



Intel®  
Software  
Innovator

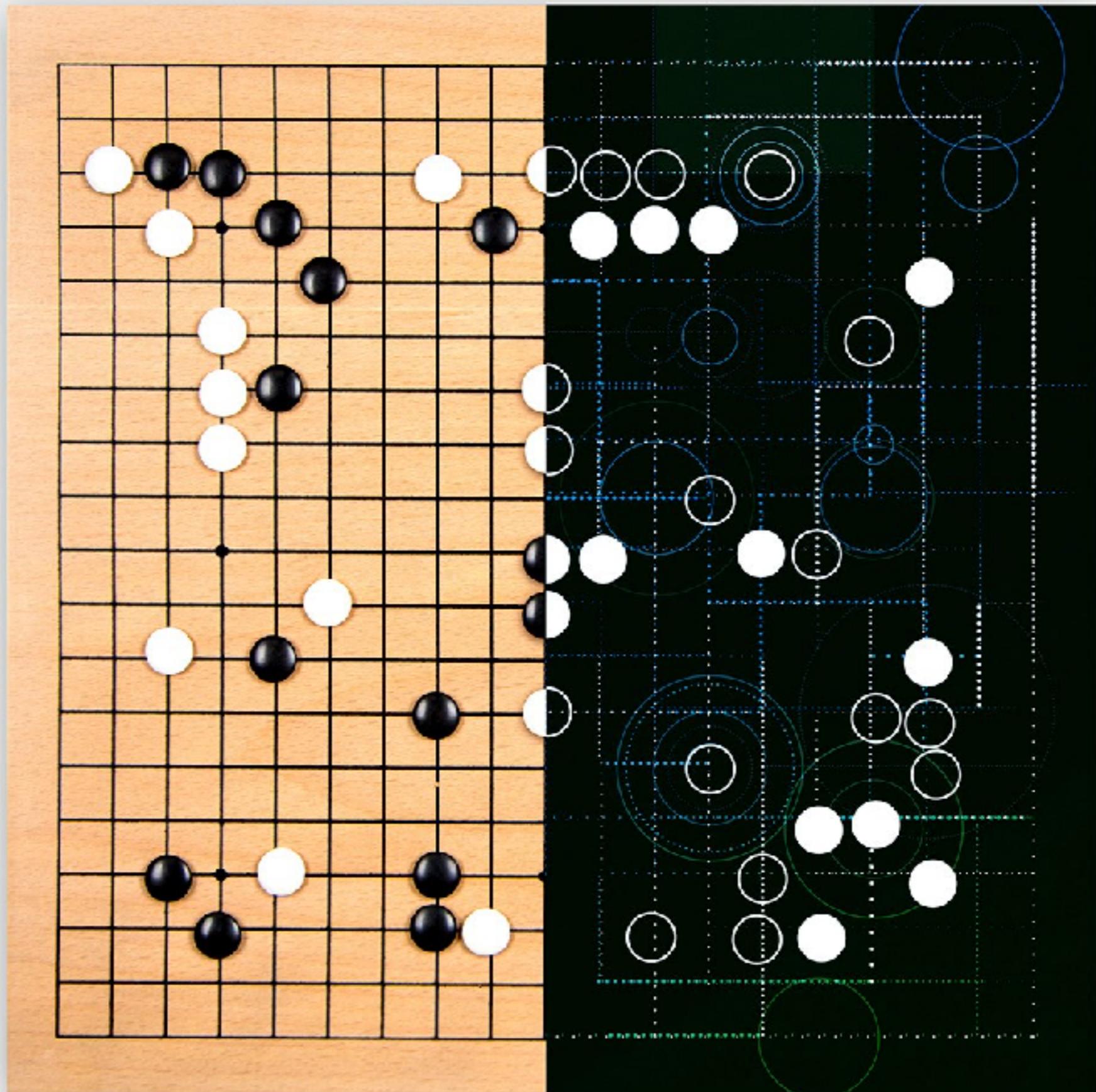


GDAŃSK UNIVERSITY  
OF TECHNOLOGY

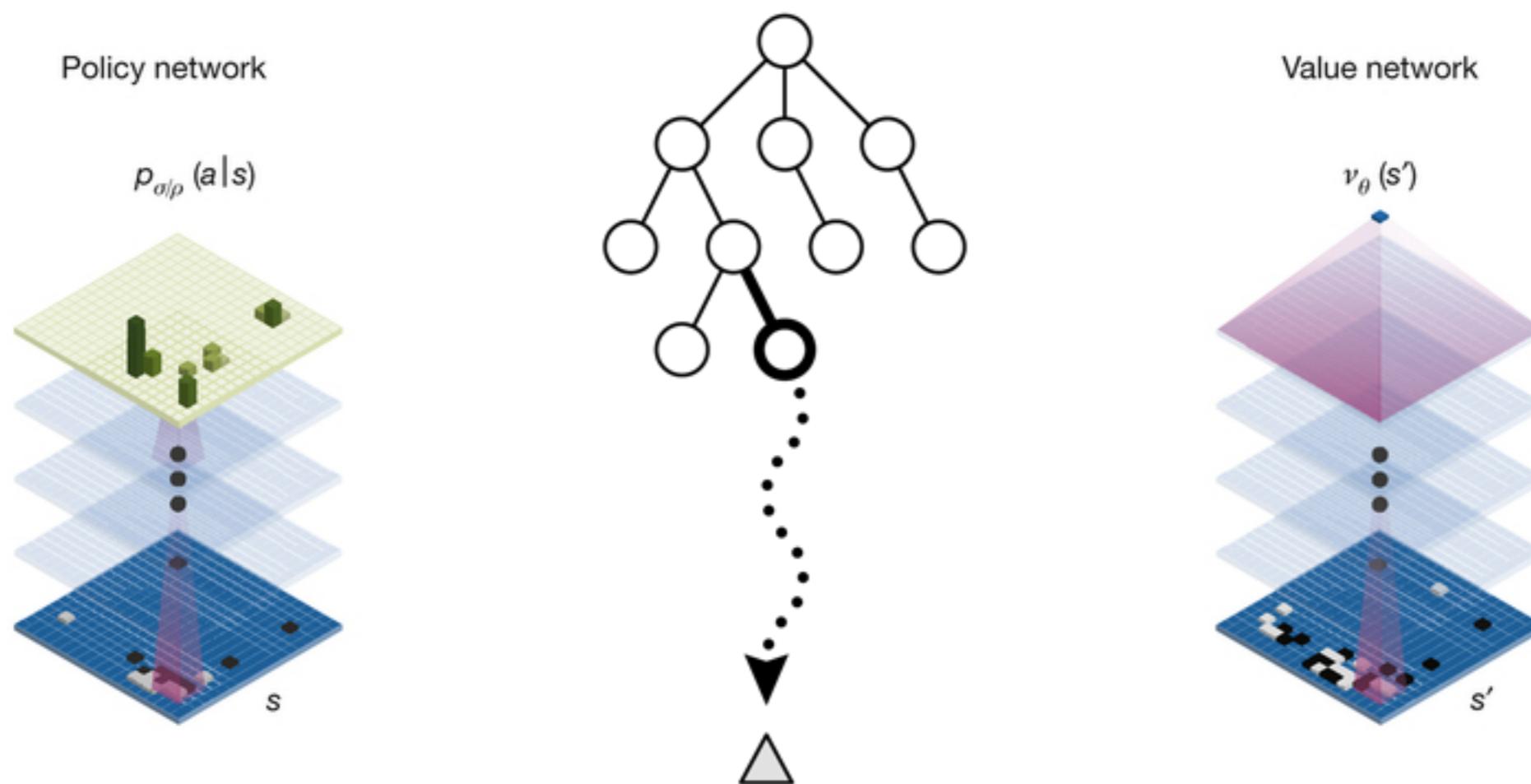
# Deep Reinforcement Learning on AlphaZero example



Piotr Januszewski



# AlphaZero architecture



# DeepMind's AlphaGo Lee

became the first AI to beat a  
human grandmaster of Go.



