Direct Policy Search Reinforcement Learning for Robot Control

Andres El-Fakdi [1], Marc Carreras and Narcís Palomeras
University of Girona, Spain

Abstract.
In this paper, we present Policy Methods as an alternative to Value Methods to solve Reinforcement Learning problems. The paper proposes a Direct Policy Search algorithm that uses a Neural Network to represent the control policies. Details about the algorithm and the update rules are given. The main application of the proposed algorithm is to implement robot control systems, in which the generalization problem usually arises. In this paper, we point out the suitability of our algorithm in a RL benchmark, that was specially designed to test the generalization capability of RL algorithms. Results check out that policy methods obtain better results than value methods in these situations.

Keywords. Reinforcement learning, Direct Policy Search and Robot Learning

1. Introduction

A commonly used methodology in robot learning is Reinforcement Learning (RL) [1]. In RL, an agent tries to maximize a scalar evaluation (reward or punishment) obtained as a result of its interaction with the environment. The goal of a RL system is to find an optimal policy which maps the state of the environment to an action which in turn will maximize the accumulated future rewards. Most RL techniques are based on Finite Markov Decision Processes (FMDP) causing finite state and action spaces. The main advantage of RL is that it does not use any knowledge database, so the learner is not told what to do as occurs in most forms of machine learning, but instead must discover actions yield the most reward by trying them. Therefore, this class of learning is suitable for online robot learning. The main disadvantages are a long convergence time and the lack of generalization among continuous variables.

The dominant approach for solving the RL problem has been the use of a value-function but, although it has demonstrated to work well in many applications, it has several limitations. If the state-space is not completely observable (POMDP), small changes in the estimated value of an action cause it to be, or not be, selected; and this will detonate in convergence problems [2]. Over the past few years, studies have shown that approximating directly a policy can be easier than working with value functions, and better results can be obtained [3,4]. Instead of approximating a value function, new methodolo-

[1] Correspondence to: Andres El-Fakdi, Edifici PIV, Campus Montilivi, Universitat de Girona, 17071 Girona, Spain. Tel.: +34 972 419 871; Fax: +34 972 418 259; E-mail: aelfakdi@eia.udg.es.

gies approximate a policy using an independent continuous function approximator with its own parameters, trying to maximize the expected reward. Examples of direct policy methods are the REINFORCE algorithm [5], the direct-gradient algorithm [6] and certain variants of the actor-critic framework [7]. The advantages of policy methods against value-function based methods are various. A problem for which the policy is easier to represent should be solved using policy algorithms [4]. Working this way should represent a decrease in the computational complexity and, for learning control systems which operate in the physical world, the reduction in time-consuming would be notorious. Furthermore, learning systems should be designed to explicitly account for the resulting violations of the Markov property. Studies have shown that stochastic policy-only methods can obtain better results when working in POMDP than those ones obtained with deterministic value-function methods [8]. On the other side, policy methods learn much more slowly than RL algorithms using value function [3] and they typically find only local optima of the expected reward [9].

We propose the use of an online Direct Policy Search (DPS) algorithm, based on Baxter and Bartlett's direct-gradient algorithm OLPOMDP [10], for its application in the control system of a real system, such as a robot. This algorithm has the goal of learning a state/action mapping that will be applied in the control system. The policy is represented by a neural network whose input is a representation of the state, whose output is action selection probabilities, and whose weights are the policy parameters. The proposed method is based on a stochastic gradient descent with respect to the policy parameter space, it does not need a model of the environment to be given and it is incremental, requiring only a constant amount of computation step. The objective of the agent is to compute a stochastic policy [8], which assigns a probability over each action.

The work presented in this paper is the continuation of a research line about robot learning using RL, in which a more conventional value-function algorithm was first investigated [11,12]. The robot task used to test the algorithm was the learning of a target following behavior with an underwater robot. This robot task has already been tested in a simulation environment, obtaining very satisfactory results [13] . In this paper, we describe in detail our DPS algorithm and show its efficiency in a RL benchmark, the "mountain-car" task, to show the high generalization capability of policy methods.

