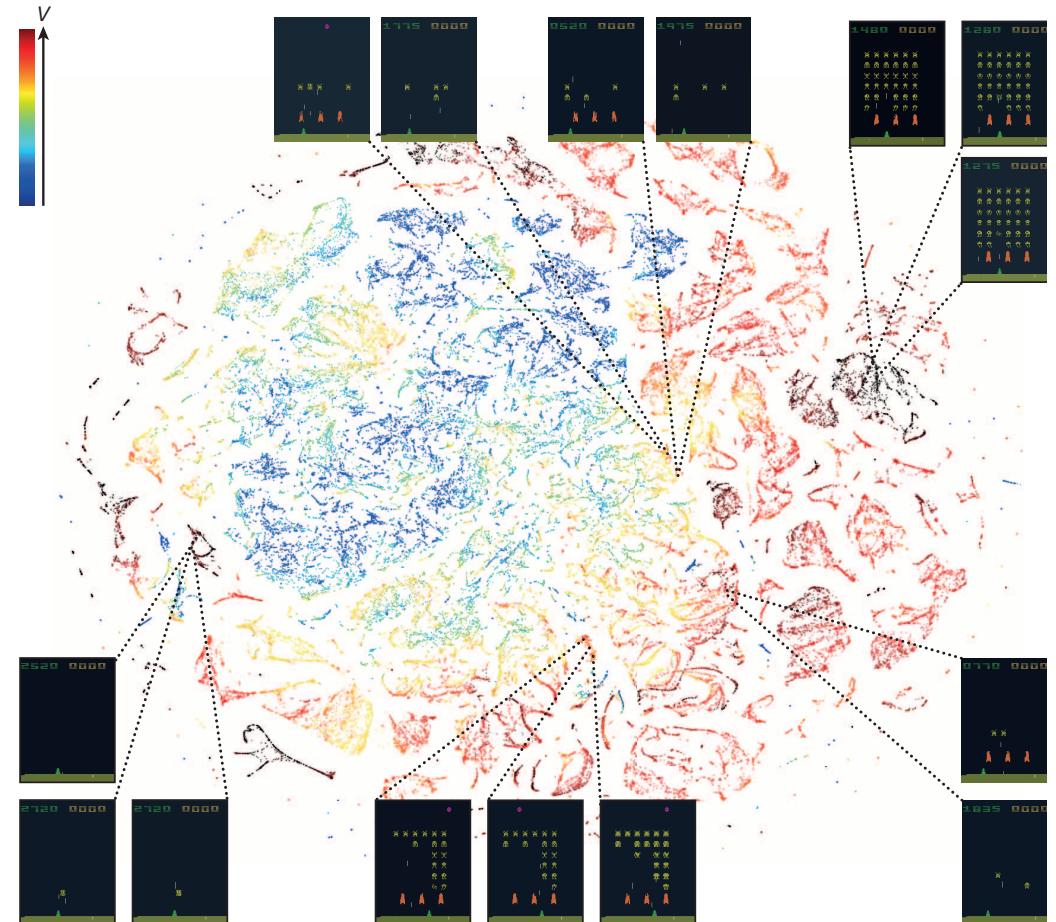


# Reinforcement Learning Introduction

**Reinforcement Learning**  
September 22, 2022



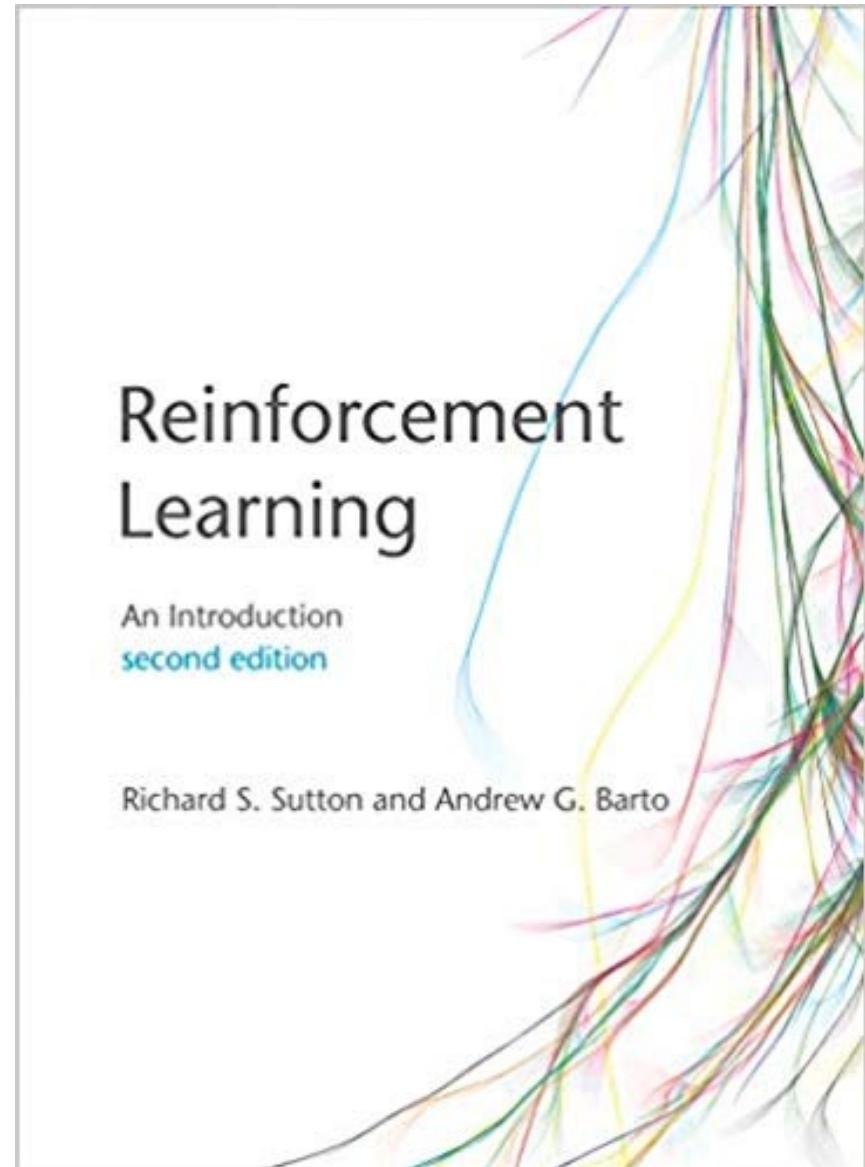
# Welcome to the course

The course uses the textbook:

*Reinforcement Learning: An Introduction*, Richard S. Sutton and Andrew G. Barto, 2<sup>nd</sup> edition

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

Reading assignments are given for each topic



# Course Administration

Most topics will contain

- quizzes on Illias and
- exercises.

Both are mandatory for the testat. Exercises will give 10 points each, and a minimum number of points is needed for each exercise.

There (should) be enough time during the course to solve the exercises.  
Exercises are available on a kubernetes cluster using jupyter lab and nbgrader.

# Exercise Environment

Login to <https://gpuhub.el.eee.intern> using your enterpriselab account.

Select *Reinforcement Learning Course Image*

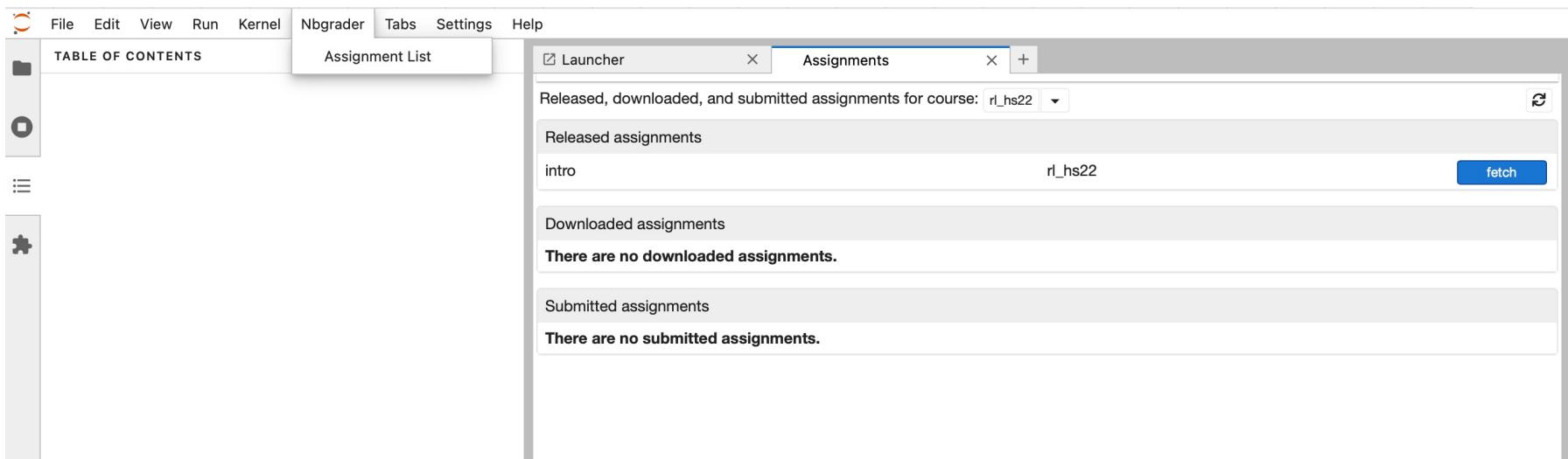
## Server Options

- Minimal environment**  
Spawns the baseline JupyterLab server
- Tensorflow & PyTorch environment**  
Spawns a JupyterLab server with Tensorflow and PyTorch
- Reinforcement Learning Course**  
Spawns a JupyterLab server for the RL course
- Reinforcement Learning Admin**  
Only for RL course administration
- Deep Learning 4 Games Course**  
Spawns a JupyterLab server for the DL4G course

Start

# Exercises Environment

## Select nbgrader->Assignment List



Select *fetch* to get the assignment and *submit* to submit it when solved

# Intro to python

- The exercise are in python 3.
- In the first week there are no exercises, but a python course and some python basic exercises (that do not need to hand in)
- If you have not programmed python yet, please familiarize yourself with python in the first week ☺.



# Learning Objectives: Introduction

- Differentiate between Reinforcement Learning (RL) and other Machine Learning (ML) Techniques
- Know when RL methods can be applied and when not
- Explain the interaction of a RL technique with the environment
- Know the different types of RL agents

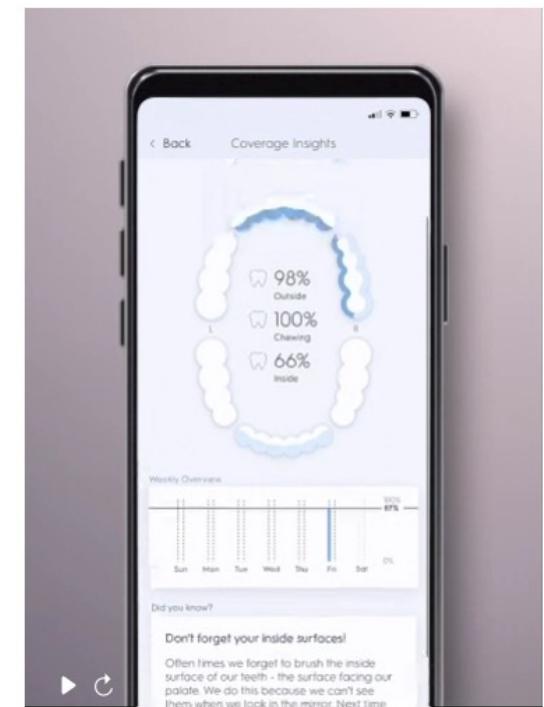
# In products...



## KEY FEATURES

# ASSESS YOUR BRUSHING HABITS

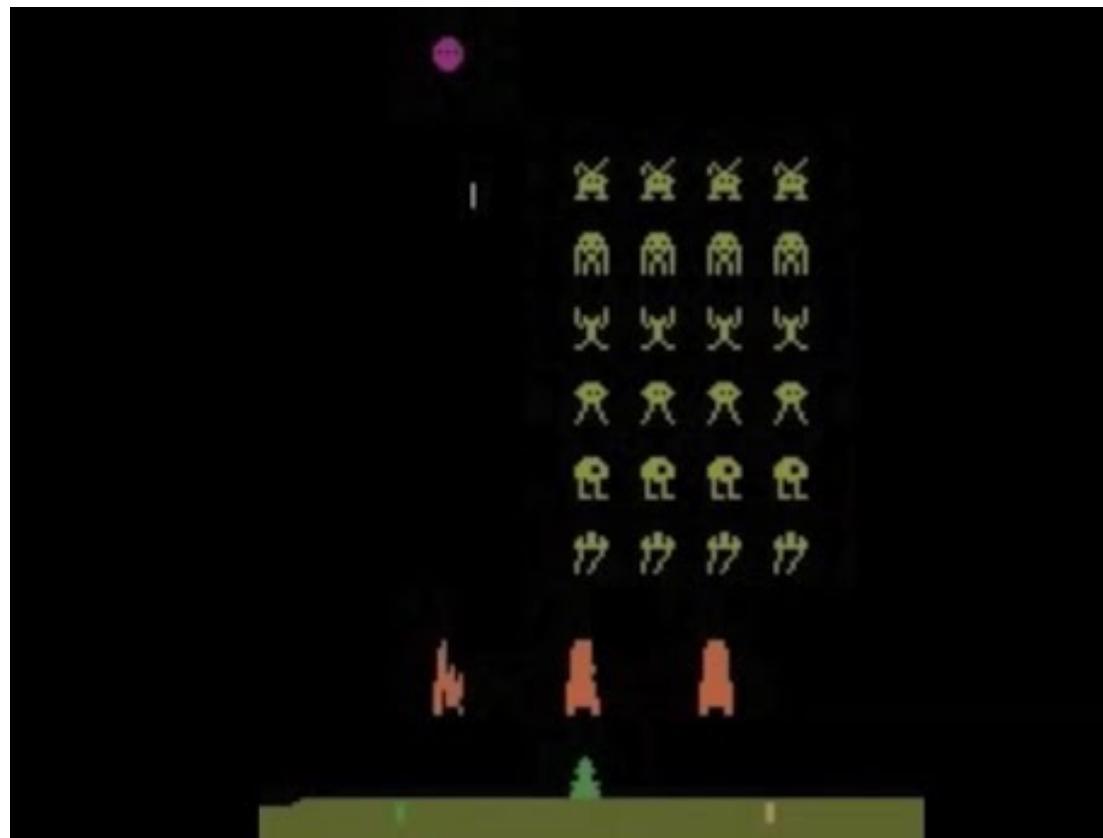
Brushing habits are deeply ingrained – most of us brush or miss the same teeth over and over again. With the new Oral-B app, you can see your old brushing patterns and actively work on improving them. And ensure that you're spending enough time on each zone.



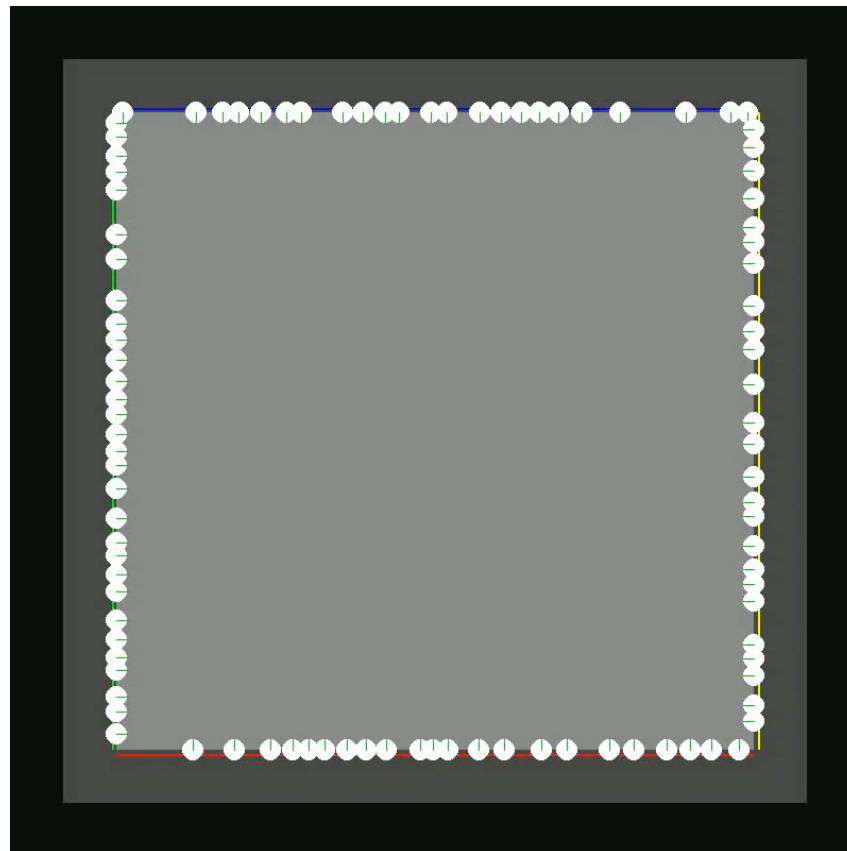
Example:



## Example: Atari Games (Deepmind)



## Example: Crossing (HSLU, ABIZ)



## Example: Hide and Seek



# Reinforcement Learning

What is reinforcement learning?

What is reinforcement learning not?

It is not supervised learning:

- There is not data available from an external expert

It is not unsupervised learning:

- It is not about finding structures in data or interpreting unlabeled data.

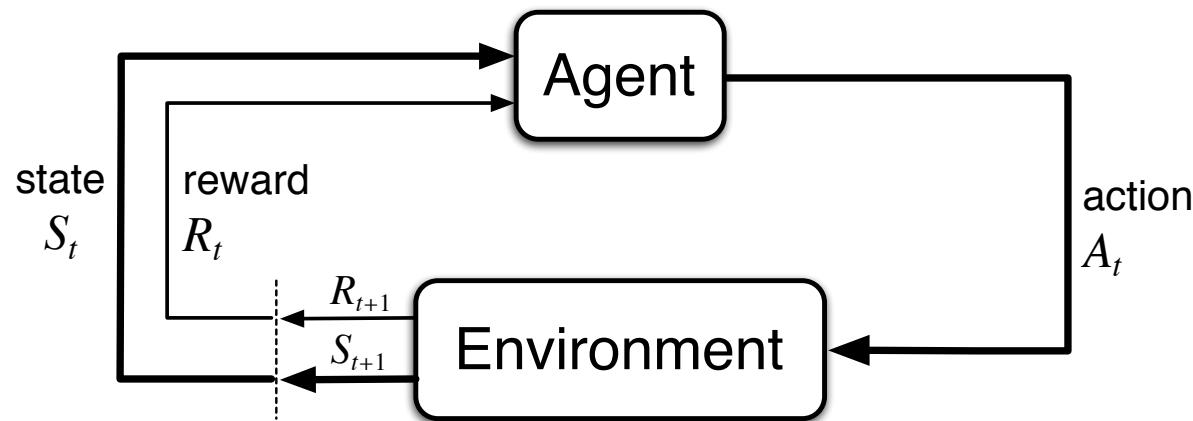
# Goal of Reinforcement Learning

In reinforcement learning:

- An agent tries different **actions** and
- Receives a **reward**

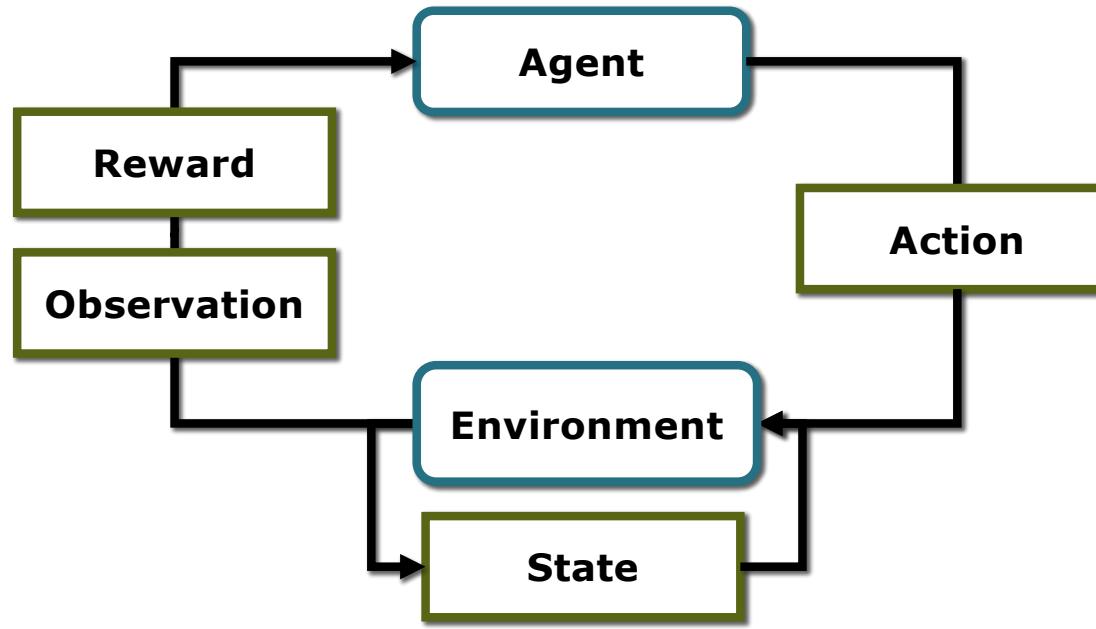
**All goals can be described by  
the maximization of the  
expected cumulative reward**

# Agent and Environment



from Sutton & Barto, 2018

## Agent and Environment (II)



In many problems and also in the most common implementations, the agent might not receive the full state, but a so called **observation** of the state.

Either for simplicity or because the agent is not able to observe the full state (for example in a game like Poker or Jass)

# Cumulative Rewards

Maximizing the cumulative reward or expected return:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

Often, a *discounted* return is used:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$
$$0 \leq \gamma \leq 1$$

# Concepts and Notation

$A_t$	action at time $t$
$S_t$	state at time $t$ , typically due, stochastically, to $S_{t-1}$ and $A_{t-1}$
$R_t$	reward at time $t$ , typically due, stochastically, to $S_{t-1}$ and $A_{t-1}$
$\pi$	policy (decision-making rule)
$\pi(s)$	action taken in state $s$ under <i>deterministic</i> policy $\pi$
$\pi(a s)$	probability of taking action $a$ in state $s$ under <i>stochastic</i> policy $\pi$
$G_t$	return following time $t$
$v_\pi(s)$	value of state $s$ under policy $\pi$ (expected return)
$v_*(s)$	value of state $s$ under the optimal policy
$q_\pi(s, a)$	value of taking action $a$ in state $s$ under policy $\pi$
$q_*(s, a)$	value of taking action $a$ in state $s$ under the optimal policy

# Types of RL Agents (will be covered in the lecture)

## **Value based:**

- No Policy (implicit)
- Value Function

## **Model Free**

- Policy and/or Value Function
- No Model

## **Policy Based:**

- Policy
- No Value Function

## **Model**

- Policy and/or Value Function
- Model (explicit or learned)

## **Actor Critic**

- Policy
- Value Function

## **Tabular Methods**

- Policy and/or Value Function for each state

# Multiarmed Bandits

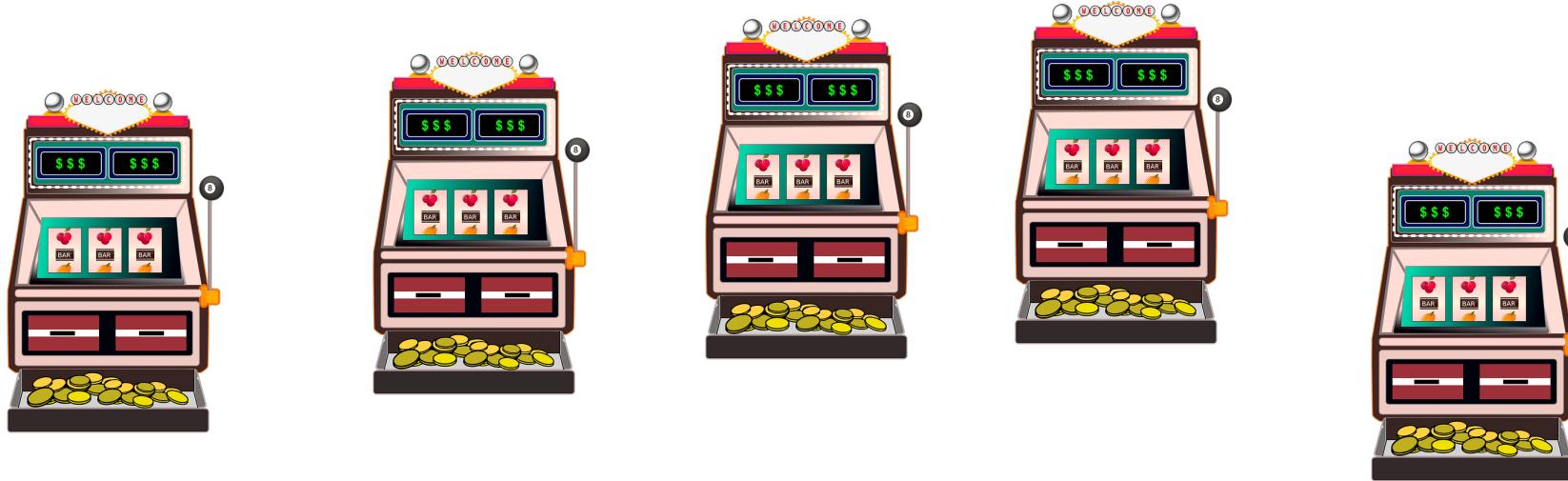
**Reinforcement Learning**  
September 22, 2022



# Learning Objectives

- Define multi-armed bandits as an RL problem
- Understand the meaning of value and policy functions
- Understand exploration and why RL algorithms need it
- Know epsilon and epsilon greedy policies
- Define the update of the value function from experience

# Multi-Armed Bandit Problem



- k Slot Machines with reward distributed by (different) probability functions
- 1000 coins to play
- Goal: Maximise the total reward
- What is the best **policy** to play?

# Multi-Armed Bandit

How would **you** play?

We want to teach an agent to play a multi-armed bandit:

- What are the possible actions?
- How does the value function look like?

# Actions in Reinforcement Learning

- An agent **evaluates** actions in RL
- An agent **does not instruct** the correct actions
- An agent employs **active exploration** to search for good (or the best) behavior

(this refers to training, a trained agent will follow the learned (optimal) policy)

## Formulation of the problem

- The actual value of an action  $a$  is the expected reward

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a]$$

(which is not known)

- The estimated value of an action  $a$  is called the (action-) value function

$$Q_t(a)$$

which we would like to be close to the true value

## Action-Value Methods

A simple method to estimate the action values is to average the rewards whenever the action was taken

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n - 1}$$

(however, we need to keep all the rewards)

## Incremental calculation

We would prefer an incremental calculation instead

$$\begin{aligned} Q_{n+1} &= \frac{1}{n} \sum_{i=1}^n R_i \\ &= \frac{1}{n} (R_n + \sum_{i=1}^{n-1} R_i) \\ &= \frac{1}{n} (R_n + (n-1) \frac{1}{n-1} \sum_{i=1}^{n-1} R_i) \\ &= \frac{1}{n} (R_n + (n-1)Q_n) \\ &= \frac{1}{n} (R_n + nQ_n - Q_n) \\ &= Q_n + \frac{1}{n} (R_n - Q_n) \end{aligned}$$

## General Update Formula

$$Q_{n+1} = Q_n + \frac{1}{n}(R_n - Q_n)$$

The last line in the previous equation can be written as

NewEstimate  $\leftarrow$  OldEstimate + StepSize [Target – OldEstimate]

The term [Target – OldEstimate] is an *error* that we want to reduce by taking a step towards the Target

*Many RL algorithm use this formula with different values of the error and the StepSize*

# Exploitation and Exploration

## Exploitation:

- Exploit current knowledge by taking the action with the maximal estimated value
- Greedy action

## Exploration:

- Explore the value of other actions to get better estimates
- Non greedy actions

# Epsilon Greedy Methods

Exploitation:

With probability  $1-\varepsilon$ :

- Take action with maximal  $Q_t(a)$  (greedy action)

Exploration:

With probability  $\varepsilon$ :

- Take any valid action with equal probability

Implementation:

- Draw random variable in  $[0..1]$
- Compare with threshold  $\varepsilon$

# Simple Multi Armed Bandit Agent

## A simple bandit algorithm

Initialize, for  $a = 1$  to  $k$ :

$$Q(a) \leftarrow 0$$

$$N(a) \leftarrow 0$$

Loop forever:

$$A \leftarrow \begin{cases} \arg \max_a Q(a) & \text{with probability } 1 - \epsilon \quad (\text{breaking ties randomly}) \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$$

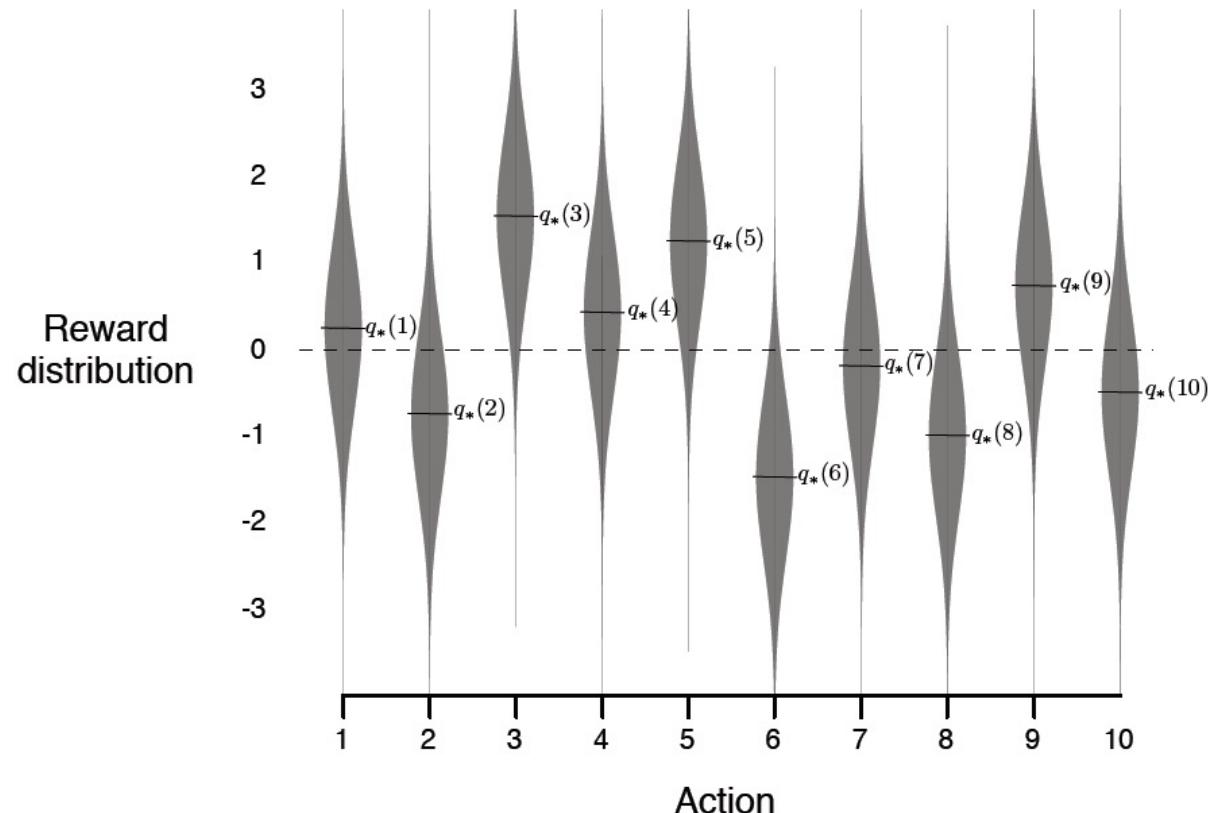
$$R \leftarrow \text{bandit}(A)$$

$$N(A) \leftarrow N(A) + 1$$

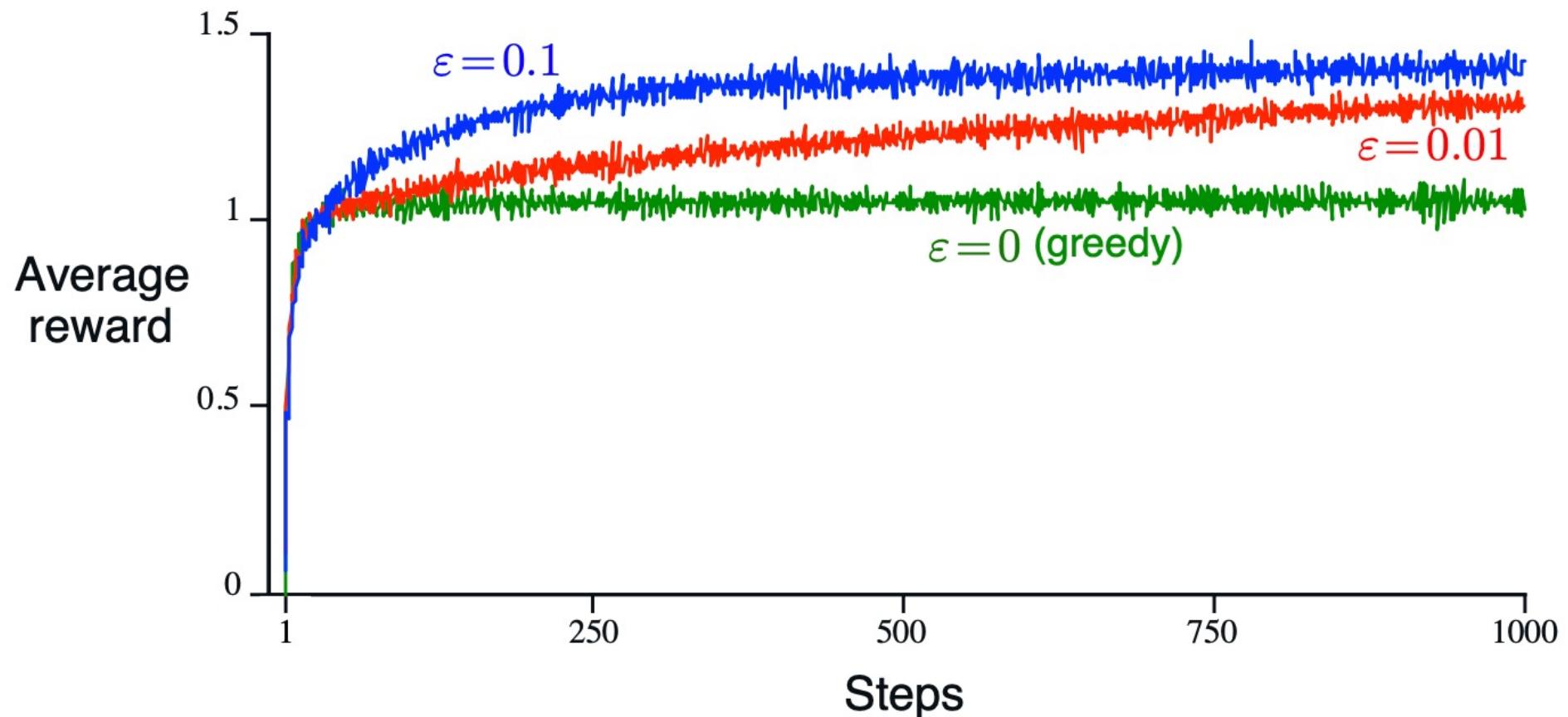
$$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$$

# Testbed 10-armed bandits

- 10 bandits with different mean reward (drawn from Normal probability distribution with mean 0)
- Return reward with Normal distribution ( $\sigma=1.0$ ) around mean value



## Comparison of exploration



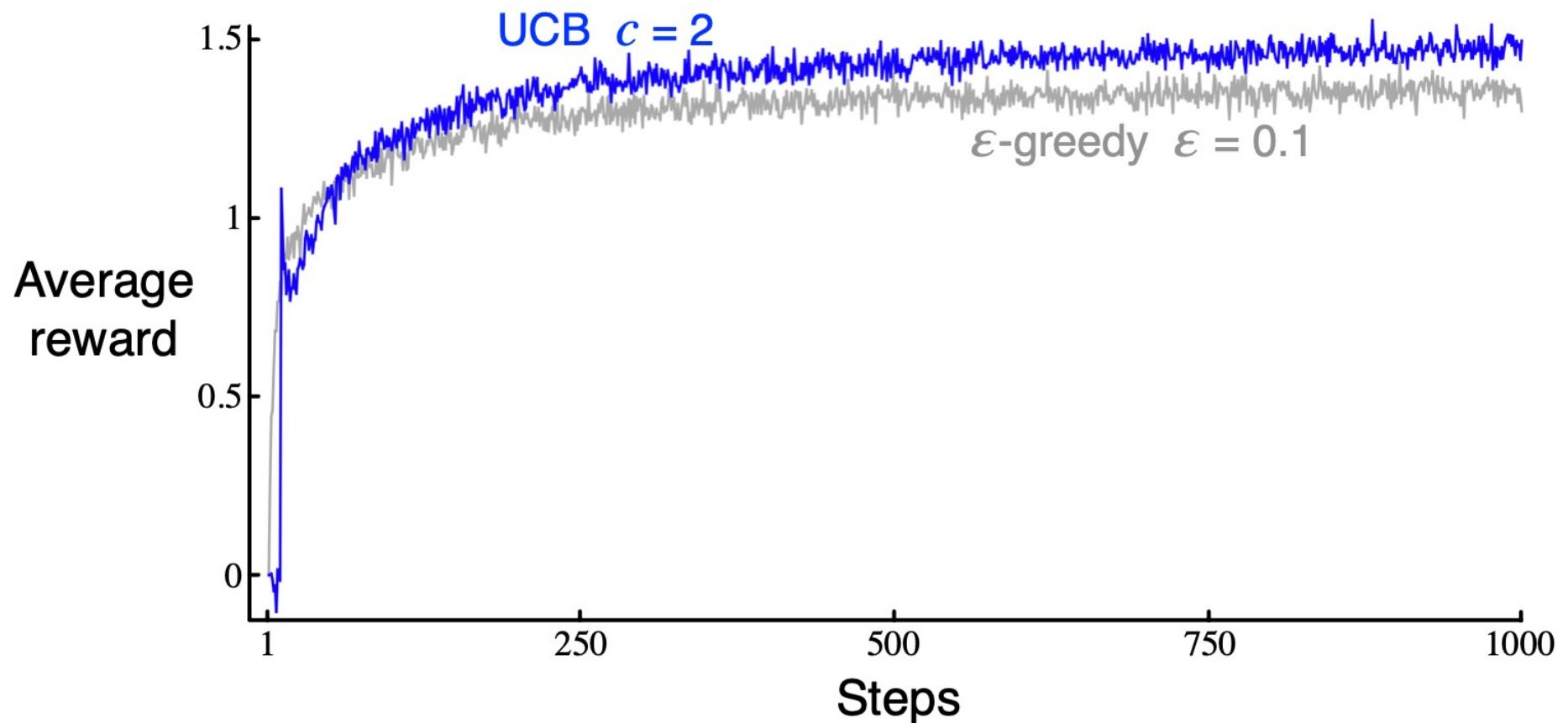
# Upper Confidence Bound

- Epsilon-greedy methods are not selecting the most promising methods during exploration, and
- Epsilon-greedy methods are not efficient once the best method has been found
- A better method is the upper confidence bound (UCB) algorithm that includes a term to measure the uncertainty in the estimate

$$A_t \doteq \arg \max_a \left[ Q_t(a) + c \sqrt{\frac{\ln t}{N_t(a)}} \right]$$


Number of times that this action has been selected previously

## Upper Confidence Bound



# Introduction & Multiarmed Bandits

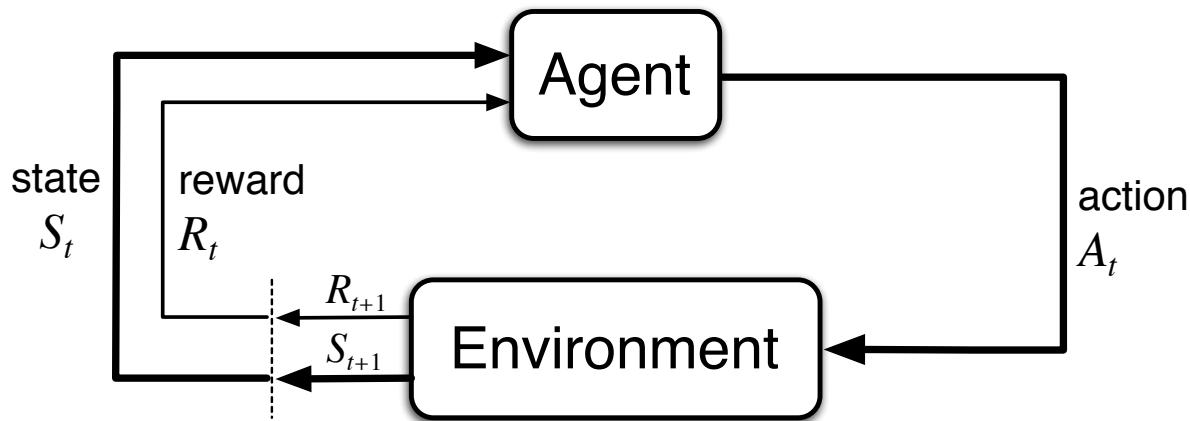
Summary

**Reinforcement Learning**  
September 29, 2022



# Agent and Environment

In RL: An agent interacts with an environment using actions and gets a reward for each action



The agent's goal is to maximize the expected cumulative reward

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

# Multi-Armed Bandits

- Solving the multi-armed bandit problem required exploring different actions and exploiting the action which currently seems best
- The expected rewards of each action are estimated by calculating a value function incrementally using

$$Q_{n+1} = Q_n + \frac{1}{n}(R_n - Q_n)$$

NewEstimate  $\leftarrow$  OldEstimate + StepSize [Target – OldEstimate]

- A epsilon-greedy policy can be defined using this value function and selecting greedy and non-greedy actions with probabilities  $1-\epsilon$ , respectively  $\epsilon$
- UCB better balances exploration better in the long run

# Simple Multi Armed Bandit Agent

## A simple bandit algorithm

Initialize, for  $a = 1$  to  $k$ :

$$Q(a) \leftarrow 0$$

$$N(a) \leftarrow 0$$

Loop forever:

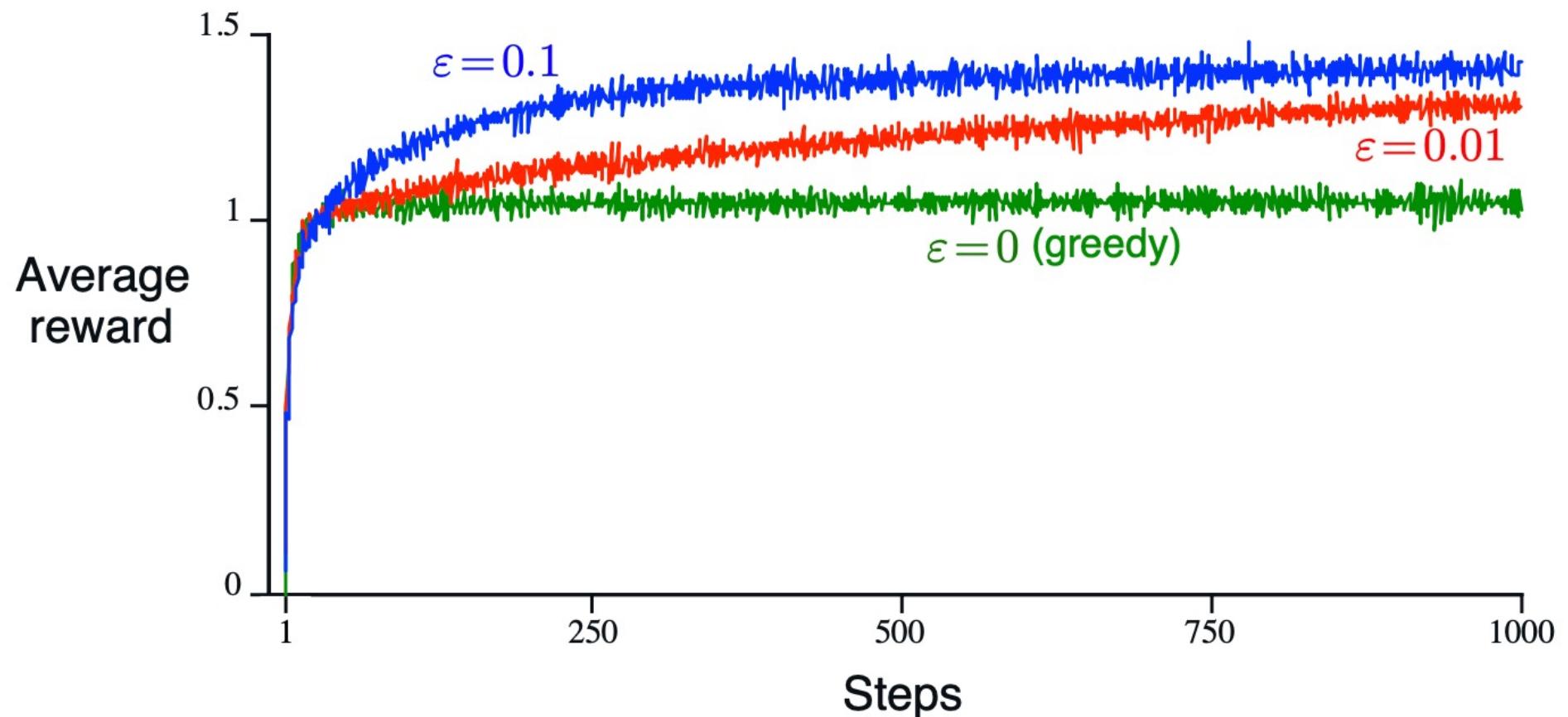
$$A \leftarrow \begin{cases} \arg \max_a Q(a) & \text{with probability } 1 - \epsilon \quad (\text{breaking ties randomly}) \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$$

$$R \leftarrow \text{bandit}(A)$$

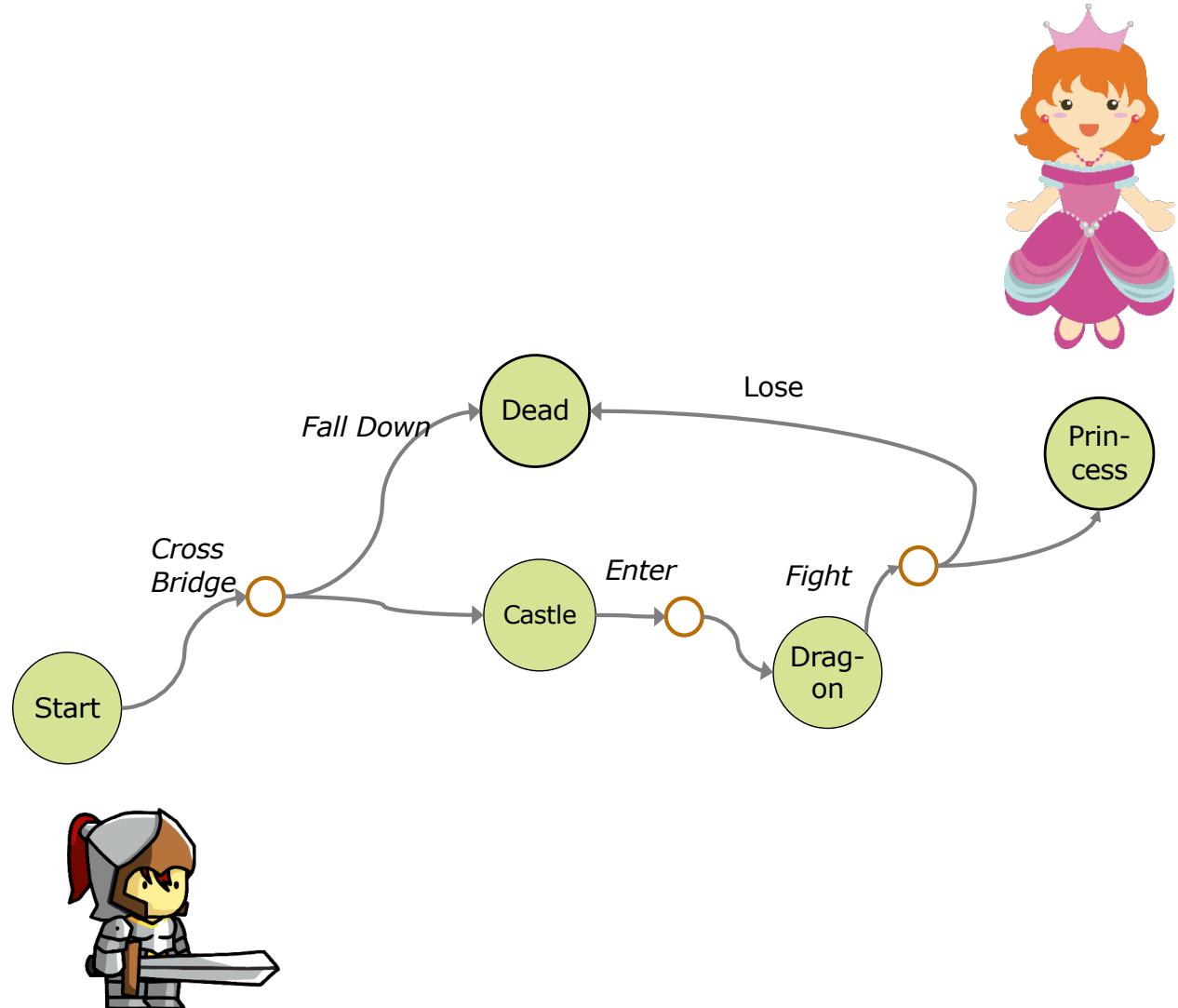
$$N(A) \leftarrow N(A) + 1$$

$$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$$

## Comparison of exploration



# Markov Decision Processes

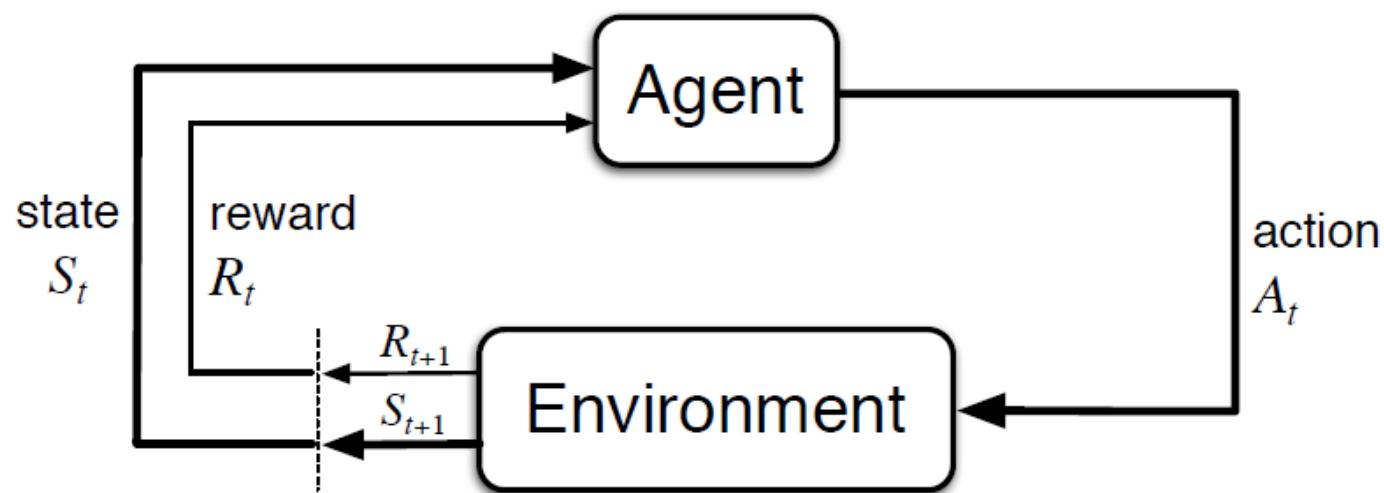


**Reinforcement Learning**  
October 6, 2022

# Learning Objectives

- How to formulate a RL problem as Markov Decision Process
- Understand episodic tasks and episodes
- Differentiate between episodic and continuous tasks
- Understand how discounts are used for returns in continuing tasks
- Derive the Bellman equation
- Differentiate between state-value and action-value functions
- Understand backup diagrams

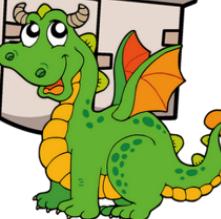
# Agent and Environment



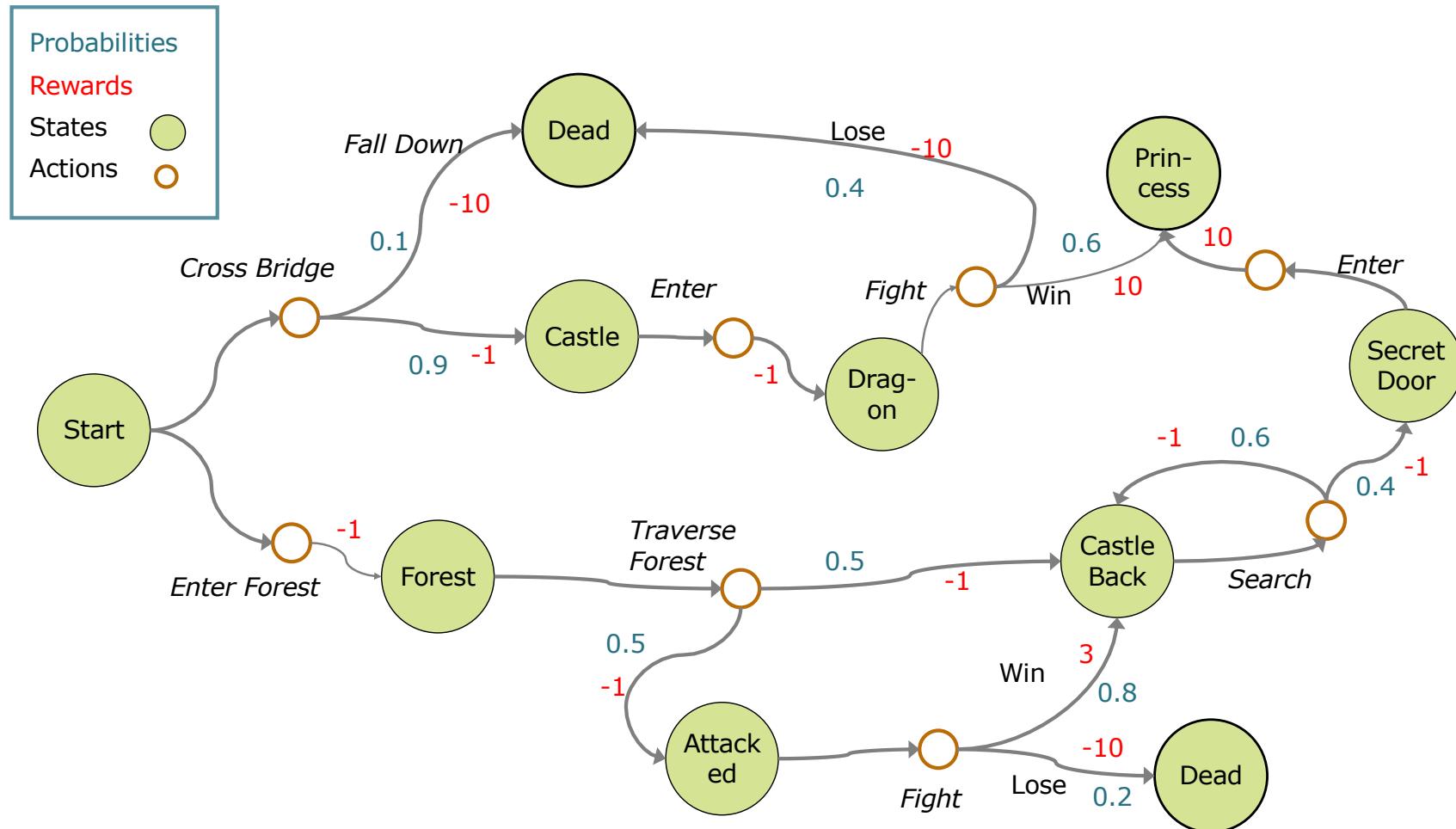
# Markov Decision Process (MDP)

- (not Markov Process, which has no decisions))
- Markov property: *The future is independent of the past given the present*
- An Environment consists of states  $S_t$
- An action (decision)  $a_t$  that can be taken in each state
- The action will give a reward and a new state

Which way



# Markov Decision Process



## Examples

Sample episodes:

Start → *Cross Bridge* → Castle → *Enter* → Dragon → *Fight* → Princess

Start → *Enter Forest* → Forest → *Traverse Forest* → Castle Back → *Search* → Castle Back → *Search* → Princess

What are the rewards?

## How to solve this?

Goal: Find the sequence of best actions, i.e. maximize the cumulative reward

How would you solve the problem?

## How to solve....

... depends on the information available:

- Do we know all the states, or do we have to discover them?
- Do we know the probabilities? Explicitly or can we sample from them?
- Do we have a full model of the problem?
- Can we start exploring at every state?
- ...

## Definition of MDP

The dynamics of an MDP is defined as

$$p(s', r | s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

(this can be viewed as a function of 4 parameters)

Goal (remains the same for MDP)

Maximizing the expected return:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

Maximizing the *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$
$$0 \leq \gamma \leq 1$$

## Episodic vs. Continuing tasks

- Many MDP are naturally episodic, where they end at a terminal state, these are *episodic tasks*
- Others are *continuous tasks* which do not end but run forever



MACH



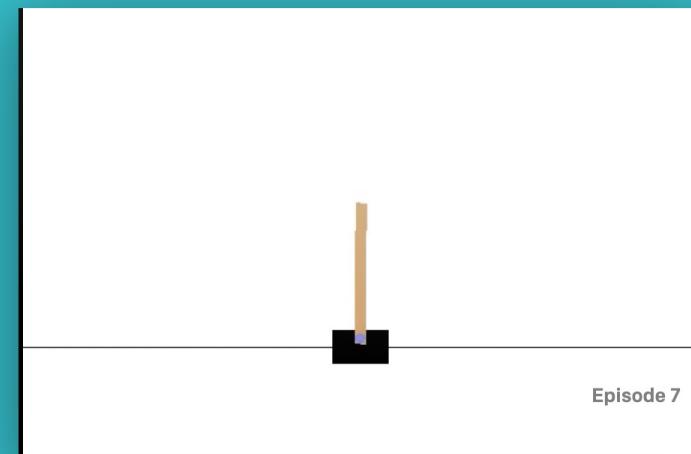
## CartPole-v1

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

*This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson [Barto83].*

**[Barto83]** AG Barto, RS Sutton and CW Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem", *IEEE Transactions on Systems, Man, and Cybernetics*, 1983.

❖ [VIEW SOURCE ON GITHUB](#)



*RandomAgent on CartPole-v1*

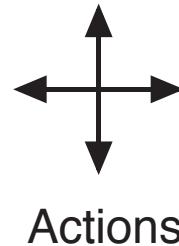
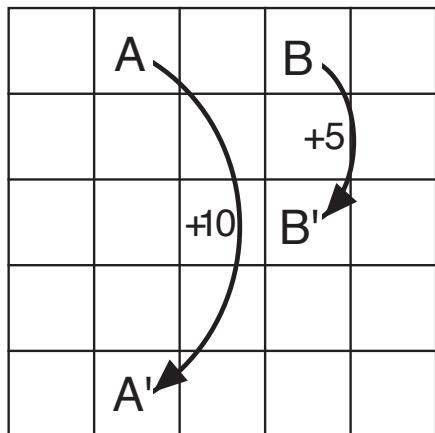
<https://gym.openai.com/envs/CartPole-v1/>

## Recursive calculation of returns

$$\begin{aligned}G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\&= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) \\&= R_{t+1} + \gamma G_{t+1}\end{aligned}$$

## Examples of MDPs

- Grid world environment
- After reaching A (or B) and agent is transferred to the state A' (or B') with the indicated reward



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

value function

# Policy

A policy is a mapping from states to probabilities of selecting each possible action:

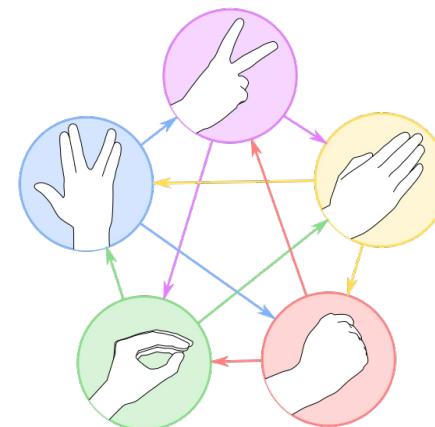
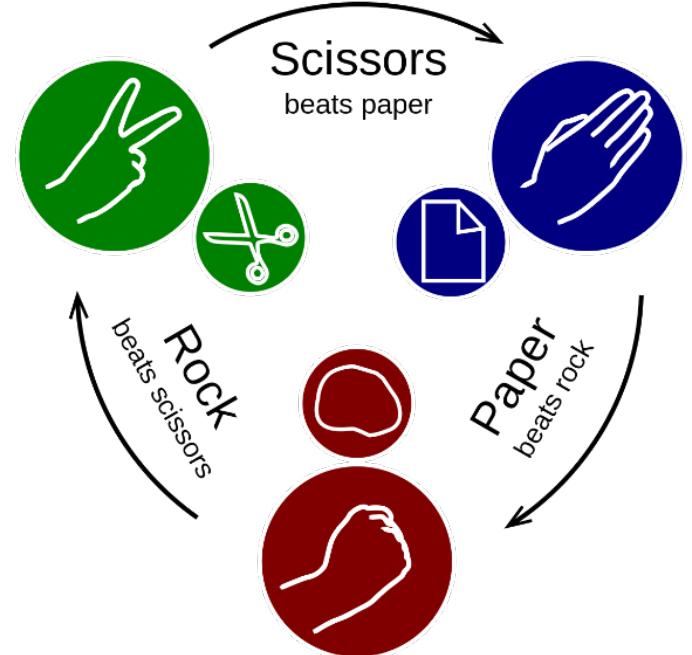
$$\pi(a|s) \doteq \Pr\{A_t = a | S_t = s\}$$

*Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.*

(Barton & Sutton)

## Stochastic / Deterministic policies

- As defined before, a policy is generally **stochastic**, meaning that there are different possible actions taken under the policy with different probabilities for each action
- For example, the optimal policy for rock, paper, scissor is  $(1/3, 1/3, 1/3)$
- In a **deterministic** policy, there is a single action  $a$  that is (always) taken in a state  $s$



## State-Value Function

The value function of a state  $s$  under a policy  $\pi$  is the expected return by

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s], \text{ for all } s \in \mathcal{S}$$

by following  $\pi$  from  $s$

## Action-value

The value of taking action  $a$  in state  $s$  under a policy  $\pi$ , is the expected return

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

by following  $\pi$  after taking action  $a$

## Bellman Equation

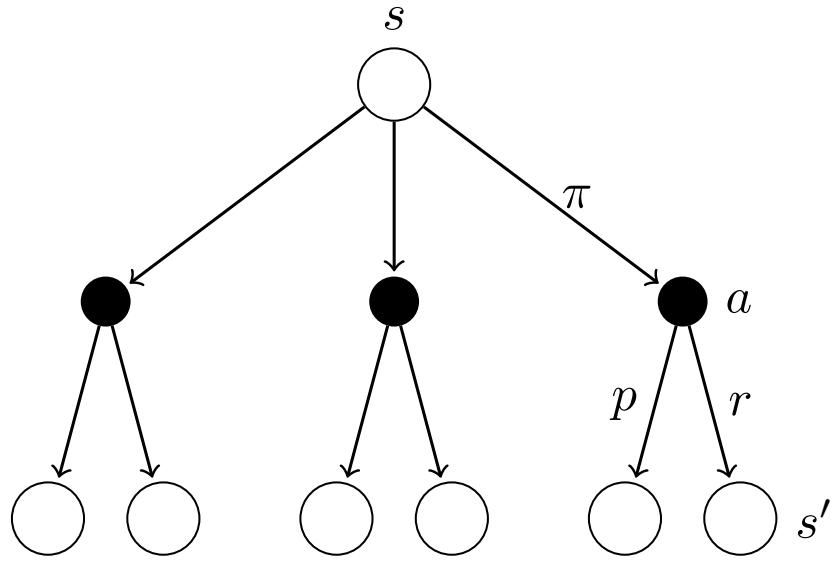
There is a fundamental, recursive calculation for the value-function:

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a) [r + \gamma v_{\pi}(s')], \text{ for all } s \in \mathcal{S}$$

This is the *Bellman equation (for the state-value function)*

## Backup Diagram for the State-Value Function

- This recursive relationship can be visualized using a *backup diagram*:

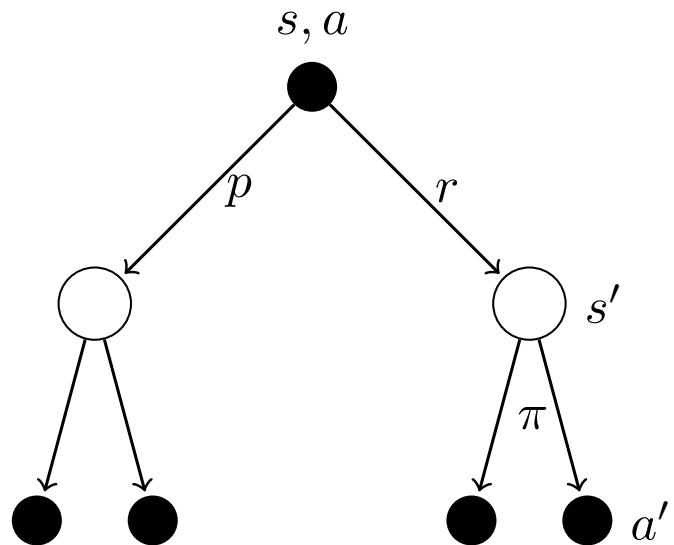


$$v_{\pi}(s) =$$

$$\sum_a \pi(a|s) \cdot$$

$$\sum_{s',r} p(s', r | s, a) [r + \gamma v_{\pi}(s')]$$

# Backup Diagram for the Action-Value Function



How does the equation of  
the action-value function  
look like?

# Optimal Policies

- We can compare policies and say that a policy  $\pi$  is better or equal to another policy  $\pi'$  if its expected return is greater than or equal to that of  $\pi$ .
- There is at least one policy that is better or equal to all the other policies. This is an *optimal* policy.

## Optimal State-Value Function

- the optimal state-value function is then defined as

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s)$$

- and the optimal action-value function as

$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a)$$

- which can also be written using the state-value function:

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

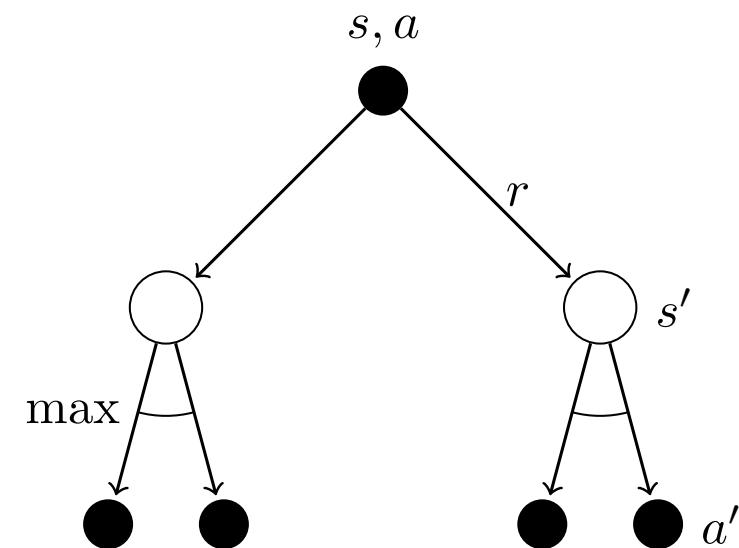
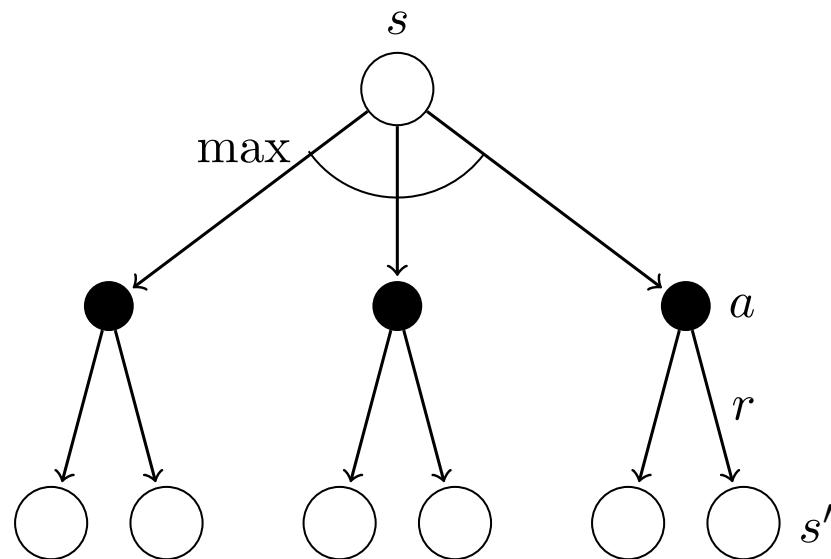
## Bellman Optimality Equation for the State-Value Function

$$\begin{aligned}v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\&= \max_a \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_*(s')]\end{aligned}$$

## Bellman Optimality Equation for the Action-Value Function

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

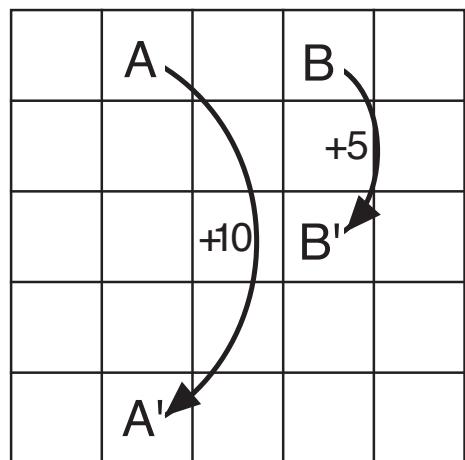
## Backup diagrams for the optimal functions



## Optimal Policies

Given  $v_*$ , any policy that is *greedy* with respect to  $v_*$  is an optimal policy

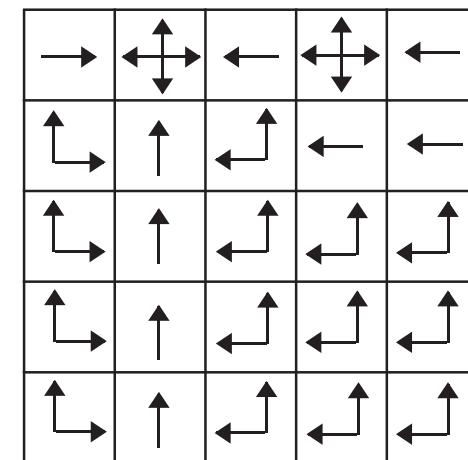
## Example: Gridworld



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

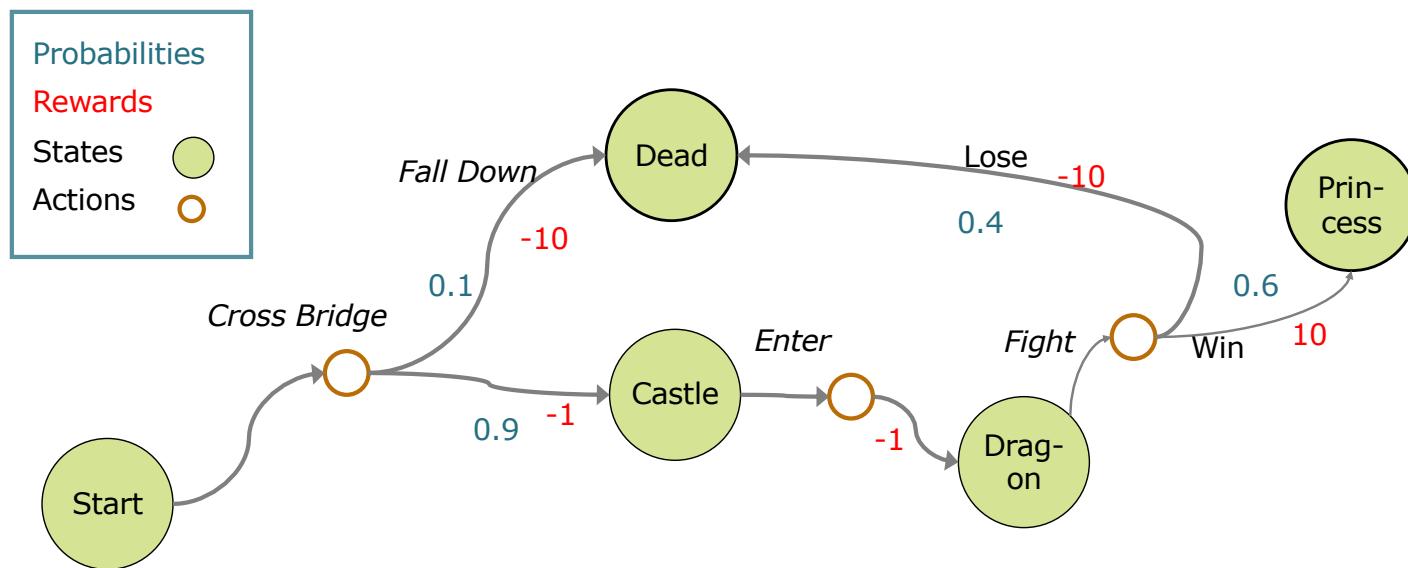
$v_*$



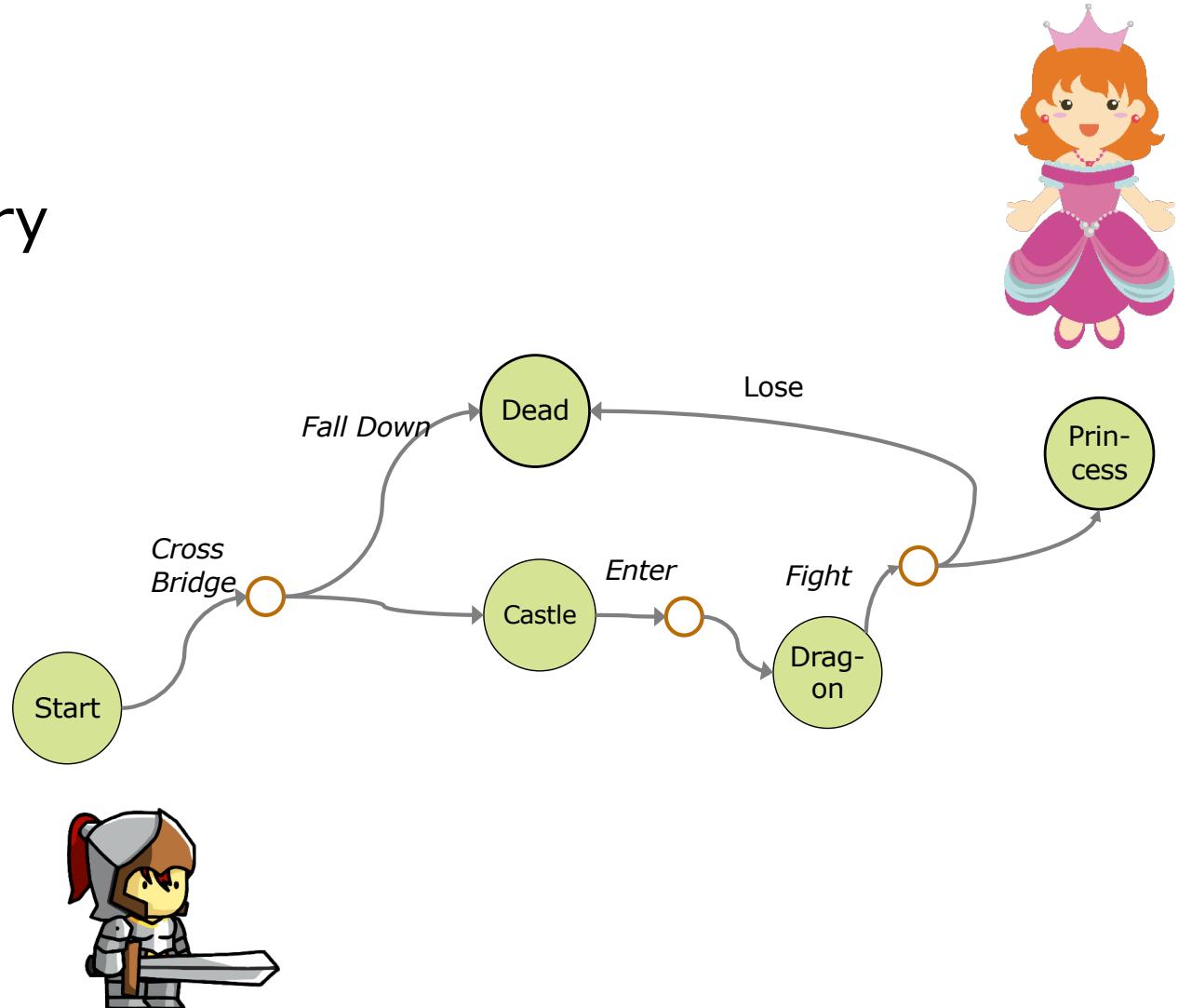
$\pi_*$

## Example: Princess

What are the value functions for the states *Dragon*, *Castle* and *Start*?



# Markov Decision Processes Summary



**Reinforcement Learning**  
October 13, 2022

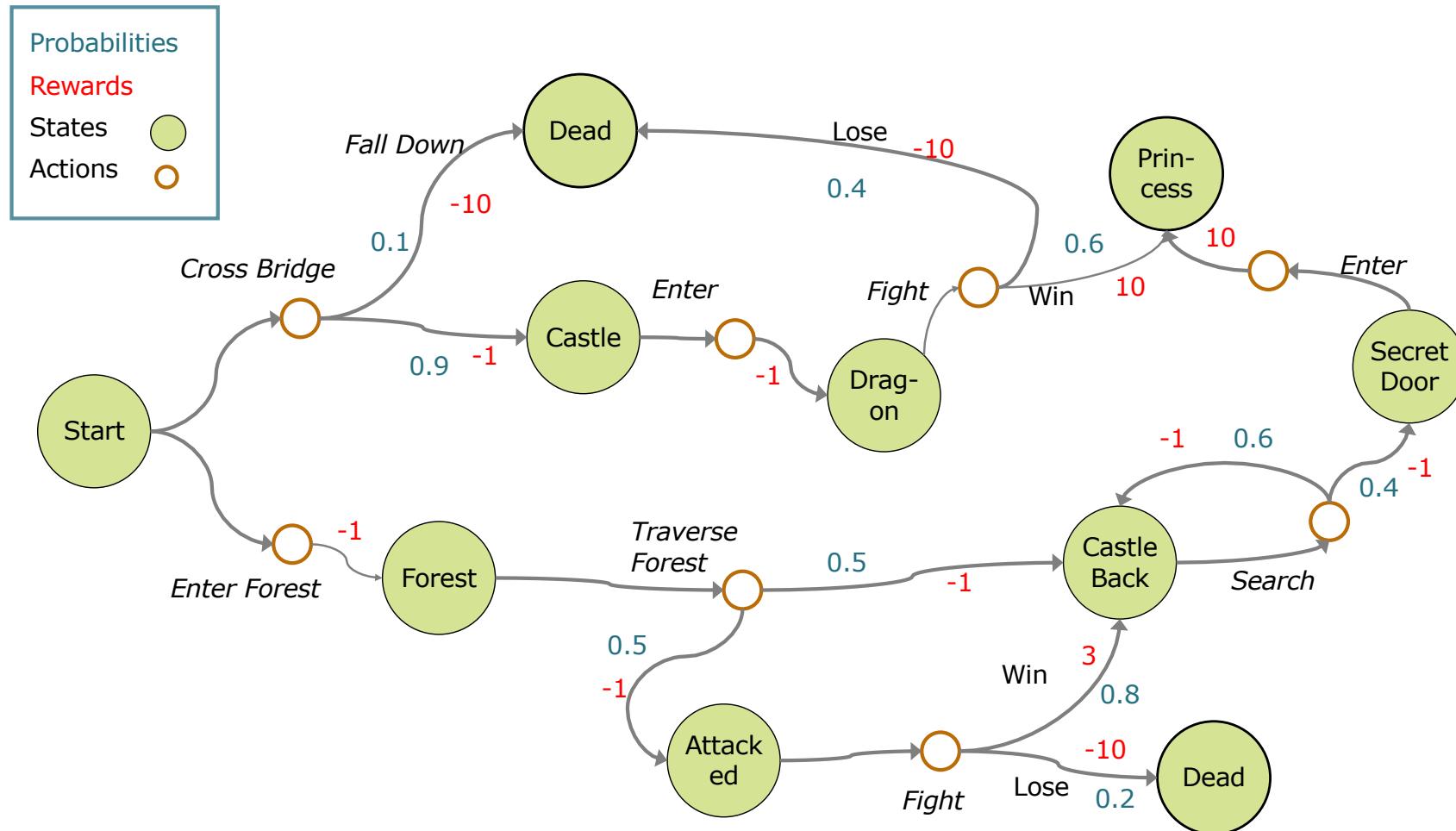
## Definition of MDP

The dynamics of an MDP is defined as

$$p(s', r | s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

(this can be viewed as a function of 4 parameters)

# Markov Decision Process



# Policy and Value Functions

A **policy** is a mapping from states to probabilities of selecting each possible action:

$$\pi(a|s) \doteq \Pr\{A_t = a | S_t = s\}$$

The **state-value function** of a state  $s$  under a policy  $\pi$  is the expected return by following  $\pi$  from  $s$ :

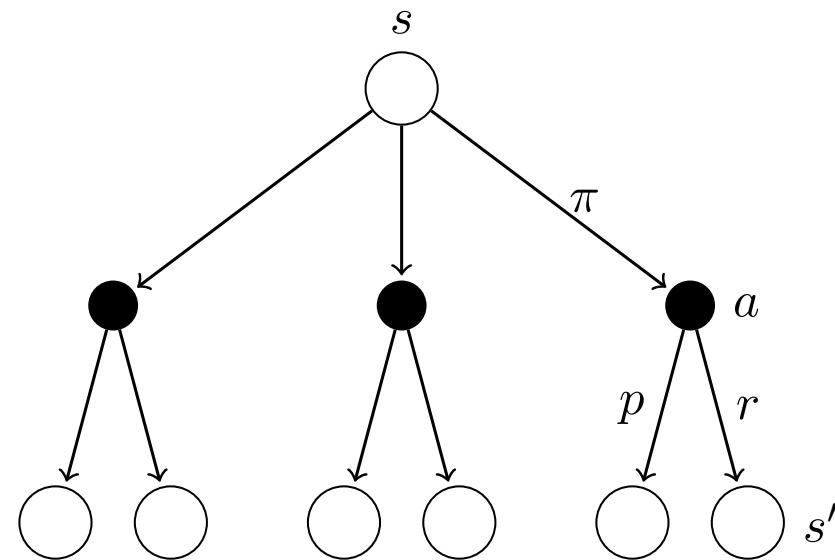
$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s], \text{ for all } s \in \mathcal{S}$$

The **action-value function** is the expected return by taking action  $a$  in state  $s$  and then following  $\pi$ :

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

# Bellman Equation

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a) [r + \gamma v_{\pi}(s')], \text{ for all } s \in \mathcal{S}$$



# Optimal State-Value Function

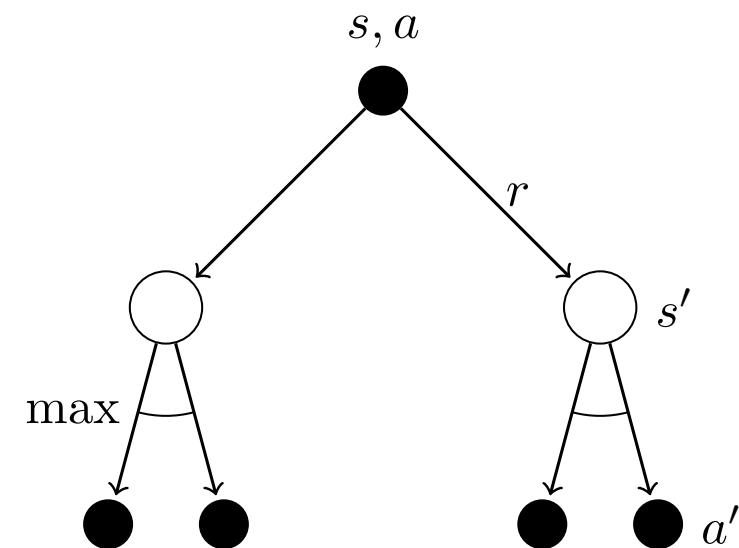
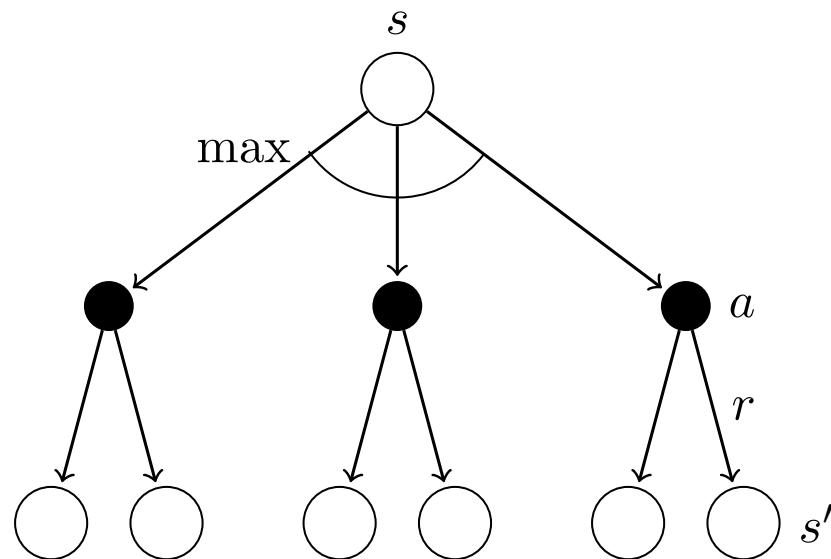
- the optimal state-value function is defined as

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s)$$

- and the optimal action-value function as

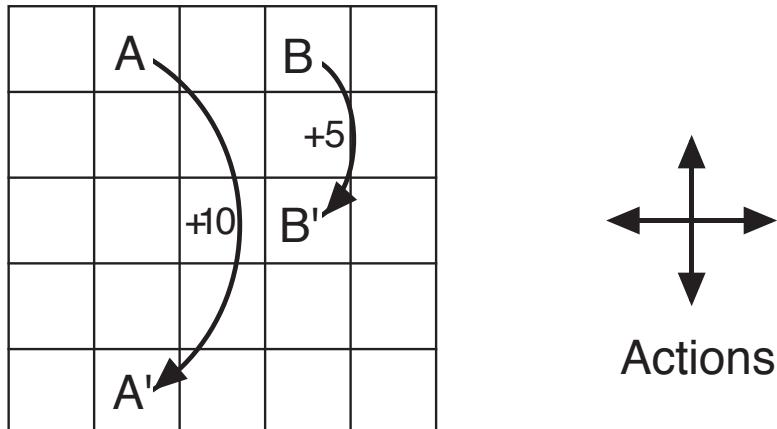
$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a)$$

## Backup diagrams for the optimal functions



# Examples of MDPs

- Grid world environment
- In state A (or B), the agent is transferred to the state A' (or B') with the indicated reward by any action
- Action that take the agent off the grid have reward -1, other actions 0

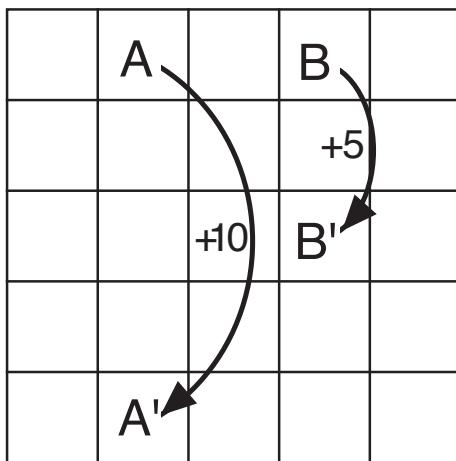


3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

value function for  
random policy and  
discount factor 0.9

## Example: Gridworld

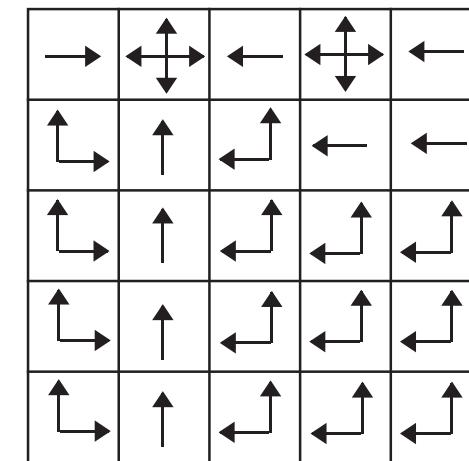
Optimal value function (and policy) for the gridworld problem, using the Bellman Equation



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

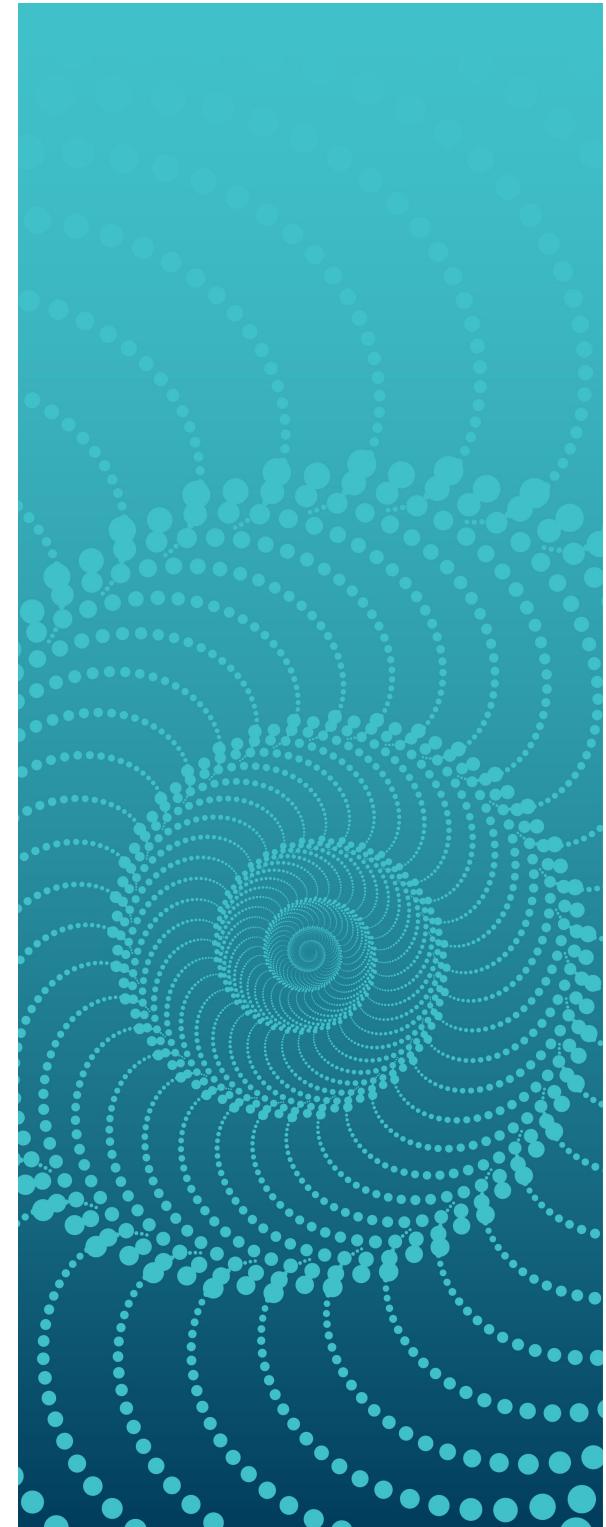
$v_*$



$\pi_*$

# Dynamic Programming

**Reinforcement Learning**  
October 13, 2022



# Learning Objectives

Understand the Bellman equation (better ☺ )

Distinguish between policy evaluation and control

Explain where dynamic programming can be used and where not

Explain how to compute value functions using policy iteration

Understand policy improvement

...

