Reinforcement Learning

William Ogier

April 29, 2022

Declaration of Authorship

This piece of work is a result of my own work except where it forms an assessment based on group project work. In the case of a group project, the work has been prepared in collaboration with other members of the group. Material from the work of others not involved in the project has been acknowledged and quotations and paraphrases suitably indicated.

Contents

Chapter 1

Introduction

1.1 What is reinforcement learning

Reinforcement learning is a category of machine learning where computer algorithms can improve themselves over time through their own experience, without human interaction. Reinforcement learning stems from the most natural and fundamental ideas of learning, ideas which are encoded within our brains [1]. Think of a baby, new to the earth with no understanding of the environment around them, how does it learn? The baby will take actions, observe consequences and it's from these consequences the baby bases it's learning. For example, they may touch a stinging nettle and feel pain, try to walk moving both feet forward at the same time and fall over, or may call to their mother and receive food. This idea of being rewarded after we act, whether this reward be good or bad, is fundamental to reinforcement learning, and choosing actions to maximise future reward is at the heart of all reinforcement learning algorithms.

But how does this idea of reward work in computers? Obviously these feelings of pain or hunger don't translate well to the inanimate computers, so, instead, we let reward be a scalar feedback signal that the program observes after taking actions. This simplifies greatly the idea of maximising future reward, as the real numbers are perfectly ordered, but this seems almost too good to be true. Reinforcement learning has proven to be immensely useful, solving a variety of complex problems, so can the goals of all these algorithms be converted to maximising the future return of a single scalar feedback signal? That is exactly what the reward hypothesis postulates. As stated by Sutton [2, p. 57]:

> That all of what we mean by goals and purposes can be well thought of as the maximisation of the expected value of the cumulative sum of a received scalar signal (called reward).

With the limited complexity of examples and applications within this project, we shall see that whether this hypothesis is true or not isn't a problem for us. Converting our ideas of goals into a scalar reward will be fairly straightforward, but that isn't to say this is always an easy task.

Reinforcement learning is an active area of research and has seen impressive results in a large variety of situations. Some of these include:

- Games - Computer programs have been trained to play games at superhuman levels. Notably, in 2016 Deepmind's AlphaGo program beat arguably the greatest Go player of all time, as documented in the film AlphaGo [3]. Other feats have been computers taught to perfectly play the entire Atari game library.
- Autonomous Machines - From robots opening doors to self-driving cars, there is a lot of research into training machines through reinforcement learning. A particularly visu-

ally impressive example being professors at Stanford, Abbeel et al. [4], able to perform autonomous aerobatic helicopter flight.

In a broad sense, reinforcement learning uses a large amount of trial and error combined with this reward system to learn about the optimal actions to take. The specific algorithms will be discussed later, but this general idea leads to one of the core problems in reinforcement learning - *exploration vs exploitation*. If a program knows certain actions lead to large rewards it should *exploit* these actions as the aim is to maximize total reward. However, how does it know these are optimal? *Exploration* is needed to guarantee the program isn't missing the best possible actions. Again, these ideas link nicely to human nature. We may want to keep eating our favourite ham and cheese sandwiches for lunch everyday, we know they taste good and adventuring away may just ruin our meal. However, unless we explore we will never experience the taste of the elite tuna sandwich. Disaster! We will see certain simple methods to try overcome this problem, but in general it's an active area of research with many approaches.

1.2 Agent & Environment

Moving to the proper terminology of reinforcement leaning, there is an *agent* and an *environment*. The agent occupies *states*, and in these states they choose *actions*. Through these actions the agent learns and bases it's future decisions. The environment is basically the world in which the agent lives. By performing an action, the agent interacts with the environment and, in return, receives a numerical reward and is transitioned to a new state. These terms are very general but this is because the scope of reinforcement learning is so broad and it's applications so varied. There is a powerful mathematical framework underlying these components, namely *Markov Decision Processes*. However, we shall first get a more intuitive sense of these core components through a few examples.

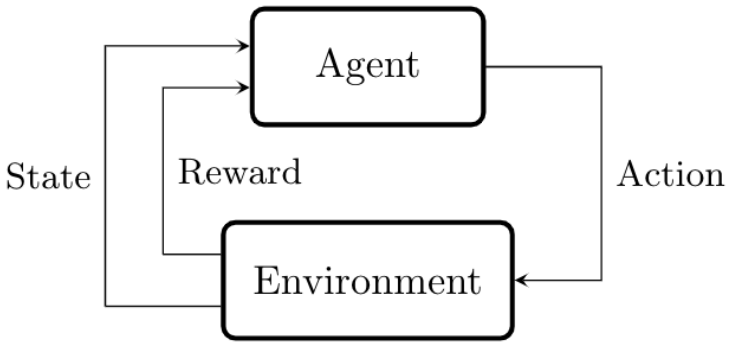


Figure 1.2.1: The main reinforcement learning cycle

Consider a game of chess. The agent is one of the players and the states are all possible configurations of the board. The actions the agent can take are simply the legal moves, but how would we define the reward? It may be tempting to reward the agent when they capture pieces and then give a larger reward for winning. But the goal of chess is not to capture pieces, but to win. This definition may cause the agent to prioritise capturing as many of the opponents pieces over beating the opponent. A reward of $+1$ for a winning move, -1 for a losing move and 0 for everything else is, therefore, sensible. The power of reinforcement learning means that the agent will still learn to capture pieces and protect it's own.

Consider now the game Space Invaders. The agent is controlling the cannon, able to both move horizontally and shoot. The states are the different images produced by the raw pixels. While, to humans, this gives valuable information such as location of the cannon and number of lives, the computer program will simply store this image as a matrix of numbers. Here we have an

obvious choice for the reward, simply the in-game score, since trying to maximise this is the ultimate aim of the game. Through playing many games of space invaders the agent will learn which actions to take in certain states in order to gain large in-game scores.

We can see from these examples the idea of reward, and how it is defined, is hugely important in reinforcement learning and that through the generality of states and actions these ideas can be applied in a range of settings. So while, yes, there has been a variety of specific computer programs that beat professional chess players, such as Deep Blue in 1997 [5], they have required expert chess knowledge written into the code. Meanwhile, reinforcement learning provides a general common framework with which we can solve a variety of problems.

1.3 Markov Decision Processes

We shall now introduce the popular mathematical framework in which we will set up the environment of all our reinforcement learning problems: Markov decision processes (MDPs). We return to the interaction of agent and environment and now let these interactions occur at discrete time steps $t \in \{0, 1, \dots\}$. At time t the agent occupies *state* $S_t \in \mathcal{S}$ and chooses *action* $A_t \in \mathcal{A}$. This leads us onto the next time step, $t+1$, where the environment gives the agent a possibly random *reward* $R_{t+1} \in \mathscr{R} \subset \mathbb{R}$ and transitions them to a new state S_{t+1}, which, again, is random in general. The process then repeats. This gives us a sequence of states, actions and rewards : $S_0, A_0, R_1, S_1, A_1, R_2, \dots$ Through this project the majority of our focus will be on finite MDPs where the sets $\mathcal{S}, \mathcal{A}$ and $\mathscr{R}$ are all finite, but as we shall see there is capability to delve into more continuous concepts of states and actions. We can now define a finite MDP.

Definition 1.3.1 (Finite Markov Decision Process)**.** A finite Markov decision process, as given by Sutton [2, p. 67] is a 4-tuple $\langle \mathcal{S}, \mathcal{A}, p, \gamma \rangle$ where

- $\mathcal{S}$ is a finite set of states that follow the Markov property
- $\mathcal{A}$ is a finite set of actions with $\mathcal{A}(s) \subseteq \mathcal{A}$ the possible actions in state s
- $p : \mathcal{S} \times \mathscr{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a transition probability function giving the probability of going from state s to s' and receiving reward r given an action a
  - $p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a]$
- $\gamma \in [0, 1]$ is the discount factor which shortly comes into play

Although the function p completely describes the dynamics of the MDP, it can be cumbersome to work with. So, we define two further functions. Firstly, the state transition matrix $\mathcal{P}$ giving the conditional probability of going from state s to s' given an action a,

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] = \sum_{r \in \mathscr{R}} p(s', r|s, a). \tag{1.3.2}$$

Secondly, a reward function $\mathcal{R}$ giving the conditional expected reward given you are in state s taking action a,

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in \mathscr{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a). \tag{1.3.3}$$

To expand on a few things, firstly, the Markov property of the states means the following equality holds,

$$\mathbb{P}[S_{t+1} = s_{t+1} | S_t = s_t] = \mathbb{P}[S_{t+1} = s_{t+1} | S_t = s_t, S_{t-1} = s_{t-1}, \dots, S_0 = s_0]. \tag{1.3.4}$$

This gives rise to the famous saying *the future is independent of the past given the present.* The current state will always fully characterise the future progression, we don't need to keep a log

of all previous states. A good way to think about this property is that it means the states are sufficiently detailed so that it tells us all that we need to know from the history. For example in a game the state will not only tell us the location of our character, but the number of lives, the items in the inventory, the different locations we have opened up and much more. Secondly, the probabilities and expectations of $\mathcal{P}$ and $\mathcal{R}$ allow for generality to stochastic scenarios, where given a state and an action there are multiple different states and rewards the environment can return to the agent, each with their own probabilities. Due to the finite size of $\mathcal{S}, \mathcal{A}$ and $\mathscr{R}$ these probabilities can be well defined and are a key component when building an MDP. Often, however, we see more deterministic outcomes, whereby the MDP is built so that when in a state and taking an action there is only one reward and one state that the environment is able to return to the agent. Finally, we have time-homogeneity throughout our MDPs, meaning any transition probability or reward expectation is in-dependant of the time step t.

Example 1.3.5. We now graphically present a very simple MDP in Figure 1.3.1, depicting an agent in a car at a set of traffic lights. We have that:

- $\mathcal{S} = \{Red,\ Red/Orange,\ Green,\ Crash,\ Ticket\}$
- $\mathcal{A} = \{wait,\ drive\}$
- $\mathscr{R} = \{-10,\ -5,\ 0,\ +1,\ +10\}$
- Transition Probabilities as depicted in the diagram below
- $\gamma = 1$

This particular MDP shows deterministic reward dynamics whereby going from state s to s' through action a gives a singular reward r with probability 1. So now, for example, if the agent occupies state 'Red' and takes action 'wait', there is a 0.2 chance they will be transitioned back to state 'Red' and gain 0 reward, and a 0.8 chance they will be transitioned to state 'Red/Orange' and gain +1 reward. The specific rewards are fairly arbitrary here, however, their relative values denote obvious things such as crashing being viewed as worse than getting a ticket, whilst waiting for the green light is better than getting a ticket. Finally, we see three absorbing states, Crash, Ticket and Green which the agent cannot leave and denote the end of agents task.

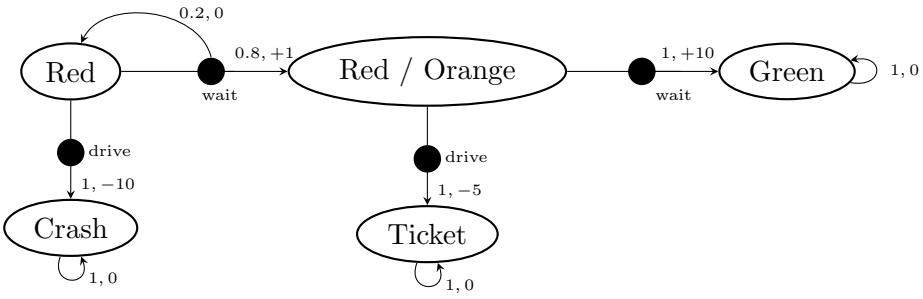


Figure 1.3.1: Traffic light MDP

It should be stated that, although there may exist an MDP perfectly describing the environment, we may not actually know the exact dynamics and probabilities of the MDP. For example, if the state space is extremely large, or if the MDP is just hugely complex. Furthermore, this is actually the general case, and for most interesting applications we don't have this knowledge. The theory of acting optimally when we do comes under *dynamic programming* and we shall still first cover these ideas as it will give us the foundations to then tackle more general reinforcement learning problems.

1.4 Future Cumulative Reward

As touched on before, reinforcement learning aims to maximise the future reward our agent will earn. To formalise this idea, there first needs to be consideration of episodic and continuing tasks. Episodic tasks are those that terminate, for example a game of chess, while continuous tasks go on forever, such as robot being trained to constantly clean a warehouse. To model the environment of these episodic tasks using MDPs we let the terminating states, such as a checkmate, be absorbing states. That is, they transition to themselves with probability 1 and gain 0 reward for each transition, as seen by the states $\{Green, Crash, Ticket\}$ from our traffic light MDP in Figure 1.3.1. This unifies episodic and continuous tasks since they now both produce an infinite sequence $S_0, A_0, R_1, S_1, A_1, R_2, \dots$ of states, actions and rewards, and from this we can formalise our ideas of future cumulative reward.

Definition 1.4.1 (Return)**.** Given a discount factor γ and a time step t Sutton [2, p. 60] defines the return G_t as

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

the discounted sum of future rewards from time t.

Since the rewards are random, the return is also random, and thus we shall see our aims are to choose actions that will maximise the return in expectation, but more on this later. We first discuss the discount factor and the two very important roles it plays. Firstly, it gives a finite return in the case of continuous tasks, where an infinite number of non-zero rewards can be gained. Due to the finite size of $\mathscr{R}$ we have that the absolute value of any reward R_t is bounded by some $M \geq 0$. Therefore for $\gamma \in [0, 1)$,

$$|G_t| = \left| \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \right| \leq \sum_{k=0}^{\infty} |\gamma^k R_{t+k+1}| \leq \sum_{k=0}^{\infty} |\gamma^k M| = \frac{M}{1-\gamma} \tag{1.4.2}$$

So now this idea of choosing actions to maximise return is a lot simpler due to this bound on G_t. In episodic MDPs, where the agent almost surely ends up in an absorbing state, we can still have $\gamma = 1$, since an infinite return isn't possible in this scenario. Secondly, the discount factor acts as a slider varying how much we value rewards that are closer in time. In the extreme case of $\gamma = 0$, we see $G_t = R_{t+1}$ meaning the agent only cares about maximising the instant reward from it's next action. While if $\gamma = 1$ we have an un-discounted scenario where rewards at different times are viewed equally. Discount factors are arise frequently in financial problems where the *time value of money* comes into play. Due to things like banks interest rates, an amount of money is more valuable now than it is later. But away from finance, these ideas hold in general life, you would often rather have a reward sooner rather than later, and so this discount factor enables these concepts to translate into reinforcement learning problems.

Finally, the definition of return gives way to a recursive relationship between G_t and G_{t+1} that will allow us to derive the famous *Bellman Equations*.

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} \dots) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{1.4.3}$$

1.5 Policies and Value Functions

We haven't yet formalised what it means for an agent to choose actions. A few initial options spring to mind. The agent could choose each available action uniformly at random, could choose

any action that maximises the reward function $\mathcal{R}_s^a$ (1.3.3), or, for our traffic light example the agent could never choose the action 'wait'. Each of these fully describe how the agent will act and are examples of *policies*.

Definition 1.5.1 (Policy)**.** A policy, π, is a probability distribution for actions given a state,

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s].$$

This tells us how likely each action is in a given state and again is time-homogeneous. Often a policy will be deterministic whereby, given a state s, the agent will take action

$$a = \pi(s)$$

with probability 1, even though multiple actions may be available. So now, given an MDP, a policy and an initial starting state, S_0, we have fully characterised how an agent will act, although the stochastic nature of policies and MDPs means we may not know the exact sequence of states, actions and rewards. Of course, in reinforcement learning the goal is for the agent to learn through experience, so by sticking to one policy the agent will never improve. Therefore we shall see reinforcement learning algorithms evolve policies overtime, with an overall aim to find a policy that maximises the return. To do this, we first need to introduce *value functions*, using the notation and ideas given by Sutton [2, p. 70].

Definition 1.5.2 (State-value function)**.** Given an MDP and a policy, π, the state-value function, $v_\pi(s)$, is the expected return given you are in state s and follow policy π,

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s].$$

Definition 1.5.3 (Action-value function)**.** Given an MDP and a policy, π, the action-value function, $q_\pi(s, a)$, is the expected return given you are in state s, take the specific action a and then follow policy π,

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a].$$

These expectations that are sub-scripted by policies just mean we need to take the randomness of actions into account, on top of the randomness of the variable over which we are taking an expectation. For example, we can write

$$\mathbb{E}_\pi[R_{t+1}|S_t = s] = \sum_{a\in\mathcal{A}} \pi(a|s)\mathcal{R}_s^a,$$

so the expectation of the next reward under policy π is the weighted average of expected rewards given actions, weighting by the probability of these actions under policy π. These value functions allow us to compare how good particular states or state-action pairs are under a policy, but also allow comparisons of different policies.

Example 1.5.4. We now return back to our MDP from Example 1.3.5 and show its state values under a policy π. For this example we let the policy be to choose action 'wait' with probability one given the agent is in state 'Red', and when the agent is in state 'Red/Orange' they take each possible action uniformly random. Using our notation, this means

$$\begin{aligned} \pi(\text{wait}|\text{Red}) &= 1, \\ \pi(\text{drive}|\text{Red}) &= 0, \\ \pi(\text{wait}|\text{Red/Orange}) &= 0.5, \end{aligned}$$

$$\pi(\text{drive}|\text{Red/Orange}) \quad = \quad 0.5.$$

One run through of this MDP, given our agent begins at state 'Red', could be the following sequence of states, actions and rewards: Red, wait, 0, Red, wait, 0, Red, wait, 1, Red/Orange, drive, -5, Ticket.

