Value-Based Reinforcement Learning algorithms in Sparse Distributed Memories to solve the Mountain-Car Problem

Author: Arnau Francí i Rodon.
Advisor: Paolo Pellegrino
Facultat de Física, Universitat de Barcelona, Diagonal 645, 08028 Barcelona, Spain.*

Abstract: In the framework of digital electronics optimization of the memory resources used is a crucial issue. Therefore many Control algorithms are studied in order to improve the trade-off between computational power and memory requirements. In this work we explore some possibilities to improve current state-of-the-art Temporal-Difference (TD) Reinforcement Learning (RL) strategies. We made use of a type of local function approximation structures known as Sparse Distributed Memories (SDMs). The interest of this investigation underlies on the belief that SDMs architectures can help to avoid the exponential increase of memory sizes due to a linear increase in the state's variables. Because RL doesn´t rely in prior information of the environment this is a frequent problem for these algorithms, as a lot of different features can appear to play a role when in fact only few of them are really relevant for the agent; a sampling of the states along with a method to generalize unseen states' values becomes a must.The main achievement has been a method capable to distribute the memory locations which ensured that regions in the state space more needed had a more intense coverage, with the purpose to improve approximations' resolution while keeping low memory requirements and high-dimensional scalability. We gave attention also to another issues as the reduction in the number of parameters.

I. INTRODUCTION

A. Reinforcement Learning and Sparse Distributed Memories

As a branch of Machine Learning, Reinforcement Learning [RL] is a computational approach that learns from interactions with the surrounding environment and concerned with sequential decision making in unknown environments to achieve high-level goals. Usually, no sophisticated prior knowledge is available and all required information to achieve the goals ought to be obtained through trials, that is Reinforcement Learning uses experience instead of Dynamics to pinpoint optimal policies.

Value-based RL use a function approximator to represent the value function and a policy that is based on the current deducted values (which won't match in general with the real values). Function approximation for value functions guarantees a fast, incremental learning that allows to learn during the interaction. While the agent interaction progresses both the input distribution and the target outputs change, therefore the function approximator must be able to handle non-stationarity very well. Consequently a lot of RL applications use linear and memory-based approximators.

The Reinforcement Learning is meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision-maker is called the *agent*. It interacts with the *environment.* They constantly interact, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent. The environment also gives rise to rewards, numerical values that the agent must try to maximize over time. A complete specification of an environment defines a *task*, one instance of the RL problem

A standard RL task for our studies can be defined as $(\mathcal{S}, \mathcal{A}, r)$, $\mathcal{S}$ being the state space, $\mathcal{A}$ the set of actions, which can be discrete or continuous, and a set of possible rewards depending on the state-action performed, $\mathcal{R}: \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

The agent and the environment interact at each of a sequence of discrete time steps $t = 0, 1, 2 \ldots$ At each time step t the agent receives some representation of the environment's state, $s_t \in \mathcal{S}$, and on that basis selects an action $a_t \in \mathcal{A}_{s_t}$. After that, as a function of the pair (s_t, a_t) the agent receives a numerical *reward* $r \in \mathcal{R}$ and the agent find itself in a new state s_{t+1} and the cycle starts again.

The goal of the agent is to maximize the total *Return* $[R_t]$ defined as the expected cumulated reward from s_t up to a

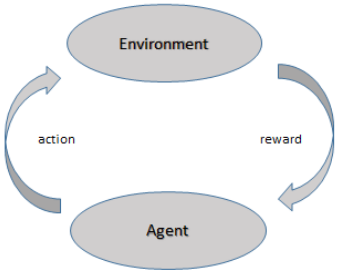

Figure 1. RL setup: the agent interacts with the environment by taking actions.

terminal state:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Where γ is the discount rate and acts as a weight to determine the present value of future rewards.

* Electronic address:afrancro13.alumnes@ub.edu

Temporal-Difference Learning. SARSA

One of the solution methods best fitted for Reinforcement Learning approaches is the Temporal-Difference (TD) Learning. TD methods can learn directly from raw experience without a model of the environment's dynamics and they update estimates based partially based on other learned estimates (bootstrapping).

