Lab 2 SARSA

Eduardo Santoro Morgan
PUC-Rio
ELE2761 - Reinforcement Learning
Prof. Wouter Caarls

April 22, 2019

1 Understanding the code

1.1

Each trial executes simtime/simstep simulations. For simtime = 3 and simstep = 0.03, the total number of simulations is 100.

1.2

```
Sanity checking sarsa.m
Error using ex2_test (line 9)
Random action rate out of bounds, check get_parameters/par.epsilon
```

This error message indicates that there is some error in the SARSA algorithm where the parameter ϵ is getting out-of-bounds, it should be $0 < \epsilon < 1$.

1.3

The function *get_parameters* is setting ϵ to 0. This parameter is the exploration parameter, which gives a probability that the policy will take a random action given the current state, as opposed to picking the best valued action from the state-action function. If SARSA executes using $\epsilon = 0$ it will only learn the optimal policy given the starting action-value pairs and, since there

is no guarantee it will explore all state-action pairs, there might be better better ones not explored.

2 Setting the learning parameters

2.1

The learning rate determines to what extend newly acquired information overrides current existing information, i.e., how much weight we give to the next state-action pair given the current taken action. If we set α to 0, the agent doesn't learn anything, whilst setting it to 1, it will ignore previously learned information and the learning process will not happen smoothly since, at each step, the value for the current state-action pair is going to be exactly the current reward plus the future state-action pair value, ignoring what it had already learned for the current state-action, thus, not carrying the information it had already learned from previous visits to such state-action pair.

2.2

$$2\pi x = \frac{\pi}{15} \implies x = 30 \tag{1}$$

We have 30 intervals, so 31 discretized states for the position.

2.3

$$24\pi x = \frac{4\pi}{5} \implies x = 30 \tag{2}$$

We have 30 intervals, so 31 discretized states for the velocity.

2.4

We have 3 possible actions and 31x31 possible states (31 for the position and 31 for the velocity), therefore we have 2883 state-action pairs.

The algorithm performs one update per simulation and we're running 100 simulations, so we're performing a total of $100 * 1000 = 100,000$ updates. Considering the state-action pairs are visited uniformly we perform, on average, $100,000/2883 \approx 35$ updates per pair. That is why we need to many trials, with fewer, some state-action pairs would probably never get visited!

3 Initialization

3.1

There are two most common ways to initialize the values for the state-action pairs: the realistic initialization and the optimistic initialization. The latter basically consists in initialization all state-action to high values so that all rewards it can achieve are lesser than the initialized ones and, therefore encouraging exploration. This approach is not well-suited to non-stationary problems or for problems that do not require a lot of exploration., Since the inverse pendulum problem has a very well-defined goal, the realistic method was used for the initialization, therefore all values were initialized to 0.1.

3.2

```
function init_Q()
    Q = 0.1 * ones(par.pos_states, par.vel_states, par.actions);
    Q(fix(par.pos_states/2), fix(par.vel_states/2), :) = 0;
end
```

4 Discretization

4.1

See 4.2.

4.2

```
function s = discretize_state(x)
    s = [];
    s(1) = discretize(x(1), -pi, pi, par.pos_states);
    s(2) = discretize(x(2), -12*pi, 12*pi, par.vel_states);
end
```

4.3

4.4

If we double the states' resolution the matrix Q is going to be 4 times larger.

If we had 10 dimension states, doubling the resolution would make the matrix Q 2^{10} times larger.

4.5

```
function u = take_action(a)
    voltages = linspace(-par.maxvoltage, par.maxvoltage, par.actions
    u = voltages(a);
end
```

Looking at Figure 1, the discretization seeems correct, since there are 31 states (30 intervals) for both position and velocity and they all look the same size and equally distributed along the y axis.

5

5.1

The simplest reward function can be 1 if the state is reached, i.e. $s' = [0, 0]$ and 0 otherwise.

5.2

```
function r = observe_reward(a, sP)
    if isequal(sP, discretize_state([0 0]))
        r = 1;
    else
        r = 0;
    end
end
```

5.3

If you want to minimize energy you could penalize (add a negative reward) proportional to the amount of voltage applied at each action.

The reward function is shown in the bottom left plot in Figure 1. We can see that all states' rewards are 0, except for the state in the center of the plot which corresponds to [0, 0].

