Reinforcement Learning: An Introduction - Chapter 2

Thomas Hopkins

Exercise 2.1

The $\epsilon = 0.01$-greedy action selection method will perform the best in the long run (as the number of steps $\to \infty$) because it will be able to try each state an infinite number of times and it also exploits its current knowledge more than the $\epsilon = 0.1$-greedy action selection method. This method will be 10 times better than the $\epsilon = 0.1$ version.

Exercise 2.2

Let us first implement the ϵ-greedy action selection method.

In [1]:

```
function egreedy(q_values, e=0.0)
    if rand() < e
        return rand(1:length(q_values))
    end
    return argmax(q_values)
end
```

Out[1]: `egreedy (generic function with 2 methods)`

Now we implement the Gibbs or Boltzmann action selection method.

In [2]:

```
function sample(items, weights)
    weights = cumsum(weights)
    weights[end-1] = 1.0
    return findfirst(weights .>= rand())
end

function softmax(values, t)
    exponentials = exp.(values ./ t)
    denom = sum(exponentials)
    return exponentials ./ denom
end

function gibbs(q_values, t=0.0)
    if t == 0.0
        return argmax(q_values)
    end
    return sample(q_values, softmax(q_values, t))
end
```

Out[2]: `gibbs (generic function with 2 methods)`

With that out of the way, we can now use `ReinforcementLearningEnvironments` and import the `MultiArmBanditsEnv` to run the experiment. We will generate 2,000 10-armed bandit tasks and have both the ϵ-greedy and gibbs action selection methods learn to select optimal actions with varying ϵ and τ.

In [3]:

```
using Random
using Distributions


mutable struct MultiArmBanditsEnv
    true_values::Vector{Float64}
    distributions::Vector{Normal{Float64}}
    # cache
    reward::Float64
    is_terminated::Bool
end

function MultiArmBanditsEnv(; k = 10, rng = Random.GLOBAL_RNG)
    true_values = rand(rng, k)
    distributions = [Normal(true_values[i], ) for i = 1:length(true_values)
    MultiArmBanditsEnv(true_values, distributions, 0.0, false)
end

function (env::MultiArmBanditsEnv)(action)
    env.reward = rand(env.distributions[action])
    env.is_terminated = true
end



n = 10
num_tasks = 2000
iterations = 1000
params = [0.0, 0.01, 0.1]
rewards_epsilon = zeros(Float64, (length(params), iterations))
rewards_temperature = zeros(Float64, (length(params), iterations))
for i = 1:num_tasks
    env = MultiArmBanditsEnv(k=n, rng=Normal(0.0, 1.0))
    q_values_epsilon = zeros(Float64, (length(params), n))
    q_values_temperature = zeros(Float64, (length(params), n))
    q_counts_epsilon = zeros(Int, (length(params), n))
    q_counts_temperature = zeros(Int, (length(params), n))
    for j = 1:iterations
        for k = 2:length(params)
            e_action = egreedy(q_values_epsilon[k, :], params[k])
            t_action = gibbs(q_values_temperature[k, :], params[k]*100)
            env(e_action)
            e_reward = env.reward
            env(t_action)
            t_reward = env.reward
            rewards_epsilon[k, j] += ((1 / (i + 1)) * (e_reward - rewards_ep
            rewards_temperature[k, j] += ((1 / (i + 1)) * (t_reward - rewar
            q_values_epsilon[k, e_action] += ((1 / (q_counts_epsilon[k, e_ac
            q_values_temperature[k, t_action] += ((1 / (q_counts_temperatur
            q_counts_epsilon[k, e_action] += 1
            q_counts_temperature[k, t_action] += 1
        end
    end
end
```

In [4]:

```
using Plots

plot(1:iterations, transpose(rewards_epsilon), label = ["e = 0.0" "e = 0.01'
plot!(1:iterations, transpose(rewards_temperature), label = ["t = 0.0" "t =
```

Out[4]:

Exercise 2.3

The full gibbs distribution is given as

$$\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$$

Using only two actions ($n = 2$) we fix one of them to have a Q-value of 0. Then,

$$\begin{aligned}\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}} &= \frac{e^{Q_t(a)/\tau}}{e^{Q_t(a)/\tau} + e^{0}} \\ &= \frac{e^{Q_t(a)/\tau}}{e^{Q_t(a)/\tau} + 1}\end{aligned}$$