Now we shall calculate the state-values, the expected return from a state, remembering $\gamma = 1$ so there is no discounting and the return will simply be the sum of all future rewards. Clearly the expected return from any absorbing state is 0, as the agent continuously gains 0 reward. From state 'Red/Orange', half the time our agent will choose action 'wait' and gain reward +10, and half the time will choose 'drive', gaining -5 reward. Each of these then lead to a terminating state, so the expected return is simply $0.5 \times 10 + 0.5 \times -5 = 2.5 = v_\pi(\text{Red/Orange})$. From state 'Red', the agent always takes action 'wait' which either returns the agent to the same state and gives 0 reward or transitions to the state 'Red/Orange' giving reward +1. The agent will eventually always reach state 'Red/Orange' and receive this reward of +1, since there is 0 probability that the agent will just loop back to state 'Red' an infinite number of times. Once at state 'Red/Orange' the rewards work as previously described, therefore, the expected sum of future rewards given the agent is in state 'Red', is $1 + 0.5 \times 10 + 0.5 \times -5 = 3.5 = v_\pi(\text{Red})$. These values can be seen in Figure 1.5.1. It should be stressed that the very simplistic nature of this MDP meant we could easily calculate these state-values. In general, these calculations are complex - think of very large state and action spaces within a continuous task, so returns are infinite discounted sums. As mentioned before, we shall instead exploit certain iterative methods that will allow us to find these state values, rather than undertaking brute force calculations like we have just seen.

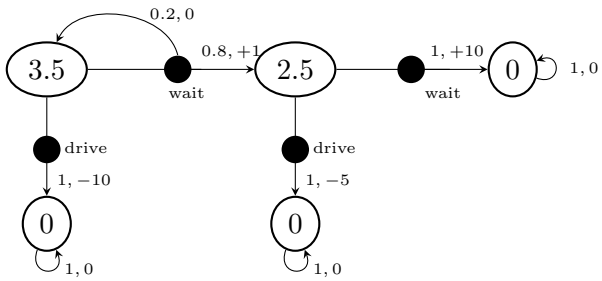


Figure 1.5.1: Traffic light MDP state-values for a specific policy

Moving on in the theory, it is possible to decompose these definitions into recursive relationships between the value of a state and the value of the next possible states.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s]. \end{aligned} \tag{1.4.3}$$

Now by using the partition theorem for expectation we have

$$v_\pi(s) = \sum_{a,r,s'} \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a, R_{t+1} = r, S_{t+1} = s']\mathbb{P}[A_t = a, R_{t+1} = r, S_{t+1} = s'|S_t = s],$$

and since G_{t+1} is conditionally independent of S_t, A_t and R_{t+1} given S_{t+1}, from the Markov property of states, we have

$$v_\pi(s) = \sum_{a,r,s'} \mathbb{E}_\pi[r + \gamma G_{t+1}|S_{t+1} = s'] \frac{\mathbb{P}[A_t = a, R_{t+1} = r, S_{t+1} = s', S_t = s]}{\mathbb{P}[S_t = s]}$$

$$
\begin{aligned}
&= \sum_{a,r,s'} \mathbb{E}_\pi[r + \gamma G_{t+1} | S_{t+1} = s'] \frac{\mathbb{P}[S_t = s, A_t = a]}{\mathbb{P}[S_t = s]} \frac{\mathbb{P}[A_t = a, R_{t+1} = r, S_{t+1} = s', S_t = s]}{\mathbb{P}[S_t = s, A_t = a]} \\
&= \sum_{a,r,s'} \mathbb{E}_\pi[r + \gamma G_{t+1} | S_{t+1} = s'] \mathbb{P}[A_t = a | S_t = s] \mathbb{P}[S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a] \\
&= \sum_{a,r,s'} \mathbb{E}_\pi[r + \gamma G_{t+1} | S_{t+1} = s'] \pi(a|s) p(s', r|s, a) \\
&= \sum_{a,r,s'} [r + \gamma v_\pi(s')] \pi(a|s) p(s', r|s, a) \\
&= \sum_{a} \pi(a|s) \sum_{r,s'} p(s', r|s, a)[r + \gamma v_\pi(s')] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]. \qquad (1.5.5)
\end{aligned}
$$

This can also be written by plugging the functions $\mathcal{P}$ (1.3.2) and $\mathcal{R}$ (1.3.3) in as follows,

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right). \qquad (1.5.6)$$

Sutton names equation (1.5.6) as the *Bellman Equation* of v_π and there exists a similar recursive equation for action-value functions. What this equation is telling us is that under policy π the value of state s can be viewed as the expected reward as you leave s plus the discounted expectation of the value of the next following state. This next state is random and so this expectation makes sense, even though state-values are deterministic values. If you break this Bellman equation down, it is simply a system of $|\mathcal{S}|$ linear equations with $|\mathcal{S}|$ unknowns and so can be solved through a variety of methods. Iterative methods are mostly used due to matrix inversion being of higher computational cost when the state spaces get larger. We shall prove later on that there is a unique solution to the Bellman equation for v_π, and so we can actually define the value function v_π as the solution to equation (1.5.6). Furthermore, we shall state and prove certain iterative methods that can find the state-value function.

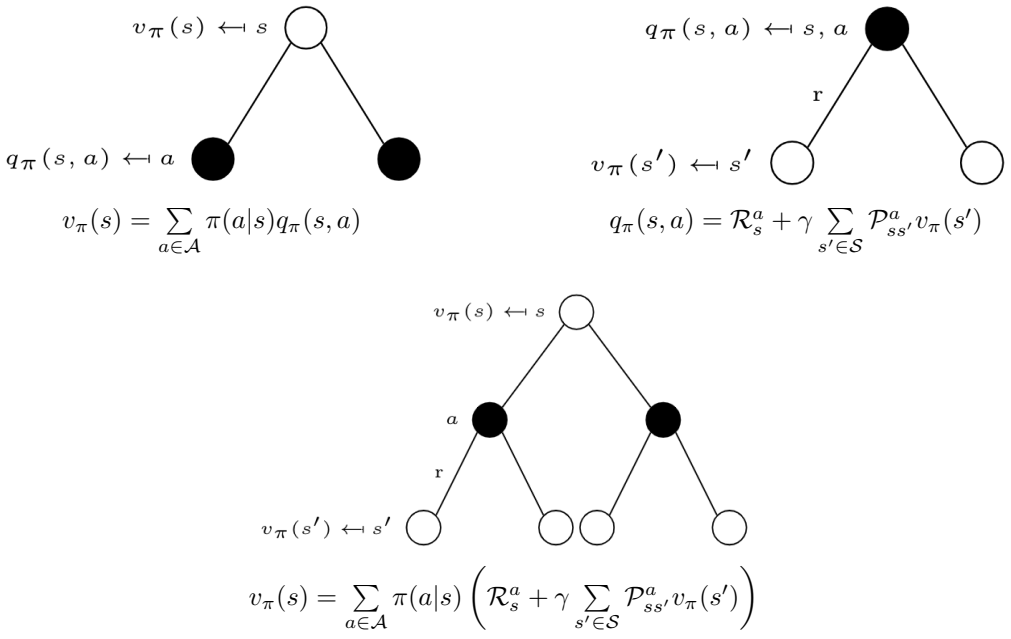


Figure 1.5.2: Backup Diagrams

We first gain a more visual understanding of this equation through the use of *backup diagrams*, as introduced by Sutton [2, p. 72, Fig 3.4]. These show the general paths an agent could take

in an MDP, and use white circles to denote states and black circles to denote actions. The three diagrams in Figure 1.5.2 help us visualise why the Bellman equation (1.5.6) is true. The first diagram shows a state s that we are trying to find the value of with regard to v_π i.e the expected return from s following π. From s we can take a range of possible actions each with their own probabilities due to π, in this diagram only two are shown for simplicity. Each of these actions have their own action-values given by q_π, which we treat as known, telling us the expected return from taking a specific action and then following policy π. So we can calculate $v_\pi(s)$ by averaging over these q_π values and weighting by their respective probabilities of actions given by the policy π. This gives the first equation,

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a). \tag{1.5.7}$$

In the second diagram, we now start at a state-action pair and want to find its action-value i.e the expected return following policy π thereafter. Due to the dynamics of the MDP, after taking an action in a state the environment can give a variety of rewards and transition the agent to a variety of states. Each of these states have their own state-values telling us their expected return. We can therefore think of $q_\pi(s, a)$ as the expected reward when taking action a in state s, $\mathcal{R}^a_s$, plus the discounted average expected return at the new possible states weighted by the probability of reaching that state. This discount is needed since the return received at these new states is received in the next time-step. This gives us the second equation,

$$q_\pi(s, a) = \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} v_\pi(s'). \tag{1.5.8}$$

Now through combining the two smaller diagram we get the full backup diagram for v_π and combining the two equations gives us the Bellman equation (1.5.6).

Example 1.5.9. We will now verify the the Bellman equation (1.5.6) for v_π does indeed hold, particularly for the state 'Red'. Under π only one action is possible in state 'Red', namely 'wait'. As Figure 1.5.1 shows us, the expected reward our agent earns from this action is $0.2 \times 0 + 0.8 \times 1 = 0.8$. With probability 0.8 this sends the agent to state 'Red/Orange' which has state value 2.5, and with probability 0.2 sends the agent to state 'Red' which has state value 3.5. Plugging this all into the Bellman equation gives

$$\begin{aligned} 3.5 = v_\pi(Red) &= \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} v_\pi(s') \right) \\ &= 1[0.8 + 1(0.8 \times 2.5) + 1(0.2 \times 3.5)] \\ &= 3.5. \end{aligned}$$

So, as expected, the Bellman equation does hold for this value function, and we see this recursive relationship between successive state values.

1.6 Optimality

As stated before, just knowing the state or action values with respect to a specific policy isn't, alone, terribly useful. Instead we need some notion of making optimal decisions that give us the best expected return from a state. We, therefore, introduce optimal value functions, using the notation of Sutton [2, p. 75,].

Definition 1.6.1 (Optimal state-value function)**.** The optimal state-value function, $v_*(s)$ is the maximum value function over all policies,

$$v_*(s) = \max_\pi v_\pi(s).$$

Definition 1.6.2 (Optimal action-value function)**.** The optimal action-value function is the maximum action-value function over all policies,

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

These functions tell us at either a state or state-action pair the largest expected return we can gain over all possible policies. These are the expected returns given our agent acts completely optimally. Our aim is to find a policy that achieves these optimal values i.e a policy π^* with the following property,

$$v_{\pi^*}(s) = v_*(s) \quad \forall s, \tag{1.6.3}$$

and a similar property involving action-values. Following this policy would mean we always have the largest expected return from every state and so it would be optimal. From the definition of v_*, it may be tempting to think that in order to find these optimal returns we may need to follow different policies at different states, due to this maximum over all policies for each state s. But if we follow different policies at different states, we can just combine these into a single policy that will give the maximum expected return for all states, meaning we do indeed search for a single policy π^* satisfying $v_{\pi^*}(s) = v_*(s)$.

Now, to compare how good different policies are, as given by Sutton [2, p. 75], we define a partial ordering of π and π' as

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \ \ \forall s. \tag{1.6.4}$$

In words, π is better than or equal to another policy π' if and only if the expected return from each state under π is greater than or equal to the expected return under π', a sensible definition. We now come onto an important theorem in the world of finite MDPs.

Theorem 1.6.5. *As stated by Puterman [6, Theorem 6.2.7]: for any finite Markov decision process with a discount factor $\gamma \in [0,1)$:*

- *there always exists at least one optimal deterministic policy, that is a policy π_* such that*

$$\pi_* \geq \pi \ \ \forall \pi.$$

When we stated our aim as finding an optimal policy that will always give the largest expected return from each state, it wasn't obvious that such a policy even exists. However, through this theorem we now know that not only does at least one exist, but there is a deterministic optimal policy, so π_* will simply map states to a single action. We can have multiple optimal policies, a simple case where this may occur is if two different actions transition you to the same state with the same reward. The expected return is the same (and is maximal) but optimal policy π_*^1 may choose one action and policy π_*^2 chooses the other. This theorem is quite powerful and we shall not only see a proof in the following chapter, but the proof is of the constructive type, meaning it will show us one method of how to find an optimal deterministic policy π_*. We now finish off this subsection and chapter by introducing another set of Bellman equations, this time showing recursive relationships of the optimal state-value and action-value functions. Again, we employ backup diagrams to help visualisation.

We begin at a state s in Figure 1.6.1 and want to calculate it's optimal state-value $v_*(s)$, the largest expected return we can gain from this state. Again, due to the dynamics of the MDP, we have a finite set of actions we can take and each state-action pair has their own optimal values, given by q_*. These tell us the greatest possible expected return after taking a specific action, which we treat as known. The best expected return from state s will therefore be achieved by taking the action that maximises this q_* function, as depicted by the arc in Figure 1.6.1. This gives

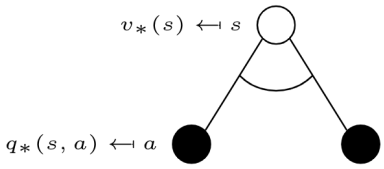


Figure 1.6.1: Partial v_* backup

$$v_*(s) = \max_a q_*(s, a). \tag{1.6.6}$$

Now we begin at a state-action pair (s, a) in Figure 1.6.2 and want to find it's optimal action-value $q_*(s, a)$. So now, as always, given a state and an action the environment can transition the agent to a variety of states and give a variety of rewards, outlined by the probabilities in the MDP. The agent has no control after taking an action. Each following state has their own optimal state-values which we assume known, as remember there is a sort of inductive reasoning going on in these equations. The action-value is, therefore, given by the expected reward due to this state-action pair plus an average over these optimal state-values, weighted by their transition probabilities. This gives

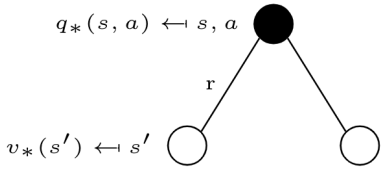


Figure 1.6.2: Partial q_* backup

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s'). \tag{1.6.7}$$

Now we can combine these two equations to give us the *Bellman optimality equation for* v_*,

$$v_*(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right). \tag{1.6.8}$$

Another similar recursive relationship for q_* holds by swapping the order in which these equations are combined. Again, as we shall see later on, there is only one set of values that satisfy this optimality equation, and so this equation can be treated as the definition of the v_* values. This recursive relationship can be viewed as a set of $|\mathcal{S}|$ equations in $|\mathcal{S}|$ unknowns, however now these are non-linear equations. In the following chapter we shall see common methods for calculating these optimal values and how we can in turn find optimal policies.

Example 1.6.9. We return back to our traffic light MDP for one final time and show its optimal state values, the expected return given the agent acted optimally from these states. Due to the simple deterministic setup of this MDP, finding these values is a basic calculation. It is clear to see that the optimal policy for our agent, in terms of gaining as large rewards as possible, is to always take the action 'wait'. From the state 'Red' always taking action 'wait' will give return +11 almost surely, whilst from state 'Red/Orange' taking action 'wait' gives return +10 almost surely. Thus, the expected returns from these states under an optimal policy are +11 and +10, as seen in Figure 1.6.3. These values are indeed greater than the state values of the previous policy in Figure 1.5.1, showing that the policy used there was not optimal.

We will now verify the the Bellman optimality equation (1.6.8) is satisfied for the state Red. Two actions are possible from this state, so we will need to take the maximum over two values. One action, 'drive', always causes the agent to gain reward -10, and transitions them to an absorbing state with an optimal state value of 0. The other possible action, 'wait', has expected

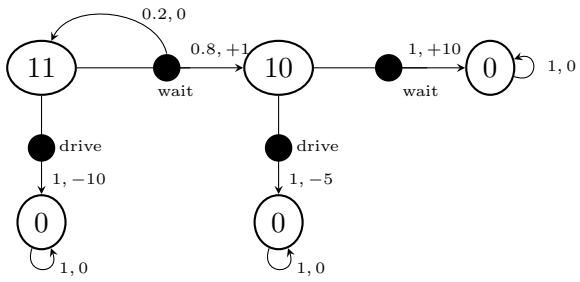


Figure 1.6.3: Traffic light MDP optimal state-values

reward 0.8, and transitions the agent either back to state 'Red', with optimal state value 11, or to 'Red/Orange' with optimal state value 10. Plugging this all into the Bellman optimality equation gives:

$$\begin{aligned} 11 = v_*(Red) &= \max_{a\in\mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right) \\ &= \max\{-10 + 1(1\times 0)\ ,\ 0.8 + 1(0.2\times 11 + 0.8\times 10)\} \\ &= \max\{-10\ ,\ 11\} \\ &= 11 \end{aligned}$$

So, as expected, the Bellman optimality equation does hold for this state, and all other states, and we see this recursive relationship between optimal state values.

1.7 Overview

This chapter can be seen as the all the necessary setup that will allow us to then study the rich theory of reinforcement learning. We saw that reinforcement learning is the general idea of learning how to act in sequential decision problems, problems where, overtime, different actions are required and we want to find out which ones are optimal. We saw that the mathematical framework underlying these sequential decision problems is a Markov decision processes. The main components of an MDP are a collection of states which our agent occupies, a collection of actions which our agent can take, and a collection of rewards our agent earns for different actions. Our aim is therefore to find a policy, something that tells our agent how to act, that is optimal in that it maximises the expected total future reward our agent will earn from each state. Practically all the following theory in this report will have this goal in mind. Finally, we introduced different value functions, v_π, q_π, v_* and q_*. The standard value functions tell us the expected return for our agent given a specific policy π, whilst the optimal value functions tell us the expected return given our agent is acting optimally. We derived multiple Bellman equations that these functions always satisfy and we shall see how we can use equations to find optimal policies in the following chapter.

Chapter 2

Dynamic Programming

We now tackle the problem of finding an optimal policy. As stated before, we use the term *dynamic programming* to mean the general theory of trying to achieve optimality when full knowledge of our MDP is known. This means the transition probability function $p : \mathcal{S} \times \mathscr{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is available, giving us the reward and state transition dynamics. There are two components to our task, using Sutton's terminology.

- *Prediction*: We are given an MDP and a policy π and we return the corresponding value function v_π
- *Control* We are given an MDP and we return an optimal policy π_*

We must first master *prediction* before we can undertake the slightly harder challenge of *control*.

