TheRLProblem

August 30, 2018

The Reinforcement Learning Problem

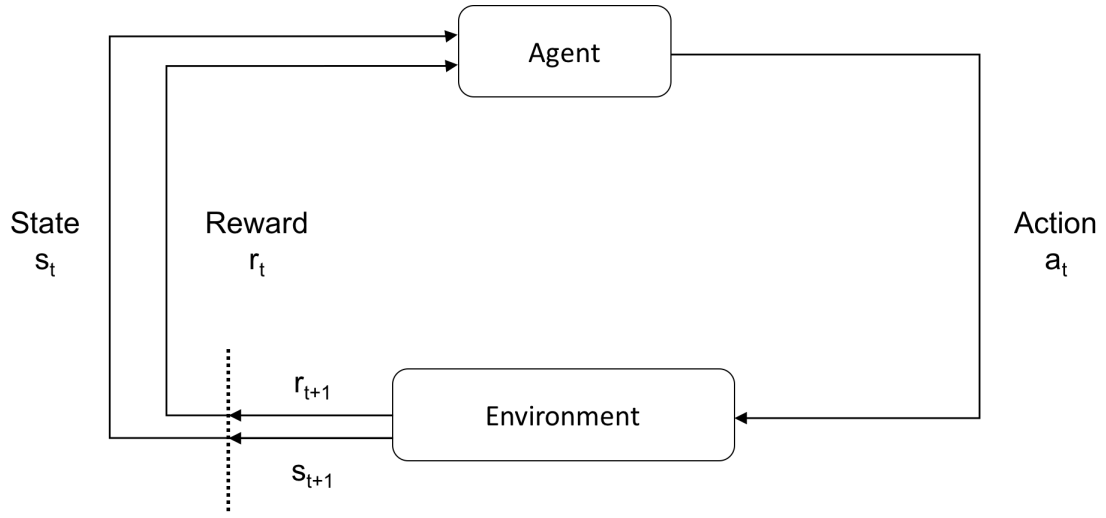
In this notebook I hope to introduce you to the basic components of the reinforcement learning (RL) problem. A good grasp of these components will make the approaches to solving the problem in later notebooks much easier to understand. We call it the RL problem because RL is best characterised by the problem it is trying to solve rather than by any specific method or approach. In general terms, RL is concerned with how an agent interacts with an environment in order to achieve maximum reward. Importantly, there are three main signals in the RL problem that characterise this interaction between the agent and the environment:

1. s_t - The state of the environment at any given point in time ($s_t \in S$)
2. a_t - The action chosen by the agent given the state of the environment ($a_t \in A$)
3. r_{t+1} - The reward recieved by the agent as a result of the state (s_t), action (a_t) and subsequent state of the environment s_{t+1} ($r_{t+1} \in \Re$)

Where S and A are the sets of all possible states and actions respectively. How these signals characterise the interaction between the agent and its environment is often represented graphically as follows:

```
In [1]: from IPython.display import Image
        Image(filename='RLProblem.png')
```

```
Out[1]:
```



From this diagram we can see that the interaction is a constant loop. The agent chooses an action based on the environmental state, which the environment then responds to by genarating a new state and an appropriate reward signal. It is this reward signal that provides the agent with the feedback it needs to evaluate its actions and ultimately make decisions that lead to the most reward. Indeed, the goal of any reinforcement learning algorithm is to maximise the amount of reward it obtains. This is often called maximising the expected 'return', where the return is some function R_t of the reward sequence that the agent experiences from time t onwards. An extremely common way to define the return R_t is to make it equal to the sum of discounted future rewards:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \end{aligned}$$

Where γ is called the 'discount factor' and lies between 0 and 1 inclusive ($0 \leq \gamma \leq 1$). The discount factor is responsible for controlling how far- or near-sighted the agent is. As γ approaches 1 the agent will become increasingly far-sighted because it values rewards in the future more and more. In contrast if γ is set to 0 then the agent will only consider the immediate reward at time $t + 1$ when choosing actions. It is worth noting that discounting using γ is neccessary for problems that are not episodic i.e. have no terminal end point. This is because γ ensures that the return at any given time point t will converge to a finite value as k approaches ∞. In episodic problems this is not an issue because the sum of reward values is finite.

