Reinforcement Learning (70028)

Markov Processes (Let's Go Markov)

In reinforcement learning, we need a real, tangible method for managing complexity. It's important to distinguish between **Markov Processes** and **Markov Decision Processes**.

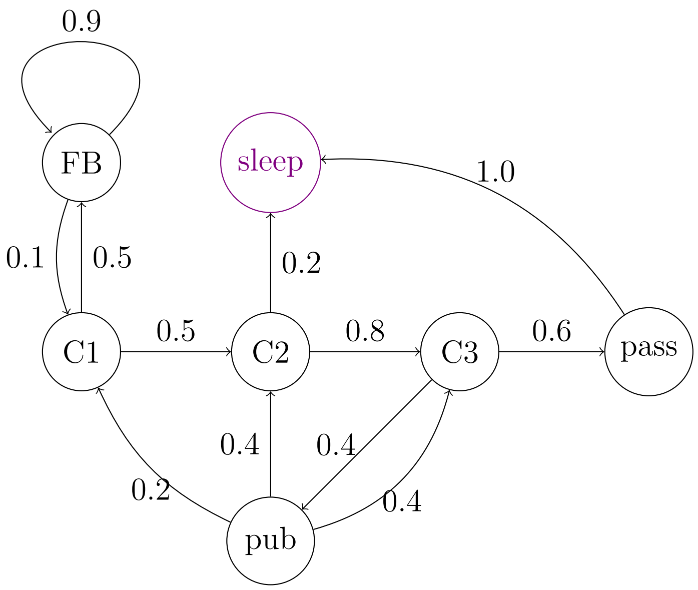
A Markov process is a tuple $(\mathcal{S}, \mathcal{P})$, where $\mathcal{S}$ is a set of states, and $\mathcal{P}_{ss'}$ is a matrix giving us the probability of transitioning from one state to another; note that the probabilities are based only on the current state (one at time t), and not looking beyond that - short memory is important;

$$\mathcal{P}_{ss'} = P[S_{t+1} = s' \mid S_t = s]$$

A Markov process generates a chain of states governed by probabilistic transitions.

A state s_t is Markov iff $P[s_{t+1} \mid s_t] = P[s_{t+1} \mid s_1, \dots, s_t]$; the conditional probability of transitioning to a particular state depends only on the particular state (previous states don't really matter). The equation states that the probabilities are equal, whether it be the current state, or all states preceding - the future is independent of the past given the present. Another way this can be thought of is that the present state, s_t, captures all information in the history of the agent's events, any data of the history is no longer needed once the state is known, or the current state is a sufficient statistic of the future.

An example is as follows, note that the black states are transient states (where it can lead to another state) and the violet states are terminal states. The following is Markovian as the probabilities don't change based on the history (how we got to a state).



The entries must be probabilities (hence between 0 and 1). The matrix defines transition probabilities from all states s to all successor states s'. Since all probabilities have to be accounted for (all rows of the matrix sum to 1) - after leaving s, we need to end up somewhere, which could also mean returning to s;

$$\sum_{s'} \mathcal{P}_{ss'} = 1$$

In the example above, the probability of going to sleep after C2 (class 2) in the morning could be different depending on the time of day (i.e. constantly changing). If $P[s_{t+1} \mid s_t]$ doesn't depend on t, but rather just the origin and destination states, then the Markov chain is stationary or homogenous.

Markov Reward Process

A Markov Reward Process is a Markov chain which emits rewards (the reward hypothesis states that all of what we think of as goals and purposes can be thought of as the maximisation of the expected value of the cumulative sum of a scalar signal known as reward); hence a tuple $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$. This has the following components;

- $\mathcal{S}$ — a set of states
- $\mathcal{P}_{ss'}$ — a state transition probability matrix
- $\mathcal{R}_s = \mathbb{E}[r_{t+1} \mid S_t = s]$

