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**.

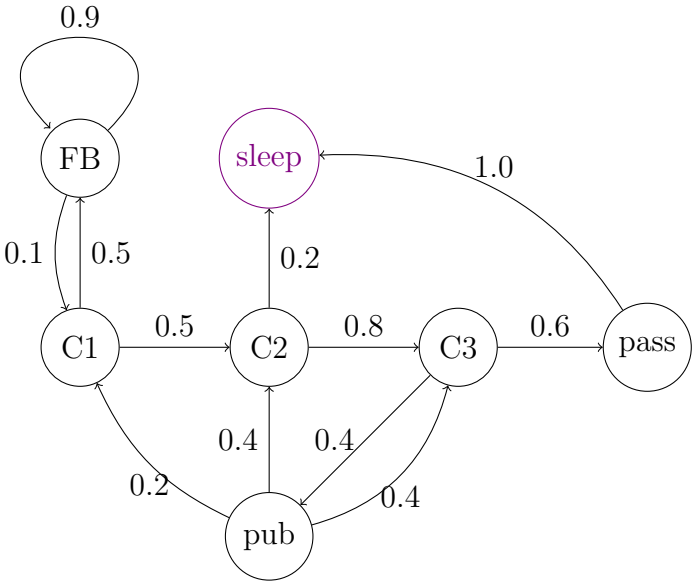
A Markov process is a tuple $(\mathcal{S}, \mathcal{P})$, where $\mathcal{S}$ is a set of states, and $\mathcal{P}_{ss'}$ is a matrix giving us the probability of transitioning from one state to another; note that the probabilities are based only on the current state (one at time t), and not looking 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
- $\mathcal{R}_s = \mathbb{E}[r_{t+1} \mid S_t = s]$

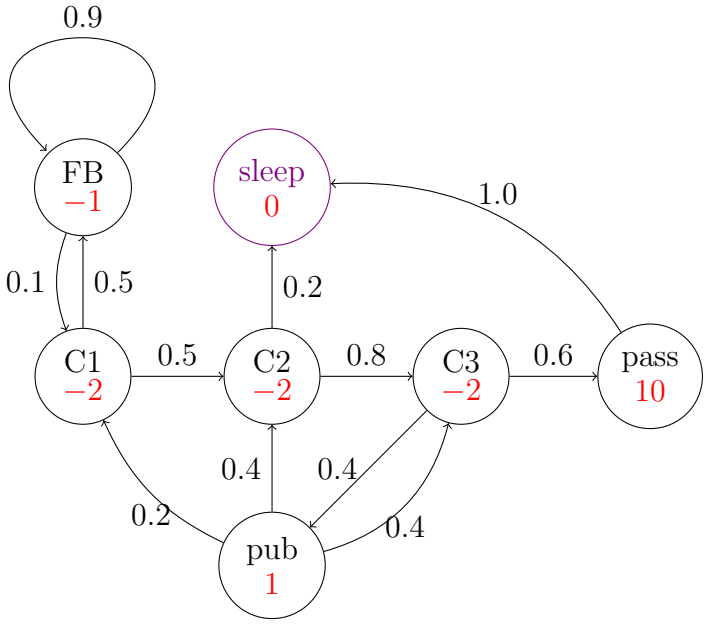
  an expected immediate reward, collected upon departing state s (collection occurs at time $t+1$, we are at state s at time t)
- $\gamma \in [0, 1]$ — discount factor

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.

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

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
- $\mathcal{P}^a_{ss'}$ transition probability $p(s_{t+1} \mid s_t, a_t)$

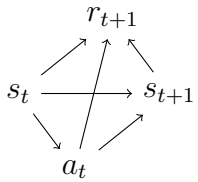
  probability of transitioning to the next state s_{t+1}, given the current state s_t and action a_t taken
- $\gamma \in [0, 1]$ discount factor
- $\mathcal{R}^a_{ss'} = r(s, a, s')$ immediate reward function

  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;



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);

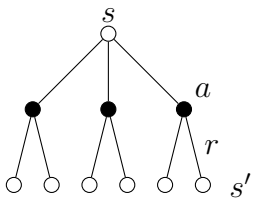
$$V^\pi(s) = \mathbb{E}_\pi[R_t \mid S_t = s] = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, S_t = s\right]$$

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;

$$\mathbb{E}[r_{t+1} \mid S_t = s] = \sum_{a \in \mathcal{A}} P\left[a \mid s\right] \left(\sum_{s' \in \mathcal{S}} P[s' \mid s, a] r(s, a, s') \right)$$

$$\mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_t = s \right] = \sum_{a \in \mathcal{A}} P\left[a \mid s\right] \left(\sum_{s' \in \mathcal{S}} P[s' \mid s, a] \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \right)$$

$$\begin{aligned} V^\pi(s') &= \mathbb{E}[R_{t+1} \mid S_{t+1} = s'] \\ &= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_{t+1} = s' \right] \end{aligned}$$

We can combine all of this as follows;

$$V^\pi(s) = \mathbb{E}_\pi[R_t \mid S_t = s] \tag{1}$$

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

$$= \mathbb{E}_\pi\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_t = s \right] \tag{3}$$

$$= \sum_{a \in \mathcal{A}} \pi(a, s) \left(\sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, S_{t+1} = s' \right] \right) \right) \tag{4}$$

$$= \sum_{a \in \mathcal{A}} \pi(a, s) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} (\mathcal{R}^a_{ss'} + \gamma V^\pi(s')) \tag{5}$$

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

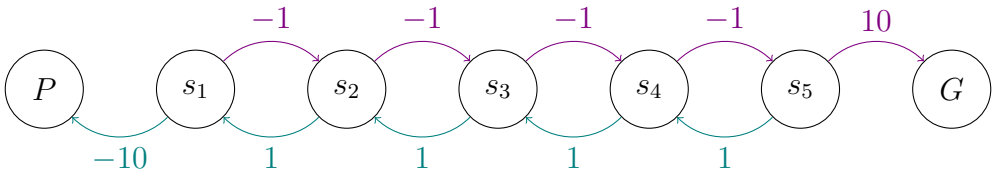
This is a consistency condition imposed on the value function and also has a unique solution. Computing the value function for an arbitrary policy is known as policy evaluation or prediction problem. Now, we need to iterate applications to obtain better estimates - note that the subscripts in $V_1(s), V_2(s), \ldots, V_k(s)$ 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
⋮							
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;

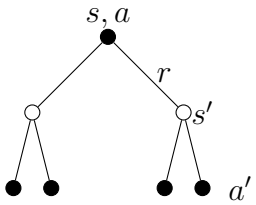
$$V^*(s) = \max_\pi V^\pi(s), \forall s \in \mathcal{S}$$

