"Mountain Car"
Project 2 - Unsupervised and Reinforcement Learning in Neural Networks

Dario Pavllo - 273251 - dario.pavllo@epfl.ch

***Abstract*—This project aims to solve the *mountain car* problem (a standard benchmark for reinforcement learning techniques) using the SARSA(λ) algorithm and Gaussian radial basis functions to approximate continuous states. According to various experiments, the best technique uses a softmax policy with a decaying temperature parameter, eligibility traces with high decay factors, and optimistic weight initialization.**

I. Introduction

The *mountain-car* task is a well-known problem in the reinforcement learning domain, and it is defined as follows: an underpowered car is placed at the bottom of a convex landscape, and it must drive up a hill. However, since the car does not have enough power to reach the goal position directly, it must oscillate backwards and forwards in order to accumulate sufficient potential energy. Initially, the car does not know how to reach the goal, and it gradually learns by trial and error by means of a reinforcement learning technique. Each time the top of the hill is reached, the agent (i.e. the car) receives a positive reward and assimilates the actions that lead to the final result.

The car is described by its state, namely its position x and its speed $\dot{x}$. Moreover, at any time instant, the car can either accelerate forwards, accelerate backwards, or remain in neutral gear (i.e. do nothing).

One of the challenging aspects of this problem is represented by the continuous state space, as common reinforcement learning techniques such as *SARSA* or *Q-Learning* require a discrete state space. Therefore, the continuous problem must be approximated as a discrete problem, and this can be practically achieved in two ways:

- Discretization: the state space is subdivided into a set of buckets (a regular grid), and the agent is assigned to the bucket associated with its current state, that is, the nearest one.
- Function approximation: likewise, the state space is subdivided into a grid, but the agent is not assigned to one state. Instead, a set of basis functions is used to approximate the Q-function in the current state, by taking into account both the value of the nearest bucket and the nearby ones (with a weight that depends on the distance from their preferred state).

In this project, the *function approximation* technique has been deployed, along with the SARSA algorithm. At every time instant, the next action in the current state is chosen according to a softmax policy, as described in Equation (2). Furthermore, the Q-values for the current state $Q(s,a)$ are approximated using a Gaussian radial basis function Φ (see Equation (1)). The inputs are multiplied by a weight matrix $\mathbf{W}$ of size $N \times 3$, where N is the number of subdivisions of the state space along the x and $\dot{x}$ axes, i.e. $N = N_x \cdot N_{\dot{x}}$, and 3 is the number of possible actions, namely forwards, backwards and neutral. The result of the matrix multiplication $\mathbf{W}\,\Phi(x,\dot{x})$ is a vector of length 3 that represents the Q-values for next action $Q(s,a')$. The latter are supplied to the policy function Π, which returns the output probabilities for each action according to the softmax rule. This model can be viewed as a simple neural network with no hidden layers, i.e. a single layer perceptron, where every cluster corresponds to an input neuron.

II. Implementation details

The state space has been segmented into $N_x = 20$ neurons for the x dimension (linearly spaced along the $[-150\,m,\ \ 30\,m]$ range), and $N_{\dot{x}} = 5$ neurons for the $\dot{x}$ dimension ($[-15\,m/s,\ \ 15\,m/s]$ range), for a total of $N = 100$ neurons. The agent receives a reward once it reaches a state that satisfies $x \geq 0$.

Algorithm 1 describes the implemented training procedure, including the SARSA(λ) update rule. For every training trial, the car is spawned at a random position and the simulation runs until the reward is obtained.

The parameters and functions of the algorithm are defined as follows:

- N: number of input neurons ($N_x \cdot N_{\dot{x}}$)
- $\mathbf{W}$: neural weight matrix, which encodes $N \times 3$ weights (one for every state-action pair)
- λ: eligibility trace decay factor ($0 \leq \lambda \leq 1$)
- η: learning rate ($0 < \eta < 1$)
- γ: discount factor for future rewards ($0 \leq \gamma \leq 1$)
- $\Phi(s)$: approximation function that maps the current state to the neural inputs. It is a Gaussian function, defined as follows:

$$\Phi_i(s) = \Phi_i(x,\dot{x}) = \exp\left(-\frac{(x-x_i)^2}{\sigma_x^2} - \frac{(\dot{x}-\dot{x}_i)^2}{\sigma_{\dot{x}}^2}\right) \quad (1)$$

where the pair $(x_i, \dot{x}_i)$ corresponds to the preferred state of the i-th neuron and $(\sigma_x, \sigma_{\dot{x}})$ are defined as

