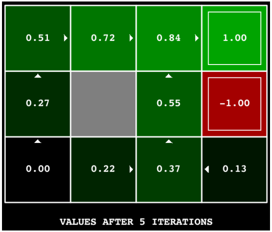
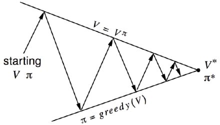
Reinforcement Learning

Solving MDPs



Marcello Restelli

February, 2022



Brute Force

- Solving an MDP means finding an **optimal policy**
- A **naive** approach consists of
  - **enumerating** all the deterministic Markov policies
  - **evaluate** each policy
  - **return** the best one
- The number of policies is **exponential**: $|\mathcal{A}|^{|\mathcal{S}|}$
- Need a **more intelligent search** for best policies
  - **restrict the search** to a subset of the possible policies
  - using **stochastic optimization** algorithms

What is Dynamic Programming?

- **Dynamic**: sequential or temporal component to the problem
- **Programming**: optimizing a “program”, i.e., a policy
  - c.f. linear programming
- A method for solving **complex** problems
- By breaking them down into **subproblems**
  - **Solve** the subproblems
  - **Combine** solutions to subproblems

Requirements for Dynamic Programming

- Dynamic Programming is a **very general** solution method for problems which have **two properties**:
  - **Optimal substructure**
    - **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 revursive 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
- **Prediction**
  - Input: MDP $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle$ and policy π (i.e., MRP $\langle \mathcal{S}, P^\pi, R^\pi, \gamma, \mu \rangle$)
  - Output: value function V^π
- **Control**
  - Input: MDP $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma, \mu \rangle$
  - Output: value function V^* and optimal policy π^*

Other Applications of Dynamic Programming

Dynamic Programming is used to solve many other problems:

- 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)

Finite–Horizon Dynamic Programming

- **Principle of optimality**: the tail of an optimal policy is optimal for the "tail" problem
- **Backward induction**
  - **Backward recursion**

$$V_k^*(s) = \max_{a \in \mathcal{A}_k} \left\{ R_k(s,a) + \sum_{s' \in \mathcal{S}_{k+1}} P_k(s'|s,a) V_{k+1}^*(s') \right\}, \quad k = N-1, \dots, 0$$

  - **Optimal policy**

$$\pi_k^*(s) \in arg \max_{a \in \mathcal{A}_k} \left\{ R_k(s,a) + \sum_{s' \in \mathcal{S}_{k+1}} P_k(s'|s,a) V_{k+1}^*(s') \right\}, \quad k = 0, \dots, N-1$$

- **Cost**: $N|\mathcal{S}||\mathcal{A}|$ vs $|\mathcal{A}|^{N|\mathcal{S}|}$ of brute force policy search
- From now on, we will consider **infinite–horizon discounted** MDPs

Policy Evaluation

- For a **given policy** π compute the **state–value function** V^π
- Recall
  - State–value function for policy π:

$$V^\pi(s) = \mathbb{E}\left\{\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right\}$$

  - **Bellman equation** for V^π:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)\left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s')\right]$$

  - A **system** of $|\mathcal{S}|$ simultaneous **linear equations**
  - Solution in **matrix** notation (complexity $O(n^3)$):

$$V^\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

Iterative Policy Evaluation

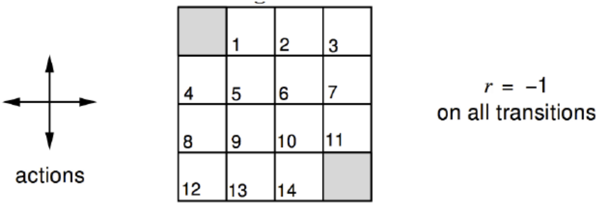
- Iterative application of Bellman expectation backup
- $V_0 \rightarrow V_1 \rightarrow \cdots \rightarrow V_k \rightarrow V_{k+1} \rightarrow \cdots \rightarrow V^\pi$
- A **full policy–evaluation backup**:

$$V_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_k(s') \right]$$

- A **sweep** consists of applying a backup operation to each state
- Using **synchronous** backups
  - At each iteration $k+1$
  - For all states $s \in \mathcal{S}$
  - Update $V_{k+1}(s)$ from $V_k(s')$

Example

Small Gridworld



- **Undiscounted episodic** MDP
  - $\gamma = 1$
  - All episodes terminate in **absorbing** terminal state
- **Transient** states $1, \ldots, 14$
- One **terminal** state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state **unchanged**
- Reward is -1 until the terminal state is reached