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;

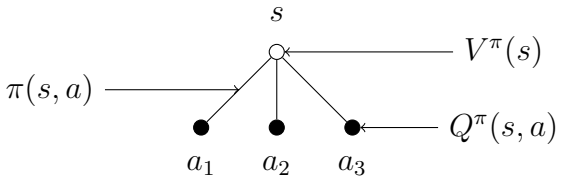
$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} = \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}) \mid S_t = s, A_t = a]$$

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;

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{a \in \mathcal{A}} \pi(s,a) Q^\pi(s,a) \tag{1}$$

$$= \max_a Q^{\pi^*}(s,a) \tag{2}$$

$$= \max_a \mathbb{E}[R_t \mid S_t = s, A_t = a] \tag{3}$$

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

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

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

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

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

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;

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

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;

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

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;

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;

$$\pi(s) = \operatorname*{argmax}_a \sum_{s'} \mathcal{P}^a_{ss'}(\mathcal{R}^a_{ss'} + \gamma V^\pi(s'))$$

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

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. $\forall s \in \mathcal{S}\ \texttt{returns}(S) \leftarrow []$ 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 $\texttt{returns}(s)$
      iii. $\hat{V}(s) \leftarrow \texttt{average}(\texttt{returns}(s))$

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.

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

Temporal Difference Learning

Dynamic programming and Monte-Carlo techniques have desirable properties that conflict with each other. The former bootstraps (updates value estimates with other estimates) and the latter samples from the environment. In DP, visiting all rewards and successor state pairs in our backup diagram. TD performs bootstrapping from 1-step samples, therefore it sits between the DP methods and the MC methods by combining sampling as well as bootstrapping. Temporal Difference methods have the following properties;

- learn directly from episodes of experience (can also work for non-episodic)
- model-free; no knowledge of transitions or rewards required
- uses bootstrapping to learn from incomplete episodes
- updates a guess towards a guess

Recall that the MC update rule was $V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_t))$ (using only actual measurements to update the return, no other estimates). However, TD methods update it towards the estimated return (rather than the measured return) - a combination of measurements r_{t+1} with the current estimate of the value of the successor state $V(s_{t+1})$;

$$V(s_t) \leftarrow V(s_t) + \alpha(\underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{estimated return}} - V(s_t))$$

Note that the estimated return is also known as the temporal difference target and $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is known as the temporal difference error. The algorithm for performing TD value estimation is as follows;

1. initialise $\hat{V}(s)$ to be an arbitrary value for all $s \in \mathcal{S}$
2. repeat the following for each episode;
   (a) initialise s
   (b) repeat the following for each step of the episode, until s is absorbing;
      i. action a chosen from π at s
      ii. take action a, observe the reward r and next state s'
      iii. $\delta \leftarrow r + \gamma \hat{V}(s') - \hat{V}(s)$
      iv. $\hat{V}(s) \leftarrow \hat{V}(s) + \alpha\delta$
      v. $s \leftarrow s'$

TD is able to learn online after every step (learn before knowing the final outcome), whereas MC must wait until the episode is complete before the return is known. It can also therefore learn without the final outcome. MC can only learn from complete sequences whereas TD works in non-terminating environments (even when there is no episodic structure).

The sample return $R = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-1} r_T$ is an unbiased estimate of $V^\pi(s_t)$. The true TD target $r_{t+1} + \gamma V^\pi(s_{t+1})$ is also an unbiased estimate, However, the estimated TD target $r_{t+1} + \gamma V(s_{t+1})$ (bootstrapped value) is a biased estimate as the $V(s_{t+1})$ was set arbitrarily and has no bearing on the true value - this bias decays over time as real values are used in the update process. The TD target has much lower variance than the return (the sample return is based on many random actions, transitions and rewards, whereas the TD target is based on only one action, transition, and reward). As such, TD has low accuracy and high precision, whereas MC has high accuracy but low precision.

In conclusion, MC has high variance and no bias, has good convergence properties even with function approximation, and is not very sensitive to the initial value. On the other hand, TD exploits the Markov property (therefore is more efficient in these environments), and has low variance with some bias, however has no convergence guarantee with function approximation and is more sensitive to the initial value.

MC and TD both sample, whereas DP does not. DP and TD both bootstrap, whereas MC does not.

Model-Free Control

The basics of Model-Free Learning involved estimating the value function for an unknown MDP. Model-Free Control involves optimising the value function. This is used in two main scenarios; when the model is known but too big to use (other than with samples), or when the model is unknown but can be sampled (experience).

Generalised policy iteration (GPI) repeatedly alters the value function to approach the value function for the current policy, which is repeatedly improved based on the value function. A greedy policy (using the current value function) is used in policy improvement. We are limited to an action-value function (hence no model is required to construct the policy).

Our use of V or Q implies our knowledge of the model;

- $\pi' = \text{argmax}_{a \in \mathcal{A}} \mathcal{R}^a_{ss'} + \mathcal{P}^a_{ss'} V(s')$ requires model-knowledge of MDP
- $\pi' = \text{argmax}_{a \in \mathcal{A}} Q(s, a)$ model-free

Instead of learning the values of states and transitions between states, we now consider the values of state-action pairs and the transitions between them.

For now, assume we observe infinitely many episodes and start in many different initial states (performing exploring starts / random states) - this may not be feasible from experience, only from simulations.

The policy improvement theorem encountered in DP applies here too, however we have an average empirical return rather than a mean return. Two assumptions must be made for the proof to hold; evaluation can be done with an infinite number of episodes, which have exploring starts to ensure we fully experience the world. As such, MC methods can be used to find optimal policies with no knowledge of the environment other than sample episodes. However, both assumptions should be removed for practical algorithms. The proof for MC policy improvement is as follows (π_k and π_{k+1}), as the following applies for all $s \in \mathcal{S}$;

$$\begin{aligned} Q^{\pi_k}(s, \pi_{k+1}(s)) &= Q^{\pi_k}(s, \operatorname*{argmax}_a Q^{\pi_k}(s, a)) \\ &= \max_a Q^{\pi_k}(s, a) \\ &\geq Q^{\pi_k}(s, \pi_k(s)) \\ &= V^{\pi_k}(s) \end{aligned}$$

Recall that exploring starts is unlikely in experience - the general way to ensure all actions are selected infinitely often is for the agent to continuously select them in one of two methods;

