Sparse Distributed Memories in a Bounded Metric State Space: Some Theoretical and Empirical Results.

Alexandre Bouchard-Côté
School of Computer Science
McGill University
alexandre dot bouchard @ mail.mcgill.ca

September 7, 2004

Abstract

Sparse Distributed Memories (SDM) [7] is a linear, local function approximation architecture that can be used to represent cost-to-go or state-action value functions of reinforcement learning (RL) problems. It offers a possibility to reconcile the convergence guarantees of linear approximators and the potential to scale to higher dimensionality typically exclusive to nonlinear architectures. We investigated both avenues to see if SDM can fulfill its promises in the context of a RL problem with a bounded metric state space. On the theoretical side, we found that using properties of interpolative approximators provided by [3], it can be proven that a version of Q-learning converges with probability one when it is used with a SDM architecture. On the other hand, an important non-divergence result on the SARSA algorithm, an approximate, optimistic policy iteration algorithm, failed to apply in our case because of our loose assumptions on the nature of the state space. However, one of the most important convergence result in reinforcement learning [6], the convergence of TD(λ) was successfully translated into the language of measure theory to cover the case of Markov processes with a general probability state space. This forms the foundation for an eventual proof of convergence of an approximate policy iteration algorithm. On the empirical side, the first step was to design and implement a specialized data structure to store and retrieve "hard locations" (basis points). The specifications of this hash-based data structure will be discussed, as well as statistics on the performances of SDM equipped with this data structure. We conclude by presenting potential extensions to SDM. Their motivations and drawbacks will be examined, focusing on the implications related to the hard location storage data structure. These results form our basic toolbox for our current work targeting a general-purpose, "tweaking-free" reinforcement learning algorithm with flexible assumptions on the space, convergence guarantees and high-dimensional scalability.

Contents

1 Introduction

The study of SDM as a value function approximator for reinforcement learning is motivated by a well know duality problem between the convergence guarantees of linear architectures and the dimensional scalability of nonlinear approximation architectures. This architecture was proposed by Kanerva in 1993 [7], used in RL for the first time by Sutton and Barto in 1998 and revived by Doina Precup and Bohdana Ratitch in 2003. This paper is a collection of many theoretical and empirical results related to SDM in the context of a bounded metric state space. They form our basic toolbox for our current work targeting a general-purpose, "tweaking-free" reinforcement learning algorithm with flexible assumptions on the space, convergence guarantees and high-dimensional scalability.

1.1 Reinforcement Learning

We are interested in a standard reinforcement learning task (S, A, K, g, γ), S being the state space, A, the finite set of actions and each element of this set being bijectively associated to a probability transition kernel in $K := \{Ker_a(\cdot,\cdot)\}_{a\in A}$. We will use the terminology *policy* [1] ($\mathfrak{P}$) for sequences of maps from S to A and *stationary policy* for constant sequences of such maps. Note that given a starting state x_0 and a policy $\mathfrak{p} := (\mathfrak{p}_i)$, the environment becomes markovian by letting the state x_{i+1} be distributed according to $Ker_{\mathfrak{p}_i(x_i)}(x_i, \cdot)$, given that the previous state was x_i. Denoting by $E_{\mathfrak{p}}[\cdot|x_0]$ the expectation taken with respect to this Markov chain, we use the *reward function* $g : S \times A \times S \to \mathbb{R}$ and the discount factor γ to define an ordering on the policies, given by comparing:

$$\liminf_{N\to\infty} E_{\mathfrak{p}}\left[\sum_{i=0}^{N} \gamma^i g(x_i, \mathfrak{p}_i(x_i), x_{i+1})\Big| x_0\right].$$

We call this quantity the *cost-to-go function* associated to policy $\mathfrak{p}$ and we denote it by $J_{\mathfrak{p}}(x_0)$. The goal is to find the optimal policy, or equivalently, the *optimal cost-to-go function* [2]:

$$J^\star(x) := \min_{\mathfrak{p}\in\mathfrak{P}} J_{\mathfrak{p}}(x).$$

[1] *Stochastic policies*, that map state to probability distributions on A will also be used, and the RL problem can be stated with very little changes to take them into account.

