Sparse Distributed Memories in Reinforcement Learning: Case Studies

Bohdana Ratitch **Swaminathan Mahadevan** **Doina Precup**
{bohdana,smahad1,dprecup}@cs.mcgill.ca, McGill University, Canada

Abstract

In this paper, we advocate the use of Sparse Distributed Memories (SDMs) (Kanerva, 1993) for on-line, value-based reinforcement learning (RL). The SDMs model was originally designed for the case, where a very large input (address) space has to be mapped into a much smaller physical memory. SDMs provide a linear, local function approximation scheme, which is often preferred in RL. In our recent work (Ratitch & Precup 2004), we developed an algorithm for learning simultaneously the structure and the content of the memory online. In this paper, we investigate the empirical performance of the Sarsa algorithm using the SDM function approximator on three domains: the traditional Mountain-car task, a variant of a hunter-prey task and a motor-control problem called Swimmer (Coulom 2002). The second and third tasks are highly-dimensional and exhibit complex dynamics, yet our approach provides good solutions.

Introduction

Value-based RL methods typically rely on function approximators in order to represent value functions in large or continuous domains. Linear approximators are usually preferred to non-linear ones due to better theoretical guarantees and ease of use. At the same time, local approximators are often preferred to global ones, because they can incorporate new data faster, and are less vulnerable to correlated and non-stationary training data. Many practical RL applications have been built around linear and/or local approximators, e.g., CMACs (Sutton & Barto 1998), piecewise linear interpolations (Munos & Moore 2000) and memory-based methods (Atkeson, Moore, & Schaal 1997; Santamaria, Sutton, & Ram 1998). Radial Basis Function Networks (RBFNs) have been used much less (Gordon 1995; Sutton & Barto 1998), because of the difficulty of choosing their centers and widths. One important problem cited in application papers, e.g., (Munos & Moore 2000), is the poor scaling of such approximators with the dimensionality of the state space.

In our recent work (Ratitch & Precup 2004), we investigated the use of Sparse Distributed Memories (SDMs) (Kanerva 1993) as a function approximator for value-based RL algorithms. SDMs were originally designed for very large, highly dimensional input spaces. They provide a linear, local approximation scheme. In general, local architectures, SDMs included, can be subject to the curse of dimensionality, as some target functions may require, in the worst case, an exponential number of local units to be approximated accurately across the entire input space. However, it is widely believed that most decision-making systems need high accuracy only around low-dimensional manifolds of the state space, or important state "highways". We combined the SDM memory model with the ideas from instance-based learning, which provides an approximator that can dynamically adapt its structure and resolution in order to cover such state "highways". One of the advantages of instance-based methods (Atkeson, Moore, & Schaal 1997) is that they do not require choosing the size or the structure of the approximator in advance, but shape it based on the observed data. We introduced a new algorithm for training SDMs in the context of RL, which configures the memory architecture on-line during learning, similarly to instance-based methods, while also limiting the growth of the memory size. Unlike other function approximators from supervised learning, our approach for memory allocation is robust with respect to the non-stationary data distribution caused by the fact that control strategies change during learning. The resulting learning algorithms remain close (though outside) of the scope of current theoretical results on value-based RL with function approximation. In this paper, we present experiments using our approach in the Mountain-Car task, as well as case studies in two complex domains: a motor-control task called the Swimmer (Coulom 2002) and a variant of a hunter-prey domain. Based on our experiments, the proposed approach has great practical potential by providing good performance while being very efficient in terms of the resulting memory sizes as well as the computation time.

Sparse Distributed Memories

SDMs are a generalized random-access memory, based on an array of addressable storage locations of fixed capacity. For large virtual address spaces, a memory location cannot be allocated for every possible address. A reasonably large sample of addresses is chosen instead so that it is sufficient to approximate a target function. There are no restrictions in the generic architecture design as to how this sample has to be chosen. However, it is important that the sample addresses approximately correspond to the underlying distribution of inputs in the approximation problem. The original work of Kanerva (1993) developed SDMs for the case of binary addresses and memory contents. We focus instead on continuous addresses and memory contents, as our goal is to handle continuous RL tasks.

