Lecture 01: Introduction to Reinforcement Learning

André Bodmer

Table of contents

Lecturer and Schedule

André
Bodmer

Contact

- E-Mail: andre.bodmer@sbb.ch or LinkedIn

Schedule

- 08:15 - 12:30 and 17:00 - 18:30 on both days

Course outline

The course will cover the following content:

- Conceptual basics and historical overview
- Markov decision processes
- Dynamic programming
- Monte Carlo learning
- Temporal difference learning
- Multi-step bootstrapping
- Planning and model-based RL
- Function approximation and (deep) learning
- Policy gradient methods
- Contemporary (deep) RL algorithms
- Further practical RL challenges (meta learning, safety, ...)

Recommended textbooks and links

- Reinforcement Learning: an introduction (textbook)
  - R. Sutton and G. Barto
  - MIT Press, 2nd edition, 2018 (click here)
- Reinforcement learning (lecture script)
  - D. Silver
  - Entire slide set (click here)
  - YouTube lecture series (click here)
- OpenAI Spinning Up in Deep Reinforcement Learning (click here)
- Google Colab
- Gymnasium
- OneDrive (exercises)
- Learn Git Branching
- ChatGPT (click here and here)
- Google DeepMind (Gemini)

Table of contents

The basic reinforcement learning structure

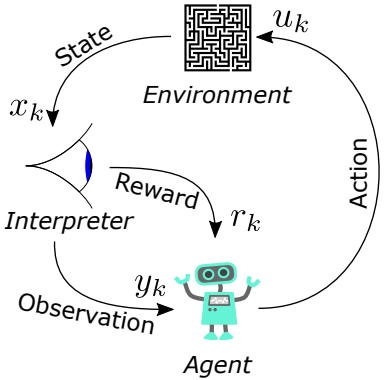


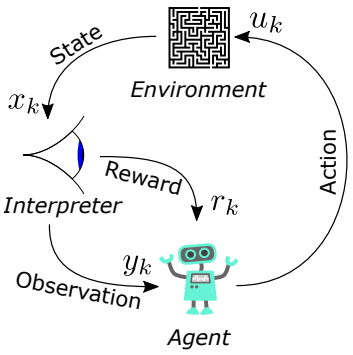
Fig. 1.1: The basic RL operation principle (derivative of www.wikipedia.org, CC0 1.0)

Key characteristics:

- No supervisor
- Data-driven
- Discrete time steps
- Sequential data stream (not i.i.d. data)
- Agent actions affect subsequent data (sequential decision making)

The nomenclature of this slide set is based on the default variable usage in control theory. In other RL books, one often finds s as state, a as action and o as observation.

Agent and environment



At each step k the agent:

- Picks an action u_k.
- Receives an observation y_k.
- Receives a reward r_k.

At each step k the environment:

- Receives an action u_k.
- Emits an observation y_{k+1}.
- Emits a reward r_{k+1}.

The time increments $k \leftarrow k + 1$.

Remark on time

A one step time delay is assumed between executing the action and receiving the observation as well as reward. We assume that the resulting time interval $\Delta t = t_k - t_{k+1}$ is constant.

Some basic definitions from the literature

What is reinforcement?

- "*Reinforcement is a consequence applied that will strengthen an organism's future behavior whenever that behavior is preceded by a specific antecedent stimulus.[...]There are four types of reinforcement: positive reinforcement, negative reinforcement, extinction, and punishment.*", wikipedia.org (obtained 2023-03-31)

What is learning?

- "*Acquiring knowledge and skills and having them readily available from memory so you can make sense of future problems and opportunities.*", From Make It Stick: The Science of Successful Learning, Brown et al., Harvard Press, 2014

Context around reinforcement learning

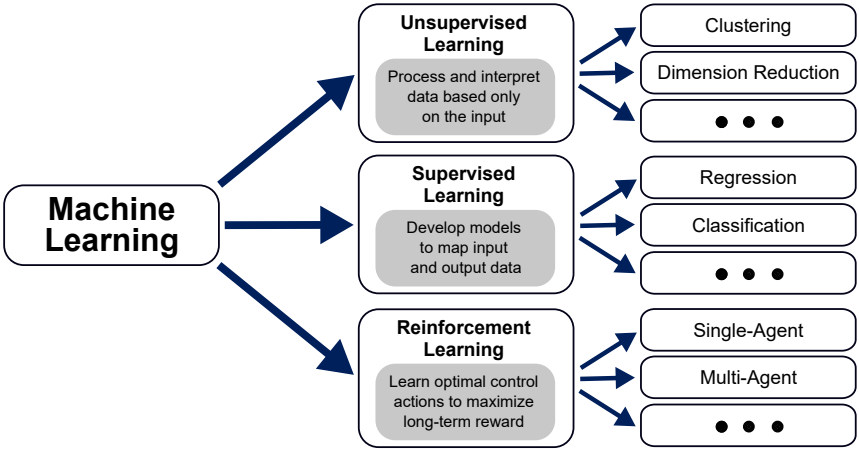
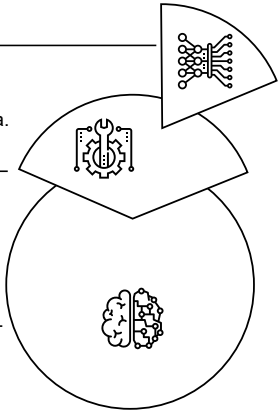


Fig. 1.2: Disciplines of machine learning

Context around machine learning



Deep Learning (DL)
A class of ML which uses large, layered models (e.g., vast artifical neural networks) to progressively extract more information from the data.

Machine Learning (ML)
A subset of AI involved with the creation of algorithms which can modify itself without human intervention to produce desired output by feeding itself through structured data.

Artifical Intelligence (AI)
Any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals. AI is often used to describe machines that mimic "cognitive" functions that humans associate with the human mind.

Fig. 1.3: The broader scope around machine learning

Many faces of reinforcement learning

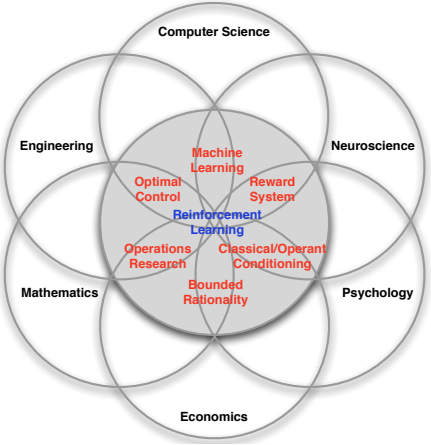


Fig. 1.4: RL and its neighboring domains
(source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

Table of contents

Methodical origins

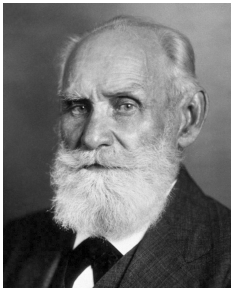


Fig. 1.5: Ivan Pavlov (1849-1936)

- Classical conditioning

Fig. 1.6: Andrei Markov (1856-1922)

- Stochastic process formalism

Fig. 1.7: Richard Bellman (1920-1984)

- Optimal sequential decision making

History of reinforcement learning

Huge field with many interconnections to different fields. One could give a lecture only on the historic development. Hence, interested readers are referred to:

- Chapter 1.7 of Barto/Sutton, *Reinforcement learning: an introduction*, 2nd edition, MIT Press, 2018
- 30 minutes talk of A. Barto (YouTube link)
- Survey papers on historic as well as more recent developments:
  - Kaelbling et al., *Reinforcement learning: A survey*, in Journal of Artificial Intelligence Research, vol. 4, pp. 237 - 285, 1996
  - Arulkumaran et al., *Deep reinforcement learning: a brief survey*, in IEEE Signal Processing Magazine, vol. 34, no. 6, pp. 26-38, 2017
  - Botvinick et al., *Reinforcement learning, fast and slow*, in Trends in Cognitive Sciences, vol. 23, iss. 5, pp. 408-422, 2019

Contemporary application examples

Limited selection from a broad field:

- Controlling electric drive systems
- Swinging-up and balance a cart-pole / an inverted pendulum
- Flipping pancakes with a roboter arm
- Drifting with a RC-car
- Driving an autonomous car
- Playing Atari Breakout
- Play strategy board game Go at super-human performance
- Training chat bots (like chatGPT)
- ...

Table of contents

Reward

- A reward is a scalar random variable R_k with realizations r_k.
- Often it is considered a real-number $r_k \in \mathbb{R}$ or an integer $r_k \in \mathbb{Z}$.
- The reward function (interpreter) may be naturally given or is a design degree of freedom (i.e., can be manipulated).
- It fully indicates how well an RL agent is doing at step k.
- The agent's task is to maximize its reward over time.

Theorem 1.1: Reward hypothesis

All goals can be described by the maximization of the expected cumulative reward:

$$\max \mathbb{E}\left[\sum_{i=0}^{\infty} R_{k+i+1}\right]. \tag{1.1}$$

Reward examples

- Flipping a pancake:
  - Pos. reward: catching the $180°$ rotated pancake
  - Neg. reward: droping the pancake on the floor
- Stock trading:
  - Trading portfolio monetary value
- Playing Atari games:
  - Highscore value at the end of a game episode
- Driving an autonomous car:
  - Pos. reward: getting save from A to B without crashing
  - Neg. reward: hitting another car, pedestrian, bicycle,...
- Classical control task (e.g., electric drive, inverted pendulum,...):
  - Pos. reward: following a given reference trajectory precisely
  - Neg. reward: violating system constraints and/or large control error

Reward characteristics

Rewards can have many different flavors and are highly depending on the given problem:

- Actions may have short and/or long term consequences.
  - The reward for a certain action may be delayed.
  - Examples: Stock trading, strategic board games,...
- Rewards can be positive and negative values.
  - Certain situations (e.g., car hits wall) might lead to a negative reward.
- Exogenous impacts might introduce stochastic reward components.
  - Example: A wind gust pushes an autonomous helicopter into a tree.

Remark on reward

The RL agent's learning process is heavily linked with the reward distribution over time. Designing expedient rewards functions is therefore crucially important for successfully applying RL. And often there is no predefined way on how to design the "best reward function".

The reward function hassle

- "Be careful what you wish for - you might get it" (pro-verb)
- "...it grants what you ask for, not what you should have asked for or what you intend." (Norbert Wiener, American mathematician)

Fig. 1.8: Midas and daughter (good as gold)
(source: www.flickr.com, by Robin Hutton CC BY-NC-ND 2.0)

Task-dependent return definitions

Episodic tasks

- A problem which naturally breaks into subsequences (finite horizon).
- Examples: most games, maze,...
- The return becomes a finite sum:

$$g_k = r_{k+1} + r_{k+2} + \cdots + r_N \,. \tag{1.2}$$

- Episodes end at their terminal step $k = N$.

Continuing tasks

- A problem which lacks a natural end (infinite horizon).
- Example: process control task
- The return should be discounted to prevent infinite numbers:

$$g_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{k+i+1} \,. \tag{1.3}$$

- Here, $\gamma \in \{\mathbb{R} | 0 \leq \gamma \leq 1\}$ is the discount rate.

Discounted rewards

Numeric viewpoint

- If $\gamma = 1$ and $r_k > 0$ for $k \to \infty$, g_k in (1.3) gets infinite.
- If $\gamma < 1$ and r_k is bounded for $k \to \infty$, g_k in (1.3) is bounded.

Strategic viewpoint

- If $\gamma \approx 1$: agent is farsighted.
- If $\gamma \approx 0$: agent is shortsighted (only interested in immediate reward).

Mathematical options

- The current return is the discounted future return:

$$\begin{aligned} g_k &= r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \cdots = r_{k+1} + \gamma \left(r_{k+2} + \gamma r_{k+3} + \cdots \right) \\ &= r_{k+1} + \gamma g_{k+1} \,. \end{aligned} \tag{1.4}$$

- If $r_k = r$ is a constant and $\gamma < 1$ one receives:

$$g_k = \sum_{i=0}^{\infty} \gamma^i r = r \sum_{i=0}^{\infty} \gamma^i = r \frac{1}{1-\gamma} \,. \tag{1.5}$$

State (1)

Environment state

- Random variable $\boldsymbol{X}_k^{\mathrm{e}}$ with realizations $\boldsymbol{x}_k^{\mathrm{e}}$
- Internal status representation of the environment, e.g.,
  - Physical states, e.g., car velocity or motor current
  - Game states, e.g., current chess board situation
  - Financial states, e.g., stock market status
- Fully, limited or not at all visible by the agent
  - Sometimes even 'foggy' or uncertain
  - In general: $\boldsymbol{Y}_k = \boldsymbol{f}(\boldsymbol{X}_k)$ as the measurable outputs of the environment
- Continuous or discrete quantity

Bold symbols are non-scalar multidimensional quantities, e.g., vectors and matrices.
Capital symbols denote random variables and small symbols their realizations.

State (2)

Agent state

- Random variable $\boldsymbol{X}_k^{\mathrm{a}}$ with realizations $\boldsymbol{x}_k^{\mathrm{a}}$
- Internal status representation of the agent
- In general: $\boldsymbol{x}_k^{\mathrm{a}} \neq \boldsymbol{x}_k^{\mathrm{e}}$, e.g., due to measurement noise or an additional agent's memory
- Agent's condensed information relevant for next action
- Dependent on internal knowledge / policy representation of the agent
- Continuous or discrete quantity

History and information state

Definition 1.1: History

The history is the past sequence of all observations, actions and rewards

$$\mathbb{H}_k = \{\boldsymbol{y}_0, r_0, \boldsymbol{u}_0, \ldots, \boldsymbol{u}_{k-1}, \boldsymbol{y}_k, r_k\} \tag{1.6}$$

up to the time step k.

If the current state $\boldsymbol{x}_k$ contains all useful information from the history it is called an information or Markov state (history is fully condensed in $\boldsymbol{x}_k$):

Definition 1.2: Information state

A state $\boldsymbol{X}_k$ is called an information state if and only if

$$\mathbb{P}\left[\boldsymbol{X}_{k+1} | \boldsymbol{X}_k\right] = \mathbb{P}\left[\boldsymbol{X}_{k+1} | \boldsymbol{X}_0, \boldsymbol{X}_1, \ldots, \boldsymbol{X}_k\right] . \tag{1.7}$$

Model examples with Markov states

Linear time-invariant (LTI) state-space model

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{A}\boldsymbol{x}_k + \boldsymbol{B}\boldsymbol{u}_k \,, \\ \boldsymbol{y}_k &= \boldsymbol{C}\boldsymbol{x}_k + \boldsymbol{D}\boldsymbol{u}_k \,. \end{aligned} \tag{1.8}$$

Nonlinear time-invariant state-space model:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{f}\left(\boldsymbol{x}_k, \boldsymbol{u}_k\right) \,, \\ \boldsymbol{y}_k &= \boldsymbol{h}\left(\boldsymbol{x}_k, \boldsymbol{u}_k\right) \,. \end{aligned} \tag{1.9}$$

Degree of observability

Full observability

- Agent directly measures full environment state (e.g., $\boldsymbol{y}_k = \boldsymbol{I}\boldsymbol{x}_k$).
- If $\boldsymbol{x}_k$ is Markov: Markov decision process (MDP).

Partial observability

- Agent does not have full access to environment state (e.g., $\boldsymbol{y}_k = \begin{bmatrix} \boldsymbol{I} & 0 \\ 0 & 0 \end{bmatrix} \boldsymbol{x}_k$).
- If $\boldsymbol{x}_k$ is Markov: partial observable MDP (POMDP).
- Agent may reconstructs state information $\hat{\boldsymbol{x}}_k \approx \boldsymbol{x}_k$ (belief, estimate).

POMDP examples

- Technical systems with limited sensors (cutting costs)
- Poker game (unknown opponents' cards)
- Human health status (too complex system)

Action

- An action is the agent's degree of freedom in order to maximize its reward.
- Major distinction:
  - Finite action set (FAS): $\boldsymbol{u}_k \in \{\boldsymbol{u}_{k,1}, \boldsymbol{u}_{k,2}, \ldots\} \in \mathbb{R}^m$
  - Continuous action set (CAS): Infinite number of actions: $\boldsymbol{u}_k \in \mathbb{R}^m$
  - Deterministic $\boldsymbol{u}_k$ or random $\boldsymbol{U}_k$ variable
  - Often state-dependent and potentially constrained:$\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k) \subseteq \mathbb{R}^m$
- Examples:
  - Take a card during Black Jack game (FAS)
  - Drive an autonomous car (CAS)
  - Buy stock options for your trading portfolio (FAS/CAS)

Remark on state and action spaces

Evaluating the state and action spaces (e.g., finite vs. continuous) of a new RL problem should be always the first steps in order to choose appropriate solution algorithms.

Policy

- A policy $\boldsymbol{\pi}$ is the agent's internal strategy on picking actions.
- Deterministic policies: maps state and action directly:

$$\boldsymbol{u}_k = \boldsymbol{\pi}(\boldsymbol{x}_k)\,. \tag{1.10}$$

- Stochastic policies: maps a probability of the action given a state:

$$\boldsymbol{\pi}(\boldsymbol{U}_k|\boldsymbol{X}_k) = \mathbb{P}\left[\boldsymbol{U}_k|\boldsymbol{X}_k\right]\,. \tag{1.11}$$

- RL is all about changing $\boldsymbol{\pi}$ over time in order to maximize the expected return.

Value functions

- The state-value function is the expected return being in state $\boldsymbol{x}_k$ following a policy $\boldsymbol{\pi}$: $v_\pi(\boldsymbol{x}_k)$.
- Assuming an MDP problem structure the state-value function is

$$v_\pi(\boldsymbol{x}_k) = \mathbb{E}_\pi \left[G_k | \boldsymbol{X}_k = \boldsymbol{x}_k \right] = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \middle| \boldsymbol{x}_k \right]. \tag{1.12}$$

- The action-value function is the expected return being in state $\boldsymbol{x}_k$ taken an action $\boldsymbol{u}_k$ and, thereafter, following a policy $\boldsymbol{\pi}$: $q_\pi(\boldsymbol{x}_k, \boldsymbol{u}_k)$.
- Assuming an MDP problem structure the action-value function is

$$q_\pi(\boldsymbol{x}_k, \boldsymbol{u}_k) = \mathbb{E}_\pi \left[G_k | \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k \right] = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \middle| \boldsymbol{x}_k, \boldsymbol{u}_k \right]. \tag{1.13}$$

- A key task in RL is to estimate $v_\pi(\boldsymbol{x}_k)$ and $q_\pi(\boldsymbol{x}_k, \boldsymbol{u}_k)$ based on sampled data.

Exploration and exploitation

- In RL the environment is initially unknown. How to act optimal?
- Exploration: find out more about the environment.
- Exploitation: maximize current reward using limited information.
- Trade-off problem: what's the best split between both strategies?

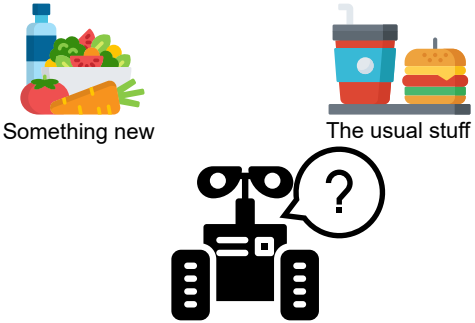


Fig. 1.9: The exploration-exploitation dilemma

Model

- A model predicts what will happen inside an environment.
- That could be a state model $\boldsymbol{\mathcal{P}}$:

$$\boldsymbol{\mathcal{P}} = \mathbb{P}\left[\boldsymbol{X}_{k+1} = \boldsymbol{x}_{k+1} | \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k\right] . \tag{1.14}$$

- Or a reward model $\mathcal{R}$:

$$\mathcal{R} = \mathbb{P}\left[R_{k+1} = r_{k+1} | \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k\right] . \tag{1.15}$$

- In general, those models could be stochastic (as denoted above) but in some problems relax to a deterministic form.
- Using data in order to fit a model is a learning problem of its own and often called system identification.

Table of contents

Maze example

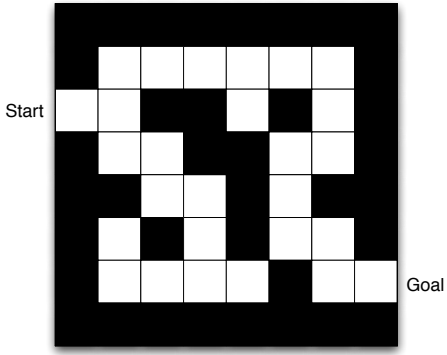


Fig. 1.10: Maze setup
(source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

Problem statement:

- Reward: $r_k = -1$
- At goal: episode termination
- Actions: $u_k \in \{N, E, S, W\}$
- State: agent's location
- Deterministic problem (no stochastic influences)

Maze example: RL-solution by policy

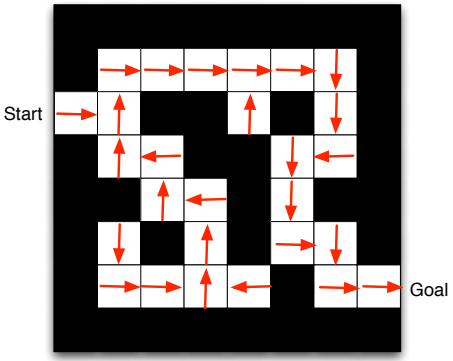


Fig. 1.11: Arrows represent policy $\pi(x_k)$ (source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

Key characteristics:

- For any state there is a direct action command.
- The policy is explicitly available.

Maze example: RL-solution by value function

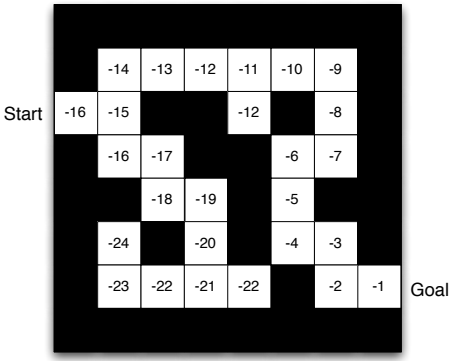


Fig. 1.12: Numbers represent value $v_\pi(x_k)$ (source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

Key characteristics:

- The agent evaluates neighboring maze positions by their value.
- The policy is only implicitly available.

Maze example: RL-solution by model evaluation

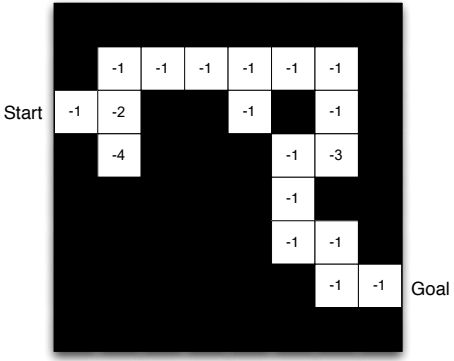


Fig. 1.13: Grid layout represents state model $\mathcal{P}$ and numbers depict the estimate by the reward model $\mathcal{R}$. (source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

Key characteristics:

- Agent uses internal model of the environment.
- The model is only an estimate (inaccurate, incomplete).
- The agent interacts with the model before taking the next action (e.g., by numerical optimizers).

RL agent taxonomy

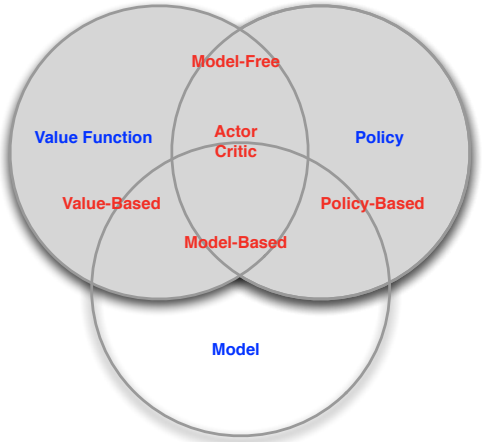


Fig. 1.14: Main categories of reinforcement learning algorithms
(source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

Summary: what you've learned in this lecture

- Understanding the role of RL in machine learning and optimal sequential decision making.
- Become acquainted with the basic RL interaction loop (agent, environment, interpreter).
- Finding your way around the basic RL vocabulary.
- Internalize the significance of proper reward formulations (design parameter).
- Differentiate solution ideas on how to retrieve an optimal agent behavior (policy).
- Delimit RL towards model predictive control as a sequential decision making alternative.

Lecture 02: Markov Decision Processes

André Bodmer

Preface

- Markov decision processes (MDP) are a mathematically idealized form of RL problems.
- They allow precise theoretical statements (e.g., on optimal solutions).
- They deliver insights into RL solutions since many real-world problems can be abstracted as MDPs.
- In the following we'll focus on:
  - fully observable MDPs (i.e., $\boldsymbol{x}_k = \boldsymbol{y}_k$) and
  - finite MDPs (i.e., finite number of states & actions).

		All states observable?	
		Yes	No
Actions?	No	Markov chain	Hidden Markov model
	Yes	Markov decision process (MDP)	Partially observable MDP

Tab. 2.1: Different Markov models

Scalar and vectorial representations in finite MDPs

- The position of a chess piece can be represented in two ways:
  - Vectorial: $\boldsymbol{x} = \begin{bmatrix} x_{\mathrm{h}} & x_{\mathrm{v}} \end{bmatrix}^{\mathsf{T}}$, i.e., a two-element vector with horizontal and vertical information,
  - Scalar: simple enumeration of all available positions (e.g., $x = 3$).
- Both ways represent the same amount of information.
- We will stick to the scalar representation of states and actions in finite MDPs.