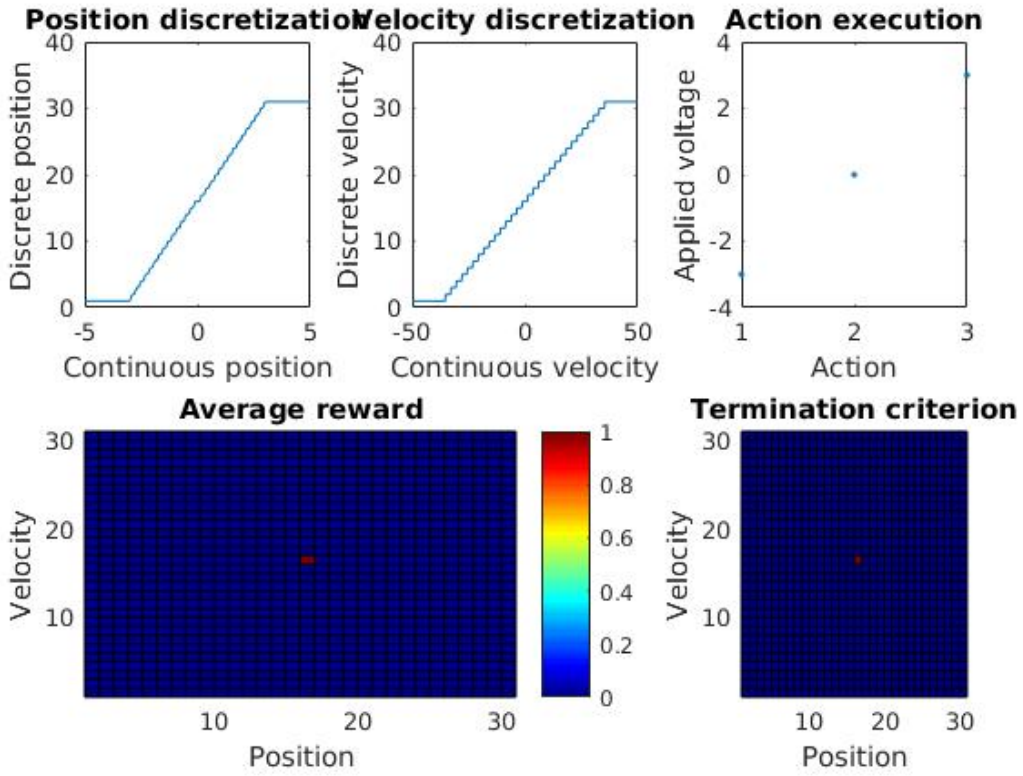

Figure 1: Sanity Check

5.4

```
function t = is_terminal(sP)
    t = isequal(sP, discretize_state([0 0]))
end
```

The bottom right plot in Figure 1 displays the termination criterion for the problem. We can see a hotter value around the center of the plot, exactly where the state [0, 0] should be.

6

6.1

```
function a = execute_policy(s)
```

```
    explore = rand(1) <= par.epsilon;
    if explore
        a = fix(rand(1) * par.actions) + 1;
    else
        max_val = max(Q(s(1), s(2), :));
        candidates = find(Q(s(1), s(2), :) == max_val)
        idx = randi(length(candidates));
        a = candidates(idx);
    end
end
```

6.2

```
function update_Q(s, a, r, sP, aP)
    current_value = Q(s(1), s(2), a);
    Q(s(1), s(2), a) =  current_value + par.alpha * ...
        (r + par.gamma * Q(sP(1), sP(2), aP) - current_value);
end
```

The plots displayed in Figure 1 all look as they should be, according to Figure 1 from the assignment description.

7

7.1

Figure 2 displays the results from running SARSA with the default parameters. We can see that it quickly converges to the desired state.

7.2

Lowering the discount rate causes the learning procedure to carry less weight to rewards which are further in the future in the computation of the expected return. The practical result we see in this problem is that the agent takes more time to learn the desired policy, from the learning curve in Figure 3 we can see that the learning process has not yet converged, as opposed to what happened in Figure 2.

