"SARSA(λ) algorithm applied to Mountain Car problem"
Project 2 - Unsupervised and Reinforcement Learning in Neural Networks

Mattia Martinelli
271711

***Abstract*—The *Mountain Car* problem is a well-known test in reinforcement learning. In this project, we aim to realize an agent, implementing a SARSA(λ) algorithm, that can accomplish this task successfully. The report shows results of several experiments and examines the behaviour of the algorithm according to different parameters.**

I. Introduction

The SARSA (State-Action-Reward-State-Action) is a temporal-difference (TD) algorithm widely used in reinforcement learning. In summary, the underlying idea is to explore different states of a given environment to optimize a Markov decision process. An agent using such an algorithm, known as an on-policy learning algorithm, interacts with the environment and updates the Q-values based on actions taken.

The aim of this project is to solve a *Mountain-Car* problem with a variation of the original SARSA algorithm: the SARSA(λ). In this version, a parameter λ (i.e. the *eligibility trace*) is used to accelerate convergence.

This report explores the behaviour of the agent given different parameters. Firstly, Section II provides a brief overview on the implementation of the algorithm. The rest of the document describes the experiments and is organised as follows: Section III shows the average escape latency of the agent, Section IV shows the behaviour of the agent during the learning process; finally Section V, Section VI and Section VII analyse the impact of the parameters on the algorithm.

II. Implementation of the algorithm

Conceptually, the Mountain Car problem is accomplished by a neural network, whose weights are updated by a SARSA(λ) algorithm. This network is simply composed of one input layer and one output layer, the former with a population of 100 neurons (this value was chosen according to the available computational power).

The original SARSA algorithm is conceived to work with discrete states. However, the Mountain Car problem is defined in a continuous domain, i.e. the car can assume any position and velocity at a given time. The input neural layer must adapt the current state to the continuous domain by means of a continuous activation function, in our case the Gaussian:

$$r_j(\mathbf{s}) = \exp\left(\frac{-(x_j - x)^2}{\sigma_x^2}\frac{-(\psi_j - \dot{x})^2}{\sigma_\psi^2}\right)$$

where r_j is the response of the *j*-neuron to **s**, i.e. the current state assumed by the car, expressed in terms of position *x* and speed $\dot{x}$. The quantities x_j and ψ_j represent the preferred state of the neuron, namely its coordinates in the position-velocity domain.

The output of the network **s** is given by

$$\mathbf{Q(s,a)} = \sum_j w_{a,j} r_j$$

where $w_{a,j}$ is an element of the weight matrix **W**. The action a is used to control the car and can assume one of the following values: "force to the left", "force to the right" and "no force".

Q-values are then used to generate a probability distribution that indicates what following action the agent should take. In our case, probabilities are generated with a *softmax* strategy. In more general terms, Q-values can be used to find an optimal action for any given (finite) Markov decision process. It is important to highlight that, in the SARSA algorithm, actions are chosen and Q-value updates are performed with the same policy.

The implementation of the algorithm in shown below:

Algorithm 1 SARSA(λ) algorithm

function SarsaLambda
  initialize $\mathbf{W}$
  repeat
    initialize $\mathbf{e}, Q$
    repeat
      get $Q_{t+1}(s', a'), r, a'$ ▷ Softmax policy
      $\delta_t \leftarrow R_{t+1} - (Q_t(s,a) - \gamma\, Q_{t+1}(s', a'))$
      $\mathbf{e_t} \leftarrow \gamma\, \lambda\, \mathbf{e_{t-1}}$
      $\mathbf{e_{ta'}} \leftarrow \mathbf{e_{ta'}} + r$ ▷ a' = action taken
      $\mathbf{W} \leftarrow \mathbf{W} + \eta\, \delta_t\, \mathbf{e_t}$
      Update $Q_t(s,a), s$
    until $R_t > 0$ ▷ Reward obtained
  until finish
  return $\mathbf{W}$
end function