A particular on-policy control method of TD Learning is the State-Action-Reward-State-Action (SARSA), which owes its name to the fact that uses the information from the quintuple of events $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ to update the values $[Q(s,a)]$. This can be formulated as

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + Q(s',a')] \qquad (1)$$

The general form of the SARSA control algorithm is expressed in *Pseudocode 1*. To shorten following algorithms we will refer to this whole process as simply "SARSA".

Initialize $Q(s,a)$

For each step

Initialize s

Select a using policy derived from Q

For each step

Take a , observe r, s'

Choose a' from s' using policy derived from Q

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

Only for the activated locations

$s \leftarrow s',\ a \leftarrow a'$

Until s is terminal

Pseudocode1. The SARSA algorithm

B. Mountain-Car Problem

The problem has many variations and definitions, and here is followed the most popular, the one defined in p. 214 of [3] but with fixed and particularly challenging initial conditions instead of random initial conditions. The task consists in driving an underpowered car up a steep mountain. The difficulty is that gravity is stronger than the car's engine, and even at full throttle the car cannot accelerate up the steep slope. The solution passes for swinging the car back and forward until it achieves enough inertia to climb the mountain with its power.

The mountain surface is given by $\sin(3x)$, the gravity value is $g = 2.5 \cdot 10^{-3}$ and the car can choose among 3 actions, $action \in \mathcal{A}$, where $\mathcal{A} = \{+1$ (full throttle forward), 0 (zero throttle), -1 (full throttle reverse) $\}$; the acceleration value resulting from this actions is $a = 10^{-3} \cdot action$

The state space is continuous, with $\mathcal{S} = \forall(x,v) |\ x \in [-1.2, 0.5], v \in [-0.07, 0.07]$. The forces diagram is pictured in fig(1) and we chose a moving Reference System, centered on the position of the car and with the x-axis parallel to the tangent line of the curve in each location. The forces balance, considering for simplicity a mass $m = 1$, is then

$$F = g \sin \phi_2 + a \qquad (B.1)$$

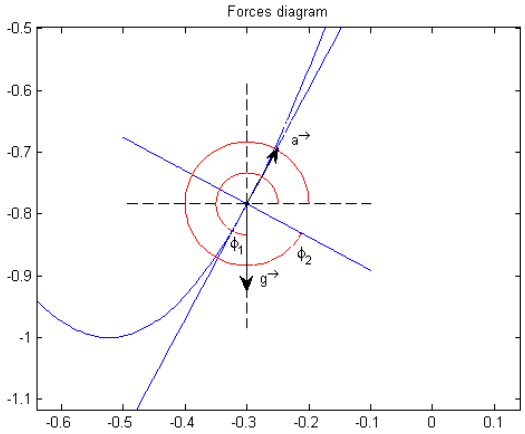

Figure 2 Forces diagram of the Mountain-Car Problem

$$F = -g\cos(3x) + a \qquad (B.2)$$

From here we can integrate by discrete steps

$$v_{t+\delta t} = v_t + \int_t^{t+\delta t} (-g\cos(3x) + a)\ dt \cong v_t + (-g\cos(3x_t) + a_t) \cdot \delta t \qquad (B.3)$$

$$x_{t+\delta t} = x_t + \int_t^{t+\delta t} \left(v_t + \int_t^{t+\delta t} -\mathrm{gcos}(3x) + a\ dt \right) dt \cong v_t \delta t + (-g\cos(3x_t) + a_t) \cdot \delta t^2 \qquad (B.4)$$

Taking $\delta t = 1$ that leads to

$$v_{t+1} \cong v_t - g\cos(3x_t) + a_t \qquad (B.5)$$

$$x_{t+1} \cong x_t + v_{t+1} \qquad (B.6)$$

Where (B.5) and (B.6) are the discrete time step dynamic equations used to solve the problem with numerical methods

II. MODEL PROPOSED TO MERGE SARSA AND A LOCAL APPROXIMATOR

Thestandard structure of the models can be split in three fundamental blocks:

i. Establish a function to determine the weight of each address in the memory when calculating action-value function in a state action, the "approximator"

ii. Allocate the memory and which memories should be suppressed and how to reallocate them when the memory limit is reached or some condition is accomplished

iii. The Reinforcement Learning algorithm

Even though the different sections often are thought with some degree of interdependence in an optimal method, it is also true that they admit independent analysis. In our case the Reinforcement Learning algorithm is always going to be the same, and the changes will spin mainly around the parts related to the memory distribution and also a little about the part related with the approximator too. Evaluating them individually can be helpful when comparing the overall results for each method. In this chapter (i) and (iii) will be introduced. Chapter III is entirely dedicated to (ii) as it has been the target to improve in the project.