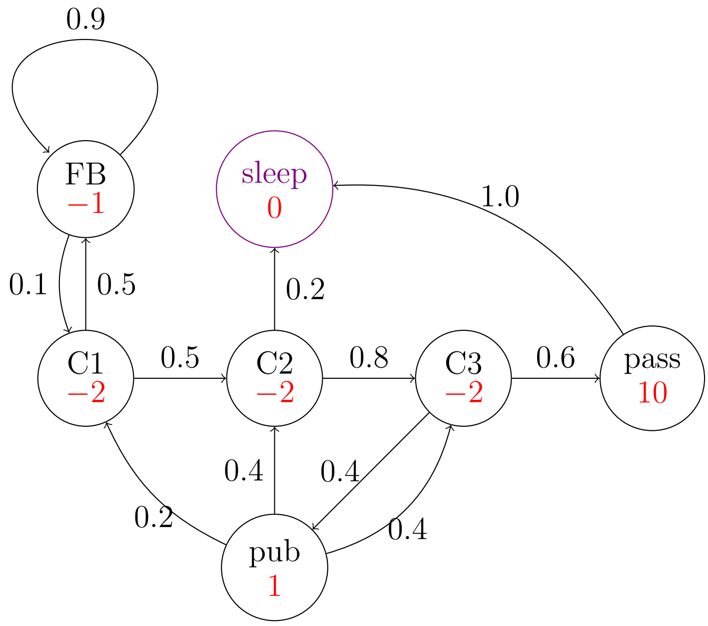
  an expected immediate reward, collected upon departing state s (collection occurs at time $t+1$, we are at state s at time t)
- $\gamma \in [0, 1]$ — discount factor

We can define the return R_t as the total discounted reward from time-step t (note that we use $t+1$ as the first element, since it's collected at $t+1$);

$$R_t = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

The factor γ is how we discount the present value of future rewards; the value of receiving a reward r after $k+1$ time steps is $\gamma^k r$, valuing immediate reward higher than a delayed reward - hence γ closer to 0 leads to short-sighted evaluation, whereas γ closer to 1 leads to far-sighted evaluation (taking future rewards more strongly).

We can add a reward to the previous example as follows (in red);



For example, consider a certain run, where the starting state $S_1 = \text{C1}$ and $\gamma = \frac{1}{2}$, T is the time to reach the terminal state;

$$R_1 = r_2 + \gamma r_3 + \cdots + \gamma^{T-2} r_T$$

Consider the run where the student attends all classes in order and passes; hence C1, C2, C3, pass, sleep;

$$R_1 = -2 + \frac{1}{2} \cdot -2 + \frac{1}{2}^2 \cdot -2 + \frac{1}{2}^3 \cdot 10$$

Most MRPs are discounted with $\gamma < 1$, as it's mathematically convenient by avoiding infinite returns in cyclic / infinite processes (by causing convergence). It also aids in expressing uncertainty in future rewards. A more tangible example is a financial reward, where immediate rewards can be put into a bank and earn interest, similarly, animal decision making shows preference for immediate rewards rather than future rewards.

We can define the state value function $v(s)$ of a MRP as the expected return R starting from state s at time t, thinking of the state as a function parameter;

$$v(s) = \mathbb{E}[R_t \mid S_t = s]$$

The lecture then goes over an example using golf, which is actually quite intuitive.

The Bellman Equation for MRPs is as follows. We can express it in a recurrence relation, as the immediate reward and the discounted return of the successor state.

$$\begin{aligned} v(s) &= \mathbb{E}[R_t \mid S_t = s] \\ &= \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \mid S_t = s] \\ &= \mathbb{E}[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \dots) \mid S_t = s] \\ &= \mathbb{E}[r_{t+1} + \gamma R_{t+1} \mid S_t = s] \\ &= \mathbb{E}[r_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] \end{aligned}$$

The equation can also be written as the sum notation (the previous one was the expectation notation, this has the expectation written out) - there are a total of n of these equations, as there's one for each state;

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in S} \mathcal{P}_{ss'} v(s')$$

As such, this can be written in vector notation as follows, with $\boldsymbol{v}$ being n-dimensional;

$$\boldsymbol{v} = \mathcal{R} + \gamma \mathcal{P} \boldsymbol{v}$$

This can be directly solved as follows, as it's linear and self-consistent;

$$\begin{aligned} \boldsymbol{v} &= \mathcal{R} + \gamma \mathcal{P} \boldsymbol{v} \\ (\mathbb{1} - \gamma \mathcal{P}) \boldsymbol{v} &= \mathcal{R} \\ \boldsymbol{v} &= (\mathbb{1} - \gamma \mathcal{P})^{-1} \mathcal{R} \end{aligned}$$

Since matrix inversion is computationally expensive, being in the order of n^3 for n states, a direct solution is only feasibly for small MRPs. Iterative methods for solving large MRPs include (and all three will be covered);

- dynamic programming
- Monte-Carlo evaluation
- Temporal-Difference learning

Policies

A policy π is a function of the state, formalising the actions to take at a given state. A rigid, deterministic policy can be disadvantageous (e.g. rock, paper, scissors) - exposing the agent to being systematically exploited. A policy can be formally described as the conditional probability distribution to execute an action $a \in \mathcal{A}$ given that one is in state $s \in \mathcal{S}$ at time t;

$$\pi_t(a, s) = P[A_t = a \mid S_t = s]$$

The general form of the policy is probability, or stochastic, hence π is a probability. However, if the policy is deterministic (only a single a is possible for state s), then $\pi(a, s) = 1$, $\pi(a', s) = 0$, $\forall a \neq a'$.

