Reinforcement Learning

10 January 2021 10:10

Objectives:

- Utility, rewards, and values
- Decision-theoretic planning and Markov Decision Processes
- State-based reinforcement learning algorithms: Q-learning and SARSA
- The explore-exploit dilemma and solutions
- On-poly and off-policy reinforcement learning

AIFCA 12.1-12.7, 9.5, AIMA 21

See also https://www.davidsilver.uk/teaching/ for more detailed explanations of MDP's and Policies.

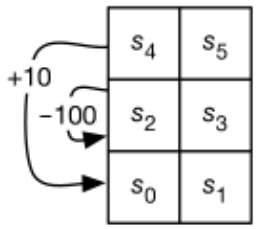
A RL agent acts in an environment, observing its state and receiving rewards. From its perceptual and reward information, it must determine what to do.

The learning agent is given the possible states and the set of actions it can carry out
At each time the agent observes the state of the environment and the reward received. We are assuming the environment is fully-observable.
At each time, after observing the state and reward, the agent carries out an action.
The goal of the agent is to maximize its discounted reward, for some discount factor.

RL can be formalized in terms of Markov Decision Processes, in which the agent initially only knows the set of possible states and the set of possible actions.

The dynamics, $P(s'|a,s)$, and the reward function, $R(s,a)$, are not given to the agent. As in an MDP, after each action, the agent observes the state it is in and receives a reward.

We can imagine this as a tiny RL environment:



The agent knows only that it has four actions, and what state it is in.
he agent does not know how the states are configured, what the actions do, or how rewards are earned.

RL is Hard

- The **credit assignment problem** or the blame attribution problem is the problem of determining which action was responsible for a reward or punishment. The action responsible may have occurred a long time before the reward was received. Moreover, not just a single action but rather a combination of actions carried out in the appropriate circumstances may be responsible for a reward.
- even if the dynamics of the world do not chance, the effect of an action of the agent depends on what the agent will do in future. What may initially seem like a bad thing for the agent to do may end up being an optimal action because of what the agent does in the future. This is common among planning problems, but it is complicated in the reinforcement learning context because the agent does not know a priori, the effects of its actions
- The **explore-exploit dilemma** - if an agent has worked out a good course of actions, should it continue to follow these actions? exploiting what it has determined? or should it explore more

to find better actions? An agent that never explores may act forever in a way that could have been much better if it had explored earlier.

Decision Processes

The decision networks of the previous section were for finite-stage partially observable domains. Here, we consider indefinite horizon and infinite horizon problems

Often an agent must reason about an ongoing process or it does not know how many actions it will be required to do. These are called finite horizon problems when the process may go on forever or indefinite horizon problems when the agent will eventually stop but it does not know when.

For ongoing processes, it may not make sense to only the utility at the end, because the agent may never get to the end. Instead, an agent can receive a sequence of rewards, that incorporate the action costs in addition to any prizes or penalties that may be awarded. Indefinite horizon problems can be modelled using a stopping state. A stopping state is a state in which all actions have no effects; that is, when the agent is in that state, all actions immediately return to that state with a zero reward. Goal achievement can be modelled by having a reward for entering such a stopping state.

A Markov Decision Process can be seen as a Markov Chain augmented with actions and rewards or as a decision network extended in time. At each stage, the agent decides which action to perform; the reward and resulting state depend on both the previous state and the action performed.

We only consider stationary models where the state transitions and the rewards do not depend on the time.

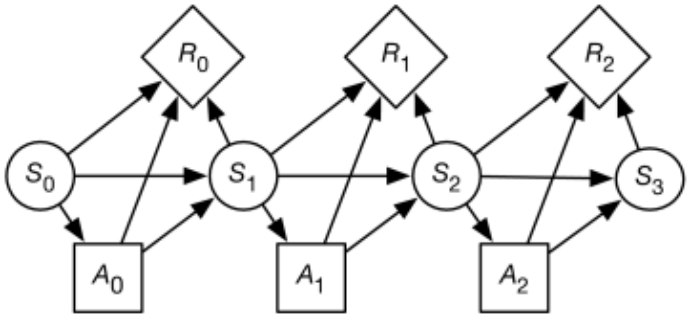
A **Markov decision process** or an **MDP** consists of

