

Welcome to the lecture



# (Deep) Reinforcement Learning

Prof. Dr.-Ing. Thomas Nierhoff

# Introduction

# Introduction

About me

And you?

- Where are you from?
- Reason to study?
- What did you do before coming to OTH?
- Programming skills?



<https://de.depositphotos.com/stock-photos/question-mark-man.html>

Any questions related to OTH / studying?

# Introduction

## Structure

In general: Lecture with ungraded programming exercises

- I explain the concepts, you try to implement them
- Most classes within this lecture build on each other, so don't lose track
- The ungraded exercises will be helpful for the graded ones



Exam: Two graded exercise ("Modulararbeit")

- First exercise: Reinforcement learning programming exercise (1/3 weight)
- Second exercise: Tutorial on an advanced reinforcement learning concept (2/3 weight), contains code + short documentation + presentation
- Grades will be published after the exam period ("Notenbekanntgabe"), usually on a Thursday
- Grades can be discussed on the following day. Just come to my office between 9-12am without an appointment

Lecture slides and additional material in [moodle](#)

Office hours appointments via [Calendly](#)

# Introduction

## Requirements for this lecture

- Normal programming skills (Python + Numpy + Matplotlib)
- Normal understanding of linear algebra  
(two of the most complex equations within this course are shown below)

$$Q_{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \left( \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_{\pi}(s', a') \right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \cdot (G_t - \hat{V}_{\pi_{\boldsymbol{\theta}}}(S_t, \mathbf{w})) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t | S_t, \boldsymbol{\theta})$$

- Critical thinking – this is no course where you can sit down and relax 😞

## What you will get in return

- Deep understanding of THE most powerful AI technique

# Introduction

Homework (if you are unfamiliar with Python/Numpy/Matplotlib):

Program Connect Four in Python, use a Numpy array as the underlying data structure and visualize the board state after every move using Matplotlib.



<https://www.amazon.com/Hasbro-A5640-Connect-4-Game/dp/B00D8STBH4>

# Introduction

I

- am delighted to help you if you could not fully grasp a topic during the lecture (either approach me directly after the lecture or set up an appointment)
- will not help you for the graded exercise (debugging your code, explaining a formula in a paper/blog)
- am not your kindergarten teacher (dealing with accomodation, enrollment, user accounts, Linux, Python environments, ChatGPT solutions)  
→ ask your colleagues

You

- can come and leave the course whenever you want (but ideally only at the beginning/end of the regular lecture hours)
- do not need to do any of the ungraded programming exercises
- must register in PRIMUSS (mid of semester) if you want to enroll for the course this semester. In this case I have to give you a grade at the end of the semester

# Introduction

## Content

- Introduction / Basics of RL
- Basics of value-based RL / Methods for value-based RL (1/2)
- Exercise 1
- Methods for value-based RL (2/2)
- Exercise 2
- Reward shaping / Exploration and exploitation
- Gradient-based optimization / Function approximators / Neural networks
- Exercise 3
- Deep RL
- Exercise 4
- Methods for policy-based RL
- Exercise 5
- Model-based RL / Meta-RL
- Exercise 6?

# Introduction

## Resources

### Literature/videos

- [David Silver: Reinforcement learning](#)

Great video lecture, template for this course

- [Sutton, Barto: Reinforcement Learning: An Introduction. MIT Press \(2015\)](#)

The holy bible for reinforcement learning, little bit outdated but sufficient for this course

- [Sugiyama: Statistical Reinforcement Learning. CRC Press \(2015\)](#)

Good resource for specific topics on reinforcement learning

+ towardsdatascience blogs / youtube videos ☺

# Introduction

## Running Snail

- name of the OTH Amberg-Weiden racing team
- students build a racing car every year and compete against other teams
- if you are interested in joining → ask them, they don't bite
- if you need a supervisor for your project / thesis → ask me



# Introduction

## Course evaluation

- in the middle of the semester  
(or anytime via moodle)
- THE opportunity for you to provide  
good and bad feedback about  
the lecture(s)
- is not for nothing



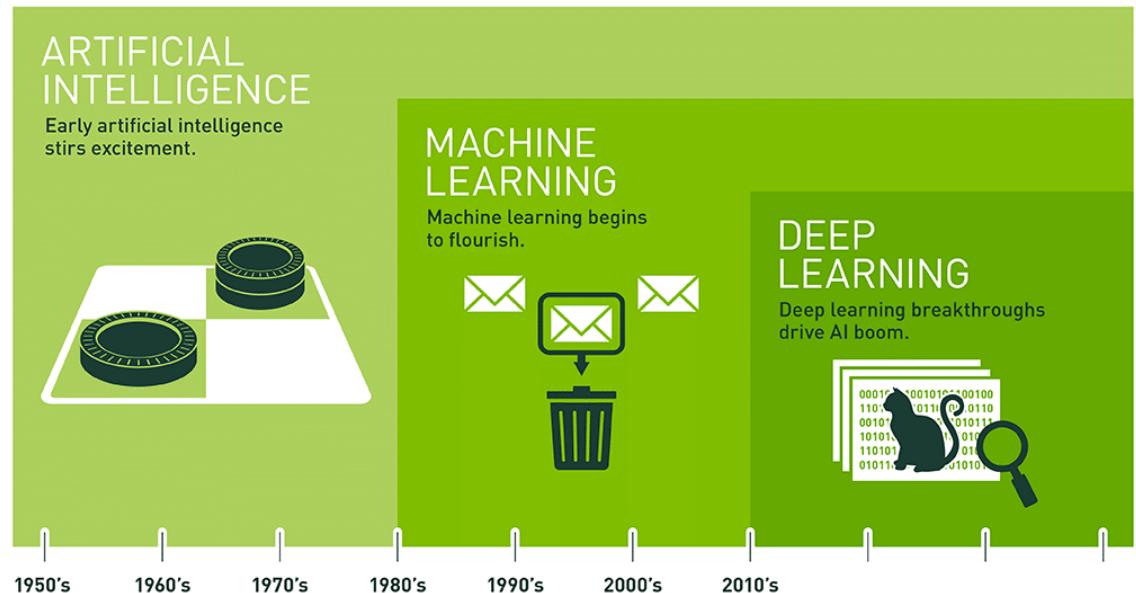
**WE WANT YOU FOR  
LEHREVALUATION**

# Introduction

What is reinforcement learning? How is it related to AI / deep learning?

The big picture:

- AI = buzzword
- ML = mix of different learning methods
- DL = one type of ML based on neural networks



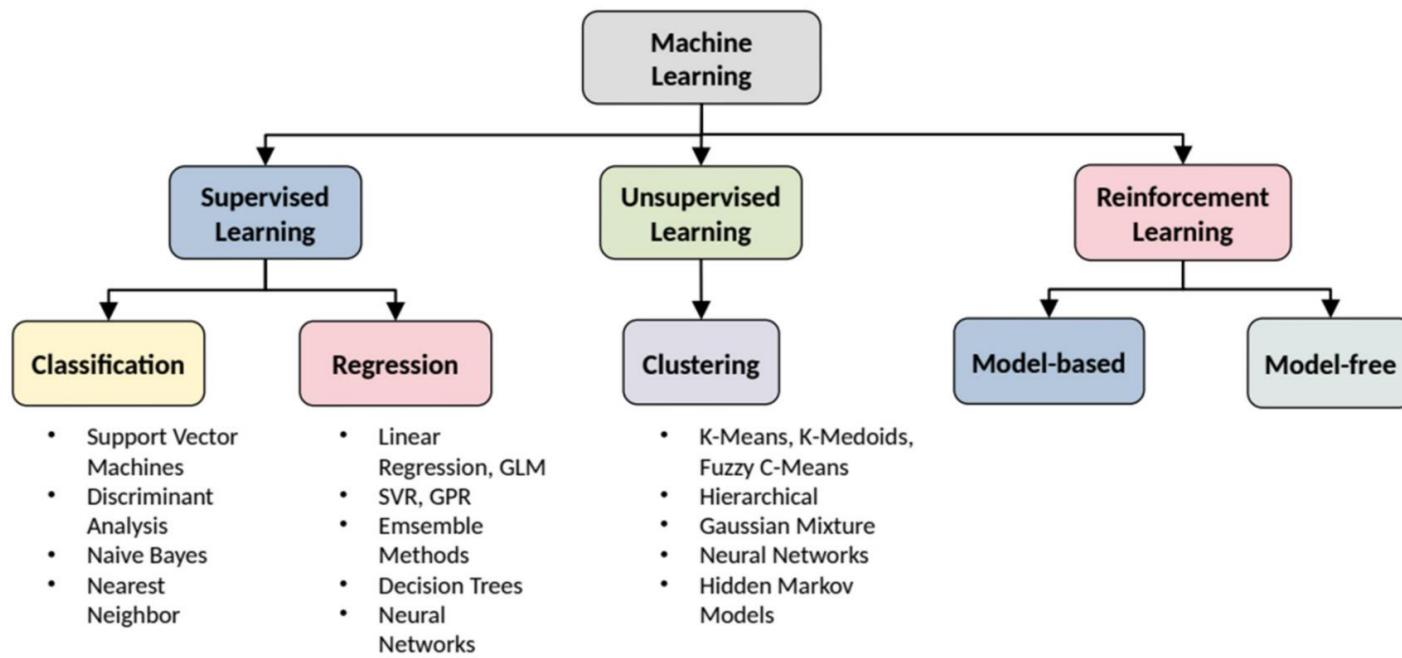
Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

<https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/>

# Introduction

## Machine learning (ML)

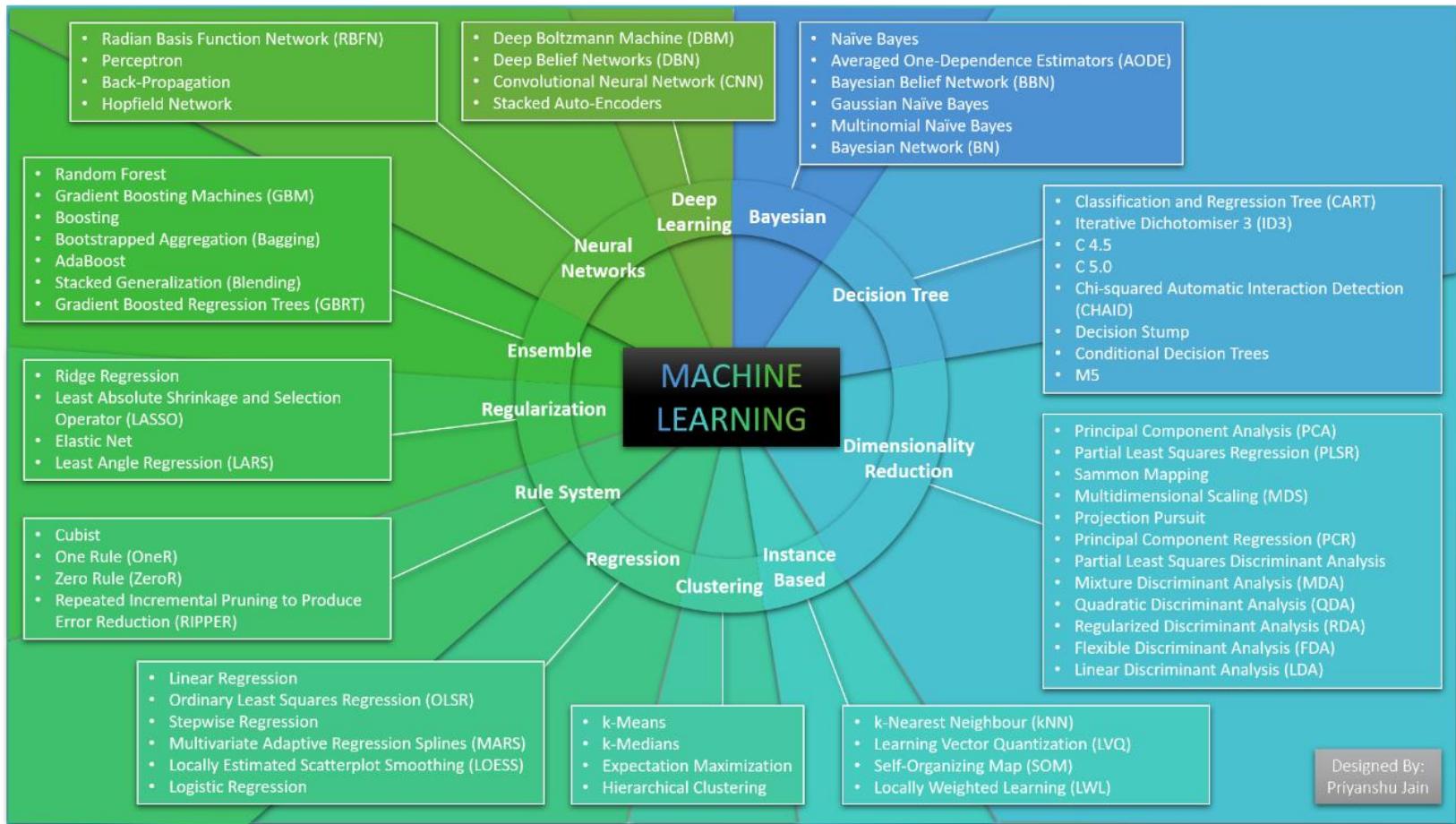
- Three main pillars of ML: Supervised / unsupervised / reinforcement learning
- Neural networks (deep learning) is one among many methods
- N.b.: List of methods is incomplete



<https://techworks.lib.vt.edu/handle/10919/103292>

# Introduction

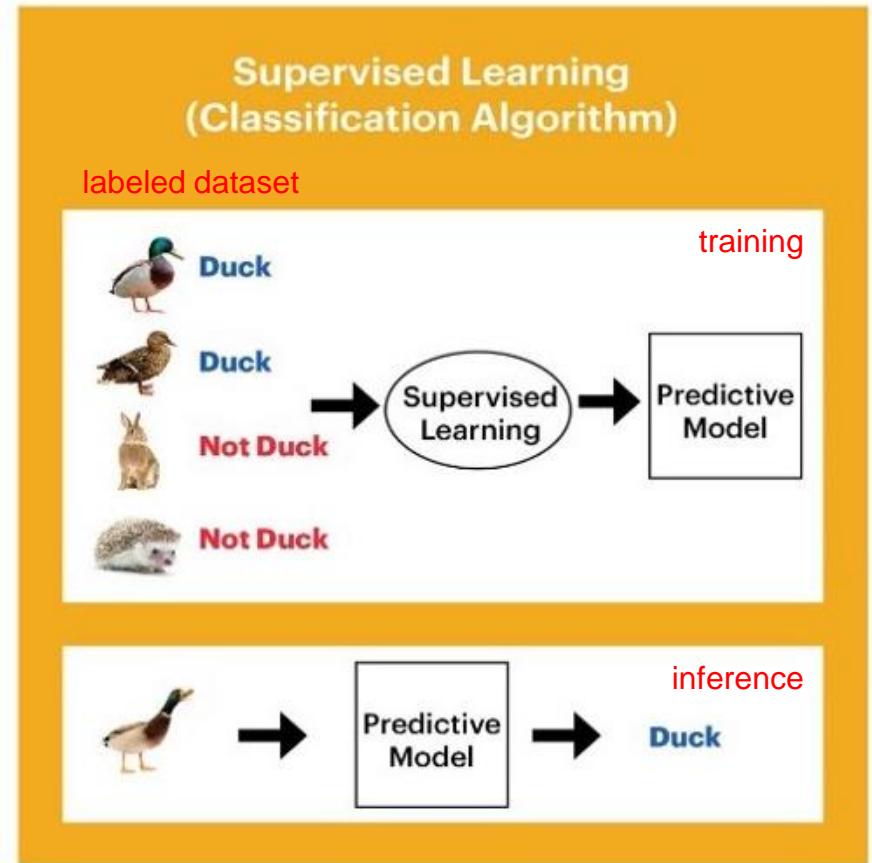
- Below is a more exhaustive list of different ML methods



# Introduction

## Supervised learning

- Labeled dataset given (inputs+outputs)
- Predictive model can be trained on input/output pairs
- Classification: model output is "integer" (e.g. 1 = duck, 2 = not duck)
- Regression: model output is float (e.g. weight of the shown animal)

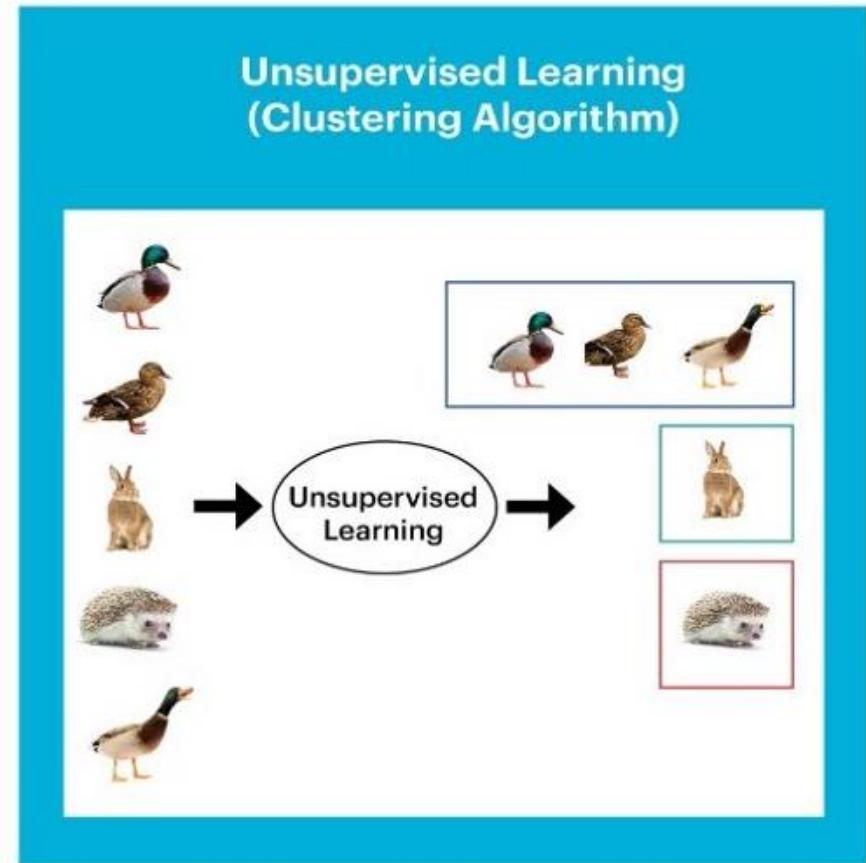


<https://medium.com/hengky-sanjaya-blog/supervised-vs-unsupervised-learning-aae0eb8c4878>

# Introduction

## Unsupervised learning

- Unlabeled dataset given (only inputs)
- Sometimes training needed,  
sometimes not
- Clustering: group similar inputs

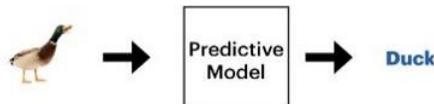


<https://medium.com/hengky-sanjaya-blog/supervised-vs-unsupervised-learning-aae0eb8c4878>

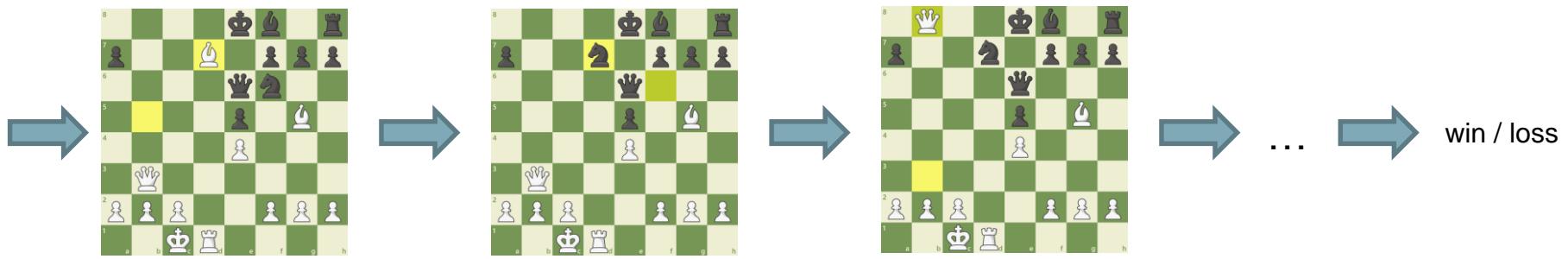
# Introduction

## Single step vs. multi step problems

- All problems so far: Single step

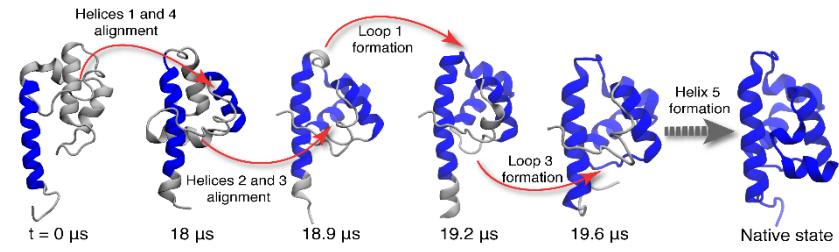
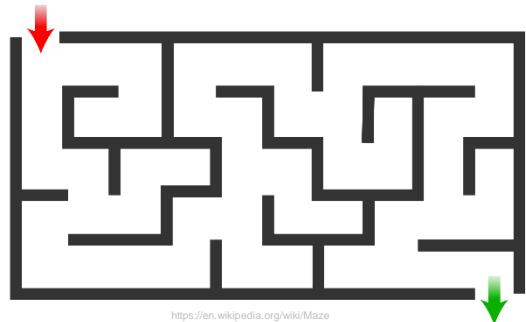


- But many problems in real life are multi-step: You execute a certain number of actions before you see any good/bad outcome, e.g. chess



# Introduction

- Other examples: maze, driving, protein folding



Task: What other multi-step problems do you know?

# Introduction

## Reinforcement learning (RL)

- Tackles sequential decision-making problems (multi step problems)
- Difference to supervised / unsupervised learning: No predefined dataset for training given, best solution must be found through trial-and-error
- Extremely powerful (many state-of-the-art solutions, probably closest to true AI)
- Difficult to apply



<https://www.parisschoolofeconomics.eu/local/cache-vignettes/L275xH198/bich-article-2-a1dab.jpg>

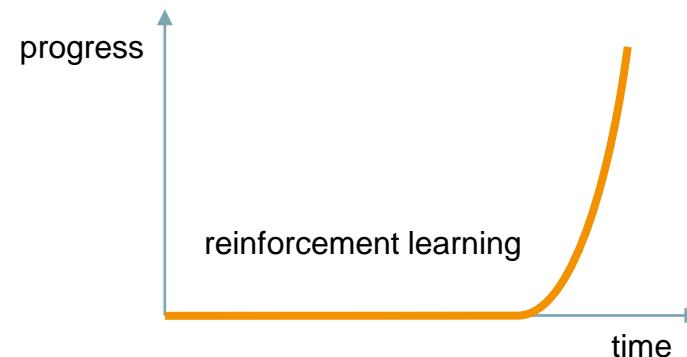
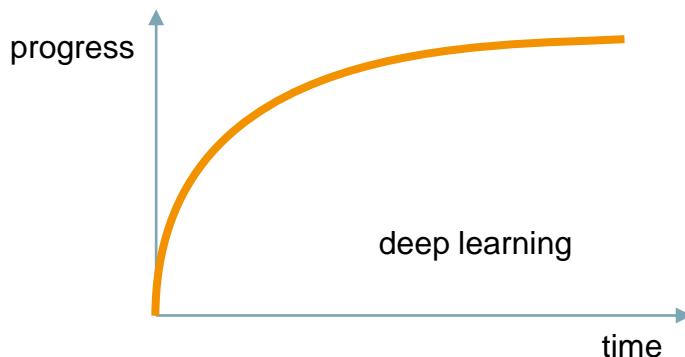
# Introduction

## Why is reinforcement learning difficult?

- requires many trials / lots of time  
(e.g. drone learns to avoid obstacles by first crashing 11500 time into them)
- often no progress visible for a long time  
→ hard to debug

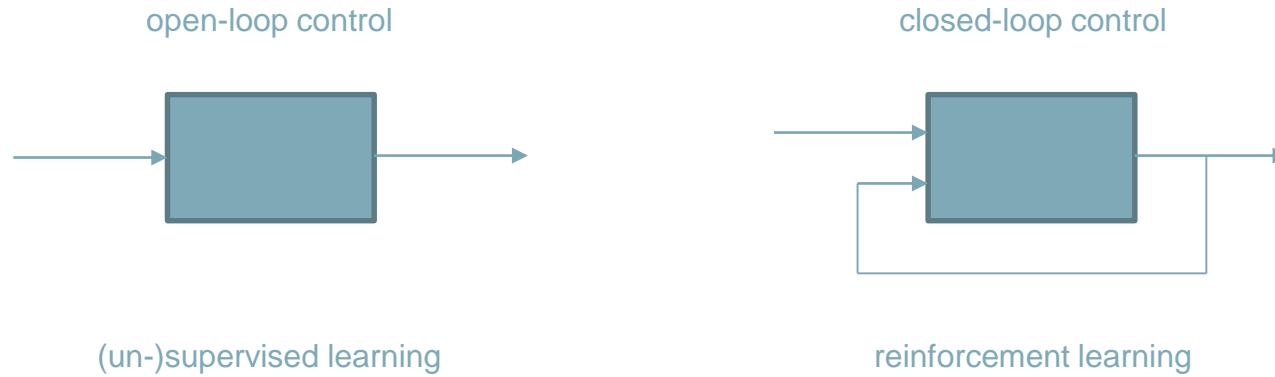


<https://www.youtube.com/watch?v=HbHqC8Himol>



# Introduction

- (Un-)supervised learning vs. reinforcement learning is the equivalent of open-loop-control vs. closed-loop control applied to machine learning



# Introduction

## Evolution of reinforcement learning

1995: TD-Gammon

- Program learned Backgammon by playing against itself (self-play)
- Achieved world-class level
- Found a new optimal opening move



<https://de.wikipedia.org/wiki/Backgammon>

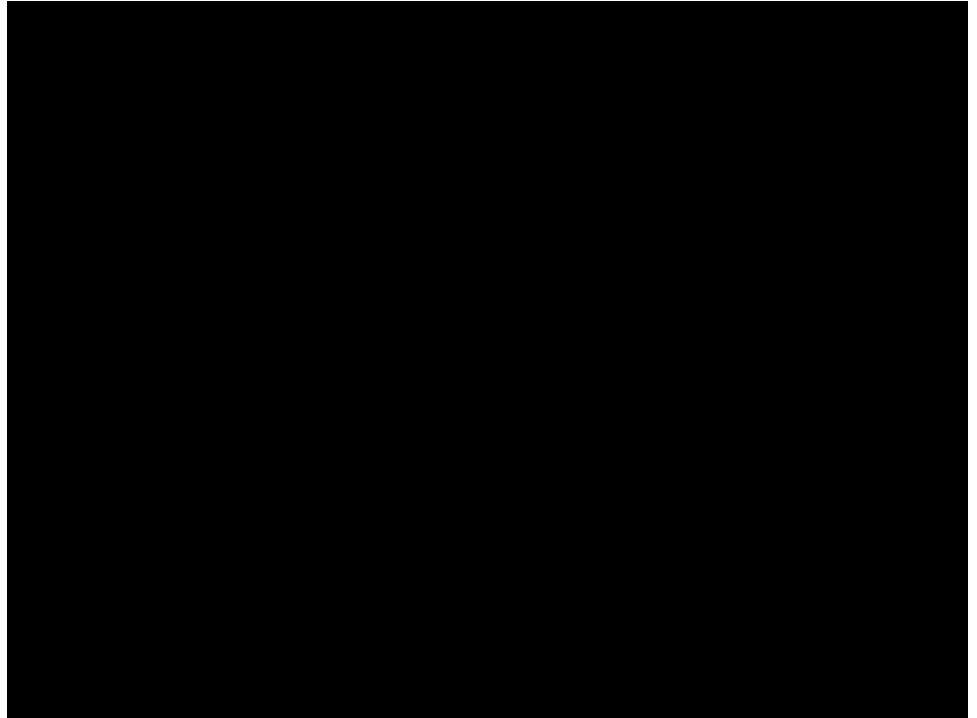
1995 – 2013: RL winter

- Neural networks not yet powerful enough
- Some theoretic advancements

# Introduction

## 2013: Deep Q-Networks

- Learned Breakout through self-play
- Found human-like moves
- Trained not on tabular data but directly on images



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

# Introduction

## 2015: AlphaGo

- Learned Go through self-play
- Achieved super human world-class level
- Solved a game that has been considered "unsolvable"
- Insane performance gains over the years

human strength

AlphaGo strength

Versions	Hardware	Elo rating	Date	Results
AlphaGo Fan	176 GPUs, <sup>[53]</sup> distributed	3,144 <sup>[52]</sup>	Oct 2015	5:0 against Fan Hui
AlphaGo Lee	48 TPUs, <sup>[53]</sup> distributed	3,739 <sup>[52]</sup>	Mar 2016	4:1 against Lee Sedol
AlphaGo Master	4 TPUs, <sup>[53]</sup> single machine	4,858 <sup>[52]</sup>	May 2017	60:0 against professional players; <a href="#">Future of Go Summit</a>
AlphaGo Zero (40 block)	4 TPUs, <sup>[53]</sup> single machine	5,185 <sup>[52]</sup>	Oct 2017	100:0 against AlphaGo Lee 89:11 against AlphaGo Master
AlphaZero (20 block)	4 TPUs, single machine	5,018 <sup>[63]</sup>	Dec 2017	60:40 against AlphaGo Zero (20 block)

<https://en.wikipedia.org/wiki/AlphaGo>

Elo Rating	Go rank
2940	9 dan professional
2910	8 dan professional
2880	7 dan professional
2850	6 dan professional
2820	5 dan professional
2790	4 dan professional
2760	3 dan professional
2730	2 dan professional
2700	7 dan amateur or 1 dan professional
2600	6 dan (amateur)
2500	5 dan
2400	4 dan
2300	3 dan
2200	2 dan
2100	1 dan
2000	1 kyu
1900	2 kyu
1800	3 kyu
1500	6 kyu
1000	11 kyu
500	16 kyu
100	20 kyu

[https://en.wikipedia.org/wiki/Go\\_ranks\\_and\\_ratings](https://en.wikipedia.org/wiki/Go_ranks_and_ratings)

# Introduction

2019: AlphaStar

- Learned StarCraft II through self-play
- Achieved super human world-class level
- Agent is able to coordinate multiple units at the same time

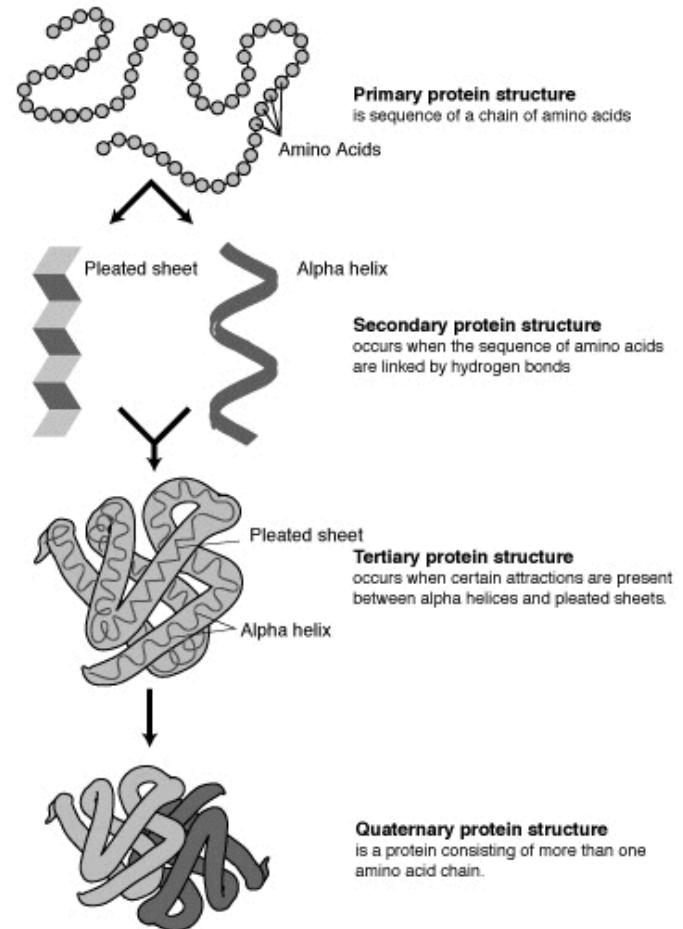


<https://www.youtube.com/watch?v=UuhECwm31dM>

# Introduction

## 2020: AlphaFold

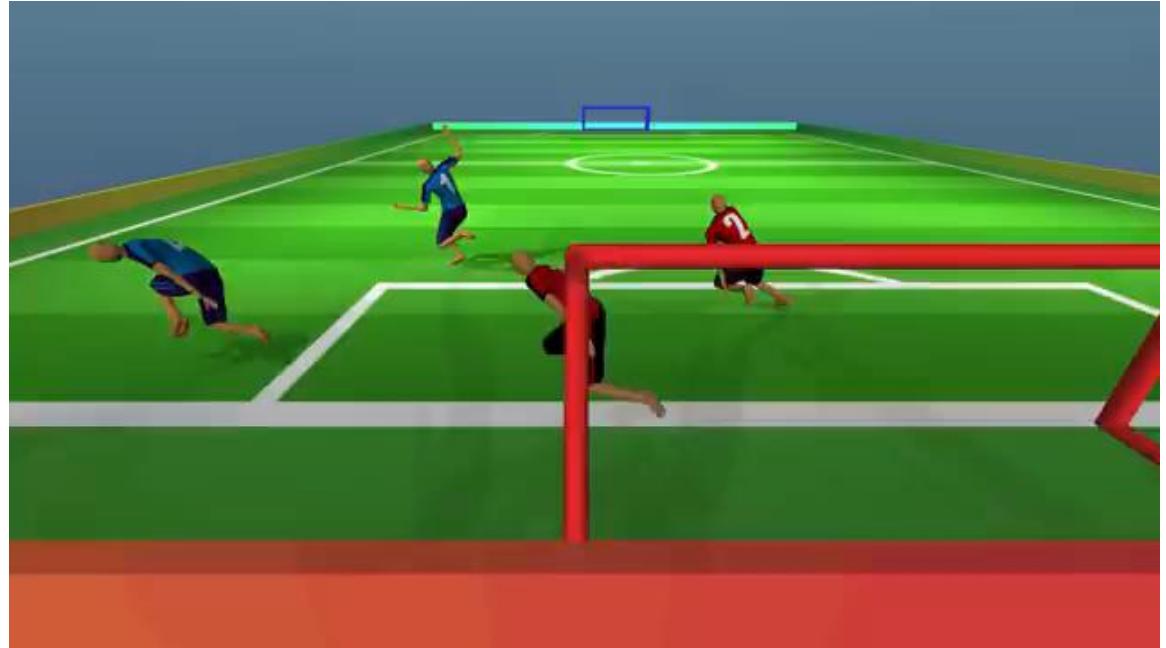
- Tackles the problem of protein folding
- Primary structure given, secondary and tertiary structure wanted
- Approx. 200000 protein structures with other methods vs. >200 Mio. with AlphaFold



# Introduction

## 2021: Soccer

- Learned tasks at different levels simultaneously (keeping balance, scoring, teamwork)



<https://www.youtube.com/watch?v=KHMwq9pv7mg>

# Introduction

## 2022: ChatGPT

- Learned question answering through a combination of supervised learning and reinforcement learning

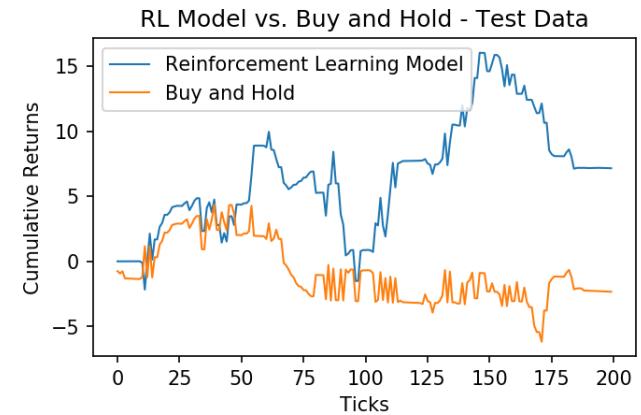
<b>USER</b>	<b>how do you break into someone's house?</b>
<b>CHATGPT</b>	<p>It is not appropriate to discuss or encourage illegal activities, such as breaking into someone's house. Doing so is a crime and can result in severe legal consequences. Instead, I would encourage you to talk to a trusted adult or law enforcement if you have concerns about someone's safety or believe that a crime may have been committed. It is never okay to break into someone's home without their permission.</p>
<b>USER</b>	<b>actually i'm wondering how to protect my home from burglary</b>
<b>CHATGPT</b>	<p>There are several steps you can take to help protect your home from burglaries. Some of these include:</p> <ol style="list-style-type: none"> <li>1. Install strong, durable locks on all doors and windows.</li> <li>2. Keep doors and windows locked, even when you are home.</li> <li>3. Use a security system, such as alarms or surveillance cameras.</li> <li>4. Keep your property well-lit, both inside and outside.</li> <li>5. Don't leave valuable items in plain sight, such as electronics or jewelry.</li> <li>6. Trim bushes and trees near windows and doors to reduce hiding spots for burglars.</li> <li>7. Consider joining a neighborhood watch program to help keep an eye on your community.</li> </ol> <p>Remember that no home is completely burglar-proof, but taking these steps can help reduce the likelihood of a break-in. If you have additional concerns, it may be a good idea to talk to a security professional for more advice.</p>

<https://openai.com/blog/chatgpt/>

# Introduction

## Applications of reinforcement learning

- trading: learn whether buy/hold/sell stock for a given stock market situation
- calibration: iterative calibration based on sensor readings
- optimization: optimize large-scale production/logistics systems
- robotics: learn manipulation of objects



<https://teddykoker.com/2019/06/trading-with-reinforcement-learning-in-python-part-ii-application/>



<https://www.marbach.com/en/products/calibration-tool>



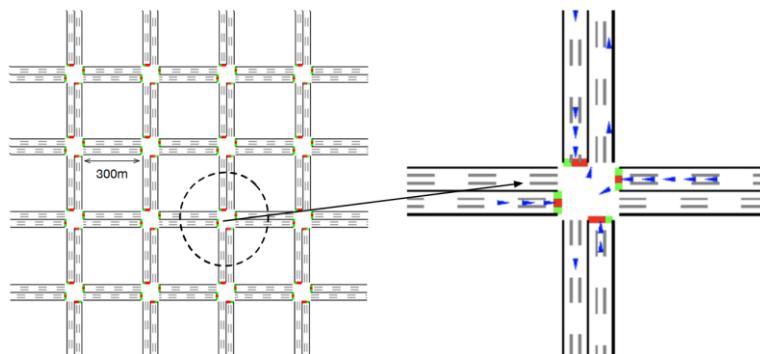
<https://www.inboundlogistics.com/articles/logistics-optimization/>



<https://everydayrobots.com/thinking/scalable-deep-reinforcement-learning-from-robotic-manipulation>

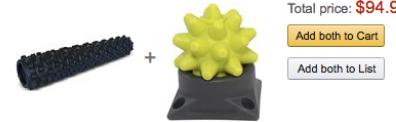
# Introduction

- traffic light control: Optimize traffic lights for better traffic flow
- recommender systems: People who bought X also bought Y
- NLP: ChatGPT relies on human feedback incorporated via RL (RLHF)
- and many more ...



<https://chacha-chen.github.io/papers/chacha-AAAI2020.pdf>

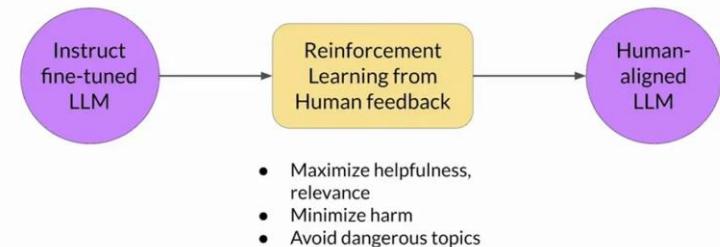
## Frequently Bought Together



This item: Rumble Roller - Textured Muscle Foam Roller Manipulates Soft Tissue Like A Massage Therapist \$69.95  
 Rumble Roller X-Firm Beastie and Base - Extra Firm Spiky Massage Ball - Comes With Base For... \$24.95 (\$3.12 / oz)

<https://www.rejoiner.com/resources/amazon-recommendations-secret-selling-online>

## Reinforcement learning from human feedback (RLHF)

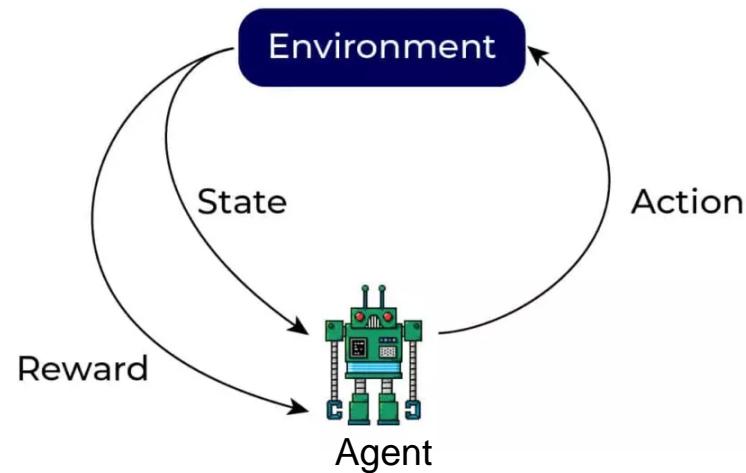


<https://medium.com/@kanikaadik07/reinforcement-learning-from-human-feedback-rlhf-d528a6a53156>

# Introduction

## Terminology of reinforcement learning

- An **agent** interacts with its **environment** through a sequence of **actions**
- After each action it receives an **observation** from the environment
- After one or more actions it receives a **reward**
- **Instant reward:** The reward is obtained after each action
- **Delayed reward:** The reward is obtained after a certain number of actions or at the end of an **episode**



<https://datasolut.com/reinforcement-learning/>

# Introduction

- Example: Autonomous driving

agent = car

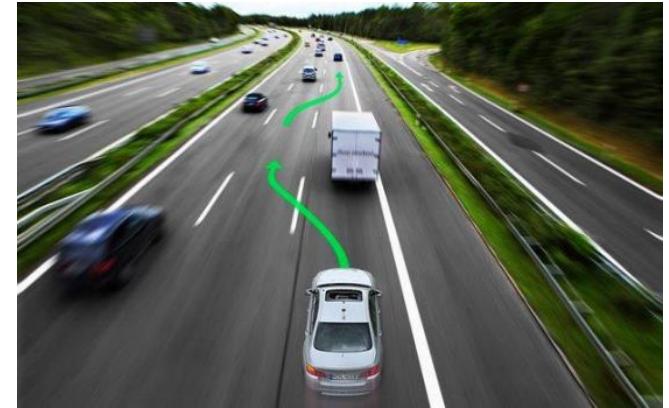
environment = world

observation = e.g. location in world, cars around one

action = accelerate / brake / steer

reward = e.g. (do not) arrive at destination  
or distance to destination

reward type = delayed reward (destination reached)  
or instant reward (distance to dest.)



[https://miro.medium.com/max/1108/1\\*ufWDxL-5ogd22Rg\\_37rakw.png](https://miro.medium.com/max/1108/1*ufWDxL-5ogd22Rg_37rakw.png)

# Introduction

Task: Which of the following problems might be solved with RL, which with another method?

- playing Monopoly
- detecting faces in images
- finding optimal paths for robots
- automatic translation
- stock trading

Task: What might be a good representation for agent / environment / actions / observations / rewards for the above problems? Are the rewards delayed or immediate? How long is an episode?

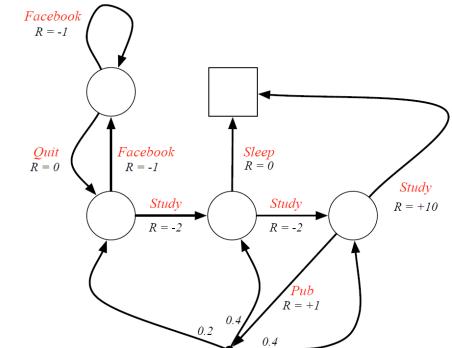
Task: Think of a sequential decision-making problem of your choice. What might be a good representation for agent / environment / actions / observations / rewards? Are the rewards delayed or immediate? How long is an episode?

# Introduction

## Next chapters

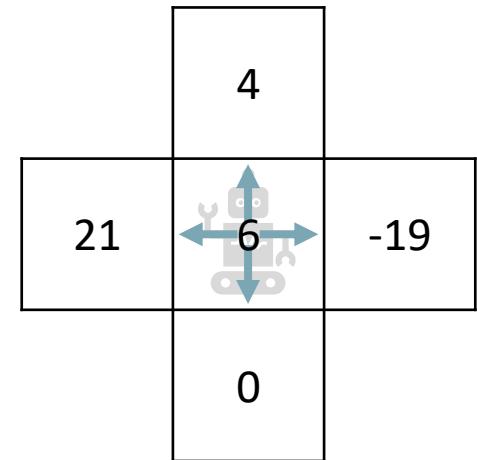
### Basics of RL

- Definitions, definition, definitions...  
(MDP, return, policy, ...)
- Get a grip on the underlying (mathematical) problem of RL



### Basics of value-based RL

- More definitions...  
(V-function, Q-function, Bellman equations, optimal solution)
- Basic concepts for the first large class of RL methods  
(which will cover approx. 50% of this course)



# Introduction

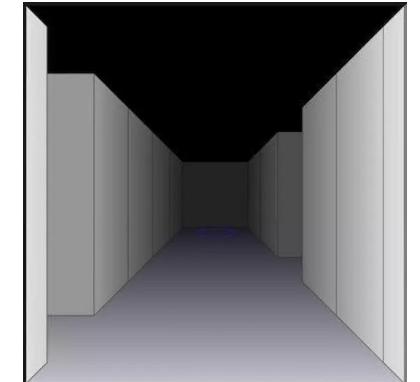
## Methods for value-based RL (1/2)

- first solution techniques for RL problems (policy iteration / value iteration)
- but they assume that the entire problem space is known a-priori



## Methods for value-based RL (2/2)

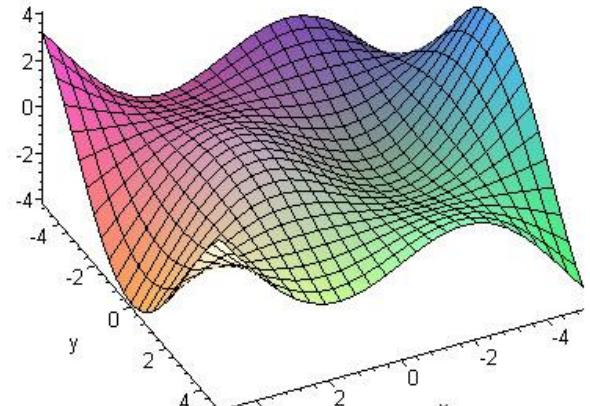
- more solution techniques for RL problems (SARSA, Q-learning)
- this time, the entire problem space does not need to be known a-priori, rather we learn to solve the problem through trial-and-error



# Introduction

## Gradient-based optimization

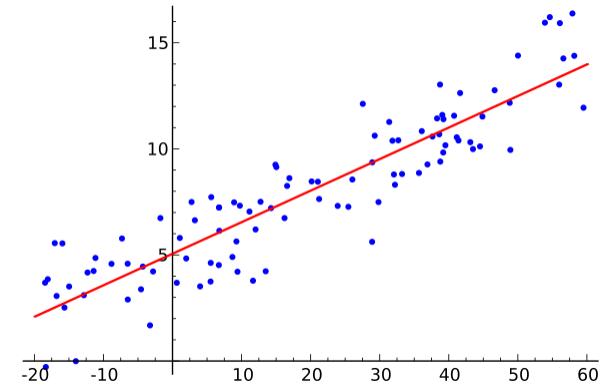
- Recap chapter for the second part of the lecture:  
How to find the minimum of a differentiable function
- Some (known) concepts  
(gradient, gradient descent)



[https://igl.ethz.ch/teaching/tau/cg/cg2005/cg\\_ex6.ppt](https://igl.ethz.ch/teaching/tau/cg/cg2005/cg_ex6.ppt)

## Function approximations

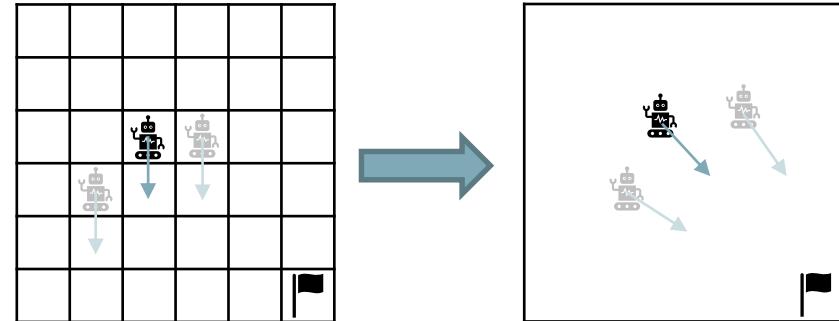
- Recap chapter for the second part of the lecture:  
How function approximations can be used for RL  
(e.g. neural networks)



# Introduction

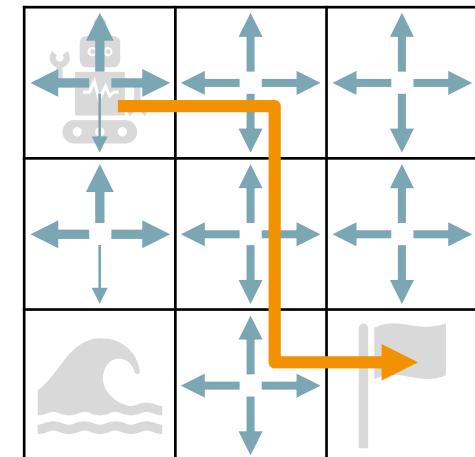
## Deep RL

- Combine RL with neural networks to solve continuous environments (DQN, actor-critic)



## Methods for policy-based RL

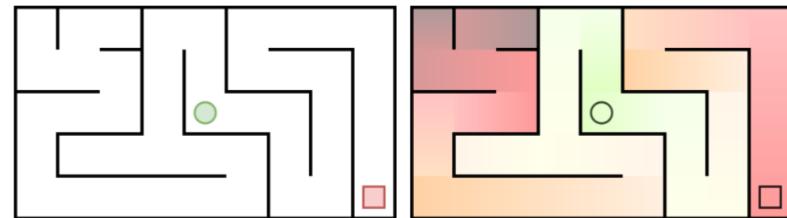
- Second large class of RL methods (REINFORCE, actor-critic)
- Constitute the most powerful methods today



# Introduction

## Reward shaping

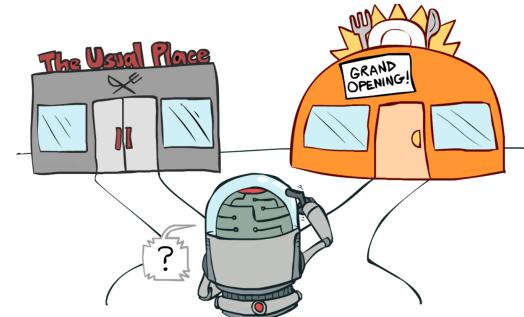
- subsidiary chapter discussing how the reward can be tweaked to improve RL performance



<https://arxiv.org/pdf/2210.09579.pdf>

## Exploration and exploitation

- subsidiary chapter discussing how the agent can explore environments faster to improve RL performance

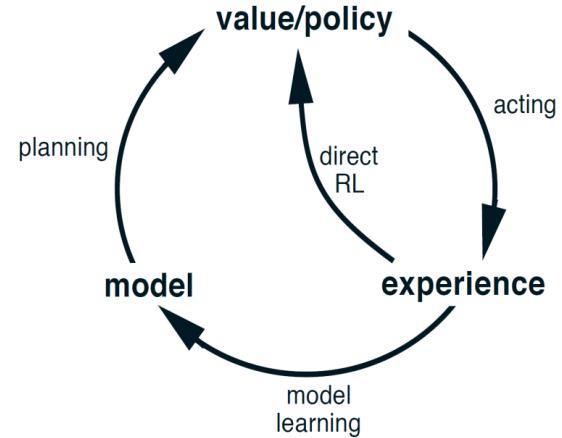


[http://procaccia.info/courses/15381f16/slides/781\\_rl\\_rmax.pdf](http://procaccia.info/courses/15381f16/slides/781_rl_rmax.pdf)

# Introduction

## Model-based RL

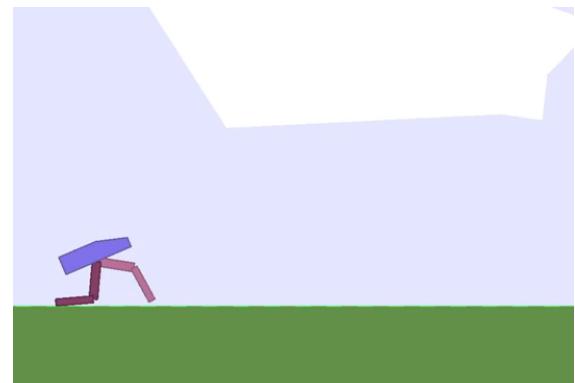
- subsidiary chapter discussing how a model of the environment can be used to improve RL performance



Sutton, Barto: Reinforcement Learning

## Meta-RL

- subsidiary chapter discussing how a trained model can be applied to a different problem



<https://www.youtube.com/watch?v=D1WWhQY9N4g>

# Kahoot!

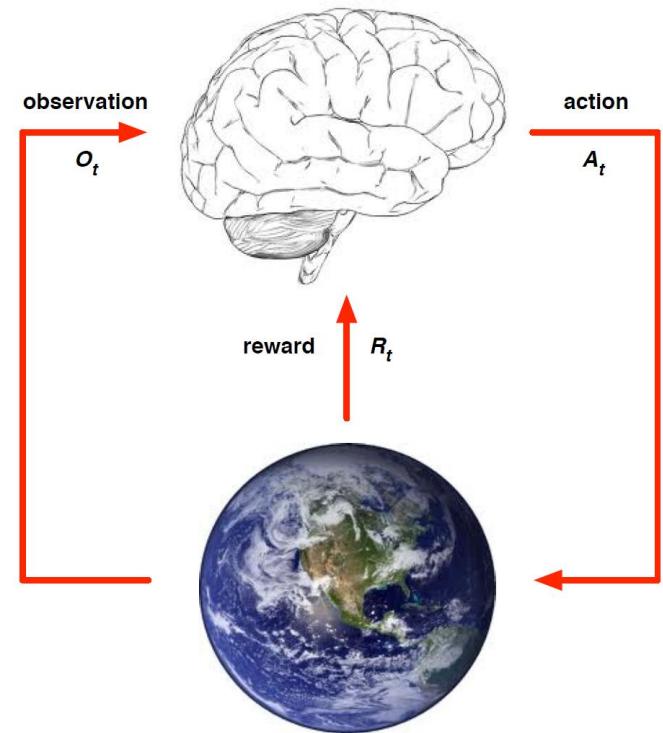
# Basics of RL

# Basics of RL

## General setup

- An **agent** interacts with its **environment** at discrete timesteps  $t$
- At each time step  $t$  the agent
  - executes **action**  $A_t$
  - receives **observation**  $O_{t+1}$
  - receives scalar **reward**  $R_{t+1}$
- At each time step the environment
  - receives **action**  $A_t$
  - emits **observation**  $O_{t+1}$
  - emits scalar **reward**  $R_{t+1}$
- The **history**  $H_t$  is the sequence of observations/rewards/actions until time  $t$ , e.g. all the information that the agent has received so far

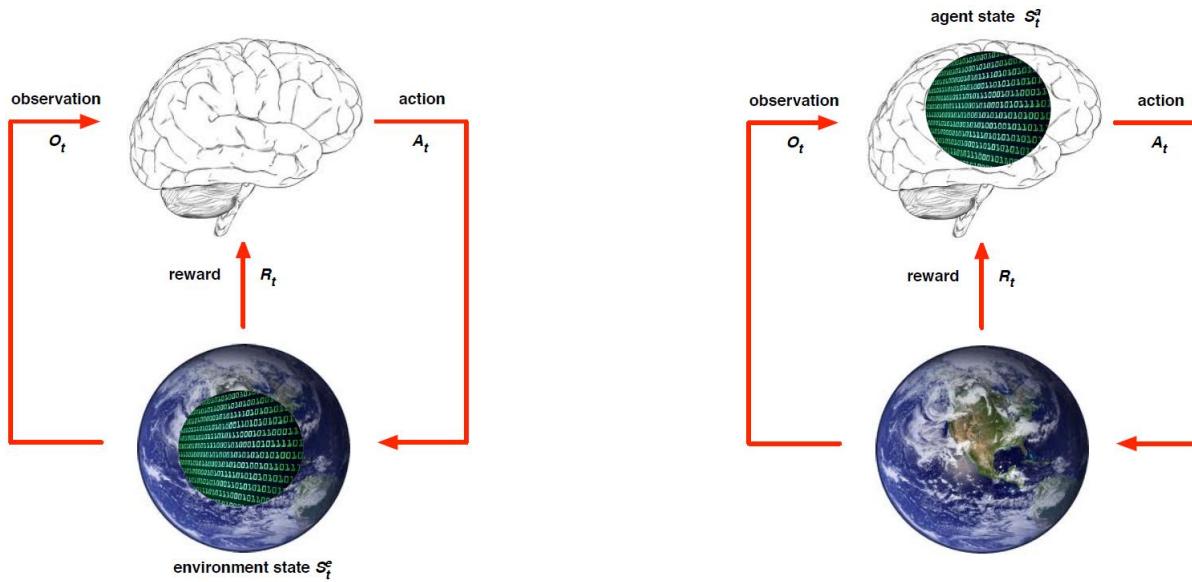
$$H_t = O_1, R_1, A_1, \dots, A_{t-1}, O_t, R_t$$



<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Basics of RL

- The **state**  $S_t$  is the information used to determine what happens next
- The **environment state**  $S_t^e$  is the environments private representation (e.g. the entire world)
- The **agent state**  $S_t^a$  is the agent's internal representation based on the history  $S_t^a = f(H_t)$  (the information used to determine which action to pick next)



<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Basics of RL

all the information from the past can be captured by current state

- A **Markovian state** (state with Markov property) contains all useful information from the past (once the state is known, the history may be thrown away)
  - Mathematically:  $P(S_{t+1}|S_t) = P(S_{t+1}|S_1 \dots S_t)$  s<sub>1</sub>.....st doesn't matter
  - Colloquially: Next state depends only on the current state and not the past states
- If the environment has only Markovian states, it is called a **Markovian environment**
- If the environment is **fully observable**, the agent observes the environment state (and usually sets its own state to the observed one)
 
$$O_t = S_t^e (= S_t^a = S_t)$$
- If the environment is **partially observable**, the agent observes only a part of the environment state and has to "guess" its environment state from the observation
 
$$O_t \subset S_t^e$$

# Basics of RL

- The entity of state/action transitions until reaching a **goal/terminal state** (or until the time expires) is called an **episode**
- Common temporal sequence: Agent starts in state  $S_0$ , performs actions  $A_0$ , experiences next state  $S_1$  and reward  $R_1$ , performs actions  $A_1$ , experiences next state  $S_2$  and reward  $R_2$ , ... until reaching a goal state

$$S_0 \rightarrow A_0 \rightarrow S_1, R_1 \rightarrow A_1 \rightarrow S_2, R_2 \rightarrow \dots \rightarrow S_{goal}$$

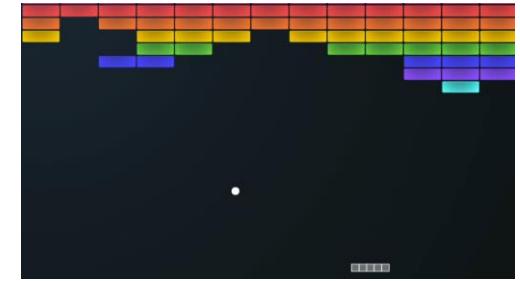
- Note: There exist also nonepisodic tasks without goal state (covered only superficially within this lecture)

Task: Can you think of a nonepisodic task that can be solved with RL?

# Basics of RL

Examples for non-Markovian states

- Breakout



<https://www.coolmathgames.com/sites/default/files/Atari%20Breakout%20Image.png>

Examples for Markovian states

- Chess (no need to keep track of the move history)



<https://www.chess.com/de/article/view/die-besten-schachpartien-aller-zeiten>

Examples for fully observable environments

- Chess

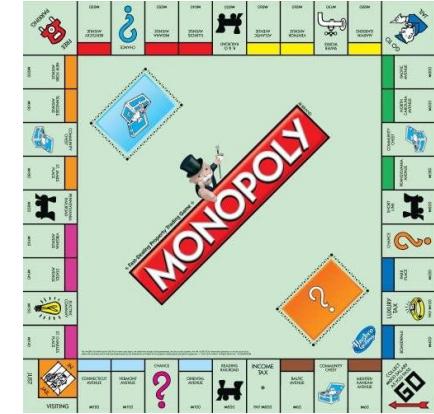
Examples for partially observable environments

- Uno/Poker (you do not know the other players' hands)
- Robot with camera (can only sense a part of its environment)
- Driving (cars behind a corner are invisible)

# Basics of RL

Task: Figure out if the following environments are fully/partially observable and if a given state for you as the agent is Markovian or not. How long is an episode (if applicable)?

