

# Reinforcement Learning: An Introduction - Notes

Scott Jeen

April 26, 2021

## **Contents**

# 1 Introduction

## 1.1 Overview

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

## 1.2 Elements of Reinforcement Learning

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

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

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

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

## 2 Multi-arm Bandits

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

### 2.1 An n-Armed Bandit Problem

**The problem:**

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

### 2.2 Action-Value Methods

**The estimated action value**

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

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

**The greedy action selection method:**

$$A_t = \operatorname{argmax}_a Q_t(a) \quad (2)$$

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

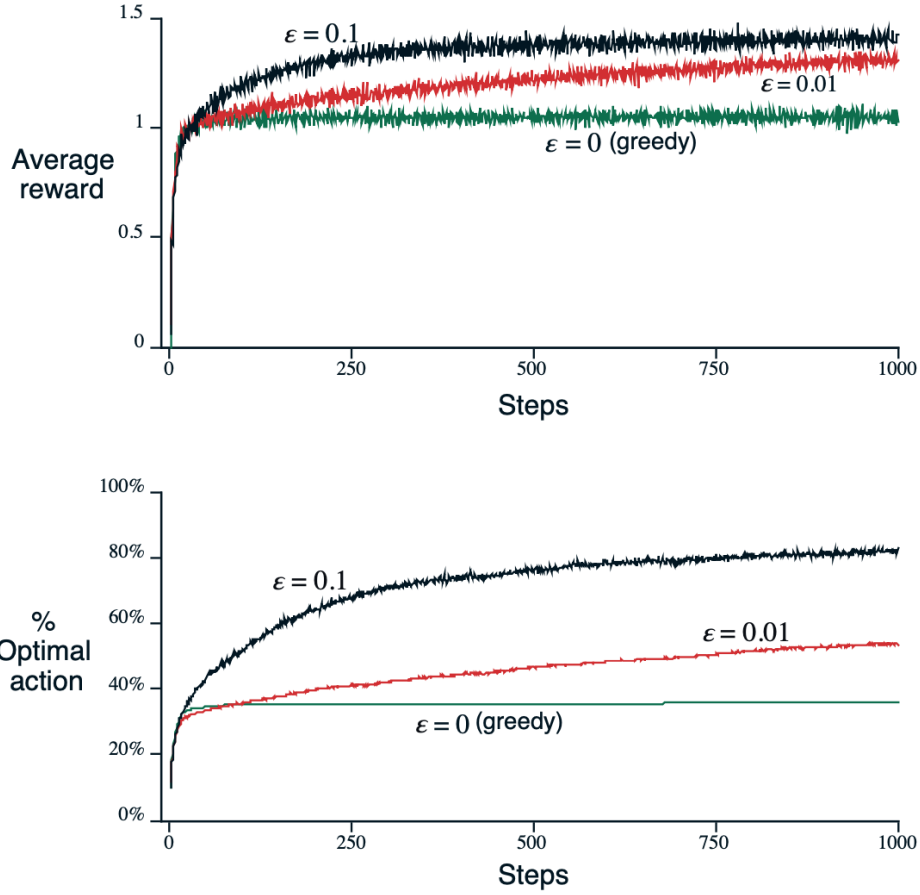


Figure 1: *tbc*

### 2.3 Incremental Implementation

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

$$\begin{aligned}
 Q_{k+1} &= \frac{1}{k} \sum_i^k R_i \\
 &= \frac{1}{k} \left( R_k + \sum_{i=1}^{k-1} R_i \right) \\
 &= \vdots
 \end{aligned} \tag{3}$$

$$= Q_k + \frac{1}{k} [R_k - Q_k] \tag{4}$$

$$\tag{5}$$

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

$$NewEstimate \leftarrow OldEstimate + StepSize [Target - OldEstimate] \quad (6)$$

$\alpha$  is used to denote the stepsize ( $\frac{1}{k}$ ) in the rest of this book.

## 2.4 Tracking a Nonstationary Problem

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

$$Q_{k+1} = Q_k + \alpha [R_k - Q_k] \quad (7)$$

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

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

$$(9)$$

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

## 2.5 Optimistic Initial Values

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

## 2.6 Upper-confidence-bound Action Selection

$\epsilon$ -greedy action selection forces the agent to explore new actions, but it does so indiscriminately. It would be better to select among non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainty in those estimates. One way to do this is to select actions as:

$$A_t = \operatorname{argmax}_a \left[ Q_t(a) + c \sqrt{\frac{\ln t}{N_t(a)}} \right] \quad (10)$$

where  $c > 0$  controls the degree of exploration.

- The square root term is a measure of the uncertainty in our estimate. It is proportional to  $t$  i.e. how many timesteps have passed and inversely proportional to  $N_t(a)$  i.e. how many times that action has been visited. The more time has passed, and the less we have sampled an action, the higher our upper-confidence-bound.
- As the timesteps increases, the denominator dominates the numerator as the  $\ln$  term flattens.
- Each time we select an action our uncertainty decreases because  $N$  is the denominator of this equation.
- UCB will often perform better than  $\epsilon$ -greedy methods

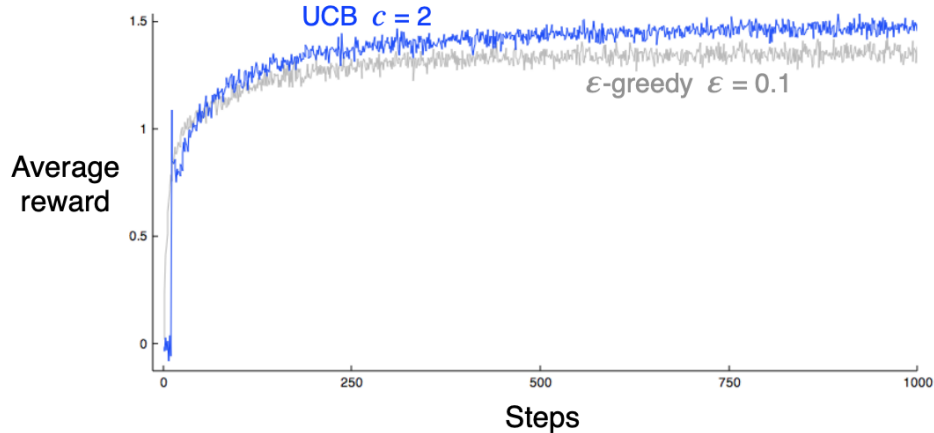


Figure 2: UCB performs better than  $\epsilon$ -greedy in the  $n$ -armed bandit problem

## 2.7 Associative Search (Contextual Bandits)

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

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

## 2.8 Key Takeaways

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

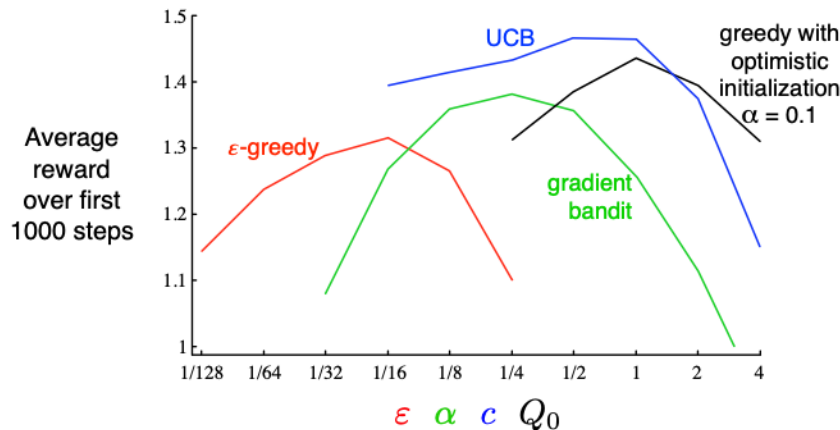


Figure 3: Performance of each of the bandit algorithms explored in this chapter

## 3 Finite Markov Decision Processes

### 3.1 The Agent-Environment Interface

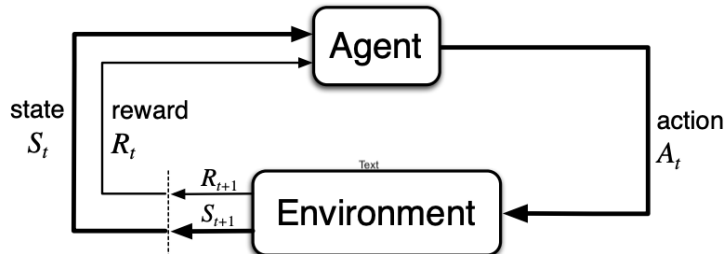


Figure 4: The agent-environment interface in reinforcement learning

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

### 3.2 Goals and Rewards

The *reward hypothesis*:

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

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

### 3.3 Returns and Episodes

The return  $G_t$  is the sum of future rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_t \quad (11)$$

- This approach makes sense in applications that finish, or are periodic. That is, the agent-environment interaction breaks into *episodes*.
- We call these systems *episodic tasks*. e.g playing a board game, trips through a maze etc.
- Notation for state space in an episodic task varies from the conventional case ( $s \in \mathcal{S}$ ) to ( $s \in \mathcal{S}^+$ )



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

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (12)$$

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

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

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

### 3.4 Unified Notation for Episodic and Continuing Tasks

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

### 3.5 The Markov Property

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

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

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

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

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

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

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

### 3.6 Markov Decision Process (MDP)

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

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

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

### 3.7 Policies and Value Functions

- Value functions are functions of states or functions of state-value pairs.
- They estimate how good it is to be in a given state, or how good it is to perform a given action in a given state.
- Given future rewards are dependent on future actions, value functions are defined with respect to particular policies as the value of a state depends on the action an agent takes in said state.
- A *policy* is a mapping from states to probabilities of selecting each possible action.
- RL methods specify how the agent's policy changes as a result of its experience.
- For MDPs, we can define  $\pi(s)$  formally as:

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

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

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

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

- We call  $q_\pi$  the **action-value function for policy  $\pi$**
- Both value functions are estimated from experience.

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

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

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

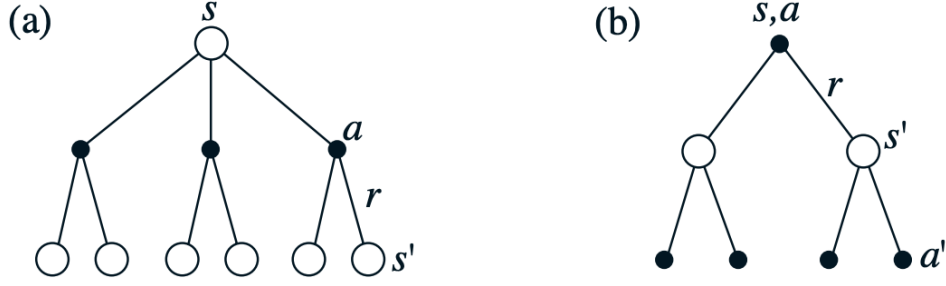


Figure 5: Backup diagrams for  $v_\pi$  and  $q_\pi$

### 3.8 Optimal Policies and Value Functions

- A policy  $\pi'$  is defined as better than policy  $\pi$  if its expected return is higher for all states.
- There is always AT LEAST one policy that is better than or equal to all other policies - this is the *optimal policy*.
- Optimal policies are denoted  $\pi^*$
- Optimal state-value functions are denoted  $v^*$
- Optimal action-value functions are denoted  $q^*$
- We can write  $q^*$  in terms of  $v^*$ :

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

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

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

and secondly for  $q_*$ :

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

- Using  $v^*$  the optimal expected long term return is turned into a quantity that is immediately available for each state. Hence a one-step-ahead search, acting greedily, yield the optimal long-term actions.
- Fully solving the Bellman optimality equations can be hugely expensive, especially if the number of states is huge, as is the case with most interesting problems.
- Solving the Bellman optimality equation is akin to exhaustive search. We play out *every* possible scenario until the terminal state and collect their expected reward. Our policy then defines the action that maximises this expected reward.
- In the continuous case the Bellman optimality equation is unsolvable as the recursion on the next state's value function would never end.

### 3.9 Optimality and Approximation

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

### 3.10 Key Takeaways

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

## 4 Dynamic Programming

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

### 4.1 Policy Evaluation (Prediction)

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

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

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

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

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

### 4.2 Policy Improvement

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

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

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

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

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

### 4.3 Policy Iteration

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

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

This process can be illustrated as:

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

Figure 6: Iterative policy evaluation and improvement

### 4.4 Value Iteration

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

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

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

### 4.5 Asynchronous Dynamic Programming

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

### 4.6 Generalised Policy Iteration

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

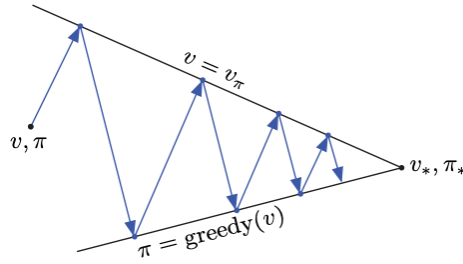


Figure 7: Generalised policy iteration leading to optimality

## 4.7 Key Takeaways

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

## 5 Monte Carlo Methods

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

### 5.1 Monte Carlo Policy Prediction

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



Figure 8: Monte carlo backup diagram for one episode

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



## 5.2 Monte Carlo Estimation of Action Values

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

## 5.3 Monte Carlo Control

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

## 5.4 Monte Carlo Control without Exploring Starts

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

- The idea of on-policy Monte Carlo control is still that of GPI. We use first visit MC methods to estimate the action-value function i.e. to do policy evaluation, but we cannot then make improve our policy merely by acting greedily w.r.t our value function because that would prevent further exploration of non-greedy actions. We must maintain exploration and so improve the  $\epsilon$ -greedy version of our policy. That is to say, when we find the greedy action (the action that maximises our reward for our given value function) we assign it probability  $1 - \epsilon + \frac{\epsilon}{\mathcal{A}(S_t)}$  of being selected so that the policy remains stochastic.
- Note: doing the above will only find us the best policy amongst the  $\epsilon$ -soft policies, which may not be the optimal policy for the environment, but it does allow us to remove the need for exploratory starts.

## 5.5 Off-policy Prediction via Importance Sampling

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

- Off-policy learning methods are powerful and more general than on-policy methods (on-policy methods being a special case of off-policy where target and behaviour policies are the same). They can be used to learn from data generated by a conventional non-learning controller or from a human expert.
- If we consider an example of finding target policy  $\pi$  using episodes collected through a behaviour policy  $b$ , we require that every action in  $\pi$  must also be taken, at least occasionally, by  $b$  i.e.  $\pi(a|s) > 0$  implies  $b(a|s) > 0$ . This is called the assumption of *coverage*.
- Almost all off-policy methods utilize *importance sampling*, a general technique for estimating expected values under one distribution given samples from another. Given a starting state  $S_t$ , the probability of the subsequent state-action trajectory occurring under any policy  $\pi$  is:

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

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

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

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

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

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

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

## 5.6 Incremental Implementation

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

$$NewEstimate \leftarrow OldEstimate + StepSize[Observation - OldEstimate] \quad (32)$$

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

$$V_{n+1} = V_n + \frac{W_n}{C_n} [G_n - V_n] \quad (33)$$

where,

$$C_{n+1} = C_n + W_{n+1} \quad (34)$$

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

## 5.7 Off-policy Monte Carlo Control

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

## 5.8 Key Takeaways

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

```

Off-policy MC control, for estimating  $\pi \approx \pi_*$ 
Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :
   $Q(s, a) \in \mathbb{R}$  (arbitrarily)
   $C(s, a) \leftarrow 0$ 
   $\pi(s) \leftarrow \operatorname{argmax}_a Q(s, a)$  (with ties broken consistently)
Loop forever (for each episode):
   $b \leftarrow$  any soft policy
  Generate an episode using  $b$ :  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ 
   $G \leftarrow 0$ 
   $W \leftarrow 1$ 
  Loop for each step of episode,  $t = T-1, T-2, \dots, 0$ :
     $G \leftarrow \gamma G + R_{t+1}$ 
     $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$ 
     $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$ 
     $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$  (with ties broken consistently)
    If  $A_t \neq \pi(S_t)$  then exit inner Loop (proceed to next episode)
     $W \leftarrow W_{k(A_t|S_t)}$ 

```

Figure 9: Off-policy monte carlo control

- Monte carlo methods use GPI (see chapter 4) in much the same way dynamic programming does. We evaluate our policy, then improve by acting greedily w.r.t our new value function until we converge on the optimal value function and policy.
- Differences with dynamic programming methods: 1) They do not require a model of the environment as they learn directly from experience, and 2) They do not bootstrap i.e. the value function estimates are an average of all real returns accumulated after visiting the state.
- Maintaining sufficient exploration is crucial with monte carlo methods; if our policy is deterministic we will likely not explore the full states space during our roll-outs. To deal with this we have several options: 1) Start every episode randomly with uniform probability such that we make sure we start at each possible state—called *exploring starts*, unrealistic in the real world as we can't make a robot restart from all possible states. Or 2) Use  $\epsilon$ -soft policies that have a non-zero probability of selecting all possible states. The downside of doing this is that we will converge on the optimal  $\epsilon$ -soft, which is not necessarily the optimal policy for the environment, because it needs to learn how account for its own randomness. This is the price we pay for exploration.
- Monte carlo methods can either be *on-policy* or *off-policy*.
- On-policy methods use the same policy to collect data as is evaluated and improved. This suffers the downsides outlined above.
- Off-policy methods have two policies, one that collects the data called the *behaviour policy*  $b$  and the other which we look to improve called the target policy  $\pi$ . We find trajectories from the behaviour policy that line up with our target policy, that is to say, that could have been produced by our target policy. This process only works if the behaviour policy has non-zero probability of selecting each of the actions in the target policy, aka *coverage* of the target policy. The agent explores, but learns a deterministic optimal policy offline that may be unrelated to the behaviour policy used to collect the experience.
- Based on rewards observed by running the behaviour policy, we update our value function using *importance sampling*, which measures, if effect, how likely the observed behaviour would have been given our target policy. For example, the target policy may take 1 of 4 actions with equal probability in each state. If we observe two timesteps from our behaviour policy, then our probability of selecting the actions taken by the behaviour policy would be  $0.25 \times 0.25$ .

