

# Reinforcement Learning

## Lecture 1: Introduction

Hado van Hasselt  
Senior Staff Research Scientist, DeepMind

Reinforcement learning  
2021



# Lecturers



Diana Borsa



Matteo Hessel



# Background material

## **Background material**

Reinforcement Learning: An Introduction, Sutton & Barto 2018

<http://incompleteideas.net/book/the-book-2nd.html>



## Admin for UCL students

- ▶ Check Moodle for updates
- ▶ Use Moodle for questions
- ▶ Grading: assignments



# About this course



# What is reinforcement learning?



# Artificial Intelligence



# Motivation

- ▶ First, automation of repeated physical solutions
  - ▶ **Industrial revolution** (1750 - 1850) and Machine Age (1870 - 1940)
- ▶ Second, automation of repeated mental solutions
  - ▶ **Digital revolution** (1950 - now) and Information Age
- ▶ Next step: allow machines to find solutions themselves
  - ▶ **Artificial Intelligence**
- ▶ We then only needs to specify a problem and/or goal
- ▶ This requires learning autonomously how to make decisions



Can machines think?

## I.—COMPUTING MACHINERY AND INTELLIGENCE

BY A. M. TURING

### 1. *The Imitation Game.*

I PROPOSE to consider the question, ‘Can machines think ?’

– Alan Turing, 1950



*In the process of trying to imitate an adult human mind we are bound to think a good deal about the process which has brought it to the state that it is in. We may notice three components,*

- a. *The initial state of the mind, say at birth,*
- b. *The education to which it has been subjected,*
- c. *Other experience, not to be described as education, to which it has been subjected.*

*Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain. Presumably the child-brain is something like a note-book as one buys it from the stationers. Rather little mechanism, and lots of blank sheets. (Mechanism and writing are from our point of view almost synonymous.) Our hope is that there is so little mechanism in the child-brain that something like it can be easily programmed.*

– Alan Turing, 1950



# What is artificial intelligence?

- ▶ We will use the following definition of intelligence:

*To be able to learn to make decisions to achieve goals*

- ▶ Learning, decisions, and goals are all central



# What is Reinforcement Learning?

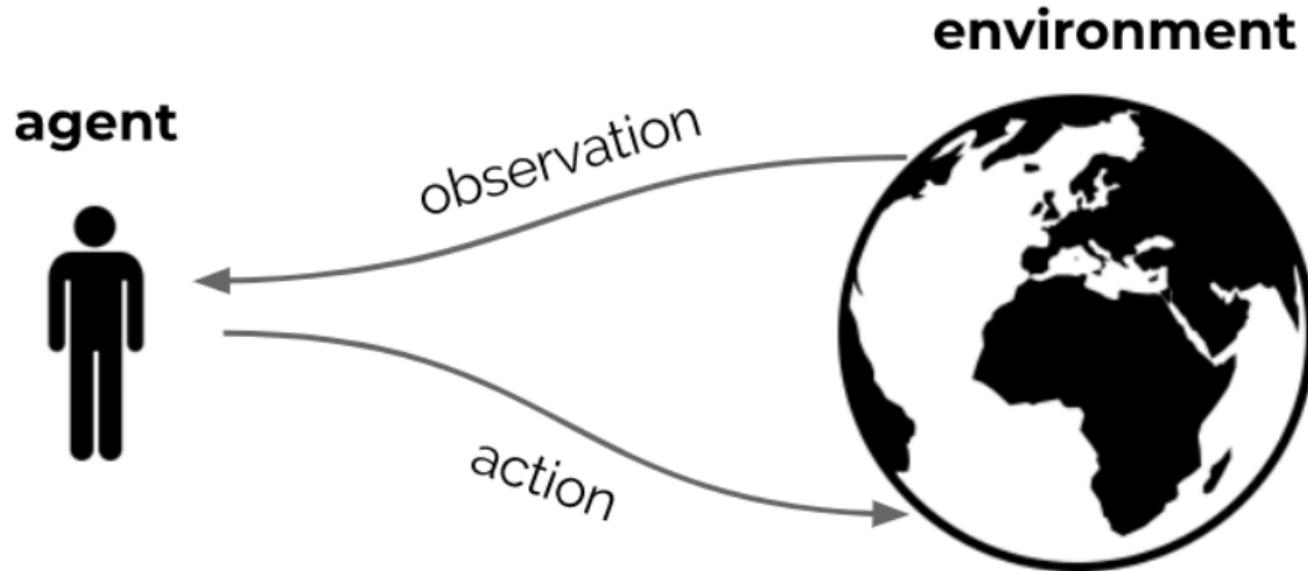


# What is reinforcement learning?

- ▶ People and animals learn by **interacting with our environment**
- ▶ This differs from certain other types of learning
  - ▶ It is **active** rather than passive
  - ▶ Interactions are often **sequential** — future interactions can depend on earlier ones
- ▶ We are **goal-directed**
- ▶ We can learn **without examples** of optimal behaviour
- ▶ Instead, we optimise some **reward signal**



# The interaction loop



**Goal:** optimise sum of rewards, through repeated interaction



# The reward hypothesis

Reinforcement learning is based on the **reward hypothesis**:

*Any goal can be formalized as the outcome of maximizing a cumulative reward*



# Examples of RL problems

- ▶ Fly a helicopter → Reward: air time, inverse distance, ...
- ▶ Manage an investment portfolio → Reward: gains, gains minus risk, ...
- ▶ Control a power station → Reward: efficiency, ...
- ▶ Make a robot walk → Reward: distance, speed, ...
- ▶ Play video or board games → Reward: win, maximise score, ...

If the goal is to learn via interaction, these are all reinforcement learning problems  
(Irrespective of which solution you use)



# What is reinforcement learning?

There are distinct reasons to learn:

## 1. Find **solutions**

- ▶ A program that plays chess really well
- ▶ A manufacturing robot with a specific purpose

## 2. **Adapt online**, deal with unforeseen circumstances

- ▶ A chess program that can learn to adapt to you
- ▶ A robot that can learn to navigate unknown terrains
- ▶ Reinforcement learning can provide algorithms for both cases
- ▶ Note that the second point is not (just) about generalization — it is about continuing to learn efficiently online, during operation



# What is reinforcement learning?

- ▶ Science and framework of **learning to make decisions** from **interaction**
- ▶ This requires us to think about
  - ▶ ...time
  - ▶ ...long-term consequences of actions
  - ▶ ...actively gathering experience
  - ▶ ...predicting the future
  - ▶ ...dealing with uncertainty
- ▶ Huge potential scope
- ▶ A formalisation of the AI problem



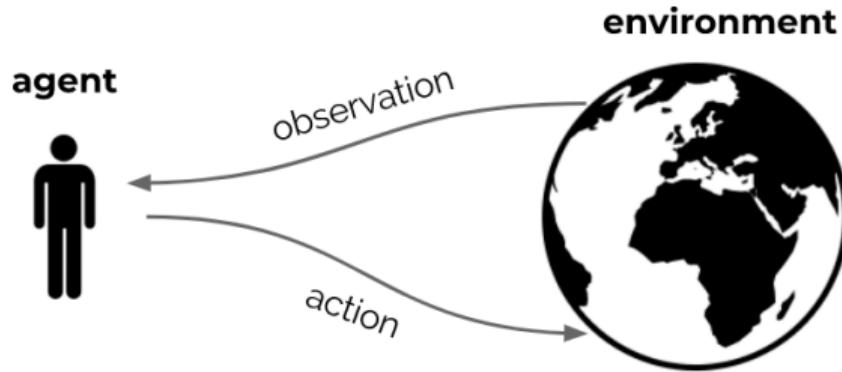
# Example: Atari



# Formalising the RL Problem



# Agent and Environment



- ▶ At each step  $t$  the agent:
  - ▶ Receives observation  $O_t$  (and reward  $R_t$ )
  - ▶ Executes action  $A_t$
- ▶ The environment:
  - ▶ Receives action  $A_t$
  - ▶ Emits observation  $O_{t+1}$  (and reward  $R_{t+1}$ )



# Rewards

- ▶ A **reward**  $R_t$  is a scalar feedback signal
- ▶ Indicates how well agent is doing at step  $t$  — defines the goal
- ▶ The agent's job is to maximize cumulative reward

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots$$

- ▶ We call this the **return**

Reinforcement learning is based on the **reward hypothesis**:

*Any goal can be formalized as the outcome of maximizing a cumulative reward*



# Values

- ▶ We call the expected cumulative reward, from a state  $s$ , the **value**

$$\begin{aligned}v(s) &= \mathbb{E}[G_t \mid S_t = s] \\&= \mathbb{E}[R_{t+1} + R_{t+2} + R_{t+3} + \dots \mid S_t = s]\end{aligned}$$

- ▶ The value depends on the actions the agent takes
- ▶ Goal is to **maximize value**, by picking suitable actions
- ▶ Rewards and values define **utility** of states and action (no supervised feedback)
- ▶ Returns and values can be defined recursively

$$\begin{aligned}G_t &= R_{t+1} + G_{t+1} \\v(s) &= \mathbb{E}[R_{t+1} + v(S_{t+1}) \mid S_t = s]\end{aligned}$$



# Maximising value by taking actions

- ▶ Goal: **select actions to maximise value**
- ▶ Actions may have long term consequences
- ▶ Reward may be delayed
- ▶ It may be better to sacrifice immediate reward to gain more long-term reward
- ▶ Examples:
  - ▶ Refueling a helicopter (might prevent a crash in several hours)
  - ▶ Defensive moves in a game (may help chances of winning later)
  - ▶ Learning a new skill (can be costly & time-consuming at first)
- ▶ A mapping from states to actions is called a **policy**



## Action values

- ▶ It is also possible to condition the value on **actions**:

$$\begin{aligned} q(s, a) &= \mathbb{E}[G_t \mid S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + R_{t+2} + R_{t+3} + \dots \mid S_t = s, A_t = a] \end{aligned}$$

- ▶ We will talk in depth about state and action values later



# Core concepts

The reinforcement learning formalism includes

- ▶ **Environment** (dynamics of the problem)
- ▶ **Reward** signal (specifies the goal)
- ▶ **Agent**, containing:
  - ▶ Agent state
  - ▶ Policy
  - ▶ Value function estimate?
  - ▶ Model?
- ▶ We will now go into the agent



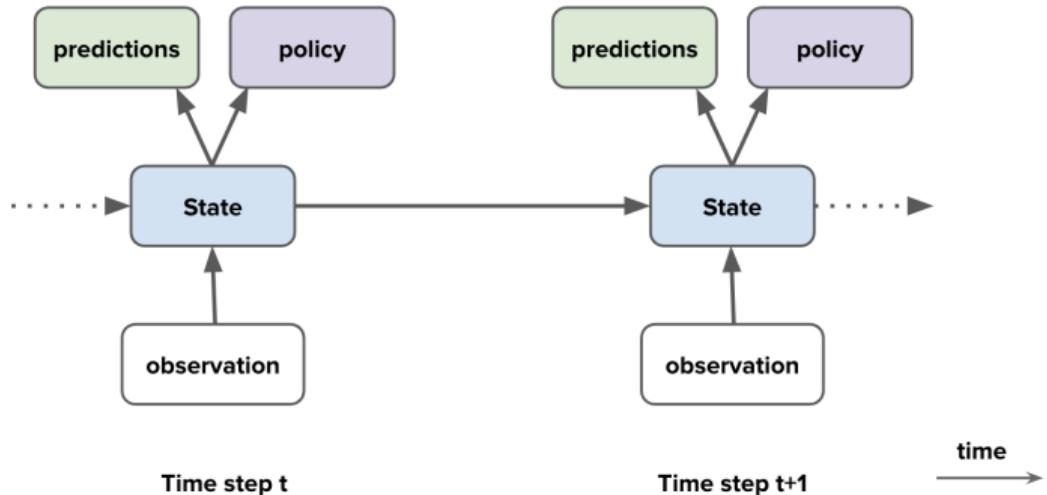
# Inside the Agent: the Agent State



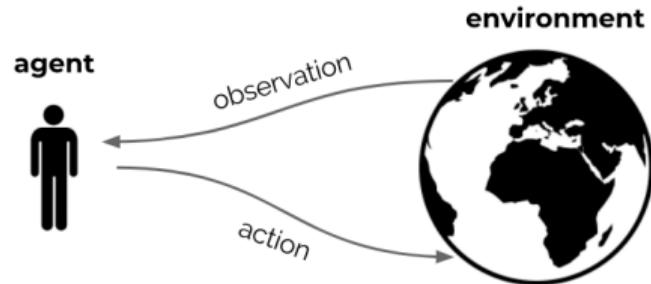
# Agent components

## Agent components

- ▶ Agent state
- ▶ Policy
- ▶ Value functions
- ▶ Model



# Environment State



- ▶ The **environment state** is the environment's internal state
- ▶ It is usually invisible to the agent
- ▶ Even if it is visible, it may contain lots of irrelevant information



## Agent State

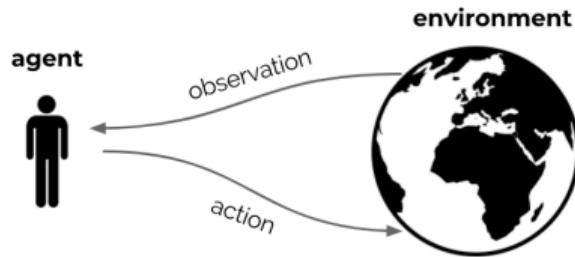
- ▶ The **history** is the full sequence of observations, actions, rewards

$$\mathcal{H}_t = O_0, A_0, R_1, O_1, \dots, O_{t-1}, A_{t-1}, R_t, O_t$$

- ▶ For instance, the sensorimotor stream of a robot
- ▶ This history is used to construct the **agent state**  $S_t$



# Fully Observable Environments



## Full observability

Suppose the agent sees the full environment state

- ▶ observation = environment state
- ▶ The agent state could just be this observation:

$$S_t = O_t = \text{environment state}$$



# Markov decision processes

**Markov decision processes** (MDPs) are a useful mathematical framework

## Definition

A decision process is Markov if

$$p(r, s | S_t, A_t) = p(r, s | \mathcal{H}_t, A_t)$$

- ▶ This means that the state contains all we need to know from the history
- ▶ Doesn't mean it contains everything, just that adding more history doesn't help
- ▶  $\implies$  Once the state is known, the history may be thrown away
  - ▶ The full environment + agent state is Markov (but large)
  - ▶ The full history  $\mathcal{H}_t$  is Markov (but keeps growing)
- ▶ Typically, the agent state  $S_t$  is some compression of  $\mathcal{H}_t$
- ▶ Note: we use  $S_t$  to denote the **agent state**, not the **environment state**

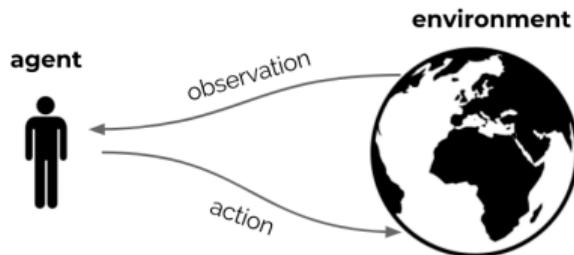


# Partially Observable Environments

- ▶ **Partial observability:** The observations are not Markovian
  - ▶ A robot with camera vision isn't told its absolute location
  - ▶ A poker playing agent only observes public cards
- ▶ Now using the observation as state would not be Markovian
- ▶ This is called a **partially observable Markov decision process** (POMDP)
- ▶ The **environment state** can still be Markov, but the agent does not know it
- ▶ We might still be able to construct a Markov agent state



# Agent State



- ▶ The agent's actions depend on its state
- ▶ The **agent state** is a function of the history
- ▶ For instance,  $S_t = O_t$
- ▶ More generally:

$$S_{t+1} = u(S_t, A_t, R_{t+1}, O_{t+1})$$

where  $u$  is a ‘state update function’

- ▶ The agent state is often **much** smaller than the environment state



## Agent State

The full environment state of a maze



# Agent State

A potential observation



# Agent State

An observation in a different location



## Agent State

The two observations are indistinguishable



## Agent State

These two states are not Markov



How could you construct a Markov agent state in this maze (for any reward signal)?



# Partially Observable Environments

- ▶ To deal with partial observability, agent can construct suitable state representations
- ▶ Examples of agent states:
  - ▶ Last observation:  $S_t = O_t$  (might not be enough)
  - ▶ Complete history:  $S_t = \mathcal{H}_t$  (might be too large)
  - ▶ A generic update:  $S_t = u(S_{t-1}, A_{t-1}, R_t, O_t)$  (but how to pick/learn  $u$ ?)
- ▶ Constructing a fully Markovian agent state is often not feasible
- ▶ More importantly, the state should allow good policies and value predictions



# Inside the Agent: the Policy



# Agent components

## Agent components

- ▶ Agent state
- ▶ **Policy**
- ▶ Value function
- ▶ Model



# Policy

- ▶ A **policy** defines the agent's behaviour
- ▶ It is a map from agent state to action
- ▶ Deterministic policy:  $A = \pi(S)$
- ▶ Stochastic policy:  $\pi(A|S) = p(A|S)$



# Inside the Agent: Value Estimates



# Agent components

## Agent components

- ▶ Agent state
- ▶ Policy
- ▶ **Value function**
- ▶ Model



# Value Function

- ▶ The actual value function is the expected return

$$\begin{aligned}v_{\pi}(s) &= \mathbb{E}[G_t \mid S_t = s, \pi] \\&= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s, \pi]\end{aligned}$$

- ▶ We introduced a **discount factor**  $\gamma \in [0, 1]$ 
  - ▶ Trades off importance of immediate vs long-term rewards
- ▶ The value depends on a policy
- ▶ Can be used to evaluate the desirability of states
- ▶ Can be used to select between actions



## Value Functions

- ▶ The return has a recursive form  $G_t = R_{t+1} + \gamma G_{t+1}$
- ▶ Therefore, the value has as well

$$\begin{aligned}v_\pi(s) &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t \sim \pi(s)] \\&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t \sim \pi(s)]\end{aligned}$$

Here  $a \sim \pi(s)$  means  $a$  is chosen by policy  $\pi$  in state  $s$  (even if  $\pi$  is deterministic)

- ▶ This is known as a **Bellman equation** (Bellman 1957)
- ▶ A similar equation holds for the optimal (=highest possible) value:

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$

This does **not** depend on a policy

- ▶ We heavily exploit such equalities, and use them to create algorithms



# Value Function approximations

- ▶ Agents often approximate value functions
- ▶ We will discuss algorithms to learn these efficiently
- ▶ With an accurate value function, we can behave optimally
- ▶ With suitable approximations, we can behave well, even in intractably big domains



# Inside the Agent: Models



# Agent components

## Agent components

- ▶ Agent state
- ▶ Policy
- ▶ Value function
- ▶ **Model**



# Model

- ▶ A **model** predicts what the environment will do next
- ▶ E.g.,  $\mathcal{P}$  predicts the next state

$$\mathcal{P}(s, a, s') \approx p(S_{t+1} = s' \mid S_t = s, A_t = a)$$

- ▶ E.g.,  $\mathcal{R}$  predicts the next (immediate) reward

$$\mathcal{R}(s, a) \approx \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

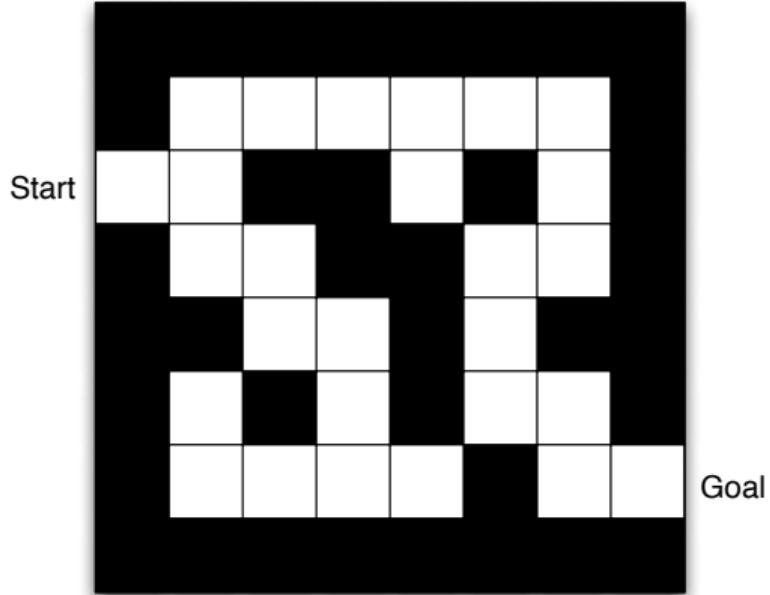
- ▶ A model does not immediately give us a good policy - we would still need to plan
- ▶ We could also consider **stochastic (generative)** models



# An Example



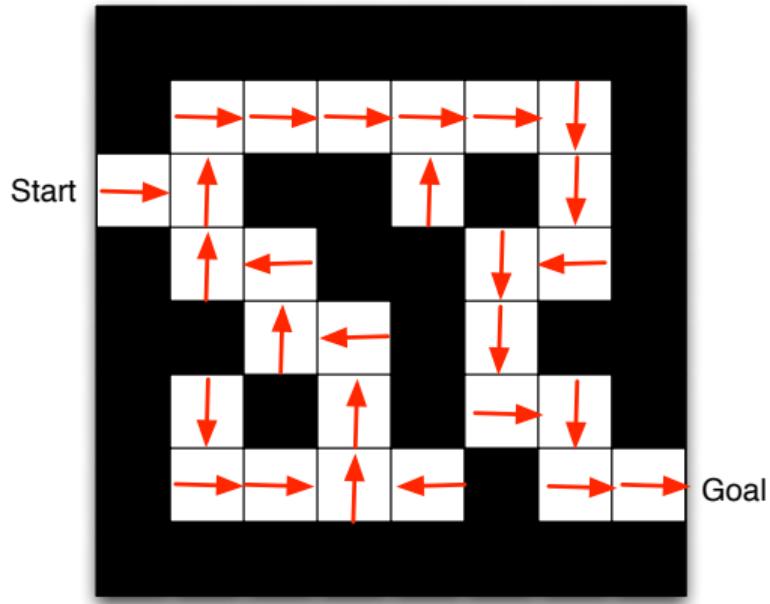
## Maze Example



- ▶ Rewards: -1 per time-step
- ▶ Actions: N, E, S, W
- ▶ States: Agent's location



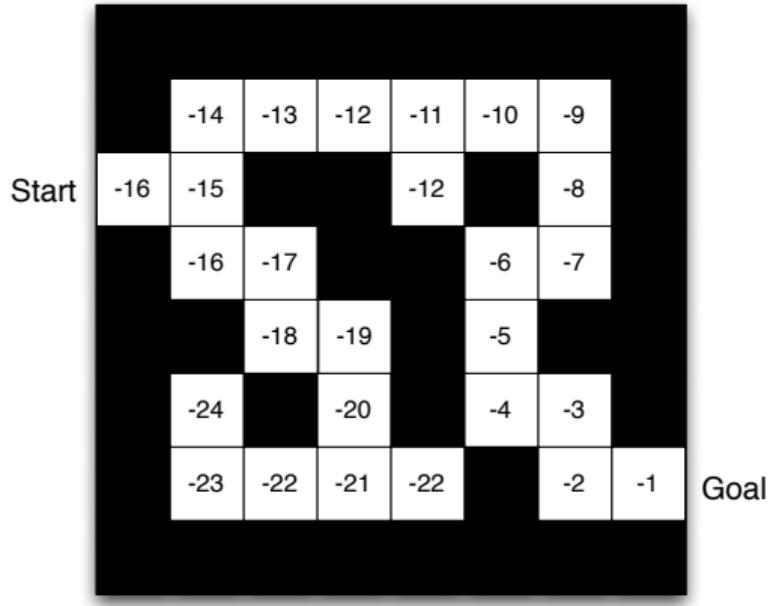
## Maze Example: Policy



- ▶ Arrows represent policy  $\pi(s)$  for each state  $s$



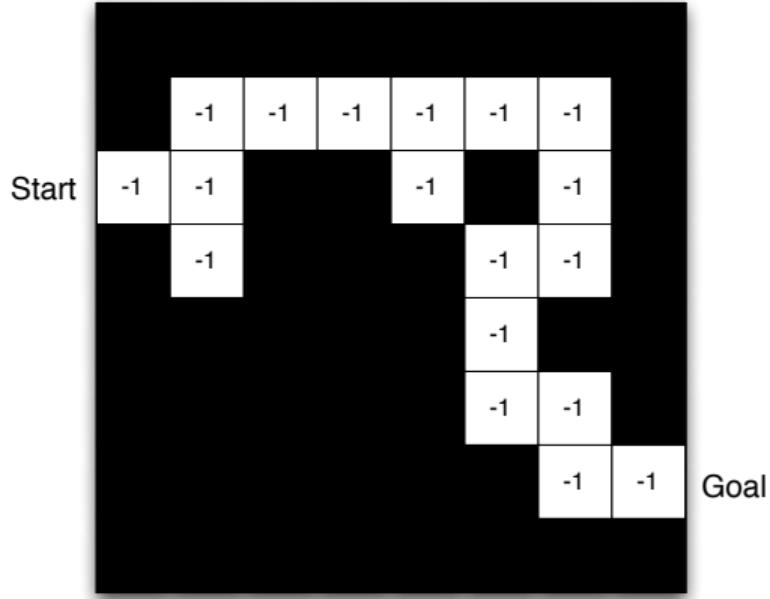
## Maze Example: Value Function



- ▶ Numbers represent value  $v_\pi(s)$  of each state  $s$



## Maze Example: Model



- ▶ Grid layout represents partial transition model  $\mathcal{P}_{ss'}^a$
- ▶ Numbers represent immediate reward  $\mathcal{R}_{ss'}^a$ , from each state  $s$  (same for all  $a$  and  $s'$  in this case)



# Agent Categories



# Agent Categories

- ▶ Value Based
  - ▶ No Policy (Implicit)
  - ▶ Value Function
- ▶ Policy Based
  - ▶ Policy
  - ▶ No Value Function
- ▶ Actor Critic
  - ▶ Policy
  - ▶ Value Function



# Agent Categories

- ▶ Model Free
  - ▶ Policy and/or Value Function
  - ▶ No Model
- ▶ Model Based
  - ▶ Optionally Policy and/or Value Function
  - ▶ Model



# Subproblems of the RL Problem



# Prediction and Control

- ▶ **Prediction**: evaluate the future (for a given policy)
- ▶ **Control**: optimise the future (find the best policy)
- ▶ These can be strongly related:

$$\pi_*(s) = \operatorname{argmax}_{\pi} v_{\pi}(s)$$

- ▶ If we could predict **everything** do we need anything else?



# Learning and Planning

Two fundamental problems in reinforcement learning

- ▶ Learning:
  - ▶ The environment is initially unknown
  - ▶ The agent interacts with the environment
- ▶ Planning:
  - ▶ A model of the environment is given (or learnt)
  - ▶ The agent plans in this model (without external interaction)
  - ▶ a.k.a. reasoning, pondering, thought, search, planning



# Learning Agent Components

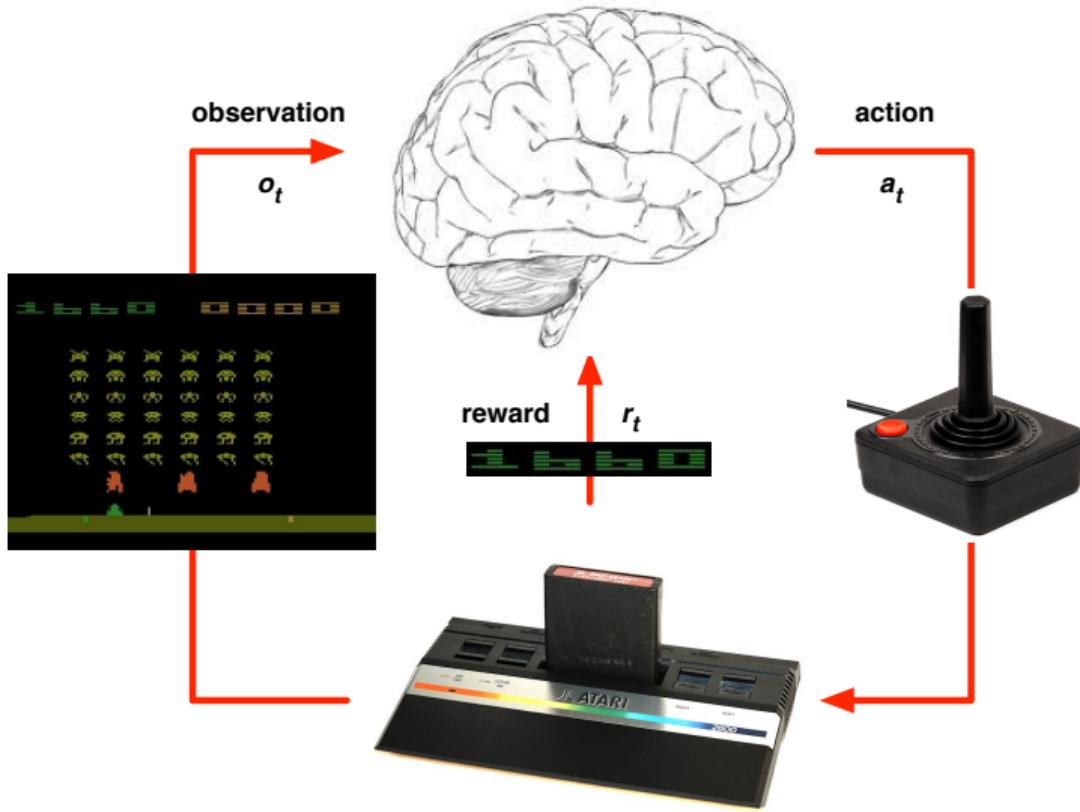
- ▶ All components are functions
  - ▶ Policies:  $\pi : \mathcal{S} \rightarrow \mathcal{A}$  (or to probabilities over  $\mathcal{A}$ )
  - ▶ Value functions:  $v : \mathcal{S} \rightarrow \mathbb{R}$
  - ▶ Models:  $m : \mathcal{S} \rightarrow \mathcal{S}$  and/or  $r : \mathcal{S} \rightarrow \mathbb{R}$
  - ▶ State update:  $u : \mathcal{S} \times \mathcal{O} \rightarrow \mathcal{S}$
- ▶ E.g., we can use neural networks, and use **deep learning** techniques to learn
- ▶ Take care: we do often violate assumptions from supervised learning (iid, stationarity)
- ▶ Deep learning is an important tool
- ▶ Deep reinforcement learning is a rich and active research field



# Examples



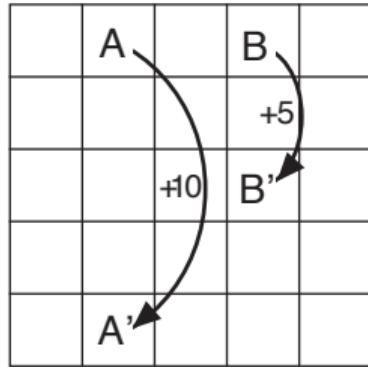
# Atari Example: Reinforcement Learning



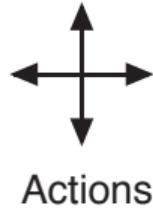
- ▶ Rules of the game are unknown
- ▶ Learn directly from interactive game-play
- ▶ Pick actions on joystick, see pixels and scores



## Gridworld Example: Prediction



(a)



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

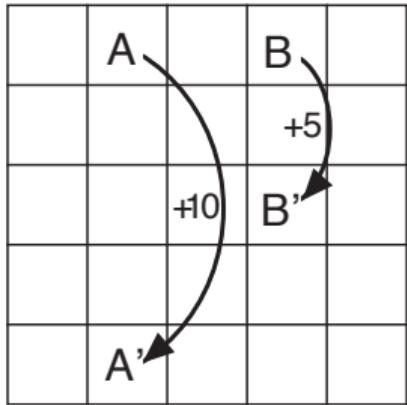
(b)

Reward is  $-1$  when bumping into a wall,  $\gamma = 0.9$

What is the value function for the uniform random policy?



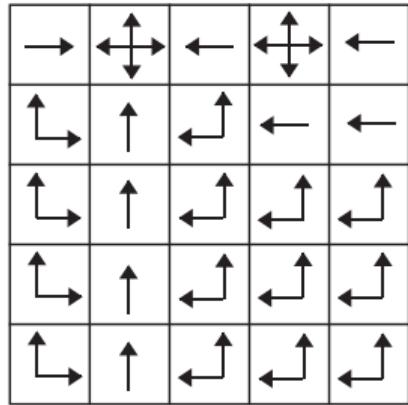
# Gridworld Example: Control



a) gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

b)  $V^*$



c)  $\pi^*$

What is the optimal value function over all possible policies?  
What is the optimal policy?



# Course

- ▶ In this course, we discuss how to learn by interaction
- ▶ The focus is on understanding core principles and learning algorithms

Topics include

- ▶ Exploration, in bandits and in sequential problems
- ▶ Markov decision processes, and planning by dynamic programming
- ▶ Model-free prediction and control (e.g., Q-learning)
- ▶ Policy-gradient methods
- ▶ Deep reinforcement learning
- ▶ Integrating learning and planning
- ▶ ...



# Example: Locomotion



# End of Lecture



# Lecture 2: Exploration and Exploitation

Hado van Hasselt

Reinforcement learning, 2021



# Background

Recommended reading:

Sutton & Barto 2018, Chapter 2

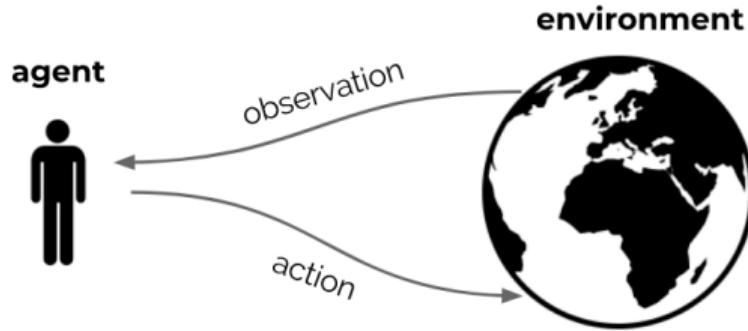
Further background material:

*Bandit Algorithms*, Lattimore & Szepesvári, 2020

*Finite-time analysis of the multiarmed bandit problem*, Auer, Cesa-Bianchi, Fischer, 2002



# Recap



- ▶ Reinforcement learning is the science of learning to make decisions
- ▶ Agents can learn a **policy**, **value function** and/or a **model**
- ▶ The general problem involves taking into account **time** and **consequences**
- ▶ Decisions affect the **reward**, the **agent state**, and **environment state**
- ▶ Learning is **active**: decisions impact data



# This Lecture

In this lecture, we simplify the setting

- ▶ The environment is assumed to have only a **single state**
- ▶ ⇒ actions no longer have long-term consequences in the environment
- ▶ ⇒ actions still do impact **immediate reward**
- ▶ ⇒ other observations can be ignored
- ▶ We discuss how to learn a policy in this setting



# Blackboard: Example



# Exploration vs. Exploitation

- ▶ Learning agents need to trade off two things
  - ▶ **Exploitation:** Maximise performance based on current knowledge
  - ▶ **Exploration:** Increase knowledge
- ▶ We need to gather information to make the best overall decisions
- ▶ The best long-term strategy may involve short-term sacrifices



# Formalising the problem



# The Multi-Armed Bandit

- ▶ A multi-armed bandit is a set of distributions  $\{\mathcal{R}_a | a \in \mathcal{A}\}$
- ▶  $\mathcal{A}$  is a (known) set of actions (or “arms”)
- ▶  $\mathcal{R}_a$  is a distribution on rewards, given action  $a$
- ▶ At each step  $t$  the agent selects an action  $A_t \in \mathcal{A}$
- ▶ The environment generates a reward  $R_t \sim \mathcal{R}_{A_t}$
- ▶ The goal is to maximise cumulative reward  $\sum_{i=1}^t R_i$
- ▶ We do this by learning a **policy**: a distribution on  $\mathcal{A}$



# Values and Regret

- ▶ The **action value** for action  $a$  is the expected reward

$$q(a) = \mathbb{E}[R_t | A_t = a]$$

- ▶ The **optimal value** is

$$v_* = \max_{a \in \mathcal{A}} q(a) = \max_a \mathbb{E}[R_t | A_t = a]$$

- ▶ **Regret** of an action  $a$  is

$$\Delta_a = v_* - q(a)$$

- ▶ The regret for the optimal action is zero



# Regret

- ▶ We want to minimise **total regret**:

$$L_t = \sum_{n=1}^t v_* - q(A_n) = \sum_{n=1}^t \Delta_{A_n}$$

- ▶ Maximise cumulative reward  $\equiv$  minimise total regret
- ▶ The summation spans over the full ‘lifetime of learning’



# Algorithms



# Algorithms

- ▶ We will discuss several algorithms:
  - ▶ Greedy
  - ▶  $\epsilon$ -greedy
  - ▶ UCB
  - ▶ Thompson sampling
  - ▶ Policy gradients
- ▶ The first three all use **action value estimates**  $Q_t(a) \approx q(a)$



## Action values

- ▶ The **action value** for action  $a$  is the expected reward

$$q(a) = \mathbb{E}[R_t | A_t = a]$$

- ▶ A simple estimate is the average of the sampled rewards:

$$Q_t(a) = \frac{\sum_{n=1}^t \mathcal{I}(A_n = a) R_n}{\sum_{n=1}^t \mathcal{I}(A_n = a)}$$

$\mathcal{I}(\cdot)$  is the **indicator** function:  $\mathcal{I}(\text{True}) = 1$  and  $\mathcal{I}(\text{False}) = 0$

- ▶ The **count** for action  $a$  is

$$N_t(a) = \sum_{n=1}^t \mathcal{I}(A_n = a)$$



## Action values

- ▶ This can also be updated incrementally:

$$Q_t(A_t) = Q_{t-1}(A_t) + \underbrace{\alpha_t (R_t - Q_{t-1}(A_t))}_{\text{error}},$$

$$\forall a \neq A_t : Q_t(a) = Q_{t-1}(a)$$

with

$$\alpha_t = \frac{1}{N_t(A_t)} \quad \text{and} \quad N_t(A_t) = N_{t-1}(A_t) + 1,$$

where  $N_0(a) = 0$ .

- ▶ We will later consider other **step sizes**  $\alpha$
- ▶ For instance, constant  $\alpha$  would lead to **tracking**, rather than averaging



# Algorithms: greedy



## The greedy policy

- ▶ One of the simplest policies is **greedy**:
  - ▶ Select action with highest value:  $A_t = \operatorname{argmax}_a Q_t(a)$
  - ▶ Equivalently:  $\pi_t(a) = \mathcal{I}(A_t = \operatorname{argmax}_a Q_t(a))$  (assuming no ties are possible)



Example:  
Regret of the greedy policy





# Algorithms: $\epsilon$ -greedy



# $\epsilon$ -Greedy Algorithm

- ▶ Greedy can get stuck on a suboptimal action forever  
     $\implies$  linear expected total regret
- ▶ The  $\epsilon$ -greedy algorithm:
  - ▶ With probability  $1 - \epsilon$  select greedy action:  $a = \operatorname{argmax}_{a \in \mathcal{A}} Q_t(a)$
  - ▶ With probability  $\epsilon$  select a random action
  - ▶ Equivalently:

$$\pi_t(a) = \begin{cases} (1 - \epsilon) + \epsilon / |\mathcal{A}| & \text{if } Q_t(a) = \max_b Q_t(b) \\ \epsilon / |\mathcal{A}| & \text{otherwise} \end{cases}$$

- ▶  $\epsilon$ -greedy continues to explore  
 $\Rightarrow$   $\epsilon$ -greedy with constant  $\epsilon$  has linear expected total regret



# Algorithms: Policy gradients



## Policy search

- ▶ Can we learn policies  $\pi(a)$  directly, instead of learning values?
- ▶ For instance, define **action preferences**  $H_t(a)$  and a policy

$$\pi(a) = \frac{e^{H_t(a)}}{\sum_b e^{H_t(b)}} \quad (\text{softmax})$$

- ▶ The preferences are not values: they are just learnable policy parameters
- ▶ Goal: learn by optimising the preferences



# Policy gradients

- ▶ Idea: update policy parameters such that expected value increases
- ▶ We can use **gradient ascent**
- ▶ In the bandit case, we want to update:

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta} \mathbb{E}[R_t | \pi_{\theta_t}],$$

where  $\theta_t$  are the current policy parameters

- ▶ Can we compute this gradient?



## Gradient bandits

- Log-likelihood trick (also known as REINFORCE trick, Williams 1992):

$$\begin{aligned} &= q(a) \\ \nabla_{\theta} \mathbb{E}[R_t | \pi_{\theta}] &= \nabla_{\theta} \sum_a \pi_{\theta}(a) \overbrace{\mathbb{E}[R_t | A_t = a]} \\ &= \sum_a q(a) \nabla_{\theta} \pi_{\theta}(a) \\ &= \sum_a q(a) \frac{\pi_{\theta}(a)}{\pi_{\theta}(a)} \nabla_{\theta} \pi_{\theta}(a) \\ &= \sum_a \pi_{\theta}(a) q(a) \frac{\nabla_{\theta} \pi_{\theta}(a)}{\pi_{\theta}(a)} \\ &= \mathbb{E} \left[ R_t \frac{\nabla_{\theta} \pi_{\theta}(A_t)}{\pi_{\theta}(A_t)} \right] &= \mathbb{E} [R_t \nabla_{\theta} \log \pi_{\theta}(A_t)] \end{aligned}$$



# Gradient bandits

- ▶ Log-likelihood trick (also known as REINFORCE trick, Williams 1992):

$$\nabla_{\theta} \mathbb{E}[R_t | \theta] = \mathbb{E}[R_t \nabla_{\theta} \log \pi_{\theta}(A_t)]$$

- ▶ We can sample this!
- ▶ So

$$\theta = \theta + \alpha R_t \nabla_{\theta} \log \pi_{\theta}(A_t),$$

this is **stochastic gradient ascent** on the (true) value of the policy

- ▶ Can use **sampled** rewards — does not need value estimates



# Gradient bandits

- ▶ For soft max:

$$\begin{aligned} H_{t+1}(a) &= H_t(a) + \alpha R_t \frac{\partial \log \pi_t(A_t)}{\partial H_t(a)} \\ &= H_t(a) + \alpha R_t (\mathcal{I}(a = A_t) - \pi_t(a)) \end{aligned}$$

- ▶  $\Rightarrow$

$$\begin{aligned} H_{t+1}(A_t) &= H_t(A_t) + \alpha R_t (1 - \pi_t(A_t)) \\ H_{t+1}(a) &= H_t(a) - \alpha R_t \pi_t(a) && \text{if } a \neq A_t \end{aligned}$$

- ▶ Preferences for actions with higher rewards increase more (or decrease less), making them more likely to be selected again



# Theory: what is possible?



# How well can we do?

## Theorem (Lai and Robbins)

*Asymptotic total regret is at least logarithmic in number of steps*

$$\lim_{t \rightarrow \infty} L_t \geq \log t \sum_{a | \Delta_a > 0} \frac{\Delta_a}{KL(\mathcal{R}_a || \mathcal{R}_{a*})}$$

(Note:  $KL(\mathcal{R}_a || \mathcal{R}_{a*}) \propto \Delta_a^2$ )

- ▶ Note that **regret grows at least logarithmically**
- ▶ That's still a whole lot better than linear growth! Can we get it in practice?
- ▶ Are there algorithms for which the **upper bound** is logarithmic as well?



# Counting Regret

- ▶ Recall  $\Delta_a = v_* - q(a)$
- ▶ Total regret depends on action regrets  $\Delta_a$  and action counts

$$L_t = \sum_{n=1}^t \Delta_{A_n} = \sum_{a \in \mathcal{A}} N_t(a) \Delta_a$$

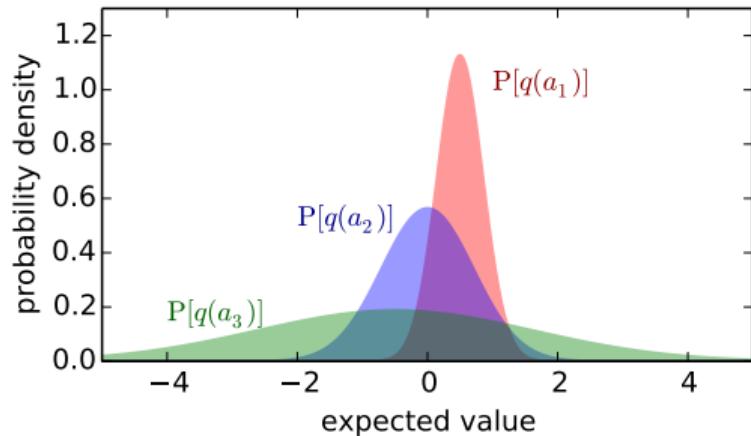
- ▶ A good algorithm ensures small counts for large action regrets



# Optimism in the face of uncertainty



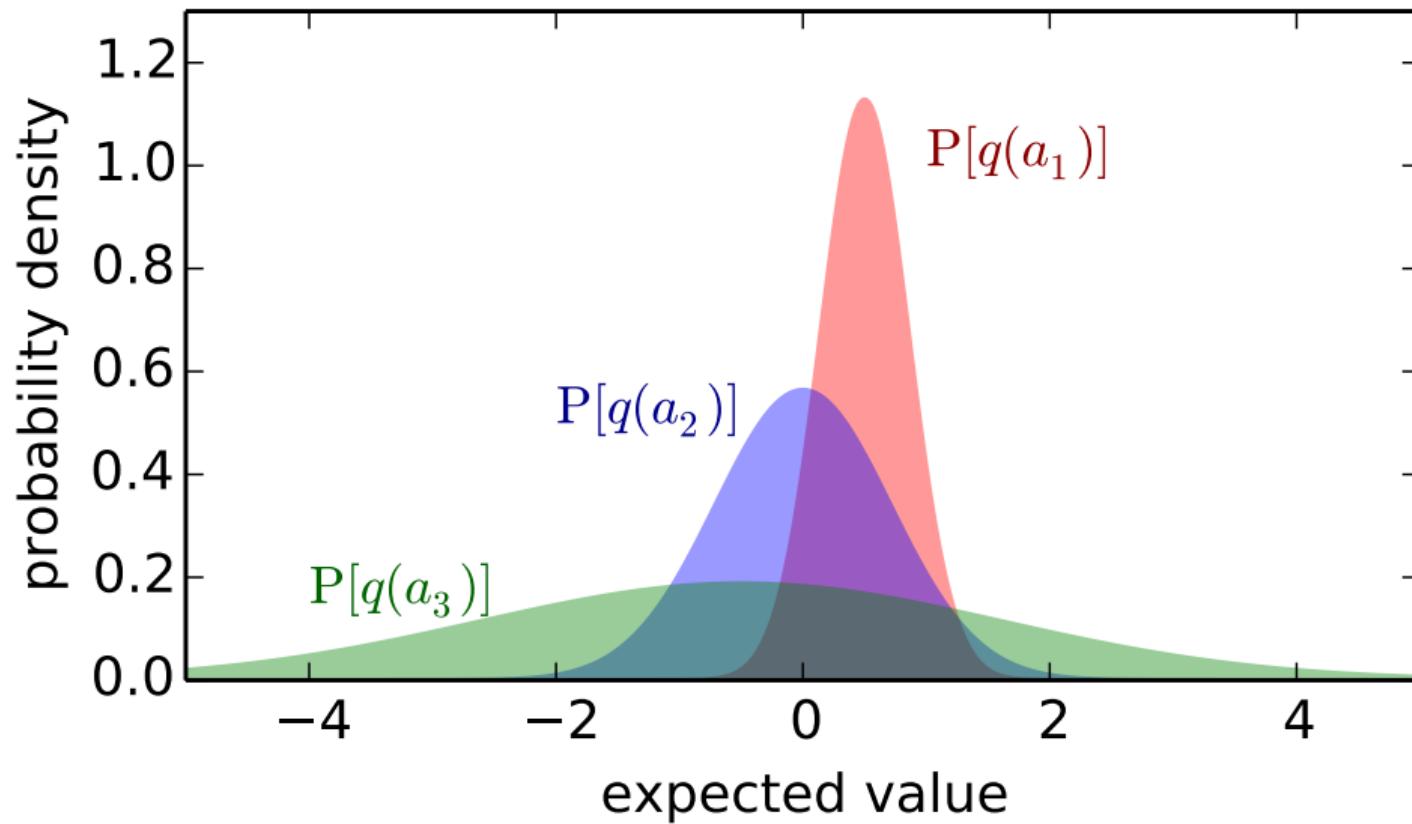
# Optimism in the Face of Uncertainty



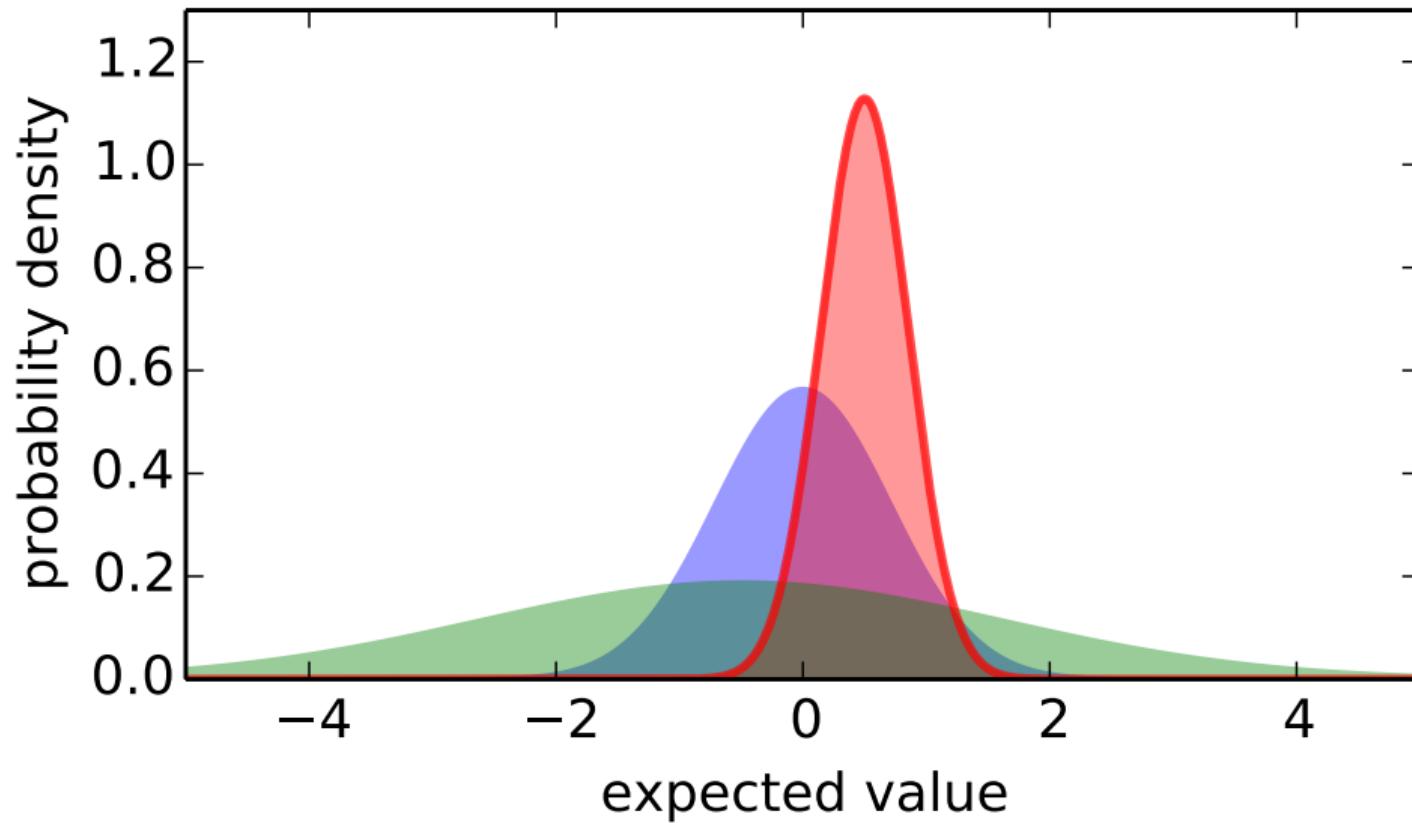
- ▶ Which action should we pick?
- ▶ More uncertainty about its value: more important to explore that action