Retrieval/Prediction. When a value is to be retrieved from some address $\mathbf{x}$, a set $H_\mathbf{x}$ of *nearby* locations is activated, as determined by a *similarity measure* $\mu^k = \mu(\mathbf{h}^k, \mathbf{x})$ between the target address and the locations $\mathbf{h}^k, k = 1, \ldots M$, where M is the memory size. The similarity measure can be defined in many different ways, e.g., Hamming distance, if addresses are binary. We focus on the case of real-valued input vectors. For our experiments, we chose a similarity measure between input vector $\mathbf{x} = \langle x_1, ..., x_n \rangle$ and location $\mathbf{h} = \langle h_1, ..., h_n \rangle$ based on symmetric triangular functions:

$$\mu(\mathbf{h},\mathbf{x}) = \min_{i=1,\ldots,n} \mu_i(\mathbf{h},\mathbf{x})$$
$$\mu_i(\mathbf{h},\mathbf{x}) = \begin{cases} 1 - \frac{|x_i - h_i|}{\beta_i} & \text{if } |x_i - h_i| \le \beta_i \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Here, $\langle h_1, ..., h_n \rangle$ represent the *location address* and β_i are the *activation radii* in each dimension. The similarity measure directly translates into the location's degree of activation, which, in this case, is continuous in the $[0,1]$ interval.

Let w_k be a value stored at $\mathbf{h}^k$. Then the predicted value for the target address $\mathbf{x}$ is computed as:

$$\hat{f}(\mathbf{x}) = \frac{\sum_{k \in H_\mathbf{x}} \mu^k w_k}{\sum_{k \in H_\mathbf{x}} \mu^k}. \tag{2}$$

The normalized activations of the memory locations, $\frac{\mu^m}{\sum_{k \in H_\mathbf{x}} \mu^k}$, can be viewed as features of the input $\mathbf{x}$. Hence, the prediction is a linear combination of local features.

Storage/Learning. Upon receiving a training sample $\langle \mathbf{x}, f(\mathbf{x}) \rangle$, the values stored in all the active locations are updated using the standard gradient descent algorithm for linear function approximation:

$$w_m := w_m + \alpha \left[f(\mathbf{x}) - \hat{f}(\mathbf{x}) \right] \frac{\mu^m}{\sum_{k \in H_\mathbf{x}} \mu^k}, \forall m \in H_\mathbf{x} \tag{3}$$

where $\hat{f}(\mathbf{x})$ is the prediction and α is the learning rate.

SDMs can be combined with RL algorithms in a straightforward way. For instance, in order to combine SDMs with SARSA(0) (Sutton & Barto 1998), one approximator is used to represent the action-value function $Q(s,a)$ for each action. The values stored in the SDMs are updated after every transition $\langle s, a \rangle \xrightarrow{r} \langle s', a' \rangle$ as follows:

$$w_m(a) := w_m(a) + \alpha \left[r + \gamma Q(s', a') - Q(s,a) \right] \frac{\mu^m}{\sum_{k \in H_\mathbf{x}} \mu^k} \tag{4}$$

for all $m = 1, \ldots M_a$. The SDMs can also be easily used with the eligibility traces (Ratitch & Precup 2004).

Choosing the SDM structure

The distribution of memory locations across the input space is very important for the success of SDMs and related models such as RBFNs. We refer the reader to (Ratitch & Precup 2004) for a discussion of the methods used in supervised learning to choose automatically the addresses of the memory locations or the structure of RBFNs. From our past experience, it is usually not appropriate to import directly such approaches into RL, because the non-stationary and correlated nature of the training data makes these methods perform poorly. Instead, we developed an algorithm that allocates and adapts memory resources dynamically, based on the observed data, and is suitable for RL tasks. Below we present the most important ideas of our algorithm; more detailed information can be found in (Ratitch & Precup 2004).

Our *dynamic allocation* algorithm starts with an empty memory and gradually adds locations based on the observed data. This is reminiscent of instance-based learning, which memorizes all data. However, because the maximum size of the memory is limited, and because the samples obtained in RL are correlated in space and time, just memorizing samples until the memory is filled tends to create very densely populated areas, while leaving other parts of the state space uncovered. Hence, our goal is to add locations only if the memory is too sparse around the training samples.

