

# Reinforcement Learning Introduction

Christian B. Mendl

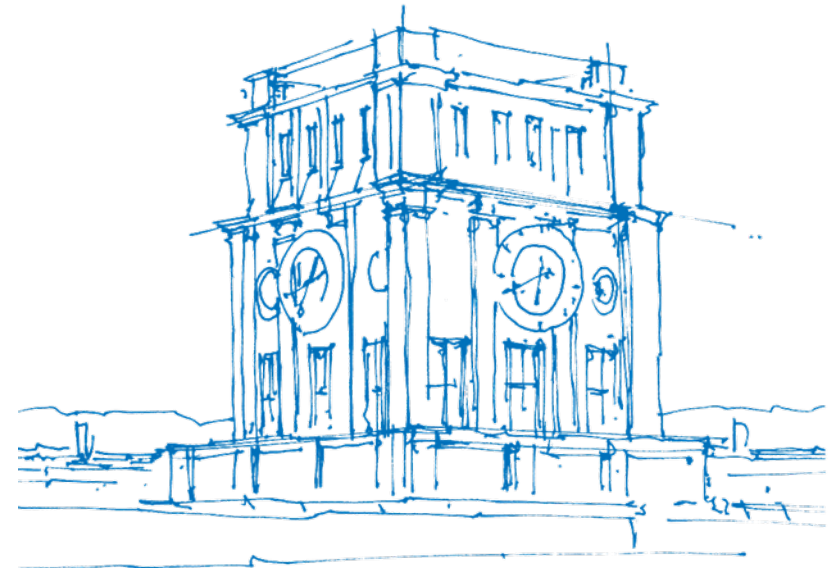
Technische Universität München

Department of Informatics 5

Chair of Scientific Computing in Computer Science (SCCS)

March 10, 2020

[https://gitlab.lrz.de/tum-i05/chair-retreat/reinforcement\\_learning](https://gitlab.lrz.de/tum-i05/chair-retreat/reinforcement_learning)



*TUM Uhrenturm*

# Outline

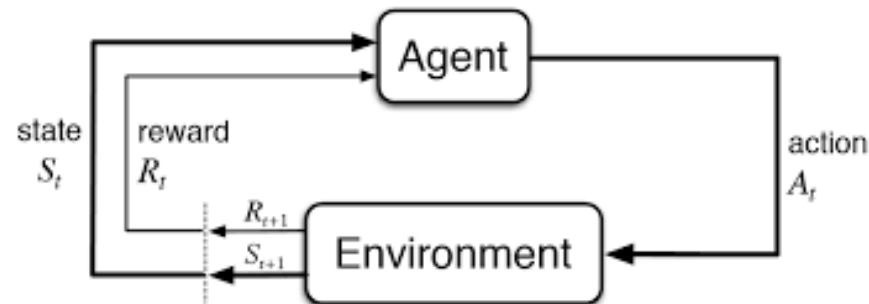
- Introduction and motivation
- Theoretical framework
  - Markov decision processes (MDPs)
  - Iterative “dynamic programming” algorithms
  - Deep reinforcement learning
  - Policy gradients
- Hands-on example: simplified Pac-Man

# Introduction and motivation

# Motivation

Goal-oriented behavior in a complex, non-deterministic environment

Example autonomous driving: steer vehicle safely from A to B, “environment” consists of other vehicles, bikes, pedestrians, street signs, ...