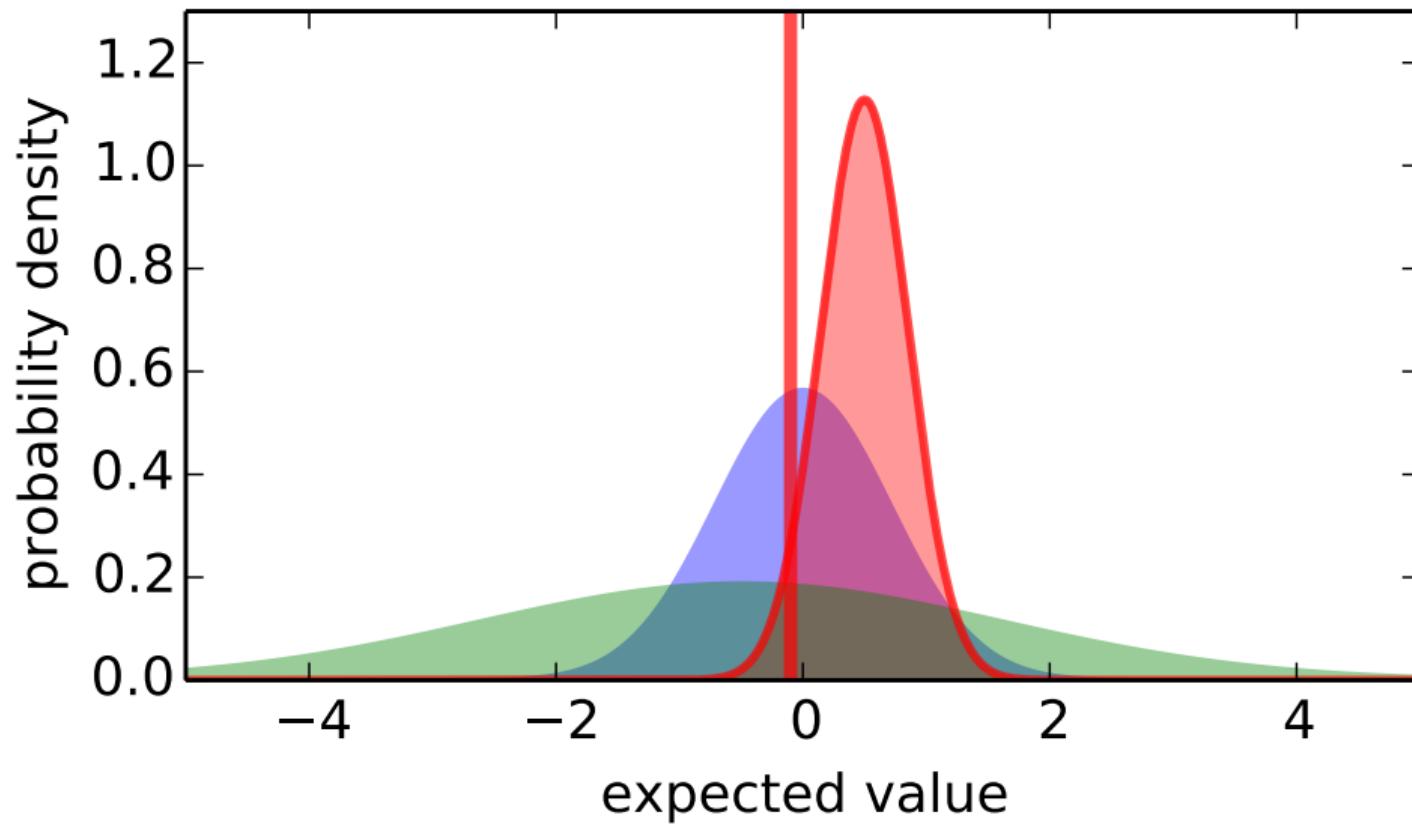
# Optimism in the Face of Uncertainty



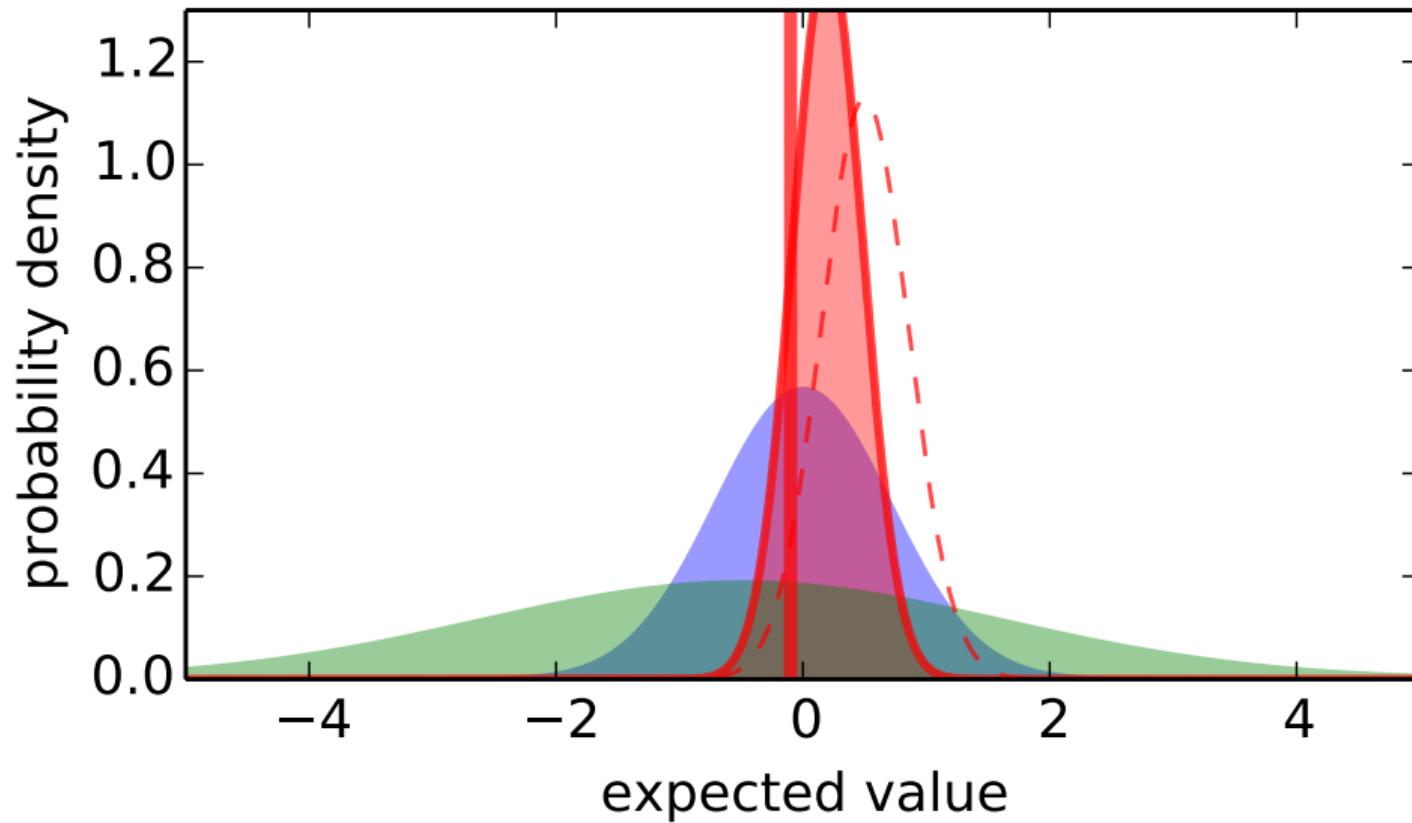
# Optimism in the Face of Uncertainty



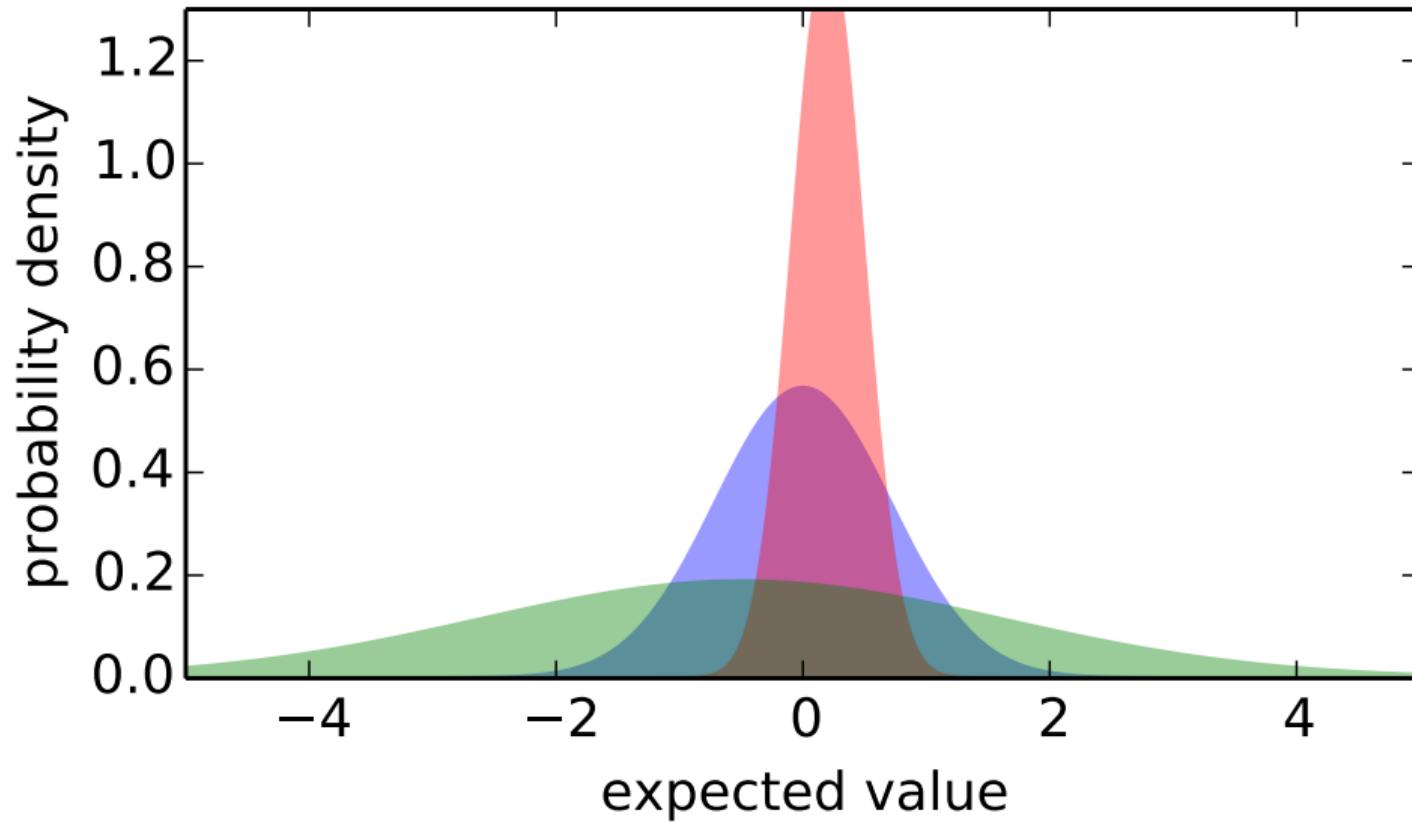
# Optimism in the Face of Uncertainty



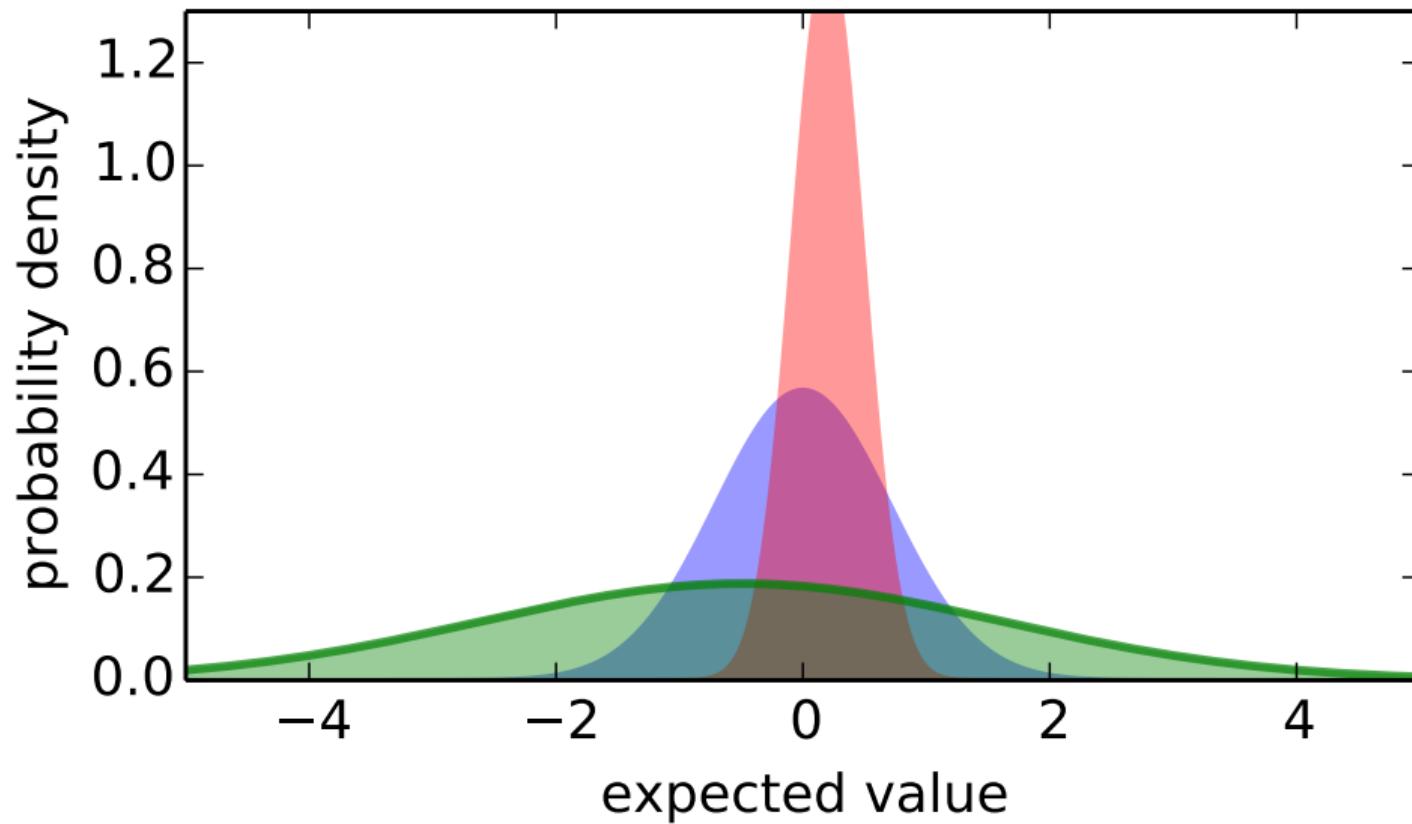
# Optimism in the Face of Uncertainty



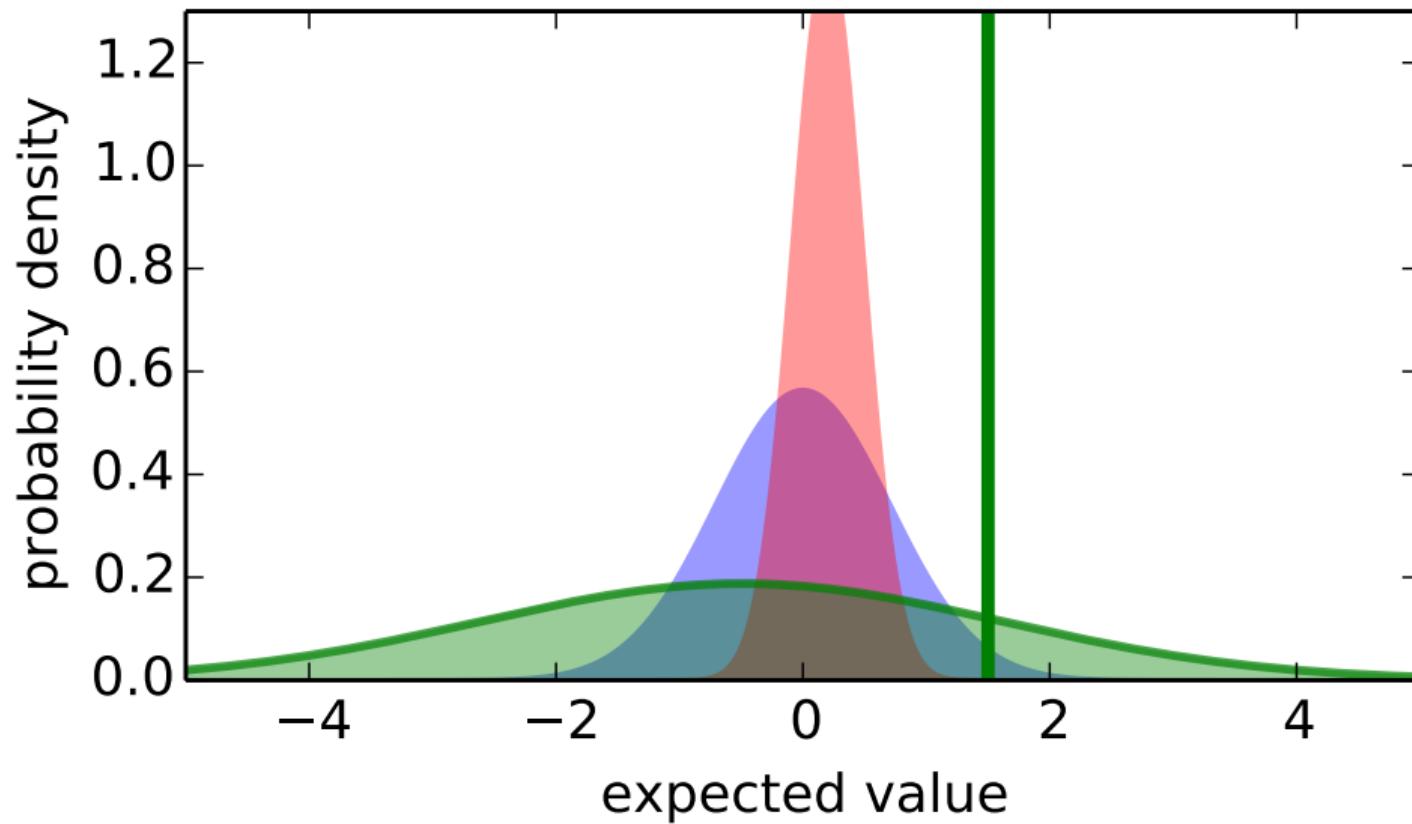
# Optimism in the Face of Uncertainty



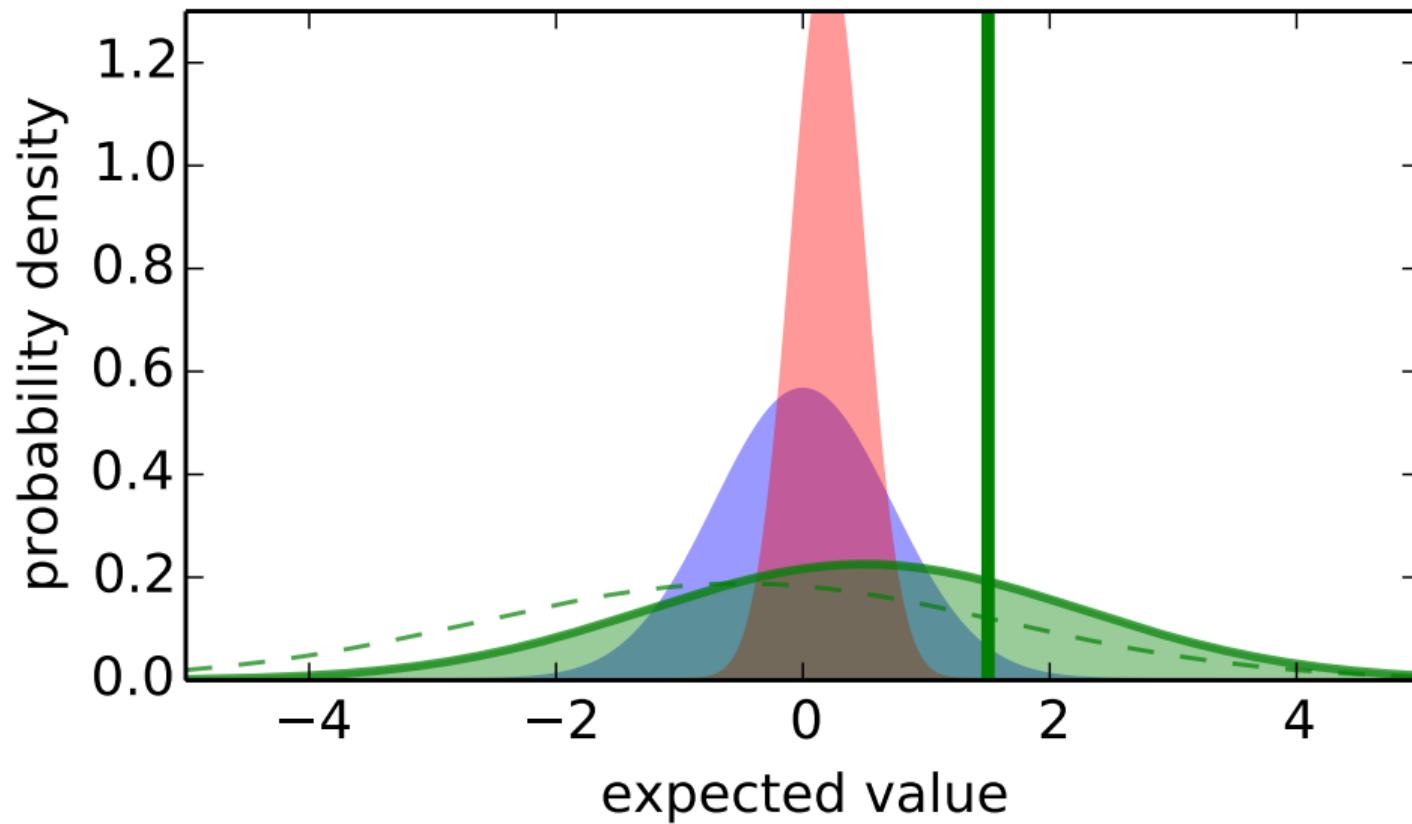
# Optimism in the Face of Uncertainty



# Optimism in the Face of Uncertainty



# Optimism in the Face of Uncertainty



# Algorithms: UCB



# Upper Confidence Bounds

- ▶ Estimate an upper confidence  $U_t(a)$  for each action value, such that  $q(a) \leq Q_t(a) + U_t(a)$  with high probability
- ▶ Select action maximizing **upper confidence bound** (UCB)

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q_t(a) + U_t(a)$$

- ▶ The uncertainty should depend on the number of times  $N_t(a)$  action  $a$  has been selected
  - ▶ Small  $N_t(a) \Rightarrow$  large  $U_t(a)$  (estimated value is uncertain)
  - ▶ Large  $N_t(a) \Rightarrow$  small  $U_t(a)$  (estimated value is accurate)
- ▶ Then  $a$  is only selected if either...
  - ▶ ... $Q_t(a)$  is large (=good action), or
  - ▶ ... $U_t(a)$  is large (=high uncertainty) (or both)
- ▶ Can we **derive** an optimal bound?



# Theory: the optimality of UCB



# Hoeffding's Inequality

## Theorem (Hoeffding's Inequality)

Let  $X_1, \dots, X_n$  be i.i.d. random variables in  $[0,1]$  with true mean  $\mu = \mathbb{E}[X]$ , and let  $\bar{X}_t = \frac{1}{n} \sum_{i=1}^n X_i$  be the sample mean. Then

$$p(\bar{X}_n + u \leq \mu) \leq e^{-2nu^2}$$

- ▶ We can apply Hoeffding's Inequality to bandits with bounded rewards
- ▶ If  $R_t \in [0, 1]$ , then

$$p(Q_t(a) + U_t(a) \leq q(a)) \leq e^{-2N_t(a)U_t(a)^2}$$

- ▶ By symmetry, we can also flip it around

$$p(Q_t(a) - U_t(a) \geq q(a)) \leq e^{-2N_t(a)U_t(a)^2}$$



## Calculating Upper Confidence Bounds

- We can pick a maximal desired probability  $p$  that the true value exceeds an upper bound and solve for this bound  $U_t(a)$

$$e^{-2N_t(a)U_t(a)^2} = p$$

$$\implies U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

We then know the probability that this happens is smaller than  $p$

- Idea: reduce  $p$  as we observe more rewards, e.g.,  $p = 1/t$

$$U_t(a) = \sqrt{\frac{\log t}{2N_t(a)}}$$

- This ensures that we always keep exploring, but not too much



## UCB

- ▶ UCB:

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q_t(a) + c \sqrt{\frac{\log t}{N_t(a)}}$$

- ▶ Intuition:
  - ▶ If  $\Delta_a$  is large, then  $N_t(a)$  is small, because  $Q_t(a)$  is likely to be small
  - ▶ So either  $\Delta_a$  is small or  $N_t(a)$  is small
  - ▶ In fact, we can prove  $\Delta_a N_t(a) \leq O(\log t)$ , for all  $a$



# UCB

- ▶ UCB:

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q_t(a) + c \sqrt{\frac{\log t}{N_t(a)}}$$

where  $c$  is a hyper-parameter

Theorem (Auer et al., 2002)

*UCB with  $c = \sqrt{2}$  achieves logarithmic expected total regret*

$$L_t \leq 8 \sum_{a | \Delta_a > 0} \frac{\log t}{\Delta_a} + O\left(\sum_a \Delta_a\right), \quad \forall t.$$



# Blackboard: UCB derivation





# Bayesian approaches



# Bayesian Bandits

- ▶ We could adopt **Bayesian** approach and model distributions over values  $p(q(a) | \theta_t)$
- ▶ This is interpreted as our **belief** that, e.g.,  $q(a) = x$  for all  $x \in \mathbb{R}$
- ▶ E.g.,  $\theta_t$  could contain the means and variances of Gaussian belief distributions
- ▶ Allows us to inject rich prior knowledge  $\theta_0$
- ▶ We can then use posterior belief to guide exploration



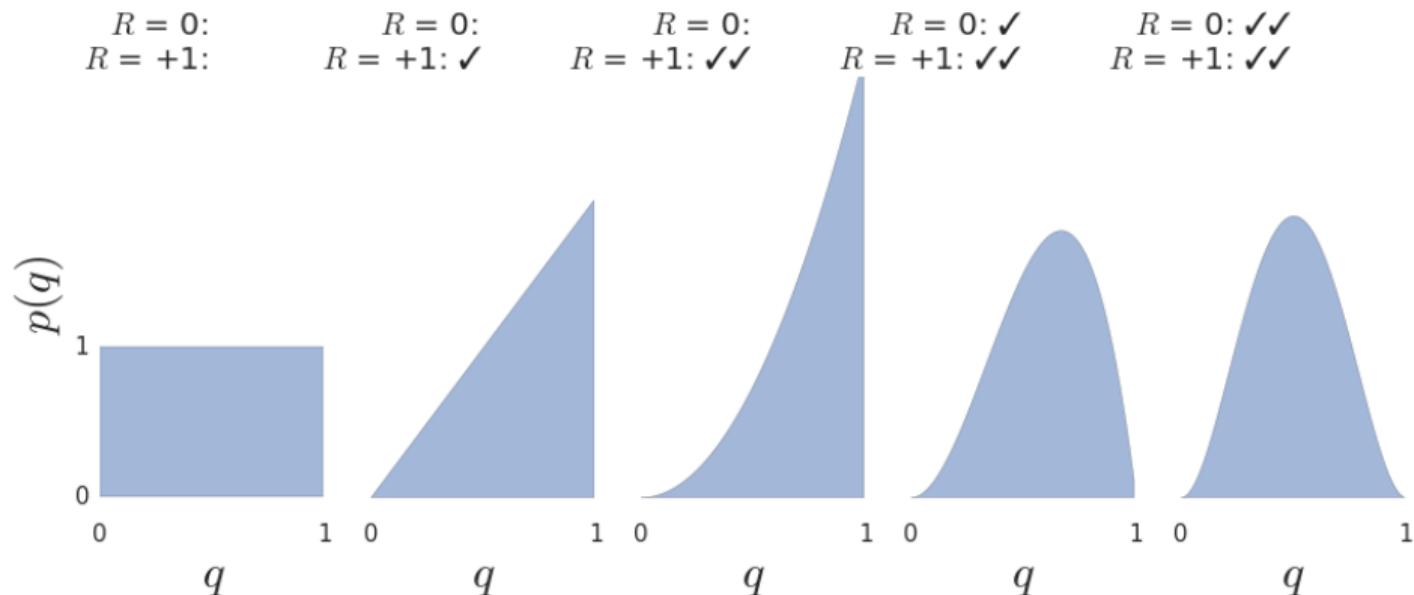
## Bayesian Bandits: Example

- ▶ Consider bandits with **Bernoulli** reward distribution: rewards are 0 or +1
- ▶ For each action, the prior could be a **uniform distribution** on [0, 1]
- ▶ This means we think each value in [0, 1] is equally likely
- ▶ The posterior is a Beta distribution  $\text{Beta}(x_a, y_a)$  with initial parameters  $x_a = 1$  and  $y_a = 1$  for each action  $a$
- ▶ Updating the posterior:
  - ▶  $x_{A_t} \leftarrow x_{A_t} + 1$  when  $R_t = 0$
  - ▶  $y_{A_t} \leftarrow y_{A_t} + 1$  when  $R_t = 1$

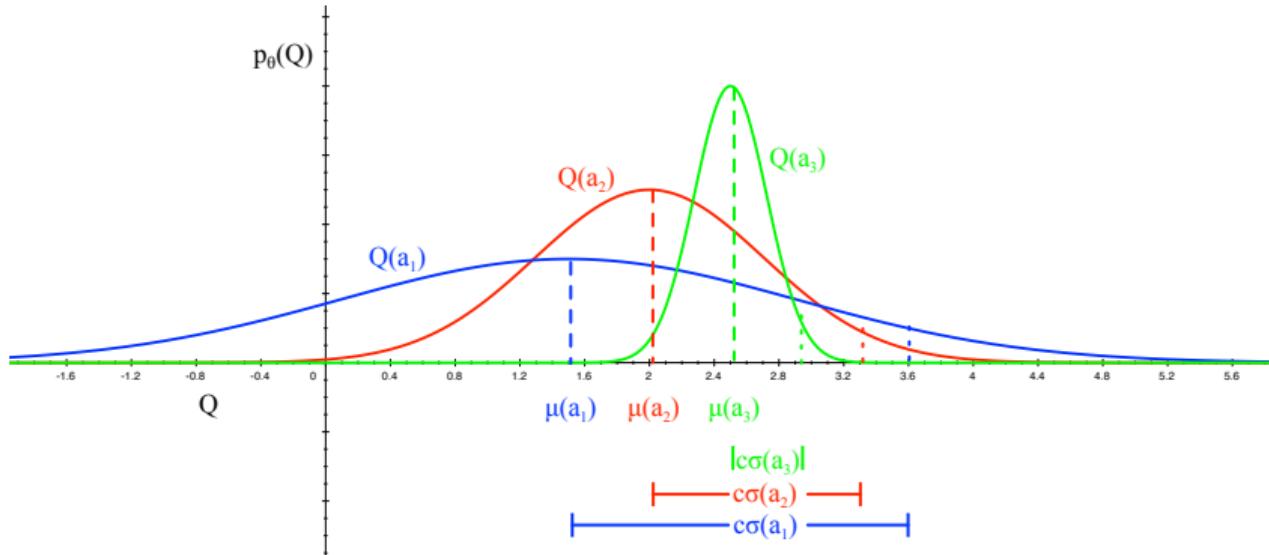


# Bayesian Bandits: Example

Suppose:  $R_1 = +1, R_2 = +1, R_3 = 0, R_4 = 0$



# Bayesian Bandits with Upper Confidence Bounds



- ▶ We can estimate upper confidences from the posterior
  - ▶ e.g.,  $U_t(a) = c\sigma_t(a)$  where  $\sigma(a)$  is std dev of  $p_t(q(a))$
- ▶ Then, pick an action that maximises  $Q_t(a) + c\sigma(a)$



# Algorithms: Thompson sampling



## Probability Matching

- ▶ A different option is to use **probability matching**:  
Select action  $a$  according to the probability (belief) that  $a$  is optimal

$$\pi_t(a) = p \left( q(a) = \max_{a'} q(a') \mid \mathcal{H}_{t-1} \right)$$

- ▶ Probability matching is optimistic in the face of uncertainty:  
Actions have higher probability when either the estimated value is high, or the uncertainty is high
- ▶ Can be difficult to compute  $\pi(a)$  analytically from posterior (but can be done numerically)



# Thompson Sampling

- ▶ Thompson sampling (Thompson 1933):
  - ▶ Sample  $Q_t(a) \sim p_t(q(a))$ ,  $\forall a$
  - ▶ Select action maximising sample,  $A_t = \operatorname{argmax}_{a \in \mathcal{A}} Q_t(a)$
- ▶ **Thompson sampling** is sample-based probability matching

$$\begin{aligned}\pi_t(a) &= \mathbb{E} \left[ \mathcal{I}(Q_t(a) = \max_{a'} Q_t(a')) \right] \\ &= p \left( q(a) = \max_{a'} q(a') \right)\end{aligned}$$

- ▶ For Bernoulli bandits, Thompson sampling achieves Lai and Robbins lower bound on regret, and therefore is **optimal**



# Planning to explore



# Information State Space

- ▶ We have viewed bandits as **one-step** decision-making problems
- ▶ Can also view as **sequential** decision-making problems
- ▶ Each step the agent updates state  $S_t$  to summarise the past
- ▶ Each action  $A_t$  causes a transition to a new **information state**  $S_{t+1}$  (by adding information), with probability  $p(S_{t+1} | A_t, S_t)$
- ▶ We now have a Markov decision problem
- ▶ The state is fully internal to the agent
- ▶ State transitions are random due to rewards & actions
- ▶ Even in bandits actions affect the future after all, via learning



## Example: Bernoulli Bandits

- ▶ Consider a Bernoulli bandit, such that

$$p(R_t = 1 \mid A_t = a) = \mu_a$$

$$p(R_t = 0 \mid A_t = a) = 1 - \mu_a$$

- ▶ E.g., win or lose a game with probability  $\mu_a$
- ▶ Want to find which arm has the highest  $\mu_a$
- ▶ The information state is  $I = (\alpha, \beta)$ 
  - ▶  $\alpha_a$  counts the pulls of arm  $a$  where reward was 0
  - ▶  $\beta_a$  counts the pulls of arm  $a$  where reward was 1



# Solving Information State Space Bandits

- ▶ We formulated the bandit as an infinite MDP over information states
- ▶ This can be solved by reinforcement learning
- ▶ E.g., learn a Bayesian reward distribution, plan into the future
- ▶ This is known as **Bayes-adaptive** RL:  
optimally trades off exploration with respect to the prior distribution
- ▶ Can be extended to full RL, by also learning a transition model
- ▶ Can be unwieldy... unclear how to scale effectively



# Example



End of lecture



# Lecture 3: Markov Decision Processes and Dynamic Programming

Diana Borsa

January 15, 2021

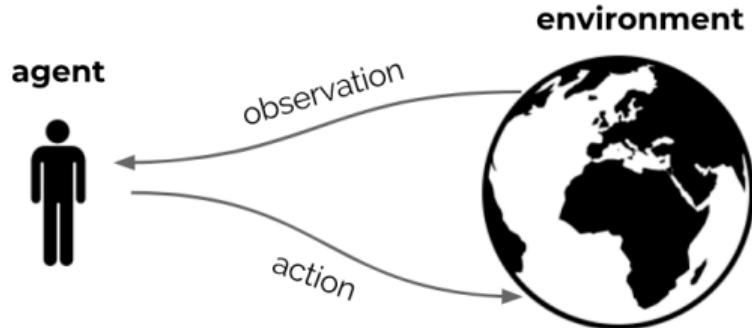


# Background

Sutton & Barto 2018, Chapter 3 + 4



## Recap



- ▶ Reinforcement learning is the science of learning to make decisions
- ▶ Agents can learn a **policy**, **value function** and/or a **model**
- ▶ The general problem involves taking into account **time** and **consequences**
- ▶ Decisions affect the **reward**, the **agent state**, and **environment state**



# This Lecture

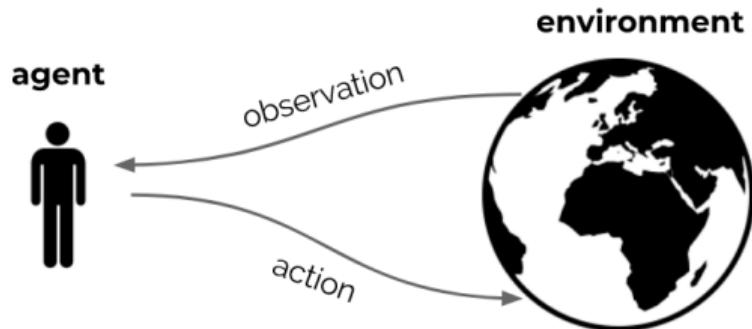
- ▶ Last lecture: multiple actions, but only one state—no model
- ▶ This lecture:
  - ▶ Formalise the problem with full **sequential structure**
  - ▶ Discuss first class of solution methods which assume **true model is given**
  - ▶ These methods are called **dynamic programming**
- ▶ Next lectures: use similar ideas, but use sampling instead of true model



# Formalising the RL interaction



# Formalising the RL interface



- ▶ We will discuss a mathematical formulation of the agent-environment interaction
- ▶ This is called a **Markov Decision Process (MDP)**
- ▶ Enables us to talk clearly about the **objective** and **how to achieve it**



## MDPs: A simplifying assumption

- ▶ For now, assume the environment is **fully observable**:  
⇒ the current **observation** contains all relevant information
  
- ▶ Note: Almost all RL problems can be formalised as MDPs, e.g.,
  - ▶ Optimal control primarily deals with continuous MDPs
  - ▶ Partially observable problems can be converted into MDPs
  - ▶ Bandits are MDPs with one state



# Markov Decision Process

Definition (Markov Decision Process - *Sutton & Barto 2018*)

A **Markov Decision Process** is a tuple  $(\mathcal{S}, \mathcal{A}, p, \gamma)$ , where

- ▶  $\mathcal{S}$  is the set of all possible states
- ▶  $\mathcal{A}$  is the set of all possible actions (e.g., motor controls)
- ▶  $p(r, s' | s, a)$  is the joint probability of a reward  $r$  and next state  $s'$ , given a state  $s$  and action  $a$
- ▶  $\gamma \in [0, 1]$  is a discount factor that trades off later rewards to earlier ones

Observations:

- ▶  $p$  defines the **dynamics** of the problem
- ▶ Sometimes it is useful to marginalise out the state transitions or expected reward:

$$p(s' | s, a) = \sum_r p(s', r | s, a) \quad \mathbb{E}[R | s, a] = \sum_r r \sum_{s'} p(r, s' | s, a).$$



# Markov Decision Process: Alternative Definition

## Definition (Markov Decision Process)

A Markov Decision Process is a tuple  $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$ , where

- ▶  $\mathcal{S}$  is the set of all possible states
- ▶  $\mathcal{A}$  is the set of all possible actions (e.g., motor controls)
- ▶  $p(s' | s, a)$  is the probability of transitioning to  $s'$ , given a state  $s$  and action  $a$
- ▶  $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  is the expected reward, achieved on a transition starting in  $(s, a)$

$$r = \mathbb{E}[R | s, a]$$

- ▶  $\gamma \in [0, 1]$  is a discount factor that trades off later rewards to earlier ones

Note: These are equivalent formulations: no additional assumptions w.r.t the previous def.



## Markov Property: *The future is independent of the past given the present*

### Definition (Markov Property)

Consider a sequence of random variables,  $\{S_t\}_{t \in \mathbb{N}}$ , indexed by time. A state  $s$  has the **Markov** property when for states  $\forall s' \in \mathcal{S}$

$$p(S_{t+1} = s' \mid S_t = s) = p(S_{t+1} = s' \mid h_{t-1}, S_t = s)$$

for all possible histories  $h_{t-1} = \{S_1, \dots, S_{t-1}, A_1, \dots, A_{t-1}, R_1, \dots, R_{t-1}\}$

In a Markov Decision Process **all states** are assumed to have the Markov property.

- ▶ The state captures all relevant information from the history.
- ▶ Once the state is known, the history may be thrown away.
- ▶ The state is a sufficient statistic of the past.



## Markov Property in a MDP: Test your understanding

In a Markov Decision Process **all states** are assumed to have the Markov property.

Q: In an MDP this property implies: (Which of the following statements are true?)

$$p(S_{t+1} = s' \mid S_t = s, A_t = a) = p(S_{t+1} = s' \mid S_1, \dots, S_{t-1}, A_1, \dots, A_t, S_t = s) \quad (1)$$

$$p(S_{t+1} = s' \mid S_t = s, A_t = a) = p(S_{t+1} = s' \mid S_1, \dots, S_{t-1}, S_t = s, A_t = a) \quad (2)$$

$$p(S_{t+1} = s' \mid S_t = s, A_t = a) = p(S_{t+1} = s' \mid S_1, \dots, S_{t-1}, S_t = s) \quad (3)$$

$$p(R_{t+1} = r, S_{t+1} = s' \mid S_t = s) = p(R_{t+1} = r, S_{t+1} = s' \mid S_1, \dots, S_{t-1}, S_t = s) \quad (4)$$



## Example: cleaning robot

- ▶ Consider a robot that cleans soda cans
- ▶ Two states: **high** battery charge or **low** battery charge
- ▶ Actions: {wait, search} in high, {wait, search, recharge} in low
- ▶ Dynamics may be stochastic
  - ▶  $p(S_{t+1} = \text{high} \mid S_t = \text{high}, A_t = \text{search}) = \alpha$
  - ▶  $p(S_{t+1} = \text{low} \mid S_t = \text{high}, A_t = \text{search}) = 1 - \alpha$
- ▶ Reward could be expected number of collected cans (deterministic), or actual number of collected cans (stochastic)

Reference: Sutton and Barto, Chapter 3, pg 52-53.

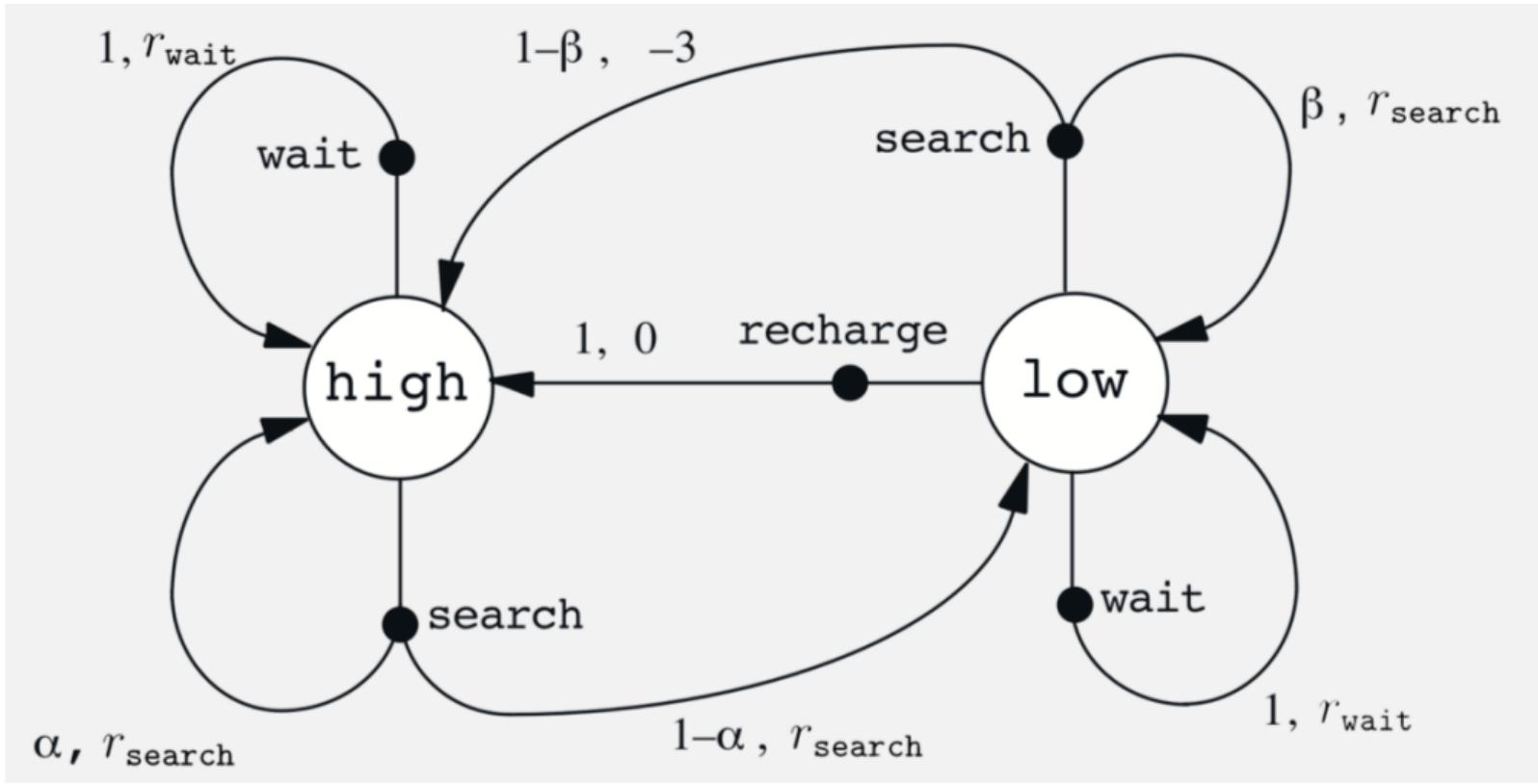


## Example: robot MDP

$s$	$a$	$s'$	$p(s'   s, a)$	$r(s, a, s')$
high	search	high	$\alpha$	$r_{\text{search}}$
high	search	low	$1 - \alpha$	$r_{\text{search}}$
low	search	high	$1 - \beta$	-3
low	search	low	$\beta$	$r_{\text{search}}$
high	wait	high	1	$r_{\text{wait}}$
high	wait	low	0	$r_{\text{wait}}$
low	wait	high	0	$r_{\text{wait}}$
low	wait	low	1	$r_{\text{wait}}$
low	recharge	high	1	0
low	recharge	low	0	0



## Example: robot MDP



# Formalising the objective



## Returns

- ▶ Acting in a MDP results in **immediate rewards**  $R_t$ , which leads to **returns**  $G_t$ :
  - ▶ Undiscounted return (episodic/finite horizon pb.)

$$G_t = R_{t+1} + R_{t+2} + \dots + R_T = \sum_{k=0}^{T-t-1} R_{t+k+1}$$

- ▶ Discounted return (finite or infinite horizon pb.)

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t} R_T = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

- ▶ Average return (continuing, infinite horizon pb.)

$$G_t = \frac{1}{T-t-1} (R_{t+1} + R_{t+2} + \dots + R_T) = \frac{1}{T-t-1} \sum_{k=0}^{T-t-1} R_{t+k+1}$$

Note: These are random variables that depends on **MDP** and **policy**



## Discounted Return

- ▶ Discounted **returns**  $G_t$  for infinite horizon  $T \rightarrow \infty$ :

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- ▶ The **discount**  $\gamma \in [0, 1]$  is the present value of future rewards
  - ▶ The marginal value of receiving reward  $R$  after  $k + 1$  time-steps is  $\gamma^k R$
  - ▶ For  $\gamma < 1$ , immediate rewards are more important than delayed rewards
  - ▶  $\gamma$  close to 0 leads to "myopic" evaluation
  - ▶  $\gamma$  close to 1 leads to "far-sighted" evaluation



# Why discount?

Most Markov decision processes are discounted. Why?

- ▶ Problem specification:
  - ▶ Immediate rewards may actually be more valuable (e.g., consider earning interest)
  - ▶ Animal/human behaviour shows preference for immediate reward
- ▶ Solution side:
  - ▶ Mathematically convenient to discount rewards
  - ▶ Avoids infinite returns in cyclic Markov processes
- ▶ The way to think about it: **reward and discount together determine the goal**



# Policies

## Goal of an RL agent

To find a behaviour policy that maximises the (**expected**) return  $G_t$

- ▶ A **policy** is a mapping  $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$  that, for every state  $s$  assigns **for each action  $a \in \mathcal{A}$  the probability of taking that action in state  $s$** . Denoted by  $\pi(a|s)$ .
- ▶ For deterministic policies, we sometimes use the notation  $a_t = \pi(s_t)$  to denote the action taken by the policy.



## Value Functions

- ▶ The **value function**  $v(s)$  gives the long-term value of state  $s$

$$v_{\pi}(s) = \mathbb{E} [G_t \mid S_t = s, \pi]$$

- ▶ We can define **(state-)action values**:

$$q_{\pi}(s, a) = \mathbb{E} [G_t \mid S_t = s, A_t = a, \pi]$$

- ▶ (Connection between them) Note that:

$$v_{\pi}(s) = \sum_a \pi(a \mid s) q_{\pi}(s, a) = \mathbb{E} [q_{\pi}(S_t, A_t) \mid S_t = s, \pi] , \forall s$$



# Optimal Value Function

## Definition (Optimal value functions)

The **optimal state-value function**  $v^*(s)$  is the maximum value function over all policies

$$v^*(s) = \max_{\pi} v_{\pi}(s)$$

The **optimal action-value function**  $q^*(s, a)$  is the maximum action-value function over all policies

$$q^*(s, a) = \max_{\pi} q_{\pi}(s, a)$$

- ▶ The optimal value function specifies the best possible performance in the MDP
- ▶ An MDP is “solved” when we know the optimal value function



# Optimal Policy

Define a partial ordering over policies

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s), \forall s$$

## Theorem (Optimal Policies)

For any Markov decision process

- ▶ There exists an *optimal policy*  $\pi^*$  that is better than or equal to all other policies,  
 $\pi^* \geq \pi, \forall \pi$   
(There can be more than one such optimal policy.)
- ▶ All optimal policies achieve the optimal value function,  $v^{\pi^*}(s) = v^*(s)$
- ▶ All optimal policies achieve the optimal action-value function,  $q^{\pi^*}(s, a) = q^*(s, a)$



## Finding an Optimal Policy

An optimal policy can be found by maximising over  $q^*(s, a)$ ,

$$\pi^*(s, a) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} q^*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

Observations:

- ▶ There is always a **deterministic optimal policy** for any MDP
- ▶ If we **know  $q^*(s, a)$** , we immediately have **the optimal policy**
- ▶ There can be **multiple optimal policies**
- ▶ If multiple actions maximize  $q_*(s, \cdot)$ , we can also just pick any of these (including stochastically)



# Bellman Equations



## Value Function

- ▶ The value function  $v(s)$  gives the long-term value of state  $s$

$$v_\pi(s) = \mathbb{E}[G_t \mid S_t = s, \pi]$$

- ▶ It can be defined recursively:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, \pi] \\ &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t \sim \pi(S_t)] \\ &= \sum_a \pi(a \mid s) \sum_r \sum_{s'} p(r, s' \mid s, a) (r + \gamma v_\pi(s')) \end{aligned}$$

- ▶ The final step writes out the expectation explicitly



## Action values

- We can define state-action values

$$q_{\pi}(s, a) = \mathbb{E}[G_t \mid S_t = s, A_t = a, \pi]$$

- This implies

$$\begin{aligned} q_{\pi}(s, a) &= \mathbb{E}[R_{t+1} + \gamma v_{\pi}(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_r \sum_{s'} p(r, s' \mid s, a) \left( r + \gamma \sum_{a'} \pi(a' \mid s') q_{\pi}(s', a') \right) \end{aligned}$$

- Note that

$$v_{\pi}(s) = \sum_a \pi(a \mid s) q_{\pi}(s, a) = \mathbb{E}[q_{\pi}(S_t, A_t) \mid S_t = s, \pi], \forall s$$



# Bellman Equations

## Theorem (Bellman Expectation Equations)

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , for any policy  $\pi$ , the value functions obey the following expectation equations:

$$v_\pi(s) = \sum_a \pi(s, a) \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) v_\pi(s') \right] \quad (5)$$

$$q_\pi(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|a, s) \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a') \quad (6)$$



# The Bellman Optimality Equations

## Theorem (Bellman Optimality Equations)

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , the **optimal value functions** obey the following expectation equations:

$$v^*(s) = \max_a \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) v^*(s') \right] \quad (7)$$

$$q^*(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|a, s) \max_{a' \in \mathcal{A}} q^*(s', a') \quad (8)$$

There can be no policy with a higher value than  $v_*(s) = \max_\pi v_\pi(s)$ ,  $\forall s$



## Some intuition

(Reminder) Greedy on  $v^*$  = Optimal Policy

- ▶ An optimal policy can be found by maximising over  $q^*(s, a)$ ,

$$\pi^*(s, a) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} q^*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

- ▶ Apply the Bellman Expectation Eq. (6):

$$\begin{aligned} q_{\pi^*}(s, a) &= r(s, a) + \gamma \sum_{s'} p(s'|a, s) \underbrace{\sum_{a' \in \mathcal{A}} \pi^*(a'|s') q_{\pi^*}(s', a')}_{\max_{a'} q^*(s', a')} \\ &= r(s, a) + \gamma \sum_{s'} p(s'|a, s) \max_{a' \in \mathcal{A}} q^*(s', a') \end{aligned}$$



# Solving RL problems using the Bellman Equations



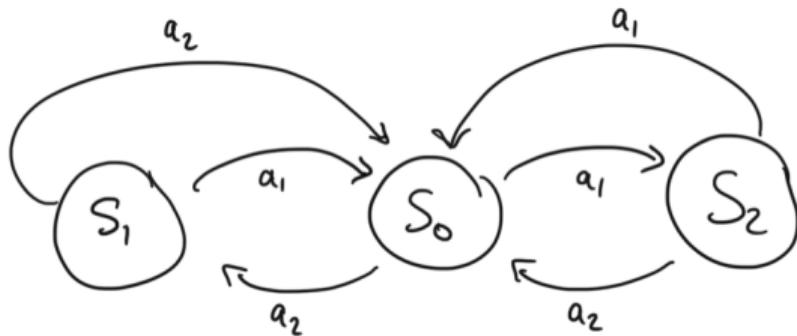
## Problems in RL

- ▶ *Pb1:* Estimating  $v_\pi$  or  $q_\pi$  is called **policy evaluation** or, simply, **prediction**
  - ▶ Given a policy, what is my expected return under that behaviour?
  - ▶ Given this treatment protocol/trading strategy, what is my expected return?
- ▶ *Pb2:* Estimating  $v_*$  or  $q_*$  is sometimes called **control**, because these can be used for **policy optimisation**
  - ▶ What is the optimal way of behaving? What is the optimal value function?
  - ▶ What is the optimal treatment? What is the optimal control policy to minimise time, fuel consumption, etc?



## Exercise:

- ▶ Consider the following MDP:

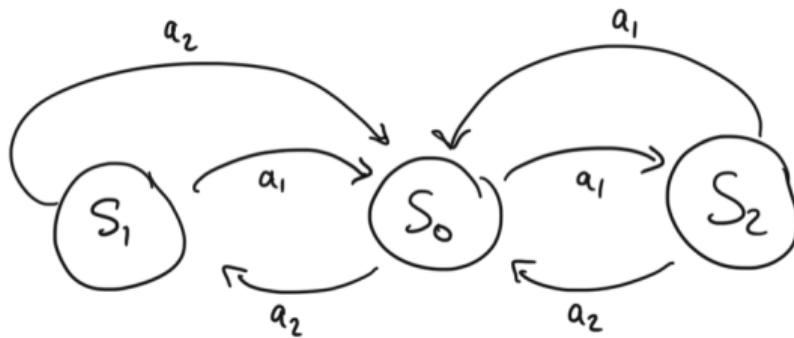


- ▶ The actions have a 0.9 probability of success and with 0.1 probably we remain in the same state
- ▶  $R_t = 0$  for all transitions that end up in  $S_0$ , and  $R_t = -1$  for all other transitions



## Exercise: (pause to work this out)

- ▶ Consider the following MDP:



- ▶ The actions have a 0.9 probability of success and with 0.1 probably we remain in the same state
- ▶  $R_t = 0$  for all transitions that end up in  $S_0$ , and  $R_t = -1$  for all other transitions
- ▶ **Q:** Evaluation problems (Consider a discount  $\gamma = 0.9$ )
  - ▶ What is  $v_\pi$  for  $\pi(s) = a_1(\rightarrow), \forall s?$
  - ▶ What is  $v_\pi$  for the uniformly random policy?
  - ▶ Same policy evaluation problems for  $\gamma = 0.0$ ? (What do you notice?)



A solution



## Bellman Equation in Matrix Form

- ▶ The Bellman value equation, for given  $\pi$ , can be expressed using matrices,

$$\mathbf{v} = \mathbf{r}^\pi + \gamma \mathbf{P}^\pi \mathbf{v}$$

where

$$v_i = v(s_i)$$

$$r_i^\pi = \mathbb{E}[R_{t+1} \mid S_t = s_i, A_t \sim \pi(S_t)]$$

$$P_{ij}^\pi = p(s_j \mid s_i) = \sum_a \pi(a \mid s_i) p(s_j \mid s_i, a)$$



## Bellman Equation in Matrix Form

- ▶ The Bellman equation, for a given policy  $\pi$ , can be expressed using matrices,

$$\mathbf{v} = \mathbf{r}^\pi + \gamma \mathbf{P}^\pi \mathbf{v}$$

- ▶ This is a linear equation that can be solved directly:

$$\begin{aligned}\mathbf{v} &= \mathbf{r}^\pi + \gamma \mathbf{P}^\pi \mathbf{v} \\ (\mathbf{I} - \gamma \mathbf{P}^\pi) \mathbf{v} &= \mathbf{r}^\pi \\ \mathbf{v} &= (\mathbf{I} - \gamma \mathbf{P}^\pi)^{-1} \mathbf{r}^\pi\end{aligned}$$

- ▶ Computational complexity is  $O(|S|^3)$  — only possible for small problems
- ▶ There are iterative methods for larger problems
  - ▶ Dynamic programming
  - ▶ Monte-Carlo evaluation
  - ▶ Temporal-Difference learning



# Solving the Bellman Optimality Equation

- ▶ The Bellman optimality equation is non-linear
- ▶ Cannot use the same direct matrix solution as for policy optimisation (in general)
- ▶ Many iterative solution methods:
  - ▶ Using models / **dynamic programming**
    - ▶ Value iteration
    - ▶ Policy iteration
  - ▶ Using samples
    - ▶ Monte Carlo
    - ▶ Q-learning
    - ▶ Sarsa



# Dynamic Programming



# Dynamic Programming

*The 1950s were not good years for mathematical research. I felt I had to shield the Air Force from the fact that I was really doing mathematics. What title, what name, could I choose? I was interested in planning, in decision making, in thinking. But planning is not a good word for various reasons. I decided to use the word 'programming.' I wanted to get across the idea that this was dynamic, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has a precise meaning, namely dynamic, in the classical physical sense. It also is impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought **dynamic programming** was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.*

– Richard Bellman  
(slightly paraphrased for conciseness)



# Dynamic programming

*Dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP).*

Sutton & Barto 2018

- ▶ We will discuss several dynamic programming methods to solve MDPs
- ▶ All such methods consist of two important parts:

policy evaluation    and    policy improvement



## Policy evaluation

- ▶ We start by discussing how to estimate

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid s, \pi]$$

- ▶ Idea: turn this equality into an update

### Algorithm

- ▶ First, initialise  $v_0$ , e.g., to zero
- ▶ Then, iterate

$$\forall s : v_{k+1}(s) \leftarrow \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid s, \pi]$$

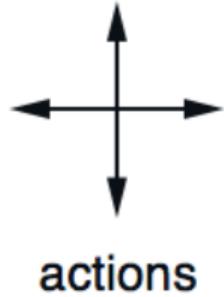
- ▶ Stopping: whenever  $v_{k+1}(s) = v_k(s)$ , for all  $s$ , we must have found  $v_\pi$

- ▶ Q: Does this algorithm always converge?

Answer: Yes, under appropriate conditions (e.g.,  $\gamma < 1$ ). More next lecture!



## Example: Policy evaluation



	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$R_t = -1$   
on all transitions



## Policy evaluation

$k = 0$

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

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0



## Policy evaluation

$k = 3$

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

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

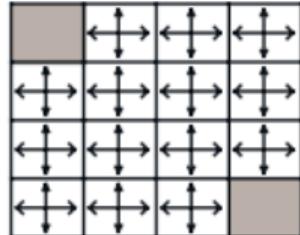
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



## Policy evaluation + Greedy Improvement

$k = 0$

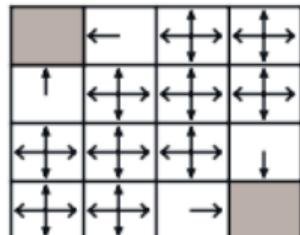
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



random policy

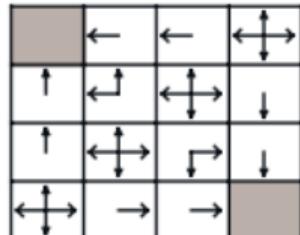
$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0



$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0



## Policy evaluation + Greedy Improvement

$k = 3$

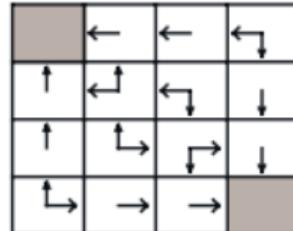
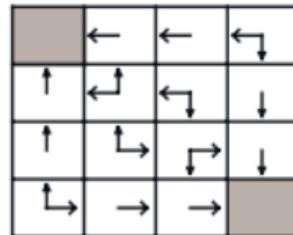
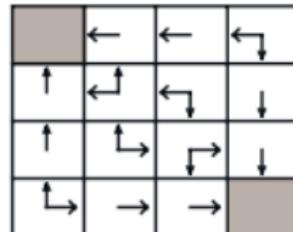
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

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



optimal  
policy



## Policy Improvement

- ▶ The example already shows we can use evaluation to then improve our policy
- ▶ In fact, just being greedy with respect to the values of the random policy sufficed! (That is not true in general)

### Algorithm

Iterate, using

$$\begin{aligned}\forall s : \pi_{\text{new}}(s) &= \underset{a}{\operatorname{argmax}} q_{\pi}(s, a) \\ &= \underset{a}{\operatorname{argmax}} \mathbb{E}[R_{t+1} + \gamma v_{\pi}(S_{t+1}) \mid S_t = s, A_t = a]\end{aligned}$$

Then, evaluate  $\pi_{\text{new}}$  and repeat

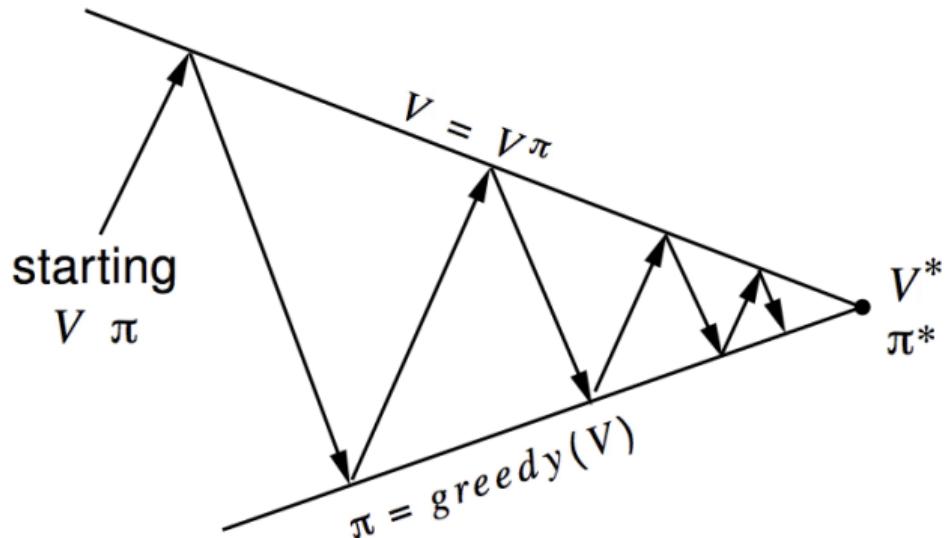
- ▶ Claim: One can show that  $v_{\pi_{\text{new}}}(s) \geq v_{\pi}(s)$ , for all  $s$



Policy Improvement:  $q_{\pi_{\text{new}}}(s, a) \geq q_{\pi}(s, a)$

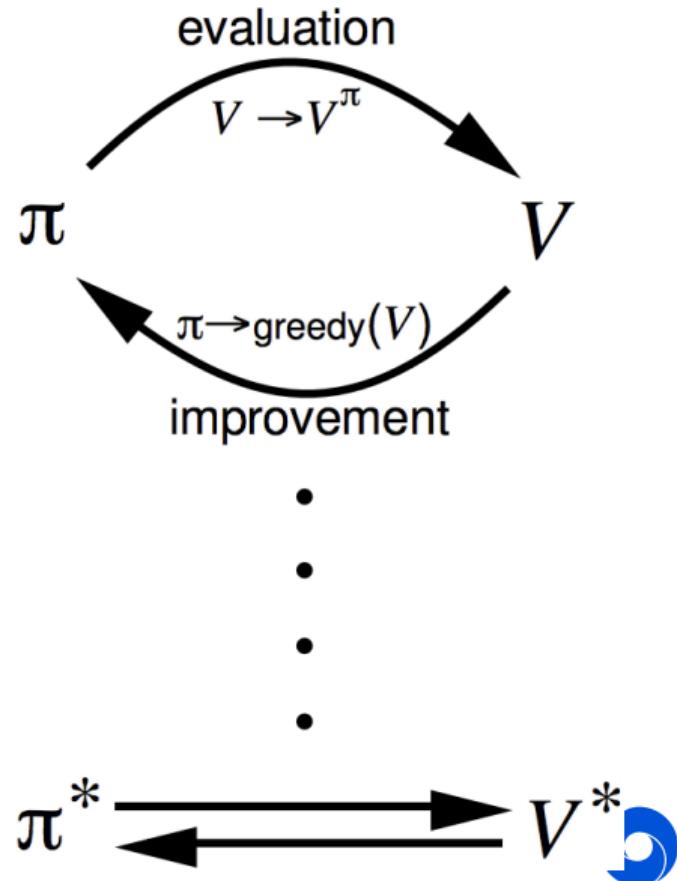


## Policy Iteration



Policy evaluation Estimate  $v^\pi$

Policy improvement Generate  $\pi' \geq \pi$



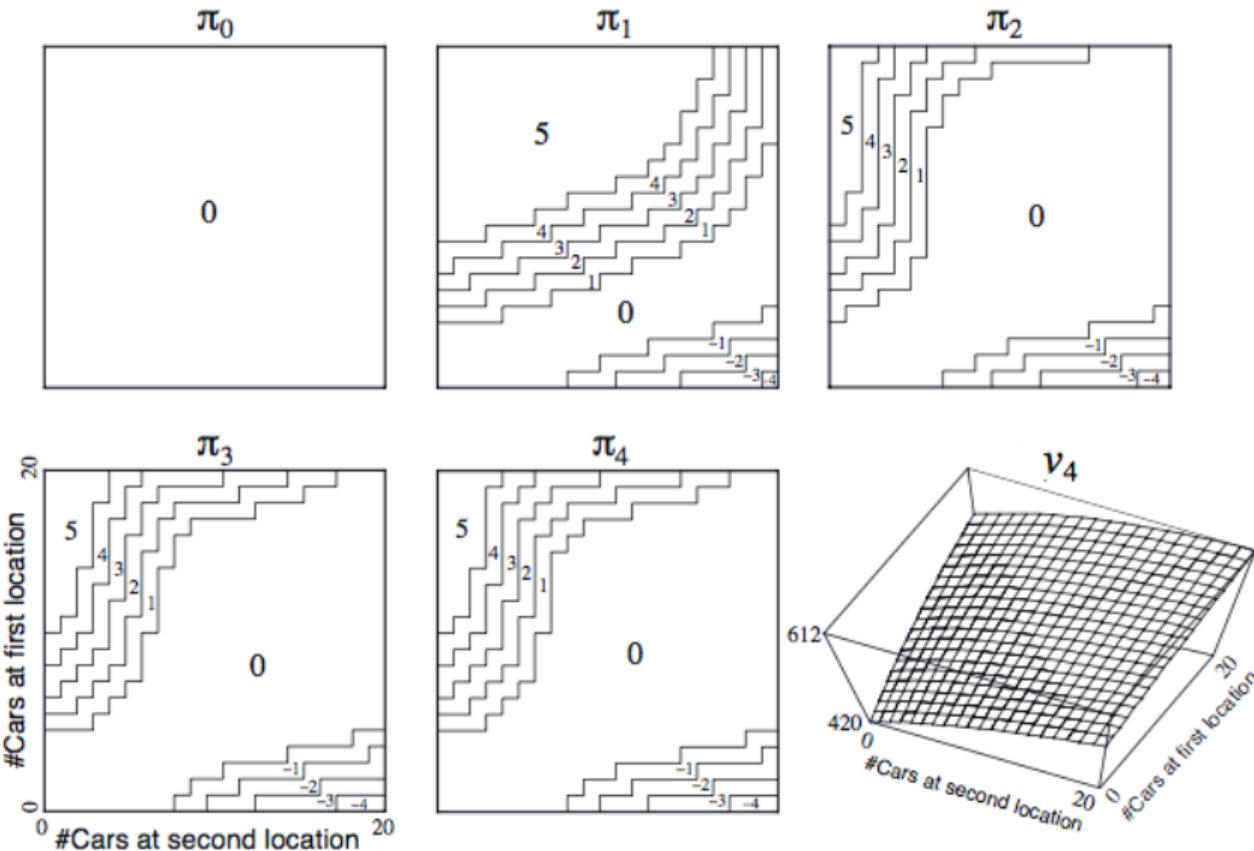
## Example: Jack's Car Rental



- ▶ States: Two locations, maximum of 20 cars at each
- ▶ Actions: Move up to 5 cars overnight (-\$2 each)
- ▶ Reward: \$10 for each available car rented,  $\gamma = 0.9$
- ▶ Transitions: Cars returned and requested randomly
  - ▶ Poisson distribution,  $n$  returns/requests with prob  $\frac{\lambda^n}{n!} e^{-\lambda}$
  - ▶ 1st location: average requests = 3, average returns = 3
  - ▶ 2nd location: average requests = 4, average returns = 2



## Example: Jack's Car Rental – Policy Iteration



## Policy Iteration

- ▶ Does policy evaluation need to converge to  $v^\pi$ ?
- ▶ Or should we stop when we are ‘close’?  
(E.g., with a threshold on the change to the values)
  - ▶ Or simply stop after  $k$  iterations of iterative policy evaluation?
  - ▶ In the small gridworld  $k = 3$  was sufficient to achieve optimal policy
- ▶ **Extreme:** Why not update policy every iteration — i.e. stop after  $k = 1$ ?
  - ▶ This is equivalent to **value iteration**



## Value Iteration

- We could take the Bellman **optimality** equation, and turn that into an update

$$\forall s : v_{k+1}(s) \leftarrow \max_a \mathbb{E} [R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = s]$$

- This is equivalent to **policy iteration**, with  **$k = 1$  step of policy evaluation** between each two (greedy) policy improvement steps

### Algorithm: Value Iteration

- Initialise  $v_0$
- Update:  $v_{k+1}(s) \leftarrow \max_a \mathbb{E} [R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = s]$
- **Stopping:** whenever  $v_{k+1}(s) = v_k(s)$ , for all  $s$ , we must have found  $v^*$



## Example: Shortest Path

g			

Problem

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

$V_1$

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

$V_2$

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

$V_3$

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

$V_4$

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

$V_5$

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

$V_6$

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

$V_7$



# Synchronous Dynamic Programming Algorithms

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

Observations:

- ▶ Algorithms are based on state-value function  $v_\pi(s)$  or  $v^*(s) \Rightarrow$  complexity  $O(|\mathcal{A}||\mathcal{S}|^2)$  per iteration, for  $|\mathcal{A}|$  actions and  $|\mathcal{S}|$  states
- ▶ Could also apply to action-value function  $q_\pi(s, a)$  or  $q^*(s, a) \Rightarrow$  complexity  $O(|\mathcal{A}|^2|\mathcal{S}|^2)$  per iteration



# Extensions to Dynamic Programming



# Asynchronous Dynamic Programming

- ▶ DP methods described so far used **synchronous** updates (all states in parallel)
- ▶ **Asynchronous DP**
  - ▶ backs up states individually, in any order
  - ▶ can significantly reduce computation
  - ▶ guaranteed to converge if all states continue to be selected



# Asynchronous Dynamic Programming

Three simple ideas for asynchronous dynamic programming:

- ▶ In-place dynamic programming
- ▶ Prioritised sweeping
- ▶ Real-time dynamic programming



# In-Place Dynamic Programming

- ▶ Synchronous value iteration stores two copies of value function

for all  $s$  in  $\mathcal{S}$  :  $v_{\text{new}}(s) \leftarrow \max_a \mathbb{E} [R_{t+1} + \gamma v_{\text{old}}(S_{t+1}) \mid S_t = s]$   
 $v_{\text{old}} \leftarrow v_{\text{new}}$

- ▶ In-place value iteration only stores one copy of value function

for all  $s$  in  $\mathcal{S}$  :  $v(s) \leftarrow \max_a \mathbb{E} [R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]$



## Prioritised Sweeping

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

$$\left| \max_a \mathbb{E} [R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] - v(s) \right|$$

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



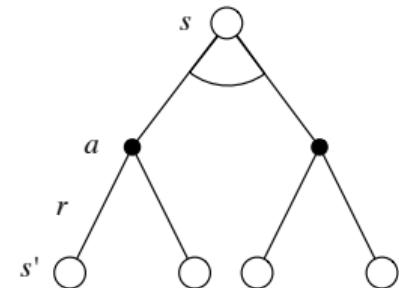
## Real-Time Dynamic Programming

- ▶ Idea: only update states that are relevant to agent
- ▶ E.g., if the agent is in state  $S_t$ , update that state value, or states that it expects to be in soon



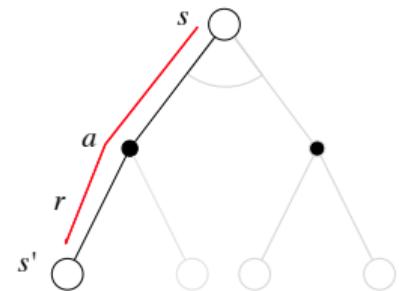
# Full-Width Backups

- ▶ Standard DP uses **full-width backups**
- ▶ For each backup (sync or async)
  - ▶ Every successor state and action is considered
  - ▶ Using true model of transitions and reward function
- ▶ DP is effective for medium-sized problems (millions of states)
- ▶ For large problems DP suffers from **curse of dimensionality**
  - ▶ Number of states  $n = |\mathcal{S}|$  grows exponentially with number of state variables
- ▶ Even one full backup can be too expensive



# Sample Backups

- ▶ In subsequent lectures we will consider **sample backups**
- ▶ Using sample rewards and sample transitions  $\langle s, a, r, s' \rangle$   
(Instead of reward function  $r$  and transition dynamics  $p$ )
- ▶ Advantages:
  - ▶ Model-free: no advance knowledge of MDP required
  - ▶ Breaks the curse of dimensionality through sampling
  - ▶ Cost of backup is constant, independent of  $n = |\mathcal{S}|$



# Summary



## What have we covered today?

- ▶ Markov Decision Processes
- ▶ Objectives in an MDP: different notion of return
- ▶ Value functions - expected returns, condition on state (and action)
- ▶ Optimality principles in MDPs: optimal value functions and optimal policies
- ▶ Bellman Equations
- ▶ Two class of problems in RL: evaluation and control
- ▶ How to compute  $v_\pi$  (aka solve an evaluation/prediction problem)
- ▶ How to compute the optimal value function via dynamic programming:
  - ▶ Policy Iteration
  - ▶ Value Iteration



## Questions?

*The only stupid question is the one you were afraid to ask but never did.*  
-Rich Sutton

For questions that may arise during this lecture please use Moodle and/or the next Q&A session.



