Sparse Distributed Memories for On-Line Value-Based Reinforcement Learning

Bohdana Ratitch[1] and Doina Precup[1]

McGill University, Montreal, Canada
{bohdana,dprecup}@cs.mcgill.ca

Abstract. In this paper, we advocate the use of Sparse Distributed Memories (SDMs) for on-line, value-based reinforcement learning (RL). SDMs provide a linear, local function approximation scheme, designed to work when a very large/high-dimensional input (address) space has to be mapped into a much smaller physical memory. We present an implementation of the SDM architecture for on-line, value-based RL in continuous state spaces. An important contribution of this paper is an algorithm for dynamic on-line allocation and adjustment of memory resources for SDMs, which eliminates the need for choosing the memory size and structure a priori. In our experiments, this algorithm provides very good performance while efficiently managing the memory resources.

1 Introduction

The use of function approximators with on-line, value-based reinforcement learning (RL) algorithms is the subject of much recent research and presents important theoretical and practical challenges. For various reasons, detailed in Sect.2, linear, local approximators are often the preferred choice. Many practical RL applications have been built around such approximators, e.g., CMACs [27], variable-resolution discretizations [18, 23, 32] and memory-based methods [3, 8, 24, 26]. Radial basis function networks (RBFNs) with fixed centers and widths have been used much less [10, 27], the main difficulty being in the choice of parameters for the basis functions. Most of these methods, however, still face important difficulties when applied to on-line learning in large domains. For example, CMACs and variable-resolution discretization approaches do not scale well to high dimensions; related methods proposed in [18, 32] are intended for off-line learning; the memory-based methods in [3, 26] do not address the issue of limiting the memory size, which can grow very big during on-line reinforcement learning. Global and/or nonlinear approximators, e.g., Neural Networks (NNs) and Support Vector Machines (SVMs) scale better, in principle. However, with on-line reinforcement learning, they have no convergence guarantees and are subject to some other practical problems. For example, NNs suffer from catastrophic forgetting and are notoriously hard to tune when combined with RL algorithms; SVMs (even with recent incremental training methods) rely on batches of previously seen data [5, 6, 16, 17, 20] which can be problematic with on-line RL due to the non-stationary data distribution.

In this paper, we revive older ideas [27, 24, 8] of using Sparse Distributed Memories (SDMs) [13] and instance-based training [2] for value-function approximation in on-line RL. SDMs provide a linear, local architecture, designed for the case where a

J.-F. Boulicaut et al. (Eds.): ECML 2004, LNAI 3201, pp. 347–358, 2004.

very large input space has to be mapped into a much smaller physical memory. One of the advantages of instance-based methods 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. In general, local architectures, SDMs included, can be subject to the curse of dimensionality, as an exponential number of local units may be required in order to approximate some target functions accurately across the entire input space. However, many researchers believe that most decision-making systems need high accuracy only around low-dimensional manifolds of the state space or important state "highways". The SDM model enables us to take advantage of this fact.

In this paper, we explore the flexibility of the SDM model and some principles of the instance-based learning to provide a function approximator that automatically allocates resources only as needed based on the observed data. We propose a new approach for such allocation and adaptation of SDMs, which unlike many methods from supervised learning, is capable of adapting limited memory resources to the changing data distribution as the control strategies evolve during reinforcement learning. Based on our experimental results, the proposed approach has great practical potential by providing high levels of performance while being very efficient both in terms of the resulting memory sizes and computational time. It also remains close to the scope of existing theoretical convergence guarantees.

The paper is organized as follows. In Sect.2, we introduce the notation for value-based RL algorithms. We summarize the standard framework of SDMs and then present our implementation of the SDM idea for the case of RL tasks in continuous state spaces in Sect.3. In Sect.4, we describe our approach for dynamic memory allocation and adjustment in SDMs, designed to work with on-line RL algorithms. Experimental results are presented in Sect.5. We end with conclusions and future work in Sect.6.

