ASEN 5519-003 Decision Making under Uncertainty
Homework 4: Tabular Reinforcement Learning

February 27, 2024

1 Conceptual Questions

Question 1. (30 pts) Consider a 3-armed Bernoulli Bandit with payoff probabilities $\theta = [0.2, 0.3, 0.7]$.

a) After a very large number of pulls, what is the expected payoff per pull of an ϵ-greedy policy with $\epsilon = 0.15$ (and no decay)?
b) After a very large number of pulls, what is the probability of selecting arm 3 when using a softmax policy with $\lambda = 5$ (and a "precision factor" of 1.0)?
c) Suppose that you are maintaining a Bayesian belief over the parameters θ starting with initial prior of Beta(1,1). Plot or sketch[1] the pdfs of the posterior probability distributions for each θ assuming the following numbers of wins and losses for each arm: $w = [0, 1, 3]$, $l = [1, 0, 2]$.
d) Given the situation in (c), describe one iteration of Thompson sampling. What quantities are sampled from what distributions? Choose some plausible values for the random samples and indicate which arm will be pulled.

Question 2. (20 pts) Consider the following simple MDP: $S = \{1, 2\}$, $A = \{L, R\}$. The initial state is 1, and 2 is a terminal state. Both actions result in deterministic transitions to state 2. $R(1, L) = 10$, $R(1, R) = 20$. Consider a policy parameterized with $\theta = [\theta_L, \theta_R]$, where

$$\pi_\theta(a \mid s) = \frac{e^{\theta_a}}{e^{\theta_L} + e^{\theta_R}}.$$

Calculate the policy gradients at $\theta = [0.5, 0.5]$ without baseline subtraction for two trajectories: $(1, L, 10, 2)$ and $(1, R, 20, 2)$.

2 Exercises

Question 3. (50 pts) Implement **two** tabular or deep learning algorithms to learn a policy for the `DMUStudent.HW4.gw` grid world environment. You will submit the following deliverables:

a) The **source code** for both of the algorithms.

b) **Two learning curve plots**[2]. The y-axis of both plots should be the average **undiscounted** reward per episode from the *learned policy* (not the exploration policy). Each plot should contain learning curves from both algorithms for easy comparison. The x-axis should be as follows for the two plots, respectively:

   1) The number of steps taken in the environment (calls to `act!`).

[1] You may wish to use `https://homepage.divms.uiowa.edu/~mbognar/applets/beta.html` for this.
[2] These are the same plots shown in the SARSA notebook from class, and you may copy code from there.

2) The cumulative wall-clock time for training.

c) **Write** a short paragraph describing the relative strengths of the algorithms. Which one has higher sample complexity? Which one learns faster in terms of wall clock time?

Some algorithms to consider implementing are:

- Policy Gradient
- Max-Likelihood Model Based RL
- Q-Learning[3]
- SARSA
- Actor Critic

Please meet the following requirements and consider the following tips:

1. *One* of algorithms may be copied from the course notebooks or from any other reinforcement learning library you can find online, but at least one must be implemented by you from scratch or modified from the notebooks. You may also implement both from scratch.
2. It is possible to achieve average **undiscounted** cumulative reward per episode of greater than 5. At least one of your algorithms must reach this level of performance.
3. Use only the functions from `CommonRLInterface` to interact with the environment, and use the `HW4.render` function if you want to render the environment.
4. You may also wish to modify these algorithms with techniques discussed in class, such as improved exploration policies, eligibility traces, double Q learning, or entropy regularization.

[3]Q-Learning is probably the easiest of these to implement since it is only a small modification from SARSA.

