Combinatorial Optimization Augmentede Machine Learning

Learning By Experience

Prof. Dr. Maximilian Schiffer

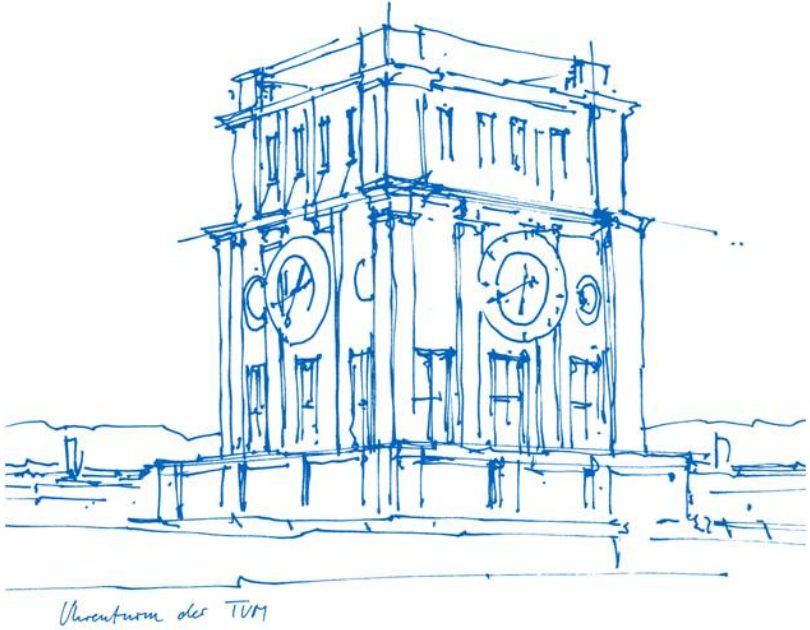
Professorship for Business Analytics and Intelligent Systems

TUM School of Management

Munich Data Science Institute

Technical University of Munich

Winter Term 2023/2024



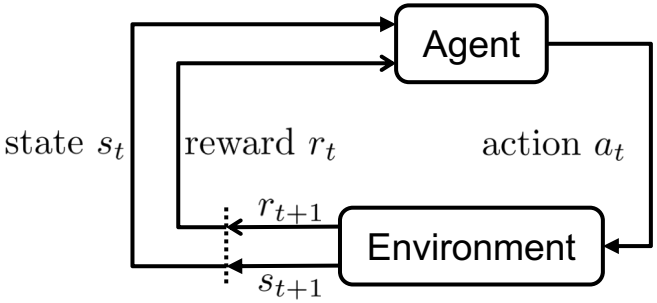
Topics for this week

Agenda

1. Markov Decision Process
2. Reinforcement Learning
3. Deep Q Networks
4. Advanced DRL Architectures

Markov Decision Process: Introduction

- In a Markov Decision Process **(MDP)** an **agent** takes **actions** in an **environment**
- The environment responds to an action with a **new state** and a **reward**



[1]

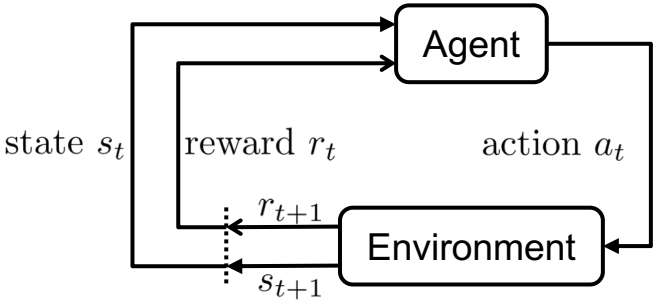
Example: Recycling robot

[2]

- A robot searches for cans in the park
- The robot has two battery levels (states): {high, low}
- The robot can choose three actions: {search, wait, recharge}

Markov Decision Process: Introduction

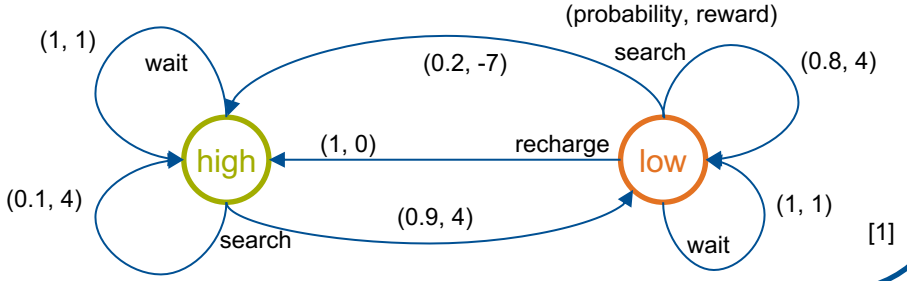
- In a Markov Decision Process **(MDP)** an **agent** takes **actions** in an **environment**
- The environment responds to an action with a **new state** and a **reward**



[1]

Example: Recycling robot

- When the robot runs out of battery → reward: -7
- When the robot searches (waits) → reward: 4 (1)
- When the robot has low (high) battery level and searches, with probability 0.8 (0.1) battery level stays low (high) and with probability 0.2 (0.9) the robot runs out of energy (into the low battery level)



[1]

In this lecture, we learn how the recycling robot finds the optimal action for each state.
We call (the optimal) action to state mapping a **policy** (which is a solution to an MDP)

Markov Decision Processes: Notation

- An **MDP** is defined by:
  - A **state** space S

Example Recycling robot

[1]

Markov Decision Processes: Notation

- An **MDP** is defined by:
  - A **state** space S
  - An **action** space $A(s),\ s \in S$

Example Recycling robot

Actions = {wait, search, recharge}

The robot chooses the action to execute in the next steps

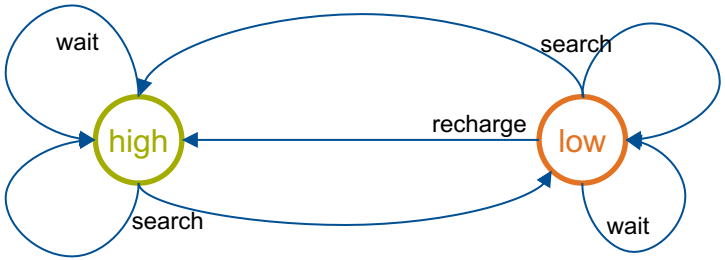
[1]

Markov Decision Processes: Notation

- An **MDP** is defined by:
  - A **state** space S
  - An **action** space $A(s),\ s \in S$
  - A **transition** function
    $T(s, a, s')$ with $s, s' \in S, a \in A(s)$

Example Recycling robot

The **transition function** describes the **effect of an action in each state**



[1]

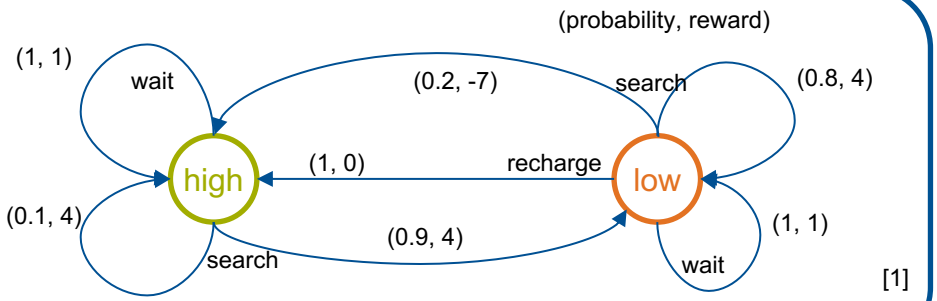
Markov Decision Processes: Notation

- An **MDP** is defined by:
  - A **state** space S
  - An **action** space $A(s),\ s \in S$
  - A **transition** function
    $T(s, a, s')$ with $s, s' \in S, a \in A(s)$
    - Probability that a at s leads to s', $P(s'|s, a)$
  - A **reward** function
    $R(s, a, s')$ with $s, s' \in S, a \in A(s)$

Example Recycling robot

P(high|low, search) = 0.2
Probability to make a certain transition

The **reward** the robot gains for every transition



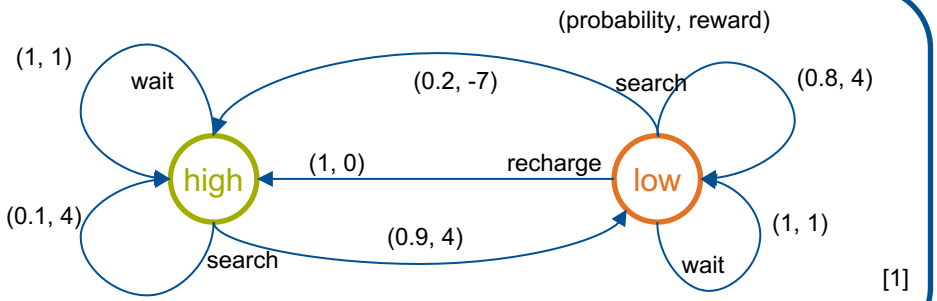
[1]

Markov Decision Processes: Notation

- An **MDP** is defined by:
  - A **state** space S
  - An **action** space $A(s),\ s \in S$
  - A **transition** function
    $T(s, a, s')$ with $s, s' \in S, a \in A(s)$
    - Probability that a at s leads to s', $P(s'|s, a)$
  - A **reward** function
    $R(s, a, s')$ with $s, s' \in S, a \in A(s)$
  - A **start state** $s \in S$

Example Recycling robot

The **start state** is the state where the MDP begins. In our case the robot starts with a high battery level



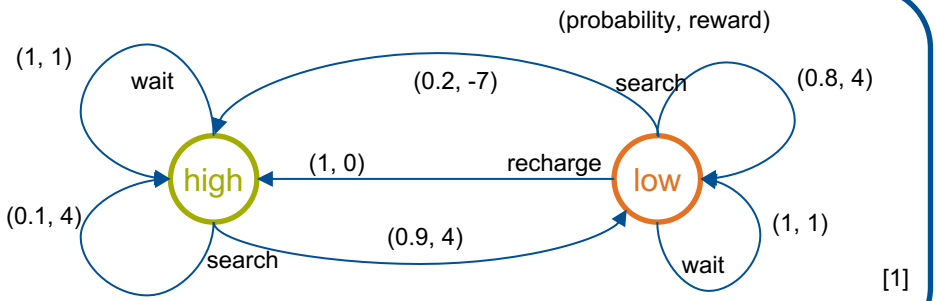
[1]

Markov Decision Processes: Notation

- An **MDP** is defined by:
  - A **state** space S
  - An **action** space $A(s),\ s \in S$
  - A **transition** function
    $T(s, a, s')$ with $s, s' \in S, a \in A(s)$
    - Probability that a at s leads to s', $P(s'|s, a)$
  - A **reward** function
    $R(s, a, s')$ with $s, s' \in S, a \in A(s)$
  - A **start state** $s \in S$
  - A **terminal state** $s \in S$ (a terminal state can be infinitely far in the future)

Example Recycling robot

The Markov Decision Process ends when the agent reaches a **terminal state**. In our example there **is no terminal state**



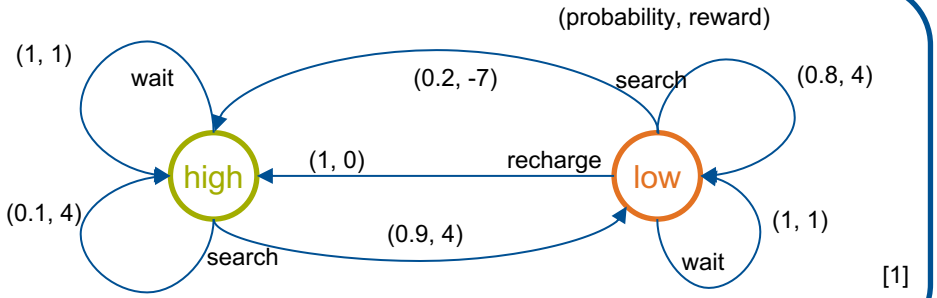
[1]

Markov Decision Processes: Notation

- An **MDP** is defined by:
  - A **state** space S
  - An **action** space $A(s),\ s \in S$
  - A **transition** function
    $T(s, a, s')$ with $s, s' \in S, a \in A(s)$
    - Probability that a at s leads to s', $P(s'|s, a)$
  - A **reward** function
    $R(s, a, s')$ with $s, s' \in S, a \in A(s)$
  - A **start state** $s \in S$
  - A **terminal state** $s \in S$ (a terminal state can be infinitely far in the future)
- **Policy**: mapping from states to actions $(\pi : S \to A)$. The policy defines the action that the agent chooses in each state

Example Recycling robot

An example **policy**:
high → search
low → recharge



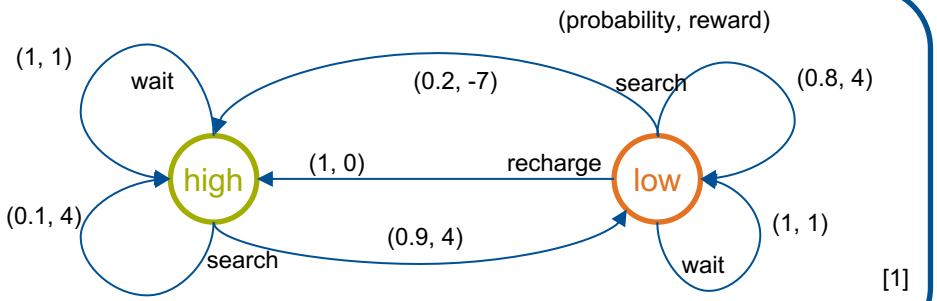
[1]

Markov Decision Processes: Notation

- An **MDP** is defined by:
  - A **state** space S
  - An **action** space $A(s),\ s \in S$
  - A **transition** function
    $T(s, a, s')$ with $s, s' \in S, a \in A(s)$
    - Probability that a at s leads to s', $P(s'|s, a)$
  - A **reward** function
    $R(s, a, s')$ with $s, s' \in S, a \in A(s)$
  - A **start state** $s \in S$
  - A **terminal state** $s \in S$ (a terminal state can be infinitely far in the future)
- **Policy**: mapping from states to actions $(\pi : S \to A)$. The policy defines the action that the agent chooses in each state
- **Objective**: find a policy $\pi : S \to A$ that maximizes the expected sum of rewards

Example Recycling robot

The **objective** is to collect as many cans as possible (maximize reward)



[1]

Markov Decision Processes: Episodic and continuous tasks

Episodic task

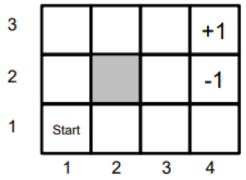
- natural notion of a **final time step**
- environment interaction **breaks naturally into subsequences, called episodes**
- Can either have…
  - … terminal states (indefinite horizon)
  - … „pre-defined“ number of time-steps (finite horizon)

Continuous task

- **No terminal state or number of time steps**, i.e., process can continue „forever“ (infinite horizon)