# Classical Dynamic Programming

Some basic ideas for the use of (classical) dynamic programming are:

- Divide and conquer
- Subdivide larger problems into smaller problems
- Solve smaller problems just once (and store solution)
- Make decisions in stages

## Example: 0-1 Knapsack Problem

Given a set of items with weights and values, pack a knapsack with given maximal weight to contain items of maximal total value

# values and corresponding weights

```
v = [20, 5, 10, 40, 15, 25]  
w = [1, 2, 3, 8, 7, 4]
```

# maximal weight

```
W = 10
```



# Knapsack Problem

Solution:

- Recursively calculate a solution with/without the item
- Take the maximal value of both

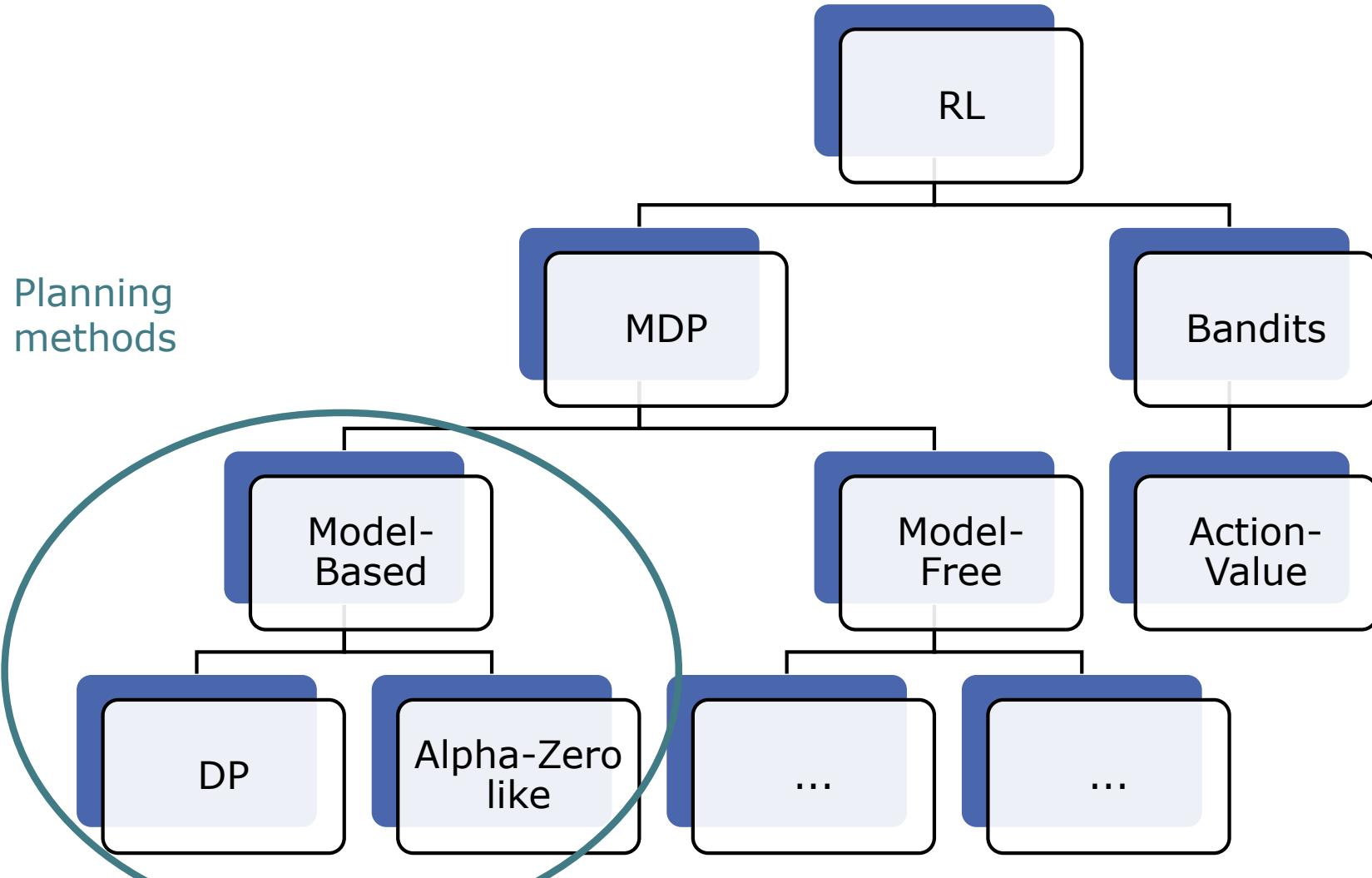
```
def knapsack(v, w, n, W):  
    """  
    v: values, w: weights, n: item to consider, W: weight left  
    """  
    if W < 0:  
        return -sys.maxsize  
    if n < 0 or W == 0:  
        return 0  
    include = v[n] + knapsack(v, w, n - 1, W - w[n])  
    exclude = knapsack(v, w, n - 1, W)  
    return max(include, exclude)
```

# Dynamic Programming in RL

Key ideas:

- Algorithms to compute optimal value functions and policies given a perfect **model** of the MDP
- Finite MDP environment
- Use value functions to organize the search for good policies
- An optimal policy can be obtained from an optimal value function
- DP is often limited in RL due to the need of a model and the large computational expense

# RL Methods



# Bellman Equation

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a) [r + \gamma v_{\pi}(s')], \text{ for all } s \in \mathcal{S}$$

- Calculates the value of a state by following policy  $\pi$
- Could be solved for  $v_{\pi}$  using a system of linear equations, but...
- ... iterative solutions are usually preferred (as they are computationally more efficient)

# Policy Evaluation (Prediction)

- Given a policy, what is its value function?
- Iterative computation using the Bellman equation
- Calculate a sequence of values that approach the correct solution

$$\begin{aligned}v_{k+1}(s) &\doteq \mathbb{E}_{\pi}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\&= \sum_a \pi(a|s) \sum_{s',r} p(s',r \mid s, a) [r + \gamma v_k(s')]\end{aligned}$$

- This is called *iterative policy evaluation*.

## Iterative Policy Evaluation, estimate $v_\pi$

Input: a policy  $\pi$

Initialize:

$V(s) \in \mathbb{R}$  arbitrarily, except  $V(\text{terminal}) = 0$

Loop:

$\Delta \leftarrow 0$

Loop for each  $s \in \mathcal{S}$ :

$v \leftarrow V(s)$

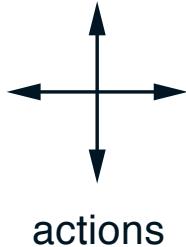
$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r | s,a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until  $\Delta < \theta$

# Example for Policy Evaluation

- Small *Gridworld* Example
- Find terminal state (grey) from any position in minimal number of steps
- Start with random policy
- Initialize all values with 0 (any other value is possible too, except for terminal states which must be 0)



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

$R_t = -1$   
on all transitions

# Policy Evaluation

- The iterative policy evaluation calculates increasingly better estimates of the value function
- (Example from the book, values are given in only 1 digit precision)

$v_k$  for the  
Random Policy

$k = 0$

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

$k = 3$

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

$k = 1$

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

$k = 10$

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

$k = 2$

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

$k = \infty$

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

## Policy Improvement

What happens if we change an action  $a$ , but follow  $\pi$  otherwise?

Recall that

$$q_\pi(s, a) = \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_\pi(s')]$$

If  $\pi$  and  $\pi'$  are deterministic policies with

$$q_\pi(s, \pi'(s)) \geq v_\pi(s), \quad \forall s \in \mathcal{S}$$

(i.e. we take the first action from  $\pi'$  then follow  $\pi$ ), then

$$v_{\pi'}(s) \geq v_\pi(s), \quad \forall s \in \mathcal{S}$$

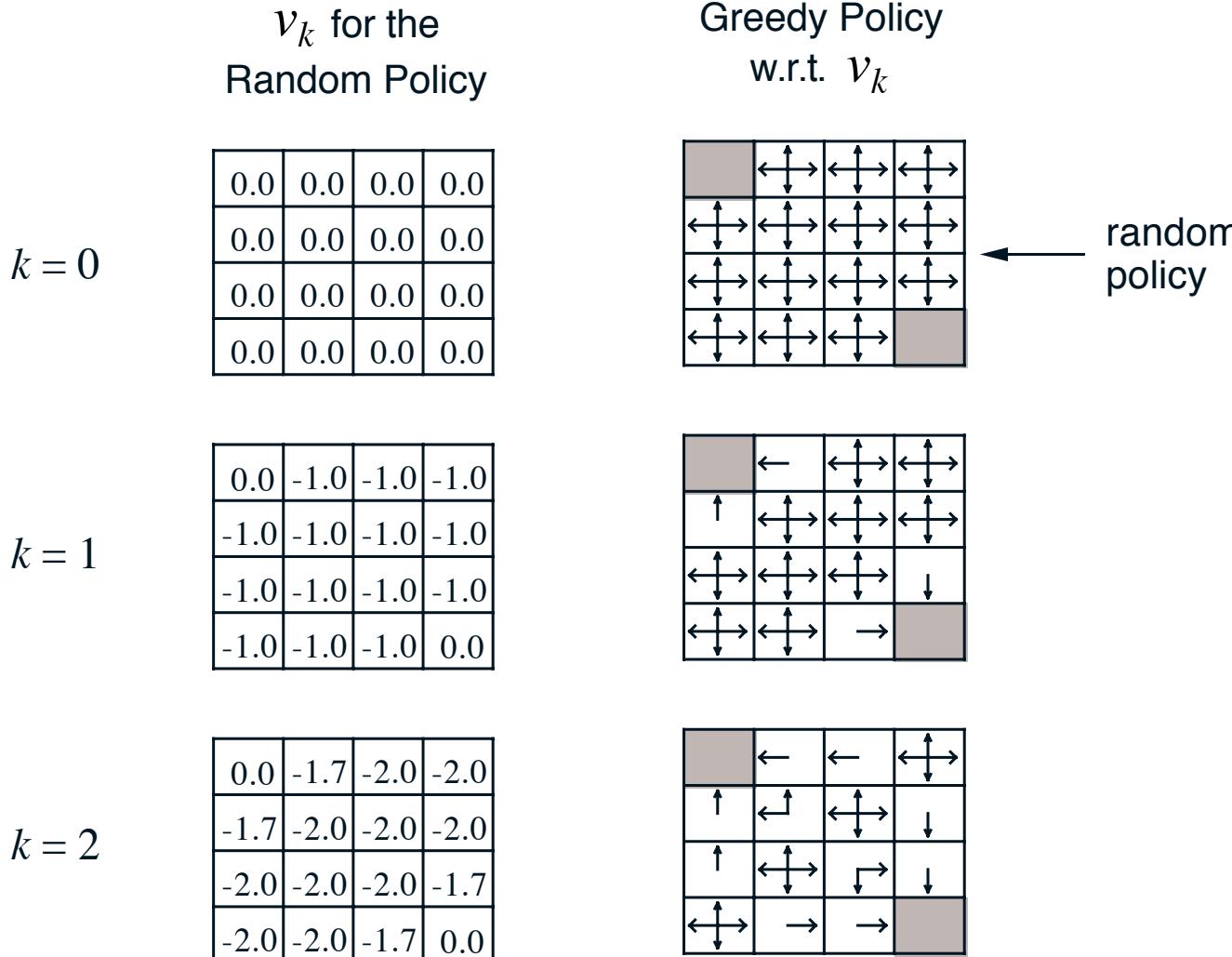
## Policy Improvement

So, if we take a greedy action  $a$ , this defines a new policy

$$\begin{aligned}\pi'(s) &\doteq \operatorname{argmax}_a q_\pi(s, a) \\ &= \operatorname{argmax}_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \operatorname{argmax}_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')]\end{aligned}$$

which is as good as, or better than the old policy  $\pi$

# Policy Improvement



# Policy Improvement

$k = 3$

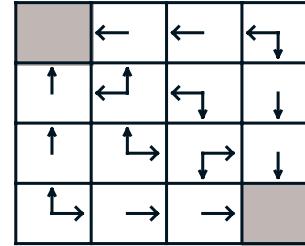
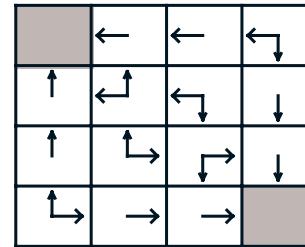
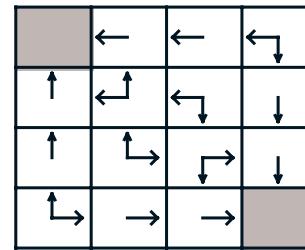
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

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

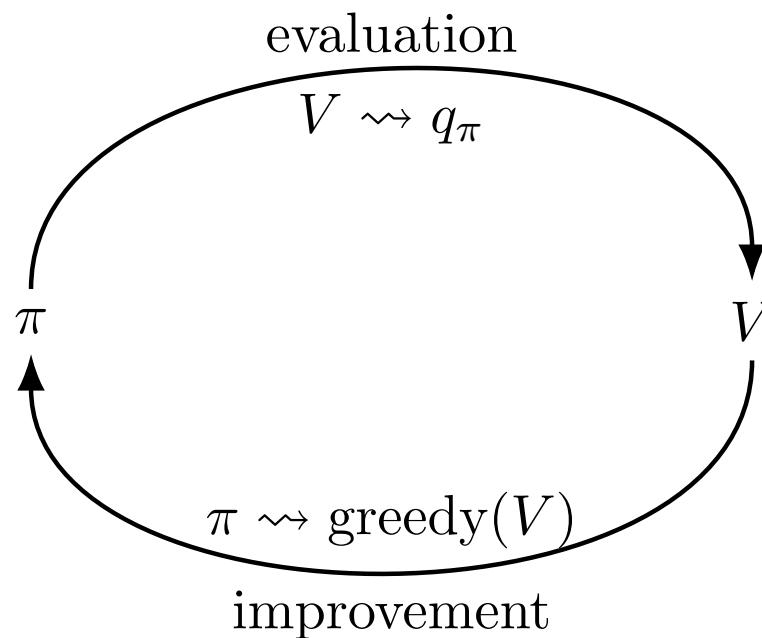


optimal  
policy

# Policy Iteration

Once a policy  $\pi$  has been improved using  $v_\pi$  to yield a better policy  $\pi'$ , we can compute  $v_{\pi'}$  and improve it again:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$



# Policy Iteration

Policy Iteration, estimate  $\pi \approx \pi_*$

Initialize:

$V(s) \in \mathbb{R}$  and  $\pi(s)$  arbitrarily, except  $V(\text{terminal}) = 0$

Loop:

Policy Evaluation:

estimate  $V \approx v_\pi$  (see previous algorithm)

Policy Improvement:

$policy-stable \leftarrow true$

For each  $s \in \mathcal{S}$ :

$old-action \leftarrow \pi(s)$

$\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s'} p(s', r | s, a) [r + \gamma V(s')]$

if  $old-action \neq \pi(s)$ , then  $policy-stable \leftarrow false$

until  $policy-stable$

return  $V \approx v_*, \pi \approx \pi_*$

## Value Iteration

- Does the policy evaluation need to converge? (see gridworld example) ?
- If we stop policy evaluation after one sweep, we obtain an algorithm called *value iteration*
- The update for this can be written as

$$v_{k+1}(s) = \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_k(s')] , \quad \forall s \in \mathcal{S}$$

# Value Iteration

Value Iteration, estimate  $\pi \approx \pi_*$

Initialize:

$V(s) \in \mathbb{R}$  arbitrarily, except  $V(\text{terminal}) = 0$

Loop:

$\Delta \leftarrow 0$

Loop for each  $s \in \mathcal{S}$ :

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s',r} p(s',r \mid s,a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until  $\Delta < \theta$

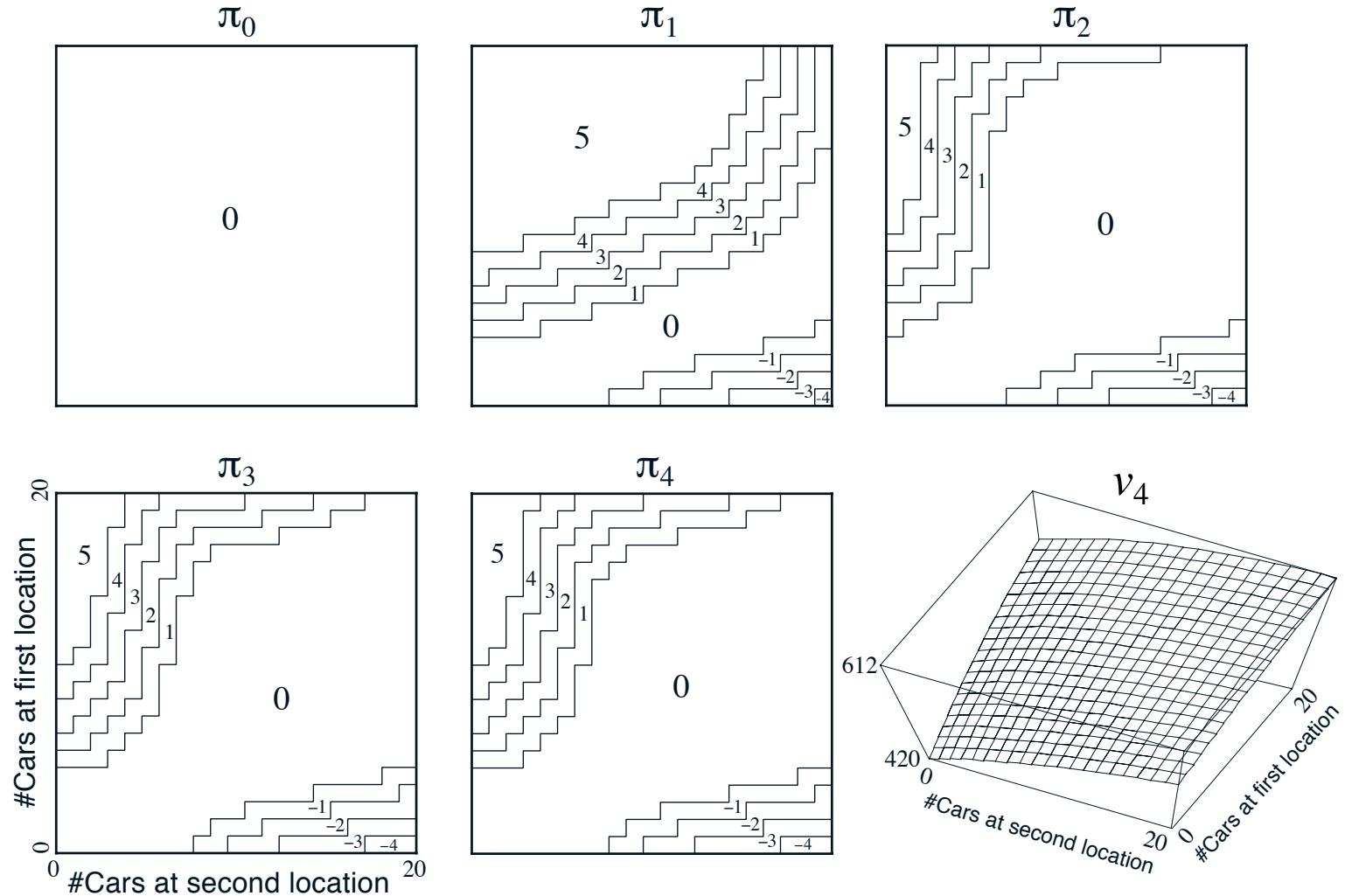
Output deterministic policy, such that

$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r \mid s,a) [r + \gamma V(s')]$

## **Example: Jack's Car Rental**

- 2 car rental locations with max. 20 cars each
- Cars do not have to be returned to the same place
- If cars are available, they can be rented out for \$10, if a customer arrives
- Jack can move up to 5 cars during the night at a cost of \$2 (this is the action)
  
- (Poisson model of how the cars are rented and returned, not equal in both locations)
- Policy: Move n cars from first to second location
- What is the optimal policy? I.e., how many cars should be moved in each different state?

# Example: Jack's Car Rental



# **Asynchronous vs. Synchronous Programming**

Synchronous DP:

- Update value function in sweeps
- All new values are calculated from the old values
- (Generally requires 2 arrays: one for the old values and one for the new ones)

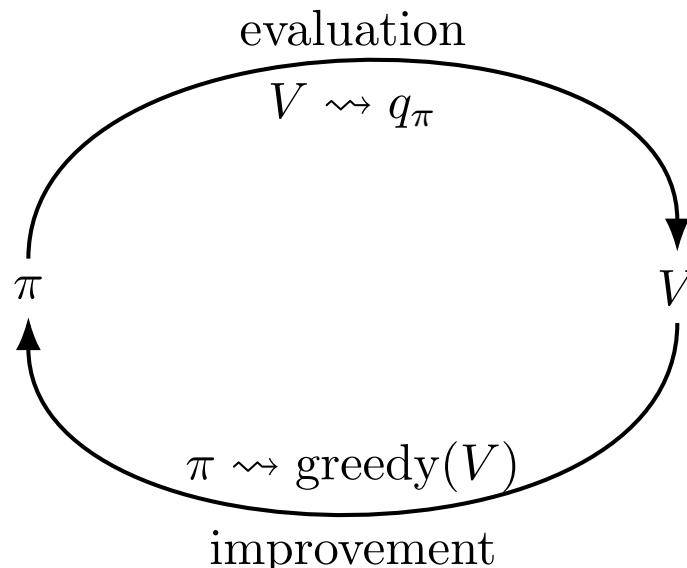
Asynchronous DP:

- Update value function in any order
- Update values in place
- All new values are calculated from the current values

# Generalized Policy Iteration

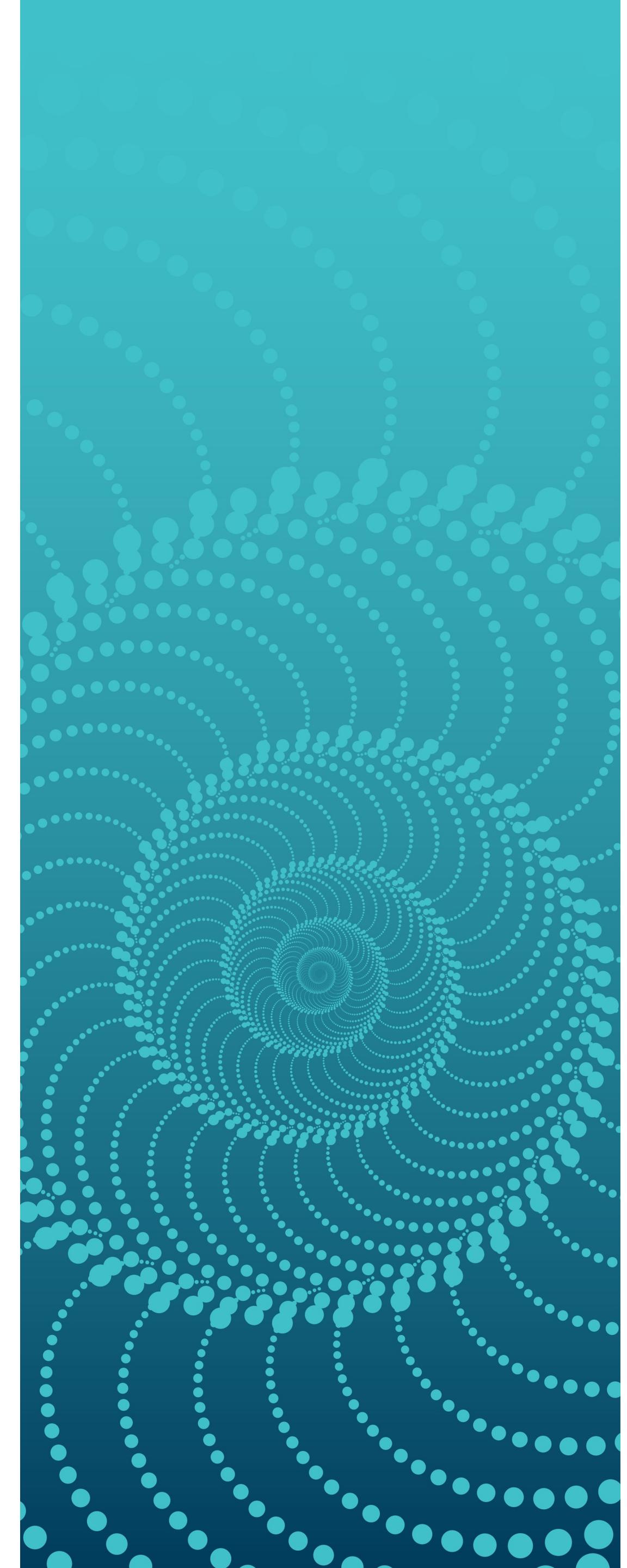
Generalized Policy Iteration (GPI):

- Interaction between *policy-evaluation* and *policy-improvement*
- (*independent of the granularity of the processes, i.e., if they are run just for one step or until convergence*)
- Many reinforcement learning methods can be described as GPI



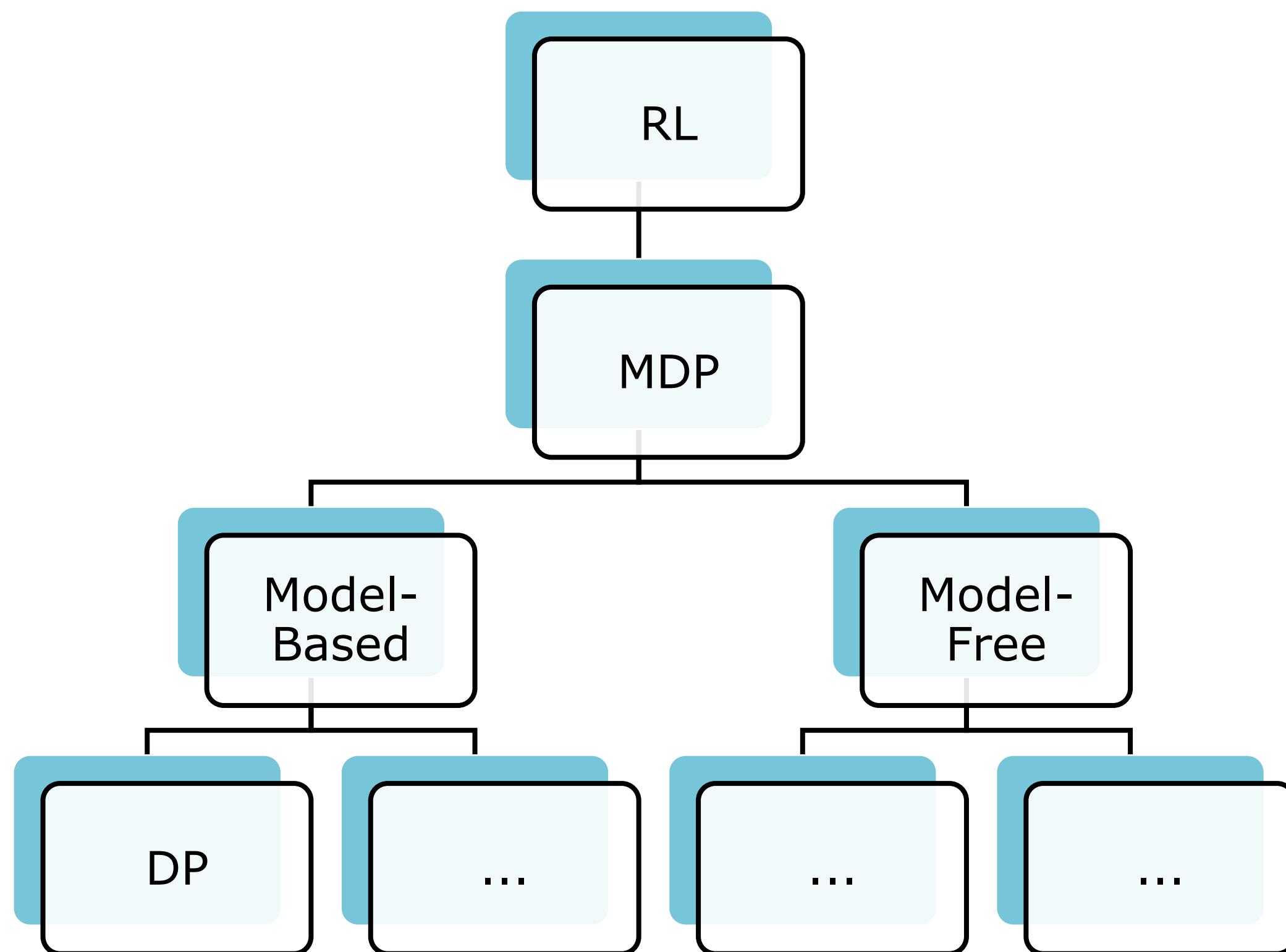
# Dynamic Programming Summary

**Reinforcement Learning**  
October 20, 2022



# Dynamic Programming in RL

Algorithms to compute optimal value functions and policies given a perfect **model** of the MDP



# Policy Evaluation (Prediction)

- Iteratively calculates the value function of a given policy:

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

Iterative Policy Evaluation, estimate  $v_\pi$

Input: a policy  $\pi$

Initialize:

$V(s) \in \mathbb{R}$  arbitrarily, except  $V(\text{terminal}) = 0$

Loop:

$\Delta \leftarrow 0$

Loop for each  $s \in \mathcal{S}$ :

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r | s, a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until  $\Delta < \theta$

# Policy Improvement

Calculate a new policy from a value function by using only greedy actions

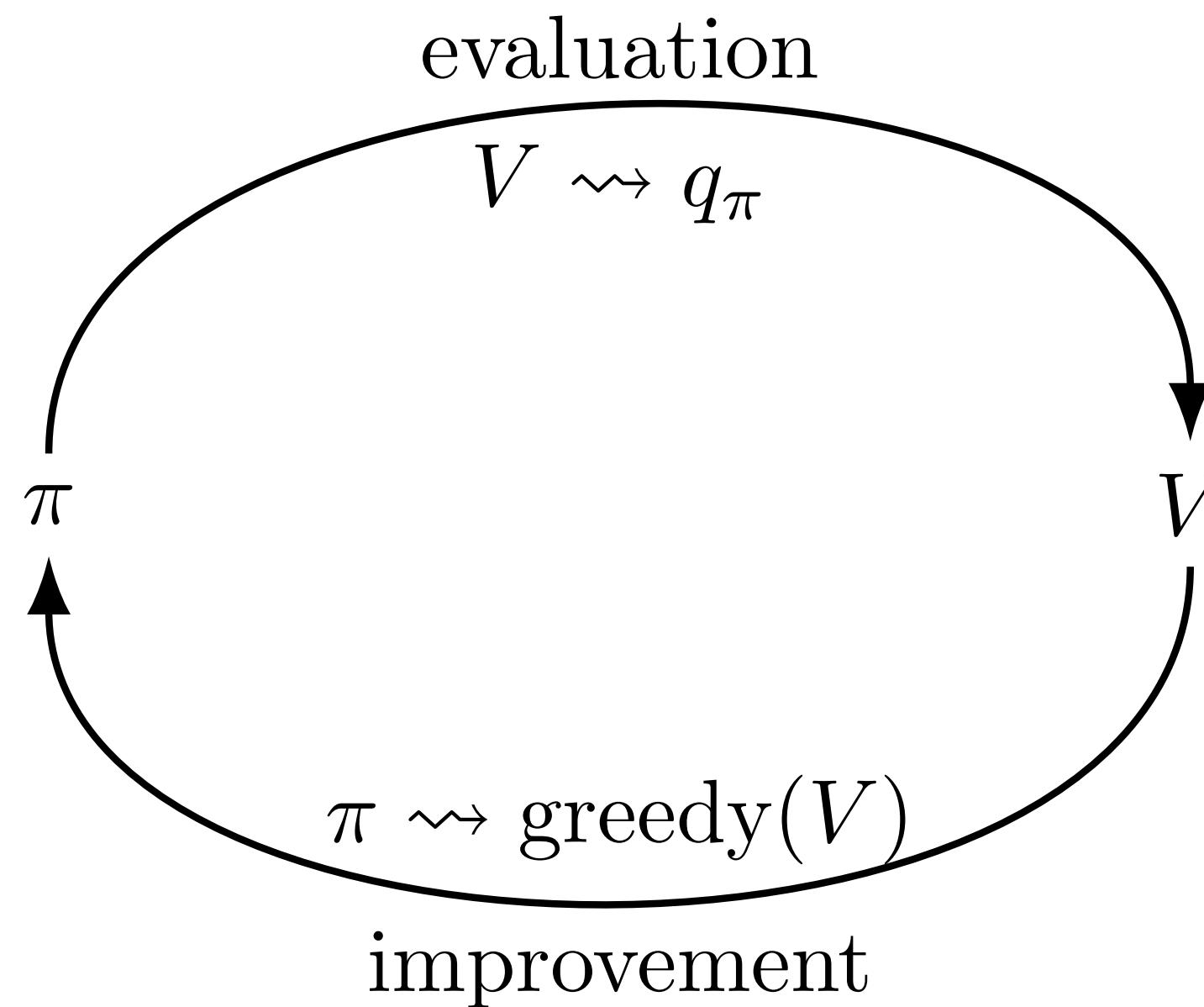
$$\begin{aligned}\pi'(s) &\doteq \operatorname{argmax}_a q_\pi(s, a) \\ &= \operatorname{argmax}_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_\pi(s')]\end{aligned}$$

# Policy Iteration

Repeatedly calculate

- the value function from a policy, and then
- a better policy (greedy) from the value function

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$



# Value Iteration

- Calculate value function after evaluating the policy ones (for all states)

$$v_{k+1}(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_k(s')], \quad \forall s \in \mathcal{S}$$

Value Iteration, estimate  $\pi \approx \pi_*$

Initialize:

$V(s) \in \mathbb{R}$  arbitrarily, except  $V(\text{terminal}) = 0$

Loop:

$\Delta \leftarrow 0$

Loop for each  $s \in \mathcal{S}$ :

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until  $\Delta < \theta$

Output deterministic policy, such that

$\pi(s) = \operatorname{argmax}_a \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$

**Reinforcement Learning**  
October 20, 2022



**{{DN:Hierarchy|Organisation Bezeichnung Spez.EN|ID:32|Hierarchy:1}}**

Research

**Prof. Dr. Thomas Koller**

Lecturer

Phone direct +41 41 757 68 32

[thomas.koller@hslu.ch](mailto:thomas.koller@hslu.ch)

# Monte Carlo Methods



**Reinforcement Learning**  
October 27, 2022



# Learning Objectives

- Understand how value functions are estimated with Monte-Carlo methods
- Understand how Monte-Carlo methods fit into the GPI pattern
- Understand why exploration is needed in MC methods
- Understand the difference between on-policy and off-policy methods
- Understand importance sampling
- Understand Monte Carlo prediction and control using off-policy methods

# Introduction

- Dynamic Programming require a complete model of the environment and a sweep through all states for an update
- We now look at methods that do not need a model but learn from experience which can be actual or simulated
- While a simulated experience still needs a kind of model, it only needs it to generate sample transitions

# Monte Carlo Methods

- Monte Carlo (MC) methods look at whole episodes and then average the complete returns
- The tasks must be episodic, so that all episodes eventually terminate (with the used policy!)
- Value estimation and policies are only changed **on the completion of an episode**
- (While the term is often used for other methods with some random components, we use it here for methods based on averaging complete returns)

# Monte Carlo Methods

- The idea of general policy iterations (GPI) from dynamic programming can be adapted to MC methods
- We first consider the prediction problem of estimating the value function(s) for a known policy
- We can then look at the control problem, i.e., finding the best policy

# Monte Carlo Prediction

- We want to estimate  $v_\pi(s)$ , the value of a state  $s$  under policy  $\pi$
- Given is a set of episodes (that pass through  $s$ )
- An episode might pass multiple times through  $s$ , we will only consider it once, this is known as *first-visit MC*
- We average the returns of all episodes

# Monte Carlo Prediction (first visit)

Monte Carlo Prediction for estimating  $v_\pi$

Input: a policy  $\pi$

Initialize:

$V(s) \in \mathbb{R}$  (arbitrarily)

$\text{Returns}(s) \leftarrow$  an empty list, for all  $s \in \mathcal{S}$

Loop forever:

Generate episode following  $\pi$ :  $S_0, A_0, R_0, S_1, \dots, R_T$

$G \leftarrow 0$

Loop for each step of the episode,  $t = T - 1, T - 2, \dots, 0$ :

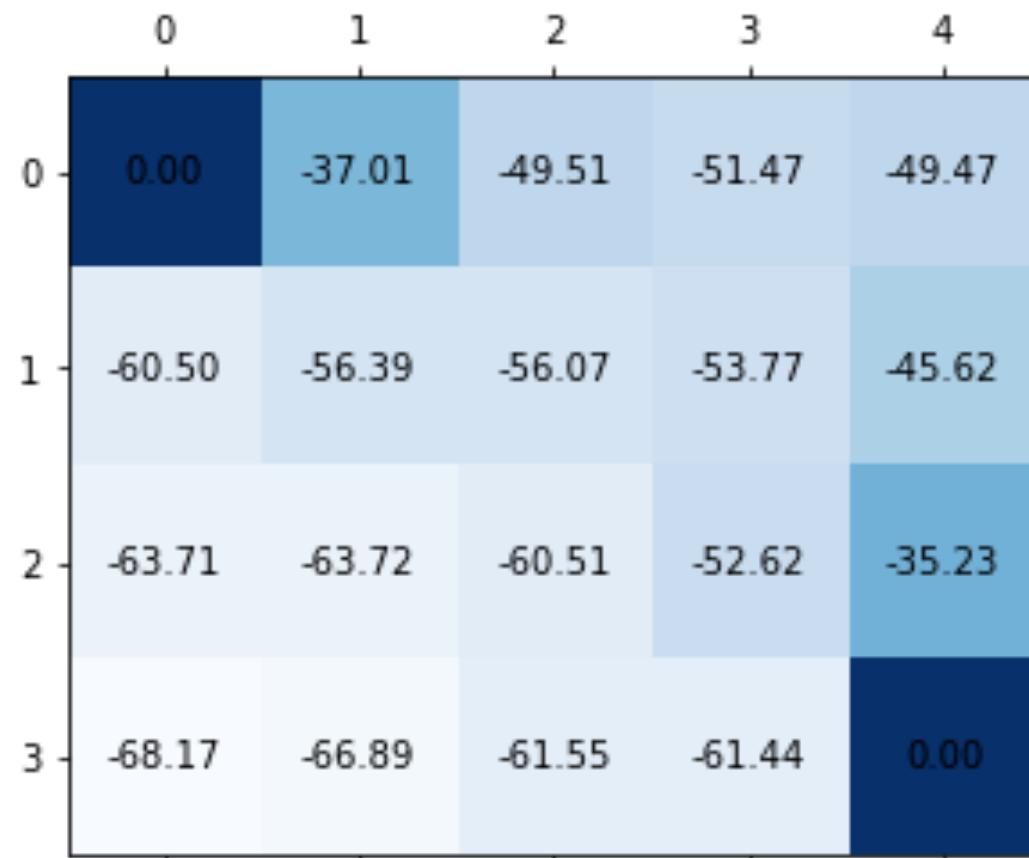
$G \leftarrow \gamma G + R_{t+1}$

Unless  $S_t$  appears in  $S_{t-1}, \dots, S_0$ :

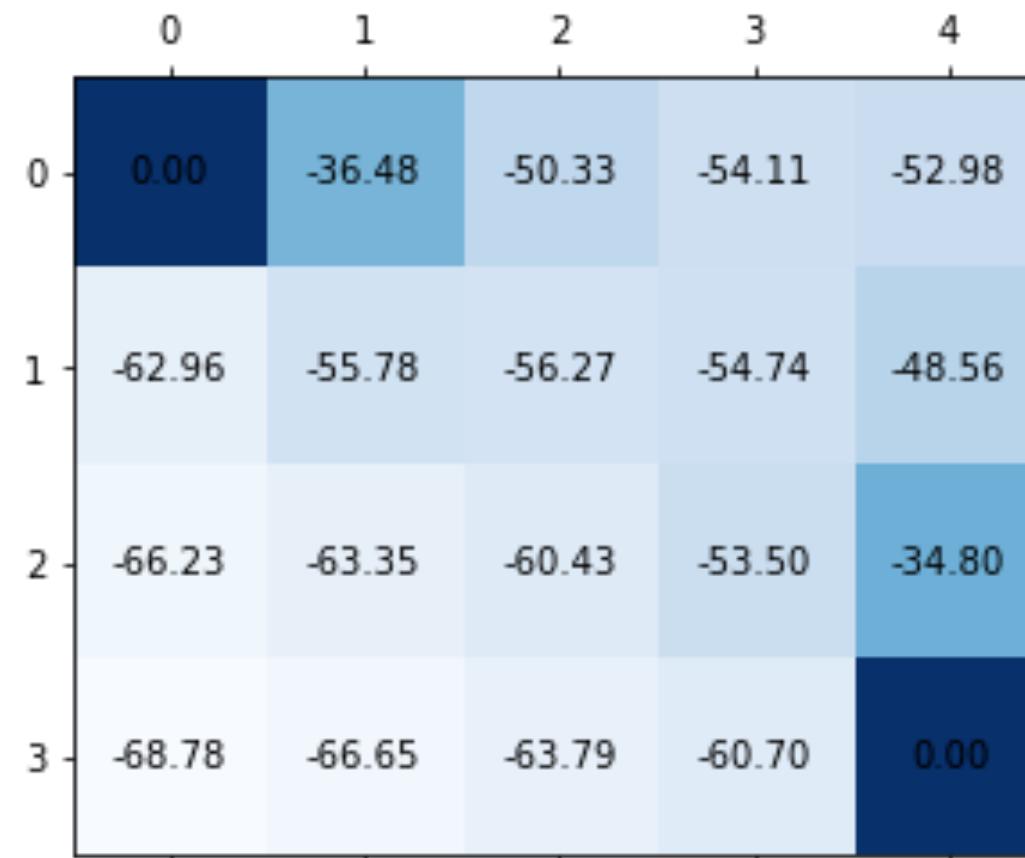
Append  $G$  to  $\text{Returns}(S_t)$

$V(S_t) \leftarrow \text{average}(\text{Returns}(S_t))$

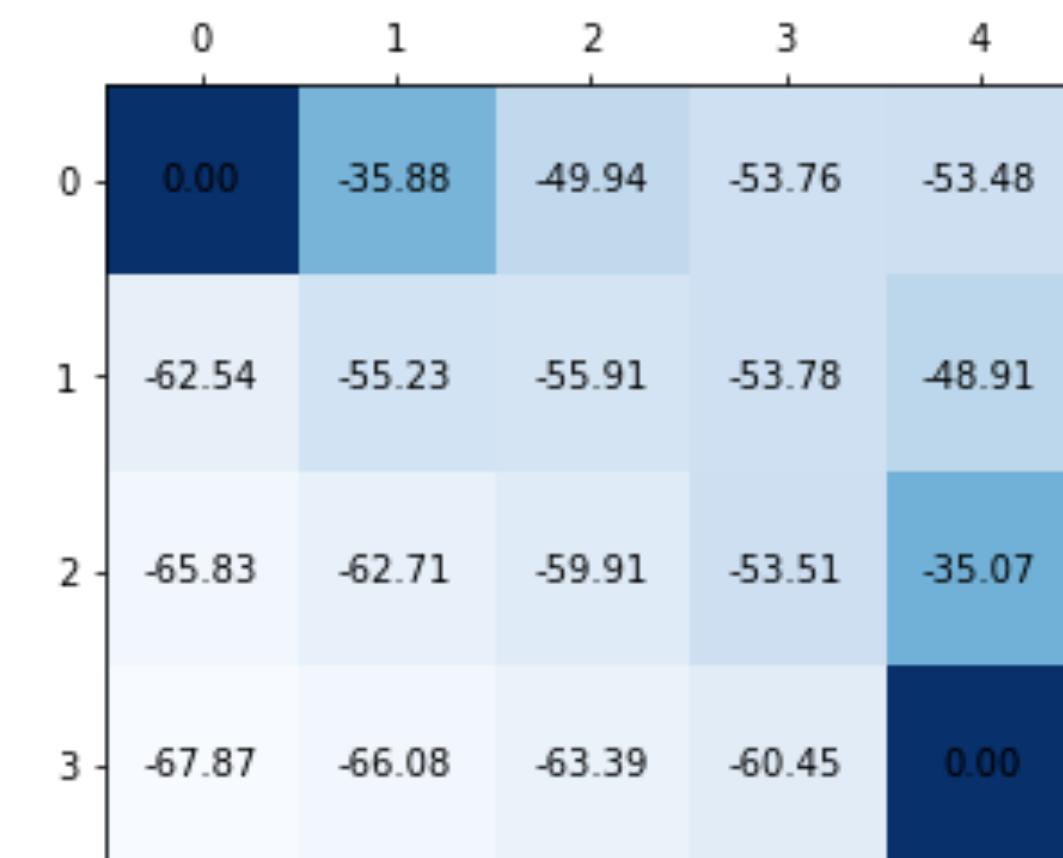
# Monte Carlo Prediction



1'000 episodes



10'000 episodes



100'000 episodes

Example on gridworld (with interior walls) with random policy

# Black Jack Example

Ace can be 11 or 1, if the ace can be used as 11 it is called useable (and counted as 11)