In this paper, we assume that the activation radii of the memory locations are uniform and fixed by the user and our algorithm automatically chooses the addresses of the memory locations. The algorithm has only one parameter, denoted N, which is the minimum number of locations that we would like to see activated for a data sample. It is also important to ensure that these locations are "evenly distributed" across their local neighborhoods; hence, we do not allow locations to be too close. More specifically, for any pair of locations $\mathbf{h}^i, \mathbf{h}^j$, we enforce the condition:

$$\mu(\mathbf{h}^i, \mathbf{h}^j) \le \begin{cases} 1 - \frac{1}{N-1} & N \ge 3 \\ 0.5 & N = 2 \end{cases} \tag{5}$$

This condition means that the fewer locations are required in a neighborhood (the smaller N), the farther apart these locations should be. Our experiments showed that this condition makes a big impact on the performance of SDMs, as it prevents allocating resources in clumps. One or more new locations can be added upon observing any new sample $\langle (s,a), \underline{Q}(s,a) \rangle$, where $s = \langle s_1, ..., s_n \rangle$ represents the input to the SDM for the value function of action a, and $\underline{Q}(s,a)$ represents the target for the taken action a. For example, $\underline{Q}(s,a) = r + \gamma Q(s', a')$ in the case of SARSA algorithm.

In order to ensure that at least N locations are activated in the neighborhood of state s, we use the following heuristic (which we will refer to as the *N-based heuristic* later in this paper): 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. The new locations are screened to ensure that condition (5) is not violated. The content of each location is initialized with value currently predicted by the memory for the corresponding address. With this heuristic, memory resources are allocated relatively close to the actual data samples. The parameter N is reminiscent of the parameter K in K-nearest-neighbor methods, but we do not store all the data, as is typical in nearest-neighbor methods. Instead, we use this parameter to obtain good space coverage, while also controlling the memory size.

If the memory size limit is reached but we still encounter a data sample for which the number of active locations N' is smaller then the minimum desired number N, we allow existing locations to move around. To this end, $N - N'$ inactive locations are picked at random and removed; the corresponding number of new locations are added to the neighborhood of the current sample using the previous heuristic. This approach, which we call *adaptive reallocation*, allows the memory to react quickly to a lack of resources in the regions visited under the current behavior policy. At the same time, the randomized nature of the removals and the fact that there are sufficient locations in most of the previously visited regions does not dramatically affect the approximation in the areas where the removals occur.

Resource allocation proceeds in parallel with learning the memory content. On each time step, new locations are added, if necessary, then the content stored in the memory locations is updated as presented in Eq.(4). We also experimented with a version in which the memory structure can be updated on prediction as well as learning steps, which exhibited positive effects on performance.

Related approaches. The instance-based approach of Forbes (2002) is conceptually similar to ours. It also uses heuristics for selectively adding new instances to the memory and for removing some of them when the memory capacity limit is reached. The method was formulated in the classical instance-based framework, based on the definition of two functions: a distance metric in the input space, e.g., the Euclidean distance, and a weighting function, e.g., Gaussian, that transforms the distances into weights to be used in locally weighted regression. In (Forbes 2002), as well as earlier in (Santamaria, Sutton, & Ram 1998), new instances are added to the memory if they are farther away from the existing instances than a specified threshold. The threshold was defined in terms of the distance metric and was not related to the bandwidths of the weighting functions. If this correpondance is not explicitly addressed, the obtained memory can be too sparse for the weighting functions that are being used. While it is easy to prevent this in the case of a uniform and fixed bandwidth of the weighting functions, such a formulation does not generalize to the varying bandwidths. In (Forbes 2002), no discussion of the practical behavior of the method and its parameter settings is provided (though there is a claim of using adaptive bandwidths).

Our approach, on the other hand, is directly related to the similarity function. It ensures that the memory locations are spread appropriately with respect to the radii of the similarity function and allows a coherent extension to the case of variable radii. In our approach, the similarity threshold is implied implicitly from the parameter N (minimum desired number of activated locations). Although this may seem equivalent, our experience with a fixed threshold, corresponding to some N through condition (5), showed that many more than N training samples can satisfy the threshold condition and thus be added to the memory. Using the parameter N provides a more stringent way to control the size of the memory, as illustrated in our experiments discussed in the next section.