2 Reinforcement Learning

In the standard RL framework, a learning agent interacts with a stochastic environment at discrete time steps. On each time step t, the environment assumes some state s from the *state space* S and the agent picks an action a from the *action space* A. As a result, the environment transitions to a new state s' and the agent receives a numerical (stochastic) *reward* r. In a Markovian environment, the state transition distribution and the rewards depend only on $\langle s,a\rangle$. The goal of the agent is to find a *policy* $\pi : S \times A \rightarrow [0,1]$ (a way of choosing actions) that optimizes a *long-term performance* criterion, called *return*. Returns are usually defined as a cumulative function of rewards received over time. Many RL algorithms compute *value functions*, which are *expected returns*. For instance, the *optimal action-value function* with a discount factor $\gamma \in (0,1]$ is $Q^*(s,a) = \max_\pi E_\pi \{r_{t+1} + \gamma r_{t+2} + \ldots | s_t = s, a_t = a\}$.

In this paper, we focus on RL algorithms that iteratively compute estimates of the optimal action-value function from samples obtained by interacting with the environment. For example, at each time step, the SARSA algorithm [27] updates the value of the current state-action pair $\langle s,a\rangle$ based on the observed reward r and the next state-action pair $\langle s',a'\rangle$, using learning rate $\alpha \in (0,1)$, as follows:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma Q(s',a')] \tag{1}$$

In domains with large or continuous state spaces, value functions can be represented by function approximators. In this case, RL methods sample state-action pairs $\langle s,a\rangle$, which represent inputs, and estimates of the action-value function, e.g., $[r+\gamma Q(s',a')]$, representing targets. However, the problem of function approximation is more difficult in the context of RL. Targets do not come from the true optimal value function, they are "guesses" based on the current approximation (see Eq.1). Also, in on-line RL, the agent's action choices typically depend on the current value estimates in a (semi) greedy manner. Hence, both the input distribution and the target function are non-stationary. Moreover, during on-line learning, the training samples are not independent.

Some researchers (see, e.g., [3, 19]) argue that local methods are more suitable for RL than global ones. Local approximators allow only a few local parameters to be updated on every step, based on a distance to the current input. This is in contrast with global models (e.g., sigmoid NNs), in which all parameters are updated on every step. Local approximators do not suffer from catastrophic forgetting, which can be caused by non-independent and non-stationary sampling in RL. Also, they quickly incorporate new data in a local region of the input space, thus adjusting faster to the non-stationarity.

Theoretically, convergence properties are best understood for linear approximators, which compute the state value as a linear combination of some features of the state. Relevant results include the convergence of policy evaluation [31], the convergence of approximate dynamic programming with averagers [10], and non-divergence of SARSA(λ) [11]. The behavior of non-linear approximators is still poorly understood in theory, while practical evidence is not consistent.

3 Sparse Distributed Memory

The Sparse Distributed Memory architecture [13] was originally proposed for learning input-output associations between data drawn from a binary space. The input can be viewed as an "address" and the output is the desired content to be stored at that address. The physical memory available is typically much smaller than the space of all possible inputs, so the physical memory locations have to be distributed sparsely.

In SDMs, a sample of addresses is chosen (in any suitable manner) and physical memory *locations* are associated only with these addresses. When some address $\mathbf{x}$ has to be accessed, a set of the *nearby* locations is activated, as determined by a *similarity measure* (e.g. Hamming distance, if addresses are binary). The original SDM design assumes that the data to be memorized consists of bit vectors (with 0s substituted by -1s). When such a vector $f(\mathbf{x})$ needs to be stored, it is distributed between all the locations activated by $\mathbf{x}$, using bitwise addition. When the value for input (address) $\mathbf{x}$ is retrieved, the content of all active locations is combined by summation and thresholding.

In this paper, we focus on the case in which the inputs are vectors of real values and the outputs are also reals: $f(\mathbf{x}) : R^n \rightarrow R$. In this case, other popular approximators, e.g., CMACs and RBFNs, can be related to SDMs. In RBFNs, each RBF unit can be viewed as a memory location, where the center of the unit is the address and the similarity measure is determined by the widths of the basis functions. The relationship between CMACs and SDMs is discussed in Sect.5.

