Decision Making Under Uncertainty: Homework 4

Julian Lema Bulliard*
University of Colorado Boulder

I. Questions

A. Question 1

a) After a very large number of pulls and using $\epsilon = 0.15$ we will use the following equation to get the expected payoff per pull. In the following equation K is the number of arms and theta is the expected the payoff probabilities.

$$E = (1 - \epsilon) \max \theta_i + \frac{\epsilon}{K} \sum_{i=1}^{K} \theta_i$$

$$E = (1 - 0.15) * 0.7 + \frac{0.15}{3}(0.2 + 0.3 + 0.7) = 0.655$$

b) The probability of selecting an arm based on the softmax policy of $\lambda = 5$ and a precision factor of 1.0, can be found with the following equation. Note that K is the number of arms in the problem.

$$P(arm) = \frac{e^{\rho_a \lambda}}{\sum_{i=1}^{K} e^{\rho_i \lambda}}$$

From here we can plug in our given data and find the following probabilities for arm three.

$$P(K = 3) = 0.821$$

*Graduate Student, Aerospace Engineering Department

c) For the given wins and losses for each arm of w = [0,1,3], l = [1,0,2] we get the following plots.

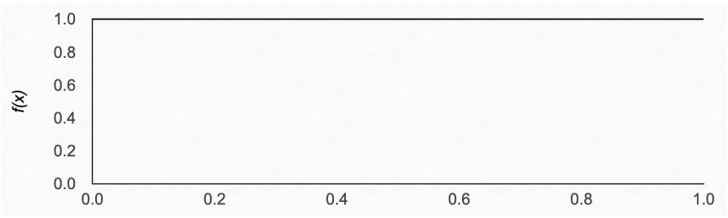

Fig. 1 Standard beta(1,1)

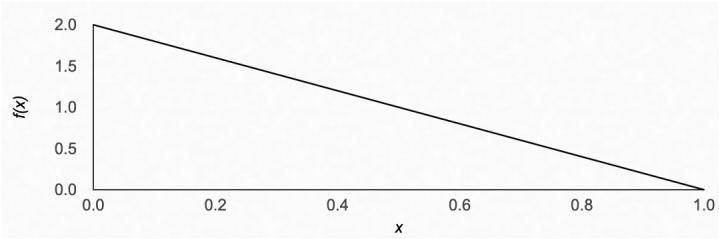

Fig. 2 Arm 1: beta(w + 1, l + 1) = beta(1,2)

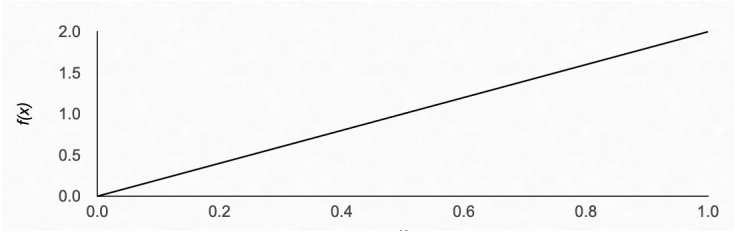

Fig. 3 Arm 2: beta(w + 1, l + 1) = beta(2,1)

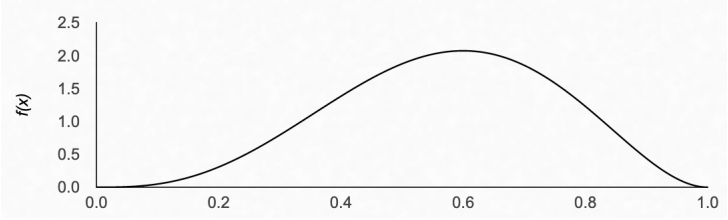

Fig. 4 Arm 3: beta(w + 1, l + 1) = beta(4,3)

d) For Thompson Sampling, we will take the above distributions for each of our arms and we will randomly sample a value for each of those distributions. If for example, we got that: $\theta_1 = 0.3$, $\theta_2 = 0.2$, $\theta_1 = 0.1$, then we would pull arm one, as this is the arm that had the highest sampled value.