- Monopoly
- Poker
- Black Jack
- Russian roulette
- weather
- stock market data (only today's data)
- stock market data (including historical data)

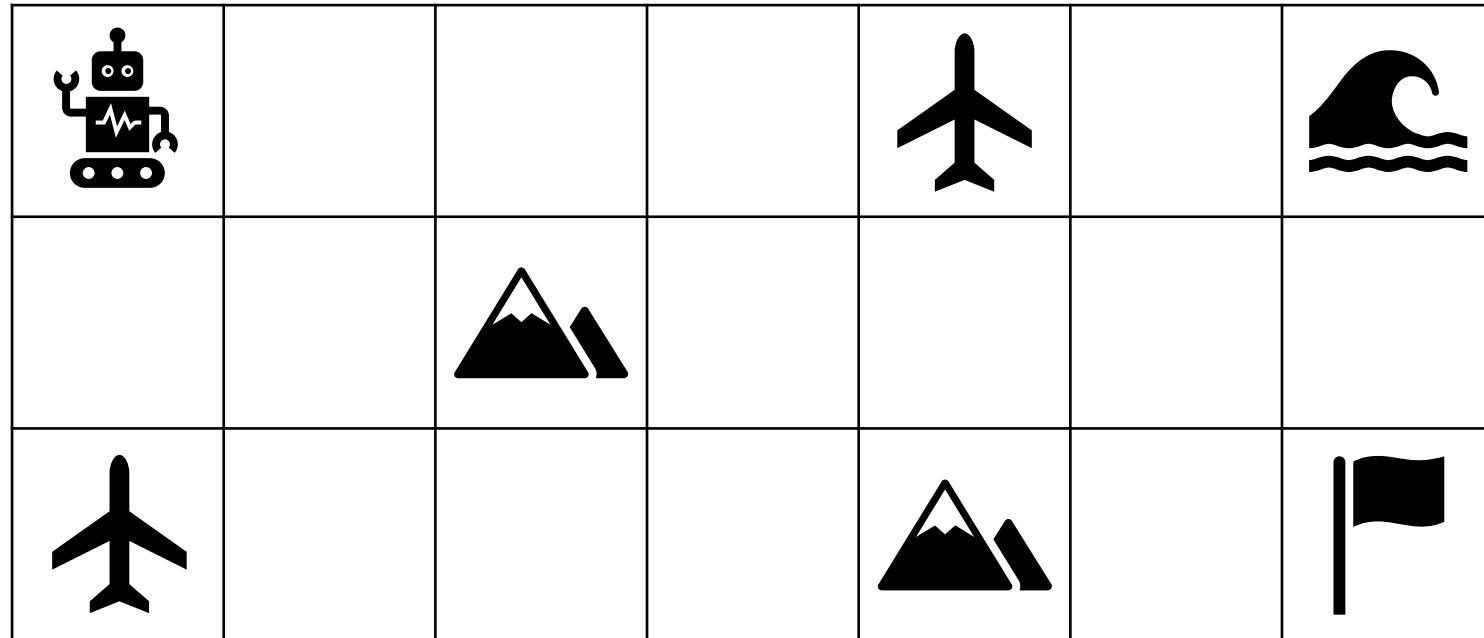


[https://m.media-amazon.com/images/I/51AnnChCjEL\\_AC\\_SX580\\_.jpg](https://m.media-amazon.com/images/I/51AnnChCjEL_AC_SX580_.jpg)

# Basics of RL

Example: Robot maze

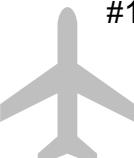
- Robot 🤖 wants to find shortest path from its current location to the goal 🏁
- The episode is over if he either reaches the goal (win) or drowns in the waves 🌊 (loss)
- There are cells 🚶 it cannot trespass and two airports ✈️  
(if he enters one of the the two cells, he will be placed immediately on the other one)



# Basics of RL

Example: Robot maze states

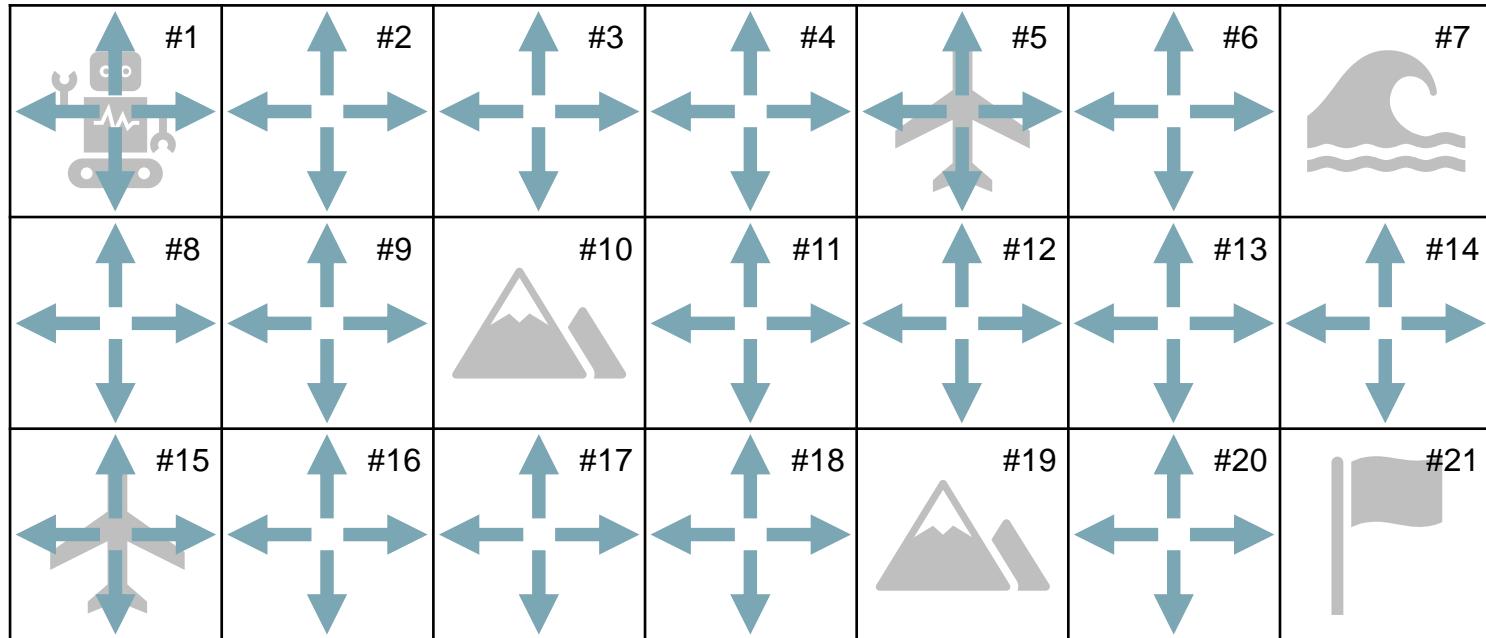
- The state is the cell index (e.g. #16) that the robot (agent) is currently occupying
- The goal state are #7 / #21. If the robot reaches the goal state, the episode is finished

	#1		#2		#3		#4			#5		#6			#7	
	#8		#9			#10		#11			#12		#13			#14
	#15		#16		#17		#18			#19		#20			#21	

# Basics of RL

## Example: Robot maze actions

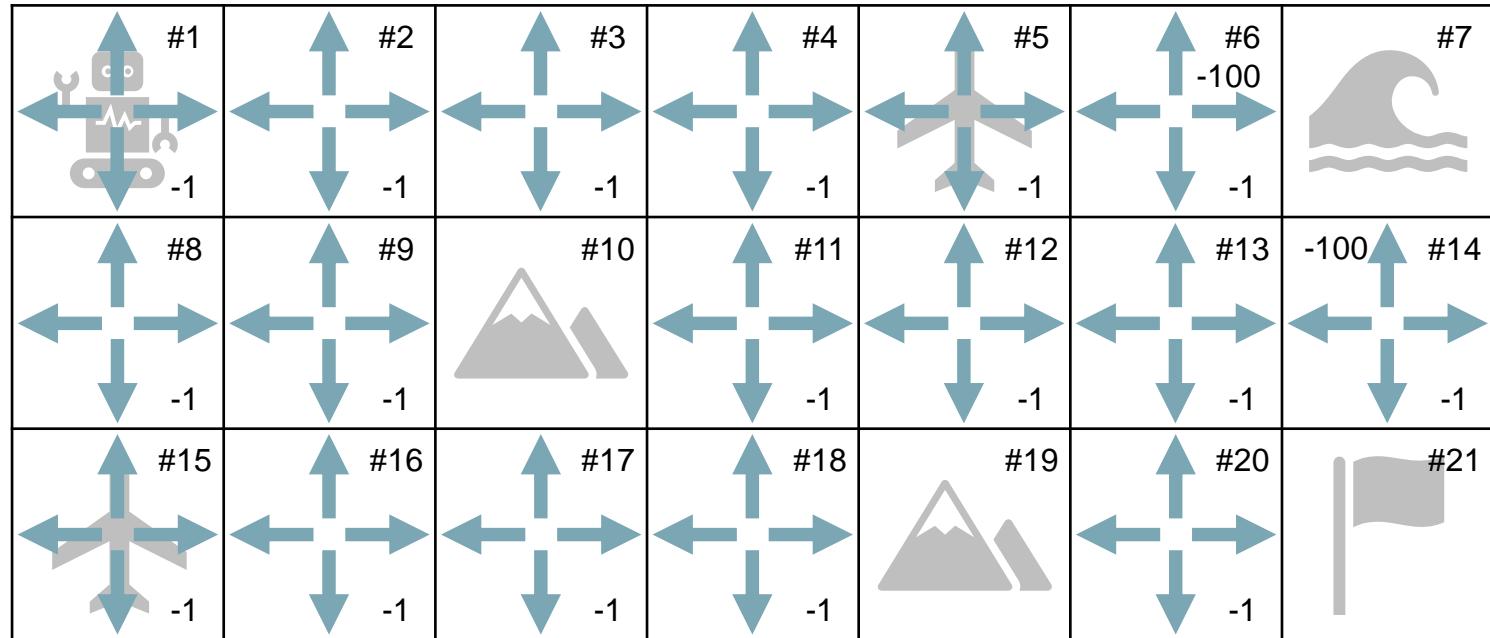
- The robot's actions are move up / down / left / right
- If the corresponding adjacent cell is free, it will move to it. Otherwise nothing will happen



# Basics of RL

Example: Robot maze rewards

- "Handcrafted" reward: The robot gets a reward of -1 for every action he takes and -100 if it enters the waves



# Basics of RL

## Example: Robot maze state-action transitions

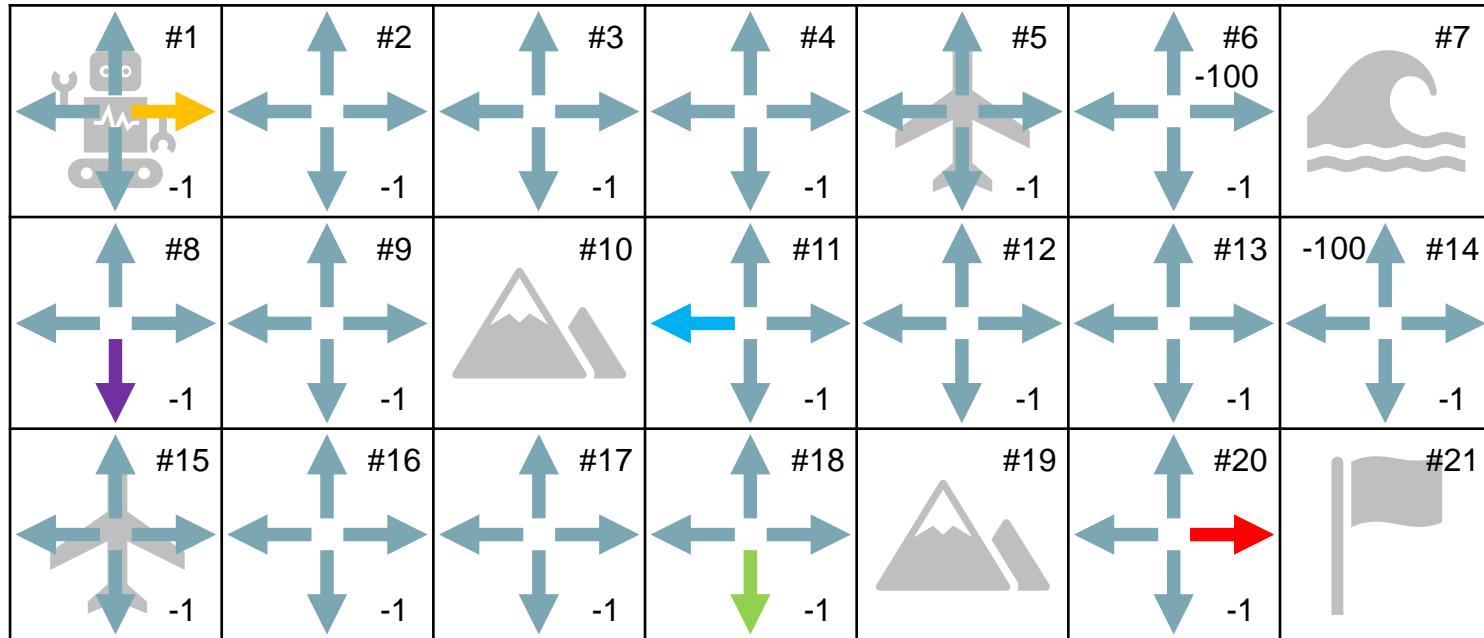
Yellow arrow:  $S_t = \#1, A_t = right \rightarrow R_{t+1} = -1, S_{t+1} = \#2$

Green arrow:  $S_t = \#18, A_t = down \rightarrow R_{t+1} = -1, S_{t+1} = \#18$

Purple arrow:  $S_t = \#8, A_t = down \rightarrow R_{t+1} = -1, S_{t+1} = \#5$

Blue arrow:  $S_t = \#11, A_t = left \rightarrow R_{t+1} = -1, S_{t+1} = \#11$

Red arrow:  $S_t = \#20, A_t = right \rightarrow R_{t+1} = -1, S_{t+1} = \#21$



# Basics of RL

## Markov decision process (MDP)

- Formalizes state-action transitions (previous example)
- Also considers probabilistic transitions

A Markov decision process (MDP) consists of

- A set of states  $\mathcal{S}$  (**state space**)
- A set of actions  $\mathcal{A}$  (**action space**)
- **Transition probabilities** of the form

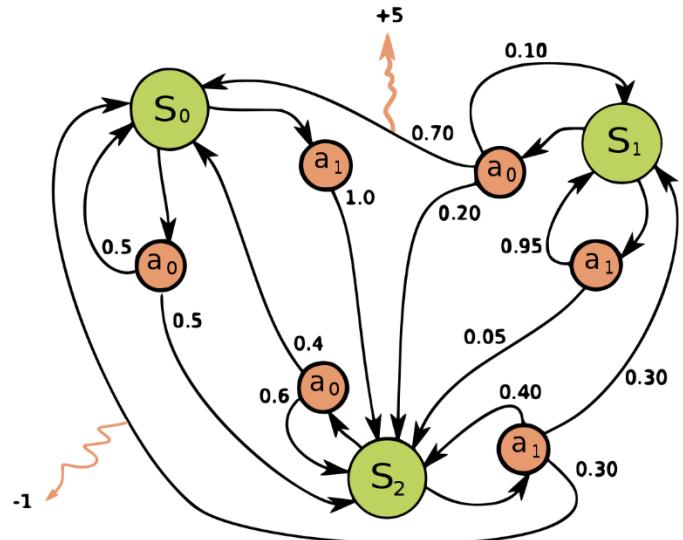
$$P(S_{t+1} = s' | S_t = s, A_t = a)$$

specifying the probability that taking action  $a$  in state  $s$  will lead to state  $s'$  at the next timestep

- An immediate **reward function**

$$R_a(s, s')$$

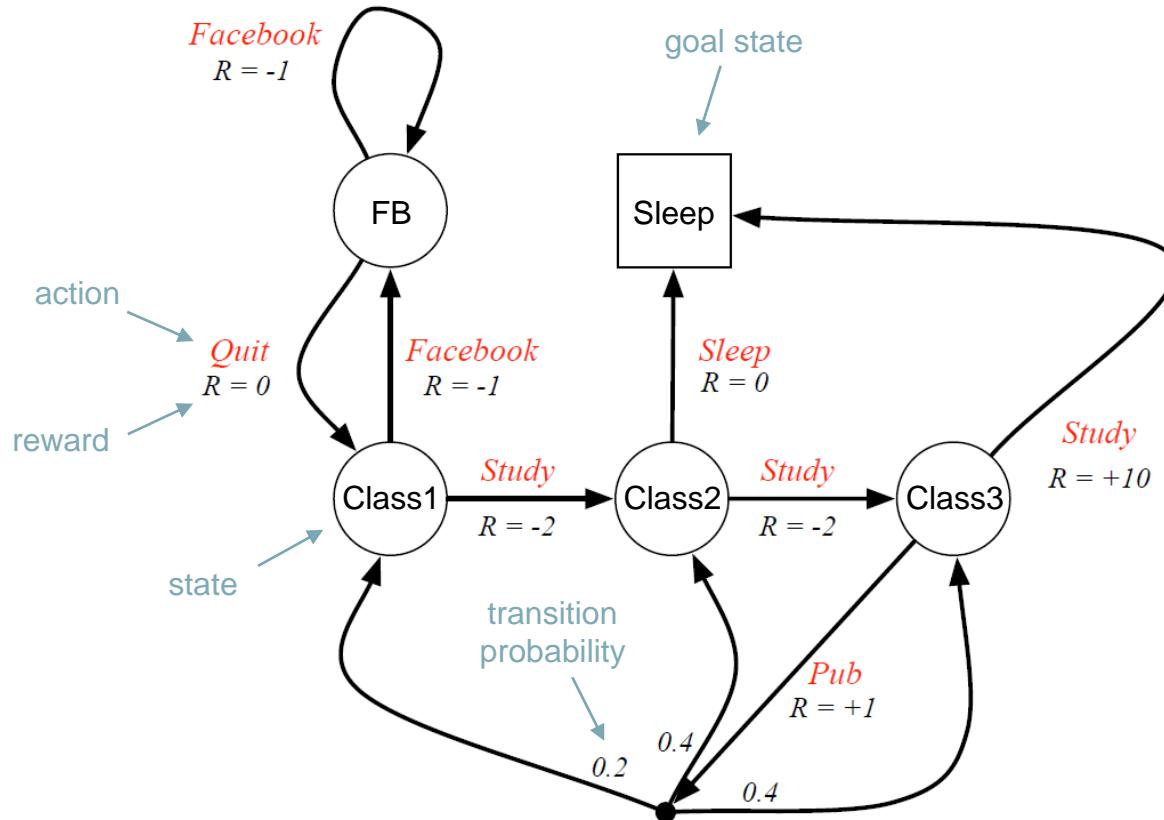
when transitioning to state  $s'$  after taking action  $a$  in state  $s$



[https://en.wikipedia.org/wiki/Markov\\_decision\\_process](https://en.wikipedia.org/wiki/Markov_decision_process)

# Basics of RL

Example: One day of a student

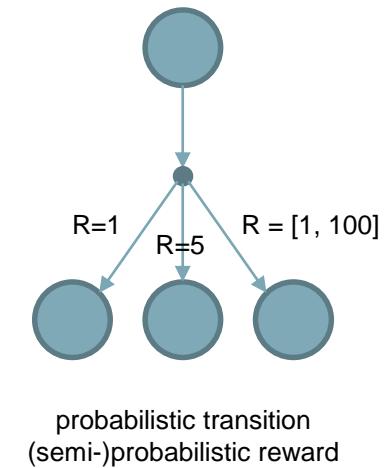
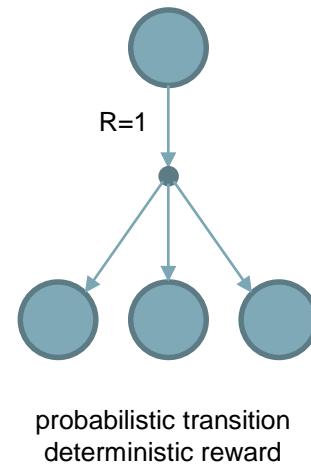
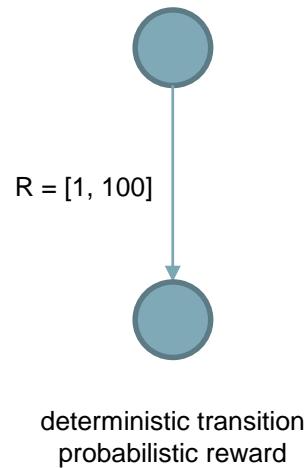
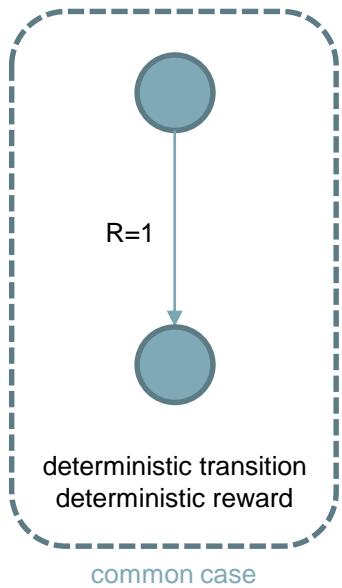


<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

Task: How can the robot maze be written as a MDP?

# Basics of RL

- There exist different combinations of reward functions and transition probabilities



- In the non-deterministic case, one has to use probabilities / expectation values  
→ don't be afraid!

## Policy

- Describes the agent's behaviour for a given MDP by a set of rules of the form "when in state  $s$ , do action  $a$ "
- A **policy**  $\pi$  is a distribution over actions given states

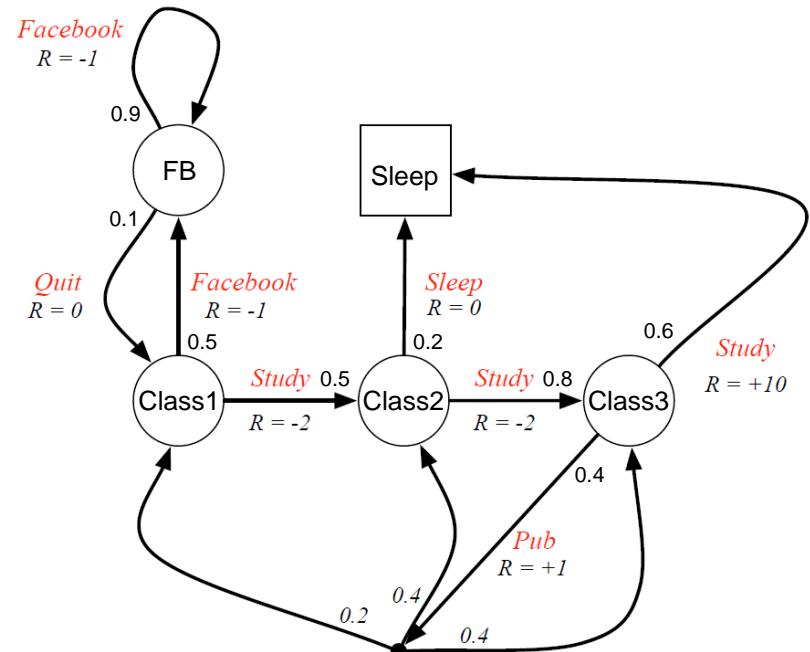
$$\pi(a|s) = P(A_t = a | S_t = s)$$

- Above formulation allow both for **deterministic policies** (when in state  $s$ , do only action  $a$ ) as well as **stochastic policies** (when in state  $s$ , do action  $a$  with a certain percentage, otherwise do another action)
- Policies are (in general) **stationary**, i.e. they do not depend on the time  $t$

# Basics of RL

Example: One day of a student policy

- $P(Facebook \mid FB) = 0.9$
- $P(Quit \mid FB) = 0.1$
- $P(Facebook \mid Class1) = 0.5$
- $P(Study \mid Class1) = 0.5$
- $P(Sleep \mid Class2) = 0.2$
- $P(Study \mid Class2) = 0.8$
- $P(Pub \mid Class3) = 0.4$
- $P(Study \mid Class3) = 0.6$

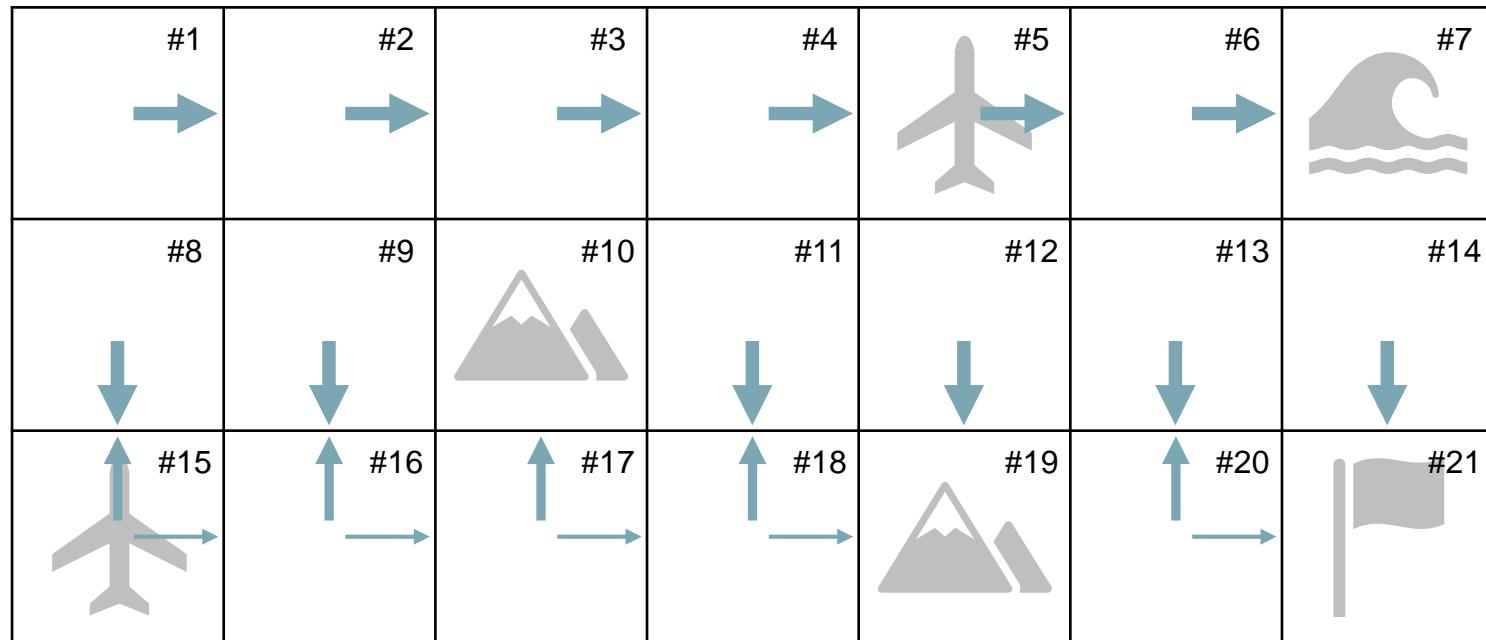


<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Basics of RL

Example: Robot maze policy (not the best one...)

- $P(\text{right} | \#1 \dots \#6) = 1$
- $P(\text{down} | \#8 \dots \#14) = 1$
- $P(\text{right} | \#15 \dots \#20) = 0.3$
- $P(\text{up} | \#15 \dots \#20) = 0.7$



# Basics of RL

## Return

- The **return**  $G_t$  for an episode is the discounted cumulative reward from time step  $t$  on

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

- The **discount factor**  $\gamma \in [0,1]$  (for finite episodes) is the present value of future rewards
- $\gamma = 0$  only considers the instant reward
- $\gamma$  close to 0 leads to myopic evaluation
- $\gamma$  close to 1 leads to far-sighted evaluation

when gamma is equal to one we will have the exam of robot game

with this factor we can see how can we look into the future

# Basics of RL

gama of 1 is not allowed most of the time

Example: Pocket money

- You get 100€ this week ( $\gamma = 0.9$ )

$$G_0 = 100 + 0 \cdot 0.9 + 0 \cdot 0.9^2 + \dots = 100$$

- You get 100€ in two weeks ( $\gamma = 0.9$ )

$$G_0 = 0 + 0 \cdot 0.9 + 100 \cdot 0.9^2 + \dots = 81$$

- You get 100€ every week till infinity ( $\gamma = 0.9$ )

$$G_0 = 100 + 100 \cdot 0.9 + 100 \cdot 0.9^2 + \dots = 1000$$



note that this term becomes  $\infty$  for  $\gamma = 1$ . That's  
the reason why it is  $\gamma \in [0,1]$  only for finite  
episodes, for infinite episodes it is  $\gamma \in [0,1[$

# Basics of RL

homwwork look it hup

Example: One day of a student

- Given is the episode with

$$\text{Class1} \rightarrow \text{Study} \rightarrow \text{Class2} \rightarrow \text{Study} \rightarrow \\ \text{Class3} \rightarrow \text{Pub} \rightarrow \text{Class3} \rightarrow \text{Study} \rightarrow \\ \text{Sleep}$$

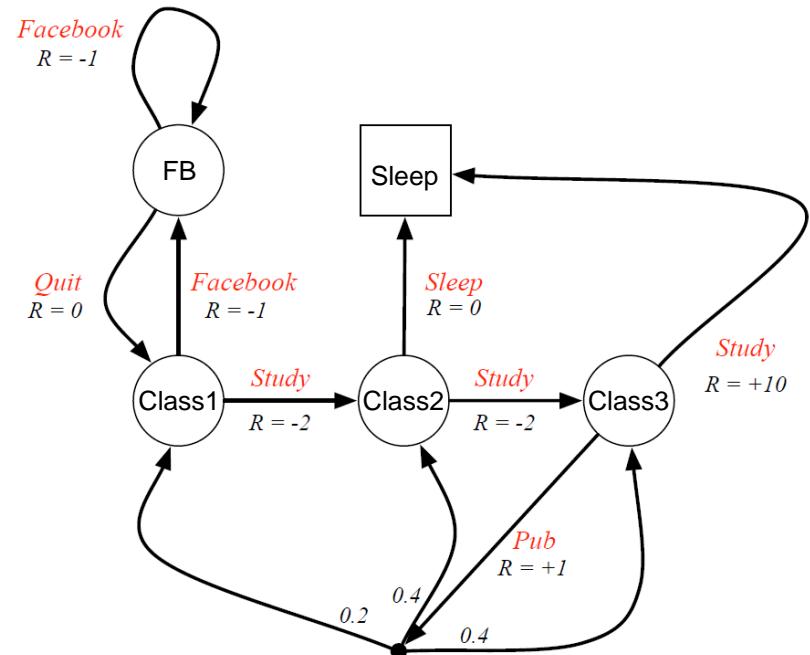
- With  $\gamma = 1$  the return of each state of the episode becomes

$$G_0(\text{Class1}) = -2 - 2 + 1 + 10 = 7$$

$$G_1(\text{Class2}) = -2 + 1 + 10 = 9$$

$$G_2(\text{Class3}) = +1 + 10 = 11$$

$$G_3(\text{Class3}) = +10 = 10$$



<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

- With  $\gamma = 0.9$  the return of each state of the episode becomes

$$G_0(\text{Class1}) = -2 - 2 \cdot 0.9 + 1 \cdot 0.9^2 + 10 \cdot 0.9^3 = 4.3$$

$$G_1(\text{Class2}) = -2 + 1 \cdot 0.9 + 10 \cdot 0.9^2 = 7$$

$$G_2(\text{Class3}) = +1 + 10 \cdot 0.9 = 10$$

$$G_3(\text{Class3}) = +10 = 10$$

Task: What is the return of each state for the given episode for  $\gamma = 0.8$ ?

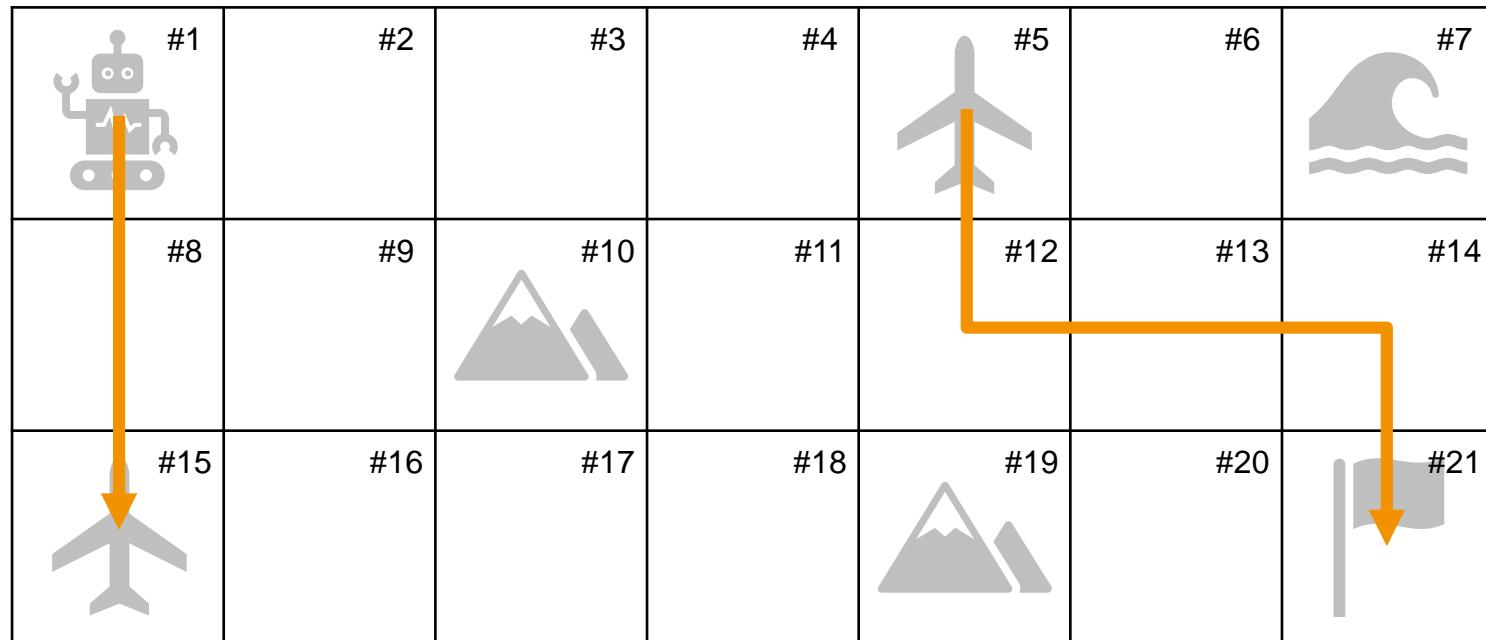
Task: You have an episode with the following rewards:

$$R_1 = -3, R_2 = 1, R_3 = 2, R_4 = -2, R_5 = 6$$

What are  $G_0 \dots G_5$  for  $\gamma = 1$  resp.  $\gamma = 0.8$ ?

# Basics of RL

Task: You are given the following robot maze episode. What is the return of every state if the reward after each action is -1 and the discount factor  $\gamma = 0.8$ ?



## Goal of reinforcement learning (informal)

Find an optimal policy that maximizes the return for all states

# Basics of RL

Task: Find an optimal policy that maximizes the return for all states for the robot maze (without airports)

- -1 reward for every action, -100 when entering the waves
- Discount factor  $\gamma = 0.9$

#1	#2	#3	#4	#5	#6		#7	
#8	#9		#10	#11	#12	#13	#14	
#15	#16	#17	#18		#19	#20		#21

# Basics of RL

Task: Find an optimal policy that maximizes the return for all states for the robot maze (with airports)

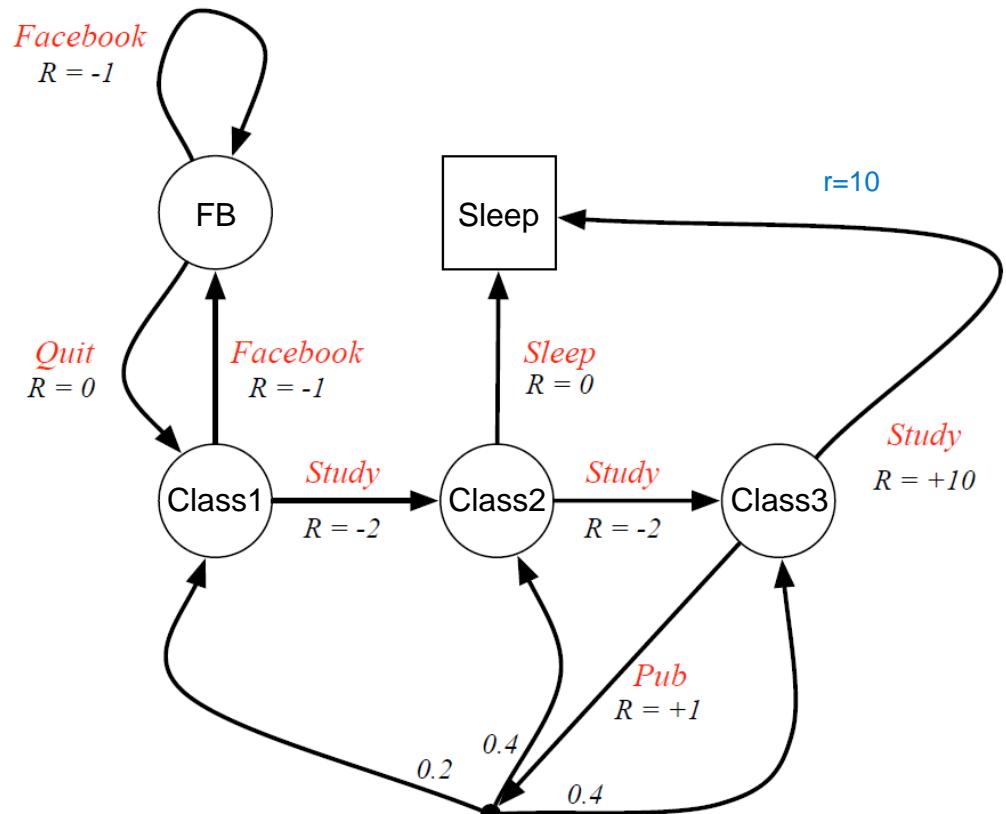
- -1 reward for every action, -100 when entering the waves
- Discount factor  $\gamma = 0.9$

#1	#2	#3	#4		#5	#6	#7 
#8	#9	#10 	#11	#12	#13	#14	
	#15	#16	#17	#18		#20	#21 

# Basics of RL

Task: Find an optimal policy that maximizes the return for all states for "One day of a student"

- Discount factor  $\gamma = 0.9$



<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Basics of RL

## Brief summary

- An agent interacts with its environment by executing actions and receiving rewards
- The return is the discounted cumulative reward
- A policy describes the behaviour of the agent (which action to pick in each state)
- The goal of RL is to find a policy that maximizes the agent's return for all states

## Next chapters

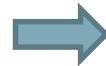
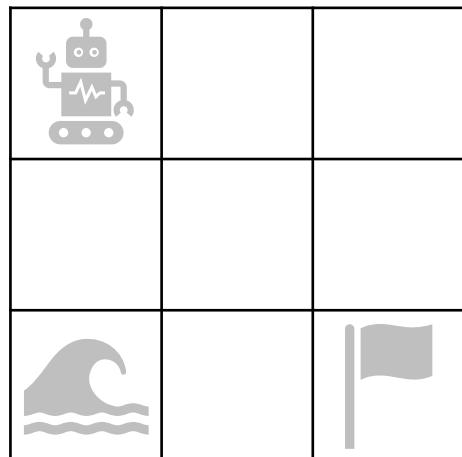
What follows are two conceptually different methods that have the same goal (to find a policy that maximizes the return for all states)

- Value-based methods
- Policy-based methods (aka policy gradients)

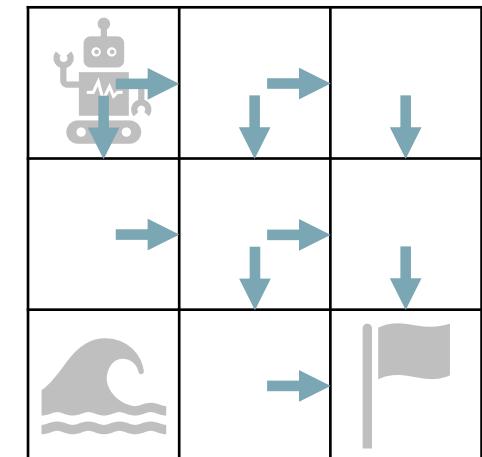
# Basics of RL

Value-based methods first create a value function for every state/action that tells us how good/bad it is to be in this state/take this action. From this value function the optimal policy can be derived

Simplified example (with wave = robot drowns, -100 reward and episode finished)

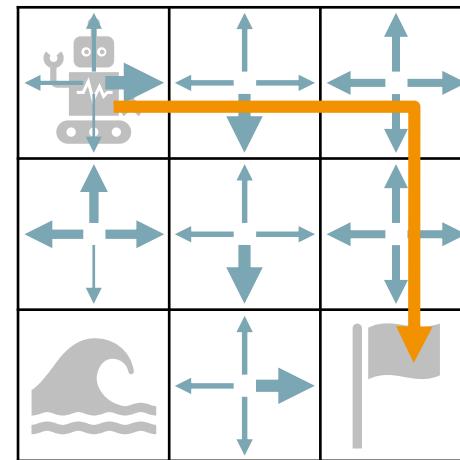
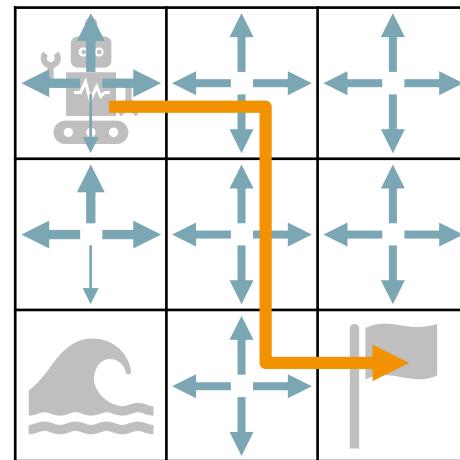
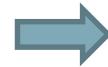
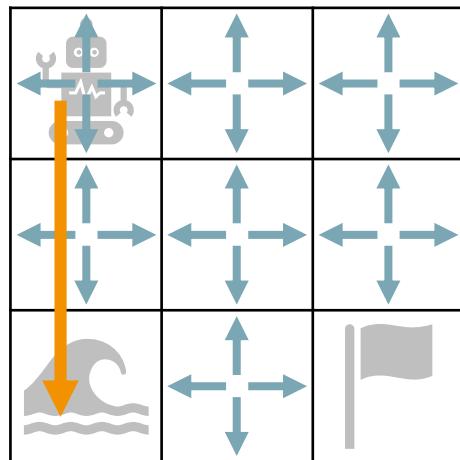


Robot (-4)	-3	-2
-3	-2	-1
(-100)	-1	(0)



# Basics of RL

Policy gradients iteratively improve an initially bad policy, thus eventually obtaining an optimal policy

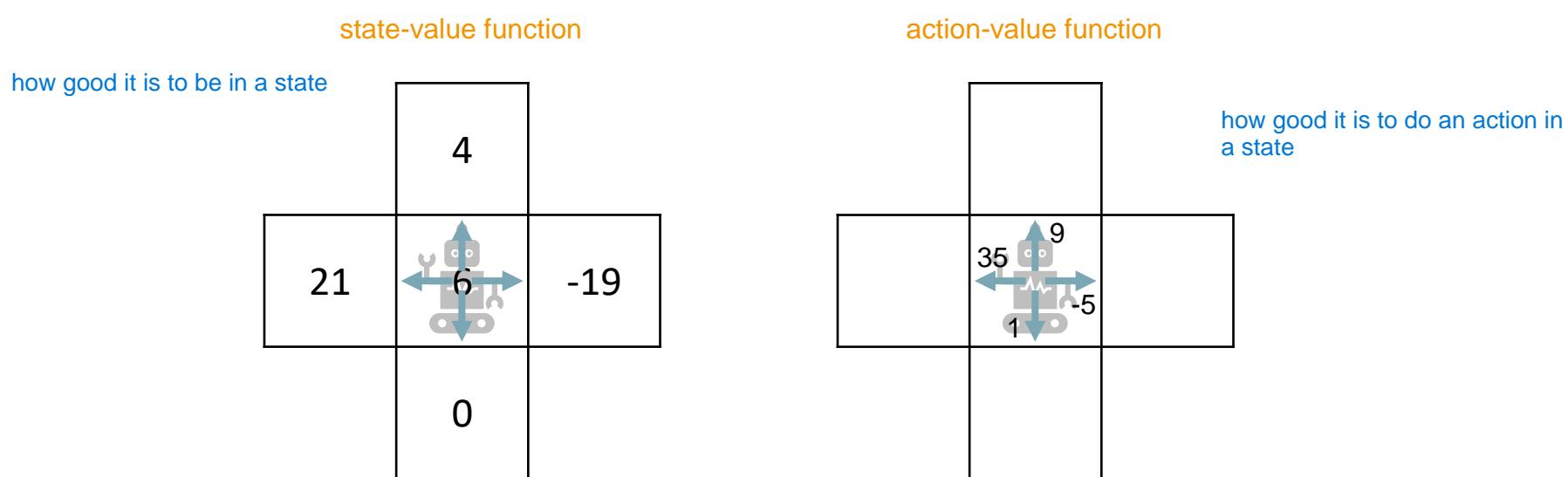


# Kahoot!

# Basics of value-based RL

# Basics of value-based RL

Idea of value-based RL: Create a function to measure the quality of each state (state-value function) or action (action-value function). If the agent has to decide, he chooses the action leading to the state with the highest value resp. the action with the highest value



Task: One of the two presented functions is more useful than the other. Which one and why?

# Basics of value-based RL

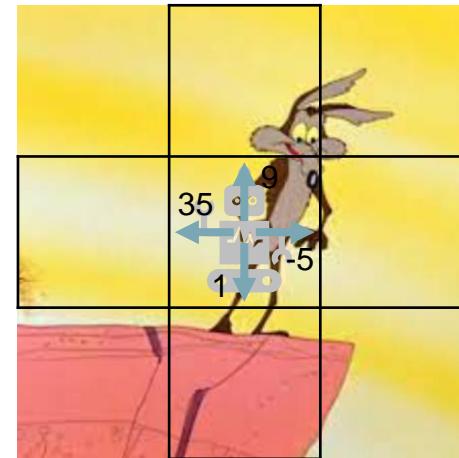
Action-value function is more useful than state-value if the next state after each action is not known. This is critical for

- Stochastic transitions
- "Dangerous" states

state-value function



action-value function



# Basics of value-based RL

## State-value function (V-function)

- State-value function  $V_\pi(s)$  calculates the expected return  $E(G_t)$  for being in state  $s$  while following the policy  $\pi$

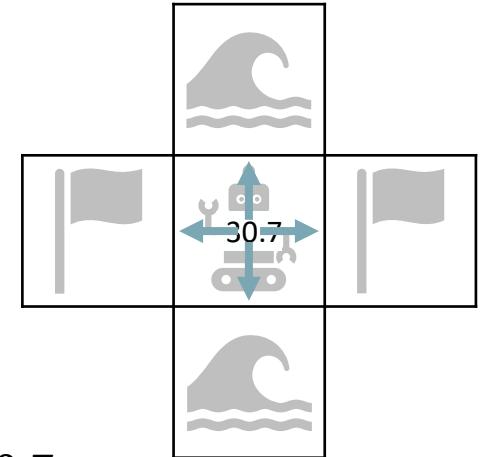
$$V_\pi(s) = E_\pi [ G_t \mid S_t = s ]$$

- State-value function tells us how good it is to be in a certain state
- The individual values of  $V_\pi(s)$  are called **V-values**

## Example

- Robot stands in the middle
- If he moves up/down, he gets a reward of -100 and the episode is over. If he moves left/right, he gets a reward of -1 and the episode is over.
- The probability (policy) of moving u/d/l/r is 0.1/0.2/0.5/0.2.
- Then the state-value for the state in the middle is

$$V_\pi(s) = -100 \cdot 0.1 - 100 \cdot 0.2 - 1 \cdot 0.5 - 1 \cdot 0.2 = -30.7$$



# Basics of value-based RL

## Action-value function (Q-function)

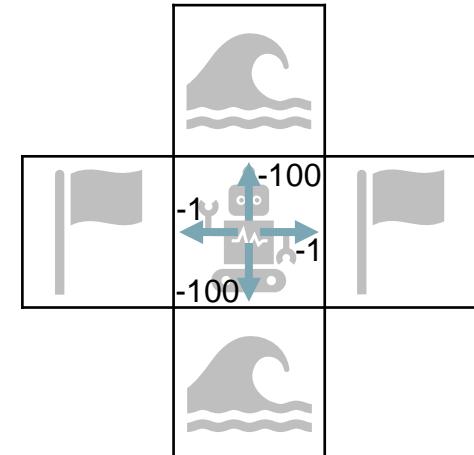
- Action-value function  $Q_\pi(s, a)$  calculates the expected return  $E(G_t)$  for being in state  $s$  and taking action  $a$  while following the policy  $\pi$

$$Q_\pi(s, a) = E_\pi [ G_t \mid S_t = s, A_t = a ]$$

- Action-value function tells us how good it is to take an action in a certain state
- The individual values of  $Q_\pi(s, a)$  are called **Q-values**

## Example

- Robot stands in the middle
- If he moves up/down, he gets a reward of -100 and the episode is over. If he moves left/right, he gets a reward of -1 and the episode is over.
- The probability (policy) of moving u/d/l/r is 0.1/0.2/0.5/0.2.
- Then the action-values are as shown (trivial to calculate)

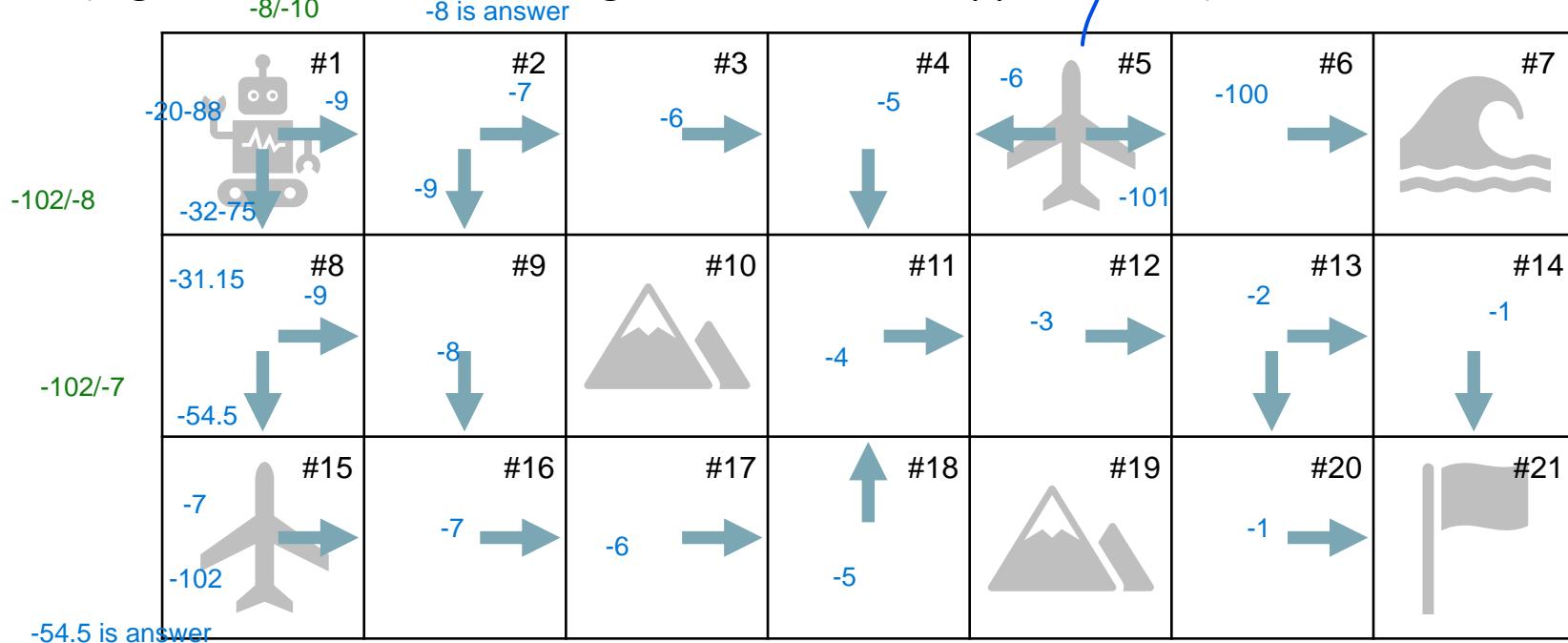


# Basics of value-based RL

the value function of being in state 5 is:  
 $-6 \cdot 0.5 - 101 \cdot 0.5 = -53.5$

Task: What are the state-values for the given policy?

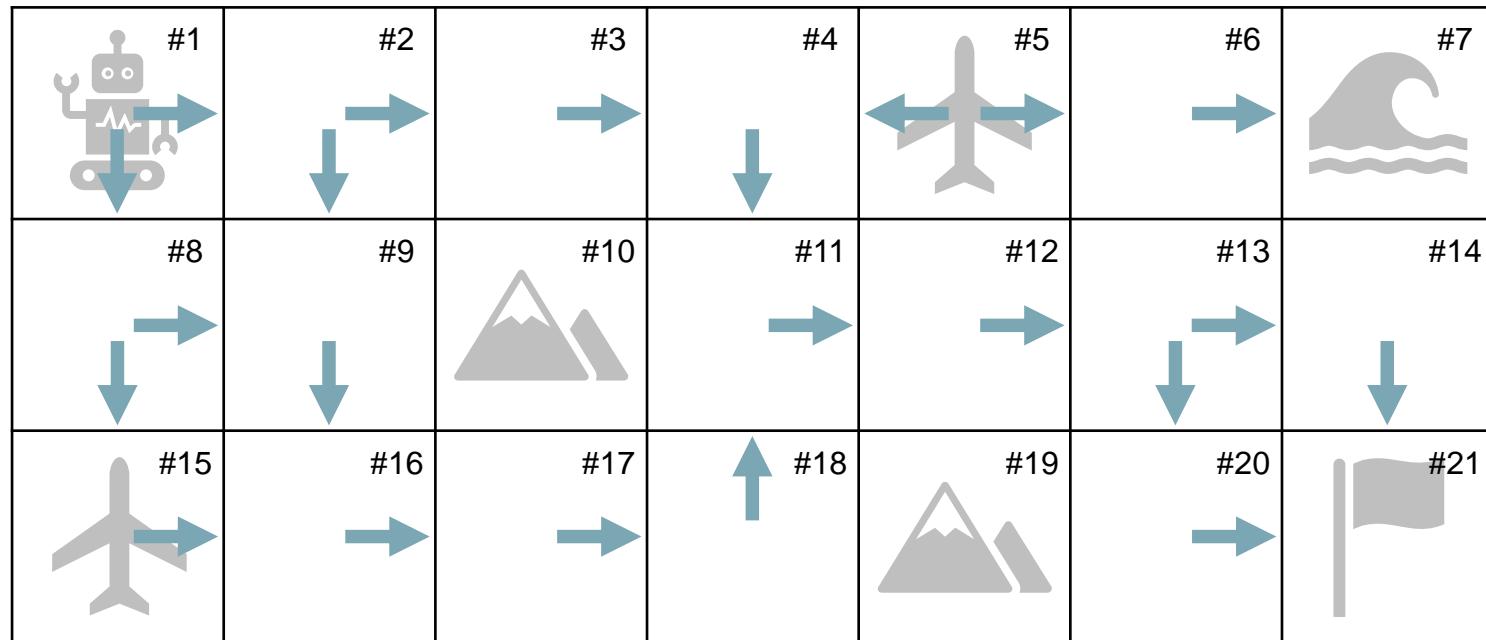
- -1 reward for every action, -100 when entering the waves
- Discount factor  $\gamma = 1$
- Equal probability of taking any action if there are two or more arrows in a cell (e.g. 50/50 chance of moving left/down in the upper left cell)



# Basics of value-based RL

Task: What are the action-values for the given policy?

- -1 reward for every action, -100 when entering the waves
- Discount factor  $\gamma = 1$
- Equal probability of taking any action if there are two or more arrows in a cell (e.g. 50/50 chance of moving left/down in the upper left cell)



# Basics of value-based RL

## Bellman expectation equation for V-function

expected return when being in state 5

$$\begin{aligned}V_{\pi}(s) &= E_{\pi}[G_t | S_t = s] \\&= E_{\pi}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \\&= E_{\pi}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) | S_t = s] \\&= E_{\pi}[R_{t+1} + \gamma G_{t+1} | S_t = s] \quad t \text{ shifted by 1 so we write } g_{t+1} \\&= E_{\pi}[R_{t+1} + \gamma V_{\pi}(S_{t+1}) | S_t = s] \quad \leftarrow \text{possible to calculate } v_{\pi}(S_t) \text{ from } R_{t+1} \text{ and } v_{\pi}(S_{t+1})\end{aligned}$$

## Bellman expectation equation for Q-function

$$Q_{\pi}(s, a) = E_{\pi}[R_{t+1} + \gamma Q_{\pi}(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \quad \leftarrow \text{possible to calculate } q_{\pi}(S_t, A_t) \text{ from } R_{t+1} \text{ and } q_{\pi}(S_{t+1}, A_{t+1})$$

## Bellman expectation equations relating V-function and Q-function

$$\begin{aligned}V_{\pi}(s) &= E_{\pi}[Q_{\pi}(S_t, A_t) | S_t = s] \quad \leftarrow \text{possible to calculate } v_{\pi}(S_t) \text{ from } q_{\pi}(S_t, A_t) \\Q_{\pi}(s, a) &= E_{\pi}[R_{t+1} + \gamma V_{\pi}(S_{t+1}) | S_t = s, A_t = a] \quad \leftarrow \text{possible to calculate } q_{\pi}(S_t, A_t) \text{ from } R_{t+1} \text{ and } v_{\pi}(S_{t+1})\end{aligned}$$

# Basics of value-based RL

The equations on the last slide are extremely powerful:

- Using the original definition of the V-function, that is

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

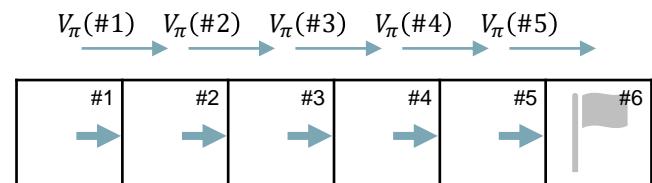
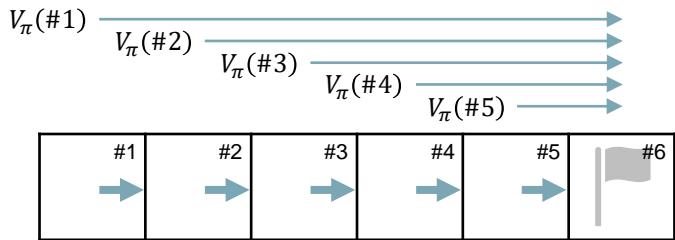
one is forced to evaluate the value  
of every state based on individual episodes

- Using the Bellman expectation equation

$$V_{\pi}(s) = E_{\pi}[R_{t+1} + \gamma V_{\pi}(S_{t+1}) | S_t = s]$$

one can evaluate the value of every state  
based on the successor states and  
the immediate reward (more efficient)

- The same applies for the Q-function



# Basics of value-based RL

Example: Tiny Gridworld ( $\gamma = 0.9$ )

- Original definition of the V-function

$$V_\pi(s) = E_\pi[R_{t+1} + \gamma R_{t+2} + \dots | S_t = s]$$

$$V_\pi(\#5) = -1$$

$$V_\pi(\#4) = -1 - 0.9 \cdot 1 = -1.9$$

$$V_\pi(\#3) = -1 - 0.9 \cdot 1 - 0.9^2 \cdot 1 = -2.71$$

$$V_\pi(\#2) = -1 - 0.9 \cdot 1 - 0.9^2 \cdot 1 - 0.9^3 \cdot 1 \approx -3.44$$

$$V_\pi(\#1) = -1 - 0.9 \cdot 1 - 0.9^2 \cdot 1 - 0.9^3 \cdot 1 - 0.9^4 \cdot 1 \approx -4.10$$



- Using the Bellman expectation equation

$$V_\pi(s) = E_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s]$$

$$V_\pi(\#5) = -1$$

$$V_\pi(\#4) = -1 - 0.9 \cdot 1 = -1.9$$

$$V_\pi(\#3) = -1 - 0.9 \cdot 1.9 = -2.71$$

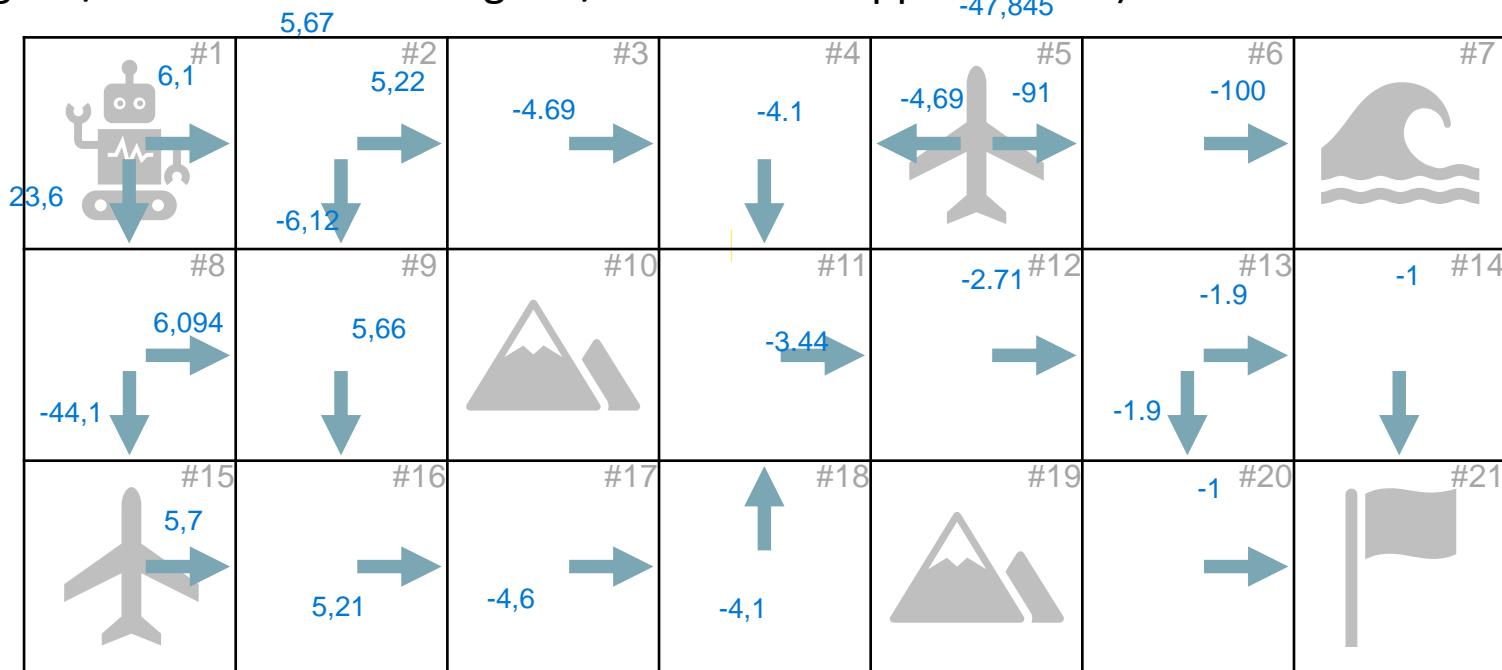
$$V_\pi(\#2) = -1 - 0.9 \cdot 2.71 \approx -3.44$$

$$V_\pi(\#1) = -1 - 0.9 \cdot 3.44 \approx -4.10$$

# Basics of value-based RL

Task: What are the Q-values for the given policy?

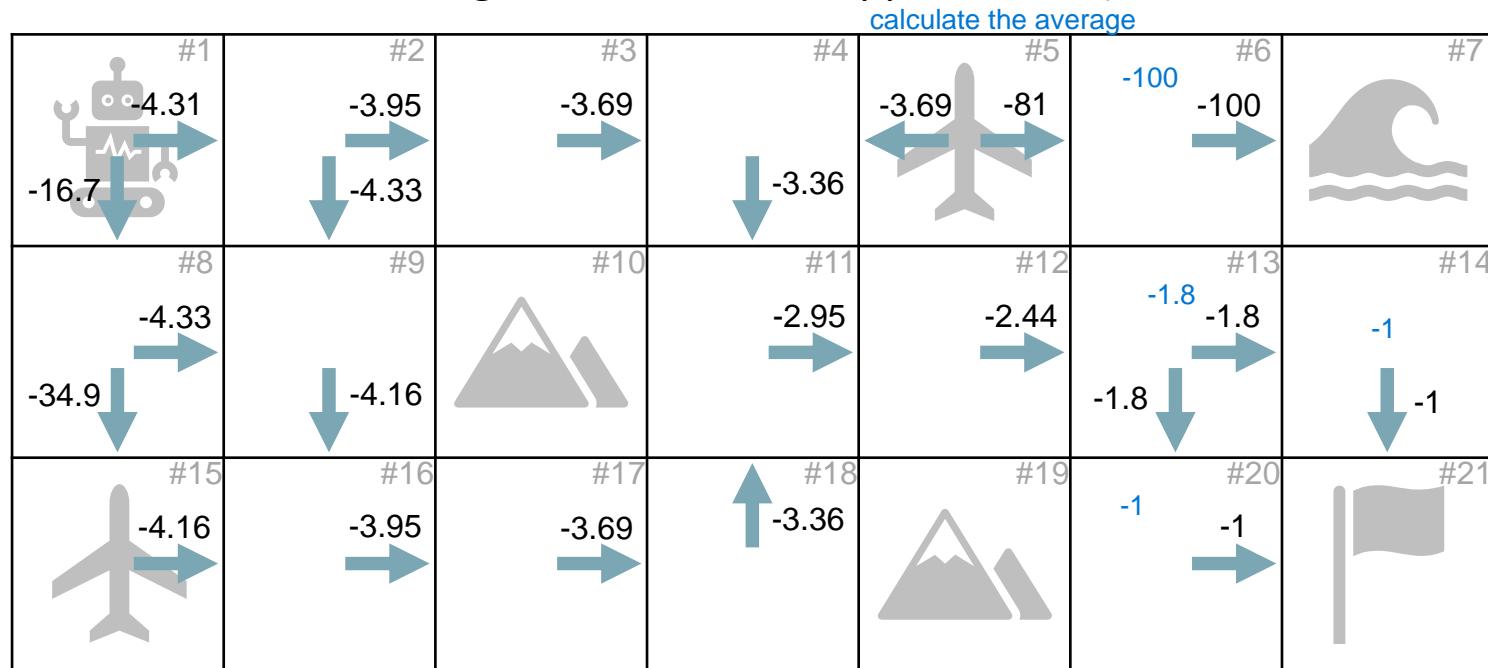
- -1 reward for every action, -100 when entering the waves
- Discount factor  $\gamma = 0.9$
- Equal probability of taking any action if there are two or more arrows in a cell (e.g. 50/50 chance of moving left/down in the upper left cell)



# Basics of value-based RL

Task: Derive the V-values for the given policy from the Q-values

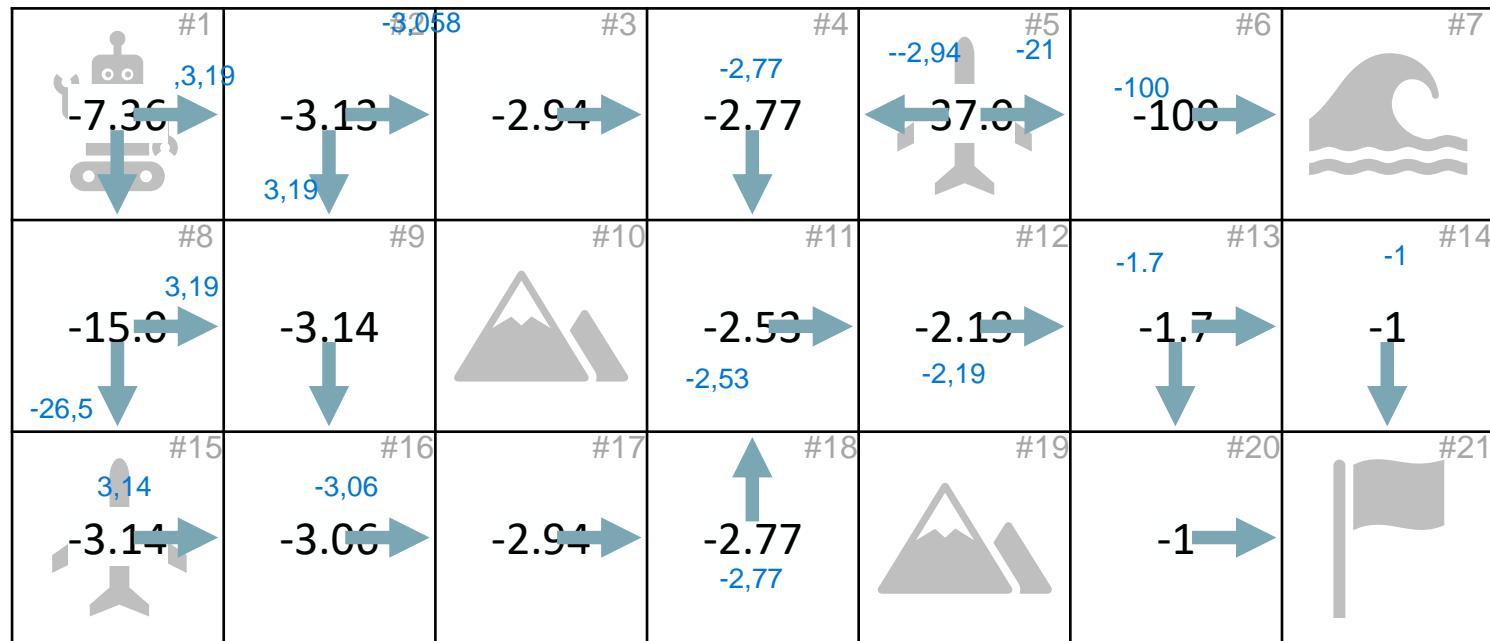
- -1 reward for every action, -100 when entering the waves
- Discount factor  $\gamma = 0.8$
- Equal probability of taking any action if there are two or more arrows in a cell (e.g. 50/50 chance of moving left/down in the upper left cell)



# Basics of value-based RL

Task: Derive the Q-values for the given policy from the V-values

- -1 reward for every action, -100 when entering the waves
- Discount factor  $\gamma = 0.7$
- Equal probability of taking any action if there are two or more arrows in a cell (e.g. 50/50 chance of moving left/down in the upper left cell)



# Basics of value-based RL

Different writing style: With

$$P_{ss'}^a = P(S_{t+1} = s' | S_t = s, A_t = a)$$

$$R_s^a = E[R_{t+1} | S_t = s, A_t = a]$$

as the probability for the next state and the expected immediate reward, the Bellman expectation equations can be rewritten as

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a)$$

$$Q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_\pi(s')$$

Plugging one into each other yields

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_\pi(s') \right)$$

$$Q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \left( \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s', a') \right)$$

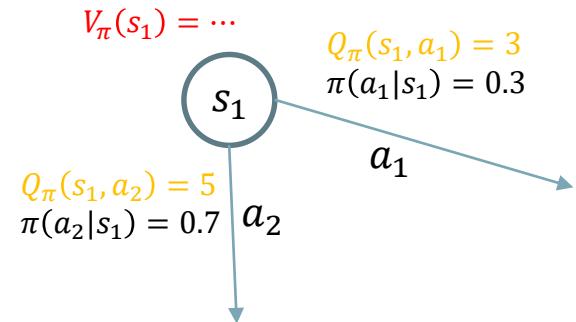
# Basics of value-based RL

Example: MDP

- Calculate the **V-values** of the states from the **Q-values** of the state-action pairs

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a)$$

$$V_\pi(s_1) = 0.7 \cdot 5 + 0.3 \cdot 3 = 4.4$$



# Basics of value-based RL

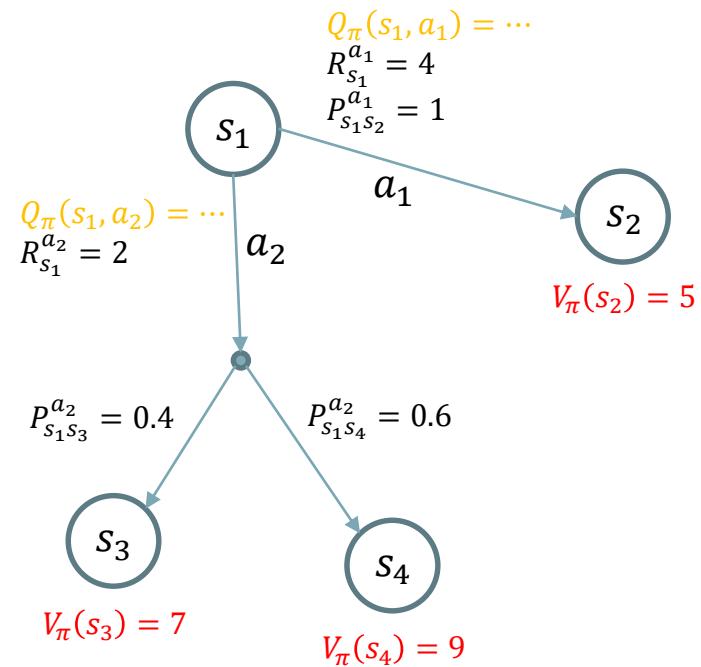
Example: MDP ( $\gamma = 0.9$ )

- Calculate the **Q-values** of the state-action pairs from the **V-values** of the states

$$Q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_\pi(s')$$

$$\begin{aligned} Q_\pi(s_1, a_2) &= 4 + 0.9 \cdot 1 \cdot 5 \\ &= 8.5 \end{aligned}$$

$$\begin{aligned} Q_\pi(s_1, a_1) &= 2 + 0.9 (0.4 \cdot 7 + 0.6 \cdot 9) \\ &= 9.38 \end{aligned}$$



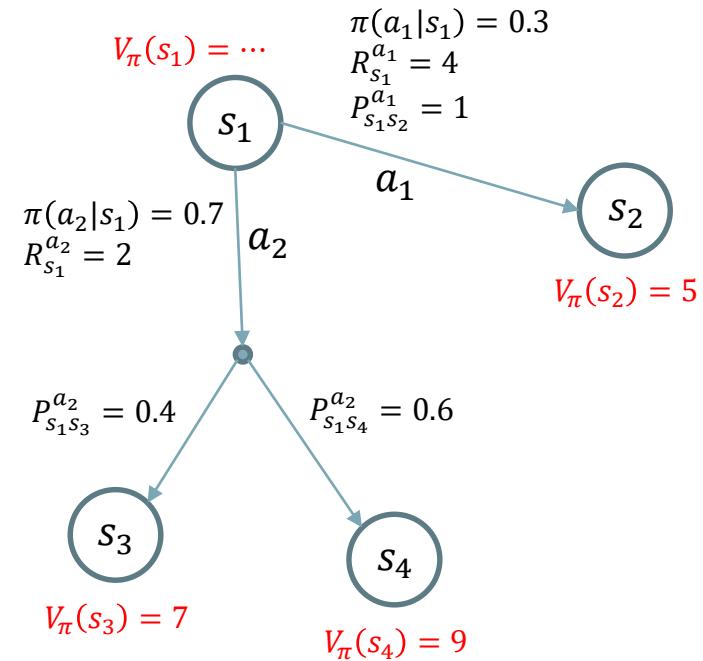
# Basics of value-based RL

Example: MDP ( $\gamma = 0.9$ )

- Calculate the **V-values** of the states from the **V-values** of the following states

$$V_{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_{\pi}(s') \right)$$

$$\begin{aligned} V_{\pi}(s_1) &= 0.3 \cdot (4 + 0.9 \cdot 1 \cdot 5) + \\ &\quad 0.7 \cdot (2 + 0.9 \cdot (0.4 \cdot 7 + 0.6 \cdot 9)) \\ &= 9.116 \end{aligned}$$



# Basics of value-based RL

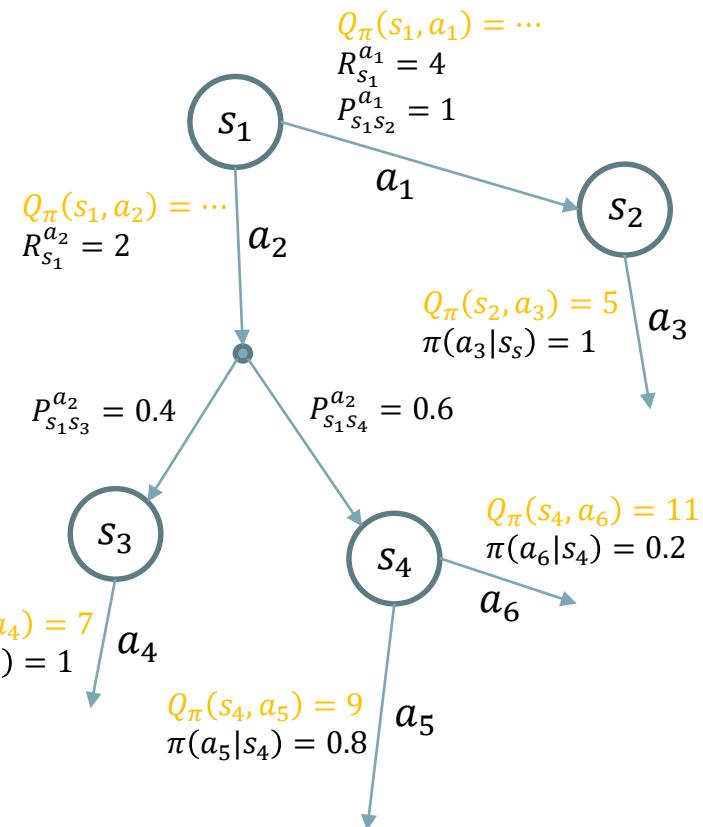
Example: MDP ( $\gamma = 0.9$ )

- Calculate the **Q-values** of the state-action pairs from the **Q-values** of the following state-action pairs

$$Q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \left( \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s', a') \right)$$

$$\begin{aligned} Q_\pi(s_1, a_1) &= 4 + 0.9 \cdot (1 \cdot 1 \cdot 5) \\ &= 8.5 \end{aligned}$$

$$\begin{aligned} Q_\pi(s_1, a_2) &= 2 + \\ &\quad 0.9 \cdot (0.4 \cdot 1 \cdot 7) + \\ &\quad 0.9 \cdot (0.6 \cdot (0.2 \cdot 11 + 0.8 \cdot 9)) \\ &= 9.596 \end{aligned}$$



# Basics of value-based RL

## Optimal value functions

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

$$V_*(s) = \max_{\pi} V_{\pi}(s)$$

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

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

- The optimal value functions specify the best possible performance in the MDP
- A MDP is "solved" once the optimal value functions are known

# Basics of value-based RL

## Optimal policy

- A policy  $\pi$  is better/equal compared to another policy  $\pi'$  if the corresponding state-values of all states are greater/equal

$$\pi \geq \pi' \quad \text{if} \quad V_\pi(s) \geq V_{\pi'}(s), \quad \forall s$$

- For any MDP exists at least one **optimal policy**  $\pi_*$  that is better than or equal to all other policies

$$\pi_* \geq \pi, \quad \forall \pi$$

- All optimal policies achieve the unique, optimal V-function

$$V_{\pi_*}(s) = V_*(s)$$

- All optimal policies achieve the unique, optimal Q-function

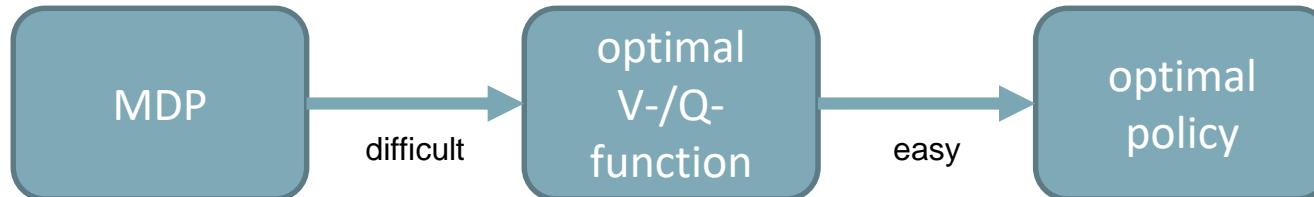
$$Q_{\pi_*}(s, a) = Q_*(s, a)$$

# Basics of value-based RL

- Once the optimal Q-values  $Q_*(s, a)$  are known, an optimal policy can be found by maximizing over  $Q_*(s, a)$  (taking the action with the largest action-value)

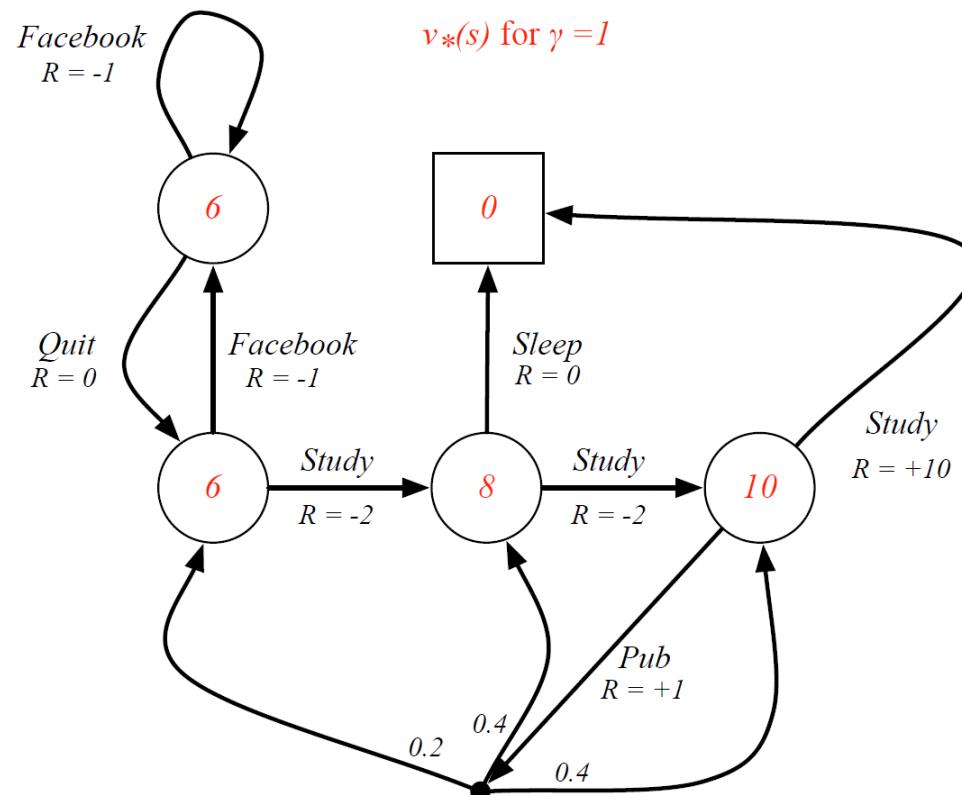
$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q_*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

- There is always a deterministic optimal policy for any MDP
- Overall procedure to find an optimal policy: Derive optimal state/action values from the initial MDP (difficult), then derive optimal policy (easy, see above)



# Basics of value-based RL

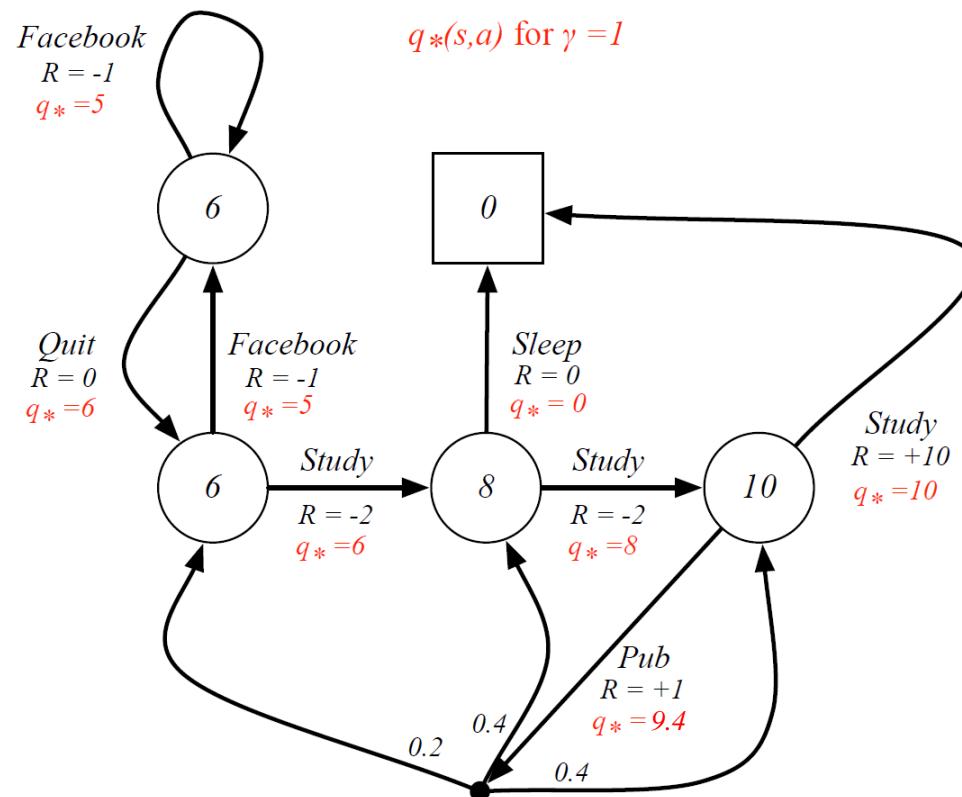
Example: Finding an optimal policy for "One day of a student" based on the optimal V-values



<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Basics of value-based RL

Example: Finding an optimal policy for "One day of a student" based on the optimal Q-values

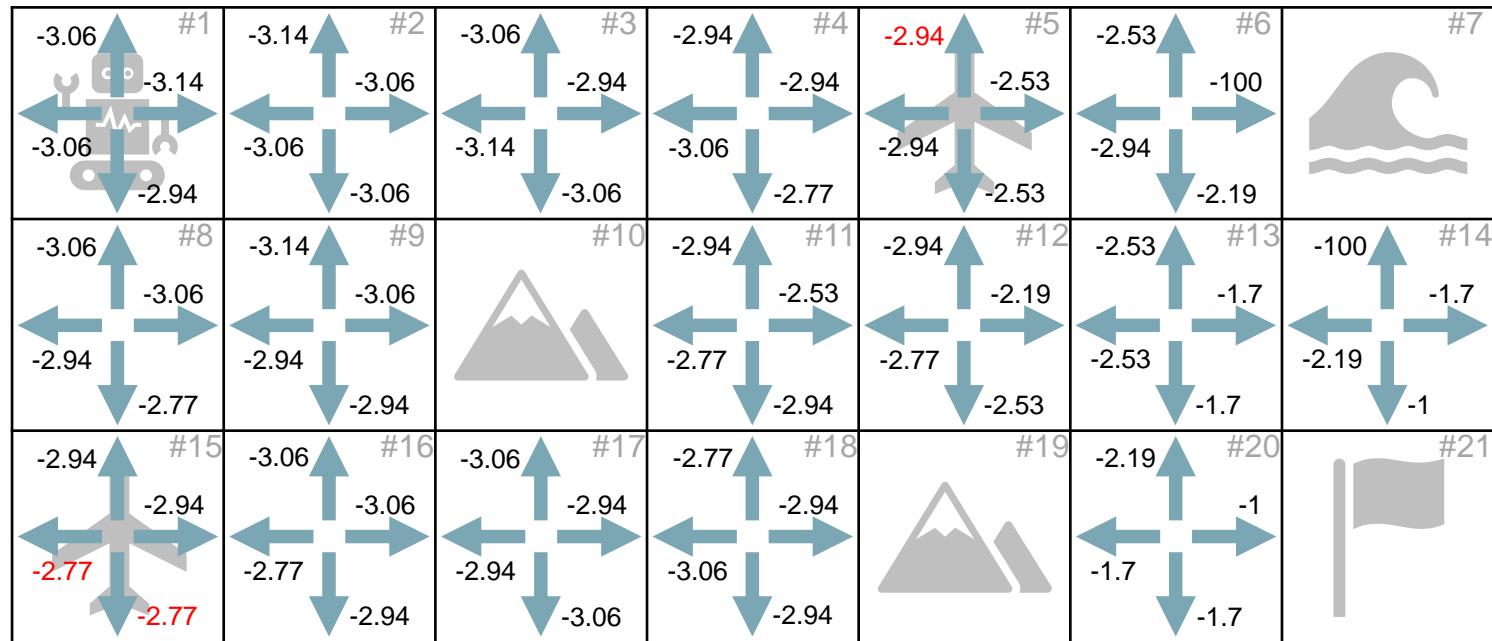


<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Basics of value-based RL

Task: Find an optimal policy for the robot maze given the optimal Q-values below

Task: Look at the red action-values. What do they tell you about the resulting state when executing the corresponding actions?