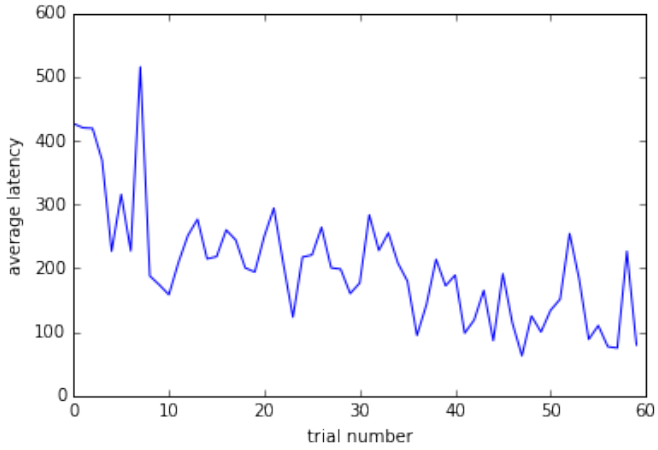


Figure 1: Average escape latency, in relation with the number of trials.

where $\mathbf{Q_t}$ are the current Q-values, $\mathbf{W}$ contains the weights, η is the learning rate, $\mathbf{e}$ is the eligibility trace and λ is the eligibility trace decay rate, i.e. how much the previous information is reduced at each step. It is important to notice that only the eligibility trace of the taken action $\mathbf{e_{t(a')}}$ is updated. When the reward is obtained, the algorithm starts a new trial with the acquired learning weights $\mathbf{W}$.

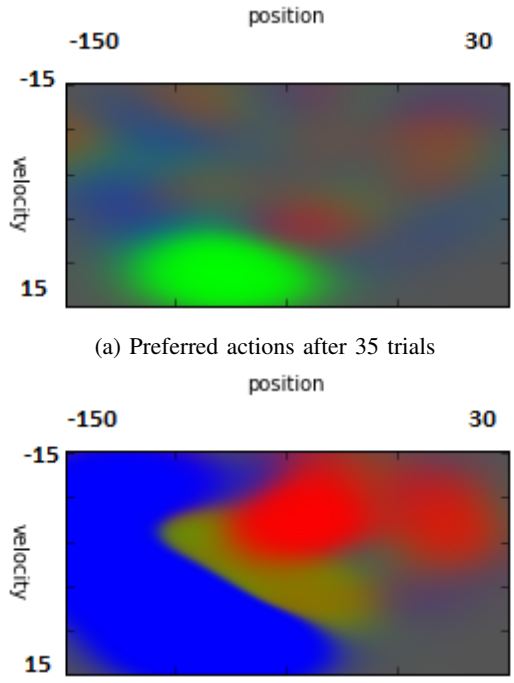
III. Escape Latency

The algorithm trains the network in a stochastic fashion, meaning that the initial state of agent is defined randomly. The required time to reach the reward, therefore, changes according to the initial state, which can either be advantageous or not. In order to study the escape latency, the algorithm is executed on 10 different agents and their results are averaged, as shown in Figure 1.

It is possible to remark that the average latency decreases over the number of trials. The values are slightly noisy, however, it can be assumed that the algorithm converges after an average of 50 trials. The agents were tested with the same parameters: $\lambda = 0.8$, $\eta = 0.01$, $\tau = 0.1$ and weights initialized to 0 (unless stated otherwise, these are the default values). These parameters where chosen empirically according to several tests, as explained in detail in the following sections.

IV. Behaviour of the Agent

Vector fields show preferred actions given a particular state. The x-axis and y-axis plot, respectively, the position and speed of the agent. Figure 2 shows how the agent learnt the actions during the training. At the end, as can be seen in Figure 2b, the field assumes well-defined regions that can explain how the agent behaves. The blue region indicates that the agent is moving toward the centre, since it occupies the left part of the scenario, while the red region indicates



(a) Preferred actions after 35 trials

(b) Preferred actions after 100 trials

Figure 2: Vector fields showing the preferred action given a state.

that the car is moving backward. It is also possible to notice that the field assumes a circular morphology, allowing the car to accumulate inertia. As the agent learns the task, neutral regions becomes less recognizable.