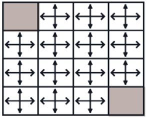
Policy Evaluation in Small Gridworld

V_k for the Random Policy

Greedy Policy w.r.t. V_k

$k = 0$

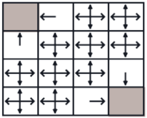
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



random policy

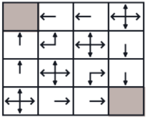
$k = 1$

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 in Small Gridworld

$k = 3$

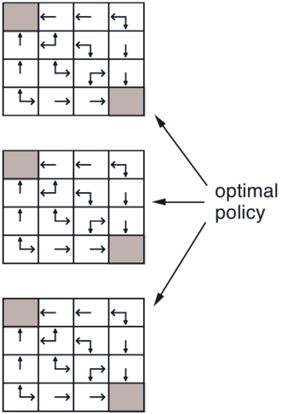
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

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

$k = \circ$

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



Policy Improvement

- Consider a **deterministic policy** π
- For a given state s, would it **better** to do an action $a \neq \pi(s)$?
- We can **improve** the policy by acting greedily

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

- This improves the value from **any** state s over one step

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

Policy Improvement Theorem

Theorem

Let π and π' be any pair of deterministic policies such that

$$Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s) \quad , \quad \forall s \in \mathcal{S}$$

Then the policy π' must be as good as, or better than π

$$V^{\pi'}(s) \geq V^{\pi}(s) \quad , \quad s \in \mathcal{S}$$

Proof.

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

$\square$

Policy Iteration

- What if improvements **stops** ($V^{\pi'} = V^{\pi}$)?

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

- But this is the **Bellman optimality equation**
- Therefore $V^{\pi}(s) = V^{\pi'}(s) = V^{*}(s)$ for all $s \in \mathcal{S}$
- So π is an **optimal** policy!

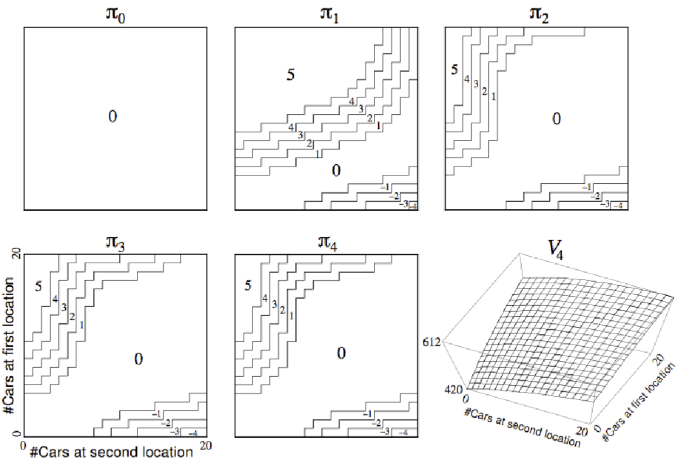
$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \cdots \rightarrow \pi^* \rightarrow V^* \rightarrow \pi^*$$

Example of Policy Iteration

Jack's Car Rental

- **States**: Two locations, maximum of 20 cars each
- **Actions**: Move up to 5 cars between two locations overnight
- **Reward**: \$10 for each car rented (must be available)
- **Transitions**: Cars returned and requested randomly
  - **Poisson distribution**, n returns/request with probability $\frac{\lambda^n}{n!}e^{-\lambda}$
  - **First location**: average requests = 3, average returns = 3
  - **Second location**: average requests = 4, average returns = 2

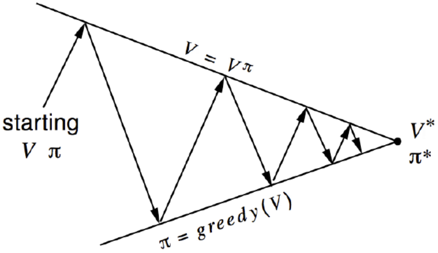
Example of Policy Iteration
Jack's Car Rental

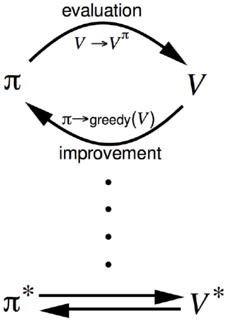


Modified Policy Iteration

- Does policy evaluation **need to converge** to V^π ?
- Or should we introduce a **stopping condition**
  - e.g., ϵ–convergence of value function
- Or simply **stop after** k **iterations** of iterative policy evaluation?
- For example, in the small gridworld $k = 3$ was sufficient to achieve optimal policy
- Why not update policy **every iteration**? i.e. stop after $k = 1$