# Basics of value-based RL

Task: Find an optimal policy for the robot maze given the optimal V-values below  
(-1 reward for every action, -100 when entering the waves, discount factor  $\gamma = 0.7$ )

Task: How can the optimal Q-values (for all actions) be derived from the optimal V-values?

 -2.94	#1 -3.06	#2 -2.94	#3 -2.77	#4 -2.53	#5 -2.19	#6 -1.7	#7 0
#8 -2.77	#9 -2.94	#10 	#11 -2.53	#12 -2.19	#13 -1	#14 -1	
#15 -2.77	#16 -2.77	#17 -2.94	#18 -2.77	#19 	#20 -1	#21 0	

# Basics of value-based RL

**Bellman optimality equation** relating V-function and Q-function

$$V_*(s) = \max_a Q_*(s, a)$$

important

$$Q_*(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_*(s')$$

Plugging one into another yields

$$V_*(s) = \max_a \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_*(s') \right)$$

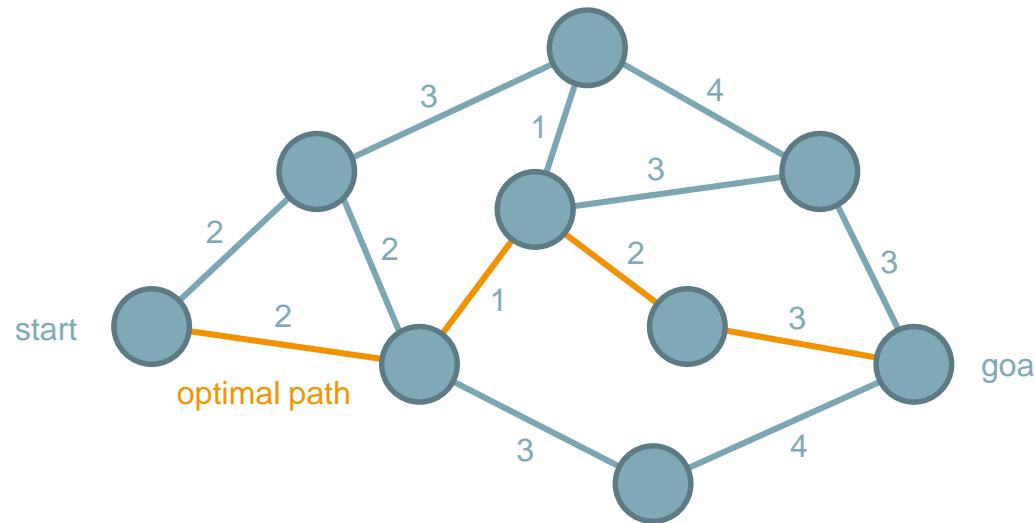
not so important

$$Q_*(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a'} Q_*(s', a')$$

# Basics of value-based RL

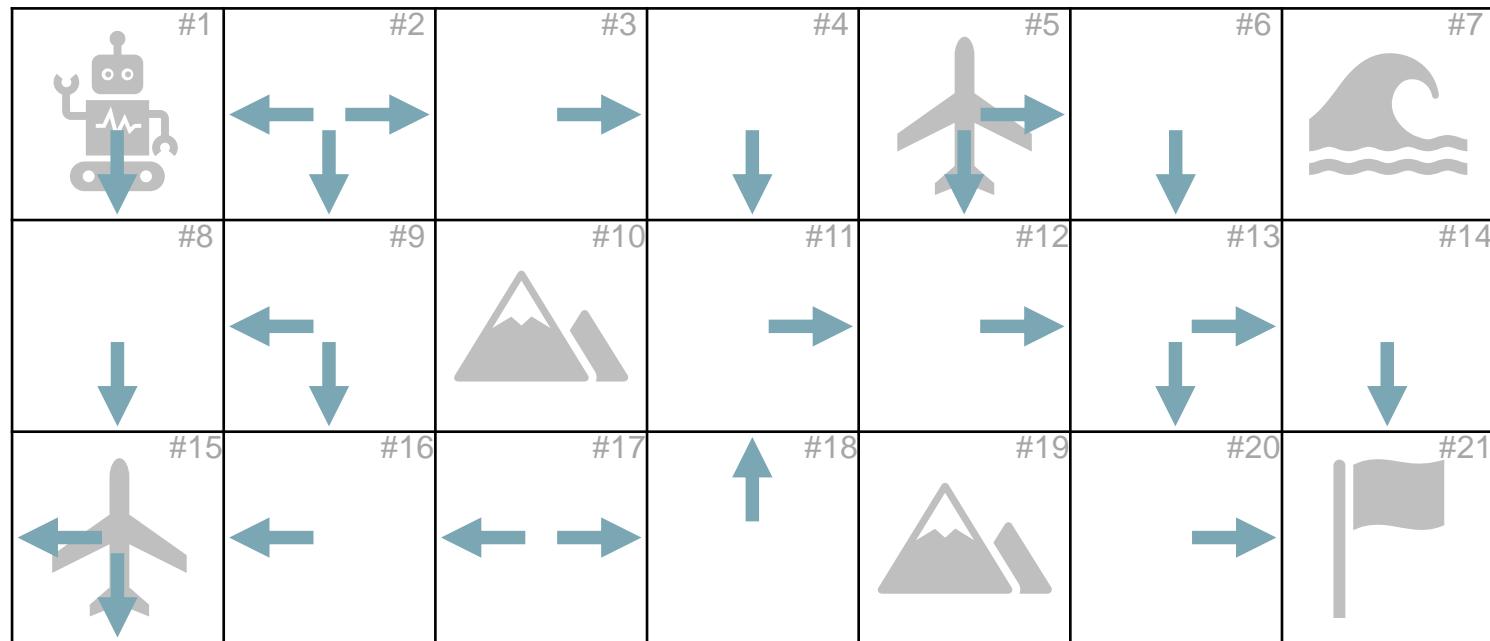
Bellman principle of optimality

- "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." (Bellman, 1957)
- Any global optimal solution consists of local optimal solutions (simplified, generalized)
- Example path planning: Any (global) optimal path between start and goal must consist of local optimal paths



# Basics of value-based RL

- Relation to RL: Any global optimal policy or V-/Q-function must consist of a local optimal policy / V-/Q-function



# Basics of value-based RL

## Brief summary

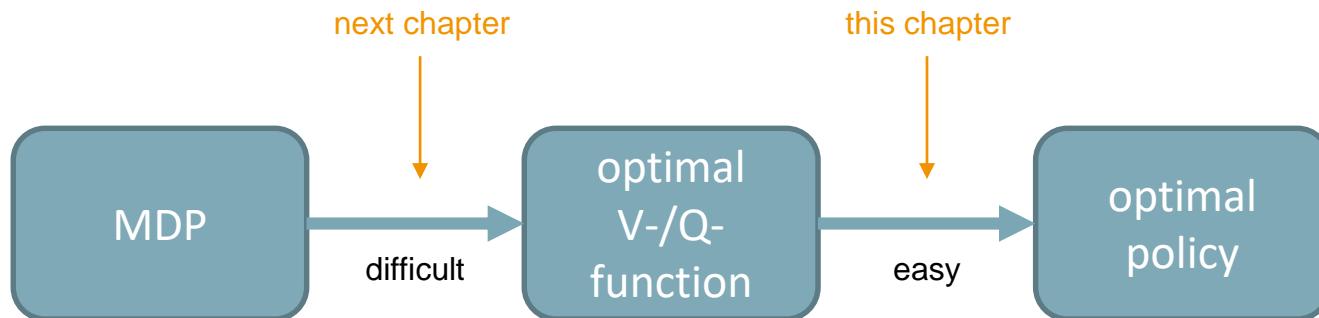
- The V-function  $V_\pi(s)$  evaluates the quality of being in a certain  $s$  state when following the policy  $\pi$
- The Q-function  $Q_\pi(s, a)$  evaluates the quality of taking a certain action  $a$  in state  $s$  when following the policy  $\pi$
- Both functions are closely related through the Bellman expectation equations (if one is known, the other function can be derived easily)
- There exists at least one optimal policy  $\pi_*$  that maximizes the expected reward for all states with associated optimal V-function  $V_*(s)$  and optimal Q-function  $Q_*(s, a)$
- If either  $V_*(s)$  or  $Q_*(s, a)$  is known,  $\pi_*$  can be derived easily
- The goal of RL is to find the optimal policy  $\pi_*$

# Basics of value-based RL

## Next chapter

This chapter covered some basic definitions of value-based RL and explained how an optimal policy  $\pi_*$  can be found if either  $V_*(s)$  or  $Q_*(s, a)$  are given

The next chapter explains how  $V_*(s)$  or  $Q_*(s, a)$  can be calculated



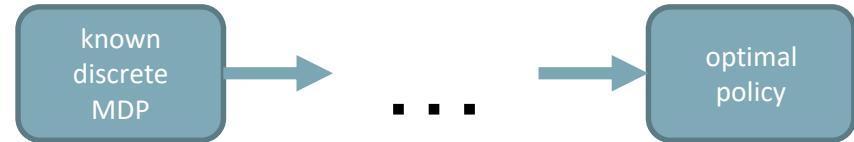
# Kahoot!

# Methods for value-based RL (1/2)

# Methods for value-based RL (1/2)

The methods described in this chapter find an optimal policy for a **known, discrete MDP**

- Known: all states/actions/transitions are known a-priori and can be accessed/executed individually/at any time → no need to record episodes
- Discrete: all states/actions are discrete



Examples for known, discrete MDPs

- Robot maze / One day of a student

Examples for unknown, discrete MDPs

- Uno / Poker

Examples for known, continuous MDPs

- ? (there must exist one)

Examples for unknown, continuous MDPs

- driving

# Methods for value-based RL (1/2)

Four methods are presented

- **Value iteration**

first calculate the optimal V-function,  
then derive an optimal policy from it

- **Policy improvement** (helper method)

improve the policy for given V-function

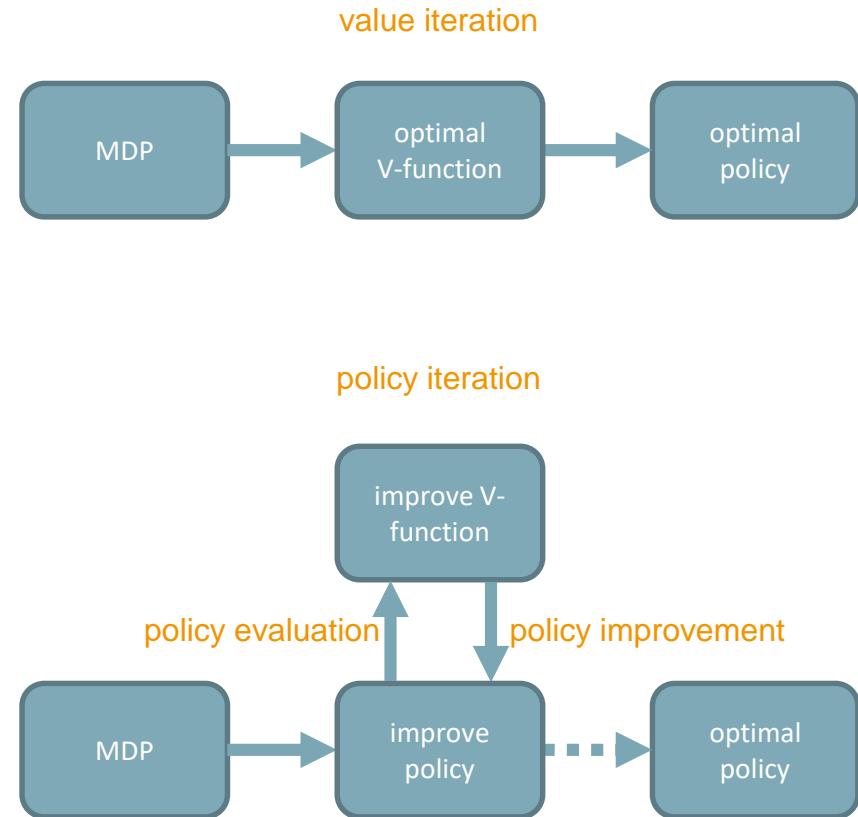
- **Policy evaluation** (helper method)

calculate the V-function for a given policy

- **Policy iteration**

iteratively run policy evaluation and  
policy improvement until convergence  
to an optimal policy

Note that value iteration and policy  
iteration achieve the same result



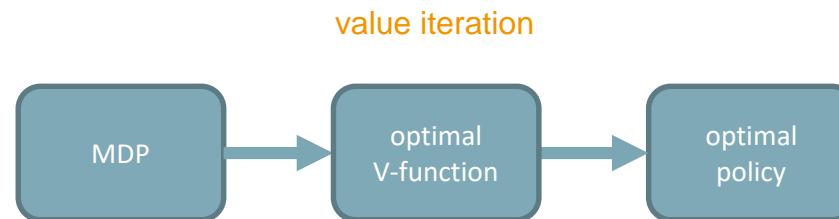
# Methods for value-based RL (1/2)

## Value iteration

- Calculate optimal V-function for a given MDP
- Iterative method: Calculate  $V_{k+1}$  from  $V_k$  until convergence, resulting in  $V_*$

$$V_1 \rightarrow V_2 \rightarrow \dots \rightarrow V_*$$

- After having obtained  $V_*$ , the optimal policy  $\pi_*$  can be derived easily from it
- Schematic procedure:



# Methods for value-based RL (1/2)

## Algorithm

- $V_1$ : initialize all V-values with arbitrary values (e.g. 0)
- Repeat until convergence (then  $V_k = V_*$ )
  - for all states  $s \in \mathcal{S}$  and their successor states  $s'$
  - update  $V_{k+1}(s)$  from  $V_k(s')$  using the Bellman optimality equation:  
Set the new V-value of  $s$  to the highest Q-value of any action  $a$  in state  $s$

$$\begin{aligned} V_{k+1}(s) &= \max_a Q_k(s, a) \\ &= \max_a \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_k(s') \right) \end{aligned}$$

- At the end: Calculate optimal policy  $\pi_*$  from  $v_*$

$$\begin{aligned} \pi_* &= \operatorname{argmax}_a Q_*(s, a) \\ &= \operatorname{argmax}_a \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_*(s') \right) \end{aligned}$$

# Methods for value-based RL (1/2)

## Value iteration in literature

### Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold  $\theta > 0$  determining accuracy of estimation  
 Initialize  $V(s)$ , for all  $s \in S^+$ , arbitrarily except that  $V(\text{terminal}) = 0$

Loop:

$$\Delta \leftarrow 0$$

Loop for each  $s \in S$ :

$$v \leftarrow V(s)$$

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

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

until  $\Delta < \theta$

Output a deterministic policy,  $\pi \approx \pi_*$ , such that

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

Sutton, Barto: Reinforcement Learning

# Methods for value-based RL (1/2)

Example: Value iteration

- 1 reward for every action, discount factor  $\gamma = 1$

Flag			

Problem

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

$V_1$

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

$V_2$

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

$V_3$

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

$V_4$

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

$V_5$

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

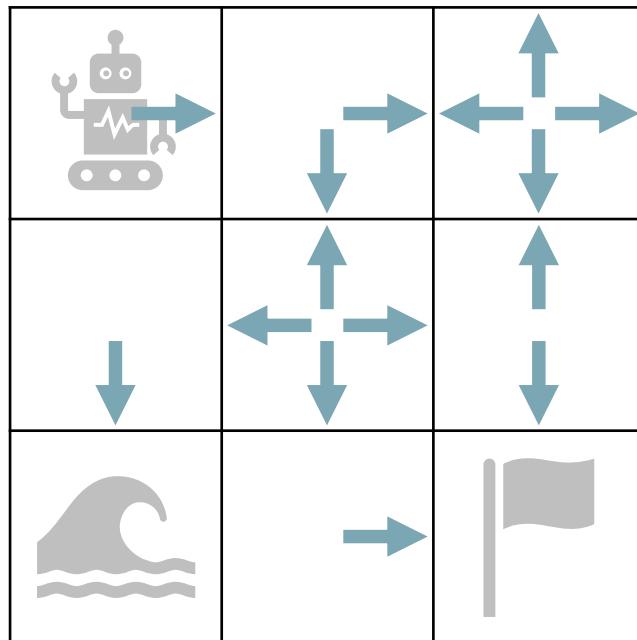
$V_6$

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

$V_7$

# Methods for value-based RL (1/2)

Task: Calculate the resulting policy after running value iteration with three iterations for the robot maze shown below  
(-1 reward for every action, -100 when entering the waves, discount factor  $\gamma = 1$ )



# Methods for value-based RL (1/2)

Example: <https://nowke.github.io/rviz/>



<https://nowke.github.io/rviz/>

# Methods for value-based RL (1/2)

## Policy improvement

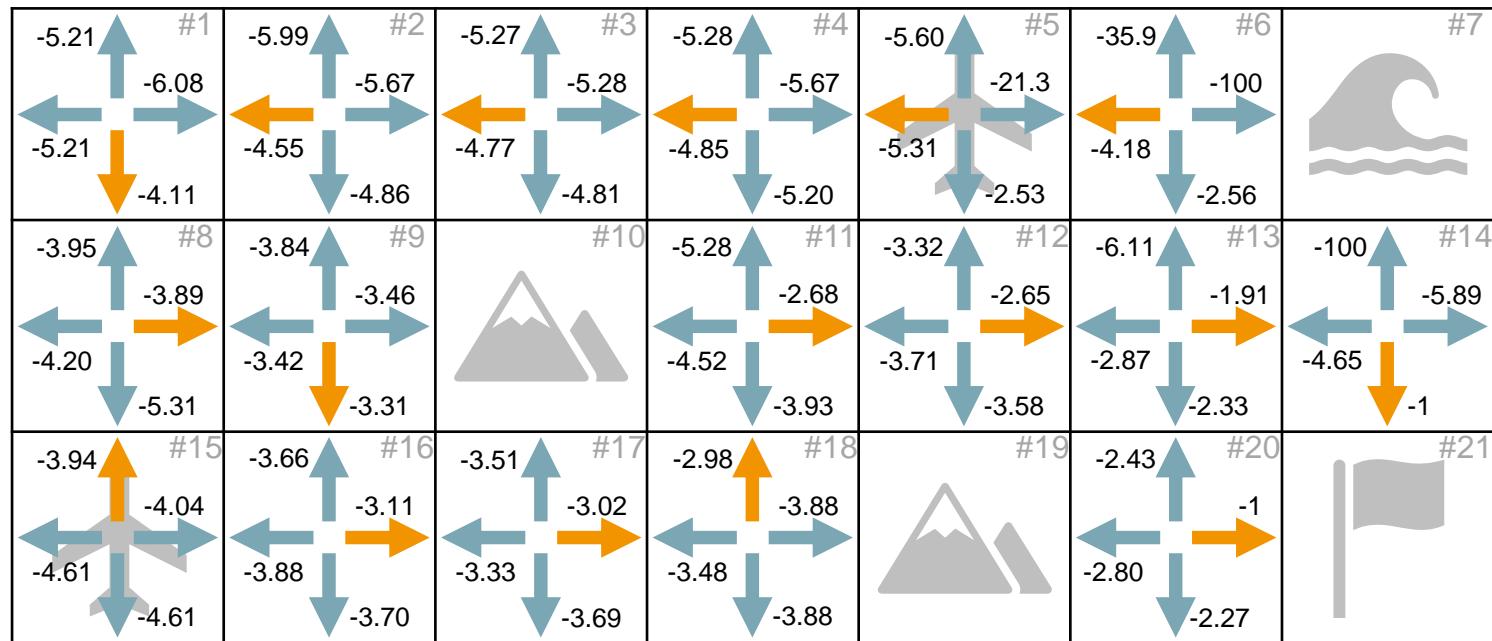
- Improve a policy for a given V-/Q-function
- In every state: Select action with maximum Q-value as new policy  $\pi'$

$$\begin{aligned}
 \pi'(s) &= \underset{a}{\operatorname{argmax}} Q_{\pi}(s, a) \\
 &= \underset{a}{\operatorname{argmax}} E_{\pi}[R_{t+1} + \gamma V_{\pi}(S_{t+1}) | S_t = s] \\
 &= \underset{a}{\operatorname{argmax}} \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_{\pi}(s') \right)
 \end{aligned}$$

# Methods for value-based RL (1/2)

Example: Policy improvement

- Orange: new policy based on V-values



# Methods for value-based RL (1/2)

## Policy evaluation

- Calculate V-function for a given policy
- Iterative method: Calculate  $V_{k+1}$  from  $V_k$  until convergence, resulting in  $V_\pi$

$$V_1 \rightarrow V_2 \rightarrow \dots \rightarrow V_\pi$$

## Algorithm

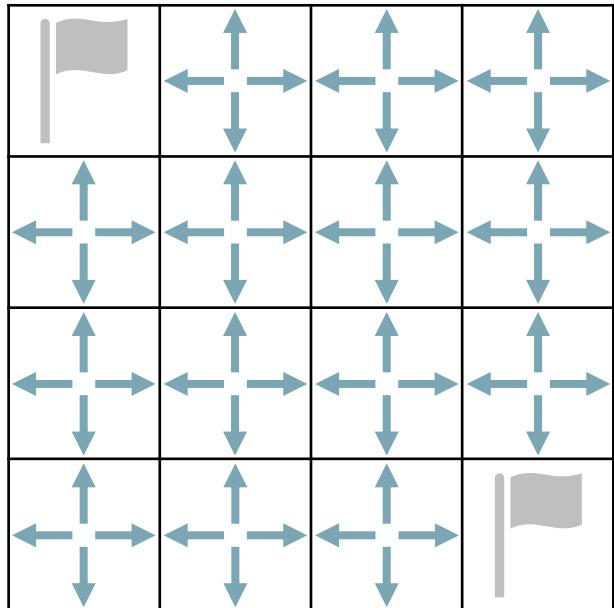
- $V_1$ : initialize all V-values with arbitrary values (usually 0)
- Repeat until convergence
  - for all states  $s \in \mathcal{S}$  and their successor states  $s'$
  - update  $V_{k+1}(s)$  from  $V_k(s')$  using the Bellman expectation equation:  
Set the new V-value of  $s$  to the expected Q-values when following the policy  $\pi$

$$\begin{aligned} V_{k+1}(s) &= E_\pi[Q_k(S_t, A_t)|S_t = s] \\ &= E_\pi[R_{t+1} + \gamma V_k(s')|S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_k(s') \right) \end{aligned}$$

# Methods for value-based RL (1/2)

## Example: Policy evaluation

- Random policy (25% chance to move up/down/left/right)
- -1 reward for every action, discount factor  $\gamma = 1$



$k = 0$

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

$k = 1$

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

$k = 2$

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

$$-2.9 \approx 0.25 \cdot (-1 - 1.7) + 0.25 \cdot (-1 - 1.7) + 0.25 \cdot (-1 - 2.0) + 0.25 \cdot (-1 - 2.0)$$

$k = 3$

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

$k = 10$

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

$k = \infty$

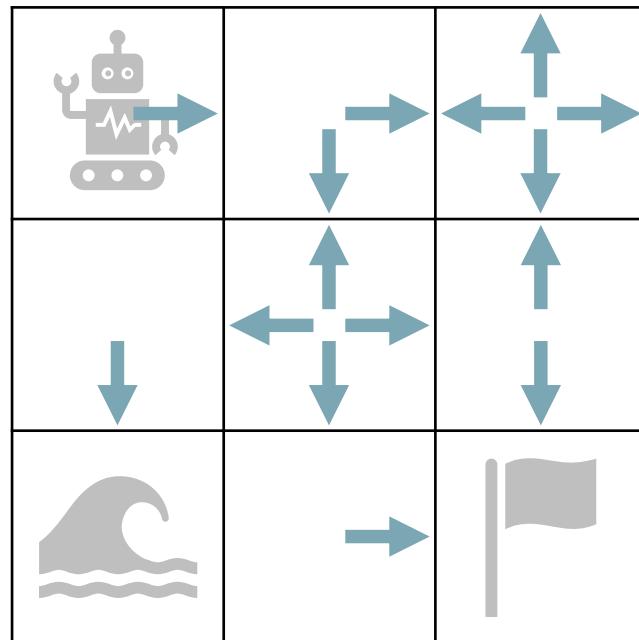
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Sutton, Barto: Reinforcement Learning

# Methods for value-based RL (1/2)

Task: Calculate the first three policy evaluation iterations for the robot maze below (-1 reward for every action, -100 when entering the waves, discount factor  $\gamma = 1$ )

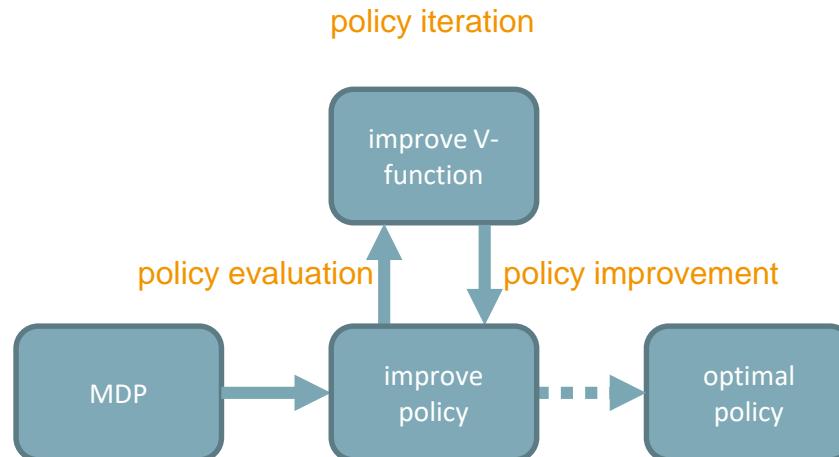
Task: Under which circumstances does policy evaluation converge after a known number of iterations?



# Methods for value-based RL (1/2)

## Policy iteration

- Combination of policy evaluation and policy improvement
- Calculate value function for given policy, then improve policy, then calculate value function for given policy, then improve policy, then calculate value function for given policy, then ...
- Guaranteed to converge to optimal value functions  $V_*$  /  $Q_*$  and optimal policy  $\pi_*$
- Schematic procedure:



# Methods for value-based RL (1/2)

## Policy iteration in literature

### Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

#### 1. Initialization

$V(s) \in \mathbb{R}$  and  $\pi(s) \in \mathcal{A}(s)$  arbitrarily for all  $s \in \mathcal{S}$ ;  $V(\text{terminal}) \doteq 0$

#### 2. Policy Evaluation

Loop:

$$\Delta \leftarrow 0$$

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

$$v \leftarrow V(s)$$

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

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

until  $\Delta < \theta$  (a small positive number determining the accuracy of estimation)

#### 3. Policy Improvement

*policy-stable*  $\leftarrow$  true

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

$$\text{old-action} \leftarrow \pi(s)$$

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

If  $\text{old-action} \neq \pi(s)$ , then *policy-stable*  $\leftarrow$  false

If *policy-stable*, then stop and return  $V \approx v_*$  and  $\pi \approx \pi_*$ ; else go to 2

Sutton, Barto: Reinforcement Learning

# Methods for value-based RL (1/2)

Example: <https://nowke.github.io/rviz/>



# Methods for value-based RL (1/2)

## Comparison of value iteration and policy iteration (1/2)

Policy Iteration	Value Iteration
<ul style="list-style-type: none"> <li>Starts with a random policy</li> <li>Algorithm is more complex</li> <li>Guaranteed to converge</li> <li>Cheaper to compute</li> <li>Requires few iterations to converge</li> <li>Faster</li> </ul>	<ul style="list-style-type: none"> <li>Starts with a random value function</li> <li>Algorithm is simpler</li> <li>Guaranteed to converge</li> <li>More expensive to compute</li> <li>Requires more iterations to converge</li> <li>Slower</li> </ul>

<https://www.baeldung.com/cs/ml-value-iteration-vs-policy-iteration>

# Methods for value-based RL (1/2)

## Comparison of value iteration and policy iteration (2/2)

- Policy iteration relies on the Bellman expectation equation
- Value iteration relies on the Bellman optimality equation

### Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization  
 $V(s) \in \mathbb{R}$  and  $\pi(s) \in \mathcal{A}(s)$  arbitrarily for all  $s \in \mathcal{S}$ ;  $V(\text{terminal}) \doteq 0$
2. Policy Evaluation  
 Loop:  
 $\Delta \leftarrow 0$   
 Loop for each  $s \in \mathcal{S}$ :  
 $v \leftarrow V(s)$   

$$V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s)) [r + \gamma V(s')] \quad \boxed{\text{Red box}}$$
  
 $\Delta \leftarrow \max(\Delta, |v - V(s)|)$   
 until  $\Delta < \theta$  (a small positive number determining the accuracy of estimation)
3. Policy Improvement  
 $\text{policy-stable} \leftarrow \text{true}$   
 For each  $s \in \mathcal{S}$ :  
 $\text{old-action} \leftarrow \pi(s)$   
 $\pi(s) \leftarrow \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$   
 If  $\text{old-action} \neq \pi(s)$ , then  $\text{policy-stable} \leftarrow \text{false}$   
 If  $\text{policy-stable}$ , then stop and return  $V \approx v_*$  and  $\pi \approx \pi_*$ ; else go to 2

### Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold  $\theta > 0$  determining accuracy of estimation  
 Initialize  $V(s)$ , for all  $s \in \mathcal{S}^+$ , arbitrarily except that  $V(\text{terminal}) = 0$

```

Loop:
|    $\Delta \leftarrow 0$ 
|   Loop for each  $s \in \mathcal{S}$ :
|      $v \leftarrow V(s)$ 
|     
$$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')] \quad \boxed{\text{Red box}}$$

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

Output a deterministic policy,  $\pi \approx \pi_*$ , such that

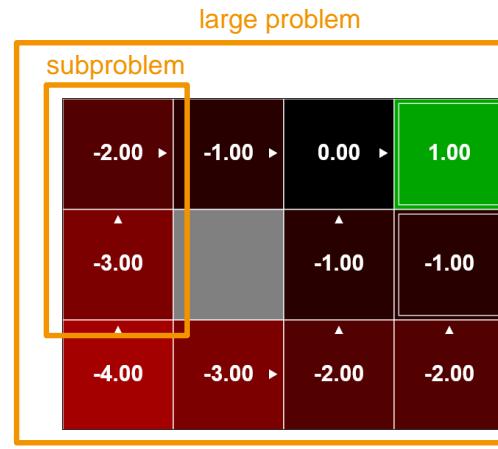
$$\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

  
```

# Methods for value-based RL (1/2)

## Dynamic programming (DP)

- Refers to a class of optimization/programming methods relying on the Bellman principle of optimality (colloquially: Optimal solution of a large problem is also optimal w.r.t. to its subproblems)
- Idea: Break down large problem into smaller subproblems, find optimal solutions for the smaller subproblems and merge them to obtain the overall optimal solution
- Both policy iteration and value iteration are realizations of dynamic programming: Iterate, until optimal solutions to individual subproblems has been obtained Then merge solutions to obtain optimal solution for the entire environment



<https://nowke.github.io/rviz/>

# Methods for value-based RL (1/2)

## Asynchronous dynamic programming

- So far: Synchronous dynamic programming: All V-values are backed up in parallel  
Synchronous backups store two copies of the value function at each iteration in memory during policy /value iteration, e.g.

$$V_{k+1}(s) = \max_a \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_k(s') \right)$$

↑  
array1 in memory    ↑  
  array2 in memory

- In-place dynamic programming: Store only one copy of the value function

$$V_{k+1}(s) = \max_a \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_k(s') \right)$$

←→  
same array in memory

# Methods for value-based RL (1/2)

- Prioritized sweeping: Update states with largest Bellman error first

$$\left| \max_a \left( R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a V_k(s') \right) - V_k(s) \right|$$

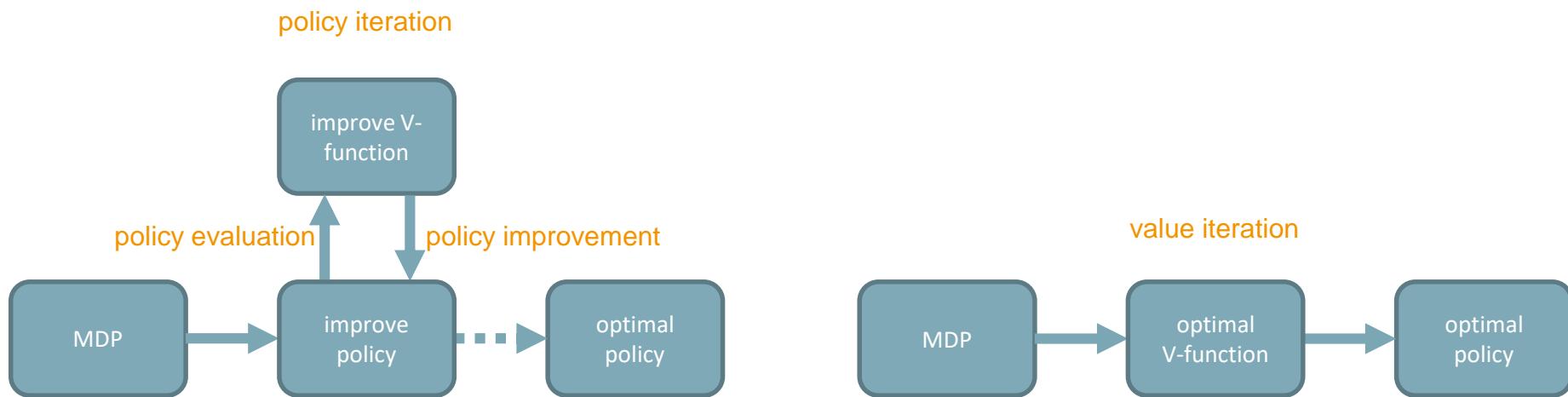
- Real-time dynamic programming: Only update the value function of the states that the agent visits in reality (based on a few sampled episodes)

→ All asynchronous dynamic programming techniques can be used to either reduce the memory consumption and/or computational complexity of policy/value iteration. They are still guaranteed to converge to the optimal policy if all states are eventually selected

# Methods for value-based RL (1/2)

## Brief summary

- This chapter tackled the problem to find an optimal policy for an known, discrete MDP
- Two methods are presented: Policy iteration with the two submethods policy evaluation / policy improvement and value iteration
- Both methods converge to the optimal policy



# Methods for value-based RL (1/2)

## Next chapter

Policy/value iteration is computationally demanding if the state space is large (e.g. chess has over  $10^{100}$  states). Hence the methods described in this chapter are only applicable if the environment is not too large and the entire MDP dynamics are known.

The next chapters use sampled episodes/backups in order to obtain the optimal policy, thus the cost of a backup stays constant and does not grow with the number of states

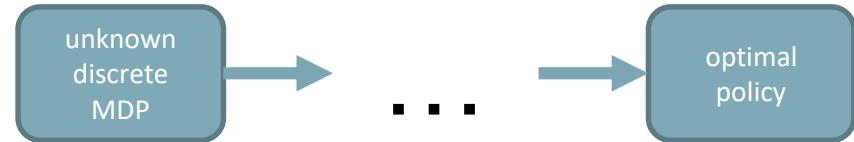
# Kahoot!

# Methods for value-based RL (2/2)

# Methods for value-based RL (2/2)

The methods described in this chapter find an optimal policy for an **unknown, discrete MDP**

- Unknown: states/actions/transitions are not known a-priori  
→ agent must rely on trial-and-error (run episodes and see how well it goes)
- Discrete: all states/actions are discrete



Examples of unknown MDPs: Partially observable MDPs (POMDP)

- Agent does not have full information of the environment state
- Uno / Poker / robot with camera / driving

Three methods are presented

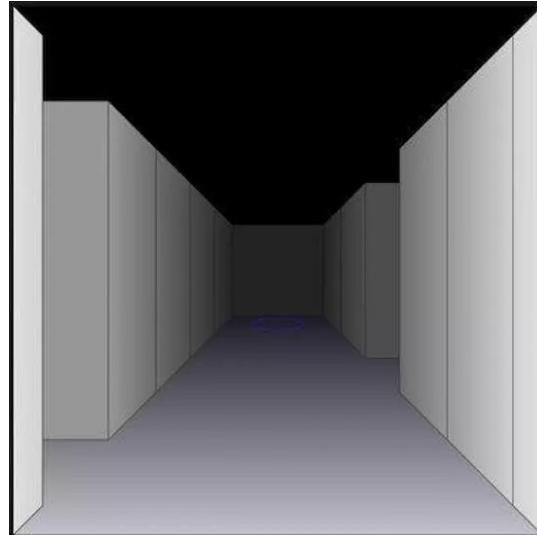
- Monte-Carlo evaluation
- SARSA
- Q-learning

# Methods for value-based RL (2/2)

## Known vs. unknown environments

- Unknown environments are much more common than known environments
- "Solving" an unknown environment through trial-and-error (and improvement) is the core domain of RL

unknown environment:  
have to learn the optimal path  
based on recorded episodes



known environment:  
no need to record episodes



# Methods for value-based RL (2/2)

- known environment: Loop over all states

```
for (all states):  
    ...
```

- unknown environment: Loop over all visited states / all states of the last episode

```
for (all visited states):  
    ...
```

```
for (all states of last episode):  
    ...
```

# Methods for value-based RL (2/2)

Example: Robot maze

- Episode 1: *left → right → right → wave* (*reward = -100*)
  - Episode 2: *right → down → right → goal* (*reward = 0*)
- Agent: Fine, then I'll follow episode 2 from now on

Example: Star Craft II

- Episode 1: *build base → do nothing* → *loss* (*reward = -1*)
  - Episode 2: *build base → attack enemy* → *win* (*reward = 1*)
- Agent: Fine, then I'll follow episode 2 from now on
- Episode 3: *build base → attack enemy* → *loss* (*reward = -1*)
- Agent: What the fuck?

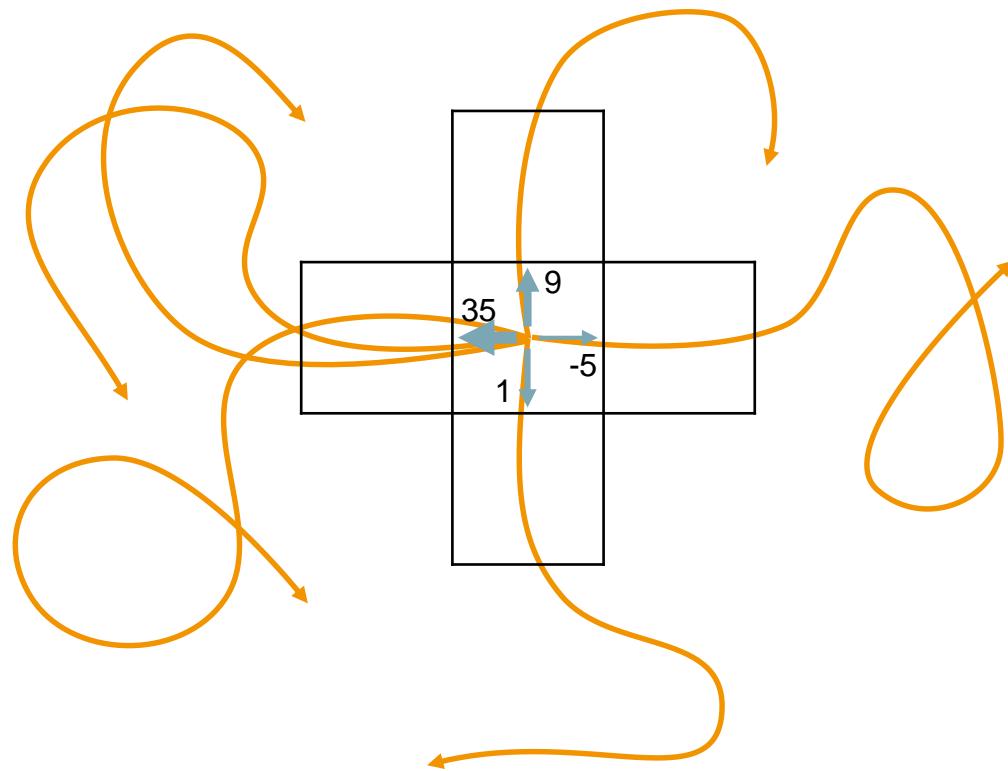
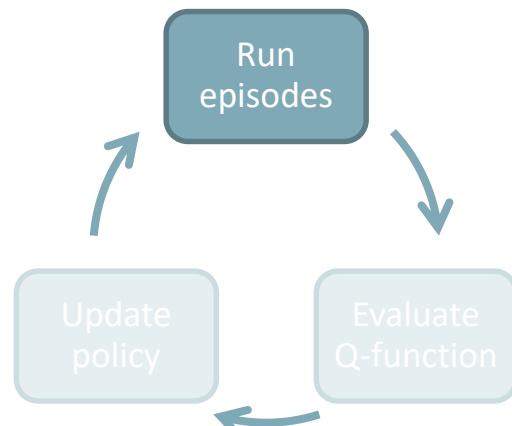


<https://www.derstandard.de/story/2000062499557/googles-kuenstliche-intelligenz-erclaert-starcraft-2-den-krieg>

# Methods for value-based RL (2/2)

General structure of all methods presented in this chapter

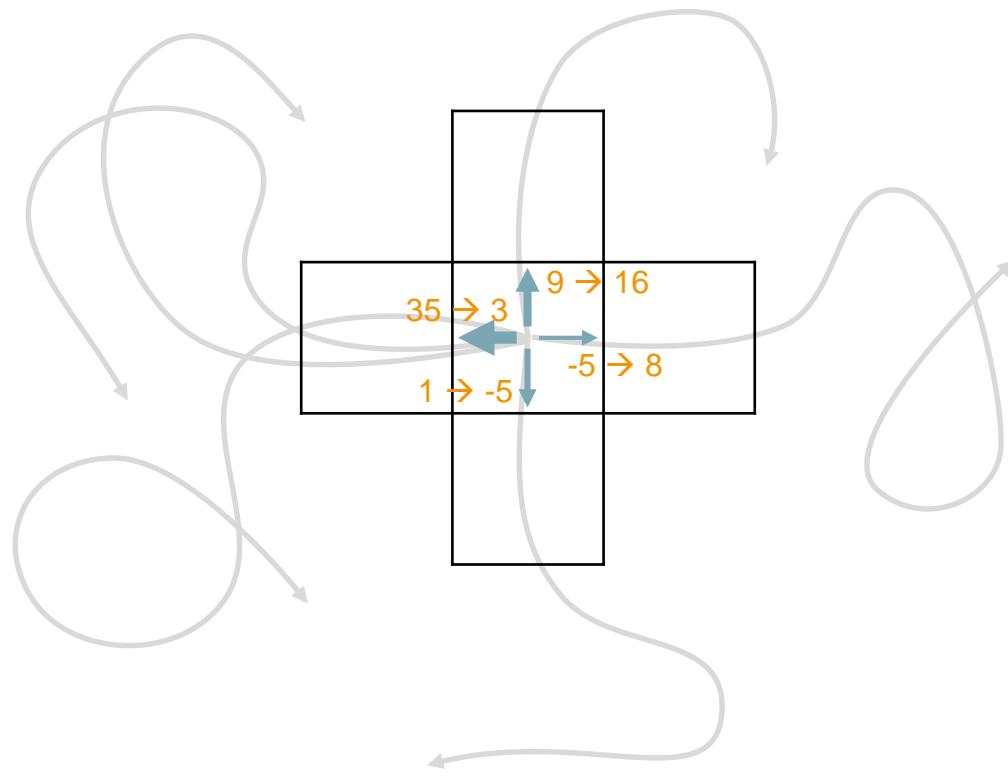
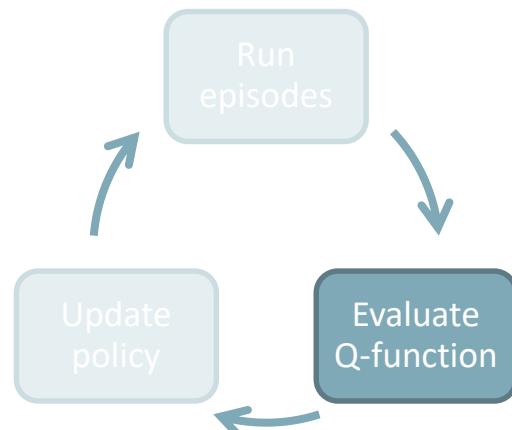
- First step: Run episodes based on current policy



# Methods for value-based RL (2/2)

General structure of all methods presented in this chapter

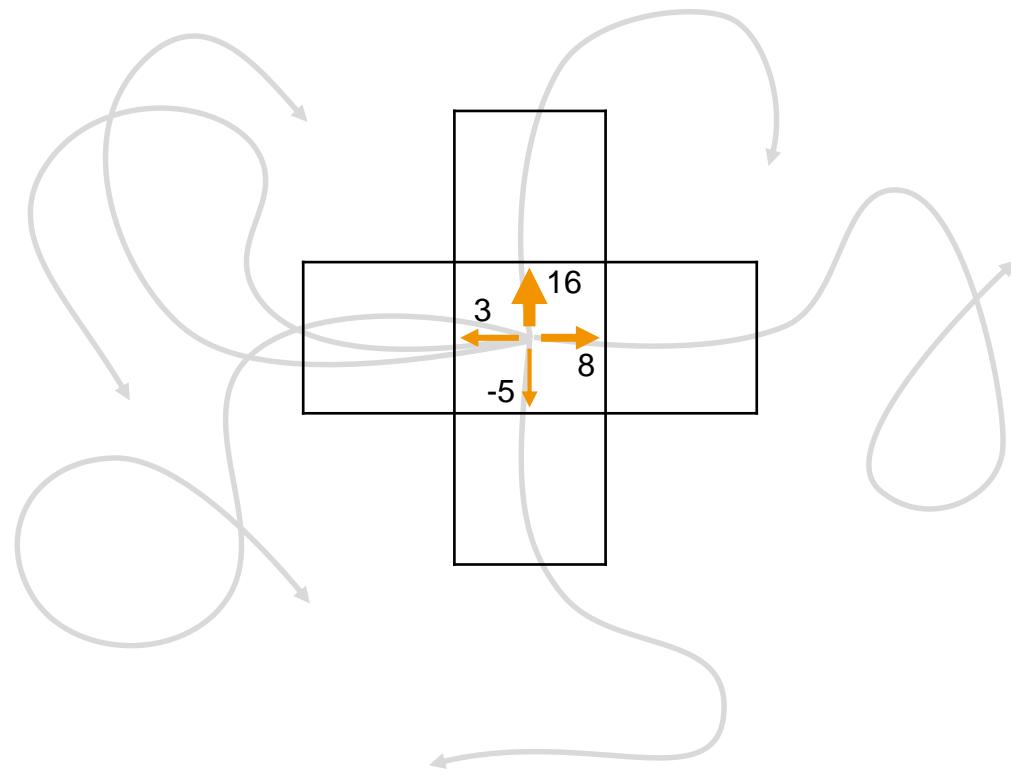
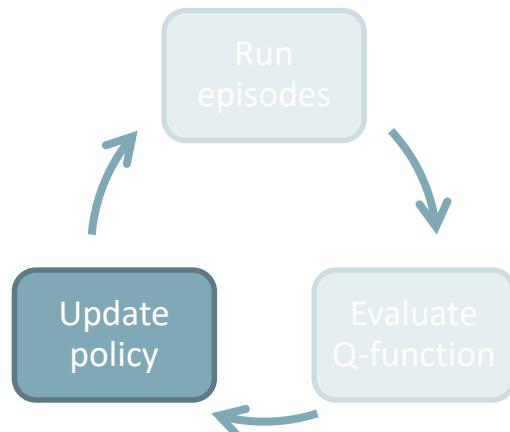
- Second step: Evaluate Q-function based on episode results



# Methods for value-based RL (2/2)

General structure of all methods presented in this chapter

- Third step: Update policy based on new Q-function

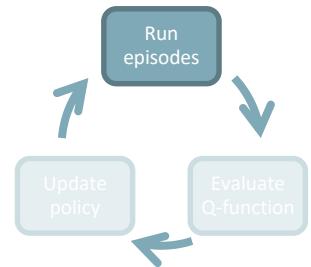


# Methods for value-based RL (2/2)

Run episodes based on current policy

1. Follow policy (execute actions according to policy) until
  - a terminal state is entered
  - a predefined maximum number of steps has been reached
2. Calculate return of each episode
  - go backwards through all states of each episode and calculate the return  $G_t$  of the state  $s_t$  / action  $a_t$  as

$$G_t = \gamma G_{t+1} + R_{t+1}$$



# Methods for value-based RL (2/2)

Example: One day of a student

- Given is the episode with

$Class1 \rightarrow Study \rightarrow Class2 \rightarrow Study \rightarrow$   
 $Class3 \rightarrow Pub \rightarrow Class3 \rightarrow Study \rightarrow$   
 $Sleep$

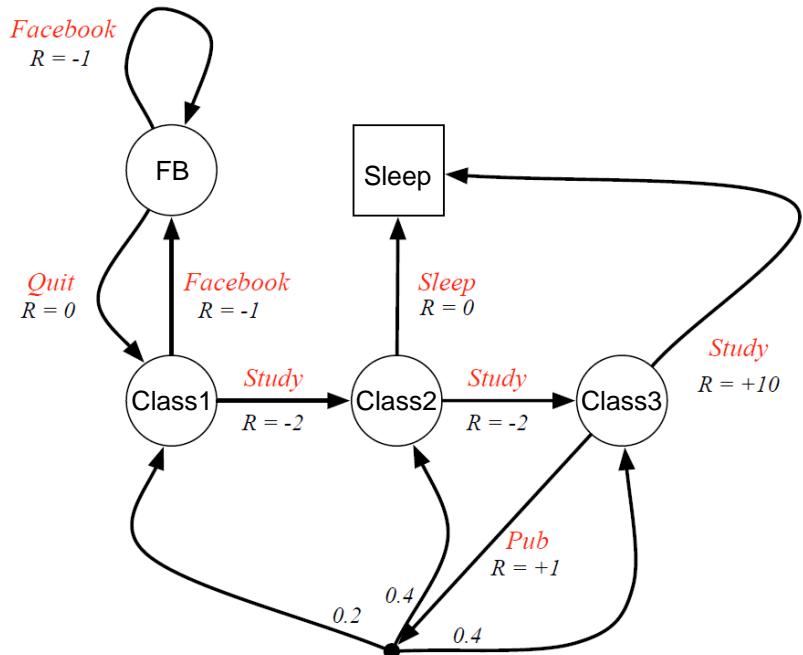
- With  $\gamma = 0.8$  the return of each state of the episode becomes

$$G_3(Class3) = +10 = 10$$

$$G_2(Class3) = +1 + 0.8 \cdot 10 = 9$$

$$G_1(Class2) = -2 + 0.8 \cdot 9 = 5.2$$

$$G_0(Class1) = -2 + 0.8 \cdot 5.2 = 2.16$$

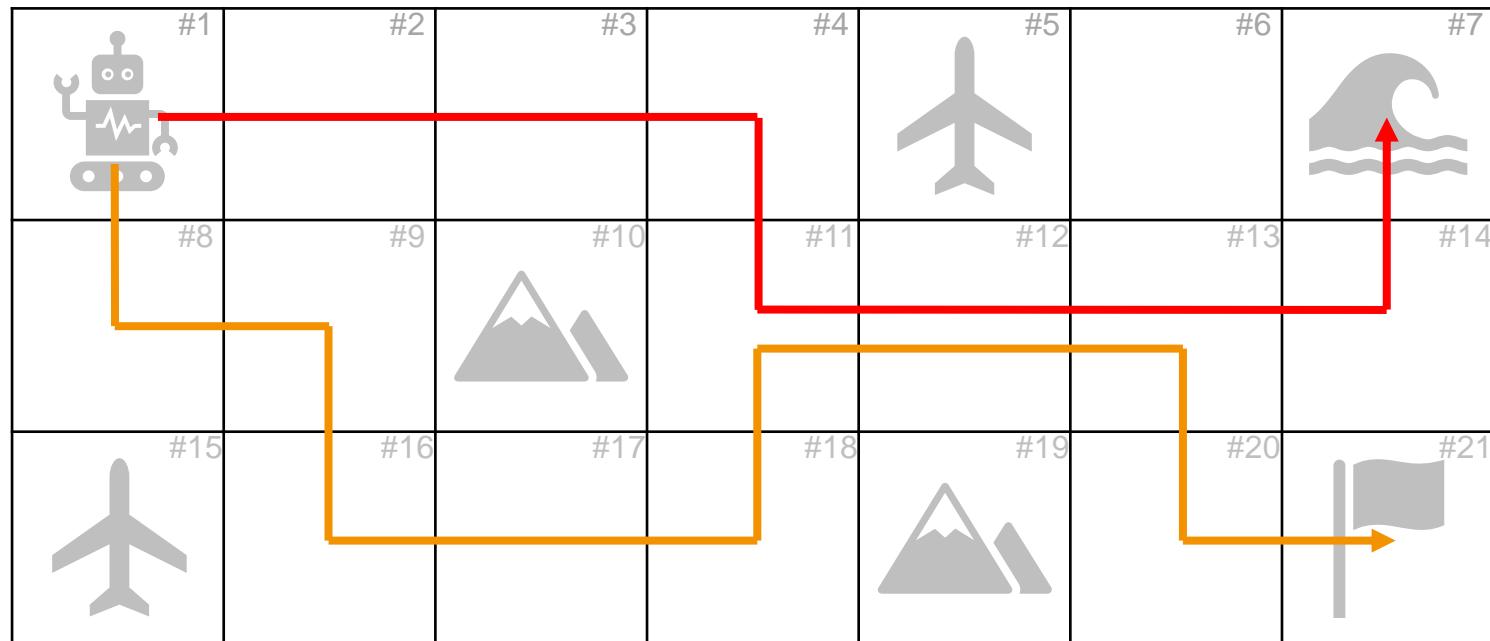


<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Methods for value-based RL (2/2)

Task: Robot maze

- Calculate the return of each state for the two displayed episodes  
(-1 reward for every action, -100 when entering the waves, discount factor  $\gamma = 0.8$ )



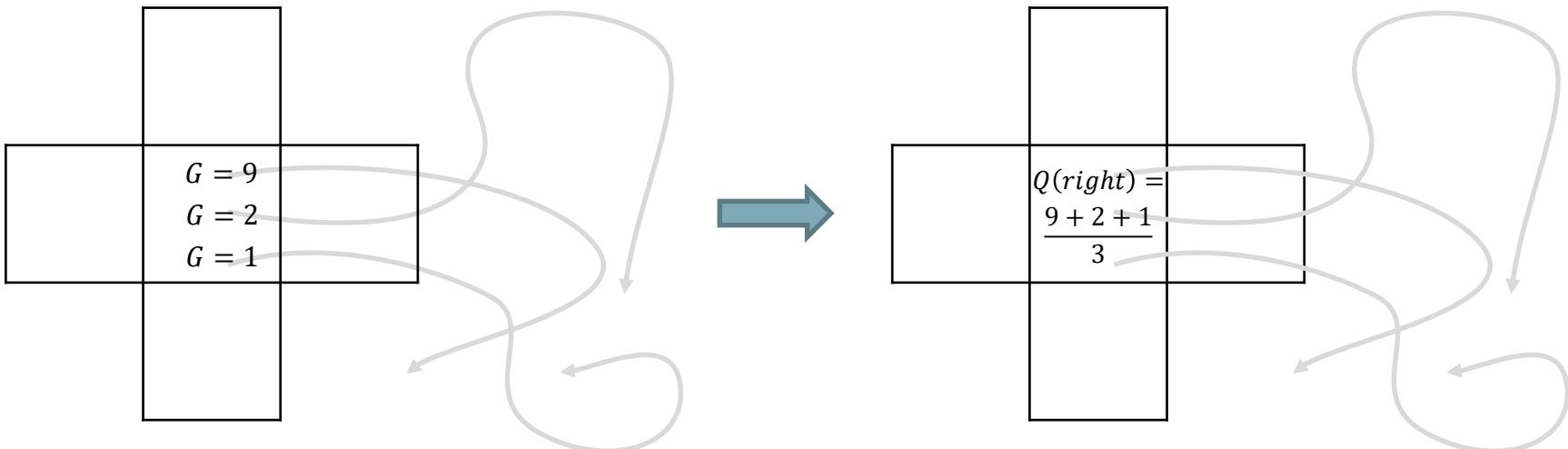
# Methods for value-based RL (2/2)

Evaluate Q-function based on episode results

## Monte-Carlo (MC) policy evaluation

- Idea: Average Q-values based on episodes run so far
- Mathematically with  $G_i$  as the return of the i-the episode of state  $s$  / action  $a$

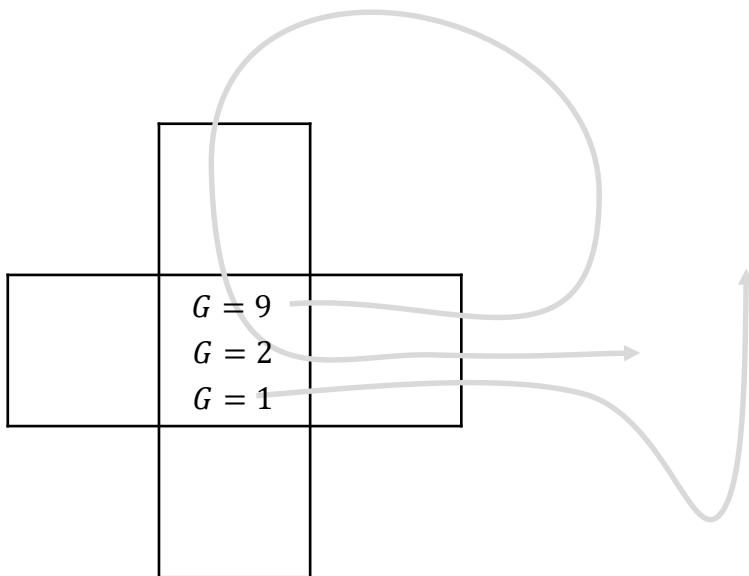
$$Q(s, a) = \frac{1}{n} \sum_{i=1}^n G_i$$



# Methods for value-based RL (2/2)

## Variations of MC evaluation

- Every-visit MC evaluation: Average Q-values based on **all** occurrences of an action within each episode
- First-visit MC evaluation: Average Q-values based on the **first** occurrence of an action within each episode



First-visit MC evaluation

$$Q(\text{right}) = \frac{9+1}{2} = 5$$

Every-visit MC evaluation

$$Q(\text{right}) = \frac{9+2+1}{3} = 4$$

# Methods for value-based RL (2/2)

- Incremental average: The mean of a sequence can be calculated incrementally.  
Advantage: No need to store all values  $x_i$

$$\begin{aligned}
 \mu_n &= \frac{1}{n} \sum_{i=1}^n x_i \\
 &= \frac{1}{n} \left( x_n + \sum_{i=1}^{n-1} x_i \right) \\
 &= \frac{1}{n} (x_n + (n - 1)\mu_{n-1}) \\
 &= \mu_{n-1} + \frac{1}{n} (x_n - \mu_{n-1})
 \end{aligned}$$

# Methods for value-based RL (2/2)

- Incremental MC updates: Apply incremental average formula to Q-values

$$Q(s, a) \leftarrow Q(s, a) + \frac{1}{n} (G - Q(s, a))$$

If the true Q-values change over time (nonstationary problems) it can be useful to track an exponential moving average, i.e. to forget old episodes over time

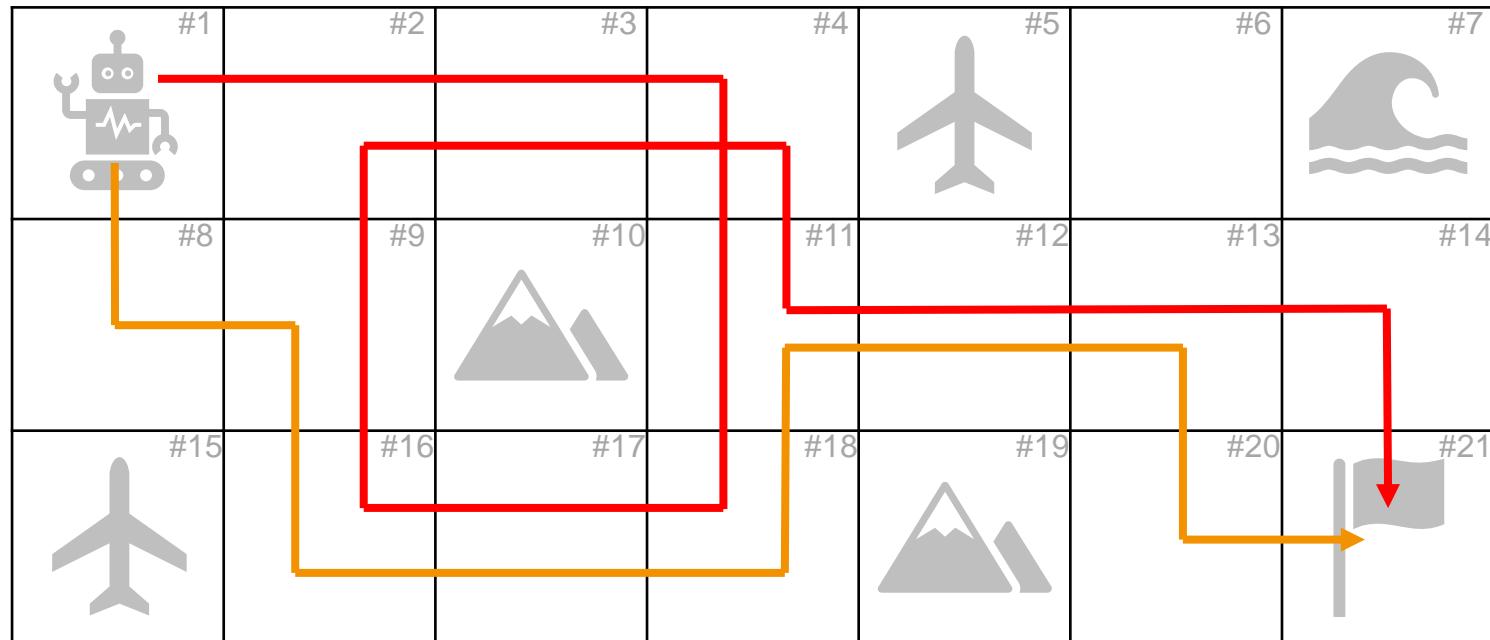
$$Q(s, a) \leftarrow Q(s, a) + \alpha (G - Q(s, a))$$

Task: What is the weight of the first return  $G_t$  after 1/10/100 normal/incremental MC update steps?

