Dynamic programming

Dynamic programming (DP) algorithms can solve an MDP reinforcement learning task given the model of the environment (the state-transition probabilities and the reward function).

Finite-state MDP A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods.

Key idea the use of value functions to organize and structure the search for good policies.
Policy evaluation

How to compute the state-value function for a given policy?

- Choose initial approximation arbitrary, eg $\forall s, V_0(s) = 0$
- **Successive approximations are based on the Bellman equation**

$$V_{k+1}(s) = \sum_a \pi(a, s) \sum_{s', r} p(s', r | s, a) \left( r + \gamma V_k(s') \right)$$  \(1\)

- Stop when the value function stops changing

$$\max_s |V_{k+1}(s) - V_k(s)| < \theta$$
The reason for computing the value function for a given policy is to be able to find better policies.

**Policy improvement theorem** Let $\pi$ and $\pi'$ be any pair of deterministic policies such that, $\forall s \in S, Q_{\pi}(s, \pi'(s)) \geq V_{\pi}(s)$. Then the policy $\pi'$ must be as good as, or better than, $\pi$.

**Policy improvement**: constructing a greedy policy $\pi'$ which actions are better than the original policy $\pi$ in short term

$$
\pi'(s) = \arg \max_a \sum_{s', r} p(s', r| s, a) (r + V_{\pi}(s')) \tag{2}
$$

If the new policy is not better than the old policy then they are optimal.
Policy iteration

Once a policy $\pi$ has been improved using $V_\pi$ to yield a better policy $\pi'$, we can then compute $V_{\pi'}$ and improve it again to yield an even better $\pi''$.

Algorithm 1 Policy iteration

1: Initialise $V$ and $\pi$ arbitrarily
2: repeat
3: Evaluate $V$ using $\pi$ (Eq 1)
4: Improve $\pi$ using $V$ (Eq 2)
5: until convergence

$V_{k+1}^\pi(s) = \sum_a \tilde{\pi}(a|s) \sum_{s',r} p(s',r|s,a) (r + \gamma V_k^\pi(s'))$

$\pi$
Value iteration

One drawback of policy iteration is that it involves evaluating a policy at every step. Instead we can perform value iteration:

- Initialise values arbitrarily, e.g. $V_0(s) = 0$ for every $s$
- Successive approximations/improvements are based on

$$V_{k+1}(s) = \max_a \sum_{s', r} p(s', r | s, a) (r + \gamma V_k(s'))$$

(3)

- Stop when the value function stops changing

$$\max_s |V_{k+1}(s) - V_k(s)| < \theta$$
Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement.

**Algorithm 2 Value iteration**

1: Initialise $V_0$ arbitrarily  
2: repeat  
3: Improve $V_{k+1}$ using the estimate of $V_k$ (Eq 3)  
4: until convergence

$$V_{k+1}(s) = \max_a \sum_{s', r} p(s', r | s, a) (r + \gamma V_k(s'))$$
Asynchronous dynamic programming

**Drawback of DP methods**  They involve operations over the entire state set of the MDP. If the state space is very large, this becomes computationally prohibitive.

**Asynchronous algorithms**  Back up the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be backed up several times before the values of others are backed up once.
Generalised policy iteration

Policy evaluation and policy improvement processes interact, independent of the granularity of the two processes. The evaluation and improvement processes in GPI can be viewed as competing. They compete in the sense that they pull in opposing directions. Making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes that policy no longer to be greedy.

They can also be viewed as cooperating. In the long run, however, these two processes interact to find a single joint solution.
One might also think of the interaction between the evaluation and improvement processes in GPI in terms of two constraints or goals—for example, as two lines in two-dimensional space as suggested by the diagram to the right. Although the real geometry is much more complicated than this, the diagram suggests what happens in the real case. Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals. The goals interact because the two lines are not orthogonal. Driving directly toward one goal causes some movement away from the other goal. Inevitably, however, the joint process is brought closer to the overall goal of optimality. The arrows in this diagram correspond to the behavior of policy iteration in that each takes the system all the way to achieving one of the two goals completely. In GPI one could also take smaller, incomplete steps toward each goal. In either case, the two processes together achieve the overall goal of optimality even though neither is attempting to achieve it directly.