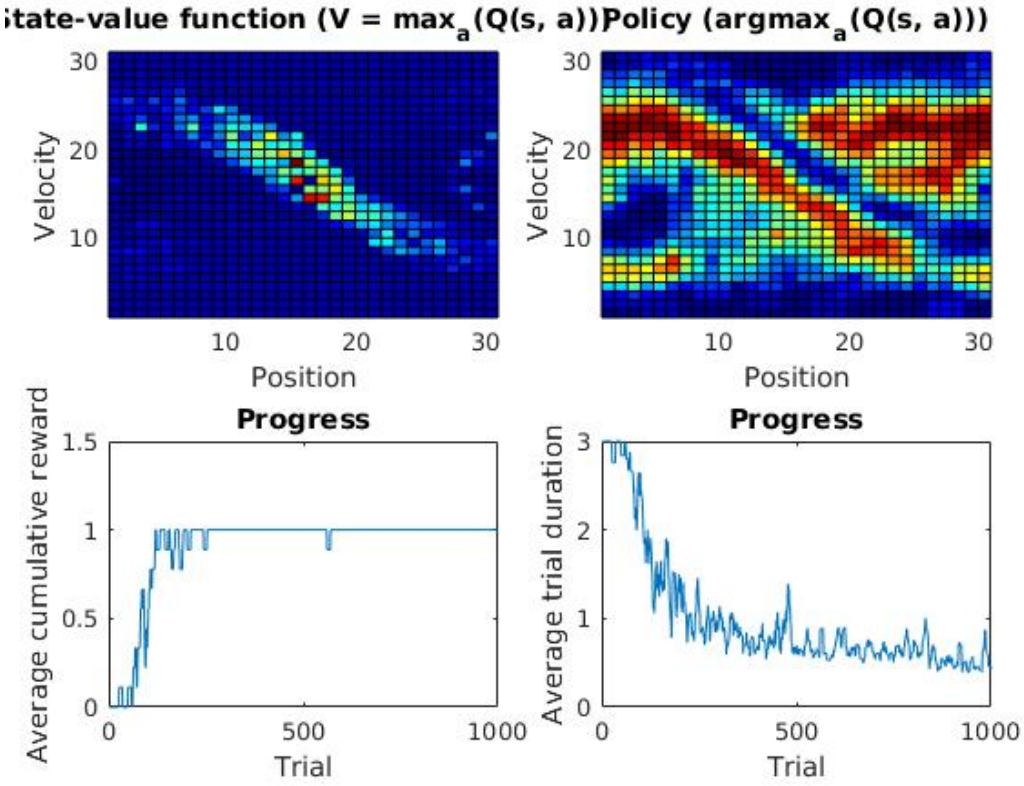

Figure 2: Result 7.1

7.3

Figure 4 displays such behavior. This happens since the SARSA update rule only updates one state-value pair at each iteration, therefore, in the case where there's no discount, the updated value at each iteration is spread almost equally across all state-action pairs that can be reached from the current one. At each iteration where a new pair is explored, its value will change and carried over to all other state-action pairs.