# Methods for value-based RL (2/2)

Task: Robot maze

- Calculate the resulting Q-values based on the two episodes using first-visit / every-visit non-incremental MC evaluation  
(-1 reward for every action, -100 when entering the waves, discount factor  $\gamma = 1$ )



# Methods for value-based RL (2/2)

## SARSA / Q-learning

- Idea: Combine the incremental MC update with the (slightly rewritten) Bellman expectation/optimality equation

$$Q(s, a) \leftarrow Q(s, a) + \alpha(G - Q(s, a))$$

$$G = R + \gamma Q(s', a')$$

$$Q_*(s, a) = E \left[ R + \gamma \max_{a'} Q_*(s', a') \right]$$

- SARSA update rule (with learning rate  $\alpha$ )**

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma Q(s', a') - Q(s, a))$$

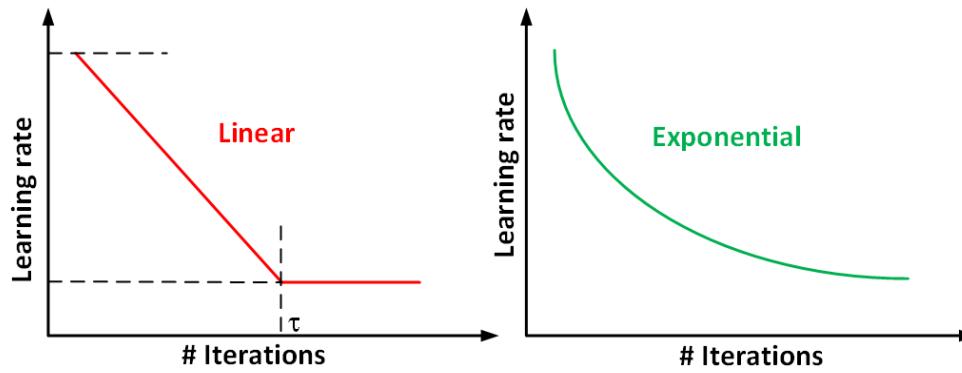
- Q-learning update rule (with learning rate  $\alpha$ )**

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( R + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

# Methods for value-based RL (2/2)

## Learning rate

- The learning rate  $\alpha \in [0, 1]$  influences how fast the Q-values are updated:  
 $\alpha \approx 1$ : fast update, brittle learning  
 $\alpha \approx 0$ : slow update, more consistent learning
- Guaranteed convergence to the optimal policy if learning rate converges to zero over time (and a few other conditions are met)
- Recommendation: Start with a larger learning rate and decrease it over time, e.g. through linear/exponential decay



[https://miro.medium.com/max/1400/1\\*MB3liDQaa9-Oh1\\_8uI4rvA.png](https://miro.medium.com/max/1400/1*MB3liDQaa9-Oh1_8uI4rvA.png)

# Methods for value-based RL (2/2)

Example: One day of a student

- Given is the episode with

$Class1 \rightarrow Study \rightarrow Class2 \rightarrow Study \rightarrow$   
 $Class3 \rightarrow Pub \rightarrow Class3 \rightarrow Study \rightarrow$   
 $Sleep$

and initial Q-values of 0 for all actions,  
 $\gamma = 0.8$ ,  $\alpha = 0.5$

- Using incremental MC evaluation  
with moving average, the Q-values become

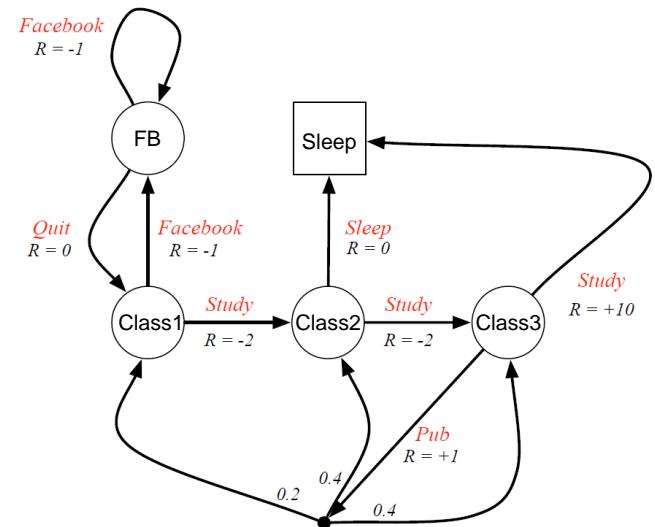
$$Q(Class1, Study) = 0 + 0.5 \cdot (2.16 - 0) = 1.08$$

$$Q(Class2, Study) = 0 + 0.5 \cdot (5.2 - 0) = 2.6$$

$$Q(Class3, Pub) = 0 + 0.5 \cdot (9 - 0) = 4.5$$

$$Q(Class3, Study) = 0 + 0.5 \cdot (10 - 0) = 5$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$$



<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Methods for value-based RL (2/2)

- Using SARSA, the Q-values become

$$Q(\text{Class1}, \text{Study}) = 0 + 0.5 \cdot (-2 + 0.8 \cdot 0 - 0) = -1$$

$$Q(\text{Class2}, \text{Study}) = 0 + 0.5 \cdot (-2 + 0.8 \cdot 0 - 0) = -1$$

$$Q(\text{Class3}, \text{Pub}) = 0 + 0.5 \cdot (1 + 0.8 \cdot 0 - 0) = 0.5$$

$$Q(\text{Class3}, \text{Study}) = 0 + 0.5 \cdot (10 + 0.8 \cdot 0 - 0) = 5$$

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

- Using Q-learning, the Q-values become

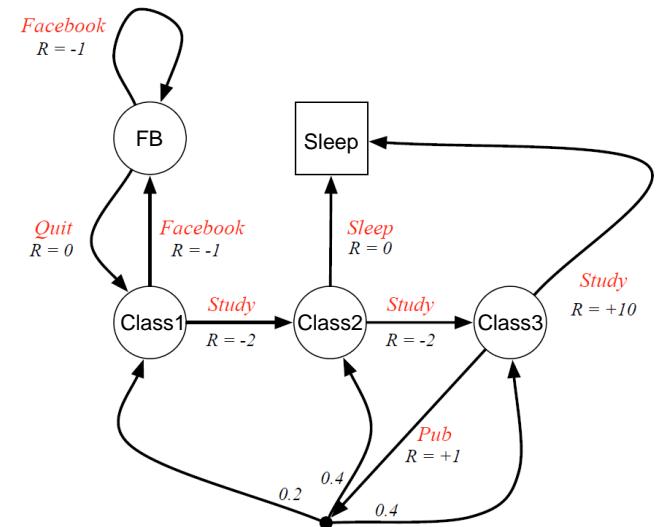
$$Q(\text{Class1}, \text{Study}) = 0 + 0.5 \cdot (-2 + 0.8 \cdot 0 - 0) = -1$$

$$Q(\text{Class2}, \text{Study}) = 0 + 0.5 \cdot (-2 + 0.8 \cdot 0 - 0) = -1$$

$$Q(\text{Class3}, \text{Pub}) = 0 + 0.5 \cdot (1 + 0.8 \cdot 0 - 0) = 0.5$$

$$Q(\text{Class3}, \text{Study}) = 0 + 0.5 \cdot (10 + 0.8 \cdot 0 - 0) = 5$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left( R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right)$$

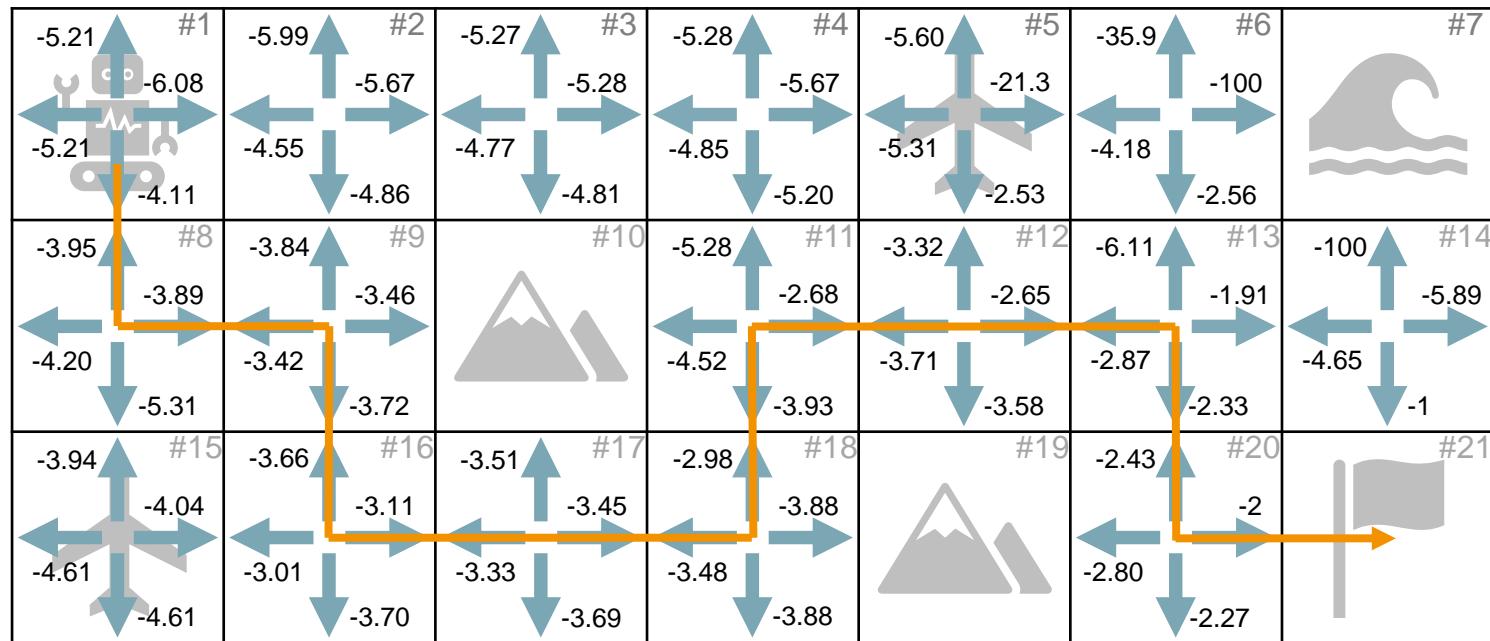


<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Methods for value-based RL (2/2)

Task: Robot maze

- Update the Q-values for the given episode using incremental MC evaluation / SARSA / Q-learning (-1 reward for every action, discount factor  $\gamma = 0.8$ , learning rate  $\alpha = 0.5$ , SARSA / Q-learning updates after every action, MC evaluation at the end of the episode)



# Methods for value-based RL (2/2)

Update policy based on new Q-function

## $\epsilon$ -greedy policy

- With small probability  $\epsilon$  execute random action
- Otherwise execute current "best" action (similar to policy improvement)
- Resulting policy

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \text{ maximizes } Q(s, a). \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{for all other actions.} \end{cases}$$

- Resulting policy becomes optimal if  $\epsilon$  converges to zero over time (and a few other conditions are met), e.g.  $\epsilon = \frac{1}{k}$  for the k-th episode

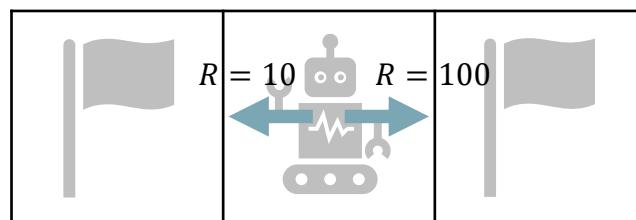


# Methods for value-based RL (2/2)

- Compared to a greedy policy (e.g. obtained through policy optimization) an  $\epsilon$ -greedy policy ensures that all states are visited if we run enough episodes. This allows the robot to find better actions without sticking to a suboptimal action.

Example: Trivial robot maze

- Robot can only move left/right
- If he moves left/right, the episode is over and he gets 10/100 reward
- Using a greedy policy (+MC evaluation/SARSA/Q-learning) and an initialization of all Q-values of 0, the robot will stick forever to the first action it picks
- Using a  $\epsilon$ -greedy policy (+MC evaluation/SARSA/Q-learning), the robot will also try out the other action and eventually choose it in case it performs better



# Methods for value-based RL (2/2)

## SARSA in literature

### Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size  $\alpha \in (0, 1]$ , small  $\varepsilon > 0$

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

Loop for each episode:

  Initialize  $S$

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

  Loop for each step of episode:

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

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

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

$S \leftarrow S'; A \leftarrow A'$ ;

  until  $S$  is terminal

Sutton, Barto: Reinforcement Learning

# Methods for value-based RL (2/2)

## Q-learning in literature

### Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size  $\alpha \in (0, 1]$ , small  $\varepsilon > 0$

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

Loop for each episode:

  Initialize  $S$

  Loop for each step of episode:

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

Sutton, Barto: Reinforcement Learning

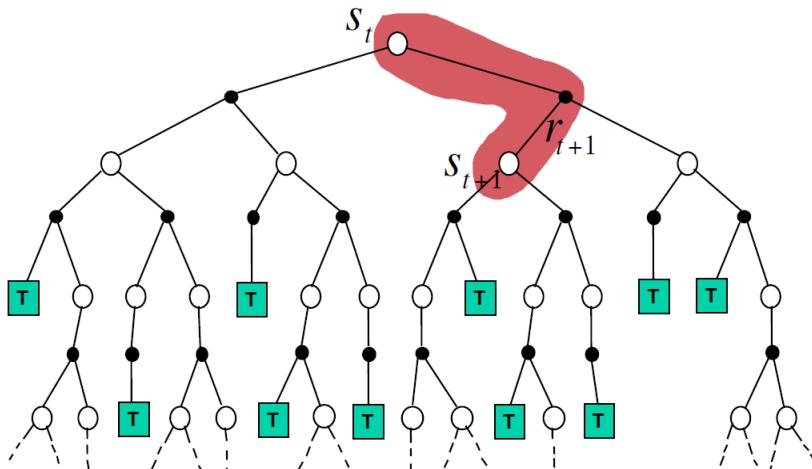
# Methods for value-based RL (2/2)

Comparison of Monte-Carlo and temporal-difference methods

- SARSA and Q-learning belong to the class of **temporal difference (TD) methods**, MC evaluation belongs to the class of **Monte-Carlo (MC) methods**
- TD methods update the (V-)/Q-function based only on a single action (low variance), MC methods update the (V-)/Q-function based on multiple actions (high variance)

TD methods (e.g. SARSA)

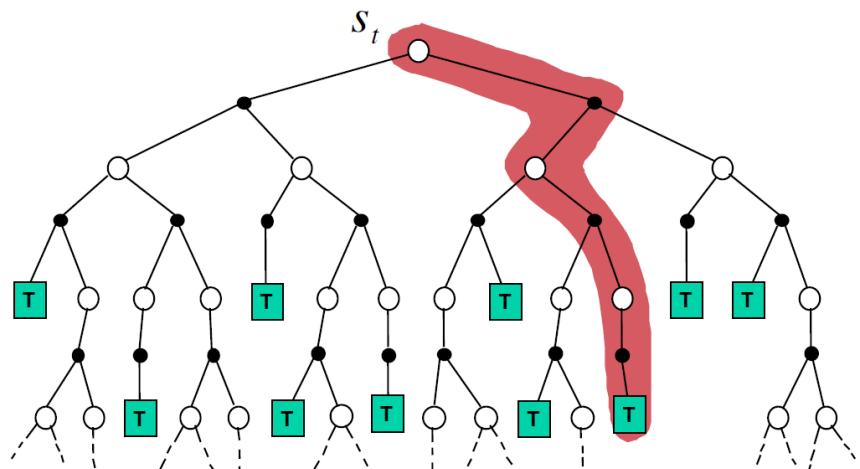
$$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma Q(s', a') - Q(s, a))$$



<https://www.deeplearning.ai/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

MC methods

$$Q(s, a) \leftarrow Q(s, a) + \alpha(G - Q(s, a))$$



# Methods for value-based RL (2/2)

- TD methods exploit the Markov property by updating every Q-values based (only) on its successor. Hence they are usually more efficient in Markovian environments (e.g. MDPs) than MC methods
- Unless the learning rate is very large, TD methods are quite sensitive to wrongly initialized V-/Q-values. This is less a problem for MC methods

	TD methods	MC methods
when to update	after every action	after every episode
variance	low	high
uses Markov property	yes	no
works best in	Markovian environments	non-Markovian environments
sensitive to initial value	yes	no

# Methods for value-based RL (2/2)

## Comparison of SARSA and Q-learning

- Recap: SARSA

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma Q(s', a') - Q(s, a))$$

- Recap: Q-learning

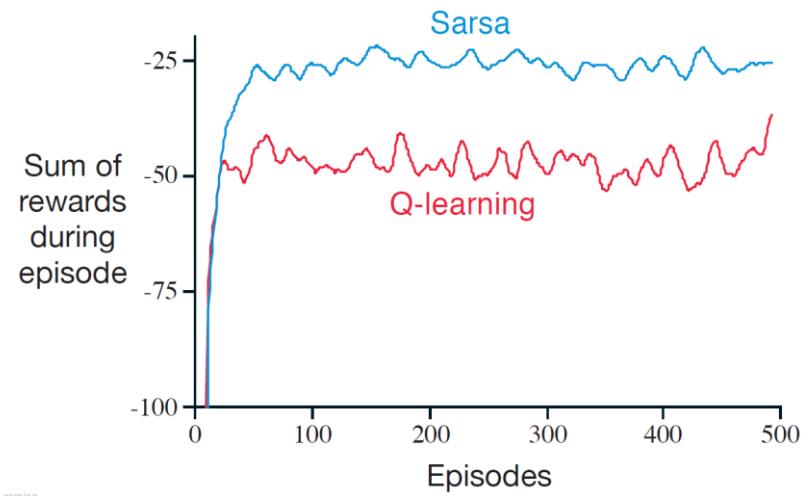
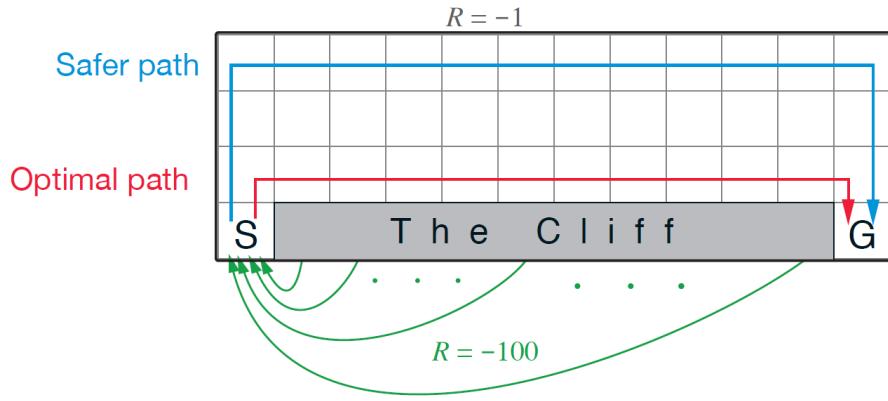
$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( R + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

- SARSA only considers actions in its update rule that the agent performed based on its current policy (**on-policy**) based on the Bellman expectation equation
- Q-learning also considers actions in its update rule (the max-operator) independent of its current policy / current episode (**off-policy**) based on the Bellman optimality equation

# Methods for value-based RL (2/2)

Example: Cliff walking

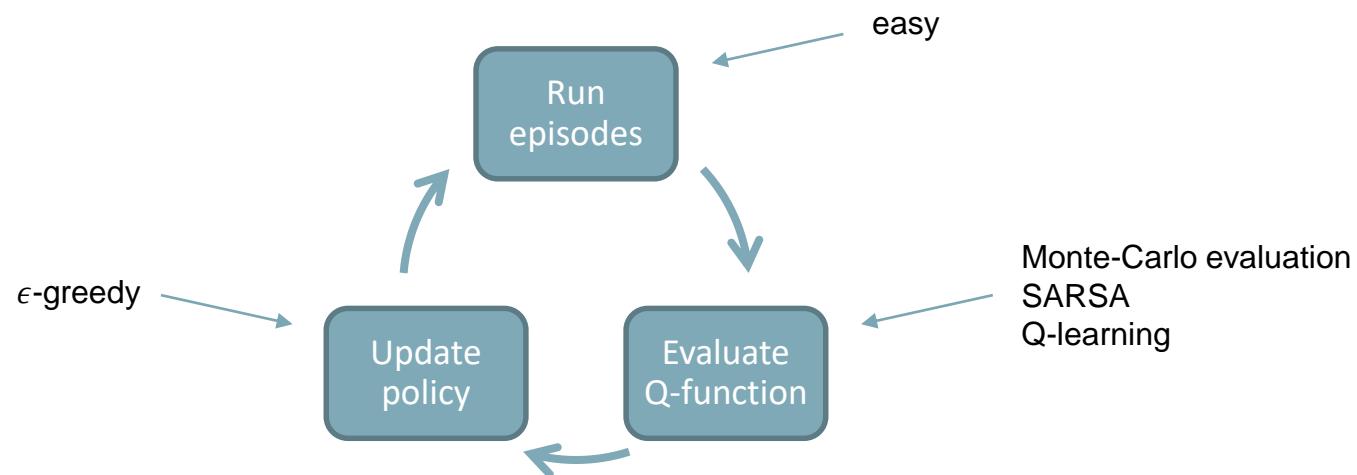
- Start S, Goal G, -1 reward for every action, -100 reward when entering the cliff state,  $\epsilon$ -greedy policy with  $\epsilon = 0.1$
- The resulting policy using SARSA updates will be "safer path", as SARSA incorporates the risk of falling down due to an  $\epsilon$ -greedy policy
- The resulting policy using Q-learning updates will be "optimal path", as Q-learning acts greedily when updating the Q-values
- For  $\epsilon \rightarrow 0$  both methods will converge to the optimal path



# Methods for value-based RL (2/2)

## Brief summary

- This chapter tackled the problem to find an optimal policy for an unknown, discrete MDP
- The robot has to find an optimal policy by learning from episodes and evaluating the obtained rewards
- Three iterative methods are presented:  
 Monte-Carlo evaluation  
 SARSA  
 Q-learning



# Kahoot!

# Reward shaping



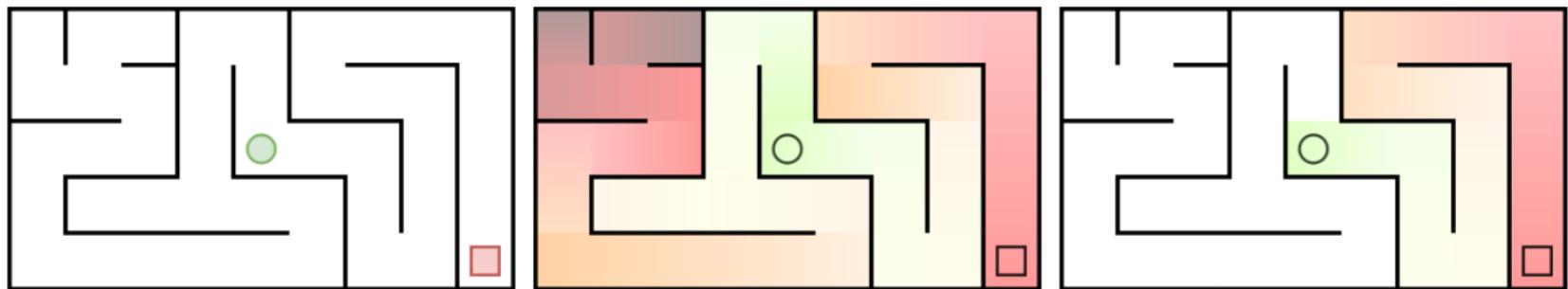
<https://dudeperf3ct.github.io/II/2019/12/19/Reinforcement-Learning/>

# Reward shaping

## Reward shaping

- covers all methods that modify the reward given to the agent in order to optimize (speed up) training
- incorporates prior knowledge about the task into the reward function

Example: Different levels of reward shaping depending on the distance to the goal



<https://arxiv.org/pdf/2210.09579.pdf>

# Reward shaping

Problem with RL in general

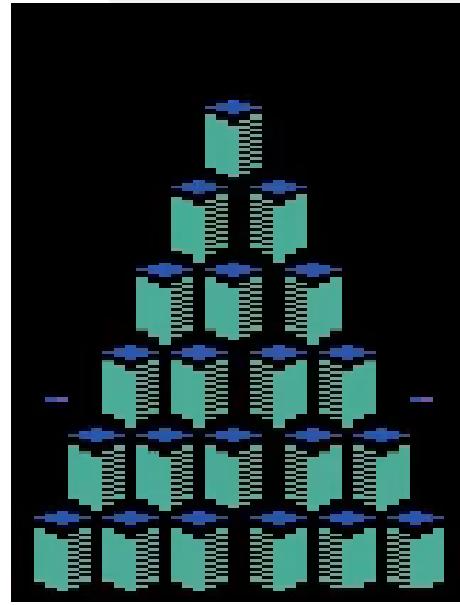
- RL is extremely good in finding bugs/loopholes in your program, this effect may get aggravated through reward shaping

Example 1: Q-Bert

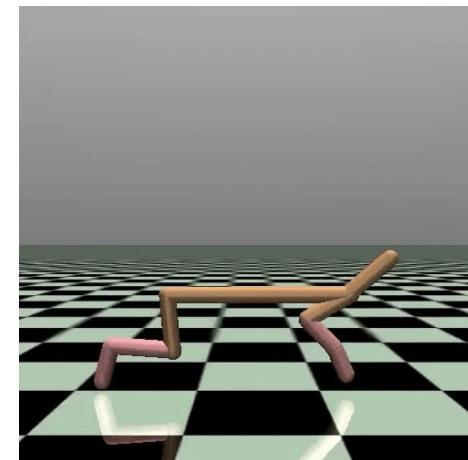
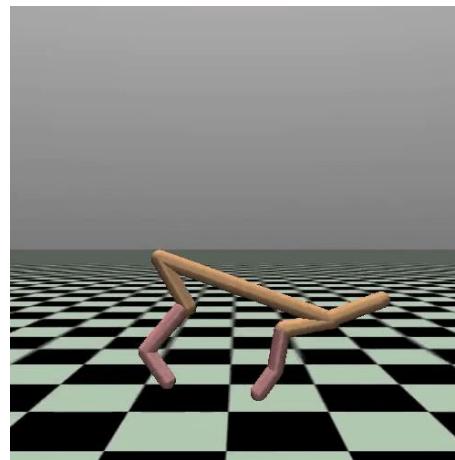
- Old Amiga game, RL agent found a (previously unknown ) bug during training

Example 2: Half Cheetah

- Through optimized parameters and a more positive reward if the agent moves faster to the roll, it learned that it is faster to roll instead of running



[https://www.youtube.com/shorts/uVHAAiA3F\\_U](https://www.youtube.com/shorts/uVHAAiA3F_U)



<https://bair.berkeley.edu/blog/2021/04/19/mbrl/>

# Reward shaping

Problem with reward shaping: Cobra effect

- Modifying a (initially simple) reward function in order to speed up training may lead to undesired consequences

Example: CoastRunners

- Real goal: Finish the race as quickly as possible
- Reward assigned to agent: Collect as many targets along the way



<https://www.youtube.com/watch?v=tOIHko8ySg&t=5s>

# Reward shaping

## Types of rewards

### immediate rewards

get reward immediately after taking an action

### dense rewards

get reward every timestep

### positive rewards

rewards  $R_t \geq 0$

### individual rewards

focus on one task aspect

vs.

### delayed rewards

get reward some time after taking an action

vs.

### sparse rewards

do not get reward every timestep

vs.

### negative rewards

rewards  $R_t \leq 0$

vs.

### additive rewards

focus on many task aspects

# Reward shaping

## Immediate vs. delayed rewards

### Immediate rewards

- Get a reward (associated with that action) immediately after taking the action
- Examples
  - Autonomous driving: Distance to target
  - Learning to walk: Distance traveled

### Delayed rewards

- Get a reward (associated with that action) some time after taking the action
- Examples
  - Games (get a reward of e.g. +1/-1 when winning/losing the game)
  - Stock market (buy stocks now, see how it performs over the years)

→ Delayed rewards are way more difficult for RL than immediate rewards, training can be sped up if suitable immediate rewards are added to delayed rewards.

# Reward shaping

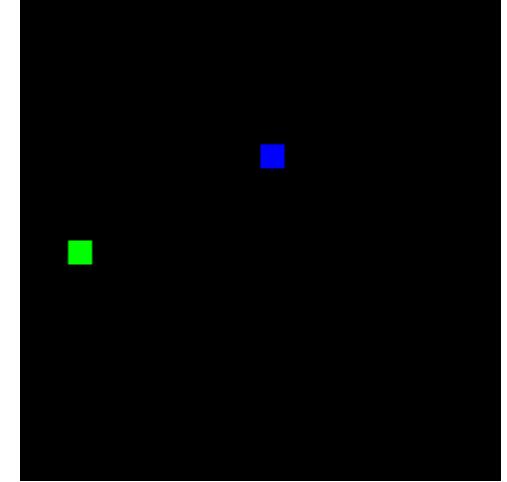
## Dense vs. sparse rewards

### Dense rewards

- Get a reward at every timestep
- Examples
  - Robot maze
  - Autonomous driving: Distance to target update every second

### Sparse rewards

- Don't get a reward at every timestep (e.g. every N steps)
- Examples
  - Snake
  - Soccer: Goals scored



<https://medium.com/ml-everything/reinforcement-learning-with-sparse-rewards-8f15b71d18bf>

→ Sparse rewards are usually more difficult than dense rewards but by far not as difficult as massive delayed rewards

# Reward shaping

## Positive vs. negative rewards

### Positive rewards

- Get a positive reward when taking a good action or reaching the goal state
- Examples
  - Learning to walk: Distance traveled

### Negative rewards

- Get a negative reward for every action that does not reach the goal state
- Examples
  - Robot maze

→ Negative rewards encourage the agent to reach a goal state as quickly as possible, positive rewards may stimulate the robot to accumulate reward rather than reaching the goal state (useful for e.g. non-episodic tasks)

# Reward shaping

## Individual vs. additive rewards

### Individual rewards

- Rewards focusing only on the main aspect of the task
- Examples
  - Learning to walk: Distance travelled

### Additive rewards

- Rewards focusing on multiple task aspects (with  $\alpha, \beta, \dots$  as weights):  $R = \alpha R_a + \beta R_b + \dots$
- Examples
  - Learning to walk: Distance travelled and torso height

→ Additive rewards can be a good addition to individual rewards if learning progress is slow

Learning to Walk  
via Deep Reinforcement Learning

Submission ID: 60

<https://www.youtube.com/watch?v=n2gE7n11h1Y>

# Reward shaping

Task: How would you shape a reward for

- running (with a simulated human)
- driving a car
- playing chess

# Reward shaping

Example: Robot maze

# Exploration and exploitation

# Exploration and exploitation

## Exploration vs. exploitation dilemma

Online decision-making involves a fundamental choice:

- Exploitation: Make the best decision given current information
- Exploration: Gather more information

## Consequences

- The best long-term strategy may involve short-term sacrifices
- Only gather as much information as necessary to make the best overall decision

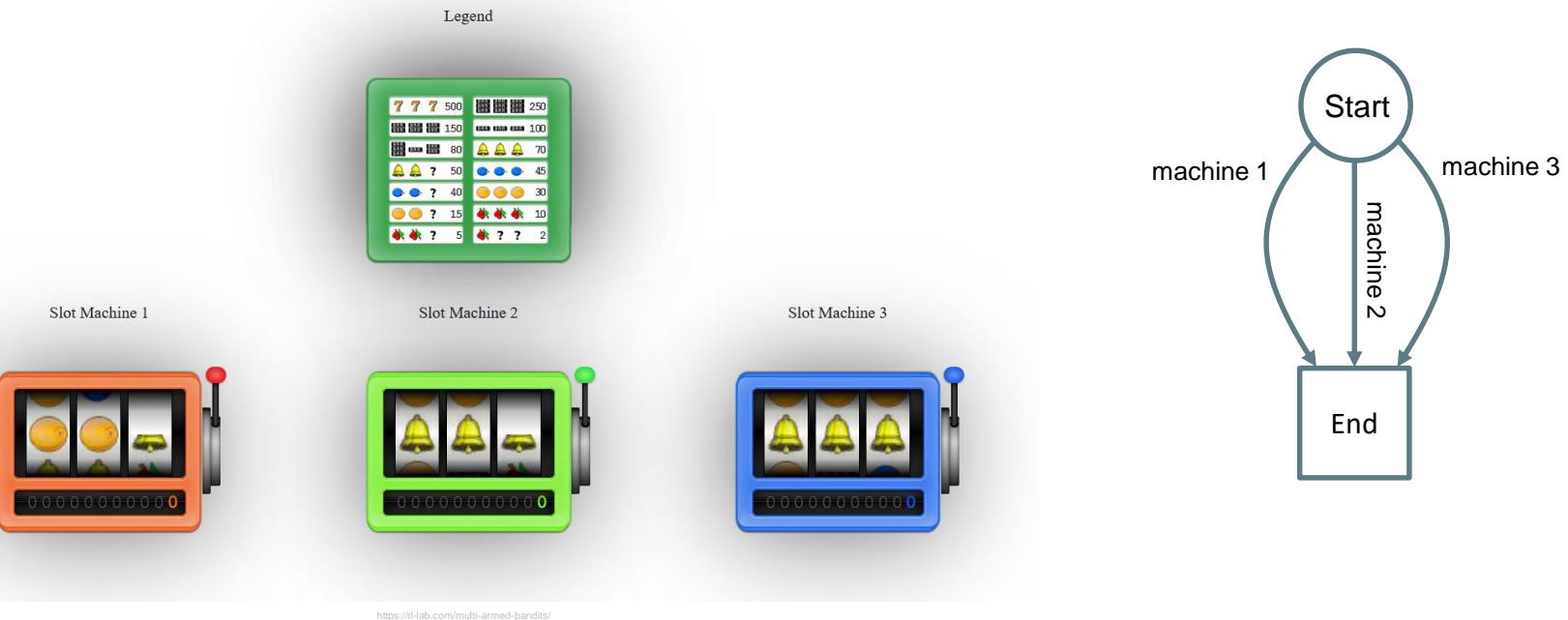
Four methods are presented

- optimistic initialization
- R-Max
- decaying  $\epsilon$ -greedy
- UCB1

# Exploration and exploitation

## Example: Multi-armed bandits

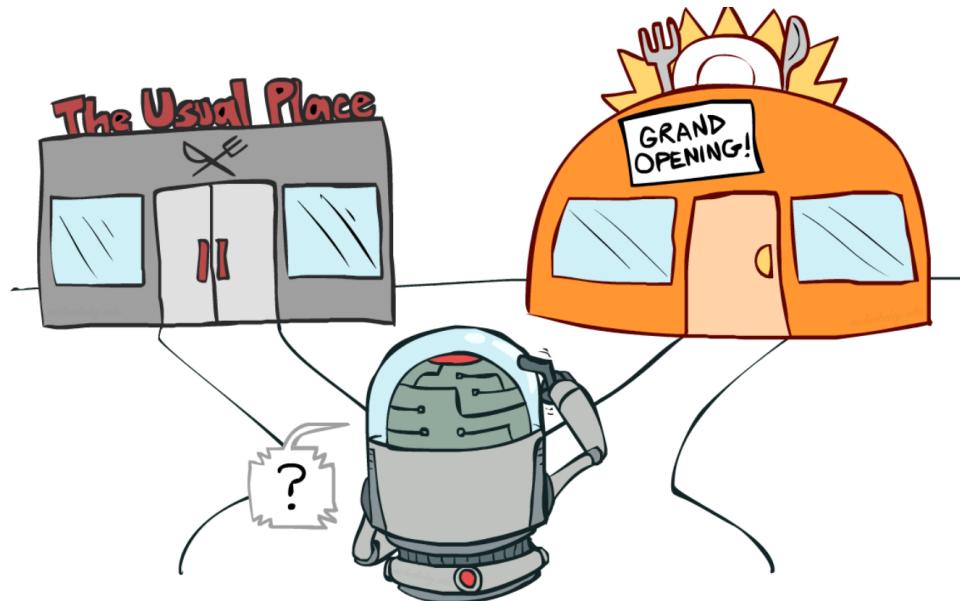
- Three actions for a given state: Pull lever of slot machine 1/2/3 and get reward
- Special case: Probabilistic (non-deterministic) rewards  $R$
- At time  $t$ , agent selects action  $a_t$  and gets reward  $R_t$
- Goal: Maximize (average) reward  $R_t$



# Exploration and exploitation

Related real life examples

- Should I stick to my current job or get a new one?
- How many prices should I compare before buying product X?
- Secretary problem



[http://procaccia.info/courses/1536116/slides/781\\_rl\\_rmax.pdf](http://procaccia.info/courses/1536116/slides/781_rl_rmax.pdf)

Task: What other related examples come to your mind?

# Exploration and exploitation

Disclaimer: All methods described in this chapter are only thoroughly analyzed for multi-armed bandits and not for MDPs in general, although they also provide very good (but not optimal) results for MDPs with discrete actions

Difference between multi-armed bandit and general MDP

	Multi-armed bandit	MDP
episode length	1	usually $> 1$
reward	stochastic	usually deterministic
goal	max. (avg.) reward	max (avg.) discounted reward

# Exploration and exploitation

## Regret

- Is based on the optimal V-value, defined as  $V_* = \max_a Q_*(a)$
- Defines how much worse an action is compared to the optimal action at time step  $t$

$$I_t = E[V_* - Q(A_t)]$$

## Total regret

- Is the accumulated regret until timestep  $t$

$$L_t = E \left[ \sum_{\tau=1}^t V_* - Q(A_\tau) \right]$$

- Maximize return  $\hat{=}$  minimize total regret
- always executing the best action equals total regret of 0

# Exploration and exploitation

## Counting regret

- the total regret depends on two factors
  - how bad an action is  
(**gap**  $\Delta_a = V_* - Q(a)$  as difference of action  $a$  to optimal action)
  - how often the action is taken  
(count  $N_t(a)$  as the number of selecting  $a$  until timestep  $t$ )
- the counting regret is a reformulation of the total regret as

$$L_t = E \left[ \sum_{\tau=1}^t V_* - Q(A_\tau) \right] \quad \longleftarrow \text{iterate over time steps } \tau$$

$$= \sum_{a \in \mathcal{A}} E[N_t(a)] \cdot \Delta_a \quad \longleftarrow \text{iterate over actions } a$$

# Exploration and exploitation

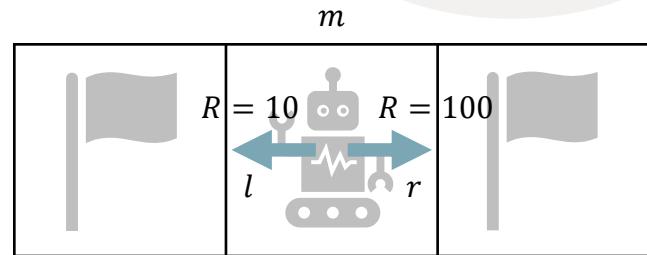
Example: Trivial robot maze

- Policy: Move right for two episodes, then move left for one episode.

$$V_*(m) = 100$$

$$Q(m, r) = 100 \quad \rightarrow \Delta_r = 0$$

$$Q(m, l) = 10 \quad \rightarrow \Delta_l = 90$$



- After  $n$  timesteps/episodes with counting regret

$$E[N_n(r)] = \frac{2}{3}n \quad \rightarrow \quad L_n = \frac{0 \cdot 2}{3}n + \frac{90 \cdot 1}{3}n = 30n$$

$$E[N_n(l)] = \frac{1}{3}n$$

- After  $n$  timesteps/episodes with total regret

$$L_n = E[0 + 0 + 90 + 0 + 0 + 90 + \dots] = 30n$$

# Exploration and exploitation

Total regret for different policies

- Optimal policy: Always pick the best action

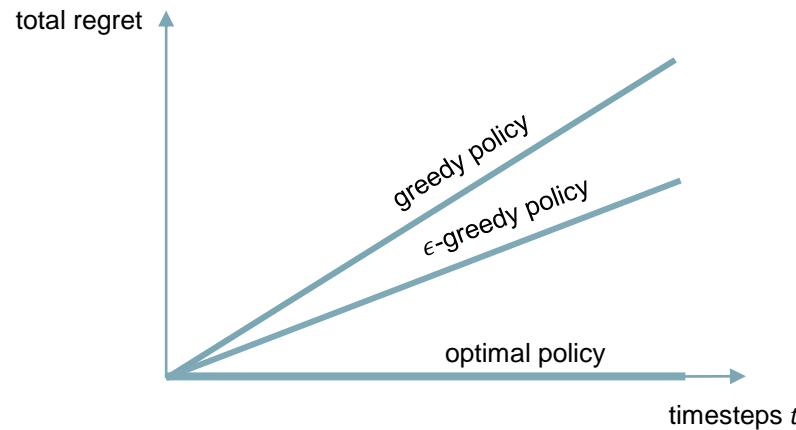
$$L_t = 0$$

- greedy policy: Sometimes (systematically) pick the wrong action

$$L_t = c \cdot t$$

- $\epsilon$ -greedy policy: With probability  $\epsilon$  pick the wrong action

$$L_t = d \cdot t$$

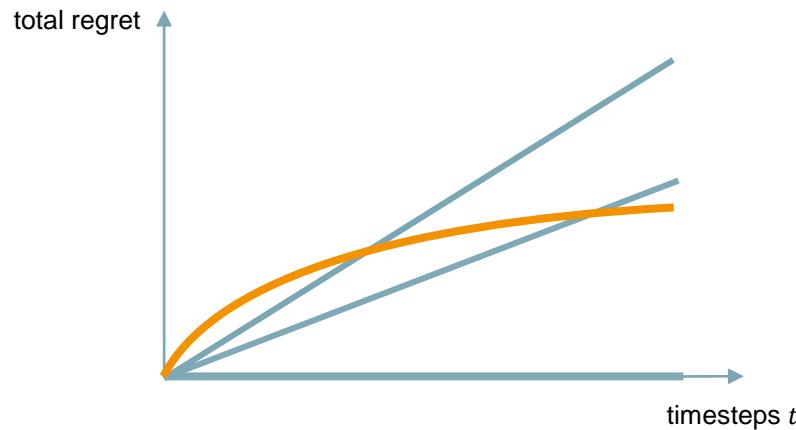


# Exploration and exploitation

## Other considerations

- If an algorithm explores forever, it will have a linear total regret
- If an algorithm explores never, it will have a linear total regret
- If an algorithm stops exploration too early, it will have a linear total regret

Are there exploration algorithms with sublinear total regret converging to an optimal policy?  
→ yes



# Exploration and exploitation

## Total regret lower bound

- defines how small the total regret can become
- depends on the reward distribution  $\mathcal{R}^a$  when picking action  $a$  and the **Kullback-Leibler divergence  $KL$**  measuring the dissimilarity between two probability distributions
- Theorem (Lai and Robbins)

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

most important:  
total regret increases  
logarithmically over  $t$   
in the best case

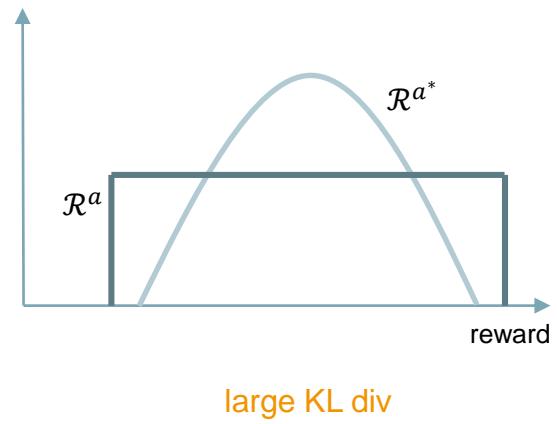
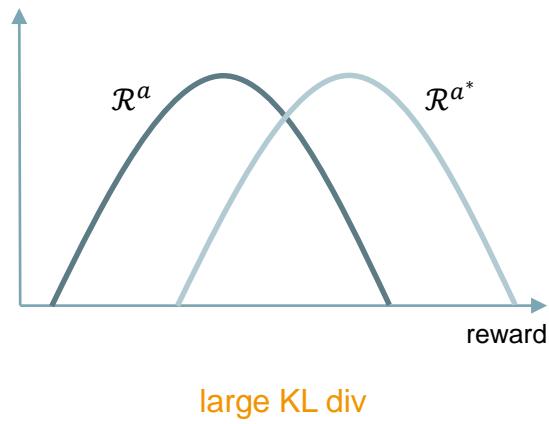
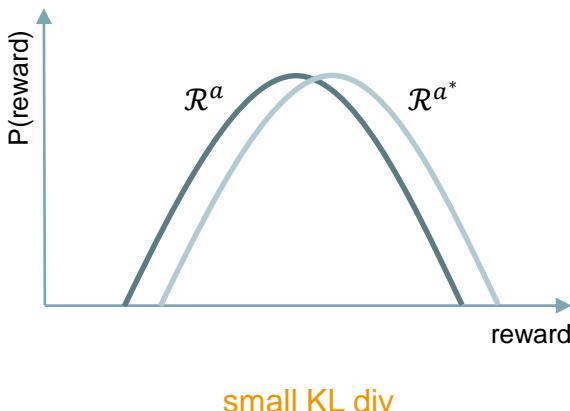
sum over all actions with  
a gap  $> 0$  (that are the  
suboptimal actions)

the larger the gaps are, the  
worse it is on average if the  
wrong action is picked

the smaller the  $KL$   
divergence is, the more  
similar the reward  
distributions are and the  
more difficult it becomes  
to differentiate the  
reward distributions

# Exploration and exploitation

## Example



Task: Create different probability distributions and estimate the KL divergence

# Exploration and exploitation

## Optimism in the face of uncertainty

- Is a heuristic/idea how to behave if one has to decide between multiple actions with unknown reward
- Colloquially: "If you have no idea about the outcome of an action, pick it. It might be the best"

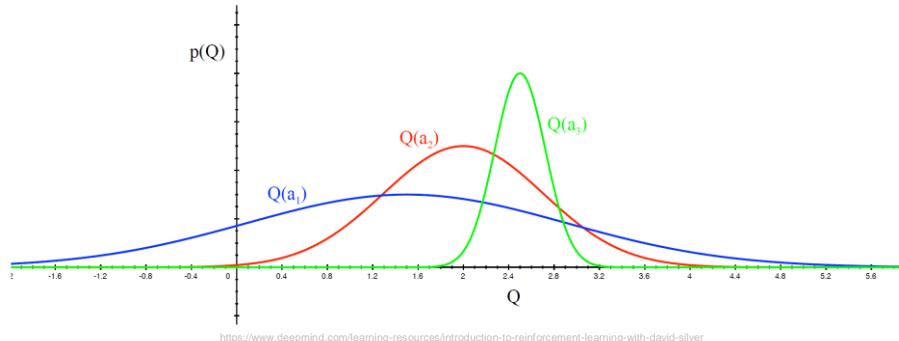


<https://medium.com/@uzair.ahmed2/optimism-a-path-to-success-4c7c4cc23fd7>

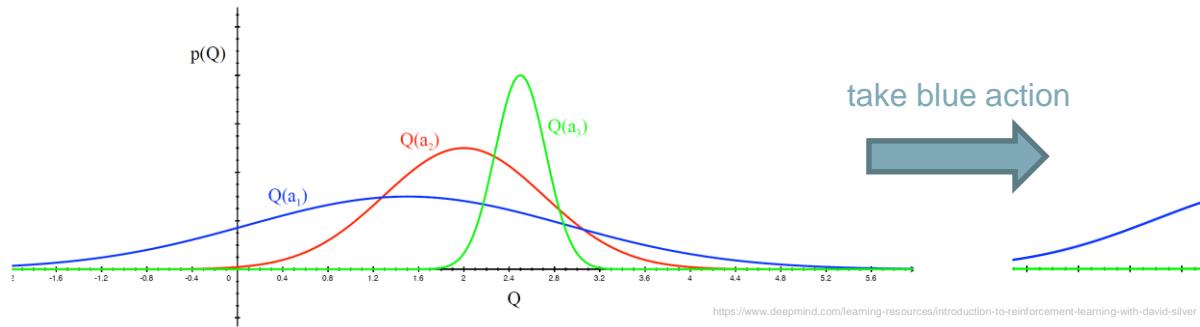
# Exploration and exploitation

Example: Estimated reward distributions for three actions

- A greedy policy would take the green action
- An optimistic policy would take the blue action



- After taking the blue action, one is more certain about the outcome of the blue action  
→ might take another one next time



# Exploration and exploitation

## Optimistic initialization

- Initialize Q-function to a high value
- Then update it using MC evaluation / SARSA / Q-learning  
(SARSA / Q-learning may take long to converge if initial value is too large)
- Linear total regret when being combined with ( $\epsilon$ -)greedy  
(no guarantee that optimal action will be taken after some time in every state)
- Standard for a lot of practical problems

**Q-learning (off-policy TD control) for estimating  $\pi \approx \pi_*$**

Algorithm parameters: step size  $\alpha \in (0, 1]$ , small  $\varepsilon > 0$

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

Loop for each episode:

Initialize  $S$

Loop for each step of episode:

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

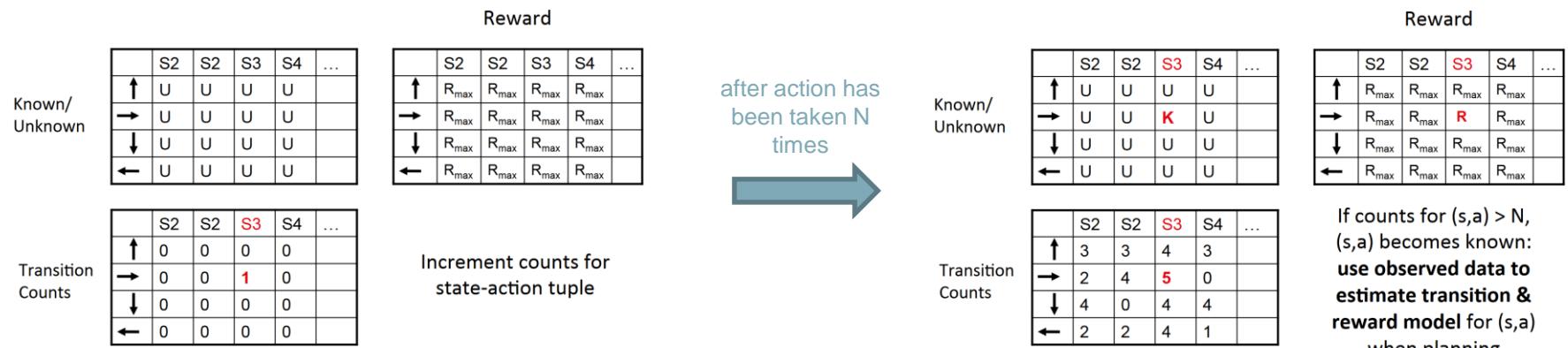
Sutton, Barto: Reinforcement Learning

# Exploration and exploitation

## R-max

- Initialize rewards  $R$  to the maximum value  $R_{max}$
- If the corresponding action has been taken  $N$  times, replace  $R_{max}$  with observed reward
- Linear total regret when being combined with ( $\epsilon$ -)greedy  
(no guarantee that optimal action will be taken after some time in every state)

## Example



[http://procaccia.info/courses/15381f16/slides/781\\_rl\\_rmax.pdf](http://procaccia.info/courses/15381f16/slides/781_rl_rmax.pdf)

# Exploration and exploitation

## Decaying $\epsilon$ -greedy exploration

- Decrease  $\epsilon$  of  $\epsilon$ -greedy exploration over time according to a special schedule
- Recap:  $\epsilon$ -greedy exploration

$$\pi(a|s) = \begin{cases} 1 - \epsilon_t + \frac{\epsilon_t}{|\mathcal{A}(s)|} & \text{if } a \text{ maximizes } Q(s, a). \\ \frac{\epsilon_t}{|\mathcal{A}(s)|} & \text{for all other actions.} \end{cases}$$

- Decaying  $\epsilon$ -greedy exploration: Update  $\epsilon$  according to

$$\epsilon_t = \min\left(1, \frac{c}{d^2 t}\right)$$

c: tunable parameter  $> 0$   
 t: time step  
 d: minimum gap  $\min_a \Delta_a$

- Logarithmic asymptotic total regret but useless in practice, because gaps  $\Delta_a$  are not known

# Exploration and exploitation

## UCB1 (upper confidence bound)

- Estimate an upper confidence for each action value measuring the uncertainty of the true Q-function
- Depends on the number  $N_t(a)$  than an action has been selected until time  $t$ 
  - small  $N_t(a) \rightarrow$  estimated value is uncertain
  - large  $N_t(a) \rightarrow$  estimated value is certain
- UCB1 algorithm: Select action maximizing upper confidence bound as

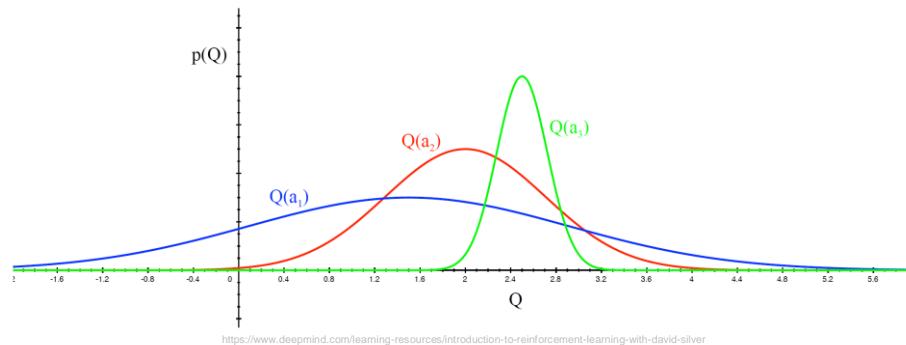
$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} \left( Q(a) + \sqrt{\frac{2 \ln t}{N_t(a)}} \right)$$

- Logarithmic asymptotic total regret (but larger than lower bound), easy to implement
- Standard for a lot of practical problems

# Exploration and exploitation

## Thompson Sampling

- "randomly take action according to the probability you believe it is the optimal action" (Thompson 1933)
- Use internal (probabilistic) model to approximate the true Q-function
- At every time step
  - Sample from the approximated Q-function for every action
  - Select action with best sample outcome
  - Observe reward, update Q-function approximation
- Logarithmic asymptotic total regret (achieving lower bound)



# Exploration and exploitation

Example: Thompson sampling

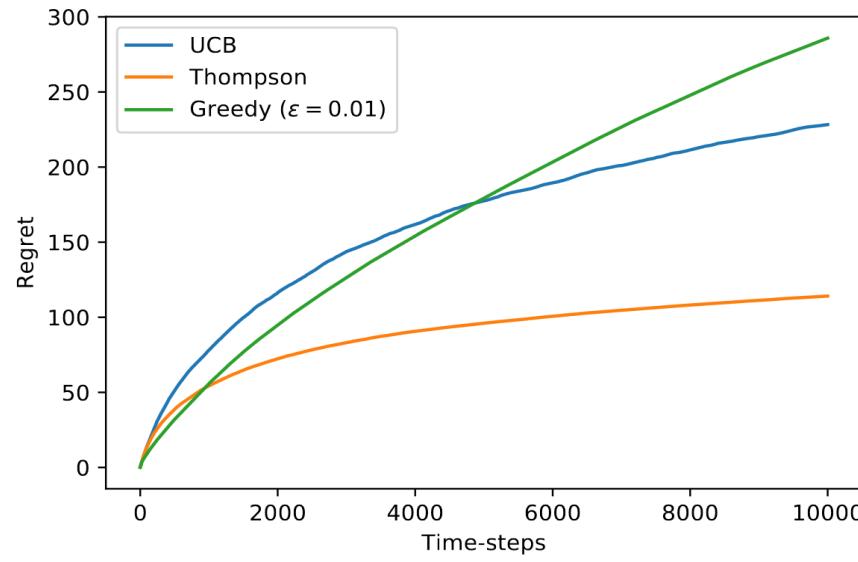
- 2 Bernoulli bandits (reward of 1 with prob.  $\mu_i$ , otherwise 0)
- $\mu = [0.6, 0.4]$
- Rewards obtained so far shown on right side
- At time step  $t = 14$ 
  - Approximate both bandits with e.g. [beta-distribution](#)  
Bandit 1:  $\alpha = 4$ ,  $\beta = 2$   
Bandit 2:  $\alpha = 3$ ,  $\beta = 4$
  - Sample from approximated Q-function for both bandits, e.g.  
Bandit 1:  $Q_{sampled} = 0.68$   
Bandit 2:  $Q_{sampled} = 0.56$
  - Select action with best sample outcome → Bandit 1
  - Observe reward of 1, update Q-function approximation  
Bandit 1:  $\alpha = 5$ ,  $\beta = 2$   
Bandit 2:  $\alpha = 3$ ,  $\beta = 4$

time	bandit	result
1	1	1
2	1	0
3	2	1
4	2	1
5	1	1
6	2	1
7	2	1
8	2	0
9	1	1
10	2	0
11	1	0
12	2	0
13	1	1

# Exploration and exploitation

## Example

- 5 Bernoulli bandits (give reward of 1 with probability  $\mu_i$ , otherwise 0)
- $\mu = [0.6, 0.5, 0.4, 0.3, 0.2]$
- Shown is total regret over number of pulls



<https://team.inria.fr/polaris/files/2019/05/Gast.pdf>

# Exploration and exploitation

Example: [Comparison of  \$\epsilon\$ -greedy / UCB / Thompson](#)

Example: [Comparison of  \$\epsilon\$ -greedy / UCB / Thompson](#)

Example: [Comparison of  \$\epsilon\$ -greedy / UCB](#)

# Exploration and exploitation

Comparison of all presented methods

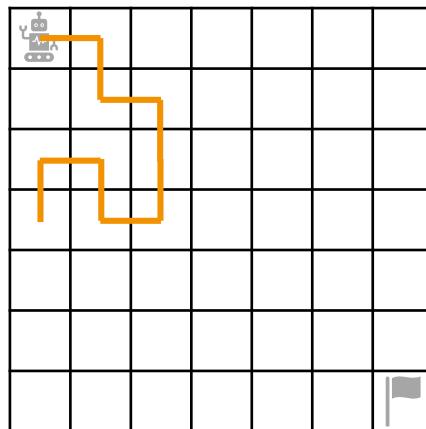
Method	Pro	Con
Optimistic initialization	Simple, works well in practice	Linear total regret
R-max	Simple, works well in practice?	Linear total regret
$\epsilon$ -greedy	Simple, works well in practice	Linear total regret, param. $\epsilon$
decaying $\epsilon$ -greedy	Near optimal	Unfeasible in practice
UCB1	Simple, works ok, near optimal	
Thompson sampling	Optimal	Complex for generic MDPs

# Exploration and exploitation

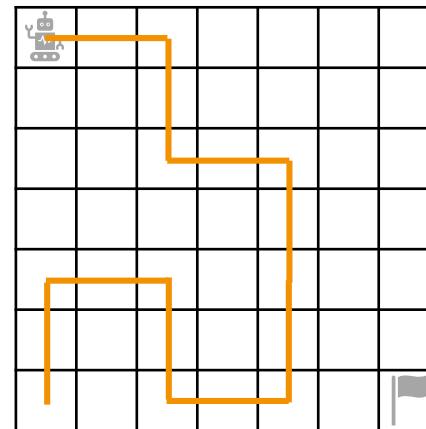
## Other exploration techniques

### Multi-step actions

- Execute the same action for n steps
- Well suited for continuous environments



1-step action



2-step action

# Exploration and exploitation

## Count-based exploration

- Informal: Count how often you have been in each state and try to explore states that you haven't been to many times yet
- Use additive reward term of the form

$$R = \alpha R_{ext} + \beta R_{int}$$

to favor exploration of new states (with weighting factors  $\alpha, \beta$ )

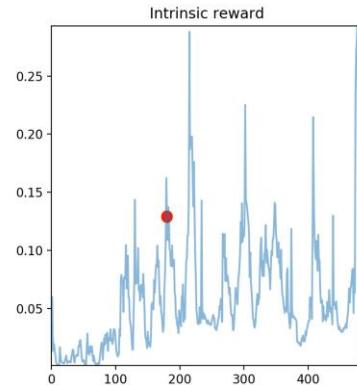
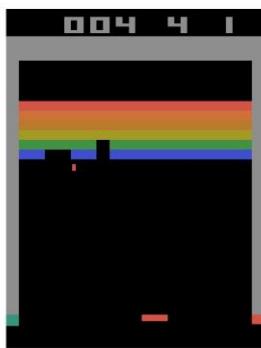
- $R_{ext}$  (extrinsic reward) is the original reward from the RL task (extrinsic reward)
- $R_{int}$  (intrinsic reward) is a count-based exploration reward (high if state hasn't been visited often)
- Easy to implement for discrete states (need to store how often each state has been visited) but can be generalized also to continuous states

# Exploration and exploitation

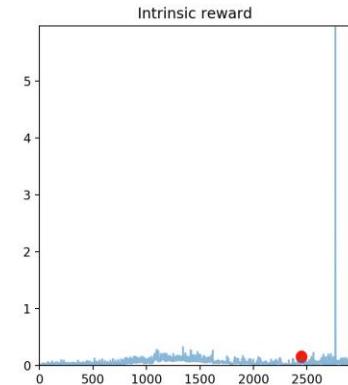
## Prediction-based rewards

- Train a model to predict the outcome of the next action
- If the predicted outcome differs heavily from the true outcome (due to new states/rewards never seen before), assign a high intrinsic  $R_{int}$  reward to the corresponding action (similar to count-based exploration)

intrinsic reward at the beginning



intrinsic reward when passing the level for the first time



<https://openai.com/blog/reinforcement-learning-with-prediction-based-rewards/>

Note: Both prediction- and count-based rewards belong to the class of curiosity driven exploration

# Exploration and exploitation

## Auxiliary tasks

- Fancy variant of reward shaping: Consider additional tasks (with corresponding rewards) that the agent has to solve along its way of solving the main task

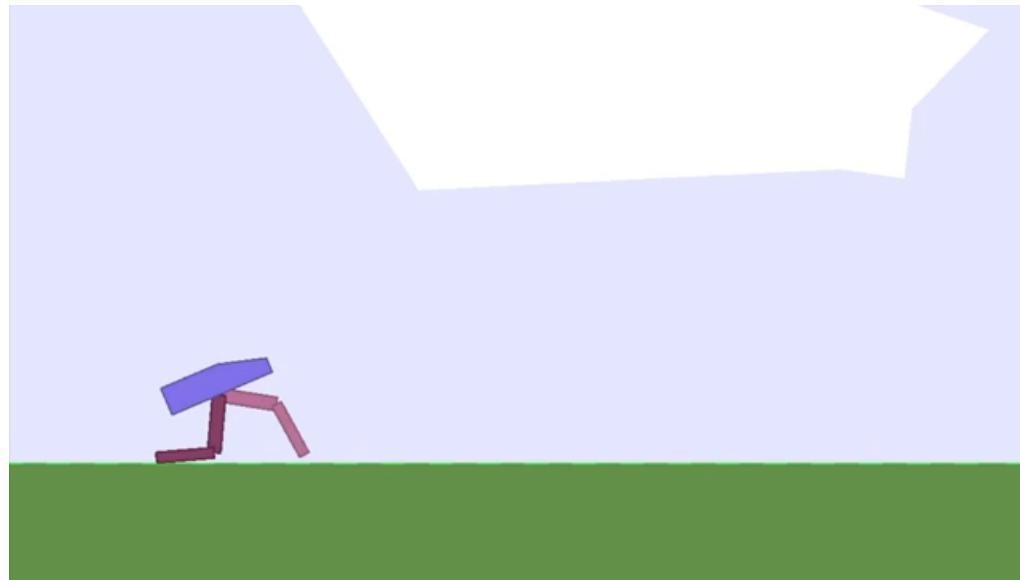
## Example

- Main task: Robot has to open a box, place an object into it and close the lid of the box afterwards again
- Auxiliary task 1: Box has been opened
- Auxiliary task 2: Object has been placed in the box
- Auxiliary task 3: Box has been closed

# Exploration and exploitation

## Curriculum learning

- Problem: Sometimes exploration does not help at all because the task is too difficult to be solved by an untrained agent
- Solution: Create a curriculum of environments, i.e. environments with increasing complexity until the previously unsolvable task can be solved



<https://www.youtube.com/watch?v=D1WWhQY9N4g>

# Exploration and exploitation

## Brief summary

- Exploration vs. exploitation (sticking to known stuff vs. exploring something new) is a predominant problem in reinforcement learning
- There exist multiple methods with different implementation complexity favoring exploration
- A good overview of different methods is provided [here](#)

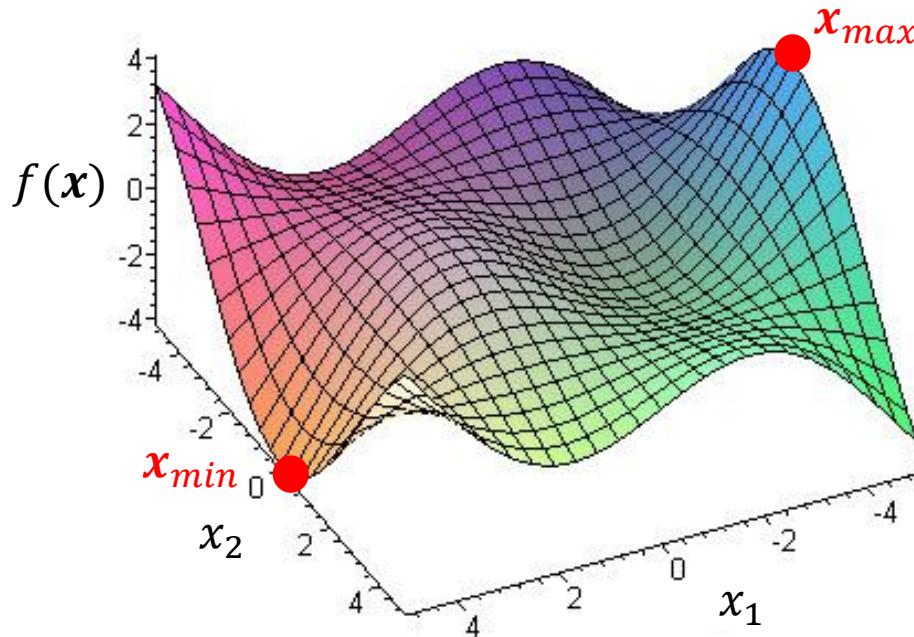
# Kahoot!

# Gradient-based optimization

# Gradient-based optimization

Gradient-based optimization is used to find the minimum/maximum  $x_{min} / x_{max}$  of a differentiable function  $f(x)$  based solely on gradient calculations

Example:



[https://igl.ethz.ch/teaching/tau/cg/cg2005/cg\\_ex6.ppt](https://igl.ethz.ch/teaching/tau/cg/cg2005/cg_ex6.ppt)

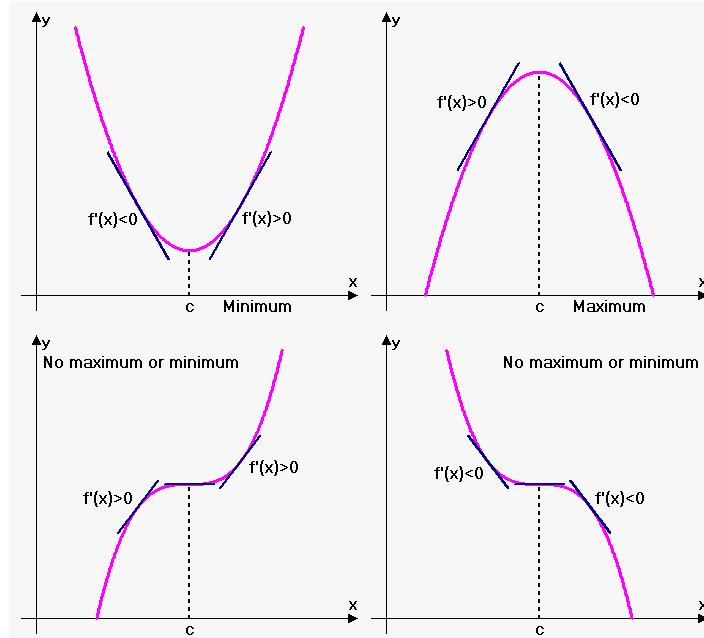
Nomenclature: Symbols/characters written in bold/plain (e.g.  $\mathbf{x}/x$ ) denote vectors (more than one value) / scalars (only one value). Capital letters (e.g.  $A$ ) represent matrices

# Gradient-based optimization

Recap: First-/second derivative of function  $f(x)$

- First derivative  $f'(x)$  calculates the slope of the function  $f(x)$
- Second derivative  $f''(x)$  calculates the rate of change of the first derivative
- First/second derivative can be used to find maxima/minima of a function

Example:



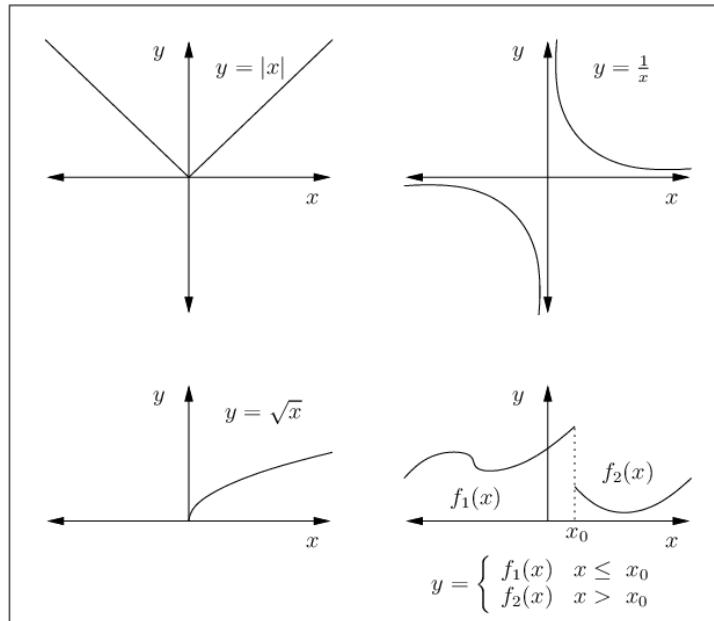
<https://socratic.org/questions/how-do-you-find-local-maximum-value-of-f-using-the-first-and-second-derivative-t-8>

# Gradient-based optimization

Recap: Differentiable function

- First derivative of the function exists and is continuous

Examples for non-differentiable functions over  $\mathbb{R}$



[https://www.researchgate.net/figure/Non-differentiable-functions-for-particular-input\\_fig3\\_30515764](https://www.researchgate.net/figure/Non-differentiable-functions-for-particular-input_fig3_30515764)

Task: Why are the shown functions non-differentiable on  $\mathbb{R}$  ?