[2] Also equivalently, optimal state-action value functions can be targeted (e.g. when there is no model for the environement available):

$$Q^\star(x, a) := \int_S Ker_a(x, dy)(g(x, a, y) + J^\star(y)).$$

1.2 The SDM architecture

Assume that the state space is a bounded metric space [3]. A SDM architecture contains:

- A *similarity function* or *similarity*

$$\sigma : S \times S \to [0, 1]$$

  such that $(1 - \sigma)$ is a metric on S (such a function exists iff S is a metric space),

- A set H of *hard locations* or *basis points*

$$\left\{s_1, \cdots, s_K : s_i \in S\right\}$$

  such that every hard location is associated with a scalar weight w_i. The sequence $\vec{w} := (w_i)_{i=1}^K$ forms the parameters vector of the architecture.

When an evaluation $\tilde{f}(s)$ is required for a given point $s \in S$, the first step is to find the set H_s of *active locations*, that is, the set of hard locations with a positive similarity with s. The estimate at s is then compute using the simple rule that the weights of the active locations that are closer to s should have more impact on the approximation of the value at s. Formally:

$$\tilde{f}_{\vec{w}}(s) := \frac{\sum_{s_k \in H_s} \sigma(s_k, s) w_k}{\sum_{s_k \in H_s} \sigma(s_k, s)}.$$

Training can be done using a standard gradient descent, and note that the gradient takes this simple form:

$$(\nabla \tilde{f}_{\vec{w}}(s))_i := \frac{\sigma(s_i, s)}{\sum_{s_k \in H_s} \sigma(s_k, s)}.$$

From this definition, two important observations can be made:

- For a fixed state, the gradient is a constant function of $\vec{w}$: SDM is a linear approximation architecture. We are therefore in a good position to get convergence guarantees with standard RL algorithms. This is what we will discuss in section 2.
- The density of the hard locations across the space need not be constant: we can use a mechanism that builds the set of hard locations so that "important" regions of the state space are more densely covered by hard locations [4]. This is why we think that SDM can "break the curse of dimensionality", and we will discuss how it can be implemented in practice in section 3.

[3] Note that we avoid the term "continuous space". Although common in RL literature, it is often left undefined and the closest mathematical concept that we know, the notion of *continuum* (a compact connected set), does not correspond to the kind of hypothesis that we are looking for.

[4] Precup and Ratitch provide an example of such a dynamic allocation algorithm [2].

2 Theoretical Results

The converge results on RL algorithms using SDM are quite promising, as it was expected, and can be split into two categories. The first category deals with value iteration techniques and is largely based on the work of Szepesvári and Smart[3] on interpolative linear approximation architectures. The second category concentrates on methods related to policy iteration. The main result of this latter category, the convergence of TD(λ), is essentially a translation into the language of measure theory of the celebrated result by Bertsekas and Van Roy [6], but is more general in its assumptions on the state space (in particular, closer to the type of state space that we are interested in our framework).

2.1 Value Iteration Methods

Approximate value iteration algorithms such as Q-learning play an important role in RL, independently of their poor convergence guarantees. Even a linear approximator hypothesis is not sufficient, for there is an example of divergence of Q-learning with a linear approximation architecture. There is, however, a special convergence result that applies in the case of *interpolative* SDM's.

An SDM is said to be interpolative if, for any basis point, the approximator evaluated at this point equals to the weight carried by the basis point. From this definition, it is easy to see that a SDM with a local similarity function (e.g. symmetric triangular similarity functions) can be made into an interpolative SDM simply by requiring the centers not to be activated by other hard locations.

The main result that apply for approximate value algorithms using SDM's assumes the following update rule [5]:

$$\vec{w}_{a,t+1} := \vec{w}_{a,t} + \alpha_t \nabla(\tilde{f}_a)_{\vec{w}_{a,t}} \Big(g(x_t, a_t, x_{t+1}) + \gamma \max_{b \in A} (\tilde{f}_b)_{\vec{w}_{b,t}}(x_{t+1}) - (\tilde{f}_a)_{\vec{w}_{a,t}}(x_t) \Big),$$

where $\{\tilde{f}_a\}_{a \in A}$ is a collection of interpolative SDM's that represent state-action values, $\vec{w}_{a,t}$ are the corresponding parameter vectors at iteration t, (α_t) is the step-size sequence, and (x_t, a_t, x_{t+1}) is the observed transition generated by a stochastic exploration policy. The important assumptions [6] include:

- S must be a *Polish space* (the homeomorphic image of a complete, separable metric space),
- $\tilde{f}_a$, seen as a map $\mathbb{R}^K \to B(S)$ (we will denote the space of bounded function on X by $B(X)$), must be a non-expansion,
- the stochastic exploration policy must be stationary.