# Lecture 4: Theoretical Fundamentals of Dynamic Programming

Diana Borsa

21st January 2021, UCL



## This Lecture

- ▶ Last lecture: MDP, DP, Value Iteration (VI), Policy Iteration (PI)
- ▶ This lecture:
  - ▶ Deepen the mathematical formalism behind the MDP framework.
  - ▶ Revisit the **Bellman equations** and introduce their corresponding **operators**.
  - ▶ Re-visit the paradigm of **dynamic programming**: VI and PI.
- ▶ Next lectures: approximate, sampled versions of these paradigms, mainly in the absence of perfect knowledge of the environment.



# Preliminaries

## (Quick Recap of Functional Analysis)



# Normed Vector Spaces

- ▶ Normed Vector Spaces: vector space  $\mathcal{X}$  + a norm  $\|\cdot\|$  on the elements of  $\mathcal{X}$ .
- ▶ Norms are defined a mapping  $\mathcal{X} \rightarrow \mathbb{R}$  s.t:
  1.  $\|x\| \geq 0, \forall x \in \mathcal{X}$  and if  $\|x\| = 0$  then  $x = \mathbf{0}$ .
  2.  $\|\alpha x\| = |\alpha| \|x\|$  (homogeneity)
  3.  $\|x_1 + x_2\| \leq \|x_1\| + \|x_2\|$  (triangle inequality)
- ▶ For this lecture:
  - ▶ Vector spaces:  $\mathcal{X} = \mathbb{R}^d$
  - ▶ Norms:
    - ▶ max-norm/ $L_\infty$  norm  $\|\cdot\|_\infty$
    - ▶ (weighted)  $L_2$  norms  $\|\cdot\|_{2,\rho}$



# Contraction Mapping

## Definition

Let  $\mathcal{X}$  be a vector space, equipped with a norm  $\|\cdot\|$ . A mapping  $\mathcal{T} : \mathcal{X} \rightarrow \mathcal{X}$  is a  **$\alpha$ -contraction** mapping if for any  $x_1, x_2 \in \mathcal{X}$ ,  $\exists \alpha \in [0, 1)$  s.t.

$$\|\mathcal{T}x_1 - \mathcal{T}x_2\| \leq \alpha \|x_1 - x_2\|$$

- ▶ If  $\alpha \in [0, 1]$ , then we call  $\mathcal{T}$  **non-expanding**
- ▶ Every contraction is also (by definition) **Lipschitz**, thus it is also **continuous**. In particular this means:

$$\text{If } x_n \rightarrow_{\|\cdot\|} x \text{ then } \mathcal{T}x_n \rightarrow_{\|\cdot\|} \mathcal{T}x$$



# Fixed point

## Definition

A point/vector  $x \in \mathcal{X}$  is a **fixed point** of an operator  $\mathcal{T}$  if  $\mathcal{T}x = x$ .



# Banach Fixed Point Theorem

## Theorem (Banach Fixed Point Theorem)

Let  $\mathcal{X}$  a *complete normed* vector space, equipped with a norm  $\|\cdot\|$  and  $\mathcal{T} : \mathcal{X} \rightarrow \mathcal{X}$  a  *$\gamma$ -contraction* mapping, then:

1.  $\mathcal{T}$  has a *unique fixed point*  $x \in \mathcal{X}$ :  $\exists ! x^* \in \mathcal{X}$  s.t.  $\mathcal{T}x^* = x^*$
2.  $\forall x_0 \in \mathcal{X}$ , the sequence  $x_{n+1} = \mathcal{T}x_n$  converges to  $x^*$  in a geometric fashion:

$$\|x_n - x^*\| \leq \gamma^n \|x_0 - x^*\|$$

Thus  $\lim_{n \rightarrow \infty} \|x_n - x^*\| \leq \lim_{n \rightarrow \infty} (\gamma^n \|x_0 - x^*\|) = 0$ .



# Markov Decision Processes and Dynamic Programming (Recap)



## (Recap) MDPs

- ▶ **Markov Decision Processes** (MDPs) formally describe an environment:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r, \gamma)$$

- ▶ Almost all RL problems can be formalised as MDPs, e.g.
  - ▶ Optimal control primarily deals with continuous MDPs
  - ▶ Partially observable problems can be converted into MDPs
  - ▶ Bandits are MDPs with one state



## (Recap) Value functions

- ▶ State value function, for a policy  $\pi$  :

$$v_\pi(s) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R_{t+1} | s_0 = s; \pi \right]$$

- ▶ Action value function, for a policy  $\pi$ :

$$q_\pi(s, a) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R_{t+1} | s_0 = s, a_0 = a, \pi \right]$$

- ▶ Optimal value functions:  $q^* = \max_\pi q_\pi$  ( $v^* = \max_\pi v_\pi$ )



## (Recap) Bellman Equations

### Theorem (Bellman Expectation Equations)

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , for any policy  $\pi$ , the value functions obey the following expectation equations:

$$v_\pi(s) = \sum_a \pi(s, a) \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) v_\pi(s') \right] \quad (1)$$

$$q_\pi(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|a, s) \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a') \quad (2)$$



## (Recap) The Bellman Optimality Equation

### Theorem (Bellman Optimality Equations)

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , the *optimal value functions* obey the following expectation equations:

$$v^*(s) = \max_a \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) v^*(s') \right] \quad (3)$$

$$q^*(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|a, s) \max_{a' \in \mathcal{A}} q^*(s', a') \quad (4)$$



# Bellman Operators



# The Bellman Optimality Operator

## Definition (Bellman Optimality Operator $T_{\mathcal{V}}^*$ )

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , let  $\mathcal{V} \equiv \mathcal{V}_{\mathcal{S}}$  be the space of bounded real-valued functions over  $\mathcal{S}$ . We define, point-wise, the **Bellman Optimality operator**  $T_{\mathcal{V}}^* : \mathcal{V} \rightarrow \mathcal{V}$  as:

$$(T_{\mathcal{V}}^* f)(s) = \max_a \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) f(s') \right], \quad \forall f \in \mathcal{V} \quad (5)$$

As a common convention we drop the index  $\mathcal{V}$  and simply use  $T^* = T_{\mathcal{V}}^*$



## Properties of the Bellman Operator $T^*$

1. It has one **unique fixed point  $v^*$ .**

$$T^*v^* = v^*$$

2.  $T^*$  is a  **$\gamma$ -contraction** wrt. to  $\|.\|_\infty$

$$\|T^*v - T^*u\|_\infty \leq \gamma \|v - u\|_\infty, \forall u, v \in \mathcal{V}$$

3.  $T^*$  is **monotonic**:

$\forall u, v \in \mathcal{V}$  s.t.  $u \leq v$ , component-wise, then  $T^*u \leq T^*v$



## Properties of the Bellman Operator $T^*$ (Proofs)

Prop. (2):  $T^*$  is a  **$\gamma$ -contraction** wrt. to  $\| \cdot \|_\infty$

Proof.

$$|T^*v(s) - T^*u(s)| = |\max_a [r(s, a) + \gamma \mathbb{E}_{s'|s,a} v(s')] - \max_b [r(s, b) + \gamma \mathbb{E}_{s''|s,b} u(s'')]| \quad (6)$$

$$\leq \max_a |[r(s, a) + \gamma \mathbb{E}_{s'|s,a} v(s')] - [r(s, a) + \gamma \mathbb{E}_{s'|s,a} u(s')]| \quad (7)$$

$$= \gamma \max_a |\mathbb{E}_{s'|s,a} [v(s') - u(s')]| \quad (8)$$

$$\leq \gamma \max_{s'} |[v(s') - u(s')]| \quad (9)$$

Thus we get:

$$\|T^*v - T^*u\|_\infty \leq \gamma \|v - u\|_\infty, \forall u, v \in \mathcal{V}$$



Note: Step (6)-(7) uses:  $|\max_a f(a) - \max_b g(b)| \leq \max_a |f(a) - g(a)|$



## Properties of the Bellman Operator $T^*$ (Proofs)

Prop. (3):  $T^*$  is **monotonic**

Proof.

Given  $v(s) \leq u(s), \forall s \Rightarrow r(s, a) + \mathbb{E}_{s'|s,a}v(s') \leq r(s, a) + \mathbb{E}_{s'|s,a}u(s')$ .

$$T^*v(s) - T^*u(s) = \max_a [r(s, a) + \gamma \mathbb{E}_{s'|s,a}v(s')] - \max_b [r(s, b) + \gamma \mathbb{E}_{s''|s,b}u(s'')] \quad (10)$$

$$\leq \max_a ([r(s, a) + \gamma \mathbb{E}_{s'|s,a}v(s')] - [r(s, a) + \gamma \mathbb{E}_{s'|s,a}u(s')]) \quad (11)$$

$$\leq 0, \forall s. \quad (12)$$

Thus  $T^*v(s) \leq T^*u(s), \forall s \in \mathcal{S}$ .



# Value Iteration through the lens of the Bellman Operator

## Value Iteration

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = T^*v_k$ .

As  $k \rightarrow \infty$ ,  $v_k \rightarrow_{\|\cdot\|_\infty} v^*$ .

*Proof:* Direct application of the *Banach Fixed Point Theorem*.

$$\begin{aligned}\|v_k - v^*\|_\infty &= \|T^*v_{k-1} - v^*\|_\infty \\&= \|T^*v_{k-1} - T^*v^*\|_\infty \quad (\text{fixed point prop.}) \\&\leq \gamma\|v_{k-1} - v^*\|_\infty \quad (\text{contraction prop.}) \\&\leq \gamma^k\|v_0 - v^*\|_\infty \quad (\text{iterative application})\end{aligned}$$



# The Bellman Expectation Operator

## Definition (Bellman Expectation Operator)

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , let  $\mathcal{V} \equiv \mathcal{V}_{\mathcal{S}}$  be the space of bounded real-valued functions over  $\mathcal{S}$ . For any policy  $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ , we define, point-wise, the **Bellman Expectation operator**  $T_{\mathcal{V}}^{\pi} : \mathcal{V} \rightarrow \mathcal{V}$  as:

$$(T_{\mathcal{V}}^{\pi} f)(s) = \sum_a \pi(s, a) \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) f(s') \right], \forall f \in \mathcal{V} \quad (13)$$



## Properties of the Bellman Operator $T^\pi$

1. It has one **unique fixed point**  $v_\pi$ .

$$T^\pi v_\pi = v_\pi$$

2.  $T^\pi$  is a  **$\gamma$ -contraction** wrt. to  $\|.\|_\infty$

$$\|T^\pi v - T^\pi u\|_\infty \leq \gamma \|v - u\|_\infty, \forall u, v \in \mathcal{V}$$

3.  $T^\pi$  is **monotonic**:

$\forall u, v \in \mathcal{V}$  s.t.  $u \leq v$ , component-wise, then  $T^\pi u \leq T^\pi v$



## Properties of the Bellman Operator $T^\pi$ (Proofs)

Prop. (2):  $T^\pi$  is a  **$\gamma$ -contraction** wrt. to  $\|.\|_\infty$

Proof.

$$\begin{aligned} T^\pi v(s) - T^\pi u(s) &= \sum_a \pi(a|s) [r(s, a) + \gamma \mathbb{E}_{s'|s,a} v(s') - r(s, a) - \gamma \mathbb{E}_{s'|s,a} u(s')] \\ &= \gamma \sum_a \pi(a|s) \mathbb{E}_{s'|s,a} [v(s') - u(s')] \tag{14} \\ \Rightarrow |T^\pi v(s) - T^\pi u(s)| &\leq \gamma \max_{s'} |[v(s') - u(s')]| \end{aligned}$$

Thus we get:

$$\|T^\pi v - T^\pi u\|_\infty \leq \gamma \|v - u\|_\infty, \forall u, v \in \mathcal{V}$$

□

Note: (14) gives us also Prop. (3), monotonicity of  $T^\pi$ .



# Policy Evaluation

## (Iterative) Policy Evaluation

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = T^\pi v_k$ .

As  $k \rightarrow \infty$ ,  $v_k \rightarrow_{\|\cdot\|_\infty} v_\pi$ .

*Proof:* Direct application of the *Banach Fixed Point Theorem*.



# (Summary) Dynamic Programming with Bellman Operators

## Value Iteration

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = T^*v_k$ .

## Policy Iteration

- ▶ Start with  $\pi_0$ .
- ▶ Iterate:
  - ▶ Policy Evaluation:  $v_{\pi_i}$ 
    - ▶ (E.g. For instance, by iterating  $T^\pi$ :  $v_k = T^{\pi_i}v_{k-1} \Rightarrow v_k \rightarrow v^{\pi_i}$  as  $k \rightarrow \infty$ )
  - ▶ Greedy Improvement:  $\pi_{i+1} = \arg \max_a q_{\pi_i}(s, a)$



Similarly for  $q^\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  functions

### Definition (Bellman Expectation Operator)

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , let  $\mathcal{Q} \equiv \mathcal{Q}_{\mathcal{S}, \mathcal{A}}$  be the space of bounded real-valued functions over  $\mathcal{S} \times \mathcal{A}$ . For any policy  $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ , we define, point-wise, the **Bellman Expectation operator**  $T_Q^\pi : \mathcal{Q} \rightarrow \mathcal{Q}$  as:

$$(T_Q^\pi f)(s, a) = r(s, a) + \gamma \sum_{s'} p(s' | a, s) \sum_{a' \in \mathcal{A}} \pi(a' | s') f(s', a') , \forall f \in \mathcal{Q}$$

- ▶ This operator has **unique fixed point** which corresponds to the *action-value function*  $q_\pi$  in our MDP  $\mathcal{M}$ .
- ▶ Same properties as  $T^\pi$ :  $\gamma$ -contraction and **monotonicity**.



Similarly for  $q^* : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  functions

### Definition (Bellman Optimality Operator)

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , let  $\mathcal{Q} \equiv \mathcal{Q}_{\mathcal{S}, \mathcal{A}}$  be the space of bounded real-valued functions over  $\mathcal{S} \times \mathcal{A}$ . We define the **Bellman Optimality operator**  $T_{\mathcal{Q}}^* : \mathcal{Q} \rightarrow \mathcal{Q}$  as:

$$(T_{\mathcal{Q}}^* f)(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|a, s) \max_{a' \in \mathcal{A}} f(s', a') , \forall f \in \mathcal{Q}$$

- ▶ This operator has **unique fixed point** which corresponds to the *action-value function*  $q^*$  in our MDP  $\mathcal{M}$ .
- ▶ Same properties as  $T^*$ :  **$\gamma$ -contraction** and **monotonicity**.



# Approximate Dynamic Programming



## Approximate DP

- ▶ So far, we have assumed **perfect knowledge** of the MDP and **perfect/exact representation** of the value functions.
- ▶ Realistically, more often than not:
  - ▶ We **won't know the underlying MDP** (like in the next two lectures)
  - ▶ We **won't be able to represent the value function exactly** after each update (lectures to come)



# Approximate DP

- ▶ Realistically, more often than not:
  - ▶ We won't know the underlying MDP.  
⇒ sampling/estimation error, as we don't have access to the true operators  $T^\pi$  ( $T^*$ )
  - ▶ We won't be able to represent the value function exactly after each update.  
⇒ approximation error, as we approximate the true value functions within a (parametric) class (e.g. linear functions, neural nets, etc).
- ▶ Objective: Under the above conditions, come up with a policy  $\pi$  that is (close to) optimal.



## (Reminder) Value Iteration

### Value Iteration

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = T^*v_k$ .

As  $k \rightarrow \infty$ ,  $v_k \rightarrow_{\|\cdot\|_\infty} v^*$ .



## Approximate Value Iteration

### Approximate Value Iteration

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = \mathcal{A}T^*v_k$ .  $(v_{k+1} \approx T^*v_k)$

Question: As  $k \rightarrow \infty$ ,  $v_k \rightarrow_{\|\cdot\|_\infty} v^*$ ? **X**

Answer: In general, **no**.



## ADP: Approximating the value function

- ▶ Using a **function approximator**  $v_\theta(s)$ , with a parameter vector  $\theta \in \mathbb{R}^m$
- ▶ The estimated value function at iteration  $k$  is  $v_k = v_{\theta_k}$
- ▶ Use dynamic programming to compute  $v_{\theta_{k+1}}$  from  $v_{\theta_k}$

$$T^*v_k(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s]$$

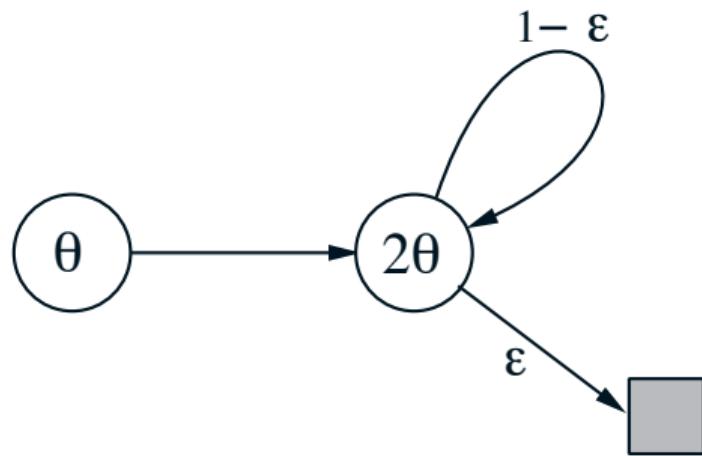
- ▶ Fit  $\theta_{k+1}$  s.t.  $v_{\theta_{k+1}} \approx T^*v_k(s)$ 
  - ▶ For instance, with respect to a squared loss over the state-space.

$$\theta_{k+1} = \arg \min_{\theta_{k+1}} \sum_s (v_{\theta_{k+1}}(s) - T^*v_k(s))^2$$



## Example of divergence with dynamic programming

- ▶ Tsitsiklis and Van Roy made an example where dynamic programming with linear function approximation can diverge. Consider the two state example below, where the rewards are all zero, there are no decisions, and there is a single parameter for estimating the value.

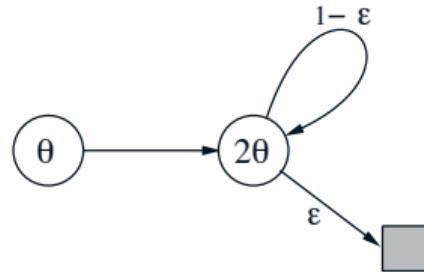


# Blackboard (Tsitsiklis and Van Roy's Example)



## Example of divergence with dynamic programming

- ▶ Tsitsiklis and Van Roy made an example where dynamic programming with linear function approximation can diverge. Consider the two state example below, where the rewards are all zero, there are no decisions, and there is a single parameter for estimating the value.



$$\begin{aligned}\theta_{k+1} &= \operatorname{argmin}_{\theta} \sum_{s \in S} (v_{\theta}(s) - \mathbb{E}[v_{\theta_k}(S_{t+1}) \mid S_t = s])^2 \\ &= \operatorname{argmin}_{\theta} (\theta - \gamma 2\theta_k)^2 + (2\theta - \gamma(1 - \epsilon)2\theta_k)^2 \\ &= \frac{2(3 - 2\epsilon)\gamma}{5} \theta_k\end{aligned}$$

- ▶ What is  $\lim_{k \rightarrow \infty} \theta_k$  when  $\theta_0 = 1$ ,  $\epsilon = \frac{1}{8}$ , and  $\gamma = 1$ ?
- ▶ This is only a problem when we update the states, e.g., synchronously, without looking at the time an agent would spend in each state



# Approximate Value Iteration

## Approximate Value Iteration

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = \mathcal{A}T^*v_k$ .  $(v_{k+1} \approx T^*v_k)$

Question: As  $k \rightarrow \infty$ ,  $v_k \rightarrow_{\|\cdot\|_\infty} v^*$ ? **X**

Answer: In general, **no**.

Hopeless? Not quite!

- ▶ Sample versions of these algorithms converge under mild conditions
- ▶ Even for the function approximation case, the theoretical danger of divergence is rarely materialised in practice
- ▶ There may **many value functions** that can induce the **optimal policy**!



Example from last lecture: Many value functions  $\Rightarrow$  same optimal policy

$k = 3$

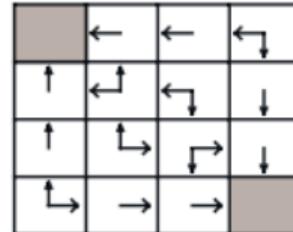
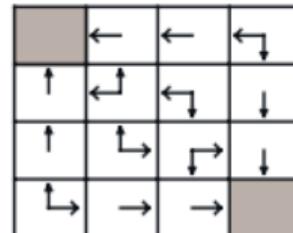
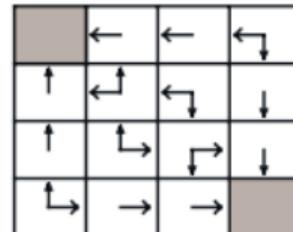
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

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



optimal  
policy



## Performance of a Greedy Policy

### Theorem (Value of greedy policy)

Consider a MDP. Let  $q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  be an arbitrary function and let  $\pi$  be the greedy policy associated with  $q$ , then:

$$\|q^* - q^\pi\|_\infty \leq \frac{2\gamma}{1-\gamma} \|q^* - q\|_\infty$$

where  $q^*$  is the optimal value function associated with this MDP.



## Performance of a Greedy Policy (Proof)

Statement:  $\|q^* - q^\pi\|_\infty \leq \frac{2\gamma}{1-\gamma} \|q^* - q\|_\infty$

Proof.

$$\|q^* - q^\pi\|_\infty = \|q^* - T^\pi q + T^\pi q - q^\pi\|_\infty \quad (15)$$

$$\leq \|q^* - T^\pi q\|_\infty + \|T^\pi q - q^\pi\|_\infty \quad (16)$$

$$= \|T^* q^* - T^* q\|_\infty + \|T^\pi q - T^\pi q^\pi\|_\infty \quad (17)$$

$$\leq \gamma \|q^* - q\|_\infty + \gamma \underbrace{\|q - q^\pi\|_\infty}_{\leq \|q - q^*\|_\infty + \|q^* - q^\pi\|_\infty} \quad (18)$$

$$\leq 2\gamma \|q^* - q\|_\infty + \gamma \|q^* - q^\pi\|_\infty \quad (19)$$

Re-arranging:  $(1 - \gamma) \|q^* - q^\pi\|_\infty \leq 2\gamma \|q^* - q\|_\infty$ . □



## Performance of a Greedy Policy: Test your understanding!

### Theorem (Value of greedy policy)

Consider a MDP. Let  $q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  be an arbitrary function and let  $\pi$  be the greedy policy associated with  $q$ , then:

$$\|q^* - q^\pi\|_\infty \leq \frac{2\gamma}{1-\gamma} \|q^* - q\|_\infty$$

where  $q^*$  is the optimal value function associated with this MDP.

Observations:

- ▶ Small values of  $\gamma$  obtain a better(smaller) upper bound on the potential loss of performance. How do you interpret that?
- ▶ In particular, what happens for  $\gamma = 0$ ? How do you explain this?
- ▶ What if  $q = q^*$ ? What does this bound imply in that case?



# (Reminder) Policy Iteration

## Policy Iteration

- ▶ Start with  $\pi_0$ .
- ▶ Iterate:
  - ▶ Policy Evaluation:  $q_i = q_{\pi_i}$
  - ▶ Greedy Improvement:  $\pi_{i+1} = \arg \max_a q_{\pi_i}(s, a)$

As  $i \rightarrow \infty$ ,  $q_i \rightarrow_{\|\cdot\|_\infty} q^*$ . Thus  $\pi_i \rightarrow \pi^*$ .



# Approximate Policy Iteration

## Approximate Policy Iteration

- ▶ Start with  $\pi_0$ .
- ▶ Iterate:
  - ▶ Policy Evaluation:  $q_i = \mathcal{A}q_{\pi_i}$   $(q_i \approx q_{\pi_i})$
  - ▶ Greedy Improvement:  $\pi_{i+1} = \arg \max_a q_i(s, a)$

Question 1: As  $i \rightarrow \infty$ , does  $q_i \rightarrow_{\|\cdot\|_\infty} q^*$ ? **X**

Answer: In general, **no**.

Question 2: Or does  $\pi_i$  converge to the optimal policy? **X**

Answer: In general, **no**.

Hopeless? In some cases, **no**, depending on the nature of  $\mathcal{A}$ . (More: Next lectures)



# (Summary) Approximate Dynamic Programming

## Approximate Value Iteration

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = \mathcal{A}T^*v_k$ .  $(v_{k+1} \approx T^*v_k)$

## Approximate Policy Iteration

- ▶ Start with  $\pi_0$ .
- ▶ Iterate:
  - ▶ Policy Evaluation:  $q_i = \mathcal{A}q_{\pi_i}$   $(q_i \approx q_{\pi_i})$
  - ▶ Greedy Improvement:  $\pi_{i+1} = \arg \max_a q_i(s, a)$



## Questions?

*The only stupid question is the one you were afraid to ask but never did.*  
-Rich Sutton

For questions that may arise during this lecture please use Moodle and/or the next Q&A session.



# Lecture 5: Model-Free Prediction

Hado van Hasselt

UCL, 2021



# Background

Sutton & Barto 2018, Chapters 5 + 6 + 7 + 9 +12

Don't worry about reading all of this at once!

Most important chapters, for now: 5 + 6

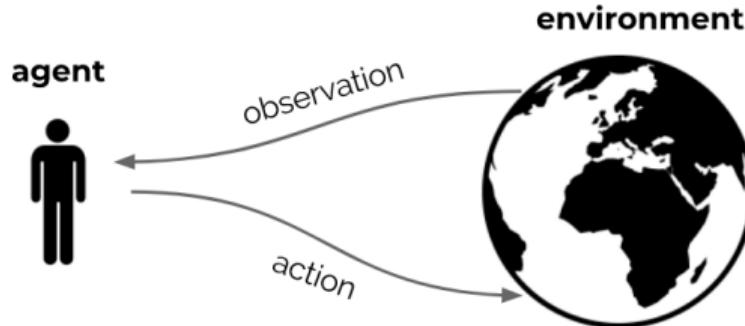
You can also defer some reading, e.g., until the reading week



Don't forget to pause



# Recap



- ▶ Reinforcement learning is the science of learning to make decisions
- ▶ Agents can learn a **policy**, **value function** and/or a **model**
- ▶ The general problem involves taking into account **time** and **consequences**
- ▶ Decisions affect the **reward**, the **agent state**, and **environment state**



# Lecture overview

- ▶ Last lectures (3+4):
  - ▶ Planning by **dynamic programming** to solve a known MDP
- ▶ This and next lectures (5→8):
  - ▶ **Model-free prediction** to **estimate** values in an **unknown** MDP
  - ▶ **Model-free control** to **optimise** values in an **unknown** MDP
  - ▶ **Function approximation** and (some) **deep reinforcement learning** (but more to follow later)
  - ▶ **Off-policy** learning
- ▶ Later lectures:
  - ▶ Model-based learning and planning
  - ▶ Policy gradients and actor critic systems
  - ▶ More deep reinforcement learning
  - ▶ More advanced topics and current research



# Model-Free Prediction: Monte Carlo Algorithms



# Monte Carlo Algorithms

- ▶ We can use experience **samples** to learn without a model
- ▶ We call direct sampling of episodes **Monte Carlo**
- ▶ MC is **model-free**: no knowledge of MDP required, only samples



# Monte Carlo: Bandits

- ▶ Simple example, **multi-armed bandit**:
  - ▶ For each action, average reward samples

$$q_t(a) = \frac{\sum_{i=0}^t \mathcal{I}(A_i = a) R_{i+1}}{\sum_{i=0}^t \mathcal{I}(A_i = a)} \approx \mathbb{E}[R_{t+1}|A_t = a] = q(a)$$

- ▶ Equivalently:

$$\begin{aligned} q_{t+1}(A_t) &= q_t(A_t) + \alpha_t(R_{t+1} - q_t(A_t)) \\ q_{t+1}(a) &= q_t(a) \quad \forall a \neq A_t \end{aligned}$$

$$\text{with } \alpha_t = \frac{1}{N_t(A_t)} = \frac{1}{\sum_{i=0}^t \mathcal{I}(A_i = a)}$$

- ▶ Note: we changed notation  $R_t \rightarrow R_{t+1}$  for the reward after  $A_t$   
In MDPs, the reward is said to arrive on the time step after the action



## Monte Carlo: Bandits with States

- ▶ Consider bandits with different states
  - ▶ episodes are still one step
  - ▶ actions do not affect state transitions
  - ▶  $\implies$  no long-term consequences
- ▶ Then, we want to estimate

$$q(s, a) = \mathbb{E} [R_{t+1} | S_t = s, A_t = a]$$

- ▶ These are called **contextual bandits**



# Introduction Function Approximation



# Value Function Approximation

- ▶ So far we mostly considered **lookup tables**
  - ▶ Every state  $s$  has an entry  $v(s)$
  - ▶ Or every state-action pair  $s, a$  has an entry  $q(s, a)$
- ▶ Problem with large MDPs:
  - ▶ There are too many states and/or actions to store in memory
  - ▶ It is too slow to learn the value of each state individually
  - ▶ Individual states are often **not fully observable**



# Value Function Approximation

Solution for large MDPs:

- ▶ Estimate value function with **function approximation**

$$v_{\mathbf{w}}(s) \approx v_{\pi}(s)$$

(or  $v_*(s)$ )

$$q_{\mathbf{w}}(s, a) \approx q_{\pi}(s, a)$$

(or  $q_*(s, a)$ )

- ▶ Update parameter  $\mathbf{w}$  (e.g., using MC or TD learning)
- ▶ Generalise from to unseen states



## Agent state update

Solution for large MDPs, if the environment state is not fully observable

- ▶ Use the **agent state**:

$$S_t = u_{\omega}(S_{t-1}, A_{t-1}, O_t)$$

with parameters  $\omega$  (typically  $\omega \in \mathbb{R}^n$ )

- ▶ Henceforth,  $S_t$  denotes the agent state
- ▶ Think of this as either a vector inside the agent,  
or, in the simplest case, just the current observation:  $S_t = O_t$
- ▶ For now we are **not** going to talk about how to learn the agent state update
- ▶ Feel free to consider  $S_t$  an observation



# Linear Function Approximation



## Feature Vectors

- ▶ A useful special case: **linear functions**
- ▶ Represent state by a **feature vector**

$$\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_m(s) \end{pmatrix}$$

- ▶  $\mathbf{x} : \mathcal{S} \rightarrow \mathbb{R}^m$  is a fixed mapping from agent state (e.g., observation) to features
- ▶ Short-hand:  $\mathbf{x}_t = \mathbf{x}(S_t)$
- ▶ For example:
  - ▶ Distance of robot from landmarks
  - ▶ Trends in the stock market
  - ▶ Piece and pawn configurations in chess



# Linear Value Function Approximation

- ▶ Approximate value function by a linear combination of features

$$v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{j=1}^n x_j(s) w_j$$

- ▶ Objective function ('loss') is quadratic in  $\mathbf{w}$

$$L(\mathbf{w}) = \mathbb{E}_{S \sim d}[(v_{\pi}(S) - \mathbf{w}^\top \mathbf{x}(S))^2]$$

- ▶ Stochastic gradient descent converges on **global** optimum
- ▶ Update rule is simple

$$\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) = \mathbf{x}(S_t) = \mathbf{x}_t \quad \implies \quad \Delta \mathbf{w} = \alpha(v_{\pi}(S_t) - v_{\mathbf{w}}(S_t)) \mathbf{x}_t$$

Update = **step-size** × **prediction error** × **feature vector**



## Table Lookup Features

- ▶ Table lookup is a special case of linear value function approximation
- ▶ Let the  $n$  states be given by  $\mathcal{S} = \{s_1, \dots, s_n\}$ .
- ▶ Use one-hot feature:

$$\mathbf{x}(s) = \begin{pmatrix} \mathcal{I}(s = s_1) \\ \vdots \\ \mathcal{I}(s = s^n) \end{pmatrix}$$

- ▶ Parameters  $\mathbf{w}$  then just contains value estimates for each state

$$v(s) = \mathbf{w}^\top \mathbf{x}(s) = \sum_j w_j x_j(s) = w_s .$$



# Model-Free Prediction: Monte Carlo Algorithms

(Continuing from before...)



## Monte Carlo: Bandits with States

- ▶  $q$  could be a parametric function, e.g., neural network, and we could use loss

$$L(\mathbf{w}) = \frac{1}{2} \mathbb{E} [(R_{t+1} - q_{\mathbf{w}}(S_t, A_t))^2]$$

- ▶ Then the gradient update is

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \alpha \nabla_{\mathbf{w}_t} L(\mathbf{w}_t) \\ &= \mathbf{w}_t - \alpha \nabla_{\mathbf{w}_t} \frac{1}{2} \mathbb{E} [(R_{t+1} - q_{\mathbf{w}_t}(S_t, A_t))^2] \\ &= \mathbf{w}_t + \alpha \mathbb{E} [(R_{t+1} - q_{\mathbf{w}_t}(S_t, A_t)) \nabla_{\mathbf{w}_t} q_{\mathbf{w}_t}(S_t, A_t)] .\end{aligned}$$

We can sample this to get a **stochastic gradient update** (SGD)

- ▶ The tabular case is a special case (only updates the value in cell  $[S_t, A_t]$ )
- ▶ Also works for large (continuous) state spaces  $\mathcal{S}$  — this is just **regression**



## Monte Carlo: Bandits with States

- ▶ When using linear functions,  $q(s, a) = \mathbf{w}^\top \mathbf{x}(s, a)$  and

$$\nabla_{\mathbf{w}_t} q_{\mathbf{w}_t}(S_t, A_t) = \mathbf{x}(s, a)$$

- ▶ Then the SGD update is

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(R_{t+1} - q_{\mathbf{w}_t}(S_t, A_t))\mathbf{x}(s, a).$$

- ▶ Linear update = **step-size** × **prediction error** × **feature vector**
- ▶ Non-linear update = **step-size** × **prediction error** × **gradient**



# Monte-Carlo Policy Evaluation

- ▶ Now we consider **sequential decision problems**
- ▶ Goal: learn  $v_\pi$  from episodes of experience under policy  $\pi$

$$S_1, A_1, R_2, \dots, S_k \sim \pi$$

- ▶ The **return** is the total discounted reward (for an episode ending at time  $T > t$ ):

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T$$

- ▶ The value function is the expected return:

$$v_\pi(s) = \mathbb{E}[G_t \mid S_t = s, \pi]$$

- ▶ We can just use **sample average** return instead of **expected** return
- ▶ We call this **Monte Carlo policy evaluation**



# Example: Blackjack



# Blackjack Example

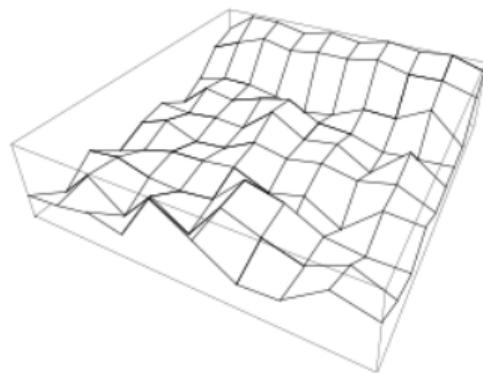
- ▶ States (200 of them):
  - ▶ Current sum (12-21)
  - ▶ Dealer's showing card (ace-10)
  - ▶ Do I have a "useable" ace? (yes-no)
- ▶ Action **stick**: Stop receiving cards (and terminate)
- ▶ Action **draw**: Take another card (random, no replacement)
- ▶ Reward for **stick**:
  - ▶ +1 if sum of cards > sum of dealer cards
  - ▶ 0 if sum of cards = sum of dealer cards
  - ▶ -1 if sum of cards < sum of dealer cards
- ▶ Reward for **draw**:
  - ▶ -1 if sum of cards > 21 (and terminate)
  - ▶ 0 otherwise
- ▶ Transitions: automatically **draw** if sum of cards < 12



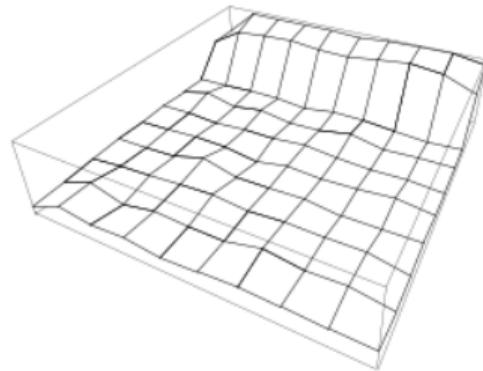
# Blackjack Value Function after Monte-Carlo Learning

After 10,000 episodes

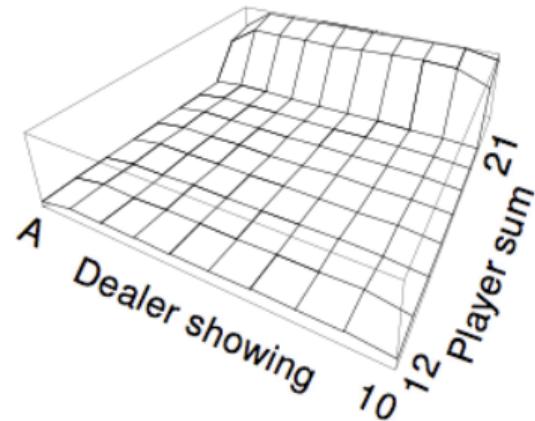
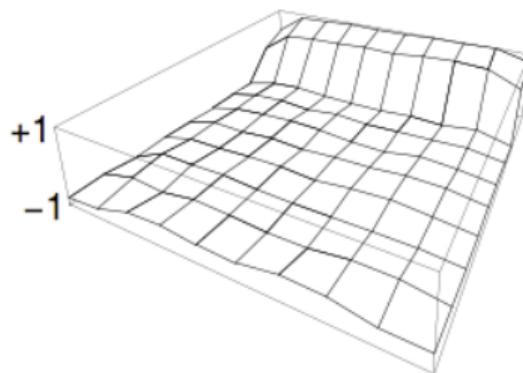
Usable  
ace



No  
usable  
ace



After 500,000 episodes



# Disadvantages of Monte-Carlo Learning

- ▶ We have seen MC algorithms can be used to learn value predictions
- ▶ But when episodes are long, learning can be slow
  - ▶ ...we have to wait until an episode ends before we can learn
  - ▶ ...return can have high variance
- ▶ Are there alternatives? (Spoiler: yes)



# Temporal-Difference Learning



# Temporal Difference Learning by Sampling Bellman Equations

- ▶ Previous lecture: Bellman equations,

$$v_\pi(s) = \mathbb{E} [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t \sim \pi(S_t)]$$

- ▶ Previous lecture: Approximate by iterating,

$$v_{k+1}(s) = \mathbb{E} [R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t \sim \pi(S_t)]$$

- ▶ We can sample this!

$$v_{t+1}(S_t) = R_{t+1} + \gamma v_t(S_{t+1})$$

- ▶ This is likely quite noisy — better to take a small step (with parameter  $\alpha$ ):

$$v_{t+1}(S_t) = v_t(S_t) + \alpha_t \left( \underbrace{R_{t+1} + \gamma v_t(S_{t+1}) - v_t(S_t)}_{\text{target}} \right)$$

(Note: tabular update)



# Temporal difference learning

- ▶ **Prediction** setting: learn  $v_\pi$  online from experience under policy  $\pi$
- ▶ Monte-Carlo
  - ▶ Update value  $v_n(S_t)$  towards sampled return  $\mathbf{G}_t$

$$v_{n+1}(S_t) = v_n(S_t) + \alpha (\mathbf{G}_t - v_n(S_t))$$

- ▶ Temporal-difference learning:
  - ▶ Update value  $v_t(S_t)$  towards estimated return  $\mathbf{R}_{t+1} + \gamma v(S_{t+1})$

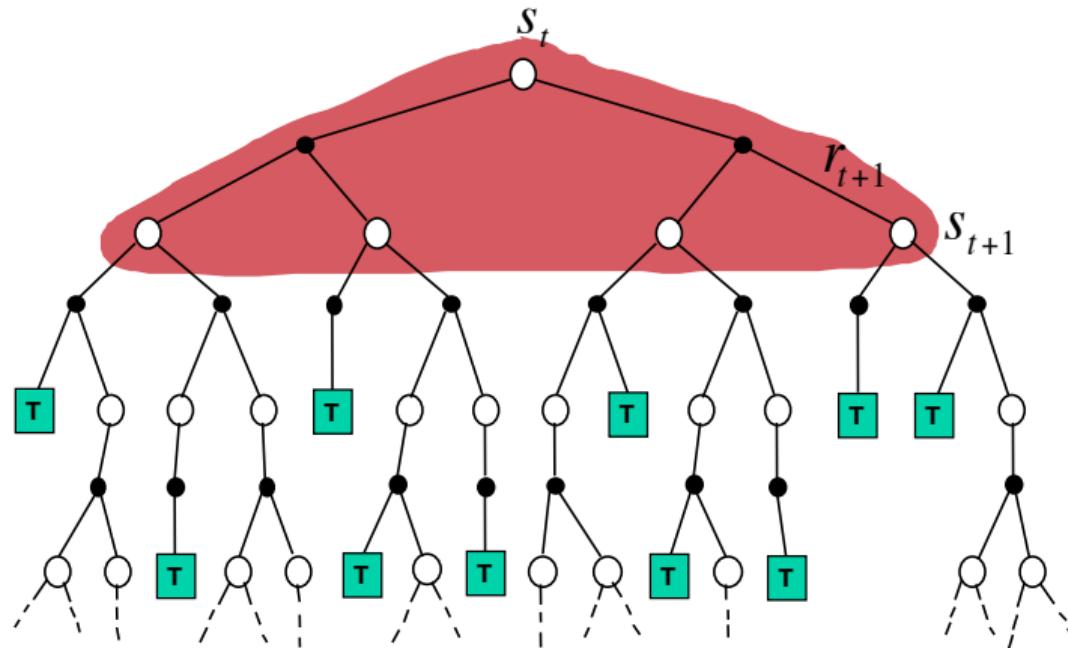
$$v_{t+1}(S_t) \leftarrow v_t(S_t) + \alpha \left( \underbrace{\mathbf{R}_{t+1} + \gamma v_t(S_{t+1}) - v_t(S_t)}_{\text{target}} \right)$$

- ▶  $\delta_t = \mathbf{R}_{t+1} + \gamma v_t(S_{t+1}) - v_t(S_t)$  is called the TD error



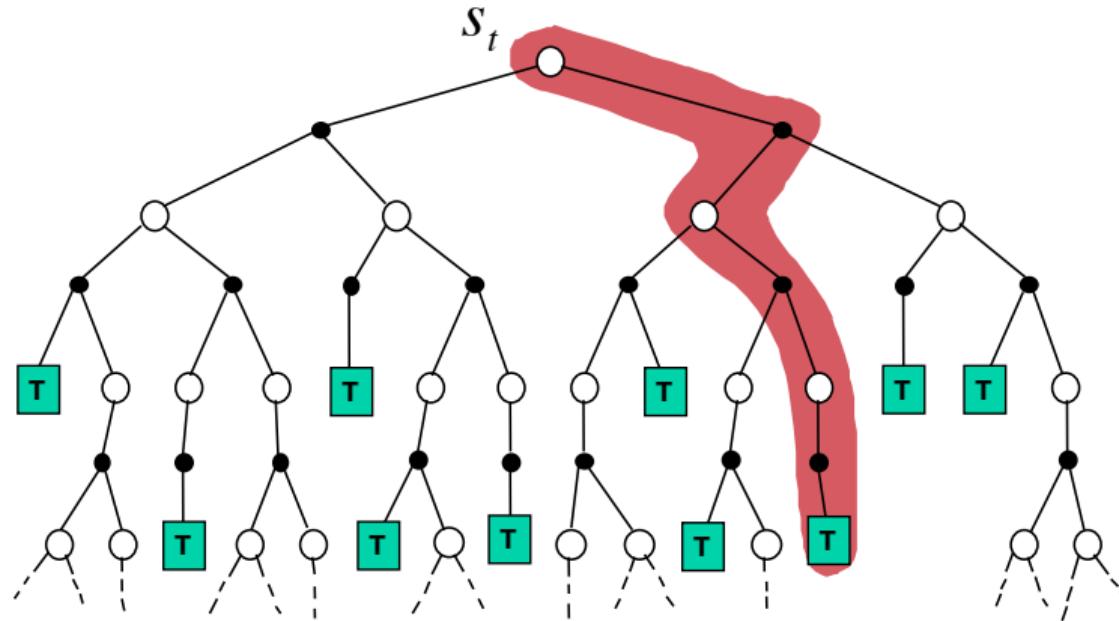
# Dynamic Programming Backup

$$v(S_t) \leftarrow \mathbb{E} [R_{t+1} + \gamma v(S_{t+1}) \mid A_t \sim \pi(S_t)]$$



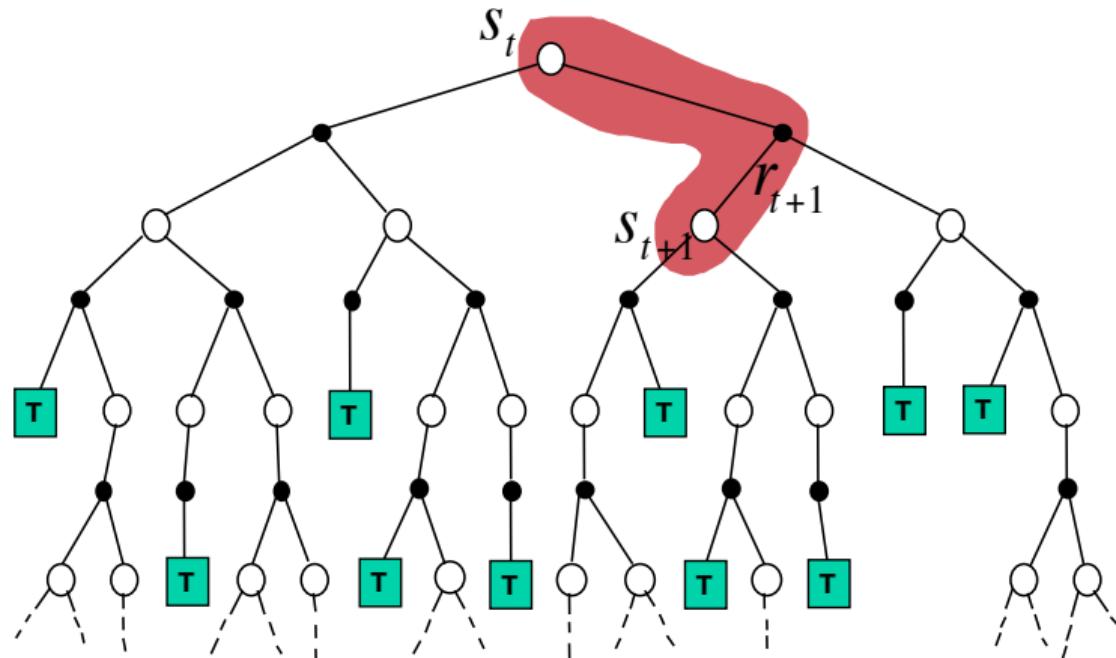
# Monte-Carlo Backup

$$v(S_t) \leftarrow v(S_t) + \alpha (G_t - v(S_t))$$



# Temporal-Difference Backup

$$v(S_t) \leftarrow v(S_t) + \alpha (R_{t+1} + \gamma v(S_{t+1}) - v(S_t))$$



# Bootstrapping and Sampling

- ▶ **Bootstrapping:** update involves an estimate
  - ▶ MC does not bootstrap
  - ▶ DP bootstraps
  - ▶ TD bootstraps
- ▶ **Sampling:** update samples an expectation
  - ▶ MC samples
  - ▶ DP does not sample
  - ▶ TD samples



# Temporal difference learning

- ▶ We can apply the same idea to **action values**
- ▶ Temporal-difference learning for action values:
  - ▶ Update value  $q_t(S_t, A_t)$  towards estimated return  $R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$

$$q_{t+1}(S_t, A_t) \leftarrow q_t(S_t, A_t) + \alpha \left( \underbrace{R_{t+1} + \gamma q_t(S_{t+1}, A_{t+1}) - q_t(S_t, A_t)}_{\text{TD error}} \right)$$

- ▶ This algorithm is known as **SARSA**, because it uses  $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$



# Temporal-Difference Learning

- ▶ TD is **model-free** (no knowledge of MDP) and learn directly from experience
- ▶ TD can learn from **incomplete** episodes, by **bootstrapping**
- ▶ TD can learn **during** each episode



# Example: Driving Home



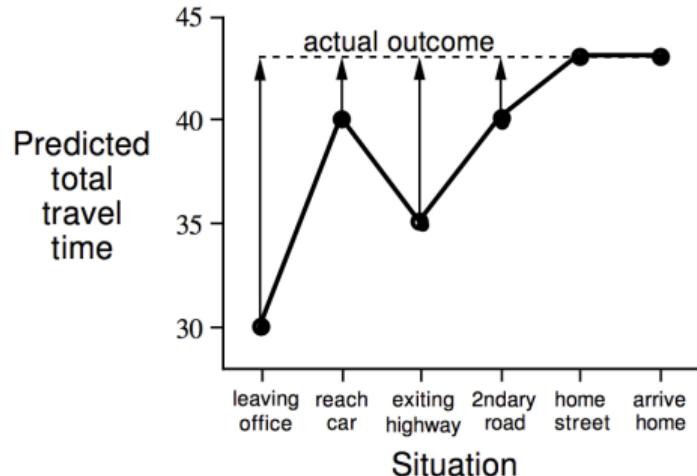
## Driving Home Example

<b>State</b>	<b>Elapsed Time (minutes)</b>	<b>Predicted Time to Go</b>	<b>Predicted Total Time</b>
leaving office	0	30	30
reach car, raining	5	35	40
exit highway	20	15	35
behind truck	30	10	40
home street	40	3	43
arrive home	43	0	43

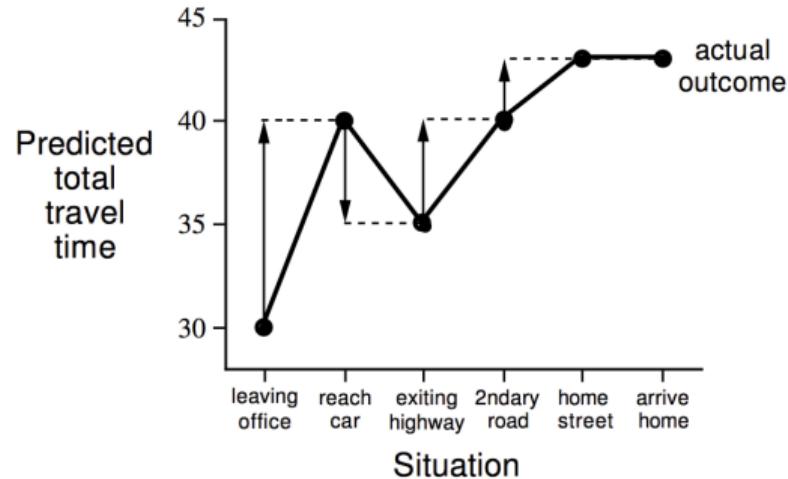


# Driving Home Example: MC vs. TD

Changes recommended by Monte Carlo methods ( $\alpha=1$ )



Changes recommended by TD methods ( $\alpha=1$ )



# Comparing MC and TD



# Advantages and Disadvantages of MC vs. TD

- ▶ TD can learn **before** knowing the final outcome
  - ▶ TD can learn online after every step
  - ▶ MC must wait until end of episode before return is known
- ▶ TD can learn **without** the final outcome
  - ▶ TD can learn from incomplete sequences
  - ▶ MC can only learn from complete sequences
  - ▶ TD works in continuing (non-terminating) environments
  - ▶ MC only works for episodic (terminating) environments
- ▶ TD is **independent of the temporal span** of the prediction
  - ▶ TD can learn from single transitions
  - ▶ MC must store all predictions (or states) to update at the end of an episode
- ▶ TD needs reasonable value estimates



## Bias/Variance Trade-Off

- ▶ MC return  $G_t = R_{t+1} + \gamma R_{t+2} + \dots$  is an **unbiased** estimate of  $v_\pi(S_t)$
- ▶ TD target  $R_{t+1} + \gamma v_t(S_{t+1})$  is a **biased** estimate of  $v_\pi(S_t)$  (unless  $v_t(S_{t+1}) = v_\pi(S_{t+1})$ )
- ▶ But the TD target has **lower variance**:
  - ▶ Return depends on **many** random actions, transitions, rewards
  - ▶ TD target depends on **one** random action, transition, reward



## Bias/Variance Trade-Off

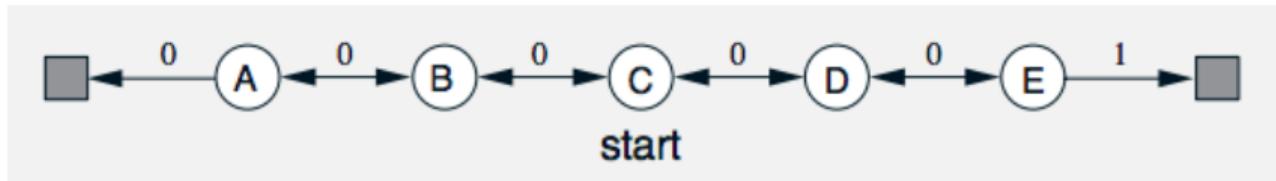
- ▶ In some cases, TD can have irreducible bias
- ▶ The world may be partially observable
  - ▶ MC would implicitly account for all the latent variables
- ▶ The function to approximate the values may fit poorly
- ▶ In the tabular case, both MC and TD will converge:  $v_t \rightarrow v_\pi$



# Example: Random Walk



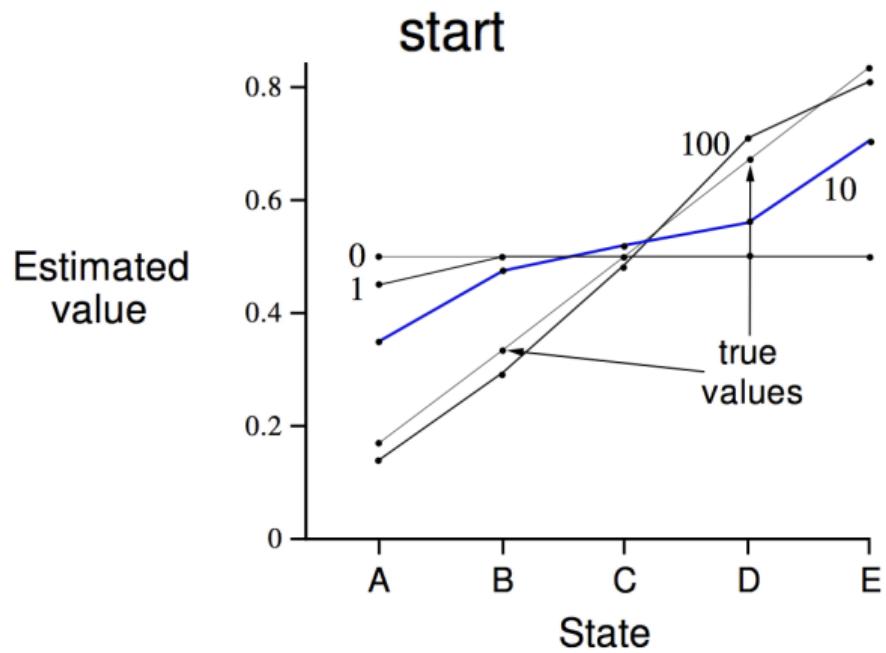
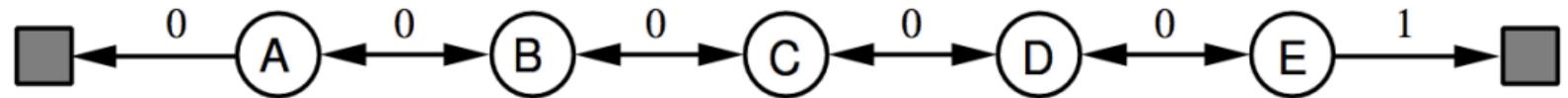
## Random Walk Example



- ▶ Uniform random transitions (50% left, 50% right)
- ▶ Initial values are  $v(s) = 0.5$ , for all  $s$
- ▶ True values happen to be  
 $v(A) = \frac{1}{6}$ ,  $v(B) = \frac{2}{6}$ ,  $v(C) = \frac{3}{6}$ ,  $v(D) = \frac{4}{6}$ ,  $v(E) = \frac{5}{6}$

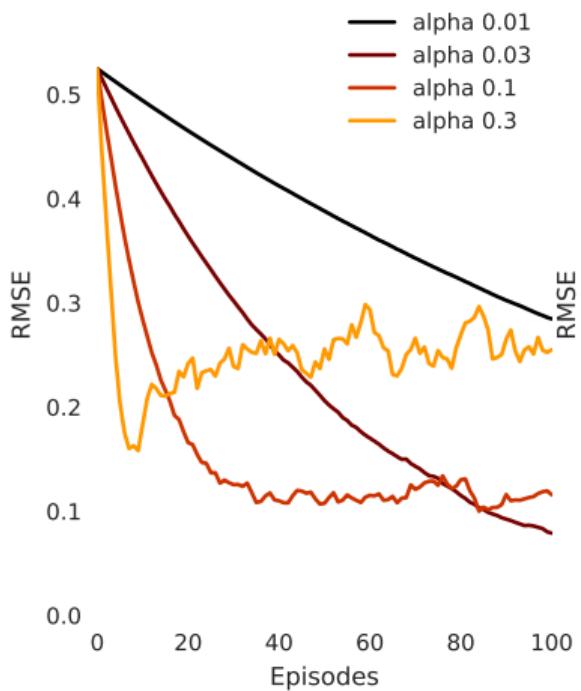


## Random Walk Example

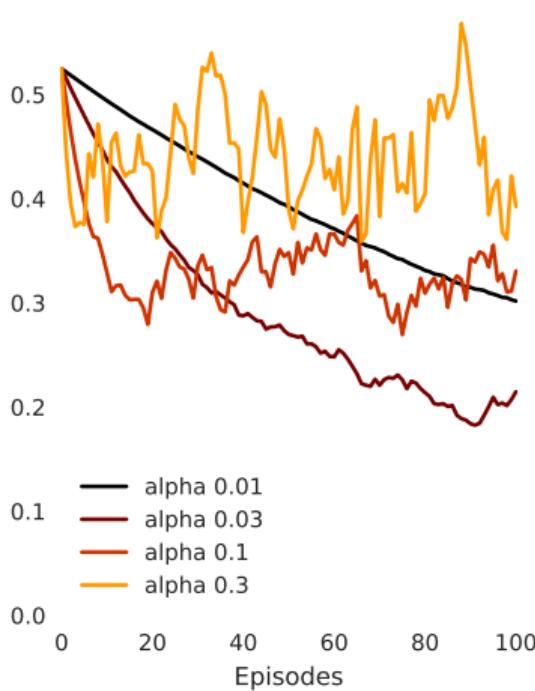


# Random Walk: MC vs. TD

TD



MC



# Batch MC and TD



## Batch MC and TD

- ▶ Tabular MC and TD converge:  $v_t \rightarrow v_\pi$  as experience  $\rightarrow \infty$  and  $\alpha_t \rightarrow 0$
- ▶ But what about finite experience?
- ▶ Consider a fixed batch of experience:

episode 1:  $S_1^1, A_1^1, R_2^1, \dots, S_{T_1}^1$

⋮

episode K:  $S_1^K, A_1^K, R_2^K, \dots, S_{T_K}^K$

- ▶ Repeatedly sample each episode  $k \in [1, K]$  and apply MC or TD(0)
  - ▶ = sampling from an **empirical model**



# Example: Batch Learning in Two States



## Example: Batch Learning in Two States

Two states  $A, B$ ; no discounting; 8 episodes of experience

A, 0, B, 0

B, 1

B, 1

B, 1

B, 1

B, 1

B, 1

B, 0

What is  $v(A), v(B)$ ?



