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

Juan Cruz Barsce
Depto. de Ing. en Sistemas de Información
Facultad Regional Villa Maria
Universidad Tecnológica Nacional
Email: jbarsce@frvm.utn.edu.ar

Jorge A. Palombarini
CIT Villa María - CONICET-UNVM
GISIQ
Facultad Regional Villa María
Universidad Tecnológica Nacional
Email: jpalombarini@frvm.utn.edu.ar

Ernesto C. Martínez
Instituto de Desarrollo y Diseño
CONICET-UTN
Facultad Regional Santa Fe
Universidad Tecnológica Nacional
Email: 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 obtaining good performances regardless of user expertise in the inner workings of the techniques and methodologies. In particular, for a reinforcement learning task, the efficiency of an agent learning a policy in an uncertain environment has a strong dependency on how hyper-parameters in the algorithm are set. In this work, an autonomous framework that employs Bayesian optimization and Gaussian process regression to optimize the hyper-parameters of a reinforcement learning algorithm is proposed. A gridworld example is discussed in order to show how hyper-parameter configurations of a learning algorithm (SARSA) are iteratively improved based on two performance functions.**

***Index Terms*—autonomous reinforcement learning, hyper-parameter optimization, Bayesian optimization, Gaussian process regression**

I. 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 better 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 that influence on attributes such as model complexity or the learning rate of an algorithm, which are a key issues in order to extrapolate better to unseen situations. 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 [2]–[6], with methods such as random search, gradient search, Bayesian optimization, among others.

978-1-5386-3057-0/17/$31.00 ©2017 IEEE

In the particular case of reinforcement learning (RL) [7], as opposed to supervised learning, there are not examples of desired behavior in the form of datasets containing a label for each vector of features, but instead it is possible to collect score examples of behavior according to some performance criterion: given a state s of the agent's environment, the RL problem is about obtaining the optimal policy (a function that given sensed states define 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. On the other hand, the Bayesian optimization strategy provides an approach designed to maximize the effectiveness of an expensive function since it is derivative-free and less prone to be caught in local minimum [9].

In the particular case of RL algorithms, such as Q-Learning [10] or SARSA(λ) [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 a modification of this parameter may have in the agent's performance solving the task is normally unclear prior to its execution. Even more, when comparing the results of an agent run with the results of a run of a different agent, it will be unclear that a solution outperforms the other because it is fundamentally superior, or

simply because it has a better parameter setting. Moreover, if the environment is complex enough such that each agent learning episode is computationally expensive, having a bad parameter configuration could result in higher execution times that can eventually give rise to mediocre performance [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 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. 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.

II. Background

A. Reinforcement Learning

As a method that can be combined with a large set of algorithms such as deep learning models, as presented in [8], and employed in applications such as self-driving cars [14], reinforcement learning (RL) is a form of learning where an agent is immersed in an environment E and its objective is to converge to an optimal policy by performing sequences of actions in different environmental states and get a reinforcement signal after each action. At any 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 reward, 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 resulting 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 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 the action a. An episode is defined by the finite sequence $s_0, a_0, r_1, s_1, a_1, ..., a_{n-1}, r_n, s_n$. 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 that summed up the experience with past interactions with the environment and aims to maximize future rewards. To balance the aim 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* $Q_\pi(s)$, which represents the expected reward to obtain 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

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

where $P(s'|s, a)$ 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 $Q(s, a)$ updates 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

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

Other well-known reinforcement learning algorithms includes *on-policy* algorithms such as SARSA [11], *model-based* algorithms such as Dyna versions of Q and algorithms that incorporate *eligibility traces* such as Q(λ) [15], 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

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

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

B. Bayesian Optimization

As an approach used to maximize an expensive function such as the one measuring the performance of a reinforcement learning agent in an uncertain environment, Bayesian optimization [16] seeks to optimize an unknown objective function $f(X)$, where $X \in \mathbb{R}^n$, by treating it as a black box function and placing a prior over it. 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 query budget 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* using sampling methods such as Latin hypercube [17]. 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 new feasible regions of interest where the function maximum can be located. Emphasizing exploitation accelerates convergence, but the agent may often fall in a local maximum; on the other hand, emphasizing exploration ensures a more distributed sampling, but can significantly increase the time it takes to converge. An acquisition function commonly chosen is the *expected improvement function* [18], [19], because it has a good balance 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 the current 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 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 $Z = \frac{\mu(X) - f(X^+)}{\sigma(X)}$.