2.1 Policy Evaluation

Policy evaluation is our solution to the *prediction* problem. We first remind ourselves of the *Bellman Equation (1.5.6) of* v_π,

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right).$$

This is the recursive equation that holds for any policy π and it's corresponding state values given by v_π. It turns out that by starting with arbitrary state-values, usually all 0, and then iteratively applying the Bellman equation (1.5.6) we converge onto the actual value function. More formally, we begin with a vector of arbitrary state-values, v_0, usually all chosen to be 0. At iteration $k + 1$, we create state-values v_{k+1} from v_k via the equation

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right). \tag{2.1.1}$$

One iteration is completed when this update is done for every $s \in \mathcal{S}$. So, for each state, in order to update its state-value we look at all possible succeeding states, find their current state-values, given by v_k, and plug these into the right hand side of Bellman equation (1.5.6) to give $v_{k+1}(s)$. To prove the convergence of this algorithm, we need to use the famous *Banach Fixed-Point Theorem*.

Theorem 2.1.2 (Banach Fixed-Point Theorem)**.** *As proved by Puterman [6, Theorem 6.2.3], given a normed vector space* $(V, \|\cdot\|)$ *and an operator* $L : V \to V$*, we say* L *is a contraction*

mapping if there exists a $\gamma \in [0,1)$ *such that for all* $u, v \in V$ *we have*

$$\|Lv - Lu\| \leq \gamma \|v - u\|. \tag{2.1.3}$$

In this case

- *there exists a unique* $v_* \in V$ *such that* $v_* = Lv_*$
- *for arbitrary* $v_0 \in V$ *the sequence* $\{v_k\}$ *defined by*

$$v_{k+1} = Lv_k$$

converges to v_*.

To use this theorem it is convenient to switch to matrix notation. Firstly, note that the Bellman equation of v_π (1.5.6) can be written equivalently as

$$\begin{aligned} v_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right), \forall s \\ \mathbf{v}^\pi &= \mathbf{r}^\pi + \gamma \mathbf{T}^\pi \mathbf{v}^\pi. \end{aligned} \tag{2.1.4}$$

Here $\mathbf{v}^\pi$ is the vector of all state-values, $\mathbf{r}^\pi$ is an $|\mathcal{S}|$-dimensional vector with jth entry $\sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_{s_j}^a$, for s_j the jth state. Finally, $\mathbf{T}^\pi$ is an $|\mathcal{S}| \times |\mathcal{S}|$ matrix whose (i,j) entry is $\sum_{a \in \mathcal{A}} \pi(a|s_i) \mathcal{P}_{s_i s_j}^a$.

Note that $\mathbf{T}^\pi$ is a right stochastic matrix, meaning its rows sum to 1. This can be seen by multiplication with $\mathbb{1} = (1, 1, \ldots, 1)^T$.

$$\begin{aligned} (\mathbf{T}^\pi \mathbb{1})_i &= \sum_{j=1}^{|\mathcal{S}|} \sum_{a \in \mathcal{A}} \pi(a|s_i) \mathcal{P}_{s_i s_j}^a \\ &= \sum_{a \in \mathcal{A}} \pi(a|s_i) \sum_{j=1}^{|\mathcal{S}|} \mathcal{P}_{s_i s_j}^a \\ &= \sum_{a \in \mathcal{A}} \pi(a|s_i) \\ &= 1, \end{aligned} \tag{2.1.5}$$

due to π and $\mathcal{P}$ being probability distributions. We deduce $\mathbf{T}^\pi \mathbb{1} = \mathbb{1}$ and are now ready to prove the convergence of policy evaluation.

Theorem 2.1.6 (Convergence of Policy Evaluation)**.** *Given a finite MDP with* $\gamma \in [0,1)$ *and a policy* π*, the policy evaluation algorithm described by equation (2.1.1) converges onto the true value function* v_π*.*

Proof. It is enough to show that the operator L defined by

$$Lv = \mathbf{r}^\pi + \gamma \mathbf{T}^\pi v \tag{2.1.7}$$

is a contraction mapping on the normed space $(V, \|\cdot\|_\infty)$ where $V : \mathcal{S} \to \mathbb{R}^{|\mathcal{S}|}$ is the set of bounded functions on $\mathcal{S}$ and $\|\cdot\|_\infty$ is the ∞-norm defined as

$$\|u - v\|_\infty = \max_{s \in \mathcal{S}} |u(s) - v(s)|. \tag{2.1.8}$$

Since then, by the fixed point theorem, from any starting vector of state-values we can iteratively apply the operator L and reach the vector $\mathbf{v}^\pi$ satisfying equation (2.1.4), i.e the actual state-values. Furthermore, due to the Banach Fixed-Point theorem we have that $\mathbf{v}_\pi$ is unique for a given MDP and policy π, so we know these are indeed the state-values.

Now we show that L is indeed a contraction mapping. As shown by Tom Mitchell [7, slide 12] we have that for arbitrary $v, u \in V$

$$\begin{aligned} \|Lv - Lu\|_\infty &= \|\mathbf{r}^\pi + \gamma \mathbf{T}^\pi v - \mathbf{r}^\pi + \gamma \mathbf{T}^\pi u\|_\infty \\ &= \|\gamma \mathbf{T}^\pi (v - u)\|_\infty \\ &\leq \|\gamma \mathbf{T}^\pi (\mathbb{1} \cdot \|v - u\|_\infty)\|_\infty \\ &= \|\gamma (\mathbf{T}^\pi \mathbb{1}) \|v - u\|_\infty \|_\infty \\ &= \gamma \|v - u\|_\infty. \end{aligned} \tag{2.1.9}$$

Hence, L is a contraction mapping and we can apply the Banach Fixed-Point theorem (2.1.2).

■

Example 2.1.3 (Gridworld)

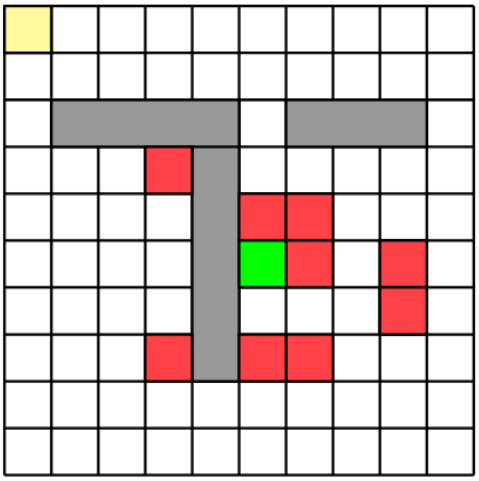


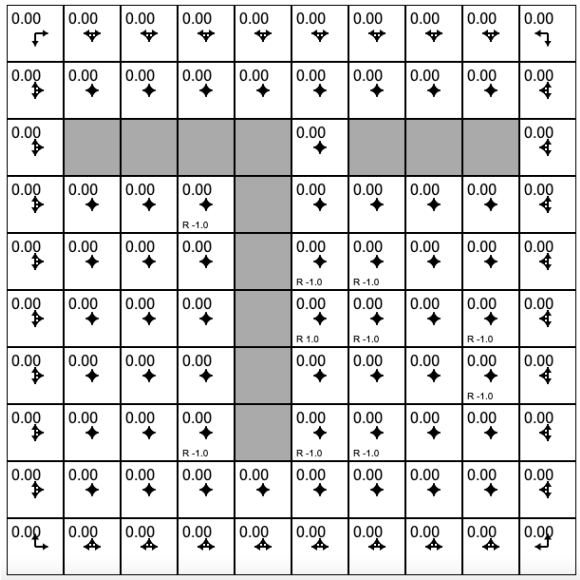
Figure 2.1.1: An example gridworld problem

We now introduce a simple gridworld problem and calculate the state-value function for the uniform probability policy. Consider the diagram of figure 2.1.1. This 10×10 grid denotes a simple problem our agent faces. The aim is to travel from the top left starting square to the green square via the shortest route possible. The grey squares are blocked off to the agent and the red squares are made of molten lava and should be avoided if possible. To a human, it is fairly obvious what paths are optimal but how could a computer solve this? One option is enumerating all possible paths, however, there are a lot of paths, an infinite number including paths that can crossover themselves. Whats more, knowing we have cycled through all possible paths isn't a trivial problem. Instead, we can formulate this problem into a deterministic MDP and use dynamic programming.

We let the states be the 88 squares the agent can move within. The actions are any legal moves of one square up, down, left and right, meaning the agent cannot leave the grid or go into a grey square. We want to avoid the dangerous red squares so we associate them with negative reward. In this example we give a reward of -1 for any action within one of these red squares, we shall see later on this slightly simplifies solving the problem, but equally we could give negative rewards for actions that lead into the red squares. As we want to reach the green square we give a reward of +1 for any action within the green square, and these transition the agent back to the starting square, meaning this MDP denotes a continuous task for the agent and the agent never stops traversing the gridworld. This is in comparison to the green state being an absorbing state that ends the task, causing an episodic MDP. All other actions gives 0 reward. We let the discount factor be $\gamma = 0.9$ and this will implicitly cause the agent to find the shortest path to the green square, as if it reaches it quicker the reward of +1 will be discounted less and so the return, the discounted sum of all future rewards, will be larger.

The initial policy is uniform, choosing each available action with equal probability and we want to find the state-values, the expected return from each state under this policy. To do so, we employ policy evaluation and iterate the Bellman equation, with the results shown in Figure 2.1.2 where state values are shown both numerically and by the colouring of the squares.

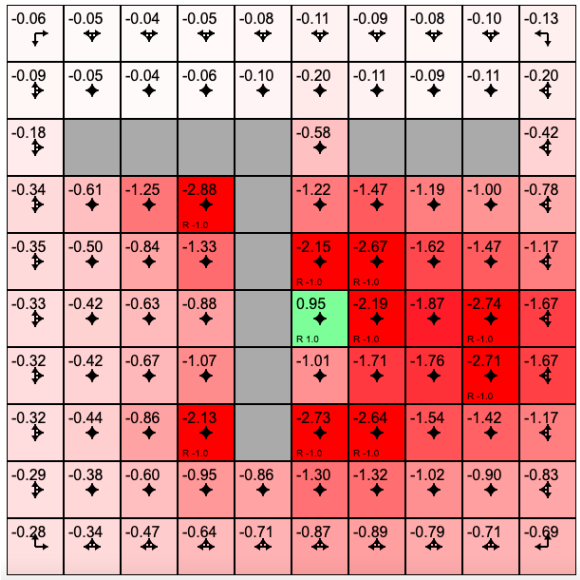
We now walk through the update of a single state-value in the first iteration of policy evaluation to emphasise how the algorithm works. Take the the green goal state, which we call s. Like all others it begins with state-value 0 and after one iteration we shall see it has state-value 1.



v_0

v_1

v_{20}



$v_{53} = v_\pi$

Figure 2.1.2: Finding the state-values for the uniform policy on the Gridworld via Policy Evaluation. Done through the website https://cs.stanford.edu/people/karpathy/reinforcejs/index.html Converges after 53 iterations.

Remember our update rule is

$$v_1(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_0(s') \right).$$

Our formation of the MDP means the available actions from this state are moves up, down and right, since we cannot move into a grey square. Each of these actions have probability $\pi(a|s) = 1/3$ due to the uniform policy π. Again due to the formation of our MDP the agent will receive a deterministic reward of $+1$ for each of these actions, and each action will transition the agent back to the starting top left square, which has current state value 0. So $\mathcal{P}_{ss'}^a = 1$ for

s' the starting state and for all three actions, and is 0 for all other states and actions. Plugging this all in gives,

$$\begin{aligned} v_1(s) &= \sum_{a\in\mathcal{A}(s)} \frac{1}{3}\left(1+\gamma(1\times 0)\right) \\ &= 1, \end{aligned} \tag{2.1.11}$$

matching the diagram. We see that once policy evaluation has converged onto the actual state-values the expected return from our starting state is -0.06, not very good but nothing more than we expect from a uniformly random policy.

2.2 Policy Improvement

Now we know the actual state-values, our aim is to try and improve on our policy π. Intuitively it is reasonable to favour actions that can send the agent to states with larger state-values as these have higher expected return. This can be described as acting *greedily* with respect to v_π. But care is needed, not only are large state-values important but the probabilities that we transition to these states given an action and the immediate reward received after taking the action all need to be taken into account. For example, let's say action a_1 gives reward $+10$ and transitions the agent to a new state with state-value 5. Meanwhile action a_2 has 0.999 chance of sending the agent to a new state with state-value 0.5 and gives reward 0, and 0.001 chance of sending the agent to a new state with state-value 20, giving reward 3. We see here that although action a_2 can send the agent to a larger state-value, we see that action a_1 is the better choice. Thankfully, we have already defined something that can help us, the action-value function q_π which tells us the expected return given you are in state s, take specific action a and follow policy π thereafter. Although defined by (1.5.3), it is more useful in its recursive form derived by the backup diagrams in Figure (1.5.2), since policy evaluation gives us the state-values v_π. We repeat this recursive form here,

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s').$$

From this we can create a new deterministic policy, π', that is defined by acting greedily with respect to q_π.

$$\begin{aligned} \pi'(s) &= \arg\max_{a\in\mathcal{A}} q_\pi(s,a) \\ &= \arg\max_{a\in\mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s'). \end{aligned} \tag{2.2.1}$$

Remember, since π' is a deterministic policy $\pi'(s)$ is the specific action taken in state s, rather than thinking of π' as a distribution. As mentioned before, acting greedily with respect to this means our agent is taking into account the succeeding state values, the probabilities of going to these states and the initial rewards gained, as seen in the equation above. If there are ties, we simply just chose one of the tied actions. Since the full MDP dynamics are known to us, and we have just worked out the value function v_π, we can indeed find this new policy π'. We now need to prove that this new policy is indeed better than our initial one, where we use the ordering of policies defined by (1.6.4). To do this, as mentioned but not proven by Sutton [2, p. 94], it's first helpful to see that given any state s, it is better (larger expected return) to follow the deterministic policy π' for one action and then follow π thereafter, then just to follow our initial policy π. Using our notation this means we want to show

$$q_\pi(s,\pi'(s)) \geq v_\pi(s) \tag{2.2.2}$$

where $\pi'(s)$ is the deterministic action taken in state s. We quickly show this here,

$$\begin{aligned} v_\pi(s) &= \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a) \\ &\leq \sum_{a\in\mathcal{A}} \pi(a|s) [\max_{a'\in\mathcal{A}} q_\pi(s,a')] = \max_{a\in\mathcal{A}} q_\pi(s,a) \\ &= q_\pi(s, \arg\max_{a\in\mathcal{A}} q_\pi(s,a)) = q_\pi(s,\pi'(s)). \end{aligned} \tag{1.5.7}$$

We have used the fact that π is a probability distribution and so summing over the actions gives value 1. Intuitively, if at any state s it is better to first follow policy π', then policy π, we may as well always follow policy π' since when we reach the next state s', again this tells it is better to first follow π' for one step, and this can be repeated infinitely. This reasoning is formalised in the *Policy Improvement Theorem* which is stated and proved by Sutton [2, p. 95] for a deterministic new policy π'. We shall slightly generalise the proof to include stochastic policies, as later chapters require this. To do so, since the action $\pi'(s)$ is only defined for deterministic policies, we use Sutton's definition,

$$q_\pi(s,\pi'(s)) = \sum_a \pi'(a|s) q_\pi(s,a).$$

So, the expected return from taking an action from policy π' and then following policy π thereafter is the average of q_π action-values weighted by the probability of choosing these actions under π'.

Theorem 2.2.3 (Policy Improvement Theorem)**.** *Given two policies π and π', if we have that for all $s \in \mathcal{S}$*

$$q_\pi(s,\pi'(s)) \geq v_\pi(s)$$

then $\pi' \geq \pi$. This means for all states the expected return following π' is at least the expected return following π,

$$v_{\pi'}(s) \geq v_\pi(s), \forall s \in \mathcal{S}.$$

Proof. Firstly, note an alternative way of writing $q_\pi(s,\pi'(s)$ explicitly as an expectation.

$$\begin{aligned} q_\pi(s,\pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s,a) \\ &= \sum_a \pi'(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right) \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \end{aligned} \tag{1.5.8}$$

Now by repeated application of this expectation and the assumed inequality (1.6.4), we have

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s,\pi'(s)) \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}] | S_t = s]. \end{aligned}$$

Now the law of total expectation states $\mathbb{E}[\mathbb{E}[X|Y]] = \mathbb{E}[X]$, therefore,

$$\begin{aligned} &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2})] | S_t = s] \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] \end{aligned}$$

$$
\begin{aligned}
&\vdots \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) | S_t = s] \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots | S_t = s] \\
&= v_{\pi'}(s).
\end{aligned}
$$

■

So we have now found a new policy π' that is indeed an improvement on our initial policy π, since our definition of π' satisfies the inequality (2.2.2) by design, as shown before.