The heuristic in (Forbes 2002) for removing instances when the memory capacity limit is reached is also different from ours. It suggests to discard instances whose removal introduces the least error in the prediction of the values of their neighbors:

$$error_m = \frac{1}{|H_{\mathbf{h}_m}|} \sum_{k \in H_{\mathbf{h}_m}} |Q(\mathbf{h}_k, a) - Q_{-m}(\mathbf{h}_k, a)| \qquad (6)$$

where $Q_{-m}(\mathbf{h}_k, a)$ is the prediction for input $\mathbf{h}_k$ without the instance $\mathbf{h}_m$. In the next section, we illustrate empirically the fact that this *error-based heuristic* and the randomized heuristic behave differently in practice. The former is also more expensive computationally: it requires either to perform a complete memory sweep when the reallocation is necessary, or to perform $(|H_{\mathbf{h}_m}| - 1)$ additional predictions on every memory access in order to maintain (approximate) error estimates. The cost of the randomized heuristic, on the other hand, is that of generating a random number and applies only when a new location actually has to be added in an underrepresented region of the input space.

Experimental Results

In all experiments, we used SARSA(0) (Sutton & Barto 1998) with ε-greedy exploration, where the action-value functions were represented using SDM function approximators. The longer version of the paper (Ratitch & Precup 2004) contains extensive results on the standard Mountain Car domain (Sutton & Barto 1998), where we compare the performance of SDMs to that of CMACs as well as to the related memory allocation method from (Forbes 2002). Here, we first highlight some of those results to illustrate the relative performance of our approach and that of (Forbes 2002). Then, we present new results from two case studies: one on a motor-control domain, known as Swimmer (Coulom 2002) and the other on a version of a predator-prey domain.

Mountain Car domain. We compared the dynamic allocation method based on our N-based heuristic with a dynamic allocation method in the style of (Forbes 2002). In the latter approach, a new location is added when the similarity of the new sample to all existing locations is below some threshold μ^*, without checking what the number of active locations is. We will refer to it as the *threshold-based heuristic*. We set the similarity thresholds μ^* to the values that would be obtained from Eq.(5) for the values of N and the activation radii used in the corresponding experiments with our heuristic. The objective was to investigate the resulting memory sizes, layouts and the performance based on the two heuristics.

Graphs (a) and (b) of Fig.1 present the returns of the greedy policies learned by using dynamic allocation with each of the two heuristics. In these experiments, the memory size limit was set sufficiently high to ensure that it would not be reached and we could test the dynamic allocation method alone. The asymptotic performance of the SDMs with the threshold-based heuristic is similar to that of our heuristic, but the learning is slower. The resulting memories are between 2-4 times larger with the threshold-based heuristic, which slows down learning, because more training is required for larger architectures. As mentioned before, our heuristic, which relies solely on the number of active locations, enables a better control over the amount of allocated resources and, as the experiments show, results in faster learning.

Graph (a) of Fig.2 shows the performance of the adaptive reallocation method, which allows moving the existing locations when the memory size limit is reached. The experiments were performed for the case in which the agent always starts the episodes from a single state, in which the car is at rest, at the bottom of the hill. We used our heuristic for adding locations. We tested two removal approaches: the randomized one, introduced in this paper and the error-based, suggested in (Forbes 2002). The graph shows experiments with memory parameters $N = 5, \beta = \langle 0.17, 0.014 \rangle$. The memory size limits were chosen to be equal to 230 and 175 so that the static memories of the same sizes were not able to learn a good policy. The SDMs were initialized with all locations distributed uniformly randomly across the state space and then allowed to move according to the heuristics used. As can be seen from graph (a), both removal heuristics exhibit very similar performance. However, as shown on graph (b), the behavior of the two heuristics is quite different. With the randomized heuristic, most reallocations happen at the beginning of learning and then their number decreases almost to zero. With the error-based heuristic the number of reallocations is much higher. This happens because the addition heuristic is density-based and the removal heuristic is error-based, and their objectives are not "in agreement". Graph (c) depicts 3000 location moves at the end of one training run, where removed locations are plotted with black dots and added locations with white. A mixed black-and-white cloud in one region of the state space shows that most removals happen in a particular region where the value function is relatively flat, but the same region is then visited and found to be too sparsely represented by the addition heuristic, which causes locations to be added back. Apparently such a cycle repeats. As mentioned earlier, with the randomized heuristic, no specific area of the input space is affected by removals more than others, thus cyclic behavior is minimized. The randomized heuristic is computationally much cheaper while showing more stable behavior and providing good policies. The error-based heuristic can still be an interesting choice, provided that it is in tune with the addition heuristic.