C. Gaussian Process

In a supervised learning task, the agent is given with examples in the form feature-response pairs $D_n = \{(X_i, f(X_i))\}$, $i = 1, ..., n$ which are used to learn an approximate relationship which generalizes to 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 performing a *regression*, or categorically valued, i.e. $f(X) \in \{C_1, C_2, ..., C_n\}$, where in such case the agent is performing 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

$$MSE = n^{-1} \sum_{i}^{n} (f(X_i) - \hat{f}(X_i))^2$$

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 with a prior mean function $\mu_0(X) \rightarrow \mathbb{R}$ and a covariance function $k(X_i, X_j) \rightarrow \mathbb{R}$. By taking that assumption and incorporating the pairs D_n obtained from querying the objective function n times, we are using a Gaussian process [20] as the surrogate model. A Gaussian process (GP) places a prior over the data set D_n, and, by applying the *kernel trick* [20], it constructs a Bayesian regression model as follows: given D_n, the mean and variance of a new point X is given by

$$\mu_{n+1}(X) = \mu_0(X) + k(X)^T K^{-1}(Y - \mu_0) \tag{5}$$

$$\sigma^2_{n+1}(X) = k(X, X) - k(X)^T K^{-1} k(X) \tag{6}$$

where K is the covariance matrix, where each element $K_{ij} = k(X_i, X_j)$ and $k(X) = (k(X_1, X), k(X_2, X), ..., k(X_n, X))$ 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. A common choice is the squared exponential function with automatic relevance determination [20], given by

$$\begin{aligned} k_{se}(X_i, X_j) &= \sigma_f^2 \exp\{-\frac{1}{2}(X_i - X_j)^T \\ &\quad diag(l)^{-2}(X_i - X_j)\} + \sigma_n^2 \delta_{ij} \end{aligned} \tag{7}$$

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. Other common function 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

Algorithm 1: Bayesian Optimization

Input : α_{model} - the acquisition function
for $t = 1, 2, ...$ **do**
  $X_t = argmax_X(\alpha_{model}(X|D))$
  Sample the objective function $y_t = f(X_t) + \epsilon$
  $D_{t+1} = add(D_n, (X_{t+1}, y_{t+1}))$
  Update the statistical model
end
Output: $argmax_X f(X)$

$$\log P(y|X,\theta_{GP}) = -\frac{1}{2}(y-\mu_0)^T K^{-1}(y-\mu_0) - \frac{1}{2}\log|K| - \frac{n}{2}\log 2\pi \quad (8)$$

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.

III. 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 objective of RLOpt is 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 $f_A(\theta) \rightarrow \mathbb{R}$ which measures the influence of hyper-parameters on the learning curve. The output of such function is the performance measure in learning to solving the task. In other words, the hyper-parameters of the RL algorithm are taken as predictors of a real valued response, 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 the best policy to interact with the environment. Therefore, the objective of the RLOpt framework is resorting to data gathered $D_n = \{(\theta_i, f_A(\theta_i))\}$ as a feedback to train a regression model and then use Bayesian optimization to learn good 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 five components, as it is depicted in Fig. 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. The second component is the Bayesian Optimizer 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, according to Θ, is set to run a certain amount of meta-episodes (i.e. queries

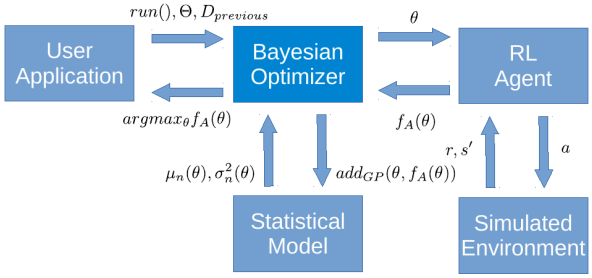

Fig. 1. The Autonomous RLOpt Framework.