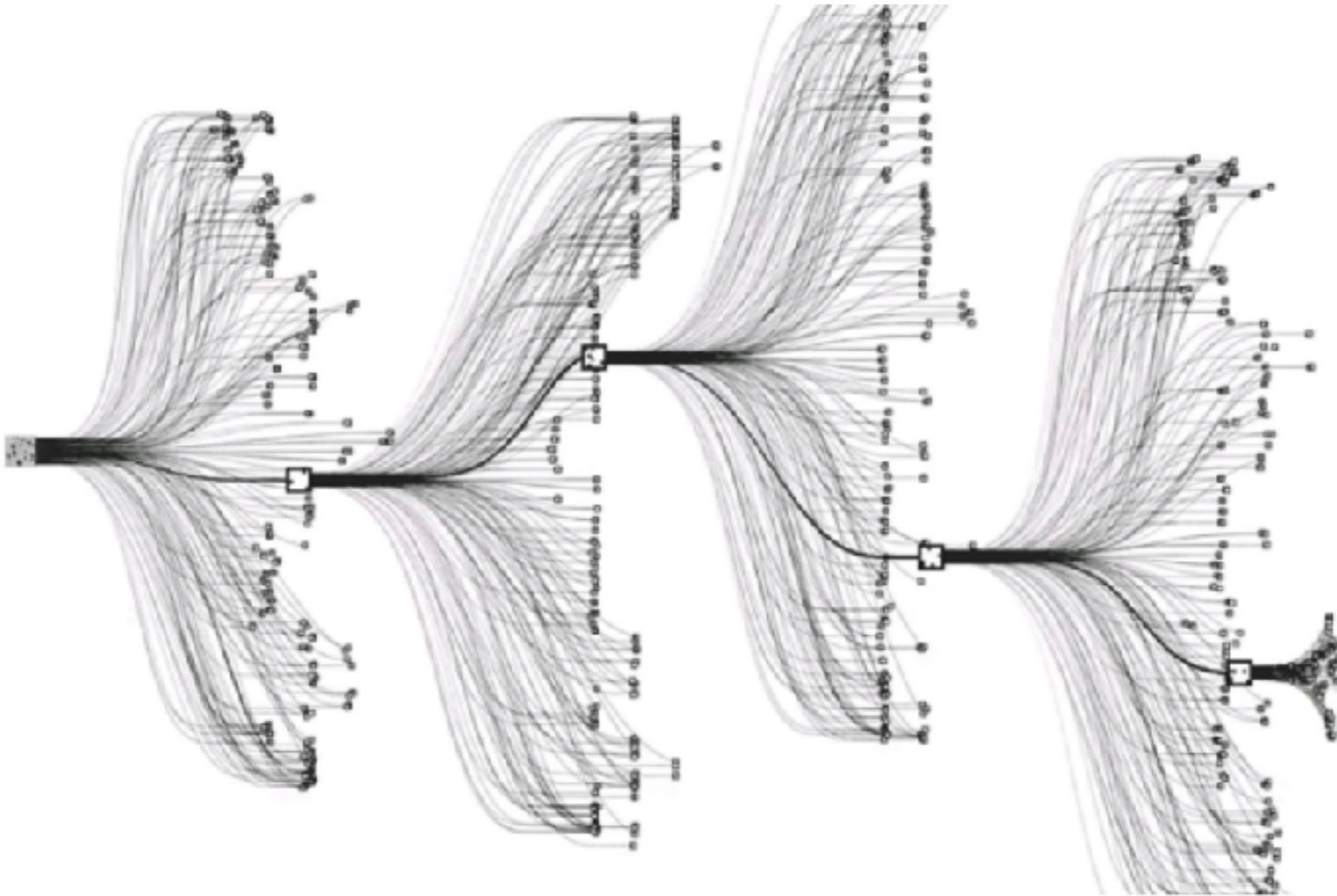
Up to 361 legal moves



Around  $10^{170}$  states



Around  $10^{80}$  atoms



**Black is winning?...**



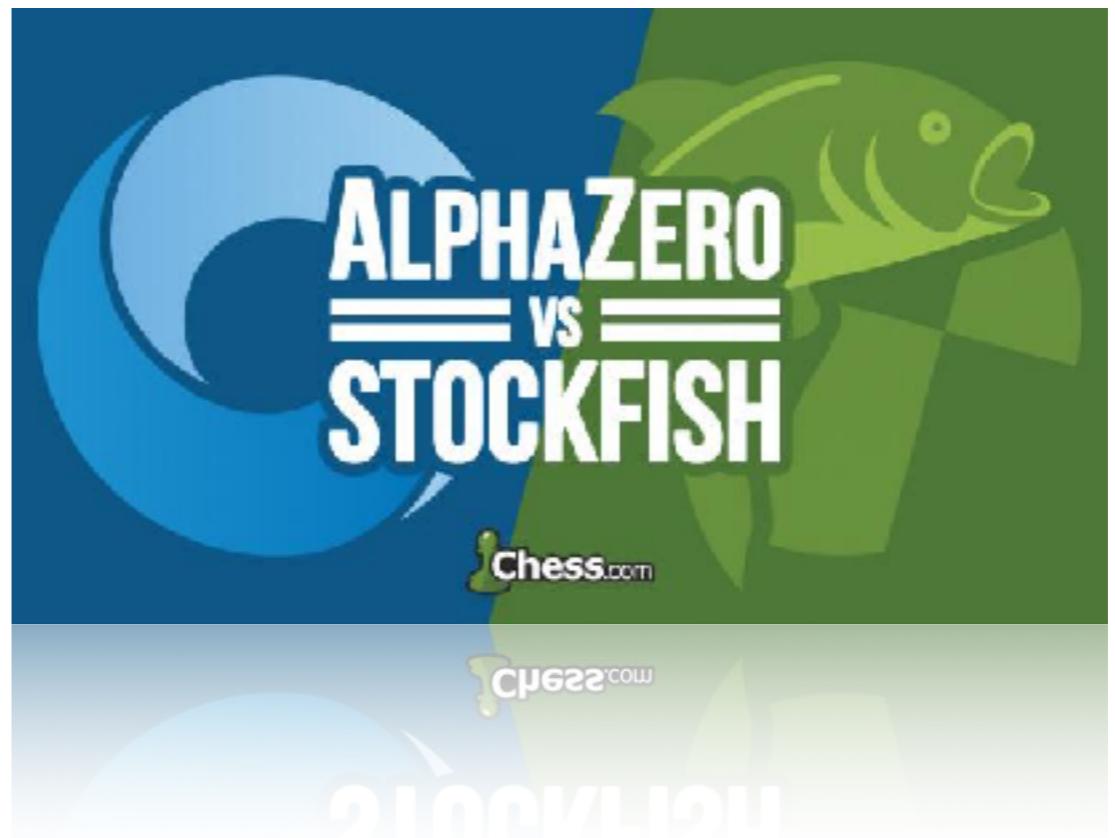
**...or white is winning?**



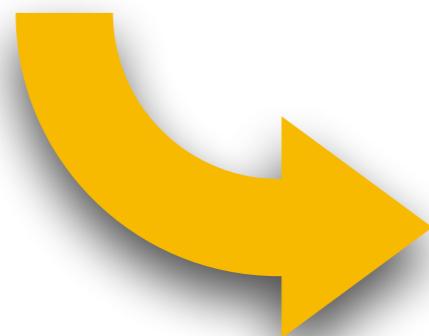
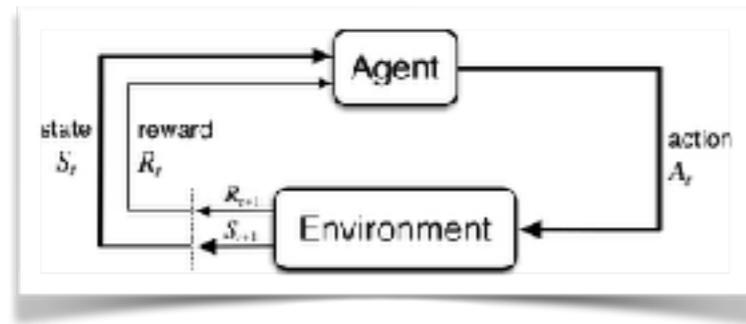
# Learning procedure

# DeepMind's AlphaZero

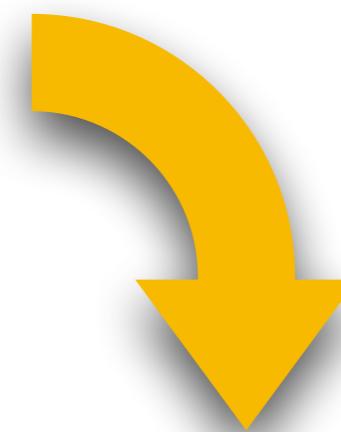
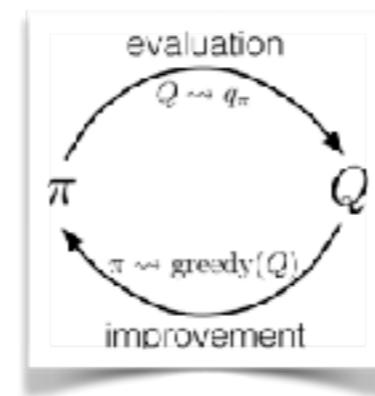
can play other games like  
Chess and Shogi.



