# Reinforcement Learning: An Introduction - Notes

Scott Jeen

April 5, 2021

## Contents

# 1 Introduction

## 1.1 Overview

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

## 1.2 Elements of Reinforcement Learning

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

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

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

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

# 2 Multi-arm Bandits

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

## 2.1 An n-Armed Bandit Problem

**The problem**:

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

## 2.2 Action-Value Methods

**The estimated action value**

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

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

**The greedy action selection method:**

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

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

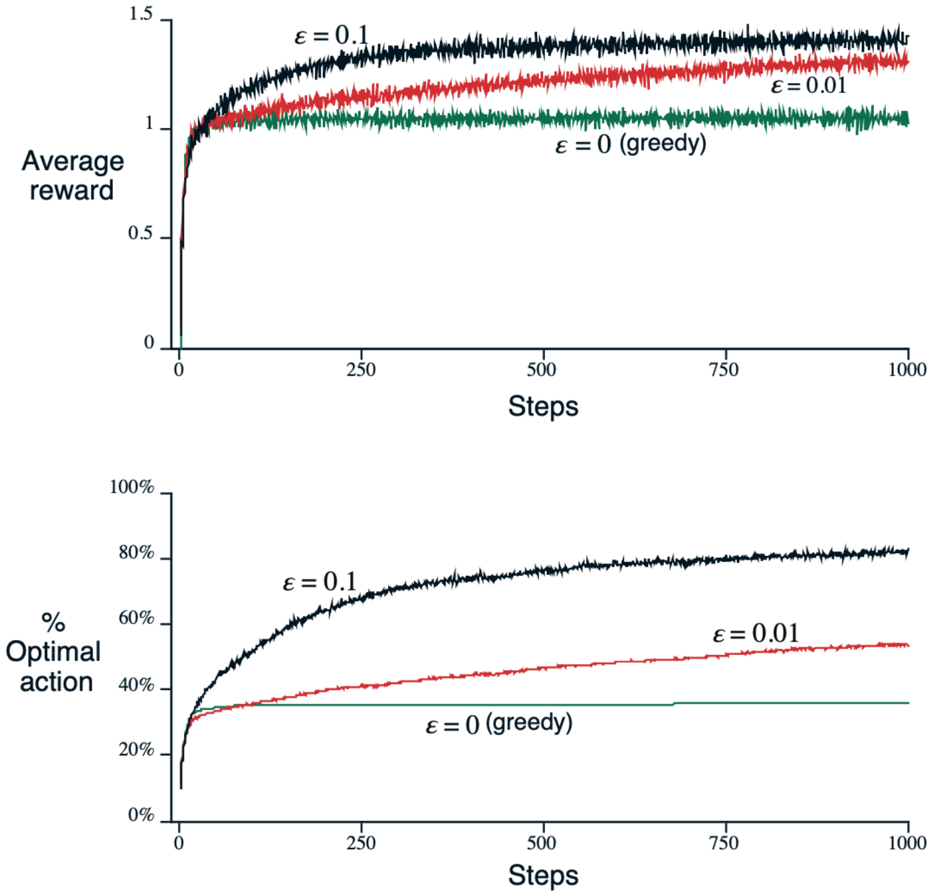


*Figure 1: tbc*

### 2.3 Incremental Implementation

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

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

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

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

$\alpha$ 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.

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

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

$$\begin{aligned}
Q_{k+1} &= Q_k + \alpha\,[R_k - Q_k] \\
&= \alpha R_k + (1-\alpha)Q_k \\
&= \alpha R_k + (1-\alpha)[\alpha R_{k-1} + (1-\alpha)Q_{k-1}] \\
&= \alpha R_k + (1-\alpha)\alpha R_{k-1} + (1-\alpha)^2 Q_{k-1} \\
&= \vdots \\
&= (1-\alpha)^k Q_1 + \sum_{i}^{k} \alpha(1-\alpha)^{k-i} R_i \qquad (8) \\
& \qquad (9)
\end{aligned}$$

- 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 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 it's samples because they are less than the initial estimate and so will explore elsewhere until the values converge.
- The above method of exploration is called *Optimistic Initial Values*.

## 2.6 Upper-confidence-bound Action Selection

$\epsilon$-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:

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

where c > 0 controls the degree of exploration.