II. Question 2

a) For the source code, please refer to the functions section at the bottom of the homework

b) See the two graphs below...

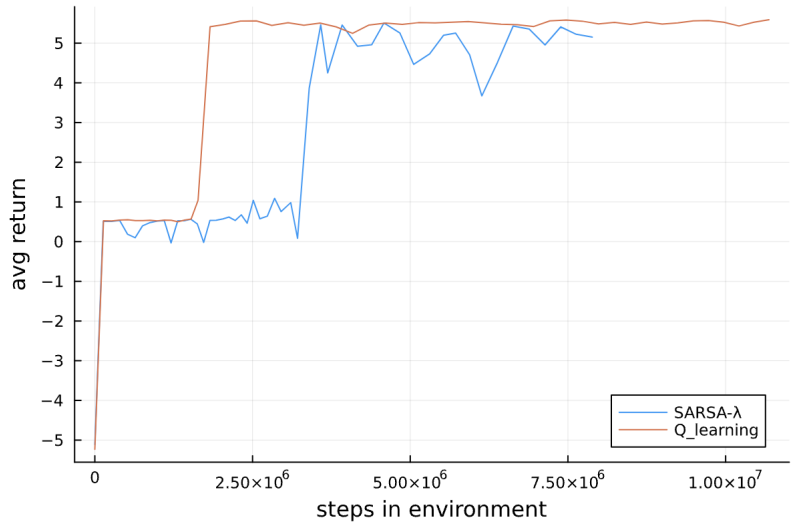

Fig. 5 Reward vs Step

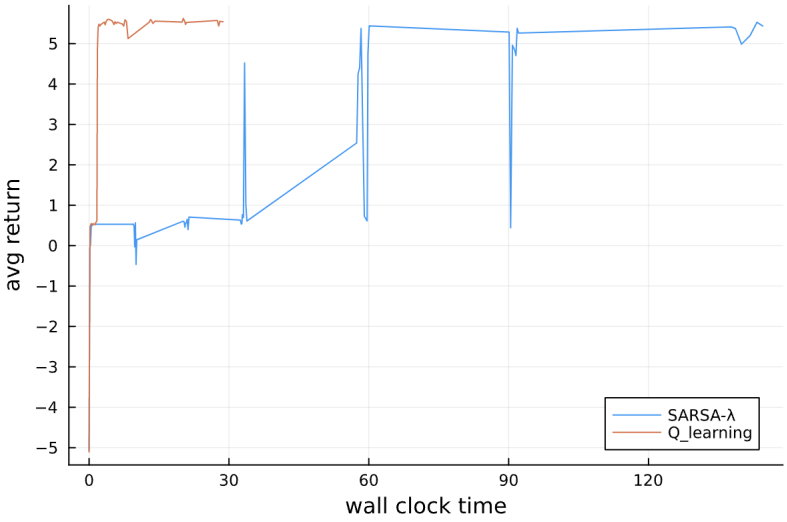

Fig. 6 Wall Clock vs Reward

c) From the two graphs above we can see a lot of differences between the two algorithms. Firstly, in general, most of my runs saw that Q Learning was significantly quicker in learning the policy than SARSA λ. We can see that not only was it quicker to learn the policy, but it was a much more efficient algorithm in learning quicker time-wise (wall clock time). In terms of sample complexity, it is evident that SARSA-λ has a much higher sample complexity as it has to interact with the environment many more time than Q-Learning to be able to

learn an optimal policy.

Apart from this, we should note that Q-Learning is an off-policy algorithm while SARSA-λ is an on-policy. In general, Q-Learning is a lot more computationally efficient (hence why DQN algorithms are used in industry and in video games). In our case, we did not really run into the following issue, but in general, SARSA-λ is a lot more stable when it comes to continuous spaces, as well as high-dimensional/complex environments, while Q-Learning may have some trouble learning in these scenarios, especially with noise.

