Asynchronous Neurocomputing for optimal control and reinforcement learning with large state spaces

Bruno Scherrer

Asynchronous Neurocomputing for optimal control and reinforcement learning with large state spaces

Bruno Scherrer

scherrer@csail.mit.edu

Computer Science and Artificial Intelligence Laboratory
NE43-783, 200 Technology Square
Cambridge, MA 02139-4307, USA

LORIA INRIA-Lorraine, CORTEX/MAIA Teams
Campus Scientifique BP 239
54506 Vandœuvre-lès-Nancy, France

Abstract

We consider two machine learning related problems, optimal control and reinforcement learning. We show that, even when their state space is very large (possibly infinite), natural algorithmic solutions can be implemented in an asynchronous neurocomputing way, that is by an assembly of interconnected simple neuron-like units which does not require any synchronization. From a neuroscience perspective, this work might help understanding how an asynchronous assembly of simple units can give rise to efficient control. From a computational point of view, such neurocomputing architectures can exploit their massively parallel structure and be significantly faster than standard sequential approaches.

The contributions of this paper are the following: 1) We introduce a theoretically sound methodology for designing a whole class of asynchronous neurocomputing algorithms. 2) We build an original asynchronous neurocomputing architecture for optimal control in a small state space, then we show how to improve this architecture so that also solves the reinforcement learning problem. 3) Finally, we show how to extend this architecture to address the case where the state space is large (possibly infinite) by using an asynchronous neurocomputing adaptive approximation scheme. We illustrate this approximation scheme on two continuous space control problems.

Key words: neurocomputing, optimal control, reinforcement learning

For reading convenience, we sum up here the abbreviations we use throughout the paper:

- AN: asynchronous neurocomputing

- CM: contraction mapping
- CMFP: contraction mapping fixed point
- MDP: Markov decision process
- PADC: parallel asynchronous distributed computation
- RL: reinforcement learning

Introduction

Research in neurocomputing combines two intimately related (and sometimes contradictory) motivations 1) Doing good computer science using interesting ideas taken from neuroscience and 2) Understanding neuroscience issues better with computer (theoretical or simulation-based) modelling. We think that a way to satisfy both motivations at the same time is to make or strenghten the relation between a certain computational "intellectual" capacity C (memory, generalization, control, etc...) and a specific brain-like process P. To achieve this, one encounters two opposite and complementary trends in the literature: a) the *bottom-up* neurocomputing researchers copy real neurons, study the resulting process P from a computational point of view, and update the model until it exhibits capacity C; while, b) the *top-down* neurocomputing researchers formalize the computational "intellectual" capacity C and try to find a process P which respects most of the constraints of the neurocomputing paradigm. There are advantages and drawbacks in both approaches: the former is often closer to neuroscience but tends to be more empirical. The latter is closer to computational science but often lacks biological plausibility. The research we present in this paper belongs to the latter top-down approach: we consider the related capacities of control and reinforcement learning as they are formalized in the machine learning literature, and show that, even when the problem is hard (when the state space is big), they can be addressed by neurocomputing. Rather than proposing new computational methods for solving these machine learning problems, our aim is here to show that their standard mathematically-motivated solutions are naturally compatible with the neurocomputing paradigm.

As there is no general accepted definition of what neurocomputing is, (and to make sure that the reader can quickly understand the assumptions of this work), we now explain what we exactly mean by neurocomputing. In this paper, a neurocomputing algorithm will be an assembly of interconnected units. Each such unit will have a number of internal variables (or internal states) and will receive a number of input variables from other units. A unit will only be able to perform basic computations (*in this paper:* `sum`, `max`, `argmax` and access to a composant of a finite local array) with all these variables and store the result in one of its local variables. Pieces of numerical information will be able to flow from specific units' variables (called units' outputs) to other

units through weighted connections; when a numerical value goes from one unit to another through a given connection, it will be modulated (*in this paper:* multiplied) by the connection weight. Connection weights might evolve over time using some local rules. Finally, and in contrast with some research in neurocomputing (like the seminal work by McCulloch & Pitts [11] about the universal computation property of *synchronous* assemblies of neurons), the neurocomputing processes which we will describe will not require global synchronization: there will be no general clock for deciding when each unit needs to make a computational step; each unit will follow an individual local process at its own rythm, taking as inputs the available (not necessarily up-to-date) outputs of the other units. To stress this particularity, we will refer to these computations as *asynchronous neurocomputing* (AN).

The paper is organized as follows: Section 1 reviews a theoretical result of the parallel computing literature showing that the computation of a contraction mapping fixed point (CMFP) on a reasonably small space can efficiently be done with parallel asynchronous distributed computations (PADC). This result will stand for a foundation for building asynchronous neurocomputing (AN) algorithms throughout the rest of the paper. In section 2, we formalize the optimal control problem and recall that it involves the computation of a CMFP. We exploit this property in order to build an AN architecture for solving small state space optimal control. We also show that adding two simple learning rules allows us to tackle the case where the problem parameters are uncompletely known, that is the reinforcement learning problem. Section 3 addresses the large state space case: we propose and analyze an approximation scheme, state aggregation, which only relies on computations of CMFPs on reasonably small spaces. We use this result to add two AN layers to our architecture that allow to compute how to efficiently refine the approximation. The potential of this final architecture is illustrated on two continuous state space control problems.

1 A link between global and local computations

Consider a network of distributed and asynchronous processing units. In general, it is difficult to make the link between the local computations and the global activity of the network. It is often more complicated when local computations are not globally synchronized. This section reviews a well-known result about the computation of a contraction mapping fixed point (CMFP), for which we can make the link between the local and the global levels. This result is fundamental as it stands for a general argument for the remainder of the paper: any computation that can be characterized in terms of a CMFP can be treated with parallel asynchronous distributed computations (PADC).

Consider a finite set X, a norm $\|.\|$ on $\mathbb{R}^X$, and a contraction mapping $M : \mathbb{R}^X \longmapsto \mathbb{R}^X$, such that $\|M(f') - M(f)\| \leq \gamma.\|f' - f\|$ with $\gamma \in [0, 1)$. It is a well-known result (see for instance [1]) that M has one and only one fixed point f^*, that is a function of $\mathbb{R}^X$ that satisfies

$$f^* = M(f^*). \tag{1}$$

In traditional (sequential) computations, a technique for computing the fixed point f^* is to iterate the following process:

$$\text{For all } x \in X, \text{ do } f^{t+1}(x) \leftarrow \left[M(f^t)\right](x). \tag{2}$$

Indeed we have $f^t \overset{t\to\infty}{\longrightarrow} f^*$ with an exponential rate of convergence:

$$\|f^t - f^*\| \leq \gamma.\|f^{t-1} - f^*\| \leq \gamma^t.\|f^0 - f^*\|.$$

The temporal complexity of this iterative process can be seen as the product of two terms:

- t_1: the time required for doing the iteration given by equation 2
- t_2: the number of iterations needed to approximate f^* with enough precision.

It is shown in [1] that f^* can be computed with PADC. Indeed, the following process

$$f(x) \leftarrow [M(f)](x)$$

can be distributed over all $x \in X$ and f will also converge to f^*. Furthermore, these computations need not be synchronized: updates of $f(x)$ for all $x \in X$ can be done in an arbitrary order, or even with different frequencies. If one updates each $f(x)$ infinitely often, the process will converge to the CMFP. If we neglict communication delays, parallelizing this way might roughly divide the time t_1 (thus the global temporal complexity for estimating the fixed point) by the size $|X|$ of X.

The computation of a CMFP can be done with PADC and can be significantly faster (roughly $|X|$ times faster) than the sequential iterative process described by equation 1. Therefore, any problem that can be formulated as the computation of a CMFP can be solved more rapidly with PADC than with the iterative process described by equation 2. Similarly, any neurocomputing algorithm whose goal will be to compute a CMFP will not require global synchronization.

2 Asynchronous Neurocomputing for small state space optimal control

In this section, we formalize two problems, optimal control and reinforcement learning, in the common framework of Markov Decision Processes, and we argue that they can be solved by AN if the state space is reasonably small.

