Dynamic Programming

Weizi Li

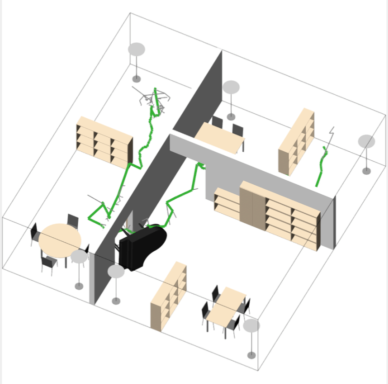
University of Tennessee, Knoxville

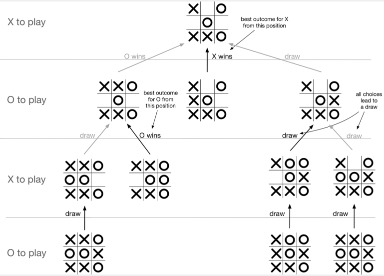
Content

Introduction

- Reinforcement Learning: agent improves its policy by interacting with an initially unknown environment
  - Model-free learning: agent doesn't infer the model of the environment during learning
  - Model-based learning: agent infers the model of the environment and uses the inferred model during learning

- Planning: agent improves its policy by computing with its internal model of the environment (model-based)





- Dynamic Programming (DP) assumes full knowledge of the MDP (including the model, i.e., transition function and reward function) and is used for planning in the MDP
- DP's focus is to solve MDP using the least computational effort

- DP is a general method for problems that have two properties:
  - 1) Optimal substructure: optimal solution can be decomposed for subproblems
  - 2) Overlapping subproblems: subproblems recur many times and their solutions can be reused
- DP solves a problem by breaking it down into subproblems
  - solve the subproblems
  - combine solutions to subproblems
- Example problem: finding the shortest path between A and B

- Scheduling algorithms
- String algorithms (e.g., sequence alignment)
- Graph algorithms (e.g., shortest path algorithms)
- Graphical models (e.g., Viterbi algorithm)
- Bioinformatics (e.g., lattice models)
- ...

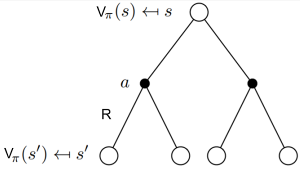
- MDPs have both properties required by DP:
  - Bellman equation gives recursive decomposition
  - Value function stores solutions for reuse
- An optimal policy can be subdivided into two components:
  - An optimal first action A^*
  - Followed by an optimal policy from successor state S'

- Prediction: evaluate a given policy (evaluate the future)
  - E.g., what is the value function for the uniform random policy?
  - Input: MDP and policy π
  - Output: value function V^π
- Control: find the best policy (optimize the future)
  - What is the optimal value function over all possible policies (hence the optimal policy)?
  - Input: MDP
  - Output: optimal value function V^* and optimal policy π^*

Policy Evaluation

- Policy evaluation solves the prediction problem: compute the state-value function V^π for an arbitrary policy π
- Recall: one direct method to solve Bellman expectation equation is via the normal equation (only works for small MDPs)
- The DP approach is to iteratively applying Bellman expectation equation until converge to V^π

- Initialize $V_0^\pi(s) = 0$
- At the $(k+1)$th iteration, update $V_{k+1}^\pi(s)$ from $V_k^\pi(s')$
- DP uses *bootstrapping*: updates a guess $\left(V_{k+1}^\pi(s)\right)$ using a guess $\left(V_k^\pi(s')\right)$

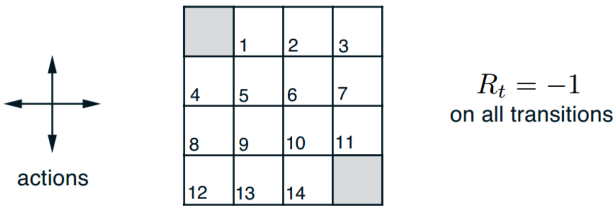


$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

Iterative Policy Evaluation: Example

- Evaluate the uniform random policy in an undiscounted, episodic task
  - Actions that would take the agent off the grid will leave the state unchanged

Example 4.1 Consider the 4×4 gridworld shown below.



Iterative Policy Evaluation: Example

- {Row 3, Column 4} from $k = 2$ to $k = 3$
- $-1.7 \rightarrow -2.4$:
  -0.25(0+2+2+1.7)-1 $=$ -2.425

