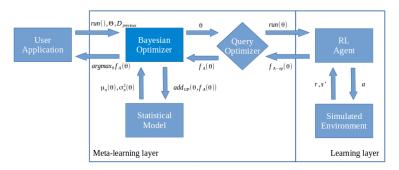

Figure 1: The Autonomous RLOpt Framework.

1. The first component is a user application U, which sets the hyper-parameters Θ of the framework and adds previous agent-environment interaction pairs $(\theta, f_A(\theta))$, if they are available.

2. The second component is the Bayesian Optimizer module BO, which uses a statistical model and Bayesian optimization approach to decide what configuration is best to maximize the efficiency of the learning curve. The BO module, according to Θ, is set to run a certain amount of meta-episodes (i.e. queries to $f_A(\theta)$ which represents higher abstraction level episodes where several RL agent episodes are run under the same θ configuration).

3. The third component is the query optimization module QO, which stores all the pairs of the previous queries $(\theta, f_A(\theta))$ and, after receiving new θ vectors from BO, decides how many queries the agent will perform with the same θ configuration, in order to ensure that at least a small sample of queries in relevant regions are performed so as to reduce the impact of the stochasticity and to have a more accurate sample of the real value of f for a given θ. The purpose of the QO component is to balance the trade-off between the precision of the $f(\theta)$ point and the computational cost of those queries.

4. The fourth component, is the reinforcement learning agent A, which receives configuration vectors from QO and interacts with the environment, learning a policy by iteratively sensing the environment state and taking actions seeking a trade-off between exploitation and exploration. After receiving vectors of hyper-parameters, the agent starts each new simulation from having no prior knowledge about the policies found in previous instances with other vector of hyper-parameters.

5. The fifth component is the environment E, which provides the simulation environment used to train the agent by receiving inputs from A that can modify its state and returns rewards accordingly.

6. Finally, the sixth component is the regression model (in this work a Gaussian process is employed) at the meta-level of learning which iteratively incorporates new data into its modeling dataset, whenever it is available, and use its current data in order to make a regression of the objective function at the level where hyper-parameters are learned. This regression model is employed to obtain a new configuration θ that maximizes the acquisition function so as to choose, hopefully, the next configuration of the agent learning A where the maximum of f_A has the greatest probability of being located, considering the previous meta-episodes. In the current implementation of the RLOpt, the statistical model employed is a Gaussian process; the main benefit of this model in the framework is that, by assuming a Gaussian prior about the data, it can learn after a few queries instead of needing a vast amount of training examples as methods such as deep learning models (see e.g. [45]). After receiving the signal from the $run()$ method initiated by the user U, the BO proceeds to instantiate A, E and the Gaussian Process as the statistical model.

Based on prior experience, if it is previously provided by the user, the BO module sets the configuration of A and passes it to the QO module, which, in turn, sets the configuration of A and then sets an agent to run learning episodes for the first θ configuration. Then A proceeds to successively apply actions which are defined by its current policy π on E, sensing its next state and receiving a numeric reward. This process is repeated until a goal state or some stopping criterion is reached, and then the agent starts a new episode. After finishing a fixed amount of episodes, the results obtained by each episode of the agent are averaged in order to to calculate $f_A(\theta)$. Then, the average of the episode with the other outputs from different θ vectors is compared in the QO module and, if the minimum number of queries for the same θ has been reached, a bandit algorithm is used to define if the current vector is good enough to query it again in order to see if whether it was a result of the stochastic effect. If that is assumed to be the case, then the same θ is used to query f again. Otherwise, if the maximum amount of queries of the same vector has been reached, the current query is finished in the QO and $f_A(\theta)$ is passed to the BO so the configuration vector θ, and the pair $(\theta, f_A(\theta))$ are added into the dataset used to fit the Gaussian process (statistical model). Once incorporated into the statistical model, the BO starts using this new information to calculate μ_{n+1} and σ_{n+1}^2, in order to optimize the acquisition function so as to determine the next best configuration to perform in a new learning meta-episode. The optimizer then proceeds to initialize the agent with the new hyper-parameter configuration, repeating the cycle until a preset number of learning meta-episodes with their corresponding agent (lower-level) episodes have been made. The overall algorithm for the autonomous RLOpt Framework is depicted in Algorithm 2.

In Algorithm 2, Θ is the framework configuration vector. This vector is composed by A, which is the RL algorithm; μ_0 which is the prior mean vector; σ_f^2, the noiseless variance of f; σ_n^2, the noise level of the GP; l the vector that defines the magnitude of the GP covariance function; α, the acquisition function; α_{opt} which is the function that optimizes α; the cut-off time k; $minRuns_\theta$ and $maxRuns_\theta$ which are the minimum and maximum amount of learning episodes to be made under the same configuration θ, respectively; $init_{LH}$, the amount of optional training meta-episodes used to add information sampled by a Latin hypercube to the covariance matrix of the GP, and finally $episodes_{BO}$, which is the number of the BO meta-episodes

---

Algorithm 2: RLOpt framework.