4.7 Efficiency of Dynamic Programming

DP may not be practical for very large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient. If we ignore a few technical details, then the (worst case) time DP methods take to find an optimal policy is polynomial in the number of states and actions. If \( n \) and \( k \) denote the number of states and actions, this means that a DP method takes a number of computational operations that is less than some polynomial function of \( n \) and \( k \). A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is \( k^n \). In this sense, DP is exponentially faster than any direct search in policy space could be, because direct search would have to exhaustively examine each policy to provide the same guarantee. Linear programming methods can also be used to solve MDPs, and in some cases their worst-case convergence guarantees are better than those of DP methods. But linear programming methods become impractical at a much smaller number of states than do DP methods (by a factor of about 100). For the largest problems, only DP methods are feasible.

DP is sometimes thought to be of limited applicability because of the curse of dimensionality, the fact that the number of states often grows exponentially with the number of state variables. Large state sets do create difficulties, but these are inherent difficulties of the problem, not of DP as a solution method. In fact, DP is comparatively better suited to handling large state spaces than competing methods such as direct search and linear programming.

In practice, DP methods can be used with today’s computers to solve MDPs with millions of states. Both policy iteration and value iteration are widely used, and it

---

**Generalised policy iteration**

![Diagram showing the process of generalised policy iteration](image)
Efficiency of dynamic programming

- If \( n \) and \( k \) denote the number of states and actions, the total number of (deterministic) policies is \( k^n \).
- A DP method takes a number of computational operations that is less than some polynomial function of \( n \) and \( k \).
- DP has limited applicability because of the curse of dimensionality, the fact that the number of states often grows exponentially with the number of state variables.
- This is the inherent difficulty of the problem, not of DP as a solution method.

\[
\vec{s} = (s_1, \ldots, s_m) \quad \vec{s} \in \{0,1\}^m \\
|S| = 2^m
\]
Policy evaluation refers to the iterative computation of the value functions for a given policy.

Policy improvement refers to the computation of an improved policy given the value function for that policy.

Putting these two computations together, we obtain policy iteration and value iteration, the two most popular DP methods.

Generalized policy iteration is the general idea of two interacting processes revolving around an approximate policy and an approximate value function.
Model-free reinforcement learning

- The complete model of the environment is not always available.
- In that case, the agent learns from *experience*.
- The experience can be obtained from *interaction* with simulated or real environment.
- Even-though the agent doesn’t have the model of the environment, it can still find the optimal behaviour.
Monte Carlo methods

- Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns.
- Monte Carlo methods sample and average returns for each state-action pair and average rewards for each action.
- They are typically applied to episodic tasks.
- The policy is updated only at the end of an episode.
Monte Carlo prediction

Estimates value function for a given policy.

Algorithm 3 Monte Carlo prediction

1: Initialise $V$ arbitrarily
2: $Returns(s) \leftarrow \text{empty list } \forall s \in \mathcal{S}$
3: repeat
4:  Generate an episode using $\pi$
5:  for $s$ in the episode do
6:     $Returns(s) \leftarrow \text{append return following } s$
7:  end for
8:  $V(s) = \text{average}(Returns(s))$
9: until convergence

Each average is an unbiased estimate, and the standard deviation of its error falls as $\frac{1}{\sqrt{n}}$, where $n$ is the number of returns averaged.
Monte Carlo estimation of the Q-function

state-action pairs are followed instead of states
maintains exploration all state-action pairs must be visited
In reinforcement learning we distinguish between:

**On-policy methods** which attempt to evaluate or improve the policy that is used to make decisions.

**Off-policy methods** which evaluate or improve a policy different from that used to generate the data.
On-policy Monte Carlo control