For presentation purposes and for our experiments, we chose a similarity measure based on symmetric triangular functions. The similarity between input vector $\mathbf{x} = \langle x_1,...,x_n\rangle$ and location $\mathbf{h} = \langle h_1,...,h_n\rangle$ is given by:

$$\mu(\mathbf{h},\mathbf{x}) = \min_{i=1,\ldots,n} \mu_i(\mathbf{h},\mathbf{x})$$
$$\mu_i(\mathbf{h},\mathbf{x}) = \begin{cases} 1 - \frac{|x_i - h_i|}{\beta_i} & \text{if } |x_i - h_i| \leq \beta_i \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Here, $\langle h_1, \ldots, h_n \rangle$ represents 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 $[0,1]$. This factorized similarity function allows an immediate symbolic interpretation of the location's semantics with respect to each input dimension. Of course, the similarity measure can be defined in many different ways (see, e.g., [2, 25]). It is possible to implement SDMs efficiently so that isolating active locations does not require computing the similarity of a data point to all locations, for example, by using *kd*-trees as in instance-based learning [26], or by inverted indexing.

To predict the value of input $\mathbf{x}$, we first find the set of active locations, $H_{\mathbf{x}}$. Let $\mu^k = \mu(\mathbf{h}^k, \mathbf{x})$ be the similarity between input $\mathbf{x}$ and the k^{th} location, $\mathbf{h}^k$, as in (2). Let w_k be a value stored at $\mathbf{h}^k$. Then the predicted value of $\mathbf{x}$ is:

$$\hat{f}(\mathbf{x}) = \frac{\sum_{k \in H_{\mathbf{x}}} \mu^k w_k}{\sum_{k \in H_{\mathbf{x}}} \mu^k} \quad (3)$$

This representation is equivalent to Normalized RBFNs. Using normalization provides a better support (i.e., reduces non-smoothness in the approximate function) in the regions of the input space where the basis functions overlap only little [15]. 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 computed as a linear combination of local features.

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

$$w_m := w_m + \alpha \frac{\mu^m}{\sum_{k \in H_{\mathbf{x}}} \mu^k} \left[f(\mathbf{x}) - \hat{f}(\mathbf{x}) \right], \forall m \in H_{\mathbf{x}} \quad (4)$$

where $\hat{f}(\mathbf{x})$ is the prediction for input $\mathbf{x}$ and $\alpha \in (0,1)$ is the learning rate. In Sect.4, we discuss how the addresses of the memory locations can be selected and updated.

SDMs can be incorporated into RL algorithms in a straightforward way. For instance, in order to combine SDMs with SARSA(λ) [27], 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 $\langle s,a \rangle \xrightarrow{r} \langle s',a' \rangle$:

$$w_m(\bar{a}) := w_m(\bar{a}) + \alpha e_m(\bar{a})[r + \gamma Q(s',a') - Q(s,a)], \forall \bar{a} \in A \text{ and } m = 1, \ldots, M_{\bar{a}}$$

Here, $e_m(\bar{a})$ are the eligibility traces associated with each location. For the replacing traces method, they decay by $\gamma\lambda$, $\lambda \in [0,1]$, for all $\bar{a} \neq a$, and are reset to $\frac{\mu^m}{\sum_{k \in H_s(a)} \mu^k}$ for the performed action a. If the memory is big, a list of locations with traces greater than some threshold can be maintained in order to perform this update efficiently.

4 Dynamic Resource Allocation

The distribution of the memory locations is crucial for the performance of SDMs and related models, such as RBFNs. It is usually assumed that the memory size is fixed at the beginning of learning and locations are either distributed uniformly randomly across the input space, or determined by unsupervised learning methods, such as clustering. In the second case, a batch of training data is assumed to exist from the beginning. Then the parameters of the local units can be additionally adjusted during learning. Both in the SDM and RBFN literature, there are several methods for doing this automatically.

For SDMs, the methods in [9, 12] periodically delete some units based on their activation frequencies to free up resources for allocation elsewhere: rarely activated locations are removed. According to our past experiments, this approach does not work well with RL, as it often results in the removal of units associated with very important states, such as goal states and catastrophic states, which usually have relatively low activation frequencies for some time after they have been initially discovered.

Memory layout can also be adjusted using on-line, unsupervised learning methods, as in [21, 28]. The approach introduced in [28] for binary SDMs slowly moves the existing memory locations toward observed data. If the number of active locations for a given training sample is too small, an inactive location is selected at random and moved toward the current input in one, randomly selected, dimension. A symmetric adjustment is made if too many locations are active. We implemented a version of this algorithm, but it did not allow stable learning, despite significant tuning because it was unable to track quickly the non-stationary data distribution produced by changing policies.

