Reinforcement Learning: An Introduction - Notes

Scott Jeen

March 25, 2021

Chapter 1

Introduction

1.1 Overview

- Supervised learning = learning with labels defined by human; Unsupervised learning = finding patterns in data. Reinforcement learning is a 3rd machine learning paradigm, in which the agent tries to maximise its reward signal.
- Exploration versus exploitation problem - agent wants to do what it has already done to maximise reward by exploitation, but there may be a bigger reward available if it were to explore.
- RL is based on the model of human learning, similar to that of the brain's reward system.

1.2 Elements of Reinforcement Learning

Policy Defines the agent's way of behaving at any given time. It is a mapping from the perceived states of the environment to actions to be taken when in those states.

Reward Signal The reward defines the goal of the reinforcement learning problem. At each time step, the environment sends the RL agent a single number, a *reward*. It is the agent's sole objective to maximise this reward. In a biological system, we might think of rewards as analogous to pain and pleasure. The reward sent at any time depends on the agent's current action and the agent's current state. If my state is hungry and I choose the action of eating, I receive positive reward.

Value function Reward functions indicate what is good in the immediately, but value functions specify what is good in the long run. The value function is the total expected reward an agent is likely to accumulate in the

future, starting from a given state. E.g. a state might always yield a low immediate reward, but is normally followed by a string of states that yield high reward. Or the reverse. Rewards are, in a sense, primary, whereas values, as predictions of rewards, are secondary. Without rewards there could be no value. Nevertheless it is values with which we are most concerned when evaluating decisions. We seek actions that bring the highest value, not the highest reward, because they obtain the greatest amount of reward over the long run. Estimating values is not trivial, and efficiently and accurately estimating them is the core of RL.

Model of environment (optionally) Something that mimics the behaviour of the true environment, to allow inferences to be made about how the environment will behave. Given a state and action, the model might predict the resultant next state and next reward. They are used for *planning*, that is, deciding on a course of action by considering possible future situations before they are actually experienced.

Chapter 2

Multi-arm Bandits

RL evaluates the actions taken rather than instructs correct actions like other forms of learning.

2.1 An n-Armed Bandit Problem

The problem:

- You are faced repeatedly with a choice of n actions.
- After each choice, you receive a reward from a stationary probability distribution.
- Objective is to maximise total reward over some time period, say 100 time steps.
- Named after of slot machine (one-armed bandit problem), but n levers instead of 1.
- Each action has an expected or mean reward based on it's prob distribution. We shall call that the *value* of the action. We do not know these values with certainty.
- Because of this uncertainty, there is always an exploration vs exploitation problem. We always have one action that we deem to be most valuable at any instant, but it is highly likely, at least initially, that there are actions we are yet to explore that are more valuable.

2.2 Action-Value Methods

The estimated action value

$$Q_t(a) = \frac{R_1 + R_2 + \cdots + R_{N_t(a)}}{N_t(a)} \tag{2.1}$$

The true value (mean reward) of an action is q(a), but the estimated value at the *t*th time-step is Q(a), given by Equation 2.1 (our estimate after N selections of an action yielding N rewards).

The greedy action selection method:

$$A_t = \underset{a}{\operatorname{argmax}}\, Q_t(a) \tag{2.2}$$

- Simplest action selection rule is to select the action with the highest estimated value.
- Argmax a means the value of a for which Qt is maximised.
- e-greedy methods are where the agent selects the greedy option most of the time, and selects the non-optimal action with probability e.
- Three algorithms are tried: one with e=0 (pure greedy), one with e=0.01 and another with e=0.1
- Greedy method gets stuck performing sub-optimal actions.
- e=0.1 explores more and usually finds the optimal action earlier, but never selects it more that 91/
- e=0.01 method improves more slowly, but eventually performs better than the e=0.1 method on both performance measures.
- It is possible to reduce e over time to try to get the best of both high and low values.

2.3 Incremental Implementation

The sample-average technique used to estimate action-values above has a problem: memory and computation requirements grow over time. This isn't necessary we can devise an incremental solution instead.

$$\begin{aligned} Q_{k+1} &= \frac{1}{k}\sum_{i}^{k} R_i \\ &= \frac{1}{k}\left(R_k + \sum_{i=1}^{k-1} R_i\right) \\ &= \vdots \qquad\qquad = Q_k + \frac{1}{k}\left[R_k - Q_k\right] \end{aligned} \tag{2.3}$$

We are effectively updating our estimate of Qk (Qk+1) by adding a discounted version of the error between the reward just received and our estimate for that reward Qk.

