Reinforcement Learning: An Introduction - Chapter 6

Thomas Hopkins

Exercise 6.1

Given the scenario offered in the book, moving to a new building for work but entering the highway at the same place on your way home gives a big advantage for TD methods compared to Monte Carlo methods. This is because with a large amount of experience driving home from the old building, the estimate for the time it takes to get home from the highway entrance is very accurate. This data is utilized by TD methods but *not* by Monte Carlo methods. This is one way in which boostrapping is powerful. It can rapidly adapt to changes when new states are encountered. This kind of thing happens even in the original scenario, coming home from the old building.

Exercise 6.2

This does not tell much about what happened in the first episode other than that the episode terminated on the left. The update to the value occured as

$$V(A) = 0.5 + 0.1(0.0 + (1.0)(0.0) - 0.5) = 0.5 - 0.05 = 0.45$$

This is because the value of terminal states are always 0, by definition.

Exercise 6.3

Given only 100 episodes, I do not think that changing the value of α would lower the RMS error. Given a larger, but still finite, number of episodes, a lower α would be better since it will take smaller steps. The problem with a limit of 100 episodes to learn from is that a smaller α may not converge by the time that limit is reached. One could get the best of both worlds by decreasing the learning rate as the number of episodes increases, however, finding the correct schedule for decreasing the learning rate is its own problem.

Exercise 6.4

As learning continues, this will always occur. The severity of this issue depends on the size of the learning rate, α. TD methods continue to update the states using this learning rate, so fluctuations can occur and errors can propagate. This could be an issue of how the approximate value function was initialized since it could have introduced bias into the function that would be difficult to unlearn. For instance, $V(A) = 0.5$ is quite far from its true value while $V(C) = 0.5$ is spot on.

Exercise 6.5

One way is by reasoning that $V(C) = 0.5$ and then we can compute the values of all of the other states by solving a system of linear equations. The other way is by using dynamic programming methods. Solving the linear system is probably easier.

The linear system would be

$$V(C) = \frac{1}{2}$$

$$V(A) = \frac{V(B)}{2}$$

$$V(B) = \frac{V(A) + V(C)}{2}$$

$$V(D) = \frac{V(C) + V(E)}{2}$$

$$V(E) = \frac{1 + V(D)}{2}$$

Exercise 6.6

Here is my implementation of the Windy Gridworld environment. As we can see, including diagonal moves allows us to reach the goal in only 8 steps compared to 16 steps without. Allowing for no movement as a 9th action, we also reach the goal in 8 steps but along a different path.

In [6]:

```
using Base.Iterators
using Random
using Distributions
using Plots
gr()

STATES = collect(product(1:7, 1:10))
ACTIONS = [(0, -1), (-1, 0), (0, 1), (1, 0)] # (left, up, right, down)
WIND = [0, 0, 0, 1, 1, 1, 2, 2, 1, 0]
START = (4, 1)
GOAL = (4, 8)


function select_action(Q_values, state; epsilon = 0.1)
    if rand() < epsilon
        return rand(ACTIONS)
    end
    return argmax(Q_values[state])
end

function step(state, action; stochastic_wind = false)
    new_state = [state[1] + action[1], state[2] + action[2]]
    wind = WIND[state[2]]
    if stochastic_wind
        r = rand()
        if r < 1 / 3
            wind -= 1
        elseif 1 / 3 <= r < 2 / 3
            wind += 1
        end
    end
    new_state[1] -= WIND[state[2]]
    if new_state[1] <= 0
        new_state[1] = 1
    elseif new_state[1] > 7
        new_state[1] = 7
    end
    if new_state[2] <= 0
        new_state[2] = 1
    elseif new_state[2] > 10
        new_state[2] = 10
    end
    new_state = (new_state[1], new_state[2])
    reward = -1.0
    terminated = false
    if new_state == GOAL
        reward = 0.0
        terminated = true
    end
    return reward, new_state, terminated
end

function sarsa!(Q_values; num_episodes = 1000, epsilon = 0.1, alpha = 0.1, 
    for e = 1:num_episodes
        state = START
        action = select_action(Q_values, state; epsilon = epsilon)
        terminated = false
        while !terminated
```