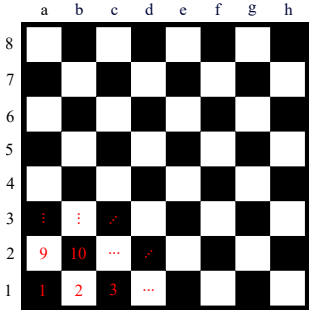


Table of contents

Markov chain

Definition 2.1: Finite Markov chain

A finite Markov chain is a tuple $\langle \mathcal{X}, \boldsymbol{\mathcal{P}} \rangle$ with

- $\mathcal{X}$ being a finite set of discrete-time states $X_k \in \mathcal{X}$,
- $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{P}}_{xx'} = \mathbb{P}\left[X_{k+1} = x' | X_k = x\right]$ is the state transition probability.

- Specific stochastic process model
- Sequence of random variables $X_k, X_{k+1}, \ldots$
- 'Memoryless', i.e., system properties are time invariant
- In continuous-time framework: Markov process[1]

[1] However, this results in a literature nomenclature inconsistency with Markov decision/reward 'processes'.

State transition matrix

Definition 2.2: State transition matrix

Given a Markov state $X_k = x$ and its successor $X_{k+1} = x'$, the state transition probability $\forall \{x, x'\} \in \mathcal{X}$ is defined by the matrix

$$\boldsymbol{\mathcal{P}}_{xx'} = \mathbb{P}\left[X_{k+1} = x' | X_k = x\right]. \tag{2.1}$$

Here, $\boldsymbol{\mathcal{P}}_{xx'} \in \mathbb{R}^{n \times n}$ has the form

$$\boldsymbol{\mathcal{P}}_{xx'} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & & & \vdots \\ \vdots & & & \vdots \\ p_{n1} & \cdots & \cdots & p_{nn} \end{bmatrix}$$

with $p_{ij} \in \{\mathbb{R} | 0 \leq p_{ij} \leq 1\}$ being the specific probability to go from state $x = X_i$ to state $x' = X_j$. Obviously, $\sum_j p_{ij} = 1 \, \forall i$ must hold.

Example of a Markov chain (1)

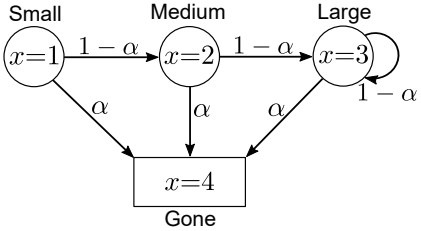


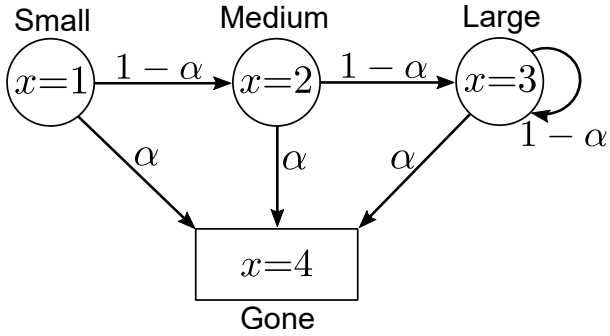
Fig. 2.1: Forest tree Markov chain

$$x \in \{1, 2, 3, 4\}$$
$$= \{\text{small}, \text{medium}, \text{large}, \text{gone}\}$$

$$\boldsymbol{\mathcal{P}} = \begin{bmatrix} 0 & 1-\alpha & 0 & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- At $x = 1$ a small tree is planted ('starting point').
- A tree grows with $(1 - \alpha)$ probability.
- If it reaches $x = 3$ (large) its growth is limited.
- With α probability a natural hazard destroys the tree.
- The state $x = 4$ is terminal ('infinite loop').

Example of a Markov chain (2)



Possible samples for the given Markov chain example starting from $x = 1$ (small tree):

- Small $\rightarrow$ gone
- Small $\rightarrow$ medium $\rightarrow$ gone
- Small $\rightarrow$ medium $\rightarrow$ large $\rightarrow$ gone
- Small $\rightarrow$ medium $\rightarrow$ large $\rightarrow$ large $\rightarrow \ldots$

Table of contents

Markov reward process

Definition 2.3: Finite Markov reward process

A finite Markov reward process (MRP) is a tuple $\langle \mathcal{X}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ with

- $\mathcal{X}$ being a finite set of discrete-time states $X_k \in \mathcal{X}$,
- $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{P}}_{xx'} = \mathbb{P}\left[X_{k+1} = x' | X_k = x\right]$ is the state transition probability,
- $\mathcal{R}$ is a reward function, $\mathcal{R} = \mathcal{R}_x = \mathbb{E}\left[R_{k+1} | X_k = x_k\right]$ and
- γ is a discount factor, $\gamma \in \{\mathbb{R} | 0 \leq \gamma \leq 1\}$.

- Markov chain extended with rewards
- Still an autonomous stochastic process without specific inputs
- Reward R_{k+1} only depends on state X_k

Example of a Markov reward process

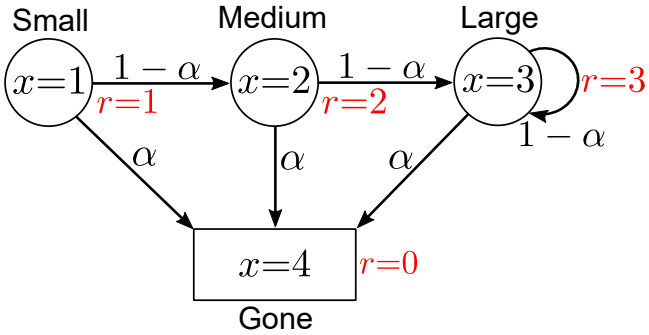


Fig. 2.2: Forest Markov reward process

- Growing larger trees is rewarded, since it will be
  - appreciated by hikers and
  - useful for wood production.
- Loosing a tree due to a hazard is unrewarded.

Recap on return

Return

The return G_k is the total discounted reward starting from step k onwards. For episodic tasks it becomes the finite series

$$G_k = R_{k+1} + \gamma R_{k+2} + \gamma^2 R_{k+3} + \cdots = \sum_{i=0}^{N} \gamma^i R_{k+i+1} \tag{2.2}$$

terminating at step N while it is an infinite series for continuing tasks

$$G_k = R_{k+1} + \gamma R_{k+2} + \gamma^2 R_{k+3} + \cdots = \sum_{i=0}^{\infty} \gamma^i R_{k+i+1}\,. \tag{2.3}$$

- The discount γ represents the value of future rewards.

Value function in MRP

Definition 2.4: Value function in MRP

The state-value function $v(x_k)$ of an MRP is the expected return starting from state x_k

$$v(x_k) = \mathbb{E}\left[G_k | X_k = x_k\right]. \tag{2.4}$$

- Represents the long-term value of being in state X_k.

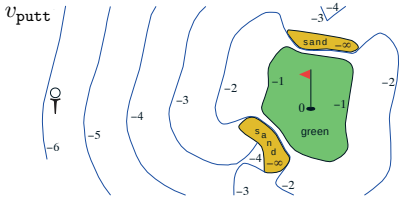
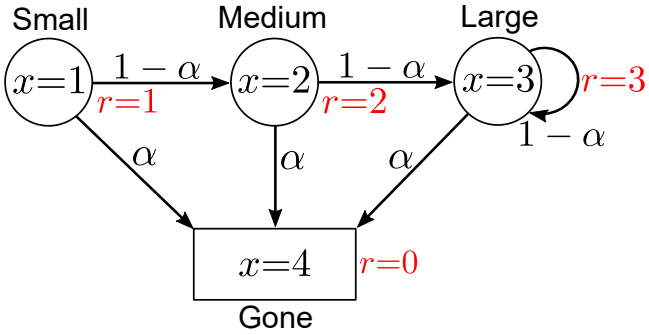


Fig. 2.3: Isolines indicate state value of different golf ball locations (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

State-value samples of forest MRP



Exemplary samples for $\hat{v}$ with $\gamma = 0.5$ starting in $x = 1$:

$$
\begin{aligned}
&x = 1 \to 4, && \hat{v} = 1, \\
&x = 1 \to 2 \to 4, && \hat{v} = 1 + 0.5 \cdot 2 = 2.0, \\
&x = 1 \to 2 \to 3 \to 4, && \hat{v} = 1 + 0.5 \cdot 2 + 0.25 \cdot 3 = 3.75, \\
&x = 1 \to 2 \to 3 \to 3 \to 4, && \hat{v} = 1 + 0.5 \cdot 2 + 0.25 \cdot 3 + 0.125 \cdot 3 = 4.13.
\end{aligned}
$$

Bellman equation for MRPs (1)

Problem: How to calculate all state values in closed form?
Solution: Bellman equation.

$$
\begin{aligned}
v(x_k) &= \mathbb{E}\left[G_k | X_k = x_k\right] \\
&= \mathbb{E}\left[R_{k+1} + \gamma R_{k+2} + \gamma^2 R_{k+3} + \ldots | X_k = x_k\right] \\
&= \mathbb{E}\left[R_{k+1} + \gamma \left(R_{k+2} + \gamma R_{k+3} + \ldots\right) | X_k = x_k\right] \\
&= \mathbb{E}\left[R_{k+1} + \gamma G_{k+1} | X_k = x_k\right] \\
&= \mathbb{E}\left[R_{k+1} + \gamma v(X_{k+1}) | X_k = x_k\right]
\end{aligned}
\tag{2.5}
$$

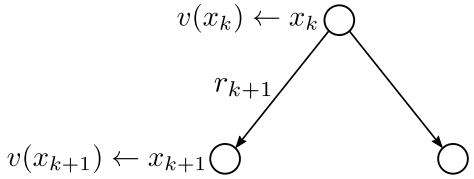


Fig. 2.4: Backup diagram for $v(x_k)$

Bellman equation for MRPs (2)

Assuming a known reward function $\mathcal{R}(x)$ for every state $X = x \in \mathcal{X}$

$$\boldsymbol{r}_{\mathcal{X}} = \begin{bmatrix}\mathcal{R}(x_1) & \cdots & \mathcal{R}(x_n)\end{bmatrix}^{\mathsf{T}} = \begin{bmatrix}\mathcal{R}_1 & \cdots & \mathcal{R}_n\end{bmatrix}^{\mathsf{T}} \tag{2.6}$$

for a finite number of n states with unknown state values

$$\boldsymbol{v}_{\mathcal{X}} = \begin{bmatrix}v(x_1) & \cdots & v(x_n)\end{bmatrix}^{\mathsf{T}} = \begin{bmatrix}v_1 & \cdots & v_n\end{bmatrix}^{\mathsf{T}} \tag{2.7}$$

one can derive a linear equation system based on Fig. 2.4:

$$\begin{aligned} \boldsymbol{v}_{\mathcal{X}} &= \boldsymbol{r}_{\mathcal{X}} + \gamma \boldsymbol{\mathcal{P}}_{xx'} \boldsymbol{v}_{\mathcal{X}}, \\ \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} &= \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & & \vdots \\ p_{n1} & \cdots & p_{nn} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}. \end{aligned} \tag{2.8}$$

Solving the MRP Bellman equation

Above, (2.8) is a normal equation in $\boldsymbol{v}_{\mathcal{X}}$:

$$\begin{aligned} \boldsymbol{v}_{\mathcal{X}} &= \boldsymbol{r}_{\mathcal{X}} + \gamma \boldsymbol{\mathcal{P}}_{xx'} \boldsymbol{v}_{\mathcal{X}}, \\ \Leftrightarrow \underbrace{(\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}_{xx'})}_{\boldsymbol{A}} \underbrace{\boldsymbol{v}_{\mathcal{X}}}_{x} &= \underbrace{\boldsymbol{r}_{\mathcal{X}}}_{\boldsymbol{b}}. \end{aligned} \tag{2.9}$$

Possible solutions are (among others):

- Direct inversion (Gaussian elimination, $\mathcal{O}(n^3)$),
- Matrix decomposition (QR, Cholesky, etc. , $\mathcal{O}(n^3)$),
- Iterative solutions (e.g., Krylov-subspaces, often better than $\mathcal{O}(n^3)$).

In RL identifying and solving (2.9) is a key task, which is often realized only approximately for high-order state spaces.

Example of a MRP with state values

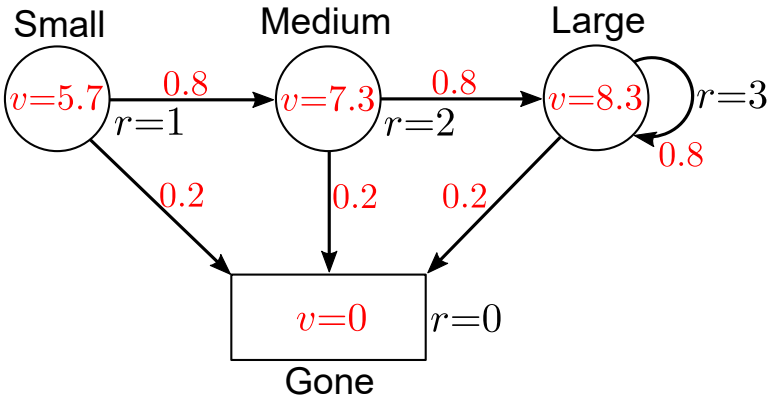


Fig. 2.5: Forest Markov reward process including state values

- Discount factor $\gamma = 0.8$
- Disaster probability $\alpha = 0.2$

Table of contents

Markov decision process

Definition 2.5: Finite Markov decision process

A finite Markov decision process (MDP) is a tuple $\langle \mathcal{X}, \mathcal{U}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ with

- $\mathcal{X}$ being a finite set of discrete-time states $X_k \in \mathcal{X}$,
- $\mathcal{U}$ as a finite set of discrete-time actions $U_k \in \mathcal{U}$,
- $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{P}}^u_{xx'}$ is the state transition probability $\boldsymbol{\mathcal{P}} = \mathbb{P}\left[X_{k+1} = x' | X_k = x_k, U_k = u_k\right]$,
- $\mathcal{R}$ is a reward function, $\mathcal{R} = \mathcal{R}^u_x = \mathbb{E}\left[R_{k+1} | X_k = x_k, U_k = u_k\right]$ and
- γ is a discount factor, $\gamma \in \{\mathbb{R} | 0 \leq \gamma \leq 1\}$.

- Markov reward process is extended with actions / decisions.
- Now, rewards also depend on action U_k.

Example of a Markov decision process (1)

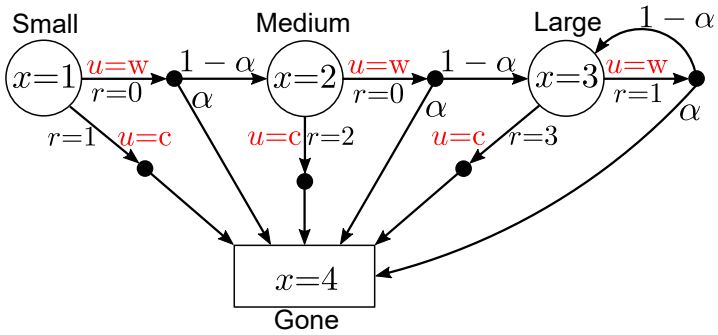


Fig. 2.6: Forest Markov decision process

- Two actions possible in each state:
  - Wait $u = w$: let the tree grow.
  - Cut $u = c$: gather the wood.
- With increasing tree size the wood reward increases as well.

Example of a Markov decision process (2)

For the previous example the state transition probability matrix and reward function are given as:

$$\boldsymbol{\mathcal{P}}_{xx'}^{u=c} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{\mathcal{P}}_{xx'}^{u=w} = \begin{bmatrix} 0 & 1-\alpha & 0 & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\boldsymbol{r}_{\mathcal{X}}^{u=c} = \begin{bmatrix} 1 & 2 & 3 & 0 \end{bmatrix}^{\mathsf{T}}, \quad \boldsymbol{r}_{\mathcal{X}}^{u=w} = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^{\mathsf{T}}.$$

- The term $\boldsymbol{r}_{\mathcal{X}}^{u}$ is the abbreviated form for receiving the output of $\mathcal{R}$ for the entire state space $\mathcal{X}$ given the action u.

Policy (1)

Definition 2.6: Policy in MDP (1)

In an MDP environment, a policy is a distribution over actions given states:

$$\pi(u_k|x_k) = \mathbb{P}\left[U_k = u_k | X_k = x_k\right] . \tag{2.10}$$

- In MDPs, policies depend only on the current state.
- A policy fully defines the agent's behavior (which might be stochastic or deterministic).

Fig. 2.7: What is you best Monopoly policy? (source: Ylanite Koppens on Pexels)

Policy (2)

Given a finite MDP $\langle \mathcal{X}, \mathcal{U}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ and a policy $\boldsymbol{\pi}$:

- The state sequence $X_k, X_{k+1}, \ldots$ is a Markov chain $\langle \mathcal{X}, \boldsymbol{\mathcal{P}}^{\pi} \rangle$ since the state transition probability is only depending on the state:

$$\boldsymbol{\mathcal{P}}^{\pi}_{xx'} = \sum_{u_k \in \mathcal{U}} \boldsymbol{\pi}(u_k|x_k) \boldsymbol{\mathcal{P}}^{u}_{xx'} \,. \tag{2.11}$$

- Consequently, the sequence $X_k, R_{k+1}, X_{k+1}, R_{k+2}, \ldots$ of states and rewards is a Markov reward process $\langle \mathcal{X}, \boldsymbol{\mathcal{P}}^{\pi}, \mathcal{R}^{\pi}, \gamma \rangle$:

$$\mathcal{R}^{\pi}_{xx'} = \sum_{u_k \in \mathcal{U}} \boldsymbol{\pi}(u_k|x_k) \mathcal{R}^{u}_{x} \,. \tag{2.12}$$

Recap on MDP value functions

Definition 2.7: State-value function

The state-value function of an MDP is the expected return starting in x_k following policy π:

$$v_\pi(x_k) = \mathbb{E}_\pi \left[G_k | X_k = x_k \right] = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \middle| X_k \right] .$$

Definition 2.8: Action-value function

The action-value function of an MDP is the expected return starting in x_k taking action u_k following policy π:

$$q_\pi(x_k, u_k) = \mathbb{E}_\pi \left[G_k | X_k = x_k, U_k = u_k \right] = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \middle| X_k, U_k \right] .$$

Bellman expectation equation (1)

Analog to (2.5), the state value of an MDP can be decomposed into a Bellman notation:

$$v_\pi(x_k) = \mathbb{E}_\pi \left[R_{k+1} + \gamma v_\pi(X_{k+1}) | X_k = x_k \right] . \tag{2.13}$$

In finite MDPs, the state value can be directly linked to the action value (cf. Fig. 2.8):

$$v_\pi(x_k) = \sum_{u_k \in \mathcal{U}} \pi(u_k|x_k) q_\pi(x_k, u_k) . \tag{2.14}$$

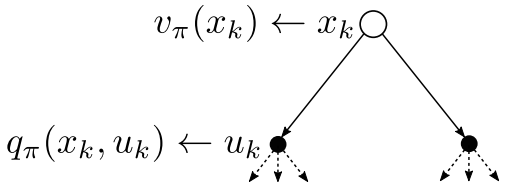


Fig. 2.8: Backup diagram for $v_\pi(x_k)$

Bellman expectation equation (2)

Likewise, the action value of an MDP can be decomposed into a Bellman notation:

$$q_\pi(x_k, u_k) = \mathbb{E}_\pi \left[R_{k+1} + \gamma q_\pi(X_{k+1}, U_{k+1}) | X_k = x_k, U_k = u_k \right] . \tag{2.15}$$

In finite MDPs, the action value can be directly linked to the state value (cf. Fig. 2.9):

$$q_\pi(x_k, u_k) = \mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u v_\pi(x_{k+1}) . \tag{2.16}$$

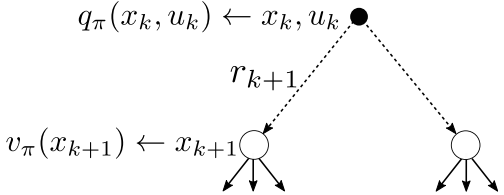


Fig. 2.9: Backup diagram for $q_\pi(x_k, u_k)$

Bellman expectation equation (3)

Inserting (2.16) into (2.14) directly results in:

$$v_\pi(x_k) = \sum_{u_k \in \mathcal{U}} \pi(u_k|x_k) \left(\mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u v_\pi(x_{k+1}) \right) . \tag{2.17}$$

Conversely, the action value becomes:

$$q_\pi(x_k, u_k) = \mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u \left(\sum_{u_{k+1} \in \mathcal{U}} \pi(u_{k+1}|x_{k+1}) q_\pi(x_{k+1}, u_{k+1}) \right) . \tag{2.18}$$

Bellman expectation equation in matrix form

Given a policy π and following the same assumptions as for (2.8), the Bellman expectation equation can be expressed in matrix form:

$$\begin{aligned} \boldsymbol{v}_{\mathcal{X}}^{\pi} &= \boldsymbol{r}_{\mathcal{X}}^{\pi} + \gamma \boldsymbol{\mathcal{P}}_{xx'}^{\pi} \boldsymbol{v}_{\mathcal{X}}^{\pi}, \\ \begin{bmatrix} v_1^{\pi} \\ \vdots \\ v_n^{\pi} \end{bmatrix} &= \begin{bmatrix} \mathcal{R}_1^{\pi} \\ \vdots \\ \mathcal{R}_n^{\pi} \end{bmatrix} + \gamma \begin{bmatrix} p_{11}^{\pi} & \cdots & p_{1n}^{\pi} \\ \vdots & & \vdots \\ p_{n1}^{\pi} & \cdots & p_{nn}^{\pi} \end{bmatrix} \begin{bmatrix} v_1^{\pi} \\ \vdots \\ v_n^{\pi} \end{bmatrix}. \end{aligned} \tag{2.19}$$

Here, $\boldsymbol{r}_{\mathcal{X}}^{\pi}$ and $\boldsymbol{\mathcal{P}}_{xx'}^{\pi}$ are the rewards and state transition probability following policy π. Hence, the state value can be calculated by solving (2.19) for $\boldsymbol{v}_{\mathcal{X}}^{\pi}$, e.g., by direct matrix inversion:

$$\boldsymbol{v}_{\mathcal{X}}^{\pi} = \left(\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}_{xx'}^{\pi}\right)^{-1} \boldsymbol{r}_{\mathcal{X}}^{\pi}. \tag{2.20}$$

Bellman expectation equation & forest tree example (1)

Let's assume following very simple policy ('*fifty-fifty*')

$$\pi(u = \mathsf{cut}|x) = 0.5, \quad \pi(u = \mathsf{wait}|x) = 0.5 \quad \forall x \in \mathcal{X}\,.$$

Applied to the given environment behavior

$$\boldsymbol{\mathcal{P}}_{xx'}^{u=c} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{\mathcal{P}}_{xx'}^{u=w} = \begin{bmatrix} 0 & 1-\alpha & 0 & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 1-\alpha & \alpha \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\boldsymbol{r}_{\mathcal{X}}^{u=c} = \begin{bmatrix} 1 & 2 & 3 & 0 \end{bmatrix}^{\mathsf{T}}, \quad \boldsymbol{r}_{\mathcal{X}}^{u=w} = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^{\mathsf{T}},$$

one receives:

$$\boldsymbol{\mathcal{P}}_{xx'}^{\pi} = \begin{bmatrix} 0 & \frac{1-\alpha}{2} & 0 & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{r}_{\mathcal{X}}^{\pi} = \begin{bmatrix} 0.5 \\ 1 \\ 2 \\ 0 \end{bmatrix}.$$

Bellman expectation equation & forest tree example (2)

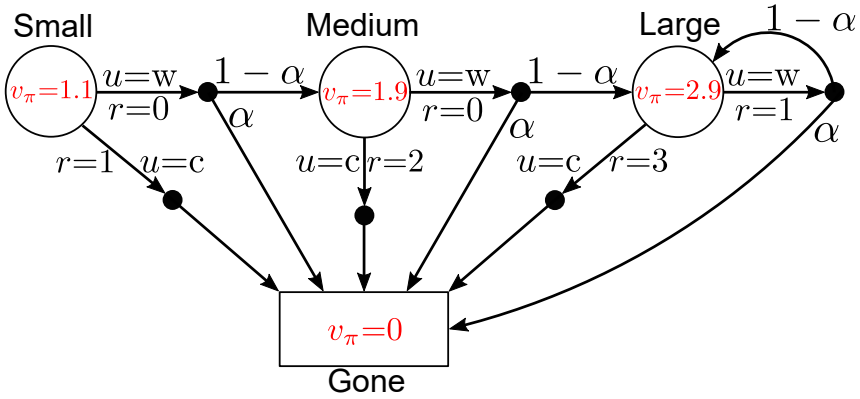


Fig. 2.10: Forest MDP with fifty-fifty policy including state values

- Discount factor $\gamma = 0.8$
- Disaster probability $\alpha = 0.2$

Bellman expectation equation & forest tree example (3)

Using the Bellman expectation eq. (2.16) the action values can be directly calculated:

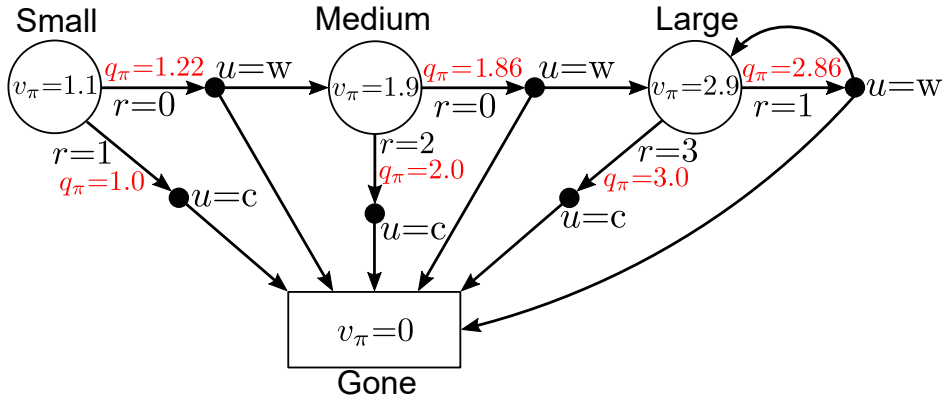


Fig. 2.11: Forest MDP with fifty-fifty policy plus action values

Table of contents

Optimal value functions in an MDP

Definition 2.9: Optimal state-value function

The optimal state-value function of an MDP is the maximum state-value function over all polices:

$$v^*(x) = \max_\pi v_\pi(x)\,. \tag{2.21}$$

Definition 2.10: Optimal action-value function

The optimal action-value function of an MDP is the maximum action-value function over all polices:

$$q^*(x,u) = \max_\pi q_\pi(x,u)\,. \tag{2.22}$$

- The optimal value function denotes the best possible agent's performance for a given MDP / environment.
- A (finite) MDP can be easily solved in an optimal way if $q^*(x,u)$ is known.