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

## 6 Temporal-Difference Learning

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

### 6.1 TD Prediction

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

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

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

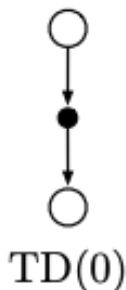


Figure 10: Backup diagram for TD(0)

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

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

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

### 6.2 Advantages of TD Prediction Methods

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

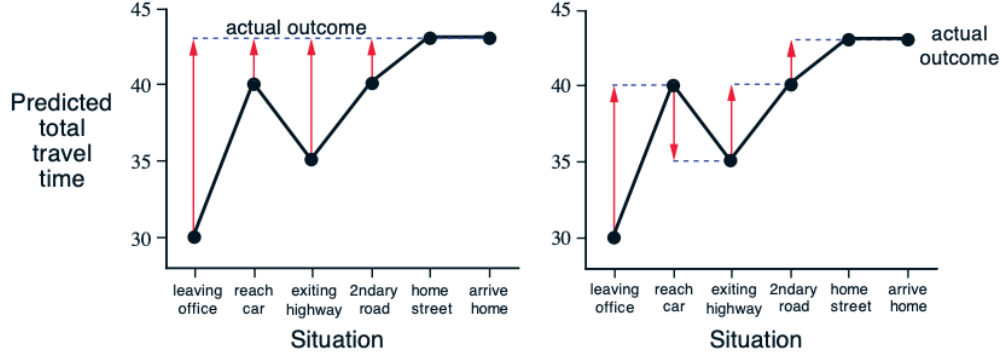


Figure 11: Monte carlo updates (left) and TD-updates (right) to the value function associated with a daily commute

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

### 6.3 Optimality of TD(0)

If we only have a finite number of episodes or training data to learn from we can update our value function in *batches*. We repeatedly play the same data, and update our value function by the sum of each of the increments for each state at the end of the batch.

TD batching and Monte Carlo batching converge often on two different estimates of the value function. The monte carlo batch estimate can only judge based on observed rewards from a state, but TD batching can make estimates based on later states using bootstrapping. Example 6.4 (*You are the predictor*) on pg 127 explains this excellently. The monte carlo estimate will converge on the correct estimate of the value function produced by the training data, but TD methods will generalise better to future rewards because they preserve the transition from state-to-state in the TD update, and thus bootstrap better from values of other states.

In general batch MC methods always minimise the mean squared error on the training set whereas TD(0) finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. This is called the *certainty equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated.

### 6.4 Sarsa: On-policy TD Control

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

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

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

## 6.5 Q-learning: Off-policy TD Control

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

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

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

## 6.6 Expected Sarsa

Instead of updating our value function with the value maximising action at  $S_{t+1}$  (as is the case with Q-learning) or with the action prescribed by our  $\epsilon$ -greedy policy (as is the case with SARSA), we could make updates based on the *expected value* of  $Q$  at  $S_{t+1}$ . This is the premise of expected sarsa. Doing so reduces the variance induced by selecting random actions according to an  $\epsilon$ -greedy policy. Its update is described by:

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

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

$$(41)$$

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

## 6.7 Maximization Bias and Double Learning

Many of the algorithms discussed so far in these notes use a maximisation operator to select actions - either  $\epsilon$ -greedy or greedy action selection. Doing so means we implicitly favour positive numbers. If the true values of state action pairs are all zero i.e.  $Q(S, A) = 0 \forall s, a$ , then, at some stage in learning, we are likely to have distributed value estimates around 0. Our maximisation operator will select the positive value estimates, despite the latent values being 0. Thus we bias positive values, a so-called *maximisation bias*.

We can counter this by learning two estimates of the value function  $Q_1$  and  $Q_2$ . We use one to select actions from a state e.g.  $A^* = \operatorname{argmax}_a Q_1(a)$  and then use the other to provide an estimate of its value  $Q_2(A^*) = Q_2(\operatorname{argmax}_a Q_1(a))$ . This estimate will be unbiased in the sense that  $\mathbb{E}[Q_2(A^*)] = q(A^*)$ .

## 6.8 Games, Afterstates, and Other Special Cases

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



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

## 6.9 Key Takeaways

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

## 7 $n$ -step Bootstrapping

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

### 7.1 $n$ -step TD Prediction

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

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

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

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

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

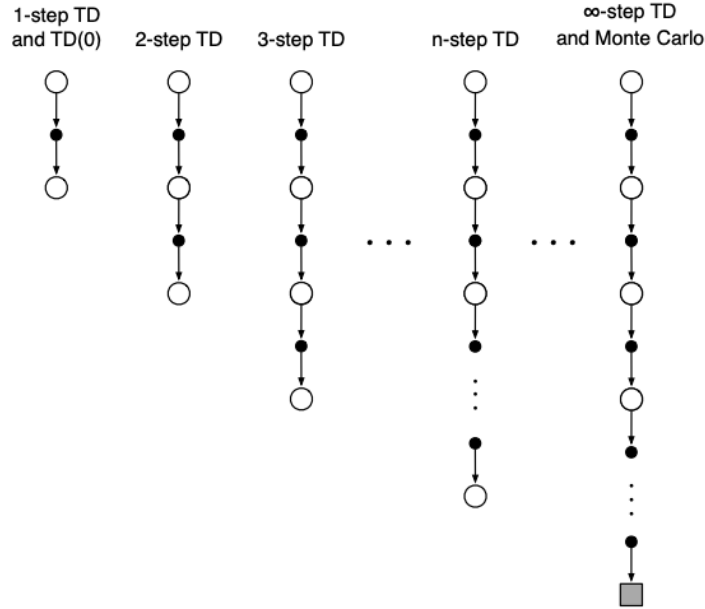


Figure 12: Backup diagram for TD(0)

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

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

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

## 7.2 $n$ -step Sarsa

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

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

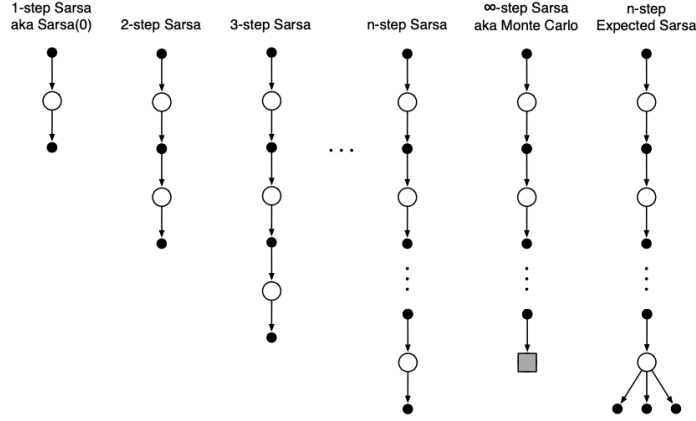


Figure 13: Backup diagrams for family of  $n$ -step Sarsa algorithms

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

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

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

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

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

## 7.3 $n$ -step Off-policy Learning

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

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

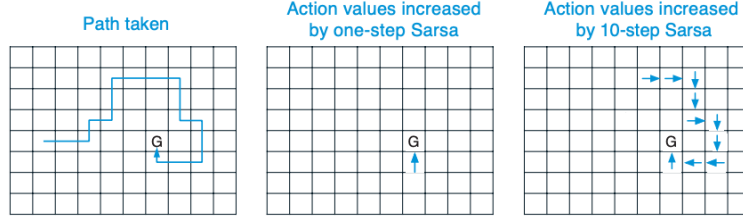


Figure 14: Gridworld example of the speedup of policy learning due to the use of  $n$ -step methods. The first panel shows the path taken by an agent in a single episode, ending at a location of high reward, marked by the  $G$ . In this example the values were all initially 0, and all rewards were zero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