to $f_A(\theta)$ which represents higher abstraction level episodes where several RL agent episodes are run under the same θ configuration). The third component is the reinforcement learning agent A, which receives configuration vectors from BO and interacts with the environment, learning a policy by sensing the environment state taking actions without having no prior knowledge about the optimal policy for each new vector of hyper-parameters. The fourth 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. Finally, the fifth component is the GP model 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 greater probability to be located, considering the previous learning meta-episodes. In the current implementation of the RLOpt, the statistical model employed is a Gaussian process; the main benefit of a GP 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 in methods such as deep learning models. 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 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. After that, the results obtained by the agent are averaged in order to to calculate $f_A(\theta)$, and the query $(\theta, f_A(\theta))$ is 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 optimize the acquisition function in order 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 episodes have been made. The overall algorithm for the autonomous RLOpt Framework is depicted in Algorithm 2.

Algorithm 2: RLOpt framework.

```
Input : Θ
for n = 1 to episodes_BO do
    θ ← argmax_θ(α, α_opt)
    for run = 1 to runs_θ do
        init(A, episodes_A)
        f_A−avg(θ) ← 0
        for ep = 1 to episodes_A do
            restart(A)
            run(A|θ)
            saveExecution(A)
            f_A−avg(θ) ← f_A−avg(θ) + (f_A−ep(θ) + σ²_n−ep)
        end
        f_A−avg(θ) ← f_A−avg(θ)/episodes_A
        add_GP(θ, f_A−avg(θ))
    end
end
Output: arg max_θ f_A−avg(θ)
```

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; k the cut-off time; $runs_\theta$ the amount of learning episodes to be made under the same configuration θ; $init_{LH}$, 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 to run. On the other hand, the random function $f_A(\theta)$ is obtained 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 specified. 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 $f_A(\theta)$ may vary from a simulation to another even with the same configuration θ. In order to minimize the impact of the stochastic effect, the result of the query of the objective function for a given configuration θ is averaged over all queries made according to $runs_\theta$, in order to obtain a sample of the expected return value of f_A.

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 $run(A|\theta)$ 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 of the episode, e.g. it saves the amount of steps the agent required 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 and incorporates 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 measures were integrated into the framework. The first is the average amount of time steps per episode,

$$f_A(\theta) = \frac{\sum_i^{n_{ep}} t_i}{n_{ep}} \tag{9}$$

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 measure 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 $n_{ep} \leqslant f(\theta) \leqslant kn_{ep}$. 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 measure used is the amount of *successful runs* per episode. The following measure 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 measure is defined by

$$f_A(\theta) = \frac{\sum_i^{n_{ep}} s_i}{n_{ep}} \tag{10}$$

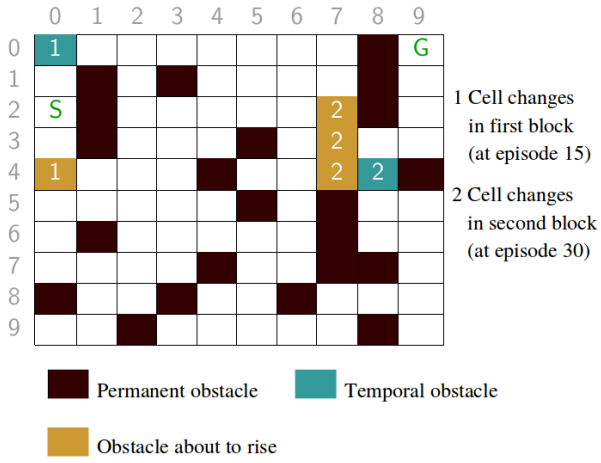

Fig. 2. Double blocking gridworld.

