Monte-Carlo Swarm Policy Search

Jeremy Fix and Matthieu Geist

Supélec, IMS research group
2 rue edouard Belin, 57070 Metz (France)
firstname.lastname@supelec.fr

Abstract. Finding optimal controllers of stochastic systems is a particularly challenging problem tackled by the optimal control and reinforcement learning communities. A classic paradigm for handling such problems is provided by Markov Decision Processes. However, the resulting underlying optimization problem is difficult to solve. In this paper, we explore the possible use of Particle Swarm Optimization to learn optimal controllers and show through some non-trivial experiments that it is a particularly promising lead.

Keywords: particle swarm optimization, optimal control, policy search

1 Introduction

Reinforcement Learning (RL) [12] addresses the optimal control problem. In this paradigm, at each (discrete) time step the system to be controlled is in a given state (or configuration). Based on this information, an agent has to choose an action to be applied. The system reacts by stochastically stepping to a new configuration, and an oracle provides a reward to the agent, depending on the experienced transition. This reward is a local hint of the quality of the control, and the aim of the agent is to choose a sequence of actions in order to maximize some cumulative function of the rewards. A notable advantage of this paradigm is that the oracle quantifies how the agent behaves without specifying what to do (for example, when learning to play chess, a reward would be given for wining the game, not for taking the queen).

The mapping from configurations to actions is called a policy (or a controller); its quality is quantified by the so-called value function which associates to each state an expected measure of cumulative reward from starting in this state and following the policy. The best policy is the one with associated maximal value function. Among other approaches, direct policy search algorithms (*e.g.*, [1]) adopt a parametric representation of the policy and maximize the value function (as a function of the controller's parameters). This approach is sound, but the underlying optimization problem is difficult. Even for simple policies, computing the gradient of the related objective function is far from being straightforward.

In the numerical optimization community, several algorithms requiring only to evaluate the objective function have been devised, among which one finds

genetic algorithms, particle swarm optimization and ant algorithms. These approaches involve a set of individuals (each representing a set of parameters related to the objective function of interest) that are combined, trying to reach a global optima. Particle swarm optimization is one of these algorithms, proposed originally by [7]. Variations of this algorithm have been proposed and a thorough review can be found in [3]. It has been shown that PSO (the original or one of its variations) performs well for optimization problems whether uni- or multi-modal, with static or dynamic fitness and even in large search space [4].

In this article, we introduce a simple but new RL policy search algorithm using particle swarm optimization at its core. We show that it is particularly efficient for optimizing the parameters of controllers for three classical benchmark problems in reinforcement learning : the inverted pendulum, the mountain car and the acrobot. The two first problems involve noise in the evolution of the system which introduces random fluctuations in the fitness landscape. In the last problem, we evaluate the performence of PSO in a large search space. The acrobot is known as being a very difficult problem in the RL community, and most approaches fail to solve it.

2 Monte Carlo Swarm Policy Search (MCSPS)

A Markov Decision Process (MDP) is a tuple $\{S, A, P, R\}$ with the state space S, action space A, a set of Markovian transition probabilities P and a reward function R. A policy is a mapping from states to probabilities over actions: $\pi : S \to \mathcal{P}(A)$. At each time step i, the system to be controlled is in a state s_i, the agent chooses an action a_i according to a policy π, $a_i \sim \pi(.|s_i)$. It is applied to the system which stochastically transits to s_{i+1} according to $p(.|s_i, a_i)$. The agent receives a reward $r_i = R(s_i, a_i, s_{i+1})$. Its goal is to find the policy which maximizes some cumulative function of the rewards, over the long run; this is the so-called value function. There are many ways to define a value function. The more common one is to consider the expected discounted cumulative reward (expectation being according to stochasticity of transitions and of the policy): $V^\pi(s) = E[\sum_{i=0}^{\infty} \gamma^i r_i | s_0 = s, \pi]$, the term $\gamma \in (0, 1)$ being the discount factor and weighting long-term rewards. Another common criterion hold if a finite horizon T is considered: $V^\pi(s) = E[\sum_{i=0}^{T} r_i | s_0 = s, \pi]$. Another possible criterion, less common because less convenient (from a mathematical point of view) is the mean reward: $V^\pi(s) = \lim_{n\to\infty} \frac{1}{n} E[\sum_{i=0}^{n} r_i | s_0 = s, \pi]$..