1) Consider a 3-armed Bernoulli Bandit with payoff probabilities
$\theta = \{0.2, 0.3, 0.7\}$

a) $0.7(1-\varepsilon) + \frac{1}{3}[0.2(\varepsilon) + 0.3(\varepsilon) + 0.7(\varepsilon)] = \underline{0.655}$

b) $P(a) = \frac{e^{\lambda p_a}}{\sum_i e^{\lambda p_i}}$ $\quad \sum_i e^{\lambda p_i} = e^{5(0.7)} + e^{5(0.3)} + e^{5(0.2)} = 40.315$

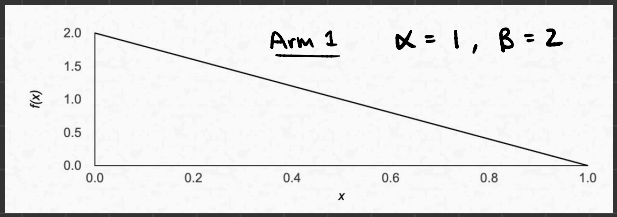
$P(1) = \frac{e^{5(0.2)}}{40.315} = 0.06743 = 6.743\%$

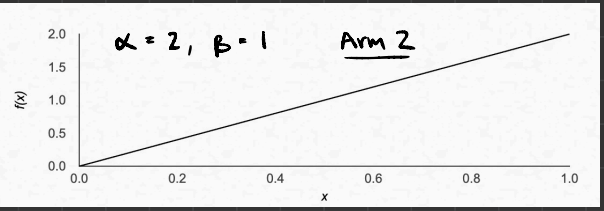
$P(2) = \frac{e^{5(0.3)}}{40.315} = 0.11117 = 11.117\%$

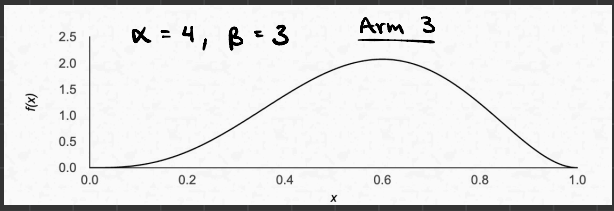
$P(3) = \frac{e^{5(0.7)}}{40.315} = 0.82141 = 82.141\%$

$[0.06743, 0.11117, \underline{0.82141}]$

c)

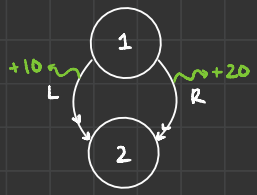






d) One possible iteration of Thompson Sampling would be to select 0.2, 0.8 and 0.6 from arms 1, 2, and 3 respectively. The outcome of this would be to pull arm 2 since the belief is the highest, then after pulling the arm, the beta distribution for arm 2 would be updated.

2)



$$\pi_\theta(a|s) = \frac{e^{\theta_a}}{e^{\theta_L} + e^{\theta_R}}$$

$\tau_L = (1, L, 10, 20)$:

$$\nabla_\theta \log \pi_\theta(a = L, s = 1) = \frac{\partial}{\partial \theta}\left[\log\left(\frac{e^{\theta_L}}{e^{\theta_L} + e^{\theta_R}}\right)\right]$$

$$= \frac{\partial}{\partial \theta}\left[\log(e^{\theta_L}) - \log(e^{\theta_L} + e^{\theta_R})\right]$$

$$= \frac{\partial}{\partial \theta}(\theta_L) - \frac{\partial}{\partial \theta}\left[\log(e^{\theta_L} + e^{\theta_R})\right]$$

$$= \left[1 - \frac{e^{\theta_L}}{e^{\theta_L} + e^{\theta_R}}, -\frac{e^{\theta_R}}{e^{\theta_L} + e^{\theta_R}}\right] = [0.5, -0.5]$$

$$\widehat{\nabla_\theta U(\theta)} = \nabla_\theta \log \pi_\theta \, R(\tau) = [5, -5]$$

$\tau_R = (1, R, 20, 2)$:

$$\nabla_\theta \log \pi_\theta(a = R, s = 1) = \frac{\partial}{\partial \theta}\left[\log\left(\frac{e^{\theta_R}}{e^{\theta_L} + e^{\theta_R}}\right)\right]$$

