Reinforcement Learning: An Introduction - Notes

Scott Jeen

April 16, 2021

Contents

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.

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

$$Q_t(a) = \frac{R_1 + R_2 + \cdots + R_{N_t(a)}}{N_t(a)} \tag{1}$$

The true value (mean reward) of an action is q, but the estimated value at the tth time-step is Q(a), given by Equation 1 (our estimate after N selections of an action yielding N rewards).

The greedy action selection method:

$$A_t = \underset{a}{\operatorname{argmax}}\, Q_t(a) \tag{2}$$

- Simplest action selection rule is to select the action with the highest estimated value.
- Argmax a means the value of a for which Q_t is maximised.
- ϵ-greedy methods are where the agent selects the greedy option most of the time, and selects a random action with probability ϵ.
- 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% of the time.
- 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.

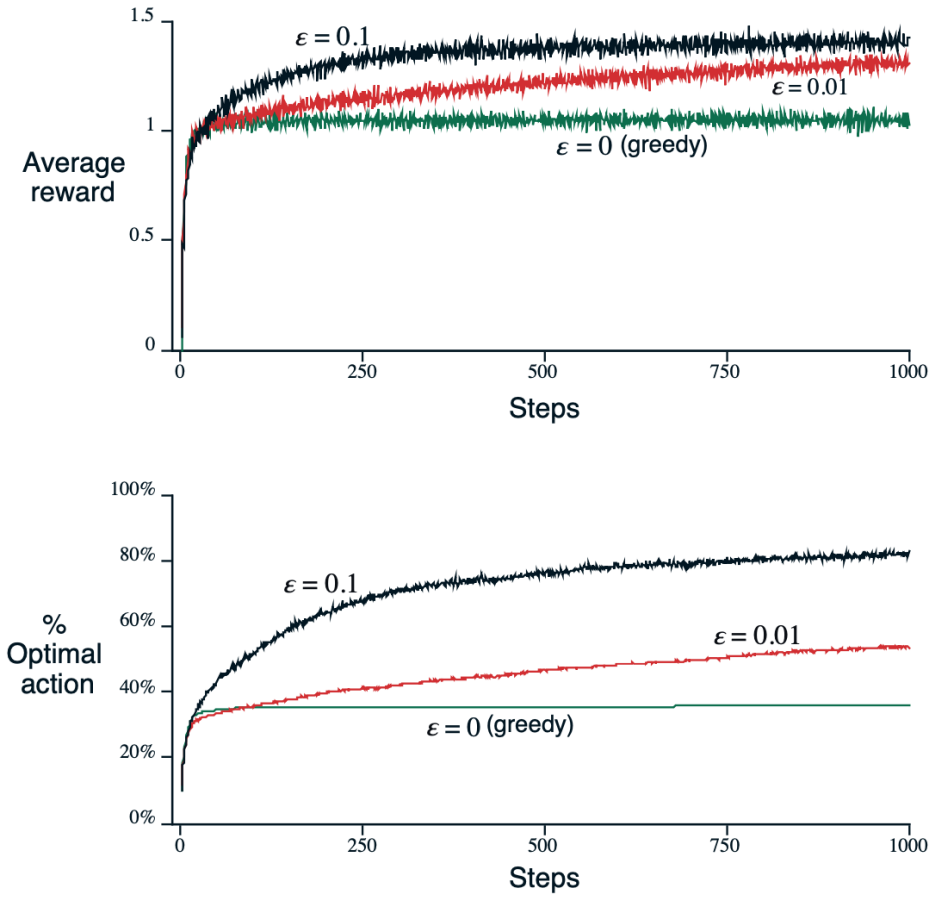

Figure 1: tbc

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:

$$\begin{aligned} Q_{k+1} &= \frac{1}{k}\sum_{i}^{k} R_i \\ &= \frac{1}{k}\left(R_k + \sum_{i=1}^{k-1} R_i\right) \\ &= \vdots \qquad (3) \\ &= Q_k + \frac{1}{k}\left[R_k - Q_k\right] \qquad (4) \\ & \qquad (5) \end{aligned}$$

We are updating our estimate of Q_{k+1} by adding the discounted error between the reward just received and our estimate for that reward Q_k.

$$NewEstimate \leftarrow OldEstimate + StepSize\,[Target - OldEstimate] \tag{6}$$

α is used to denote the stepsize ($\frac{1}{k}$) 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 by using a constant step-size parameter.

$$Q_{k+1} = Q_k + \alpha\,[R_k - Q_k] \tag{7}$$

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:

$$\begin{aligned} Q_{k+1} &= Q_k + \alpha\,[R_k - Q_k] \\ &= \alpha R_k + (1-\alpha)Q_k \\ &= \alpha R_k + (1-\alpha)[\alpha R_{k-1} + (1-\alpha)Q_{k-1}] \\ &= \alpha R_k + (1-\alpha)\alpha R_{k-1} + (1-\alpha)^2 Q_{k-1} \\ &= \vdots \\ &= (1-\alpha)^k Q_1 + \sum_i^k \alpha(1-\alpha)^{k-i} R_i \end{aligned} \tag{8}$$

$$\tag{9}$$

- 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

ϵ-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:

$$A_t = \underset{a}{\text{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{10}$$

where c > 0 controls the degree of exploration.

- The square root term is a measure of the uncertainity in our estimate. It is proportional to t i.e. how many timesteps have passed and inversely proportional to $N_t(a)$ i.e. how many times that action has been visited. The more time has passed, and the less we have sampled an action, the higher our upper-confidence-bound.
- As the timesteps increases, the denominator dominates the numerator as the ln term flattens.
- 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

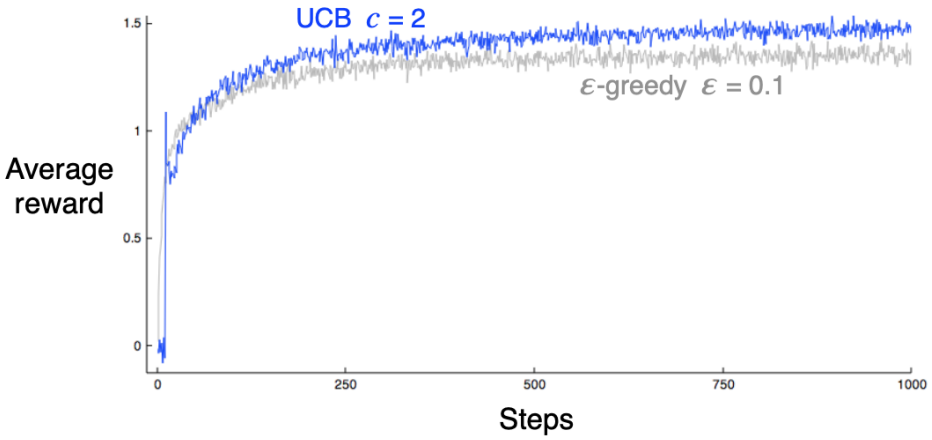

Figure 2: UCB performs better than e-greedy in the n-armed bandit problem

2.7 Associative Search (Contextual Bandits)

- Thus far we have been discussing the stationary k-armed bandit problem, where the value of each arm is unknown but nonetheless remains stationary. Now, we consider a problem where the task could change from step to step, but the value distributions of the arms in each task remain the same. This is called contextual bandits, and in the toy example we are usually given a hint that the task has changed e.g. the slot machine changes colour for each task.

