**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.

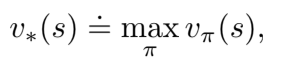
Exercise 2.4 If the step-size parameters, αn, are not constant, then the estimate Qn is

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=1Õn [1- αi]Q1 + i=1ån [αi · j=1Õi [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

3.

4.

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 = []

8.

9. **for** i in range(n):

10. bandits.append([np.random.normal(mean, std), b\_std])

11.

12. **return** bandits

13.

14.

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

19.

20.

21. def main():

22.

23. num\_bandits = 10

24.

25. #amount of episodes for convergence - what is the difference in reruns vs episodes from logical perspective

26. episodes = 1000

27.

28. #step size parameter

29. alpha = 0.1

30.

31. #exploration chance parameter

32. epsilon = 0.1

33.

34. #random walk shift for each bandit's reward over time.

35. shift = 0.01

36.

37. #What is this?

38. merge\_choices\_num = 1

39.

40. #amount of reruns for convergence

41. reruns = 1000

42.

43. merged\_choices = []

44.

45. #rerun to get a better average of the performance

46. **for** h in range(reruns):

47.

48. #keep track of the number of times we chose the best bandit (made the correct choice)

49. correct\_choices = []

50.

51. #create our bandits

52. bandits = create\_bandits(num\_bandits)

53.

54. max\_bandit = 0

55. best\_bandit = 0

56. **for** i in range(len(bandits)):

57.

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)

64.

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)

70.

71. # repeat for a number of episodes to get a better average for the values for the bandits.

72. **for** i in range(episodes):

73.

74. #shift bandit randomly for better or worse by shift amount

75. bandit\_random\_walk(bandits, shift) # Random walk by shift amount

76.

77. #select the bandit with the best estimated value

78. selected\_bandit = np.argmax(q)

79.

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))

83.

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])

86.

87. #increment n (count) for this bandit

88. n[selected\_bandit] += 1

89.

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

94.

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))

98.

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)

106.

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)

113.

114. print str(merged\_choices[i])

115.

116. plt.axis([0, episodes, -0.5, 1.5])

117. plt.plot(range(episodes), merged\_choices)

118. plt.show()

119.

120.

121. **if** \_\_name\_\_ == "\_\_main\_\_":

122. main()

123.

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.

Chris: Following the analysis in (2.6) we can see what happens. Let’s first look at what Q\_2 is.

(B = beta)

Q\_2 = Q\_1 + B\_1(R\_1 - Q\_1)

= B\_1 R\_1 + (1 - B\_1) Q\_1

So far this is all exactly as in (2.6), just with B\_1 instead of alpha\_1. What is B\_1?

B\_1 = alpha / o-bar\_1

o-bar\_1 is defined to be alpha in the problem statement, so this gives us B\_1 = alpha / alpha = 1

Plugging this back into our equation for Q\_2 we have

Q\_2 = R\_1 + (1-1)Q\_1

= R\_1

So the initial Q\_1 gets multiplied in the later estimates by a factor of 0 = (1-B\_1), and so doesn’t contribute to, or impact, or *bias*, the estimates. If you work through what Q\_3, or Q\_n for any n > 2, you will see that they all can be rewritten in terms of the earlier Q’s, eventually Q\_2, which is R\_1, and breaks any connection to Q\_1.

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. ??

Chris:

The continuous formulation includes in its return the discounted future rewards (penalties) from future failures. The episodic only includes in its return the discounted next failure penalty, however far away that is. If we assume that all failures happen k steps apart, then the return when starting a new episode is (where g = gamma)

Episodic case: - g^k

Continuing case: -1/(1-g^k)

The continuing case return is because every k steps in the future we will get a reward of -1 discounted by an additional g^k. This return is the sum from i=1 to infinity of gamma^(ik), which gives us what I said.

I am not clear exactly on how those different returns will lead to anything different in the learning, but as you vary k and g in the plot linked below you can kind of see how their relative magnitude changes, so I am sure there will be some effect.

Link to graph where I was playing around with things: <https://www.desmos.com/calculator/2qvp3evrwf>

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. I believe this would also

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(y^k(Rk+t+1 + c)) as k=0 to k=infinity

Which can be simplified to V(c) = sum(y^k)c

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.