Optimal policy in an MDP

Define a partial ordering over polices

$$\pi \geq \pi' \quad \text{if} \quad v_\pi(x) \geq v_{\pi'}(x) \quad \forall x \in \mathcal{X}\,. \tag{2.23}$$

Theorem 2.1: Optimal policies in MDPs

For any finite MDP

- there exists an optimal policy $\pi^* \geq \pi$ that is better or equal to all other policies,
- all optimal policies achieve the same optimal state-value function $v^*(x) = v_{\pi^*}(x)$,
- all optimal policies achieve the same optimal action-value function $q^*(x, u) = q_{\pi^*}(x, u)$.

Bellman optimality equation (1)

Theorem 2.2: Bellman's principle of optimality

"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." (R.E. Bellman, Dynamic Programming, 1957)

- Any policy (i.e., also the optimal one) must satisfy the self-consistency condition given by the Bellman expectation equation.
- An optimal policy must deliver the maximum expected return being in a given state:

$$\begin{aligned} v^*(x_k) &= \max_u q_{\pi^*}(x_k, u)\,, \\ &= \max_u \mathbb{E}_{\pi^*}\left[G_k | X_k = x_k, U = u\right]\,, \\ &= \max_u \mathbb{E}_{\pi^*}\left[R_{k+1} + \gamma G_{k+1} | X_k = x_k, U = u\right]\,, \\ &= \max_u \mathbb{E}\left[R_{k+1} + \gamma v^*(X_{k+1}) | X_k = x_k, U = u\right]\,. \end{aligned} \tag{2.24}$$

Bellman optimality equation (2)

Again, the Bellman optimality equation can be visualized by a backup diagram:

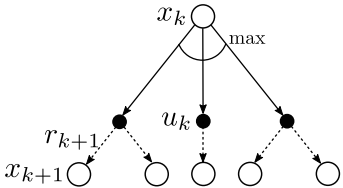


Fig. 2.12: Backup diagram for $v^*(x_k)$

For a finite MDP, the following expression results:

$$v^*(x_k) = \max_{u_k} \mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u v_{\pi^*}(x_{k+1}). \tag{2.25}$$

Bellman optimality equation (3)

Likewise, the Bellman optimality equation is applicable to the action value:

$$q^*(x_k, u_k) = \mathbb{E}\left[R_{k+1} + \gamma \max_{u_{k+1}} q^*(X_{k+1}, U_{k+1}) | X_k = x_k, U_k = u_k\right]. \tag{2.26}$$

And, in the finite MDP case:

$$q^*(x_k, u_k) = \mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u \max_{u_{k+1}} q^*(x_{k+1}, u_{k+1}). \tag{2.27}$$

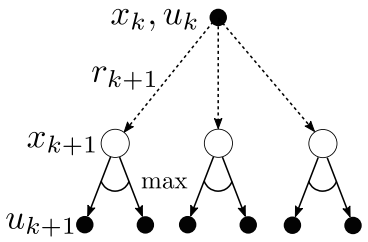


Fig. 2.13: Backup diagram for $q^*(x_k, u_k)$

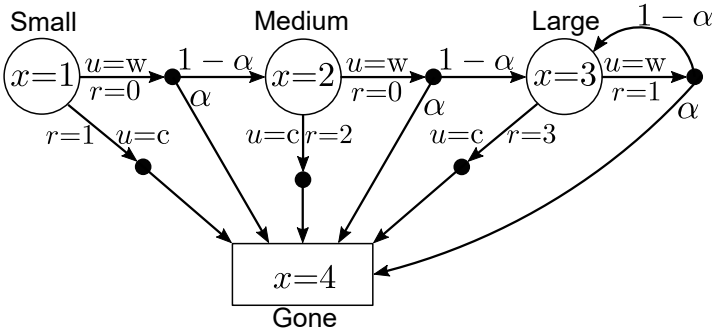
Solving the Bellman optimality equation

- In finite MDPs with n states, (2.25) delivers an algebraic equation system with n unknowns and n state-value equations.
- Likewise, (2.27) delivers an algebraic equation system with up to $n \cdot m$ unknowns and $n \cdot m$ action-value equations (m =number of actions).

- If environment is exactly known, solving for v^* or q^* directly delivers optimal policy.
  - If $v(x)$ is known, a one-step-ahead search is required to get $q(x, u)$.
  - If $q(x, u)$ is known, directly choose q^*.
- Even though above decisions are very short sighted (one-step-ahead search for v or direct choice of q), by Bellman's principle of optimality one receives the long-term maximum of the expected reward.

Optimal policy for forest tree MDP

Remember the forest tree MDP example:



- Two actions possible in each state:
  - Wait $u = w$: let the tree grow.
  - Cut $u = c$: gather the wood.
- Lets first calculate $v^*(x)$ and then $q^*(x, u)$.

Optimal policy for forest tree MDP: state value (1)

Start with $v(x=4)$ ('gone') and then continue going backwards:

$$
\begin{aligned}
v^*(x=4) &= 0\,, \\
v^*(x=3) &= \max \begin{cases} 1+\gamma\left[(1-\alpha)v^*(x=3)+\alpha v^*(x=4)\right]\,, \\ 3+\gamma v^*(x=4)\,, \end{cases} \\
&= \max \begin{cases} 1+\gamma\left[(1-\alpha)v^*(x=3)\right]\,, \\ 3\,, \end{cases} \\
v^*(x=2) &= \max \begin{cases} 0+\gamma\left[(1-\alpha)v^*(x=3)+\alpha v^*(x=4)\right]\,, \\ 2+\gamma v^*(x=4)\,, \end{cases} \\
&= \max \begin{cases} \gamma\left[(1-\alpha)v^*(x=3)\right]\,, \\ 2\,, \end{cases} \\
v^*(x=1) &= \max \begin{cases} \gamma\left[(1-\alpha)v^*(x=2)\right]\,, \\ 1\,. \end{cases}
\end{aligned}
$$

Optimal policy for forest tree MDP: state value (2)

- Possible solutions:
  - numerical optimization approach (e.g., simplex method, gradient descent,...)
  - manual case-by-case equation solving (dynamic programming, cf. next lecture)

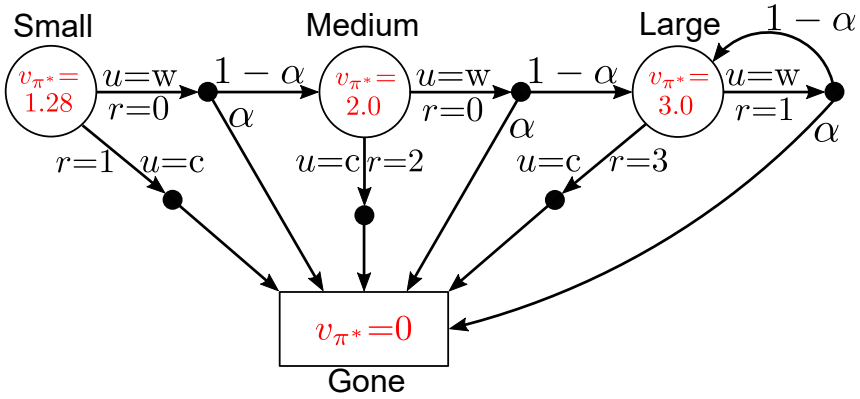


Fig. 2.14: State values under optimal policy ($\gamma = 0.8$, $\alpha = 0.2$)

Optimal policy for forest tree MDP: state value (3)

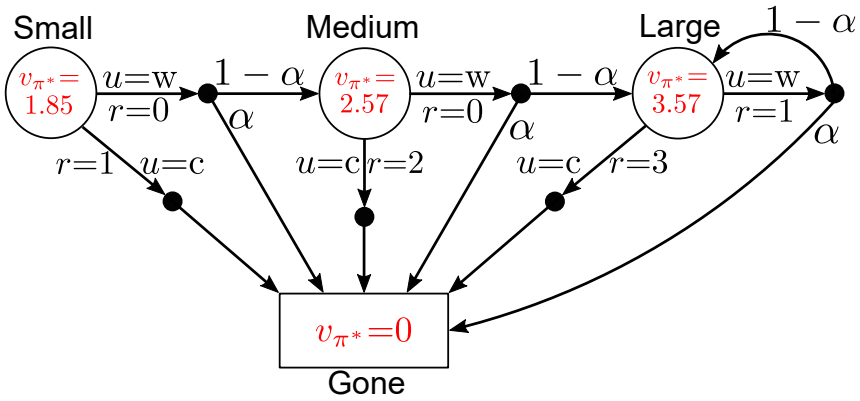


Fig. 2.15: State values under optimal policy ($\gamma = 0.9$, $\alpha = 0.2$)

Optimal policy for forest tree MDP: action value (1)

Use $u_{k+1} = u'$ to set up equation system:

$$
\begin{aligned}
q^*(x = 1, u = c) &= 1\,, \\
q^*(x = 1, u = w) &= \gamma(1 - \alpha) \max_{u'} q^*(x = 2, u')\,, \\
q^*(x = 2, u = c) &= 2\,, \\
q^*(x = 2, u = w) &= \gamma(1 - \alpha) \max_{u'} q^*(x = 3, u')\,, \\
q^*(x = 3, u = c) &= 3\,, \\
q^*(x = 3, u = w) &= 1 + \gamma(1 - \alpha) \max_{u'} q^*(x = 3, u')\,.
\end{aligned}
$$

- There are six action-state pairs in total.
- Three of them can be directly determined.
- Three unknowns and three equations remain.

Optimal policy for forest tree MDP: action value (2)

Rearrange $\max$ expressions for unknown action values:

$$q^*(x=1,u=w) = \gamma(1-\alpha)\max\begin{cases}\gamma(1-\alpha)\max\begin{cases}1+\gamma(1-\alpha)q^*(3,w),\\ 3,\end{cases}\\ 2,\end{cases}$$

$$q^*(x=2,u=w) = \gamma(1-\alpha)\max\begin{cases}1+\gamma(1-\alpha)q^*(3,w),\\ 3,\end{cases}$$

$$q^*(x=3,u=w) = 1+\gamma(1-\alpha)\max\begin{cases}q^*(3,w),\\ 3.\end{cases}$$

Again, retrieve unknown optimal action values by numerical optimization solvers or manual backwards calculation (dynamic programming).

Optimal policy for forest tree MDP: action value (3)

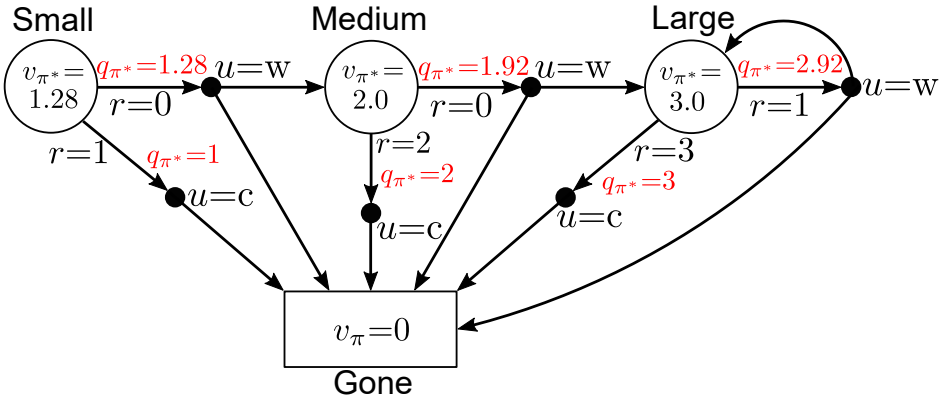


Fig. 2.16: Action values under optimal policy ($\gamma = 0.8$, $\alpha = 0.2$)

Optimal policy for forest tree MDP: action value (4)

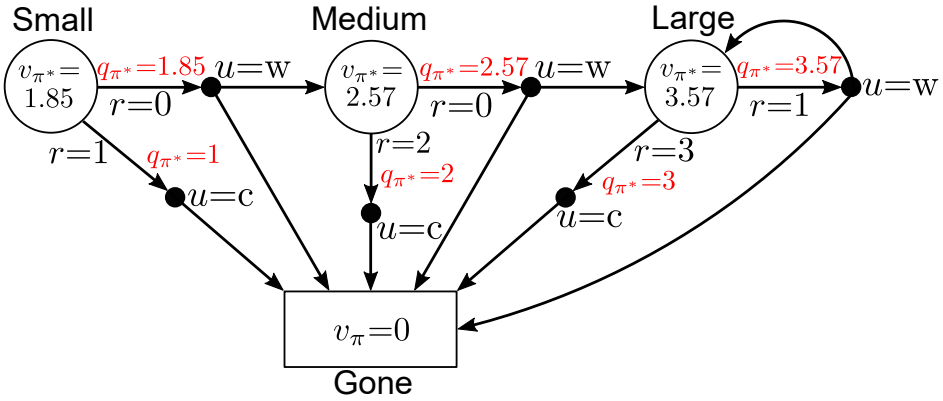


Fig. 2.17: Action values under optimal policy ($\gamma = 0.9$, $\alpha = 0.2$)

Direct numerical state and action-value calculation

- Possible only for small action and state-space MDPs
  - 'Solving' Backgammon with $\approx 10^{20}$ states?
- Another issue: total environment knowledge required

Framing the reinforcement learning problem

Facing the above issues, RL addresses mainly two topics:

- Approximate solutions of complex decision problems.
- Learning of such approximations based on data retrieved from environment interactions potentially without any a priori model knowledge.

Summary: what you've learned in this lecture

- Differentiate finite Markov process models with or w/o rewards and actions.
- Interpret such stochastic processes as simplified abstractions of real-world problems.
- Understand the importance of value functions to describe the agent's performance.
- Formulate value-function equation systems by the Bellman principle.
- Recognize optimal policies.
- Setting up nonlinear equation systems for retrieving optimal policies by the Bellman principle.
- Solving for different value functions in MRP/MDP by brute force optimization.

Lecture 03: Dynamic Programming

André Bodmer

Table of contents

What is dynamic programming (DP)?

Basic DP definition

- Dynamic: sequential or temporal problem structure
- Programming: mathematical optimization, i.e., numerical solutions

Further characteristics:

- DP is a collection of algorithms to solve MDPs and neighboring problems.
  - We will focus only on finite MDPs.
  - In case of continuous action/state space: apply quantization.
- Use of value functions to organize and structure the search for an optimal policy.
- Breaks problems into subproblems and solves them.

Requirements for DP

DP can be applied to problems with the following characteristics.

- Optimal substructure:
  - Principle of optimality applies.
  - Optimal solution can be derived from subproblems.
- Overlapping subproblems:
  - Subproblems recur many times.
  - Hence, solutions can be cached and reused.

How is that connected to MDPs?

- MDPs satisfy above's properties:
  - Bellman equation provides recursive decomposition.
  - Value function stores and reuses solutions.

Example: DP vs. exhaustive search (1)

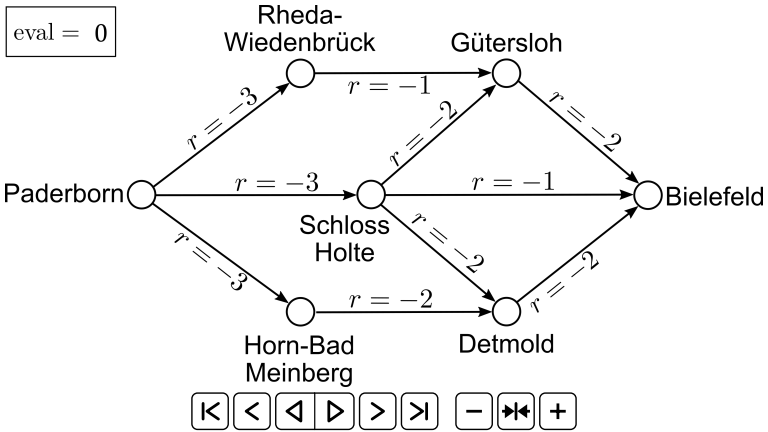


Fig. 3.1: Shortest path problem to travel from Paderborn to Bielefeld: Eshaustive search requires 14 travel segment evaluations since every possible travel route is evaluated independently.

Example: DP vs. exhaustive search (2)

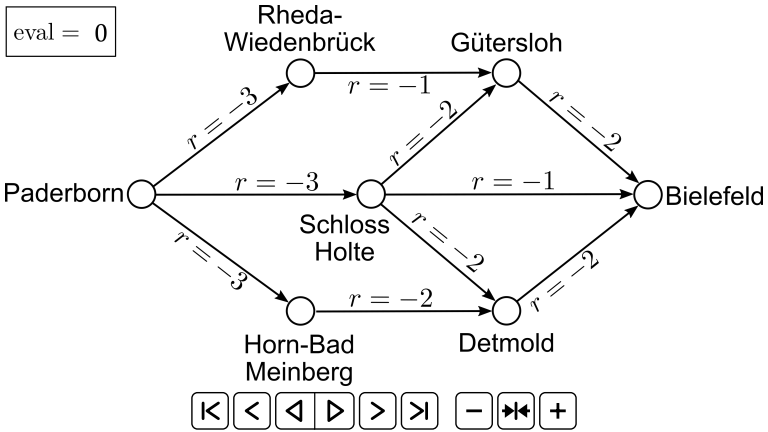


Fig. 3.2: Shortest path problem to travel from Paderborn to Bielefeld: DP requires only 10 travel segment evaluations in order to calculate the optimal travel policy due to the reuse of subproblem results.

Utility of DP in the RL context

DP is used for iterative model-based prediction and control in an MDP.

- Prediction:
  - Input: MDP $\langle \mathcal{X}, \mathcal{U}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ and policy π
  - Output: (estimated) value function $\hat{v}_\pi \approx v_\pi$
- Control:
  - Input: MDP $\langle \mathcal{X}, \mathcal{U}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$
  - Output: (estimated) optimal value function $\hat{v}^*_\pi \approx v^*_\pi$ or policy $\hat{\pi}^* \approx \pi^*$

In both applications DP requires full knowledge of the MDP structure.

- Feasibility in real-world engineering applications (model vs. system) is therefore limited.
- But: following DP concepts are largely used in modern data-driven RL algorithms.

Table of contents

Policy evaluation background (1)

- Problem: evaluate a given policy π to predict v_π.
- Recap: Bellman expectation equation for $x_k \in \mathcal{X}$ is given as

$$\begin{aligned} v_\pi(x_k) &= \mathbb{E}_\pi \left[G_k | X_k = x_k \right], \\ &= \mathbb{E}_\pi \left[R_{k+1} + \gamma G_{k+1} | X_k = x_k \right], \\ &= \mathbb{E}_\pi \left[R_{k+1} + \gamma v_\pi(X_{k+1}) | X_k = x_k \right]. \end{aligned}$$

- Or in matrix form:

$$\boldsymbol{v}_{\mathcal{X}}^\pi = \boldsymbol{r}_{\mathcal{X}}^\pi + \gamma \boldsymbol{\mathcal{P}}_{xx'}^\pi \boldsymbol{v}_{\mathcal{X}}^\pi,$$

$$\begin{bmatrix} v_1^\pi \\ \vdots \\ v_n^\pi \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1^\pi \\ \vdots \\ \mathcal{R}_n^\pi \end{bmatrix} + \gamma \begin{bmatrix} p_{11}^\pi & \cdots & p_{1n}^\pi \\ \vdots & & \vdots \\ p_{n1}^\pi & \cdots & p_{nn}^\pi \end{bmatrix} \begin{bmatrix} v_1^\pi \\ \vdots \\ v_n^\pi \end{bmatrix}.$$

- Solving the Bellman expectation equation for v_π requires handling a linear equation system with n unknowns (i.e., number of states).

Policy evaluation background (2)

- Problem: directly calculating v_π is numerically costly for high-dimensional state spaces (e.g., by matrix inversion).
- General idea: apply iterative approximations $\hat{v}_i(x_k) = v_i(x_k)$ of $v_\pi(x_k)$ with decreasing errors:

$$\|v_i(x_k) - v_\pi\|_\infty \to 0 \quad \text{for} \quad i = 1, 2, 3, \ldots \tag{3.1}$$

- The Bellman equation in matrix form can be rewritten as:

$$\underbrace{(\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}^\pi_{xx'})}_{\boldsymbol{A}} \underbrace{\boldsymbol{v}^\pi_{\mathcal{X}}}_{\boldsymbol{\zeta}} = \underbrace{\boldsymbol{r}^\pi_{\mathcal{X}}}_{\boldsymbol{b}} . \tag{3.2}$$

- To iteratively solve this linear equation $\boldsymbol{A\zeta} = \boldsymbol{b}$, one can apply numerous methods such as
  - General gradient descent,
  - Richardson iteration,
  - Kyrlov subspace methods.

Richardson iteration (1)

In the MDP context, the Richardson iteration became the default solution approach to iteratively solve:

$$\boldsymbol{A}\boldsymbol{\zeta} = \boldsymbol{b}.$$

The Richardson iteration is

$$\boldsymbol{\zeta}_{i+1} = \boldsymbol{\zeta}_i + \omega(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{\zeta}_i) \tag{3.3}$$

with ω being a scalar parameter that has to be chosen such that the sequence $\boldsymbol{\zeta}_i$ converges. To choose ω we inspect the series of approximation errors $\boldsymbol{e}_i = \boldsymbol{\zeta}_i - \boldsymbol{\zeta}$ and apply it to (3.3):

$$\boldsymbol{e}_{i+1} = \boldsymbol{e}_i - \omega\boldsymbol{A}\boldsymbol{e}_i = (\boldsymbol{I} - \omega\boldsymbol{A})\,\boldsymbol{e}_i. \tag{3.4}$$

To evaluate convergence we inspect the following norm:

$$\|\boldsymbol{e}_{i+1}\|_\infty = \|(\boldsymbol{I} - \omega\boldsymbol{A})\,\boldsymbol{e}_i\|_\infty\,. \tag{3.5}$$

Richardson iteration (2)

Since any induced matrix norm is sub-multiplicative, we can approximate (3.5) by the inequality:

$$\|\boldsymbol{e}_{i+1}\|_\infty \leq \|(\boldsymbol{I} - \omega \boldsymbol{A})\|_\infty \|\boldsymbol{e}_i\|_\infty . \tag{3.6}$$

Hence, the series converges if

$$\|(\boldsymbol{I} - \omega \boldsymbol{A})\|_\infty < 1. \tag{3.7}$$

Inserting from (3.2) leads to:

$$\|(\boldsymbol{I}(1-\omega) + \omega\gamma \boldsymbol{\mathcal{P}}^\pi_{xx'})\|_\infty < 1. \tag{3.8}$$

For $\omega = 1$ we receive:

$$\gamma \|(\boldsymbol{\mathcal{P}}^\pi_{xx'})\|_\infty < 1. \tag{3.9}$$

Since the row elements of $\boldsymbol{\mathcal{P}}^\pi_{xx'}$ always sum up to 1,

$$\gamma < 1 \tag{3.10}$$

follows. Hence, when discounting the Richardson iteration always converges for MDPs even if we assume $\omega = 1$.