- Now we want to learn the correct action to take in a particular setting, given the task colour observed. This is an intermediary between the stationary problem and the full reinforcement learning problem. See exercise 2.10 below.

2.8 Key Takeaways

- The value of an action can be summarised by $Q_t(a)$, the sample average return from an action
- When selecting an action, it is preferable to maintain exploration, rather than only selecting the action we believe to be most valuable at any given timestep, to ensure we continue to improve our best estimate of the optimal action. We do so using ϵ)-greedy policies.
- If our problem is non-stationary, rather than taking a standard average of every return received after an action, we can take a weighted average that gives higher value to more recently acquired rewards. We call this an *exponential* or *recency-weighted* average.
- Optimistic initial values encourage lots of early exploration as our returns always decrease our estimate of Q_t meaning the greedy actions remain exploratory. Only useful for stationary problems.
- ϵ-greedy policies can be adapted to give more value to actions that have been selected less-often, i.e. actions where we have higher uncertainty in their value, using *upper-confidence-bound* action selection.
- Lastly, each of these techniques have varied performance on the n-armed bandit test dependent on their parametrisation. Their performance is plotted in Figure 3.

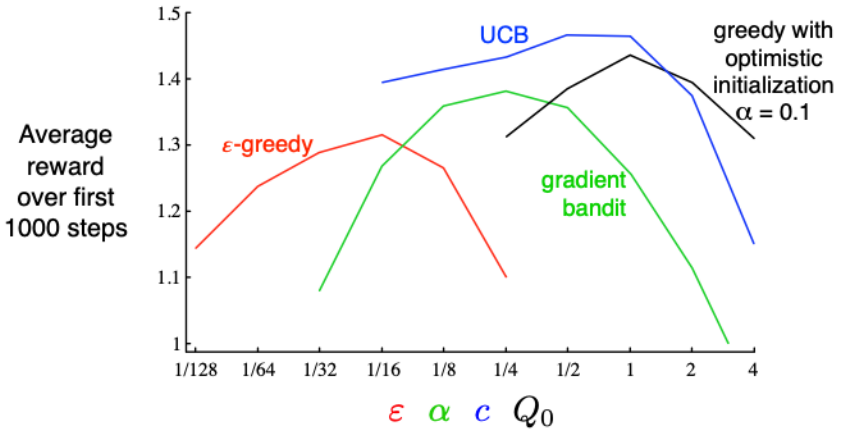

Figure 3: Performance of each of the bandit algorithms explored in this chapter

3 Finite Markov Decision Processes

3.1 The Agent-Environment Interface

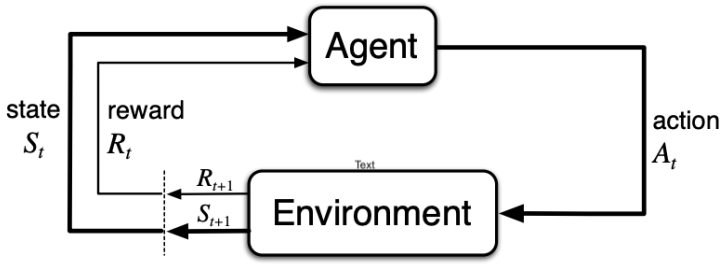

Figure 4: The agent-environment interface in reinforcement learning

- At each timestep the agent implements a mapping from states to probabilities of selecting a possible action. The mapping is called the agents *policy*, denoted π, where $\pi(a|s)$ is the probability of the agent selecting actions a in states.
- In general, actions can be any decision we want 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 than is first intuitive. E.g. 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 and Episodes

The return G_t is the sum of future rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_t \tag{11}$$

- 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.
- Notation for state space in an episodic task varies from the conventional case ($s \in \mathcal{S}$) to ($s \in \mathcal{S}^+$)

- The opposite, continuous applications are called *continuing tasks*.
- For these tasks we use *discounted returns* to avoid a sum of returns going to infinity.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{12}$$

If the reward is a constant + 1 at each timestep, cumulative discounted reward G_t becomes:

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{13}$$

Discounting is a crucial topic in RL. It allows us to store a finite value of any state (summarised by its expected cumulative reward) for continuous tasks, where the non-discounted value would run to infinity.

3.4 Unified Notation for Episodic and Continuing Tasks

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{14}$$

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:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{15}$$

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

$$p(s', r|s, a) = Pr\{R_{r+1} = r, S_{t+1} = s' | S_t, A_t\} \tag{16}$$

- 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:

$$p(s', r|s, a) = Pr\{R_{r+1} = r, S_{t+1} = s'|S_t, A_t\} \tag{17}$$

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 Policies and 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 as the value of a state depends on the action an agent takes in said state.
- A *policy* is a mapping from states to probabilities of selecting each possible action.
- RL methods specify how the agent's policy changes as a result of its experience.
- For MDPs, we can define nu-pi(s) formally as:

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t|S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{18}$$

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

$$q_\pi(s, a) = \mathbb{E}_\pi\left[G_t|S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{19}$$

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**:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \tag{20}$$

This recursion looks from one state through to all possible next states given our policy and the dynamics as suggested by 5:

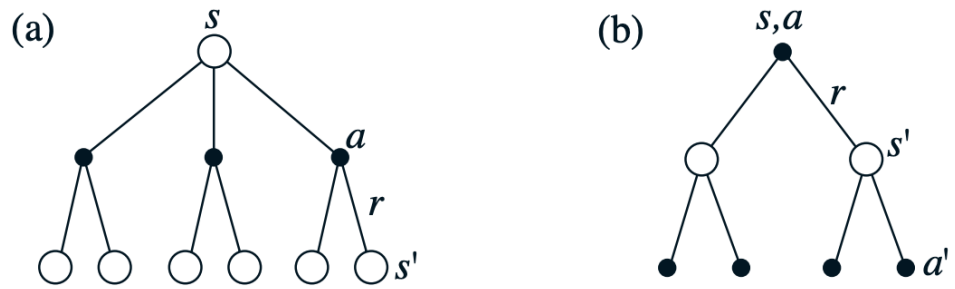

Figure 5: Backup diagrams for v_π and q_π

3.8 Optimal Policies and 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 $v*$
- Optimal action-value functions are denoted $q*$
- We can write $q*$ in terms of $v*$:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \tag{21}$$

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

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_*(s')\right] \tag{22}$$

and secondly for q_*:

$$q_*(s) = \sum_{s',r} p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \tag{23}$$

- Using $v*$ 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.
- Solving the Bellman optimality equation is akin to exhaustive search. We play out *every* possible scenario until the terminal state and collect their expected reward. Our policy then defines the action that maximises this expected reward.
- In the continuous case the Bellman optimality equation is unsolvable as the recursion on the next state's value function would never end.