One question that you may have about the environment-agent loop is where exactly does the boundary between the environment and the agent exist? Commonly the boundary is drawn so that anything that the agent doesn't have absolute control over is considered the environment. For example in a simple robot that has to learn how to move, the motors may be considered part of the environment because the robot doesn't have perfect control over them. This illustrates how the boundary between the agent and the environment doesn't necessarily have to be a physical one.

It is often helpful to look at how this RL problem differs from standard supervised learning. Firstly, the agent may not recieve reward at every time step i.e. the rewards are *sparse*. This is in contrast to standard supervised learning where every training example has a label that represents the target for the machine learning algorithm. In the RL problem an agent may need to choose an action in order to receive a reward at a later date. Indeed there are often cases in RL where an agent may need forgo reward in the present in order to achieve larger reward in the future. Secondly, supervised learning is genrally concerned with single i.i.d datasets that do not change. In comparison, in the RL problem the data the agent is recieving is dependent on the agent's actions and so the input distribution is dependent on the predictions of the machine learning algorithm.

The simple formulation of the RL problem outlined above is extremely flexible and can be applied to many different scenarios depending on how you choose to define the different signals. One key assumption of most RL algorithms is that the state signal s_t follows the Markov property, which we shall discuss in the next part of this notebook. Critically, the Markov property allows us to frame the RL problem as a Markov Decision Process, which forms the theoretical basis for the majority of the solution methods in RL. Therefore a good understanding of this assumption is essential if you want to represent your problem in a way that allows RL methods to solve it.

Markov Decision Processes

In RL we commonly treat the state at any given time point (s_t) as a Markov state, which means that it satisfies the Markov property. If a state satisfies the markov property then the probability of future states given past and present states only depends on the current state. Another way of saying this is that future outcomes are only a function of the current state and not of previous states. Mathematically we can write the Markov property as follows:

$$P(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, ...) = P(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t)$$

$$\forall s', r, s_t, a_t$$

That is the probability of the next state being equal to s' and the next reward being equal to r given all previous outcomes is equal to the probability of the next state being equal to s' and the next reward being equal to r given the current state and action. This basic assumption allows us to greatly simplfy the RL problem because now the agent only needs to consider it's current state in order to select an action. In theory the agent should be able to pick the action that maximises the expected return without considering everything that has happened before!

Of course there are many real-world problems where the Markov property simply doesn't hold and past states do have an impact upon the probability of future states. Many RL algorithms will perform well even when the states it encounters are not strictly Markov. However the closer we can get the states to satisfying the Markov property the better the RL algorithm will perform. Often we can construct the state signal so that it approximates a Markov state. For example, if you wanted to train an agent to play a video game using RL then it is common practice to provide the agent with the last four game frames as the state signal. This provides the agent with enough information to infer what objects are present on the screen, what direction they are moving in and how fast they are moving. These key variables should provide enough information for the agent to choose optimal actions, with anything preceeding the four game frames being largely irrelevant.

Now that we have assumed that our states satisfy the Markov property we can frame the RL problem as a Markov Decision Process (MDP). There are five main components of a MDP, some of which we have already encountered:

1. S - The set of all possible states ($s_t \in S$)
2. A - The set of all possible actions ($a_t \in A$)
3. $P^a_{ss'}$ - The transition function
4. $R^a_{ss'}$ - The reward function
5. γ - The discount factor ($0 \leq \gamma \leq 1$)

$P^a_{ss'}$ is called the transition function and it defines the probability distribution over the next state given the current state and action. Mathematically we can define this as:

$$P^a_{ss'} = P(s_{t+1} = s' \mid s_t = s, a_t = a)$$

$R^a_{ss'}$ specifes the expected reward value for transitioning from state s to state s' via action a. Mathematically we can define this as:

$$R^a_{ss'} = \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s']$$

Importantly, $P^a_{ss'}$ and $R^a_{ss'}$ provide us with all the information we need to describe the dynamics of the environment and to make decisions that maximise the expected return. Notice how both $P^a_{ss'}$ and $R^a_{ss'}$ rely on the Markov property of the state signal, both of these components would be much more complicated if the Markov property didn't hold!