In the model the inputs can be viewed as an "address" and the output is the desired content to be stored at that address; in our case this will correspond to some of the value functions (Vor Q) that both ae functions of the type:

$$f(\boldsymbol{s})\colon R^n \to R \tag{2}$$

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 $\boldsymbol{s} = (s_1, \dots, s_n)$ has to be accessed, a set of nearby locations $\{\boldsymbol{\mathcal{H}}\}_{\boldsymbol{s}} = \{\boldsymbol{h^1} = (h_1^1, \dots, h_n^1), \dots \boldsymbol{h^k} = (h_1^k, \dots, h_n^k)\}$ is activated, as determined by a *Similarity* measure. It represents the location's degree of activation, continuous in [0,1] for both definitions presented.

The Similarity measure can be defined in many ways as shown in [8]. Gaussian weightings are used in [1] and triangular in [4]. In my tests (not shown here due to extension constraints) I tried both triangular and hyperbolic and opted for the hyperbolic. They have been chosen because of their lightness.

An apparent drawback of the hyperbolic weighting is that it would diverge if an agent state happened to land over a state stored in the hard memory. However as we are dealing with continuous spaces the probability to land over exactly one of the points of the memory is infinitesimal. In contrast it is expected to give a finer response giving much more weight to closer points while keeping the simplicity in the calculations.

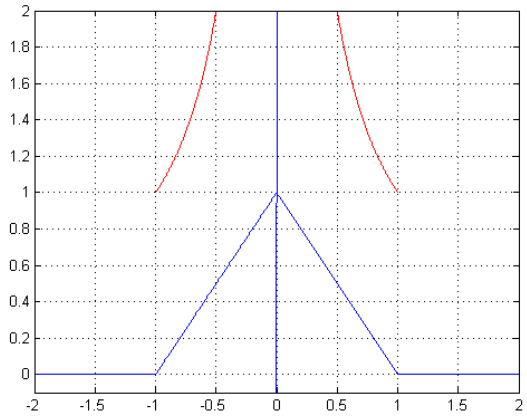

Figure 3 Hyperbolic (red) and triangular (blue) functions

- Triangular

$$\sigma_i(\boldsymbol{h}, \boldsymbol{s}) = \begin{cases} 1 - \dfrac{|s_i - h_i|}{w_i} & \text{if} |s_i - h_i| \le w_i \\ 0 & \text{if} |s_i - h_i| > w_i \end{cases} \tag{3}$$

- Hyperbolic

$$\sigma_i(\boldsymbol{h}, \boldsymbol{s}) = \begin{cases} \dfrac{w_i}{|s_i - h_i|} & \text{if} |s_i - h_i| \le w_i \\ 0 & \text{if} |s_i - h_i| > w_i \end{cases} \tag{4}$$

Where

$$\sigma(\boldsymbol{h}, \boldsymbol{s}) = \min_{i=1,\dots n} \sigma_i(\boldsymbol{h}, \boldsymbol{s}) \tag{5}$$

andw_i is Γ_i for radii methods and the range $s_{i_{max}} - s_{i_{min}}$ for the kNN methods.

To predict the value of input $\boldsymbol{s}$we first find the set of active locations$\boldsymbol{\mathcal{H}_s}$. Being $\sigma^k \equiv \sigma(\boldsymbol{h^k}, \boldsymbol{s})$ and θ^k the value stored in $\boldsymbol{h^k}$ the predicted value of $\boldsymbol{s}$can be computed in the same form that Normalized RBFNs:

$$f(\boldsymbol{s}) = \frac{\sum_{k\in\mathcal{H}_s} \sigma^k \cdot \theta^k}{\sum_{k\in\mathcal{H}_s} \sigma^k} \quad (6)$$