3.9 Optimality and Approximation

- We must approximate because calculation of optimality is too expensive.
- 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

- We summarise our goal for the agent as a *reward*; its objective is to maximise the cumulative sum of future rewards
- For episodic tasks, returns terminate (and are backpropogated) when the episode ends. For the continuous control case, returns are discounted so they do not run to infinity.
- A state signal that succeeds in retaining all relevant information about the past is *Markov*.
- Markov Decision Processes (MDPs) are the mathematically idealised version of the RL problem. They have system dynamics: $p(s', r|s, a) = Pr\{R_{r+1} = r, S_{t+1} = s'|S_t, A_t\}$
- Policies are a (probabilistic) mapping from states to actions.
- Value functions estimate how good it is for an agent to be in a state (v_π) or to take an action from a state (q_π). They are always defined w.r.t policies as the value of a state depends on the policy one takes in that state. Value functions are the e*expected cumulative sum of future rewards* from a state or state-action pair.
- Knowing our policy and system dynamics, we can define the state value function is defined by the Bellman equation: $v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) [r + \gamma v_\pi(s')]$
- An optimal policy (π_*) is the policy that maximises expected cumulative reward from all states. From the optimal policy we can derive the optimal value functions q_* and v_*.

4 Dynamic Programming

Dynamic Programming (DP) refers to the collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP. DP can rarely be used in practice because of their great cost, but are nonetheless important theoretically as all other approaches to computing the value function are, in effect, approximations of DP. DP algorithms are obtained by turning the Bellman equations into assignments, that is, into update rules for improving approximations of the desired value functions.

4.1 Policy Evaluation (Prediction)

We know from Chapter 3 that the value function can be represented as follows:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \tag{24}$$

If the dynamics are known perfectly, this becomes a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns, where the unknowns are $v_\pi(s), s \in \mathcal{S}$. If we consider an iterative sequence of value function approximations $v_0, v_1, v_2, \ldots$, with initial approximation v_0 chosen arbitrarily e.g. $v_k(s) = 0 \, \forall s$ (ensuring terminal state = 0). We can update it using the Bellman equation:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right] \tag{25}$$

Eventually this update will converge when $v_k = v_\pi$ after infinite sweeps of the state-space, the value function for our policy. This algorithm is called *iterative policy evaluation*. We call this update an *expected update* because it is based on the expectation over all possible next states, rather than a sample of reward/value from the next state. We think of the updates occurring through *sweeps* of state space.

4.2 Policy Improvement

We can obtain a value function for an arbitrary policy π as per the policy evaluation algorithm discussed above. We may then want to know if there is a policy π' that is better than our current policy. A way of evaluating this is by taking a new action a in state s that is not in our current policy, running our policy thereafter and seeing how the value function changes. Formally that looks like:

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \tag{26}$$

Note the mixing of action-value and state-value functions. If taking this new action in state s produces a value function that is greater than or equal to the previous value function for all states then we say the policy π' is an improvement over π:

$$v_{\pi'}(s) \geq v_\pi \forall s \in \mathcal{S} \tag{27}$$

This is known as the *policy improvement theorem*. Critically, the value function must be greater than the previous value function for all states. One way of choosing new actions for policy improvement is by acting greedily w.r.t the value function. Acting greedily will always produce a new policy $\pi' \geq \pi$, but it is not necessarily the optimal policy immediately.

4.3 Policy Iteration

By flipping between policy evaluation and improvement we can achieve a sequence of monotonically increasing policies and value functions. The algorithm is roughly:

1. Evaluate policy π to obtain value function V_π
2. Improve policy π by acting greedily with respect to V_π to obtain new policy π'
3. Evaluate new policy π' to obtain new value function $V_{\pi'}$
4. Repeat until new policy is no longer better than the old policy, at which point we have obtained the optimal policy. (Only for finite MDPs)

This process can be illustrated as:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

Figure 6: Iterative policy evaluation and improvement

4.4 Value Iteration

Above, we discussed policy iteration which requires full policy evaluation at each iteration step, an often expensive process which (formally) requires infinite sweeps of the state space to approach the true value function. In value iteration, the policy evaluation is stopped after one visit to each $s \in \mathcal{S}$, or one *sweep* of the state space. Value iteration is achieved by turning the Bellman optimality equation into an update rule:

$$v_{k+1}(s) = \operatorname*{argmax}_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right] \tag{28}$$

for all $s \in \mathcal{S}$. Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement.

4.5 Asynchronous Dynamic Programming

Each of the above methods has required full sweeps of the state space to first evaluate a policy then improve it. Asynchronous dynamic programming does not require us to evaluate the entire state space each sweep. Instead, we can perform in place updates to our value function as they are experienced, focusing only on the most relevant states to our problem initially, then working to less relevant states elsewhere. This can mean that our agent can learn to act well more quickly and save optimality for later.

4.6 Generalised Policy Iteration

Generalised Policy Iteration is the process of letting policy evaluation and policy improvement interact, independent of granularity. That is to say, improvement/evaluation can be performed by doing complete sweeps of the state space, or it can be performed after every visit to a state (as is the case with value iteration). The level of granularity is independent of our final outcome: convergence to the optimal policy and optimal value function. This process can be illustrated as two convergening lines - Figure 7. We can see that policy improvement and policy evaluation work both in opposition and in cooperation - each time we act greedily we get further away from our true value function; and each time we evaluate our value function our policy is likely no longer greedy.

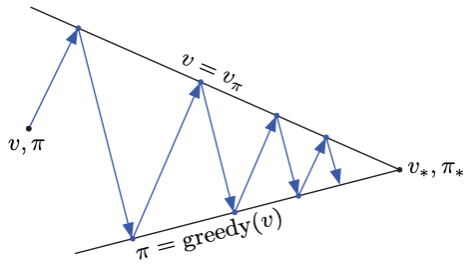

Figure 7: Generalised policy iteration leading to optimality

4.7 Key Takeaways

- *Policy evaluation* is the iterative computation of a value function for a given policy. The value function is only truly evaluated in the limit.
- *Policy improvement* is the computation of an improved policy given the value function for an existing policy.
- By combining the above we achieve *Policy Iteration* and by performing each directly on the value function at each visit to a state we achieve *Value iteration* - both of which can be used to obtain optimal value functions and policies given complete knowledge of a finite Markov Decision Process.
- *Generalised policy iteration* is the process of performing policy evaluation and improvement regardless of granularity - leading to convergence to the optimal value functions.
- DP methods operate in sweeps through the state space performing an *expected update* operation at each state. These updates are based on expected values at all subsequent states and their probabilities of occurring. In general this idea is known as *bootstrapping*, many RL methods bootstrap to update current beliefs from past beliefs.