For any definition of the value function, the criterion to be optimized is the expected value over a distribution p_0 of initial states:

$$\rho^\pi = E[V^\pi(s_0) | s_0 \sim p_0]. \tag{1}$$

The optimal policy π^* is the one maximizing this criterion:

$$\pi^* = \underset{\pi: S\to\mathcal{P}(A)}{\operatorname{argmax}} \ \rho^\pi. \tag{2}$$

In the considered policy search context, we make some assumptions. First, the model (that is transition probabilities and the reward function) is unknown.

However, we assume that a simulator is available, so that we can sample trajectories according to any policy of interest (which can be a well-founded hypothesis, depending on the problem of interest). Second, we assume that a parametric structure is chosen for the controller beforehand: any policy π_θ is parameterized by a parameter vector θ (for example, it can be a Gibbs sampler constructed from a radial basis function networks, and the parameters are the weights of the kernels). The optimization problem to be solved is therefore the following:

$$\theta^* = \underset{\theta \in \mathbb{R}^p}{\operatorname{argmax}} \, \rho^{\pi_\theta}. \tag{3}$$

Indeed, this is a quite difficult optimization problem. It has been proposed to solve it using a gradient ascent [1] or cross-entropy [9], among other approaches. As the model is unknown, the gradient should be estimated from simulation, which causes a high variance.

In this paper, we introduce a simple idea: using a particle swarm optimizer to solve this difficult optimization problem. Each particle holds a parameter vector, that is a controller, and the fitness function is ρ^{π_θ}. As the model is unknown, it cannot be computed analytically. However, as a simulator is available, it can be estimated using Monte Carlo. For example, consider the finite horizon value function. One generates M trajectories, starting in a random state s_0 sampled according to p_0, and a trajectory of length T is obtained by applying the policy π_θ and following the system's dynamic. From such trajectories $\{(s_0^m, a_0^m, s_1^m, r_0^m \dots s_T^m, r_{T-1}^m)_{1 \leq m \leq M}\}$, one can compute

$$\hat{\rho}^{\pi_\theta} = \frac{1}{M} \sum_{m=1}^{M} \sum_{i=0}^{T-1} r_i^m, \tag{4}$$

which is an unbiased estimate of the true fitness function ρ^{π_θ}.

More precisely, we consider a swarm with N particles with a von Neumann topology. In all the simulations presented below, we used a swarm of 5×5 particles. Different rules to update the position and velocity of the particles have been proposed in the litterature (see [3] for a review). We used the basic PSO with a constriction factor [7, 2]. Namely, we use the following equations to update the velocity $\mathbf{v}_i$ and position $\mathbf{p}_i$ of a particle i:

$$\begin{aligned} \mathbf{v}_{ij} &= w\mathbf{v}_{ij} + c_1 r_1 . (\mathbf{b}_{ij} - \mathbf{p}_{ij}) + c_2 r_2 . (\mathbf{l}_{ij} - \mathbf{p}_{ij}) \\ \mathbf{p}_i &= \mathbf{p}_i + \mathbf{v}_i \end{aligned} \tag{5}$$

with $w = 0.729844, c_1 = c_2 = 1.496180$, r_1, r_2 are random numbers uniformly drawn from $[0, 1]$, $\mathbf{b}_i$ is the best position ever found by the particle i and $\mathbf{l}_i$ the best position ever found by one particle in the neighborhood of particle i. The position of the particles are initialized randomly in the parameter space while the velocities are initialized to zero. The position and velocity of the particles are updated asynchronously. At each iteration, we need to compute the fitness of a particle and update its position given the position and fitness of the particles within their neighborhood. Given our problems are stochastic we evaluate the

fitness of a particle each time its position changed and also reevaluate the fitness of its best position each time we want to change it. Each update of a particle's state and fitness is propagated to the neighborhoods to which the particle belongs. The scripts for all the simulations presented in the paper are available online [5].