## Example: Batch Learning in Two States

Two states  $A, B$ ; no discounting; 8 episodes of experience

A, 0, B, 0

B, 1

B, 1

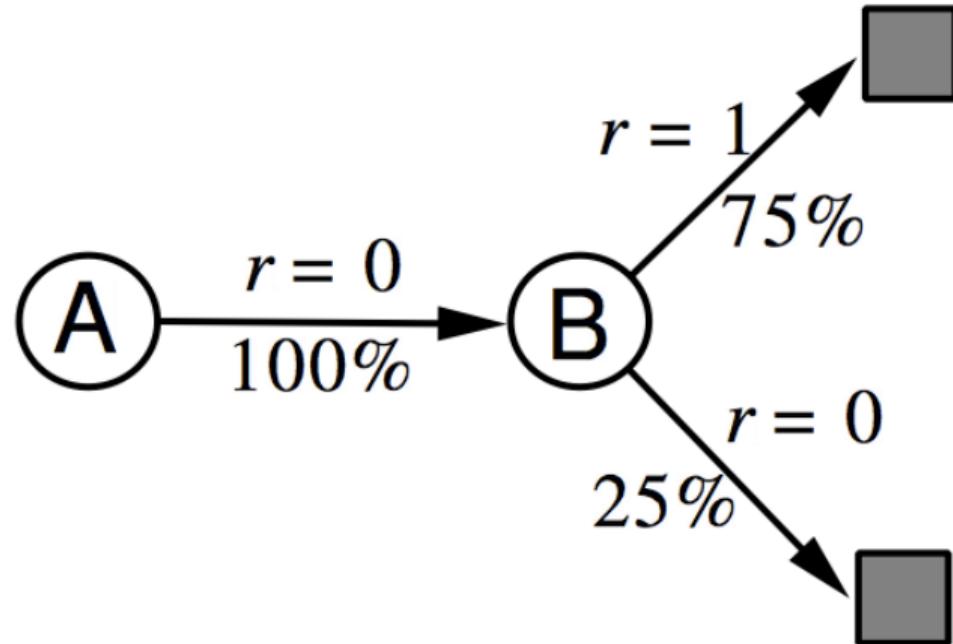
B, 1

B, 1

B, 1

B, 1

B, 0



What is  $v(A), v(B)$ ?



## Differences in batch solutions

- ▶ MC converges to best mean-squared fit for the observed returns

$$\sum_{k=1}^K \sum_{t=1}^{T_k} \left( G_t^k - v(S_t^k) \right)^2$$

- ▶ In the AB example,  $v(A) = 0$
- ▶ TD converges to solution of max likelihood Markov model, given the data
  - ▶ Solution to the empirical MDP  $(\mathcal{S}, \mathcal{A}, \hat{p}, \gamma)$  that best fits the data
  - ▶ In the AB example:  $\hat{p}(S_{t+1} = B \mid S_t = A) = 1$ , and therefore  $v(A) = v(B) = 0.75$



## Advantages and Disadvantages of MC vs. TD

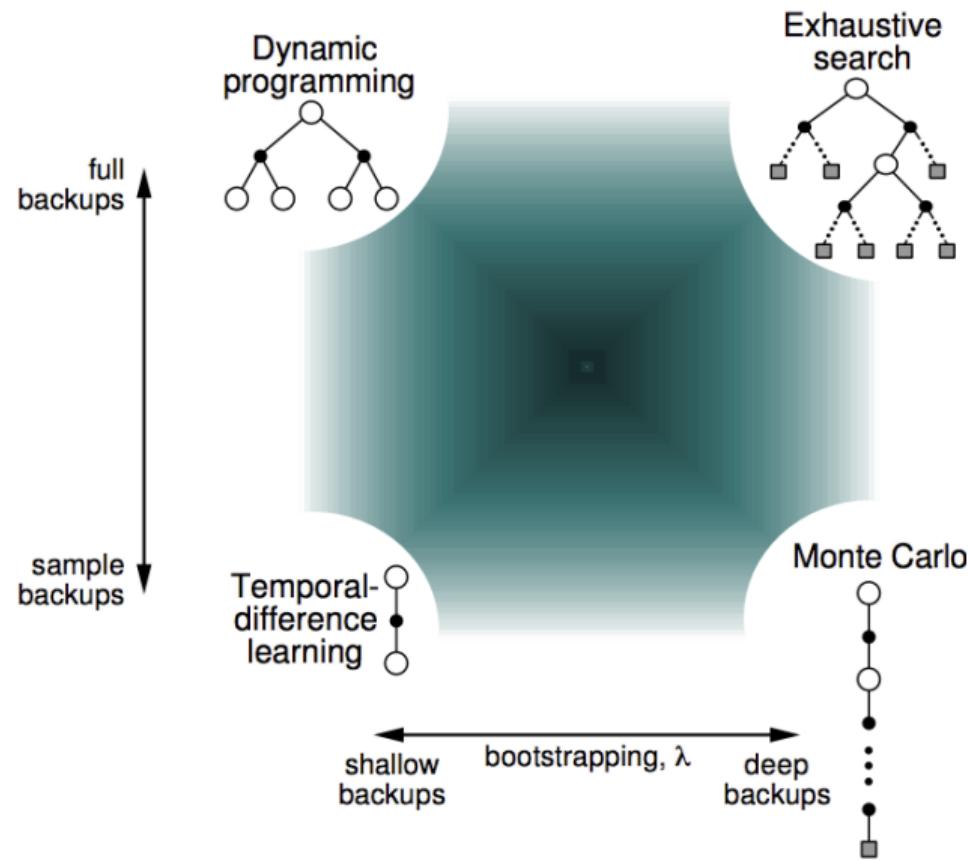
- ▶ TD exploits Markov property
  - ▶ Can help in fully-observable environments
- ▶ MC does not exploit Markov property
  - ▶ Can help in partially-observable environments
- ▶ With finite data, or with function approximation, **the solutions may differ**



# Between MC and TD: Multi-Step TD



# Unified View of Reinforcement Learning



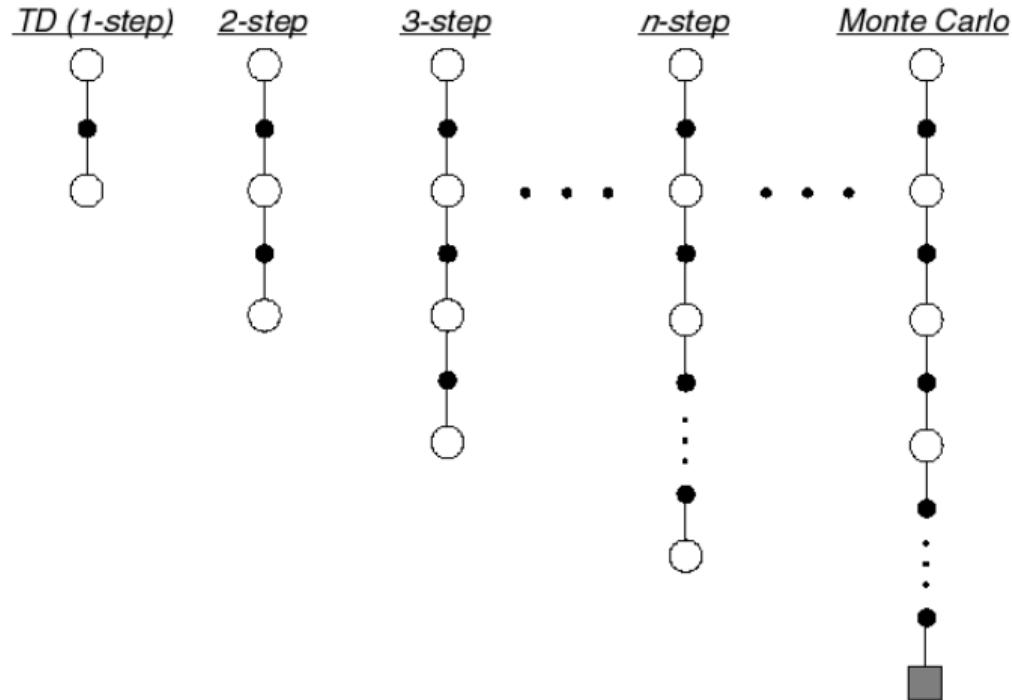
## Multi-Step Updates

- ▶ TD uses value estimates which might be inaccurate
- ▶ In addition, information can propagate back quite slowly
- ▶ In MC information propagates faster, but the updates are noisier
- ▶ We can go in between TD and MC



# Multi-Step Prediction

- Let TD target look  $n$  steps into the future



## Multi-Step Returns

- ▶ Consider the following  $n$ -step returns for  $n = 1, 2, \infty$ :

$$\begin{aligned} n = 1 & \quad (\text{TD}) \quad G_t^{(1)} = R_{t+1} + \gamma v(S_{t+1}) \\ n = 2 & \quad G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 v(S_{t+2}) \\ & \vdots \quad \vdots \\ n = \infty & \quad (\text{MC}) \quad G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T \end{aligned}$$

- ▶ In general, the  $n$ -step return is defined by

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n v(S_{t+n})$$

- ▶ Multi-step temporal-difference learning

$$v(S_t) \leftarrow v(S_t) + \alpha \left( G_t^{(n)} - v(S_t) \right)$$

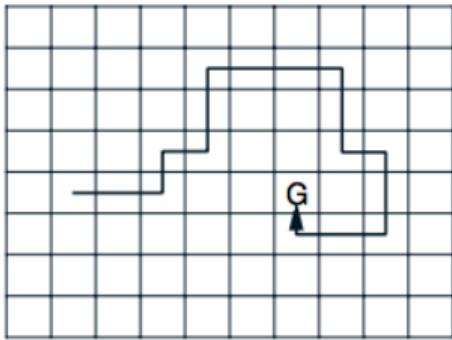


# Multi-Step Examples

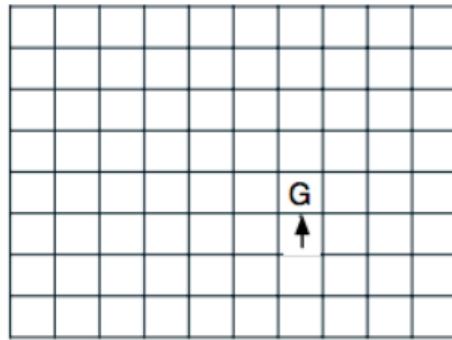


# Grid Example

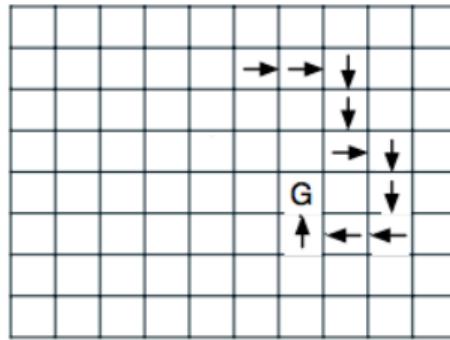
Path taken



Action values increased  
by one-step Sarsa



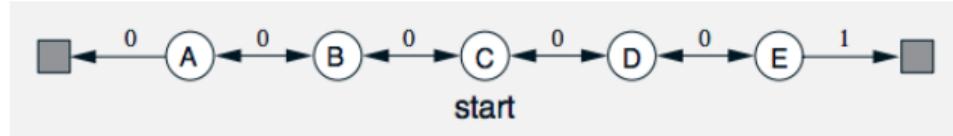
Action values increased  
by 10-step Sarsa



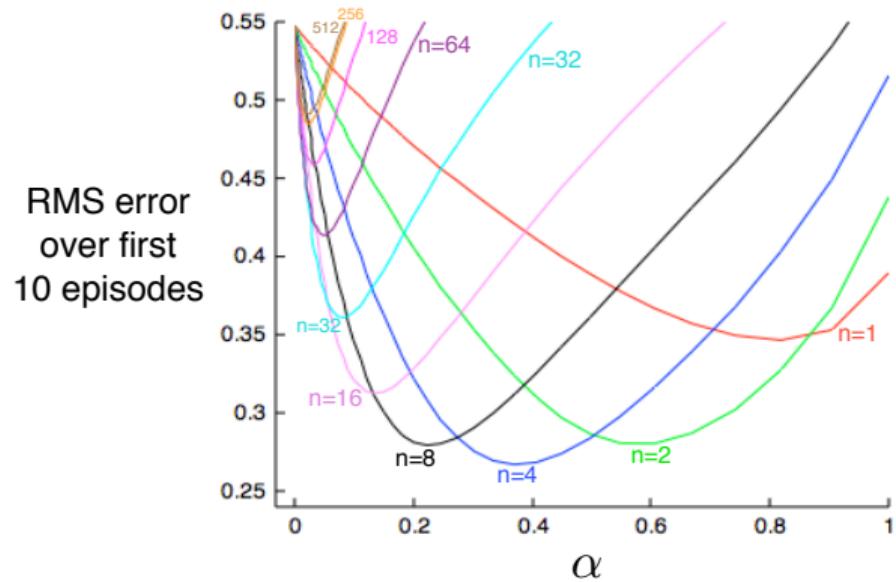
(Reminder: SARSA is TD for action values  $q(s, a)$ )



# Large Random Walk Example



..., but with 19 states, rather than 5



# Mixed Multi-Step Returns



## Mixing multi-step returns

- ▶ Multi-step returns bootstrap on one state,  $v(S_{t+n})$ :

$$G_t^{(n)} = R_{t+1} + \gamma G_{t+1}^{(n-1)} \quad (\text{while } n > 1, \text{ continue})$$
$$G_t^{(1)} = R_{t+1} + \gamma v(S_{t+1}). \quad (\text{truncate \& bootstrap})$$

- ▶ You can also bootstrap a little bit on multiple states:

$$G_t^\lambda = R_{t+1} + \gamma \left( (1 - \lambda)v(S_{t+1}) + \lambda G_{t+1}^\lambda \right)$$

This gives a weighted average of  $n$ -step returns:

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

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



## Mixing multi-step returns

$$G_t^\lambda = R_{t+1} + \gamma \left( (1 - \lambda)v(S_{t+1}) + \lambda G_{t+1}^\lambda \right)$$

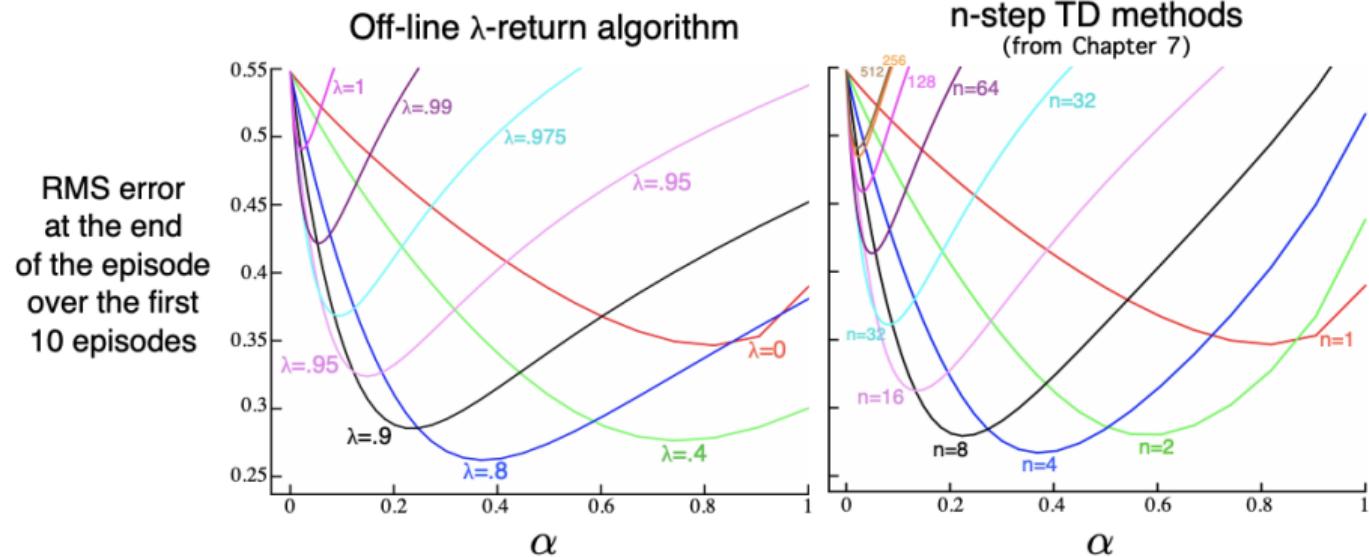
Special cases:

$$G_t^{\lambda=0} = R_{t+1} + \gamma v(S_{t+1}) \quad (\text{TD})$$

$$G_t^{\lambda=1} = R_{t+1} + \gamma G_{t+1} \quad (\text{MC})$$



# Mixing multi-step returns



Intuition:  $1/(1 - \lambda)$  is the 'horizon'. E.g.,  $\lambda = 0.9 \approx n = 10$ .



# Benefits of Multi-Step Learning



## Benefits of multi-step returns

- ▶ Multi-step returns have benefits from both TD and MC
- ▶ Bootstrapping can have issues with **bias**
- ▶ Monte Carlo can have issues with **variance**
- ▶ Typically, intermediate values of  $n$  or  $\lambda$  are good (e.g.,  $n = 10, \lambda = 0.9$ )



# Eligibility Traces



## Independence of temporal span

- ▶ MC and multi-step returns are not **independent of span** of the predictions:  
To update values in a long episode, you have to wait
- ▶ TD can update immediately, and is independent of the span of the predictions
- ▶ Can we get both?



# Eligibility traces

- ▶ Recall **linear function approximation**
- ▶ The Monte Carlo and TD updates to  $v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s)$  for a state  $s = S_t$  is

$$\Delta \mathbf{w}_t = \alpha(G_t - v(S_t))\mathbf{x}_t \quad (\text{MC})$$

$$\Delta \mathbf{w}_t = \alpha(R_{t+1} + \gamma v(S_{t+1}) - v(S_t))\mathbf{x}_t \quad (\text{TD})$$

- ▶ MC updates all states in episode  $k$  at once:

$$\Delta \mathbf{w}_{k+1} = \sum_{t=0}^{T-1} \alpha(G_t - v(S_t))\mathbf{x}_t$$

where  $t \in \{0, \dots, T-1\}$  enumerate the time steps in this specific episode

- ▶ Recall: tabular is a special case, with one-hot vector  $\mathbf{x}_t$



## Eligibility traces

- ▶ Accumulating a whole episode of updates:

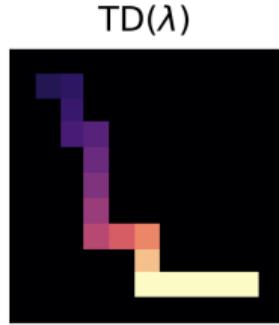
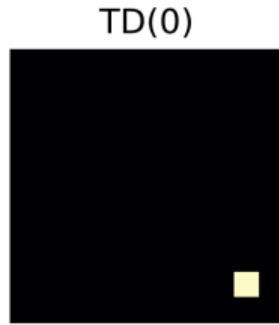
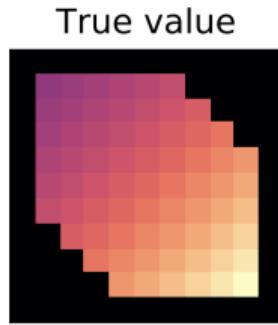
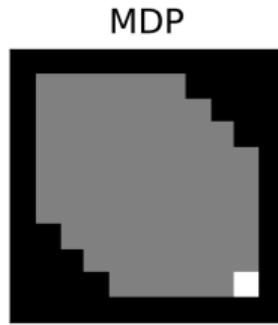
$$\Delta \mathbf{w}_t \equiv \alpha \delta_t \mathbf{e}_t \quad (\text{one time step})$$

$$\text{where } \mathbf{e}_t = \gamma \lambda \mathbf{e}_{t-1} + \mathbf{x}_t$$

- ▶ Note: if  $\lambda = 0$ , we get one-step TD
- ▶ Intuition: decay the **eligibility** of past states for the current TD error, then add it
- ▶ This is kind of magical: we can update **all past states** (to account for the new TD error) with a single update! No need to recompute their values.
- ▶ This idea extends to function approximation:  $\mathbf{x}_t$  does not have to be one-hot



# Eligibility traces



## Eligibility traces

We can rewrite the MC error as a sum of TD errors:

$$\begin{aligned} G_t - v(S_t) &= R_{t+1} + \gamma G_{t+1} - v(S_t) \\ &= \underbrace{R_{t+1} + \gamma v(S_{t+1}) - v(S_t)}_{= \delta_t} + \gamma(G_{t+1} - v(S_{t+1})) \\ &= \delta_t + \gamma(G_{t+1} - v(S_{t+1})) \\ &= \dots \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - v(S_{t+2})) \\ &= \dots \\ &= \sum_{k=t}^T \gamma^{k-t} \delta_k \end{aligned}$$

(used in the next slide)



## Eligibility traces

- Now consider accumulating a whole episode (from time  $t = 0$  to  $T$ ) of updates:

$$\begin{aligned}\Delta \mathbf{w}_k &= \sum_{t=0}^{T-1} \alpha(G_t - v(S_t)) \mathbf{x}_t \\ &= \sum_{t=0}^{T-1} \alpha \left( \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k \right) \mathbf{x}_t && \text{(Using result from previous slide)} \\ &= \sum_{k=0}^{T-1} \alpha \sum_{t=0}^k \gamma^{k-t} \delta_k \mathbf{x}_t && \text{(Using } \sum_{i=0}^m \sum_{j=i}^m z_{ij} = \sum_{j=0}^m \sum_{i=0}^j z_{ij} \text{)} \\ &= \sum_{k=0}^{T-1} \alpha \delta_k \underbrace{\sum_{t=0}^k \gamma^{k-t}}_{\equiv \mathbf{e}_k} \mathbf{x}_t && \\ &= \sum_{k=0}^{T-1} \alpha \delta_k \mathbf{e}_k &= \underbrace{\sum_{t=0}^{T-1} \alpha \delta_t \mathbf{e}_t}_{\text{renaming } k \rightarrow t} .\end{aligned}$$



## Eligibility traces

Accumulating a whole episode of updates:

$$\begin{aligned}\Delta \mathbf{w}_k &= \sum_{t=0}^{T-1} \alpha \delta_t \mathbf{e}_t && \text{where} & \mathbf{e}_t &= \sum_{j=0}^t \gamma^{t-j} \mathbf{x}_j \\ &&&& &= \sum_{j=0}^{t-1} \gamma^{t-j} \mathbf{x}_j + \mathbf{x}_t \\ &&&& &= \underbrace{\gamma \sum_{j=0}^{t-1} \gamma^{t-1-j} \mathbf{x}_j}_{= \mathbf{e}_{t-1}} + \mathbf{x}_t \\ &&&& &= \gamma \mathbf{e}_{t-1} + \mathbf{x}_t .\end{aligned}$$

The vector  $\mathbf{e}_t$  is called an **eligibility trace**

Every step, it decays (according to  $\gamma$ ) and then the current feature  $\mathbf{x}_t$  is added



## Eligibility traces

- ▶ Accumulating a whole episode of updates:

$$\Delta \mathbf{w}_t \equiv \alpha \delta_t \mathbf{e}_t \quad (\text{one time step})$$

$$\Delta \mathbf{w}_k = \sum_{t=0}^{T-1} \Delta \mathbf{w}_t \quad (\text{whole episode})$$

$$\text{where } \mathbf{e}_t = \gamma \mathbf{e}_{t-1} + \mathbf{x}_t .$$

(And then apply  $\Delta \mathbf{w}$  at the end of the episode)

- ▶ Intuition: the same TD error shows up in multiple MC errors—grouping them allows applying it to all past states in one update



# Eligibility Traces: Intuition



## Eligibility traces

Consider a batch update on an episode with four steps:  $t \in \{0, 1, 2, 3\}$

$$\begin{array}{l} \Delta \mathbf{v} = \\ \begin{array}{ccccc} \delta_0 \mathbf{e}_0 & \delta_1 \mathbf{e}_1 & \delta_2 \mathbf{e}_2 & \delta_3 \mathbf{e}_3 \\ (G_0 - v(S_0)) \mathbf{x}_0 & \delta_0 \mathbf{x}_0 & \gamma \delta_1 \mathbf{x}_0 & \gamma^2 \delta_2 \mathbf{x}_0 & \gamma^3 \delta_3 \mathbf{x}_0 \\ (G_1 - v(S_1)) \mathbf{x}_1 & & \delta_1 \mathbf{x}_1 & \gamma \delta_2 \mathbf{x}_1 & \gamma^2 \delta_3 \mathbf{x}_1 \\ (G_2 - v(S_2)) \mathbf{x}_2 & & & \delta_2 \mathbf{x}_2 & \gamma \delta_3 \mathbf{x}_2 \\ (G_3 - v(S_3)) \mathbf{x}_3 & & & & \delta_3 \mathbf{x}_3 \end{array} \end{array}$$



# Mixed Multi-Step Returns and Eligibility Traces



## Mixing multi-step returns & traces

- ▶ Reminder: mixed multi-step return

$$G_t^\lambda = R_{t+1} + \gamma \left( (1 - \lambda)v(S_{t+1}) + \lambda G_{t+1}^\lambda \right)$$

- ▶ The associated error and trace update are

$$G_t^\lambda = \sum_{k=0}^{T-t} \lambda^k \gamma^k \delta_{t+k} \quad (\text{same as before, but with } \lambda\gamma \text{ instead of } \gamma)$$

$$\implies \mathbf{e}_t = \gamma \lambda \mathbf{e}_{t-1} + \mathbf{x}_t \quad \text{and} \quad \Delta \mathbf{w}_t = \alpha \delta_t \mathbf{e}_t .$$

- ▶ This is called an **accumulating trace** with decay  $\gamma\lambda$
- ▶ It is exact for batched episodic updates ('offline'), similar traces exist for online updating



# End of Lecture

Next lecture:  
Model-free control



# Lecture 6: Model-Free Control

Hado van Hasselt

UCL, 2021

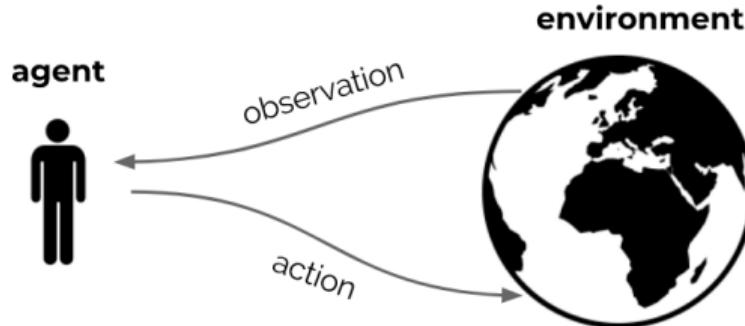


# Background

Sutton & Barto 2018, Chapter 6



# Recap



- ▶ Reinforcement learning is the science of learning to make decisions
- ▶ Agents can learn a **policy**, **value function** and/or a **model**
- ▶ The general problem involves taking into account **time** and **consequences**
- ▶ Decisions affect the **reward**, the **agent state**, and **environment state**



# Model-Free Control

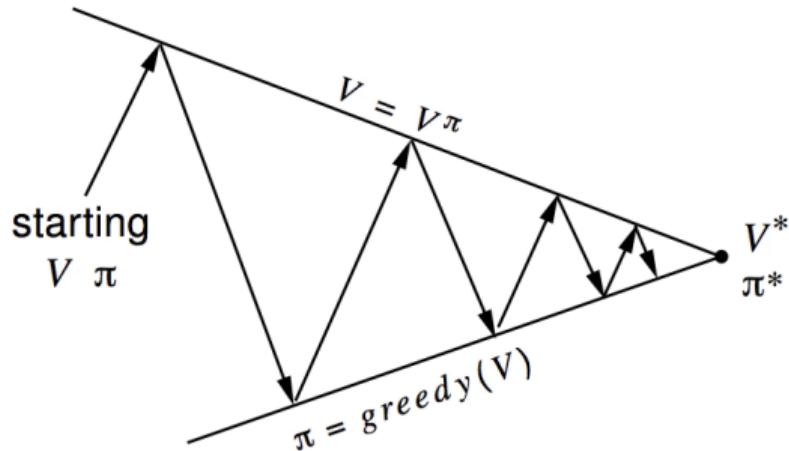
- ▶ Previous lecture: **Model-free prediction**  
**Estimate** the value function of an unknown MDP
- ▶ This lecture: **Model-free control**  
**Optimise** the value function of an unknown MDP



# Monte-Carlo Control



# Generalized Policy Iteration (Refresher)

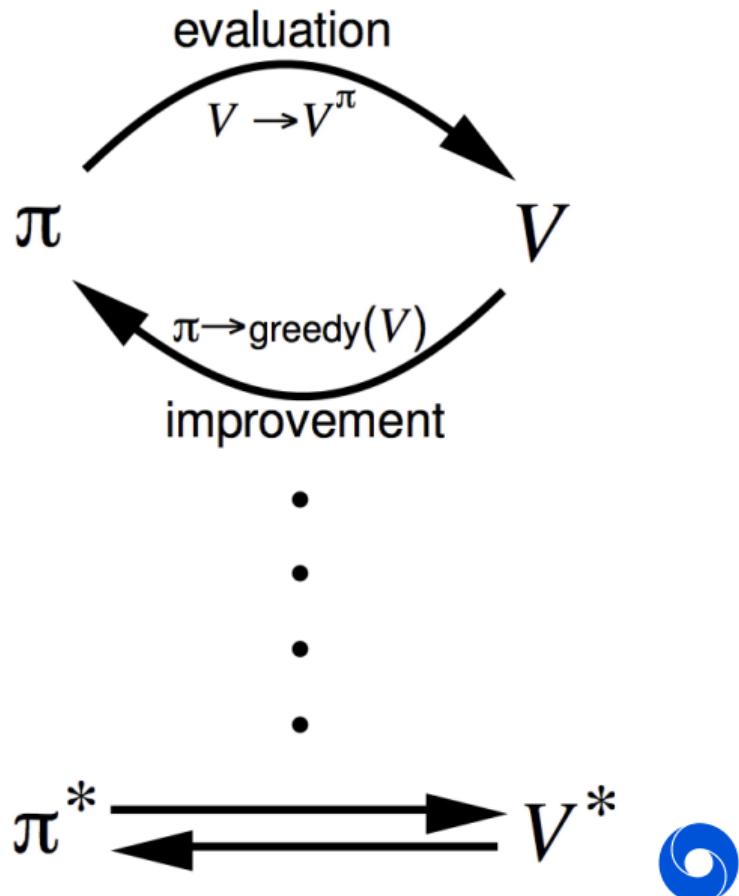


## ► Policy evaluation

Estimate  $v_\pi(s)$  for all  $s$

## ► Policy improvement

Generate  $\pi'$  such that  $v_{\pi'}(s) \geq v_\pi(s)$  for all  $s$



## Recap: Model-Free Policy Evaluation

$$v_{n+1}(S_t) = v_n(S_t) + \alpha (G_t - v_n(S_t))$$

► Variants:

$$G_t^{\text{MC}} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

$$= R_{t+1} + \gamma G_{t+1}^{\text{MC}}$$

MC

$$G_t^{(1)} = R_{t+1} + \gamma v_t(S_{t+1})$$

TD(0)

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n v_t(S_{t+n})$$

$$= R_{t+1} + \gamma G_{t+1}^{(n-1)}$$

n-step TD

$$G_t^\lambda = R_{t+1} + \gamma [(1-\lambda)v_t(S_{t+1}) + \lambda G_{t+1}^\lambda]$$

TD( $\lambda$ )

In all cases, for given  $\pi$  goal is estimating  $v_\pi$ , data is generated to  $\pi$



## Model-Free Policy Iteration Using Action-Value Function

- ▶ Greedy policy improvement over  $v(s)$  requires model of MDP

$$\pi'(s) = \operatorname{argmax}_a \mathbb{E} [R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s, A_t = a]$$

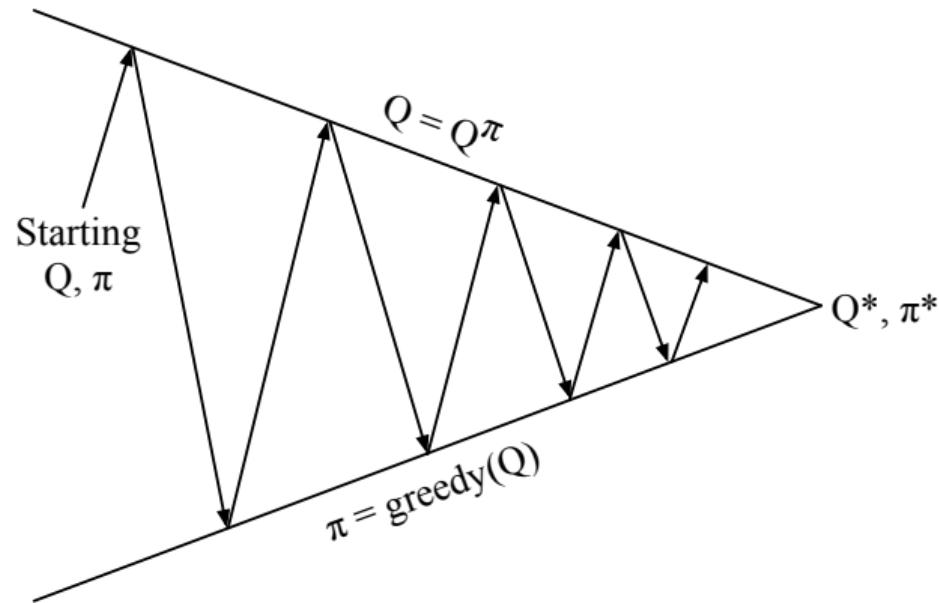
- ▶ Greedy policy improvement over  $q(s, a)$  is **model-free**

$$\pi'(s) = \operatorname{argmax}_a q(s, a)$$

- ▶ This makes action values convenient



# Generalised Policy Iteration with Action-Value Function

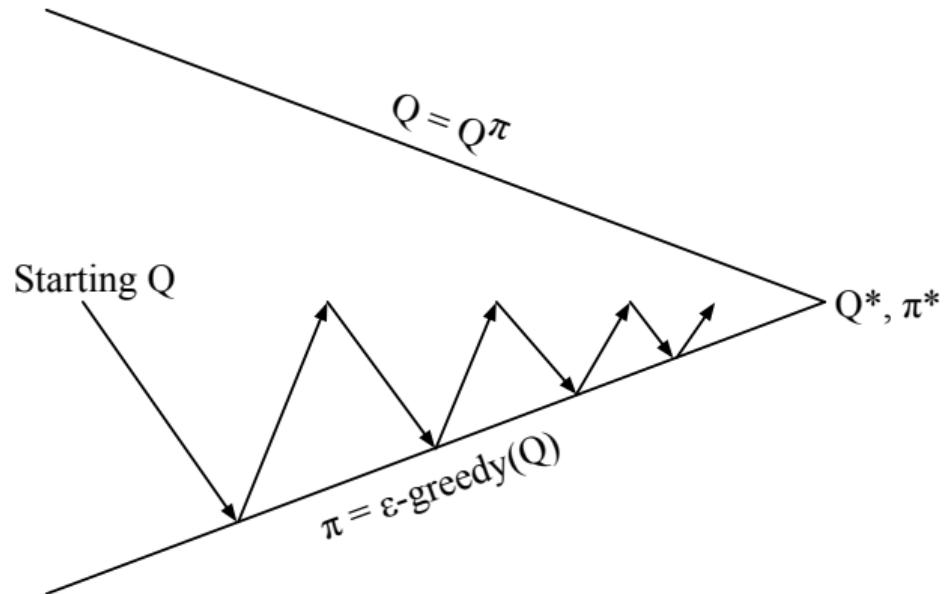


Policy evaluation Monte-Carlo policy evaluation,  $q \approx q_\pi$

Policy improvement Greedy policy improvement? No exploration!  
(Can't sample all  $s, a$ , when learning by interacting)



# Monte-Carlo Generalized Policy Iteration



**Every episode:**

Policy evaluation Monte-Carlo policy evaluation,  $\mathbf{q} \approx \mathbf{q}_\pi$

Policy improvement  $\epsilon$ -greedy policy improvement



# Model-free control

Repeat:

- ▶ Sample episode  $1, \dots, k, \dots$ , using  $\pi$ :  $\{S_1, A_1, R_2, \dots, S_T\} \sim \pi$
- ▶ For each state  $S_t$  and action  $A_t$  in the episode,

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha_t (G_t - q(S_t, A_t))$$

- ▶ E.g.,

$$\alpha_t = \frac{1}{N(S_t, A_t)} \quad \text{of} \quad \alpha_t = 1/k$$

- ▶ Improve policy based on new action-value function

$$\epsilon \leftarrow 1/k$$

$$\pi \leftarrow \epsilon\text{-greedy}(q)$$

(Generalises the  $\epsilon$ -greedy bandit algorithm)



# GLIE

## Definition

### Greedy in the Limit with Infinite Exploration (GLIE)

- ▶ All state-action pairs are explored infinitely many times,

$$\forall s, a \quad \lim_{t \rightarrow \infty} N_t(s, a) = \infty$$

- ▶ The policy converges to a greedy policy,

$$\lim_{t \rightarrow \infty} \pi_t(a|s) = \mathcal{I}(a = \operatorname{argmax}_{a'} q_t(s, a'))$$

- ▶ For example,  $\epsilon$ -greedy with  $\epsilon_k = \frac{1}{k}$



# GLIE

## Theorem

*GLIE Model-free control converges to the optimal action-value function,  $q_t \rightarrow q_*$*



# Temporal-Difference Learning For Control

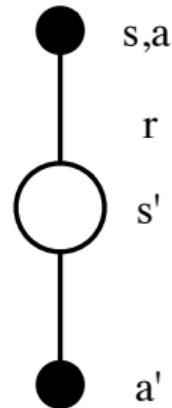


## MC vs. TD Control

- ▶ Temporal-difference (TD) learning has several advantages over Monte-Carlo (MC)
  - ▶ Lower variance
  - ▶ Online
  - ▶ Can learn from incomplete sequences
- ▶ Natural idea: use TD instead of MC for control
  - ▶ Apply TD to  $q(s, a)$
  - ▶ Use, e.g.,  $\epsilon$ -greedy policy improvement
  - ▶ Update every time-step



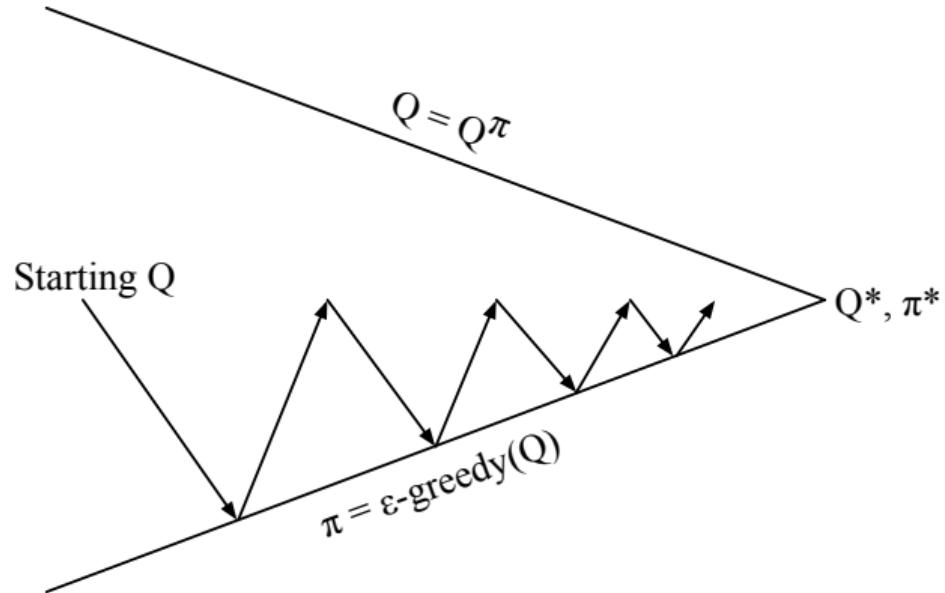
# Updating Action-Value Functions with SARSA



$$q_{t+1}(S_t, A_t) = q_t(S_t, A_t) + \alpha_t (R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t))$$



# SARSA



Every **time-step**:

Policy evaluation **SARSA**,  $q \approx q_\pi$

Policy improvement  $\epsilon$ -greedy policy improvement



## Tabular SARSA

Initialize  $Q(s, a)$  arbitrarily

Repeat (for each episode):

    Initialize  $s$

    Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $a$ , observe  $r, s'$

        Choose  $a'$  from  $s'$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$$

$s \leftarrow s'; a \leftarrow a'$ ;

    until  $s$  is terminal



# Updating Action-Value Functions with SARSA

$$q_{t+1}(S_t, A_t) = q_t(S_t, A_t) + \alpha_t (R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t))$$

## Theorem

*Tabular SARSA converges to the optimal action-value function,  $q(s, a) \rightarrow q_*(s, a)$ , if the policy is GLIE*



# Off-policy TD and Q-learning



# Dynamic programming

- We discussed several dynamic programming algorithms

$$v_{k+1}(s) = \mathbb{E} [R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t \sim \pi(S_t)] \quad (\text{policy evaluation})$$

$$v_{k+1}(s) = \max_a \mathbb{E} [R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \quad (\text{value iteration})$$

$$q_{k+1}(s, a) = \mathbb{E} [R_{t+1} + \gamma q_k(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \quad (\text{policy evaluation})$$

$$q_{k+1}(s, a) = \mathbb{E} \left[ R_{t+1} + \gamma \max_{a'} q_k(S_{t+1}, a') \mid S_t = s, A_t = a \right] \quad (\text{value iteration})$$



# TD learning

- ▶ Analogous model-free TD algorithms

$$v_{t+1}(S_t) = v_t(S_t) + \alpha_t (R_{t+1} + \gamma v_t(S_{t+1}) - v_t(S_t)) \quad (\text{TD})$$

$$q_{t+1}(s, a) = q_t(S_t, A_t) + \alpha_t (R_{t+1} + \gamma q_t(S_{t+1}, A_{t+1}) - q_t(S_t, A_t)) \quad (\text{SARSA})$$

$$q_{t+1}(s, a) = q_t(S_t, A_t) + \alpha_t \left( R_{t+1} + \gamma \max_{a'} q_t(S_{t+1}, a') - q_t(S_t, A_t) \right) \quad (\text{Q-learning})$$

- ▶ Note, no trivial analogous version of value iteration

$$v_{k+1}(s) = \max_a \mathbb{E} [R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a]$$

Can you explain why?



# On and Off-Policy Learning

- ▶ **On-policy** learning
  - ▶ Learn about **behaviour** policy  $\pi$  from experience sampled from  $\pi$
- ▶ **Off-policy** learning
  - ▶ Learn about **target** policy  $\pi$  from experience sampled from  $\mu$
  - ▶ Learn ‘counterfactually’ about other things you could do: “what if...?”
    - ▶ E.g., “What if I would turn left?”  $\implies$  new observations, rewards?
    - ▶ E.g., “What if I would play more defensively?”  $\implies$  different win probability?
    - ▶ E.g., “What if I would continue to go forward?”  $\implies$  how long until I bump into a wall?



# Off-Policy Learning

- ▶ Evaluate target policy  $\pi(a|s)$  to compute  $v_\pi(s)$  or  $q_\pi(s, a)$
- ▶ While using behaviour policy  $\mu(a|s)$  to generate actions
- ▶ Why is this important?
  - ▶ Learn from observing humans or other agents (e.g., from logged data)
  - ▶ Re-use experience from old policies (e.g., from your own past experience)
  - ▶ Learn about **multiple** policies while following **one** policy
  - ▶ Learn about **greedy** policy while following **exploratory** policy
- ▶ **Q-learning** estimates the value of the **greedy** policy

$$q_{t+1}(s, a) = q_t(S_t, A_t) + \alpha_t \left( R_{t+1} + \gamma \max_{a'} q_t(S_{t+1}, a') - q_t(S_t, A_t) \right)$$

**Acting** greedy all the time would not explore sufficiently



# Q-Learning Control Algorithm

## Theorem

*Q-learning control converges to the optimal action-value function,  $q \rightarrow q^*$ , as long as we take each action in each state infinitely often.*

Note: no need for greedy behaviour!

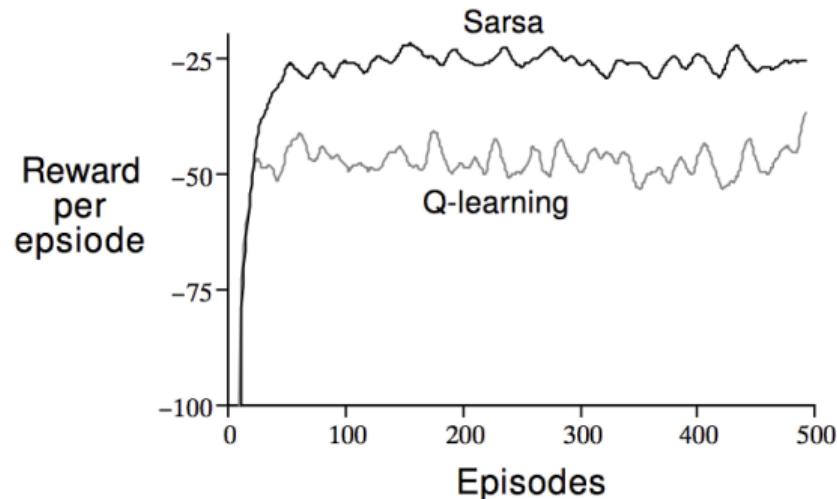
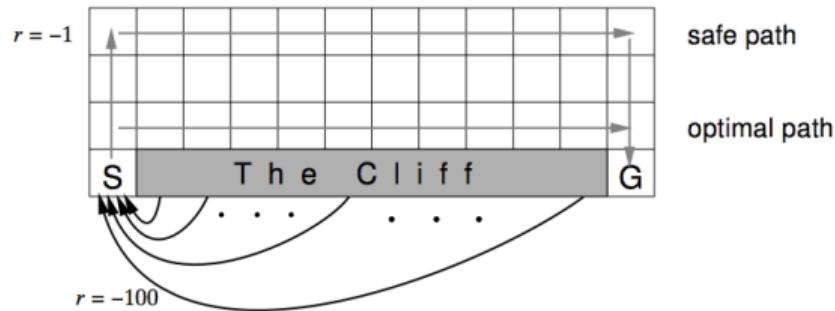
Works for **any** policy that eventually selects all actions sufficiently often  
(Requires appropriately decaying step sizes  $\sum_t \alpha_t = \infty$ ,  $\sum_t \alpha_t^2 < \infty$ ,  
E.g.,  $\alpha = 1/t^\omega$ , with  $\omega \in (0.5, 1)$ )



# Example



# Cliff Walking Example



# Overestimation in Q-learning



# Q-learning overestimation

- ▶ Classical Q-learning has potential issues
- ▶ Recall

$$\max_a q_t(S_{t+1}, a) = q_t(S_{t+1}, \operatorname{argmax}_a q_t(S_{t+1}, a))$$

- ▶ Uses same values to **select** and to **evaluate**
- ▶ ... but values are approximate
  - ▶ **more** likely to select **overestimated values**
  - ▶ **less** likely to select **underestimated values**
- ▶ This causes upward bias



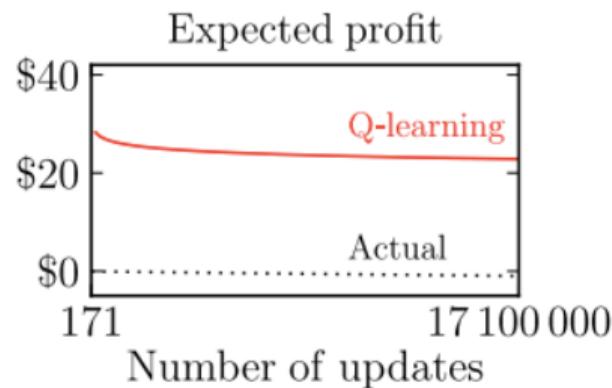
## Q-learning overestimation: roulette example

- ▶ Roulette: gambling game
- ▶ Here, 171 actions: bet \$1 on one of 170 options, or ‘stop’
- ▶ ‘Stop’ ends the episode, with \$0
- ▶ All other actions have high variance reward, with negative expected value
- ▶ Betting actions do not end the episode, instead can bet again



## Q-learning overestimation: roulette example

- ▶ Roulette: gambling game
- ▶ Here, 171 actions: bet \$1 on one of 170 options, or ‘stop’
- ▶ ‘Stop’ ends the episode, with \$0
- ▶ All other actions have high variance reward, with negative expected value
- ▶ Betting actions do not end the episode, instead can bet again



## Q-learning overestimation

- ▶ Q-learning overestimates because it uses the same values to **select** and to **evaluate**

$$\max_a q_t(S_{t+1}, a) = q_t(S_{t+1}, \operatorname{argmax}_a q_t(S_{t+1}, a))$$

- ▶ Roulette: quite likely that some actions have won, on average
- ▶ Q-learning will update if the state actually has high value
- ▶ Solution: decouple selection from evaluation



# Double Q-learning

## ► Double Q-learning:

- ▶ Store two action-value functions:  $q$  and  $q'$

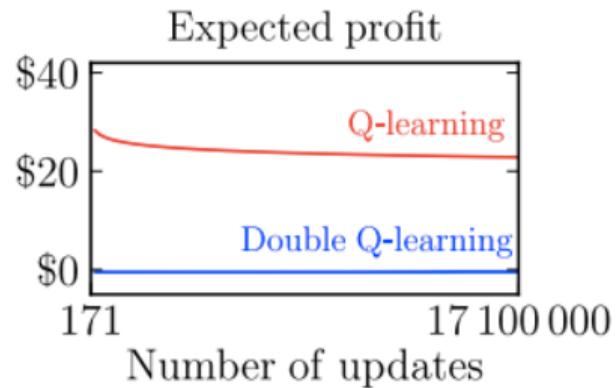
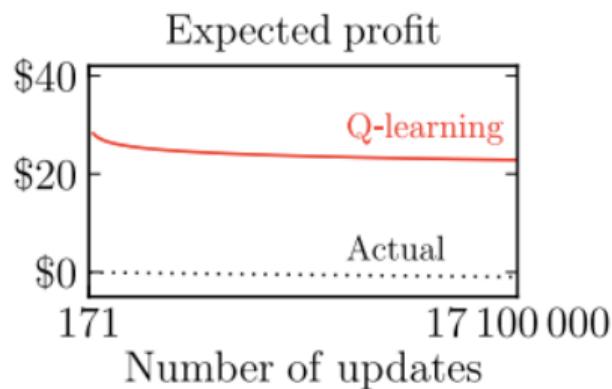
$$R_{t+1} + \gamma \underset{a}{\operatorname{argmax}} q'_t(S_{t+1}, a) \quad (1)$$

$$R_{t+1} + \gamma q_t(S_{t+1}, \underset{a}{\operatorname{argmax}} q'_t(S_{t+1}, a)) \quad (2)$$

- ▶ Each  $t$ , pick  $q$  or  $q'$  (e.g., randomly) and update using (1) for  $q$  or (2) for  $q'$
- ▶ Can use both to act (e.g., use policy based on  $(q + q')/2$ )
- ▶ Double Q-learning also converges to the optimal policy under the same conditions as Q-learning

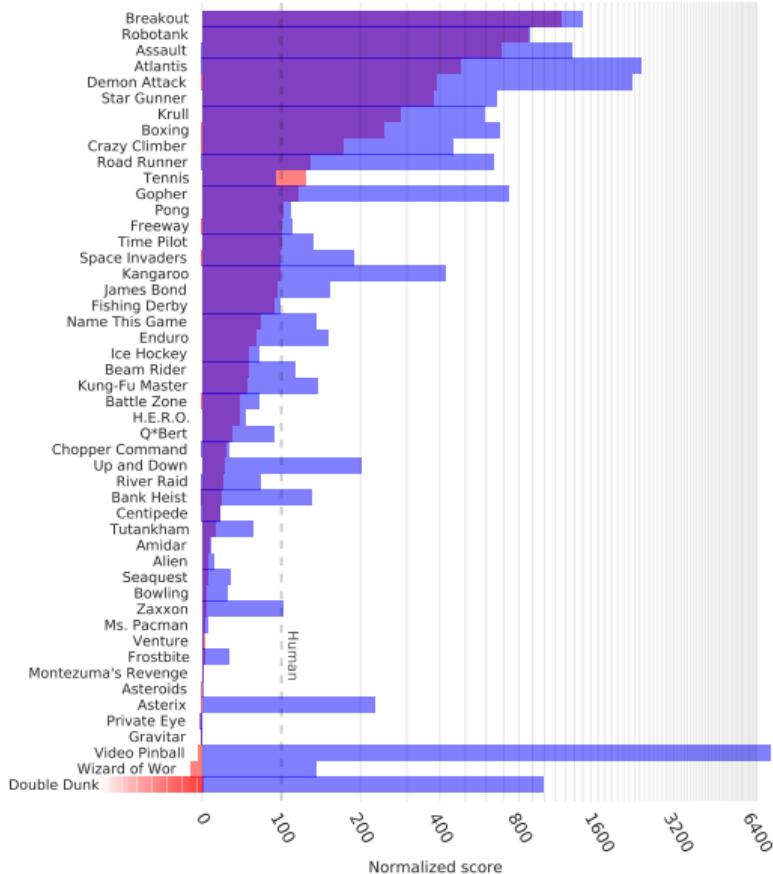


## Roulette example



# Double DQN on Atari

DQN  
Double DQN  
(This used a ‘target network’,  
to be explained later)



## Double learning

- ▶ The idea of double Q-learning can be generalised to other updates
  - ▶ E.g., if you are (soft-) greedy (e.g.,  $\epsilon$ -greedy), then SARSA can also overestimate
  - ▶ The same solution can be used
  - ▶  $\implies$  double SARSA



# Example



# Off-Policy Learning: Importance Sampling Corrections



## Off-policy learning

- ▶ Recall: off-policy learning means learning about one policy  $\pi$  from experience generated according to a different policy  $\mu$
- ▶ Q-learning is one example, but there are other options
- ▶ Fortunately, there are general tools to help with this
- ▶ Caveat: you can't expect to learn about things you **never** do



## Importance sampling corrections

- ▶ Goal: given some function  $f$  with random inputs  $X$ , and a distribution  $d'$ , estimate the expectation of  $f(X)$  under a different (target) distribution  $d$
- ▶ Solution: weight the data by the ratio  $d/d'$

$$\begin{aligned}\mathbb{E}_{x \sim d}[f(x)] &= \sum d(x)f(x) \\ &= \sum d'(x)\frac{d(x)}{d'(x)}f(x) \\ &= \mathbb{E}_{x \sim d'}\left[\frac{d(x)}{d'(x)}f(x)\right]\end{aligned}$$

- ▶ Intuition:
  - ▶ scale up events that are rare under  $d'$ , but common under  $d$
  - ▶ scale down events that are common under  $d'$ , but rare under  $d$



# Importance sampling corrections

- ▶ Example: estimate one-step reward
- ▶ Behaviour is  $\mu(a|s)$

$$\begin{aligned}\mathbb{E}[R_{t+1} \mid S_t = s, A_t \sim \pi] &= \sum_a \pi(a|s)r(s, a) \\ &= \sum_a \mu(a|s) \frac{\pi(a|s)}{\mu(a|s)} r(s, a) \\ &= \mathbb{E} \left[ \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} R_{t+1} \mid S_t = s, A_t \sim \mu \right]\end{aligned}$$

- ▶ Ergo, when following policy  $\mu$ , can use  $\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} R_{t+1}$  as unbiased sample



# Importance Sampling for Off-Policy Monte-Carlo

- ▶ Goal: estimate  $v_\pi$
- ▶ Data: trajectory  $\tau_t = \{S_t, A_t, R_{t+1}, S_{t+1}, \dots\}$  generated with  $\mu$
- ▶ Solution: use return  $G(\tau_t) = G_t = R_{t+1} + \gamma R_{t+2} + \dots$ , and correct:

$$\begin{aligned}\frac{p(\tau_t|\pi)}{p(\tau_t|\mu)} G(\tau_t) &= \frac{p(A_t|S_t, \pi)p(R_{t+1}, S_{t+1}|S_t, A_t)p(A_{t+1}|S_{t+1}, \pi)\dots}{p(A_t|S_t, \mu)p(R_{t+1}, S_{t+1}|S_t, A_t)p(A_{t+1}|S_{t+1}, \mu)\dots} G_t \\ &= \frac{p(A_t|S_t, \pi)p(\cancel{R_{t+1}, S_{t+1}}|S_t, A_t)\cancel{p(A_{t+1}|S_{t+1}, \pi)\dots}}{p(A_t|S_t, \mu)\cancel{p(R_{t+1}, S_{t+1}|S_t, A_t)}\cancel{p(A_{t+1}|S_{t+1}, \mu)\dots}} G_t \\ &= \frac{p(A_t|S_t, \pi)p(A_{t+1}|S_{t+1}, \pi)\dots}{p(A_t|S_t, \mu)p(A_{t+1}|S_{t+1}, \mu)\dots} G_t \\ &= \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \dots G_t\end{aligned}$$



# Importance Sampling for Off-Policy TD Updates

- ▶ Use TD targets generated from  $\mu$  to evaluate  $\pi$
- ▶ Weight TD target  $r + \gamma v(s')$  by importance sampling
- ▶ Only need a single importance sampling correction

$$v(S_t) \leftarrow v(S_t) + \alpha \left( \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma v(S_{t+1})) - v(S_t) \right)$$

- ▶ Much lower variance than Monte-Carlo importance sampling
- ▶ Policies only need to be similar over a single step



# Importance Sampling for Off-Policy TD Updates

► Proof:

$$\begin{aligned}& \mathbb{E}_{\mu} \left[ \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma v(S_{t+1})) - v(S_t) \mid S_t = s \right] \\&= \sum_a \mu(a|s) \left( \frac{\pi(a|s)}{\mu(a|s)} \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s, A_t = a] - v(s) \right) \\&= \sum_a \pi(a|s) \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s, A_t = a] - \sum_a \mu(a|s)v(s) \\&= \sum_a \pi(a|s) \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s, A_t = a] - \sum_a \pi(a|s)v(s) \\&= \sum_a \pi(a|s) \left( \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s, A_t = a] - v(s) \right) \\&= \mathbb{E}_{\pi} \left[ R_{t+1} + \gamma v(S_{t+1}) - v(s) \mid S_t = s \right]\end{aligned}$$



## Expected SARSA

- ▶ We now consider off-policy learning of action-values  $q(s, a)$
- ▶ No importance sampling is required
- ▶ Next action may be chosen using behaviour policy  $A_{t+1} \sim \mu(\cdot | S_{t+1})$
- ▶ But we consider probabilities under  $\pi(\cdot | S_t)$
- ▶ Update  $q(S_t, A_t)$  towards value of alternative action

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha \left( R_{t+1} + \gamma \sum_a \pi(a | S_{t+1}) q(S_{t+1}, a) - q(S_t, A_t) \right)$$

- ▶ Called **Expected SARSA** (sometimes called ‘General Q-learning’)
- ▶ Q-learning is a special case with greedy target policy  $\pi$



# Summary



## Model-Free Policy Iteration

- ▶ We can learn action values to predict the current policy  $\pi$
- ▶ Then we can do policy improvement, e.g., make the policy greedy  $\pi \rightarrow \pi'$
- ▶ Q-learning is akin to value iteration: immediately estimate the **current greedy policy**
- ▶ (Expected) SARSA can be used more similar to policy iteration:  
evaluate current behaviour, then (immediately) update behaviour
- ▶ Sometimes we want to estimate some different policy: this is off-policy learning
- ▶ Learning about the greedy policy is a special case of off-policy learning



# Off-Policy Control with Q-Learning

- We want behaviour and target policies to **improve**
- E.g., the target policy  $\pi$  is **greedy** w.r.t.  $q(s, a)$

$$\pi(S_{t+1}) = \operatorname{argmax}_{a'} q(S_{t+1}, a')$$

- The behaviour policy  $\mu$  can explore: e.g.  **$\epsilon$ -greedy** w.r.t.  $q(s, a)$
- The Q-learning target is:

$$\begin{aligned} R_{t+1} + \gamma \sum_a \pi^{\text{greedy}}(a|S_{t+1})q(S_{t+1}, a) \\ = R_{t+1} + \gamma \max_a q(S_{t+1}, a) \end{aligned}$$



## On-Policy Control with SARSA

- ▶ In SARSA, the target and behaviour policies are the same

$$\text{target} = R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$$

- ▶ Then, for convergence to  $q^*$ , we need the addition requirement that  $\pi$  becomes greedy
- ▶ For instance,  $\epsilon$ -greedy or softmax with decreasing exploration



# Summary

- ▶ Q-learning uses a **greedy** target policy
- ▶ SARSA uses a **stochastic sample from the behaviour** as target policy
- ▶ Expected SARSA uses **any** target policy
- ▶ Double learning uses a **separate value function** to evaluate the policy (for any policy)
- ▶ Double learning is not necessary if there is no correlation between target policy and value function (e.g., pure prediction)
- ▶ When using a greedy policy (Q-learning), there are strong correlations. Then double learning (Double Q-learning) can be useful



Please use Moodle to ask questions

*The only stupid question is the one you were afraid to ask but never did.*

*-Rich Sutton*



# Lecture 7: Function approximation in reinforcement learning (And deep reinforcement learning)

Hado van Hasselt

UCL, 2021

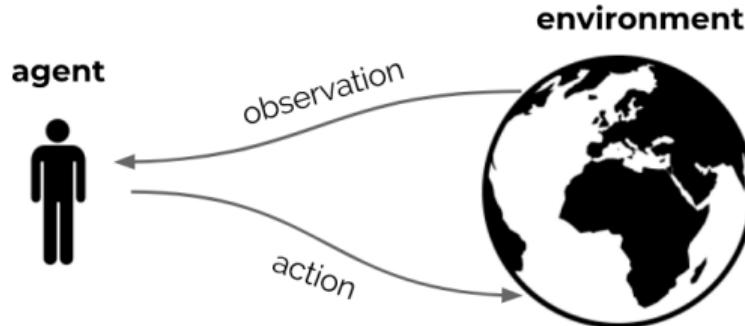


# Background

Sutton & Barto 2018, Chapters 9 + 10 (+ 11)



# Recap



- ▶ Reinforcement learning is the science of learning to make decisions
- ▶ Agents can learn a **policy**, **value function** and/or a **model**
- ▶ The general problem involves taking into account **time** and **consequences**
- ▶ Decisions affect the **reward**, the **agent state**, and **environment state**



# Why function approximation?



# Function approximation and deep reinforcement learning

- ▶ The **policy**, **value function**, **model**, and **agent state update** are all functions
- ▶ We want to learn these from experience
- ▶ If there are too many states, we need to approximate
- ▶ This is often called **deep reinforcement learning**,  
when using **neural networks** to represent these functions
- ▶ The term is fairly new ( $\pm 7\text{-}8$  years) — the combination is fairly old ( $\pm 50$  years)



# Function approximation and deep reinforcement learning

This lecture

- ▶ We consider learning **predictions** (value functions; including value-based control)

Upcoming lectures

- ▶ Off-policy learning
- ▶ Approximate dynamic programming (theory with function approximation)
- ▶ Learn explicit policies (policy gradients)
- ▶ Model-based RL



# Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve **large** problems, e.g.

