Section overview

Dynamic Programming

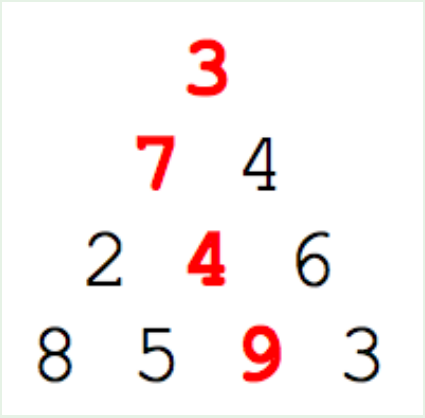
Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). We assume for convenience all MDPs to be finite. DP provides an essential foundation for understanding the RL methods out there. In fact, all of these machine learning methods can be viewed as ways to obtain the same effect as DP, only with less computation and without assuming a perfect model of the environment.

Maximum path sum

Example (Maximum path sum 1)

By starting at the top of the triangle below and moving to adjacent numbers on the row below, find the path with the maximum sum.



The maximum total from top to bottom is $3 + 7 + 4 + 9 = 23$.

Maximum path sum 2

Example

```
                75
               95 64
              17 47 82
             18 35 87 10
            20 04 82 47 65
           19 01 23 75 03 34
          88 02 77 73 07 63 67
         99 65 04 28 06 16 70 92
        41 41 26 56 83 40 80 70 33
       41 48 72 33 47 32 37 16 94 29
      53 71 44 65 25 43 91 52 97 51 14
     70 11 33 28 77 73 17 78 39 68 17 57
    91 71 52 38 17 14 91 43 58 50 27 29 48
   63 66 04 68 89 53 67 30 73 16 69 87 40 31
  04 62 98 27 23 09 70 98 73 93 38 53 60 04 23
```

How many possible routes are there? 16384

What is the maximum path sum? 1074

Maximum Path Sum solutions

Brute Force

The brute force approach to this problem would involve tracing every path and computing the maximum path cost. The time complexity of this algorithm is $\mathcal{O}(2^{n-1})$ where n is the number of rows in the triangle.

For 100 rows we need to evaluate 2^{99} paths.

Dynamic Programming

An efficient maximum path solution algorithm is to take a dynamic programming approach. We can split up the triangle into small sub-triangles, and working from the bottom-up, we can then compute the maximum path in a single pass. For each cell in the triangle we find the max of the two nodes below it and add that to the node value. This algorithm has time complexity $\mathcal{O}(n)$, where n is the number of nodes, a vast improvement over brute force.

Dynamic Programming – Origins 1

Bellman developed Dynamic programming (DP) to solve Markov Decision Processes. DP has grown beyond its original purpose into both a mathematical optimisation and a computer programming method. In these contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.
DP exploits the fact that decisions that span several points in time do often break apart recursively. Bellman called this the "Principle of Optimality".
A problem that can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems is said to have optimal substructure. If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems. In the optimisation literature this relationship is called the Bellman equation.

Dynamic Programming – Origins 2

To be be able to apply Dynamic Programming requires problems to have

1. Optimal substructure, meaning that the solution to a given optimisation problem can be obtained by the combination of optimal solutions to its sub-problems.
2. Overlapping sub-problems, meaning that the space of sub-problems must be small, i.e., any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.

Example

The maximum path sum problem or Dijkstra's algorithm for the shortest path problem are dynamic programming solutions. In contrast, if a problem can be solved by combining optimal solutions to **non-overlapping** sub-problems, the strategy is called "divide and conquer" instead (e.g. quick sort).

Policy Improvement

Our reason for computing the value function for a policy is to help find better policies.

Example

Suppose we have determined the value function for an arbitrary deterministic policy π. We know how good it is to follow the current policy π from s – the value is $V^{\pi}(s)$. Would it be better or worse to change to the new policy π', e.g. $\pi(s) = a$ and $\pi'(s) = a'$? One way is to select for state s alone a different action and continue using the old policy otherwise. The value of this must be, by definition $Q^{\pi}(s, a')$.
The key test is whether $Q^{\pi}(s, a') > V^{\pi}(s)$ or not. If it is greater, that is, if it is better to select a' in state s and thereafter follow $\pi(s)$, then the new policy would in fact be better overall.

Policy Improvement Theorem

Policy Improvement Theorem

Let π and π' be any two deterministic policies 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:
$V^{\pi'}(s) \geq V^{\pi}(s), \forall s \in \mathcal{S}$.

Note, that if the first inequality is strict in any state, then the latter inequality must be strict in at least one state.

Policy Improvement Theorem: Proof I

Consider a deterministic policy $\pi(s) = a$.