It is often useful to work through a simple example of an MDP so that you can see how each of the components is defined. A classic example, which can be found in the book 'Reinforcement Learning: An Introduction' by Sutton and Barto (1998), is that of a recycling robot. Lets say that we have a simple robot that can choose from the following actions at discrete points in time:

1. Search for a can to recycle
2. Stay still and wait for someone to bring it a can
3. Return to base to recharge its battery

The robot uses its battery levels (high or low) to decide which action to take. Straight away we can see that these pieces of information defines our set of state signals S and actions A:

$$\begin{aligned} S &= \{high, low\} \\ A(high) &= \{search, wait\} \\ A(low) &= \{search, wait, recharge\} \end{aligned}$$

Note how our action set is a function of the current state s, if the robot has a high battery level then it makes no sense to recharge further. Now that we have S and A defined for our problem we need to also define the environment's dynamics with $P^a_{ss'}$ and $R^a_{ss'}$. For many RL problems, $P^a_{ss'}$ and $R^a_{ss'}$ may be parametric functions but in this simple case we can define them in a tabular manner as follows:

s	s'	a	Transition Function $P^a_{ss'}$	Reward Function $R^a_{ss'}$
high	high	search	α	R^{search}
high	low	search	$1-\alpha$	R^{search}
low	high	search	$1-\beta$	-3
low	low	search	β	R^{search}
high	high	wait	1	R^{wait}
high	low	wait	0	R^{wait}
low	high	wait	0	R^{wait}
low	low	wait	1	R^{wait}
low	high	recharge	1	0
low	low	recharge	0	0

This table details the values of $P^a_{ss'}$ and $R^a_{ss'}$ for all possible combinations of s, s' and a. For example in the case of $P^a_{ss'}$, if the robot starts off with a high battery level and decides to search for a can then the robot's battery level will remain high with probability α or drop to low with probability $1-\alpha$. In contrast, if the robot has a low battery level and decides to search then the battery level will remain low with probability β or deplete with propbability $1-\beta$. For the other combinations, deciding to wait does not change the battery level and deciding to recharge increases the battery level from low to high. In the case of $R^a_{ss'}$, choosing to search or wait will return an expected number of cans, denoted R^{search} and R^{wait} respectively. The exception occurs when the robot chooses to search with a low battery and the battery depletes as a result, requiring someone to rescue and recharge it ($s = low$, $s' = high$, $a = search$). When this happens the reward is -3 to represent an unwanted outcome. If the robot decides to recharge then the reward is simply 0 because no cans will be collected.

Hopefully this simple description of a recycling robot and how it can be formulated as an MDP is useful for cementing what each of the components mean. It is suprising how many problems can be represented as an MDP with just these simple components. The real power of RL comes from being able to train agents, such as the recycling robot, to obtain alot of reward over time i.e. pick up alot of cans without needing to be rescued. In order to do this most RL algorithms rely on something called a value function, which we shall explore next.

Value Functions

Now that we have covered MDPs we are ready to talk about value functions, which represent one of the most critical concepts in RL. At the heart of most solution methods in RL you will find some form of value function. In simple terms, value functions estimate the expected return from a given state with respect to a particular policy. Policies, commonly denoted π are a mapping from states to actions and they specify which action an agent will take given the current state:

$$\pi : s \mapsto a$$

π may be a detemerministic or stochastic mapping but either way it specifies how the agent acts in each state. Value functions are dependent on the policy π because the expected return from a given state will be highly dependent on the action the agent will take in that state and also future

states. Typically we can define two different types of value function, either a state-value function $V^\pi(s)$ or an action-value function $Q^\pi(s,a)$. A state-value function estimates the expected return of a given state s, whereas an action-value function estimates the expected return of an action a from a given state s. Using the discounted sum of future rewards as our return we can define these value functions as follows:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[R_t \mid s_t = s] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s] \\ Q^\pi(s,a) &= \mathbb{E}_\pi[R_t \mid s_t = s, a_t = a] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a] \end{aligned}$$

A key idea in RL is that we can now take these value functions and use the components of an MDP to show that they satisfy particular recursive relationships. Using the state-value function as an example, we can now define it as a function of itself using $P^a_{ss'}$ and $R^a_{ss'}$:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s] \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_{t+1} = s']] \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \end{aligned}$$