Iterative policy evaluation by Richardson iteration (1)

Applying the Richardson iteration (3.3) with $w = 1$ to the Bellman equation (2.17) for any $x_k \in \mathcal{X}$ at iteration i results in:

$$v_{i+1}(x_k) = \sum_{u_k \in \mathcal{U}} \boldsymbol{\pi}(u_k|x_k) \left(\mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u v_i(x_{k+1}) \right) . \tag{3.11}$$

Matrix form based on (2.19) then is:

$$\boldsymbol{v}_{\mathcal{X},i+1}^{\pi} = \boldsymbol{r}_{\mathcal{X}}^{\pi} + \gamma \boldsymbol{\mathcal{P}}_{xx'}^{\pi} \boldsymbol{v}_{\mathcal{X},i}^{\pi} . \tag{3.12}$$

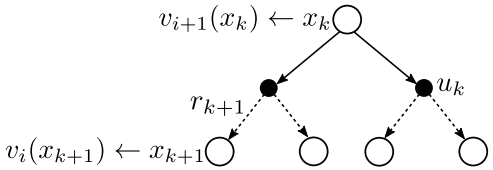


Fig. 3.3: Backup diagram for iterative policy evaluation

Iterative policy evaluation by Richardson iteration (2)

- During one Richardson iteration the 'old' value of x_k is replaced with a 'new' value from the 'old' values of the successor state x_{k+1}.
  - Update $v_{i+1}(x_k)$ from $v_i(x_{k+1})$, see Fig. 3.3.
  - Updating estimates (v_{i+1}) on the basis of other estimates (v_i) is often called bootstrapping.
- The Richardson iteration can be interpreted as a gradient descent algorithm for solving (3.2).
- This leads to synchronous, full backups of the entire state space $\mathcal{X}$.
- Also called expected update because it is based on the expectation over all possible next states (utilizing full model knowledge).
- In subsequent lectures, the expected update will be supplemented by data-driven samples from the environment.

Iterative policy evaluation example: forest tree MDP

Let's reuse the forest tree MDP example from Fig. 2.10 with *fifty-fifty policy*:

$$\boldsymbol{\mathcal{P}}^{\pi}_{xx'} = \begin{bmatrix} 0 & \frac{1-\alpha}{2} & 0 & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & \frac{1-\alpha}{2} & \frac{1+\alpha}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{r}^{\pi}_{\mathcal{X}} = \begin{bmatrix} 0.5 \\ 1 \\ 2 \\ 0 \end{bmatrix}.$$

i	$v_i(x=1)$	$v_i(x=2)$	$v_i(x=3)$	$v_i(x=4)$
0	0	0	0	0
1	0.5	1	2	0
2	0.82	1.64	2.64	0
3	1.03	1.85	2.85	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
∞	1.12	1.94	2.94	0

Tab. 3.1: Policy evaluation by Richardson iteration (3.12) for forest tree MDP with $\gamma = 0.8$ and $\alpha = 0.2$

Variant: in-place updates

Instead of applying (3.12) to the entire vector $\boldsymbol{v}^{\pi}_{\mathcal{X},i+1}$ in 'one shot' (synchronous backup), an elementwise in-place version of the policy evaluation can be carried out:

input: full model of the MDP, i.e., $\langle \mathcal{X}, \mathcal{U}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$ including policy π
parameter: $\delta > 0$ as accuracy termination threshold
init: $v_0(x) \, \forall \, x \in \mathcal{X}$ arbitrary except $v_0(x) = 0$ if x is terminal
repeat
  $\Delta \leftarrow 0$;
  for $\forall \, x_k \in \mathcal{X}$ **do**
    $\tilde{v} \leftarrow \hat{v}(x_k)$;
    $\hat{v}(x_k) \leftarrow \sum_{u_k \in \mathcal{U}} \pi(u_k|x_k) \left(\mathcal{R}^u_x + \gamma \sum_{x_{k+1} \in \mathcal{X}} p^u_{xx'} \hat{v}(x_{k+1}) \right)$;
    $\Delta \leftarrow \max \left(\Delta, |\tilde{v} - \hat{v}(x_k)| \right)$;
until $\Delta < \delta$;

Algo. 3.1: Iterative policy evaluation using in-place updates (output: estimate of $\boldsymbol{v}^{\pi}_{\mathcal{X}}$)

In-place policy evaluation updates for forest tree MDP

- In-place algorithms allow to update states in a beneficial order.
- May converge faster than regular Richardson iteration if state update order is chosen wisely (sweep through state space).
- For forest tree MDP: reverse order, i.e., start with $x = 4$.
- As can be seen in Tab. 3.2 the in-place updates especially converge faster for the 'early states'.

i	$v_i(x=1)$	$v_i(x=2)$	$v_i(x=3)$	$v_i(x=4)$
0	0	0	0	0
1	1.03	1.64	2	0
2	1.09	1.85	2.64	0
3	1.11	1.91	2.85	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
∞	1.12	1.94	2.94	0

Tab. 3.2: In-place updates for forest tree MDP

Table of contents

General idea on policy improvement

- If we know v_π of a given MDP, how to improve the policy?
- The simple idea of policy improvement is:
  - Consider a new (non-policy conform) action $u \neq \pi(x_k)$.
  - Follow thereafter the current policy π.
  - Check the action value of this 'new move'. If it is better than the 'old' value, take it:

$$q_\pi(x_k, u_k) = \mathbb{E}\left[R_{k+1} + \gamma v_\pi(X_{k+1}) | X_k = x_k, U_k = u_k\right] . \tag{3.13}$$

Theorem 3.1: Policy improvement

If for any deterministic policy pair π and π'

$$q_\pi(x, \pi'(x)) \geq v_\pi(x) \quad \forall x \in \mathcal{X} \tag{3.14}$$

applies, then the policy π' must be as good as or better than π. Hence, it obtains greater or equal expected return

$$v_{\pi'}(x) \geq v_\pi(x) \quad \forall x \in \mathcal{X}. \tag{3.15}$$

Proof of policy improvement theorem

Start with (3.14) and recursively reapply (3.13):

$$
\begin{aligned}
v_\pi(x_k) &\leq q_\pi(x_k, \pi'(x_k)), \\
&= \mathbb{E}\left[R_{k+1} + \gamma v_\pi(X_{k+1}) | X_k = x_k, U_k = \pi'(x_k)\right], \\
&= \mathbb{E}_{\pi'}\left[R_{k+1} + \gamma v_\pi(X_{k+1}) | X_k = x_k\right], \\
&\leq \mathbb{E}_{\pi'}\left[R_{k+1} + \gamma q_\pi(x_{k+1}, \pi'(x_{k+1})) | X_k = x_k\right], \\
&= \mathbb{E}_{\pi'}\left[R_{k+1} + \gamma \mathbb{E}_{\pi'}\left[R_{k+2} + \gamma v_\pi(X_{k+2}) | X_{k+1}, \pi'(x_{k+1})\right] | X_k = x_k\right], \\
&= \mathbb{E}_{\pi'}\left[R_{k+1} + \gamma R_{k+2} + \gamma^2 v_\pi(X_{k+2}) | X_k = x_k\right], \\
&\leq \mathbb{E}_{\pi'}\left[R_{k+1} + \gamma R_{k+2} + \gamma^2 R_{k+3} + \gamma^3 v_\pi(X_{k+3}) | X_k = x_k\right], \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}\left[R_{k+1} + \gamma R_{k+2} + \gamma^2 R_{k+3} + \gamma^3 R_{k+4} + \cdots | X_k = x_k\right], \\
&= v_{\pi'}(x_k).
\end{aligned}
\tag{3.16}
$$

Greedy policy improvement (1)

- So far, policy improvement addressed only changing the policy at a single state.
- Now, extend this scheme to all states by selecting the best action according to $q_\pi(x_k, u_k)$ in every state (greedy policy improvement):

$$\begin{aligned}\pi'(x_k) &= \arg\max_{u_k \in \mathcal{U}} q_\pi(x_k, u_k), \\ &= \arg\max_{u_k \in \mathcal{U}} \mathbb{E}\left[R_{k+1} + \gamma v_\pi(X_{k+1}) | X_k = x_k, U_k = u_k\right], \\ &= \arg\max_{u_k \in \mathcal{U}} \mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u v_\pi(x_{k+1}).\end{aligned} \tag{3.17}$$

Greedy policy improvement (2)

- Each greedy policy improvement takes the best action in a one-step look-ahead search and, therefore, satisfies Theo. 3.1.
- If after a policy improvement step $v_\pi(x_k) = v_{\pi'}(x_k)$ applies, it follows:

$$\begin{aligned} v_{\pi'}(x_k) &= \max_{u_k \in \mathcal{U}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi'}(X_{k+1}) | X_k = x_k, U_k = u_k\right], \\ &= \max_{u_k \in \mathcal{U}} \mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u v_{\pi'}(x_{k+1}). \end{aligned} \tag{3.18}$$

- This is the Bellman optimality equation, which guarantees that $\pi' = \pi$ must be optimal policies.
- Although proof for policy improvement theorem was presented for deterministic policies, transfer to stochastic policies $\pi(u_k|x_k)$ is possible.
- Takeaway message: policy improvement theorem guarantees finding optimal policies in finite MDPs (e.g., by DP).

Table of contents

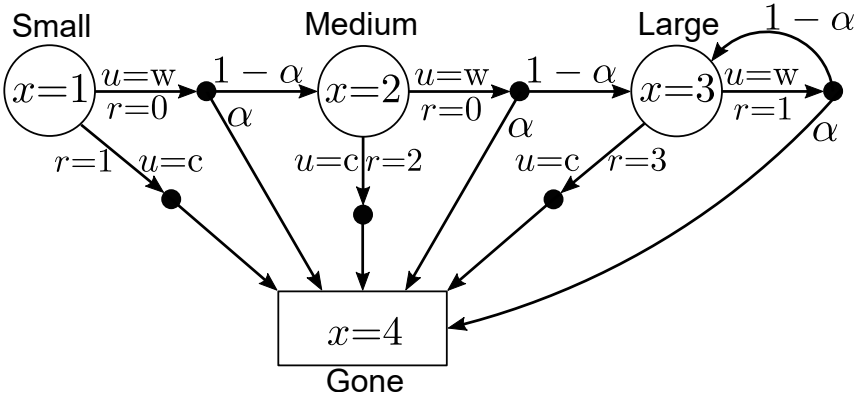
Concept of policy iteration

- Policy iteration combines the previous policy evaluation and policy improvement in an iterative sequence:

$$\pi_0 \rightarrow v_{\pi_0} \rightarrow \pi_1 \rightarrow v_{\pi_1} \rightarrow \cdots \pi^* \rightarrow v_{\pi^*} \tag{3.19}$$

- Evaluate $\rightarrow$ improve $\rightarrow$ evaluate $\rightarrow$ improve ...
- In the 'classic' policy iteration, each policy evaluation step in (3.19) is fully executed, i.e., for each policy π_i an exact estimate of v_{π_i} is provided either by iterative policy evaluation with a sufficiently high number of steps or by any other method that fully solves (3.2).

Policy iteration example: forest tree MDP (1)



- Two actions possible in each state:
  - Wait $u = w$: let the tree grow.
  - Cut $u = c$: gather the wood.

Policy iteration example: forest tree MDP (2)

Assume $\alpha = 0.2$ and $\gamma = 0.8$ and start with 'tree hater' initial policy:

1. $\pi_0 = \pi(u_k = \mathsf{c}|x_k) \quad \forall x_k \in \mathcal{X}$.
2. Policy evaluation: $v_{\mathcal{X}}^{\pi_0} = \begin{bmatrix} 1 & 2 & 3 & 0 \end{bmatrix}^{\mathsf{T}}$
3. Greedy policy improvement:

$$\begin{aligned} \pi_1(x_k) &= \arg\max_{u_k \in \mathcal{U}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi_0}(X_{k+1}) | X_k = x_k, U_k = u_k\right], \\ &= \{\pi(u_k = \mathsf{w}|x_k = 1), \pi(u_k = \mathsf{c}|x_k = 2), \pi(u_k = \mathsf{c}|x_k = 3)\} \end{aligned}$$

4. Policy evaluation: $v_{\mathcal{X}}^{\pi_1} = \begin{bmatrix} 1.28 & 2 & 3 & 0 \end{bmatrix}^{\mathsf{T}}$
5. Greedy policy improvement:

$$\begin{aligned} \pi_2(x_k) &= \arg\max_{u_k \in \mathcal{U}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi_1}(X_{k+1}) | X_k = x_k, U_k = u_k\right], \\ &= \{\pi(u_k = \mathsf{w}|x_k = 1), \pi(u_k = \mathsf{c}|x_k = 2), \pi(u_k = \mathsf{c}|x_k = 3)\}, \\ &= \pi_1(x_k) \\ &= \pi^* \end{aligned}$$

Policy iteration example: forest tree MDP (3)

Assume $\alpha = 0.2$ and $\gamma = 0.8$ and start with 'tree lover' initial policy:

1. $\pi_0 = \pi(u_k = \mathsf{w}|x_k) \quad \forall x_k \in \mathcal{X}$.
2. Policy evaluation: $v_{\mathcal{X}}^{\pi_0} = \begin{bmatrix} 1.14 & 1.78 & 2.78 & 0 \end{bmatrix}^{\mathsf{T}}$
3. Greedy policy improvement:

$$\begin{aligned} \pi_1(x_k) &= \arg\max_{u_k \in \mathcal{U}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi_0}(X_{k+1}) | X_k = x_k, U_k = u_k\right], \\ &= \{\pi(u_k = \mathsf{w}|x_k = 1), \pi(u_k = \mathsf{c}|x_k = 2), \pi(u_k = \mathsf{c}|x_k = 3)\} \end{aligned}$$

4. Policy evaluation: $v_{\mathcal{X}}^{\pi_1} = \begin{bmatrix} 1.28 & 2 & 3 & 0 \end{bmatrix}^{\mathsf{T}}$
5. Greedy policy improvement:

$$\begin{aligned} \pi_2(x_k) &= \arg\max_{u_k \in \mathcal{U}} \mathbb{E}\left[R_{k+1} + \gamma v_{\pi_1}(X_{k+1}) | X_k = x_k, U_k = u_k\right], \\ &= \{\pi(u_k = \mathsf{w}|x_k = 1), \pi(u_k = \mathsf{c}|x_k = 2), \pi(u_k = \mathsf{c}|x_k = 3)\}, \\ &= \pi_1(x_k) \\ &= \pi^* \end{aligned}$$

Value iteration (1)

- Policy iteration involves full policy evaluation steps between policy improvements.
- In large state-space MDPs the full policy evaluation may be numerically very costly.

- Value iteration: One step iterative policy evaluation followed by policy improvement.
- Allows simple update rule which combines policy improvement with truncated policy evaluation in a single step:

$$
\begin{aligned}
v_{i+1}(x_k) &= \max_{u_k \in \mathcal{U}} \mathbb{E}\left[R_{k+1} + \gamma v_i(X_{k+1}) | X_k = x_k, U_k = u_k\right], \\
&= \max_{u_k \in \mathcal{U}} \mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u v_i(x_{k+1}) \,.
\end{aligned} \tag{3.20}
$$

Value iteration (2)

input: full model of the MDP, i.e., $\langle \mathcal{X}, \mathcal{U}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$
parameter: $\delta > 0$ as accuracy termination threshold
init: $v_0(x) \, \forall \, x \in \mathcal{X}$ arbitrary except $v_0(x) = 0$ if x is terminal
repeat
  $\Delta \leftarrow 0$;
  for $\forall \, x_k \in \mathcal{X}$ **do**
    $\tilde{v} \leftarrow \hat{v}(x_k)$;
    $\hat{v}(x_k) \leftarrow \max_{u_k \in \mathcal{U}} \left(\mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u \hat{v}(x_{k+1}) \right)$;
    $\Delta \leftarrow \max \left(\Delta, |\tilde{v} - \hat{v}(x_k)| \right)$;
until $\Delta < \delta$;
output: deterministic policy $\pi \approx \pi^*$, such that
$\pi(x_k) \leftarrow \arg\max_{u_k \in \mathcal{U}} \left(\mathcal{R}_x^u + \gamma \sum_{x_{k+1} \in \mathcal{X}} p_{xx'}^u \hat{v}(x_{k+1}) \right)$;

Algo. 3.2: Value iteration (note: compared to policy iteration, value iteration does not require an initial policy but only a state-value guess)

Value iteration example: forest tree MDP

- Assume again $\alpha = 0.2$ and $\gamma = 0.8$.
- Similar to in-place update policy evaluation, reverse order and start value iteration with $x = 4$.
- As shown in Tab. 3.3 value iteration converges in one step (for the given problem) to the optimal state value.

i	$v_i(x=1)$	$v_i(x=2)$	$v_i(x=3)$	$v_i(x=4)$
0	0	0	0	0
1	1.28	2	3	0
*	1.28	2	3	0

Tab. 3.3: Value iteration for forest tree MDP

Table of contents

Summarizing DP algorithms

- All DP algorithms are based on the state value $v(x)$.
  - Complexity is $\mathcal{O}(m \cdot n^2)$ for m actions and n states.
  - Evaluate all n^2 state transitions while considering up to m actions per state.
- Could be also applied to action values $q(x, u)$.
  - Complexity is inferior with $\mathcal{O}(m^2 \cdot n^2)$.
  - There are up to m^2 action values which require n^2 state transition evaluations each.

Problem	Relevant Equations	Algorithm
prediction	Bellman expectation eq.	policy evaluation
control	Bellman expectation eq. & greedy policy improvement	policy iteration
control	Bellman optimality eq.	value iteration

Tab. 3.4: Short overview addressing the treated DP algorithms

Curse of dimensionality

- DP is much more efficient than an exhaustive search over all n states and m actions in finite MDPs in order to find an optimal policy.
  - Exhaustive search for deterministic policies: m^n evaluations.
  - DP results in polynomial complexity regarding m and n.
- Nevertheless, DP uses full-width backups:
  - For each state update, every successor state and action is considered.
  - While utilizing full knowledge of the MDP structure.
- Hence, DP is can be effective up to medium-sized MDPs (i.e., million finite states)
- For large problems DP suffers from the curse of dimensionality:
  - Single update step may become computational infeasible.
  - Also: if continuous states need quantization, number of finite states n grows exponentially with the number of state variables (assuming fixed number of discretization levels).

Generalized policy iteration (GPI)

- Almost all RL methods are well-described as GPI.
- Push-pull: Improving the policy will deteriorate value estimation.
- Well balanced trade-off between evaluating and improving is required.

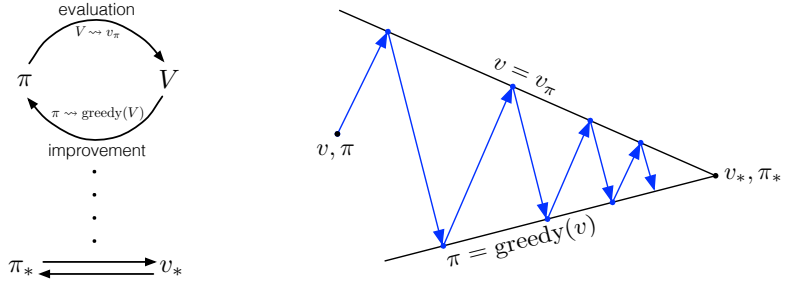


Fig. 3.4: Interpreting generalized policy iteration to switch back and forth between (arbitrary) evaluations and improvement steps (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Summary: what you've learned in this lecture

- DP is applicable for prediction and control problems in MDPs.
- But requires always full knowledge about the environment (i.e., it is a model-based solution).
- DP is more efficient than exhaustive search.
- But suffers from the curse of dimensionality for large MDPs.
- (Iterative) policy evaluations and (greedy) improvements solve MDPs.
- Both steps can be combined via value iteration.
- This idea of (generalized) policy iteration is a basic scheme of RL.
- Implementing DP algorithms comes with many degrees of freedom regarding the update order.

Lecture 04: Monte Carlo Methods

André Bodmer

Table of contents

Monte Carlo methods vs. dynamic programming

Dynamic programming:

- Model-based prediction and control
- Planning inside known MDPs

Monte Carlo methods:

- Model-free prediction and control
- Estimating value functions and optimize policies in unknown MDPs
- But: still assuming finite MDP problems (or problems close to that)
- In general: broad class of computational algorithms relying on repeated random sampling to obtain numerical results

General Monte Carlo (MC) methods' characteristics

- Learning from experience, i.e., sequences of samples $\langle x_k, u_k, r_{k+1} \rangle$
- Main concept: Estimation by averaging sample returns
- To guarantee well-defined returns: limited to episodic tasks
- Consequence: Estimation and policy updates only possible in an episode-by-episode way compared to step-by-step (online)

Fig. 4.1: Monte Carlo port
(source: www.flickr.com, by Miguel Mendez CC BY 2.0)

Table of contents

Task description and basic solution

MC prediction problem statement

- Estimate state value $v_\pi(x)$ for a given policy π.
- Available are samples $\langle x_{k,j}, u_{k,j}, r_{k+1,j} \rangle$ for episodes $j = 1, \ldots, J$.

MC solution approach:

- Average returns after visiting state x_k over episodes $j = 1, \ldots$

$$v_\pi(x_k) \approx \hat{v}_\pi(x_k) = \frac{1}{J} \sum_{j=1}^{J} g_{k,j} = \frac{1}{J} \sum_{j=1}^{J} \sum_{i=0}^{T_j} \gamma^i r_{k+i+1,j} \,. \tag{4.1}$$

- Above, T_j denotes the terminating time step of each episode j.
- First-visit MC: Apply (4.1) only to the first state visit per episode.
- Every-visit MC: Apply (4.1) each time visiting a certain state per episode (if a state is visited more than one time per episode).

Algorithmic implementation: MC-based prediction

input: a policy π to be evaluated
output: estimate of $\boldsymbol{v}_{\mathcal{X}}^{\pi}$ (i.e., value estimate for all states $x \in \mathcal{X}$)
init: $\hat{v}(x) \, \forall \, x \in \mathcal{X}$ arbitrary except $v_0(x) = 0$ if x is terminal
$l(x) \leftarrow$ an empty list for every $x \in \mathcal{X}$
for $j = 1, \ldots, J$ *episodes* **do**
Generate an episode following π: $x_0, u_0, r_1, \ldots, x_{T_j}, u_{T_j}, r_{T_j+1}$;
Set $g \leftarrow 0$;
for $k = T_j - 1, T_j - 2, T_j - 3, \ldots, 0$ *time steps* **do**
$g \leftarrow \gamma g + r_{k+1}$;
if $x_k \notin \langle x_0, x_1, \ldots, x_{k-1} \rangle$ **then**
Append g to list $l(x_k)$;
$\hat{v}(x_k) \leftarrow$ average($l(x_k)$);

Algo. 4.1: MC state-value prediction (first visit)

Incremental implementation

- Algo. 4.1 is inefficient due to large memory demand.
- Better: use incremental / recursive implementation.
- The sample mean $\mu_1, \mu_2, \ldots$ of an arbitrary sequence $g_1, g_2, \ldots$ is:

$$\begin{aligned} \mu_J &= \frac{1}{J}\sum_{i=1}^{J} g_i = \frac{1}{J}\left[g_J + \sum_{i=1}^{J-1} g_i\right] \\ &= \frac{1}{J}\left[g_J + (J-1)\mu_{J-1}\right] = \mu_{J-1} + \frac{1}{J}\left[g_J - \mu_{J-1}\right]. \end{aligned} \tag{4.2}$$

- If a given decision problem is non-stationary, using a forgetting factor $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$ allows for dynamic adaption:

$$\mu_J = \mu_{J-1} + \alpha\left[g_J - \mu_{J-1}\right]. \tag{4.3}$$

Statistical properties of MC-based prediction (1)

First-time visit MC:

- Each return sample g_J is independent from the others since they were drawn from separate episodes.
- One receives i.i.d. data to estimate $\mathbb{E}[\hat{v}_\pi]$ and consequently this is bias-free.
- The estimate's variance $\mathrm{Var}[\hat{v}_\pi]$ drops with $1/n$ (n: available samples).

Every-time visit MC:

- Each return sample g_J is not independent from the others since they might be obtained from same episodes.
- One receives non-i.i.d. data to estimate $\mathbb{E}[\hat{v}_\pi]$ and consequently this is biased for any $n < \infty$.
- Only in the limit $n \to \infty$ one receives $(v_\pi(x) - \mathbb{E}[\hat{v}_\pi(x)]) \to 0$.

More information: S. Singh and R. Sutton, "Reinforcement Learning with Replacing Eligibility Traces", Machine Learning, Vol. 22, pp. 123-158, 1996

Statistical properties of MC-based prediction (2)

- State-value estimates for each state are independent.
- One estimate does not rely on the estimate of other states (no bootstrapping compared to DP).
- Makes MC particularly attractive when one requires state-value knowledge of only one or few states.
  - Hence, generating episodes starting from the state of interest.