2.1 Markov decision processes

Markov decision processes (MDPs) [13] provide the theoretical foundations of challenging problems to researchers in artificial intelligence and operation research. These problems include optimal control and reinforcement learning [18].

An *MDP* is a controlled stochastic process satisfying the Markov property with rewards (numerical values) assigned to state-action pairs. Formally, an MDP is a four-tuple $\langle S, A, T, R \rangle$ where S is the *state space*, A is the *action space*, T is the *transition function* and R is the *reward function*. T is the state-transition probability distribution conditioned by the action. For all states s and s' and actions a,

$$T(s, a, s') \stackrel{def}{=} \Pr(s_{t+1} = s' | s_t = s, a_t = a).$$

Without loss of generality, we define $R(s, a) \in \mathbb{R}$ as the instantaneous reward for taking action $a \in A$ in state S.

As an illustration, consider a navigation problem where an agent moves in an environmenent with a P shape and whose goal is to reach a specific zone of the environment (see figure 1). We model this problem with a $16 \times 16 = 256$ state MDP. The state of the system corresponds to the location of the agent. The set of actions of this agent consists of 8 desired movements. The effect of a desired movement is slightly corrupted with noise and is cancelled if it makes the agent hit a wall. We give a positive reward (+1) when the agent reaches the goal zone. In all other cases, the agent gets no feedback (a zero reward).

2.2 Optimal Control

In the MDP framework, the *optimal control problem* consists in finding a *policy*, that is a mapping $\pi : S \to A$ from states to actions, that maximises the expected long-term amount of rewards, also called value function of policy

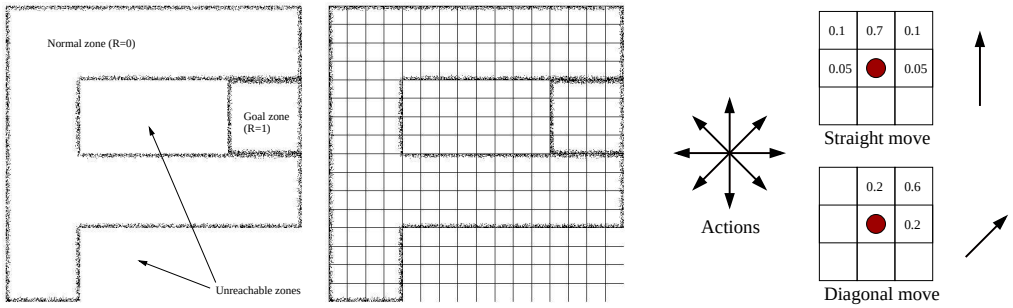

Fig. 1. **A simple navigation problem.** *Left*: We model a navigation problem in a P-shape environment. *Center*: We represent the navigation problem with a grid-MDP; at each instant, the agent is in one and only one position of the grid. *Right*: The agent can move in the 8 directions; his moves are slightly corrupted with noise; for instance, when the agent decides to go North-East, the probability of actually getting to the North-East position is 0.6.

π:

$$V^{\pi}(s) = \mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t . R(s_t, \pi(s_t)) | s_0 = s\right].$$

We assume here that the MDP has an infinite horizon and that future rewards are discounted exponentially with a discount factor $\gamma \in (0, 1)$. It is shown [13] that there exists a unique optimal value function V which is the fixed point of the following contraction mapping, also called Bellman operator:

$$[B.f](s) = \max_a \left(R(s, a) + \gamma . \sum_{s'} T(s, a, s') . f(s')\right). \tag{3}$$

Once an optimal value function V is computed, an optimal policy can immediately be derived as follows:

$$\pi(s) \in \arg\max_a \left(R(s, a) + \gamma . \sum_{s'} T(s, a, s') . V(s')\right).$$

As the Bellman operator B is a contraction mapping, the standard digital approach for solving the optimal control problem is to use a sequential iterative procedure similar to equation 2, known as *Value Iteration* [13], which consists in repeating the process described in algorithm 1. Using an argument similar

Algorithm 1 One iteration of Value Iteration

for all $s \in S$ **do**
  for all $a \in A$ **do**
    $Q(s, a) \leftarrow R(s, a) + \gamma . \sum_{s'} T(s, a, s') . V(s')$
  end for
  $V(s) \leftarrow \max_a Q(s, a)$
  $\pi(s) \leftarrow \arg\max_a Q(s, a)$
end for

to the one we gave in section 1, the authors of [1] argue that optimal control

can be done with PADCs. While their (parallel computing) aim was to come up with efficient parallel implementations on real parallel computers, ours is here a bit different as we want the optimal control computation to fit in the (virtual) neurocomputing paradigm. Providing a neurocomputing description is what we do from now on in this subsection.

For each state $s \in S$, define one unit which contains the following internal variables:

- $R[|A|]$: an array of $|A|$ floats which contains the immediate reward for taking actions in state s
- V : a float which stands for $V(s)$
- $Q[|A|]$: an array of $|A|$ floats which contains the values for $Q(s,a)$
- π : an element of A which corresponds to the optimal action

Define the V variable as an output for each unit, that is a variable which might be propagated to other units through connections. For each state-action-state possible transition (s,a,s'), define a connection $c^a_{ss'}$ with weight $w^a_{ss'} = \gamma.T(s,a,s')$, and $s \leftarrow s'$ orientation (information for computing the optimal value function needs to flow in the opposite direction of the dynamics).

We are now going to argue that solving the optimal control problem can be seen as a neurocomputing process which is distributed over all units. Indeed, asynchronous Value Iteration is processed if each single unit keeps on doing the following operations described in algorithm 2. In other words, for each

Algorithm 2 Local process followed by each unit for optimal control

for all $a \in A$ **do**
  $Q[a] \leftarrow R[a] + \sum_{s'} w^a_{ss'}.V(s')$
end for
$V \leftarrow \max_a Q[a]$
$\pi \leftarrow \arg\max_a Q[a]$

action a, a unit makes the standard weighted sum of its neighbors' V values for action a (through incident connections), adds it to its internal value $R[a]$ and assigns the result to $Q[a]$; then it updates its internal variables V and π with `max` and `argmax` operators. This neurocomputing architecture and the process performed by one unit are sketched in figure 2.

The computation of optimal control can thus be seen as the result of a distributed activity (V values) over a recurrent network of simple processors, activity which always ends up stabilizing (iterative CMs converge to the unique corresponding fixed point). The parameters of the problem are embedded in the architecture: the transition function is stored in the connection weights and the reward function is spread over local unit variables.

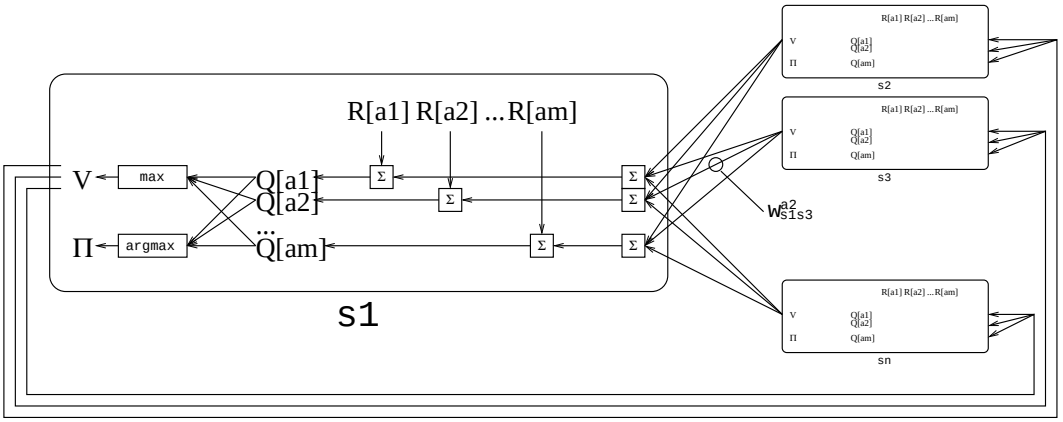

Fig. 2. **An asynchronous neurocomputing architecture for optimal control.** We focus on one unit s_1 and show how it updates its internal variables and how it interacts with the other units s_2, s_3, ..., s_n. For each action a, s_1 makes the standard weighted sum of its inputs (V values of its connected units) for action a, adds it to its internal value $R[a]$ and assigns the result to $Q[a]$; then it updates its internal variables V and π with `max` and `argmax` operators. The arrows stand for the connection as well as for the flow of information. As an illustration, one of the connections is identified by its weight ($w^a_{s_1 s_3}$).

