

# Reinforcement Learning for Run-Time Performance Management

Gabriele Russo Russo

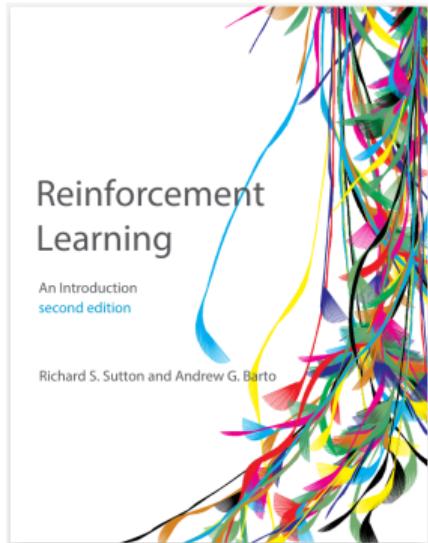
*University of Rome Tor Vergata, Italy*

March–April 2024

# Reference Book

**Reinforcement Learning: An Introduction**  
Richard S. Sutton and Andrew G. Barto

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



## Gabriele Russo Russo

Research Fellow (RTDa) at *University of Rome Tor Vergata*

Main research interests:

- ▶ Cloud & Edge Computing
- ▶ Run-time Performance Management of Distributed Applications
- ▶ Serverless Computing Systems

## Info (2)

Course composed of 5 lectures:

- ▶ Mon, March 25 (15:00-17:00)
- ▶ Mon, April 8 (15:00-17:00)
- ▶ Mon, April 15 (15:00-17:00)
- ▶ Mon, April 22 (15:00-17:00)
- ▶ Mon, April 29 (15:00-17:00)

[Slides](#) will be available after the lectures on my website

[Q&A](#): interrupt me at any time for questions!

# Agenda

- ▶ Run-time performance management
  - ▶ Why?
  - ▶ Examples
  - ▶ Key challenges
- ▶ Introduction to Reinforcement Learning
  - ▶ Markov Decision Processes
  - ▶ Tabular RL
  - ▶ Deep RL
  - ▶ Policy methods

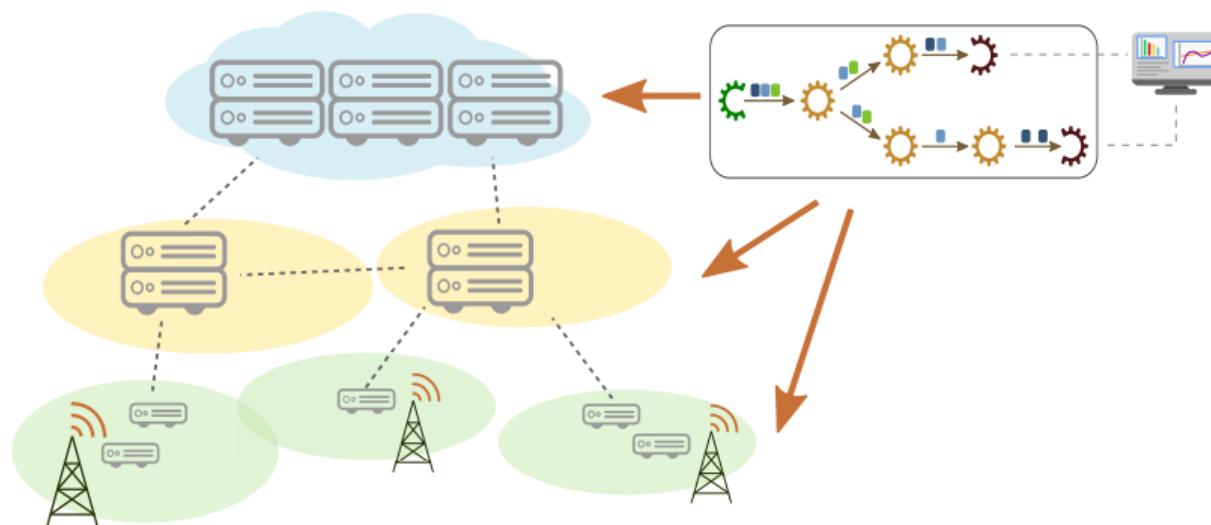
# Scope

Run-time Performance Management

Performance of ...???

# Scenario

Distributed systems and applications in heterogeneous computing environments with Quality-of-Service requirements



e.g., microservice-based apps, data processing pipelines

## Scenario (2)

Distributed systems and applications in heterogeneous computing environments  
with Quality-of-Service requirements

- ▶ This is the scenario I will mostly refer to
- ▶ But the techniques we discuss are not specific to this class of systems!

# Performance Management

- ▶ Almost every computing/network system is expected to guarantee a desired level of performance
  - ▶ a desired Quality-of-Service level, in general
- ▶ Requirements impact system design, implementation, deployment, ...
  - ▶ e.g., architectural choices oriented by performance requirements
  - ▶ e.g., capacity planning for system deployment
- ▶ A lot of work already done before the system is “on”
- ▶ More challenges to come over time!

# Uncertainty

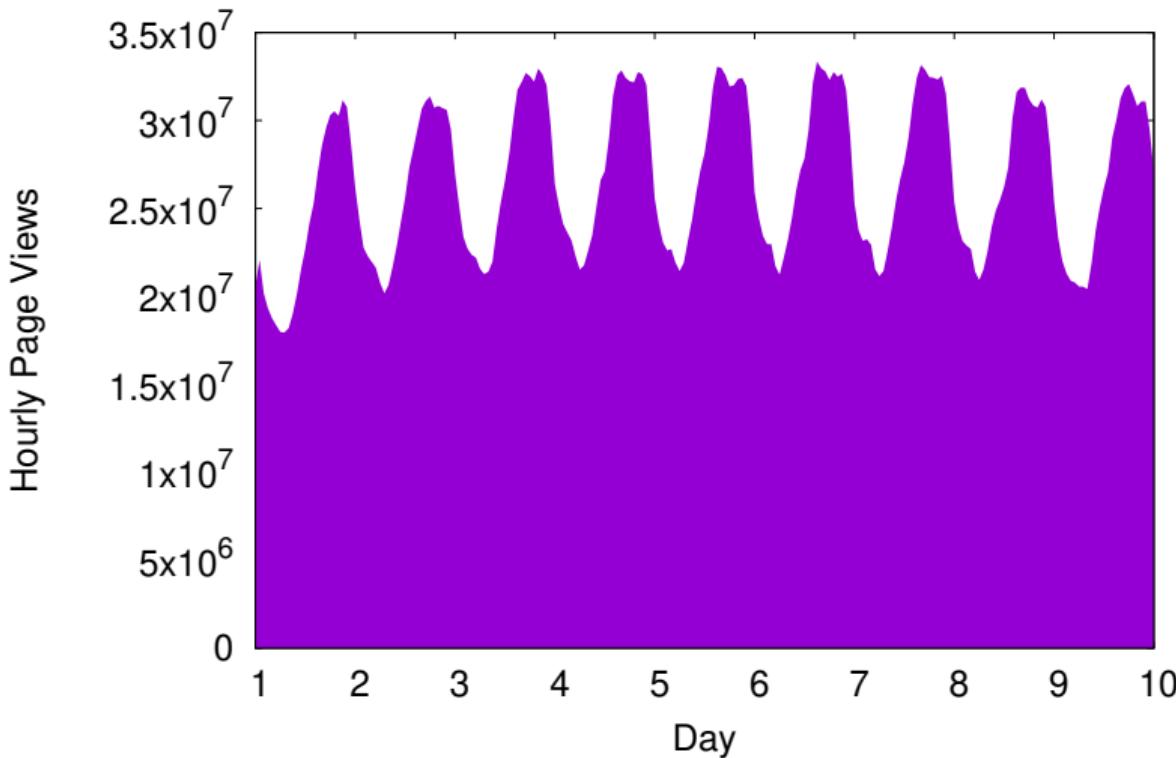
- ▶ We would like systems to consistently deliver the expected level of performance during their operation
- ▶ However, designing and developing systems to meet this goal is very difficult
- ▶ A fundamental challenge ahead: **uncertainty**
- ▶ (Partial or complete) **lack of knowledge** regarding elements of the system and the environment at design time
- ▶ Unpredictable situations (both internal to the system and external) which the system needs to deal with at run-time

# Uncertainty in Modern Computing Environments

- ▶ Working conditions likely changing over time
  - ▶ Workloads
  - ▶ Network conditions (e.g., due to congestion)
  - ▶ Performance of (virtualized) multi-tenant computing resources
  - ▶ Security attacks
  - ▶ Variable monetary costs for on-demand resources
  - ▶ Intermittent energy supplies (e.g., solar power)
  - ▶ User mobility
- ▶ Black-box specifications of applications
- ▶ Heterogeneous computing and network resources
- ▶ Humans in the loop
- ▶ How to develop **automated** performance management solutions to cope with uncertainty?

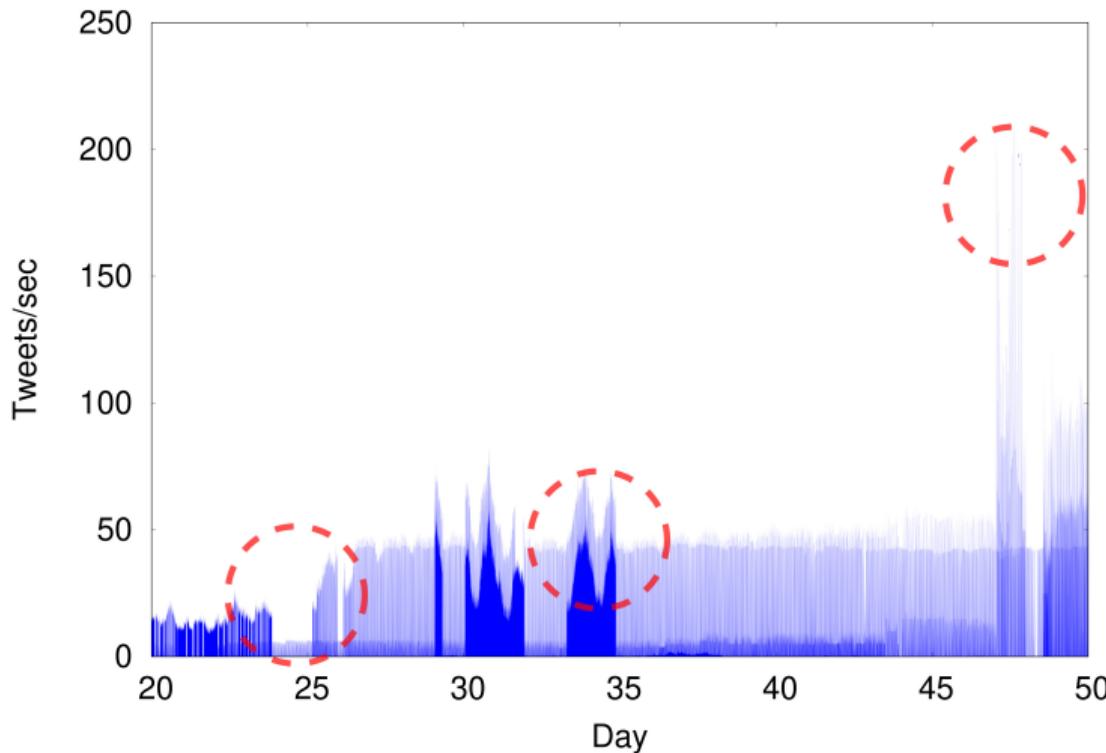
# Example: Workload Variability

Wikipedia page views at beginning of 2016



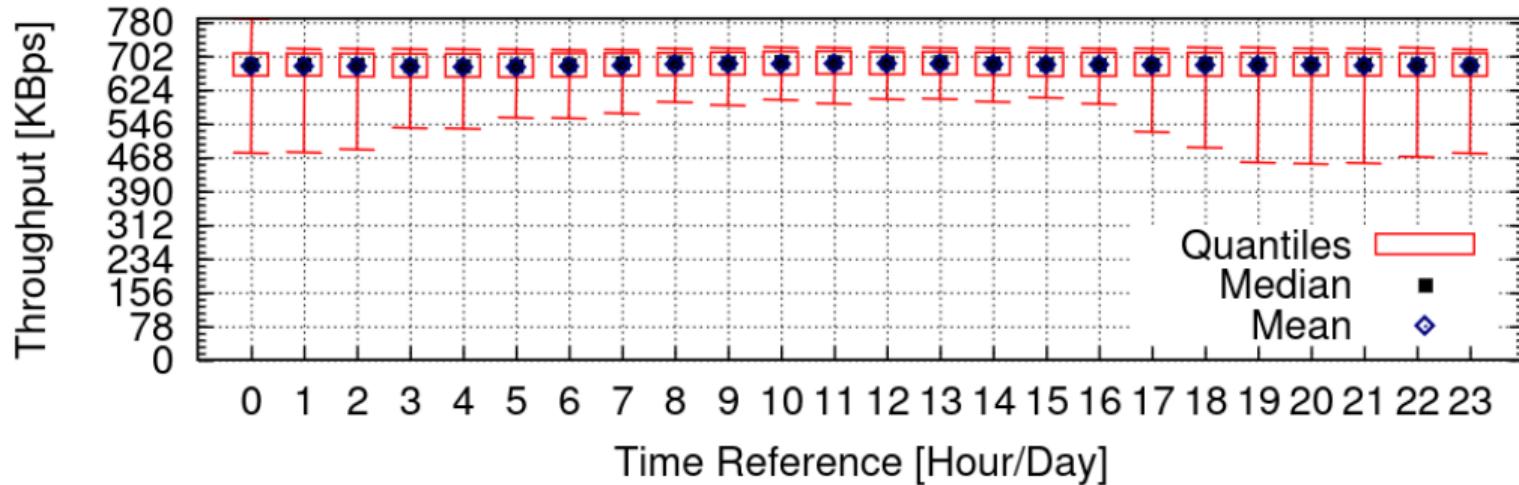
## Example: Workload Variability (2)

Tweets about “COVID” at beginning of 2020



# Example: Cloud Performance Variability

Throughput of Amazon S3 (measurements from 2009)



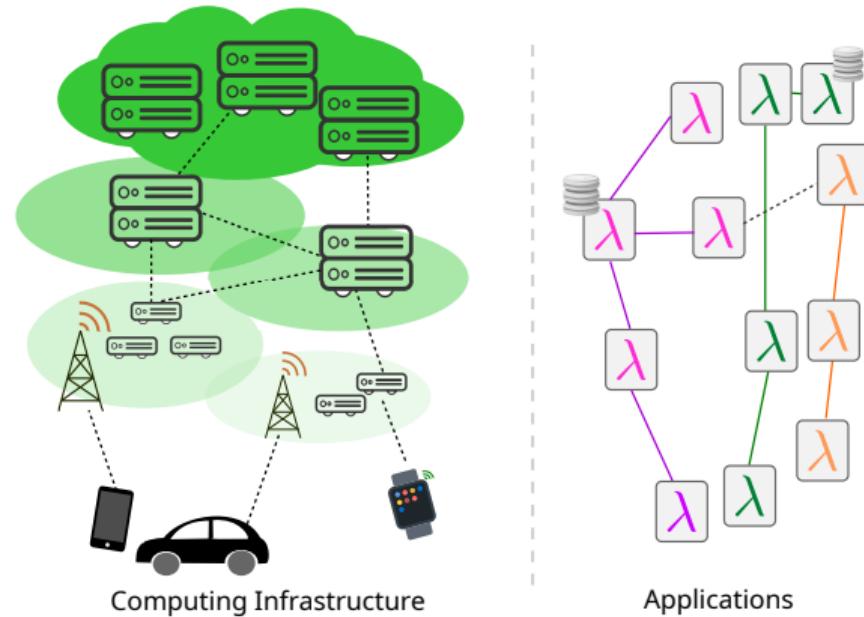
Iosup et al., "On the performance variability of production cloud services" (2011):  
<https://ieeexplore.ieee.org/abstract/document/8102951>