Here is a comparison of the two:

In [5]:

```
q_values = [0, 1]
println(softmax(q_values, 1.0))
sigmoid(x, t) = exp(x/t) / (exp(x/t) + 1.0)
v = sigmoid(q_values[2], 1.0)
println(1-v, ", ", v)
```

```
[0.2689414213699951, 0.7310585786300049]
0.2689414213699951, 0.7310585786300049
```

Exercise 2.4

In this stochastic case of the binary bandit task, we have one correct and one wrong action, a and b, respectively. With probability p, b is signaled to be the correct choice. Clearly if $p = 0$ then we have an optimal algorithm. If $p = 1$ then the algorithm performs the worst possible since it always believes that b is the correct choice. Only with $p < 0.5$ do we have the supervised learning algorithm converge to an optimal algorithm. With $p > 0.5$ it will converge toward always choosing the wrong action.

Exercise 2.5

```
Intialize Q(a) <- 0, K(a) <- 0  for a = 1,..., n
do forever
    a <- argmax(Q)
    r <- bandit(a)
    K(a) <- K(a) + 1
    Q(a) <- Q(a) + (1 / K(a))*(r - Q(a))
end do
```

Exercise 2.6

The weighting on each prior reward for the general case is the same derivation of (2.7) but using $\alpha_k(a)$ instead of the constant α. It gives

$$Q_k = Q_{k-1} + \alpha_k[r_k - Q_{k-1}]$$

$$= \alpha_k r_k + (1 - \alpha_k)Q_{k-1}$$

$$= \alpha_k r_k + (1 - \alpha_k)\alpha_{k-1}r_{k-1} + (1 - \alpha_{k-1})^2 Q_{k-2}$$