- S, a set of states of the world
- A, a set of actions
- $P : S \times S \times A \to [0, 1]$, which specifies the **dynamics**. This is written as $P(s' \mid s, a)$, the probability of the agent transitioning into state s' given that the agent is in state s and does action a. Thus,

$$\forall s \in S \; \forall a \in A \; \sum_{s' \in S} P(s' \mid s, a) = 1.$$

- $R : S \times A \times S \to \Re$, where $R(s, a, s')$, the **reward function**, gives the expected immediate reward from doing action a and transitioning to state s' from state s. Sometimes it is convenient to use $R(s, a)$, the expected value of doing a in state s, which is $R(s, a) = \sum_{s'} R(s, a, s') * P(s' \mid s, a)$.

A Markov Decision Process can be depicted using a decision network:



With decision networks, the designer has to consider what information is available to the agent

when it decides what to do. There are two common variations:

- Fully Observable Markov Decision Processes - the agent gets to observe the current state when deciding what to do
- Partially Observable Markov Decision Processes - a combination of an MDP and a hidden markov model. At each time, the agent gets to make some ambiguous and possibly noisy observations that depend on the state. The agent only has access to the history of rewards, observations and previous actions when making a decision. It cannot directly observe the current state.

Rewards

To decide what to do, the agent compares different sequences of rewards. The most common way to do this is to convert a sequence of rewards into a number called the value, the cumulative reward or the return. To do this, the agent combines an immediate reward with other rewards in the future. Suppose the agent receives the sequence of rewards r1, r2, r3, r4...

Total reward $V = \sum_{i=1}^{\infty} r_i$. In this case, the value is the sum of all of the rewards. This works when you can guarantee that the sum is finite; but if the sum is infinite, it does not give any opportunity to compare which sequence of rewards is preferable. For example, a sequence of $1 rewards has the same total as a sequence of $100 rewards (both are infinite). One case where the total reward is finite is when there are stopping states and the agent always has a non-zero probability of eventually entering a stopping state.

Average reward $V = \lim_{n \to \infty}(r_1 + \cdots + r_n)/n$. In this case, the agent's value is the average of its rewards, averaged over for each time period. As long as the rewards are finite, this value will also be finite. However, whenever the total reward is finite, the average reward is zero, and so the average reward will fail to allow the agent to choose among different actions that each have a zero average reward. Under this criterion, the only thing that matters is where the agent ends up. Any finite sequence of bad actions does not affect the limit. For example, receiving $1,000,000 followed by rewards of $1 has the same average reward as receiving $0 followed by rewards of $1 (they both have an average reward of $1).

Discounted reward $V = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{i-1} r_i + \cdots$, where γ, the **discount factor**, is a number in the range $0 \leq \gamma < 1$. Under this criterion, future rewards are worth less than the current reward. If γ was 1, this would be the same as the total reward. When $\gamma = 0$, the agent ignores all future rewards. Having $0 \leq \gamma < 1$ guarantees that, whenever the rewards are finite, the total value will also be finite.

The discounted reward can be rewritten as

$$\begin{aligned} V &= \sum_{i=1}^{\infty} \gamma^{i-1} r_i \\ &= r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{i-1} r_i + \cdots \\ &= r_1 + \gamma \left(r_2 + \gamma \left(r_3 + \cdots \right) \right). \end{aligned}$$

Suppose V_k is the reward accumulated from time k:

$$\begin{aligned} V_k &= r_k + \gamma \left(r_{k+1} + \gamma \left(r_{k+2} + \cdots \right) \right) \\ &= r_k + \gamma V_{k+1}. \end{aligned}$$

To understand the properties of V_k, suppose $S = 1 + \gamma + \gamma^2 + \gamma^3 + \cdots$, then $S = 1 + \gamma S$. Solving for S gives $S = 1/(1-\gamma)$. Thus, with the discounted reward, the value of all of the future is at most $1/(1-\gamma)$ times as much as the maximum reward and at least $1/(1-\gamma)$ times as much as the minimum reward. Therefore, the eternity of time from now only has a finite value compared with the immediate reward, unlike the average reward, in which the immediate reward is dominated by the cumulative reward for the eternity of time.