# Run-Time Adaptation

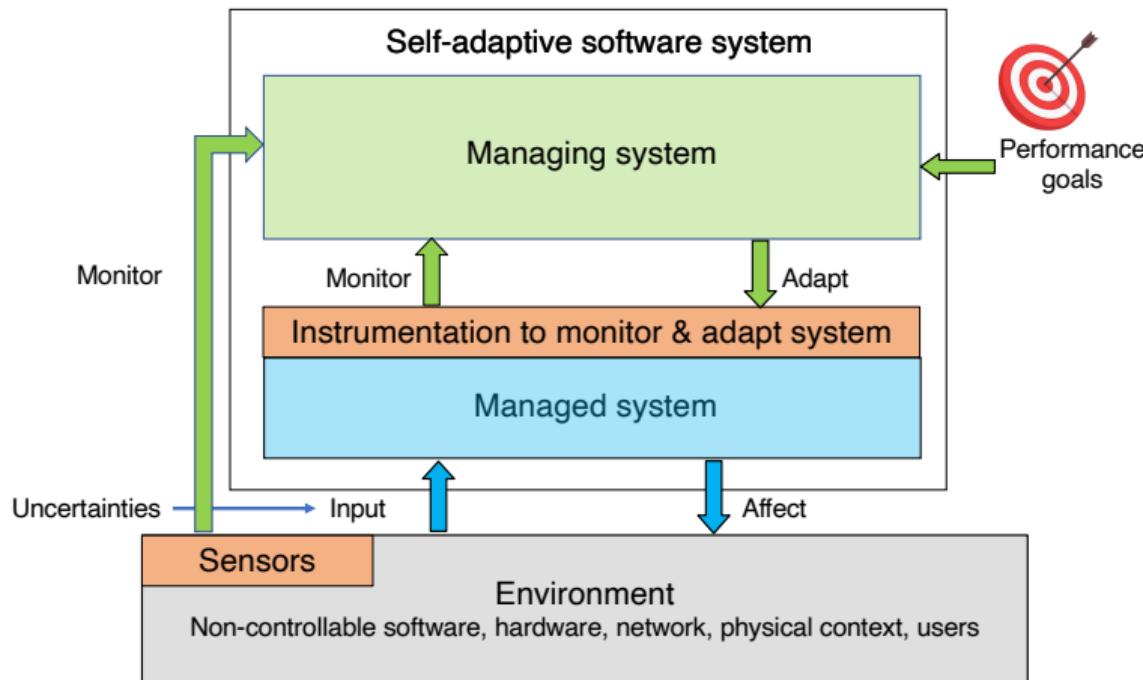
- ▶ How can computing systems deal with changes at run-time?
- ▶ Need to adapt to changing working conditions
  - ▶ Migrating components
  - ▶ Provisioning more/less resources
  - ▶ ...

# Self-Adaptation

- ▶ Adaptation cannot be driven and controlled by humans alone...
- ▶ Scale and complexity of modern applications and infrastructures
- ▶ Applications must be able to self-adapt!



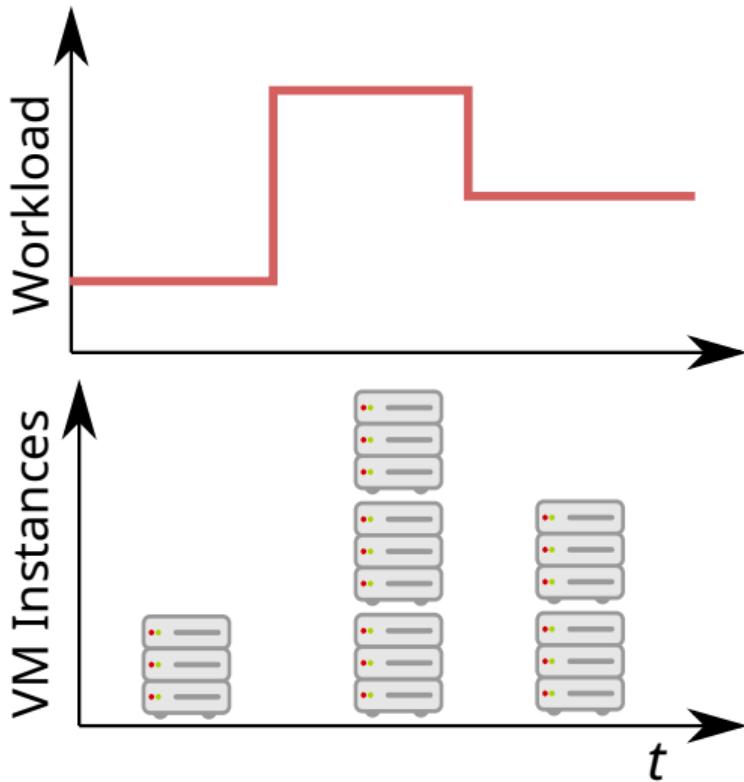
# Self-Adaptive System



Reference book: "An Introduction to Self-Adaptive Systems" [Weyns 2020]

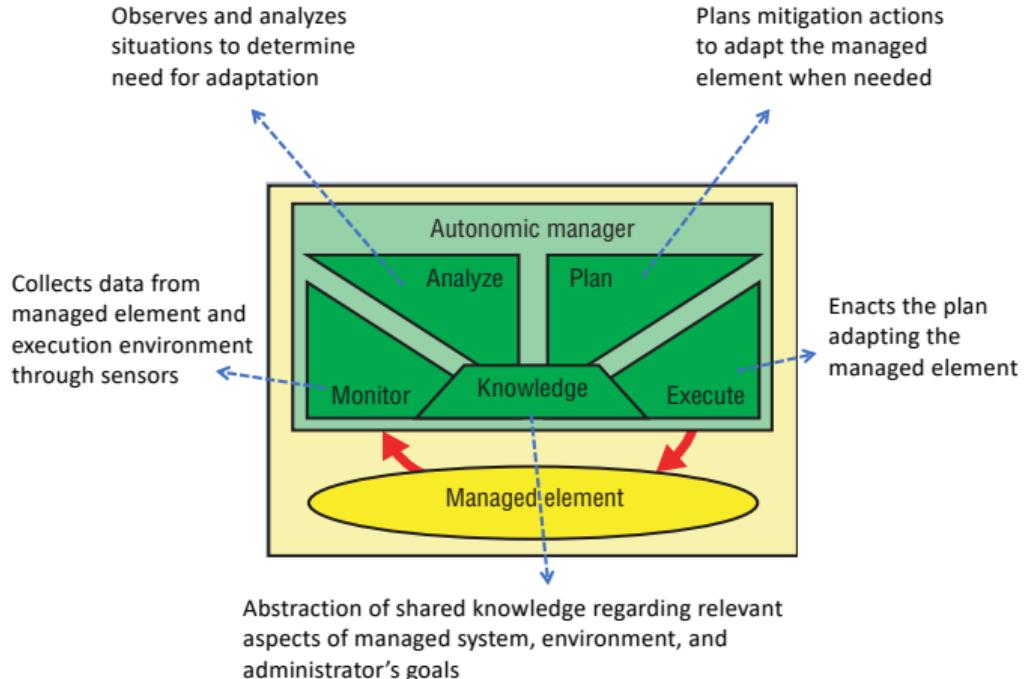
# Example: VM Auto-scaling

- ▶ Consider a simple web application
- ▶ We can acquire **virtual machines** (VMs) from a cloud provider to deploy the application server(s)
- ▶ We can **elastically** acquire/release VMs at any time based on current demand
- ▶ e.g., “whenever the average CPU utilization of the VMs exceeds 70%, create a new VM”



# Reference Architecture for Self-Adaptive Systems

- ▶ Monitor-Analyze-Plan-Execute (MAPE), with shared Knowledge (MAPE-K)



Seminal paper by [Kephart and Chess 2003]

# MAPE: Monitor & Analyze Phases

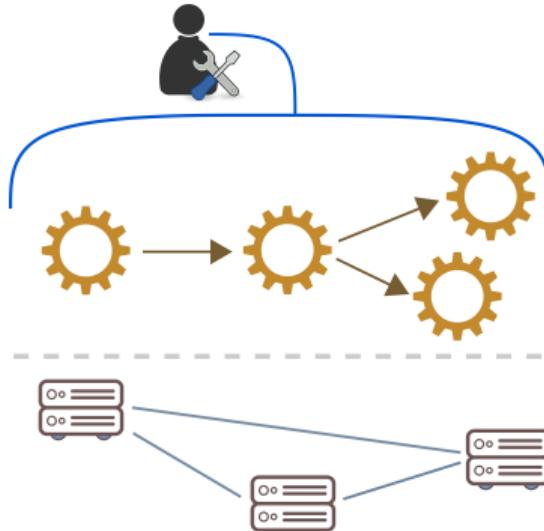
- ▶ Monitor: main design options
  - ▶ When: continuously, on demand
  - ▶ What: resources, workload, application performance, ...
  - ▶ How: architecture (centralized vs. decentralized), methodology (active vs. passive)
  - ▶ Where to store monitored data and how (e.g., some pre-processing)
- ▶ Analyze: main design options
  - ▶ When: event- or time-triggered
  - ▶ How: reactive vs. proactive
    - ▶ Reactive: in reaction to events that have already occurred (e.g., scale-out to react to workload increase)
    - ▶ Proactive: based on prediction so to plan adaptation actions in advance (e.g., scale-out before workload increase effectively occurs)

# MAPE: Plan Phase

- ▶ The most studied MAPE phase
- ▶ A variety of methodologies
  - ▶ Queueing theory
  - ▶ Optimization theory
  - ▶ Meta-heuristics
  - ▶ Control theory
  - ▶ Machine learning (including reinforcement learning)

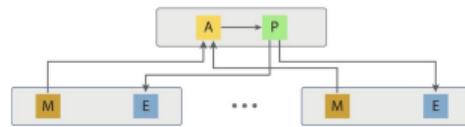
# Architecture of the Managing System

- ▶ How to design the control architecture for distributed applications?
- ▶ First idea: **Centralized MAPE**
  - ▶ All MAPE components in the same node
  - ▶ Global view of the system ✓
  - ▶ Lack of scalability and resiliency X

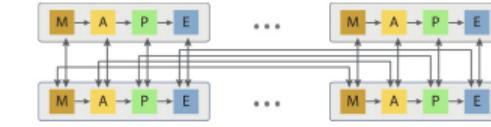


# Architecture of the Managing System

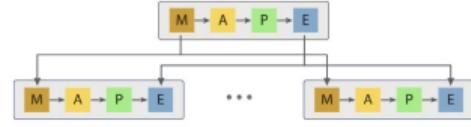
- ▶ Alternative approach: decentralized MAPE
  - ▶ Several patterns to decentralize and distribute MAPE phases
  - ▶ e.g., master-worker, fully decentralized, and hierarchical [Weyns et al. 2013]
  - ▶ No clear winner, depending on system, environment and application features and requirements



Master-worker

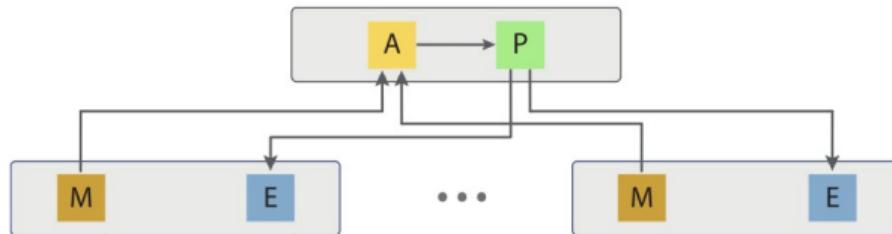
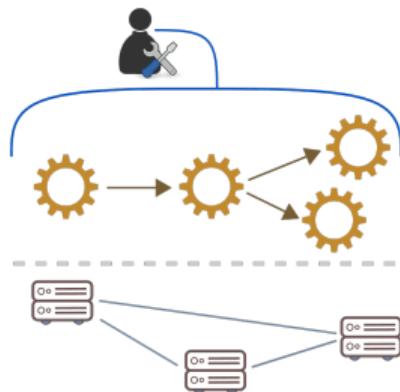


Fully decentralized (e.g., coordinated)



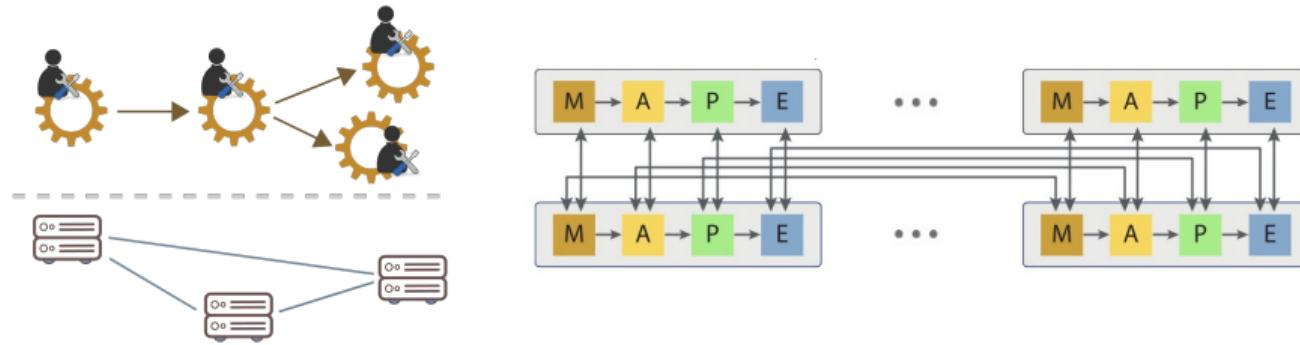
Hierarchical

# Decentralized MAPE: Master-Worker



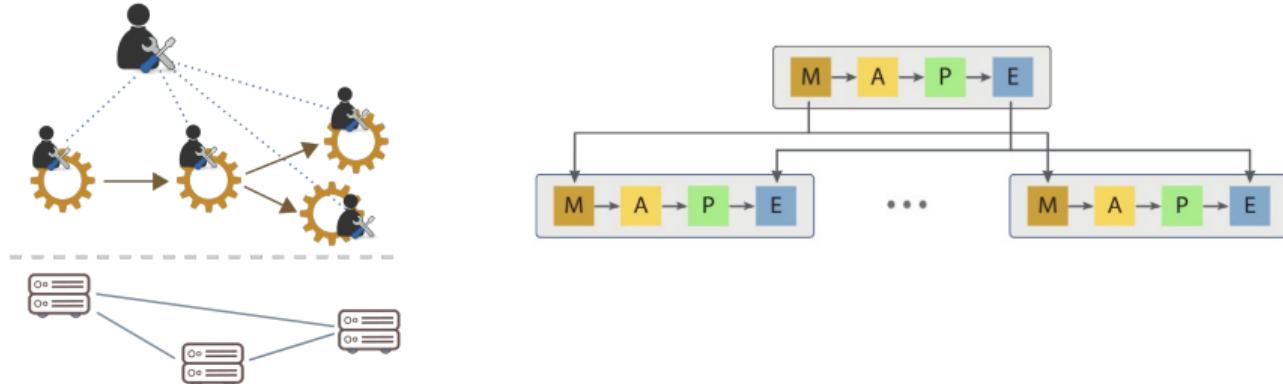
- ▶ Decentralize M and E on workers, keep A and P centralized on master
- ▶ Global view on master ✓
- ▶ Communication overhead, bottleneck risk and single point of failure on master X

