**Reinforcement Learning: An Introduction Exercises**

Chapter 1

Exercise 1.1: Self-Play: Suppose, instead of playing against a random opponent, the

reinforcement learning algorithm described above played against itself, with both sides

learning. What do you think would happen in this case? Would it learn a different policy

for selecting moves?

It may learn to make moves to maximize its reward in for both players. It could end up being a fixed path of one always winning or a cyclical path of tradeoffs of winning. Tying would be eliminated.

Exercise 1.2: Symmetries: Many tic-tac-toe positions appear different but are really

the same because of symmetries. How might we amend the learning process described

above to take advantage of this? In what ways would this change improve the learning

process? Now think again. Suppose the opponent did not take advantage of symmetries. In that case, should we? Is it true, then, that symmetrically equivalent positions should necessarily have the same value?

We could eliminate some of the state-space if states are equal in different orientations of the board. It would make the learning process more efficient. If the opponent does not take advantage of symmetries, then neither should our algorithm because the player will react differently meaning our algorithm may be suboptimal if it takes advantage of symmetries.

Exercise 1.3: Greedy Play: Suppose the reinforcement learning player was greedy, that is, it always played the move that brought it to the position that it rated the best. Might it

learn to play better, or worse, than a nongreedy player? What problems might occur?

It will find an initial method that gives some reward and stick with it possibly missing out on actions that could yield higher rewards for more exploratory policies

Exercise 1.4: Learning from Exploration: Suppose learning updates occurred after all

moves, including exploratory moves. If the step-size parameter is appropriately reduced

over time (but not the tendency to explore), then the state values would converge to a

set of probabilities. What are the two sets of probabilities computed when we do, and

when we do not, learn from exploratory moves? Assuming that we do continue to make

exploratory moves, which set of probabilities might be better to learn? Which would

result in more wins?

When we learn from exploratory moves we could update a state's overall value based upon exploratory moves being suboptimal even if the state itself leads to an optimal reward in every other case. It is better not to learn from the exploratory actions so as to avoid this.

Exercise 1.5: Other Improvements: Can you think of other ways to improve the reinforcement learning player? Can you think of any better way to solve the tic-tac-toe problem as posed?

We could make a list of good moves and use this to speed the learning of our algorithm. We could use something other than reinforcement learning, such as a deterministic best move algorithm since the environment can be completely mapped and a best move could be determined from there.

Chapter 2

Exercise 2.1 In ε-greedy action selection, for the case of two actions and ε = 0.5, what is

the probability that the greedy action is selected?

50% for both the greedy and exploratory actions.

Exercise 2.2: Bandit example: Consider a k-armed bandit problem with k = 4 actions,

denoted 1, 2, 3, and 4. Consider applying to this problem a bandit algorithm using

ε-greedy action selection, sample-average action-value estimates, and initial estimates

of Q1(a) = 0, for all a. Suppose the initial sequence of actions and rewards is A1 = 1,

R1 = 1, A2 = 2, R2 = 1, A3 = 2, R3 = 2, A4 = 2, R4 = 2, A5 = 3, R5 = 0. On some

of these time steps the ε case may have occurred, causing an action to be selected at

random. On which time steps did this definitely occur? On which time steps could this

possibly have occurred?

A2 and A5 are definitely exploratory. We know this because if A2 was greedy, it would have chosen the action that had resulted in the most reward so far, 1, but instead chose 2. We know the same for A5 since the best action was 2, but it chose 3.

Exercise 2.3 In the comparison shown in Figure 2.2, which method will perform best in

the long run in terms of cumulative reward and probability of selecting the best action?

How much better will it be? Express your answer quantitatively.

epsilon-greedy with an epsilon of .1 will perform the best in the long run. It will result in the highest cumulative reward and probability of selecting the best action. It will do so due to the epsilon chance of an exploratory action. It may not gain as much reward in the beginning, but it will find the best actions due to its exploration. It will find the best actions faster than the other two. The .01 epsilon greedy algorithm will eventually find the best state-action pairs as the .1 epsilon greedy did. It will take the actions it thinks or knows as best more often than exploring meaning it may suffer from selecting sub-par actions for a longer period than the .1 epsilon greedy algorithm. The greedy algorithm will never explore and will almost certainly remain making sub-par decisions about state-actions.