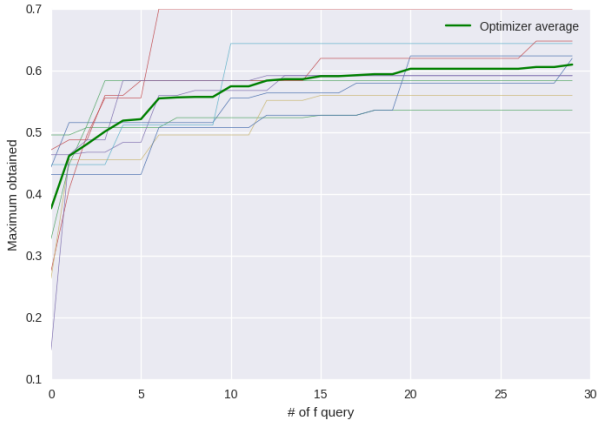

Fig. 3. Convergence towards the maximum number of successes per episode

Where, for each episode i, $s_i = 1$ if that episode was a success; $s_i = 0$ otherwise. The idea behind this performance measure is to adapt to problems where, being the unknown objective function expensive to query, such as the function $f_A(\theta)$ defined above, it is more important for the episode to succeed 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 [21].

IV. Case Study

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 are temporary accessible for the agent but in a certain episodes 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 example, two variants of the framework were employed: the first one which uses as performance measure the average amount of time steps per episode, and the second which uses as measure 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). Each one of these variants 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 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. The algorithm selected for the agents 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 $\epsilon - greedy$ 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 were used: $\sigma_f^2 = 0.8$, $\sigma_n^2 = 0.17$, $\mu_0 = (0, 0, 0, 0)$, $l = (-0.12, -0.12, -0.12, -0.12)$, expected improvement as the acquisition function and squared exponential as the covariance function. Each execution took an average of 2:25 hours each and the two variants of the framework were run in parallel, both having finished in about 26:30 hours under an environment with a hardware configuration that consisted of a computer with 8 GB of RAM and 4 x 3.20 GHz processors.

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 measure and steps per episode variants, 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 variant.