- **on-policy** attempt to evaluate / improve the policy used

  learning on the job; learn about π from experience sampled from π

  For soft policies, $\forall s \in \mathcal{S}\ \forall a \in \mathcal{A}\ \pi(a, s) > 0$ - we have a finite probability to explore all actions. ϵ-greedy policies are a form of soft policy, where all actions other than the greedy action have an equal share of ϵ (ϵ divided by the number of actions), and the greedy policy has a high probability;

$$\begin{aligned} \epsilon &\in [0, 1] \\ a^* &= \operatorname*{argmax}_a Q(s, a) \\ \pi(s, a) &= \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|} & \text{if } a = a^* \\ \frac{\epsilon}{|A(s)|} & \text{if } a \neq a^* \end{cases} \end{aligned}$$

  The ϵ-greedy first-visit MC control algorithm is as follows;

  1. initialise for all $s \in \mathcal{S}, a \in \mathcal{A}$

- $Q(s,a)$ arbitrarily
- $\texttt{returns}(s,a)$ to an empty list
- $\pi(a|s)$ arbitrary ϵ-soft policy

2. infinitely repeat the following;
   (a) generate an episode using π
   (b) for each pair s, a in the episode;
      i. G is the return following the first occurrence of s, a
      ii. append G to $\texttt{returns}(s,a)$
      iii. $Q(s,a) \leftarrow \texttt{average}(\texttt{returns}(s,a))$
   (c) for each s in the episode;
      i. $a^* = \text{argmax}_a Q(s,a)$
      ii. for all $a \in \mathcal{A}(s)$

$$\pi(a|s) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|} & \text{if } a = a^* \\ \frac{\epsilon}{|A(s)|} & \text{if } a \neq a^* \end{cases}$$

GLIE tells us when a schedule for adapting the exploration parameter is sufficient to ensure convergence. For example, the ϵ-greedy operation is GLIE if ϵ reduces to zero with $\epsilon_k = \frac{1}{k}$. The definition of GLIE (greedy in the limit with infinite exploration) has the two properties;

- all state-action pairs are explored infinitely many times $\lim_{k\to\infty} N_k(s,a) = \infty$
- the policy converges on a greedy policy $\lim_{k\to\infty} \pi_k(s,a) = (a == \text{argmax}_{a' \in \mathcal{A}} Q_k(s,a'))$
  note the == is 1 when equal and 0 when not equal

- **off-policy** evaluate / improve a policy different from the one used to generate data
  looking over someone's shoulder; learn about π from experience sampled from π' (another policy)

Monte-Carlo methods can also be batched (estimates based on a large collection of transitions);

1. initialise $\hat{Q}(s,a)$ to an arbitrary value for all $s \in \mathcal{S}, a \in \mathcal{A}$
2. initialise π to $\epsilon - \texttt{greedy}(\hat{Q})$
3. repeat for each batch;
   (a) initialise $\texttt{returns}(s,a)$ to an empty list for all $s \in \mathcal{S}$
   (b) iterate $i \in [1,n]$;
      i. get trace τ using π
      ii. for all (s,a) in τ;
         A. set R as the return from the first appearance of (s,a) in τ
         B. append R to $\texttt{returns}(s,a)$
   (c) for all $s \in \mathcal{S}, a \in \mathcal{A}$;
      i. $\hat{Q}(s,a) \leftarrow \texttt{average}(\texttt{returns}(s,a))$
   (d) $\pi \leftarrow \epsilon - \texttt{greedy}(\hat{Q})$

Similarly, it can also be iterative; the memory is changed after each transition;

1. initialise $\hat{Q}(s,a)$ to an arbitrary value for all $s \in \mathcal{S}, a \in \mathcal{A}$
2. initialise π to $\epsilon - \texttt{greedy}(\hat{Q})$
3. iterate $i \in [1,n]$;
   (a) get trace τ using π

(b) for all (s, a) in τ;

i. set R as the return from the first appearance of (s, a) in τ
ii. $\hat{Q}(s, a) \leftarrow \hat{Q}(s_t, a_t) + \alpha(R - \hat{Q}(s_t, a_t))$

(c) $\pi \leftarrow \epsilon - \texttt{greedy}(\hat{Q})$

To summarise MC control methods; we followed a generalised policy iteration scheme; a simple average of many returns that start in a state is used rather than a model computing the value of each state. Since the state's value is the expected return, the average approximates this value. Maintaining exploration is an issue in MC control methods, as we need to obtain returns for alternative actions (not just selecting the current best, according to the estimate) - they may be learned to be actually better. One approach is to assume episodes begin with randomly selected state-action pairs that cover all possibilities; this could be arranged in simulated episodes, but is unlikely from real experience. In both on-policy and off-policy methods the agent explores, but the latter learns a deterministic optimal policy that might be unrelated to the policy followed and the former tries to find the best policy that still explores.

As we've seen, there are a number of advantages for temporal-difference over Monte-Carlo, including a lower variance, online learning, as well as the ability to handle incomplete sequences. Intuitively, we could use TD over MC in the control loop, by applying TD to $Q(s, a)$, using ϵ-greedy policy improvement with updates every time-step. For on-policy TD control, the same pattern for generalised policy iteration is followed, albeit with TD methods in the evaluation / prediction parts. To learn an action-value function, rather than a state-value function, we must estimate $Q^\pi(s, a)$ for a policy π and for all states s and actions a. The theorems that ensure convergence for state values also apply to state-action pairs.

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$$

The main function for updating the Q-value depends on the following (**SARSA**);

- s_1 the current state of the agent
- a_1 the action the agent chooses
- r_2 the reward the agent gets for choosing the action
- s_2 the state the agent will be in after taking the action
- a_2 the next action the agent will choose in the new state

As such, the on-policy learning TD control is as follows;

1. initialise $Q(s, a)$ arbitrarily $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}$ and $Q(\texttt{terminal}, \cdot) = 0$
2. repeat for each episode;

(a) initialise S
(b) choose A from S using policy derived from Q (such as with ϵ-greedy)
(c) repeat for each step of the episode, until S is terminal;

i. take action A, observe R and S'
ii. choose A' from S' using policy derived from Q (same as before)
iii. $Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$
iv. $S \leftarrow S'$ and $A \leftarrow A'$

SARSA converges ($Q(S, A) \rightarrow Q^\infty(S, A)$) to the optimal action-value function when the following conditions are met - hence we need to control both the exploration ϵ and the learning rate α over time;

- GLIE sequence of policies $\pi^k(a, s)$

- Robbins-Munro sequence of step-sizes α_t

$$\sum_{t=1}^{\infty} \alpha_t = \infty$$
$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

To deal with sparse rewards, SARSA-Lambda is an approach that uses eligibility traces to propagate sparse rewards through a trace. Another approach, Hindsight Experience Replay converts unsuccessful episodes into artificial rewarded episodes (failed to do X but achieved Y).