3 Results

3.1 Inverted pendulum

Problem

The inverted pendulum is a classic benchmark problem in reinforcement learning and has already been addressed with several methods (see e.g. [8]). We use the same setting as in [8]. It consists of finding the force to apply to a cart, on which a pendulum is anchored, in order to maintain the pendulum still at the vertical position. The state of the system is the angle of the pendulum relative to the upright and its angular speed $(\theta, \dot{\theta})$, which are updated according to the equations : $\ddot{\theta} = \frac{g\sin(\theta) - \alpha ml\sin(2\theta)\dot{\theta}^2/2 - \alpha\cos(\theta)(f+\eta)}{4l/3 - \alpha ml\cos^2(\theta)}$, where g is the gravity constant ($g = 9.8m/s^2$), m and l are the mass and length of the pole ($m = 2.0$ kg, $l = 0.5$ m), M the mass of the cart ($M = 8.0$ kg) and $\alpha = \frac{1}{m+M}$. The time-step τ is set to 0.1s. The pole must be held in $[-\frac{\pi}{2}; \frac{\pi}{2}]$. An episode is constrained to last at most 3000 iteractions. At each interaction, a reward of 0 is given until the pole exits this domain which ends the episode and leads to a reward of -1. This reward is actually poorly informative as it is only indicating that the pole should not fall but not that the optimal position is around 0 (which can be induced by a cosine reward for example). The pole is initialized close to equilibrium ($\theta_0 \in [-0.1, 0.1], \dot{\theta}_0 \in [-0.1, 0.1]$). The pole-cart system is controlled by applying a force $f_t \in \{-50, 0, 50\}$ Newtons perturbed by a uniform noise $\eta \in [-5; 5]$ Newtons to the cart.

The controller is defined with a radial basis function network (RBF) with 9 Gaussians and a constant term per action. The means of the basis functions are evenly spread in $[-\pi/4, \pi/4] \times [-1.0, 1.0]$ and the standard deviation is set to $\sigma = 1.0$. Optimizing this controller means finding 30 parameters, i.e. the amplitude of the 27 basis functions and the 3 constant terms. The RBF associated to each action defines the probability to select that action (c_i and $a_{i,j}$ being parameters to be learnt): $\forall i \in [1, 3], P_i = \frac{1}{P}\exp(c_i + \sum_{j=1}^{9} a_{i,j} exp(-\frac{(\theta-\theta_j)^2 + (\dot{\theta}-\dot{\theta}_j)^2}{2\sigma^2}))$, where P is a normalizing term so that the probabilities sum to 1.0. An action is selected with a probabilistic toss biased by these probabilities.

Experimental results

The experiment is repeated 1000 times. For each iteration of one swarm, the fitness of a particle is evaluated using a single trajectory which makes an iteration much faster but also much more subject to the stochasticity of the problem due to the definition of the initial state, to the selection of the action using a random

toss and to the uniform noise added to the controller. Random fluctuations of the fitness remains, as it was checked on some trials by evaluating several times the fitness of a set of parameters but this is not shown here. The fitness we report on the illustrations is the fitness of the best particle evaluated on 500 trajectories to get an accurate estimate of it. The swarm is allowed to evolve during 200 iterations.

The average number of balancing steps of the best particle, and its standard deviation, are plotted over the iteration of the swarms on figure 1. As shown on the figure, all the trials converged to a good policy allowing to keep the pendulum balancing for the 3000 time steps, the maximal length of an episode. On average, it took approximately 50 iterations of the swarm to converge to a very good policy.

A standard approach for policy search consists in performing a gradient ascent of the value function respectively to the parameters of the policy [1]. It also requires to simulate trajectories. For this problem, unreported experiments shows that gradient ascent took an order of 200.10^3 trajectories before reaching an optimal policy. The proposed approach took an order of 1250 trajectories to reach the same result (25 particles, 50 iterations and one simulated trajectory per fitness evaluation). Meta parameters (c_1, c_2, w for PSO, learning rates and forgetting factor for the gradient ascent) could certainly be better defined. However this shows that MCPSO easily achieves state of the art policy search performance.