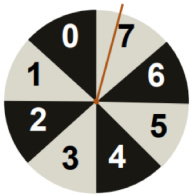
Examples

Terminal states



Ex. Sheet 2, exercise 4

Number of time steps

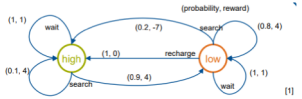


Ex. Sheet 2, exercise 3

Continuous

[2]

Recycling robot (see lecture on MDPs, pg. 4)

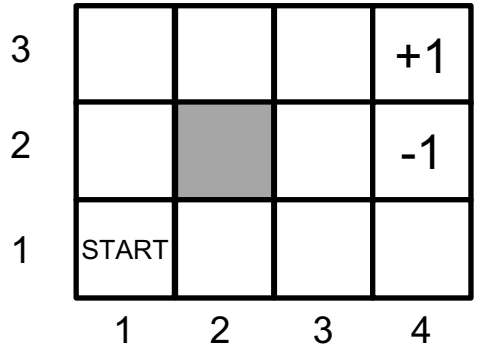


[1]

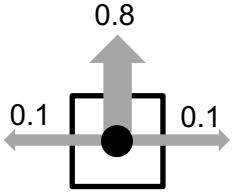
Markov Decision Process: Gridworld

Gridworld: An environment consisting of 4 x 3 cells forming a grid

- A robot starts in the START cell and wants to maximize its reward
- The robot can perform the actions {*left, right, up, down*}
- With probability 0.8 the robot performs the correct action. With probability 0.1 (0.1) the agent slips at right angle in the left (right) direction of the intended direction
- State (4,3) with reward +1, and state (4,2) with reward -1 are terminal states. All other states have a reward of 0.
- When the robot moves into the wall (grey cell) or out of gridworld, the robot stays in the original cell



Gridworld



Probability distribution of moving direction when choosing action *up*

Markov Decision Process: Policy

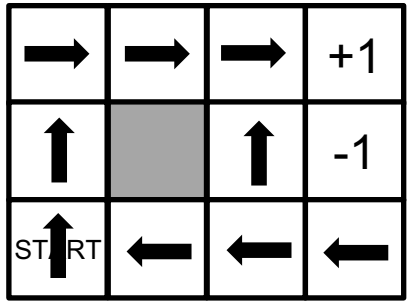
Policy = A mapping from states to actions $(\pi : S \rightarrow A)$. The policy defines the action that the agent chooses in each state

Difference between Policy and Expectimax

- The objective in **deterministic single-agent search problems** is to find an optimal **plan** (sequence of actions, from start to a goal).
  - Recall: Expectimax (cf. Lecture 2) would compute only the action for a single state but not an entire policy (Action for every state)
- For **MDPs**, the objective is to find an optimal **policy:** $\pi^* : S \rightarrow A$
  - A policy π specifies the action for each state
  - An optimal policy maximizes the expected utility

Example for policy

Gridworld: Each cell is a state, and a robot wants to maximize its reward, starting in cell START.



Policy:

State(4,1): left
State(3,1): left
State(2,1): left
State(1,1) : up
State(1,2): up
State(3,2): up
State(1,3): right
State(2,3): right
State(3,3): right

Markov Decision Process: Policy

Formal optimization objective

- The agent aims to maximizes its **expected total reward over all time steps**, given an initial state S_0. The set of all possible policies is Π. Then,

$$\pi_0^*(s_0) = \arg\max_{\pi \in \Pi} \mathbb{E}_{s_t, a_t \sim \pi}\left[\sum_{t_0}^{T-1} R_t(s_t, a_t, s_{t+1} | s_0)\right] = \arg\max_{\pi \in \Pi} \mathbb{E}_{\pi}\left[\sum_{t_0}^{T-1} R_t(s_t, a_t, s_{t+1} | s_0)\right]$$

State Action Reward function Policy Sum over all timesteps

- A policy can also be **stochastic** with conditional probability distributions for each state. The sum of all probabilities for actions in a certain state sum to 1.
  - For example when the underlying MDP is unknown
  - For example a stochastic policy fits the game: Rock, Paper, Scissors

Agenda

1. Markov Decision Process
2. Utilities
3. Bellman equation
4. Value Iteration
5. Policy Iteration

Utilities

1. Discounting
2. Preferences
3. Q-Value and V-Value

We learned about Markov Processes:

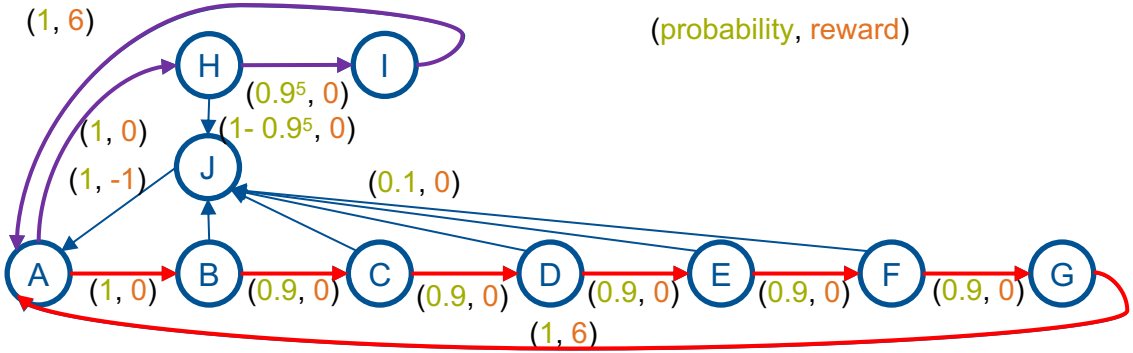
- A Markov Process must fulfill the **Markov property**
- A Markov Process consists of **States, Actions, Transitions, Rewards, Start State, (Terminal State)**
- **Policy**: outputs the action for every state ($S \rightarrow A$)

In the next section, we learn how to specify the utility for an agent being in a certain state

Utilities: Discounting

- The utility of an agent for being at a certain state depends on the reward that it can reach from this state in the future. Here, discounting allows to include future rewards into the present evaluation.

- We consider the following Markov Decision Process:
- Do we prefer the purple path or the red path?

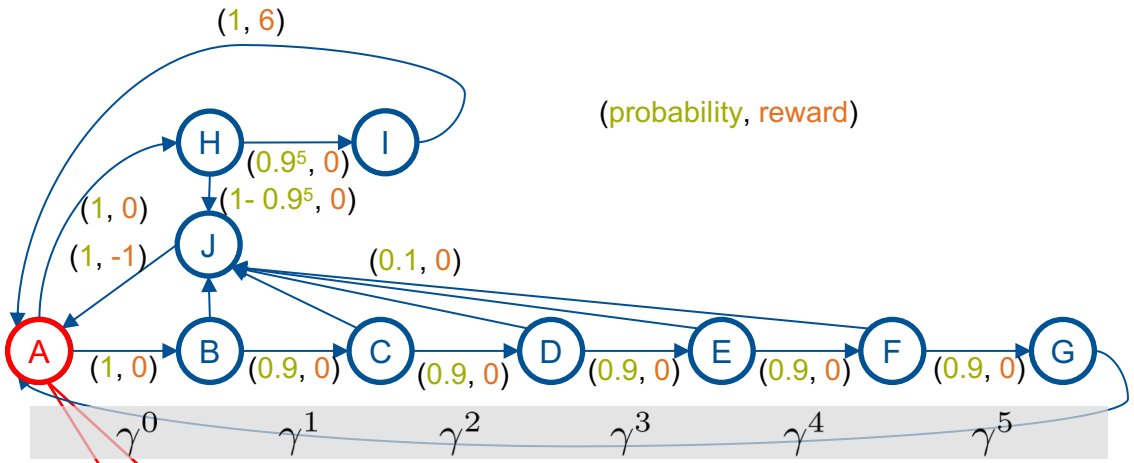


- The agent should have **preferences for earlier (after less actions) rewards**. Because:
  - The agent may not know how many time steps are left
  - The agent prefers fast feedback for the policy learning
  - The further the agent plans into the future, the more uncertainty needs to be considered
  - If the game lasts forever, we would reach infinite reward
- It is reasonable to **maximize the sum of rewards**
- It is also reasonable to **prefer rewards now to rewards later**

Utilities: Discounting

- We want to prefer earlier rewards over later rewards

- **One solution:** values of rewards decay exponentially
- We introduce a discount factor
  $0 \leq \gamma \leq 1$
- For each time step considered further into the future, we multiply the discount factor once
- A reward n steps away is discounted by γ^n
- Due to the discounting of future rewards the **agent prefers earlier rewards over late rewards**



(1, 6) → (1, 0.8^6 x 6 = 1.57)

We assume starting in state A and having a discount factor of 0.8

→ Utility = Sum of discounted rewards

Utilities: Infinite problems

- If a game lasts forever there would be **infinite reward** to collect.
- There exist three different approaches to avoid infinite rewards:

Discounting

- **Discounting**: use $0 < \gamma < 1$

Sum over all timesteps

Constant maximum reward

$$U([r_0, \ldots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma}$$

Utility function

Reward at time t

Discount factor

  - Smaller γ implies a shorter “horizon”

Utilities: Infinite problems

- If a game lasts forever there would be **infinite reward** for the agent to collect.
- There exist three different approaches to avoid infinite rewards:

Discounting

- **Discounting**: use $0 < \gamma < 1$

Sum over all timesteps

Constant maximum reward

Geometric series

$$U([r_0, \ldots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma}$$

Utility function

Reward at time t

Discount factor

  - Smaller γ implies a shorter "horizon"

$$U = R_{max} + \gamma R_{max} + \gamma^2 R_{max} + \cdots + \gamma^n R_{max}$$

$$\gamma U = \gamma R_{max} + \gamma^2 R_{max} + \cdots + \gamma^n R_{max} + \gamma^{n+1} R_{max} \quad \Big| \cdot \left(\frac{1}{\gamma} - 1\right)$$

$$U - \gamma U = R_{max} - R_{max}\gamma^{n+1}$$

$$U(1-\gamma) = R_{max}(1-\gamma^{n+1})$$

$$U = R_{max}\left(\frac{1-\gamma^{n+1}}{1-\gamma}\right)$$

Utilities: Infinite problems

- If a game lasts forever there would be **infinite reward** for the agent to collect.
- There exist three different approaches to avoid infinite rewards:

Discounting

- **Discounting**: use $0 < \gamma < 1$

Sum over all timesteps

Constant maximum reward

Geometric series

$$U([r_0, \ldots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma}$$

Utility function

Reward at time t

Discount factor

  - Smaller γ implies a shorter "horizon"

Finite horizon

- (similar to depth-limited search)
  - **Terminate episodes** after T steps (e.g., life)
  - Gives nonstationary policies (π depends on time left)

Absorbing state

- **Guarantee** that for every policy, a **terminal** state will eventually be reached

Utilities

- The utility specifies the **value for being in a certain state** of the environment **considering future rewards**
- The utility is the discounted sum of **realized** rewards $U = \sum_{t=0}^{\infty} \gamma^t r_t$
- In the future, (lecture: Policy gradient onwards) we denote the **discounted sum of realized rewards** with G

V - Value

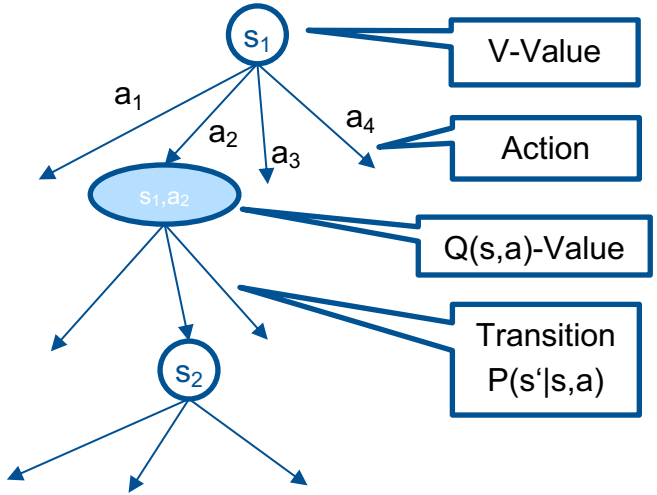
- The expected utility for being at state s

$V^*(s) =$ **expected** utility for being in state s and acting optimally thereafter: $\mathbb{E}_{\pi^*, s' \sim P}[U]$

Q - Value

- The expected utility for being at state s and taking action a

$Q^*(s, a_1) =$ **expected** utility when taking action a_1 in state s and acting optimally thereafter: $\mathbb{E}_{\pi^*, s' \sim P}[U | a_1]$



In this lecture, we focus mostly on the V-Value, whereas we will revisit the Q-Value when we talk about Reinforcement Learning

Utilities: Gridworld

V-Values

0.64	0.74	0.85	+1
0.57		0.57	-1
0.49	0.43	0.48	0.28

Q-Values

0.59 0.57 0.64 0.53	0.67 0.60 0.74 0.67	0.77 0.66 0.85 0.57	+1
0.57 0.51 0.51 0.46		0.57 0.53 -0.60 0.30	-1
0.49 0.45 0.41 0.44	0.40 0.43 0.42 0.40	0.48 0.40 0.29 0.41	-0.65 0.28 0.13 0.27

[3]

Noise = 0.2 | Discount factor = 0.9 | Living reward = 0

Agenda

1. Makov Decision Process
2. Utilities
3. Bellman equation
4. Value Iteration
5. Policy Iteration

We learned about Utilities:

- An agent has preferences for higher utilities
- Utilities are discounted over the future
- **V-Value** | $V^{\pi}(s)$: expected discounted return starting from state s and following policy π
- **Q-Value** | $Q^{\pi}(s,a)$: expected discounted return starting from state s, taking action a, and following policy π

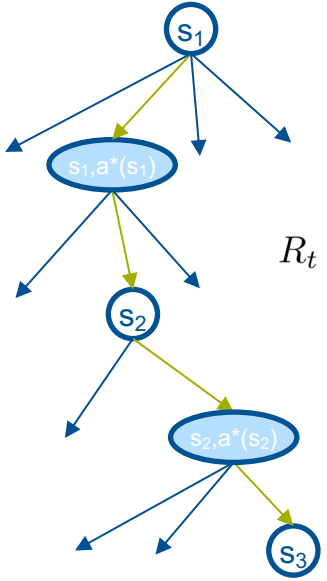
In the next section, we define utilities in a Markov Decision Process using Bellman equations

The Bellman equations

Intuition

- The **Bellman equations** calculate the **utility** (considering future rewards) for each state and each state-action pair, when following a given policy.
- The Bellman equations follow the idea: “Take the correct first action and keep being optimal afterwards”

- The Bellman equations define the “optimal utility” via recursive expectimax, considering a simple **one-step lookahead relationship amongst optimal utility values**
  - Expected utility under optimal action
  - Average sum of (discounted) rewards



When we have the optimal action for each state s, we can calculate the reward for being in state s_1 with:

One-step lookahead relationship

$$R_t = \overbrace{r_{t+1} + \gamma(R_{t+1})}$$
$$= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \ldots + \gamma^{T-2} r_T)$$
$$= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \gamma^{T-1} r_T$$