# Decentralized MAPE: Fully Decentralized



- ▶ Multiple control loops, each one in charge of some part of the controlled system, possibly coordinated through interaction (or no interaction at all)
- ▶ Cloud/edge applications: distinct control loop per application component
- ▶ Improved scalability ✓
- ▶ More difficult to take joint adaptation decisions X

# Decentralized MAPE: Hierarchical



- ▶ Multiple MAPE loops, which can operate at different time scales and with separation of concerns
- ▶ Top-level MAPE can achieve global goals ✓
- ▶ Non-trivial to identify multiple levels of control X
- ▶ e.g., for Cloud/Edge applications:
  - ▶ Top-level MAPE: entire application
  - ▶ Bottom-level MAPE: application component

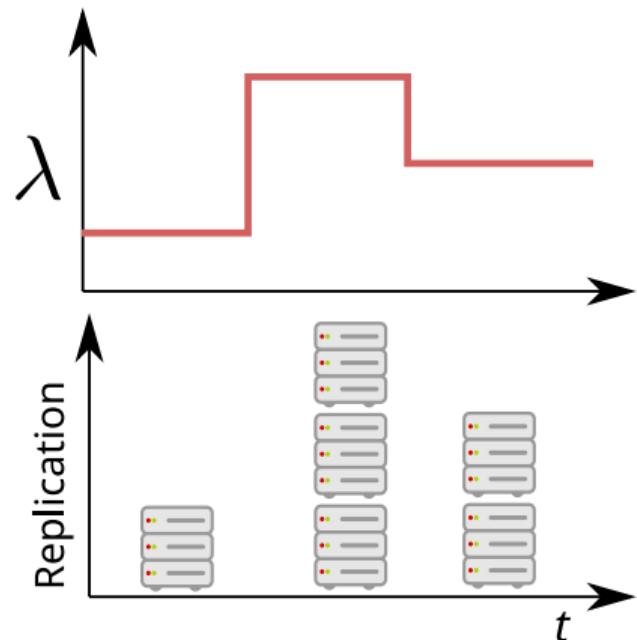
# Introduction to Reinforcement Learning

# Planning Adaptation Actions

- ▶ We focus on the **Analyze** and **Plan** phases of the MAPE loop
- ▶ How to identify and plan the most suitable adaptation actions?

# Example: Auto-scaling

- ▶ Automated provisioning of VMs/containers/threads at run-time
- ▶ Conflicting performance and cost indices:
  - ▶ Performance (e.g., response time)
  - ▶ Monetary cost of allocated resources
  - ▶ Reconfiguration overhead
- ▶ Traditional solution: **threshold-based policies**



# Threshold-based Policies

- ▶ Resource allocation varies according to a set of rules
- ▶ Rule = condition + action
- ▶ Conditions defined in terms of thresholds
- ▶ Model-free approach

if  $x_1 > H_1$  and/or  $x_2 > H_2$  and/or ... for  $D_H$  seconds  
scale-out(N)

if  $x_1 < L_1$  and/or  $x_2 < L_2$  and/or ... for  $D_L$  seconds  
scale-in(N)

## Threshold-based Policies (2)

- ▶ Conditions may involve one or more metrics
  - ▶ e.g., CPU utilization, used memory
- ▶ For each metric, multiple conditions (hence, thresholds) can be defined
- ▶ Usually **upper** (or, **high**) and **lower** (or, **low**) thresholds are used
  - ▶ e.g., for CPU utilization, upper threshold 75% and lower 20%
- ▶ Conditions may also require thresholds to be exceeded for a certain amount of time before triggering an action

## Example of Rules

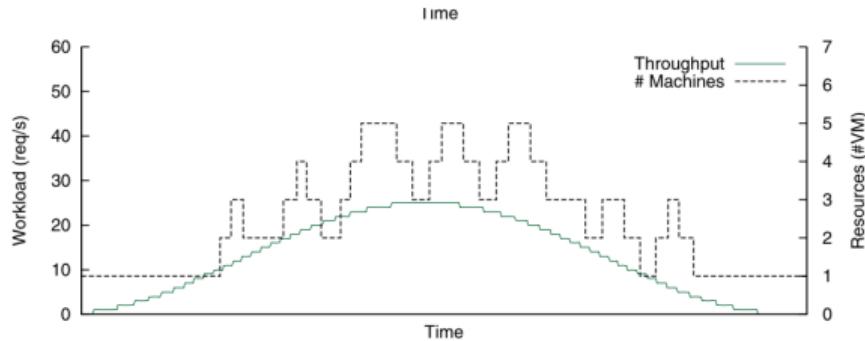
- ▶ If CPU utilization > 70% for at least 1 minute, add one VM
- ▶ If CPU utilization < 30% for at least 5 minutes, terminate one VM
- ▶ If avg. response time > 100ms for at least 30s, increase CPU frequency by 10%

# Issues with Thresholds

- ▶ Threshold-based policies are easy to implement and execute
- ▶ But defining suitable rules is not trivial!
- ▶ Which metrics? System vs application-oriented
  - ▶ System metrics may work across different apps
  - ▶ Application metrics directly mapped onto QoS requirements
- ▶ Which thresholds? Tuning required!

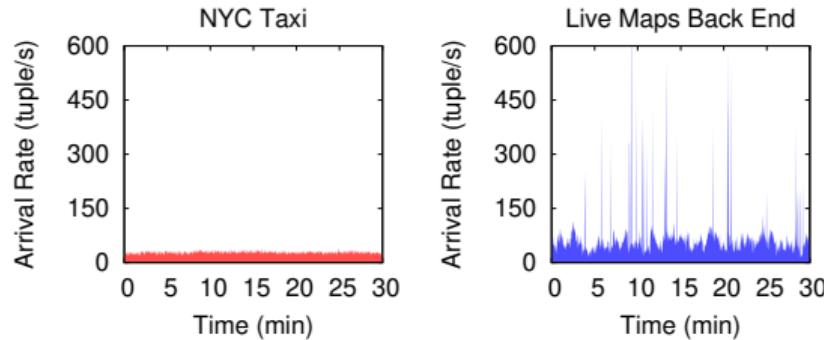
# Issues with Thresholds: Oscillation

- ▶ If thresholds are too close, frequent oscillations may occur
- ▶ Oscillations negatively impact performance
  - ▶ System oscillates between under- and over-provisioning
  - ▶ Scaling may introduce additional overhead too
- ▶ Possible workaround: **cooldown** (or, **calm**) period
  - ▶ Auto-scaler inhibited for a short period after every scaling action



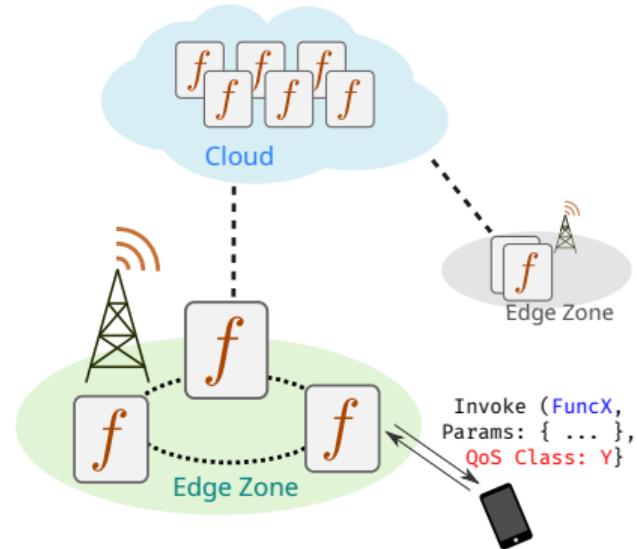
# Issues with Thresholds: Bursts

- ▶ Scaling conditions are usually evaluated over short-medium time periods (e.g., seconds, minutes)
- ▶ Not all workloads can be characterized looking at average metrics over such time windows
- ▶ These workloads lead to similar average utilization over 1-minute windows, but they are profoundly different due to [bursts \[Russo Russo et al. 2021\]](#)



# Example: Edge-Cloud Computation Offloading

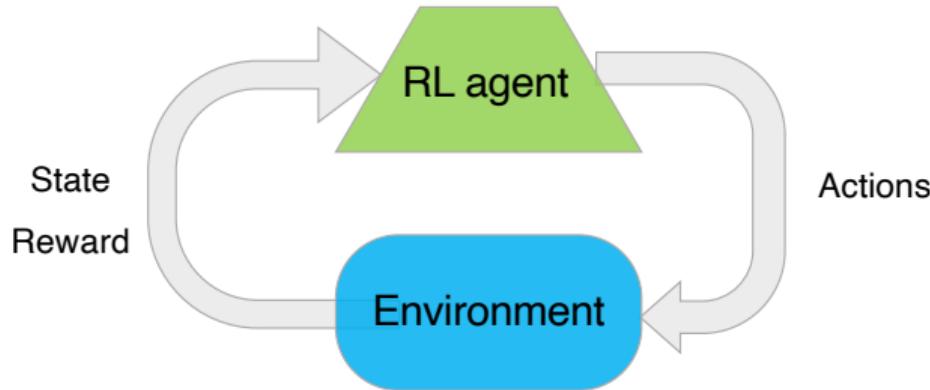
- ▶ Serverless functions invoked on Edge nodes
- ▶ Nodes can **offload** execution on neighboring nodes or to the Cloud
- ▶ Must guarantee maximum response time for functions
- ▶ Different resource cost for execution
- ▶ Traditional approach: greedy heuristics



# Reinforcement Learning

- ▶ Supervised learning
- ▶ Unsupervised learning
- ▶ Reinforcement learning
  - ▶ Goal-directed learning
  - ▶ Learning from interaction with an environment, rather than from examples
- ▶ Learning from interaction is probably the first idea to occur when thinking to the nature of learning

# Reinforcement Learning: Overview



- ▶ Sequential decision-making
- ▶ Agent interacts with an environment
  - ▶ Agent perceives the **state** of the environment
  - ▶ Agent performs **actions**
- ▶ Feedback in the form of **reward** (and new states)
- ▶ Goal: maximizing cumulated reward over the long run

# Reinforcement Learning: Key Ideas

- ▶ The learner is not told which actions to take (or which actions are the best)
- ▶ The agent must discover which actions yield the most reward by trying them ([trial-and-error](#))
- ▶ Often, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards ([delayed rewards](#))
- ▶ These two characteristics are the most important distinguishing features of RL

# A Bit of History

- ▶ Modern RL emerged around 1980's as two research trends intertwined
  - ▶ Learning from trial and error (originated in the psychology of animal learning)
  - ▶ Optimal control of stochastic systems (usually through dynamic programming)
- ▶ Consistent advancements through 80's and 90's
- ▶ New wave of popularity (and many new applications) with **deep** RL, after 2018

## Example: Tic-Tac-Toe

- ▶ **State:** representation of the board (3x3 matrix)
- ▶ **Actions:** available cells to mark
- ▶ **Reward:** 1 for a winning move, 0 otherwise

X	O	O
O	X	X
		X

# Example: AlphaZero by DeepMind

- ▶ Software able to play Go, Chess and Shogi
  - ▶ Board games with huge number of legal positions (i.e., state space)
  - ▶ Number of legal board positions in Go approximately  $2 \times 10^{170}$ , far greater than the number of atoms in the observable universe
- ▶ Trained via self-play and advanced deep RL techniques
- ▶ Superhuman level of play with 24-hour training
- ▶ First presented in 2017; in 2019 MuZero, generalization to play Atari games and other board games without prior rule knowledge



Paper: <https://arxiv.org/abs/1712.01815>

## Example: AlphaDev by DeepMind

- ▶ Announced in 2023<sup>1</sup>
- ▶ RL used to develop new C++ sorting algorithm, now accepted in the standard library
- ▶ 70% faster on short sequences (2-3 items), 1.7% faster on long sequences
- ▶ State: instructions generated so far and state of the CPU
- ▶ Actions: assembly instructions to add
- ▶ Reward: based on sorting correctness and efficiency

---

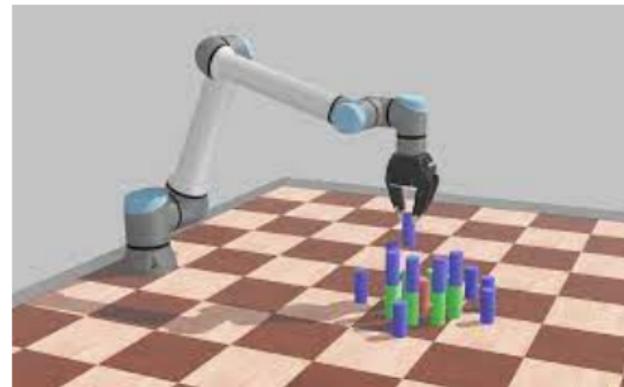
<sup>1</sup><https://www.deepmind.com/blog/alphadev-discovers-faster-sorting-algorithms>

# Example: deep RL agent playing Breakout

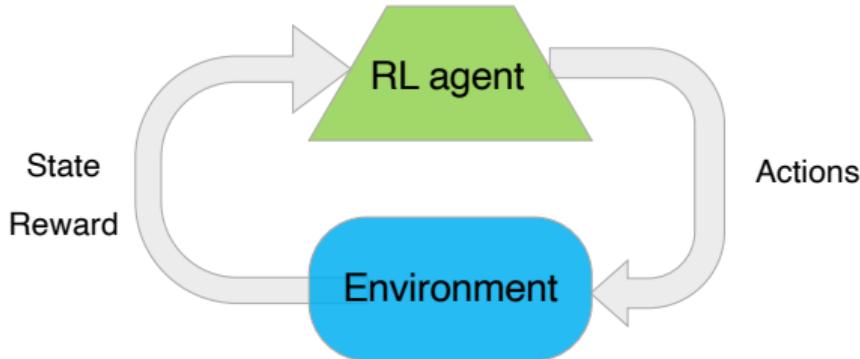
<https://www.youtube.com/watch?v=TmPfTpjtdgg>

# Other Examples

- ▶ Autonomous vehicles
- ▶ Robot control
- ▶ Trading
- ▶ Autonomous network and computer systems
- ▶ Videogames
- ▶ ...