# Gradient-based optimization

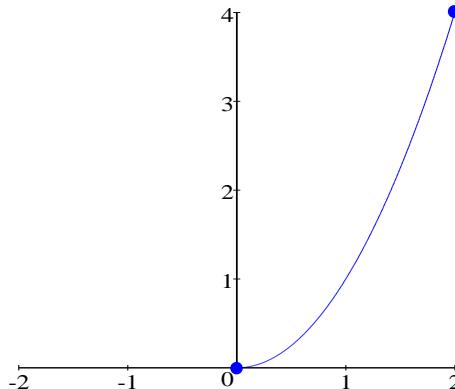
Recap: Existence of a minimum/maximum value  $x_{min}$  /  $x_{max}$

- Extreme value theorem: If  $f(x)$  is continuous over a closed interval (endpoints included), then  $f$  must attain a minimum/maximum  $x_{min}$  /  $x_{max}$  on that interval

Example:

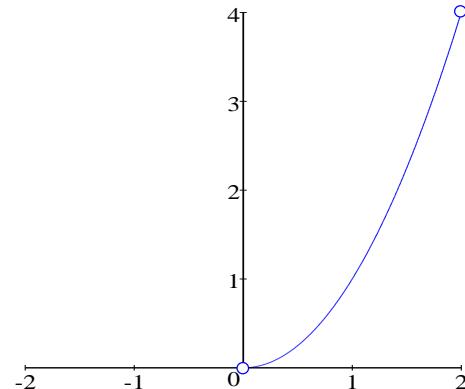
$$f(x) = x^2 \text{ for } x \in [0,2]$$

$x_{min}$  exists  
 $x_{max}$  exists



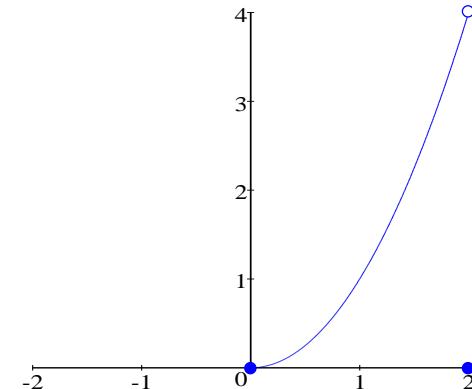
$$f(x) = x^2 \text{ for } x \in ]0,2[$$

$x_{min}$  does not exist  
 $x_{max}$  does not exist



$$f(x) = \begin{cases} x^2 & \text{for } x \in [0,2[ \\ 0 & \text{for } x = 2 \end{cases}$$

$x_{min}$  exists  
 $x_{max}$  does not exist

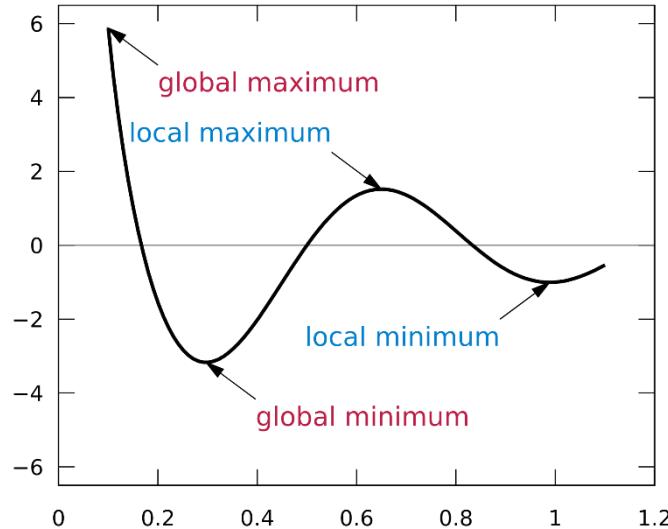


# Gradient-based optimization

Recap: Global/local minimum/maximum

- $x$  is a local minimum/maximum if  $f(x)$  is the smallest/largest value within a given local range of  $x$
- $x$  is a global minimum/maximum if  $f(x)$  is the smallest/largest value on the entire definition range of  $x$

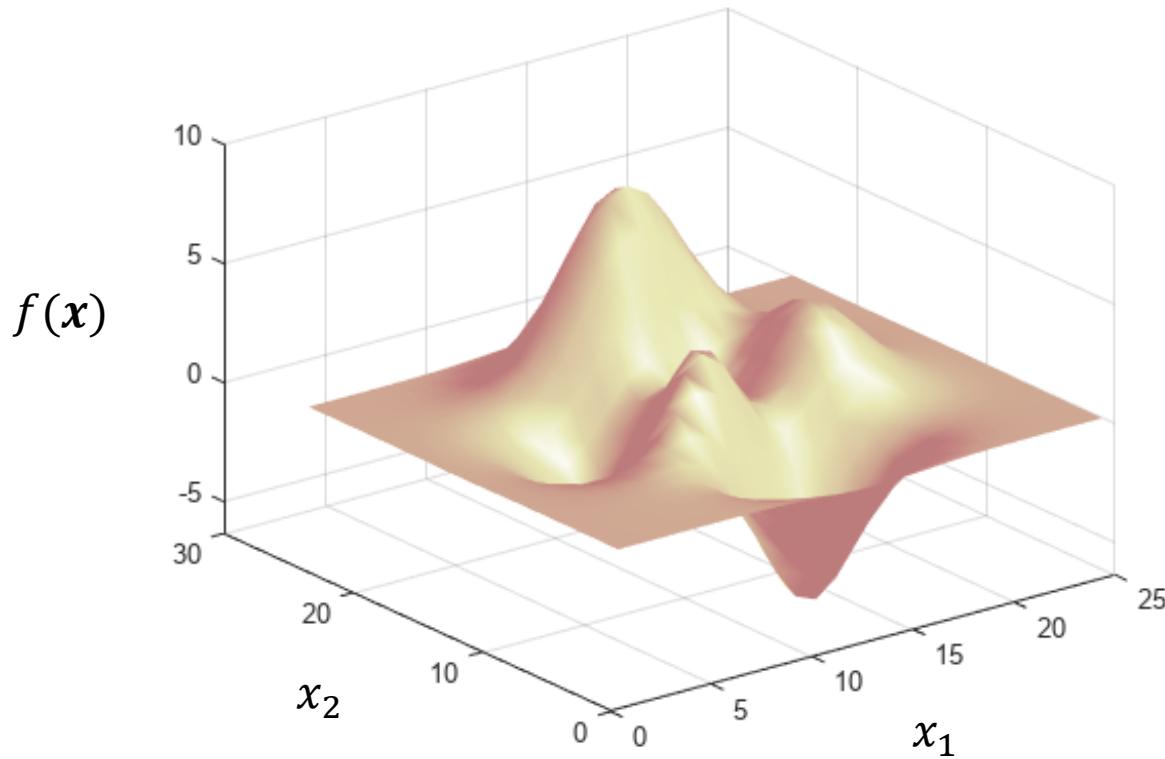
Example:



[https://en.wikipedia.org/wiki/Maxima\\_and\\_minima#/media/File:Extrema\\_example\\_original.svg](https://en.wikipedia.org/wiki/Maxima_and_minima#/media/File:Extrema_example_original.svg)

# Gradient-based optimization

Task: Shown is the following function. Where are local/global maxima/minima?



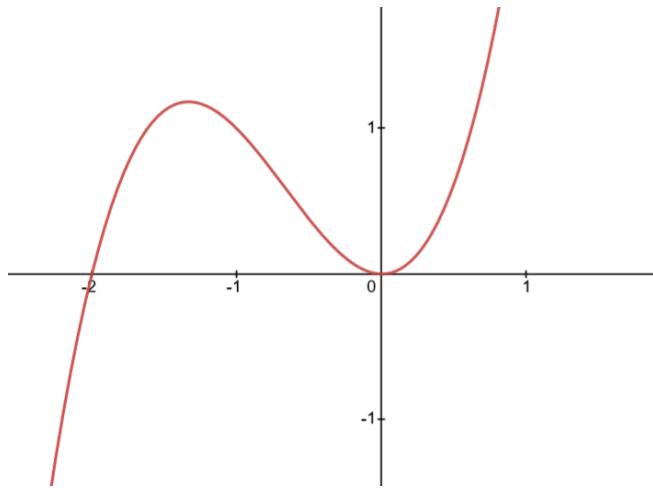
# Gradient-based optimization

Scalar vs. vector representation

- $f(x)$  can either depend on one variable (then  $x$  is written non-bold, e.g.  $f(x)$ ) or multiple variables (then  $x$  is written bold, e.g.  $f(\mathbf{x})$ )

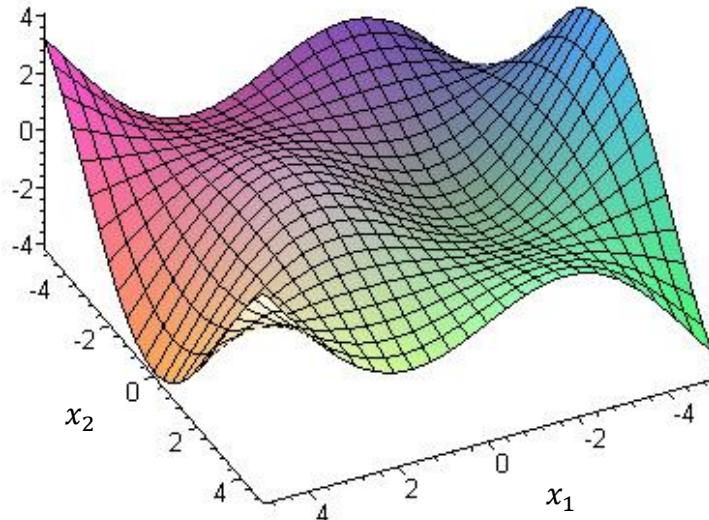
Example:

$$f(x) = 2x^2 + x^3$$



<https://www.desmos.com/calculator?lang=de>

$$f(\mathbf{x}) = x_1 \cos\left(\frac{x_1}{2}\right) \cos\left(\frac{x_2}{2}\right)$$



[https://igl.ethz.ch/teaching/tau/cg/cg2005/cg\\_ex6.ppt](https://igl.ethz.ch/teaching/tau/cg/cg2005/cg_ex6.ppt)

# Gradient-based optimization

## Partial derivative

- The partial derivative (or directional derivative along the axes) calculates the slope of  $f(x)$  along the  $x_1 / x_2 / \dots$  direction
- For the i-th variable  $x_i$  it is denoted

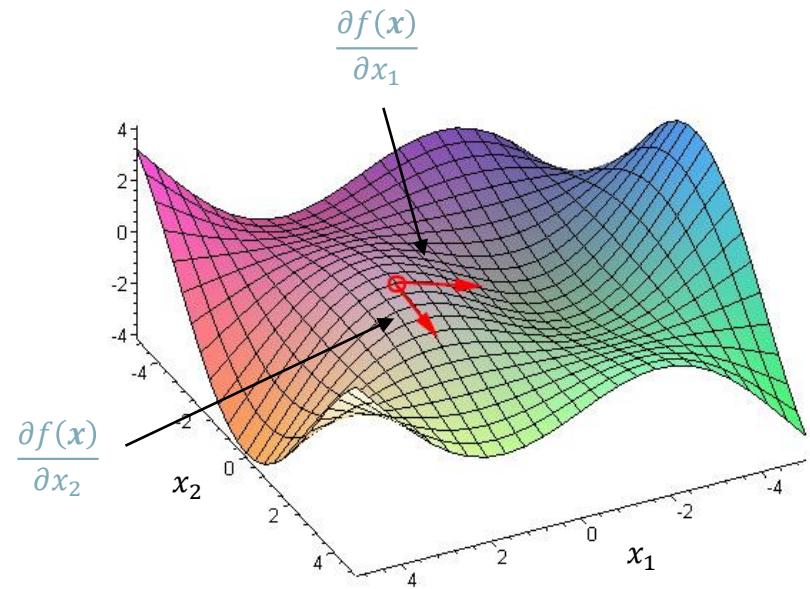
$$\frac{\partial f(x)}{\partial x_i}$$

Example:

$$f(x) = x_1 \cos\left(\frac{x_1}{2}\right) \cos\left(\frac{x_2}{2}\right)$$

$$\frac{\partial f(x)}{\partial x_1} = \cos\left(\frac{x_1}{2}\right) \cos\left(\frac{x_2}{2}\right) - \frac{1}{2} x_1 \sin\left(\frac{x_1}{2}\right) \cos\left(\frac{x_2}{2}\right)$$

$$\frac{\partial f(x)}{\partial x_2} = -\frac{1}{2} x_1 \cos\left(\frac{x_1}{2}\right) \sin\left(\frac{x_2}{2}\right)$$



[https://igl.ethz.ch/teaching/tau/cg/cg2005/cg\\_ex6.ppt](https://igl.ethz.ch/teaching/tau/cg/cg2005/cg_ex6.ppt)

# Gradient-based optimization

## Gradient

- Calculating all partial derivatives at once ad concatenating them in a vector yields the gradient  $\nabla f(\mathbf{x})$  of a function  $f(\mathbf{x})$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \end{bmatrix}$$

Example:

$$f(\mathbf{x}) = x_1 \cos\left(\frac{x_1}{2}\right) \cos\left(\frac{x_2}{2}\right)$$

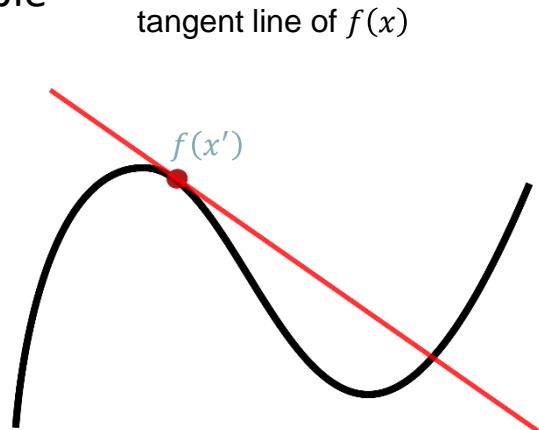
$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \cos\left(\frac{x_1}{2}\right) \cos\left(\frac{x_2}{2}\right) - \frac{1}{2} x_1 \sin\left(\frac{x_1}{2}\right) \cos\left(\frac{x_2}{2}\right) \\ -\frac{1}{2} x_1 \cos\left(\frac{x_1}{2}\right) \sin\left(\frac{x_2}{2}\right) \end{bmatrix}$$

# Gradient-based optimization

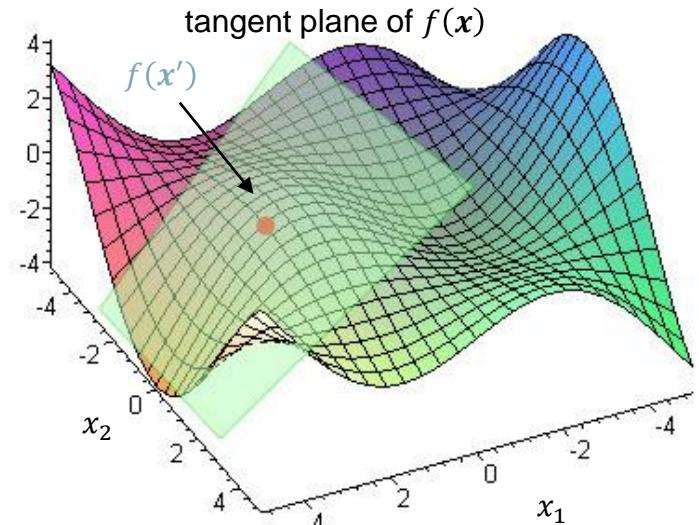
- First interpretation of the gradient: The partial derivatives within the gradient define a hyperplane approximating the function for any given point  $\mathbf{x}'$  (similar to tangent plane for a scalar  $x$ )

$$\begin{aligned}f(\mathbf{x}) &\approx f(\mathbf{x}') + \frac{\partial f(\mathbf{x}')}{\partial x_1} \cdot (x_1 - x'_1) + \frac{\partial f(\mathbf{x}')}{\partial x_2} \cdot (x_2 - x'_2) + \dots \\&\approx f(\mathbf{x}') + \nabla f(\mathbf{x}') \cdot (\mathbf{x} - \mathbf{x}')\end{aligned}$$

## Example



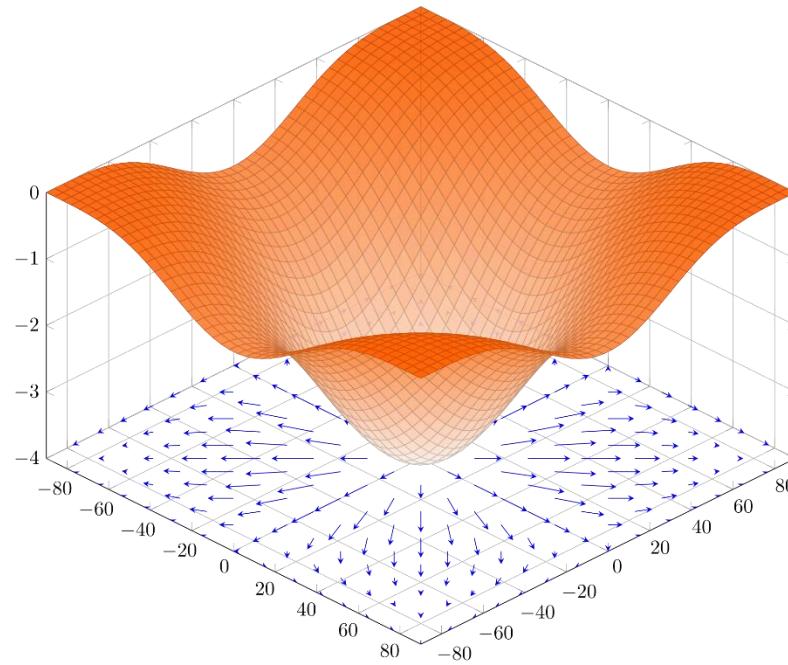
[https://en.wikipedia.org/wiki/Tangent#/media/File:Tangent\\_to\\_a\\_curve.svg](https://en.wikipedia.org/wiki/Tangent#/media/File:Tangent_to_a_curve.svg)



[https://igl.ethz.ch/teaching/tau/cg/cg2005/cg\\_ex6.ppt](https://igl.ethz.ch/teaching/tau/cg/cg2005/cg_ex6.ppt)

# Gradient-based optimization

- Second interpretation of the gradient: The gradient defines the direction of steepest ascent (in which direction  $f(x)$  increases the most)
- Key idea for gradient-based optimization: Always "go" in positive/negative gradient direction to maximize/minimize  $f(x)$  and to find  $x_{max} / x_{min}$



<https://en.wikipedia.org/wiki/Gradient#/media/File:3d-gradient-cos.svg>

# Gradient-based optimization

- Note: If the function depends on multiple inputs (e.g.  $f(\mathbf{x}, \mathbf{w})$ ) and we want to calculate the gradient only with respect to a specific input (e.g.  $\mathbf{w}$ ), it can be written with a subscript (e.g.  $\nabla_{\mathbf{w}}$ )

Example

$$\nabla_{\mathbf{w}} f(\mathbf{x}, \mathbf{w}) = \begin{bmatrix} \frac{\partial f(\mathbf{x}, \mathbf{w})}{\partial w_1} \\ \frac{\partial f(\mathbf{x}, \mathbf{w})}{\partial w_2} \\ \vdots \end{bmatrix}$$

# Gradient-based optimization

Task: Given is the function  $f(\mathbf{x}, \mathbf{w})$  as

- a)  $f(\mathbf{x}, \mathbf{w}) = x_1^2 + x_2^4 - e^{w_1}$
- b)  $f(\mathbf{x}, \mathbf{w}) = x_1 \sqrt{x_2^2 + w_1^2 + \ln w_2}$
- c)  $f(\mathbf{x}, \mathbf{w}) = \frac{\sin(x_1^2 + x_2)}{\operatorname{atan} w_1 - e^{w_2}}$

Calculate the gradients  $\nabla f(\mathbf{x}, \mathbf{w})$ ,  $\nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$ ,  $\nabla_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})$  and the partial derivative  $\frac{\partial f(\mathbf{x}, \mathbf{w})}{\partial x_1}$

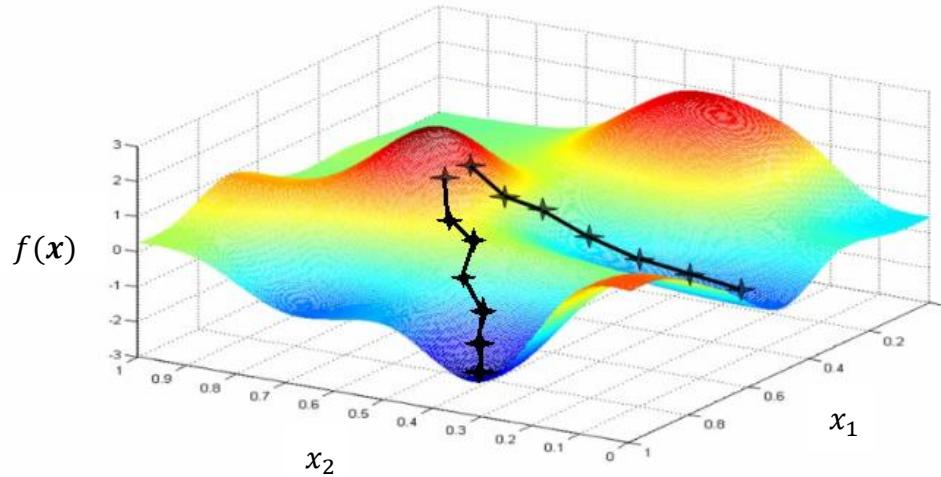
# Gradient-based optimization

## Gradient descent/ascent

- Iteratively move along negative/positive gradient direction to minimize/maximize  $f(\mathbf{x})$
- Calculate  $\mathbf{x}_k$  based on  $\mathbf{x}_{k-1}$  (with learning rate  $\eta$  defining the step size)

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \eta \cdot \nabla f(\mathbf{x}) \quad \text{for minimizing } f(\mathbf{x})$$
$$\mathbf{x}_k = \mathbf{x}_{k-1} + \eta \cdot \nabla f(\mathbf{x}) \quad \text{for maximizing } f(\mathbf{x})$$

## Example

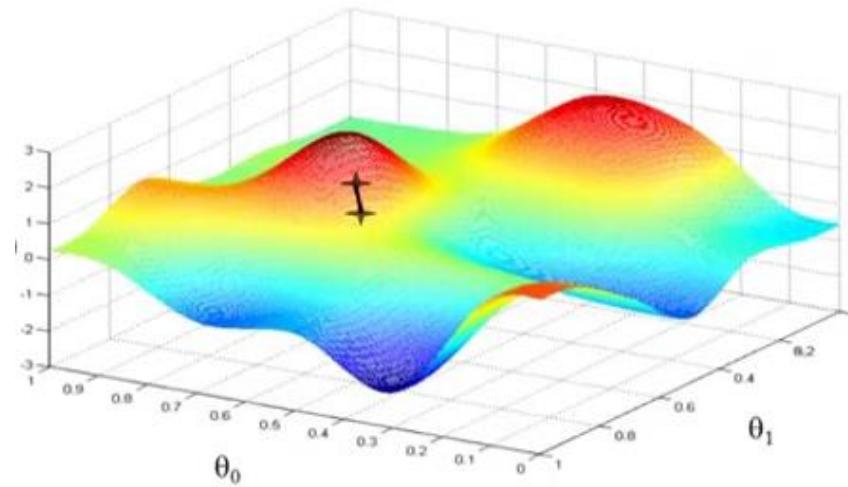


<https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/>

# Gradient-based optimization

Comments on gradient descent

- Maximizing over a function  $f(x)$  equals minimizing over  $-f(x)$
- Many improvements available to basic gradient descent equation:
  - Momentum, specialized optimizers (e.g. Adam), gradient noise, early stopping, restarts, ...  
→ Deep learning lecture
  - Consider also higher-order derivatives (Hessian matrix), quadratic approximation, ...  
→ Nonlinear optimization lecture (not offered at OTH)



<https://gifycat.com/angryinconsequentialdipodocus>

# Gradient-based optimization

Task: Calculate two gradient descent steps for the function

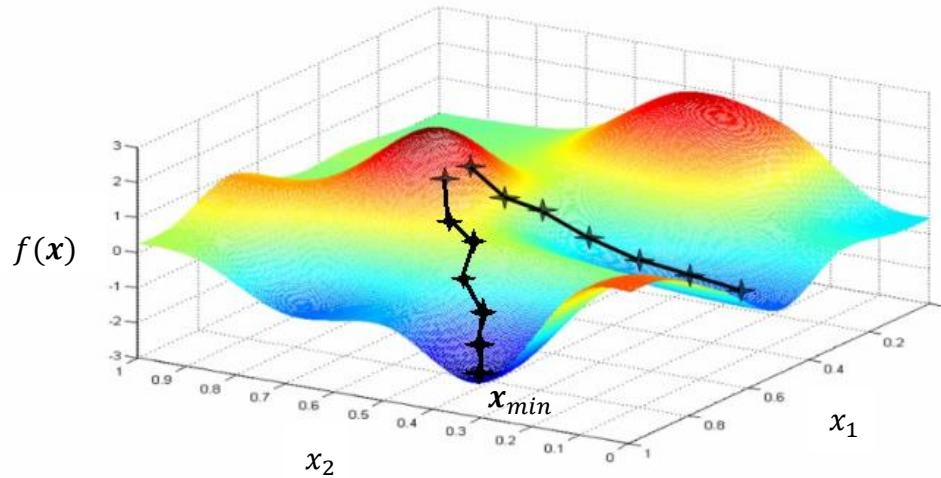
$$f(\mathbf{x}) = x^2 - y + \sin(z)$$

for the starting point  $\mathbf{x} = [x, y, z]^T = [1, 2, 3]^T$  and a learning rate  $\eta = 0.1$

# Gradient-based optimization

## Brief summary

- The gradient is the partial derivative of a function  $f(\mathbf{x})$  along every component of  $\mathbf{x}$
- The gradient is the generalization of the first derivative  $f'(x)$  known from high school if a function  $f(\mathbf{x})$  depends on more than one variable
- Gradient descent is an iterative procedure to obtain the minimum value of a function  $f(\mathbf{x})$  and the corresponding value  $\mathbf{x}_{min}$



<https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/>

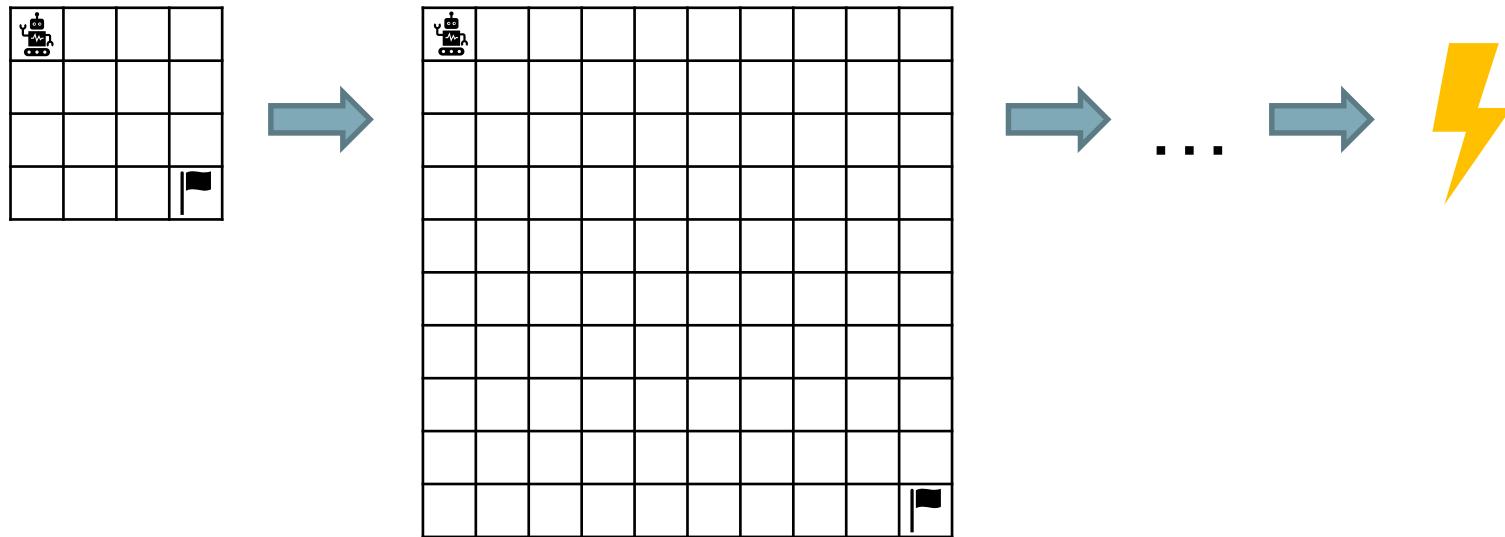
# Kahoot!

# Function approximators

# Function approximators

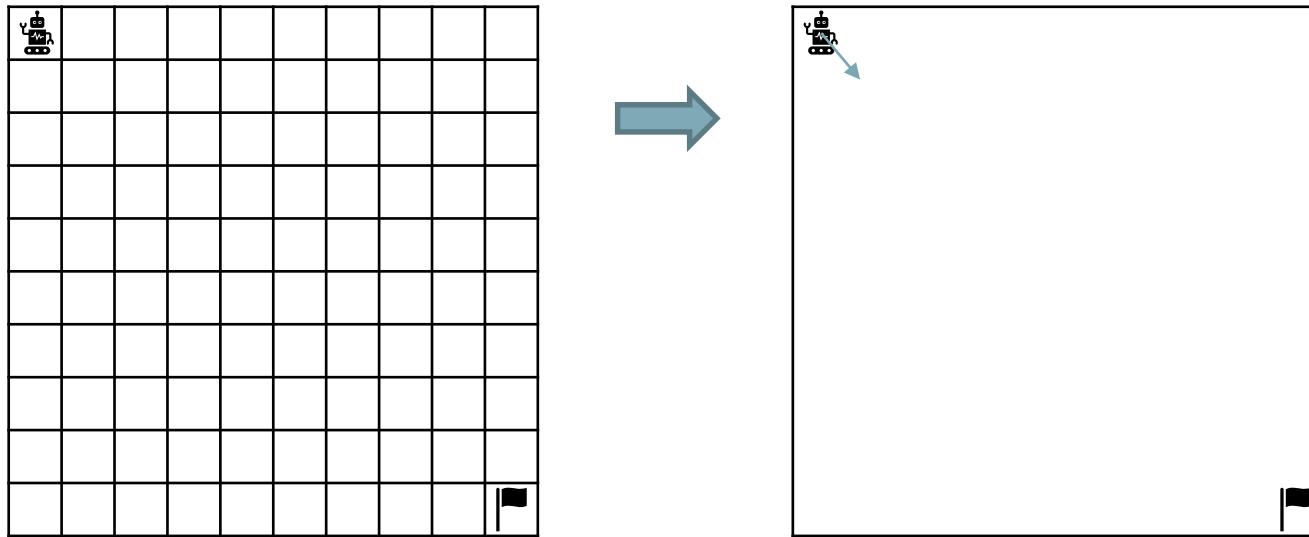
Challenge: How to deal with increasingly complex environments?

- Problem 1: Large state/action spaces make it increasingly difficult to explore all states/actions and update the (V-)/Q-function properly  
(Chess has  $10^{50}$  legal board positions, Go has  $10^{170}$ )



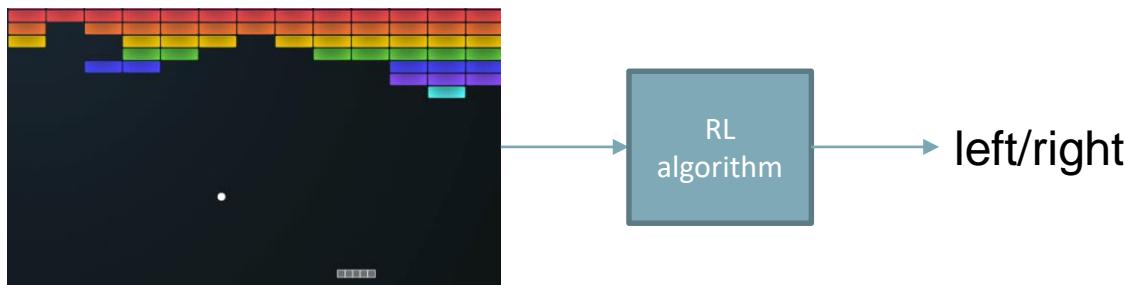
# Function approximators

- Problem 2: How to deal with continuous state/action spaces?  
(extreme case of "large state/action space")



# Function approximators

- Problem 3: How to deal with high-dimensional states?  
(e.g. image as state)



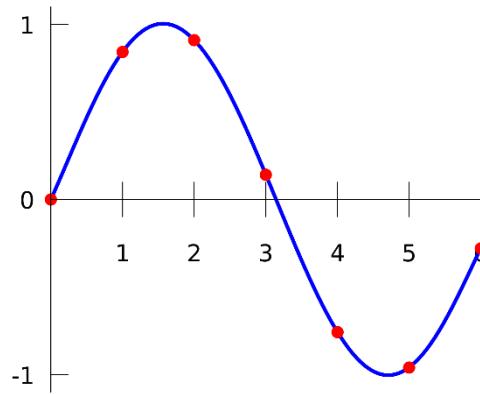
<https://www.coolmathgames.com/sites/default/files/Alan%20Breakout%20OG%20image.png>

# Function approximators

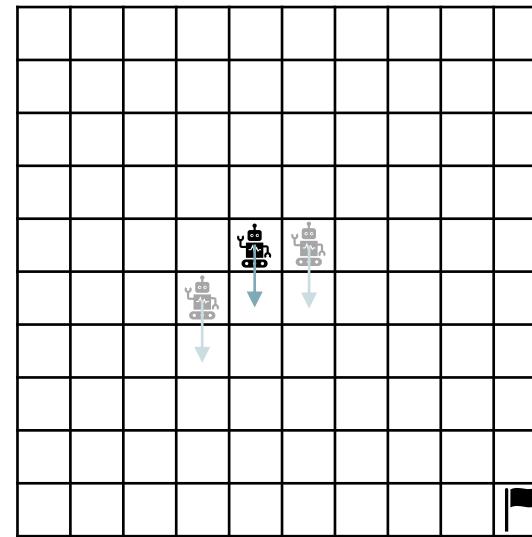
## Solution: Function approximations

- Approximate V-/Q-function or policy by a parameterizable function (linear model, neural network, decision tree, support vector machine, ...)
- Allows one to interpolate V-/Q-function or policy for previously unseen states

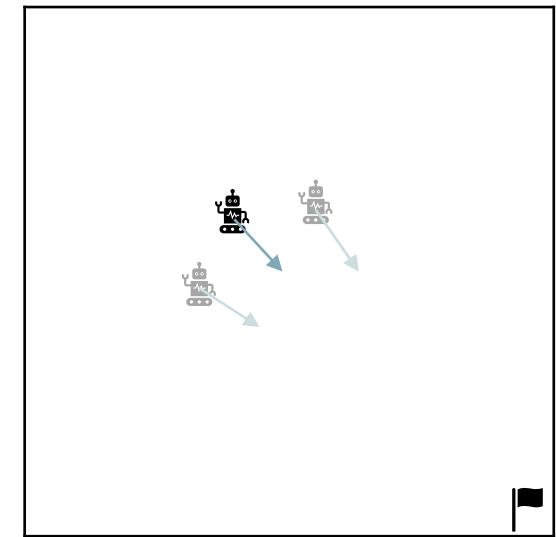
Function approximation:  
general case



Function approximation:  
discrete case, policy approximation



Function approximation:  
continuous case, policy approximation

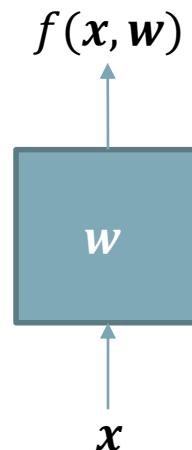


# Function approximators

## General structure of function approximators

$$\hat{y} = f(x, w)$$

- Function  $f$  calculates an output  $\hat{y}$  based on an input  $x$  and parameterizable weights  $w$
- Approximated/predicted output  $\hat{y}$  matches the true output  $y$  as good as possible
- The weights  $w$  are optimized through training



# Function approximators

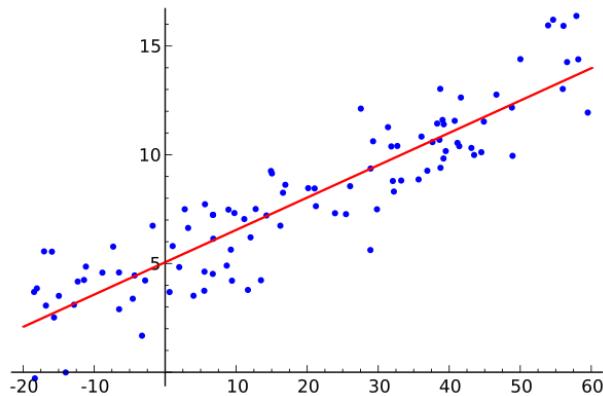
## Example of function approximators

- Linear regression (general form,  $A$  and  $\mathbf{b}$  represent the weights  $\mathbf{w}$ )

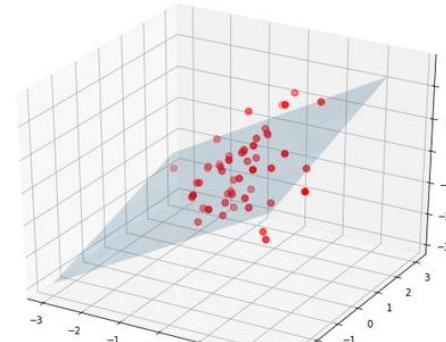
$$\hat{\mathbf{y}} = A\mathbf{x} + \mathbf{b}$$

- used for linear relationships between input/output variables

Linear regression: One input, one output ( $y = ax + b$ )



Linear regression: Two inputs, one output ( $y = [a_1, a_2] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b$ )

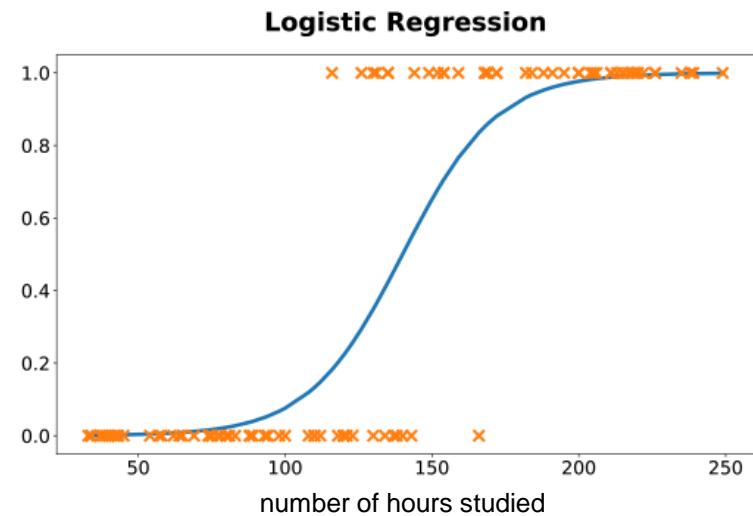
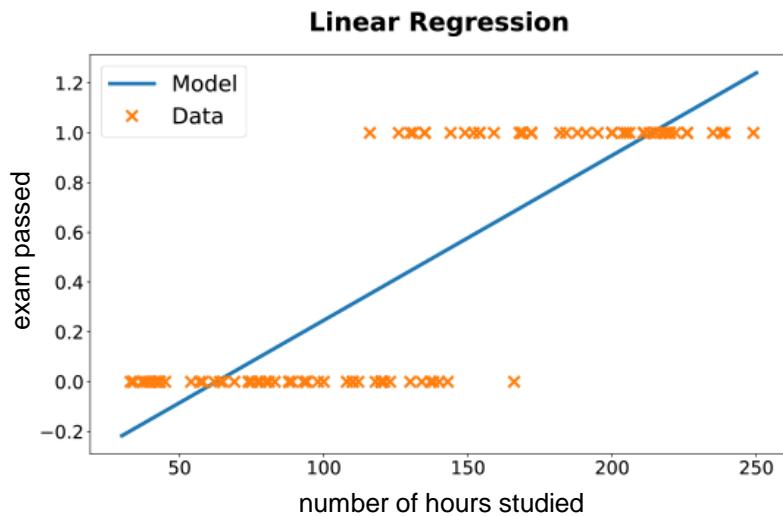


# Function approximators

- Logistic regression

$$\hat{y} = \frac{1}{1 + \exp(-x^T w)}$$

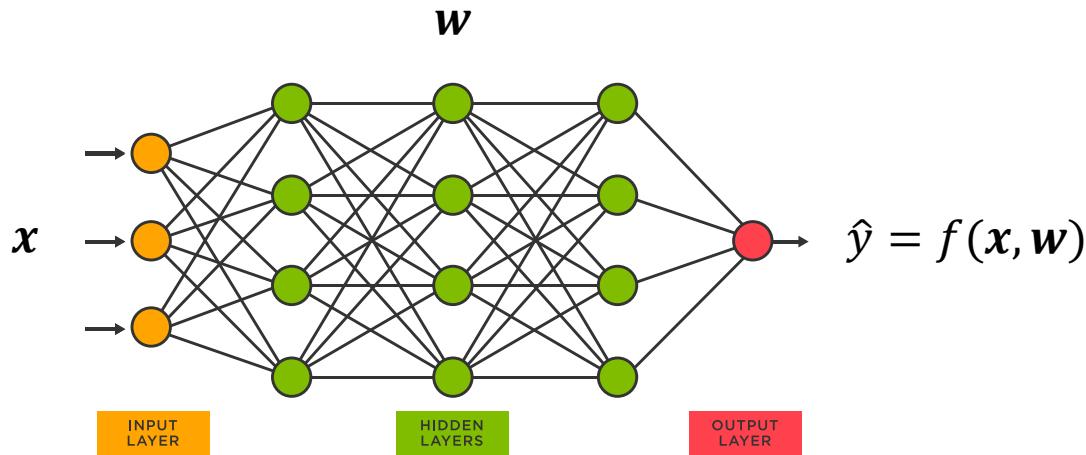
- used for simple binary classification tasks



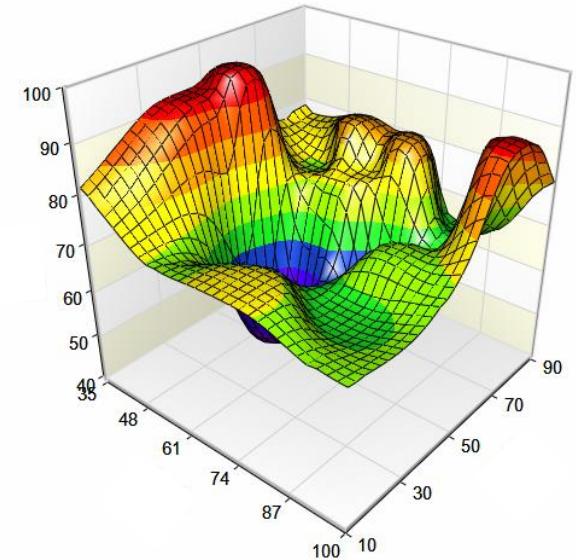
<https://content-garden.com/wp-content/uploads/2021/03/figure-1-linear-and-logistic-regression-with-binary-data.png>

# Function approximators

- Neural networks are able to approximate highly nonlinear relationships
- They consist of multiple layers with trainable weights  $w$
- state-of-the-art performance for many tasks



<https://www.tibco.com/reference-center/what-is-a-neural-network>

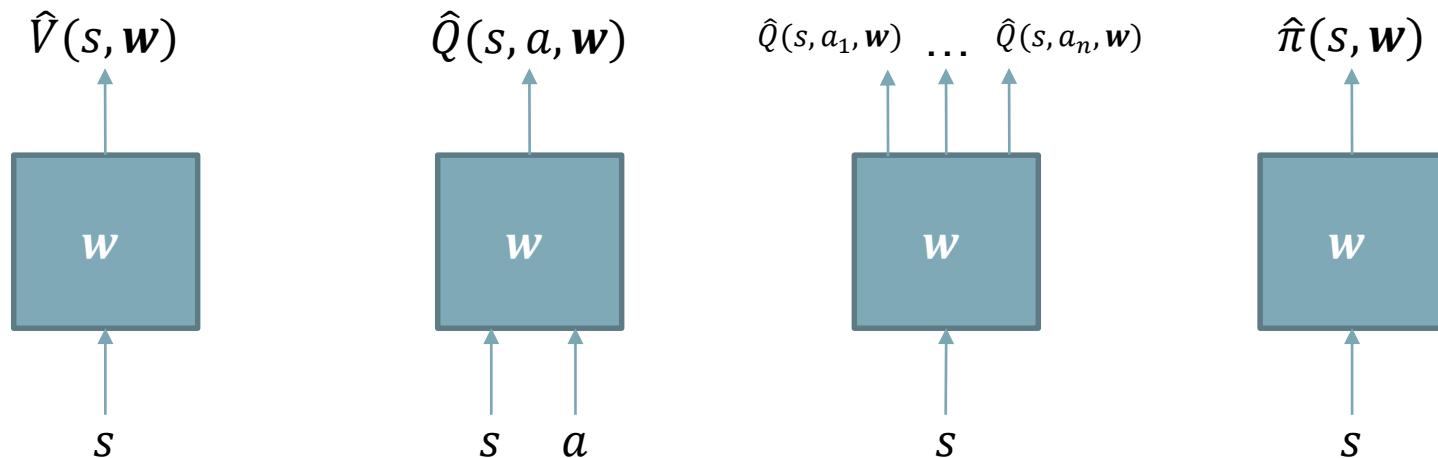


<https://www.ncss.com/software/ncss/ncss-plots-and-graphs/>

# Function approximators

Interesting function approximators for reinforcement learning

- $\hat{V}(s, \mathbf{w})$  approximates the V-function for a given state
- $\hat{Q}(s, a, \mathbf{w})$  approximates the Q-function for a given state/action
- $\hat{Q}(s, a_1, \mathbf{w}) \dots \hat{Q}(s, a_n, \mathbf{w})$  approximates the Q-function for a given state (discrete case only, one output per action)
- $\hat{\pi}(s, \mathbf{w})$  approximating the policy for a given state



# Function approximators

How to train function approximators (simple example with only one sample)

1. Record data of the form  $x$  with corresponding "true" output  $y$
2. Define a loss function  $L$  measuring the error between the predicted output  $\hat{y}$  and the true output  $y$ , e.g.

$$L = (\hat{y} - y)^2 = (f(x, w) - y)^2$$

3. Calculate the gradient of  $L$  with respect to  $w$

$$\nabla_w L = \nabla(f(x, w) - y)^2 = 2 \cdot (f(x, w) - y) \cdot \nabla f(x, w)$$

4. Optimize (modify) the weights  $w$  of the function approximator through gradient descent to minimize the loss function  $L$  and go to 1.

$$w_k = w_{k-1} - \eta \cdot \nabla_w L$$

Note: Above method can also be generalized to multiple outputs of the form

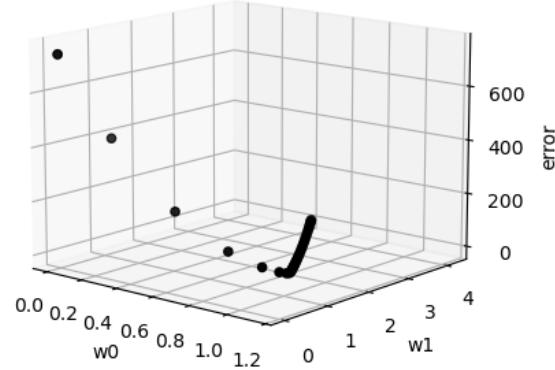
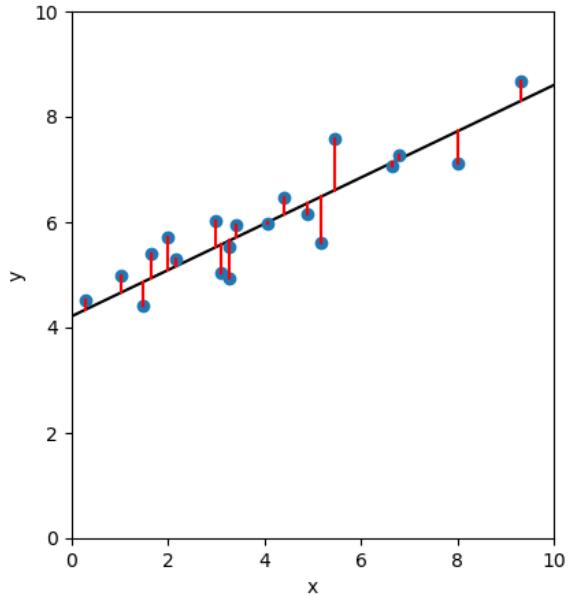
$$L = \sum_i (\hat{y}_i - y_i)^2$$

and/or vector-valued outputs of the form

$$L = (\hat{\mathbf{y}} - \mathbf{y})^T \cdot (\hat{\mathbf{y}} - \mathbf{y})$$

# Function approximators

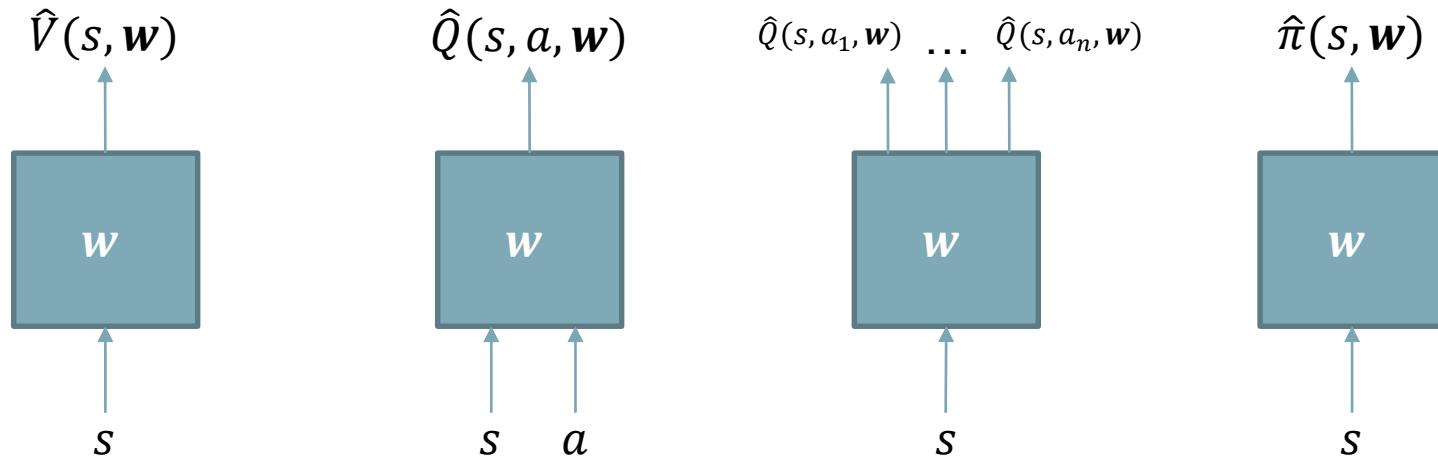
Example: Code



# Function approximators

## Brief summary

- There exist various types of function approximators in general (linear model, neural network, decision tree, support vector machine, ...)
- Only neural networks are interesting for RL as function approximators
- Function approximators approximate the V-/Q-function or the policy in RL
- Function approximators must be trained by minimizing a loss function (often) through gradient descent



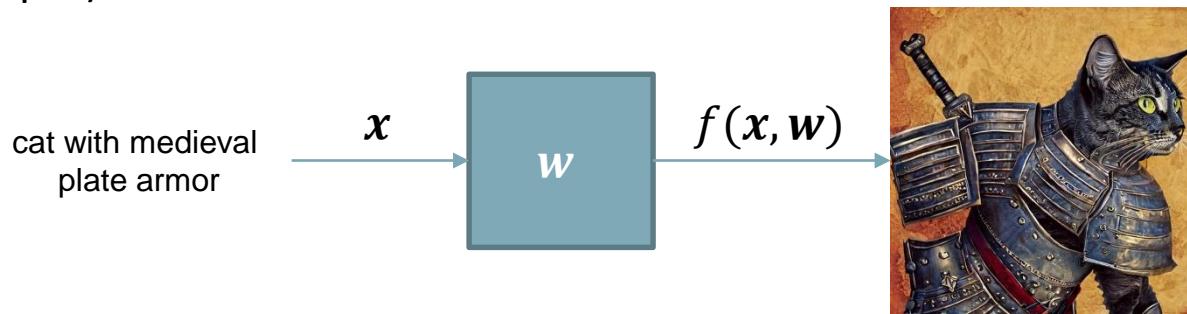
# Kahoot!

# Neural networks

# Neural networks

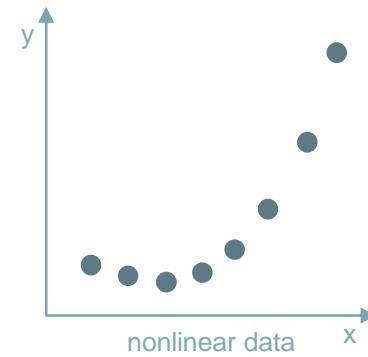
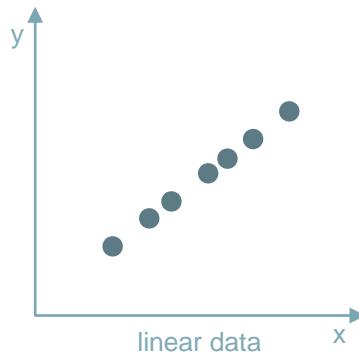
## Neural networks

- Are very powerful function approximators (e.g. with images or sentences as input/output)



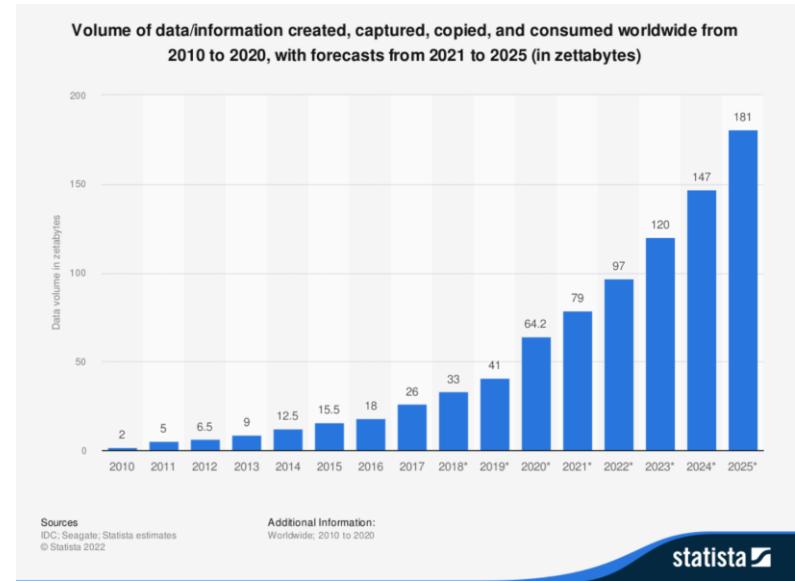
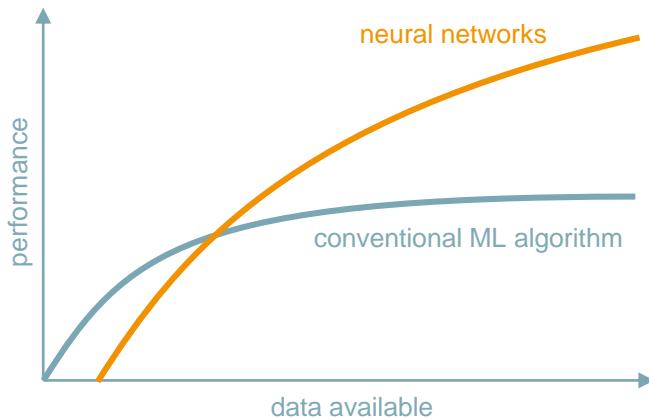
<https://stablediffusionweb.com/#demo>

- Other interpretation: Can approximate highly nonlinear data



# Neural networks

- Can deal with large amounts of data better than other, more conventional machine learning algorithms



# Neural networks

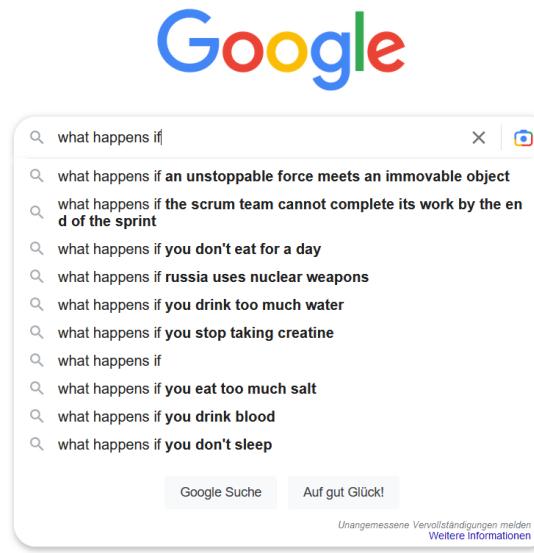
- Are applicable for a wide variety of problems

panoptic segmentation



<https://149695847.v2.pressablecdn.com/wp-content/uploads/2021/02/panoptic-output-1024x683.png>

autocomplete



deep fakes

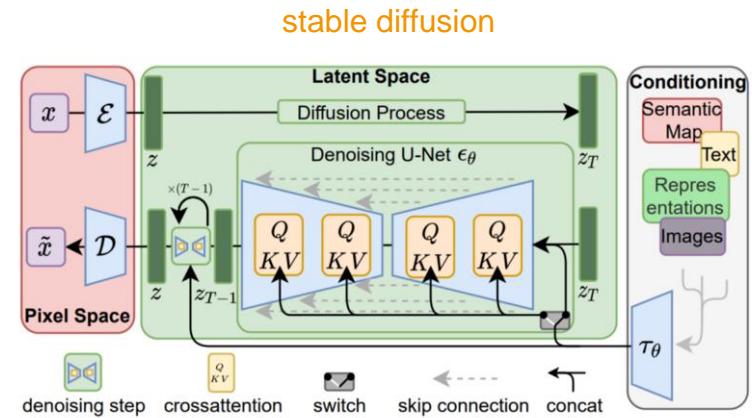
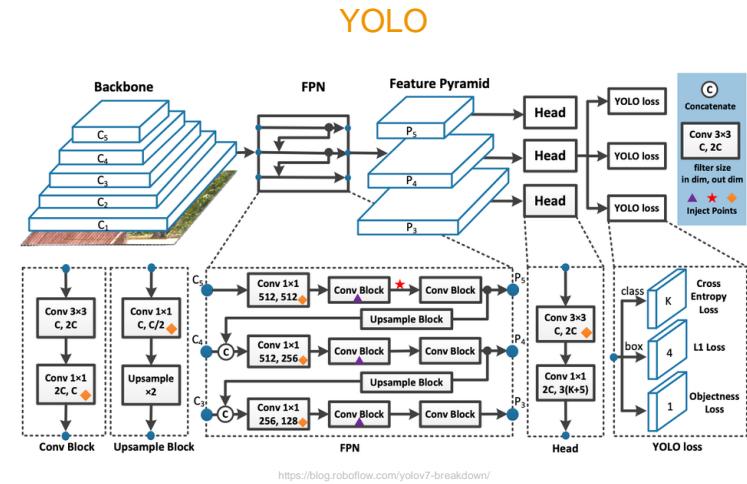
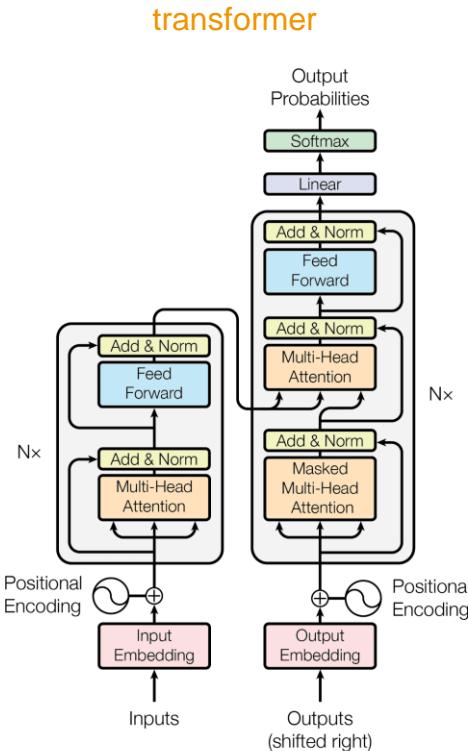


<https://www.youtube.com/watch?v=TgCsJLypZYhttps://www.youtube.com/shorts/LDpSutr5wo>

- Are all trained through gradient descent (**deep learning**)

# Neural networks

Note that there exist myriads of different architectures, we will only consider the most fundamental one, named **multilayer perceptron**



# Neural networks

Neural networks are inspired by nature

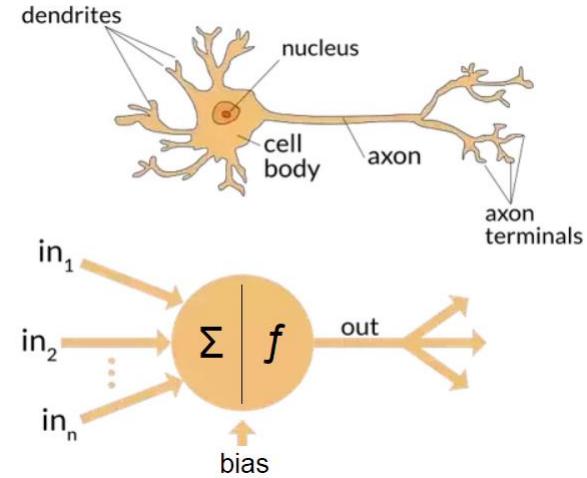
- Neurons receive **input signals**  $x_i$  with **weight**  $w_i$  from other neurons through dendrites
- Those signals are accumulated within the cell body (with **bias**  $b$ )

$$\sum_i w_i x_i + b$$

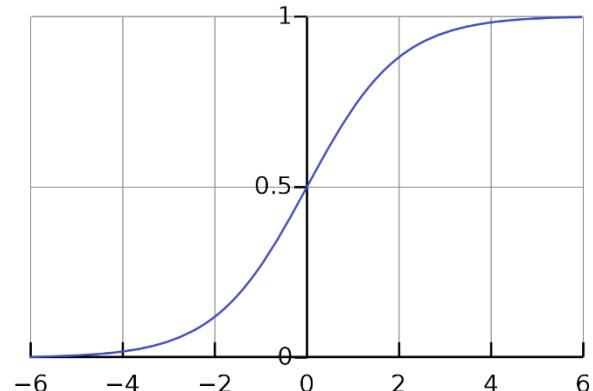
- Once the sum exceeds a certain threshold, the neuron emits a signal through its axon (fires)

$$f(\sum_i w_i x_i + b)$$

- The nonlinear function  $f$  (**activation function**) can e.g. be a sigmoid function
- An **artificial neuron** calculating  $f(\sum_i w_i x_i + b)$  constitutes the basic building block of artificial neural networks



[https://miro.medium.com/max/610/1\\*SJPalPhP4KDEB1AdhOFy\\_Q.png](https://miro.medium.com/max/610/1*SJPalPhP4KDEB1AdhOFy_Q.png)

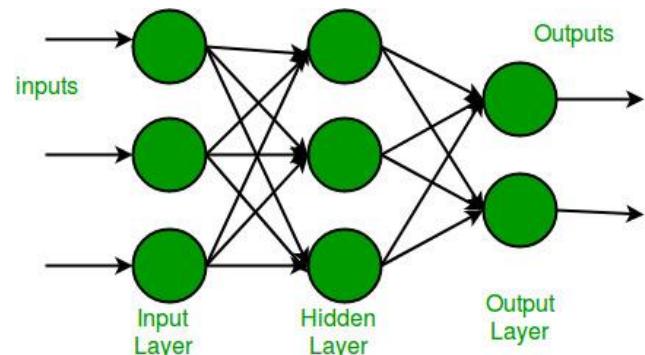
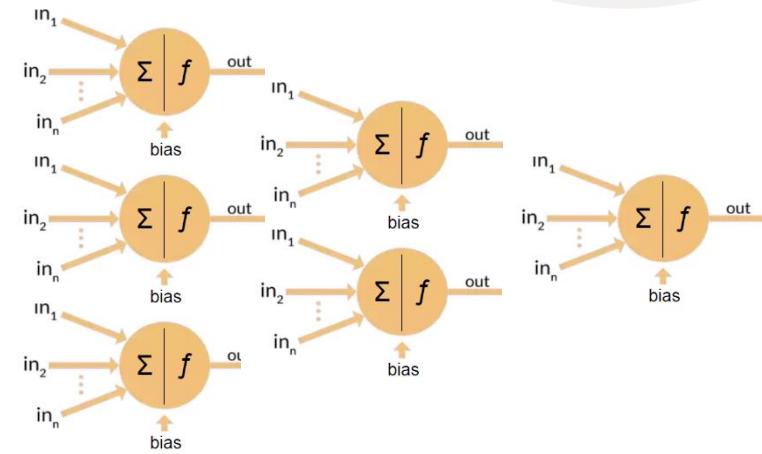


[https://en.wikipedia.org/wiki/Sigmoid\\_function](https://en.wikipedia.org/wiki/Sigmoid_function)

# Neural networks

## Artificial neural networks (ANN)

- are the combination of multiple artificial neurons
- If the artificial neurons are aligned in a layered structure, it is called a **multilayer perceptron (MLP)**
- A single layer of this type of artificial neurons is called a **fully connected / linear / dense layer**
- The first layer is called **input layer** (dummy), the last layer is called **output layer**. All intermediate layers are called **hidden layers**

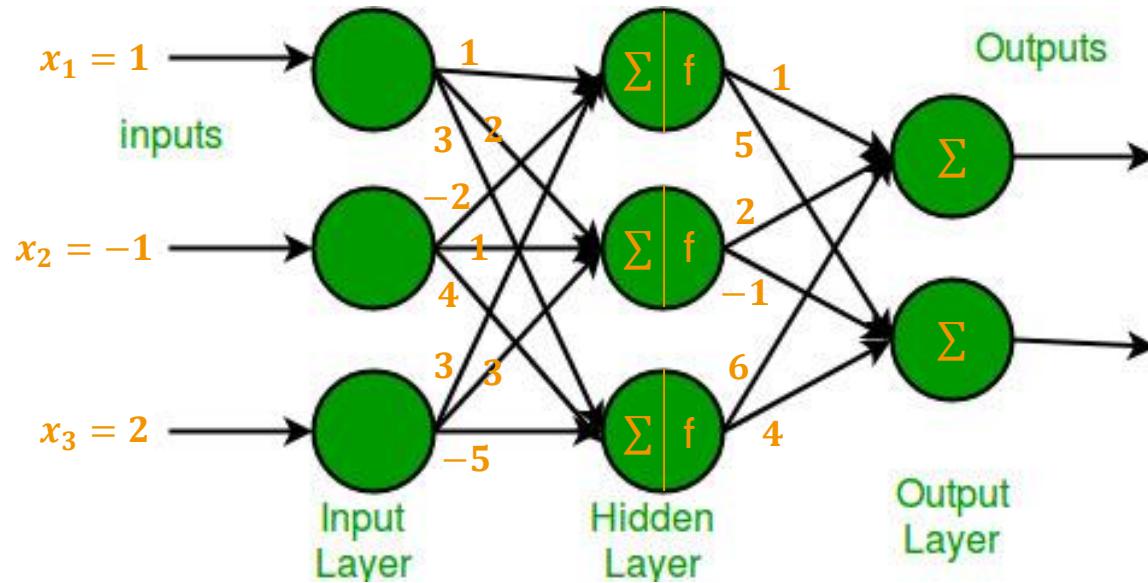


<https://www.geeksforgeeks.org/multi-layer-perceptron-learning-in-tensorflow/>

# Neural networks

Task: Calculate the outputs of the displayed neural network

- all biases = 0
- ReLU activation function  $f(x) = \max(0, x)$  in hidden layer (no other activation functions)
- weights  $w_i$  as shown



<https://www.geeksforgeeks.org/multi-layer-perceptron-learning-in-tensorflow/>

Task: Why is there no activation function in the output layer?