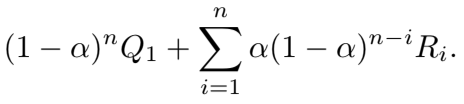
Exercise 2.4 If the step-size parameters, αn, are not constant, then the estimate Qn is

a weighted average of previously received rewards with a weighting different from that

given by (2.6). What is the weighting on each prior reward for the general case, analogous to (2.6), in terms of the sequence of step-size parameters?

If the step-size parameters are not constant, then we must keep up with them.

Qn+1 = i=1∏n [1- αi]Q1 + i=1∑n [αi • j=1∏i [1- αj]\*Ri] instead of



Qn+1 =

Exercise 2.5 (programming): Design and conduct an experiment to demonstrate the

difficulties that sample-average methods have for nonstationary problems. Use a modified version of the 10-armed testbed in which all the q∗(a) start out equal and then take independent random walks (say by adding a normally distributed increment with mean zero and standard deviation 0.01 to all the q∗(a) on each step). Prepare plots like

Figure 2.2 for an action-value method using sample averages, incrementally computed,

and another action-value method using a constant step-size parameter, α = 0.1. Use

ε = 0.1 and longer runs, say of 10,000 steps.

1. import numpy as np
2. import matplotlib.pyplot as plt

5. #create our bandits with initial values for reward and a std for that reward
6. def create\_bandits(n, mean=0, std=1, b\_std=1):
7. bandits = []
9. **for** i in range(n):
10. bandits.append([np.random.normal(mean, std), b\_std])
12. **return** bandits

15. def bandit\_random\_walk(bandits, shift):
16. **for** bandit in bandits:
17. bandit[0] += shift **if** np.random.uniform() < 0.5 **else** -shift
18. **return** bandits

21. def main():
23. num\_bandits = 10
25. #amount of episodes for convergence - what is the difference in reruns vs episodes from logical perspective
26. episodes = 1000
28. #step size parameter
29. alpha = 0.1
31. #exploration chance parameter
32. epsilon = 0.1
34. #random walk shift for each bandit's reward over time.
35. shift = 0.01
37. #What is this?
38. merge\_choices\_num = 1
40. #amount of reruns for convergence
41. reruns = 1000
43. merged\_choices = []
45. #rerun to get a better average of the performance
46. **for** h in range(reruns):
48. #keep track of the number of times we chose the best bandit (made the correct choice)
49. correct\_choices = []
51. #create our bandits
52. bandits = create\_bandits(num\_bandits)
54. max\_bandit = 0
55. best\_bandit = 0
56. **for** i in range(len(bandits)):
58. #set max bandit to the reward of our best bandit
59. #set best bandit to the number of the best bandit
60. **if** bandits[i][0] > max\_bandit:
61. max\_bandit = bandits[i][0]
62. best\_bandit = i
63. print str(max\_bandit)
65. #instantiate q and n for all bandits to zero
66. #q being the estimated value for that bandit (?)
67. #n being the number of times we've chosen that bandit.
68. q = np.zeros(num\_bandits)
69. n = np.zeros(num\_bandits)
71. # repeat for a number of episodes to get a better average for the values for the bandits.
72. **for** i in range(episodes):
74. #shift bandit randomly for better or worse by shift amount
75. bandit\_random\_walk(bandits, shift)    # Random walk by shift amount
77. #select the bandit with the best estimated value
78. selected\_bandit = np.argmax(q)
80. #decide if we will keep selected bandit with best estimated value (greedy) or if we will explore a random other bandit
81. **if** np.random.uniform() < epsilon:
82. selected\_bandit = np.random.choice(len(q))
84. #get the actual reward based on the reward with a std of b\_std from bandit creation function
85. reward = np.random.normal(bandits[selected\_bandit][0], bandits[selected\_bandit][1])
87. #increment n (count) for this bandit
88. n[selected\_bandit] += 1
90. ##Update the estimated value for the future for this bandit.
91. q[selected\_bandit] += (reward - q[selected\_bandit]) / n[selected\_bandit]    # Sample Average
92. #alpha = 1/(i + 1)  # Variable alpha
93. #q[selected\_bandit] += alpha \* (reward - q[selected\_bandit])   # Using alpha
95. #if we selected the bandit with the highest actual reward, then append a 1 (true) to this array
96. #else append a 0 (false) to this array.
97. correct\_choices.append(**int**(selected\_bandit == best\_bandit))
99. #if we are on an episode before merge\_choices\_num, then merged\_choice = correct choice percentage
100. **if** i <= merge\_choices\_num:
101. merged\_choice = sum(correct\_choices)/len(correct\_choices)
102. #else
103. **else**:
104. merged\_choice = sum(correct\_choices[-merge\_choices\_num:])/merge\_choices\_num
105. print str(merged\_choice)
107. # if this is the first run, append the merged\_choice into merged\_choices
108. **if** h == 0:
109. merged\_choices.append(merged\_choice)
110. # else update the merged\_choice for this episode with the
111. **else**:
112. merged\_choices[i] += (merged\_choice - merged\_choices[i]) / (h + 1)
114. print str(merged\_choices[i])
116. plt.axis([0, episodes, -0.5, 1.5])
117. plt.plot(range(episodes), merged\_choices)
118. plt.show()