Input : Θ

1 for $n = 1$ *to* $episodes_{BO}$ **do**

2 $\theta \leftarrow argmax_\theta(\alpha, \alpha_{opt})$

3 $episodeQueries = 0$

4 $nextQuery = True$

5   **while** $nextQuery$ **do**

6 $episodeQueries = episodeQueries + 1$

7 $init(A, episodes_A)$

8 $f_{A-avg}(\theta) \leftarrow 0$

9   **for** $ep = 1$ *to* $episodes_A$ **do**

10 $restart(A)$

11 $run(A \mid \theta)$

12 $saveExecution(A)$

13 $f_{A-avg}(\theta) \leftarrow f_{A-avg}(\theta) + f_{A-ep}(\theta)$

14   **end**

15 $nextQuery = decideIfNextQuery(A)$

16   **end**

17 $f_{A-avg}(\theta) = f_{A-avg}(\theta)/(episodeQueries * episodes_A)$

18 $add_{GP}(\theta, f_{A-avg}(\theta))$

19 end

Output: $\arg \max_\theta f_{A-avg}(\theta)$

---

to run. On the other hand, the random function $f_A(\theta)$ is sampled after executing a fixed amount of agent episodes given by $episodes_A$ and under a given configuration vector θ. After initializing the optimizer, the RL algorithm of A that runs the simulation and the acquisition and covariance functions are instantiated. If the BO has not prior query points, before the beginning of the first episode the covariance matrix is initialized by sampling a random number of $init_{LH}$ points in order to start the first set of GP model predictions with non-trivial values of mean and variance (this step is skipped if $init_{LH} = 0$). Whenever the BO changes the hyper-parameters of A and resets its knowledge, it also resets E by returning it to its initial state, so as to each agent can interact with the environment in similar conditions.

Because A involves a stochastic nature in its decision making policy to make room for exploration, the result of the performance metric $f_A(\theta)$ may vary from one simulation run to another even with the same configuration θ. In order to minimize its variance and obtain a more significant value of $f(\theta)$ for a given θ, $f(\theta)$ can be queried several times with the same θ and the result of the queries of the objective function is averaged over all queries made with the same vector. The core method where the QO module decision is performed is the $decideIfNextQuery$ function, that returns $True$ if it has been decided that the best choice is to query the objective function with the same configuration, and $False$ otherwise. In order to make this decision, a N-armed bandit algorithm is employed such that, at any time, there are two arms that can be chosen: the first is an arm that represents the decision to not keep querying the objective function with the same configuration, because the sampled mean and the variance of the $f(\theta)$ samples with the current θ vector are not good enough, in comparison to results obtained using other configuration vectors. Likewise, the other arm represents the decision of performing another query to the objective function with the same configuration, because the mean and variance of $f(\theta)$ under the same θ are sufficiently good compared to the other values of the performance metric.

On the other hand, the method $init(A, episodes_A)$ sets the agent to its initial environmental state, thus erasing its previous knowledge. This is done for two reasons: firstly, because previous knowledge relied on hyper-parameters which are different to the current set, and therefore the new knowledge is biased in unpredictable ways; secondly, in order to give the agent a new unbiased configuration to interact with the environment so as to evaluate how the learning curve converges when starting with no prior knowledge. On the other hand, the method $restart(A)$, used to start a new agent episode, restores the agent to the initial state of the environment, keeping its acquired knowledge. The procedure $run(A \mid \theta)$ runs the agent episodes under θ in its environment from its initial state and until A reaches a final state or the amount of steps corresponding to the cut-off time. The method $saveExecution(A)$ stores the results obtained in the episode, $f_{A-ep}(\theta)$, for further use, e.g. by saving the amount of steps required for the agent to reach the final state. Finally, when the last episode of A under the same configuration θ finishes, the BO saves and averages the results obtained, attaching them to the dataset D, incorporating this new observation in the statistical model. The cycle repeats itself until a certain amount of BO meta-episodes are completed, and

then the configuration θ that maximizes f_A is returned.

In order to compare the efficiency between an agent A against another agent A' in how both performs when taking actions to solve a task, two performance metrics were integrated into the framework. The first is the average amount of time steps per episode,

$$f_A(\theta) = \frac{\sum_i^{n_{ep}} t_i}{n_{ep}} \tag{14}$$

Where t_i is the amount of time steps employed in the episode i, and n_{ep} is the total number of episodes experienced by A under configuration θ. This metric aims to reward the agent that solves the task in the minimum possible time, i.e. taking the minimum possible amount of time steps to reach the goal state, where $n_{ep} \leqslant f(\theta) \leqslant k n_{ep}$. That is, if A cannot reach its objective before the cut-off time, the amount of steps employed in that episode will be added. When using this measure, the framework will treat the optimization as a minimization problem by searching for $\arg\min_\theta f_A(\theta)$.

On the other hand, the second metric used is the amount of *successful runs* per episode. The following metric adds a notion of success to the agent episodes by making a distinction between successful and unsuccessful episodes of the agent, instead of measuring the performance by a numerical score. For an episode to be considered as a success, it considers an objective that must be accomplished regardless of any other considerations such as the amount of steps the agent required to reach the goal state. The metric is defined by

$$f_A(\theta) = \frac{\sum_i^{n_{ep}} s_i}{n_{ep}} \tag{15}$$

Where, for each episode i, $s_i = 1$ if that episode was a success; $s_i = 0$ otherwise. The idea behind this performance metric is to adapt to problems where, being the unknown objective function is expensive to query, such as the function $f_A(\theta)$ defined above. Thus, it is more important for the learning agent to succeed while experiencing the episode rather than maximizing some other performance metric. For example, consider an agent which uses reinforcement learning in order to learn how to better perform rescheduling tasks. For such an agent, it is mandatory to find a repaired schedule which is feasible in minimum time rather than finding the best schedule that is both feasible and optimum in some sense, e.g. having a minimum total tardiness[46].