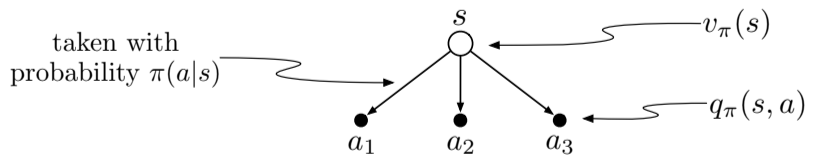
G(t) = (sum\_i (sum\_k\_i (y^k\_i )c)/i) as k = 0 to k = n and as i = 0 to i = m (m=number of reruns) (k=number of states per episode) ← This can almost definitely be simplified. I just wanted to rewrite while I had it figured out mentally. It would differ in that each state would be visited a limited number of times (possibly only once per state) per episode. Once the episode started over, the discount factor’s exponent would be reset. The task would ensue again. I would think these values would be relatively close to the values from a similar continuous task. I think this question is somewhat analogous to the question about pole-balancing in which episodic versus continual was used. The discount factor would cause some differences that would affect estimated value especially in the short term. The episodic formulation, I think would perform better (at least on episodic type tasks). The continuous might perform better on continuous type tasks that did not involve drift.

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.

V\_pi (s) = E\_pi {R\_t | S\_t = s}

= sum\_a (E\_pi{R\_t | S\_t = s, A\_t = a} x p(S\_t = s | A\_t = a))

= sum\_a (E\_pi{R\_t | S\_t = s, A\_t = a} x pi(s,a))

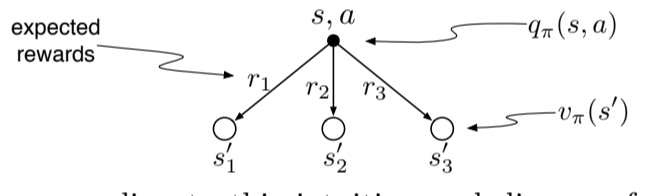
= sum\_a (q\_pi (s,a) x pi(a|s)

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.

Q\_pi (s,a) = E\_pi {r\_t+1 | S\_t = s, A\_t = a} + yE\_pi {sum\_k (y^k(r\_t+k+2) | S\_t = s, A\_t = a)}

Both the first and second term simplify from here.

E\_pi {r\_t+1 | S\_t = s, A\_t = a} = sum\_s’ (E\_pi {r\_t+1 | S\_t = s, A\_t = a, S\_t+1 = s’} x p(S\_t+1 = s’ | S\_t = s, A\_t = a)

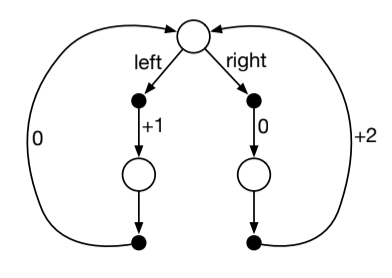
= sum\_s’ (R\_s,s’,a x P\_s,s’,a)

And

E\_pi {sum\_k (y^k(r\_t+k+2) | S\_t = s, A\_t = a)} = sum\_s’ (E\_pi {sum\_k (y^k(r\_t+k+2) | S\_t = s, A\_t = a, S\_t+1 = s’)} x p(S\_t+1 = s’ | S\_t = s, A\_t = a))

= sum\_s’ (E\_pi {sum\_k (y^k(r\_t+k+2) | S\_t+1 = s’)})

Q\_pi (s,a) = sum\_s’ (R\_a,s,s’ + yV\_pi (s’)))P\_a,s,s’



Exercise 3.20 Consider the continuing MDP shown 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?

If y = 0, then the agent would get the full first reward and the following rewards multiplied by the discount factor 0^k. Thus, left would be better since the agent will receive a reward of 1 followed by infinite rewards of 0. Right would give the agent a reward of 0 as a first reward followed by infinite rewards of 0.

If y=0.9, then the agent would receive the full first reward, then a discounted reward of (0.9)^k x (r). This would result in 1 + 0.9^2 + 0.9^4 … for the left action and 2(0.9)^1 + 2(0.9)^4 + 2(0.9)^6 … for the right action. For left this would be ((1 / (1-y)) / 2) = 5 and for right this would be ((2 / (1-y)) / 2)(y) = 9. Right would be better. Reference 3.10 and 3.9 equations.

If y=0.5 it will be similar to the rewards when y=0.9 but instead will be the value of 0.5 as the discount factor. Right = ((1 / (1-y)) / 2) = 1 and Left = ((2 / (1-y)) / 2)(y) = 1. Both give the same.

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

v\*(s) = q\*(s, v\*(s))

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

Π∗ = argmax of Bellman(v\*s)

probabilities of choosing the next action

choose the next state that has the maximum value

Each optimal policy Π∗ shares the same optimal state-value function, but what exactly does that mean?