Consider the simple navigation problem we introduced earlier. Figure 3 shows the connectivity of the corresponding network while figure 4 shows the evo-

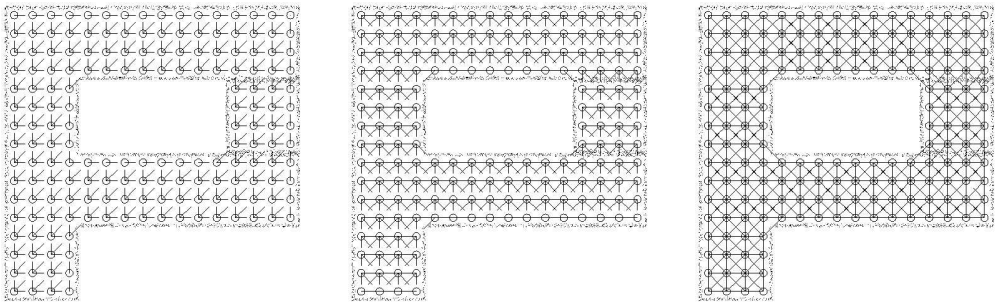

Fig. 3. **Connexions of the network for the navigation problem.** *Left:* connections corresponding to action North-East. *Center:* connections for action South. *Right:* superposition of connections for all actions.

lution of the activity V, iteration after iteration[1], and the eventual optimal policy. In this example (where dynamics is local and isotropic), we can observe that the activity V is propagating from the goal zone to every region of the state space and that the optimal policy consists in locally climbing up the gradient of V.

This sort of activity propagation is reminiscent of bottom-up works on neu-

[1] For simplicity, we used synchronous updates over the whole state space in this simple experiment but we insist again on the fact that synchronicity is not required.

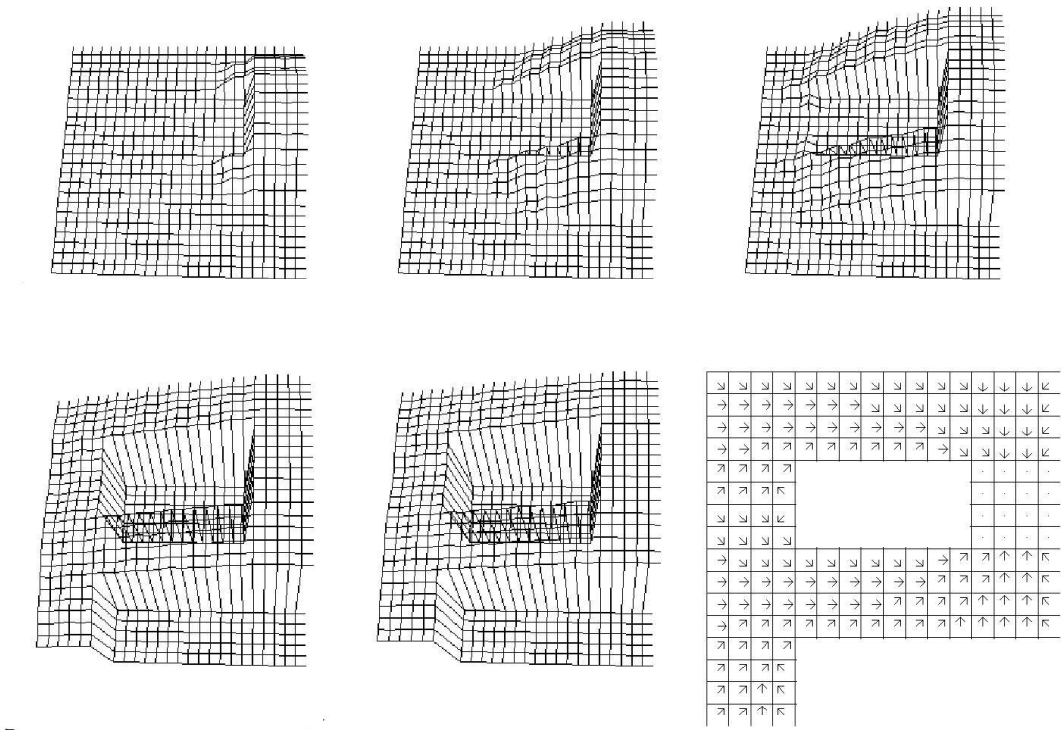

Fig. 4. **Evolution of the value function computed by the network and eventual optimal policy.** The activity V is propagating from the goal zone to every region of the state space and the optimal policy consists in locally climbing up the gradient of V.

rocomputing architectures for control (see [4], [3] and [14]) inspired by a neurobiological theory of cortex [2]. In these models, a unit corresponding to a goal is put in a specific activity, called *desire state*. This *desire state* spreads along inter-unit connections, until it reaches the *active unit*, that is the unit corresponding to the current system state. The chosen control is then the one that locally maximizes the gradient of the *desire activity*. The whole process is called *call propagation*, and is analogical to the value propagation in an MDP through an algorithm like Value Iteration.

2.3 Reinforcement Learning

A variant of optimal control where the MDP parameters (R and T) are initially incompletely known, and therefore must be estimated through sample experiments, is called *reinforcement learning* (RL) [18]. When optimal control only involves planning, RL involves both learning (estimation of the parameters)

and planning and is therefore a slightly more difficult problem.

We won't discuss here important RL issues such as

- the problem of effectively balancing planning and learning, known as the exploration-exploitation dilemma, and
- the possibility of planning without maintaining an internal representation of the parameters R and T, in the so-called model-free or Q-Learning-like approaches.

The interested reader should for instance consult [18] or [9] for general pointers. However, we are going to propose a natural extension to the architecture we have just described, so that it be able to handle reinforcement learning. A straightforward solution to the RL problem, known as model-based RL, considers planning and learning as two independent though complementary processes. Given some samples about the problem to be solved, that is some sequence $H = (s_0, a_0, r_0, s_1, a_1, r_1, ...)$ of state transitions and associated rewards, a natural way to solve the RL problem is 1) to approximate the unknown parameters with the following standard unbiased estimates:

$$\begin{cases} R(s,a) \leftarrow \frac{\sum_{(s_i,a_i)=(s,a)} r_i}{\#(s,a)} \\ T(s,a,s') \leftarrow \frac{\#(s,a,s')}{\#(s,a)} \end{cases} \tag{4}$$

where $\#(x)$ counts the number of times subsequence x appears in H and 2) to do optimal control based on these approximate parameters (using for instance the architecture we described in the previous subsection). Such a technique is sound as it is proved that it converges when the amount of samples tends to infinity [7].

From our neurocomputing architecture perspective, a strictly equivalent way to write the estimation procedure of equation 4 is to state that, for each new sample (s, a, r, s'), one updates the current parameters of our neurocomputing architecture as described in algorithm 3. This process is modulated by a

Algorithm 3 Updates of the parameters of s when observing sample (s, a, r, s')

$R(s,a) \leftarrow (1 - \alpha_{sa}).R(s,a) + \alpha_{sa}.r$
for all s'' **do**
$$w^a_{ss''} \leftarrow (1 - \alpha_{sa}).w^a_{ss'} + \begin{cases} \alpha_{sa}.\gamma \text{ if } s'' = s' \\ 0 \text{ else} \end{cases}$$
end for
$\alpha_{sa} \leftarrow \frac{\alpha_{sa}}{\alpha_{sa}+1}$

learning rate α_{sa}, which decreases sample after sample, following the sequence 1, $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, ...