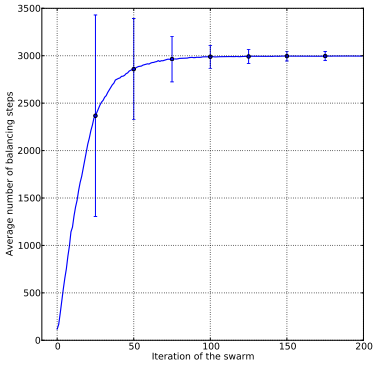


Fig. 1. Average number of balancing steps for the best particle. This average is computed over the 1000 trials, and using 500 trajectories for each trial at each epoch to get an accurate estimate of it. Please note that we plot here the number of balancing steps and not the fitness which is more intuitive. Error bars indicate one standard deviation. The simulation scripts are available at [5].

3.2 Mountain car

Problem

The second problem we consider is the mountain car as described in [12]. The goal is to control a vehicle in order to escape from a valley. Given the car has a limited power, it must swing forward and backward in the valley to reach the exit. The state is defined as the position $x \in [-1.2, 0.5]$ and velocity $\dot{x} \in [-0.07, 0.07]$ of the car. Three discrete actions are allowed : accelerating to the left, doing nothing or accelerating to the right $a \in \{-1, 0, 1\}$. The system evolves according to discrete time equations provided in [12, Chap 8]. The position is bounded in the domain $[-1.2, 0.5]$. The cart is initialized randomly close to the worst cases, at the bottom of the valley with a speed close to zero ($x_0 \in [-0.75, -0.25], \dot{x}_0 \in [-0.02, 0.02]$). When the cart's position reaches the lower bound, the velocity is set to 0.0. When the cart reaches the upper bound, the episode ends with a reward of 0; the reward is set to -1 otherwise. The length of an episode is limited to 1500 interactions. The goal of the swarm is to find a set of parameters that maximizes the reward which is equivalent to minimizing the number of steps necessary to escape from the valley.

The controller is defined by a set of 9 basis functions (Gaussians) plus a constant term for each action, leading to 30 parameters to optimize. If the state (position and velocity) is scaled in $[0, 1]$, the centers of the basis functions are evenly spread in $[0, 1] \times [0, 1]$ and the standard deviation set to $\sigma = 0.3$. Similar to the inverted pendulum problem, the value of these 3 basis networks is used as probabilities to toss an action.

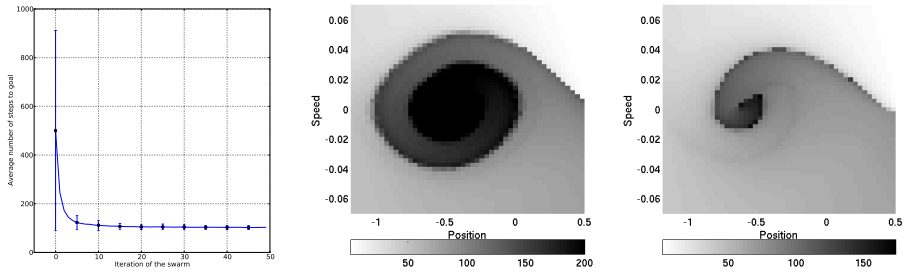
Experimental results

We repeated 1000 experiments. For each experiment, the swarm is allowed to evolve during 50 epochs (which was enough to get a good policy). At each epoch, the fitness of a particle is evaluated using 30 trajectories (it does not suppress the stochasticity of the fitness as we checked on some trials, but this is not shown here). The reported fitness of the best particle is evaluated using 1000 trajectories to get an accurate estimate of it. The evolution of the average number of steps to reach the goal of the best particles is shown on figure 2a), with its standard deviation. The average number of steps to reach the goal for the initial and final best particles of a typical trial are plotted on figures 2b,c.