Example 2.2.4 (Policy improvement in the gridworld)

Recall the gridworld problem which we setup in example (2.1.10). We will now apply policy improvement on this problem to find a better policy for the maze. It was mentioned before that with the way that our MDP was setup, giving rewards of -1 for any action within a red square and rewards of $+1$ for any action within the green square, simplified solving this problem. The reason why is that under this setup, given any state s, the reward gained from the possible actions is always the same i.e $\mathcal{R}_s^a = c$ is constant for all a. This, combined with the fact our MDP is completely deterministic meaning given a state and action our agent will transition to a single new state with probability one, means the new greedy policy π' can simply act greedily towards the v_π values. More formally, by definition of the new policy π' (2.2.1),

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s').$$

Now given for any action a,

$$\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') = c + \gamma v_\pi(s'(a))$$

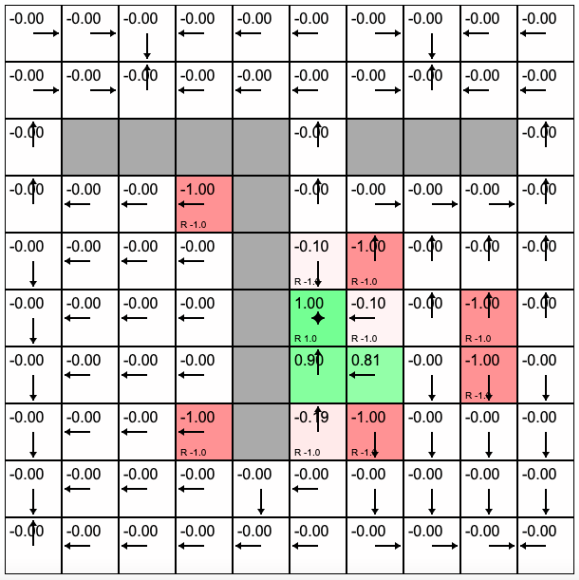
for $s'(a)$ the deterministic new state our agent will transition to and c the constant reward the agent receives for any action in this state. Therefore,

$$
\begin{aligned}
\pi'(s) &= \arg\max_{a \in \mathcal{A}} c + \gamma v_\pi(s'(a)) \\
&= \arg\max_{a \in \mathcal{A}} v_\pi(s'(a)).
\end{aligned}
$$

We conclude that the new policy π' simply chooses the action that leads to the largest state-value of the possible succeeding states, whilst in general it is a slightly more complex procedure requiring greediness towards q_π values. Now from our previous example, we left off with the state-value function v_{π_0} for our initial uniform random policy π_0. In Figure 2.2.1 we see the new greedy policy π_1 derived from these values.

We can see the policy π_1, which is denoted by the arrows on the second diagram, is given by acting greedily towards the v_{π_0} values. For example, from the starting top left state our new policy π_1 tells us to go right one square. This is because the subsequent v_{π_0} values are -0.05 and -0.09 for actions right and down respectively, so choosing the larger action greedily tells us to go right. The state-values of v_{π_1} are then found via policy evaluation as normal and from the diagrams we do see that $v_{\pi_1}(s) \geq v_{\pi_0}(s) \ \ \forall s$. Meaning we have indeed found an improved policy, although clearly it still isn't great. From the starting state our agent will always get trapped in a loop between two squares getting 0 reward each time, but this is better then the expected negative return from our starting state due to our initial policy. There is still work to be done, and this leads us into the next section.

π_0 & v_{π_0}



π_1 & v_{π_1}

Figure 2.2.1: Improving a uniform random policy in the gridworld. https://cs.stanford.edu/people/karpathy/reinforcejs/index.html

2.3 Policy Iteration

Policy iteration is culmination of our dynamic programming chapter and our solution to the *control* problem. Following on from our previous sections, we can now repeat this procedure, undertaking policy evaluation on our new policy π' to find $v_{\pi'}(s)$, then using policy evaluation to again improve on policy π'. This gives us a sequence of policies and value-functions $\{\pi_k, v_{\pi_k}\}_{k\geq 1}$. We call this *Policy Iteration*. However, there still remains the question whether this procedure will converge onto any sort of optimality, in both the value-functions and the policy. Thankfully, it does. Filling in the gaps of Sutton [2, p. 95 - 96], we first consider what would happen if in a cycle of policy iteration our new policy π' didn't strictly improve on π but stayed of equal value, that is

$$\pi' = \pi \iff v_{\pi'}(s) = v_\pi(s) \ \ \forall s.$$

Then by our definition of the new policy π' (2.2.1), and since $\pi = \pi'$, we now have

$$\begin{aligned} \pi'(s) &= \arg\max_{a\in\mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \\ &= \arg\max_{a\in\mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_{\pi'}(s'). \end{aligned}$$

By the *Bellman equation of* $v_{\pi'}$, (1.5.6) we have the value of state s satisfies the following recursive relationship,

$$v_{\pi'}(s) = \sum_{a\in\mathcal{A}} \pi'(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_{\pi'}(s') \right). \tag{2.3.1}$$

But we remember that π' is a deterministic policy and specifically chooses action $\pi'(s) = \arg\max_{a\in\mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_{\pi'}(s')$, as shown before. Therefore,

$$v_{\pi'}(s) = \max_{a\in\mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_{\pi'}(s') \right).$$

But we notice now that this is the *Bellman optimality equation for* v_* (1.6.8)! If we can show there is only one unique vector that fits this equation we have then found the optimal state-values.

Proposition 2.3.2 (Uniqueness of v_*)**.** There is only one value function that obeys the Bellman optimality equation for v_*,

$$v_*(s) = \max_{a\in\mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right)$$

Proof. As shown by Sompolinsky [8, p. 11 - 12] we assume there exists two value functions that obey this equation and try to find a contradiction. Let v_1 and v_2 be such that $v_1 \neq v_2$ and

$$\begin{aligned} v_1(s) &= \max_{a\in\mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_1(s') \right), \\ v_2(s) &= \max_{a\in\mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_2(s') \right), \\ v_2(s) &= v_1(s) + e(s). \end{aligned}$$

Now as $v_1 \neq v_2$ we have that $e(s)$ is non zero for at least one s, and we define s^* as the maximal argument of $|e(s)|$,

$$s^* = \arg\max_s |e(s)|.$$

We let $e(s^*) > 0$ without loss of generality since we could swap v_1 and v_2. Now, we see that

$$\begin{aligned} v_2(s^*) &= \max_{a\in\mathcal{A}} \left(\mathcal{R}_{s^*}^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{s^*s'}^a v_2(s') \right) \\ &= \max_{a\in\mathcal{A}} \left(\mathcal{R}_{s^*}^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{s^*s'}^a (v_1(s') + e(s')) \right) \\ &\leq \max_{a\in\mathcal{A}} \left(\mathcal{R}_{s^*}^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{s^*s'}^a v_1(s') \right) + \gamma \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \mathcal{P}_{s^*s'}^a e(s') \\ &\leq v_1(s^*) + \gamma e(s^*) \\ &< v_1(s^*) + e(s^*). \end{aligned} \tag{2.3.3}$$

But we have that $v_2(s^*) = v_1(s^*) + e(s^*)$, a contradiction. We deduce that $v_1 = v_2$ and so there is only one value function obeying the optimality equation.

■

We can now finally clear everything up and bring together policy iteration, thus proving theorem (1.6.5). We have seen via *policy iteration* that if in a cycle our new policy didn't improve on our previous policy but stayed of equal value then we have found the optimal value function

that satisfies the Bellman optimality equation. But is this guaranteed to happen and is the corresponding policy an optimal policy? The answer to both of these questions is yes.

Firstly, apart from our initial arbitrary policy π_0, all succeeding policies were deterministic. Since in a finite MDP there is only a finite number of deterministic policies, our policies will have to stop improving at some point. We cannot have $\pi_k < \pi_{k+1}$ for all k as this would give an infinite number of deterministic policies, so at some point we have to have $\pi_k = \pi_{k+1}$ meaning our sequence $\pi_0 \to v_{\pi_0} \to \pi_1 \to v_{\pi_1} \to \cdots \to \pi_*$ converges, with v_* the corresponding value function. Furthermore, since we can start from any arbitrary policy π_0, we do indeed have this policy π_* is optimal. If there existed a better policy, starting from this we would converge onto the improved π_* - a contradiction. Thus we have proven Theorem (1.6.5) and conclude our chapter on dynamic programming. As mentioned before, reinforcement learning is really about solving problems when the exact dynamics of the MDP are unknown to us. We, therefore, will move onto exciting chapters that will allow us to drop an agent into completely unknown conditions and learn to act optimally.

Example 2.3.1 (Solving the gridworld)

We continue on from Example 2.2.4, as policy iteration tells us to do, by evaluating value functions then acting greedily towards them to produce a new policy. By the 9th cycle we get the optimum policy and corresponding state-values.

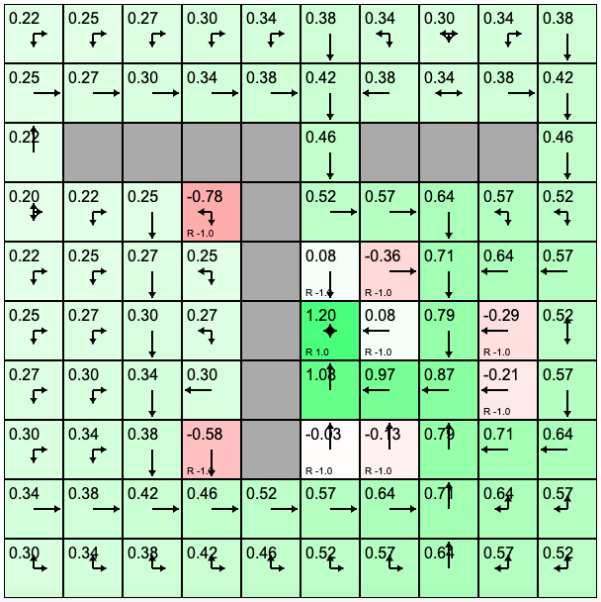


Figure 2.3.1: Gridworld solved in the 9th iteration of policy iteration. https://cs.stanford.edu/people/karpathy/reinforcejs/index.html

Starting from the initial top left square there are multiple paths denoting different optimal policies, each arriving at the goal square in the shortest number of moves. This is due to some optimal state values having equal value. The expected return from our initial state when acting optimally is about 0.22, which we can verify is the correct value analytically as the return is deterministic when taking the shortest possible paths. From this initial starting state the shortest paths to the end goal take 16 moves and so we receive the reward of $+1$ after 17 moves when the agent is transitioned back to the start state. So every 17 moves the agent will receive this reward if acting optimally, and the reward will be more severely discounted each time as it is earned further in time. By the definition of return (1.4.2) from the starting top left state S_0, we have that

$$\begin{aligned} G_0 &= \sum_{k=1}^{\infty} \gamma^{16k} \times 1 \\ &= \frac{0.9^{16}}{1 - 0.9^{16}} \\ &\approx 0.22 \end{aligned}$$

2.4 Overview

This chapter has shown us that when we have full knowledge of the MDP at hand, dynamic programming can always be used to find an optimal policy for our agent. *Policy evaluation*

allowed us to, when given a policy, evaluate its state values to give v_π. *Policy improvement* then allowed us to, given an initial policy and its state value function, find a new improved policy that acts greedily towards the action value function, q_π. This policy was improved in the sense that in every state, the expected return for our agent was at least as great. Finally, *policy iteration* combined evaluation and improvement to give a sequence of improving policies that always converges to a final optimal policy.

Although powerful, the main problem is that knowing the exact transition and reward dynamics is a big ask. As we shall see in later chapters, often the MDP depicting the task at hand is simply too complex to know these dynamics, rendering the ideas of dynamic programming useless. Dynamic programming allowed us to find an optimal policy for our agent without ever actually experiencing the MDP, due to this complete knowledge. We shall see that in the upcoming chapters, now we do not have this knowledge, our agent must actually experience and play the MDP to allow them to learn optimal policies. Although this is quite a big difference, we shall also see that the ideas of policy evaluation and policy improvement carry nicely into the next chapters, and we can really build on what we have achieved in this chapter.

Chapter 3

Model-Free with Monte Carlo

We now begin to explore the theory of *model-free* methods. This means, although there exists an MDP perfectly describing the problem our agent faces, we don't know the transition probability function p telling us the probability of going from state s to s' and receiving reward r given an action a. The agent will still know the state they exist in, their possible actions and observe the rewards received from the environment, but this is the entirety of the information at hand. From this the agent must learn to act optimally. As mentioned before, the main ideas of dynamic programming extend nicely into the model free methods. Namely, *prediction*, where we are given a policy and want to evaluate how good it is, and *control*, where we try to find an optimal policy.

3.1 Monte Carlo Methods

We begin by recapping what Monte Carlo methods entail and why they work. The basic idea is using empirical means to estimate the true expected value. Expected values are hugely useful, whether this be in statistics when we use unbiased estimators of certain parameters, or, in quantum mechanics due to the probability densities that govern the position of particles. Although the definition of expectation is clear, and it's values are deterministic, often we cannot calculate expectations analytically. The underlying probabilities may be too complicated or the distributions may simply be unknown. To overcome this we use Monte Carlo methods with repeated simulation. The main mathematical theory underlying Monte Carlo is the well known *law of large numbers*.

Theorem 3.1.1 (Strong Law of Large Numbers)**.** *As proven by Shiryaev [9, p. 391 - 393], let $X_1, X_2, \ldots$ be an infinite sequence of independent and identically distributed random variables such that $\mathbb{E}[|X_1|] < \infty$. Further, let $\overline{X}_n$ be the average of the first n random variables. Then,*

$$\overline{X}_n \xrightarrow{a.s} \mu \quad as \quad n \to \infty.$$

So this tells us that we don't need to explicitly calculate an expected value if we can instead simulate many independent instances of a random variable and calculate their average. Due to the stochastic nature of MDPs and the fact our main aim throughout this project has been to maximise future expected rewards, we shall see Monte Carlo methods lend themselves very nicely to reinforcement learning.

3.2 Monte Carlo Prediction

Let us first recap a few things. Suppose we are given a policy π, which is a conditional probability distribution of actions given a state. We want to know how 'good' this policy is, meaning we

want its value function,

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s].$$

This is the expected return (total discounted reward) the agent will receive given they are in state s and, from there, follow policy π. Before, in dynamic programming, to calculate these values we had to compose recursive relationships between successive states via the Bellman equation. Now we can just use Monte Carlo methods to estimate these state values by simulating many episodes of our MDP and averaging the return values our agent observes, converging almost surely onto the true values by the law of large numbers . This all sounds well and good, however, we see one of the major problems in using Monte Carlo methods is that we need to simulate episodes, meaning our tasks need to be episodic and cannot be continuous, as defined in Section 1.4. In continuous tasks, the returns are infinite sums that cannot be computed and so the methods break down. We require episodes to be fully completed before we can update any state-values, since G_t is the sum of all future rewards after being in state S_t, and so cannot be computed till every reward is seen. However, with each action the agent learns valuable new information, in the form of a reward, that surely should be taken into account instantly. Both of these negatives can be overcome in methods that we shall see later on.

Before we formalise how Monte Carlo prediction works in practise, we first introduce a neat way of calculating incremental means, as introduced by David Silver [10, Lecture 3, slide 10]. Let $\overline{x}_n$ be the mean of $x_1, x_2, \ldots, x_n$. The following relationship can easily be verified to show holds between successive means,

$$\overline{x}_n = \overline{x}_{n-1} + \frac{1}{n}(x_n - \overline{x}_{n-1}). \tag{3.2.1}$$

This incremental mean setup is used for two reasons. Firstly, calculating averages this way means we don't have to keep track of the sum of $x_1, \ldots, x_n$ as this is implicitly tracked by the mean values, which means slightly less computation as we continually calculate means. Secondly, remember that the mean values, $\overline{x}_n$, are an estimate of some expected value. Equation (3.2.1) is therefore showing how we are updating our current estimate, $\overline{x}_{n-1}$, by an error term, $x_n - \overline{x}_{n-1}$. This idea of having an estimate, seeing new data and updating our estimates towards this new data crops up again and again in many algorithms and so it is useful to introduce this concept here.

Now, consider a general episode that our agent experiences when following policy π,

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots, R_T. \tag{3.2.2}$$

Using Silver's notation, we let $N(s)$ be the number of times the agent visits state s and $V(s)$ be our current estimate of the state-value, $v_\pi(s)$. We initialise our estimates as 0 for all states. After seeing this episode be played out we can then update our state-values. At time step t the agent occupied state S_t and consequently received return G_t. Therefore, for each time step we:

1. Increase the state counter by 1
   - $N(S_t) \leftarrow N(S_t) + 1$
2. Update our estimate of expected return from that state via the incremental update formula
   - $V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}[G_t - V(S_t)]$

The backwards arrow notation used here is a sort of pseudo code which denotes the assignment of a variable. This allow us to more clearly write these updates above, instead of, for example, using $V_k(s)$ for the kth update of our estimate of $v_\pi(s)$. $V(s)$ is keeping a track of the mean

value of the returns seen from each state, and so by the *law of large numbers* we have that as we visit state s more and more,

$$V(s) \xrightarrow{a.s} v_\pi(s).$$

So we have indeed solved the prediction problem with Monte Carlo, but what about the harder control problem? This brings use to our next section, and we shall it isn't as plain sailing as dynamic programming.

3.3 Problems with Monte Carlo control

Recall the general ideas in policy iteration from the chapter in dynamic programming. We evaluate a policy's state-values, improve on this policy by creating a new greedy policy based on these values, and rinse and repeat until we reach the optimum policy. We have just seen Monte Carlo methods allow estimation of the value of our policy, giving v_π almost surely in the limit. However, we shall now see our ideas break down when creating our new greedy policies. There are two major problems we face, as Silver discusses [10, Lecture 5, slides 8 - 10]. So far, we have been focusing greatly on the state-values given by v_π, and our methods have relied on storing these values as we go on, which in turn allowed the definition of our improved π' as

$$\begin{aligned} \pi'(s) &= \arg\max_{a \in \mathcal{A}} q_\pi(s,a) \\ &= \arg\max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s'). \end{aligned}$$

Given we knew the v_π values, we can do a one step look ahead to all possible successive states along with the expected reward $\mathcal{R}_s^a$ and transition probabilites $\mathcal{P}_{ss'}^a$ to find this argmax. But remember, we are now in the realm of model-free methods, meaning we don't know these expected rewards or transition probabilities to plug into this equation and so we cannot form this new policy. However, as the equation above hints at, if we instead store state-action values directly, we skip the need for these dynamics of the MDP. Recall the definition of $q_\pi(s,a)$ (1.5.3), it tells us, given a state and a specific action, the expected return our agent will gain given they follow policy π thereafter. The Monte Carlo methods we covered before can easily transfer to evaluating q_π.