- We can always be as good or improve by acting greedily

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} Q^{\pi}(s, a) \tag{28}$$

- This must improve the value from any state s for one step,

$$Q^{\pi}(s, \pi'(s)) = \max_{a \in \mathcal{A}} Q^{\pi}(s, a) \geq Q^{\pi}(s, \pi(s)) = V^{\pi}(s) \tag{29}$$

Policy Improvement Theorem: Proof II

- Therefore, it follows it improves the value function, $V^{\pi'}(s) \geq V^{\pi}(s)$

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

- Halting condition: If there is no more improvement. This implies

$$Q^{\pi}(s, \pi(s')) = \max_{a \in \mathcal{A}} Q^{\pi}(s, a) = Q^{\pi}(s, \pi(s)) = V^{\pi}(s) \quad (30)$$

Policy Improvement Theorem: Proof III

- Then the Bellman Optimality Equation (BOE) must have been satisfied

$$V^{\pi}(s) = \max_{a \in \mathcal{A}} Q^{\pi}(s, a) \tag{31}$$

- Hence

$$V^{\pi}(s) = V^{\pi^*}(s) = V^*(s) \ \forall s \in \mathcal{S} \text{ and } \pi = \pi^* \qquad \square \tag{32}$$

Policy Iteration

Definition

Once a policy, π , has been improved using V^{π} to yield a better policy π', we can compute $V^{\pi'}$ and improve it again to π'', to yield an even better $V^{\pi''}$. We can thus obtain a sequence of monotonically improving policies and value functions. This way of finding an optimal policy is called policy iteration.

Note, that each policy evaluation is an iterative computation in itself, and is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation and policy iteration often converges in surprisingly few iterations. The policy improvement theorem assures us that these policies are better than the original policy.

Bellman's Principle of Optimality

Let us collect what we learned together:

Theorem (Principle of Optimality)

A policy $\pi(a|s)$ achieves the optimal value from state s, $V^{\pi}(s) = V^{*}(s)$, if and only if

1. *For any state s' reachable from s, i.e. $\exists a : p(s', s, a) > 0$*
2. *π achieves the optimal value from state s', $V^{\pi}(s') = V^{*}(s')$.*

Any optimal policy can be subdivided into two components:

1. An optimal action a^{*}.
2. Followed by an optimal policy from successor state s'

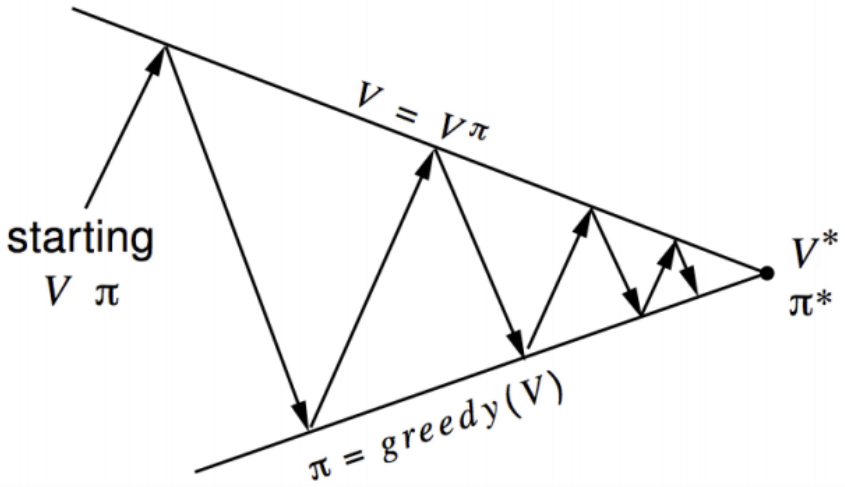
Policy Iteration Algorithm

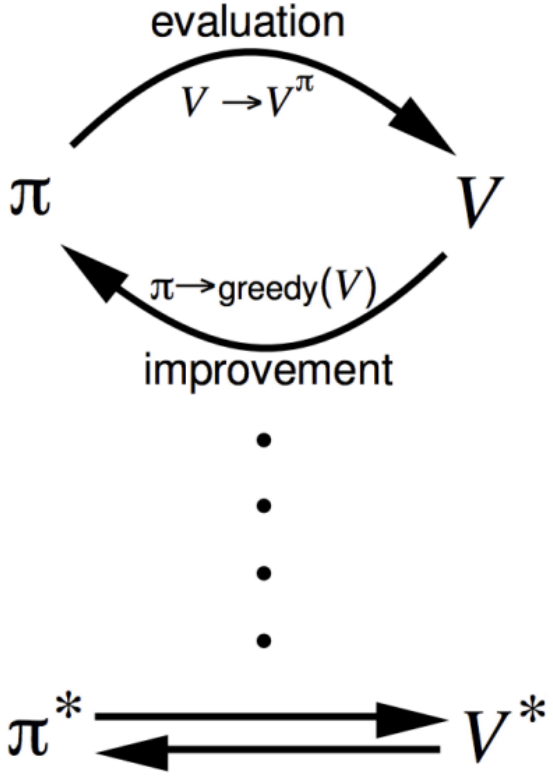
1. Initialization
   $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s') \right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $b \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s') \right]$
   If $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop; else go to 2

