Bootstrapping Q-Learning for Robotics from Neuro-Evolution Results

Matthieu Zimmer, Stephane Doncieux

Bootstrapping Q-Learning for Robotics from Neuro-Evolution Results

Matthieu Zimmer, Stephane Doncieux, *Member, IEEE*

***Abstract*—Reinforcement learning problems are hard to solve in a robotics context as classical algorithms rely on discrete representations of actions and states, but in robotics both are continuous. A discrete set of actions and states can be defined, but it requires an expertise that may not be available, in particular in open environments. It is proposed to define a process to make a robot build its own representation for a reinforcement learning algorithm. The principle is to first use a direct policy search in the sensori-motor space, i.e. with no predefined discrete sets of states nor actions, and then extract from the corresponding learning traces discrete actions and identify the relevant dimensions of the state to estimate the value function. Once this is done, the robot can apply reinforcement learning (1) to be more robust to new domains and, if required, (2) to learn faster than a direct policy search. This approach allows to take the best of both worlds: first learning in a continuous space to avoid the need of a specific representation, but at a price of a long learning process and a poor generalization, and then learning with an adapted representation to be faster and more robust.**

***Index Terms*—Developmental robotics, reinforcement learning, continuous environment, representation redescription**

I. Introduction

A large number of reinforcement learning algorithms explore a set of states and actions to find the sequence of actions maximizing a long-term reward [1]. The definition of the set of actions and states has a critical importance. It needs to cover all relevant behaviours, but a too large set of actions and states will drastically decrease learning efficiency: this is the curse of dimensionality. In a robotic context, choosing actions and states is particularly difficult. The movements and most of the perceptions of a robot are continuous. A rough discretization results in too many actions and states. The design of smaller sets of relevant actions and states requires a deep insight on robot features and on the behaviours expected to solve the task. An adapted representation with appropriate approximation functions and a good exploitation of prior knowledge are, in practice, the key to success [2]. A learning algorithm is expected to let the robot discover its own abilities and to reduce the burden of robot behaviour implementation. The design of appropriate actions and states is then a strong limitation to the application of such algorithms to a robotics context. Reinforcement learning algorithms in continuous spaces do not require this design step, but they require exploring large and continuous search spaces and typically rely on demonstrations by an expert to bootstrap, or on non-random initial policies, or else on dedicated policy representations [2,3,4].

The first motivation of this work is to reduce the expertise required and to let the robot discover on its own the skills it needs to fulfil its mission. Besides simplifying the robot behavior design process, it should make the robot more adaptive to situations unknown during robot programming phase [5]. The second motivation is to let the robot build upon its experience in a developmental robotics perspective [6], to have the robot acquire progressively better generalization, and to learn faster as it acquires more experience. To satisfy these motivations, it is proposed to use several learning algorithms requiring different representations and different levels of a priori knowledge about the robot and the task. The hypothesis explored here is that *a robot with adapted representations (1) is more robust to new situations and (2) can adapt its behaviour faster*. The goal is then to allow the robot to change the representation it uses and the corresponding learning algorithm. The robot is provided with two learning algorithms. A first learning algorithm is slow but task agnostic. It relies on low-level sensori-motor values, i.e. raw sensor and effector values. A second one (Q-learning with discrete actions and continuous states) is expected to be faster, but needs high level representations, typically actions and states that are adapted to the robot task. The goal of providing both strategies and not only the fastest one is to bootstrap the second strategy with the knowledge extracted from the results of the first one (Figure 1). The slow and agnostic algorithm is then used first and followed by an analysis step that extracts from the corresponding learning traces the information relevant for the second learning algorithm. The proposed approach has then three different steps: (1) slow learning with a task agnostic algorithm, (2) analysis of the learning traces and extraction of actions and sensors to take into account and (3) exploitation of the representation-specific learning algorithm when facing a new situation. The generation of the first successful behaviours and the representation building process are not performed simultaneously, they are separated into two completely different processes.

The agnostic algorithm is a neuroevolution algorithm, i.e. an evolutionary algorithm that generates both the structure and parameters of a neural network [7,8]. Up to millions of neural networks are explored to reach a result of interest. The first step of the proposed approach takes time, but has the advantage of requiring very little a priori knowledge about the task. This algorithm generates a neural network that is well

M. Zimmer is with Sorbonne Universités, UPMC Univ Paris 06, CNRS, Institut des Systèmes Intelligents et de Robotique (ISIR), Equipe AMAC, 4 place Jussieu, 75252 PARIS cedex 05, France and with Université de Loraine, LORIA, UMR 7503, F-54000, Nancy, France

S. Doncieux is with Sorbonne Universités, UPMC Univ Paris 06, CNRS, Institut des Systèmes Intelligents et de Robotique (ISIR), Equipe AMAC, 4 place Jussieu, 75252 PARIS cedex 05, France

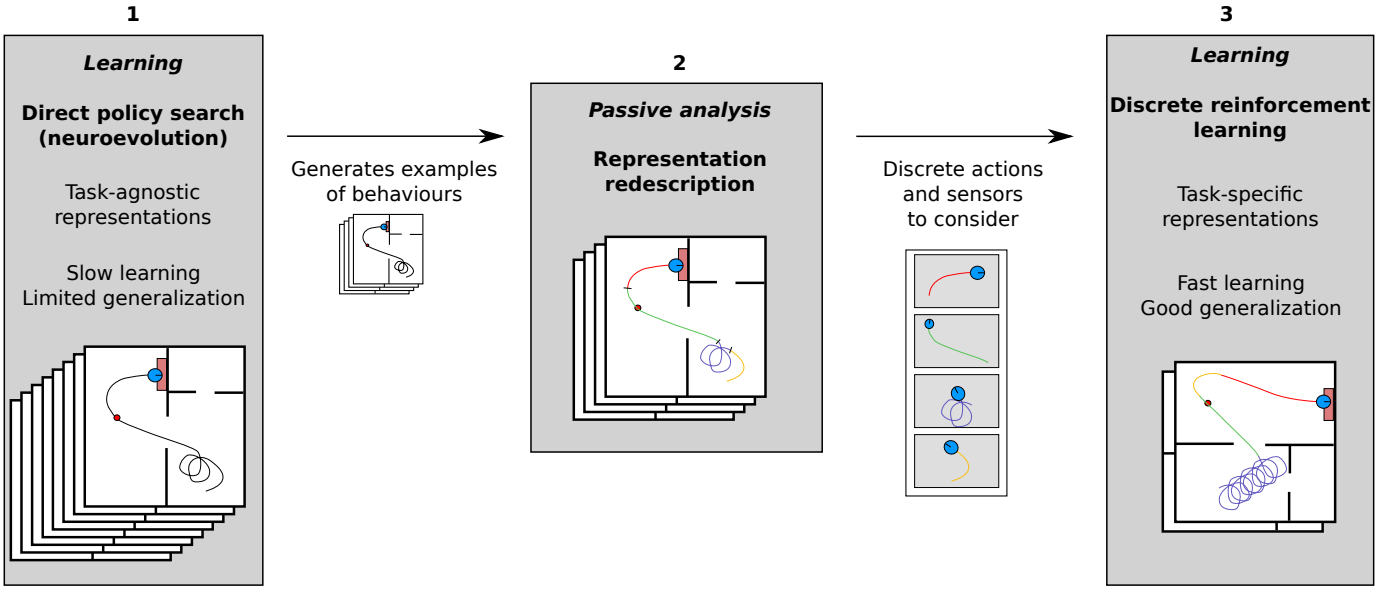

Fig. 1. Overview of the proposed approach.

suited for the domain used to evaluate the potential solutions during the learning experiment. When the robot faces a new domain, there is no guarantee that this neural network will generate the expected behaviour — this is the generalization problem [9]. If the neural network fails to exhibit the expected behaviour, the robot will have to learn again from scratch. The solutions generated on the first task nonetheless contain information about how to solve it. It contains many examples of the orders to send to motors in response to a particular set of sensor values. An analysis and extraction step is then proposed to extract relevant data so that when the robot faces a new situation, it relies on these data and uses another reinforcement learning algorithm to find out how to maximize the long-term reward. The space to explore being much smaller than for neuroevolution, the time required to learn is expected to be much shorter. The proposed approach is tested on two different tasks: a ball collecting task [10] and a box pushing task [11].

II. Related work

Robot positions, perceptions and actions are continuous. Furthermore robot state estimation is prone to error because of noise or perceptual aliasing and actions may fail. Applying a reinforcement learning algorithm to a robot is then not straightforward. Some examples of discrete reinforcement learning with predefined actions and states in a robotics context will be briefly reviewed in this section, followed by a review of approaches in which actions and states are built or updated using the experience acquired previously.

A. Predefined representations

In some problems, a simple and straightforward discretization gives satisfying results. Policies for controlling a ball in a beam [12], a crawler robot [13] or a one degree of freedom (DOF) ball-in-the-cup [14] have thus been learned with a discrete reinforcement learning algorithm. It is possible because of a limited number of actions, two for [12], four for [13] and five for [14].

Tasks involving a higher number of possible action values can be solved, but at the price of a more careful design of the actions available for the reinforcement learning algorithm. Willgoss and Iqbal solved an obstacle avoidance task with reinforcement learning by predefining actions like 'forward', 'forward left', 'backward', etc [15]. Konidaris et al. provided navigation actions (reach a particular position among a set of task-relevant positions) and robot arm movement actions in navigation tasks requiring to manipulate levers and buttons [16]. Several participants of the Robocup Soccer competition have applied reinforcement learning. Kwok and Fox [17] have used it for the robot to decide where to point its video camera when kicking a ball into the opponent's goal: the robot had to decide what to look for at a particular moment (ball, goal, opponents, landmarks, etc) as its limited field of view did not allow it to have a complete view of the scene. They have used a reinforcement learning algorithm to decide it with one 'looking at' action for each possible point of interest. Several works of the literature [18,19] used sets of macro actions to apply reinforcement learning algorithms to various RoboCup domains. Depending on the considered soccer behavior they wanted their robot to exhibit, Riedmiller et al. provided either a rough discretization of motor commands (resulting in 76 different actions), or a set of possible motor increase or decrease commands for a set of five well chosen commands for the three motors of the robot [20].

A first conclusion from the review of these examples is that it is possible to make a robot learn with a reinforcement learning algorithm exploiting a set of discrete actions. The difficulty with such an approach is actually to design the set of actions [2]. Extending the learning process to build this representation on the fly while the system is learning or developing would make learning algorithms much easier to

use in a robotics context.

B. Adapting representations

In a life-long learning perspective [21], it makes sense to extract knowledge from the acquired experience in order to improve future problem resolution. This is the main motivation of transfer learning, which is a widely studied field in machine learning in general [22] and in reinforcement learning in particular [23]. The principle of transfer learning is to rely on the knowledge acquired while solving a source task to solve a target task more efficiently. There are many different kinds of transfer learning approaches in reinforcement learning, for instance transfer of the value function [24] or through reward shaping [25,26]. In the following, we focus on the transfer of knowledge through policies and the way they are represented.

Options are an extension of Markov Decision Processes to introduce sub-goals and the corresponding sequences of actions as new actions available for learning [27]. Konidaris and Barto proposed an algorithm to discover them automatically [28]. Furthermore, options learned in a particular setup can be transferred to a new task or domain if they are defined in a problem independent space [29]. Koga et al. have proposed an approach to learn abstract policies in order to increase the generalization ability [30,31]. Riano and McGinnity proposed to represent sequences of actions with a finite state automaton and to explore its structure with evolutionary algorithms [32]. The advantage of this approach lies mainly in the versatility of evolutionary algorithms that allows both to design the finite state automaton and to tune actions on the fly or even create new ones. All of these approaches require providing a set of primitive actions to bootstrap the system. Once primitive actions are available, they can discover how to combine them to build more abstract actions, but none of those works addresses the question of bootstrapping the process and discovering primitive actions.

Neumann et al. propose to learn complex motion from simpler motion templates [33]. Starting from given motion templates with some initial and relevant parameter values (although not optimal), a reinforcement learning algorithm searches for combinations of these templates together with their parameters. This approach does not require primitive actions, but primitive motion templates need to be defined. It goes further than the previously mentioned approaches with respect to bootstrapping, as a single motion template may result in different primitive actions depending on their parametrization, but it still requires providing significant prior knowledge on the task, a prerequisite that this work tries to avoid.