At time step t the agent occupies state S_t, takes action A_t and consequently receives return G_t. Therefore, after the episode finishes, for each time step we:

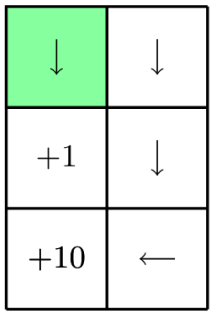
1. Increase the state-action counter by 1
   - $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
2. Update our estimate of expected return from this state-action pair via the incremental update formula
   - $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}[G_t - Q(S_t, A_t)]$

By the usual Monte Carlo theory, we have that as we visit the state-action pairs more and more our estimates of the state-action values, given by Q, will converge almost surely onto the actual q_π values, and thus we can now create our new greedy policy, $\pi'(s) = \arg\max_{a \in \mathcal{A}} q_\pi(s,a)$.

However, this leads to another problem, which is one of *exploration*. In order for these Monte Carlo methods to work, our agent needs to keep visiting every different state-action pair again and again for our estimates, Q, to converge almost surely onto all the real values, q_π. But by playing random episodes of our task this isn't guaranteed. There may be scarce state-action pairs that our agent rarely observes, or due to our current policy our agent may never reach some certain states. We then cannot update these values and thus policy iteration fails when

defining a new policy greedily based on the Q values. The problem of *exploration vs exploitation* discussed in Section 1.1 is prevalent here. Currently our methods cause the agent to largely *exploit* what they know by defining the policy to act completely greedily towards the Q values, and thus is lacking on the *exploration*, we need more balance!

Example 3.3.1 (The Monte Carlo exploration problem)



Take this gridworld on the left. Our agent starts in the green square and there are two terminal states giving rewards of +1 and +10. We have that $\gamma = 1$, so this is an undiscounted episodic MDP. We have an initial deterministic policy π, denoted by the arrows on the grid that clearly isn't optimal as the agent only gets reward +1. If we tried to do policy iteration via Monte Carlo methods we would first let all state-action values be 0 and play episodes, updating the estimates after each episode. Due to our initial policy, in each episode the agent will move down and receive reward +1 each time, and so our Monte Carlo estimate of the value of this state-action pair converges to 1, the true value. However, since our agent never takes the action right from the initial starting state our Monte Carlo estimate for this state-action pair stays at 0, whilst the true value is 10, since following the policy after initially going right always gives return 10.

Episode	(state, action)	MC estimate - $Q(s,a)$	True value - $q_\pi(s,a)$
0	(start, down)	0	1
≥ 1	(start, down)	1	1
0	(start, right)	0	10
≥ 1	(start, right)	0	10

So if we use these Q values to give an updated greedy policy π' we see that we don't actually improve on this sub-optimal policy,

$$\begin{aligned} \pi'(start) &= \arg\max_{a \in \mathcal{A}} Q(start, a) \\ &= down. \end{aligned} \tag{3.3.2}$$

The new policy tells the agent to again take action down from the starting state, and this will keep happening again and again, never giving an optimal policy. The key point is that the theory of dynamic programming meant policy evaluation always converged on the state-values for every single state, whilst with MC we see that isn't the case as the agent isn't guaranteed to keep trying each state-action pair, and so our ideas break down. So how can we try and fix this? We need to keep some essence of exploration within the agent as this completely greedy way of thinking clearly fails.

3.4 Exploration vs Exploitation

Our main idea to tackle the problem of *exploration vs exploitation* will be ϵ-greedy methods, a simple idea, given by Suttonn [2, p. 125] that allows continual exploration for our agent. Our agent needs to keep on exploring many different states and actions to learn and update different action values. But also, our agent must exploit what they think are good actions to try maximise return. The solution to finding this balance is to define our policies so that with probability $1 - \epsilon$ the agent chooses an action greedily with respect to the Q values and with

probability ϵ chooses a completely random action. ϵ is usually chosen to be small with $\epsilon = 0.1$ a fairly common value. Mathematically our policy π is now,

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{n} + 1 - \epsilon & \text{if } a = \arg\max_{a \in \mathcal{A}} Q(s,a) \\ \frac{\epsilon}{n} & \text{else} \end{cases}$$

where n is the number of possible actions in state s. This random action can be the argmax of our Q function, which causes the $\frac{\epsilon}{n}$ to be seen in both parts of this policy definition, and if there are tied values for this argmax we let the $1-\epsilon$ probability be split uniformly between these tied values. So now, because of this non-zero chance of choosing an action randomly, our agent will in theory visit each state-action pair and so by simulating many episodes we can theoretically converge onto q_π, allowing us to then define a new π' as usual. It should be stated, however, that due to this process converging onto q_π only in a theoretical limit often we don't wait until convergence onto the true action value function before updating our new policy. In practice, we update our policy after every single episode, using the new information we gain as quickly as possible. So our ϵ-greedy algorithm in full is:

1: Initialise all action values as 0 and set initial policy as uniform random
2: **repeat**
3: Play a single episode of the MDP under current policy
4: **for all** time steps in previous episode **do**
5: $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
6: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}[G_t - Q(S_t, A_t)]$
7: **end for**
8: Form new ϵ-greedy policy from updated Q values

Example 3.4.1 (Simple Monte Carlo gridworld)**.**

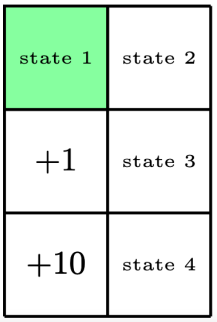


Figure 3.4.1: Simple gridworld with labelled states

We return to this simple gridworld from Example 3.3.1 and show that, now, with an ϵ-greedy strategy we overcome the exploration problem witnessed before. My code that created this gridworld environment and implemented an ϵ-greedy strategy can be found in the file 'small_gridworld_eps_greedy.py' on my GitHub page [11]. We play around 100 episodes to see how our agent learns, and then play 100,000 episodes to see the Q values this process converges on. Remembering our agent starts at in the green square, state 1, we initialise all Q values as 0 and begin the algorithm. Acting greedily towards these initial and equal Q values causes the agent to act uniformly random. By chance, our first episode shows the agent take the following sequence of actions: right, left, down. This ends with a reward of +1, and so the sampled return for each state action pair visited is 1. Our updated Q values are, therefore, as follows:

	State			
Action	1	2	3	4
Up			0	0
Down	1	0	0	
Left		1	0	0
Right	1			

So, given these Q values and since $\epsilon = 0.1$, from our starting state the agent has two actions available, each with Q value 1, and so will just pick uniformly randomly between them. If the

agent chooses action 'right' from state 1, leading to state 2, the possible actions then are 'left' or 'down' with Q values 1 and 0, respectively. Therefore, 90 percent of the time the agent will choose action 'left' in this state, returning to state 1, and 10 percent of the time will choose a uniformly random action. If our agent acted solely greedily towards these current Q values, like in the previous example, our agent would get stuck in this local minimum, only gaining reward +1 each time. The next 20 episodes have our agent, by chance, only choosing actions that maximise the Q function, staying solely in states 1 and 2 until they receive reward +1. This keeps the table of Q values constant, since the returns are always +1, giving an average of 1. On the 21st episode our agent, by chance, takes the following sequence of actions, 'right', 'down', 'down', 'left', with probability $0.5 \times 0.1 \times 0.33 \times 0.5 = 1/120$. The agent gains reward +10 and has now visited new state-action pairs, giving a new Q table of:

	State			
Action	1	2	3	4
Up			0	0
Down	1	10	10	
Left		1	0	10
Right	1.36			

In the first 20 episodes our agent visited the state action pair (1, right) a total of 24 times, each giving a return of +1. The average return after this 21st episode is therefore $\frac{1}{25}(24 \times 1 + 1 \times 10) = 1.36$, as seen in the table above. From our initial state, acting greedily towards the Q values now causes the agent to take the optimal path through the gridworld. The caveat is that there is now a slight chance of taking a random, often sub optimal, action. After 100,000 episodes the Q values converge onto the following values.

	State			
Action	1	2	3	4
Up			9.51	9.66
Down	1	9.67	9.98	
Left		9.14	1	10
Right	9.65			

There are more sophisticated ideas that try solve the problem of *exploration vs exploitation*, for example, Thompson sampling, as described in Wikipedia [12], that uses Bayesian statistical thinking. However, ϵ-greedy strategies are nice and simple, both theoretically and in implementation. They allow us to exactly define a probability distribution for new policy π, have elegant mathematical properties and extend into methods seen in later chapters. Because of all this, it is our method of choice.

3.5 Multi-armed Bandits

We now investigate an example comparing an ϵ-greedy strategy to Thompson Sampling with *multi-armed bandits*. As Russo et al. mention [13], Thompson sampling describes a general method of decision making that is effective across a variety of sequential decision problems. Different tasks can have very different reward dynamics and action spaces, and so due to the Bayesian approach taken by Thompson sampling, this corresponds to different prior and posterior distributions related to the rewards for these differing tasks. Thompson Sampling describes the general process of handling these prior and posterior distributions once they already chosen, rather than how to actually choose them, and in turn tells the agent which action to take. Since our main focus will be on ϵ-greedy methods we do not go into detail talking about the

general ideas of Thompson sampling, and instead describe the process fully in the specific case of multi-armed bandits.

So what is the multi-armed bandit problem? Consider a gambler in a room with n slot machines (or bandits as we call them). The rewards given by each bandit follow different unknown probability distributions and the agent must try to maximise the total reward they earn. This is a very famous example that highlights the problem of *exploration vs exploitation* very well. Our agent needs to explore to find which bandit will give the best rewards otherwise they may take sub-optimal actions, but they must also exploit what they believe to be better rewarding machines to maximise return. In this simple scenario we let $n = 4$ and each bandit gives rewards that follow a Bernoulli distribution with success θ_k. So with probability θ_k the kth machine will give a reward of 1 and with probability $1 - \theta_k$ will give zero reward. The specific θ values are 0.25, 0.3, 0.35 and 0.4 for machines 1, 2, 3 and 4, respectively, so the 4th bandit is the optimal machine to use. In an MDP setup this means there are two states, an initial starting state with 4 different actions corresponding to the different machines, and these all transition the agent to a terminal absorbing state. An episode of this problem is therefore just a single spin of one machine.

We first talk through the ϵ-greedy algorithm. Under this setup, our estimates of the action-values, the expected return given an action, will simply be the proportion of successful spins of each machine since the rewards are either 1 or 0. As the number of spins of each machine tends to infinity these values would converge almost surely onto the true θ values, via the law of large numbers. So for each episode, as the ϵ-greedy strategy algorithm tells us, our agent will, with probability 1-ϵ, choose the bandit with the largest Q value. This corresponds to the bandit with the largest proportion of successful spins so far. With probability ϵ the agent will simply choose a random bandit. After an episode, only one Q value will be updated as only one machine is spun per episode, and the process repeats. The fact that ϵ is constant means that these methods are susceptible to sub-optimal actions being taken. If ϵ is high our agent will explore more and be more likely to find which actions are optimal, but will cause the agent to take random actions often, which are rarely optimal. On the other hand, with a low ϵ the agent won't explore much and it may take the agent a huge number of episodes to have an accurate perception of the reward probabilities. This can cause the agent to often take sub optimal actions which they believe are optimal. The choice of ϵ generally needs to be fine tuned for each specific problem and isn't a trivial process. A value of $\epsilon = 0.1$ was empirically found to be an effective middle ground, as shown in the upcoming graphs of Figure 3.5.2.

We now look at Thompson sampling in the Bernoulli bandit problem, following the approach used by Russo et al. [13, p. 13 - 14]. As stated before Thompson sampling takes a Bayesian approach in choosing which machine to use. In the context of bandits, this means our agent has distributions for the θ values, which are the proportion of successes each machine gives, that are updated each time actions are took. Since the rewards are Bernoulli(θ_k), the mathematically convenient conjugate prior distribution of the Beta distribution is used for these θ values. Given a prior distribution for the θ_k of Beta(α_k, β_k), if we spin machine k and observe reward $r \in \{0, 1\}$, our posterior distribution will then be Beta($\alpha_k + r, \beta_k + 1 - r$). We begin with $(\alpha_k, \beta_k) = (1, 1)$ for all k, which is equal in distribution to a Uniform(0,1), since we have no information of the θ values. Following the update rule described just before, this means at any one point our distribution of θ_k will be Beta($1+\alpha, 1 + \beta$) for α the number of successes and β the number of failures of machine k. For $X \sim \text{Beta}(1 + \alpha, 1 + \beta)$, we have that

$$\begin{aligned} \mathbb{E}[X] &= \frac{1+\alpha}{1+\alpha+1+\beta}, \\ \mathbb{V}\text{ar}[X] &= \frac{(\alpha+1)(\beta+1)}{(\alpha+1+\beta+1)^2(\alpha+1+\beta+1+1)}. \end{aligned} \tag{3.5.1}$$

The expectation can be thought as a current guess of θ_k, and we see it is approximately the proportion of successful spins, $\frac{\alpha}{\alpha+\beta}$. Furthermore, as the number of spins increases in a machine the variance tends to 0, since the denominator of the variance grows quicker than the numerator. Various posterior distributions are shown in Figure 3.5.1. We see, in agreement with these expectations and variances, the peaks of the distributions are close to the proportion of successful spins and the distributions become more concentrated around their peak as the number of spins increases.

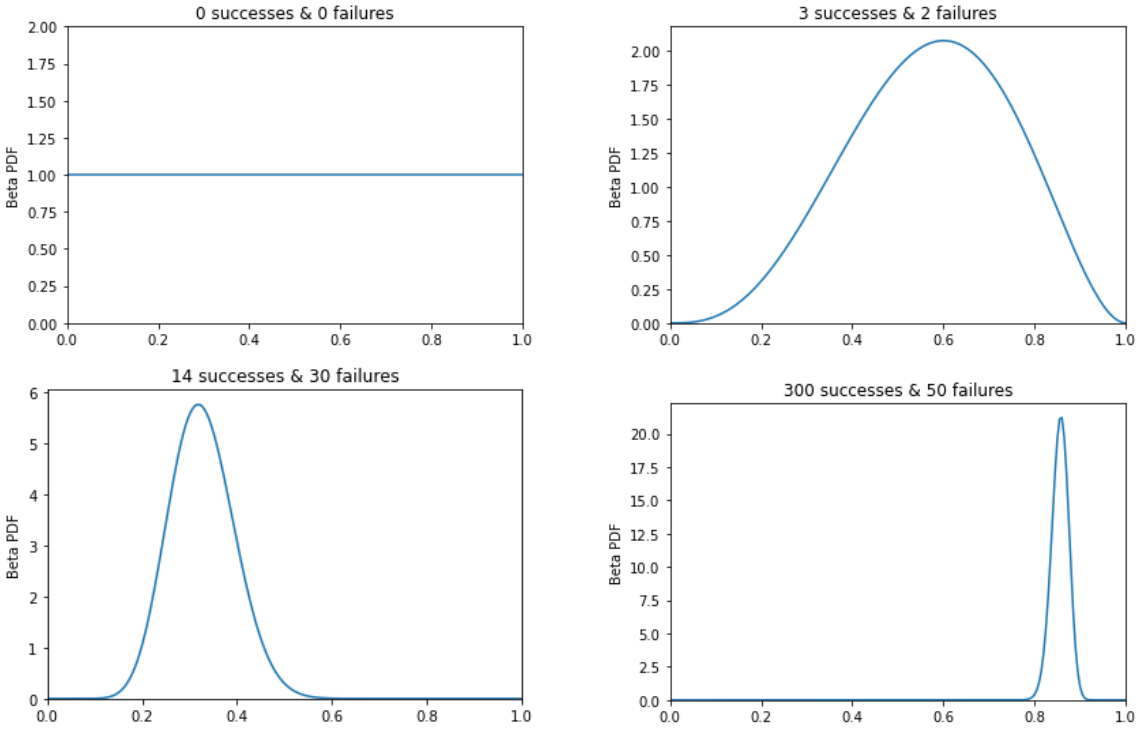


Figure 3.5.1: Various posterior Beta probability density functions

So, now we have described the Bayesian update rule for θ, we use this to decide which action our agent will take. To do this via Thompson sampling we follow these three steps.

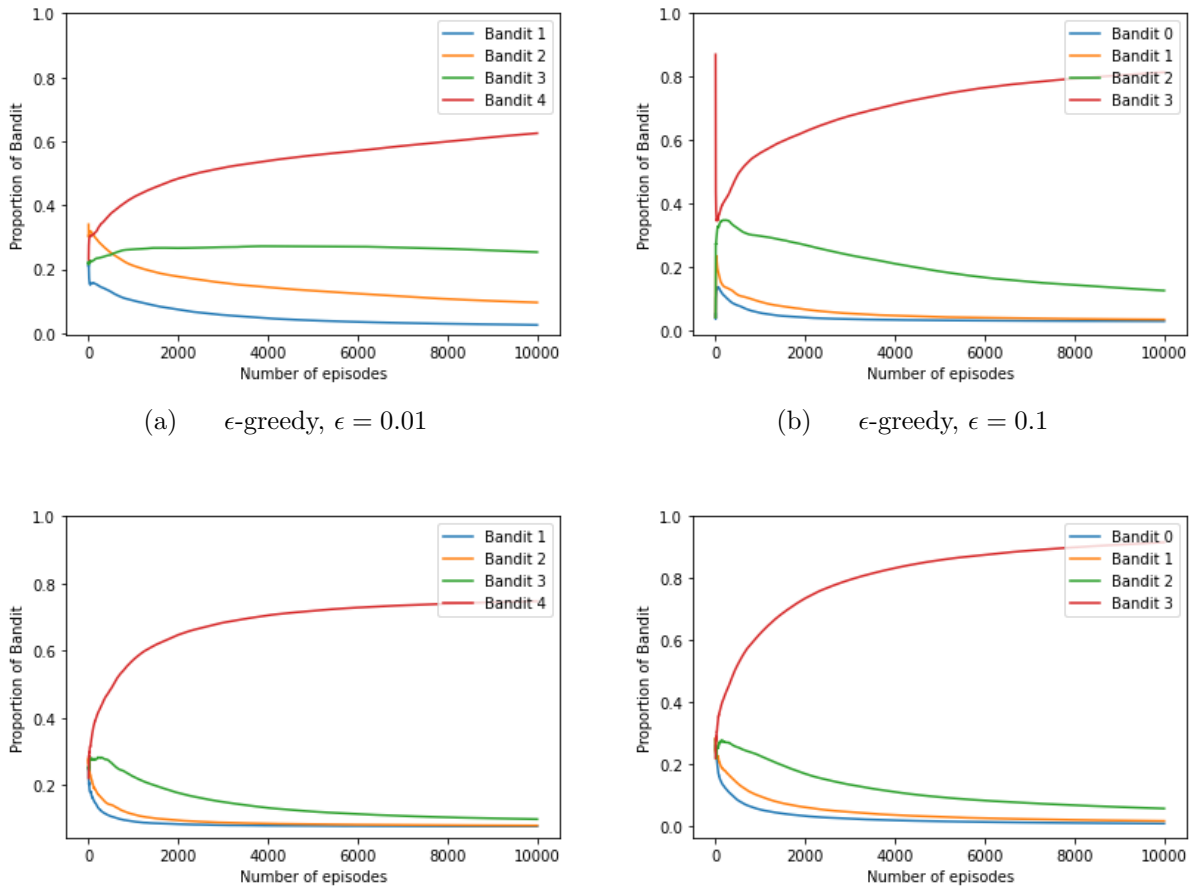
1. Before each spin calculate the posterior distributions $\theta_k \sim Beta(\alpha_k, \beta_k)$ for each bandit, or give uniform distributions for the first episode
2. Sample a θ value from each of the four distributions
3. Play the bandit that gave the largest of the four θ values

We see, unlike ϵ-greedy strategies, this doesn't truly define a policy π which is a probability distribution for the different actions, but describes a method that allows the agent to choose an action.

