Learning Cost-Efficient Control Policies with XCSF: Generalization Capabilities and Further Improvement

Didier Marin, Jérémie Decock, Lionel Rigoux, and Olivier Sigaud
Institut des Systèmes Intelligents et de Robotique
Paris, France
marin@isir.upmc.fr, jeremie.decock@gmail.com, rigoux@isir.upmc.fr,
olivier.sigaud@upmc.fr

ABSTRACT

In this paper we present a method based on the "*learning from demonstration*" paradigm to get a cost-efficient control policy in a continuous state and action space. The controlled plant is a two degrees-of-freedom planar arm actuated by six muscles. We learn a parametric control policy with XCSF from a few near-optimal trajectories, and we study its capability to generalize over the whole reachable space. Furthermore, we show that an additional *Cross-Entropy Policy Search* method can improve the global performance of the parametric controller.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning—*Parameter learning*

General Terms

Theory, Algorithms

Keywords

XCSF, Reinforcement Learning, Cross-Entropy, Control

1. INTRODUCTION

The design of adaptive systems able to deal with sequential decision problems in continuous state and action spaces is a difficult matter. The straightforward approach to this problem consists in adapting the conceptual tools of the discrete Reinforcement Learning (RL) framework to the continuous case. This generates intensive research and, in particular, the direct Policy Search methods are providing more and more interesting results (e.g. [9, 10]). But these methods are complex, require a lot of tuning and are plagued with local minima problems. In particular, they include no generalization mechanism. In [11], Lanzi presented Learning Classifier Systems (LCSs) as RL systems endowed with a generalization property. In the recent LCS literature, some preliminary attempts have addressed the continuous action case, but along one dimension only and with limited results [23, 28].

Meanwhile, LCS research has significantly shifted towards supervised learning problems, either classification or regression (e.g. [1]). In that context, solving single-step problems has received more and more attention in the community by contrast with multi-step, sequential decision problems. One of the most celebrated LCS in the literature, XCSF, can be seen as the ultimate representative of this tendency. Indeed, it is a competitive general purpose function approximation tool based on regression mechanisms, from which all sequential decision mechanisms and the notion of action have been removed.

However, one can solve sequential decision or control problems with classification or regression tools by learning from demonstration. This approach provides a more indirect, but also easier way towards adaptive control or continuous state and action decision making, provided a pre-existing non-adaptive solution. In this paper, we present an instantiation of such an approach, where the excellent regression capabilities of XCSF are combined with a stochastic optimization process so as to learn a policy from demonstration and then optimize it by tuning its parameters.

The broader context of the study is the design of a computational model of skill consolidation in reaching movements based on a previous model of human motor control. In [14], the authors presented a model of reaching movements based on a computationally expensive variational calculus process. A crucial feature of their model is that the generated movements do not depend on time. This results in the possibility to learn stationary policies from the model. Here, we focus on the capability of XCSF to learn such continuous state and action policies from a set of trajectories generated by this planning system. In particular, we show that the propensity of XCSF to generalize based on learning from only a few planned movements can result in the generation of an efficient controller for the whole state space of the plant.

However, the resulting policy is generally suboptimal. Thus, in a second step, we show that the policy represented by XCSF classifiers can be improved by a stochastic optimization process that acts on the population parameters. This additional process can be seen as a way of reintroducing an RL process into XCSF. We study experimentally the capability of this additional process to improve the performance of the resulting policy.

The paper is organized as follows. In Section 2, we present all the methods used in the paper to realize the skill consol-

GECCO'11, July 12–16, 2011, Dublin, Ireland.
Copyright 2011 ACM 978-1-4503-0557-0/11/07 ...$10.00.

idation model. In Section 3, we present the model itself and the design of the experiments. Results are given in Section 4 and discussed in Section 5. Finally, Section 6 concludes and presents the perspectives of this work.

2. METHODS

In this section, we describe the methods used in our work. We start with the background necessary to understand the optimal control method used in [14] and the Cross-Entropy Policy Search (CEPS) algorithm used to optimize a parametric policy. Then, we give an overview of XCSF and its use for learning the controller optimized by CEPS.

2.1 Optimal control of reaching movements

2.1.1 Optimal control and MDPs

Optimal control and Markov Decision Processes (MDPs) with continuous state and actions are two similar frameworks for solving problems where a plant must perform a task while optimizing an objective function [2]. At a time step t, the state of the plant is a vector $\boldsymbol{x}_t \in \mathcal{X}$ and the action it performs is a vector $\boldsymbol{u}_t \in \mathcal{U}$. The objective function $r : \mathcal{X} \times \mathcal{U} \to \mathbb{R}$ indicates how interesting it is to be in a given state and to perform a given action, relatively to the problem at hand. In the optimal control literature, the objective function is expressed as a cost function to be minimized. In the RL framework, it is called the "reward function" and must generally be maximized.

The quality of a policy π for a given state $\boldsymbol{x}$ is expressed by its value function, which is the expectation of the discounted sum of costs/rewards over time

$$V^{\pi}(\boldsymbol{x}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(\boldsymbol{x}_t, \boldsymbol{u}_t) | \boldsymbol{x}_0 = \boldsymbol{x}\right]$$

where γ is the discount factor which is related to the uncertainty about the future. In optimal control, the equivalent of the value function is called the cost-to-go.

2.1.2 A model of reaching movements

In [14], the authors proposed a model of human reaching movement based on the now prominent optimal control paradigm in the human motor control literature [7, 22]. The model is based on the assumption that motor control is governed by an optimal feedback policy computed at each visited state with respect to the value function V, for a reward function that involves a trade-off between muscular effort and goal-related reward

$$r(\boldsymbol{x}_t, \boldsymbol{u}_t) = \alpha \|\boldsymbol{u}_t\|^2 - \beta g(\boldsymbol{x}_t) \quad (1)$$

where α is a weight on the effort term, g is a function which is null everywhere except for the target state where it is 1, and β is a weight on the reward term.

