Reinforcement learning

Oliver Wallscheid

Universität Siegen / IAS Interconnected Automation Systems

CC BY NC SA

Table of contents

1. Introduction to reinforcement learning
2. Markov decision processes
3. Dynamic programming
4. Monte Carlo methods
5. Temporal-difference learning
6. Multi-step bootstrapping
7. Planning and learning with tabular methods
8. Summary of part I: finite state and action spaces
9. Supervised learning
10. Approximative on-policy prediction
11. Approximative value-based control
12. Stochastic policy gradient methods
13. Deterministic policy gradient methods
14. Further contemporary RL algorithms
15. Outlook and practical research insights
16. Summary of part II: continuous state and action spaces

Table of contents

1 Introduction to reinforcement learning
- Course framework
- Reinforcement learning: what is it?
- Application examples and historic review
- Basic terminology
- Main categories of reinforcement learning algorithms
- Small comparison to model predictive control

The teaching team

Ali
Abdelwanis

Oliver
Wallscheid

Contact

- Email: see chair's homepage
- Offices: H-A building, 4th floor
- Individual appointments on request (remote or personally)

The legacy team

Barnabas Haucke-Korber

Darius Jakobeit

Wilhelm Kirchgässner

Marvin Meyer

Maximilian Schenke

Hendrik Vater

Daniel Weber

Credits

This course and its content was originally created by a team at Paderborn University. Many thanks to the above staff members for their contributions to the creation and maintenance of the materials since the first course run in 2020.

Examination regulations

Final exam

- Oral examination
- Average 45 minutes for presentation and discussion
- Individual appointment request via email (at least 2 weeks in advance)

Pre-exam homework assignment

- Will be made available via moodle at end of the lecture series.
- Practical RL programming task, i.e., solve a typical RL problem.
- Further regulations:
  - Submit your final programming solution via moodle at least two days before the exam.
  - Prepare a concise, high-quality presentation to be given at the exam start (roughly 10-15 minutes). Analyze and evaluate your own results critically.

Recommended textbooks

- Reinforcement learning: an introduction,
  - R. Sutton and G. Barto
  - MIT Press, 2nd edition, 2018
  - Available here

- Reinforcement learning (lecture script)
  - D. Silver
  - Entire slide set available here
  - YouTube lecture series (click here)

- Reinforcement learning and optimal control
  - D. Bertsekas
  - Athena Scientific, 2019

Table of contents

1 Introduction to reinforcement learning
- Course framework
- Reinforcement learning: what is it?
- Application examples and historic review
- Basic terminology
- Main categories of reinforcement learning algorithms
- Small comparison to model predictive control

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)

Note: the nomenclature of this slide set is based on the default variable usage in control theory. In 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 refers to consequences that increase the likelihood of an organism's future behavior, typically in the presence of a particular antecedent stimulus.[...] Reinforcers serve to increase behaviors whereas punishers serve to decrease behaviors; thus, positive reinforcers are stimuli that the subject will work to attain, and negative reinforcers are stimuli that the subject will work to be rid of or to end.*", source: wikipedia.org (obtained 2025-03-19)

What is learning?

"*Acquiring knowledge and skills and having them readily available from memory so you can make sense of future problems and opportunities.*", source: 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

1 Introduction to reinforcement learning
- Course framework
- Reinforcement learning: what is it?
- Application examples and historic review
- Basic terminology
- Main categories of reinforcement learning algorithms
- Small comparison to model predictive control

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)[1]

Optimal sequential decision making

[1] Illustrative picture since an actual photo of Bellman is not freely available.

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
- Nuclear fusion reactor plasma control
- Training chat bots (like chatGPT)
- ...

Table of contents

1 Introduction to reinforcement learning
- Course framework
- Reinforcement learning: what is it?
- Application examples and historic review
- Basic terminology
- Main categories of reinforcement learning algorithms
- Small comparison to model predictive control

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., $\boldsymbol{y}_k = \begin{bmatrix} \boldsymbol{I} & \boldsymbol{0} \end{bmatrix} \boldsymbol{x}_k$).
- 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.

Deterministic policy example

Find optimal gains $\{K_\mathrm{p}, K_\mathrm{i}, K_\mathrm{d}\}$ given the reward $r_k = -e_k^2$:

- Agent's behavior is explicitly determined by $\{K_\mathrm{p}, K_\mathrm{i}, K_\mathrm{d}\}$.
- Reference value is part of the environment state: $\boldsymbol{x}_k = \begin{bmatrix} y_k & y_k^* \end{bmatrix}^\mathsf{T}$.
- Control output is the agent's action: $\boldsymbol{u}_k = \boldsymbol{\pi}(\boldsymbol{x}_k | K_\mathrm{p}, K_\mathrm{i}, K_\mathrm{d})$.

Fig. 1.9: Classical PID control loop with scalar quantities (derivative of www.wikipedia.org, by Arturo Urquizo CC BY-SA 3.0)

Stochastic policy example

Two-player game of extended rock-paper-scissors:

- A deterministic policy can be easily exploited by the opponent.
- A uniform random policy would be instead unpredictable (assuming an ideal random number generator).

Fig. 1.10: Rock paper scissors lizard Spock game mechanics
(source: www.wikipedia.org, by Diriector Doc CC BY-SA 4.0)

Value functions

- The state-value function is the expected return being in state $\boldsymbol{x}_k$ following $\boldsymbol{\pi}$: $v_\pi(\boldsymbol{x}_k)$.
- 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. 1.11: 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

1 Introduction to reinforcement learning
- Course framework
- Reinforcement learning: what is it?
- Application examples and historic review
- Basic terminology
- Main categories of reinforcement learning algorithms
- Small comparison to model predictive control

Maze example

Fig. 1.12: 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. 1.13: 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. 1.14: 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. 1.15: 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. 1.16: Main categories of reinforcement learning algorithms
(source: D. Silver, "Reinforcement learning", 2015. CC BY-NC 4.0)

Table of contents

1 Introduction to reinforcement learning
- Course framework
- Reinforcement learning: what is it?
- Application examples and historic review
- Basic terminology
- Main categories of reinforcement learning algorithms
- Small comparison to model predictive control

RL vs. planning

Two fundamental solutions to sequential decision making:

- Reinforcement learning:
  - The environment is initially unknown.
  - The agents interacts with the environment.
  - The policy is improved based on environment feedback (reward).
- Planning:
  - An a priori environment model exists.
  - The agents interacts with its own model.
  - The policy is improved based on the model feedback ('virtual reward').

Remark on learning and planning

Above the two extreme cases are confronted:

- RL = learning based on data obtained from interacting with the system.
- Planning = iterating on a model without improving it based on data.

Can one of these extreme cases lead alone to an efficient and optimal solution?

Problem reconsideration

The reward hypothesis in Theo. 1.1 is basically an (discounted) infinite-horizon optimal control problem (with r_k interpreted as costs):

$$v_k^*(\boldsymbol{x}_k) = \min_{\boldsymbol{u}_k} \sum_{i=0}^{\infty} \gamma^i r_{k+i+1}(\boldsymbol{x}_{k+i}, \boldsymbol{u}_{k+i}) \,. \tag{1.16}$$

For certain cases closed-form solutions can be found, e.g., a LTI system with quadratic costs and no further constraints can be optimally controlled by a linear-quadratic regulator (LQR). However, for arbitrary cases that is not possible and one relaxes the problem to a finite-horizon optimal control problem:

$$v_k^*(\boldsymbol{x}_k) = \min_{\boldsymbol{u}_k} \sum_{i=0}^{N_\mathrm{p}} \gamma^i r_{k+i+1}(\boldsymbol{x}_{k+i}, \boldsymbol{u}_{k+i}) \,. \tag{1.17}$$

Here, an internal model $\boldsymbol{x}_{k+1} = \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k)$ is utilized to predict the system behavior for N_p future steps. This model predictive control (MPC) approach can be numerically solved.

MPC and constraints

While in RL the desired system behavior must be solely represented by r_k, MPC can directly take into account system constraints:

$$v_k^* = \min_{\boldsymbol{u}_k} \sum_{i=0}^{N_\mathrm{p}} \gamma^i r_{k+i+1}(\boldsymbol{x}_{k+i}, \boldsymbol{u}_{k+i}) \,, \tag{1.18}$$

$$\text{s.t.} \qquad \boldsymbol{x}_{k+i+1} = \boldsymbol{f}(\boldsymbol{x}_{k+i}, \boldsymbol{u}_{k+i}), \quad \boldsymbol{x}_{k+i} \in \mathcal{X}, \quad \boldsymbol{u}_{k+i} \in \mathcal{U} \,.$$

Fig. 1.17: Basic MPC scheme (source: www.wikipedia.org, by Martin Behrendt CC BY-SA 3.0)

MPC vs. RL

Hence, MPC and RL are two sides of the same coin. Both share the same general goal (sequential optimal decision making), but follow their own philosophy:

Property	MPC	RL
Objective	minimize costs	maximize return
A priori model	required ✗	not required ✓
Pre-knowledge integration	easy ✓	rather complex ✗
Constraint handling	inherent ✓	only indirect ✗
Adaptivity	requires add-ons ✗	inherent ✓
Online complexity	it depends ✓/✗	it depends ✓/✗
Stability theory	mature ✓	immature ✗

Tab. 1.1: Key differences on MPC vs. RL
(inspired from D. Görges, *Relations between Model Predictive Control and Reinforcement Learning*, IFAC PapersOnine 50-1, pp. 4920-4928, 2017)

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.

Table of contents

2 Markov decision processes
- Finite Markov chains
- Finite Markov reward processes
- Finite Markov decision processes
- Optimal policies and value functions

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.

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.
- With α probability a natural hazard destroys the tree.
- If it reaches $x = 3$ (large) its growth is limited.
- 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

2 Markov decision processes
- Finite Markov chains
- Finite Markov reward processes
- Finite Markov decision processes
- Optimal policies and value functions

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

2 Markov decision processes
- Finite Markov chains
- Finite Markov reward processes
- Finite Markov decision processes
- Optimal policies and value functions

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.

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:

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

$$\begin{bmatrix} v_1^{\pi} \\ \vdots \\ v_n^{\pi} \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1^{\pi} \\ \vdots \\ \mathcal{R}_n^{\pi} \end{bmatrix} + \gamma \begin{bmatrix} p_{11}^{\pi} & \cdots & p_{1n}^{\pi} \\ \vdots & & \vdots \\ p_{n1}^{\pi} & \cdots & p_{nn}^{\pi} \end{bmatrix} \begin{bmatrix} v_1^{\pi} \\ \vdots \\ v_n^{\pi} \end{bmatrix}. \tag{2.19}$$

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*')

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

Applied to the given environment behavior

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

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

one receives:

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

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 from (2.16) the action values can be directly calculated:

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

Table of contents

2 Markov decision processes
- Finite Markov chains
- Finite Markov reward processes
- Finite Markov decision processes
- Optimal policies and value functions

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.

Table of contents

3 Dynamic programming
- Policy evaluation
- Policy improvement
- Policy and value iteration
- Further aspects

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.

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

3 Dynamic programming
- Policy evaluation
- Policy improvement
- Policy and value iteration
- Further aspects

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

3 Dynamic programming
- Policy evaluation
- Policy improvement
- Policy and value iteration
- Further aspects

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

3 Dynamic programming
- Policy evaluation
- Policy improvement
- Policy and value iteration
- Further aspects

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.

Table of contents

4 Monte Carlo methods
- Basic Monte Carlo prediction
- Basic Monte Carlo control
- Extensions to Monte Carlo on-policy control
- Monte Carlo off-policy prediction and control

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)

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 $\mathbb{E}\left[\hat{v}_\pi\right]$ and consequently this is bias-free.
- The estimate's variance $\mathrm{Var}\left[\hat{v}_\pi\right]$ 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 $\mathbb{E}\left[\hat{v}_\pi\right]$ and consequently this is biased for any $n < \infty$.
- Only in the limit $n \to \infty$ one receives $\left(v_\pi(x) - \mathbb{E}\left[\hat{v}_\pi(x)\right]\right) \to 0$.

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

4 Monte Carlo methods
- Basic Monte Carlo prediction
- Basic Monte Carlo control
- Extensions to Monte Carlo on-policy control
- Monte Carlo off-policy prediction and control

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

4 Monte Carlo methods
- Basic Monte Carlo prediction
- Basic Monte Carlo control
- Extensions to Monte Carlo on-policy control
- Monte Carlo off-policy prediction and control

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:

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

- 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

4 Monte Carlo methods
- Basic Monte Carlo prediction
- Basic Monte Carlo control
- Extensions to Monte Carlo on-policy control
- Monte Carlo off-policy prediction and control

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:

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

- 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.

Table of contents

5 Temporal-difference learning
- Temporal-difference prediction
- Temporal-difference on-policy control: SARSA
- Temporal-difference off-policy control: Q-learning
- Maximization bias and double learning

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).

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 (5.1) one has to wait until the episode's termination to get g_k.

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 can learn from incomplete sequences, i.e., in continuing 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 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

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

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

Tab. 5.1: Example state-reward sequences for Fig. 5.8

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:

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

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

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:

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

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

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

5 Temporal-difference learning
- Temporal-difference prediction
- Temporal-difference on-policy control: SARSA
- Temporal-difference off-policy control: Q-learning
- Maximization bias and double learning

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: $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, $\quad \alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
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 $\alpha_{\mathrm{SARSA}} = 0.1$ 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

5 Temporal-difference learning
- Temporal-difference prediction
- Temporal-difference on-policy control: SARSA
- Temporal-difference off-policy control: Q-learning
- Maximization bias and double learning

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: $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, $\quad \alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
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

5 Temporal-difference learning
- Temporal-difference prediction
- Temporal-difference on-policy control: SARSA
- Temporal-difference off-policy control: Q-learning
- Maximization bias and double learning

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: $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, $\quad \alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
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?

Table of contents

6 Multi-step bootstrapping
- n-step TD Prediction
- n-step Control
- n-step Off-Policy Learning
- TD(λ)

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)

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 $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{v}(x_{\tau+n})$;
            $\hat{v}(x_\tau) \leftarrow \hat{v}(x_\tau) + \alpha \left[g - \hat{v}(x_\tau) \right]$;
    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

6 Multi-step bootstrapping
- n-step TD Prediction
- n-step Control
- n-step Off-Policy Learning
- TD(λ)

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

6 Multi-step bootstrapping
- n-step TD Prediction
- n-step Control
- n-step Off-Policy Learning
- TD(λ)

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:

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

- 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 $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{v}(x_{\tau+n})$;
            $\hat{v}(x_\tau) \leftarrow \hat{v}(x_\tau) + \alpha\rho \left[g - \hat{v}(x_\tau)\right]$;
    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 $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{q}(x_{\tau+n}, u_{\tau+n})$;
            $\hat{q}(x_\tau, u_\tau) \leftarrow \hat{q}(x_\tau, u_\tau) + \alpha\rho \left[g - \hat{q}(x_\tau, u_\tau)\right]$;
            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

6 Multi-step bootstrapping
- n-step TD Prediction
- n-step Control
- n-step Off-Policy Learning
- TD(λ)

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:

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

- 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:

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

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 $u_{k+1} \sim \pi(\cdot|x_{k+1})$
    $$z_k(x,u) \leftarrow \gamma\lambda z_{k-1}(x,u) + \begin{cases} 0, & \text{if } x_k \neq x \text{ or } u_k \neq u, \\ 1, & \text{if } x_k = x \text{ and } u_k = u. \end{cases} \quad \forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$$
    $\delta \leftarrow r_{k+1} + \gamma\hat{q}(x_{k+1}, u_{k+1}) - \hat{q}(x_k, u_k)$
    $\hat{q}(x,u) \leftarrow \hat{q}(x,u) + \alpha\delta z_k(x,u)\ \forall\ \{x \in \mathcal{X}, u \in \mathcal{U}\}$
    $k \leftarrow k + 1$;
  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.

Table of contents

7 Planning and learning with tabular methods
- Repetition: model-based and model-free RL
- Dyna: integrated planning, acting and learning
- Prioritized sweeping
- Planning at decision time

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)

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 $\{X_0, U_0, R_1, \dots, X_T\}$.
- 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):

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

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.

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

7 Planning and learning with tabular methods
- Repetition: model-based and model-free RL
- Dyna: integrated planning, acting and learning
- Prioritized sweeping
- Planning at decision time

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

7 Planning and learning with tabular methods
- Repetition: model-based and model-free RL
- Dyna: integrated planning, acting and learning
- Prioritized sweeping
- Planning at decision time

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: $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, $n \in \{\mathbb{N} | n \geq 1\}$, $\theta \in \{\mathbb{R} | \theta \geq 0\}$
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

7 Planning and learning with tabular methods
- Repetition: model-based and model-free RL
- Dyna: integrated planning, acting and learning
- Prioritized sweeping
- Planning at decision time

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.

Table of contents

Summary of part I

Reinforcement learning in finite state and action spaces

1. Introduction to reinforcement learning
2. Markov decision processes
3. Dynamic programming
4. Monte Carlo methods
5. Temporal-difference learning
6. Multi-step bootstrapping
7. Planning and learning with tabular methods

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 BY-NC-ND 2.0)

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.

Table of contents

9 Supervised learning
- Supervised learning problem statement
- Feature engineering
- Typical machine learning models
- Linear regression
- Artificial neural networks

The machine learning triad

Fig. 9.1: Disciplines of machine learning

Introductory material

Machine learning (ML) and especially the field of supervised learning (SL) is extensively researched and taught.

- Renowned online courses
  - *Coursera ML* by Stanford's Andrew Ng
  - *Practical deep learning for coders* by fast.ai
  - *Intro to ML* by Kaggle Courses
- Books classics
  - *Pattern Recognition and Machine Learning* by C. M. Bishop
  - *The Elements of Statistical Learning* by Hastie et al.
  - *Deep Learning* by I. Goodfellow, Y. Bengio, and A. Courville

Machine learning in industry

Machine learning applications are a fast growing industry itself, and enhance more and more automation in classical industry as well.
Among others, popular industries are:

- Embedded systems,
- Mobility, and
- Digital assistants

Most applications are of the supervised type.
The demand for highly skilled ML engineers is growing correspondingly.

Instances of ML applications

- Recommendation systems
  - Which ads to display on a website?
  - Which items are most likely put into cart next by the user?
- Forecasting
  - Weather, sales, geospatial Uber calls, restaurant/website traffic
  - Material attrition in engineering processes (predictive maintenance)
- Classification/Regression
  - Speech assistants (Alexa/Siri), pedestrial detection (autonomous driving), fault detection in engineering processes
  - large language models (LLM), credit scoring (fintech)
- Generative models

ML competitions with price pool

Fig. 9.2: Kaggle and DrivenData

Open ML competition platforms like kaggle or DrivenData offer a multitude of diverse competitions to participate in at no cost.

- Most competitions come with a decent price pool of 15 tsd. dollars up to 1 mil. dollars hosted by stakeholders from the industry and government.
- These competitions are almost exclusively of the supervised type, but RL challenges are increasing.

Typical supervised learning pipeline

Fig. 9.3: A typical supervised learning pipeline – sometimes more art than science

Supervised learning in reinforcement learning

SL approximates functions, RL approximates policies.

However, there are two situations where SL is auxiliary in RL:

- Function approximation of (action-)state values, if the number of possible states exceeds any reasonable memory capabability, which is often the case.
  - $v_\pi(x) \approx \hat{v}(x, \boldsymbol{w})$ with $\boldsymbol{w}$ being a trainable weight vector.
- Imitation learning. A simple-to-implement, deterministic baseline policy is often available, but an RL agent might fail to achieve that performance when learning from scratch. With SL, this baseline policy can be approximated to be the initial behavior of the agent.
  - Expert moves in board games.
  - Basic linear controllers in engineering applications with feedback-loops.

Supervised learning problem statement

Supervised learning

Given a labeled data set $\langle \boldsymbol{x}_k, \boldsymbol{y}_k \rangle \in \boldsymbol{\mathcal{D}}$ with $k \in [0, K-1]$ and K being the data set size, approximate the mapping function $f^* : \boldsymbol{x}_k \mapsto \boldsymbol{y}_k$ with a parameterizable ML model $f_{\boldsymbol{w}} : \boldsymbol{x}_k \mapsto \hat{\boldsymbol{y}}_k \approx \boldsymbol{y}_k \quad \forall k$.

- Goodness of fit can be measured by a manifold of metrics (e.g., mean squared error, classification accuracy, etc.).
- Reducing the look-up-table-like mapping f^* to a parameterized function $f_{\boldsymbol{w}}$ degrades any metric on the data set but enables interpolation to unseen data.
- The dimension ξ of model parameters $\boldsymbol{w} \in \mathbb{R}^{\xi}$ is adjustable in many model families, which trades off bias with variance (among other factors, leading to so-called under- and overfitting).
- On top of $\boldsymbol{w}$, an ML model might also have hyperparameters that can be optimized (e.g., number of layers in a neural network).

Bias and variance

Fig. 9.4: Left: Decision boundaries in binary classification, k-nearest neighbors with one (bright) and nine (dark) neighbors. Right: Regression example, least squares (dark) and 2-nearest neighbors (bright).

Generalization error

Supervised learning performance

SL performance is measured by a model's generalization error, i.e., goodness of fit on unseen data.

A data set is often finite as opposed to RL environments generating arbitrarily many observations.