Hunter-Prey Domain. In this task, H hunters team up against a prey. RL is used to learn how to control the prey, while the hunters behave according to fixed, heuristic strategies. The state is given by $2H$ continuous variables, representing the position of each hunter relative to a polar coordinate system centered on the prey, and one integer variable for the number of alive hunters. To capture the prey, C hunters have to approach it within a circle of radius of 5, and the angle between adjacent hunters has to be $\leq \frac{2*\pi}{C} + 0.6$ radians. However, if fewer than K hunters are within 5 units of the prey, the closest one is killed. The hunters start each episode at random positions inside a circle of radius 50 around the prey. A stochastic controller moves the hunters individually as follows: w.p. 0.3, a hunter moves clockwise or counterclockwise 0.2 radians; w.p. 0.7, the hunter moves in the radial direction, either towards the prey, if this is safe, or away from the prey by 5 units. The episode ends when the prey is captured or when fewer than C hunters are alive. The prey can move north, south, east or west, 5 units per time step. The prey receives a reward of 1 if it kills a hunter, -200 if captured, and -1 per time step otherwise.

Figure 3 presents the results of experiments for 2-, 3- and 5-hunter tasks (5, 7 and 11 state dimensions respectively) with $C = H$ and $K = 2$ in each case. Graph (a) shows the performance of the SDM and CMAC (Sutton & Barto 1998) architectures on the 2-hunter task. The SDMs were trained with the dynamic allocation method. The learning step and the exploration rate parameters were optimized for each model and each configuration. As shown in the graph, CMACs were not able to learn the task even with considerable memory sizes. Graph (b) shows the results on the 3-hunter task, illustrating the effect of changing the activation radii and value of N. The choice of the activation radii seems to have a stronger impact on the overall performance compared to the N parameter. Graph (c) shows the average learning curve for the 5-hunter task, using the SDM resolution that seemed best in the 2- and 3-hunter tasks. Despite the higher dimension of the task, the performance achieved is the same as in the 2 and 3-hunter case, while the memory size does not increase significantly.

Swimmer task. The Swimmer motor control task (Coulom 2002) involves a multi-link robot with the links connected by joints, at which control torques can be applied. The swimmer is moving in a two-dimensional pool, where movement is due to the viscous friction with the water. The goal is to swim along the positive **x** direction as fast as possible. In the experiments reported here, we used a 2-link swimmer. The state is defined by the angles θ_i of the segments with respect to the x axis, the Cartesian coordinates G_x and G_y of the center of mass, and their derivatives. This would yield a total of 8 state variables. However, the reward function and the optimal control do not depend on G_x and G_y, so in the experiments we use only 6 state variables, namely $< \dot{G}_x, \dot{G}_y, \theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2 >$. We use a discrete action space, where the torque choices are limited to two values, -5 and $+5$. The immediate reward of the swimmer is the velocity of its center of mass along the x-axis, $\dot{G}_x$. We used the implementation of the Swimmer available from Coulom's web page[1]. We modeled the Swimmer as an episodic task (with $\gamma = 1$), where each episode (or trial) corresponded to 5 sec. of the physical time and consisted of 2000 discrete-time decision stages.

As a baseline, we consider the performance of a heuristic controller, which makes the swimmer exhibit a behavior similar to the human leg-movement during swimming. To achieve such behavior, we apply a negative (-5) torque whenever the angle of the first segment is in a specified range ($[-\theta, \theta]$ in Fig.4(a)), and positive (+5) torque otherwise. We measure the performance of the swimmer in terms of distance swum during a simulation equivalent to 5 seconds of

[1] http://remi.coulom.free.fr/swimmer.tar.bz2

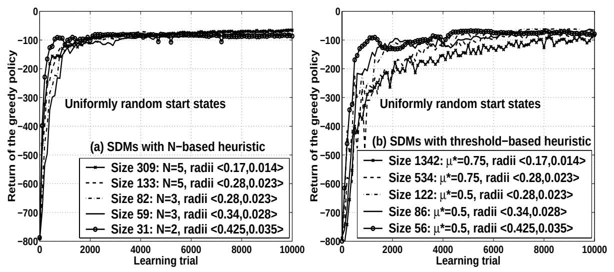

Figure 1: Mountain Car task and dynamic allocation method. Returns of the greedy policies are averaged over 30 runs. On graphs (a)-(c), returns are also averaged over 50 fixed starting test states. SDM sizes represent maximum over 30 runs. The exploration parameter ε and the learning step α were optimized for each architecture. Graphs (g) and (h) are for SDMs with radii $\langle 0.34, 0.028\rangle$, and $N = 5$ and $\mu^* = 0.5$ respectively.