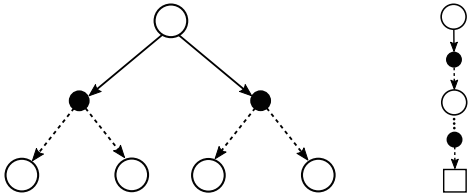


Fig. 4.2: Back-up diagrams for DP (left) and MC (right) prediction: shallow one-step back-ups with bootstrapping vs. deep back-ups over full epsiodes

MC-based prediction example: forest tree MDP (1)

Let's reuse the forest tree MDP example with *fifty-fifty policy* and discount factor $\gamma = 0.8$ plus disaster probability $\alpha = 0.2$:

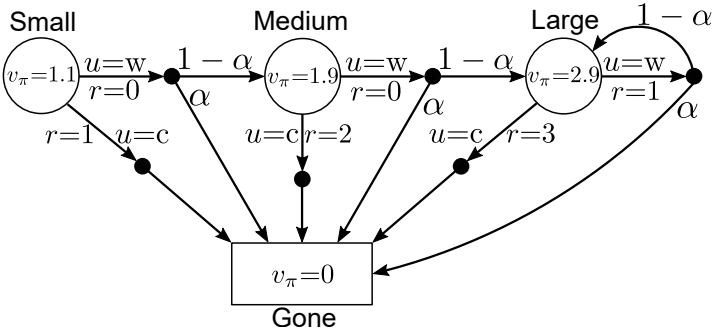


Fig. 4.3: Forest MDP with fifty-fifty-policy including state values

MC-based prediction example: forest tree MDP (2)

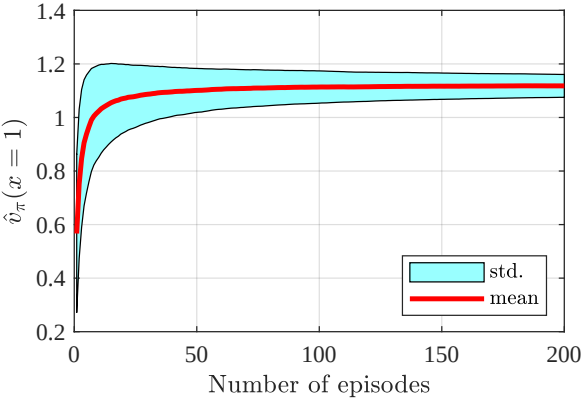


Fig. 4.4: State-value estimate of forest tree MDP initial state using MC-based prediction over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)

MC estimation of action values

Is a model available (i.e., tuple $\langle \mathcal{X}, \mathcal{U}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$)?

- Yes: state values plus one-step prediction deliver optimal policy.
- No: action values are very useful to directly obtain optimal choices.
- Recap policy improvement from last lecture.

Estimating $q_\pi(x, u)$ using MC approach:

- Analog to state values summarized in Algo. 4.1.
- Only small extension: a visit refers to a state-action pair (x, u).
- First-visit and every-visit variants exist.

Possible problem when following a deterministic policy π:

- Certain state-action pairs (x, u) are never visited.
- Missing degree of exploration.
- Workaround: exploring starts, i.e., starting episodes in random state-action pairs (x, u) and thereafter following π.

Table of contents

Applying generalized policy iteration (GPI) to MC control

GPI concept is directly applied to MC framework using action values:

$$\pi_0 \to \hat{q}_{\pi_0} \to \pi_1 \to \hat{q}_{\pi_1} \to \cdots \pi^* \to \hat{q}_{\pi^*} \,. \tag{4.4}$$

- Degree of freedom: Choose number of episodes to approximate $\hat{q}_{\pi_i}$.
- Policy improvement is done by greedy choices:

$$\pi(x) = \arg\max_u q(x,u) \quad \forall x \in \mathcal{X} \,. \tag{4.5}$$

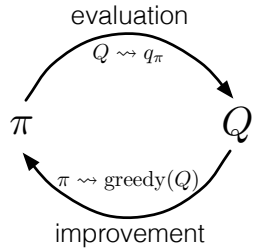


Fig. 4.5: Transferring GPI to MC-based control (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Policy improvement theorem

Assuming that one is operating in an unknown MDP, the policy improvement theorem Theo. 3.1 is still valid for MC-based control:

Policy improvement for MC-based control

$$
\begin{aligned}
q_{\pi_i}(x, \pi_{i+1}(x)) &= q_{\pi_i}(x, \arg\max_u q_{\pi_i}(x, u)), \\
&= \max_u q_{\pi_i}(x, u), \\
&\geq q_{\pi_i}(x, \pi_i(x)), \\
&\geq v_{\pi_i}(x).
\end{aligned}
\tag{4.6}
$$

- Each π_{i+1} is uniformly better or just as good (if optimal) as π_i.
- Assumption: All state-action pairs are evaluated due to sufficient exploration.
  - For example using exploring starts.

Algorithmic implementation: MC-based control

output: Optimal deterministic policy π^*
init: $\pi_{i=0}(x) \in \mathcal{U}$ arbitrarily $\forall x \in \mathcal{X}$
$\hat{q}(x,u)$ arbitrarily $\forall \{x \in \mathcal{X}, u \in \mathcal{U}\}$
$n(x,u) \leftarrow$ an empty list for state-action visits $\forall \{x \in \mathcal{X}, u \in \mathcal{U}\}$
repeat
$i \leftarrow i+1$;
Choose $\{x_0, u_0\}$ randomly such that all pairs have probability > 0 ;
Generate an episode from $\{x_0, u_0\}$ following π_i until termination step T_i;
Set $g \leftarrow 0$;
for $k = T_i - 1, T_i - 2, T_i - 3, \ldots, 0$ *time steps* **do**
$g \leftarrow \gamma g + r_{k+1}$;
if $\{x_k, u_k\} \notin \langle \{x_0, u_0\}, \ldots, \{x_{k-1}, u_{k-1}\} \rangle$ **then**
$n(x_k, u_k) \leftarrow n(x_k, u_k) + 1$;
$\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + 1/n(x_k, u_k) \cdot (g - \hat{q}(x_k, u_k))$;
$\pi_i(x_k) \leftarrow \arg\max_u \ \hat{q}(x_k, u)$;
until $\pi_{i+1} = \pi_i$;

Algo. 4.2: MC-based control using exploring starts (first visit)

Table of contents

Off- and on-policy learning

- On-policy learning
  - Evaluate or improve the policy used to make decisions.
  - Agent picks own actions.
  - Exploring starts (ES) is an on-policy method example.
  - However: ES is a restrictive assumption and not always applicable (in some cases the starting state-action pair cannot be choosen freely).

- Off-policy learning
  - Evaluate or improve a policy different from that used to generate data.
  - Agent cannot apply own actions.
  - Will be focused in the next sections.

ε-greedy as an on-policy alternative

- Exploration requirement:
  - Visit all state-action pairs with probability:

$$\pi(u|x) > 0 \quad \forall \{x \in \mathcal{X}, u \in \mathcal{U}\} . \tag{4.7}$$

  - Policies with this characteristic are called: soft.
  - Level of exploration can be tuned during the learning process.

- ε-greedy on-policy learning
  - With probability ε the agent's choice (i.e., the policy output) is overwritten with a random action.
  - Probability of all non-greedy actions:

$$\varepsilon/|\mathcal{U}| . \tag{4.8}$$

  - Probability of the greedy action:

$$1 - \varepsilon + \varepsilon/|\mathcal{U}| . \tag{4.9}$$

  - Above, $|\mathcal{U}|$ is the cardinality of the action space.

Algorithmic implementation ε-greedy MC-control

output: Optimal ε-greedy policy $\pi^*(u|x)$, **parameter:** $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: $\pi_{i=0}(u|x)$ arbitrarily soft $\forall \{x \in \mathcal{X}, u \in \mathcal{U}\}$
$\hat{q}(x,u)$ arbitrarily $\forall \{x \in \mathcal{X}, u \in \mathcal{U}\}$
$n(x,u) \leftarrow$ an empty list counting state-action visits $\forall \{x \in \mathcal{X}, u \in \mathcal{U}\}$
repeat
Generate an episode following π_i: $x_0, u_0, r_1, \ldots, x_{T_j}, u_{T_j}, r_{T_{j+1}}$;
$i \leftarrow i + 1$;
Set $g \leftarrow 0$;
for $k = T_i - 1, T_i - 2, T_i - 3, \ldots, 0$ *time steps* **do**
$g \leftarrow \gamma g + r_{k+1}$;
if $\{x_k, u_k\} \notin \langle \{x_0, u_0\}, \ldots, \{x_{k-1}, u_{k-1}\} \rangle$ **then**
$n(x_k, u_k) \leftarrow n(x_k, u_k) + 1$;
$\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + 1/n(x_k, u_k) \cdot (g - \hat{q}(x_k, u_k))$;
$\tilde{u} \leftarrow \arg\max_u \hat{q}(x_k, u)$;
$$\pi_i(u|x_k) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{U}|, & u = \tilde{u} \\ \varepsilon/|\mathcal{U}|, & u \neq \tilde{u} \end{cases} ;$$
until $\pi_{i+1} = \pi_i$;

Algo. 4.3: MC-based control using ε-greedy approach

ε-greedy policy improvement (1)

Theorem 4.1: Policy improvement for ε-greedy policy

Given an MDP, for any ε-greedy policy π the ε-greedy policy π' with respect to q_π is an improvement, i.e., $v_{\pi'} > v_\pi \quad \forall x \in \mathcal{X}$.

Small proof:

$$\begin{aligned} q_\pi(x, \pi'(x)) &= \sum_u \pi'(u|x) q_\pi(x, u), \\ &= \frac{\varepsilon}{|\mathcal{U}|} \sum_u q_\pi(x, u) + (1 - \varepsilon) \max_u q_\pi(x, u), \\ &\geq \frac{\varepsilon}{|\mathcal{U}|} \sum_u q_\pi(x, u) + (1 - \varepsilon) \sum_u \frac{\pi(u|x) - \frac{\varepsilon}{|\mathcal{U}|}}{1 - \varepsilon} q_\pi(x, u). \end{aligned} \tag{4.10}$$

In the inequality line, the second term is the weighted sum over action values given an ε-greedy policy. This weighted sum will be always smaller or equal than $\max_u q_\pi(x, u)$.

ε-greedy policy improvement (2)

Continuation:

$$
\begin{aligned}
q_\pi(x, \pi'(x)) &\geq \frac{\varepsilon}{|\mathcal{U}|} \sum_u q_\pi(x,u) + (1-\varepsilon) \sum_u \frac{\pi(u|x) - \frac{\varepsilon}{|\mathcal{U}|}}{1-\varepsilon} q_\pi(x,u), \\
&= \frac{\varepsilon}{|\mathcal{U}|} \sum_u (q_\pi(x,u) - q_\pi(x,u)) + \sum_u \pi(u|x) q_\pi(x,u), \\
&= \sum_u \pi(u|x) q_\pi(x,u), \\
&= v_\pi(x).
\end{aligned}
\tag{4.11}
$$

- Policy improvement theorem is still valid when comparing ε-greedy policies against each other.
- But: There might be a non-ε-greedy policy which is better.

MC-based control example: forest tree MDP (1)

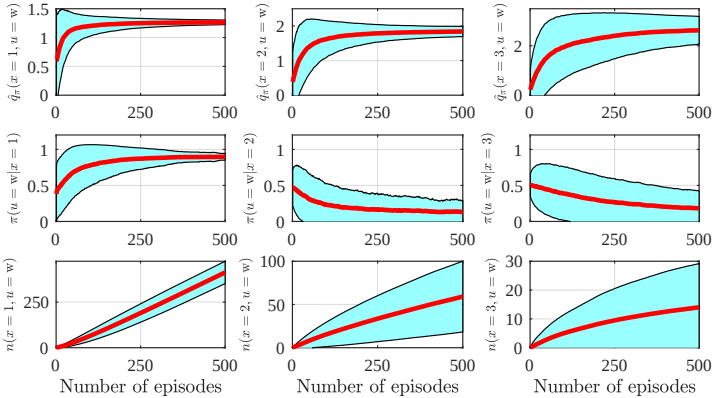


Fig. 4.6: Different estimates of forest tree MDP ($\alpha = 0.2, \gamma = 0.8$) using MC control with $\varepsilon = 0.2$ over the number of episodes. Mean is red and standard deviation is light blue, both calculated based on 2000 independent uns.

MC-based control example: forest tree MDP (2)

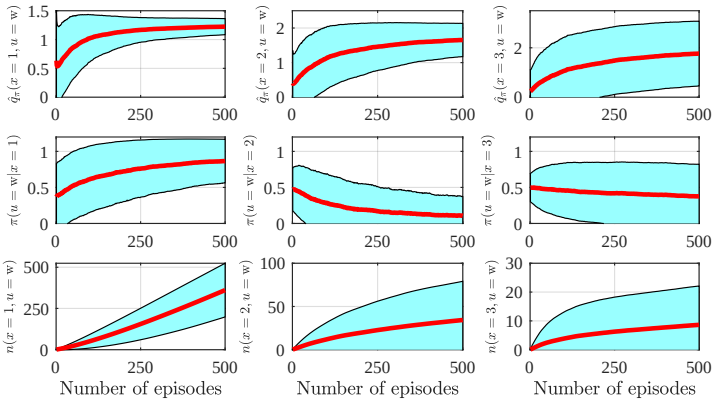


Fig. 4.7: Different estimates of forest tree MDP ($\alpha = 0.2, \gamma = 0.8$) using MC control with $\varepsilon = 0.05$ over the number of episodes. Mean is red and standard deviation is light blue, both calculated based on 2000 independent runs.

MC-based control example: forest tree MDP (3)

Observations on forest tree MDP with ε-greedy MC-based control:

- Rather slow convergence rate: quite a number of episodes is required.
- Significant uncertainty present in a single sequence.
- Later states are less often visited and, therefore, more uncertain.
- Exploration is controlled by ε: in a totally greedy policy the state $x = 3$ is not visited at all (cf. Fig. 2.16). With ε-greedy this state is visited occasionally.
- Nevertheless, the above figures highlight that MC-based control for the forest tree MDP tend towards the optimal policies discovered by dynamic programming (cf. Tab. 3.3).

Greedy in the limit with infinite exploration (GLIE)

Definition 4.1: Greedy in the limit with infinite exploration (GLIE)

A learning policy π is called GLIE if it satisfies the following two properties:

- If a state is visited infinitely often, then each action is chosen infinitely often:

$$\lim_{i \to \infty} \pi_i(u|x) = 1 \quad \forall \{x \in \mathcal{X}, u \in \mathcal{U}\} \, . \tag{4.12}$$

- In the limit ($i \to \infty$) the learning policy is greedy with respect to the learned action value:

$$\lim_{i \to \infty} \pi_i(u|x) = \pi(x) = \arg\max_u q(x, u) \quad \forall x \in \mathcal{X} \, . \tag{4.13}$$

GLIE Monte Carlo control

Theorem 4.2: Optimal decision using MC-control with ε-greedy

MC-based control using ε-greedy exploration is GLIE, if ε is decreased at rate

$$\varepsilon_i = \frac{1}{i} \tag{4.14}$$

with i being the increasing episode index. In this case,

$$\hat{q}(x, u) = q^*(x, u) \tag{4.15}$$

follows.

Remarks:

- Limited feasibility: infinite number of episodes required.
- ε-greedy is an undirected and unmonitored random exploration strategy. Can that be the most efficient way of learning?

Table of contents

Off-policy learning background

Drawback of on-policy learning:

- Only a compromise: comes with inherent exploration but at the cost of learning action values for a near-optimal policy.

Idea off-policy learning:

- Use two separated policies:
  - Behavior policy $b(u|x)$: explores in order to generate experience.
  - Target policy $\pi(u|x)$: learns from that experience to become the optimal policy.
- Use cases:
  - Learn from observing humans or other agents/controllers.
  - Re-use experience generated from old policies ($\pi_0, \pi_1, \ldots$).
  - Learn about multiple policies while following one policy.

Off-policy prediction problem statement

MC off-policy prediction problem statement

- Estimate v_π and/or q_π while following $b(u|x)$.
- Both policies are considered fixed (prediction assumption).

Requirement:

- Coverage: Every action taken under π must be (at least occasionally) taken under b, too. Hence, it follows:

$$\pi(u|x) > 0 \Rightarrow b(u|x) > 0 \quad \forall \{x \in \mathcal{X}, u \in \mathcal{U}\}. \tag{4.16}$$

- Consequences from that:
  - In any state b is not identical to π, b must be stochastic.
  - Nevertheless: π might be deterministic (e.g., control applications) or stochastic.

Importance sampling

Probability of observing a certain trajectory on random variables $U_k, X_{k+1}, U_{k+1}, \ldots, X_T$ starting in X_k while following π:

$$\begin{aligned}\mathbb{P}\left[U_k, X_{k+1}, U_{k+1}, \ldots, X_T | X_k, \pi\right] &= \pi(U_k|X_k)p(X_{k+1}|X_k, U_k)\pi(U_{k+1}|X_{k+1})\cdots, \\ &= \prod_k^{T-1} \pi(U_k|X_k)p(X_{k+1}|X_k, U_k).\end{aligned} \tag{4.17}$$

Above p is the state-transition probability (cf. Def. 2.5).

Definition 4.2: Importance sampling ratio

The relative probability of a trajectory under the target and behavior policy, the importance sampling ratio, from sample step k to T is:

$$\rho_{k:T} = \frac{\prod_k^{T-1} \pi(U_k|X_k)p(X_{k+1}|X_k, U_k)}{\prod_k^{T-1} b(U_k|X_k)p(X_{k+1}|X_k, U_k)} = \frac{\prod_k^{T-1} \pi(U_k|X_k)}{\prod_k^{T-1} b(U_k|X_k)}. \tag{4.18}$$

Importance sampling for Monte Carlo prediction

Definition 4.3: State-value estimation via Monte Carlo importance sampling

Estimating the state value v_π following a behavior policy b using (ordinary) importance sampling (OIS) results in scaling and averaging the sampled returns by the importance sampling ratio per episode:

$$\hat{v}_\pi(x_k) = \frac{\sum_{k \in \mathcal{T}(x_k)} \rho_{k:T(k)} g_k}{|\mathcal{T}(x_k)|}. \tag{4.19}$$

Notation remark:

- $\mathcal{T}(x_k)$: set of all time steps in which the state x_k is visited.
- $T(k)$: Termination of a specific episode starting from k.

General remark:

- From (4.18) it can be seen that $\hat{v}$ is bias-free (first-visit assumption).
- However, if ρ is large (distinctly different policies) the estimate's variance is large (i.e., uncertain for small numbers of samples).

Off-policy Monte Carlo control

Just put everything together:

- MC-based control utilizing GPI (cf. Fig. 4.5),
- Off-policy learning based on importance sampling (or variants like weighted importance sampling, cf. Barto/Sutton book chapter 5.5).

Requirement for off-policy MC-based control:

- Coverage: behavior policy b has nonzero probability of selecting actions that might be taken by the target policy π.
- Consequence: behavior policy b is soft (e.g., ε-soft).

Summary: what you've learned today

- MC methods allow model-free learning of value functions and optimal policies from experience in the form of sampled episodes.
- Using deep back-ups over full episodes, MC is largely based on averaging returns.
- MC-based control reuses generalized policy iteration (GPI), i.e., mixing policy evaluation and improvement.
- Maintaining sufficient exploration is important:
  - Exploring starts: not feasible in all applications but simple.
  - On-policy ε-greedy learning: trade-off between optimality and exploration cannot be resolved easily.
  - Off-policy learning: agent learns about a (possibly deterministic) target policy from an exploratory, soft behavior policy.
- Importance sampling transforms expectations from the behavior to the target policy.
  - This estimation task comes with a bias-variance-dilemma.
  - Slow learning can result from ineffective experience usage in MC methods.

Lecture 05: Temporal-Difference Learning

André Bodmer

Temporal-difference learning and the previous methods

Temporal-difference (TD) learning combines the previous ideas introduced in DP and MC:

- From Monte Carlo (MC) methods: Learns directly from experience.
- From dynamic programming (DP): Updates estimates based on other learned estimates (bootstrap).

Hence, TD characteristics are:

- Allows model-free prediction and control in unknown MDPs.
- Updates policy evaluation and improvement in an online fashion (i.e., not per episode) by bootstrapping.
- Still assumes finite MDP problems (or problems close to that).

Table of contents

General TD prediction updates

Recap the every-visit MC update rule (4.3) for non-stationary problems:

$$\hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha \left[g_k - \hat{v}(x_k) \right]. \tag{5.1}$$

- $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$ is the forgetting factor / step size.
- g_k is the target of the incremental update rule.
- To execute the update (5.1) one has to wait until the episode's termination since only then g_k is available (MC requirement).

One-step TD / TD(0) update

$$\hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha \left[r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k) \right]. \tag{5.2}$$

- Here, the TD target is $r_{k+1} + \gamma \hat{v}(x_{k+1})$.
- TD is bootstrapping: estimate $\hat{v}(x_k)$ based on $\hat{v}(x_{k+1})$.
- Delay time of one step and no need to wait until the episode's end.

Algorithmic implementation: TD-based prediction

input: a policy π to be evaluated
output: estimate of $\boldsymbol{v}_{\mathcal{X}}^{\pi}$ (i.e., value estimates for all states $x \in \mathcal{X}$)
init: $\hat{v}(x) \, \forall \, x \in \mathcal{X}$ arbitrary except $v_0(x) = 0$ if x is terminal
for $j = 1, \ldots, J$ *episodes* **do**
    Initialize x_0;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action from $\pi(x_k)$;
        Observe x_{k+1} and r_{k+1};
        $\hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha \left[r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k) \right]$;
        Exit loop if x_{k+1} is terminal;

Algo. 5.1: Tabular TD(0) prediction

- Note that the algorithm can be directly adapted to action-value prediction as it will be used for the later TD-based control approaches.

TD error

Fig. 5.1: Back up diagram for TD(0)

- TD as well as MC use sample updates.
- Looking ahead to a sample successor state including its value and the reward along the way to compute a backed up value estimate.

The TD error is:

$$\delta_k = r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k). \tag{5.3}$$

- δ_k is available at time step $k+1$.
- Iteratively δ_k converges towards zero.

TD error and its relation to the MC error

Let's assume that the TD(0) estimate $\hat{v}(x)$ is not changing over one episode as it would be for MC prediction:

$$\begin{aligned}
\underbrace{g_k - \hat{v}(x_k)}_{\text{MC-error}} &= r_{k+1} + \gamma g_{k+1} - \hat{v}(x_k) + \gamma\hat{v}(x_{k+1}) - \gamma\hat{v}(x_{k+1}), \\
&= \delta_k + \gamma(g_{k+1} - \hat{v}(x_{k+1})), \\
&= \delta_k + \gamma\delta_{k+1} + \gamma^2(g_{k+2} - \hat{v}(x_{k+2})), \\
&= \delta_k + \gamma\delta_{k+1} + \gamma^2\delta_{k+2} + \gamma^3(g_{k+3} - \hat{v}(x_{k+3})) = \cdots, \\
&= \sum_{i=k}^{T-1} \gamma^{i-k}\delta_i.
\end{aligned} \tag{5.4}$$

- MC error is the discounted sum of TD errors in this simplified case.
- If $\hat{v}(x)$ is updated during an episode (as expected in TD(0)), the above identity only holds approximately.

Overview of the RL methods considered so far

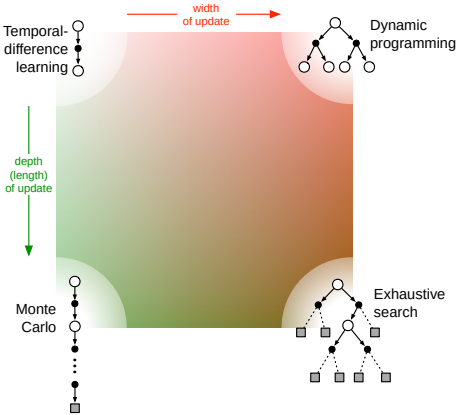


Fig. 5.2: Comparison of the RL methods considered so far with regard to the update rules (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Driving home example

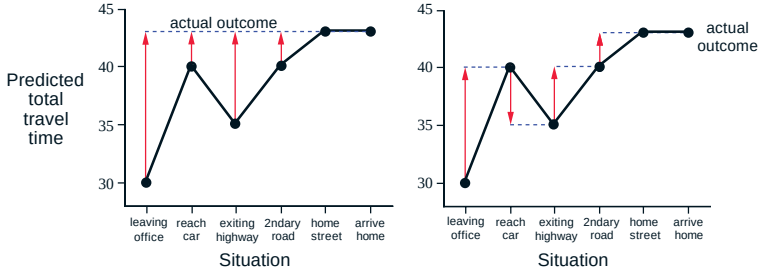


Fig. 5.3: Updates by MC (left) and TD (right) for $\alpha = 1$ (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- TD can learn before knowing the final outcome.
  - TD learns after every step.
  - MC must wait until the episode's end.
- TD could learn without a final outcome.
  - MC is only applicable to episodic tasks.