For RBFNs, one standard approach is to use gradient descent on the mean squared error to adjust the centers and widths of RBFs [7, 15]. Interestingly, it was observed in [15] that with RL, basis functions tend to move to regions in the state space with small temporal-difference errors while leaving large portions of the state space uncovered and failing to provide a good policy. As suggested in [15], a better method should also incorporate information about the density of the visited states.

Resource-allocating RBF networks [4, 19, 1] are initially empty, and new units are added based on the distance between the new data and the existing units, as well as based on the prediction error on the new data (with different variations on how the error is measured). These methods require lots of parameters to be chosen by the user, and it is not clear, in general, whether reliance on the prediction error is robust in on-line RL, because the error can vary a lot with changes in policy.

We propose a new method for determining automatically the SDM size and location addresses based on the observed data. In this paper, we assume that the activation radii of the memory locations are uniform and fixed by the user. The approach in [8], formulated in the instance-based learning framework, is conceptually similar to ours. However, technical differences between the two algorithms have some important implications, which we will discuss in more detail after the presentation of our method.

Our *dynamic allocation* algorithm starts with an empty memory, and locations are added based on the observed data. Since the samples obtained during on-line RL are correlated, memorizing all samples until the memory is filled can create unnecessary densely populated areas, while leaving other parts of the state space uncovered. Our goal is to add locations only if the memory is too sparse around the training samples.

Our 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 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 a condition on their similarity:

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

This condition means that the fewer locations are required in a neighborhood (the smaller N), the farther apart these locations should be.

A new location can be added upon observing any new sample $\langle (s,a), \bar{Q}(s,a) \rangle$, where $s = \langle s_1, ..., s_n \rangle$ represents the input to the SDM for the action-value function of action a, and $\bar{Q}(s,a)$ represents the target for the current state-action pair (s,a). For example, $\bar{Q}(s,a) = r + \gamma Q(s',a')$ in the case of SARSA algorithm. The following *N-based heuristic* is aimed at ensuring a minimum of N active locations in the vicinity of s:

`Rule 1:` If fewer than N locations are activated by the input s, add a new location centered at s, if its addition does not violate condition (5). The current target value, $\bar{Q}(s,a)$, is stored in this location.

If during learning there is not enough exploration to ensure a good spread of the visited states, the allocation using only the above heuristic proceeds very slowly, and learning can be stalled for a long time (a phenomenon we observed in preliminary experiments). To counteract this problem, we use an extension of the above heuristic, which sets up memory resources faster, while still allocating them close to the actual data samples:

`Rule 2:` If after applying `Rule 1`, the number of active locations is $N' < N$, then $(N - N')$ locations are randomly placed in the neighborhood of the current sample. The addresses of new locations are sampled uniformly randomly from the intervals $[s_i - \beta_i, s_i + \beta_i]$ in each dimension, while enforcing condition (5). The value currently predicted by the memory for the corresponding address is stored in such a location.

The parameter N in the above heuristics is reminiscent of the parameter k in the k-nearest-neighbor methods, which determines the number of instances that are used for locally weighted learning. Unlike the classical instance-based approach, our method provides a way to selectively store training samples to obtain a good space coverage with respect to this parameter while 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 than the minimum desired number N, we also allow existing locations to move around. Unlike the approach described in [28], we do not adjust the existing addresses slowly. Instead, we pick at random and remove one inactive location (or $(N - N')$, if `Rule 2` is used). The corresponding number of new locations are added to the neighborhood of the current sample using `Rule 1` or 2. When a location $\mathbf{h}$ is to be removed, we first find, among locations in the active set $H_{\mathbf{h}}(a)$, the location $\mathbf{h}'$ that is closest to $\mathbf{h}$. Then, $\mathbf{h}$ and $\mathbf{h}'$ are both replaced by another location, $\mathbf{h}''$, placed midway between them. The value of $\mathbf{h}''$ is set to the average of the values of $\mathbf{h}$ and $\mathbf{h}'$. This approach, which we call *randomized reallocation*, allows the memory to react quickly to the 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 ensure that it does not affect dramatically any particular area of the input space. The method is cheap both in terms of computation and space, since the choice of locations to be removed is not based on any extra information, like in other algorithms [9, 12, 14, 8].