From the probability distributions in Figure 3.5.1, we see the real beauty of this method and why it is arguably more efficient than an ϵ-greedy strategy. When the agent hasn't played a bandit much, when there is more uncertainty in the rewards it gives, the variance of our θ value is larger and so the pdf is less concentrated around its peak. Therefore, as we sample θ values it has a higher chance of being the largest value and the agent is likely to explore the bandit more. But, as the agent plays bandit k more and more, the pdf becomes more concentrated around its peak since the variances of θ_k decreases, corresponding to the agent having stronger beliefs in the value of θ_k. Distributions with lower means are then much less likely to give the largest sampled value, and thus the corresponding machine is unlikely to be played. In ϵ-greedy

methods, as the agent plays more and more games, the chance of taking a random move always stays the same, whilst Thompson sampling will effectively rule out different machines as the agent learns.

In Figure 3.5.2 we see the proportion of each machine used over 10,000 episodes, averaged over 1000 simulations to see how effective different methods are. The code used to undertake this analysis can be found in the files 'bernoulli_bandit_proportions.py' and 'bernoulli_bandit_regret.py' on my GitHub page [11]. We see, firstly, that $\epsilon = 0.1$ is indeed the most effective value for ϵ. The smaller value of 0.01 causes the agent to too often act greedily with respect to their Q values, choosing the machine with the current largest proportion of successful spins, and doesn't allow enough exploration. With less exploration, the agents Q values, the proportion of successes from each machine, are likely to not be close to the true θ values, and thus the action with the largest Q value will often actually be sub-optimal. The average proportion where the agent uses the optimal fourth bandit is therefore lower in graph (a). Now comparing values 0.1 and 0.3 we see that both of these values allow the agent to learn which bandit is optimal at approximately the same rate, with the beginning of the red curves in (b) and (c) following fairly similar shapes. However, $\epsilon = 0.1$ gives a higher proportion of optimal bandit use in the long run due to the smaller ϵ meaning our agent takes random, often sub optimal, actions with lower probability. Finally, we see graph (d) which shows Thompson sampling to beat all previous ϵ-greedy strategies. As seen in the figure, not only does Thompson Sampling learn to more quickly use the optimal fourth bandit, but in the long run uses it at a much higher proportion compared to all other ϵ-greedy strategies and so overall is more efficient.



(a) ϵ-greedy, $\epsilon = 0.01$

(b) ϵ-greedy, $\epsilon = 0.1$

(c) ϵ-greedy, $\epsilon = 0.3$

(d) Thompson sampling

Figure 3.5.2: Average proportion of each bandit used across 10000 episodes, using different strategies

In sequential decision problems another metric that is often used to compare different policies is *regret*. This is defined for each action and is given by the expectation of reward from an optimal action minus the expectation of reward from the chosen action. As the agent learns to take better actions the regret should decrease, which makes sense. Since our rewards are Bernoulli(θ_k), with expectation θ_k, this is simply given by $0.4 - \theta_{\text{chosen}}$, for θ_{chosen} the true θ value of the chosen slot machine and 0.4 the θ value of the optimal bandit. In Figure 3.5.3 we plot these values over 10,000 actions (episodes), again averaged over 1000 simulations, to reduce random effect, to see how effective the two methods are at learning.

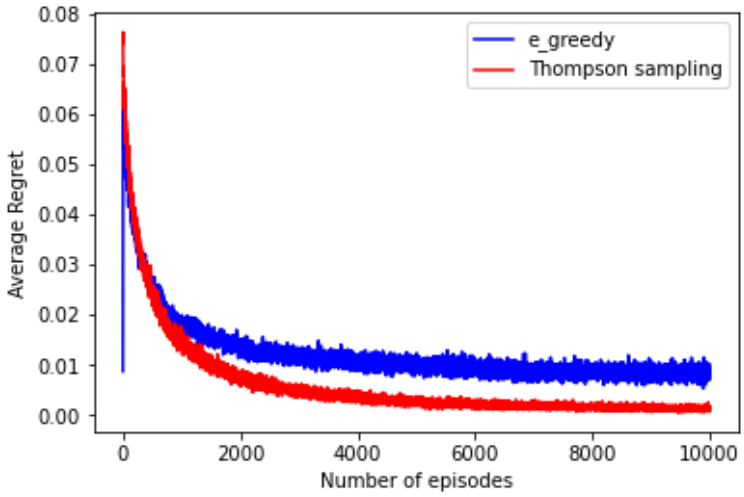


Figure 3.5.3: Comparing the average regret of an ϵ-greedy method ($\epsilon = 0.1$) to Thomson sampling

Figure 3.5.3 shows the regret from Thompson sampling is not only lower than the ϵ-greedy strategy, and in fact tends to 0, but it is also much less volatile. So from these results anyone would ask why we are then going to focus on ϵ-greedy methods. Thompson sampling has shown to be superior in every graph and it doesn't require setting a value for a parameter which can be a tricky task. This is because in the simple setting of Bernoulli Bandits we can exploit these very simply conjugate distributions and their practical mathematical properties to solve the problem, but in the general case using Bayesian methods can be very complex. It can be computationally costly to calculate posterior distributions and sample from them, whilst ϵ-greedy methods retain their simplistic theory and implementation as the problems get much more complex. What's more, ϵ-greedy methods have nice proven mathematical properties and extend to a powerful final method that will conclude this chapter.

One of these nice properties is that given an ϵ-greedy policy π, if we converge onto the true q_π values by playing enough episodes and create a new ϵ-greedy policy π' it is guaranteed that $\pi' \geq \pi$. Meaning the expected return from every state under π' is at least the expected return under π. What's more, it can be proven that if we iterate through this procedure, where we keep playing many episodes, converge onto the action values and then improve our policy, we will converge onto the optimal ϵ-greedy policy. This is all very similar to policy iteration of Chapter 2 and becomes fairly obsolete when considering better methods, so we omit the proofs of these facts, which are all given by Sutton [2, p. 125 - 126].

3.6 GLIE

From our experience with the Multi-armed Bandit we see that ϵ-greedy strategies leave a lot to be desired. Thompson sampling was able to explore machines more when uncertainty of their

rewards was high, but as the agent gained more information they practically ruled out certain actions, acting optimally more often. The definition of ϵ-greedy methods meant the agent would never follow a truly optimal policy, since with probability ϵ they acted randomly. What we really desire is a process that in the limit causes the agent to solely take optimal actions. This is where the GLIE comes in. GLIE stands for *Greedy in the Limit with Infinite Exploration* and is defined by Singh et al. [14, p. 291].

Definition 3.6.1 (GLIE)**.** A policy learning method is GLIE if and only if:

- All state-action pairs are visited an infinite number of times
- In the limit, the policy is greedy with respect to the found Q values (state-action values)

GLIE gives the best of both worlds. The agent will visit each state-action pair infinitely often, sufficiently exploring the whole state and action space, and so they gain a lot of information across the whole MDP. But also, in the limit the agent will approach a truly greedy policy, only taking actions they believe to be optimal. Note that the greedy policy found in the limit isn't generally an optimal policy, but just greedy with respect to the agents Q values. The extra condition of these Q values converging onto the true q_* values is needed for the policy to be optimal.

Our current definition of ϵ-greedy methods satisfy the first condition, due to a constant non-zero chance of taking a random action, but we see this, then, is the reason why the second condition is never satisfied. No matter how long the process goes on for our policy will only be ϵ-greedy with respect to the Q values. To overcome this, as Silver describes [10, Lecture 5, slide 15], we stop using a constant value of ϵ and instead let it decrease over time. We call this decreasing of ϵ an ϵ-schedule, and in the context of Monte Carlo methods we decrease its value after each episode. So after sampling the kth episode, we update the Q values and form a new ϵ_k-greedy policy. This idea makes a lot of intuitive sense, since at the beginning, when there is a larger ϵ value, the agent is much more likely to explore and try to gain information, but as the episodes go on and the agent learns more, the value of ϵ decreases, and the agent is more likely to exploit the information they have gained. Common schedules include

$$\begin{aligned} \epsilon_k &= 1/k, \\ \epsilon_k &= \nu^k \text{ for some } \nu \text{ close to 1}, \\ \epsilon_k &= \max\{1 - \beta k, 0\} \text{ for some } \beta \text{ close to 0}. \end{aligned}$$

So, for example, if $\epsilon_k = 0.99^k$, our first few episodes will have $\epsilon \approx 1$ meaning our agent acts practically uniformly random. By episode 100, $\epsilon \approx 0.37$ and by episode 1000 $\epsilon \approx 0$, meaning the agent acts almost entirely greedily to the Q values.

The actual ν or β values are fine-tuned for each MDP, and Silver calls this process GLIE Monte Carlo Control. It isn't a trivial fact that a certain ϵ-schedule satisfies the first condition of a GLIE method, namely each state-action pair is visited an infinite number of times. Singh et al. [14, appendix B, p. 301 - 303] show this for a certain schedule, using quite advanced probability theory, but we still refer to all decreasing ϵ-schedules as GLIE Monte Carlo Control. So now, to finish this chapter off we shall explicitly write out the GLIE Monte Carlo control algorithm and put its learning capabilities to the test in the game of Blackjack!

Initialise all action values, $Q(s, a)$, as 0, $\epsilon = 1$ and set initial policy as uniform random
repeat
    Play a single episode of the MDP under current policy
    for all time steps in episode **do**
        $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$

6: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}[G_t - Q(S_t, A_t)]$
7: **end for**
8: Decrease ϵ via a chosen schedule
9: Form new ϵ-greedy policy from updated Q values

It should be said that often in practise we update our action values via a slightly different rule, given by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)] \tag{3.6.2}$$

for $\alpha \in (0, 1]$. We call α the learning rate and is a hyperparamater that is finetuned for each problem, but is usually quite small with 0.1 a common choice. This, firstly, means we don't have to keep track of our state-action counters anymore as we use this constant α value. But, more importantly, this gives a larger weighting to more recent returns seen when updating the Q values. It may be tempting to think that with a constant α, equal weightings are given to each of the returns seen to form our Q values. However, equal weightings would mean taking an arithmetic average of the returns, which results in using $\frac{1}{N(S_t, A_t)}$ as described beforehand. As we play episodes and keep visiting state-action pairs, $\frac{1}{N(S_t, A_t)}$ will get much smaller than any α value we choose, meaning less weight is given to the more recent return values seen compared to a constant α. This change in weighting is particularly useful since as we update our Q values we are in turn updating and improving our policy. We, therefore, want our agent to give higher weight to the more recent returns that are due to our improved policy, which is achieved by this constant α.

3.7 Beating the House!

The final section of this chapter uses GLIE Monte Carlo control to train an agent to play the game of blackjack! The aim of blackjack is to get your total card count closer to 21 than the dealer, whilst not going over 21. Face cards are worth 10, ace cards can either be 1 or 11 and all other numbered cards are worth their number. The agent and dealer are dealt two cards each to begin with, both of these are face up for the agent whilst only one is face up for the dealer. The agent can then 'hit', meaning they get a new card and they keep hitting until they either go bust or are satisfied with their score and 'stick'. Once the agent has chosen 'stick', the dealer then reveals their last card and then plays similarly. If both the agent and casino have equal sums it is counted as a draw. As with all casino games, the chances are not in your favour due to the agent having to play first and the dealer observing all our cards, however it is often said that blackjack has some of the best odds in the casino (but in the long term you will lose all your money).

We slightly simplify the game to make it easier to solve via our Monte Carlo methods, matching the blackjack environment 'Blackjack-v1' from the OpenaI gym Python module [15]. Firstly, we treat the deck as being an infinite deck, meaning anyone of your cards can be any value and there is no benefit in keeping track of already played cards. Secondly, we let the dealer play a constant strategy, which is that they will hit until they reach a value of 17 or more and then stick. We use a GLIE Monte Carlo control method where the number of episodes (games) used to train the agent was 10,000,000 and an epsilon schedule of $\epsilon_k = 0.99999^k$ for the kth episode.

Converting this game into an MDP setup is fairly simple. The different states the agent can occupy are governed by

- Agent's total sum
- Dealer's face up card value

- Whether the agent has an ace or not

So for example the 3-tuple (14, 7, False) is one state our agent could be in. Winning, losing or drawing simply sends the agent to a terminal state, meaning our MDP is episodic and so can be tackled via Monte Carlo methods. The actions are 'hit' or 'stick' in each of these states, and rewards are 1, -1 and 0 for winning, losing and drawing, respectively. Due to the simplification of the game, it is technically possible to calculate the dynamics of the MDP i.e the expected rewards given state-action pairs and state-transition probabilities. It is therefore technically possible to use dynamic programming and policy iteration to find an optimal policy. However, these probabilities and expectation are very complex calculations. This is, therefore, an example where the state space is small enough to be able to use dynamic programming but we are hindered by not knowing the dynamics.

We now talk through the first iteration of our algorithm to help our understanding. We initialise all action values as 0 and let $\epsilon = 1$. In this first game the agent receives cards 4 and 9, whilst the dealer receives a Jack and a 7, but only the Jack is visible to us. Our initial state, S_0, is therefore $(13, 10, \text{False})$, since the agent doesn't hold an ace. From this state our agent takes an action that is ϵ-greedy with respect to their current Q values. Since $\epsilon = 1$, our agent takes a random action, which turns out to be 'hit', giving the agent 5, taking them to the state $(18, 10, \text{False})$. Again, taking a uniformly random action causes the agent to now 'stick'. The dealer reveals their second card to be a 7, giving a total of 17 and causing the dealer to 'stick'. The agents 18 beats the dealers 17 and the agent receives reward +1, meaning the return, the future cumulative reward, is +1 for both state-action pairs. The Q values are then updated as

$$\begin{aligned} Q((13, 10, \text{False}), \text{hit}) &= 0 + 1(1 - 0) \\ &= 1, \\ Q((18, 10, \text{False}), \text{stick}) &= 0 + 1(1 - 0) \\ &= 1. \end{aligned} \tag{3.7.1}$$

We decrease ϵ via the schedule, giving $\epsilon = 0.99999$ and now repeat for a further 9,999,999 episodes!

There are some basic ideas that an agent hopefully should learn. These include always hitting if their sum is 11 or less, as there is no chance of going bust in this case, or using the ace as a 1 to carry on hitting. However the exact details of how to play well are not clear so it will be interesting to see the final greedy policy our agent comes up with. Once we have trained our agent we play a further 10,000 episodes to calculate an approximate win rate, and compare this to a uniformly random policy as a baseline, which we will hopefully improve upon! Again, my code can be found on this GitHub page [11], in the file 'blackjack.py'. Inspiration in the use of dictionaries was taken from Phil Tabor [16], and the code used to plot the found blackjack policies can be found here [17].