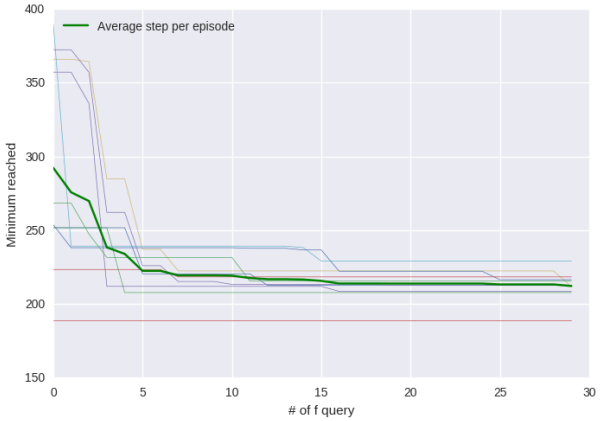

Fig. 4. Learning curve for the number of steps per episode

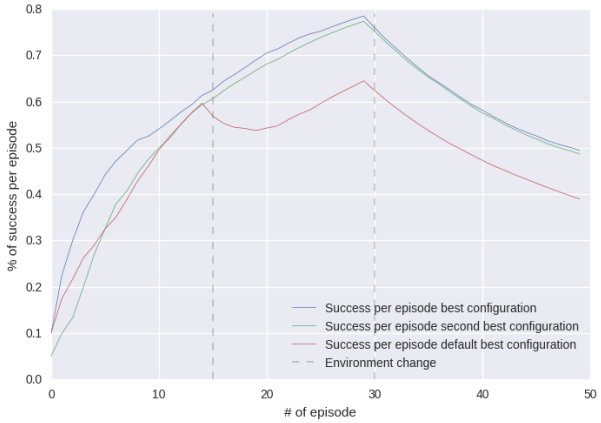

Fig. 5. Learning curve of the variant with the success measure

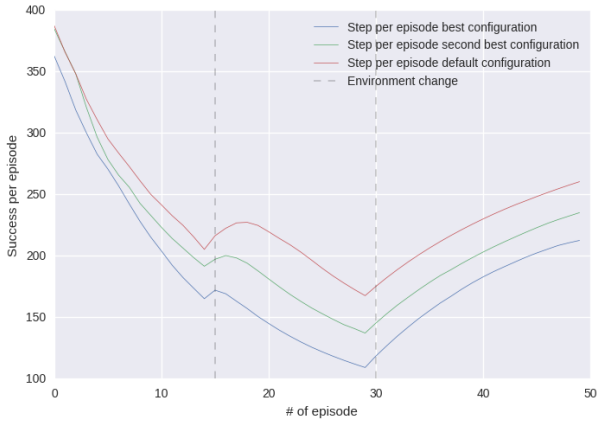

Fig. 6. Learning curve of the variant with the step per episode measure

Regarding the variant with the success measure, 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 of success 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 local maxima: there were a total of 12 queries where not a single of the 10 executions increased its maximum at all. Another notable 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 in that interval is 6.88%; and after that interval the average increase is reduced to 0.41%.

On the other hand, regarding the variant using the number of steps per episode measure 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 a single run reached was 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 not a single of the 10 executions changed its minimum in this variant is 14. As in the other variant, a remarkable aspect is that, between the 2th and the 6th query of all the runs, a notable decrease it is produced in the minimum of the majority of the curves, where various of the definitive minimums are obtained in between this queries. 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 variants this can be interpreted as that the first few queries are the most important to determine the optimum of each execution.

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 of each variant had been run again 20 times in a RL agent and averaged in order to 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 [22] (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 I and Table II, 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 measure and for the variant with the step per episode measure, 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 than the second best configuration

TABLE I
HYPER-PARAMETERS AND RESULTS FOR THE SUCCESS MEASURED VARIANT

	Hyper-parameters	Measure
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 II
HYPER-PARAMETERS AND RESULT FOR THE STEPS-PER-EPISODE VARIANT

	Hyper-parameters	Measure
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

but their adaptation is significantly worse compared to the first environmental change. The best configuration started 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 started 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 measure 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.

V. CONCLUDING REMARKS

A framework for approaching 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 Bayes' 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 to be located, increasing the efficiency of the search of the hyper-parameter configuration that maximizes the learning efficiency of a reinforcement learning agent.

For future 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 measures with labels that belongs to a certain category $C_i \in \{C_1, C_2, ..., C_n\}$ 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 erase of queries that has a low information content 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. Finally, the fourth direction approaches the implementation of other regression models such as random forests, and the comparison with GP with different covariance functions.

VI. ACKNOWLEDGEMENT

This work was supported by the National Technological University of Argentina (UTN) and the National Scientific and Technical Research Council of Argentina (CONICET), and by the projects UTI4375TC and EIUTNVM0003581, both funded by the UTN.

REFERENCES

[1] F. Hutter, J. Lücke, and L. Schmidt-Thieme, "Beyond Manual Tuning of Hyperparameters," *KI - Künstliche Intelligenz*, vol. 29, pp. 329–337, Nov. 2015.

[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.), pp. 2546–2554, Curran Associates, Inc., 2011.

[3] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An Automatic Algorithm Configuration Framework," *J. Artif. Intell. Res. (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]*, June 2012. arXiv: 1206.2944.

[5] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *arXiv:1206.5533 [cs]*, June 2012. arXiv: 1206.5533.

[6] J. Bergstra and Y. Bengio, "Random Search for Hyper-parameter Optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*. 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, pp. 529–533, Feb. 2015.

[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.

[10] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[11] G. A. Rummery and M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 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*, pp. 216–224, Morgan Kaufmann, 1990.

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

[14] 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.

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

[16] J. Močkus, "On Bayesian Methods for Seeking the Extremum," in *Optimization Techniques IFIP Technical Conference* (P. D. G. I. Marchuk, ed.), Lecture Notes in Computer Science, pp. 400–404, Springer Berlin Heidelberg, 1975. DOI: 10.1007/978-3-662-38527-2_55.

[17] 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.

[18] 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.), pp. 117–129, North-Holand, 1978.

[19] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient Global Optimization of Expensive Black-Box Functions," *Journal of Global Optimization*, vol. 13, pp. 455–492, Dec. 1998.

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

[21] 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.

[22] J. Laird, *The Soar cognitive architecture*. Cambridge,Mass ; London, England: MIT Press, 2012.