Resource allocation proceeds in parallel with learning the memory content. On each learning step, new locations are added or moved, if necessary, then the values stored in the memory are updated as presented in Sect. 3. The resource adjustments can be performed on prediction steps as well. If a new location is added in this case, the value currently predicted for the corresponding address is stored in it. In our experiments, this proved to be very beneficial, as it allowed the memory layout to adapt faster. It allows the SDMs for all actions to get adjusted to the current state distribution, as opposed to adjusting only the SDM for the performed action.

Our approach is conceptually similar to the instance-based approach presented in [8], which 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 [2], based on the definition of two functions: a distance metric in the input space, e.g., the Euclidean distance, and weighting functions, e.g., Gaussian, that transform the distances into weights to be used in locally weighted regression. In [8], as well as earlier in [24], new instances are added to the memory if they are farther away from the existing instances than a specified threshold. Such a threshold is defined in terms of the distance metric and is not related to the bandwidths of the weighting functions. If this correspondence is not explicitly addressed, the obtained memory can be too sparse. While it is easy to prevent this in the case of a uniform, fixed bandwidth of the weighting functions, the approach does not generalize to varying bandwidths. Although adaptive bandwidths were claimed to be used in [8], no discussion was provided for the practical behavior of the method and its parameter settings.

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 by the parameter N (minimum desired number of activated locations). This may seem to be equivalent; however, more than N training samples can satisfy the similarity threshold and thus be added to the memory. Thus, using the parameter N provides a more conservative way to control the size of the memory, as was confirmed by our experiments (see Sect.5).

The heuristic in [8] for removing instances in the case when the memory capacity limit is reached is also different from ours. It suggests discarding the instances whose removal introduces the least error in the prediction of the values of their neighbors:

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

where $Q_{-m}(\mathbf{h}_k, a)$ is the prediction for input $\mathbf{h}_k$ without the instance $\mathbf{h}_m$. In Sect.5, we experimentally show 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 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.

5 Experimental Results

We tested SDMs incorporated into the SARSA(0) algorithm with ε-greedy exploration. In this paper, we provide detailed experimental results on the standard Montain-Car benchmark [27], commonly used in the RL community. Due to the space limitation, we cannot include here the results on other domains, but we refer the reader to [22] for case studies on two other tasks: a variant of the hunter-prey domain with up to 11 state variables and an instance of a swimmer motor-control task with 6 state variables.

Mountain-Car is an episodic task, with a two-dimensional continuous state space, where the agent has to learn to drive up a hill from a valley. Episodes were terminated when the goal was reached, or after 1000 steps. To obtain a baseline for performance, we used the popular CMAC (tile coding) approximator [27], which is particularly successful on this domain. CMACs are related to SDMs, but the memory layout is fixed a priori, with the locations (tiles) arranged in several superimposed grids (tilings). Each input activates one tile in each tiling, and the activation mechanism is binary. Since CMACs rely on the discretization of the input space, their size scales exponentially with the input dimensionality. We used CMACs and SDMs of "similar resolutions": If a CMAC had T tilings, we set the parameter for dynamic memory allocation with the N-based heuristic as $N = T$. The activation radii of the SDMs were set equal to the size of the CMAC tiles. We also tested a dynamic allocation method in the style of [8], where a new location was added when the similarity of the new sample to all existing locations was below some threshold μ^*, whithout checking whether the number of active locations already exceeds N. We will refer to it as the *threshold-based heuristic*. In this case, 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 the N-based heuristic. The objective was to investigate the resulting memory sizes, layouts and the performance based on the two heuristics.