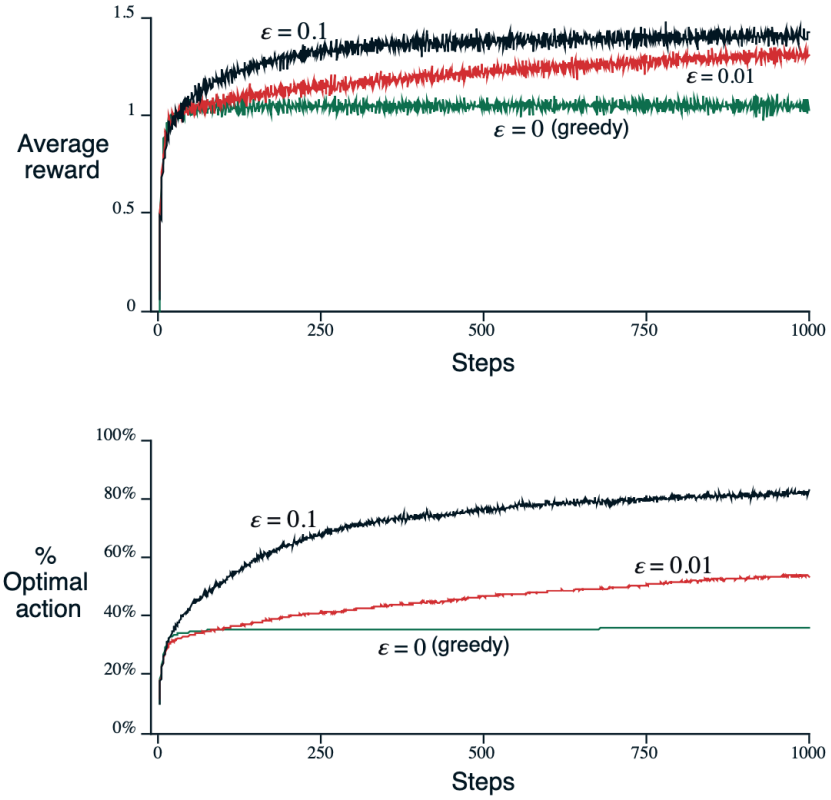

Figure 2.1: tbc

$$NewEstimate \leftarrow OldEstimate + StepSize\,[Target - OldEstimate] \quad (2.4)$$

α is used to denote the stepsize (1/k) in the rest of this book.

2.4 Tracking a Nonstationary Problem

- The averaging methods discussed above do not work if the bandit is changing over time. In such cases it makes sense to weight recent rewards higher than long-past ones. The popular way of doing this is using a constant step-size parameter.

$$Q_{k+1} = Q_k + \alpha\,[R_k - Q_k] \quad (2.5)$$

where the step-size parameter $\alpha \in (0, 1]$) is constant. This results in Q_{k+1} being a weighted average of the past rewards and the initial estimate Q_1:

$$\begin{aligned} Q_{k+1} &= Q_k + \alpha \left[R_k - Q_k\right] \\ &= \alpha R_k + (1-\alpha) Q_k \\ &= \alpha R_k + (1-\alpha)[\alpha R_{k-1} + (1-\alpha) Q_{k-1}] \\ &= \alpha R_k + (1-\alpha)\alpha R_{k-1} + (1-\alpha)^2 Q_{k-1} \\ &= \vdots \\ &= (1-\alpha)^k Q_1 + \sum_i^k \alpha (1-\alpha)^{k-i} R_i \end{aligned} \tag{2.6}$$

$$\tag{2.7}$$

- Because the weight given to each reward depends on how many rewards ago it was observed, we can see that more recent rewards are given more weight. Note the weights α sum to 1 here, ensuring it is indeed a weighted average where more weight is allocated to recent rewards.
- In fact, the weight given to each reward decays exponentially into the past. This sometimes called an *exponential* or *recency-weighted* average.

2.5 Optimistic Initial Values

- The methods discussed so far are dependent to some extent on the initial action-value estimate i.e. they are biased by their initial estimates.
- For methods with constant α) this bias is permanent.
- In effect, the initial estimates become a set of parameters for the model that must be picked by the user.
- In the above problem, by setting initial values to +5 rather than 0 we encourage exploration, even in the greedy case. The agent will almost always be disappointed with it's samples because they are less than the initial estimate and so will explore elsewhere until the values converge.
- The above method of exploration is called *Optimistic Initial Values*.