5 Monte Carlo Methods

If we do not have knowledge of the transition probabilities (model of the environment) then we must learn directly from experience. To do so, we use Monte Carlo methods. Monte carlo methods are most effective in episodic tasks where there is a terminal state and the value of the states visited en route to the terminal state can be updated based on the reward received at the terminal state. We use general policy iteration as outlined in chapter 4, but this time instead of *computing* the value function we learn it from samples. We first consider the prediction problem to obtain v_π and/or q_π for a fixed policy, then we look to improve using policy improvement, then we use it for control.

5.1 Monte Carlo Policy Prediction

- Recall that the value of a state is the expected discounted future reward from that state. One way of estimating that value is by observing the rewards obtained after visiting the state, we would expect that in the limit this would converge toward the true value.
- We can therefore run a policy in an environment for an episode. When the episode ends, we receive a reward and we assign that reward to each of the states visited en route to the terminal state.
- Where DP algorithms perform one-step predictions to *every* possible next state; monte-carlo methods only sample one trajectory/episode. This can be summarised in a new backup diagram as follows:

Figure 8: Monte carlo backup diagram for one episode

- Importantly, monte carlo methods do not bootstrap in the same way DP methods do. They take the reward at the end of an episode, rather than estimated reward based on the value of the next state.
- Because of the lack of bootstrapping, this expense of estimating the value of one state is independent of the number of states, unlike DP. A significant advantage, in addition to the other advantages of being able to learn from experience without a model or from simulated experience.

5.2 Monte Carlo Estimation of Action Values

- With a model we only need to estimate the state value function v as, paired with our model, we can evaluate the rewards and next states for each of our actions and pick the best one.
- With model free methods we need to estimate the state-action value function q as we must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. (If we only have the values of states, and don't know how states are linked through a model, then selecting the optimal action is impossible)
- One serious complication arises when we do not visit every state, as can be the case if our policy is deterministic. If we do not visit states then we do not observe returns from these states and cannot estimate their value. We therefore need to *maintain exploration* of the state space. One way of doing so is stochastically selected a state-action pair to start an episode, giving every state-action pair a non-zero probability of being selected. In this case, we are said to be utilising ***exploring starts***.
- Exploring starts falls down when learning from real experience because we cannot guarantee that we start in a new state-action pair in the real world.
- An alternative approach is to use stochastic policies that have non-zero probability of selecting each state-action pair.

5.3 Monte Carlo Control

- Much like we did with value iteration, we do not need to fully evaluate the value function for a given policy in monte carlo control. Instead we can merely *move* the value toward the correct value and then switch to policy improvement thereafter. It is natural to do this episodically i.e. evaluate the policy using one episode of experience, then act greedily w.r.t the previous value function to improve the policy in the next episode.
- If we use a deterministic policy for control, we must use exploring starts to ensure sufficient exploration. This creates the *Monte Carlo ES* algorithm.

5.4 Monte Carlo Control without Exploring Starts

- To avoid having to use exploring starts we can use either *on-policy* or *off-policy* methods. The only way to ensure we visit everything is to visit them directly.
- On-policy methods attempt to improve or evaluate or improve the policy that is making decisions.
- On-policy control methods are generally *soft* meaning that they assign non-zero probability to each possible action in a state e.g. e-greedy policies.
- We take actions in the environment using e-greedy policy, after each episode we back propagate the rewards to obtain the value function for our e-greedy policy. Then we perform policy improvement by updating our policy to take the **new** greedy reward in each state. Note: based on our new value function, the new greedy action may have changed in some states. Then we perform policy evaluation using our new e-greedy policy and repeat (as per generalised policy iteration).

- The idea of on-policy Monte Carlo control is still that of GPI. We use first visit MC methods to estimate the action-value function i.e. to do policy evaluation, but we cannot then make improve our policy merely by acting greedily w.r.t our value function because that would prevent further exploration of non-greedy actions. We must maintain exploration and so improve the ϵ-greedy version of our policy. That is to say, when we find the greedy action (the action that maximises our reward for our given value function) we assign it probability $1 - \epsilon + \frac{\epsilon}{\mathcal{A}(S_t)}$ of being selected so that the policy remains stochastic.

- Note: doing the above will only find us the best policy amongst the ϵ-soft policies, which may not be the optimal policy for the environment, but it does allow us to remove the need for exploratory starts.

5.5 Off-policy Prediction via Importance Sampling

We face a dilemma when learning control: we want to find the optimal policy, but we can only find the optimal policy by acting suboptimally to explore sufficient actions. What we saw with on-policy learning was a compromise - it learns action values not for the optimal policy but for a near-optimal policy that still explores. An alternative is off-policy control where we have two policies: one used to generate the data (behaviour policy) and one that is learned for control (target policy). This is called *off policy learning*.

- Off-policy learning methods are powerful and more general than on-policy methods (on-policy methods being a special case of off-policy where target and behaviour policies are the same). They can be used to learn from data generated by a conventional non-learning controller or from a human expert.

- If we consider an example of finding target policy π using episodes collected through a behaviour policy b, we require that every action in π must also be taken, at least occasionally, by b i.e. $\pi(a|s) > 0$ implies $b(a|s) > 0$. This is called the assumption of ***coverage***.

- Almost all off-policy methods utilize *importance sampling*, a general technique for estimating expected values under one distribution given samples from another. Given a starting state S_t, the probability of the subsequent state-action trajectory occuring under any policy π is:

$$\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k) \tag{29}$$

  where p is the state transition probability function. We can then obtain the relative probability of the trajectory under the target and behaviour policies (the importance sampling ratio) as:

$$p_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{30}$$

  We see here that the state transition probability function helpfully cancels.

- We want to estimate the expected returns under the target policy but we only have returns from the behaviour policy. To address this we simply multiply expected returns from the behaviour policy by the importance sampling ratio to get the value function for our target policy.

$$\mathbb{E}\left[p_{t:T-1}G_t|S_t = s\right] = v_\pi(s) \tag{31}$$

- Note: importance sampling ratios are only non-zero for episodes where the target-policy has non-zero probability of acting *exactly* like the behaviour policy b. So, if the behaviour policy takes 10 steps in an episode, each of these 10 steps have to have been *possible* by the target policy, else $\pi(a|s) = 0$ and $\rho_{t:T-1} = 0$.

5.6 Incremental Implementation

We perform monte carlo policy evaluation (prediction) incrementally in the same way as was done in Chapter 2 for the bandit problem. Generally incremental implementation follows this formula:

$$NewEstimate \leftarrow OldEstimate + StepSize\left[Observation - OldEstimate\right] \tag{32}$$

With on-policy monte carlo methods, this update is performed exactly after each episode for each visit to a state given the observed rewards, with off-policy methods the update is slightly more complex. With ordinary importance sampling, the step size is $1/n$ where n is the number of visits to that state, and so acts as an average of the scaled returns. For weighted importance sampling, we have to form a weighted average of the returns which requires us to keep track of the weights. If the weight takes the form $W_i = \rho_{t:T(t)-1}$ then our update rule is:

$$V_{n+1} = V_n + \frac{W_n}{C_n}\left[G_n - V_n\right] \tag{33}$$

where,

$$C_{n+1} = C_n + W_{n+1} \tag{34}$$

with $C_0 = 0$. This allows us to keep tracking of the corrected weighted average term for each update as they are made. Note here that the weighted average gives more weight to updates based on common trajectories from b in π that we have some more often.