Reward at time step Discount factor

Optimal path: assuming we always continue with the optimal path

The Bellman equations

- The Bellmann Equation calculates the **utility for being at state s** and acting optimal in each following step

$$V^*(s) = \max_a Q^*(s,a)$$

- The optimal Q-value is the **utility when being at state s and taking action a, acting optimally afterwards**. It results from the transition probability for being at state s and taking action a multiplied with discounted rewards for subsequent optimal decisions

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Transition function

Reward function

Discount factor

Utility for being at this state assuming optimal subsequent actions

- The optimal V-value is the **utility when being at state s**. It results from the transition probability for being at state s and taking the optimal action a multiplied with discounted rewards for subsequent optimal decisions

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Transition function

Maximum over all actions

Reward function

Discount factor

Utility for being at this state assuming optimal subsequent actions

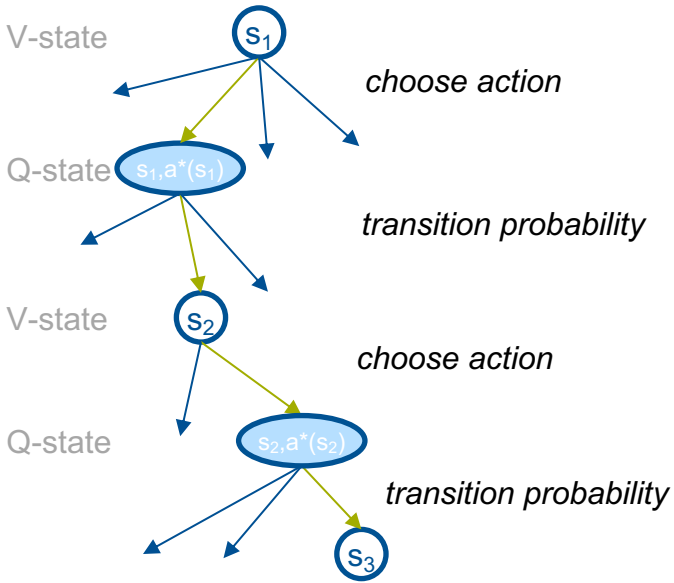
Recap: Markov Decision Processes

Markov decision processes

- State space S
- Action space $A(s), \ s \in S$
- Transition function $T(s, a, s'), \ s, s' \in S, \ a \in A(s)$
- Reward function $R(s, a, s'), \ s, s' \in S, \ a \in A(s)$
- Start state s_0

Quantities:

- **Policy** = map of states to actions
- **Utility** = sum of discounted rewards
- **V-Values** = expected future utility from a state (max node)
- **Q-Values** = expected future utility from a Q-state (chance node)



Next, we learn how to find the optimal **policy** for a Markov Decision Process

Agenda

1. Markov Decision Process
2. Utilities
3. Bellman equation
4. Value Iteration
5. Policy Iteration

We can model the utilities in a Markov Decision Process with the **Bellman equations**.

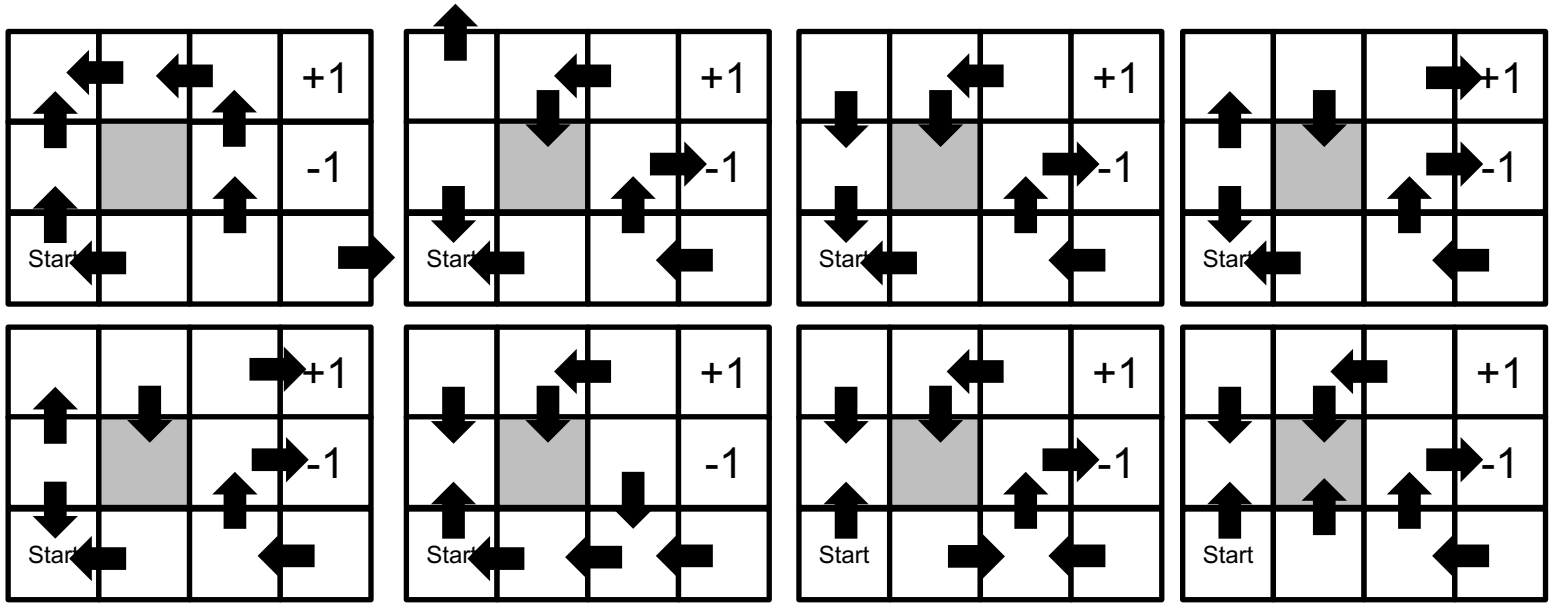
In the next section, we calculate the utilites in a Markov Decision Process

Value + Policy

1. **Value Iteration**
   1. Find the optimal value function
   2. Policy Extraction
2. **Policy Iteration**
   1. Policy Evaluation
   2. Policy Improvement

Value and Policy Iteration: Motivation

For an MDP with $|S|$ states and $|A|$ actions, $|A|^{|S|}$ different policies exist.



How can we **find** the best policy?

Value and Policy Iteration: Motivation

How can we **find** the best policy?

Intuition 1

We formalize an iterative procedure that converges to the optimal fixed point.

Banach fixed-point theorem : with two utilities U_i and U_i' and the Bellman update $B(U)$

$$U_{i+1} \leftarrow B(U_i)$$

we can show that the Bellman update is a contraction

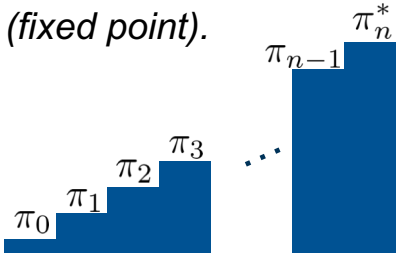
$$||B(U_i) - B(U_i')|| \leq \zeta ||U_i - U_i'||$$

and therefore converges towards a fixed point.

Value Iteration

Intuition 2

We know that there only exists a fixed number of different policies. When we can ensure to improve the policy in each iteration, we will find the optimal policy (fixed point).



Policy Iteration

The Bellman equation: Overview

The two iterative approaches to find a policy for a Markov Decision Process

1. Value Iteration

- Start with a random value function
- In each value iteration:

1. <u>Find the optimal value function</u>: find a new improved value function for each state in an iterative process till value functions converge
2. <u>Policy extraction</u> : Retrieve the optimal policy (action for each state) from the optimal value function for each state

$$V \xrightarrow{\textit{improve}} V \xrightarrow{\textit{extract}} \pi^*_V$$

2. Policy Iteration

- Start with a random policy
- In each policy iteration:

1. <u>Policy evalution:</u> find the utilities for the given policy for each state
2. <u>Policy improvement</u>: find an improved policy based on the actual utilites for each state

$$\pi_0 \xrightarrow{\textit{evaluate}} V_{\pi_0} \xrightarrow{\textit{improve}} \pi_1 \xrightarrow{\textit{evaluate}} V_{\pi_1} \xrightarrow{\textit{improve}} \dots \pi^*_K$$

Both approaches calculate the optimal utility (→ allow to deduce the optimal policy) for each state s

Conclusion

Key takeaways

- We learned to model Markov Decision Processes
- We learned how to compute optimal V-Values by using **value iteration** or **policy iteration**
  - Value iteration **maximizes over actions**
  - Policy iteration **uses a fixed policy**
- Value iteration and Policy iteration use **one-step lookahead expectimax fragments**
- Value iteration and Policy iteration are variants of **Bellman updates**

Questions to think about

- What is the difference between a Markov Chain and a Markov Decision Process?
- Why does value iteration and policy iteration converge?
- What is the difference between value iteration and policy iteration?

In the next lecture, we'll introduce **Reinforcement Learning,** that learns an optimal policy without knowing the underlying transition function and reward function!

Related literature

- R. S. Sutton and A. G. Barto, „Reinforcement Learning: An Introduction“ (Chapter: 3 Finite Markov Decision Processes and Chapter 4: Dynamic Programming)

References

[1] R. S. Sutton and A. G. Barto, „Reinforcement Learning: An Introduction“, second edition, MIT Press, Cambridge, MA, 2018 (available online at: http://incompleteideas.net/book/RLbook2020.pdf)

[2]: J. Spreer, Tech & Nature, „Angsa Robotics: Münchener Roboter „Clive“ reinigt Parks lupenrein “ (available online at: https://www.techandnature.com/angsa-robotics-munchener-roboter-clive-reinigt-parks-lupenrein/)

[3]: UC Berkeley, lecture slides “Markov Decision Processes I+II” within CS 188 Introduction to Artificial Intelligence (Spring 2014) (available online at: http://ai.berkeley.edu/lecture_slides.html)

Topics for this week

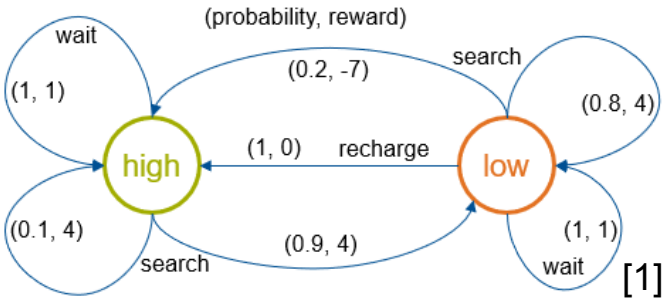
Agenda

1. Markov Decision Process
2. Reinforcement Learning
3. Deep Q Networks
4. Advanced DRL Architectures

Differences between value/policy iteration and reinforcement learning

We still consider **a Markov decision process** (MDP), including

- A state space S
- An action space $A(s),\ s \in S$
- A transition function $T(s, a, s'),\ s, s' \in S,\ a \in A(s)$
- A reward function $R(s, a, s'),\ s, s' \in S,\ a \in A(s)$



[1]

We still want to find a „good“ (as close to optimal as possible) **policy** $\pi(s),\ s \in S$

However, we now assume that **transition and reward function are unknown**, i.e.,

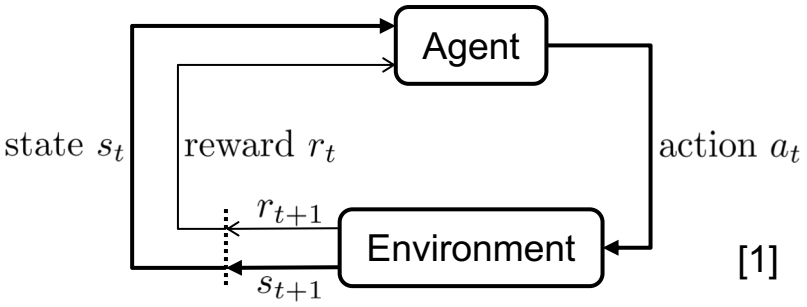
- The impact of actions is unknown
- We do not know which states are good

➔ We must try actions and states to **learn** about the environment

Basic ideas behind reinforcement learning

Basic ideas

- Agent receives **feedback** in the form of rewards (agent's utility is given by reward function)
- Agent learns to act according to a policy that **maximizes its expected rewards**
- Learning is based on **observed samples** of outcomes



[1]

Fundamental challenges

- **Exploration**: an agent must try unknown actions to get information (but eventually must exploit the obtained knowledge)
- **Generalization**: what an agent learns in one state might apply to other states, too

Offline (MDPs) vs. Online (RL)

Offline

- When solving MDPs, one can search **offline** for a **solution**, without actually interacting with the environment, as actions can be planned, and their impact anticipated
- **Example**: If one knows the rules of a board game (e.g., chess), one can (in theory) find out how to play well by just thinking about it, without playing the game

Online

- In RL, the agent must actually interact with the environment since the impact of actions is unknown – this is called **online learning**
- **Example:** If one does not know the rules of a board game (and can also not look them up), one must play the game to learn the rules and either subsequently or simultaneously find out how to play well ("learning by doing")

Both methods can nevertheless be used to **solve online control problems**, where decisions are taken sequentially in each time step based on the latest information available (vs. offline planning problems where decisions for the entire time horizon are taken in advance)

Passive reinforcement learning: direct evaluation

Goal

Learning state values without knowledge or explicit model of transitions and rewards

Idea

Agent executes a fixed input policy, i.e., it has no choice what actions to take, and learns from experience

- The agent acts according to **input policy**
- Every time a state is visited, the **observed sum of discounted rewards** is written down
- Those **sample values are averaged**

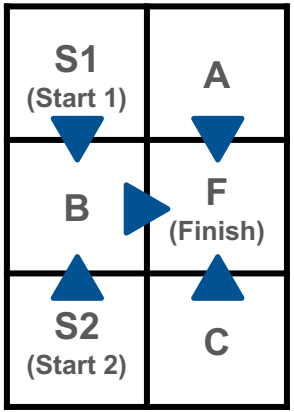
Pseudo code

Initialize state value to zero for all states
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose $a \in A(s)$ using input policy π
    Take action a, observe r, s'
    $s \leftarrow s'$
  until s is terminal
  For each state visit, compute sum of discounted rewards obtained afterwards until end of episode
  For each state visit, add sum to state value
Divide each state value by no. of state visits

This is **not offline planning** as the agent takes actions in the environment and **not model-based RL** since we do not build an explicit model of the environment but directly obtain state values

Example: direct evaluation

Input policy π



$(\gamma = 1)$

Observed samples (training)

Episode 1

S1, ↓, B, -1
B, →, F, +5

Episode 2

S1, ↓, B, -1
B, →, F, +5

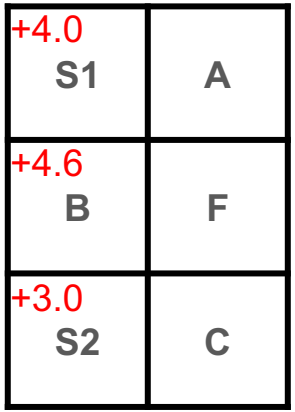
Episode 3

S2, ↑, B, -1
B, →, F, +5

Episode 4

S2, ↑, B, -1
B, →, S1, -1
S1, ↓, B, -1
B, →, F, +5

Output values



If S1 and S2 both go to B under this policy, how can their values be different?