The corresponding near optimal deterministic policy is obtained through a computationally expensive variation calculus method. The feedback controller, resulting from the coupling of this policy with an optimal state estimator, drives the plant towards the rewarded state. Given that the policy does not take the presence of noise in the model into account, the actions must be computed again at each time step depending on the new state reached by the plant. Overall, generating a trajectory with this method is extremely costly. Hereafter, this controller is called Quasi-Optimal Planning System (QOPS). Indeed, the trajectories are not optimal in the strict sense, given the presence of non-modeled noise.

2.2 Policy Search methods

In the control approach described above, the controls are computed only along the current trajectory of the system. By contrast, in the MDP framework, the choice of an action according to a state is determined by a stationary policy $\pi : \mathcal{X} \to \Pi(\mathcal{U})$, which maps each state to a distribution over actions: $\pi(\boldsymbol{u}_t|\boldsymbol{x}_t) = P(\boldsymbol{u}_t|\boldsymbol{x}_t, \pi)$. In this paper, we consider only deterministic policies written $\boldsymbol{u}_t = \pi(\boldsymbol{x}_t)$ for simplicity. Since the number of states and actions is infinite, one cannot represent exactly a non-trivial policy in this context. This is why we call upon parametric policies.

2.2.1 Parametric policies

A parametric policy spans a family of policies defined by a function $\pi : \Theta \times \mathcal{X} \to \Pi(\mathcal{U})$ chosen *a priori*. Different policies are obtained by spanning a set of vectors Θ. We notate π_θ the policy parametrized by $\boldsymbol{\theta} \in \Theta$ and we simply write it $\boldsymbol{\theta}$ when $\pi_{\boldsymbol{\theta}}$ is used as a parameter.

Given a distribution $\mathcal{P}_0$ of the initial state of a plant, the global performance of the controller can be expressed as the objective function

$$J(\boldsymbol{\theta}) = \mathbb{E}\left[V^{\boldsymbol{\theta}}(\boldsymbol{x}) \mid \boldsymbol{x} \sim \mathcal{P}_0\right]$$

.

The optimal policy is therefore the one that maximizes the objective function. Since we only consider the subset of policies parametrized by a real vector, we can only find the best policy within this subset

$$\boldsymbol{\theta}^* = \text{argmax}_{\boldsymbol{\theta}} \ J(\boldsymbol{\theta})$$

where $\boldsymbol{\theta}^*$ denotes an optimal set of policy parameters. The problem at hand can thus be considered as a continuous and stochastic optimization problem.

2.2.2 Gradient Policy Search methods

A standard way to optimize a parametric policy given an objective function consists in performing a gradient descent over the space of parameters. This class of problems has attracted a lot of attention in RL (see [13] for an overview). The approach that makes the fewest assumption is called the Finite Differences methods. It simply consists in computing the performance of the policy for different values of the parameters and ascending the gradient of the performance through these values. More powerful methods assume that the derivative of the objective function with respect to parameters is known, which makes the computation of the gradient more efficient. This is the case of Episodic REINFORCE [24], Natural Policy Gradient methods [13] or the PoWER algorithm [9]. The latter corresponds to a recent approach of continuous RL problem based on Expectation-Maximization algorithms that, among other interesting properties, can be used to combine nicely RL and some learning from demonstration methods. However, despite convincing results in robotics [9, 10], such methods are complex, sensitive to many hyper-parameters, and still difficult to apply to large continuous state and action problems.

2.2.3 The Cross-Entropy method

The Cross-Entropy method [15] is a general Monte-Carlo approach which can be used for continuous optimization

Algorithm 1 Cross-Entropy Policy Search

```
Require: $(\boldsymbol{\mu}_0, \boldsymbol{\sigma}_0^2)$: initial mean and standard deviation of
  the parameters distribution
  $\rho$: proportion of the best samples to use for the update
  $\bar{\sigma}^2$: additional noise term
  $T$: number of iterations
  $N$: number of parameters samples to draw
  $M$: number of trials for each policy evaluation
  $H$: time horizon of the simulations

  for $t = 1 \cdots T-1$ do
    $\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \cdots, \boldsymbol{\theta}^{(N)} \leftarrow \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\sigma}_t^2)$
    for $i = 1 \cdots N$ do
      Perform $M$ simulations using policy $\pi_{\boldsymbol{\theta}^{(i)}}$ up to $H$
      Compute an estimated performance of policy $\pi_{\boldsymbol{\theta}^{(i)}}$
      $J^{(i)} = \frac{1}{M} \sum_{j=1}^{M} \sum_{h=0}^{H-1} \gamma^h r_{j,h}$ where $r_{j,h}$ is the reward
      at time $h$ for the $j^{th}$ episode
    end for
    Sort the parameter samples $\boldsymbol{\theta}^{(i)}$ according to $J^{(i)}$ and
    compute the set $\mathcal{S}_\rho$ of the $\rho$-best parameter samples
    $\boldsymbol{\mu}_{t+1} \leftarrow \mathrm{mean}(\mathcal{S}_\rho)$
    $\boldsymbol{\sigma}_{t+1}^2 \leftarrow \mathrm{std}(\mathcal{S}_\rho) + \bar{\sigma}^2$
  end for
  return optimized policy parameters $\boldsymbol{\theta} = \boldsymbol{\mu}_T$
```

[16]. As opposed to the gradient descent methods described above, it does not assume that the objective function is differentiable or even continuous.