4 Case Study

4.1 Experimental Setup

In order to illustrate the proposed approach, results obtained for a gridworld problem are presented. Based on the Blocking Maze example in [7], the proposed example consists of a grid where an agent starts from an initial state S aiming to reach a final state G, in which the environment returns a reward of 1 (as opposed to transitions to any other state where the reward obtained by the learning agent is always 0). There is a set of obstacles between them, as depicted in Figure 2. Such obstacles are either permanent obstacles where the agent cannot pass, obstacles that can be temporarily overcome for the agent, but at a certain episode they become permanent obstacles, and temporal obstacles where the agent will be able to pass after some amount of episodes have elapsed. There are two instances where the environment may change in each agent simulation: the first happens whenever an agent instance reaches episode 15 and the other when the same agent reaches the episode 30. The idea behind those environmental changes is to test how the agent adapts to a somehow different environment with a given configuration of parameters.

In this work, two variants of the framework were employed: the first one which uses as performance metric the average amount of time steps per episode, and the second one which uses as metric the amount of successful steps per episode, which, in this example, is the amount of episodes where the agent reached the final state (i.e. in less time-steps than the cutoff time). The algorithm selected for the agents to perform the different experiments was SARSA(λ), which is a variant of the on-policy SARSA algorithm that uses eligibility traces, a mechanism to propagate the $Q(s,a)$ values to the state-action pairs visited in the past, proportionally to how much time steps have elapsed since the last time the agent visited each pair. The algorithm, in turn, is run under an ϵ-greedy policy and with a cut-off time of 400 time-steps. In this particular setting, each agent instance A_i has as the configuration vector $\theta_i = (\alpha_i, \epsilon_i, \gamma_i, \lambda_i)$, where the parameters represents the learning rate, the exploration rate, the discount factor and the eligibility traces decay rate, respectively. Regarding the statistical model configuration, the GP hyper-parameters were not optimized after each step in order to use the same configuration between all the simulations. Instead, the following hyper-parameters, which were determined empirically, were used: $\sigma_f^2 = 0.8$, $\sigma_n^2 = 0.17$, $\mu_0 = (0,0,0,0)$,
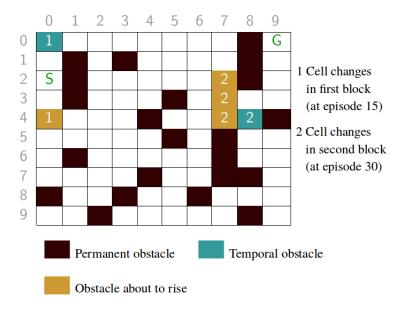
Figure 2: Double blocking gridworld environment.

$l = (-0.12, -0.12, -0.12, -0.12)$, expected improvement as the acquisition function and squared exponential as the covariance function. The hardware configuration used to run the different experiments consisted in a computer with 8 GB of RAM and 4 x 3.20 GHz processors.

Three different set of experiments were run in order to validate RLOpt. In the first experiment set, the focus is to validate how the framework iteratively seeks for a θ vector that optimizes a performance metric f in 30 meta-episodes considering two alternative ways for measuring it. On the other hand, the second experiment set uses the best two vectors of θ for each metric of f found by the optimizers performing a simulation, showing how their corresponding learning curves converge in comparison with a default θ configuration. Finally, in the third experiment set, an extension of the first experiment is presented, that illustrates how adding a N-armed bandit algorithm in the meta-learning layer makes room for a considerable reduction in the amount of queries with the same theta and, at the same time, maintains a similar or better convergence for both performance metrics. The experiments and their corresponding results and analysis are discussed in the subsections below.

4.2 Convergence of the optimizers towards the optimum of the objective function

In this set of experiments, the two variants of RLOpt were executed 10 times in order to compare how fast they converge towards the maximum value of f, where each execution consisted of 30 different queries to the performance function. For each query, a RL agent with a given configuration θ is engaged in running 5 times 50 episodes, in order to obtain the average of the query. Each execution of the success metric variant took an average of 2:12 hours, whereas the execution of the variant with the step-per-episode metric took an average execution time of 2:39 hours. The two variants of the framework were run in parallel, where the ten executions were run in batch, having finished in about 26:30 hours. The results of the optimization for the number of successes and the number of steps per episode are depicted in Fig. 3 and Fig. 4, respectively, where each curve represents how, for the success metric and steps per episode metric, the maximum and minimum that were found for each execution changes for a given meta-episode, and the thicker curve represents the average optimum value found for all runs made using the same framework configuration alternative.