# Reinforcement Learning



- ▶ Agent, environment, actions, state, rewards, ...
- ▶ Modeling depends on the specific task
  - ▶ e.g., autonomous car uses different state information compared to chess player
- ▶ The problem tackled by RL is formally described as a **Markov Decision Process (MDP)**
- ▶ A mathematical framework to model sequential decision making, in situations where outcomes are partly random

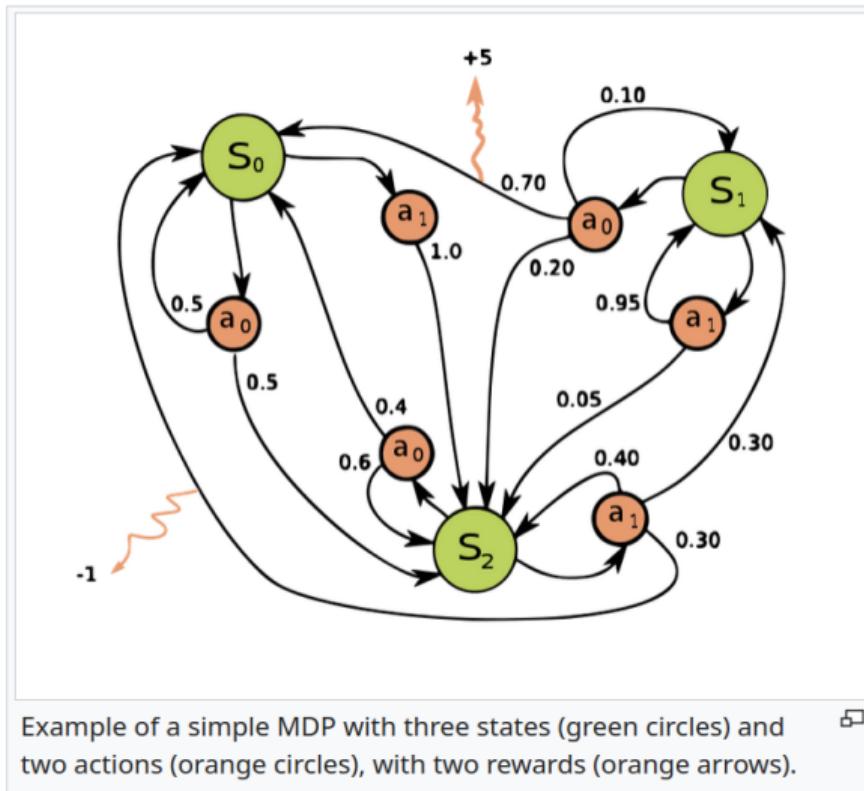
# Markov Decision Process (MDP)

- ▶ Extension of discrete-time [Markov chains](#)
  - ▶ “A stochastic model describing a sequence of possible events, occurring at discrete time steps, in which the probability of each event depends only on the state attained in the previous event.”
- ▶ At each time step  $t$ , the process is in some state  $s_t$
- ▶ The agent chooses an action  $a_t$  among those available in state  $s_t$ 
  - ▶ e.g., robot observes current position and decides direction to move; some directions might be blocked by obstacles

## Markov Decision Process (2)

- ▶ Following  $a_t$ , the process moves to (random) state  $s_{t+1}$ 
  - ▶ e.g., autonomous drone chooses an action to reduce altitude; actual outcome may depend on (unpredictable) wind speed
- ▶ Agent receives a **reward**  $r_t$  (or, equivalently, pays a **cost**)
  - ▶ e.g., robot may get a reward for reaching its final destination
  - ▶ e.g., chess player rewarded at the end of a match

# Example



# Markov Decision Process (3)

What defines an MDP?

- ▶  $\mathcal{S}$ : a (finite) set of states
- ▶  $\mathcal{A}$ : a (finite) set of actions
- ▶  $p$ : state transition probabilities

$$p(s'|s, a) = P[s_{t+1} = s' | s_t = s, a_t = a]$$

- ▶  $r$ : reward function
  1.  $r(s, a) = E[r_t | s_t = s, a_t = a]$
  2.  $r(s, a, s') = E[r_t | s_t = s, a_t = a, s_{t+1} = s'] \longrightarrow r(s, a) = \sum_{s'} p(s'|s, a)r(s, a, s')$

# Markov Property

“The future is independent of the past given the present”

## Definition

A state  $s_t$  is **Markov** if and only if

$$P[s_{t+1}|s_1, \dots, s_t] = P[s_{t+1}|s_t]$$

- ▶ The state captures all relevant information from the history
- ▶ i.e., the state is a sufficient statistic of the future

# Markov Property and MDPs

Within the MDP framework:

## Definition

$$P[s_{t+1}, r_t | s_1, a_1, \dots, s_t, a_t] = P[s_{t+1}, r_t | s_t, a_t]$$

- ▶ The current state **and action** are a sufficient statistic of the future

# Modeling the Task

- ▶ The MDP framework is flexible and can be applied to many different problems
- ▶ However, identifying a suitable model for the task at hand can be challenging
- ▶ Which [states, actions, rewards?](#)

# Modeling States, Actions and Rewards

- ▶ **Time steps** need not refer to fixed intervals of real time
  - ▶ e.g., arbitrary successive stages of decision making
- ▶ **Actions**: low-level controls (e.g., voltages applied to the motors of a robot arm), or high-level decisions (e.g., whether or not to reach a certain location)
- ▶ Similarly, **states** can be determined by low-level sensations (e.g., sensor readings), or they can be high-level and abstract (e.g., symbolic descriptions of objects in a room)
- ▶ **Rewards**: immediately identifiable (e.g., monetary gain of an automated trader), or more abstract (e.g., rewarding a software system for completing a job within the deadline)

# Modeling States, Actions and Rewards (2)

- ▶ When defining an MDP-based model of the task, it is important to avoid violating the Markov property

## Example

**(Simplified) self-driving vehicle.** State  $s = (x, y, z)$  comprises vehicle 3D coordinates (assuming constant speed). Only available actions are “turn right”, “turn left”, “go straight”.

Turning right and then left (or left and then right) causes the agent to lose control of the vehicle. We decide to assign a negative reward when conflicting actions are chosen in consecutive time steps. **OK?**

# Modeling the Environment

- ▶ The boundary between agent and environment is typically not the same as the physical boundary of a robot's or animal's body
- ▶ Usually, the boundary is drawn closer to the agent than that
- ▶ Anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of its environment

## Objective (informal)

- ▶ Informally, the goal of the agent is maximizing the amount of reward it receives
  - ▶ not immediate reward, but cumulative reward in the long run
- ▶ The idea of using a reward signal to formalize the idea of a goal is a distinguishing feature of RL

# Objective: Episodic Tasks

- ▶ Let's consider an **episodic task**, where the agent-environment interaction naturally terminates at some final time step  $T$ 
  - ▶  $T$  is a random variable
  - ▶ e.g., the end of a chess match
  - ▶ e.g., the time a robot reaches its destination or runs out of battery
- ▶ Final state  $s_T$  is called **terminal state**
- ▶ At time  $t$ , we aim to maximize the **expected return**  $G_t$

$$G_t = r_t + r_{t+1} + \dots + r_T$$

# Objective: Continuing Tasks

- ▶ In many cases the agent-environment interaction does not break naturally into identifiable episodes, but goes on continually without limit
  - ▶ e.g., a robot with a long life span
  - ▶ e.g., an agent managing VMs in a data center
  - ▶ e.g., the control system of RL-based traffic lights
- ▶ We could still consider the expected return, with  $T = \infty$
- ▶ But we could easily get infinite returns!

## Objective: Continuing Tasks (2)

- ▶ In this scenario, the agent maximizes the expected discounted return
- ▶ We introduce a discount factor  $\gamma \in [0, 1]$ 
  - ▶  $\gamma$  weights future rewards

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

- ▶  $\gamma = 0$ : a myopic agent
- ▶  $\gamma = 1$ : no discounting
- ▶ Common values are 0.9, 0.99

# Unified Notation

- ▶ We adopt a unified notation to cope with both episodic and continuing tasks
  - ▶ the unified notation is basically the one used for continuing tasks
  - ▶ 2 issues to fix for episodic tasks
- ▶ For episodic tasks, we should refer to state “at time  $i$  of episode  $j$ ”  $s_{i,j}$
- ▶ In practice, we are (almost) never interested in distinguishing between episodes, so we simply write  $s_i$ 
  - ▶ we either refer to a single episode or discuss things valid for any episode
- ▶ The return in episodic tasks is defined in terms of a final time step  $T$
- ▶ We can assume that final states in episodic tasks are **absorbing states** (no transitions to other states) that do not generate any reward
- ▶ Therefore, the objective can be expressed as a (discounted) infinite sum

## Reward vs Cost

- ▶ We can either maximize the expected reward or minimize the expected cost

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

- ▶ The two formulations are equivalent;
- ▶ You can easily switch between them by setting

$$r(s, a) = -c(s, a)$$

- ▶ In the following, we will often refer to costs; keep in mind this equivalence

## Definition

A **policy**  $\pi$  is a distribution over actions for every state  $s$

$$\pi(a|s) = P(A_t = a|S_t = s)$$

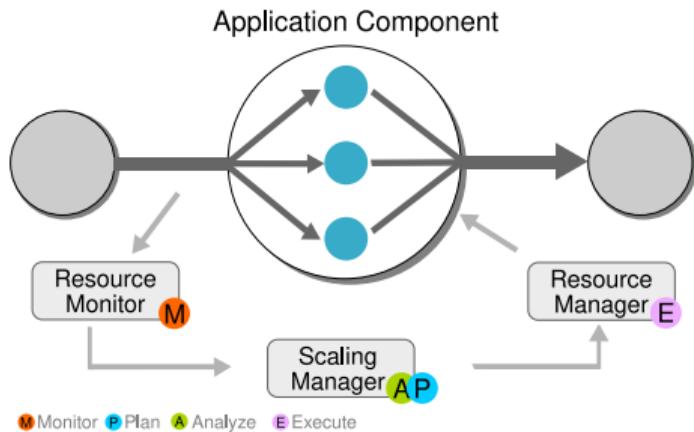
- ▶ A policy fully defines agent's behavior
- ▶ MDP policies depend on the current state only
- ▶ RL defines how policies change based on experience
- ▶ Special case: **deterministic policy**

$$\pi : \mathcal{S} \rightarrow \mathcal{A}$$

## Example: Deterministic Policy

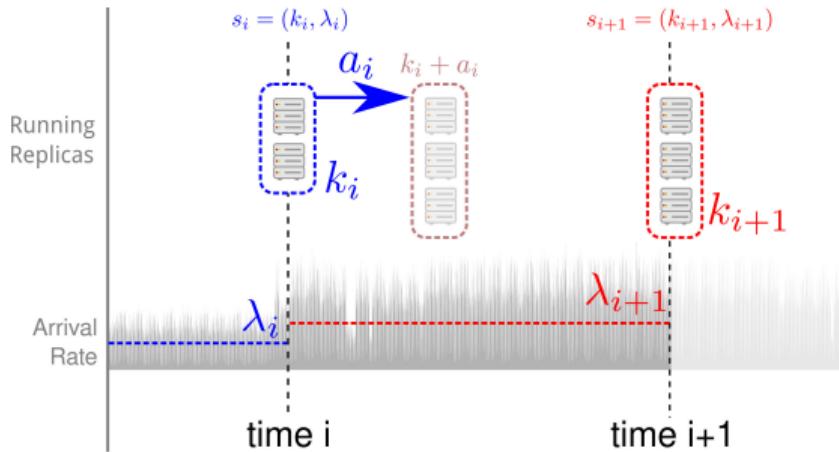
<i>State</i>	<i>Action</i>
$s_1$	$a_1$
$s_2$	$a_1$
$s_3$	$a_2$
$s_4$	$a_1$

# Example: Cloud Auto-scaling



- ▶ We periodically make a decision about scaling in/out an app component (a thread, a VM, a container, ...)
- ▶ We are concerned with 3 objectives:
  - ▶ Monetary resource cost (or, resource usage in general)
  - ▶ Performance requirements (e.g., max response time)
  - ▶ Scaling overhead

# Auto-scaling: MDP formulation



- ▶ State at time slot  $i$ :  $s_i = (k_i, \lambda_i)$ 
  - ▶  $k_i$  component parallelism
  - ▶  $\lambda_i$  avg. arrival rate (of requests, jobs, data, ...)
- ▶ Action at time slot  $i$ :  $a_i \in \{0, +1, -1\}$

# MDP Model: Transition Probabilities

- ▶ State of the system  $s = (k, \lambda)$ 
  - ▶  $1 \leq k \leq K^{\max}$  Component parallelism
  - ▶  $\lambda$  avg. input rate
    - ▶  $\lambda$  is discretized, i.e.,  $\lambda_i \in \{0, \Delta\lambda, 2\Delta\lambda, (L-1)\Delta\lambda\}$
    - ▶  $\Delta\lambda$  quantization step size,  $L$  number of discrete values
- ▶ Available actions  $\mathcal{A} = \{-1, 0, +1\}$
- ▶ Transition probabilities  $p(s'|s, a) = p((k', \lambda')|(k, \lambda), a)$

$$\begin{aligned} p(s'|s, a) &= P[s_{t+1} = (k', \lambda')|s_t = (k, \lambda), a_t = a] = \\ &= \begin{cases} P[\lambda_{t+1} = \lambda'|\lambda_t = \lambda] & k' = k + a \\ 0 & \text{otherwise} \end{cases} = \\ &= \mathbb{1}_{\{k'=k+a\}} P[\lambda_{t+1} = \lambda'|\lambda_t = \lambda] \end{aligned}$$

# MDP Model: Cost Function

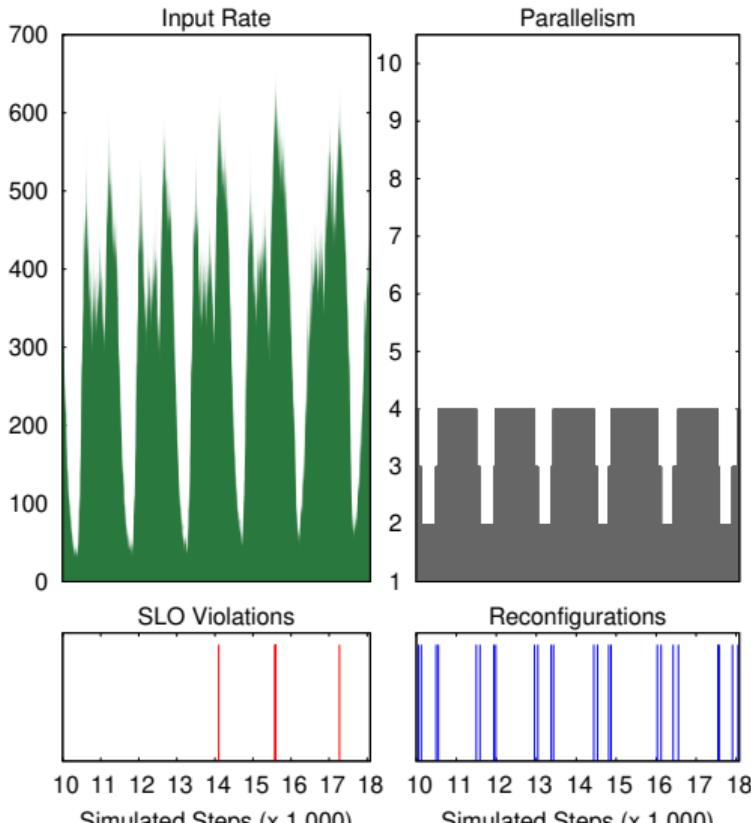
$$c(s, a, s') = w_{res} \frac{k + a}{K^{max}} + w_{perf} \mathbb{1}_{\{R(s, a, s') > R^{max}\}} + w_{rcf} \mathbb{1}_{\{a \neq 0\}}$$

Resource Cost      Performance      Reconfig.