State of the system (from the point of the player):

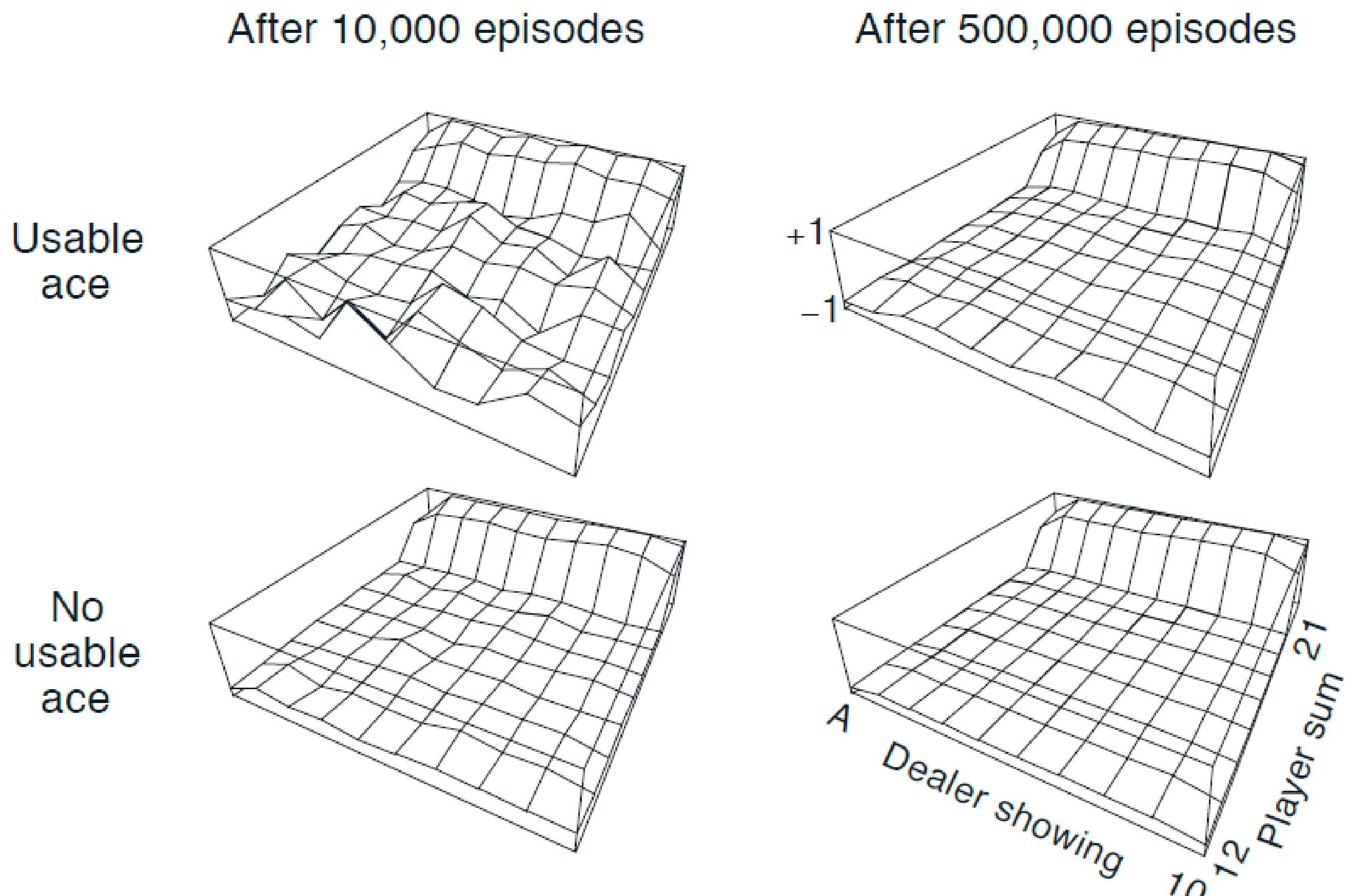
- current sum of the player
- card of the dealer
- ace usable or not



# Black Jack Example

- Player vs. Dealer
- 2 cards dealt each at the beginning, one card of the dealer is shown
- Actions: player can request another card (hit) or stop (stick)
- Dealer has fixed strategy: stick with 17 or more, hit otherwise
- Rewards: -1 (loss), 0 (draw), 1 (win) at end
- Episodic tasks

# Blackjack



Value function for policy that sticks with 20 or 21 points

# Action-Value Function Estimation

- With a model, we can determine the best action by
  - using the state-value function
  - looking one step ahead (for all possible actions)
  - choose the best action (reward and next state value)
- Without a model, we cannot look ahead, so we want to estimate the action-value function instead of the state-value function:

$$q_{\pi}(s, a)$$

- I.e., the expected return from starting in  $s$  with action  $a$  and following  $\pi$

# Prediction of Action Values

The prediction algorithm can be adapted for action values

However:

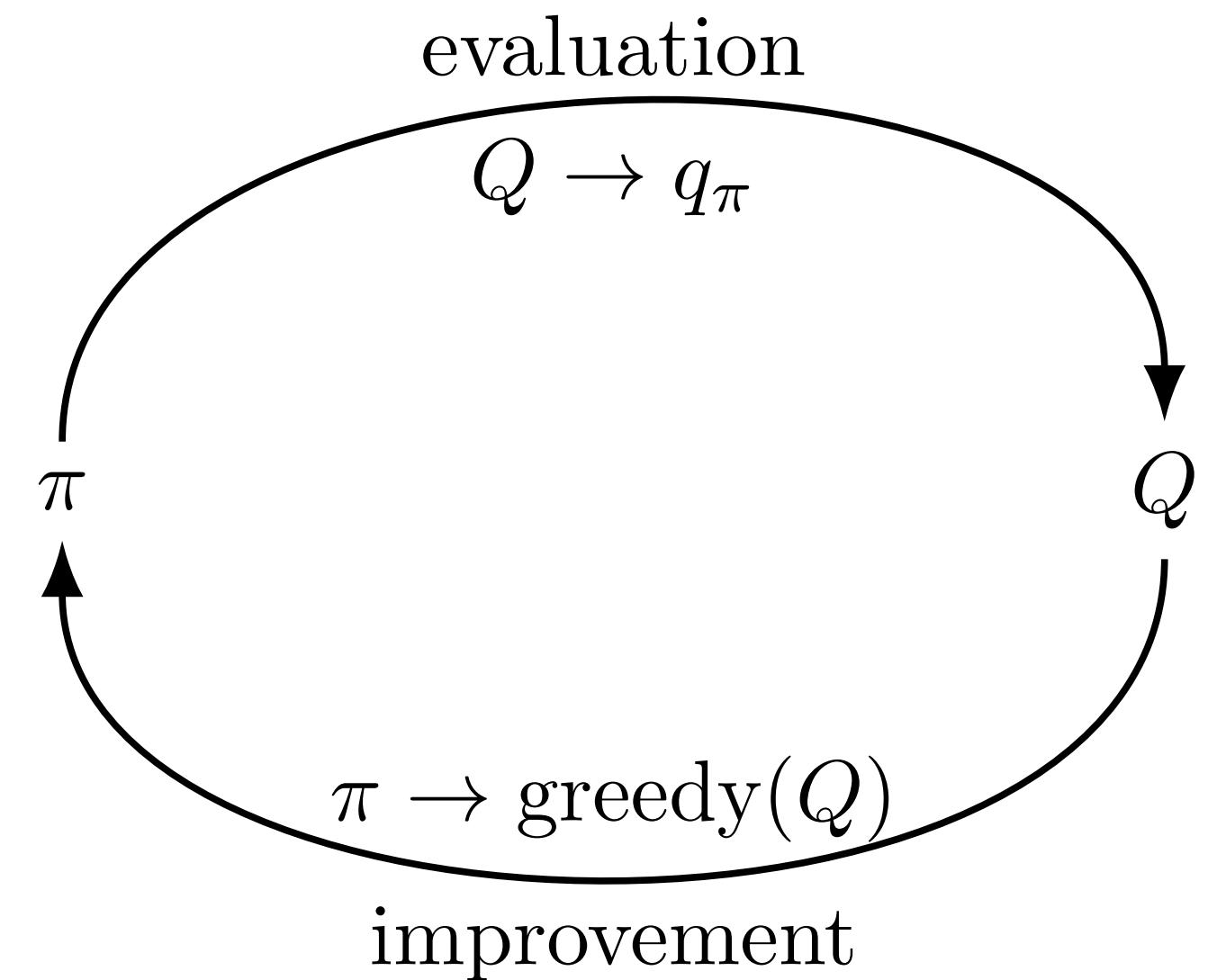
- Some state-action pairs might never be visited

Solutions:

- Exploring Starts: Every state-action pair has a nonzero probability to be selected as start, or
- Maintain exploration by using a policy that has a nonzero probability for every action (this is called an epsilon-soft policy)

# Monte Carlo Control

- Generalized policy iteration (GPI) can be used with Monte Carlo Prediction
- Policy Estimation is done as described above
- Policy Improvement is done by choosing a greedy policy with respect to the action-value function



$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} q_{\pi_*}$$

## On-policy first-visit MC control, estimates $\pi \approx \pi_*$

Input: (small)  $\epsilon > 0$

Initialize:

$\pi \leftarrow$  an arbitrary  $\epsilon$ -soft policy

$Q(s, a) \in \mathbb{R}$  (arbitrarily, for example = 0)

$Returns(s, a) \leftarrow$  empty list

Loop forever: (for each episode)

Generate an episode following  $\pi$ :  $S_0, A_0, R_0, S_1, \dots, R_T$

$G \leftarrow 0$

Loop for each step of the episode,  $t = T - 1, T - 2, \dots, 0$ :

$G \leftarrow \gamma G + R_{t+1}$

Unless  $(S_t, A_t)$  appears in  $(S_{t-1}, A_{t-1}), \dots, (S_0, A_0)$ :

Append  $G$  to  $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$  average( $Returns(S_t, A_t)$ )

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$ , ties broken arbitrarily

For all  $a \in \mathcal{A}(S_t)$ :

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)| & \text{otherwise} \end{cases}$$

# On policy vs. off policy algorithm

- On-policy algorithms learn a policy while following this policy in the algorithm
- Off-policy algorithms learn a policy different from the one used to generate the data
- On-policy algorithms require a *soft* policy, which means that

$$\pi(a|s) > 0, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}$$

# Importance Sampling

- epsilon-soft policies are a compromise:
- we use an almost optimal policy to learn action values while still exploring
- We would like to use 2 policies:
  - A target policy  $\pi$  that we want to learn
  - A behavior policy  $b$  that we use for exploring
- constraint: every action under  $\pi$  must also be taken under  $b$  with non-zero probability

# Importance Sampling

The probability of a state trajectory under a policy is:

$$\begin{aligned} & \Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T | S_t, A_{t:T-1} \sim \pi\} \\ &= \pi(A_t | S_t) p(S_{t+1} | S_t, A_t) \pi(A_{t+1} | S_{t+1}) \cdots p(S_T | S_{T-1}, A_{T-1}) \\ &= \prod_{k=1}^{T-1} \pi(A_k | S_k) p(S_{k+1} | S_k, A_k) \end{aligned}$$

So the relative probability of two policies is

$$\begin{aligned} \rho_{t:T} &\doteq \frac{\prod_{k=1}^{T-1} \pi(A_k | S_k) p(S_{k+1} | S_k, A_k)}{\prod_{k=1}^{T-1} b(A_k | S_k) p(S_{k+1} | S_k, A_k)} \\ &= \prod_{k=t}^{T-1} \frac{\pi(A_k | S_k)}{b(A_k | S_k)} \end{aligned}$$

# Corection of averaged returns

Ordinary importance sampling:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}$$

Weighted importance sampling  
(derivation in the book)

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}}$$

where  $\mathcal{T}(s)$  is the set of all timesteps  
that pass-through s

## Off-policy MC prediction, estimates $Q \approx q_\pi$

Input: a policy  $\pi$

Initialize, for all  $s \in \mathcal{S}, a \in \mathcal{A}$ :

$$Q(s, a) \in \mathbb{R} \text{ (arbitrarily, for example = 0)}$$

$$C(s, a) \leftarrow 0$$

Loop forever (for each episode):

$b \leftarrow$  any soft policy compatible with  $\pi$

Generate an episode following b:  $S_0, A_0, R_0, S_1, \dots, R_T$

$$G \leftarrow 0$$

$$W \leftarrow 1$$

Loop for each step of the episode,  $t = T - 1, T - 2, \dots, 0$ , while  $W \neq 0$ :

$$G \leftarrow \gamma G + R_{t+1}$$

$$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$$

$$W \leftarrow W \frac{\pi(A_t | S_t)}{b(A_t | S_t)}$$

## Off-policy MC control, estimates $\pi \approx \pi_*$

Initialize, for all  $s \in \mathcal{S}, a \in \mathcal{A}$ :

$$Q(s, a) \in \mathbb{R} \text{ (arbitrarily, for example = 0)}$$

$$C(s, a) \leftarrow 0$$

$$\pi(s) \leftarrow \operatorname{argmax}_a Q(s, a) \quad (\text{with ties broken consistently})$$

Loop forever (for each episode):

$b \leftarrow$  any soft policy

Generate an episode following b:  $S_0, A_0, R_0, S_1, \dots, R_T$

$$G \leftarrow 0$$

$$W \leftarrow 1$$

Loop for each step of the episode,  $t = T - 1, T - 2, \dots, 0$ :

$$G \leftarrow \gamma G + R_{t+1}$$

$$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$$

$$\pi(a|S_t) \leftarrow \operatorname{argmax}_a Q(S_t, A_t) \quad (\text{with ties broken consistently})$$

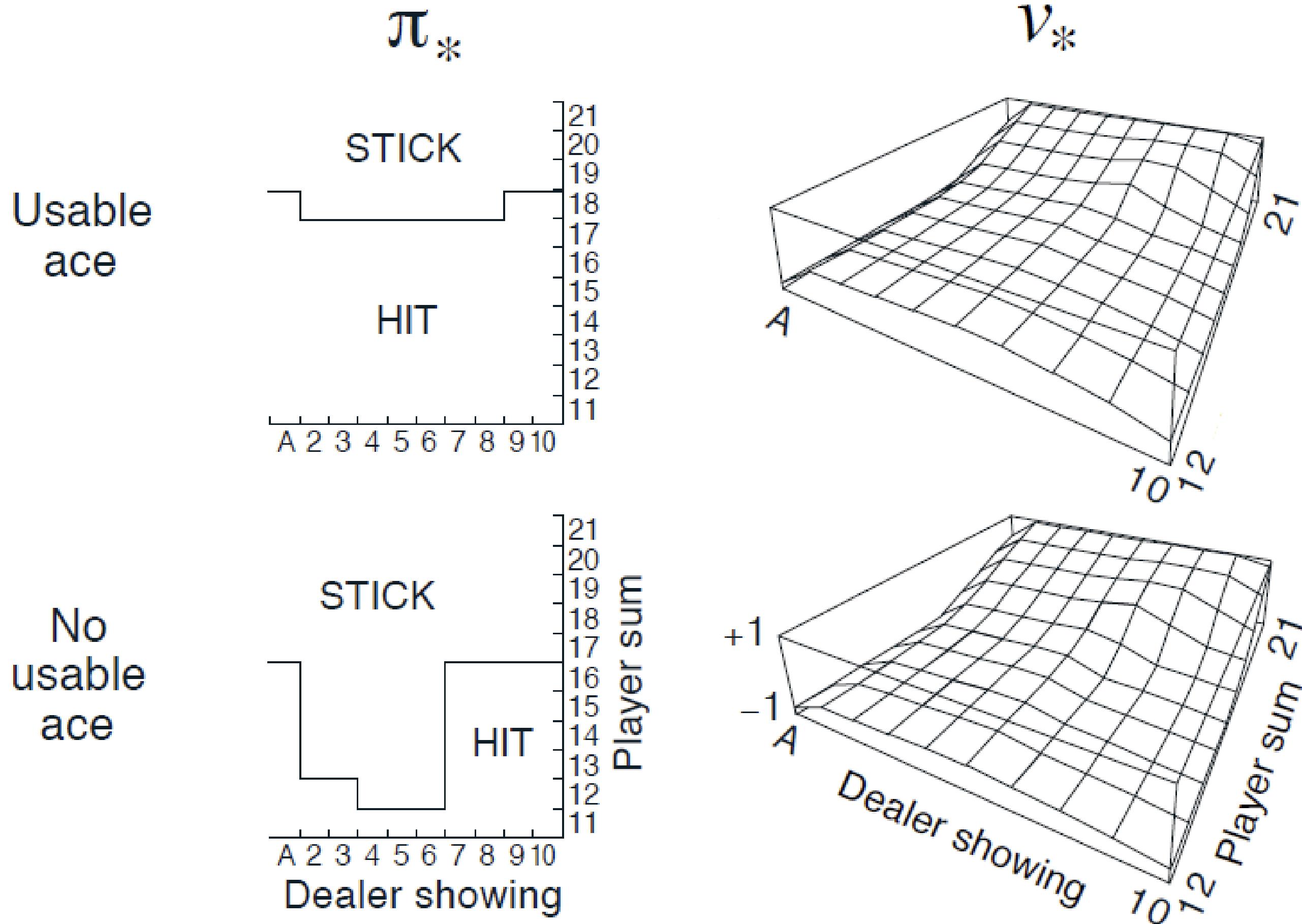
If  $A_t \neq \pi(S_t)$  then exit inner Loop (next episode)

$$W \leftarrow W \frac{1}{b(A_t|S_t)}$$

# **Black Jack Example**

Everything is known about the black jack environment...

Could the problem be solved using Dynamic Programming?



Optimal policy and value function of blackjack

# Summary (I)

- In MC, experience comes from *sample episodes*.
- Advantages over DP:
  - Interaction is with the environment, so no model of the environment's dynamic is needed
  - If models are available, they can be used with simulation in the form of sample models
  - MC can focus on a small subset of states
- Control methods follow the scheme of *generalized policy iteration* (GPI)

## Summary (II)

- MC control methods need to maintain sufficient exploration
- No bootstrapping (learning from other value estimates) is possible
- In on-policy methods, the agent tries to optimize the policy it follows
- In off-policy methods, a deterministic optimal policy is learned, while following a possibly unrelated policy

**{{DN:Hierarchy|Organisation Bezeichnung Spez.EN|ID:32|Hierarchy:1}}**

Research

**Prof. Dr. Thomas Koller**

Lecturer

Phone direct +41 41 757 68 32

[thomas.koller@hslu.ch](mailto:thomas.koller@hslu.ch)

# Monte Carlo Methods Summary



**Reinforcement Learning**  
November 2, 2022

# Monte Carlo (MC) Methods

- Monte Carlo (MC) methods look at **whole episodes** and then average the complete returns
- Value estimation and policies are only changed **on the completion of an episode**
- **GPI** (Generalized Policy Iteration) can be used for the control problem
- MC generally uses **action-value estimates** in order to compute a greedy policy
- We must maintain **exploration** to update all action-value estimates

# Monte Carlo Prediction (first visit)

Monte Carlo Prediction for estimating  $v_\pi$

Input: a policy  $\pi$

Initialize:

$V(s) \in \mathbb{R}$  (arbitrarily)

$\text{Returns}(s) \leftarrow$  an empty list, for all  $s \in \mathcal{S}$

Loop forever:

Generate episode following  $\pi$ :  $S_0, A_0, R_0, S_1, \dots, R_T$

$G \leftarrow 0$

Loop for each step of the episode,  $t = T - 1, T - 2, \dots, 0$ :

$G \leftarrow \gamma G + R_{t+1}$

Unless  $S_t$  appears in  $S_{t-1}, \dots, S_0$ :

Append  $G$  to  $\text{Returns}(S_t)$

$V(S_t) \leftarrow \text{average}(\text{Returns}(S_t))$

# On policy vs. off policy algorithm

- **On-policy** algorithms learn a policy while following this policy in the algorithm
- **Off-policy** algorithms learn a policy **different** from the one used to generate the data
- On-policy algorithms require a *soft* policy, which means that

$$\pi(a|s) > 0, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}$$

## On-policy first-visit MC control, estimates $\pi \approx \pi_*$

Input: (small)  $\epsilon > 0$

Initialize:

$\pi \leftarrow$  an arbitrary  $\epsilon$ -soft policy

$Q(s, a) \in \mathbb{R}$  (arbitrarily, for example = 0)

$Returns(s, a) \leftarrow$  empty list

Loop forever: (for each episode)

Generate an episode following  $\pi$ :  $S_0, A_0, R_0, S_1, \dots, R_T$

$G \leftarrow 0$

Loop for each step of the episode,  $t = T - 1, T - 2, \dots, 0$ :

$G \leftarrow \gamma G + R_{t+1}$

Unless  $(S_t, A_t)$  appears in  $(S_{t-1}, A_{t-1}), \dots, (S_0, A_0)$ :

Append  $G$  to  $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$  average( $Returns(S_t, A_t)$ )

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$ , ties broken arbitrarily

For all  $a \in \mathcal{A}(S_t)$ :

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)| & \text{otherwise} \end{cases}$$

# Importance Sampling

- Importance sampling *corrects* the value of the return by the probability ratio that this state would be visited by the policies

$$\begin{aligned}\rho_{t:T} &\doteq \frac{\prod_{k=1}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=1}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} \\ &= \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}\end{aligned}$$

# Importance Sampling

Ordinary importance sampling averages the results:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}$$

while weighted importance sampling uses a weighted average:

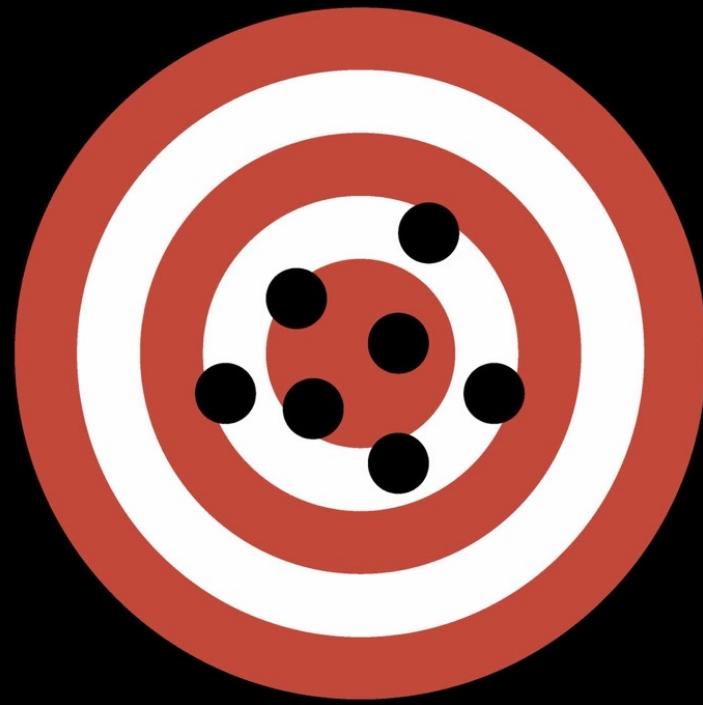
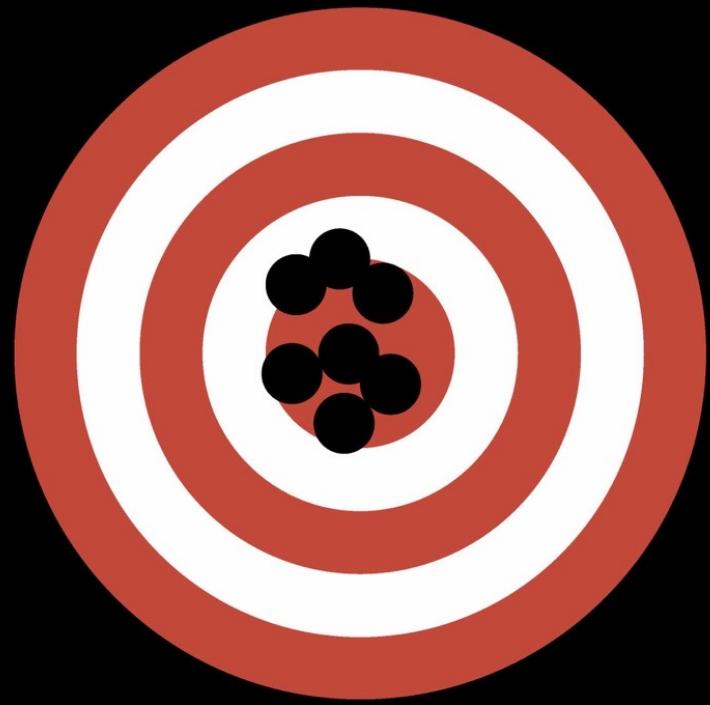
$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}}$$

Formally, ordinary importance sampling is unbiased but has a high (unbounded) variance, while weighted importance sampling is biased, but has a low (converging to zero) variance.

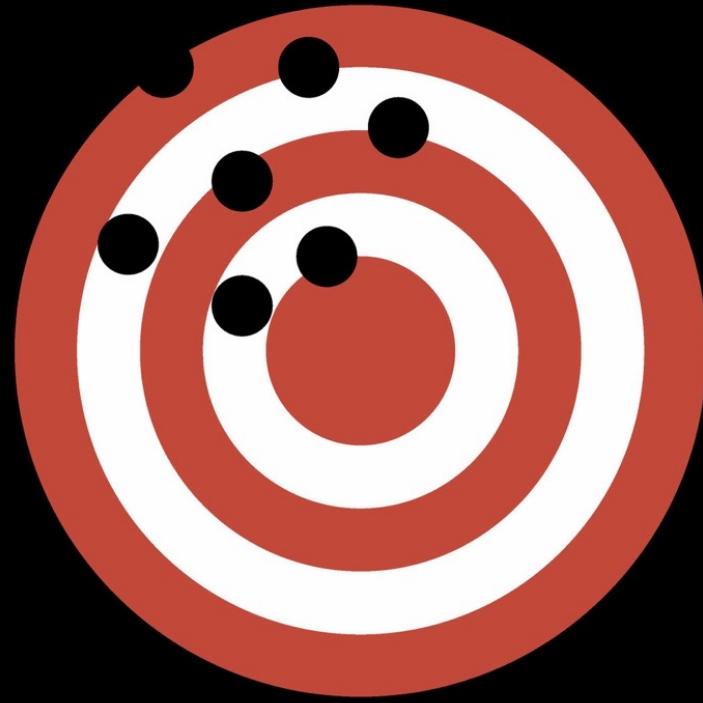
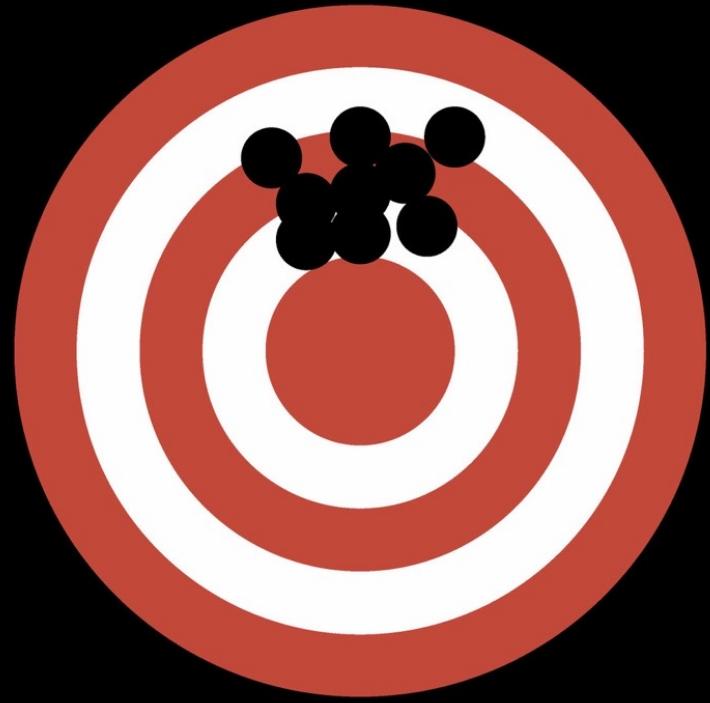
Low Variance

High Variance

Low Bias



High Bias



## Bias-Variance Tradeoff

## Off-policy MC control, estimates $\pi \approx \pi_*$

Initialize, for all  $s \in \mathcal{S}, a \in \mathcal{A}$ :

$$Q(s, a) \in \mathbb{R} \text{ (arbitrarily, for example = 0)}$$

$$C(s, a) \leftarrow 0$$

$$\pi(s) \leftarrow \operatorname{argmax}_a Q(s, a) \quad (\text{with ties broken consistently})$$

Loop forever (for each episode):

$b \leftarrow$  any soft policy

Generate an episode following b:  $S_0, A_0, R_0, S_1, \dots, R_T$

$$G \leftarrow 0$$

$$W \leftarrow 1$$

Loop for each step of the episode,  $t = T - 1, T - 2, \dots, 0$ :

$$G \leftarrow \gamma G + R_{t+1}$$

$$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$$

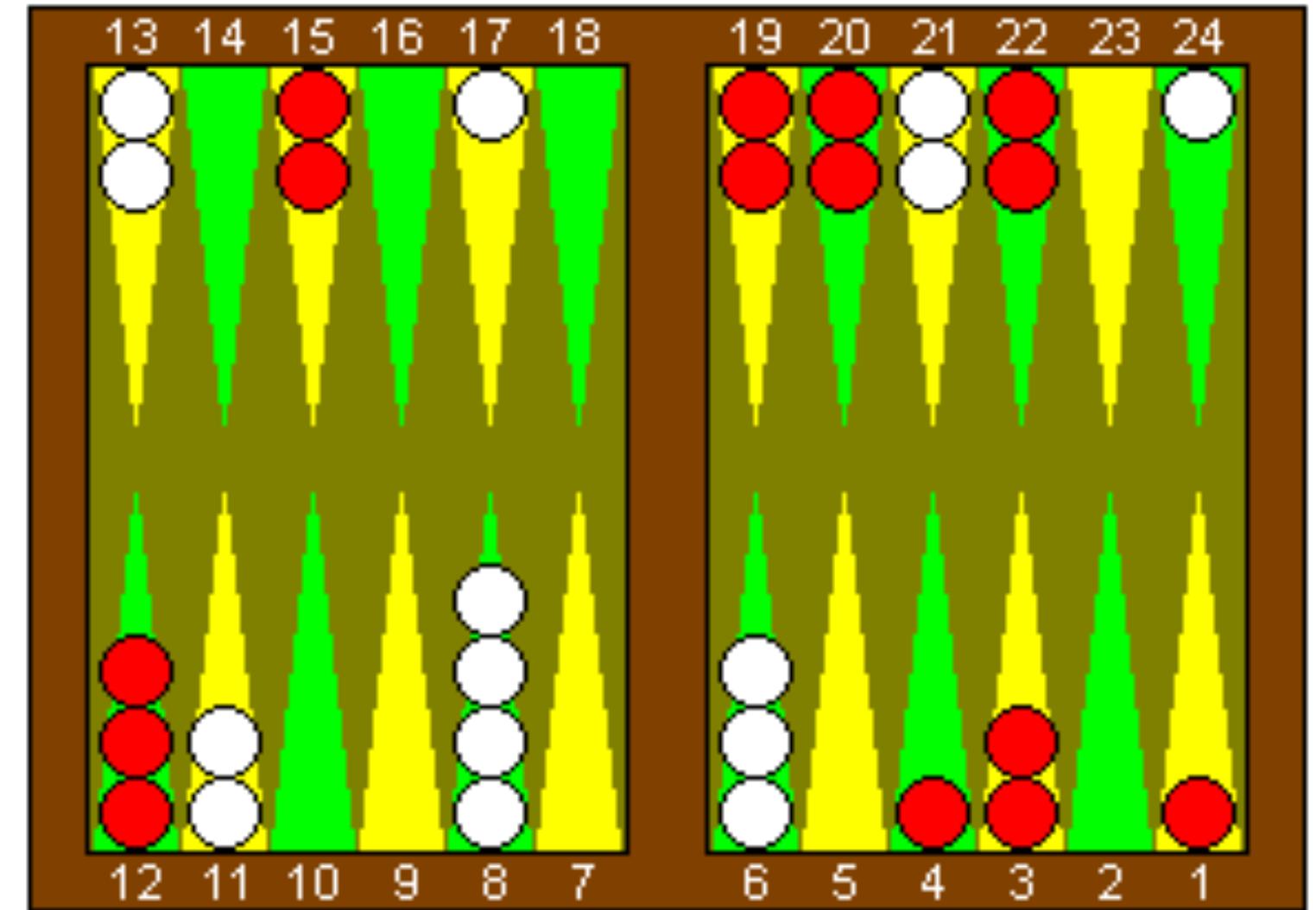
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$$

$$\pi(a|S_t) \leftarrow \operatorname{argmax}_a Q(S_t, A_t) \quad (\text{with ties broken consistently})$$

If  $A_t \neq \pi(S_t)$  then exit inner Loop (next episode)

$$W \leftarrow W \frac{1}{b(A_t|S_t)}$$

# Temporal Difference Learning



Copyright © 1995 by ACM

# Learning Objectives

- Understand how value functions are updated in Temporal Difference (TD) learning
- Understand SARSA and Q-learning for the control problem
- Understand the differences between those algorithms
- Understand how Q-learning is an off-policy algorithms without a model or importance sampling
- Understand the benefits (and drawbacks) of TD methods

# Introduction

- Dynamic Programming (DP) methods required a model of the environment
- DP updated the value function after one time step
- Monte Carlo (MC) methods updated the value function only after the end of the episodes
- Can we combine an update after one step without using a model?

# Temporal Difference Learning

- Temporal difference (TD) learning combines elements from Dynamic Programming (DP) and Monte Carlo (MC)
- TD learning is model free
- TD learning updates estimates without waiting for the final outcome (bootstrapping)

# Prediction

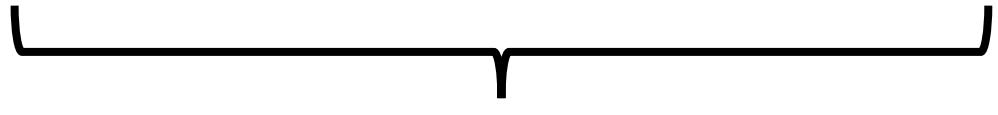
- recall MC prediction with constant step size

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

  
Target

- TD methods make the update immediately based on the estimates for the next state

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

  
Target

## General formulation

- More generally, the update can be written as

$$V(S_t) \leftarrow V(S_t) + \alpha \delta_t$$

- With the *error term*, defined as

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

- This is the error in the estimate made at the time. As it depends on the next state, it is only available at t+1
- This TD is known as TD(0), as the update takes place after 1 step

# TD(0) Prediction

TD(0) for estimating  $v_\pi$

Input:

the policy  $\pi$  to be evaluated  
step size  $\alpha \in (0, 1]$

Initialize:

$V(S)$  arbitrarily (except  $V(\text{terminal}) = 0$ )

Loop for each episode:

    Initialize  $S$

    Loop for each step of the episode:

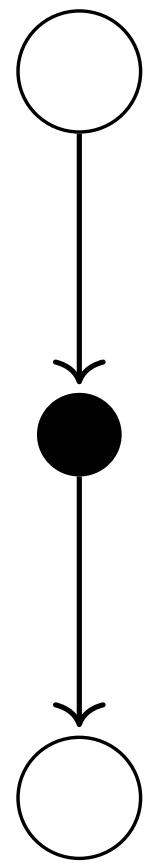
$A \leftarrow$  action given by  $\pi$  for  $S$

        Take action  $A$ , observe  $R, S'$

$V(S_t) \leftarrow V(S_t) + \alpha[R + \gamma V(S') - V(S)]$

$S \leftarrow S'$

    until  $S$  is terminal



TD(0)

# Example

How long does it take you to go home:

- Leave school
- Arrive at bus station
- Take bus
- Exit bus
- Go to bicycle
- Arrive home

# Example

State	Elapsed time	Predicted time (from state)	Predicted time (total)
Leave school	0	45	45
Arrive bus station	5	50	55
Take bus	10	45	55
Exit bus	40	20	60
Go to bicycle	45	15	60
Arrive home	58	0	58

Rewards are the time elapsed; the value is the expected time to go home

# Example

State	Elapsed time	Predicted time (from state)	Predicted time (total)	MC Error (alpha = 1.0)	TD Error
Leave school	0	45	45	$58 - 45 = 13$	$55 - 45 = 10$
Arrive bus station	5	50	55	$58 - 55 = 3$	$55 - 55 = 0$
Take bus	10	45	55	$58 - 55 = 3$	$60 - 55 = 5$
Exit bus	40	20	60	$58 - 60 = -2$	$60 - 60 = 0$
Go to bicycle	45	15	60	$58 - 60 = -2$	$58 - 60 = -2$
Arrive home	58	0	58	-	-

# Advantages of TD

- Updates are based partly on existing estimates, this is called *bootstrapping*
- No model is required
- Updates are fully incremental and online. There is no need to wait for the end of the episode.
- Does it work? Yes,  $\text{TD}(0)$  has been shown to converge to the true  $v$

# Batch Updating

- If only a finite amount of experience is available, this is presented to the algorithm repeatedly until convergence.
- This is called batch updating.

# SARSA: On-policy TD Control

- We use generalized policy iteration (GPI) for the control problem.
- As in MC methods, we must balance exploration and exploitation
- TD control methods generally learn an action-value function instead of a state-value function
- So we look at the transition from  $(\mathbf{S}, \mathbf{A})$  with reward  $(\mathbf{R})$  to the next  $(\mathbf{S}, \mathbf{A})$  (SARSA)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

# SARSA

Sarsa for estimating  $Q \approx q_*$

Input:

step size  $\alpha \in (0, 1]$

small  $\epsilon > 0$

Initialize:

$Q(s, a)$  for all  $s \in \mathcal{S}^+, a \in \mathcal{A}$  arbitrarily (except  $Q(\text{terminal}, \cdot) = 0$ )

Loop for each episode:

    Initialize  $S$

    Choose  $A$  from  $S$  using a policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Loop for each step of the episode:

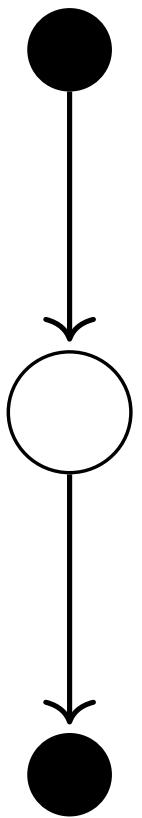
        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using a policy derived from  $Q$  (e.g.,  $\epsilon$ -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



# Q-learning: Off-policy TD Control

Q-learning is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

- The learned action-value function  $Q$  directly approximate  $q_*$  and is independent of the policy being followed.
- The policy determines which state-action pairs are visited and updated, so it must visit all pairs eventually

# Q-Learning

Q-learning for estimating  $Q \approx q_*$

Input:

step size  $\alpha \in (0, 1]$

small  $\epsilon > 0$

Initialize:

$Q(s, a)$  for all  $s \in \mathcal{S}^+, a \in \mathcal{A}$  arbitrarily (except  $Q(\text{terminal}, \cdot) = 0$ )

Loop for each episode:

Initialize  $S$

Loop for each step of the episode:

Choose  $A$  from  $S$  using a policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