Mugan et al. proposed QLAP, Qualitative Learner of Action and Perception, a complete framework to bootstrap a cognitive system [11]. Dynamic Bayesian Networks (DBNs) are built to predict events with an adaptive discretization whose goal is to make models more reliable. The DBNs are then converted into plans that are used to build actions. Exploration starts from random motion and is then driven by a motivation inspired from artificial curiosity [34,35]. Kompella et al. similarly proposed CCSA, Continual Curiosity driven Skill Acquisition, an approach to learn abstractions that consists in exploring, collecting observations and discretizing it to augment the agent state-space [36,37]. Both approaches can then bootstrap with no primitive actions. It has then the same goal as the approach proposed here. QLAP relies on an initial set of random motor commands and tries to build the representation on the fly while it is learning and trying to maximize the reward. CCSA starts from at least one exploratory behaviour that is supposed to be given. This behaviour can be considered as a primitive action. The difference with respect to previously mentioned approaches, is that CCSA can create other primitive actions.

All the work reviewed in this section either relies on random actions or on dedicated primitives to generate the data that the developmental system will use to build states and actions. Avoiding to rely on predefined primitives is the main motivation of this work. Furthermore, random actions may not be enough to generate data of interest. In the case of interaction with objects, for instance, random motor actions leads, unsurprisingly, to very few interactions if the robot has enough degrees-of-freedom [38], and may thus not provide the system with relevant data to model object interaction. This limitation has motivated the development of the approach proposed here. Contrary to what is done in QLAP and CCSA, in this work the representation is not built while learning, but after having succeeded with a task-agnostic algorithm. The task of building the representation, i.e. actions and states, is then decomposed in three different and clearly separated steps: (1) obtaining behaviours maximizing the reward, (2) analysing them and building the representation and (3) learning in new contexts while exploiting the representation. The advantages and drawbacks of each approach together with their complementarity will be discussed in Section VIII.

III. Background

Reinforcement learning (RL) [1,39] is a framework that models sequential decision problems where an agent has to learn to make better decisions by maximizing a reward informing it about the quality of its previous actions.

The underlying formalism of RL algorithms is *Markov Decision Processes* (MDP). A MDP is formally defined as a tuple $\langle S, A, T, R\rangle$ where S is a set of states, A a set of actions, $T : S \times A \times S \rightarrow [0, 1]$ are transition probabilities between states ($T(s, a, s') = p(s'|a, s)$ is the probability of reaching state s' from state s after executing action a) and $R : S \times A \rightarrow \mathbb{R}$ is a reward signal. A *policy* $\pi : S \rightarrow A$ is a mapping from states to actions (the action to be taken in each state) and encodes how the agent behaves. An *optimal* policy π^* is a policy maximizing the expected long-term reward:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t \times R(s_t, \pi_t(s_t))\Big], \qquad (1)$$

where γ is a discount factor. In MDPs, T and R are given. An optimal policy π^* can be computed using dynamic programming (e.g., value iteration, policy iteration) [40] for instance.

In RL, the agent does not know the transition probabilities T and the reward function R. *Model-based* methods aim at learning T and R to enable planning and *model-free* methods

only estimate expected rewards. In this work, *model-free* methods are considered. They can be categorized into three approaches : (1) *critic-only* methods aim at learning value functions and deduce a policy from them, (2) *actor-only* methods attempt to learn directly the policy function, and (3) *actor-critic* methods intend to learn both a value function and a policy. *Critic-only* methods (Section III-A) are very difficult to apply to continuous actions and rather need a discrete set of actions, making them task-specific. They will be used during the second learning phase, after the creation of a set of discrete actions and the autonomous design of a state estimator. *Actor-only* methods [4,41] can deal with continuous state and action spaces, but they are slow. Such a method will be used as a task-agnostic algorithm during the first learning phase (Section III-B2). *Actor-critic* methods [42,43] can also deal with continuous state and action spaces but not always with a better data efficiency [44]. The *actor-only* method used here is neuroevolution (see Section III-B).

A. Reinforcement learning with value functions

The function Q estimates the long-term reward to be expected when an action a is chosen in state s ($Q : S \times A \rightarrow \mathbb{R}$):

$$Q_\pi(s,a) = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_t = s, a_t = a\Big]$$

For the choice of an action, we will consider an ϵ-greedy strategy, that exploits most of the time and performs a random exploration move with a small probability, i.e., $\pi(s) = \arg\max_a Q(s,a)$ with probability $1-\epsilon$ and $\pi(s)$ is a random action uniformly drawn in A with probability ϵ.

RL with Q values stored in a table suffers from the lack of generalization: what is learned in a state does not impact the knowledge about the value of similar states. A solution to this problem is to use an approximation function to provide a means for both efficient computation and support for generalization. In these approaches, Q values are approximated by a function:

$$\psi_\theta : S \times A \rightarrow \mathbb{R},$$

parametrized by a vector θ whose values are updated whenever reward is received ($Q(s,a) \approx \psi_\theta(s,a)$). This allows to represent Q-values for continuous domains. Moreover, this allows generalizing what is learned in a state to another similar state (notice that ψ_θ can take many forms compatible with supervised learning methods: neural networks, linear regression, mixture of Gaussian, etc). In this work, we will rely on neural networks [45,46,47], where each action is associated to a different neural network. In this case, the inputs of the neural network are the components of the state, the single output is the Q value associated to this action, and finally θ are the connection weights. In Q-learning [1], the goal is to train the neural network to approximate:

$$Q(s_t,a_t) = R(s_t,a_t) + \gamma \max_{a' \in A} Q(s_{t+1},a'). \qquad (2)$$

If the temporal extensions of actions are not identical, and the reward function depends only on the state, the algorithm can be extended to [48]:

$$Q(s_t,a) = \Big(\sum_{i=1}^{d_a} \gamma^{i-1} R(s_{t+i})\Big) + \gamma^{d_a} \max_{a' \in A} Q(s_{t+d_a},a'), \quad (3)$$

where d_a is the duration of the action a.

B. Evolutionary Design of Robot's Neuro-controllers

Direct policy search methods optimize policy parameters with respect to the reward function. If the cumulated reward only is taken into account (i.e. intermediate rewards are ignored), it has been shown that direct policy search is equivalent to black-box optimization [49]. Evolutionary algorithms are powerful black-box optimization tools that can be used to optimize policies. The use of evolutionary algorithms to design robot parts is called evolutionary robotics [50]. Neural networks are often used to represent policies for their ability to approximate any kind of function with an arbitrary precision [51]. The use of evolutionary algorithms to design neural networks is named neuroevolution [7]. These algorithms have been used on a large range of robotic tasks [52] that can be grouped in three main categories: locomotion, navigation and foraging tasks. Locomotion tasks typically involve hexapod [53,54] or biped robots [55,56]. If most work are focused on locomotion in a straight line, recent work have defined a control architecture including evolutionary approaches and able to learn to walk in any direction [57], and even to adapt to motor failures [58]. Navigation tasks are performed on a two-wheeled robot equipped with infrared sensors and range from avoiding obstacles [59] to finding the output of a maze [60]. Foraging tasks are also performed on two-wheeled robot and, besides the navigation abilities, require the robot to find objects, pick them up and bring them to a goal location [10].

Evolutionary robotics methods require few information about the task and do not need to define finite sets of discrete states or actions. There is however a price to pay: they need to explore many different policies and are thus in general very slow.

The general principles of evolutionary algorithms are first introduced, the specificities of their application to robotics are then presented and this section finally describes how to use them to generate neural networks.

1) General principles of evolutionary algorithms: Evolutionary algorithms rely on the Darwinian principles of variation and selection. They are iterative, population-based, stochastic algorithms [61]. An initial set of candidate solutions is randomly chosen. Each solution – called a *genotype* – is evaluated from a fitness function $\mathcal{F} : \mathcal{G} \rightarrow \mathbb{R}^{n_o}$. $\mathcal{G}$ is the genotype space. Depending on the needs, it may be a string of binary digits, a vector of real values, a program or a graph, for instance. In this work, genotypes are neural networks, i.e. valued directed graphs. Once each solution is evaluated, a selection process determines the genotypes that will feed the next generation.

New genotypes are copies of their ancestors with some random modifications called mutations. The mutation is a stochastic operator that blindly modifies a genotype. It is the

only search operator we will consider here.[1] New genotypes are also evaluated and selected. This process is repeated iteratively to maximize $\mathcal{F}$. Each of these loops is called a generation.

When $n_o > 1$, the problem is said to be multi-objective. Evolutionary algorithms, because of their population-based feature, are easy to adapt to this kind of problems [62]. Considering separately each objective allows to optimize them in parallel and offers the opportunity to deal with policy learning issues through the definition of new and specific objectives [63]. The Non-dominated Sorting Genetic Algorithm II (NSGA-II) [41], a state-of-the-art multi-objective evolutionary algorithm, will be used in the following. It is an elitist algorithm that divides the current population into different fronts: the first front is the set of non-dominated solutions in the population[2]. The second front is the set of non dominated solutions once the first front has been removed, etc. The highest ranked fronts survive to the next generation and are used to generate new solutions [41].

2) Evolutionary robotics: Evolutionary robotics relies on evolutionary algorithms to design policies [50,64].[3] Evolutionary algorithms are black-box optimization tools, but when used in robotics, it was shown to be interesting to take into account some of its specificities [63].

The main specificity of evolutionary algorithms in this context, is the evaluation step. The genotype describes a robot policy. To evaluate it, the robot is placed in its environment in some initial conditions. The corresponding policy determines the orders to be sent to the motors out of robot perceptions. This sensori-motor loop determines the robot's behaviour for a given time length. The fitness value associated to the genotype is deduced from these observations. Task-oriented objectives will evaluate to what extent the robot fulfils the given task, but it has been shown that it was also important to take into account the behaviour in a task-independent way. Evolutionary algorithms include diversity keeping mechanisms that allow to balance exploration and exploitation. They were shown not to be sufficient in robotics applications, unless they take into account robot behaviours. Several approaches were thus defined to enhance exploration in the behaviour space [63]. Novelty search objectives reward individuals that have a novel behaviour with respect to the current population and an archive of encountered behaviours [56]. In this work, we will rely on behavioural diversity [65]. In this approach, a behavioural diversity objective $\mathcal{BD}(x)$ rewards a genotype with respect to the difference of its behaviour with respect to the behaviour of other individuals in the current population:

$$\mathcal{BD}(x) = \frac{1}{size(P)} \sum_{y \in P} d(x, y),$$

where P is the current population, x and y are genotypes. $d(x, y)$ is a distance in the behavioural space. Different definitions of behaviours have been tested and all revealed to lead to better results than approaches in which no behavioural diversity was used [65]. Behavioural diversity is optimized as a separate objective in a multi-objective EA [65].

Several diversity measures can be used in a same run. It avoids the need to choose a single one and also revealed to speed up convergence [54]. The behavioural diversity objective becomes:

$$\mathcal{BD}_k(x) = \frac{1}{size(P)} \sum_{y \in P} d_k(x, y),$$

where several behavioural distances $d_k(x, y)$ are defined. k is changed after a given number of generations. This approach will be used here and the definition of the behavioural distances used in the following experiments will be described in sections V and VI.

3) Neuroevolution: Evolutionary algorithms offer the possibility to optimize both the structure and the parameters of a neural networks [7]. We have used this possibility here, with an approach called DNN [65], a direct encoding inspired from NEAT. NEAT is a direct encoding that revealed to outperform many other encodings [66]. In NEAT, the mutation operator directly modifies the graph of neurons and connections. A specific mechanism is used to compare neural networks, define a crossover operator and to protect new networks when they appear. DNN is a simplified version of NEAT that do not include this mechanism. It was shown that DNN was actually more efficient than NEAT when associated with behavioural diversity [65]. DNN consists in randomly generating neural networks without hidden nodes. Several mutation operators are defined. A first operator randomly modifies connection weights using the polynomial mutation [62]. Other mutation operators can add a connection between two randomly selected neurons or remove connections. The last mutation operators can add a neuron by splitting an existing connection, or remove a randomly selected neuron with all of its connections [65].

IV. Method

The proposed method makes no assumption about useful actions or sensors. It aims to extract them from preliminary experiments relying on direct policy search using raw sensori-motor values. It is proposed to generate policies using neuroevolution, a task-agnostic direct policy search. Neural networks take sensor values as inputs and send their outputs directly to robot effectors. They do not need finite set of states nor discrete actions. The price to pay is the time required for them to converge: up to a million policies evaluations may be required to converge. Furthermore, there is no guarantee that a policy maximizing the reward in a domain will still maximize it in a new domain. When facing a new situation, the direct policy search algorithm may need to be executed again.

It is proposed to extract from direct policy search learning traces, states and actions to be used with a reinforcement learning algorithm. Future learning episodes rely on it and are expected to converge faster to a policy maximizing the reward.

[1] Another operator exists in evolutionary algorithms literature: the crossover. It is a stochastic operator that builds a new genotype out of several parents. It raises difficult issues when dealing with neural networks, and is then often not used at all in this context.

[2] A solution is non dominated if no solution is at least equal on all objectives and better on at least one objective.