- How to generate unseen data?
  - Hold out portions of the data set for cross-validation.

k-fold cross-validation

Fig. 9.5: k-fold CV with five folds

- Cross-validation (CV) can be conducted with k-fold CV.
- Training is repeated k times with k different splits of the training set.
- Each observation serves as unseen instance at least once.
- The validation error is an indicator for tuning hyperparameters.

Means to improve an SL model

SL performance can be improved by:

- Collecting more data, i.e., increasing K (more data is always better).
- Choosing a more appropriate model.
- Optimizing hyperparameters of the model.
- Averaging over several different models (ensembling).
- Most effectively: Revealing the most predictive patterns in the data to the model (feature engineering).

Table of contents

9 Supervised learning
- Supervised learning problem statement
- Feature engineering
- Typical machine learning models
- Linear regression
- Artificial neural networks

Feature engineering

Additional features might be:

- Coming from the real world via additional sensors or additional tracking mechanisms (think of a user's click behavior on a website)
- Hand-designed (*engineered*) by experts in the corresponding domain from the original feature set
- Automatically built according to properties of each feature in the original set (Auto-ML)

Caution

Adding more features is not equivalent to having more data (which is always better). Having a fixed data set size, adding arbitrarily many features, regardless of their origin, increases chances to align statistical fluctuations with the target $\boldsymbol{y}_k$ - overfitting is the result.

Feature engineering example (classification)

Fig. 9.6: Features $r = \sqrt{\text{width}^2 + \text{height}^2}$ and $\theta = \arctan\left(\frac{\text{height}}{\text{width}}\right)$ reveal linearly separable class distribution

Feature engineering example (regression)

Fig. 9.7: Log-transform of the target signal exhibits linear relationship to the regressor

Normalization

Most models require data to be normalized before training (apart from tree-based models). Typical normalizaton schemes:

- Standard scaling: $\tilde{\boldsymbol{x}} = (\boldsymbol{x} - \mathsf{Avg}(\boldsymbol{x}))/\mathsf{Std}(\boldsymbol{x})$
- Min-Max scaling: $\tilde{\boldsymbol{x}} = (\boldsymbol{x} - \min(\boldsymbol{x}))/(\max(\boldsymbol{x}) - \min(\boldsymbol{x}))$
- Plain scaling: $\tilde{\boldsymbol{x}} = \boldsymbol{x}/\max(|\boldsymbol{x}|)$

In an unnormalized data set, features with high variance will eclipse patterns in other features.

Data types

Several different data types can be utilized for ML:

- Binary: 1 or 0 (True or False).
- Integer: $\mathbb{N}$ (e.g., number of rooms in a building).
- Real-valued: $\mathbb{R}$ (e.g., temperature).
- Categorical: like {blue, green, red}
- Ordinal: Categoricals that can be ordered, e.g., educational experience (From elementary school to Ph.D.)

Data type specific normalization

How to normalize categorical data?

- *One-hot* encoding
  - Replace a categorical of n values with n binary features.
  - Feature space gets sparse and might get too big for memory.
- Mean target encoding
  - Replace each value of a categorical with the average (regression) or mode (classification) of the dependent variable being observed with the corresponding value.
  - This might lead to information *leaking* from the dependent variables into the independent variables, and might exhibit high performance that cannot be reproduced on unseen data.
- Entity embeddings
  - Let a neural network find a cardinality-constrained set of real-valued features for each categorical.
  - Works well in practice but is more intricate than alternatives.

Typical feature engineering schemes

Feature design is often of the following form (tricks of the trade):
Given K feature vectors $\boldsymbol{x}_k \in \mathbb{R}^P$ with, e.g., $P = 3$ (two real-valued regressors and a categorical independent variable $\boldsymbol{x}_k = (x_{k,r_1}, x_{k,r_2}, x_{k,c})$):

- $\tilde{x}_k = x_{k,r_1} + x_{k,r_2}$ (or any other combination, e.g., product, division, subtraction, also cf. Fig. 9.6),
- $\tilde{x}_k = x_{k,r} - \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_{i,r} \quad \forall r = \{r_1, r_2\}$ with $\mathcal{B} = \{i : x_{i,c} = x_{k,c}\}$,
- Clip/drop/aggregate outliers away,
- Coordinate transformations for spatial features (e.g., rotation),
- In time domain:
  - $\tilde{\boldsymbol{x}}_k = (x_{k,r_1}, x_{k-1,r_1}, x_{k-2,r_1}, x_{k,r_2}, x_{k,c})$ (lag features),
  - $\tilde{x}_k = (1 - \alpha)\tilde{x}_{k-1} + \alpha x_{k,r}$ (moving averages).
- In frequency domain:
  - Amplitude and index of frequencies from a fast fourier transform (FFT)

Table of contents

9 Supervised learning
- Supervised learning problem statement
- Feature engineering
- Typical machine learning models
- Linear regression
- Artificial neural networks

Model landscape

When trying to find an appropriate mapping between input and output data, one can choose from a variety of models:

- Linear/logistic regression (with regularization)
  - The simplest data-fitting algorithm
- Support vector machines (SVM)
  - Most popular algorithm before 2012
- (Deep) neural networks (DNN)
  - Also coined as *deep learning*, soared in popularity since 2012
  - Most prevalent in the domains of natural language processing (NLP) and image processing
- Gradient Boosting Machines (GBM)
  - Chaining of *weak* models (most of the time decision trees)
  - The best performing stand-alone model in tabular ML competitions

Model choice

Fig. 9.8: Choose models appropriate for the problem! (Source: Adapted from reddit)

Table of contents

9 Supervised learning
- Supervised learning problem statement
- Feature engineering
- Typical machine learning models
- Linear regression
- Artificial neural networks

Linear regression (1)

Linear models assume a linear relationship between $\boldsymbol{x}_k = (1, x_{k,1}, x_{k,2}, \ldots, x_{k,P})$ and y_k via trainable coefficients $\boldsymbol{w} \in \mathbb{R}^{P+1}$:

$$f(\boldsymbol{x}_k) = \hat{y}_k = w_0 + \sum_{p=1}^{P} x_{k,p} w_p, \tag{9.1}$$

$$\hat{\boldsymbol{y}} = \boldsymbol{\Xi} \boldsymbol{w}, \tag{9.2}$$

where $\boldsymbol{\Xi} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_K)$. Among other methods, $\boldsymbol{w}$ can be estimated from K samples by minimizing the residual sum of squares (RSS), which is coined the least squares method:

$$\mathrm{RSS}(\boldsymbol{w}) = \sum_{k=1}^{K} (y_k - f(\boldsymbol{x}_k))^2 = (\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w})^\mathsf{T} (\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w}). \tag{9.3}$$

Linear regression (2)

Deriving (9.3) with respect to $\boldsymbol{w}$ and setting it to zero while assuming $\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{\Xi}$ is positive-definite, yields an analytically closed solution form:

$$\hat{\boldsymbol{y}} = \boldsymbol{\Xi}\hat{\boldsymbol{w}} = \boldsymbol{\Xi}(\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{\Xi})^{-1}\boldsymbol{\Xi}^\mathsf{T}\boldsymbol{y}. \tag{9.4}$$

Multicollinearity

If two regressors exhibit strong linear correlation, their coefficients can grow indeterministically. This corresponds to high variance in $\hat{\boldsymbol{w}}$. Regularization of $\hat{\boldsymbol{w}}$ alleviates this effect - it induces bias for less variance. Most prevalent linear regularized techniques are LASSO and Ridge:

$$\mathsf{RSS}_{\mathsf{LASSO}}(\boldsymbol{w}) = (\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w})^\mathsf{T}(\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w}) + \lambda||\boldsymbol{w}||_1, \tag{9.5}$$

$$\mathsf{RSS}_{\mathsf{Ridge}}(\boldsymbol{w}) = (\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w})^\mathsf{T}(\boldsymbol{y} - \boldsymbol{\Xi}\boldsymbol{w}) + \lambda||\boldsymbol{w}||_2, \tag{9.6}$$

where λ controls the growth penalty.

Table of contents

9 Supervised learning
- Supervised learning problem statement
- Feature engineering
- Typical machine learning models
- Linear regression
- Artificial neural networks

Artificial neural networks

Artificial neural networks (ANNs) describe nonlinear approximators $\hat{\boldsymbol{y}} = f(\boldsymbol{\Xi}; \boldsymbol{w})$ that are end-to-end differentiable.

Fig. 9.9: A typical neuron as the key building block of ANNs.

- An ANN consists of nodes or neurons in one or more layers.
- Each node transforms the weighted sum of all previous nodes through an activation function.
- The weighted connections are called edges, which represent the ANN's parameters.

Multi-layer perceptron

A vanilla ANN is the so-called feed-forward ANN or multi-layer perceptron.

Fig. 9.10: Multi-layer perceptron.

- Only forward-flowing edges.
- The depth L and width $H^{(l)}$ are hyperparameters.

With $\varphi^{(l)}$ and $\mathcal{Z}^{(l)}$ denoting the activation function and activation of layer l respectively, we get for the output matrix $\mathcal{H}^{(l)}$

$$\mathcal{H}^{(l)} = \varphi^{(l)}\big(\underbrace{\mathcal{H}^{(l-1)}\mathcal{W}^{(l)} + \boldsymbol{b}^{(l)}}_{\mathcal{Z}^{(l)}}\big).$$

Weight matrix $\mathcal{W}^{(l)} \in \mathbb{R}^{H^{(l-1)} \times H^{(l)}}$ and (broadcasted) bias matrix $\boldsymbol{b}^{(l)} \in \mathbb{R}^{K \times H^{(l)}}$ are iteratively optimized and denote the full set of parameters $\boldsymbol{w}$.

Activation functions

Within hidden layers most prevalent activation functions $\varphi(\cdot)$ are

- $h = \tanh(z)$
- $h = \frac{1}{1+e^{-z}}$ (sigmoid)
- $h = \max(0, z)$
  (rectified linear unit (ReLU))

Fig. 9.11: Common activation functions

Whereas $\varphi^{(L)}(\cdot)$ is task-dependent:

- Regression: $\hat{y} = h^{(L)} = z^{(L)}$
- Binary classification: sigmoid
- Multi-class classification:

$$h_c^{(L)} = \frac{e^{z_c}}{\sum_{i=1}^{C} e^{z_i}} \quad \text{(softmax)}$$

Training neural networks (1)

ANN parameters are usually iteratively optimized via a variant of gradient descent, e.g., stochastic gradient descent (SGD).

$$\boldsymbol{\mathcal{W}}^{(l)} \leftarrow \boldsymbol{\mathcal{W}}^{(l)} - \alpha \nabla_{\boldsymbol{\mathcal{W}}^{(l)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}), \tag{9.7}$$

$$\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \alpha \nabla_{\boldsymbol{b}^{(l)}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}), \tag{9.8}$$

with α being the step size and $\mathcal{L}(\cdot)$ denoting the loss between the ground truth vector and the estimation vector.
Typical loss functions:

- Regression: (root) mean squared error (RMSE), mean absolute error
- Classification: Cross-entropy (CE)

Several iterations over the data set $\boldsymbol{\mathcal{D}}$ are called epochs.

Training neural networks (2)

Fig. 9.12: BGD vs. SGD

Gradient descent alternatives:

- Batch gradient descent (BGD): Average gradients over all samples, then update weights.
- Stochastic gradient descent (SGD): Update weights after each sample.

SGD is more computationally efficient, but steps are more random.
Nowadays, mini-batch gradient descent (mix of SGD and BGD) and further improvements are used, e.g., momentum and second derivatives, to ensure faster convergence to better optima.

Training neural networks (3)

How to retrieve the gradients:

Recall chain rule for vector derivatives, e.g., with $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$ where $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$:

$$\nabla_{\boldsymbol{x}} z = \frac{\partial z}{\partial \boldsymbol{x}} = \underbrace{\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\mathsf{T}}}_{\text{Jacobian of g}} \cdot \underbrace{\frac{\partial z}{\partial \boldsymbol{y}}}_{\text{gradient}} = \sum_j \frac{\partial y_j}{\partial \boldsymbol{x}} \cdot \frac{\partial z}{\partial y_j}. \tag{9.9}$$

This can be used equivalently for matrices/tensors of any shape $\nabla_{\boldsymbol{\Xi}} y = \frac{\partial y}{\partial \boldsymbol{\Xi}}$ when we assume to enumerate each element of the tensor consecutively and loop through them.

Error Backpropagation

After a forward step through the network, make a backward step in which the gradient $\boldsymbol{\gamma}$ of the loss $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is computed w.r.t the ANN's parameters from the output layer back to the input layer.

Training neural networks (4)

init: $\mathcal{H}^{(0)} \leftarrow \Xi$
// forward propagation
for $l = 1, \ldots, L$ *layers* **do**
  $\mathcal{Z}^{(l)} \leftarrow \mathcal{H}^{(l-1)}\mathcal{W}^{(l)} + \boldsymbol{b}^{(l)}$
  $\mathcal{H}^{(l)} \leftarrow \varphi^{(l)}(\mathcal{Z}^{(l)})$
// backward propagation
$\boldsymbol{\gamma} \leftarrow \nabla_{\boldsymbol{h}^{(L)}}\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ // note that $\boldsymbol{h}^{(L)} = \hat{\boldsymbol{y}}$
for $l = L, \ldots, 1$ *layers* **do**
  $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} \odot \partial(\varphi^{(l)})(\mathcal{Z}^{(l)}) = \nabla_{\mathcal{Z}^{(l)}}\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ // $\odot$: elementwise mult.
  Append $\boldsymbol{\gamma} = \nabla_{\boldsymbol{b}^{(l)}}\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ to list of bias gradients
  Append $(\mathcal{H}^{(l-1)})^{\mathsf{T}} \cdot \boldsymbol{\gamma} = \nabla_{\mathcal{W}^{(l)}}\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$ to list of weight gradients
  $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} \cdot (\mathcal{W}^{(l)})^{\mathsf{T}} = \nabla_{\mathcal{H}^{(l-1)}}\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$

Algo. 9.1: Error backpropagation

Error backpropagation example (1)

Assume $\boldsymbol{x}_0 = [2, 5, 7], y_0 = 2.5$, and a two-layered ANN with the MSE cost, and sigmoid activation functions $\sigma(z) = \frac{1}{1+e^{-z}}$. The hidden layer contains two neurons with output $\boldsymbol{h}^{(1)} \in \mathbb{R}^2$, while the weight vectors are initialized with $\boldsymbol{\mathcal{W}}^{(1)} = \left[\begin{smallmatrix} 0.1 & -0.3 & 0.2 \\ 0.0 & 0.4 & -0.9 \end{smallmatrix}\right]^\mathsf{T}, \boldsymbol{b}^{(1)} = [0.05, -0.03]$, and $\boldsymbol{\mathcal{W}}^{(2)} = [0.2, -0.8]^\mathsf{T}, \boldsymbol{b}^{(2)} = [0.1]$. Applying SGD, we start with forward propagation:

$$\begin{aligned} \boldsymbol{h}^{(1)} &= \varphi^{(1)}(\boldsymbol{x}_0 \boldsymbol{\mathcal{W}}^{(1)} + \boldsymbol{b}^{(1)}) \\ &= \sigma([0.1, -4.3] + [0.05, -0.03]) = [0.53, 0.01] \\ \hat{y}_0 &= \boldsymbol{h}^{(1)} \boldsymbol{\mathcal{W}}^{(2)} + \boldsymbol{b}^{(2)} = 0.198 \end{aligned}$$

Error backpropagation example (2)

Backpropagation (with $\sigma'(z) = \partial_z \sigma(z) = \sigma(z)(1 - \sigma(z))$):

$$
\begin{aligned}
\boldsymbol{\gamma}^{(2)} = \nabla_{\hat{y}_0} \mathcal{L}(y_0, \hat{y}_0) &= \nabla_{\hat{\boldsymbol{y}}} (y_0 - \hat{y}_0)^2 = -2(y_0 - \hat{y}_0) = -4.604 \\
\nabla_{\boldsymbol{b}^{(2)}} \mathcal{L}(y_0, \hat{y}_0) &= \boldsymbol{\gamma}^{(2)} \odot \partial(\varphi^{(2)})(\boldsymbol{z}^{(2)}) = \boldsymbol{\gamma}^{(2)} \\
\nabla_{\boldsymbol{\mathcal{W}}^{(2)}} \mathcal{L}(y_0, \hat{y}_0) &= (\boldsymbol{h}^{(1)})^\mathsf{T} \cdot \boldsymbol{\gamma}^{(2)} = [-2.44, -0.046]^\mathsf{T} \\
\boldsymbol{\gamma}^{(1)} = \nabla_{\boldsymbol{h}^{(1)}} \mathcal{L}(y_0, \hat{y}_0) &= \boldsymbol{\gamma}^{(2)} \cdot (\boldsymbol{\mathcal{W}}^{(2)})^\mathsf{T} = [-0.921, 3.683] \\
\nabla_{\boldsymbol{b}^{(1)}} \mathcal{L}(y_0, \hat{y}_0) &= \boldsymbol{\gamma}^{(1)} \odot \partial(\varphi^{(1)})(\boldsymbol{z}^{(1)}) = \boldsymbol{\gamma}^{(1)} \odot \sigma'(\boldsymbol{x}_0 \boldsymbol{\mathcal{W}}^{(1)} + \boldsymbol{b}^{(1)}) \\
&= \boldsymbol{\gamma}^{(1)} \odot (\boldsymbol{h}^{(1)}(1 - \boldsymbol{h}^{(1)})) = [-0.229, 0.036] \\
\nabla_{\boldsymbol{\mathcal{W}}^{(1)}} \mathcal{L}(y_0, \hat{y}_0) &= \boldsymbol{x}_0^\mathsf{T} \cdot \boldsymbol{\gamma}^{(1)} = \begin{bmatrix} -1.84 & -4.605 & -6.447 \\ 7.366 & 18.415 & 25.781 \end{bmatrix}^\mathsf{T}
\end{aligned}
$$

Now update weights and biases according to (9.7) and (9.8).

Weight initialization

Early in deep learning research, it was found that random uniform or random normal weight initialization leads to poor training.
According to Glorot and Bengio[1], use the following layer-specific initialization schemes (with H_{in} and H_{out} denoting amount of hidden units of previous and current layer, respectively):

- uniform: $\boldsymbol{w} \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{H_{\text{in}}+H_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{H_{\text{in}}+H_{\text{out}}}}\right)$
- normal: $\boldsymbol{w} \sim \mathcal{N}\left(0, \frac{\sqrt{2}}{\sqrt{H_{\text{in}}+H_{\text{out}}}}\right)$

Please note that generally due to the random weight initialization the result of repeated error backpropagation training is always different regardless of having the same hyperparameters and the same data.
This equals to local optimization in highly non-linear parameter spaces at random starting points.

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", *Proceedings of Machine Learning Research*, 2010

Regularizing neural networks

In order to mitigate overfitting, ANNs must be regularized by

- weight decay, i.e., adding an ℓ_2 penalty term to the weights, see (9.6),
- layer normalization during training,
  - i.e all layers' activations are normalized by standard scaling separately,
- dropout, i.e., randomly disable nodes' contribution.
  - This helps especially in deep networks,
  - and effectively builds an ensemble of ANNs with shared edges.

Advanced topologies

Fig. 9.13: Recurrent (left) and 1-D convolutional (right) ANNs are more appropriate in time domains, e.g., where the given data set has a dynamic system background

Hyperparameter optimization (1)

Fig. 9.14: The three levels of optimization

Hyperparameter optimization (2)

- Hyperparameter optimization is, again, a non-linear optimization problem.
- Evaluation of any point in this space can be very costly, though.
- Information gathered during a search must be fully utilized.
- Toolboxes (incomprehensive)
  - Optuna
  - Scikit-optimize
  - Pyswarm

SL toolboxes

- Deep learning
  - Tensorflow 2 (Keras)
  - PyTorch
  - Chainer
  - CNTK
- Gradient boosting machines
  - XGBoost
  - LightGBM
  - CatBoost
- Linear, tree-based, memory-based models, SVMs, among others
  - Scikit-learn

Summary: what you've learned today

- Industry has high demand for ML applications.
- Higher bias trades off variance for a better overall score.
- How to cross-validate and improve SL models.
- How features are engineered and normalized.
- Fundamentals of linear regression and neural networks.

Table of contents

10 Approximative on-policy prediction
- Impact of function approximation to the RL task
- Gradient-based prediction
- Batch learning

Preface

Until further notice we assume that:

- The state space is consisting of at least one continuous quantity or an unfeasible large amount of discrete states (quasi-continuous).
  - The state is considered a vector: $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \cdots \end{bmatrix}^\mathsf{T}$.
- The action space remains discrete and feasible small.
  - The action can be represented as a scalar: $\boldsymbol{u} = u$ (cf. 2nd lecture).
- The applied approximation functions $J(\boldsymbol{w})$ are differentiable with the parameter vector $\boldsymbol{w}$.
  - Therefore, the gradient $\nabla J(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{w})}{\partial w_1} & \frac{\partial J(\boldsymbol{w})}{\partial w_2} & \cdots \end{bmatrix}^\mathsf{T}$ exists.

Focus of this and the next lecture:

- Transferring previous RL methods from discrete to continuous state-space problems in the on-policy case.
- Applying off-policy approaches with function approximation is not straightforward and will be largely skipped.
  - For further insights we refer to chapter 11 in *R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018*.