2. The DPS algorithm

A partially observable Markov decision process (POMDP) consists of a state space *S*, an observation space *Y* and a control space *U*. For each state $i \in S$ there is a deterministic reward *r(i)*. As mentioned before, the algorithm is designed to work on-line, at every time step the learner (our robot) will be given an observation of the state and, according to the policy followed at that moment, it will generate a control action. As a result, the learner will be driven to another state and will receive a reward associated to this new state. This reward will allow us to update the controller's parameters that define the policy followed at every iteration, resulting in a final policy considered to be optimal or closer to optimal. The algorithm procedure is summarized in Table 1. The schema of the ANN, used to implement the control policy, can be seen in Figure 1.

The algorithm works as follows: having initialized the parameters vector θ_0, the initial state i_0 and the gradient $z_0 = 0$, the learning procedure will be iterated *T* times. At

Table 1. Algorithm: Baxter & Bartlett's OLPOMDP

OLPOMDP algorithm
1: Given: • $T > 0$ • Initial parameter values $\theta_0 \in \mathbb{R}^K$ • Arbitrary starting state i_0
2: Set $z_0 = 0$ ($z_0 \in \mathbb{R}^K$)
3: **for** $t = 0$ to T **do**
4: Observe state y_t
5: Generate control action u_t according to current policy $\mu(\theta, y_t)$
6: Observe the reward obtained $r(i_{t+1})$
7: Set $z_{t+1} = \beta z_t + \frac{\nabla \mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)}$
8: Set $\theta_{t+1} = \theta_t + \alpha r(i_{t+1}) z_{t+1}$
9: **end for**

every iteration, the parameters gradient z_t will be updated. According to the immediate reward received $r(i_{t+1})$, the new gradient vector z_{t+1} and a fixed learning parameter α, the new parameter vector θ_{t+1} can be calculated. The current policy μ_t is directly modified by the new parameters becoming a new policy μ_{t+1} that will be followed next iteration, getting closer, as $t \to T$ to a final policy μ_T that represents a correct solution of the problem.

In order to clarify the steps taken, the next lines will relate the update parameter procedure of the algorithm closely. The controller uses a neural network as a function approximator that generates a stochastic policy. Its weights are the policy parameters that are updated on-line every time step. The accuracy of the approximation is controlled by the parameter $\beta \in [0, 1)$.

The first step in the weight update procedure is to compute the ratio:

$$\frac{\nabla \mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)} \tag{1}$$

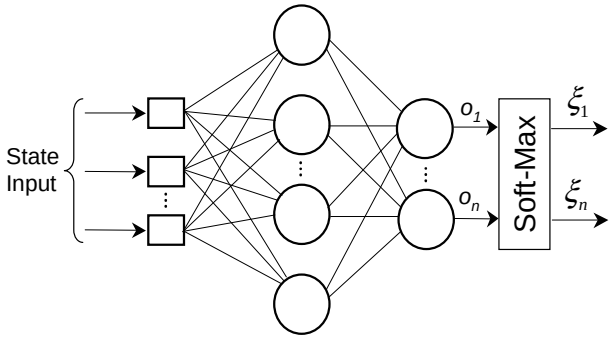

Figure 1. Schema of the ANN architecture used.

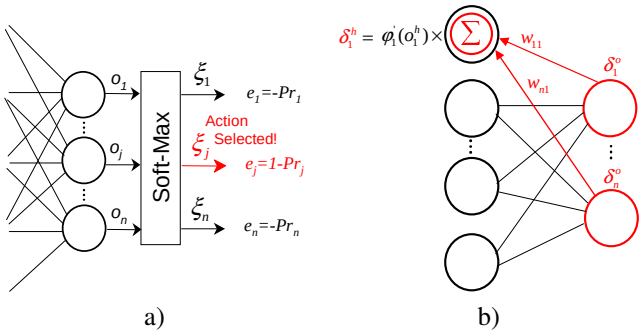

Figure 2. (a) Soft-Max error computation for every output. (b) Gradient computation for a hidden-layer neuron.

for every weight of the network. In an AANs, like the one used in the algorithm, the expression defined in step 7 of Table 1 can be rewritten as:

$$z_{t+1} = \beta z_t + \delta_t y_t \tag{2}$$

At any step time t, the term z_t represents the estimated gradient of the reinforcement sum with respect to the network's layer weights. In addition, δ_t refers to the local gradient associated to a single neuron of the ANN and it is multiplied by the input to that neuron y_t. In order to compute these gradients, we evaluate the soft-max distribution for each possible future state exponentiating the real-valued ANN outputs $\{o_1, ..., o_n\}$, being n the number of neurons of the output layer [14].