This equivalent formulation is all the more interesting that the resulting connection weight update rule can be interpreted as a hebbian-like learning, because it is similar to what Hebb postulated as the basic form of learning in the brain [8]. Hebb suggested that changes of synaptic weights are related to the correlation between the firing of pre- and post-synaptic neurons. In our architecture, if we think of the transition $s \to s'$ as a close sequence of firings involving pre-unit s then post-unit s', hebbian learning would reinforce the link between s and s' and decrease the weight between s and all other possible post-unit s''. This is exactly what happens in our network: when for action a, we observe transition $s \to s'$, $w^a_{ss'}$ slightly increases while $w^a_{ss''}$ decreases for all $s'' \neq s'$.

Another interesting feature of this formulation is that it explicitly emphasizes the estimation process as an amplitude-decreasing learning process. If decreasing the amplitude of learning makes sense for stationary environments (i.e. fixed R and T reinforcement learning parameters), keeping a constant learning factor $\alpha_{sa} = \alpha_0$ would in theory allow to follow non-stationary reinforcement learning parameters (see [6] for more discussion about the stakes related to the choice of the learning rate).

To conclude, we showed that it is straightforward to augment the AN architecture that we introduced for optimal control so that it also handle RL. Indeed, we can use the model-based approach for RL which builds estimates of the unknown parameters and this can be done through two simple learning rules, one of which being hebbian.

3 Asynchronous neurocomputing for large state space optimal control

From a computational point of view, the use of AN is particularly interesting because it involves PADCs and therefore can be significantly faster than standard digital computing. Even if PADCs can accelerate the computation of the optimal value function V in an MDP, it might not be sufficient when the state space S is very large. When the state space is infinite, it might even be impossible to use such an approach as it would require an infinite number of units. Moreover, the theoretical number of required iterations t_2 in order to approximate V with enough precision can dramatically grow with the number of states [10]. PADCs only diminish the part t_1 of the temporal complexity (see page 4) and this might not be sufficient for large state spaces. Eventually, in the reinforcement learning problem, the size of the state space might also have a dramatic effect on the time required for estimating R and T with a reasonable precision.

In this section, we propose and analyse an approximation scheme, state aggregation, which allows to use PADC for large state space MDPs. We begin by reviewing some recently published foundations for approximation in MDPs. We then apply them to the state aggregation case and show that it only involves CMFP on reasonably small spaces. Finally, we illustrate this approximation on two (infinite) continuous state space control problems.

3.1 Safely approximating an MDP

This subsection reviews some theoretical results by [12] for analysing and iteratively improving an MDP approximation. Consider an MDP $\mathcal{M} = \langle S, A, T, R \rangle$ and an approximation of it $\widehat{\mathcal{M}} = \langle S, A, \widehat{T}, \widehat{R} \rangle$. Let B be the exact Bellman operator of $\mathcal{M}$ (as in equation 3) and let $\widehat{B}$ be the approximate Bellman operator:

$$\left[\widehat{B}.f\right](s) = \max_a \left(\widehat{R}(s,a) + \gamma . \sum_{s'} \widehat{T}(s,a,s').f(s') \right).$$

Let V and $\widehat{V}$ be their fixed points. In practice, we would like to compute the exact value function V but we can only compute its approximation $\widehat{V}$. It is then interesting to evaluate the *approximation error*:

$$E(s) \stackrel{def}{=} |V(s) - \widehat{V}(s)|.$$

The authors of [12] show that the approximation error depends on a quantity they call *interpolation error*:

$$e(s) \stackrel{def}{=} |\widehat{B}.V(s) - B.V(s)| = |\widehat{B}.V(s) - V(s)|$$

which is the error due to one approximate mapping $\widehat{B}$ of the real value function V ; it measures how the approximate parameters $(\widehat{R}(s,a),\ \widehat{T}(s,a,.))$ locally differ from the real parameters $(R(s,a), T(s,a,.))$. Indeed, we can show [16] that for some constant K

$$e(s) \leq \max_a \left| R(s,a) - \widehat{R}(s,a) \right| + K. \max_a \left(\sum_{s' \in S} \left| T(s,a,s') - \widehat{T}(s,a,s') \right| \right). \quad (5)$$

If $\overline{e}(s)$ is an upper bound of $e(s)$ (e.g. take the bound given by equation 5), then, an upper bound $\overline{E}(s)$ of $E(s)$ is the fixed point of the following contraction mapping:

$$[E.f](s) = \overline{e}(s) + \max_a \left(\gamma . \sum_{s'} \widehat{T}(s,a,s').f(s') \right). \quad (6)$$

The authors of [12] also explain how to improve an approximation, i.e. how to distribute resources for describing the parameters R and T over the whole state space. Indeed, they introduce the notion of *influence* $I_{S_0}(s)$ of the local interpolation error of state s on the approximation error over a subset $S_0 \subset S$:

$$I_{S_0}(s) \stackrel{def}{=} \frac{\partial \sum_{s' \in S_0} \overline{E}(s')}{\partial \overline{e}(s)}.$$

Typically, S_0 is chosen to be the set of possible starting states of the process. Say we add or remove some resources near some state s. This might change the interpolation error by $\Delta\overline{e}(s)$. A gradient argument shows the effect this will have on the approximation error:

$$\Delta \left(\sum_{s' \in S_0} \overline{E}(s) \right) \simeq I_{S_0}(s).\Delta\overline{e}(s). \tag{7}$$

This result suggests to remove resources where the above quantity is very little and to add resources where it is big. It is proven that the influence I_{S_0} is also the fixed point of a contraction mapping:

$$[D.f](s) = \begin{cases} 1 \text{ iff } s \in S_0 \\ 0 \text{ iff } s \notin S_0 \end{cases} + \gamma . \sum_{s'} \widehat{T}(s', \pi_{err}(s'), s).f(s')$$

where $\pi_{err}(s) = \arg\max_a \sum_{s'} \widehat{T}(s, a, s').E(s')$ is the policy that generates the maximum uncertainty (see [12] for more details).

The approximation error E and the influence I_{S_0} are characterized as CMFPs on the state space S. This means that they are, in theory, as difficult to compute as the optimal value function in the real MDP $\mathcal{M}$. The next subsection describes a practical solution to this issue.

3.2 The state aggregation approximation

In this subsection, we introduce and discuss the MDP approximation with state aggregation. We apply the analysis of the previous subsection to this approximation scheme and show that the computations of the approximation error E and the influence I_{S_0} become tractable.

Given an MDP $\mathcal{M} = \langle S, A, T, R \rangle$, the state aggregation approximation consists in introducing the MDP $\widehat{\mathcal{M}} = \langle \widehat{S}, A, \widehat{T}, \widehat{R} \rangle$ where the state space $\widehat{S}$ is a partition of the real state space S. Every element of $\widehat{S}$, which we call macro-state, is a subset of S and every element of S belongs to one and only one macro-state. In addition, every object defined on $\widehat{S}$ can be seen as an object of S which is constant on every macro-state (this is also called piecewise

constant approximation). The number of elements of $\widehat{S}$ can be chosen little enough so that it is feasible to compute CMFPs on $\widehat{S}$. When doing a state aggregation approximation, natural choices[2] for $\widehat{R}$ and $\widehat{T}$ are the averages of the real parameters on each macro-state:

$$\begin{cases} \widehat{R}(\widehat{s},a) = \frac{1}{|\widehat{s}|}.\sum_{s\in\widehat{s}} R(s,a) \\ \widehat{T}(\widehat{s_1},a,\widehat{s_2}) = \frac{1}{|\widehat{s_1}|}.\sum_{(s,s')\in\widehat{s_1}\times\widehat{s_2}} T(s,a,s') \end{cases} . \tag{8}$$

As a possible illustration of this approximation scheme, let us consider state aggregation in a continuous multi-dimensional state space, which is where most real-life human control problems happen to be. A common tool for doing state aggregation in such continuous spaces is variable resolution discretization with kd-trees [5]. The principle is the following: One uses a tree to represent a discretization of the state space. The root of this tree covers the whole state space ; it has two branches which cut the space in two (according to one of the dimensions). Recursively, each node in the tree covers a region of the state space and its children cover half of it. The leaves of this tree correspond to a variable resolution discretization of the space. In the example of figure 5,