Non-stationarity

- Standard assumption of supervised ML: static and i.i.d. data processes
- Deviating impacts in the RL framework:
  - Changing environments (e.g., by tear and wear)
  - Dynamic learning in control tasks, i.e., changing policy (next lecture)

Fig. 10.1: GPI changes the underlying stochastic processes generating data inputs to be learned by function approximators (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Prediction framework with function approximation (1)

- Estimate true value function $v_\pi(\boldsymbol{x})$ using a parametrizable approximate value function

$$\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w}) \approx v_\pi(\boldsymbol{x}). \tag{10.1}$$

- The state $\boldsymbol{x}$ might be enhanced by feature engineering (i.e., additional signal inputs are derived in the feature vector $\tilde{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) \in \mathbb{R}^\kappa$).
- Above, $\boldsymbol{w} \in \mathbb{R}^\zeta$ is the parameter vector.
- Typically, $\zeta << |\mathcal{X}|$ applies (otherwise approximation is pointless).

Generalization

Due to the usage of function approximation one incremental learning step changes at least one element $w_i \in \boldsymbol{w}$ which

- affects the estimated value of many states compared to
- the tabular case where one update step affects only one state.

Prediction framework with function approximation (2)

- In the tabular case a specific prediction objective was not needed:
  - The learned value function could exactly match the true value.
  - The value estimate at each state was decoupled from other states.
- Due to generalization impact we need to define an accuracy metric on the entire state space (the RL prediction goal):

Definition 10.1: Mean Squared Value Error

The RL prediction objective is defined as the mean squared value error

$$\overline{\mathrm{VE}}(\boldsymbol{w}) = \int_{\mathcal{X}} \mu(\boldsymbol{x}) \left[v_\pi(\boldsymbol{x}) - \hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})\right]^2 \tag{10.2}$$

with $\mu(\boldsymbol{x}) \in \{\mathbb{R} | \mu(\boldsymbol{x}) \geq 0\}$ being a state distribution weight with $\int_{\mathcal{X}} \mu = 1$.

- Practical note: As the true value $v_\pi(\boldsymbol{x})$ is most likely unknown in most tasks, (10.2) cannot be computed exactly but only estimated.

Simplification for on-policy prediction

- For prediction we focus entirely on the on-policy case.
- Hence, $\mu(\boldsymbol{x})$ is the on-policy distribution under π.
- For practical usage we can therefore approximate the weighted integration over the entire state space $\mathcal{X}$ in (10.2) by the sampled MSE of the visited state trajectory:

$$\overline{\mathsf{VE}}(\boldsymbol{w}) \approx J(\boldsymbol{w}) = \sum_k \left[v_\pi(\boldsymbol{x}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})\right]^2 . \tag{10.3}$$

- If we would perform off-policy prediction we have to transform the sampled value (estimates) from the behavior to the target policy.
- Likewise when doing this for tabular methods, this increases the prediction variance.
- In combination with generalization errors due to function approximation, the overall risk of diverging is significantly higher compared to the on-policy case.

Prediction challenges with function approximation

Summarizing the two previous slides:

- The goal is to find

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w}). \tag{10.4}$$

First challenge:

- Function approximator $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ requires certain form to fit $v_\pi(\boldsymbol{x})$.

Second challenge:

- If $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is linear: convex optimization problem.
  - The nice case: the local optimum equals the global optimum and is uniquely discoverable. But requires linear feature dependence.
- If $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is non-linear: non-linear optimization problem.
  - The ugly case: possible multitude of local optima with no guarantee to locate the global one.
  - Depending on optimization strategy the RL algorithm may diverge.

Table of contents

10 Approximative on-policy prediction
- Impact of function approximation to the RL task
- Gradient-based prediction
- Batch learning

Updating the parameter vector to find (local) optimum

Transferring the idea of incremental learning steps from the tabular case

$$\hat{v}(x) \leftarrow \hat{v}(x) + \alpha \left[v_\pi(x) - \hat{v}(x) \right] \tag{10.5}$$

to function approximation using a gradient descent update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w}). \tag{10.6}$$

- The search direction is the prediction objective gradient $\nabla_{\boldsymbol{w}} J(\boldsymbol{w})$.
- The learning rate α determines the step size of one update.

How to retrieve the gradient?

Fig. 10.2: Exemplary optimization paths for (stochastic) gradient descent (derivative work of www.wikipedia.org, CC0 1.0)

- Full calculus of $\nabla_{\boldsymbol{w}} J(\boldsymbol{w})$:
  - Batch evaluation on sampled sequence $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$ might be computationally costly.
  - In RL control: since π changes over time, past data in batch is not fully representative.
- SGD: sample gradient at a given state $\boldsymbol{x}_k$ and parameter vector $\boldsymbol{w}_k$:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \approx -\left[v_\pi(\boldsymbol{x}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k).$$

- Regular gradient descent leads to same result as SGD in expectation (averaging of samples).

Asking an expert on convergence properties

The optimization task (10.4) could be

- non-linear,
- multidimensional and
- non-stationary.

Applying gradient descent to such a problem requires:

- Enormous luck to initialize $\boldsymbol{w}_0$ close to the global optimum.
- Cautious tuning of α to prevent diverging or chattering of $\boldsymbol{w}_k$.

SGD-based learning step

Despite the possible problems we apply SGD-based learning due to its striking simplicity (and wide distribution in the literature):

Gradient-based parameter update

To optimize (10.4) by an appropriate function approximator $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ the incremental learning update per step is

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha \left[v_\pi(\boldsymbol{x}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k) \right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k). \tag{10.7}$$

Nevertheless, the true update target $v_\pi(\boldsymbol{x}_k)$ is often unknown due to

- noise or
- the learning process itself (e.g., bootstrapping estimates).

Generalization example for parameter update

- Function approximation $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w}) = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 & x_2 & 1 \end{bmatrix}^\mathsf{T}$
- Initial parameter: $\boldsymbol{w}_0^\mathsf{T} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$, $v_\pi(\boldsymbol{x}_0 = \begin{bmatrix} 1 & 1 \end{bmatrix}^\mathsf{T}) = 1$, $\alpha = 0.1$
- New parameter set:

$$\begin{aligned} \boldsymbol{w}_1^\mathsf{T} &= \boldsymbol{w}_0^\mathsf{T} + \alpha \left[v_\pi(\boldsymbol{x}_0) - \hat{v}(\tilde{\boldsymbol{x}}_0, \boldsymbol{w}_0) \right] \left(\nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_0, \boldsymbol{w}_0) \right)^\mathsf{T} \\ &= \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} + 0.1 \left(1 - 3\right) \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.8 & 0.8 & 0.8 \end{bmatrix} \end{aligned}$$

Algorithmic implementation: gradient Monte Carlo

- Direct transfer from tabular case to function approximation
- Update target becomes the sampled return $v_\pi(\boldsymbol{x}_k) \approx g_k$

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$
input: a differentiable function $\hat{v} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
    generate an episode following π: $x_0, u_0, r_1, \ldots, x_T$;
    calculate every-visit return g_k;
    for $k = 0, 1, \ldots, T-1$ *time steps* **do**
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g_k - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})$;

Algo. 10.1: Every-visit gradient MC (output: parameter vector $\boldsymbol{w}$ for $\hat{v}_\pi$)

Semi-gradient methods

- If bootstrapping is applied, the true target $v_\pi(\boldsymbol{x}_k)$ is approximated by a target depending on the estimate $\hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})$.
- If $\hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})$ does not fit $v_\pi(\boldsymbol{x}_k)$, the update target becomes a biased estimate of $v_\pi(\boldsymbol{x}_k)$.
  - For example, in the TD(0) case applying SGD we receive:

$$
\begin{aligned}
v_\pi(\boldsymbol{x}) &\approx r + \gamma \hat{v}(\tilde{\boldsymbol{x}}', \boldsymbol{w}), \\
J(\boldsymbol{w}) &\approx \sum_k \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k)\right]^2, \\
\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &\approx \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k)\right] \\
&\quad \nabla_{\boldsymbol{w}} \left[\gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k)\right].
\end{aligned}
\tag{10.8}
$$

Semi-gradient methods

When bootstrapping is applied, the gradient does not take into account any gradient component of the bootstrapped target estimate.

- Motivation: speed up gradient calculation while assuming that the simplification error is small (e.g., due to discounting).

Algorithmic implementation: semi-gradient TD(0)

The semi-gradient of $J(\boldsymbol{w})$ for TD(0) from prev. slide is then

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \approx -\left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k). \tag{10.9}$$

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$
input: a differentiable function $\hat{v} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$ with $\hat{v}(\tilde{\boldsymbol{x}}_T, \cdot) = 0$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action from $\pi(\boldsymbol{x}_k)$;
        observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} + \gamma \hat{v}(\tilde{\boldsymbol{x}}_{k+1}, \boldsymbol{w}) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})$;
        exit loop if $\boldsymbol{x}_{k+1}$ is terminal;

Algo. 10.2: Semi-gradient TD(0) (output: parameter vector $\boldsymbol{w}$ for $\hat{v}_{\pi}$)

input: a policy π to be evaluated, a feature representation $\tilde{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$
input: a differentiable function $\hat{v} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$ with $\hat{v}(\tilde{\boldsymbol{x}}_T, \cdot) = 0$
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, prediction steps $n \in \mathbb{Z}^+$
init: value-function weights $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2 \ldots$ *episodes* **do**
    initialize and store $\boldsymbol{x}_0$;
    $T \leftarrow \infty$;
    repeat $k = 0, 1, 2, \ldots$
        if $k < T$ **then**
            take action from $\pi(\boldsymbol{x}_k)$, observe and store $\boldsymbol{x}_{k+1}$ and r_{k+1};
            if $\boldsymbol{x}_{k+1}$ 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 $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{v}(\tilde{\boldsymbol{x}}_{\tau+n}, \boldsymbol{w})$;
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g - \hat{v}(\tilde{\boldsymbol{x}}_{\tau}, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_{\tau}, \boldsymbol{w})$;
    until $\tau = T - 1$;

Algo. 10.3: n-step semi-gradient TD (output: parameter vector $\boldsymbol{w}$ for $\hat{v}_{\pi}$)

Table of contents

10 Approximative on-policy prediction
- Impact of function approximation to the RL task
- Gradient-based prediction
- Batch learning

Background and motivation

- As already discussed in the tabular case: incremental learning is not data efficient (cf. example Fig. 5.8).
  - During one incremental learning step we are not utilizing the given information to the maximum possible extent.
  - Also applies to SGD-based updates with function approximation.

- Alternative: batch learning methods
  - Find $\boldsymbol{w}^*$ given a fixed, consistent data set $\boldsymbol{\mathcal{D}} = \{\langle \boldsymbol{x}_0, v_\pi(\boldsymbol{x}_0)\rangle, \langle \boldsymbol{x}_1, v_\pi(\boldsymbol{x}_1)\rangle, \ldots\}$.

- What batch learning options do we have?
  - Experience replay (cf. planning and learning lecture, e.g., Fig. 7.6)
  - If $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is linear: closed-form least-squares solution

SGD with experience replay

Based on the data set

$$\boldsymbol{\mathcal{D}} = \{\langle \boldsymbol{x}_0, v_\pi(\boldsymbol{x}_0)\rangle, \langle \boldsymbol{x}_1, v_\pi(\boldsymbol{x}_1)\rangle, \ldots\}$$

repeat:

1. Sample uniformly $i = 1, \ldots, b$ state-value pairs from experience (so-called mini batch)

$$\langle \boldsymbol{x}_i, v_\pi(\boldsymbol{x}_i)\rangle \sim \boldsymbol{\mathcal{D}}.$$

2. Apply (semi) SGD update step:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \frac{\alpha}{b} \sum_{i=1}^{b} \left[v_\pi(\boldsymbol{x}_i) - \hat{v}(\tilde{\boldsymbol{x}}_i, \boldsymbol{w}_i)\right] \nabla_{\boldsymbol{w}} \hat{v}(\tilde{\boldsymbol{x}}_i, \boldsymbol{w}_i).$$

- Universally applicable: $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ can be any differentiable function.
- The usual technical tuning requirements regarding α apply.
- True target $v_\pi(\boldsymbol{x})$ is usually approximated by MC or TD targets.

(Ordinary) least squares

Assuming the following applies:

- $\hat{v}(\tilde{\boldsymbol{x}}, \boldsymbol{w})$ is a linear estimator and
- $\boldsymbol{\mathcal{D}}$ a fixed, representative data set following the on-policy distribution.

Then, minimizing the quadratic cost function (10.3) becomes

- an ordinary least squares (OLS) / linear regression problem.

We focus on the combination of OLS and TD(0) (so-called LSTD), but the following can be equally extended to n-step learning or MC.

- Rewriting $J(\boldsymbol{w})$ from (10.3) using linear approximation TD(0) target:

$$v_\pi(\boldsymbol{x}_k) \approx r_{k+1} + \gamma \hat{v}(\boldsymbol{x}_{k+1}) = r_{k+1} + \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \boldsymbol{w} \tag{10.10}$$

$$J(\boldsymbol{w}) = \sum_k \left[v_\pi(\boldsymbol{x}_k) - \hat{v}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w})\right]^2 = \sum_k \left[r_{k+1} - \left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T}\right) \boldsymbol{w}\right]^2 .$$

Ordinary LSTD

The quadratic cost function

$$J(\boldsymbol{w}) = \sum_k \left[r_{k+1} - \left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \right) \boldsymbol{w} \right]^2$$

obtains the least squares

- target / dependent variable r_{k+1} and
- regressor / independent variable $\left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \right)$.

With b samples we can form a target vector $\boldsymbol{y}$ and regressor matrix $\boldsymbol{\Xi}$:

$$\boldsymbol{y} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_b \end{bmatrix}, \quad \boldsymbol{\Xi} = \begin{bmatrix} \left(\tilde{\boldsymbol{x}}_0^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_1^\mathsf{T} \right) \\ \left(\tilde{\boldsymbol{x}}_1^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_2^\mathsf{T} \right) \\ \vdots \\ \left(\tilde{\boldsymbol{x}}_{b-1}^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_b^\mathsf{T} \right) \end{bmatrix}. \tag{10.11}$$

Ordinary LSTD and regularization

Applying the linear regression solution (9.4) from previous lecture:

LSTD solution

Having arranged $i = 1, \ldots, b$ samples $\langle \boldsymbol{x}_i, v_\pi(\boldsymbol{x}_i) \rangle \sim \boldsymbol{\mathcal{D}}$ using TD(0) and linear function approximation as in (10.11), the LSTD solution is

$$\boldsymbol{w}^* = (\boldsymbol{\Xi}^\mathsf{T} \boldsymbol{\Xi})^{-1} \boldsymbol{\Xi}^\mathsf{T} \boldsymbol{y}. \tag{10.12}$$

- The parameter $\boldsymbol{w}^*$ is also called the TD fixed point.
- The state-value prediction is simply $\hat{v}(\tilde{\boldsymbol{x}}_k) = \tilde{\boldsymbol{x}}_k^\mathsf{T} \boldsymbol{w}^*$.

Depending on the policy π the rows in $\boldsymbol{\Xi}$ might be linearly correlated.

- Bad matrix condition of $\boldsymbol{\Xi}^\mathsf{T} \boldsymbol{\Xi}$ can lead to unfeasible values in $\boldsymbol{w}^*$.
- Counter measure: Add Tikhonov regularization (Ridge regression with penalty term ϵ, cf. (9.6)):

$$\boldsymbol{w}^*_{\text{Ridge}} = (\boldsymbol{\Xi}^\mathsf{T} \boldsymbol{\Xi} + \epsilon \boldsymbol{I})^{-1} \boldsymbol{\Xi}^\mathsf{T} \boldsymbol{y}. \tag{10.13}$$

Recursive least squares

- OLS computational complexity is in the range of $\mathcal{O}(\kappa^{2.3}) \dots \mathcal{O}(\kappa^3)$.
  - κ being the number of features.
- Computational costly if new data points $\langle \boldsymbol{x}_i, v_\pi(\boldsymbol{x}_i) \rangle$ are added to $\boldsymbol{\mathcal{D}}$.
- Consider supplement / extension: recursive least square (RLS).
  - Each RLS update complexity is $\mathcal{O}(\kappa^2)$.
- In the following, we briefly represent the recipe-style RLS equations.
  - Detailed derivation can be found e.g. R. Isermann and M. Münchhof, *Identification of Dynamic Systems*, Springer-Verlag Berlin Heidelberg, 2011 (also as electronic copy on Panda).

RLS-TD

After every step we receive

- a new regressor vector $\boldsymbol{\xi}_{k+1}^{\mathsf{T}} = \left(\tilde{\boldsymbol{x}}_k^{\mathsf{T}} - \gamma \tilde{\boldsymbol{x}}_{k+1}^{\mathsf{T}}\right)$ and
- a new update target $y_{k+1} = r_{k+1}$.

The RLS update rule is then

$$\begin{aligned} \boldsymbol{c}_k &= \frac{\boldsymbol{P}_k \boldsymbol{\xi}_{k+1}}{\lambda_{k+1} + \boldsymbol{\xi}_{k+1}^{\mathsf{T}} \boldsymbol{P}_k \boldsymbol{\xi}_{k+1}}, \\ \boldsymbol{w}_{k+1} &= \boldsymbol{w}_k + \boldsymbol{c}_k \left(y_{k+1} - \boldsymbol{\xi}_{k+1}^{\mathsf{T}} \boldsymbol{w}_k\right), \\ \boldsymbol{P}_{k+1} &= \left(\boldsymbol{I} - \boldsymbol{c}_k \boldsymbol{\xi}_{k+1}^{\mathsf{T}}\right) \frac{\boldsymbol{P}_k}{\lambda_{k+1}}, \end{aligned} \tag{10.14}$$

with

- $\lambda_k \in \{\mathbb{R} | 0 < \lambda \leq 1\}$ is an optional forgetting factor,
- $\boldsymbol{P}_k$ is the covariance matrix and
- $\boldsymbol{c}_k$ is an adaptive correction to reduce the error $\left(y_{k+1} - \boldsymbol{\xi}_k^{\mathsf{T}} \boldsymbol{w}_{k+1}\right)$.

Algorithmic implementation: RLS-TD

input: a policy π to be evaluated
input: a feature representation $\tilde{\boldsymbol{x}}$ with $\tilde{\boldsymbol{x}}_T = 0$ (i.e., $\hat{v}(\tilde{\boldsymbol{x}}_T, \cdot) = 0$)
parameter: forgetting factor $\lambda \in \{\mathbb{R} | 0 < \lambda \leq 1\}$
init: weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily, covariance $\boldsymbol{P} > 0$ (e.g. $\boldsymbol{P} = \beta \boldsymbol{I}$)
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action from $\pi(\boldsymbol{x}_k)$, observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        $y \leftarrow r_{k+1}$;
        $\boldsymbol{\xi}^\mathsf{T} \leftarrow \tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T}$;
        $\boldsymbol{c} \leftarrow (\boldsymbol{P}\boldsymbol{\xi}) \,/\, \big(\lambda + \boldsymbol{\xi}^\mathsf{T} \boldsymbol{P} \boldsymbol{\xi}\big)$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{c}\big(y - \boldsymbol{\xi}^\mathsf{T} \boldsymbol{w}\big)$;
        $\boldsymbol{P} \leftarrow \big(\boldsymbol{I} - \boldsymbol{c}\boldsymbol{\xi}^\mathsf{T}\big)\, \boldsymbol{P}/\lambda$;
        exit loop if $\boldsymbol{x}_{k+1}$ is terminal;

Algo. 10.4: RLS-TD (output: parameter vector $\boldsymbol{w}$ for $\hat{v}_\pi$)

Some remarks on RLS usage in RL prediction

- Covariance matrix $\boldsymbol{P}$ can be inspected for certainty analysis.
  - Small-valued elements in $\boldsymbol{P}$ suggest an accurate estimate.
- For $\lambda = 1$ the RLS converges to a static solution.
  - Never forgets something (i.e., problematic for non-stationary problem).
  - Given the same data set $\boldsymbol{\mathcal{D}}$ the RLS converges to OLS solution.
- However, if RLS-TD should be used online $\lambda \in [0.95, 0.99]$ is typical.
  - Application-dependent λ_k might be adapted online after each step.
  - As seen in (10.14), $\lambda < 1$ increases the covariance which potentially could lead to numerical instabilities depending on the given data set.
  - In this case, regularization is required.[1]
- General RLS approach (10.14) is also applicable to MC or n-step TD.
  - Derivation follows presented scheme based on the altered update rules.

[1] Recommended reading: S. Gunnarson, *Combining Tracking and Regularization in Recursive Least Squares Identification*, Proceedings of 35th IEEE Conference on Decision and Control, 1996

Summary: what you've learned today

- To cover unfeasible large or continuous state spaces function approximation is required.
  - Feature engineering supports the learning process.
- On-policy prediction seems rather straightforward with function approximation:
  - Just transfer the incremental learning from tabular case to gradient descent on parameter vector $\boldsymbol{w}$.
  - Stochastic gradient descent allows step-by-step based updates.
- Gradient-based prediction is not risk free (especially non-linear case):
  - no convergence guarantees,
  - local optima vs. global optimum.
- If bootstrapping is applied, the update target depends on $\boldsymbol{w}$.
  - True gradient becomes computationally more complex.
  - Semi-gradient methods reduce computational burden at accuracy costs.
- Batch learning squeezes out all available prediction information from a given data set.
  - If linear function approximation is applied, closed-form solutions exist.