- ▶ Backgammon:  $10^{20}$  states
- ▶ Go:  $10^{170}$  states
- ▶ Helicopter: continuous state space
- ▶ Robots: real world

How can we apply our methods for **prediction** and **control**?



# Value function approximation



# Value Function Approximation

- ▶ So far we mostly considered **lookup tables**
  - ▶ Every state  $s$  has an entry  $v(s)$
  - ▶ Or every state-action pair  $s, a$  has an entry  $q(s, a)$
- ▶ Problem with large MDPs:
  - ▶ There are too many states and/or actions to store in memory
  - ▶ It is too slow to learn the value of each state individually
  - ▶ Individual environment states are often **not fully observable**



# Value Function Approximation

Solution for large MDPs:

- ▶ Estimate value function with **function approximation**

$$v_{\mathbf{w}}(s) \approx v_{\pi}(s) \quad (\text{or } v_*(s))$$

$$q_{\mathbf{w}}(s, a) \approx q_{\pi}(s, a) \quad (\text{or } q_*(s, a))$$

- ▶ Update parameter  $\mathbf{w}$  (e.g., using MC or TD learning)
- ▶ Generalise to unseen states



## Agent state update

When the environment state is not fully observable ( $S_t^{\text{env}} \neq O_t$ )

- ▶ Use the **agent state**:

$$\mathbf{s}_t = u_{\omega}(\mathbf{s}_{t-1}, A_{t-1}, O_t)$$

with parameters  $\omega$  (typically  $\omega \in \mathbb{R}^n$ )

- ▶ Henceforth,  $S_t$  or  $\mathbf{s}_t$  denotes the agent state
- ▶ Think of this as either a vector inside the agent,  
or, in the simplest case, just the current observation:  $S_t = O_t$



# Function classes



# Classes of Function Approximation

- ▶ **Tabular**: a table with an entry for each MDP state
- ▶ **State aggregation**: Partition environment states (or observations) into a discrete set
- ▶ **Linear function approximation**
  - ▶ Consider fixed agent state update (e.g.,  $S_t = O_t$ )
  - ▶ Fixed feature map  $\mathbf{x} : \mathcal{S} \rightarrow \mathbb{R}^n$
  - ▶ Values are linear function of features:  $v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s)$
  - ▶ Note: state aggregation and tabular are special cases of linear FA
- ▶ **Differentiable function approximation**
  - ▶  $v_{\mathbf{w}}(s)$  is a differentiable function of  $\mathbf{w}$ , could be **non-linear**
  - ▶ E.g., a convolutional neural network that takes pixels as input
  - ▶ Another interpretation: features are not fixed, but learnt



# Classes of Function Approximation

In principle, **any** function approximator can be used, but RL has specific properties:

- ▶ Experience is not i.i.d. — successive time-steps are correlated
- ▶ Agent's policy affects the data it receives
- ▶ Regression targets can be **non-stationary**
  - ▶ ...because of changing policies (which can change the target and the data!)
  - ▶ ...because of bootstrapping
  - ▶ ...because of non-stationary dynamics (e.g., other learning agents)
  - ▶ ...because the world is large (never quite in the same state)



# Classes of Function Approximation

Which function approximation should you choose?

This depends on your goals.

- ▶ **Tabular**: good theory but does not scale/generalise
- ▶ **Linear**: reasonably good theory, but requires good features
- ▶ **Non-linear**: less well-understood, but scales well
  - Flexible, and less reliant on picking good features first (e.g., by hand)
- ▶ (Deep) neural nets often perform quite well, and remain a popular choice



# Gradient-based algorithms



# Gradient Descent

- ▶ Let  $J(\mathbf{w})$  be a differentiable function of parameter vector  $\mathbf{w}$
- ▶ Define the **gradient** of  $J(\mathbf{w})$  to be

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial w_n} \end{pmatrix}$$

- ▶ Goal: to minimise of  $J(\mathbf{w})$
- ▶ Method: move  $\mathbf{w}$  in the direction of negative gradient

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where  $\alpha$  is a step-size parameter



## Approximate Values By Stochastic Gradient Descent

- ▶ Goal: find  $\mathbf{w}$  that minimise the difference between  $v_{\mathbf{w}}(s)$  and  $v_{\pi}(s)$

$$J(\mathbf{w}) = \mathbb{E}_{S \sim d}[(v_{\pi}(S) - v_{\mathbf{w}}(S))^2]$$

where  $d$  is a distribution over states (typically induced by the policy and dynamics)

- ▶ Gradient descent:

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w}) = \alpha \mathbb{E}_d(v_{\pi}(S) - v_{\mathbf{w}}(S)) \nabla_{\mathbf{w}} v_{\mathbf{w}}(S)$$

- ▶ **Stochastic gradient descent** (SGD), sample the gradient:

$$\Delta \mathbf{w} = \alpha(G_t - v_{\mathbf{w}}(S_t)) \nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

Note: Monte Carlo return  $G_t$  is a sample for  $v_{\pi}(S_t)$

- ▶ We often write  $\nabla v(S_t)$  as short hand for

$$\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)|_{\mathbf{w}=\mathbf{w}_t}$$



# Linear function approximation



## Feature Vectors

- ▶ Represent state by a **feature vector**

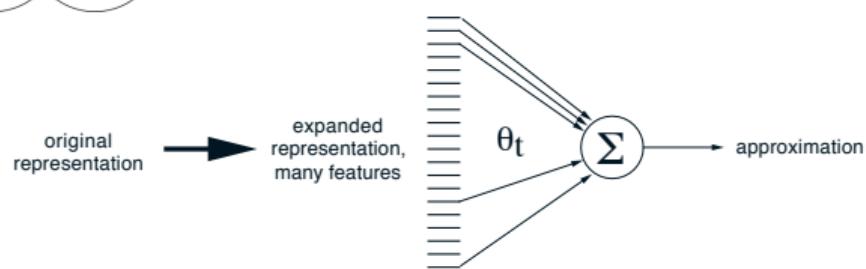
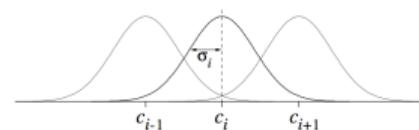
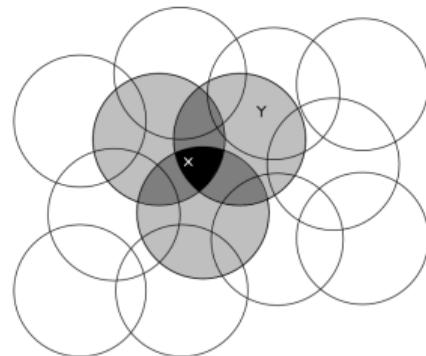
$$\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

- ▶  $\mathbf{x} : \mathcal{S} \rightarrow \mathbb{R}^n$  is a fixed mapping from state (e.g., observation) to features
- ▶ Short-hand:  $\mathbf{x}_t = \mathbf{x}(S_t)$
- ▶ For example:
  - ▶ Distance of robot from landmarks
  - ▶ Trends in the stock market
  - ▶ Piece and pawn configurations in chess

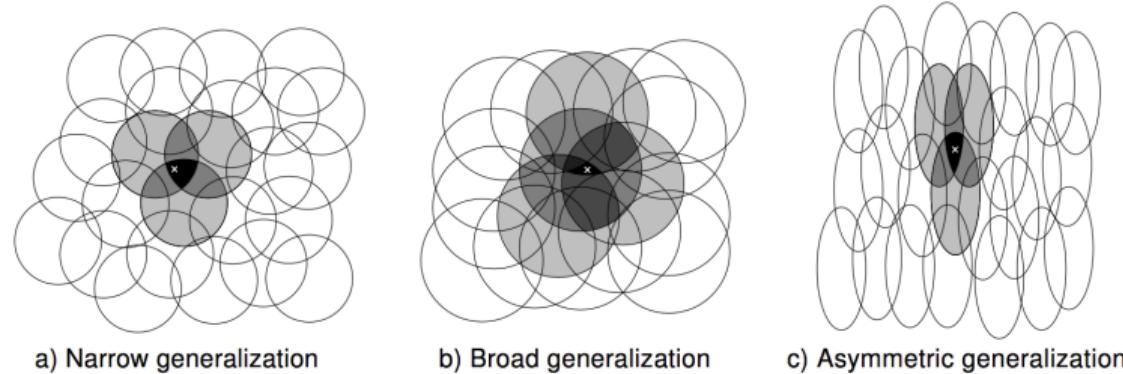


# Feature construction example: coarse coding

- ▶ **Coarse coding** provides large feature vector  $\mathbf{x}(s)$
- ▶ Parameter vector  $\mathbf{w}$  gives a value to each feature



# Generalization in Coarse Coding



- ▶ We aggregate multiple states
- ▶ This means the resulting feature vector/agent state is **non-Markovian**
- ▶ This is the common case when using function approximation
- ▶ Consider whether good solutions exist for given features + function approximation
- ▶ Neural networks tend to be more flexible



# Linear model-free prediction



# Linear Value Function Approximation

- ▶ Approximate value function by a linear combination of features

$$v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{j=1}^n \mathbf{x}_j(s) w_j$$

- ▶ Objective function ('loss') is quadratic in  $\mathbf{w}$

$$J(\mathbf{w}) = \mathbb{E}_{S \sim d}[(v_{\pi}(S) - \mathbf{w}^\top \mathbf{x}(S))^2]$$

- ▶ Stochastic gradient descent converges on **global** optimum
- ▶ Update rule is simple

$$\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) = \mathbf{x}(S_t) = \mathbf{x}_t \quad \implies \quad \Delta \mathbf{w} = \alpha(v_{\pi}(S_t) - v_{\mathbf{w}}(S_t)) \mathbf{x}_t$$

Update = **step-size** × **prediction error** × **feature vector**



## Incremental prediction algorithms

- ▶ We can't update towards the true value function  $v_\pi(s)$
- ▶ We substitute a **target** for  $v_\pi(s)$ 
  - ▶ For MC, the target is the return  $G_t$

$$\Delta \mathbf{w}_t = \alpha(\mathbf{G}_t - v_{\mathbf{w}}(s)) \nabla_{\mathbf{w}} v_{\mathbf{w}}(s)$$

- ▶ For TD, the target is the TD target  $R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1})$

$$\Delta \mathbf{w}_t = \alpha(\mathbf{R}_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)) \nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

- ▶ TD( $\lambda$ ):

$$\Delta \mathbf{w}_t = \alpha(\mathbf{G}_t^\lambda - v_{\mathbf{w}}(S_t)) \nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

$$G_t^\lambda = R_{t+1} + \gamma \left( (1 - \lambda)v_{\mathbf{w}}(S_{t+1}) + \lambda G_{t+1}^\lambda \right)$$



# Monte-Carlo with Value Function Approximation

- ▶ The return  $G_t$  is an **unbiased** sample of  $v_\pi(s)$
- ▶ Can therefore apply “supervised learning” to (online) “training data”:

$$\{(S_0, G_0), \dots, (S_t, G_t)\}$$

- ▶ For example, using **linear Monte-Carlo policy evaluation**

$$\begin{aligned}\Delta \mathbf{w}_t &= \alpha(G_t - v_{\mathbf{w}}(S_t)) \nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) \\ &= \alpha(G_t - v_{\mathbf{w}}(S_t)) \mathbf{x}_t\end{aligned}$$

- ▶ Linear Monte-Carlo evaluation converges to the global optimum
- ▶ Even when using non-linear value function approximation it converges (but perhaps to a local optimum)



# TD Learning with Value Function Approximation

- ▶ The TD-target  $R_{t+1} + \gamma v_w(S_{t+1})$  is a **biased** sample of true value  $v_\pi(S_t)$
- ▶ Can still apply supervised learning to “training data”:

$$\{(S_0, R_1 + \gamma v_w(S_1)), \dots, (S_t, R_{t+1} + \gamma v_w(S_{t+1}))\}$$

- ▶ For example, using **linear TD**

$$\begin{aligned}\Delta w_t &= \underbrace{\alpha (R_{t+1} + \gamma v_w(S_{t+1}) - v_w(S_t)) \nabla_w v_w(S_t)}_{= \delta_t, \text{ 'TD error'}} \\ &= \alpha \delta_t x_t\end{aligned}$$

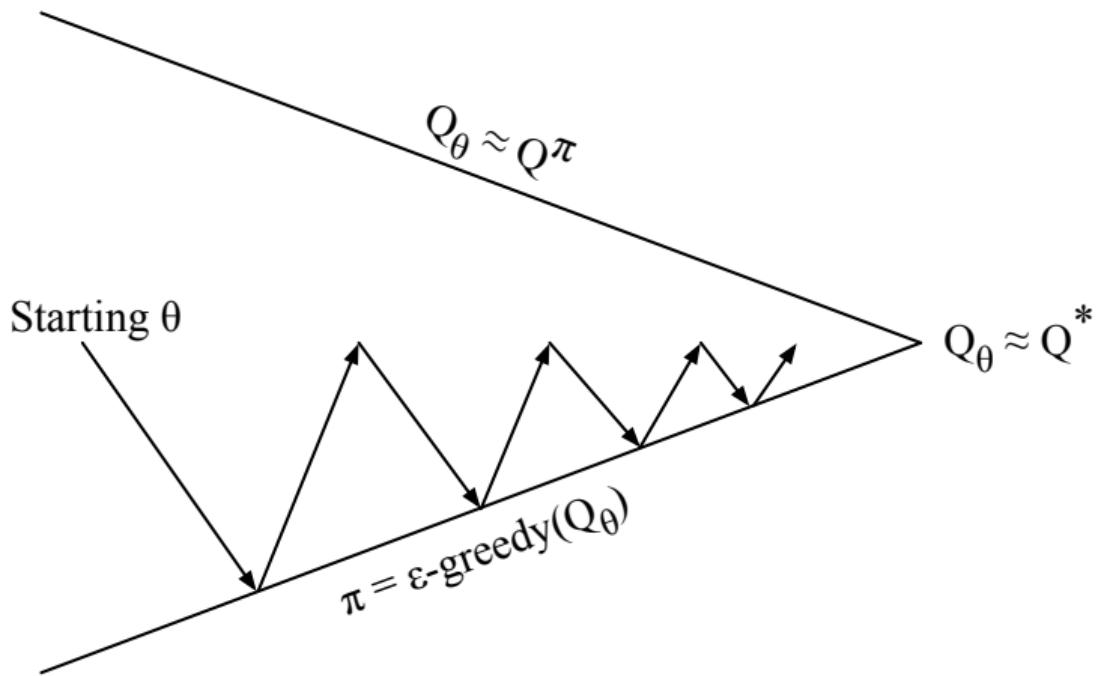
- ▶ This is akin to a non-stationary regression problem
- ▶ But it's a bit different: the target depends on our parameters!



# Control with value-function approximation



## Control with Value Function Approximation



Policy evaluation **Approximate** policy evaluation,  $q_w \approx q_\pi$

Policy improvement E.g.,  $\epsilon$ -greedy policy improvement



## Action-Value Function Approximation

- ▶ Approximate the action-value function  $q_w(s, a) \approx q_\pi(s, a)$
- ▶ For instance, with linear function approximation **with state-action features**

$$q_w(s, a) = \mathbf{x}(s, a)^\top \mathbf{w}$$

- ▶ Stochastic gradient descent update

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(q_\pi(s, a) - q_w(s, a)) \nabla_{\mathbf{w}} q_w(s, a) \\ &= \alpha(q_\pi(s, a) - q_w(s, a)) \mathbf{x}(s, a)\end{aligned}$$



# Action-Value Function Approximation

- ▶ Approximate the action-value function  $q_{\mathbf{w}}(s, a) \approx q_{\pi}(s, a)$
- ▶ For instance, with linear function approximation **with state features**

$$\begin{aligned}\mathbf{q}_{\mathbf{w}}(s) &= \mathbf{Wx}(s) & (\mathbf{W} \in \mathbb{R}^{m \times n}, \mathbf{x}(s) \in \mathbb{R}^n \implies \mathbf{q} \in \mathbb{R}^m) \\ q_{\mathbf{w}}(s, a) &= \mathbf{q}_{\mathbf{w}}(s)[a] = \mathbf{x}(s)^{\top} \mathbf{w}_a & (\text{where } \mathbf{w}_a = \mathbf{W}_a^{\top}).\end{aligned}$$

- ▶ Stochastic gradient descent update

$$\begin{aligned}\Delta \mathbf{w}_a &= \alpha(q_{\pi}(s, a) - q_{\mathbf{w}}(s, a)) \nabla_{\mathbf{w}} q_{\mathbf{w}}(s, a) \\ &= \alpha(q_{\pi}(s, a) - q_{\mathbf{w}}(s, a)) \mathbf{x}(s)\end{aligned}$$

$$\forall a \neq b : \Delta \mathbf{w}_b = 0$$

$$\text{Equivalently: } \Delta \mathbf{W} = \alpha(q_{\pi}(s, a) - q_{\mathbf{w}}(s, a)) \mathbf{i}_a \mathbf{x}(s)^{\top}$$

where  $\mathbf{i}_a = (0, \dots, 0, 1, 0, \dots, 0)$  with  $\mathbf{i}_a[a] = 1, \mathbf{i}_a[b] = 0$  for  $b \neq a$



# Action-Value Function Approximation

- ▶ Should we use action-in, or action-out?
  - ▶ Action in:  $q_{\mathbf{w}}(s, a) = \mathbf{w}^\top \mathbf{x}(s, a)$
  - ▶ Action out:  $\mathbf{q}_{\mathbf{w}}(s) = \mathbf{Wx}(s)$  such that  $q_{\mathbf{w}}(s, a) = \mathbf{q}_{\mathbf{w}}(s)[a]$
- ▶ One reuses the same weights, the other the same features
- ▶ Unclear which is better in general
- ▶ If we want to use continuous actions, action-in is easier (later lecture)
- ▶ For (small) discrete action spaces, action-out is common (e.g., DQN)



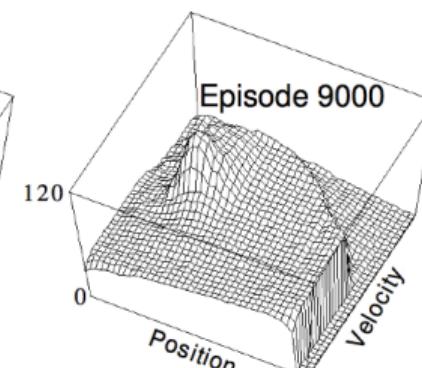
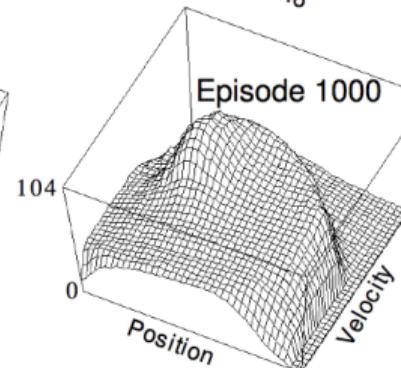
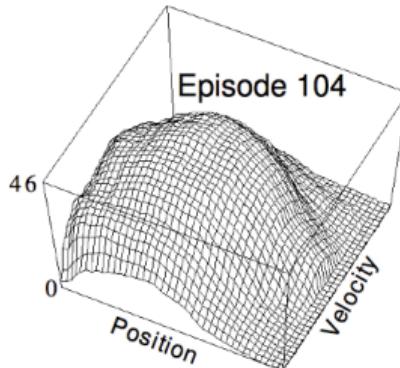
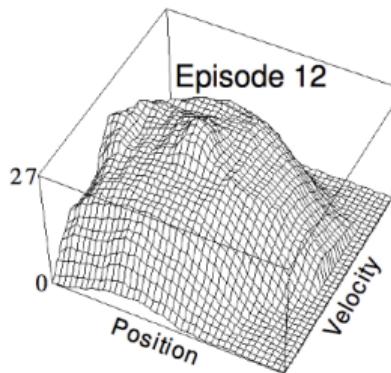
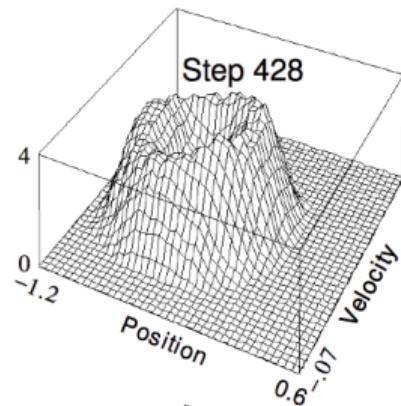
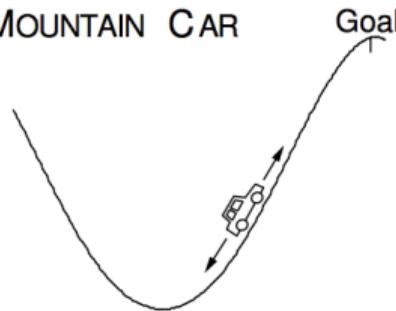
## Action-Value Function Approximation

- ▶ SARSA is TD applied to state-action pairs
- ▶  $\implies$  Inherits same properties
- ▶ But easier to do policy optimisation, and therefore policy iteration



# Linear Sarsa with Coarse Coding in Mountain Car

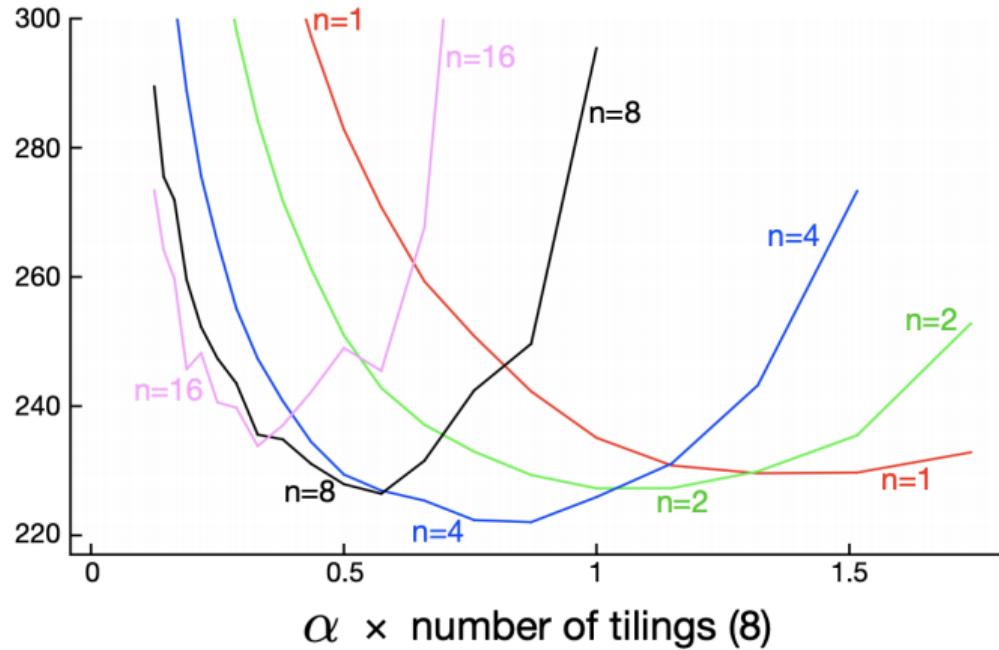
MOUNTAIN CAR



# Linear Sarsa with Tile Coding

## Mountain Car Steps per episode

averaged over  
first 50 episodes  
and 100 runs



Tile coding is similar to coarse coding:  
Overlaying different discretisations of the state space



# Convergence and divergence



# Convergence Questions

- ▶ When do incremental prediction algorithms converge?
  - ▶ When using **bootstrapping** (i.e. TD)?
  - ▶ When using (e.g., linear) value **function approximation**?
  - ▶ When using **off-policy** learning?
- ▶ Ideally, we would like algorithms that converge in all cases
- ▶ Alternatively, we want to understand when algorithms do, or do not, converge



## Convergence of MC

- With linear functions (and suitably decaying step size), MC converges to

$$\mathbf{w}_{\text{MC}} = \underset{\mathbf{w}}{\operatorname{argmin}} \mathbb{E}_{\pi}[(G_t - v_{\mathbf{w}}(S_t))^2] = \mathbb{E}_{\pi}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}_{\pi}[G_t \mathbf{x}_t]$$

(Notation: here the state distribution implicitly depends on  $\pi$ )

- Verifying the fixed point:

$$\begin{aligned}\nabla_{\mathbf{w}_{\text{MC}}} \mathbb{E}[(G_t - v_{\mathbf{w}_{\text{MC}}}(S_t))^2] &= \mathbb{E}[(G_t - v_{\mathbf{w}_{\text{MC}}}(S_t)) \mathbf{x}_t] = 0 \\ \mathbb{E}[(G_t - \mathbf{x}_t^\top \mathbf{w}_{\text{MC}}) \mathbf{x}_t] &= 0 \\ \mathbb{E}[G_t \mathbf{x}_t - \mathbf{x}_t \mathbf{x}_t^\top \mathbf{w}_{\text{MC}}] &= 0 \\ \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top] \mathbf{w}_{\text{MC}} &= \mathbb{E}[G_t \mathbf{x}_t] \\ \mathbf{w}_{\text{MC}} &= \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[G_t \mathbf{x}_t]\end{aligned}$$

- Agent state**  $S_t$  does not have to be Markov:  
the fixed point only depends on observed data (and features)



# Convergence of TD

- ▶ With linear functions, TD converges to

$$\mathbf{w}_{\text{TD}} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]^{-1} \mathbb{E}[R_{t+1} \mathbf{x}_t]$$

(in continuing problems with fixed  $\gamma < 1$ , and with appropriately decaying  $\alpha_t \rightarrow 0$ )

- ▶ Verify (assuming  $\alpha_t$  does not correlate with  $R_{t+1}, \mathbf{x}_t, \mathbf{x}_{t+1}$ ):

$$\begin{aligned}\mathbb{E}[\Delta \mathbf{w}_{\text{TD}}] &= 0 = \mathbb{E}[\alpha_t(R_{t+1} + \gamma \mathbf{x}_{t+1}^\top \mathbf{w}_{\text{TD}} - \mathbf{x}_t^\top \mathbf{w}_{\text{TD}}) \mathbf{x}_t] \\ 0 &= \mathbb{E}[\alpha_t R_{t+1} \mathbf{x}_t] + \mathbb{E}[\alpha_t \mathbf{x}_t (\gamma \mathbf{x}_{t+1}^\top - \mathbf{x}_t^\top) \mathbf{w}_{\text{TD}}]\end{aligned}$$

$$\begin{aligned}\mathbb{E}[\alpha_t \mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top] \mathbf{w}_{\text{TD}} &= \mathbb{E}[\alpha_t R_{t+1} \mathbf{x}_t] \\ \mathbf{w}_{\text{TD}} &= \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]^{-1} \mathbb{E}[R_{t+1} \mathbf{x}_t]\end{aligned}$$

- ▶ This **differs** from the MC solution
- ▶ Typically, the asymptotic MC solution is preferred (smallest prediction error)
- ▶ TD often converges faster (especially intermediate  $\lambda \in [0, 1]$  or  $n \in \{1, \dots, \infty\}$ )



## Convergence of TD

- With linear functions, TD converges to

$$\mathbf{w}_{\text{TD}} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]^{-1} \mathbb{E}[R_{t+1} \mathbf{x}_t]$$

- Let  $\overline{\text{VE}}(\mathbf{w})$  denote the **value error**:

$$\overline{\text{VE}}(\mathbf{w}) = \|v_\pi - v_{\mathbf{w}}\|_{d_\pi} = \sum_{s \in S} d_\pi(s)(v_\pi(s) - v_{\mathbf{w}}(s))^2$$

- The Monte Carlo solution minimises the value error

### Theorem

$$\overline{\text{VE}}(\mathbf{w}_{\text{TD}}) \leq \frac{1}{1-\gamma} \overline{\text{VE}}(\mathbf{w}_{\text{MC}}) = \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w})$$



## TD is not a gradient

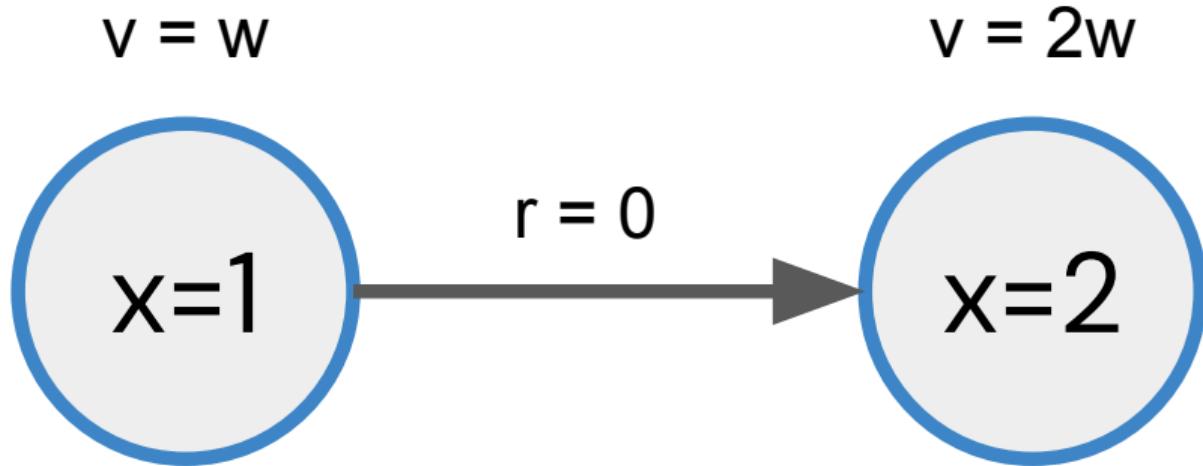
- ▶ The TD update is not a true gradient update:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(r + \gamma v_{\mathbf{w}}(s') - v_{\mathbf{w}}(s)) \nabla v_{\mathbf{w}}(s)$$

- ▶ That's okay: it is a **stochastic approximation** update
- ▶ Stochastic approximation algorithms are a broader class than just SGD
- ▶ SGD always converges (with bounded noise, decaying step size, stationarity, ...)
- ▶ TD does **not** always converge...



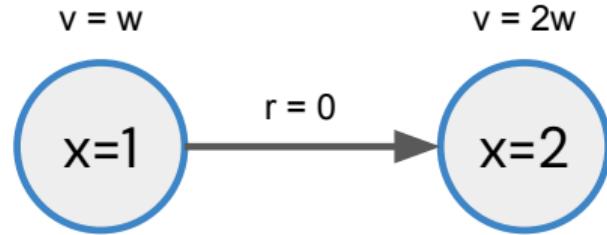
## Example of divergence



What if we use TD only on this transition?



## Example of divergence

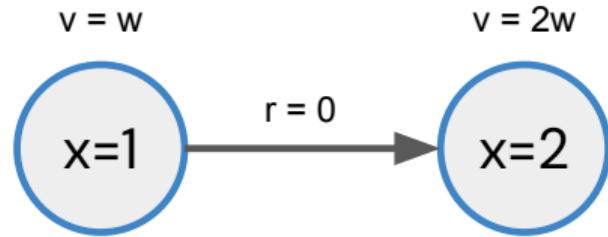


$$\begin{aligned}w_{t+1} &= w_t + \alpha_t(r + \gamma v(s') - v(s))\nabla v(s) \\&= w_t + \alpha_t(r + \gamma v(s') - v(s))x(s) \\&= w_t + \alpha_t(0 + \gamma 2w_t - w_t) \\&= w_t + \alpha_t(2\gamma - 1)w_t\end{aligned}$$

Consider  $w_t > 0$ . If  $\gamma > \frac{1}{2}$ , then  $w_{t+1} > w_t$ .  
 $\implies \lim_{t \rightarrow \infty} w_t = \infty$



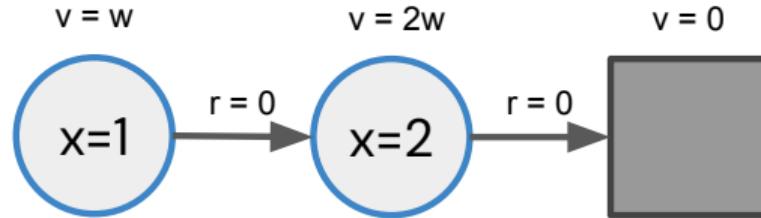
## Example of divergence



- ▶ Algorithms that combine
  - ▶ bootstrapping
  - ▶ off-policy learning, and
  - ▶ function approximation
- ...may diverge
- ▶ This is sometimes called the **deadly triad**



## Deadly triad



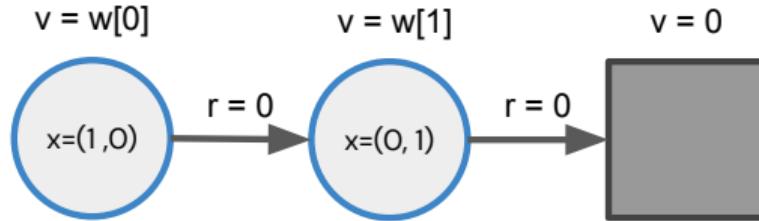
- ▶ Consider sampling **on-policy**, over an episode. Update:

$$\begin{aligned}\Delta w &= \alpha(0 + 2\gamma w - w) + \alpha(0 + \gamma 0 - 2w) \\ &= \alpha(2\gamma - 3)w\end{aligned}$$

- ▶ The multiplier is negative, for all  $\gamma \in [0, 1]$
- ▶  $\implies$  convergence ( $w$  goes to zero, which is optimal here)



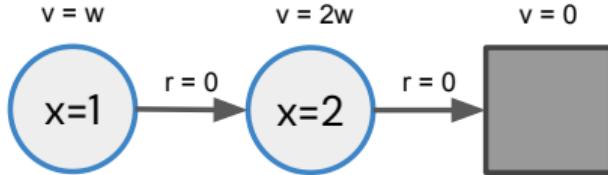
# Deadly triad



- ▶ With tabular features, this is just regression
- ▶ Answer may be sub-optimal, but no divergence occurs
- ▶ Specifically, if we only update  $v(s)$  (=left-most state):
  - ▶  $v(s) = w[0]$  will converge to  $\gamma v(s')$
  - ▶  $v(s') = w[1]$  will stay where it was initialised



# Deadly triad



- ▶ What if we use multi-step returns?
- ▶ Still consider only updating the left-most state

$$\begin{aligned}\Delta w &= \alpha(r + \gamma(G_t^\lambda - v(s))) \\ &= \alpha(r + \gamma((1 - \lambda)v(s') + \lambda(r' + v(s''))) - v(s)) \quad (r = r' = v(s'') = 0) \\ &= \alpha(2\gamma(1 - \lambda) - 1)w\end{aligned}$$

- ▶ The multiplier is negative when  $2\gamma(1 - \lambda) < 1 \implies \lambda > 1 - \frac{1}{2\gamma}$
- ▶ E.g., when  $\gamma = 0.9$ , then we need  $\lambda > 4/9 \approx 0.45$



## Residual Bellman updates

$$\text{TD: } \Delta \mathbf{w}_t = \alpha \delta \nabla v_{\mathbf{w}}(S_t) \quad \text{where} \quad \delta_t = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)$$

- ▶ This update ignores dependence of  $v_{\mathbf{w}}(S_{t+1})$  on  $\mathbf{w}$
- ▶ Alternative: **Bellman residual gradient** update

$$\text{loss: } \mathbb{E}[\delta_t^2] \quad \text{update: } \Delta \mathbf{w}_t = \alpha \delta_t \nabla_{\mathbf{w}}(v_{\mathbf{w}}(S_t) - \gamma v_{\mathbf{w}}(S_{t+1}))$$

- ▶ This tends to **work worse** in practice
- ▶ Bellman residuals smooth, whereas TD methods predict
- ▶ Smoothed values may lead to suboptimal decisions



## Residual Bellman updates

- ▶ Alternative: minimise the **Bellman error**

$$\text{loss: } \mathbb{E}[\delta_t]^2 \quad \text{update: } \Delta \mathbf{w}_t = \alpha \delta_t \nabla_{\mathbf{w}}(v_{\mathbf{w}}(S_t) - \gamma v_{\mathbf{w}}(S'_{t+1}))$$

- ▶ ...but requires a second independent sample  $S'_{t+1}$  which could (randomly) differ from  $S_{t+1}$   
(So we can't use this online)



# Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗



## Convergence of Control Algorithms

- ▶ Tabular control learning algorithms (e.g., Q-learning) can be extended to FA (e.g., Deep Q Network — DQN)
- ▶ The theory of control with function approximation is not fully developed
- ▶ **Tracking** is often preferred to convergence  
(I.e., continually adapting the policy instead of converging to a fixed policy)



# Batch Methods



# Batch Reinforcement Learning

- ▶ Gradient descent is simple and appealing
- ▶ But it is not **sample** efficient
- ▶ Batch methods seek to find the best fitting value function for a given a set of past experience ("training data")



# Least Squares Temporal Difference

- ▶ Which parameters  $\mathbf{w}$  give the **best fitting** linear value function  $v_{\mathbf{w}}(s) = \mathbf{w}^T \mathbf{x}(s)$ ? Recall:

$$\begin{aligned}\mathbb{E}[(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)) \mathbf{x}_t] &= \mathbf{0} \\ \implies \mathbf{w}_{\text{TD}} &= \mathbb{E}[\mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T]^{-1} \mathbb{E}[R_{t+1} \mathbf{x}_t]\end{aligned}$$

- ▶ We can use a closed-form **empirical loss**:

$$\begin{aligned}\frac{1}{t} \sum_{i=0}^t (R_{i+1} + \gamma v_{\mathbf{w}}(S_{i+1}) - v_{\mathbf{w}}(S_i)) \mathbf{x}_i &= \mathbf{0} \\ \implies \mathbf{w}_{\text{LSTD}} &= \left( \sum_{i=0}^t \mathbf{x}_i (\mathbf{x}_i - \gamma \mathbf{x}_{i+1})^T \right)^{-1} \left( \sum_{i=0}^t R_{i+1} \mathbf{x}_i \right)\end{aligned}$$

- ▶ This is called least-squares TD (**LSTD**)



# Least Squares Temporal Difference

$$\mathbf{w}_t = \underbrace{\left( \sum_{i=0}^t \mathbf{x}_i (\mathbf{x}_i - \gamma \mathbf{x}_{i+1})^\top \right)^{-1}}_{= \mathbf{A}_t^{-1}} \underbrace{\left( \sum_{i=0}^t R_{i+1} \mathbf{x}_i \right)}_{= \mathbf{b}_t} \quad (\text{LSTD estimate})$$

- We can update  $\mathbf{b}_t$  and  $\mathbf{A}_t^{-1}$  incrementally **online**
- **Naive approach** ( $O(n^3)$ )

$$\mathbf{A}_{t+1} = \mathbf{A}_t + \mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top \quad \mathbf{b}_{t+1} = \mathbf{b}_t + R_{t+1} \mathbf{x}_t$$

- **Faster approach** ( $O(n^2)$ ): directly update  $\mathbf{A}^{-1}$  with Sherman-Morrison:

$$\mathbf{A}_{t+1}^{-1} = \mathbf{A}_t^{-1} - \frac{\mathbf{A}_t^{-1} \mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top \mathbf{A}_t^{-1}}{1 + (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top \mathbf{A}_t^{-1} \mathbf{x}_t} \quad \mathbf{b}_{t+1} = \mathbf{b}_t + R_{t+1} \mathbf{x}_t$$

- Still more compute per step than TD ( $O(n)$ )



## Least Squares Temporal Difference

- ▶ In the limit, LSTD and TD converge to the same fixed point
- ▶ We can extend LSTD to multi-step returns:  $\text{LSTD}(\lambda)$
- ▶ We can extend LSTD to action values:  $\text{LSTDQ}$
- ▶ We can also interlace with policy improvement:  
least-squares policy iteration (LSPI)



# Experience Replay

Given experience consisting of trajectories of experience

$$\mathcal{D} = \{S_0, A_0, R_1, S_1, \dots, S_t\}$$

Repeat:

1. Sample transition(s), e.g.,  $(S_n, A_n, R_{n+1}, S_{n+1})$  for  $n \leq t$
2. Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(R_{n+1} + \gamma v_{\mathbf{w}}(S_{n+1}) - v_{\mathbf{w}}(S_n)) \nabla_{\mathbf{w}} v_{\mathbf{w}}(S_n)$$

3. Can re-use old data

This is also a form of batch learning

Beware: the data may be off-policy if the policy changes



# Deep reinforcement learning

(briefly, more later)



# Deep neural networks (blackboard)



# Deep reinforcement learning

- ▶ Many ideas immediately transfer when using deep neural networks:
  - ▶ TD and MC
  - ▶ Double learning (e.g., double Q-learning)
  - ▶ Experience replay
  - ▶ ...
- ▶ Some ideas do not easily transfer
  - ▶ UCB
  - ▶ Least squares TD/MC



## Example: neural Q-learning

- ▶ Online neural Q-learning may include:
  - ▶ **Neural network:**  $O_t \mapsto \mathbf{q}_w$  (action-out)
  - ▶ **Exploration policy:**  $\pi_t = \epsilon\text{-greedy}(\mathbf{q}_t)$ , and then  $A_t \sim \pi_t$
  - ▶ **Weight update:** for instance Q-learning

$$\Delta w \propto \left( R_{t+1} + \gamma \max_a q_w(S_{t+1}, a) - q_w(S_t, A_t) \right) \nabla_w q_w(S_t, A_t)$$

- ▶ An **optimizer** to minimize the loss (e.g., SGD, RMSProp, Adam)
- ▶ Often, we implement the weight update via a ‘loss’

$$L(w) = \frac{1}{2} \left( R_{t+1} + \gamma \left[ \max_a q_w(S_{t+1}, a) \right] - q_w(S_t, A_t) \right)^2$$

where  $\llbracket \cdot \rrbracket$  denotes stopping the gradient, so that the **semi-gradient** is  $\Delta w$

- ▶ Note that  $L(w)$  is not a real loss, it just happens to have the right gradient



## Example: DQN

- ▶ DQN (Mnih et al. 2013, 2015) includes:
  - ▶ A **neural network**:  $O_t \mapsto q_w$  (action-out)
  - ▶ An **exploration policy**:  $\pi_t = \epsilon\text{-greedy}(q_t)$ , and then  $A_t \sim \pi_t$
  - ▶ A **replay buffer** to store and sample past transitions  $(S_i, A_i, R_{i+1}, S_{i+1})$
  - ▶ **Target network parameters**  $w^-$
  - ▶ A Q-learning **weight update** on  $w$  (uses replay and target network)

$$\Delta w = \left( R_{i+1} + \gamma \max_a q_{w^-}(S_{i+1}, a) - q_w(S_i, A_i) \right) \nabla_w q_w(S_i, A_i)$$

- ▶ Update  $w_t^- \leftarrow w_t$  occasionally (e.g., every 10000 steps)
- ▶ An **optimizer** to minimize the loss (e.g., SGD, RMSprop, or Adam)
- ▶ Replay and target networks make RL look more like supervised learning
- ▶ Neither is strictly necessary, but they helped for DQN
- ▶ “DL-aware RL”



# End of Lecture



# Planning and models

Matteo Hessel

2021

## Recap

In the previous lectures:

- ▶ **Bandits**: how to trade-off exploration and exploitation.
- ▶ **Dynamic Programming**: how to solve prediction and control given full knowledge of the environment.
- ▶ **Model-free prediction and control**: how to solve prediction and control from interacting with the environment.
- ▶ **Function approximation**: how to generalise what you learn in large state spaces.

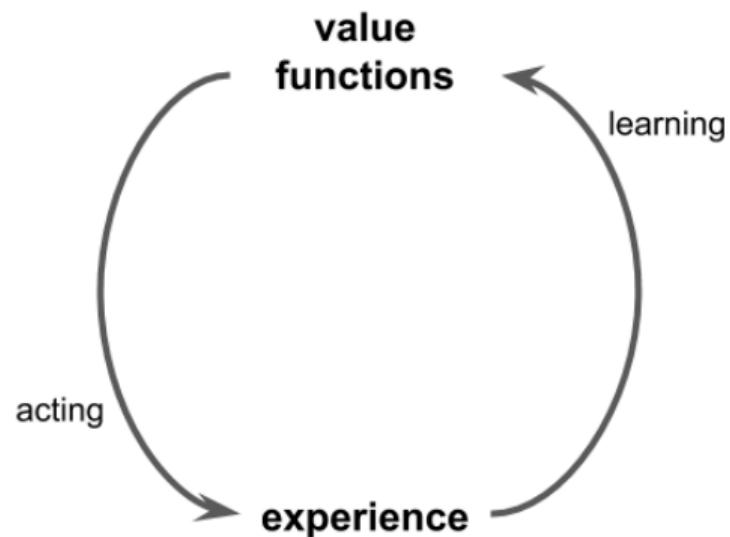
# Dynamic Programming and Model-Free RL

- ▶ Dynamic Programming
  - ▶ Assume a model
  - ▶ **Solve** model, no need to interact with the world at all.
- ▶ Model-Free RL
  - ▶ No model
  - ▶ **Learn** value functions from experience.

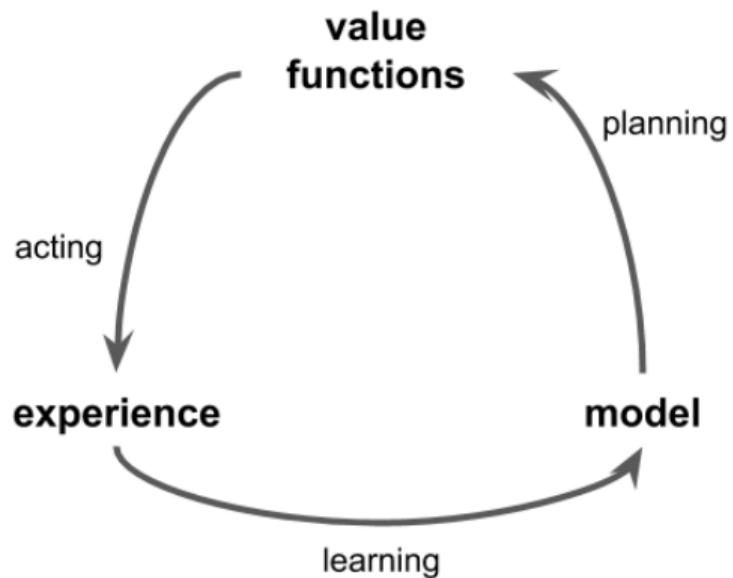
# Model-Based RL

- ▶ Model-Based RL
  - ▶ **Learn** a model from experience
  - ▶ **Plan** value functions using the learned model.

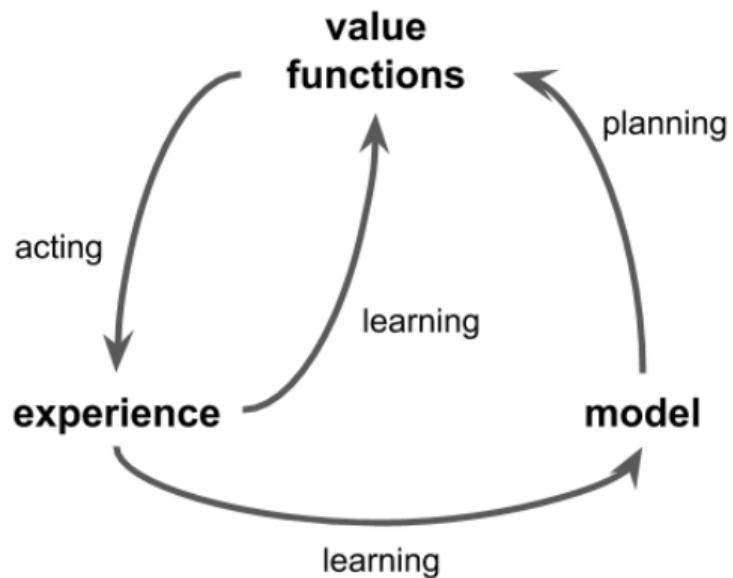
# Model-Free RL



# Model-Based RL



# Model-Based RL



## Why should we even consider this?

One clear disadvantage:

- ▶ First learn a model, then construct a value function
  - ⇒ two sources of approximation error
- ▶ Learn a value function directly
  - ⇒ only one source of approximation error

However:

- ▶ Models can efficiently be learned by supervised learning methods
- ▶ Reason about model uncertainty (better exploration?)
- ▶ Reduce the interactions in the real world (data efficiency? faster/cheaper?).

# Learning a Model

Matteo Hessel

2021

## What is a Model?

A **model**  $\mathcal{M}_\eta$  is an approximate representation of an MDP  $\langle \mathcal{S}, \mathcal{A}, \hat{p} \rangle$ ,

- ▶ For now, we will assume the states and actions are the same as in the real problem
- ▶ That the dynamics ,  $\hat{p}_\eta$  is parametrised by some set of weights  $\eta$
- ▶ The model directly approximates the state transitions and rewards  $\hat{p}_\eta \approx p$ :

$$R_{t+1}, S_{t+1} \sim \hat{p}_\eta(r, s' \mid S_t, A_t)$$

## Model Learning - I

Goal: estimate model  $\mathcal{M}_\eta$  from experience  $\{S_1, A_1, R_2, \dots, S_T\}$

- ▶ This is a supervised learning problem

$$S_1, A_1 \rightarrow R_2, S_2$$

⋮

$$S_{T-1}, A_{T-1} \rightarrow R_T, S_T$$

- ▶ over a dataset of state transitions observed in the environment.

## Model Learning - II

How do we learn a suitable function  $f_\eta(s, a) = r, s'$ ?

- ▶ Choose a functional form for  $f$
- ▶ Pick loss function (e.g. mean-squared error),
- ▶ Find parameters  $\eta$  that minimise empirical loss
- ▶ This would give an **expectation model**
- ▶ If  $f_\eta(s, a) = r, s'$ , then we would hope  $s' \approx \mathbb{E}[S_{t+1} | s = S_t, a = A_t]$

## Expectation Models

- ▶ Expectation models can have disadvantages:
  - ▶ Image that an action randomly goes left or right past a wall
  - ▶ Expectation models can interpolate and put you **in** the wall
- ▶ But with linear values, we are mostly alright:
  - ▶ Consider an expectation model  $f_\eta(\phi_t) = \mathbb{E}[\phi_{t+1}]$  and value function  $v_\theta(\phi_t) = \theta^\top \phi_t$

$$\begin{aligned}\mathbb{E}[v_\theta(\phi_{t+1}) \mid S_t = s] &= \mathbb{E}[\theta^\top \phi_{t+1} \mid S_t = s] \\ &= \theta^\top \mathbb{E}[\phi_{t+1} \mid S_t = s] \\ &= v_\theta(\mathbb{E}[\phi_{t+1} \mid S_t = s]).\end{aligned}$$

- ▶ If the model is also linear:  $f_\eta(\phi_t) = P\phi_t$  for some matrix  $P$ .
  - ▶ then we can even unroll an expectation model even multiple steps into the future,
  - ▶ and still have  $\mathbb{E}[v_\theta(\phi_{t+n}) \mid S_t = s] = v_\theta(\mathbb{E}[\phi_{t+n} \mid S_t = s])$

## Stochastic Models

- ▶ We may not want to assume everything is linear
- ▶ Then, expected states may not be right — they may not correspond to actual states, and iterating the model may do weird things
- ▶ Alternative: **stochastic models** (also known as **generative models**)

$$\hat{R}_{t+1}, \hat{S}_{t+1} = \hat{p}(S_t, A_t, \omega)$$

where  $\omega$  is a noise term

- ▶ Stochastic models can be chained, even if the model is non-linear
- ▶ But they do add noise

## Full Models

- ▶ We can also try to model the complete transition dynamics
- ▶ It can be hard to iterate these, because of branching:

$$\mathbb{E}[v(S_{t+1}) \mid S_t = s] = \sum_a \pi(a \mid s) \sum_{s'} \hat{p}(s, a, s') (\hat{r}(s, a, s') + \gamma v(s'))$$

$$\begin{aligned} \mathbb{E}[v(S_{t+n}) \mid S_t = s] &= \sum_a \pi(a \mid s) \sum_{s'} \hat{p}(s, a, s') \left( \hat{r}(s, a, s') + \right. \\ &\quad \left. \gamma \sum_{a'} \pi(a' \mid s') \sum_{s''} \hat{p}(s', a', s'') \left( \hat{r}(s', a', s'') + \right. \right. \\ &\quad \left. \left. \gamma^2 \sum_{a''} \pi(a'' \mid s'') \sum_{s'''} \hat{p}(s'', a'', s''') \left( \hat{r}(s'', a'', s''') + \dots \right) \right) \right) \end{aligned}$$

## Examples of Models

We typically decompose the dynamics  $p_\eta$  into separate parametric functions

- ▶ for transition and reward dynamics

For each of these we can then consider different options:

- ▶ Table Lookup Model
- ▶ Linear Expectation Model
- ▶ Deep Neural Network Model

## Table Lookup Models

- ▶ Model is an explicit MDP
- ▶ Count visits  $N(s, a)$  to each state action pair

$$\hat{p}_t(s' \mid s, a) = \frac{1}{N(s, a)} \sum_{k=0}^{t-1} I(S_k = s, A_k = a, S_{k+1} = s')$$

$$\mathbb{E}_{\hat{p}_t}[R_{t+1} \mid S_t = s, A_t = a] = \frac{1}{N(s, a)} \sum_{k=0}^{t-1} I(S_k = s, A_k = a) R_{k+1}$$

## AB Example

Two states  $A, B$ ; no discounting; 8 episodes of experience

A, 0, B, 0

B, 1

B, 1

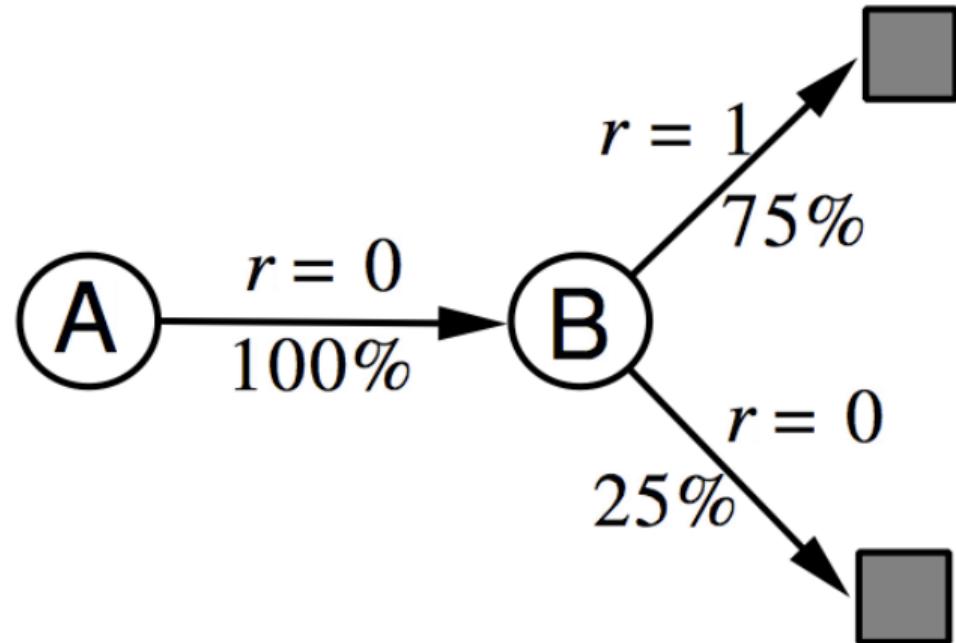
B, 1

B, 1

B, 1

B, 1

B, 0



We have constructed a **table lookup model** from the experience

## Linear expectation models

In linear expectation models

- ▶ we assume some feature representation  $\phi$  is given
- ▶ so that we can encode any state  $s$  as  $\phi(s)$
- ▶ we then parametrise separately rewards and transitions
- ▶ each as a linear function of the features

## Linear expectation models

- ▶ expected next states are parametrised by a square matrix  $T_a$ , for each action  $a$

$$\hat{s}'(s, a) = T_a \phi(s)$$

- ▶ the rewards are parametrised by a vector  $w_a$ , for each action  $a$

$$\hat{r}(s, a) = w_a^T \phi(s)$$

- ▶ On each transition  $(s, a, r, s')$  we can then apply a gradient descent step
- ▶ to update  $w_a$  and  $T_a$  so as to minimise the loss:

$$L(s, a, r, s') = (s' - T_a \phi(s))^2 + (r - w_a^T \phi(s))^2$$

# Planning for Credit Assignment

Matteo Hessel

2021

# Planning

In this section we investigate **planning**

- ▶ This concept means different things to different communities
- ▶ For us planning is the process of investing compute to improve values and policies
- ▶ Without the need to interact with the environment
- ▶ Dynamic programming is the best example we have seen so far
- ▶ We are interested in planning algorithms that don't require privileged access to a perfect specification of the environment
- ▶ Instead, the planning algorithms we discuss today use **learned models**

## Dynamic Programming with a learned Model

Once learned a model  $\hat{p}_\eta$  from experience:

- ▶ Solve the MDP  $\langle \mathcal{S}, \mathcal{A}, \hat{p}_\eta \rangle$
- ▶ Using favourite dynamic programming algorithm
  - ▶ Value iteration
  - ▶ Policy iteration
  - ▶ ...