The two ending states of the environments can also be seen below

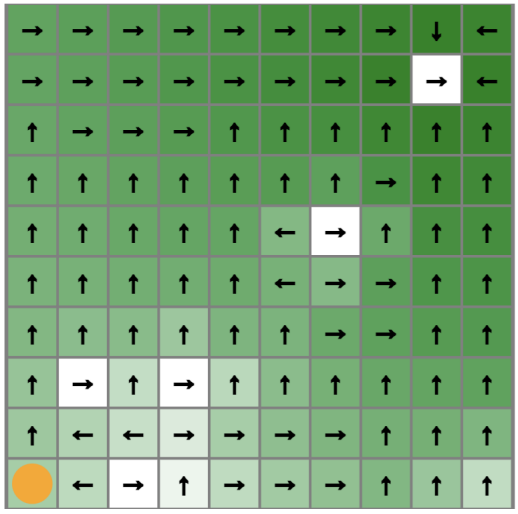

Fig. 7 Q-Learning

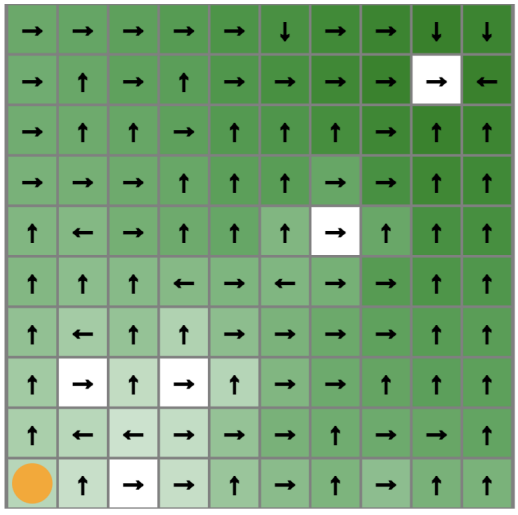

Fig. 8 SARSA λ

III. Code Appendix

A. Imports and Setting Up Environment

```
using DMUStudent
using DMUStudent.HW4: gw
# using POMDPs: actions, @gen, isterminal, discount, statetype, actiontype, simulate, states, initia
using StaticArrays: SA
using Statistics: mean, std, mean
using Random
using CommonRLInterface: render, actions, act!, observe, reset!, AbstractEnv, observations, termina
env = DMUStudent.HW4.gw
```

B. Q Learning Algorithm

```
function q_learning_episode!(Q, env; =0.10, =0.99, =0.05)

    start = time()

    function policy(s)
        if rand() <
            return rand(actions(env))
        else
            return argmax(a->Q[(s, a)], actions(env))
        end
    end

    s = observe(env)
    a = policy(s)
    r = act!(env, a)
    sp = observe(env)
    hist = [s]

    while !terminated(env)
        ap = policy(sp)
```

```
        a_opt = argmax(a->Q[(sp, a)], actions(env))
        Q[(s, a)] = Q[(s, a)] + *(r +  *Q[(sp,a_opt)] - Q[(s,a)])



        s = sp
        a = ap
        r = act!(env, a)
        sp = observe(env)
        push!(hist, sp)
    end


     a_opt = argmax(a->Q[(sp, a)], actions(env))
    Q[(s, a)] += *(r +  *(Q[(sp,a_opt)]) - Q[(s,a)])


    return (hist=hist, Q = copy(Q), time=time()-start)
end



function q_learning!(env; n_episodes=100, kwargs...)
    Q = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
    episodes = []

    for i in 1:n_episodes
        reset!(env)
        push!(episodes, q_learning_episode!(Q, env;
                                            =max(0.01, 1-i/n_episodes),
                                            kwargs...))
    end
```

```
    return episodes
end



q_episodes = q_learning!(env, n_episodes=500000, =0.01);



Q_temp = q_episodes[500000].Q
reset!(gw)
HW4.render(gw, color=s->maximum(map(a->Q_temp[(s, a)], actions(gw))), policy=s->argmax(a->Q_temp[(s, a)
```