Generialised Policy Iteration Algorithm





Policy evaluation Estimate v_π
Any policy evaluation algorithm

Policy improvement Generate $\pi' \geq \pi$
Any policy improvement algorithm

More than one way to iterate a policy

One drawback to policy iteration is that each iteration involves a full policy evaluation (which can be protracted iterative computation requiring multiple sweeps through the state set). If policy evaluation is done iteratively, then convergence exactly to occurs only in the limit. Do we need to wait for policy evaluation need to converge to V^π?
We can introduce a stopping condition:

- ϵ-convergence of value function (e.g. $\forall s : V_{100}(s) - V_{101}(s) \leq \epsilon$)
- Stop after k iterations of iterative policy evaluation.

Prediction: Smaller k means more policy improvements and fewer policy iterations are executed till convergence.
Is there a trade-off between the two that minimises total numerical effort?

The $k = 1$ case

- The policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one backup of each state) of the value function. This algorithm is called value iteration.
- Value iteration is obtained simply by turning the Bellman Optimality Equation into an update rule.

Value Iteration

The Dynamic Programming you knew (shortest path algorithms or similar) are in fact just deterministic policy MDPs with deterministic actions, lets recast them in the language of MDPs:

- If we know the solution to subproblems $V^*(s')$
- Then solution $V^*(s)$ can be found by one-step look-ahead

$$V^*(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')] \tag{33}$$

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards (consider our maximal path sum example or ...)

Value Iteration Algorithm

Initialize V arbitrarily, e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$

Unlike policy iteration, there is no explicit policy.

Deterministic Value Iteration

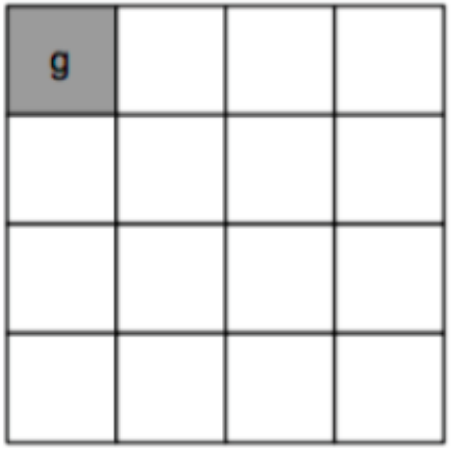
The Dynamic Programming you knew (shortest path algorithms or similar) are in fact just deterministic policy MDPs with deterministic actions, lets recast them in the language of MDPs:

- If we know the solution to subproblems $V^*(s')$
- Then solution $V^*(s)$ can be found by one-step look-ahead

$$V^*(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')] \tag{34}$$

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards (consider our maximal path sum example or ...)

