Sparse Memory Structures Detection

Alexandre Bouchard-Côté
Student ID: 110228970
Final Report for COMP-652

December 21, 2004

Exploiting lower dimensional structures of the state space is often considered as a potential cure for the lack of dimensional scalability of reinforcement learning algorithms. Indeed, some approximation architectures, notably the Sparse Distributed Memories architecture (SDM[3]), attempt to locate regions of the state space that are "more interesting" and allocate proportionally more memory resources to model them accurately. Unfortunately, this attractive and intuitive idea is poorly formalized and the only empirical evidences of its efficiency come from indirect observations such as total reward improvement on reinforcement learning tasks. The aim of this paper is to describe a first attempt to directly observe automatic lower dimensional structure discovery. A larger program targeting a better understanding of these structures will also be described.

1 Notation and Terminology

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* ($\mathfrak{P}$) for sequences of maps from S. 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* [1]:

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

1.2 The SDM architecture

Assume that the state space is a closed multi-interval. 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*

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

  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.

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

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

- 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

On the basis of the second observation, D. Precup and B. Ratitch [1] implemented a dynamic memory allocation algorithm based on the following rules:

- If fewer than N locations are activated by the input $\vec{s}$, add a new location centered at $\vec{s}$, if its addition does not violate the following condition:

$$\sigma(\vec{s},\vec{t}) < \begin{cases} 1-\frac{1}{N-1} & \text{if } N \geq 3 \\ \frac{1}{2} & \text{if } N = 2 \end{cases} \qquad \forall \vec{t} \in H_{\vec{s}}. \tag{1}$$

  The current target value is stored in this location.

- If the number of activated locations is $N' < N$, then $(N - N')$ locations are randomly placed in the neighborhood of the current sample. The addresses of new locations are set by sampling uniformly randomly from the intervals $[s_i - \beta_i, s_i + \beta_i]$ in each dimension, where $\vec{\beta} = (\beta_1, \cdots, \beta_n)$ is such that $\mu(\vec{x}_1, \vec{x}_2) = 0$ whenever for some i the absolute value of the i-th coordinate of the difference of $\vec{x}_1$ and $\vec{x}_2$ is greater than or equal to β_i (this quantity is called the activation radius, and it is a small value for the type of SDM that we use since we assume σ to be symmetric triangular functions). The new locations are selected such that condition (1) is not violated. The value, currently predicted by the memory for the corresponding address, is stored in such a location.

Refer to [1] for more details and for the specification of the special rules used when the memory is full (in which case some memory locations are moved around). When SDM is paired with this dynamic memory allocation algorithm, it is observed that it outperforms (in terms of total rewards) CMAC (or SDM with a static memory layout) in the mountain car task, especially when difficult memory constraints are imposed. The current hypothesis is that it is caused by the distribution of the hard locations generated by the above algorithm that favors important regions of the state space. The next section describes an experiment that attempts a first step towards proving this hypothesis: determining how far is this distribution from the uniform distribution (in terms of $\mathscr{L}^2$-distance of their estimated probability density functions).

2 $\mathscr{L}^2$ distances

2.1 A Probability Density Function Estimator

Given a set of points $\Xi = \{\vec{\xi}_1, \vec{\xi}_2, \cdots, \vec{\xi}_m\}$ in the unit n-dimensional multi-interval $\mathfrak{I}$, we need to find a way to approximate the probability density function

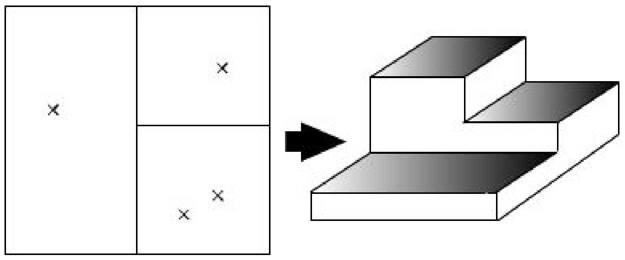
f that generated them. In particular, we would like to have this approximation in a form that is easy to integrate and to combine with other measurable functions.

The most straightforward approach is to partition each dimension into k sub-intervals of the form $\left[\frac{a}{k}, \frac{a+1}{k}\right]$ and to count the number of points p in each sub-multi-interval $\left[\frac{a_1}{k}, \frac{a_1+1}{k}\right] \times \cdots \times \left[\frac{a_n}{k}, \frac{a_n+1}{k}\right]$. The probability density function could then be approximated by a simple measurable function taking the value:

$$\frac{\text{total area}}{\text{partition area}} \times \frac{\text{number of points in the partition}}{\text{total number of points}} = \frac{k^n p}{m} \tag{2}$$

in the corresponding partition.