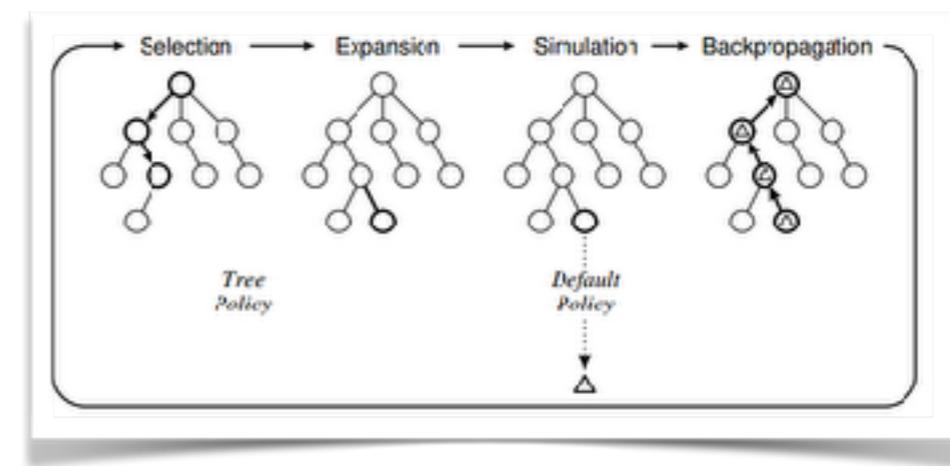
## Reinforcement Learning



## Policy iteration



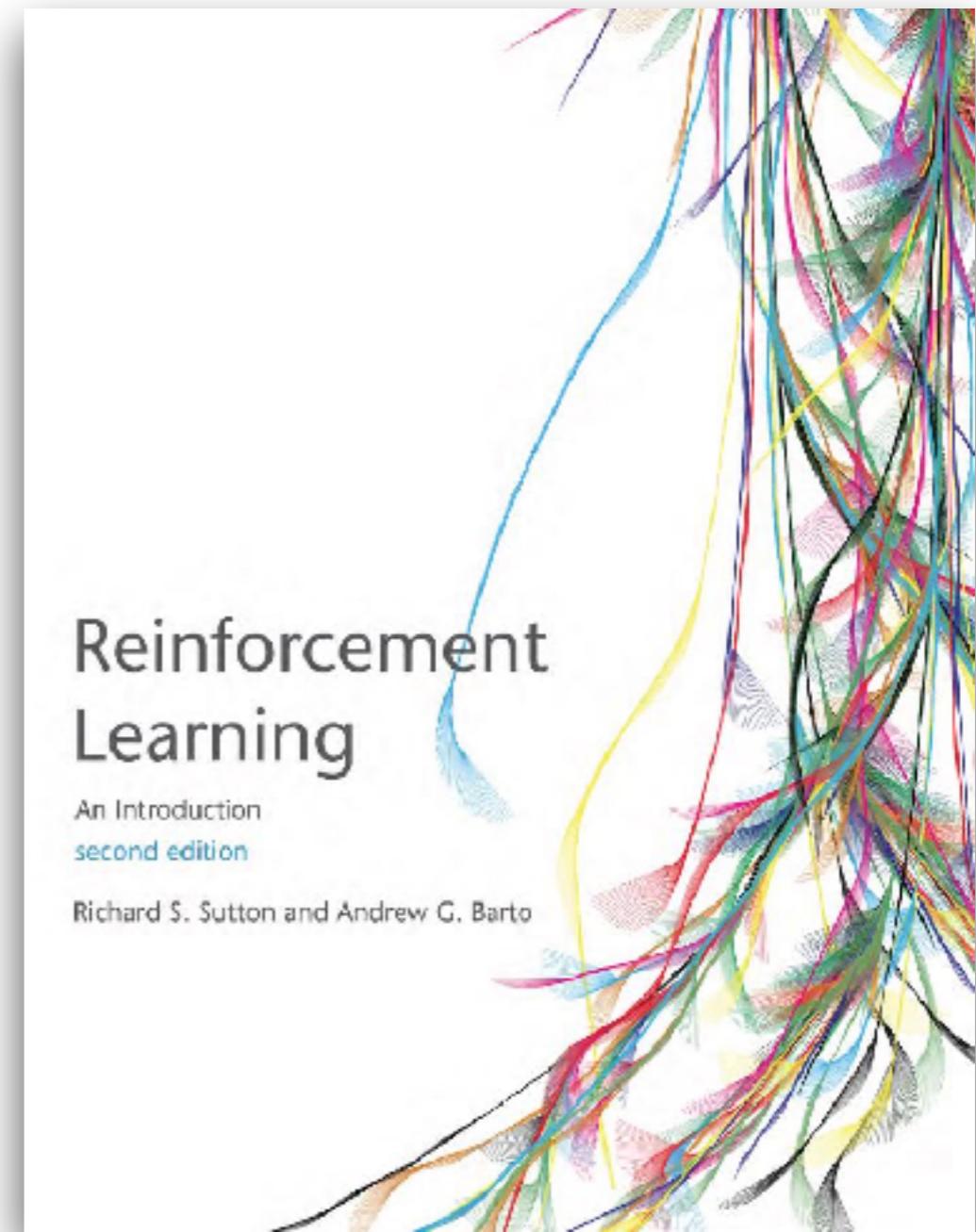
## Simulation-based search and AlphaZero



# Amazing textbook

*Reinforcement Learning: An Introduction*, 2nd edition

Full pdf [here](#) from authors!



# Reinforcement Learning



# Where is it used?

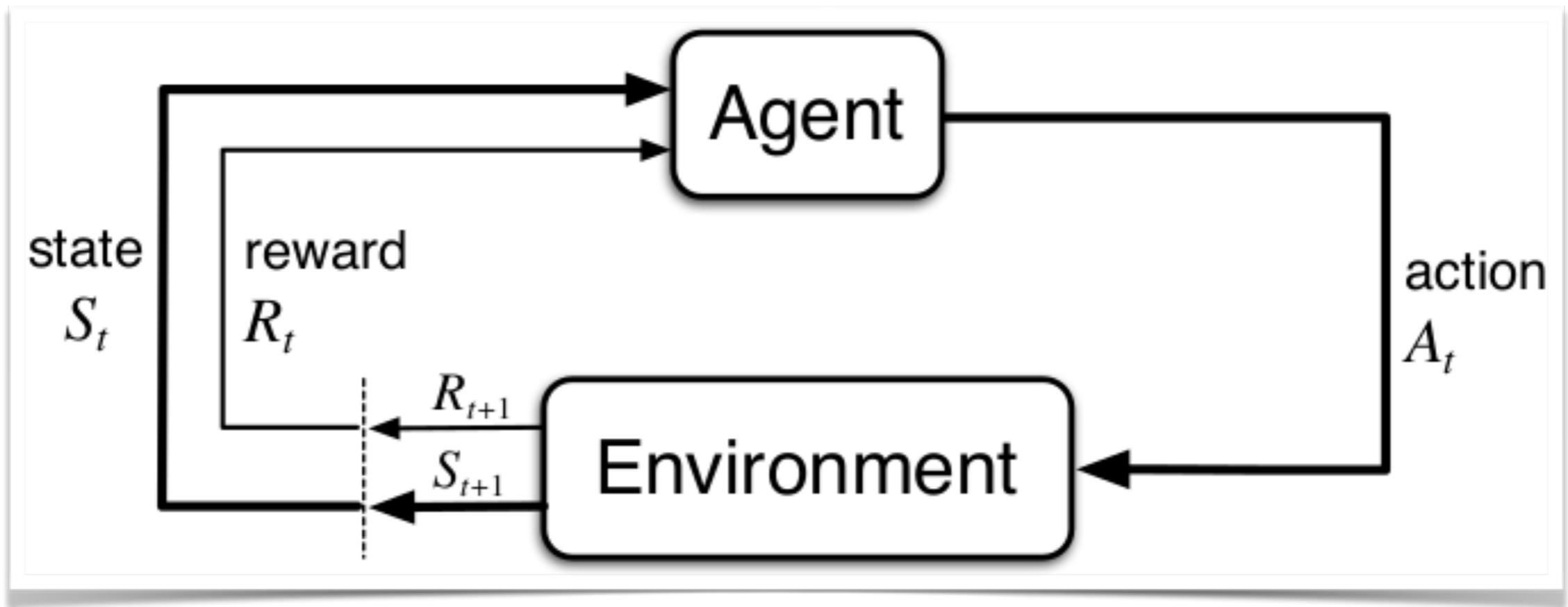
- Robotics
- Video games
- Conversational systems
- Medical intervention
- Algorithm improvement
- Improvisational theatre
- Autonomous driving
- Prosthetic arm control
- Financial trading
- Query completion

# RL is a natural way of learning for animals and humans

E.g. playing video games:

1. You start with exploration, what is possible?
2. You learn what gives you high rewards.
3. And finally, you exploit gained knowledge to win.

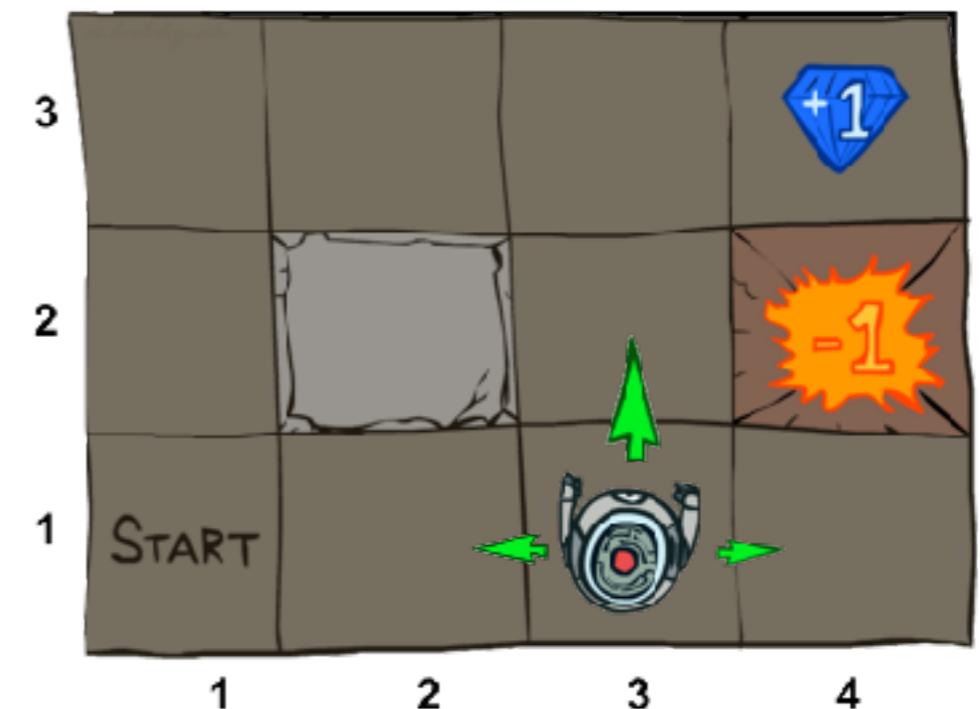




- Inspired by psychology – Agent + Environment
  - Agent selects actions to maximise total reward (return).
- Learning by trial-and-error, in real-time.
- Improves with experience.