## 7.4 Per-decision Methods with Control Variates

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

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

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

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

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

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

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

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

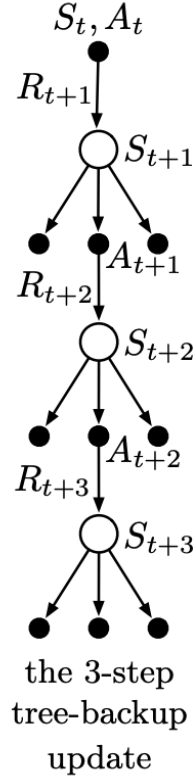


Figure 15: The 3-step tree-backup update

in the general case this becomes:

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

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

## 7.6 A Unifying Algorithm: $n$ -step $Q(\sigma)$

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

## 7.7 Key Takeaways

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

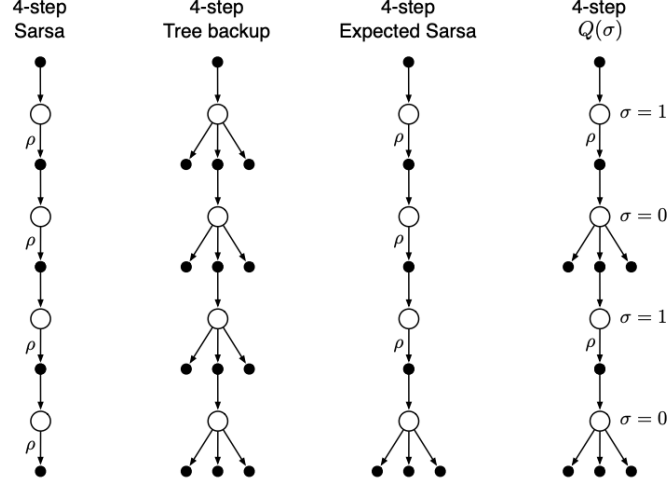


Figure 16: The backup diagrams of the three kinds of  $n$ -step action-value updates considered so far in this chapter (4-step case) plus the backup diagram of a fourth kind of update that unifies them all. The label ‘ $\rho$ ’ indicates half transitions on which importance sampling is required in the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

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

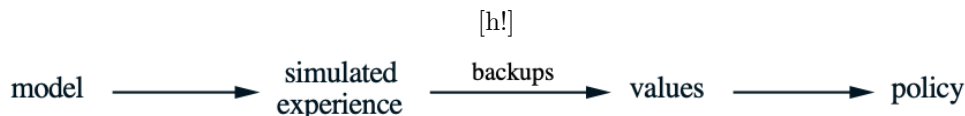


Figure 17: The framework for updating our policy using our model of the environment

## 8 Planning and Learning with Tabular Methods

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

### 8.1 Models and Planning

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

### 8.2 Dyna: Integrated Planning, Acting, and Learning

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

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

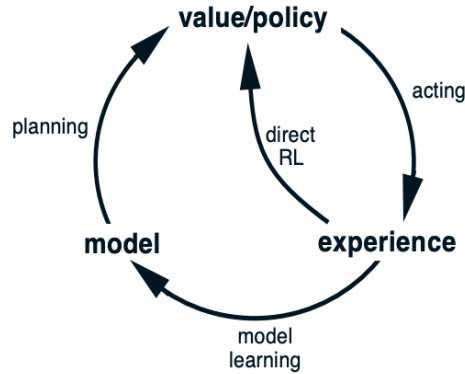


Figure 18: Direct RL and model-learning, either achieved using data collected from experience

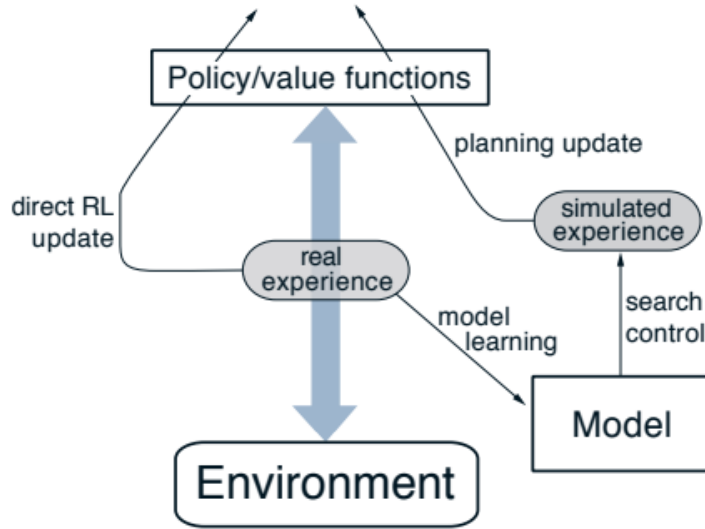


Figure 19: Generalised Dyna architecture

- Planning (in Dyna Q) is the process of running our Q updates (from Q-learning) using our model. We select random states and actions previously taken, find the reward and next state using our model and make the Q update. We do this  $n$  times for each timestep taken in the real environment, where  $n$  is a hyper parameter. For each unit of real experience collected, we are updated our value function  $n$  simultaneously.

### 8.3 When the Model is Wrong

Often our model will be wrong. This can happen initially because the environment is stochastic and we haven't collected enough samples to evaluate this stochasticity accurately. Otherwise, it can happen because we've used function approximation (e.g. neural networks) to predict the model, which generally arrive with non-zero error. When the model is incorrect the planning process is likely to compute a suboptimal policy. Often the suboptimal policy quickly leads to the discovery and correction of modelling error, as



when the agent takes actions prescribed by the planned policy in the real environment, it quickly finds the expected rewards are incorrect or do not exist.

Issues arise when the environment changes to become *better* or more favourable at some stage during learning. In these cases the optimal policy/route to goal remains unchanged, but there is an even better policy available that the agent may never access because it has no reason to doubt its previously learned optimal policy. To address this, another algorithm is proposed called Dyna-Q+. Here, the agent keeps track of the number of timesteps elapsed  $\tau$  since a state-action pair had been selected, and, if sufficient time has elapsed, it is presumed that the dynamics of the environment from that state have changed. The modelled rewards for each state-action pair now take the form  $r + k\sqrt{\tau}$  for some small  $k$ .

## 8.4 Prioritized Sweeping

Until now, updates made to the value function during planning have been made arbitrarily, that is to say, state-actions have been sampled uniformly from past experience regardless of how likely we are to visit those state-actions. It would be more useful to prioritize updates to the value function that would be most effected by the reward we have recently received. This general idea is termed *backward focusing* of planning computations. If we prioritize the updates according to a measure of urgency and perform them in order of priority, we call this *prioritized sweeping*. The prioritization queue is assembled based on the expected update to a state-value pair from a new reward. We then loop through this queue, and for each state-value pair, find the state-value pairs we would predict lead to it, and calculated *their* expected update and add them to the queue. The additions to the queue are only made if the expected update is larger than some value  $\theta$ . In this way, we only plan based on state-value pairs whose value updates are non-trivial. This technique is regularly 5-10 times faster than Dyna-Q.

## 8.5 Expected vs. Sample Updates

This book began by discussing dynamic programming; a way of conducting policy evaluation and improvement given a distribution model of the environment. Then we discussed sampling methods like: monte carlo, temporal-difference methods and n-step bootstrapping to estimate value functions in the absence of a model. Implicit here, was the idea that a distribution model than can be used to compute expectations is better than sampling which often leads to higher variance through sampling error. But is this true? To properly assess the relative merits of expected and sample updates for planning we must control for their different computational requirements.

In practice, the computation required by update operations is usually dominated by the number of state-action pairs at which  $Q$  is evaluated. For a particular starting pair,  $s, a$ , let  $b$  be the *branching factor* (i.e. the number of possible next states,  $s'$  for which  $\hat{p}(s'|s, a) > 0$ ). Then an expected update of this pair requires roughly  $b$  times as much computation as a sample update. If we have sufficient time to complete an expected update, it is usually better than that of  $b$  sample updates because of the absence of sampling error. But if there is insufficient time, sampling is always better because some update is better than no update. The key question is therefore: given a unit of computational power, is it better to perform one expected update, or  $b$  sample updates? An answer to this question is provide in Figure ?? . For large values of  $b$  sampling has huge efficiency gains over expected updates. In reality this effect would be more pronounced as the values of the successor states would be estimates that are themselves updated. By causing estimates to be more accurate sooner, sample updates will have a second advantage in that the values backed up from the successor states will be more accurate.