TD(0) prediction example: forest tree MDP (1)

Let's reuse the forest tree MDP example with *fifty-fifty policy* and discount factor $\gamma = 0.8$ plus disaster probability $\alpha = 0.2$:

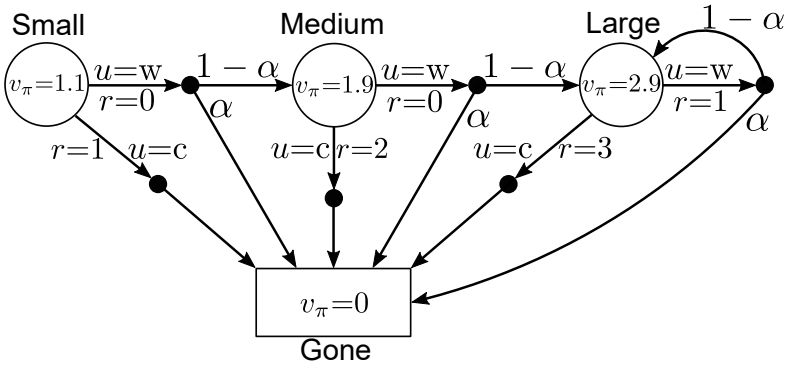


Fig. 5.4: Forest MDP with fifty-fifty-policy including state values

TD(0) prediction example: forest tree MDP (2)

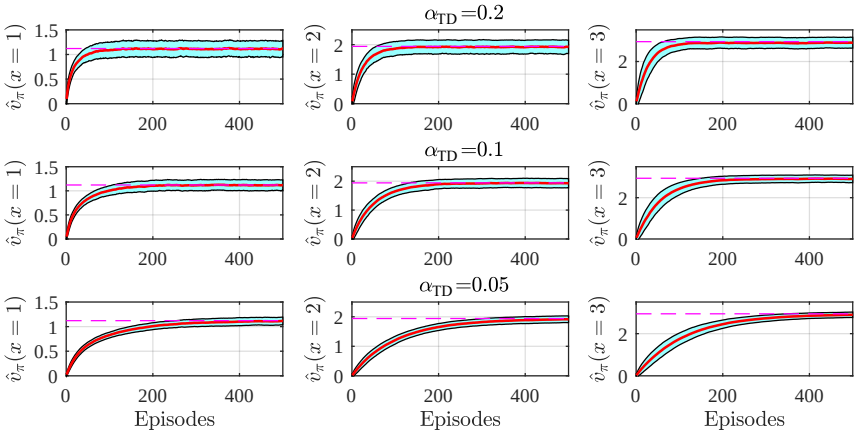


Fig. 5.5: State-value estimate of forest tree MDP using TD(0) prediction over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)

TD(0) vs. MC prediction example: forest tree MDP (1)

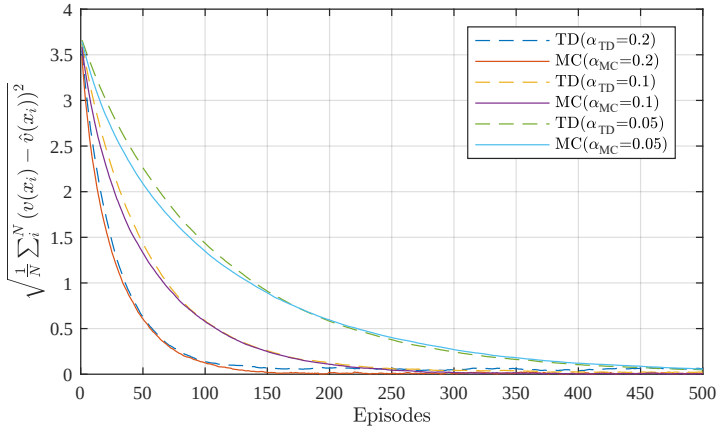


Fig. 5.6: Averaged mean of state-value estimates of forest tree MDP using TD(0) and MC over 1000 independent runs with $\hat{v}_0(x) = 0 \, \forall x \in \mathcal{X}$

TD(0) vs. MC prediction example: forest tree MDP (2)

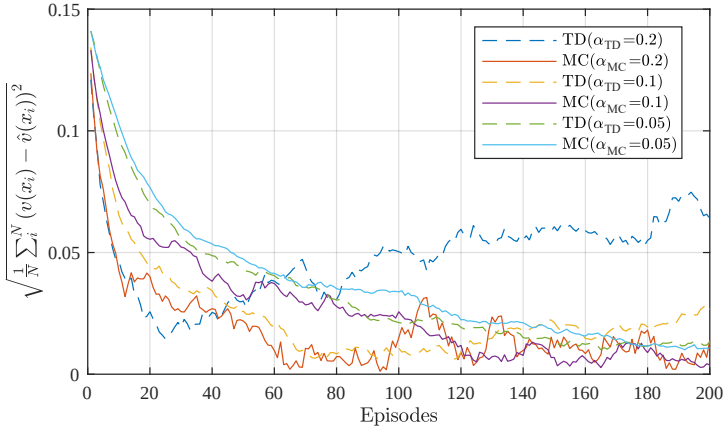


Fig. 5.7: Averaged mean of state-value estimates of forest tree MDP using TD(0) and MC over 1000 independent runs with $\hat{v}_0(x) \approx v(x) \, \forall x \in \mathcal{X}$

Convergence of TD(0)

Theorem 5.1: Convergence of TD(0)

Given a finite MDP and a fixed policy π the state-value estimate of TD(0) converges to the true v_π

- in the mean for a constant but sufficiently small step-size α and
- with probability 1 if the step-size holds the condition

$$\sum_{k=1}^{\infty} \alpha_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty. \tag{5.5}$$

Above k is the sample index (i.e., how often the TD update was applied).

- In particular, $\alpha_k = \frac{1}{k}$ meets the condition (5.5).
- Often TD(0) converges faster than MC, but there is no guarantee.
- TD(0) can be more sensitive to bad initializations $\hat{v}_0(x)$ compared to MC.

Batch training

- If experience $\to \infty$ both MC and TD converge $\hat{v}(x) \to v(x)$.
- But how to handle limited experience, i.e., a finite set of episodes

$$
\begin{aligned}
&x_{1,1}, u_{1,1}, r_{2,1}, \ldots, x_{T_1,1},\\
&x_{1,2}, u_{1,2}, r_{2,2}, \ldots, x_{T_2,2},\\
&\vdots\\
&x_{1,j}, u_{1,j}, r_{2,j}, \ldots, x_{T_j,j},\\
&\vdots\\
&x_{1,J}, u_{1,J}, r_{2,J}, \ldots, x_{T_J,J}.
\end{aligned}
$$

Batch training

- Process all available episodes $j \in [1, J]$ repeatedly to MC and TD.
- If the step size α is sufficiently small both will converge to certain steady-state values.

Batch training: AB-example (1)

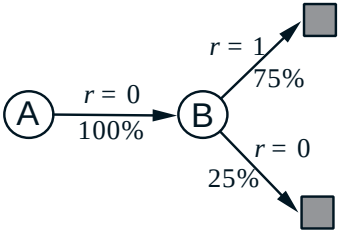


Fig. 5.8: Example environment (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- Only two states: A, B
- No discounting
- 8 episodes of experience available (see Tab. 5.1)
- What is $\hat{v}(A)$ and $\hat{v}(B)$ using batch training TD(0) and MC?

A, 0, B, 0	B,1
B,1	B,1
B,1	B,1
B,1	B,0

Batch training: AB-example (2)

First, recap MC and TD(0) update rules:

$$\begin{aligned}\text{MC}: &\quad \hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha\left[g_k - \hat{v}(x_k)\right], \\ \text{TD}: &\quad \hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha\left[r_{k+1} + \gamma\hat{v}(x_{k+1}) - \hat{v}(x_k)\right].\end{aligned}$$

Then, in steady state one receives:

$$\begin{aligned}\text{MC}: &\quad 0 = \alpha\left[g_k - \hat{v}(x_k)\right] = g_k - \hat{v}(x_k), \\ \text{TD}: &\quad 0 = \alpha\left[r_{k+1} + \gamma\hat{v}(x_{k+1}) - \hat{v}(x_k)\right] = r_{k+1} + \gamma\hat{v}(x_{k+1}) - \hat{v}(x_k).\end{aligned}$$

Considering a batch learning sweep over $j = 1, \ldots, J$ episodes:

$$\begin{aligned}\text{MC}: &\quad 0 = \sum_{j=1}^{J} g_{k,j} - \hat{v}(x_{k,j}), \\ \text{TD}: &\quad 0 = \sum_{j=1}^{J} r_{k+1,j} + \gamma\hat{v}(x_{k+1,j}) - \hat{v}(x_{k,j}).\end{aligned}$$

Batch training: AB-example (3)

Apply the previous equations first to state B. Since B is a terminal state, $\hat{v}(x_{k+1}) = 0$ and $g_{k,j} = r_{k+1,j}$ apply, i.e., the MC and TD updates are identical for B:

$$\text{MC}|_{x=\text{B}}: \quad 0 = \sum_{j=1}^{J} g_{k,j} - \hat{v}(x_{k,j}) \quad \Leftrightarrow \quad \hat{v}(\text{B}) = \frac{1}{J} \sum_{j=1}^{J} g_{k,j},$$

$$\text{TD}|_{x=\text{B}}: \quad 0 = \sum_{j=1}^{J} r_{k+1,j} - \hat{v}(x_{k,j}) \quad \Leftrightarrow \quad \hat{v}(\text{B}) = \frac{1}{J} \sum_{j=1}^{J} g_{k,j}.$$

This is the average return of the available episodes from Tab. 5.1 , i.e., 6×1 and 2×0:

$$\hat{v}(\text{B})|_{\text{MC}} = \hat{v}(\text{B})|_{\text{TD}} = \frac{6}{8} = 0.75\,. \tag{5.6}$$

Batch training: AB-example (4)

Now consider state A assuming the steady state of batch learning process:

- The instantaneous reward is always $r = 0$.
- The TD bootstrap estimate of B is $\hat{v}(x_{k+1,j}) = \hat{v}(B) = \frac{3}{4}$.

$$\text{MC}: \quad 0 = \sum_{j=1}^{J} g_{k,j} - \hat{v}(x_{k,j}) = \sum_{j=1}^{J} g_{k,j} - \hat{v}(A),$$

$$\text{TD}: \quad 0 = \sum_{j=1}^{J} r_{k+1,j} + \gamma\hat{v}(x_{k+1,j}) - \hat{v}(x_{k,j}) = \sum_{j=1}^{J} \gamma\hat{v}(B) - \hat{v}(A).$$

Looking at Tab. 5.1 there is only one episode visiting state A, where the sample return is $g_{k,j} = 0$. Hence, it follows:

$$\hat{v}(\text{A})|_{\text{MC}} = 0, \quad \hat{v}(\text{A})|_{\text{TD}} = \gamma\hat{v}(B) = \frac{3}{4}.$$

Where does this mismatch between the MC and TD estimates come from?

Certainty equivalence

- MC batch learning converges to the least squares fit of the sampled returns:

$$\sum_{j=1}^{J}\sum_{k=1}^{T_j}\left(g_{k,j}-\hat{v}(x_{k,j})\right)^2. \tag{5.7}$$

- TD batch learning converges to the maximum likelihood estimate such that $\left\langle \mathcal{X}, \mathcal{U}, \hat{\mathcal{P}}, \hat{\mathcal{R}}, \gamma \right\rangle$ explains the data with highest probability:

$$\begin{aligned}\hat{p}^u_{xx'} &= \frac{1}{n(x,u)}\sum_{j=1}^{J}\sum_{k=1}^{T_j} 1(X_{k+1}=x'|X_k=x, U_k=u),\\ \hat{\mathcal{R}}^u_x &= \frac{1}{n(x,u)}\sum_{j=1}^{J}\sum_{k=1}^{T_j} 1(X_k=x|U_k=u) r_{k+1,j}.\end{aligned} \tag{5.8}$$

- Here, TD assumes a MDP problem structure and is absolutely certain that its internal model concept describes the real world perfectly (so-called certainty equivalence).

Table of contents

Applying generalized policy iteration (GPI) to TD control

GPI concept is directly applied to the TD framework using action values:

$$\pi_0 \to \hat{q}_{\pi_0} \to \pi_1 \to \hat{q}_{\pi_1} \to \cdots \pi^* \to \hat{q}_{\pi^*} \,. \tag{5.9}$$

One-step TD / TD(0) action-value update (SARSA)

The TD(0) action-value update is:

$$\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha \left[r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1}) - \hat{q}(x_k, u_k) \right]. \tag{5.10}$$

SARSA: state, action, reward, (next) state, (next) action evaluation

- In contrast to MC: continuous online updates of policy evaluation and improvement.
- On-policy approach requires exploration, e.g., by an ε-greedy policy:

$$\pi_i(u|x) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{U}|, & u = \tilde{u}, \\ \varepsilon/|\mathcal{U}|, & u \neq \tilde{u}. \end{cases} \tag{5.11}$$

TD-based on-policy control (SARSA)

parameter: $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$, $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$
init: $\hat{q}(x, u)$ arbitrarily (except terminal states) $\forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$
for $j = 1, 2, \ldots$ *episodes* **do**
    Initialize x_0;
    Choose u_0 from x_0 using a soft policy (e.g., ε-greedy) derived from $\hat{q}(x, u)$;
    $k \leftarrow 0$;
    repeat
        Take action u_k, observe r_{k+1} and x_{k+1};
        Choose u_{k+1} from x_{k+1} using a soft policy derived from $\hat{q}(x, u)$;
        $\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha\left[r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1}) - \hat{q}(x_k, u_k)\right]$;
        $k \leftarrow k + 1$;
    until x_k *is terminal*;

Algo. 5.2: TD-based on-policy control (SARSA)

Convergence properties are comparable to MC-based on-policy control:

- Policy improvement theorem Theo. 4.1 holds.
- Greedy in the limit with infinite exploration (GLIE) from Def. 4.1 and step-size requirements in Theo. 5.1 apply.

SARSA example: forest tree MDP (1)

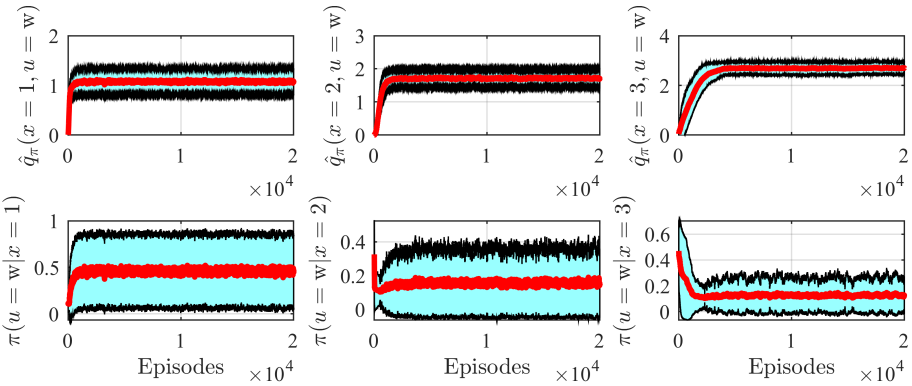


Fig. 5.9: SARSA-based control with $\alpha_{\mathrm{SARSA}} = 0.2$ and ε-greedy policy with $\varepsilon = 0.2$ of forest tree MDP over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)

SARSA example: forest tree MDP (2)

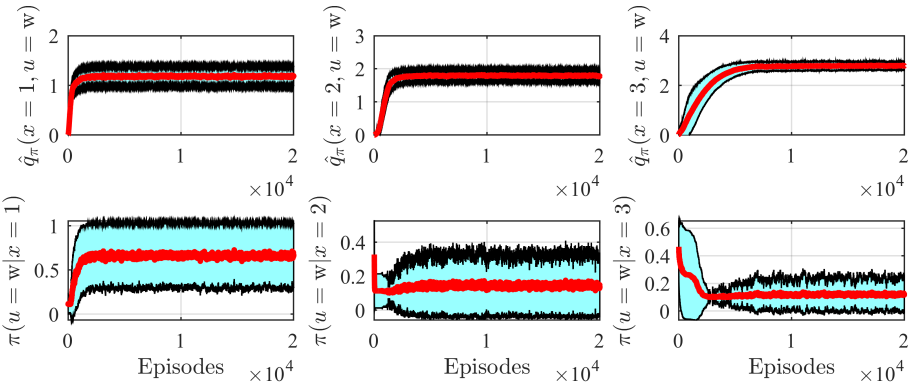


Fig. 5.10: SARSA-based control with $\alpha_{\text{SARSA}} = 0.1$ and ε-greedy policy with $\varepsilon = 0.2$ of forest tree MDP over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)

SARSA example: forest tree MDP (3)

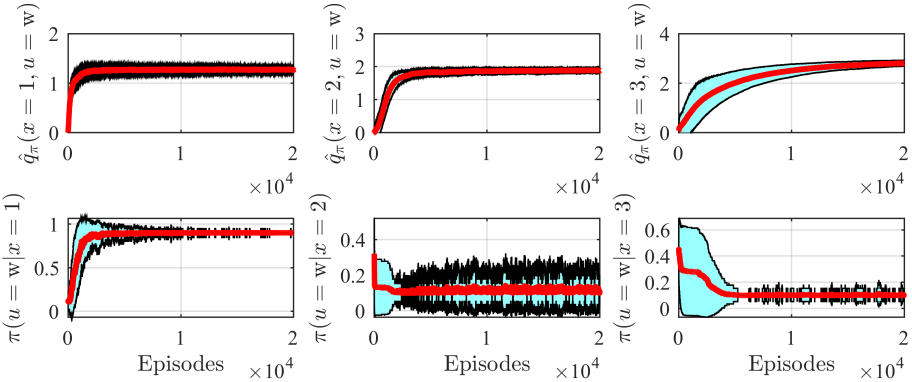


Fig. 5.11: SARSA-based control with adaptive $\alpha_{\mathrm{SARSA}} = \frac{1}{\sqrt{j}}$ (j =episode) and ε-greedy policy with $\varepsilon = 0.2$ of forest tree MDP over the number of episodes being evaluated (mean and standard deviation are calculated based on 2000 independent runs)

Table of contents

Q-learning approach

Similar to SARSA updates, but Q-learning directly estimates q^*:

Q-learning action-value update

The Q-learning action-value update is:

$$\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha \left[r_{k+1} + \gamma \max_u \hat{q}(x_{k+1}, u) - \hat{q}(x_k, u_k) \right]. \tag{5.12}$$

This is an off-policy update, since the optimal action-value function is updated independent of a given behavior policy.

Requirement for Q-learning control:

- Coverage: behavior policy b has nonzero probability of selecting actions that might be taken by the target policy π.
- Consequence: behavior policy b is soft (e.g., ε-soft).
- Step-size requirements (5.5) regarding α apply.

TD-based off-policy control (Q-learning)

parameter: $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$, $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$
init: $\hat{q}(x, u)$ arbitrarily (except terminal states) $\forall \{x \in \mathcal{X}, u \in \mathcal{U}\}$
for $j = 1, 2, \ldots$ *episodes* **do**
    Initialize x_0;
    $k \leftarrow 0$;
    repeat
        Choose u_k from x_k using a soft behavior policy;
        Take action u_k, observe r_{k+1} and x_{k+1};
        $\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha \left[r_{k+1} + \gamma \max_u \hat{q}(x_{k+1}, u) - \hat{q}(x_k, u_k)\right]$;
        $k \leftarrow k + 1$;
    until x_k *is terminal*;

Algo. 5.3: TD-based off-policy control (Q-learning)

- As discussed with MC-based off-policy control: avoidance of the exploration-optimality trade-off for on-policy methods.
- No importance sampling required as for off-policy MC-based control.

Q-learning control example: cliff walking

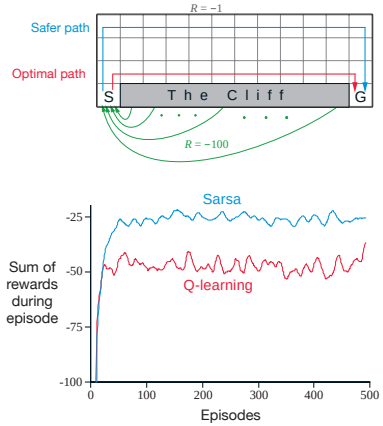


Fig. 5.12: Cliff walking environment (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- $r = -1$ per time step
- Large penalty if you fall off the cliff
- No discounting
- $\varepsilon = 0.1$

- Why is SARSA better in this example?
- And what policy's performance is shown here in particular?

Table of contents

Maximization bias

All control algorithms discussed so far involve maximization operations:

- Q-learning: target policy is greedy and directly uses $\max$ operator for action-value updates.
- SARSA: typically uses an ε-greedy framework, which also involves $\max$ updates during policy improvement.

This can lead to a significant positive bias:

- Maximization over sampled values is used implicitly as an estimate of the maximum value.
- This issue is called maximization bias.

Small example:

- Consider a single state x with multiple possible actions u.
- The true action values are all $q(x, u) = 0$.
- The sampled estimates $\hat{q}(x, u)$ are uncertain, i.e., randomly distributed. Some samples are above and below zero.
- Consequence: The maximum of the estimate is positive.

Double learning approach

Split the learning process:

- Divide sampled experience into two sets.
- Use sets to estimate independent estimates $\hat{q}_1(x, u)$ and $\hat{q}_2(x, u)$.

Assign specific tasks to each estimate:

- Estimate the maximizing action:

$$u^* = \arg\max_u \hat{q}_1(x, u). \tag{5.13}$$

- Estimate corresponding action value:

$$q(x, u^*) \approx \hat{q}_2(x, u^*) = \hat{q}_2(x, \arg\max_u \hat{q}_1(x, u)). \tag{5.14}$$

Double Q-learning algorithm

parameter: $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$, $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$
init: $\hat{q}_1(x,u)$, $\hat{q}_2(x,u)$ arbitrarily (except terminal states) $\forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$
for $j = 1, 2, \ldots$ *episodes* **do**
    Initialize x_0;
    $k \leftarrow 0$;
    repeat
        Choose u_k from x_k using the policy ε-greedy based on $\hat{q}_1(x,u) + \hat{q}_2(x,u)$;
        Take action u_k, observe r_{k+1} and x_{k+1};
        if $n \sim \mathcal{N}(\mu = 0, \sigma) > 0$ **then**
            $\hat{q}_1(x_k,u_k) \leftarrow \hat{q}_1(x_k,u_k) + \alpha\left[r_{k+1} + \gamma \hat{q}_2(x_{k+1}, \arg\max_u \hat{q}_1(x_{k+1},u)) - \hat{q}_1(x_k,u_k)\right]$;
        else
            $\hat{q}_2(x_k,u_k) \leftarrow \hat{q}_2(x_k,u_k) + \alpha\left[r_{k+1} + \gamma \hat{q}_1(x_{k+1}, \arg\max_u \hat{q}_2(x_{k+1},u)) - \hat{q}_2(x_k,u_k)\right]$;
        $k \leftarrow k + 1$;
    until x_k *is terminal*;

Algo. 5.4: TD-based off-policy control with double learning

- Doubles memory demand while computational demand per episode is remains unchanged
- Less sample efficient than regular Q-learning (samples are split between two estimators)

Maximization bias example

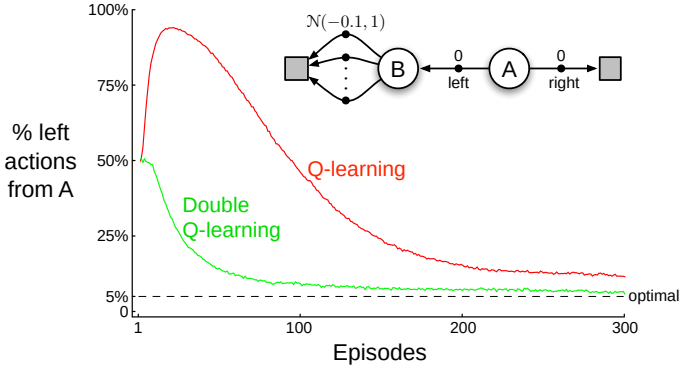


Fig. 5.13: Comparison of Q-learning and double Q-learning on a simple episodic MDP. Q-learning initially learns to take the left action much more often than the right action, and always takes it significantly more often than the 5% minimum probability enforced by ε-greedy action selection with $\varepsilon = 0.1$. In contrast, double Q-learning is essentially unaffected by maximization bias. These data are averaged over 10,000 runs. The initial action-value estimates were zero. (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Summary: what you've learned today

- TD unites two key characteristics from DP and MC:
  - From MC: Sample-based updates (i.e., operating in unknown MDPs).
  - From DP: Update estimates based on other estimates (bootstrapping).
- TD allows certain simplifications and improvements compared to MC:
  - Updates are available after each step and not after each episode.
  - Off-policy learning comes without importance sampling.
  - Exploits MDP formalism by maximum likelihood estimates.
  - Hence, TD prediction and control exhibit a high applicability for many problems.
- Batch training can be used when only limited experience is available, i.e., the available samples are re-processed again and again.
- Greedy policy improvements can lead to maximization biases and, therefore, slow down the learning process.
- TD requires careful tuning of learning parameters:
  - Step size α: how to tune convergence rate vs. uncertainty / accuracy?
  - Exploration vs. exploitation: how to visit all state-action pairs?

Lecture 06: Multi-Step Bootstrapping

André Bodmer

Lets unify MC and TD learning

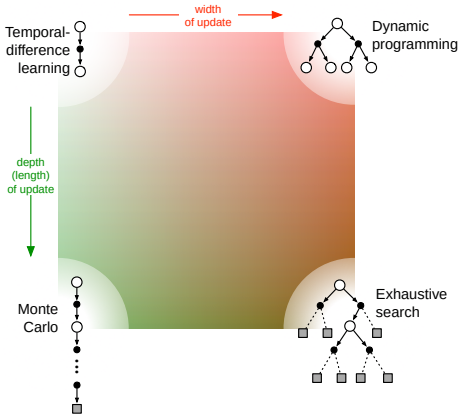


Fig. 6.1: MC and TD are the 'extreme options' in terms of the update's depth: what about intermediate solutions? (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Table of contents

n-step bootstrapping idea

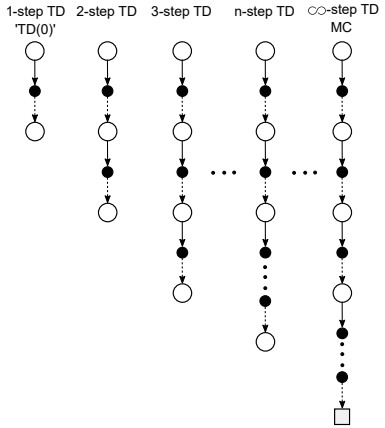


Fig. 6.2: Different backup diagrams of n-step state-value prediction methods

- n-step update: consider n rewards plus estimated value n-steps later (bootstrapping).
- Consequence: Estimate update is available only after an n-step delay.
- TD(0) and MC are special cases included in n-step prediction.