# Example: Grid World

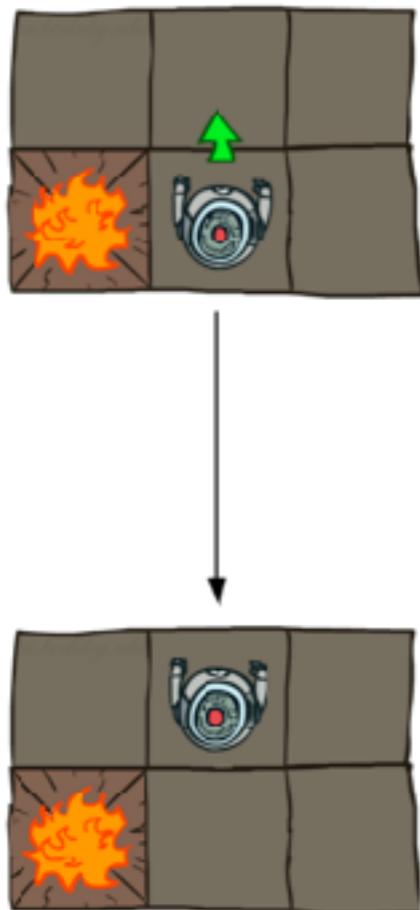
- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
  - States are robot positions
- The agent receives rewards each time step
  - Small “living” reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: maximise sum of rewards



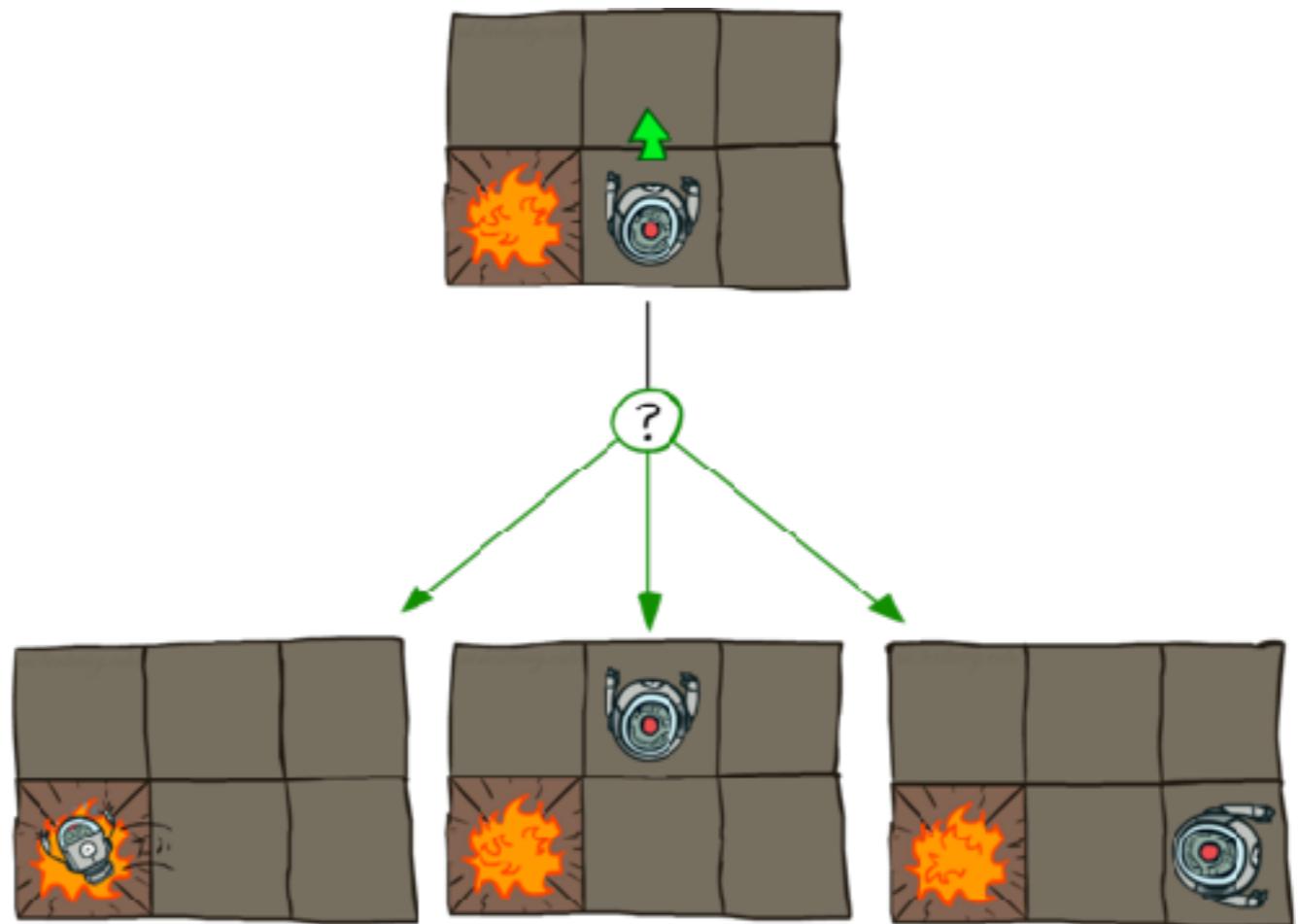
Source: [UC Berkeley CS188 Intro to AI](#)