Pros and cons of direct evaluation

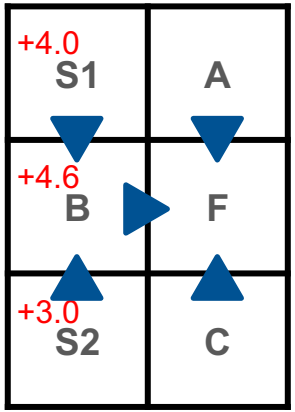
Pros:

- Does not require knowledge of transition/reward function
- Eventually computes the correct average values, using just sample transitions
- Is easy to understand and implement

Cons:

- Wastes information about state connections
- Learns value for each state separately
- Consequently, takes a long time to learn

Output values



If S1 and S2 both go to B under this policy, how can their values be different?

Why not use policy evaluation?

Policy evaluation

Simplified **Bellman updates used to calculate value function** for a fixed policy: each iteration, value function estimate is improved with a one-step-look-ahead, **exploiting information about state connections**

$$V_0^\pi(s) = 0 \quad , \quad V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') \left[R(s, \pi(s), s') + \gamma V_k^\pi(s') \right]$$

Problem: knowledge of transition/reward function needed to do this

Idea: sample-based policy evaluation to do value function update without this knowledge

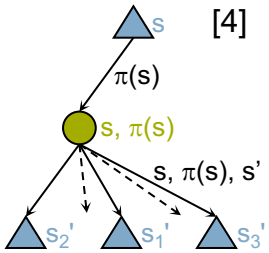
Sample-based policy evaluation

Agent does action $\pi(s)$ multiple times to **get samples of outcomes** s' **; observed values are averaged**:

$$V_{k+1}^\pi(s) \leftarrow \frac{1}{n} \sum_i \text{sample}_i \quad \text{with } \text{sample}_1 = r_1 + \gamma V_k^\pi(s_1') \ , \ \dots \ , \ \text{sample}_n = r_n + \gamma V_k^\pi(s_n')$$

- Here, r_i denotes the observed reward corresponding to outcome s_i'
  (always: R denotes the reward function, r denotes an observed/sampled reward)
- Through sampling, we approximate T and R, and thereby $V_{k+1}^\pi(s)$ (which is an expected value)

Problem: we cannot rewind time over and over again to get sample after sample starting from state s

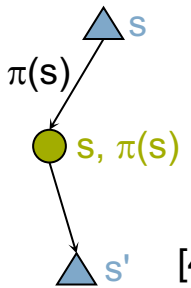


[4]

Temporal difference learning (TD learning)

Idea: learning immediately from every experience

- Value function $V(s)$ is **updated after each transition** (s, a, s', r)
- Value function estimate is not fully replaced, but **adapted slightly towards the most recent sample** obtained through acting in the environment, using running average
  → no need to rewind time to get multiple outcomes s'
- Still, over time, **likely outcomes** s' **will contribute updates more often**
- Temporal difference learning allows to update the value function while **interacting with the environment along an entire trajectory** without rewinding time



[4]

"If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning." (Sutton/Barto) [1]

Temporal difference update

Exponential moving average

Running interpolation update: $\bar{x}_n = (1-\alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$

- Makes **recent samples more important** while forgetting the past (distant past values wrong anyway[1])
- **Decreasing "learning rate"** α can give **converging** averages

Temporal difference update

- We still consider a fixed input policy, and still perform **evaluation**
- We **move the value function estimate towards the most recent sample** obtained through acting in the environment, using **exponential moving average** with "learning rate" α:

$$\text{sample} = r + \gamma V^\pi(s') \quad , \quad V^\pi(s) \leftarrow (1-\alpha) \cdot V^\pi(s) + \alpha \cdot \text{sample}$$

- **Equivalent update formula** shows why this is called temporal difference learning:

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha \cdot (\text{sample} - V^\pi(s))$$

- **Learning rate is a hyperparameter** (to be chosen by user of algorithm, not given by problem setting)

[1] Since early values depend strongly on the chosen initialization. When the policy is not fixed anymore (later in this lecture), past values might be wrong since they were obtained under an old policy.

Example: temporal difference learning

States

S1 (Start 1)	**A**
B	**F** (Finish)
S2 (Start 2)	**C**

$(\gamma = 1)$

Observed transitions and value function updates

S1, ↓, B, -1	B, →, F, +5	Restart	S1, ↓, B, -1	B, →, F, +5

0	0	-0.5	0	-0.5	0	-0.5	0	0.5	0	0.5	0
0	0	0	0	2.5	0	2.5	0	2.5	0	3.75	0
0	0	0	0	0	0	0	0	0	0	0	0

$$V^\pi(s) \leftarrow (1-\alpha) \cdot V^\pi(s) + \alpha \cdot (r + \gamma V^\pi(s'))$$

Assume $\alpha = 0.5$ for this illustration (not a good value for real applications)

The problem with TD value learning

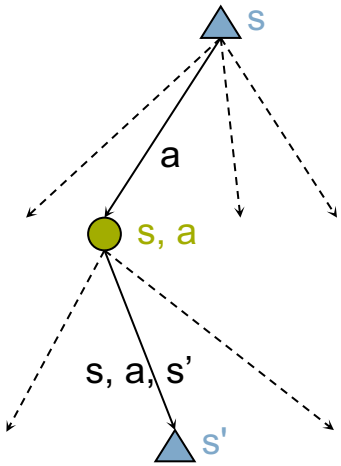
The problem

- TD value learning is a model-free method to do policy evaluation
- However, we want to find a good policy – so far, we have only evaluated the input policy which is most likely not (close to) optimal
- So, we need to turn the obtained value function estimates into a new policy:

$$\pi(s) = \arg\max_a \; Q(s,a)$$

$$Q(s,a) = \sum_{s'} T(s,a,s') \cdot [R(s,a,s') + \gamma V(s')]$$

- In our model-free setting, this does not work, since we do not know the transition and reward functions



[4]

New idea

- Learning Q-values instead of V-values
- This would make action selection (finding a good policy) model-free, too

Q-value iteration

Value iteration

Finding the value function by repeating a **value update until convergence**:

- Starting with $V_0(s) = 0$
- Then, given V_k, **iteratively calculating improved values for all states**:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

Q-value iteration

Q-values are **more useful in our model-free context** since we can directly extract a policy from them, so we want to compute them instead:

- Starting with $Q_0(s,a) = 0$
- Given Q_k, **calculating the next Q-values for all Q-states**:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right]$$

Q-learning: overview

Pseudo code for Q-learning [1]

Initialize $Q(s,a)$ for all $s \in S, a \in A(s)$, arbitrarily
Initialize $Q(\text{terminal state}, \cdot) = 0$
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose $a \in A(s)$ using policy derived from Q (e.g., ϵ-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
  until s is terminal

Interacting with environment and learning Q-values over multiple episodes until some stopping criteria is reached (e.g., total no. of episodes or change of Q-values below threshold)

No longer policy evaluation! There is no input policy anymore. Transitions are sampled based on policy derived from current Q-values (argmax over actions). Need for exploitation/exploration strategy (more on that later).

TD update to Q-values

Q-learning: sample-based Q-value iteration

Q-value iteration

Finding the Q-value function by **repeating a Q-value update until convergence** for all Q-states:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a') \right]$$

Sample-based Q-value iteration

Same principle as in TD value learning – now, **temporal difference Q-value learning**

- Reminder: TD learning updates value function while interacting with environment along full trajectory
- Transition sample is received: (s, a, s', r)
- Old estimate $Q(s,a)$ and new sample estimate $\text{sample} = r + \gamma \max_{a'} Q(s',a')$ are considered
- New estimate is incorporated into running average: $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot \text{sample}$
- Equivalently: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$

Example: Q-learning [4]

Q-learning: properties

Q-learning is “full” RL

- **Goal**: learning the optimal policy without knowledge of transition and reward functions
- **Actions are actually taken** in the environment and outcomes observed (in contrast to offline planning)
- Actions are **chosen dynamically by the learning agent** (in contrast to policy evaluation)
  - Thus, Q-learning is **“active RL”** (in contrast to “passive RL” with a fixed input policy)

Convergence

Q-learning converges to the optimal policy (even if the learning agent acts suboptimally)

Challenges

Learning rate (cf. Lecture 5 for details on learning rates):

- To ensure that running average of Q-values converges, learning rate must eventually be small enough
- The smaller the learning rate, the slower the learning process (i.e., more iterations needed)
- Can be difficult to find appropriate learning rate and decrease it not too fast but also not too slow

Fundamental tradeoff between exploration and exploitation: see next slides

Exploitation vs. exploration

Exploitation

The agent tries to **exploit what it has already learned**

- For example, if a certain state-action pair leads to a positive reward, the agent chooses this action when it is in this state (and the other way around, avoiding negative rewards)

Exploration

The agent tries to **acquire new information** about states/actions that are …

- … unknown because they were not visited yet, or …
- … not sufficiently explored since they were visited already but the outcome is still ambiguous

Assuming the agent is in a state:

- Should it choose the action that will certainly lead to a positive reward?
- Or should it explore if a different action would lead to an even better reward (although thereby it risks to obtain a zero/negative reward)?

How to exploit and explore

How to exploit

Acting on current policy, as this is the latest “best guess” what the optimal behavior is

For Q-learning, this means to choose actions according to policy based on current Q-values:

$$\pi(s) = \arg\max_a \; Q(s,a)$$

How to explore

Several schemes for forcing exploration exist

Simplest: ϵ**-greedy**

- In every step, the agent acts randomly with (small) probability ϵ
- It acts on current policy with (large) probability $1-\epsilon$
- State-action space is explored, but agent keeps acting randomly even when learning is already done (can partly be mitigated by lowering ϵ over time)

Example: Q-learning with ϵ-greedy exploration [4]

Conclusion

Key takeaways

- RL is used to find optimal policies when transition/reward function are unknown
- We distinguish between model-based and model-free RL and focus on the latter
- TD learning allows to learn value function while interacting with the environment
- Q-learning lets agent choose actions based on current Q-value estimates
- An exploration strategy is needed to address the exploration/exploitation tradeoff

Questions to think about

- How can function approximation be incorporated into Q-learning to generalize (thereby enabling our first full RL algorithm to be used to solve large-scale problems)?

Related literature

- R. S. Sutton and A. G. Barto, „Reinforcement Learning: An Introduction“ (Chapter 6: Temporal-Difference Learning) [1]

References

[1] R. S. Sutton and A. G. Barto, „Reinforcement Learning: An Introduction“, second edition, MIT Press, Cambridge, MA, 2018 (available online at: http://incompleteideas.net/book/RLbook2020.pdf)

[2] A. Poddiachyi, „Reinforcement Learning, Brain, and Psychology: Introduction“ (available online at: https://towardsdatascience.com/reinforcement-learning-brain-and-psychology-part-1-introduction-b5f79a0475ab)

[3] Two Minute Papers, “Google DeepMind's Deep Q-learning playing Atari Breakout” (available online at: https://www.youtube.com/watch?v=V1eYniJ0Rnk)

[4] UC Berkeley, lecture slides “Reinforcement Learning I/II” within CS 188 Introduction to Artificial Intelligence (Spring 2021) (available online at: https://inst.eecs.berkeley.edu/~cs188/sp21/)

Topics for this week

Agenda

1. Markov Decision Process
2. Reinforcement Learning
3. Deep Q Networks
4. Advanced DRL Architectures

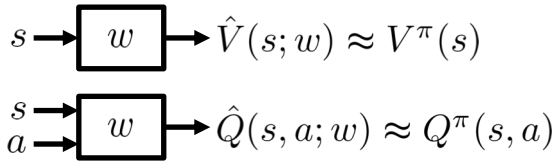
From tabular RL to large-scale RL with function approximation

Tabular case

- **Every state** (or state action pair) has an **individual value** that must be learned and stored in memory
- For interesting problems, MDPs are typically very **large** (e.g., the board game Go has 10^{170} states) or the state space is **continuous** (e.g., for robot control)
- **Learning** then becomes **too slow** or impossible and **memory requirements too large**

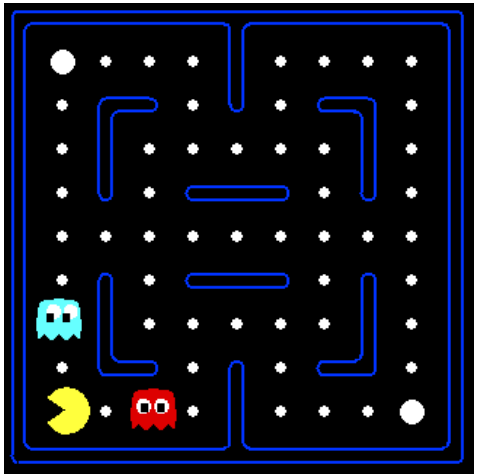
Value function approximation

- Value function is **estimated** using an approximation
- Approximate function is **parameterized** by weights w
- These **parameters are learned** instead of learning the value directly
- Then, we can **generalize** from seen to unseen states, since we learned a value function that can take any state as input

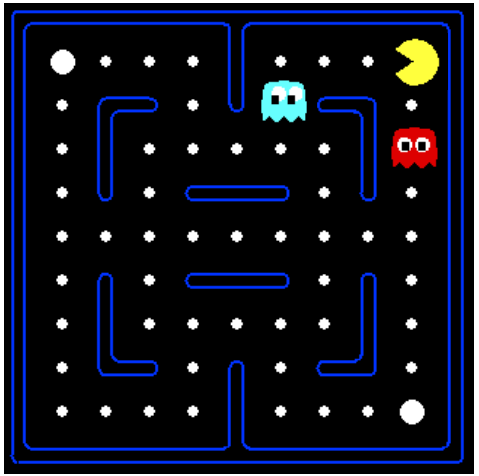


From tabular RL to large-scale RL with function approximation

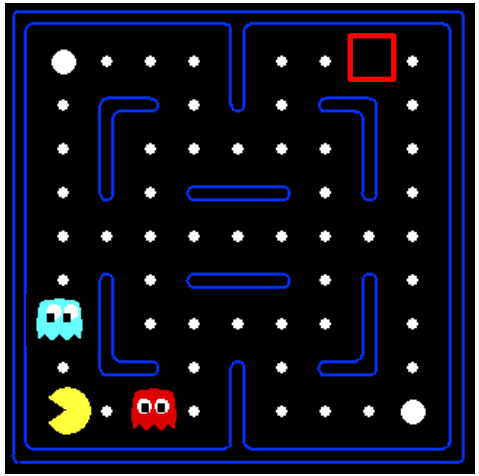
Let's say we discover through experience that this state is bad:



In naïve q-learning, we know nothing about this state:



Or even this one!