Generalized Policy Iteration





- **Policy evaluation**: Estimate V^π
  - e.g., Iterative policy evaluation
- **Policy improvement**: Generate $\pi' \geq \pi$
  - e.g., Greedy policy improvement

Value Iteration

- **Problem**: find optimal policy π
- **Solution**: iterative application of Bellman optimality backup
- $V_1 \to V_2 \to \cdots \to V^*$
- Using **synchronous backups**
  - At each iteration $k+1$
  - For all states $s \in \mathcal{S}$
  - Update $V_{k+1}(s)$ from $V_k(s')$
- Unlike policy iteration there is **no explicit policy**
- **Intermediate** value functions **may not correspond** to any policy

Value Iteration demo:

`http://www.cs.ubc.ca/ poole/demos/mdp/vi.html`

Convergence and Contractions

Define the max–norm: $\|V\|_\infty = \max_s |V(s)|$

Theorem

Value Iteration converges to the optimal state-value function $\lim_{k\to\infty} V_k = V^*$

Proof.

$\|V_{k+1} - V^*\|_\infty = \|T^* V_k - T^* V^*\|_\infty \leq \gamma \|V_k - V^*\|_\infty \leq \cdots \leq \gamma^{k+1} \|V_0 - V^*\|_\infty \xrightarrow{k\to\infty} 0$ $\square$

Theorem

$$\|V_{i+1} - V_i\|_\infty < \epsilon \Rightarrow \|V_{i+1} - V^*\|_\infty < \frac{2\epsilon\gamma}{1-\gamma}$$

Synchronous Dynamic Programming Algorithms

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

- Algorithms are based on **state–value function** $V^{\pi}(s)$ or $V^{*}(s)$
- Complexity $O(mn^2)$ **per iteration**, for m actions and n states
- Could also apply to **action–value function** $Q^{\pi}(s,a)$ or $Q^{*}(s,a)$
- Complexity $O(m^2n^2)$ per **iteration**

Efficiency of DP

- To find optimal policy is **polynomial** in the number of states...
- **but**, the number of states is often astronomical, e.g., often growing **exponentially** with the number of state variables: **curse of dimensionality**
- In practice, classical DP can be applied to problems with a few millions states
- **Asynchronous DP** can be applied to larger problems, and appropriate for parallel computation
- It is surprisingly **easy** to come up with MDPs for which methods are not practical

Complexity of DP

- DP methods are **polynomial time** algorithms for **fixed–discounted** MDPs
- **Value Iteration**: $O(|\mathcal{S}|^2|\mathcal{A}|)$ for each iteration
- **Policy Iteration**: Cost of policy evaluation + Cost of policy iteration
  - Policy evaluation:
    - Linear system of equations: $O\left(|\mathcal{S}|^3\right)$ or $O\left(|\mathcal{S}|^{2.373}\right)$
    - Iterative: $O\left(|\mathcal{S}|^2 \frac{log(\frac{1}{\epsilon})}{log(\frac{1}{\gamma})}\right)$
  - Policy improvement: recently proven to be $O\left(\frac{|\mathcal{A}|}{1-\gamma}\log\left(\frac{|\mathcal{S}|}{1-\gamma}\right)\right)$
- **Each iteration** of PI is computationally **more expensive** than each iteration of VI
- PI typically requires fewer iterations to converge than VI
- **Exponentially faster** than any **direct policy search**
- Number of states often **grows exponentially** with the number of state variables

Asynchronous Dynamic Programming

- DP methods described so far used **synchronous** backups
  - i.e., all state 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 ideas for asynchronous DP:
  - In–place DP
  - Prioritized sweeping
  - Real–time DP

In–place Dynamic Programming

- **Synchronous** value iteration stores **two copies** of value function
  for all $s \in \mathcal{S}$
  $$V_{new}(s) \leftarrow \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_{old}(s') \right)$$
  $V_{old} \leftarrow V_{new}$
- **In–place** value iteration only stores **one copy** of value function
  for all $s \in \mathcal{S}$
  $$V(s) \leftarrow \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right)$$

Prioritized Sweeping

- Use the magnitude of **Bellman error** to guide state selection, e.g.,

$$\left| \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_k(s') \right) - V(s) \right|$$

- Backup the state with the **largest** remaining Bellman error
- **Update** Bellman error of affected states after each backup
- Requires knowledge of **reverse dynamics** (predecessor states)
- Can be implemented efficiently by maintaining a **priority queue**