After applying the soft-max function, the outputs of the neural network give a weighting, $\xi_j \in (0, 1)$, to each of the vehicle's thrust combinations. Finally, the probability of the i^{th} thrust combination is then given by:

$$\Pr_i = \frac{\exp(o_i)}{\sum_{z=1}^{n} \exp(o_z)} \tag{3}$$

Actions have been labelled with the associated thrust combination, and they are chosen at random from this probability distribution. Once we have computed the output distribution over the possible control actions, next step is to calculate the gradient for the action chosen by applying the chain rule; the whole expression is implemented similarly to *error back propagation* [15]. Before computing the gradient, the error on the neurons of the output layer must be calculated. This error is given by next expression:

$$e_j = d_j - \Pr_j \tag{4}$$

The desired output d_j will be equal to 1 if the action selected was o_j and 0 otherwise (see Figure 2). With the soft-max output error calculation completed, next phase consists in computing the gradient at the output of the ANN and back propagate it to the rest of the neurons of the hidden layers. For a local neuron *j* located in the output layer we may express the local gradient for neuron j as:

$$\delta_j^o = e_j \cdot \varphi_j'(o_j) \tag{5}$$

Where e_j is the soft-max error at the output of neuron j, $\varphi_j^{'}(o_j)$ corresponds to the derivative of the activation function associated with that neuron and o_j is the function signal at the output for that neuron. So we do not back propagate the gradient of an error measure, but instead we back propagate the soft-max gradient of this error. Therefore, for a neuron j located in a hidden layer the local gradient is defined as follows:

$$\delta_j^h = \varphi_j^{'}(o_j) \sum_k \delta_k w_{kj} \tag{6}$$

When computing the gradient of a hidden-layer neuron, the previously obtained gradient of the following layers must be back propagated. In Equation 6 the term $\varphi_j^{'}(o_j)$ represents de derivative of the activation function associated to that neuron, o_j is the function signal at the output for that neuron and finally the summation term includes the different gradients of the following neurons back propagated by multiplying each gradient to its corresponding weighting (see Figure 2).

Having all local gradients of all neurons calculated, the expression in Equation 2 can be obtained and finally, the old parameters are updated following the expression:

$$\theta_{t+1} = \theta_t + \gamma r(i_{t+1}) z_{t+1} \tag{7}$$

The vector of parameters θ_t represents the network weights to be updated, $r(i_{t+1})$ is the reward given to the learner at every time step, z_{t+1} describes the estimated gradients mentioned before and γ is the learning rate of the DPS algorithm.

3. Experimental Results

3.1. The "mountain-car" task.

The "mountain-car" benchmark [16] was designed to evaluate the generalization capability of RL algorithms. In this problem, a car has to reach the top of a hill, see Figure 3. However, the car is not powerful enough to drive straight to the goal. Instead, it must first reverse up the opposite slope in order to accelerate, acquiring enough momentum to reach the goal. The states of the environment are two continuous variables, the position *p* and the velocity *v* of the car. The action *a* is the force of the car, which can be positive and negative. The reward is -1 everywhere except at the top of the hill, where it is 1. The dynamics of the system can be found in [16]. The episodes in the mountain-car task start in a random position and velocity, and they run for a maximum of 200 iterations or until the goal has been reached. The optimal state/action mapping is not trivial since depending on the position and the velocity, the action has to be positive or negative.