Table of contents

11 Approximative value-based control
- On-policy control with (semi-)gradients
- Least squares policy iteration (LSPI)
- Deep q-networks (DQN)

Preface

Problem space: it is further assumed that

- the states $\boldsymbol{x}$ are (quasi-)continuous and
- the actions u are discrete.

Today's focus:

- valued-based control tasks, i.e., transferring the established tabular methods to work with function approximation.
- Hence, we need to extend the previous prediction methods to action values

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \approx q_\pi(\boldsymbol{x}, u). \tag{11.1}$$

- And apply the well-known generalized policy iteration scheme (GPI) to find optimal actions:

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \approx q^*(\boldsymbol{x}, u). \tag{11.2}$$

Types of action-value function approximation

Fig. 11.1: Possible function approximation settings for discrete actions

- Left: one function with both states and actions as input
- Middle: one function with $i = 1, 2, \ldots$ outputs covering the action space (e.g., ANN with appropriate output layer)
- Right: multiple (sub-)functions one for each possible action u_i (e.g., multitude of linear approximators in small action spaces)

Feature engineering

- Also for action-value estimation a proper feature engineering (FE) is of vital importance.
- Compared to the state-value prediction, the action becomes part of the FE processing:

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) = \hat{q}\left(\boldsymbol{f}\left(\boldsymbol{x}, u\right), \boldsymbol{w}\right). \tag{11.3}$$

- Above, $\boldsymbol{f}(\boldsymbol{x}, u) \in \mathbb{R}^{\kappa}$ is the FE function.
- For sake of notation simplicity we write $\hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ and understand that FE has already been considered (i.e., is a part of $\hat{q}$).

Gradient-based action-value learning

- Transferring the objective (10.3) from on-policy prediction to control yields:

$$J(\boldsymbol{w}) = \sum_k \left[q_\pi(\boldsymbol{x}_k, u_k) - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right]^2 . \tag{11.4}$$

- Analogous, the (semi-)gradient-based parameter update from (10.7) is also applied to action values:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha \left[q_\pi(\boldsymbol{x}_k, u_k) - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}_k)\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}_k). \tag{11.5}$$

- Depending on the control approach, the true target $q_\pi(\boldsymbol{x}_k, u_k)$ is approximated by:
  - Monte Carlo: full episodic return $q_\pi(\boldsymbol{x}_k, u_k) \approx g$,
  - SARSA: one-step bootstrapped estimate $q_\pi(\boldsymbol{x}_k, u_k) \approx r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, u_{k+1}, \boldsymbol{w}_k)$,
  - n-step SARSA: $q_\pi(\boldsymbol{x}_k, u_k) \approx r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{q}(\boldsymbol{x}_{k+n}, u_{k+n}, \boldsymbol{w}_{k+n-1})$.

Houston: we have a problem

- Recall tabular policy improvement theorem (Theo. 3.1): guarantee to find a globally better or equally good policy in each update step.
- With parameter updates (11.5) generalization applies.
- Hence, when reacting to one specific state-action transition other parts of the state-action space within $\hat{q}$ are affected too.

Fig. 11.2: GPI

Loss of policy improvement theorem

- Is not applicable with function approximation!
- We may improve and impair the policy at the same time!

Algorithmic implementation: gradient MC control

- Direct transfer from tabular case to function approximation
- Update target becomes the sampled return $q_\pi(\boldsymbol{x}_k, u_k) \approx g_k$
- If operating ε-greedy on $\hat{q}$: baseline policy (given by $\boldsymbol{w}_0$) must (successfully) terminate the episode!

input: a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$
input: a policy π (only if estimating q_π)
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^\zeta$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate episode following π or ε-greedy on $\hat{q}$: $x_0, u_0, r_1, \ldots, x_T$;
  calculate every-visit return g_k;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g_k - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})$;

Algo. 11.1: Every-visit gradient MC-based action-value estimation (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_\pi$ or $\hat{q}^*$)

Algorithmic implementation: semi-gradient SARSA

input: a differentiable function $\hat{q} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\varsigma} \to \mathbb{R}$
input: a policy π (only if estimating q_{π})
parameter: step size $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^{\varsigma}$ arbitrarily
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action from $\pi(\boldsymbol{x}_k)$ or ε-greedy on $\hat{q}(\boldsymbol{x}_k, \cdot, \boldsymbol{w})$;
        observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        if $\boldsymbol{x}_{k+1}$ *is terminal* **then**
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})$;
            go to next episode;
        choose u' from $\pi(\boldsymbol{x}_{k+1})$ or ε-greedy on $\hat{q}(\boldsymbol{x}_{k+1}, \cdot, \boldsymbol{w})$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, u', \boldsymbol{w}) - \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w})$;

Algo. 11.2: Semi-gradient SARSA action-value estimation (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_{\pi}$ or $\hat{q}^*$)

SARSA application example: mountain car (1)

Fig. 11.3: Classic RL control example: mountain car (derivative work based on https://github.com/openai/gym, MIT license)

- Two cont. states: position, velocity
- One discrete action: acceleration given by $\{\text{left}, \text{none}, \text{right}\}$
- $r_k = -1$, i.e., goal is to terminate episode as quick as possible
- Episode terminates when car reaches the flag (or max steps)
- Simplified longitudinal car physics with state constraints
- Position initialized randomly within valley, zero initial velocity
- Car is underpowered and requires swing-up

SARSA application example: mountain car (2)

Fig. 11.4: Cost-to-go function $-\max_u \hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ for mountain car task using linear approximation with SARSA and tile coding (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Tile coding

- Problem space is grouped into (overlapping) partitions / tiles.
- Performs a discretization of the problem space.
- Function approximation serves as interpolation between tiles.
- Find an example here: https://github.com/MeepMoop/tilecoding .

Fig. 11.5: Tile coding example in 2D (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

input: a differentiable function $\hat{q} : \mathbb{R}^{\kappa} \times \mathbb{R}^{\zeta} \to \mathbb{R}$
input: a policy π (only if estimating q_π)
parameter: $\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, $n \in \mathbb{Z}^+$
init: parameter vector $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
for $j = 1, 2 \ldots$ *episodes* **do**
    initialize and store $\boldsymbol{x}_0$;
    select and store $u_0 \sim \pi(\boldsymbol{x}_0)$ or ε-greedy w.r.t. $\hat{q}(\boldsymbol{x}_0, \cdot, \boldsymbol{w})$;
    $T \leftarrow \infty$;
    repeat $k = 0, 1, 2, \ldots$
        if $k < T$ **then**
            take action u_k observe and store $\boldsymbol{x}_{k+1}$ and r_{k+1};
            if $\boldsymbol{x}_{k+1}$ *is terminal* **then** $T \leftarrow k + 1$;
            else select & store $u_{k+1} \sim \pi(\boldsymbol{x}_{k+1})$ or ε-greedy w.r.t. $\hat{q}(\boldsymbol{x}_{k+1}, \cdot, \boldsymbol{w})$;
        $\tau \leftarrow k - n + 1$ (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 $\tau + n < T$: $g \leftarrow g + \gamma^n \hat{q}(\boldsymbol{x}_{\tau+n}, u_{\tau+n}, \boldsymbol{w})$;
            $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[g - \hat{q}(\boldsymbol{x}_\tau, u_\tau, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_\tau, u_\tau, , \boldsymbol{w})$;
    until $\tau = T - 1$;

Algo. 11.3: n-step semi-gradient SARSA (output: parameter vector $\boldsymbol{w}$ for $\hat{q}_\pi$ or $\hat{q}^*$)

Table of contents

11 Approximative value-based control
- On-policy control with (semi-)gradients
- Least squares policy iteration (LSPI)
- Deep q-networks (DQN)

Transferring LSTD-style batch learning to action values

- In the previous lecture we developed a closed-form batch learning tool: LSTD.
  - Linear function approximation.
  - Fixed, representative data set $\mathcal{D}$.
- Same idea can be transferred to action values when bootstrapping with one-step Sarsa, called LS-SARSA (or sometimes LSTDQ):

$$\begin{aligned} q_\pi(\boldsymbol{x}_k, u_k) &\approx r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, u_{k+1}, \boldsymbol{w}_k), \\ \hat{q}(\boldsymbol{x}_k, u_k, \boldsymbol{w}_k) &= \hat{q}(\tilde{\boldsymbol{x}}_k, \boldsymbol{w}_k) = \tilde{\boldsymbol{x}}_k^\mathsf{T} \boldsymbol{w}_k. \end{aligned} \tag{11.6}$$

- The cost function for action-value prediction is then:

$$J(\boldsymbol{w}) = \sum_k \left[r_{k+1} - \left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T} \right) \boldsymbol{w} \right]^2. \tag{11.7}$$

- Hence, the closed-form least squares solution for the action values is the same as for the state value case but the feature vector depends also on the actions:

$$\tilde{\boldsymbol{x}}_k = \boldsymbol{f}(\boldsymbol{x}_k, u_k).$$

On and off-policy LS-SARSA

With b samples we can form a target vector $\boldsymbol{y}$ and regressor matrix $\boldsymbol{\Xi}$:

$$\boldsymbol{y} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_b \end{bmatrix}, \quad \boldsymbol{\Xi} = \begin{bmatrix} \left(\tilde{\boldsymbol{x}}_0^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_1^\mathsf{T}\right) \\ \left(\tilde{\boldsymbol{x}}_1^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_2^\mathsf{T}\right) \\ \vdots \\ \left(\tilde{\boldsymbol{x}}_{b-1}^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_b^\mathsf{T}\right) \end{bmatrix}. \tag{11.8}$$

Regarding the data input to $\boldsymbol{\Xi}$ we can distinguish two cases: The actions u_k and u_{k+1} in the feature pair $\left(\tilde{\boldsymbol{x}}_k^\mathsf{T} - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T}\right)$ per row in $\boldsymbol{\Xi}$ either descends from the

- same policy π (on-policy learning) or
- the action u_{k+1} in $\tilde{\boldsymbol{x}}_{k+1} = \boldsymbol{f}(\boldsymbol{x}_{k+1}, u_{k+1})$ is chosen based on an arbitrary policy π' (off-policy learning).

If we apply off-policy LS-SARSA then

- we retrieve the flexibility to collect training samples arbitrarily
- at the cost of an estimation bias based on the sampling distribution.

LS-SARSA

LS-SARSA solution

Having arranged $i = 1, \dots, b$ samples $\langle \boldsymbol{x}_i, u_i, r_{i+1}, \boldsymbol{x}_{i+1}, u_{i+1} \rangle \sim \boldsymbol{\mathcal{D}}$ using one-step bootstrapping (11.6) and linear function approximation as in (11.8), the LS-SARSA solution is

$$\boldsymbol{w}^* = (\boldsymbol{\Xi}^\mathsf{T} \boldsymbol{\Xi})^{-1} \boldsymbol{\Xi}^\mathsf{T} \boldsymbol{y}. \tag{11.9}$$

Again, basic usage distinction:

- If $\{u_i, u_{i+1}\} \sim \pi$: on-policy prediction (as in LSTD)
- If $u_i \sim \pi$ and $u_{i+1} \sim \pi'$: off-policy prediction (useful for control)

Possible modifications:

- To prevent numeric instability regularization is possible, cf. (10.13)
- Recursive implementation for online usage straightforward, cf. (10.14)

Least squares policy iteration (LSPI)

General idea:

- Apply general policy improvement (GPI) based on data set $\mathcal{D}$,
- Policy evaluation by off-policy LS-SARSA,
- Policy improvement by greedy choices on predicted action values.

Some remarks:

- LSPI is an offline and off-policy control approach.
- Exploration is required by feeding suitable sampling distributions in $\mathcal{D}$:
  - Such as ε-greedy choices based on $\hat{q}$.
  - But also complete random samples are conceivable.

Algorithmic implementation: LSPI

input: a feature representation $\tilde{\boldsymbol{x}}$ with $\tilde{\boldsymbol{x}}_T = 0$ (i.e., $\hat{q}(\tilde{\boldsymbol{x}}_T, \cdot) = 0$)
input: a data set $\langle \boldsymbol{x}_i, u_i, r_{i+1}, \boldsymbol{x}_{i+1} \rangle \sim \boldsymbol{\mathcal{D}}$ with $i = 1, \ldots, b$ samples
parameter: an accuracy threshold $\Delta \in \{\mathbb{R} | 0 < \Delta\}$
init: linear approximation function weights $\boldsymbol{w} \in \mathbb{R}^{\zeta}$ arbitrarily
$\pi \leftarrow \arg\max_u \hat{q}(\cdot, u, \boldsymbol{w})$ (greedy choices based on $\hat{q}(\boldsymbol{w})$);
repeat
    $\boldsymbol{w}' \leftarrow \boldsymbol{w}$;
    $\boldsymbol{w} \leftarrow$ LS-SARSA($\boldsymbol{\mathcal{D}}, u_{i+1} \sim \pi$);
    $\pi \leftarrow \arg\max_u \hat{q}(\cdot, u, \boldsymbol{w})$;
until $||\boldsymbol{w}' - \boldsymbol{w}|| < \Delta$;

Algo. 11.4: Least squares policy iteration (output: $\boldsymbol{w}$ for $\hat{q}^*$)

- In a (small) discrete action space the $\arg\max_u$ operation is straightforward.
- After one full LSPI evaluation the data set $\boldsymbol{\mathcal{D}}$ might be altered to include new data obtained based on the updated $\boldsymbol{w}$ vector.
- Source: M. Lagoudakis and R. Parr, *Least-Squares Policy Iteration*, Journal of Machine Learning Research 4, pp. 1107-1149, 2003

LSPI application example: inverted pendulum (1)

Fig. 11.6: Classic RL control example: inverted pendulum (source: www.wikipedia.org, CC0 1.0)

- Two continuous states: angular position θ and velocity $\dot{\theta}$
- One discrete action: acceleration force (i.e., torque at shaft)
- Action noise as disturbance
- Non-linear system dynamics
- State initialization randomly close to upper equilibrium
- $r_k = 0$ if pendulum is above horizontal line
- $r_k = -1$ if below horizontal line and episode terminates
- $\gamma = 0.95$

LSPI application example: inverted pendulum (2)

- Initial training samples for $\mathcal{D}$ following a policy selecting actions at uniform probability
- Additional samples have been manually added during the training
- Radial basis function as feature engineering

Fig. 11.7: Balancing steps before episode termination with a clipping of maximum 3000 steps (source: M. Lagoudakis and R. Parr, *Least-Squares Policy Iteration*, Journal of Machine Learning Research 4, pp. 1107-1149, 2003)

Algorithmic implementation: online LSPI

input: a feature representation $\tilde{\boldsymbol{x}}$ with $\tilde{\boldsymbol{x}}_T = 0$ (i.e., $\hat{q}(\tilde{\boldsymbol{x}}_T, \cdot, \cdot) = 0$)
parameter: forgetting factor $\lambda \in \{\mathbb{R} | 0 < \lambda \leq 1\}$, $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, update factor $k_w \in \{\mathbb{N} | 1 \leq k_w\}$
init: weights $\boldsymbol{w} \in \mathbb{R}^\varsigma$ arbitrarily, policy π being ε-greedy w.r.t. $\hat{q}(\boldsymbol{w})$, covariance $\boldsymbol{P} > 0$
for $j = 1, 2, \ldots$ *episodes* **do**
  initialize $\boldsymbol{x}_0$ and set $u_0 \sim \pi(\boldsymbol{x}_0)$;
  for $k = 0, 1, 2 \ldots$ *time steps* **do**
    apply action u_k, observe $\boldsymbol{x}_{k+1}$ and r_{k+1}, set $u_{k+1} \sim \pi(\boldsymbol{x}_{k+1})$;
    $y \leftarrow r_{k+1}$;
    $\boldsymbol{\xi}^\mathsf{T} \leftarrow \tilde{\boldsymbol{x}}_k^\mathsf{T}(\boldsymbol{x}_k, u_k) - \gamma \tilde{\boldsymbol{x}}_{k+1}^\mathsf{T}(\boldsymbol{x}_{k+1}, u_{k+1})$;
    $\boldsymbol{c} \leftarrow (\boldsymbol{P}\boldsymbol{\xi}) \,/\, \left(\lambda + \boldsymbol{\xi}^\mathsf{T}\boldsymbol{P}\boldsymbol{\xi}\right)$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{c}\left(y - \boldsymbol{\xi}^\mathsf{T}\boldsymbol{w}\right)$;
    $\boldsymbol{P} \leftarrow \left(\boldsymbol{I} - \boldsymbol{c}\boldsymbol{\xi}^\mathsf{T}\right)\boldsymbol{P}/\lambda$;
    if $k \mod k_w = 0$ **then**
      $\pi \leftarrow \varepsilon$-greedy w.r.t. $\hat{q} = \tilde{\boldsymbol{x}}^\mathsf{T}(\boldsymbol{x}, u)\boldsymbol{w}$;
    exit loop if $\boldsymbol{x}_{k+1}$ is terminal;

Algo. 11.5: Online LSPI with RLS-SARSA (output: $\boldsymbol{w}$ for $\hat{q}^*$)

Remarks on online LSPI

- k_w depicts the number of steps between policy improvement cycles.
- Forgetting factor λ and k_w require mutual tuning:
  - After each policy improvement the policy evaluation requires sample updates to accurately predict the altered policy.
  - Numerically instability may occur for $\lambda < 1$ and requires regularization.
- Hence, the algorithm is online-capable but its policy is normally not updated in a step-by-step fashion.
- Alternative online LSPI with OLS-SARSA can be found in L. Buşoniu et al., *Online least-squares policy iteration for reinforcement learning control*, American Control Conference, 2010.

Table of contents

11 Approximative value-based control
- On-policy control with (semi-)gradients
- Least squares policy iteration (LSPI)
- Deep q-networks (DQN)

General background on DQN

- Recall incremental learning step from tabular Q-learning:

$$\hat{q}(x,u) \leftarrow \hat{q}(x,u) + \alpha \left[r + \gamma \max_u \hat{q}(x',u) - \hat{q}(x,u) \right].$$

- Deep Q-networks (DQN) transfer this to an approximate solution:

$$\boldsymbol{w} = \boldsymbol{w} + \alpha \left[r + \gamma \max_u \hat{q}(\boldsymbol{x}',u,\boldsymbol{w}) - \hat{q}(\boldsymbol{x},u,\boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x},u,\boldsymbol{w}). \tag{11.10}$$

However, instead of using above semi-gradient step-by-step updates, DQN is characterized by

- an experience replay buffer for batch learning (cf. prev. lectures),
- a separate set of weights $\boldsymbol{w}^-$ for the bootstrapped Q-target.

Motivation behind:

- Efficiently use available data (experience replay).
- Stabilize learning by trying to make targets and feature inputs more like i.i.d. data from a stationary process (prevent windup of values).

Summary of DQN working principle (1)

- Take actions u based on $\hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ (e.g., ε-greedy).
- Store observed tuples $\langle \boldsymbol{x}, u, r, \boldsymbol{x}' \rangle$ in memory buffer $\boldsymbol{\mathcal{D}}$.
- Sample mini-batches $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$.
- Calculate bootstrapped Q-target with a delayed parameter vector $\boldsymbol{w}^-$ (so-called target network):

$$q_\pi(\boldsymbol{x}, u) \approx r + \gamma \max_u \hat{q}(\boldsymbol{x}', u, \boldsymbol{w}^-).$$

- Optimize MSE loss between above targets and the regular approximation $\hat{q}(\boldsymbol{x}, u, \boldsymbol{w})$ using $\boldsymbol{\mathcal{D}}_b$

$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma \max_u \hat{q}(\boldsymbol{x}', u, \boldsymbol{w}^-) \right) - \hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \right]^2_{\boldsymbol{\mathcal{D}}_b} . \tag{11.11}$$

- Update $\boldsymbol{w}^-$ based on $\boldsymbol{w}$ from time to time.

Summary of DQN working principle (2)

Fig. 11.8: DQN structure from a bird's-eye perspective (derivative work of Fig. 1.1 and wikipedia.org, CC0 1.0)

Algorithmic implementation: DQN

input: a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}$ (including feature eng.)
parameter: $\varepsilon \in \{\mathbb{R} | 0 < \varepsilon << 1\}$, update factor $k_w \in \{\mathbb{N} | 1 \leq k_w\}$
init: weights $\boldsymbol{w} = \boldsymbol{w}^- \in \mathbb{R}^\zeta$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$ with certain capacity
for $j = 1, 2, \ldots$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, 2 \ldots$ *time steps* **do**
        $u_k \leftarrow$ apply action ε-greedy w.r.t $\hat{q}(\boldsymbol{x}_k, \cdot, \boldsymbol{w})$;
        observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        store tuple $\langle \boldsymbol{x}_k, u_k, r_{k+1}, \boldsymbol{x}_{k+1} \rangle$ in $\boldsymbol{\mathcal{D}}$;
        sample mini-batch $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$ (after initial memory warmup);
        for $i = 1, \ldots, b$ *samples* **do** calculate Q-targets
            if $\boldsymbol{x}_{i+1}$ *is terminal* **then** $y_i = r_{i+1}$;
            else $y_i = r_{i+1} + \gamma \max_u \hat{q}(\boldsymbol{x}_{i+1}, u, \boldsymbol{w}^-)$;
        fit $\boldsymbol{w}$ on loss $\mathcal{L}(\boldsymbol{w}) = [y_i - \hat{q}(\boldsymbol{x}_i, u_i, \boldsymbol{w})]^2_{\boldsymbol{\mathcal{D}}_b}$;
        if $k \mod k_w = 0$ **then** $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$ (update target weights);

Algo. 11.6: DQN (output: parameter vector $\boldsymbol{w}$ for $\hat{q}^*$)

Remarks on DQN implementation

- General framework is based on V. Mnih et al., *Human-level control through deep reinforcement learning*, Nature, pp. 529-533, 2015.
- Often 'deep' artificial neural networks are used as function approximation for DQN.
  - Nevertheless, other model topologies are fully conceivable.
- The fit of $\boldsymbol{w}$ on loss $\mathcal{L}$ is an intermediate supervised learning step.
  - Comes with degrees of freedom regarding solver choice.
  - Has own optimization parameters which are not depicted here in details (many tuning options).
- Mini-batch sampling from $\boldsymbol{\mathcal{D}}$ is often randomly distributed.
  - Nevertheless, guided sampling with useful distributions for a specific control task can be beneficial (cf. Dyna discussion in 7th lecture).
- Likewise, the simple ε-greedy approach can be extended.
  - Often a scheduled/annealed trajectory ε_k is used.
  - Again referring to the Dyna framework, many more exploration strategies are possible.