The problem with this method is that the resolution —controlled by the parameter k— is the same on the entire space. This is not practical in our case, where the density of the points can vary greatly over the space. Thus, variable resolutions are required (i.e. more resolution where more points are available). To overcome this problem, we use instead a tree-based scheme (which shares some similarities with kd-trees). To create an adaptive partitioning of the state space, we start with the whole space, and we iteratively select the partition containing the highest number of points and split it into two halves. A priority queue is used in order to do this efficiently. We iterate $\frac{m}{\alpha}$ times, where α is a parameter controlling the number of points we would like to have in the partitions. Again, we construct a simple measurable function from the obtained partition by setting the value at each partition as in the left hand side of (2). For instance:



In order to set the parameter α and to test the obtained method, we generated points using various distributions and compared the approximation obtained with the true result both quantitatively and qualitatively. The quantitative test was given by validating the $\mathscr{L}^2$ distance of the approximation $\tilde{f}$ from the uniform distribution:

$$\int_{\mathfrak{I}} (\tilde{f} - 1)^2 d\mu \tag{3}$$

The results were conclusive (see Figure 1 and 2 for qualitative results) and showed that a parameter α of about 10 is optimal in most of the cases.

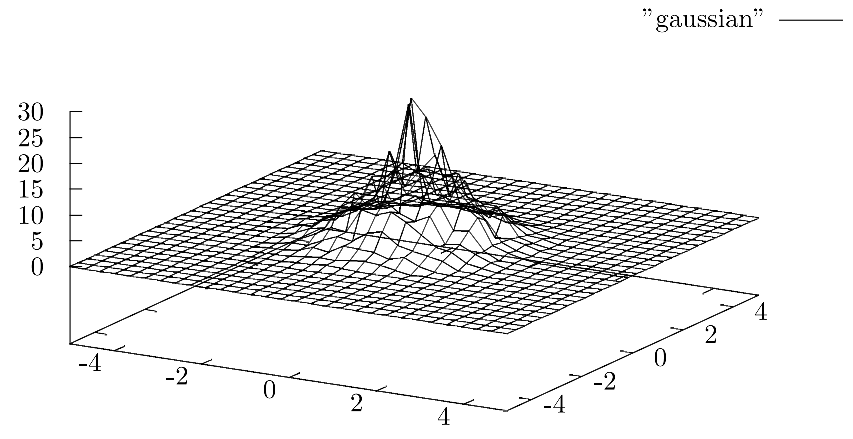


Figure 1: Approximation obtained from points drawn from a gaussian distribution ($\mu_1 = \mu_2 = \rho = 0, \sigma_1 = \sigma_2 = 1$)

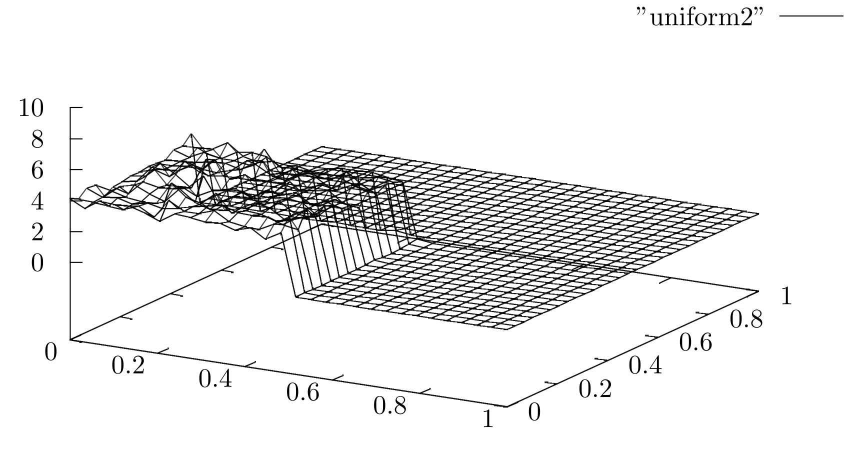


Figure 2: Approximation obtained from points drawn from the uniform distribution on $[0, 0.5] \times [0, 0.5]$