3.2. Results with a value-function algorithm

To provide a performance baseline, the classic Q_learning algorithm, which is based on a value function, was applied. The state space was finely discretized, with 180 states for the position and 150 for the velocity. The action space contained only three values, -1, 0 and 1. Therefore, the size of the Q table was 81000 cells. The exploration strategy was an $\epsilon - greedy$ policy with ϵ set at 30%. The discount factor was $\gamma = 0.95$ and

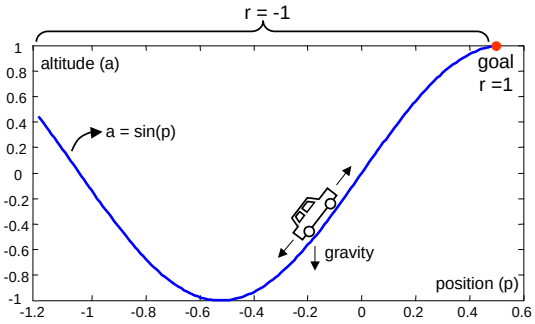

Figure 3. The "mountain-car" task domain.

the learning rate $\alpha = 0.5$, which were found experimentally. The Q table was randomly generated at the beginning of each experiment.In each experiment, a learning phase and an evaluation phase were repeatedly executed. In the learning phase, a certain number of iterations were executed, starting new episodes when it was necessary. In the evaluation phase, 500 episodes were executed. The *effectiveness* of learning was evaluated by looking the averaged number of iterations needed to finish the episode. After running 100 experiments with discrete Q_learning, the average number of iterations when the optimal policy had been learnt was 50 with 1.3 standard deviation. And the number of learning iterations to learn this optimal policy was $1x10^7$ learning iterations. Figure 4a shows the effectiveness evolution of the Q_learning algorithm in front of the learning iterations. It is interesting to compare this mark with other state/action policies. If a forward action ($a = 1$) is always applied, the average episode length is 86. If a random action is used, the average is 110. These averages depend highly on the fact that the maximum number of iterations in an episode is 200, since in a lot of episodes these policies do not fulfill the goal.

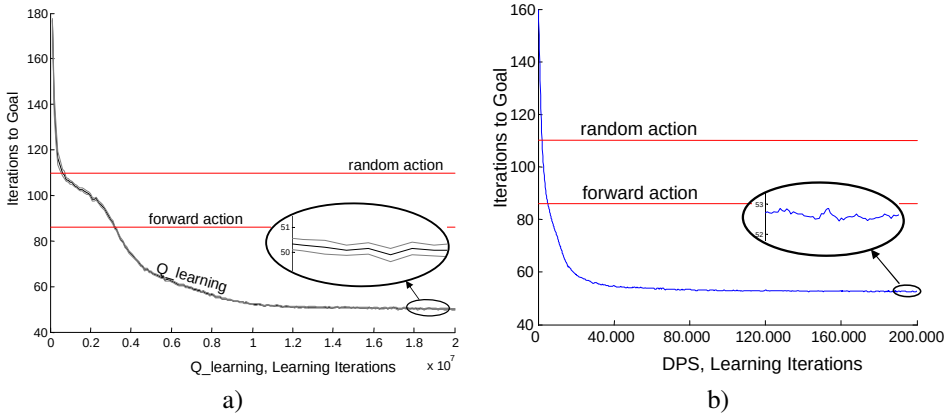

Figure 4. a) Effectiveness of the Q_learning algorithm with respect to the learning iterations. After converging, the effectiveness was maximum, requiring only 50 iterations to accomplish the goal. b) Effectiveness of the DPS algorithm with respect to the learning iterations. The convergence time was much smaller, while a similar effectiveness (52 iterations) was achieved.

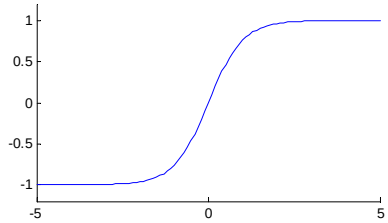

Figure 5. The hyperbolic tangent function.

3.3. Results with the DPS algorithm

A one-hidden-layer neural-network with 2 input nodes, 10 hidden nodes and 2 output nodes has been used to generate a stochastic policy. One of the inputs corresponds to the vehicle's position, the other one represents the vehicleŠs velocity. Each hidden and output layer has the usual additional bias term. The activation function used for the neurons of the hidden layer is the hyperbolic tangent type, see Equation 8 and Figure 5, while the output layer nodes are linear. The two output neurons have been exponentiated and normalized as explained in section 2 to produce a probability distribution. Control actions are selected at random from this distribution.