In economics, γ is related to the interest rate: getting $\$1$ now is equivalent to getting $\$(1+i)$ in one year, where i is the interest rate. You could also see the discount rate as the probability that the agent survives; γ can be seen as the probability that the agent keeps going.

Policies

In a FOMDP, the agent gets to observe its currents state before deciding which action to carry out. For now, we assume that the MDP is FO. A policy specifies what the agent should do as a function of the state it is in. A stationary policy is a function:

$\pi : S \rightarrow A$

In a non-stationary policy the action is a function of the state and the time; we assume policies are stationary.

Given a reward criterion, a policy has an expected value for every state. Let:

$V^{\pi}(s)$

Be the expected value of following pi in state s. This specifies how much value the agent expects to receive from following the policy in that state.

Policy pi is an **optimal policy** if there is no policy pi' and no state s such that V^pi'(s) > V^pi(s). That is, it is a policy that has a greater or equal expected value at every state than any other policy.

if there are 100 states and 4 actions that can be performed in each state, there are 4^100 possible stationary policies. Each policy specifies an action for each state.

For infinite horizon problems, a stationary MDP always has an optimal stationary policy. However, for finite-state problems, a non-stationary policy might be better than all stationary policies.

For example, if the agent had to stop at time n, for the last decision in some state, the agent would act to get the largest immediate reward without considering the future actions, but for earlier decisions it may decide to get a lower reward immediately to get a larger reward in the future.

Value of a Policy

Consider how to compute the expected value using the discounted reward of a policy, given a discount factor of y. The value is defined in terms of two interrelated functions:

- $V^{\pi}(s)$ is the expected value of following policy π in state s.
- $Q^{\pi}(s,a)$, is the expected value, starting in state s of doing action a, then following policy π. This is called the **Q–value** of policy π.

You can define Q and V recursively in terms of each other.

If the agent is in state s', performs action a, and arrives in state s', it gets the immediate reward of R(s, a, s'), plus the discounted future reward:

$$\gamma V^{\pi}(s')$$

When the agent is planning it does not know the actual resulting state, so it uses the expected value, average over the possible resulting states:

$$\begin{aligned} Q^{\pi}(s,a) &= \sum_{s'} P(s' \mid s,a)\,(R(s,a,s') + \gamma V^{\pi}(s')) \\ &= R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V^{\pi}(s') \end{aligned}$$

where $R(s,a) = \sum_{s'} P(s' \mid s,a)\, R(s,a,s')$.

$V^{\pi}(s)$ is obtained by doing the action specified by π and then following π:

$$V^{\pi}(s) = Q^{\pi}(s, \pi(s)).$$

Value of an Optimal Policy

Let $Q^*(s,a)$, where s is a state and a is an action, be the expected value of doing a in state s and then following the optimal policy. Let $V^*(s)$, where s is a state, be the expected value of following an optimal policy from state s.

Q^* can be defined analogously to Q^π:

$$\begin{aligned} Q^*(s,a) &= \sum_{s'} P(s' \mid s,a)\,(R(s,a,s') + \gamma V^*(s')) \\ &= R(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\,\gamma V^*(s')\,. \end{aligned}$$

$V^*(s)$ is obtained by performing the action that gives the best value in each state:

$$V^*(s) = \max_a Q^*(s,a)\,.$$

An optimal policy π^* is one of the policies that gives the best value for each state:

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

where $\arg\max_a Q^*(s,a)$ is a function of state s, and its value is an action a that results in the maximum value of $Q^*(s,a)$.

Value Iteration

Value iteration is a *method of computing an optimal policy for an MDP and its value*.

Value iteration starts at the end and works backwards, refining an estimate of either **Q*** or **V***.

There really is no end, so it uses an arbitrary end point. Let **Vk** be the value function assuming there are k stages to go

Let **Qk** be the Q-function assuming there are **k** stages to go. These can be defined recursively. Value iteration starts with an arbitrary function **V0**.