[3] It can also design robot morphology but we will focus here on policy.

The steps of the proposed approach are the following:

1) Learn with direct policy search in a source domain
2) Extract actions and states from the learning traces:
   a) extract a finite set of actions from traces:
      i) split the motor flow
      ii) build linear models of actions
      iii) select the most representative actions with a clustering algorithm
   b) identify relevant sensor inputs for state estimation
   c) learn with Q-learning on the source domain with the extracted action and states (updates the estimator of Q-values)
3) Learn with Q-learning in the target domain.

A. Direct policy search and generation of learning traces

Neuroevolution relies on the principles of variation and selection to generate policies described by neural networks. Many policies are generated and evaluated. These evaluations provide examples of the behaviours that policies may generate, starting from inefficient policies and finishing with efficient ones. A subset of all the evaluations made is selected for actions and state extraction. It should be representative of what a robot can experience. We propose to rely on a sample of evaluations that contains successful individuals as well as inefficient ones. Instead of relying on randomly selected solutions, it is proposed to use the horizon-finite lineage of the best-of-run policy.

In an evolutionary run, a new policy π_{n+1} is generated through the mutation of another policy π_n.[4] π_n is the parent policy of π_{n+1}. The lineage of a solution is the set $\{\pi_0, \pi_1, \ldots, \pi_n\}$ where π_i is the parent of π_{i+1}. π_0 is a randomly generated policy. Some evolutionary algorithms, including NSGA-II, are elitists, which means that the best individuals are systematically part of the next generation. A policy can then be generated at generation 10, stay in the population, and still generate new policies at generation 100 or more. The parent of a policy π may then have been generated many generations before π enters the current population.[5]

As evolution is a stochastic process, each evolutionary run will generate a different best policy. This is the reason why we use the term best-of-run policy.

The horizon-finite lineage of a best-of-run policy contains the range of policies from the H last ancestors to the most efficient observed ones. It has then been selected as the set of behaviours to consider for action and state extraction with horizon $H = 500$.

Learning traces are made up with the sensori-motor flow experienced by the policies in the lineage of the best-of-run policy. Let $\mathcal{O}_i$ be the set of observations (sensor values) made by the robot while it is controlled by policy π_i and let $\mathcal{U}_i$ be the set of effector commands proposed by the policy in response to those observations ($len(\mathcal{O}_i) = len(\mathcal{U}_i) = n_h$, n_h being the evaluation length).

[4]Crossover may also contribute to the generation of new individuals, but this operator is not used here.

[5]As a consequence, π_n is not necessarily generated at generation n.

In the following, the learning traces saved in a direct policy run are the set $\{(\mathcal{O}_0,\mathcal{U}_0),(\mathcal{O}_1,\mathcal{U}_1),\ldots,(\mathcal{O}_n,\mathcal{U}_n)\}$, where $\{\pi_0, \pi_1, \ldots, \pi_n = \tilde{\pi}^*\}$ is the horizon-finite lineage of the best-of-run policy $\tilde{\pi}^*$.

The learning traces are generated with NSGA-II [41] as evolutionary algorithm and DNN as neural network encoding [65] (see section III-B).

B. Discrete actions and continuous commands

The effector commands in the learning traces are continuous: $\mathcal{U}_i = (u_i(0), u_i(1), \ldots, u_i(n_h))$ where $u_i(t) \in \mathbb{R}^{n_e}$, n_e being the number of effectors. It is proposed to extract from these traces the set of discrete actions A required for discrete RL [1]. A mapping between continuous commands and discrete actions then needs to be defined. We have chosen to define actions as follows:

- actions have different temporal extensions. Let us call d_a the duration of action a;
- actions map to linear functions in the effector space.

Let us call Γ_a the mapping between an action a, time and effector commands:

$$\Gamma_a : \{0, 1, \ldots, d_a - 1\} \rightarrow \mathbb{R}^{n_e}.$$

Γ_a is defined as follows:

$$\Gamma_a(t) = \begin{cases} \Delta_1^{(a)}(t) \\ \vdots \\ \Delta_{n_e(t)}^{(a)} \end{cases}, \tag{4}$$

where $\Delta_i^{(a)}(t)$ is the value to send to effector i at time t for the action a. Δ are linear models:

$$\Delta_i^{(a)}(t) = u_i \times t + v_i. \tag{5}$$

The set of parameters $\mathcal{V}_a$ describing Γ_a is then the following:

$$\mathcal{V}_a = \left(\begin{pmatrix} u_1 & v_1 \\ \vdots & \vdots \\ u_{n_e} & v_{n_e} \end{pmatrix}, d_a \right). \tag{6}$$

Figure 2 shows examples of actions associated to two effectors.

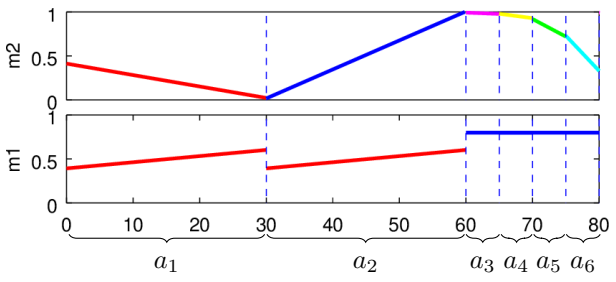

Fig. 2. A sequence of 6 linear actions with different temporal extensions over r=2 effectors $\{m1, m2\}$. $d_{a_1} = d_{a_2} = 30$, $d_{a_3} = d_{a_4} = d_{a_5} = d_{a_6} = 5$.

This formalism could be seen as an SMDP [67], where the terminal conditions of options are based on the time d_a and the internal policy of one option is restricted to a linear model.

We propose to extract $\mathcal{V}_a$ from learning traces.

C. Extracting actions from traces

The learning traces contain the set of effector commands $\mathcal{U}_i$ generated by each policy π_i in the lineage of the best-of-run and in the evaluation conditions used during the neuroevolution search process. Each $\mathcal{U}_i$ is a set of n_h continuous value vectors: the values that were sent to the effectors at each time step. To extract actions from the different $\mathcal{U}_i$, a 2-steps process is defined: a heuristic splits $\mathcal{U}_i$ in many linear actions, then a clustering algorithm extracts the most representative ones.

The proposed splitting heuristic considers the second derivative of effector commands. It binarizes these values: 0 if it is below the mean second derivative, 1 otherwise. The resulting binary sequence is then used to decide where to put splitting points. Each '1' in the sequence is a potential splitting point. In case of two following '1', the point that corresponds to the lowest second-derivative value is changed to '0'. In the case of a sequence of '1' whose length is above two, only the extreme points are kept. All the others are changed to '0'. The '1' in the sequence after these modifications correspond to the splitting points. The idea behind this strategy is to consider as a single action the sequences of effector commands with slow variations, but also action sequences in which there are fast oscillations. Both sequences are replaced by linear actions. It seems natural for the sequences with slow variations. For the sequence with oscillating values, the linear approximation is very far from the initial sequence. Our motivation was to avoid actions lasting only one time step. Our assumption is that the robot actually filters down the variations with the highest frequencies and thus that the error in practice is lower than what it seems. Robots are in general built with this property, notably to avoid fast oscillations that may have a dramatic impact on robot stability because of its inertia.

The algorithm used to determine the splitting points is provided in a python-like pseudo-language:

function SPLIT POINTS($\mathcal{U} \in \mathbb{R}^{n_e \times n_h}$)
  $\mathcal{U}'' \leftarrow$ numerical_second_derivative($\mathcal{U}$)
  $\mathcal{M} \leftarrow mean\_over\_effectors(abs(\mathcal{U}''))$ ▷ $\mathcal{M} \in \mathbb{R}^{n_h}$
  $L \leftarrow mean(\mathcal{M})$ ▷ The limit to overcome
  $P \leftarrow map(lambda\ x : x > L, \mathcal{M})$ ▷ $P \in \{0,1\}^{n_h}$
  $i \leftarrow 0$
  while $i < n_h - 1$ **do**
    $j \leftarrow 1$
    if $P[i] == 1$ **then**
      while $(i + j < n_h)$ and $(P[i + j] == 1)$ **do**
        $j \leftarrow j + 1$
      end while
      if $j == 2$ **then** ▷ Handle (1,1) sequences.
        $k \leftarrow index\_min([\mathcal{M}_i, \mathcal{M}_{i+1}])$
        $P[k] \leftarrow 0$
      else if $j > 2$ **then** ▷ Handle longer sequences.
        $P[i + 1 : i + j - 2] = [0] * (j - 2)$
      end if
    end if
    $i \leftarrow i + j$
  end while
  return P
end function

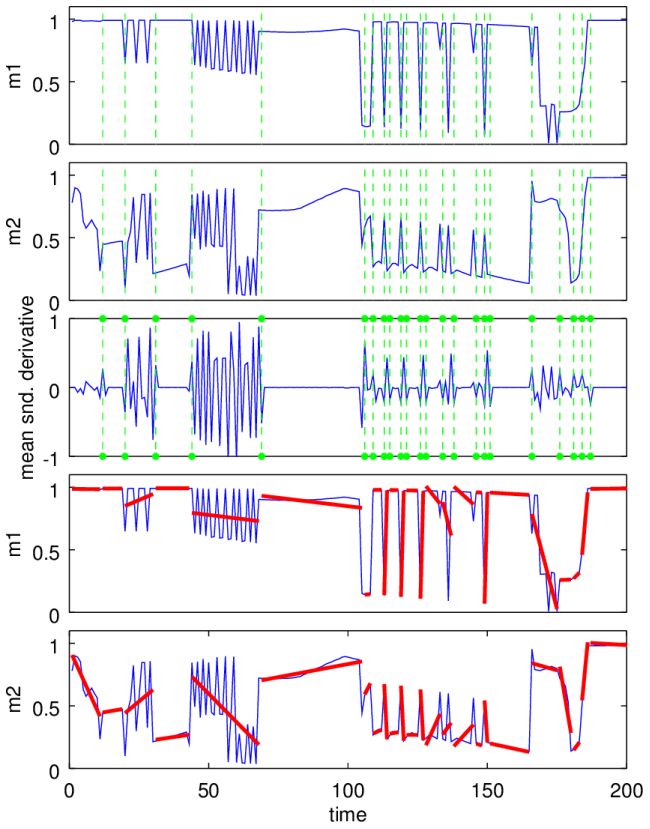

Fig. 3. Example of traces with $n_e = 2$ motors $(m1, m2)$ and $n_h = 200$. The third plot is the mean of the numeric second derivative from the previous traces. The splitting points found by the heuristic can be seen in green. Finally, linear models are learned in red on the 2 last plot.

Once the splitting points are defined, a linear regression is used to learn $\mathcal{V}_a$ (d_a is set to the distance between the splitting points).

Repeating this process on $\{\mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_n\}$ produces a huge number of different actions. A fuzzy c-means clustering is used to extract N_a representatives from this set of actions [68]. The actions extracted are the centroids of the N_a clusters found by the fuzzy c-means algorithm. N_a is a parameter of the proposed approach, its impact on the performance is studied in Section VII-C.

D. Identification of relevant inputs

Perceptual aliasing is a major problem when applying RL to robotics as it prevents an accurate estimation of the state of the system: identical perceptions can be associated to different states and different actions. Consider the case of a robot that is blocked against a wall with a uniform colour. No matter where the robot exactly is, the perceptions are the same. Suppose that the robot must find its way out of the room. If it is blocked against the wall at the left of the door, then it needs to turn right to go out and if it is at the right of the door, then it must turn left. Instantaneous perceptions do not allow to discriminate these two states. A simple solution consists in taking the past into account. As soon as a previous perception differs, the perceptual aliasing phenomenon vanishes. In our robot navigation example, before crashing into the wall, the

robot will probably have different perceptions: it may perceive the door or notice that the left wall is closer or farther, for instance.

Here we assume that taking into account enough previous sensory inputs allows to discriminate states. Considering all past sensory information allows then for sure to determine the robot state, but it makes model training more difficult as much more parameters need to be tuned. There is then a trade-off between the efficiency of the $Q - value$ estimation and the non-ambiguity of the state identification.

We propose to rely on a lasso linear model regression [69] to identify relevant inputs. Identified inputs are then used to learn a non-linear model of the $Q - values$ with a neural network trained with a back-propagation algorithm during the Q-learning process.

$\mathcal{O}^{(j)}$ is the set of all observations in the learning traces in which only the j-th sensor is taken into account:

$$\mathcal{O}^{(j)} = \left\{ o_i^{(j)}(t) | i \in \{0, \ldots, n\}, t \in \{0 \ldots, n_h\} \right\},$$

where $o_i^{(j)}(t)$ is the j-th component of $o_i(t)$ and $o_i(t) \in \mathbb{R}^{n_s}$ the observations experienced by individual i at time t, n_s being the number of sensors.