121. **if** \_\_name\_\_ == "\_\_main\_\_":
122. main()

I put comments in the above code that I found online to try to understand this better. I think I understand the problem and the solution. Sample-average methods cannot weight the most recent rewards properly to account for non-stationary problems. Non-stationary problems experience drift in the rewards and the estimated values need to reflect this shift. With sample-average methods, our later rewards carry less weight than needed to reflect the drift in rewards whereas with constant step-size methods we are able to weight recent rewards appropriately to account for this drift.

I did have a question about the above code. When I run it, the graph appears to just be y=0. Maybe it is incorrect. We might should discuss it some.

Exercise 2.6: Mysterious Spikes: The results shown in Figure 2.3 should be quite reliable because they are averages over 2000 individual, randomly chosen 10-armed bandit tasks. Why, then, are there oscillations and spikes in the early part of the curve for the optimistic method? In other words, what might make this method perform particularly better or worse, on average, on particular early steps?

The optimistic initial values cause the algorithm to explore in the beginning since these values are far above the actual rewards that each action will receive. This means that each action will selected and receive a disappointing reward, thus encouraging other actions to be selected in future episodes. This can be used even with a greedy policy as the greedy policy will continue to select the high-valued, optimistic actions over the actions that have been selected thus far. This type of method will perform well on tasks that are stationary as it will maximize the choice of reward in the long run for stationary tasks. On the other hand, it is not well-suited for non-stationary tasks because it will not continue to explore as the rewards are updated for each state-action pair.

Exercise 2.7: Unbiased Constant-Step-Size Trick: In most of this chapter we have used

sample averages to estimate action values because sample averages do not produce the initial bias that constant step sizes do (see the analysis in (2.6)). However, sample

averages are not a completely satisfactory solution because they may perform poorly

on nonstationary problems. Is it possible to avoid the bias of constant step sizes while

retaining their advantages on nonstationary problems? One way is to use a step size of

βt = α/ot, (2.8)

where α > 0 is a conventional constant step size, and o ̄t is a trace of one that starts at 0:

ot+1 = ot + α(1 − ot), for t ≥ 1, with o1 = α. (2.9)

Carry out an analysis like that in (2.6) to show that βt is an exponential recency-weighted average without initial bias.

If using βt, our initial selection of each action will result in that action being set to an estimated value of 0 since our step size will be 1. Then we will slowly make our way back toward the value of α as βt, as described above, converges to the value of α.

I’m not entirely sure that I’m correct on this one.

Exercise 2.8: UCB Spikes: In Figure 2.4 the UCB algorithm shows a distinct spike

in performance on the 11th step. Why is this? Note that for your answer to be fully

satisfactory it must explain both why the reward increases on the 11th step and why it

decreases on the subsequent steps. Hint: if c = 1, then the spike is less prominent.

UCB selects each action of our k-number of bandits once at the start as they are seen to be maximizing actions if they have not been selected previously. It then selects the best action of those to maximize reward with regards to how many times the action has been selected and what time-step we are currently at. With a confidence level, c, of 2, we get a very prominent spike after the initial choosing of each action due to the algorithm choosing actions with high rewards a number of times before falling back and choosing those actions that have not been selected very many times due to the weight it puts on uncertainty in the estimated value of an action.