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} \tag{8}$$

In each experiment, a learning phase and an evaluation phase were repeatedly executed. In the learning phase, 500 number of iterations were executed, starting new episodes when it was necessary. In the evaluation phase, 200 episodes were executed. The *effectiveness* of learning was evaluated by looking the averaged number of iterations needed to finish the episode. After running 100 experiments with the DPS algorithm, the average number of iterations when the optimal policy had been learnt was 52.5. And the number of learning iterations to learn this optimal policy was 40.000 learning iterations. Figure 4b shows the effectiveness evolution of the DPS algorithm in front of the learning iterations.

3.4. Comparison

After performing the experiments with the Q_learning algorithm and the DPS algorithm it can be concluded:

Simplicity A very simple NN configuration was able to learn the necessary policy. However, Q_learning, which was affected by the generalization problem, required 81000 cells to obtain a similar policy.

Effectiveness The *minimum iterations to goal* achieved by DPS (52.5) was practically equal than the ones achieved by Q_learning (50).

Swiftness Although policy methods learn usually slower than value methods, in this case, the DPS algorithm was much faster than Q_learning (affected by the generalization problem)

4. Conclusions and Further Work

This paper has presented Policy Methods as an alternative to Value Methods to solve Reinforcement Learning problems. The paper has proposed a Direct Policy Search algorithm based on Baxter and Bartlett's direct-gradient algorithm, with a Neural Network to represent the policies. Details about the algorithm with all the update rules were given. The main application of the proposed algorithm is to implement robot control systems, in which the generalization problem usually arises. In this paper, we have pointed out the suitability of our algorithm in a RL benchmark, specially designed to test the generalization capability of RL algorithms. Results have shown better results of policy methods in these situations. Future work will consist on testing the DPS algorithm with real robots.

References

[1] R. Sutton and A. Barto. *Reinforcement Learning, an introduction*. MIT Press, 1998.

[2] D.P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[3] R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems*, 12:1057–1063, 2000.

[4] C. Anderson. Approximating a policy can be easier than approximating a value function. Technical Report Computer Science CS-00-101, Colorado State University, 2000.

[5] R. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[6] J. Baxter and P.L. Barlett. Reinforcement learning in POMDPs via direct gradient ascent. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[7] V.R. Konda and J.N. Tsitsiklis. On actor-critic algorithms. *SIAM Journal on Control and Optimization*, 42(4):1143–1166, 2003.

[8] S.P. Singh, T. Jaakkola, and M.I. Jordan. Learning without state-estimation in partially observable markovian decision processes. In *Proceedings of the Eleventh International Conference on Machine Learning*, New Jersey, USA, 1994.

[9] N. Meuleau, L. Peshkin, and K. Kim. Exploration in gradient-based reinforcement learning. Technical Report AI Memo 2001-003, MIT, 2001.

[10] J. Baxter and P.L. Bartlett. Direct gradient-based reinforcement learning i: Gradient estimation algorithms. Technical report, Australian National University, 1999.

[11] M. Carreras, P. Ridao, and A. El-Fakdi. Semi-online neural-q-_learning for real-time robot learning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Las Vegas, USA, 2003.

[12] M. Carreras and P. Ridao. Solving a RL generalization problem with the SONQL algorithm. In *Seventh Catalan Conference on Artificial Intelligence*, 2004.

[13] A. El-Fakdi, M. Carreras, N. Palomeras, and P. Ridao. Autonomous underwater vehicle control using reinforcement learning policy search methods. In *IEEE Conference and Exhibition Oceans'05 Europe*, June 2005.

[14] Aberdeen D. A. *Policy Gradient Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Australian National University, 2003.

[15] S. Haykin. *Neural Networks, a comprehensive foundation*. Prentice Hall, 2nd ed. edition, 1999.

[16] A.W. Moore. Variable resolution dynamic programming: Efficiently learning action maps on multivariate real-value state-spaces. In *Proceedings of the Eighth International Conference on Machine Learning*, 1991.