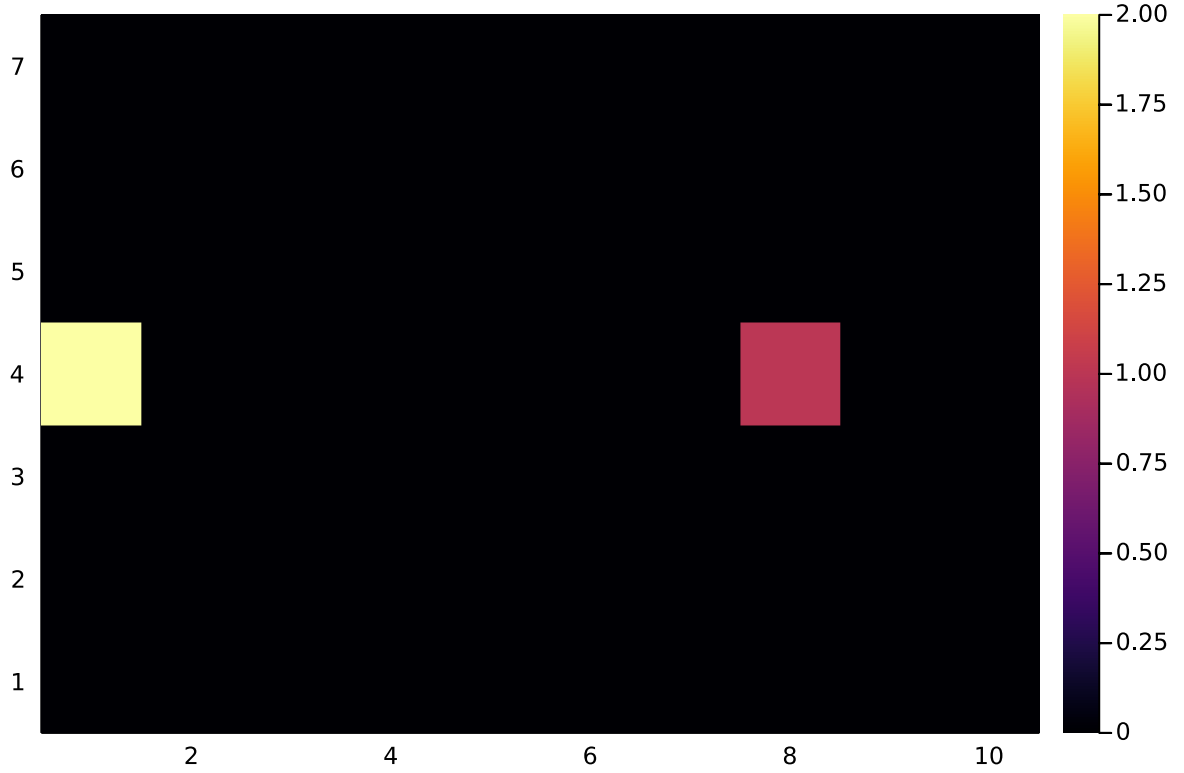
```
            reward, new_state, terminated = step(state, action; stochastic_w
            new_action = select_action(Q_values, new_state; epsilon = epsilo
            Q_values[state][action] += alpha * (reward + gamma * Q_values[ne
            state = new_state
            action = new_action
        end
    end
    return Q_values
end

function eval(Q_values; stochastic_wind = false)
    grid = zeros(Int, 7, 10)
    state = START
    action = select_action(Q_values, state; epsilon = 0.0)
    terminated = false
    while !terminated
        grid[state[1], state[2]] = 1
        reward, new_state, terminated = step(state, action; stochastic_wind
        new_action = select_action(Q_values, new_state; epsilon = 0.0)
        state = new_state
        action = new_action
    end
    grid[state[1], state[2]] = 1
    return heatmap(reverse(grid; dims=1))
end;
```