Regarding the variant with the success metric, as can be seen in Fig. 3, the autonomous learning agent is able to found iteratively the average maximum percentage of success per episode starting from an average of near 0.3772 and converging to a maximum of 0.61 after 30 queries, which represents a 61.7% increase in the success rate from the initial to the final query, with about a 2.12% average increase of the maximum for each query. The learning curves reached a maximum of 0.7 successful episodes per episode and the worst case was 0.536 successful runs per episode. On the other hand, regarding the convergence, it can be appreciated in various learning curves and in the average curve that this variant is prone to be caught in a local maximum: there were a total of 12 meta-episodes where not a single of the 10 runs increased its maximum value at all. Another noticeable aspect is that the major increase of the maximum happened between the 2th and the 7th query on all the runs made, where the average increase of the maximum of all the queries made in that interval is 6.88%; and after that interval the average increase is reduced to 0.41%.
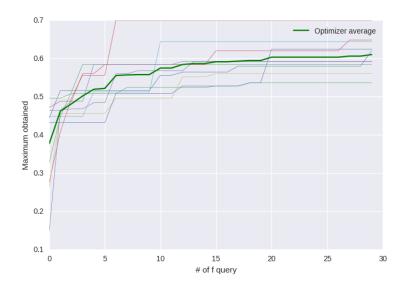
Figure 3: Convergence towards the maximum number of successes per episode.
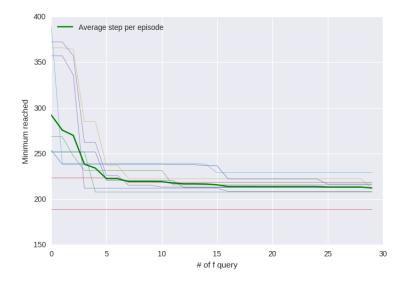

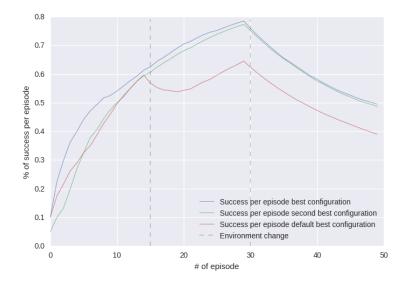Figure 4: Convergence towards the minimum number of steps per episode.

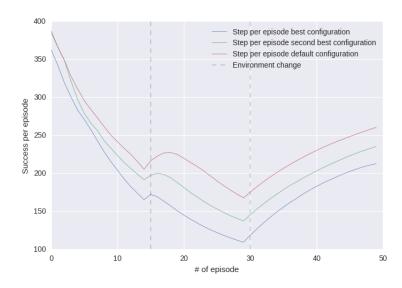Figure 5: Learning curve of the variant with the success metric.

Figure 6: Learning curve of the variant with the step-per-episode metric.

On the other hand, regarding the variant using the number-of-steps-per-episode metric displayed in Fig 4, the minimum of the average amount of steps per episode starts in 276.4 in the first query, converging after 30 queries to a minimum of 211,7, which is a decrease of about 23% of the initial average steps per episode, with an average of 0.80% decrease of the minimum of each run; the best minimum value for a single run reached 188.95 average-steps-per-episode while the worst minimum value is 229.06 average-steps-per-episode. Regarding the convergence of each execution, it can also be observed that in the different curves the optimizer is prone to stuck in a local minimum. The number of episodes where in no one of the 10 executions is able to change its minimum value using this metric which is 14. As in the other variant, a remarkable aspect is that, between the 2th and the 6th query of all the runs, a worth noting decrease is produced in the minimum of the majority of the curves, where various of the minima can be seen. The average decrease of the minimum found in this period is 3.6%. After that gap, the convergence slows down to an average decrease of 0.19% per episode. In both performance metrics used this fact can be interpreted as that the first few queries are the most important to determine the optimum of each execution, thus suggesting that a minor amount of meta-episodes could have been used instead of 30 to save computational time while obtaining similar results.

4.3   Comparison of Learned Policies with a Default Configuration

In order to visualize how is the learning of the different agents that use the global optimum obtained from each variant, and how they compare to a default configuration used in RL, the best two configurations for each alternative performance metric had been run again 20 times in a RL agent and averaged in order to

Table 1: Hyper-parameters and results for the success metric variant

	Hyper-parameters	Metric
Best Configuration	$(0.538, 0.49, 0.69, 0.686)$	0.495
Second Best Configuration	$(0.582, 0.553, 0.653, 0.321)$	0.487
Soar Default Configuration	$(0.3, 0.1, 0.9, 0.001)$	0.39

Table 2: Hyper-parameters and result for the steps-per-episode variant

	Hyper-parameters	Metric
Best Configuration	$(0.607, 0.191, 0.667, 0.707)$	212.49
Second Best Configuration	$(0.291, 0.38, 0.5, 0.784)$	235.07
Soar Default Configuration	$(0.3, 0.1, 0.9, 0.001)$	260.29

compare them with a default configuration for RL agents. For such default configuration, the parameters selected as default were the default parameters employed for a SARSA(λ) agent in the Soar Cognitive Architecture [47] (specified in the Soar manual), that consisted on ($\alpha = 0.3, \epsilon = 0.1, \gamma = 0.9, \lambda = 0.001$). The results can be seen in Table 1 and Table 2, whereas the convergence, represented by a learning curve that averages all the previous episodes, can be appreciated in Fig. 5 and Fig. 6 for the variant with the success metric and for the variant with the step-per-episode metric, respectively.