Given a problem and looking for a solution w that optimizes an objective function S, instead of using a single solution that is updated over iterations, it consists in improving a distribution $f \in \mathcal{F}$ over solutions $\boldsymbol{w} \in \mathcal{W}$ in terms of their value.

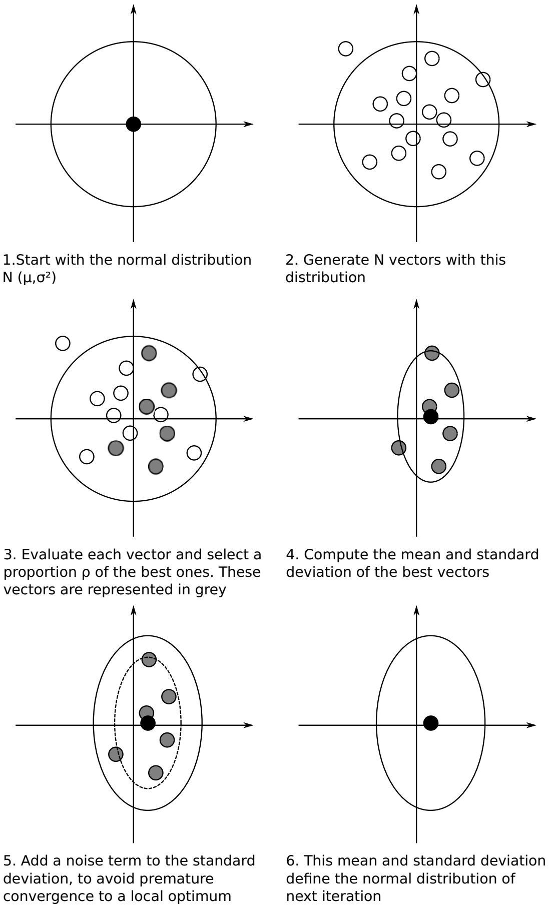
At iteration t, it computes a new distribution f_{t+1} by bringing the current distribution f_t as close as possible to the distribution $g_t = \{\forall \boldsymbol{w} | S(\boldsymbol{w}) > \gamma_t\}$ of solutions which evaluation is over a threshold γ_t. The distance between distributions is expressed in terms of cross-entropy i.e., it minimizes $H(f_{t+1}, g_t) = \mathbb{E}_{f_{t+1}}[-\log g_t]$. This computation is performed by evaluating N samples from f_t, which provides an estimation of $H(f_{t+1}, g_t)$ by importance sampling. Each threshold γ_t is computed using these samples, by selecting a proportion $\rho \in]0,1[$ of the best samples.

By choosing the normal law as family of distributions $\mathcal{F}$, we get f_{t+1} simply by computing the mean $\boldsymbol{\mu}_t$ and the standard deviation $\boldsymbol{\sigma}_t^2$ from the ρ-best samples of f_t. Moreover, [6] suggests to add a small noise $\bar{\sigma}^2$ to the standard deviation to avoid premature convergence. This general method is illustrated in Fig. 1 (borrowed from [21]).

2.2.4 Cross-Entropy Policy Search

We apply this technique to policy search, using the policy parameters $\boldsymbol{\theta}$ as the solutions w and the performance criterion J as the objective function S. The complete algorithm, named Cross-Entropy Policy Search (CEPS) is described in Algorithm 1.

This approach was already suggested in [20] to learn an optimal policy for the game Tetris but, despite its surprising efficiency and simplicity compared to competitive direct Policy Search in RL, it has not been applied yet to continuous state and action problems.



1.Start with the normal distribution N (μ,σ²)

2. Generate N vectors with this distribution

3. Evaluate each vector and select a proportion ρ of the best ones. These vectors are represented in grey

4. Compute the mean and standard deviation of the best vectors

5. Add a noise term to the standard deviation, to avoid premature convergence to a local optimum

6. This mean and standard deviation define the normal distribution of next iteration

Figure 1: Schematic view of the Cross-Entropy method.

2.3 XCSF

The XCS Learning Classifier System [3, 25] is an efficient accuracy-based LCS designed to solve classification problems and sequential decision problems. The eXtended Classifier System for Function approximation (XCSF) algorithm [26, 27] is an evolution of XCS towards function approximation.

As any LCS, XCSF manages a population of rules, called *classifiers*. These classifiers contain a condition part and a prediction part. In XCSF, the condition part defines the region of validity of a local model whereas the prediction part contains the local model itself. XCSF is a generic framework that can use different kinds of prediction models (linear, quadratic, etc.) and can pave the input space with different families of regions (Gaussian, hyper-rectangular, etc.). In the context of this paper, we only consider the case of linear prediction models and Gaussian regions.

A classifier defines a domain $\phi_i(\boldsymbol{z})$ and uses a corresponding linear model β_i to predict a local output vector $\boldsymbol{y}_i$ relative to an input vector $\boldsymbol{x}_i$. The linear model is updated using the Recursive Least Squares (RLS) algorithm, the incremental version of the Least Squares method.