Consider the following example, where there are two actions, a_1, a_2 where we either play the lottery (costing 1), or save (not costing anything). The two states, s_1 and s_2 correspond to winning or losing the lottery.

$$a^* = \underset{a_i}{\text{argmax}} \sum_{j=1}^{2} \mathcal{R}_{s_j}^{a_j} P[s_j \mid a_i]$$

Markov Decision Process

The emphasis decision process, with decision being the key, combines the policies with MRPs. The MDP consists of the following;

- $\mathcal{S}$ — state space
- $\mathcal{A}$ — action space
- $\mathcal{P}_{ss'}^{a}$ — transition probability $p(s_{t+1} \mid s_t, a_t)$

  probability of transitioning to the next state s_{t+1}, given the current state s_t and action a_t taken
- $\gamma \in [0, 1]$ — discount factor
- $\mathcal{R}_{ss'}^{a} = r(s, a, s')$ — immediate reward function

  in temporal notation, $r_{t+1} = r(s_{t+1}, s_t, a_t)$ - reward is collected upon the transition from s_t to s_{t+1}, which occurs at time $t + 1$
- π — policy, can be either stochastic or deterministic

  stochastic is written as the following; $\boldsymbol{a} \sim p_\pi(\boldsymbol{a} \mid \boldsymbol{s}) = \pi(\boldsymbol{a} \mid \boldsymbol{s}) \equiv \pi(a, s)$ - being a probability distribution

  deterministic is written as $\boldsymbol{a} = \pi(\boldsymbol{s})$ (indicator function)

Note that the transition probability and policy both take the action into account, as parameters. We can graphically represent this as follows, with nodes denoting variables, and edges denoting conditional dependencies between these variables;

$$\begin{tikzcd}
& r_{t+1} & \\
s_t \arrow[ur] \arrow[rr] \arrow[dr] & & s_{t+1} \arrow[ul] \\
& a_t \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

This tells us that the action a_t depends on the current state s_t, the next state s_{t+1} depends on both the action and the current state, and the reward r_{t+1} depends on all three.

Value Function

The goodness of a given state is defined with the value function (where R_t is a discounted total return, and r_{t+k+1} are immediate rewards);

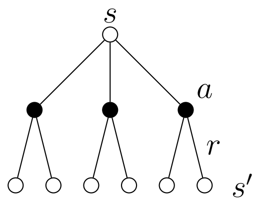
$$V^\pi(s) = \mathbb{E}_\pi[R_t \mid S_t = s] = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, S_t = s\right]$$

This is quite similar to the derivation of the Bellman equation for MRPs, but now including the action (see the policies π). Note that expectation is a linear operator, hence we can justify the final line, also note in the penultimate line we separate out the next reward from the discounted rewards;

$$\begin{aligned}
V^\pi(s) &= \mathbb{E}_\pi[R_t \mid S_t = s] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, S_t = s\right] \\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_t = s\right] \\
&= \mathbb{E}[r_{t+1} \mid S_t = s] + \gamma \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_t = s\right]
\end{aligned}$$

Backup Diagrams

We start at a white node, at a particular state s (since we are conditioning on a particular state $S_t = s$). From this state, we can take several actions, represented by the black nodes, which leads us to following states s', with a reward r. This state value information is transferred back up to s from its successor state s', performing the **update** or **backup** operation at the heart of the reinforcement learning method.



In order to calculate the value of state s, we need to average over all possible traces, which is what's going on behind the scenes in the expectation operator - an average weighted by probabilities. All of which live inside the MDP;

- probability of the chosen action a is given by the policy $P[a \mid s] = \pi(a, s)$
- probability of a transition to s' is given by the transition probability $P[s' \mid s, a] = \mathcal{P}^a_{ss'}$
- instantaneous reward r is given by the reward function $r(s, a, s') = \mathcal{R}^a_{ss'}$
- the value of the next state s', weighted by the probability functions is given recursively by $v(s')$

Writing it out, note that we have the value function of the state s' in violet;

$$
\begin{aligned}
\mathbb{E}[r_{t+1} \mid S_t = s] &= \sum_{a \in \mathcal{A}} P\left[a \mid s\right] \left(\sum_{s' \in \mathcal{S}} P[s' \mid s, a] r(s, a, s') \right) \\
\mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_t = s \right] &= \sum_{a \in \mathcal{A}} P\left[a \mid s\right] \left(\sum_{s' \in \mathcal{S}} P[s' \mid s, a] {\color{violet} \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}} \right) \\
V^\pi(s') &= \mathbb{E}[R_{t+1} \mid S_{t+1} = s'] \\
&= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_{t+1} = s' \right]
\end{aligned}
$$