2.6 Upper-confidence-bound Action Selection

e-greedy action selection forces the agent to explore new actions, but it does so indiscriminately. It would be better to select among non-greedy actions according to their potential for actually being optimal, taking into account

both how close their estimates are to being maximal and the uncertainty in those estimates. One way to do this is to select actions as:

$$A_t = \underset{a}{\text{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{2.8}$$

where $\ln t$ is the number e (2.72) would need to be raised to in order to equal t, and c ¿ 0 controls the degree of exploration.

- The square root term is a measure of the uncertainity in our estimate, and, since it aded to our estimated value, this acts as an 'Upper Confidence Bound' of our estimate.
- For every timestep t, the numerator of the square root increases (first quickly and then more slowly like the ln n function) so our uncertainty increases.
- Each time we select an action our uncertainty decreases because N is the denominator of this equation.
- UCB will often perform better than e-greedy methods

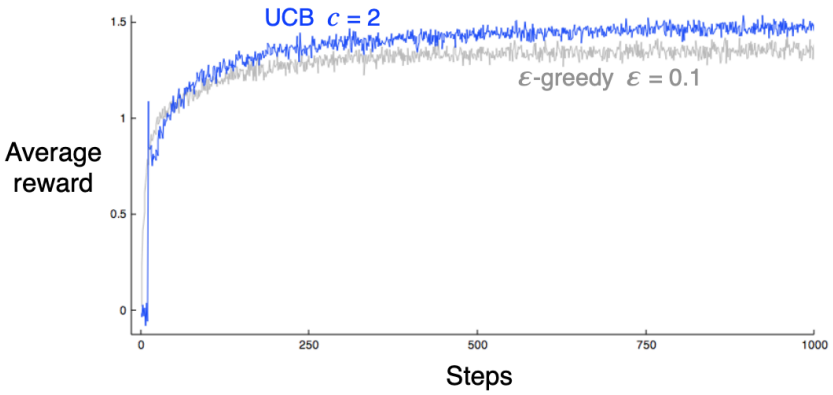

Figure 2.2: UCB performs better than e-greedy in the n-armed bandit problem

2.7 Key Takeaways

2.8 Exercises

Chapter 3

Finite Markov Decision Processes

3.1 The Agent-Environment Interface

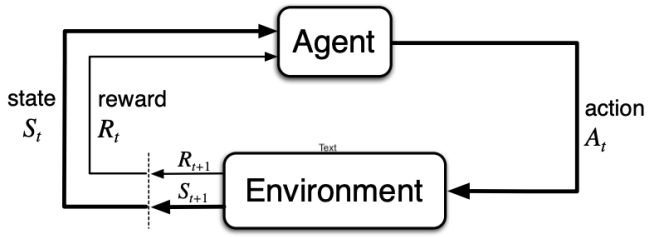

Figure 3.1: The agent-environment interface in reinforcement learning

- At each timestep the agent implements a mapping from states to probabilties of selecting a possible action. The mapping is called the agents *policy,* denoted PI, where PI(a — s) is the probability of the agent selecting actions a in states.
- In general, actions can be any decision we ant to learn how to make, and states can be any interpretation of the world that might inform those actions.
- The boundary between agent and environment is much closer to the agent that is first intuitive. Eg if we are controlling a robot, the voltages or stresses in its structure are part of the environment, not the agent. Indeed reward signals are part of the environment, despite very possibly being produced by the agent e.g. dopamine.

3.2 Goals and Rewards

The *reward hypothesis*:

> All we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)

The reward signal is our way of communicating to the agent what we want to achieve not how we want to achieve it.

3.3 Returns

The return G_t is the sum of future rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_t \tag{3.1}$$

- This approach makes sense in applications that finish, or are periodic. That is, the agent-environment interaction breaks into *episodes*.
- We call these systems *episodic tasks*. e.g playing a board game, trips through a maze etc.
- The opposite, continuous applications are called *continuing tasks.*
- For these tasks we use *discounted returns* to avoid a sum of returns going to infinity.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.2}$$

3.4 Unified Notation for Episodic and Continuing Tasks

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{3.3}$$

3.5 The Markov Property

A state signal that succeeds in retaining all relevant information about the past is *Markov*. Examples include:

- A cannonball with known position, velocity and acceleration
- All positions of chess pieces on a chess board.