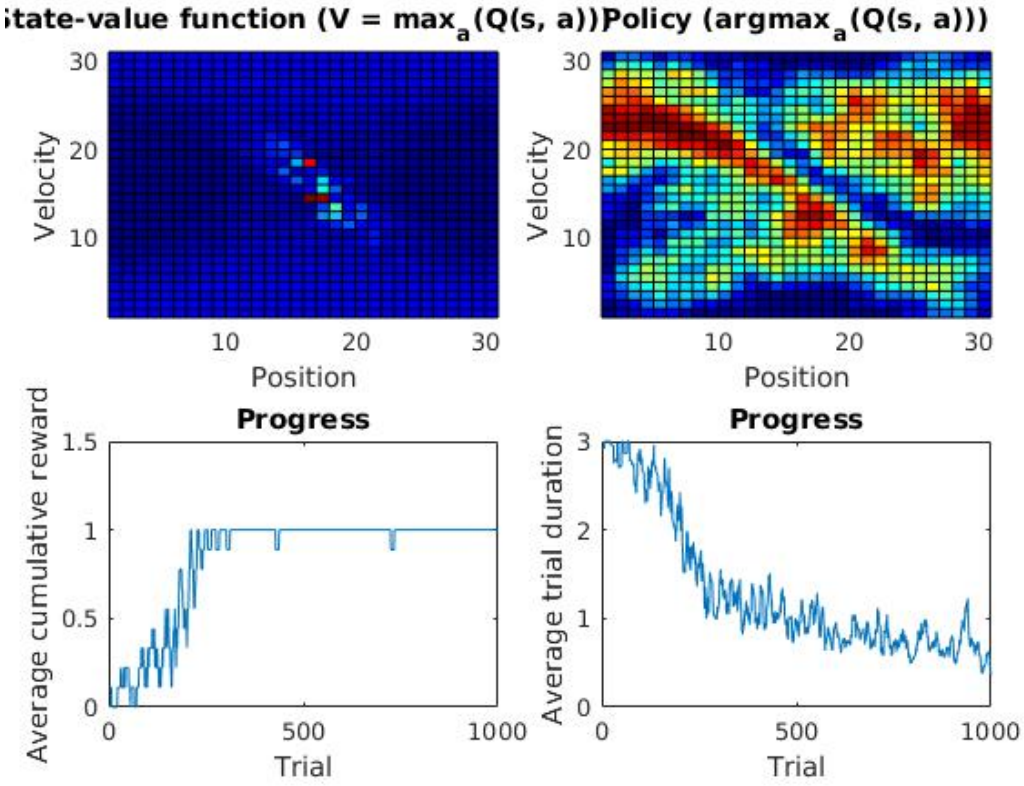

Figure 3: Result 7.2

8

8.1

8.2

```
function update_Q(s, a, r, sP, aP)
        e = par.gamma * par.lambda * e;
        e(s(1), s(2), a) =  1;
        Q = Q + par.alpha * ...
        (r + par.gamma * Q(sP(1), sP(2), aP) - Q(s(1), s(2), a)) * e;
    end
```

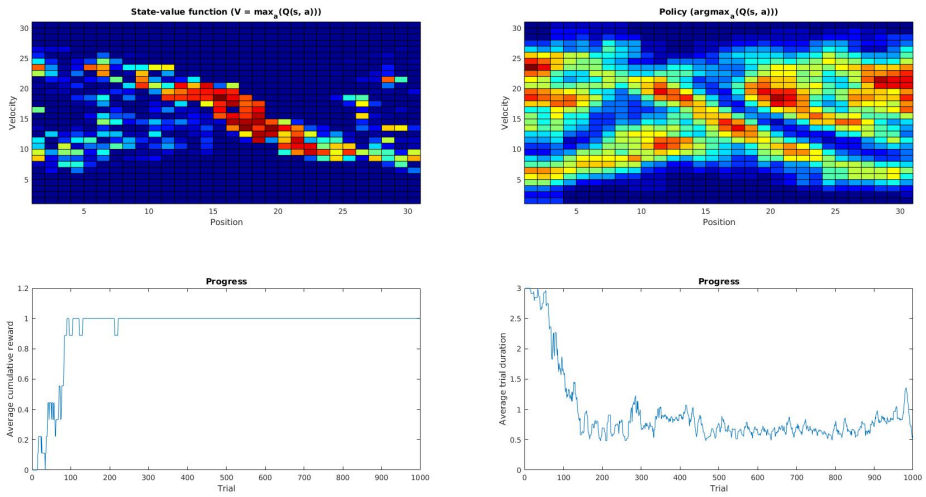

Figure 4: Result 7.3

Figure 5 shows the result obtained using the eligibility traces to make the new update rule. We can see that learning happened much faster since now many states are updated at each iteration.

The update rule was implemented using the eligibility replacement approach.

8.3

Figure 6 shows the algorithm running with the trace decay rate $\lambda = 0.9$. This parameter determines how eligible are previously visited states. Making the parameter higher makes it that previously visited states become almost as eligible as the current one, which, for this problem, makes the learning process more difficult, since it is giving too much weight to such states instead of the current one.