[7]

Example: linear Q-function approximation

- State-action pair can also be represented by a feature vector: $f(s,a) = (f_1(s,a), f_2(s,a), ..., f_n(s,a))^T$
- The corresponding Q-function approximation is given by $\hat{Q}(s,a;w) = f(s,a)^T w$
- The gradient is given by $\nabla_w \hat{Q}(s,a;w) = f(s,a)$

Example: linear value function approximation

Reminder: feature representation

- Describe state using feature vector $f(s) = (f_1(s), f_2(s), ..., f_n(s))^T$
- Examples of features: distance of robot from landmark, no. of ghosts (Pacman)
- Value function is linear combination of features: $\hat{V}(s; w) = f(s)^T w = \sum_{i=1}^{n} f_i(s) \cdot w_i$

SGD update for linear value function approximation

- Loss function: $J(w) = \mathbb{E}_\pi \left[(V^\pi(s) - f(s)^T w)^2 \right]$
- SGD update: $\Delta w = \alpha \cdot \left(V^\pi(s) - \hat{V}(s; w) \right) \cdot \nabla_w \hat{V}(s; w) = \alpha \cdot \left(V^\pi(s) - \hat{V}(s; w) \right) \cdot f(s)$
- $\text{Update} = \text{learning rate} \cdot \text{prediction error} \cdot \text{feature value}$

Value function parameter update in RL

TD target in value function approximation

- Want to update parameters of value function estimate using $\Delta w = \alpha \cdot \left(V^\pi(s) - \hat{V}(s;w)\right) \cdot \nabla_w \hat{V}(s;w)$
- In RL, we **do not know the true value function** $V^\pi(s)$, since only transition/trajectory data from the agent's interaction with the environment can be used for training
- Thus, we must replace the true value function with an **estimate based on observed rewards**: we substitute a "target" for $V^\pi(s)$
- Considering a transition (s, a, r, s'), we can use the idea from TD learning and use the sampled **TD target** $r + \gamma \cdot \hat{V}(s';w)$ as an estimate for the true value function, leading to a new update rule:

$$\Delta w = \alpha \cdot \left(r + \gamma \cdot \hat{V}(s';w) - \hat{V}(s;w)\right) \cdot \nabla_w \hat{V}(s;w) =: \alpha \cdot \delta \cdot \nabla_w \hat{V}(s;w)$$

- This makes the target dependent on the parameters w which we want to update – called **semi-gradient method**, no convergence guarantee (but can work well in practice)

Example: linear value function approximation

$$\hat{V}(s;w) = f(s)^T w \quad , \quad \Delta w = \alpha \cdot \delta \cdot f(s)$$

Q-function approximation

Q-learning with function approximation

- Analogously to the value function, we can do the **same procedure for the Q-function**, to get:

$$\hat{Q}(s,a;w) \approx Q^{\pi}(s,a)$$

$$\Delta w = \alpha \cdot \left(r + \gamma \cdot \max_{a'} \hat{Q}(s',a';w) - \hat{Q}(s,a;w)\right) \cdot \nabla_w \hat{Q}(s,a;w)$$

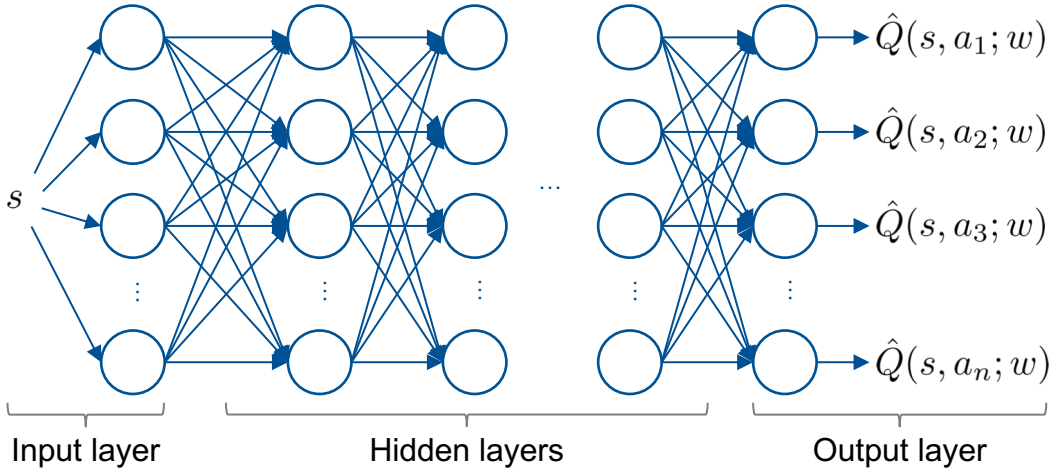
- This gives a **Q-learning algorithm with function approximation**: the update of the tabular Q-values is replaced by the update of the parameters derived above, everything else remains unchanged

Example: linear Q-function approximation

- State-action pair can also be represented by a feature vector: $f(s,a) = (f_1(s,a), f_2(s,a), ..., f_n(s,a))^T$
- The corresponding Q-function approximation is given by $\hat{Q}(s,a;w) = f(s,a)^T w$
- The gradient is given by $\nabla_w \hat{Q}(s,a;w) = f(s,a)$

Q-learning with function approximation is **not guaranteed to converge** – however, using (deep) **neural networks** as function approximators can lead to **converging algorithms in practice**

Neural networks as (non-linear) function approximators (1/2)



- **Backpropagation gives gradient** of Q-value estimates w.r.t. parameters (weights/biases of the NN)
- Thus, **parameter update** remains the same as before, only the procedure to calculate the gradient changes
- We now have a **Q-learning algorithm with NN** as function approximator

State is input, depending on problem and NN architecture this can be, e.g., a vector or a picture

No more hand-crafted features, “black box”

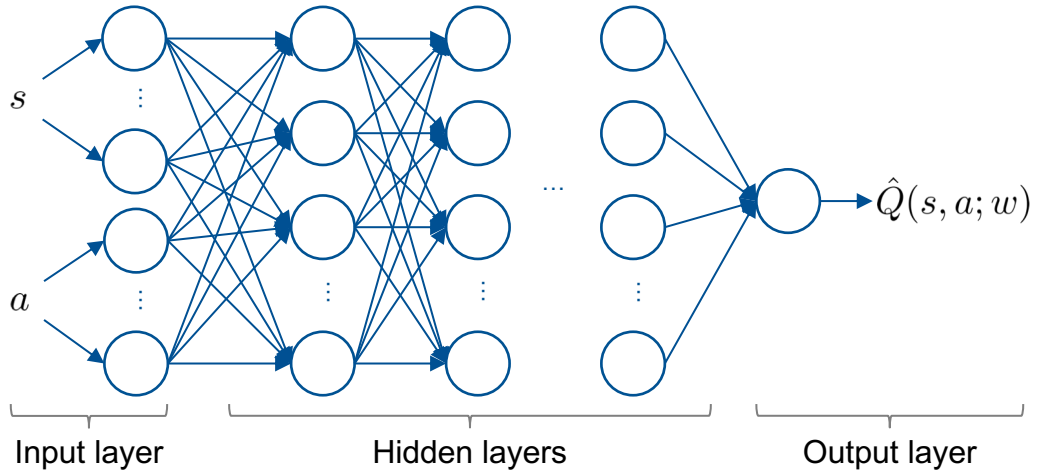
One Q-value estimate per action, usually feed-forward layer without activation s.t. Q-values can take any real number

The network can have any kind of architecture, incl. feed-forward, convolutional, and recurrent layers

Neural networks as (non-linear) function approximators (2/2)

Alternative structure of input and output of the NN which may be used (depending on the specific setting):

State and one specific action taken from that state as input



One Q-value as output (for the action given as input)

This structure may also be used to estimate the value function $\hat{V}(s; w)$ using a NN, the only difference is that no action but only the state is given as input to the network

Convergence problems of vanilla Q-learning combined with NNs

Q-learning algorithm as we know it in combination with NNs still has convergence problems for two reasons:

Correlations between samples

- We learn from successive transitions, which are correlated (not independent)
- E.g., in video games, two successive frames look very similar (often with identical optimal action)
- However, to properly approximate the expectation from the original loss function, we need random (independent) samples
- Correlated samples cause the NN to overfit – it gets trapped in a suboptimal local minimum

Solution: experience replay

Non-stationary targets

- The target $r + \gamma \cdot \max_{a'} \hat{Q}(s', a'; w)$ to which we fit $\hat{Q}$ depends on the parameters w and thus changes with each training iteration
- I.e., the parameter update tries to hit a moving target, “like a dog chasing its tail”
- NNs do not perform well on such non-stationary problems

Solution: fixed Q-targets through target networks

Explanations based on julien-vitay.net/deeprl/Valuebased

Experience replay: idea

- We **want to decorrelate the samples** used for training, i.e., successive parameter updates should not be based on transitions that occur successively when the agent interacts with the environment
- In Q-learning as introduced so far, the **action** taken by the agent in the environment is **not always derived from the current Q-values** (e.g., due to ϵ-greedy exploration), but the update to the Q-value function is computed based on the transition following that action
- I.e., we may **learn from transitions that are not based on the most recent Q-value estimate**
- **New idea** based on this observation:
  - **Store transitions** when they are observed/experienced; for parameter update, **randomly sample transitions** from the storage (we can do this based on the insight described above)
  - This **decorrelates the transitions** which are used for successive parameter updates, as the random sampling will usually not successively choose transitions which were also successively observed in the environment
  - In addition, this allows to **reuse transitions** as they are stored and can be sampled multiple times, making learning more efficient

Experience replay: details

Adding data to replay buffer

- The storage $\mathcal{D}$, called replay buffer, is initialized with some **maximal size**
- When a transition (s, a, r, s') is observed through interaction with the environment, it is added to the replay buffer ("**experience collection**")
- If the replay buffer is already full, the **oldest stored data point is overwritten**

Using data from replay buffer for parameter updates

- In each training iteration, a **(mini-)batch of transitions** $D \subset \mathcal{D}$ **is randomly sampled** from the replay buffer (i.e., (mini-)batch size $|D|$ many transitions are sampled)
- Stochastic gradient descent **update** is applied **for each transition** in the (mini-)batch:

$$\Delta w = \alpha \cdot \left(r + \gamma \cdot \max_{a'} \hat{Q}(s', a'; w) - \hat{Q}(s, a; w) \right) \cdot \nabla_w \hat{Q}(s, a; w)$$

- On GPUs, the computation of the SGD updates can be efficiently **parallelized** across the (mini-)batch
- Thereby, the collected experience is used multiple times (thus called "experience replay") and **learning becomes faster** as multiple parameter updates are applied between experience collection steps

Fixed Q-targets through target networks

- We want to have a **stationary (fixed) target** for the parameter update, instead of the following target which changes with each parameter update: $r + \gamma \cdot \max_{a'} \hat{Q}(s', a'; w)$
- We introduce a **second Q-network**, called the **target network**, which is an exact copy of the primary Q-network except for its parameter values, which we denote by w^-
- The **parameters of the target network** are not trained, but **copied** over from the primary network **at fixed intervals**
- The target value for the **parameter update of the primary Q-network** is calculated using the target network:

$$\Delta w = \alpha \cdot \left(r + \gamma \cdot \max_{a'} \hat{Q}(s', a'; w^-) - \hat{Q}(s, a; w) \right) \cdot \nabla_w \hat{Q}(s, a; w)$$

- Thereby, the **target is stationary** between the updates of the target network parameters
- It still becomes a better estimate of the true target over the long-term since the **target network weights are regularly updated**
- Although the target is still not really fixed, it is much less volatile than before, and **training is stabilized**

Deep Q-Networks (DQN)

We combine **Q-learning** with **deep NNs** as function approximators and **experience replay** and **target networks** to get **DQN** [2], an algorithm which marked a breakthrough for deep RL

Pseudo code for DQN

Initialize empty replay buffer $\mathcal{D}$
Initialize network weights w, $w^- \leftarrow w$
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose $a \in A(s)$ using policy derived from $\hat{Q}$ (e.g., ϵ -greedy)
    Take action a, observe r, s'
    $d \leftarrow 1$ if s' is terminal, 0 otherwise
    Store (s, a, r, s', d) in $\mathcal{D}$
    Sample random (mini-)batch $D \subset \mathcal{D}$
    $$w \leftarrow w + \alpha \cdot \frac{1}{|D|} \cdot \sum_{(s,a,r,s',d) \in D} \Big(r + \gamma \cdot (1-d) \cdot \max_{a'} \hat{Q}(s', a'; w^-) - \hat{Q}(s, a; w) \Big) \cdot \nabla_w \hat{Q}(s, a; w)$$
    $s \leftarrow s'$
    After fixed no. of total steps: $w^- \leftarrow w$
  until s is terminal

It is also possible to do **several consecutive experience collection** steps between parameter updates

With NNs, we cannot enforce $\hat{Q}(\text{terminal state}, \cdot\,; w) = 0$ anymore, s.t. we need to explicitly account for terminal states in the parameter update, for which we introduce the **"done flag"** d

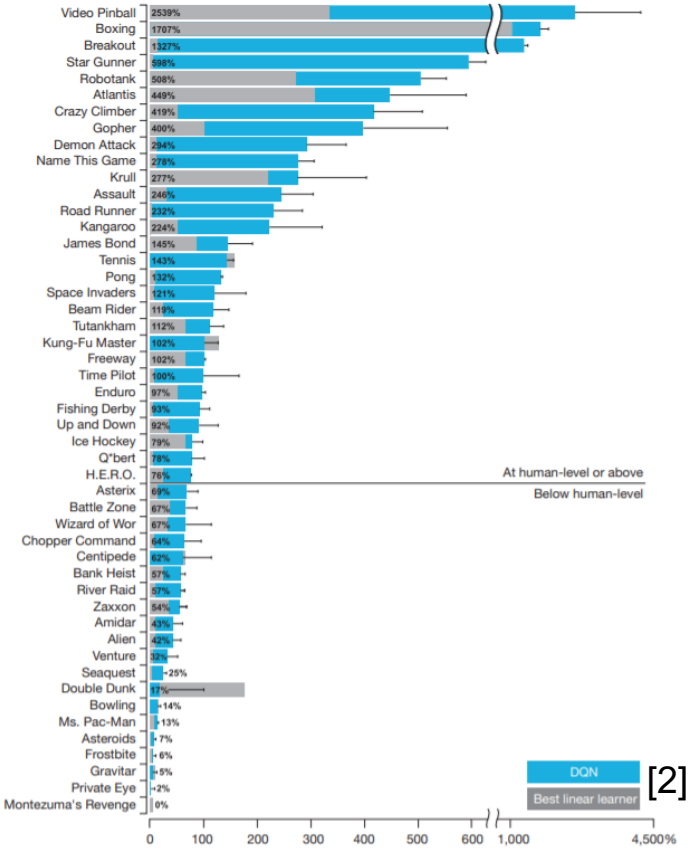
Parameter update accounting for terminal states and averaging over the (mini-)batch

Hyperparameters in DQN

- The user of DQN **must specify many hyperparameters**, e.g.:
  - NN architecture
  - If ϵ-greedy used for exploration: ϵ
  - Learning rate
  - Size of replay buffer
  - Size of (mini-)batch
  - Target network update interval
- **Performance of DQN depends strongly on the choice of hyperparameters**
- What is a good choice for a specific parameter **depends strongly on the problem** at hand and can usually not be properly determined a priori using heuristics
- Thus, hyperparameters must be **tuned** empirically in a trial-and-error process