## Sample-Based Planning with a learned Model

A simple but powerful approach to planning:

- ▶ Use the model **only** to generate samples
- ▶ **Sample** experience from model

$$S, R \sim \hat{p}_\eta(\cdot \mid s, a)$$

- ▶ Apply **model-free** RL to samples, e.g.:
  - ▶ Monte-Carlo control
  - ▶ Sarsa
  - ▶ Q-learning

## Back to the AB Example

- ▶ Construct a table-lookup model from real experience
- ▶ Apply model-free RL to sampled experience

Real experience

A, 0, B, 0

B, 1

B, 1

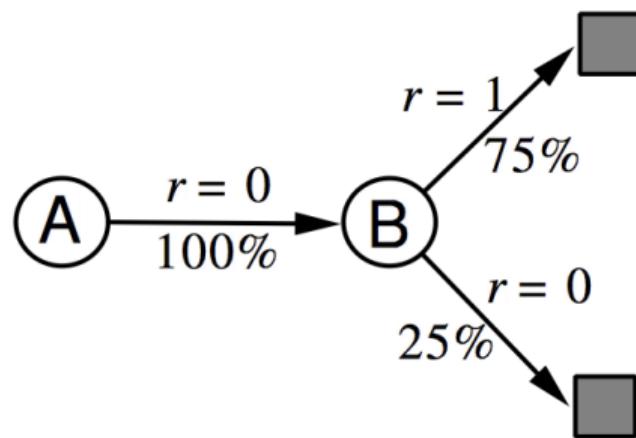
B, 1

B, 1

B, 1

B, 1

B, 0



Sampled experience

B, 1

B, 0

B, 1

A, 0, B, 1

B, 1

A, 0, B, 1

B, 1

B, 0

e.g. Monte-Carlo learning:  $V(A) = 1, V(B) = 0.75$

## Limits of Planning with an Inaccurate Model - I

Given an imperfect model  $\hat{p}_\eta \neq p$ :

- ▶ The planning process may compute a suboptimal policy
- ▶ Performance is limited to optimal policy for approximate MDP  $\langle \mathcal{S}, \mathcal{A}, \hat{p}_\eta \rangle$
- ▶ Model-based RL is only as good as the estimated model

## Limits of Planning with an Inaccurate Model - II

How can we deal with the inevitable inaccuracies of a learned model?

- ▶ Approach 1: when model is wrong, use model-free RL
- ▶ Approach 2: reason about model uncertainty over  $\eta$  (e.g. Bayesian methods)
- ▶ Approach 3: Combine model-based and model-free methods in a single algorithm.

## Real and Simulated Experience

We consider two sources of experience

Real experience Sampled from environment (true MDP)

$$r, s' \sim p$$

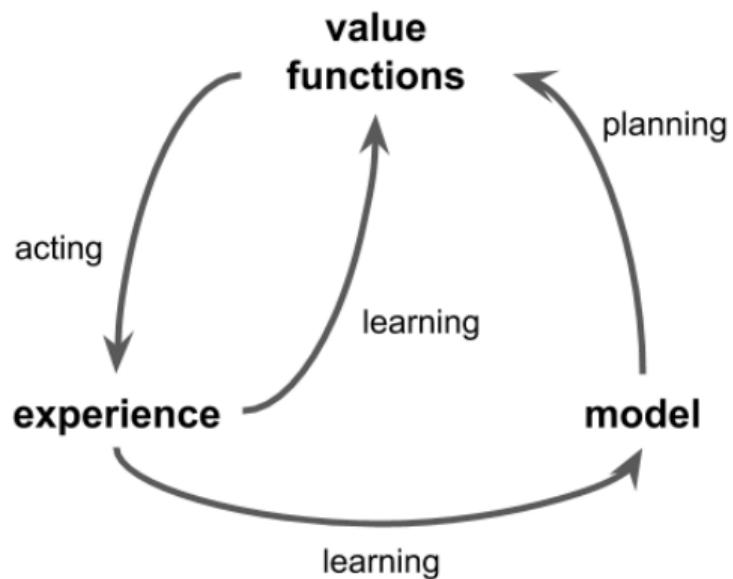
Simulated experience Sampled from model (approximate MDP)

$$r, s' \sim \hat{p}_\eta$$

# Integrating Learning and Planning

- ▶ Model-Free RL
  - ▶ No model
  - ▶ **Learn** value function (and/or policy) from real experience
- ▶ Model-Based RL (using Sample-Based Planning)
  - ▶ Learn a model from real experience
  - ▶ **Plan** value function (and/or policy) from simulated experience
- ▶ Dyna
  - ▶ Learn a model from real experience
  - ▶ **Learn AND plan** value function (and/or policy) from real and simulated experience
  - ▶ Treat real and simulated experience equivalently. Conceptually, the updates from learning or planning are not distinguished.

# Dyna Architecture



## Dyna-Q Algorithm

Initialize  $Q(s, a)$  and  $Model(s, a)$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$

Do forever:

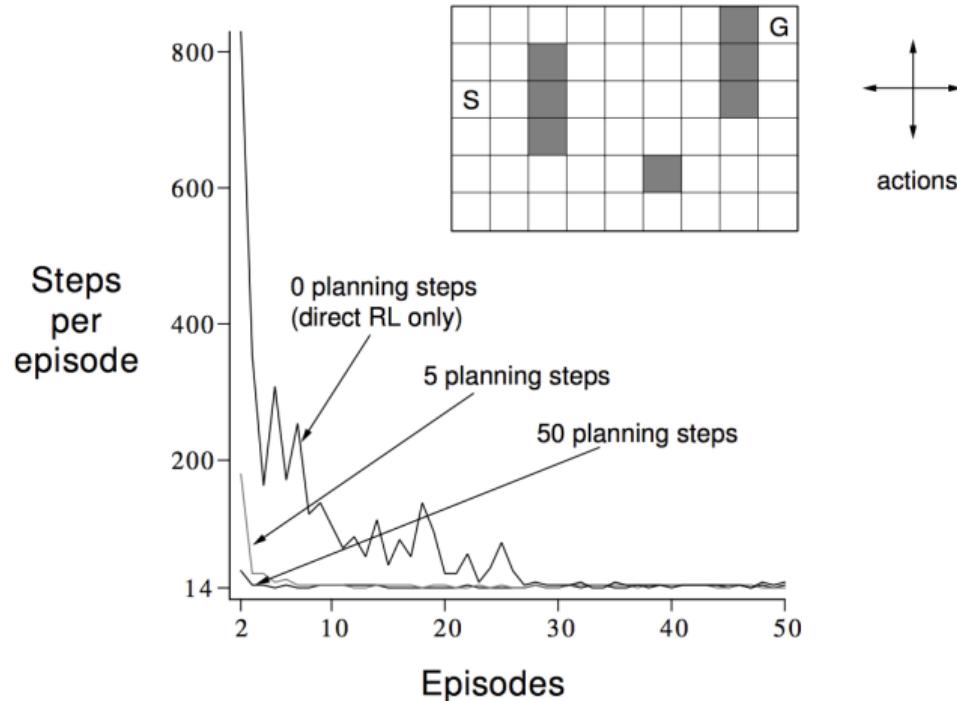
- (a)  $s \leftarrow$  current (nonterminal) state
- (b)  $a \leftarrow \varepsilon\text{-greedy}(s, Q)$
- (c) Execute action  $a$ ; observe resultant state,  $s'$ , and reward,  $r$
- (d)  $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
- (e)  $Model(s, a) \leftarrow s', r$  (assuming deterministic environment)
- (f) Repeat  $N$  times:
  - $s \leftarrow$  random previously observed state
  - $a \leftarrow$  random action previously taken in  $s$
  - $s', r \leftarrow Model(s, a)$
  - $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

## Advantages of combining learning and planning.

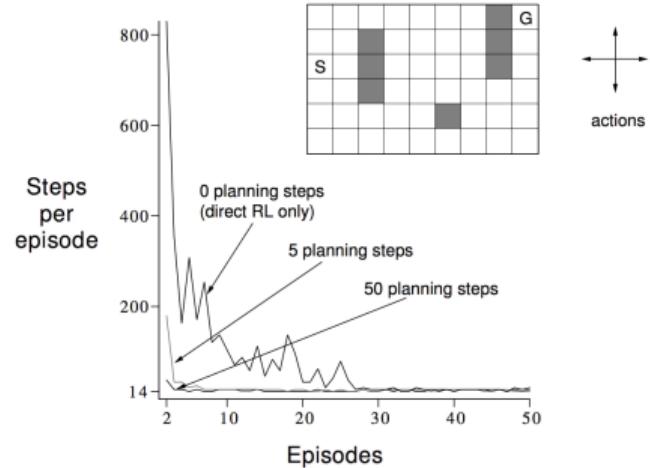
What are the advantages of this architecture?

- ▶ We can sink in more compute in order to learn more efficiently.
- ▶ This is especially important when collecting real data is
  - ▶ expensive / slow (e.g. robotics)
  - ▶ unsafe (e.g. autonomous driving)

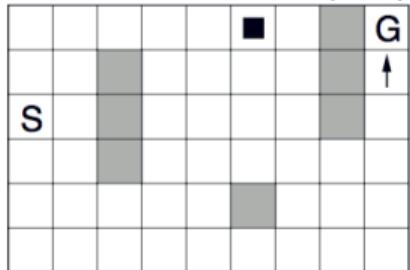
# Dyna-Q on a Simple Maze



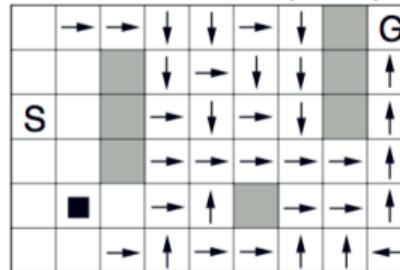
# Dyna-Q on a Simple Maze



WITHOUT PLANNING ( $n=0$ )

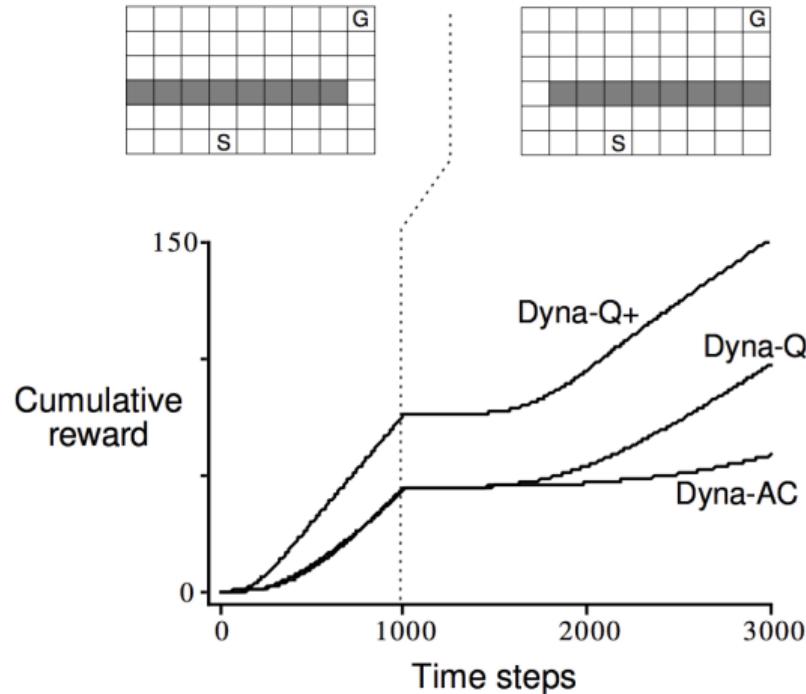


WITH PLANNING ( $n=50$ )



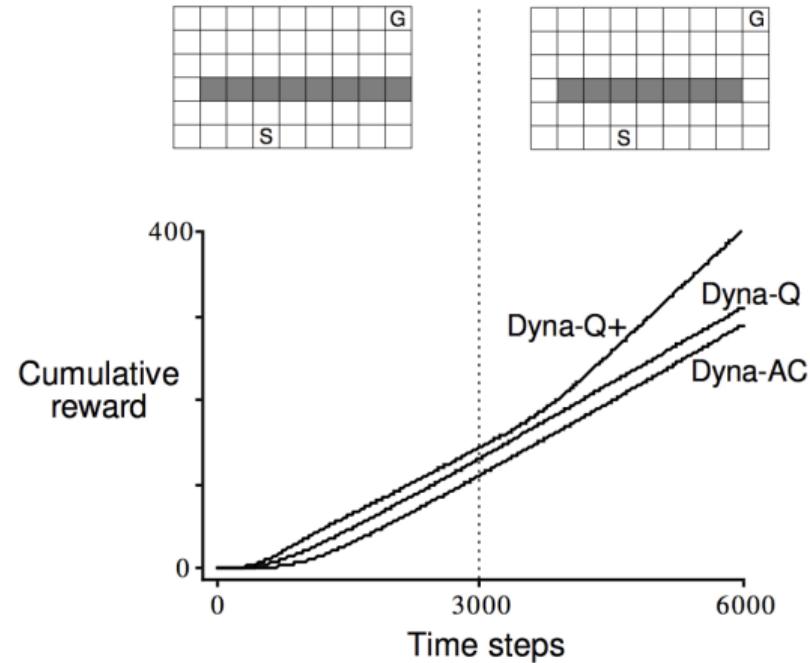
## Dyna-Q with an Inaccurate Model

- The changed environment is **harder**



## Dyna-Q with an Inaccurate Model (2)

- The changed environment is **easier**



# Planning and Experience Replay

Matteo Hessel

2021

## Conventional model-based and model-free methods

Traditional RL algorithms did not explicitly store their experiences,  
It was easy to place them into one of two groups.

- ▶ **Model-free** methods update the value function and/or policy and do not have explicit dynamics models.
- ▶ **Model-based** methods update the transition and reward models, and compute a value function or policy from the model.

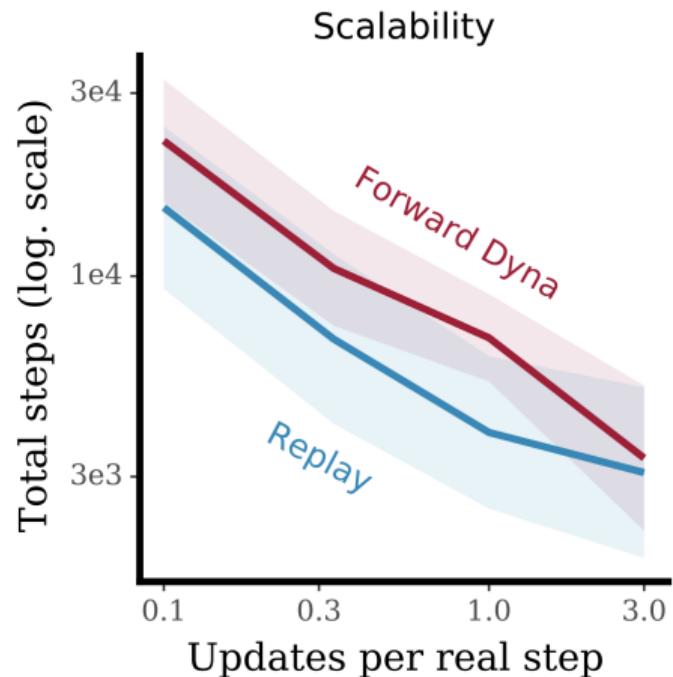
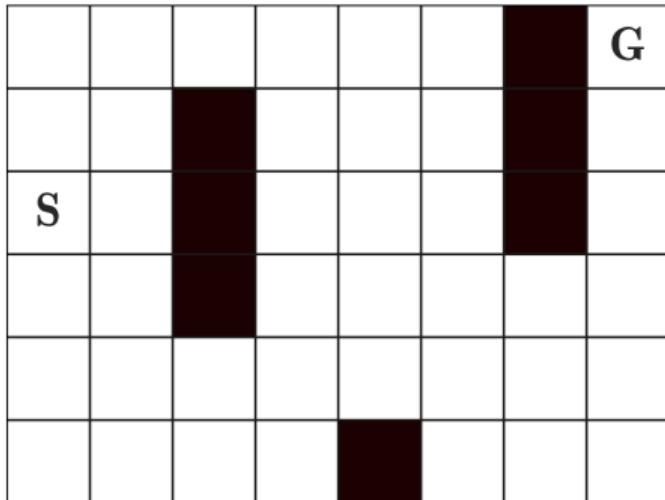
## Moving beyond model-based and model-free labels

The sharp distinction between model-based and model-free is now less useful:

1. Often agents store transitions in an *experience replay buffer*
2. Model-free RL is then applied to experience sampled from the replay buffer,
3. This is just Dyna, with the experience replay as a non-parametric model
  - ▶ we plan by sampling an entire transition  $(s, a, r, s')$ ,
  - ▶ instead of sampling just a state-action  $(s, a)$  and inferring  $r, s'$  from the model.
  - ▶ we can still sink in compute to make learning more efficient,
  - ▶ by making many updates on past data for every new step we take in the environment.

# Scalability

The maze



## Comparing parametric model and experience replay - I

- ▶ For tabular RL there is an exact output equivalence between some conventional model-based and model free algorithms.
- ▶ If the model is perfect, it will give the same output as a non-parametric replay system for every  $(s, a)$  pair
- ▶ In practice, the model is not perfect, so there will be differences
- ▶ Could model inaccuracies lead to better learning?
- ▶ Unlikely if we only use the model to sample imagined transitions from the actual past state-action pairs.
- ▶ But a parametric model is more flexible than a replay buffer

## Comparing parametric model and experience replay - II

- ▶ Plan for action-selection!
  - ▶ query a model for action that you \*could\* take in the future
- ▶ Counterfactual planning.
  - ▶ query a model for action that you \*could\* have taken in the past, but did not

## Comparing parametric model and experience replay - III

- ▶ Backwards planning
  - ▶ model the inverse dynamics and assign credit to different states that \*could\* have led to a certain outcome
- ▶ Jumpy planning for long-term credit assignment,
  - ▶ plan at different timescales

## Comparing parametric model and experience replay - IV

Computation:

- ▶ Querying a replay buffer is very cheap!
- ▶ Generating a sample from a learned model can be very expensive
- ▶ E.g. if the model is large neural network based generative model.

Memory:

- ▶ The memory requirements of a replay buffer scale linearly with its capacity
- ▶ A parametric model can achieve good accuracy with a fixed and comparably small memory footprint

# Planning for Action Selection

Matteo Hessel

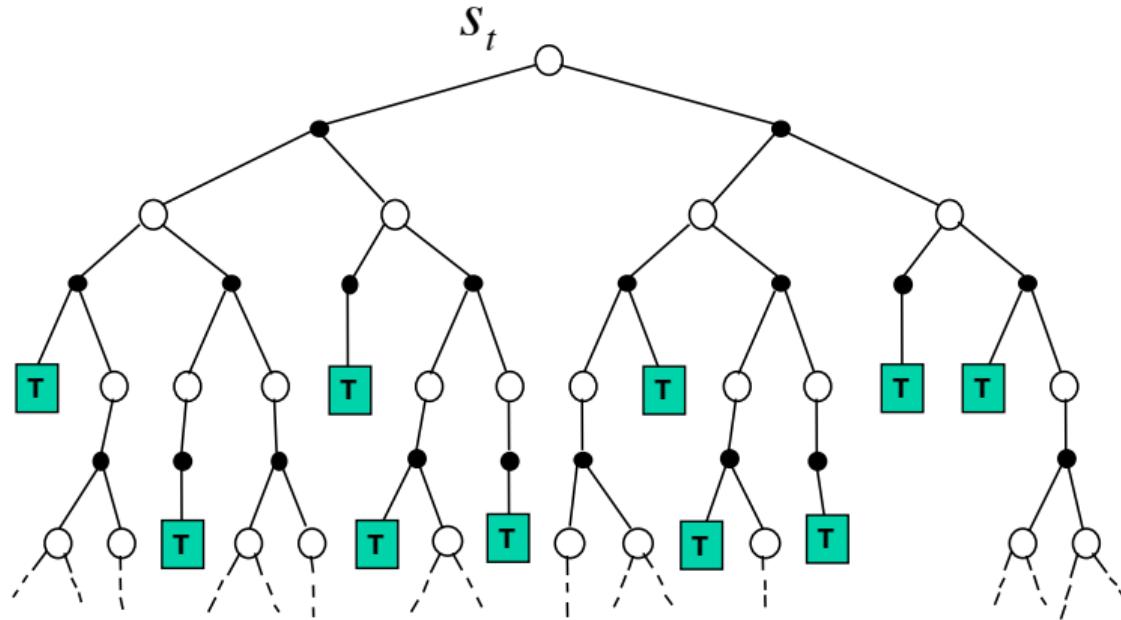
2021

## Planning for Action Selection

- ▶ We considered the case where planning is used to improve a global value function
- ▶ Now consider planning for the near future, to select the next action
- ▶ The distribution of states that may be encountered from **now** can differ from the distribution of states encountered from a starting state
- ▶ The agent may be able to make a more accurate local value function (for the states that will be encountered soon) than the global value function
- ▶ Inaccuracies in the model may result in interesting exploration rather than in bad updates.

## Forward Search

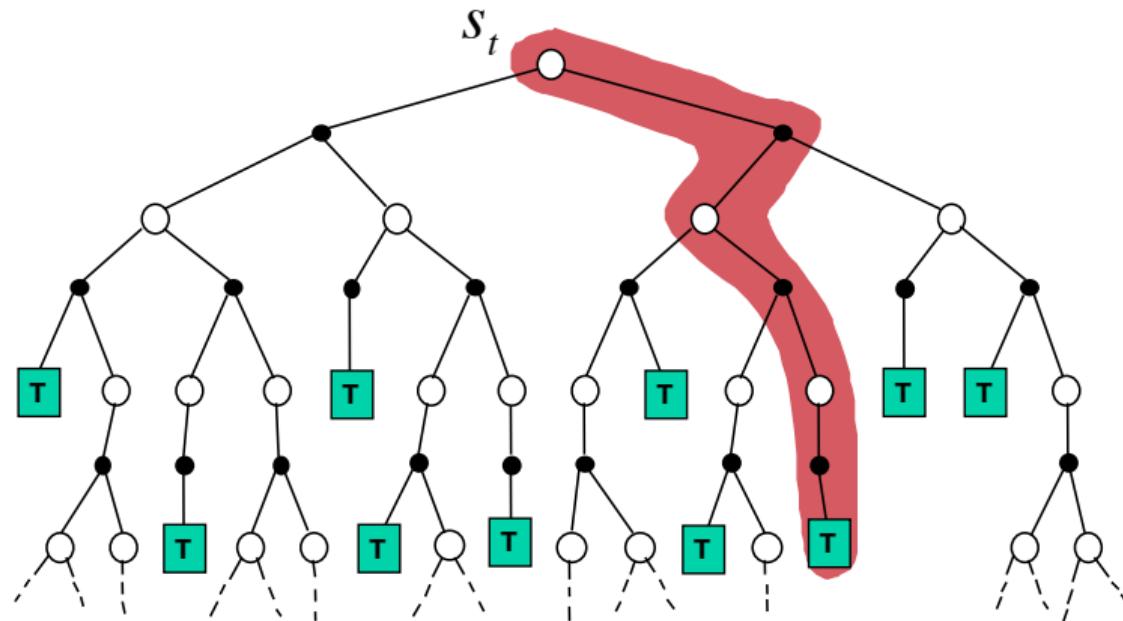
- ▶ Forward search algorithms select the best action by lookahead
- ▶ They build a search tree with the current state  $s_t$  at the root
- ▶ Using a model of the MDP to look ahead



- ▶ No need to solve whole MDP, just sub-MDP starting from now

# Simulation-Based Search

- ▶ Sample-based variant of **Forward search**
- ▶ **Simulate** episodes of experience from **now** with the model
- ▶ Apply **model-free** RL to simulated episodes



## Prediction via Monte-Carlo Simulation

- ▶ Given a parameterized model  $\mathcal{M}_\eta$  and a **simulation policy**  $\pi$
- ▶ Simulate  $K$  episodes from current state  $S_t$

$$\{S_t^k = \textcolor{red}{S_t}, A_t^k, R_{t+1}^k, S_{t+1}^k, \dots, S_T^k\}_{k=1}^K \sim \hat{p}_\eta, \pi$$

- ▶ Evaluate state by mean return (**Monte-Carlo evaluation**)

$$v(\textcolor{red}{S_t}) = \frac{1}{K} \sum_{k=1}^K G_t^k \rightsquigarrow v_\pi(S_t)$$

## Control via Monte-Carlo Simulation

- ▶ Given a model  $\mathcal{M}_\eta$  and a **simulation policy**  $\pi$
- ▶ For each action  $a \in \mathcal{A}$ 
  - ▶ Simulate  $K$  episodes from current (real) state  $s$

$$\{S_t^k = s, A_t^k = a, R_{t+1}^k, S_{t+1}^k, A_{t+1}^k, \dots, S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

- ▶ Evaluate actions by mean return (**Monte-Carlo evaluation**)

$$q(s, a) = \frac{1}{K} \sum_{k=1}^K G_t^k \rightsquigarrow q_\pi(s, a)$$

- ▶ Select current (real) action with maximum value

$$A_t = \operatorname{argmax}_{a \in \mathcal{A}} q(S_t, a)$$

# Monte-Carlo Tree Search - I

In MCTS, we incrementally build a search tree containing visited states and actions, Together with estimated action values  $q(s, a)$  for each of these pairs

- ▶ Repeat (for each simulated episode)
  - ▶ **Select** Until you reach a leaf node of the tree, pick actions according to  $q(s, a)$ .
  - ▶ **Expand** search tree by one node
  - ▶ **Rollout** until episode termination with a fixed simulation policy
  - ▶ **Update** action-values  $q(s, a)$  for all state-action pairs in the tree

$$q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^K \sum_{u=t}^T \mathbf{1}(S_u^k, A_u^k = s, a) G_u^k \rightsquigarrow q_\pi(s, a)$$

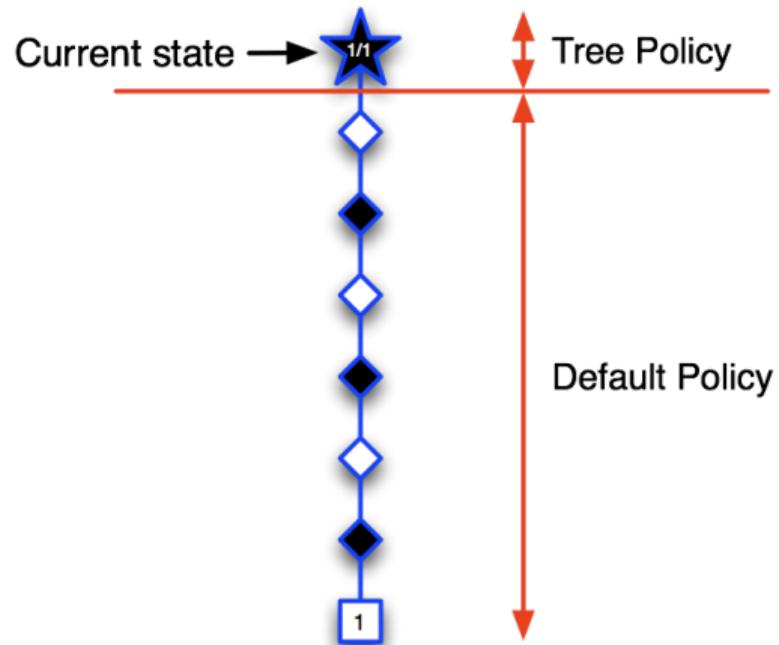
- ▶ Output best action according to  $q(s, a)$  in the root node when time runs out.

## Monte-Carlo Tree Search - II

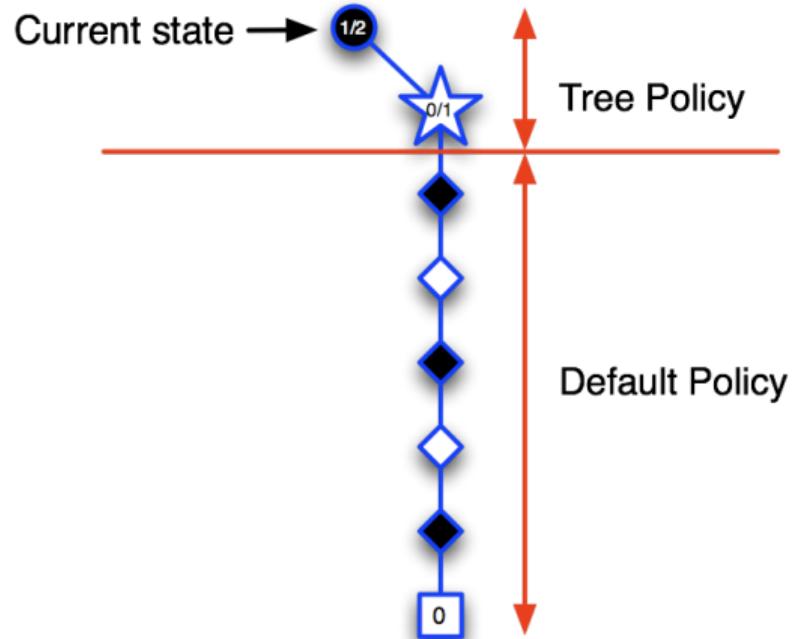
Note that we effectively have two simulation policies:

- ▶ a **Tree policy** that **improves** during search.
- ▶ a **Rollout policy** that is held fixed: often this may just be picking actions randomly.

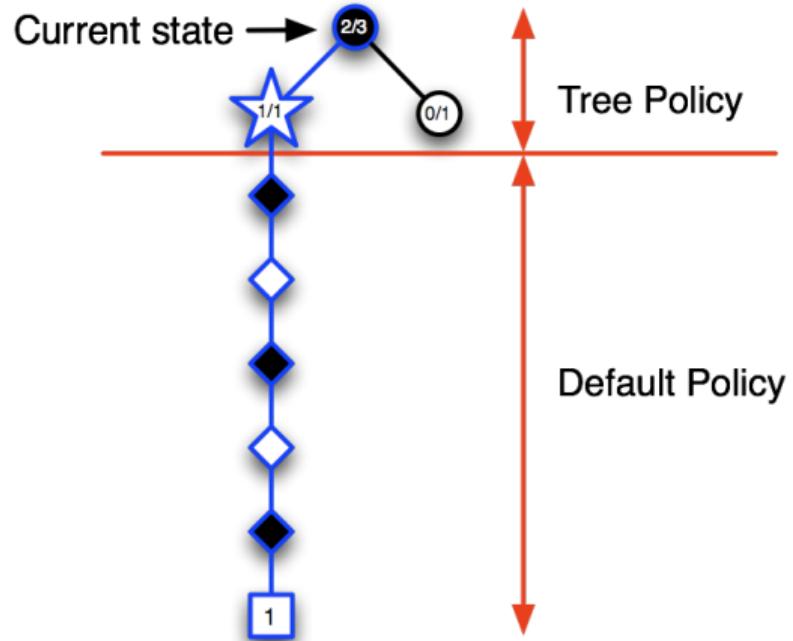
## Applying Monte-Carlo Tree Search (1)



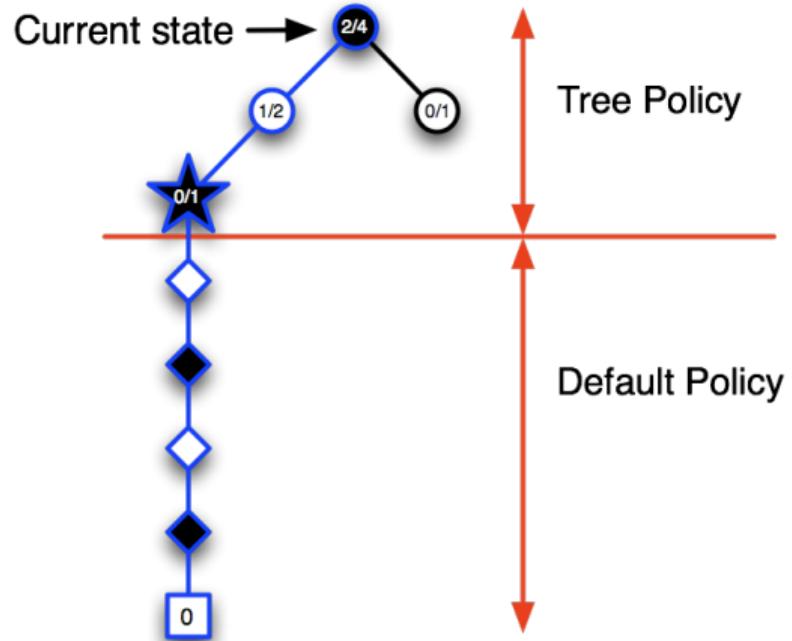
## Applying Monte-Carlo Tree Search (2)



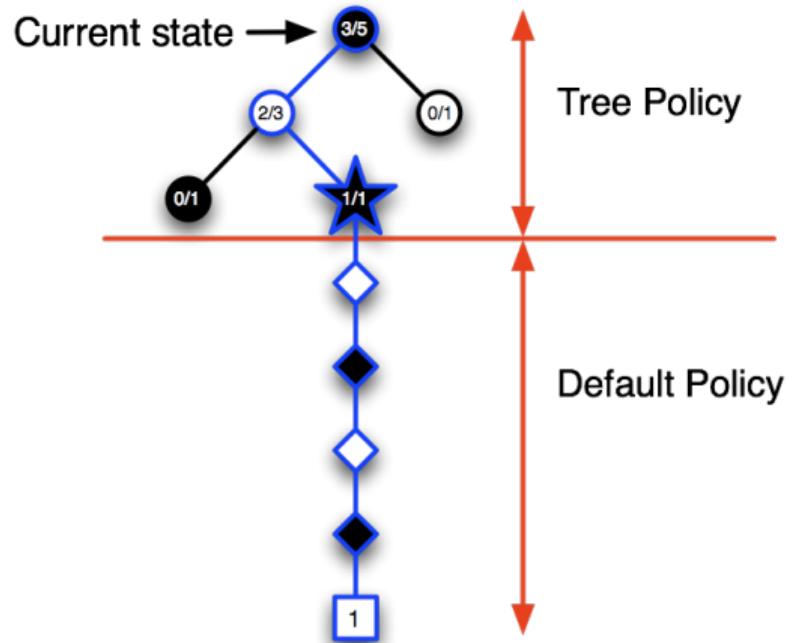
## Applying Monte-Carlo Tree Search (3)



## Applying Monte-Carlo Tree Search (4)



## Applying Monte-Carlo Tree Search (5)



## Advantages of Monte-Carlo Tree Search

- ▶ Highly selective best-first search
- ▶ Evaluates states **dynamically** (unlike e.g. DP)
- ▶ Uses sampling to break curse of dimensionality
- ▶ Works for “black-box” models (only requires samples)
- ▶ Computationally efficient, anytime, parallelisable

## Search tree and value function approximation - I

- ▶ Search tree is a table lookup approach
- ▶ Based on a **partial** instantiation of the table
- ▶ For model-free reinforcement learning, table lookup is naive
  - ▶ Can't store value for all states
  - ▶ Doesn't generalise between similar states
- ▶ For simulation-based search, table lookup is less naive
  - ▶ Search tree stores value for easily reachable states
  - ▶ But still doesn't generalise between similar states
  - ▶ In huge search spaces, value function approximation is helpful

# Lecture 9: Policy Gradients and Actor Critics

Hado van Hasselt

UCL, 2021



Background reading: Sutton & Barto, 2018, Chapter 13



*“Do no solve a more general problem as an intermediate step.”*

— Vladimir Vapnik, 1998

If we care about optimal behaviour: why not learn a policy directly?



# General overview

## ► Model-based RL

- + ‘Easy’ to learn a model (supervised learning)
- + Learns ‘all there is to know’ from the data
  - Uses compute & capacity on irrelevant details
  - Computing policy (=planning) is non-trivial and expensive (in compute)

## ► Value-based RL

- + Easy to generate policy (e.g.,  $\pi(a|s) = \mathcal{I}(a = \operatorname{argmax}_a q(s, a))$ )
- + Close to true objective
- + Fairly well-understood, good algorithms exist
- Still not the true objective:
  - May focus capacity on irrelevant details
  - Small value error can lead to larger policy error

## ► Policy-based RL

- + Right objective!
- o More pros and cons on later slide



# General overview

**Model-based RL**

**Value-based RL**

**Policy-based RL**

- ▶ All of these generalise in different ways
- ▶ Sometimes learning a model is easier (e.g., simple dynamics)
- ▶ Sometimes learning a policy is easier (e.g., “always move forward” is optimal)



# Policy-Based Reinforcement Learning

- ▶ Previously we approximated parametric value functions

$$\begin{aligned}v_w(s) &\approx v_\pi(s) \\q_w(s, a) &\approx q_\pi(s, a)\end{aligned}$$

- ▶ A policy can be generated from these values (e.g., greedy)
- ▶ In this lecture we directly parametrise the **policy** directly

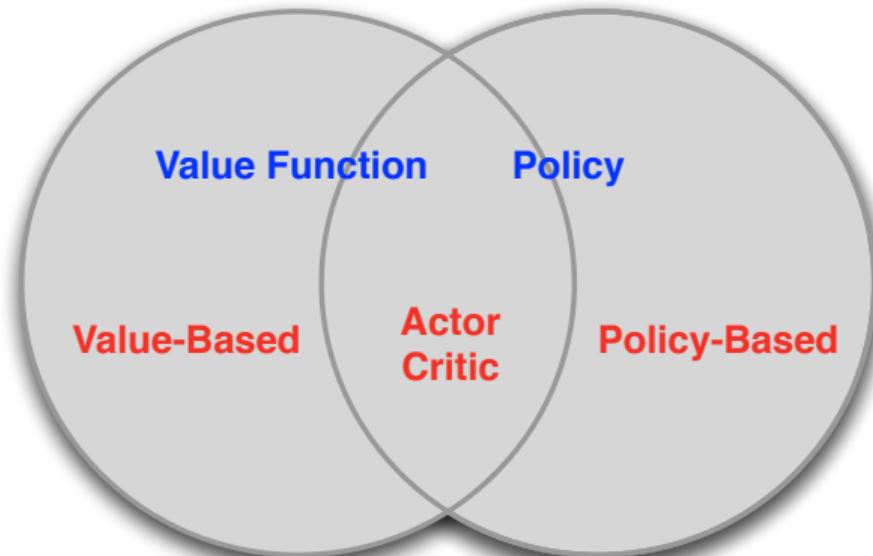
$$\pi_\theta(a|s) = p(a|s, \theta)$$

- ▶ This lecture, we focus on **model-free** reinforcement learning



# Value-based and policy-based RL: terminology

- ▶ **Value Based**
  - ▶ Learn values
  - ▶ Implicit policy (e.g.  $\epsilon$ -greedy)
- ▶ **Policy Based**
  - ▶ No values
  - ▶ Learn policy
- ▶ **Actor-Critic**
  - ▶ Learn values
  - ▶ Learn policy



# Advantages and disadvantages of policy-based RL

Advantages:

- ▶ True objective
- ▶ Easy extended to **high-dimensional** or **continuous** action spaces
- ▶ Can learn **stochastic** policies
- ▶ Sometimes policies are **simple** while values and models are complex
  - ▶ E.g., complicated dynamics, but optimal policy is always “move forward”

Disadvantages:

- ▶ Could get stuck in local optima
- ▶ Obtained knowledge can be **specific**, does not always generalise well
- ▶ Does not necessarily extract all useful information from the data  
(when used in isolation)



# Benefits of stochastic policies



# Stochastic policies

Why could we need stochastic policies?

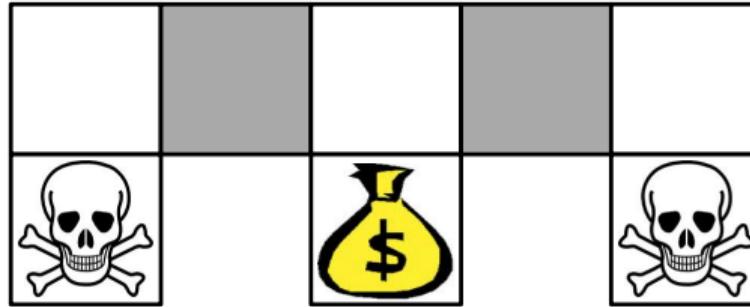
- ▶ In MDPs, there is always an optimal **deterministic** policy
- ▶ But, most problems are **not fully observable**
  - ▶ This is the common case, especially with function approximation
  - ▶ The optimal policy may then be stochastic
- ▶ Search space is smoother for stochastic policies  $\implies$  we can use gradients
- ▶ Provides some ‘exploration’ during learning



# Stochastic Policy Example: Aliased Grid World



## Example: Aliased Grid World



- ▶ The grey states look the same
- ▶ Consider features:

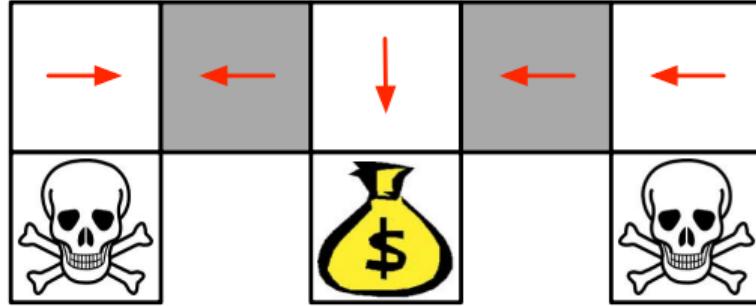
$$\phi(s) = \left( \underbrace{1}_{\text{up}} \quad \underbrace{0}_{\text{right}} \quad \underbrace{1}_{\text{down}} \quad \underbrace{0}_{\text{left}} \right)$$

walls=state

- ▶ Compare **deterministic** and **stochastic** policies



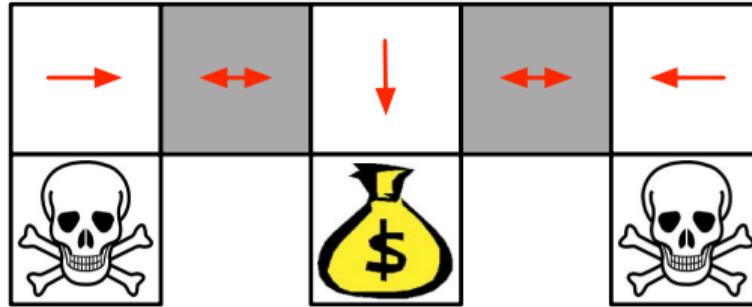
## Example: Aliased Gridworld



- ▶ Under aliasing, an optimal **deterministic** policy will either
  - ▶ move left in both grey states (shown by red arrows)
  - ▶ or move right in both grey states
- ▶ Either way, it can get stuck and never reach the money



## Example: Aliased Gridworld



- ▶ An optimal **stochastic** policy moves randomly left or right in grey states

$$\pi_{\theta}(\text{right} \mid \text{wall up and down}) = 0.5$$

$$\pi_{\theta}(\text{left} \mid \text{wall up and down}) = 0.5$$

- ▶ Will reach the goal state in a few steps with high probability
- ▶ Directly learning the policy parameters, we can learn an optimal stochastic policy
- ▶ Also when optimal policy does not give equal probability  
(So this differs from random tie-breaking with values.)



# Policy Learning Objective



# Policy Objective Functions

- ▶ Goal: given policy  $\pi_\theta(s, a)$ , find best parameters  $\theta$
- ▶ How do we measure the quality of a policy  $\pi_\theta$ ?
- ▶ In episodic environments we can use the average total return per episode
- ▶ In continuing environments we can use the average reward per step



## Policy Objective Functions: Episodic

- ▶ **Episodic-return objective:**

$$\begin{aligned} J_G(\boldsymbol{\theta}) &= \mathbb{E}_{S_0 \sim d_0, \pi_{\boldsymbol{\theta}}} \left[ \sum_{t=0}^{\infty} \gamma^t R_{t+1} \right] \\ &= \mathbb{E}_{S_0 \sim d_0, \pi_{\boldsymbol{\theta}}} [G_0] \\ &= \mathbb{E}_{S_0 \sim d_0} [\mathbb{E}_{\pi_{\boldsymbol{\theta}}} [G_t \mid S_t = S_0]] \\ &= \mathbb{E}_{S_0 \sim d_0} [v_{\pi_{\boldsymbol{\theta}}}(S_0)] \end{aligned}$$

where  $d_0$  is the start-state distribution This objective equals the expected value of the start state



# Policy Objective Functions: Average Reward

## ► Average-reward objective

$$\begin{aligned} J_R(\theta) &= \mathbb{E}_{\pi_\theta} [R_{t+1}] \\ &= \mathbb{E}_{S_t \sim d_{\pi_\theta}} [\mathbb{E}_{A_t \sim \pi_\theta(S_t)} [R_{t+1} \mid S_t]] \\ &= \sum_s d_{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \sum_r p(r \mid s, a)r \end{aligned}$$

where  $d_\pi(s) = p(S_t = s \mid \pi)$  is the probability of being in state  $s$  in the long run  
Think of it as the ratio of time spent in  $s$  under policy  $\pi$



# Policy Gradients



# Policy Optimisation

- ▶ Policy based reinforcement learning is an **optimization** problem
- ▶ Find  $\theta$  that maximises  $J(\theta)$
- ▶ We will focus on **stochastic gradient ascent**, which is often quite efficient (and easy to use with deep nets)
- ▶ Some approaches do not use gradient
  - ▶ Hill climbing / simulated annealing
  - ▶ Genetic algorithms / evolutionary strategies



# Policy Gradient

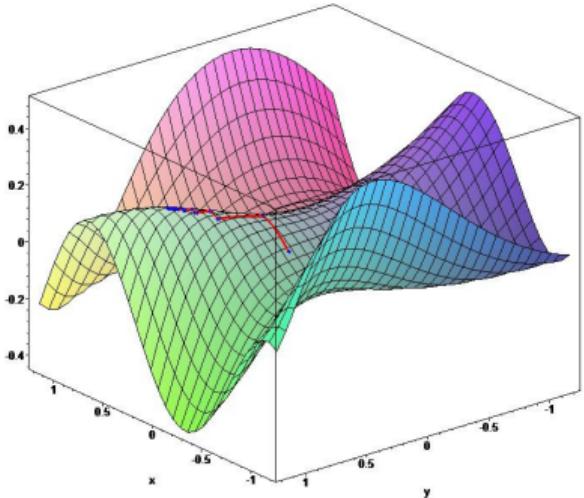
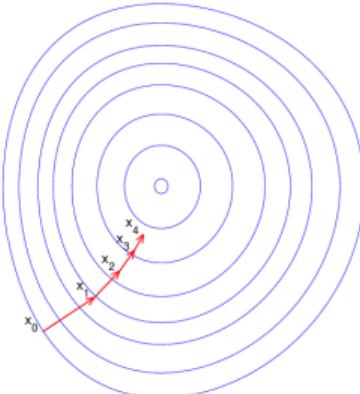
- ▶ Idea: ascent the gradient of the objective  $J(\theta)$

$$\Delta\theta = \alpha \nabla_{\theta} J(\theta)$$

- ▶ Where  $\nabla_{\theta} J(\theta)$  is the **policy gradient**

$$\nabla_{\theta} J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$$

- ▶ and  $\alpha$  is a step-size parameter
- ▶ Stochastic policies help ensure  $J(\theta)$  is smooth (typically/mostly)



# Gradients on parameterized policies

- ▶ How to compute this gradient  $\nabla_{\theta} J(\theta)$ ?
- ▶ Assume policy  $\pi_{\theta}$  is differentiable almost everywhere (e.g., neural net)
- ▶ For average reward

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} \mathbb{E}_{\pi_{\theta}}[R].$$

- ▶ How does  $\mathbb{E}[R]$  depend on  $\theta$ ?



# Contextual Bandits Policy Gradient

- ▶ Consider a one-step case (a contextual bandit) such that  $J(\theta) = \mathbb{E}_{\pi_\theta}[R(S, A)]$ .  
(Expectation is over  $d$  (states) and  $\pi$  (actions))  
(For now,  $d$  does **not** depend on  $\pi$ )
- ▶ We cannot sample  $R_{t+1}$  and then take a gradient:  
 $R_{t+1}$  is just a number and does not depend on  $\theta$ !
- ▶ Instead, we use the identity:

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[R(S, A)] = \mathbb{E}_{\pi_\theta}[R(S, A) \nabla_\theta \log \pi(A|S)] .$$

(Proof on next slide)

- ▶ The right-hand side gives an expected gradient that can be sampled
- ▶ Also known as REINFORCE (Williams, 1992)



## The score function trick

Let  $r_{sa} = \mathbb{E}[R(S, A) \mid S = s, A = s]$

$$\begin{aligned}\nabla_{\theta} \mathbb{E}_{\pi_{\theta}}[R(S, A)] &= \nabla_{\theta} \sum_s d(s) \sum_a \pi_{\theta}(a|s) r_{sa} \\&= \sum_s d(s) \sum_a r_{sa} \nabla_{\theta} \pi_{\theta}(a|s) \\&= \sum_s d(s) \sum_a r_{sa} \pi_{\theta}(a|s) \frac{\nabla_{\theta} \pi_{\theta}(a|s)}{\pi_{\theta}(a|s)} \\&= \sum_s d(s) \sum_a \pi_{\theta}(a|s) r_{sa} \nabla_{\theta} \log \pi_{\theta}(a|s) \\&= \mathbb{E}_{d, \pi_{\theta}}[R(S, A) \nabla_{\theta} \log \pi_{\theta}(A|S)]\end{aligned}$$



# Contextual Bandit Policy Gradient

$$\nabla_{\theta} \mathbb{E}[R(S, A)] = \mathbb{E}[\nabla_{\theta} \log \pi_{\theta}(A|S) R(S, A)]$$

(see previous slide)

- ▶ This is something we **can** sample
- ▶ Our stochastic policy-gradient update is then

$$\theta_{t+1} = \theta_t + \alpha R_{t+1} \nabla_{\theta} \log \pi_{\theta_t}(A_t|S_t).$$

- ▶ In expectation, this is the following the actual gradient
- ▶ So this is a pure (unbiased) stochastic gradient algorithm
- ▶ Intuition: increase probability for actions with high rewards



## Policy gradients: reduce variance

- ▶ Note that, in general

$$\begin{aligned}\mathbb{E}[b \nabla_{\theta} \log \pi(A_t | S_t)] &= \mathbb{E}\left[\sum_a \pi(a | S_t) b \nabla_{\theta} \log \pi(a | S_t)\right] \\ &= \mathbb{E}\left[b \nabla_{\theta} \sum_a \pi(a | S_t)\right] \\ &= \mathbb{E}[b \nabla_{\theta} 1] \quad = 0\end{aligned}$$

- ▶ This is true if  $b$  does not depend on the action (but it can depend on the state)
- ▶ Implies we can subtract a **baseline** to reduce variance

$$\theta_{t+1} = \theta_t + \alpha(R_{t+1} - b(S_t)) \nabla_{\theta} \log \pi_{\theta_t}(A_t | S_t).$$

- ▶ We will also use this fact in proofs below



## Example: Softmax Policy

- ▶ Consider a softmax policy on action preferences  $h(s, a)$  as an example
- ▶ Probability of action is proportional to exponentiated weight

$$\pi_{\theta}(a|s) = \frac{e^{h(s,a)}}{\sum_b e^{h(s,b)}}$$

- ▶ The gradient of the log probability is

$$\nabla_{\theta} \log \pi_{\theta}(A_t | S_t) = \underbrace{\nabla_{\theta} h(S_t, A_t)}_{\text{gradient of preference}} - \underbrace{\sum_a \pi_{\theta}(a|S_t) \nabla_{\theta} h(S_t, a)}_{\text{expected gradient of preference}}$$



# Policy Gradient Theorem



# Policy Gradient Theorem

- ▶ The policy gradient approach also applies to (multi-step) MDPs
- ▶ Replaces reward  $R$  with long-term return  $G_t$  or value  $q_\pi(s, a)$
- ▶ There are actually two policy gradient theorems (Sutton et al., 2000):

**average return per episode**      &      **average reward per step**



## Policy gradient theorem (episodic)

### Theorem

For any differentiable policy  $\pi_{\theta}(s, a)$ , let  $d_0$  be the starting distribution over states in which we begin an episode. Then, the policy gradient of  $J(\theta) = \mathbb{E}[G_0 \mid S_0 \sim d_0]$  is

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} \left[ \sum_{t=0}^T \gamma^t q_{\pi_{\theta}}(S_t, A_t) \nabla_{\theta} \log \pi_{\theta}(A_t | S_t) \mid S_0 \sim d_0 \right]$$

where

$$\begin{aligned} q_{\pi}(s, a) &= \mathbb{E}_{\pi}[G_t \mid S_t = s, A_t = a] \\ &= \mathbb{E}_{\pi}[R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \end{aligned}$$



## Policy gradients on trajectories

- ▶ Policy gradients do **not** need to know the MDP dynamics
- ▶ Kind of surprising; shouldn't we know how the policy influences the states?



