

Solutions to exercises of "Reinforcement Learning: An Introduction" (Sutton and Barto)

Jeremy Scheurer

Created in February 2018

Contents

| | | |
|----------|--|----------|
| 1 | The reinforcement learning problem | 2 |
| 1.1 | Self-Play | 2 |
| 1.2 | Symmetries | 2 |
| 1.3 | Greedy Play | 3 |
| 1.4 | Learning from Exploration | 3 |
| 1.5 | Other improvements | 4 |
| 2 | Multi-armed Bandits | 4 |
| 2.1 | Longterm epsilon greedy | 4 |
| 2.2 | Step-size parameters | 4 |
| 2.3 | Optimistic vs Realistic greedy | 5 |
| 2.4 | Programming: Comparison of stepsizes | 5 |
| 2.5 | Spike in UCB plot | 6 |
| 3 | Finite Markov Decision Processes | 7 |
| 3.1 | Reinforcement Learning Applications | 7 |
| 3.2 | Exception Problems | 8 |
| 3.3 | Driving | 8 |
| 3.4 | Pole-Balancing | 9 |
| 3.5 | Maze-robot | 10 |
| 3.6 | Broken Vision System | 10 |
| 3.7 | Bellmann equation | 11 |

1 The reinforcement learning problem

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?

Well for one thing we cannot let the learning rate go to zero, because our opponent is constantly changing(learning) and thus we need to adopt aswell. Before we assumed that our opponent plays imperfectly, if this is still the case for ourself, then exploiting a certain weakness might only work once, because the opponent will learn about it. I guess it would still learn the same policies but as both sides improve in the end they will both have found out the best possible moves and thus nobody will ever win.

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 have not a different value estimation for each possible state the board can be in, but for groups of states. These groups of states each contain states that are symmetric to each other. By doing this we would have less states in total which would mean we have to explore less to visit each state for at least one time. That means we can have faster convergence to an optimal solution.

Well the problem is that if our opponent does not take advantage of symmetries he could be playing very differently in several states that are symmetric. He could always be playing perfectly in one state and imperfectly in the other. We would then not be able to capture this difference, because both states are just one state in our model. So it is not necessarily true that symmetrically equivalent positions should have the same value.

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?

For this question I will assume that we are not using the UCB algorithm or similar algorithms where taking the greedy choice also includes sometimes exploring states. Thus always making the greedy choice obviously neglects exploring other states. So if you are very lucky you might just make all the optimal actions by always choosing the greedy choice. But this is very unlikely, most probably you will never be able to reach the optimal solution. If we compare this with an intelligent non-greedy algorithm which will explore different actions, this greedy-algorithm clearly underperforms.

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?

The problem is that with exploratory actions anything can happen. So if I have a state which is actually the best state to win and then take an exploratory action which turns out to lead to a very bad state, I will then update the previous state into the direction of the new(bad) state. Depending on the learning rate this can lead to updating the previous state more strongly or not. But obviously this could lead to a scenario where the previous state, which was very good, now is considered as a bad state(with obviously no good reason for that, because we were only exploring). So in the first case, where we do not update after exploring, the probabilities we try to approximate are the ones that lead to the most wins. So we try to learn how our opponent plays and exploit its deficiencies. In the second case basically anything could happen and the probabilities cannot be interpreted with sense.

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?*

One important thing was already mentioned, namely to keep the learning-rate from becoming zero. Only then can we always adapt to a changing opponent. One could also play against itself to keep improving and to learn different opposing behaviour. One important thing we also have to introduce is the uncertainty about our state-values. We should also explore actions we are uncertain about, more often (see later UCB algorithm).

2 Multi-armed Bandits

2.1 Longterm epsilon greedy

In the comparison shown in Figure 2.2, which method will perform best in the long run in terms of cumulative reward and cumulative probability of selecting the best action? How much better will it be? Express your answer quantitatively.

The greedy algorithm with $\epsilon = 0.01$ will perform better than the one with $\epsilon = 0.1$ in the long run. The second algorithm will be able to find the best action more quickly but never takes it more than 0.91 % of the time. In contrast the first algorithm will take longer to find the optimal action, but will eventually find it. And once it does, it will take it about 0.99 % of the time. So it will be about 8-9 % better than the second option.

2.2 Step-size parameters

If the step-size parameters α_n , are not constant, then the estimate Q_n 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?

The α 's are not constant but can be any value such as $\alpha = \frac{1}{n}$. Thus we

get the following formula from 2.6 by replacing the alpha's:

$$(1 - \alpha_1)^n Q_1 + \sum_{i=1}^n \alpha_i (1 - \alpha_i)^{n-i} R_i$$

2.3 Optimistic vs Realistic greedy

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?