When the performance metric is the success, it can be appreciated that both agents with its configurations obtained as a result of the optimization perform similarly. Moreover, both behave better than the default configuration, which in turn start converging similarly to the second best configuration but their adaptation is significantly worse compared to the first environmental change. The best configuration starts consistently better than the other two, whereas their average is significantly decreased after the second environmental change, in a similar profile in comparison to the other two runs. On the other hand, regarding the steps-per-episode variant, the agent with the default configuration again performed worse compared to the agent with the second best configuration, despite it starts with a poor performance in comparison to the agent with the default configuration. Compared with the other variants, there was a clearer distinction in this metric between the performance of the different agents, where at least 20 steps per episode separated all three runs and this separation increases after the first environmental change.

4.4 Comparison with random search

In order to assess how the results obtained in the precedent section compares with other hyper-parameter optimization methods, convergence curves obtained used the random search (RS) method help providing a representative example (as described in Section 2.1). An implementation of the RS algorithm was run with the same budget and configuration used in RLOpt: 10 executions of the algorithm, where each execution consists of 30 meta-episodes. The ten executions were set to run twice, one for each performance metric.

The results of the convergence can be seen in Fig. 7 and Fig. 8 for the success and the steps- per-episode metric, respectively. The blue and red curves represent the observed behavior for RLOpt and Random Search, respectively. In each learning curve, the solid line in the middle corresponds to the mean of ten independent executions, whereas the lighter area around that line has drawn based on the standard deviation and confidence intervals. As can be seen in both curves, RLOpt consistently tends to find better optima conbinations of hyper-parameters, on average, than RS. More specifically, for the success metric, RLOpt consistently increases the distance with respect to RS. on the other hand, for the step-per-episode metric, RLOpt does a better start than RS in the first 5 meta-episodes. Later on, RS starts to reduce its distance to RLOpt from the 11th meta-episode and onwards, reaching the minimum distance at the 28th meta-episode.

4.5 Variants with N-Armed Bandits Decision Algorithms

In order to reduce the total amount of times $f(\theta)$ is queried, and therefore execution times, the experiments presented in this section are based on the experiments made previously in Section 4.2, extending them by adding executions with N-armed bandits for each variant of the optimizer. Five variants of the bandit-based algorithms presented in Section 2.6 were used: ϵ-greedy with $\epsilon = 0.2$, greedy (i.e. ϵ-greedy with $\epsilon = 0$), Softmax with $\tau = 1$, UCB1 and UCB1Tuned. On each execution of the optimizers, a setting of
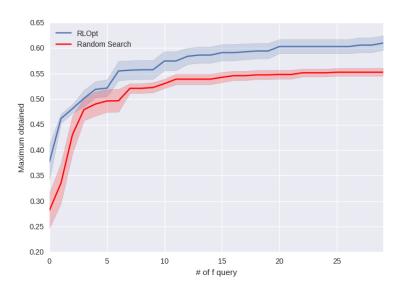
Figure 7: Comparison of RLOpt convergence to the maximum against the convergence of random search (success metric).
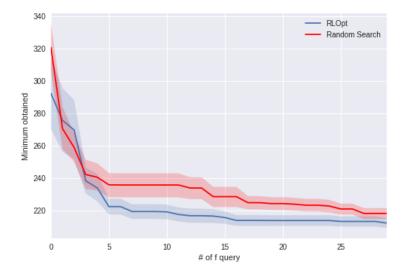

Figure 8: Comparison of RLOpt convergence to the minimum against the convergence of random search (steps-per-episode metric).
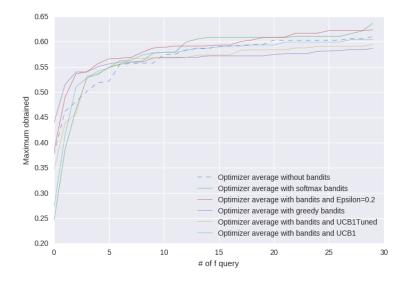
Figure 9: Comparison with the different average convergence curves for the success metric optimizers with different bandit algorithms.

Table 3: Execution times and average number of queries of different optimizer variants (success metric).

	No bandits	Softmax	ϵ-greedy	Greedy	UCB1	UCB1Tuned
Avg. number of queries	150.0	83.6	97.1	115.9	126.4	125.4
Avg. time per optimizer execution	2:12:25	1:19:47	1:27:33	1:44:48	1:54:37	1:52:53
% of time reduction	-	39.7%	33.8%	20.8%	13.4%	14.7%

$minRuns_\theta = 2$ and $maxRuns_\theta = 5$ were used, so each optimizer run would at least query f twice for each θ, and at most 5 times in order to avoid impasses whenever a local optimum is found and thus saving computational resources.