[5] The original result by Szepesvári and Smart is actually more general and the reader should refer to the relevant paper [3] if different update rules are needed (e.g. with eligibility traces).

[6] Again, the reader should refer to [3] for the complete list of assumptions. The paper also contains an extension of great interest for us, that takes into account SDM with a set of hard locations that is constructed dynamically.

The two first items fit very well in our framework, but the third one is much more restrictive. Indeed, except in the case that a suboptimal exploration policy or a partially-tuned value function is available, it makes the exploration of the space very slow.

The proof idea is to reduce the problem of the convergence of the approximate value iteration algorithm to the simpler case of a tabular algorithm (the finite set of states of this subproblem being the set of basis points). This is done using the interpolative non-expansion property of the approximation architecture. The stationary assumption on the exploration policy is used to prove the convergence of the subproblem. A potential way to relax the third condition could be to reduce the convergence to a different subproblem that would not require it. Another planned future work related to this result is to test empirically if the interpolative property actually improves the behavior of the algorithm or if it is only an artifact needed by the proof of convergence.

2.2 Policy Iteration Methods

In this section, we look closely at some existing convergence results for policy iteration methods using linear approximation architectures. The main difference between those and the algorithms we want to consider is on the assumptions on the state space. Indeed, most existing results assume either a finite state space or a countable state space equipped with a counting measure [7]. We are rather interested in *general state space* (that is, equipped with a countably generated σ-field) because we would like to assign probabilities to intervals, for instance, instead to points only [8].

The first result we considered is a non-divergence proof of SARSA under certain assumptions given by Gordon [5]. It uses the following facts to build a region of convergence:

- If the policy were not changed at each iteration (SARSA is an optimistic policy iteration algorithm), the weights would converge to a fixed point by the convergence of the policy evaluation algorithm TD(λ),
- For a fixed ϵ, there are finitely many ϵ-greedy policies the algorithm can consider.

We see that the second argument clearly does not hold for an arbitrary metric space (it could be infinite). However, the failure of a proof is not a proof of failure, and there is still a lot of research to be done on

[7] In particular, in the Neuro-Dynamic Programming Book [4], Bertsekas and Tsitsikilis claim that the "continuous case" is treated in [6], but the latter reference is restricted to the case of a state space with a counting measure.

[8] One may argue that it is futile to do this extension, for Turing machine cannot represent elements of uncountable sets in general. We think that this approach is still worthy because higher abstraction often play an important role to progress in mathematics. Also, in a more pragmatic point of view, we will see shortly that it eliminates convergence results that depends on the cardinality of S and that would give unrealistically big bounds for very large state spaces.

the analysis of the asymptotic behavior of optimistic approximate policy iteration algorithms [9].

On the other hand, the first argument (the convergence of TD(λ)) can be translated into the language of measure theory to cover the case of a general state space:

Theorem 1. *Consider the sequence generated by the recurrence:*

$$\vec{w}_{t+1} := \vec{w}_t + \alpha_t d_t \vec{z}_t$$

where:

$$d_t := g(x_t, x_{t+1}) + \gamma \tilde{J}_{\vec{w}_t}(x_{t+1}) - \tilde{J}_{\vec{w}_t}(x_t)$$

is the temporal difference *computed as soon as the transition generated by the stationary policy* $\mathfrak{p}$ [10] *is observed* [11]*,*

$$\tilde{J}_{\vec{w}}(x) := \sum_{k=1}^{K} (\vec{w})_k \phi_k(x)$$

is the linear approximation architecture with basis functions ϕ_k *and*

$$\vec{z}_t := \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \nabla \tilde{\vec{J}_{\vec{w}}}(x_k) \tag{1}$$

$$= \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \vec{\phi}(x_k) \tag{2}$$

$$= \gamma\lambda \vec{z}_t + \vec{\phi}(x_{t+1}) \tag{3}$$

is the eligibility vector *with* $\lambda \in [0,1]$ *arbitrary but fixed and* $\vec{\phi}(x) := (\phi_i(x))_{i=1}^K$.