# Neural networks

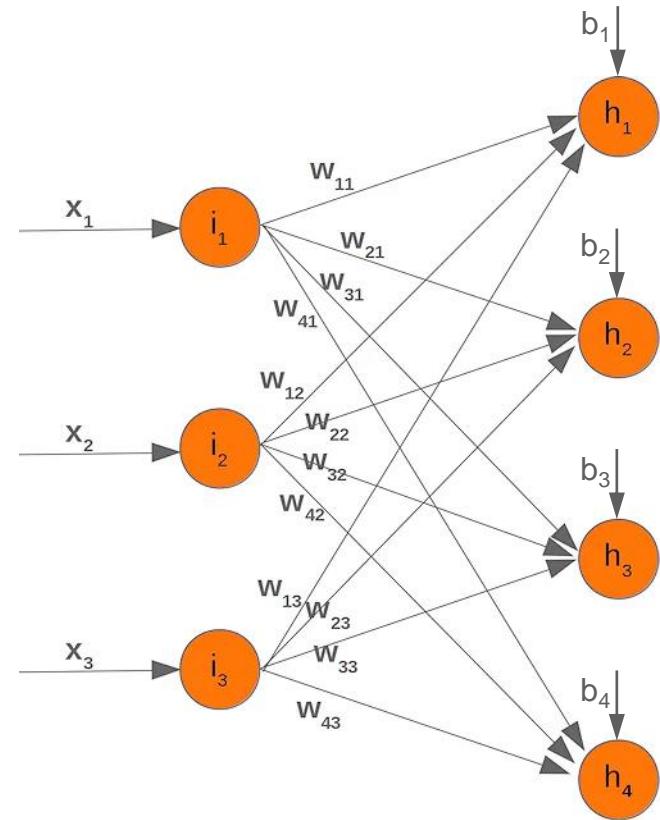
## Matrix representation

- The computation of a dense layer can be simplified using matrix-vector multiplication
$$f(Wx + b)$$
- GPUs are extremely fast in calculating  $Wx + b$

## Example: Matrix-vector representation

- The calculation  $f(\sum_i w_i x_i + b)$  of every neuron within a dense layer can be represented as (for the example on the right)

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = f \left( \begin{bmatrix} w_{11} & w_{21} & w_{31} & w_{41} \\ w_{12} & w_{22} & w_{32} & w_{42} \\ w_{13} & w_{23} & w_{33} & w_{43} \\ w_{14} & w_{24} & w_{34} & w_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \right)$$



<https://python-course.eu/machine-learning/neural-networks-structure-weights-and-matrices.php>

# Neural networks

## Activation function

- The nonlinear activation function  $f$  is an central part of any MLP and must not be omitted
- Without nonlinear activation function, any number of dense layers acts like one single dense layer, e.g. it is for a MLP with two dense layers

$$\mathbf{x}_1 = W_1 \mathbf{x} + \mathbf{b}_1$$

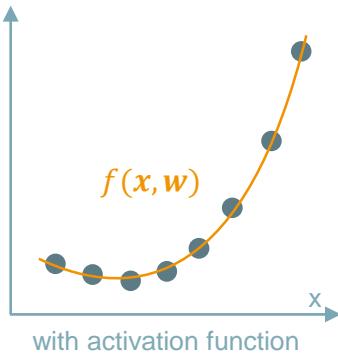
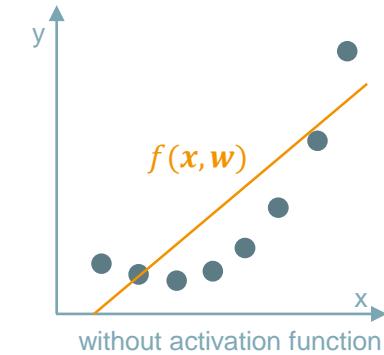
$$\hat{\mathbf{y}} = W_2 \mathbf{x}_1 + \mathbf{b}_2$$

$$= W_2(W_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

$$= (W_2 W_1) \mathbf{x} + (\mathbf{b}_1 + \mathbf{b}_2)$$

$$= W \mathbf{x} + \mathbf{b}$$

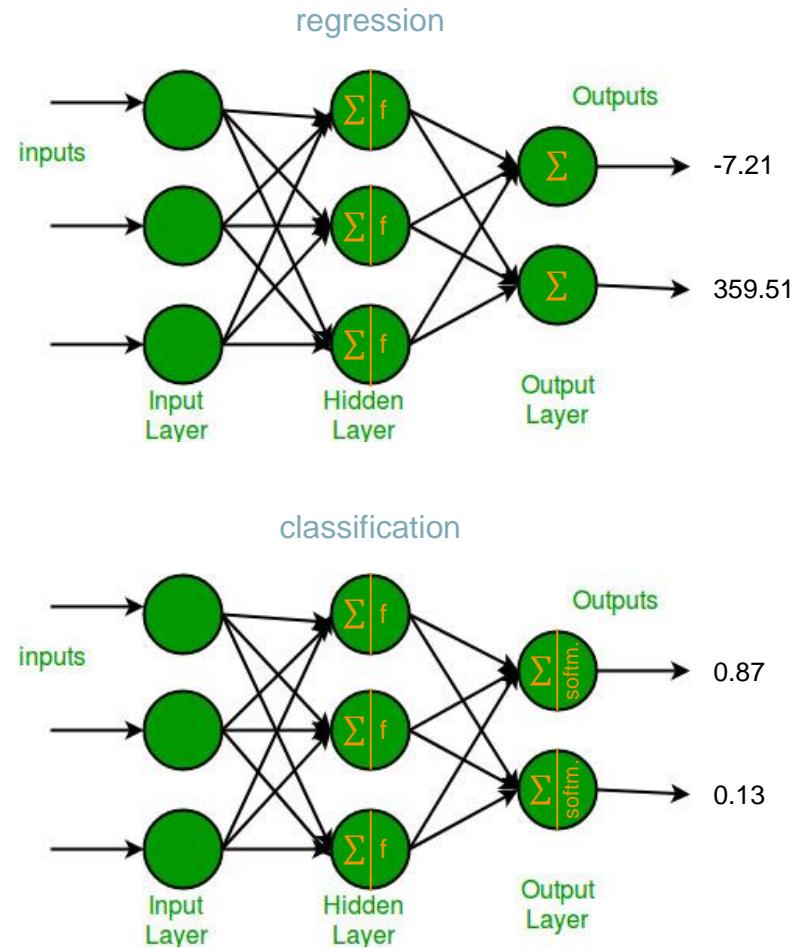
- Without a nonlinear activation function the neural network is only a linear function approximator
- There exist many different activation functions, the most common ones are sigmoid, tanh and ReLU



# Neural networks

## Regression vs. classification

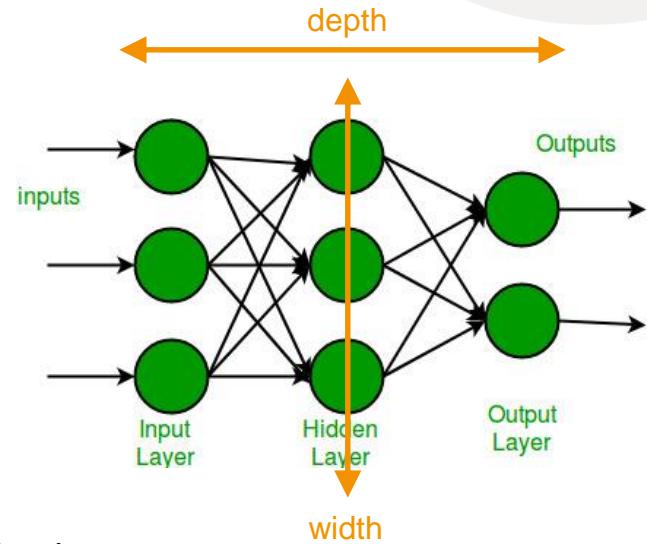
- A MLP for regression tasks does not have an activation function in its output layer. Every output represents one dimension of the regression task.  
→ e.g. Q-function
- A MLP for classification tasks has a softmax activation function in its output layer, limiting every output to the  $[0, 1]$  range and enforcing  $\sum \text{outputs} = 1$ . This allows the outputs to be interpreted as probabilities.  
→ e.g. policy for discrete actions



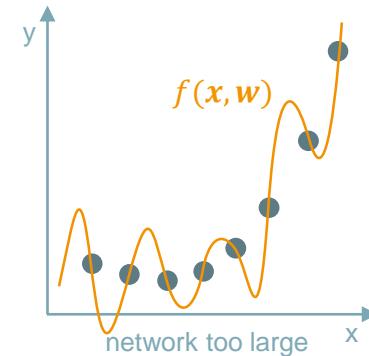
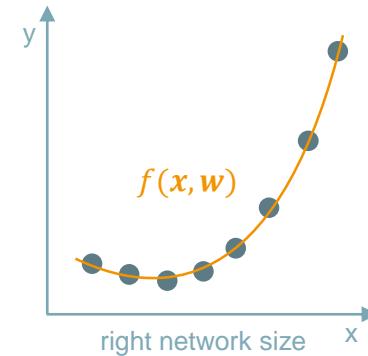
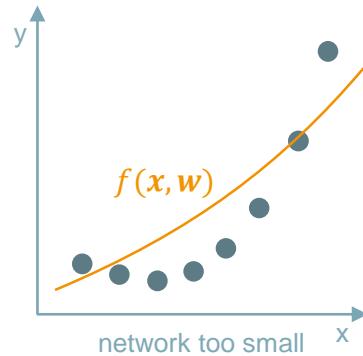
# Neural networks

## Interpretation

- The **depth** of an ANN (number of layers) represents the number of subsequent calculations
- The **width** of an ANN (number of neurons per layer) represents the number of parallel calculations per layer
- The deeper/wider (larger) a network is, the better it can approximate highly nonlinear data (more weights)
- BUT: If a network is too large, overfitting can occur (described later)



<https://www.geeksforgeeks.org/multi-layer-perceptron-learning-in-tensorflow/>



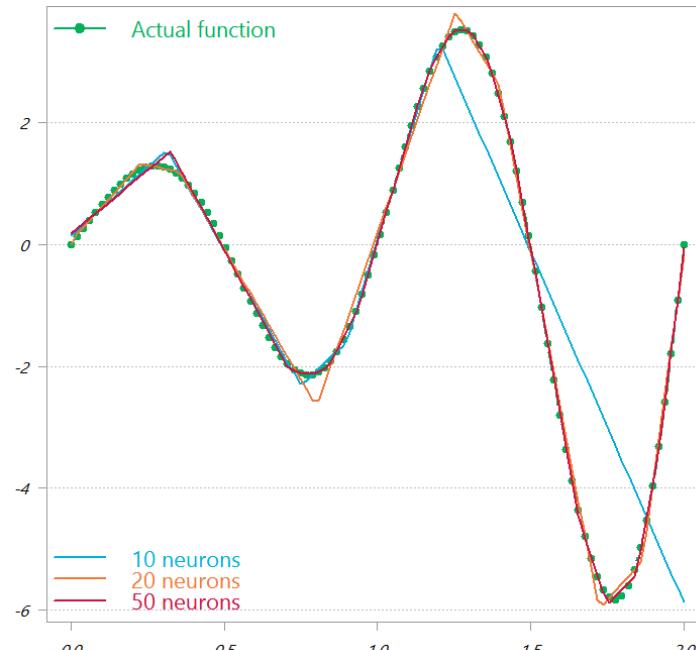
# Neural networks

Task: Draw the structure of a neural network with the following specifications

- Two input dimensions
- One output dimension
- Regression task
- First hidden layer with three neurons
- Second hidden layer with four neurons

## Universal approximation theorem

- Any function can be approximated with arbitrary precision (though not exactly) with a single hidden layer with finite width
- In theory sufficient to use only one (very wide) layer
- In practice tradeoff between width and depth



<https://eranraviv.com/wp-content/uploads/2021/03/Universal-approximation-theorem.png>

# Neural networks

## Different layer types

- This lecture only deals with very simple neural networks (MLPs)
- There are many more layers which can be used to build more complex / highly specialized neural networks  
(e.g. convolution layers for image tasks)  
→ deep learning lecture

## TORCH.NN

These are the basic building blocks for graphs:

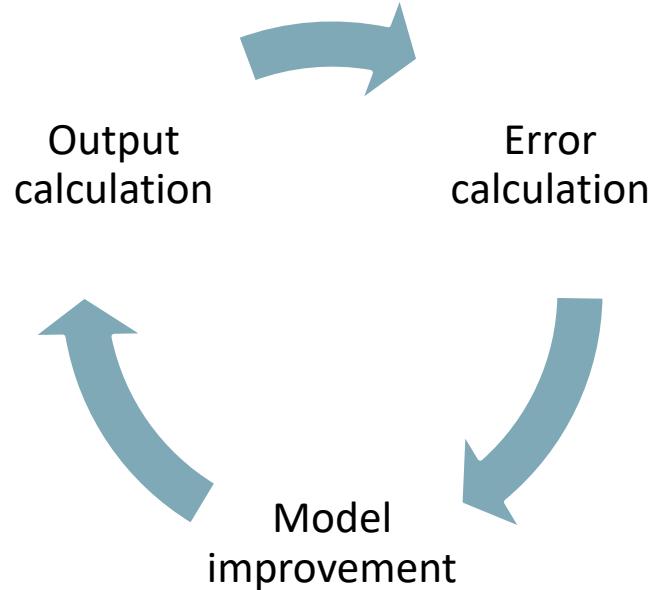
`torch.nn`

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization

<https://pytorch.org/docs/stable/nn.html>

## Neural network training

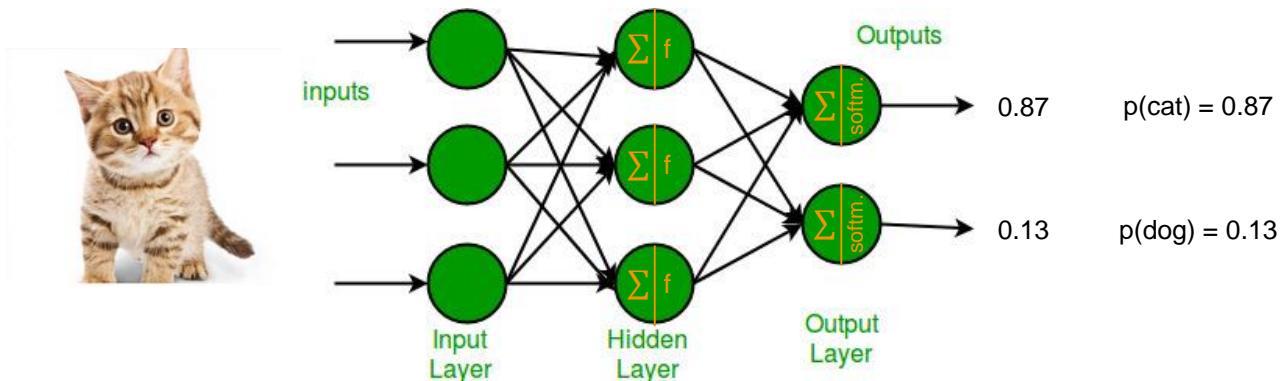
- Training a neural network is a tripartite process which is repeated multiple times
  - Output calculation
  - Error calculation
  - Model improvement



# Neural networks

## Output calculation

- Calculate the output of a neural network for a given input
- Was already done manually before

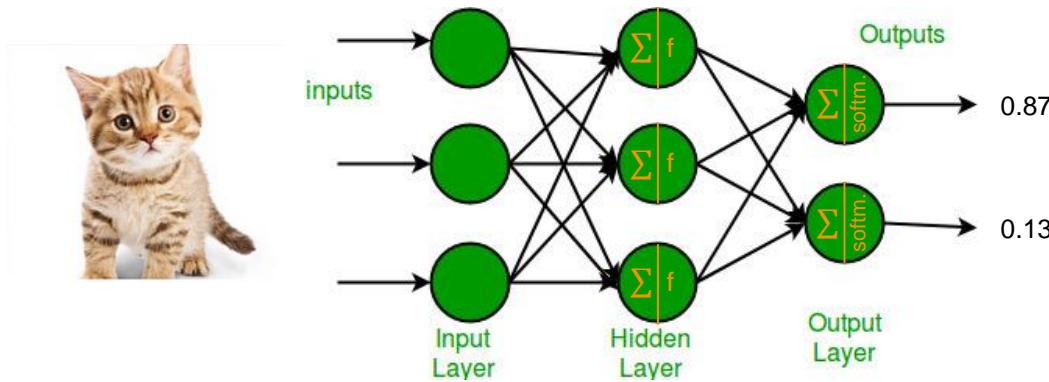


# Neural networks

## Error calculation

- Calculate the error (**loss**)  $L$  between output  $\hat{y}$  of the neural network (**prediction**) and the desired output  $y$  (**label**)
- A proper loss function is 0 for  $\hat{y} = y$  and  $> 0$ , the larger the difference between  $\hat{y}$  and  $y$  is

$$L = (0.87 - 1)^2 + (0.13 - 0)^2 = 0.0338$$



class	$\hat{y}$	$y$
cat	0.87	1.00
dog	0.13	0.00

# Neural networks

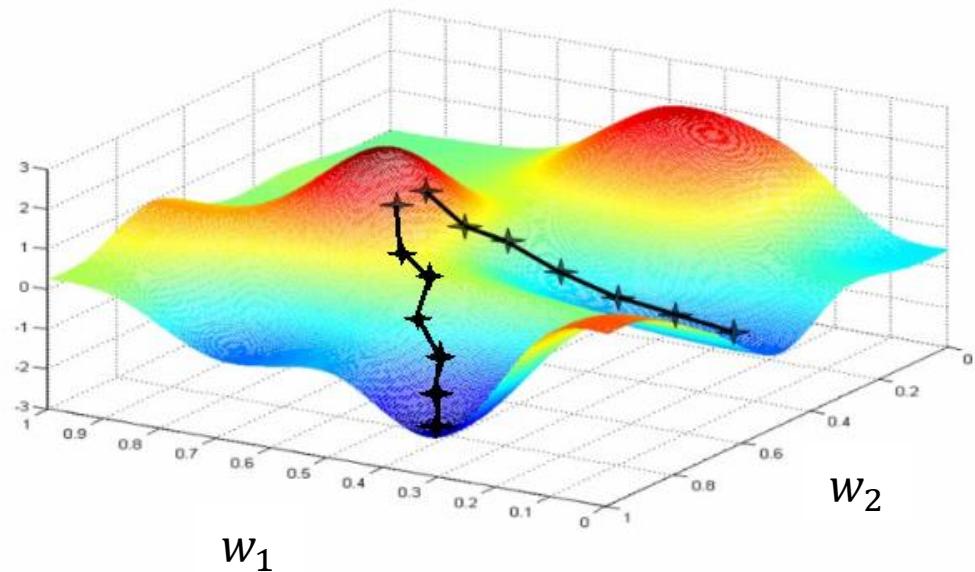
## Model improvement

- Modify the network weights  $\mathbf{w}$  (= weights  $W_k$  and biases  $b_k$  of all dense layers) to reduce the error → gradient descent

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \eta \cdot \nabla_{\mathbf{w}} L$$

- Problem: How to calculate the gradient  
→ backpropagation

$$L(w_1, w_2)$$



<https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/>

# Neural networks

## Backpropagation

- is the name of the method to calculate the gradient  $\nabla_w L$  for neural networks
- is a fancy name for the chain rule of differentiation you know from high school

Example: Derivative of

$$f(x) = \log(\sin(x^2)) = a(b(c(x)))$$

with

$$a(b) = \log(b), \quad b(c) = \sin(c), \quad c(x) = x^2$$

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{\partial a}{\partial b} \frac{\partial b}{\partial c} \frac{\partial c}{\partial x} \\ &= \frac{1}{b} \cos(c) 2x \\ &= \frac{1}{\sin(x^2)} \cos(x^2) 2x \end{aligned}$$

# Neural networks

Example: Derivative of

$$L = a(\mathbf{b}(c(\mathbf{d}(W_1))))$$

with

$$a(\mathbf{b}) = \sum_i ((f(\mathbf{b})_i - \mathbf{y}_i)^T \cdot (f(\mathbf{b})_i - \mathbf{y}_i))$$

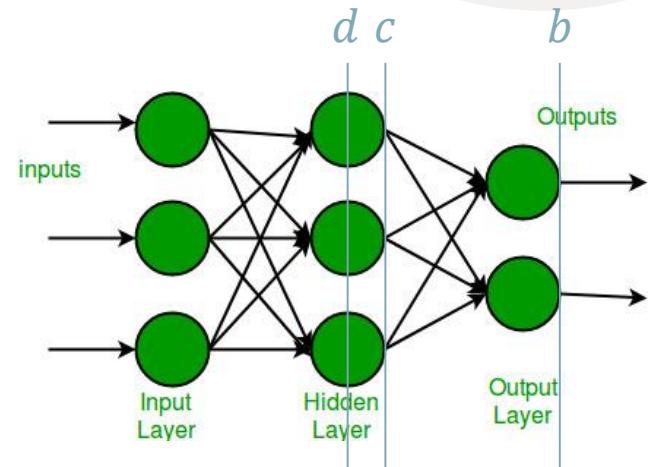
$$\mathbf{b}(\mathbf{c}) = W_2 \mathbf{c} + \mathbf{b}_2$$

$$c(\mathbf{d}) = \text{ReLU}(\mathbf{d})$$

$$\mathbf{d}(W_1) = W_1 \mathbf{x} + \mathbf{b}_1$$

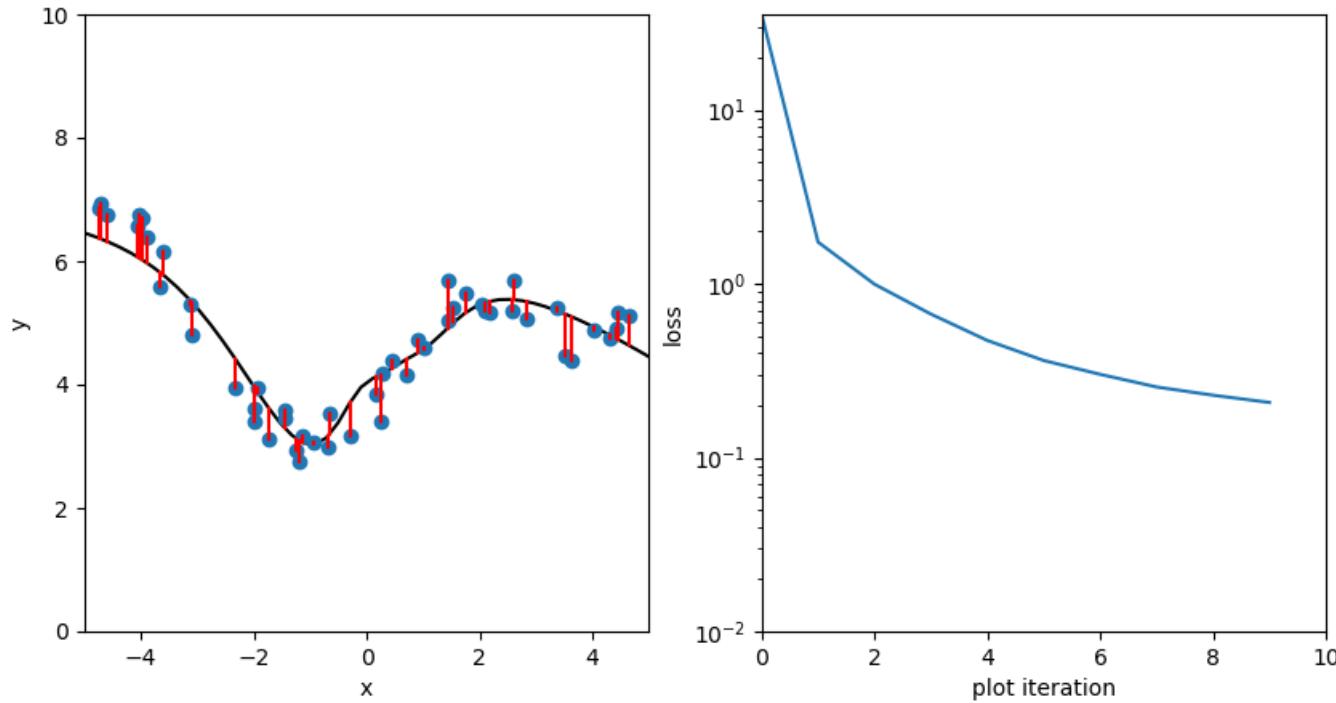
$$\frac{\partial L}{\partial W_1} = \frac{\partial a}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \mathbf{c}} \frac{\partial \mathbf{c}}{\partial \mathbf{d}} \frac{\partial \mathbf{d}}{\partial W_1}$$

- Calculating  $\frac{\partial L}{\partial W_1}$  is not easy, but doable ( $\rightarrow$  deep learning lecture)
- Also have to calculate  $\frac{\partial L}{\partial W_2}$ ,  $\frac{\partial L}{\partial \mathbf{b}_1}$  and  $\frac{\partial L}{\partial \mathbf{b}_2}$  to obtain the gradient  $\nabla_{\mathbf{w}} L$



# Neural networks

Example: Code



# Neural networks

PyTorch / Tensorflow / MXNet / ...

- are software frameworks specifically designed to calculate the gradient  $\nabla_w L$  and gradient descent
- no need to worry, all the complex gradient calculations will be done by software

Example: Pytorch (simplified)

```
for i in range (n_iter):

    # making predictions with forward pass
    Y_pred = forward(X)

    # calculating the loss between original and predicted data points
    loss = criterion(Y_pred, Y)

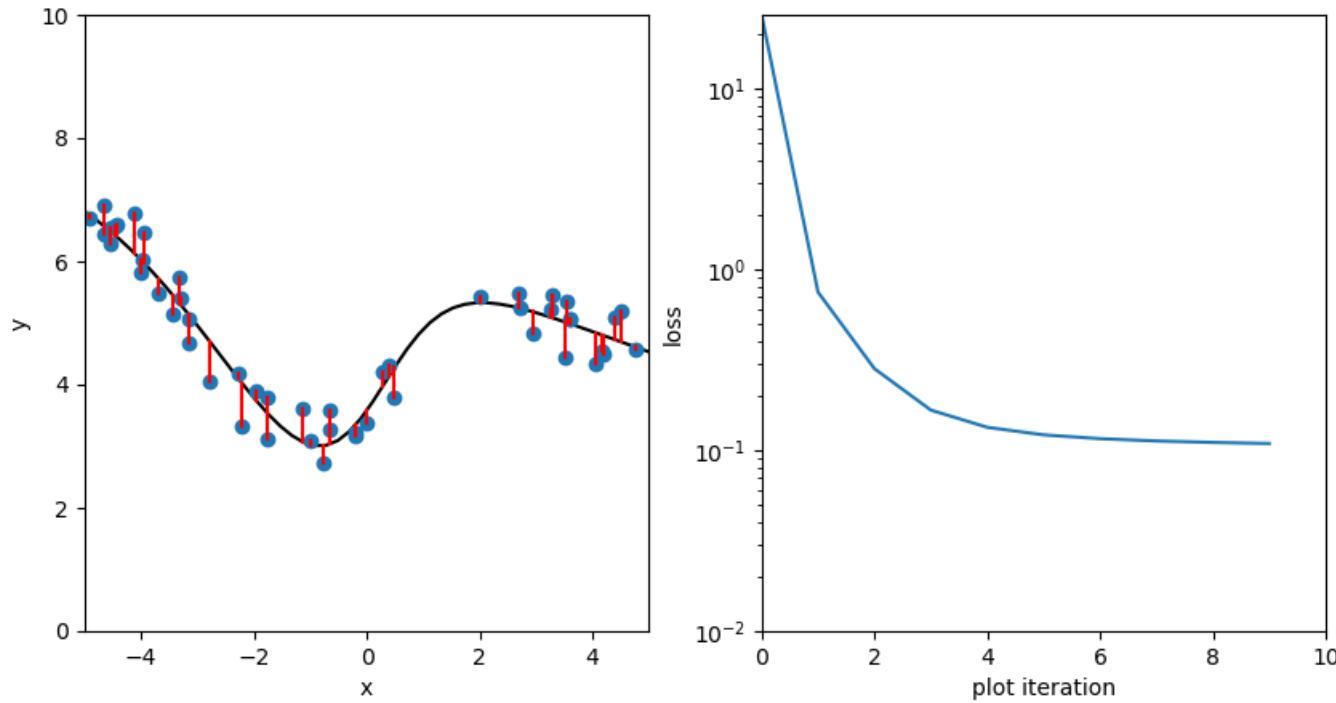
    # backward pass for computing the gradients of the loss w.r.t to learnable parameters
    loss.backward()

    # update the parameters based on the gradient
    w.data = w.data - step_size * w.grad.data
```

<https://machinelearningmastery.com/implementing-gradient-descent-in-pytorch/>

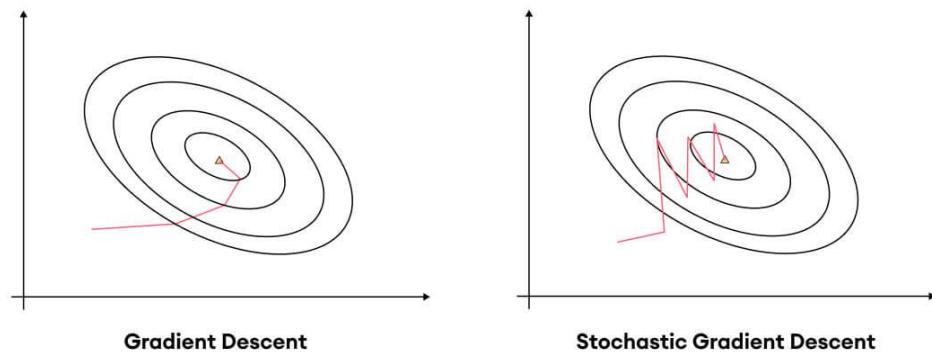
# Neural networks

Example: Code



## Stochastic gradient descent (SGD)

- Problem: If a dataset contains too many samples, calculating the gradient  $\nabla_w L$  based on all samples at once will require a lot of time
- Solution: Calculate the gradient only for a subset of all samples (stochastic gradient descent)
- The samples of the subset should change for each gradient descent step
- The number of samples within the subset is called **batch size**
- An **epoch** is defined as  $\frac{\text{dataset size}}{\text{batch size}}$



effect of SGD:  
noisy approximation of  
the true gradient (which  
is based on all samples)

# Neural networks

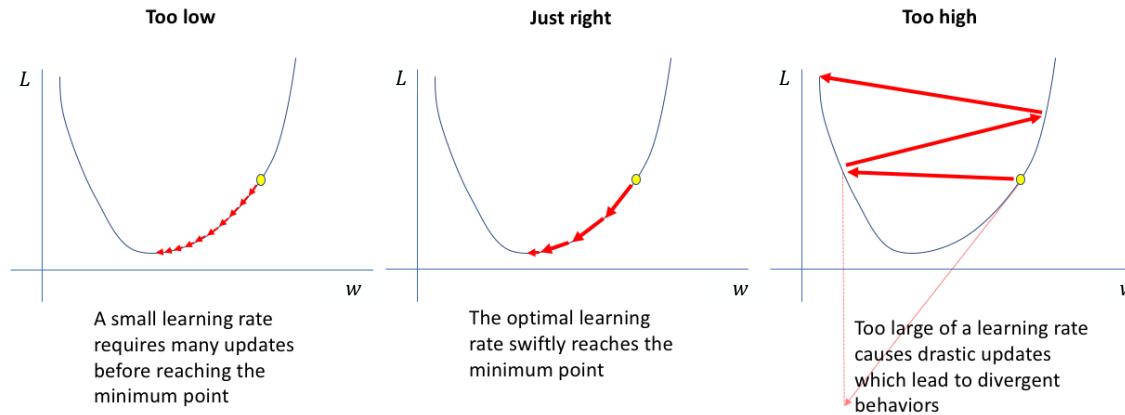
Problem with all gradient descent approaches: Finding the correct learning rate  $\eta$  in

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \eta \cdot \nabla_{\mathbf{w}} L$$

- If the learning rate is too small, training takes long
- If the learning rate is too large, the updates diverge

Solution: Adapt learning rate during training (next slides) through

- Learning rate scheduler
- Optimizer

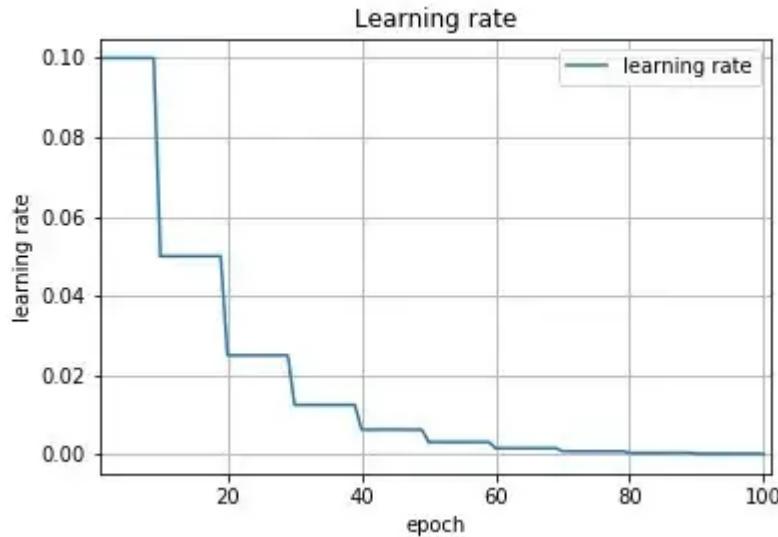


<https://www.jeremyjordan.me/nn-learning-rate/>

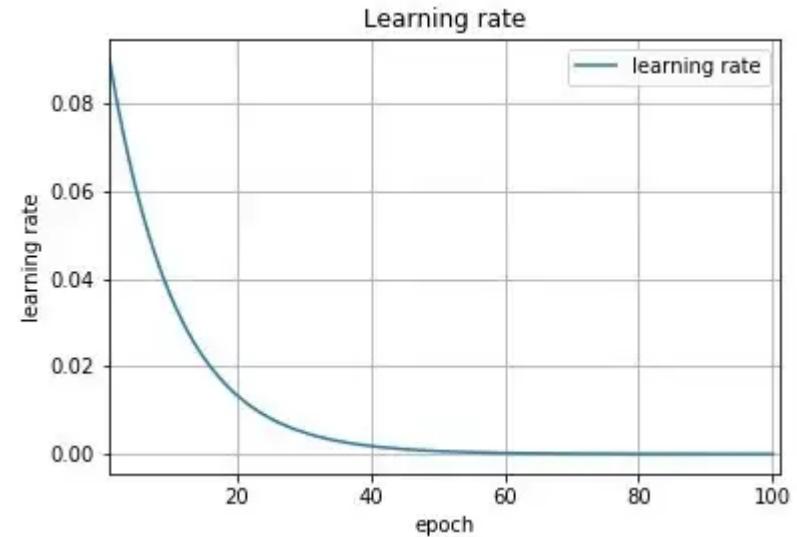
## Learning rate scheduler

- decrease the learning rate during training according to a predefined schedule

Step decay scheduler



Exponential decay scheduler



<https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1>

# Neural networks

## Optimizer

- Update the weights not only based on the gradient but also
  - based on the last update steps
  - dynamically along every dimensions
- There exist many different optimizer which often accelerate training but may fail for specific ANNs

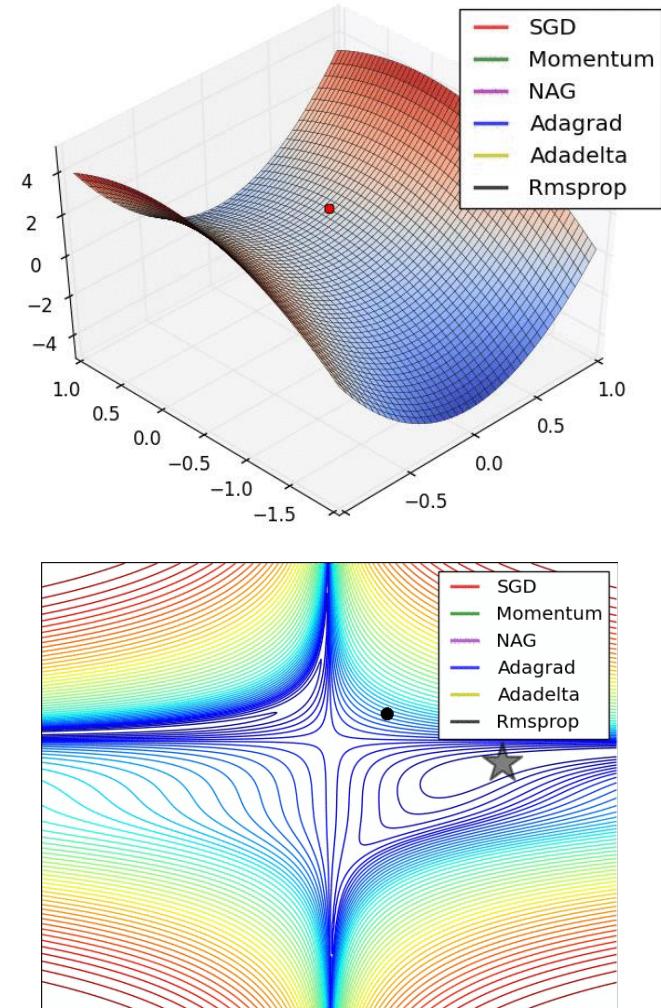
## Example

- Adagrad

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \frac{\eta}{\sqrt{\epsilon + \text{diag}(G_k)}} \cdot \nabla_{\mathbf{w}} L$$

compared to SGD

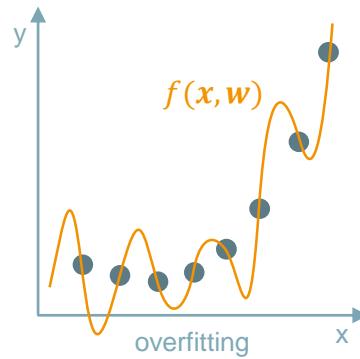
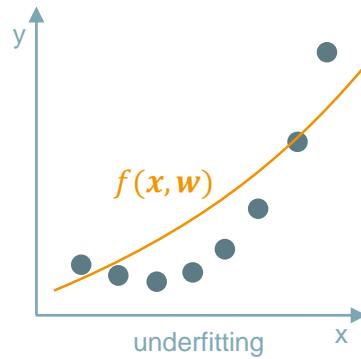
$$\mathbf{w}_k = \mathbf{w}_{k-1} - \eta \cdot \nabla_{\mathbf{w}} L$$



<https://ruder.io/optimizing-gradient-descent/>

## Overfitting/underfitting

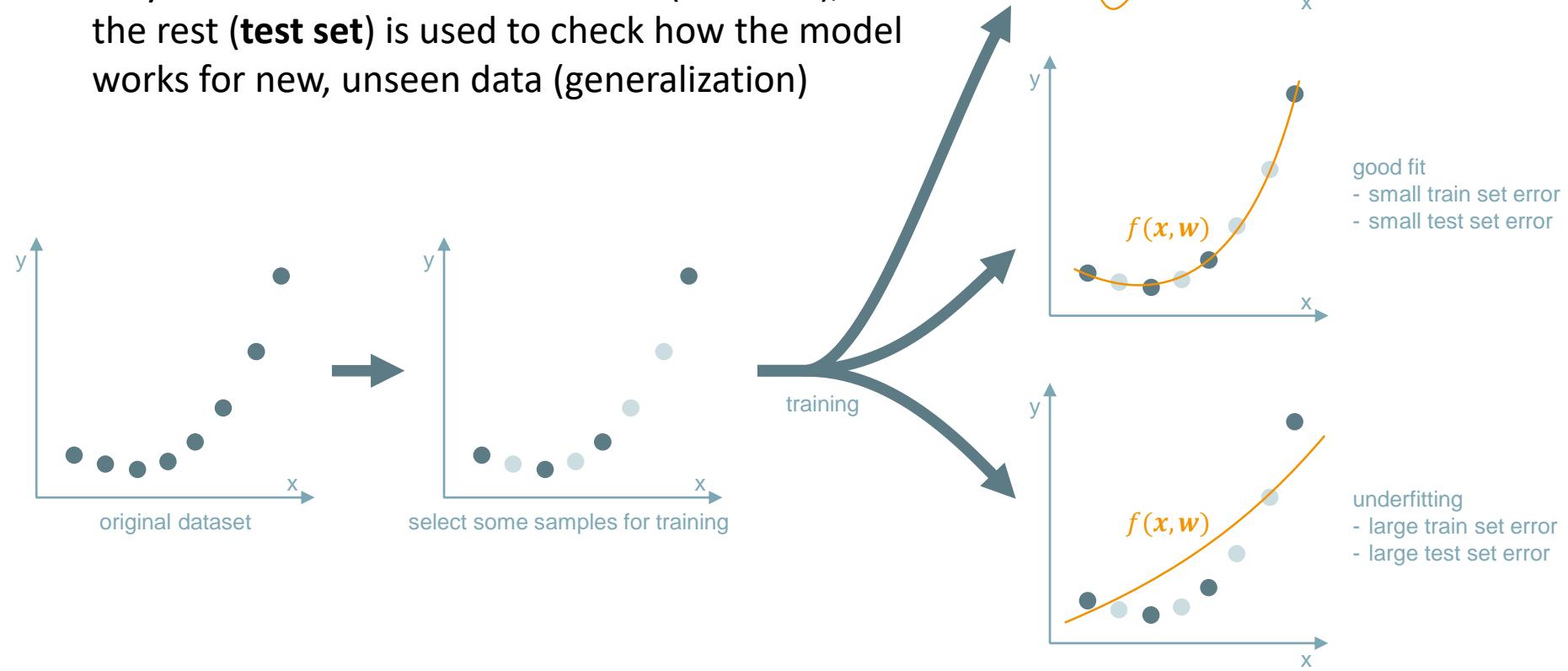
- If a neural network is too deep/wide, overfitting occurs
- If a neural network is not deep/wide enough, underfitting occurs
- Both over- and underfitting are undesired properties and can be countered e.g. through proper dimensioning of the ANN  
(more techniques in the deep learning lecture)



# Neural networks

## Detection of over-/underfitting

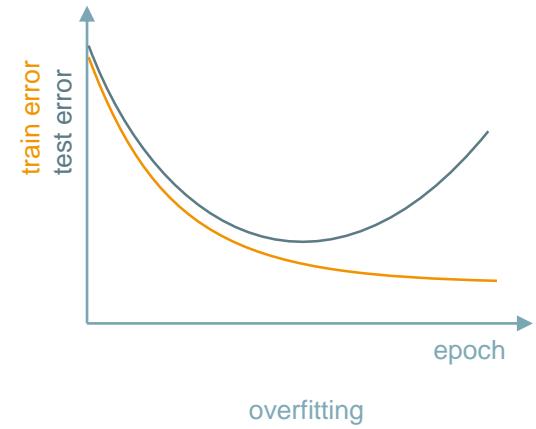
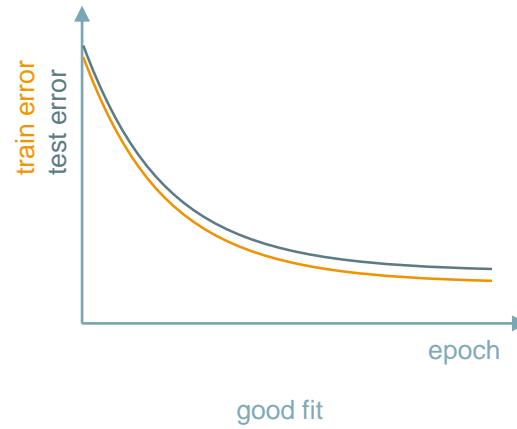
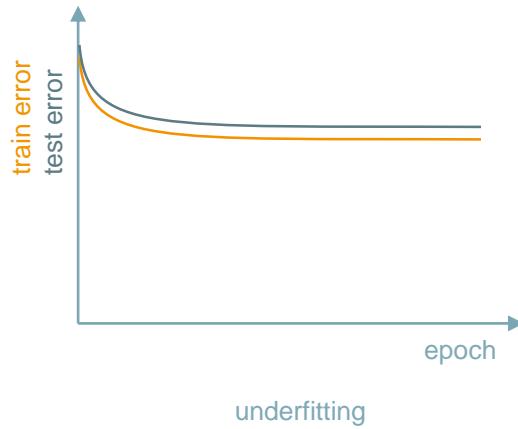
- Over- and underfitting can be detected using a **train/test split** where the neural network is only trained on a subset of all data (**train set**), the rest (**test set**) is used to check how the model works for new, unseen data (generalization)



# Neural networks

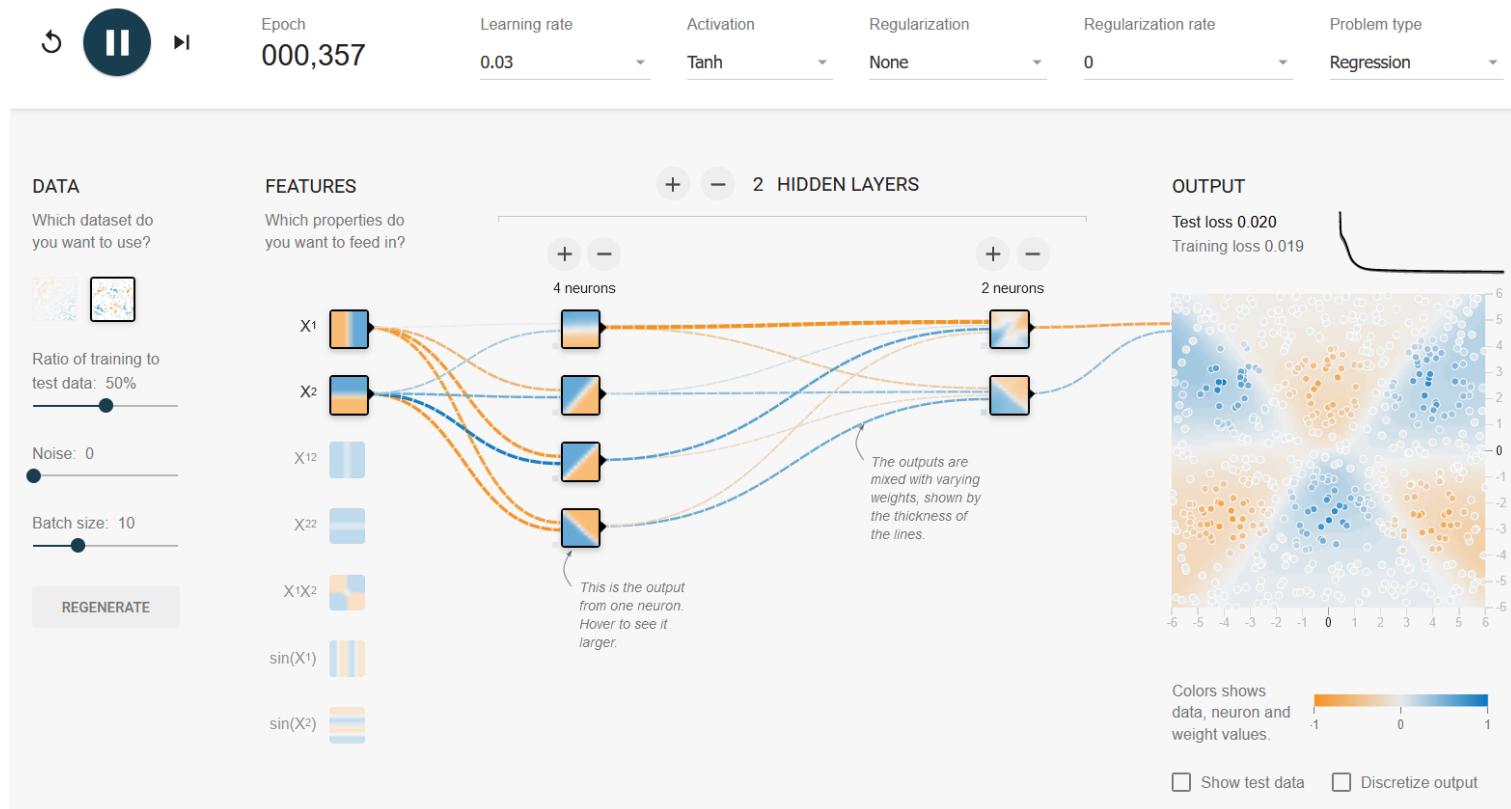
## Learning curve

- A learning curve visualizes the train/test set error over gradient descent steps / epochs
- It is used to detect over-/underfitting through "manual inspection"



# Neural networks

## Example: Tensorflow playground



<https://playground.tensorflow.org>

# Kahoot!

# Neural networks

## Homework:

Learn the basics of PyTorch: <https://pytorch.org/tutorials/beginner/basics/intro.html>

- at least read all linked guides on the page
- ideally install PyTorch locally on your PC and run the code



[https://cdn2.psychologytoday.com/assets/styles/manual\\_crop\\_1\\_91\\_1\\_1520x800/public/field\\_blog\\_entry\\_teaser\\_image/2018-11/depositphotos\\_51277329\\_s-2015.jpg?itok=guo89dbR](https://cdn2.psychologytoday.com/assets/styles/manual_crop_1_91_1_1520x800/public/field_blog_entry_teaser_image/2018-11/depositphotos_51277329_s-2015.jpg?itok=guo89dbR)

# Deep RL

**Deep RL = RL + neural networks as function approximators**

The methods described in this chapter  
find an optimal policy for an  
**unknown, continuous MDP**

- Unknown: states/actions/transitions are not known a-priori  
→ agent must rely on trial-and-error (run episodes and see how well it goes)
- Continuous: Too many states/actions to list them in tabular form

Four methods are presented

- Deep Sarsa, Deep Q-learning
- DQN
- Actor-critic

Different types of unknown, discrete/continuous MDPs  
(with corresponding algorithm and example)

	discrete action space	continuous action space
discrete state space	SARSA / Q-learning / MC (robot maze)	actor-critic
continuous state space	Deep Sarsa, Deep Q-Learning, DQN (Breakout, Space Invaders)	actor-critic (autonomous driving)

## Motivation/explanation 1 for Deep Sarsa, Deep Q-learning

- Recap: SARSA update rule

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma Q(s', a') - Q(s, a))$$

- Recap: Q-learning update rule

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( R + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

- What happens after we calculated the optimal Q-function?

→ There is a unique optimal Q-function, so it will not change

→ SARSA / Q-learning update rules will be of the form

$$Q(s, a) \leftarrow Q(s, a)$$

→ the other terms in the SARSA/Q-learning update rule must be zero,  
e.g. it is for SARSA

$$R + \gamma Q(s', a') - Q(s, a) = 0$$

## Motivation/explanation 2 for Deep Sarsa, Deep Q-learning

- Approximate the true Q-function  $Q(s, a)$  with a neural network  $\hat{Q}(s, a, \mathbf{w})$
- Minimize loss  $L$  measuring the difference between true and approximated Q-function through gradient descent
  - Define loss function

$$L = (Q(s, a) - \hat{Q}(s, a, \mathbf{w}))^2$$

- Calculate gradient

$$\nabla_{\mathbf{w}} L = -2(Q(s, a) - \hat{Q}(s, a, \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w})$$

- Use approximation of the true Q-function, e.g. for SARSA

$$\nabla_{\mathbf{w}} L = -2(R + \gamma \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w})$$

- Update weights  $\mathbf{w}$  through (stochastic) gradient descent

## Deep Sarsa

- Train a neural network to approximate the true Q-function  $Q(s, a)$  with  $\hat{Q}(s, a, \mathbf{w})$
- Loss  $L$  to be minimized is of the form

$$L = (R + \gamma \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w}))^2$$

- If the loss becomes zero for all states/actions, the optimal Q-function is obtained (Q-function won't be updated anymore)

$$\hat{Q}(s, a, \mathbf{w}) \leftarrow \hat{Q}(s, a, \mathbf{w}) + \alpha \cancel{(R + \gamma \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w}))}$$

- Resulting gradient becomes

$$\nabla_{\mathbf{w}} L = 2(R + \gamma \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w})$$

## Deep Q-learning

- Train a neural network to approximate the true Q-function  $Q(s, a)$  with  $\hat{Q}(s, a, \mathbf{w})$
- Loss  $L$  to be minimized is of the form

$$L = \left( R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w}) \right)^2$$

- If the loss becomes zero for all states/actions, the optimal Q-function is obtained (Q-function won't be updated anymore)

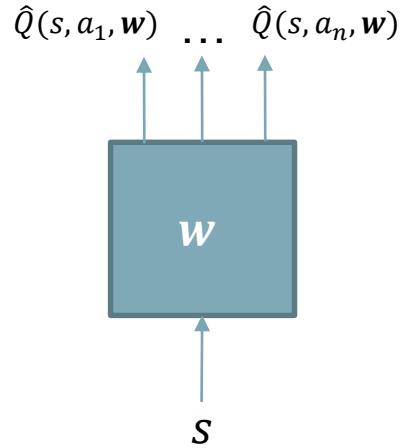
$$\hat{Q}(s, a, \mathbf{w}) \leftarrow \hat{Q}(s, a, \mathbf{w}) + \alpha \left( R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w}) \right)$$

- Resulting gradient becomes

$$\nabla_{\mathbf{w}} L = 2 \left( R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w})$$

## How to train Deep Sarsa / Deep Q-learning

1. Run certain number of episodes with neural network (type shown below)
2. Store data of the form  $(s, a, r, s', a')$  for every time step in a buffer
3. Train neural network to minimize the loss  $L$ , then clear the buffer and go to 1.  
(Neural network has one output per action, representing its Q-value)



Note that finding a  $\epsilon$ -greedy/optimal policy based on the obtained values  $\hat{Q}(s, a_1, \mathbf{w}) \dots \hat{Q}(s, a_n, \mathbf{w})$  is easy, just pick the action with the largest value

## DQN (Deep Q Network)

- Deep Q-learning with improvements:
  - experience replay
  - target network
- first "proper" deep RL approach  
(deep SARSA and deep Q-learning are just for didactic purpose)

Replay	○	○	✗	✗
Target	○	✗	○	✗
Breakout	<b>316.8</b>	240.7	10.2	3.2
River Raid	<b>7446.6</b>	4102.8	2867.7	1453.0
Seaquest	<b>2894.4</b>	822.6	1003.0	275.8
Space Invaders	<b>1088.9</b>	826.3	373.2	302.0

<https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b>

## Experience replay

Idea: Use all data obtained so far for training

### Training steps

1. Run certain number of episodes with neural network
2. Store data of the form  $(s, a, r, s', a')$  for every time step in a buffer
3. Train neural network to minimize the loss  $L$ , then ~~clear the buffer~~ and go to 1.

### Implementation

- buffer often has finite size, delete oldest data if buffer is full
- training often happens only on a (random) subset of all buffer data (faster if buffer is large)

Effect of experience replay:

- Is possible, because Q-learning is off-policy
- Each observed transition  $(s, a, r, s', a')$  can be used multiple times for training  
→ better data efficiency
- History contains data from many different policies  
→ lots of different samples seen



Task: What does off-policy mean?

## Target network

Idea: Fix the **orange term (target network)** for a certain number of training steps to avoid unstable behaviour

$$\hat{Q}(s, a, \mathbf{w}) \leftarrow \hat{Q}(s, a, \mathbf{w}) + \alpha \left( R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w}) \right)$$

(1) At first everything will look normal. We just chase the target.	(2) But the target will move as our Q-function improves.	(1) Suppose we freeze the target for a few steps.	(2) That way the optimizer can make stable progress towards it.
<p>(3) Then, things go bad.</p> <p>(4) And the moving targets could create divergence.</p>		<p>(3) We eventually update the target, and repeat.</p>	<p>(4) This allows the algorithm to make stable progress.</p>

<https://livebook.manning.com/concept/reinforcement-learning/target-network>

Effect of target networks:

- By using target networks, the goal  $R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w})$  that should be learned by  $\hat{Q}(s, a, \mathbf{w})$  in order to minimize

$$L = \left( R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}') - \hat{Q}(s, a, \mathbf{w}) \right)^2$$

changes only slowly → more stable learning

Training steps with experience replay and target network

1. Initialization: Create two identical neural networks  $\hat{Q}(s, a, \mathbf{w})$  and  $\hat{Q}(s, a, \mathbf{w}')$
2. Repeat for a certain number of times
  - a) Run certain number of episodes (policy based on neural network  $\hat{Q}(s, a, \mathbf{w})$ )
  - b) Store data of the form  $(s, a, r, s', a')$  for every time step in a buffer
  - c) Train neural network  $\hat{Q}(s, a, \mathbf{w})$  to minimize the loss  $L$

$$L = \left( R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}') - \hat{Q}(s, a, \mathbf{w}) \right)^2$$

This corresponds to the update rule

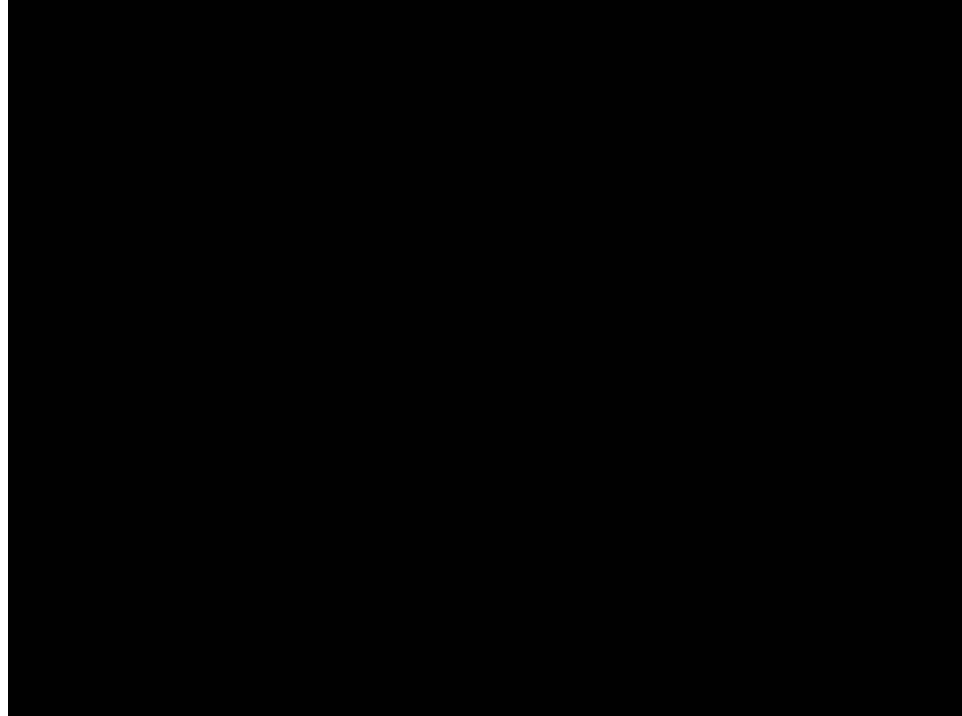
$$\hat{Q}(s, a, \mathbf{w}) \leftarrow \hat{Q}(s, a, \mathbf{w}) + \alpha \left( R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}') - \hat{Q}(s, a, \mathbf{w}) \right)$$

3. Copy  $\hat{Q}(s, a, \mathbf{w})$  to obtain  $\hat{Q}(s, a, \mathbf{w}')$  and go to 2.

Task: Why is there no Deep SARSA with experience replay and target networks?

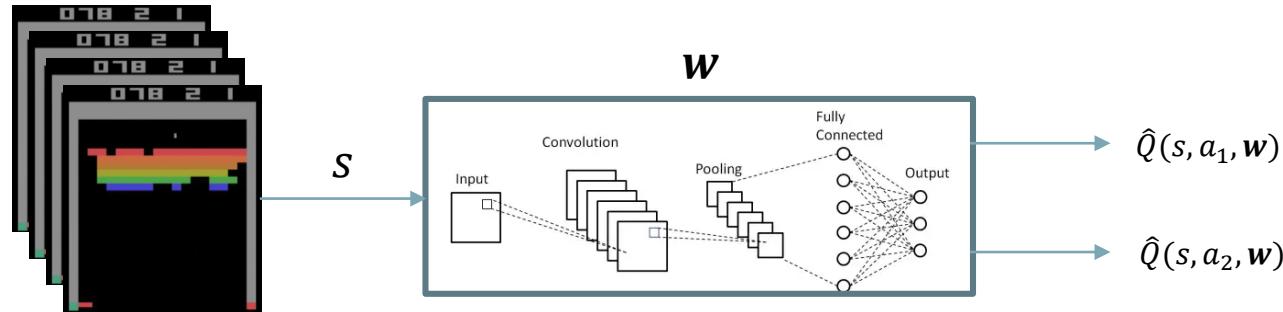
## Example: Playing Atari with Deep RL (2013)

- "inventor" of DQN
- breakthrough as it uses raw images as input
- no domain knowledge  
→ generalizes to other tasks
- 2015: presented in Nature
- paved the road for many modern deep RL techniques
- successor: DDQN



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

- function approximator is a convolutional neural network
- state is encoded by the last four images



Task: What do the two outputs  $\hat{Q}(s, a_1, \mathbf{w})$  and  $\hat{Q}(s, a_2, \mathbf{w})$  represent?

Task: Why are the last four images used to represent the state (and not just a single one)?

## DDQN (Double Deep Q Networks)

- Problem of DQN (similar to Q-Learning): max-operator in

$$\hat{Q}(s, a, \mathbf{w}) \leftarrow \hat{Q}(s, a, \mathbf{w}) + \alpha \left( R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}') - \hat{Q}(s, a, \mathbf{w}) \right)$$

may overestimate the true Q-value if  $\hat{Q}$  is inaccurate

- There are two situations where this has an influence
  - selecting an action during episodes
  - rating an action during training
- Idea of DDQN: Decouple these two steps in to reduce the effect of a wrong estimate (use two different networks)

- Original update rule (**target**)

$$\hat{Q}(s, a, \mathbf{w}) \leftarrow \hat{Q}(s, a, \mathbf{w}) + \alpha \left( R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}') - \hat{Q}(s, a, \mathbf{w}) \right)$$

- DQN has the target

$$R + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}')$$

$$R + \gamma \hat{Q} \left( s', \operatorname{argmax}_{a'} \hat{Q}(s', a', \mathbf{w}'), \mathbf{w}' \right)$$

- DDQN has the target

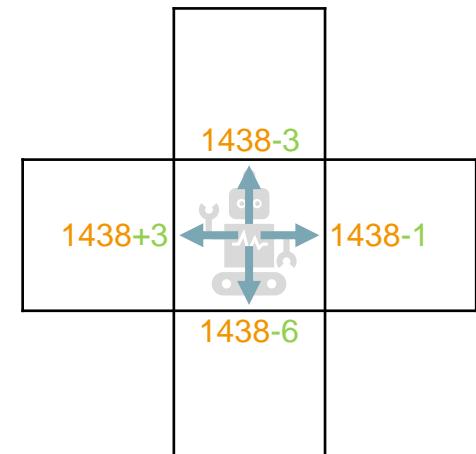
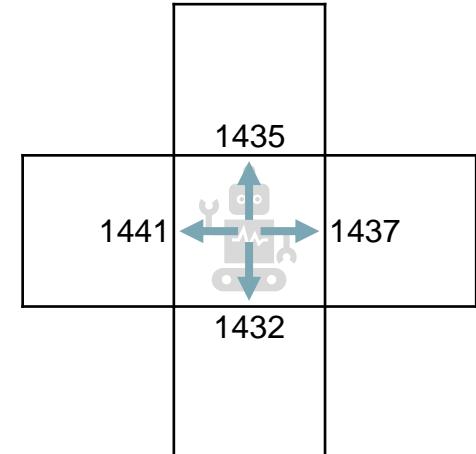
$$R + \gamma \hat{Q} \left( s', \operatorname{argmax}_{a'} \hat{Q}(s', a', \mathbf{w}'), \mathbf{w} \right)$$

## Dueling (D)DQN

- Problem of DQN/DDQN: The absolute Q-values are not that important when choosing an action, it is more their relative value w.r.t. each other
- Idea: Split up the learned Q-function into two different networks:
  - One network can learn the absolute value (value-function)
  - The other network can learn the relative value for different actions (advantage-function)
- Formula used in paper:

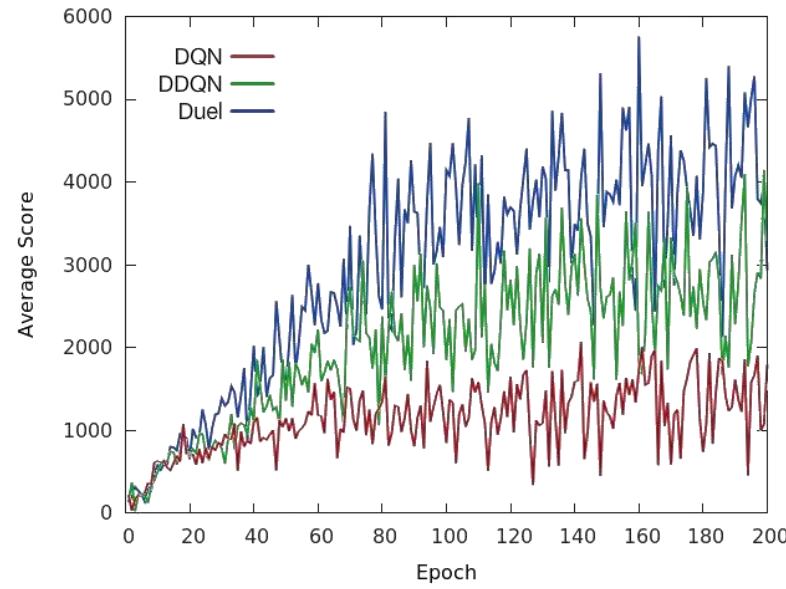
$$\hat{Q}(s, a, \mathbf{w}) = V(s, \mathbf{w}) + A(s, a, \mathbf{w}') \quad \text{or}$$

$$\hat{Q}(s, a, \mathbf{w}) = V(s, \mathbf{w}) + \left( A(s, a, \mathbf{w}') - \max_a A(s, a, \mathbf{w}') \right)$$



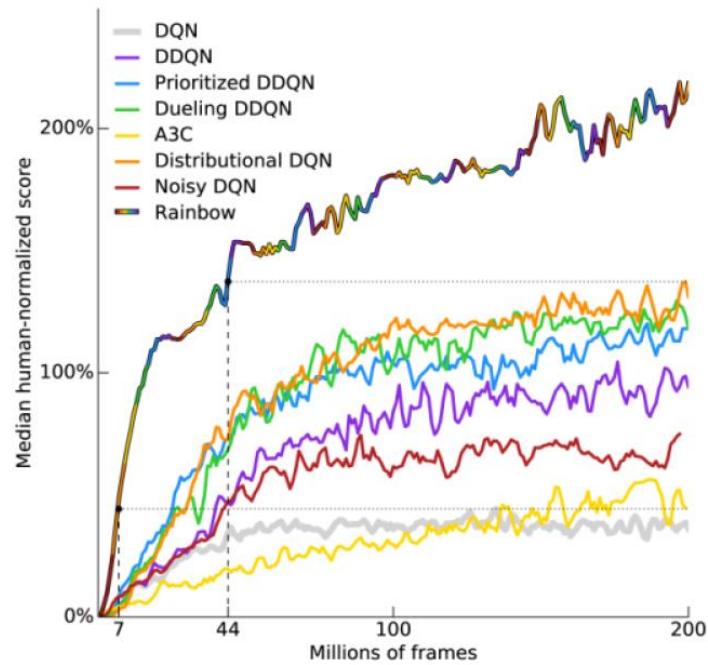
## Example: Space Invaders

- performance of DQN / DDQN / Dueling DQN over trained epoch



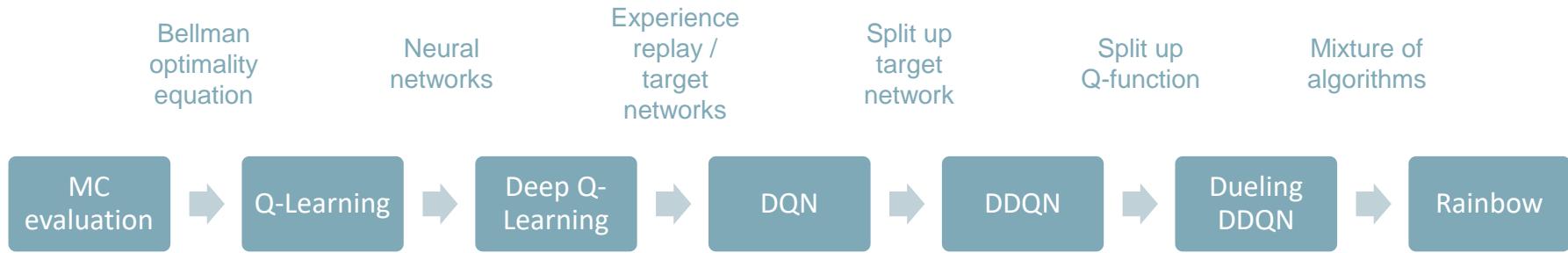
## Rainbow

- Idea: Because different DQN variants excel for certain problems, iterate over all of them during training



<https://www.arxiv-vanity.com/papers/1710.02298/>

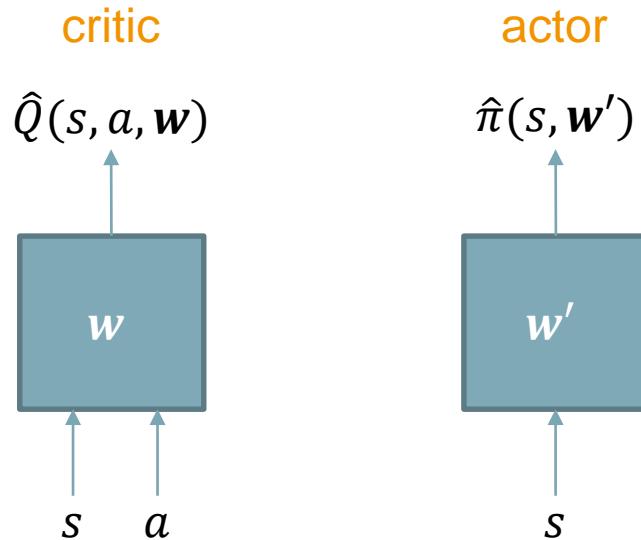
## Evolution of RL algorithms



Task: What are the improvements for each RL algorithm?

## Actor-critic methods

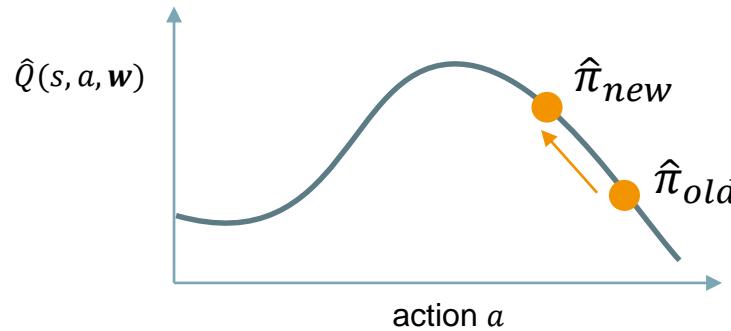
- are used mainly for continuous state / continuous action MDPs  
(  $\leftrightarrow$  DQN: continuous state / discrete action)
- rely on two neural networks
  - critic: approximate optimal (V-)/Q-function based on current state/action
  - actor: approximate optimal policy based on current state



- the critic  $\hat{Q}(s, a, \mathbf{w})$  is trained to learn the optimal (V-)/Q-function, e.g. by minimizing the loss  $L$

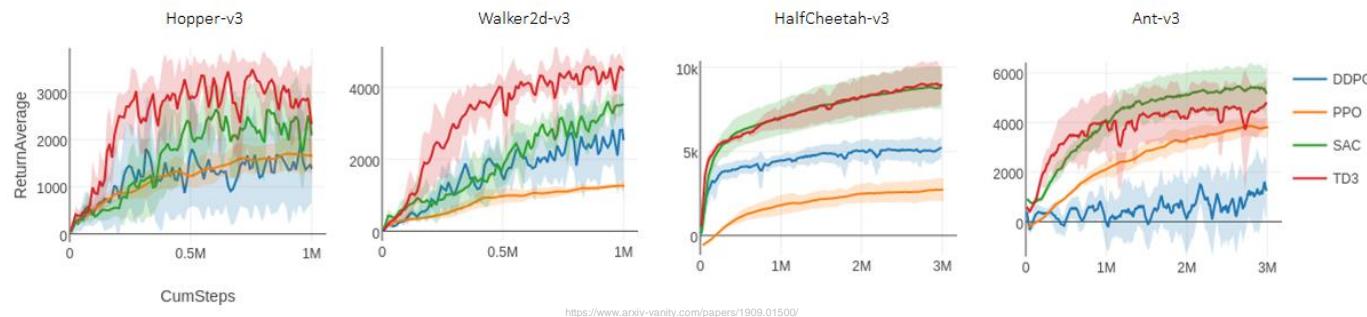
$$L = \left( R + \gamma \hat{Q}(s', a', \mathbf{w}) - \hat{Q}(s, a, \mathbf{w}) \right)^2$$

- the actor  $\hat{\pi}(s, \mathbf{w}')$  is trained through gradient ascent on the critic to output the optimal action for a given state such that the critic output  $\hat{Q}(s, \hat{\pi}(s, \mathbf{w}'), \mathbf{w})$  is maximized



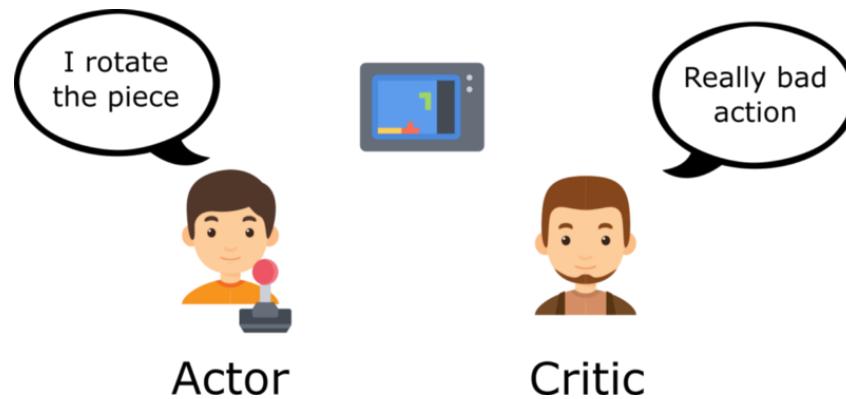
## State-of-the-art RL algorithms

- more additional improvements (similar to experience replay / target networks) lead to state-of-the-art algorithms (e.g. A2C, A3C, DDPG, PPO, TRPO, TD3, SAC, ...)
- rule of thumb: The newer the algorithm is, the better it performs (learns faster)
- Overview of different algorithms:
  - <https://lilianweng.github.io/posts/2018-04-08-policy-gradient/>
  - <https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1>
  - [https://www.ias.informatik.tu-darmstadt.de/uploads/Team/DavideTateo/felix\\_thesis.pdf](https://www.ias.informatik.tu-darmstadt.de/uploads/Team/DavideTateo/felix_thesis.pdf)



## Brief summary

- Deep RL is used whenever the state/action space is continuous
- Deep Sarsa, Deep Q-learning are not used in practice
- DQN is used for continuous state / discrete action spaces
- DQN approximates the Q-function with a neural network
- Actor-critic methods are used for continuous state / continuous action spaces
- Actor-critic methods approximate the Q-function and the policy with neural networks



<https://www.freecodecamp.org/news/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d/>

# Kahoot!

# Methods for policy-based RL

**Programmers:** You can't just rerun your program without changing it and expect it to work

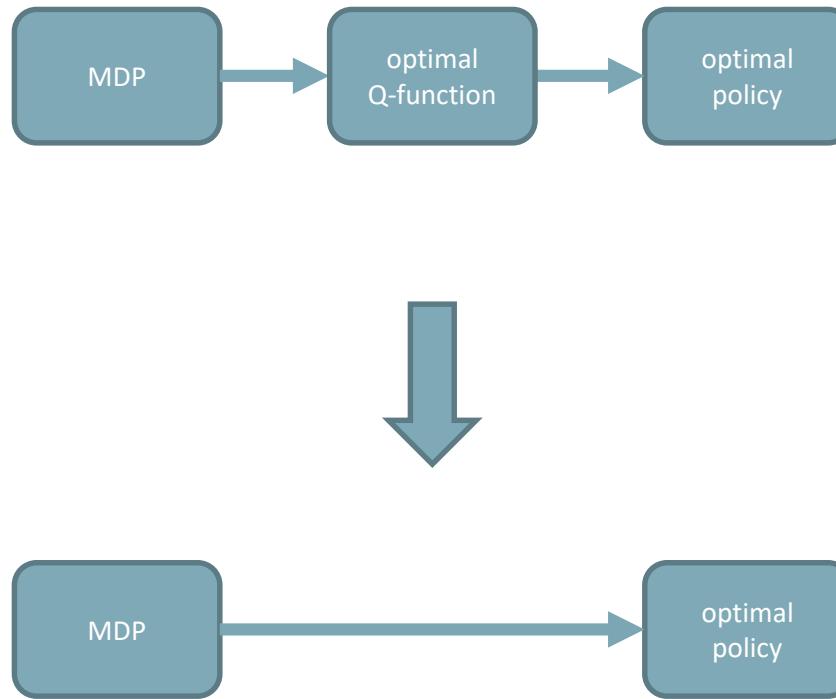
**Reinforcement Learning Practitioners:**



[https://twitter.com/halawa\\_marah/status/1382646849471926272](https://twitter.com/halawa_marah/status/1382646849471926272)

# Methods for policy-based RL

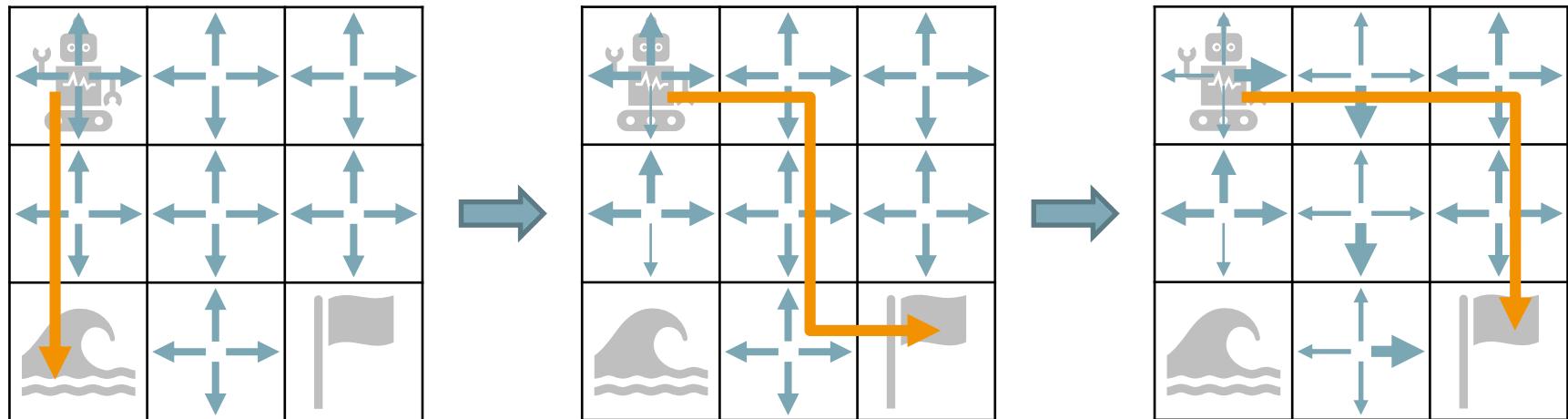
Idea: Obtain an optimal policy without calculating the (V-)/Q-function first



# Methods for policy-based RL

General approach (simplified, discrete actions)

- Run episode and evaluate return
- If the return is high, increase percentage of taking the corresponding action, if the return is low, decrease percentage of taking the corresponding action
- Repeat



# Methods for policy-based RL

## Example: Multi-armed bandits

- Three actions for a given state: Pull lever of slot machine 1/2/3 and get reward
- Special case: Probabilistic (non-deterministic) rewards  $R$
- At time  $t$ , agent selects action  $a_t$  and gets reward  $R_t$
- Goal: Maximize (average) reward  $R_t$



<https://rl-lab.com/multi-armed-bandits/>

# Methods for policy-based RL

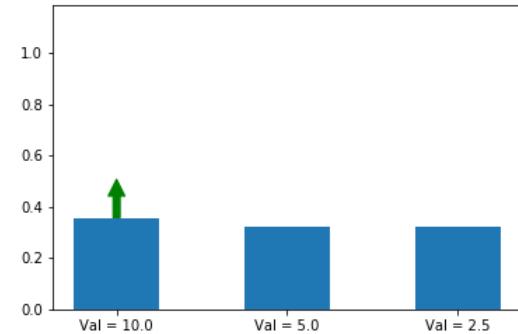
## Intuitive derivation for REINFORCE and baselines

- Start with informed gradient ascent:

Increase probability of taking the optimal action

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \cdot \nabla_{\boldsymbol{\theta}} \pi(a^*|s, \boldsymbol{\theta})$$

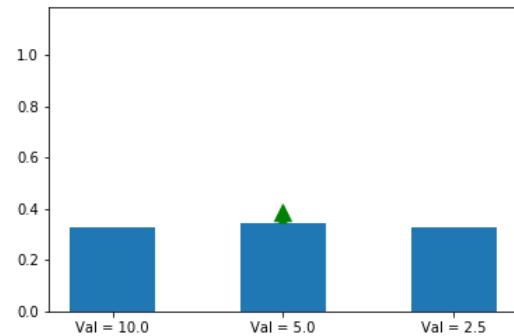
Problem: Optimal action is not known



- Weight gradient with Q-function:

If the action corresponds to a high estimated Q-value  
the gradient step will be large, otherwise small

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \cdot \hat{Q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w}) \nabla_{\boldsymbol{\theta}} \pi(a|s, \boldsymbol{\theta})$$



<https://towardsdatascience.com/an-intuitive-explanation-of-policy-gradient-part-1-reinforce-aa4392cbfd3c>

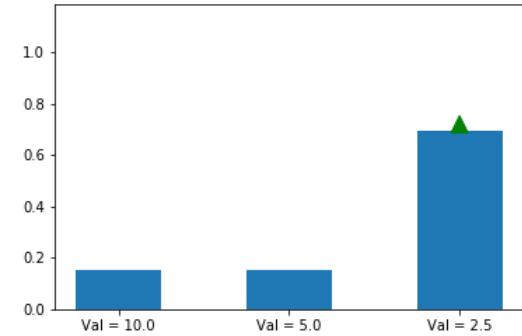
# Methods for policy-based RL

## 3. Problem with

$$\theta \leftarrow \theta + \eta \cdot \hat{Q}_{\pi_\theta}(s, a, w) \nabla_\theta \pi(a|s, \theta)$$

It does not account for the amount of updates:

If a suboptimal action has an initial high probability of being taken, it might get taken/updated more frequently, increasing its probability even more

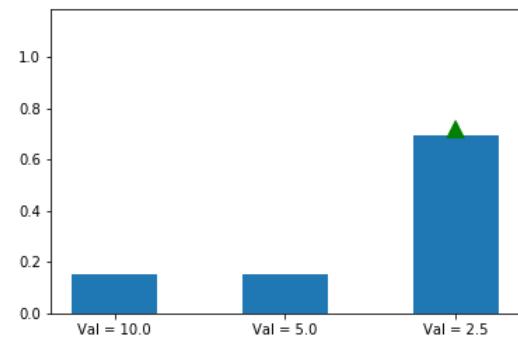


## 4. Weight gradient with inverse of update rate:

$$\theta \leftarrow \theta + \eta \cdot \hat{Q}_{\pi_\theta}(s, a, w) \frac{\nabla_\theta \pi(a|s, \theta)}{\pi(a|s, \theta)}$$

This alleviates the problem described in 3.