I don’t know how to define a policy in mathematical terms.

Chapter 4

Exercise 4.1 In Example 4.1, if π is the equiprobable random policy, what is qπ(11, down)?

What is qπ(7, down)?

Q\_pi (11, down) = R\_down, 11, T + yV\_pi (T) = 1 + 0 = 1

Exercise 4.2 In Example 4.1, suppose a new state 15 is added to the gridworld just below

state 13, and its actions, left, up, right, and down, take the agent to states 12, 13, 14,

and 15, respectively. Assume that the transitions from the original states are unchanged.

What, then, is vπ(15) for the equiprobable random policy? Now suppose the dynamics of

state 13 are also changed, such that action down from state 13 takes the agent to the new

state 15. What is vπ(15) for the equiprobable random policy in this case?

V(15) = Sum\_a (pi(a | 15) sum\_s’\_r (p(s’, r | S\_t = 15, A\_t = a)(r + yV\_k (s’)))

= ¼(R\_left,15,12 + y(V(12)) + R\_right,15,14 + y(V(14)) + R\_down, 15, 15 + y(V(15)) + R\_up, 15, 13 + y(V(13)))

= ¼(-1 - 22y - 1 - 14y - 1 - yV(15) - 1 - 20y)

= -1 - 14y - (y/4)(V(15))

We need to calculate V(13) then we can recalculate V(15) using the new V(13)

V(13) = Sum\_a (pi(a | 13) sum\_s’\_r (p(s’, r | S\_t = 13, A\_t = a)(r + yV\_k (s’)))

= ¼ (R\_left,13,12 + y(V(12)) + R\_right,13,14 + y(V(14)) + R\_down, 13, 15 + y(V(15)) + R\_up, 13, 9 + y(V(9)))

= ¼(-4 + yV(9) + yV(12) + yV(14) + yV(15))

= -1 + (y/4)(-20 - 22 - 14 + yV(15)) = -1 + (y/4)(-56 + yV(15))

V(15) = ¼(R\_left,15,12 + y(V(12)) + R\_right, 15,14 + y(V(14)) + R\_down,15,15 + y(V(15)) + R\_up,15,13 + y(V(13)))

= -1 + y/4(V(13) + V(15) - 14 - 22)

= -1 + y/4(V(13) + V(15)) - 9y

We can solve for V(13) and V(15) with the following :

V(13) - y/4(V(15)) = -1 - 14y

-y/4(V(13)) + (1-(y/4))(V(15)) = -1 - 9y

Exercise 4.5 How would policy iteration be defined for action values? Give a complete

algorithm for computing q∗, analogous to that on page 80 for computing v∗. Please pay

special attention to this exercise, because the ideas involved will be used throughout the

rest of the book.

Initialization:

Initialize Q(s, a) and pi(s) arbitrarily for every s in S and a in A(s)

Policy Evaluation

Loop:

Tri <- 0

Loop for every (s, a) pair:

q <- Q(s, a)

Q(s, a) <- Sum\_s′ ((R\_a,s,s′)(P\_a,s,s′)) + γ Sum\_s’,a′ (Q\_pi(s′, a′)pi(s′, a′) x P\_a,s,s′ ∗

Tri <- max(Tri, |q - Q(s, a)|)

Until Tri < Theta (small positive number)

Exercise 4.7 (programming) Write a program for policy iteration and re-solve Jack’s car

rental problem with the following changes. One of Jack’s employees at the first location

rides a bus home each night and lives near the second location. She is happy to shuttle

one car to the second location for free. Each additional car still costs $2, as do all cars

moved in the other direction. In addition, Jack has limited parking space at each location.

If more than 10 cars are kept overnight at a location (after any moving of cars), then an

additional cost of $4 must be incurred to use a second parking lot (independent of how

many cars are kept there). These sorts of nonlinearities and arbitrary dynamics often

occur in real problems and cannot easily be handled by optimization methods other than

dynamic programming. To check your program, first replicate the results given for the

original problem. If your computer is too slow for the full problem, cut all the numbers

of cars in half.

Answer here.

Exercise 4.8 Why does the optimal policy for the gambler’s problem have such a curious

form? In particular, for capital of 50 it bets it all on one flip, but for capital of 51 it does

not. Why is this a good policy?