Algorithm 1 SARSA(λ) with continuous states

```
1:  procedure LearningTrial($\eta$, $\gamma$, $\lambda$, $\mathbf{W}[N \times 3]$)
2:      Reset car
3:      $\mathbf{e} \leftarrow \mathbf{0}$                ▷ Initialize eligibility trace ($N \times 3$)
4:      $s \leftarrow (x, \dot{x})$                         ▷ Random initial state
5:      $a \leftarrow \Pi(\mathbf{W}\,\Phi(s))$ ▷ Next action according to policy
6:      while no reward do
7:          Apply action $a$ to car
8:          Observe state $s' = (x, \dot{x})$
9:          Observe reward $r$
10:         $a' \leftarrow \Pi(\mathbf{W}\,\Phi(s'))$
11:         $\Delta Q(s,a) \leftarrow \eta(r + \gamma Q(s', a') - Q(s,a))$
12:         $\mathbf{e} \leftarrow \gamma\lambda\mathbf{e}$
13:         $\mathbf{e_a} \leftarrow \mathbf{e_a} + \Phi(s)$
14:         $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{e} \cdot \Delta Q(s,a)$
15:         $s \leftarrow s'$
16:         $a \leftarrow a'$
17:     end while
18: end procedure
```

the distance between the clusters. The function $\Phi(s)$ returns a vector of length N.

- $\Pi(\mathbf{a})$: policy that selects the next action given the current state s and the set of Q-values for the possible actions in the current state. In this case, the probability that the action a is chosen is dictated by the following *softmax* function:

$$P(a) = \frac{\exp(Q(s,a)/\tau)}{\sum_{a'} \exp(Q(s,a')/\tau)} \tag{2}$$

where τ is the *temperature* parameter. Low values of τ encourage the policy to pick the action associated with the highest Q-value, whereas high values of τ encourage the policy to select a random action (since all probabilities tend to the same value). In order to avoid numerical stability issues when too low or too high values of τ are used, the softmax function has been implemented in log scale:

$$P(a) = \exp\left(\frac{Q(s,a)}{\tau} - \ln \sum_{a'} \exp\left(\frac{Q(s,a')}{\tau}\right)\right) \tag{3}$$

where the second term with the logarithm is calculated using the *log-sum-exp* trick.

III. Experiments

In order to find the best parameters for this task, several experiments have been conducted. Each experiment consists of 10 agents that run the simulation in parallel (using a *thread pool* to maximize performance). To ensure that the results are reproducible, every agent uses a different pseudo-random number generator instance and a unique seed.

The objective is to minimize the average *time to escape* (or *escape latency*), i.e. the number of time steps it takes for the car to obtain the reward.

For all the experiments mentioned below, unless specified differently, the baseline parameters are $\eta = 0.01$, $\gamma = 0.95$, $\lambda = 0.5$, initial $w_{ij} = 0$.

A. Temperature parameter

The *temperature* parameter τ of the softmax policy determines the trade-off between exploration and exploitation. A high value encourages the former, whereas a low value encourages the latter. One extreme case is $\tau = 0$, which is equivalent to $\arg\max$, i.e. the action associated with the highest Q-value is picked (greedy strategy); the other extreme case is represented by $\tau = +\infty$, which causes the policy to always pick a random action. It is reasonable to believe that both the extreme cases would lead to sub-optimal behaviour, as the former does not allow the agent to sufficiently explore the state space, and the latter does not involve any intelligent behaviour.

The constant values $\tau = +\infty$, $\tau = 1$, $\tau = 0.1$, $\tau = 0.01$, and $\tau = 0$ have been tested. The first three have proved to be too high and are not reported here, as they cause the agent not to learn anything (even after a large number of trials). $\tau = 0.01$ (shown in Figure 2) is effective, although it a requires many trials to converge. $\tau = 0$ causes the car to get stuck into a steady state, without being able to escape (because no random actions can be taken).

Furthermore, a time decaying function has been tried, which consists in an exponential decay of τ, starting from $\tau = 10$ and decaying with a time constant $\alpha = 0.8$ (Figure 3). The same experiment has been repeated with a different decay rate: $\alpha = 0.9$, which has lead to similar results (with a lower convergence speed). The function is defined as follows: at trial t, τ is equal to α^t.

The best approach seems to be the last one (with $\alpha = 0.8$), which converges to a satisfactory result after just 40 trials.