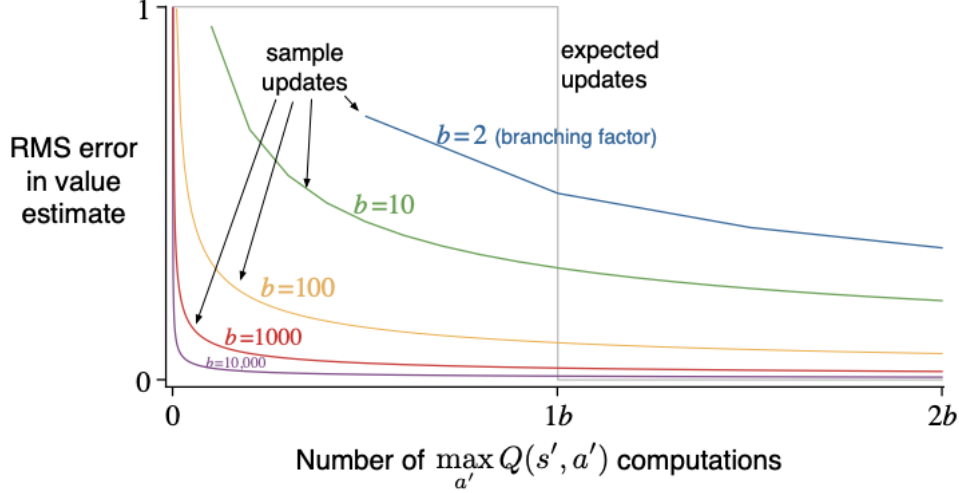


Figure 20: Comparison of efficiency of expected and sample updates

## 8.6 Trajectory Sampling

One could update every state-action pair, or state value, by performing full sweeps of the state space using dynamic programming. As discussed on several occasions this is computationally inefficient, given many of the states may have small probability of occurring. Instead we could sample from state-action space according to some distribution; uniformly in the case of Dyna Q, or according to an on-policy distribution that we observed following the current policy. We can do so by interacting with the model, simulating explicit individual trajectories and updating state or state-action pairs encountered along the way. We call this *trajectory sampling*.

A question arises around whether on-policy updates, that follow the model's transition distribution, are better than updates made with uniform probability across the state-space. It seems for problems with large state-spaces, and small branching factors, it is advantageous to use on-policy distribution sampling. In the long run, however, focusing on the on-policy distribution may hurt because our estimates for these values are already well defined and we are not seeing unusual parts of the state space.

## 8.7 Real-time Dynamic Programming

Real-time dynamic programming (RTDP) is an on-policy trajectory-sampling version of the value iteration algorithm of dynamic programming (DP). Because the algorithm follows the on-policy distribution, our value iteration updates are more efficient than normal DP because we implicitly ignore irrelevant states to our policy in our updates. RTDP is guaranteed to find a policy that is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all, under certain conditions. These are:

- The initial value of every goal state is zero
- There exists at least one policy that guarantees that a goal state will be reached with probability one from any start state.
- All rewards for transitions from non-goal states are strictly negative, and

- All the initial values are equal to, or greater than, their optimal values (which can be satisfied by setting initial values to 0.)

Tasks with these properties are usually called *stochastic optimal path problems*. RTDP can find optimal policies for these tasks with approximately 50% of the computation required for traditional sweep-based value iteration i.e. dynamic programming.

## 8.8 Planning at Decision Time

The techniques discussed thus far have involved planning in between action-selection, we call this *background planning*. Another way to do planning is at the time of action-selection i.e. to run trajectories or simulations in our model, and decide the optimal action to select based on the outcome of these trajectories. We call this *decision-time planning*.

So we now have two ways of thinking about planning:

1. Using simulated experience to gradually improve our policy or value function, or
2. Using simulated experience to select an action for the current state

Usually the results of this kind of planning are not used to update our value function or policy directly, and are generally discarded after action selection. This is fine for large state spaces, but if the state spaces are smaller than it can be expedient to keep these values and update our value function, such that we combine the two types of planning outlined above.

Decision-time planning is most useful in applications where fast response is not required, like chess. If low-latency actions are required, it's generally better to do planning in the background.

## 8.9 Heuristic Search

The canonical decision time planning method is *heuristic search*. For each state encountered, heuristic search dedicates computational resource to searching a large tree of possible continuations, or state permutations thereafter. The search evaluates leaf nodes at the end of the search and backs up the values to the state-action nodes for the current states. The value maximising action from the current state is found and then selected. The values are usually discarded. This kind of planning is effective because it focuses only on pertinent next states and actions, and focuses resource on obtaining the next best one-step action. The backup diagram for heuristic search is shown in Figure ??.

## 8.10 Rollout Algorithms

Rollout algorithms apply Monte Carlo control to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns from many simulations in a model using the current policy. These algorithms, like heuristic search, make immediate use of these action value estimates then discard them, they do not attempt to estimate  $q_*$  like monte carlo policy iteration as discussed in chapter 5. They take advantage of the mathematical properties of policy iteration discussed in chapter 4, that is,  $\pi' \neq \pi$  if both are identical except that  $\pi'(s) = a \neq \pi(s)$  and  $q_{\pi}(s, a) \geq v_{\pi}(s)$ .

The limited factor with these types of algorithms is always computational resource, which itself is dependent on the size of the state-space. However there are ways around this, including pruning the rollouts to only involve the most probable next states, or to truncate simulated trajectories so they do not run to the end of the episode.

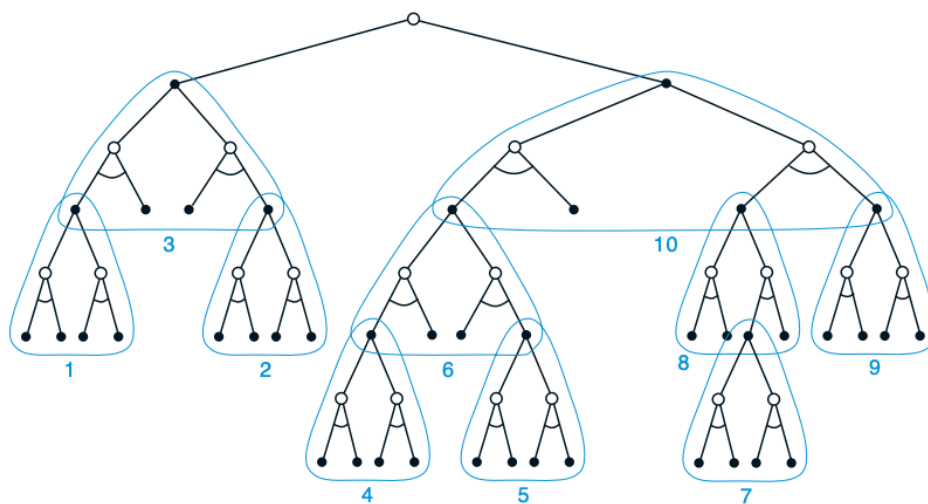


Figure 21: Heuristic search can be implemented as a sequence of one-step updates (shown here outlined in blue) backing up values from the leaf nodes toward the root. The ordering shown here is for a selective depth-first search.

## 8.11 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a rollout algorithm as described above, but with some modifications. Instead of rolling out trajectories randomly, MCTS first selects a small set of actions according to its *tree policy*, then sometimes expands this selection to one or more actions, and then performs a full rollout to end of episode using a rollout policy. The value found upon termination of this trajectory is then backed up to the current state. This process is repeated for as long as time/computational constraints allow, then the agent selects an action usually based on which action produced best future estimated reward. MCTS is illustrated in Figure ??