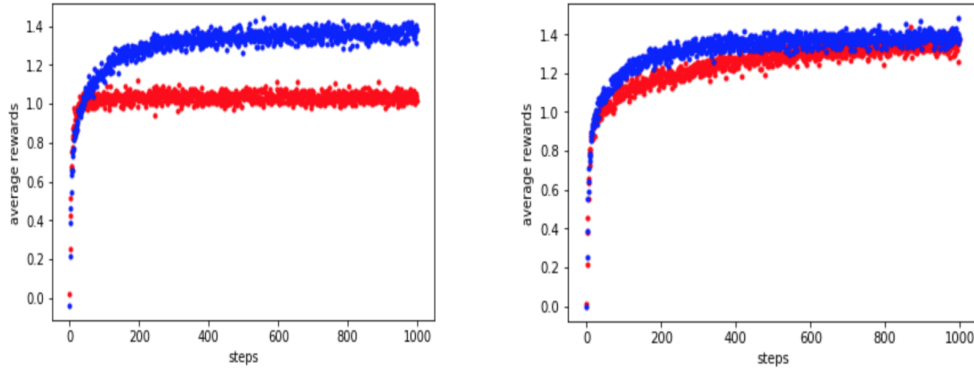
Because compared to the realistic greedy method, in the first n steps each of the n arms is pulled exactly once. That means that after exploring each arm, for one round at least we are very likely to choose the optimal action. This implies that we will definitely have a sharp spike in the optimal action vs steps plot. For comparison, the realistic greedy method explores much less and has much less chance of finding out about the optimal action in the early rounds.

2.4 Programming: Comparison of stepsizes

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. Prepare plots like Figure 2.2 for an action-value method using sample averages, incrementally computed by $\alpha = 1$, and another n action-value method using a constant step-size parameter, $\alpha = 0.1$. Use $\epsilon = 0.1$ and, if necessary, runs longer than 1000 steps.*

Remember the update rule of the iterative sample-average method: New estimate = Old Estimate + stepsize * [old estimate - target]. As seen in the book, the problem about non-constant step-sizes like $\frac{1}{n}$ is that if we have a non-stationary problem, they cannot adapt anymore after a certain time, because we are not updating the old estimate enough anymore (because the stepsize goes to zero).

By running different experiments with 1000 steps and a 1000 repetitions we got the following:



(a) Epsilon greedy algorithm on a stationary problem.
Blue: $\epsilon = 0.1$, Red: $\epsilon = 0.0$

(b) Epsilon greedy algorithm with $\epsilon = 0.1$ on a stationary problem.
Blue: stepsize = $\frac{1}{n}$, Red: stepsize = 0.1

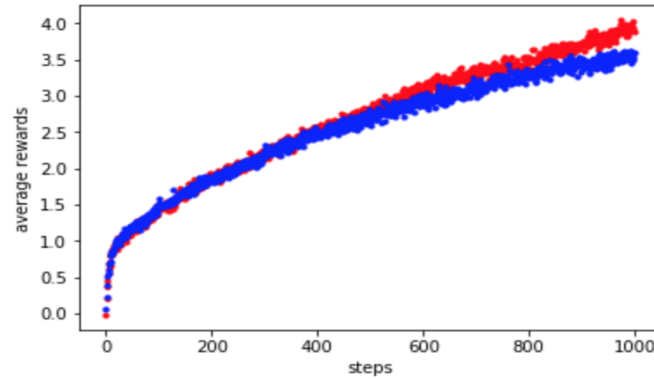


Figure 1: Epsilon greedy algorithm with $\epsilon = 0.1$ on a non-stationary problem.
Blue: stepsize = $\frac{1}{n}$, Red: stepsize = 0.1

For code solutions see Coding/Chapter02/Exercise2.4

2.5 Spike in UCB plot

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 much less prominent.

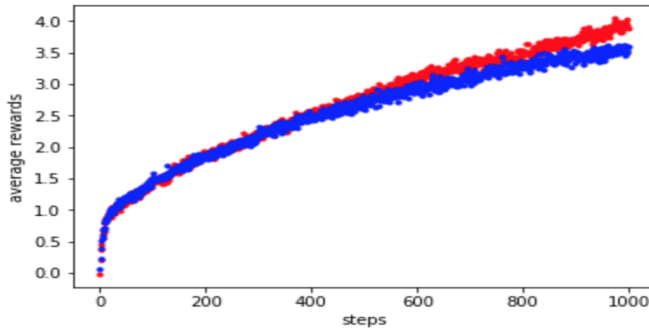


Figure 2: Epsilon greedy algorithm with $\epsilon = 0.1$ on a non-stationary problem.

Blue: stepsize = $\frac{1}{n}$, Red: stepsize = 0.1

I am not so sure about this answer, but I think it is a similar reasoning as my answer in 2.3. If we have a larger c , we explore more. We see that the initial average rewards are very small, thus increasing t will make the confidence bound of the not yet selected arms definitely larger than the already explored arms. This leads to a probable scenario where in the first 10 steps we explore each arm exactly once. So from the eleventh step on we will not consider "bad" arms for a longer period (until t is large enough) and will choose one of the better arms with a high likelihood.

Note to myself: The fishy thing about this idea is that after one round, our confidence about which arm is good or bad is very low. So there is a high likelihood that a "bad" arm yields a good reward and vice versa. This contradicts the argument from above.

