Reinforcement Learning: SARSA Algorithm Applied to Pathfinding inside the Morris Water-Maze

Student: ***Bojan Karlaš*** *Date:* ***January 2016***

This project is aimed at utilizing reinforcement learning to solve a problem of pathfinding within a square, unit-area arena, inspired by the Morris water-maze behavioral procedure. The given task is *episodic*, and in each trial the agent (i.e. the rat) is placed on the same starting position $\vec{s}_{t_0} = [0.1 \quad 0.1]^T$ and at each time step makes a step of $l = 0.03$ in one of the $D = 8$ directions until it reaches the goal area at which point the trial is finished. The goal area is a circle centered at $\vec{g}_c = [0.8 \quad 0.8]^T$ with radius $g_r = 0.1$. As the state of the agent is defined by its position $\vec{s}_t \in \mathbb{R}^2$, and its actions are movements in one of the chosen directions, we can see that the state transitions in this environment are *fully deterministic*, given the action chosen by the agent. The rewards are also deterministic: $R_g = +10$ for reaching the goal and $R_w = -2$ for crossing the wall.

In this report, we first go over the introduction to the SARSA algorithm for reinforcement learning. After that we extend it to continuous state spaces and apply it to our particular pathfinding problem. Finally, we analyze some results of our implementation. The theoretical background presented in this report is mostly based on [1].

The SARSA Algorithm

Temporal Difference (TD) learning methods have proven themselves to be most effective for solving reinforcement learning problems. The reason for this is not only because, unlike Dynamic Programming methods, they don't assume having a model of the complete state space and can easily scale to large state spaces, but they also address the shortcomings of Monte Carlo methods because they don't need to wait until the end of an episode to update the model. Instead, they do it in real-time. From the class of TD methods, we look at the **State-Action-Reward-State-Action** algorithm (SARSA), which aims at estimating the optimal *action-value function* $Q^*(s, a)$ (also called Q-value function) while at the same time controlling the agent with a policy that follows those estimates in a greedy fashion. In other words, given the current state $s_t \in S$ of the agent, the next action that will be taken is chosen out of a set of available actions for that state $\mathcal{A}(s_t)$ as follows:

$$a_t = \underset{a \in \mathcal{A}(s_t)}{\operatorname{argmax}} Q^*(s_t, a) \tag{1}$$

These properties put the SARSA algorithm in the category of on-policy Q-learning algorithms. Its update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta \underbrace{\left(r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right)}_{\delta_t} \tag{2}$$

Here $\eta > 0$ is the *learning rate*, r_{t+1} is the reward collected upon taking action a_t and $\gamma \in [0, 1]$ is the *discount factor* which controls the foresightedness of the agent (i.e. the balance between the perceived value of an immediate reward and that of a future reward). The term marked as δ_t is called the *temporal difference (TD) error* which converges as the action-value function reaches its optimum:

$$\begin{gathered} \delta_t \to 0 \quad \Leftrightarrow \quad Q(s_t, a_t) \approx Q^*(s_t, a_t) \\ Q^*(s_t, a_t) = r_{t+1} + \gamma \cdot Q^*(s_{t+1}, a_{t+1}) \end{gathered} \tag{3}$$

Notice that the quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ of factors used to form the update rule also form the name of the algorithm.

Originally, SARSA is a single-step algorithm, meaning that at each point when the agent is about to enter a new state, the action-value update is performed only for the immediately preceding (s, a) pair. For larger state spaces, this is unacceptable as propagation of estimates is very slow which in turn results in very low convergence rate. To remedy this we use *eligibility traces*, denoted $e(s, a)$, which are basically multipliers kept for each (s, a) pair that measure how much is the action-value of that pair eligible for update. The idea is to make the more recently visited pairs more eligible, which is done by the following update rule of the eligibility trace:

$$e_t(s, a) \leftarrow \begin{cases} \gamma\lambda e_t(s,\, a) + 1, & \text{if } s = s_t \text{ and } a = a_t \\ \gamma\lambda e_t(s,\, a), & \text{otherwise} \end{cases}$$

$$\forall s \in S,\ \forall a \in \mathcal{A}(s)$$

We see that the eligibility trace is an exponentially decaying quantity which gets incremented by 1 each time the particular (s, a) pair is visited. For this reason, the parameter $\lambda \in [0, 1]$

is referred to as the *eligibility trace decay rate*. Thus, the SARSA update rule extended with the eligibility trace becomes:

$$Q(s,a) \longleftarrow Q(s,a) + \eta\delta_t e(s,a), \qquad \forall s \in S,\ \forall a \in \mathcal{A}(s)$$