Formal notation (1)

Recap the update targets for the incremental prediction methods (4.3):

- Monte Carlo: builds on the complete sampled return series

$$g_{k:T} = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \cdots + \gamma^{T-k-1} r_T. \tag{6.1}$$

  - $g_{k:T}$ denotes that all steps until termination at T are considered to derive an estimate target adressing step k.

- TD(0): utilizes a one-step bootstrapped return

$$g_{k:k+1} = r_{k+1} + \gamma \hat{v}_k(x_{k+1}). \tag{6.2}$$

  - For TD(0), $g_{k:k+1}$ highlights that only one future sampled reward step is considered before bootstrapping.
  - $\hat{v}_k$ is an estimate of v_π at time step k.

Formal notation (2)

n-step state-value prediction target

Now, the target is generalized to an arbitrary n-step target:

$$g_{k:k+n} = r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{v}_{k+n-1}(x_{k+n}). \tag{6.3}$$

- Approximation of full return series truncated after n-steps.
- If $k + n \geq T$ (i.e., n-step prediction exceeds termination lookahead), then all missing terms are considered zero.

n-step TD

The state-value estimate using the n-step return approximation is

$$\hat{v}_{k+n}(x_k) = \hat{v}_{k+n-1}(x_k) + \alpha \left[g_{k:k+n} - \hat{v}_{k+n-1}(x_k) \right], \quad 0 \leq k < T. \tag{6.4}$$

- Delay of n-steps before $\hat{v}(x)$ is updated.
- Additional auxiliary update steps required at the end of each episode.

Convergence

Theorem 6.1: Error reduction property

The worst error of the expected n-step return is always less than or equal to γ^n times the worst error under the estimate $\hat{v}_{k+n-1}$:

$$\max_x \left|\mathbb{E}_\pi \left[G_{k:k+n} | X_k = x\right] - v_\pi(x)\right| \leq \gamma^n \max_x \left|\hat{v}_{k+n-1}(x) - v_\pi(x)\right|. \tag{6.5}$$

- Assuming an infinite number of steps/episodes and an appropriate step-size control according to Theo. 5.1, n-step TD prediction converges to the true value.
- In a more practical framework with limited number of steps/episodes:
  - Choosing the best n-step lookahead horizon is an engineering degree of freedom.
  - This is highly application-dependent (i.e., no predefined optimum).
  - Prediction/estimation errors can remain due to limited data.

Algorithmic implementation: n-step TD prediction

input: a policy π to be evaluated, **parameter:** step size $\alpha \in (0, 1]$, prediction steps $n \in \mathbb{Z}^+$
init: $\hat{v}(x) \, \forall \, x \in \mathcal{X}$ arbitrary except $v_0(x) = 0$ if x is terminal
for $j = 1, \ldots, J$ *episodes* **do**
    initialize and store x_0;
    $T \leftarrow \infty$;
    repeat $k = 0, 1, 2, \ldots$
        if $k < T$ **then**
            take action from $\pi(x_k)$, observe and store x_{k+1} and r_{k+1};
            if x_{k+1} is terminal: $T \leftarrow k + 1$;
        $\tau \leftarrow k - n + 1$ (τ time index for estimate update);
        if $\tau \geq 0$ **then**
            $g \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} r_i$;
            if $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{v}(x_{\tau+n})$;
            $\hat{v}(x_\tau) \leftarrow \hat{v}(x_\tau) + \alpha \, [g - \hat{v}(x_\tau)]$;
    until $\tau = T - 1$;

Algo. 6.1: n-step TD prediction (output is an estimate $\hat{v}_\pi(x)$)

Example: 19 state random walk

Fig. 6.3: Exemplary random walk Markov reward process (MRP)

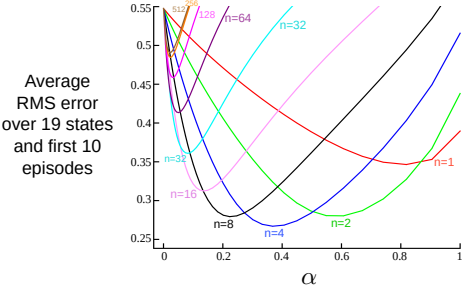


Fig. 6.4: n-step TD performance (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- Early stage performance after only 10 episodes
- Averaged over 100 independent runs
- Best result here: $n = 4, \alpha \approx 0.4$
- Picture may change for longer episodes (no generalizable results)

Table of contents

Transfer the n-step approach to state-action values (1)

- For on-policy control by SARSA action-value estimates are required.
- Recap the one-step action-value update as required for 'SARSA(0)':

$$\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha \left[\underbrace{r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1})}_{\text{target } g} - \hat{q}(x_k, u_k) \right]. \tag{6.6}$$

n-step state-action value prediction target

Analog to n-step TD, the state-action value target is rewritten as:

$$g_{k:k+n} = r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{q}_{k+n-1}(x_{k+n}, u_{k+n}). \tag{6.7}$$

- Again, if an episode terminates within the lookahead horizon ($k + n \geq T$) the target is equal to the Monte Carlo update:

$$g_{k:k+n} = g_k. \tag{6.8}$$

Transfer the n-step approach to state-action values (2)

- For n-step expected SARSA, the update is similar but the state-action value estimate at step $k+n$ becomes the expected approximate value of x under the target policy valid at time step k:

$$g_{k:k+n} = r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \sum_u \pi(u|x)\hat{q}_k(x,u). \tag{6.9}$$

- Finally, the modified n-step targets can be directly integrated to the state-action value estimate update rule of SARSA:

n-step SARSA

$$\hat{q}_{k+n}(x_k,u_k) = \hat{q}_{k+n-1}(x_k,u_k) + \alpha\left[g_{k:k+n} - \hat{q}_{k+n-1}(x_k,u_k)\right], \quad 0 \le k < T. \tag{6.10}$$

n-step bootstrapping for state-action values

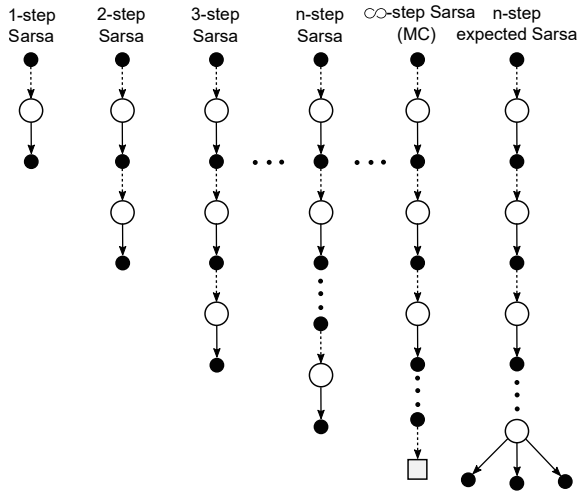


Fig. 6.5: Different backup diagrams of n-step state-action value update targets

Algorithmic implementation: n-step SARSA

parameter: $\alpha \in (0, 1]$, $n \in \mathbb{Z}^+$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$
init: $\hat{q}(x, u)$ arbitrarily (except terminal states) $\forall \{x \in \mathcal{X}, u \in \mathcal{U}\}$
init: π to be ε-greedy with respect to $\hat{q}$ or to a given, fixed policy
for $j = 1, \ldots, J$ *episodes* **do**
    initialize x_0 and action $u_0 \sim \pi(\cdot|x_0)$ and store them;
    $T \leftarrow \infty$;
    repeat $k = 0, 1, 2, \ldots$
        if $k < T$ **then**
            take action u_k, observe and store x_{k+1} and r_{k+1};
            if x_{k+1} *is terminal* **then** $T \leftarrow k + 1$ **else** store $u_{k+1} \sim \pi(\cdot|x_{k+1})$;
        $\tau \leftarrow k - n + 1$ (τ time index for estimate update);
        if $\tau \geq 0$ **then**
            $g \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} r_i$;
            if $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{q}(x_{\tau+n}, u_{\tau+n})$;
            $\hat{q}(x_\tau, u_\tau) \leftarrow \hat{q}(x_\tau, u_\tau) + \alpha [g - \hat{q}(x_\tau, u_\tau)]$;
            if $\pi \approx \pi^*$ is being learned, ensure $\pi(\cdot|x_\tau)$ is ε-greedy w.r.t $\hat{q}$;
    until $\tau = T - 1$;

Algo. 6.2: n-step SARSA (output is an estimate $\hat{q}_\pi$ or $\hat{q}^*$)

Illustration with grid-world example

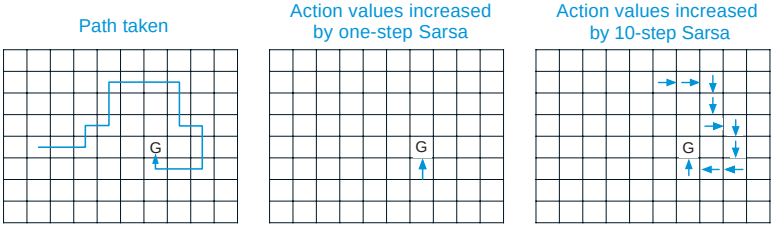


Fig. 6.6: Executed updates (highlighted by arrows) for different n-step SARSA implementations during an episode (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- For one-step SARSA, one state-action value is updated.
- For ten-step SARSA, ten state-action values are updated.
- Consequence: a trade-off between the resulting learning delay and the number of updated state-action values results.

Table of contents

Recap on off-policy learning with importance sampling

Consider two separate policies in order to break the on-policy optimality trade-off:

- Behavior policy $b(u|x)$: Explores in order to generate experience.
- Target policy $\pi(u|x)$: Learns from that experience to become the optimal policy.
- Important requirement is coverage: Every action taken under π must be (at least occasionally) taken under b, too. Hence, it follows:

$$\pi(u|x) > 0 \Rightarrow b(u|x) > 0 \quad \forall \{x \in \mathcal{X}, u \in \mathcal{U}\}. \tag{6.11}$$

Importance sampling ratio (revision from Def. 4.2)

The relative probability of a trajectory under the target and behavior policy, the importance sampling ratio, from sample step k to T is:

$$\rho_{k:T} = \frac{\prod_k^{T-1} \pi(u_k|x_k) p(x_{k+1}|x_k, u_k)}{\prod_k^{T-1} b(u_k|x_k) p(x_{k+1}|x_k, u_k)} = \frac{\prod_k^{T-1} \pi(u_k|x_k)}{\prod_k^{T-1} b(u_k|x_k)}. \tag{6.12}$$

Transfer importance sampling to n-step updates

For a straightforward n-step off-policy TD-style update, just weight the update by the importance sampling ratio:

$$\hat{v}_{k+n}(x_k) = \hat{v}_{k+n-1}(x_k) + \alpha\rho_{k:k+n-1}\left[g_{k:k+n} - \hat{v}_{k+n-1}(x_k)\right], \quad 0 \leq k < T,$$
$$\rho_{k:h} = \prod_{k}^{\min(h,T-1)} \frac{\pi(u_k|x_k)}{b(u_k|x_k)}. \tag{6.13}$$

- $\rho_{k:k+n-1}$ is the relative probability under the two polices taking n actions from u_k to u_{k+n}.

Analog, an n-step off-policy SARSA-style update exists:

$$\hat{q}_{k+n}(x_k, u_k) = \hat{q}_{k+n-1}(x_k, u_k) + \alpha\rho_{k+1:k+n}\left[g_{k:k+n} - \hat{q}_{k+n-1}(x_k, u_k)\right], \quad 0 \leq k < T. \tag{6.14}$$

- Here, ρ starts and ends one step later compared to the TD case since state-action pairs are updated.

Algorithmic implementation: off-policy n-step TD-based prediction

input: a target policy π and a behavior policy b with coverage of π
parameter: step size $\alpha \in (0, 1]$, prediction steps $n \in \mathbb{Z}^+$
init: $\hat{v}(x) \, \forall \, x \in \mathcal{X}$ arbitrary except $v_0(x) = 0$ if x is terminal
for $j = 1, \ldots, J$ *episodes* **do**
    initialize and store x_0 and set $T \leftarrow \infty$;
    repeat $k = 0, 1, 2, \ldots$
        if $k < T$ **then**
            take action from $b(x_k)$, observe and store x_{k+1} and r_{k+1};
            if x_{k+1} is terminal: $T \leftarrow k + 1$;
        $\tau \leftarrow k - n + 1$ (τ time index for estimate update);
        if $\tau \geq 0$ **then**
            $\rho \leftarrow \prod_{i=\tau}^{\min(\tau+n-2, T-1)} \frac{\pi(u_i|x_k)}{b(u_i|x_i)}$;
            $g \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} r_i$;
            if $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{v}(x_{\tau+n})$;
            $\hat{v}(x_\tau) \leftarrow \hat{v}(x_\tau) + \alpha\rho \, [g - \hat{v}(x_\tau)]$;
    until $\tau = T - 1$;

Algo. 6.3: Off-policy n-step TD prediction (output is an estimate $\hat{v}_\pi(x)$)

Algorithmic implementation: off-policy n-step SARSA

input: an arbitrary behavior policy b with $b(u|x) > 0 \forall \{x \in \mathcal{X}, u \in \mathcal{U}\}$
parameter: $\alpha \in (0, 1]$, $n \in \mathbb{Z}^+$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: $\hat{q}(x, u) \; \forall \; \{x \in \mathcal{X}, u \in \mathcal{U}\}$ and a policy π to be greedy with respect to $\hat{q}$ or to a given, fixed policy
for $j = 1, \ldots, J$ *episodes* **do**
    initialize x_0 and action $u_0 \sim b(\cdot|x_0)$ and store them, set also $T \leftarrow \infty$;
    repeat $k = 0, 1, 2, \ldots$
        if $k < T$ **then**
            take action $u_k \sim b(\cdot|x_k)$, observe and store x_{k+1} and r_{k+1};
            if x_{k+1} *is terminal* **then** $T \leftarrow k + 1$ **else** store $u_{k+1} \sim b(\cdot|x_{k+1})$;
        $\tau \leftarrow k - n + 1$ (τ time index for estimate update);
        if $\tau \geq 0$ **then**
            $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n-1, T-1)} \frac{\pi(u_i|x_i)}{b(u_i|x_i)}$;
            $g \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} r_i$;
            if $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{q}(x_{\tau+n}, u_{\tau+n})$;
            $\hat{q}(x_\tau, u_\tau) \leftarrow \hat{q}(x_\tau, u_\tau) + \alpha\rho[g - \hat{q}(x_\tau, u_\tau)]$;
            if $\pi \approx \pi^*$ is being learned, ensure $\pi(\cdot|x_\tau)$ is ε-greedy w.r.t to $\hat{q}$;
    until $\tau = T - 1$;

Algo. 6.4: Off-policy n-step SARSA (output is an estimate $\hat{q}_\pi$ or $\hat{q}^*$)

Table of contents

Averaging of n-step returns

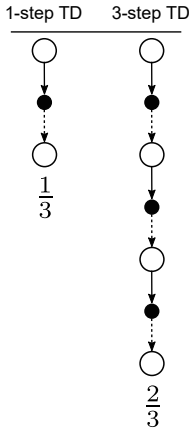


Fig. 6.7: Exemplary averaging of n-step returns

- Averaging different n-step returns is possible without introducing a bias (if sum of weights is one).
- Example on the left:

$$g = \frac{1}{3} g_{k:k+1} + \frac{2}{3} g_{k:k+3}$$

- Horizontal line in backup diagram indicates the averaging.
- Enables additional degree of freedom to reduce prediction error.
- Such updates are called compound updates.

λ-return (1)

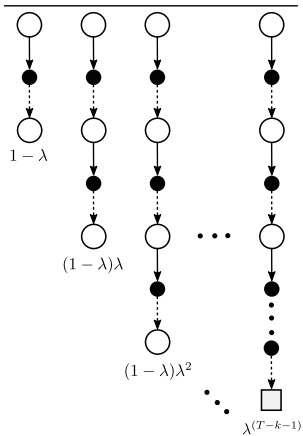


Fig. 6.8: Backup diagram for λ-returns

- λ-return: is a compound update with exponentially decaying weights:

$$g_k^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{(n-1)}g_{k:k+n}\,. \qquad (6.15)$$

- Parameter is $\lambda \in \{\mathbb{R}|0 \leq \lambda \leq 1\}$.
- Geometric series of weights is one:

$$(1-\lambda)\sum_{n=1}^{\infty}\lambda^{(n-1)} = 1$$

λ-return (2)

- Rewrite λ-return for episodic tasks with termination at $k = T$:

$$g_k^\lambda = (1-\lambda) \sum_{n=1}^{T-k-1} \lambda^{(n-1)} g_{k:k+n} + \lambda^{T-k-1} g_k \,. \tag{6.16}$$

- Return g_k after termination is weighted with residual weight λ^{T-k-1}.
- Above, (6.16) includes two special cases:
  - If $\lambda = 0$: becomes TD(0) update.
  - If $\lambda = 1$: becomes MC update.

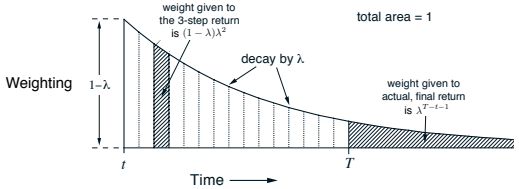


Fig. 6.9: Weighting overview in λ-return series (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Truncated λ-returns for continuing tasks

- Using λ-returns as in (6.15) is not feasible for continuing tasks.
- One would have to wait infinitely long to receive the trajectory.
- Intuitive approximation: truncate λ-return after h steps

$$g^{\lambda}_{k:h} = (1-\lambda) \sum_{n=1}^{h-k-1} \lambda^{(n-1)} g_{k:k+n} + \lambda^{h-k-1} g_{k:h} \,. \tag{6.17}$$

- Horizon h divides continuing tasks in rolling episodes.

Forward view

- Both, n-step and λ-return updates, are based on a forward view.
- We have to wait for future states and rewards to arrive before we are able to perform an update.
- Currently, λ-returns are only an alternative to n-step updates with different weighting options.

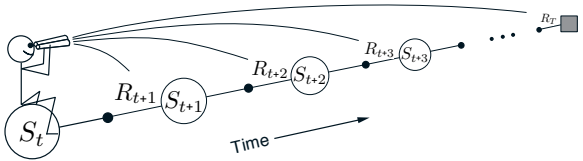


Fig. 6.10: The forward view: an update of the current state value is evaluated by future transitions (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Backward view of TD(λ)

General idea:

- Use λ-weighted returns looking into the past.
- Implement this in a recursive fashion to save memory.
- Therefore, an eligibility trace z_k denoting the importance of past events to the current state update is introduced.

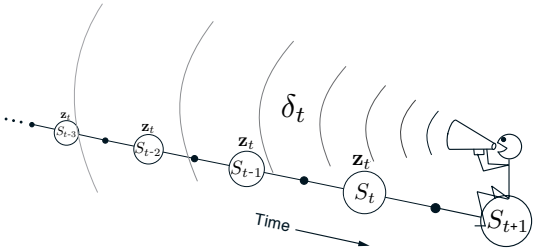


Fig. 6.11: The backward view: an update of the current state value is evaluated based on a trace of past transitions (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Eligibility trace

The eligibility trace $z_k(x) \in \mathbb{R}$ is defined and tracked for each state x separately:

$$z_0(x) = 0,$$
$$z_k(x) = \gamma\lambda z_{k-1}(x) + \begin{cases} 0, & \text{if } x_k \neq x, \\ 1, & \text{if } x_k = x. \end{cases} \tag{6.18}$$

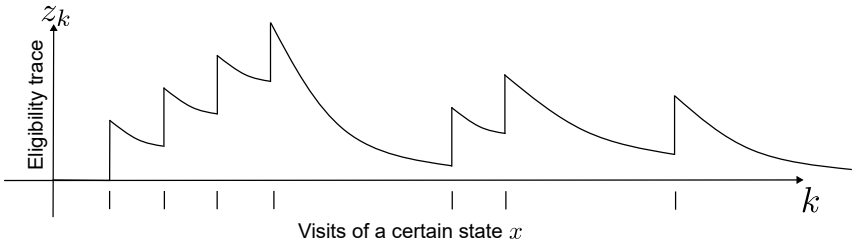


Fig. 6.12: Simplified representation of updating an eligibility trace of an arbitrary state in a finite MDP

TD(λ) updates using eligibility traces

Based on the eligibility trace definition from (6.18) we can modify our value estimates:

TD(λ) state-value update

The TD(λ) state-value update is:

$$\hat{v}(x_k) \leftarrow \hat{v}(x_k) + \alpha \left[r_{k+1} + \gamma \hat{v}(x_{k+1}) - \hat{v}(x_k) \right] z_k(x_k). \tag{6.19}$$

SARSA(λ) action-value update

The SARSA(λ) action-value update is:

$$\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha \left[r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1}) - \hat{q}(x_k, u_k) \right] z_k(x_k, u_k). \tag{6.20}$$

Already known prediction and control methods can be modified accordingly. In contrast to n-step forward updates, one can conclude:

- Advantage: recursive updates based on past updates (no additional waiting time),
- Disadvantage: effort for storing an eligibility trace for each state (scaling problem).

Algorithmic implementation: SARSA(λ)

parameter: $\alpha \in (0,1]$, $\lambda \in (0,1]$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$
init: $\hat{q}(x,u)$ arbitrarily (except terminal states) $\forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$
init: π to be ε-greedy with respect to $\hat{q}$ or to a given, fixed policy
for $j = 1, \ldots, J$ *episodes* **do**
    initialize x_0 and action $u_0 \sim \pi(\cdot|x_0)$;
    initialize $z_0(x,u) = 0\ \forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$
    repeat
        take action u_k, observe x_{k+1} and r_{k+1};
        choose $u_{k+1} \sim \pi(\cdot|x_{k+1})$
        $z_k(x,u) \leftarrow \gamma\lambda z_{k-1}(x,u) + \begin{cases} 0, & \text{if } x_k \neq x \text{ or } u_k \neq u, \\ 1, & \text{if } x_k = x \text{ and } u_k = u. \end{cases} \qquad \forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$
        $\delta \leftarrow r_{k+1} + \gamma\hat{q}(x_{k+1}, u_{k+1}) - \hat{q}(x_k, u_k)$
        $\hat{q}(x,u) \leftarrow \hat{q}(x,u) + \alpha\delta z_k(x,u)\ \forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$
        $k \leftarrow k + 1$;
    until x_k *is terminal*;

Algo. 6.5: SARSA(λ) (output is an estimate $\hat{q}_\pi$ or $\hat{q}^*$)

SARSA learning comparison in gridworld example

- λ can be interpreted as the discounting factor acting on the eligibility trace (see right-most panel below).
- Intuitive interpretation: more recent transitions are more certain/relevant for the current update step.

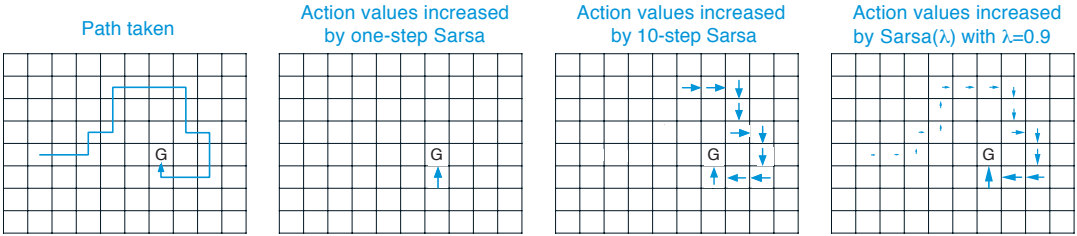


Fig. 6.13: SARSA variants after an arbitrary episode within a gridworld environment – arrows indicate action-value change starting from initially zero estimates (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Summary: what you've learned today

- n-step updates allow for an intermediate solution in between temporal difference and Monte Carlo:
  - $n = 1$: TD as special case,
  - $n = T$: MC as special case.
- The parameter n is a delicate degree of freedom:
  - It contains a trade-off between the learning delay and uncertainty reduction when considering more or less steps.
  - Choosing it is non-trivial and sometimes more art than science.
- λ-returns lead to compound updates which introduce an exponential weighting to visited states.
  - Rationale: states which have been already visited long ago are less important for the current learning step.
- TD(λ) transfers this idea into a recursive, backward oriented approach.
  - Eligibility traces store the long-term visiting history of each state in a recursive fashion.