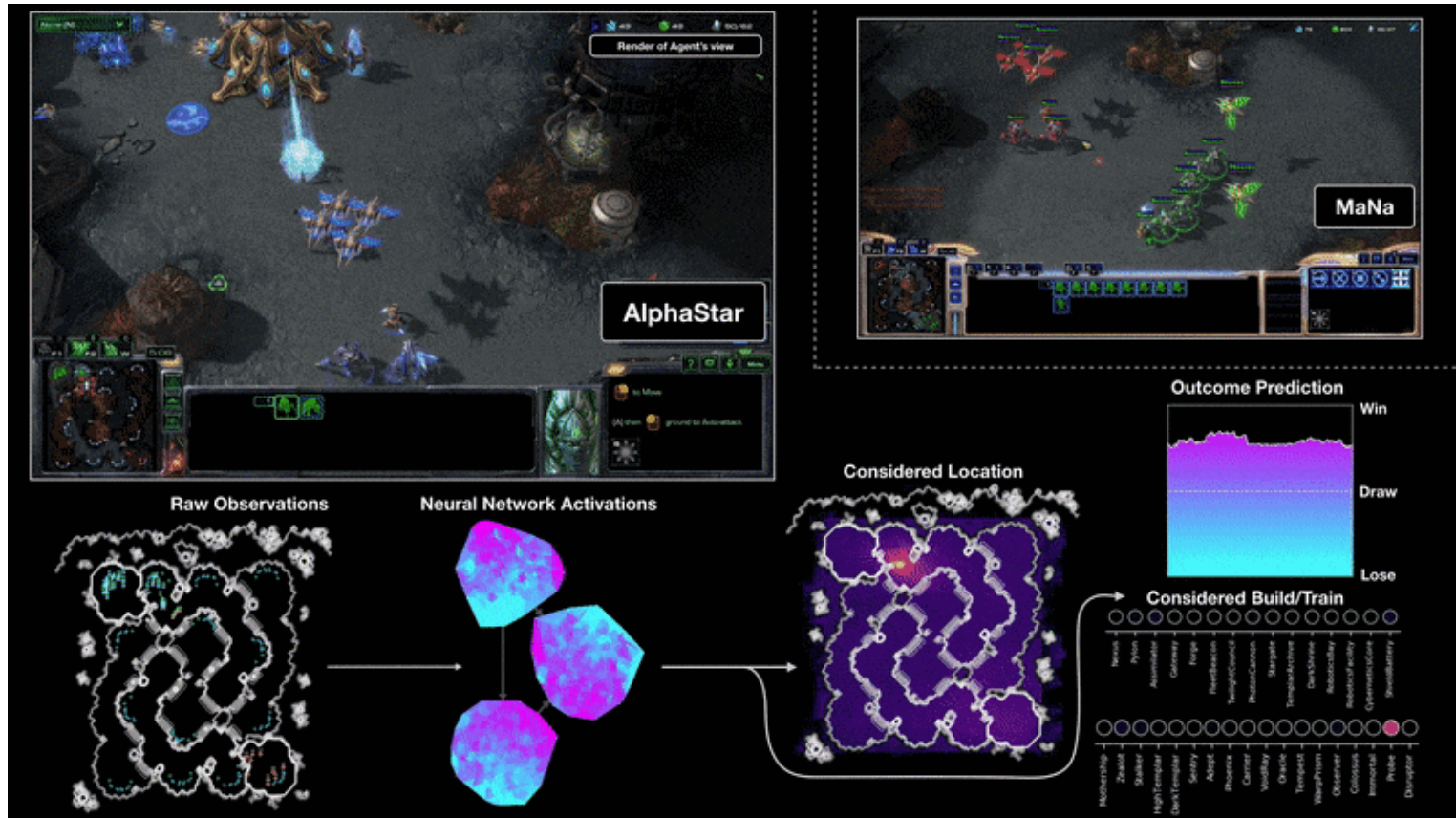


# Deepmind's AlphaGo



# Deepmind's Alphastar

Reinforcement learning to train an agent to play StarCraft II

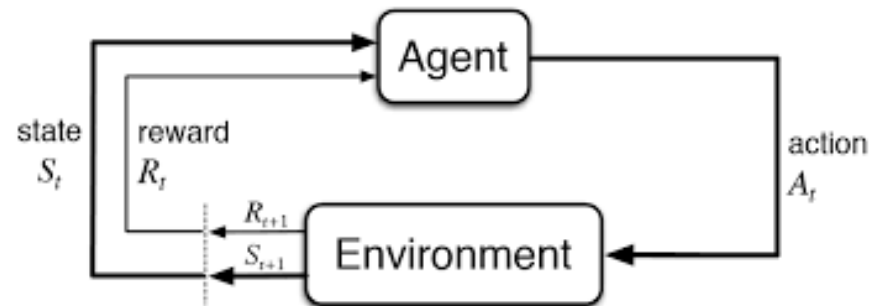


<https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii>

# Theoretical framework

# MDPs: agent and environment

Formal description: an agent interacts with an environment



**State** of agent and environment at time points  $t = 0, 1, 2, \dots$  denoted  $s_t \in S$  (finite-dimensional space)

Agent chooses an **action**  $a_t \in A \rightsquigarrow$  system transitions to next state  $s_{t+1} \in S$ , transition probability

$$\mathbb{P}(s_{t+1} | s_t, a_t).$$

*Markov property*: probability does not explicitly depend on previous time points