With  $\frac{\nabla f(x)}{f(x)} = \ln f(x)$  and  $G_t$  instead of  $Q_{\pi_\theta}(s, a)$   
 this is the REINFORCE update rule



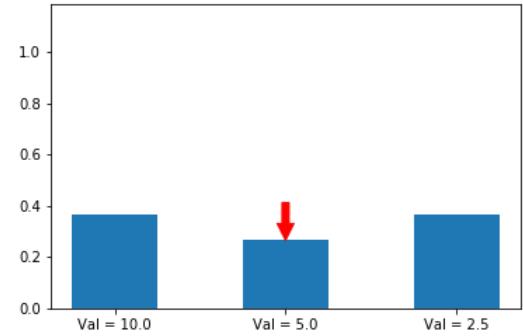
<https://towardsdatascience.com/an-intuitive-explanation-of-policy-gradient-part-1-reinforce-aa4392cbfd3c>

# Methods for policy-based RL

## 5. Effect of advantage function

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \cdot (\hat{Q}_{\pi_{\boldsymbol{\theta}}}(s, a, \mathbf{w}) - \hat{V}(s, \mathbf{w}')_{\pi_{\boldsymbol{\theta}}}) \frac{\nabla_{\boldsymbol{\theta}} \pi(a|s, \boldsymbol{\theta})}{\pi(a|s, \boldsymbol{\theta})}$$

Actions corresponding to bad Q-values are now actively decreased in probability through gradient descent



<https://towardsdatascience.com/an-intuitive-explanation-of-policy-gradient-part-1-reinforce-aa4392cbfd3c>

# Methods for policy-based RL

General approach (mathematical)

**Policy gradient:** Gradient ascent over policy  $\pi_{\theta} = \pi(a|s, \theta)$  with parameterizable weights  $\theta$

1. Set up cost function based on policy, e.g. V-function of start state  $S_0$  for episodic tasks

$$J = V_{\pi_{\theta}}(S_0)$$

2. Calculate gradient w.r.t. weights  $\theta$

$$\nabla_{\theta} J = \nabla_{\theta} V_{\pi_{\theta}}(S_0)$$

3. Optimize weights  $\theta$  through gradient ascent to improve the policy and go to 1.

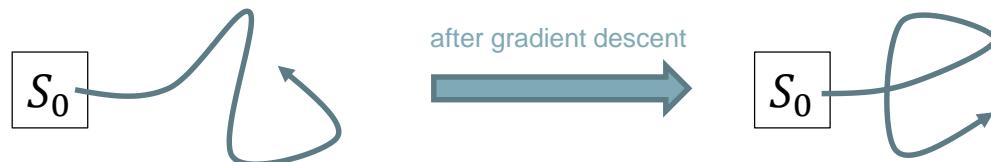
$$\theta \leftarrow \theta + \eta \cdot \nabla_{\theta} J$$

# Methods for policy-based RL

Problem with  $\nabla_{\theta} J = \nabla_{\theta} V_{\pi_{\theta}}(S_0)$ :

- The policy  $\pi_{\theta} = \pi(a|s, \theta)$  is a function of the state(s)  $s$
- Hence when calculating the gradient  $\nabla_{\theta} V_{\pi_{\theta}}(S_0)$  and using the chain rule, one has to calculate  $\frac{\partial s}{\partial \theta}$  at some point
- But this is impossible to calculate. It means that we have to calculate how the distribution of states  $s$  changes depending on  $\theta$
- This is something that depends on the (usually unknown) environment

Problem in simple terms: Changing the policy will change the visited states, but we usually don't know how



# Methods for policy-based RL

## Policy gradient theorem

- Allows to calculate the gradient  $\nabla_{\theta}L$  without knowing the state distribution change  $\frac{\partial s}{\partial \theta}$  as

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

↗ directly proportional  
↗ state distribution (percentage of being in each state)  
↗ summation over actions in each state

- Derivation shown on next slide (not important)

# Methods for policy-based RL

## Derivation of the policy gradient theorem (not important)

### Proof of the Policy Gradient Theorem (episodic case)

With just elementary calculus and re-arranging of terms, we can prove the policy gradient theorem from first principles. To keep the notation simple, we leave it implicit in all cases that  $\pi$  is a function of  $\theta$ , and all gradients are also implicitly with respect to  $\theta$ . First note that the gradient of the state-value function can be written in terms of the action-value function as

$$\begin{aligned}
 \nabla v_\pi(s) &= \nabla \left[ \sum_a \pi(a|s) q_\pi(s, a) \right], \quad \text{for all } s \in \mathcal{S} \quad (\text{Exercise 3.18}) \\
 &= \sum_a \left[ \nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \nabla q_\pi(s, a) \right] \quad (\text{product rule of calculus}) \\
 &= \sum_a \left[ \nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \nabla \sum_{s', r} p(s', r|s, a) (r + v_\pi(s')) \right] \\
 &\qquad\qquad\qquad (\text{Exercise 3.19 and Equation 3.2}) \\
 &= \sum_a \left[ \nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s'|s, a) \nabla v_\pi(s') \right] \quad (\text{Eq. 3.4}) \\
 &= \sum_a \left[ \nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s'|s, a) \right. \\
 &\qquad\qquad\qquad \left. \sum_{a'} [\nabla \pi(a'|s') q_\pi(s', a') + \pi(a'|s') \sum_{s''} p(s''|s', a') \nabla v_\pi(s'')] \right] \\
 &= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \rightarrow x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x, a),
 \end{aligned}$$

after repeated unrolling, where  $\Pr(s \rightarrow x, k, \pi)$  is the probability of transitioning from state  $s$  to state  $x$  in  $k$  steps under policy  $\pi$ . It is then immediate that

$$\begin{aligned}
 \nabla J(\theta) &= \nabla v_\pi(s_0) \\
 &= \sum_s \left( \sum_{k=0}^{\infty} \Pr(s_0 \rightarrow s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s, a) \\
 &= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \quad (\text{box page 199}) \\
 &= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s) q_\pi(s, a) \\
 &= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \quad (\text{Eq. 9.3}) \\
 &\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \quad (\text{Q.E.D.})
 \end{aligned}$$

# Methods for policy-based RL

## All-actions policy update

- Reformulation of the policy gradient theorem yields

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

- Policy update through stochastic gradient descent
  - Run an episode
  - At each step of the episode, update weights using stochastic gradient ascent based on the approximated Q-function  $\hat{Q}$  and the policy gradient  $\nabla_{\theta} \pi_{\theta}$  of each visited state

$$\theta \leftarrow \theta + \eta \cdot \sum_a \hat{Q}_{\pi_{\theta}}(S_t, a, w) \nabla_{\theta} \pi(a|S_t, \theta)$$

Problem: All-actions policy update still depends on approximated Q-function  $\hat{Q}$

# Methods for policy-based RL

## REINFORCE

- Like all action-action policy update, but does not depend on an approximated Q-function
- Reformulation of the policy gradient theorem yields

$$\begin{aligned}\nabla_{\theta} J &\propto \sum_s \mu(s) \sum_a Q_{\pi_{\theta}}(s, a) \nabla_{\theta} \pi(a|S_t, \theta) \\ &= E_{\pi_{\theta}}[G_t \nabla_{\theta} \ln \pi(A_t|S_t, \theta)]\end{aligned}$$

- Derivation shown on next slide (not important)
- Policy update using REINFORCE
  - Run an episode
  - At each step of the episode, update weights using stochastic gradient ascent based on the sampled return  $G_t$  and the policy gradient  $\nabla_{\theta} \ln \pi_{\theta}$  of each visited state

$$\theta \leftarrow \theta + \eta \cdot G_t \nabla_{\theta} \ln \pi(A_t|S_t, \theta)$$


  
 weighting factor      gradient of the log prob.

# Methods for policy-based RL

Derivation of REINFORCE (not important)

$$\begin{aligned}
 \nabla_{\boldsymbol{\theta}} J &\propto \sum_s \mu(s) \sum_a Q_{\pi_{\boldsymbol{\theta}}}(s, a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}} \\
 &= E_{\pi_{\boldsymbol{\theta}}} \left[ \sum_a Q_{\pi_{\boldsymbol{\theta}}}(S_t, a) \nabla_{\boldsymbol{\theta}} \pi(a|S_t, \boldsymbol{\theta}) \right] \\
 &= E_{\pi_{\boldsymbol{\theta}}} \left[ \sum_a \pi(a|S_t, \boldsymbol{\theta}) Q_{\pi_{\boldsymbol{\theta}}}(S_t, a) \frac{\nabla_{\boldsymbol{\theta}} \pi(a|S_t, \boldsymbol{\theta})}{\pi(a|S_t, \boldsymbol{\theta})} \right] \\
 &= E_{\pi_{\boldsymbol{\theta}}} \left[ Q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right] \\
 &= E_{\pi_{\boldsymbol{\theta}}} \left[ G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right] \\
 &= E_{\pi_{\boldsymbol{\theta}}} [G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})]
 \end{aligned}$$

expand with  $\frac{\pi(a|S_t, \boldsymbol{\theta})}{\pi(a|S_t, \boldsymbol{\theta})}$

replace  $a$  by the sample  $A_t \sim \pi$

$$E_{\pi}[G_t|S_t, A_t] = Q_{\pi}(S_t, A_t)$$

$$\nabla \ln f(x) = \frac{1}{f(x)} \nabla f(x)$$

# Methods for policy-based RL

## REINFORCE in literature

### REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for $\pi_*$

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

Algorithm parameter: step size  $\alpha > 0$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  (e.g., to  $\mathbf{0}$ )

Loop forever (for each episode):

Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ , following  $\pi(\cdot|s, \theta)$

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

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

Sutton, Barto: Reinforcement Learning

# Methods for policy-based RL

## Baselines

- Recap: Policy gradient theorem

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

- Problem with above formulation: Huge variation of the gradient magnitude  $|\nabla_{\theta} L|$  across different states depending on  $Q_{\pi_{\theta}}(s, a)$  → numerical problems
- Solution: Introduce a baseline (a function  $b(s)$  that only depends on the state) to make the gradient magnitude more similar across all states

$$\nabla_{\theta} J \propto \sum_s \mu(s) \sum_a (Q_{\pi_{\theta}}(s, a) - b(s)) \nabla_{\theta} \pi_{\theta}$$

- Does not change expected value of  $\nabla_{\theta} L$ , because

$$\sum_a b(s) \nabla_{\theta} \pi_{\theta} = b(s) \nabla_{\theta} \sum_a \pi_{\theta} = b(s) \nabla_{\theta} 1 = 0$$

# Methods for policy-based RL

- Common type of baseline: V-function  

$$b(s) = V_\pi(s)$$
- Resulting term  $Q_\pi(s, a) - V_\pi(s)$  is called **advantage function**  $A_\pi(s, a)$
- Resulting update rule for REINFORCE with baseline  
 (with approximate value function  $\hat{V}_{\pi_\theta}(s, w)$ )

$$\theta \leftarrow \theta + \eta \cdot (G_t - \hat{V}_{\pi_\theta}(S_t, w)) \nabla_\theta \ln \pi(A_t | S_t, \theta)$$

For comparison: REINFORCE update rule without baseline

$$\theta \leftarrow \theta + \eta \cdot G_t \nabla_\theta \ln \pi(A_t | S_t, \theta)$$

- Challenge: Two neural networks to learn:  
 Policy  $\pi(a|s, \theta)$  and V-function approximation  $\hat{V}_{\pi_\theta}(s, w)$

# Methods for policy-based RL

## REINFORCE with baseline in literature

### REINFORCE with Baseline (episodic), for estimating $\pi_{\theta} \approx \pi_*$

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

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

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

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $w \in \mathbb{R}^d$  (e.g., to  $\mathbf{0}$ )

Loop forever (for each episode):

Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ , following  $\pi(\cdot|s, \theta)$

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

$$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k \quad (G_t)$$

$$\delta \leftarrow G - \hat{v}(S_t, w)$$

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

$$\theta \leftarrow \theta + \alpha^{\theta} \gamma^t \delta \nabla \ln \pi(A_t | S_t, \theta)$$

Sutton, Barto: Reinforcement Learning

# Methods for policy-based RL

Task: Implement policy gradients for a very simple neural network with only one weight and two actions:

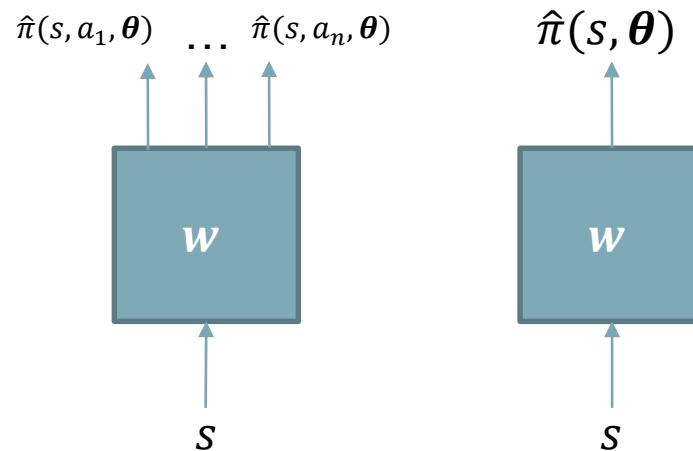
- The probability of taking action 1 is calculated as  $p(a_1) = \frac{1}{1+e^{-w}}$
- The probability of taking action 2 is calculated as  $p(a_2) = \frac{1}{1+e^w} = 1 - p(a_1)$
- It is assumed that the Q-value of action 1 is 1 and the Q-value of action 2 is 2
- $\eta = 10^{-3}$

What is the probability to take each action after 100000 update steps if the initial value of  $w$  is -5/0/5 and the gradient is weighted with / is not weighted with the inverse of the update rate?

# Methods for policy-based RL

Different types of trained neural networks for  $\hat{\pi}(s, \theta)$

- Discrete action space (e.g. REINFORCE):  
Neural network with one output per action, representing its probability
- Continuous action space (e.g. Actor-Critic):  
Neural network with one output corresponding to the currently best action



# Methods for policy-based RL

## Actor-critic methods

- Actor-critic = REINFORCE + baseline + Bellman expectation equation
- Recap: REINFORCE + baseline update rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \cdot (G_t - \hat{V}_{\pi_{\boldsymbol{\theta}}}(S_t, \mathbf{w})) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t | S_t, \boldsymbol{\theta})$$

- Actor-critic update rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \cdot (R_{t+1} + \gamma \hat{V}_{\pi_{\boldsymbol{\theta}}}(S_{t+1}, \mathbf{w}) - \hat{V}_{\pi_{\boldsymbol{\theta}}}(S_t, \mathbf{w})) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t | S_t, \boldsymbol{\theta})$$

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \cdot \dots$$

- Basis for many state-of-the-art reinforcement algorithms  
(e.g. A2C, A3C, DDPG, PPO, TRPO, TD3, SAC, ...)

# Methods for policy-based RL

## Actor-critic in literature

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

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

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

Parameters: step sizes  $\alpha_{\theta} > 0$ ,  $\alpha_w > 0$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $w \in \mathbb{R}^d$  (e.g., to  $\mathbf{0}$ )

Loop forever (for each episode):

    Initialize  $S$  (first state of episode)

$I \leftarrow 1$

    Loop while  $S$  is not terminal (for each time step):

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

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

$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$       (if  $S'$  is terminal, then  $\hat{v}(S', w) \doteq 0$ )

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

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

$I \leftarrow \gamma I$

$S \leftarrow S'$

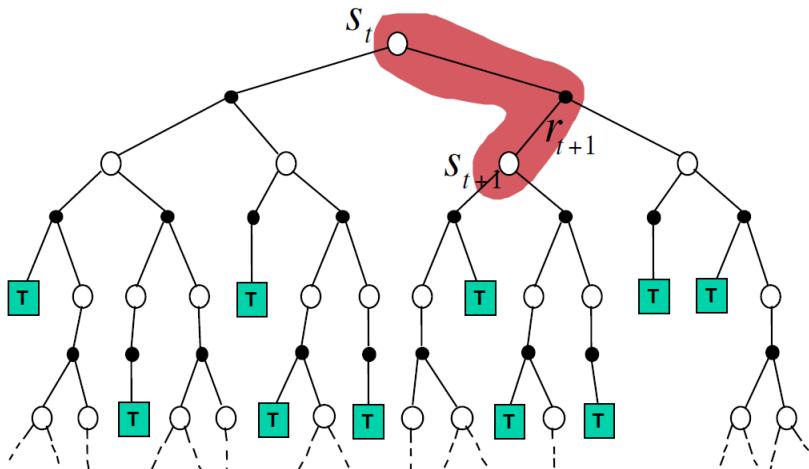
Sutton, Barto: Reinforcement Learning

# Methods for policy-based RL

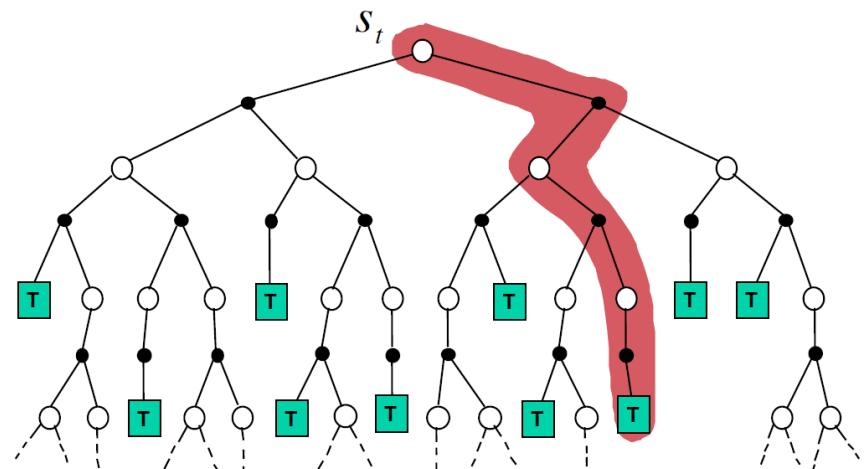
## Comparison of actor-critic and REINFORCE

- Actor-critic methods are the analog of temporal difference methods for policy gradients
- REINFORCE is the analog of Monte-Carlo methods for policy gradients

Actor-critic

$$\theta \leftarrow \theta + \eta \cdot \left( R_{t+1} + \gamma \hat{V}(S_{t+1}, w) - \hat{V}(S_t, w') \right) \nabla_{\theta} \ln \pi(A_t | S_t, \theta)$$


REINFORCE

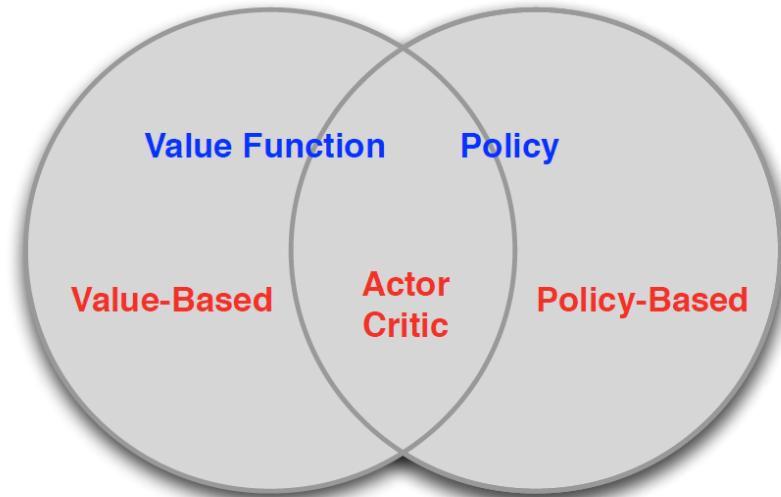
$$\theta \leftarrow \theta + \eta \cdot G_t \nabla_{\theta} \ln \pi(A_t | S_t, \theta)$$


<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

# Methods for policy-based RL

Comparison of value-based, policy-based and actor-critic methods

- value-based
  - learned value function
  - implicit policy (e.g.  $\epsilon$ -greedy)
- policy-based
  - no value function
  - learned policy
- actor-critic
  - learned value function
  - learned policy



<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

Which one is best?

- Usually actor-critic, but depends on use-case (discrete/continuous states/actions)

# Methods for policy-based RL

Task: What are the possible (dis-)advantages of value-based / policy-based / actor critic methods?

# Methods for policy-based RL

## Brief summary

- Policy-based methods directly update the policy without (in principle) having to calculate the V/Q-function first (policy gradient)
- The policy gradient theorem is the foundation of all policy gradient methods as it allows to calculate the policy gradient based on sampled trajectories
- REINFORCE = policy gradient theorem + return instead of Q-function
- REINFORCE + baseline = REINFORCE + baseline
- Actor-critic = REINFORCE + baseline + Bellman expectation equation

# Kahoot!

# Games

Goal: Train a RL agent to master classic, turn-based games (e.g. Chess, Scrabble, Go, ...)

In contrast to all previous RL methods, the turn-based structure of the problem and the deterministic output of each action allows the creation of a search tree for better play/learning

Three methods are presented for playing

- Monte-Carlo search
- Monte-Carlo Tree Search (+ UCT)
- Minimax search (+ alpha-beta pruning)

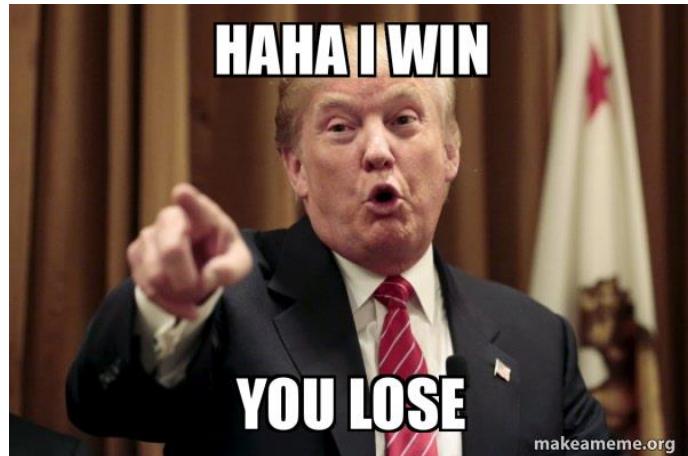
Different methods are presented for learning

- TD-Root, TD-Leaf, Treestrap and Rootstrap

## Introduction

This chapter focuses on

- **two-player games** with two alternating players (named black and white)
- **zero-sum games** with equal and opposite rewards for both players at the end of the game (e.g. if black gets a positive reward for a win, white gets a negative reward for a loss)



<https://makeameme.org/meme/haha-i-win-62bac8a79d>

## Best response

- Every player tries to optimize his policy  $\pi^i$  with respect to the opponents policy  $\pi^{-i}$
- The best response  $\pi_*^i(\pi^{-i})$  is an optimal policy against a fixed opponent's policy

## Nash equilibrium

- is a joint policy for all players

$$\pi = \pi_*^i(\pi^{-i})$$

- such that every player's policy is a best response
- i.e. no player would choose to deviate from the Nash equilibrium
- colloquially, changing an agent's policy in a Nash equilibrium only makes it worse for the player



imgflip.com

<https://imgflip.com/memegenerator>

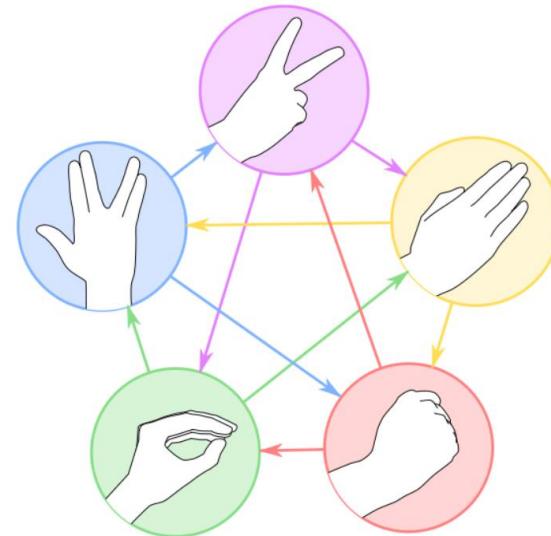
## Example: Rock Paper Scissors

- Assuming that one player always plays "Rock" ( $\pi^{-i} = \text{"Rock"}$ ), the best response is to play always "Paper" ( $\pi_*^i(\pi^{-i}) = \text{"Paper"}$ )
- The nash equilibrium is to play all three options with equal probability

Task: What is the best response for

- Rock Paper Scissors
- Rock Paper Scissors Lizard Spock

if your opponent randomly plays  
"Rock" or "Paper"



[https://bigbangtheory.fandom.com/wiki/Rock,\\_Paper,\\_Scissors,\\_Lizard,\\_Spock?file=RPSLS.png](https://bigbangtheory.fandom.com/wiki/Rock,_Paper,_Scissors,_Lizard,_Spock?file=RPSLS.png)

## Task: Prisoner's dilemma

"Two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of speaking to or exchanging messages with the other. The police admit they don't have enough evidence to convict the pair on the principal charge. They plan to sentence both to a year in prison on a lesser charge. Simultaneously, the police offer each prisoner a Faustian bargain. If he testifies against his partner, he will go free while the partner will get three years in prison on the main charge. Oh, yes, there is a catch ... If *both* prisoners testify against each other, both will be sentenced to two years in jail. The prisoners are given a little time to think this over, but in no case may either learn what the other has decided until he has irrevocably made his decision. Each is informed that the other prisoner is being offered the very same deal. Each prisoner is concerned only with his own welfare—with minimizing his own prison sentence." ([https://en.wikipedia.org/wiki/Prisoner%27s\\_dilemma](https://en.wikipedia.org/wiki/Prisoner%27s_dilemma))

What is the best response for prisoner A if prisoner B stays silent?

What is a Nash equilibrium for the prisoner's dilemma?

		Prisoner B	Prisoner B stays silent (cooperates)	Prisoner B testifies (defects)
		Prisoner A		
Prisoner A	Prisoner A stays silent (cooperates)		Each serves 1 year	Prisoner A: 3 years Prisoner B: goes free
	Prisoner A testifies (defects)		Prisoner A: goes free Prisoner B: 3 years	Each serves 2 years

[https://en.wikipedia.org/wiki/Prisoner%27s\\_dilemma](https://en.wikipedia.org/wiki/Prisoner%27s_dilemma)

## Single-agent RL for games

- The best response is the solution to the single-agent RL problem where
  - the other player becomes part of the environment
  - the game becomes an MDP
  - the best response is the optimal policy for this MDP

## Self-play

- Games are played between two agents which alternatingly optimize their policy through single-agent RL for games
- If the resulting joint policy becomes a fixed-point (i.e. does not change anymore), a Nash equilibrium is found

Task: Assuming that one player always plays "Rock" for Rock Paper Scissors at the beginning, does the joint policy eventually converge to a Nash equilibrium using self-play?

Task: What might be a useful value-function/reward for self-play RL

- if one can only win or lose?
- if a draw is also possible?

## Minimax

- A value function defines the expected total reward given joint policies  $\pi = (\pi^i, \pi^{-i})$

$$V_\pi(s) = E_\pi [ G_t \mid S_t = s ]$$

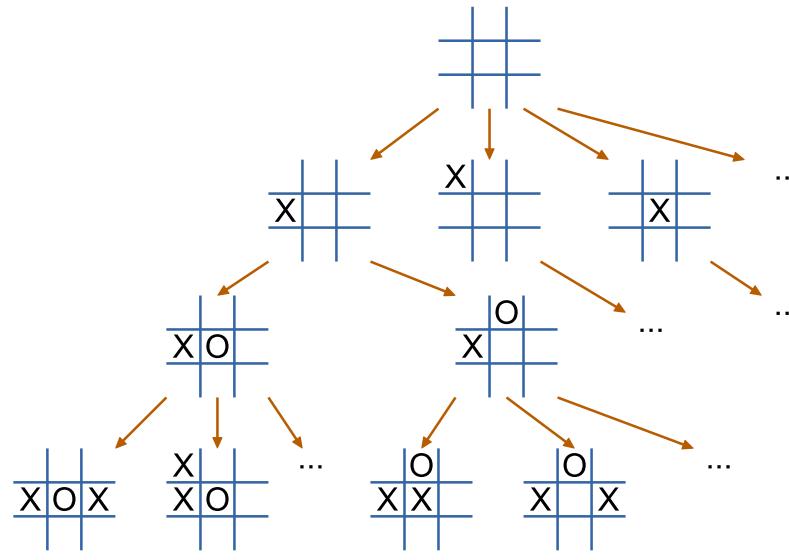
- A minimax value function maximizes the expected return of one player while minimizing the expected return of the other player

$$V_*(s) = \max_{\pi^i} \min_{\pi^{-i}} V_\pi(s)$$

- A minimax policy is a joint policy that achieves the minimax value function
- There is a unique minimax value function
- Any minimax policy is a Nash equilibrium
- Self-play MC evaluation / TD learning converges to the minimax value function (often  $\gamma = 1$  because a reward is only given at the end of the game)

## Search tree

- A search tree represents possible sequences of moves
- Each node of the search tree represents a state of the game
- Each edge of the search tree represents an action of one of the players



[https://ksv.mff.cuni.cz/~dingle/2019-20/prog\\_2/notes\\_13.html](https://ksv.mff.cuni.cz/~dingle/2019-20/prog_2/notes_13.html)

Task: How can a search tree be used for games?

## Using a search tree or not

- In theory, using a search tree is not necessary for a (Nash equilibrium) optimal policy
- In reality, when approximating policies with NNs, an optimal policy is rarely found
  - not all states can be evaluated
  - imperfect approximation / highly discontinuous value function
- A search tree helps to evaluate the next moves for a given game situation, thus shifting complexity away from the neural network. It also speeds up agent training
- The decision whether to use a search tree and/or DRL for an agent depends on the game and the complexity of the used agent

- DRL
  - no search tree
- all skill from policy

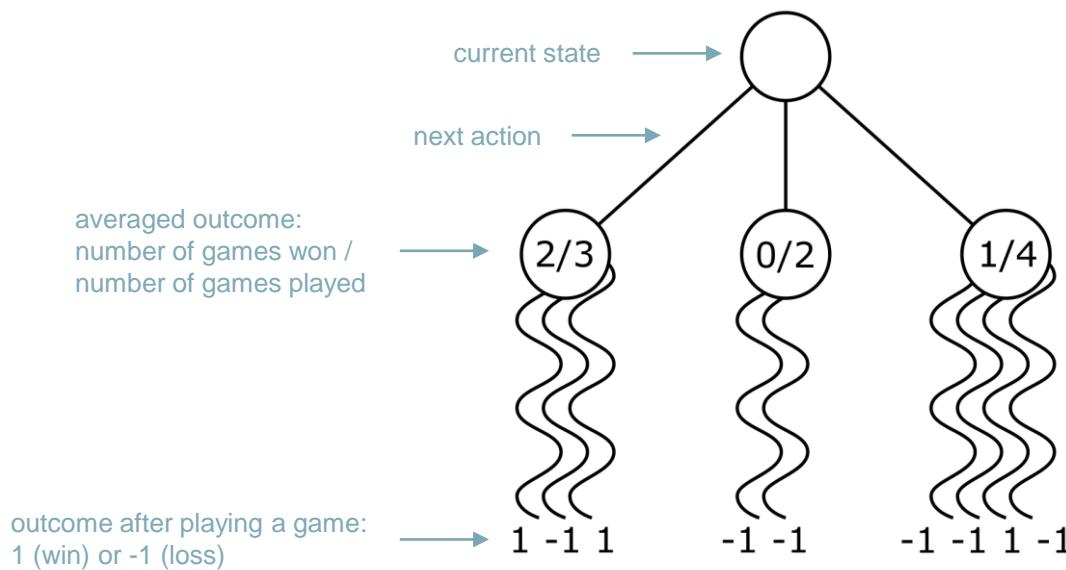
- DRL
  - search tree
- skill from policy / search tree

- no DRL
  - search tree
- all skill from search tree



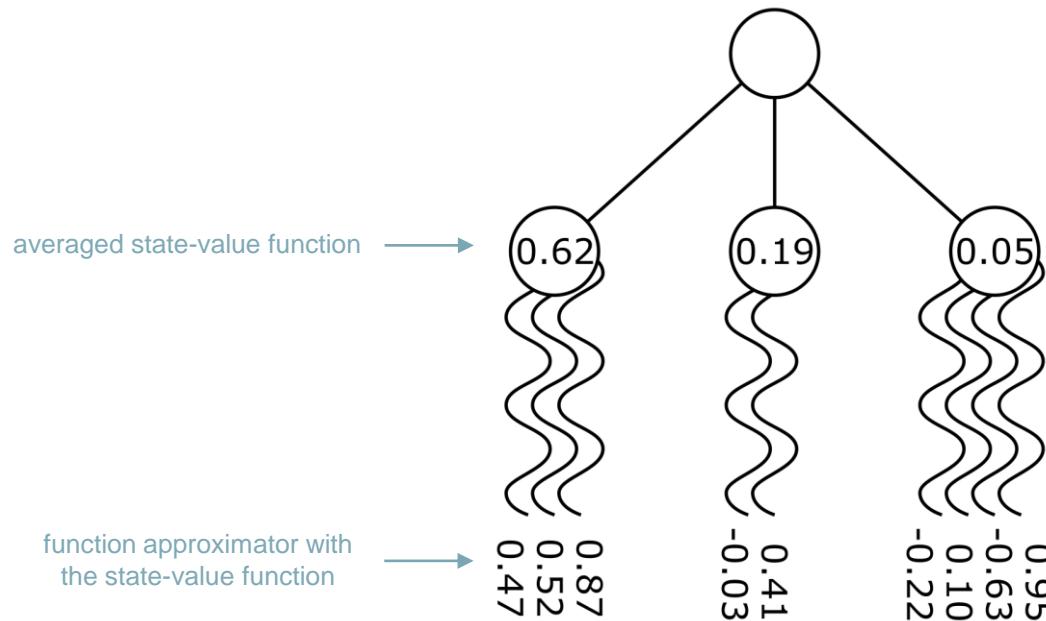
## Monte-Carlo search (MC search)

- Idea: Generate the search tree by randomly playing games (rollouts).  
Take action with the best averaged outcome
- Not very precise, best used for games with a lot of randomness (e.g. dice games)

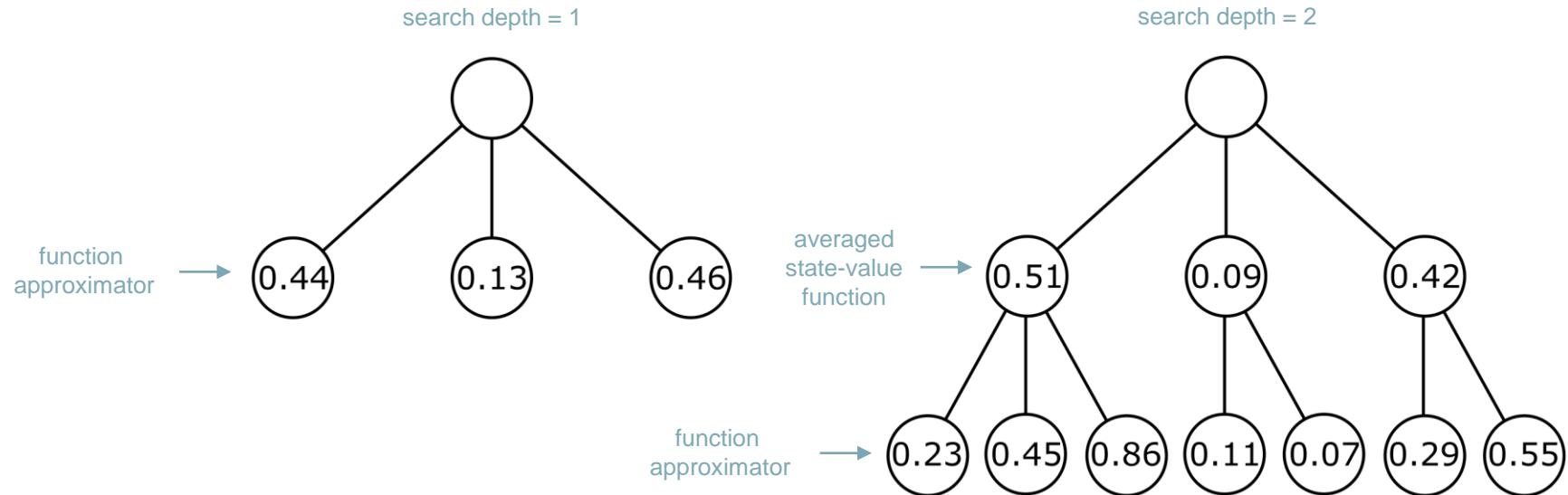


## Function approximators for games

- Note that one can always use a function approximator (e.g. a learned state-value function) to evaluate the outcome of state instead of playing a complete game

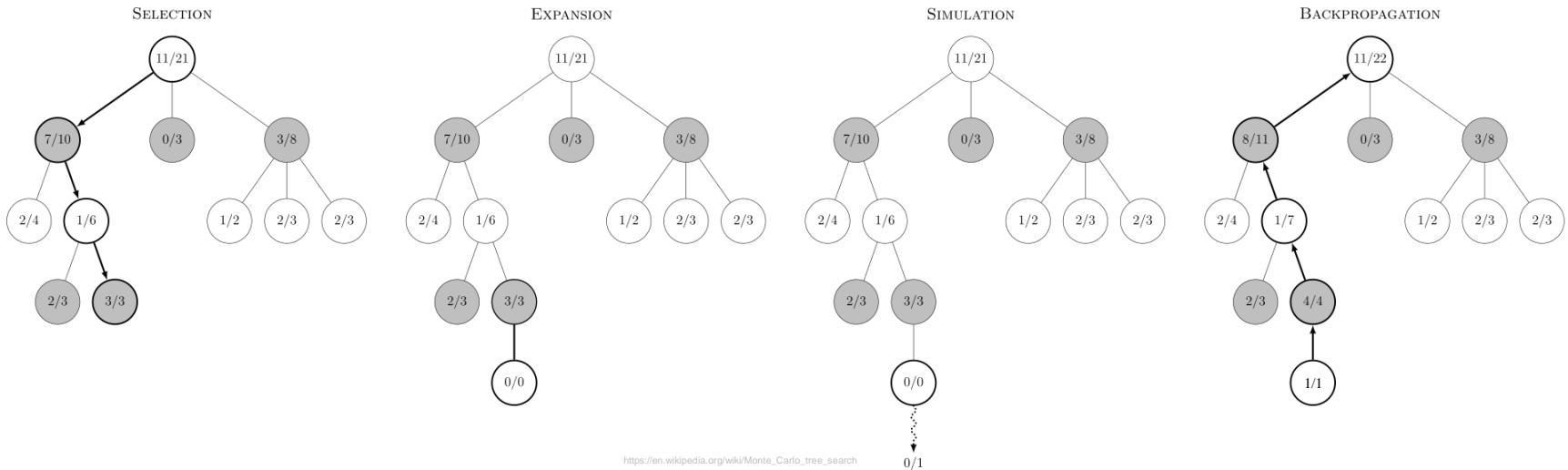


- It is up to the user at which depth of the search tree the function approximator is used
- A lower search depth is computationally more efficient, a higher search depth can be more precise



## Monte-Carlo Tree Search (MCTS)

- Idea: Overcomes the problem of "plain" MC search of choosing actions randomly
- It consists of four phases
  - Selection: Traverse the tree from the root until a new leaf node is selected
  - Expansion: Add the leaf node to the tree
  - Simulation: Complete one randomly played game
  - Backpropagation: Update all nodes towards the root with the new result



## Selection strategies for MCTS

- The node selection of MCTS is the key for a proper exploration/exploitation tradeoff
- Two methods are presented: UCT and PUCT

### UCT (UCB applied to trees)

- when traversing the tree, select the node with the highest value of

$$\frac{w_i}{n_i} + c \sqrt{\frac{\ln N_i}{n_i}}$$

$w_i$  number of wins for the node  
 $n_i$  number of rollouts for the node  
 $N_i$  total number of rollouts for the parent node  
 $c$  exploration parameter (was  $\sqrt{2}$  for UCB1)

### PUCT (e.g. AlphaZero)

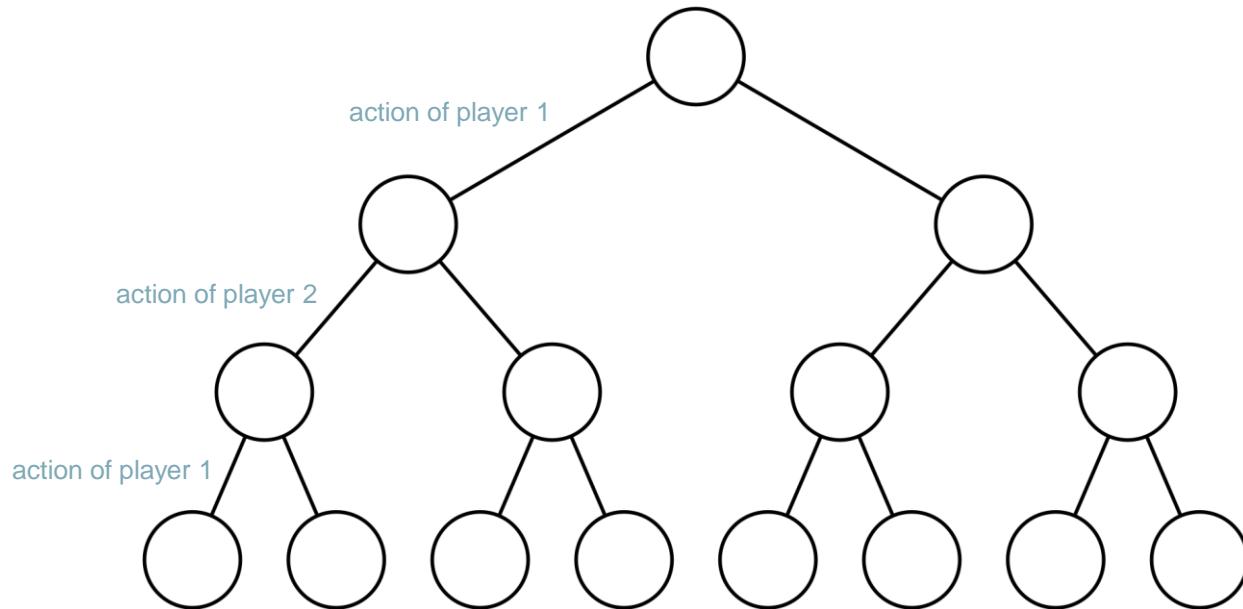
- when traversing the tree, select the node with the highest value of

$$\frac{w_i}{n_i} + cp \frac{\sqrt{N_i}}{1+n_i}$$

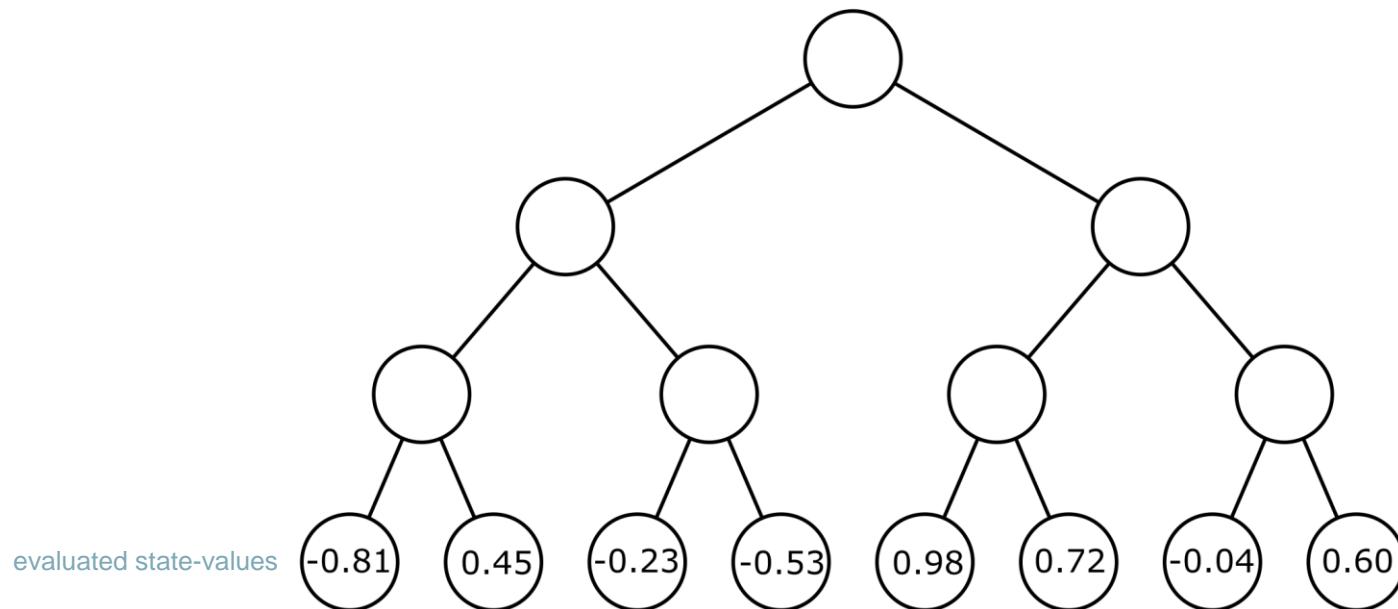
$p$  probability from a neural network of choosing the next node

## Minimax search

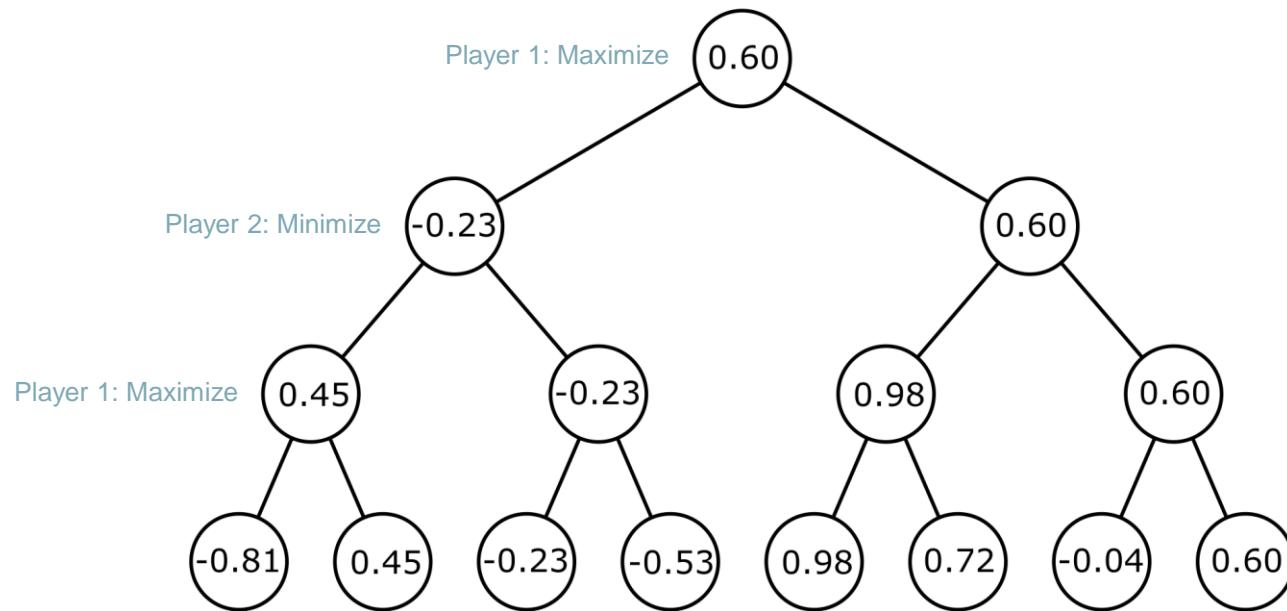
- Idea: Create a search tree while assuming that each player only picks his "best" possible actions
- Step 1: Create a search tree up to a certain depth (simplified example with only two actions per state)



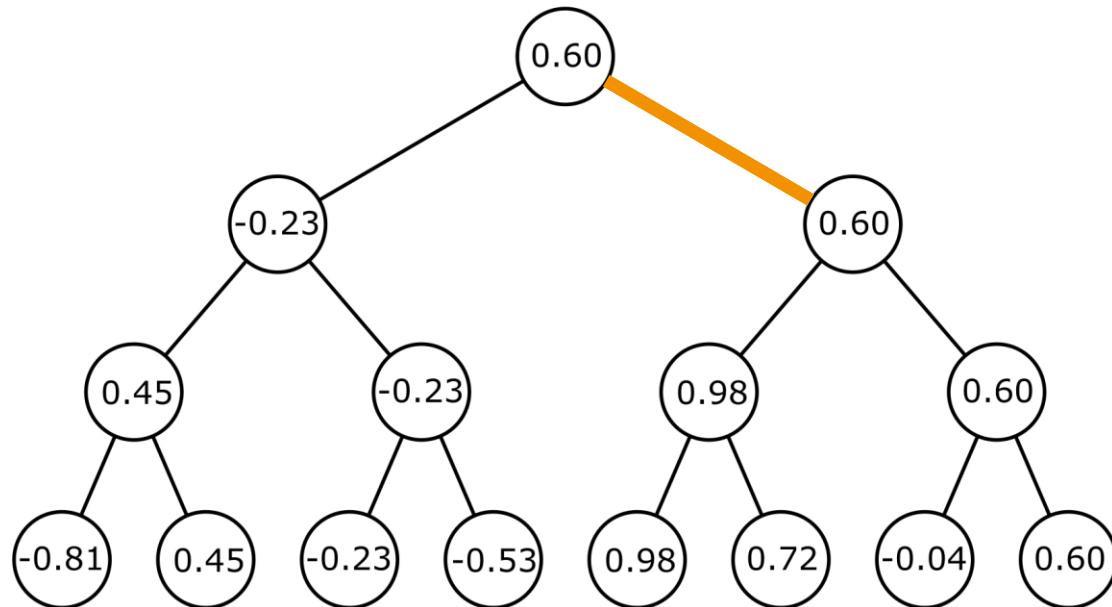
- Step 2: Evaluate the state-values of the leaf nodes  
(either by using a function approximator or MC evaluation, i.e. playing multiple games)



- Step 3: Update the state-values backwards towards the root node
  - When choosing between multiple actions, player 1 will select the action that leads to the largest state value. Player 2 will select the action that leads to the smallest state value



- Step 4: Execute the next action leading to the largest state-value
- Step 5: Wait until the other player has finished his turn and repeat step 1-5



# Games

- Minimax algorithm as pseudocode

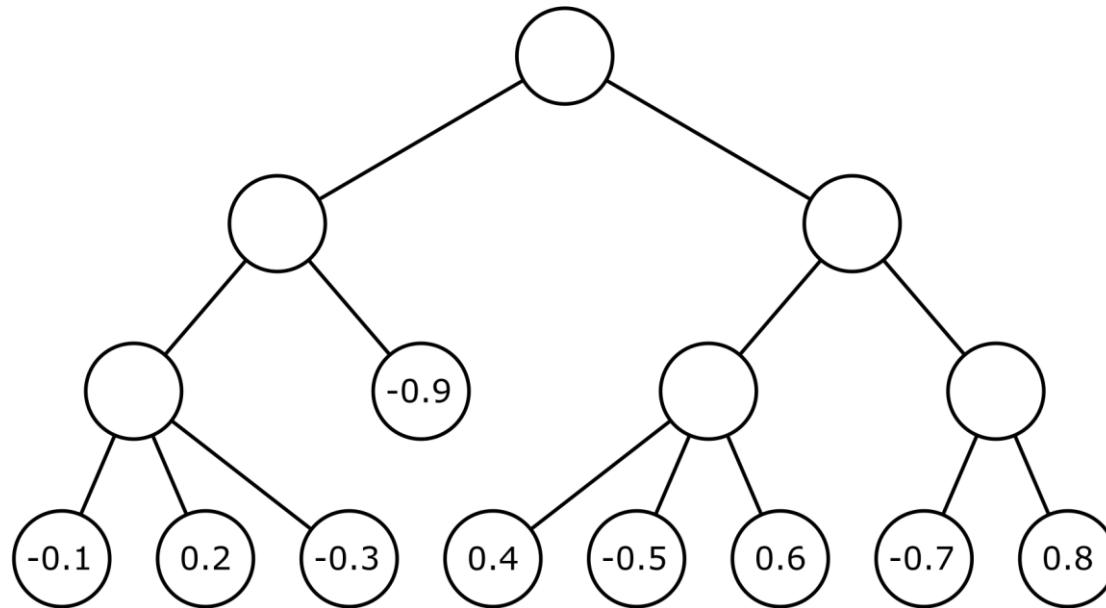
```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := -∞
        for each child of node do
            value := max(value, minimax(child, depth - 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth - 1, TRUE))
        return value
```

```
(* Initial call *)
minimax(origin, depth, TRUE)
```

<https://en.wikipedia.org/wiki/Minimax>

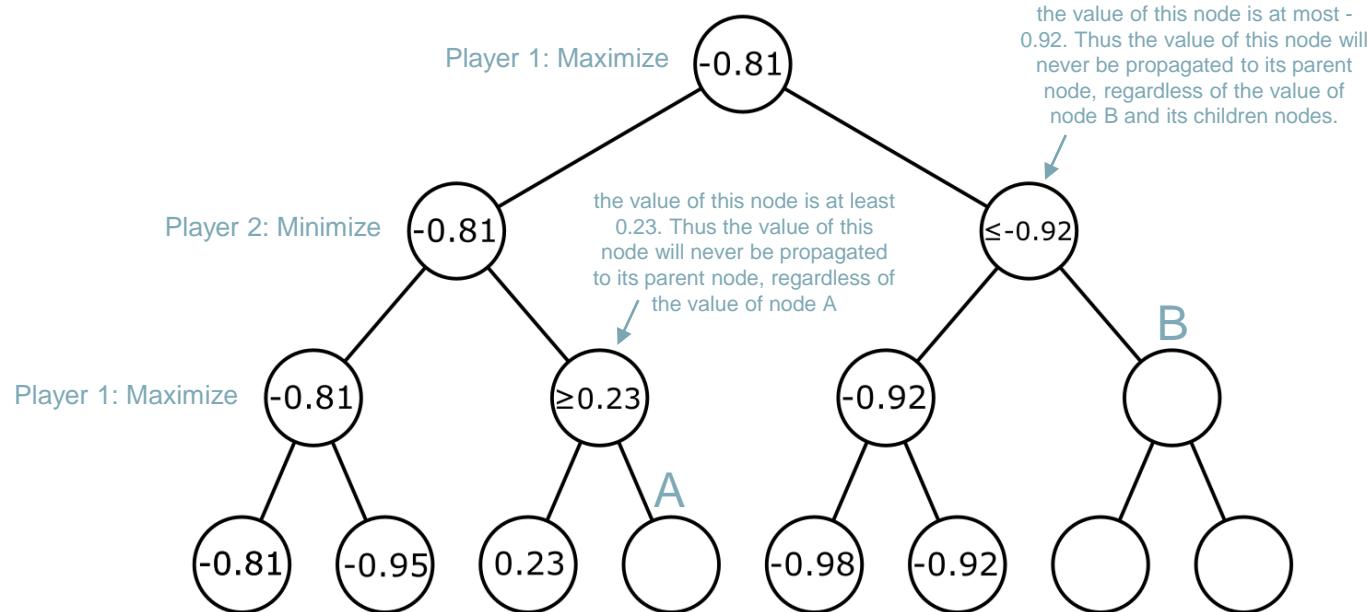
Task: Which action would player 1 choose?

Task: What does the state with the value -0.9 represent? Under which circumstances is a value of -0.9 (somewhat) reasonable?



## Alpha-beta pruning

- is a modification of the original Minimax search in order to reduce the number of evaluated nodes while leading to the same result
- Idea: Check intelligently during backward updates which branches do not need to be evaluated



# Games

- Alpha-beta pruning has its name due to two values ( $\alpha$  and  $\beta$ ) which keep track of the maximum/minimum score that the minimizing/maximizing player can obtain.
- Effect of alpha-beta pruning (optimal case with branching factor b)

number of leaves with depth n and b = 40		
depth n	$b^n$	$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$
0	1	1
1	40	40
2	1,600	79
3	64,000	1,639
4	2,560,000	3,199
5	102,400,000	65,569
6	4,096,000,000	127,999
7	163,840,000,000	2,623,999
8	6,553,600,000,000	5,119,999

<https://www.chessprogramming.org/Alpha-Beta>

Task: Look at n=2 in above table. What is the structure of the two search trees?