DQN application example: Atari games (1)

- End-to-end learning of $\hat{q}(\boldsymbol{x}, u)$ from monitor pixels $\boldsymbol{x}$
- Feature engineering obtains stacking of raw pixes from last 4 frames
- Actions u are 18 possible joystick/button combinations
- Reward is the change of highscore per step
- Interesting lecture from V. Minh with more details: YouTube

Fig. 11.9: Network architecture overview used for DQN in Atari games (source: D. Silver, "Reinforcement learning", 2015. CC BY-NC 4.0)

DQN application example: Atari games (2)

Fig. 11.10: DQN performance results in Atari games against human performance (source: D. Silver, "Reinforcement learning", 2015. CC BY-NC 4.0)

Summary: what you've learned today

- From a simplified perspective, the procedures from the approximate prediction can simply be transferred to value-based control.
- On the contrary, the policy improvement theorem no longer applies in the approximate RL case (generalization impact).
  - Control algorithms may diverge completely.
  - Or a performance trade-off between different parts of the problem space could emerge.
- Off-policy batch learning approaches allow for efficient data usage.
  - LSPI uses LS-SARSA on linear function approximation.
  - DQN extends Q-learning on non-linear approximation with additional tweaks (experience replay, target networks,...).
  - However, a prediction bias results (off-policy sampling distribution).

Table of contents

12 Stochastic policy gradient methods
- Stochastic policy approximation and the policy gradient theorem
- Monte Carlo policy gradient
- Actor-critic methods

Preface (1)

Shift from (indirect) value-based approaches

$$\hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \approx q(\boldsymbol{x}, u) \tag{12.1}$$

to (direct) policy-based solutions:

$$\boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x}) = \mathbb{P}\left[\boldsymbol{U} = \boldsymbol{u}|\boldsymbol{X} = \boldsymbol{x}\right] \approx \boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})\,. \tag{12.2}$$

- Above, $\boldsymbol{\theta} \in \mathbb{R}^d$ is the policy parameter vector.
- Note, that $\boldsymbol{u}$ is now vectorial and might contain multiple continuous quantities.

Goal of today's lecture

- Introduce an algorithm class based on a parameterizable policy $\pi(\boldsymbol{\theta})$.
- Extend the action space to continuous actions.
- Combine the policy-based and value-based approach.

Preface (2)

Fig. 12.1: Main categories of reinforcement learning algorithms
(source: D. Silver, "Reinforcement learning", 2015. CC BY-NC 4.0)

Motivating example: strategic gaming

Task: Two-player game of extended rock-paper-scissors

- A deterministic policy (i.e., value-based with given feature representation) can be easily exploited by the opponent.
- Conversely, a uniform random policy would be unpredictable.

Fig. 12.2: Rock paper scissors lizard Spock game mechanics
(source: www.wikipedia.org, by Diriector Doc CC BY-SA 4.0)

Example policy function: discrete action space

Assumption:

- Action space is discrete and compact.

A typical policy function is:

- Soft-max in action preferences

$$\pi(u|\boldsymbol{x}, \boldsymbol{\theta}) = \frac{e^{h(\boldsymbol{x},u,\boldsymbol{\theta})}}{\sum_i e^{h(\boldsymbol{x},i,\boldsymbol{\theta})}} \tag{12.3}$$

  with $h(\boldsymbol{x}, u, \boldsymbol{\theta}) : \mathcal{X} \times \mathcal{U} \times \mathbb{R}^d \to \mathbb{R}$ being the numerical preference per state-action pair.
- Denominator of (12.3) sums up action probabilities to one per state.
- Is designed as a stochastic policy but can approach deterministic behavior in the limit.
- The preference is parametrized via a function approximator, e.g., linear in features

$$h(\boldsymbol{x}, u, \boldsymbol{\theta}) = \boldsymbol{\theta}^\mathsf{T} \tilde{\boldsymbol{x}}(\boldsymbol{x}, u). \tag{12.4}$$

Example policy function: continuous action space (1)

Assumption:

- Action space is continuous and there is only one scalar action $u \in \mathbb{R}$.

A typical policy function is:

- Gaussian probability density

$$\pi(u|\boldsymbol{x}, \boldsymbol{\theta}) = \frac{1}{\sigma(\boldsymbol{x}, \boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(u - \mu(\boldsymbol{x}, \boldsymbol{\theta}))^2}{2\sigma(\boldsymbol{x}, \boldsymbol{\theta})^2}\right) \tag{12.5}$$

  with mean $\mu(\boldsymbol{x}, \boldsymbol{\theta}) : \mathcal{X} \times \mathbb{R}^d \to \mathbb{R}$ and standard deviation $\sigma(\boldsymbol{x}, \boldsymbol{\theta}) : \mathcal{X} \times \mathbb{R}^d \to \mathbb{R}$ given by parametric function approximation.
- Variants regarding function μ and σ:
  1. Both share a mutual parameter set $\boldsymbol{\theta}$ (e.g., artificial neural network with multiple outputs).
  2. Both are parametrized independently $\boldsymbol{\theta} = \begin{bmatrix}\boldsymbol{\theta}_\mu & \boldsymbol{\theta}_\sigma\end{bmatrix}^\mathsf{T}$ (e.g., by two linear regression functions).
  3. Only $\mu(\boldsymbol{x}, \boldsymbol{\theta})$ is parametrized while σ is scheduled externally.

Example policy function: continuous action space (2)

- Output of the functions $\mu_k = (\boldsymbol{x}_k, \boldsymbol{\theta}_k)$ and $\sigma_k = (\boldsymbol{x}_k, \boldsymbol{\theta}_k)$ can change in every time step.
- Depending on σ exploration is an inherent part of the (stochastic) policy.

Fig. 12.3: Exemplary univariate Gaussian probability density functions (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Example policy function: continuous action space (3)

Assumption:

- Action space is continuous and there are multiple actions $\boldsymbol{u} \in \mathbb{R}^m$.

A typical policy function is:

- Multivariate Gaussian probability density

$$\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) = \frac{1}{\sqrt{(2\pi)^m \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{u} - \boldsymbol{\mu})^\mathsf{T} \boldsymbol{\Sigma}^{-1} (\boldsymbol{u} - \boldsymbol{\mu})\right) \tag{12.6}$$

  with mean $\boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta}) : \mathcal{X} \times \mathbb{R}^d \to \mathbb{R}^m$ and covariance matrix $\boldsymbol{\Sigma}(\boldsymbol{x}, \boldsymbol{\theta}) : \mathcal{X} \times \mathbb{R}^d \to \mathbb{R}^{m \times m}$ given by parametric function approximation.
- Same parametrization variants apply to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as in the scalar action case.
- In addition, $\boldsymbol{\Sigma}$ can be considered a diagonal matrix and clipped to reduce complexity as well as ensure nonsingularity.

Example policy function: continuous action space (4)

- Below we find an example for

$$\boldsymbol{\mu} = \begin{bmatrix} -0.4 & 0.3 \end{bmatrix}^\mathsf{T} \quad \text{and} \quad \boldsymbol{\Sigma} = \begin{bmatrix} 0.04 & 0 \\ 0 & 0.02 \end{bmatrix}.$$

Fig. 12.4: Exemplary bivariate Gaussian probability density function

Policy objective function

- Goal: find optimal $\boldsymbol{\theta}^*$ given the policy $\boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$.
- Problem: which measure of optimality should we use?

Possible optimality metrics:

- Start state value (in episodic tasks):

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{x}_0) = \mathbb{E}\left[v|\boldsymbol{X} = \boldsymbol{x}_0, \boldsymbol{\theta}\right] \tag{12.7}$$

- Average reward (in continuing tasks):

$$J(\boldsymbol{\theta}) = \overline{r}_{\pi_{\boldsymbol{\theta}}} = \int_{\mathcal{X}} \mu_\pi(\boldsymbol{x}) \int_{\mathcal{U}} \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \int_{\mathcal{X},\mathcal{R}} p(\boldsymbol{x}', r|\boldsymbol{x}, \boldsymbol{u}) r \tag{12.8}$$

  - Above, $\mu_\pi(\boldsymbol{x})$ is again the steady-state distribution $\mu_\pi(\boldsymbol{x}) = \lim_{k\to\infty} \mathbb{P}\left[\boldsymbol{X}_k = \boldsymbol{x}|\boldsymbol{U}_{0:k-1} \sim \pi\right]$.

Policy optimization

- In essence, policy-based RL is an optimization problem.
- Depending on the policy function and task, finding $\boldsymbol{\theta}^*$ might be a
  - non-linear,
  - multidimensional and
  - non-stationary problem.
- Hence, we might consider global optimization techniques[1] like
  - Simple heuristics: random search, grid search,...
  - Meta-heuristics: evolutionary algorithms, particle swarm,....
  - Surrogate-model-based optimization: Bayes opt.,...
  - Gradient-based techniques with multi-start initialization.

[1] Recommended reading: J. Stork et al., *A new Taxonomy of Continuous Global Optimization Algorithms*, https://arxiv.org/abs/1808.08818, 2020

Policy gradient

Fig. 12.5: Exemplary optimization paths for (stochastic) gradient ascent
(derivative work of www.wikipedia.org, CC0 1.0)

- We will focus on gradient-based methods (policy gradient).
- Hence, we will assume that the gradient

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} & \cdots & \frac{\partial J}{\partial \theta_d} \end{bmatrix}^\mathsf{T}$$

  required for gradient ascent optimization always exists:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

- True gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is usually approximated, e.g., by stochastic gradient descent (SGD) or derived variants.

Policy gradient theorem

Theorem 12.1: Policy Gradient

Given a metric $J(\boldsymbol{\theta})$ for the undiscounted episodic (12.7) or continuing tasks (12.8) and a parameterizable policy $\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})$ the policy gradient is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[q_{\pi}(\boldsymbol{x},\boldsymbol{u})\frac{\nabla_{\boldsymbol{\theta}}\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})}{\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})}\right]. \tag{12.9}$$

- Having samples $\langle \boldsymbol{x}_i, \boldsymbol{u}_i \rangle$, an estimate of q_{π} and the policy function $\pi(\boldsymbol{\theta})$ available we receive an analytical solution for the policy gradient!
- Using identity $\nabla \ln a = \frac{\nabla a}{a}$ we can re-write to

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[q_{\pi}(\boldsymbol{x},\boldsymbol{u})\nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})\right] \tag{12.10}$$

  with $\nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})$ also called the score function.
- Derivation available in chapter 13.2 / 13.6 in the lecture book of Barto and Sutton.

Intuitive interpretation of policy parameter update

- Inserting the policy gradient theorem into gradient ascent approach:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \mathbb{E}_\pi \left[q_\pi(\boldsymbol{x}, \boldsymbol{u}) \frac{\nabla_{\boldsymbol{\theta}} \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})}{\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})} \right].$$

- Move in the direction that favor actions that yield an increased value.
- Scale the update step size inversely to the action probability to compensate that some actions are selected more frequently.

Also note:

- The policy gradient is not depending on the state distribution!
- Hence, we do not need any knowledge of the environment and receive a model-free RL approach!

Simple score function examples

Soft-max policy with linear function approximation:

$$\pi(u|\boldsymbol{x},\boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u)}}{\sum_i e^{\boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},i)}}$$

$$\Leftrightarrow \quad \nabla_{\boldsymbol{\theta}} \ln \pi(u|\boldsymbol{x},\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \left(\boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u) - \ln \left(\sum_i e^{\boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},i)} \right) \right)$$

$$= \tilde{\boldsymbol{x}}(\boldsymbol{x},\boldsymbol{u}) - \mathbb{E}_\pi \left[\tilde{\boldsymbol{x}}(\boldsymbol{x},\cdot) \right]$$

Univariate Gaussian policy with linear function approximation and given σ:

$$\pi(u|\boldsymbol{x},\boldsymbol{\theta}) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(u - \boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u))^2}{2\sigma^2} \right)$$

$$\Leftrightarrow \quad \nabla_{\boldsymbol{\theta}} \ln \pi(u|\boldsymbol{x},\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \left(\ln \left(\frac{1}{\sigma\sqrt{2\pi}} \right) - \frac{(u - \boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u))^2}{2\sigma^2} \right)$$

$$= \frac{\left(u - \boldsymbol{\theta}^\mathsf{T}\tilde{\boldsymbol{x}}(\boldsymbol{x},u)\right)\tilde{\boldsymbol{x}}(\boldsymbol{x},u)}{\sigma^2}$$

Pro and cons: policy vs. value-based approaches

Pro value-based solutions (e.g., Q-learning):

- Estimated value is an intuitive performance metric.
- Considered sample-efficient (cf. replay buffer or bootstrapping).

Pro policy-based solutions (e.g., using policy gradient):

- Seamless integration of stochastic and dynamic policies.
- Straightforward applicable to large/continuous action spaces. In contrast, value-based approaches would require explicit optimization

$$\boldsymbol{u}^* = \arg\max_{\boldsymbol{u}} q(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}).$$

Mutual hassle:

- Gradient-based optimization with (non-linear) function approximation is likely to deliver only suboptimal and local policy optima.

Table of contents

12 Stochastic policy gradient methods
- Stochastic policy approximation and the policy gradient theorem
- Monte Carlo policy gradient
- Actor-critic methods

Basic concept

Initial situation:

- Score function $\nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$ can be calculated analytically using suitable policy and chain rule (e.g., by algorithmic differentiation).
- Open question: how to retrieve $q_\pi(\boldsymbol{x}, \boldsymbol{u})$ in

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[q_\pi(\boldsymbol{x}, \boldsymbol{u}) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \right] \ ?$$

Monte Carlo policy gradient:

- Use sampled episodic return g_k to approximate $q_\pi(\boldsymbol{x}, \boldsymbol{u})$:

$$q_\pi(\boldsymbol{x}, \boldsymbol{u}) \approx g_k$$

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \gamma^k g_k \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k, \boldsymbol{\theta}_k).$$

- The discounting of the policy gradient is introduced as an extension to Theo. 12.1 (which assumed an undiscounted episodic task).
- Also known as *REINFORCE* approach.

Algorithmic implementation: Monte Carlo policy gradient (REINFORCE)

- Usual technical convergence requirements regarding α apply.
- Use sampled return as unbiased estimate of q.
- Recall previous MC-based methods: high variance, slow learning.

input: a differentiable policy function $\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})$
parameter: step size $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$
init: parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate an episode following $\pi(\cdot|\cdot,\boldsymbol{\theta})$: $\boldsymbol{x}_0, \boldsymbol{u}_0, r_1, \ldots, \boldsymbol{x}_T$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $g \leftarrow \sum_{i=k+1}^{T} \gamma^{i-k-1} r_i$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^k g \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k,\boldsymbol{\theta})$;

Algo. 12.1: Monte Carlo policy gradient (output: parameter vector $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta}^*)$)

REINFORCE example: short-corridor problem (1)

- Gridworld style problem with two actions: left (l), right (r)
- Second-left state's action execution is reversed
- Feature representation: $\tilde{x}(x, u = \mathrm{r}) = \begin{bmatrix}1 & 0\end{bmatrix}^\mathsf{T}$, $\tilde{x}(x, u = \mathrm{l}) = \begin{bmatrix}0 & 1\end{bmatrix}^\mathsf{T}$
- A policy-based approach searches for the optimal probability split

Fig. 12.6: Short-corridor problem with $\varepsilon = 0.1$ (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

REINFORCE example: short-corridor problem (2)

Fig. 12.7: Comparison of Monte Carlo policy gradient approach on short-corridor problem from Fig. 12.6 for different learning rates (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Baseline

- Motivation: add a comparison term to the policy gradient to reduce variance while not affecting its expectation.
- Introduce the baseline $b(\boldsymbol{x})$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[\left(q_{\pi}(\boldsymbol{x}, \boldsymbol{u}) - b(\boldsymbol{x})\right) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})\right]. \tag{12.11}$$

- Since $b(\boldsymbol{x})$ is only depending on the state but not on the actions/policy we did not change the policy gradient in expectation:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}\left[q_{\pi}(\boldsymbol{x}, \boldsymbol{u}) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})\right] - \underbrace{\mathbb{E}_{\pi}\left[b(\boldsymbol{x}) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})\right]}_{=0}.$$

- Consequently, the Monte Carlo policy parameter update yields:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \gamma^k \left(g_k - b(\boldsymbol{x}_k)\right) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k, \boldsymbol{\theta}_k).$$

Advantage function

- Intuitive choice of the baseline is the state value $b(\boldsymbol{x}) = v_\pi(\boldsymbol{x})$.
- The resulting policy gradient becomes

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[\left(q_\pi(\boldsymbol{x}, \boldsymbol{u}) - v_\pi(\boldsymbol{x}) \right) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \right]. \tag{12.12}$$

- Here, the difference between action and state value is the advantage function

$$a_\pi(\boldsymbol{x}, \boldsymbol{u}) = q_\pi(\boldsymbol{x}, \boldsymbol{u}) - v_\pi(\boldsymbol{x}). \tag{12.13}$$

- Interpretation: value difference taking (arbitrary) action $\boldsymbol{u}$ and thereafter following policy π compared to the state value following same policy (i.e., choosing $\boldsymbol{u} \sim \pi$) given the state.
- Hence, we might rewrite to:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[a_\pi(\boldsymbol{x}, \boldsymbol{u}) \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}) \right]. \tag{12.14}$$

Algo. implementation: MC policy gradient with baseline

- Implementation requires approximation $b(\boldsymbol{x}) \approx \hat{v}(\boldsymbol{x}, \boldsymbol{w})$.
- Hence, we are learning two parameter sets $\boldsymbol{\theta}$ and $\boldsymbol{w}$.
- Keep using sampled return as action-value estimate: $q_\pi(\boldsymbol{x}, \boldsymbol{u}) \approx g_k$.

input: a differentiable policy function $\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$ and state-value function $\hat{v}(\boldsymbol{x}, \boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha < 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  generate an episode following $\pi(\cdot|\cdot, \boldsymbol{\theta})$: $\boldsymbol{x}_0, \boldsymbol{u}_0, r_1, \ldots, \boldsymbol{x}_T$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $g \leftarrow \sum_{i=k+1}^{T} \gamma^{i-k-1} r_i$;
    $\delta \leftarrow g - \hat{v}(\boldsymbol{x}_k, \boldsymbol{w})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w \delta \nabla_{\boldsymbol{w}} \hat{v}(\boldsymbol{x}_k, \boldsymbol{w})$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \gamma^k \delta \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k, \boldsymbol{\theta})$;

Algo. 12.2: Monte Carlo policy gradient with baseline (output: parameter vector $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}^*)$) and $\boldsymbol{w}^*$ for $\hat{v}^*(\boldsymbol{x}, \boldsymbol{w}^*)$)

REINFORCE comparison w/o baseline

Fig. 12.8: Comparison of Monte Carlo policy gradient on short-corridor problem from Fig. 12.6 where both algorithms' learning rates have been tuned (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Table of contents

12 Stochastic policy gradient methods
- Stochastic policy approximation and the policy gradient theorem
- Monte Carlo policy gradient
- Actor-critic methods

General actor-critic idea

Conclusion of Monte Carlo policy gradient with baseline:

- Will learn an unbiased policy gradient.
- As the other MC-based methods: learns slowly due to high variance.
- Updates only available after full episodes.

Alternative: use an additional function approximator, the so-called critic, to estimate q_π (i.e., approximate policy gradient):

$$\begin{aligned} v(\boldsymbol{x}) &\approx \hat{v}(\boldsymbol{x}, \boldsymbol{w}_v), \\ q(\boldsymbol{x}, \boldsymbol{u}) &\approx \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}_q), \\ a(\boldsymbol{x}, \boldsymbol{u}) &\approx \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}_q) - \hat{v}(\boldsymbol{x}, \boldsymbol{w}_v). \end{aligned}$$

- Realization: any prediction tool discussed so far (TD(0), LSTD,...).
- Potential: we can use online step-by-step updates to estimate $\hat{q}$.
- Disadvantage: we would train two value estimates by $\boldsymbol{w}_v$ and $\boldsymbol{w}_q$.