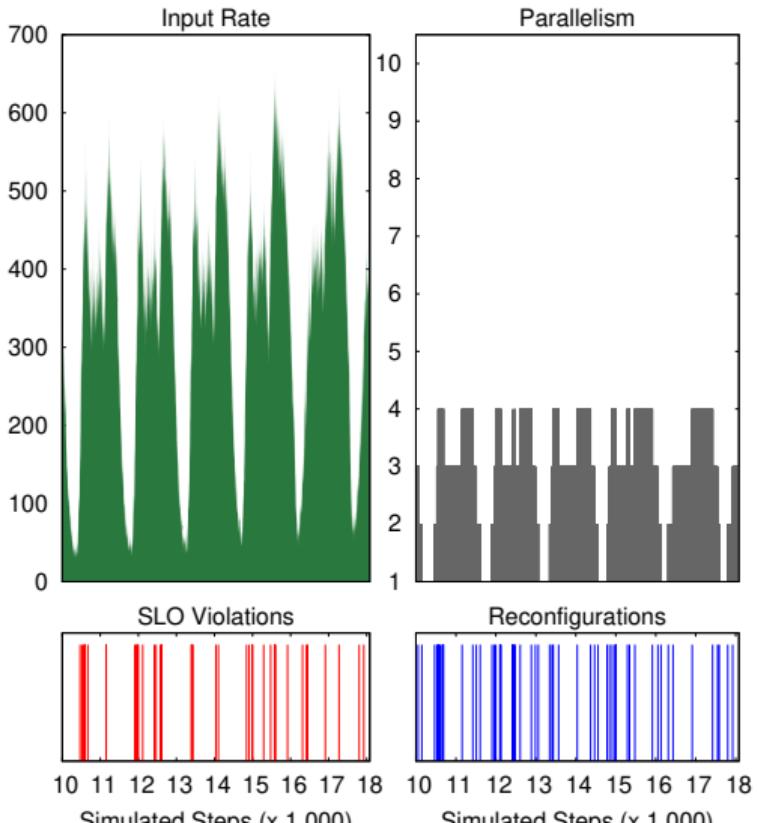
- ▶  $w_{res} + w_{perf} + w_{rcf} = 1$ ,  $w_x \geq 0$ ,  $x \in \{res, perf, rcf\}$
- ▶  $R(s, a, s')$ : performance index, e.g, response time
- ▶  $R^{max}$ : reference performance value
- ▶ We want to minimize  $\sum_{t=0}^{\infty} \gamma^t c(s_t, a_t, s_{t+1})$ ,  $\gamma \in [0, 1)$

# Auto-Scaling: Trading-off Objectives

$$w_{perf} = 0.6, w_{res} = w_{rcf} = 0.2$$



$$w_{res} = 0.6, w_{perf} = w_{rcf} = 0.2$$



# Value Function

Informally, a value function is an estimate of future rewards

- ▶ can be used to evaluate how good/bad states and/or actions are
- ▶ and therefore to select actions e.g. in a greedy way

State/Action	$a_1$	$a_2$	$a_3$
$s_1$	10	5	13
$s_2$	8	6	14
$s_3$	6	9	6
$s_4$	5	8	6
$s_5$	4	8	7
$s_6$	10	5	9
$s_7$	20	9	15

$s$	$\pi(s)$
$s_1$	$a_3$
$s_2$	$a_3$
$s_3$	$a_2$
$s_4$	$a_2$
$s_5$	$a_2$
$s_6$	$a_1$
$s_7$	$a_1$

## Value Function (2)

- ▶ The **state-value function** of state  $s$  under policy  $\pi$ ,  $v_\pi(s)$  is the expected return starting in  $s$  and following  $\pi$  thereafter

$$v_\pi(s) = \mathbb{E}_\pi [G_t | S_t = s] \quad \forall s \in \mathcal{S}$$

- ▶ where  $\mathbb{E}_\pi [\cdot]$  denotes the expected value of a random variable given that the agent follows policy  $\pi$

# Action Value Function

- ▶ The **action value function** of state  $s$  and action  $a$  under policy  $\pi$ ,  $Q_\pi(s, a)$  is the expected return starting in  $s$ , choosing action  $a$  and following  $\pi$  thereafter

$$Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a] \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$$

- ▶ a.k.a. **Q function**

# Monte Carlo Methods

- ▶ Simple approach to estimate the value functions from experience
- ▶ Monte Carlo methods: averaging over many random samples of actual returns
- ▶ Agent follows policy  $\pi$  and maintains an average, for each state encountered, of the actual returns that have followed that state
- ▶ Average will converge to the state value  $v_\pi(s)$ , as the number of times that state is encountered approaches infinity
- ▶ Of course, if there are very many states, then it may not be practical to keep separate averages for each state individually
  - ▶ function approximation might be used

# Bellman Equation

The **action value function** can be decomposed into two parts:

- ▶ immediate reward
- ▶ discounted rewards from successor state  $S_{t+1}$

$$\begin{aligned} Q_\pi(s, a) &= E_\pi[G_t | S_t = s, A_t = a] \\ &= E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \dots | S_t = s, A_t = a] \\ &= E_\pi[R_t + \gamma (R_{t+1} + \gamma R_{t+2} \dots) | S_t = s, A_t = a] \\ &= E_\pi[R_t + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= r(s, a) + \gamma E_\pi[G_{t+1} | S_t = s, A_t = a] \end{aligned}$$

**Bellman equation:**

$$Q_\pi(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) Q_\pi(s', \pi(s'))$$

## Bellman Equation (2)

The value function can be similarly decomposed into two parts:

- ▶ immediate reward  $r_t$
- ▶ discounted reward from successor state  $V(S_{t+1})$

$$\begin{aligned}V_\pi(s) &= E_\pi[G_t | S_t = s] \\&= E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \dots | S_t = s] \\&= E_\pi[R_t + \gamma (R_{t+1} + \gamma R_{t+2} \dots) | S_t = s] \\&= E_\pi[R_t + \gamma G_{t+1} | S_t = s]\end{aligned}$$

Bellman equation:

$$V_\pi(s) = r(s, \pi(s)) + \gamma \sum_{s'} p(s'|s, \pi(s)) V_\pi(s')$$

## Bellman Equation (3)

- ▶ If dealing with non-deterministic policies ...

$$V_{\pi}(s) = r(s, \pi(s)) + \gamma \sum_{s'} p(s'|s, \pi(s)) V_{\pi}(s')$$

- ▶ the equation becomes:

$$V_{\pi}(s) = \sum_a \pi(a|s) r(s, a) + \gamma \sum_{s'} p(s'|s, a) V_{\pi}(s')$$

# Optimal Policies

- ▶ Solving a RL task means, roughly, finding a policy that achieves a lot of reward over the long run
- ▶ For finite MDPs, we can precisely define an optimal policy
- ▶ Value functions define a partial ordering over policies
- ▶ A policy  $\pi$  is defined to be better than or equal to a policy  $\pi'$  if its expected return is greater than or equal to that of  $\pi'$  for all states

$$\pi \geq \pi' \text{ iff } v_\pi(s) \geq v_{\pi'}(s) \forall s \in \mathcal{S}$$

## Optimal Policies (2)

- ▶ There is always at least one policy that is better than or equal to all other policies (**optimal policy**)
- ▶ There may be more than one, and we denote all the optimal policies by  $\pi^*$
- ▶ They share the same state-value function, called the **optimal state-value function**, denoted  $v^*$ , and defined as

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

# Optimal Value Function

## Optimal state value function

$V^*(s)$  is the minimum value function over all policies

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

## Optimal action value function

$Q^*(s; a)$  is the maximum action-value function over all policies

$$Q^*(s, a) = \max_{\pi} Q_{\pi}(s, a)$$

# Bellman Optimality Equations

$$V_{\pi}(s) = r(s, \pi(s)) + \gamma \sum_{s'} p(s'|s, \pi(s)) V_{\pi}(s')$$

- ▶ The Bellman Equation holds for the optimal value function as well
- ▶ We can drop the reference to any specific policy and obtain the [Bellman Optimality Equation](#)

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

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

## Bellman Optimality Equations (2)

For the action value function:

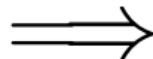
$$Q^*(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \max_{a'} Q^*(s', a')$$

# Optimal Policy

Given  $Q^*(s, a)$  the optimal action when the system is in state  $s$  is:

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

State	$a_1$	$a_2$	$a_3$
$s_1$	10	5	3
$s_2$	8	6	4
$s_3$	6	5	6
$s_4$	5	4	6
$s_5$	4	8	7
$s_6$	1	5	9
$s_7$	0	9	15



$\pi^*(s)$
$a_1$
$a_1$
$a_1$
$a_3$
$a_2$
$a_3$
$a_3$

## How to compute $V^*$ ?

- ▶ If we know the optimal value function, we have an optimal policy!
- ▶ **But...** how to compute the optimal value function??
- ▶ We already mentioned Monte Carlo methods
- ▶ An alternative approach relies on dynamic programming

# Value Iteration

## Bellman Equation

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \max_{a'} Q^*(s', a')$$

- ▶ Suppose we know the solution to subproblems  $Q^*(s', a')$
- ▶  $Q^*(s, a)$  can be computed by one-step lookahead

$$Q^*(s, a) \leftarrow r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \max_{a'} Q^*(s', a')$$

- ▶ The idea is to apply these updates iteratively
- ▶ Proven to converge to  $Q^*$  for any finite MDP (see, Sutton-Barto, Ch. 4)

# Value Iteration: Algorithm

## Value Iteration

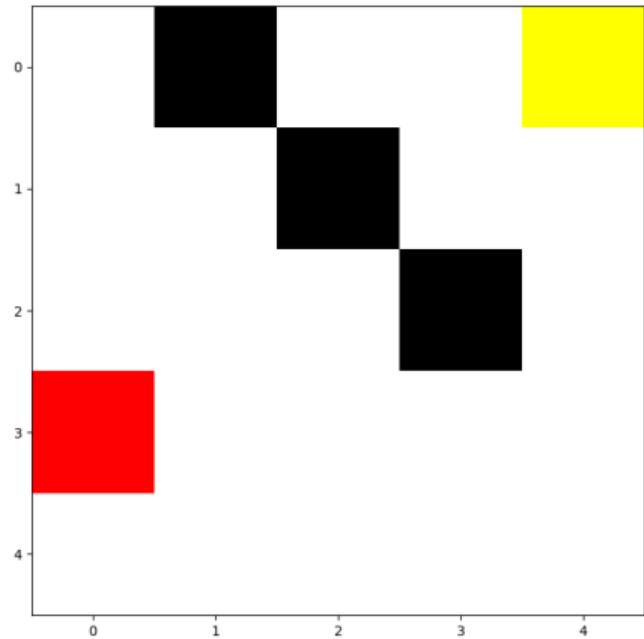
```
1  $i \leftarrow 0$ 
2  $Q_i(s, a) \leftarrow 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$ 
3 repeat
4   forall  $s \in \mathcal{S}$  do
5     forall  $a \in \mathcal{A}(s)$  do
6        $Q_{i+1}(s, a) \leftarrow r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) \max_{a' \in \mathcal{A}(s')} Q_i(s', a')$ 
7     end
8   end
9    $i \leftarrow i + 1$ 
10 until  $\max_{s,a} |Q_i(s, a) - Q_{i-1}(s, a)| < \epsilon$ 
11  $\pi^*(s) = \arg \max_a Q_i(s, a), \forall s \in \mathcal{S}$ 
```

## Value Iteration - Alternative Algorithm

```
1  $i \leftarrow 0$ 
2  $Q_i(s, a) \leftarrow 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$ 
3  $V_i(s) \leftarrow 0, \forall s \in \mathcal{S}$ 
4 repeat
5   forall  $s \in \mathcal{S}$  do
6     forall  $a \in \mathcal{A}(s)$  do
7        $| Q_{i+1}(s, a) \leftarrow r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V_i(s)$ 
8     end
9      $V_{i+1}(s) = \max_{a' \in \mathcal{A}(s)} Q_{i+1}(s, a')$ 
10    end
11     $i \leftarrow i + 1$ 
12 until  $\max_{s,a} |Q_i(s, a) - Q_{i-1}(s, a)| < \epsilon$ 
13  $\pi^*(s) = \arg \max_a Q_i(s, a), \forall s \in \mathcal{S}$ 
```

## Example: Maze

- ▶ Consider a  $S \times S$  grid
- ▶ Episodes start with agent randomly located in a cell in the first column
- ▶ Goal: reaching target cell  $(1, S)$
- ▶ Some cells are blocked
- ▶ Some cells are slippery: when entering, the agent has a probability  $p_{slip}$  of slipping one cell ahead along the current direction

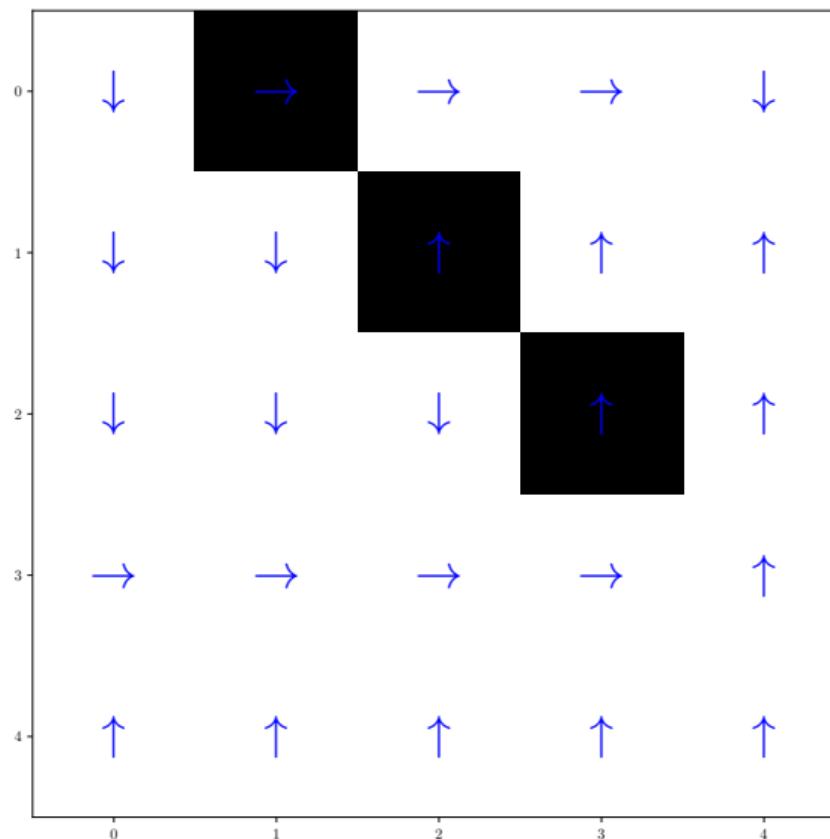


## Example: Maze (2)

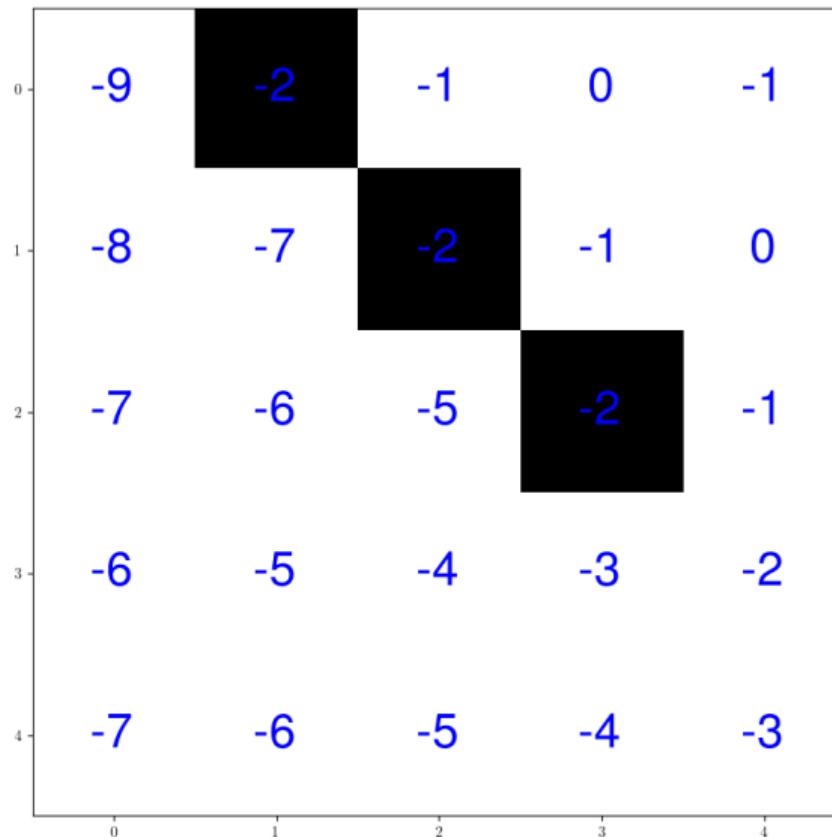
- ▶ State:  $s = (x, y)$
- ▶ Actions:  $a \in \{(0, 1), (0, -1), (1, 0), (-1, 0)\}$
- ▶ Reward:
  - ▶ 0 for entering the goal cell
  - ▶  $-M$  for exiting the grid or crashing into a blocked cell ( $M \gg 1$ )
  - ▶ -1 otherwise

 maze.py (--agent mdp)