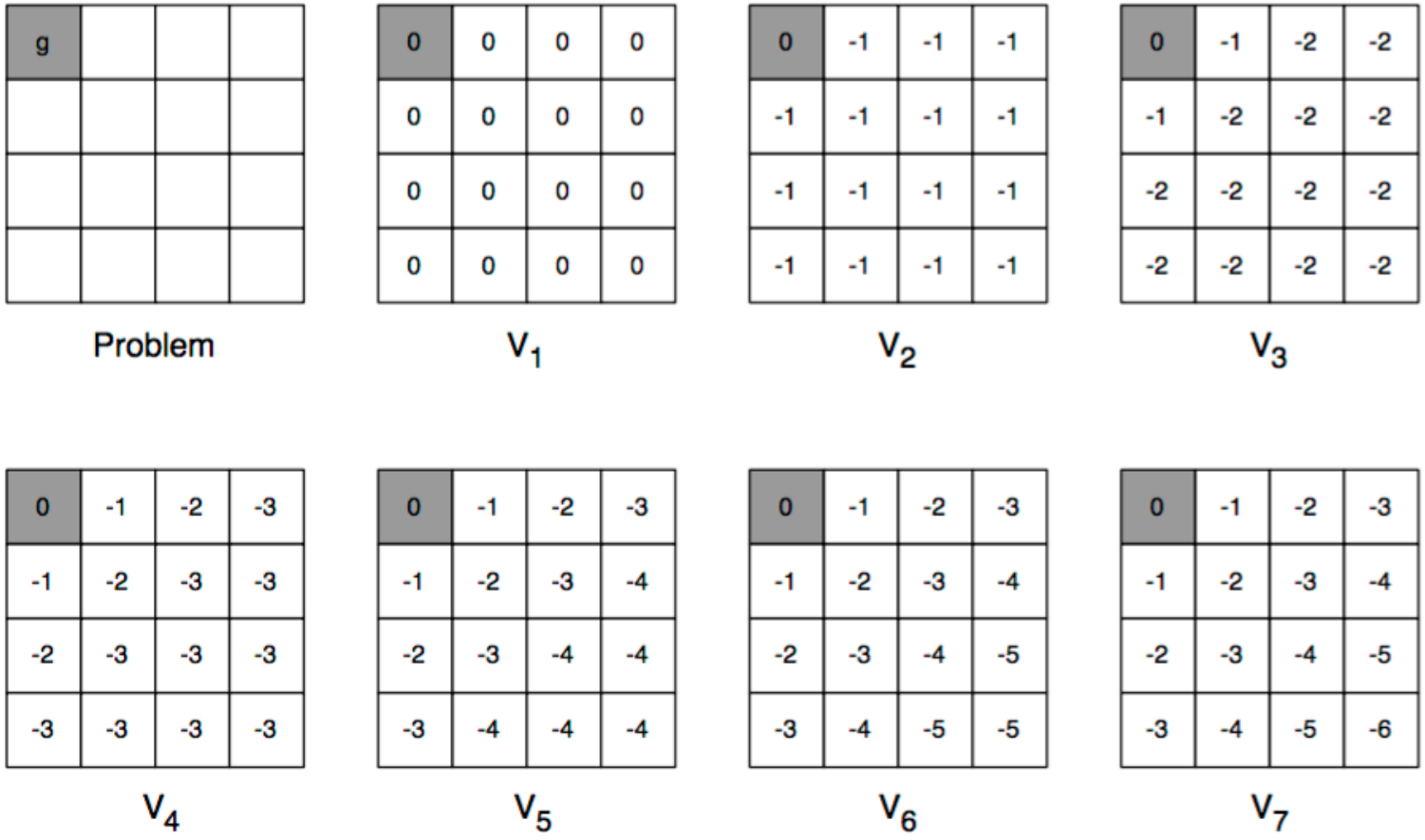
Shortest path problem in a grid world



Problem

- States equals locations in grid
- Walls impede transitions.
- 4 possible actions $\mathcal{A} = W, N, E, S$.
  No standing still.
- Each step one cost.
- Terminal/goal state (0 reward)
  top-left corner.

Deterministic value iteration in shortest path



Dynamic Programming performs full backups

Synchronous vs Asynchronous backups

- DP methods can use synchronous backups i.e. all states are backed up in parallel (this requires two copies of the value function)
- Asynchronous DP backups states target only states individually – in any order (one copy of value function)
  - For each selected state we apply the appropriate backup in-place
  - This can significantly reduce computation and
  - we are still guaranteed to converge if over time all states continue to be selected

Let us look at 2 simple ideas for smarter asynchronous DP

Let us take a step back from DP

- DP uses full-width backups
- For each synchronous or asynchronous backup
  - We need complete knowledge of the MDP transitions and reward function:
    aside from optimising the policy, our agent is not doing a lot of machine learning
  - Every successor state and action is considered
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers the Curse of Dimensionality:
  Number of states $N = |\mathcal{S}\|$ grows exponentially with number of (continuous) state variables
- Even one backup can be too expensive (e.g. consider control of an n-joint robot arm).

Bootstrapping

Dynamic Programming updates value estimates based on other value estimates – efficient use of data, thanks to the optimal sub-problem structure.

Bootstrapping: Update value estimates based on other value estimates. This is akin to the "magical" ability to pull oneself up by ones boot straps.

Note, in the case of DP here we look at self-consistency, so in the end each states values is grounded in all the states immediate

Let the learning begin

We will consider next sample backups:

- Using the rewards and transitions, which the agent samples by experience: S, A, R, S?
- instead of given reward function $\mathcal{R}$ and transition dynamics $\mathcal{P}$
- Advantages:
  1. Model-free: no advance knowledge of MDP required
  2. Breaks the curse of dimensionality through sampling (no full-backups, instead sample backups)
  3. Cost of backups remains constant and independent of $N = \|\mathcal{S}\|$