The results of this experiment on the success metric variant optimizer are summarized in Table 3 for the total amount of queries on $f(\theta)$ and execution times, and in Fig 9 learning curves are shown. The average of the convergence of each variant with its corresponding bandit algorithm is presented in continuous curves, while the average convergence without bandits (as was mentioned in Section 4.2) is displayed as a non-continuous curve. It can be appreciated, in terms of the number of queries made, how the bandit-based approach achieves a significant saving, between 15% and 40%, depending on the algorithm used. More specifically, it is worth noting that the ϵ-greedy bandit and, specially, the Softmax bandit gives rise to the biggest reduction of all in the average amount of queries and elapsed time, followed by the greedy algorithm. On the other hand, the UCB1 and UCB1Tuned algorithms correspond to the lowest performance. By observing the learning curves, it can be seen how, in terms of convergence to the maximum, the variant with ϵ-greedy bandit algorithm converges slightly better than the performance average with no bandits by an amount of 3,1%, whereas Softmax starts with a poor convergence and then recovered to an average of 0.16% increase. Finally, the greedy bandit exhibits a similar convergence (-0.1%) and, regarding the UCB1 and UCB1Tuned algorithms, they slightly underperformed in comparison to the average performance without bandits (-0.48% and -1.46%, respectively).

Finally, for the step-per-episode measured optimizers, the results are shown in Table 4 and in Fig. 10. Regarding the number of queries made, it is shown that the ϵ-greedy, Softmax and greedy algorithms were those that reduced the most the number of queries, and the UCB and UCB1Tuned obtained the least reduction, similarly to the success metric variant. The main differences in this experiment compared with the previous one is that the algorithm that achieves the biggest reduction was the ϵ-greedy rather than Softmax. Moreover, this time the greedy algorithm performed consistently better. In terms of execution time reduction, ϵ-greedy obtained the highest reduction with near 46%, followed by a 40% reduction of the greedy algorithm (which almost doubles the reduction obtained in the previous experiment). A slightly less significant reduction (39.4%) is obtained using Softmax, which maintains almost the same reduction achieved in the previous experiment. In terms of convergence, all the bandit algorithms tried follow a better convergence, on average, towards the minimum, reporting less amount of steps-per-episode in comparison with the variant without bandits. Here, the ϵ-greedy algorithm achieves best convergence, which found a
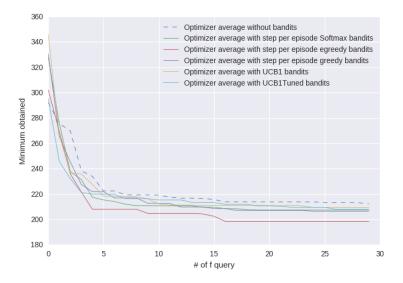
Figure 10: Comparison with the different average convergence curves for the step-per-episode measured optimizers with different bandit algorithms.

Table 4: Execution times and average number of queries of different optimizer variants (step-per-episode metric).

	No bandits	Softmax	ϵ-greedy	Greedy	UCB1	UCB1Tuned
Avg. number of queries	150.0	89.4	83.8	93.3	127.3	127.9
Avg. time per optimizer execution	2:39:23	1:36:28	1:26:36	1:34:41	2:15:29	2:14:09
% of time reduction	-	39.4%	45.6%	40.5%	14.9%	15.8%

minimum that is -6,4% on average, compared with the optimizer variant that does not use bandits, and followed by Softmax, greedy, UCB1Tuned and UCB1 (-2.83%, -2.63%, -2.51% and -1.48%, respectively).

Analyzing both experiments with and without bandits-based algorithms, it can be said that the main benefit of adding a bandit algorithm in the meta-learning layer is that, depending of the chosen algorithm, there is a considerable reduction regarding the number of queries to the performance function f. As a result,the execution time that every optimizer requires to complete their corresponding number of meta-episodes is also reduced. Regarding the convergence to the optimum, it is seen that some algorithms achieve, on average, a slight better performance than the variant with no bandits, while other algorithms obtained virtually the same convergence rate or even under-performed slightly. A problem to be addressed in a future work is the reliance of bandit algorithms on their own hyper-parameters; this is clearly seen in the greedy algorithm, which is the ϵ-greedy algorithm with $\epsilon = 0$, which performed consistently worse than the $\epsilon = 0.2$ algorithm in both experiments. This is a plausibe problem that UCB1 and UCB1Tuned also had in their respective experiments. Even though they do not have hyper-parameters, the numeric scales can be considerably different than those of the problem as it happens in the experiments made. Thus, having a limited amount of queries, adding a hyper-parameter to change the scales can be an addition to be considered. Another aspect to address is that, even with the inclusion of bandit algorithms, the stagnation in the convergence it is repeated as in the variants without bandits, as it can be seen in the convergence curves. It can be said that the convergence to the maximum starts rapidly in the meta-episodes 2-5 (5.7% increase and 4.77% decrease per meta-episode for the different success and step-per-episode variants, in average), slowing down to an average of about 0.45% increase and 0.24% decrease from there, respectively. As in the first set of experiments, this means that it would be almost unnoticeable in terms of convergence of the learning if the optimizer were run for half or even less meta-episodes.

5   Concluding Remarks

A framework for automating the quest for a near-optimal policy in an autonomous reinforcement learning task has been presented. The proposed framework allows the learning agent to automatically find a good policy abstracting the user from having to manually tune the configuration, resorting to an expensive trial-

and-error approach or using a possibly sub-optimal, default hyper-parameter configuration not suited for the specifics of the learning task at hand. By using Bayesian optimization, the framework takes into account all the previous points and observed values. By assuming a Gaussian prior of the data at the meta-level of learning, the framework uses a Gaussian process regression model to estimate the point where the maximum cumulative reward has a greater probability of being located, thus increasing the efficiency of the search of the hyper-parameter configuration that maximizes the efficiency of a reinforcement learning agent. Finally, by incorporating a bandit learning algorithm in order to decide which regions are potentially more appealing to explore further in order to reduce the stochasticity of the samples, the framework can reduce the amount of computational time required while maintaining a good convergence towards the optimum.