# Maze: Optimal Policy



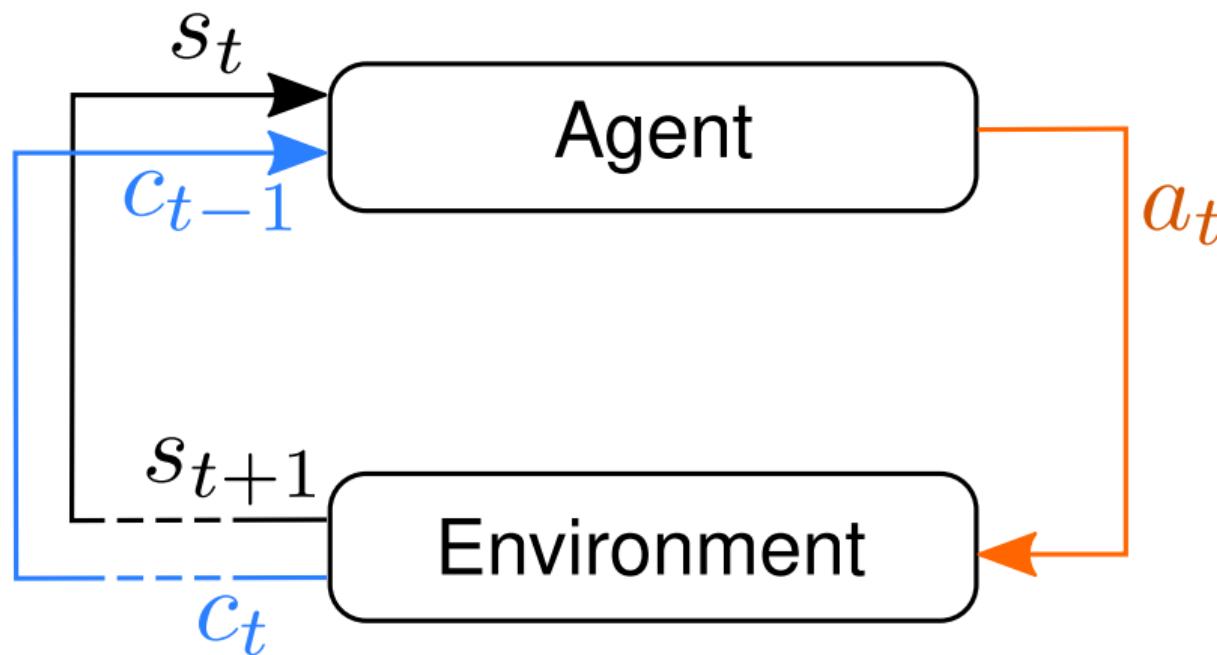
# Maze: Optimal Value Function



# MDP Resolution

- ▶ We can use the Value Iteration algorithm to solve the MDP
  - ▶ i.e., finding the optimal policy
- ▶ Is this enough?
- ▶ Unfortunately, solving the MDP requires exact and complete knowledge of the underlying model
  - ▶ state transition probabilities
  - ▶ reward/cost function
- ▶ In practice, we don't have such information!
- ▶ Moreover, Dynamic programming approaches can become computationally unfeasible for very large state-action spaces

# Reinforcement Learning



- ▶ RL aims to learn the optimal policy through interaction and evaluative feedback

# Model-free vs Model-based RL

- ▶ **Model-free** RL: no model of the environment is available or used; the optimal policy is learned through experience only
- ▶ **Model-based** RL: a (possibly partial) model of the environment is available and used to derive the optimal policy
  - ▶ a partial model can boost learning speed
  - ▶ RL may also be used in presence of a complete model instead of VI; e.g., with a large number of rarely visited states VI would unnecessarily run for a long time!
  - ▶ You may also try to learn the model online and use it to compute a policy

# Value-based vs Policy-based RL

- ▶ **Value-based** RL: aims to learn the optimal value function through experience; the policy is derived from it
  - ▶ Simplest RL algorithms belong to this group
- ▶ **Policy-based** RL: aims to directly learn the optimal policy through experience; no explicit computation/learning of the value function
- ▶ **Hybrid** approaches: e.g., the Actor-Critic framework

# Simple Value-based RL Algorithm

## A simple RL algorithm

```
1  $t \leftarrow 0$ 
2 Initialize  $Q$ 
3 Loop
4   |  $t \leftarrow t + 1$ 
5 EndLoop
```

# Q-learning

- ▶ Proposed by Chris Watkins in early 90's ([Watkins and Dayan 1992](#))
- ▶ One of the most known (and simplest) RL algorithms
- ▶ Proven to converge to the optimal policy under mild assumptions
  - ▶ ...after  $n$  steps, with  $n \rightarrow \infty$ , and every state visited an infinite number of times
- ▶ Very slow to convergence in practice!

# Q-learning: Action Selection

- ▶ How to choose an action at every time step?
- ▶ Exploration vs Exploitation dilemma
- ▶ Exploitation: using available knowledge to maximize reward
  - ▶ choose the “best” action, i.e.,  $a_t = \arg \max_a Q(s_t, a)$
- ▶ Exploration: discovering more information about the environment
  - ▶ choose other actions to learn more about the environment

Q-learning converges only if all state-action pairs are visited an infinite number of times as  $t \rightarrow \infty$

- ▶ you can't exploit all the time
- ▶ you can't explore all the time

## $\epsilon$ -Greedy Exploration

- ▶ Popular approach for the exploration-exploitation dilemma
- ▶ With probability  $1 - \epsilon$  choose the greedy action  $a^* = \arg \max_{a \in \mathcal{A}} Q(s, a)$
- ▶ With probability  $\epsilon$  choose an action at random
  
- ▶ Improvement:  $\epsilon$ -greedy with decaying  $\epsilon$  (similar to decaying learning rate in SGD)

# Softmax Action Selection

- ▶ Alternative to the  $\epsilon$ -greedy strategy
- ▶ All actions assigned non-zero probability of being chosen
- ▶ Action  $a \in \mathcal{A}$  is selected with probability

$$\pi(a|s) = \frac{\exp(Q(s, a)/\tau)}{\sum_{a' \in \mathcal{A}} \exp(Q(s, a')/\tau)}$$

- ▶  $\tau$  is the “temperature”
  - ▶ Small  $\tau$  leads to greedy behavior
  - ▶ Large  $\tau$  leads to random action selection
  - ▶ You usually start with a large temperature value and let it decay

# Q-learning: Updating Q

With known model, we can compute  $Q$  iteratively using:

$$Q(s, a) \leftarrow r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \max_{a'} Q(s', a')$$

Q-learning uses *point estimates* on experience  $\{s_t, a_t, r_t, s_{t+1}\}$ :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left[ r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

Learning Rate      Target

# Q-learning: Algorithm

## Q-learning

- 1  $t \rightarrow 0$
- 2 Initialize  $Q$  (e.g., zero-initialized)
- 3 **Loop**
  - 4 choose  $a_t$  (e.g.,  $\epsilon$ -greedy or softmax selection)
  - 5 observe next state  $s_{t+1}$  and reward  $r_t$
  - 6 
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t [r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t)]$$
  - 7  $t \leftarrow t + 1$
- 8 **EndLoop**

## Example: Maze

▶ `python maze.py --agent qlearning --episodes N  
[-- plot_reward]`

 maze.py

 qlearning.ipynb

# SARSA

- ▶ Q-learning is an **off-policy** algorithm
  - ▶ It estimates the return for state-action pairs assuming a greedy policy is followed, despite the fact that it's not following a greedy policy (e.g.,  $\epsilon$ -greedy)
- ▶ **SARSA**: **on-policy** algorithm similar to Q-learning
- ▶ The same policy is used to choose next action and to update  $Q$

# SARSA: Algorithm

## SARSA

- 1  $t \rightarrow 0$
- 2 Initialize  $Q$  (e.g., zero-initialized)
- 3 choose  $a_t$  (e.g.,  $\epsilon$ -greedy or softmax selection)
- 4 **Loop**
  - 5 observe next state  $s_{t+1}$  and reward  $r_t$
  - 6 choose  $a_{t+1}$  (e.g.,  $\epsilon$ -greedy or softmax selection)
  - 7 
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t [r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$
  - 8  $t \leftarrow t + 1$
- 9 **EndLoop**

# Dealing with Large State Spaces: Deep RL

# Issues with Tabular RL

- ▶ So far, we have considered **tabular** representations of the value function

<i>State/Action</i>	$a_1$	$a_2$	...
$s_1$	$Q(s_1, a_1)$	$Q(s_1, a_2)$	...
$s_2$	$Q(s_2, a_1)$	$Q(s_2, a_2)$	...
...	...		
$s_n$	$Q(s_n, a_1)$	$Q(s_n, a_2)$	...

- ▶ Not ideal as the state space grows...
- ▶ **Memory demand:**  $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$
- ▶ **No generalization**
- ▶ How to handle **continuous state spaces?**

# Value Function Approximation

Idea: using a **parametric approximation** of the value function

$$V_\pi(s) \approx \hat{V}(s, \mathbf{w}), \text{ or}$$

$$Q_\pi(s, a) \approx \hat{Q}(s, a, \mathbf{w})$$

- ▶  $\mathbf{w} \in \mathbb{R}^d$  is a vector of parameters
- ▶ We need to store  $\mathbf{w}$  instead of the Q table
  - ▶ Reduced memory demand if  $d < |S|$  ✓
- ▶ Potential generalization ✓
  - ▶ The experience gained in a state used to update  $\mathbf{w}$
  - ▶ A single update possibly impacts the value of several states!
  - ▶ Can deal with continuous state spaces ✓

## Value Function Approximation (2)

- ▶ How to choose a function  $\hat{Q}$ ?
- ▶ How to determine the value of  $w$ ?
- ▶ We search for a function and a vector  $w$  so as to approximate  $V$  (or  $Q$ ) "well"
- ▶ First of all, what does "well" means?

# Function Approximation: Objective

- ▶ A simple and natural choice is to minimize MSE:

$$J(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s) [V_\pi(s) - \hat{V}(s, \mathbf{w})]^2$$

- ▶  $\mu(s) \geq 0$  is a distribution over states
- ▶  $\mu(s)$  should reflect the importance or frequency of states

# Optimizing Parameters

We can compute parameters  $\mathbf{w}$  through gradient descent

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w}_t) = \\ &= \mathbf{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) \left[ V_{\pi}(s) - \hat{V}(s, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{V}(s, \mathbf{w})\end{aligned}$$

Two potential issues:

1. Summation over all states (may be expensive!)
2. We don't have the true values  $V_{\pi}(s)$ !

# Optimizing Parameters: Issue 1

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) [V_\pi(s) - \hat{V}(s, \mathbf{w})] \nabla_{\mathbf{w}} \hat{V}(s, \mathbf{w})$$

Gradient computed over all states...

Stochastic gradient descent

one (or few) samples  $(s_t, V_\pi(s_t))$  at each step

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [V_\pi(\mathbf{s}_t) - \hat{V}(\mathbf{s}_t, \mathbf{w})] \nabla_{\mathbf{w}} \hat{V}(\mathbf{s}_t, \mathbf{w})$$

# Optimizing Parameters: Issue 2

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) \left[ V_\pi(s) - \hat{V}(s, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{V}(s, \mathbf{w})$$

How to get exact values?

Stochastic semi-gradient descent:

we replace  $V_\pi(s_t)$  with a noisy approximation  $U_t$ , based on estimated value func.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[ U_t - \hat{V}(s_t, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{V}(s_t, \mathbf{w})$$

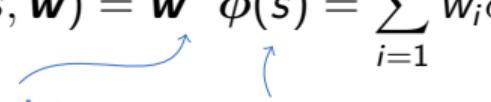
A possible approach (inspired by Q-learning):

$$U_t = r_t + \gamma \hat{V}(s_{t+1}, \mathbf{w}_t)$$

# Linear Function Approximation

The simplest possible approximation model:

$$\hat{V}(s, \mathbf{w}) = \mathbf{w}^T \phi(s) = \sum_{i=1}^d w_i \phi_i(s)$$



Weights                          Features

$\mathbf{w} \in \mathbb{R}^d$                        $\phi : \mathcal{S} \rightarrow \mathbb{R}^d$

Update rule becomes very simple:

$$\nabla_{\mathbf{w}} \hat{V}(s, \mathbf{w}) = \phi(s)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[ U_t - \hat{V}(s_t, \mathbf{w}_t) \right] \phi(s_t)$$

# Linear Function Approximation (2)

We have equivalent formulas for  $Q$ :

$$\hat{Q}(s, a, \mathbf{w}) = \mathbf{w}^T \phi(s, a) = \sum_{i=1}^d w_i \phi_i(s, a)$$

Weights                          Features  
 $\mathbf{w} \in \mathbb{R}^d$                $\phi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^d$

$$\nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w}) = \phi(s, a)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[ U_t - \hat{Q}(s_t, a_t, \mathbf{w}_t) \right] \phi(s_t, a_t)$$

??