As you can see, by averaging over the policy and the one-step dynamics of the MDP we can define the state-value function in terms of itself. In other words, the value of any given state is simply the reward obtained immediately from that state plus the value (expected return) of the next state, averaged over the policy and one-step dynamics of the environment. We can equally define a recursive relationship for the action-value function:

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a] \\ &= \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_{t+1} = s']] \\ &= \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')] \end{aligned}$$

Here the action-value function is defined as the immediate reward given a specific action in the current state, averaged over the environment dynamics, plus the value of actions in the successor state, averaged over the environment dynamics and the agents policy.

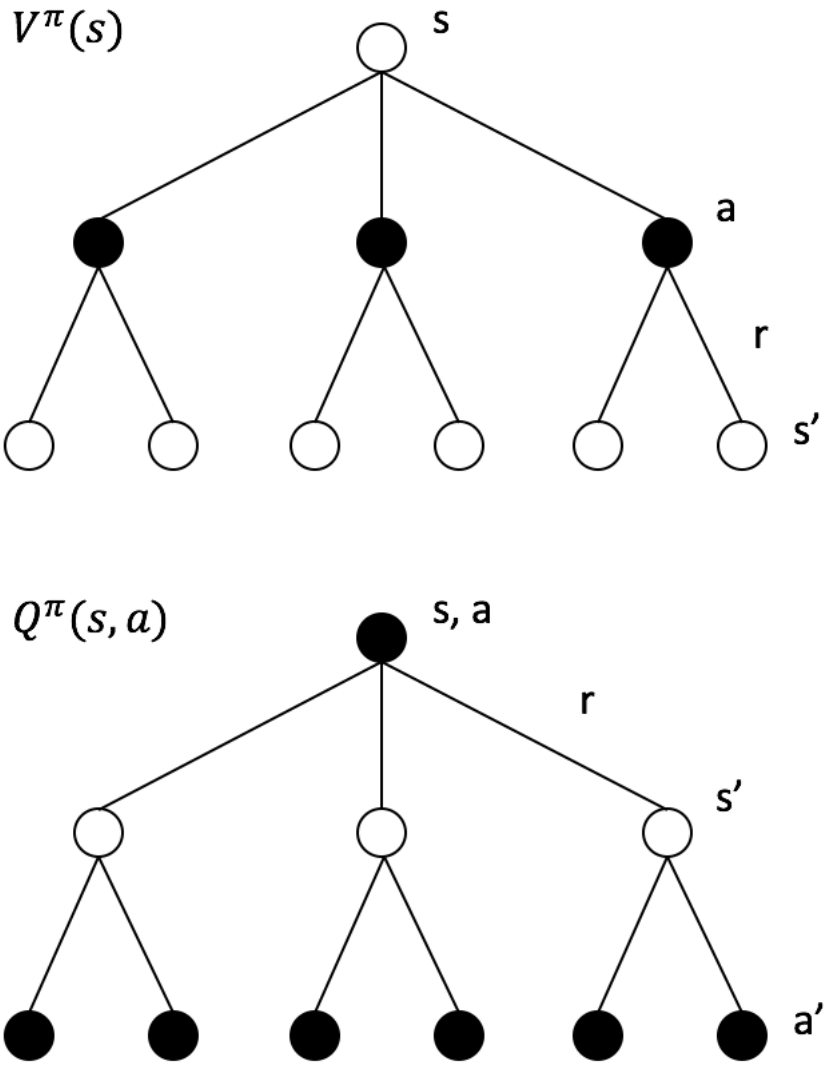
The following two recursive equations are known as the Bellman equations and form the basis of many RL solution methods including dyanmic programming and temporal difference learning.