For future research work, there are several directions were this work is to be extended. The first direction points towards the extension of the performance functions and the respective analysis and comparison with the current ones in different environments and including metrics with labels that belongs to a certain category $C_i \in \{C_1, C_2, ..., C_n\}$ e.g. whether an execution is a failure or a success, instead of being real valued. The second direction is about the extension of the case study to an industrial example for a rescheduling task, in order to analyze how the framework performs with the increase of complexity in the environment, adapting the framework to such complexity by making modifications such as the use of more complex reinforcement learning algorithms or erasing queries that have low information contents after the covariance matrix reaches a certain rank. Also, another research avenue is the extension of the framework to a higher level for hyper-parameter selection such as RL algorithm selection and agent policy selection, among others. Additionally, other research direction is to add in the framework methods to detect which regions of hyper-parameters can ensure an adequate performance in the agent's learned policy, and then perform an optimization inside those smaller regions. The fifth direction address the incorporation in RLOpt of the optimization of the Gaussian process own hyper-parameters, by maximizing Eq. (9) instead of using empirically determined values. Finally, the sixth direction approaches the implementation of other regression models such as random forests, and the comparison of GP with different covariance functions.

6 Acknowledgement

References

[1] F. Hutter, J. Lücke, and L. Schmidt-Thieme, "Beyond Manual Tuning of Hyperparameters," *KI - Künstliche Intelligenz*, vol. 29, no. 4, pp. 329–337, Nov. 2015. [Online]. Available: https://link.springer.com/article/10.1007/s13218-015-0381-0

[2] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for Hyper-Parameter Optimization," in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2011, pp. 2546–2554. [Online]. Available: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

[3] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An Automatic Algorithm Configuration Framework," *Journal of Artificial Intelligence Research (JAIR)*, vol. 36, pp. 267–306, 2009.

[4] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," *arXiv:1206.2944 [cs, stat]*, Jun. 2012, arXiv: 1206.2944. [Online]. Available: http://arxiv.org/abs/1206.2944

[5] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *arXiv:1206.5533 [cs]*, Jun. 2012, arXiv: 1206.5533. [Online]. Available: http://arxiv.org/abs/1206.5533

[6] J. Bergstra and Y. Bengio, "Random Search for Hyper-parameter Optimization," *Journal of Machine Learning Research*, vol. 13, no. 1, pp. 281–305, Feb. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2503308.2188395

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, ser. Adaptive computation and machine learning. Cambridge, Mass: MIT Press, 1998.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://www.nature.com/nature/journal/v518/n7540/abs/nature14236.html

[9] E. Brochu, V. M. Cora, and N. de Freitas, "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning," *arXiv:1012.2599 [cs]*, Dec. 2010, arXiv: 1012.2599. [Online]. Available: http://arxiv.org/abs/1012.2599

[10] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, May 1992. [Online]. Available: http://link.springer.com/article/10.1007/BF00992698

[11] G. A. Rummery and M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," Cambridge University Engineering Department, CUED/F-INFENG/TR 166, 1994.

[12] R. S. Sutton, "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," in *In Proceedings of the Seventh International Conference on Machine Learning.* Morgan Kaufmann, 1990, pp. 216–224.

[13] R. Sutton, "Planning by Incremental Dynamic Programming," in *In Proceedings of the Eighth International Workshop on Machine Learning.* Morgan Kaufmann, 1991, pp. 353–357.

[14] J. C. Barsce, J. A. Palombarini, and E. C. Martínez, "Towards autonomous reinforcement learning: Automatic setting of hyper-parameters using Bayesian optimization," in *2017 XLIII Latin American Computer Conference (CLEI)*, Sep. 2017, pp. 1–9. [Online]. Available: http://ieeexplore.ieee.org/document/8226439/

[15] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Gradient-based Hyperparameter Optimization through Reversible Learning," *arXiv:1502.03492 [cs, stat]*, Feb. 2015, arXiv: 1502.03492. [Online]. Available: http://arxiv.org/abs/1502.03492

[16] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential Model-Based Optimization for General Algorithm Configuration," in *Learning and Intelligent Optimization*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Jan. 2011, pp. 507–523. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-25566-3_40

[17] B. Shahriari, K. Swersky, Z. Wang, R. Adams, and N. de Freitas, "Taking the Human Out of the Loop: A Review of Bayesian Optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

[18] R. Sutton and S. P. Singh, "On Step-Size and Bias in Temporal-Difference Learning," in *Center for Systems Science, Yale University*, 1994, pp. 91–96.

[19] W. Dabney and A. G Barto, "Adaptive Step-Size for Online Temporal Difference Learning," *Proceedings of the National Conference on Artificial Intelligence*, vol. 2, Jan. 2012.

[20] A. R. Mahmood, R. Sutton, T. Degris, and P. Pilarski, "Tuning-free step-size adaptation," in *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*, Mar. 2012.

[21] N. Yoshida, E. Uchibe, and K. Doya, "Reinforcement learning with state-dependent discount factor," Aug. 2013, pp. 1–6.