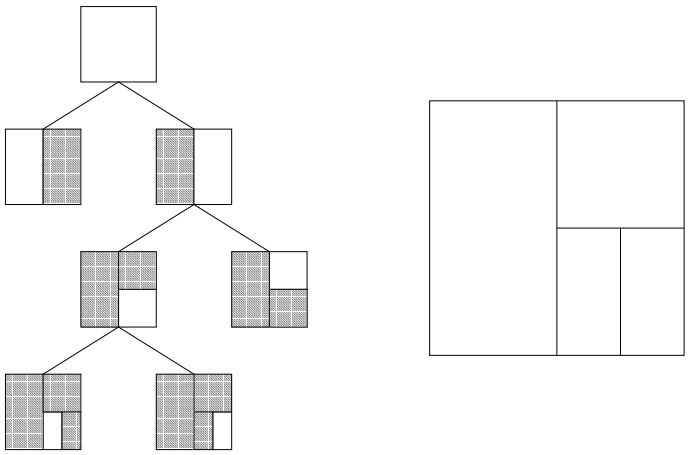

Fig. 5. **Example of a variable resolution discretization with a kd-tree.** We consider a 2-dimensional space and the discretization (right) is characterized by the tree (left). The root of this tree covers the whole state space ; it has two branches which cut the space in two (according to the vertical dimension). Recursively, each node in the tree covers a region of the state space and its children cover half of it (the region covered by each node is shown in white). The 4 leaves of the tree correspond to the 4 regions of the discretization.

doing the state aggregation approximation amounts to defining an MDP on the 4-state space induced by the kd-tree leaves with parameters deduced from

[2] We want the interpolation error to be as little as possible and we need to satisfy consistency constraints such as $\sum_{\widehat{s_2}} T(\widehat{s_1},a,\widehat{s_2}) = 1$.

equations 8. The AN for computing the optimal control is thus defined on the macro-states (which are identified to the kd-tree leaves).

Let us come back to the theoretical analysis of this approximation scheme. We can deduce [15] from equations 5 and 8 an upper bound of the interpolation error on the macro-state $\widehat{s_1} \in \widehat{S}$:

$$\forall s_1 \in \widehat{s_1}, e(s_1) \leq \overline{e}(\widehat{s_1}) = \overline{\Delta R}(\widehat{s_1}) + K. \sum_{\widehat{s_2} \in \widehat{S}} \overline{\Delta T}(\widehat{s_1}, \widehat{s_2}) \tag{9}$$

$$\text{with } \begin{cases} \overline{\Delta R}(\widehat{s}) = \max_{a \in A, (s,s') \in \widehat{s}} |R(s,a) - R(s',a)| \\ \overline{\Delta T}(\widehat{s_1}, \widehat{s_2}) = \max_{a \in A, (s_1, s_1') \in \widehat{s_1}} |\sum_{s_2 \in \widehat{s_2}} T(s_1, a, s_2) - T(s_1', a, s_2)| \end{cases}.$$

The approximation by state aggregation is particularly interesting because the approximation error can be computed with a complexity that is similar to the one required for computing the optimal control in the *approximate* model $\widehat{\mathcal{M}}$. Indeed, as the upper bound of the interpolation error is constant over each macro-state, equation 6 becomes a contraction mapping on $\widehat{S}$ (and *not anymore* on S):

$$[E'.f](\widehat{s_1}) = \overline{e}(\widehat{s_1}) + \max_a \left(\gamma. \sum_{\widehat{s_2}} \widehat{T}(\widehat{s_1}, a, \widehat{s_2}).f(\widehat{s_1}) \right). \tag{10}$$

Using the same arguments, the influence of the interpolation error on the approximation for a subset S_0 is the fixed point of the following contraction mapping defined on $\hat{S}$:

$$[D'.f](\widehat{s}) = \begin{cases} 1 \text{ iff } \widehat{s} \subset S_0 \\ 0 \text{ iff } \widehat{s} \not\subset S_0 \end{cases} + \gamma. \sum_{\widehat{s'}} \widehat{T}(\widehat{s'}, \pi_{err}(\widehat{s'}), \widehat{s}).f(\widehat{s'}). \tag{11}$$

As well, equation 7 is now defined on the finite set of macro-states $\widehat{S}$:

$$\Delta \left(\sum_{\widehat{s'} \subset S_0} \overline{E}(\widehat{s}) \right) \simeq I_{S_0}(\widehat{s}).\Delta \overline{e}(\widehat{s}).$$

In a continuous state space representation with kd-tree based discretization, the influence indicates whether it might be useful to add new children to a leave or to remove two twin leaves. Indeed, removing or adding leaves has a direct effect on the interpolation error, and this effect is transmitted to the approximation error according to the above equation. As we will see in the experiments of the next subsection, the influence allows to efficiently update the tree in order to minimize the approximation error.

Before we describe some experiments, we now show how to integrate the approximation error and the influence computations in the AN architecture of

figure 2 (page 8). The computations of the approximation error and the influence require to add a few local variables to each unit (or macro-state) :

- e: a float which stands for the interpolation error $e(\widehat{s})$
- E: a float which stands for the approximation error $E(\widehat{s})$
- π_{err} : an element of A which is the most uncertain action $\pi_{err}(\widehat{s})$ (introduced on page 13)
- 1/0: a bit that says whether the unit/macro-state belongs or not to the subset S_0 where one wants to minimize the approximation error
- I: a float which stands for the influence $I_{S_0}(\widehat{s})$

The updated AN architecture is sketched in figure 6. Though the new archi-

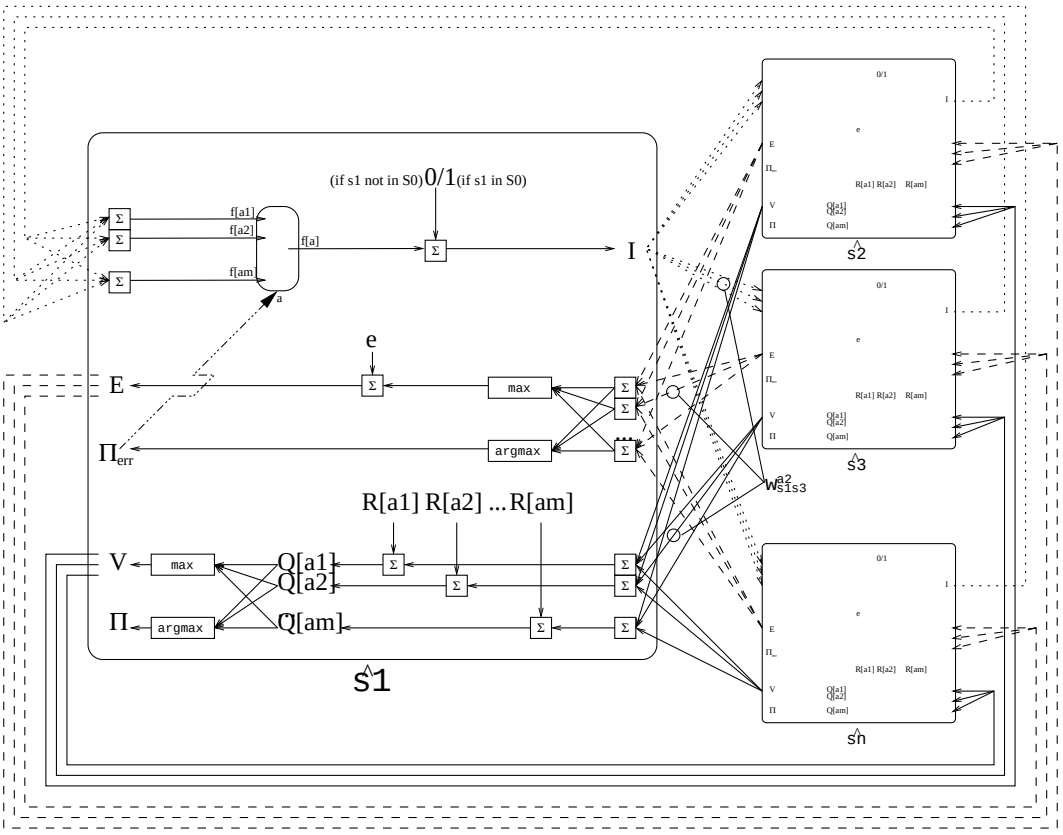

Fig. 6. **An asynchronous neurocomputing architecture for optimal control with large state space.** We focus on one unit/macro-state $\widehat{s_1}$ and show how it updates its internal variables and how it interacts with the other units $\widehat{s_2}$, $\widehat{s_3}$, ..., $\widehat{s_n}$. This architecture is made of three layers. The bottom layer computes the optimal value function, the middle layer the approximation error and the upper layer the influence (see text for more details). The arrows show the flows of information rather than the connections; as indicated by the $w^a_{s_1 s_3}$ legend, several flows share the same network connection.