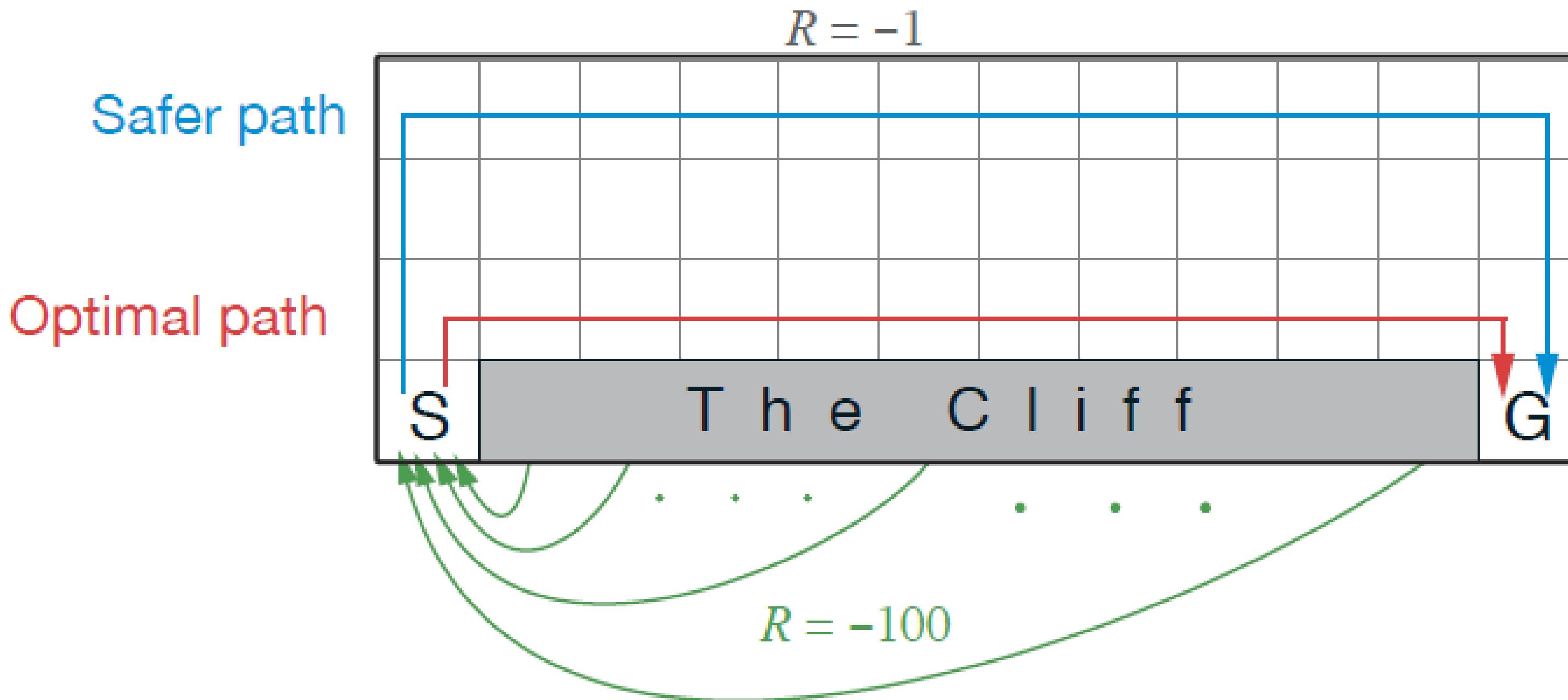
Take action  $A$ , observe  $R, S'$

$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

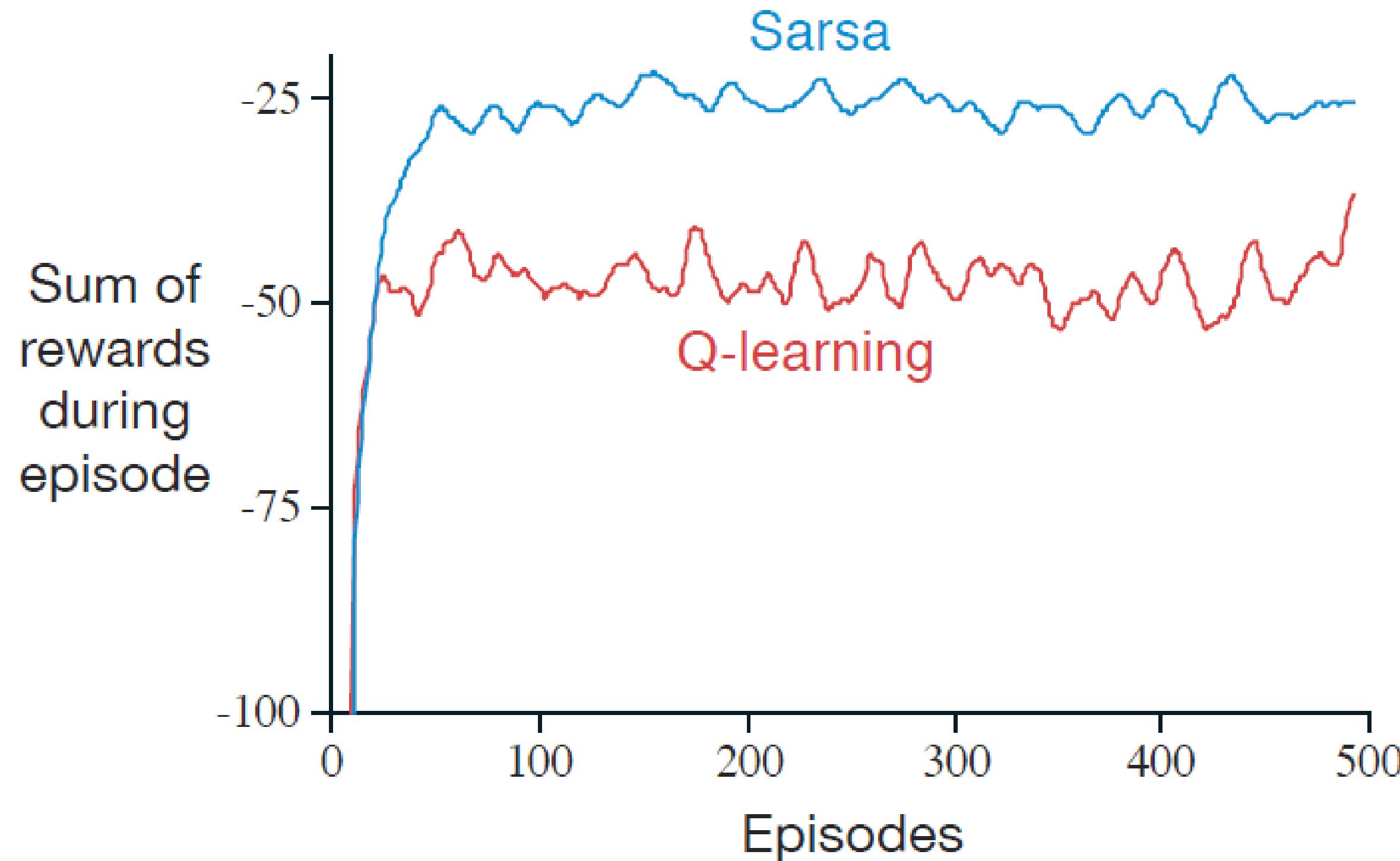
$S \leftarrow S'$

until  $S$  is terminal

# Example



# Example



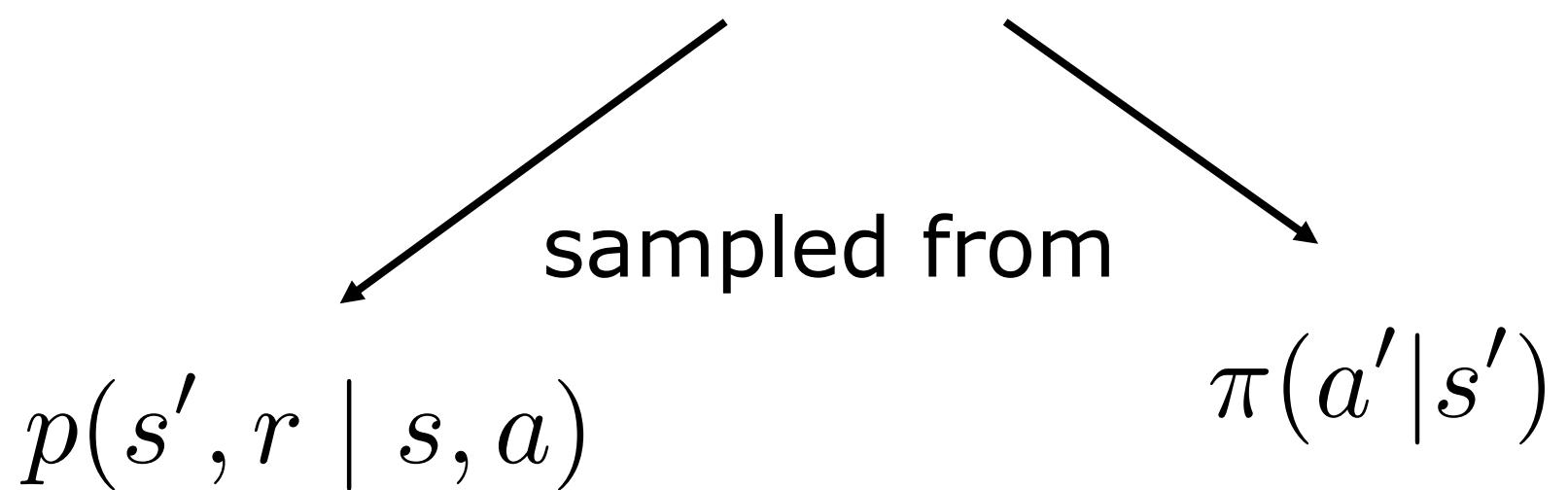
# Expected SARSA

Bellman equation

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

Sarsa:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

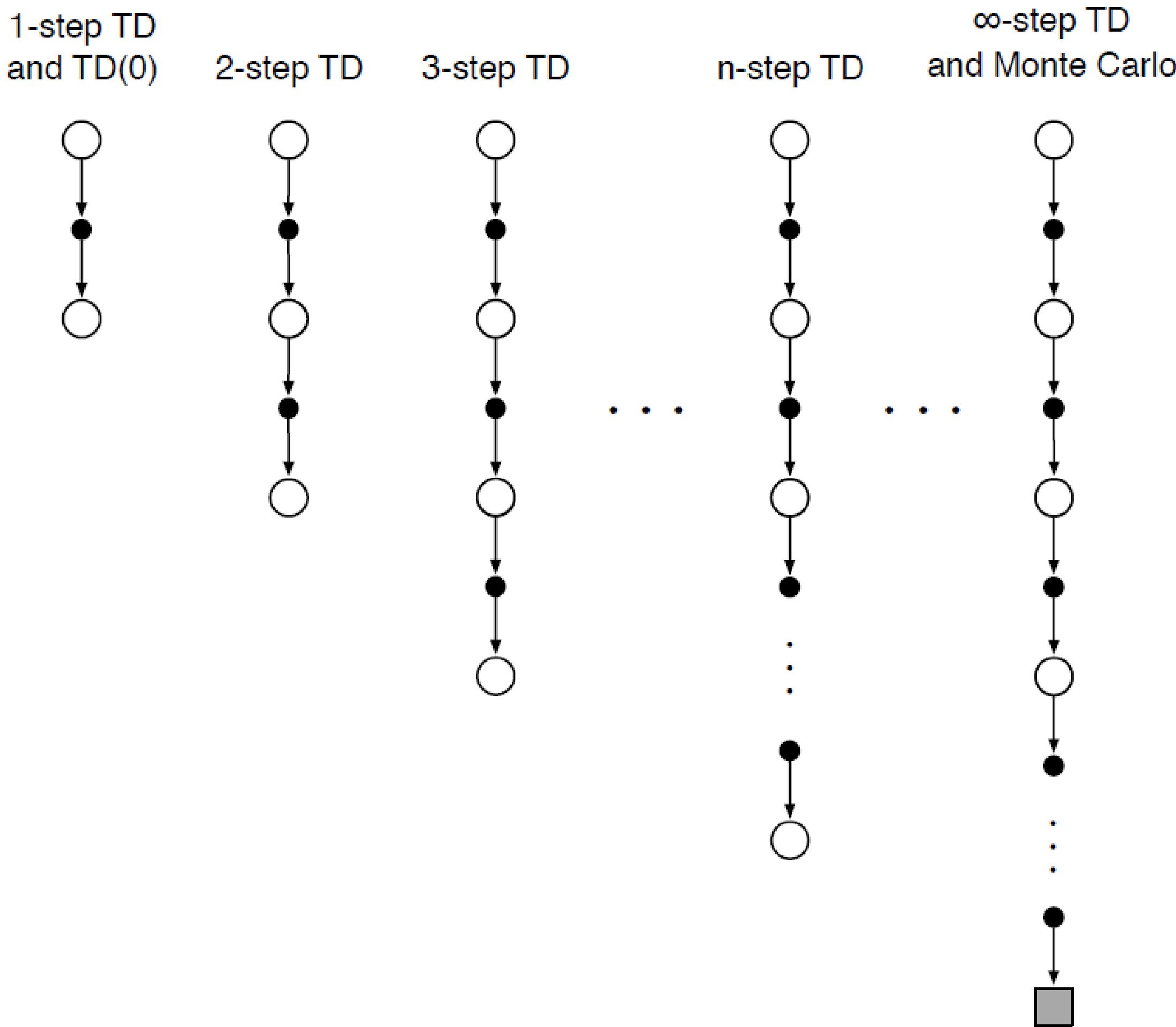


# Expected SARSA

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Expected value of the return

# n-step TD Prediction



## Targets of n-step returns

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

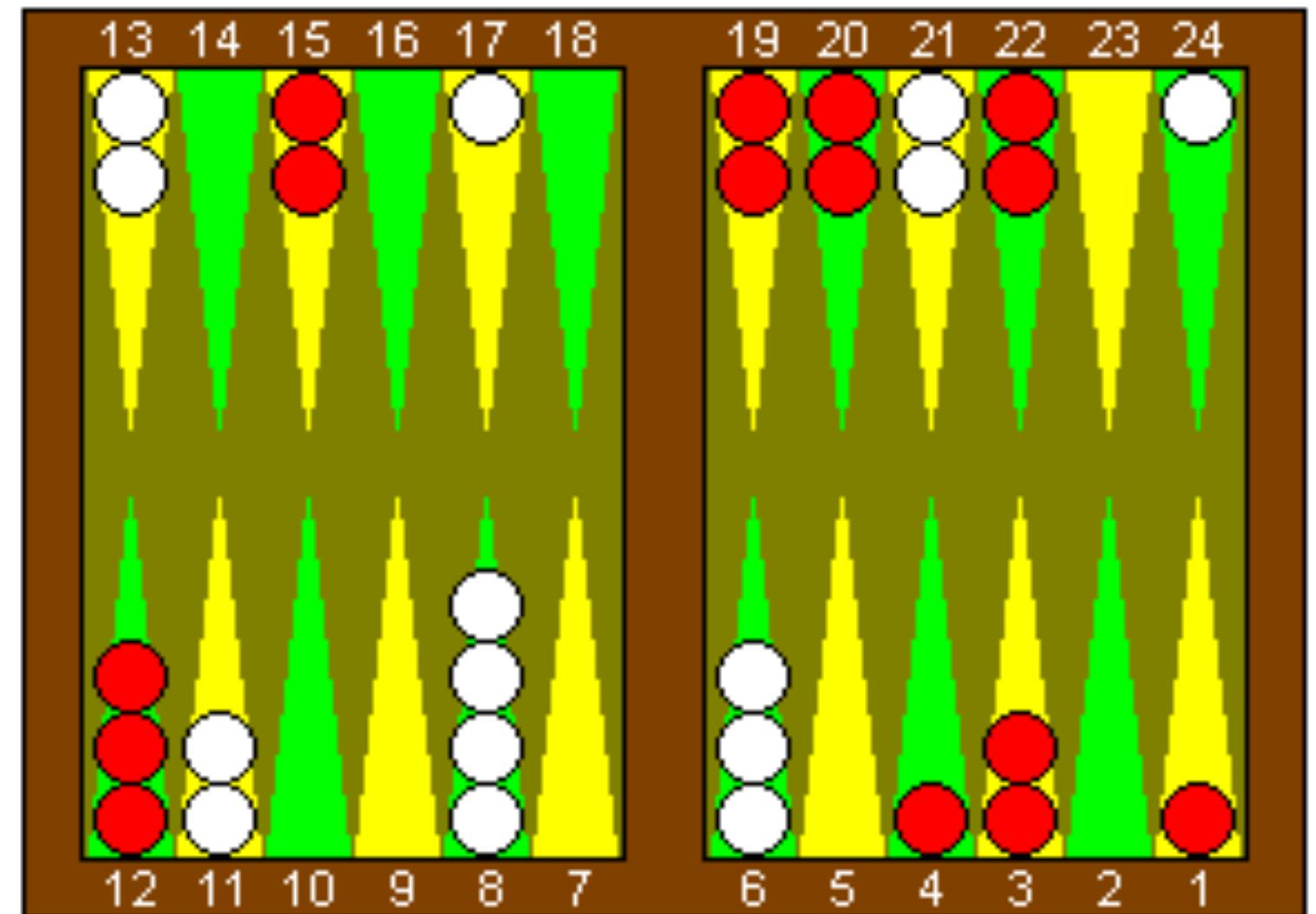
$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} + \gamma^n V_{t+n-1}(S_{t+n})$$

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)]$$

# Summary

- TD(0) methods allow updating value estimates after only 1 step
- New estimates are therefore based on the estimates of the successor states, this is called bootstrapping
- SARSA is an on-policy control algorithm that uses TD(0) as prediction step with an  $\varepsilon$ -soft policy
- Q-Learning is an off-policy control algorithm that uses TD(0) as prediction step and learns a greedy policy while following an  $\varepsilon$ -soft policy
- Expected SARSA improves SARSA by summing over the possible actions from the policy

# Temporal Difference Learning: Summary



Copyright © 1995 by ACM

# TD Prediction

TD(0) methods update the state- or action-value function based on the estimates for the next state or state-action pair

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



Target

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$



Target

# TD(0) Prediction

TD(0) for estimating  $v_\pi$

Input:

the policy  $\pi$  to be evaluated  
step size  $\alpha \in (0, 1]$

Initialize:

$V(S)$  arbitrarily (except  $V(\text{terminal}) = 0$ )

Loop for each episode:

    Initialize  $S$

    Loop for each step of the episode:

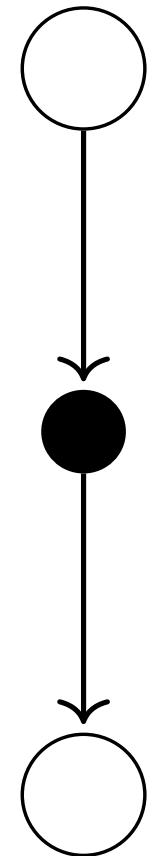
$A \leftarrow$  action given by  $\pi$  for  $S$

        Take action  $A$ , observe  $R, S'$

$V(S_t) \leftarrow V(S_t) + \alpha[R + \gamma V(S') - V(S)]$

$S \leftarrow S'$

    until  $S$  is terminal



TD(0)

# TD(0) Control

There are 3 TD(0) control methods which all use Generalized-Policy-Iteration (GPI)

## Sarsa

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

## Q-Learning

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

## Expected Sarsa

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

# SARSA

Sarsa for estimating  $Q \approx q_*$

Input:

step size  $\alpha \in (0, 1]$

small  $\epsilon > 0$

Initialize:

$Q(s, a)$  for all  $s \in \mathcal{S}^+, a \in \mathcal{A}$  arbitrarily (except  $Q(\text{terminal}, \cdot) = 0$ )

Loop for each episode:

    Initialize  $S$

    Choose  $A$  from  $S$  using a policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Loop for each step of the episode:

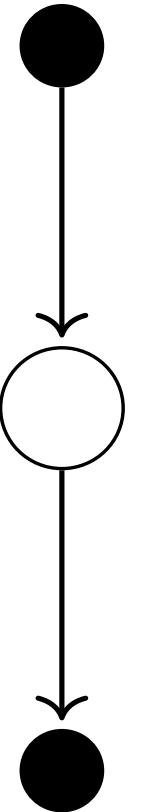
        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using a policy derived from  $Q$  (e.g.,  $\epsilon$ -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



# Q-Learning

Q-learning for estimating  $Q \approx q_*$

Input:

step size  $\alpha \in (0, 1]$

small  $\epsilon > 0$

Initialize:

$Q(s, a)$  for all  $s \in \mathcal{S}^+, a \in \mathcal{A}$  arbitrarily (except  $Q(\text{terminal}, \cdot) = 0$ )

Loop for each episode:

Initialize  $S$

Loop for each step of the episode:

Choose  $A$  from  $S$  using a policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

Take action  $A$ , observe  $R, S'$

$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$

until  $S$  is terminal

## Targets of n-step returns

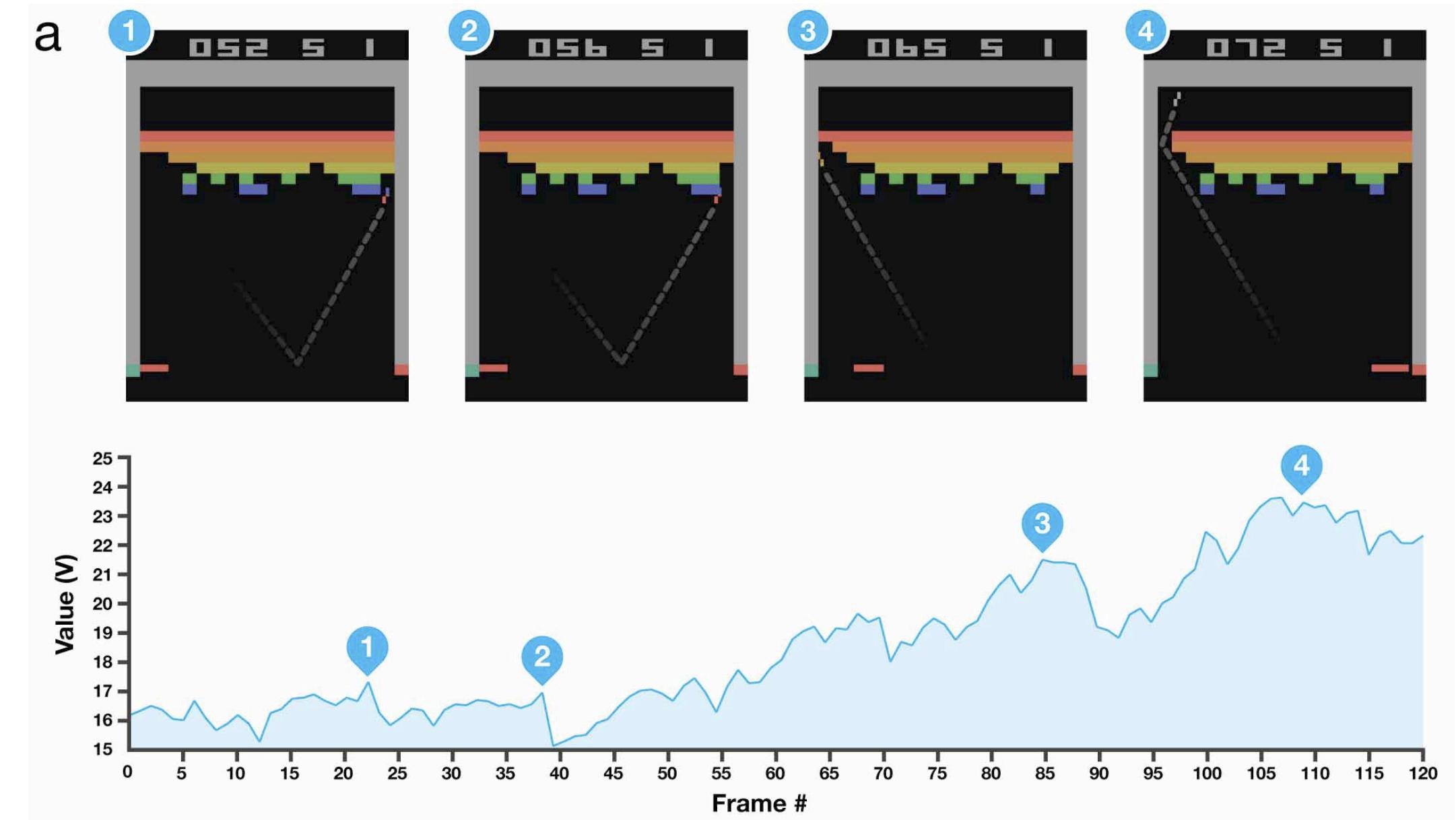
$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} + \gamma^n V_{t+n-1}(S_{t+n})$$

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)]$$

# Function Approximation Methods



# Learning Objectives

- Motivation for using function approximation over tabular methods
- Understanding the objective that should be learned
- Stochastic gradient descent and semi-gradient algorithms
- Understanding Sarsa and Q-Learning with function approximation

# Tabular methods

So far, we have used tabular methods:

- Value Functions (and sometimes policies) were stored in a table for each state or state-action

Problems:

- State spaces are arbitrarily large in many RL methods
- State spaces and/or action spaces might be continuous and not discrete
- Many states will not be encountered, we need to **generalize** from other states, i.e. we would like to have a reasonable response from a trained agent, even if a state has never been encountered

# Function Approximation

- We want to approximate the state-values or the state-action-values by a function that is parametrized by some parameter  $w$

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

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

- The dimensionality  $d$  of  $w$  will generally be much lower than the dimensionality of the state space

$$d \ll |\mathcal{S}|$$

# Target Update

Different (tabular) RL methods have different target functions:

- MC Target:  $G_t$
- TD(0) Target:  $R_{t+1} + \gamma V(S_{t+1})$
- n-step TD Target:  $G_{t:t+n}$

Each update moves the value function in direction of the target

$$V(S_t) \leftarrow V(S_t) + \alpha [Target - V(S_t)]$$

# Updates

Tabular Methods:

- Updates for state values were calculated directly to get a better estimate of the target

Function Approximation:

- Each update can be viewed as a training example
- We can then use **supervised learning** on those training examples

However: in RL the updates must be online as the agents learns while interacting with the environment

An agent must be able to learn efficiently from incrementally acquired data

# Prediction Objective

As an objective, we would like to learn to approximate the value function at each state:

$$[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

However, an update to one state, will affect the other states

We want to introduce a function  $\mu(s)$  that measures how important each error is and then minimize the mean weighted error between the value function and its approximation

$$\overline{\text{VE}}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s) [v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

# Prediction Objective: Caveats

(It is not completely clear that VE is the best objective to optimize for RL:

- The purpose of finding the value function, is to find a better policy
- The best value function for that, might not be the best for minimizing VE.
- Further more, a *global optimum* in minimizing VE is unlikely to be found)

# On-policy distribution

$\mu(s)$  can be calculated as the fraction of how much time is spent in  $s$ , i.e. how often is  $s$  visited

in on-policy methods, this is called the on-policy distribution, for episodic tasks, that also depends on how the starting states are chosen, let  $h(s)$  be the probability of choosing starting state  $s$ , then  $\eta(s)$  is the average time spent in  $s$  in a single episode

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s|\bar{s}, a), \quad \text{for all } s \in \mathcal{S}$$

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')} \quad \text{for all } s \in \mathcal{S}$$

# Stochastic gradient descent

Assume that in each step we observe a state  $s$  and its (true) value under the policy.

We can use stochastic gradient-descend (SGD) by adjusting the weights vector to minimize the error in the observed examples

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \\ &= \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]\nabla \hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

# Monte Carlo Prediction

## Gradient Monte Carlo Prediction

Input:

a policy  $\pi$

a differentiable function  $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$  with parameters  $\mathbf{w}$

a step size parameter  $\alpha > 0$

Initialize:

$\mathbf{w}$  arbitrarily

Loop forever:

Generate episode following  $\pi$ :  $S_0, A_0, R_0, S_1, \dots, R_T$

$G \leftarrow 0$

Loop for each step of the episode,  $t = T - 1, T - 2, \dots, 0$ :

$G \leftarrow \gamma G + R_{t+1}$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$

# Semi Gradient Methods

Bootstrapping methods are not true gradient descent methods, as the *targets* themselves depend on the weights  $w$

They only contain part of the gradient and are therefore called *semi-gradient* methods

However, the calculation works just as before....

# TD(0) Prediction

Semi-gradient TD(0) for estimating  $\hat{v} \approx v_\pi$

Input:

a policy  $\pi$

a differentiable function  $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$  with parameters  $\mathbf{w}$

( $\hat{v}(\text{terminal}, \cdot)$ ) must be 0

a step size parameter  $\alpha > 0$

Initialize:

$\mathbf{w}$  arbitrarily

Loop for each episode)

    Initialize  $S$

    Loop for each step of the episode:

$A \leftarrow$  action given by  $\pi$  for  $S$

        Take action  $A$ , observe  $R, S'$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$

$S \leftarrow S'$

    until  $S$  is terminal

# Episodic Semi-gradient Control

For control, we want to approximate action-value functions

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

The update target is again any approximation of  $q$ , for example for (1-step) Sarsa:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$$

# Episodic Semi-gradient Control

- Generalized policy iteration (GPI) can be used again
- For on policy methods a soft approximation to the greedy policy can be used for policy improvement
- If the action set is discrete and not too large, the greedy action can be calculated directly

$$A_{t+1}^* = \arg \max_a \hat{q}(S_{t+1}, a, w_t)$$

## Semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input:

- a differentiable function  $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$  with parameters  $\mathbf{w}$
- a step size parameter  $\alpha > 0$ , small  $\epsilon > 0$

Initialize:

$\mathbf{w}$  arbitrarily

Loop for each episode)

    Initialize  $S$

    Choose  $A$  as a function of  $\hat{q}(S, ., \mathbf{w})$  (e.g.,  $\epsilon$ -greedy)

    Loop for each step of the episode:

        Take action  $A$ , observe  $R, S'$

        If  $S'$  is terminal:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R - \hat{q}(S, A, \mathbf{w})]\nabla\hat{q}(S, A, \mathbf{w})$$

        Go to next episode

        Choose  $A'$  as a function of  $\hat{q}(S', ., \mathbf{w})$  (e.g.,  $\epsilon$ -greedy)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})]\nabla\hat{q}(S, A, \mathbf{w})$$

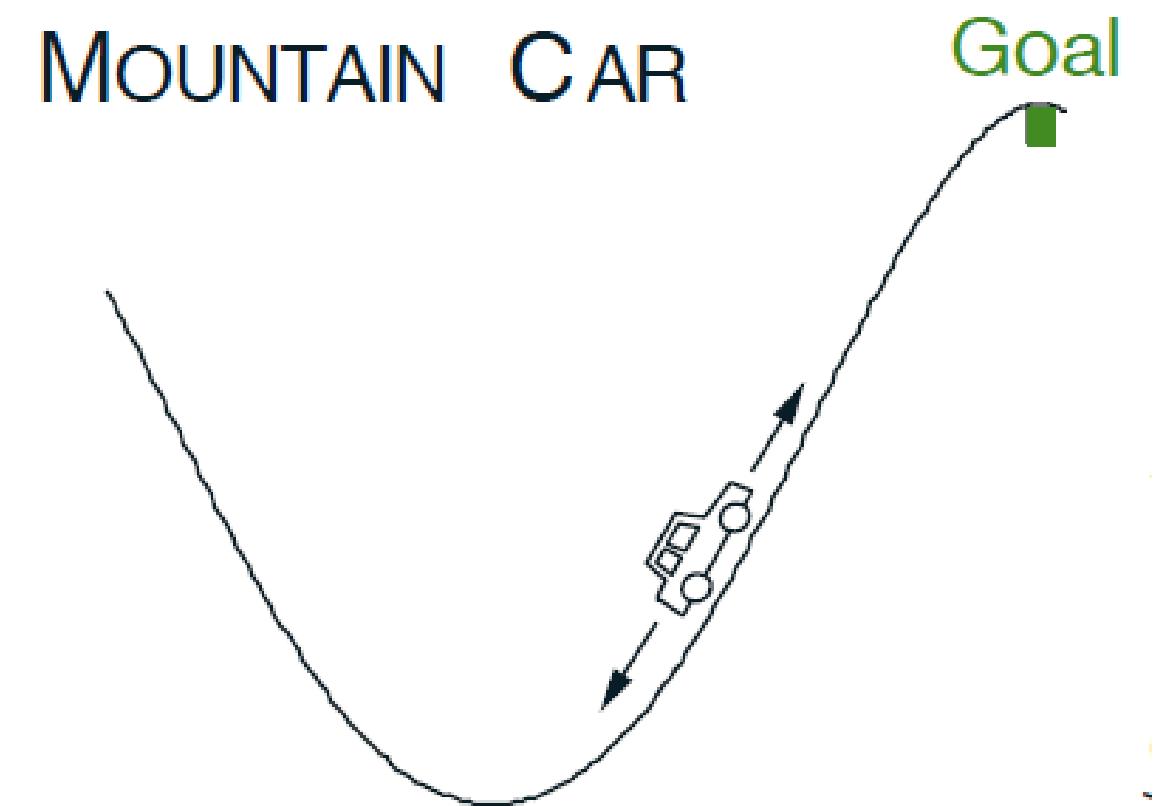
$$S \leftarrow S'$$

$$A \leftarrow A'$$

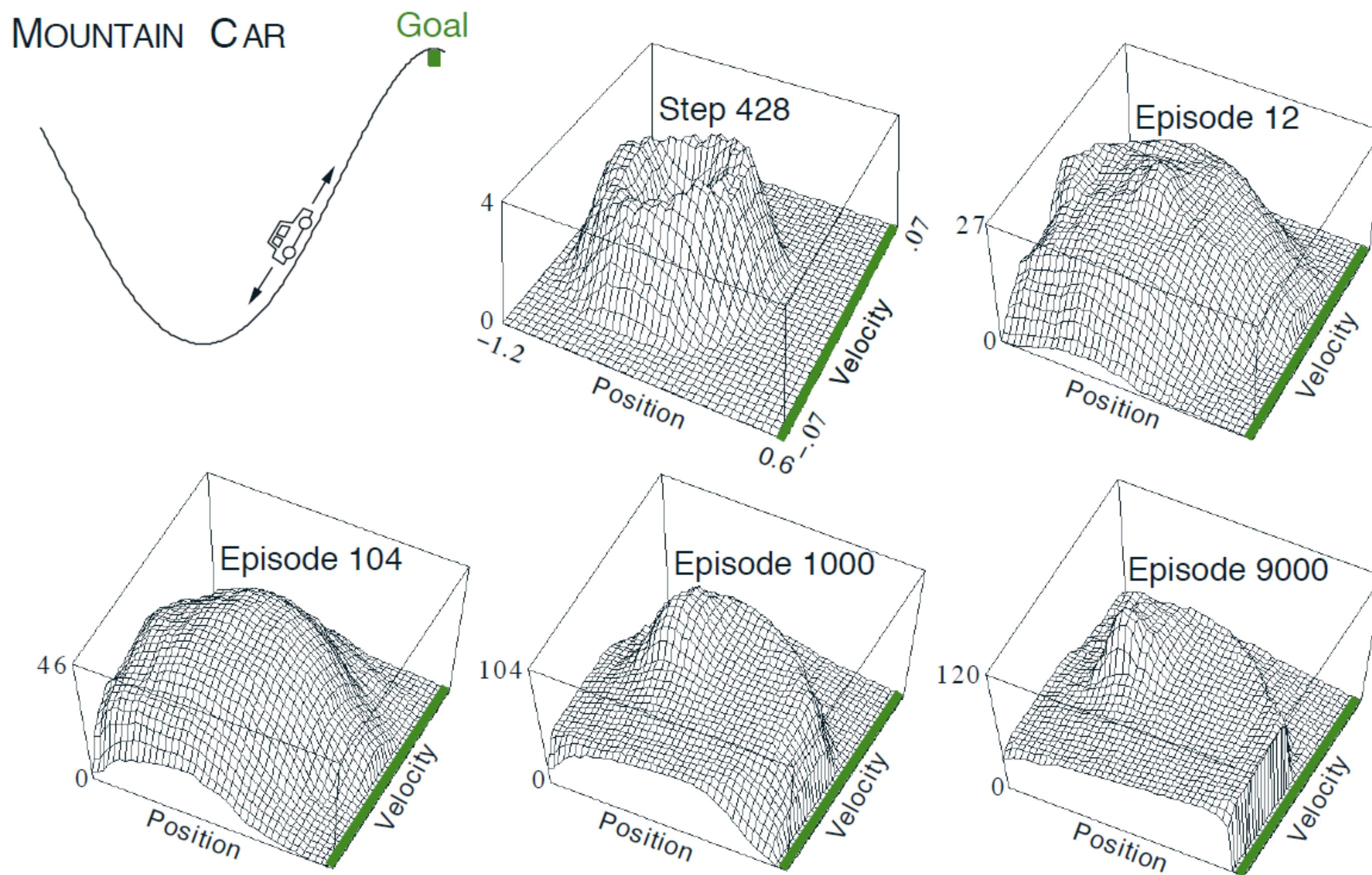
# Mountain Car Example

The car wants to move up the mountain to reach the goal, but the gravity is larger than its acceleration

- 3 Actions: Full throttle forward, full throttle backwards or no throttle
- Reward is -1 for each time step (0 for reaching goal)
- Each episode starts at random position (with velocity 0)

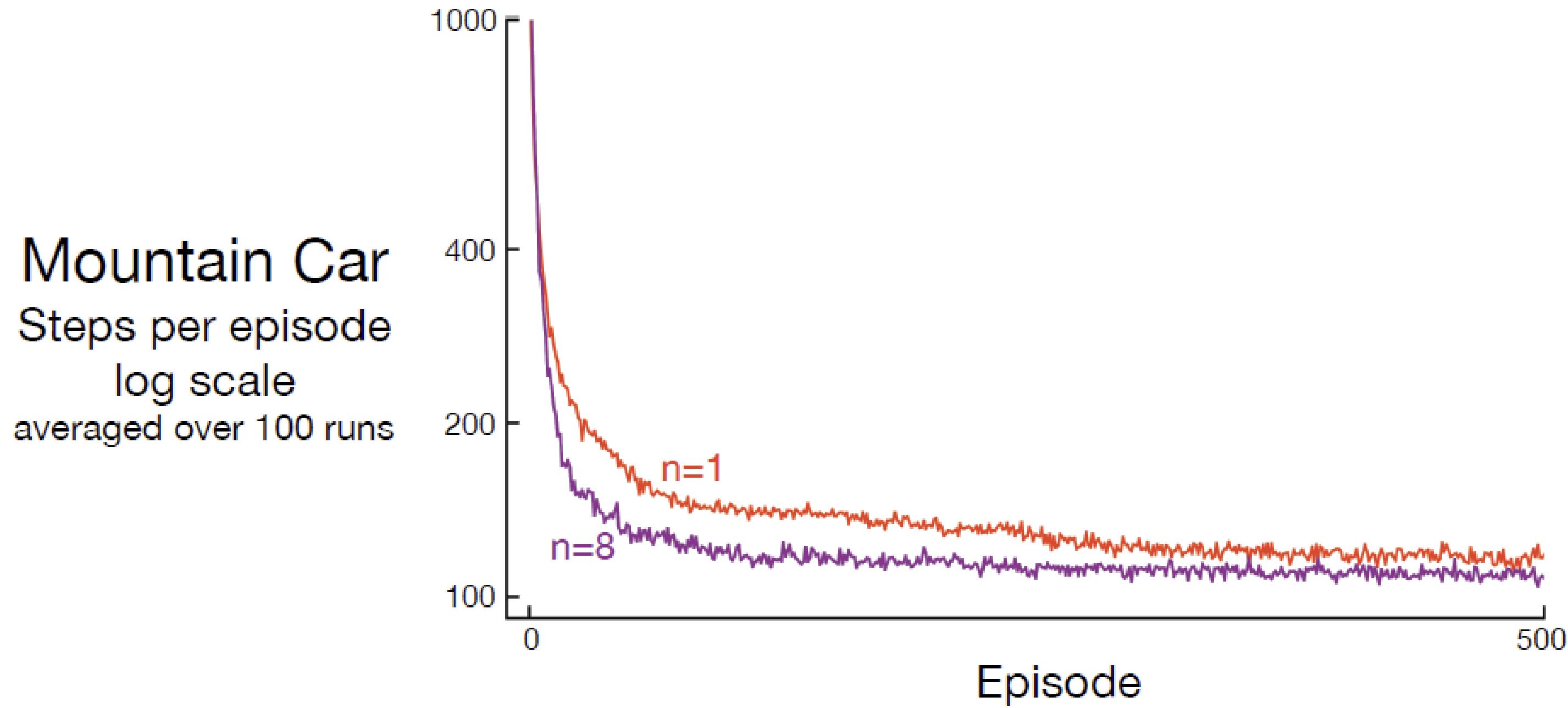


# Mountain Car Example

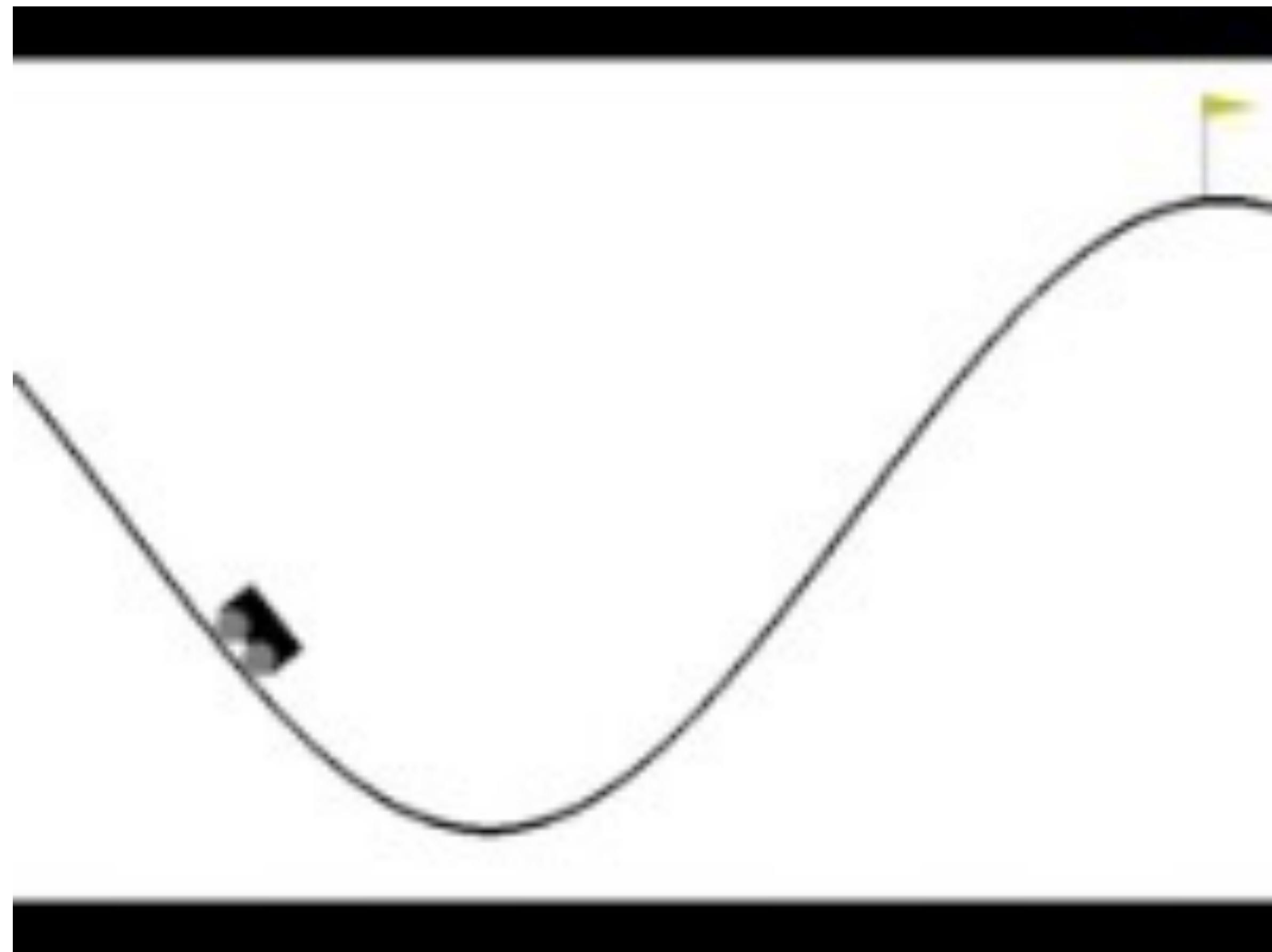


$$-\max_a \hat{q}(s, a, \mathbf{w})$$

# Mountain Car Example



# 1-step vs. 8-step semi-gradient Sarsa



# Off-Policy Learning

Off-policy learning can be unstable or even diverge with nonlinear function approximation:

- Correlations in the sequence of observations
- Small updates to  $q$  may significantly change the policy
- Correlations between the action values  $q$  and the target values

# Deep Q-Learning

---

## Playing Atari with Deep Reinforcement Learning

---

Volodymyr Mnih   Koray Kavukcuoglu   David Silver   Alex Graves   Ioannis Antonoglou  
Daan Wierstra   Martin Riedmiller  
DeepMind Technologies

## Human-level control through deep reinforcement learning

Volodymyr Mnih<sup>1\*</sup>, Koray Kavukcuoglu<sup>1\*</sup>, David Silver<sup>1\*</sup>, Andrei A. Rusu<sup>1</sup>, Joel Veness<sup>1</sup>, Marc G. Bellemare<sup>1</sup>, Alex Graves<sup>1</sup>, Martin Riedmiller<sup>1</sup>, Andreas K. Fidjeland<sup>1</sup>, Georg Ostrovski<sup>1</sup>, Stig Petersen<sup>1</sup>, Charles Beattie<sup>1</sup>, Amir Sadik<sup>1</sup>, Ioannis Antonoglou<sup>1</sup>, Helen King<sup>1</sup>, Dharshan Kumaran<sup>1</sup>, Daan Wierstra<sup>1</sup>, Shane Legg<sup>1</sup> & Demis Hassabis<sup>1</sup>

# Deep Q-Learning

Deep Q-Learning uses a deep Q network (DQN), that estimates the action value function

- Uses experience replay
- Updates the action-values iteratively towards target
- Target values are only updated periodically

$$J(\mathbf{w}_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[ (r + \gamma \max_a Q(s', a', \mathbf{w}_i^-)) - Q(s, a, \mathbf{w}_i)^2 \right]$$

# Deep Q-Learning

$$J(\mathbf{w}_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[ (r + \gamma \max_a Q(s', a', \mathbf{w}_i^-) - Q(s, a, \mathbf{w}_i))^2 \right]$$

- The target network parameters  $\mathbf{w}_i^-$  are only updated with the Q-network parameters every C steps
- The network is actually trained to learn

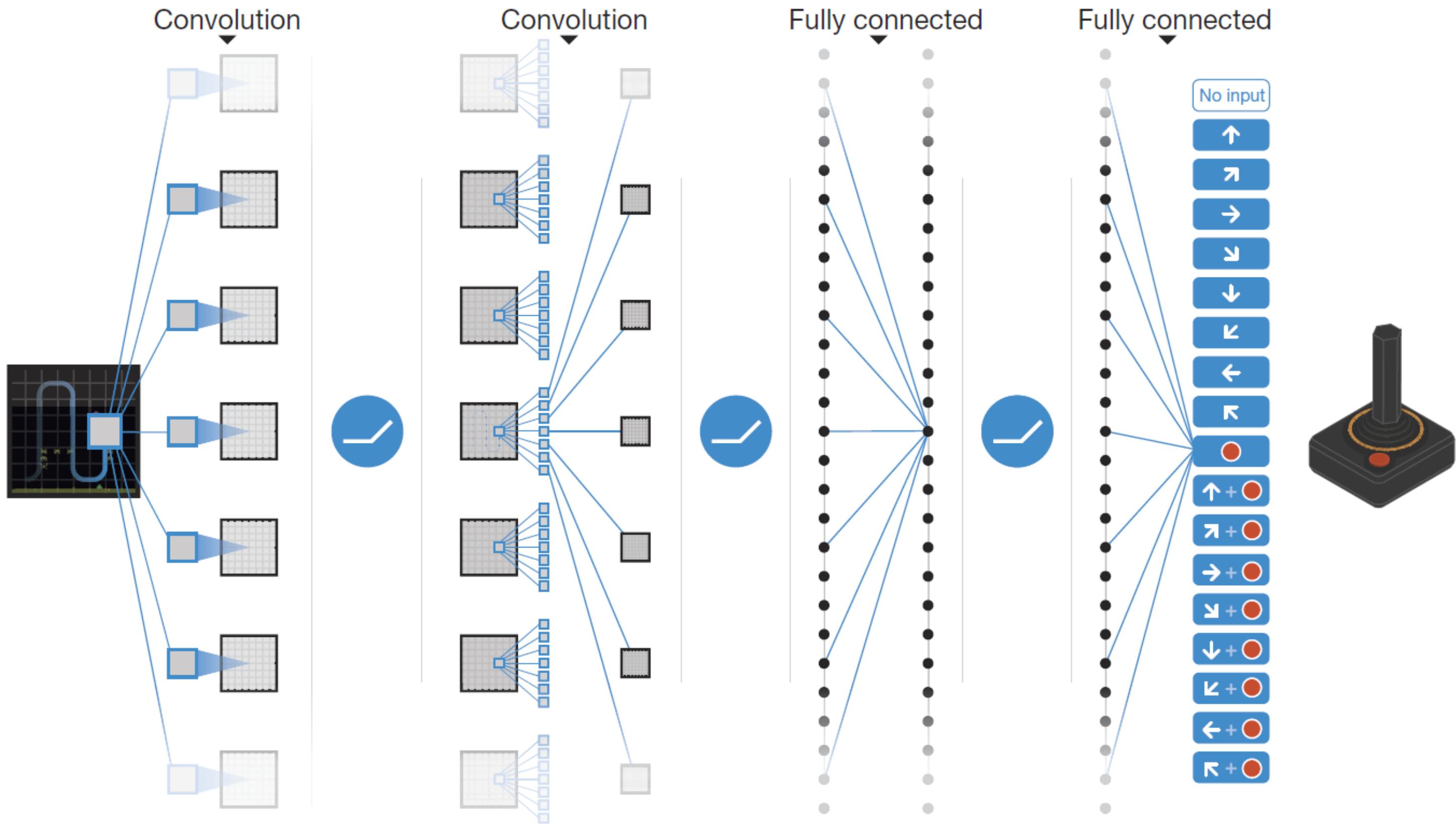
$$q(a|s) = Q(s, \mathbf{w})$$

i.e. it returns the q-value of each action in s given s.