## 8.12 Key Takeaways

- Planning algorithms use models. These models can be either *distribution models* that provide a probability distribution over next states (as required to compute expectations for dynamic programming), or can be *sample models* that provide a sample of future states drawn from the underlying distribution that we are yet to access, or cannot access.
- Algorithms used for both learning and planning are often identical, the difference being that *learning* involves updating our value function from real experience, whereas *planning* involves updating our value function based on simulated experience from our model.
- Models can often be wrong, and the best way to ensure they stay up to date is to continually maintain them through true experience. Sometimes we will obtain an optimal policy using a model that was once correct, but has since become redundant. We can attempt to keep an accurate model of our environment by taking actions in the environment that have not been attempted for some time to test the truth of our model. This technique is exhibited by Dyna Q+.
- Sample updates are often more efficient than expected updates for large state-spaces

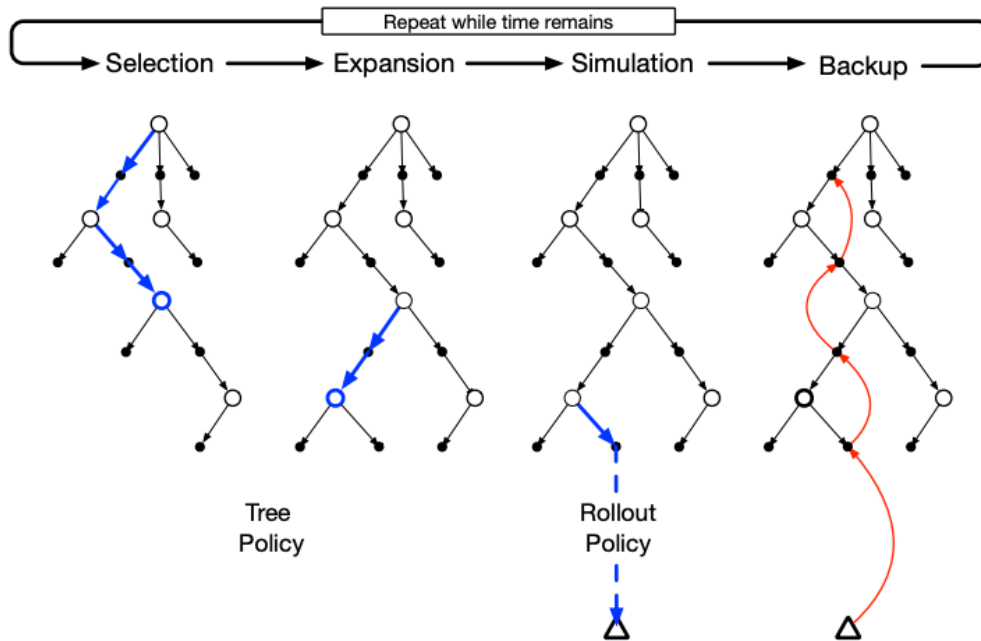


Figure 22: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations *Selection*, *Expansion* (though possibly skipped on some iterations), *Simulation*, and *Backup*, as explained in the text and illustrated by the bold arrows in the trees.

with large branching factors (a measure of the expected number of possible next states from an action).

- Trajectory sampling in our model can be used for planning. This can be useful when conducted on-policy, because we prioritise making updates to our value function that are particularly relevant to our policy.
- Planning can be conducted pre-action selection, to update our value function and inform future action-selection, or it can be done at the time of action-selection. The former is called *background planning* and the latter is called *decision time planning*.
- Examples of decision time planning include Heuristic Search (where value maximising actions are taken in a simulation of various next states, and backed up to the root node), Rollout Algorithms (where monte carlo rollouts are performed over many simulations of episodes and the value maximising action returned), or MCTS (a special case of rollout algorithms that effectively combines heuristic search over some small number of actions and then performs monte carlo rollouts thereafter).
- Decision time planning can be effective for tasks where low-latency action selection is not required. They are fundamentally limited by the time constraints on action-selection and on the available computational resource.

### 8.13 Summary of Part 1

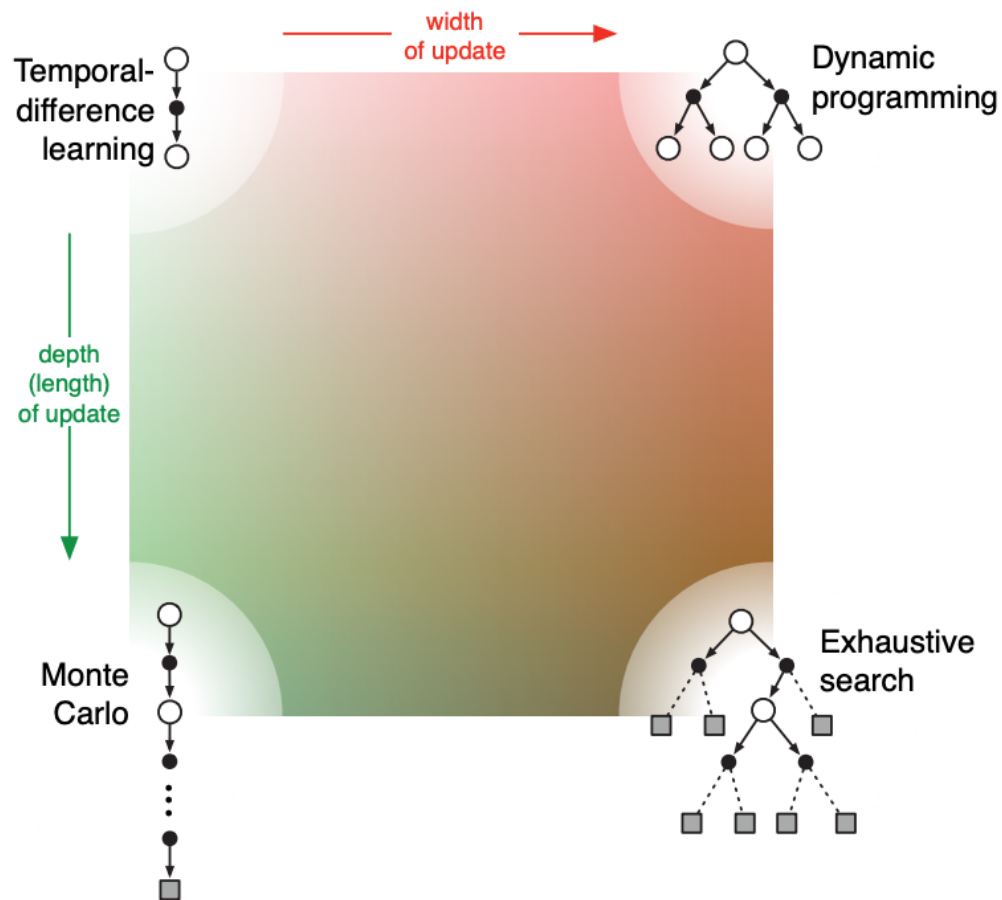


Figure 23: A slice through the space of reinforcement learning methods, highlighting the two of the most important dimensions explored in Part I of this book: the depth and width of the updates.

## 9 On-policy Prediction with Approximation

### 9.1 Value-function Approximation

Until now we have made updates to our value function in the tabular setting, we can, however, store our value function using some kind of function approximating technique with parameters  $\mathbf{w} \in \mathbb{R}^d$ . Most function approximation techniques are examples of *supervised learning* that require parameters  $\mathbf{w}$  to be found using training examples. In our case, we pass the function approximators our update to the value function online i.e.  $s \mapsto u$  where  $u$  is the update of the value function we'd like to make at state  $s$ .

### 9.2 The Prediction Objective ( $\bar{V}E$ )

Given we have far less parameters  $\mathbf{w}$  than states  $s$  we cannot feasibly approximate the value function perfectly. Indeed any updates we make the weights based on one state update will invariably lead to increased error at other states. We must therefore define which states we care about getting correct most, which we do with a state distribution  $\mu(s) \geq 0, \sum_s \mu(s) = 1$ . We can weight the error in prediction between our approximate value function  $\hat{v}(s, \mathbf{w})$  and the true value function  $v_\pi(s)$  by this distribution to obtain our objective function, the *mean square value error*:

$$\bar{V}E(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) [v_\pi(s) - \hat{v}(s, \mathbf{w})]^2 \quad (53)$$

Often  $\mu(s)$  is chosen to be the fraction of time spent in  $s$ , called the *on-policy distribution* in on-policy training.

### 9.3 Stochastic-gradient and Semi-gradient Methods

Our function approximator can be parametrised by a weight vector with a fixed number of real valued components,  $\mathbf{w} \doteq (w_1, w_2, \dots, w_d)^T$ . In *stochastic gradient descent*, we update the weight vector at each timestep by moving it in the direction that minimises the error most quickly for the example shown:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2} \alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \quad (54)$$

$$= \mathbf{w}_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t) \quad (55)$$

where the second equation is obtained via the chain rule. Here,  $\nabla f(\mathbf{w})$  for any scalar expression  $f(\mathbf{w})$  that is a function of a vector (here  $\mathbf{w}$ ), denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

$$\nabla f(\mathbf{w}) \doteq \left( \frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \dots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^T. \quad (56)$$

Often we will not know the *true* value function that we are trying to approximate, and instead we will be using a bootstrapped value or some other approximation. In this case we replace  $v_\pi(s)$  with  $U_t$  which yields the following generalised SGD method for state-value prediction:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [U_t - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t) \quad (57)$$

If  $U_t$  is an unbiased estimate, that is, if  $\mathbb{E}[U_t | S_t = s] = v_\pi(s)$  then  $\mathbf{w}_t$  is guaranteed to converge to the local optimum. An example of an unbiased estimator is the monte carlo estimate for state  $s_t$ , unfortunately methods that use bootstrapping are biased in that their target is not independent of the the weights  $\mathbf{w}_t$ . We therefore call these methods *semi-gradient methods*.