In [12]:

```
grid = zeros(Int, 7, 10)
grid[START[1], START[2]] = 2
grid[GOAL[1], GOAL[2]] = 1
heatmap(reverse(grid; dims=1))
```

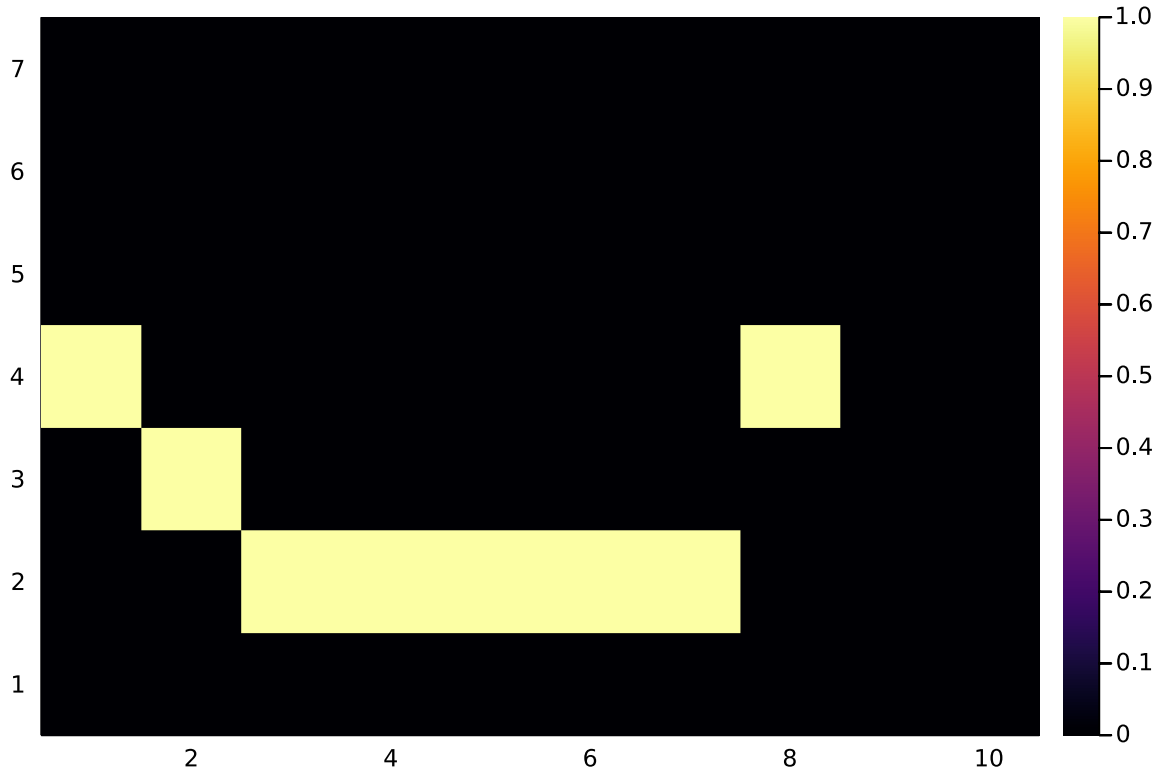
Out[12]:



In [7]:

```
Random.seed!(32)
ACTIONS = [(-1, -1), (-1, 0), (-1, 1),  # all cardinal directions
           (0, -1), (0, 1),
           (1, -1), (1, 0), (1, 1)]
Q_values = Dict(s => Dict(a => 0.0 for a in ACTIONS) for s in STATES)
Q_updated = sarsa!(Q_values)
eval(Q_updated)
```