Off-Policy Methods

Suppose we want to estimate Q^π or V^π, but only have episodes generated from another policy π'. Let π be the target policy and π' be the behaviour policy. For a behaviour policy to work, allowing us to use π' episodes to estimate π, we require every action taken under π to be occasionally taken under π' - the assumption of coverage; (we require $\pi(a, s) > 0 \Rightarrow \pi'(a, s) > 0$). Coverage requires π' to be stochastic when it is not identical to π.

Q-learning is an off-policy TD control algorithm. The next action is chosen with the behaviour policy $a_{t+1} \sim \pi'(\cdot|s_t)$. We consider the alternative successor action $a' \sim \pi(\cdot|s_t)$. We update $Q(s_t, a_t)$ in the direction of the alternative / better action;

$$\max_a Q(s_{t+1}, a)$$

Both the behaviour and target policies improve;

- target policy π is greedy with respect to $Q(s, a)$ $\qquad\qquad$ $\pi(s_{t+1}) = \operatorname{argmax} Q(s_{t+1}, a')$
- behaviour policy is (for example) ϵ-greedy with respect to $Q(s, a)$

The Q-learning target is simplified as;

$$\begin{aligned} r_{t+1} + \gamma Q(s_{t+1}, a') &= r_{t+1} + \gamma Q(s_{t+1}, \operatorname{argmax} Q(s_{t+1}, a')) \\ &= r_{t+1} + \max_{a'} \gamma Q(s_{t+1}, a')) \end{aligned}$$

The Q-learning algorithm is similar to the on-policy learning;

1. initialise $Q(s, a)$ arbitrarily $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$ and $Q(\texttt{terminal}, \cdot) = 0$
2. repeat for each episode;
   (a) initialise S
   (b) choose A from S using policy derived from Q (such as with ϵ-greedy)
   (c) repeat for each step of the episode, until S is terminal;
      i. take action A, observe R and S'
      ii. choose A' from S' using policy derived from Q (same as before)
      iii. $Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', a') - Q(S, A))$
      iv. $S \leftarrow S'$

Note that the target policy is implicit in the greedy term $\max_a Q(s', a)$ and the behaviour policy is the ϵ-greedy version of the target policy. Both policies are updated on each step.

Instead of experience replay, we can train multiple agents in parallel allowing for weights to be updated asynchronously, which results in better exploration. Experience replay is essentially off-policy training.

Summary

Note that $x \leftarrow^{\alpha} y$ is defined as $x \leftarrow x + \alpha(y - x)$.

Bellman	full backup (DP)	sample backup (TD)
BEE for $v_\pi(s)$	iterative policy evaluation $V(s) \leftarrow \mathbb{E}[R + \gamma V(S') \mid s]$)	TD learning $V(S) \leftarrow^{\alpha} R + \gamma V(S')$
BEE for $q_\pi(s, a)$	Q-policy iteration $Q(s, a) \leftarrow \mathbb{E}[R + \gamma Q(S', A') \mid s, a]$	SARSA $Q(S, A) \leftarrow^{\alpha} R + \gamma Q(S', A')$
BOE for $q_*(s, a)$	Q-value iteration $Q(s, a) \leftarrow \mathbb{E}[R + \gamma \max_{a' \in \mathcal{A}} Q(S', a') \mid s, a]$	Q-learning $Q(S, A) \leftarrow^{\alpha} R + \gamma \max_{a' \in \mathcal{A}} Q(S', a')$

MC methods learn value functions and optimal policies (from experience) using sample episodes - giving advantages over DP methods. GPI averages many returns from a state, rather than using a model; since a state's value is the expected return, the average is a good approximation.

Function Approximation

The prediction methods that have been covered can be described as updates to an estimated value function. The state-action pairs that we've encountered so far have been small enough to be represented as arrays or tables (tabular solutions). Approximate methods on the other hand can only find approximate solutions (whereas the former could find the optimal solutions), but can be applied to much larger problems.

In the real world, we have continuous spaces, hence one $Q(S, A)$ may be the same for multiple states (when we have a grid world). This prevents us from accurately representing the real world. Another limitation of tabular Q-learning is that all state-action pairs must be visited once to learn about its reward (and therefore Q-value); related states still need to be updated independently. This makes exploration inefficient for large state-action spaces.

The solution to this is to turn $Q(S, A)$ into a continuous function that approximates the true underlying table. If this is parameterised to $Q_\theta(S, A)$, for example as a neural network, then the memory is constant. The knowledge is also generalised, which solves the second limitation; the need to visit every state-action pair, neighbouring points are also updated.

Deep Learning 101

This entire lecture is based on `Week 4 - Deep Learning 101.ipynb`

Deep learning is a class of ML algorithms that use multiple layers to extract higher-level features from raw input. The lecture starts with some brief history.

Let $\boldsymbol{x}$ denote a vector of inputs, $\boldsymbol{w}$ be a vector of weights, b be the scalar bias / offset, and σ be the activation function (in this case, the Heaviside step function);

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b) \qquad \text{basic perceptron}$$

$$\sigma(v) = \begin{cases} 0 & v < 0 \\ 1 & v \geq 0 \end{cases}$$

The lecture then continues with some examples of how a perceptron can be used to represent binary logical functions. We want to automate the fitting of a perceptron, as guessing weights and biases is tedious. Let $\hat{y}$ be the output of the model and y be the true value (target). We can define the zero-one loss function as follows. Note that the zero-one loss is a piecewise constant function of the parameters; the derivatives are both zero meaning that changing either by a small amount doesn't change the loss;

$$\hat{y} = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$$

$$\ell_{0-1}(\hat{y}, y) = \begin{cases} 0 & \hat{y} \neq y \\ 1 & \hat{y} = y \end{cases}$$
$$\frac{\partial \ell_{0-1}}{\partial w_j} = 0$$
$$\frac{\partial \ell_{0-1}}{\partial b} = 0$$

Instead, we can define the surrogate loss function (use a loss function that is easier to optimise in place of the loss function we care about) as the squared error;

$$\ell_{\text{SE}}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$
$$\frac{\partial \ell_{\text{SE}}}{\partial \boldsymbol{w}} = (\hat{y} - y)\boldsymbol{x}$$
$$\frac{\partial \ell_{\text{SE}}}{\partial b} = \hat{y} - y$$

After this, the lecture goes over implementing this (fitting) in Python.

Note that `XOR` cannot be done with a single layer. However, `XOR` can be represented by a combination of layers (two dense layers separated by activation functions).