The sequence $(\vec{w}_t)$ *converges with probability one, assuming that:*

1. *the stepsize* α_t *are positive, deterministic and satisfy* $\sum_{t=0}^{\infty} \gamma_t = \infty$ *and* $\sum_{t=0}^{\infty} \gamma_t^2 < \infty$*,*
2. *the Markov chain induced by* $\mathfrak{p}$ *has an invariant measure* π *on the countably generated* σ*-field of* S
3. *the basis functions are linearly independent,*
4. *the expected value of* $A(X_t)$ *converges at least exponentially fast in Euclidean matrix norms, where:*

$$A(X_t) := \vec{z}_t \big(\gamma \vec{\phi}(x_{t+1})^T - \vec{\phi}(x_t)^T\big).$$

To prove this theorem, we define, as Bertsekas and Tsitsiklis, a special *weighted quadratic norm* using the invariant measure and start by

[9] A good characterization of the asymptotic behavior of SARSA is considered as one of the main open problems in RL.

[10] The result can be easily extended for ϵ-soft policies.

[11] Note that the dependance on the policy is removed from the notation since the policy is stationary and fixed.

establishing some properties of this norm [12]. For a measurable function $f : S \to \mathbb{R}$, put:

$$\|f\|_\pi^2 := \int_S (f(x))^2 \pi(dx).$$

We will now prove that the properties needed by the proof of the main theorem still hold with uncountable space, and the main proof itself rely on the same idea as [6], but is more messy and will not be included.

Lemma 1. *If $f : S \to \mathbb{R}$ is a measurable function, then:*

$$\Big\| \int_S f(y) Ker(\cdot, dy) \Big\|_\pi \leq \|f\|_\pi.$$

Proof. We have, by Jensen's inequality:

$$\Big\| \int_S f(y) Ker(\cdot, dy) \Big\|_\pi^2 = \int_S \pi(dx) \Big(\int_S Ker(x, dy) f(y) \Big)^2 \tag{4}$$

$$\leq \int_S \pi(dx) \int_S Ker(x, dy) (f(y))^2. \tag{5}$$

Now, using Fubini Theorem and the definition of an invariant measure and of a $\| \cdot \|_\pi$ norm, we get:

$$= \int_S \int_S \pi(dx) Ker(x, dy) (f(y)))^2 \tag{6}$$

$$= \int_S \pi(dy) (f(y))^2 \tag{7}$$

$$= \|f\|_\pi^2. \tag{8}$$

□

Lemma 2.

$$\Big\| \int_S f(y) Ker^m(\cdot, dy) \Big\|_\pi \leq \|f\|_\pi.$$

Proof. We shall proceed inductively on m. The base case is given by lemma 1. Suppose that the claim is true up to m. We then have:

$$\Big\| \int_S f(y) Ker^m(\cdot, dy) \Big\|_\pi^2 = \Big\| \int_S f(y) \int_S Ker(\cdot, dz) P^{m-1}(z, dy) \Big\|_\pi^2 \tag{9}$$

$$= \Big\| \int_S Ker(\cdot, dz) \int_S f(y) Ker^{m-1}(z, dy) \Big\|_\pi^2 \tag{10}$$

$$\leq \Big\| \int_S f(y) Ker^{m-1}(\cdot, dy) \Big\|_\pi^2 \tag{11}$$

$$\leq \|f\|_\pi^2. \tag{12}$$

□

[12] One easily checks that it is indeed a norm.

Lemma 3.

$$\Big\|(1-\lambda)\sum_{m=0}^{\infty}\lambda^m\gamma^{m+1}\int_S Ker^m(\cdot,dy)f(y)\Big\|_\pi^2 = \Big\|M(\cdot,dy)f(y)\Big\|_\pi^2 \tag{13}$$

$$\leq \frac{\gamma(1-\lambda)}{1-\lambda\gamma}\|f\|_\pi^2, \tag{14}$$

where:

$$M(x,A) := (1-\lambda)\sum_{m=0}^{\infty}\lambda^m\gamma^{m+1}Ker^{m+1}(x,A).$$

Proof. Using the previous lemma and triangle inequality:

$$\Big\|(1-\lambda)\sum_{m=0}^{\infty}\lambda^m\gamma^{m+1}\int_S Ker^m(\cdot,dy)f(y)\Big\|_\pi^2 \leq (1-\lambda)\gamma\sum_{m=0}^{\infty}\Big\|\int_S Ker^m(\cdot,dy)f(y)\Big\|_\pi^2 \tag{15}$$

$$\leq \frac{\gamma(1-\lambda)}{1-\lambda\gamma}\|f\|_\pi^2. \tag{16}$$

□

Lemma 4.