5.7 Off-policy Monte Carlo Control

Using incremental implementation (updates to the value function) and importance sampling we can now discuss *off-policy monte carlo control*–the algorithm for obtaining optimal policy π_* by using rewards obtained through behaviour policy b. This works in much the same way as in previous sections; b must be ϵ-soft to ensure the entire state space is explored in the limit; updates are only made to our estimate for q_π, Q, if the sequence of states an actions produced by b could have been produced by π. This algorithm is also based on GPI: we update our estimate of Q using Equation 33, then update π by acting greedily w.r.t to our value function. If this policy improvement changes our policy such that the trajectory we are in from b no longer obeys our policy, then we exit the episode and start again. The full algorithm is shown in 9.

5.8 Key Takeaways

- In the absence of a model of the environment, monte carlo methods allow us to evaluate and improve our value function based on *experience*
- We roll-out trajectories to terminal states, and back-propagate the rewards to the states visited en-route in several ways

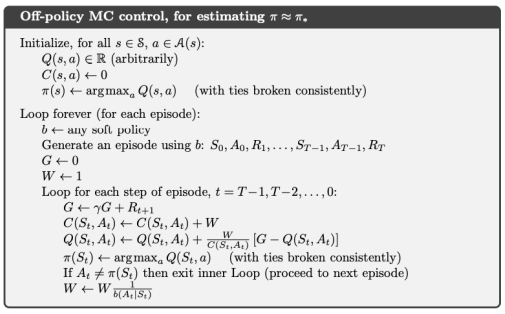

Figure 9: Off-policy monte carlo control

- Monte carlo methods use GPI (see chapter 4) in much the same way dynamic programming does. We evaluate our policy, then improve by acting greedily w.r.t our new value function until we converge on the optimal value function and policy.

- Differences with dynamic programming methods: 1) They do not require a model of the environment as they learn directly from experience, and 2) They do not bootstrap i.e. the value function estimates are an average of all real returns accumulated after visiting the state.

- Maintaining sufficient exploration is crucial with monte carlo methods; if our policy is deterministic we will likely not explore the full states space during our roll-outs. To deal with this we have several options: 1) Start every episode randomly with uniform probability such that we make sure we start at each possible state–called *exploring starts*, unrealistic in the real world as we can't make a robot restart from all possible states. Or 2) Use ϵ-soft policies that have a non-zero probability of selecting all possible states. The downside of doing this is that we will converge on the optimal ϵ-soft, which is not necessarily the optimal policy for the environment, because it needs to learn how account for its own randomness. This is the price we pay for exploration.

- Monte carlo methods can either be *on-policy* or *off-policy*.

- On-policy methods use the same policy to collect data as is evaluated and improved. This suffers the downsides outlined above.

- Off-policy methods have two policies, one that collects the data called the *behaviour policy* b and the other which we look to improve called the target policy π. We find trajectories from the behaviour policy that line up with our target policy, that is to say, that could have been produced by our target policy. This process only works if the behaviour policy has non-zero probability of selecting each of the actions in the target policy, aka *coverage* of the target policy. The agent explores, but learns a deterministic optimal policy offline that may be unrelated to the behaviour policy used to collect the experience.

- Based on rewards observed by running the behaviour policy, we update our value function using *importance sampling*, which measures, if effect, how likely the observed behaviour would have been given our target policy. For example, the target policy may take 1 of 4 actions with equal probability in each state. If we observe two timesteps from our beaviour policy, then our probability of selecting the actions taken by the behaviour policy would be $0.25x0.25$.

- We weight each return using the x*importance sampling ratio*, a measure of how likely we were to produce the roll-out using the target policy compared to how likely we were to produce the roll-out using the behaviour policy.
- Importance sampling can be *ordinary* i.e. an average of the returns observed from a state or *weighted* where trajectories viewed more often, or with a higher importance sampling ratio give the value update more weight.

6 Temporal-Difference Learning

TD learning is novel to RL. It is a hybrid that lies between monte carlo and dynamic programming methods. As with those, TD learning uses GPI; control (policy improvement) works in much the same way, it is in the prediction of the value function (policy evaluation) where the distinctions lie.

6.1 TD Prediction

Monte carlo methods wait until the end of the episode before back-propagating the return to the states visited en-route. The simplest TD method (TD(0)) does not wait until the end of the episode, in fact, it updates its estimate of the value function $V(s)$ based on the instantaneous reward as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{35}$$

Note here, that TD(0) *bootstraps* in much the same way DP methods do i.e. it bases it's estimate of $V(s)$ on the value of the next state $V(s')$. TD methods therefore take both sampling from monte carlo, and bootstrapping from DP. The TD(0) backup diagram is shown in Figure 10.

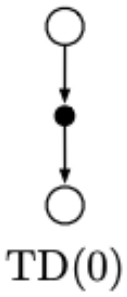

Figure 10: Backup diagram for TD(0)

The quantity in the brackets of equation 35 is known as the *TD error*. Formally this is expressed as:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{36}$$

In Figure 11 we see the differences between monte carlo and TD updates to the value function. In the case of monte carlo, we would need to wait until we arrived home before we could update our predictions for journey length. With TD learning, we can adjust these estimates on the fly based on the error in our predictions en-route. We see here that the adjustment we make changes at each timestep depending on the difference between predictions, that is to say, the update is proportional to the *temporal differences* between our predictions across timesteps.

6.2 Advantages of TD Prediction Methods

- TD methods generally converge faster than MC methods, although this has not been formally proven.
- TD methods do converge on the value function with a sufficiently small step size parameter, or with a decreasing stepsize.

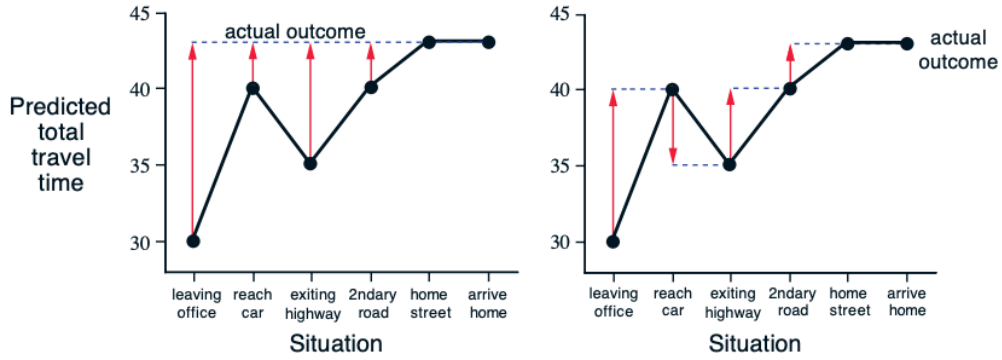

Figure 11: Monte carlo updates (left) and TD-updates (right) to the value function associated with a daily commute

- They are extremely useful for continuing tasks that cannot be broken down in episodes as required by MC methods.