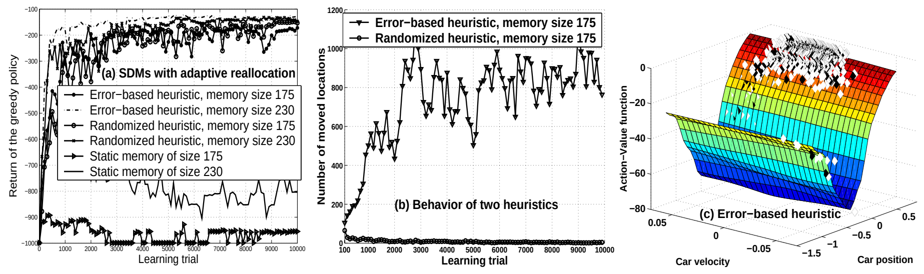

Figure 2: Mountain Car tasks and adaptive reallocation method. Each point on graph (b) represents the average over 100 trials and 30 runs. Graph (c) depicts the action-value function for "positive throttle".

the physical time (which is equivalent to the length of the episodes we performed for training and testing the RL controller).

Figure 4(a) shows the performance of the heuristic controller as a function of the length of the angle interval inside of which we apply the negative torque, as explained above. Intuitively, the shorter the interval, the higher the rate of "flapping" and hence the faster the movement. Interestingly, the relation between the interval length and the performance is not linear and differs across states. The best performance is observed in the case of very small intervals. Based on this observation, the RL controller would require a high resolution in the value function representation along the state dimensions representing the angles. At the same time, this task pushes the SDMs with the uniform radii of activation to its limitation: small radii (fine resolution) may result in big memory sizes, while large radii would not allow good performance. Ideally, we would like to have variable radii for the SDM locations, so that we have non-overlapping activation neighborhoods along the boundary where the optimal control changes, while having memory locations with wide activation radii in areas where the optimal control remains the same. In this paper, we investigate how good a performance we can obtain using uniform radii and adaptive addresses.

Each learning trial started from a state where the angles of the segments were picked uniformly randomly in the interval $[0, 2\pi]$. During each learning run, we performed testing of the current greedy policy after each 50 trials. The testing was performed for 5 random starting states (generated in the same manner as for learning trials and fixed ahead of time) as well as for the state corresponding to a vertical position. For each starting state, we simulated 5 testing trials with the current greedy policy and averaged the observed returns.

In our experiments, we used relatively large activation radii for the SDM, namely $\beta = \langle 0.8, 5, 1, 1.2, 1, 1.2\rangle$. We compared the performance of the learned strategies with the performance of the heuristic swimmer of a "similar resolution". That is, the (half) size of the angle interval, inside of which the heuristic controller applies the (-5) torque, is equal to the the SDM's radii in the state dimensions, corresponding to the segment angles (radii of 1 in this case). The number of memory locations that we wish to be activated on every memory access was set to $N = 10$.

The RL swimmer is able to learn this complex task: as can be seen from Fig.4 (b) and (c), it improves its performance with training and outperforms the heuristic swimmer, both for the vertical start state and for the random start states. The controller obtained in this setting uses a relatively small memory, as illustrated on Fig.4 (d). The experiments reported here are still preliminary; in particular, we have not yet used SDMs of fine resolution, with which we would expect to get better performance. This is left for future work.

Conclusions and Future Work

Our experiments demonstrated the ability of the value-based RL agents using SDM function approximators to learn complex, high-dimensional continuous control tasks. The proposed algorithm for allocation of the memory resources is robust in the context of RL and produces relatively compact representations of the action-value functions. However, the trade-off between memory resolution and size is not optimally resolved by adaptively choosing only the addresses of the memory locations. The activation neighborhoods should