# Deep Q-Learning

- For the Atari games, the input was the actual image of the game, so a convolutional neural network was used.
- The images are 210x160 pixels and have been converted to grey-value images, down-sampled and cropped to 84x84
- Input are the last 4 frames
- Rewards are the score of the game (normalized and clipped at +1,-1)
- A new action is calculated every 4-th frame and repeated in between frames

# Deep Q-Learning: Network Configuration



# Batch Methods

- For function approximation with neural networks, it is not efficient to update the network after one step
- Furthermore, the sample is only used once
- In supervised learning approaches with neural networks, the training goes over many episodes of the same data
- Solution: Collect a batch of experiences and store them in a buffer to use multiple times
- This is called **Experience Replay**

# Experience Replay

The experience at each time step is stored into a data set

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

$$\mathcal{D} = e_1, \dots, e_N$$

- A fixed size buffer is used that overwrites the oldest experience
- Training data is then sampled from the data set

Advantages:

- Each experience can be used in many updates
- Learning from consecutive samples would be inefficient as they are correlated

## Deep Q-Learning with experience replay

Initialize:

the replay memory  $D$  to capacity  $N$

action-value function  $Q$  with random weights  $\mathbf{w}$

target action-value function  $\hat{Q}$  with weights  $\mathbf{w}^- = \mathbf{w}$

Loop for each episode:

Initialize  $S_1$

For every step  $t = 1, T$  in the episode:

Choose  $A_t$  as a function of  $Q(S_t, ., \mathbf{w})$  (e.g.,  $\epsilon$ -greedy)

Take action  $A_t$ , observe  $R_t, S_{t+1}$

Store transition  $(S_t, A_t, R_r, S_{t+1})$  in  $D$

Sample a random minibatch of transitions  $(S_j, A_j, R_j, S_{j+1})$  from  $D$

$$y_j = \begin{cases} R_j & \text{if } S_{j+1} \text{ is terminal} \\ R_j + \gamma \max_{A'} \hat{Q}(S_{j+1}, A', \mathbf{w}^-) & \text{otherwise} \end{cases}$$

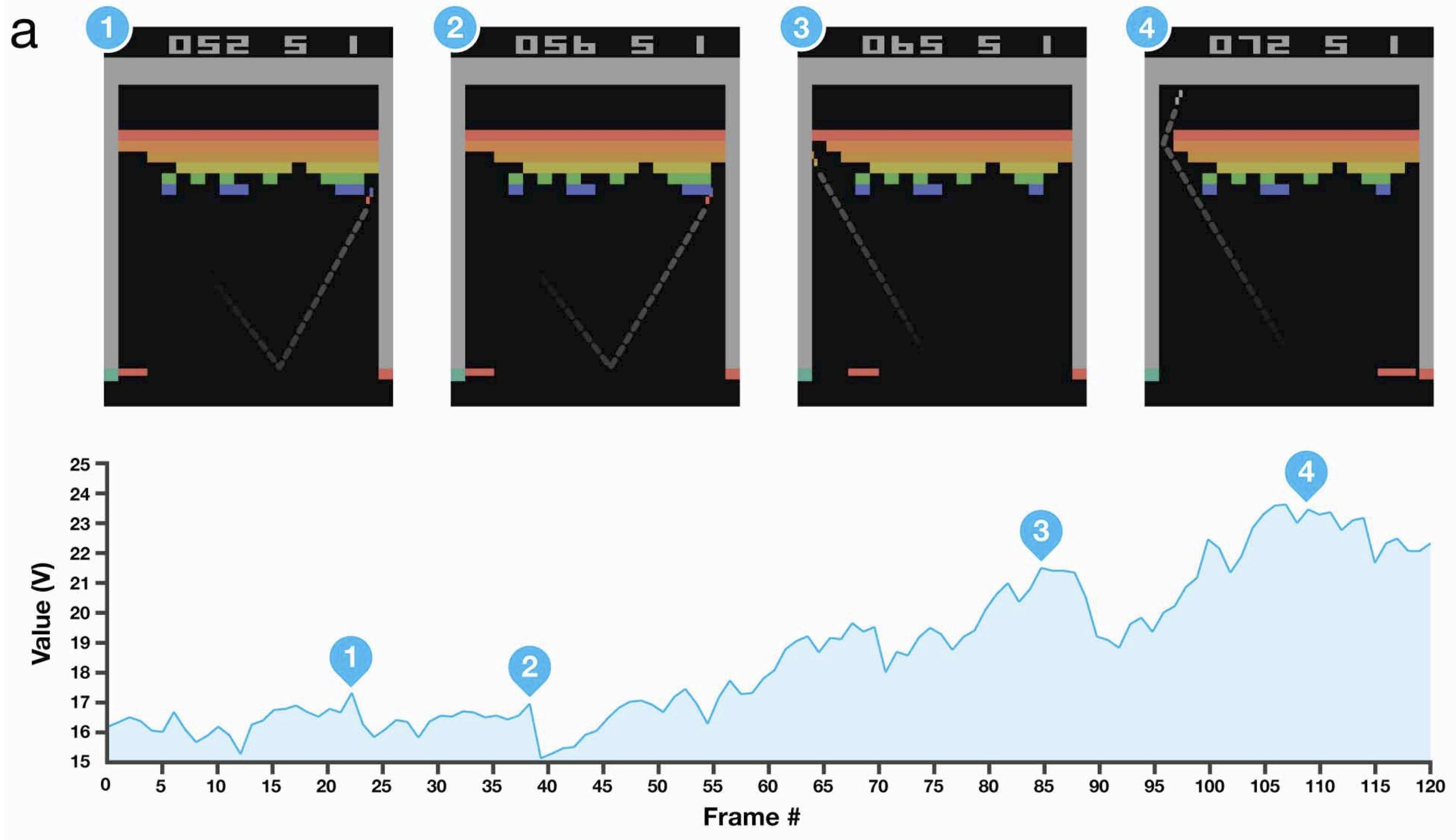
Perform a gradient step on  $(y_j - Q(S_j, A_j, \mathbf{w}))^2$  with respect to  $\mathbf{w}$

Every C steps reset  $\hat{Q} = Q$

# Example: Breakout

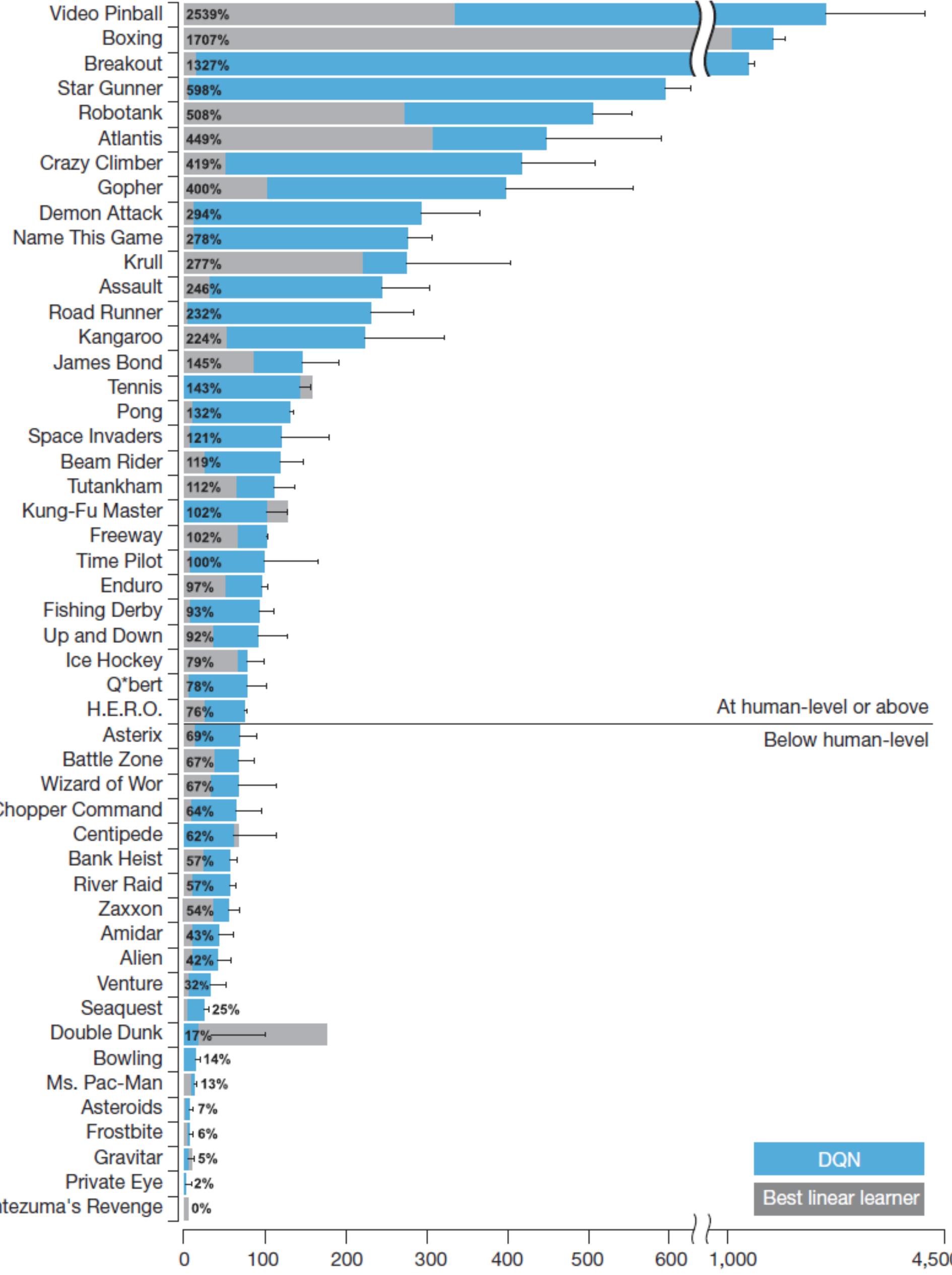


# Example: Breakout



# Results on Atari Games

Comparison of the DQN agent with other reinforcement learning methods and human game tester



# Prioritized Experience Replay

Idea:

- An agent should be able to learn more efficiently from some transitions than from others
- By what criterion can we measure the importance of each transition?

**TD error** indicates how surprising or unexpected the transition is, so samples with a high TD error should be prioritized

However:

- Only using the prioritized transitions can lead to errors too
- Use stochastic sampling

# Prioritized Experience Replay

The probability of sampling a transition  $i$  is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where  $p_i$  is the priority of the transition  $i$

Two approaches for calculating  $p_i$ :

$$p(i) = |\delta_i| + \epsilon \quad p(i) = \frac{1}{\text{rank}(i)}$$

# Double Q-Learning

- Q-learning is known to overestimate action-values under certain conditions
- The Q function is used to select the best action and to evaluate it:

$$y_j^{\text{DQN}} \doteq R_j + \gamma Q(S_{j+1}, \arg \max_a Q(S_{j+1}, a, \mathbf{w}^-), \mathbf{w}^-)$$

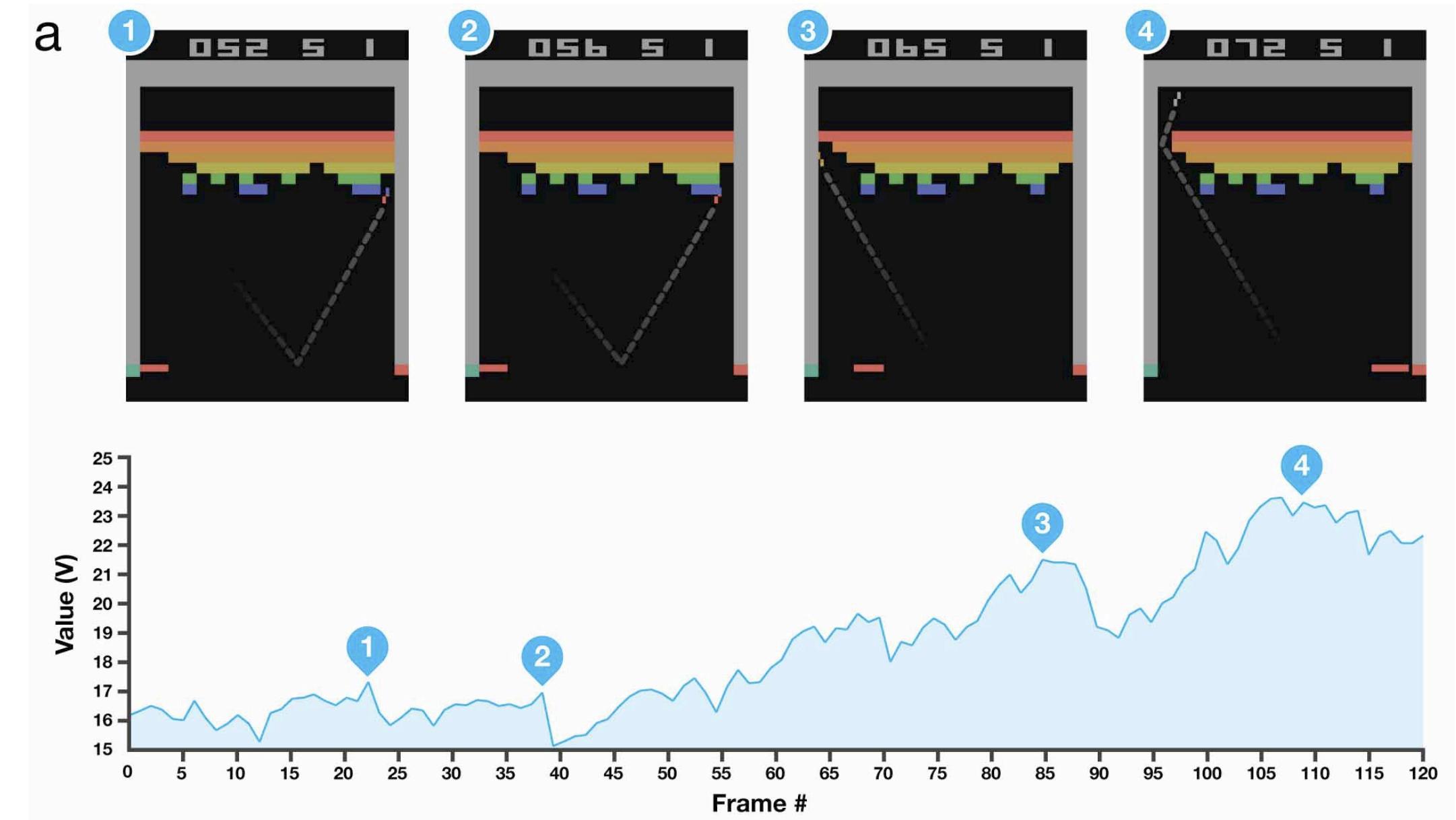
- it can be shown it is better to use different networks for those two tasks:

$$y_j^{\text{DoubleDQN}} \doteq R_j + \gamma Q(S_{j+1}, \arg \max_a Q(S_{j+1}, a, \mathbf{w}), \mathbf{w}^-)$$

# Conclusion

- The methods seen so far, DP, MC and TD Learning can be adapted to function approximation methods, where the state- or action-value function is approximated
- The object is to minimize the **mean squared error** between the function approximation and the target function
- Minimization is done using **stochastic gradient descent**
- Deep Q-Learning Methods use a **neural network** for function approximation and can tackle difficult RL problems by only using *raw* observation as input (no feature design)
- Training time is generally long
- In order to speed up training, replay buffers are used

# Function Approximation Methods



# Function Approximation

- Approximate the state- or action-value function by a parametrized function
- The objective (or loss function) is to minimize the mean square error between the value function and its approximation

$$\overline{VE}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

# Stochastic gradient descent

Assume that in each step we observe a state  $s$  and its (true) value under the policy.

We can use stochastic gradient-descend (SGD) by adjusting the weights vector to minimize the error in the observed examples

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \\ &= \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]\nabla \hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

# TD(0) Prediction

Semi-gradient TD(0) for estimating  $\hat{v} \approx v_\pi$

Input:

- a policy  $\pi$
- a differentiable function  $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$  with parameters  $\mathbf{w}$   
( $\hat{v}(\text{terminal}, \cdot)$ ) must be 0)
- a step size parameter  $\alpha > 0$

Initialize:

$\mathbf{w}$  arbitrarily

Loop for each episode)

    Initialize  $S$

    Loop for each step of the episode:

$A \leftarrow$  action given by  $\pi$  for  $S$

        Take action  $A$ , observe  $R, S'$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$

$S \leftarrow S'$

    until  $S$  is terminal

## Semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input:

- a differentiable function  $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$  with parameters  $\mathbf{w}$
- a step size parameter  $\alpha > 0$ , small  $\epsilon > 0$

Initialize:

$\mathbf{w}$  arbitrarily

Loop for each episode)

    Initialize  $S$

    Choose  $A$  as a function of  $\hat{q}(S, ., \mathbf{w})$  (e.g.,  $\epsilon$ -greedy)

    Loop for each step of the episode:

        Take action  $A$ , observe  $R, S'$

        If  $S'$  is terminal:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R - \hat{q}(S, A, \mathbf{w})]\nabla\hat{q}(S, A, \mathbf{w})$$

        Go to next episode

        Choose  $A'$  as a function of  $\hat{q}(S', ., \mathbf{w})$  (e.g.,  $\epsilon$ -greedy)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})]\nabla\hat{q}(S, A, \mathbf{w})$$

$$S \leftarrow S'$$

$$A \leftarrow A'$$

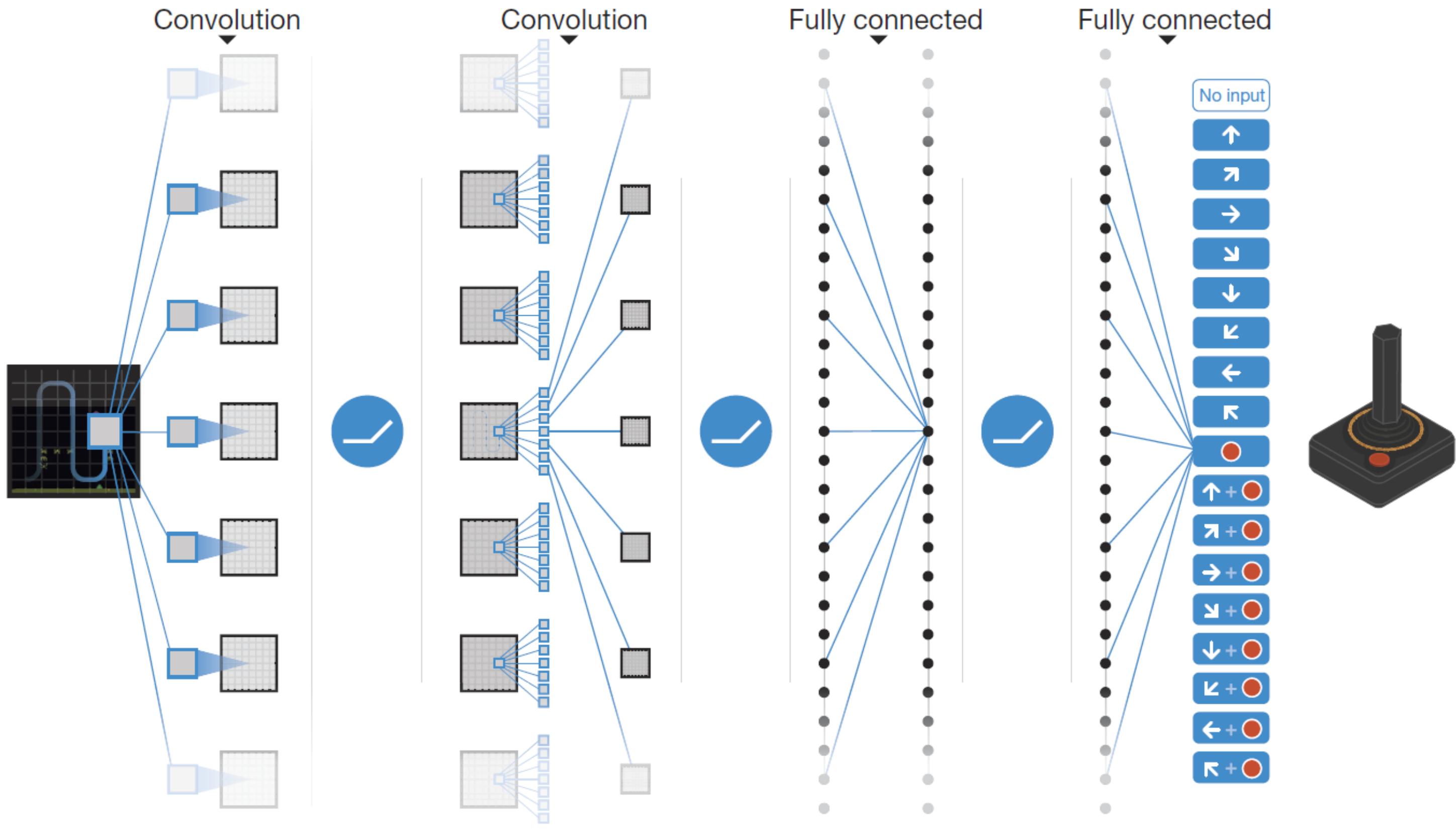
# Deep Q-Learning

Deep Q-Learning uses a deep Q network (DQN), that estimates the action value function

- Uses experience replay
- Updates the action-values iteratively towards target
- Target values are only updated periodically

$$J(\mathbf{w}_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[ (r + \gamma \max_a Q(s', a', \mathbf{w}_i^-)) - Q(s, a, \mathbf{w}_i)^2 \right]$$

# Deep Q-Learning: Network Configuration



## Deep Q-Learning with experience replay

Initialize:

the replay memory  $D$  to capacity  $N$

action-value function  $Q$  with random weights  $\mathbf{w}$

target action-value function  $\hat{Q}$  with weights  $\mathbf{w}^- = \mathbf{w}$

Loop for each episode:

Initialize  $S_1$

For every step  $t = 1, T$  in the episode:

Choose  $A_t$  as a function of  $Q(S_t, ., \mathbf{w})$  (e.g.,  $\epsilon$ -greedy)

Take action  $A_t$ , observe  $R_t, S_{t+1}$

Store transition  $(S_t, A_t, R_r, S_{t+1})$  in  $D$

Sample a random minibatch of transitions  $(S_j, A_j, R_j, S_{j+1})$  from  $D$

$$y_j = \begin{cases} R_j & \text{if } S_{j+1} \text{ is terminal} \\ R_j + \gamma \max_{A'} \hat{Q}(S_{j+1}, A', \mathbf{w}^-) & \text{otherwise} \end{cases}$$

Perform a gradient step on  $(y_j - Q(S_j, A_j, \mathbf{w}))^2$  with respect to  $\mathbf{w}$

Every C steps reset  $\hat{Q} = Q$

# Policy Gradient Methods



go right



Wait, I am still  
calculating Q-  
values.....

# Learning Objectives

- Define policies as parametrized functions
- Understand the advantages (and disadvantages) of parametrized policies over action-value based methods
- Understand the objective function for policy gradient methods
- Understand the policy-gradient theorem
- Describe the actor-critic algorithm for control with function approximation

# Policy Gradient Methods

- So far, almost all methods have been *action-value methods*
- Policies were only calculated from those action-value estimates (using GPI)
- We now turn to methods that directly learn a parametrized policy

$$\pi(a|s, \theta) = \Pr\{A_t = a | S_t = s, \theta_t = \theta\}$$

where  $\theta$  are the parameters (weights)

- (If we also need a parametrized value function, we will use  $w$  for its parameters to distinguish between the two function approximations)

# Constraints

The policy should be a probability over the different actions and must use exploration, therefor:

- The probability of any action should be greater than 0:

$$\pi(a|s, \theta) > 0, \quad \text{for all } a \in \mathcal{A}, s \in \mathcal{S}$$

- The sum of all probabilities must be 1:

$$\sum_a \pi(a|s, \theta) = 1, \quad \text{for all } s \in \mathcal{S}$$

# Softmax for action preferences

- One common possibility to ensure those constraints is to use parameterized action-preferences:

$$h(s, a, \theta) \in \mathbb{R}$$

- and then compute action probabilities using the softmax function:

$$\pi(a|s, \theta) \doteq \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}$$

# Advantages of Policy Parametrization

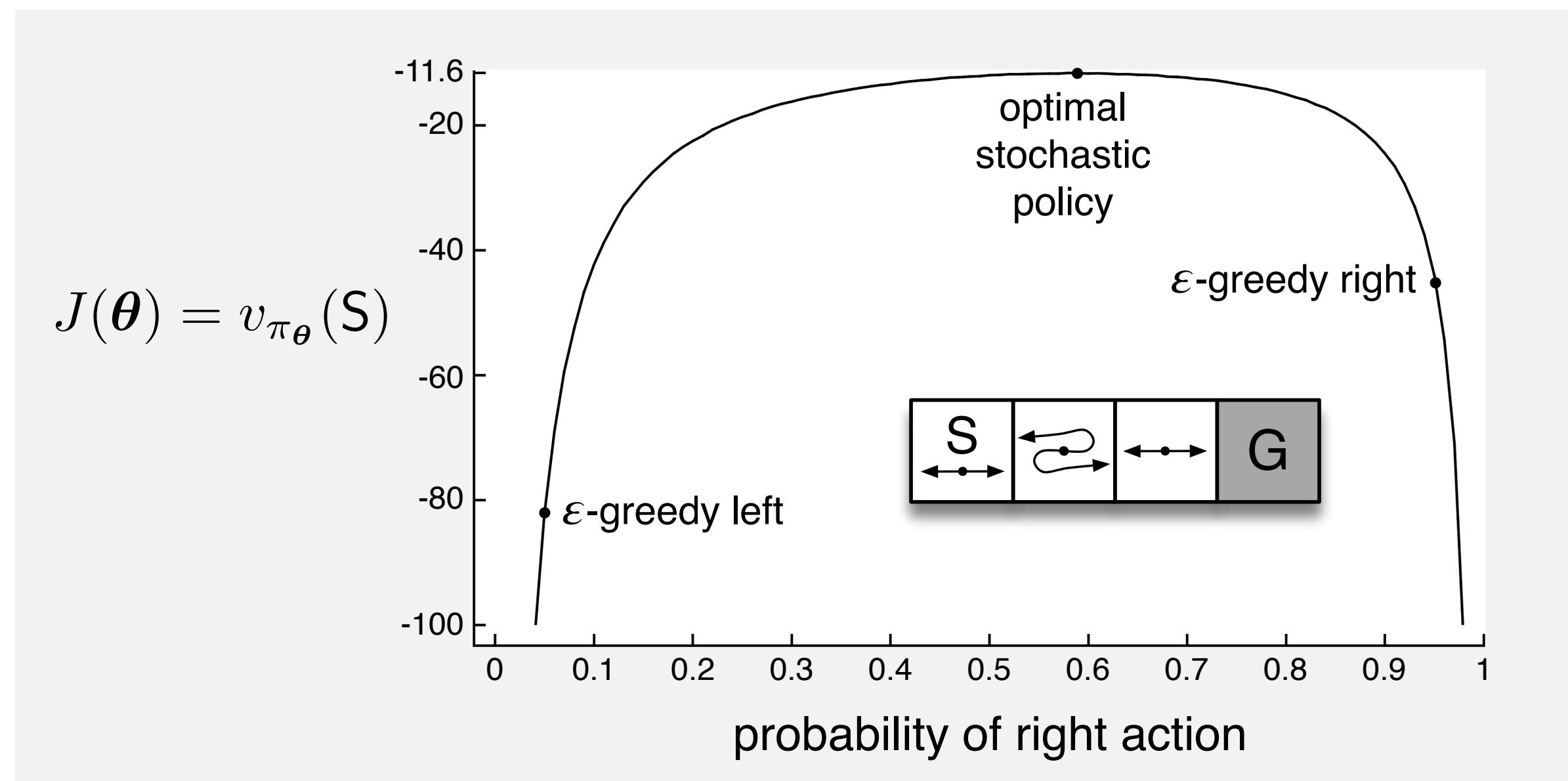
A parametrized policy

- can approach a deterministic policy over time (in comparison to epsilon-greedy, which will always explore)
- can model stochastic policies

# Example for stochastic policy:

Small stochastic corridor:

- Reward = -1 for all steps
- Actions are left / right
- Second state is reversed: if the action is left it will go right
- All states appear identical under function approximation



# Episodic case

- Goal: optimize the total return from a (particular) state

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0)$$

- Problem:  $v$  depends on state distribution
- Policy gradient theorem (see book for derivation):

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_{\pi}(s, a) \nabla \pi(a|s, \boldsymbol{\theta})$$

↑  
Policy Gradient

# REINFORCE: Monte Carlo Policy Gradient

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \boldsymbol{\theta}) \\&= \mathbb{E}_\pi \left[ \sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \boldsymbol{\theta}) \right] \\&= \mathbb{E}_\pi \left[ \sum_a \pi(a|s, \boldsymbol{\theta}) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t, \boldsymbol{\theta})}{\pi(a|s, \boldsymbol{\theta})} \right] \\&= \mathbb{E}_\pi \left[ q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|s, \boldsymbol{\theta})} \right], \quad \text{replacing } a \text{ by a sample } A_t \\&= \mathbb{E}_\pi \left[ G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|s, \boldsymbol{\theta})} \right], \quad \text{because } \mathbb{E}_\pi[G_t|S_t, A_t] = q_\pi(S_t, A_t)\end{aligned}$$

# REINFORCE: Monte Carlo Policy Gradient

- Update the weights according to:

$$\begin{aligned}\theta_{t+1} &\doteq \theta_t + \alpha G_t \frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | s, \theta)} \\ &= \theta_t + \alpha G_t \nabla \ln \pi(A_t | S_t, \theta)\end{aligned}$$

- The second line can be derived from

$$\nabla \ln(f(x)) = \frac{\nabla f(x)}{f(x)}$$

- This is gradient **ascent**, as we want to maximize the return

# REINFORCE

## REINFORCE: MC Policy-Gradient Control (episodic)

Input:

a differentiable policy parameterization  $\pi(a|w, \theta)$   
step size  $\alpha > 0$

Initialize:

policy parameters  $\theta$

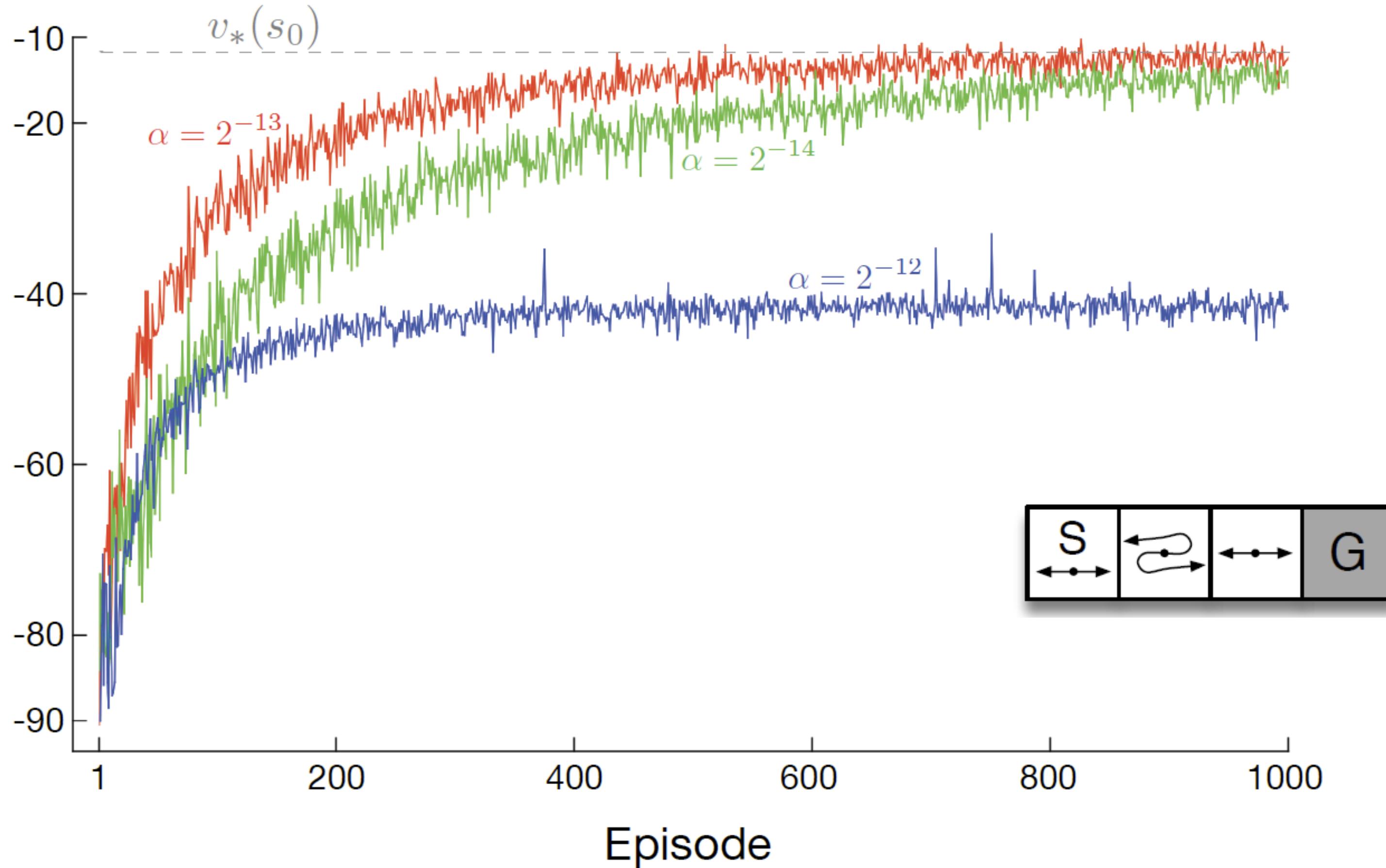
Loop for each episode:

Generate an episode following  $\pi$ :  $S_0, A_0, R_0, S_1, \dots, R_T$

For every step  $t = 0, T$  in the episode:

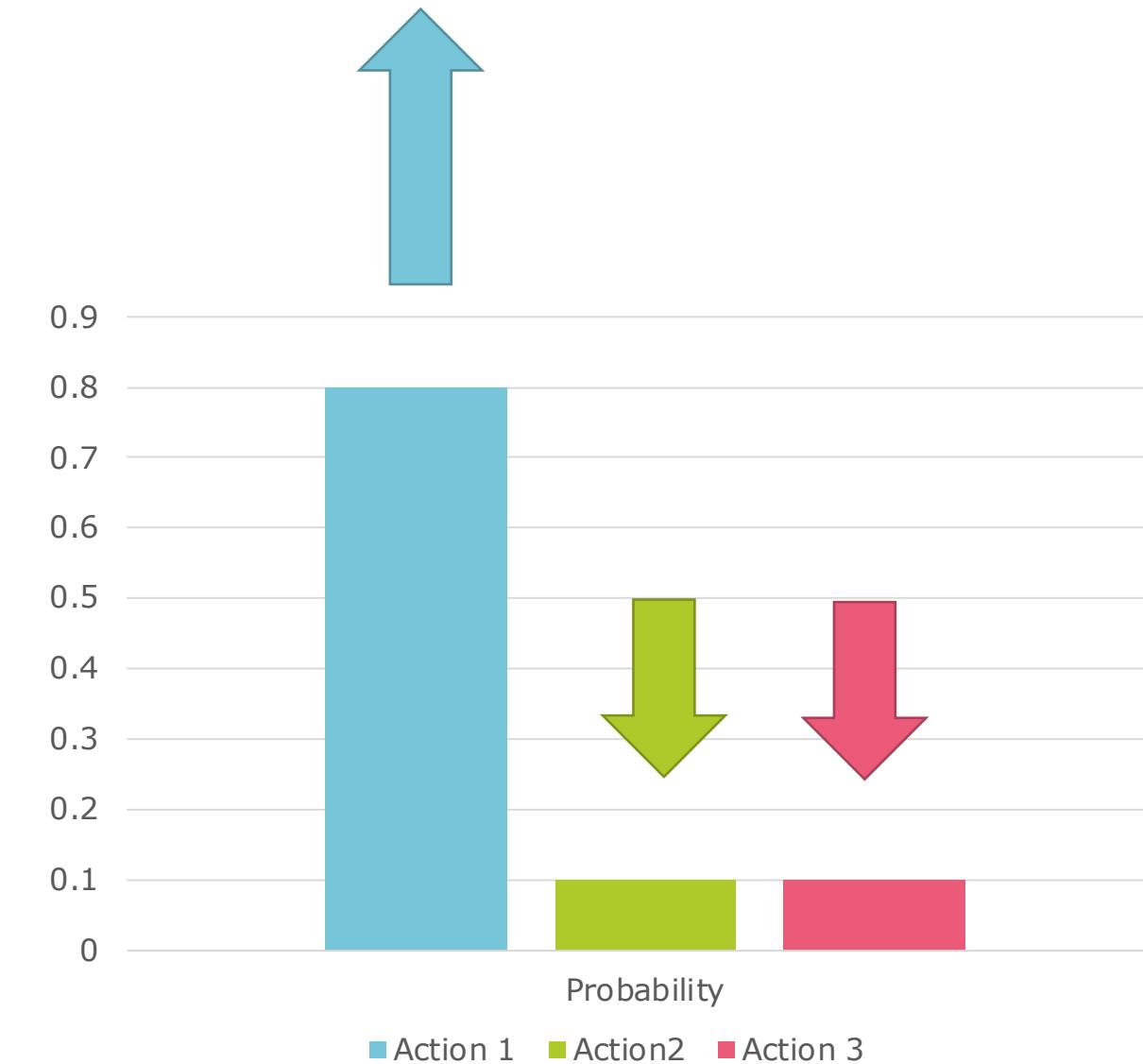
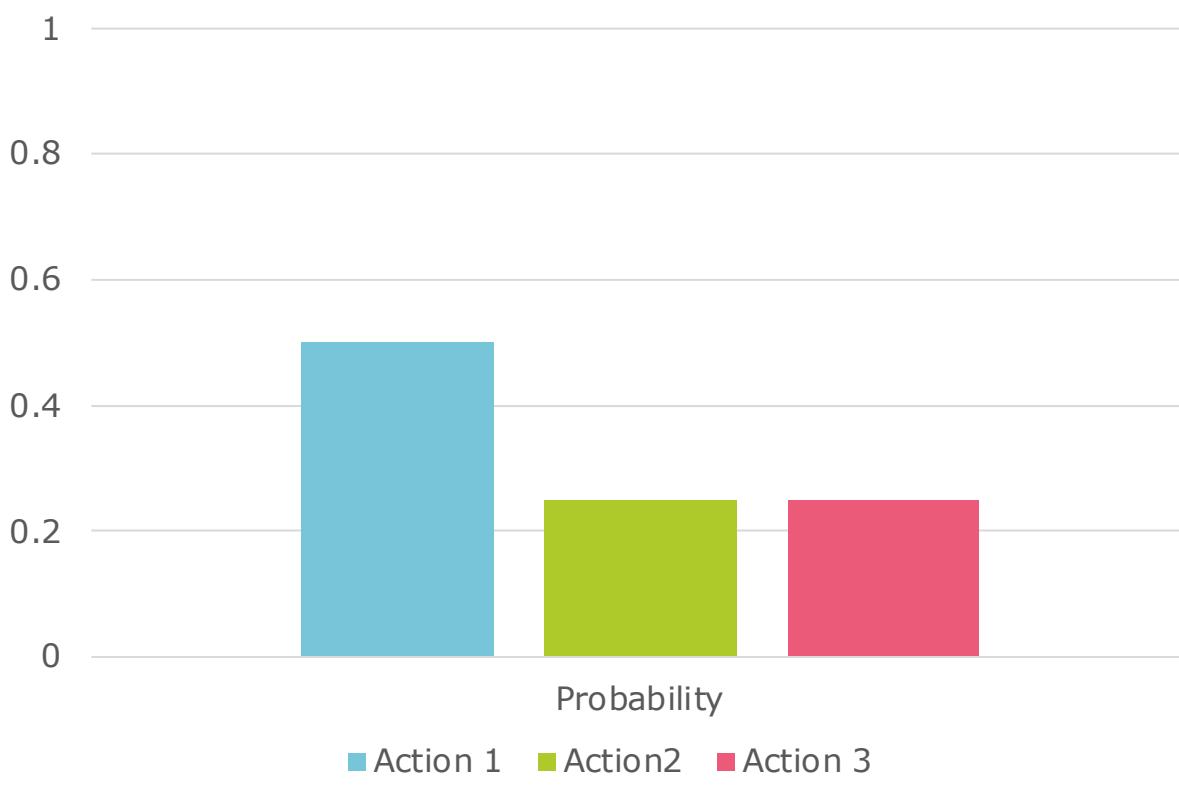
$$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k$$
$$\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A_t | S_t, \theta)$$

# Example



# How does Policy Gradient work?

- The gradient will push the selected action to have a higher probability (at the expense of the others)
- The amount that it will be pushed depends on the return  $G$



# Baseline

A "trick" for faster convergence is to subtract a baseline from the q values, where the baseline can be any function that does not depend on the action

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a (q_\pi(s, a) - b(s)) \nabla \pi(a|s, \boldsymbol{\theta})$$

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha(G_t - b(S_t)) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|s, \boldsymbol{\theta})}$$

# Baseline

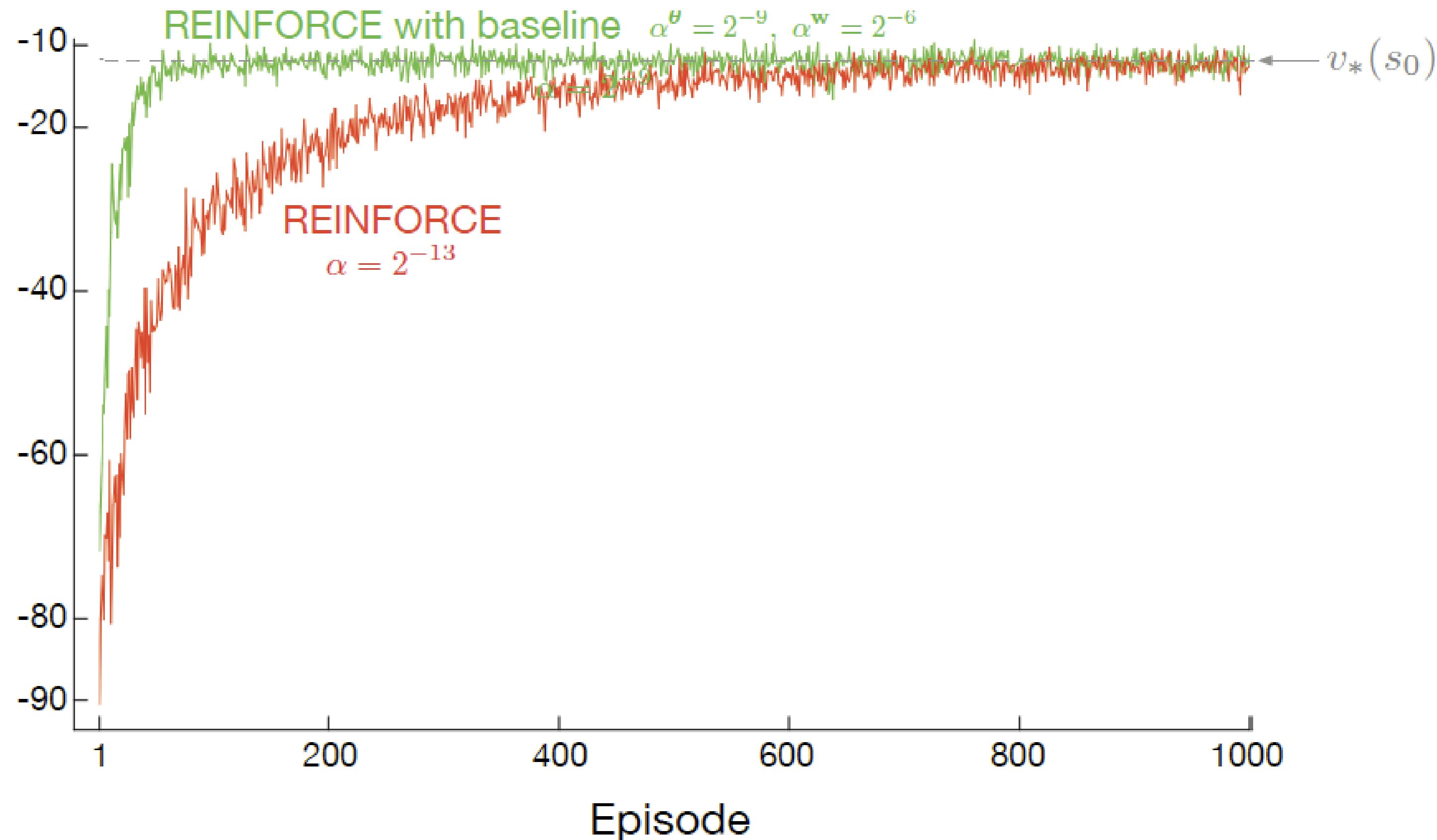
- A natural choice for  $b$ , is to estimate a state value

$$\hat{v}(S_t, \mathbf{w})$$

- where the weights would also be learned using the Monte-Carlo method
- The function is not dependent on the policy parametrization, so the gradient remains the same
- The difference between  $q$  and  $v$  is also called the *advantage* function:

$$q(s, a) - v(s)$$

# REINFORCE with Baseline



# Actor Critic: Policy Gradient Method with Critic

- We would like to implement a 1-step (or n-step) method like TD(0) for the policy
- However, we then need a **value** function, so we can replace the full return in REINFORCE with the 1-step return:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha (R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})) \nabla \ln \pi(A_t | S_t, \boldsymbol{\theta}_t) \\ &= \boldsymbol{\theta}_t + \alpha \delta_t \nabla \ln \pi(A_t | S_t, \boldsymbol{\theta}_t)\end{aligned}$$