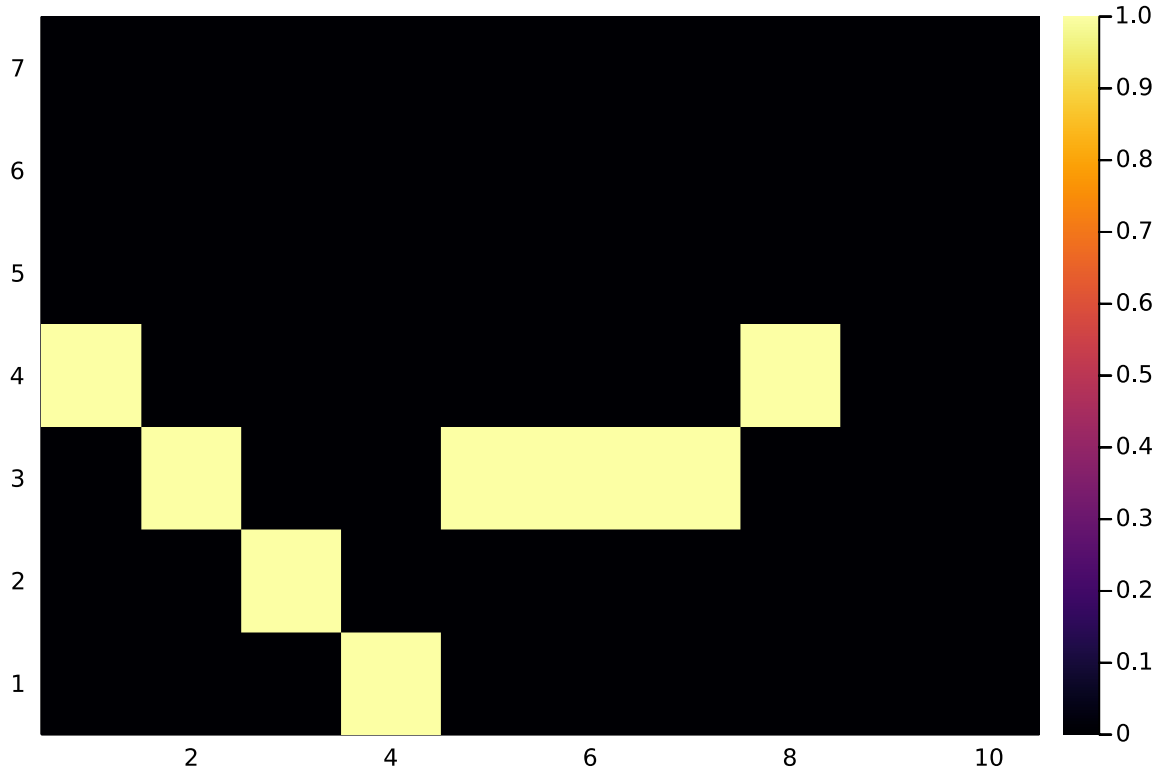
Out[7]:



In [8]:

```
Random.seed!(32)
ACTIONS = collect(product(-1:1, -1:1)) # all cardinal directions including
Q_values = Dict(s => Dict(a => 0.0 for a in ACTIONS) for s in STATES)
Q_updated = sarsa!(Q_values)
eval(Q_updated)
```

Out[8]:



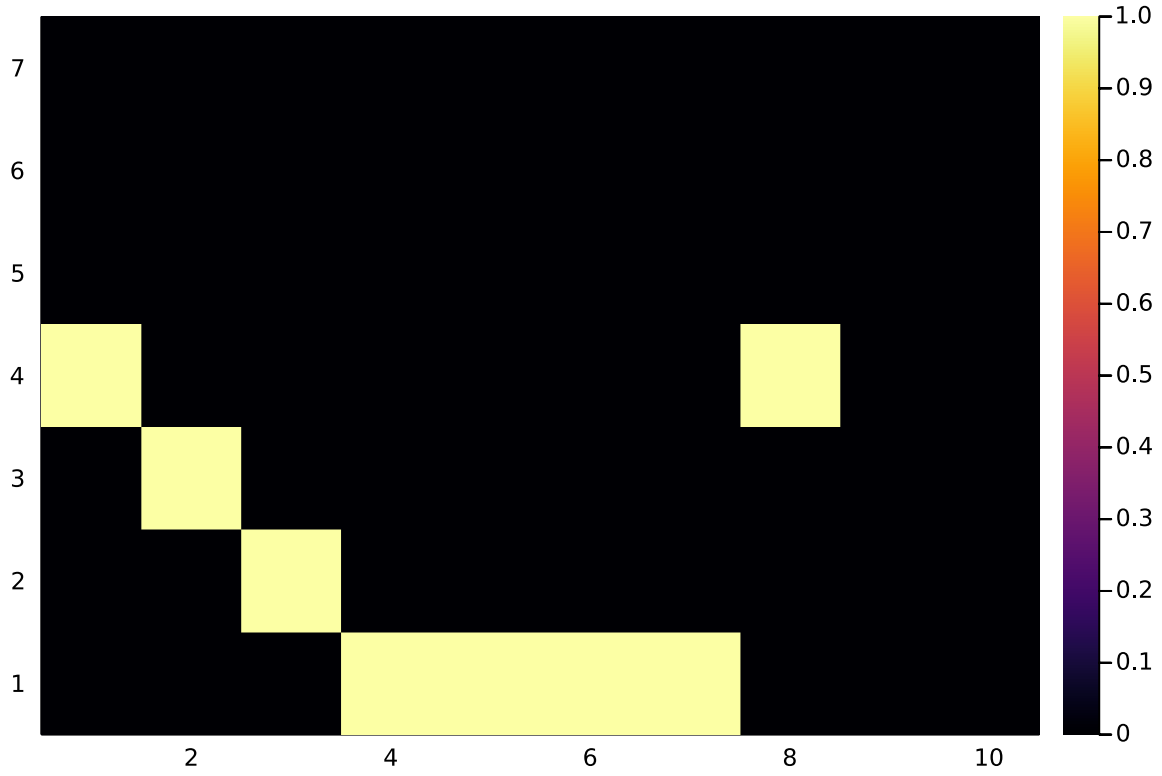
Exercise 6.7

For the stochastic wind case, the agent decides that going to the bottom fast is the best strategy.

In [9]:

```
Random.seed!(32)
ACTIONS = [(-1, -1), (-1, 0), (-1, 1),  # all cardinal directions
           (0, -1), (0, 1),
           (1, -1), (1, 0), (1, 1)]
Q_values = Dict(s => Dict(a => 0.0 for a in ACTIONS) for s in STATES)
Q_updated = sarsa!(Q_values; stochastic_wind = true)
eval(Q_updated; stochastic_wind = true)
```