tecture looks much more complicated, there is nothing in it which is fundamentally different from what we had before. This new architecture can be seen

as made of three layers. The bottom layer computes the optimal control in the approximate model and is identical to what it was in the previous architecture (figure 2). The second layer (involving the variables e, E and π_{err}) enables to compute the approximation error. This computation is similar to the optimal control computation ; it is even a bit simpler as the interpolation error does not depend on the action. Finally, the upper layer is devoted to the computation of the influence and is drawn the other way round because the flow of information is of the opposite direction (compare for instance the $\widehat{T}$ terms in equations 10 and 11). It uses the local variable π_{err} to choose the right flow of incoming information and adds the bit 0/1 in order to update the local variable I. As all these computations are CMFPs, there is no need for synchronization and we are sure that the variables V, E and I will eventually stabilize to the optimal value function, the approximation error and the influence.

To sum up, when an MDP has a large state space, the approximation by state aggregation is particularly interesting because it enables to compute the approximation error and the influence with CMFPs on the approximate aggregate state space $\widehat{S}$. Therefore, all these computations can be done with PADC, and can straight-fully be integrated in the architecture we introduced in section 2.

3.3 Experiments

So far, we have built a theoretically sound AN architecture for optimal control. We illustrate in this last subsection its potential on two continuous space control problems. We begin by describing our experimental protocol.

We use the kd-tree structure to define the aggregation/discretization in the continuous state spaces. For iteratively improving the approximation, we use algorithm 4. Thus, we constrain the number of states on which we need to compute CMFPs and we keep on reorganizing the kd-tree in order to minimize the approximation error. For each problem we are going to describe, we evaluate the evolution of performance, iteration after iteration, using the policies computed by the AN architecture. The performance is here estimated by the average discounted sum of rewards of 500 trajectories of 500 steps from random initial states (we take $S_0 = S$).

Note that this high-level procedure for improving an optimal control approximation somehow makes a split with the previous algorithms in the sense that it is not anymore pure asynchronous neurocomputing. The very update of a kd-tree implies to add and remove units in the architecture, and this has to be done in a centralized way. Nevertheless, most of the processes involved

Algorithm 4 Iterative improvement of a kd-tree aggregation

Choose an initial number of units/macro-states and initialize a regular discretization (with a balanced kd-tree)

loop

1. Estimate $\widehat{T}$, $\widehat{R}$, e (through ΔT and ΔR), Δe^+ (the decrease of interpolation error if we add some leaves) and Δe^- (the increase of interpolation error if we remove some leaves) using samples of R and T
2. Use the AN architecture of figure 6 to compute V, E and I.
3. In the kd-tree, remove the 10% leaves that have the smallest $|I.\Delta e^-|$ scores and replace them under the leaves that have the biggest $|I.\Delta e^+|$ scores.

end loop

(estimating the parameters, computing V, E and I) extensively uses our AN architecture. Though it is not sure whether it would be computationnally interesting, the full integration of large state space optimal control (including the kd-tree adaptive process) in a pure asynchronous neurocomputing architecture will be studied in future works, at least because it might be interesting from a neuroscience point of view.

3.3.1 Experiment 1: mountain-car

The first problem we consider, known as "mountain-car", is typical of the RL literature [18]. We want to optimally control the acceleration of an underpowered car so that it climbs up a steep mountain road. As gravity is stronger than the car's power, this has to gain energy by oscillating in a bassin in order to get to the top of the mountain. Also, this car must pay attention to not get too much speed so that it does not fall into a ravine (see figure 7). By integrating the equations of physics (using the approximation $dt = 1s$), this problem can be described as a 2-dimensional continuous space (x, v) MDP where x is the position of the car ($-1.2 \leq x \leq 0.6$) and v is its speed ($-0.07 \leq v \leq 0.07$). The car only has 2 possible actions: accelerate front ($a = +1$) or backward ($a = -1$). The T function is induced by the following dynamics [18]:

$$\begin{cases} x_{t+1} = x_t + v_t \\ v_{t+1} = v_t + 0.001.a_t - 0.0025.cos(3.x_t) \end{cases}$$

We also add the following constraint: if the car falls in the ravine (if $x \in (-1.2, -1.15)$), then it gets stuck into it. To fully specify the MDP and to characterize the car's goal, one gives a $+1$ reward when it gets on the top of the hill ($x \in (0.55, 0.6)$) and a -1 penalty if it falls in the ravine (if $x \in (-1.2, -1.15)$). All other states have no feed-back (i.e. a 0 reward).

The shape of the optimal value function and some typical optimal trajecto-

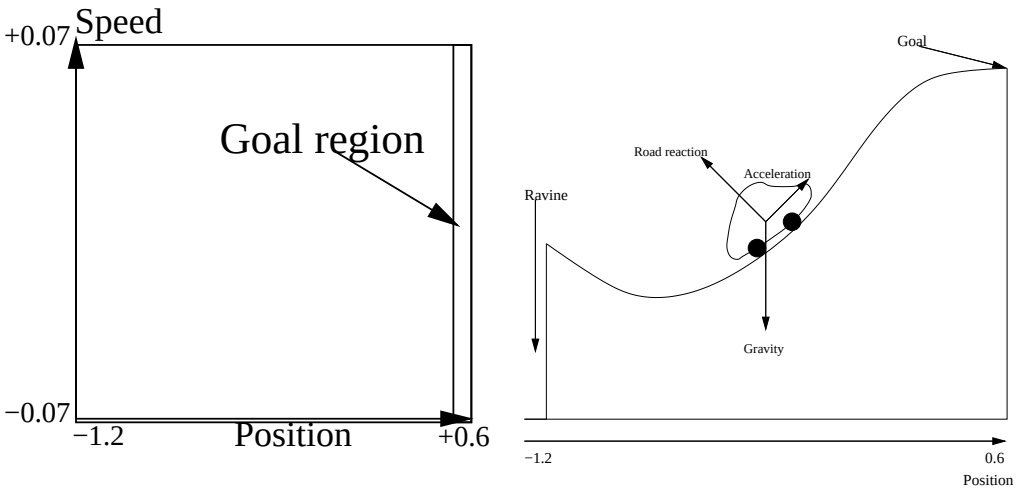

Fig. 7. **The mountain-car problem.** An underpowered car has to climb up a steep mountain road while not falling into a ravine. As gravity is stronger than the car's power, the car has to gain energy by oscillating in a bassin in order to get to the top of the mountain. This problem can be described as a MDP on the 2-dimensional space involving position and speed of the car.