Example: DQN to play Atari games

- In their 2015 Nature publication, Mnih et al. use DQN to play various Atari games and **surpass the performance of professional human players** for many of them
- A combination of convolutional and fully-connected layers is used
- **Input** to network: raw pixels (identical to what is displayed on the screen of a human player)
- **Action space**: the 18 possible joystick/button positions
- **Reward**: change in score within a step
- The network gives the Q-values for all possible actions in a single forward pass
- **Q-values are learned end-to-end directly from pixels**



[2]

DQN: importance of experience replay and target network

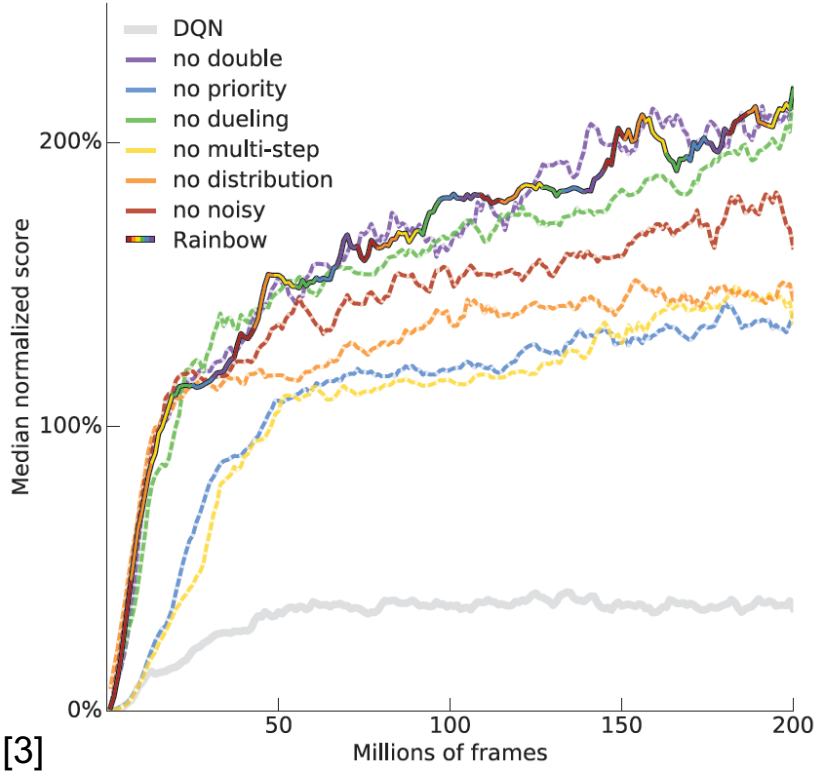
Scores for a selection of Atari games with and without experience replay and a target network

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

[2]

- Empirically, one observes that both experience replay and the target network are important components to ensure the algorithm's performance
- The magnitude of the respective effects varies between environments, but both experience replay and the target network always have a strongly positive impact

Rainbow: improvements of DQN



[3]

- **Double Q-learning** [4]: reduces overestimation bias
- **Prioritized replay** [5]: prioritizes transitions from which there is much to learn when sampling from the replay buffer
- **Dueling networks** [6]: improves network architecture leading to better Q-value estimates *Deep-dive in the following*
- **Multi-step learning**: incorporates multiple steps to calculate the TD target, leading to faster learning
- **Distributional RL** [7]: learns to approximate the distribution of returns instead of the expected return
- **Noisy nets** [8]: adapts NN to improve exploration

In [3], the improvements are combined into "**rainbow**" and the impact of the individual components on performance is studied (see figure on the left)

Conclusion

Key takeaways

- To generalize to unseen states and use RL for large problems, we use differentiable function approximators to estimate a parameterized value function
- SGD and TD learning can be combined to obtain a Q-learning algorithm with function approximation
- NNs with state as input and one output node per action can approximate Q-values
- Experience replay and target networks are important to overcome convergence issues caused by correlated samples and non-stationary targets
- DQN integrates these ingredients into one algorithm

Questions to think about

- Can we use DQN if the action space is continuous (e.g., in robotic control)?
- So far, we have always learned a value function and derived a policy/actions from it – could we also directly parameterize a policy using NNs?

Related literature

- R. S. Sutton and A. G. Barto, „Reinforcement Learning: An Introduction“ (Chapter 9: On-Policy Prediction with Approximation) [1]
- Papers on next page, in particular [2]

References

[1] R. S. Sutton and A. G. Barto, „Reinforcement Learning: An Introduction“, second edition, MIT Press, Cambridge, MA, 2018 (available online at: http://incompleteideas.net/book/RLbook2020.pdf)

[2] V. Mnih, et al., “Human-level control through deep reinforcement learning”, Nature 518 (2015) 529-533

[3] M. Hessel, et al., “Rainbow: Combining Improvements in Deep Reinforcement Learning”, AAAI 2018

[4] H. van Hasselt, et al., “Deep Reinforcement Learning with Double Q-Learning”, AAAI 2016

[5] T. Schaul, et al., “Prioritized Experience Replay”, ICLR 2016

[6] Z. Wang, et al., “Dueling Network Architectures for Deep Reinforcement Learning”, ICML 2016

[7] M. G. Bellemare, et al., “A Distributional Perspective on Reinforcement Learning”, ICML 2017

[8] M. Fortunato, et al., “Noisy Networks for Exploration”, ICLR 2018

Topics for this week

Agenda

1. Markov Decision Process
2. Reinforcement Learning
3. Deep Q Networks
4. Advanced DRL Architectures

Advanced DRL Architectures

… REINFORCE & Actor Critic

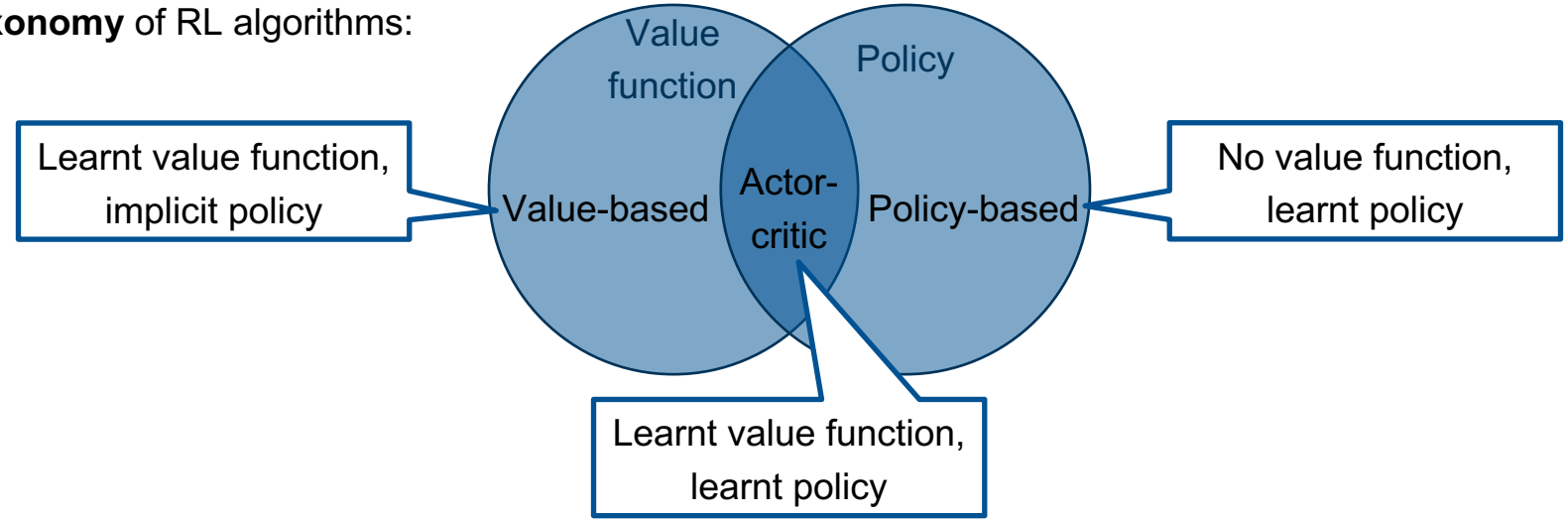
… TD(λ), TRPO/PPO

From value-based to policy-based RL

- So far, we approximated the value or Q-value function using parameters w:

$$\hat{V}(s;w) \approx V^{\pi}(s)\ ,\ \hat{Q}(s,a;w) \approx Q^{\pi}(s,a)$$

- A policy was derived from the Q-value function (e.g., an ϵ-greedy policy)
- Now, we will **directly parameterize the policy** with parameters θ: $\pi(s,a;\theta) = P(a|s;\theta)$
- **Taxonomy** of RL algorithms:



Why policy-based RL?

Motivating example

- Assume that two players play multiple games of rock-paper-scissors against each other
- A player with a **deterministic policy** always chooses the same move (e.g., always rock)
- The other player can exploit this, as the first player's moves become predictable after a few games
- The Nash equilibrium is a uniform random policy, i.e., the **optimal policy is stochastic**
- **With DQN, we cannot learn stochastic policies**, as the action corresponding to the maximal Q-value is always chosen (except for exploration, which is not based on the learned policy)

Advantages of policy-based RL

- Able to learn **stochastic** policies
- Applicable to **continuous** action spaces
- Effective for **high-dimensional** action spaces
- Better **convergence** properties (remember the difficulties making DQN converge)

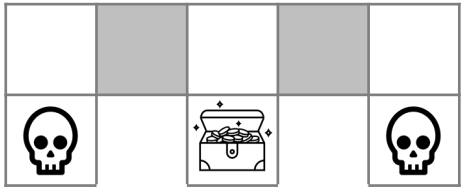
Disadvantages of policy-based RL

- Tends to converge to **local instead of global optima**
- Simple policy-based RL algorithms are **inefficient** and have **high variance** (details will follow)

Example: stochastic policy

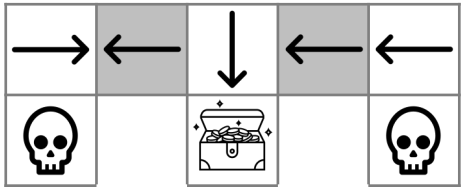
Setting

- Agent starts somewhere in the top row
- Agent can move to the left, right, or down
- Agent cannot distinguish between the two gray states (e.g., because agent sees only what lies immediately to the left/right/above/below)



Value-based RL

- A deterministic policy is learned
- Agent moves either to the right or to the left in both gray states
- Thus, in one of the gray states, the agent gets stuck

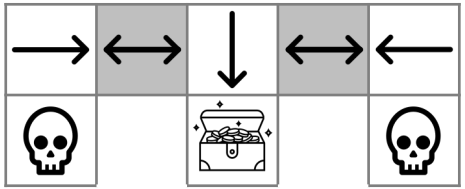


Optimal stochastic policy

- Optimal policy moves left or right in the gray states with equal probability:

$$\pi(\text{gray state}, \leftarrow) = \pi(\text{gray state}, \rightarrow) = 0.5$$

- The goal is reached with high probability within a few steps



Policy search/optimization with policy gradient

- We now want to **find parameters** θ **that maximize a policy objective function** $J(\theta)$
  - Policy-based reinforcement learning tries to solve the **optimization problem** $\max_\theta J(\theta)$
- To do so, approaches without gradients exist (e.g., hill climbing, genetic algorithms), but gradient-based methods are often more efficient
- We again focus on stochastic gradient descent (many extensions of SGD can be used)

Policy gradient

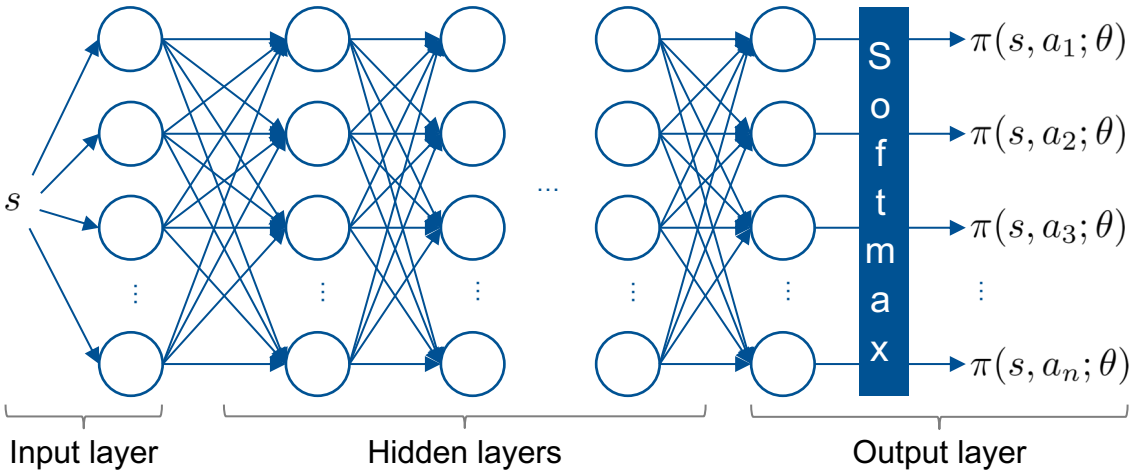
- Since we now maximize the objective function, we use **stochastic gradient ascent**
- So, we iteratively apply the following **update to the parameters of the policy**:

$$\theta \leftarrow \theta + \Delta\theta \quad , \quad \Delta\theta = \alpha \cdot \nabla_\theta J(\theta)$$

- The term $\nabla_\theta J(\theta)$ is called **policy gradient** and we again must choose a learning rate α
- In general, we find a **local** maximum of the policy objective function

Example for differentiable policy: neural network

Illustrative neural network for a discrete action space



- Using **softmax** activation in output layer, the NN outputs a **probability distribution given state** s
- Can be interpreted as **action probabilities** if the number of output nodes corresponds to the number of possible actions
- So, we get a **stochastic policy**

A neural network can represent a parameterized, stochastic policy

REINFORCE: a first policy-based RL algorithm

Idea

- We want to get a **new RL algorithm** which learns "good" parameters of a policy function
- Parameters updated with stochastic gradient ascent, using policy gradient theorem to **sample the gradient**:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\nabla_\theta \log \pi(s,a;\theta) \cdot Q^\pi(s,a)\right]$$

- Therefore, we need to know Q^π – we do not know its value, but can **sample it using the return**[1]

$$G_t = \sum_{k=t}^{T-1} \gamma^{k-t} \cdot r_k$$

- Return is sum of **discounted rewards** along trajectory from time step t to end of an episode at time step T:

$$\tau = \{s_t, a_t, r_t, s_{t+1}, a_{t+1}, ..., s_{T-1}, a_{T-1}, r_{T-1}, s_T\}$$

- To obtain a sample of Q^π, the trajectory must be obtained **executing the policy** π

Pseudo code

Initialize parameters θ

Repeat (for each episode):

Initialize s_1

$\tau \leftarrow \{s_1, a_1, ..., r_{T-1}, s_T\} \sim \pi(s,a;\theta)$

For $t = 1$ to $T-1$:

$G_t \leftarrow \sum_{k=t}^{T-1} \gamma^{k-t} \cdot r_k$

$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi(s_t, a_t; \theta) \cdot G_t$

This algorithm is called **REINFORCE** and is the most basic policy-based RL algorithm with function approximation

Properties of REINFORCE: on-policy (vs. off-policy)

- Parameter updates are based on trajectory data collected with current policy, thus **REINFORCE is an on-policy** algorithm
- In **Q-learning**, experience is partly not collected under policy derived from current Q-values since exploration must be ensured (e.g., ϵ-greedy) – this is called **off-policy** learning
- Note: the policy learned by REINFORCE is stochastic and thus **"automatically" explores** the whole action space, explicit exploration is not needed

- Off-policy algorithms can use a **replay buffer** since the transitions based on which parameters are updated do not need to be collected under current policy (see DQN)
- **On-policy algorithms cannot use a replay buffer** since the policy under which trajectory data must be obtained changes in each iteration
- Not using a replay buffer is **"sample inefficient"**, as collected experience is used only once
- For applications where obtaining sample transitions is expensive, this **can prevent use of on-policy algorithms** if a lot of iterations are needed (e.g., training robots without access to a good simulator)

Properties of REINFORCE: Monte Carlo method (vs. TD-learning)

- In REINFORCE, agent needs to interact with environment for a **full episode before collected data can be used to update parameters**, since return calculation requires rewards for entire trajectory
- Such algorithms are called **Monte Carlo methods (MC)**
- In contrast, **TD-learning updates parameters after each time step**

- **Monte Carlo** methods work only for **episodic environments**
- In **non-terminating environments**, the return can never be calculated and thus **no parameter update** can be calculated using **Monte Carlo methods**

Properties of REINFORCE: bias and variance

Bias

- Monte Carlo methods (incl. REINFORCE) are **unbiased** algorithms
- This means that estimated returns are only based on **observed values, without making assumptions**
- Algorithms based on TD-learning (incl. Q-learning) have **high bias**, since return estimates of the form $r + \gamma V(s')$ make the **strong assumption** that $V(s)$ is correct
- Particularly early in the learning process, $V(s)$ depends strongly on parameter initialization

Variance

- In Monte Carlo methods, estimated returns depend on **full trajectories** (i.e., many transitions), and are thus exposed to "**a lot of randomness** along the way"
- Thus, REINFORCE has **high variance**
- In TD-learning, parameter updates are based on **single transitions**, leading to **low variance**
- The training process is then more **stable** and potentially faster as it makes use of what has already been learned

High variance of REINFORCE is a problem in practice

Actor-critic: reducing variance

Idea

- To **reduce variance**, we need a different approach to get Q^π instead of sampling it using the return
- We can **estimate the Q-values** with a parameterized function: $Q(s, a; w) \approx Q^\pi(s, a)$
- Then, the policy parameters are updated using samples from an **approximate policy gradient**:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_\pi \left[\nabla_\theta \log \pi(s, a; \theta) \cdot Q(s, a; w)\right] \quad , \quad \Delta\theta = \alpha \cdot \nabla_\theta \log \pi(s, a; \theta) \cdot Q(s, a; w)$$

- We now have **two parameterized functions**, each with a set of parameters
  - **Critic**: Q-function with parameters w
  - **Actor**: policy function with parameters θ

Learning the critic

- The critic does **policy evaluation** (how good is the policy π with the current parameters θ?)
- So, e.g., we can use **TD-learning with NNs** (the policy evaluation part of DQN) to learn the critic
- Policy parameters are then updated in the **direction suggested by the critic** (thus the name “critic”)

Conclusion

Some in-between takeaways

- Policy-based RL can learn stochastic policies for discrete or continuous action spaces
- REINFORCE (unbiased, on-policy, MC policy-based algorithm) has high variance
- Actor-critic algorithms (two sets of parameters) reduce variance by introducing a critic

Advanced DRL Architectures

… REINFORCE & Actor Critic

… TD(λ), TRPO/PPO

Motivation: bias/variance trade-off

MC methods

High variance, no bias

- Converge to correct value, even with function approximation
- Not very sensitive to initialization
- Simple

TD-learning

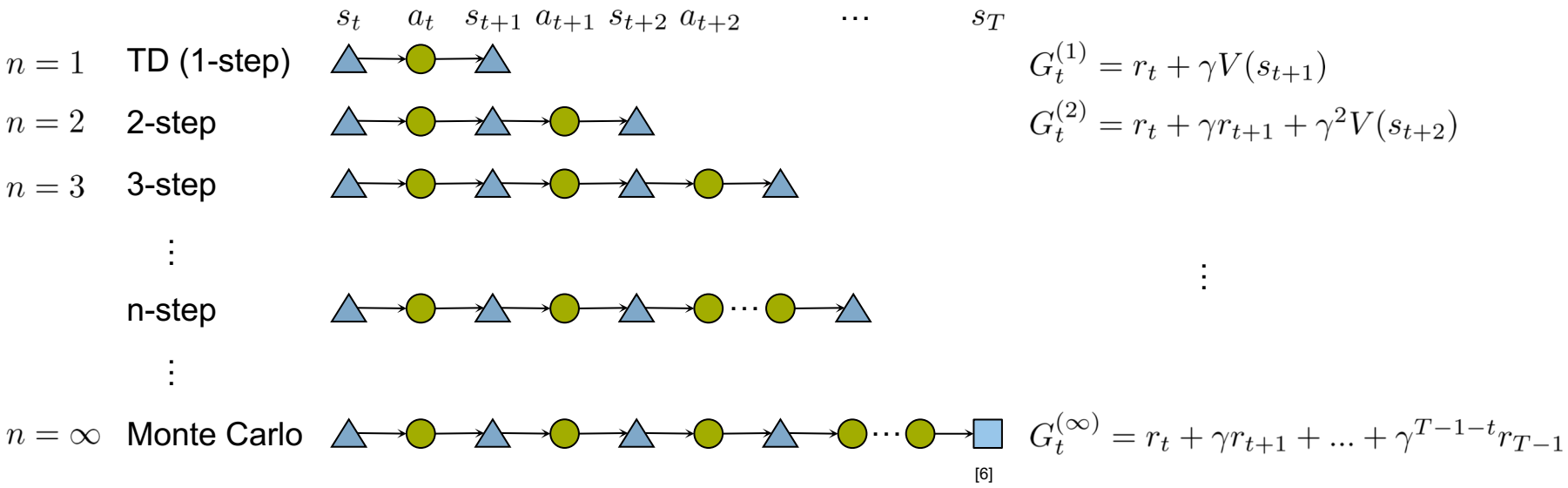
Low variance, biased

- Converge to correct value, but not always with function approximation
- More sensitive to initialization
- Usually more efficient than MC

Can we take an approach “in between” MC and TD, that gives more **control over bias/variance trade-off?**

n-step temporal-difference learning

Idea: we assume a tabular setting for now and **let TD target look n steps into the future**



New update rule

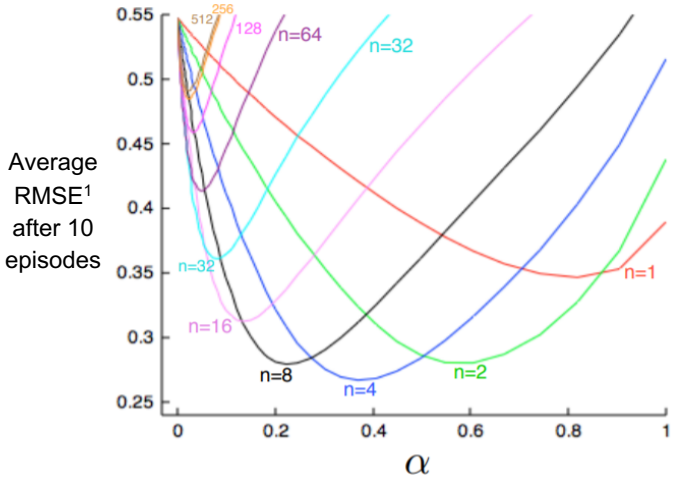
- We define the n-step return $G_t^{(n)} = \sum_{k=t}^{t+n-1} \gamma^{k-t} \cdot r_k + \gamma^n V(s_{t+n})$
- We obtain a new update rule for the tabular value function estimate: $V(s_t) \leftarrow V(s_t) + \alpha \left(G_t^{(n)} - V(s_t) \right)$

Example: n-step temporal-difference learning

Random walk: performance of n-step TD methods for different n as a function of α

Example of random walk

[1]



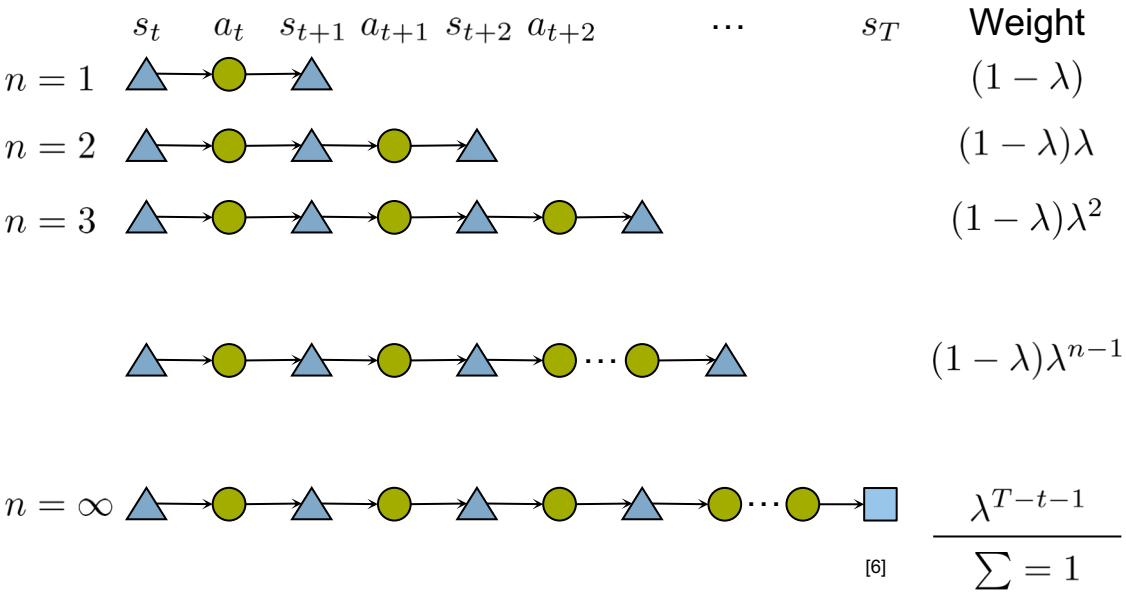
[6]

Intermediate values for n are usually better than the extremes (1-step or Monte Carlo)

1 Root mean squared error between true and predicted value

Forward-view TD(λ): averaging n-step returns

Idea: we combine all n-step returns $G_t^{(n)}$ into one weighted average, the λ-return G_t^{λ}



New update rule

- We define the λ-return

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

- For episodes with terminal state T

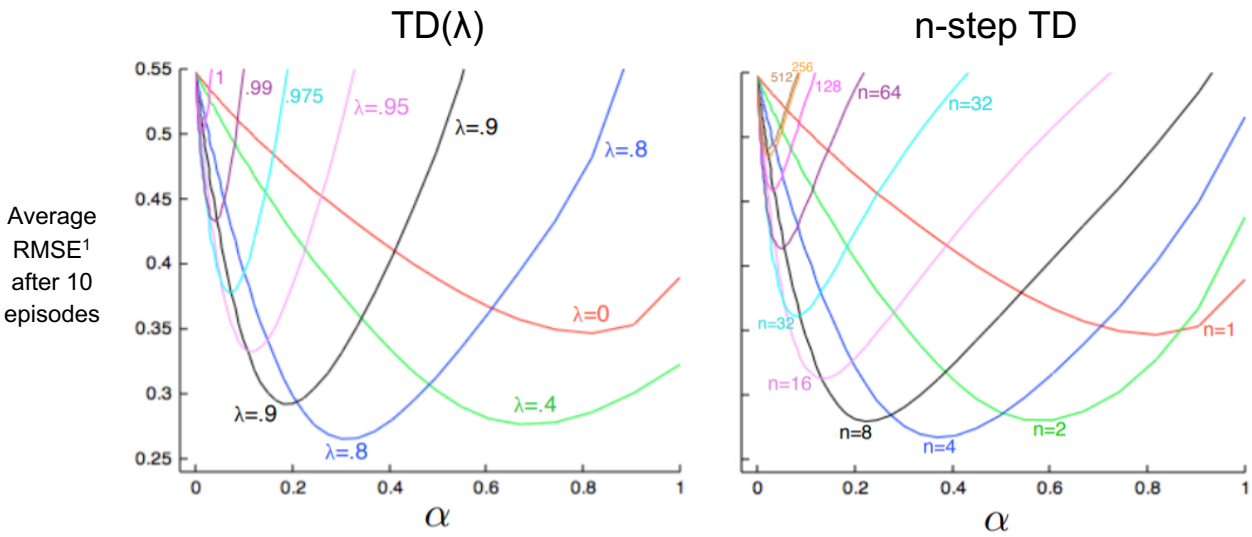
$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_t^{(n)} + \lambda^{T-t-1}G_t$$

- We obtain a new update rule called forward-view TD(λ):

$$V(s_t) \leftarrow V(s_t) + \alpha\left(G_t^{\lambda} - V(s_t)\right)$$

Example: TD(λ)

Random walk with 19 states: performance of TD(λ) for different λ as a function of α



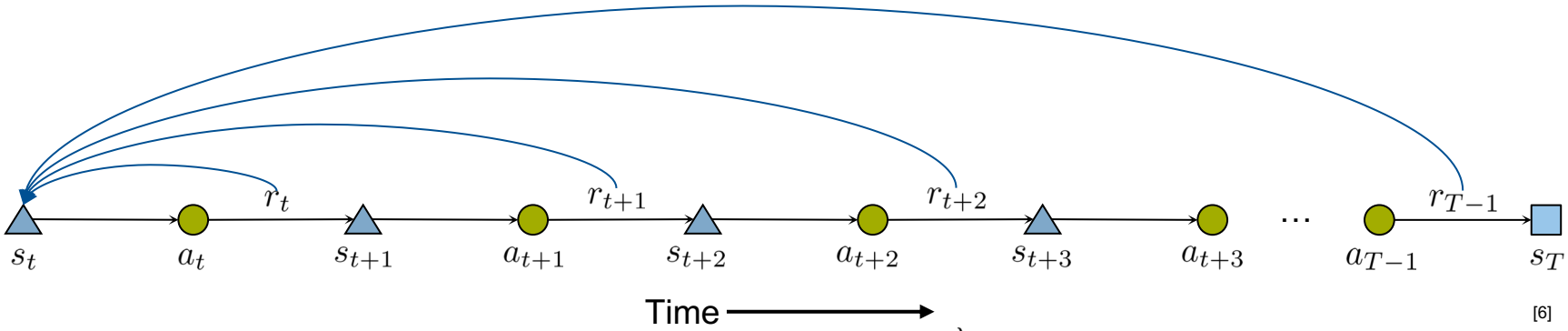
[6]

Again, intermediate values (now for λ) lead to the best performance

1 Root mean squared error between true and predicted value

Forward-view and backward-view TD(λ)

Forward-view TD(λ), as described so far:



[6]

- Forward-view TD(λ) looks into the future to compute λ-return G_t^λ which is required to update $V(s_t)$
- This means that updates can only be applied at the end of an episode, as in Monte Carlo methods
- Thus, forward-view TD(λ) suffers from some of the problems of MC, e.g., need for episodic environment

Forward-view provides theory. We however need a **mechanism to compute updates after every step, from incomplete trajectories („online“) – called backward-view TD(λ)**

Backward-view TD(λ): eligibility traces

Credit assignment problem

Example:

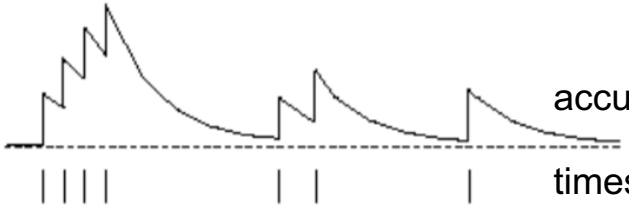
Did the bells or the light cause the shock?