We can combine all of this as follows;

$$
\begin{aligned}
V^\pi(s) &= \mathbb{E}_\pi[R_t \mid S_t = s] && (1) \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, S_t = s \right] && (2) \\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_t = s \right] && (3) \\
&= \sum_{a \in \mathcal{A}} \pi(a, s) \left(\sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_{t+1} = s' \right] \right) \right) && (4) \\
&= \sum_{a \in \mathcal{A}} \pi(a, s) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} (\mathcal{R}^a_{ss'} + \gamma V^\pi(s')) && (5)
\end{aligned}
$$

Here we are performing the following steps;

(2) write the definition of the return

(3) separate immediate reward

(4) split expectation in two, as it's a linear operator, also write out expectation weighted by probabilities, and using proper notation for policies $\pi(a, s)$, transition probabilities $\mathcal{P}^a_{ss'}$, and reward function $\mathcal{R}^a_{ss'}$

(5) substitute with recursive definition

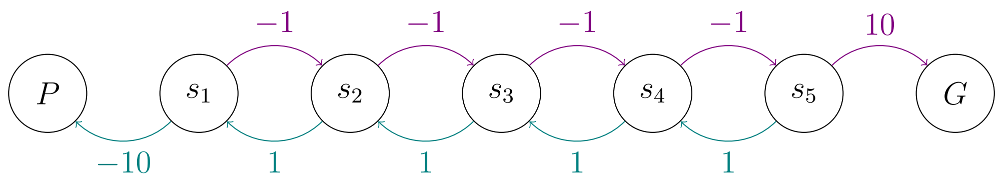
This is a consistency condition imposed on the value function and also has a unique solution. Computing the value function for an arbitrary policy is known as policy evaluation or prediction problem. Now, we need to iterate applications to obtain better estimates - note that the subscripts in $V_1(s), V_2(s), \dots, V_k(s)$ denote iterations, not states; this is guaranteed to converge. This is known as iterative policy evaluation. A stopping condition can be achieved by checking that the **largest** change in the value function, between iterations, is below a certain small threshold. This can be formalised as follows - note that the value function is on a particular policy;

1. input π, the policy to be evaluated
2. initialise $V(s) = 0$ for all $s \in \mathcal{S}^+$
3. repeat the following until $\Delta < \theta$ (where θ is some small positive number)
   (a) $\Delta \leftarrow 0$
   (b) for each $s \in \mathcal{S}$;
      i. $v \leftarrow V(s)$ store old value
      ii. $V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a, s) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}(\mathcal{R}^a_{ss'} + \gamma V^\pi(s'))$ sweep through successors, a full backup
      iii. $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
4. output $V \approx V^\pi$

Note that this replaces values, in place, converging faster than a two-array method, which would have both an old and new array.

Stair Climbing MDP

Consider the following example; for brevity, a violet edge means a Right action, and a teal edge denotes a Left action. P and G are both absorbing / terminal states, assume we start at s_3 with $\gamma = 0.9$, an unbiased policy (such that all actions are equally probable) with $\pi(s, L) = \pi(s, R) = 0.5$, hence randomly selecting actions.



Note that in the first iteration, the only changes are to s_1 and s_5, as they are the only ones with successor states that have different rewards (all other states will cancel out), also note the symmetry stemming from the symmetrical problem.

V	P	s_1	s_2	s_3	$s_4$1	s_5	G
V_0	0	0	0	0	0	0	0
V_1	0	-5.5	0	0	0	5.5	0
V_2	0	-5.5	-2.48	0	2.48	5.5	0
V_2	0	-6.61	-2.48	0	2.48	6.61	0
$\vdots$							
V_∞	0	-6.9	-3.1	0	3.1	6.9	0

Note that we are still equally likely to go to the left, despite being significantly worse; thus knowing the value of the policy can improve the policy.

State-Action Value Function

This takes in two parameters; a state s and an action a, giving us a function that determines the value of taking a certain action at a state.

$$Q^\pi(s,a) = \mathbb{E}[R_t \mid S_t = s, A_t = a] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

The relation between the state value function and this is;

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) Q^\pi(s,a)$$

Bellman Optimality Equations

Previously, we discussed arbitrary policies. However, we can define an ordering on policies (such that some policies are better than others) by saying a policy is better than, or equal to, another policy if its expected return is also greater than or equal to the other policy for all states; $\pi \geq \pi'$ iff $\forall s \in \mathcal{S}\ [V^\pi(s) \geq V^{\pi'}(s)]$. As such, the optimal value function is defined as;