3 Finite Markov Decision Processes

3.1 Reinforcement Learning Applications

Devise three example tasks of your own that fit into the reinforcement learning 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.

1. Flying an airplane: The framework would learn how to operate an

airplane by toggling all the buttons etc. The states would be the current setting of all possible buttons etc, and the necessary information for the state of the airplane (like height, speed etc.). The actions as mentioned above is changing the possible buttons and steering the steering wheel. The rewards would be positive for keeping the airplane steady and negative for crashing the airplane or quickly changing the airplane's direction.

2. Recommender System: The framework would learn to recommend personalized news articles, according to ones clicks on articles. The state's would be the different sets of recommended articles. The reward would be positive if the user clicked on any article and negative if he/she didn't. The action is to recommend a set of articles.
3. Traffic Light System: One could use the framework to handle the traffic lights of a city such that the traffic flow is optimal and traffic jams are mostly avoided. This is very feasible, because such a framework could be trained with simulated traffic, which can easily be done. The states would be time intervalls such as 2 seconds. After each such intervall the framework has to take an action. The actions are to change the traffic lights at different locations. Obviously there are lots of traffic lights which can be on or off, thus there are also many actions. The reward is negative the longer the cars have to wait. Thus you could sum up all the waiting time of all the cars. Minimizing their waiting time means thus to maximize the reward.

3.2 Exception Problems

Is the reinforcement learning framework adequate to usefully represent all goal-directed learning tasks? Can you think of any clear exceptions?

It does obviously not work for problems that don't have a clear reward signal. Although for these cases there is some research done into yielding intrinsic curiosity rewards. Unfortunately currently nothing concretely comes to my mind.

3.3 Driving

Consider the problem of driving. You could define the actions in terms of the accelerator, steering wheel, and brake, that is, where your body meets the machine. Or you could define them farther out - say, where the rubber meets

the road, considering your actions to be tire torques. Or you could define them farther in - say, where your brain meets your body, the actions being muscle twitches to control your limbs. Or you could go to a really high level and say that your actions are your choices of where to drive. What is the right level, the right place to draw the line between agent and environment? On what basis is one location of the line to be preferred over another? Is there any fundamental reason for preferring one location over another, or is it a free choice?

I would say the right level to draw the line between agent and environment is outside of the actions with which the agent has most control over the task, but is not helplessly overwhelmed by the choice. So to make it concrete in this case, I guess by defining the actions with the accelerator, steering wheel and brake the agent has most control over the car but not so much that he won't figure out how to steer the car. If you put the boundary at the choice of where to drive, the car won't do anything, so the agent has no real control. If you put the boundary at the twitches of your muscle, you have too much control and the agent might never figure out how to drive the car. Because lots of muscle's are not needed for this task and thus completely irrelevant. So the best choice is a trade-off between control and not having too many options. But that means it always depends on the goal, because only if you know your goal you can figure out what actions are irrelevant to it.

3.4 Pole-Balancing

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?

I somehow find this a tricky question. As described in the pole-balancing example, for a continuous task with discounting, the return at each time would be related to $-\gamma^k$ where k are the number of timesteps before failure. If we now apply this same discounting rate to an episodic task, where every episode is a single timestep, most of the rewards for each timestep would be 0 except for the failure which would be -1. To me it makes sense to look at another example, what if the episodes would be 10 time steps. Then in step i of this episode you would look ahead $10-i$ steps and discount them accordingly. But if no failure occurs in this episode, the sum of the rewards

will always be zero, except if a failure occurs in this time episode. Remember though that we had episodes that were 1 timestep large, this implies that only on a failure, the rewards will be $\gamma \cdot -1$. Obviously this is not very helpful because only when you have a failure you recognize that you will fail (but then it already happened).

3.5 Maze-robot

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 (3.1). 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?

No you have not correctly communicated what you want the agent to do. You want it to escape from the maze as quickly as possible. Thus you will have to enforce a negative reward for each time step that the robot has not escaped the maze. Maximizing this reward means escaping the maze as quickly as possible.

3.6 Broken Vision System

Imagine that you are a vision system. When you are first turned on for the day, an image floods into your camera. You can see lots of things, but not all things. You can't see objects that are occluded, and of course you can't see objects that are behind you. After seeing that first scene, do you have access to the Markov state of the environment? Suppose your camera was broken that day and you received no images at all, all day. Would you have access to the Markov state then?

In the first case, yes you do have access to the Markov state of the environment. No matter how the things that you currently see came into this scene, or how long they are there, you know everything you can about this scene for this moment. The current scene is a summary of everything that has happened so far to this scene. If you received no feed at all you would not

have access to the Markov state of the scene. Because you would know nothing about it and obviously nothing is not a summary of all that has happened before.

3.7 Bellmann equation

What is the Bellman equation for action values, that is, for q_π ? It must give the action value $q_\pi(s, a)$ in terms of the action values, $q_\pi(s', a')$, of possible successors to the state-action pair (s, a) . As a hint, the backup diagram corresponding to this equation is given in Figure 3.4 (right). Show the sequence of equations analogous to (3.12), but for action values.