## Episodic policy gradients: proof

- ▶ Consider trajectory  $\tau = S_0, A_0, R_1, S_1, A_1, R_1, S_2, \dots$  with return  $G(\tau)$

$$\nabla_{\theta} J_{\theta}(\pi) = \nabla_{\theta} \mathbb{E}[G(\tau)] = \mathbb{E}[G(\tau) \nabla_{\theta} \log p(\tau)] \quad (\text{score function trick})$$

$$\begin{aligned}\nabla_{\theta} \log p(\tau) &= \nabla_{\theta} \log \left[ p(S_0) \pi(A_0|S_0) p(S_1|S_0, A_0) \pi(A_1|S_1) \dots \right] \\ &= \nabla_{\theta} \left[ \log p(S_0) + \log \pi(A_0|S_0) + \log p(S_1|S_0, A_0) + \log \pi(A_1|S_1) + \dots \right] \\ &= \nabla_{\theta} \left[ \log \pi(A_0|S_0) + \log \pi(A_1|S_1) + \dots \right]\end{aligned}$$

So:

$$\nabla_{\theta} J_{\theta}(\pi) = \mathbb{E}_{\pi} [G(\tau) \nabla_{\theta} \sum_{t=0}^T \log \pi(A_t|S_t)]$$



## Episodic policy gradients: proof (continued)

$$\nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}(\pi) = \mathbb{E}_{\pi}[G(\tau) \sum_{t=0}^T \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)]$$

$$= \mathbb{E}_{\pi}[\sum_{t=0}^T G(\tau) \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)]$$

$$= \mathbb{E}_{\pi}[\sum_{t=0}^T \left( \sum_{k=0}^T \gamma^k R_{k+1} \right) \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)]$$

$$= \mathbb{E}_{\pi}[\sum_{t=0}^T \left( \sum_{k=t}^T \gamma^k R_{k+1} \right) \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)]$$

$$= \mathbb{E}_{\pi}[\sum_{t=0}^T \left( \gamma^t \sum_{k=t}^T \gamma^{k-t} R_{k+1} \right) \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)]$$

$$= \mathbb{E}_{\pi}[\sum_{t=0}^T (\gamma^t G_t) \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)]$$

$$= \mathbb{E}_{\pi}[\sum_{t=0}^T \gamma^t q_{\pi}(S_t, A_t) \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)]$$



## Episodic policy gradients algorithm

$$\nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}(\pi) = \mathbb{E}_{\pi} \left[ \sum_{t=0}^T \gamma^t q_{\pi}(S_t, A_t) \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t) \right]$$

- ▶ We can sample this, given a whole episode
- ▶ Typically, people pull out the sum, and split up this into separate gradients, e.g.,

$$\Delta \boldsymbol{\theta}_t = \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)$$

such that  $\mathbb{E}_{\pi} [\sum_t \Delta \boldsymbol{\theta}_t] = \nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}(\pi)$

- ▶ Typically, people ignore the  $\gamma^t$  term, use  $\Delta \boldsymbol{\theta}_t = G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t)$
- ▶ This is actually okay-ish — we just partially pretend on each step that we could have started an episode in that state instead  
(alternatively, view it as a slightly biased gradient)



# Policy gradient theorem (average reward)

## Theorem

For any differentiable policy  $\pi_\theta(s, a)$ , the policy gradient of  $J(\theta) = \mathbb{E}[R \mid \pi]$  is

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi [q_{\pi_\theta}(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t | S_t)]$$

where

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} - \rho + q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

$$\rho = \mathbb{E}_\pi[R_{t+1}] \quad (\text{Note: global average, not conditioned on state or action})$$

(Expectation is over both states and actions)



# Policy gradient theorem (average reward)

Alternatively (but equivalently):

## Theorem

For any differentiable policy  $\pi_\theta(s, a)$ , the policy gradient of  $J(\theta) = \mathbb{E}[R | \pi]$  is

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi [R_{t+1} \sum_{n=0}^{\infty} \nabla_\theta \log \pi_\theta(A_{t-n} | S_{t-n})]$$

(Expectation is over both states and actions)



# Actor Critics



## Policy gradients: reduce variance

- ▶ Recall  $\mathbb{E}_\pi[b(S_t)\nabla \log \pi(A_t|S_t)] = 0$ , for any  $b(S_t)$  that does not depend on  $A_t$
- ▶ A common baseline is  $v_\pi(S_t)$

$$\nabla_{\theta} J_{\theta}(\pi) = \mathbb{E} \left[ \sum_{t=0} \gamma^t (q_{\pi}(S_t, A_t) - v_{\pi}(S_t)) \nabla_{\theta} \log \pi(A_t | S_t) \right]$$

- ▶ Typically, we estimate  $v_w(s) \approx v_\pi(s)$  explicitly, and sample

$$q_{\pi}(S_t, A_t) \approx G_t$$

- ▶ We can minimise variance further by **bootstrapping**, e.g.,  $G_t = R_{t+1} + \gamma v_w(S_{t+1})$
- ▶ More on these techniques in the next lecture



## Critics

- ▶ A critic is a value function, learnt via **policy evaluation**:  
What is the value  $v_{\pi_\theta}$  of policy  $\pi_\theta$  for current parameters  $\theta$ ?
- ▶ This problem was explored in previous lectures, e.g.
  - ▶ Monte-Carlo policy evaluation
  - ▶ Temporal-Difference learning
  - ▶  $n$ -step TD



# Actor-Critic

Critic Update parameters  $w$  of  $v_w$  by TD (e.g., one-step) or MC

Actor Update  $\theta$  by policy gradient

**function** ONE-STEP ACTOR CRITIC

Initialise  $s, \theta$

**for**  $t = 0, 1, 2, \dots$  **do**

    Sample  $A_t \sim \pi_\theta(S_t)$

    Sample  $R_{t+1}$  and  $S_{t+1}$

$\delta_t = R_{t+1} + \gamma v_w(S_{t+1}) - v_w(S_t)$  [one-step TD-error, or **advantage**]

$w \leftarrow w + \beta \delta_t \nabla_w v_w(S_t)$  [TD(0)]

$\theta \leftarrow \theta + \alpha \delta_t \nabla_\theta \log \pi_\theta(A_t | S_t)$  [Policy gradient update (ignoring  $\gamma^t$  term)]



## Policy gradient variations

- ▶ Many extensions and variants exist
- ▶ Take care: bad policies lead to bad data
- ▶ This is different from supervised learning  
(where learning and data are independent)



## Increasing robustness with trust regions

- ▶ One way to increase stability is to **regularise**
- ▶ A popular method is to **limit the difference between subsequent policies**
- ▶ For instance, use the Kullbeck-Leibler divergence:

$$\text{KL}(\pi_{\text{old}} \| \pi_{\theta}) = \mathbb{E} \left[ \int \pi_{\text{old}}(a | S) \log \frac{\pi_{\theta}(a | S)}{\pi_{\text{old}}(a | S)} da \right].$$

(Expectation is over states)

- ▶ A divergence is like a distance between distributions
  - ▶ Then maximise  $J(\theta) - \eta \text{KL}(\pi_{\text{old}} \| \pi_{\theta})$ , for some hyperparameter  $\eta$
- c.f. **TRPO** (Schulman et al. 2015), **PPO** (Abbeel & Schulman 2016), **MPO** (Abdolmaleki et al. 2018)



# Continuous action spaces



# Continuous actions

- ▶ Pure value-based RL can be non-trivial to extend to **continuous action spaces**
  - ▶ How to approximate  $q(s, a)$ ?
  - ▶ How to compute  $\max_a q(s, a)$ ?
- ▶ When directly updating the policy parameters, continuous actions are easier
- ▶ Most algorithms discussed today can be used for discrete and continuous actions
- ▶ Note: exploration in high-dimensional continuous spaces can be challenging



## Example: Gaussian policy

- ▶ As example, consider a **Gaussian policy**
- ▶ E.g., mean is some function of state  $\mu_{\theta}(s)$
- ▶ For simplicity, let's consider fixed variance of  $\sigma^2$  (can be parametrized as well)
- ▶ Policy is Gaussian,  $A_t \sim \mathcal{N}(\mu_{\theta}(S_t), \sigma^2)$   
(here  $\mu_{\theta}$  is the mean — not to be confused with the behaviour policy!)
- ▶ The gradient of the log of the policy is then

$$\nabla_{\theta} \log \pi_{\theta}(s, a) = \frac{A_t - \mu_{\theta}(S_t)}{\sigma^2} \nabla \mu_{\theta}(s)$$

- ▶ This can be used, for instance, in REINFORCE / actor critic



## Example: Policy gradient with Gaussian policy

- ▶ Gaussian policy gradient update:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \beta(G_t - v(S_t))\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t | S_t) \\ &= \boldsymbol{\theta}_t + \beta(G_t - v(S_t)) \frac{A_t - \mu_{\boldsymbol{\theta}}(S_t)}{\sigma^2} \nabla \mu_{\boldsymbol{\theta}}(S_t)\end{aligned}$$

- ▶ Intuition: if return was high, move  $\mu_{\boldsymbol{\theta}}(S_t)$  toward  $A_t$



## Gradient ascent on value

- ▶ Policy gradients work well, but do not strongly exploit the critic
- ▶ If values generalise well, perhaps we can rely on them more?
  1. Estimate  $q_w \approx q_\pi$ , e.g., with Sarsa
  2. Define **deterministic actor**:  $A_t = \pi_\theta(S_t)$
  3. Improve actor (**policy improvement**) by **gradient ascent on the value**:

$$\Delta\theta \propto \frac{\partial Q_\pi(s, a)}{\partial \theta} = \frac{\partial Q_\pi(s, \pi_\theta(S_t))}{\partial \pi_\theta(S_t)} \frac{\partial \pi_\theta(S_t)}{\partial \theta}$$

- ▶ Known under various names:
  - “Action-dependent heuristic dynamic programming” (ADHDP; Werbos 1990, Prokhorov & Wunsch 1997)
  - “Gradient ascent on the value” (van Hasselt & Wiering 2007)
  - These days, mostly know as: “**Deterministic policy gradient**” (DPG; Silver et al. 2014)
- ▶ It’s a form of **policy iteration**



## Continuous actor-critic learning automaton (Cacla)

We can also define the error in action space, rather than parameter space

1.  $a_t = \text{Actor}_{\theta}(S_t)$  (get current (continuous) action proposal)
2.  $A_t \sim \pi(\cdot | S_t, a_t)$  (e.g.,  $A_t \sim \mathcal{N}(a_t, \Sigma)$ ) (explore)
3.  $\delta_t = R_{t+1} + \gamma v_w(S_{t+1}) - v_w(S_t)$  (compute TD error)
4. Update  $v_w(S_t)$  (e.g., using TD) (policy evaluation)
5. If  $\delta_t > 0$ , update  $\text{Actor}_{\theta}(S_t)$  towards  $A_t$  (policy improvement)

$$\theta_{t+1} \leftarrow \theta_t + \beta(A_t - a_t)\nabla_{\theta_t} \text{Actor}_{\theta_t}(S_t)$$

6. If  $\delta_t \leq 0$ , do not update  $\text{Actor}_{\theta}$

Note: update magnitude does not depend on the value magnitude

Note: don't update 'away' from 'bad' actions



# Video

(Peng, Berseth, van de Panne 2016)



# End of Lecture



# Lecture 10: Approximate Dynamic Programming

Diana Borsa

February 2020, UCL



# This Lecture

- ▶ Last lectures:
  - ▶ MDP, DP, Model-free Prediction, Model-free Control
  - ▶ Bellman equations and their corresponding operators.
  - ▶ RL under function approximation.
- ▶ This lecture:
  - ▶ Revisit the framework of Approximate Dynamic Programming.
  - ▶ Under the 2 sources of error (estimation + function approximation), what can we say about resulting estimates?
- ▶ Next lectures: (more) approximate versions of these paradigms, mainly in the absence of perfect knowledge of the environment + (deep) neural networks parametrisation.



# Preliminaries (Quick Recap)



## (Reminder) The Bellman Optimality Operator

### Definition (Bellman Optimality Operator $T_{\mathcal{V}}^*$ )

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , let  $\mathcal{V} \equiv \mathcal{V}_{\mathcal{S}}$  be the space of bounded real-valued functions over  $\mathcal{S}$ . We define, point-wise, the **Bellman Expectation operator**  $T_{\mathcal{V}}^* : \mathcal{V} \rightarrow \mathcal{V}$  as:

$$(T_{\mathcal{V}}^* f)(s) = \max_a \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) f(s') \right], \quad \forall f \in \mathcal{V} \quad (1)$$

As a common convention we drop the index  $\mathcal{V}$  and simply use  $T^* = T_{\mathcal{V}}^*$



## (Reminder) The Bellman Expectation Operator

### Definition (Bellman Expectation Operator)

Given an MDP,  $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ , let  $\mathcal{V} \equiv \mathcal{V}_{\mathcal{S}}$  be the space of bounded real-valued functions over  $\mathcal{S}$ . For any policy  $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ , we define, point-wise, the **Bellman Expectation operator**  $T_{\mathcal{V}}^{\pi} : \mathcal{V} \rightarrow \mathcal{V}$  as:

$$(T_{\mathcal{V}}^{\pi} f)(s) = \sum_a \pi(s, a) \left[ r(s, a) + \gamma \sum_{s'} p(s'|a, s) f(s') \right], \forall f \in \mathcal{V} \quad (2)$$



# (Reminder) Dynamic Programming with Bellman Operators

## Value Iteration

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = T^*v_k$ .

## Policy Iteration

- ▶ Start with  $\pi_0$ .
- ▶ Iterate:
  - ▶ Policy Evaluation:  $v_{\pi_i}$ 
    - ▶ (E.g. For instance, by iterating  $T^\pi$ :  $v_k = T^{\pi_i}v_{k-1} \Rightarrow v_k \rightarrow v^{\pi_i}$  as  $k \rightarrow \infty$ )
  - ▶ Greedy Improvement:  $\pi_{i+1} = \arg \max_a q_{\pi_i}(s, a)$



# Approximate DP

- ▶ More often than not:
  - ▶ We won't know the underlying MDP.  
⇒ sampling/estimation error, as we don't have access to the true operators  $T^\pi$  ( $T^*$ )
  - ▶ We won't be able to represent the value function exactly after each update.  
⇒ approximation error, as we approximate the true value functions within a (parametric) class (e.g. linear functions, neural nets, etc).
- ▶ Objective: Under the above conditions, come up with a policy  $\pi$  that is (close to) optimal.



# Approximate Value Iteration (+ friends)



## (Reminder) Value Iteration

### Value Iteration

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = T^*v_k$ .

As  $k \rightarrow \infty$ ,  $v_k \rightarrow_{\|\cdot\|_\infty} v^*$ . (Direct application for the Banach's Fixed Point theorem)



## Approximate Value Iteration

### Approximate Value Iteration

- ▶ Start with  $v_0$ .
- ▶ Update values:  $v_{k+1} = \mathcal{A}T^*v_k$ .  $(v_{k+1} \approx T^*v_k)$
- ▶ Return control policy:  $\pi_{k+1} = \text{Greedy}(v_{k+1})$

Question: As  $k \rightarrow \infty$ ,  $v_k \rightarrow_{\|\cdot\|_\infty} v^*$ ? Generally **X**. But maybe we don't need to!

**Good news:** interested in the **quality** of  $\pi_n$  after  $n$  iterations:  $v_{\pi_n}$  (or  $q_{\pi_n}$ )



## Approximate Value Iteration ( $q$ -value version)

### Approximate Value Iteration

- ▶ Start with  $q_0$ .
- ▶ Update values:  $q_{k+1} = \mathcal{A}T^*q_k$ .  $(q_{k+1} \approx T^*q_k)$
- ▶ Return control policy:  $\pi_{k+1} = \text{Greedy}(q_{k+1})$

Question: As  $k \rightarrow \infty$ ,  $q_k \rightarrow_{\|\cdot\|_\infty} q^*$ ? Generally **X**.



## Performance of AVI

Theorem (Bertsekas & Tsitsiklis, 1996)

Consider a MDP. And let  $q_k$  be the value function returned by AVI after  $k$  steps and let  $\pi_k$  be its corresponding greedy policy, then:

$$\|q^* - q_{\pi_n}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} \|T^* q_k - \mathcal{A} T^* q_k\|_\infty + \frac{2\gamma^{n+1}}{(1-\gamma)} \epsilon_0$$

where

$$\epsilon_0 = \|q^* - q_0\|_\infty$$

and  $T^*$  is the optimal Bellman operator associated with this MDP



## Performance of AVI

Theorem (Bertsekas & Tsitsiklis, 1996)

Consider a MDP. And let  $q_k$  be the value function returned by AVI after  $k$  steps and let  $\pi_k$  be its corresponding greedy policy, then:

$$\|q^* - q_{\pi_n}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} \|T^* q_k - \mathcal{A} T^* q_k\|_\infty + \frac{2\gamma^{n+1}}{(1-\gamma)} \underbrace{\epsilon_0}_{(\text{initial error})}$$

where

$$\epsilon_0 = \|q^* - q_0\|_\infty$$

and  $T^*$  is the optimal Bellman operator associated with this MDP



## Performance of AVI

Theorem (Bertsekas & Tsitsiklis, 1996)

Consider a MDP. And let  $q_k$  be the value function returned by AVI after  $k$  steps and let  $\pi_k$  be its corresponding greedy policy, then:

$$\|q^* - q_{\pi_n}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} \underbrace{\|T^*q_k - \mathcal{A}T^*q_k\|_\infty}_{\text{approximation error at iter. } k} + \frac{2\gamma^{n+1}}{(1-\gamma)} \underbrace{\epsilon_0}_{(\text{initial error})}$$

where

$$\epsilon_0 = \|q^* - q_0\|_\infty$$

and  $T^*$  is the optimal Bellman operator associated with this MDP



## Performance of AVI (Proof)

Statement:  $\|q^* - q_{\pi_n}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} \|T^* q_k - \mathcal{A} T^* q_k\|_\infty + \frac{2\gamma^{n+1}}{(1-\gamma)} \|q^* - q_0\|_\infty$

Proof.

Let's denote  $\epsilon = \max_{0 \leq k < n} \|T^* q_k - \mathcal{A} T^* q_k\|_\infty$ . Then for all  $k < n$ :

$$\|q^* - q_{k+1}\|_\infty \leq \|q^* - T^* q_k\|_\infty + \|T^* q_k - q_{k+1}\|_\infty \quad (3)$$

$$\leq \|T^* q^* - T^* q_k\|_\infty + \epsilon \quad (4)$$

$$\leq \gamma \|q^* - q_k\|_\infty + \epsilon \quad (5)$$

Thus:

$$\|q^* - q_k\|_\infty \leq \gamma \|q^* - q_{k-1}\|_\infty + \epsilon \quad (6)$$

$$\leq \gamma(\gamma \|q^* - q_{k-2}\|_\infty + \epsilon) + \epsilon \quad (7)$$

...

$$\leq \gamma^k \|q^* - q_0\|_\infty + \epsilon(1 + \gamma + \dots + \gamma^{K-1}) \quad (8)$$

$$\leq \gamma^k \|q^* - q_0\|_\infty + \frac{1}{(1-\gamma)} \epsilon \quad (9)$$



## Performance of AVI (Proof continued)

Statement:  $\|q^* - q_{\pi_n}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} \|T^* q_k - \mathcal{A} T^* q_k\|_\infty + \frac{2\gamma^{n+1}}{(1-\gamma)} \|q^* - q_0\|_\infty$

### Proof.

Let's denote  $\epsilon = \max_{0 \leq k < n} \|T^* q_k - \mathcal{A} T^* q_k\|_\infty$ . Then for all  $k$ , we have

$$\|q^* - q_k\|_\infty \leq \gamma^k \|q^* - q_0\|_\infty + \frac{1}{(1-\gamma)} \epsilon \quad (10)$$

Now recall, the performance of a greedy policy,  $\pi_k$  based on  $q_k$ :

$$\|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{1-\gamma} \|q^* - q_k\|_\infty \quad (11)$$

Combining the two results, we get the statement of the theorem. □



## Performance of AVI: Breakdown

Statement:

$$\|q^* - q_{\pi_n}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} \|T^* q_k - \mathcal{A} T^* q_k\|_\infty + \frac{2\gamma^{n+1}}{(1-\gamma)} \|q^* - q_0\|_\infty$$

---

Some implications:

- ▶ As  $n \rightarrow \infty$ ,  $\Rightarrow 2\gamma^n/(1-\gamma) \rightarrow 0$
- ▶ What if  $q_0 = q^*$ ?

$$\|q^* - q_{\pi_n}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} \|T^* q_k - \mathcal{A} T^* q_k\|_\infty$$

- ▶ Consider iteration 1:  $q_1 = \mathcal{A} T^* q_0 = \mathcal{A} q^*$ . In general  $\Rightarrow \|q_1 - q_0\|_\infty > 0$ .



## Performance of AVI: Breakdown

Statement:

$$\|q^* - q_{\pi_n}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} \|T^* q_k - \mathcal{A} T^* q_k\|_\infty + \frac{2\gamma^{n+1}}{(1-\gamma)} \|q^* - q_0\|_\infty \xrightarrow{0 \text{ as } n \rightarrow \infty}$$

- ▶ Consider a hypothesis space  $\mathcal{F}$ .
- ▶ What if  $\mathcal{A} = \Pi_\infty$  is the projection operator in  $L_\infty$ :

$$\Pi_\infty g := \arg \inf_{f \in \mathcal{F}} \|g - f\|_\infty$$

- ▶ We obtain:

$$q_{k+1} = \Pi_\infty T^* q_k = \arg \inf_{f \in \mathcal{F}} \|T^* q_k - f\|_\infty$$

- ▶ Note that  $\mathcal{A} T^* = \Pi_\infty T^*$  is a **contraction** operator in  $L_\infty$ .
- ▶ Algorithm converges for its fixed point:  $f = \Pi_\infty T^* f$
- ▶ If  $q^* \in \mathcal{F}$ , the above will converge to  $q^*$ .



Some concrete instances of AVI



# Fitted Q-iteration with Linear Approximation

Propose Algorithm:

$$q_{k+1} = \Pi_\infty T^* q_k = \arg \inf_{f \in \mathcal{F}} \|T^* q_k - f\|_\infty$$

---

- ▶ Consider a **linear** hypothesis space  $\mathcal{F}_\phi = \{q_w(s, a) = w^T \phi(s, a) | \forall w \in B\}$ .
- ▶ We obtain:

$$q_{k+1} = \arg \inf_{f \in \mathcal{F}_\phi} \|T^* q_k - f\|_\infty \tag{12}$$

$$\Leftrightarrow w_{k+1} = \arg \inf_{w \in B} \|T^*(w_k^T \phi) - w^T \phi\|_\infty \tag{13}$$

- ▶ Potential problems:
  - ▶ P1:  **$L_\infty$  minimisation** typically hard to carry out efficiently.
  - ▶ P2:  $T^*$  is typically **unknown** and will be approximated as well.



# Fitted Q-iteration with Linear Approximation

Proposals:

- ▶ **P1:**  $L_\infty \rightarrow L_2$ , wrt to a probability distribution  $\mu$  over  $\mathcal{S} \times \mathcal{A}$ .

$$q_{k+1} = \arg \inf_{f \in \mathcal{F}} \|T^* q_k - f\|_\mu^2.$$

- 
- ▶ **P2:** Sample to approximate  $T^*$ . (see previous lectures on Model-free control)
    - ▶ Sample  $(S_t, A_t, R_{t+1}, S_{t+1}) \sim \mu, P$
    - ▶ Approximate  $T^* q_k(S_t, A_t)$  by

$$Y_t = R_{t+1} + \gamma \max_a q_k(S_{t+1}, a) := \tilde{T}^* q_k$$

---

- ▶ Every iteration  $k$ :

$$q_{k+1} = \arg \min_{q_w \in \mathcal{F}} \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} (Y_t - q_w(S_t, A_t))^2$$



# Fitted Q-iteration with other Approximations

Algorithm:

- ▶ Every iteration  $k + 1$ :

$$q_{k+1} = \arg \min_{q_\theta \in \mathcal{F}} \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} (Y_t - q_\theta(S_t, A_t))^2 \quad (14)$$

$$= \arg \min_{q_\theta \in \mathcal{F}} \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} (\tilde{T}^* q_k(S_t, A_t) - q_\theta(S_t, A_t))^2 \quad (15)$$

- ▶  $\mathcal{F} = \mathcal{F}_\theta$  can be:

- ▶ Linear functions
- ▶ Neural networks
- ▶ Kernel functions
- ▶ ...



# Fitted Q-iteration (General recipe)

Algorithm:

- ▶ Every iteration  $k + 1$ :

$$q_{k+1} = \arg \min_{q_\theta \in \mathcal{F}} \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} \left( \tilde{T}^* q_k(S_t, A_t) - q_\theta(S_t, A_t) \right)^2$$

for samples  $(S_t, A_t, R_{t+1}, S_{t+1}) \sim \mu, P$ .

$\mathcal{F} = \mathcal{F}_\theta$  can be:

- ▶ Linear functions
- ▶ Neural networks
- ▶ Kernel functions
- ▶ ...

Samples:

- ▶ Online
- ▶ Fixed Dataset
- ▶ Replay Memory
- ▶ Generative Model

Targets:

- ▶  $\tilde{T}^* q_k = R_{t+1} + \gamma \max_a q_k(S_{t+1}, a)$
- ▶  $\tilde{T}^* q_{target} = \tilde{T}^* q_{\theta^-}$
- ▶ Off-policy updates (next lecture)
- ▶ Multi-step operators (next lecture)



# Fitted Q-iteration (General recipe: DQN)

Algorithm:

- ▶ Every iteration  $k + 1$ :

$$q_{k+1} = \arg \min_{q_\theta \in \mathcal{F}} \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} \left( \tilde{T}^* q_k(S_t, A_t) - q_\theta(S_t, A_t) \right)^2$$

$\mathcal{F} = \mathcal{F}_\theta$  can be:

- ▶ Linear functions
- ▶ Neural networks
- ▶ Kernel functions
- ▶ ...

Samples:

- ▶ Online
- ▶ Fixed Dataset
- ▶ Replay Memory
- ▶ Generative Model

Targets:

- ▶  $\tilde{T}^* q_k = R_{t+1} + \gamma \max_a q_k(S_{t+1}, a)$
- ▶  $\tilde{T}^* q_{target} = \tilde{T}^* q_{\theta-}$
- ▶ Off-policy updates (next lecture)
- ▶ Multi-step operators (next lecture)



## Fitted Q-iteration (General recipe: Batch RL - 1)

Algorithm:

- ▶ Every iteration  $k + 1$ :

$$q_{k+1} = \arg \min_{q_\theta \in \mathcal{F}} \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} \left( \tilde{T}^* q_k(S_t, A_t) - q_\theta(S_t, A_t) \right)^2$$

$\mathcal{F} = \mathcal{F}_\theta$  can be:

- ▶ Linear functions
- ▶ Neural networks
- ▶ Kernel functions
- ▶ ...

Samples:

- ▶ Online
- ▶ Fixed Dataset
- ▶ Replay Memory
- ▶ Generative Model

Targets:

- ▶  $\tilde{T}^* q_k = R_{t+1} + \gamma \max_a q_k(S_{t+1}, a)$
- ▶  $\tilde{T}^* q_{target} = \tilde{T}^* q_{\theta^-}$
- ▶ Off-policy updates (next lecture)
- ▶ Multi-step operators (next lecture)



## Fitted Q-iteration (General recipe: Batch RL - 2)

Algorithm:

- ▶ Every iteration  $k + 1$ :

$$q_{k+1} = \arg \min_{q_\theta \in \mathcal{F}} \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} \left( \tilde{T}^* q_k(S_t, A_t) - q_\theta(S_t, A_t) \right)^2$$

$\mathcal{F} = \mathcal{F}_\theta$  can be:

- ▶ Linear functions
- ▶ Neural networks
- ▶ Kernel functions
- ▶ ...

Samples:

- ▶ Online
- ▶ Fixed Dataset
- ▶ Replay Memory
- ▶ Generative Model

Targets:

- ▶  $\tilde{T}^* q_k = R_{t+1} + \gamma \max_a q_k(S_{t+1}, a)$
- ▶  $\tilde{T}^* q_{target} = \tilde{T}^* q_{\theta^-}$
- ▶ Off-policy updates (next lecture)
- ▶ Multi-step operators (next lecture)



# Fitted Q-iteration (General recipe: Dyna)

Algorithm:

- ▶ Every iteration  $k + 1$ :

$$q_{k+1} = \arg \min_{q_\theta \in \mathcal{F}} \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} \left( \tilde{T}^* q_k(S_t, A_t) - q_\theta(S_t, A_t) \right)^2$$

$\mathcal{F} = \mathcal{F}_\theta$  can be:

- ▶ Linear functions
- ▶ Neural networks
- ▶ Kernel functions
- ▶ ...

Samples:

- ▶ Online
- ▶ Fixed Dataset
- ▶ Replay Memory
- ▶ Generative Model

Targets:

- ▶  $\tilde{T}^* q_k = R_{t+1} + \gamma \max_a q_k(S_{t+1}, a)$
- ▶  $\tilde{T}^* q_{target} = \tilde{T}^* q_{\theta^-}$
- ▶ Off-policy updates (next lecture)
- ▶ Multi-step operators (next lecture)



# Approximate Policy Iteration



# (Reminder) Policy Iteration

## Policy Iteration

- ▶ Start with  $\pi_0$ .
- ▶ Iterate:
  - ▶ Policy Evaluation:  $q_i = q_{\pi_i}$
  - ▶ Greedy Improvement:  $\pi_{i+1} = \arg \max_a q_{\pi_i}(s, a)$

As  $i \rightarrow \infty$ ,  $q_i \rightarrow_{\|\cdot\|_\infty} q^*$ . Thus  $\pi_i \rightarrow \pi^*$ .



## (Reminder) Approximate Policy Iteration

### Approximate Policy Iteration

- ▶ Start with  $\pi_0$ .
- ▶ Iterate:
  - ▶ Policy Evaluation:  $q_i = \mathcal{A}q_{\pi_i}$   $(q_i \approx q_{\pi_i})$
  - ▶ Greedy Improvement:  $\pi_{i+1} = \arg \max_a q_i(s, a)$

Question 1: As  $i \rightarrow \infty$ , does  $q_i \rightarrow_{\|\cdot\|_\infty} q^*$ ?

Question 2: Or does  $\pi_i$  converge to the optimal policy?

In general, what is the **quality**,  $q_{\pi_i}$ , of the obtained policy  $\pi_i$ ?



# Performance of API

## Theorem (API Performance)

Consider a MDP. And let  $q_k$  and  $\pi_k$  be the value function and respectively evaluated (greedy) policy achieved by API at iteration  $k$ , then:

$$\lim \sup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \lim \sup_{k \rightarrow \infty} \|q_k - q_{\pi_k}\|_\infty$$



# Performance of API

## Theorem (API Performance)

Consider a MDP. And let  $q_k$  and  $\pi_k$  be the value function and respectively evaluated (greedy) policy achieved by API at iteration  $k$ , then:

$$\lim \sup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \lim \sup_{k \rightarrow \infty} \underbrace{\|q_{\pi_k} - q_k\|_\infty}_{\text{approximation error at iter. } k}$$



# Performance of API (Proof)

Notation:

- ▶ Matrix  $P$  (transition probabilities):  $n_a n_s \times n_s$

$$P((s, a), s') = Prob(s' | s, a)$$

- ▶ Matrix  $P^\pi$  (transition probabilities, given policy  $\pi$ ):  $n_a n_s \times n_a n_s$

$$P((s, a), s', a') = Prob(s', a' | s, a) = Prob(s' | s, a) \pi(a' | s')$$

- ▶ Note, that under this notation:

$$T^\pi q = R + \gamma P^\pi q$$

where  $R \in \mathbb{R}^{n_s n_a}$  is a vector enumerating all rewards  $r(s, a)$ .



# Performance of API (Proof)

Statement:  $\limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \|\underbrace{q_{\pi_k} - q_k}_{e_k}\|_\infty$

Proof.

Let's denote  $gain_k := q_{\pi_{k+1}} - q_{\pi_k}$ , for all iterations  $k$ .

$$\begin{aligned} gain_k &= q_{\pi_{k+1}} - q_{\pi_k} \\ &= T^{\pi_{k+1}} q_{\pi_{k+1}} - T^{\pi_k} q_{\pi_k} \end{aligned} \tag{16}$$

$$= T^{\pi_{k+1}} q_{\pi_{k+1}} - \textcolor{green}{T^{\pi_{k+1}} q_{\pi_k}} + \tag{17}$$

$$+ \textcolor{green}{T^{\pi_{k+1}} q_{\pi_k}} - \textcolor{blue}{T^{\pi_{k+1}} q_k} + \tag{18}$$

$$+ \textcolor{blue}{T^{\pi_{k+1}} q_k} - \textcolor{orange}{T^{\pi_k} q_k} + \tag{19}$$

$$+ \textcolor{orange}{T^{\pi_k} q_k} - T^{\pi_k} q_{\pi_k} \tag{20}$$



# Performance of API (Proof)

$$\text{Statement: } \limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \underbrace{\|q_{\pi_k} - q_k\|_\infty}_{e_k}$$

Proof.

Let's denote  $gain_k := q_{\pi_{k+1}} - q_{\pi_k}$ , for all iterations  $k$ .

$$\begin{aligned} gain_k &= q_{\pi_{k+1}} - q_{\pi_k} \\ &= T^{\pi_{k+1}} q_{\pi_{k+1}} - T^{\pi_{k+1}} q_{\pi_k} + \\ &\quad + T^{\pi_{k+1}} q_{\pi_k} - T^{\pi_{k+1}} q_k + \\ &\quad + T^{\pi_{k+1}} q_k - T^{\pi_k} q_k + \\ &\quad + T^{\pi_k} q_k - T^{\pi_k} q_{\pi_k} \end{aligned}$$



## Performance of API (Proof)

Statement:  $\limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \underbrace{\|q_{\pi_k} - q_k\|_\infty}_{e_k}$

Proof.

Let's denote  $gain_k := q_{\pi_{k+1}} - q_{\pi_k}$ , for all iterations  $k$ .

$$\begin{aligned} gain_k &= q_{\pi_{k+1}} - q_{\pi_k} \\ &= T^{\pi_{k+1}} q_{\pi_{k+1}} - T^{\pi_{k+1}} q_{\pi_k} + \\ &\quad + T^{\pi_{k+1}} q_{\pi_k} - T^{\pi_{k+1}} q_k + \\ &\quad + \cancel{T^{\pi_{k+1}} q_k} - \cancel{T^{\pi_k} q_k} + \\ &\quad + T^{\pi_k} q_k - T^{\pi_k} q_{\pi_k} \\ &= \gamma P^{\pi_{k+1}} (q_{\pi_{k+1}} - q_{\pi_k}) = \gamma P^{\pi_{k+1}} gain_k \\ &= \gamma P^{\pi_{k+1}} (q_{\pi_k} - q_k) = \gamma P^{\pi_{k+1}} e_k \\ &\geq 0 \\ &= \gamma P^{\pi_k} (q_k - q_{\pi_k}) = -\gamma P^{\pi_k} e_k \end{aligned}$$

Unpacking explicitly  $T^\pi q_k \leq T^{\pi_{k+1}} q_k, \forall \pi$

$$\begin{aligned} T^{\pi_{k+1}} q_k(s, a) &= r(s, a) + \gamma \sum_{a'} \pi_{k+1}(a'|s') q_k(s', a') \\ &= r(s, a) + \gamma \max_{a'} q_k(s', a') \quad (\text{as } \pi_{k+1} = \arg \max_{a'} q_k(s', a')) \end{aligned}$$



## Performance of API (Proof)

Statement:  $\limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \|\underbrace{q_{\pi_k} - q_k}_{e_k}\|_\infty$

### Proof.

Let's denote  $gain_k := q_{\pi_{k+1}} - q_{\pi_k}$ , for all iterations  $k$ .

$$\begin{aligned} gain_k &= q_{\pi_{k+1}} - q_{\pi_k} \\ &= T^{\pi_{k+1}} q_{\pi_{k+1}} - T^{\pi_{k+1}} q_{\pi_k} + && = \gamma P^{\pi_{k+1}} (q_{\pi_{k+1}} - q_{\pi_k}) = \gamma P^{\pi_{k+1}} gain_k \\ &\quad + T^{\pi_{k+1}} q_{\pi_k} - T^{\pi_{k+1}} q_k + && = \gamma P^{\pi_{k+1}} (q_{\pi_k} - q_k) = \gamma P^{\pi_{k+1}} e_k \\ &\quad + T^{\pi_{k+1}} q_k - T^{\pi_k} q_k + && \geq 0 \\ &\quad + T^{\pi_k} q_k - T^{\pi_k} q_{\pi_k} && = \gamma P^{\pi_k} (q_k - q_{\pi_k}) = -\gamma P^{\pi_k} e_k \\ &\geq \gamma P^{\pi_{k+1}} gain_k + \gamma (P^{\pi_{k+1}} - P^{\pi_k}) e_k \end{aligned}$$

Re-arranging, we get:

$$gain_k \geq \gamma(I - \gamma P^{\pi_{k+1}})^{-1}(P^{\pi_{k+1}} - P^{\pi_k})e_k$$



## Performance of API - Performance gain via Greedy step

Statement:

$$gain_k \geq \gamma(I - \gamma P^{\pi_{k+1}})^{-1}(P^{\pi_{k+1}} - P^{\pi_k})e_k$$

---

Some implications:

- ▶ What if  $e_k = 0$ ? (perfect evaluation at iter.  $k$ )

$$gain_k \geq 0$$

aka  $q_{\pi_{k+1}} \geq q_{\pi_k}$ .

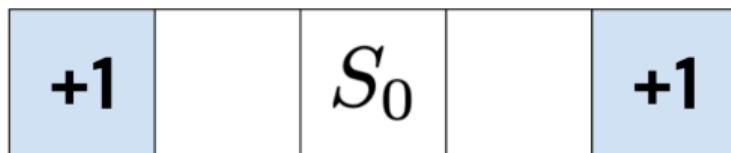
- ▶ Can  $gain_k < 0$ ?



## Performance of API - Performance gain via Greedy step

**Q:** Can  $gain_k := q_{\pi_{k+1}} - q_{\pi_k}$  be negative?

**Simple MDP**



$$\mathcal{A} = \{\leftarrow, \rightarrow\}$$



## Performance of API - Performance gain via Greedy step

- Q: Can  $gain_k := q_{\pi_{k+1}} - q_{\pi_k}$  be negative?

Deterministic policy  $\pi_k$ :

	$s_1$	$s_0$	$s_2$
$\pi_k(a s)$	→	→	→

Evaluation  $\pi_k$ :

$q_{\pi_k}(s, a)$	$s_1$	$s_0$	$s_2$
$a_1 = \rightarrow$	0.81	0.9	1.0
$a_2 = \leftarrow$	1.0	0.73	0.81

Consider an approx.  $q_k$ :

$q_k(s, a)$	$s_1$	$s_0$	$s_2$
$a_1 = \rightarrow$	0.8	0.83	0.85
$a_2 = \leftarrow$	1.1	0.75	0.87

Greedy policy  $\pi_{k+1}$ :

	$s_1$	$s_0$	$s_2$
$\pi_{k+1}(a s)$	←	→	←



# Performance of API - Performance gain via Greedy step

- Q: Can  $gain_k := q_{\pi_{k+1}} - q_{\pi_k}$  be negative?

Deterministic policy  $\pi_k$ :

	$s_1$	$s_0$	$s_2$
$\pi_k(a s)$	$\rightarrow$	$\rightarrow$	$\rightarrow$

Evaluation  $\pi_k$ :

$q_{\pi_k}(s, a)$	$s_1$	$s_0$	$s_2$
$a_1 = \rightarrow$	0.81	0.9	1.0
$a_2 = \leftarrow$	1.0	0.73	0.81

Greedy policy  $\pi_{k+1}$ :

	$s_1$	$s_0$	$s_2$
$\pi_k(a s)$	$\leftarrow$	$\rightarrow$	$\leftarrow$

Evaluation  $\pi_{k+1}$ :

$q_{\pi_{k+1}}(s, a)$	$s_1$	$s_0$	$s_2$
$a_1 = \rightarrow$	0.0	0.9	1.0
$a_2 = \leftarrow$	1.0	0.0	0.0



## Performance of API (Proof - continuing)

Statement:  $\limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \|\underbrace{q_{\pi_k} - q_k}_{e_k}\|_\infty$

Proof.

Let's denote  $L_k := q^* - q_{\pi_k}$ , for all iterations  $k$ . ("Loss in performance")

$$\begin{aligned} L_{k+1} &= q^* - q_{\pi_{k+1}} \\ &= T^{\pi^*} q_{\pi^*} - T^{\pi_{k+1}} q_{\pi_{k+1}} \end{aligned} \tag{21}$$

$$\begin{aligned} &= T^{\pi^*} q_{\pi^*} - T^{\pi^*} q_{\pi_k} + \end{aligned} \tag{22}$$

$$\begin{aligned} &\quad + T^{\pi^*} q_{\pi_k} - T^{\pi^*} q_k + \end{aligned} \tag{23}$$

$$\begin{aligned} &\quad + T^{\pi^*} q_k - T^{\pi_{k+1}} q_k + \end{aligned} \tag{24}$$

$$\begin{aligned} &\quad + T^{\pi_{k+1}} q_k - T^{\pi_{k+1}} q_{\pi_k} + \end{aligned} \tag{25}$$

$$\begin{aligned} &\quad + T^{\pi_{k+1}} q_{\pi_k} - T^{\pi_{k+1}} q_{\pi_{k+1}} \end{aligned} \tag{26}$$



## Performance of API (Proof - continuing)

Statement:  $\limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \|\underbrace{q_{\pi_k} - q_k}_{e_k}\|_\infty$

Proof.

Let's denote  $L_k := q^* - q_{\pi_k}$ , for all iterations  $k$ . ("Loss in performance")

$$\begin{aligned} L_{k+1} &= q^* - q_{\pi_{k+1}} \\ &= T^{\pi^*} q_{\pi^*} - T^{\pi^*} q_{\pi_k} + && = \gamma P^{\pi^*} (q_{\pi^*} - q_{\pi_k}) = \gamma P^{\pi^*} L_k \\ &\quad + T^{\pi^*} q_{\pi_k} - T^{\pi^*} q_k + && = \gamma P^{\pi^*} (q_{\pi_k} - q_k) = \gamma P^{\pi^*} e_k \\ &\quad + T^{\pi^*} q_k - T^{\pi_{k+1}} q_k + && \leq 0 \\ &\quad + T^{\pi_{k+1}} q_k - T^{\pi_{k+1}} q_{\pi_k} + && = \gamma P^{\pi_{k+1}} (q_k - q_{\pi_k}) = -\gamma P^{\pi_{k+1}} e_k \\ &\quad + T^{\pi_{k+1}} q_{\pi_k} - T^{\pi_{k+1}} q_{\pi_{k+1}} && = \gamma P^{\pi_{k+1}} (q_{\pi_k} - q_{\pi_{k+1}}) = -\gamma P^{\pi_{k+1}} g_k \\ &\leq \gamma P^{\pi^*} L_k + \gamma(P^{\pi^*} - P^{\pi_{k+1}}) e_k - \gamma P^{\pi_{k+1}} g_k \end{aligned}$$



## Performance of API (Proof - continuing)

Statement:  $\limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \|\underbrace{q_{\pi_k} - q_k}_{e_k}\|_\infty$

### Proof.

Thus we have:

$$L_{k+1} \leq \gamma P^{\pi^*} L_k + \gamma(P^{\pi^*} - P^{\pi_{k+1}})e_k - \gamma P^{\pi_{k+1}} g_k \quad (27)$$

$$\leq \gamma P^{\pi^*} L_k + \gamma(P^{\pi^*} - P^{\pi_{k+1}})e_k - \gamma P^{\pi_{k+1}} (\gamma(I - \gamma P^{\pi_{k+1}})^{-1}(P^{\pi_{k+1}} - P^{\pi_k})e_k) \quad (28)$$

$$\leq \gamma P^{\pi^*} L_k + \gamma \left( P^{\pi^*} + \gamma P^{\pi_{k+1}} (I - \gamma P^{\pi_{k+1}})^{-1}(P^{\pi_{k+1}} - P^{\pi_k}) - P^{\pi_{k+1}} \right) e_k \quad (29)$$

$$\leq \gamma P^{\pi^*} L_k + \gamma \left( P^{\pi^*} + P^{\pi_{k+1}} (I - \gamma P^{\pi_{k+1}})^{-1}(I - \gamma P^{\pi_k}) \right) e_k \quad (30)$$



## Performance of API (Proof - continuing)

Statement:  $\limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \|\underbrace{q_{\pi_k} - q_k}_{e_k}\|_\infty$

Proof.

Thus we have:

$$L_{k+1} \leq \gamma P^{\pi^*} L_k + \gamma(P^{\pi^*} - P^{\pi_{k+1}})e_k - \gamma P^{\pi_{k+1}}g_k \quad (31)$$

$$\leq \gamma P^{\pi^*} L_k + \gamma(P^{\pi^*} - P^{\pi_{k+1}})e_k - \gamma P^{\pi_{k+1}}(\gamma(I - \gamma P^{\pi_{k+1}})^{-1}(P^{\pi_{k+1}} - P^{\pi_k})e_k) \quad (32)$$

$$\leq \gamma P^{\pi^*} L_k + \gamma \left( P^{\pi^*} + \gamma P^{\pi_{k+1}}(I - \gamma P^{\pi_{k+1}})^{-1}(P^{\pi_{k+1}} - P^{\pi_k}) - P^{\pi_{k+1}} \right) e_k \quad (33)$$

$$\leq \gamma P^{\pi^*} L_k + \gamma \left( P^{\pi^*} + P^{\pi_{k+1}}(I - \gamma P^{\pi_{k+1}})^{-1}(I - \gamma P^{\pi_k}) \right) e_k \quad (34)$$

Asymptotic regime  $k \rightarrow \infty$ :

$$\limsup_{k \rightarrow \infty} L_k \leq \gamma(I - \gamma P^{\pi^*})^{-1} \limsup_{k \rightarrow \infty} \left( P^{\pi^*} + P^{\pi_{k+1}}(I - \gamma P^{\pi_{k+1}})^{-1}(I - \gamma P^{\pi_k}) \right) e_k$$



## Performance of API (Proof - continuing)

Statement:  $\limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \|\underbrace{q_{\pi_k} - q_k}_{e_k}\|_\infty$

### Proof.

Asymptotic regime  $k \rightarrow \infty$ :

$$\limsup_{k \rightarrow \infty} L_k \leq \gamma(I - \gamma P^{\pi^*})^{-1} \limsup_{k \rightarrow \infty} \left( P^{\pi^*} + P^{\pi_{k+1}}(I - \gamma P^{\pi_{k+1}})^{-1}(I - \gamma P^{\pi_k}) \right) e_k$$

Thus, taking the  $L_\infty$  norm:

$$\limsup_{k \rightarrow \infty} \|L_k\|_\infty \leq \frac{\gamma}{1-\gamma} \limsup_{k \rightarrow \infty} \left\| \left( P^{\pi^*} + P^{\pi_{k+1}}(I - \gamma P^{\pi_{k+1}})^{-1}(I - \gamma P^{\pi_k}) \right) \right\| \cdot \|e_k\|_\infty \quad (35)$$

$$\leq \frac{\gamma}{1-\gamma} \left( \frac{1+\gamma}{1-\gamma} + 1 \right) \cdot \limsup_{k \rightarrow \infty} \|e_k\|_\infty \quad (36)$$

Note: Here we used that  $\|P\|_\infty = 1$  for all (row-)stochastic matrices  $P$ .



A concrete instance



## (Reminder) TD( $\lambda$ ) with Linear Approximation

- ▶ Consider a **linear** hypothesis space  $\mathcal{F}_\phi = \{q_w(s, a) = w^T \phi(s, a) | \forall w \in \mathcal{B}\}$ .
- ▶ Temporal difference error:

$$\delta_t = R_{t+1} + \gamma q_{w_t}(S_{t+1}, \pi(S_{t+1})) - q_{w_t}(S_t, A_t) \quad (37)$$

- ▶ Parameters update:  $w_{t+1} = w_t + \alpha_t \delta_t \phi(s_t, a_t)$
- ▶ Properties:
  - ▶ This converges  $\lim_{t \rightarrow \infty} w_t = w^*$ , if  $\sum_t \alpha_t = \infty$  and  $\sum \alpha_t^2 < \infty$ . (Tsitsiklis et Van Roy'97).
  - ▶ Furthermore:

$$\|q_{w^*} - q_\pi\|_{2, \mu^\pi} \leq \frac{1 - \lambda\gamma}{1 - \gamma} \inf_w \|q_w - q_\pi\|_{2, \mu^\pi} \quad (38)$$



## TD( $\lambda$ ) with Linear Approximation

Statement:

$$\|q_{w^*} - q_\pi\|_{2,\mu^\pi} \leq \frac{1 - \lambda\gamma}{1 - \gamma} \inf_w \|q_w - q_\pi\|_{2,\mu^\pi}$$

---

Some implications:

- ▶ **Q:** For which  $\lambda$  is the RHS minimised (tightest bound)?
  - ▶ **A:**  $\lambda = 1$  (TD(1) = Monte Carlo).
- ▶ **Q:** What if  $q_\pi \in \mathcal{F}_\phi$ ?
  - ▶ **A :** RHS = 0. Thus  $q_{w^*} = q_\pi$ .
- ▶ **Q:** What if  $q_\pi \notin \mathcal{F}_\phi$ ?
  - ▶ **A :** RHS  $\neq 0$ . In general the FP  $q_{w^*} \neq \inf_w \|q_w - q_\pi\|_{2,\mu^\pi}$



# Summary



# AVI in general

Statement:

$$\|q^* - q_{\pi_n}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} \underbrace{\|T^* q_k - q_{k+1}\|_\infty}_{\epsilon_k} + \frac{2\gamma^{n+1}}{(1-\gamma)} \|q^* - q_0\|_\infty \xrightarrow{0 \text{ as } n \rightarrow \infty}$$

---

Some lessons:

- ▶ In general, convergence is **not guaranteed**. (In practice, fairly well behaved)
- ▶ Control the approximation errors  $\epsilon$ 
  - ▶ Two sources of error: **estimation(sampling) + approximation( $\mathcal{F}$ )**
  - ▶ For efficient optimisation:  $L_\infty \rightarrow L_{2,\mu}$
- ▶ Convergence point **is not always  $q^*$ !**
- ▶  $q^* \in \mathcal{F}$  is useful, but not enough!



## API in general

Statement:  $\limsup_{k \rightarrow \infty} \|q^* - q_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \limsup_{k \rightarrow \infty} \|\underbrace{q_{\pi_k} - q_k}_{e_k}\|_\infty$

---

Some lessons:

- ▶ In general, convergence is **not guaranteed**. (In practice, fairly well behaved)
- ▶ Control the approximation errors  $e_k$ 
  - ▶ Two sources of error: **estimation(sampling) + approximation( $\mathcal{F}$ )**
  - ▶ For efficient optimisation:  $L_\infty \rightarrow L_{2,\mu^{\pi_i}}$  (safe on-policy)
- ▶ Depending on the conditions/function class, we can obtain convergence:
  - ▶ Convergence point **is not always  $q^*$  or  $q_\pi$ !**
  - ▶ Convergence points might not be unique.
- ▶  $q^* \in \mathcal{F}$  is usually not enough!



## Questions?

*The only stupid question is the one you were afraid to ask but never did.*  
-Rich Sutton

For questions that may arise during this lecture please use Moodle and/or the next Q&A session.



# Lecture 11: Off-policy and multi-step learning

Hado van Hasselt

UCL, 2021



# Background

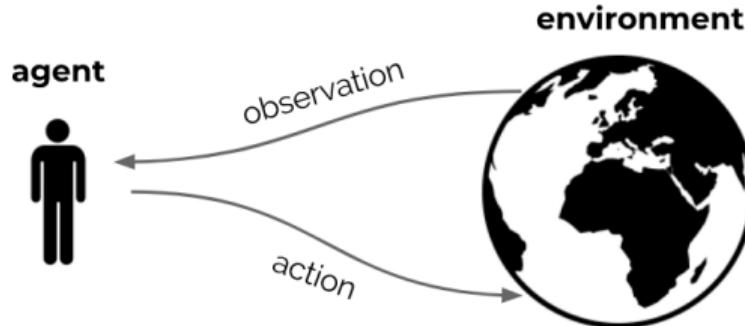
Sutton & Barto 2018, Chapter 5, 7, 11



# Recap



# Recap



- ▶ Reinforcement learning is the science of learning to make decisions
- ▶ Agents can learn a **policy**, **value function** and/or a **model**
- ▶ The general problem involves taking into account **time** and **consequences**
- ▶ Decisions affect the **reward**, the **agent state**, and **environment state**



# High level

- ▶ Previous lectures:
  - ▶ Model-free prediction & control
  - ▶ Multi-step updates (and eligibility traces)
  - ▶ Understanding dynamic programming operators
  - ▶ Predictions with function approximation
  - ▶ Model-based algorithms
  - ▶ Policy gradients and actor-critic algorithms
- ▶ This lecture:
  - ▶ **Off-policy learning**, especially when combined with **multi-step** updates and **function approximation**



# Motivation



# Why learn off-policy?

## Why learn off-policy?

- ▶ Off-policy learning is important to learn about **hypothetical, counterfactual** events (i.e, “what if” question)
- ▶ Use cases include
  - ▶ learning about the greedy policy
  - ▶ learning about (many) other policies
  - ▶ learning from observed data (e.g., stored logs / other agents)
  - ▶ learning from past policies
- ▶ This is also important to correct for **mismatch in data distributions**, for instance for policy gradients



## One-step off-policy

With action values, **one-step** off-policy learning seems relatively straightforward:

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha_t(R_{t+1} + \sum_a \pi(a|S_{t+1})q(S_{t+1}, a) - q(S_t, A_t))$$

For instance

- ▶ Q-learning: let  $\pi$  be greedy  $\implies \sum_a \pi_{sa} q_{sa} = \max_a q_{sa}$
- ▶ Expected Sarsa: let  $\pi$  be the current behaviour policy
- ▶ Sarsa: let  $\pi$  put all probability mass on the action the behaviour picked



## Multi-step off-policy

For **multi-step updates**, we can use **importance-sampling corrections**

E.g., for a Monte Carlo return on a trajectory  $\tau_t = \{S_t, A_t, R_{t+1}, \dots, S_T\}$

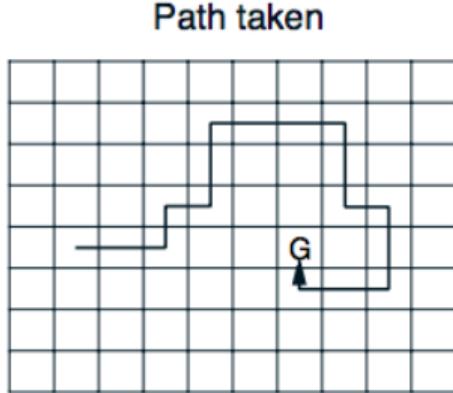
$$\hat{G}_t \equiv \frac{p(\tau_t|\pi)}{p(\tau_t|\mu)} G_t = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \dots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)} G_t,$$

then  $\mathbb{E}[\hat{G}_t \mid \mu] = \mathbb{E}[G_t \mid \pi]$

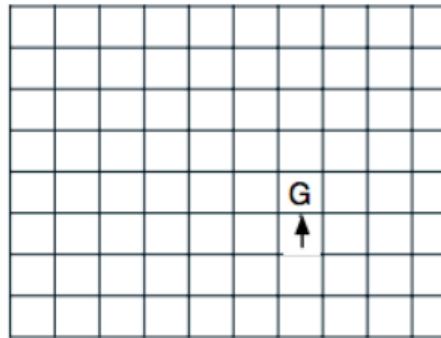


## Multi-step off-policy

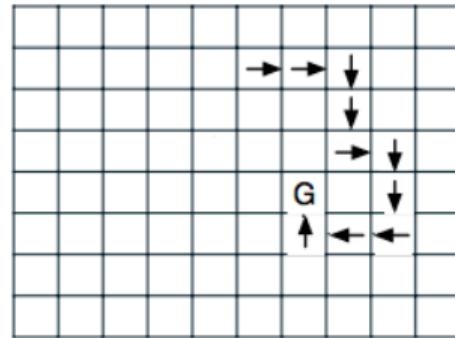
- ▶ We know multi-step updates often more efficiently propagate information
- ▶ Neither full Monte Carlo, nor one-step bootstrapping, is typically the best trade-off



Action values increased  
by one-step Sarsa



Action values increased  
by 10-step Sarsa



## Off-policy corrections for policy gradients

- ▶ Recall: for policy gradient methods we want to sample/estimate

$$\mathbb{E}[q_\pi(S_t, A_t) \nabla \log \pi(A_t | S_t)] .$$

- ▶ On-policy can sample multi-step returns  $G_t$  such that  $\mathbb{E}[G_t | \pi] \approx q_\pi(s, a)$
- ▶ But what if the behaviour is  $\mu \neq \pi$ ?
- ▶  $\implies$  we might not be following a gradient direction



# Issues in off-policy learning



## Issues with off-policy learning

The following issues (especially) arise when learning off-policy

- ▶ High variance (especially when using multi-step updates)
- ▶ Divergent and inefficient learning (especially when using one-step updates)

We will discuss both in this lecture



# Issues in off-policy learning: Variance



## Variance of importance sampling corrections

- ▶ A big issue in using importance-sampling corrections is **high variance**
- ▶ First, consider a one-step reward
- ▶ Verify the expectation, for a given state  $s$ :

$$\begin{aligned}\mathbb{E} \left[ \frac{\pi(A_t|s)}{\mu(A_t|s)} R_{t+1} \mid A_t \sim \mu \right] &= \sum_a \mu(a|s) \frac{\pi(a|s)}{\mu(a|s)} r(s, a) \\ &= \sum_a \pi(a|s) r(s, a) \\ &= \mathbb{E}[R_{t+1} \mid A_t \sim \pi]\end{aligned}$$

- ▶ But typically the variance will be larger, sometimes greatly so

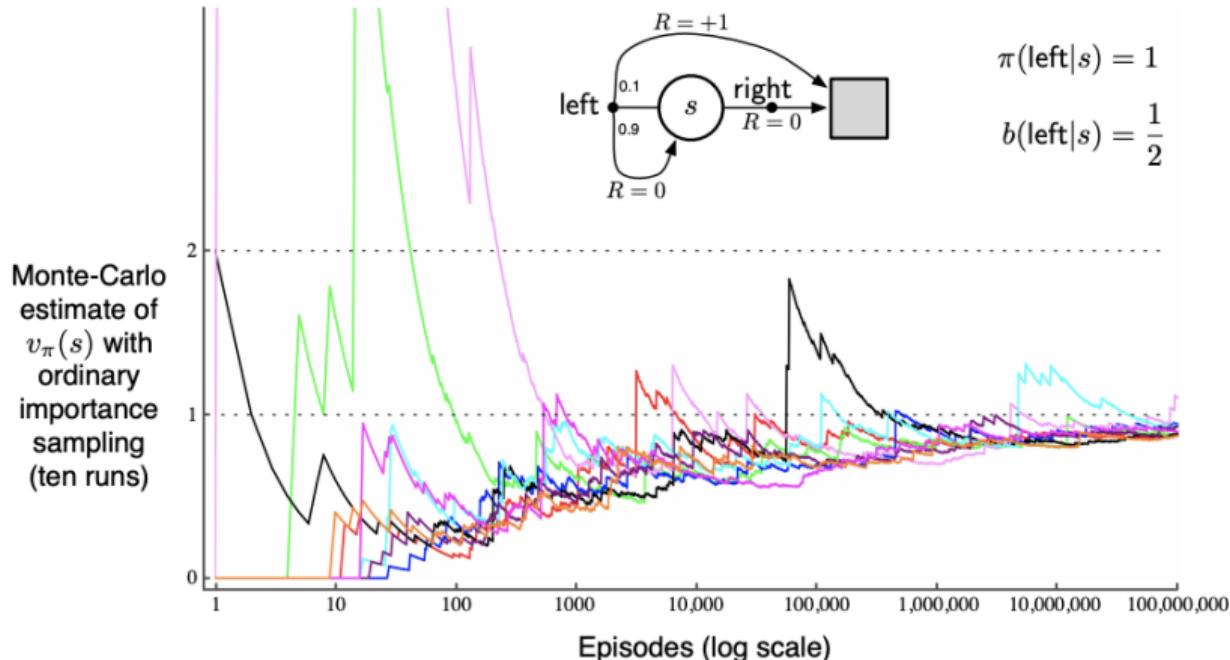


# Variance example



# Variance of importance sampling corrections

- In some cases the variance of an importance-weighting return can even be **infinite** (see: Sutton & Barto, Example 5.5)



# Mitigating variance

- ▶ There are multiple ways to reduce variance
- ▶ We will discuss three:
  - ▶ Per-decision importance weighting
  - ▶ Control variates
  - ▶ (Adaptive) bootstrapping



Reducing variance:  
Per-decision importance weighting



## Mitigating variance: with per-decision importance weighting

Consider some state  $s$ . For any random  $X$  that does not correlate with (random) action  $A$  we have

$$\mathbb{E}[X \mid \pi] = \mathbb{E} \left[ \frac{\pi(A|s)}{\mu(A|s)} X \mid \mu \right] = \mathbb{E}[X \mid \mu]$$

Intuition: the expectation does not depend on the policy, so we don't need to correct



## Mitigating variance: with per-decision importance weighting

Proof:

$$\begin{aligned} & \mathbb{E} \left[ \frac{\pi(A|s)}{\mu(A|s)} X \mid \mu \right] \\ &= \mathbb{E}[X \mid \mu] \mathbb{E} \left[ \frac{\pi(A|s)}{\mu(A|s)} \mid \mu \right] && \text{(Because } X \text{ and } \frac{\pi}{\mu} \text{ are uncorrelated)} \\ &= \mathbb{E}[X \mid \mu] \sum_a \mu(a|s) \frac{\pi(a|s)}{\mu(a|s)} \\ &= \mathbb{E}[X \mid \mu] \sum_a \pi(a|s) \\ &= \mathbb{E}[X \mid \mu] && \text{(Because } \sum_a \pi(a|s) = 1 \text{)} \end{aligned}$$

Similarly, in general, we have  $\mathbb{E}\left[\frac{\pi(A|s)}{\mu(A|s)} \mid \mu\right] = 1$



## Notation

Shorthand notations:

$$\rho_t \equiv \frac{\pi(A_t | S_t)}{\mu(A_t | S_t)}$$

$$\rho_{t:t+n} \equiv \prod_{k=t}^{t+n} \rho_k = \prod_{k=t}^{t+n} \frac{\pi(A_k | S_k)}{\mu(A_k | S_k)}$$

Then the reweighted MC return from state  $S_t$  terminating at time  $T$  can be written as

$$\underbrace{\left( \prod_{k=t}^{T-1} \frac{\pi(A_k | S_k)}{\mu(A_k | S_k)} \right)}_{= \rho_{t:T-1}} \underbrace{\left( \sum_{k=t}^{T-1} \gamma^{k-t} R_{k+1} \right)}_{= G_t} = \rho_{t:T-1} G_t = \sum_{k=t}^{T-1} \rho_{t:T-1} \gamma^{k-t} R_{k+1}$$

We can interpret the importance-weight  $\rho_{t:T-1}$  as applying to each reward



## Mitigating variance: with per-decision importance weighting

$$\rho_{t:T-1} G_t = \sum_{k=t}^{T-1} \rho_{t:T-1} \gamma^{k-t} R_{k+1}$$

**Earlier** rewards cannot depend on **later** actions. This means:

$$\begin{aligned}\mathbb{E}[\rho_{t:T-1} G_t \mid \mu] &= \mathbb{E}\left[\sum_{k=t}^{T-1} \rho_{t:T-1} \gamma^{k-t} R_{k+1} \mid \mu\right] \\ &= \mathbb{E}\left[\sum_{k=t}^{T-1} \rho_{t:k} \gamma^{k-t} R_{k+1} \mid \mu\right]\end{aligned}$$

Recursive definition of the latter:

$$G_t^\rho = \rho_t (R_{t+1} + \gamma G_{t+1}^\rho)$$



## Mitigating variance: with per-decision importance weighting

- ▶ Per-decision importance-weighted return

$$G_t^\rho = \rho_t(R_{t+1} + \gamma G_{t+1}^\rho)$$

We can use this to learn  $v_\pi$  from data generated under  $\mu \neq \pi$

- ▶ To learn **action values**  $q_\pi$ , we can use

$$G_t^\rho = R_{t+1} + \gamma \rho_{t+1} G_{t+1}^\rho$$

- ▶ How and why are these different?



# Reducing variance: Control variates



# Example: control variates



## Control variates for multi-step returns

The idea of control variates can be extended to multi-step returns

- ▶ First, recall

$$\begin{aligned}\delta_t^\lambda &\equiv G_t^\lambda - v(S_t) \\&= R_{t+1} + \gamma((1-\lambda)v(S_{t+1}) + \gamma\lambda G_{t+1}^\lambda) - v(S_t) \\&= \underbrace{R_{t+1} + \gamma v(S_{t+1}) - v(S_t)}_{= \delta_t} + \underbrace{\gamma\lambda(G_{t+1}^\lambda - v(S_{t+1}))}_{= \delta_{t+1}^\lambda} \\&= \delta_t + \gamma\lambda\delta_{t+1}^\lambda\end{aligned}$$



## Control variates for multi-step returns

The idea of control variates can be extended to multi-step returns

- ▶ Now, let's add per-decision importance weights

$$\delta_t^\lambda = \delta_t + \gamma \lambda \delta_{t+1}^\lambda$$

$$\delta_t^{\rho\lambda} = \rho_t (\delta_t + \gamma \lambda \delta_{t+1}^{\rho\lambda})$$

- ▶ By design this includes the  $(1 - \rho_t)v(S_t)$  control variate terms
- ▶ Sometimes called 'error weighting' (to contrast to 'reward weighting')



## Control variates for multi-step returns

$$\delta_t^{\rho\lambda} = \rho_t(\delta_t + \gamma\lambda\delta_{t+1}^{\rho\lambda})$$

- One can show that

$$\mathbb{E}[\delta_t^{\rho\lambda} \mid \mu] = \mathbb{E}[G_t^{\rho\lambda} - v(S_t) \mid \mu]$$

where

$$G_t^{\rho\lambda} = \rho_t \left( R_{t+1} + \gamma \left( (1 - \lambda)v(S_{t+1}) + \lambda G_{t+1}^{\rho\lambda} \right) \right)$$

is the per-decision importance-weighted  $\lambda$ -return.