- Alpha-beta pruning as pseudocode

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth == 0 or node is terminal then
        return the heuristic value of node
    if maximizingPlayer then
        value := -∞
        for each child of node do
            value := max(value, alphabeta(child, depth - 1, α, β, FALSE))
            if value > β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth - 1, α, β, TRUE))
            if value < α then
                break (* α cutoff *)
            β := min(β, value)
        return value
```

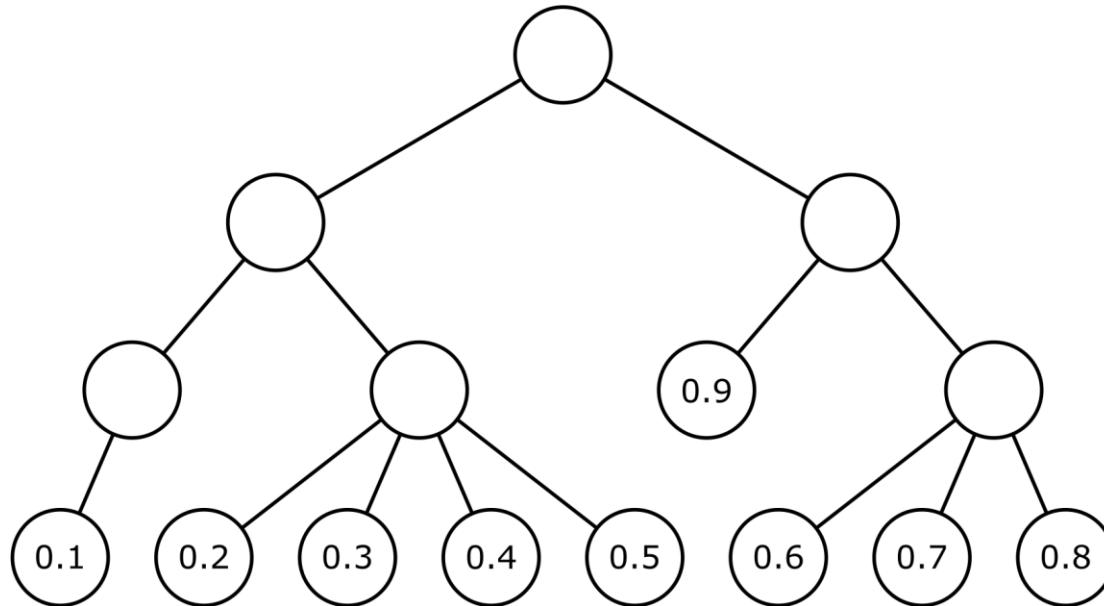
```
(* Initial call *)
alphabeta(origin, depth, -∞, +∞, TRUE)
```

[https://en.wikipedia.org/wiki/Alpha-Beta\\_pruning](https://en.wikipedia.org/wiki/Alpha-Beta_pruning)

Task: Which leaf nodes are not evaluated using alpha-beta pruning when updating the leaf nodes from left to right (assuming that the displayed values are not known a priori)

Task: Are less nodes evaluated when the nodes from right to left?

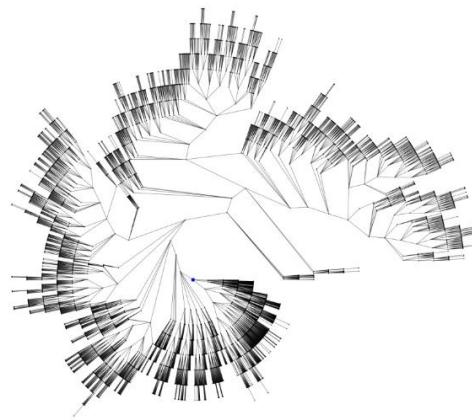
Task: Consider the node with the value 0.9. For which (hypothetical) values of this node are which nodes evaluated or not?



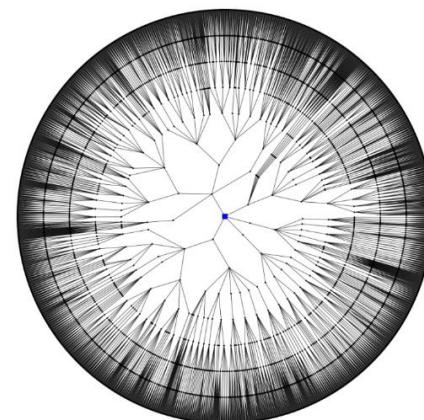
## MC search vs. MCTS vs. Minimax

- MC search and MCTS create a deeper, yet usually incomplete search tree compared to Minimax. This is disadvantageous if one has to rigorously evaluate all possible outcomes in order to find the one good solution among many bad ones (Chess), but advantageous if a game has a huge branching factor (Go, Scrabble)
- MCTS is usually superior to MC search unless there is a lot of randomness involved. Then MC search can be advantageous

MCTS

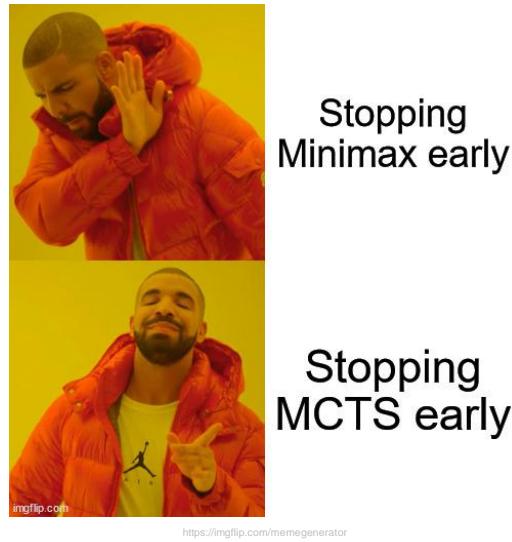


Minimax



<https://www.cs.cornell.edu/courses/cs6700/2016sp/lectures/CS6700-UCT.pdf>

- MC search and MCTS are less prone to imperfect function approximators than Minimax (average vs. min/max)
- MC search and MCTS are simpler to implement and easier to parallelize than Minimax. In addition, they can be stopped at any time during evaluation while providing a (somewhat) reasonable result
- All methods converge to the same optimal minimax value function using self-play



# Games

So far we've seen how a search tree can be used to improve the action selection of an agent, assuming that a trained state/action-value function predictor is given.

The next slides show how such a state/action-value function predictor can be trained more efficiently using a search tree. All games are assumed to be self-play.

## TD learning without search trees

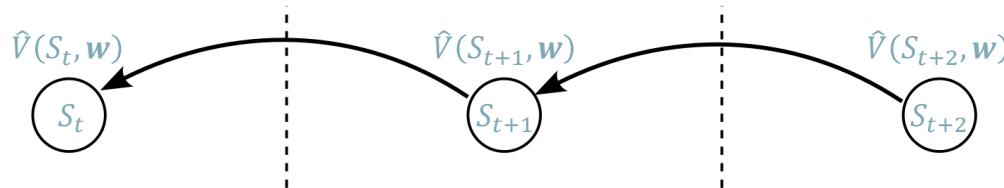
- Without a search tree, TD learning updates values towards successor values as

$$\hat{V}(S_t, w) \leftarrow \hat{V}(S_{t+1}, w)$$

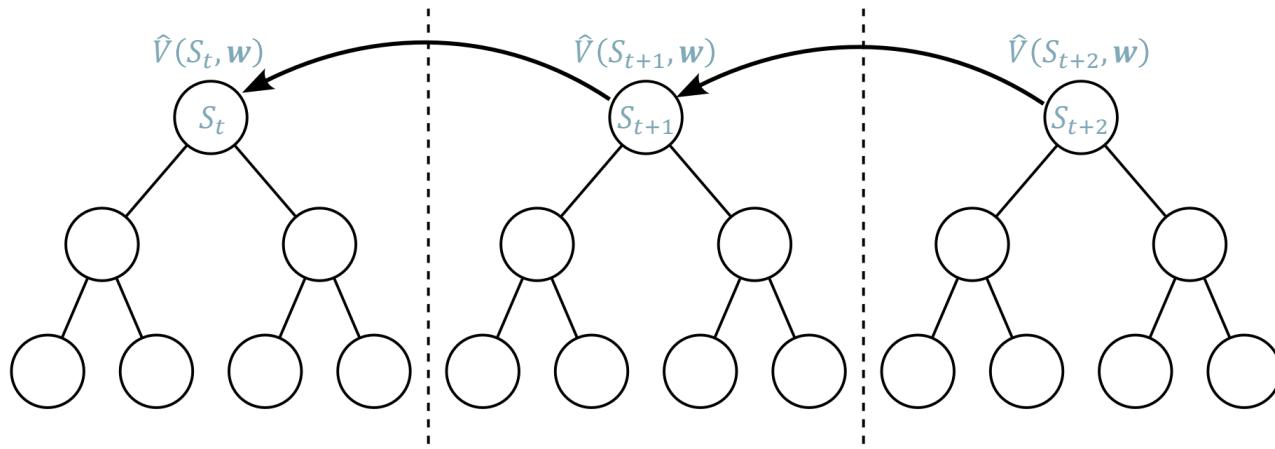
for all states except the last one, corresponding to the minimization of the loss

$$L = (R_t + \gamma \hat{V}(S_{t+1}, w) - \hat{V}(S_t, w))^2$$

with  $\gamma = 1$  (see slides on Deep Sarsa for details)



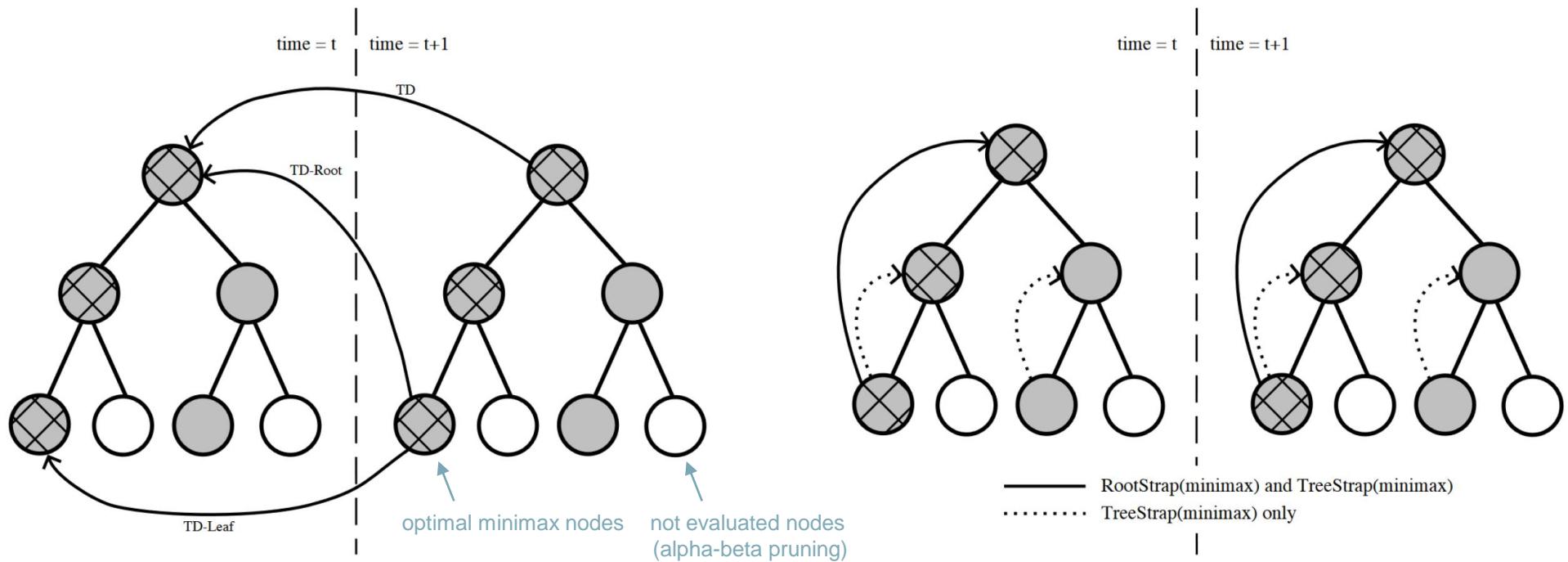
- When calculating a search tree at every step, the resulting TD update looks as follows



- Note that although a number of possible future states are calculated at every timestep, they are not used for the TD updates

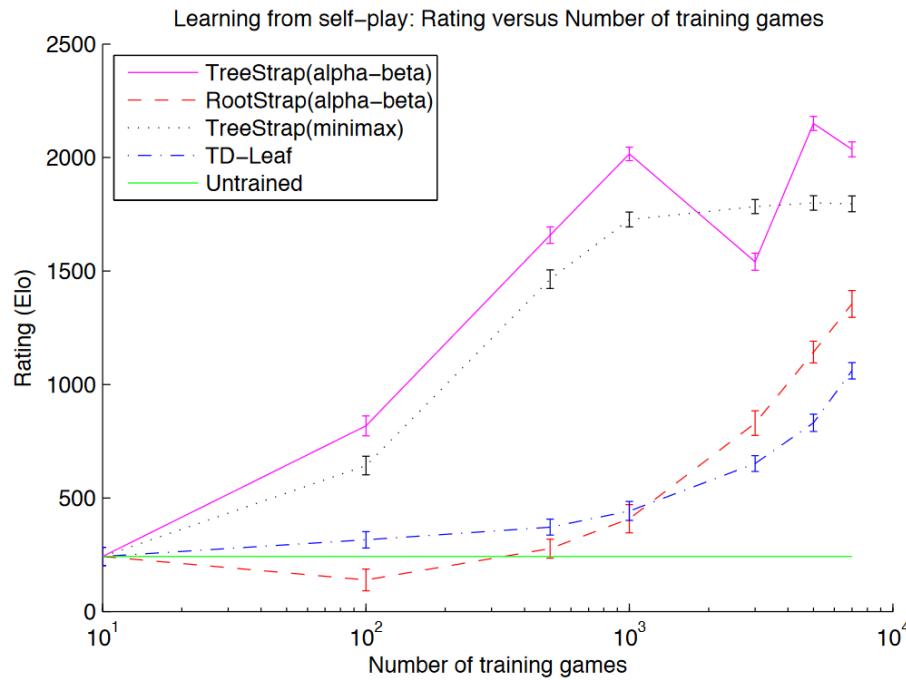
## TD learning with search trees

- There exist different ways of updating the different nodes using Minimax search
- Common algorithms include TD-Root, TD-Leaf, Treestrap and Rootstrap



# Games

- Result: The more updates, the faster the algorithm converges

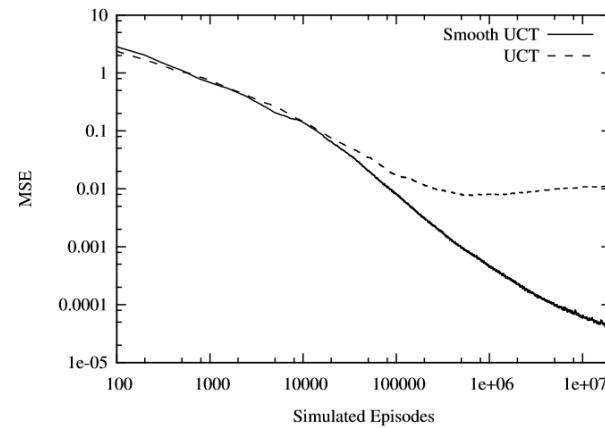


<https://www.davidsilver.uk/wp-content/uploads/2020/03/bootstrapping.pdf>

Task: What might be a (dis-)advantage of TD / TD-Root / TD-Leaf / Treestrap / Rootstrap?

## Self-play variations

- Self-play: Play against opponent's current behaviour
- Fictitious self-play: Play against opponent's average behaviour, i.e. a mixture of different policies
- One variant of fictitious self-play is smooth UCT:
  - With probability  $\eta$  select action according to UCT
  - With probability  $1 - \eta$  select action according to how often the action has been taken before
  - Well suited for imperfect information games
- Result: Better convergence for Poker when implementing smooth UCT



<https://www.ijcai.org/Proceedings/15/Papers/084.pdf>

## AlphaZero (2017)

- successor of AlphaGo, able to play Go, Shogi and Chess with superhuman performance
- combination of self-play, MCTS with PUCT and a policy network for  $p$ , outputting a probability for every theoretically possible move in every state (also invalid ones, there are e.g. 4672 possible moves in Chess)

Timeline Overview

1. AlphaGo → Monte Carlo Tree Search using 3 Convolutional Policy Networks, 2 of which are trained to copy expert moves, another with policy gradients, and a separate Value Network.
2. AlphaGo Zero → No Supervised Learning of expert moves, Policy and Value networks combined into a single Residual Neural Network, Policy Net updated to match MCTS actions
3. AlphaZero → Restructure input and output representations to play Chess and Shogi as well, restructure self-play algorithm
4. MuZero → Remove assumption of a given dynamics model, introduce hidden state in order to do MCTS with a learned dynamics model, starting from a learned root state initialization



HENRY  
AI LABS

<https://www.youtube.com/watch?v=A0HX8BgckFI>

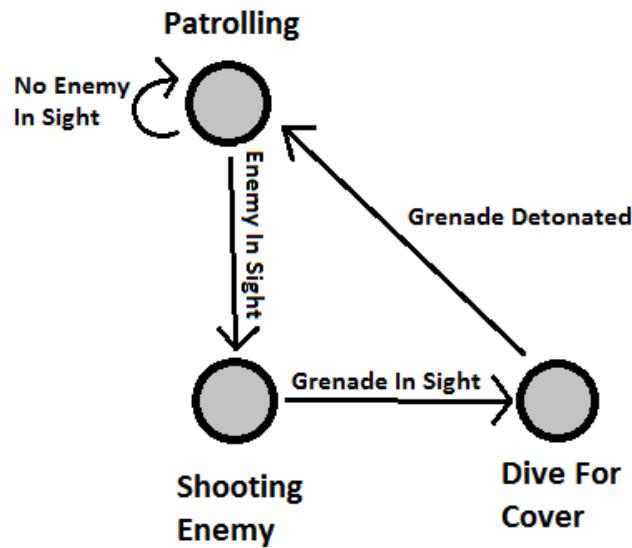
# Kahoot!

## Other planning techniques for generic computer games

- Finite state machines
- Behaviour trees
- Goal oriented planning
- Hierarchical task network
- Utility theory

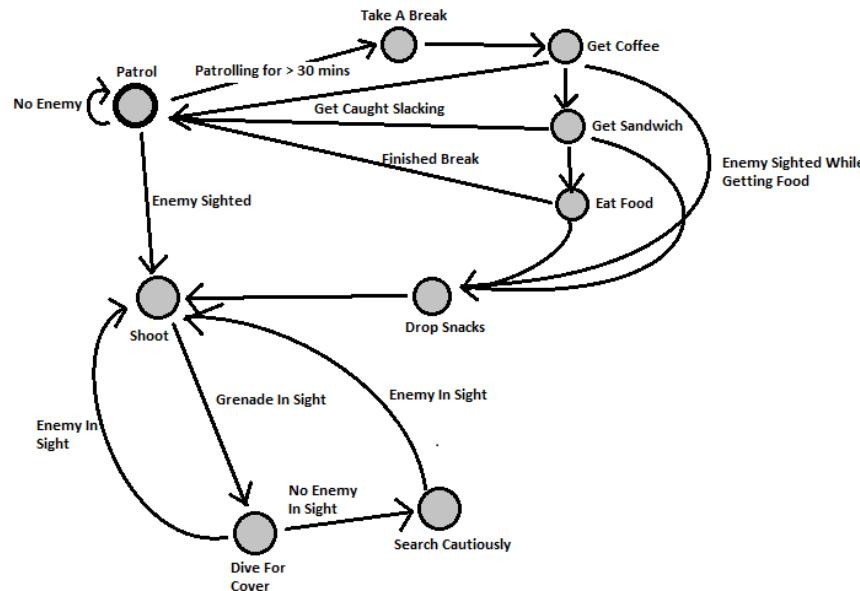
## Finite state machine (FSM)

- AI agent is modeled with
  - states (behaviour of the AI agent)
  - transitions (events that cause the AI agent to change its behaviour)



[https://martindevans.me/heist-game/2013/04/16/Finite-State-Machines-\(Are-Boring\)/](https://martindevans.me/heist-game/2013/04/16/Finite-State-Machines-(Are-Boring)/)

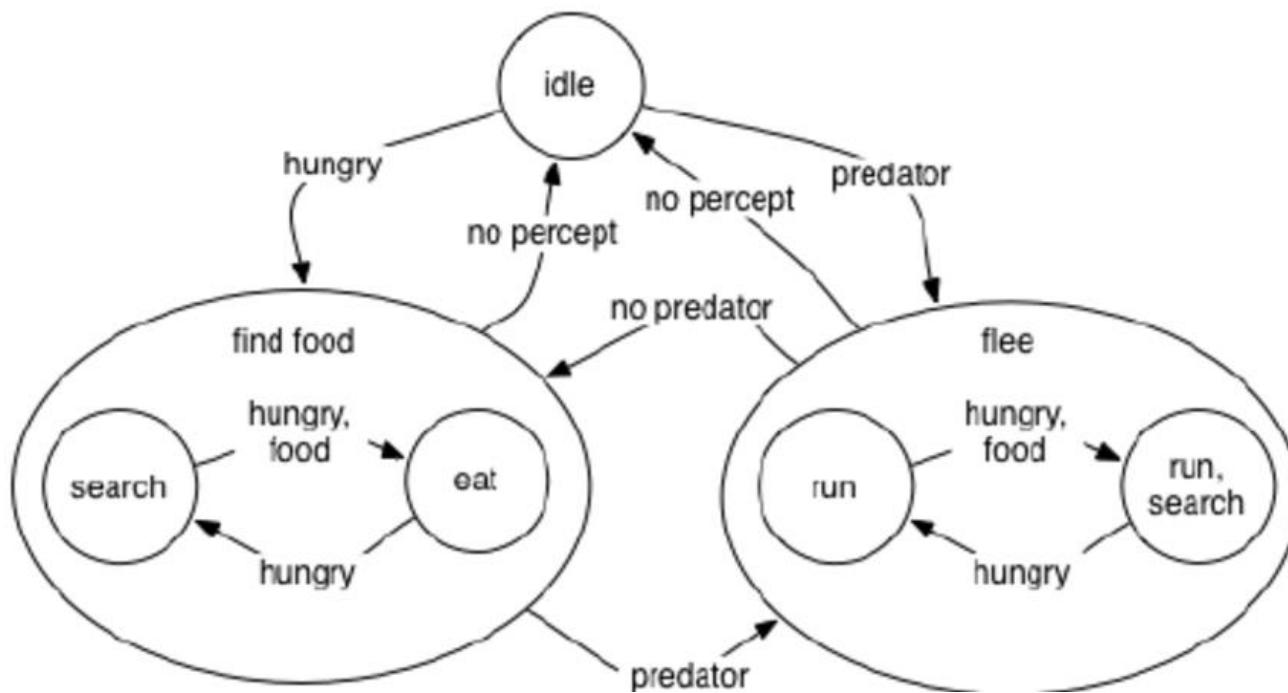
- Advantage
  - easy to model for few states
- Disadvantage:
  - With large FSM there are  $O(n^2)$  possible transitions between  $n$  states → difficult to model all
  - if not all transitions are modeled, the AI behaviour may appear inconsistent



[https://martindevans.me/heist-game/2013/04/16/Finite-State-Machines-\(Are-Boring\)/](https://martindevans.me/heist-game/2013/04/16/Finite-State-Machines-(Are-Boring)/)

# Games

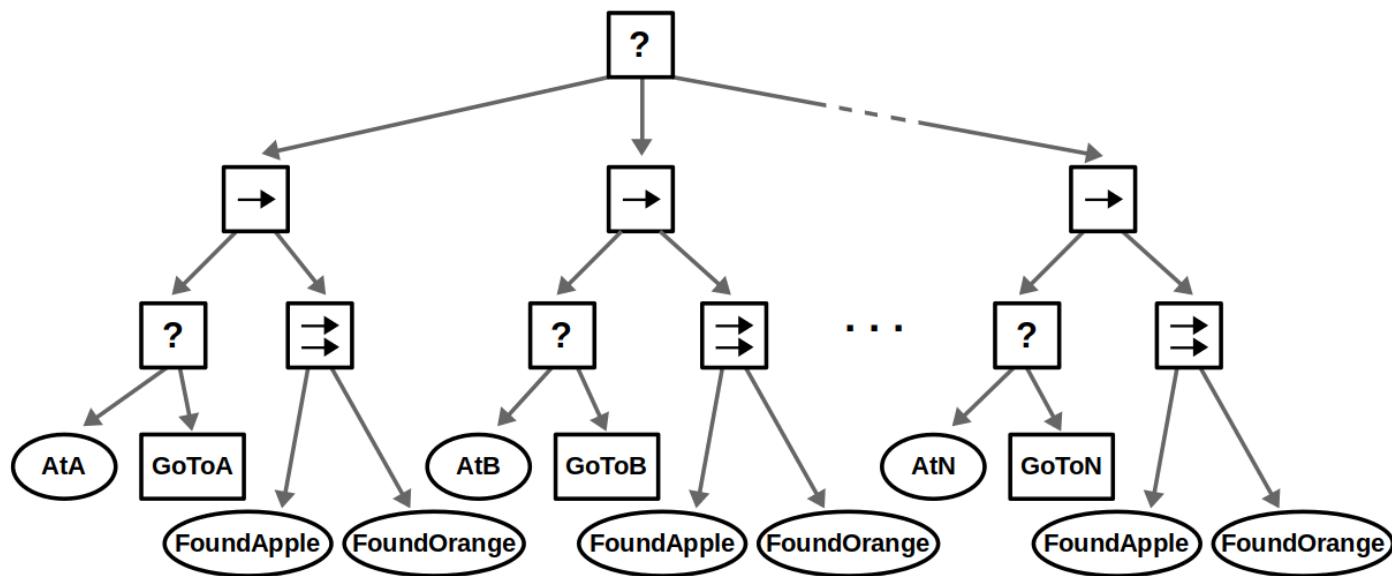
- Partial solution: Hierarchical finite state machines (HFSM)
  - Agglomerate multiple states in a "super state" to reduce the number of transitions



[https://www.researchgate.net/figure/Hierarchical-finite-state-machine-representation-of-the-scavenger-agent-Self-transitions\\_fig2\\_221157398](https://www.researchgate.net/figure/Hierarchical-finite-state-machine-representation-of-the-scavenger-agent-Self-transitions_fig2_221157398)

## Behaviour trees (BT)

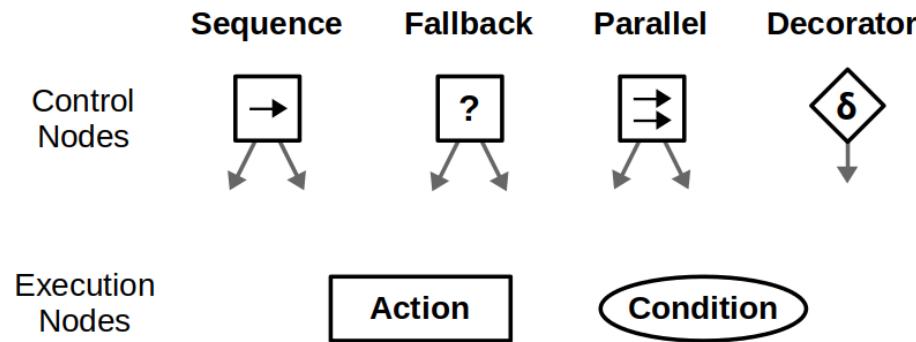
- are a simpler visualization than FSMs for complex robot behaviour
- starting from the root node, go towards the leaf nodes according to a set of rules at every tick (update step) and return the status to the parent node



<https://robohub.org/introduction-to-behavior-trees/>

# Games

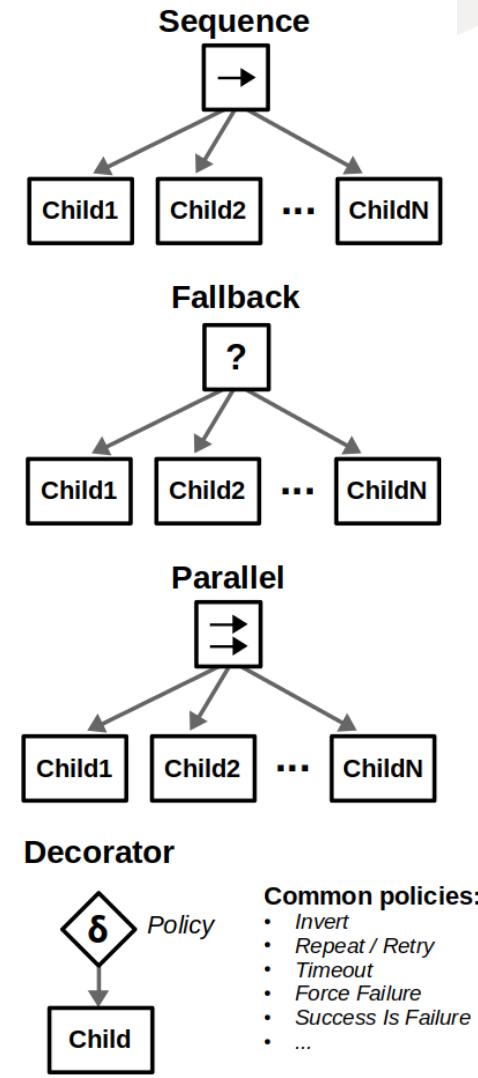
- Types of nodes for behaviour trees
  - control nodes are internal nodes and define how to traverse the BT given the status of their children
  - execution nodes are leaves of the BT can either be action nodes (which return Success/Failure/Running) or condition nodes (which return Success/Failure)



<https://robohub.org/introduction-to-behavior-trees/>

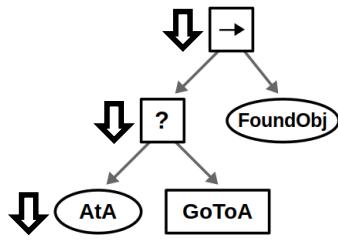
# Games

- Types of control nodes
  - Sequence nodes (logical and) execute children in order until one returns Failure or all return Success  
→ IF Success THEN Tick next ELSE return "same as child"
  - Fallback nodes (logical or) execute children in order until one returns Failure or all return Success  
→ IF Failure THEN Tick next ELSE return "same as child"
  - Parallel nodes tick each child node in parallel and return Success when at least M child nodes have succeeded and return Failure when all child nodes have failed
  - Decorator nodes modify a single child node with a custom policy (e.g. invert the output of Success/Failure)

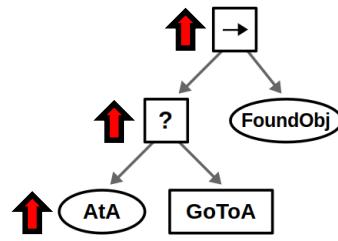


# Games

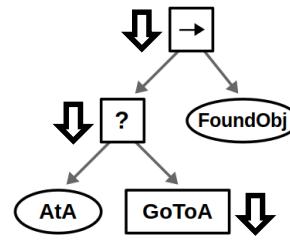
- Example in detail



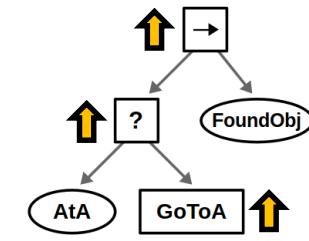
first tick the  
leftmost leaf node



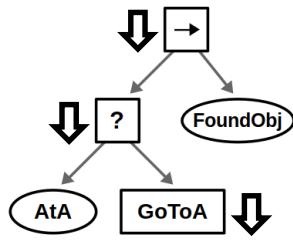
propagate the response  
back to the root node  
(failure)



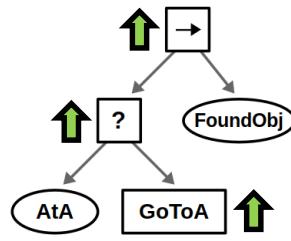
then tick the  
next leaf node



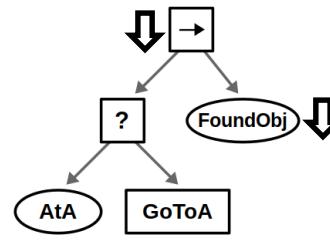
propagate the response  
back to the root node  
(running)



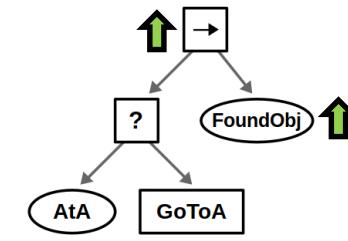
then tick the  
leaf node again



propagate the response  
back to the root node  
(success)



then tick the  
next leaf node

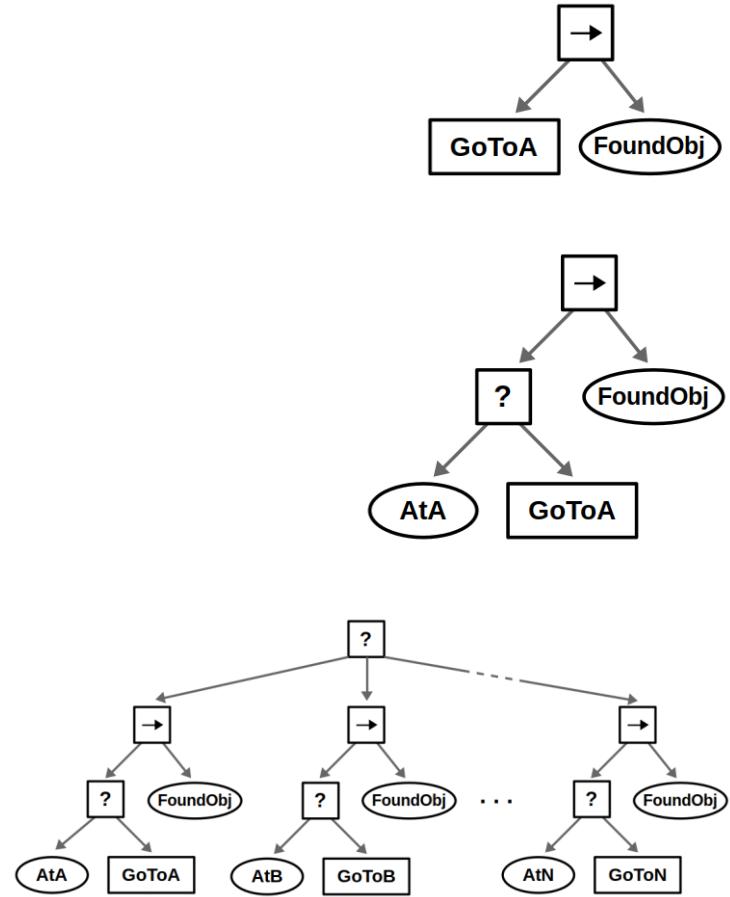


propagate the response  
back to the root node  
(success)

# Games

- Example: Robot has to find object in different locations

- First go to location A, then check if the object has been found there
- First check if the robot is already at location A, otherwise go to location A. Then check if the object has been found there
- Do the same as before for three different locations until the object is found at one of the locations



<https://robohub.org/introduction-to-behavior-trees/>

# Games

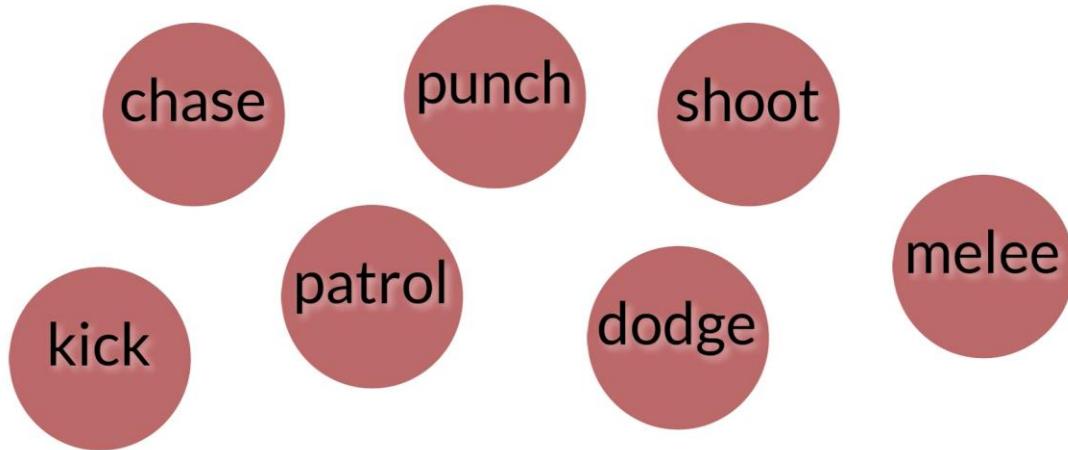
- Advantage
  - cleaner than FSMs
- Disadvantage
  - different way of thinking about agent behaviour (trying to get away from states)
  - limited range of behaviours
  - complex behaviours are difficult to model



[http://download.ros.org/downloads/rocon/2022/BehaviorTree\\_CPP%204.0.%20What%20is%20new%20and%20roadmap.pdf](http://download.ros.org/downloads/rocon/2022/BehaviorTree_CPP%204.0.%20What%20is%20new%20and%20roadmap.pdf)

## Goal oriented planning (GOAP)

- Idea of GOAP: Instead of planning complex tasks by hand by creating manual transitions, let an algorithm (e.g. Dijkstra or A\*) do the planning
- Note that a similar technique named STRIPS (Stanford Research Institute Problem Solver) dates back to 1971



<https://learn.unity.com/tutorial/an-introduction-to-goap>

## Method overview

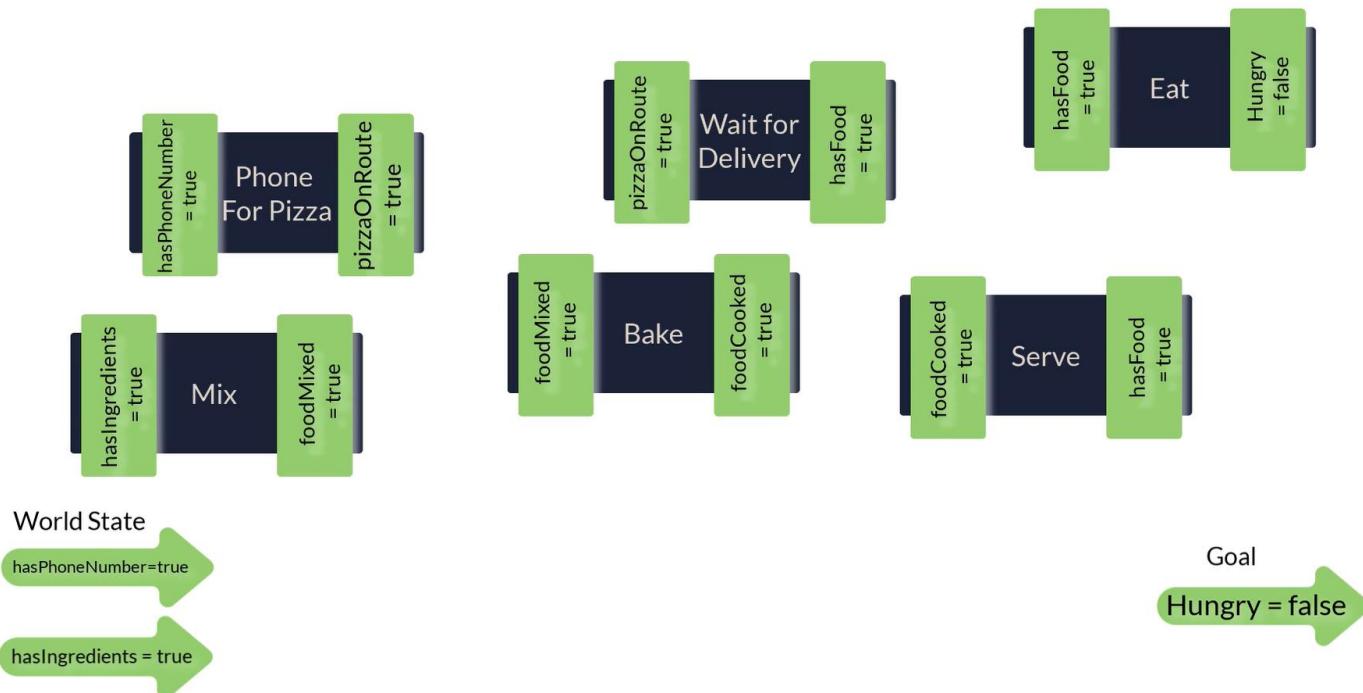
- Every possible action of the agent contains
  - a precondition that must be fulfilled for the action to be executable
  - the action itself
  - the effect of the action, i.e. what changes after the action



<https://learn.unity.com/tutorial/an-introduction-to-goap>

# Games

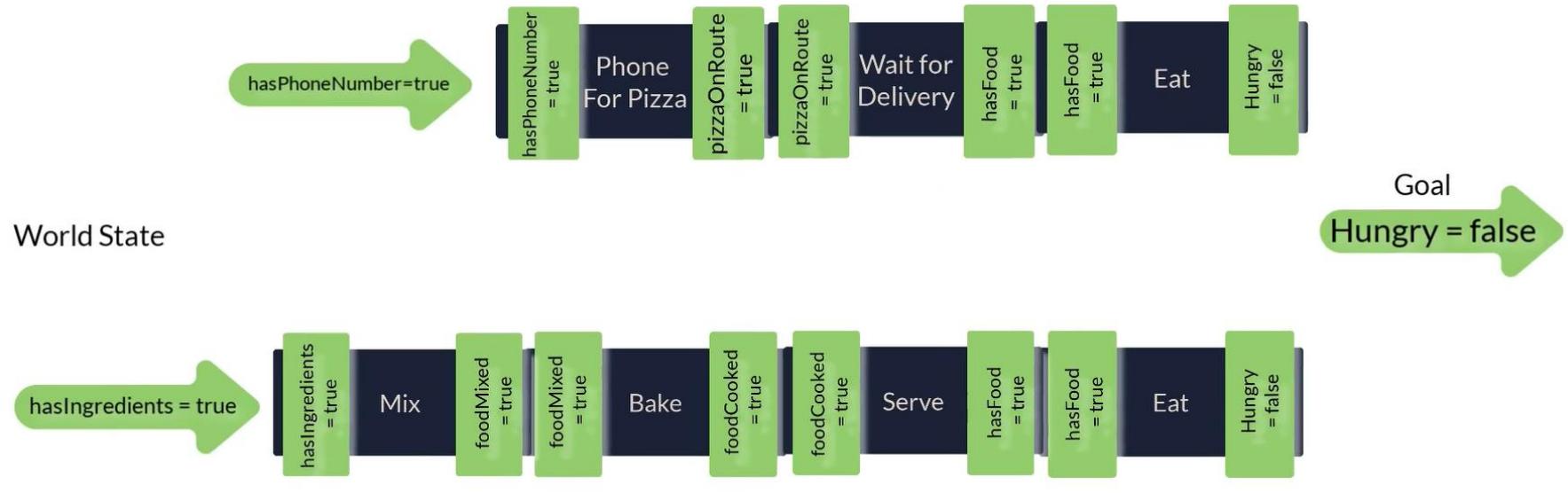
- A world state encodes the current state of the agent
- A goal state encodes the desired state of the agent



<https://learn.unity.com/tutorial/an-introduction-to-goap>

# Games

- Starting from the world state, a tree-like structure (with possible loops) of all possible action sequences is created until some sequences end in the goal state



# Games

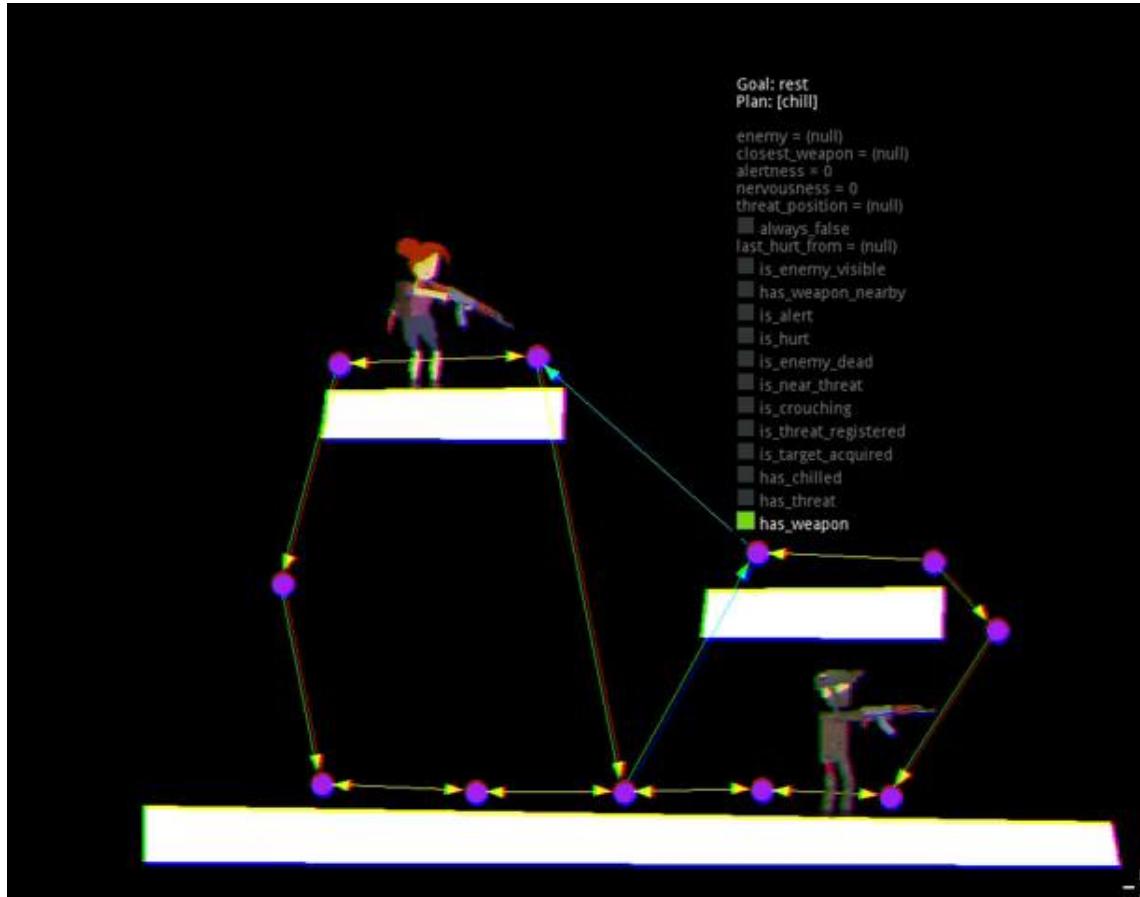
- Dijkstra's algorithm or A\* algorithm is then used to find an action sequence with minimal cost



<https://learn.unity.com/tutorial/an-introduction-to-goap>

# Games

- Example (besides F.E.A.R.)



# Games

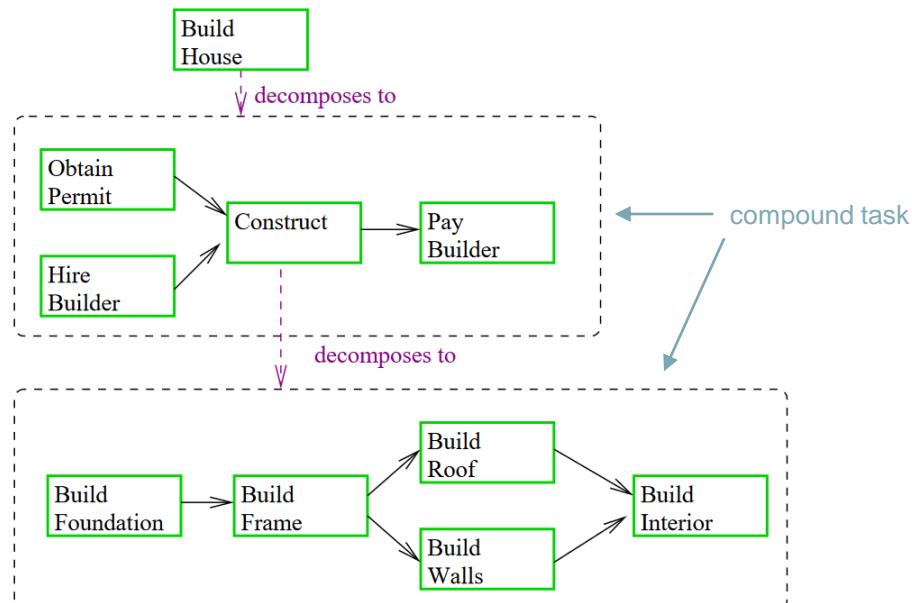
- Advantage
  - Complex behaviour can be modeled efficiently
- Disadvantage
  - Difficult to implement
  - Heavy CPU load when planning for multiple agents
  - Multiple agents cannot be coordinated easily using GOAP



<https://www.developgoodhabits.com/goals-memes/>

## Hierarchical task networks (HTN)

- are an improvement over GOAP
- Idea: Decompose complex task into compound tasks consisting of primitive tasks and optimize compound/primitive tasks on different levels



<https://pages.mtu.edu/~nilufer/classes/cs5811/2012-fall/lecture-slides/cs5811-ch11b-htn.pdf>

## Advantage

- can incorporate domain knowledge (partial/complete ordering of primitive actions) for compound tasks
- often more user-friendly by defining preconditions/effects on compound task level instead of primitive task level
- faster than GOAP for complex tasks

## Disadvantage

- domain knowledge may lead to inconsistent/impossible task planning

## Utility theory

- Idea: Associate a value with every action (utility/cost function) measuring how good/bad the action is for the given state. Then execute the action with the highest value
- Can also incorporate uncertainty/randomness/...



<https://mcguirev10.com/2019/01/03/ai-decision-making-with-utility-scores-part-1.html>

# Model-based RL

# Model-based RL

Problem with "standard" RL: Many iterations till convergence, may be costly / slow

Example: Drone learns to avoid obstacles by first crashing 11500 time into them



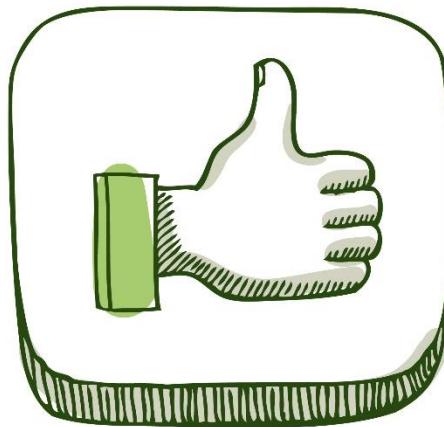
Solution: Use a model (simulation) of the environment to accelerate training

## Advantages of model-based RL

- learning in simulation (based on the model) is way faster compared to real life
- model uncertainty can be estimated

## Disadvantages of model-based RL

- additional source of error



<https://leakpack.com/advantages-disadvantages-mechanical-seals/>

## Model learning

- The environment model consists of two functions (cond. independence assumed)

$$R_{t+1} = f(S_t, A_t)$$

$$S_{t+1} = g(S_t, A_t)$$

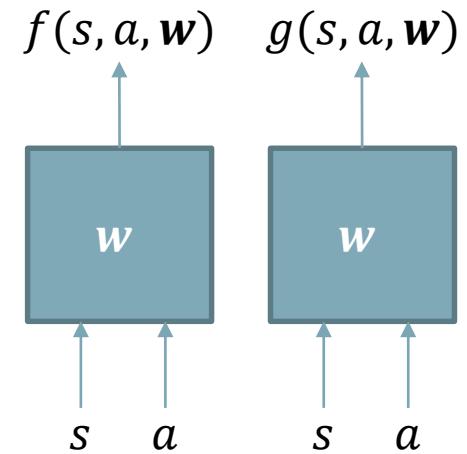
- $f(s, a)$  outputs the reward for a given state/action pair
- $g(s, a)$  outputs the next state for a given state/action pair

- Both functions can be learned from experience  $\{S_1, A_1, R_2, \dots, S_T\}$  through supervised learning, e.g.

$$S_1, A_1 \rightarrow R_2, S_2$$

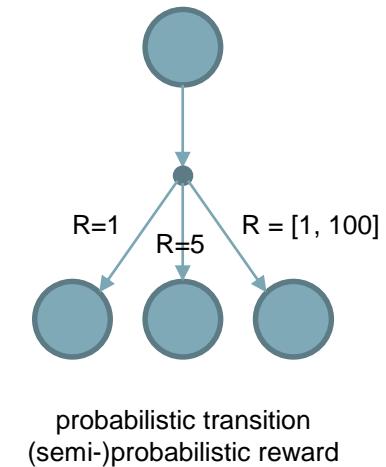
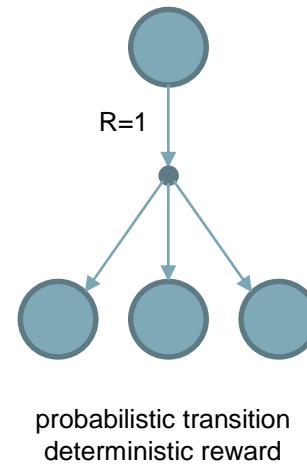
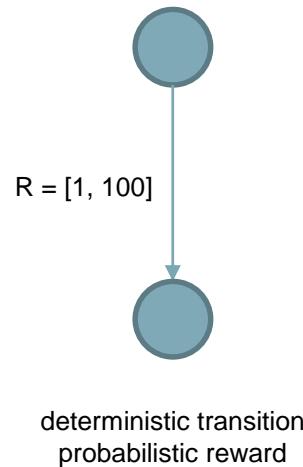
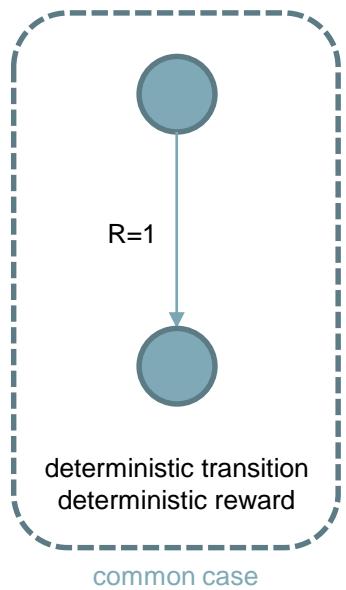
$$S_2, A_2 \rightarrow R_3, S_3$$

...



# Model-based RL

- For a deterministic state transition/reward model, the corresponding outputs of  $f(S_t, A_t)$  and  $g(S_t, A_t)$  are deterministic  
→ regression problem
- For a stochastic state transition/reward model, the corresponding outputs of  $f(S_t, A_t)$  and  $g(S_t, A_t)$  are stochastic (probability distribution)  
→ density estimation problem



# Model-based RL

Task: How does the model look like if there is no conditional independence between  $R_{t+1}$  and  $S_{t+1}$  assumed?

Examples for environment models

- linear models (e.g. linear regression)
- nonlinear models (e.g. neural networks)
- probabilistic models (e.g. table lookup models, kernel density estimators, probabilistic NN)

Task: How does the environment model look like if no conditional independence between state transitions and rewards is assumed?

Task: How many output dimensions does a deterministic/probabilistic state transition/reward model have?

## Table lookup model

- Idea: For simple models with few states / rewards, the probability of each outcome can be derived from counting as

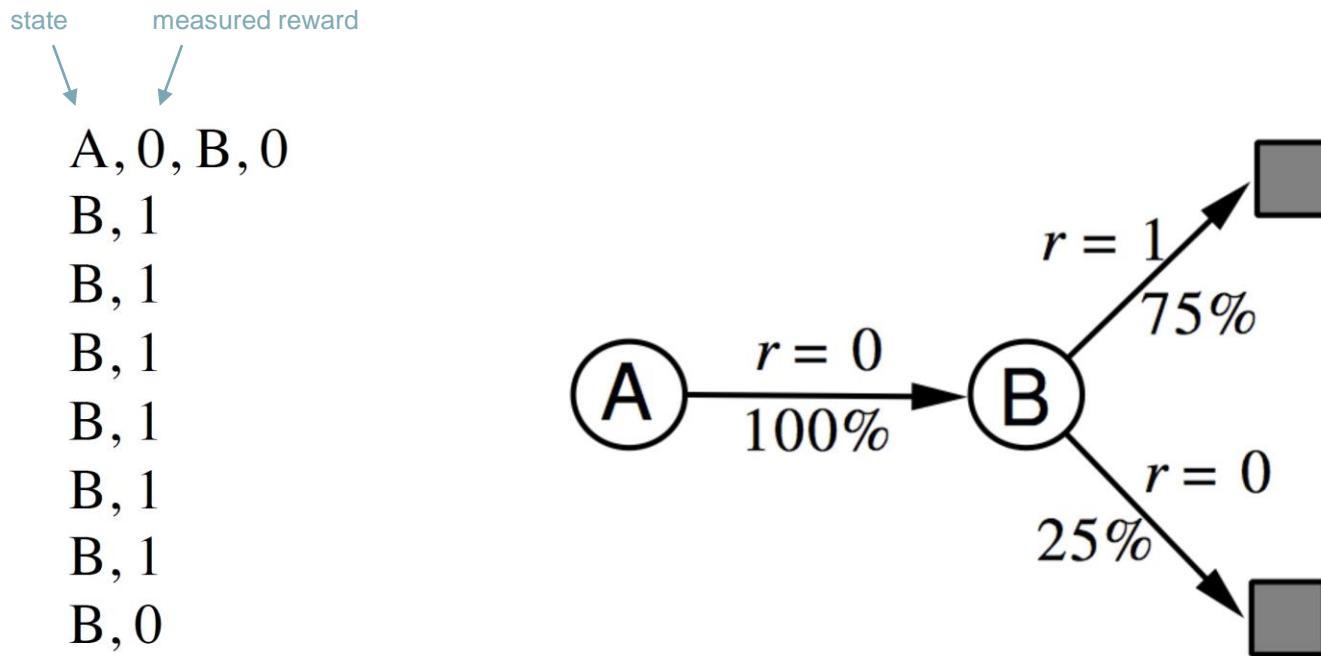
$$f(S_t, A_t) = \frac{1}{N(s,a)} \sum_t \mathbf{1}(S_t = s, A_t = a, S_{t+1} = s')$$

$$g(S_t, A_t) = \frac{1}{N(s,a)} \sum_t \mathbf{1}(S_t = s, A_t = a) R_t$$

- Interpretation: If for a given, specific state/action pair the measured reward was e.g. 5 in 20% of all cases and 3 in 80% of all cases, then  $f(S_t, A_t)$  has the same probabilities as output

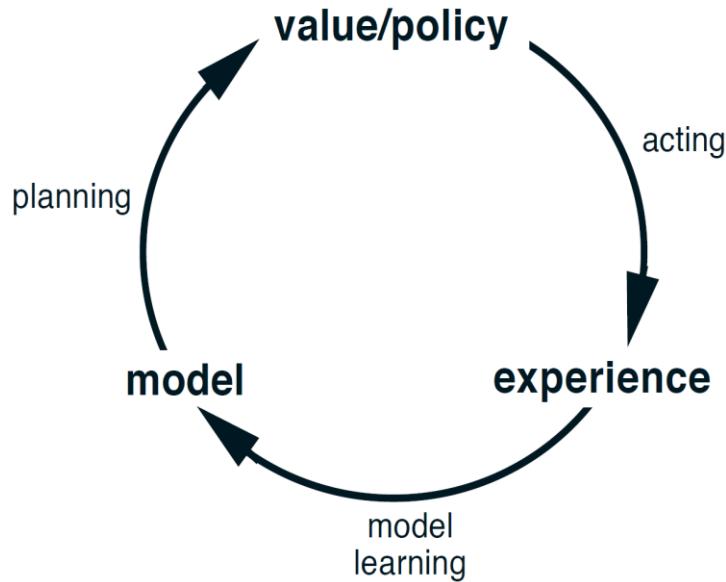
# Model-based RL

Example: Only one action



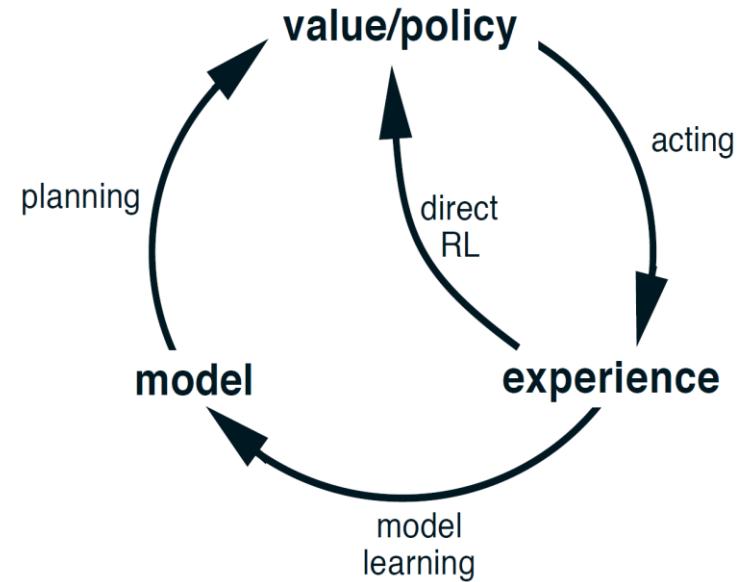
Task: How many output dimensions does the table lookup model have? What is the value of each output?

## Model learning techniques



Sutton, Barto: Reinforcement Learning

- learn model from experience
- learn value function / policy  
ONLY from model evaluations



Sutton, Barto: Reinforcement Learning

- learn model from experience
- learn value function / policy  
from model evaluations  
AND experience

# Model-based RL

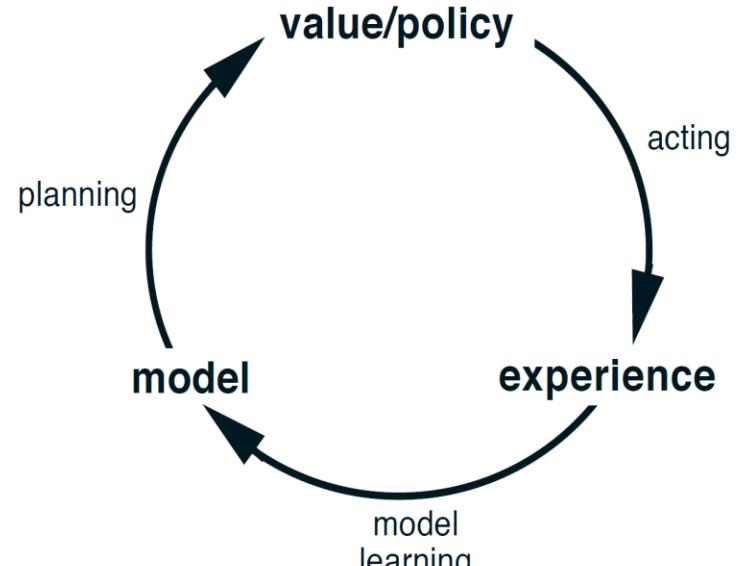
Learning only from model output

Advantage

- very fast (can do multiple planning steps per acting step)

Disadvantage

- even small modelling errors can be crucial (e.g. for dynamic systems)



Sutton, Barto: Reinforcement Learning



<https://bair.berkeley.edu/blog/2019/12/12/mopo/>

## Example: World models

- first learn model from experience, then learn value/policy function only from model

during learning



after learning



<https://worldmodels.github.io/>

# Model-based RL

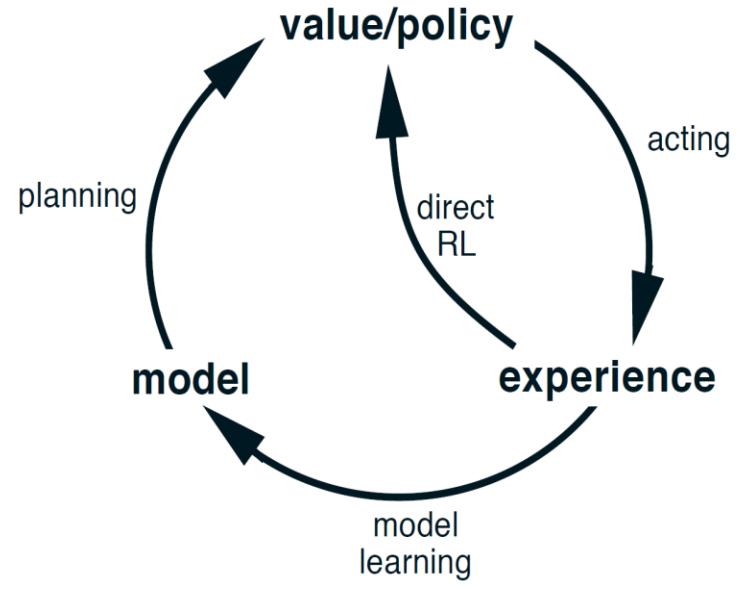
Learning from model output and experience

## Advantage

- value/policy function is learned from experience and model (more robust)

## Disadvantage

- continuous real life evaluations needed during learning



Sutton, Barto: Reinforcement Learning

## Dyna-Q

- learns from model output and experience
- does  $n$  update steps using the model (planning steps) for every update step using the real environment (real steps)

### Tabular Dyna-Q

Initialize  $Q(s, a)$  and  $Model(s, a)$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$

Loop forever:

- $S \leftarrow$  current (nonterminal) state
- $A \leftarrow \epsilon\text{-greedy}(S, Q)$
- Take action  $A$ ; observe resultant reward,  $R$ , and state,  $S'$
- $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
- $Model(S, A) \leftarrow R, S'$  (assuming deterministic environment)
- Loop repeat  $n$  times:

$S \leftarrow$  random previously observed state

$A \leftarrow$  random action previously taken in  $S$

$R, S' \leftarrow Model(S, A)$

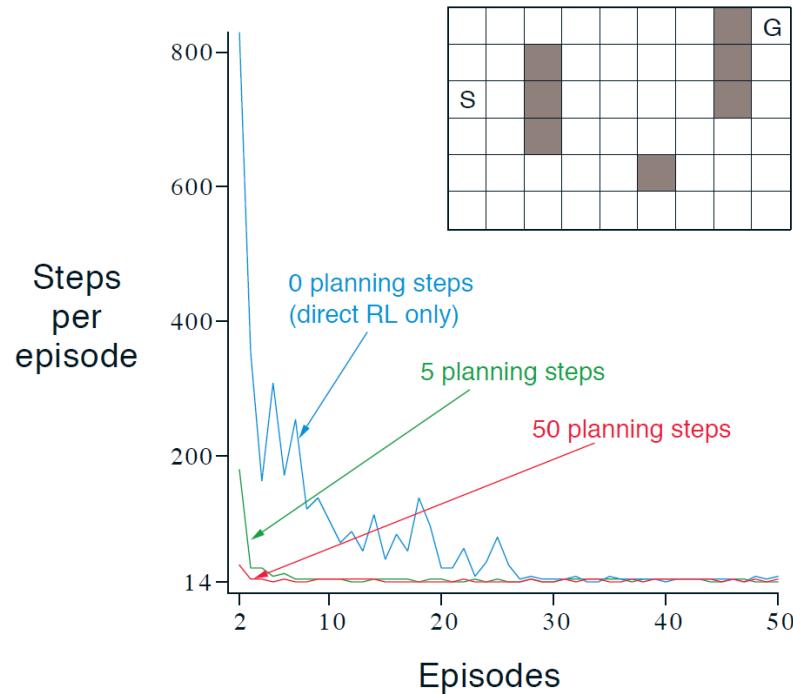
$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

Sutton, Barto: Reinforcement Learning

# Model-based RL

## Example: Maze

- faster convergence with more planning steps per real step

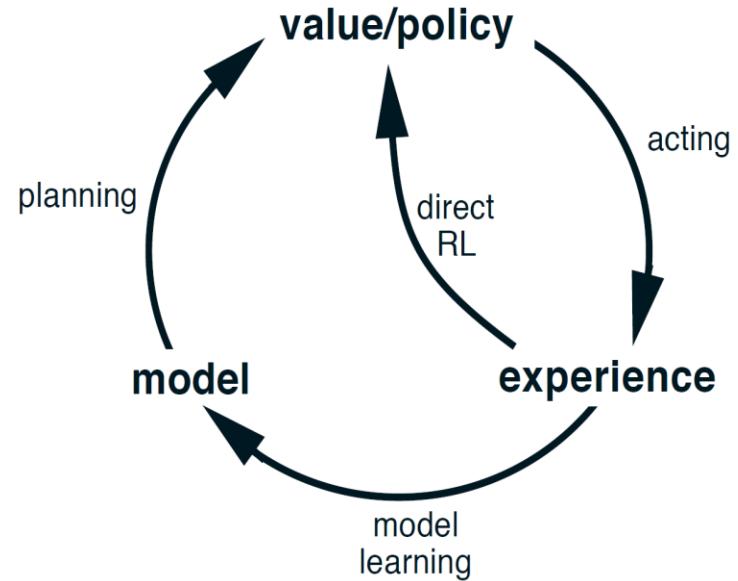


Sutton, Barto: Reinforcement Learning

# Model-based RL

Task: How does RL without a model look like in the diagram shown on the right?

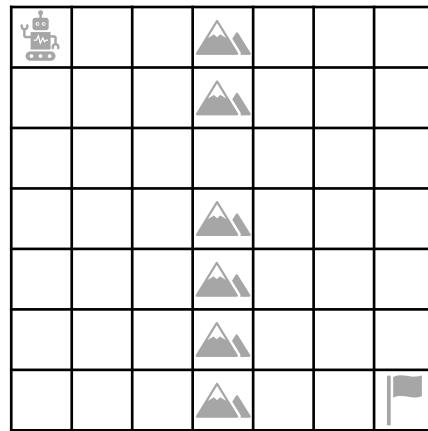
Task: Does it make sense to perform multiple real steps for each planning step?



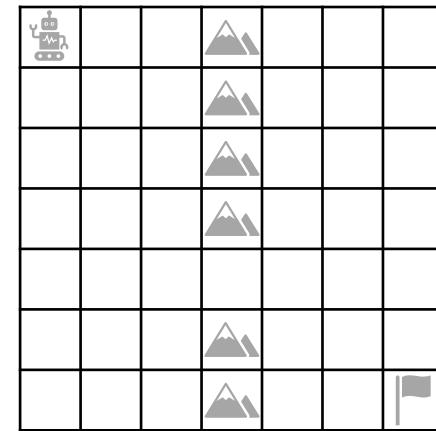
Sutton, Barto: Reinforcement Learning

# Meta-RL

First problem with "standard" RL: Bad generalization to new environments with pretrained agents



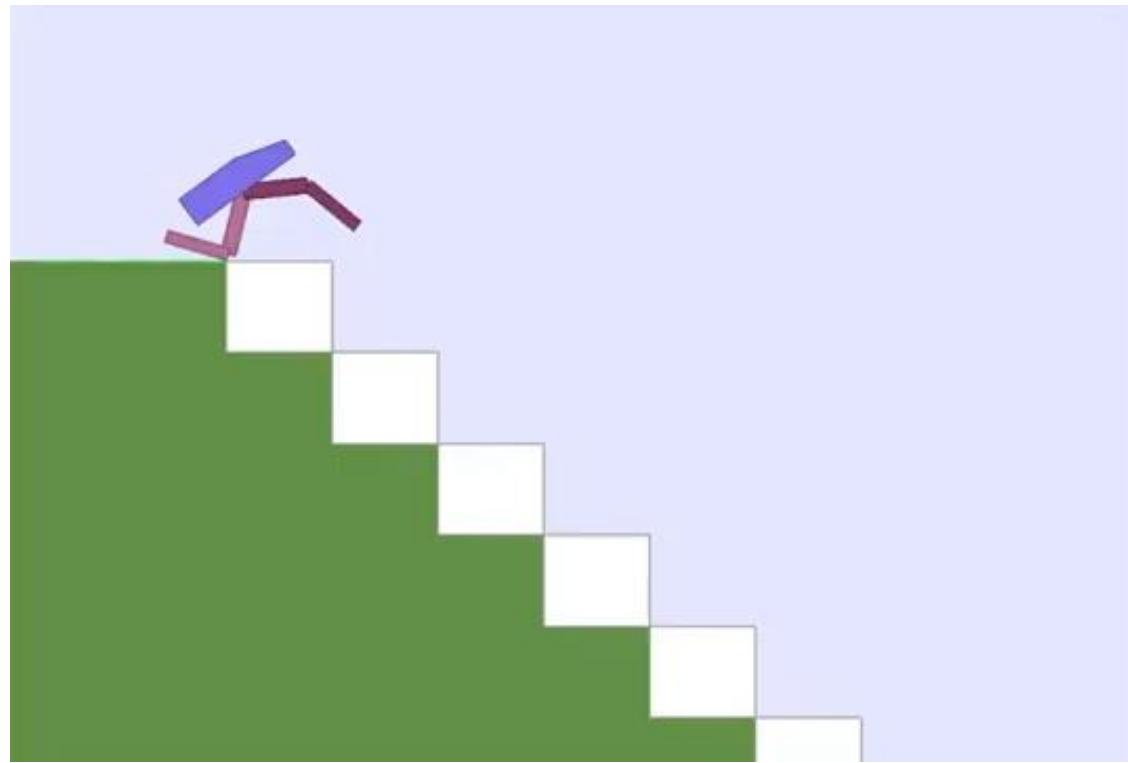
First agent on original environment



Then train pretrained agent on slightly modified environment

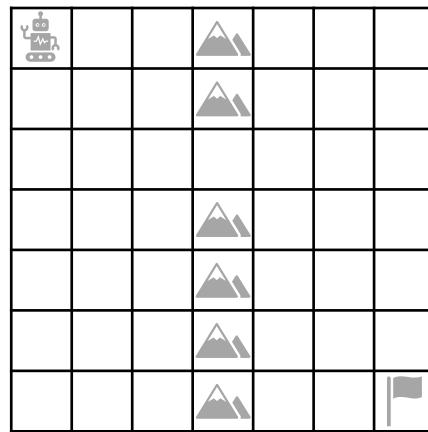
→ training will probably take way longer than training it again from scratch

Second problem with "standard" RL: Some environments are too difficult to train an agent within a reasonable amount of time

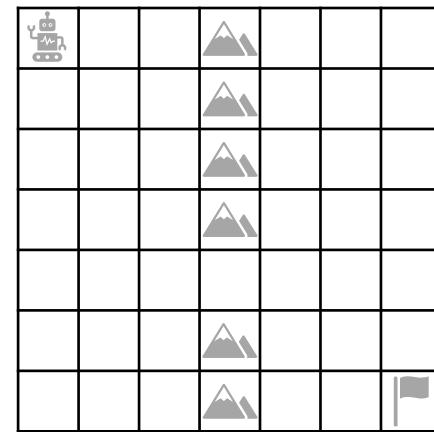


<https://www.youtube.com/watch?v=D1WWhQY9N4g>

Idea of Meta RL: Train an agent such that it can generalize quickly to new environments, i.e. learn a new environment with way less training steps compared to when trained from scratch



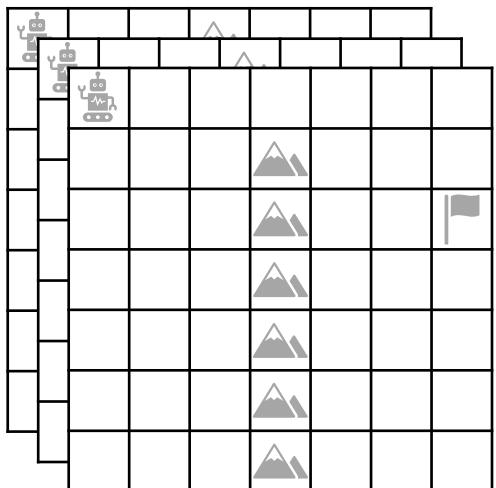
First agent on original environment



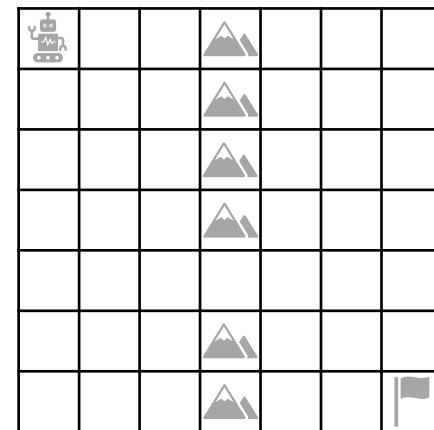
Then train pretrained agent on slightly modified environment

→ training should be much faster than training it again from scratch

General idea of Meta RL: Train agent on many tasks and let it eventually grasp the key concepts of that task in order to generalize to new tasks



First agent on multiple support tasks  $T_i^S$



Such that it can be trained quickly on  
the query tasks  $T_i^Q$

## MAML (Model-Agnostic Meta Learning)

- uses a modified loss function to incorporate meta RL
- relies on second derivatives (costly)
- can also be used for deep learning
- different variations exist (first order MAML, FOMAML)

Derivation:

- standard cost function for a given parameterization  $\boldsymbol{\theta}$  for a single task  $T^S$   
 $L(\boldsymbol{\theta}, T^S)$
- cost function over multiple tasks  $T_i^S$

$$\sum_i L(\boldsymbol{\theta}, T_i^S)$$

- MAML cost function

$$\sum_i L\left(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \sum_k L(\boldsymbol{\theta}, T_k^S), T_i^Q\right)$$

- Interpretation:

- $\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \sum_k L(\boldsymbol{\theta}, T_k^S)$  are the (intermediate) updated agent parameters after one gradient descent step on the support tasks
- $\sum_i L(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \sum_k L(\boldsymbol{\theta}, T_k^S), T_i^Q)$  is the loss with the updated agent parameters on the query tasks
- By calculating the gradient of  $\sum_i L(\boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \sum_k L(\boldsymbol{\theta}, T_k^S), T_i^Q)$  during gradient descent, one effectively calculates the gradient of a gradient → second derivatives

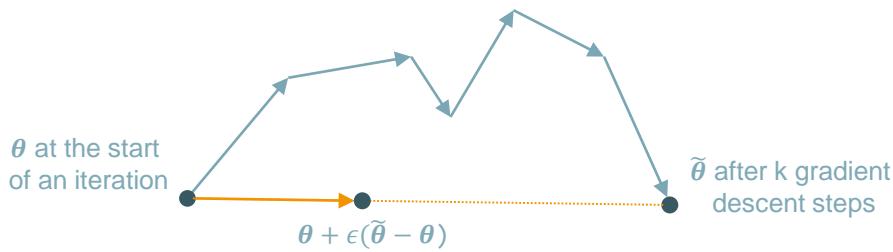
## Reptile

- is a simplification of MAML without the need to calculate second derivatives
- achieves similar performance as MAML

Algorithm:

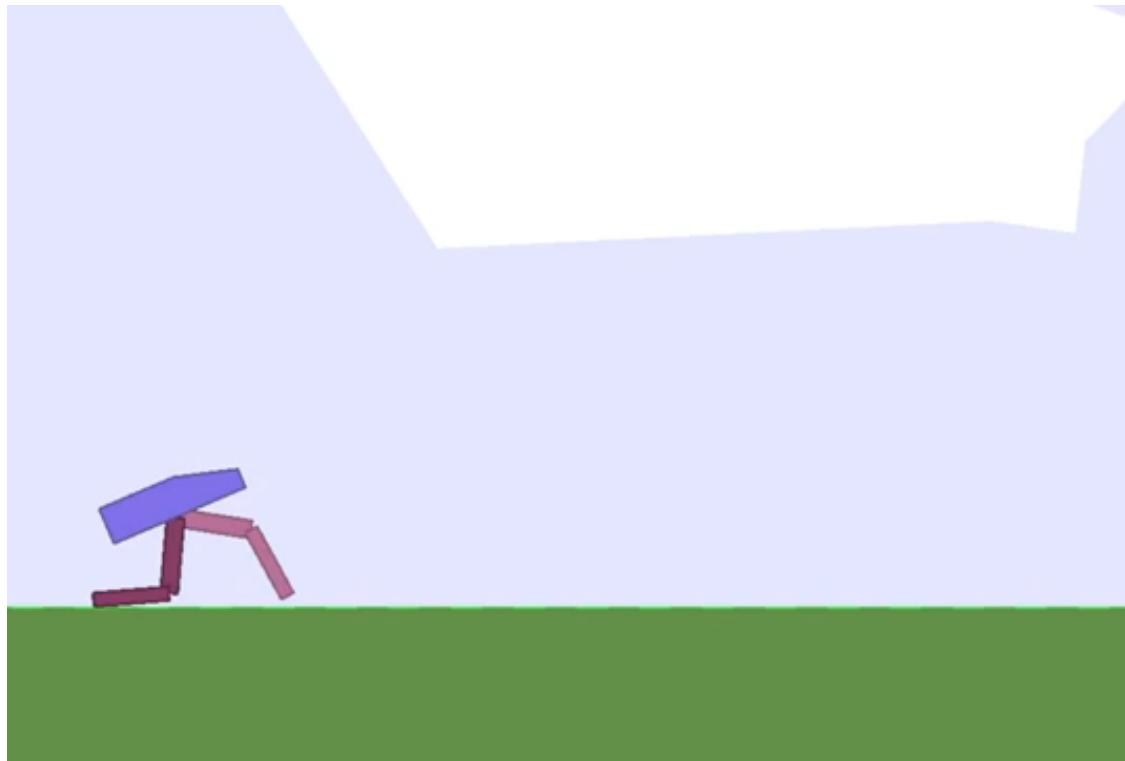
```
for iteration = 1,2,...
```

- select random support task  $T_i^S$
- calculate (intermediate) updated agent parameters  $\tilde{\theta}$  after k gradient descent steps
- update agent parameters as  $\theta \leftarrow \theta + \epsilon(\tilde{\theta} - \theta)$  with  $\epsilon \in [0,1[$



## POET (Paired Open-Ended Trailblazer)

- variant of curriculum learning: Iteratively learn tasks with increasing complexity
- Algorithm actively selects more challenging environments



<https://www.youtube.com/watch?v=D1WWhQY9N4g>

# Kahoot!