$$\int_S \pi(dx)\int_S M(x,dy)(f(x)f(y)) \leq \alpha\|f\|_\pi^2.$$

Proof. Using the Cauchy inequality and the previous lemmas:

$$\int_S \pi(dx)\int_S M(x,dy)(f(x)f(y)) = \int_S f(x)\int_S M(x,dy)f(y)\pi(dx) \tag{17}$$

$$= \Big(\int_S (f(x))^2\pi(dx)\Big)^{\frac{1}{2}}\Big(\int_S M(x,dy)f(y)\Big)^{\frac{1}{2}} \tag{18}$$

$$= \|f\|_\pi\Big\|\int M(\cdot,dy)f(y)\Big\|_\pi \tag{19}$$

$$\leq \|f\|_\pi\frac{\gamma(1-\lambda)}{1-\gamma\lambda}\|f\|_\pi \tag{20}$$

$$\leq \|f\|_\pi^2. \tag{21}$$

□

3 Empirical Results

As we mentioned in the introduction, we hope that it is possible to use the first phases of learning to construct the set of hard locations so that it has desirable properties (by using a dynamic memory allocation algorithm). However, this algorithm involves a large amount of insertion and deletion of hard locations. This fact explains the need for a specialized data structure to index efficiently the hard locations. More precisely, the operations that need to be optimized are:

- *Find potentially active locations* Find the potentially active locations for a given point in the state space.
- *Add a hard location* Given a location in the state space, add a new hard location at that point (if no hard location is already at that position).
- *Delete a hard location* Given the location of a hard location in the state space, remove the hard location from the data structure (if such exists).

The post-condition for *find* is that it should return a set covering the set of active locations corresponding to the given state. For obvious performance reasons, we also expect this cover to be of cardinality close to the cardinality of the set of active locations. The first part of this section, covering a list-intersection and a hash method, concentrates on interpolative SDM's in a finite dimensional vector space. The second part discusses variations on the definition of SDM's, focusing on the implications on the hard location data structure.

3.1 Active Locations Search for Interpolative SDM's

We assume in this section that the set of states S is a bounded subset of a finite dimensional vector space V. Let σ be a similarity function on S such that $\{(\sigma, H_\alpha)\}$ is the collection of interpolative SDM architectures that can be generated by a fixed dynamic memory allocation algorithm. Suppose also without loss of generality that V is a n-affine F vector space with orthogonal coordinates such that $M \subseteq V$ is a hypercube with unit edges parallel to the axis that encloses S. Observe that by the interpolative property of the generated SDM's, there is a δ such that:

$$d(x_1, x_2) \geq \delta \Rightarrow \sigma(x_1, x_2) = 0 \quad \forall H_\alpha \; \forall x_1, x_2 \in H_\alpha.$$

We will call the smallest such scalar the *activation radius* and we will denote it from now on by δ. Now by the symmetry of σ, we see that we have reduced the problem of finding the set of activated location to a standard range search problem of radius δ.

The existing code, written by Bohdana Ratitch, uses the following strategy to organize the storage of the locations: each dimension i is partitioned into intervals $I_1^i, \cdots, I_m^i$ of length δ. For each such interval, the sets t_j^i of locations with center in $\{\vec{y} : (\vec{y})_i \in I_j^i\}$ are maintained. On input $\vec{x}$, we find the set of intervals $\{I_{j_i}^i\}_{i=1}^n$ such that for all dimensions,

$(\vec{x})_i \in I^i_{j_i}$. Since

$$P := \bigcap_{i=1}^{n} (t^i_{j_i-1} \cup t^i_{j_i} \cup t^i_{j_i+1}) \supseteq H_{\vec{x}}$$

and that this set P of potentially active locations is usually much smaller than the whole set of hard locations H_α, this gives a relatively more efficient way of finding $H_{\vec{x}}$ than testing all the elements of H_α. Unfortunately, computing the above intersection can be time consuming and we will see that we can achieve better performances using an alternative method to store the locations.

3.2 Better Performances Using a Hash

Two alternative to list-intersection were considered: kd-tree-based and hash-based. We selected the latter because it uses the extra information available that *all the range search queries will have the same radius*, and we will see shortly how this hash-based method partitions the space in a way that is optimal for this type of queries.