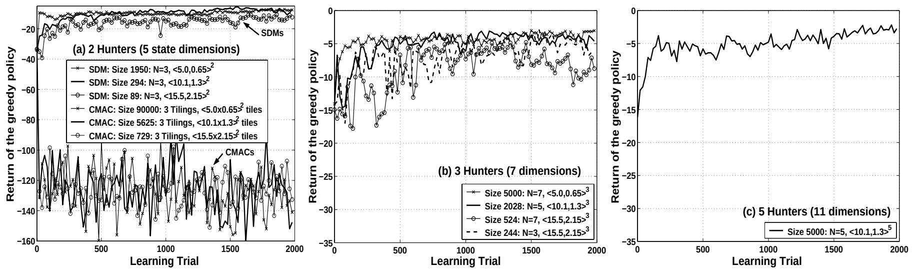

Figure 3: Hunter-prey domain. Returns are averaged over 20 runs and 100 fixed starting test states, sampled uniformly randomly. The exploration parameter $\varepsilon = 0.05$ and the learning step α was optimized for each architecture and task.

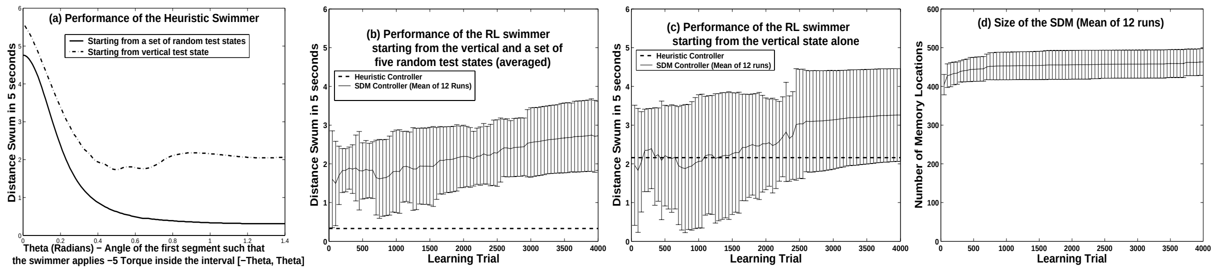

Figure 4: Performance on the Swimmer domain. Graph (a) shows the performance of the heuristic swimmer as a function of the interval length used in the swimmer heuristic design (see text). Graphs (b) and (c) show the performance of the RL vs. heuristic swimmer from five randomly chosen start states and the vertical starting state alone respectively. The SDM size growth during learning is shown on graph (d).

also vary in size depending on the properties of the target function. While we are working on extending our algorithm to this case, one simple solution, based on the current algorithm is also possible.

In order to obtain better performance while minimizing the increase in the computational time, we are currently experimenting with the idea of a *progressive refinement*, where we use a coarser memory resolution at the beginning of learning (corresponding to large activation radii) and then switch to a finer one. Intuitively, a small memory first learns a coarse representation of the action-value function and then refines it where necessary. The fine-grained SDM inherits from the coarse SDM locations that already have reasonable values; hence, learning can proceed faster. Also, as certain parts of the state space are visited less with a reasonable coarse policy, fewer new locations are added to the finer SDM. In contrast to other approximators, e.g., neural networks, SDMs easily allow changing structural parameters, without retraining the model from scratch.

Acknowledgments

This research was supported in part by research grants to Doina Precup from NSERC, CFI and FQRNT.

References

Atkeson, C.; Moore, A.; and Schaal, S. 1997. Locally weighted learning for control. *Artificial Intelligence Review* (11):75–113.

Coulom, R. 2002. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. Ph.D. Dissertation, Institut National Polytechnique de Grenoble.

Forbes, J. 2002. *Reinforcement Learning for Autonomous Vehicles*. Ph.D. Dissertation, Computer Science Department, University of California at Berkeley.

Gordon, G. J. 1995. Stable function approximation in dynamic programming. In *ICML'95*. Morgan Kaufman.

Kanerva, P. 1993. Sparse distributed memory and related models. In Hassoun, M., ed., *Associative Neural Memories: Theory and Implementation*. N.Y.: Oxford University Press. 50–76.

Munos, R., and Moore, A. 2000. Variable resolution discretization in optimal control. *Machine learning* (49):291–323.

Ratitch, B., and Precup, D. 2004. Sparse distributed memories for value-based reinforcement learning. Will be available from www.cs.mcgill.ca/∼sonce/sdm-paper.pdf after March 24, 2004.

Santamaria, J. C.; Sutton, R. S.; and Ram, A. 1998. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior* 6:163–218.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning. An Introduction.* Cambridge, MA: The MIT Press.