Real–Time Dynamic Programming

- **Idea**: only states that are **relevant** to agent
- Use agent's **experience** to guide the **selection** of states
- After each time–step s_t, a_t, r_{t+1}

$$a_t \in arg \max_{a \in \mathcal{A}} \left(R(s_t, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s_t, a) V(s') \right)$$

- Backup the state s_t

$$V(s_t) \leftarrow \max_{a \in \mathcal{A}} \left(R(s_t, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s_t, a) V(s') \right)$$

Theorem

If $V_0 \geq V^*$ then $\exists \bar{t}$ such that a_t are optimal for all $t \geq \bar{t}$ (where $\bar{t} < \infty$ with probability 1)

Full–Width Backups

- Dynamic programming uses **full–width** backups
- For each backup (synchronous or asynchronous)
  - **Every** successor state and action is considered
  - Using knowledge of the MDP **transitions** and **reward function**
- Dynamic programming is effective for **medium–size** problems (millions of states)
- For large problems dynamic programming suffers Bellman's **curse of dimensionality**
  - Number of states $n = |\mathcal{S}|$ grows **exponentially** with number of states variables
- Even **one backup** can be too **expensive**

Sample Backups

- Reinforcement Learning techniques exploit **sample backups**
- Sample backups do not use reward function R and transition dynamics P
- Uses sample rewards and sample transitions $\langle s, a, s', r\rangle$
- Advantages
  - **Model–free**: no prior knowledge of MDP required
  - Breaks the curse of dimensionality through sampling
  - Cost of backups is constant, independent of $n = |\mathcal{S}|$

Approximate Dynamic Programming

- **Approximate** the value function
- Using a **function approximator** $V^{\theta}(s) = f(s, \theta)$
- Apply dynamic programming to V^{θ}
  - e.g., **Fitted Value Iteration** repeats at each iteration k
    - Sample states $\tilde{\mathcal{S}} \subseteq \mathcal{S}$
    - For each state $s \in \tilde{\mathcal{S}}$, estimate target value using Bellman optimality equation

$$\tilde{V}_k(s) = \max_{a \in \mathcal{A}} \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V_k^{\theta}(s') \right)$$

    - Train next value function V_{k+1}^{θ} using targets $\{\langle s, \tilde{V}_k(s) \rangle\}$

Infinite Horizon Linear Programming

- Recall, at value iteration convergence we have

$$\forall s \in \mathcal{S}: \quad V^*(s) = \max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right\}$$

- LP formulation to find V^*:

$$\begin{array}{ll} \min_V & \sum_{s \in \mathcal{S}} \mu(s) V(s) \\ \text{s. t.} & V(s) \geq R(s,a) + \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s'), \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \end{array}$$

  - $|\mathcal{S}|$ variables
  - $|\mathcal{S}||\mathcal{A}|$ constraints

Theorem

V^* is the solution of the above LP.

Theorem Proof

Let T^* be the **optimal Bellman operator**, then the LP can be written as:

$$\begin{array}{ll} \min_V & \mu^\mathrm{T} V \\ \text{s. t.} & V \geq T^*(V) \end{array}$$

- **Monotonicity property**: if $U \geq V$ then $T^*(U) \geq T^*(V)$.
- Hence, if $V \geq T^*(V)$ then $T^*(V) \geq T^*(T^*(V))$, and by **repeated application**,
$V \geq T^*(V) \geq T^{*2}(V) \geq T^{*3}(V) \geq \cdots \geq T^{*\infty}(V) = V^*$
- Any **feasible solution** to the LP must satisfy $V \geq T^*(V)$, and hence must satisfy $V \geq V^*$
- Hence, assuming all entries μ are positive, V^* is the **optimal solution** to the LP

Dual Linear Program

$$\begin{aligned}
\max_{\lambda} \quad & \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \lambda(s,a) R(s,a) \\
\text{s. t.} \quad & \sum_{a' \in \mathcal{A}} \lambda(s',a') = \mu(s) + \gamma \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} \lambda(s,a) P(s'|s,a), \quad \forall s' \in \mathcal{S} \\
& \lambda(s,a) \geq 0, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}
\end{aligned}$$

- **Interpretation**
  - $\lambda(s,a) = \sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s, a_t = a)$
  - Equation 2: ensures λ has the above meaning
  - Equation 1: maximize expected discounted sum of rewards
- **Optimal policy**: $\pi^*(s) = arg\max_a \lambda(s,a)$

Complexity of LP

- LP **worst–case** convergence guarantees are better than those of DP methods
- LP methods become **impractical** at a much smaller number of states than DP methods do