Linear Function Approximation

Previously, we had tables representing the value function $V(s)$ and action-value function $Q(s, a)$. However, large MDPs have many states (curse of dimensionality; the number of samples required to estimate an arbitrary function grows exponentially with respect to the number of input variables (dimensionality) of the function). There are too many states to store in memory, leading to the agent taking a long time to learn the value of each state individually. In reinforcement learning, even sparse matrices tend to become dense as values are eventually filled in.

A solution is to estimate the value function with function approximation. It generalises from seen states to unseen states. The parameter $\boldsymbol{w}$ is updated with MC or TD learning;

$$V^\pi(s) \approx \hat{V}(s, \boldsymbol{w})$$
$$Q^\pi(s, a) \approx \hat{Q}(s, a, \boldsymbol{w})$$

For function approximators, we limit ourselves to differentiable functions, thus simplifying learning. The training method needs to be suitable for non-stationary and non-iid data.

The goal is to find the parameter vector which minimise the MSE between the approximate value function and the true value function;

$$J(\boldsymbol{w}) = \mathbb{E}[(V^\pi(s) - \hat{V}(s, \boldsymbol{w}))^2]$$

The gradient is as follows (note the $\frac{1}{2}$ is used to cancel out the derivative);

$$\Delta \boldsymbol{w} = -\frac{1}{2}\alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w})$$
$$= \alpha \mathbb{E}[(V^\pi(s) - \hat{V}(s, \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{V}(s, \boldsymbol{w})]$$

SGD samples the gradient and the average update is equivalent to the full gradient update.

State can be represented as a feature vector $\boldsymbol{x}(s) = [x_1(s), \ldots, x_n(s)]^\top$. This may differ from the precise state definition, as the real world may not have a strong definition of state. Our previous derivations can be considered a special case of function approximation, where the feature vector is the state vector.

If some state is inside a circle, then the corresponding feature has the value 1, otherwise the feature is 0 (binary feature). Given a state, the present binary features indicate where the state lines (within the circles), and coarsely code its location. Coarse coding is the process of representing a state with features that overlap (don't necessarily need to be binary or circles). Corresponding to each circle is a component of $\boldsymbol{w}$ that is affected by learning - if we train at one point / state x, all circles that intersect it will be affected. As such, the approximate value function will be affected at all points within the union of the circles. If the circles are small, the generalisation will be over a small distance. See the slides for a visual example of a square wave function. With narrow features, only the close neighbours of each trained point were changed (leading to a 'bumpy' function). Features are crafted, not learnt.

Tile coding is a form of coarse coding, well suited for computers and efficient on-line learning. The receptive fields of the features are grouped into partitions of the input space, with each partition being called a tiling, and each element of the partition being called a tile. The number of features is strictly controlled and independent of the input state.

Tile coding uses exclusively binary valued features, the weighted sum which makes up the approximated value function is simple to compute. The computation of the indices (from a point in the space x, y) is easy to perform. When multiple tilings are used, each is offset by a different amount. The width of the tiles should be chosen to match the width of generalisation that is expected to be appropriate. The number of tilings determines the resolution of the approximation. Hashing can also be used to collapse a large tiling into a smaller set of tiles.

Radial basis functions are the natural generalisation of coarse coding to continuous-valued features (values lie in the interval $[0, 1]$). This can reflect the various degrees in which the feature is present. An RBF network is a linear function approximation that uses RBFs for its features. Each $\phi_s(i)$ is the i^{th} basis function that maps a continuous state space variable s, where σ_i is the width and c_i is the centre;

$$\phi_s(i) = e^{-\frac{|s-c_i|^2}{2\sigma_i^2}}$$

The main advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. Some learning methods change the centres and widths of each RBF.

Hybrid images combines the low spatial frequencies of one picture with the high spatial frequencies of another, producing an image that has an interpretation which changes with viewing distance.

Inductive / learning bias of a learning algorithm is the set of assumptions that the learner uses to predict outputs of given unencountered inputs. An appropriately chosen algorithm can reduce the inductive bias of the person designing the algorithm. Some function approximations can identify complex patterns that can't be seen by a human, or identify weak patterns that are undetectable by humans.

The value function is represented with a linear combination of features;

$$\hat{v}(s, \boldsymbol{w}) = \boldsymbol{x}(s)^\top \boldsymbol{w} = \sum_{j=1}^{n} x_j(s) w_j$$

The objective function is quadratic in the parameters $\boldsymbol{w}$ and stochastic gradient converges on the global optimum, with an appropriate step size. In linear function approximation;

$$\nabla_{\boldsymbol{w}} \hat{V}(s, \boldsymbol{w}) = \boldsymbol{x}(s)$$
$$\Delta \boldsymbol{w} = \alpha(V^\pi(s) - \hat{V}(s, \boldsymbol{w}))\boldsymbol{x}(s)$$

In MC, the return R_t is an unbiased but noisy sample of the true value $V^\pi(s_t)$. With linear Monte-Carlo policy evaluation (converges to the global optimum);

$$\Delta \boldsymbol{w} = \alpha(R_t - \hat{V}(s, \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{V}(s, \boldsymbol{w}) = \alpha(R_t - \hat{V}(s, \boldsymbol{w}))\boldsymbol{x}(s_t)$$

Similarly, this can be done with TD. However, the TD-target $R_{t+1} + \gamma\hat{V}(s_{t+1}, \boldsymbol{w})$ is a biased (by the initial values) single sample of the true value.

$$\begin{aligned}\Delta\boldsymbol{w} &= \alpha(r + \gamma\hat{V}(s', \boldsymbol{w}) - \hat{V}(s, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{V}(s_t, \boldsymbol{w}) \\ &= \alpha(r + \gamma\hat{V}(s', \boldsymbol{w}) - \hat{V}(s, \boldsymbol{w}))\boldsymbol{x}(s_t)\end{aligned}$$

It converges close to the global optimum. A simple algorithm is as follows;

1. initialise $\boldsymbol{w} = \mathbf{0}$, $k = 1$
2. loop
   (a) sample tuple (s_k, a_k, r_k, s_{k+1}) given π
   (b) update the weights $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(r + \gamma\boldsymbol{x}(s')^\top\boldsymbol{w} - \boldsymbol{x}(s)^\top\boldsymbol{w})\boldsymbol{x}(s)$
   (c) $s_k \leftarrow s_k + 1$
   (d) $k \leftarrow k + 1$

Introduction to Deep Learning and DQN

An example is the Atari DQN by *DeepMind*. The function $Q(s, a)$ is approximated by a neural network, and the raw pixels are processed by convolutional layers for feature extraction. There are a total of 18 actions; 8 directions, pushing the button with 8 directions, doing nothing, and pushing the button without touching the joystick.