From Figure (3.7.1) we see our agents learned policy. The bottom axis for the total card sum begins at 11 because, as expected, the agent has learnt that for sums of 11 or below the correct move is to always hit. The graph would just be a large grey patch for these values and so they are omitted. We see that, for example, our agent has learnt that if the dealer is showing 4, they have no usable ace and their current sum is 14 then the best idea is to stick and let the dealer take cards. The agent believes in this case the best chance of winning is to stop taking more cards and hope the dealer ends up going bust. The left graph shows the policy when our agent holds an ace. It is a vastly different shape and shows a single boundary line between a total sum of 17 and 18. This line is the point at which the agent stops using the ace as a value of 1 and starts using at as 11 instead. They believe after 18 there is greater chance of winning

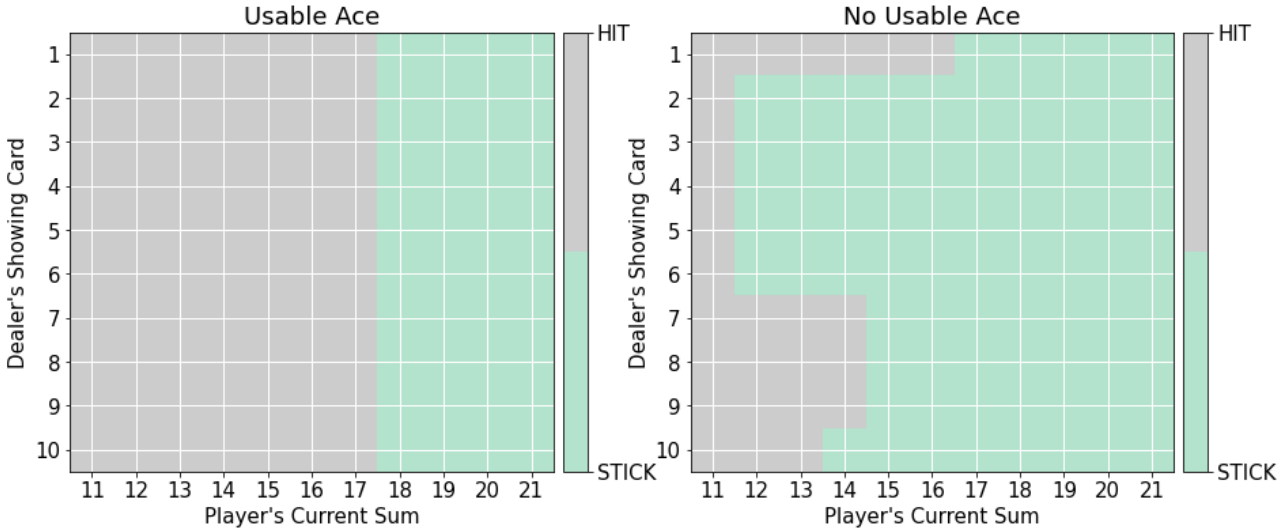


Figure 3.7.1: A trained agent's policy in the simplified game of Blackjack. It tells the agent to either hit or stick depending on their current total, the dealers face up card and whether they have an ace or not

with their current sum then using the lower valued ace to try get a score closer to 21, and that this is true regardless of the dealers card. We see in the table below the win rates over 10,000 games comparing a random policy to this learnt policy. Our agent has significantly improved their win rate and has even got their loss rate to an impressive 0.49, losing less than half the games, something to be proud of!

Policy	Win rate	Loss rate	Draw rate
Uniform random	0.28	0.68	0.04
Learnt policy	0.43	0.49	0.08

3.8 Overview

We have covered a lot in this Chapter. The main idea is that now, when we are in model-free territory, meaning the full MDP dynamics are not known to us, our agent must learn through their own experience. Whereas in dynamic programming, due to our knowledge of the transition and reward dynamics, our agent could learn an optimal policy before ever attempting the task. We then saw that learning through experience is where the main dilemma in reinforcement comes into play, *exploration vs exploitation*. We saw through the small gridworld that acting solely greedily to the agents action values could cause the agent to be stuck in some sub-optimal policy, and thus, we required ϵ-greedy methods that enabled our agent to continue exploring different routes, to find truly optimal actions. Finally, we saw that we could actually decrease the value of ϵ over time, effectively weaning the agent off of this exploration as the agent receives enough information through their experience of the MDP, giving a final powerful algorithm that taught an agent to play Blackjack.

So this concludes our penultimate chapter, but where to go next? Although Monte Carlo methods have proven to be impressive, there are some definite problems. Firstly, and quite importantly, all Monte Carlo methods require an episode to be finished before our agent can do anything. As mentioned before this restricts learning only to the subsection of episodic tasks.

But what's more, our agent is gaining a lot of new information in the middle of each episode, as the environment gives rewards for certain actions. This information isn't in use until after the episode finishes, but surely the agent should use all the information they know up to the point of taking an action, to base their decisions. Secondly, if we recall the chapter in dynamic programming all our ideas there were based on the many different Bellman equations. These equations hold for every single MDP and can be thought of as a defining feature of them. For example,

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s],$$

telling us the expected return from a state is the expectation of the following reward plus the discounted value of the following state. These equations, in general, related various state or action values to succeeding state or action values , giving a variety of recursive like relationships. In the Monte Carlo chapter we haven't taken advantage of these equations whatsoever, but these recursive relationships are something we can, and should, exploit. All of these problems are dealt with in our next chapter!

Chapter 4

Temporal Difference Learning

4.1 TD(0)

So far, we have two approaches in trying to make our agent act optimally, dynamic programming and Monte Carlo methods. Temporal difference (TD) learning incorporates ideas from both of these approaches to give a new method. Again, we first look at the problem of prediction, where we are given a policy π and wish to calculate it's value function, v_π, the expected return from each state. Like Monte Carlo methods, TD learning is model free and so doesn't require the exact reward dynamics and transition probabilities to be known, instead, the agent learns through their own experience. When doing Monte Carlo prediction we initialised state-values as 0 and then after each completed episode following policy π, we used the following update rule for each visited state,

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}[G_t - V(S_t)].$$

We can also use the learning rate $\alpha \in (0, 1]$ instead of $\frac{1}{N(S_t)}$, which we introduced at the end of Section 3.6, giving

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]. \tag{4.1.1}$$

As mentioned before, this update rule can be thought of as having an estimate of the state-value, $V(S_t)$, seeing new data in the form of a new return from this state, G_t, and updating our estimate towards this new value. In TD learning, this new value we use to update our estimates is of a different form, and this is where we see similarities to dynamic programming. Recall the Bellman equation for v_π (1.5.5), which we repeat here,

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s].$$

It tell us that the value of a state is the expectation of the next reward plus the discounted value of the following state, and by iteratively applying this equation we were able to perform *policy evaluation* in Chapter 2. From this equation we see that, given the agent is in state S_t, $R_{t+1} + \gamma v_\pi(S_{t+1})$ is an unbiased estimate of the state value, and so by the law of large numbers if we sampled enough of these values and averaged them, this would converge almost surely onto the true state value. But, of course, we do not have the value function v_π required to sample these values, since this is what we are trying to find. We, therefore, as Sutton describes [2, p. 144], instead use our current estimate of the value of the state, $V(S_t)$, giving

$$R_{t+1} + \gamma V(S_{t+1}), \tag{4.1.2}$$

which we call the TD target. The process, named *TD(0)*, is, therefore, as follows. Initialise all state-values arbitrarily, usually all 0, and let our agent follow policy π. After taking action A_t

in state S_t and receiving reward R_{t+1}, update our estimate of the state-value $V(S_t)$ towards the TD target, via

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)). \tag{4.1.3}$$

Again, we use a learning rate $\alpha \in (0, 1]$. Even though the initial state-values are arbitrary, and subsequent values are based on these initial estimates, the real information our agent receives in the form of the rewards, R_{t+1}, is actually enough for these values to converge onto the true values. Sutton proved [18, theorem 2, p. 24] that for an α small enough, and specific to each MDP, this process will converge onto the true v_π values in the mean, meaning

$$\lim_{n\to\infty} \mathbb{E}[|V^{(n)}(s) - v_\pi(s)|] = 0 \quad \forall s, \tag{4.1.4}$$

where $V^{(n)}(s)$ is the nth estimate of the value of state s. The proof is lengthy, requiring intermediate results from different literature, and so is omitted from this report. Peter Dayan [19, Section 3] was then able to extend this proof, showing that if instead, for each state, a specific sequence of decreasing learning rates is used, then our estimates converge to the true values almost surely. Formally, let $\alpha_n(s)$ be the learning rate used when doing the nth update of $V(s)$. If,

$$\begin{aligned} \sum_{n=1}^{\infty} \alpha_n(s) &= \infty \quad \forall s, \quad \text{and} \\ \sum_{n=1}^{\infty} \alpha_n(s)^2 &< \infty \quad \forall s, \end{aligned} \tag{4.1.5}$$

then we have almost sure convergence. These conditions are called the *Robbins-Monro* conditions, and they often appear in papers to guarantee certain theoretical results. The intuitive reasoning behind them is that they allow the agent to strike a balance between updating their estimates towards the new information seen by a sufficient amount, but also zoning in on specific V values as they gain enough information. The infinite sum means the learning rates are in total large enough, and so our agent updates by a sufficient amount. The finite sum means the learning rates decrease by a sufficient amount overtime, specifically decreasing to 0, causing the agent to zone in on specific V values. However, as Sutton mentions, in practice there is often too much fine tuning required to get good rates of convergence, and so these sequences of learning rates are rarely used. Instead, constant, but small, α values are much preferred.

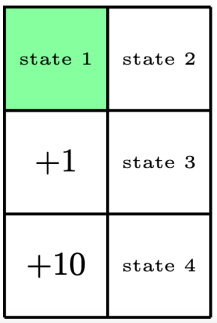
We first discuss the theoretical differences between TD(0) and Monte Carlo methods, before looking at results empirically in an example. We say TD learning bootstraps since we update our estimate of the state value via another estimate, $V(S_{t+1})$. This is alike to *policy evaluation* in Chapter 2, where we updated state-values based on estimates of all possible succeeding state values, as seen in Equation (2.1.1). Monte Carlo doesn't involve bootstrapping since we use the true returns, G_t, to update our state-values, no estimations required. In fact, this is the main idea that underlies all the theory within temporal difference learning, in that we update estimates via other estimates. Another difference is that with Monte Carlo methods you are forced to wait until the episode finishes to update estimates of the state-values, so the actual returns are known. In TD learning as soon as an action is taken and reward received, we can update our values. We have finally broken the shackles that were restricting us to solely episodic tasks, and our agent is now using information they gain as soon as it is given to them.

Another important difference between TD(0) and Monte Carlo is the *bias-variance trade off* which Silver talks of within his lectures [10, Lecture 4, slide 17]. Monte Carlo used the return, $G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T$, which is an unbiased estimate of $v_\pi(S_t)$, by definition. Our agent is following policy π, which means from state S_t there is randomness in the next action they take, then due to the MDP there is randomness in the reward they receive and also the

next state they are transitioned to. This is just over one time step, and the return G_t depends on all proceeding time steps, meaning there is huge amounts of noise and so the variance of the return can be very large. With 0 bias, although Monte Carlo methods have good theoretical convergence, it can take a long time in practise. Meanwhile, TD learning uses the TD target $R_{t+1} + \gamma V(S_{t+1})$. As mentioned before, $R_{t+1} + \gamma v_\pi(S_{t+1})$ is an unbiased estimate of $v_\pi(S_t)$ by the Bellman equation for v_π, meanwhile the TD target is biased. This is due to the arbitrary starting state-values we choose, and the fact that the TD target is based on these estimates. For example, if we initialise them all as 0, the first time we visit state S_t, the TD target would be $R_{t+1} + \gamma \times 0$. This is a biased estimate of $v_\pi(S_t)$, since the state-value is expected return, not the expected initial reward. Thankfully, however, in return for this bias, the TD target has much lower variance, due it's value only depending on one action, transition and reward. This means TD(0) can be more efficient, but is more sensitive to initial values used.

The reader may be wondering why this method is called TD(0). This is due to TD(0) being a specific case of a spectrum of methods, which we call TD(λ). We do not go into the details in this introductory project, but this is the main focus of Chapter 5 in Sutton's textbook [2]. The parameter λ allows the user more control in the bias-variance trade off, and thus allows more fine tuning of our methods to certain problems. In fact, the other extreme value of $\lambda = 1$ can be shown to correspond to Monte Carlo prediction that we saw in Section 3.2. Thus, TD(λ) is a method that bridges the gap between temporal difference learning and Monte Carlo, and with a more general framework can allow more efficient learning. But, do not despair, we can still allow our agent to do some pretty impressive learning with this more simple method!

Example 4.1.6 (Comparing TD(0) to Monte Carlo)**.**



We return to the simple gridworld example. As we are covering the task of prediction, we require a policy π, and we choose a uniformly random policy. The actual state-values can easily be found through a variety of methods. We could invoke policy evaluation from Chapter 2, however, due to the simplicity of this MDP it is easier to just solve the four linear equations given by the Bellman equation for v_π (1.5.6). Solving these gives the following state values:

State	1	2	3	4
State-value	1.8181	2.6363	3.4545	6.7272

We firstly talk through one episode and the subsequent updating of state values via both TD(0) and Monte Carlo methods. Lets say, for example, the sequence of states, actions and rewards of our first episode was:

$$\text{state 1, right, 0, state 2, down, 0, state 3, left, +1.} \tag{4.1.7}$$

Since the state values were initialised as 0, and the TD(0) update rule is

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)),$$

we have that the three updated state values would be

$$\begin{aligned} V(\text{state 1}) &= 0 + \alpha(0 + 1 \times 0 - 0) \\ &= 0, \\ V(\text{state 2}) &= 0 + \alpha(0 + 1 \times 0 - 0) \\ &= 0, \end{aligned}$$

$$\begin{aligned} V(\text{state 3}) &= 0 + \alpha(1 + 1 \times 0 - 0) \\ &= \alpha. \end{aligned}$$

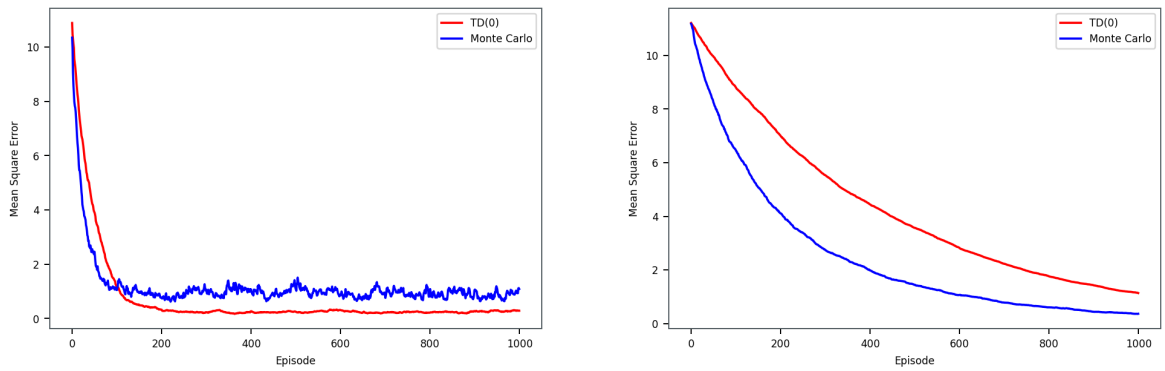
And remember, each update would be done as soon as our agent takes an action and receives a reward. Meanwhile, Monte Carlo prediction uses the update rule of

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)].$$

Since the reward of +1 was received at the end, the return from the three visited states is 1, and thus, all three state values are updated after the episode finishes via

$$\begin{aligned} V(s) &= 0 + \alpha(1 - 0) \\ &= \alpha. \end{aligned} \tag{4.1.8}$$

We now run the two algorithms, initialising state-values as 0 and letting our agent play 1000 episodes. We investigate how the mean square error (MSE) changes with the episode number, where we are taking the MSE between our estimated state-values and the true values. To reduce the effect of the randomness, we do the above simulation 100 times and plot the average of the MSE at the end of each episode. We use two different learning rates to see how much effect this has on our agents learning in the gridworld. The code can be found in the file 'gridworld_MC_vs_TD.py' on my GitHub page [11].



(a) $\alpha = 0.1$ (b) $\alpha = 0.01$

Figure 4.1.1: Average mean square error of our agents estimate of the state-values under a uniform random policy in a gridworld.

There are few things to unpack in Figure 4.1.1. Firstly, in Figure (a) we clearly see the higher variance in Monte Carlo methods, where even with these averaged values there is still a lot of random fluctuation in the graph, whilst the TD(0) line is much smoother. The MSE can be seen to decrease at similar rates for both Monte Carlo and TD(0), showing similar rates of learning, but due to the higher variance of Monte Carlo, TD(0) flattens out at a slightly smaller mean square error. In Figure (b) with the smaller learning rate we see a substantially different graph. The learning rate is now too small, meaning the agent updates the state-values too slowly towards either the real returns seen in Monte Carlo, or the TD targets in TD(0). The MSE, therefore, decreases much more slowly, showing much less efficient learning in our agent as they require around 5 times the number of episodes to reach the same MSE as in Figure (a). This small learning rate is also the reason why we don't see the random fluctuations in the Monte Carlo line, as in Figure (a). Even though there is larger variance in the returns used in Monte Carlo methods, since our agent updates values very slightly towards the returns seen, this balances out the effects of higher variance, causing a smooth line.

4.2 SARSA

Having now introduced the idea of TD(0) to predict a value function, we can very naturally extend it to the control problem, where we want our agent to learn what actions to take. Our approach, which is called SARSA and given by Sutton [2, p. 155], is similar to the Monte Carlo chapter, in that we keep updating our estimates of our Q values, and then use an ϵ-greedy policy based on these values. The problem of *exploration vs exploitation* is still prevalent, and so in order for our agent to keep up exploration to facilitate better learning, our agent acts only ϵ-greedily. Just as in Chapter 3, we need to move away from using state-values and now focus on state-action values, due to our ϵ-greedy policies being based on these Q values. TD(0) updated the state-values via

$$V(S_t) \leftarrow V(S_t) + \alpha(R_t + \gamma V(S_{t+1}) - V(S_t)).$$

Applying the same idea to Q values means using the following update rule,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)). \tag{4.2.1}$$

Since we are now using TD learning methods, this update occurs at each time-step as our agent takes actions, meaning the agents policy is updated at each time step. Implementing the above update rule on the Q values with an ϵ-greedy policy gives the following algorithm.

Initialise all action values as 0 and let $\epsilon = 1$
for each episode **do**
    Initialise first state S and choose action A via an ϵ-greedy policy based on Q values
    repeat
        At state S, take action A, observe reward R and transition to state S'
        At state S', choose action A' via ϵ-greedy policy based on Q values
        Update single Q value via $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'$, $A \leftarrow A'$
        Decrease ϵ via some schedule
    until episode finishes

If the task is continuous, and so non-episodic, this algorithm will just loop forever so it is simply stopped after a chosen number of time steps. We can now see why this process is called SARSA. Given our agent is at state-action pair (S, A), in order to update this action-value, our agent needs to see the reward R gained from this action, observe the next state they are transitioned to, S', and choose the following action A' through an ϵ-greedy policy based on the current Q values. Thus, each update is based on S, A, R, S', A'.