6.3 Optimality of TD(0)

If we only have a finite number of episodes or training data to learn from we can update our value function in *batches*. We repeatedly play the same data, and update our value function by the sum of each of the increments for each state at the end of the batch.
TD batching and Monte Carlo batching converge often on two different estimates of the value function. The monte carlo batch estimate can only judge based on observed rewards from a state, but TD batching can make estimates based on later states using bootstrapping. Example 6.4 (*You are the predictor*) on pg 127 explains this excellently. The monte carlo estimate will converge on the correct estimate of the value function produced by the training data, but TD methods will generalise better to future rewards because they preserve the transition from state-to-state in the TD update, and thus bootstrap better from values of other states.
In general batch MC methods always minimise the mean squared error or the training set whereas TD(0) finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. This is called the *certainty equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated.

6.4 Sarsa: On-policy TD Control

We'll now look at using TD prediction methods for control. We'll follow, as before, the framework of Generalised Policy Iteration (GPI), only this time using TD methods for predicting the value function. For on-policy control, we wish to learn the state-action value function for our policy $q_\pi(s, a)$ and all states s and actions a. We amend the TD update formula provided in Equation 35 to account for state-action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{37}$$

This produces a quintuple of events: $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ giving rise to the name SARSA. As with other on-policy control algorithms, we update our policy to be greedy w.r.t our ever-changing value function.

6.5 Q-learning: Off-policy TD Control

Q-learning, much like SARSA, makes 1-step updates to the value function, but does so in subtly different ways. SARSA is on-policy, meaning it learns the optimal version of the policy used for collecting data i.e. an ϵ-greedy policy to ensure the state-space is covered. This limits its performance as it needs to account for the stochasticity of exploration with probability ϵ. Q-learning, on the other hand, is off-policy and directly predicts the optimal value function q_* whilst using an ϵ-greedy policy for exploration. Updates to the state-action value function are performed as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{38}$$

We observe that the max action selection at next state S_{t+1} means we are no longer following an ϵ-greedy policy for our updates. Q-learning has been shown to converge to q_* with probability 1.

6.6 Expected Sarsa

Instead of updating our value function with the value maximising action at S_{t+1} (as is the case with Q-learning) or with the action prescribes by our ϵ-greedy policy (as is the case with SARSA), we could make updates based on the *expected value* of Q at S_{t+1}. This is the premise of expected sarsa. Doing so reduces the variance induced by selecting random actions according to an ϵ-greedy policy. It's update is described by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}_\pi Q(S_{t+1}, A_{t+1} | S_{t+1}) - Q(S_t, A_t) \right] \tag{39}$$

$$\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{40}$$

$$\tag{41}$$

We can adapt expected SARSA to be off-policy by making our target policy π greedy, in which case expected SARSA becomes Q-learning. It is therefore seen as a generalisation of Q-learning that reliably improves over SARSA.

6.7 Maximization Bias and Double Learning

Many of the algorithms discussed so far in these notes use a maximisation operator to select actions - either ϵ-greedy or greedy action selection. Doing so means we implicitly favour positive numbers. If the true values of state action pairs are all zero i.e. $Q(S, A) = 0 \forall\ s, a$, then, at some stage in learning, we are likely to have distributed value estimates around 0. Our maximisation operator will select the postive value estimates, despite the latent values being 0. Thus we bias positive values, a so-called *maximisation bias*.
We can counter this by learning two estimates of the value function Q_1 and Q_2. We use one to select actions from a state e.g. $A^* = \text{argmax}_a\, Q_1(a)$ and then use the other to provide an estimate of its value $Q_2(A^*) = Q_2(\text{argmax}_a\, Q_1(a))$. This estimate will be unbiased in the sense that $\mathbb{E}\left[Q_2(A^*)\right] = q(A^*)$.

6.8 Games, Afterstates, and Other Special Cases

Much of the theory in this book revolves around action-value functions, where the value of an action in a given state is quantified **before** the action is taken. In some settings, it is expedient to quantify value functions after an action, as many actions may lead to the same state afterward. This is the case for the game tic-tac-toe, as described in the book.

In certain circumstances, it can be beneficial to amend value functions to accommodate afterstates if this property is shown.

6.9 Key Takeaways

- TD methods update value functions en-route to episode termination, rather than waiting to the end of the episode like Monte Carlo methods. Updates to the value function are made in proportion to the *temporal differences* between value function estimates.
- TD methods bootstrap off of value function estimates elsewhere in the state space, and consequently generalise to new data better than MC methods, which cannot make predictions outside of the observed training data.
- TD methods continue to follow the framework of Generalised Policy Iteration (GPI) as discussed in previous chapters.
- SARSA is the eminent on-policy TD method, but is limited in that in can only ever learn the optimal ϵ-soft behaviour policy.
- Q-learning is the eminent off-policy TD method, that will learn the optimal policy π_* with probability 1.
- Expected SARSA makes updates based on the expected value of the next state given a policy, reducing the variance induced by an ϵ-soft behaviour policy, performing better than SARSA given the same experience.
- The postive bias of action selection via maximisation can be mitigated by double learning methods.

7 n-step Bootstrapping

In the previous chapter we discussed one-step TD prediction, and in the chapter previous to that we discussed monte carlo methods where predictions are made based on returns at the end of the episode. In this chapter we will discuss an approach between these two, where predictions are made after n-steps in the future. Doing so is often more effective than either of the two previously presented approaches.

7.1 n-step TD Prediction

The spectrum of n-step TD methods is summarised by Figure 12. Recall from the previous chapter that our one-step return used for TD(0) was:

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1}) \tag{42}$$

we can generalise this to the n-step case as follows:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{43}$$

for all n, t such that $n \geq 1$ and $0 \leq t \leq T - n$. All n-step returns can be considered approximations to the full return, truncated after n steps and then corrected for the remaining missing terms by $V_{t+n-1}(S_{t+n})$. If the n-step return extends to or beyond termination then all the missing terms are taken as zero.