- The policy is called the *actor*, the value function the *critic*

## One step Actor-Critic (episodic) for estimating $\pi_{\theta} \approx \pi_*$

Input:

a differentiable policy parameterization  $\pi(a|s, \theta)$

a differentiable state-value function parametrization  $\hat{v}(s, w)$

step sizes  $\alpha^{\theta} > 0, \alpha^w > 0$

Initialize:

policy parameters  $\theta$  and state-value weights  $w$

Loop for each episode:

Initialize  $S$ , first state of episode

$I \leftarrow 1$

For each time step of the episode:

Choose  $A \sim \pi(\cdot|S, \theta)$

Take action  $A$ , observe  $S', R$

$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$

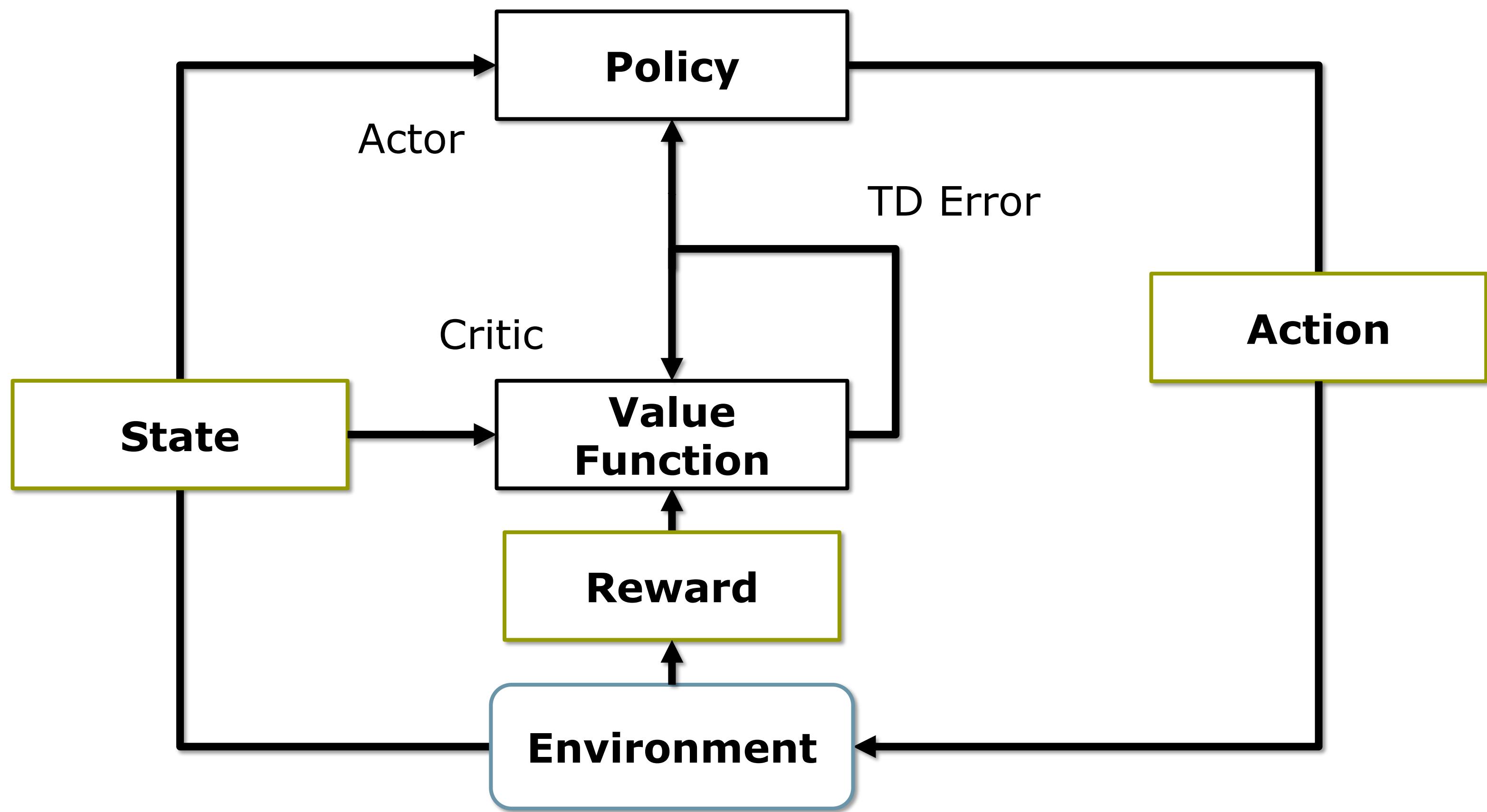
$w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S, w)$

$\theta \leftarrow \theta + \alpha^{\theta} I \delta \nabla \ln \pi(A|S, \theta)$

$I \leftarrow \gamma I$

$S \leftarrow S'$

# Actor-Critic



# Example: Flight Simulator Landing



# Average Reward for Continuing Tasks

- In the function approximation approaches, the discounted returns can be problematic
- We can instead use average-rewards

$$\begin{aligned} r(\pi) &\doteq \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{t=1}^h \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \lim_{h \rightarrow \infty} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)r \end{aligned}$$

# Average Rewards: Differential Settings

If we are using average rewards, the returns are defined in terms of the difference between the obtained rewards and the average reward:

$$G_t = \sum_t^{\infty} R_t - r(\pi)$$

Similarly, we can define the other value functions, for example for the *differential* state-value function, we get:

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a) [r - r(\pi) + v_{\pi}(s')]$$

## One step Actor-Critic (continuing) for estimating $\pi_{\theta} \approx \pi_*$

Input:

a differentiable policy parameterization  $\pi(a|w, \theta)$

a differentiable state-value function parametrization  $\hat{v}(s, w)$

Parameters:  $\alpha^{\theta} > 0, \alpha^w > 0, \alpha^R > 0$

Initialize:

$\bar{R} \in \mathbb{R}$ , for example to 0

policy parameters  $\theta$  and state-value weights  $w$

Initialize  $S \in \mathcal{S}$

Loop forever (for each time step):

Choose  $A \sim \pi(\cdot|S, \theta)$

Take action  $A$ , observe  $S', R$

$$\delta \leftarrow R - \bar{R} + \gamma \hat{v}(S', w) - \hat{v}(S, w)$$

$$\bar{R} \leftarrow \bar{R} + \alpha^R \delta$$

$$w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S, w)$$

$$\theta \leftarrow \theta + \alpha^{\theta} \delta \nabla \ln \pi(A|S, \theta)$$

$$S \leftarrow S'$$

# Advanced RL Methods

**Reinforcement Learning**  
December 1, 2022



Hollandwinkel.nl

# Trust Region Policy Optimization

In policy gradient methods, the update is calculated as

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

The new values for  $\theta$  should be near to the old values, as we use a small step size, however, the **policy** could still change significantly with a change of  $\theta$

We would like to make sure that the ***new and old policies*** are not too far apart (not that only the parameters are close)

---

## Trust Region Policy Optimization

---

John Schulman  
Sergey Levine  
Philipp Moritz  
Michael Jordan  
Pieter Abbeel

University of California, Berkeley, Department of Electrical Engineering and Computer Sciences

JOSCHU@EECS.BERKELEY.EDU  
SLEVINE@EECS.BERKELEY.EDU  
PCMORITZ@EECS.BERKELEY.EDU  
JORDAN@CS.BERKELEY.EDU  
PABBEEL@CS.BERKELEY.EDU

# Trust Region Policy Optimization

- We would like to compare the policies, but these are distributions (probabilities for each action)
- We can use the Kullback-Leibler divergence (KL-divergence)

$$D_{KL}(P||Q) \doteq \sum_x P(x) \log \frac{P(x)}{Q(x)}$$

# Trust Region Policy Optimization

We can write the cost function as a function of two policies and optimize that

$$J(\boldsymbol{\theta}, \boldsymbol{\theta}_t) = \mathbb{E}_{a \sim \pi} \left[ \frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_t}(a|s)} A(s, a) \right]$$

i.e. this measures how the new policy performs relative to the old policy, then we find

$$\boldsymbol{\theta}_{t+1} = \arg \max_{\boldsymbol{\theta}} J(\boldsymbol{\theta}, \boldsymbol{\theta}_t)$$

under the constraint

$$D_{KL}(\boldsymbol{\theta}_{t+1} || \boldsymbol{\theta}_t) \leq \delta$$

# Trust Region Policy Optimization

- Actual implementation is mathematically a bit more complicated 😊
- The constraint problem of optimization is solved by approximate solutions (conjugate gradient)
- As it is only an approximation, the constraints might be violated anyway and a line search through different steps sizes is done to ensure the constraint

# Proximal Policy Gradient (PPO)

- PPO tries to solve the same problem but uses clipping instead of a constraint on the KL-divergence
- Enforcing a constraint can also be viewed as imposing a penalty when the function gets near to it

Proximal Policy Optimization Algorithms

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov

OpenAI

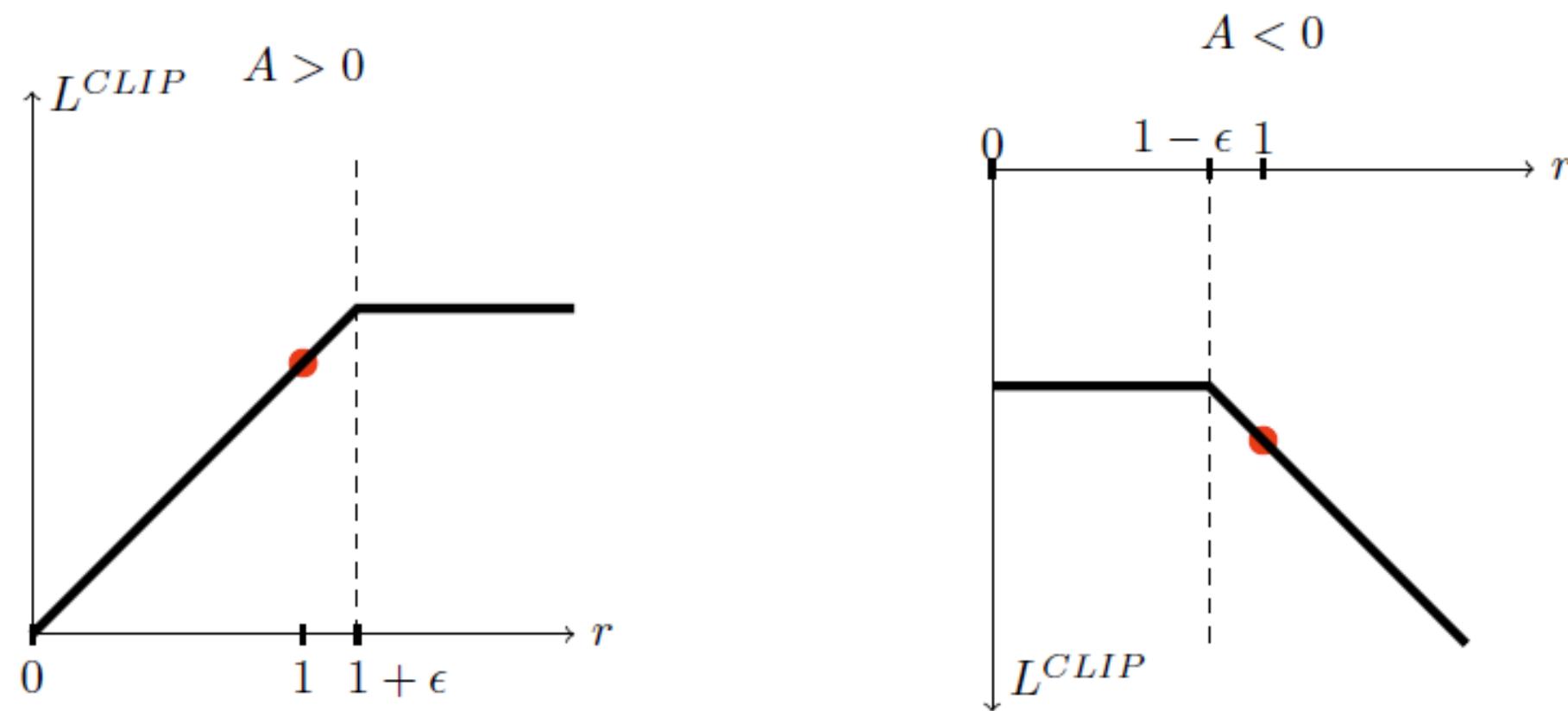
{joschu, filip, prafulla, alec, oleg}@openai.com

# PPO: Clipped Objective

(CPI: Conservative Policy Iteration)

$$L^{CPI}(\boldsymbol{\theta}) = \mathbb{E}_t \left[ \frac{\pi_{\boldsymbol{\theta}}(a_t, s_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_t | s_t)} A_t \right] = \mathbb{E}_t[r_t(\boldsymbol{\theta}) A_t]$$

$$L^{CLIP}(\boldsymbol{\theta}) = \mathbb{E}_t[\min(r_t(\boldsymbol{\theta}) A_t, \text{clip}(r_t(\boldsymbol{\theta}), 1 - \epsilon, 1 + \epsilon) A_t)]$$



# PPO: KL Penalty Coefficient

Calculate a penalty depending on the KL-divergence, but update the penalty parameter

$$L^{KLPEN}(\theta) = \mathbb{E}_t \left[ \frac{\pi_{\theta}(a_t, s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} A_t - \beta \text{KL}[\pi_{\theta_{\text{old}}}(\cdot, s_t), \pi_{\theta}(\cdot, s_t)] \right]$$

Calculate

$$d = \mathbb{E}_t [\text{KL}[\pi_{\theta_{\text{old}}}(\cdot, s_t), \pi_{\theta}(\cdot, s_t)]]$$

if d is small, decrease beta

if d is large, increase beta

# Comparison of Algorithms

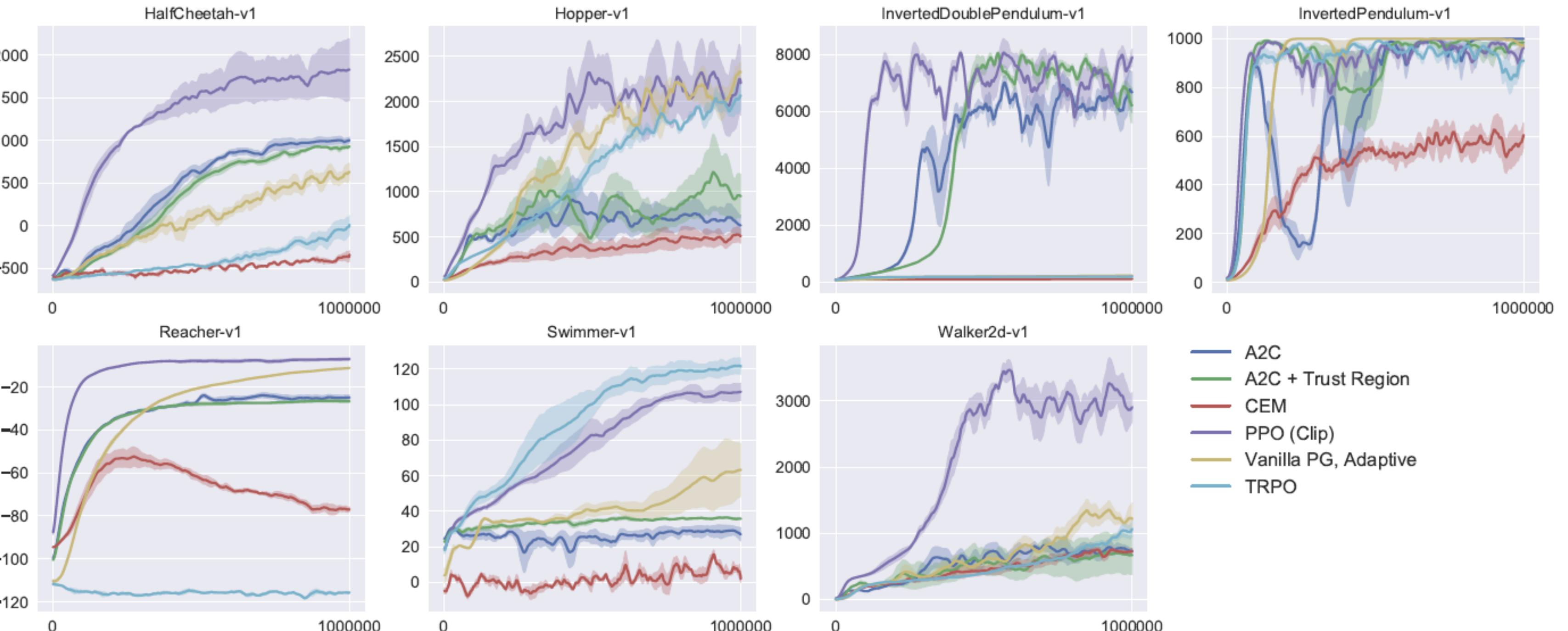


Figure 3: Comparison of several algorithms on several MuJoCo environments, training for one million timesteps.

# Examples

<https://openai.com/blog/openai-baselines-ppo/>

# Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) uses a combination of Deep Q-Learning and Policy Gradient

## Q-Learning:

- Q-Learning works well for discrete action spaces but cannot be well adapted to continuing actions.
- In order to find the maximum action from a continuous action space, we would need an expensive global maximization at every step
- Furthermore, DQN learns a deterministic policy and cannot learn stochastic policies

# **DDPG**

## **Policy Gradient:**

- Learn stochastic policy
- it is an on-policy approach, so it cannot use a different policy for exploring than the one being optimized
- Furthermore, as the policy changes constantly the "old" trajectories are not relevant anymore, making policy gradient *sample inefficient*.

# DDPG Algorithm

- Model free
- Off-policy
- Continuous and high-dimensional action space
- Actor-critic: policy network and action-value network
- Replay buffer: Store transitions and use them for training
- Target network: Use a target network, but using exponential averaging instead of copying the weights

# DDPG

- Use an actor function that deterministically maps states to actions:

$$\mu(s|\theta)$$

- Use a critic that is learned using the Bellman equation (as in Q-learning):

$$Q(s, a|w)$$

- Update the actor by applying the chain rule to the expected return:

$$\begin{aligned}\nabla_{\theta} J &\approx \mathbb{E}_{s_t \sim \rho^{\beta}} [\nabla_{\theta} Q(s, a|w)|_{s=s_t, a=\mu(s_t|\theta)}] \\ &= \mathbb{E}_{s_t \sim \rho^{\beta}} [\nabla_a Q(s, a|w)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta} \mu(s|\theta)|_{s=s_t}]\end{aligned}$$

# DDPG

- Actor and critic are updated by sampling a mini-batch from the replay buffer
- The targets weights are changed slowly:

$$\begin{aligned}\theta' &\leftarrow \tau\theta + (1 - \tau)\theta' \\ \mathbf{w}' &\leftarrow \tau\mathbf{w} + (1 - \tau)\mathbf{w}'\end{aligned}$$

- The behavior policy is constructed by adding noise to the actor function:

$$\mu'(s_t) = \mu(s_t | \theta_t) + \mathcal{N}$$

---

## Algorithm 1 DDPG algorithm

---

Randomly initialize critic network  $Q(s, a|\theta^Q)$  and actor  $\mu(s|\theta^\mu)$  with weights  $\theta^Q$  and  $\theta^\mu$ .

Initialize target network  $Q'$  and  $\mu'$  with weights  $\theta^{Q'} \leftarrow \theta^Q$ ,  $\theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer  $R$

**for** episode = 1, M **do**

    Initialize a random process  $\mathcal{N}$  for action exploration

    Receive initial observation state  $s_1$

**for** t = 1, T **do**

        Select action  $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$  according to the current policy and exploration noise

        Execute action  $a_t$  and observe reward  $r_t$  and observe new state  $s_{t+1}$

        Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $R$

        Sample a random minibatch of  $N$  transitions  $(s_i, a_i, r_i, s_{i+1})$  from  $R$

        Set  $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$

        Update critic by minimizing the loss:  $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

        Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

**end for**

**end for**

---

# Policy Gradient Methods Summary



go right



Wait, I am still  
calculating Q-  
values.....

# Policy Gradient Methods

- Learn the policy function directly

$$\pi(a|s, \theta) = \Pr\{A_t = a | S_t = s, \theta_t = \theta\}$$

- Constraints:

$$\pi(a|s, \theta) > 0, \quad \text{for all } a \in \mathcal{A}, s \in \mathcal{S}$$

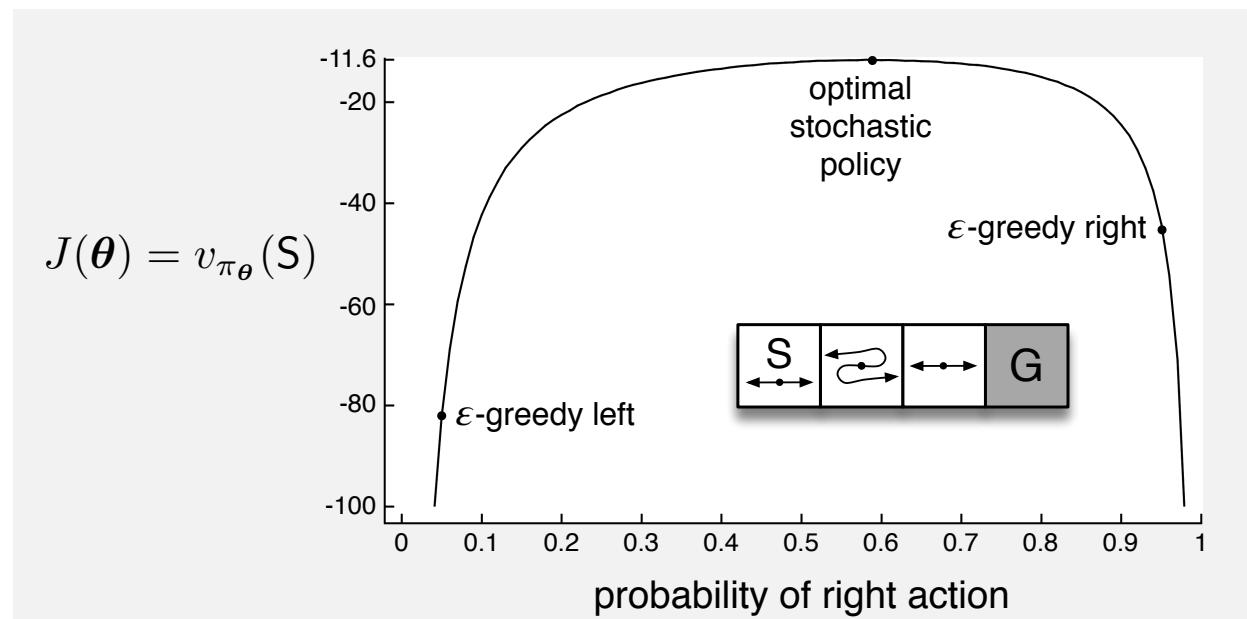
$$\sum_a \pi(a|s, \theta) = 1, \quad \text{for all } s \in \mathcal{S}$$

which can be enforced by using the softmax function

# Advantages of Policy Parametrization

A parametrized policy

- can approach a deterministic policy over time (in comparison to epsilon-greedy, which will always explore)
- can model stochastic policies



# Policy Gradient Theorem

- Goal: optimize the total return from a (particular) state

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0)$$

- The gradient of the Loss function is proportional to the gradient of the policy

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_{\pi}(s, a) \nabla \pi(a|s, \boldsymbol{\theta})$$

## REINFORCE: Monte Carlo Policy Gradient

- Update the weights according to:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t | S_t, \boldsymbol{\theta})}{\pi(A_t | s, \boldsymbol{\theta})} \\ &= \boldsymbol{\theta}_t + \alpha G_t \nabla \ln \pi(A_t | S_t, \boldsymbol{\theta})\end{aligned}$$

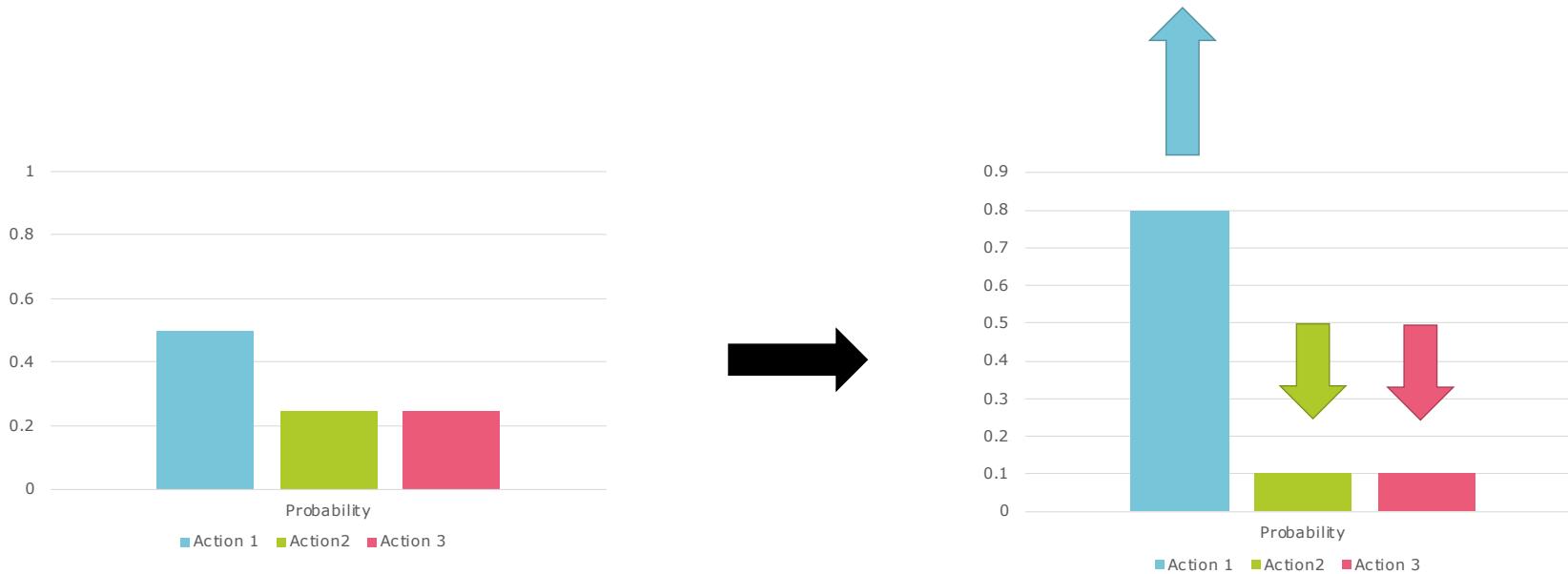
- The second line can be derived from

$$\nabla \ln(f(x)) = \frac{\nabla f(x)}{f(x)}$$

- This is gradient **ascent**, as we want to maximize the return

# How does Policy Gradient work?

- The gradient will push the selected action to have a higher probability (at the expense of the others)
- The amount that it will be pushed depends on the return  $G$



## Actor Critic: Policy Gradient Method with Critic

- For the 1-step return target function, we need a **value** function

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha (R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})) \nabla \ln \pi(A_t | S_t, \boldsymbol{\theta}_t) \\ &= \boldsymbol{\theta}_t + \alpha \delta_t \nabla \ln \pi(A_t | S_t, \boldsymbol{\theta}_t)\end{aligned}$$

- The value function is approximated too

## One step Actor-Critic (episodic) for estimating $\pi_{\theta} \approx \pi_*$

Input:

- a differentiable policy parameterization  $\pi(a|s, \theta)$
- a differentiable state-value function parametrization  $\hat{v}(s, w)$
- step sizes  $\alpha^{\theta} > 0, \alpha^w > 0$

Initialize:

policy parameters  $\theta$  and state-value weights  $w$

Loop for each episode:

    Initialize  $S$ , first state of episode

$I \leftarrow 1$

    For each time step of the episode:

        Choose  $A \sim \pi(\cdot|S, \theta)$

        Take action  $A$ , observe  $S', R$

$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$

$w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S, w)$

$\theta \leftarrow \theta + \alpha^{\theta} I \delta \nabla \ln \pi(A|S, \theta)$

$I \leftarrow \gamma I$

$S \leftarrow S'$

## TRPO and PPO Methods

In policy gradient methods, the update is calculated as

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

The new values for  $\theta$  should be near to the old values, as we use a small step size, however, the **policy** could still change significantly with a change of  $\theta$

TRPO and PPO Methods ensure that the policy does not change too much

## PPO

$$p_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_t|s_t)}$$

$$L^{CLIP}(\boldsymbol{\theta}) = \mathbb{E}_t[\min(r_t(\boldsymbol{\theta})A_t, \text{clip}(r_t(\boldsymbol{\theta}), 1 - \epsilon, 1 + \epsilon)A_t)]$$

- Clip the advantage function when it is outside of the comfort interval bounded by  $1 \pm \epsilon$
- Due to the minimum operation, the probability ratio is ignored outside the zone only if it would improve the objective, it is included if it makes the objective worse

# Multi Agent Reinforcement Learning

**Reinforcement Learning**  
December 22, 2022

FH Zentralschweiz



# **Multi Agent Reinforcement Learning**

Multi-Agent Systems have a long history:

"A multi-agent system is a group of autonomous, interacting entities sharing a common environment, which they perceive with sensors and upon which they act with actuators"

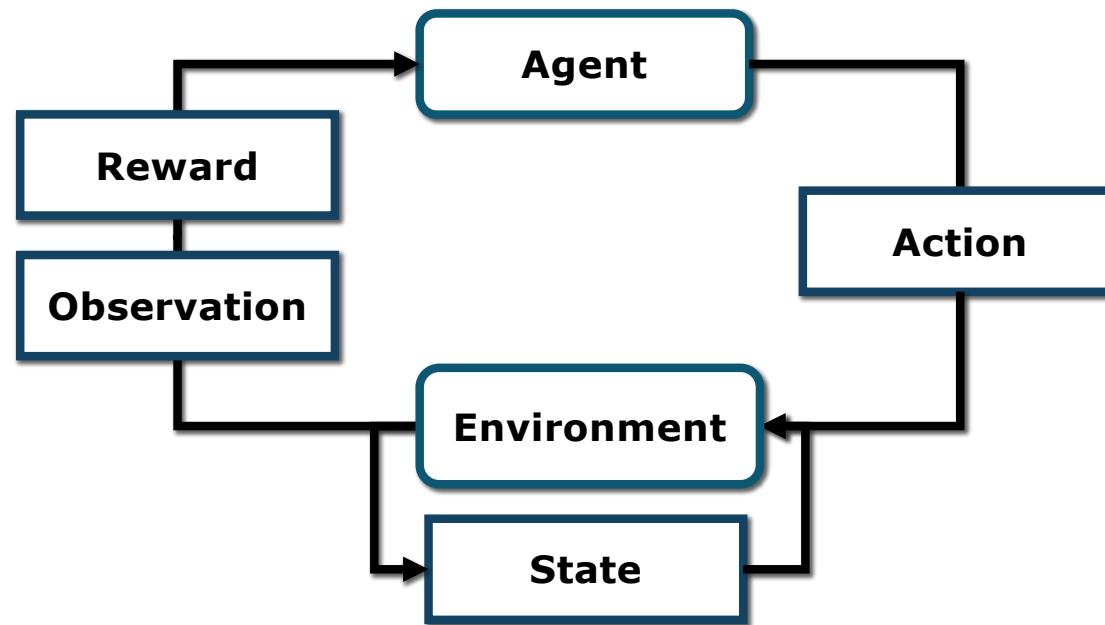
In Reinforcement Learning

- Shared Environment
- Each agent has its own observations, rewards, actions and goals
- (Agents might need to coordinate to achieve their own goals)

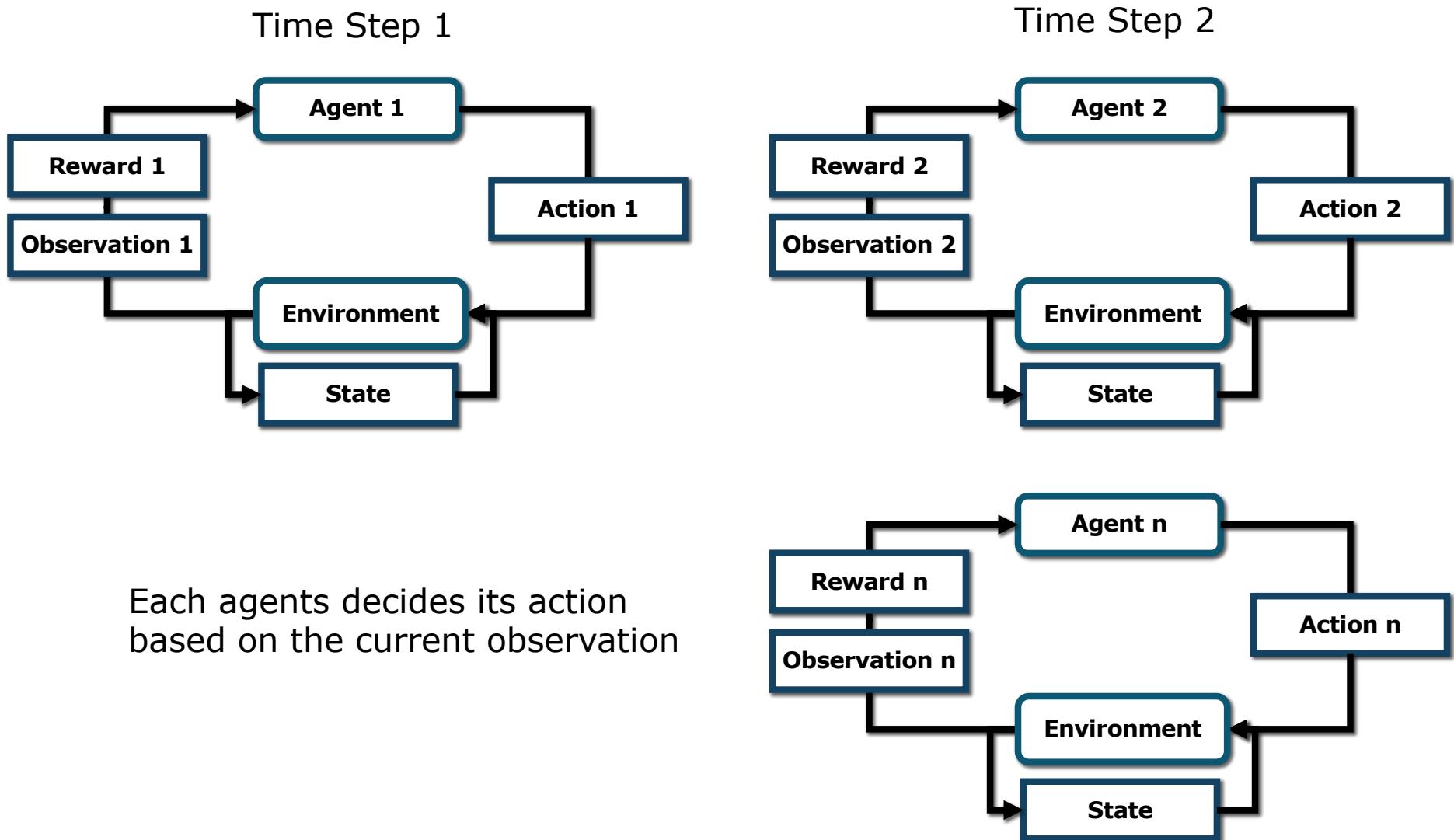
# MARL Agents

- Agents could be **collaborative** with the same goal and only receive a joint rewards.
- Agents could be collaborative with same or different goals and receive individual and/or a joint reward:
  - For example if multiple steps are necessary to reach the goal, the agents could get rewards for each step and for the goal
- Agents could be **competitive** with different goals (and rewards), for example in competitive games

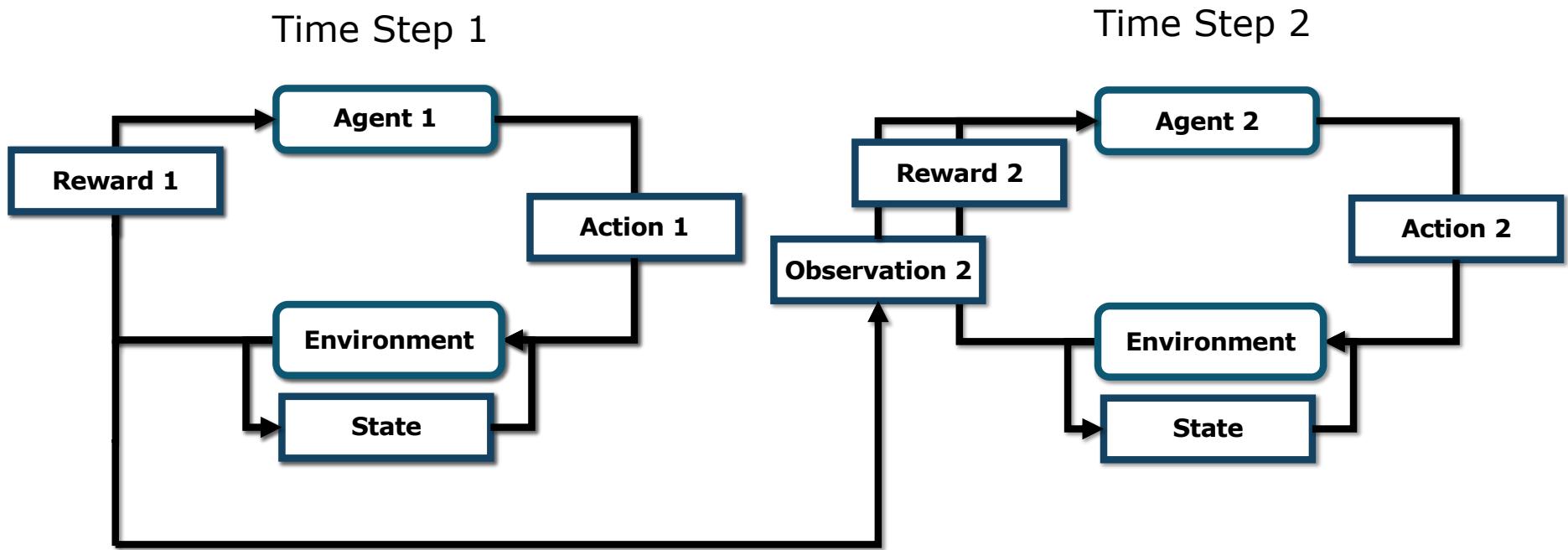
# Single Agent RL Environment



# Turn based environments

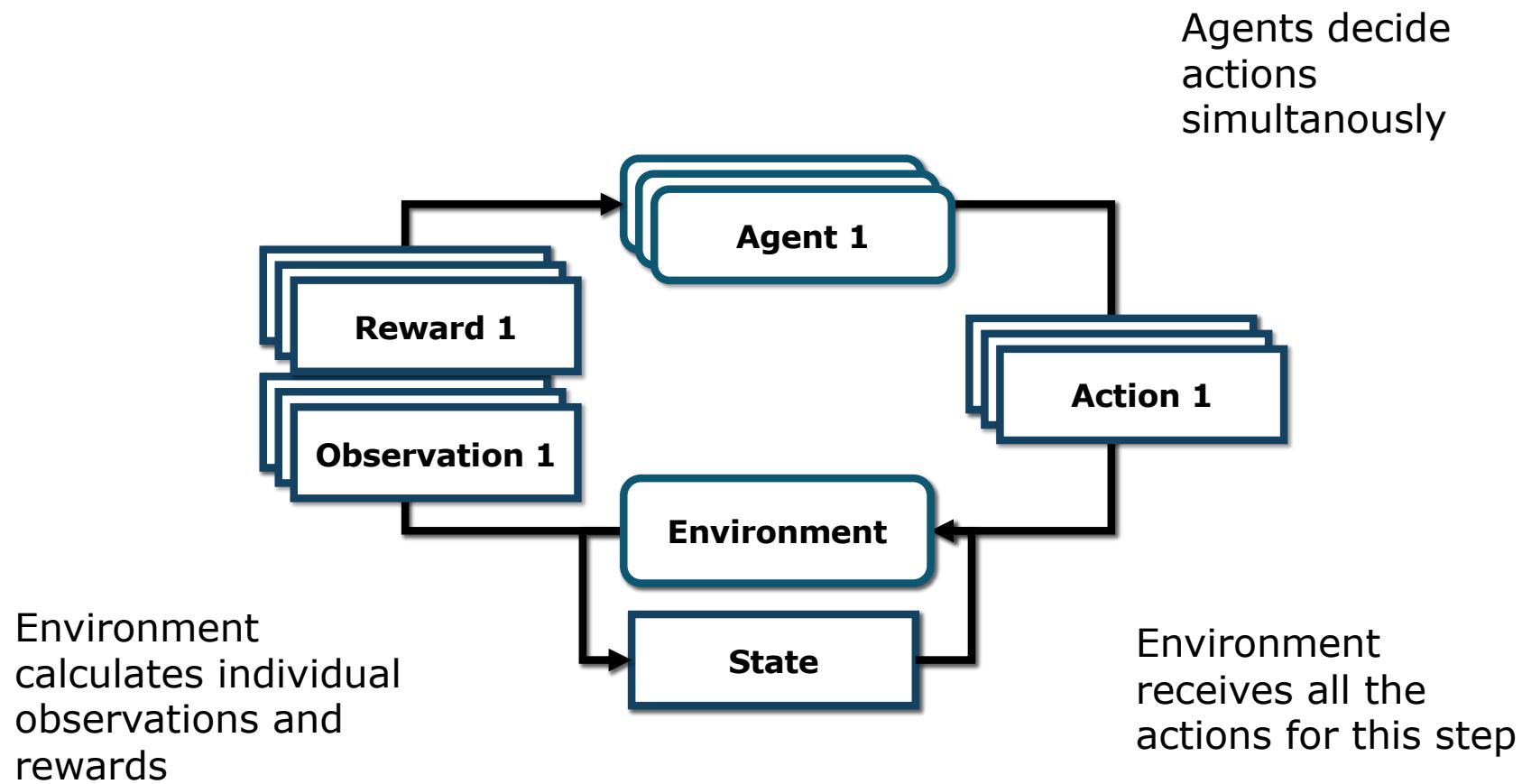


# Turn based environments

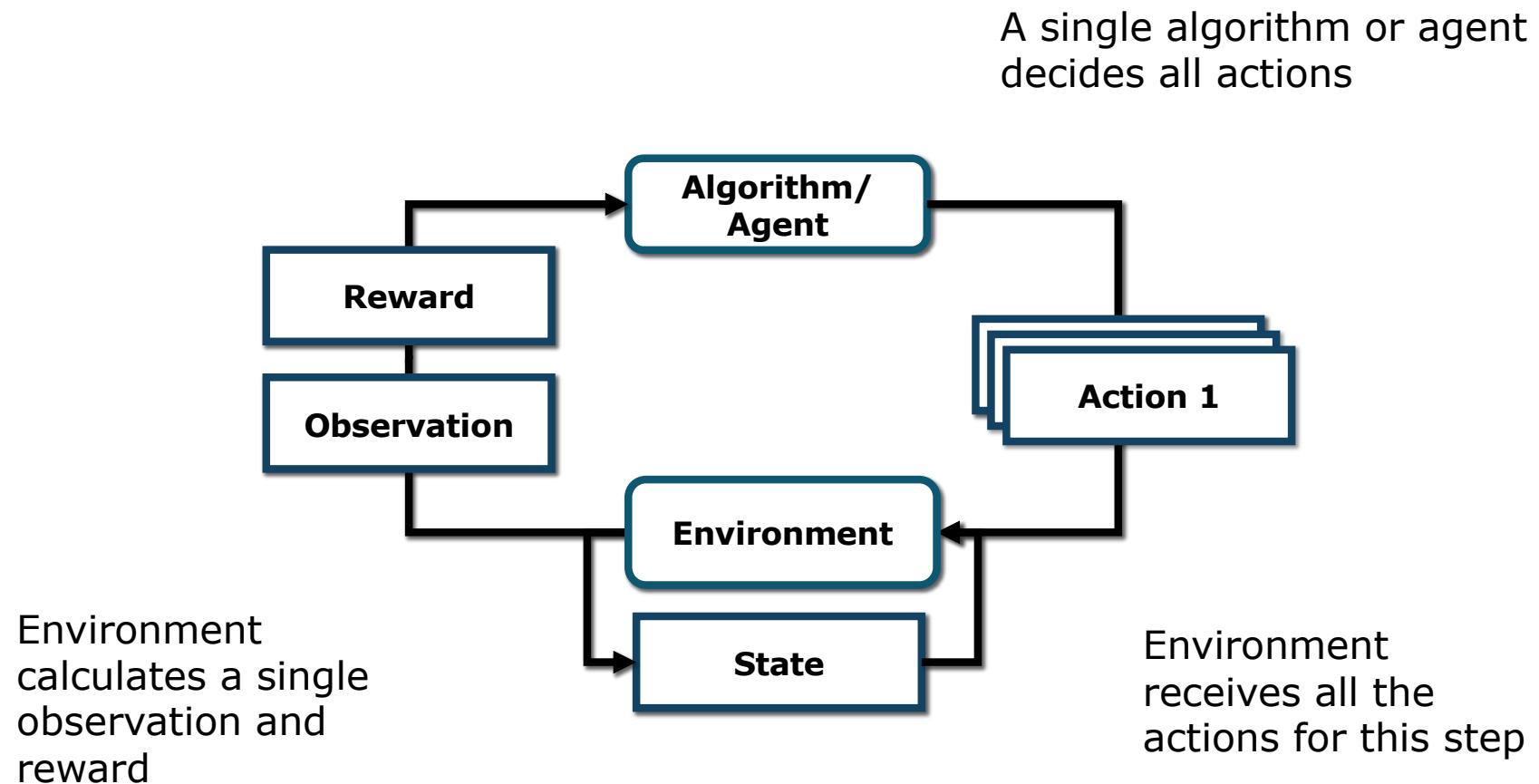


- The action from one player will change the environment, which then can generate the observation for the next player
- Rewards could be available immediately or only after all players have done an action
- For example in a turn based game such as Nine Mens Morris (Mühle), Jass or Poker

# Simultaneous, Pseudo-Real Time



# Joint Actions by a single Algorithm / Agent



# Challenges in Multi Agent Reinforcement Learning

- Non-Stationary environment:
  - For a single agent, the other agents act as a change in the environment
- Scalability (observations, joint action spaces)
  - Action spaces may grow non-linearly with the number of agents, making it infeasable to directly learn them
- Sparsity of Rewards
  - Complicated environments and tasks may only give rewards sparingly, for example at the end of a game. This makes it hard to train an agent.