Notice that the update rule is now applied to every (s,a) pair, which is made possible because the intensity of the update is controlled by $e(s,a)$. The only "temporally dependent" value is δ_t, which is calculated according to (2). This completes the description of what is known as the *SARSA(λ)* algorithm for finite state spaces. This means that both $|S| < \infty$ and $|\mathcal{A}(s)| < \infty,\ \forall s \in S$.

Extending to Continuous State Spaces

Going back to our original problem, we notice that the state $\vec{s} \in S$ of the agent (i.e. its position in the arena) is a continuous value instantiated from an infinite (and uncountable) state space $S \subseteq \mathbb{R}^2$. Therefore, we need to generalize our approach to continuous state spaces by approximating the action-values with a function that is linear w.r.t. to a weight vector $\vec{w}_a \in \mathbb{R}^K$ while the states are encoded by using a non-linear feature vector $\vec{\phi}(\vec{s}) \in \mathbb{R}^K$, i.e. $Q(\vec{s},a) = \vec{w}_a^T \cdot \vec{\phi}(\vec{s})$. Since the set of actions is still finite, we can keep a finite number of weight vectors $\vec{w}_a$ – one for each action. This is a supervised learning problem which we are essentially going to solve by using a simple two-layer neural network with K input units and an output unit for each action. The training of each output unit along with its corresponding weight vector is done in a Gradient Descent (GD) fashion.

If our goal was to approximate the single-step SARSA action-values, as given by (3), we would want to minimize the squared TD error by using gradient descent to find optimal weights. The gradient descent weight update rule would then be defined as $\vec{w}_a \longleftarrow \vec{w}_a - \frac{\eta}{2\partial\vec{w}_a}\delta_t^2 = \vec{w}_a + \eta\delta_t \frac{Q(\vec{s}_t,a_t)}{\partial\vec{w}_a} = \vec{w}_a + \eta\delta_t\vec{\phi}(\vec{s})$. In a similar fashion as before, we can augment this learning rule with eligibility traces. Our weight update rule then becomes $\vec{w}_a \longleftarrow \vec{w}_a + \eta\delta_t\vec{e}_a$. The eligibility trace is now also a K-dimensional vector with update rule $\vec{e}_a \longleftarrow \gamma\lambda\vec{e}_a + \vec{\phi}(\vec{s})$.

We spread an $N \times N$ lattice of uniformly spaced prototypes c_k (one for each input neuron) onto the unit square area of our arena. Each of the $K = N^2$ entries of our feature vector $\vec{\phi}(\vec{s}) = [\phi_k(\vec{s})]_{K\times 1}$ is a Gaussian radial basis functions that provides a distance measure between the state of the agent $\vec{s}$ and each of the prototypes $\vec{c}_k,\ (k = 1 \dots K)$.

For controlling the agent, we employ the ϵ-greedy strategy, which means: with probability ϵ, we choose an action by uniformly sampling the set of available actions; and otherwise, we choose an action according to (1). The parameter ϵ controls the tradeoff between exploration, which facilitates the acquisition of new knowledge, and exploitation, which focuses on utilizing the existing knowledge.

The single-episode SARSA learning and control method modified for continuous state spaces and applied to our water-maze problem is presented in *Algorithm 1*.