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

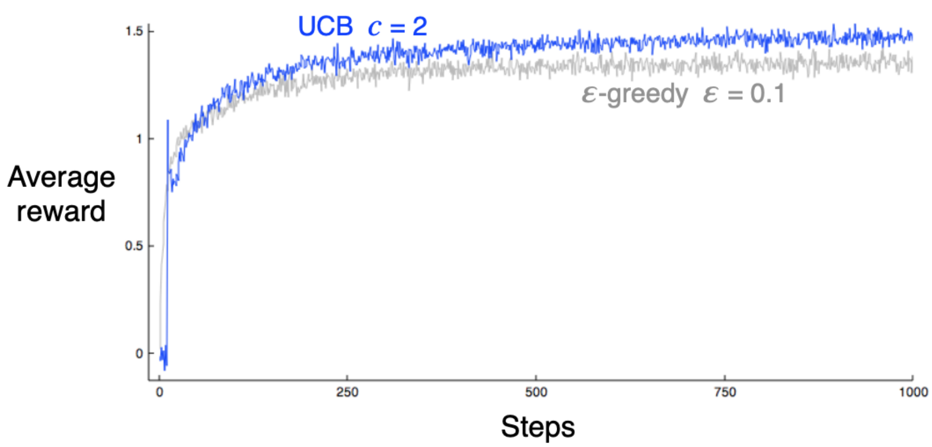


*Figure 2: UCB performs better than e-greedy in the n-armed bandit problem*

## 2.7 Associative Search (Contextual Bandits)

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

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

## 2.8 Key Takeaways

- The value of an action can be summarised by $Q_t(a)$, the sample average return from an action
- When selecting an action, it is preferable to maintain exploration, rather than only selecting the action we believe to be most valuable at any given timestep, to ensure we continue to improve our best estimate of the optimal action. We do so using $\epsilon$)-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 3.

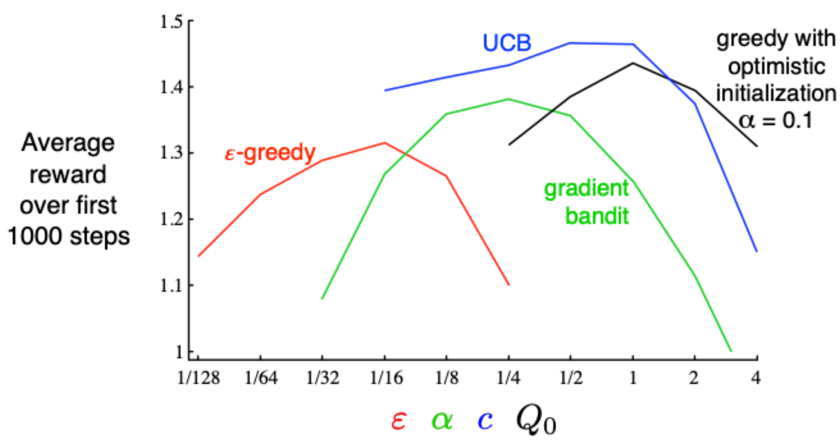


*Figure 3: Performance of each of the bandit algorithms explored in this chapter*

# 3 Finite Markov Decision Processes

## 3.1 The Agent-Environment Interface

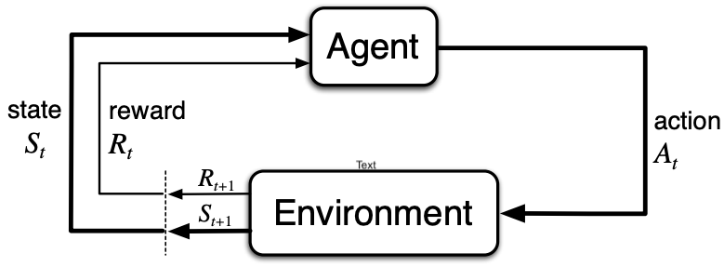


*Figure 4: The agent-environment interface in reinforcement learning*

- At each timestep the agent implements a mapping from states to probabilities of selecting a possible action. The mapping is called the agents *policy*, denoted $\pi$, 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:

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

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

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

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

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

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

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

## 3.4 Unified Notation for Episodic and Continuing Tasks

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

## 3.5 The Markov Property

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

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

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

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

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

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

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

### 3.6 Markov Decision Process (MDP)

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

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

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

### 3.7 Policies and Value Functions

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

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

i.e. the expected future rewards, given state $S_t$, and policy $\pi$. 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:

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

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

- 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**:

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

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

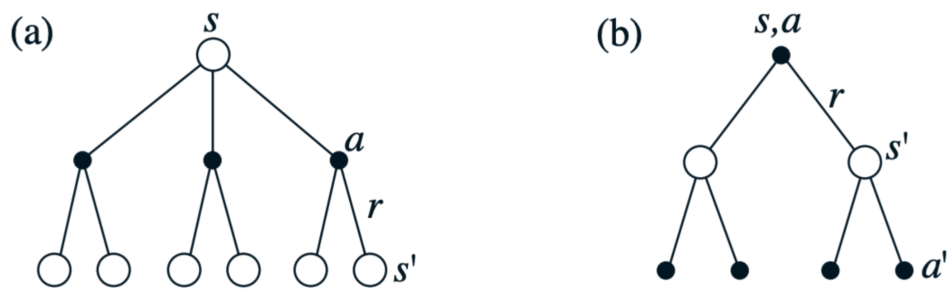


*Figure 5: Backup diagrams for $v_\pi$ and $q_\pi$*

### 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 $v*$:

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

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

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

and secondly for $q_*$:

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

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

## 3.9 Optimality and Approximation

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

## 3.10 Key Takeaways

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

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

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

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

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

## 4.2 Policy Improvement

We can obtain a value function for an arbitrary policy $\pi$ 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:

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

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

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

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

## 4.3 Policy Iteration

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

1. Evaluate policy $\pi$ 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:

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

*Figure 6: Iterative policy evaluation and improvement*

## 4.4 Value Iteration

## 4.5 Asynchronous Dynamic Programming

## 4.6 Generalised Policy Iteration

## 4.7 Efficiency of Dynamic Programming

## 4.8 Key Takeaways