It weights the value of state 50 and action of waging 50 credits much higher because it has resulted in successfully reaching 100 credits, and it appears there is no punishment for losing. It doesn’t do that with 51 because going back to 2 is not optimal for finishing in the least amount of states which is encouraged by a discount factor.

Chapter 5

Exercise 5.1 Consider the diagrams on the right in Figure 5.1. Why does the estimated

value function jump up for the last two rows in the rear? Why does it drop off for the

whole last row on the left? Why are the frontmost values higher in the upper diagrams

than in the lower?

The last two rows represent when the player has a relatively high value without busting, like 20 or 21. The dealer also will stay at 17 or higher which contributes to this win probability at 20 or 21. The drop off in the last row on the left is due to an ace showing for the dealer’s hand; thus, the chances of the dealer having a high value without busting is higher and the chances of the player having an ace are reduced. The frontmost values being higher in the upper diagrams is due to the player having a usable ace which increases the likelihood of winning while decreasing the chance that the dealer has an ace.

Exercise 5.2 Suppose every-visit MC was used instead of first-visit MC on the blackjack

task. Would you expect the results to be very different? Why or why not?

I would not expect the results to be very different because it would be very seldom that the same state would come up. And especially that the same state would come up within the same hand. It isn’t impossible, but it is improbable. Because of this the every-visit Monte Carlo would be very similar to the first-visit Monte Carlo.

Exercise 5.3 What is the backup diagram for Monte Carlo estimation of q\_pi?

State->action->state->action->state … action->terminal state.

Exercise 5.5 In learning curves such as those shown in Figure 5.3 error generally decreases

with training, as indeed happened for the ordinary importance-sampling method. But for

the weighted importance-sampling method error first increased and then decreased. Why

do you think this happened?

This is the case due to the weighted importance sampling being biased in the sense of it being an estimate for the behavior policy and not for the optimal target policy. This bias converges asymptotically to zero as more runs and experienced is gained in the behavior policy. This bias causes the rise in error in the beginning of the learning process for this blackjack scenario.

Exercise 5.7 Modify the algorithm for first-visit MC policy evaluation (Section 5.1) to

use the incremental implementation for sample averages described in Section 2.4.

To use the incremental implementation, we only need to modify the method in which the average return is calculated. Instead of appending each return G to an array of returns. We just need to calculate the average at each time step as in the incremental implementation used in the simple bandit algorithm. This would be V(s\_t) <- V(s\_t) + 1/n(G\_t) where G\_t = y(G\_t + R\_t+1)

Chapter 6

Exercise 6.2 This is an exercise to help develop your intuition about why TD methods

are often more effiicient than Monte Carlo methods. Consider the driving home example

and how it is addressed by TD and Monte Carlo methods. Can you imagine a scenario

in which a TD update would be better on average than a Monte Carlo update? Give

an example scenario – a description of past experience and a current state – in which

you would expect the TD update to be better. Here's a hint: Suppose you have lots of

experience driving home from work. Then you move to a new building and a new parking

lot (but you still enter the highway at the same place). Now you are starting to learn

predictions for the new building. Can you see why TD updates are likely to be much

better, at least initially, in this case? Might the same sort of thing happen in the original

task?

Temporal difference learning updates the value for each state immediately following the state. If we have scenarios where the original predicted time for each state varies in being too optimistic and also too pessimistic (that is, it is too high on its predictions on some legs of our drive home and too low on others), then temporal difference will be able to update this appropriately whereas Monte Carlo will shift all of them in the overall direction given the final time when we return home. This is shown in figure 1 in the book. A TD method may be better on average in a situation where we have a variance or change in one of the states, such as changing the initial starting point of driving and keeping the rest of the drive the same (much like the question hint). In Monte Carlo, all of the states will be shifted up or down depending on if this initial location is a longer time or shorter time away from the highway entrance. In TD(0), on the other hand, these states will not all be shifted. Only the first state will experience the shift. The same sort of thing could happen in the original, especially if there is a high amount of variance for one of the states or if we make a very good guess on a state initially. This will result in TD converging much quicker as it would in the scenario posed in the question of changing the initial starting point.

Exercise 6.3 From the results shown in the left graph of the random walk example it appears that the first episode results in a change in only V (A). What does this tell you about what happened on the first episode? Why was only the estimate for this one state changed? By exactly how much was it changed?