$$= (1 - \alpha_0)^k Q_0 + \sum_{i=1}^{k} \alpha_{i-1}(1 - \alpha_i)^{k-i} r_i$$

Exercise 2.7

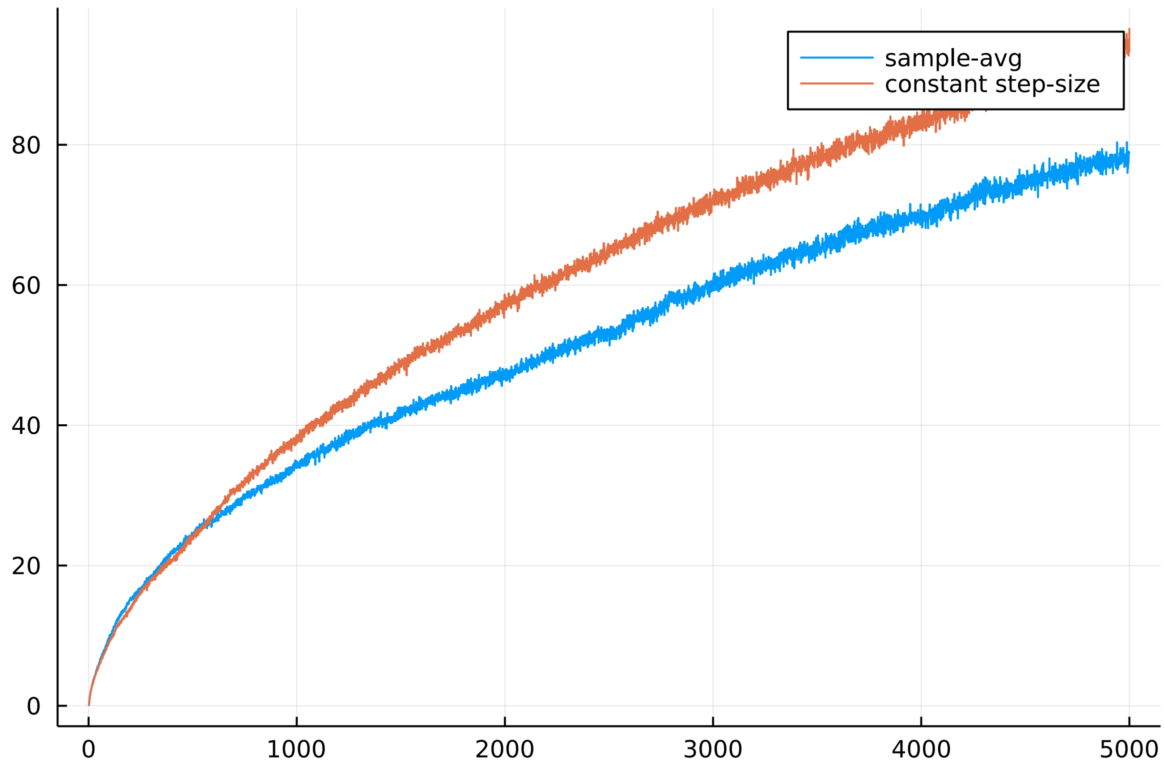
In [6]:

```
Random.seed!(123)


mutable struct MultiArmBanditsEnv
    true_values::Vector{Float64}
    distributions::Vector{Normal{Float64}}
    # cache
    reward::Float64
    is_terminated::Bool
end


function MultiArmBanditsEnv(; k = 10, rng = Random.GLOBAL_RNG)
    true_values = rand(rng, k)
    distributions = [Normal(true_values[i], ) for i = 1:length(true_values)]
    MultiArmBanditsEnv(true_values, distributions, 0.0, false)
end


function (env::MultiArmBanditsEnv)(action)
    env.reward = rand(env.distributions[action])
    env.true_values = [rand(Normal(env.true_values[i], )) for i = 1:length(e
    env.distributions = [Normal(env.true_values[i], ) for i = 1:length(env.
end


function simulate(iterations, method)
    n = 10
    num_tasks = 2000
    epsilon = 0.1
    rewards = zeros(Float64, iterations)
    for i = 1:num_tasks
        env = MultiArmBanditsEnv(k=n, rng=Normal(0.0, 1.0))
        q_values = zeros(Float64, n)
        if method == "sample-avg"
            q_counts = zeros(Int, n)
        end
        for j = 1:iterations
            e_action = egreedy(q_values, epsilon)
            env(e_action)
            e_reward = env.reward
            rewards[j] += ((1 / (i + 1)) * (e_reward - rewards[j]))
            if method == "sample-avg"
                q_values[e_action] += ((1 / (q_counts[e_action] + 1)) * (e_
                q_counts[e_action] += 1
            else
                q_values[e_action] += (0.1 * (e_reward - q_values[e_action]
            end
        end
    end
    return rewards
end


iterations = 5000
rewards_sample_avg = simulate(iterations, "sample-avg")
rewards_constant = simulate(iterations, "constant");
```

In [7]:

```
plot(1:iterations, rewards_sample_avg, label = "sample-avg")
plot!(1:iterations, rewards_constant, label = "constant step-size")
```

Out[7]:



Exercise 2.8

There are oscillations and spikes in the early part of the curve for the optimistic method because it will rapidly explore all of the possible actions early. There is a high variance in terms of which actions are selected to explore next and there are many different exploratory moves to choose from. A lucky find of the optimal action will still reduce the Q-value of that action but not by as much as the other actions. This can lead to an ealry spike in reward and will eventually lead to a rapid increase in reward compared to the ϵ-greedy method.

Exercise 2.9

I think that the temperature parameter was omitted for this method because the reference reward update accounts for this flexibility. Each $p_t(a)$ is updated with respect to the reference reward which changes over time. So, the parameters α and β will take on the role of the temperature in previous methods.

Exercise 2.10

The reference reward would just be the average preference over the actions since this would cause the update to be the same. This loses the flexibility similar to omitting the temperature from the Q-value Gibbs method in that you can no longer tune the agent to explore more or less.

Exercise 2.11

The experiment will be the same (stationary) 10-armed test bed we have used in the other exercises.