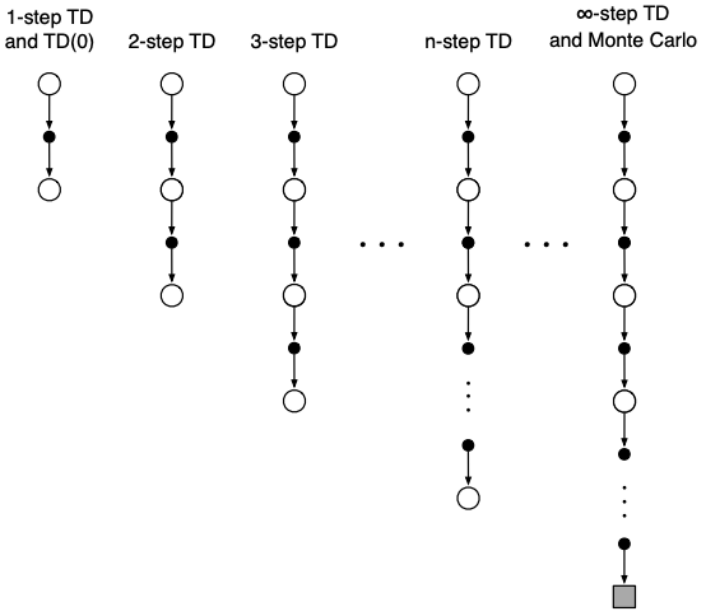

Figure 12: Backup diagram for TD(0)

The n-step return uses the value function V_{t+n-1} to correct for missing rewards beyond R_{t+n}. An important property of n-step returns is that their expectation is guaranteed to be a better estimate of v_π that V_{t+n-1} in a worst-state sense i.e.:

$$\max_s |\mathbb{E}_\pi \left[G_{t:t+n} | S_t = s \right] - v_\pi(s)| \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)| \tag{44}$$

This is called the error reduction property of n-step returns. Because of this, one can show formally that all n-step TD methods converge to t he correct predictions under appropriate conditions.

7.2 n-step Sarsa

We can extend n-step value function prediction to a control algorithm by incorporating the ideas of the Sarsa. Now, instead of making value function updates based on rewards received to the $t+n^{th}$ state, we select update our state-action value function Q by making updates up to the $t+n^{th}$ state-action pair. The family of n-step Sarsa backup diagrams is shown in Figure 13. The n-step Sarsa update is:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right] \tag{45}$$

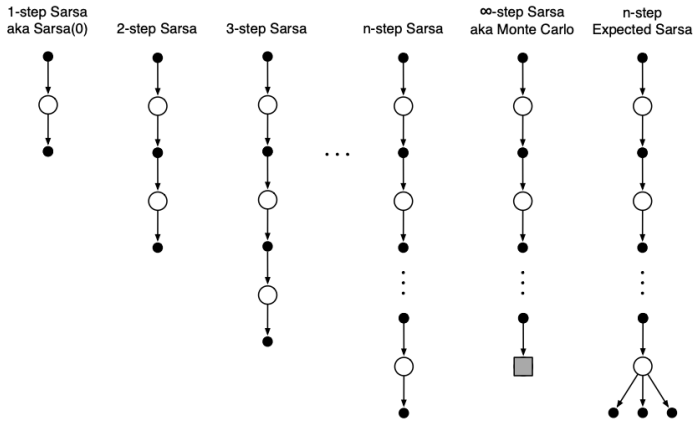

Figure 13: Backup diagrams for family of n-step Sarsa algorithms

An example of why n-step sarsa speeds up policy learning is given in Figure 14
Expected Sarsa remains an important version of Sarsa because of the variance-reducing expectation. The algorithm is described as in equation 45, but this time the expected return includes the expected value of the final value function, i.e.:

$$G_{t:t+n} = R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}) \tag{46}$$

where $\bar{V}_{t+n-1}(s)$ is the *expected approximate value* of state s, using the estimated action values at time t, under the target policy:

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) Q_t(s,a), \quad \text{for all } s \in \mathcal{S} \tag{47}$$

Expected approximate values are used in developing many of the action-value method discussed hereafter.

7.3 n-step Off-policy Learning

We can extend n-step learning to the off-policy case where we have a behaviour policy b that creates the data and a target policy π that we wish to update. As previously discussed, when we make updates in this way, we must weight the updates using the importance sampling ratio - equation 30. We can create a generalised form of the n-step algorithm, that works for both on-policy and off-policy cases:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n} \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right] \tag{48}$$

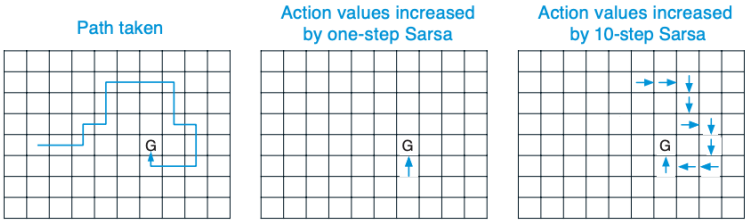

Figure 14: Gridworld example of the speedup of policy learning due to the use of n-step methods. The first panel shows the path taken by an agent in a single episode, ending at a location of high reward, marked by the G. In this example the values were all initially 0, and all rewards were zero except for a positive reward at G. The arrows in the other two panels show which action values were strengthened as a result of this path by one-step and n-step Sarsa methods. The one-step method strengthens only the last action of the sequence of actions that led to the high reward, whereas the n-step method strengthens the last n actions of the sequence, so that much more is learned from the one episode.

7.4 Per-decision Methods with Control Variates

The off-policy methods described above may not learn as efficiently as they could. If we consider the case where the behaviour policy does not match the target policy for one of the n steps used in the update, the update will go to 0 and the value function estimate will not change. Instead we can use a more sophisticated approach, that has a different definiton of the n-step horizon:

$$G_{t:h} \doteq \rho_t(R_{t+1} + \gamma G_{t+1:h}) + (1 - \rho_t)V_{h-1}(S_t) \tag{49}$$

In this approach, if ρ_t is zero, then instead of the target being zero and causing the estimate to shrink, the target is the same as the estimate and causes no change. Because the expected value of ρ_t is 1, the expected value of the second term in equation 49, called the *control variate* has expected value of 0, and so does not change the update in expectation.

7.5 Off-policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm

Here we will discuss off-policy learning without the need for importance sampling, conceptualised by the n-step Tree Backup Algorithm–Figure 15. In this tree backup update, the target now includes all rewards *plus* the estimated values of the dangling action nodes hanging off the sides, at all levels. It is an update from the *entire tree* of estimated action values. For each node in the tree backup diagram, we the estimated values of the non-selected actions are weighted by their probability of being selected under our policy $\pi(A_t|S_t)$. The value of the selected action does not contribute at all at this stage, instead its probability of being selected weights the instantaneous reward of the next state *and* each of the non-selected actions at the next state, which too are weighted by their probabilities of occurring as described previously. Formally, the one-step return is as follows:

$$G_{t:t+1} \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q_t(S_{t+1}, a) \tag{50}$$

the two-step backup return (for $t < T - 1$) is described recursively as:

$$G_{t:t+2} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})Q_t(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})G_{t+1:t+2} \tag{51}$$

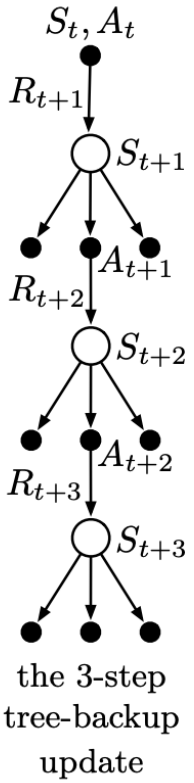

Figure 15: The 3-step tree-backup update

in the general case this becomes:

$$G_{t:t+n} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n} \qquad (52)$$

These returns are then used in the normal action-value update for n-step sarsa, as described by equation 45.