# MDPs: reward and return

At each time point the agent receives a **reward** (can be positive, zero or negative):

$$r_t = R(s_t)$$

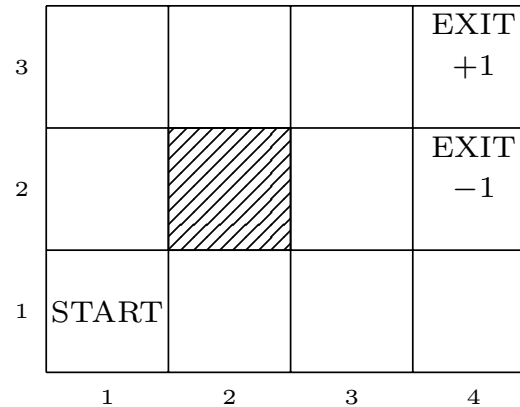
Goal: maximize **return**: cumulative discounted reward

$$\sum_{t=0}^{\infty} \gamma^t R(s_t), \quad 0 < \gamma \leq 1$$

with  $\gamma^t$  the discount factor

# Model example: grid world

$4 \times 3$  grid world with one player (hatched field (2,2) is inaccessible):

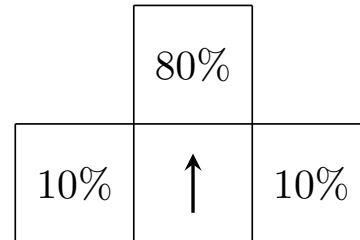


$s_t$ : current actor (player) position

Possible actions:  $\uparrow, \downarrow, \leftarrow, \rightarrow$ : “try to move in corresponding direction by one field”

# Model example: grid world, continued

Transition probability:



Formally:

$$\mathbb{P} \left( \mathbf{s}_{t+1} = \mathbf{s}_t + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \mid \mathbf{s}_t, \mathbf{a}_t = \uparrow \right) = 0.8,$$

$$\mathbb{P} \left( \mathbf{s}_{t+1} = \mathbf{s}_t + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \mid \mathbf{s}_t, \mathbf{a}_t = \uparrow \right) = 0.1,$$

$$\mathbb{P} \left( \mathbf{s}_{t+1} = \mathbf{s}_t + \begin{pmatrix} -1 \\ 0 \end{pmatrix} \mid \mathbf{s}_t, \mathbf{a}_t = \uparrow \right) = 0.1$$

Analogously for  $\downarrow, \leftarrow, \rightarrow$

Reward: +1 for exit field (4,3), -1 for exit field (4,2), -0.04 for any other field (agent wants to reach exit as fast as possible)

# MDPs: policy function and utility

How do we specify possible solutions?

Actions of agent specified by **policy**  $\pi : S \rightarrow A$ :

$$a_t = \pi(s_t)$$

Due to Markov property: only dependency on  $s_t$  required (instead of  $s_0, s_1, \dots, s_t$ )

# MDPs: policy function and utility

How do we specify possible solutions?

Actions of agent specified by **policy**  $\pi : S \rightarrow A$ :

$$a_t = \pi(s_t)$$

Due to Markov property: only dependency on  $s_t$  required (instead of  $s_0, s_1, \dots, s_t$ )

“Quality” of  $\pi$  quantified by *expected return*, denoted **utility** or **value function**:

$$U^\pi(s_0) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t) \middle| \pi \right],$$

( $s_1, s_2, \dots$  now regarded as random variables)

Notation  $\mathbb{E}[\dots | \pi]$ : use policy  $\pi$  for choosing actions  $a_t$

# Optimal policy and utility

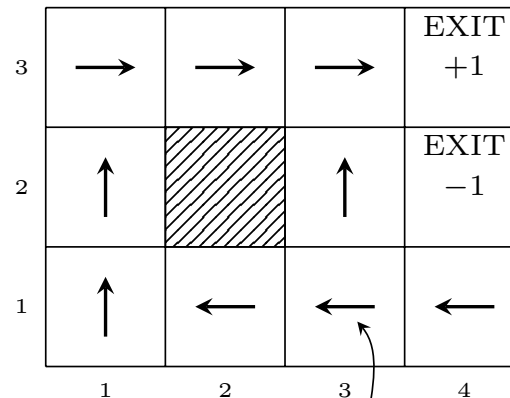
$$\pi^* = \operatorname{argmax}_{\pi} U^{\pi}(s_0)$$

$\pi^*$  does not depend on  $s_0$ !