The general idea for the hash-based method is to partition the space into a finite number of hyper-cubic cells with edges parallel to the coordinate axis. We want the cells to be the large enough, so that the hash does not take too much memory space, but we also want the edges of the cells to be no larger than 2δ for reasons that will become clear when the algorithm for finding activated locations will be explained. So we pick this optimal 2δ for the length of the edges of the partition cells. We index those cells and use this index as the key for insertion and retrieval in a hash table. In order to do that we define three auxiliary maps. First, a map $\pi : C \to \{cells\}$ that gives the cell containing the given point in the space C. Second, we assume that there is an injective map $\varphi : \{cells\} \to \{finite\_strings\}$ that gives a unique string representation for any cell, so that it can be used by standard hashing algorithms. The private method *HLsupport::computeKey(·)* in the class HLsupport.cpp (see Annex) computes the composition map $\varphi \circ \pi$ and provides additional comments on implementation details. Finally, we need a set map $\Xi : C \to 2^{\{cells\}}$ that returns, for any $\vec{x} \in C$, the set $\{\xi_i\}$ of all hyper-cubic cells (partitions) such that:

$$\xi_i \cap [(\vec{x})_1 - \delta, (\vec{x})_1 + \delta] \times \cdots \times [(\vec{x})_n - \delta, (\vec{x})_n + \delta] \neq \emptyset.$$

Note that the returned set has a cardinality of at most 2^n. (this is why we pick cells with edges of length 2δ: if the cells were bigger, the returned set would still have cardinality of at most 2^n but the union of the cells in this set would have a bigger volume, and more potentially active locations would have to be examined). Again, refer to the annex for additional comments on the implementation of the map $\varphi \circ \Xi$ in *HLsupport::getNeighborCubesRepresentatives(·)* where φ is applied elementwise on $\Xi(\cdot)$. With the above definitions, we can now state the algorithms in pseudo-code for insertion, retrieval and deletion of locations. From now on, *index* will refer to an identifier to a location and *state*, to a point in C.

INSERT($state, index$)

```
1  key ← φ ∘ π(state)
2  list ← hashMap.get(key)
3  if list = NIL
4     then list ← new List
5          list.insert(index)
6          hashMap.insert(list)
7     else if index ∉ list
8             then list.insert(index)
```

DELETE($state, index$)

```
1  key ← φ ∘ π(state)
2  list ← hashMap.get(key)
3  list.delete(index)
4  if list.isEmpty()
5     then hashMap.delete(key)
```

FIND-POTENTIALLY-ACTIVE-LOCATIONS($state$)

```
1  listOfCubes ← Ξ(state)
2  returnedList ← new List
3  for each cube ∈ listOfCubes
4        do key ← φ(cube)
5           list ← hashMap.get(key)
6           returnedList ← returnedList ∪ list
7  return returnedList
```

The method FIND-POTENTIALLY-ACTIVE-LOCATIONS returns a set covering H_{state} so it is easy from that to find the set H_{state}.

These algorithms were implemented in c++ and they correspond to the code for the class HLsupport in the SDM project. The hash table used is the one designed by sgi in their free distribution of STL [1]. The first step to verify correctness was to run low-level tests on the different methods constituting the class HLsupport. Those tests can be found in the file HLsupport.cpp. The second step was to fix the seed for the randomized functions in SDM's code and compare the output of the new implementation against the reference implementation. The fact that they behaved identically on both a supervised learning task and on the mountain-car domain suggests that the new implementation introduces no new bugs in the source code. An additional test to do would be to use a memory leak detector on the current code to insure that it is free of this type of problem.

In terms of speed, the delete and insert operations are not problematic since they only involve the insertion/deletion of the center of the location in a constant number of data structures. The speed bottleneck for the retrieval operation in both data structures is the number of locations processed by the returned List in its life cycle. First, each element in the list of potentially active locations returned by *HLsupport::potentiallyActiveSet(·)* will be used individually so it gives already $O(N' \cdot 3^n)$ and $O(N' \cdot 2^n)$ *List::find(·)*'s for respectively the old and new implementations, where N' is a parameter in our architecture that represents the maximum number