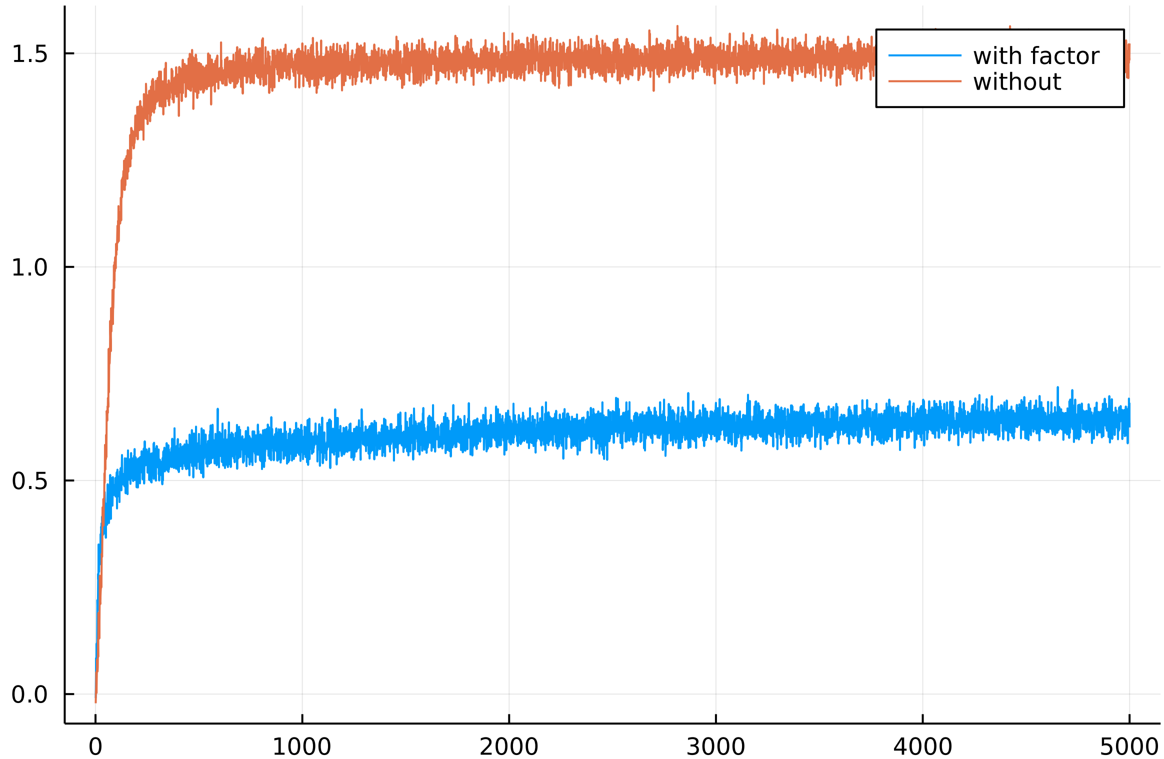
In [8]:

```
Random.seed!(123)


mutable struct MultiArmBanditsEnv
    true_values::Vector{Float64}
    distributions::Vector{Normal{Float64}}
    # cache
    reward::Float64
    is_terminated::Bool
end


function MultiArmBanditsEnv(; k = 10, rng = Random.GLOBAL_RNG)
    true_values = rand(rng, k)
    distributions = [Normal(true_values[i], ) for i = 1:length(true_values)
    MultiArmBanditsEnv(true_values, distributions, 0.0, false)
end


function (env::MultiArmBanditsEnv)(action)
    env.reward = rand(env.distributions[action])
end


function simulate(iterations, method)
    n = 10
    num_tasks = 2000
    beta = 0.1
    alpha = 0.2
    rewards = zeros(Float64, iterations)
    for i = 1:num_tasks
        env = MultiArmBanditsEnv(k=n, rng=Normal(0.0, 1.0))
        p_values = zeros(Float64, n)
        reference = 0.0
        for j = 1:iterations
            probabilities = softmax(p_values, 1.0)
            e_action = sample(p_values, probabilities)
            env(e_action)
            e_reward = env.reward
            rewards[j] += ((1 / (i + 1)) * (e_reward - rewards[j]))
            if method == "with factor"
                p_values[e_action] += (beta * (e_reward - reference) + (1 -
            else
                p_values[e_action] += (beta * (e_reward - reference))
            end
            reference += (alpha * (e_reward - reference))
        end
    end
    return rewards
end


iterations = 5000
rewards_with = simulate(iterations, "with factor")
rewards_without = simulate(iterations, "without");
```