Justification: self-similar form:

$$\mathbb{E} \left[ \sum_{t'=t}^{\infty} \gamma^{t'} R(s_{t'}) \middle| \pi, s_t = \tilde{s}_0 \right] = \gamma^t \mathbb{E} \left[ \sum_{\Delta t=0}^{\infty} \gamma^{\Delta t} R(s_{t+\Delta t}) \middle| \pi, s_t = \tilde{s}_0 \right] = \gamma^t U^{\pi}(\tilde{s}_0).$$

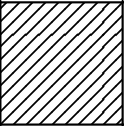
Example (for  $\gamma = 1$ ):



take detour, instead of  
risking to end up in (4, 2)

# Optimal policy and utility, continued

Corresponding value function for optimal policy:  $U = U^{\pi^*}$  (omit superscript  $\pi^*$ )

3	0.812	0.868	0.918	EXIT +1
2	0.762		0.660	EXIT -1
1	0.705	0.655	0.611	0.388
	1	2	3	4

Conversely, can obtain  $\pi^*$  from  $U$ :

$$\pi^*(s) = \operatorname{argmax}_{a \in A} \sum_{s'} \mathbb{P}(s'|s, a) U(s')$$

“Choose action which maximizes the expected return.”

# Bellman equation for $U$ , value iteration algorithm

Intuition: expected utility of a state  $s$  is equal to the instantaneous reward and the expected utility of the next state, assuming optimal behavior of the agent.

↪ *Bellman equation* for the value function:

$$U(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} \mathbb{P}(s'|s, a) U(s') \quad \forall s \in S$$

(uniquely solvable for  $\gamma < 1$ )

Corresponding **value iteration algorithm**:

$$U_{i+1}(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} \mathbb{P}(s'|s, a) U_i(s') \quad \forall s \in S, \quad i = 0, 1, \dots$$



# Policy iteration algorithm

Idea: “best” policy according to  $U_i$ , i.e.,

$$\pi_i(s) = \operatorname{argmax}_{a \in A} \sum_{s'} \mathbb{P}(s'|s, a) U_i(s'),$$

might already agree with optimal policy  $\pi^*$ , even if  $U_i$  still deviates from  $U^{\pi^*}$

$\rightsquigarrow$  *policy iteration algorithm*:

**Require:** initial policy  $\pi_0$  (e.g., randomly selected)

**for**  $i \leftarrow 0, 1, 2, \dots$  **do**

(a) policy evaluation: compute

$$U_i(s) = U^{\pi_i}(s) \quad \forall s \in S$$

(expected utility, if agent follows policy  $\pi_i$ ), by solving linear system

$$U_i(s) = R(s) + \gamma \sum_{s'} \mathbb{P}(s'|s, \pi_i(s)) U_i(s')$$

(b) policy improvement: use  $U_i$  to derive a new policy  $\pi_{i+1}$ :

$$\pi_{i+1}(s) = \operatorname{argmax}_{a \in A} \sum_{s'} \mathbb{P}(s'|s, a) U_i(s') \quad \forall s \in S$$

Stop if optimal policy has been found, i.e., if  $\pi_{i+1} = \pi_i$  or (equivalently)  $U_{i+1} = U_i$  (in this case  $U_i$  is unique fixed point of value iteration)

# Q-value function

So far: value function  $U^\pi(s)$  for state  $s$

**Q-value function:** evaluate state-action tuple  $(s, a)$ :

$$Q^\pi(s, a) := \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s, a_0 = a, \pi \right]$$

“expected utility, if agent in state  $s$  chooses action  $a$  and then follows policy  $\pi$ ”

Optimal Q-value function for optimal policy:

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

Advantage: optimal value function  $U$  and optimal policy  $\pi^*$  can be directly obtained from  $Q^*$ :

$$U(s) = \max_a Q^*(s, a),$$
$$\pi^*(s) = \operatorname{argmax}_a Q^*(s, a)$$

No knowledge of transition probability  $\mathbb{P}(s'|s, a)$  required, “**model free**”

# Bellman equation for Q-value function, iteration

Similar to Bellman equation for  $U$ :

$$\begin{aligned} Q^*(s, a) &= R(s) + \gamma \sum_{s' \in S} \mathbb{P}(s'|s, a) \max_{a' \in A} Q^*(s', a') \\ &= \mathbb{E}_{s'} \left[ R(s) + \gamma \max_{a' \in A} Q^*(s', a') \mid s, a \right], \end{aligned}$$

with  $\mathbb{E}_{s'}[\dots | s, a]$  the expectation value with respect to random variable  $s' \sim \mathbb{P}(\cdot | s, a)$