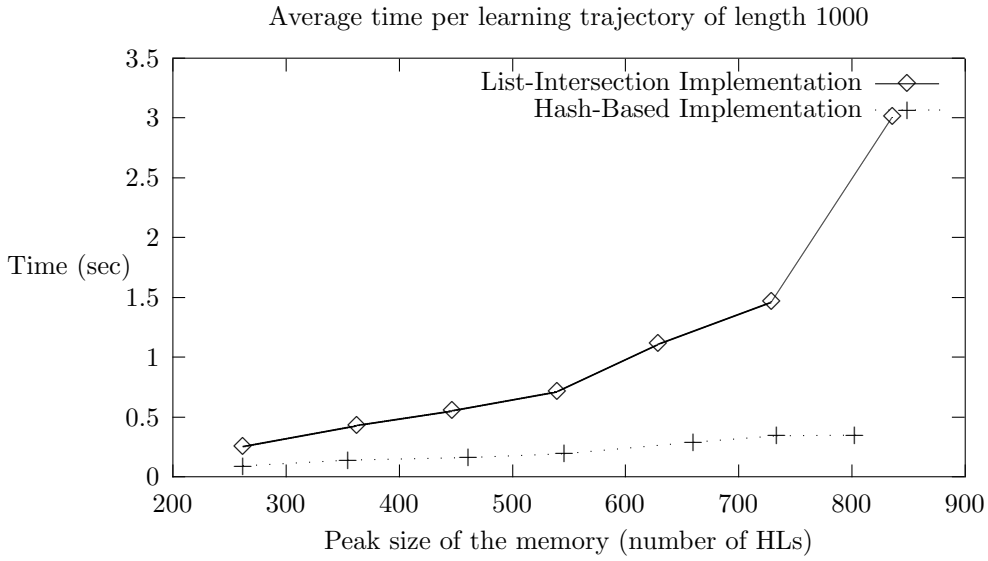

Figure 1: The comparative time required for executing a learning trajectory of length 1000 with different memory densities (averaged over the first 2500 trials of learning of three runs).

of activated locations targeted by the SDM architecture. In the case of the new implementation this is an upper bound since the other operations (in particular, searching the hash table) are constant. On the other hand, the calls of *List::listIntersection(·)* in the reference implementation can be costly, up to $O(\frac{3N'}{\delta})$. However, since the actual distribution of the locations is not uniform in general in SDMs, it is hard to evaluate an expected running time, unless a particular problem instance is fixed. This observation motivates the use of empirical methods to estimate and compare the expected running times.

Our first benchmark was the mountain-car domain, in which the new implementation outperformed the old one both asymptotically and in every test. The current candidate for the next benchmark is the hunter-prey domain, because it can be made more memory-intensive easily by increasing the number of hunters.

3.3 Extensions to SDM

In this subsection we discuss potential variations on the definition of SDM that could improve the behavior of this architecture when used with high dimensional problems. More precisely, we look for modifications that would decrease the memory requirements while keeping as much as possible the expressivity and the convergence qualities of the architecture.

The first variation assumes a sequence of *decreasing* similarity func-

tions (σ_i) instead of a unique similarity function over the experiment. By that we mean that for i-th evaluation or training of the SDM, the similarity function that will be used is σ_i and that for any $x_1, x_2 \in S$,

$$i < j \Rightarrow \sigma_i(x_1, x_2) \geq \sigma_j(x_1, x_2).$$

The motivation to using a sequence of decreasing similarity functions is to get both the fast learning of large radius SDM's and the good asymptotic quality of the approximation of small radius SDM's. These two objectives (speed of learning and quality of the solution) are often competing in applications using constant radius SDM's. Indeed, to get fast learning, what is required is large radius at the beginning of the execution, whereas to get a solution of good quality, it is a small radius that is needed in the long term. Decreasing similarity functions would also eliminate the need for tuning the radius parameter.

Note that the hash data structure that we described can also be used by decreasing radius SDM's. A periodic rehashing would be desirable however, to ensure that the cardinality of the set of potential active locations stays close to the actual number of active locations. Another data structure that would be worth trying with decreasing radius SDM is the kd-tree data structure. It is well known that kd-trees are efficient data structures for range search, but dynamically built kd-trees also work better when they are periodically rebuilt. It is not clear whether the hash or the kd-tree is better for decreasing radius and empirical tests are still needed (in particular, the answer might be environment-dependent).

The second variation is more radical and involves attaching a similarity function to each hard location instead of having a "global" similarity function. This is motivated by the fact that some regions of the state space might require a lower hard location density to get the same quality of approximation. We would cover those regions by hard locations with a bigger activation radius (by that, we mean that the similarity function attached to those hard locations would have a bigger activation radius). In other words, we would get an approximator with a variable resolution. We hope that this could be a key property in order to get dimensional scalability. There is, however, currently no theoretical or practical justifications for this hypothesis. An important planned future work is to test it empirically with problems of dimensionality 15-20.