Note that during the course of the algorithm we use the standard TD error expression defined in (2), while only for the final step we use a different notion where we disregard the Q-value of the next step. According to the *unified notion for episodic and continuing tasks* described in [1], the returns that follow the terminal state are always zero, and so the Q-value for any action taken at the terminal state must also be zero. This represents an abrupt discontinuity in the action-value function as it is surely non-zero as we approach the goal area where the agent gets rewarded. Since our approximation of $Q^*(s,a)$ is smooth, we need to model this discontinuity by explicitly setting $Q(s_{T+1}, a_{T+1}) \longleftarrow 0$ where T is the time when the goal is reached. Had we missed this part, we would have ended up with an unstable estimator with perpetual inflation of Q-values.

Algorithm 1: Perform Single SARSA Trial

Input: $\vec{s}_{t_0}, l, D, \vec{g}_c, g_r, R_g, R_w, \eta, \gamma, \lambda, N, \epsilon$

$c_{1\dots K} \longleftarrow N \times N$ equispaced lattice over unit square

$A \longleftarrow \{1 \dots D\}$

load $\vec{w}_a,\ \forall a \in A$ ▷ *preinitialized to small random values*

$\vec{e}_a \longleftarrow [0]_{K\times 1}, \qquad \forall a \in A$

$\vec{s}_t \longleftarrow \vec{s}_{t_0}, \qquad a_{t+1} \longleftarrow \underset{a\in A}{\operatorname{argmax}}\, \vec{w}_a^T \cdot \vec{\phi}(\vec{s}_t)$

while $\|\vec{s}_t - \vec{g}_c\|_2 > g_r$ **do**

$a_t \longleftarrow a_{t+1}$

$\vec{s}_{t+1} \longleftarrow \vec{s}_t + l \cdot [\cos(2\pi a_t/D) \quad \sin(2\pi a_t/D)]^T$

if $\vec{s}_{t+1} \notin [0,1]^2$ **then** $r_{t+1} \longleftarrow R_w$
else if $\|\vec{s}_t - \vec{g}_c\|_2 < g_r$ **then** $r_{t+1} \longleftarrow R_g$
else $r_{t+1} \longleftarrow 0$

$$a_{t+1} \longleftarrow \begin{cases} Unif(A), & \text{w.p. } \epsilon \\ \underset{a\in A}{\operatorname{argmax}}\, \vec{w}_a^T \cdot \vec{\phi}(\vec{s}_{t+1}), & \text{w.p. } (1-\epsilon) \end{cases}$$

$\vec{e}_a \longleftarrow \gamma\lambda\vec{e}_a, \quad \forall a \in A$

$\vec{e}_{a_t} \longleftarrow \vec{e}_{a_t} + \vec{\phi}(\vec{s}_t)$

if $\|\vec{s}_t - \vec{g}_c\|_2 < g_r$ **then** $\delta_t \longleftarrow r_{t+1} - \vec{w}_{a_t}^T \cdot \vec{\phi}(\vec{s}_t)$
else $\delta_t \longleftarrow r_{t+1} + \gamma \cdot \vec{w}_{a_{t+1}}^T \cdot \vec{\phi}(\vec{s}_{t+1}) - \vec{w}_{a_t}^T \cdot \vec{\phi}(\vec{s}_t)$

$\vec{w}_a \longleftarrow \vec{w}_a + \eta\delta_t\vec{e}_a, \quad \forall a \in A$

output $\vec{s}_t$ ▷ *return steps to build path*

$\vec{s}_t \longleftarrow \vec{s}_{t+1}$

end while

save $\vec{w}_a,\ \forall a \in A$ ▷ *for usage in subsequent trials*

Results

We implement *Algorithm 1* in MATLAB and run it repeatedly in order to study the features of the SARSA algorithm. The ***default*** set of input parameters is the following: $\vec{s}_{t_0} = [0 \quad 0]^T$, $l = 0.03$, $D = 8$, $\vec{g}_c = [0.8 \quad 0.8]^T$, $g_r = 0.1$, $R_g = 10$, $R_w = -2$, $\eta = 0.005$, $\gamma = 0.95$, $\lambda = 0.95$, $N = 20$ and $\epsilon = 0.5$. In some of the results presented, when explicitly stated, some of the parameters get changed, while others are kept at their defaults.