The classifiers in XCSF form a population P that clusters the condition space into a set of overlapping prediction models. XCSF uses only a subset of the classifiers to generate an approximation. Indeed, at each learning iteration, XCSF generates a match set M that contains all reliable classifiers

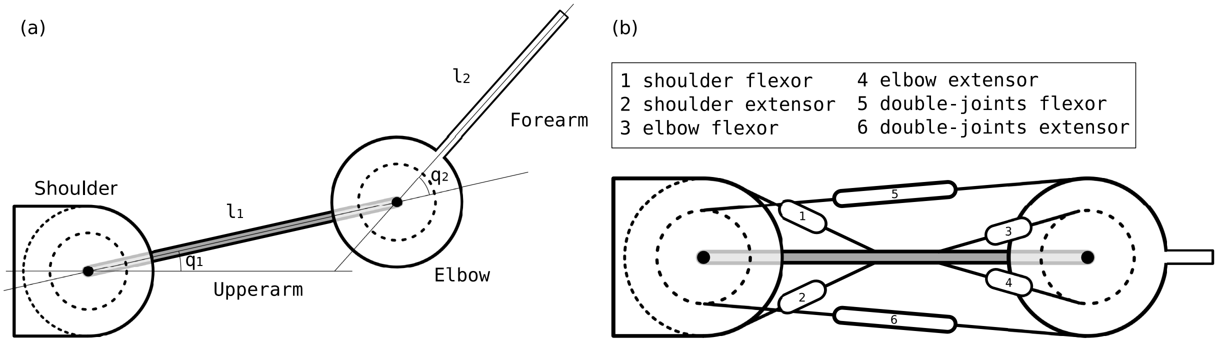


Figure 2: Arm model. (a) Schematic view of the arm mechanics. (b) Schematic view of the muscular actuation of the arm, where each number represents a muscle whose name is in the box.

in the population P whose condition space $\mathcal{Z}$ matches the input data $\boldsymbol{z}$ i.e., for which $\phi_i(\boldsymbol{z})$ is above a threshold ϕ_0[1].

In XCSF, the output $\hat{\boldsymbol{y}}$ is given for a $(\boldsymbol{x}, \boldsymbol{z})$ pair as the sum of the linear models of each matching classifier i weighted by its fitness F_i

$$\hat{\boldsymbol{y}}(\boldsymbol{x},\boldsymbol{z}) = \frac{1}{F(\boldsymbol{z})}\sum_{i=1}^{n_M} F_i(\boldsymbol{z})\,\hat{\boldsymbol{y}}_i(\boldsymbol{x}) \qquad (2)$$

where $F(\boldsymbol{z}) = \sum_{i=1}^{n_M} F_i(\boldsymbol{z})$ and n_M is the number of classifiers in the match set M. In all other respect, the mechanism that drive the evolution of the population of classifiers are directly inherited from XCS. In sum, XCSF is designed to evolve partitions in which linear approximations are maximally accurate. A more complete description of XCSF can be found in [4, 5].

2.4 General architecture

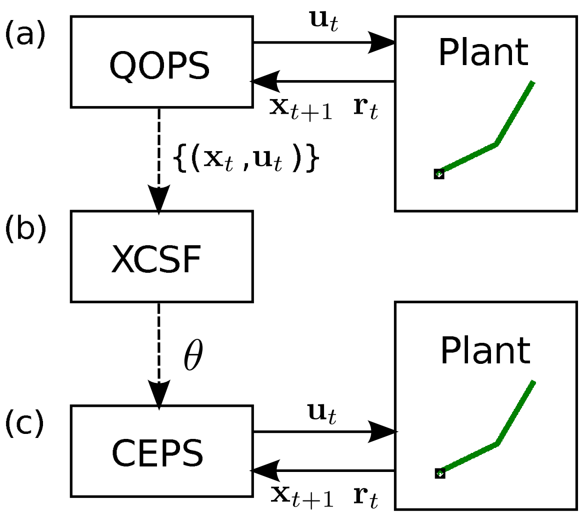

(a)
QOPS
$\mathbf{u}_t$
Plant
$\mathbf{x}_{t+1}$ $\mathbf{r}_t$
$\{(\mathbf{x}_t, \mathbf{u}_t)\}$
(b)
XCSF
θ
Plant
(c)
CEPS
$\mathbf{u}_t$
$\mathbf{x}_{t+1}$ $\mathbf{r}_t$

Figure 3: General architecture of the experiments.

Fig. 3 describes the global architecture used for our experiments. A set of near-optimal state-action trajectories generated by the QOPS (Fig. 3 (a)) provide supervised learning samples, using the state of the plant as the condition and prediction space input and the action as the output on which regression is performed.

By feeding XCSF with such samples (Fig. 3 (b)), we generate an action for any state within the range of the population of classifiers. Using a default action $\boldsymbol{u}_{default}$ for states that are not covered by the population (that is for which XCSF does not predict anything), we get a mapping from states to actions i.e., a deterministic policy. We call it the "XCSF policy". It is parametric since each classifier has parameters in its condition and prediction parts.

This XCSF policy is then adapted using CEPS (Fig. 3 (c)). We generate a set of XCSF policies by slightly changing the parameters of the initial policy. Each policy in this set corresponds to a dot in Fig. 1. In practice, we only tune the prediction part parameters of the classifiers: the shape and fitness of each classifier are not adapted. In other words, policy parameters (i.e., CEPS samples) $\boldsymbol{\theta}$ are large vectors composed of the weights of each local model.

3. EXPERIMENTAL DESIGN

We now illustrate the use of XCSF for learning to control an arm to reach a given point with its end-effector.

3.1 Arm model

The plant is a two degrees-of-freedom planar arm controlled by 6 muscles, illustrated in Fig. 2. It is a simplified version of the one described in [12]. All the angles are expressed in radians. The equations of its dynamics can be found in Appendix A.