Singh et. al [20, Section 3.1] were able to prove that if the following two conditions are held, then by using SARSA, our agent would learn an optimal policy almost surely in the limit. The conditions are, firstly, to use a decreasing sequences of learning rates that follow the Robbinson-Monro conditions (4.1.5), and secondly to use a decreasing ϵ schedule that causes a GLIE policy learning method (3.6.1). However, as mentioned before, these conditions are often neglected in practice, with, for example, a constant learning rate α commonly preferred to these theoretical optimal limits.

4.3 The Mountain Car Problem

To conclude this penultimate chapter we introduce the mountain car problem, and see if SARSA can solve the task. The idea was first introduced by Andrew Moore in his PhD thesis [21, Section 4.3], and has since become one of the standard tasks in comparing how effective different reinforcement learning algorithms are.

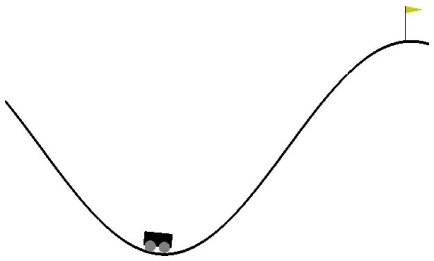


Figure 4.3.1: The mountain car problem

The idea is very simple. A car begins at the bottom of a valley, as in Figure 4.3.1, and can accelerate left or right. The aim is for the car to reach the flag at the far right. However, the car doesn't have enough power to simply drive straight up the hill. The car must, instead, alternate between driving up the left or right hill and accelerating back down until the car gains enough momentum to reach the flag. Again, we use the OpenAI gym Python package which contains a mountain car environment, as seen in this GitHub page [22].

The environment is setup in the following way. The state our agent occupies is determined by two factors, the position and velocity of the car. The position, p, is the position along the x-axis, and we have that $p \in [-1.2, 0.6]$. The velocity, v, is positive if the car is driving to the right, and negative if driving to the left, with range $v \in [-0.07, 0.07]$. The car begins with 0 velocity and starting position p_0 that is uniformly random in $[-0.6, -0.4]$. There are only three available actions to our agent, accelerate left, don't accelerate and accelerate right, and we code these as actions 0, 1 and 2, respectively. For completeness, we state the transition dynamics of the car i.e the equations governing how the cars position and velocity alters at each time step, although the exact details are not too important. Given at time step t our agent occupies state (p_t, v_t) and takes action $a \in \{0, 1, 2\}$, then

$$\begin{aligned} v_{t+1} &= v_t + 0.001(a-1) - 0.0025\cos(3p_t) \quad \text{and} \\ p_{t+1} &= p_t + v_{t+1}. \end{aligned}$$

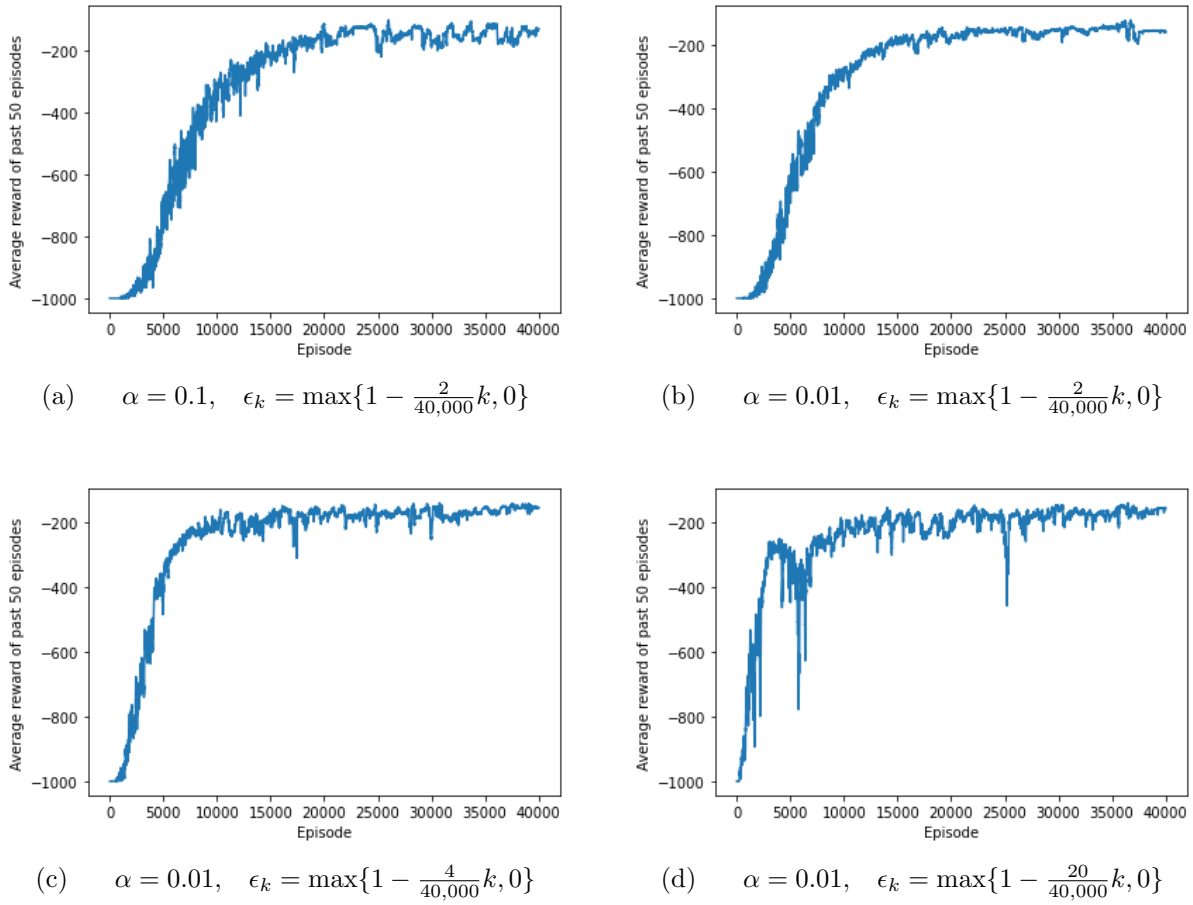
From the description of the mountain car problem it is tempting to think that the agent is taking actions continuously in time, but they are not. In this setup, we still take actions at discrete time steps, and these do slight alterations to position and velocity, transitioning the agent to a new state, via the above equations. The agent then chooses an action at this next state, and this repeats as normal, giving the classic agent environment cycle we introduced all the way back in Section 1.2. It is possible to undertake reinforcement learning in continuous time scenarios, but this outside the scope of this introductory project. Finally, since we want the agent to reach the flag as quickly as possible, we give a reward of -1 for each time step, regardless of the action taken, until either the agent reaches the flag or 1000 time steps occur, ending the episode.

From this setup we instantly see this problem differs from any example seen already, in that the state space is now continuous. There is still a finite number of actions our agent can take, and we still have actions being taken at discrete time steps, but now there are uncountably many states our agent can occupy, due to the uniform random starting position of our agent. This renders our previous methods useless. In dynamic programming we required storing of all the different state values and in Monte Carlo and TD methods we store all the state-action values. Limited by memory, a computer cannot store an infinite number of values, and so we must try to get around this somehow. Luckily, there is a vast amount of theory and research within reinforcement learning to tackle this problem, due to the fact that most problems in real life that might be tackled through reinforcement learning will have either infinite or extremely large state spaces.

Most methods come under the broad category of *function approximation*, where instead of storing all action-values directly in a table, you have a function that approximates these values and it is the function that is updated as the agent learns. Famous types of function approximation include neural networks and decision trees. The general theory of function approximation is outside the scope of this project, instead we shall just use a very simple method to tackle the continuous state space of the mountain car problem, namely, discretising the state space. Discretising the

state space means partitioning the position and velocity intervals into bins, and treating all positions or velocities within each bin as the same, therefore, giving a finite state space. In this specific example we use 20 equal sized bins for both the position and velocities, and so our state space now has $20 \times 20 = 400$ different elements, and there are $400 \times 3 = 1200$ different state-action pairs. Although very simple, there is good reasoning behind this method, in that we expect our Q values to change by a small amount when changing either the position or velocity by a small amount, and thus we can just group the values together, giving one *approximate* Q value.

Now we have setup the problem, we shall deploy the SARSA algorithm, train our agent over 40,000 episodes and hopefully see effective learning. Recalling the SARSA algorithm, we need to set two things. The constant learning rate α, which controls how much our agent updates their estimate of the Q values towards the new data, and the decreasing ϵ schedule which controls how greedily our agent acts towards their current Q values. We decrease our value of ϵ after each episode via the schedule $\epsilon_k = \max\{1-\beta k, 0\}$, so we therefore need to tune the hyper-parameters α and β. To do this, we shall plot the average total reward our agent earns over the past 50 episodes to see how quick and effective our agents learning is, recalling our agent earns a reward of -1 for every time step until they either reach the flag or 1000 time steps occur. The code used can be found on my GitHub page [11] in the file 'mountain_car_training.py', which was influenced by code from Phil Tabor [23], although he used a different algorithm to SARSA.



(a) $\alpha = 0.1, \quad \epsilon_k = \max\{1 - \frac{2}{40{,}000}k, 0\}$

(b) $\alpha = 0.01, \quad \epsilon_k = \max\{1 - \frac{2}{40{,}000}k, 0\}$

(c) $\alpha = 0.01, \quad \epsilon_k = \max\{1 - \frac{4}{40{,}000}k, 0\}$

(d) $\alpha = 0.01, \quad \epsilon_k = \max\{1 - \frac{20}{40{,}000}k, 0\}$

Figure 4.3.2: Graphs showing how effective the agent is at learning in the mountain car problem for different hyper-parameters

All four graphs in Figure 4.3.2 show fairly similar shapes. At the beginning our agent has no experience in the task, taking purely random actions that don't allow the agent to reach the flagpole, thus earning -1000 reward. As our agent plays more episodes, they learn and update

their Q values and begin to exploit actions they believe to be better, causing the average reward to increase. All graphs eventually flatten out at around a reward of -100, meaning our agent has learnt to solve the problem in around 100 time steps, which is known to be fairly optimal. Now, comparing graphs (a) and (b) where the learning rate α differs but the ϵ-schedules are identical, we see that in this case a smaller learning rate is preferred. Updating the Q values more slowly towards the new data gives consistently higher rewards compared to the more inconsistent rewards of graph (a). So now we know that the smaller α value is better, we see whether altering the ϵ schedule can improve our agents learning. Currently we have $\epsilon_k = \max\{1 - \frac{2}{40{,}000}k, 0\}$, meaning by the 20,000th episode our agent will be acting completely greedily towards the agents Q values. Graph (c) tests whether we can use an ϵ schedule that decreases slightly faster over time, meaning our agent more quickly takes actions they believe to be optimal rather than random actions that allow exploration. We see that for (c) we do have improvement over (b) in that the agent more quickly achieves higher rewards and gives approximately the same level of performance in the long run, meaning there are no negatives of using this faster decreasing ϵ schedule. Finally, in graph (d) we take this a step further, using a schedule that causes our agent to act completely greedily by episode 2000. We see that we have now decreased ϵ by too quick a rate, and we don't allow our agent enough exploration before exploiting what they know. This causes the agents average total reward per episode to be much more volatile, meaning the agent reaches the flagpole much more inconsistently. We see that even by around episode 25,000 the agent still sometimes requires around 500 time steps to solve the problem.

The file 'mountain_car_gifs.md' on my GitHub page [11] shows three GIFs displaying the agent traversing the mountain car problem. The first shows a uniformly random policy for the agent, which fails to reach the flag. The second shows the agent always accelerating right and failing to reach the flag. Finally, we see the policy our agent has learnt through SARSA, allowing the agent to traverse the hills and reach the flag!

4.4 Overview

Through our hard work in Chapter 3, where we covered Monte Carlo methods, we could easily extend a lot of the theory and ideas introduced there into this temporal difference learning chapter. Just like Chapter 3, our agent begins with no information about the task, meaning we are in model free territory where the reward or transition dynamics of the MDP are unknown to the agent, and they must learn through their own experience of the MDP. The agent, therefore, has estimates of either state values, V(s), or action values, $Q(s, a)$, and updates these estimates as they experience the task and receive new data. Furthermore, the use of ϵ-greedy methods that are a solution to the problem of *exploration vs exploitation* are still a key component when trying to get our agent to act optimally. They allow our agent to begin by exploring as much as possible, but overtime causes them to solely take actions they believe to be optimal. Really, the only difference from this Chapter is the update rule our agent uses when seeing new data, which linked to dynamic programming and the Bellman equation for v_π. This meant our agent now bootstrapped, since they update their estimates based on other estimates of state or action values, which is a defining feature of temporal difference learning methods. This new update rule allows the agent to update values continuously as they take actions, which we call *online learning*, and this allows our agent to now learn in both continuous and episodic tasks.

What needs to be stressed is that the ideas we have covered here truly only scratch the surface of temporal difference learning. To quote Richard Sutton:

> If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning.

As mentioned before, the ideas we have covered are specific cases of more general methods that

allow the user more control in fine tuning the efficiency of our agents learning. The more general TD(λ) was actually the foundation of the computer program *TD-Gammon*, which was the first program to exhibit world class skill in the game of backgammon. This was all the way back in 1992, and in the 30 years since many advancements have been made. Possibly the most popular reinforcement learning algorithm, Q learning, also comes under temporal difference learning, but due to limitations of space we do not cover these ideas.

Chapter 5

Conclusion

All good things must come to an end, and with that we conclude this introduction into reinforcement learning. There are a few final points to cover regarding reinforcement learning on a whole. Firstly, the concept of function approximation, that we touched on at the end of Chapter 4, should be seen as the key subsequent theory that follows what we have studied here. Throughout this project we have used *tabular* methods in that we have stored every single state or action value separately in large tables. From the 10×10 gridworld of Chapter 2 where each square has its own state value, to the game of Blackjack where each combination of the agent's total sum, the dealer's face up card and whether the agent had an ace or not was stored separately. As touched on before, these tabular methods become obsolete as the state and action spaces grow in complexity, as seen by the continuous state space of the mountain car problem. Not only is it a problem of memory, but the agent will often be occupying a state they have never visited. Instead of using arbitrary Q values for never before experienced states, it is better that our agent can use information learnt from other states that are similar (in one way or another). This is where function approximation comes into play.

Secondly, a comment on the level of mathematics seen within this project. In the second Chapter of this book, Dynamic Programming, we saw quite a lot more mathematical theory and proof underpinning our ideas, which led us to prove policy iteration would always converge onto an optimal policy. In the later chapters, the mathematics behind the ideas gets considerably more advanced with, for example, the proof of the convergence of SARSA to an optimal policy being the subject of a whole research article. We, therefore, instead saw more exciting examples to try compensate this, but all the relevant literature providing the rigorous mathematics underlying various algorithms has been cited where appropriate.

Lastly, as thrilling as it is to study the theory of reinforcement learning, the importance of actually implementing these ideas in your own code cannot be understated. Arguably the whole point in studying reinforcement learning is to be able to apply the theory on a computer, to see an agent learning over time in a task. Throughout this report we have twice seen the use of the OpenAI gym Python package. This package contains many different environments, from the Blackjack environment we saw in Chapter 3 to other exciting tasks such as controlling robots in 3D simulations. The gym package does a lot of the hard work in creating the environments in Python, it is then up to us to implement reinforcement learning algorithms to try and train the agent. The popularity of this package has allowed for great development and comparisons of different algorithms, since they are all being tested on identical environments. It really is the go to toolkit in reinforcement learning and should be taken advantage of. Not only does this solidify your knowledge of the theory, as you have to implement the algorithms yourself, but, believe it or not, it is actually quite fun!

Bibliography

[1] Susanne Erk Hannes Kammerer Manfred Spitzer Birgit Abler, Henrik Walter. Prediction error as a linear function of reward probability is coded in human nucleus accumbens. 2006.

[2] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. 2nd edition, 2017.

[3] AlphaGo documentary. https://www.youtube.com/watch?v=WXuK6gekU1Y.

[4] Pieter Abbeel, Adam Coated, Morgan Quigley, and Andrew Ng. An application of reinforcement learning to aerobatic helicopter flight.

[5] Murray Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu. Deep Blue. 2002.

[6] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1994.

[7] Tom Mitchell 10703 Deep Reinforcement Learning: Policy iteration, Value iteration, Asynchronous DP. https://www.andrew.cmu.edu/course/10-703/slides/lecture4_valuePolicyDP-9-10-2018.pdf.

[8] Haim Sompolinsky. *Lectures on Reinforcement Learning*. 2017.

[9] Albert Nikolaevich Shiryaev and R. P. Boas. *Probability (2nd Ed.)*. Springer-Verlag, Berlin, Heidelberg, 1995.

[10] David Silver 2015 reinforcement learning course UCL. https://www.davidsilver.uk/teaching/.

[11] William Ogier Project III code. https://github.com/Bogier333/Project-III.

[12] Thompson sampling Wikipedia page. https://en.wikipedia.org/wiki/Thompson_sampling.

[13] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A Tutorial on Thompson Sampling. 2017.

[14] Satinder Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. 2000.

[15] OpenAI gym blackjack environment - GitHub page. https://github.com/openai/gym/blob/master/gym/envs/toy_text/blackjack.py.

[16] Phil Tabor Monte Carlo blackjack code. https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/Fundamentals/blackJack-no-es.py.

[17] Blackjack policy plotting code. https://colab.research.google.com/drive/1zVdv5KRmWyoYZGt83QTGxPkY1Gm7WjDM .

[18] Richard S Sutton. Learning to predict by the methods of temporal differences. 1988.

[19] Peter Dayan. The Convergence of TD(λ) for General λ. 1992.

[20] Michael L. Littman Csaba Szepesvári Satinder Singh, Tommi Jaakkola. Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. 2000.

[21] Andrew Moore. Efficient memory-based learning for robot control. 1990.

[22] OpenAI gym mountain car environment GitHub page. https://github.com/openai/gym/blob/master/gym/envs/classic_control/mountain_car.py.

[23] Phil Tabor mountain car code. https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/Fundamentals/mountaincar.py.