The lasso method is used to build a model of the transition function T that predicts $o_i^{(j)}(t)$ values out of a set of selected inputs among previous observations: $\mathcal{O}_{i,\mu}(t-1) = (o_i(t-\mu-1), o_i(t-\mu), \ldots, o_i(t-1))$. $\mu = 8$ was empirically chosen as higher values involved too many computations. The resulting set of weights is $W = \{w^{(0)}, w^{(1)}, \ldots, w^{(n_s)}\}$, where $w^{(j)}$ are the weights to predict $\mathcal{O}^{(j)}$ ($w^{(j)}$ is a $n_s \times \mu$ matrix). What matters here is whether a value is null or not, its exact value is not used. $w^{(j)}[k,l] = 0$ means that the k-th input of $o_{l\%n_h}(l/n_h)$ can be neglected when trying to predict the next value of the j-th observation.

Let call $w_{bin}^{(j)}$ the set:

$$w_{bin}^{(j)} = \Big\{ 1 \text{ if } w^{(j)}[k,l] \neq 0,\ 0 \text{ otherwise}$$

$$\text{for all } k \in \{0, \ldots, n_s\}, l \in \{0, \ldots, n * n_h\} \Big\},$$

with n the number of individuals in the learning traces and n_h the evaluation length.

There is no reason that $w_{bin}^{(j_1)} = w_{bin}^{(j_2)}$ for any $j_1 \neq j_2$. We propose then to aggregate these different values using a majority vote: the inputs that are kept are those for which $w_{bin}^{(j)} = 1$ in more than half of the j values, i.e. inputs kept are those that are considered as relevant to predict more than half of the observations.

E. Q-learning

Q-values are computed by neural networks that takes as inputs the observations considered as relevant by the method presented in the previous section. Each action is associated to a specific neural network. Each time an action is executed, the corresponding set of observations and the reward thus obtained are used to train the neural networks with a back-propagation algorithm.

The choice of the action to apply is based on a ϵ-greedy strategy with a gradual decrease of ϵ over the episodes in order to foster exploration:

$$\epsilon(t) = (\epsilon_{max} - \epsilon_{min})\epsilon_{stepness}^{1+t} + \epsilon_{min}.$$

This strategy promotes *exploration* in the early stages, and then *exploitation*.

A second method is used to make network training more efficient: the *replay* of traces of interactions already carried out during learning [45]. To avoid the few good actions to be overwhelmed by the great majority of wrongdoing, the best tracks are replayed multiple times. A trajectory $\mathcal{T}$ of length λ is described as $< o(t), a(t), r(t+1), o(t+1) >_{t \in \{0,\ldots,\lambda-1\}}$. After an episode, a new trajectory is produced and stored with its associated score $\sum_{t=0}^{\lambda-1} \gamma^t r(t)$. The η best trajectories are replayed into the network, after each episode, in a random order. The replay of a single trajectory is done in reverse order from $\lambda - 1$ to 0 to accelerate convergence.

F. Experimental methodology

Two tasks of the literature have been considered: a ball collecting task [54] and a box pushing task [11]. For each task, two different sets of domains are defined: a source set and a target set.

The source set is used for the initial neuroevolution runs. These runs generate the learning traces out of which the states and relevant inputs are extracted. Once actions are extracted and relevant inputs are identified, neural networks, estimating the $Q - values$, are trained on the source set to check the Q-learning performance and to prepare it for tests on the target set.

The target set is used for two different goals:

- to compare the generalization ability of the neuro-controllers and of the policy learned with Q-learning (how do these policies perform without a new learning phase);
- to compare their learning speed (when learning occurs, how fast is it on the new task).

Two different configurations of neuroevolution are considered when comparing the learning speeds on the target task: (1) neuroevolution starting from randomly generated solutions and (2) neuroevolution starting from the Pareto front generated on the source setup (transfer of the best neuro-controllers).

V. Ball collecting task

A. Experimental setup

In this task, a mobile robot has to move around in its environment and collect balls before putting them into a basket. The robot does not know where the balls and the basket are, but it has sensors that can tell whether they are in front of it (front left and front right) or not. The environment contains walls that are obstacles for both robot motion and perception. The robot needs then to explore its environment to find the balls and the basket. A switch is present in the environment to make new balls appear. The environment is continuous: the robot, the balls and the basket can be at any position (except within walls).

1) Robot and environment: The robot has thirteen different sensors : three wall distance sensors, two bumpers, two ball detection sensors, two switch detection sensors, two basket detection sensors, one "carrying ball" sensor and one "switch pulled" sensor that tells the robot if it has already pulled the switch or not. It has three effectors : two wheel motors, and one crane motor to collect balls or activate the switch.

2) Neuroevolution: The same setup as Doncieux and Mouret was used [54]. NSGA-II algorithm is used with the following objectives:

1. number of collected balls
2. dynamic behavioural diversity (see section III-B2) with the following behavioural similarity measures:
   - adhoc: euclidean distance between the end position of the balls;
   - hamming: hamming distance between the discretized values of the sensors and effectors experienced by the robot;
   - entropy: euclidean distance between the vectors of entropies of sensors and effectors;
   - trajectory: edit distance between the discretized trajectories.

The behavioural similarity measure used to compute the behaviour diversity is also randomly changed every 50 generations.

3) Q-learning: The reward function used is the following:

$$R(s) = \begin{cases} 100 & \text{if a ball just dropped in the basket} \\ 0 & \text{otherwise} \end{cases} . \quad (7)$$

4) Domains: The source set is made with one environment and two ball positions. The target set is made with two new environments, each coming with three different robot and object positions (see Figure 4). The source set is then made up with two different domains and the target set with six different domains.

In the source setup, the robot can start from two starting positions (two different domains), and can bring back four balls in each. Thus, the maximum number of collected balls is eight. In the target set, there are six different domains with four balls in each. The maximum score is then twenty four.

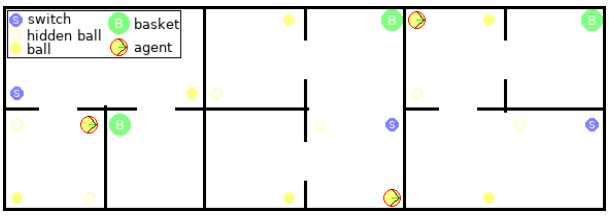

Fig. 4. Different environments A, B, C (wall positions) used in the ball collecting task. There are three different positions for each object (robot, balls, switch, basket) and for each environment. The robot has to bring back the maximum amount of yellow balls in the green basket. He can also touch the purple switch button to make other balls available.

B. Results

1) Neuroevolution results: 30 different runs have been performed with a population size of 128 and for 1,000 generations. 10 runs have been done with A environment, 10 with B and 10 with C. For one run, the time required is around 24 hours of simulation on a 2.2 GHz and 16 CPU cores computer.

The median reaches the maximum score after 287 generations, the lower quartile reaches it after 586 generations and the upper quartile after 198 generations (each generation corresponds to 128 evaluations of policies, i.e. 128 episodes, see Figure 6, ER curve). Figure 5 shows an example of a behaviour generated by neuroevolution.

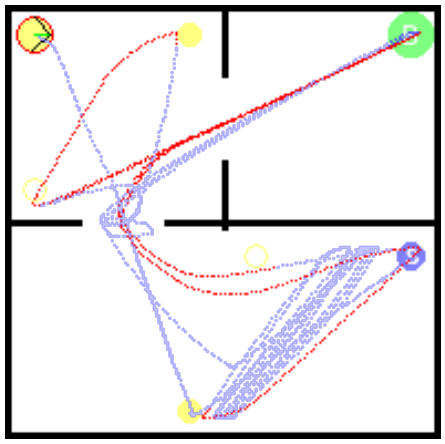

Fig. 5. Example of behaviour of a best individual generated with ER. The trajectory is plotted in red when the robot carries a ball and in blue otherwise.

2) Action and state extraction: There are on average 487 individuals in the filiation of the best individual and thus 487 elements in the learning traces. The median number of extracted actions before clustering is 384,740 (the lower quartile is 246,406, the upper quartile 799,439 and the mean 505,560). The clustering process extracts 17 actions out of it. After clustering, the median temporal extension is 9 (lower quartile 7, upper 18). To extract sensors and actions, around 3 hours is required.

Q-learning is given 5,000 episodes to learn on the source set (Figure 6). An example of generated behaviour is shown in Figure 7.

The median reaches the best performance after 481 episodes (around 3 hours of simulation for one run).

3) Testing and learning on new domains: The generalization ability of policies trained on the source domain set is evaluated on the target domain set. Their performance is compared to that of Q-learning with the extracted actions (Figure 8). Without further training, the median performance reached by Q-learning is 10, whereas the best policies trained with ER reach a median of 2. The actions extracted from the learning traces provide then a significant generalization ability with respect to the best policies of the learning traces they were extracted from. This result aligns with our hypothesis of the increased robustness of adapted representations to new situations.

Figure 9 shows the learning performance of the different approaches when tested on the target set of domains. All the curves are averaged over 30 runs: one run with the actions and inputs extracted from each of the 30 evolutionary runs for the 'QL' experiments, one run starting with the Pareto

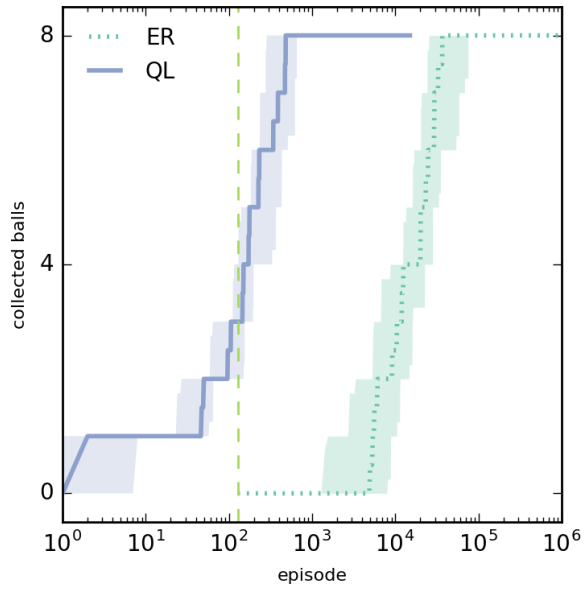

Fig. 6. Median and quartile of ER and Q-learning experiments on source set. The first registered performance of ER begins at the 128th episode because it takes the maximum of the random initial population (population that contains 128 individuals).

front of each of the 30 evolutionary runs for the 'ER+transfer' experiments. Parameters are provided in appendix B.

For Q-learning, the median reaches the maximum score after 8,616 episodes and the upper quartile after 1,013 only. For ER, the median reaches the maximum score after 268,160 episodes and the lower quartile after 1,263,360 (upper 130,304). For ER + transfer, the median reaches the maximum score after 305,152 episodes and the lower quartile after 557,440 (upper 108,160). The median reaches then the maximum performance for Q-learning three times faster than ER approaches. For the upper quartiles, this difference even goes up to two orders of magnitude. This result aligns with our hypothesis of faster learning with adapted representations.

VI. Box pushing task

A. Experimental setup

In the second task, a robotic arm with three degrees of freedom must successfully move a box [11]. It needs to push the box to the left with the end effector centred on the box, which implies to move the arm to a precise position, on the right of the box.

1) Robot and environment: There are three different sizes for the cube and its position is randomly generated. The starting position of the arm is also random. In this environment, the evaluation of a policy does not always produce the same score as the arm and cube positions are random.

The 13 sensors are the same as described by Mugan and Kuipers [11] :

- the position of hand in x,y,z directions (3 sensors);
- the distance between :
  - the top of hand and bottom of block;
  - the bottom of hand and top of block (in both x and z: 2 sensors);
  - the right side of hand and the left side of block;
  - the left side of hand and the right side of block;
  - the left edge of table and the left edge of block;
  - the right edge of table and the right edge of block
  - the top edge of table and the top edge of block.
- The location of hand in x,y directions relative to centre of block (2 sensors).

2) Neuroevolution: NSGA-II algorithm is used in a similar setup to the one used in the ball collecting task. The following objectives are used:

1. number of contacts with the block
2. dynamic behavioural diversity (see section III-B2) with the same behavioural similarity measures than what was used for the ball collecting task. Here, the *adhoc* measure is the euclidean distance between vectors containing the distance travelled by the arm and by the block in each evaluation and in each dimension. *Hamming* and *entropy* are the same and *trajectory* is the edit distance between the discretized trajectories of the end effector.

As for the ball collecting experiment, the behavioural similarity measure is changed every 50 generations [54].

