Reinforcement Learning

Lecture 4: Dynamic Programming

S. M. Ahsan Kazmi

Recap

- Last lecture: Formalise the problem with the full sequential structure
    - Markov Reward Processes
    - Markov Decision Processes

This lecture:

- Planning via DP
    - Policy Evaluation
    - Policy Iteration
    - Value Iteration
    - Extensions (Dynamic Programming)

Recap: Markov decision process (MDP)

- A Markov decision process (MDP) is a Markov reward process with decisions.
- It is an environment in which all states are Markov.

Definition

A *Markov Decision Process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is a state transition probability matrix,
  $\mathcal{P}_{ss'}^{a} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}_s^a = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$
- γ is a discount factor $\gamma \in [0, 1]$.

Recap: Solving the Bellman Optimality Equation

- Bellman Optimality Equation is non-linear
- No closed form solution (in general)
- Many iterative solution methods
  - Value Iteration
  - Policy Iteration
  - Q-learning
  - Sarsa

Ref: Sutton & Barto 2018, Chapter 3

Dynamic Programming?

Dynamic Programming

- A method for solving complex problems by breaking them down into subproblems
    - Solve the subproblems
    - Combine solutions to subproblems
- Dynamic Programming is a very general solution method for problems that have two properties:
- Optimal substructure
    - The principle of optimality applies
    - Optimal solution can be decomposed into subproblems
- Overlapping subproblems
    - Subproblems recur many times
    - Solutions can be cached and reused
- Markov decision processes satisfy both properties
    - Bellman equation gives a recursive decomposition
    - Value function stores and reuses solutions

Planning by Dynamic Programming

- Dynamic programming assumes full knowledge of the MDP
  - It is used for planning in an MDP
- For prediction:

  Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and policy π

  or: MRP $\langle \mathcal{S}, \mathcal{P}^{\pi}, \mathcal{R}^{\pi}, \gamma \rangle$

  Output: value function v_{π}

- Or for control:

  Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

  Output: optimal value function v_{*}

  and: optimal policy π_{*}

Policy Evaluation

Policy Evaluation

- Problem: evaluate a given policy π
- Solution: iterative application of Bellman expectation backup
- $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_\pi$
- Using synchronous backups

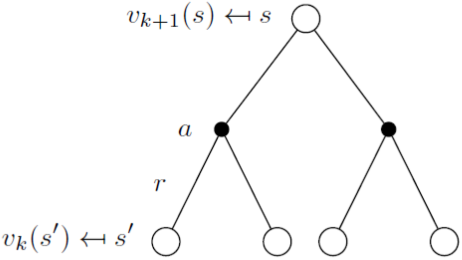
Algorithm

- First, initialise v_0, e.g., to zero
- Then, iterate

$$\forall s: \quad v_{k+1}(s) \leftarrow \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid s, \pi\right]$$

- Stopping: whenever $v_{k+1}(s) = v_k(s)$, for all s, we must have found v_π

Policy Evaluation

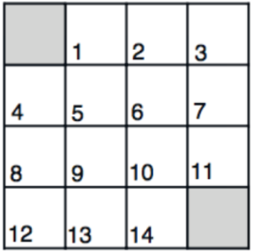


$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}^{k+1} = \boldsymbol{\mathcal{R}}^{\boldsymbol{\pi}} + \gamma \boldsymbol{\mathcal{P}}^{\boldsymbol{\pi}} \mathbf{v}^k$$

Example: Policy evaluation

actions



$R_t = -1$
on all transitions

- Undiscounted episodic MDP
- Nonterminal states 1……14
- One terminal state (shown twice as shaded squares)
- Actions leading out of the grid leave state unchanged
- Agent follows uniform random policy

$$\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25$$

Example: Policy evaluation

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

V_1 (1)= 1/4 [(−1+V_0 (1))+(−1+V_0 (T))+(−1+V_0 (2))+(−1+V_0 (5))]

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

V_2 (1)=1/4[(−1+0)+(−1+(−1))+(−1+(−1))+(−1+(−1))]=1/4[−1+−2+−2+−2]=−1.75

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Policy Iteration

How to Improve a Policy

- Given a policy π
  - Evaluate the policy π

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s\right]$$

  - Improve the policy by acting greedily with respect to v_π

$$\pi' = \text{greedy}(v_\pi)$$

- In Small Gridworld improved policy was optimal, $\pi' = \pi^*$
- In general, need more iterations of improvement/evaluation
- But this process of policy iteration always converges to $\pi*$

Policy Iteration

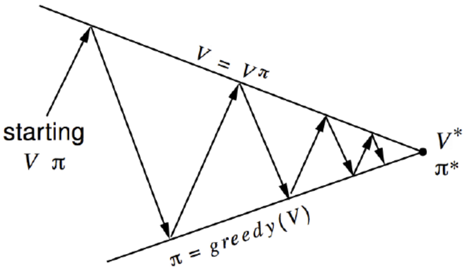
Algorithm

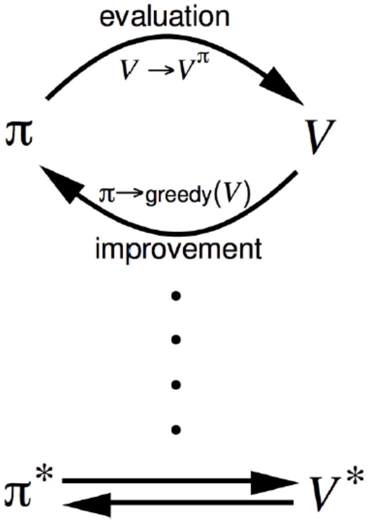
Iterate, using

$$\begin{aligned}\forall s: \quad \pi_{\text{new}}(s) &= \operatorname*{argmax}_a q_\pi(s, a) \\ &= \operatorname*{argmax}_a \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right]\end{aligned}$$

Then, evaluate π_{new} and repeat

Policy Iteration





Policy evaluation Estimate v_π

Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$

Greedy policy improvement

Modified Policy Iteration

- Does policy evaluation need to converge to v_π?