For subsequent stages, it uses the following equations to get the functions for **k+1** stages to go form the functions for k stages to go.

$$\begin{aligned} Q_{k+1}(s,a) &= R(s,a) + \gamma * \sum_{s'} P(s' \mid s,a) * V_k(s') \\ V_k(s) &= \max_a Q_k(s,a) \end{aligned}$$

It can either save the V[S] array or the Q[S,A] array. Saving the V array results in less storage, but it is more difficult to determine an optimal action, and one more iteration is needed to determine which waction results in the greatest value.

```
1: procedure Value_iteration(S, A, P, R)
2:     Inputs
3:         S is the set of all states
4:         A is the set of all actions
5:         P is state transition function specifying P(s' | s, a)
6:         R is a reward function R(s, a)
7:     Output
8:         π[S] approximately optimal policy
9:         V[S] value function
10:    Local
11:        real array V_k[S] is a sequence of value functions
12:        action array π[S]
13:    assign V_0[S] arbitrarily
14:    k := 0
15:    repeat
16:        k := k + 1
17:        for each state s do
18:            V_k[s] = max_a R(s, a) + γ * Σ_s' P(s' | s, a) * V_{k-1}[s']
19:    until termination
20:    for each state s do
21:        π[s] = arg max_a R(s, a) + γ * Σ_s' P(s' | s, a) * V_k[s']
22:    return π, V_k
```

Figure 9.16: Value iteration for MDPs, storing V

The above shows the value iteration algorithm when the V array is stored. This procedure converges no matter what the initial value function V0 is. An initial value function that approximates V* converges quicker than one that does not. The basis for many abstraction techniques for MDP's is to use some heuristic method to approximate V*, and to use this an initial seed for value iteration.

Consider this example:

Example 9.27. *Suppose Sam wanted to make an informed decision about whether to party or relax over the weekend. Sam prefers to party, but is worried about getting sick. Such a problem can be modeled as an MDP with two states, healthy and sick, and two actions, relax and party. Thus*

$$S = \{healthy, sick\}$$
$$A = \{relax, party\}$$

Based on experience, Sam estimate that the dynamics $P(s' \mid s, a)$ is given by

S	A	***Probability of*** $s' = healthy$
$healthy$	$relax$	*0.95*
$healthy$	$party$	*0.7*
$sick$	$relax$	*0.5*
$sick$	$party$	*0.1*

So, if Sam is healthy and parties, there is a 30% chance of becoming sick. If Sam is healthy and relaxes, Sam will more likely remain healthy. If Sam is sick and relaxes, there is a 50% chance of getting better. If Sam is sick and parties, there is only a 10% chance of becoming healthy.

Sam estimates the (immediate) rewards to be:

S	A	***Reward***
$healthy$	$relax$	*7*
$healthy$	$party$	*10*
$sick$	$relax$	*0*
$sick$	$party$	*2*

Thus, Sam always enjoys partying more than relaxing. However, Sam feels much better overall when healthy, and partying results in being sick more than relaxing does.

The problem is to determine what Sam should do each weekend.

Example 9.31. *Consider the two-state MDP of Example 9.27 with discount $\gamma = 0.8$. We write the value function as $[healthy\_value, sick\_value]$, and the Q-function as $[[healthy\_relax, healthy\_party], [sick\_relax, sick\_party]]$. Suppose initially the value function is $[0, 0]$. The next Q-value is $[[7, 10], [0, 2]]$, so the next value function is $[10, 2]$ (obtained by Sam partying). The next Q-value is then*

State	*Action*	***Value***
$healthy$	$relax$	*7+0.8*(0.95*10+0.05*2) = 14.68*
$healthy$	$party$	*10+0.8*(0.7*10+0.3*2) = 16.08*
$sick$	$relax$	*0+0.8*(0.5*10+0.5*2) = 4.8*
$sick$	$party$	*2+0.8*(0.1*10+0.9*2) = 4.24*

So the next value function is $[16.08, 4.8]$. After 1000 iterations, the value function is $[35.71, 23.81]$. So the Q function is $[[35.10, 35.71], [23.81, 22.0]]$. Therefore, the optimal policy is to party when healthy and relax when sick.

Asynchronous Value Iteration

A common refinement of this algorithm is **asynchronous value iteration**. Rather than sweeping through the states to create a new value function, AVI updates the states one at a time, in any order, and stores the values in a single array. AVI can store either the **Q[s,a]** array or the **V[s]** array.

it converges faster than the value iteration and is the basis of some RL algorithms. Termination can be difficult to determine if the agent must guarantee a particular error, unless it is careful about how the actions and states are selected. Often, this procedure is run indefinitely as an anytime algorithm where it is always prepared to give its vest estimate of the optimal action in a state when asked.