3) Q-learning: The reward function is defined as:

$$R(s) = \begin{cases} 10 & \text{if the block just shifted to the left} \\ 0 & \text{otherwise} \end{cases} . \quad (8)$$

The reward obtained when a block is shifted to the left is lower than that obtained when collecting a ball. This is due to the length of the sequence to generate to get a reward, that is longer in the collect ball experiment. This point is discussed in section VIII.

4) Domains: The source domain set is made up with three domains defined by objects of different sizes. The target domain set is made up with three other domains that are also defined by objects having different new sizes (Figure 11).

B. Results

1) Neuroevolution results: 30 different runs have been performed with a population size of 128 and for 1,000 generations. During an episode, the robot has five tests on three boxes of different size: the optimal score is 15 (1 for each successful test).

The median reaches the maximum score after 390 generations, the lower quartile reaches it after 426 generations and the upper quartile after 333 generations (each generation corresponds to 128 evaluations, see Figure 14, ER curve). An example of behaviour is shown in Figure 12. For one run, the time required is around 12 hours of simulation.

2) Action and state extractions: There are 500 different traces in the filiation of the best individual. The median fitness of those individuals is 10, the lower quartile is 3 and the upper 13. The median number of extracted actions before clustering is 27,998 (lower quartile 26,535, upper quartile 34,634, mean 30,739). The median temporal extension after clustering is 6 (lower quartile 4, upper 18). The Q-learning algorithm is given 5,000 episodes to learn the Q-values (Figure 14). Figure 13 shows an example of how Q-learning used extracted actions: three different actions are used, one that puts the arm at a

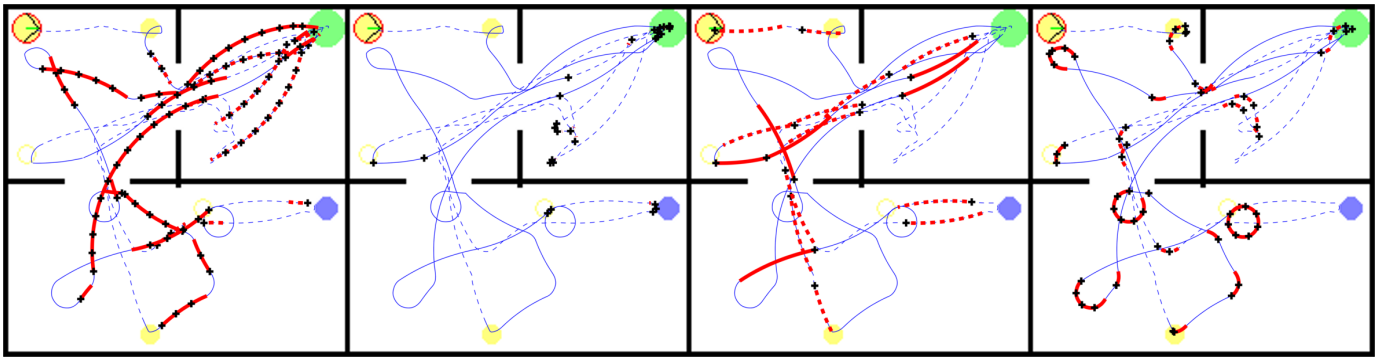

Fig. 7. Example of behaviour of a best individual generated with Q-learning with the actions and sensors extracted from the trace of ER. When the trajectory is full, the agent is carrying a ball. Each figure displays a different action of the same trajectory (the one in red). For this figure, the number of actions is reduced for clarity. In this specific case, the 4 actions can be (a posteriori) described as follows: 1) turn slightly right and keep the ball, 2) do not move and drop the ball, 3) turn slightly left and keep the ball, 4) sharp turn to left and keep the ball.

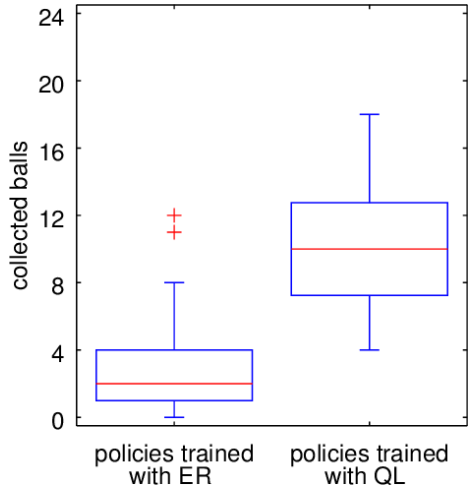

Fig. 8. Generalization ability. Performance on the target domain set of policies trained with ER or Q-learning on the source domain set (p-value : 1.084e-11).

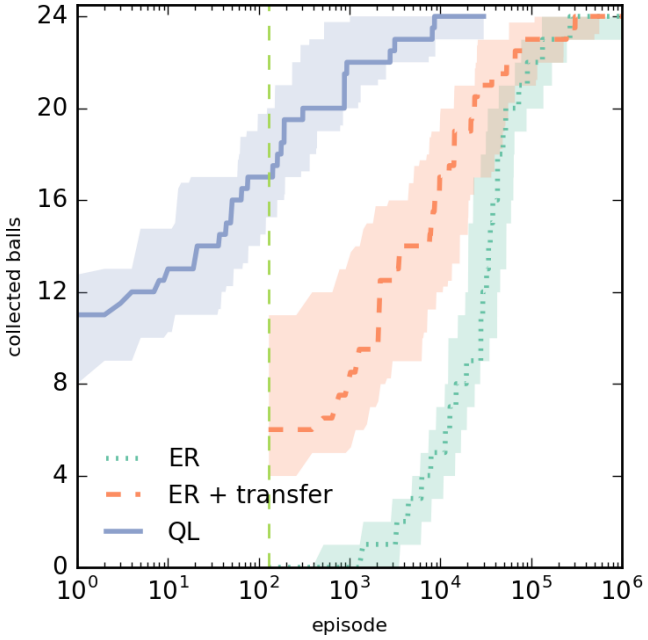

Fig. 9. Medians and quartiles performances on new unknown environments after a learning session. For the ER+transfer, the Pareto front obtained on the previous learning session becomes the new population to optimize, on the contrary, ER starts with a new random population. For the Q-learning strategy, the neural networks are those obtained after the learning session on the source task. First episodes show generalization capacities of Figure 8. ER and ER+transfer curves start after having evaluated the initial population of 128 individuals.

high position and in the lower right corner of the working space, the second action turns and starts the descent and the last one makes a straightforward trajectory towards the box until it pushes it to the left. Around 30 minutes is enough to extract sensors and actions. For one Q-learning run, the time required is around 1 hour.

3) Testing and learning on a new domain: The performance on the target domain set of policies trained on the source domain set are shown in Figure 15. As for the ball collecting task, policies trained with Q-learning generalize better, even if the difference is smaller than for the ball collecting experiment. This result is aligned with our hypothesis of a best generalization ability with adapted representations.

Figure 16 shows the learning performance of the different approaches on the target set of domains. As for the ball collecting experiment, all the curves are averaged over 30 runs: one run with the actions and inputs extracted for each of the 30 evolutionary runs for the 'QL' experiments, one run starting with the Pareto front of each of the 30 evolutionary runs for the 'ER+transfer' experiments. The parameters used are described in appendix B.

For Q-learning, the median reaches the maximum score after 2,119 episodes and the lower quartile after 5,438 (upper 1,084). For ER, the median reaches the maximum score after 243,328 episodes and the upper quartile after 149,760 episodes. For ER + transfer, the median reaches the maximum score after 10,752 episodes and the lower quartile after 37,888 episodes (upper 7,168). These results also confirm our hypothesis of a faster learning with an adapted representation.

VII. Analysis of the Proposed Method

Control experiments have been designed to estimate the contribution of specific aspects of the proposed algorithm

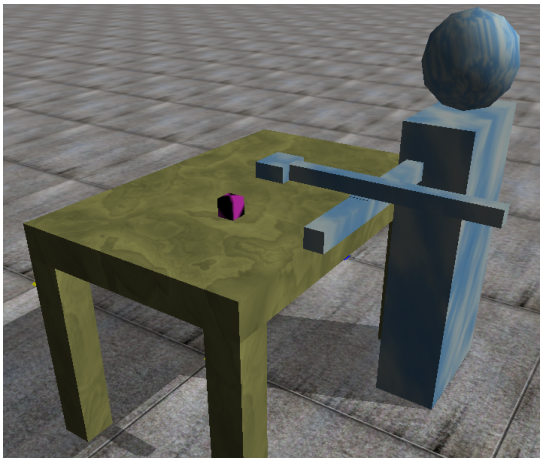

Fig. 10. Continuous environment with realistic physics [70] where a robotic arm must push boxes of different sizes by hitting them on the left.

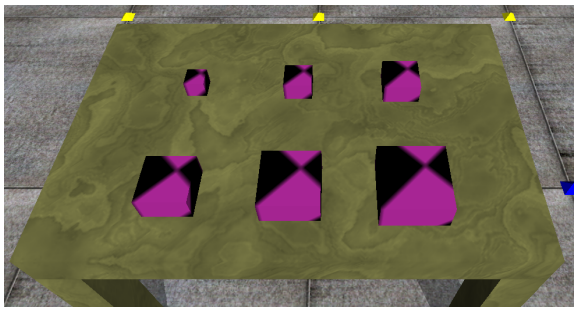

Fig. 11. The objects of the source domain set are in the top row, the objects of the target domain set are in the bottom row.

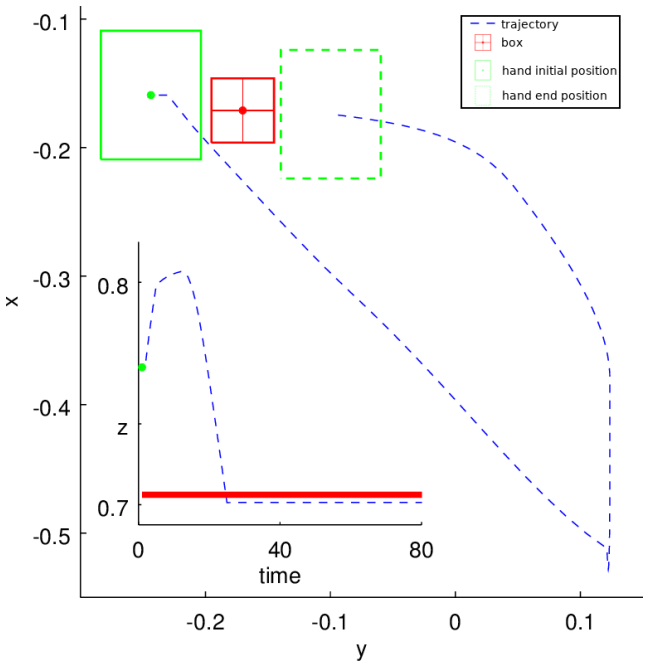

Fig. 12. Example of a best behaviour from ER. (x,y,z) is the position of the end effector. The green point is the starting point position and the red is the box position. The green box represents the end effector size (the dashed box shows the last position of the effector).

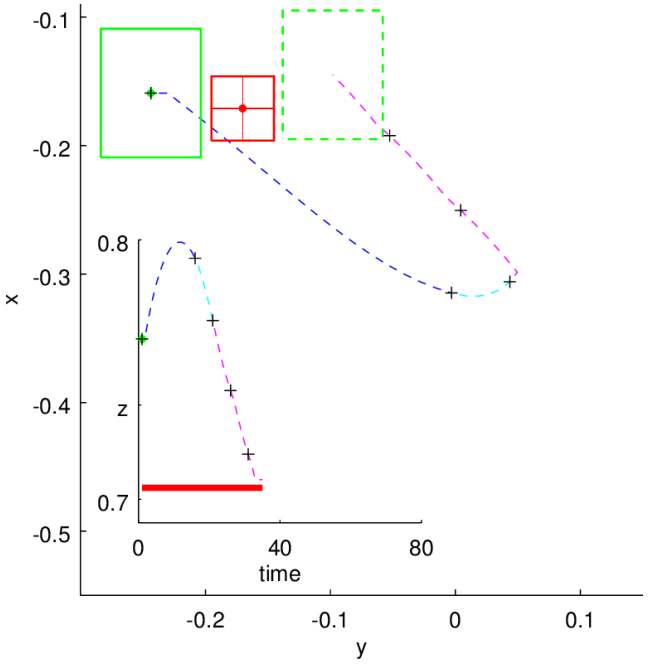

Fig. 13. Example of a converged behaviour from Q-learning. (x,y,z) is the position of the end effector. The green point is the starting position and the red point is the box position.

to the results. Action extraction and input selection are the main parts of the proposed knowledge extraction process. We compare the knowledge they have extracted to randomly generated knowledge or to human designed knowledge. Other experiments have been defined to evaluate the sensitivity of the results to the number of chosen actions.