7.6 A Unifying Algorithm: n-step Q(σ)

We'd like an algorithm that generalises the differences between n-step sarsa, as discussed at the beginning of this chapter, and n-step tree backup as discussed at the end of this chapter. To do so, we create a new algorithm parameter $\sigma \in [0, 1)]$ that acts as a linear interpolator between the two extreme cases. This allows us to do a full, expected tree back-up in some cases, and a straight sarsa update on others, or work in the continuous region between. The family of algorithms explored in this chapter are summarised in figure 16:

7.7 Key Takeaways

- n-step TD methods lie between one-step TD methods as described in the previous chapter and full Monte Carlo backups as described in the chapter two previous. They typically perform better than either of these extremes.

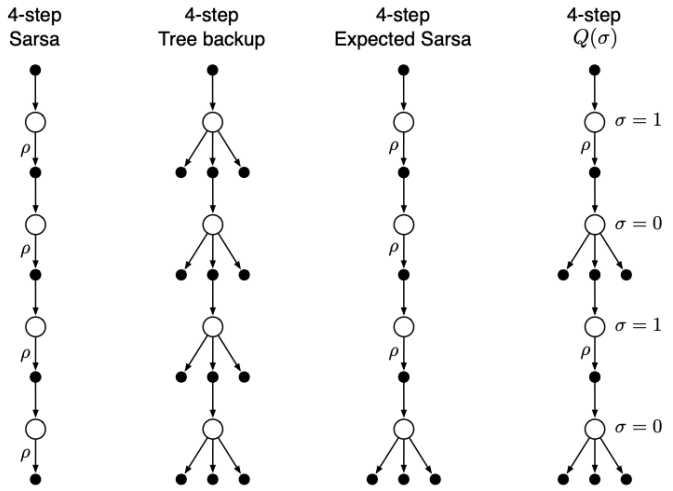

Figure 16: The backup diagrams of the three kinds of n-step action-value updates considered so far in this chapter (4-step case) plus the backup diagram of a fourth kind of update that unifies them all. The label 'ρ' indicates half transitions on which importance sampling is required in the off-policy case. The fourth kind of update unifies all the others by choosing o a state-by-state basis whether to sample ($\sigma_t = 1$) or not ($\sigma_t = 0$).

- n-step sarsa is an on-policy algorithm that updates state-action values based on rewards obtaining n timesteps into the future following our policy π.
- n-step off policy learning corrects for the failures of on-policy learning by weighting updates by the importance sampling ratio ρ_t as discussed in chapter 6.
- Variance in the off-policy n-step updates can be reduced using *per-decision methods with control variates* that stop the value update being 0 if the behaviour policy b does not match the target policy π for any of the n timesteps.
- Finally, these methods are all more complex (expensive), and require additional memory, than one-step TD methods discussed previous.
- We can do without importance sampling if our update is based on an expectation over each of the state-action values visited along our n-step trajectory. This is called the *n-step tree backup algorithm.*
- The continuum between all of these discussed methods can be traversed using the n-step $\mathrm{Q}(\sigma)$ algorithm that characterised at which time steps we take full expectations and which we use only the sarsa update.

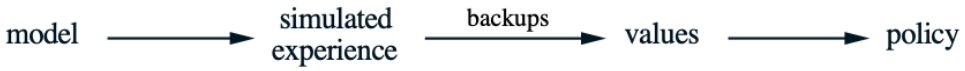

Figure 17: The framework for updating our policy using our model of the environment

8 Planning and Learning with Tabular Methods

RL methods can be described as either *model-based* or *model-free*. Model-based methods rely on *planning* as their primary component, while model-free methods rely on *learning*. Despite their differences, both still use value functions and both makeup backups to state values based on future returns or estimates. This chapter will provide a unifying framework for model-free and model-based RL methods.

8.1 Models and Planning

- Models are anything the agent can use to predict the outcome of its actions.
- Models can be either*distribution models* or *sample models*. The former is a probability distribution over all possible next states whereas the latter is just produced one of the possible next states sampled from the probability distribution.
- Distribution models are stronger than sample models because they can always be used to create samples, but sample models are easier to create in practice.
- Models are used to *simulate* the environment thus producing *simulated experience*. Distribution models could produce every possible episode, weighted by their probability of occurring, whereas sample models can only produce one episode.
- The word *planning* refers to any computational process that takes a model as input and produces or improves a policy for interacting with the modelled environment.
- The common structure of updating our policy by utilising our model is given in Figure 17.
- We can use models in place of the real environment to perform model-free learning safely i.e. using simulated experience rather than real experience.

8.2 Dyna: Integrated Planning, Acting, and Learning

Instead of planning all possible future states permutation, we may want to planning over a small number of future timesteps. When we do this we will likely want to update both our policy and model *online*. The canonical algorithm for doing so is **Dyna-Q**.

- Planning agents can use the experience they collect to do one of two things: 1) improve the model (also called *model-learning*) and 2) improve the value function and policy (also called *direct reinforcement learning (direct RL)*). These options are summarised in 18. Sometime model-learning is called *indirect RL* because improving our model, improves our policy by proxy.
- Dyna-Q agents conduct direct RL, planning, model-learning and acting simultaneously. Planning and direct learning tend to use exactly the same machinery as each other e.g. n-step sarsa and so are closely linked. Figure 19 shows the generalised Dyna architecture.

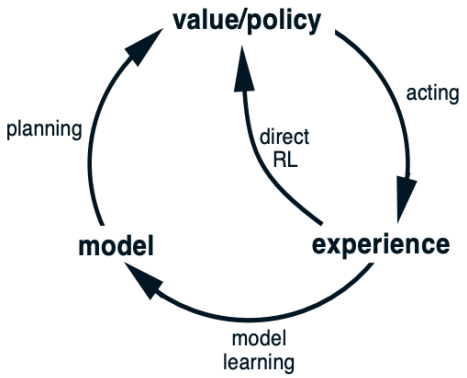

Figure 18: Direct RL and model-learning, either achieved using data collected from experience

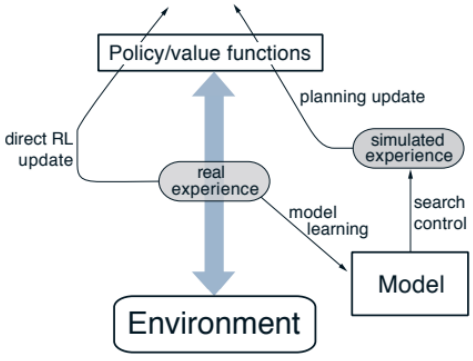

Figure 19: Generalised Dyna architecture