# Grid World Actions

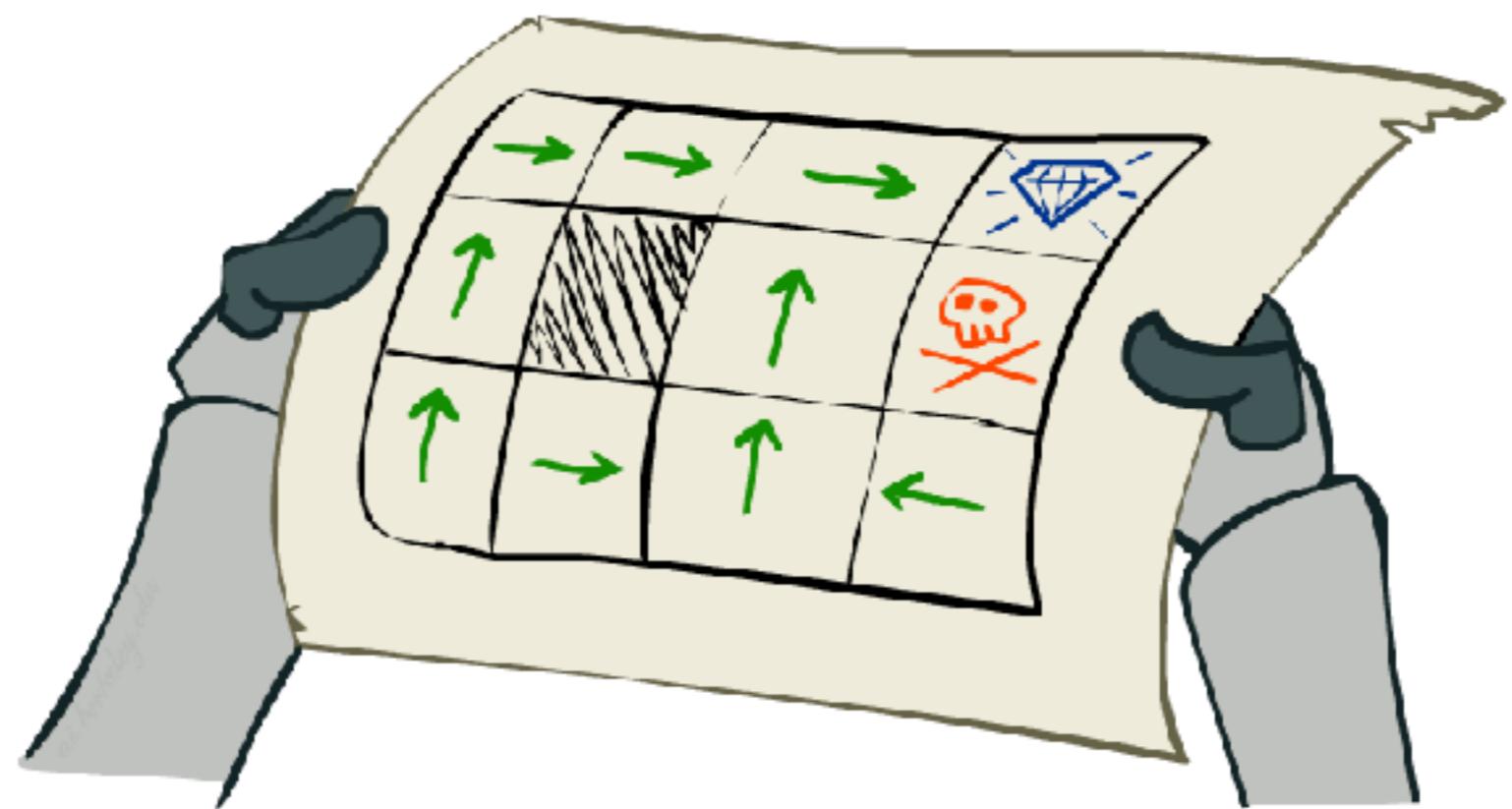
Deterministic Grid World



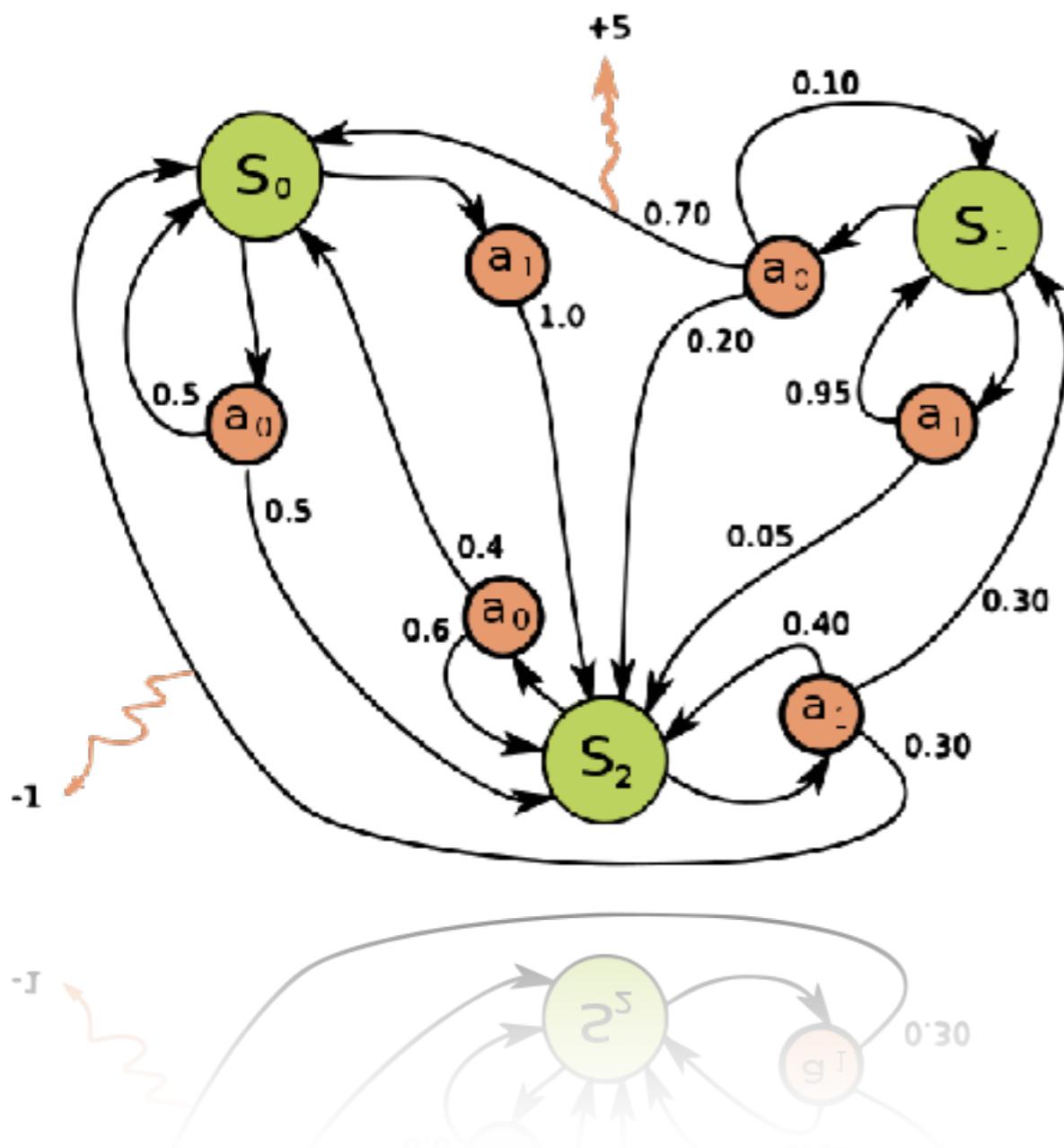
Stochastic Grid World



# Policy tells an agent how it should behave



# Markov Decision Process



- Set of states:  $S$  with initial  $s_0$
- Set of actions:  $A$
- The dynamics of the MDP or transition probabilities:  
$$P_{ss'}^a = Pr(S_{t+1} = s' | S_t = s, A_t = a)$$
- Reward function:  
$$R_{ss'}^a = R(s, a, s')$$
- Policy:  $\pi(s, a) = Pr(A_t = a | S_t = s)$
- Discounting factor:  $\gamma$

# Markov property

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

The probability of reaching  $s'$  from  $s$  depends only on  $s$  and  $a$  and not on the history of earlier states.



**Andrey A. Markov was a Russian mathematician best known for his work on stochastic processes.**

# Expected return

$G_t$  – future expected (discounted) return from time t

$R_t$  – random variable, reward at time t

**Continuous environment**

$$G_t \doteq \mathbb{E}[r_t + r_{t+1} + r_{t+2} + \dots] = \sum_{i=0}^{\infty} R_{t+i}$$

$$G_t \doteq \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots] = \sum_{i=0}^{\infty} \gamma^i R_{t+i}$$



1

Worth Now



$\gamma$

Worth Next Step



$\gamma^2$

Worth In Two Steps

**Episodic environment**

$$G_t \doteq \mathbb{E}[r_t + r_{t+1} + r_{t+2} + \dots + r_T] = \sum_{i=0}^{T-t} R_{t+i}$$

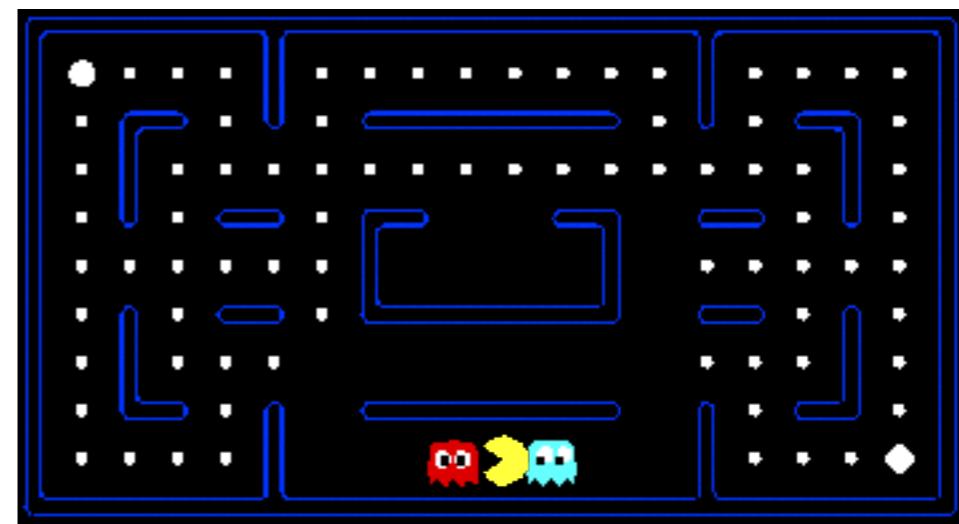
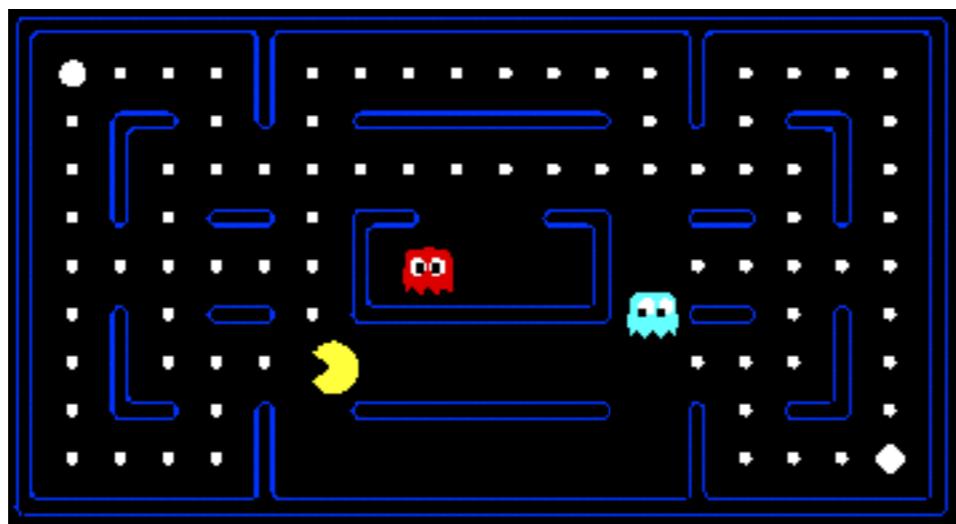
$$G_t \doteq \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{T-t} r_T] = \sum_{i=0}^{T-t} \gamma^i R_{t+i}$$