A. Action extraction

How do extracted actions compare to randomly generated or human designed actions?

Q-learning experiments have been performed on the source domain set, in which the only difference between the setups are the used actions. The set of sensors on which state estimation relies has been chosen by a human expert. It is the same for all setups.

The actions chosen by the human expert for the collect ball environment are the following : 16 actions to move at a different angle while picking and one action to drop the ball. It is pretty smart as the robot will never need to perform an action for "taking a ball when I am near one". It only has to use the moving actions to catch a ball automatically.

In the robotic arm environment, there are 7 human designed actions: one neutral (don't apply any force) and 2 for each degree of freedom in order to be able to reach every possible position. All these actions were designed before looking at the actions extracted by the proposed approach.

For comparisons to be fair, the number of extracted actions N_r and of random actions is the same: 17 for the collect ball experiment and 7 for the box pushing.

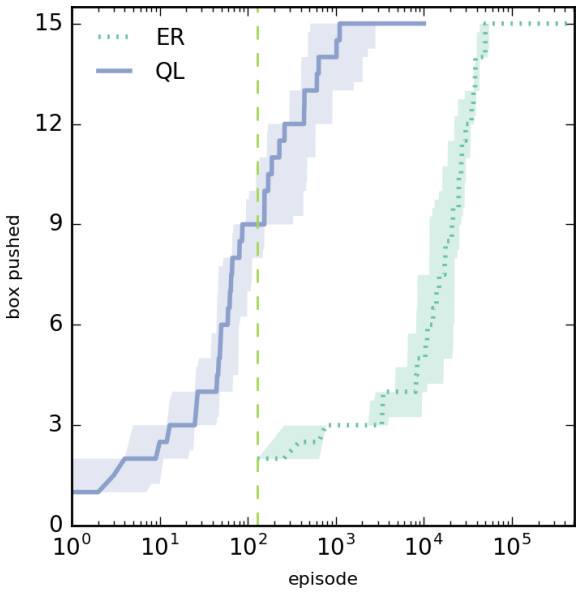

Fig. 14. Median and quartile of ER and Q-learning experiments on source set. The first registered performance of ER begins at the 128th episode because it takes the maximum of the initial random population (population that contains 128 individuals).

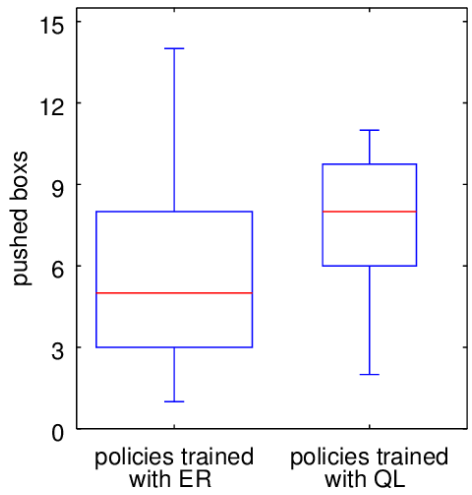

Fig. 15. Generalization ability. Performance on the target domain set of policies trained with ER or Q-learning on the source domain set (t-test p-value=0.04804).

The temporal extension of human designed actions is fixed for each actions ($d_a = 15$ for ball collecting and $d_a = 9$ for robotic arm). For the random action set, each action has a different temporal extension (drawn uniformly from $[5; 25]$ for ball collecting and from $[2; 20]$ for robotic arm).[6]

Figure 17 shows the learning speed of the different alternatives. The median is made over 30 runs with 30 different evolution traces (10 for A, 10 for B and 10 for C).

On the ball collecting task, extracted actions have a performance that is similar to human designed actions. Both setups clearly outperform the random actions setup that seldom converges (see p-values in Appendix A-A).

[6]These intervals lead to the best results.

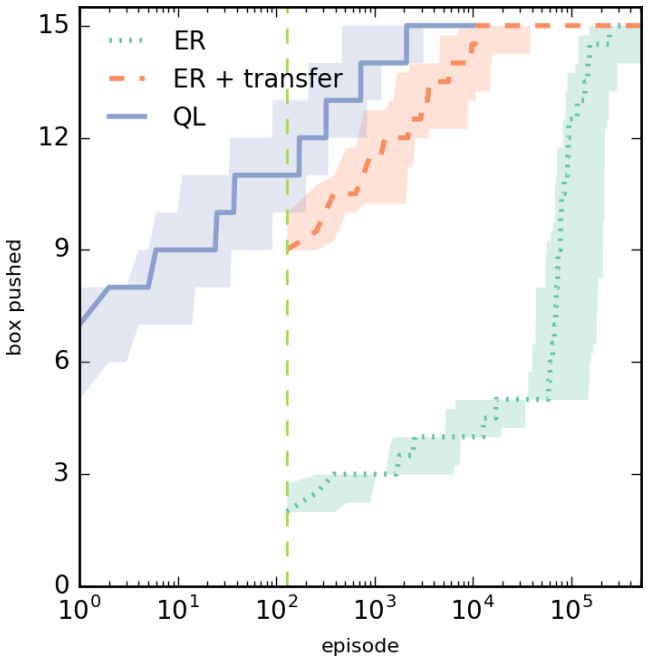

Fig. 16. Medians and quartiles performances on new unknown block size after a learning session. For the ER+transfer, the Pareto front obtained on the previous learning session becomes the new population to optimize, on the contrary, ER starts with a new random population. For the Q-learning strategy, the neural networks are those obtained after the learning session on the source task. First episodes show generalization capacities of Figure 15. ER and ER+transfer curves start after having evaluated the initial population of 128 individuals.

On the box pushing task, all setups require a small number of episodes to maximize the reward, but within an episode, the setup with extracted actions needs less steps to reach this result than the human designed one. Both require less steps to maximize the reward than the random actions setup. The variance of human designed actions is low because it is always the same action set over the 30 runs.

The performance of the extracted actions are clearly better than random actions and are competitive with human designed actions.

B. Choice of sensors used for state estimation

What is the influence of the algorithm used to choose the sensors? To answer this question, we have built a set of dedicated Q-learning experiments:

- *instant*: sensors in $o_i(t)$ are all taken into account. Sensors of previous time steps are ignored;
- *human*: sensors chosen by an expert in the set $\mathcal{O}_{i,\mu}(t)$
- *automatic selection (AS)*: sensors automatically selected with the proposed lasso approach in the set $\mathcal{O}_{i,\mu}(t)$
- *random*: sensors randomly chosen
- *rand +*: sensors randomly chosen, but their number correspond to what the *AS* setup has found.

For the *random* setup, the number of sensors is first drawn from a uniform distribution in the range $[1; 156]$ ($156 = 13 \times 12$, i.e. 12 times steps of the 13 sensors) and the sensors are then randomly chosen.

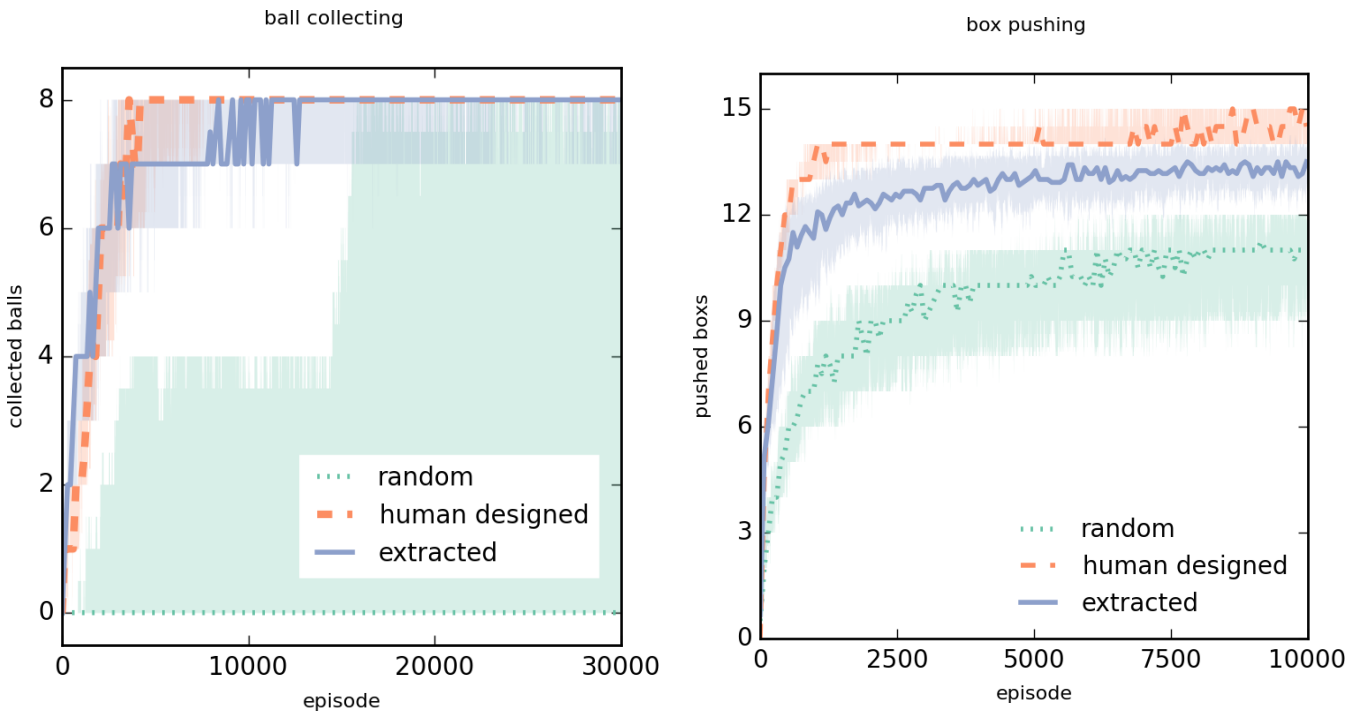

Fig. 17. Median and quartile of performance in the collecting ball environment during Q-learning learning with 3 different set of actions : human designed, extracted from evolution learning traces and randomly designed.

In these experiments the actions are fixed: these are the human designed actions described in the previous section. The performance after 10,000 learning episodes on the ball collecting task and 5,000 on the box pushing task, have been plotted on Figure 18. The number of inputs taken into account is plotted under the performance.

The 'instant' setup leads to the poorest performance on the ball collecting task, thus demonstrating the perceptual aliasing problem in this task. Its performance is maximal for the box pushing task, what suggests, unsurprisingly, the absence of perceptual aliasing in this task. The number of sensors used is minimal for this setup.

The number of input neurons selected by the AS strategy is two times lower with the box pushing than with the collect ball. This shows that the proposed strategy has automatically adapted the number of inputs to take into account the level of perceptual aliasing in the environment.

There is no significant difference between the AS and the rand + approach. This suggests that the number of inputs to take into account is an important parameter (that AS finds automatically), but once it is chosen, the sensors to take into account are of lesser importance. As there are perceptual aliasing in the environment, it is clearly important to take into account past sensor values to disambiguate the state estimation. These results suggest that most sensors can actually do it and thus that their choice is not a critical issue.

C. Influence of the number of actions

What is the influence of the number of actions? To investigate this question, we have designed experiments in which the sensors taken into account are the human chosen ones and in which the number of actions changes. Results obtained after 5,000 Q-learning episodes are plotted on Figure 19. For a high number of actions, it is expected to reach the same asymptotic performance but loosing learning speed as the space to explore is bigger. The final performance is constant with respect to the number of actions on both tasks. On the ball collecting task, the standard deviation of the average performance decreases rapidly when the number of actions increases, but its average remains constant. On the box pushing, the average performance slightly decreases. It suggests that the proposed approach is not sensitive to this parameter.

VIII. Discussion and Future Work

A. Sequential versus iterative developmental process

In the approach introduced here, three different and separated steps are responsible for (1) generating data, (2) analyzing it to build actions and states and (3) exploiting it. It implies that the actions and states that are built during the second step do not influence the exploration part. The development is then sequential. In other approaches, exploration and representation building are iterative processes in which the knowledge currently acquired influences future exploration [11,36].