Upon receiving a sample $\langle \boldsymbol{s}, f(\boldsymbol{s})\rangle$ the values stored in all active locations will be updated using the standard gradient descent for linear approximation

$$\alpha = \left(\nabla f_\theta(s)\right)_i \equiv \frac{\sigma\,(s_i, s)}{\sum_{s_k\in H_s} \sigma\,(s_k, s)} \quad (7)$$

Combining this with SARSA (λ) algorithm, the values stored in the memoryθ^m are be updated after every transition $(\boldsymbol{s}, a) \xrightarrow{r} (\boldsymbol{s}', a')$ by the action- value function $Q(\boldsymbol{s}, a)$

$$\theta^m(\bar{a}) \leftarrow \theta^m(\bar{a}) + \alpha e_m(\lambda, \bar{a})[r + \gamma Q(s', a') - Q(s, a)] \quad (8)$$

$\forall \bar{a} \in A$and $m = 1, \dots, M_{\bar{a}}$, where $e_m(\lambda, \bar{a})$ are the eligibility traces for each location. In the case of replacing traces the extinction term is λ and $\frac{\sigma^m}{\sum_{k\in H_{s(a)}} \sigma^k}$ for the performed action a.

The policy followed in all the codes implemented is $\epsilon-$ greedy with different values of ϵ which have been optimized for each of the models.

III. MEMORY STRUCTURE

In this section the memory layout proposed in the paper to work along with the previous algorithms is explained in detail, pointing out how the allocation, suppression and reallocation procedures undergo.

We have a radius of activation **Γ**and a minimum number of locations N^* in this radius neighborhood are desired at any position. Therefore if the actual number of locations in the neighborhood for a certain address is $N_{actual} < N^*$ we follow the heuristic procedure below

1. Add the current state and its value as a memory location
2. If $N_{actual} < N^*$ then add $N^* - N_{actual}$ addresses randomly within the $[\boldsymbol{s} \pm \boldsymbol{\Gamma}]$ limits and store there the value that the approximator returns for the current memory layout (target value).

All of them but need to accomplish a condition: there is a limit on the similarity that two locations in the memoryh_i , h_j can have, that is for any pair of locations stored they must

$$\sigma(h_i, h_j) < \sigma_{max} \quad (9)$$

So if when adding locations the part 1 of the procedure violates this condition we go directly to the part 2 of the procedure; 2 is repeated until having N^* addresses in the neighborhood that respect the condition. The σ_{max} value is established as

$$\sigma_{max} = 1 - \frac{1}{N-1} \quad (10)$$

Or in other terms, the locations need to be spaced within at least a distance $d = \frac{r}{N-1}$

When the memory limit is reached, the previous procedure is followed but before a vacant needs to be created in the memory, that is one location has to be suppressed; that is done taking randomly a memory address which must be out of the r neighborhood of s_a, $h \mid h \backslash \notin \{\mathcal{H}\}_{s\_a}$

IV. SIMULATIONS AND RESULTS

To test our procedure we use the results for the method described in [4]as a reference to qualify our success. We tried one of the different combinations of Γ and N^* proposed in [5], that is $\boldsymbol{\Gamma} = (0.425, 0.035)$ and$N^* = 2$. For the rest of the parameters they were set up as indicated in the paper $\lambda = 0$, $\gamma = 1$ excepting for α and ϵ the values of whom are not provided.

The method also appears to be very sensitive to the exploration parameter, what adds another parameter to be adjusted, and no clue is given in the paper about how to do that (just in p.355 is mentioned that the parameter needs to be adjusted for each case). After some attempts, even though any specific value showed a striking improvement it seemed that the best way was to work with decreasing ϵ starting with high values for it, from 0.1 to 0.25. The ratio of decay can be set too in many different ways but after trying 0.99 0.995 0.999 and 0.9999 the second one was selected for the Mountain Car problem. The problem usually is that greater exploration is needed at the beginning and taking many random movements can help to this, but after some iterations a great epsilon undermines learning.

I finally set them as $\alpha = 0.01$ and $\epsilon = 0.2 \cdot 0.995^t$, where t is a counter of the number of iterations or steps taken by the agent and acts as a decay rate. For the last 10% episodes (in that case, the last 200) ϵ was reset to 0. Starting with $\epsilon = 0$ the algorithm cannot solve the problem.