It tells us that the first episode went off the left side. We can tell this from the dip in the value of the A state. If the problem is undiscounted and alpha is equal to 0.1 then the update for the value of a state is given by: V(s\_t) <- V(s\_t) + 0.1(r\_t+1 + V(s\_t+1) - V(s\_t)). Since transitions among non-terminal states gives a reward of zero and our value function is initialized as constant across states, the first update results in no change V(s\_t) = V(s\_t). The update for state A is as follows: V(A) <- V(A) + 0.1(0 + 0 - V(A)) = .9(V(A)) = 0.45. This is our value of state A after the first episode. It was changed by .05 after the episode terminated on the left side.

Exercise 6.4 The specific results shown in the right graph of the random walk example are dependent on the value of the step-size parameter, alpha. Do you think the conclusions about which algorithm is better would be affected if a wider range of values were used? Is there a different, fixed value of alpha at which either algorithm would have performed significantly better than shown? Why or why not?

With larger values of alpha, our value function will change much more radically. The value of each state will depend more on the specific returns at each time step in this way. With smaller values of alpha, the specific returns will not matter as much as our value function will change much more slowly. In this way, the value function is much less sensitive to specific random time steps with smaller values of alpha, but it will also converge more slowly to the true value function.

Exercise 6.5 In the right graph of the random walk example, the RMS error of the TD method seems to go down and then up again, particularly at high alpha's. What could have caused this? Do you think this always occurs, or might it be a function of how the approximate value function was initialized?

The question implies that this might have something to do with the approximate value function itialization, but I do not believe that it does. I believe this error rises from the fact that if you have higher alphas then TD learning will learn specific state values base on instances of that state and weighs much more heavily on each particular instance in which that state is observed and the following next state. I believe this is the reason that the RMS error goes down and then back up. In that sense, I guess it could be related to the value function initialization. It could be making the error go down very quckly but then rising because of the alpha value.

Exercise 6.11 Why is Q-learning considered an off-policy control method?

Q-learning is considered off-policy because it does not matter which policy is used as long as each state is visited an infinite number of times in the limit (as is required by most algorithms to produce an optimal policy). Q-learning approximates q\*, the optimal action-value function, regardless of the current policy.

Exercise 6.12 Suppose action selection is greedy. Is Q-learning then exactly the same algorithm as Sarsa? Will they make exactly the same action selections and weight updates?

This is a tough question to answer as I think it comes across as slightly ambiguous. If action selection is greedy in Q-learning, then will all states even be visited continually? This goes for Sarsa as well. If so, then Q-learning will eventually converge to the best action-value function giving an optimal policy that is greedy. This greedy optimal policy should be similar to the policy from Sarsa, but could be different if there are multiple optimal policies. They should have similar weights if they end up on the same optimal policies. The updates will take place differently since Sarsa is an on-policy algorithm that will update the policy being used as it continues and Q-learning is an off-policy algorithm that will not change the policy being used for behavior and updating of the target policy.

Exercise 6.14 Describe how the task of Jack's Car Rental (Example 4.2) could be reformulated in terms of afterstates. Why, in terms of this specific task, would such a reformulation be likely to speed convergence?

In this particular case, we are trying to decide the value of having a certain number of cars at each location given the amount of business for each location, the cost of moving cars, and the cost of keeping cars at a location. It is beneficial to use after-states here since what we are really trying to use as the value here is the potential cost savings and cost production within a specific state instead of the actual return from the state. The reason for this, much like tic tac toe, is depending on the users next action (whether it be renters or the tic tac toe player), the state is rewarded or punished because of this action. The real value of the state should not be determined by the user’s next action because that is a very specific instance of the scenario. Instead, if we want it to generalize well, we would use the potential value of the state that we are in currently after taking the last action before any other input from a user or users.

Chapter 7

Exercise 7.1 In Chapter 6 we noted that the Monte Carlo error can be written as the sum of TD errors (6.6) if the value estimates don't change from step to step. Show that the n-step error used in (7.2) can also be written as a sum TD errors (again if the value estimates don't change) generalizing the earlier result.

If the estimates do not change, then the n-step error can be written as a sum of TD errors. This seems obvious since if the estimates do not change we can take the error for each step and sum them. Or even the error for a number of steps and add them. If the estimates do not change this can be applied to any similar error functions.

Exercise 7.3 Why do you think a larger random walk task (19 states instead of 5) was used in the examples of this chapter? Would a smaller walk have shifted the advantage to a different value of n? How about the change in left-side outcome from 0 to -1 made in the larger walk? Do you think that made any difference in the best value of n?