Integrating the advantage function

- The TD error is

$$\delta_\pi = r + \gamma v_\pi(\boldsymbol{x}') - v_\pi(\boldsymbol{x}). \tag{12.15}$$

- In expectation the TD error is equivalent to the advantage function

$$\begin{aligned}\mathbb{E}_\pi\left[\delta_\pi|\boldsymbol{x},\boldsymbol{u}\right] &= \mathbb{E}_\pi\left[r + \gamma v_\pi(\boldsymbol{x}')|\boldsymbol{x},\boldsymbol{u}\right] - v_\pi(\boldsymbol{x}) \\ &= q_\pi(\boldsymbol{x},\boldsymbol{u}) - v_\pi(\boldsymbol{x}) \\ &= a_\pi(\boldsymbol{x},\boldsymbol{u}).\end{aligned} \tag{12.16}$$

- Hence, the TD error can be used to calculate the policy gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_\pi\left[\delta_\pi \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})\right]. \tag{12.17}$$

- This results in requiring only one function parameter set:

$$\delta_\pi \approx r + \gamma \hat{v}_\pi(\boldsymbol{x}',\boldsymbol{w}) - \hat{v}_\pi(\boldsymbol{x},\boldsymbol{w}). \tag{12.18}$$

Actor-critic structure

- Critic (policy evaluation) and actor (policy improvement) can be considered another form of generalized policy iteration (GPI).
- Online and on-policy algorithm for discrete and continuous action spaces with built-in exploration by stochastic policy functions.

Fig. 12.9: Simplified flow diagram of actor-critic-based RL

Algo. implementation: actor-critic with TD(0) targets

- Analog to MC-based policy gradient optional discounting on the gradient updates is introduced.

input: a differentiable policy function $\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})$ and state-value function $\hat{v}(\boldsymbol{x},\boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha < 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{x}_0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    apply $\boldsymbol{u}_k \sim \pi(\cdot|\boldsymbol{x}_k,\boldsymbol{\theta})$ and observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
    $\delta \leftarrow r_{k+1} + \gamma\hat{v}(\boldsymbol{x}_{k+1},\boldsymbol{w}) - \hat{v}(\boldsymbol{x}_k,\boldsymbol{w})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w\delta\nabla_{\boldsymbol{w}}\hat{v}(\boldsymbol{x}_k,\boldsymbol{w})$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta\gamma^k\delta\nabla_{\boldsymbol{\theta}}\ln\pi(\boldsymbol{u}_k|\boldsymbol{x}_k,\boldsymbol{\theta})$;

Algo. 12.3: Actor-critic for episodic tasks using TD(0) targets (output: parameter vector $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta}^*)$) and $\boldsymbol{w}^*$ for $\hat{v}^*(\boldsymbol{x},\boldsymbol{w}^*)$)

Actor-critic generalization

- Using the TD(0) error as the target to train the critic is convenient.
- However, the usual alternatives can be applied to train $\hat{v}(\boldsymbol{x}, \boldsymbol{w})$.
- n-step bootstrapping:

$$v(\boldsymbol{x}_k) \approx r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{v}_{k+n-1}(\boldsymbol{x}_{k+n}, \boldsymbol{w}).$$

- λ-return (forward view):

$$v(\boldsymbol{x}_k) \approx (1-\lambda) \sum_{n=1}^{T-k-1} \lambda^{(n-1)} g_{k:k+n} + \lambda^{T-k-1} g_k.$$

- TD(λ) using eligibility traces (backward view):

$$\begin{aligned} \boldsymbol{z}_k &= \gamma\lambda \boldsymbol{z}_{k-1} + \nabla_{\boldsymbol{w}} \hat{v}(\boldsymbol{x}_k, \boldsymbol{w}_k), \\ \delta_k &= r_{k+1} + \gamma \hat{v}(\boldsymbol{x}_{k+1}, \boldsymbol{w}_k) - \hat{v}(\boldsymbol{x}_k, \boldsymbol{w}_k). \end{aligned}$$

Algo. implementation: actor-critic with TD(λ) targets

input: a differentiable policy function $\pi(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta})$
input: a differentiable state-value function $\hat{v}(\boldsymbol{x}, \boldsymbol{w})$
parameter: $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha < 1\}$, $\{\lambda_w, \lambda_\theta\} \in \{\mathbb{R}|0 \leq \lambda \leq 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{x}_0$, $\boldsymbol{z}_w = 0$, $\boldsymbol{z}_\theta = 0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    apply $\boldsymbol{u}_k \sim \pi(\cdot|\boldsymbol{x}_k, \boldsymbol{\theta})$ and observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
    $\delta \leftarrow r_{k+1} + \gamma \hat{v}(\boldsymbol{x}_{k+1}, \boldsymbol{w}) - \hat{v}(\boldsymbol{x}_k, \boldsymbol{w})$;
    $\boldsymbol{z}_w \leftarrow \gamma \lambda_w \boldsymbol{z}_w + \nabla_{\boldsymbol{w}} \hat{v}(\boldsymbol{x}_k, \boldsymbol{w})$;
    $\boldsymbol{z}_\theta \leftarrow \gamma \lambda_d \boldsymbol{z}_\theta + \gamma^k \nabla_{\boldsymbol{\theta}} \ln \pi(\boldsymbol{u}_k|\boldsymbol{x}_k, \boldsymbol{\theta})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w \delta \boldsymbol{z}_w$;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \delta \boldsymbol{z}_\theta$;

Algo. 12.4: Actor-critic for episodic tasks using TD(λ) targets (output: parameter vector $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x}, \boldsymbol{\theta}^*)$ and $\boldsymbol{w}^*$ for $\hat{v}^*(\boldsymbol{x}, \boldsymbol{w}^*)$)

Summary: what you've learned today

- Policy-based methods are a new class within the RL toolbox.
  - Instead of learning a policy indirectly from a value the policy is directly parametrized.
  - The policy function allows discrete and continuous actions with inherent stochastic exploration.
- Solving the underlying optimization task is complex. However, the policy gradient theorem provides a suitable theoretical baseline for gradient-based optimization.
- Anyhow, to calculate policy gradients we require a value estimate.
  - Monte Carlo prediction is straightforward, but comes with high variance and slow learning.
  - Adding a state-dependent baseline comparison does not change the policy gradient in expectation but enables decreasing the variance.
- Extending this idea naturally leads to integrating a critic network, i.e., an additional function approximation to estimate the value.
- The critic can be fed by the usual targets (TD(0), TD(λ),...).

Table of contents

13 Deterministic policy gradient methods
- Deep deterministic policy gradient (DDPG)
- Twin delayed deep deterministic policy gradient (TD3)

Background and motivation

Recap on policy gradient so far:

- The previously discussed policy functions and the policy gradient theorem were assuming stochastic polices.
- The resulting on-policy algorithms may not provide top-class learning performance:
  - Non-guided exploration with step-by-step updates and
  - Greedy actions only in the limit (i.e., infeasible long learning).

The alternative:

- Apply a deterministic policy with separate exploration.
- Enable off-policy learning (with experience replay as a possible extension).
- Hence, we will focus on a deterministic policy function

$$\pi(\boldsymbol{x}, \boldsymbol{\theta}) = \mu(\boldsymbol{x}, \boldsymbol{\theta}). \tag{13.1}$$

Deterministic policy gradient (DPG) theorem

Theorem 13.1: Deterministic Policy Gradient

Given a metric $J(\boldsymbol{\theta})$ for the undiscounted episodic (12.7) or continuing tasks (12.8) and a parameterizable policy $\mu(\boldsymbol{x}, \boldsymbol{\theta})$ the deterministic policy gradient is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mu}\left[\nabla_{\boldsymbol{\theta}} \mu(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} q(\boldsymbol{x}, \boldsymbol{u})|_{\boldsymbol{u}=\mu(\boldsymbol{x})}\right]. \tag{13.2}$$

- Again, q needs to be approximated using samples, e.g., implementing a critic via TD learning.
- It turns out that (13.2) is also (approximately) valid in the off-policy case, i.e., if the sample distribution is obtained from a behavior policy.
- Proof can be found in D. Silver et al., *Deterministic Policy Gradient Algorithms*, International Conference on Machine Learning, 2014

Exploration with a deterministic policy

- If the DPG approach is applied on-policy there is no inherent exploration.
- How to learn something?
  - The environment itself is sufficiently noisy (random impacts, measurement noise).
  - Or we have to add noise to the actions, i.e., making the approach off-policy.
  - Hence, utilizing a behavior policy is also possible.
- That additional action noise could be:
  - Simple Gaussian noise or
  - a shaped noise process like a discrete-time Ornstein-Uhlenbeck (OU) process

$$\nu_{k+1} = \lambda \nu_k + \sigma \epsilon_k$$

where ν_k is the OU noise output, $0 < \lambda < 1$ is a smoothing factor and σ is the variance scaling a standard Gaussian sequence (no mean) ϵ_k.

Algo. implementation: deterministic actor-critic

input: a differentiable deterministic policy function $\mu(\boldsymbol{x}, \boldsymbol{\theta})$
input: a differentiable action-value function $\hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R} | 0 < \alpha < 1\}$
init: parameter vectors $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily
for $j = 1, 2, \ldots,$ *episodes* **do**
    initialize $\boldsymbol{x}_0$;
    for $k = 0, 1, \ldots, T-1$ *time steps* **do**
        $\boldsymbol{u}_k \leftarrow$ apply from $\mu(\boldsymbol{x}_k, \boldsymbol{\theta})$ w/wo noise or from behavior policy;
        observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
        choose $\boldsymbol{u}'$ from $\mu(\boldsymbol{x}_{k+1}, \boldsymbol{\theta})$;
        $\delta \leftarrow r_{k+1} + \gamma \hat{q}(\boldsymbol{x}_{k+1}, \boldsymbol{u}', \boldsymbol{w}) - \hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w})$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_w \delta \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w})$;
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \gamma^k \nabla_{\boldsymbol{\theta}} \mu(\boldsymbol{x}_k, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w})|_{\boldsymbol{u}=\mu(\boldsymbol{x})}$;

Algo. 13.1: Deterministic actor-critic for episodic tasks using SARSA(0) targets applicable on- and off-policy (output: parameter vector $\boldsymbol{\theta}^*$ for $\mu^*(\boldsymbol{x}, \boldsymbol{\theta}^*)$) and $\boldsymbol{w}^*$ for $\hat{q}^*(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}^*)$)

Exemplary comparison to stochastic policy gradient

- DPG-based approach uses compatible function approximation, i.e., suitable linear $\hat{q}$ estimation. A fixed Gaussian behavior policy is applied for exploration.
- SAC uses a Gaussian policy with linear function approximation.

(a) Mountain Car (b) Pendulum (c) 2D Puddle World

Fig. 13.1: Comparison of stochastic on-policy actor-critic (SAC), stochastic off-policy actor-critic (OffPAC), and deterministic off-policy actor-critic (COPDAC) on continuous-action reinforcement learning (source: D. Silver et al., *Deterministic Policy Gradient Algorithms*, International Conference on Machine Learning, 2014)

Motivation / general idea

- The upcoming deep deterministic policy gradient (DDPG) algorithm was very much inspired by the successes of DQNs (cf. Algo. 11.6 and landmark paper by Mnih et al.) on discrete action spaces.
- However, DQNs are not directly applicable to (quasi-)continuous action spaces.
- Recall the incremental Q-learning equation using function approximation

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[r + \gamma \max_u \hat{q}(\boldsymbol{x}', u, \boldsymbol{w}) - \hat{q}(\boldsymbol{x}, u, \boldsymbol{w}) \right] \nabla_{\boldsymbol{w}} \hat{q}(\boldsymbol{x}, u, \boldsymbol{w}).$$

- For every policy inference and updating step we need to find $\max_u \hat{q}(\boldsymbol{x}', u, \boldsymbol{w})$.
- If $u \in \mathcal{U} \subset \mathbb{Z}$ (i.e., using integer-encoded actions) is a sufficiently small discrete set, that is straightforward by an exhaustive search.
- In contrast, if $\boldsymbol{u} \in \mathcal{U} \subset \mathbb{R}^m$ is a (quasi-)continuous variable solving $\max_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}', \boldsymbol{u}, \boldsymbol{w})$ requires an own optimization routine which is computationally expensive if we use nonlinear function approximation.

The deterministic policy trick

- When using a greedy, deterministic policy $\boldsymbol{\pi}(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta})$ we can utilize it to approximate

$$\max_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}', \boldsymbol{u}, \boldsymbol{w}) \approx \hat{q}(\boldsymbol{x}', \boldsymbol{\mu}(\boldsymbol{x}', \boldsymbol{\theta}), \boldsymbol{w}). \tag{13.3}$$

- Hence, we can obtain explicit Q-learning targets for continuous actions when using a deterministic policy.
- For improving the policy we reuse the deterministic policy gradient theorem in an off-policy fashion

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_b \left[\nabla_{\boldsymbol{\theta}} \boldsymbol{\mu}(\boldsymbol{X}, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} q(\boldsymbol{X}, \boldsymbol{U}) \,|\boldsymbol{U} = \boldsymbol{\mu}(\boldsymbol{X}, \boldsymbol{\theta}) \right] \tag{13.4}$$

given a behavior policy $b(\boldsymbol{u}|\boldsymbol{x})$.

DDPG $\approx$ DQN + DPG

- Hence, we can consider the DDPG approach as a combination of DQN + DPG rendering it an actor-critic off-policy approach for continuous state and action spaces.
- Similarly to DQN we will introduce several 'tweaks' to stabilize and improve the DDPG learning process.

Tweak #1: experience replay buffer

- We store $\langle \boldsymbol{x}, \boldsymbol{u}, r, \boldsymbol{x}' \rangle$ in $\boldsymbol{\mathcal{D}}$ after each transition step.
- The replay buffer $\boldsymbol{\mathcal{D}}$ is of limited capacity, i.e., it discards the oldest data sample when updating once it is full (ring memory).
- This allows us to improve the Q-learning critic minimizing the mean-squared Bellman error (MSBE):

$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma q(\boldsymbol{x}', \boldsymbol{\mu}(\boldsymbol{x}', \boldsymbol{\theta}), \boldsymbol{w}) \right) - q(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}) \right]^2_{\boldsymbol{\mathcal{D}}} . \tag{13.5}$$

Additional DDPG tweaks (1)

Tweak #2: target networks

- Similar to DQN we introduce a (delayed) target network to estimate the Q-learning target

$$r + \gamma q(\boldsymbol{x}', \boldsymbol{\mu}(\boldsymbol{x}', \boldsymbol{\theta}), \boldsymbol{w})$$

  since it depends on the same parameters $\boldsymbol{w}$ which we want to update.
- Hence, the target network's purpose it to mimic the generation of i.i.d. data as the ground truth to minimize (13.5).
- Since the policy parameters $\boldsymbol{\theta}$ are also part of the target calculation it turns out that an additional policy target network is also beneficial to stabilize the Q-learning.
- In contrast to the classical DQN implementation, the original DDPG algorithm does not perform periodically hard target network updates but continuous ones using a low-pass filter characteristic

$$\boldsymbol{w}^- \leftarrow (1-\tau)\boldsymbol{w}^- + \tau\boldsymbol{w}, \quad \boldsymbol{\theta}^- \leftarrow (1-\tau)\boldsymbol{\theta}^- + \tau\boldsymbol{\theta} \tag{13.6}$$

  with τ representing the equivalent filter constant (hyperparameter).

Additional DDPG tweaks (2)

Tweak #3: mini-batch sampling

- Given a sufficiently filled memory $\boldsymbol{\mathcal{D}}$ and the target networks parametrized by $\boldsymbol{w}^-$ and $\boldsymbol{\theta}^-$ we draw uniformly distributed mini-batch samples $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$.
- The actual Q-learning is then based on the loss

$$\mathcal{L}(\boldsymbol{w}) = \left[\left(r + \gamma q(\boldsymbol{x}', \boldsymbol{\mu}(\boldsymbol{x}', \boldsymbol{\theta}^-), \boldsymbol{w}^-)\right) - q(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})\right]^2_{\boldsymbol{\mathcal{D}}_b} . \tag{13.7}$$

Tweak #4: batch normalization

- Minimizing (13.7) is a supervised learning step within the DDPG.
- The original DDPG paper by Lillicrap et al. back in 2015/16 suggested to use batch normalization, i.e., re-centering and re-scaling the inputs of each layer in an ANN.
- This idea of batch normalization was presented at that time shortly before by Ioffe and Szegedy (cf. original paper).
- Today's perspective: stick to the current state-of-the-art supervised ML algorithms for top-class Q-learning stability and speed (which are normally well-covered in popular supervised ML toolboxes).

Additional DDPG tweaks (3)

Tweak #5: exploration

- Since our policy is deterministic we require an exploratory behavior policy.
- Similar to DPG the standard approach is to add noise to the greedy actions, e.g., again from an Ornstein-Uhlenbeck (OU) process

$$\boldsymbol{u}_k \sim \boldsymbol{b}(\boldsymbol{u}|\boldsymbol{x}_k) = \boldsymbol{\mu}(\boldsymbol{x}_k, \boldsymbol{\theta}_k) + \boldsymbol{\nu}_k, \quad \boldsymbol{\nu}_k = \lambda \boldsymbol{\nu}_{k-1} + \sigma \boldsymbol{\epsilon}_{k-1}.$$

- One might also add a schedule for λ and σ along the training procedure, e.g., starting with significant noise levels (increased exploration) while reducing it over time (focusing exploitation)[1].
- However, many other behavior policies are possible, e.g., using model or expert-based guidance.

[1] Please note that this 'lambda' is not related to TD(λ), SARSA(λ), etc. Here, it is representing the stiffness of the OU noise process.

Visual summary of DDPG working principle

Fig. 13.2: DDPG structure from a bird's-eye perspective (derivative work of Fig. 1.1 and wikipedia.org, CC0 1.0)

Algo. implementation: DDPG

input: diff. deterministic policy function $\boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta})$ and action-value function $\hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})$
parameter: step sizes and filter constant $\{\alpha_w, \alpha_\theta, \tau\} \in \{\mathbb{R} | 0 < \alpha, \tau < 1\}$
init: weights $\boldsymbol{w} = \boldsymbol{w}^- \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} = \boldsymbol{\theta}^- \in \mathbb{R}^d$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{x}_0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{u}_k \leftarrow$ apply from $\boldsymbol{\mu}(\boldsymbol{x}_k, \boldsymbol{\theta})$ w/wo noise or from behavior policy;
    observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
    store tuple $\langle \boldsymbol{x}_k, \boldsymbol{u}_k, r_{k+1}, \boldsymbol{x}_{k+1} \rangle$ in $\boldsymbol{\mathcal{D}}$;
    sample mini-batch $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$ (after initial memory warmup);
    for $i = 1, \ldots, b$ *samples* **do** calculate Q-targets
      if $\boldsymbol{x}_{i+1}$ *is terminal* **then** $y_i = r_{i+1}$;
      else $y_i = r_{i+1} + \gamma \hat{q}(\boldsymbol{x}_{i+1}, \boldsymbol{\mu}(\boldsymbol{x}_{i+1}, \boldsymbol{\theta}^-), \boldsymbol{w}^-)$;
    fit $\boldsymbol{w}$ on loss $\mathcal{L}(\boldsymbol{w}) = [y - \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})]^2_{\boldsymbol{\mathcal{D}}_b}$ with step size α_w;
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta [\nabla_{\boldsymbol{\theta}} \boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})|_{\boldsymbol{u} = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x})}]_{\boldsymbol{\mathcal{D}}_b}$;
    Update target net. $\boldsymbol{w}^- \leftarrow (1-\tau)\boldsymbol{w}^- + \tau \boldsymbol{w}$, $\boldsymbol{\theta}^- \leftarrow (1-\tau)\boldsymbol{\theta}^- + \tau \boldsymbol{\theta}$;

Algo. 13.2: Deep deterministic policy gradient (output: $\boldsymbol{\theta}^*$ for $\boldsymbol{\mu}^*(\boldsymbol{x}, \boldsymbol{\theta}^*)$ and $\boldsymbol{w}^*$ for $\hat{q}^*(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}^*)$)

Table of contents

13 Deterministic policy gradient methods
- Deep deterministic policy gradient (DDPG)
- Twin delayed deep deterministic policy gradient (TD3)

Overestimation bias

- For Q-learning in the tabular case we have already discussed the maximization bias (cf. Fig. 5.13) issue.
- Recap: Due to the greedy policy targets, $\hat{q}$ was overestimated when calculated using sampled values of stochastic MDPs.
- Additional problem when applying function approximation: the estimator itself introduces additional variance during the learning process which represents another source of the maximization bias problem.

This issue is already known in the DQN context (cf. Algo. 11.6). Similar to the tabular case, double DQN introduces a second Q-network counteracting the overestimation issue (cf. paper by van Hasselt et al.).

However, we did not address this possible problem in an actor-critic context using function approximation (e.g., DDPG).