The state-space consists of the target articular position $\boldsymbol{q}^*$, the current articular position $\boldsymbol{q}$ of the arm and its current articular speed $\dot{\boldsymbol{q}}$. The state $\boldsymbol{s} = (\boldsymbol{q}^*, \boldsymbol{q}, \dot{\boldsymbol{q}})$ has a total of 6 dimensions. The initial state is defined by $\boldsymbol{q} = [0.5, 0.59]$, null speed and a variable target position. The positions are bounded to represent the reachable space of a standard human arm, with $q_1 \in [-0.6, 2.6]$ and $q_2 \in [-0.2, 3.0]$, as shown in Fig. 4. The action-space consists of an activation signal for each muscle, which also makes a total of 6 dimensions. The action is perturbed by a multiplicative noise of standard deviation $\sigma_u^2 = .02$. The simulator uses the Euler method with a time step of $\Delta t = 2$ ms and is stopped after $H = 1000$ time steps. Reaching is successful when the Euclidean distance from the end-effector to the target is under $d = 0.01m$. There is no explicit condition on velocity, but the method used in [14] makes so that the velocity is very low close to the goal. The reward function (see Equation

[1] This threshold is named θ_m in [4]

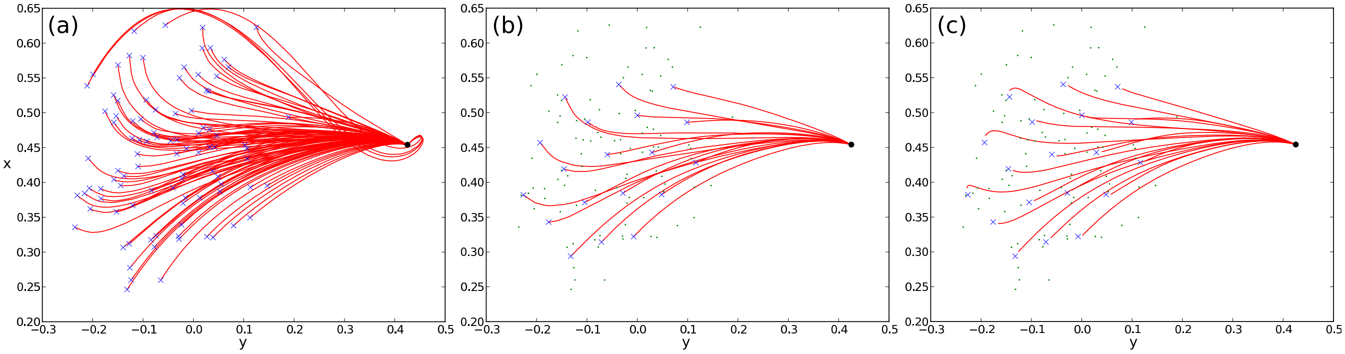


Figure 5: Trajectories obtained with QOPS for the learning targets (a), testing targets (b) and of the XCSF-based policy for the testing targets (c). The starting position is represented by a dot. The targets are represented by a cross. In (b) and (c), the dots represents the learning targets.

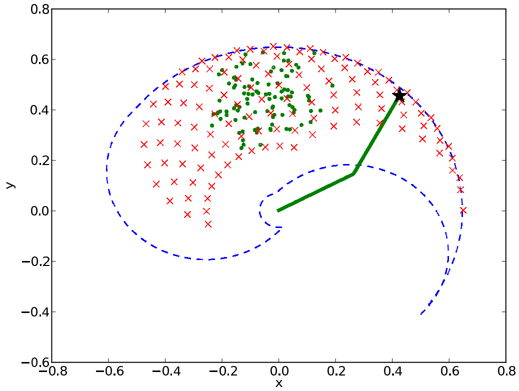


Figure 4: The arm workspace. The reachable space is delimited by a spiral-shaped envelope. The two segments of the arm are represented by two bold lines. The initial configuration is the one represented. Learning targets are indicated by dots and testing targets by crosses.

(1)) is parametrized by $\alpha = 200\Delta t$ and $\beta = 40\gamma$ with the discount factor γ set to $e^{-\Delta t}$.

3.2 Experimental setup

The QOPS is implemented in C++ with the Eigen library. It writes the computed trajectories into text files. We use the JavaXCSF [19] implementation of XCSF, and the main code for the experiments as well as the CEPS algorithm are also implemented in Java. For plotting the results, we use Python with the Matplotlib library [8]. The experiments are run on a Intel Core 2 Duo E8400 @ 3 GHz with 4 GB RAM. We perform two experiments.

3.2.1 Generalization with XCSF

First, we create two sets of target positions, one for learning and the other for testing. 100 learning targets are drawn according to a normal law centered on the center of the reachable space (mean ($x = -0.059, y = 0.44$) and standard deviation 0.1). The testing targets correspond to an 11×11 grid covering the reachable space. All trajectories in all experiments are starting from the same point in the upper-right part of the reachable space (see Fig. 4).

The QOPS is used to generate one trajectory for each learning target and each testing target. Then, XCSF learns to predict the actions of the QOPS for the learning targets, using the generated trajectories, and is tested as a policy on the testing targets. The condition and prediction space contain states $\boldsymbol{s}$. The performance and the trajectories obtained with the XCSF policy are compared to those generated by the QOPS, to see how good the XCSF policy generalizes to a broader range of targets.

3.2.2 Adaptation with CEPS

Second, the XCSF policy obtained in the previous experiment is optimized using CEPS. Each policy is evaluated on the full set of testing targets: the global performance of a policy is the mean of the performance for each testing target. We only run one episode to evaluate the performance for a target.

3.3 Parameters

XCSF is tuned as follows. The number of iterations is set to 100,000 and the maximum size of the population to 6,400. The input are normalized: the target and current positions are bounded by the reachable space and the speed is bounded by $[-100, +100]$ $rad.s^{-1}$. The default action $\boldsymbol{u}_{default}$ is set to a vector of zeros i.e., no muscular activation. After tuning empirically the parameters, the learning rate α (named `beta` in JavaXCSF) is set to 1.0, and compaction is disabled[2]. Multithreading is disabled to improve reproducibility.