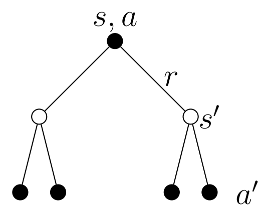
$$V^*(s) = \max_\pi V^\pi(s), \forall s \in \mathcal{S}$$

We call the policy π^* which maximises the value function the optimal policy; there will always be at least one, but multiple can exist. Similarly, there is also an optimal state-action value function;

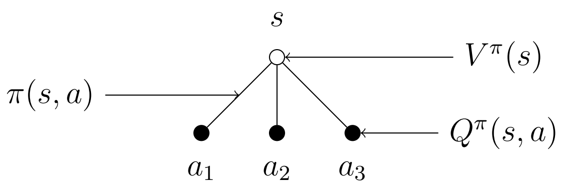
$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} = \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}) \mid S_t = s, A_t = a]$$

The Bellman equations for these are called Bellman Optimality equations.

We have already seen the backup diagram for the value function, and the state-action value function backup diagram is similar (you can think of them as the black nodes and its children);



The white nodes are associated with the value function $V^\pi(s)$, the black nodes with the value-action function $Q^\pi(s,a)$, and the paths between the nodes taken with probability $\pi(s,a)$. The relationship for the function, on just the value function backup diagram can be shown as follows;



If we want the optimal value for a state, only actions that give the highest value should be chosen;

$$\begin{aligned}
V^*(s) &= \max_{a \in \mathcal{A}} \sum_{a \in \mathcal{A}} \pi(s,a) Q^\pi(s,a) && (1)\\
&= \max_a Q^{\pi^*}(s,a) && (2)\\
&= \max_a \mathbb{E}[R_t \mid S_t = s, A_t = a] && (3)
\end{aligned}$$

$$= \max_a \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \tag{4}$$

$$= \max_a \mathbb{E}\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_t = s, A_t = a\right] \tag{5}$$

$$= \max_a \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}) \mid S_t = s, A_t = a] \tag{6}$$

$$= \max_a \sum_{s'} P[s' \mid s, a](r(s, a, s') + \gamma V^*(s')) \tag{7}$$

$$= \max_a \sum_{s'} \mathcal{P}_{ss'}^a(\mathcal{R}_{ss'}^a + \gamma V^*(s')) \tag{8}$$

We perform the following steps;

(3) write down definition of state-action value function, being the expected return conditioned on s and a

(4) perform usual expansion with the maximum on the left

(7) replace with probabilities

There is no reference to any particular policy and must therefore be satisfied by all optimal policies. The optimality equation expresses that the value of a state under an optimal policy is equal to the expected return of the best action from the state. This can be done similarly for Q^*, except the maximum is now on the inside;

$$\begin{aligned}
Q^*(s, a) &= \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \,\middle|\, S_t = s, A_t = a\right] \\
&= \sum_{s'} \mathcal{P}_{ss'}^a \left(\mathcal{R}_{ss'}^a + \gamma \max_{a'} Q^*(s', a')\right)
\end{aligned}$$

Notice that the equation doesn't require π^* at all, which is useful as we don't need to know the optimal policy to solve the optimality equations. For finite MDPs, this equation has a unique solution, independent of the policy. The bellman optimality equation is a set of N non-linear equations, where $N = |\mathcal{S}|$, with N unknowns.

An explicit solution for the optimality equation provides one route for an optimal policy - however we are often going to encounter a high-dimensional problem (large state space). This also assumes the following, which are rarely true;

- we accurately know the dynamics of the environment
- we have the resources to find the solution
- the Markov property

We therefore often settle for approximate solutions.

The BOE convergence theorem states that for an MDP with finite state and action space, the optimality equations have a unique solution and the values produced by iteration converge to the solution of the equations. The proof of this rests on the Banach Fixed Point / Contraction Mapping Theorem.

October 14 - Live Lecture

AI is a question; how do we build systems that solve tasks for which humans need intelligence? On the other hand, machine learning is the answer to the AI question, including methods, algorithms and data structures that learn to solve these tasks from data. Big data means methods, processing, and assessing very large data, broken into data science (how to ask interesting questions about the data using methods from ML) and data engineering (how to build Hadoop systems, fitting data in memory, etc.).

Reinforcement penalises negative behaviour and rewards behaviour that actually works; a goal structure is given. RL solves control problems; choosing the optimal action at the right time.

The general framework for reinforcement learning contains the following;

- agent interacts with the environment to gain knowledge; action is fundamental
- explore and receive rewards; exploration involves trying actions (can also be nothing), receiving rewards, penalty, both long-term and short-term
- actions have an effect on the state of the environment
- choose actions to maximise long-term rewards

Control is sequential decision making, and optimal control which minimises a cost, or maximises a reward. RL involves learning an optimal control of an unknown system.