We conducted two sets of experiments as follows. In the first set, the start state of each episode was chosen uniformly randomly. This is the most popular setting and it is "easier" because the starting distribution ensures good exploration. Graphs (a), (b) and (c) of Fig.1 present the returns of the greedy policies learned by CMACs and dynamically allocated SDMs with the N-based and the threshold-based heuristics respectively. 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 performance of the SDMs is either the same or (in most cases) much better than that of CMACs. It degrades more gracefully with the decrease in resolution. Moreover, SDMs with the N-based heuristic always consume fewer resources, as shown in the legends of the graphs. The asymptotic performance of the SDMs with the threshold-based heuristic is similar to that of the N-based heuristic, but 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, the N-based heuristic allows better control over the amount of allocated resources and, as the experiments show, results in faster learning.

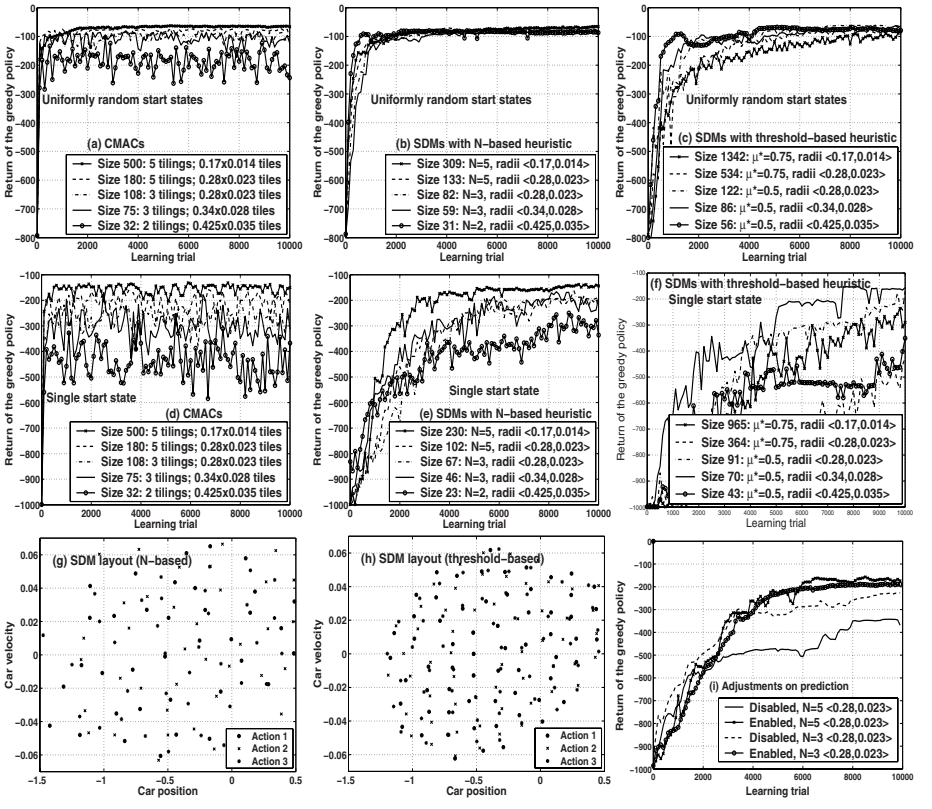

Fig. 1. Dynamic allocation method. Returns of the greedy policies are averaged over 30 runs. On graphs (a)-(c), returns are also averaged over 50 fixed starting test states. SDM sizes represent maximum over 30 runs. The exploration parameter ε and the learning step α were optimized for each architecture.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

In the second set of experiments, we used a single start state where the car starts at the bottom of the hill with zero velocity. In this setting, exploration is much more difficult. We specifically wanted to test the performance of SDMs when the training samples are highly correlated and distributed non-uniformly. The results are shown in the middle row of Fig.1. SDMs with the *N*-based heuristic (using `Rule 1` and `2`) generally learn better policies than CMACs and take advantage of the fact that not all states are visited. The resulting memory sizes for SDMs (graph (e)) are roughly 30% smaller than in the previous experiment (graph (b)). SDMs with the threshold-based heuristic, however, were much slower and exhibited much higher variance (not shown here) with this single start-state training, even though they had a large number of locations placed exactly along the followed trajectories. This demonstrates that, with restricted exploration, `Rule 2` of our approach, which allows adding locations close to but not exactly on trajectories, helps to build quickly a compact model with good