# Value Function



# Which state has higher “value”?

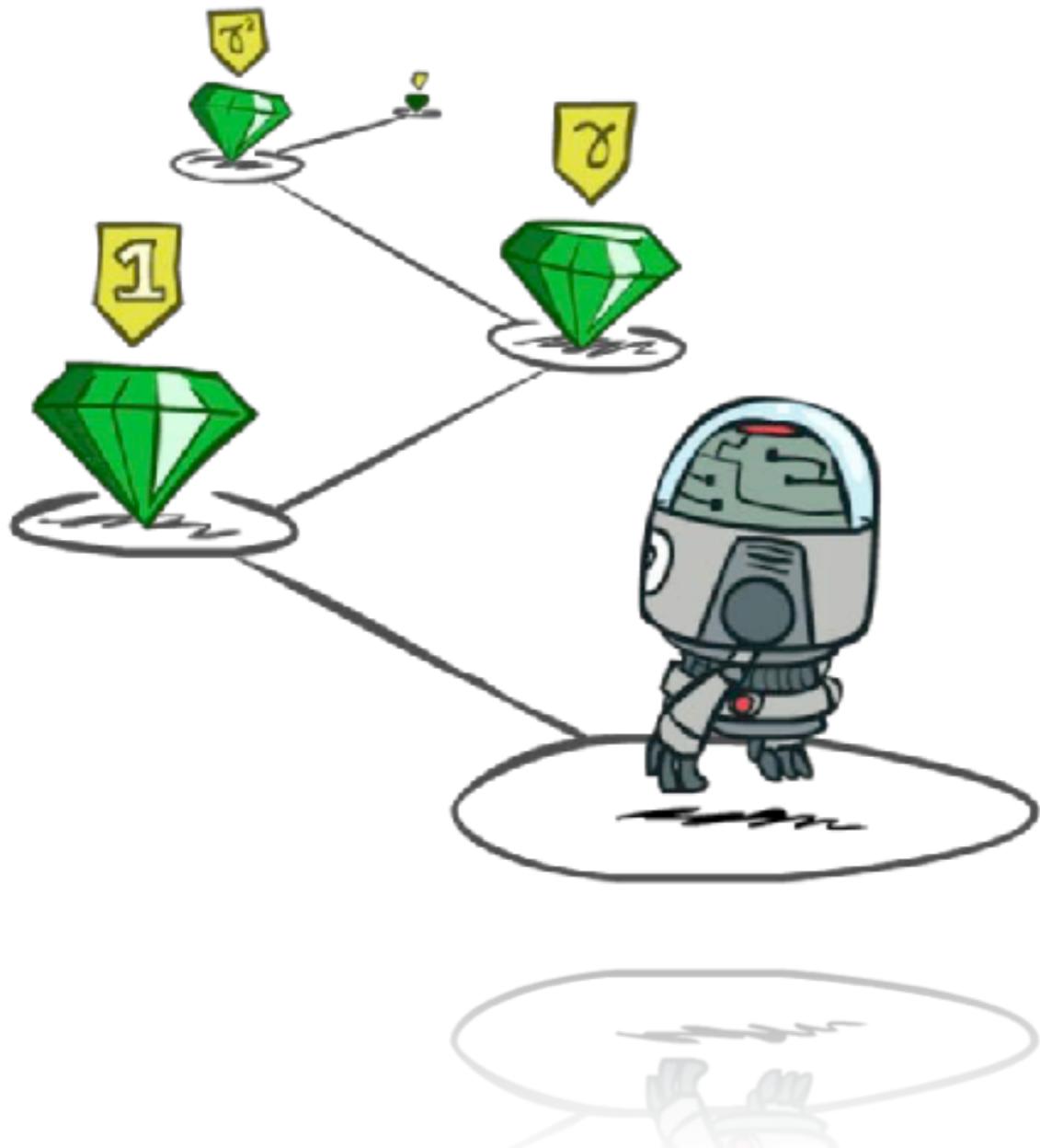
**...or which state is better to be at, if you want to maximise  
the total sum of the collected white dots?**



**But why?**

# Value function

$$v_{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s] = \mathbb{E}_{\pi}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots] = \mathbb{E}_{\pi}[R_{t+1} + \gamma v_{\pi}(S_{t+1})]$$

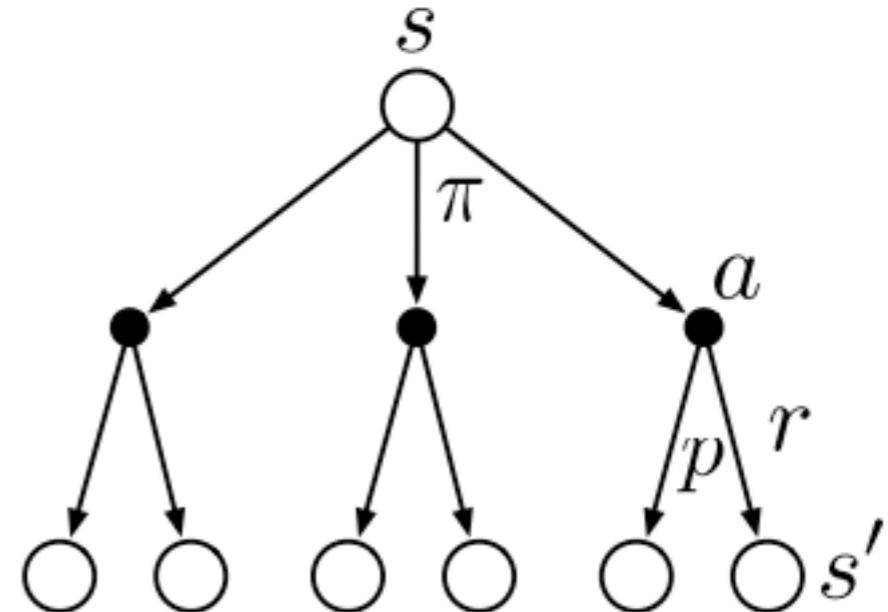


Value function estimates how good it is for the agent to be in a given state. The notion of “how good” here is defined in terms of **future rewards that can be expected**, or to be precise, in terms of **expected return from a given state following agent’s policy**.

# Value function

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})] = \sum_a \pi(s, a) \sum_{s'} Pr(S_{t+1} = s' | S_t = s, A_t = a) [r_{t+1} + \gamma v_\pi(s')]$$

The value of the state must equal the (discounted) value of the expected next state, plus the reward expected along the way.



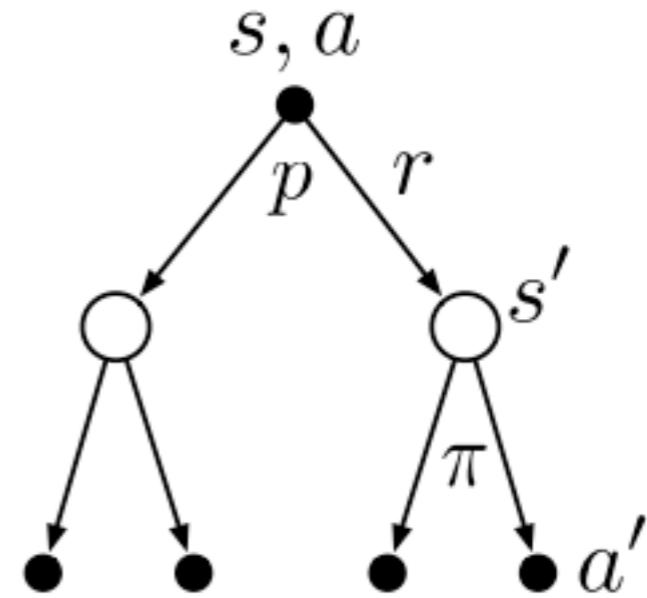
**Backup diagram for value of state “s”**

# Quality function

$$q_\pi(s, a) = \mathbf{E}_\pi[G_t | s_t = s, a_t = a] = \mathbf{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots] = \mathbf{E}_\pi[R_{t+1} + \gamma v_\pi(s')]$$

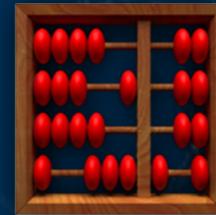
$$v_\pi(s) = \sum_a \pi(s, a) q_\pi(s, a)$$

The quality of the state-action pair is expected return when taking the given action in the given state and following agent's policy afterwards.