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{\pi}(s')]$$
$$Q^{\pi}(s,a) = \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^{\pi}(s',a')]$$

A particularly nice way to visualise what these bellman equations mean is to use 'back-up' diagrams. Back-up diagrams were first introduced by Sutton and Barto in their seminal book 'Reinforcement Learning: An Introduction'. The diagrams depict states as open circles and actions as filled in circles. The top of the diagram represents the current state, or state-action pair, and the nodes extending downwards represent possible outcomes. Below are the back up diagrams for $V^{\pi}(s)$ and $Q^{\pi}(s,a)$:

```
In [2]: from IPython.display import Image
        Image(filename='RLBellmanBackups.png')
```

```
Out[2]:
```



The bellman equation for $V^{\pi}(s)$ averages over all possible actions in the current state and the value of the resulting successor states, while $Q^{\pi}(s,a)$ averages over the resulting successor states given a single action followed by the value of all the possible actions in those successor states. In both cases the bellman equations use 'one-step lookaheads' to calculate the value function, averaging over all possible outcomes according to their probability of occuring.

Optimal Value Functions

So far in this notebook we have outlined the general principles behind the RL problem but what does it mean to actually solve the RL problem? In general, solving the RL problem equates to finding a policy π (a mapping from states to actions) that achieves alot of reward over time. An

optimal policy, denoted π^*, is one that achieves an expected return that is greater than or equal to all other policies for all states. If we can find the optimal policy (there may be more than one) then we can achieve the largest expected return and therefore solve the RL problem.

Remember that value functions are dependent on a particular policy because an agent's actions will determine the expected return from a state. We can therefore define an optimal state value function $V^*(s)$ or action value function $Q^*(s,a)$, where the expected return of a state or state-action pair is dependent on the optimal policy:

$$V^*(s) = \max_\pi V^\pi(s) \qquad \text{for all } s \in S$$
$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \qquad \text{for all } s \in S \text{ and } a \in A(s)$$

Just as we did with the normal value functions, we can write these optimal value functions as recursive equations. The recursive equations for the optimal value functions are known as the Bellman optimality equations. They differ slightly from the normal Bellman equations because they are written without reference to a specific policy. Instead the Bellman optimality equations rely on the fact that the value of a state w.r.t the optimal policy is the same as the expected return for the best action from that state. The Bellman optimality equations for $V^*(s)$ and $Q^*(s,a)$ are therefore as follows:

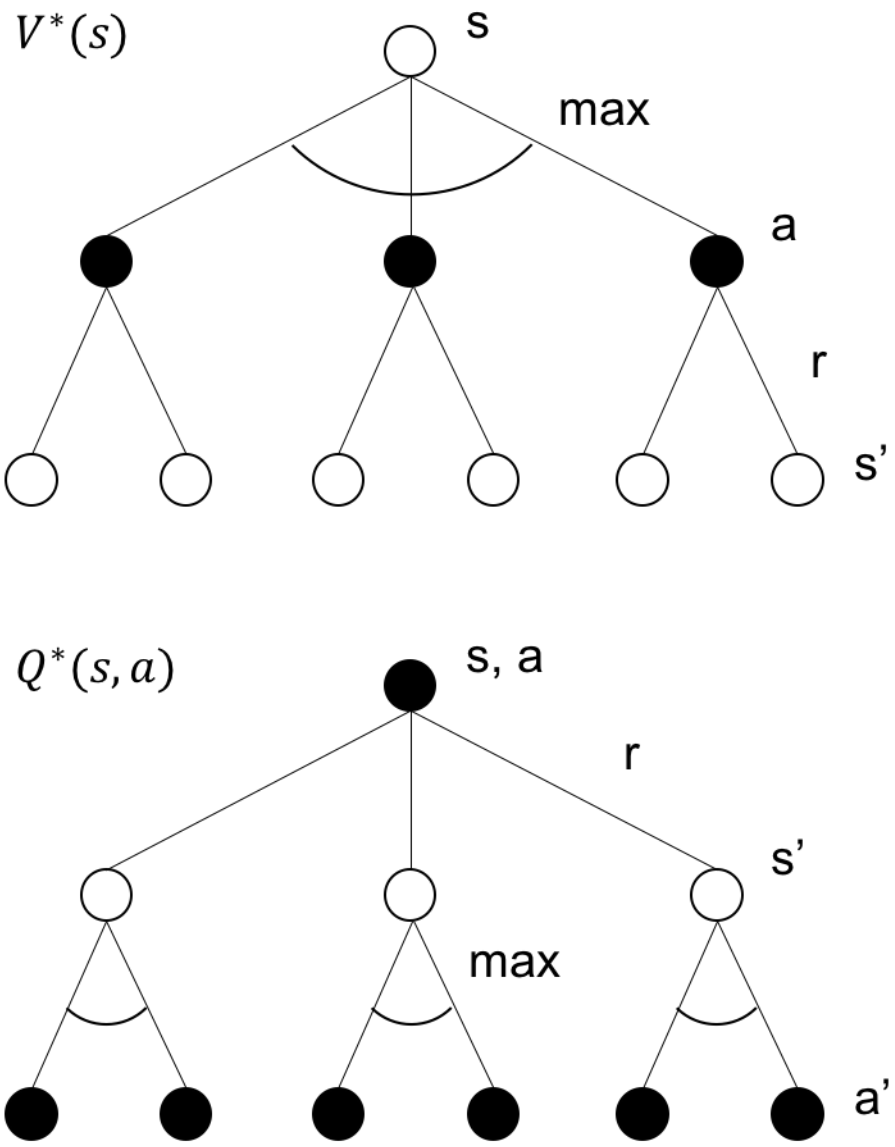
$$\begin{aligned}
V^*(s) &= \max_{a \in A(s)} Q^{\pi^*}(s,a) \\
&= \max_a \mathbb{E}_{\pi^*}[R_t \mid s_t = s, a_t = a] \\
&= \max_a \mathbb{E}_{\pi^*}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a] \\
&= \max_a \mathbb{E}_{\pi^*}[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a] \\
&= \max_a \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a] \\
&= \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')]
\end{aligned}$$