The session then goes into a refresher on probabilities.

Dynamic Programming

Dynamic programming refers to algorithms that can be used to compute optimal policies, with a perfect model of the environment as a MDP. In particular, DP methods require the distribution of next events, the environment's dynamics (may not be easy to determine in practice). Despite these limitations, it provides a useful conceptual framework for understanding RL algorithms. All of these ML methods can be seen as other ways to obtain the same effect as DP, without assuming a perfect model and with less computation. Note that we only consider finite MDPs (those with finite state and action spaces).

Consider a triangle of numbers, with the goal of getting the maximum sum of a path. A path starts at the top of the triangle and moves to adjacent numbers in the row below. The brute force solution would be simply to perform an exhaustive search and compute the cost for every path. This has a complexity of $\mathcal{O}(2^{n-1})$, where n is the number of rows in the triangle. On the other hand, the dynamic programming approach has a time complexity of $\mathcal{O}(n)$, where n is the number of nodes. The triangle is split into small sub-triangles, computing the maximum path in as single pass working bottom-up. For every cell in the triangle, we find the maximum value of the nodes below it and add it to the node value.

DP exploits the fact that decisions that span several time points can often break down recursively. In the example above, we broke down a large problem into simpler sub-problems that could be solved recursively. The Principal of Optimality states that an optimal policy has the property that whatever the initial state and initial decision are the remaining decision must constitute an optimal policy with regards to the state resulting from the first decision. A problem with an optimal substructure is one that can be solved by breaking into sub-problems and recursively finding optimal solutions. For DP to be applied, the problem must have;

- **optimal substructure** - solution can be obtained by the combination of solutions to sub-problems
- **overlapping sub-problems** - space of sub-problems must be small; a recursive algorithm should solve the same sub-problem rather than generating new ones

  if the optimal solution is found by combining solutions for non-overlapping sub-problems, it is "divide and conquer" instead (such as quick sort)

We compute the value of a policy to find a better policy. Let the value function $V^\pi(s)$ (determined) represent how good it is to follow the current policy π from state s. Let there also be another policy π' such that $\pi'(s) = a'$ - we want to know whether it's better to change to this new policy which differs from π in certain actions. A solution would be to select for s a different action, but otherwise use the old policy. The value is by definition $Q^\pi(s, a')$. If $Q^\pi(s, a') > V^\pi(s)$ (it's better to select a' in state s and follow $\pi(s)$ after that), then the new policy is better overall.

Policy improvement theorem states the following. Let there be any two deterministic policies π, π' such that $\forall s \in \mathcal{S}\ Q^\pi(s, \pi'(s)) \geq V^\pi(s)$. Then π' must be as good (or better than) π;

$$\forall s \in \mathcal{S}\ V^{\pi'}(s) \geq V^\pi(s)$$

If the first inequality is strict in **any** state, then the latter must be strict in **at least one**.

$$\begin{aligned}
\pi(s) &= a && (1)\\
\pi'(s) &= \operatorname*{argmax}_{a \in \mathcal{A}} Q^\pi(s, a) && (2)\\
Q^\pi(s, \pi'(s)) &= \max_{a \in \mathcal{A}} Q^\pi(s, a) && (3)\\
&\geq Q^\pi(s, \pi(s)) && (4)\\
&= V^\pi(s) && (5)
\end{aligned}$$

(1) start with a deterministic policy

(2) we can always be as good or improve by acting greedily (argmax picks the best action, in terms of Q value) - this creates a new greedy policy

(4) improves the value from any state s for at one step

This improves the value function $V^{\pi'}(s) \geq V^\pi(s)$ as follows;

$$\begin{aligned}
V^\pi(s) &\leq Q^\pi(s, \pi'(s))\\
&= \mathbb{E}[R_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s]_{\pi'}\\
&\leq \mathbb{E}[R_{t+1} + \gamma Q^\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s]\\
&\leq \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma Q^\pi(S_{t+2}, \pi'(S_{t+2})) \mid S_t = s]\\
&\leq \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots \mid S_t = s]\\
&= V^{\pi'}(s)
\end{aligned}$$

We only do this while there is still improvement, hence we haven't reached the halting condition;

$$Q^\pi(s, \pi(s')) = \max_{a \in \mathcal{A}} Q^\pi(s, a) = Q^\pi(s, \pi(s)) = V^\pi(s)$$

This means that the Bellman Optimality Equation has been satisfied (hence $\forall s \in \mathcal{S}\ V^\pi(s) = V^{\pi^*}(s) = V^*(s)$ and $\pi = \pi^*$);

$$V^\pi(s) = \max_{a \in \mathcal{A}} Q^\pi(s, a)$$

Policy iteration involves finding a sequence of monotonically improving policies and value functions. This is done by improving a policy π using V^π to yield π'. We can then compute $V^{\pi'}$ which can be improved to π'', and so on. However, recall that policy evaluation is iterative already. It wouldn't make sense to start with all zeroes for this, but rather start with the results of the **previous** iteration. This improves the speed of policy evaluation.