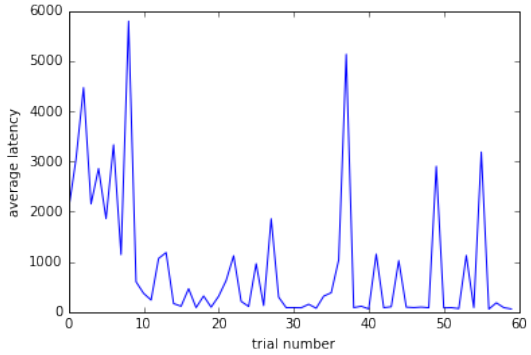
V. Exploration Temperature

The exploration temperature parameter τ is used to adjust the behaviour of the algorithm. Two different approaches are available:

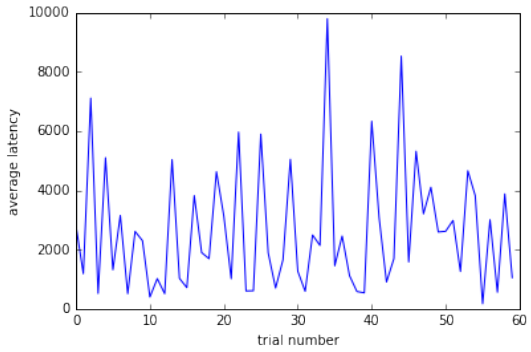
- *Exploration*: the algorithm tries to gather as much information as possible from the environment. The goal is to explore every possible state.
- *Exploitation*: the algorithm exploits the current knowledge and seeks to obtain the highest reward.

Several tests have been conducted to study how τ influences the final result and they are reported in Figure 3. It can be inferred that a τ higher or equal to 1 leads the agent to an explorative behaviour. The final result is highly noisy since actions are taken randomly. On the contrary, $\tau = 0.01$ leads the agent to an exploitative behaviour, improving the final result.

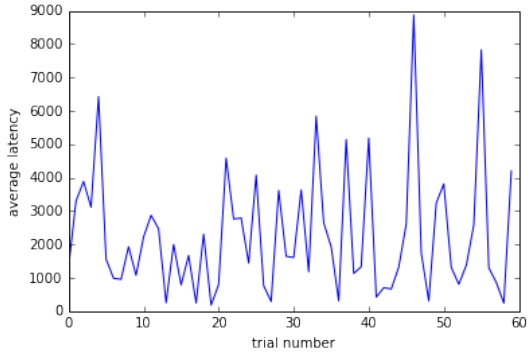
As last test, the agent was trained with a decaying τ. This approach may be more advantageous. The agent is started with τ approximately equal to 1 and then decreased over time. During the first trials, higher values of τ favourite exploration; as the number of trials increases, lower values



(a) $\tau = 0.01$



(b) $\tau = 1$



(c) $\tau = 100$

Figure 3: Learning curves with different fixed values of τ.

of τ favourites exploitation. The final result is shown in Figure 4.

VI. Eligibility trace decay

The eligibility trace allows a faster back propagation of the Q-values in the algorithm. This feature can be controlled through the parameter λ, which has a significant impact on the final result, as shown in Figure 5. A low value of λ (i.e. close to 0) results in a slower convergence, which is not visible from the graph. $\lambda = 0.95$ has been shown to be an optimal value to reach convergence in a reasonable time.