In [9]:

```
plot(1:iterations, rewards_with, label = "with factor")
plot!(1:iterations, rewards_without, label = "without")
```

Out[9]:



Exercise 2.12

Yes, the pursuit algorithm will eventually select the optimal action with probability approaching 1. This is because the action values are updated according to the reward as before and the probabilities of selecting those actions are moved toward a greedy policy over time. Therefore, it will explore early and adapt its exploration based on the reward signal from the environment. This is similar to the Gibbs action selection method in the sense that it is adaptive. This is a different way to adapt the action selection method that is potentially more flexible (with different choices of β).

Exercise 2.13

Instead of using action probabilities directly, one can use the Q-values as targets to move $\pi(a*)$ towards. The algorithm could look something like this:

```
Initialize Q(a) <- 0, pi(a) <- 1/n  for a = 1,..., n
Initialize alpha, beta to be in (0, 1)
do forever
    a <- sample(softmax(pi))
    r <- bandit(a)
    Q(a) <- Q(a) + alpha * (r - Q(a))
    a* <- argmax(Q)
    for all a != a* do
        pi(a) <- pi(a) - beta * (Q(a) - pi(a))
    pi(a*) <- pi(a*) + beta * (Q(a*) - pi(a*))
end do
```

Exercise 2.14

My algorithm did not perform very well as seen from the plot below (comparing against earlier exercises). The idea to use Q-values as a target for the "probabilities" did not work well. It is very sensitive to the choices of α and β. For small β, all of the probabilities stay the same and it basically chooses randomly. There exists a sharp spike when the perceived optimal action has been found and can no longer be changed by the algorithm.

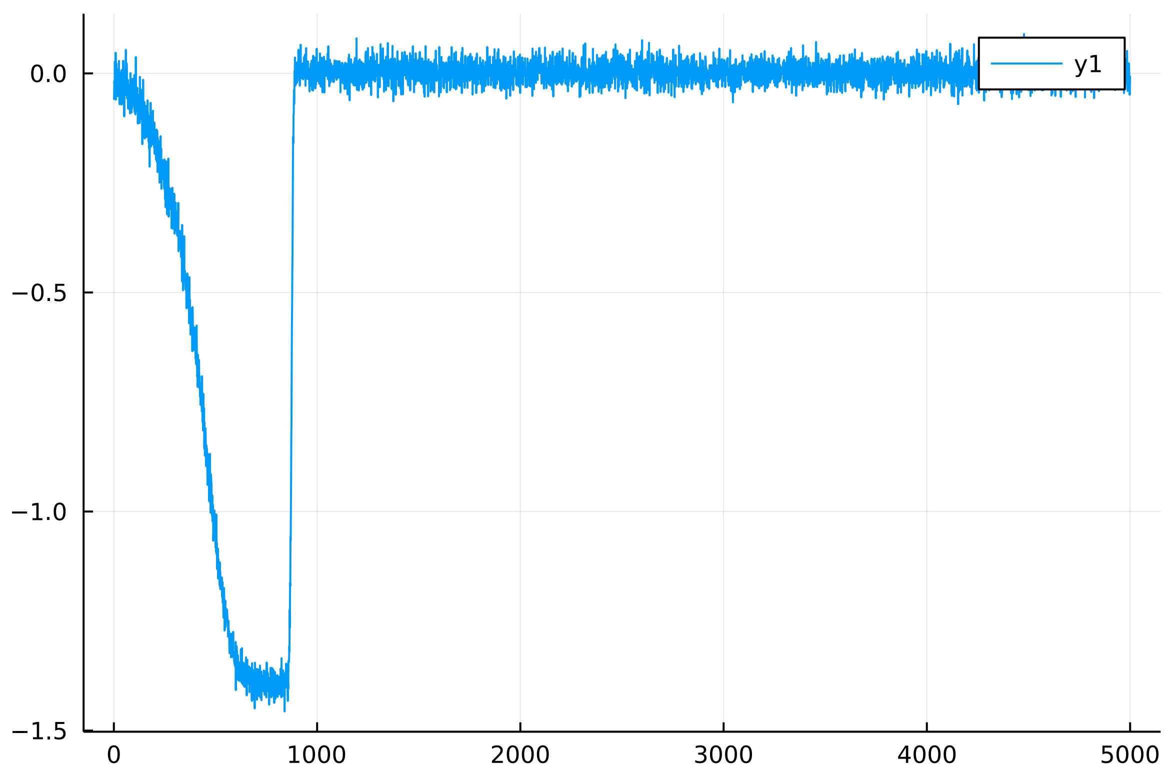
In [10]:

```
function simulate_custom(iterations)
    n = 10
    num_tasks = 2000
    beta = 0.01
    alpha = 0.001
    rewards = zeros(Float64, iterations)
    for i = 1:num_tasks
        env = MultiArmBanditsEnv(k=n, rng=Normal(0.0, 1.0))
        q_values = zeros(Float64, n)
        p_values = [1 / n for i = 1:n]
        for j = 1:iterations
            probabilities = softmax(p_values, 1.0)
            e_action = sample(p_values, probabilities)
            env(e_action)
            e_reward = env.reward
            rewards[j] += ((1 / (i + 1)) * (e_reward - rewards[j]))
            q_values[e_action] += (alpha * (e_reward - q_values[e_action]))
            g_action = argmax(q_values)
            for i = 1:n
                if g_action != i
                    p_values[i] -= (beta * (q_values[i] - p_values[i]))
                end
            end
            p_values[g_action] += (beta * (q_values[g_action] - p_values[g_a
        end
    end
    return rewards
end

iterations = 5000
rewards_custom = simulate_custom(iterations)
```