3.3 Swing-up acrobot

Problem

The aim of the acrobot problem is to swing an under-actuated two-arm pendulum, starting from a vertical position pointing down in order to reach the vertical pointing up unstable position. The system's state is defined by four continuous variables (the two joints' angle θ_1, θ_2 and their velocity $\dot{\theta}_1, \dot{\theta}_2$). The system is controlled with a torque $\tau \in \{-1, 0, 1\}$ applied to the second joint. The torque is only applied on the joint between the two segments, the system being therefore under-actuated and solving the task requires to swing the pendulum

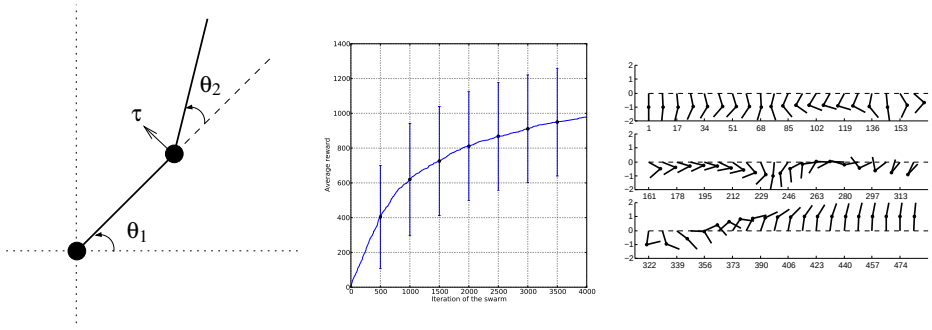


a) b) c)

Fig. 2. a) Average number of steps to reach the goal state with its standard deviation. b) Average number of steps to reach the goal state for evenly spread initial conditions $(\theta, \dot{\theta})$ during a typical trial. For the illustration, this average is bounded to 200 but reaches 1500 in the worst case (the length of an episode). c) Average number of steps to reach the goal state for the best particle after 50 iterations.

back and forth. The system's state evolves according to the discrete-time equations provided in [11] with the strength $\tau \in \{-1, 0, 1\}$, time-step $\Delta t = 0.01s.$, $\dot{\theta}_{1,t} \in [-9\pi, 9\pi]$, $\dot{\theta}_{2,t} \in [-4\pi, 4\pi]$, $m_1 = m_2 = 1$, $l_1 = l_2 = 1$, $l_{c_1} = l_{c_2} = 0.5$, $I_1 = I_2 = ml^2/12$, $g = 9.8$. The state is initialized at the vertical position pointing down with a null speed $\theta_{1,0} = 3\pi/2, \theta_{2,0} = 0, \dot{\theta}_{1,0} = \dot{\theta}_{2,0} = 0$ (see fig. 3).

Controlling the acrobot is a difficult problem[10]. To ease the problem, we considered a simplified controller which combines a RBF network with an optimal Linear Quadratic Regulator (LQR) [11]. The LQR controller can maintain the pendulum still in the vertical upward position but is unable to swing it. In addition, the LQR controller works perfectly only in a narrow range of the state space; for $\dot{\theta}_1 = \dot{\theta}_2 = 0$, the LQR controller stabilizes the pendulum if the initial state is in $\theta_2 = 0, \theta_1 = \pi/2 \pm \pi/24$. Therefore, the RBF controller has to swing the pendulum in order to bring it at the vertical position with a certain speed to allow the LQR to stabilize it. We used a continuous action defined as the tanh of a RBF involving 4 gaussians per dimension. The RBF controller therefore involves $4^4 = 256$ parameters. When the pendulum is close to the goal state $(\theta_1 = \pi/2 \pm \pi/4, \theta_2 = 0 \pm \pi/4, \dot{\theta}_1 = 0 \pm \pi/2, \dot{\theta}_2 = 0 \pm \pi/2$, denoted $\mathbf{D}_\theta)$, the controller is switched from the RBF to the LQR. It has also to be noted that the LQR controller is not optimized in these experiments but computed before-hand (see [11]). Better controllers could certainly be designed but the point here was to test the ability of PSO to find the parameters in such a large parameter space. Given the simulations are expensive, the problem is here considered deterministic (no noise in the initial state nor in the chosen action).



a) b) c)

Fig. 3. a) Setup of the acrobot problem. Starting from the vertical pointing-down position, the controller, influencing the pendulum through the torque τ shall bring and keep the pendulum still in a domain close to the vertical pointing-up unstable position b) Average number of steps the pendulum stays in the goal domain. This average is computed over 300 repetitions of the experiment. c) Behavior of one of the best policies.