The Principal of Optimality states that a policy $\pi(a|s)$ achieves the optimal value from state s, $V^\pi(s) = V^*(s)$ iff;

- for any state s' that is reachable from s ($\exists a\ p(s', s, a) > 0$)
- π achieves the optimal value starting from state s' ($V^\pi(s') = V^*(s')$)

Any optimal policy consists of two components, an optimal action a^* followed by an optimal policy from the successor state s'. The policy iteration algorithm is as follows;

1. initialise $V(s) \in \mathfrak{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily $\forall s \in \mathcal{S}$

2. policy evaluation - repeat the following until $\Delta < \theta$ (where θ is some small positive number)
   (a) $\Delta \leftarrow 0$
   (b) for each $s \in \mathcal{S}$;
      i. $v \leftarrow V(s)$ — store old value
      ii. $V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a, s) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}(\mathcal{R}^a_{ss'} + \gamma V^\pi(s'))$ — sweep through successors, a full backup
      iii. $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
3. policy improvement;
   (a) `policy-stable` $\leftarrow$ `true`
   (b) for each $s \in \mathcal{S}$
      i. $b \leftarrow \pi(s)$
      ii. $\pi(s) \leftarrow \text{argmax}_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$ replace with greedy action that maximises V function
      iii. if $b \neq \pi(s)$, then the policy hasn't stabilised (so `policy-stable` $\leftarrow$ `false`)
   (c) if `policy-stable`, then terminate, otherwise go to step 2 (evaluate again, note that V isn't reinitialised)

The two stages are policy evaluation and improvement, where evaluation estimates V^π and improvement generates $\pi' \geq \pi$. This can be represented graphically as follows;

$$\begin{tikzcd} \pi \arrow[r, bend left, "V \to V^\pi"] & V \arrow[l, bend left, "\pi \to \text{greedy}(V)"] & \cdots & \pi^* \arrow[r, leftrightarrow, "\text{no more improvement}"'] & V^* \end{tikzcd}$$

One drawback to the algorithm we've stated is that each iteration needs policy evaluation (which in turn is also iteratively computed with multiple sweeps through the state set). We can introduce another stopping condition (as currently it only occurs in the limit) for policy evaluation. This can either be when we have ϵ-convergence of the value function, such that $\forall s\ V_{i-1}(s) - V_i(s) \leq \epsilon$ or after k iterations of iterative policy evaluation. A smaller k value would mean more policy improvements, and fewer policy iterations until convergence. Note that $k = 1$ is a special case, where policy evaluation is stopped after a single sweep (this is value iteration, rather than policy iteration). It turns the BOE rather than the Bellman Equation to an update rule.

Dynamic programming, as we previously knew it, is just deterministic policy MDPs with deterministic actions;

- if we know the solution to subproblems $V^*(s')$
- the solution $V^*(s)$ can be found with a one-step look-ahead;

$$V^*(s) \leftarrow \max a \in \mathcal{A} \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')]$$

The intuition is to start with the final rewards and work backwards (for example, the maximal path sum).

In the value iteration algorithm, there is no explicit policy π;

1. initialise V arbitrarily, for example $\forall s \in \mathcal{S}^+\ V(s) = 0$
2. repeat the following until $\Delta < \theta$ (where θ is some small positive number)

(a) $\Delta \leftarrow 0$

(b) for each $s \in \mathcal{S}$;

i. $v \leftarrow V(s)$

ii. $V(s) \leftarrow \max_a \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}(\mathcal{R}^a_{ss'} + \gamma V^\pi(s'))$

iii. $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

This outputs a deterministic policy, such that;

$$\pi(s) = \underset{a}{\text{argmax}} \sum_{s'} \mathcal{P}^a_{ss'}(\mathcal{R}^a_{ss'} + \gamma V^\pi(s'))$$

The lecture then goes over a concrete example, which is the shortest path problem in a grid world (only 4 actions).

Dynamic programming performs full backups, which is somewhat like a breadth-first search. DP methods can either be synchronous (all states are backed up in parallel, requiring two copies of the value function) or asynchronous (target only states individually, in place, only one value function). Values of all the states need to be updated / all states selected to guarantee convergence (can't ignore states). We can try to order the updates to allow value updates to propagate from state to state in an efficient way (some states may not need updates as often). This also allows mixing with real-time interaction (MDP running at the same time as agent is making decisions, for example focusing updates on states that are most relevant to the agent).