Corresponding Q-value iteration:

$$Q_{i+1}(s, a) = R(s) + \gamma \sum_{s' \in S} \mathbb{P}(s'|s, a) \max_{a' \in A} Q_i(s', a') \quad \forall s \in S, a \in A, \quad i = 0, 1, \dots$$

# Classical Q-learning algorithm

So far: iteration  $Q_i \rightarrow Q_{i+1}$ , but instead of (unknown) transition probability  $\mathbb{P}(s'|s, a)$ , use observed transition  $s_t \rightarrow s_{t+1}$  based on simulation of environment

Update Q-value function at  $(s, a) = (s_t, a_t)$  (otherwise unchanged):

$$Q_{i+1}(s_t, a_t) = (1 - \eta_i)Q_i(s_t, a_t) + \eta_i \left( R(s_t) + \gamma \max_{a'} Q_i(s_{t+1}, a') \right)$$

with learning rate  $\eta_i$  ( $0 < \eta_i < 1$ )

Watkins and Dayan prove convergence  $Q_i \xrightarrow{i \rightarrow \infty} Q^*$  under mild assumptions, in the limit of infinite number of episodes, such that every tuple  $(s, a)$  appears infinitely often.

C. J. Watkins, P. Dayan (1992)

# Deep reinforcement learning overview

Issues of iterative “dynamic programming” (DP) algorithms considered so far:

1. Requires storing entries  $U(s)$ ,  $\pi(s)$  or  $Q(s, a)$  for all possible state  $s \in S$  (and actions  $a \in A$ ), only feasible for relatively small state space

Classical board games:

$$|S| \approx b^d,$$

with

$b$ : mean number of allowed moves for a given board position

$d$ : depth, i.e., typical total number of moves per game

– chess:  $b \approx 35$ ,  $d \approx 80$

– Go:  $b \approx 250$ ,  $d \approx 150$

$\rightsquigarrow$  cannot enumerate all possible board positions

2. Requires knowledge of transition probability  $\mathbb{P}(s'|s, a)$ , but in practice often unknown or hard to estimate (e.g., autonomous driving)

Instead: conceptual goal of reinforcement learning: agent should choose sensible actions in (initially unknown) environment

Ansatz **deep reinforcement learning**: approximate  $U$ ,  $\pi$  or  $Q$  by neuronal network; “deep” refers to network depth (number of layers)

# Deep reinforcement learning overview, continued

Instead of working with explicit transition probabilities, repeat many simulations of the systems; agent “learns” during the simulations

Trade-off between “**exploration**” and “**exploitation**”: try action that hasn’t been used or rarely used (even if not optimal) to gain experience, or choose action that is likely to maximize the benefit?

Concepts:

- **on-policy training**: the policy to be optimized is also used to select actions during simulations, i.e., to generate training samples
- **off-policy training**: the policy to be optimized is usually different from the policy used to select actions during the simulations

Opportunities for “exploration” in both cases:

- on-policy: use probability distribution, transition to deterministic action in the course of training (i.e., a single action has probability 1)
- off-policy: exploration using the policy for choosing actions during the simulation; often used:  **$\epsilon$ -greedy** strategy (random action with probability  $\epsilon$ )

# Deep Q-learning (now mostly historical relevance)

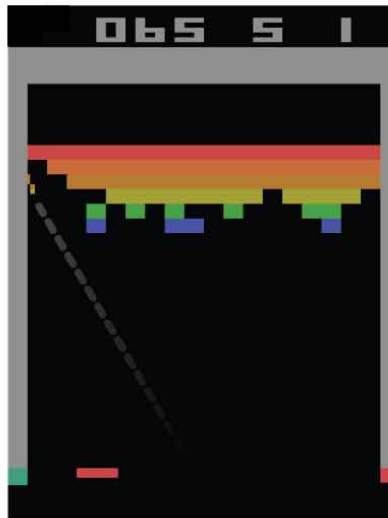
Goal: algorithms to (approximately) solve Bellmann equation

$$Q^*(s, a) = \mathbb{E}_{s'} \left[ R(s) + \gamma \max_{a' \in A} Q^*(s', a') \mid s, a \right]$$

Ansatz:

$$Q^*(s, a) \approx Q(s, a; \theta),$$

with  $Q(s, a; \theta)$  a neural network (Q network) with parameters  $\theta$  (weights and bias vectors); iterative optimization:  $\theta_{i-1} \rightarrow \theta_i, i = 1, 2, \dots$