# Q-learning + Linear FA

Recall Q-learning update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t [r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

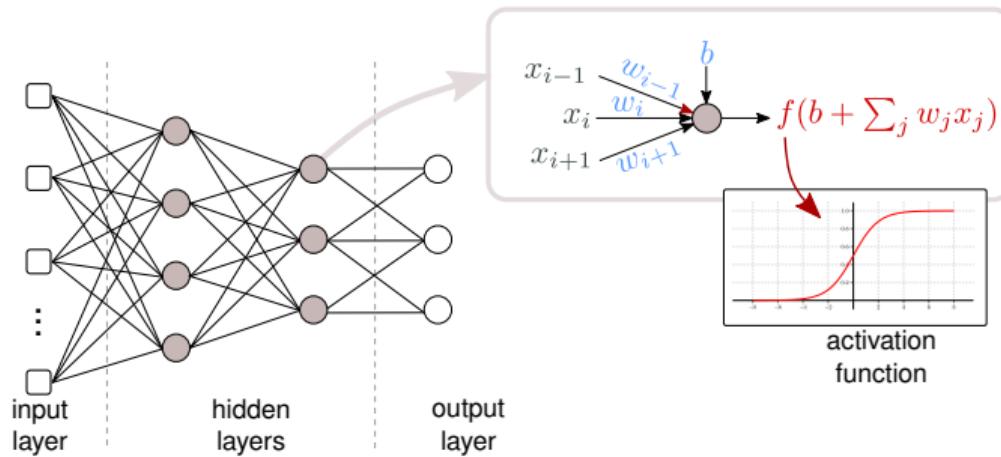
- 1  $t \rightarrow 0$
- 2 Initialize  $\mathbf{w}$
- 3 **Loop**
  - 4 choose action  $a_t$
  - 5 gather experience  $\langle s_t, a_t, r_t, s_{t+1} \rangle$
  - 6  $U_t \leftarrow r_t + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s_{t+1}, a', \mathbf{w}_t)$
  - 7  $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [U_t - \hat{Q}(s_t, a_t, \mathbf{w}_t)] \phi(s_t, a_t)$
  - 8  $t \leftarrow t + 1$
- 9 **EndLoop**

## (Linear) FA: Issues

- ▶ Linear FA+RL successfully applied on some tasks
- ▶ Nonlinear models (e.g., ANNs) have obtained significant results as well
  - ▶ e.g., [TD-Gammon](#) (1992)
- ▶ Efficacy of these approaches strongly depends on the [features](#) in use
  - ▶ how states (and actions) are represented
  - ▶ domain expertise necessary

# Deep RL

- ▶ We have seen that the key advancement enabled by DNNs is the ability of learning the features
- ▶ Idea: exploiting this ability to learn suitable features for state and action representation



# Deep Q Network

- ▶ First popular application of DNNs within RL in 2013
  - ▶ Mnih et al., “Playing Atari with Deep Reinforcement Learning”  
<https://www.cs.toronto.edu/%7Evmnih/docs/dqn.pdf>
- ▶ Task: playing Atari 2600 games
- ▶ Two key innovations:
  - ▶ DNN to approximate  $Q$  ([Deep Q Network](#))
  - ▶ [Experience Replay](#) buffer
- ▶ Learning algorithm adapted from Q-learning

## Example: Atari games

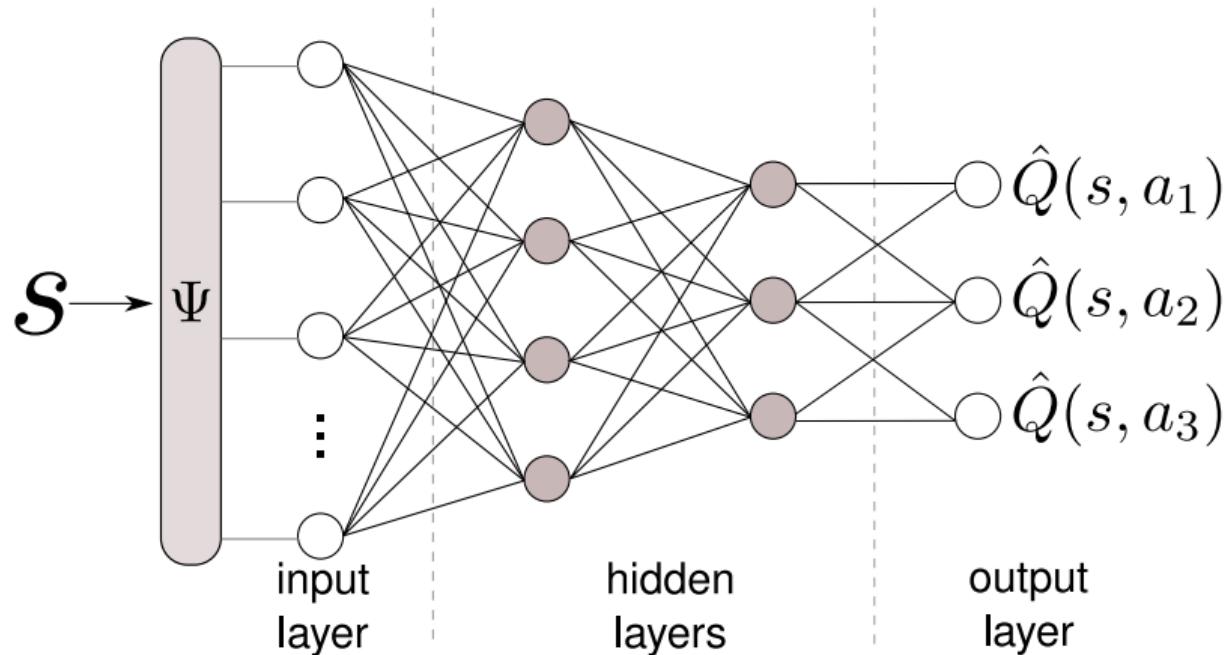


## Example: Atari games



# Deep Q Network

- ▶ **Input:** state  $s$  (possibly preprocessed)
- ▶ **Output:**  $\hat{Q}(s, a)$ , for every action  $a$



# Training

- ▶ NN training usually based on (large) training set
  - ▶ collection of examples  $(x_i, y_i)$
- ▶ To train a DQN we would need many examples  $(s_i, [Q(s_i, a_1) \cdots Q(s_i, a_n)]^T)$
- ▶ **Problem:** we don't have true examples of  $Q(s, a)$  to use!
  - ▶ agent only collects immediate rewards on-line
- ▶ We need to estimate  $Q$  on-line based on experience (as usual in RL)

## Training (2)

### Experience

$\langle s_t, a_t, s_{t+1}, r_t \rangle$

$\langle s_{t-1}, a_{t-1}, s_t, r_{t-1} \rangle$

$\langle s_{t-2}, a_{t-2}, s_{t-1}, r_{t-2} \rangle$

...

### Training Sample

$(s_t, a_t) \rightarrow r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a', \mathbf{w})$

$(s_{t-1}, a_{t-1}) \rightarrow r_{t-1} + \gamma \max_{a'} \hat{Q}(s_t, a', \mathbf{w})$

$(s_{t-2}, a_{t-2}) \rightarrow r_{t-2} + \gamma \max_{a'} \hat{Q}(s_{t-1}, a', \mathbf{w})$

- ▶ Naive idea: pick mini-batches of last  $b$  experience tuples and train the NN
  - ▶ i.e., at each iteration, train on most recent experience
- ▶ sequential observations likely correlated  $\textcolor{red}{X}$
- ▶ less recent experience possibly forgotten  $\textcolor{red}{X}$

# Experience Replay

- ▶ Smarter approach: **experience replay** buffer
- ▶ Circular FIFO buffer with capacity  $B > b$
- ▶ At each training iteration,  $b$  tuples drawn randomly from the buffer
- ▶ correlation between observations reduced/removed ✓
- ▶ if  $B$  is large, old observations are “seen” more than once ✓
  - ▶ improved data efficiency

# Deep Q-learning (DQL)

```
1 Initialize  $w$ 
2 Initialize empty buffer  $\mathcal{B}$ 
3  $i \leftarrow 0$ 
4 Loop
5   choose action  $a_i$ 
6   gather experience  $\langle s_i, a_i, r_i, s_{i+1} \rangle$  and add to  $\mathcal{B}$ 
7   sample minibatch of  $b$   $\langle s_j, a_j, r_j, s_{j+1} \rangle$  tuples from  $\mathcal{B}$ 
8    $y^{(j)} \leftarrow r_j + \gamma \max_{a'} \hat{Q}(s_{i+1}, a', w), j = 1, \dots, b$ 
9    $\mathcal{L}^{(j)} = (y^{(j)} - \hat{Q}(s_j, a_j, w))^2$  /* Loss */
10  update  $w$  using, e.g., SGD on the minibatch
11   $i \leftarrow i + 1$ 
12 EndLoop
```

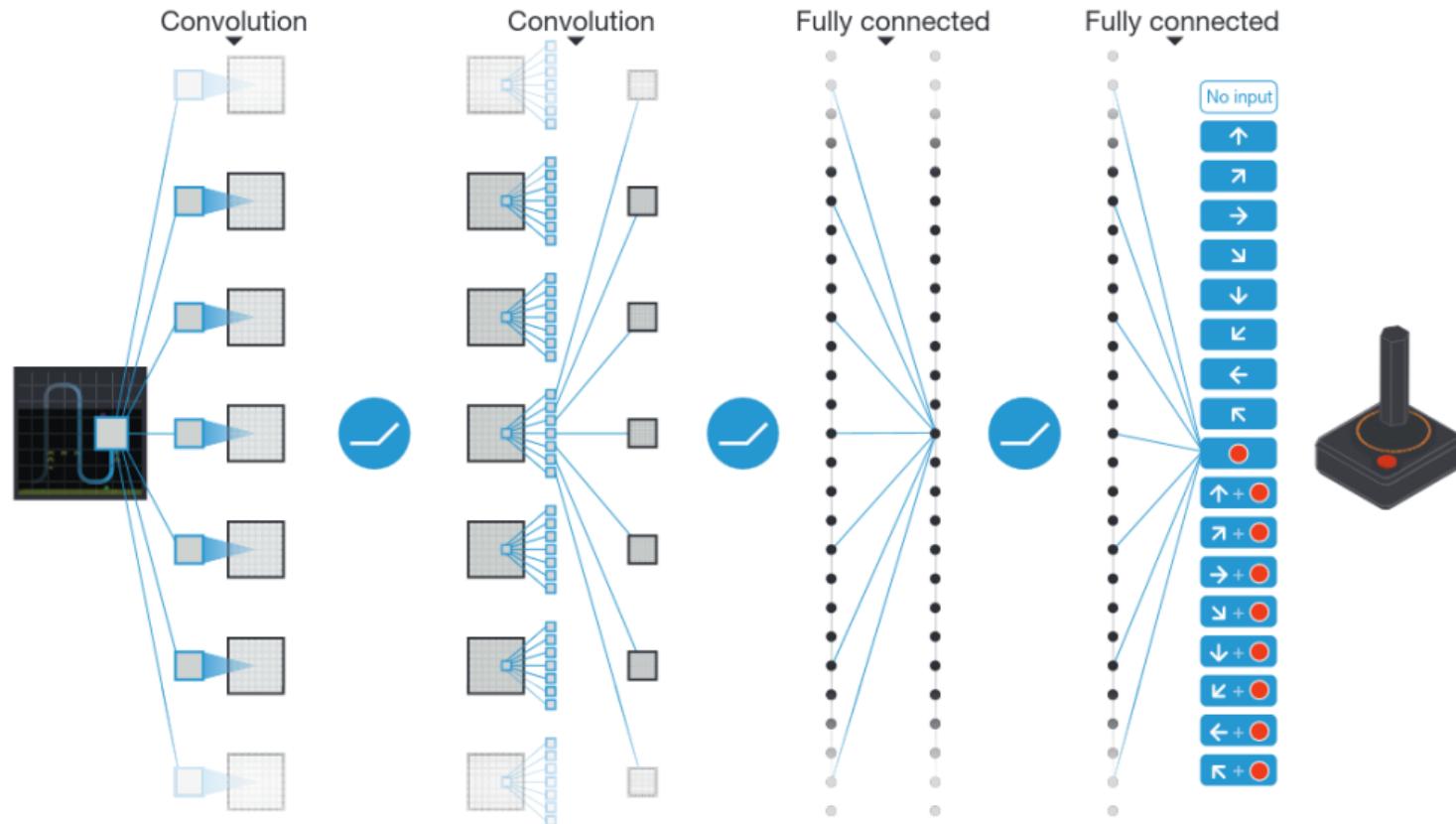
## Example: Atari

- ▶ Frames are  $210 \times 160$  pixel images with a 128 color palette
- ▶ Input dimensionality reduced via **preprocessing**
  - ▶ RGB to gray-scale conversion
  - ▶ down-sampling to  $110 \times 84$
  - ▶ cropped to  $84 \times 84$  to ease implementation
- ▶ State comprises **last 4 frames**
  - ▶ **why?**

## Example: Atari

- ▶ NN input:  $84 \times 84 \times 4$  image produced by preprocessing
- ▶ Conv. layer with 16 8x8 filters with ReLU
- ▶ Conv. layer with 32 4x4 filters with ReLU
- ▶ Fully-connected layer with 256 ReLU units
- ▶ Linear output layer with one unit for each valid action (from 4 to 18 in the considered games)
- ▶ Trained using RMSProp for a total of 50 million frames (around 38 days of game experience in total)
- ▶ Replay memory stores 1 million most recent frames

# Example: Atari



# Target Network

- ▶ DQN may suffer from instability during training, possibly preventing the algorithm to converge
- ▶ In traditional NN training, the training targets do not change over time
- ▶ In DRL, since we don't have ground-truth  $Q$  values, we use the approximated  $\hat{Q}$  in the update target value:

$$y^{(j)} \leftarrow r_j + \gamma \min_{a'} \hat{Q}(s_{i+1}, a', \mathbf{w})$$

- ▶ .....but we keep changing  $\mathbf{w}$  at each iteration
- ▶ Let's use a second neural network to stabilize the targets

# Deep Q-learning with Target Network

- 1 Initialize  $\mathbf{w}$  and  $\mathbf{w}^- = \mathbf{w}$
- 2 Initialize empty buffer  $\mathcal{B}$
- 3  $i \leftarrow 0$
- 4 **Loop**
  - 5 choose action  $a_i$
  - 6 gather experience  $\langle s_i, a_i, r_i, s_{i+1} \rangle$  and add to  $\mathcal{B}$
  - 7 sample minibatch of  $b$   $\langle s_j, a_j, r_j, s_{j+1} \rangle$  tuples from  $\mathcal{B}$
  - 8  $y^{(j)} \leftarrow r_j + \gamma \min_{a'} \hat{Q}(s_{i+1}, a', \mathbf{w}^-), j = 1, \dots, b$
  - 9  $\mathcal{L}^{(j)} = (y^{(j)} - \hat{Q}(s_j, a_j, \mathbf{w}))^2$
  - 10 update  $\mathbf{w}$  using, e.g., SGD on the minibatch
  - 11 every  $C$  steps:  $\mathbf{w}^- \leftarrow \mathbf{w}$
  - 12  $i \leftarrow i + 1$
- 13 **EndLoop**

## Remark

- ▶ DQN can seamlessly work with **continuous** state spaces
- ▶ Action space must be finite

## Example: CartPole with DQN

- ▶ Environment provided by [OpenAI Gym](#)
  - ▶ Large collection of ready-to-use environments
- ▶ DQN implemented using [TF-Agents](#)
  - ▶ RL library part of Tensorflow ecosystem
- ▶ [https://www.tensorflow.org/agents/tutorials/1\\_dqn\\_tutorial?hl=en](https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial?hl=en)

# Policy-based RL

- ▶ So far, we have considered **value-based** RL algorithms
  - ▶ Learn the value function; get a policy from it
- ▶ Now we turn our attention to **policy-based** RL (or, **policy gradient methods**)
  - ▶ Directly learn a **policy**
  - ▶ Algorithms may still learn the value function, but it is not used to derive the policy

# Policy Gradient Methods

- ▶ Algorithms learn a parameterized policy

$$\pi(a|s, \theta) = P(A_t = a | S_t = s, \theta_t = \theta)$$

$\theta \in \mathbb{R}^m$  is the vector of policy parameters

- ▶  $\pi(a|s, \theta)$  can be any function, as long as it is differentiable w.r.t. parameters  $\theta$

## Note

To avoid ambiguity, we will keep using  $w \in \mathbb{R}^d$  to denote the vector of parameters used to approximate the value function, if necessary (e.g.,  $V(s, w)$ )