We have an environment that can be sensed with sensors. The sensor data can be put into a feature extractor, that ML is applied to. This gives us knowledge about the environment, which we can perform some form of reasoning with. Planing can be done, which can lead to the generation of actions for some effector, which directly affects the environment.

Classically, deep learning is really about taking the sensor data, performing feature extraction, and doing machine learning. This is a form of representation. Reinforcement learning does the stages that involve reasoning, planning, and generation of actions, whereas recognition is the part that involves ML and recognition. Deep reinforcement learning combines all of this.

Given the TD Q-learning update (see below), we want to learn Q as a function parameterised (neural network) by $\boldsymbol{w}$

$$Q(s_t, a) \leftarrow Q(s_t, a) + \alpha[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a') - Q(s_t, a)]$$

The TD error is defined as our learning target, that we want to reduce to zero;

$$r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a'; \boldsymbol{w}) - Q(s_t, a; \boldsymbol{w})$$

Taking this gradient, with respect to $\boldsymbol{w}$, we obtain the following;

$$\Delta\boldsymbol{w} = \alpha[r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \boldsymbol{w}) - Q(s_t, a; \boldsymbol{w})]\nabla_{\boldsymbol{w}}Q(s_t, a; \boldsymbol{w})$$

However, the problem (Atari) required more engineering to solve properly;

- **experience replay**

  A function approximator, especially a CNN, has a lot of parameters, which can easily lead to overfitting. If we only learn by looking at the last few episodes, we become very limited in how the system is generalising and learning. Therefore, the first complication is the inefficient use of our interactive experience. Training a deep NN requires may updates, each of which is a timestep in the game; if we use the standard Q updates, each state transition is only used once and then discarded. As such, we'd constantly need to revisit the same state transition in order to train, which is inefficient and slow.

The distribution of experiences is also poor if we only look at the current episode. If we have an interactive learning procedure, the generated training samples are highly correlated as the agent is running the current policy. This makes a certain set of state action pairs highly likely. As our policy changes, it leads to a shift in the probability distribution of the input data. The network will forget past transitions (overwriting past memory, known as extinction). Changes in one part of the network can have side-effects on other parts in backpropagation.

Experience replay stores traces (experiences) and replays these in mini-batches to train the network on more than just the last episode. This helps to overcome the issue of correlation, as the network doesn't care about the sequentiality of the actions. Mini-batches are only feasible when multiple passes being run with the same data is stable with respect to the samples (transitions should have low variance for next immediate outcomes (reward and next state) for a given state action pair). Replaying past data is also a more efficient use of previous experience.

The learning phase is separated from gaining experience; it's based on taking random samples from the mini-batch table. DQN interleaves acting and learning; once the policy is improved, different behaviour will be experienced (explore actions that are closer to optimal ones and shift the replay buffer).

In summary;

- reduce correlation between experiences (mini-batches make samples more iid)
- increased learning speed
- avoid forgetting past transitions and rewards

The algorithm is as follows;

1. initialise replay memory $\mathcal{D}$ to capacity N
2. initialise action-value function Q with random weights (neural network)
3. for episode $\in [1, M]$, repeat;
   (a) initialise state s_t
   (b) for $t \in [1, T]$ repeat;
      i. with probability ϵ choose a random action a_t or select $a_t = \max_a Q^*(s_t, a; \theta)$
      ii. execute action a_t then observe reward r_t and state s_{t+1}
      iii. store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
      iv. set $s_{t+1} = s_t$
      v. sample random mini-batch of transitions (s_t, a_t, r_t, s_{t+1})
      vi. set y_j as follows;

$$y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} Q^*(s_t, a'; \theta) & \text{for non-terminal } s_{t+1} \end{cases}$$

      vii. perform gradient descent on $(y_j - Q(s_t, a_j, \theta))^2$

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\nabla_{\theta_i} Q(s, a; \theta)\right)\right]$$

- **target network**

The neural network can interpolate and affect nearby states. Updating one state may update nearby states, which could lead to unstable training. For example, changes to $Q(s, a)$ change $Q(s', a)$, which changes $Q(s, a)$, and so on.

This can be resolved by setting up a second network that helps us compute the TD error calculation. The general idea is to slow things down with a separate target network Q', both of

which approximate our Q-network, but one is updated at a much lower frequency (for example, every 1000 steps) with a copy of the latest learned weight parameters from the other network. The main Q-network is simply Q and the target network is Q'. When the TD-error is being calculated, Q' (slow) is used, and infrequently Q' is updated to be $Q' = Q$. While Q fluctuates a lot, the last N states aren't visible in the updates.

- **clipping of rewards**

  In Atari, each game has different score scales (for example the score scale for *Pong* is different from that of *Space Invaders*). The different scales lead to unstable training. Clipping rewards clips the rewards, with all positive rewards being set to 1 and negative rewards to -1 (no differentiation in the scale of the rewards).

- **skipping of frames**

  Games can be simulated much faster than a human can react; implying that the design and required actions can be slower. Many transitions were irrelevant, as nothing was really happening at that rate. Instead of updating the representation (state input) every frame, only every 4^{th} frame was used. The past 4 frames were used as inputs (allowing differences between the frames to be analysed).

In regular deep Q-learning, the target network is used for both identifying the action with the highest Q value and determining the Q value of this action. However, the maximum Q value may be overestimated; an unusually high value from the main Q network doesn't mean there's an unusually high value from the target Q' network. The two tasks are separated in double Q-learning (each one handling something different; one handling determining the highest value and the other determining the value). The slowly updating network determines which action has the highest value, whereas the fast updating network determines the value of the action, with more updated information. The two values are independent from each other, hence we are less susceptible to variance.

Deep Reinforcement Learning (Live Lecture)

So far, we've been looking at value-based methods, which approximates an optimal value function (mapping an action to a value, with some parameter $\boldsymbol{w}$ in the DQN example). These are more sample efficient and steady. On the other hand, there are policy-based methods, which directly find the optimal policy without a Q or V function, this has the parameter $\boldsymbol{\theta}$. These typically converge faster and are better suited for continuous and stochastic environments.

Consider an extremely simple setup; there are two boxes, which are labelled -1 and $+1$, which have the rewards -1 and 1, respectively. The two actions are to choose the -1 box or to choose the $+1$ box. However, consider at the start, the policy is equal for both, hence $p(- \mid \_) = \frac{1}{2} = p(+ \mid \_)$ (note that there is no real state, as it is a single step game).