B. Eligibility trace

Once a reward is obtained, only the Q-value for the last state-action pair is updated, and this means that many iterations are needed to propagate the information down to the initial state. The eligibility trace solves this problem by updating the Q-values along the entire trajectory of state-action pairs taken by the agent. An exponential decay factor λ determines how much the old states should be updated as a consequence of the current reward. The values $\lambda = 0$ (no eligibility trace), $\lambda = 0.5$ and $\lambda = 0.95$ have been tested. The experiment with $\lambda = 0$ does not converge to a good result, whereas $\lambda = 0.5$ and $\lambda = 0.95$ behave correctly. Specifically, $\lambda = 0.95$ (Figure 4) seems to converge within the lowest number of trials.

C. *Initialization of weights*

Two weight initialization strategies have been tried:

- $w_{ij} = 0$: this is the standard approach.
- $w_{ij} = 1$: this is known as *optimistic greedy*, and consists in overestimating the Q-values. In practice, this approach encourages exploration at the beginning of the training procedure (regardless of the policy), and gradually falls back to exploitation as the weights converge to the correct values.

As can be observed in Figure 5, the optimistic greedy approach converges faster than the first one.

IV. Conclusion

In summary, the SARSA(λ) algorithm (with the function approximation approach) is suitable for the resolution of the mountain-car task, although its convergence is strongly dependent on the adopted parameters. According to the aforementioned experiments, the following parameters seem to solve the problem in the best manner:

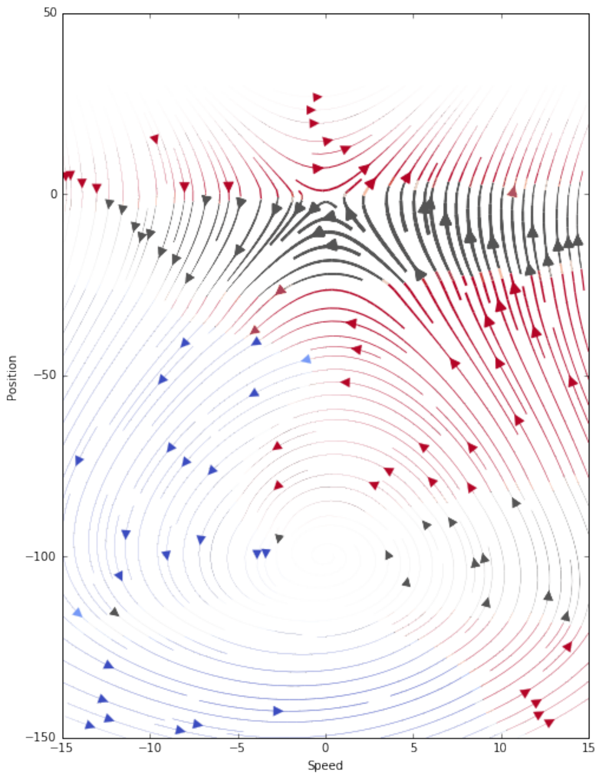
- Temperature τ: the best approach consists in starting from a high value ($\tau = 10$) that decreases exponentially with a low decay factor ($\alpha = 0.8$).
- Eligibility trace decay factor λ: high values ($\lambda = 0.95$) improve convergence speed.
- Weight initialization: initializing the weights to 1 (optimistic greedy approach) improves convergence speed by automatically adjusting the exploration/exploitation trade-off.

Figure 1 shows the behaviour learned by the agent. For each state, the action corresponding to the highest Q-value is plotted, as well as the direction of the next state. Figure 1b reveals the optimal strategy: the car oscillates forwards and backwards, and the neutral gear is never used.