Exercise 2.9: Show that in the case of two actions, the soft-max distribution is the same

as that given by the logistic, or sigmoid, function often used in statistics and artificial

neural networks.

Soft-max is a generalization of the logistic function that is used in statistics and artificial neural networks. In the logistic function we are doing a binary classification whereas in soft-max we are doing a multi-class classification. It is a multi-nomial logistic regression function.

Exercise 2.10: Suppose you face a 2-armed bandit task whose true action values change randomly from time step to time step. Specifically, suppose that, for any time step, the true values of actions 1 and 2 are respectively 0.1 and 0.2 with probability 0.5 (case A), and 0.9 and 0.8 with probability 0.5 (case B). If you are not able to tell which case you face at any step, what is the best expectation of success you can achieve and how should you behave to achieve it? Now suppose that on each step you are told whether you are facing case A or case B (although you still don’t know the true action values). This is an associative search task. What is the best expectation of success you can achieve in this task, and how should you behave to achieve it?

In the initial task where we are unsure of the situation, or context, and we are also unsure of the true action values, we could try using some of the algorithms discussed in this chapter to achieve good results, but we probably won’t achieve very good results. The best expectation is to stick with one of the bandits every time and guarantee an average reward of 1. The second task where we are aware of the case that we are facing we can use trial-and-error to figure out the best action to take for that specific case. This would help us develop a policy to maximize our reward over time.

Chapter 3

Exercise 3.1: Devise three example tasks of your own that fit into the MDP framework,

identifying for each its states, actions, and rewards. Make the three examples as different from each other as possible. The framework is abstract and flexible and can be applied in many different ways. Stretch its limits in some way in at least one of your examples.

Answer here.

Exercise 3.5: The equations in Section 3.1 are for the continuing case and need to be

modified (very slightly) to apply to episodic tasks. Show that you know the modifications

needed by giving the modified version of (3.3).

The equation would remain the same unless s’ is a terminal state in which case we would reset to the initial state.

Exercise 3.6 Suppose you treated pole-balancing as an episodic task but also used

discounting, with all rewards zero except for −1 upon failure. What then would the

return be at each time? How does this return differ from that in the discounted, continuing formulation of this task?

The return at each time would be 0 and would be a discounted -1 upon failure. This would reset upon failure resetting the discount value. This will make the next failure weight much heavier than if we were to use a continuous formulation. If we used a continuous formulation, it would receive a highly discounted negative reward upon each successive failure, making each failure not as impactful on future decisions. It will encourage more time steps before failure with the episodic task formulation. ??

Exercise 3.7: Imagine that you are designing a robot to run a maze. You decide to give it a reward of +1 for escaping from the maze and a reward of zero at all other times. The task seems to break down naturally into episodes—the successive runs through the maze—so you decide to treat it as an episodic task, where the goal is to maximize expected total reward (3.7). After running the learning agent for a while, you find that it is showing no improvement in escaping from the maze. What is going wrong? Have you effectively communicated to the agent what you want it to achieve?

If we do not give algorithm a discount factor on rewards, then there is no reward for the learning agent finishing the maze in less time steps since the reward bill be 1 no matter how many time steps it takes the learning agent to finish. The agent may have also gotten stuck before finding the exit. A good solution would be to implement a negative reward for each time step that the agent does not find the exit.

Exercise 3.8: Suppose γ = 0.5 and the following sequence of rewards is received R1 = −1, R2 = 2, R3 = 6, R4 = 3, and R5 = 2, with T = 5. What are G0, G1, . . ., G5? Hint:

Work backwards.

G5 = R6 = 0;

G4 = R5 + .5(G5) = 2 + 0 = 2

G3 = R4 + .5(G4) = 3 + 1 = 4

G2 = R3 + .5(G3) = 6 + 2 = 8

G1 = R2 + .5(G2) = 2 + 4 = 6

G0 = R1 + .5(G1) = (-1) + 3 = 2

Exercise 3.9 Suppose γ = 0.9 and the reward sequence is R1 = 2 followed by an infinite

sequence of 7s. What are G1 and G0?

G0 = 2 + .9(70) = 65

G1 = 7 / (1-(.9)) = 7 / (.1) = 70

Exercise 3.12: The Bellman equation (3.14) must hold for each state for the value function vπ shown in Figure 3.2 (right) of Example 3.5. Show numerically that this equation holds for the center state, valued at +0.7, with respect to its four neighboring states, valued at +2.3, +0.4, −0.4, and +0.7. (These numbers are accurate only to one decimal place.)