Lecture 07: Planning and Learning with Tabular Methods

André Bodmer

Recap: RL agent taxonomy

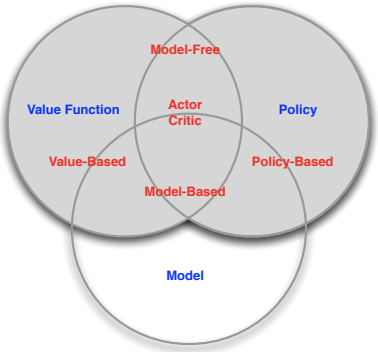


Fig. 7.1: Main categories of reinforcement learning algorithms
(source: D. Silver, Reinforcement learning, 2015. CC BY-NC 4.0)

- Up to now: independent usage of model-free (MC, TD) and model-based RL (DP)
- Today: integrating both strategies (on finite state & action spaces)

Table of contents

Model-based RL

- Plan/predict value functions and/or policy from a model.
- Requires an a priori model or to learn a model from experience.
- Solves control problems by planning algorithms such as
  - Policy or value iteration.

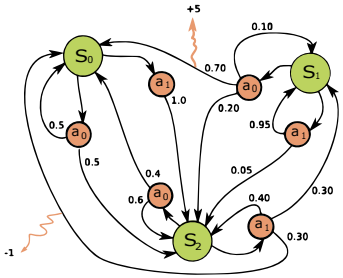


Fig. 7.2: A model for discrete state and action space problems is generally an MDP (source: www.wikipedia.org, by Waldoalvarez CC BY-SA 4.0)

What is a model?

- A model $\mathcal{M}$ is an MDP tuple $\langle \mathcal{X}, \mathcal{U}, \boldsymbol{\mathcal{P}}, \mathcal{R}, \gamma \rangle$.
- In particular, we require the
  - state-transition probability

$$\boldsymbol{\mathcal{P}} = \mathbb{P}\left[\boldsymbol{X}_{k+1} = \boldsymbol{x}_{k+1} | \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k\right] \tag{7.1}$$

  - and the reward probability

$$\mathcal{R} = \mathbb{P}\left[R_{k+1} = r_{k+1} | \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k\right] . \tag{7.2}$$

- State space $\mathcal{X}$ and action space $\mathcal{U}$ are assumed to be known.
- Discount factor γ might be given by environment or engineer's choice.
- What kind of model is available?
  - If $\mathcal{M}$ is perfectly known a priori: true MDP.
  - If $\hat{\mathcal{M}} \approx \mathcal{M}$ needs to be learned: approximated MDP.

Model learning / identification

- In many real-world applications, a model might be too complex to derive or not exactly available. Hence, estimate a model $\hat{\mathcal{M}}$ from experience $\{X_0, U_0, R_1, \ldots, X_T\}$.
- This is a supervised learning / system identification task:

$$\{X_0, U_0\} \to \{X_1, R_1\}$$
$$\vdots$$
$$\{X_{T-1}, U_{T-1}\} \to \{X_T, R_T\}$$

- Simple tabular / look-up table approach (with $n(x, u)$ visit count):

$$\hat{p}^u_{xx'} = \frac{1}{n(x,u)} \sum_{k=0}^{T} 1(X_{k+1} = x' | X_k = x, U_k = u),$$
$$\hat{\mathcal{R}}^u_x = \frac{1}{n(x,u)} \sum_{k=0}^{T} 1(X_k = x | U_k = u) r_{k+1}. \tag{7.3}$$

Distribution vs. sample models

- A model based on $\mathcal{P}$ and $\mathcal{R}$ is called a distribution model.
  - Contains descriptions of all possibilities by random distributions.
  - Has full explanatory power, but is still rather complex to obtain.
- Alternatively, use sample models to receive realization series.
  - Remember black jack examples: easy to sample by simulation but hard to model a full distributional MDP.

Fig. 7.3: Depending on the application distribution models are easily available or not (source: Josh Appel on Unsplash)

Model-free RL

- Learn value functions and/or policy directly from experience.
- Requires no model at all (policy can be considered an implicit model).
- Solves control problems by learning algorithms such as
  - Monte-Carlo,
  - SARSA or
  - Q-learning.

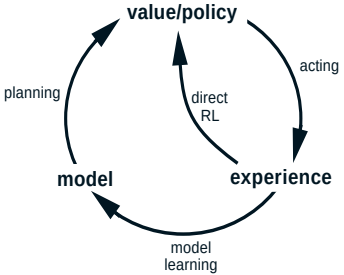


Fig. 7.4: If a perfect a priori model is not available, RL can be realized directly or indirectly (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Advantages & drawbacks: model-free vs. model-based RL

Pro model-based / indirect RL:

- Efficiently uses limited amount of experience (e.g., by replay).
- Allows integration of available a priori knowledge.
- Learned models might be re-used for other tasks (e.g., monitoring).

Pro model-free / direct RL:

- Is simpler to implement (only one task, not two consequent ones).
- Not affected by model bias / error during model learning.

Fig. 7.5: What way is better? (source: Mike Kononov on Unsplash)

Table of contents

The general Dyna architecture (1)

- Proposed by R. Sutton in 1990's
- General framework with many different implementation variants

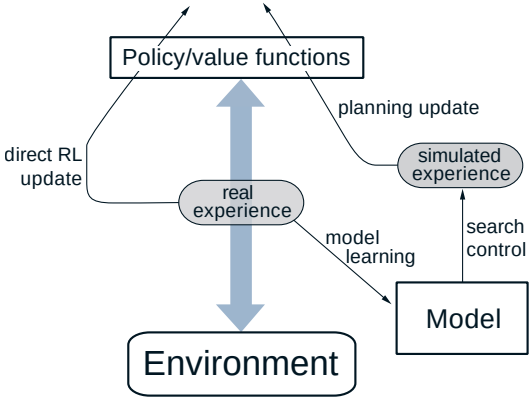
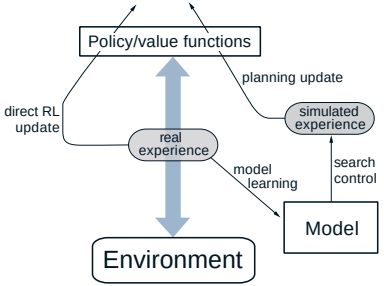


Fig. 7.6: Dyna framework (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

The general Dyna architecture (2)

- Direct RL update: any model-free algorithm: Q-learning, SARSA, ...
- Model learning:
  - In tabular case: simple distribution estimation as in (7.3)
  - Simple experience buffer to re-apply model-free algorithm
  - For large or continuous state/action spaces: function approximation by supervised learning / system identification (next lecture)
- Search control: strategies for selecting starting states and action to generate simulated experience



Algorithmic implementation: Dyna-Q

parameter: $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $\quad n \in \{\mathbb{N}|n \geq 1\}$ (planning steps per real step)
init: $\hat{q}(x,u)$ arbitrary (except terminal) and $\hat{\mathcal{M}}(x,u)\ \forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$
for $j = 1, 2, \ldots$ *episodes* **do**
    Initialize x_0;
    $k \leftarrow 0$;
    repeat
        Choose u_k from x_k using a soft policy derived from $\hat{q}(x,u)$;
        Take action u_k, observe r_{k+1} and x_{k+1};
        $\hat{q}(x_k,u_k) \leftarrow \hat{q}(x_k,u_k) + \alpha\left[r_{k+1} + \gamma \max_u \hat{q}(x_{k+1},u) - \hat{q}(x_k,u_k)\right]$;
        $\hat{\mathcal{M}}(x_k,u_k) \leftarrow \{r_{k+1}, x_{k+1}\}$ (assuming deterministic env.);
        for $i = 1, 2, \ldots$ *n* **do**
            $\tilde{x}_i \leftarrow$ random previously visited state;
            $\tilde{u}_i \leftarrow$ random previously taken action in $\tilde{x}_i$;
            $\{\tilde{r}_{i+1}, \tilde{x}_{i+1}\} \leftarrow \hat{\mathcal{M}}(\tilde{x}_i, \tilde{u}_i)$;
            $\hat{q}(\tilde{x}_i,\tilde{u}_i) \leftarrow \hat{q}(\tilde{x}_i,\tilde{u}_i) + \alpha\left[\tilde{r}_{i+1} + \gamma \max_u \hat{q}(\tilde{x}_{i+1},u) - \hat{q}(\tilde{x}_i,\tilde{u}_i)\right]$;
        $k \leftarrow k + 1$;
    until x_k *is terminal*;

Algo. 7.1: Dyna with Q-learning (Dyna-Q)

Remarks on Dyna-Q implementation

The specific Dyna-Q characteristics are:

- Direct RL update: Q-learning,
- Model: simple memory buffer of previous real experience,
- Search strategy: random choices from model buffer.

Moreover:

- Number of Dyna planning steps n is to be delimited from n-step bootstrapping (same symbol, two interpretations).
- Without the model $\hat{\mathcal{M}}$ one would receive one-step Q-learning.
- The model-based learning is done n times per real environment interaction:
  - Previous real experience is re-applied to Q-learning.
  - Can be considered a background task: choose $\max n$ s.t. hardware limitations (prevent turnaround errors).
- For stochastic environments: use a distributional model as in (7.3).
  - Update rule then may be modified from sample to expected update.

Maze example (1)

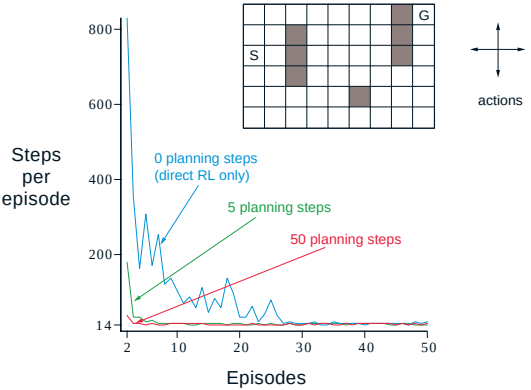


Fig. 7.7: Applying Dyna-Q with different planning steps n to simple maze (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- Maze with obstacles (gray blocks)
- Start at S and reach G
- $r_T = +1$ at G
- Episodic task with $\gamma = 0.95$
- Step size $\alpha = 0.1$
- Exploration $\varepsilon = 0.1$
- Averaged learning curves

Maze example (2)

- Blocks without an arrow depict a neutral policy (equal action values).
- Black squares indicate agent's position during second episode.
- Without planning ($n = 0$), each episodes only adds one new item to the policy.
- With planning ($n = 50$), the available experience is efficiently utilized.
- After the third episode, the planning agent found the optimal policy.

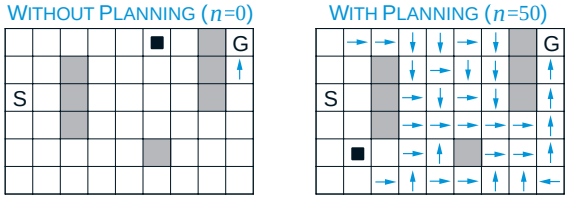


Fig. 7.8: Policies (greedy action) for Dyna-Q agent halfway through second episode (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

The shortcut maze example

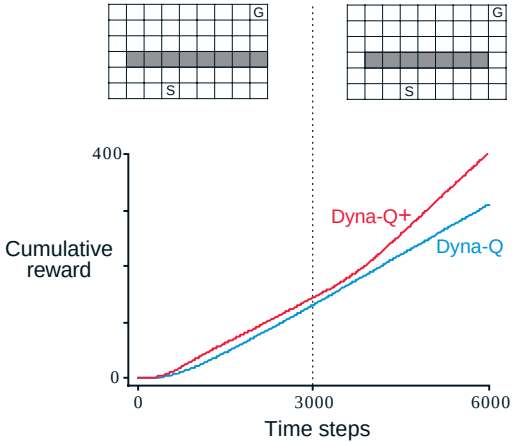


Fig. 7.9: Maze with an additional shortcut after 3000 steps (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- Maze opens a shortcut after 3000 steps
- Start at S and reach G
- $r_T = +1$ at G
- Dyna-Q with random exploration is likely not finding the shortcut
- Dyna-$Q+$ exploration strategy is able to correct internal model
- Averaged learning curves

Dyna-$Q+$ extensions

Compared to default Dyna-Q in Algo. 7.1, Dyna-$Q+$ contains the following extensions:

- Search heuristic: add $\kappa\sqrt{\tau}$ to regular reward.
  - τ: is the number of time steps a state-action transition has not been tried.
  - κ: is a small scaling factor $\kappa \in \{\mathbb{R}|0 < \kappa\}$.
  - Agent is encouraged to keep testing all accessible transitions.
- Actions for given states that had never been tried before are allowed for simulation-based planning.
  - Initial model for that: actions lead back to same state without reward.

Table of contents

Background and idea

- Dyna-Q randomly samples from the memory buffer.
  - Many planning updates maybe pointless, e.g., zero-valued state updates during early training.
  - In large state-action spaces: inefficient search since transitions are chosen far away from optimal policies.

- Better: focus on important updates.
  - In episodic tasks: backward focusing starting from the goal state.
  - In continuing tasks: prioritize according to impact on value updates.

- Solution method is called prioritized sweeping.
  - Build up a queue of every state-action pair whose value would change significantly.
  - Prioritize updates by the size of change.
  - Neglect state-action pairs with only minor impact.

Algorithmic implementation: prioritized sweeping

parameter: $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $\quad n \in \{\mathbb{N}|n \geq 1\}$, $\quad \theta \in \{\mathbb{R}|\theta \geq 0\}$
init: $\hat{q}(x,u)$ arbitrary and $\hat{\mathcal{M}}(x,u)\ \forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$, empty queue $\mathcal{Q}$
for $j = 1, 2, \ldots$ *episodes* **do**
    Initialize x_0 and $k \leftarrow 0$;
    repeat
        Take u_k from x_k using a soft policy derived from $\hat{q}(x,u)$, observe r_{k+1} and x_{k+1};
        $\hat{\mathcal{M}}(x_k, u_k) \leftarrow \{r_{k+1}, x_{k+1}\}$ (assuming deterministic env.);
        $P \leftarrow |r_{k+1} + \gamma \max_u \hat{q}(x_{k+1}, u) - \hat{q}(x_k, u_k)|$;
        if $P > \theta$ **then** insert $\{x_k, u_k\}$ in $\mathcal{Q}$ with priority P;
        for $i = 1, 2, \ldots$ *n while queue* $\mathcal{Q}$ *is not empty* **do**
            $\{\tilde{x}_i, \tilde{u}_i\} \leftarrow \arg\max_P(\mathcal{Q})$;
            $\{\tilde{r}_{i+1}, \tilde{x}_{i+1}\} \leftarrow \hat{\mathcal{M}}(\tilde{x}_i, \tilde{u}_i)$;
            $\hat{q}(\tilde{x}_i, \tilde{u}_i) \leftarrow \hat{q}(\tilde{x}_i, \tilde{u}_i) + \alpha\,[\tilde{r}_{i+1} + \gamma \max_u \hat{q}(\tilde{x}_{i+1}, u) - \hat{q}(\tilde{x}_i, \tilde{u}_i)]$;
            for $\forall\,\{\overline{x}, \overline{u}\}$ *predicted to lead to* $\tilde{x}_i$ **do**
                $\overline{r} \leftarrow$ predicted reward for $\{\overline{x}, \overline{u}, \tilde{x}_i\}$;
                $P \leftarrow |\overline{r} + \gamma \max_u \hat{q}(\tilde{x}_i, u) - \hat{q}(\overline{x}, \overline{u})|$;
                if $P > \theta$ **then** insert $\{\overline{x}, \overline{u}\}$ in $\mathcal{Q}$ with priority P;
        $k \leftarrow k + 1$;
    until x_k *is terminal*;

Remarks on prioritized sweeping implementation

The specific prioritized sweeping characteristics are:

- Direct RL update: Q-learning,
- Model: simple memory buffer of previous real experience,
- Search strategy: prioritized updates based on predicted value change.

Moreover:

- θ is a hyperparameter denoting the update significance threshold.
- Prediction step regarding $\tilde{x}_i$ is a backward search in the model buffer.
- For stochastic environments: use a distributional model as in (7.3).
  - Update rule then may be modified from sample to expected update.

Comparing against Dyna-Q on simple maze example

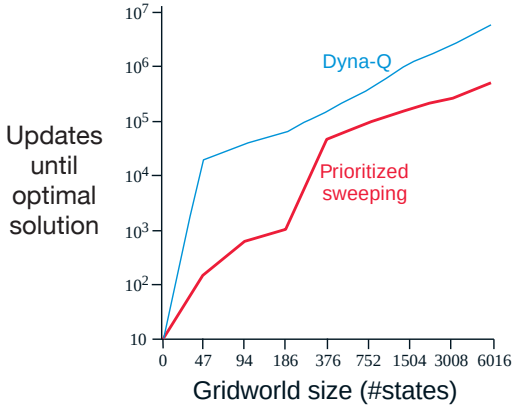


Fig. 7.10: Comparison of prioritized sweeping and Dyna-Q on simple maze (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- Environment framework as in Fig. 7.7
- But: changing maze sizes (number of states)
- Both methods can utilize up to $n = 5$ planning steps
- Prioritized sweeping finds optimal solution 5-10 times quicker

Table of contents

Background planning vs. planning at decision time

Background Planning (discussed so far):

- Gradually improves policy or value function if time is available.
- Backward view: re-apply gathered experience.
- Feasible for fast execution: policy or value estimate are available with low latency (important, e.g., for real-time control).

Planning at decision time[1] (not yet discussed alternative):

- Select single next future action through planning.
- Forward view: predict future trajectories starting from current state.
- Typically discards previous planning outcomes (start from scratch after state transition).
- If multiple trajectories are independent: easy parallel implementation.
- Most useful if fast responses are not required (e.g., turn-based games).

[1] Can be interpreted as *model predictive control* in an engineering context.

Heuristic search

- Develop tree-like continuations from each state encountered.
- Approximate value function at leaf nodes (using a model) and back up towards the current state.
- Choose action according to predicted trajectory with highest value.
- Predictions are normally discarded (new search tree in each state).

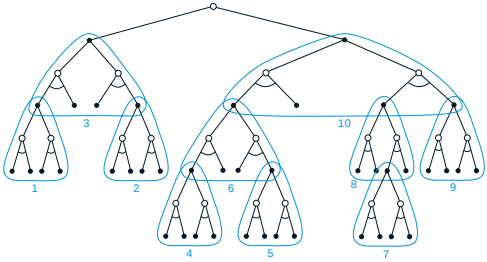


Fig. 7.11: Heuristic search tree with exemplary order of back-up operations (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Rollout algorithms

- Similar to heuristic search, but: simulate trajectories following a rollout policy.
- Use Monte Carlo estimates of action value only for current state to evaluate on best action.
- Gradually improves rollout policy but optimal policy might not be found if rollout sequences are too short.
- Predictions are normally discarded (new rollout in each state).

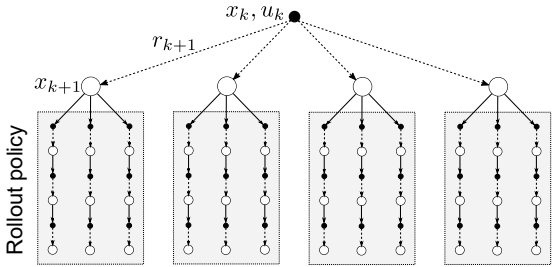


Fig. 7.12: Simplified processing diagram of rollout algorithms

Monte Carlo tree search (MCTS)

- Rollout algorithm, but:
  - accumulates values estimates from former MC simulations,
  - makes use of an informed tree policy (e.g., ε-greedy).

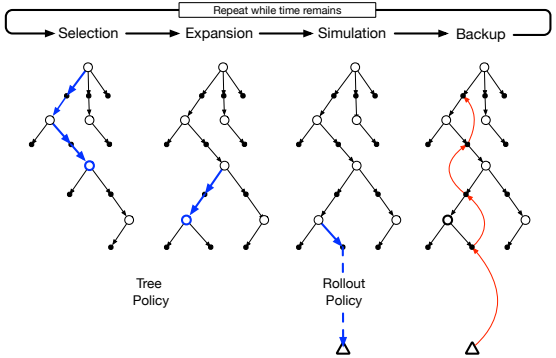


Fig. 7.13: Basic building blocks of MCTS algorithms (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Basic MCTS procedure

Repeat the following steps while prediction time is available:

1. Selection: Starting at root node, use a tree policy (e.g., ε-greedy) to travel through the tree until arriving at a leaf node.
   - The tree policy exploits auspicious tree regions while maintaining some exploration.
   - It is improved and (possibly) extended in every simulation run.
2. Expansion: Add child node(s) to the leaf node by evaluating unexplored actions (optional step).
3. Simulation: Simulate the remaining full episode using the rollout policy starting from the leaf or child node (if available).
   - The rollout policy could be random, pre-trained or based on model-free methods using real experience (if available).
4. Backup: Update the values along the traveled trajectory but only saves those within the tree policy.

Further MCTS remarks

What is happening after reaching the feasible simulation runs?

- After time is up, MCTS picks an appropriate action regarding the root node, e.g.:
  - The action visited the most times during all simulation runs or
  - The action having the largest action value.
- After transitioning to a new state, the MCTS procedure re-starts:
  - Either with a new tree incorporating only the root node or
  - by re-utilizing the applicable parts from the previous tree.

Further reading on MCTS:

- MCTS-based algorithms are not limited to game applications but were able to achieve outstanding success in this field.
  - Famous AlphaGo (cf. Keynote lecture from D. Silver)
- More in-depth lectures on MCTS can be found (among others) here:
  - Stanford Online: CS234
  - MIT OpenCourseWare
  - Extensive slide set from M. Sebag at Universite Paris Sud

Summary: what you've learned today

- Model-free RL is easy to implement and cannot suffer any model learning error while model-based approaches use a limited amount of experience much more efficient.
- Integrating these two RL branches can be achieved using the Dyna framework (background planning) incorporating the steps:
  - Direct RL updates (any model-free approach, e.g., Q-learning),
  - Model learning: use real experience to improve model predictions,
  - Search control: strategies on how to generate simulated experience.
- The Dyna framework allows many different algorithms such as Dyna-$Q(+)$ or prioritized sweeping.
  - Learning efficiency is much increased compared to pure model-based/free approaches.
  - Many degrees of freedom regarding internal update rules exist.
- In contrast, planning at decision time predicts future trajectories starting from the current state (forward view).
  - Rather computationally expensive leading to high latency responses.
  - The Monte Carlo tree search rollout algorithm is a well-known example.

Summary of Part I: Reinforcement Learning in Finite State and Action Spaces

André Bodmer

Common key ideas to all discussed rl methods so far

1. Estimating and comparing value functions
2. Backing up values along actual or possible state trajectories
3. Usage of GPI mechanism to maintain an approximate value function and policy trying to improve each of them on the basis of the other

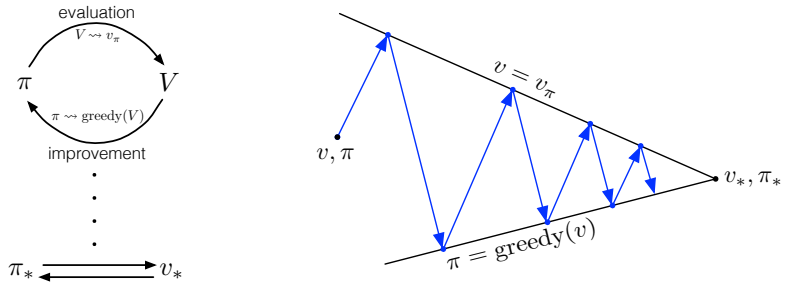


Fig. S-I.1: Generalized policy iteration (GPI) as a mutual building block of all previously discussed RL methods (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC

two important rl dimensions: update depth and width

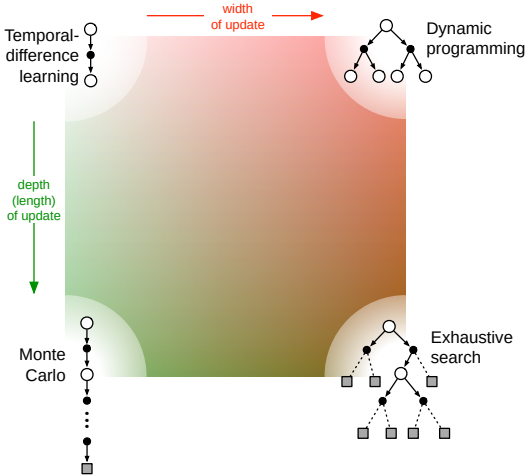


Fig. S-I.2: A slice through the RL method space (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Other important rl dimensions

Selected, non-exhaustive list:

- Problem space: How many states and actions? Stochastic vs. deterministic environment? Stationary?
- Policy objective: on-policy vs. off-policy? Explicit vs. implicit policy?
- Task: Episodic vs. continuing?
- Return definition: Discounting? General reward design?
- Value: State vs. action value estimation?
- Model: Required? Distribution vs. sample models? Learning vs. a priori (expert) knowledge?
- Exploration: How to search for new policies?
- Update order: synchronous vs. asynchronous? If latter, which order?
- Experience: simulated vs. real experience? Memory length and style?
- ...

Outlook

First part of the course:

Reinforcement learning on small finite action and state spaces

The problem space is such small that RL methods based on look-up tables are applicable.

Second part of the course::

Reinforcement learning using function approximators

The problem space is either continuous or contains an unfeasible large amount of discrete state-action pairs. Value estimates, models or explicit policies stored in look-up tables would let the memory demand explode. Modifications and extensions of available RL algorithms using function approximators are required.