I think the larger random walk task was used here because if the 5 state random-walk task had been used then n would end up being 1 due to the small nature of the state-space. With a 19 state random-walk task it added much more to the state-space making each episode much longer on average. This gave us a better look into what n might be in these types of situations, and since it isn’t 1 we can look at it as an n-step TD method instead of a TD(0) method which we had already looked at in an earlier chapter. I think shifting the left-side outcome to -1 would encourage the algorithm to take the right action more often than if it were 0, but this would suggest that the random walk terminated more quickly. This would affect the value of n since the episode would be shorter.

Exercise 7.5 Write the pseudocode for the off-policy state-value prediction algorithm described above.

This pseudocode will be very similar to that of the importance sampling pseudocode from the book. The difference being that we will update at a horizon h. If the estimate pt is zero then instead of zero we return that the estimate and target are the same thus it returns 1. We also add a control variate term.

Exercise 7.6 Prove that the control variate in the above equations does not change the expected value of the return.

The expected value of the control variate is zero due to the expected value of the importance sampling ratio being 1. 1- this importance sampling ratio yields 0. 0 x anything is 0.

Chapter 8

Exercise 8.1 The nonplanning method looks particularly poor in Figure 8.3 because it is a one-step method; a method using multi-step bootstrapping would do better. Do you think one of the multi-step bootstrapping methods from Chapter 7 could do as well asthe Dyna method? Explain why or why not.

I would not think that a multi-step bootstrapping method would do as well here. I think it would do better than the nonplanning Dyna, but I do not think it would do as well as the planning Dyna method. The reason for this is due to the Dyna method using simulated experience to improve its policies whereas the multi-step bootstrapping method would either use real experience or some n-steps of probability distribution with updates executed after each individual step or after all of the steps have completed. Either way, most of the multi-step methods are computationally intensive and would only lead to the steps that were observed being updated in each iteration. The Dyna method would update all of the states that were observed in n simulated runs.

Exercise 8.2 Why did the Dyna agent with exploration bonus, Dyna-Q+, perform better in the first phase as well as in the second phase of the blocking and shortcut experiments?

This agent performs better in these two tasks for two reasons that are interrelated. These reasons include being able to receive more reward due to additional reward being given and exploring states that have not been observed in a large period of time for additional reward. This encourages the exploration to find the best path when the environment changes. It also gives higher reward due to giving an additional reward when observing states that have not been observed in a long period of time. The DynaQ+ may also find the optimal route faster initially.

Exercise 8.3 Careful inspection of Figure 8.5 reveals that the difference between Dyna-Q+ and Dyna-Q narrowed slightly over the first part of the experiment. What is the reason for this?

I believe the reason for this would be that the DynaQ+ method has found the optimal route to the goal and the optimal policy and will continue to receive rewards for long unobserved paths, but since it has already found the optimal policy this causes a loss in reward whereas the DynaQ method eventually finds the optimal path as well but does not venture off the same as the DynaQ+ due to not receiving any incentive to do so. DynaQ will continue to receive the optimal path rewards which causes the narrowing in the gap.

Exercise 8.5 How might the tabular Dyna-Q algorithm shown on page 164 be modified to handle stochastic environments? How might this modification perform poorly on changing environments such as considered in this section? How could the algorithm be modified to handle stochastic environments and changing environments?

It would need to have compensation for a stochastic instead of deterministic environment. This would mean computing probabilities and possible actions from the current state to update all of the possible actions and next states instead of just the one state that is chose deterministically. On the other hand, an action could be taken based on a probability and the update on just that action and state.

Exercise 8.6 The analysis above assumed that all of the b possible next states were equally likely to occur. Suppose instead that the distribution was highly skewed, that some of the b states were much more likely to occur than most. Would this strengthen or weaken the case for sample updates over expected updates? Support your answer.

This will make expected updates more favorable than in the case when all states are equally likely to occur. The reason for this is because if only one stat is possible then both expected and sample updates are the same, but as expected updates become closer to having only one possible next state, it gets closer to the sample updates case. That being said, I believe there would still be some advantages to sample updates in computation.

Exercise 8.7 Some of the graphs in Figure 8.8 seem to be scalloped in their early portions, particularly the upper graph for b = 1 and the uniform distribution. Why do you think this is? What aspects of the data shown support your hypothesis?

This is due to the uniform distribution observing all states in an exhaustive fashion which makes it a slow start to performing well but eventually it will perform better in smaller problems than an on-policy observation technique since this will continually look at a collection of states past the point where it is no longer necessary to look at those states. The fact that the uniform distribution does better in the longer run on the smaller problems than the on-policy method and that the on-policy method does faster than the uniform method on the larger problem seems to support this.