Asynchronous value iteration could also be implemented by storing just the $V[s]$ array. In that case, the algorithm selects a state s and carries out the update:

$$V[s] := \max_a R(s,a) + \gamma * \sum_{s'} P(s' \mid s,a) * V[s'].$$

Although this variant stores less information, it is more difficult to extract the policy. It requires one extra backup to determine which action a results in the maximum value. This can be done using

$$\pi[s] := \arg\max_a R(s,a) + \gamma * \sum_{s'} P(s' \mid s,a) * V[s'].$$

Example 9.33. *In Example 9.32, the state one step up and one step to the left of the +10 reward state only had its value updated after three value iterations, in which each iteration involved a sweep through all of the states.*

In asynchronous value iteration, the $+10$ reward state can be chosen first. Next, the node to its left can be chosen, and its value will be $0.7 * 0.9 * 10 = 6.3$. Next, the node above that node could be chosen, and its value would become $0.7 * 0.9 * 6.3 = 3.969$. Note that it has a value that reflects that it is close to a $+10$ reward after considering 3 states, not 300 states, as does value iteration.

Temporal Differences

To understand RL, we need to consider how to average values that arrive to an agent sequentially

Suppose we have some values v1, v2, v3 etc. The goal is to predict the next value, given the previous values. One way to do this is to have a running approximation of the expected value vi.

For example, given a sequence of students grades, and the aim to predict the next grade, we can sum the total and divide by the number of assessments to get an average.

If we get another grade, we need to add that to our data. We can do this by maintaining a running average.

Let A_k be an estimate of the expected value based on the first k data points $v_1, \ldots, v_k$. A reasonable estimate is the sample average:

$$A_k = \frac{v_1 + \cdots + v_k}{k} \ .$$

Thus,

$$\begin{aligned} k * A_k &= v_1 + \cdots + v_{k-1} + v_k \\ &= (k-1)\, A_{k-1} + v_k. \end{aligned}$$

Dividing by k gives

$$A_k = \left(1 - \frac{1}{k}\right) * A_{k-1} + \frac{v_k}{k} \ .$$

Let $\alpha_k = \frac{1}{k}$; then

$$\begin{aligned} A_k &= (1 - \alpha_k) * A_{k-1} + \alpha_k * v_k \\ &= A_{k-1} + \alpha_k * (v_k - A_{k-1}) \, . \end{aligned}$$

The difference, $v_k - A_{k-1}$, is called the **temporal difference error** or **TD error**; it specifies how different the new value, v_k, is from the old prediction, A_{k-1}. The old estimate, A_{k-1}, is updated by α_k times the TD error to get the new estimate, A_k. The qualitative interpretation of the temporal difference formula is that if the new value is higher than the old prediction, increase the predicted value; if the new value is less than the old prediction, decrease the predicted value. The change is proportional to the difference between the new value and the old prediction. Note that this equation is still valid for the first value, $k = 1$, in which case $A_1 = v_1$.

The above analysis assumes that the values have an equal weight, which isn't always true. For example, old values might be less useful than new values.

In RL, the values are estimates of the effect of actions; more recent values are more accurate than earlier values because the agent is learning, and so they should be weighted more.

We can do this with alpha as a constant **0 < alpha <= 1** that does not depend on **k**. Unfortunately, this does not converge to the average value when there is variability in the values in the sequence.

You could reduce alpha more slowly and potentially have the benefits of both approaches: weighting recent observations more and still converging to the average. You can guarantee convergence if:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \text{ and } \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

The first condition is to ensure that random fluctuations and initial conditions get averaged out, and the second guarantees convergence.

One way to give more weight to more recent experiences, but also converge to the average, is to set **αk=(r+1)/(r+k)** for some **r>0**. For the first experience **α1=1**, so it ignores the prior **A0**. If **r=9**, after 11 experiences, **α11=0.5** so it weights that experience as equal to all of its prior experiences. The

parameter r should be set to be appropriate for the domain.

Note that guaranteeing convergence to the average is not compatible with being able to adapt to make better predictions when the underlying process generating the values keeps changing.