For all of the results presented here, we set the parameters according to the specific test we want to perform, we initialize the weights and run 100 *trials* of *Algorithm 1*. This makes a single *experiment*. After that we reset the weights and re-run the experiment 50 times in order to obtain a descent mean performance metric. To prevent infinite loops, if the agent doesn't find the goal area in 10000 steps, we abort the trial.

Firstly, as a baseline, we try with all the default parameter values. To examine the performance of the algorithm, we plot two graphs: the *latency curve* (Figure 1) and the *integrated reward curve* (Figure 2).

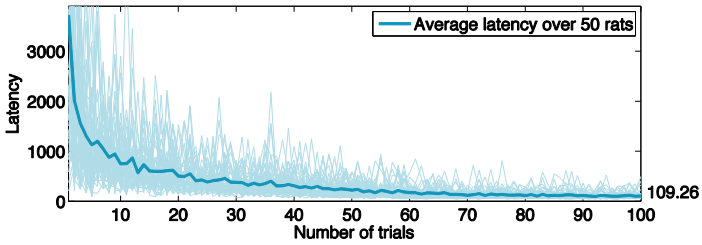

Figure 1 – Latency curve for the baseline experiment.

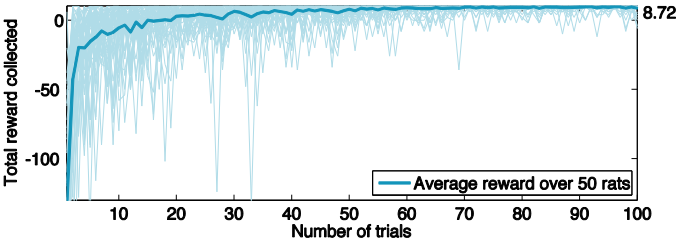

Figure 2 – Integrated reward curve for the baseline experiment.

The latency curve measures how the *latency* (i.e. number of steps taken in a single trial to reach the goal) evolves over the course of the experiment. The expectation is that in the beginning, since the agent doesn't have any knowledge about the environment, it will need a large number of steps to reach a goal as it is essentially performing a random walk. However, in subsequent trials, if the learning process was effective, the agent should be able to find the goal area more quickly. We can confirm this by looking at Figure 1. We see that just as the mean latency drops over the course of an experiment, so does the variance of the latency across different experiments. This supports the fact that in the beginning of an experiment, the movement of the agent is more random, while as time goes by it becomes more guided and predictable and the only random steps are exploratory ones. We can also see all the mentioned phenomena in Figure 3, where we plot the trajectories of the agent in the beginning of the experiment and towards its end.

The integrated reward is a measure of the total reward collected in a single trial. We reward the agent only when it reaches the goal, and penalize it only when it crosses the boundary. Therefore, we should expect that in the beginning the agent will receive a lot of penalty until it learns that it should avoid the boundaries. We clearly see this effect by looking at Figure 2.

When comparing Figure 1 and Figure 2, we notice that they both reflect the average amount of *knowledge* an agent has about the environment over the course of an experiment. They both start off with high absolute values and they begin to stabilize around the same time. Although both measures are able to capture the convergence of knowledge, the latency has more resolution as it measures the agent's effectiveness in applying that knowledge for finding optimal paths. For this reason, in the following experiments, we will continue to use only the latency as a measure of performance.

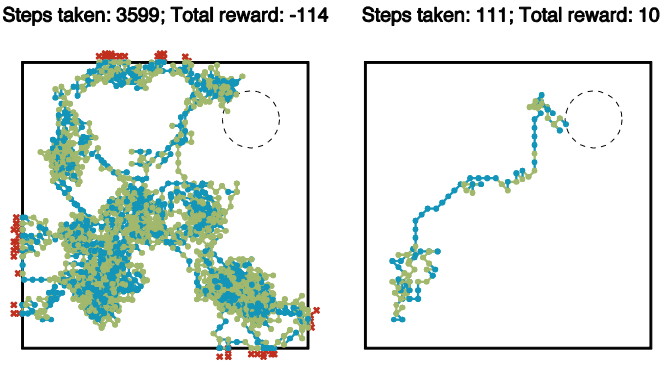

Figure 3 – Typical trajectory of the agent on the first trial (left) and after convergence (right). The starting point of the agent is in the bottom-left corner of the square arena and the circular goal area in the top-right corner is indicated with a dashed line. Blue dots indicate greedy steps that were taken based on the action-values and green dots indicate exploratory random steps. Red X's are points where the agent has crossed the boundary.