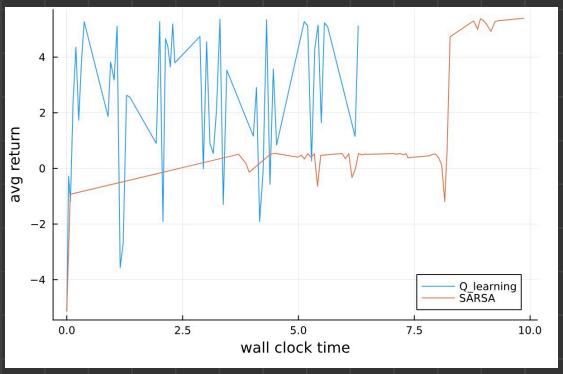
$$= \frac{\partial}{\partial \theta}\left[\log(e^{\theta_R}) - \log(e^{\theta_L} + e^{\theta_R})\right]$$

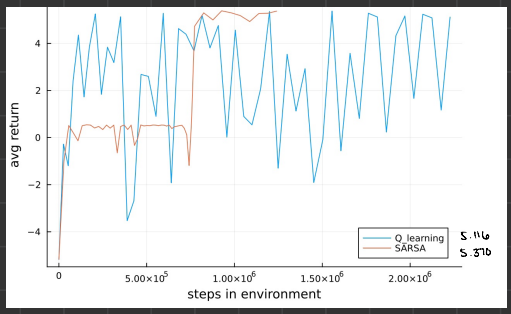
$$= \frac{\partial}{\partial \theta}(\theta_R) - \frac{\partial}{\partial \theta}\left[\log(e^{\theta_L} + e^{\theta_R})\right]$$

$$= \left[-\frac{e^{\theta_L}}{e^{\theta_L} + e^{\theta_R}}, 1 - \frac{e^{\theta_R}}{e^{\theta_L} + e^{\theta_R}}\right] = [-0.5, 0.5]$$

$$\widehat{\nabla_\theta U(\theta)} = \nabla_\theta \log \pi_\theta \, R(\tau) = [-10, 10]$$