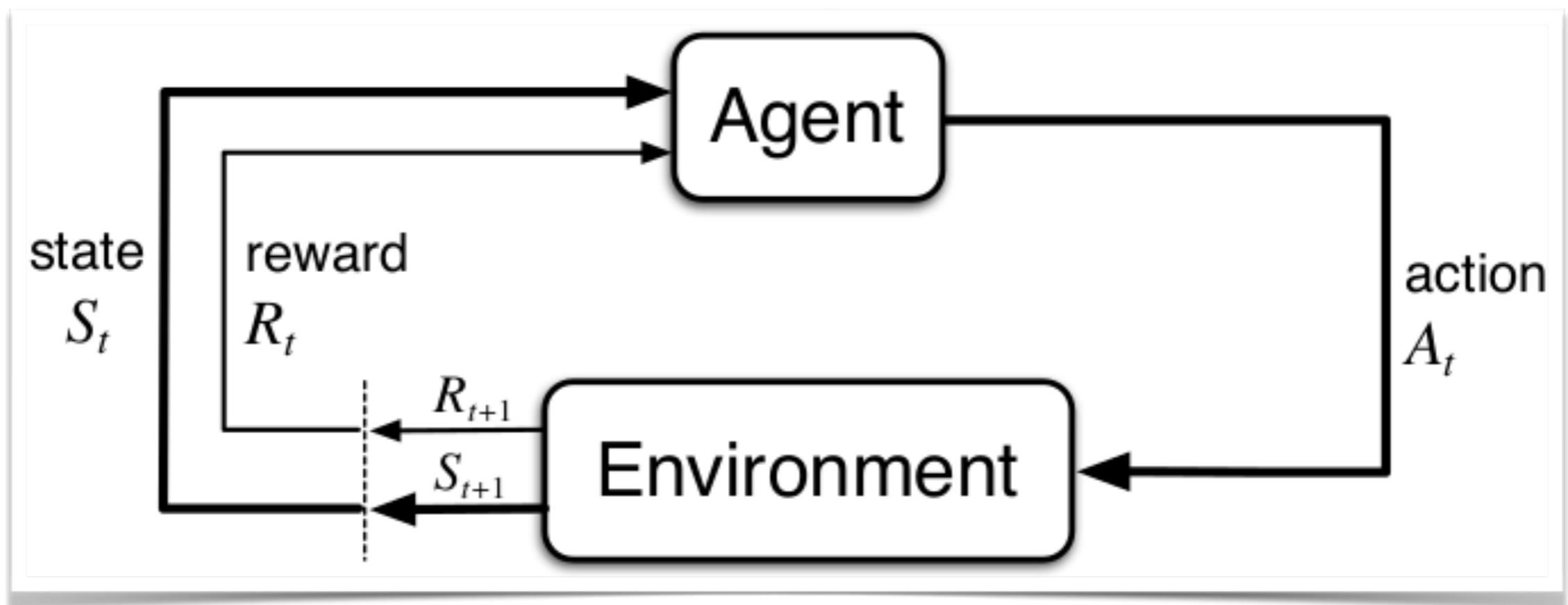


**Backup diagram for quality of state-action pair “s,a”**

# Policy Evaluation



# Reminder

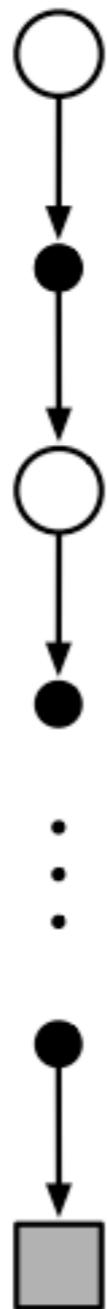


You can only take actions and receive a feedback from the environment

# Monte Carlo!



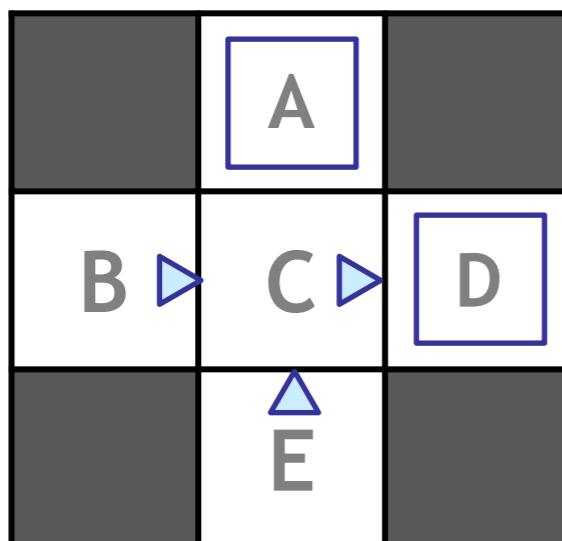
# Monte-Carlo prediction



1. Rollout a given policy to gather experience.
2. Average the returns observed after visiting each state.
3. As more returns are observed, the average will converge to the expected value.

# Monte-Carlo prediction

Input Policy  $\pi$



Assume:  $\gamma = 1$

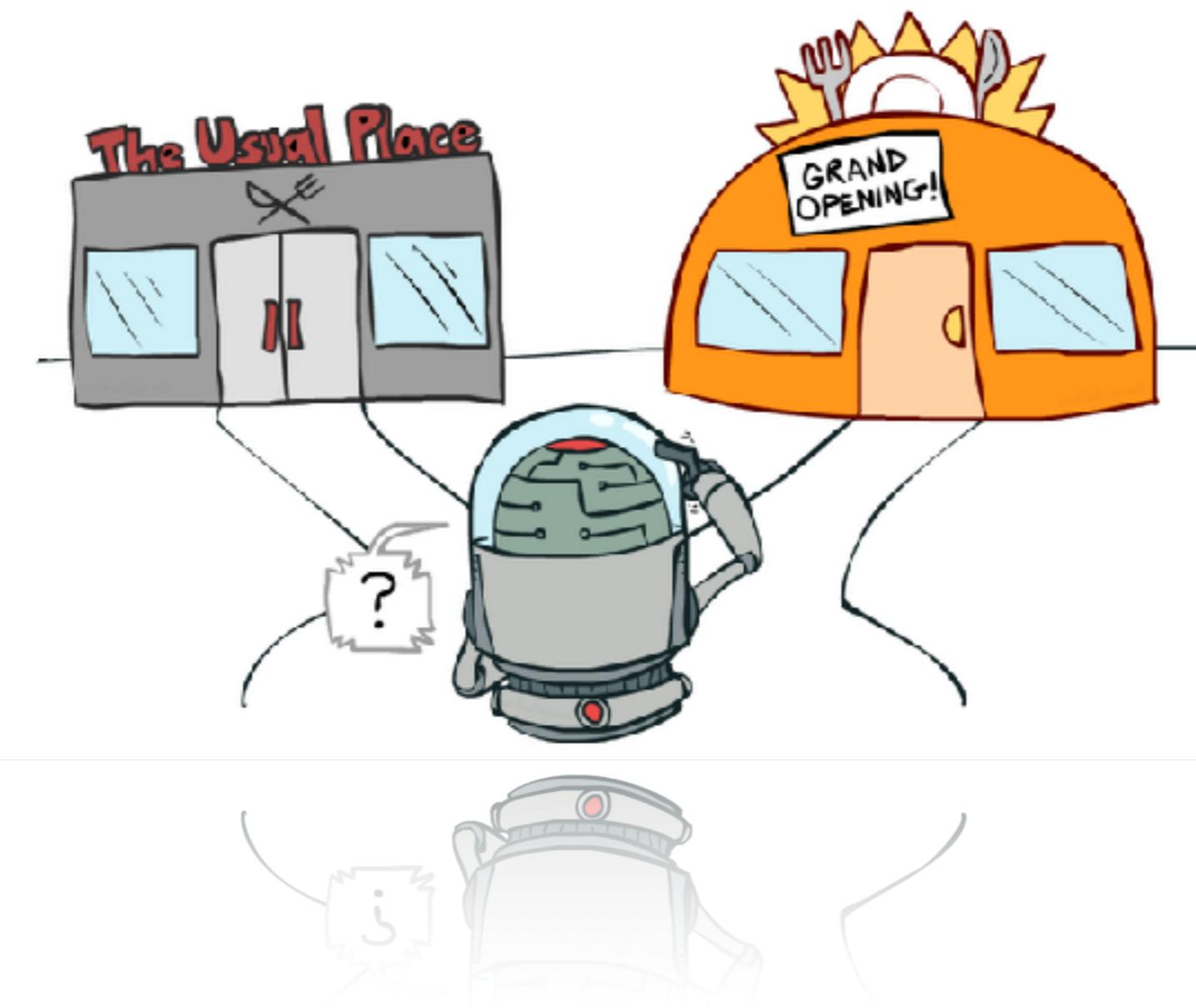
Episodes:

1. B -> East -> -1 -> C -> East -> -1 -> D -> exit -> +10
2. B -> East -> -1 -> C -> East -> -1 -> D -> exit -> +10
3. E -> North -> -1 -> C -> East -> -1 -> D -> exit -> +10
4. E -> North -> -1 -> C -> East -> -1 -> A -> exit -> -10

# Monte-Carlo estimation of quality function

- Track the return after visiting each state-action pair.
- Many state–action pairs may never be visited if a policy is deterministic.
- This means no information about how good are other actions and no way to improve our policy.

# Exploration-exploitation



# Policy Improvement



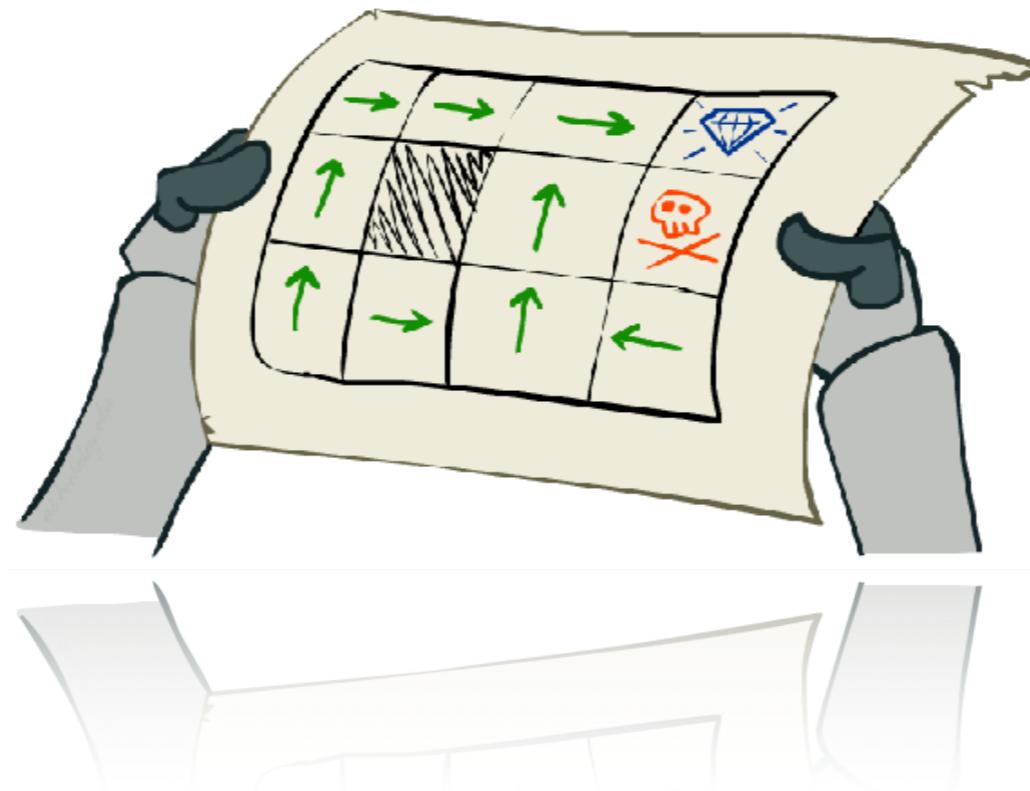
**Choose best  
action now and  
follow your policy  
from next state  
onwards**