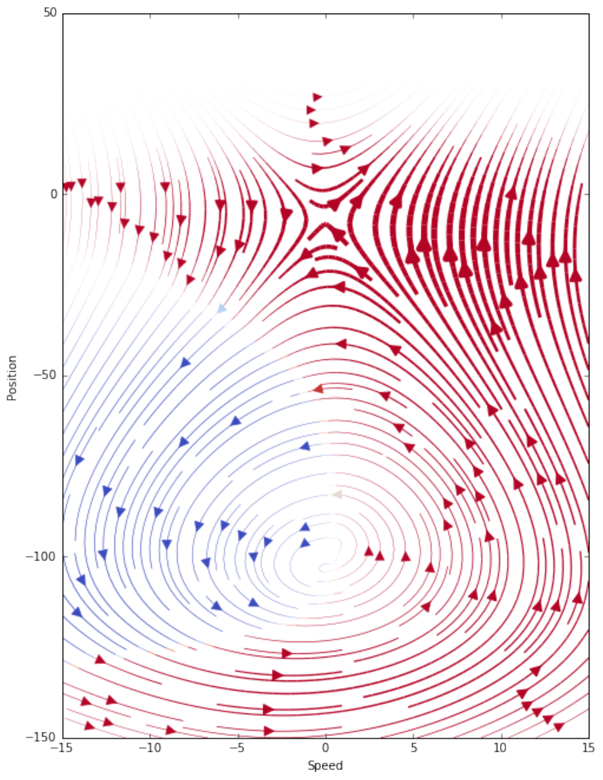
Table I reports the escape latency for each experiment, after 100 trials. The last one achieves the fastest convergence speed, as well as the lowest time to escape and the highest consistency.

τ	λ	$w_{ij}^{[0]}$	EL Average $\pm\sigma$	EL Median
0.01	0.5	0	192.6 ± 139.9	138.5
0.001	0.5	0	964.3 ± 2130.7	143.5
$10 \cdot 0.9^t$	0.5	0	163.7 ± 182.2	110.0
$10 \cdot 0.8^t$	0.5	0	108.6 ± 43.5	77.5
$10 \cdot 0.8^t$	0	0	3358.3 ± 4392.9	376.5
$10 \cdot 0.8^t$	0.95	0	67.6 ± 23.2	66.0
$10 \cdot 0.8^t$	0.5	1	66.4 ± 29.5	55.0
$\mathbf{10 \cdot 0.8^t}$	**0.95**	**1**	$\mathbf{54.3 \pm 19.4}$	**53.5**

Table I: Comparison among the experiments, after 100 trials. EL = Escape latency



(a) After 5 trials



(b) After 50 trials

Figure 1: Behaviour visualization. Legend: red = forward; blue = backward; grey = neutral

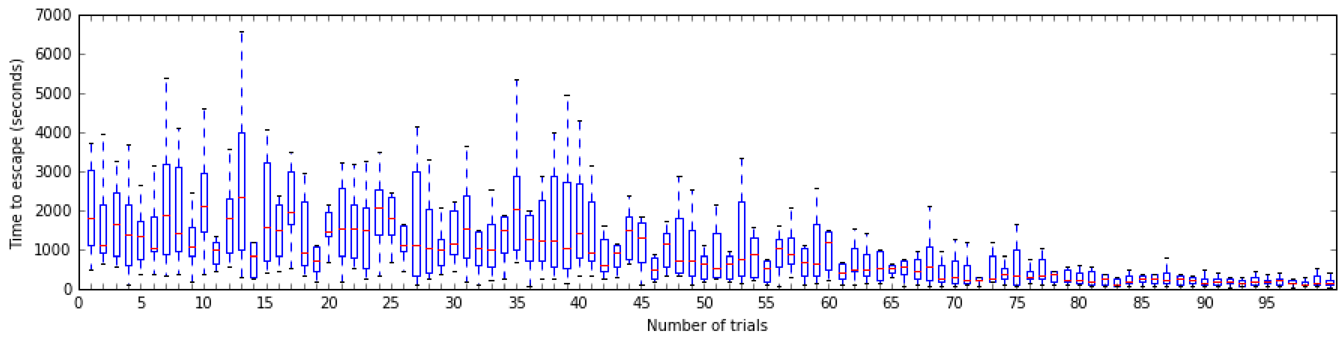


Figure 2: Constant $\tau = 0.01$; $\lambda = 0.5$; initial $w_{ij} = 0$.