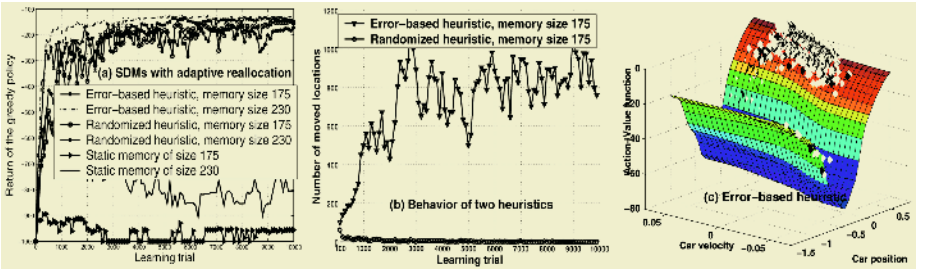

Fig. 2. Adaptive reallocation method. Each point on graph (b) represents the average over 100 trials and 30 runs. Graph (c) depicts an action-value function for action "positive throttle".

generalization capabilities. Also, under limited exploration, smaller architectures (as obtained with the N-based heuristic) should be expected to learn better as they suffer less from over-fitting. Graphs (g) and (h) show examples of SDM layouts obtained with the N-based and the threshold-based heuristics for these experiments. The SDMs obtained with the N-based heuristic are less dense and span the state space better.

Finally, graph (i) of Fig.1 shows the performance improvement achieved by allowing adjustments to the memory layout during predictions as well as during RL updates. The graph shows results for the dynamic allocation method with the N-based heuristic only, but performance improvements were observed with the threshold-based heuristic and the reallocation algorithm as well. Note that all the experiments with the SDMs (for both heuristics) discussed above were performed with this option enabled. With both heuristics, most memory locations ($\sim 85\%$) where added in the first 200 trials.

Graph (a) of Fig.2 shows the performance of the randomized reallocation method, which allows moving the existing locations when the memory size limit is reached. The experiments were performed for the single start-state problem using the N-based heuristic for location additions. We tested two removal approaches: the randomized one, introduced in this paper and the error-based, suggested in [8]. This 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 which is 100% and 75% of the size obtained for the same memory resolution with the dynamic allocation method and the N-based heuristic in the previous experiments. The SDMs were initialized with all locations distributed uniformly randomly across the state space and then allowed to move according to the heuristics used. Note that the static memories of the same sizes were not able to learn a good policy. 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 populated, so locations are added back. Apparently such a cycle repeats itself. As mentioned earlier, with the randomized heuristic, no specific area of the input space is affected by removals more than others, so 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.

6 Conclusion and Future Work

In this paper, we combined on-line value-based RL with a function approximation model based on SDMs. This model is local and linear, which is often preferred in RL and has enough flexibility to scale well with large and highly dimensional input spaces. Our main contribution is a new approach for dynamic allocation and adaptation of the memory resources specifically suited for on-line value-based RL. Our approach to adding new memory locations provides a disciplined way to control the memory size and density taking into account the location activation mechanism and can be readily used in the future with activation functions that have variable bandwidth across locations. Moreover, our method facilitates learning under constrained exploration scenarios. We demonstrated the importance of agreement between the methods for adding and removing the memory locations which have to be used together if the memory limit is reached. Our randomized approach to adaptive reallocation of the memory resources provides good performance and a stable behavior. Our algorithm allows learning good control policies while being simple to implement and efficient computationally and memory-wise.

One of the issues that we will address in the future is the automatic selection of the activation radii for the SDM locations while allowing them to vary across the state space. Our resource allocation mechanism is based only on the distribution of the inputs. In the future, we also plan to explore mechanisms that also use information about the function shape (e.g., function linearity, decision boundaries, as in [18]), so that the memory layout is adjusted taking into account the complexity of the target function. Finally, we will investigate the theoretical properties of the SDM model starting from currently available results in [29, 30].

References

1. Anderson, C. (1993). Q-learning with hidden-unit restarting. *NIPS,* 81-88.
2. Atkeson, C.G, Moore, A.W. & Schaal, S. (1997). Locally weighted learning *Artificial Intelligence Review,* 11-73.
3. Atkeson, C. G., Moore, A. W., & Schaal, S. (1997). Locally weighted learning for control. *Artificial Intelligence Review*, 75–113
4. Blanzieri, E., & Katenkamp, P. (1996). Learning RBFNs on-line. *ICML*, 37-45.
5. Dietterich, T. G., & Wang, X. (2001). Batch value function approximation via support vectors. *NIPS*, 444-450.
6. Engel, Y., Mannor, S. & Meir, R. (2003). Bayes meets Bellman: The Gaussian process approach to temporal difference learning. *ICML*, 154-161.