## 9.4 Linear Methods

Corresponding to every state  $s$ , there is a real-valued vector  $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \dots, x_d(s))^T$  with the same number of components as  $\mathbf{w}$ . Linear methods approximate the state-value function by the inner product between  $\mathbf{w}$  and  $\mathbf{x}(s)$ :

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s) \doteq \sum_{i=1}^d w_i x_i(s) \quad (58)$$

$\mathbf{x}(s)$  is called a *feature vector* representing state  $s$ . For linear methods, features are *basis functions* because they form a linear basis for the set of approximate functions. For linear methods, the gradient of the approximate value function w.r.t  $\mathbf{w}$  in this case is:

$$\hat{v}(s, \mathbf{w}) = \mathbf{x}(s) \quad (59)$$

Thus the general SGD update reduces to a simple form:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [U_t - \hat{v}(S_t, \mathbf{w})] \mathbf{x}(S_t) \quad (60)$$

For the continuing case, the update at each time  $t$  is:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha (R_{t+1} + \gamma \mathbf{w}_t^T \mathbf{x}_{t+1} - \mathbf{w}_t^T \mathbf{x}_t) \mathbf{x}_t \quad (61)$$

$$= \mathbf{w}_t + \alpha (R_{t+1} \mathbf{x}_t + \mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1}^T) \mathbf{w}_t) \quad (62)$$

$$(63)$$

Once the system has reached steady state, for any given  $\mathbf{w}_t$ , the expected next weight vector can be written as

$$\mathbb{E}[\mathbf{w}_{t+1} | \mathbf{w}_t] = \mathbf{w}_t + \alpha (\mathbf{b} - \mathbf{A} \mathbf{w}_t) \quad (64)$$

where

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1} \mathbf{x}_t] \in \mathbb{R}^d \text{ and } \mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T] \in \mathbb{R}^{d \times d} \quad (65)$$

It's therefore clear that the system must converge toward the weight vector  $\mathbf{w}_{TD}$  at which:

$$\mathbf{b} - \mathbf{A} \mathbf{w}_{TD} = 0 \quad (66)$$

$$\mathbf{b} = \mathbf{A} \mathbf{w}_{TD} \quad (67)$$

$$\mathbf{w}_{TD} \doteq \mathbf{A}^{-1} \mathbf{b} \quad (68)$$

This quantity is called the *TD fixed point*. The  $n$ -step semi-gradient TD algorithm for estimating  $\hat{v} \approx v_\pi$  is given in Figure ??.

## 9.5 Feature Construction for Linear Models

Constructing features that are appropriate for the RL task is critical for providing the agent useful prior knowledge about the domain. In this section we will discuss different ways of constructing features.

### 9.5.1 Polynomials

We may want to design features with higher complexity than can be captured with linear methods. If we think of the example of parametrising a state  $s$  by two dimensions  $s_1 \in \mathbb{R}$  and  $s_2 \in \mathbb{R}$ , we could represent this state as  $\mathbf{x}(s) = (s_1, s_2)^T$ . This, however, does not allow us to represent interactions between the two dimensions if they do indeed interact, and it would also mean that the value of our state would have to be 0 if

### **$n$ -step semi-gradient TD for estimating $\hat{v} \approx v_\pi$**

Input: the policy  $\pi$  to be evaluated  
Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  such that  $\hat{v}(\text{terminal}, \cdot) = 0$   
Algorithm parameters: step size  $\alpha > 0$ , a positive integer  $n$   
Initialize value-function weights  $\mathbf{w}$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )  
All store and access operations ( $S_t$  and  $R_t$ ) can take their index mod  $n + 1$

Loop for each episode:  
  Initialize and store  $S_0 \neq \text{terminal}$   
   $T \leftarrow \infty$   
  Loop for  $t = 0, 1, 2, \dots$  :  
    If  $t < T$ , then:  
      Take an action according to  $\pi(\cdot | S_t)$   
      Observe and store the next reward as  $R_{t+1}$  and the next state as  $S_{t+1}$   
      If  $S_{t+1}$  is terminal, then  $T \leftarrow t + 1$   
       $\tau \leftarrow t - n + 1$  ( $\tau$  is the time whose state's estimate is being updated)  
      If  $\tau \geq 0$ :  
         $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$   
        If  $\tau + n < T$ , then:  $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$  ( $G_{\tau:\tau+n}$ )  
         $\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{v}(S_\tau, \mathbf{w})] \nabla \hat{v}(S_\tau, \mathbf{w})$   
      Until  $\tau = T - 1$

Figure 24: Algorithm:  $n$ -step semi-gradient TD for estimating  $\hat{v} \approx v_\pi$

Suppose each state  $s$  corresponds to  $k$  numbers,  $s_1, s_2, \dots, s_k$ , with each  $s_i \in \mathbb{R}$ . For this  $k$ -dimensional state space, each order- $n$  polynomial-basis feature  $x_i$  can be written as

$$x_i(s) = \prod_{j=1}^k s_j^{c_{i,j}}, \quad (9.17)$$

where each  $c_{i,j}$  is an integer in the set  $\{0, 1, \dots, n\}$  for an integer  $n \geq 0$ . These features make up the order- $n$  polynomial basis for dimension  $k$ , which contains  $(n+1)^k$  different features.

Figure 25: Generalised form of polynomial basis

both dimensions were zero. We can instead use a four-dimensional feature vector like:  $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2)^T$ . This allows us to capture any combination of these dimensions in polynomial space. The generalised form of polynomials is shown in Figure ??.

Note that  $c_{i,j}$  here is a matrix with  $i$  rows corresponding to  $i$  features and  $j$  columns corresponding to  $j$  dimensions.

#### **9.5.2 Fourier Basis**

Fourier series express periodic functions as weighted sums of sine and cosine basis functions (as features) of different frequencies, with a function  $f$  being periodic if  $f(x) = f(x + \tau)$  for all  $x$  and some period  $\tau$ . The one-dimensional, order- $n$  Fourier cosine basis

consists of the  $n + 1$  features:

$$x_i(s) = \cos(i\pi s), \quad s \in [0, 1] \quad (69)$$

for  $i = 0, \dots, n$ .

### 9.5.3 Coarse Coding

Consider a two dimensional state space represented by the set  $s = \{s_1, s_2\}$ . We can lay the state space out in two dimensions, and map circles to the state space as in Figure ?? . If one state  $s$  falls within multiple overlapping circles, then another state covered by these circles  $s'$  will be updated with the same value update. How we choose to define these circles (other indeed other shapes) defines the degree of generalisation across states. In effect, we are saying the states that lay close to each other in state space should be updated generally in proportion to one other.

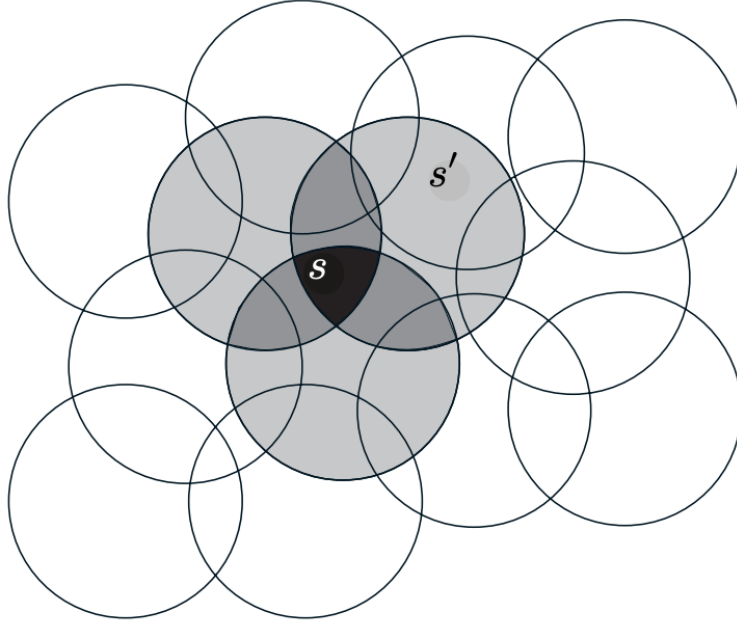


Figure 26: Coarse coding. Generalization from state  $s$  to state  $s_0$  depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

### 9.5.4 Tile Coding

Tile coding is similar to course coding, but now uses sets of overlapping, uniformly or asymmetrically distributed, tiles - Figure ?? . Tiles do not need to be squares, they can be irregular shapes, horizontal/vertical lines or log lines. Each affects the mode of generalisation in function approximation.