- But  $\delta_t^{\rho\lambda}$  can have lower variance than  $G_t^{\rho\lambda} - v(S_t)$



# Reducing variance: Adaptive Bootstrapping



## Reducing variance: bootstrapping

- ▶ For our last technique, we consider bootstrapping
- ▶ This amounts to picking  $\lambda < 1$  when using either  $\delta_t^{\rho\lambda}$  or  $G_t^{\rho\lambda}$
- ▶ Note that to learn **action values**, we can use

$$G_t^{\rho\lambda} = R_{t+1} + \gamma \left( (1 - \lambda) \sum_a \pi(a \mid S_{t+1}) q(S_{t+1}, a) + \rho_{t+1} \lambda G_{t+1}^{\rho\lambda} \right)$$

Then, if  $\lambda = 0$ , we get

$$G_t = R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) q(S_{t+1}, a)$$

⇒ no more importance weighted ⇒ low variance

- ▶ However, bootstrapping too much may open us to the **deadly triad!**

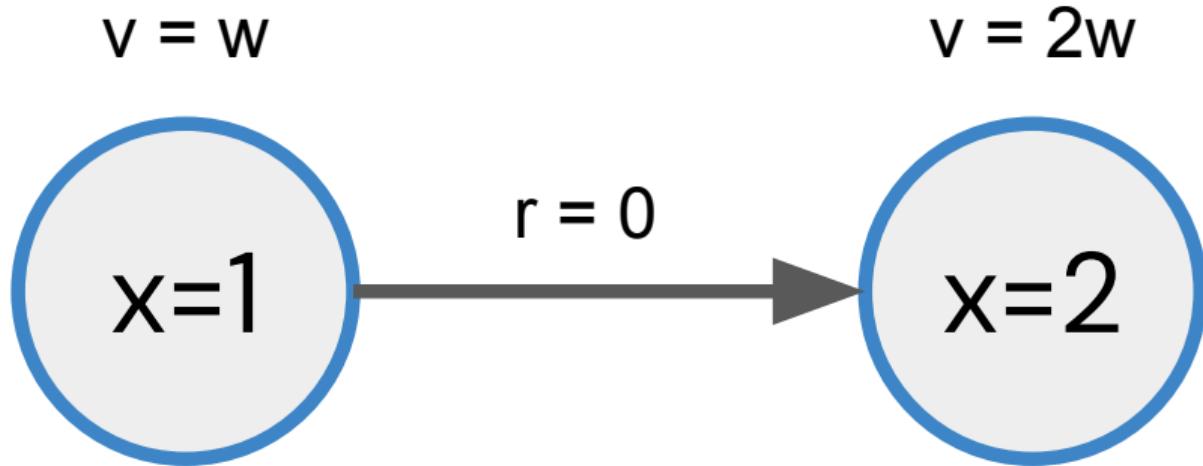


## Recap: Deadly triad

- ▶ Recall, the deadly triad refers to the possibility of divergence when we combine
  - ▶ **Bootstrapping**
  - ▶ **Function approximation**
  - ▶ **Off-policy learning**



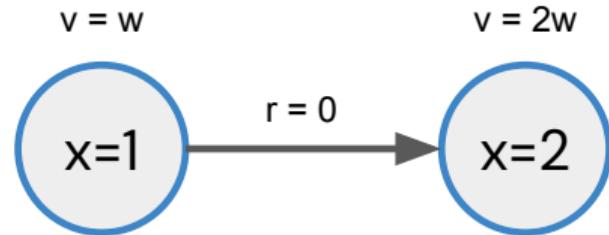
## Recap: Deadly triad



What if we use TD only on this transition?



## Recap: Deadly triad



$$\begin{aligned}w_{t+1} &= w_t + \alpha_t(r + \gamma v(s') - v(s))\nabla v(s) \\&= w_t + \alpha_t(2\gamma - 1)w_t\end{aligned}$$

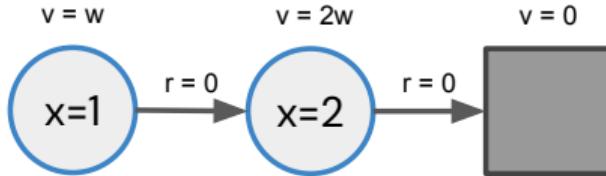
Suppose  $\gamma > \frac{1}{2}$ . Then,

When  $w_t > 0$  and , then  $w_{t+1} > w_t$

When  $w_t < 0$  and , then  $w_{t+1} < w_t \implies w_t$  diverges to  $+\infty$  or  $-\infty$



## Recap: Deadly triad



- ▶ What if we use multi-step returns?
- ▶ Still consider only updating the left-most state

$$\begin{aligned}\Delta w &= \alpha(r + \gamma(G_t^\lambda - v(s))) \\ &= \alpha(2\gamma(1 - \lambda) - 1)w\end{aligned}$$

- ▶ The multiplier is negative when  $2\gamma(1 - \lambda) < 1 \implies \lambda > 1 - \frac{1}{2\gamma}$
- ▶ E.g., when  $\gamma = 0.9$ , then we need  $\lambda > 4/9 \approx 0.45$
- ▶ Conclusion: if we do not bootstrap too much, we can learn better



## Reducing variance: **adaptive** bootstrapping

- ▶ We don't want to bootstrap too much  $\implies$  **deadly triad**
- ▶ We don't want to bootstrap too little  $\implies$  **high variance**
- ▶ Can we adaptively bootstrap 'just enough'?
- ▶ Idea: bootstrap **adaptively** only in as much as you go off-policy



## Reducing variance: adaptive bootstrapping

- ▶ Recall  $\delta_t^{\rho\lambda} = \rho_t(\delta_t + \gamma\lambda\delta_{t+1}^{\rho\lambda})$
- ▶ Let's add an initial bootstrap parameter, and make these time-dependent

$$\delta_t^{\rho\lambda} = \lambda_t\rho_t(\delta_t + \gamma\delta_{t+1}^{\rho\lambda})$$

(If  $\lambda_t = 1$ , we obtain the previous version)

- ▶ We can pick  $\lambda_t$  **separately on each time step**
- ▶ Idea: pick it such that, for all  $t$ ,  $\lambda_t\rho_t \leq 1$ :

$$\lambda_t = \min(1, 1/\rho_t)$$

- ▶ Intuition: when we are too off-policy ( $\rho$  is far from one) truncate the sum of errors
- ▶ This is the same as bootstrapping there



## Reducing variance: adaptive bootstrapping

$$\lambda_t = \min(1, 1/\rho_t)$$

- ▶ This is known as **ABTD** (Mahmood et al. 2017) or **v-trace** (Espeholt et al. 2018)
- ▶ We are free to choose different ways to bootstrap: in the tabular case all these methods will be updating towards some mixture of multi-step returns, and therefore converge
- ▶ In deep RL this really helps, especially for policy gradients  
(Policy gradients do not like biased return estimates – we will get back to that)
- ▶ This is used a lot these days



## Reducing variance: tree backup

- ▶ Picking  $\lambda_t = \min(1, 1/\rho_t)$  is not the only way to adaptively bootstrap
- ▶ One more option, consider the Bellman operator for action values

$$q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) q_\pi(S_{t+1}, a) \mid A_t = a, S_t = s]$$

- ▶ Note: the expectation does not depend on  $\pi$ , because we condition on the action  $a$
- ▶ Idea: sample this, then replace only the action you selected:

$$G_t = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) q(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1}$$

- ▶ We remove only the expectation  $q(S_{t+1}, A_{t+1})$  of the action actually selected, and replace it with the return
- ▶ This is **unbiased**, and **low variance!** ( $\pi(A_{t+1}|S_{t+1})$  plays a role similar to  $\lambda$ )
- ▶ It might bootstrap too early though — beware of deadly triads!



# Lecture End



# Deep Reinforcement Learning

Matteo Hessel

UCL 2021

## Recap: Value function approximation

- ▶ Tabular RL does not scale to large complex problems:
  1. Too many states to store in memory
  2. Too slow to learn the values of each state separately,
- ▶ We need to **generalise** what we learn across states.

## Recap: Value function approximation

- ▶ Estimate values (or policies) in an approximate way:
  1. Map states  $s$  onto a suitable "feature" representation  $\phi(s)$ .
  2. Map features to values through a parametrised function  $v_\theta(\phi)$
  3. Update parameters  $\theta$  so that  $v_\pi(s) \sim v_\theta(\phi(s))$

## Recap: Value function approximation

- ▶ Goal: find  $\theta$  that minimises the difference between  $v_\pi$  and  $v_\theta$

$$L(\theta) = E_{S \sim d}[(v_\pi(S) - v_\theta(S))^2]$$

Where  $d$  is the state visitation distribution induced by  $\pi$  and the dynamics  $p$ .

- ▶ Solution: use **gradient descent** to iteratively minimise this objective

$$\Delta\theta = -\frac{1}{2}\alpha\nabla_\theta L(\theta) = \alpha E_{S \sim d}[(v_\pi(S) - v_\theta(S))\nabla_\theta v_\theta(S)]$$

## Recap: Value function approximation

- ▶ Problem: evaluating the expectation is going to be hard in general,
- ▶ Solution: use **stochastic gradient descent**, i.e. sample the gradient update,

$$\Delta\theta = \alpha(G_t - v_\theta(S_t))\nabla_\theta v_\theta(S_t)$$

- ▶ where  $G_t$  is a suitable sampled estimate of the return,
- ▶ Monte Carlo Prediction  $\rightarrow G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots$
- ▶ TD Prediction  $\rightarrow G_t = R_t + \gamma v_\theta(S_{t+1})$

## Deep value function approximation

- ▶ In past lectures, the feature representation was typically "fixed"
- ▶ The parametrised function  $v_\theta$  was just a linear mapping
- ▶ Today, we will consider more complicated non-linear mappings  $v_\theta$
- ▶ A popular choice is to use deep neural network to parametrise such mapping
  - ▶ Known to discover useful feature representation tailored to the specific task
  - ▶ We can leverage extensive research on architectures and optimisation from SL.

## Deep value function approximation

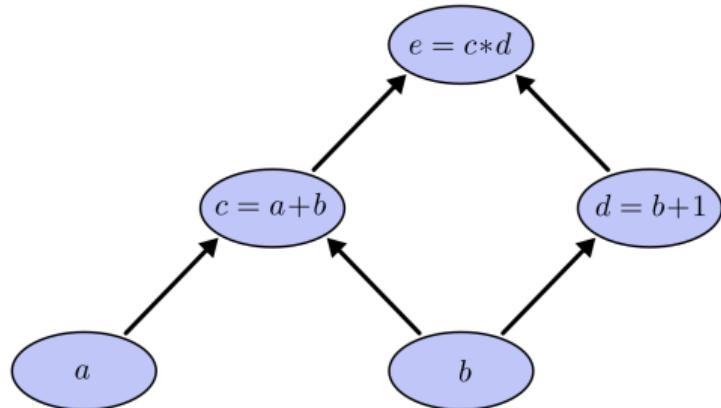
- ▶ Parametrise  $v_\theta$  using a **deep neural network**.
- ▶ For instance as a multilayer perceptron:

$$v_\theta(S) = W_2 \tanh(W_1 * S + b_1) + b_2$$

- ▶ where  $\theta = \{W_1, b_1, W_2, b_2\}$
- ▶ when  $v_\theta$  was linear  $\nabla v_\theta$  was trivial to compute
- ▶ how do we compute such gradient if  $v_\theta$  is parameterised by a deep neural net?

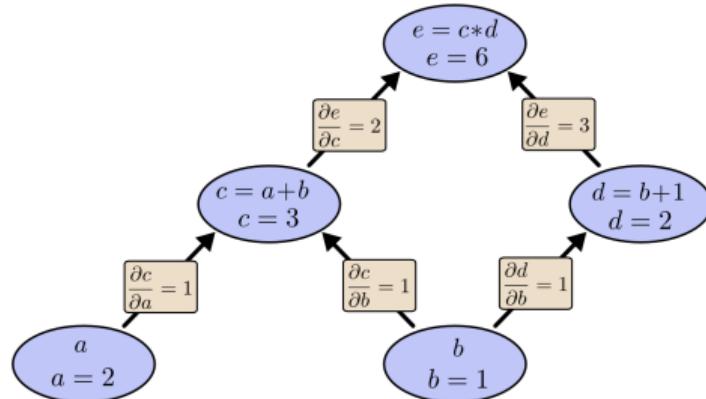
# Computational graphs

- ▶ We can represent computation via direct acyclic graphs
- ▶ specifying the sequence of operations to compute some quantity
- ▶ e.g. we can represent the sequence of operations in a neural network,



# Automatic differentiation

- ▶ If we know how to compute gradients for individual nodes wrt their inputs,
- ▶ we can compute gradients of any node wrt to any other, in one backward sweep,
- ▶ Accumulate the gradient products along paths, sum gradients when paths merge.



# JAX

- ▶ There are many autodiff frameworks to compute gradients in deep networks
- ▶ In this course we will be using JAX:

JAX = Numpy + Autodiff + Accelerators

- ▶ Numpy: the canonical Python library for defining matrices and matrix operations,
- ▶ Autodiff: implemented via tracing by `jax.grad`,
- ▶ Accelerators (GPU/TPU): supported via just in time compilation with `jax.jit`.

## Deep Q-learning

- ▶ Use a neural network to approximate  $q_\theta: O_t \mapsto \mathbb{R}^m$  for  $m$  actions:
  - ▶ first map the input state to a  $h$  dimensional vector
  - ▶ apply a non linear transformation, e.g. a tanh/relu
  - ▶ map the hidden vector to the  $m$  action values
- ▶ We could also pass state \*and\* action as input to the network

# Deep Q-learning in JAX

- ▶ First, we define the neural network  $q_\theta$  using Haiku:

```
1 import haiku as hk
2
3 def forward_pass(obs):
4     network = hk.Sequential([
5         lambda x: jnp.reshape(x, (-1,)),
6         hk.Linear(50),
7         jax.nn.relu,
8         hk.Linear(3)
9     ])
10    return network(obs)
11
12 network_init, network_apply = hk.transform(forward_pass)
```

## Deep Q-learning

- ▶ Update parameters  $\theta$  through the stochastic update:

$$\Delta\theta = \alpha(G_t - q_\theta(S_t, A_t))\nabla_\theta q_\theta(S_t, A_t), \quad G_t = R_{t+1} + \gamma \max_a q_\theta(S_{t+1}, a)$$

- ▶ For consistency with DL notation you may write this as gradient of a pseudo-loss:

$$L(\theta) = \frac{1}{2} \left( R_{t+1} + \gamma \max_a q_\theta(S_{t+1}, a) - q_\theta(S_t, A_t) \right)^2$$

- ▶ Note: we ignore the dependency of the bootstrap target on  $\theta$ ,
- ▶ Note: this is not a true loss!

# Deep Q-learning in JAX

- ▶ Next, we define the update to parameters  $\theta$ :

```
20 @jax.jit
21 def loss_fn(theta, obs_tm1, a_tm1, r_t, d_t, obs_t):
22     q_tm1 = network_apply(theta, obs_tm1)
23     q_t = network_apply(theta, obs_t)
24     target_tm1 = r_t + d_t * jnp.max(q_t)
25     td_error = jax.lax.stop_gradient(target_tm1) - q_tm1[a_tm1]
26     return 0.5 * (td_error)**2
27
28 @jax.jit
29 def update(theta, obs_tm1, a_tm1, r_t, d_t, obs_t):
30     dl_dtheta = jax.grad(loss_fn)(theta, obs_tm1, a_tm1, r_t, d_t, obs_t)
31     return tree_multimap(lambda p, g: p-alpha*g, theta, dl_dtheta)
```

## Looking forward:

Next we will explore more deeply RL with deep function approximation:

- ▶ how do ideas from the previous lectures apply in this setting?
- ▶ how can we make RL algorithms more compatible with deep learning?
- ▶ how can we make deep learning models more suitable for RL?

# Deep learning aware RL

Matteo Hessel

2021

## Issues with online deep RL

We know from deep learning literature that

- ▶ Stochastic gradient descent assumes gradients are sampled i.i.d.
- ▶ Using mini-batches instead of single samples is typically better,

However in online reinforcement learning algorithm:

- ▶ We perform an update on every new update,
- ▶ Consecutive updates are strongly correlated.

## Friendlier data distributions?

Can we make RL more deep learning friendly?

- ▶ In the planning lectures we discussed Dyna-Q and Experience Replay,
- ▶ these mix online updates with updates on data sampled from
  1. a buffer of past experience
  2. a learned model of the environment
- ▶ Both approaches can
  1. reduce correlation between consecutive updates,
  2. support mini-batch updates instead of vanilla SGD.

## Other approaches to Online Deep RL

Experience replay / planning with learned models are not the only ways to address this:

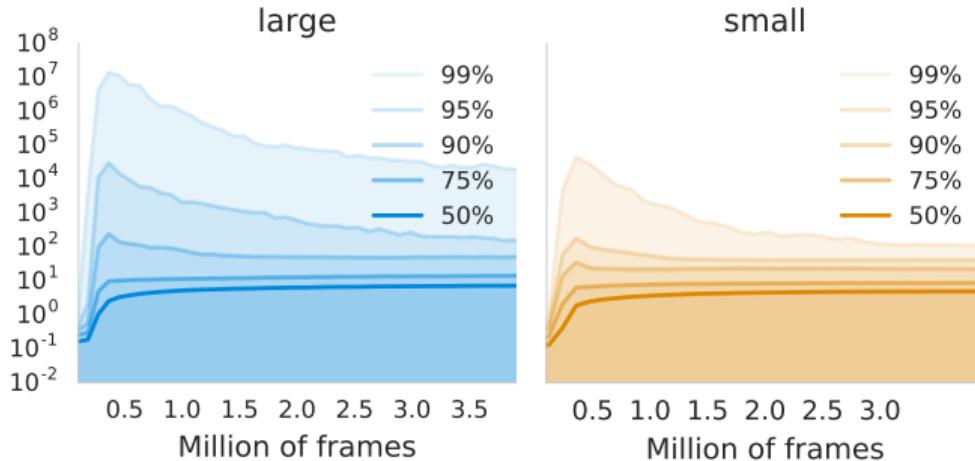
- ▶ better online algorithms: e.g. eligibility traces,
- ▶ better optimisers: e.g. momentum
- ▶ change the problem setting itself: e.g. parallel environments

## The deadly triad

- ▶ If we use Dyna-Q or experience replay (DQN), we are combining:
  1. **Function approximation**: we are using a neural network to fit action values,
  2. **Bootstrapping**: we bootstrap on  $\max_a Q_\theta(s, a)$  to construct the target,
  3. **Off-policy learning**: the replay hold data from a mixture of past policies.
- ▶ What about the deadly triad?
- ▶ Is this a sane thing to do?

## The deadly triad in deep RL (van Hasselt et al. 2018)

- ▶ Empirically we actually find that unbounded divergence is rare,
- ▶ More common are value explosions that recover after an initial phase,



- ▶ This phenomenon is also referred to as "soft-divergence".

## Target networks

- ▶ Soft divergence still cause value estimates to be quite poor for extended periods.
- ▶ We can address this in our deep RL agents using a separate **target network**:
  1. Hold fixed the parameters used to compute the bootstrap targets  $\max_a Q_\theta(s, a)$ ,
  2. Only update them periodically (every few hundreds or thousands of updates).
- ▶ This breaks the feedback loop that sits at the heart of the deadly triad.

## Deep double Q-learning (van Hasselt et al. 2016)

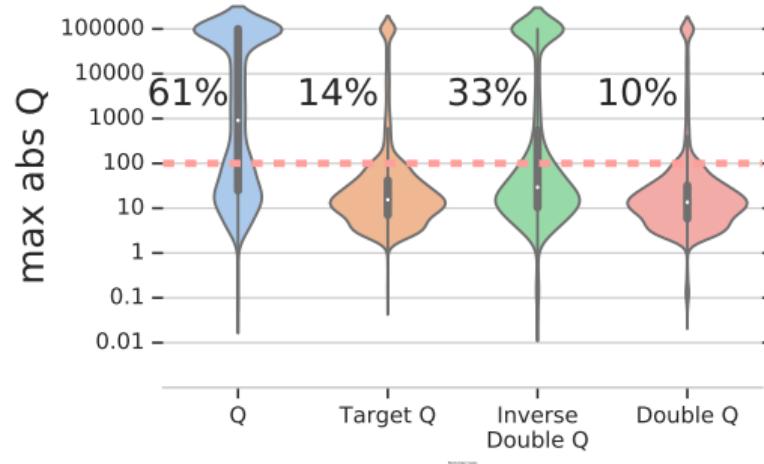
- ▶ Q-learning has an overestimation bias, that can be corrected by double Q-learning

$$L(\theta) = \frac{1}{2} \left( R_{i+1} + \gamma \left[ q_{\theta^-}(S_{i+1}, \operatorname{argmax}_a q_\theta(S_{i+1}, a)) \right] - q_\theta(S_i, A_i) \right)^2$$

- ▶ Great combination with target networks: we can use the frozen params as  $\theta^-$ .
- ▶ What is the effect of double Q-learning on the likelihood of soft divergence?

# The deadly triad in deep RL - estimators

- ▶ The form of the statistical estimator of the return matters for divergence!



## Prioritized replay (Schaul et al. 2016)

- ▶ DQN samples uniformly from replay
- ▶ Idea: prioritize transitions on which we can learn much
- ▶ Basic implementation:

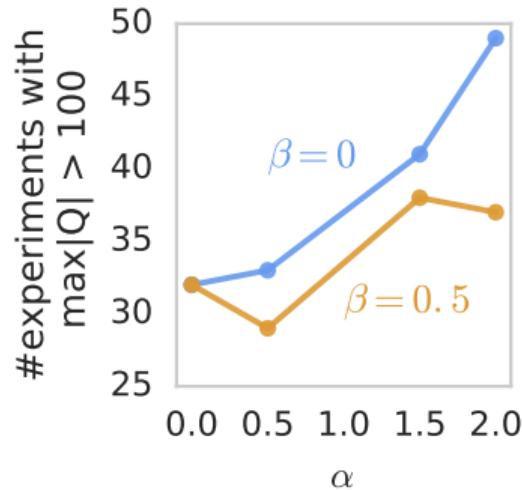
$$\text{priority of sample } i = |\delta_i|,$$

where  $\delta_i$  was the TD error on the last time this transition was sampled

- ▶ Sample according to priority
- ▶ Typically involves some additional design choices

## The deadly triad in deep RL - state distribution

- ▶ We bias sampled states away from the state visitation under the agent policy,
- ▶ Our updates are going to be even more off-policy!



- ▶ We can use importance sampling to correct at least partially.

## Multi-step control

- ▶ Define the  $n$ -step Q-learning target

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n \underbrace{q_{\theta^-}(S_{t+1}, \operatorname*{argmax}_a q_{\theta}(S_{t+1}, a))}_{\text{Double Q bootstrap target}}$$

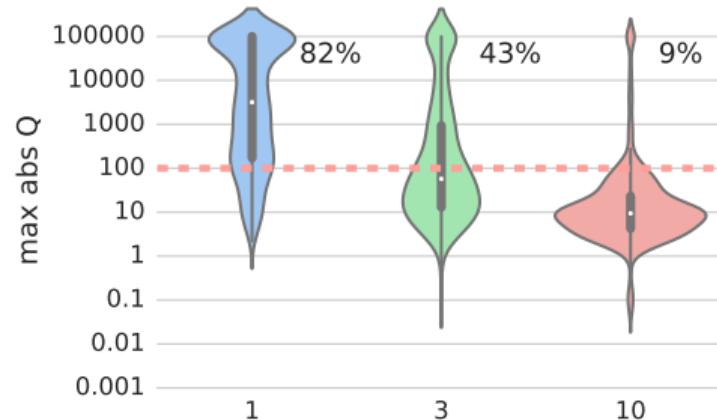
- ▶ Multi-step deep Q-learning

$$\Delta\theta = \alpha(G_t^{(n)} - q_{\theta}(S_t, A_t)) \nabla_{\theta} q_{\theta}(S_t, A_t)$$

- ▶ Return is partially on-policy, bootstrap is off-policy
- ▶ A well-defined target: “*On-policy for  $n$  steps, and then act greedy*”
- ▶ That’s okay — less greedy, but still a policy improvement.

## The deadly triad in deep RL - multi step targets

- ▶ Multi-step targets allow to trade-off bias and variance,
- ▶ They also reduce our reliance on bootstrapping,
- ▶ As a result they also reduce the likelihood of divergence.



# RL aware Deep learning

Matteo Hessel

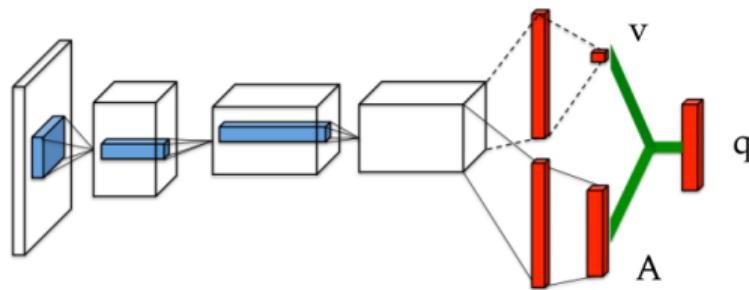
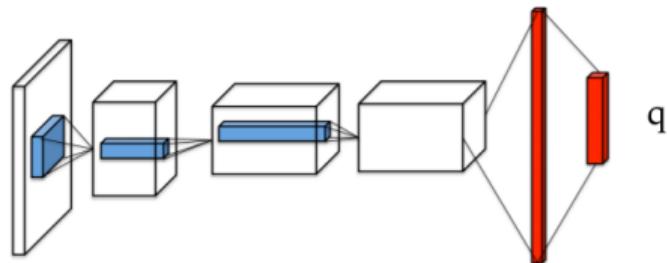
2021

## RL aware deep learning: architectures

- ▶ Much of the successes of deep learning have come from encoding the right inductive bias in the network structure:
  - ▶ Translational invariance in image recognition → convolutional nets,
  - ▶ Long term memory → gating in LSTMs,
- ▶ We shouldn't just copy architectures designed for supervised problems,
- ▶ What are the right architectures to encode inductive biases that are good for RL?

## Dueling networks (Wang et al. 2016)

- ▶ We can decompose  $q_\theta(s, a) = v_\xi(s) + A_\chi(s, a)$ , where  $\theta = \xi \cup \chi$
- ▶ Here  $A_\chi(s, a)$  is the **advantage** for taking action  $a$



## Dueling networks

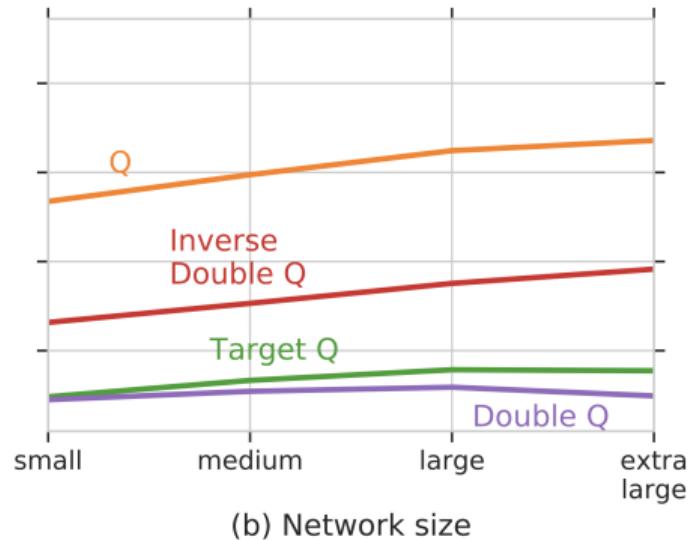


## RL aware deep learning: capacity

- ▶ In supervised deep learning we often find that:  
More Data + More capacity = Better performance
- ▶ The loss is easier to optimise, there is less interference, etc ...
- ▶ How does network capacity affect value function approximation?

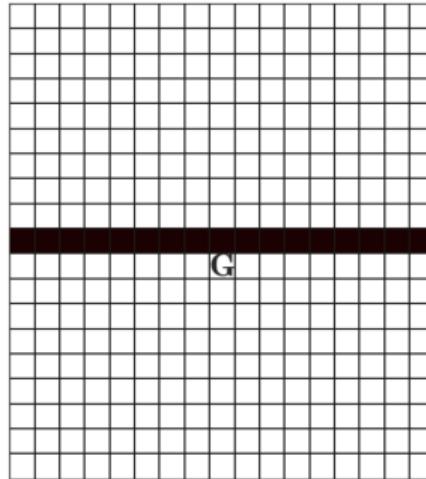
## The deadly triad in deep RL - network capacity

- ▶ Larger networks do typically perform better overall,
- ▶ But... they are however more susceptible to the deadly triad,



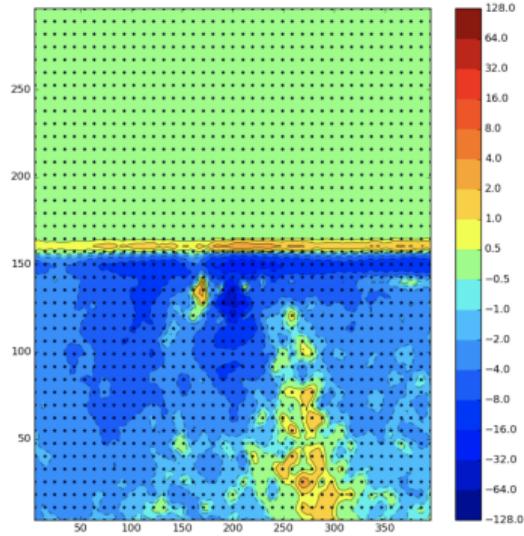
## RL aware deep learning: generalisation

- ▶ The deadly triad shows that generalization in RL can be tricky
- ▶ Consider the problem of value learning in presence of sharp discontinuities of  $v_\pi$

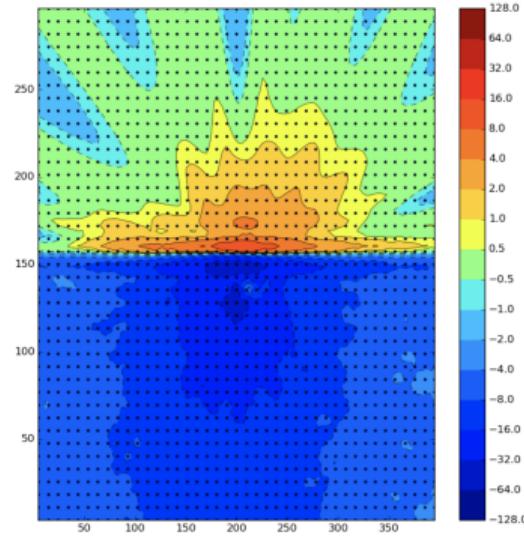


# RL aware deep learning: generalisation

- ▶ TD learning with deep function approximation leads to "leakage propagation"



(c) MC prediction error heatmap



(d) TD prediction error heatmap

## Learning about many things

- ▶ Behind "deadly triad" / "leakage propagation" is inappropriate generalisation,
- ▶ Better representations can help with these issues,
- ▶ E.g. we can share the state representation across many tasks
  1. Predict the value of different policies,
  2. Predict future observations,
  3. Predict/control other "cumulants", different from the main task reward.

# Deep Reinforcement Learning - Part 2

Matteo Hessel

Reinforcement learning, 2021



# Learning about many things

- ▶ Many deep RL algorithms only optimise for a very narrow objective
  - ▶ Narrow objectives induce narrow state representations,
  - ▶ Narrow representation can't support good generalisation,
  - ▶ Deadly triad, leakage propagation, ...
- ▶ Our agents should strive to build **rich knowledge** about the world
  - ▶ Learn about more than just the main task reward.



# Learning about many things

- ▶ But what kind of knowledge should an agent learn about?
- ▶ There are many possible choices, we will discuss two main families of ideas:
  - ▶ GVFAs and UVFAs
  - ▶ Distributional value predictions



# General Value Functions



## The reward hypothesis (Sutton and Barto 2018)

- ▶ All goals can be represented as maximization of a scalar reward,
  - ▶ All useful knowledge may be encoded as predictions about rewards
  - ▶ For instance in the form of "general" value functions (GVFs),



## General value functions (Sutton et al. 2011)

- ▶ A GVF is conditioned on more than just state and actions

$$q_{c,\gamma,\pi}(s, a) = \mathbb{E} [C_{t+1} + \gamma_{t+1} C_{t+2} + \gamma_{t+1}\gamma_{t+2} C_{t+3} + \dots | S_t = s, A_{t+i} \sim \pi(S_{t+i})]$$

where  $C_t = c(S_t)$  and  $\gamma_t = \gamma(S_t)$  where  $S_t$  could be the environment state

- ▶  $c : \mathcal{S} \rightarrow \mathbb{R}$  is the **cumulant**
  - ▶ Predict many things, including — but not limited to — reward
- ▶  $\gamma : \mathcal{S} \rightarrow \mathbb{R}$  is the **discount** or termination
  - ▶ Predict for different time horizons  $\gamma$
- ▶  $\pi : \mathcal{S} \rightarrow \mathcal{A}$  is the **target policy**
  - ▶ Predict under many different (hypothetical) policies  $\pi$



## Example: Simple predictive questions

- ▶ Policy Evaluation
  - ▶  $C_t = R_t$ , under the agent's policy  $\pi$  and discount  $\gamma$



## Example: Simple predictive questions

- ▶ Policy Evaluation
  - ▶  $C_t = R_t$ , under the agent's policy  $\pi$  and discount  $\gamma$
- ▶ Reward prediction:
  - ▶  $C_t = R_t, \gamma = 0$ , under the agent's policy  $\pi$



## Example: Simple predictive questions

- ▶ Policy Evaluation
  - ▶  $C_t = R_t$ , under the agent's policy  $\pi$  and discount  $\gamma$
- ▶ Reward prediction:
  - ▶  $C_t = R_t$ ,  $\gamma = 0$ , under the agent's policy  $\pi$
- ▶ Next state prediction:
  - ▶  $\{C_t^i = S_t^i\}_i$ ,  $\gamma = 0$ , under the agent's policy  $\pi$
  - ▶  $\{C_t^i = \phi^I(S_t)\}_i$ ,  $\gamma = 0$ , under the agent's policy  $\pi$



## Predictive state representations (Littman et al. 2002)

- ▶ A large diverse set of GVF predictions
  - ▶ will be a sufficient statistics for any other prediction,
  - ▶ including the value estimates for the main task reward.
- ▶ In **predictive state representations** (PSR):
  - ▶ Use the predictions themselves as representation of state,
  - ▶ Learn policy and values as a linear function of these predictions,

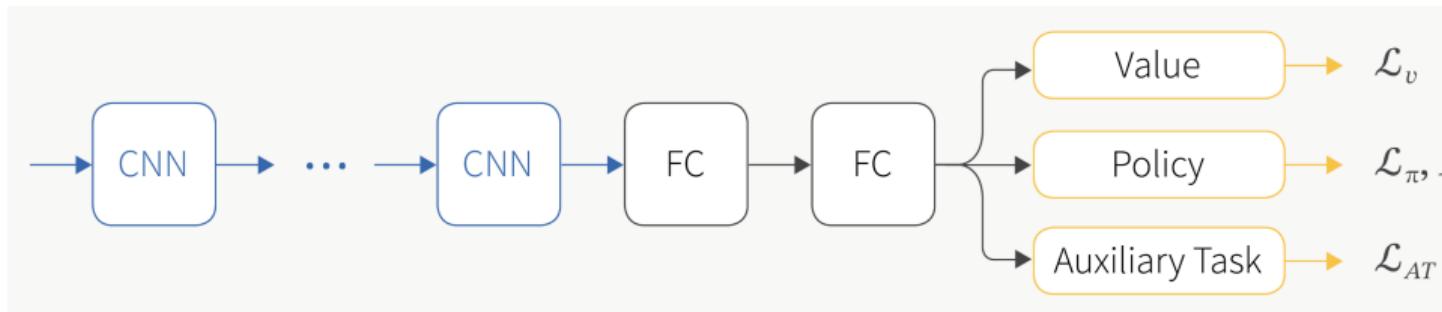


# GVFs as Auxiliary Tasks

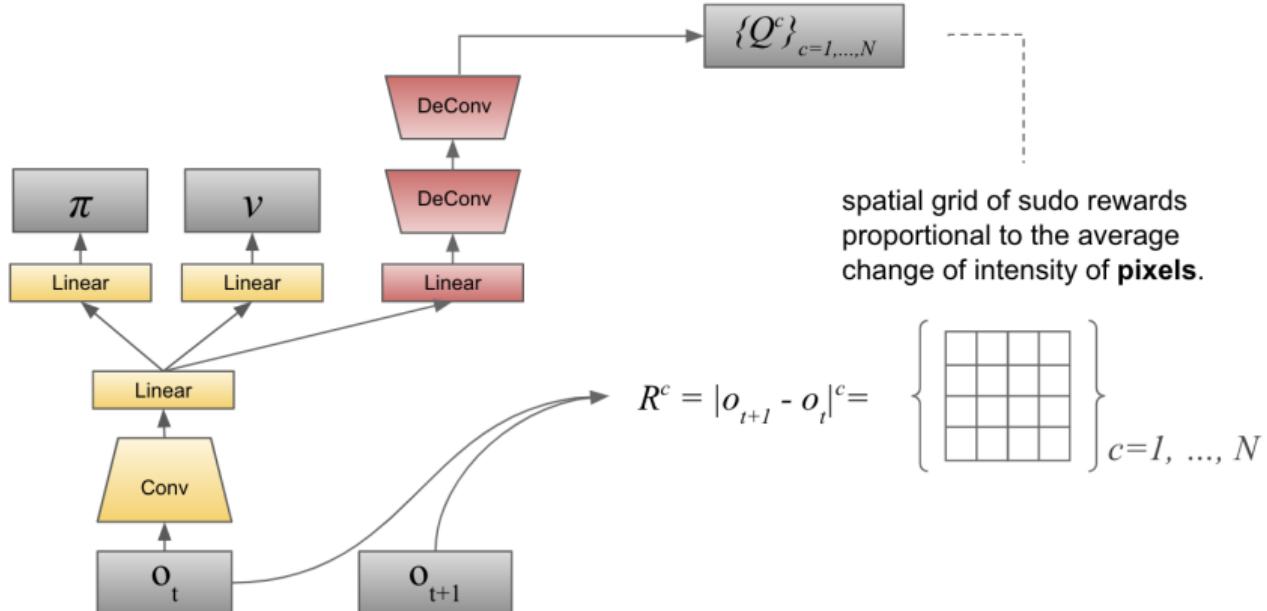


## GVPs as auxiliary tasks (Jaderberg et al., 2016)

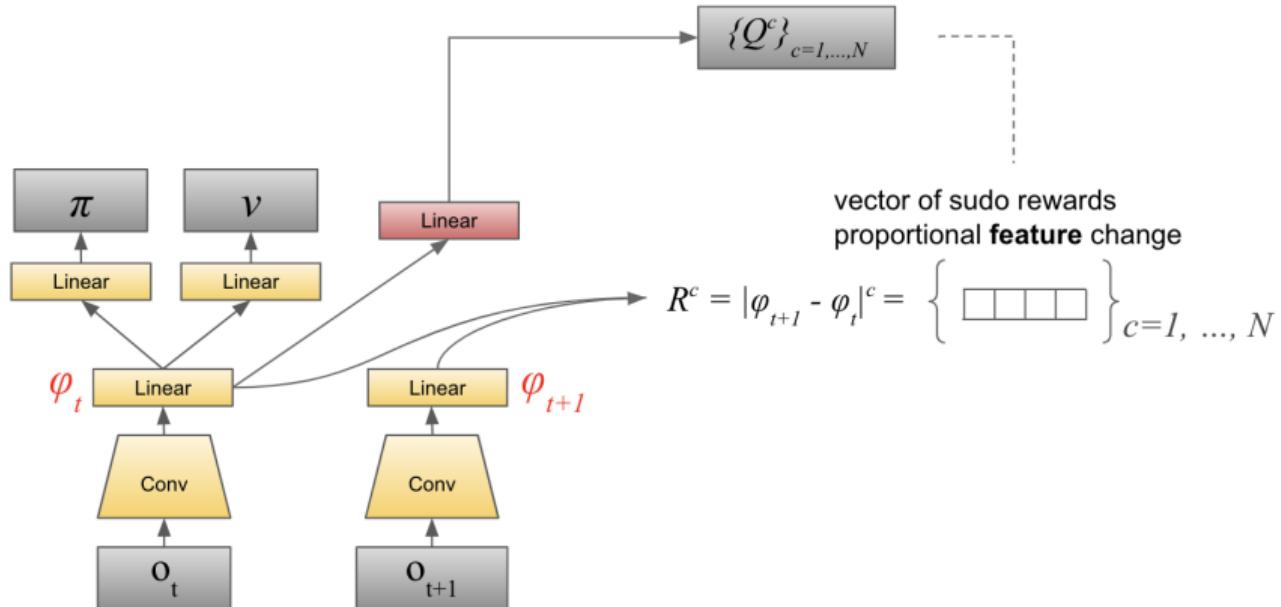
- ▶ GVPs can also be used as **auxiliary tasks**,
  - ▶ that share part of the neural network,
  - ▶ minimise jointly the losses for the main task reward and the auxiliary GVPs
  - ▶ force the shared hidden layers to be more robust,



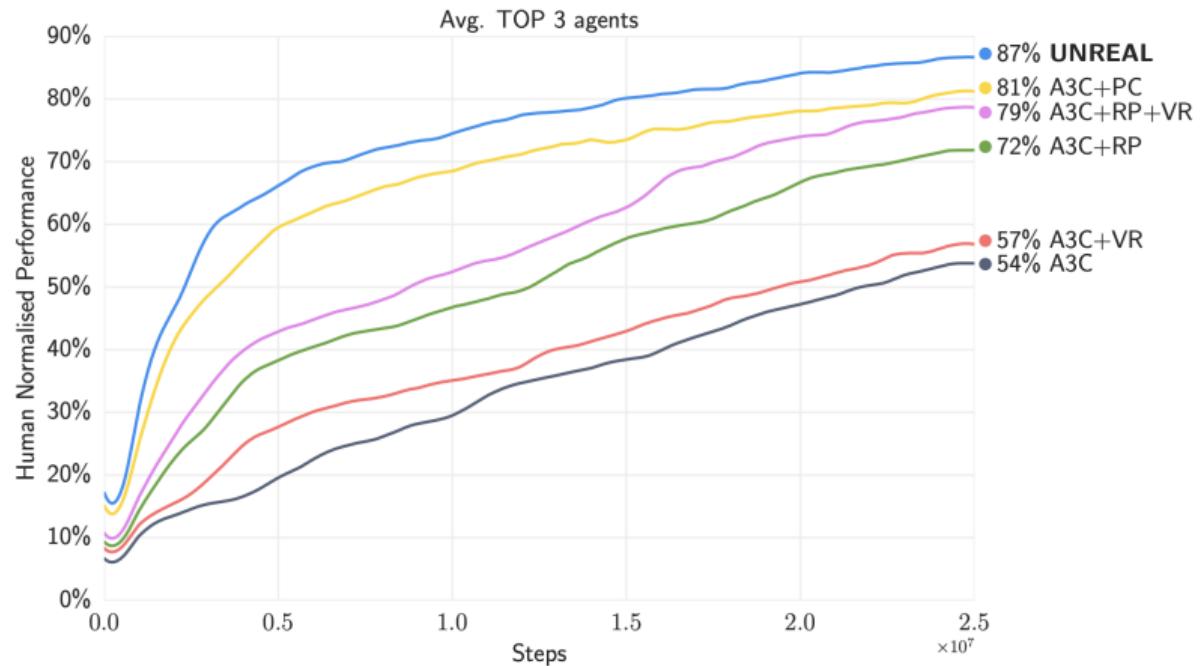
## Example: Pixel Control (Jaderberg et al. 2016)



## Example: Feature Control (Jaderberg et al. 2016)

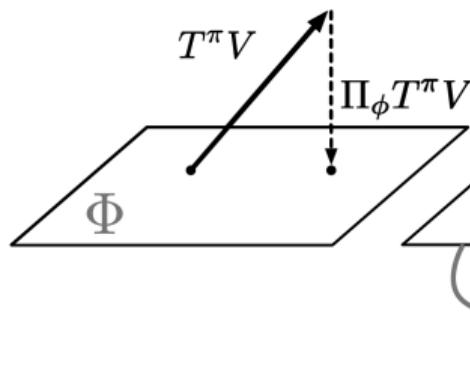


# The effect of auxiliary tasks

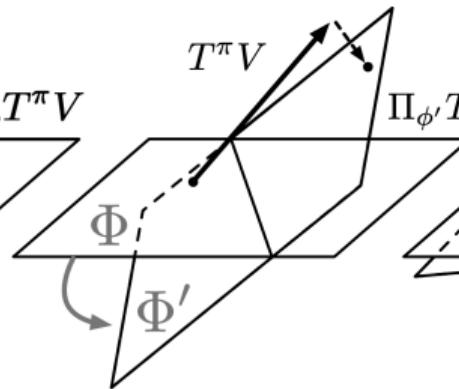


# GVPs as auxiliary tasks

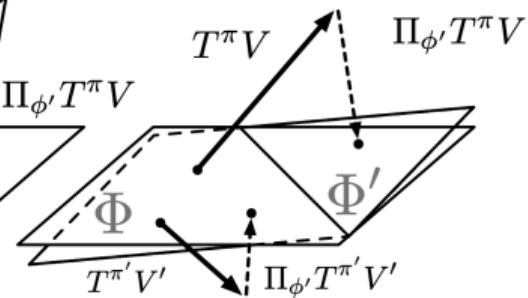
Linear RL



Deep RL

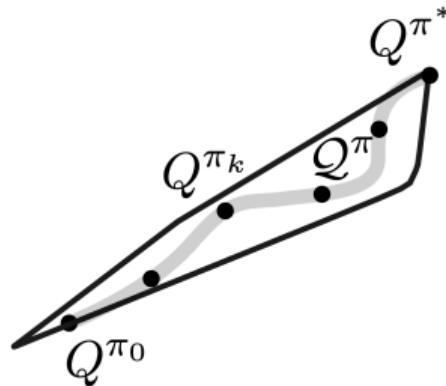


Deep RL  
(with auxiliary)



## Value-Improvement Path (Dabny et al. 2020)

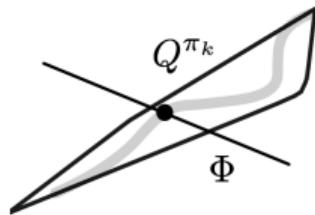
- ▶ Why regularise the representation?
  - ▶ During learning we need to approximate many value functions,
  - ▶ As we are tracking a continuously improving agent policy,
  - ▶ Must support all functions in the **value improvement path** from  $Q^{\pi_0}$  to  $Q^{\pi^*}$



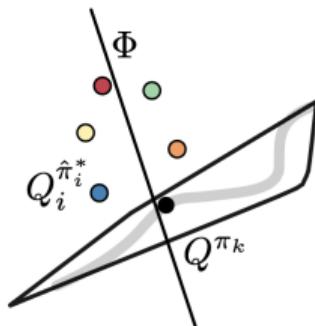
# Value-Improvement Path (Dabny et al. 2020)

- ▶ Which GVF best regularise the representation?

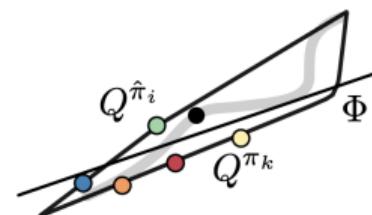
**Value-Only**



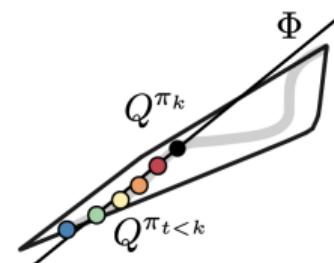
**Cumulant Value**



**Cumulant Policy**



**Past Policies**



# Trade-offs in multi-task learning



## Learning about many things: trade-offs

- ▶ We want to learn as much as possible about the world,
- ▶ We only have limited resources (e.g. capacity, computation, ...),
- ▶ Different tasks compete for these resources,
- ▶ How do we trade-off between competing needs?

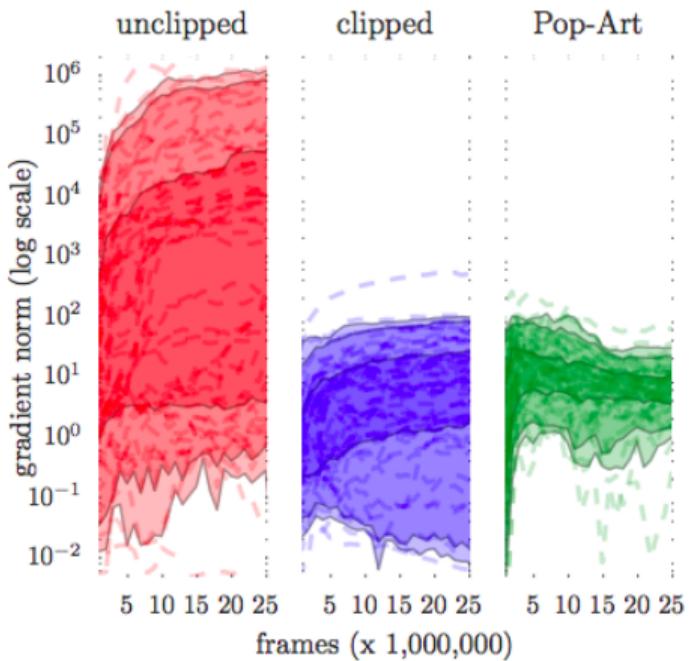


## Learning about many things: trade-offs

- ▶ There is always a trade-off,
- ▶ Even if you don't do anything explicit
- ▶ The magnitude of the updates to shared weight differs across tasks,
  - ▶ e.g. scales linearly with the frequency and size of per-task rewards,
- ▶ Different tasks contribute to different degrees to representation learning,



# Gradient norms



# Update normalization

- ▶ In supervised learning we know before hand the dataset we will learn from,
  - ▶ We can normalise inputs and targets so that they have appropriate scales
- ▶ Many deep learning models do not work well without this
- ▶ In reinforcement learning we do not access to the "full dataset"
  - ▶ The scale of the values we predict also changes over time.
- ▶ Solution: **adaptive normalization of updates**



## Adaptive target normalization (van Hasselt et al. 2016)

1. Observe target, e.g.,  $T_{t+1} = R_{t+1} + \gamma \max_a q_\theta(S_{t+1}, a)$
2. Update normalization parameters:

$$\mu_{t+1} = \mu_t + \eta(T_{t+1} - \mu_t) \quad (\text{first moment / mean})$$

$$\nu_{t+1} = \nu_t + \eta(T_{t+1}^2 - \nu_t) \quad (\text{second moment})$$

$$\sigma_{t+1} = \nu_t - \mu_t^2 \quad (\text{variance})$$

where  $\eta$  is a step size (e.g.,  $\eta = 0.001$ )

3. Network outputs  $\tilde{q}_\theta(s, a)$ , update with

$$\Delta\theta_t \propto \left( \frac{T_{t+1} - \mu_{t+1}}{\sigma_{t+1}} - \tilde{q}_\theta(S_t, A_t) \right) \nabla_\theta \tilde{q}_\theta(S_t, A_t)$$

4. Recover **unnormalized** value:  $q_\theta(s, a) = \sigma_t \tilde{q}_\theta(s, a) + \mu_t$  (used for bootstrapping)



## Preserve outputs

- ▶ Every update to the normalisation changes **all outputs**
- ▶ This seems bad: we should not update values of unrelated states
- ▶ We can avoid this. Typically:

$$\tilde{\mathbf{q}}_{\mathbf{W}, \mathbf{b}, \theta}(s) = \mathbf{W}\phi_\theta(s) + \mathbf{b}.$$

- ▶ Idea: define

$$\mathbf{W}'_t = \frac{\sigma_t}{\sigma_{t+1}} \mathbf{W} \quad \mathbf{b}'_t = \frac{\sigma_t \mathbf{b}_t + \mu_t - \mu_{t+1}}{\sigma_{t+1}}$$

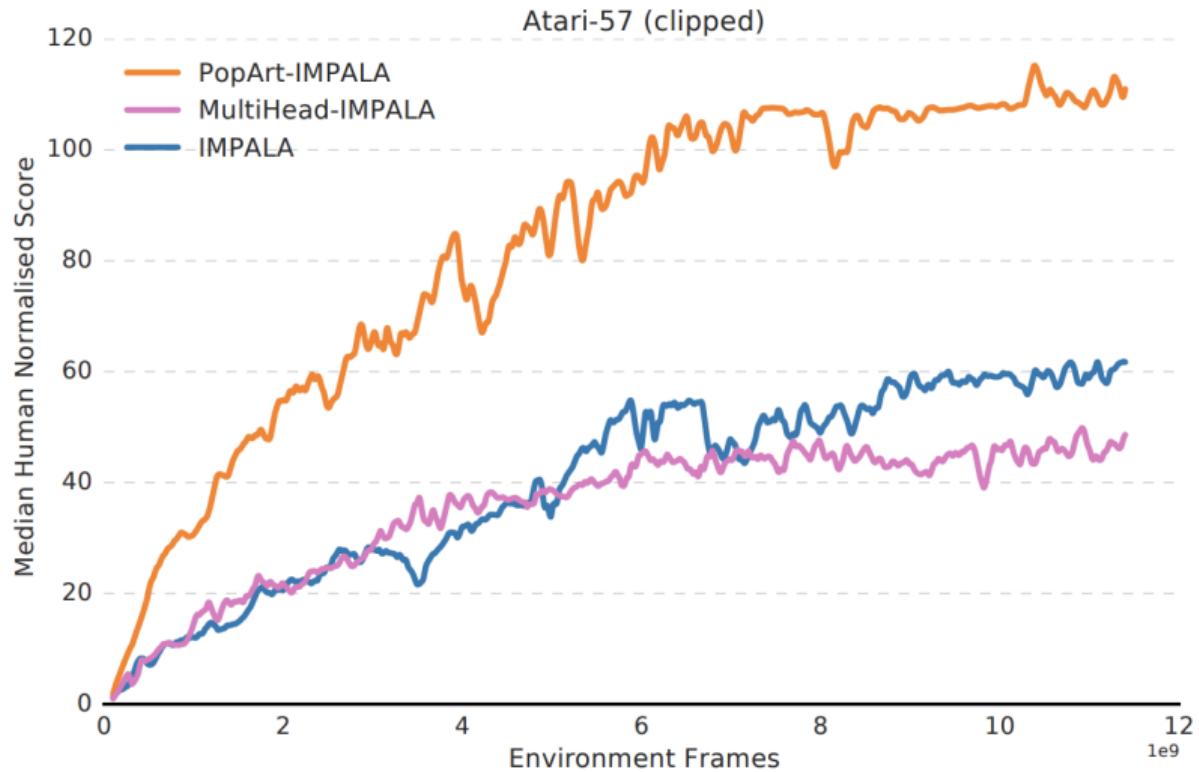
Then

$$\sigma_{t+1} \tilde{\mathbf{q}}_{\mathbf{W}'_t, \mathbf{b}'_t, \theta_t}(s) + \mu_{t+1} = \sigma_t \tilde{\mathbf{q}}_{\mathbf{W}_t, \mathbf{b}_t, \theta_t}(s) + \mu_t$$

- ▶ Then update  $\mathbf{W}'_t$ ,  $\mathbf{b}'_t$  and  $\theta_t$  as normal (e.g., SGD)



# Multi-task PopArt

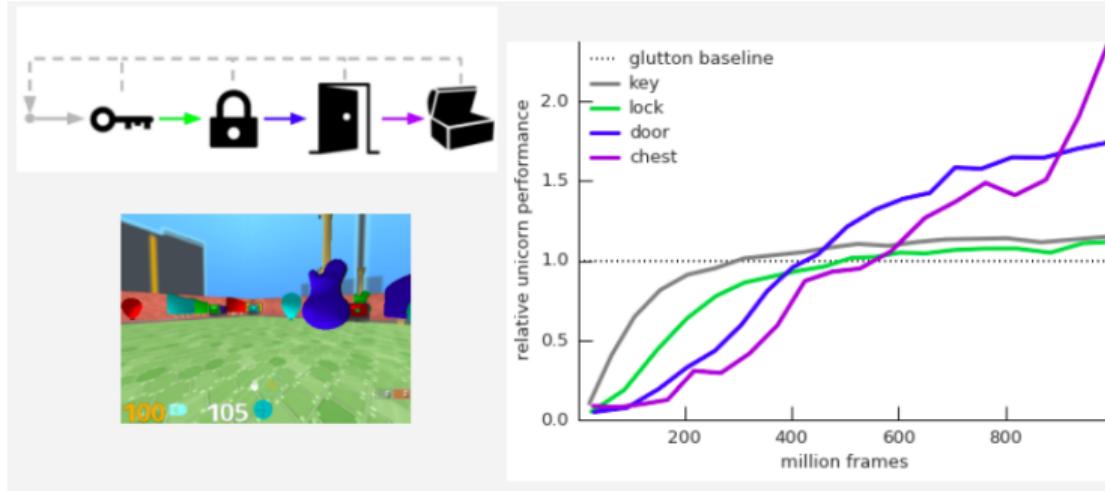


# Open problems in GVF learning



# Off-policy learning

- ▶ We can learn about one cumulant off policy
- ▶ using data from a policy that maximises a different cumulant



‘Unicorn’ (Mankowitz et al., 2018)  
Learn about many things to learn to do the hard thing



## Generalisation (Schaul et al. 2015)

- ▶ GVF-based auxiliary tasks help each other by sharing the representations,
- ▶ otherwise, each prediction is learnt independently
- ▶ Can we generalise what we learn about one GVF to other GVFs?
- ▶ Idea: feed a representation of  $(c, \gamma)$  as **input**
- ▶ Allows generalization across goals/tasks within an environment
- ▶ This kind of GVFs are referred to as **universal value functions**



## Discovery (Veeriah et al. 2019)

- ▶ Which GVF $s$  best regularise the representation?
- ▶ The notion of value improvement path seeks a general answer,
- ▶ Instead, we can learn from experience what cumulants to predict,
- ▶ Using a suitable form of **meta-learning**
- ▶ Meta-gradient RL provides a concrete mechanism to do so



# Distributional RL



# Distributional reinforcement learning

- ▶ GVF<sub>s</sub> still represent predictive knowledge in the form of expectations
  - ▶ We could also move towards learning the **distribution** of returns,
  - ▶ instead of just approximating its expected value,
- ▶ This also provides a richer learning signal
  - ▶ Can help learn more robust representations



## Categorical Return Distributions (Bellemare et al, 2017)

- ▶ A specific instance is **Categorical DQN**
- ▶ Goal: learn a good categorical approximation or the true return distribution
  - ▶ Consider a fixed ‘comb’ distribution on  $\mathbf{z} = (-10, -9.9, \dots, 9.9, 10)^\top$
  - ▶ For each point of support, we assign a ‘probability’  $p_\theta^i(S_t, A_t)$
  - ▶ The approximate distribution of the return  $s$  and  $a$  is the tuple  $(\mathbf{z}, \mathbf{p}_\theta(s, a))$
  - ▶ Our estimate of the expectation is:  $\mathbf{z}^\top \mathbf{p}_\theta(s, a) \approx q(s, a)$  – use this to act



# Categorical Return Distributions

1. Find max action:

$$a^* = \underset{a}{\operatorname{argmax}} \mathbf{z}^\top \mathbf{p}_\theta(S_{t+1}, a)$$

(use, e.g.,  $\theta^-$  for double Q)

2. Update support:

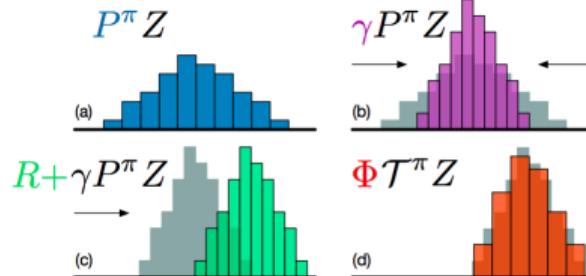
$$\mathbf{z}' = R_{t+1} + \gamma \mathbf{z}$$

3. Project distribution  $(\mathbf{z}', \mathbf{p}_\theta(S_{t+1}, a^*))$  onto support  $\mathbf{z}$

$d' = (\mathbf{z}, \mathbf{p}') = \Pi(\mathbf{z}', \mathbf{p}_\theta(S_{t+1}, a^*))$  where  
 $\Pi$  denotes projection

4. Minimize divergence

$$\text{KL}(d' \| d) = - \sum_i p'_i \frac{\log p'_i}{\log p_\theta^i(S_t, A_t)}$$



Bottom-right: target distribution  
 $\Pi(R_{t+1} + \gamma \mathbf{z}, \mathbf{p}_\theta(S_{t+1}, a^*))$   
Update  $\mathbf{p}_\theta(S_t, A_t)$  towards this



## Quantile Return Distributions (Dabney 2017)

- ▶ There are many other ways to model return distributions,
- ▶ In **Quantile Regressions** we transpose the parametrisation:
  - ▶ instead of adjusting the probabilities of a fixed support... (C51)
  - ▶ ... we can adjust the support associated to a fixed set of probabilities (QR)



End of lecture