7. Flachs, B., & J.Flynn, M. (1992). *Sparse adaptive memory* (Tech. Rep. 92-530). Computer Systems Lab., Dptm. of Electrical Engineering and Computer Science, Stanford University.
8. Forbes,J.R.N. (2002). *Reinforcement learning for autonomous vehicles.* Ph.D. Thesis, Computer Science Department, University of California at Berkeley.
9. Fritzke, B. (1997). A self-organizing network that can follow non-stationary distributions. *ICANN*, 613-618.
10. Gordon, G.J. (1995). Stable function approximation in dynamic programming. *ICML*, 261-268.
11. Gordon, G.J. (2000). Reinforcement learning with function approximation converges to a region. *NIPS*, 1040-1046.
12. Hely, T.A., Willshaw, D.J. & Hayes, G.M.(1997). A new approach to Kanerva's sparse distributed memory. *Neural Networks*, *3*, 791-794.
13. Kanerva, P. (1993). Sparse distributed memory and related models. In M. Hassoun (Ed.), *Associative neural memories: Theory and implementation*, Oxford University Press, 50-76.
14. Kondo, T., & Ito, K. (2002). A reinforcement learning with adaptive state space recruitment strategy for real autonomous mobile robots. *IROS*.
15. Kretchmar, R. & Anderson, C. (1997). Comparison of CMACs and RBFs for local function approximators in reinforcement learning. *IEEE Int. Conf. on Neural Networks*, 834-837.
16. Lagoudakis, M.G., & Parr, R. (2003). Reinforcement learning as classification: Leveraging modern classifiers. *ICML*, 424-431.
17. Martin, M. (2002). On-line support vector machine regression. *ECML*, 282-294.
18. Munos, R., & Moore, A. (2000). Variable resolution discretization in optimal control. *Machine learning*, 49, 291-323.
19. Platt, J. (1991). A resource-allocating network for function interpolation. *Neural Computation*, *3*, 213–225.
20. Ralaivola, L., & d'Alche Buc, F. (2001). Incremental support vector machine learning: a local approach. *ICANN*.
21. Rao, R. P.N., & Fuentes, O. (1998). Hierarchical learning of navigational behaviors in an autonomous robot using a predictive SDM. *Autonomous Robots*, *5*, 297-316.
22. Ratitch, B., Mahadevan, S. & Precup, D. (2004). Sparse distribute memories as function approximators in value-based reinforcement learning: Case studies. *AAAI Workshop on Learning and Planning in Markov Processes.*
23. Reynolds, S. I. (2000). Decision boundary partitioning: variable resolution model-free reinforcement learning. *ICML*, 783-790.
24. Santamaria, J. C., Sutton, R. S., & Ram, A. (1998). Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, *6*, 163–218.
25. Scholkopf, B. (2000). The kernel trick for distances. *NIPS*, 301–307
26. Smart, W. & Kaelbling, L.P. (2000). Practical reinforcement learning in continuous spaces *ICML*, 903-910.
27. Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning. An introduction.* The MIT Press.
28. Sutton, R. S., & Whitehead, S. D. (1993). Online learning with random representations. *ICML*, 314–321.
29. Szepesvari, C. & Smart, W. D. (2004). Convergent value function approximation methods. http://www.sztaki.hu/∼szcsaba/papers/szws_icml2004_rlfapp.pdf
30. Tsitsiklis, J.N. & Van Roy, B. (1996). Feature-based methods for large scale dynamic programming. *Machine Learning*, pages 59-94.
31. Tsitsiklis, J.N. & Van Roy, B. (1997). An analysis of temporal difference learning with function approximation. *IEEE Transactions on Automatic Control*, *42*, 674-690.
32. Uther, W. T. B. & Veloso, M. M. (1998). Tree based discretization for continuous state space reinforcement learning *AAAI*, 769–774.