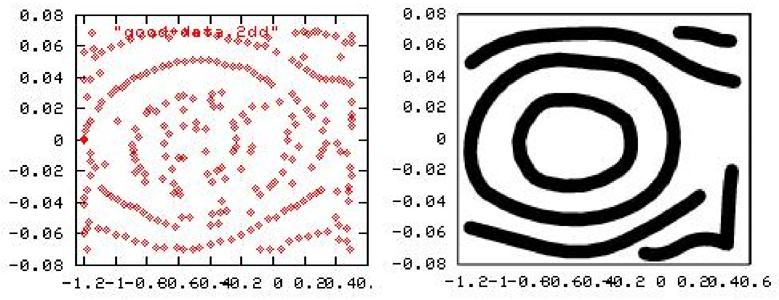


Figure 3: Left: position of the hard locations after the execution of SDM on the mountain-car domain. Right: the underlying 1-dimensional distribution is easily seen.

2.2 Results with the Mountain-Car Domain

Next, the same implementation of SDM as in [1] was used to solve the mountain car problem. The parameters used for the experiment were selected from an experiment in [1] that discriminates the total reward performances of SDM versus CMAC (i.e. a maximum of 100 hard locations, $N = 5$, single start state, activation radius $< 0.28, 0.023 >$ and exploration parameter $\epsilon = 0.01$). It was expected that this discrepancy was caused by a non-uniform distribution of the hard location and that this non-uniformity would be detected with the $\mathscr{L}^2$ metric. Surprisingly, the obtained distance from a uniform distribution is very close to zero (i.e. 0.03654). Thus, this experiment does not confirm the presence of non-uniform distribution. We shall now discuss the potential causes and implications of this result.

Since the state space of this problem is 2-dimensional (position, speed), the first thing to do was to look at the position of the hard locations in the space. Qualitative analysis of Figure 3 shows that the location are indeed distributed according to a lower-dimensional string structure (this is corroborated by the fact that hard location are inserted by an agent following 1-dimensional *trajectories*). However, these strings span a very large proportion of the state space, which explains the fact that the $\mathscr{L}^2$ distance to the uniform distribution is small. For larger domains (especially for those with a higher dimensionality) this kind of phenomenon cannot happen unless an unreasonably large number of hard location is used. Therefore, the $\mathscr{L}^2$ metric approach should not be discarded, and tests on new, higher-dimensional problems are on the way, as well as a mature implementation of SDM in java offering a easier framework to design and execute such tests (the version used to perform the test described in this section is in c++).

3 Other Metrics and Future Work

The $\mathscr{L}^2$ metric is not the only tool conceivable to investigate sparse memory structures, it is actually a rather crude one. The next generation of measurements could come from a family of metrics developed by the Labelled Markov Processes research community [2]. These distance functions measure how difficult it is to differentiate two LMP's (in terms of number of transitions and probabilities). The first challenge is to find an algorithm that computes the metrics in continuous state systems. We shall now describe a way in which this metric could be used to confirm the benefits of dynamic memory allocation algorithms. Let us look at SDM from a different perspective: instead of using the usual SDM+SARSA system to produce a value function, let us keep only the position of the hard location at the end of the execution of the algorithm. We construct 2 LMP as follow:

- the first one is an exact representation of the dynamical behavior of the environment. To start with, suppose that we only consider problems for which the objective is to reach a goal state as fast as possible (e.g. mountain-car). The first LMP has 2 labels. The first kernel of the LMP is just the transition associated with each action when the best action is greedily selected (so we restrict our study to problems for which a good approximation of the optimal value function is known; this is a perfectly reasonable assumption because we are studying the memory allocation algorithm, not the algorithm that is supposed to discover the optimal value function), and the second kernel is a self-loop with probability one corresponding to the only action enabled when a goal state is reached.
- The second LMP is identical, except that instead of basing the first kernel on the optimal value function, it is based on the value function obtained by projecting back and forth the optimal value function on the basis points.

Finally, the goal is to compare with LMP metrics these 2 systems. We would get a comparison method that is independent of the learning algorithm, numerical values of the reward and really measures the impact of the position of the hard locations. This would form a very good tool to evaluate dynamic memory allocation algorithms.

References

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

[2] R. Jagadeesan P. Panangaden J. Desharnais, V. Gupta. Metrics for labelled markov processes. *Theoretical Computer Science*, pages 323–354, 2003.

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