C. SARSA λ Algorithm

```
function sarsa_lambda_episode!(Q, env; =0.10, =0.99, =0.05, =0.9)

    start = time()

    function policy(s)
        if rand() <
            return rand(actions(env))
        else
            return argmax(a->Q[(s, a)], actions(env))
        end
    end

    s = observe(env)
    a = policy(s)
    r = act!(env, a)
    sp = observe(env)
    hist = [s]
    N = Dict((s, a) => 0.0)
```

```
    while !terminated(env)
        ap = policy(sp)

        N[(s, a)] = get(N, (s, a), 0.0) + 1

         = r + *Q[(sp, ap)] - Q[(s, a)]

        for ((s, a), n) in N
            Q[(s, a)] += **n
            N[(s, a)] *= *
        end

        s = sp
        a = ap
        r = act!(env, a)
        sp = observe(env)
        push!(hist, sp)
    end

    N[(s, a)] = get(N, (s, a), 0.0) + 1
     = r - Q[(s, a)]

    for ((s, a), n) in N
        Q[(s, a)] += **n
        N[(s, a)] *= *
    end

    return (hist=hist, Q = copy(Q), time=time()-start)
end



function sarsa_lambda!(env; n_episodes=10000, kwargs...)
```

```
    Q = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
    episodes = []

    for i in 1:n_episodes
        reset!(env)
        push!(episodes, sarsa_lambda_episode!(Q, env;
                                              =max(0.01, 1-i/n_episodes),
                                              kwargs...))
    end

    return episodes
end



lambda_episodes = sarsa_lambda!(env, n_episodes= 500000, =0.01, =0.99);



Q_temp = lambda_episodes[500000].Q
reset!(gw)
HW4.render(gw, color=s->maximum(map(a->Q_temp[(s, a)], actions(gw))), policy=s->argmax(a->Q_temp[(s, a)
```

D. Plotting

```
    using Plots

function evaluate(env, policy, n_episodes=10000, max_steps=1000, =1.0)
    returns = Float64[]
    for _ in 1:n_episodes
        t = 0
        r = 0.0
```

```
        reset!(env)
        s = observe(env)
        while !terminated(env)
            a = policy(s)
            r += ^t*act!(env, a)
            s = observe(env)
            t += 1
        end
        push!(returns, r)
    end
    return returns
end

episodes = Dict("Q_learning"=>q_episodes, "SARSA-"=>lambda_episodes)



p = plot(xlabel="steps in environment", ylabel="avg return")
n = 10000
stop = 500000
for (name, eps) in episodes
    Q = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
    xs = [0]
    ys = [mean(evaluate(env, s->argmax(a->Q[(s, a)], actions(env))))]
    for i in n:n:min(stop, length(eps))
        newsteps = sum(length(ep.hist) for ep in eps[i-n+1:i])
        push!(xs, last(xs) + newsteps)
        Q = eps[i].Q
        push!(ys, mean(evaluate(env, s->argmax(a->Q[(s, a)], actions(env)))))
    end
    plot!(p, xs, ys, label=name)
    yticks!(p, -5:1:maximum(ys))
end
```

```
# Set the axis ticks to appear by ones
# xticks!(p, 0:1:last(xs))



p
```

```
p = plot(xlabel="wall clock time", ylabel="avg return")
n = 50000
stop = 500000
for (name,eps) in episodes
    Q = Dict((s, a) => 0.0 for s in observations(env), a in actions(env))
    xs = [0.0]
    ys = [mean(evaluate(env, s->argmax(a->Q[(s, a)], actions(env))))]
    for i in n:n:min(stop, length(eps))
        newtime = sum(ep.time for ep in eps[i-n+1:i])
        push!(xs, last(xs) + newtime)
        Q = eps[i].Q
        push!(ys, mean(evaluate(env, s->argmax(a->Q[(s, a)], actions(env)))))
    end
    plot!(p, xs, ys, label=name)
    yticks!(p, -5:1:maximum(ys))
end
p
```