The results are an average of 28 runs of 2000 episodes each one. Three main aspects were tested: the evolution in the agent's ability to solve the task, the evolution of the memory size and the final memory distribution.

The results of the agent performance over the episodes are shown in

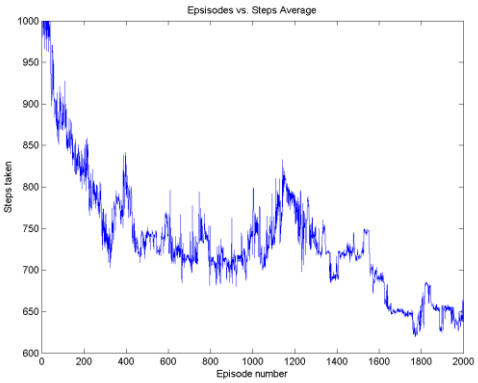

Figure 5. Steps taken in each episode for a trial, averaged of the 28 trials. Even though the individual graphs display an intense noise, the progression (learning) over the episodes is

16 successful (solved the task) and 12 failed (did not solve the task in any episode). That means that almost one of each two trials failed (43%). Both my method and [5] seem to have similar variance and evolve according the same pattern. In my experiment the total number of steps needed to finish each episodeis slightly greater than in their paper; I don't know whether it can be related with a bad tuning of the parameters ϵ and α for the settings of Γ and N^* or not. As no comment about how to adjust these parameters is provided in the paper, there is no way to know it.

Nonetheless a noticeable difference can be observed in the size of the memories. While the maximum memory size for this r and N^*according to their results is $|\mathcal{H}|_{max} = 23$ locations, for all the successful runs in my experiments the number of memory addresses are around

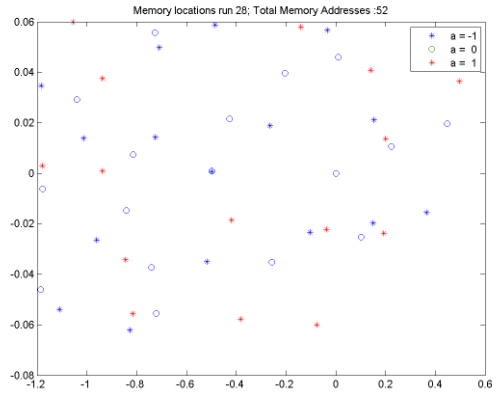

Figure 4 Example of memory layout for a successful run; x-axis position, y-axis velocity

$|\mathcal{H}| \in [40,55]$ and the minimum size and only out of this range (14^{th} run) is $|\mathcal{H}| = 21$. If 23 is the average of the memory size then the values would match too, but in the paper it's said to be the "maximum achieved over the runs".

V. CONCLUSIONS AND FUTURE WORK

The experiments demonstrate the capacity of RL agents using SDM local approximators to learn a complex continuous task. Both density-based reallocation strategies are robust in the framework of RL and produce acceptable representations of the value function.

The strategies should be implemented in other benchmark problems, like the simulations in the Cart-Pole pendulum on going at the present moment, in order to test their dimensional scalability. Another line to explore is the possibility of auto associative memories [3].

Present models show a great level of adaptation to fast-changing environments and ability to handle reduced amounts of hard memory addresses, but further research needs to be done to ensure an acceptable degree of approximation's resolution in some cases.

Acknowledgments

My most sincere gratitudeto my advisor Paolo Pellegrinofor giving me support during all these months in all the different difficulties that have arisen, beyond the strictly academic issues .

[1] Forbes J.R.N. *Reinforcement learning for autonomous vehicles*. PhD Thesis, Computer Science Department, UCB, 2002

[2] Kanerva, P. *Sparse Distributed Memory and related models*. Associative Neural Memories: Theory and Implementation, pp. 50-76, 1993.

[3] Kohonen, T. *The Self Organizing Map*. Proceedings of the IEEE, vol. 78, September 1990.

[4] Ratitch, B., Mahadevan, S., & Precup, D. *Sparse Distributed Memories as function approximators in value-based reinforcement learning: Case studies*. AAI Workshop on Learning and Planning in Markov Processes, 2004.

[5] Ratitch, B., & Precup, D. *Sparse Distributed Memories for On-line Value-Based Reinforcement Learning*, 2004.