Many different intrinsic motivations have been proposed to take into account current knowledge, and how it changes, to choose the next actions to perform [71,72]. These approaches are very powerful as the system can, for instance, focus on actions that maximizes its learning progress [34,35]. However, they may have troubles at the very beginning of the approach, when no action has been built. Two different approaches are used in the literature for this bootstrap, either relying on random actions or using dedicated primitive actions. Providing primitive actions requires an expertise about the robot and its

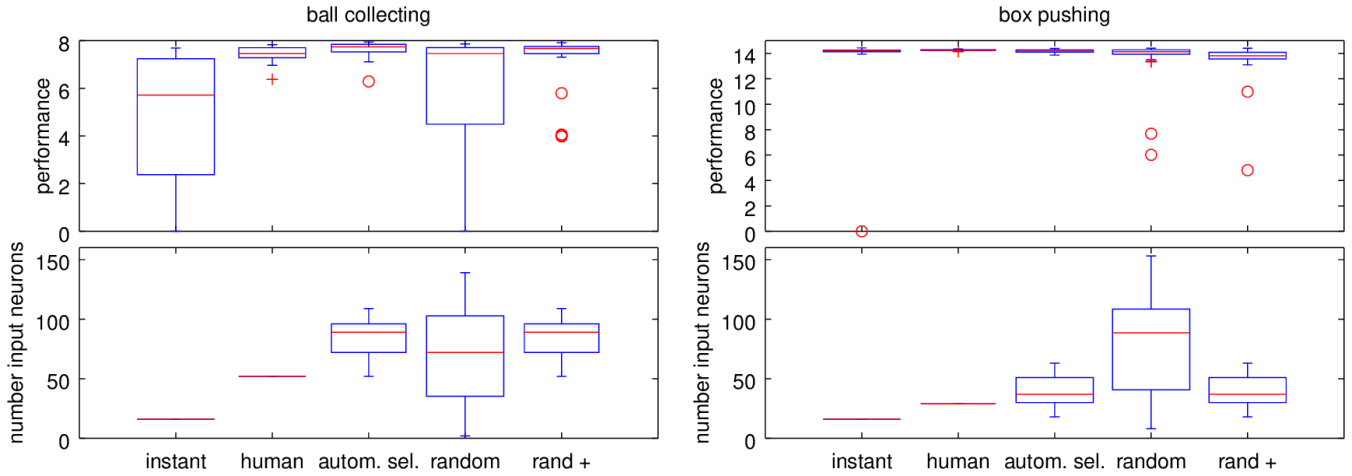

Fig. 18. Performance and number of input neurons after convergence : 5000 episodes in box pushing environment and 10000 episodes in collect ball environment. There is 5 different strategies : keeping only the sensors at the current time (instant), human selected sensors, automatic selection from traces of evolution with the presented method, totally random selected (both in number and which) and with the same number of neurons as the AS strategy but randomly selected (rand+).

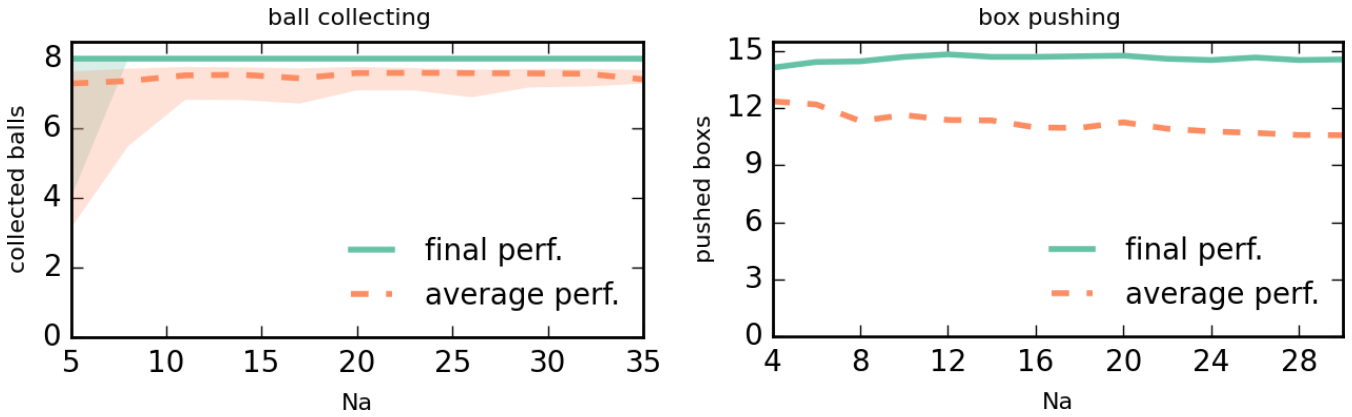

Fig. 19. Impact of the number of actions on the performance. The final performance is computed after the learning algorithm converged, whereas the average performance also takes in account episodes where the algorithm didn't already converge. The final performance represents the asymptotic performance where the average performance represents the learning speed.

environment. It goes against our first motivation. Furthermore, these primitive actions have a direct and significant impact on the very first acquired data. It may then have a strong influence on the actions and states that will be built and may bias the developmental trajectory of the system. Random actions may also create potential problems. Using them requires making an assumption about their potential for generating data of interest, i.e. data that contain the actions or states that make sense with respect to the robot mission. In an environment with objects and for a robot that has many degrees-of-freedom, for instance, very few arm movements actually lead to interactions with objects [38]. If none are observed, the system will hardly bootstrap and will remain in a local optimum in which it perfectly learns to predict and reproduce useless actions but does not go further. The approach proposed here considers the process responsible for the generation of interesting behaviors as a separate process. It corresponds to defining a babbling process in a behavioural space. It can focus on the coverage of robot abilities in the generated data while not caring about how to do it with the representations under construction. Furthermore, from a practical point of view, it allows to separate each step (data generation, data analysis and exploitation of the representation) and study each of them in isolation, what is easier, both from a scientific and technical point of view.

B. *Task-specific knowledge*

Our main motivation is to reduce the expertise on a task when using a learning algorithm in robotics. To what extent did we satisfy it with the proposed approach?

Most parameters are not specific to the task (see Appendix B). For the neuroevolution step, the only task specific parameters are the kind of used neural networks (recurrent or feed-forward) and the maximum number of steps allowed during evaluation. Most behavioural similarity measures are not specific to a particular task, only the adhoc one is. The corresponding knowledge does not require deep insight on the task, just the knowledge of the important dimensions. For RL, γ and α are to be set, as well as the number of neurons on the hidden layer of the network estimating the Q-values.

Recurrent neural networks can exhibit oscillating behaviours or generate memory-like structures. Recurrent neural networks can thus generate richer behaviours than feed-forward ones.

Generating the structure and parameters of a feed-forward neural network with neuroevolution is in any case simpler and generally faster. Recurrent neural networks can thus be a default choice when little is known about the task and the feed forward structure can be chosen if it is known that the task does not require oscillations or memories.

The number of steps used for evaluation during neuroevolution and γ are both related to the time required to solve the task. An arbitrary long number of steps for evaluation can be used, but it will make neuroevolution runs longer as the main factor influencing neuroevolution run duration is the number of evaluations. A high value of γ in RL will give more weights to future rewards (see Equation 2) and thus encourage the robot to take it into account. A low value will give more weight to immediate rewards and will thus favour immediate small rewards to the detriment of delayed high rewards. The time to get a satisfying reward is then an important parameter for both neuroevolution and reinforcement learning. A meta-learning algorithm can regulate this parameter for RL [73,74], as well as for evolution [75]. Tuning these parameters does not require a deep understanding of the tasks, notably it does not require knowing *how* the robot should solve the task, whereas providing primitive actions implies to have a clear idea about it.

C. Action extraction and second phase learning

This article proposed to use state-of-the-art neuroevolution approaches to generate data out of which action could be extracted and state could be estimated. The very simple heuristics proposed in Section IV resulted in a significant increase of generalization and learning speed. More principled algorithms like Beta Process Autoregressive Hidden Markov Model [76] could achieve a better extraction of actions. To identify relevant dimensions, it is also possible to rely on the importance of weights of the model $S_t \rightarrow A_t$ (predicting the behavior) [77] instead of relying on $S_t \rightarrow S_{t+1}$ (predicting the next state).

Many alternatives to the Q-learning algorithm, used during the second learning phase, could have been considered : (1) Neural Fitted Q Iteration [78] which aims at learning a single neural network to represent the Q-values in an offline and data efficient manner, (2) NEAT+Q [79] which combines neuro-evolutionary policy search with value-function based reinforcement learning in a continuing manner, or (3) any others data efficient reinforcement learning algorithms that relies on a finite set of actions. Future work should compare the different alternatives to identify the most relevant algorithm for each phase.

IX. Conclusion

In this work, an approach has been proposed to extract representations dedicated to discrete RL from learning traces generated by a neuroevolution approach. The goal is to bootstrap a fast RL with a slow, but task agnostic neurevolution approach.

The extracted representations consists in discrete actions with different temporal extensions and in a selection of sensors to take into account for state estimation. The RL algorithm was a Q-learning algorithm in which the Q-values were estimated with a neural network receiving as inputs the selected sensors and trained with back-propagation.

The extracted knowledge associated with the RL algorithm revealed to generalize better to new domains and to allow a faster learning on two simulated robotics tasks from the literature.

Acknowledgments

This work has been supported by the FET project DREAM[7], that has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 640891. The authors would like to thank Olivier Sigaud for his comments on a first draft of the article.

The data has been numerically analysed with the free software package GNU Octave [80] and scikit-learn [81]. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by INRIA and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Appendix A
Statistical tests

The following tables contain the $p - values$ of Mann-Whitney statistical tests. Near 0 means the 2 distributions are different.

- HD : Human designed
- R : Random
- E : Extracted from trace
- AS : Automatic Selection
- I : Instant sensors only
- R+ : Random with the number of sensors selected by the automatic selection

A. Actions extraction

		HD	E
5000	R	0.0000	0.0000
	HD		0.2018

		HD	E
3000	R	0.0000	0.0000
	HD		0.0000

		HD	E
15000	R	0.0000	0.0000
	HD		0.0004

		HD	E
6000	R	0.0000	0.0003
	HD		0.0000

		HD	E
30000	R	0.0000	0.0000
	HD		0.0081

Ball collecting

		HD	E
10000	R	0.0000	0.0006
	HD		0.0000

Box pushing

B. Sensors extraction

	HD	AS	R	R+
I	0	$8.34 \cdot 10^{-29}$	$8.56 \cdot 10^{-10}$	$8.34 \cdot 10^{-29}$
HD		$1.51 \cdot 10^{-12}$	$2.32 \cdot 10^{-2}$	$1.51 \cdot 10^{-12}$
AS			0.14	
R				0.14

Collecting ball number input neurons

[7] http://www.robotsthatdream.eu/

	HD	AS	R	R+
I	$1.39 \cdot 10^{-5}$	$1.17 \cdot 10^{-6}$	0.14	$6.3 \cdot 10^{-5}$
HD		$2.77 \cdot 10^{-2}$	$4.96 \cdot 10^{-3}$	0.37
AS			$9.19 \cdot 10^{-4}$	$6.9 \cdot 10^{-2}$
R				$1.61 \cdot 10^{-2}$

Collecting ball performance

	HD	AS	R	R+
I	0	$2.7 \cdot 10^{-13}$	$1.54 \cdot 10^{-10}$	$2.7 \cdot 10^{-13}$
HD		$8.17 \cdot 10^{-5}$	$1.17 \cdot 10^{-7}$	$8.17 \cdot 10^{-5}$
AS			$2.45 \cdot 10^{-5}$	
R				$2.45 \cdot 10^{-5}$

Box pushing number input neurons

	HD	AS	R	R+
I	0.29	0.35	0.81	0.61
HD		$3.94 \cdot 10^{-3}$	$6.53 \cdot 10^{-2}$	$1.58 \cdot 10^{-2}$
AS			$9.34 \cdot 10^{-2}$	$2.46 \cdot 10^{-2}$
R				0.74

Box pushing performance

APPENDIX B
PARAMETERS AND NOTATION

Shared parameters of evolution	
mutation probability	0.1
cross-rate probability	0
minimal weight of a connection	-5
maximal weight of a connection	5
maximum number of neuron	20
maximum number of connections	75
probability to modify a weight of a neuron	0.2
probability to add a connection	0.133
probability to add a new neuron	0.088
probability to remove a connection	0.147
probability to remove a neuron	0.098
number of input neurons	13
number of output neurons	3
population size	128
entropy discretization	9
change period of behaviour diversity	50
Unshared parameters of evolution	
topology feed-forward — ball collecting	no
topology feed-forward — box pushing	yes
maximal number of step — ball collecting	10000
maximal number of step — box pushing	300
Shared parameters of trace extraction	
τ	2
μ	8
Shared parameters of RL	
ϵ_{start}	0.15
ϵ_{end}	0.01
$\epsilon_{stepness}$	0.99
η	3
Unshared parameters of RL	
γ — ball collecting	0.9991
γ — box pushing	0.99
α — ball collecting	0.001
α — box pushing	0.1
number of hidden neurons — ball collecting	125
number of hidden neurons — box pushing	35

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[2] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement Learning in Robotics: A Survey," *International Journal of Robotics Research*, 2013.