The controllers are defined as :

$$\tau = \begin{cases} -\mathbf{K}^T.\theta, \theta = (\theta_1 - \pi/2; \theta_2; \dot{\theta_1}; \dot{\theta_2}) & \text{if } \theta \in \mathbf{D}_\theta \\ 2\tanh(\sum_{j=1}^{256} a_j e^{-\frac{\sin^2(\theta_1-\theta_1^j)}{2\sigma_1^2} - \frac{\sin^2(\theta_2-\theta_2^j)}{2\sigma_2^2} - \frac{(\dot{\theta_1}-\dot{\theta_1}^j)^2}{2\sigma_3^2} - \frac{(\dot{\theta_2}-\dot{\theta_2}^j)^2}{2\sigma_4^2}}) & \text{otherwise} \end{cases} \quad (6)$$

with $\sigma_1 = \sigma_2 = 0.25, \sigma_3 = \sigma_4 = 4.5$, the centers of the gaussians being evenly spread in $[-\pi/4, 5\pi/4] \times [-\pi/4, 5\pi/4] \times [-9, 9] \times [-9, 9]$.

Experimental results

We repeated the experiments over 300 trials. A simulation is allowed to run for at most $20s$. (2000 interactions). The swarm is evolving during 4000 iterations. A reward of $+1$ is given each time the pendulum is in the goal region (as defined above), and 0 otherwise. The average reward function of the iteration of the swarm is shown on figure 3b. As we can see, the swarm does not always converge to an optimal policy and get stuck in local minima. This is probably due to the architecture of the controller which is certainly not optimal. In addition, during the iteration of the algorithm, the fitness tends to stay on "plateau". There are nevertheless policies that are close to optimal as for example the one depicted on figure 3c. This example illustrates that PSO is able to optimize controllers even in large parameter space but the controller can be improved.

4 Discussion

Particle Swarm Optimization is an efficient algorithm for solving optimization problems. In addition to the different problems on which it has been applied before, we have shown here that it reveals to be very efficient to optimize the parameters of controllers solving challenging optimal control problems. It is also a very convenient algorithm if we compare it to the gradient-based policy search algorithm since we do not have to compute the gradient of the policy nor do we need it to be computable. Moreover, PSO is less prone to local optimum and converges more quickly than gradient-based approaches. A lack of the current approach is that it requires a simulator. However, in some cases, only data sampled according to a fixed behaviorial policy are available. To extend the current approach to this case, we envision to replace the Monte Carlo estimation of the fitness function by value function approximation [6]. Ultimately, on can envision to design an online algorithm, with an agent learning to control optimally the system while interacting with it.

References

1. Baxter, J., Bartlett, P.: Direct gradient-based reinforcement learning. JAIR (1999)
2. Clerc, M., Kennedy, J.: The particle swarm - explosion, stability, and convergence in a multidimensional complex space. IEEE Trans. Evol. Comp. 6(1), 58–73 (2002)
3. Engelbrecht, A.: Fundamentals of Computational Swarm Intelligence. Wiley (2005)
4. Engelbrecht, A.: Heterogeneous particle swarm optimization. In: Swarm Intelligence, Lecture Notes in Computer Science, vol. 6234, pp. 191–202. Springer (2010)
5. Fix, J., Geist, M.: http://jeremy.fix.free.fr/spip.php?article33
6. Geist, M., Pietquin, O.: Parametric Value Function Approximation: a Unified View. In: ADPRL 2011 (2011)
7. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings IEEE International Joint Conference on Neural Networks. pp. 1942–1948 (1995)
8. Lagoudakis, M., Parr, R.: Least-squares policy iteration. JMLR 4 (2003)
9. Mannor, S., Rubinstein, R., Gat, Y.: The cross entropy method for fast policy search. In: International Conference on Machine Learning. vol. 20, p. 512 (2003)
10. Munos, R., Moore, A.: Variable resolution discretization for high-accuracy solutions of optimal control problems. In: IJCAI. pp. 1348–1355 (1999)
11. Spong, M.W.: The swing up control problem for the acrobot. IEEE Control Systems 15, 49–55 (1995)
12. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press (1998)