**Algorithm 4 On-policy Monte Carlo control**

1. Initialise $Q$ and $\pi$ arbitrarily
2. $\text{Returns}(s, a) \leftarrow$ empty list $\forall s \in S, a \in A$
3. repeat
4. for $s \in S$ and $a \in A$ do
5. Generate an episode using $\epsilon$-greedy $\pi$ starting with $s, a$
6. for $s, a$ in the episode do
7. $\text{Returns}(s, a) \leftarrow$ append return following $s, a$
8. $Q(s, a) = \text{average}(\text{Returns}(s, a))$
9. end for
10. for $s$ in the episode do
11. $\pi(s) = \arg \max_a Q(s, a)$
12. end for
13. end for
14. until convergence
Monte Carlo off-policy methods

In off-policy methods we have two policies:

- **target policy**: the policy being learned
  - The target policy is the greedy policy with respect to $Q$.
  - The target policy must have a non-zero probability of selecting all actions that might be selected by the target policy (coverage).

- **behaviour policy**: the policy that generates behaviour
  - The behaviour policy must have a non-zero probability of selecting all actions that might be selected by the target policy (coverage).
  - To ensure this we require the behaviour policy to be soft (i.e., that it select all actions in all states with non-zero probability).
  - The behaviour policy $\mu$ can be anything, but in order to assure convergence of $\pi$ to the optimal policy, an infinite number of returns must be obtained for each pair of state and action.
Algorithm 5 Off-policy Monte Carlo control

1: Initialise $Q$ arbitrarily, $C(s, a) = 0 \ \forall s \in S, a \in A \ \pi \leftarrow$ greedy with respect to $Q$
2: repeat
3: Generate an episode $[s_0, a_0, \ldots, a_{T-1}, s_T]$ using soft policy $\mu$
4: $R \leftarrow 0, W \leftarrow 1$
5: for $t = T$ down-to 0 do
6: $R \leftarrow \gamma R + r_{t+1}$
7: $C(s_t, a_t) \leftarrow C(s_t, a_t) + W$
8: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{W}{C(s_t, a_t)} (R - Q(s_t, a_t))$
9: $\pi(s) = \arg \max_a Q(s, a)$
10: if $a_t \neq \pi(s_t)$ then
11: Exit for loop
12: end if
13: $W \leftarrow W \frac{1}{\mu(a_t, s_t)}$
14: end for
15: until convergence
Recall Bellman Equation:

\[
Q^\pi(s,a) = \mathbb{E}[r_{t+1} | s_t = s, a_t = a] + \sum_{s', r} p(s', r | s, a) \max_{a' \in \mathcal{A}} Q^\pi(s', a')
\]

Now \(\pi\) is greedy w.r.t \(Q\), therefore

\[
\pi(s,a) = \mathbb{E}[r_{t+1} | s_t = s, a_t = a] + \sum_{s', r} p(s', r | s, a) \max_{a' \in \mathcal{A}} Q^\pi(s', a')
\]

\[\pi(s) = \arg \max_a Q(s, a)\]

Line 9 of the code ensures that actions are the same for the behaviour policy \(\mu\) and the greedy policy \(\pi\). Therefore, we know that

\[
Q^\mu(s_t = s, a_t = a | a_{t+1} = a') = \mathbb{E}[r_{t+1} | s_t = s, a_t = a]
\]

\[
+ \sum_{s', r} p(s', r | s, a) \mu(a_{t+1} = a' | s_{t+1} = s') Q(s', a')
\]

\[
Q^\pi(s_t = s, a_t = a | a_{t+1} = a') = \mathbb{E}[r_{t+1} | s_t = s, a_t = a, a_{t+1} = a']
\]

\[
+ \sum_{s', r} p(s', r | s, a) \pi(s', a')
\]

Thus,

\[
Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{w}{\mu(a_t, s_t)} (R - Q(s_t, a_t))
\]

\[W \leftarrow W \frac{1}{\mu(a_t, s_t)}\]

Note that \(R\) is the current estimation of return under policy \(\mu\), which should be re-scaled by \(\mu(a_t, s_t)\) for an estimation of value function under \(\pi\).
Model-based vs model-free methods

The Monte Carlo methods learn value functions and optimal policies from experience in the form of sample episodes.

They do not require the model of the environment and can be learned directly in the interaction with the environment or in simulation.

They simply average many returns for each state-action pair.

On-policy vs off-policy methods.