Even though our main goal here is pathfinding, the means to reaching that goal is estimation of the optimal action-value function $Q^*(s,a)$. In our context, this function is defined as $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ $\left(\forall s \in S, \forall a \in \mathcal{A}(s)\right)$ where π is a policy of choosing an action a from the set of available actions, given the state s. Furthermore, we have $Q^\pi(s,a) = \mathbb{E}(R_t | s_t = s, a_t = a)$ and $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$. In other words, $Q^*(s,a)$ is nothing other than the maximum expected discounted return obtained if the agent is in state s and the first subsequent action it performs is a.

For our simple arena, the only way for the agent to be rewarded is to reach the goal area. So, the maximal discounted return is obtained by following the shortest path from any given state to the goal area. If the minimal number of steps from $\vec{s}_t$ to the goal area is d, the optimal action-value would be $Q^*(\vec{s}_t, a^*) = R_t = \gamma^d \cdot R_g$ where a^* is the optimal action with the highest Q-value among all $a \in \mathcal{A}(\vec{s}_t)$. Thus, we expect that the optimal action-value function is exponentially decaying as we move further away from the goal area.

Using this notion, in Figure 4 we can see roughly how good are the estimates of the optimal action-value function after a certain number of trials. We also see the distribution of resulting optimal actions (i.e. directions of movement). Since we initialize the weights to small random values, we see that the resulting Q-values are small and the directions are pretty much random. As we move on with the experiment, we see the estimates slowly taking shape and exhibiting the mentioned exponential decay. We also see how the arrows

become more organized, and the number of those which are pointing towards the goal area increases.

It is clear, given that our algorithm is learning from experience, the best Q-value estimates are obtained in the most frequently visited areas. This is the reason why the quality of the estimates in the area behind the circle (far upper-right) has considerably poorer estimates than the rest of the arena.

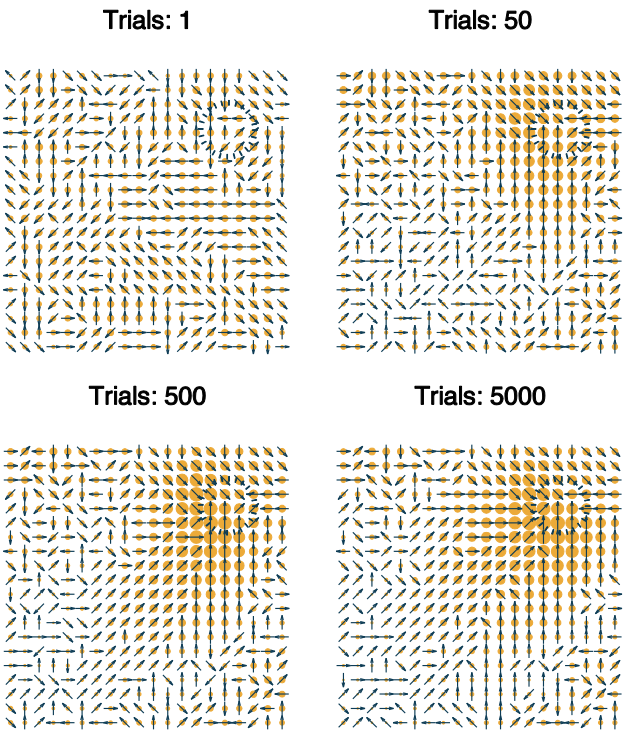

Figure 4 – Spatial distribution of estimated action-values and resulting preferred directions of movement. For the position of each prototype (neuron) $\vec{c}_k$ we draw an arrow in the direction that would have been chosen greedily $a^* = argmax_{a \in \mathcal{A}(\vec{c}_k)} Q(\vec{c}_k, a)$. Underneath each arrow is a circle with radius proportional to the corresponding action value $Q(\vec{c}_k, a^*)$.