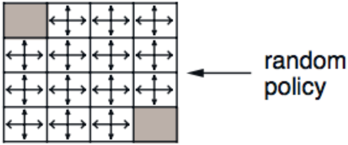
- Or should we introduce a stopping condition
  - e.g. ∈-convergence of the value function
- Or stop after k iterations of iterative policy evaluation?
  - For example, in the small grid-world, k = 3 was sufficient to achieve optimal policy

- Why not update the policy every iteration? i.e. stop after k = 1
  - This is equivalent to value iteration (next section)

Policy evaluation + Greedy Improvement

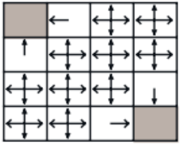
$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



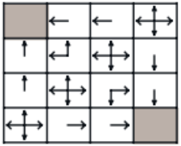
$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0



$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0



Policy evaluation + Greedy Improvement

$k = 3$

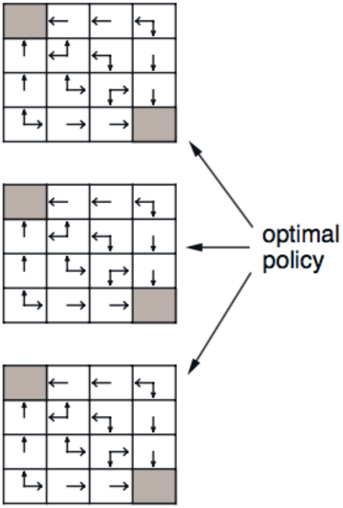
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



Value Iteration

Value Iteration

- If we know the solution to subproblems $v_*(s')$
- Then solution $v_*(s)$ can be found by one-step lookahead

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backward

Value Iteration

- Take the Bellman optimality equation, and turn that into an update

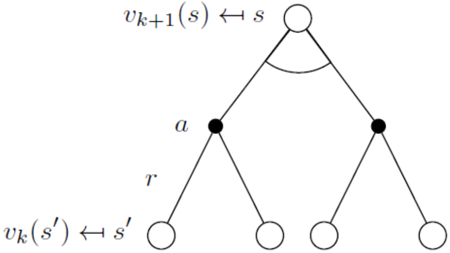
$$\forall s: \quad v_{k+1}(s) \leftarrow \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = s\right]$$

- Equivalent to policy iteration, with k = 1 step of policy evaluation between each two (greedy) policy improvement steps

Algorithm: Value Iteration

- Initialise v_0
- Update: $v_{k+1}(s) \leftarrow \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = s\right]$
- Stopping: whenever $v_{k+1}(s) = v_k(s)$, for all s, we must have found v^*

Value Iteration



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}_{k+1} = \max_{a \in \mathcal{A}} \boldsymbol{\mathcal{R}}^{\mathbf{a}} + \gamma \boldsymbol{\mathcal{P}}^{\mathbf{a}} \mathbf{v}_k$$

Example: Shortest Path

g			

Problem

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_5

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

Synchronous Dynamic Programming Algorithms

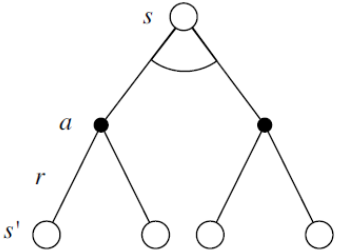
Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + (Greedy) Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

Extensions (Dynamic Programming)

Asynchronous Dynamic Programming

- DP methods described so far used synchronous backups
  - i.e. all states are backed up in parallel
- Asynchronous DP backs up states individually, in any order
  - For each selected state, apply the appropriate backup
  - Can significantly reduce computation
  - Guaranteed to converge if all states continue to be selected
- Three simple ideas for asynchronous dynamic programming:
  - In-place dynamic programming
  - Prioritised sweeping
  - Real-time dynamic programming

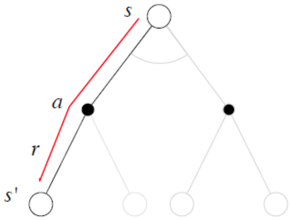
Full-Width Backups



- DP uses full-width backups
- For each backup (sync or async)
  - Every successor state and action is considered
  - Using knowledge of the MDP transitions and reward function
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers (Bellman's curse of dimensionality)
  - Number of states n grows exponentially with the number of state variables
- Even one backup can be too expensive

Sample Backups

- In subsequent lectures we will consider sample backups
- Using sample rewards and sample transitions $\langle s, a, r, s' \rangle$
  - (Instead of reward function r and transition dynamics p)
- Advantages:
  - Model-free: no advance knowledge of MDP required
  - Breaks the curse of dimensionality through sampling
  - Cost of backup is constant, independent of states



End of lecture