V. Mnih, K. Kavukcuoglu, D. Silver, . . . , D. Hassabis. *Human-level control through deep reinforcement learning*. Nature (2015)

# Policy gradient methods

Idea: compute optimal policy  $\pi^* : S \rightarrow A$  *directly*, i.e., in general without computing value function  $U(s)$  or Q-value function  $Q(s, a)$  (model free, works for high-dimensional state spaces)

Ansatz for policy function:  $\pi_\theta$ , with to-be optimized parameters  $\theta \in \mathbb{R}^m$ , e.g., weights and bias vectors of an ANN

Use (as before) the expected utility to evaluate a policy function, with a default initial state  $\hat{s}_0$ :

$$J(\theta) := U^{\pi_\theta}(\hat{s}_0) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t) \middle| s_0 = \hat{s}_0, \pi_\theta \right]$$

Goal:  $\theta^* = \operatorname{argmax}_\theta J(\theta)$



# Policy gradient theorem

In the following: policy function specifies action probabilities:  $\pi(a|s)$  (instead of  $a = \pi(s)$ )

↪ **policy gradient theorem:**

$$\nabla J(\theta) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \left( \sum_{t'=t}^{\infty} \gamma^{t'} R(s_{t'}) \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \middle| \pi_{\theta} \right]$$

Interpretation: to optimize  $J(\theta)$  via gradient descent: if remaining “cumulative discounted reward”  $\sum_{t'=t}^{\infty} \gamma^{t'} R(s_{t'})$  starting from  $t$  is positive, then increase probability  $\pi(a_t | s_t)$  of chosen actions

# REINFORCE algorithm

Directly based on formula for  $\nabla J(\theta)$ : gradient descent with learning rate  $\eta$

- 1: Chose initial parameters  $\theta$
- 2: **for** episode  $\leftarrow 0, 1, \dots$  **do**
- 3:     Run simulation using policy  $\pi_\theta$ , obtain trajectory  $(s_0, a_0, r_0, \dots, s_T, a_T, r_T)$
- 4:     **for**  $t \leftarrow 0, 1, \dots, T$  **do**
- 5:          $G_t \leftarrow \sum_{t'=t}^T \gamma^{t'-t} r_{t'}$
- 6:          $\theta \leftarrow \theta + \eta \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t | s_t)$  ▷ gradient step

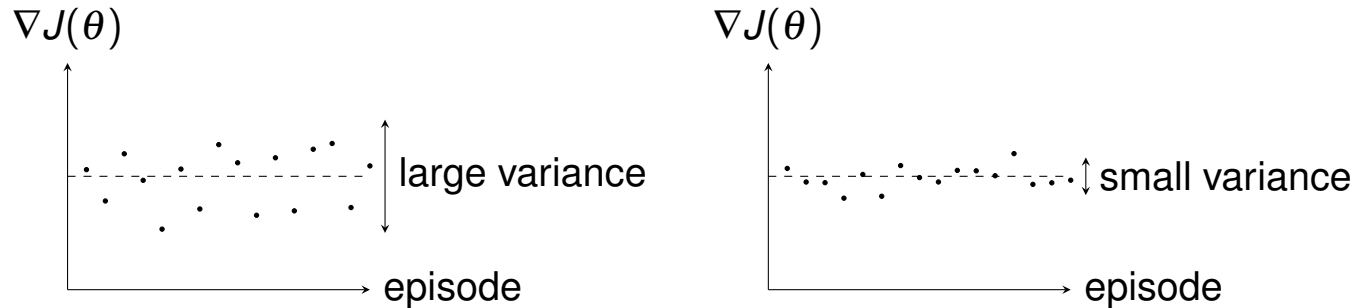
Remarks:

- Variants of the algorithm without factor  $\gamma^t$  in line 6
- Can apply gradient only “a posteriori”, i.e., after completing a game