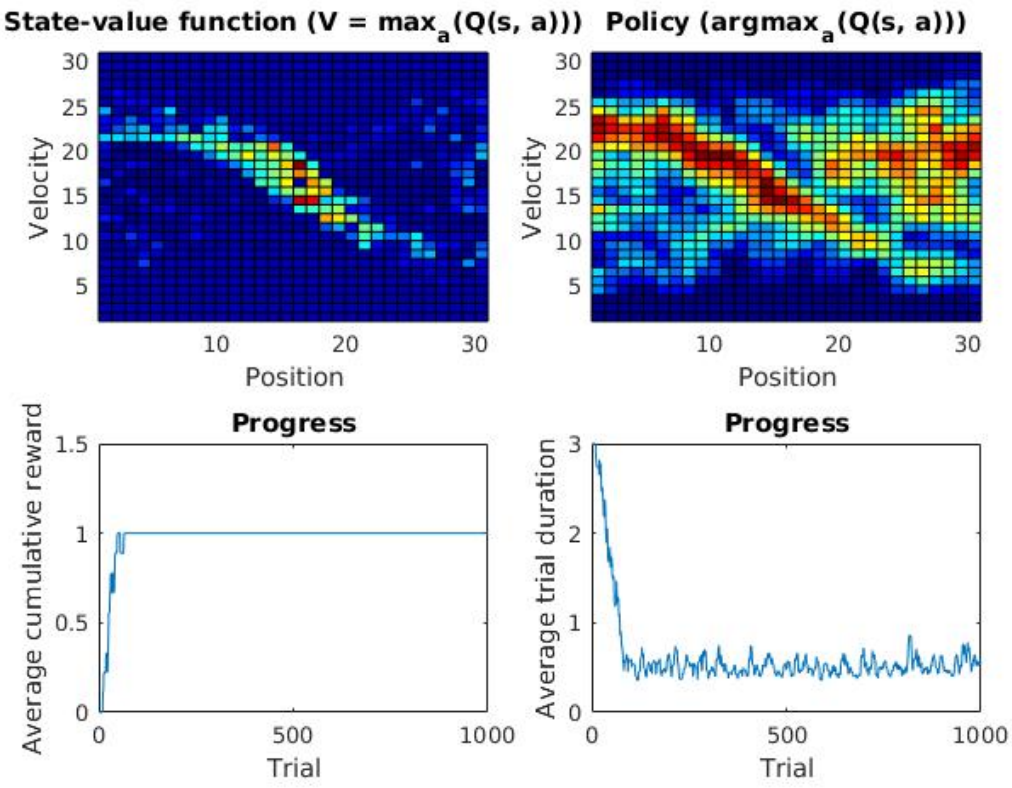

Figure 5: Result Eligibility $\lambda = 0.5$

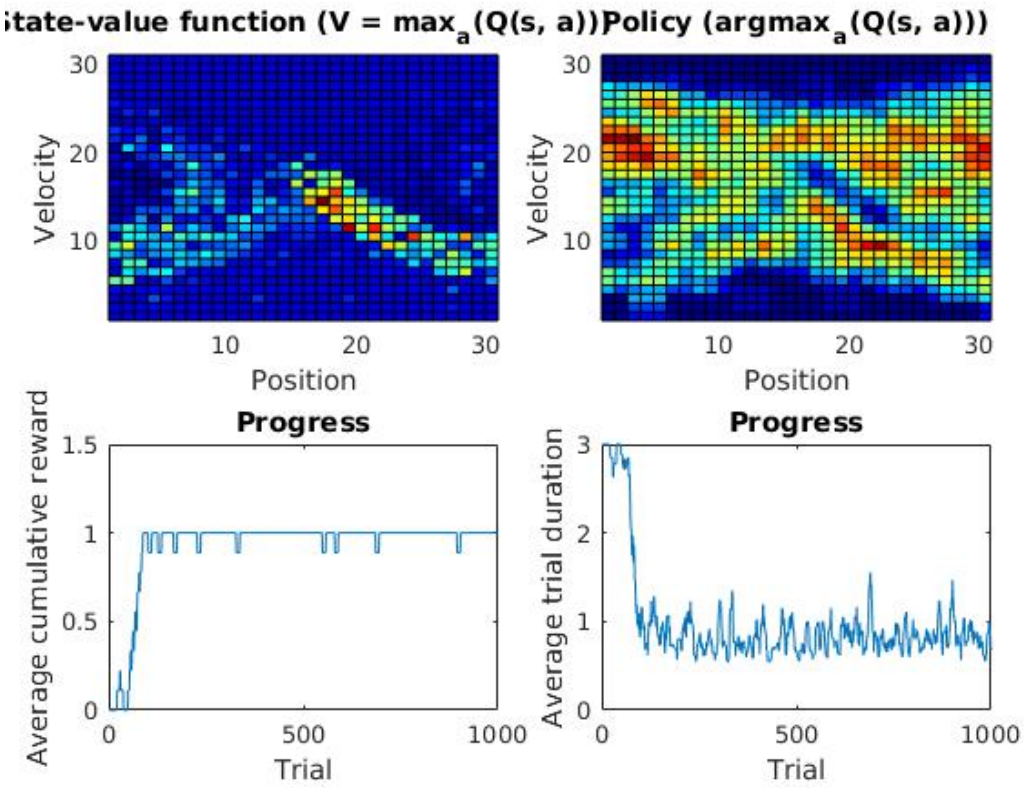

Figure 6: Result Eligibility $\lambda = 0.9$