[3] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An Application of Reinforcement Learning to Aerobatic Helicopter Flight," in *Advances in Neural Information Processing Systems: Proceedings of the 2006 Conference*. MIT Press, 2007.

[4] J. Kober and J. Peters, "Policy search for motor primitives in robotics," *Machine Learning*, vol. 84, no. 1-2, pp. 171–203, 2011.

[5] S. Doncieux, "Creativity: A driver for research on robotics in open environments," *Intellectica*, vol. 2016/1, no. 65, pp. 205–219, 2016.

[6] F. Guerin, "Learning like a baby: a survey of artificial intelligence approaches," *The Knowledge Engineering Review*, vol. 26, no. 02, pp. 209–236, 2011.

[7] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," *Evolutionary Intelligence*, vol. 1, no. 1, pp. 47–62, 2008.

[8] X. Yao, "Evolving artificial neural networks," in *Proceedings of the IEEE*, 1999, pp. 1423–1447.

[9] T. Pinville, S. Koos, J.-B. Mouret, and S. Doncieux, "How to Promote Generalisation in Evolutionary Robotics : the ProGAb Approach Formalising the Generalisation Ability," in *Proceedings of the 13th annual conference on Genetic and Evolutionary Computation*, 2011, pp. 259–266.

[10] C. Ollion and S. Doncieux, "Why and How to Measure Exploration in Behavioral Space," in *Proceedings of the 13th annual conference on Genetic and Evolutionary Computation*, 2011, pp. 267–294.

[11] J. Mugan and B. J. Kuipers, "Autonomous Learning of High-Level States and Actions in Continuous Environments," *IEEE Transactions on Autonomous Mental Development*, vol. 4, no. 1, pp. 70–86, 2012.

[12] H. Benbrahim, J. S. Doleac, J. A. Franklin, and O. G. Selfridge, "Real-time learning: a ball on a beam," in *International Joint Conference on Neural Networks*, vol. 1, 1992, pp. 98–103.

[13] M. Tokic, W. Ertel, and J. Fessler, "The Crawler, A Class Room Demonstrator for Reinforcement Learning," in *International FLAIRS Conference*, 2009, pp. 160–165.

[14] B. Nemec, M. Zorko, and L. Zlajpah, "Learning of a ball-in-a-cup playing robot," in *19th International Workshop on Robotics in Alpe-Adria-Danube Region*, 2010, pp. 297–301.

[15] R. A. Willgoss and J. Iqbal, "Reinforcement Learning of Behaviors in Mobile Robots Using Noisy Infrared Sensing," in *Australian Conference on Robotics and Automation*, 1999.

[16] G. D. Konidaris, S. Kuindersma, and A. G. Barto, "Autonomous Skill Acquisition on a Mobile Manipulator," in *National Conference of the American Association for Artificial Intelligence*, 2011.

[17] C. Kwok and D. Fox, "Reinforcement learning for sensing strategies," in *Intelligent Robots and Systems*, vol. 4, 2004, pp. 3158–3163.

[18] P. Stone, R. S. Sutton, and G. Kuhlmann, "Reinforcement Learning for RoboCup Soccer Keepaway," *Adaptive Behavior*, vol. 13, no. 3, pp. 165–188, 2005.

[19] T. Hester, M. Quinlan, and P. Stone, "Generalized model learning for Reinforcement Learning on a humanoid robot," in *IEEE International Conference on Robotics and Automation*, 2010, pp. 2369–2374.

[20] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," *Autonomous Robots*, vol. 27, no. 1, pp. 55–73, 2009.

[21] S. Thrun and T. M. Mitchell, *Lifelong robot learning*. Springer, 1995.

[22] S. J. Pan and Q. Yang, "A Survey on Transfer Learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[23] M. E. Taylor and P. Stone, "Transfer Learning for Reinforcement Learning Domains: A Survey," *Journal of Machine Learning Research*, vol. 10, pp. 1633–1685, 2009.

[24] M. E. Taylor, P. Stone, and Y. Liu, "Value functions for RL-based behavior transfer: A comparative study," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 20, 2005, p. 880.

[25] S. Doncieux, "Transfer learning for direct policy search: A reward shaping approach," *Proceedings of ICDL-EpiRob conference*, 2013.

[26] ——, "Knowledge Extraction from Learning Traces in Continuous Domains," in *Knowledge, Skill, and Behavior Transfer in Autonomous Robots, AAAI 2014 Fall Symposium Series*, 2014.

[27] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 12, pp. 181–211, 1999.

[28] G. D. Konidaris and A. S. Barto, "Skill Discovery in Continuous Reinforcement Learning Domains using Skill Chaining," in *Advances in Neural Information Processing Systems*, 2009, pp. 1015–1023.

[29] G. D. Konidaris and A. G. Barto, "Building portable options: Skill transfer in reinforcement learning," *International Joint Conferences on Artificial Intelligence*, pp. 895–900, 2007.

[30] M. L. Koga, V. F. Silva, F. G. Cozman, and A. H. Costa, "Speeding-up Reinforcement Learning Through Abstraction and Transfer Learning," in *International Conference on Autonomous Agents and Multi-agent Systems*, 2013, pp. 119–126.

[31] M. L. Koga, V. Freire, and A. H. Costa, "Stochastic Abstract Policies: Generalizing Knowledge to Improve Reinforcement Learning," *IEEE Transactions on Cybernetics*, vol. 45, no. 1, pp. 77–88, 2015.

[32] L. Riano and T. M. McGinnity, "Automatically composing and parameterizing skills by evolving Finite State Automata," *Robotics and Autonomous Systems*, vol. 60, no. 4, pp. 639–650, 2012.

[33] G. Neumann, W. Maass, and J. Peters, "Learning Complex Motions by Sequencing Simpler Motion Templates," in *International Conference on Machine Learning*. ACM, 2009, pp. 753–760.

[34] J. Schmidhuber, "Curious model-building control systems," in *International Joint Conference on Neural Networks*, 1991, pp. 1458–1463.

[35] P.-Y. Oudeyer, F. Kaplan, and V. V. Hafner, "Intrinsic motivation systems for autonomous mental development," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 2, pp. 265–286, 2007.

[36] V. R. Kompella, M. F. Stollenga, M. D. Luciw, and J. Schmidhuber, "Continual curiosity-driven skill acquisition from high-dimensional video inputs for humanoid robots," *Artificial Intelligence*, 2015.

[37] ——, "Explore to see, learn to perceive, get the actions for free: SKILL-ABILITY," in *International Joint Conference on Neural Networks*, 2014, pp. 2705–2712.

[38] C. Maestre, A. Cully, C. Gonzales, and S. Doncieux, "Bootstrapping interactions with objects from raw sensorimotor data: a Novelty Search based approach," in *IEEE International Conference on Developmental and Learning and on Epigenetic Robotics*, 2015.

[39] L. P. Kaelbling, M. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, 1996.

[40] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960.

[41] K. Deb and A. Pratap, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[42] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[43] M. Zimmer, Y. Boniface, and A. Dutech, "Neural fitted actor-critic." in *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2016.

[44] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," *arXiv preprint arXiv:1604.06778*, 2016.

[45] L. J. Lin, "Scaling up reinforcement learning for robot control," *Thenth International Conference on Machine Learning*, 1993.

[46] G. Tesauro, "Practical issues in temporal difference learning," *Machine Learning*, 1992.

[47] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[48] A. McGovern, R. S. Sutton, and A. H. Fagg, "Roles of macro-actions in accelerating reinforcement learning," in *Grace Hopper Celebration of Women in Computing*, 1997.

[49] F. Stulp and O. Sigaud, "Robot Skill Learning: From Reinforcement Learning to Evolution Strategies," *Paladyn. Journal of Behavioral Robotics*, vol. 4, no. 1, pp. 49–61, 2013.

[50] S. Doncieux, N. Bredeche, J.-B. Mouret, and A. E. G. Eiben, "Evolutionary robotics: What, why, and where to," *Frontiers in Robotics and AI*, vol. 2, p. 4, 2015.

[51] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, pp. 303–314, 1989.

[52] A. L. Nelson, G. J. Barlow, and L. Doitsidis, "Fitness functions in evolutionary robotics: A survey and analysis," *Robotics and Autonomous Systems*, vol. 57, no. 4, pp. 345–370, 2009.

[53] D. Filliat, J. Kodjabachian, and J.-A. Meyer, "Evolution of neural controllers for locomotion and obstacle-avoidance in a 6-legged robot," *Connection Science*, vol. 11, pp. 223—-240, 1999.

[54] S. Doncieux and J.-B. Mouret, "Behavioral diversity with multiple behavioral distances," in *IEEE Congress on Evolutionary Computation*, 2013.

[55] B. Allen and P. Faloutsos, "Complex networks of simple neurons for bipedal locomotion," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2009, pp. 4457–4462.

[56] J. Lehman and K. O. Stanley, "Abandoning Objectives: Evolution Through the Search for Novelty Alone," *Evolutionary Computation*, vol. 19, no. 2, pp. 189–223, 2010.

[57] A. Cully and J.-B. Mouret, "Evolving a behavioral repertoire for a walking robot," *Evolutionary computation*, vol. 24, no. 1, pp. 59–88, 2016.

[58] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret, "Robots that can adapt like animals," *Nature*, vol. 521, no. 7553, pp. 503–507, 2015.

[59] S. Nolfi and D. Floreano, *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. MIT press, 2000.

[60] J. Lehman, S. Risi, D. D'Ambrosio, and K. O. Stanley, "Encouraging reactivity to create robust machines," *Adaptive Behavior*, vol. 21, no. 6, pp. 484–500, 2013.

[61] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing (Natural Computing Series)*. Springer, 2008.

[62] K. Deb, *Multi-objective optimization using evolutionnary algorithms*. Wiley, 2001.

[63] S. Doncieux and J.-B. Mouret, "Beyond black-box optimization: a review of selective pressures for evolutionary robotics," *Evolutionary Intelligence*, vol. 7, no. 2, pp. 71–93, 2014.

[64] J. C. Bongard, "Evolutionary robotics," *Communications of the ACM*, vol. 56, no. 8, p. 74, 2013.

[65] J.-B. Mouret and S. Doncieux, "Encouraging Behavioral Diversity in Evolutionary Robotics: An Empirical Study." *Evolutionary computation*, vol. 20, no. 1, pp. 91–133, 2012.

[66] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[67] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 1-2, pp. 181–211, aug 1999.

[68] J. C. Bezdek, R. Ehrlich, and W. Full, "FCM: The fuzzy c-means clustering algorithm," *Computers & Geosciences*, 1984.

[69] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society*, 1996.

[70] R. Smith, "Open dynamics engine," 2005.

[71] P.-Y. Oudeyer and F. Kaplan, "What is Intrinsic Motivation? A Typology of Computational Approaches." *Frontiers in Neurorobotics*, vol. 1, p. 6, 2007.

[72] G. Baldassarre and M. Mirolli, *Intrinsically Motivated Learning in Natural and Artificial Systems*. Berlin, Heidelberg: Springer, 2013.

[73] K. Doya, "Metalearning and neuromodulation," *Neural Networks*, vol. 15, no. 46, pp. 495–506, 2002.

[74] M. Khamassi, C. R. Wilson, M. Rothé, R. Quilodran, P. F. Dominey, and E. Procyk, "Meta-learning, cognitive control, and physiological interactions between medial and lateral prefrontal cortex," in *Neural Basis of Motivational and Cognitive Control*. MIT Press, 2011, pp. 351–370.

[75] G. Karafotias, M. Hoogendoorn, and A. E. Eiben, "Parameter Control in Evolutionary Algorithms: Trends and Challenges," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 2, pp. 167–187, 2015.

[76] S. Niekum, S. Osentoski, G. Konidaris, S. Chitta, B. Marthi, and A. G. Barto, "Learning grounded finite-state representations from unstructured demonstrations," *The International Journal of Robotics Research*, vol. 34, no. 2, pp. 131–157, 2015.

[77] L. C. Cobo, K. Subramanian, C. L. Isbell, A. D. Lanterman, and A. L. Thomaz, "Abstraction from demonstration for efficient reinforcement learning in high-dimensional domains," *Artificial Intelligence*, vol. 216, pp. 103–128, 2014.

[78] M. Riedmiller, "Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method," in *European Conference on Machine Learning*. Springer, 2005, pp. 317–328.

[79] S. Whiteson and P. Stone, "Evolutionary function approximation for reinforcement learning," *Journal of Machine Learning Research*, vol. 7, no. May, pp. 877–917, 2006.

[80] J. W. Eaton, D. Bateman, and S. Hauberg, *GNU Octave version 3.0. 1 manual: a high-level interactive language for numerical computations*. SoHo Books, 2007.

[81] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Pettenhofer, R. Weiss, and V. Dubourg, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.