The average return from this is $\frac{1}{2} \cdot -1 + \frac{1}{2} \cdot 1 = 0$. However, we see from playing the game that choosing the $+$ action is better, so we increase the probability to $\frac{2}{3}$ (and therefore decrease the $-$ probability to $\frac{1}{3}$). This gives an average return of $\frac{1}{3}$. We can continue this sequence, decreasing the $-$ probability to $\frac{1}{4}$ and increasing the $+$ probability to $\frac{3}{4}$, and so on, eventually giving us $p(+ \mid \_) = 1$ and $p(- \mid \_) = 0$. Once we notice a change in a given direction is good; we kept changing in that direction.

In a policy-based method, we want to directly learn the policy without an intermediate value function. The parameterised policy π depends on the parameters $\boldsymbol{\theta}$. The probability of a trace, given a parameter, is as follows (starting with the probability of starting in s_1) - this is sometimes referred to as $p_{\boldsymbol{\theta}}(\tau)$;

$$p_{\boldsymbol{\theta}}(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_{\boldsymbol{\theta}}(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t)$$

The optimal policy parameters are defined as the maximum average return, across all the possible traces;

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \mathbb{E}\left[\sum_{t=1}^{T} r(s_t, a_t)\right]_{\tau \propto p_{\boldsymbol{\theta}}(\tau)}$$

Previously, the policy was generated from the value function (which was learnt, and actions selected based on estimated action-values). We want to now a parameterised policy which can select actions without consulting a value function. A value function may be used to help (see actor-critics), but is not required. An example is that in robotics, it may be difficult to quantify the value of an action. For example, we may not care how we grab an object, just that we have. Any parametric method can be used to learn the parameters.

Let us define the policy weight vector as $\boldsymbol{\theta}$ and the policy as $\pi(a \mid s, \boldsymbol{\theta})$ (a conditional probability of choosing a given that the agent is in state s with the policy weight vector). The performance measure $J(\boldsymbol{\theta}) = V^{\pi}(s_0)$ is the total expected return from starting at the state s_0. We're trying to maximise J.

The simplest way of doing this is to obtain gradients by using finite difference. The partial derivative is as follows, where $\boldsymbol{u_k}$ is a unit vector that is zero everywhere other than the k^{th} component, and ϵ is some small positive amount;

$$\frac{J(\boldsymbol{\theta} + \boldsymbol{u_k}\epsilon) - J(\boldsymbol{\theta})}{\epsilon}$$

This gives a simple update rule which updates the k^{th} parameter with some learning rate α;

$$\theta_k \leftarrow \theta_k + \alpha \frac{J(\boldsymbol{\theta} + \boldsymbol{u_k}\epsilon) - J(\boldsymbol{\theta})}{\epsilon}$$

This is simple, however but it is noisy and inefficient (since we have to run simulations). However, it works even when the policy is non-differentiable and can be effective if other PG methods fail.

A direct policy gradient directly computes values. The policy gradient theorem gives us a simplification of how to compute things;

$$J(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{t=1}^{T} r(s_t, a_t)\right]_{\tau \propto p_{\boldsymbol{\theta}}(\tau)} \approx \sum_{1}^{N} \sum_{t} r(s_{i,t}, a_{i,t})$$

Note the following identity (which comes from the chain rule when taking the derivative of a logarithm);

$$\pi_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) = \pi_{\boldsymbol{\theta}}(\tau) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\tau)}{\pi_{\boldsymbol{\theta}}(\tau)} = \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\tau)$$

The derivation is as follows;

$$\begin{aligned}
r(\tau) &= \sum_{t=1}^{T} r_{s_t, a_t} \\
J(\boldsymbol{\theta}) &= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}(\tau)}[r(\tau)] \\
&= \int \pi_{\boldsymbol{\theta}}(\tau) r(\tau) \, \mathrm{d}\tau \\
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \int \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\tau) r(\tau) \, \mathrm{d}\tau \\
&= \int \pi_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) r(\tau) \, \mathrm{d}\tau \\
&= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}(\tau)}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) r(\tau)]
\end{aligned}$$

The final set of equations is the policy gradient theorem; the derivative of the expected reward is the expectation of the product of the return and gradient of the log of the policy $\pi_{\boldsymbol{\theta}}$.

However, recall our previous definition of $\pi_{\boldsymbol{\theta}}(\tau)$, and taking logarithms of both sides;

$$\log \pi_{\boldsymbol{\theta}}(\tau) = \log p(s_1) + \sum_{t=1}^{T} (\log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) + \log p(s_{t+1} \mid s_t, a_t))$$

In the policy gradient theorem, we're taking the gradient with respect to $\boldsymbol{\theta}$, hence any terms that don't rely on $\boldsymbol{\theta}$ fall out (those in red). Therefore, we can write it as;

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right]$$

The gradient of our cost-to-go (J) is the expected return of the gradient of the log of the probability of choosing an action for a trace times the return of that trace. This removes the transition dynamics, which would be hard to measure, as well as the probability of being in a starting state.

We are essentially weighting the grad log steps by the return, and we can think about this as a way of averaging over policies by setting the weight of the policy to be proportional to the return generated. We are increasing the weights of policy steps that increase rewards, and implicitly decreasing the weight of those which are bad for rewards.

The REINFORCE algorithm involves the following steps (iterate back to 1 after 3);

1. sample $\{\tau^i\}$ from $\pi_{\boldsymbol{\theta}}(a_t \mid s_t)$ (run the policy)
2. calculate the gradient of the parameters;

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_i \left(\sum_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t^i \mid s_t^i) \right) \left(\sum_t r(s_t^i, a_t^i) \right)$$

3. update parameters

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

Not only do these methods allow us to learn without a value function, they also allow us to use continuous actions (rather than discrete).

With Gaussian policies, actions are simply set to a continuous probability distribution;

$$\begin{aligned} \pi_{\boldsymbol{\theta}}(a_t \mid s_t) &= \mathcal{N}(f_{\text{nn}}(s_t); \boldsymbol{\Sigma}) \\ \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) &= -\frac{1}{2} ||f(s_t) - a_t||_{\Sigma}^2 + c \\ \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) &= -\frac{1}{2} \boldsymbol{\Sigma}^{-1} (f(s_t) - a_t) \frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{\theta}} \end{aligned}$$

By using a Gaussian, we have the convenient property that the max of the action is the mean; this is non-trivial for general continuous policies.

The actor-critic model splits learning into two parts; the actor for computing an action based on the state, and the critic for producing the Q values of the action (intuitively, the actor does something, and the critic judges how 'good' it is). The actor takes in a state and outputs an action; it controls how the agent learns with policy-based learning (learning the optimal policy). The critic evaluates this action by computing the value function (with value-based learning). These models compete and learn more efficiently than two separate methods.