For CEPS, the number of iterations T is set to 40, each consisting in the evaluation of $N = 100$ policies. Each target being evaluated only once, the number of trials M is 121 i.e., the number of testing targets. The proportion of selected episodes ρ is set to 0.01: only the best over the 100 policies is selected (and thus becomes the new mean of the policy parameters distribution). An additional noise $\bar{\sigma}^2 = 0.01$ is used as the new variance $\boldsymbol{\sigma}^2$ of the policy parameters distribution, which is also initialized to 0.01.

4. RESULTS

In this section, we present the results obtained with the experimental setup described in the previous section. In the first part, we study whether the policy learned with XCSF is

[2] `startCompaction=resetRLSPredictionsAfterSteps=2`

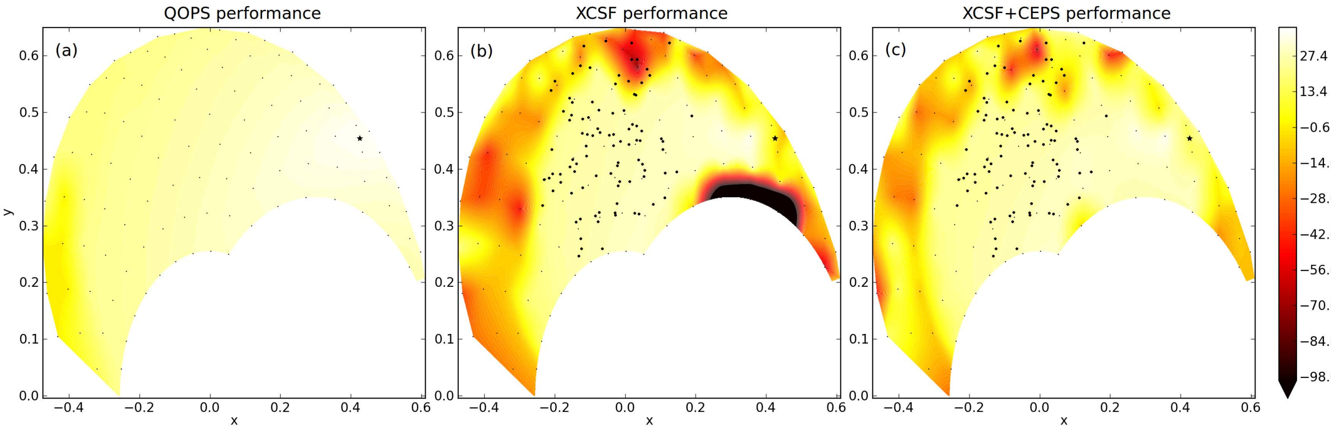


Figure 6: Performance of the QOPS (a), the XCSF policy (b) and the XCSF policy optimized by CEPS (c) given the target position, obtained by interpolating the performances for the testing targets (small dots). The learning target positions are indicated by big dots, and the starting position by a star. The performance is computed according to Equation (1) and represented as a color according to the right-hand side scale.

similar to the one obtained with the optimal control process for the same targets and how well XCSF generalizes over different targets. In the second part, we study whether CEPS is able to improve the performance of the learned policy where generalization resulted in a poor performance.

4.1 Performance of XCSF policy and generalization

The average running time to get one trajectory from the QOPS is ∼ 10 min. From the XCSF policy, it is ∼ 2.0 s. Optimizing the XCSF policy with CEPS does not increase the time necessary to generate one trajectory.

Fig. 5(a) shows the set of trajectories used for learning the XCSF policy. Fig. 5(b) shows a set of trajectories obtained from the QOPS for a few targets selected in the testing set. Fig. 5(c) shows the trajectories generated with the XCSF policy for the same targets. The general shape of these trajectories is similar, even though the QOPS generates slightly more curved trajectories for the furthest targets.

Fig. 6(a) shows the performance of the QOPS as a function of the target position, obtained by interpolating the performance of the QOPS trajectories for the testing set of targets. Fig. 6(b) shows the performance of one representative XCSF policy, which is very close to the QOPS performance for targets located near the learning set. One can also see that the performance for more distant targets is generally lower, but is equivalent to that of the QOPS on some targets. The discontinuity in the performance of the upper part of the learning targets region can be explained by the bifurcation in the learning trajectories (see Fig. 5(a)).

4.2 Performance after CEPS optimization

Fig. 6(c) shows the performance of the XCSF policy after applying CEPS to optimize its parameters.

One can see that the performance has been globally improved over the whole reachable space. Actually, this global improvement is always statistically significant since CEPS cannot decrease the global performance. In particular, the region around ($x = 0.4, y = 0.35$) where the performance of the XCSF policy was very poor has been brought much closer to the optimal performance. The results in Table 1 confirm quantitatively this visual feeling. By comparing column 2 with column 4, one can see that the large difference in performance over the whole reachable space between the QOPS and the XCSF policy is mainly due to small areas where the performance is very low. Indeed, when considering only points in the regions where all learning points are lying (*reduced set* in Table 1), one can see that the performance is closer to the optimum. But, interestingly, this performance decreases in this region after applying CEPS. These points are discussed below.

Table 1: Performance (see Equation (1)) on testing targets. The first two columns show the performance and standard deviation over targets as well as the percentage of performance with respect to QOPS for the full workspace. The last two columns show the same for a reduced space (crosses on Fig. 5 (b) and (c)).

	full set	%QOPS	reduced set	%QOPS
QOPS	25.85 ± 7.5	100%	27.27 ± 1.8	100%
XCSF	-3.59 ± 45.8	12%	26.95 ± 1.9	99%
+CEPS	9.32 ± 22.0	32%	24.06 ± 10.6	88%

5. DISCUSSION

The first and most obvious outcome of using the XCSF policy as controller is that it is about 300 times faster than the QOPS. As a result, the model may account for the real-time execution of a movement, which is far from being the case with the QOPS. In this context, the generalization capability of XCSF matters a lot, too. Indeed, training XCSF with QOPS would be very long if many trajectories were necessary to obtain a satisfactory performance of the learned policy. As shown in Fig. 6(b), a few trajectories are enough. However, as highlighted in Section 4.2, even if the performance is satisfactory in a region that is larger than the one where XCSF was trained, it can drop dramatically outside the training region and its variance is high.