# Examples / Applications

Games (Chess, Go, League of Legends, StarCraft)



Robotic Championships



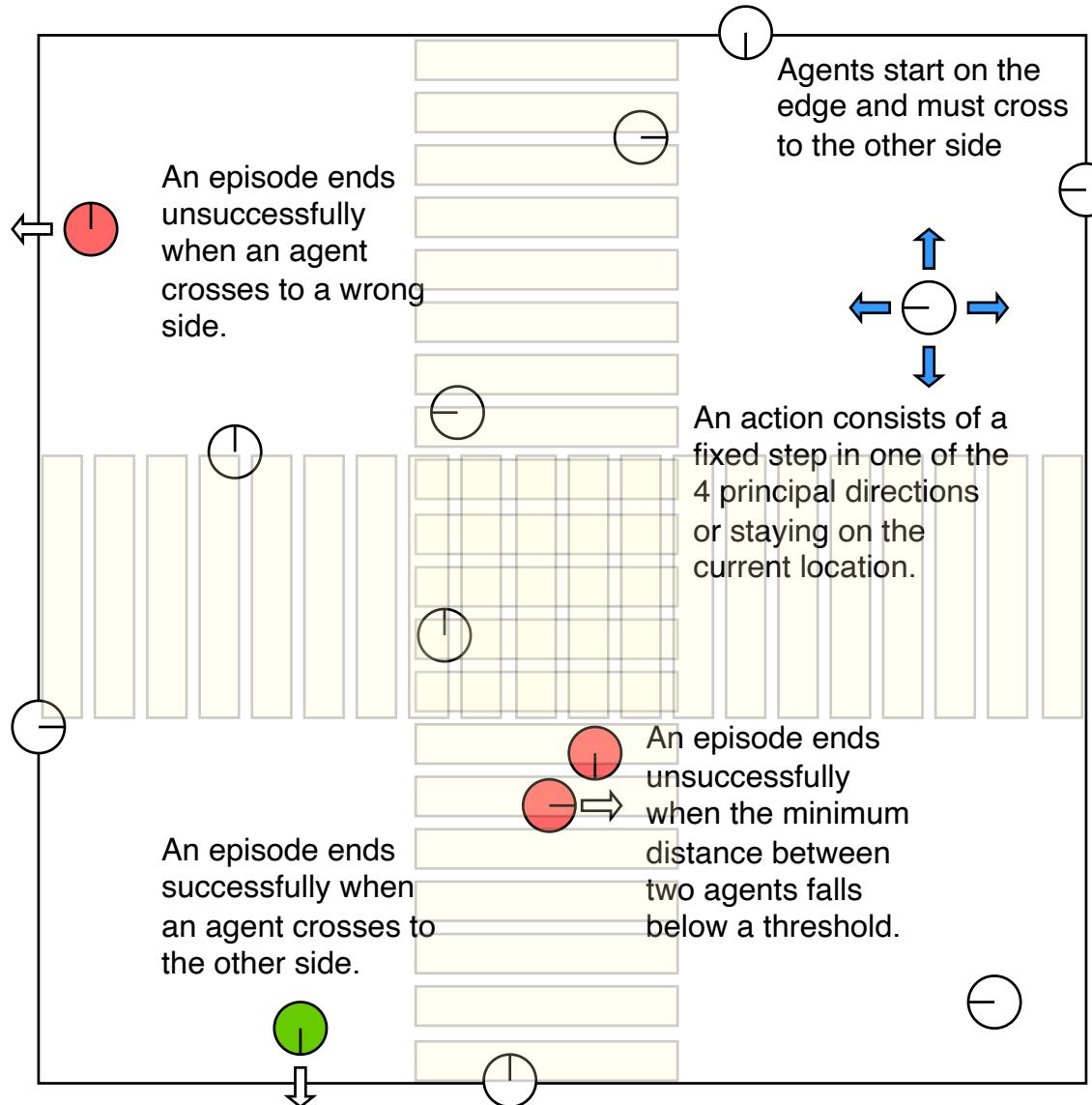
Market Negotiations



# Tokyo Shibuya Crossing



# Crossing Simulation



# Crossing Simulation

- Each bot is controlled by one agent.
- All agents have identical tasks and identical goals
- All agents use the same algorithm / policy
- The environment is unknown, there is no model that can be used to try actions and the exits are not known (but modelled by some rewards)

# Observations

Observation	Type	Remarks
Position	[2]	Position of bot
Velocity	[2]	Velocity vector of bot (due to last action)
Goal	[4]	One-hot encoded (N, E, S, W)
Distance to goal	[1]	Distance to goal

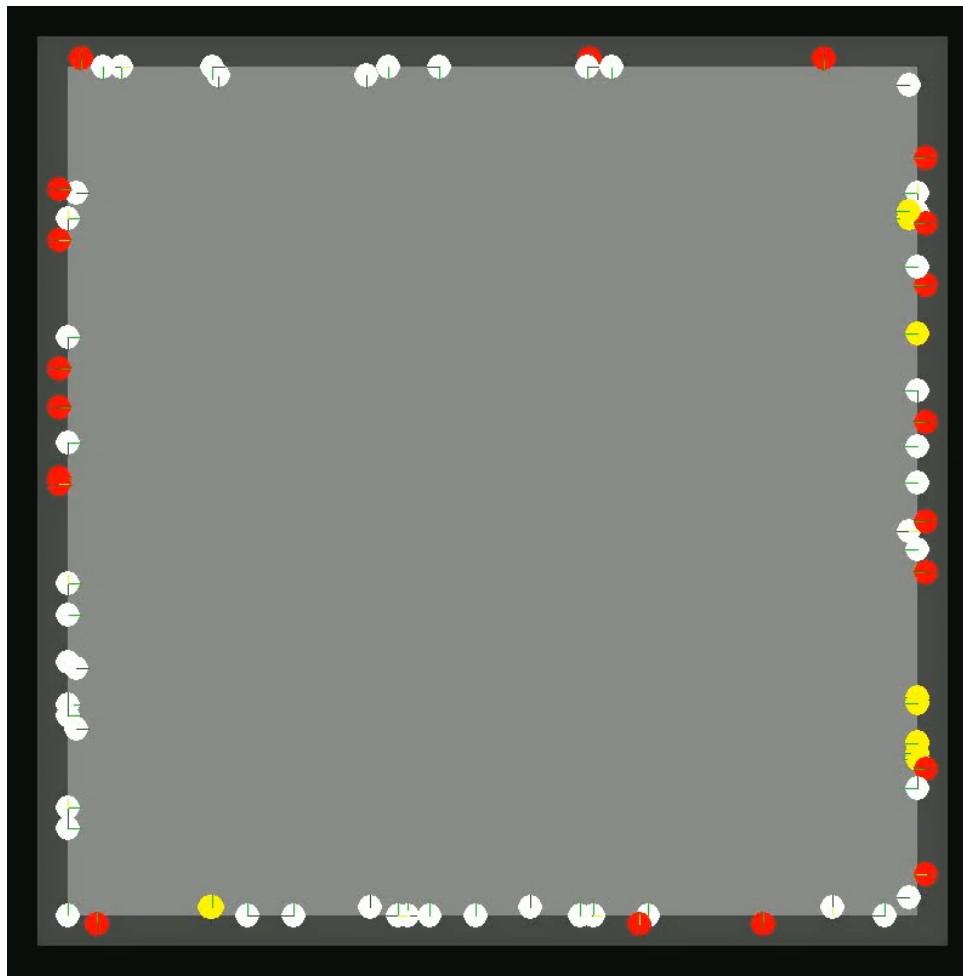
Edge Observation	Type	Remarks
Direction to neighbor	[2]	Direction to other bot
Distance to neighbor	[2]	Distance to other bot

# Rewards

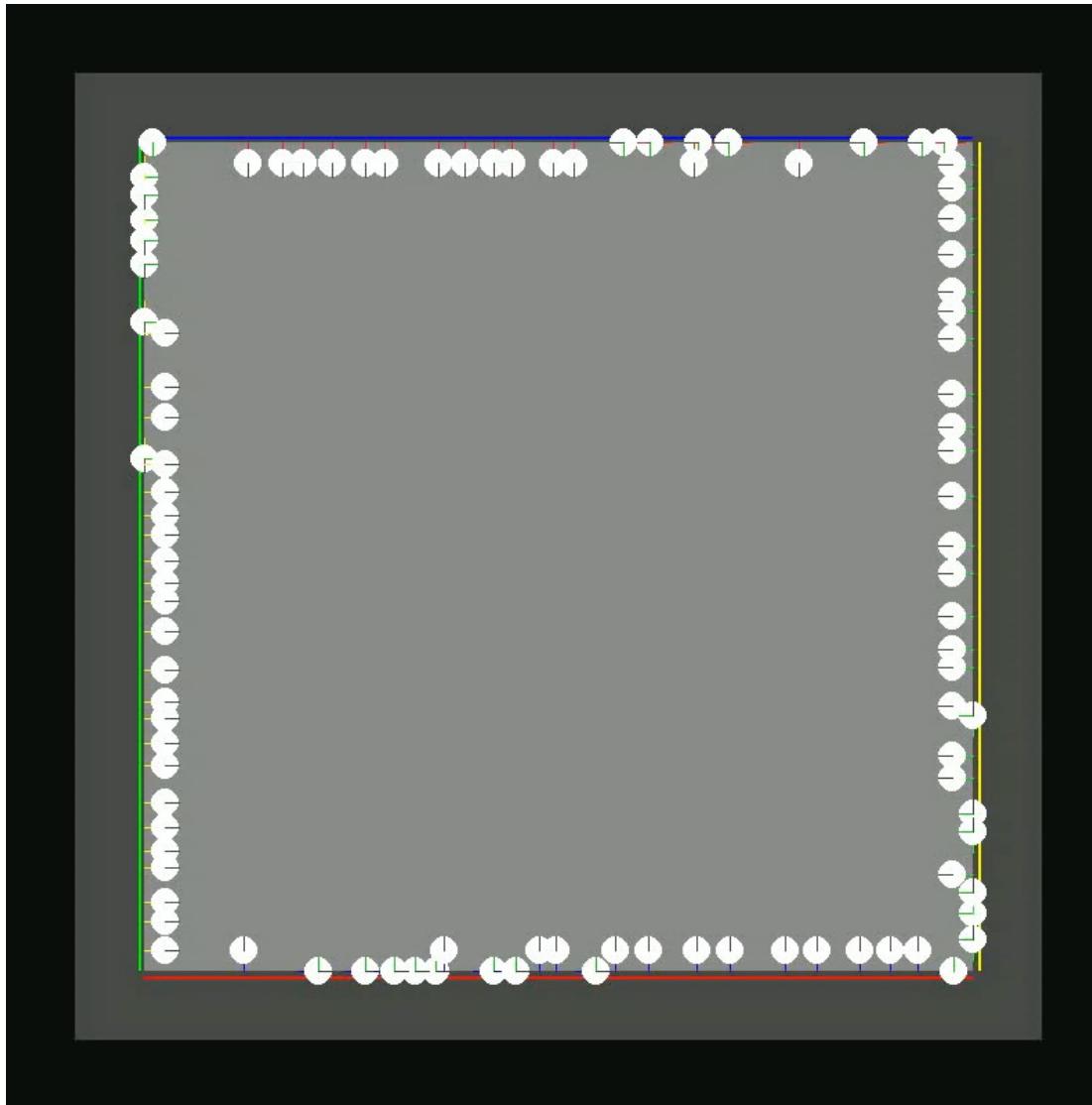
Reward	Default Value	Remarks
Timestep	-0.01	Reward at each time step
Distance to goal	0.01	Linearly decreases with distance from goal
Distance to side	0.002	Linearly increases with distance from wrong sides until a maximal distance
Distance to other bots		Linearly increases with distance from other bots up to maximal distance
Fall	-1.0	Reset to start position
Exit wrong side	-1.0	Reset to start position
Exit correct side	1.0	Goal reached, episode ends

# Crossing: Metrics for Reinforcement Learning

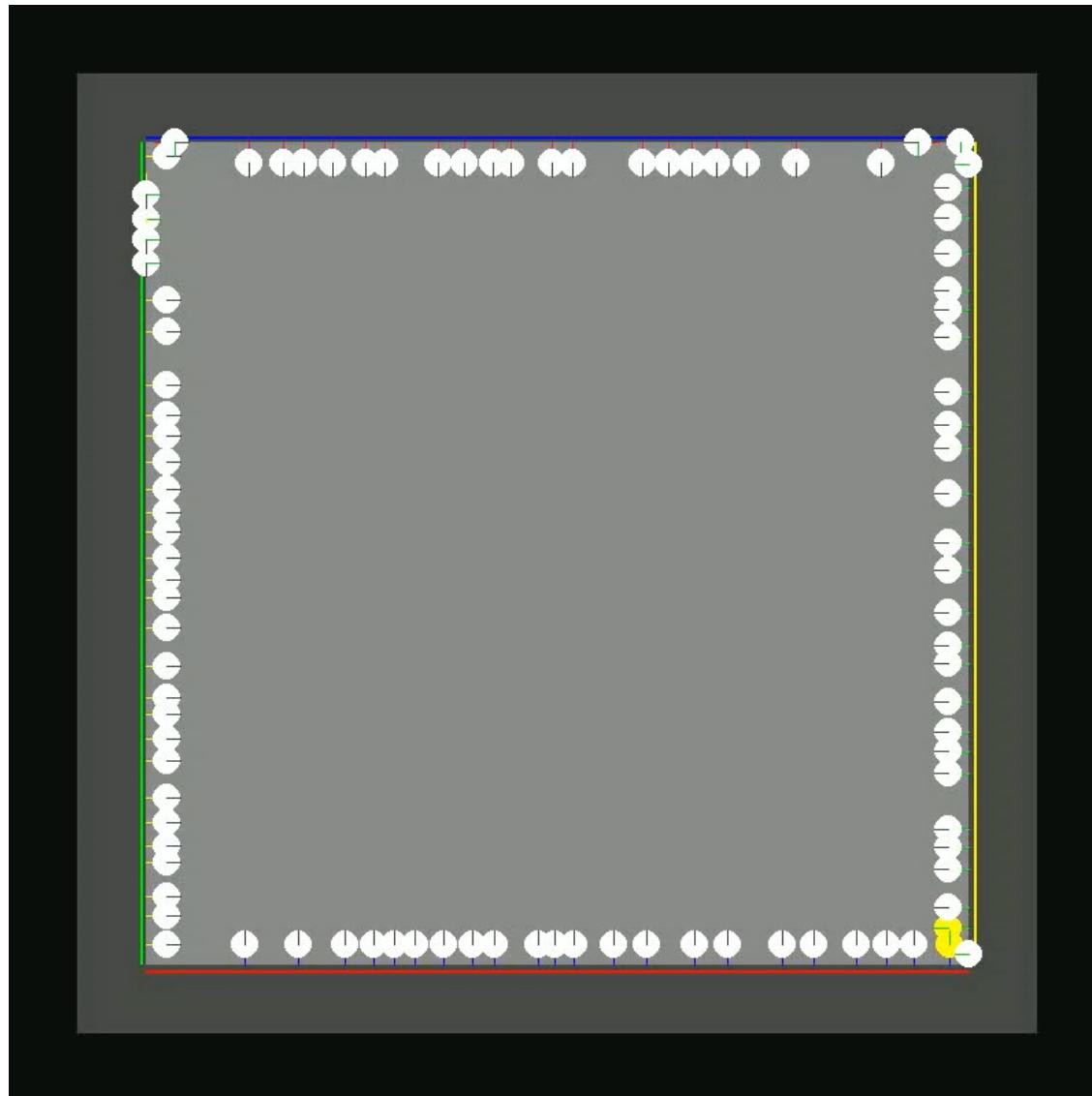
- Generally, in reinforcement learning we aim to maximize the cumulative rewards
- However, shaping the rewards in order to induce the desired behavior of the agents is one of the problems of RL
- The evaluation of the behavior should be independent of the action reward formulation
- For goal-oriented tasks, the following metrics seem adequate:
  - **Accuracy:** The percentage of episodes that complete successfully
  - **Steps:** The average number of time steps to complete a successful task



# APPO with few neighbor observations



# APEX DDPG



# “Small” Multi-Agent Systems: Game Theory

Normal-form games:

- Finite set of agents  $N = \{1, \dots, n\}$
- For each agent  $i$ :
  - Finite Set of actions  $A_i$
  - Reward function:  $u_i: A \rightarrow \mathbb{R}$ , where  $A = A_1 \times \dots \times A_n$ , (joint action space)

# Example Prisoner Dilemma

- Two prisoners are interrogated in separate rooms
- Each prisoner can Cooperate (C) or Defect (D) (stay silent)
- Reward Matrix:

		Prisoner B	
		Cooperate	Defect
Prisoner A	Cooperate	(-1,-1)	(-5,0)
	Defect	(0,-5)	(-3,-3)

# Solving a Multi-Agent Game

How do we solve such a game?

- If the game has common rewards, then solving the game is like solving a MDP  
→ Find a policy that maximizes the returns for all states  $s$

If the agent rewards differ, what should a policy optimize?

There are different solutions concepts:

- Minimax solution
- Nash equilibrium
- Pareto-Optimality
- No-Regret
- ....

# Nash Equilibrium

"No agent can improve their reward by changing their own strategy"  
(Every agents plays the best response to other agents)

		Prisoner B	
		Cooperate	Defect
Prisoner A	Cooperate	(-1,-1)	(-5,0)
	Defect	(0,-5)	(-3,-3)

The only Nash Equilibrium is (Defect / Defect)

# Repeated Game

Learning is to improve performance via experience

A normal-form game is a single interaction → no experience

Experience comes from **repeated** interactions

Repeated game:

- Repeat the same normal-form game for time steps  $t = 0, 1, 2, \dots$
- Learn by modifying the policy based on the history of the actions and rewards

## **Independent Learning: Centralized training with decentralized execution (CTDE)**

- A single agent RL algorithm such as Q-Learning, Policy-Gradient or Actor Critic Method is used.
- A centralized algorithm is trained (and shared) for all agents.
- Each agent executes the algorithm and receives its own observations and rewards.

## **Independent Learning: Centralized training with decentralized execution (CTDE)**

Advantages:

- Experience for all agents go into the training data

Disadvantages:

- Algorithms must generalise to the behaviour for all agents.
- (I.e. for the crossing, there can not be a different policy learned for the agents crossing from left to right and the ones crossing from right to left)

# Independent Learning: Decentralized Learning

- In a decentralized setting, each agent learns its own model
- For example using Actor-Critic this is called *Independent Actor Critic* (IAC)
- A combination of both cases uses a decentralized policy function but a centralized critic, this is called *Independent Actor with Centralized Critic* (IACC)

# Joint Action Learning

A single algorithm learns the joint actions for all agents.

Advantages:

- No information exchange necessary

Drawbacks:

- High dimensional action space
- Only a single reward → Credit Assignment Problem
- Can be hard to train when the number of agents increases

# Star Craft Example

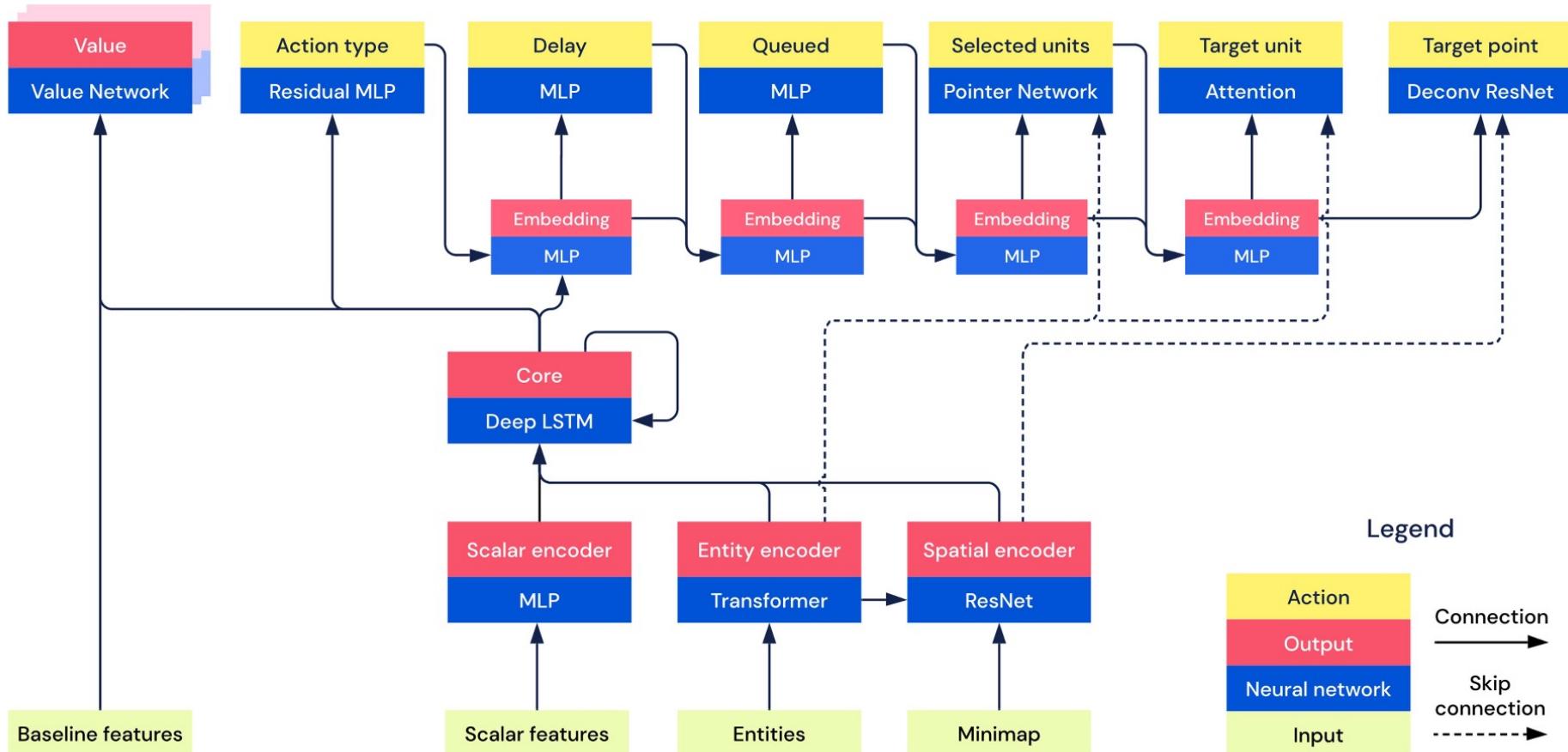
Policy represented by a neural network

$$\pi_\theta(a_t | s_t, z)$$

where  $s_t$  is the vector of all observations and actions so far, and  $z$  a statistic learned from human data

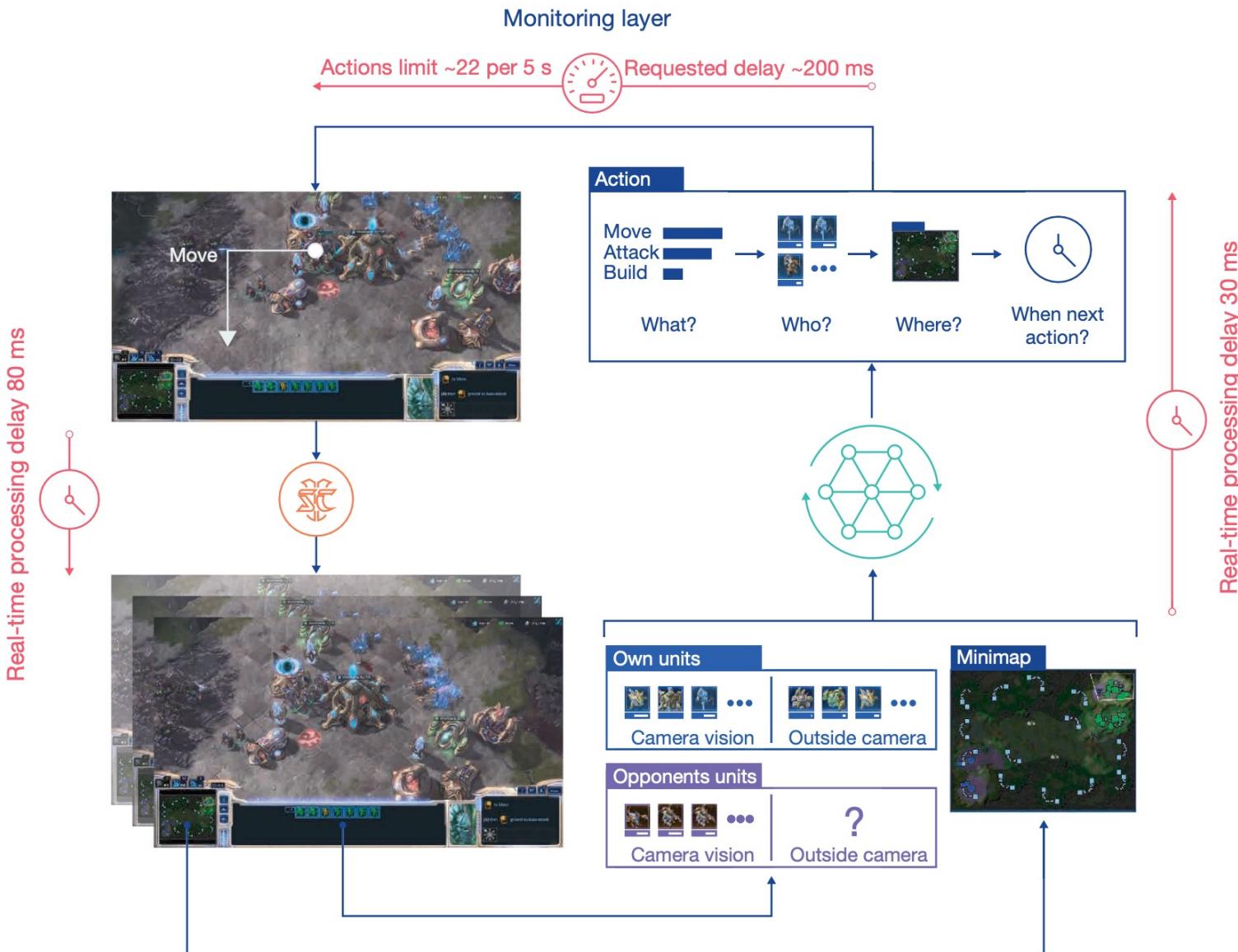
- Initially trained with supervised learning on human data
- Then RL learning with Actor-Critic approach
- Introduces League training:
  - Train against different types of opponents sampled from a collection of policies

# Star Craft Architecture



# Star Craft

a



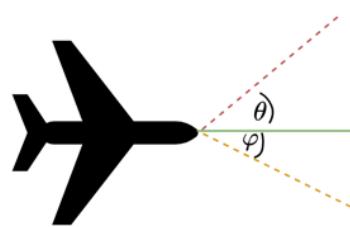
# Examples: MAM: Machine-Managed Air Mobility in Swiss Upper Airspace

Innosuisse Project between HSLU (ABIZ), SkySoft and SkyGuide

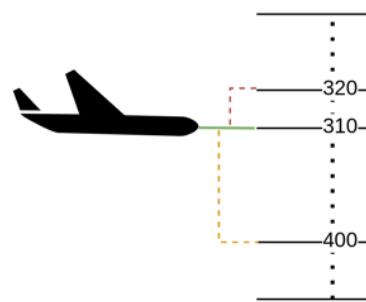
- Aircrafts want to fly as close to their flight plan as possible without getting too close to other aircrafts.
- RL agents provide controlling



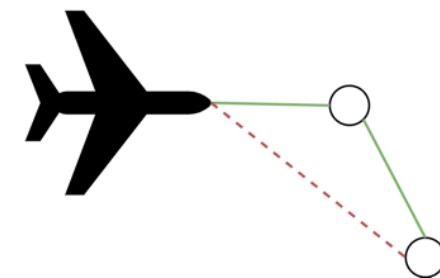
# ATM Simulation – Action Spaces



Change  
Heading (CH)



Change Flight  
Level (CFL)



Skip Route  
Waypoint (SW)

Joint = CH + CFL + SW

# Example: Joint Action Space



# Learning in MARL

What is learned in multi-agent learning:

- Best own action
- But potentially also how other agents behave

Example:

Autonomous cars

Crossing -> If all agents use the same algorithms, the solution is different from when you would have a random agent (or another policy for agents) that distur

# Communication

Communication can be viewed for different objectives in MARL:

- Exchange of information about the environment: Each individual agent may not view the exact state, but only its own observation which is a partial view of the global environment
- Agents might also exchange information about the action that they are considering to make, so that other agents can react on them
- Communication can most generally be modelled by **exchanging messages** between agents.

# **Graph Neural Networks for communication**

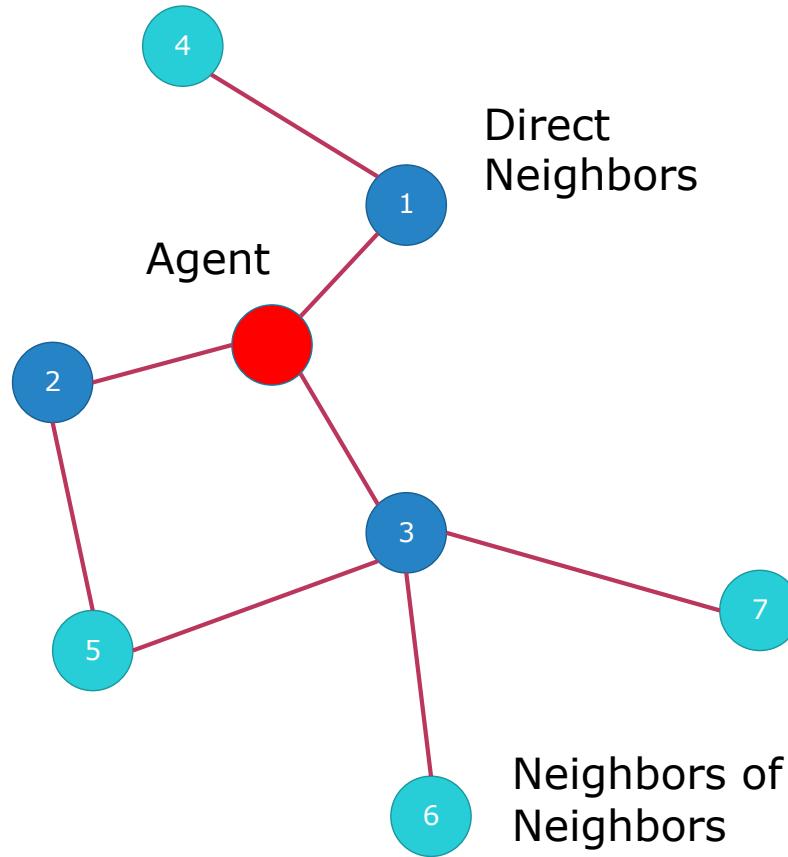
Allowing all agents do exchange messages might lead to a very complex environment.

We could also model the agents as graph, where messages can only be retrieved (or sent) to the direct neighbors.

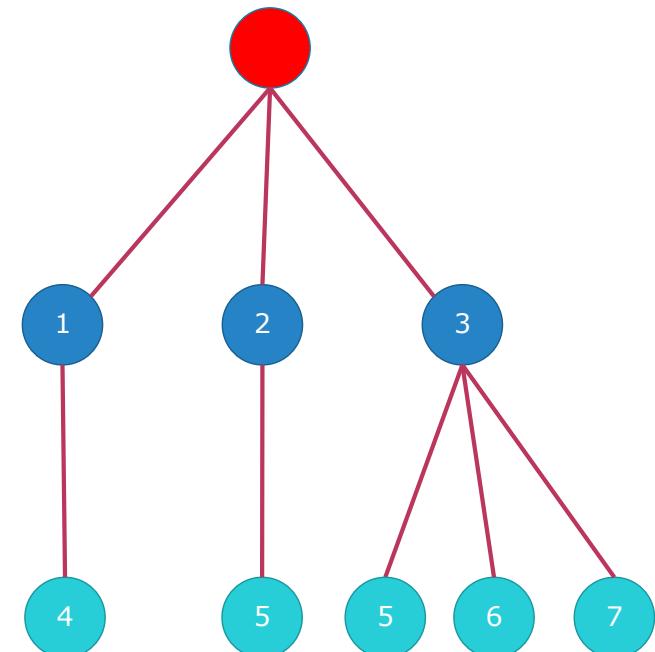
This could be modelled by graph neural networks

# Graph Neural Networks

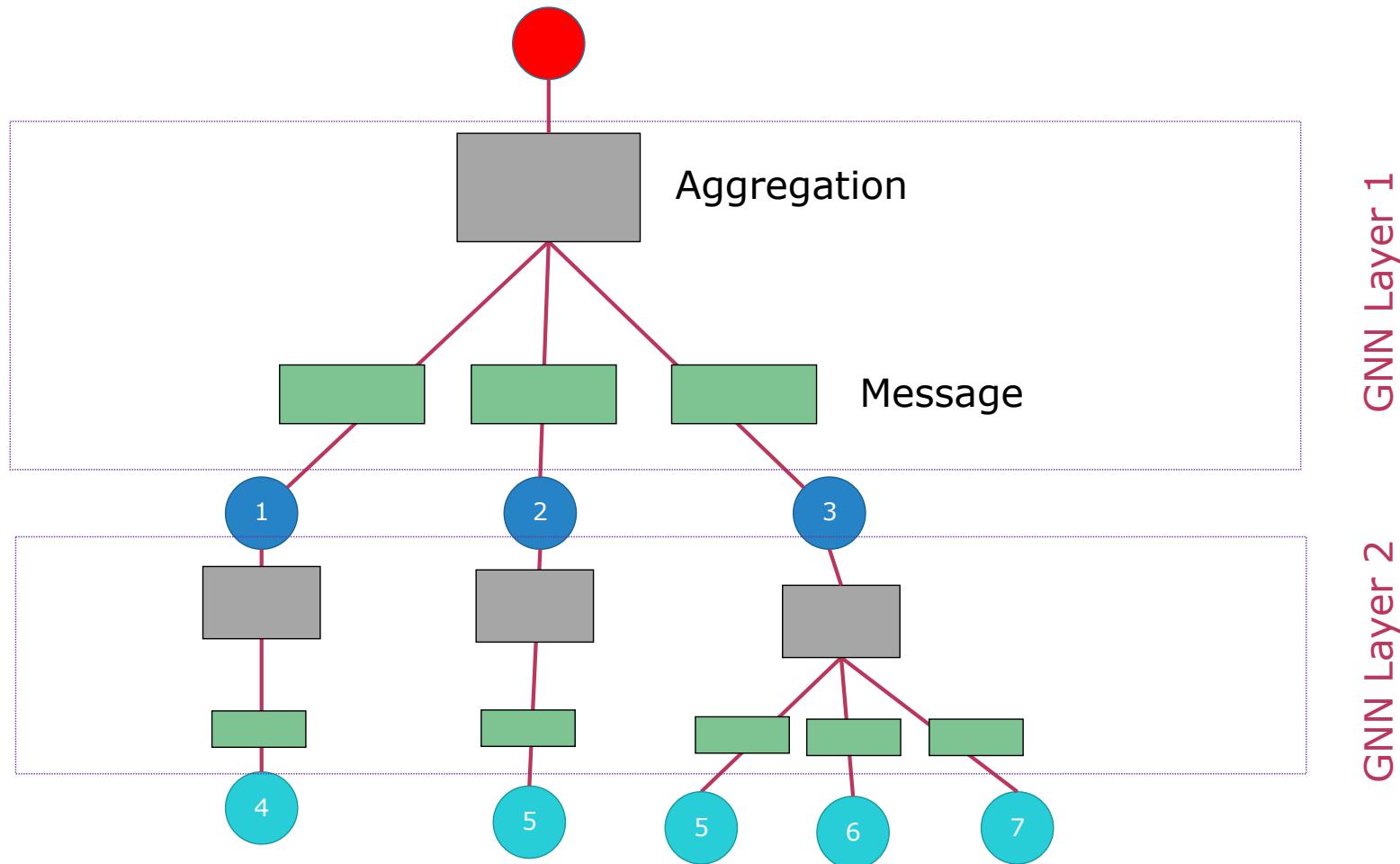
## Graph Representation



## Computational Graph



# GNN Framework



# GNN Framework

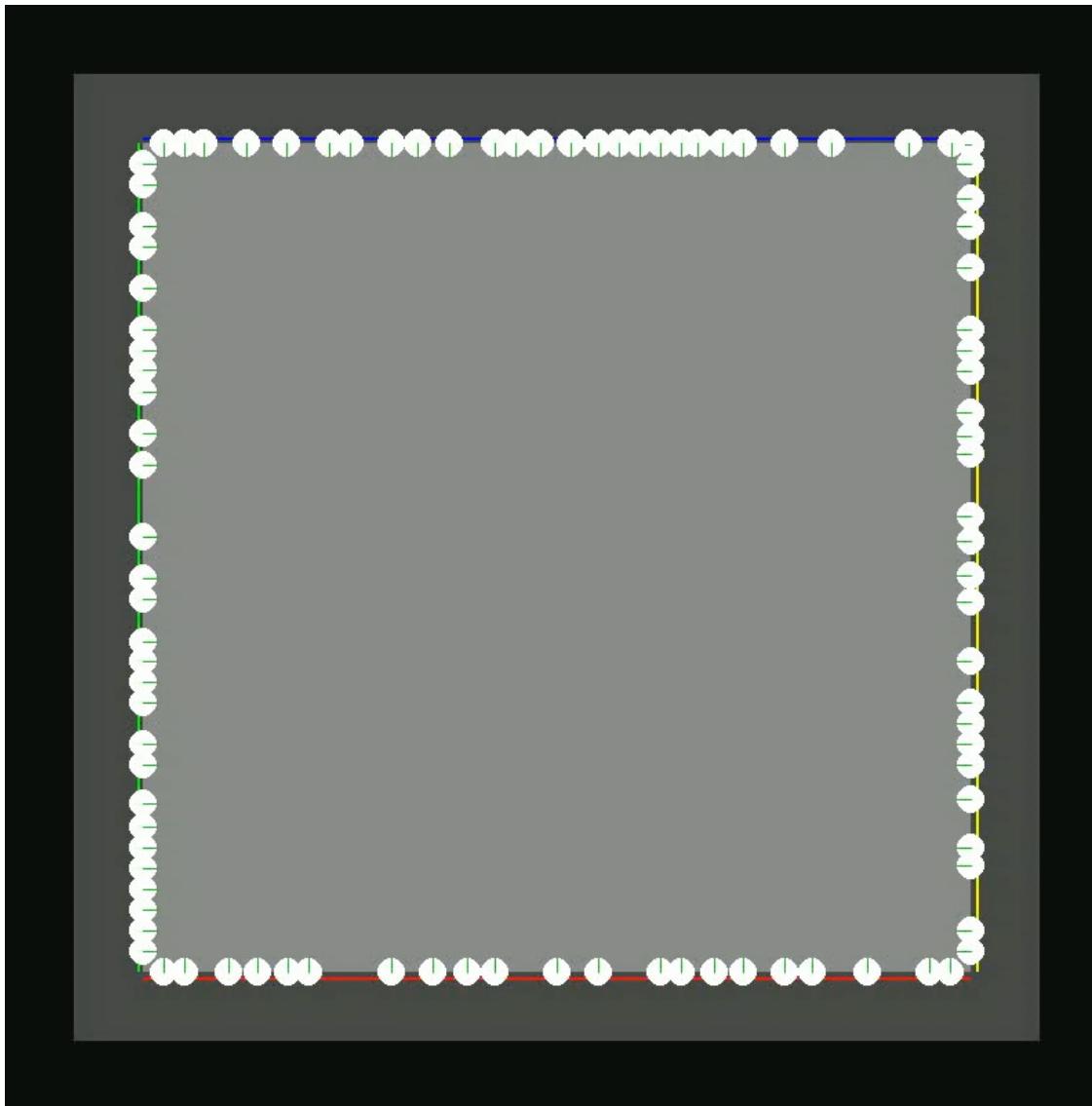
Messages:

- Use Neural Networks to calculate messages from node properties (and possibly edge properties)

Aggregation:

- Classical: Use Mean or Max from Messages
- Use Nodes from Recurrent Networks such as Simple RNNs, LSTMs etc.
- Use Multiheaded Attention Networks

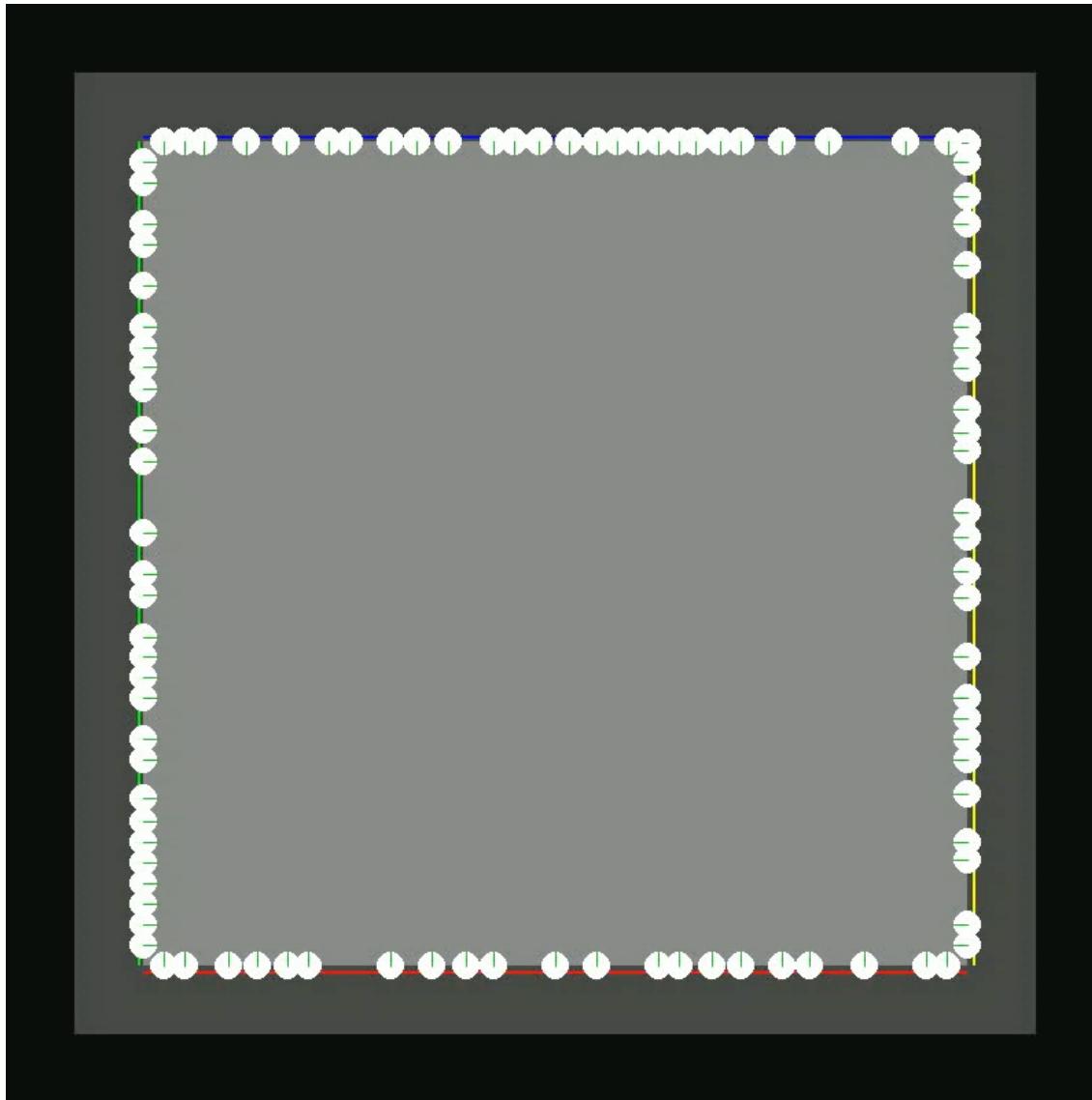
# PPO using dense model



0.96 Accuracy

52.2 Steps

# PPO using CNN with Attention Model



0.99 Accuracy

46.6 Steps