v_k for the random policy

$k = 0$

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

$k = 3$

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

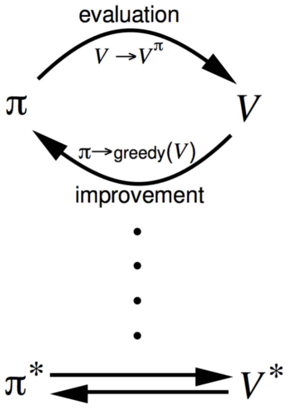
$k = \infty$

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

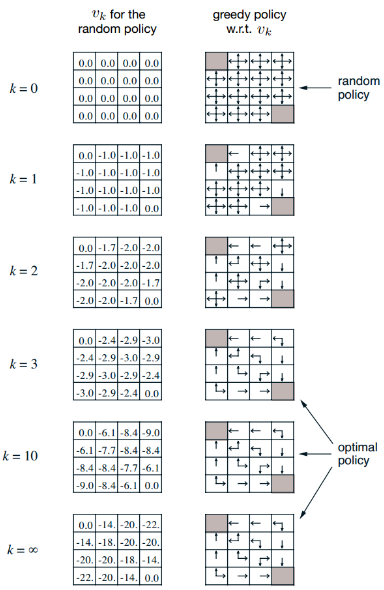
- In theory, iterative policy evaluation converges in the limit: $V_k^\pi \to V^\pi$ as $k \to \infty$
- In practice, we stop when $\| V_{k+1}^\pi - V_k^\pi \| \leq \epsilon$
- The existence and uniqueness of V^π are guaranteed providing:
  - $\gamma < 1$; or
  - any trajectory from any state under π is finite

Policy Iteration

- Policy iteration solves the control problem, i.e., finding π^*
  - policy evaluation: compute V^π for a given π
  - policy improvement: produce $\pi' \geq \pi$ by acting greedily to V^π
- Policy iteration will converge to π^*



- For a simple task, we can get π^* with one iteration of policy evaluation and policy improvement
- For the example in the figure, we can obtain π^* without converging to V^π in the first iteration of policy evaluation



- One drawback of the vanilla policy iteration is that each iteration involves policy evaluation, which can be computationally expensive if the number of states is large

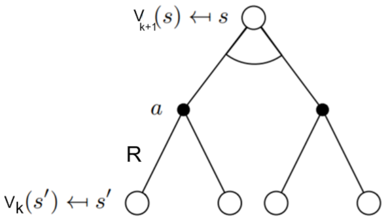
- Do we need to converge to V^{π} to obtain the optimal policy? No, we don't
  - Use a stopping condition, e.g., $\|V^{\pi}_{k+1} - V^{\pi}_{k}\| \leq \epsilon$
  - Simply stop after k iterations (we may then arrive at the optimal policy)
- In theory, the policy evaluation step can be truncated in several ways without losing the convergence guarantees of policy iteration

- Do we need to update policy in every iteration to obtain the optimal policy? No, we can use value iteration

Value Iteration

- Value iteration solves the control problem, i.e., finding π^*
- Solution: iteratively applying Bellman optimality equation until converge to V^*
- No direct method exists due to the nonlinearity of Bellman optimality equation
- Unlike policy iteration, there is no explicit policy update during the process
- Intermediate value functions may not correspond to any effective policy

- Initialize $V_0^\pi(s) = 0$
- At the $(k+1)$th iteration, update $V_{k+1}(s)$ from $V_k(s')$.



$$V_{k+1}(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_k(s') \right)$$

Value Iteration: Undiscounted Example

- Problem: find the shortest path from any state to the goal state (g)
  - Agent will stay if actions would take the agent off the grid
  - $R_t = -1$ on all transitions

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

- Policy iteration
  - policy evaluation (multiple iterations) + policy improvement (multiple iterations)
  - uses Bellman expectation equation
- Value iteration
  - finding optimal value function (multiple iterations) + optimal policy extraction (once)
  - uses Bellman optimality equation
- The performance is task-dependent
  - policy converge rate (policy iteration) vs. value convergence rate (value iteration)
  - if optimal policy is unnecessary, we can use policy iteration

- Both are widely used and converge much faster than their theoretical worst-case run times
- There is a spectrum of algorithms between Policy Iteration and Value Iteration conducting policy evaluation and policy improvement at different granularity
- In theory, initialization is independent of convergence rate; in practice, good initial value functions or policies can speed-up the convergence

Extensions of DP

- Synchronous DP processes all states at once using previous value functions.

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

- For m actions and n states:
  - Algorithms based on state-value function V, the complexity is $O(mn^2)$ per iteration
  - Algorithms based on action-value function Q, the complexity is $O(m^2n^2)$ per iteration
- DP is guaranteed to find an optimal policy in polynomial time, even though the total number of (deterministic) policies is m^n

- Asynchronous DP does not require processing all states at once, instead it can select any state to process, in any order
- Asynchronous DP can significantly reduce the computation and guarantee to converge if all states continue to be selected
- Example asynchronous DP algorithms:
  - In-place dynamic programming
  - Prioritized sweeping
  - Real-time dynamic programming

- DP uses full-width updates by considering every successor state and action
- DP is effective and efficient for medium-sized problems (millions of states)
- For large problems, DP suffers Bellman's curse of dimensionality (the number of states grows exponentially with the number of state variables); in this case, even one update can be cost prohibitive

- Update via sampling the environment transition function and reward function has two advantages
  - Model-free: no need to know the full MDP
  - Overcomes the curse of dimensionality: cost is constant, independent from the number of state variables