ries can be seen in figure 8. We test the aggregation approximation and the

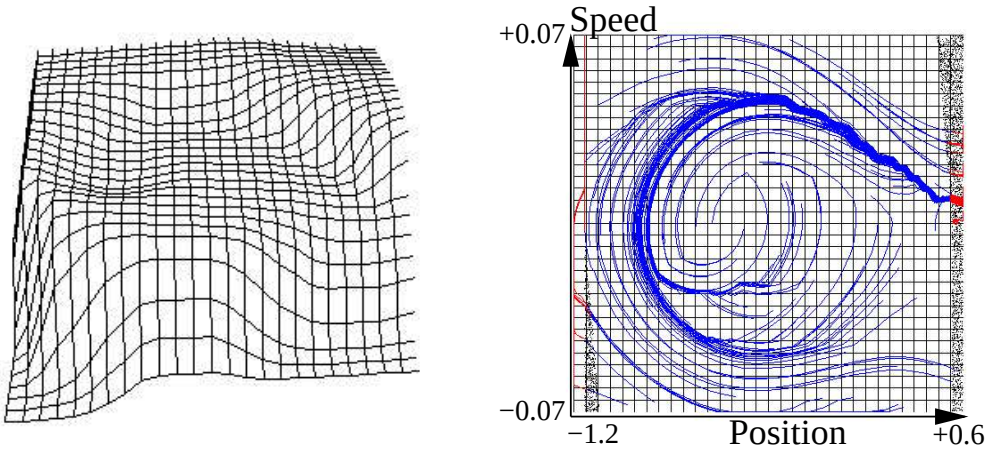

Fig. 8. **Optimal value function and optimal trajectories for the mountain-car problem.** When acting optimally, a typical trajectory of the car consists in going away from the goal to gain energy then getting to the top of the hill.

kd-tree refining procedure we described in the beginning of this subsection, using a 256-state approximate model. Figure 9 shows the evolution of performance along the iterative refinement of the kd-tree discretization and the eventual discretization. We observe that, though the number of units/states (and therefore the computational complexity) stays constant, the update of the kd-tree/discretization allows to iteratively improve the performance. The eventual discretization (the one which has the best performance) describes precisely the regions encountered while following the optimal policy (compare

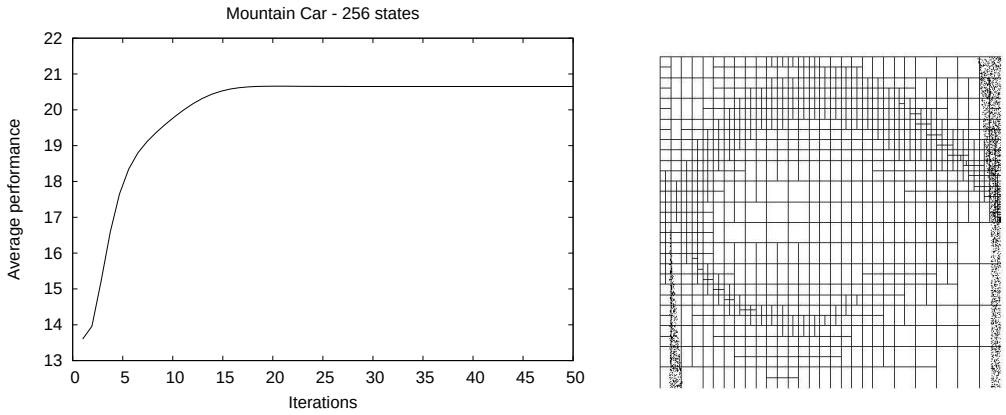

Fig. 9. **Performance evolution and final discretization for the mountain-car problem.** Iteratively refining the discretization allows to improve the performance while keeping the complexity constant. The final discretization is precise along the optimal trajectories.

figures 8 and 9).

3.3.2 Experiment 2: 5-dimensional car driving

In order to further validate our approach, we made experiments on a more difficult control problem which is inspired by autonomous mobile robotics [17]. We consider an MDP describing a car-like driving vehicle evolving in a C-shape

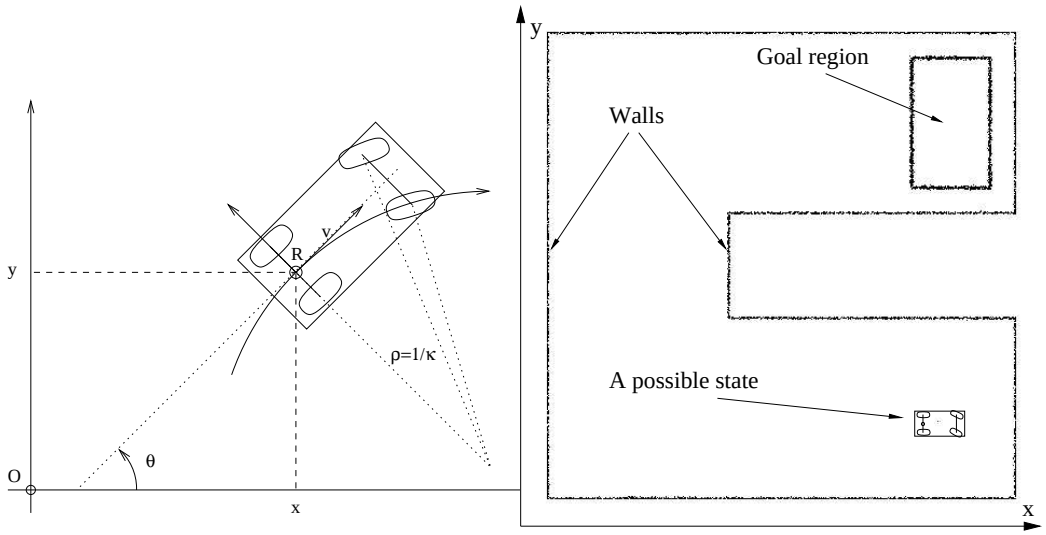

Fig. 10. **Features of the car driving dynamics and C-shape environment in which it is embedded.**

environment (see figure 10). The state is a 5-dimensional vector $(x, y, \theta, \kappa, v, a)$:

- x and y are the coordinates of a reference point R on the car which is in

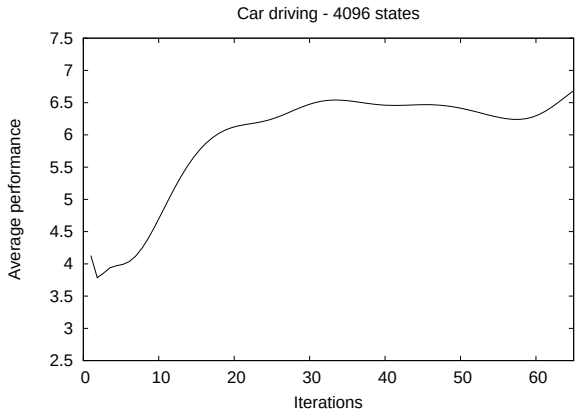

Fig. 11. **Performance evolution for the car driving problem.** Iteratively refining the discretization allows to improve the performance while keeping the complexity constant.

between the rear wheels
- θ is the car orientation (the angle with the axis (Ox))
- κ is the instantaneous curvature (i.e. the inverse of the turning circle)
- v is the linear speed of the car.

The possible actions are:

- the instantaneous acceleration $a \in \{-1; 0; +2.5\}$ and
- the instantaneous rotation wheel $\Delta\kappa \in \{-0.2; 0; +0.2\}$

9 composite actions $(a, \Delta\kappa)$ allow to control acceleration and wheel rotation at the same time. The dynamics of the problem is sum up in the following laws (which approximate the continuous time dynamics using $dt = 1s$):

$$\begin{cases} x_{t+1} \leftarrow x_t + v_t.\cos\theta_t \\ y_{t+1} \leftarrow y_t + v_t.\sin\theta_t \\ v_{t+1} \leftarrow v_t + a \\ \theta_{t+1} \leftarrow v.\kappa_t \\ \kappa_{t+1} \leftarrow \kappa_t + \Delta\kappa \end{cases}$$

We also add the following realistic constraints: $|\kappa| \leq \kappa_{max} = 0.2$, $\frac{\partial\kappa}{\partial t} \leq \sigma_{max} = 0.2$, $v \in (-2; 5)$, and of course the vehicle cannot go through the walls of the environment. Eventually, we give a $+1$ reward if the car gets to the goal region shown in figure 10, and 0 reward else.

For this problem, we use the state aggregation approximation with a 4096-leave kd-tree and start from a regular discretization. Figure 11 shows that performance increases along the iterative refinement of the discretization. As

it is hard to represent a 5-dimensional space, figure 12 shows (instead of a final discretization) some examples of trajectories at different stages of the discretization update process. Confirming the increase of performance, we visually observe that trajectories keep on improving: they get shorter and shorter, and the vehicle hits less and less the walls. Once again, our algorithm allows to improve the performance, even though we constrain the complexity (the number of states).