[22] M. White and A. White, "A Greedy Approach to Adapting the Trace Parameter for Temporal Difference Learning," *arXiv:1607.00446 [cs, stat]*, Jul. 2016, arXiv: 1607.00446. [Online]. Available: http://arxiv.org/abs/1607.00446

[23] N. Schweighofer and K. Doya, "Meta-learning in Reinforcement Learning," *Neural Networks*, vol. 16, no. 1, pp. 5–9, Jan. 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608002002289

[24] R. Laroche and R. Feraud, "Reinforcement Learning Algorithm Selection," *arXiv:1701.08810 [cs, math, stat]*, Jan. 2017, arXiv: 1701.08810. [Online]. Available: http://arxiv.org/abs/1701.08810

[25] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving," *arXiv:1610.03295 [cs, stat]*, Oct. 2016, arXiv: 1610.03295. [Online]. Available: http://arxiv.org/abs/1610.03295

[26] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016. [Online]. Available: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html

[27] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv:1712.01815 [cs]*, Dec. 2017, arXiv: 1712.01815. [Online]. Available: http://arxiv.org/abs/1712.01815

[28] J. Peng and R. J. Williams, "Incremental Multi-Step Q-Learning," in *Machine Learning.* Morgan Kaufmann, 1996, pp. 226–232.

[29] J. Močkus, "On Bayesian Methods for Seeking the Extremum," in *Optimization Techniques IFIP Technical Conference*, ser. Lecture Notes in Computer Science, P. D. G. I. Marchuk, Ed. Springer Berlin Heidelberg, 1975, pp. 400–404. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-662-38527-2_55

[30] M. D. McKay, R. J. Beckman, and W. J. Conover, "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code," *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979. [Online]. Available: http://www.jstor.org/stable/1268522

[31] J. Močkus, V. Tiesis, and A. Zilinskas, "The application of Bayesian methods for seeking the extremum," in *Towards Global Optimisation 2*, L. Dixon and G. Szego, Eds. North-Holand, 1978, pp. 117–129.

[32] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient Global Optimization of Expensive Black-Box Functions," *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, Dec. 1998. [Online]. Available: http://link.springer.com/article/10.1023/A%3A1008306431147

[33] C. E. Rasmussen and C. K. I. Williams, *Gaussian processes for machine learning*, 3rd ed., ser. Adaptive computation and machine learning. Cambridge, Mass.: MIT Press, 2008.

[34] J. Schmidhuber, J. Zhao, and M. Wiering, "Simple Principles Of Metalearning," SEE, Tech. Rep., 1996.

[35] S. Thrun and L. Pratt, "Learning to Learn: Introduction and Overview," in *Learning to Learn.* Springer, Boston, MA, 1998, pp. 3–17. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4615-5529-2_1

[36] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn," *arXiv:1611.05763 [cs, stat]*, Nov. 2016, arXiv: 1611.05763. [Online]. Available: http://arxiv.org/abs/1611.05763

[37] C. Lemke, M. Budka, and B. Gabrys, "Metalearning: a survey of trends and technologies," *Artificial Intelligence Review*, vol. 44, no. 1, pp. 117–130, Jun. 2015. [Online]. Available: https://link.springer.com/article/10.1007/s10462-013-9406-y

[38] Y. Freund and R. E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, Aug. 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002200009791504X

[39] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, "RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning," *arXiv:1611.02779 [cs, stat]*, Nov. 2016, arXiv: 1611.02779. [Online]. Available: http://arxiv.org/abs/1611.02779

[40] C. Finn, P. Abbeel, and S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks," *arXiv:1703.03400 [cs]*, Mar. 2017, arXiv: 1703.03400. [Online]. Available: http://arxiv.org/abs/1703.03400

[41] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret, "Robots that can adapt like animals," *Nature*, vol. 521, no. 7553, pp. 503–507, May 2015, arXiv: 1407.3501. [Online]. Available: http://arxiv.org/abs/1407.3501

[42] R. Pautrat, K. Chatzilygeroudis, and J.-B. Mouret, "Bayesian Optimization with Automatic Prior Selection for Data-Efficient Direct Policy Search," *arXiv:1709.06919 [cs]*, Sep. 2017, arXiv: 1709.06919. [Online]. Available: http://arxiv.org/abs/1709.06919

[43] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv:1402.6028 [cs]*, Feb. 2014, arXiv: 1402.6028. [Online]. Available: http://arxiv.org/abs/1402.6028

[44] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, vol. 47, no. 2-3, pp. 235–256, May 2002. [Online]. Available: https://link.springer.com/article/10.1023/A:1013689704352

[45] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, and M. S. Lew, "Deep learning for visual understanding: A review," *Neurocomputing*, vol. 187, no. Supplement C, pp. 27–48, Apr. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231215017634

[46] J. A. Palombarini, J. C. Barsce, and E. C. Martínez, "Generating Rescheduling Knowledge using Reinforcement Learning in a Cognitive Architecture," in *Anales del 15º Simposio Argentino de Inteligencia Artificial*, Universidad de Palermo, Buenos Aires, 2014.

[47] J. Laird, *The Soar cognitive architecture.* London, England: MIT Press, 2012.