Overestimation bias in actor-critic approaches (1)

- It turns out that the overestimation bias is also an issue for actor-critic methods[1].
- Consider an actor-critic policy with the current policy parameters $\boldsymbol{\theta}$.
- Let $\tilde{\boldsymbol{\theta}}$ define the parameters from the actor update induced by the maximization of the approximate critic $\hat{q}_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{u})$.
- Let $\boldsymbol{\theta}^*$ be the parameters from the hypothetical actor update w.r.t. the true underlying value function $q^{\boldsymbol{\pi}}(\boldsymbol{x}, \boldsymbol{u})$.
- Then, we perform the policy update

$$\begin{aligned} \tilde{\boldsymbol{\theta}} &= \boldsymbol{\theta} + \frac{\alpha}{Z_1} \mathbb{E}_\pi \left[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\boldsymbol{X}) \nabla_{\boldsymbol{u}} \hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \boldsymbol{U}) \,|\boldsymbol{U} = \boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{X}) \right], \\ \boldsymbol{\theta}^* &= \boldsymbol{\theta} + \frac{\alpha}{Z_2} \mathbb{E}_\pi \left[\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\boldsymbol{X}) \nabla_{\boldsymbol{u}} q^{\boldsymbol{\pi}}(\boldsymbol{X}, \boldsymbol{U}) \,|\boldsymbol{U} = \boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{X}) \right], \end{aligned} \tag{13.8}$$

where Z_1 and Z_2 normalize the gradient such that $Z^{-1} || \mathbb{E}\left[\cdot\right] || = 1$.

[1] Source: S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, https://arxiv.org/abs/1802.09477, 2018

Overestimation bias in actor-critic approaches (2)

- Lets denote $\tilde{\boldsymbol{\pi}}$ and $\boldsymbol{\pi}^*$ as the policies with updated parameters $\tilde{\boldsymbol{\theta}}$ and $\boldsymbol{\theta}^*$ respectively.
- As the gradient direction is a local maximizer, there exists ϵ_1 sufficiently small such that if $\alpha \leq \epsilon_1$ then the *approximate* value of $\tilde{\boldsymbol{\pi}}$ will be bounded below by the *approximate* value of $\boldsymbol{\pi}^*$:

$$\mathbb{E}\left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \tilde{\boldsymbol{\pi}}(\boldsymbol{X}))\right] \geq \mathbb{E}\left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \boldsymbol{\pi}^*(\boldsymbol{X}))\right]. \tag{13.9}$$

- Conversely, there exists ϵ_2 sufficiently small such that if $\alpha \leq \epsilon_2$ then the *true* value of $\tilde{\boldsymbol{\pi}}$ will be bounded above by the *true* value of $\boldsymbol{\pi}^*$:

$$\mathbb{E}\left[q^{\boldsymbol{\pi}}(\boldsymbol{X}, \boldsymbol{\pi}^*(\boldsymbol{X}))\right] \geq \mathbb{E}\left[q^{\boldsymbol{\pi}}(\boldsymbol{X}, \tilde{\boldsymbol{\pi}}(\boldsymbol{X}))\right]. \tag{13.10}$$

- In other words: if the approximate and true critics differ from each other, the according policy gradient updates cannot lead to better policy updates of the respective other framework.

Overestimation bias in actor-critic approaches (3)

- If the expected, estimated action value will be at least as large as the *true* action value w.r.t. $\boldsymbol{\theta}^*$

$$\mathbb{E}\left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \boldsymbol{\pi}^*(\boldsymbol{X}))\right] \geq \mathbb{E}\left[q^{\boldsymbol{\pi}}(\boldsymbol{X}, \boldsymbol{\pi}^*(\boldsymbol{X}))\right], \tag{13.11}$$

  then (13.9) and (13.10) imply

$$\mathbb{E}\left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}, \tilde{\boldsymbol{\pi}}(\boldsymbol{X}))\right] \geq \mathbb{E}\left[q^{\boldsymbol{\pi}}(\boldsymbol{X}, \tilde{\boldsymbol{\pi}}(\boldsymbol{X}))\right] \tag{13.12}$$

  with a sufficiently small $\alpha < \min\{\epsilon_1, \epsilon_2\}$.
- Hence, the maximization bias is also present in actor-critic updates.
- It can add up over several estimation updates and, therefore, may lead to suboptimal policy updates.
- A proof for unnormalized gradients can be also found in S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, 2018.

Overestimation example for DDPG

(a) Hopper-v1 (b) Walker2d-v1

Fig. 13.3: Comparison of true and estimated values averaged over 10000 states in two robotic examples from OpenAI Gym. Estimated values originate from the approximate DDPG critic while the true values are based on the average discounted return over 1000 episodes following the current policy, starting from states sampled from the replay buffer (source: S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, 2018.

Increased variance due to accumulating TD errors

- Using function approximation, the Bellman equation is never exactly satisfied leaving room for some amount of residual TD-error $\tilde{\delta}(\boldsymbol{x}, \boldsymbol{u})$:

$$\hat{q}_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{u}) = r + \gamma \mathbb{E}_{\pi} \left[\hat{q}_{\boldsymbol{w}}(\boldsymbol{X}', \boldsymbol{U}') | \boldsymbol{X}' = \boldsymbol{x}', \boldsymbol{U}' = \boldsymbol{u}' \right] - \tilde{\delta}(\boldsymbol{x}, \boldsymbol{u}). \tag{13.13}$$

- Although this error might be considered small per update step, it may accumulate over future steps if biased:

$$\hat{q}_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{u}) = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k \left(R_k - \tilde{\delta}_k(\boldsymbol{X}, \boldsymbol{U}) \right) \middle| \boldsymbol{X} = \boldsymbol{x}, \boldsymbol{U} = \boldsymbol{u} \right]. \tag{13.14}$$

- Observation: the variance of $\hat{q}$ will be proportional to the variance of future reward and residual TD-errors.
- If γ is large, the estimation variance might increase significantly.
- Mini-batch sampling will contribute to this variance issue.

TD3 extensions and modifications (1)

In order to reduce both the maximization bias and the learning variance, TD3 introduces mainly three measures on top of the DDPG algorithm. Hence, TD3 is a direct successor of DDPG.

Measure #1: clipped double Q-learning for actor-critic

- Following double Q-learning, a pair of critics $\{\hat{q}_{\boldsymbol{w}_1}, \hat{q}_{\boldsymbol{w}_2}\}$ is introduced.
- In contrast, the clipped target (with target networks $\{\boldsymbol{w}_1^-, \boldsymbol{w}_2^-\}$)

$$y = r + \gamma \min_{i=1,2} \hat{q}_{\boldsymbol{w}_i^-}(\boldsymbol{x}', \boldsymbol{u}') \tag{13.15}$$

  provides an upper-bound on the estimated action value.
- May introduce some underestimation, which is considered less critical than overestimation, since the value of underestimated actions will not be explicitly propagated through the policy update.
- The $\min$ operator will also (indirectly) favor actions leading to values with estimation errors of lower variance.

TD3 extensions and modifications (2)

Measure #2: target policy smoothing regularization

- Background: deterministic policies $\boldsymbol{\mu}$ tend to overfit to narrow peaks in the action-value estimate.
- Counteraction: fit the action value of a small area around the target action (i.e., smoothing $\hat{q}$ in the action space):

$$y = r + \gamma \hat{q}_{\boldsymbol{w}^-}(\boldsymbol{x}', \boldsymbol{\mu}_{\boldsymbol{\theta}^-}(\boldsymbol{x}') + \boldsymbol{\epsilon}). \tag{13.16}$$

- Here, $\boldsymbol{\epsilon} \sim \operatorname{clip}\left(\mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}), -\boldsymbol{c}, \boldsymbol{c}\right)$ is a mean-free, Gaussian noise with covariance $\boldsymbol{\Sigma}$, which is clipped at $\pm\boldsymbol{c}$ while $\boldsymbol{\theta}^-$ are the policy target network parameters.
- To satisfy possible action constraints (denoted by upper and lower box constraints $\{\underline{\boldsymbol{u}}, \overline{\boldsymbol{u}}\}$), we add an additional clipping:

$$\boldsymbol{u}' = \operatorname{clip}\left(\boldsymbol{\mu}_{\boldsymbol{\theta}^-}(\boldsymbol{x}') + \boldsymbol{\epsilon}, \underline{\boldsymbol{u}}, \overline{\boldsymbol{u}}\right). \tag{13.17}$$

- This modified action is then used for the target calculation (13.15).

TD3 extensions and modifications (3)

Measure #3: delayed policy updates

- Similar to DDPG, TD3 uses policy target networks $\boldsymbol{\theta}^-$ and (two) critic target networks $\{\boldsymbol{w}_1^-, \boldsymbol{w}_2^-\}$ in order to provide (rather) fixed Q-learning targets trying to stabilize the learning of $\hat{q}$.
- The target networks are also continuously updated using

$$\boldsymbol{w}_i^- \leftarrow (1-\tau)\boldsymbol{w}_i^- + \tau \boldsymbol{w}_i, \quad \boldsymbol{\theta}^- \leftarrow (1-\tau)\boldsymbol{\theta}^- + \tau \boldsymbol{\theta}.$$

- However, each policy update will inherently change the (true) Q-learning target directly adding variance to the learning process (cf. Fig. 13.4 on next slide).
- Therefore, it is argued that a policy update should not follow after each Q-learning update such that the critic can adapt properly to the previous policy update.
- The original TD3 implementation suggests a policy update every second Q-learning update, however, we can consider this update rate a hyperparameter.

TD3 extensions and modifications (4)

(a) Fixed Policy (b) Learned Policy

Fig. 13.4: Average estimated action value of a randomly selected state on Hopper-v1 environment from OpenAI Gym (source: S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, 2018.

input: diff. deterministic policy function $\boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta})$ and action-value function $\hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})$
parameter: step sizes and filter constant $\{\alpha_w, \alpha_\theta, \tau\} \in \{\mathbb{R} | 0 < \alpha, \tau < 1\}$, policy update rate $k_w \in \{\mathbb{N} | 1 \leq k_w\}$, target noise $\boldsymbol{\Sigma} \in \mathbb{R}^{m \times m}$ and $\boldsymbol{c} \in \mathbb{R}^m$
init: weights $\{\boldsymbol{w}_1 = \boldsymbol{w}_1^-, \boldsymbol{w}_2 = \boldsymbol{w}_2^-\} \in \mathbb{R}^\zeta$, $\boldsymbol{\theta} = \boldsymbol{\theta}^- \in \mathbb{R}^d$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$
for $j = 1, 2, \ldots,$ *episodes* **do**
  initialize $\boldsymbol{x}_0$;
  for $k = 0, 1, \ldots, T-1$ *time steps* **do**
    $\boldsymbol{u}_k \leftarrow$ apply from $\boldsymbol{\mu}(\boldsymbol{x}_k, \boldsymbol{\theta})$ w/wo noise or from behavior policy;
    observe $\boldsymbol{x}_{k+1}$ and r_{k+1};
    store tuple $\langle \boldsymbol{x}_k, \boldsymbol{u}_k, r_{k+1}, \boldsymbol{x}_{k+1} \rangle$ in $\boldsymbol{\mathcal{D}}$;
    sample mini-batch $\boldsymbol{\mathcal{D}}_b$ from $\boldsymbol{\mathcal{D}}$ (after initial memory warmup);
    for $i = 1, \ldots, b$ *samples* **do** calculate Q-targets
      if $\boldsymbol{x}_{i+1}$ *is terminal* **then** $y_i = r_{i+1}$;
      else
        $\boldsymbol{u}' = \operatorname{clip}\left(\boldsymbol{\mu}_{\boldsymbol{\theta}^-}(\boldsymbol{x}_{i+1}) + \operatorname{clip}\left(\mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma}), -\boldsymbol{c}, \boldsymbol{c}\right), \underline{\boldsymbol{u}}, \overline{\boldsymbol{u}}\right)$;
        $y_i = r_{i+1} + \gamma \min_{l=1,2} \hat{q}(\boldsymbol{x}_{i+1}, \boldsymbol{u}', \boldsymbol{w}_l^-)$;
    fit $\boldsymbol{w}_l$ on loss $\mathcal{L}(\boldsymbol{w}_l) = [y - \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}_l)]^2_{\boldsymbol{\mathcal{D}}_b}$ with step size α_w $\forall\, l$;
    if $k \mod k_w = 0$ **then**
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta [\nabla_{\boldsymbol{\theta}} \boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}_1)|_{\boldsymbol{u} = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x})}]_{\boldsymbol{\mathcal{D}}_b}$;
      $\boldsymbol{w}_l^- \leftarrow (1-\tau)\boldsymbol{w}_l^- + \tau \boldsymbol{w}_l,\ \boldsymbol{\theta}^- \leftarrow (1-\tau)\boldsymbol{\theta}^- + \tau \boldsymbol{\theta}$;

Algo. 13.3: Twin delayed deep deterministic policy gradient (TD3)

Summary: what you've learned today

- The deep deterministic policy gradient (DDPG) approach 'transfers' many deep Q-network (DQN) ideas to continuous action spaces.
- It mainly combines DQN $+$ deterministic policy gradients $+$ policy and value target networks (plus additional minor tweaks).
- However, the DDPG actor-critic suffers from value overestimation and high variance during learning. Hence, sampled policy gradients might not be optimal (pointing towards overrated action values).
- Twin delayed DDPG (TD3) adds clipped double Q-learning, delayed policy updates and target policy smoothing to counteract these issues.

Table of contents

14 Further contemporary RL algorithms
- Trust region policy optimization (TRPO)
- Proximal policy optimization (PPO)

Reinterpreting the stochastic policy gradient (1)

- In the following we will focus on stochastic policies $\pi(\boldsymbol{u}|\boldsymbol{x})$.
- First, we rewrite the performance metric (12.7) to obtain

$$J_\pi = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_k \right]. \tag{14.1}$$

- Using the advantage $a_\pi(\boldsymbol{x}, \boldsymbol{u}) = q_\pi(\boldsymbol{x}, \boldsymbol{u}) - v_\pi(\boldsymbol{x})$ we can calculate the performance of an updated policy $\boldsymbol{\pi} \to \tilde{\boldsymbol{\pi}}$[1]:

$$J_{\tilde{\pi}} = J_\pi + \int_{\mathcal{X}} p^{\tilde{\pi}}(\boldsymbol{x}) \int_{\mathcal{U}} \tilde{\pi}(\boldsymbol{u}|\boldsymbol{x}) a_\pi(\boldsymbol{x}, \boldsymbol{u}). \tag{14.2}$$

- While for finite MDPs, the policy improvement theorem guaranteed $J_{\tilde{\pi}} \geq J_\pi$ for each policy update, there might be some states where $\int_{\mathcal{U}} \tilde{\pi}(\boldsymbol{u}|\boldsymbol{x}) a_\pi < 0$ for continuous MDPs using function approximation.

[1] proof from: S. Kakade and J. Langford, *Approximately optimal approximate reinforcement learning*, ICML, vol. 2, pp 267-274, 2002

Reinterpreting the stochastic policy gradient (2)

- For easier calculation, we introduce a local approximation to (14.2)

$$\mathcal{L}_\pi(\tilde{\pi}) = J_\pi + \int_{\mathcal{X}} p^\pi(\boldsymbol{x}) \int_{\mathcal{U}} \tilde{\pi}(\boldsymbol{u}|\boldsymbol{x}) a_\pi(\boldsymbol{x}, \boldsymbol{u}) \tag{14.3}$$

  where $p^\pi(\boldsymbol{x})$ is used instead of $p^{\tilde{\pi}}(\boldsymbol{x})$, i.e., neglecting the state distribution change due to a policy update.
- For any parametrized and differentiable policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})$, it can be shown that

$$\begin{aligned} \mathcal{L}(\pi_{\boldsymbol{\theta}_0}) &= J(\pi_{\boldsymbol{\theta}_0}), \\ \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\pi_{\boldsymbol{\theta}_0}}(\pi_{\boldsymbol{\theta}})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} &= \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_0} \end{aligned} \tag{14.4}$$

  for any initial parameter set $\boldsymbol{\theta}_0$.
- For a sufficiently small step size, improving $\mathcal{L}_{\pi_{\boldsymbol{\theta}_0}}$ will also improve J.

However, we do not know how much the actual stochastic policy will change while moving through the parameter space. Hence, we do not have a good decision basis to choose the policy gradient step size.

Adding a trust region constraint (1)

- From the previous discussion it can be concluded that we want a metric describing how much a policy is changed in the action space when updating the policy in the parameter space.
- Against this background, we make use of the Kullback-Leibler divergence (also called relative entropy)

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log \left(\frac{p(x)}{q(x)} \right) \, \mathrm{d}x \tag{14.5}$$

  defined for continuous distributions P and Q with their probability densities p and q.
- Example: for two multivariate Gaussian distributions of equal dimensions d, with means $\boldsymbol{\mu}_0, \boldsymbol{\mu}_1$ and with (non-singular) covariance matrix $\boldsymbol{\Sigma}_0, \boldsymbol{\Sigma}_1$ we receive

$$D_{\mathrm{KL}}\left(\mathcal{N}_0 \parallel \mathcal{N}_1\right) = \frac{1}{2} \Big(\mathrm{tr}\left(\boldsymbol{\Sigma}_1^{-1} \boldsymbol{\Sigma}_0\right) + \left(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0\right)^{\mathsf{T}} \boldsymbol{\Sigma}_1^{-1} \left(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0\right) - d + \ln \left(\frac{\det \boldsymbol{\Sigma}_1}{\det \boldsymbol{\Sigma}_0} \right) \Big) .$$

Adding a trust region constraint (2)

- The trust region policy optimization (TRPO) updates the policy parameters while constraining the KL divergence between the new and the old policy distribution:

$$\begin{aligned} &\max_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}), \\ \text{s.t.} \quad &\overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k, \boldsymbol{\theta}) \leq \kappa \end{aligned} \tag{14.6}$$

  with

$$\overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k, \boldsymbol{\theta}) = \overline{D}_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}, \pi_{\boldsymbol{\theta}}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}(\cdot|\boldsymbol{X}) \parallel \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{X})) \right].$$

- Hence, we want to limit the average KL divergence w.r.t. the states visited by the old policy.
- The constraint κ is a TRPO hyperparameter (typically $\kappa << 1$).
- Although (14.6) does not provide any formal convergence guarantee, we at least have a link between changes in the parameter and policy distribution space. Therefore, we can use this tool to prevent erratic policy changes.

Smooth policy updates via TRPO

Fig. 14.1: Simplified representation of the policy evolution for a scalar action given some fixed state. Left: TRPO-style updates finding the optimal action with increasing probability. Right: Unmonitored policy distributions not converging towards an optimal policy ('policy chattering').

Sample-based objective and constraint estimation (1)

- To actually solve (14.6) we will make use of samplings from Monte Carlo rollouts.
- Expanding the objective yields

$$\max_{\boldsymbol{\theta}} \mathcal{L}_{\boldsymbol{\theta}_k}(\boldsymbol{\theta}) = \max_{\boldsymbol{\theta}} J_{\pi_k} + \int_{\mathcal{X}} p^{\pi_k}(\boldsymbol{x}) \int_{\mathcal{U}} \pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x}) a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}). \tag{14.7}$$

- The first term J_{π_k} can be dropped, since it is irrelevant for the optimization result (constant).
- Using samples we can approximate $\int_{\mathcal{X}} p^{\pi_k}(\boldsymbol{x}) \approx \frac{1}{1-\gamma} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}}[\boldsymbol{X}]$.
- Moreover, $\int_{\mathcal{U}} \pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x}) a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}) \approx \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}}\left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})} a_{\pi_k}(\boldsymbol{X}, \boldsymbol{U})\right]$ is also approximated applying importance sampling based on data from the old policy.
- Hence, the sampled objective is

$$\max_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}}\left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})} a_{\pi_k}(\boldsymbol{X}, \boldsymbol{U})\right]. \tag{14.8}$$

Sample-based objective and constraint estimation (2)

- Applying the previous sample-based estimation we obtain

$$\begin{aligned} \boldsymbol{\theta}_{k+1} &= \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})} a_{\pi_k}(\boldsymbol{X}, \boldsymbol{U}) \right], \\ \text{s.t.} \quad & \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}(\cdot|\boldsymbol{X}) \parallel \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{X})) \right] \leq \kappa. \end{aligned} \tag{14.9}$$

- Hence, we have a three-step procedure for each TRPO update:

1. Use Monte Carlo simulations based on the old policy to obtain data.
2. Use the data to construct (14.9).
3. Solve the constrained optimization problem to update the policy parameter vector.

Solving (14.9) is generally a nonlinear optimization problem. The original TRPO implementation uses a local objective and constraint approximation together with conjugate gradient and line search algorithms. However, many other constrained-nonlinear solvers are also applicable.

Generalized advantage estimation

- Having data $\langle \boldsymbol{x}, \boldsymbol{u}, r, \boldsymbol{x}' \rangle$ in $\boldsymbol{\mathcal{D}}$ from a Monte Carlo rollout available, an imporant problem is to estimate $a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$ in (14.9).
- A particular suggestion in the TRPO context is to use a generalized advantage estimator (GAE) [1] defined as

$$\hat{a}_k^{(\gamma,\lambda)} = \sum_{i=0}^{\infty} (\gamma\lambda)^i \delta_{k+i}. \tag{14.10}$$

- Here, $\delta_k = r_k + \gamma v(\boldsymbol{x}_{k+1}) - v(\boldsymbol{x}_k)$ is a single advantage sample.
- Hence, the GAE is the exponentially-weighted average of the discounted advantage samples with an additional weighting λ.
- Similar formulation compared to TD(λ) but the estimator's target is the advantage.
- The choice of $(\gamma\lambda)$ trade-offs the bias and variance of the estimator.