R. J. Williams. *Simple statistical gradient-following algorithms for connectionist reinforcement learning* (1992)

# Actor critic method

Motivation: reduce variance when estimating  $\nabla J(\theta)$  via sampling



Idea: in derivation of the policy gradient theorem:

$$\nabla J(\theta) = \sum_{t=0}^{\infty} \sum_{s \in S} \mathbb{P}_{\theta}(\hat{s}_0 \rightarrow s \text{ in } t \text{ steps}) \gamma^t \sum_a Q^{\pi_{\theta}}(s, a) \nabla_{\theta} \pi_{\theta}(a|s) :$$

replace  $\sum_a Q^{\pi_{\theta}}(s, a) \nabla_{\theta} \pi_{\theta}(a|s)$  by  $\sum_a (Q^{\pi_{\theta}}(s, a) - b(s)) \nabla_{\theta} \pi_{\theta}(a|s)$  using an arbitrary “baseline”-function  $b : S \rightarrow \mathbb{R}$ : (only needs to be independent of parameters  $\theta$ )

Overall value remains unchanged, since

$$\sum_a b(s) \nabla_{\theta} \pi_{\theta}(a|s) = b(s) \nabla_{\theta} \underbrace{\sum_a \pi_{\theta}(a|s)}_{=1} = 0$$

# Actor critic method, continued

Leads to generalized formula for  $\nabla J(\theta)$ :

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t \left( \left( \sum_{t'=t}^{\infty} \gamma^{t'-t} R(s_{t'}) \right) - b(s_t) \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \middle| \pi_{\theta} \right] \\ &= \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t \left( Q^{\pi_{\theta}}(s_t, a_t) - b(s_t) \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \middle| \pi_{\theta} \right]\end{aligned}$$

$\sum_{t'=t}^{\infty} \gamma^{t'-t} R(s_{t'}) - b(s_t)$ : deviation of observed “cumulative reward” from baseline  $\rightsquigarrow$  within gradient step: increase or decrease probability of chosen action depending on whether *deviation* positive or negative

Typical choice of  $b(s)$ : value function  $U_{\phi}(s)$  with parameters  $\phi$  (independent of  $\theta$ )

Deviation denoted **advantage**:

$$A_t = Q^{\pi_{\theta}}(s_t, a_t) - U_{\phi}(s_t)$$

# Actor critic method, continued

↪ **actor critic method:**

- actor: policy function  $\pi_\theta$
- critic: evaluation of chosen actions based on  $A_t$

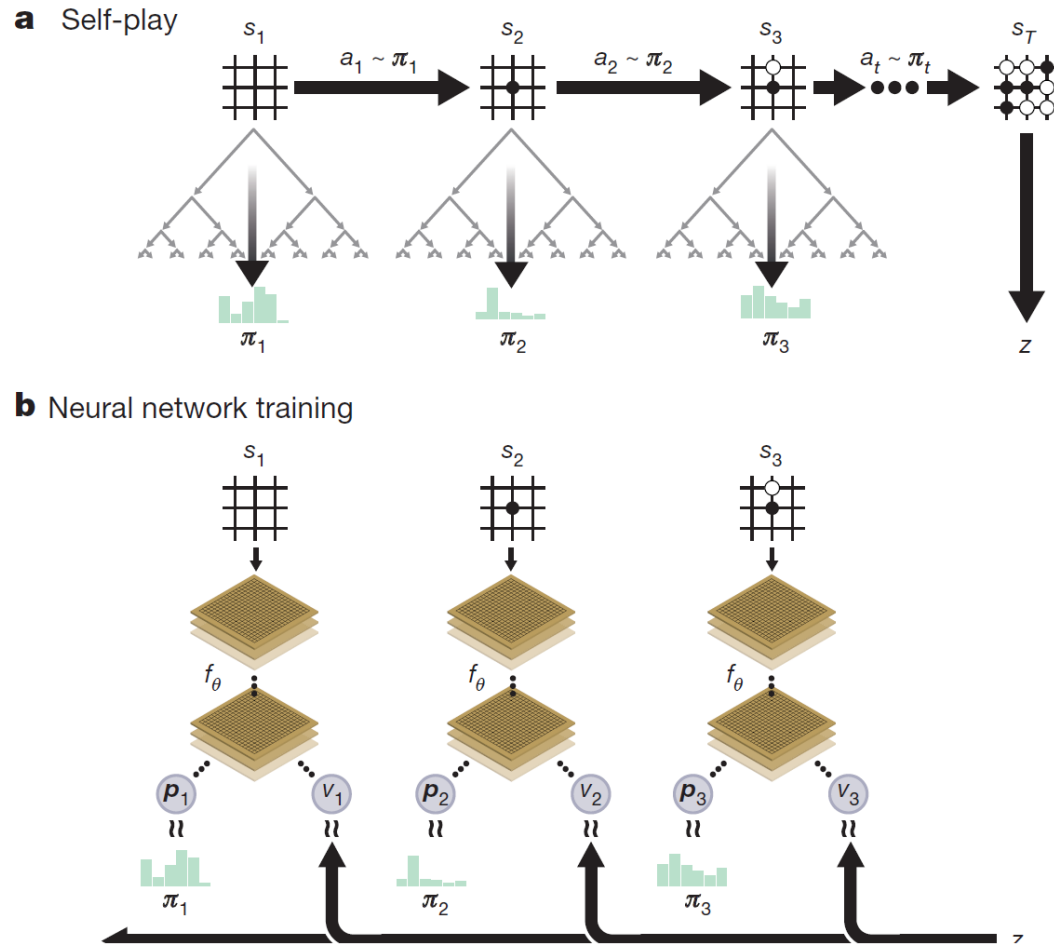
Parameters  $\theta$  and  $\phi$  are simultaneously optimized