### 9.5.5 Radial Basis Functions

Radial Basis Functions (RBFs) are the natural extension of course coding to continuous-valued domains. Rather than a feature being either off or on (0 or 1), it can take any

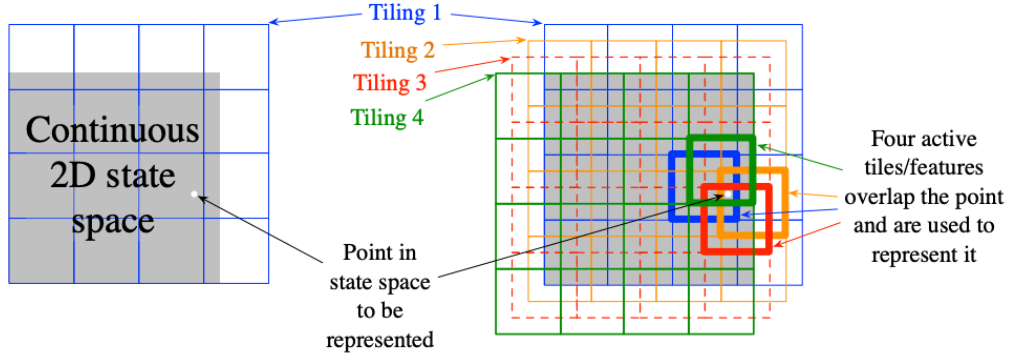


Figure 27: Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

value in the interval  $[0,1]$ . Typical RBFs use Gaussian's to parametrise the distribution of the feature, which has centre state  $c_i$  and width  $\sigma_i$ . The feature value therefore takes the form:

$$x_i(s) \doteq \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right) \quad (70)$$

Figure ?? shows the overlapping distribution of RBF features.

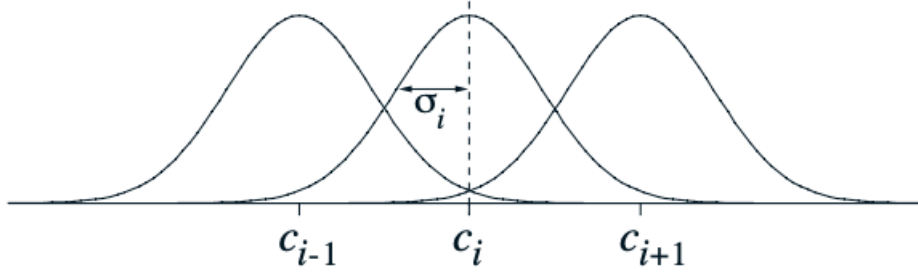


Figure 28: One-dimensional radial basis functions.

## 9.6 Selecting Step-Size Parameters Manually

Can we say in general how best to select  $\alpha$  in our function approximator? A good rule of thumb for setting the step-size parameter of linear SGD methods is:

$$\alpha \doteq (\tau \mathbb{E}[\mathbf{x}^T \mathbf{x}])^{-1} \quad (71)$$

where  $\mathbf{x}$  is a random feature vector chosen from the same distribution as input vectors will be in the SGD, and  $\tau$  is the number of experiences within which you would like to learn.

## 9.7 Nonlinear Function Approximation: Artificial Neural Networks

Artificial Neural Networks (ANNs) have proven to be some of the best non-linear function approximators. We know plenty about them already, so here is some general points:

- ANNs overfit on training data, a problem mitigated by dropout.
- Batch normalization involves normalising the output of hidden layers before passing them to the next layer, and performing this process on mini-batches of data. This improves the learning rates of the algorithms.

## 9.8 Least-Squares TD

Earlier, we established the TD fixed point as

$$\mathbf{w}_{TD} \doteq \mathbf{A}^{-1}\mathbf{b} \quad (72)$$

where

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \text{ and } \mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T] \in \mathbb{R}^{d \times d} \quad (73)$$

Instead of computing this solution iteratively, we could compute estimates of  $A$  and  $b$  and then directly compute the TD fixed point— this is the Least Squares TD algorithm (LSTD). It forms the estimates:

$$\hat{\mathbf{A}}_t \doteq \sum_{k=0}^{t-1} \mathbf{x}_k(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^T + \epsilon\mathbf{I} \text{ and } \hat{\mathbf{b}}_t \doteq \sum_{k=0}^{t-1} R_{k+1}\mathbf{x}_k \quad (74)$$

where  $\mathbf{I}$  is the identity matrix, and  $\epsilon\mathbf{I}$ , for some small  $\epsilon > 0$  ensures that  $\hat{\mathbf{A}}_t$  is always invertible. It looks like both  $\hat{\mathbf{A}}_t$  and  $\hat{\mathbf{b}}_t$  should be divided by  $t$  to achieve the expectation, but the  $t$ s cancel when used to calculate the TD fixed point as

$$\mathbf{w}_{TD} \doteq \hat{\mathbf{A}}^{-1}\hat{\mathbf{b}} \quad (75)$$

The inversion of the matrix usually requires  $O(d^3)$  computation, but can be done with  $O(d^2)$  using the *Sherman-Morrison formula*. LSTD's lack of step size parameter means it never forgets, which is sometimes desirable, but often not in RL problems where both the environment and policy change with time.

## 9.9 Memory-based Function Approximation

So far we have been parametrisng functions that approximate our value function, and these parameters are updated as we see more data. These are examples of *parametric* models. An alternative approach is to store examples of transitions or rewards from one state to the next, and querying them when we arrive in a state, similar to the approach used in Dyna - Chapter 8. These are *nonparametric* models or *memory-based* function approximators, with examples including nearest neighbours, or locally weighted regressions, with the weights being proportional to the distance between current state and queried state.

Because these methods are *local* approximations rather than *global*, they allow us to somewhat overcome the curse of dimensionality, and only approximate the value function where it is required, rather than across the entire state space. For example, if we are working with a state space with  $k$  dimensions, a tabular method, storing a global approximation requires memory exponential in  $k$ , whereas storing examples in a memory-based method requires only memory proportional to  $k$ , or linear in the number of examples  $n$ .

## 9.10 Kernel-based Function Approximation

In the previous subsection, we discussed weighted average regression, where the weight is proportional to some measure of distance. The function that assigns the weights is often called the *kernel function* or simply a *kernel* and takes the form  $k : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$  and is summarised by  $k(s, s')$ . Viewed slightly differently,  $k(s, s')$  is a measure of the strength of generalization from  $s'$  to  $s$ .

*Kernel regression* is the memory-based method that computes a kernel weighted average of the targets of *all* examples stored in memory, assigning the result to the query state. If  $\mathbb{D}$  is the set of stored examples, and  $g(s')$  denotes the target for state  $s'$  in a stored example, then the value function is

$$\hat{v}(s, \mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s') g(s') \quad (76)$$

The common kernel is the Gaussian radial basis function (RBF) as defined earlier and in the Gaussian Process literature. Kernel based methods allow us to operate in the high-dimensional feature space, only using stored examples, and without needing a complex, parametrised model. This is the so-called *kernel trick* and is the basis of many modern machine learning methods.

## 9.11 Looking Deeper at On-policy Learning: Interest and Emphasis

Until now we have treated all encountered states equally, but it is likely that we will actually be more interested in some states than others. To deal with this, we introduce two new concepts called *interest* and *emphasis*. Interest is the degree to which we value an accurate estimate of the value of a given state, generally  $I_t \in [0, 1]$  and emphasis is a scalar that multiplies the learning update and thus emphasises or de-emphasises the learning done at time  $t$ . The emphasis is determined recursively from the interest by

$$M_t = I_t + \gamma^n M_{t-n} \quad (77)$$

## 9.12 Key Takeaways

- *Generalization* of RL systems is critical to be applicable to artificial intelligence and large engineering problems. Generalization is achieved by approximating the value function using various techniques.
- We defined the prediction objective  $\bar{V}E(\mathbf{w})$ , called the *mean squared value error*, which provides a rational basis for comparing different function approximators.
- Most techniques use stochastic gradient descent (SGD) to find the set of weight parameters than minimise  $\bar{V}E(\mathbf{w})$
- Linear methods can be proven to converge to the globally optimum weight vector under certain conditions.
- Features, used to build linear methods, can take many forms, including: polynomials, fouriers, coarse codings, tile codings, and RBFs.
- Artificial NNs have been proven to be potentially the most effective nonlinear function approximator.

- Instead of using *parametric* function approximators, *nonparametric* models can be used that predict values based on the distance of a current state to past observed next states for which we have some stored values, or state representations.
- *Interest and Emphasis* allow us to focus the function approximation on parts of the state space where we would value more accurate estimates.