The fact that the performance is lower and that the variance is high outside the training region is not surprising. Indeed, XCSF has to extrapolate from its local knowledge, which may be difficult in regions where the behavior of the plant differs from what was learned in the training region. In that respect, using CEPS in order to further improve the performance has interesting effects. First, to perform parameter optimization, CEPS generates additional trajectories in the whole reachable space, thus it incorporates some knowledge of the behavior of the plant that was not accessible to XCSF during training. It does so based on the XCSF policy, thus the additional cost is limited with respect to using further QOPS trajectories. However, since CEPS optimizes a global criterion over the whole space, one can also see that the resulting averaging of performance induces a local loss of performance in the training region where XCSF produced a very efficient policy.

To circumvent this effect, we should train our system using one target at a time and design a more local criterion that would improve the performance only with respect to the current target. Furthermore, it would be interesting to derive a more local method that would act preferentially on the classifiers that most need it, without disrupting the whole population. The most appealing way of performing this new derivation would be to introduce the stochastic optimization process within the mechanisms of XCSF. This latter option is in our immediate research agenda.

6. CONCLUSION

In this paper, we have shown that XCSF could be used to learn a cost-efficient continuous state and action policy based on a costly quasi-optimal planning system. The resulting controller is fast and the generalization capability of XCSF makes the learning process reasonably easy in practice. Furthermore, we have shown that, on top of XCSF, one could use a stochastic optimization algorithm to further improve the policy, endowing the global system with RL capabilities that are hard to obtain in a continuous state and action context. However, from our empirical results, one can see that the global system is not close enough to the performance one gets with the QOPS. In order to further improve it, the most immediate options are the following:

- One may consider other prediction models (e.g. quadratic instead of linear) or definition of regions (e.g. hyper-rectangular instead of Gaussian).
- One may apply CEPS to a larger set of parameters from the XCSF population. For instance, the condition part could be exploited i.e., the classifiers shape could also be optimized. Furthermore, the population itself could be improved by additional parameters i.e., adding or deleting classifiers could be considered as changing parameters. The counterpart of optimizing over a larger set of parameters is that optimization would take more time.
- Like genetic algorithms, CEPS does not use the gradient of the performance with respect to the policy parameters to improve them. However, since the XCSF model is differentiable, we could apply some sophisticated gradient-based RL methods such as [10, 13]. Based on the results of [21], though they benefit from more relevant information, it is not clear that such methods would necessarily perform better.
- Finally, to get a significant improvement, CEPS needs to generate a lot of trajectories. Instead of actually generating them which is time consuming when the plant is complex, one may rather try to learn a model of the plant, as presented in [17, 18] for instance, and use this model to improve the policy with virtual experiments based on that learnt model.

Acknowledgements

This work was supported by the Ambient Assisted Living Joint Programme of the European Union and the National Innovation Office (DOMEO-AAL-2008-1-159), more at http://www.aal-domeo.eu.

7. REFERENCES

[1] E. Bernadó-Mansilla, X. Llorà, and J. M. Garrel. XCS and GALE : a comparative study of two Learning Classifer Systems with six other learning algorithms on classification tasks. In P.-L. Lanzi, W. Stolzmann, and S. W. Wilson, editors, *Proceedings of the fourth international workshop on Learning Classifer Systems*, 2001.

[2] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA, 1995.

[3] M. V. Butz, D. Goldberg, and P. Lanzi. Computational Complexity of the XCS Classifier System. *Foundations of Learning Classifier Systems*, 51:91–125, 2005.

[4] M. V. Butz and O. Herbort. Context-dependent predictions and cognitive arm control with XCSF. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1357–1364. ACM New York, NY, USA, 2008.

[5] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson. Toward a theory of generalization and learning in xcs. *IEEE Transactions on Evolutionary Computation*, 8(1):28–46, 2004.

[6] P.-T. de Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein. A Tutorial on the Cross-Entropy Method. *Annals of Operations Research*, 134(1):19–67, 2005.

[7] E. Guigon, P. Baraduc, and M. Desmurget. Optimality, stochasticity and variability in motor behavior. *Journal of Computational Neuroscience*, 24(1):57–68, 2008.

[8] J. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, pages 90–95, 2007.

[9] J. Kober and J. Peters. Policy search for motor primitives in robotics. *Advances in Neural Information Processing Systems (NIPS)*, pages 1–8, 2008.

[10] P. Kormushev, S. Calinon, and D. G. Caldwell. Robot Motor Skill Coordination with EM-based Reinforcement Learning. In *Proc. of IEEE/RSJ Intl Conf. on Intelligent Robots and Systems (IROS-2010)*, 2010.

[11] P.-L. Lanzi. Learning Classifier Systems from a Reinforcement Learning Perspective. *Journal of Soft Computing*, 6(3-4):162–170, 2002.

[12] W. Li. *Optimal control for biological movement systems*. PhD thesis, University of California, San Diego, 2006.

[13] J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks :*

the official journal of the International Neural Network Society, 21(4):682–97, 2008.

[14] L. Rigoux, O. Sigaud, A. Terekhov, and E. Guigon. Movement duration as an emergent property of reward directed motor control. In *Proceedings of the Annual Symposium Advances in Computational Motor Control*, 2010.