Pseudo code (given learning rate  $\eta$  and  $\tilde{\eta}$ ):

- 1: Chose initial parameters  $\theta$  and  $\phi$
- 2: **for** episode  $\leftarrow 0, 1, \dots$  **do**
- 3:     Run simulation using policy  $\pi_\theta$ , obtain trajectory  $(s_0, a_0, r_0, \dots, s_T, a_T, r_T)$
- 4:     **for**  $t \leftarrow 0, 1, \dots, T$  **do**
- 5:          $G_t \leftarrow \sum_{t'=t}^T \gamma^{t'-t} r_{t'}$
- 6:          $A_t \leftarrow G_t - U_\phi(s_t)$
- 7:          $\theta \leftarrow \theta + \eta \gamma^t A_t \nabla_\theta \log \pi_\theta(a_t | s_t)$  ▷ gradient step for  $\theta$
- 8:          $\phi \leftarrow \phi + \tilde{\eta} A_t \nabla_\phi U_\phi(s_t)$  ▷ gradient step for  $\phi$

Remarks: typical asynchronous variants with multiple agents running in parallel cf. A3C  
 (“asynchronous advantage actor critic”)

# Example: AlphaGo Zero



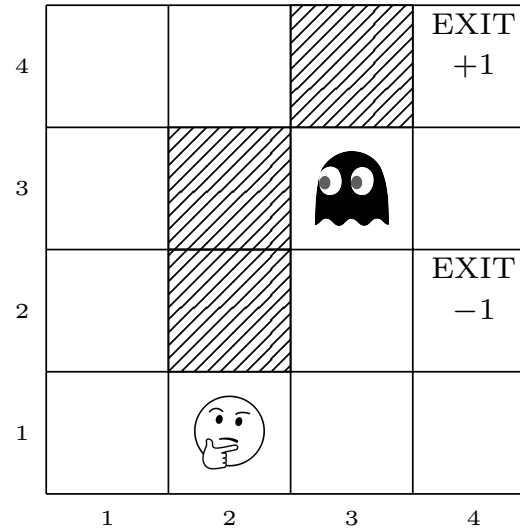
D. Silver, . . . , D. Hassabis. *Mastering the game of Go without human knowledge*. Nature (2017)

# Hands-on example: simplified Pac-Man

[https://gitlab.lrz.de/tum-i05/chair-retreat/reinforcement\\_learning](https://gitlab.lrz.de/tum-i05/chair-retreat/reinforcement_learning)

# Code structure






[https://gitlab.lrz.de/tum-i05/chair-retreat/reinforcement\\_learning](https://gitlab.lrz.de/tum-i05/chair-retreat/reinforcement_learning)



- Iterative “dynamic programming” algorithms in *mdp.py*
- Environment: plain maze or maze with ghost in *env.py*, geometry specified in text files (like *ghost\_maze.txt*)



# References

-  Mnih, V. et al. (2015). “Human-level control through deep reinforcement learning”. In: *Nature* 518, p. 529.
-  Mnih, V. et al. (2016). “Asynchronous methods for deep reinforcement learning”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Vol. 48. Proceedings of Machine Learning Research, pp. 1928–1937. URL: <https://arxiv.org/abs/1602.01783>.
-  Silver, D. et al. (2017). “Mastering the game of Go without human knowledge”. In: *Nature* 550, p. 354.
-  Watkins, C. J. and Dayan, P. (1992). “Technical note: Q-learning”. In: *Machine Learning* 8, pp. 279–292.
-  Williams, R. J. (1992). “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning* 8, pp. 229–256.