Out[10]:
```
5000-element Vector{Float64}:
 -0.06044906776575275
 -0.018203868378677712
 -0.02351648664876688
 -0.03241828222937888
  0.027208555436068718
 -0.028296218447641772
 -0.042833890491520435
 -0.0590521149686517
  0.04721218674253455
 -0.0597958668168524
  0.006751044935165389
  0.008936426401928056
  0.022173372252223787
  ⋮
 -0.021502185769916683
 -0.01062845645801696
 -0.023830393746199528
 -0.019325987791440572
  0.02075537449676534
  0.02310695742216872
  0.007474030898136711
 -0.020801084900020764
 -0.018785372706220372
 -0.048809877776523763
 -0.006398082468642156
 -0.022238959857397395
```

In [11]:
```
plot(1:iterations, rewards_custom)
```

Out[11]:



Exercise 2.15

The simplest idea would be to choose the action based on the following policy:

$$a_t = \begin{cases} random & p = \epsilon \\ softmax(\pi) & p = 1 - \epsilon \end{cases}$$

This meaning that you choose an action a randomly with probability ϵ. Otherwise, you choose based on a softmax over the "probabilities" found using the pursuit method.

Exercise 2.16

Clearly from the problem, you should choose action 2 for case A and choose action 1 for case B. Here is a table of the task:

	1	2
A	0.1	0.2
B	0.9	0.8

Now if the probability of being in case A or case B is 0.5 then,

$$E[r|a=1] = 0.5(0.1) + 0.5(0.9) = 0.5$$

$$E[r|a=2] = 0.5(0.2) + 0.5(0.8) = 0.5$$

Therefore, random policy is optimal.

For the other task, the expectation of success is much better. This is because you are told which task you are facing. For example, if you know with certainty that $p(A) = 1.0$ (probability of the case being A), then

$$E[r|a=1] = 1.0(0.1) + 0.0(0.9) = 0.1$$

$$E[r|a=2] = 1.0(0.2) + 0.0(0.8) = 0.2$$

and you would obviously choose $a = 2$ since the expected reward is greater.

On the flip side, if you know with certainty that $p(B) = 1.0$ then

$$E[r|a=1] = 0.0(0.1) + 1.0(0.9) = 0.9$$

$$E[r|a=2] = 0.0(0.2) + 1.0(0.8) = 0.8$$

and you would choose $a = 1$ since the expected reward is greater.