$$\begin{aligned}
Q^*(s,a) &= \mathbb{E}[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a] \\
&= \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a')]
\end{aligned}$$

Again we can use backup diagrams to visualise these equations and to see how the Bellman optimality equations differ from the normal Bellman equations. In these diagrams the arcs represent taking the max over the possible choices, as opposed to taking the expectation w.r.t some policy.

```
In [4]: from IPython.display import Image
        Image(filename='RLBellmanOptimalityBackups.png')
```

Out[4]:



Importantly, if we can obtain the optimal value function (V^* or $Q^*(s,a)$) for a given MDP then the optimal policy π^* is the policy that acts greedily (i.e. chooses the largest value) w.r.t to the optimal value function. In the case of the optimal state value function, the best action in any given state is the one that maximises the sum of the immediate reward and the discounted value of the next state:

$$\pi^*(s) = \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')]$$

This is akin to making a one-step lookahead search and so we just need to act greedily according to this search. Taking greedy actions works because the optimal state value function takes into

account the reward consequences of all possible future behaviours. This is powerful because it means we don't need to evaluate lots of future actions in order to make an optimal decision. For the action-value function, the best action is simply the one that has the largest value from the current state:

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

In the case of the optimal action value function we don't even need to perform a one-step lookahead search. $Q^*(s, a)$ stores the results of all one-step lookahead searches and provides the optimal expected long-term return as a locally and immediately available value. So by representing the value function as a function of states and actions, rather than just states, we can choose optimal actions without having to know anything about the environment's dynamics! This is an extremely powerful result and comes up time and time again in RL.

So if we have the optimal value function (particularly the optimal action value function) then we also have the optimal policy and have solved the RL problem. The question arises then, how do we find the optimal value function? Indeed it is this exact question that many RL solution methods aim to answer. We can infact solve for the optimal value function as a series of N nonlinear equations in N unknowns, where N is the number of states, as long as $P^a_{ss'}$ and $R^a_{ss'}$ are known. Unfortunately, this exhaustive search approach is rarely possible due to three main reasons:

1. We often do not know the evironment's dynamics i.e. $P^a_{ss'}$ and $R^a_{ss'}$
2. The amount of computational resources required becomes infeasible as N grows
3. The Markov property is often not satisfied

For any given RL problem it is usually the case that one, or a combination, of these problems arises. We therefore need other methods to solve for the optimal value function. These methods will form the basis of the next few notebooks, with all of them generally relying on iterative procedures to find a solution.

I hope this notebook has helped to outline the main components of the RL problem. Half the battle of understanding RL is getting familiar with the terminology that has been used here. A general understanding of MDPs and value functions in particular, will stand you in good stead for diving further into RL. If you have any questions then please do not hesitate to email me!