[1] cf. J. Schulmann et al., *High Dimensional Continuous Control Using Generalized Advantage Estimation*, https://arxiv.org/abs/1506.02438, 2015

TRPO summary

The TRPO's key facts are:

- The TRPO constrains policy distribution changes when updating the policy parameters (for stochastic policies and on-policy learning).
- The objective is to enable a monotonically improving learning process.
- Using trust regions, erratic policy updates should be prevented.

The TRPO's main hurdles are:

- Constructing the objective function and constraint requires Monte Carlo rollouts (time consuming, data inefficient).
- When the sampled optimization problem is set up, a nonlinear and constrained optimization step is required (no simple policy gradient, computational costly).

We will not provide any specific TRPO implementation suggestion at this point, since this is rather cumbersome. Instead we will move forward to a similar algorithm which is pursuing the same goal (prevent erratic policy changes) with a much simpler implementation.

Table of contents

14 Further contemporary RL algorithms
- Trust region policy optimization (TRPO)
- Proximal policy optimization (PPO)

Background and motivation

- The upcoming proximal policy optimization (PPO) algorithm tries to mimic the constrained TRPO problem

$$\begin{aligned} \boldsymbol{\theta}_{k+1} &= \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})} a_{\pi_k}(\boldsymbol{X}, \boldsymbol{U}) \right], \\ \text{s.t.} \quad & \mathbb{E}_{\pi_{\boldsymbol{\theta}_k}} \left[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}(\cdot|\boldsymbol{X}) \parallel \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{X})) \right] \leq \kappa. \end{aligned}$$

based on related unconstrained problems.

- Hence, the objective will be reformulated to incorporate mechanisms preventing excessively large variations of the policy distribution during a parameter update (leading to an updated policy with sufficient proximity to the old one).
- Moreover, PPO incorporates two variants which we will discuss:
  1. Clipping the surrogate objective,
  2. Adaptive tuning of a KL-associated penalty coefficient.

Clipped surrogate objective

- The first approach is based on the following clipped objective:

$$\mathbb{E}_{\pi_{\boldsymbol{\theta}_k}}\left[\min\left\{\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})}a_{\pi_k}(\boldsymbol{X},\boldsymbol{U}), \operatorname{clip}\left(\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})}, 1-\epsilon, 1+\epsilon\right)a_{\pi_k}(\boldsymbol{X},\boldsymbol{U})\right\}\right]. \tag{14.11}$$

- Above, $\epsilon < 1$ is a PPO hyperparameter serving as a regularizer.
- The first element of $\min\{\cdot\}$ is the previous TPRO objective.
- The second element of $\min\{\cdot\}$ modifies the surrogate objective by clipping the importance sampling ratio $\pi_{\boldsymbol{\theta}}/\pi_{\boldsymbol{\theta}_k}$.
- The latter should remove the incentive for moving the importance sampling ratio outside of the interval $[1-\epsilon, 1+\epsilon]$.
- The modified objective is therefore a lower bound of the unclipped TRPO objective.

Clipped surrogate objective: positive advantage

- Consider a single sample $(\boldsymbol{x}, \boldsymbol{u})$ with a positive advantage $a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$:

$$\max_{\boldsymbol{\theta}} \min \left\{ \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})} a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}), \operatorname{clip}\left(\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})}, 1-\epsilon, 1+\epsilon \right) a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}) \right\}.$$

- Because the advantage is positive, the objective will increase if the action becomes more likely, i.e., if $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})$ increases.
- If $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x}) > (1+\epsilon)\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})$ the clipping becomes active.
- Hence, the objective reduces to

$$\max_{\boldsymbol{\theta}} \min \left\{ \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})}, 1+\epsilon \right\} a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}).$$

- Due to the $\min\{\cdot\}$ operator, the entire objective is therefore limited to $(1+\epsilon)a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$.
- Interpretation: the new policy does not benefit from going very away from the old policy distribution.

Clipped surrogate objective: negative advantage

- Consider a single sample $(\boldsymbol{x}, \boldsymbol{u})$ with a negative advantage $a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$:

$$\max_{\boldsymbol{\theta}} \min \left\{ \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})} a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}), \operatorname{clip}\left(\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})}, 1-\epsilon, 1+\epsilon \right) a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}) \right\}.$$

- Because the advantage is negative, the objective will increase if the action becomes less likely, i.e., if $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})$ decreases.
- If $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x}) < (1-\epsilon)\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})$ the clipping becomes active.
- Hence, the objective reduces to

$$\max_{\boldsymbol{\theta}} \max \left\{ \frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{u}|\boldsymbol{x})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{u}|\boldsymbol{x})}, 1-\epsilon \right\} a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u}).$$

- Due to the $\max\{\cdot\}$ operator, the entire objective is limited to $(1-\epsilon)a_{\pi_k}(\boldsymbol{x}, \boldsymbol{u})$.

Adaptive KL penalty

- The second PPO variant makes use of the following KL-penalized objective

$$\mathbb{E}_{\pi_{\boldsymbol{\theta}_k}}\left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{U}|\boldsymbol{X})}{\pi_{\boldsymbol{\theta}_k}(\boldsymbol{U}|\boldsymbol{X})}a_{\pi_k}(\boldsymbol{X},\boldsymbol{U})-\beta D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_k}(\cdot|\boldsymbol{X}) \parallel \pi_{\boldsymbol{\theta}}(\cdot|\boldsymbol{X}))\right]. \tag{14.12}$$

- Transfers the KL-based constraint into a penalty for large policy distribution changes.
- The parameter β weights the penalty against the policy improvement.
- The original PPO implementation suggests an adaptive tuning of β w.r.t. the sampled average KL divergence $\overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k,\boldsymbol{\theta})$ estimated from previous experience

$$\begin{aligned}\overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k,\boldsymbol{\theta}) < \overline{D}^*_{\mathrm{KL}}: &\quad \beta \leftarrow \beta/2,\\ \overline{D}_{\mathrm{KL}}(\boldsymbol{\theta}_k,\boldsymbol{\theta}) > \overline{D}^*_{\mathrm{KL}}: &\quad \beta \leftarrow \beta\cdot 2.\end{aligned} \tag{14.13}$$

with some target value of the KL divergence $\overline{D}^*_{\mathrm{KL}}$ (additional hyperparameter).

Algo. implementation: PPO

input: diff. stochastic policy fct. $\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta})$ and value fct. $\hat{v}(\boldsymbol{x},\boldsymbol{w})$
parameter: step sizes $\{\alpha_w, \alpha_\theta\} \in \{\mathbb{R}|0 < \alpha\}$
init: weights $\boldsymbol{w} \in \mathbb{R}^\zeta$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily, memory $\boldsymbol{\mathcal{D}}$
for $j = 1, 2, \ldots,$ *(sub-)episodes* **do**
    initialize $\boldsymbol{x}_0$ (if new episode);
    collect a set of tuples $\langle \boldsymbol{x}_k, \boldsymbol{u}_k, r_{k+1}, \boldsymbol{x}_{k+1} \rangle$ by a rollout using $\pi(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta}_j)$;
    store them in $\boldsymbol{\mathcal{D}}$;
    estimate the advantage $\hat{a}_{\pi_j}(\boldsymbol{x},\boldsymbol{u})$ based on $\hat{v}(\boldsymbol{x},\boldsymbol{w}_j)$ and $\boldsymbol{\mathcal{D}}$ (e.g., GAE);
    $\boldsymbol{\theta}_{j+1} \leftarrow$ policy gradient update based on the PPO variant (14.11) or (14.12);
    $\boldsymbol{w}_{j+1} \leftarrow$ minimizing the mean-squared TD errors using $\boldsymbol{\mathcal{D}}$ (critic);
    delete entries in $\boldsymbol{\mathcal{D}}$ (due to on-policy learning);

Algo. 14.1: Proximal policy optimization (output: parameter vectors $\boldsymbol{\theta}^*$ for $\pi^*(\boldsymbol{u}|\boldsymbol{x},\boldsymbol{\theta}^*)$ and $\boldsymbol{w}^*$ for $\hat{v}^*(\boldsymbol{x},\boldsymbol{w}^*)$)

Some PPO remarks

- Clipping the surrogate objective (14.11) was reported to achieve higher performances than the KL penalty (14.12).[1]
- Like TRPO, PPO is an on-policy algorithm. Hence, the memory $\mathcal{D}$ is not a rolling replay buffer (cf. off-policy algorithms like DQN, DDPG or TD3) but a rollout buffer using one fixed policy.
- These rollouts are likely to result in an increased sample demand either using a simulator or a real experiment.

Although PPO is derived from a TRPO background pursuing monotonically increasing policy performance, its realization is based on multiple heuristics and approximations. Hence, there is no guarantee on achieving this goal and the specific performance of the PPO algorithm must be evaluated empirically given a certain application.

[1] cf. original PPO paper results by J. Schulman et al., *Proximal Policy Optimization Algorithms*, https://arxiv.org/abs/1707.06347, 2017

Exemplary performance comparison

Fig. 14.2: Learning curves for OpenAI Gym continuous control tasks. The shaded region represents half a standard deviation of the average evaluation over ten trials (source: S. Fujimoto et al., *Addressing Function Approximation Error in Actor-Critic Methods*, 2018).

Algorithmic outlook: other contemporary model-free algorithms (1)

The selection of algorithms appears endless:

- DQN variants such as
  - (Prioritized) dueling DQN
  - Noisy DQN
  - Distributional DQN
- Rainbow (combining multiple DQN extensions)
- Soft actor-critic (SAC)
- Actor critic using Kronecker-factored trust region (ACKTR)
- Asynchronous advantage actor-critic (A3C)
- ....

Remarks:

- You have already learned the basic building blocks in order to make yourself familiar with any value-/policy-based model-free RL approach.
- Use this knowledge!
- Focus on primary scientific literature for self-studying and not on unreliable sources!

Algorithmic outlook: other contemporary model-free algorithms (2)

Algorithm collections with tutorial-style documentation:

- Intel Reinforcement Learning Coach
- OpenAI Spinning Up

Algorithm collections with decent application-oriented documentation:

- RLlib (Ray)
- Stable Baselines3
- Acme
- Google Dopamine
- TF-Agents
- ...

Summary: what you've learned today

- Trust region policy optimization (TRPO) pursues monotonically increasing policy performance by limiting policy distribution changes.
- This results in a nonlinear constrained optimization problem adding computational complexity (no simple policy gradients).
- Proximal policy optimization (PPO) converts the TRPO idea into an unconstrained optimization problem by a modified objective. Likewise, the PPO's objective is to prevent erratic policy distribution changes.

Table of contents

15 Outlook and practical research insights
- Safe reinforcement learning
- Real-world implementation with fast policy inference
- Meta reinforcement learning

Recap: optimal control and constraints

Real-world systems are always subject to certain state constraints $\mathcal{X}$ and input limitations $\mathcal{U}$. Violating those can lead to safety issues.

$$v_k^* = \max_{\boldsymbol{u}_k} \sum_{i=0}^{N_\mathrm{p}} \gamma^i r_{k+i+1}(\boldsymbol{x}_{k+i}, \boldsymbol{u}_{k+i}) \,, \tag{15.1}$$

$$\text{s.t.} \qquad \boldsymbol{x}_{k+i+1} = \boldsymbol{f}(\boldsymbol{x}_{k+i}, \boldsymbol{u}_{k+i}), \quad \boldsymbol{x}_{k+i} \in \mathcal{X}, \quad \boldsymbol{u}_{k+i} \in \mathcal{U} \,.$$

Fig. 15.1: MPC scheme (source: www.wikipedia.org, by Martin Behrendt CC BY-SA 3.0)

Application examples with safety-relevant constraints

Collaborative robot control (source: www.wikipedia.org, CC BY-SA 4.0)

Autonomous car driving (source: www.wikipedia.org, CC BY-SA 4.0)

Energy system control

Medication control (source: www.wikipedia.org, CC BY-SA 4.0)

Safety levels

Fig. 15.2: Different levels of safety (derived from L. Brunke et al., *Safe Learning in Robotics: From Learning-Based Control to Safe Reinforcement Learning*, Annual Review of Control, Robotics, and Autonomous Systems, 2022)

Bird's eye view on RL concepts integrating safety

(a) Safety critic: add a critic which indicates to which extent the current data sample fits to a safe situation

(b) Safety shield: use a priori or learned model knowledge of the environment to make predictions identifying actions leading to unsafe situations

Achievable safety levels and model knowledge

Fig. 15.4: Safety and model knowledge map (derived from L. Brunke et al., *Safe Learning in Robotics: From Learning-Based Control to Safe Reinforcement Learning*, Annual Review of Control, Robotics, and Autonomous Systems, 2022)

Energy system control application

(a) Example microgrid that can be emulated in the LEA Microgrid Laboratory.

(b) Application under investigation: Three-phase grid-forming inverter disturbed by stochastic load

Reference tracking with disturbance rejection

Fig. 15.6: Simulation setting with environment modeled using OpenModelica Microgrid Gym

- Cont. state- and actionspace
- Deep deterministic policy gradient agent
- Gird-forming inverter
- Stochastic load acts as disturbance
- State per phase: $\boldsymbol{x_k} = [i_{\mathrm{f}}, v_{\mathrm{C}}]$, $v_{\mathrm{i}} = v_{\mathrm{DC}} \cdot u_k$
- $r_k = f(v_{\mathrm{C}}, v^*, i_{\mathrm{f}}) \in [1, -0.75]$
- $s_k = -1$, if limit (i_{f} or v_{C}) is exceeded

Reward design for grid-forming inverter

Fig. 15.7: Reward function 15.2 for different reference and measured voltages and currents below nominal current

- Three cases, depending on operation point

$$r = \begin{cases} \mathrm{MRE}(v_\mathrm{C}, v^*), & \text{Ⓐ} \\ \mathrm{MRE}(v_\mathrm{C}, v^*) + f(i_\mathrm{f}), & \text{Ⓑ} \\ -1, & \text{Ⓒ} \end{cases} \tag{15.2}$$

- Ⓐ $v_\mathrm{C} \leq v_\mathrm{lim} \wedge i_\mathrm{f} \leq i_\mathrm{nom}$
- Ⓑ $v_\mathrm{C} \leq v_\mathrm{lim} \wedge i_\mathrm{nom} \leq i_\mathrm{f} \leq i_\mathrm{lim}$
- Ⓒ otherwise
- Linear punishment term $f(i_\mathrm{f})$

Reference tracking with disturbance rejection using saftey shield

Fig. 15.8: Safety shield based on feasible set

- Safety shield: Ensure that action does not cause state limit violation in future system trajectories
- Such a state action pair is called feasible
- Calculation of feasible set requires a model
- Training data can be utilized to identify model
- Here, recursive least squares (RLS) is applied

Saftey shield based on feasible set - proof of concept (1)

Fig. 15.9: Accumulated unsafe events (overcurrent/-voltage) per trainingstep k

- Three different approaches
- DDPG: Agent without safety shield
- DDPG$_{SG}$: Agent with safety shield using perfect a priori knowledge
- DDPG$_{SG,RLS}$: Agent with safety shield without a priori knowledge, identifying model using RLS
- Five agents trained per approach
- Results in D. Weber et al., *Safe Reinforcement Learning-Based Control in Power Electronic Systems*, 2023

Saftey shield based on feasible set - proof of concept (2)

Fig. 15.10: Blackstart after training using $\mathrm{DDPG}_{\mathrm{SG,RLS}}$

- $\mathrm{DDPG}_{\mathrm{SG,RLS}}$ agent trained for 150000 steps
- R_{Load} changes every step based on random process
- Additional events – load steps and drifts – trigged randomly

Table of contents

15 Outlook and practical research insights
- Safe reinforcement learning
- Real-world implementation with fast policy inference
- Meta reinforcement learning

Real-time implementation aspects (1)

Fig. 15.11: DDPG implementation example (derivative work of Fig. 1.1 and wikipedia.org, CC0 1.0)

Real-time implementation aspects (2)

(a) Real-time control requirement vs. learning time

(b) Typical evolution of RL parameter weights during learning

Application example: deep Q direct torque control

Fig. 15.13: Deep Q direct torque control schematic

- The DQ-DTC is basically a DQN
- Sampling time of the plant system is $T_s = 50\,\mu s$
- DQN inference, safeguarding and system identification must fit into T_s
- Source: M. Schenke et al., *Finite-Set Direct Torque Control via Edge Computing-Assisted Safe Reinforcement Learning for a Permanent Magnet Synchronous Motor*, 2023

Fast neural network inference

Fig. 15.14: Conceptual comparison of CPU and FPGA evaluation of a neural network

- Each neuron has the same job $y_{n,l+1} = f(\boldsymbol{y}_l^\top \boldsymbol{w}_{n,l} + b_{n,l})$
- CPU must evaluate each neuron sequentially
- FPGA can evaluate each neuron at the same time
- Maximum number of parallel computations is limited

Edge reinforcement learning

Fig. 15.15: Our edge reinforcement learning pipeline

- Backward pass / learning steps are outsource to workstation
- Communication between test bench and workstation is based on TCP/IP
- Backward pass is generic and has no time constraints → low application effort

Demonstration

Table of contents

15 Outlook and practical research insights
- Safe reinforcement learning
- Real-world implementation with fast policy inference
- Meta reinforcement learning

Meta reinforcement learning - the setting (1)

(a) General problem class is similar, environments only differ in some characteristics, the agent could transfer learned behavior

(b) Solution approach: treat the environment as partially observable, distinguishing details are not directly available

Meta reinforcement learning - the setting (2)

- The agent must have some mechanism that allows adaptation to the specific environment
- This means, the distinguishing details must be extracted in some way
- Usually, they can be retrieved from a larger set of observations

Fig. 15.17: Different concepts of meta learning

Usage in electric drive control: classical agent

Fig. 15.18: Each agent must be trained individually → huge effort

Usage in electric drive control: meta agent

Fig. 15.19: One agent to control them all → effort is limited and independent of the number of controlled environments

Our setup

- Make use of context network
- Generate context $\boldsymbol{z}$ with a fix set of observations $\rightarrow \boldsymbol{z} = \text{const.}$
- Source: D. Jakobeit et al., *Meta-Reinforcement Learning-Based Current Control of Permanent Magnet Synchronous Motor Drives for a Wide Range of Power Classes*, IEEE TPEL, 2023

Fig. 15.20: A meta learning concept that we implemented successfully

Evaluation on (very) different motors

(a) Current control on a PMSM with low rated power

(b) Current control on a PMSM with high rated power

Summary

- Application of RL on technical systems comes with many challenges, e.g.,
  - Safety limits,
  - Real-time / computational constraints,
  - Varying and/or partially unknown environments.
- Real-world implementations often require more than bare RL algorithms, e.g.,
  - Integration of available a priori expert knowledge,
  - Combination with model-based control engineering tools.
- Ideal integration of data-driven RL solutions together with expert-based control engineering parts is subject to many open research question.

Table of contents

Summary of part II

Reinforcement learning in continuous state and action spaces

9 Supervised learning

10 Approximative on-policy prediction

11 Approximative value-based control

12 Stochastic policy gradient methods

13 Deterministic policy gradient methods

14 Further contemporary RL algorithms

15 Outlook and practical research insights

What was covered in the course

Fig. S-II.1: Main categories of reinforcement learning algorithms
(derived work based on D. Silver, Reinforcement learning, 2016. CC BY-NC 4.0)

Additional topics not covered in this lecture (1)

- Structured exploration: can we find a systematic way for fast and robust exploration?
  - R. Houthooft et al., "Vime: Variational information maximizing exploration", Advances in Neural Information Processing Systems, 2016
  - S. Levine, CS285 Deep Reinforcement Learning (lecture notes UC Berkeley), 2019
  - D. Silver, Reinforcement Learning (lecture notes UC London), 2015

- Imitation learning: how can we mimic the behavior of a certain baseline agent / controller / human expert?
  - A. Hussein et al., "Imitation learning: A survey of learning methods", ACM Computing Surveys (CSUR) 50.2, pp. 1-35, 2017
  - A. Attia and S. Dayan, "Global overview of imitation learning", arXiv:1801.06503, 2018

Additional topics not covered in this lecture (2)

- Multi-agent algorithms: finding solutions to distributed problems (e.g., for distributed energy systems).
  - L. Busoniu, R. Babuska and B. De Schutter. "A comprehensive survey of multiagent reinforcement learning." IEEE Transactions on Systems, Man, and Cybernetics, Part C 38.2, pp. 156-172, 2008
  - P. Hernandez-Leal, B. Kartal and M. Taylor. "Is multiagent deep reinforcement learning the answer or the question? A brief survey", Researchgate preprint, 2018
- Federated learning: finding solutions to distributed problems via multiple independent sessions, each using its own local information (addressing critical issues such as data privacy, data security, data access rights).
  - H. Zhuo et al. "Federated Deep Reinforcement Learning", arXiv:1901.08277, 2019
  - J. Qi et al. "Federated Reinforcement Learning: Techniques, Applications, and Open Challenges", arXiv:2108.11887, 2021

Learning summary

What you should have learned

- How to model decision processes using a Markov framework.
- Finding exact solutions using iterative tabular methods for discrete problem spaces.
- Finding approximate solutions for large discrete or continuous problem spaces based on function approximation.
- Application of just these techniques on a practical programming level.

Concluding remarks

- This is an introductory course to RL. We have only scratched the surface.
- Some aspects, especially within the exercises, had a control focus. In other application, specific RL solutions can look quite different.
- If you are interested in more practical RL insights in the field of electrical power systems, do not hesitate to contact us.