We examine the exploration-exploitation tradeoff in Figure 5. Note that, since we initialize our weights to small random values, the beginning of the experiment is characterized by chaotic movements which resemble a random walk. In the general case, this means even the greedy movements in the beginning of the experiment are implicitly exploratory. However, since we don't penalize the agent on every step, we could imagine that in some cases, the distribution of Q-values perpetually drives the agent in circles and prevents it from ever reaching the goal. This is exactly what causes the latency jump when $\epsilon = 0$.

On the other hand, since the rewards in our environment are deterministic and constant over time, the exploration is only needed in the beginning. Later it just unnecessarily deters the agent from its optimal path and hurts performance. Thus, the movement of the agent just needs to be "random enough" in the beginning of the experiment to enable it to reach the goal most of the time and it will eventually learn the optimal path. After that, it will benefit from the least possible amount of exploration. We can see this in Figure 5 where the algorithm performs better (and more reliably) for smaller, non-zero values of ϵ.

Interestingly, if we pay attention to the arrows in Figure 4, we notice that already around the 50th trial, the ones which lie around the most frequently visited areas actually guide the agent in a direction which is fairly close to the optimal path. This tells us that for good pathfinding, we don't necessarily need to know the correct Q-values. Instead, their relative values just need to be good enough to be able to make the agent choose a nearly-optimal action. We expect that over a long course of actions taken, the resulting path would be near-optimal. The mentioned notion contributes to the robustness of this approach and explains why we were able to obtain low average latency even after 50 trials.

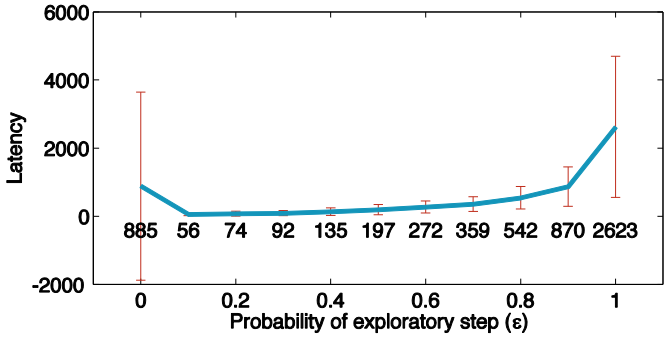

Figure 5 –Exploration-exploitation tradeoff. We change the ϵ parameter and run 50 experiments, each with 100 trials. Here we plot the mean and standard deviation of the 20 last trials from all 50 experiments combined.

We take advantage of all the mentioned observations by introducing ϵ-decay. After each trial, we multiply the ϵ parameter with a constant $c \in [0, 1]$ that controls the rate of decay of the ϵ parameter. This is a simple way to ensure there is enough exploration in the beginning of the experiment, while completely suppressing it in later stages as $\epsilon \to 0$. We plot the latency curve of the algorithm augmented with ϵ-decay in Figure 6. When comparing it to Figure 1, we see that not only is the mean latency dramatically reduced upon convergence, but also its variance as we completely suppress random steps. This seems to be a good learning strategy, given that the environment doesn't change over time.

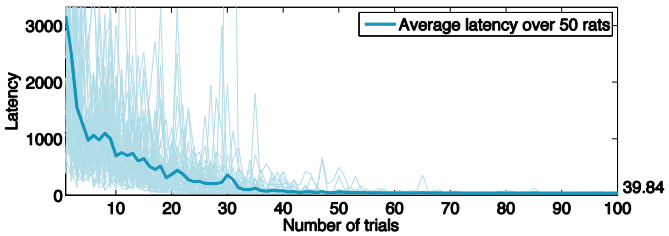

Figure 6 – Latency curve of a decaying ϵ value. We initialize ϵ to the default value, but after each trial we multiply it by $c = 0.9$.

[1] S. R. Sutton and G. A. Barto, Reinforcement Learning - An Introduction, Cambridge, Massachusetts: The MIT Press, 1998.