The naive actor-critic is as follows;

1. initialise both model parameters, $\boldsymbol{\theta}$ and $\boldsymbol{w}$, pick a starting state s and action $a = \pi_{\boldsymbol{\theta}}(a \mid s)$
2. for each timestep $t = 1, \ldots, T$;

(a) take a sample, get immediate reward r and next state s'

(b) sample the next action $a' \sim \pi_{\boldsymbol{\theta}}(a' \mid s')$

(c) update the actor (policy model parameters) - note that we use the Q value estimate, rather than the (empirical) return

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{w}} Q(s,a) \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(a \mid s)$$

(d) calculate TD error

$$\delta = r + \gamma Q_{\boldsymbol{w}}(s', a') - Q_{\boldsymbol{w}}(s,a)$$

(e) update the value model parameters

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_{\boldsymbol{w}} \delta \nabla_{\boldsymbol{w}} Q_{\boldsymbol{w}}(s,a)$$

(f) update the next state $s \leftarrow s'$ and action $a \leftarrow a'$

The critic's judgement is used, instead of the empirical returns (a form of bootstrapping). This gives lower variance since we can update at each step with the Q values (not just at the end of an episode), additionally, it also gives lower variance since we bootstrap off our own Q value predictions. This allows us to learn in a non-episodic domain, as well as faster policy convergence within an episode.

The Q function can be broken down into 2 terms, the state value function $V(s)$ (the value of being in a state) and the advantage value $A(s,a)$, which is the benefit / detriment of choosing an action, hence $Q(s,a) = V(s) + A(s,a)$. This is known as the advantage actor-critic. The advantage function, $A(s,a) = Q(s,a) - V(s)$, captures how much better an action is compared to others in a given state, or how much better it is compared to the average action (uniform random).

A2C learns A values rather than Q values; the evaluation is based on how much better the action can be as well as how good the action is. This reduces the high variance of policy networks and stabilises the model (we are multiplying by much smaller numbers). The algorithm is as follows (go to 1 after 5);

1. sample $\{\tau^i\}$ from $\pi_{\boldsymbol{\theta}}(a_t \mid s_t)$ (run the policy)
2. fit $V^{\pi}_{\boldsymbol{w}}(s)$ to the sampled reward sums
3. evaluate $A^{\pi}(s_i, a_i) = r(s_i, a_i) + V^{\pi}_{\boldsymbol{w}}(s'_i) - V^{\pi}_{\boldsymbol{w}}(s_i)$
4. calculate the gradient of the parameters;

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_i \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_i \mid s_i) A^{\pi}(s_i, a_i)$$

5. update parameters

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

Note that the reward function isn't differentiated; therefore there is no requirement for it to be continuous, hence a single reward for 'success' can be used.

The Asynchronous Advantage Actor-Critic (A3C) has multiple independent agents (own weights), who interact with different copies of the environment in parallel. This allows for a bigger part of the state action space to be explored in less time. The individual workers are trained in parallel, which periodically update a global deep network (which holds shared parameters, and are fed back into the workers). This may not be as efficient as A2C, despite exploring more.

In classic value-based Q learning, it was trivial to find the optimal action, by taking a maximum, however this is non-trivial for general continuous action policies. If the action space is continuous, and the function $Q^*(s,a)$ is differentiable, then we can perform gradient ascent to find the maximum. We can use the mean function $\mu(s) = a$ to approximate the maximum; $\max_a Q^*(s,a) \approx Q(s, \mu(s))$.

DDPG (deep deterministic policy gradients) is a deterministic policy, model-free, off-policy, actor-critic algorithm which learns the value model Q and a policy. It also learns off-policy and uses the Bellman equation to learn the Q function (first use of bootstrapping) and uses the Q function to learn the policy (second use of bootstrapping). This is considered the continuous action version of the DQN. The algorithm is as follows (note that in the notation below $\boldsymbol{\theta^Q}$ is the weights of the critic, and $\boldsymbol{\theta^\mu}$ is the weights of the actor);

1. randomly initialise the critic network $Q(s, a \mid \boldsymbol{\theta^Q})$ and actor $\mu(s \mid \boldsymbol{\theta^\mu})$
2. randomly initialise target network Q' and μ' with the weights $\boldsymbol{\theta^{Q\prime}}$ and $\boldsymbol{\theta^{\mu\prime}}$ (same as the critic and actor weights initially)
3. initialise the replay buffer R
4. for episodes in $1, M$
   (a) initialise a random process $\mathcal{N}$ for action exploration
   (b) receive initial observation state s_1
   (c) for $t = 1, T$
      i. select an action $a_t = \mu(s_t \mid \boldsymbol{\theta^\mu}) + \mathcal{N}_t$ (according to the current policy, adding exploration noise to explore around the mean action)
      ii. execute the action a_t, observe the reward r_t, and the new state s_{t+1}
      iii. store the transition (s_t, a_t, r_t, s_{t+1}) in R
      iv. sample a minibatch of N transitions from R
      v. set up $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \boldsymbol{\theta^{\mu\prime}}) \mid \boldsymbol{\theta^{Q\prime}})$; consisting of the reward, the the value that would've been obtained with the action that would've been chosen (similar to TD target)
      vi. the critic is updated by minimising the squared loss

$$L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i \mid \boldsymbol{\theta^Q}))^2$$

      vii. the actor policy is updated using the sampled policy gradient

$$\nabla_{\boldsymbol{\theta^\mu}} J \approx \frac{1}{N} \nabla_a \, Q(s, a \mid \boldsymbol{\theta^Q})\big|_{s=s_i, a=\mu(s_i)} \, \nabla_{\boldsymbol{\theta^\mu}} \mu(s \mid \boldsymbol{\theta^\mu})\big|_{s_i}$$

      viii. update the target networks;

$$\boldsymbol{\theta^{Q\prime}} \leftarrow \tau \boldsymbol{\theta^Q} + (1 - \tau) \boldsymbol{\theta^{Q\prime}}$$
$$\boldsymbol{\theta^{\mu\prime}} \leftarrow \tau \boldsymbol{\theta^\mu} + (1 - \tau) \boldsymbol{\theta^{\mu\prime}}$$

This involves performing exploration in continuous spaces with Gaussian noise (to perturb a mean action) - this can be thought as being similar to ϵ-greedy, where we choose it closer to the intended action μ. Learning variability is controlled similar to DQN, with replay buffers and minibatches. Error-based learning is used on the mean-squared Bellman error. Policy gradient based learning is used on the actor network. Similarly to DQN, learning variability is controlled by a target network. However, this isn't done in sudden steps as in DQN (discrete) - instead it blends the values with smooth updates.