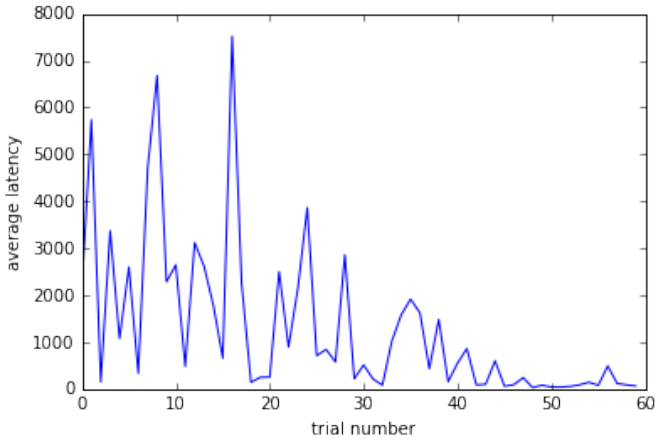
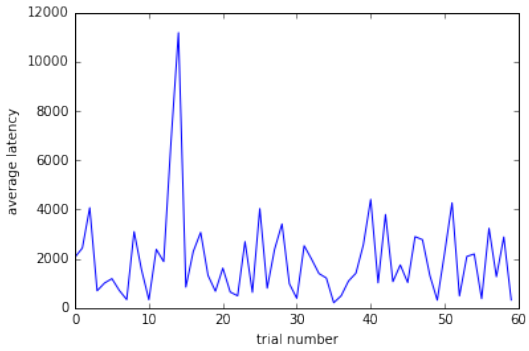
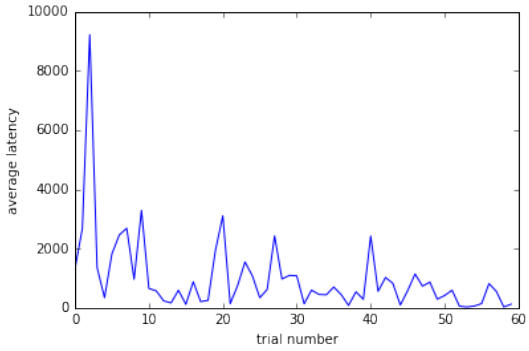


Figure 4: Learning curves with decaying τ.

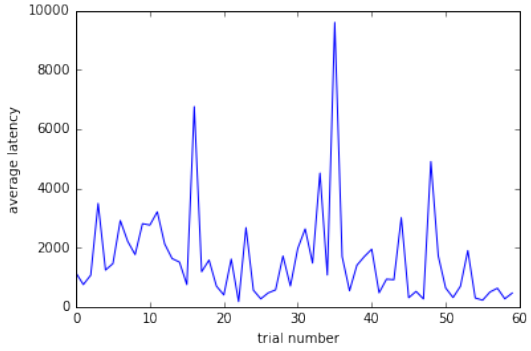


(a) $\lambda = 0$

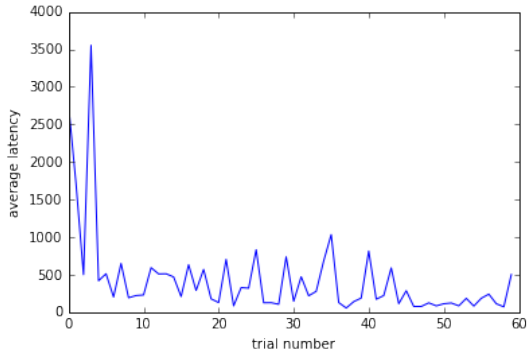


(b) $\lambda = 0.95$

Figure 5: Learning curves according to different values of λ.



(a) $\mathbf{W}$ initialized to 0



(b) $\mathbf{W}$ initialized to 1

Figure 6: Learning curves according to different initialization of the weights $\mathbf{W}$.

VII. Initialization of the weights

Weights are of fundamental importance in a neural network as they represent the synaptic plasticity. In the SARSA(λ) algorithm, weights are updated through the Q-values and their initialization can affect the final result. Some test have been conducted and results are shown in Figure 6. With weights initialized to 1, the algorithm generates better results since it overestimates the Q-values and leads to random decisions at the beginning of the training procedure, thereby favouring exploration.

VIII. Conclusion

Except for the first experiment, where 10 agents were trained at a time, sometimes tests led to noisy results. However, the following assumptions can be stated:

- The algorithm is, in general, highly sensitive to the input parameters. Even small differences can completely change the final result.
- λ and η are significantly important for the convergence of the algorithm. They must be chosen carefully in order to assure the expected results. In addition, also the weight initialization plays a crucial role.
- τ can be used to change the behaviour of the algorithm. In our case, a decaying τ has shown to produce satisfying results.