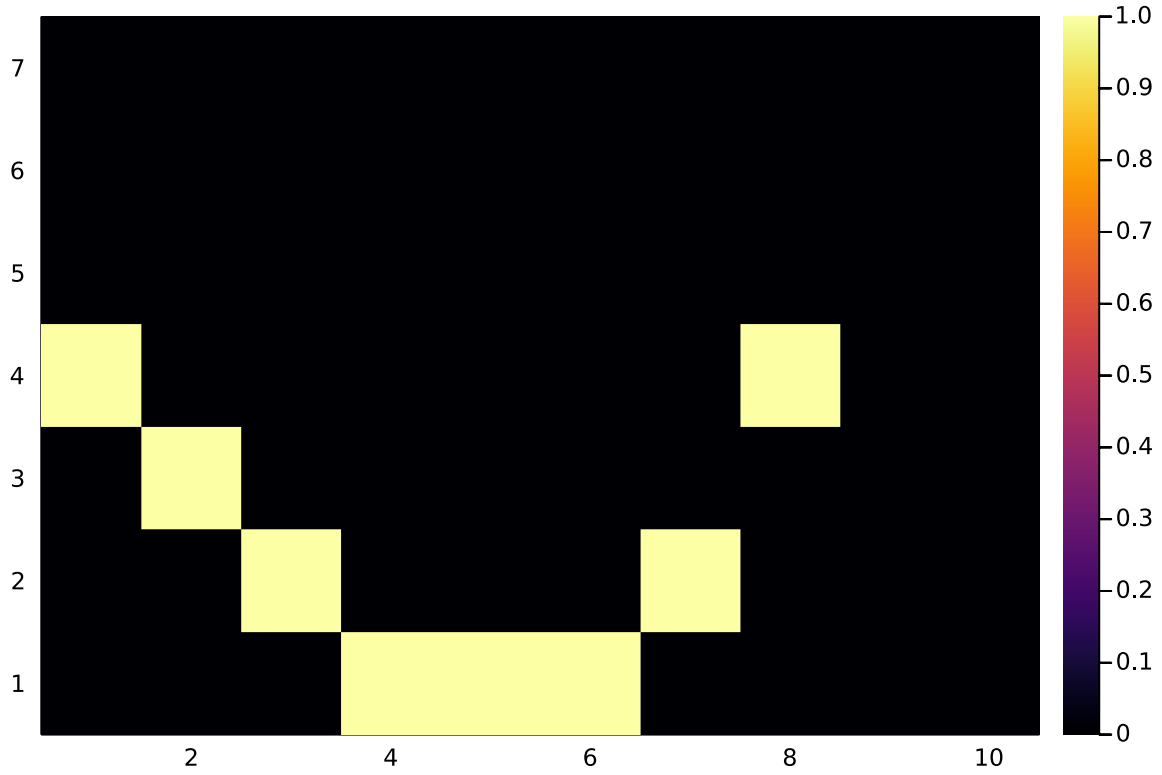
Out[9]:



In [10]:

```
Random.seed!(32)
ACTIONS = collect(product(-1:1, -1:1)) # all cardinal directions including i
Q_values = Dict(s => Dict(a => 0.0 for a in ACTIONS) for s in STATES)
Q_updated = sarsa!(Q_values; stochastic_wind = true)
eval(Q_updated; stochastic_wind = true)
```

Out[10]:



Exercise 6.8

The backup diagram for Sarsa is the same as TD(0) except that the state nodes are replaced with action nodes (white to black) and the central node changes from action to state (black to white).

Exercise 6.9

Q-learning is an *off-policy* control algorithm because it has a separate behavior policy and estimation policy.

Exercise 6.10

This new policy would be off-policy since the behavior and estimation policy are different. This algorithm is very similar to dynamic programming since it requires a model of the environment for $\pi(s, a)$. The backup diagram is similar to the one for Q-learning except a weighted sum is taken over possible next actions instead of a `max` operator. I would expect this to work better than Sarsa since it is working with a model of the environment. This consideration certainly impacts the ability to compare the two fairly.

Exercise 6.11

Simply add an action selection line and replace the update rules in Figure 6.16 as follows:

Choose action a' in s' using policy derived from Q (ϵ-greedy)

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r - \rho + \max_{a'} Q(s',a') - Q(s,a)] \Rightarrow Q(s,a) \leftarrow Q(s,a) + \alpha[r - \rho + ($$

$$\rho \leftarrow \rho + \beta[r - \rho + \max_{a'} Q(s',a') - \max_{a} Q(s,a)] \Rightarrow \rho \leftarrow \rho + \beta[r - \rho + Q(s',a') - \max_{a} Q(s$$

Exercise 6.12

Jack's Car Rental can be reformulated in terms of afterstates since the effect of taking an action (moving cars from one lot to the other) is deterministic. It would be easier to learn the values of afterstates since these would correspond to the number of cars in each lot at the beginning of each day. The original problem represents the state as the number of cars at the end of each day. This would improve learning convergence since it reduces the number of states that yield the same result. Taking different actions at the end of the day for different states can result in the same afterstate.