In normal causal processes, we would think that our expectation of the state and reward at the next timestep is a function of all previous states, rewards and actions, as follows:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{3.4}$$

If the state is Markov, however, then the state and reward right now completely characterizes the history, and the above can be reduced to:

$$p(s', r|s, a) = Pr\{R_{r+1} = r, S_{t+1} = s' | S_t, A_t\} \tag{3.5}$$

- Even for non-Markov states, it is appropriate to think of all states as at least an approximation of a Markov state.
- Markov property is important in RL because decisions and values are assumed to be a function only of the current state.
- Most real scenarios are unlikely to be Markov. In the example of controlling HVAC, the HVAC motor might heat up which affects cooling power and we may not be tracking that temperature. It is hard for a process to be Markov without sensing all possible variables.

3.6 Markov Decision Process (MDP)

Given any state and action s and a, the probability of each possible pair of next state and reward, s', r is denoted:

$$p(s', r|s, a) = Pr\{R_{r+1} = r, S_{t+1} = s' | S_t, A_t\} \tag{3.6}$$

We can think of $p(s', r|s, a)$ as the dynamics of our MDP, often called the *transition function*–it defines how we move from state to state given actions.

3.7 Value Functions

- Value functions are functions of states or functions of state-value pairs.
- They estimate how good it is to be in a given state, or how good it is to perform a given action in a given state.
- Given future rewards are dependent on future actions, value functions are defined with respect to particular policies.
- For MDPs, we can define nu-pi(s) formally as:

$$\nu_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] \tag{3.7}$$

i.e. the expected future rewards, given state S_t, and policy π. We call $\nu_\pi(s)$the **state value function for policy** pi. Similarly, we can define the value of taking action a in state s under policy π as:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \tag{3.8}$$

i.e. the expected value, taking action a in state s then following policy π.

- We call q_π the **action-value function for policy** π
- Both value functions are estimated from experience.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return. This recursive relationship is characterised by the **Bellman Equation**:

$$\nu_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma \nu_\pi(s') \right] \tag{3.9}$$

This recursion looks from one state through to all possible next states given our policy and the dynamics as suggested by 3.2:

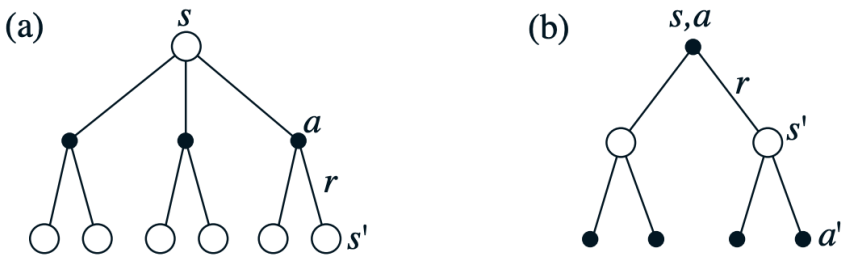

Figure 3.2: Backup diagrams for ν_π and q_π

3.8 Optimal Value Functions

- A policy π' is defined as better than policy pi if its expected return is higher for all states.

- There is always AT LEAST one policy that is better than or equal to all other policies - this is the *optimal policy.*
- Optimal policies are denoted $\pi*$
- Optimal state-value functions are denoted $\nu*$
- Optimal action-value functions are denoted $q*$
- We can write $q*$ in terms of $\nu*$:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma\nu_*(S_{t+1})|S_t = s, A_t = a\right] \tag{3.10}$$

We can adapt the Bellman equation to achieve the Bellman optimality equation, which takes two forms. Firstly for ν_*:

$$\nu_*(s) = \max_{a\in\mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)\left[r + \gamma\nu_*(s')\right] \tag{3.11}$$

and secondly for q_*:

$$q_*(s) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right] \tag{3.12}$$

- Using $\nu*$ the optimal expected long term return is turned into a quantity that is immediately available for each state. Hence a one-step-ahead search, acting greedily, yield the optimal long-term actions.
- Fully solving the Bellman optimality equations can be hugely expensive, especially if the number of states is huge, as is the case with most interesting problems.

3.9 Optimality and Approximation

- We must approximate because calculation of optimality is too computationally intense.
- A nice way of doing this, is allowing the agent to make sub-optimal decisions in scenarios it has low probability of encountering. This is a trade off for being optimal in situations that occur frequently.

3.10 Key Takeaways