[15] R. Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

[16] R. Y. Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and Computing in Applied Probability*, 1(2):127–190, 1999.

[17] C. Salaün, V. Padois, and O. Sigaud. Control of redundant robots using learned models: an operational space control approach. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 878–885, 2009.

[18] P. Stalph, J. Rubinsztajn, O. Sigaud, and M. Butz. A Comparative Study: Function Approximation with LWPR and XCSF. In *Proceedings of the 13th International Workshop on Advances in Learning Classifier Systems*, 2010.

[19] P. O. Stalph and M. V. Butz. Documentation of JavaXCSF. Technical report, COBOSLAB, 2009.

[20] I. Szita and A. Lörincz. Learning Tetris using the noisy cross-entropy method. *Neural computation*, 18(12):2936–41, 2006.

[21] C. Thiéry. *Itération sur les Politiques Optimiste et Apprentissage du Jeu de Tetris*. PhD thesis, Université Henri Poincaré - Nancy 1, 2010.

[22] E. Todorov and M. I. Jordan. Optimal feedback control as a theory of motor coordination. *Nature Neurosciences*, 5(11):1226–1235, 2002.

[23] T. H. Tran, C. Sanza, Y. Duthen, and D. T. Nguyen. XCSF with computed continuous action. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO'07)*, pages 1861–1869. ACM New York, NY, USA, 2007.

[24] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

[25] S. W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[26] S. W. Wilson. Function approximation with a classifier system. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 974–981, San Francisco, California, USA, 2001. Morgan Kaufmann.

[27] S. W. Wilson. Classifiers that Approximate Functions. *Natural Computing*, 1(2-3):211–234, 2002.

[28] S. W. Wilson. Three architectures for continuous action. Technical report, No. 2006019, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 2006.

APPENDIX

A. ARM DYNAMICS

Table 2: Parameters of the arm model.

m_i	mass of segment i (kg)
l_i	length of segment i (m)
s_i	inertia of segment i ($kg.m^2$)
d_i	distance between the center of segment i and its center of mass (m)
κ	Heaviside filter parameter
$\boldsymbol{A}$	moment arm matrix
$\boldsymbol{T}$	muscular tension
$\boldsymbol{M}$	inertia matrix
$\boldsymbol{J}$	Jacobian matrix
$\boldsymbol{C}$	Coriolis force
$\boldsymbol{\tau}$	segments torque ($N.m$)
$\boldsymbol{B}$	damping
$\boldsymbol{u}$	raw muscular activation (action)
σ_u^2	multiplicative muscular noise
$\tilde{\boldsymbol{u}}$	filtered noisy muscular activation
$\boldsymbol{q}^*$	target articular position (rad)
$\boldsymbol{q}$	current articular position (rad)
$\dot{\boldsymbol{q}}$	current articular speed ($rad.s^{-1}$)

Table 2 gives a nomenclature of the parameters and variables of the arm model. The arm dynamics is computed as follows, given the current state $\boldsymbol{x}_t = (\boldsymbol{q}^*, \boldsymbol{q}, \dot{\boldsymbol{q}})$ and action $\boldsymbol{u}_t$:

$$m_1 = 1.4,\ m_2 = 1.1,\ l_1 = .30,\ l_2 = .35$$
$$s_1 = .11,\ s_2 = .16,\ d_1 = .025,\ d_2 = .045,\ \kappa = 25$$
$$k_1 = d_1 + d_2 + m_2 l_1^2,\ k_2 = m_2 l_1 s_2,\ k_3 = d_2$$

$$\boldsymbol{A} = \begin{bmatrix} .04 & -.04 & 0 & 0 & .028 & -.035 \\ 0 & 0 & .025 & -.025 & .028 & -0.35 \end{bmatrix}^\top$$

$$\boldsymbol{T} = \begin{bmatrix} 700 & 382 & 572 & 445 & 159 & 318 \end{bmatrix}$$

$$\boldsymbol{M} = \begin{bmatrix} k_1 + 2k_2\cos(q_2) & k_3 + k_2\cos(q_2) \\ k_3 + k_2\cos(q_2) & k_3 \end{bmatrix}$$

$$\boldsymbol{J} = \begin{bmatrix} -l_1\sin(q_1) - l_2\sin(q_1+q_2) & -l_2\sin(q_1+q_2) \\ l_1\cos(q_1) + l_2\cos(q_1+q_2) & l_2\cos(q_1+q_2) \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} -\dot{q}_2(2\dot{q}_1 + \dot{q}_2)k_2\sin(q_2) & \dot{q}_1^2 k_2 \sin(q_2) \end{bmatrix}$$

$$\boldsymbol{B} = \begin{bmatrix} .05 & .025 \\ .025 & .05 \end{bmatrix} \dot{\boldsymbol{q}}_t$$

$$\tilde{\boldsymbol{u}} = \log(\exp(\kappa \times \boldsymbol{u}_t \times (1 + \mathcal{N}(0, \boldsymbol{I}\sigma_u^2))) + 1)/\kappa$$

$$\boldsymbol{\tau} = \boldsymbol{A}^\top(\boldsymbol{T} \times \tilde{\boldsymbol{u}})$$

$$\partial \boldsymbol{q}^*/\partial t = 0$$
$$\partial \dot{\boldsymbol{q}}/\partial t = \boldsymbol{M}^{-1}(\boldsymbol{\tau} - \boldsymbol{C} - \boldsymbol{B}) \qquad \boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \partial \boldsymbol{x}_t/\partial t \times \Delta t$$
$$\partial \boldsymbol{q}/\partial t = \dot{\boldsymbol{q}}$$

where × refers to the element-wise multiplication and $\boldsymbol{I}$ is a 6×6 identity matrix.