[15]

Eligibility traces

- Frequency heuristic: assign credit to most frequent state
- Recency heuristic: assign credit to most recent state
- Eligibility traces combine both heuristics: $E_0(s) = 0 \quad , \quad E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbb{1}(s_t = s)$

  Visit of state s in t=1 yields $E_1(s) = \gamma\lambda E_0(s) + 1 = 1$

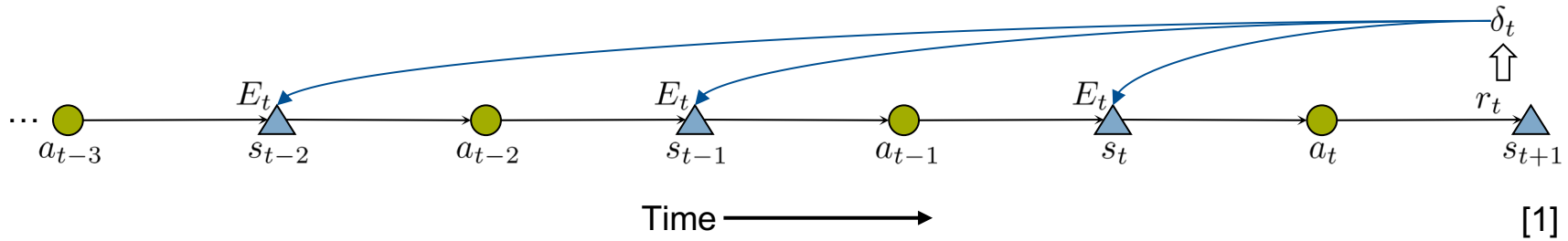


accumulating eligibility trace

[15]

times of visits to a state

Backward-view TD(λ)



[1]

- We keep an **eligibility trace for every state** (“how much did that state contribute to the current situation?”)
- After **each step**, we **update the value function estimate** $V(s)$ **for every state** s, weighting the TD error with the state’s eligibility trace:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$
$$V(s) \leftarrow V(s) + \alpha \delta_t E_t(s)$$

Advanced DRL Architectures

… REINFORCE & Actor Critic

… TD(λ), **TRPO/PPO**

Problem with actor-critic methods: fixing the learning rate (step size)

Selecting a fix learning rate can be very difficult (or even not a good idea at all), as it is not adapted to the terrain:

- Let us imagine a loss function with a shape like Half Dome (see picture)
- At the red spot, the gradient is large, such that a **large learning rate leads to a huge parameter update**
- At the yellow spot, the gradient is much smaller, such that **learning is very slow for a small learning rate** – however, if we take a large step in the direction of the arrow, we might also “fall off the cliff”
- Using the same learning rate everywhere can lead either to slow learning or to huge (potentially catastrophic) updates

[18]

Small updates to parameters can lead to huge (potentially undesirable) updates to the policy
➔ We Want to update parameters as much as possible (for **fast learning**) without changing the policy too much (to **avoid stepping too far**)

TRPO: KL-divergence constraint

KL-divergence constraint

- We want to **update the parameters** θ such that the **policy does not change too much**
- Therefore, we introduce a **constraint on the KL-divergence** between the old and the new policy:

$$D_{\mathrm{KL}}\left(\pi_{\theta_{\mathrm{old}}} || \pi_{\theta_{\mathrm{new}}}\right) \leq \delta^{\mathrm{TRPO}}$$

- The **hyperparameter** δ^{TRPO} determines the **size of the "trust region"** and must be chosen manually

TRPO

- Resulting **constrained optimization problem** defines Trust Region Policy Optimization (TRPO) [19]:

$$\max_{\theta_{\mathrm{new}}} \mathbb{E}_{s \sim \rho_{\pi_{\theta_{\mathrm{old}}}},\, a \sim \pi_{\theta_{\mathrm{old}}}} \left[\frac{\pi_{\theta_{\mathrm{new}}}(s,a)}{\pi_{\theta_{\mathrm{old}}}(s,a)} A_{\pi_{\theta_{\mathrm{old}}}}(s,a) \right]$$

$$\text{such that } D_{\mathrm{KL}}\left(\pi_{\theta_{\mathrm{old}}} || \pi_{\theta_{\mathrm{new}}}\right) \leq \delta^{\mathrm{TRPO}}$$

- As we are used to by now, the expectation can be **approximated by sampling** of states and actions
- The **KL-divergence term can be approximated** using an average over the KL-divergences for the states visited by the old policy

TRPO: properties

TRPO

- **Constrained optimization problem:**

$$\max_{\theta_{\text{new}}} \mathbb{E}_{s\sim\rho_{\pi_{\theta_{\text{old}}}},\, a\sim\pi_{\theta_{\text{old}}}} \left[\frac{\pi_{\theta_{\text{new}}}(s,a)}{\pi_{\theta_{\text{old}}}(s,a)} A_{\pi_{\theta_{\text{old}}}}(s,a)\right]$$
$$\text{such that } D_{\text{KL}}\left(\pi_{\theta_{\text{old}}} || \pi_{\theta_{\text{new}}}\right) \leq \delta^{\text{TRPO}}$$

- **Iteratively updating the policy parameters** (as well as the advantage function estimate) by solving the constrained optimization problem leads to a new actor-critic algorithm

Properties

- It can be shown that each iteration brings the policy closer to an optimum (“**monotonic improvement**”)
- **Parameter updates may be large**, as long as the **policy remains within the trust region** δ^{TRPO}. The **optimal policy within the trust region** is found in each iteration

PPO: overcoming the problems of TRPO while keeping the benefits

- We want to find an **easier algorithm** with a **flexible choice of the NN** architectures we can use with it
- We still have the **same motivation as for TRPO**: make large steps without accidently collapsing the performance of the policy
- This led to **Proximal Policy Optimization** (PPO) [20]. There are **two main versions of PPO**:

PPO-Penalty

- **Penalizes KL-divergence** in the optimization objective instead of having a constraint
- **Trade-off** between the actual objective and the penalty term controlled by a **coefficient that is automatically adjusted** during training

PPO-Clip

- **No KL-divergence** term at all
- Introduces **clipping** in the objective function, such that there is **no incentive to move the new policy far from the old one**

We **focus on PPO-Clip**, which is usually **used in practice**

PPO-Clip: clipping in the objective function

Objective in TRPO

In TRPO, the term in the expectation that is maximized is

$$L(s,a,\theta_{\text{old}},\theta_{\text{new}}) := \frac{\pi_{\theta_{\text{new}}}(s,a)}{\pi_{\theta_{\text{old}}}(s,a)} \cdot A_{\pi_{\theta_{\text{old}}}}(s,a) = h(s,a,\theta_{\text{old}},\theta_{\text{new}}) \cdot A_{\pi_{\theta_{\text{old}}}}(s,a) \quad \text{, with} \quad h(s,a,\theta_{\text{old}},\theta_{\text{new}}) := \frac{\pi_{\theta_{\text{new}}}(s,a)}{\pi_{\theta_{\text{old}}}(s,a)}$$

Objective in PPO-Clip

- We adapt this to

$$L^{\text{CLIP}}(s,a,\theta_{\text{old}},\theta_{\text{new}}) := \min\Big(h(s,a,\theta_{\text{old}},\theta_{\text{new}}) \cdot A_{\pi_{\theta_{\text{old}}}}(s,a),\ \text{clip}\left(h(s,a,\theta_{\text{old}},\theta_{\text{new}}), 1-\epsilon, 1+\epsilon\right) \cdot A_{\pi_{\theta_{\text{old}}}}(s,a)\Big)$$

- For $x, y, z \in \mathbb{R}$, the function $\text{clip}(x,y,z) = \begin{cases} y & \text{if } x < y \\ x & \text{if } y \leq x \leq z \\ z & \text{if } x > z \end{cases}$ cuts off x at the lower bound y and the upper bound z
- Thereby, we ensure that a **"large" policy update** that would lead to h being far away from 1 has **no effect on the objective function** and is therefore not encouraged (see details on next page)
- The **hyperparameter ϵ controls how different the new policy can be** from the old one

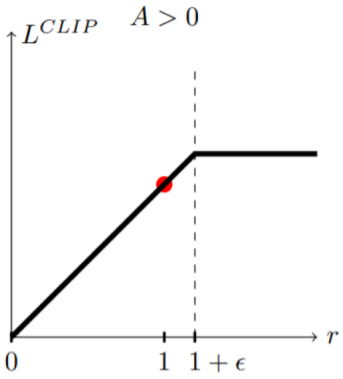
- The **KL-divergence constraint is not needed anymore**, which greatly **simplifies implementation** (no more FIM, etc.)
- Drawback: **convergence guarantee is lost** (but the algorithm **works well in practice**)

PPO-Clip: the effect of clipping

Case 1: positive advantage

- Objective term from previous page reduces to

$$L^{\text{CLIP}} = \min\left(\frac{\pi_{\theta_{\text{new}}}(s,a)}{\pi_{\theta_{\text{old}}}(s,a)},\ 1+\epsilon\right)\cdot A_{\pi_{\theta_{\text{old}}}}(s,a)$$



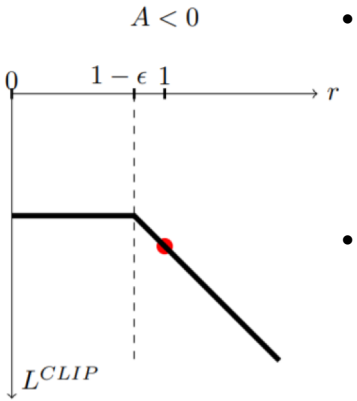
[20]

- Since the advantage is positive, maximizing the objective means to maximize $\pi_{\theta_{\text{new}}}$
- Due to the min, increasing the ratio of old and new policy h beyond $1+\epsilon$ does not change the objective

Case 2: negative advantage

- Objective term from previous page reduces to

$$L^{\text{CLIP}} = \max\left(\frac{\pi_{\theta_{\text{new}}}(s,a)}{\pi_{\theta_{\text{old}}}(s,a)},\ 1-\epsilon\right)\cdot A_{\pi_{\theta_{\text{old}}}}(s,a)$$



[20]

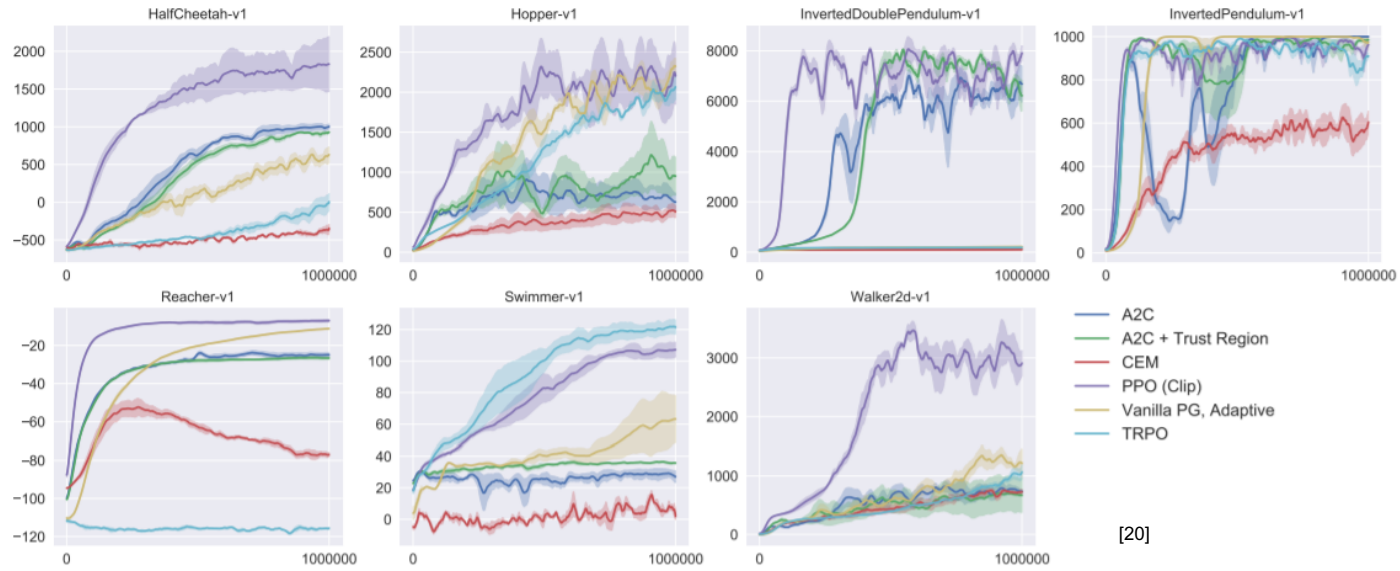
- Since the advantage is negative, maximizing the objective means to minimize $\pi_{\theta_{\text{new}}}$
- Due to the max, decreasing the ratio of old and new policy h below $1-\epsilon$ does not change the objective

Moving the new policy too far from the old one does not have a positive effect on the objective[1]

1 We clip “only on one side” since “moving into the other direction” is not encouraged by the basic maximization problem

TRPO and PPO: performance

Comparison of TRPO and PPO-Clip with other algorithms on several MuJoCo environments:



[20]

Videos with robots trained by PPO: openai.com/blog/openai-baselines-ppo/

PPO performs at least as well as TRPO, despite being much simpler

Conclusion

Key takeaways

- n-step TD-learning and TD(λ) give us control over the bias/variance trade-off; eligibility traces allow to use TD(λ) to compute updates from incomplete trajectories
- **TRPO** solves a **KL-divergence constrained optimization problem** to make large parameter update steps without risking a large loss of policy performance, since the new policy cannot be too different from the old one (it remains within the “trust region”)
- **PPO-Clip** introduces **clipping in the objective function** to get an algorithm with the same positive properties as TRPO while being less complex (since the KL-divergence constraint is removed) and better applicable in practice