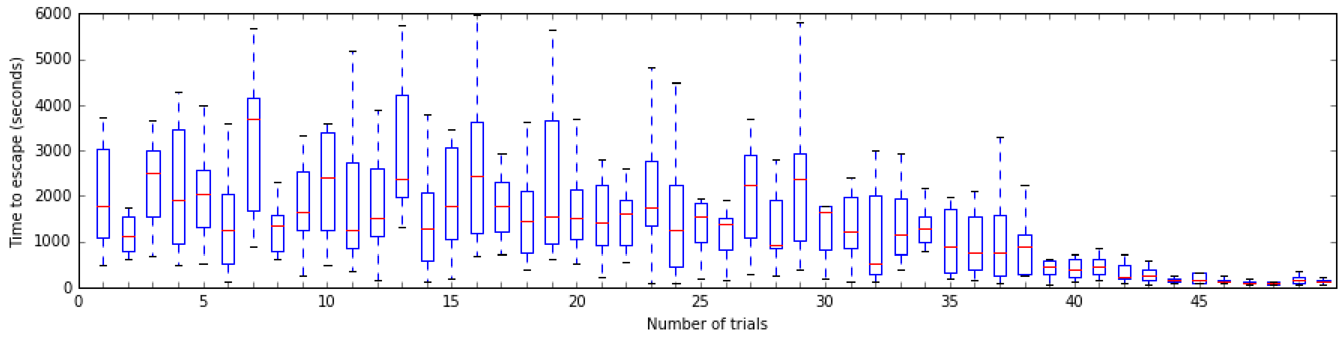


Figure 3: Decaying $\tau = 10 \cdot 0.8^t$; $\lambda = 0.5$; initial $w_{ij} = 0$.

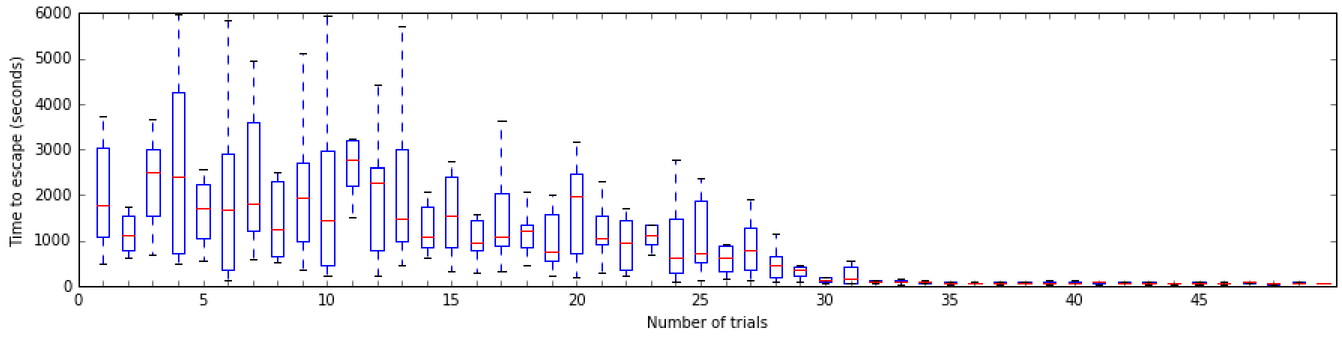


Figure 4: Decaying $\tau = 10 \cdot 0.8^t$; $\lambda = 0.95$; initial $w_{ij} = 0$.

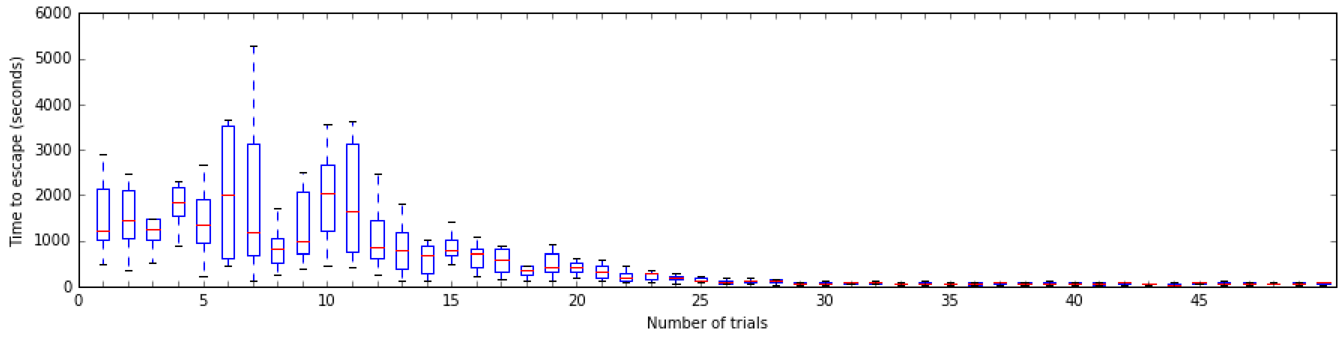


Figure 5: Decaying $\tau = 10 \cdot 0.8^t$; $\lambda = 0.5$; initial $w_{ij} = 1$. The case with $\lambda = 0.95$ is almost identical.