4 Conclusions

In this article, we recall a fundamental theoretical result: any mathematical function that can be characterized as a contraction mapping fixed point can be efficiently estimated with parallel asynchronous distributed computations. By making the link between global and local computations (section 1), such a result constitutes an interesting basis for constructing fully-understandable asynchronous neurocomputing architectures.

Indeed, we exploit this result several times in order to build asynchronous neurocomputing architectures for solving optimal control in a small state space, reinforcement learning in small state space (section 2), and eventually for quantitatively analyzing an approximation in the large state space case (section 3). Comparatively to other approaches for control, these architectures are derived from the general control formalism of MDPs. They are therefore completely *generic* (they can be applied to any problem which can be formulated as a MDP). Also, the computations we describe are mathematically sound. In section 2, algorithms are known to converge to the optimal solutions; in section 3, the refinement procedure of a discretization is based on a gradient descent of the real approximation error (the distance between the current solution and the optimal solution). We demonstrate the potential of this approximation scheme on two complex control problems: the "mountain-car" and a 5-dimensional realistic car driving problem (more details about the problems we considered and many other experiments can be found in [15]).

The computations we describe for solving difficult control problems almost entirely rely on asynchronous neurocomputing architectures. From a computational point of view, such neural architectures may benefit from their massively parallel structure to outperform usual digital computing. From a neuroscience point of view, we hope that the tight relations that we stressed between the optimal control problems and neurocomputing, and particularly some analogies between our mathematically motivated computations and some biologically inspired models as *call propagation* (section 2.2) and *hebbian law* (section 2.3), will help understand better how the brain elaborates decisions.

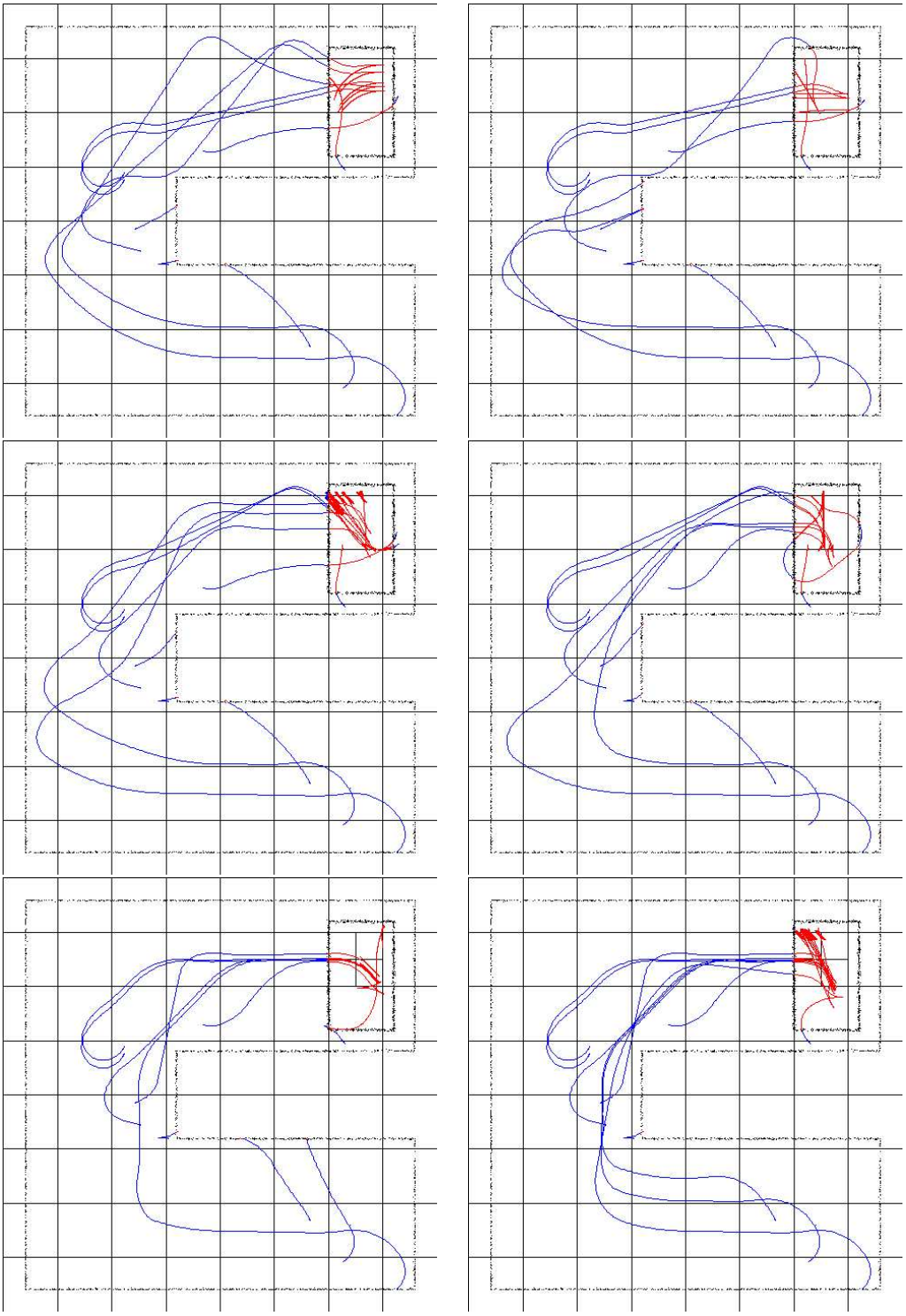

Fig. 12. **Examples of trajectory evolution for the complex car problem while updating the discretization.** We randomely chose 10 initial state position and we show how the trajectories starting from them evolve during the discretization refinement procedure. We observe that trajectories are shorter and shorter, and that they hit less and less the walls.

Aknowledgement

This work was supported by an INRIA Grant.

References

[1] D.P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice hall, 1989.

[2] Y. Burnod. *An adaptive neural network : the cerebral cortex*. Masson, Paris, 1989.

[3] F. Fleuret and E. Brunet. DEA: An architecture for goal planning and classification. *Neural Computation*, 12(9):1987–2008, 2000.

[4] H. Frezza-Buet. *Un modèle de cortex pour le comportement motivé d'un agent neuromimétique autonome*. PhD thesis, Université Henri Poincaré - Nancy 1, 1999.

[5] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *TOMS*, 3(3):209–226, 1977.

[6] B. Fritzke. Some competitive learning methods, 1997.

[7] V. Gullapalli and A. G. Barto. Convergence of indirect adaptive asynchronous value iteration algorithms. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 695–702. Morgan Kaufmann Publishers, Inc., 1994.

[8] D.O. Hebb. *The organization of behavior: a neuropsychological theory*. John Wiley & Sons, New York, 1949.

[9] L. P. Kaelbling, M. L. Littman, and A. P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[10] M. L. Littman, T. L. Dean, and L. P. Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)*, pages 394–402, Montreal, Québec, Canada, 1995.

[11] W.S McCulloch and W.P Pitts. A logical calculus in the ideas immanent in nerveous activity. *Bulletin of Mathematical Biophysics*, 1943.

[12] R. Munos and A. Moore. Rates of convergence for variable resolution schemes in optimal control. In *International Conference on Machine Learning*, 2000.

[13] M. Puterman. *Markov Decision Processes*. Wiley, New York, 1994.

[14] N. Rougier. *Modèles de mémoire pour la navigation autonome*. PhD thesis, Université Henri Poincaré - Nancy 1, 2000.

[15] B. Scherrer. *Apprentissage de représentation et auto-organisation modulaire pour un agent autonome*. PhD thesis, Université Henri Poincaré - Nancy 1, January 2003.

[16] B. Scherrer. Parallel asynchronous distributed computations of optimal control in large state space markov decision processes. In *European Symposium on Artificial Neural Networks*, Bruges, Belgique, April 2003.

[17] A. Scheuer. *Planification de chemins à courbure continue.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.

[18] R.S. Sutton and A.G. Barto. *Reinforcement Learning, An introduction.* BradFord Book. The MIT Press, 1998.