DP is effective for medium-sized problems (with millions of states), for large problems, it can suffer the curse of dimensionality. DP updates values based on other value estimates (bootstrapping) based on the optimal sub-problem structure.

Sample backups (compared to full-width backups), instead of using the transition dynamics $\mathcal{P}$ and the reward function $\mathcal{R}$, consider a single sample of what might happen. This consists of state, actions taken, rewards received, and successor state. This may be generated by real experience, or a simulation. The advantage of this is that it's model-free (no knowledge of the MDP is required in advanced, in particular the transition dynamics). It helps to break the curse of dimensionality as we don't have full-backups, and the cost of backups are constant (independent of the state space $N = ||\mathcal{S}||$).

Monte Carlo Learning

This is a model-free learning method. These do not assume complete knowledge of the environment but only require experience, sample sequences of states, actions and rewards from actual or simulated interaction with an environment. A model is required for simulation, but it doesn't need to generate the full transition dynamics, just sample transitions.

MC methods are ways of solving the RL problem based on averaging sample returns. To ensure this, we only apply this for **episodic** tasks (assume experience is divided into episodes, episodes which eventually terminate no matter what actions are selected) - returns are only given at the end of an episode. Only on the completion of an episode are value estimates and policies changed. Since it only learns from complete episodes, there is typically no bootstrapping. MC methods are therefore incremental on an episode-by-episode sense, but not on a step-by-step online sense. MC is used more broadly to refer to any estimation method which uses a significant random component.

We want to learn the value function for a given policy π. The value of a state is the expected cumulative future discounted reward (return), starting at that state. To estimate the value of a state, simply average the returns after visits to that state - with more observations of returns, the average should converge to the expected value. This idea is true for all Monte Carlo methods.

Our goal is to learn V^π from traces $\tau \equiv s_1, a_1, r_2, \dots, s_k$ of episodes of length T that we experience under policy π. The return, as before, is the total discounted reward $R_t = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{T-1} r_T$.

The value function is the expected return $V^\pi(s) = \mathbb{E}[R_t \mid S_t = s]$. MC policy evaluation instead uses the empirical mean returns rather than the expected return (sum of returns divided by the number samples). We no longer perform a full-width backup, but rather sample trace evaluations. The steps for Monte-Carlo policy evaluation are as follows;

1. $\forall s \in \mathcal{S}\ \hat{V}(s) \leftarrow$ arbitrary value
2. $\forall s \in \mathcal{S}$ `returns`$(S) \leftarrow []$ initialise with an empty list (one for each state)
3. iterate until convergence;
   (a) get trace τ using π
   (b) for all s appearing in τ
      i. $R \leftarrow$ return from first appearance of s in τ
      ii. append R to `returns(s)`
      iii. $\hat{V}(s) \leftarrow$ `average`(`returns`(s))

This is known as the **first visit** Monte-Carlo algorithm, where we only use the first occurrence of a state. A variation is **every visit** MC, where we append the return of the episode (from that point) on **every** occurrence of the state in the episode.

In online Monte-Carlo, we perform updates at the end of each episode. With Batch MC, the update is done after every n episodes (which is a parameter), and finally with vanilla MC, only one update is performed right at the very end of all the episodes.

It's important for performance reasons to compute the mean online. The mean can be computed for each new datapoint as follows;

$$\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\
&= \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right) \\
&= \frac{1}{k}(x_k + (k-1)\mu_{k-1}) \\
&= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})
\end{aligned}$$

This follows the form of an incremental estimation computation that has a small ≤ 1 weighting factor, an old estimated value, and new data - pulling the estimate towards the new data;

$$\Delta = \mu_k - \mu_{k-1} = \frac{1}{k}(x_k - \mu_{k-1})$$

Using this, we can now update the value functions without storing sample traces.

1. update $V(s)$ incrementally after step $s_t, a_t, r_{t+1}, s_{t+1}$
2. for each state s_t with a return of R_t (up to this point), and let $N(s_t)$ represent the visit counter to this state;

$$\begin{aligned}
&N(s_t) \leftarrow N(s_t) + 1 \\
&V(s_t) \leftarrow V(s_t) + \frac{1}{N(s_t)}(R_t - V(s_t))
\end{aligned}$$

   Note if the world is non-stationary, a running mean can be tracked by gradually forgetting old episodes (α is the rate of which old episodes are forgotten (learning rate)). We don't want to overlearn something that may not be relevant.

$$V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_t))$$