## Policy Gradient Methods (2)

- ▶ Suppose that  $J(\theta)$  is a performance measure of the policy resulting from parameters  $\theta$  (the higher the better)
- ▶ To maximize performance, we can update  $\theta$  by gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$$

- ▶ The expression “policy gradient” refers to all the methods based on the idea introduced above

# Policy Approximation: Why?

- ▶ Often (but not always), the policy is an easier function to approximate compared to the value function
- ▶ Policy parameterization lets action probabilities change smoothly as a function of the learned parameters, while they can change dramatically for a small change in the action values (if a different action gets the highest value)
  - ▶ stronger convergence guarantees are available
- ▶ Stochastic policies can be learned
- ▶ Continuous action spaces are supported

# Policy Approximation via Action Preferences

- ▶ Let's suppose that the action space is discrete (and not too large)
  - ▶ We will discuss later other scenarios
- ▶ A natural choice for policy approximation is softmax in action preferences:

$$\pi(a|s, \theta) = \frac{e^{h(s,a,\theta)}}{\sum_{a'} e^{h(s,a',\theta)}}$$

- ▶  $h(s, a, \theta)$  is a parameterized numerical preference value for every state-action pair

# Policy Approximation via Action Preferences

- ▶ Note that we don't need any specific strategy to determine the preference values  $h(s, a, \theta)$
- ▶ They are just a convenient way to parameterize the policy  $\pi(a|s, \theta)$
- ▶ For instance, we could use a DNN with a softmax output layer to approximate the policy
  - ▶ Hidden layers compute the action preferences
  - ▶ Output layer produces the policy probabilities

# Action Preferences vs. Action Values

- ▶ Action values  $Q(s, a)$  may differ by a small amount
  - ▶ Softmax based on  $Q$  may struggle to approach a deterministic policy (unless a very small temperature coefficient is used)
- ▶ Action preferences instead do not need to converge to specific values (e.g., the optimal value function), but rather to the best values for the policy to learn
  - ▶ if a deterministic policy is optimal, preference for the optimal action will be as higher as possible than the other actions
  - ▶ if a stochastic policy is optimal, more than one action will have a high preference value (e.g., card games with incomplete information) ...
  - ▶ ...and we can learn *arbitrary* probabilities for actions

# Policy Gradient in Episodic Tasks

- ▶ Let's consider an episodic task starting in state  $s_0$
- ▶ In this case, performance of the policy can be evaluated as

$$J(\theta) = V_{\pi_\theta}(s_0)$$

- ▶ How to update  $\theta$  to improve performance?
- ▶ Performance depends both on (1) action selection and (2) the distribution of states occurring in the episode
- ▶ Both depend on the parameters!
- ▶ (2) is particularly difficult as it also depends on the environment

# Policy Gradient Theorem

## Policy Gradient Theorem

$$\nabla_{\theta} J(\theta) \propto \sum_s \mu(s) \sum_a Q_{\pi}(s, a) \nabla_{\theta} \pi(a|s, \theta)$$

- ▶ Proportionality constant is the average length of an episode
  - ▶ in gradient ascent, constant absorbed by step size  $\alpha$  ✓
- ▶ we don't need the derivative of  $\mu(s)$  ✓

# Policy Gradient Theorem: Proof

To simplify notation, we leave it implicit that  $\pi$  is a function of  $\theta$ , and that gradients are w.r.t.  $\theta$

$$\begin{aligned}\nabla V_\pi(s) &= \nabla \left[ \sum_a \pi(a|s) Q_\pi(s, a) \right] = \\ &= \sum_a \nabla [\pi(a|s) Q_\pi(s, a)] = \\ &= \sum_a [\nabla \pi(a|s) Q_\pi(s, a) + \pi(a|s) \nabla Q_\pi(s, a)] = \\ &= \sum_a \left[ \nabla \pi(a|s) Q_\pi(s, a) + \pi(a|s) \nabla \sum_{s', r'} p(s', r'|s, a) (r + V_\pi(s')) \right] =\end{aligned}$$

## Proof (2)

$$= \sum_a \left[ \nabla \pi(a|s) Q_\pi(s, a) + \pi(a|s) \nabla \sum_{s', r'} p(s', r'|s, a) (r + V_\pi(s')) \right] =$$

1) Reward does not depend on  $\theta$  (gradient is 0)

2)  $\sum_{s'} \sum_{r'} p(s', r'|s, a) V_\pi(s') = \sum_{s'} p(s'|s, a) V_\pi(s')$

$$= \sum_a \left[ \nabla \pi(a|s) Q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s'|s, a) \nabla V_\pi(s') \right] =$$

Note: we are computing  $\nabla V_\pi(s)$  and now we have a recursive term  $\nabla V_\pi(s')$ ! Let's unroll the recursion...

## Proof (3)

$$\begin{aligned} &= \sum_a \left[ \nabla \pi(a|s) Q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s'|s, a) \cdot \right. \\ &\quad \left. \sum_{a'} \left( \nabla \pi(a'|s') Q_\pi(s', a') + \pi(a'|s') \sum_{s''} p(s''|s', a') \nabla V_\pi(s'') \right) \right] = \end{aligned}$$

After repeated unrolling ...

$$= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} P(s \rightarrow x, k, \pi) \sum_a [\nabla \pi(a|x) Q_\pi(x, a)]$$

where  $P(s \rightarrow x, k, \pi)$  is the probability of transitioning from  $s$  to  $x$  in  $k$  steps under policy  $\pi$ .

## Proof (4)

We can now write an expression for the gradient of  $J$

$$\nabla J(\theta) = \nabla V_\pi(s_0) = \sum_s \sum_{k=0}^{\infty} P(s_0 \rightarrow s, k, \pi) \sum_a [\nabla \pi(a|s) Q_\pi(s, a)] =$$

$$= \sum_s \eta(s) \sum_a [\nabla \pi(a|s) Q_\pi(s, a)] =$$

$\eta(s)$ : avg. number of steps spent in  $s$  within an episode

$$= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a [\nabla \pi(a|s) Q_\pi(s, a)] =$$

## Proof (5)

$$= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a [\nabla \pi(a|s) Q_\pi(s, a)] =$$

$$= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a [\nabla \pi(a|s) Q_\pi(s, a)]$$

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a [\nabla \pi(a|s) Q_\pi(s, a)]$$

# REINFORCE

- ▶  $\mu(s)$  is the on-policy distribution of states under  $\pi$ 
  - ▶ if  $\pi$  is followed, states will occur in that proportion

$$\begin{aligned}\nabla_{\theta} J(\theta) &\propto \sum_s \mu(s) \sum_a Q_{\pi}(s, a) \nabla_{\theta} \pi(a|s, \theta) = \\ &= E_{\pi} \left[ \sum_a Q_{\pi}(s_t, a) \nabla_{\theta} \pi(a|s_t, \theta) \right]\end{aligned}$$

- ▶ as we did to approximate the value function, we can perform a **stochastic gradient** ascent on occurring states  $s_t$

## REINFORCE (2)

- ▶ We replaced a sum over states with an expectation under  $\pi$
- ▶ We want to do the same with the sum over actions
- ▶ First, we need to weigh actions by  $\pi(a|s, \theta)$

$$\begin{aligned}\nabla_{\theta} J(\theta) &\propto E_{\pi} \left[ \sum_a Q_{\pi}(s_t, a) \nabla_{\theta} \pi(a|s_t, \theta) \right] = \\&= E_{\pi} \left[ \sum_a \cancel{\pi(a|s_t, \theta)} Q_{\pi}(s_t, a) \frac{\nabla_{\theta} \pi(a|s_t, \theta)}{\cancel{\pi(a|s_t, \theta)}} \right] = \\&= E_{\pi} \left[ Q_{\pi}(s_t, a_t) \frac{\nabla_{\theta} \pi(a_t|s_t, \theta)}{\pi(a_t|s_t, \theta)} \right] = \\&= E_{\pi} \left[ G_t \frac{\nabla_{\theta} \pi(a_t|s_t, \theta)}{\pi(a_t|s_t, \theta)} \right]\end{aligned}$$

## REINFORCE (3)

$$\nabla J(\theta) \propto E_{\pi} \left[ G_t \frac{\nabla_{\theta} \pi(a_t | s_t, \theta)}{\pi(a_t | s_t, \theta)} \right]$$

- ▶ We can sample  $G_t$  for each time step, and we got an expression proportional to the gradient
- ▶ We can use stochastic gradient ascent to update parameters
- ▶ The resulting algorithm is REINFORCE (1992)

$$\theta_{t+1} \leftarrow \theta_t + \alpha G_t \nabla_{\theta} \ln \pi(a_t | s_t, \theta)$$

$$\frac{\nabla_{\theta} \pi(a_t | s_t, \theta)}{\pi(a_t | s_t, \theta)} = \nabla_{\theta} \ln \pi(a_t | s_t, \theta)$$

# REINFORCE: Algorithm

We also include a discount factor  $\gamma$

```
1 Initialize  $\theta$  (e.g., to 0)
2 Loop
3   generate episode  $s_0, a_0, r_1, s_1, \dots, r_T$  following  $\pi$ 
4   for  $t=0,1,\dots,T$  do
5      $G_t \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} r_k$ 
6      $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \ln \pi(a_t | s_t, \theta)$ 
7   end
8 EndLoop
```

## Policy Gradient: Another Perspective

- ▶ Consider a NN used for multi-class classification
- ▶ Softmax final layer outputs a probability  $y_c$  for all classes  $c$
- ▶ Gradient of cross-entropy loss used for training

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} \left[ \sum_c \bar{y}_c \ln y_c \right]$$

- ▶ Intuitively, training will increase prob.  $y_c$  for the class  $c$  labeled as correct (ground truth)
  - ▶ likelihood maximization
- ▶ Replacing classes with “actions”, the NN outputs  $\pi(a|s, \theta)$
- ▶ REINFORCE update is proportional to  $\nabla_{\theta} \ln \pi(a_t|s_t, \theta)$ 
  - ▶ without a ground truth, prob. is increased/decreased based on return

# REINFORCE with Baseline

- ▶ A slight generalization of REINFORCE involves the use of a **baseline**  $b(s)$
- ▶ We compare the value of each action to  $b(s)$
- ▶ The policy gradient theorem remains true

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a (Q(s, a) - b(s)) \nabla \pi(a|s, \theta)$$

- ▶ Baseline can be useful to reduce the variance of the update and speed learning

$$\theta_{t+1} = \theta_t + \alpha (G_t - b(s_t)) \nabla \ln \pi(a_t|s_t, \theta)$$

# Actor-Critic

- ▶ Idea to avoid high variance of returns used by REINFORCE
- ▶ Using the one-step return  $G_{t:t+1}$  instead

$$G_{t:t+1} = r_t + \gamma V(s_{t+1})$$

- ▶ We call **critic** the role of the value function used in this way
- ▶ We call the resulting approach **actor-critic**

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha \left( G_{t:t+1} - \hat{V}(s_t, \mathbf{w}) \right) \nabla \ln \pi(a_t | s_t, \boldsymbol{\theta}) = \\ &= \boldsymbol{\theta}_t + \alpha \left( r_{t+1} + \gamma \hat{V}(s_{t+1}, \mathbf{w}) - \hat{V}(s_t, \mathbf{w}) \right) \nabla \ln \pi(a_t | s_t, \boldsymbol{\theta})\end{aligned}$$

```

1 Initialize  $\theta$  and  $w$  (e.g., to 0)
2 Loop
3   Initialize  $s$  as first state of the episode
4    $l \leftarrow 1$ 
5   while  $s$  not terminal do
6     choose action  $a$  according to  $\pi(\cdot|s, \theta)$ 
7     observe  $s'$  and  $r$ 
8      $\delta \leftarrow r + \gamma \hat{V}(s', w) - \hat{V}(s, w)$ 
9      $w \leftarrow w + \alpha^w \delta \nabla \hat{V}(s, w)$ 
10     $\theta \leftarrow \theta + l \alpha^\theta \delta \nabla \ln \pi(a|s, \theta)$ 
11     $l \leftarrow \gamma l$ 
12     $s \leftarrow s'$ 
13  end
14 EndLoop

```

## The Continuing Case

- ▶ Policy gradient theorem holds for continuing tasks as well
- ▶ The performance measure  $J(\theta)$  must be changed to the average reward
- ▶ Proof and updated Actor-Critic alg. in Sutton-Barto, 13.6

# Continuous or Large Action Spaces

- ▶ Policy gradient methods can be useful in presence of very large or continuous action spaces
  - ▶ Computing the learned probability for every action can be expensive/unfeasible
- ▶ The solution: learn the parameters of a probability distribution, instead of the probability of choosing each action
- ▶ Example: policy defined as the normal probability density

$$\pi(a|s, \theta) = \frac{1}{\sigma(s, \theta)\sqrt{2\pi}} \exp\left(-\frac{(a - \mu(s, \theta))^2}{2\sigma(s, \theta)^2}\right)$$

# Example

**CartPole with Actor-Critic in TensorFlow:** [https://www.tensorflow.org/tutorials/reinforcement\\_learning/actor\\_critic?hl=en](https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic?hl=en)

# Advanced Policy Methods

- ▶ **Deterministic Policy Gradient (DPG)**: similar to PG theorem above, but for deterministic policies
  - ▶ Silver et al. (2014), “Deterministic Policy Gradient Algorithm”,  
<http://proceedings.mlr.press/v32/silver14.pdf>
- ▶ **Deep Deterministic Policy Gradient (DDPG)**: DPG with DNNs
  - ▶ Lillicrap et al. (2016), “Continuous control with deep reinforcement learning”,  
<https://arxiv.org/abs/1509.02971>
- ▶ **Proximal Policy Optimization (PPO)**: state-of-the-art algorithm
  - ▶ Schulman et al. (2017), “Proximal Policy Optimization Algorithms”,  
<https://arxiv.org/pdf/1707.06347.pdf>

# Improved DQN Approaches

- ▶ **Rainbow**: combining several extensions of DQN
  - ▶ Hessel et al. (2017), "Rainbow: Combining Improvements in Deep Reinforcement Learning.", <https://arxiv.org/pdf/1710.02298>

# Advanced RL Topics

- ▶ Multi-agent RL
- ▶ Hierarchical RL
- ▶ RL + Heuristic Tree Search
- ▶ Transfer RL
- ▶ ...

# References I

-  Kephart, J.O. and D.M. Chess (2003). "The Vision of Autonomic Computing". In: *IEEE Computer* 36.1, pp. 41–50. DOI: [10.1109/MC.2003.1160055](https://doi.org/10.1109/MC.2003.1160055).
-  Russo Russo, G. et al. (2021). "MEAD: Model-Based Vertical Auto-Scaling for Data Stream Processing". In: *Proceedings of 21th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID '21, Virtual Event, May 10-13, 2021*, pp. 314–323. DOI: [10.1109/CCGrid51090.2021.00041](https://doi.org/10.1109/CCGrid51090.2021.00041).
-  Watkins, Christopher J. C. H. and Peter Dayan (1992). "Technical Note Q-Learning". In: *Mach. Learn.* 8, pp. 279–292.
-  Weyns, D. (2020). *An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective*. Hoboken, NJ, USA: Wiley-IEEE Computer Society Press.

## References II

-  Weyns, D. et al. (2013). "On Patterns for Decentralized Control in Self-Adaptive Systems". In: *Software Engineering for Self-Adaptive Systems II*. Vol. 7475. LNCS. Springer.