3) Q-learning : 5.116 SARSA : 5.370





c) For both algorithms in terms of average reward, they were able to achieve similar performance, 5.116 for Q-learning and 5.370 for SARSA (both of these returns were computed using 100,000 episodes). Looking at the figures above, we can see Q-learning has a higher sample complexity than SARSA, but when comparing wall clock times it is still able to learn faster. I attribute this difference to the off-policy nature of Q-learning which favors the best action and does not get as easily stuck behind the negative reward.

```
using DMUStudent.HW4: HW4, gw, render
using POMDPModels: SimpleGridWorld
using LinearAlgebra: I
using CommonRLInterface: actions, act!, observe, reset!, AbstractEnv,
observations, terminated, clone
import POMDPTools
using SparseArrays
using Statistics: mean

function sarsa_episode!(Q, env; ϵ=0.10, γ=0.99, α=0.2)
    start = time()

    function policy(s)
        if rand() < ϵ
            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)

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

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

    Q[(s,a)] += α*(r - Q[(s, a)])

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

function sarsa!(env; n_episodes=100_000)
    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_episode!(Q, env;
                                       ϵ=max(0.1, 1-i/n_episodes)))
    end

    return episodes
end

function sarsa_lambda_episode!(Q, env; ϵ=0.10, γ=0.99, α=0.05, λ=0.9)

    start = time()

    function policy(s)
        if rand() < ϵ
            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

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

        for ((s, a), n) in N
            Q[(s, a)] += α*δ*n
            N[(s, a)] *= γ*λ
        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
    δ = r - Q[(s, a)]

    for ((s, a), n) in N
        Q[(s, a)] += α*δ*n
```

```
        N[(s, a)] *= γ*λ
    end

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

function sarsa_lambda!(env; n_episodes=100_000, 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;
                                              ϵ=max(0.01, 1-i/
n_episodes),
                                              kwargs...))
    end

    return episodes
end

function Q_learning_episode!(Q, env; ϵ=0.10, γ=0.99, α=0.2)
    start = time()

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

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

    while !terminated(env)
        a = policy(s, true)
        r = act!(env, a)
        sp = observe(env)

        ap = policy(sp, false)

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

        s = sp
        push!(hist, sp)
    end
```

```
    Q[(s,a)] += α*(r - Q[(s, a)])

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

function Q_learning!(env; n_episodes=100_000)
    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;
                                        ϵ=max(0.1, 1-i/n_episodes)))
    end

    return episodes
end

function double_Q_learning_episode!(Q1, Q2, N, env; ϵ=0.10, γ=0.99,
α=0.2, c=100)
    start = time()

    function policy(s)
        bonus(nsa, ns) = nsa == 0 ? Inf : sqrt(log(ns)/nsa)
        Ns = sum(N[(s,a)] for a in actions(m))
        return argmax(a->Q1[(s,a)] + c*bonus(N[(s,a)], Ns),
actions(m))
    end

    A = collect(actions(env))
    s = observe(env)
    hist = [s]
    a = policy(s)
    r = act!(env, a)

    while !terminated(env)
        a = policy(s)
        r = act!(env, a)
        sp = observe(env)

        ap = policy(sp)

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

        Q1[(s,a)] += α*(r + γ*Q2[(sp, A[argmax(Q1[(sp,ap)] for ap in
A)])] - Q1[(s, a)])
        Q2[(s,a)] += α*(r + γ*Q1[(sp, A[argmax(Q1[(sp,ap)] for ap in
A)])] - Q2[(s, a)])
```

```
        s = sp
        push!(hist, sp)
    end

    Q1[(s,a)] += α*(r - Q2[(s, a)])
    Q2[(s,a)] += α*(r - Q1[(s, a)])

    return (hist=hist, Q = copy(Q1), time=time()-start)
end

function double_Q_learning!(env; n_episodes=100_000)
    Q1 = Dict((s, a) => 0.0 for s in observations(env), a in
actions(env))
    Q2 = Dict((s, a) => 0.0 for s in observations(env), a in
actions(env))
    N =  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, double_Q_learning_episode!(Q1, Q2, N, env;
                                        ϵ=max(0.1, 1-i/n_episodes)))
    end

    return episodes
end

m = gw
env = convert(AbstractEnv, m)

using Plots

function evaluate(env, policy, n_episodes=100_000, max_steps=1000,
γ=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 += γ^t*act!(env, a)
            s = observe(env)
            t += 1
        end
        push!(returns, r)
    end
```

```
    return returns
end

sarsa_episodes             = sarsa!(            env,
n_episodes=100_000);
Q_learning_episodes        = Q_learning!(       env,
n_episodes=100_000);
# double_Q_learning_episodes = double_Q_learning!(env,
n_episodes=100_000);
# lambda_episodes            = sarsa_lambda!(     env,
n_episodes=100_000, α=0.1, λ=0.3);

# render(env, color = s -> maximum(map(a-
>last(sarsa_episodes).Q[(s,a)],actions(env))))
# render(env, color = s -> maximum(map(a-
>last(Q_learning_episodes).Q[(s,a)],actions(env))))

episodes = Dict("Q_learning"=>Q_learning_episodes,
"SARSA"=>sarsa_episodes)

p1 = plot(xlabel="steps in environment", ylabel="avg return")
n = 2_000
stop = 100_000
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!(p1, xs, ys, label=name)
    lastval = last(ys)
    display("$name $lastval")
end
p1
display(p1)

p2 = plot(xlabel="wall clock time", ylabel="avg return")
n = 2_000
stop = 100_000
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!(p2, xs, ys, label=name)
end
p2
```