Since there is an equal probability of each state, we can sum the values of all four states to get a value of 3. Then we can divide this by the number of states, 4, and multiply by our discount factor. This gives us 0.7 if we truncate down to 1 decimal place.

Exercise 3.13: In the gridworld example, rewards are positive for goals, negative for

running into the edge of the world, and zero the rest of the time. Are the signs of these

rewards important, or only the intervals between them? Prove, using (3.8), that adding a constant c to all the rewards adds a constant, vc, to the values of all states, and thus

does not affect the relative values of any states under any policies. What is vc in terms

of c and γ?

By 3.8 we have that G(t) = Rt+1 + yRt+2 + y2Rt+3 + y3Rt+4 …

Then we have G(t) = sum(yk(Rt+k+1)) as k=0 to k=infinity

Adding a constant to each reward will result in G(t) = sum(yk(Rk+t+1 + c)) as k=0 to k=infinity

Exercise 3.14 Now consider adding a constant c to all the rewards in an episodic task,

such as maze running. Would this have any effect, or would it leave the task unchanged

as in the continuing task above? Why or why not? Give an example.

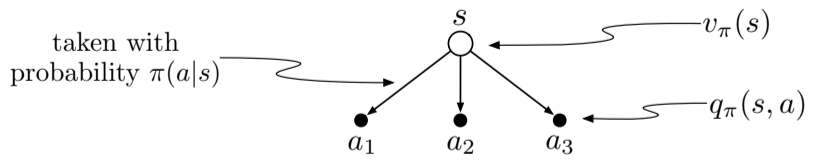
Answer here.

Exercise 3.16 The value of a state depends on the values of the actions possible in that

state and on how likely each action is to be taken under the current policy. We can

think of this in terms of a small backup diagram rooted at the state and considering each

possible action:



Give the equation corresponding to this intuition and diagram for the value at the root

node, vπ(s), in terms of the value at the expected leaf node, qπ(s, a), given St = s. This

equation should include an expectation conditioned on following the policy, π. Then give

a second equation in which the expected value is written out explicitly in terms of π(a|s)

such that no expected value notation appears in the equation.

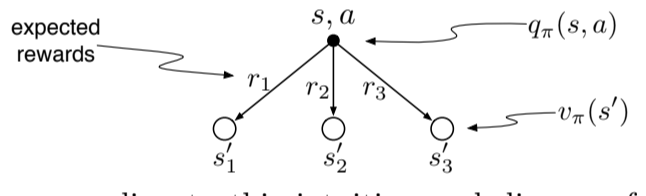
Answer here.

Exercise 3.17 The value of an action, qπ(s, a), depends on the expected next reward and

the expected sum of the remaining rewards. Again we can think of this in terms of a

small backup diagram, this one rooted at an action (state–action pair) and branching to

the possible next states:



Give the equation corresponding to this intuition and diagram for the action value,

qπ(s, a), in terms of the expected next reward, Rt+1, and the expected next state value,

vπ(St+1), given that St =s and At =a. This equation should include an expectation but

not one conditioned on following the policy. Then give a second equation, writing out the

expected value explicitly in terms of p(s0, r|s, a) defined by (3.2), such that no expected

value notation appears in the equation.

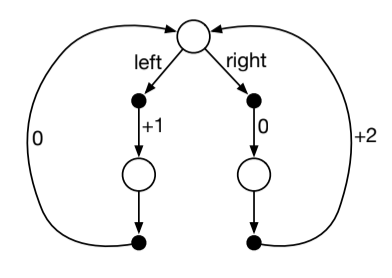
Answer here.

Exercise 3.20 Consider the continuing MDP shown on to the

right. The only decision to be made is that in the top state,

where two actions are available, left and right. The numbers

show the rewards that are received deterministically after

each action. There are exactly two deterministic policies,

πleft and πright. What policy is optimal if γ = 0? If γ = 0.9?

If γ = 0.5?

Answer here.

Exercise 3.23 Give an equation for v∗ in terms of q∗.

Answer here.

Exercise 3.25 Give an equation for π∗ in terms of q∗.

Answer here.

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 8