The modifications to the data structure required to carry these tests will be more substantial. Indeed, the problem of finding active locations can no longer be reduced to a range search problem in this case. If the data structure partitions the space in some way, then hard locations should be attached to each partition intersecting their activated region (instead of just the partition in which their center lie). Then, only those hard locations that are attached to the partition containing the query point need to be examined. The problem with this approach is that the hard location might have to be attached to a large number of partitions if the activation radius of the inserted hard location is bigger than the size of the partitions. In a kd-tree that uses hard locations as splitting points, this problem could avoided by insuring that hard locations with bigger radius are closer to the root of the tree. This would make, however, the

construction of such tree difficult for two independent constraints (the tree have to be balanced and hard locations with bigger activation radius must be close to the root) would have to be considered to get an efficient tree. We propose a hash-based data structure that is simpler to implement and that we shall call a B-hash.

If K is the maximum number of hard locations in the data strucutre, let $1 \leq \iota \leq K$ be arbitrary but fixed. Suppose that the sequence of *add*, *delete* and *find* is not known in advance but that the distribution $\mu_{d.a.r.}$ of the activation radius of the hard locations that will be inserted is given. Pick $r_1 < r_2 < \cdots < r_\iota$ such that the r_i minimize:

$$\sum \min_{r_i} (r_i - r)^2 \mu_{d.a.r.}(r).$$

Let π_r and φ_r be respectively the maps π and φ described in section 3.3 with $\delta = r$. Let $\psi : F \to \{finite\_strings\}$ be an injective map, where F is the field of the finite dimensional vector space. With those definitions, the algorithms for insertion, deletion and look-up for a B-hash are:

INSERT($state, index, radius$)

```
1  r ← arg min_{r_i} (r_i − radius)^2
2  key ← (ψ(r)) ∘ (φ_r ∘ π_r(state))
3  list ← hashMap .get(key)
4  if list = NIL
5     then list ← new List
6          list .insert(index)
7          hashMap .insert(list)
8     else if index ∉ list
9             then list .insert(index)
```

DELETE($state, index, radius$)

```
1  r ← arg min_{r_i} (r_i − radius)^2
2  key ← (ψ(r)) ∘ (φ_r ∘ π_r(state))
3  list ← hashMap .get(key)
4  list .delete(index)
5  if list .isEmpty()
6     then hashMap .delete(key)
```

FIND-POTENTIALLY-ACTIVE-LOCATIONS($state$)

```
1  returnedList ← new List
2  for each r ∈ {r_1, ··· , r_ι}
3        do
4           key ← (ψ(r)) ∘ (φ_r ∘ π_r(state))
5           list ← hashMap .get(key)
6           returnedList ← returnedList ∪ list
7  return returnedList
```

This method can be seen as maintaining multiple hashes, each one being in charge of storing hard locations with a certain interval of activation radius. The only difference is that those hashes are all combined into a single hash, and the interval of activation radius (or more precisely its

representative in the set $\{r_1, \cdots, r_\iota\}$) is also transformed into a string and concatenated to the key.

The parameter ι should be chosen so that it is the biggest value such that the expected value of the number of hard locations in line 5 that are actually active is positive for each iteration of the for loop of line 2-6. Since the distribution of the centers cannot be assumed to be uniform in general, it might be necessary to use empirical tests to set this parameter and evaluate the performance of the algorithm.

4 Annex

References

[1] Standard template library programmer's guide. http://www.sgi.com/tech/stl/index.html.

[2] D. Precup B. Ratitch. Sparse distributed memories fo on-line, value-based reinforcement learning. *ECML*, 2004.

[3] W. Smart C. Szepesvari. Interpolation-based q-learning. *ECML*, 2004.

[4] J. Tsitsiklis D. Bertsekas. *Neuro-Dynamic Programming.* Athena Scientific.

[5] G. Gordon. Reinforcement learning with function approximation converges to a region. *NIPS*, 2000.

[6] B. Van Roy J. Tsitsiklis. An analysis of temporal-difference learning with function approximation.

[7] P. Kanerva. Sparse distributed memory and related models. *Associative Neural Memories*, pages 50–76, 1993.