# One step look-ahead

$$\pi'(s) \doteq \operatorname*{argmax}_a q_\pi(s, a)$$

$$\forall_s \pi'(s) \geq \pi(s)$$



# $\epsilon$ -greedy policy



**With probability (1 - epsilon):**

$$a \leftarrow \pi(s) \doteq \underset{a}{\operatorname{argmax}} q_{\pi}(s, a)$$

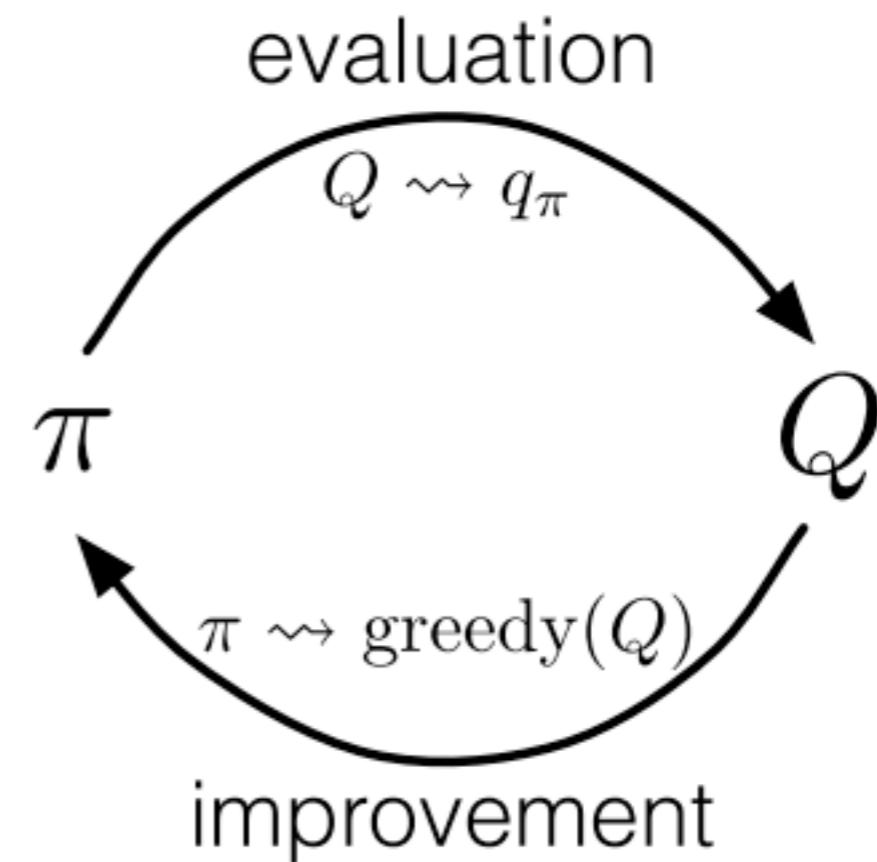
**With probability epsilon:**

$$a \leftarrow \text{random action}$$

# Policy Iteration

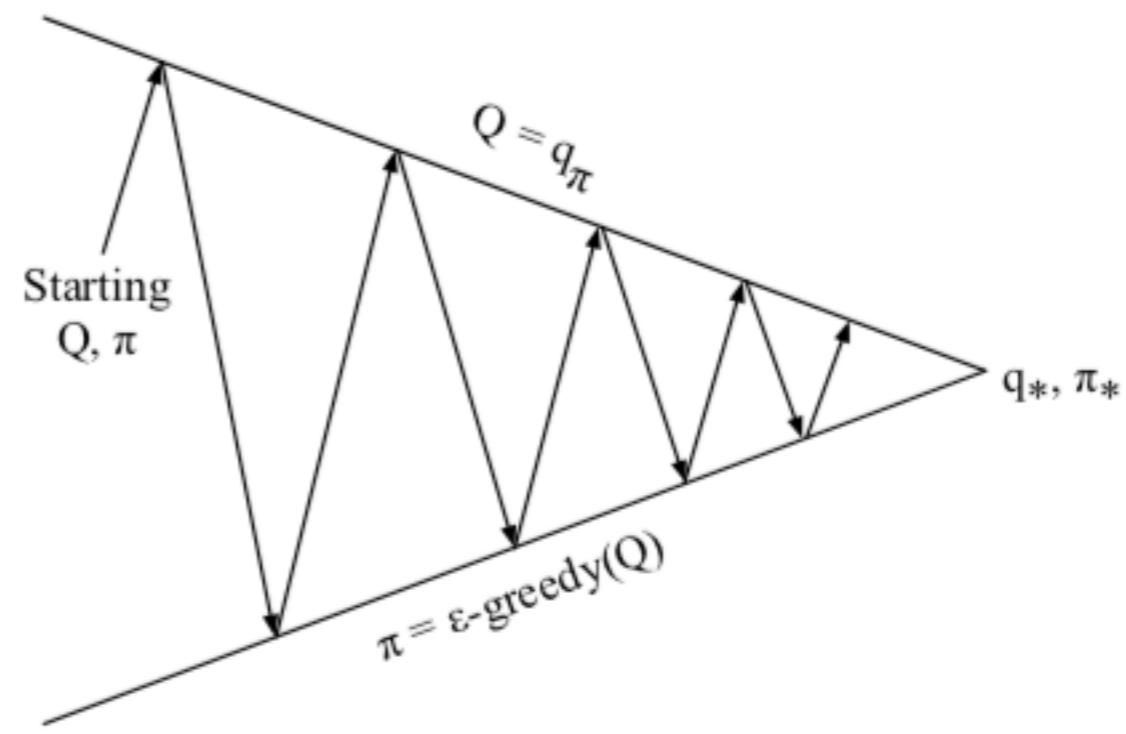


**Evaluate your  
current policy**  
then improve it and start  
over!



# Monte-Carlo control

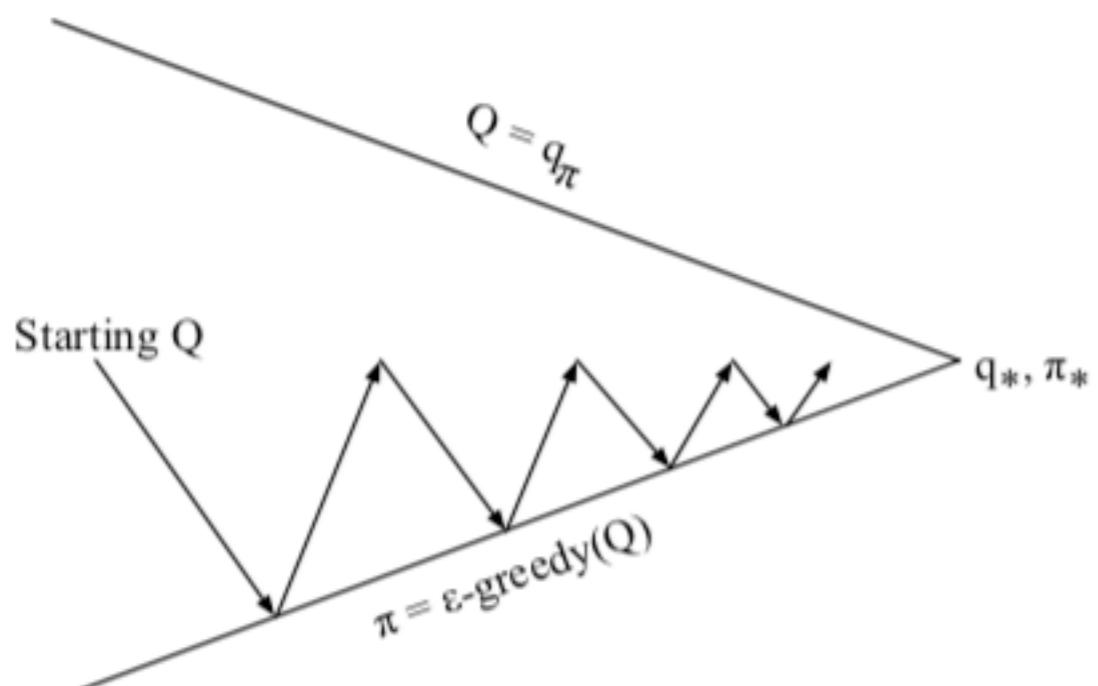
1. Start with an arbitrary policy and Q-values.
2. Alternate between policy evaluation and policy improvement.
3. When the policy doesn't change anymore, you've reached the optimal point.



$$q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_\pi(S_{t+1}, a') | S_t = s, A_t = a] \iff \pi = \pi^*$$

**Bellman optimality equation**

# Iterative implementation



Interleave policy evaluation and policy improvement on an episode-by-episode basis.

Policy iteration will converge, regardless of evaluation and improvement steps arrangement.

# Full algorithm

On-policy first-visit MC control (for  $\varepsilon$ -soft policies), estimates  $\pi \approx \pi_*$

Algorithm parameter: small  $\varepsilon > 0$

Initialize:

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

$Q(s, a) \in \mathbb{R}$  (arbitrarily), for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following  $\pi$ :  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

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

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

Unless the pair  $S_t, A_t$  appears in  $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$ :

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

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

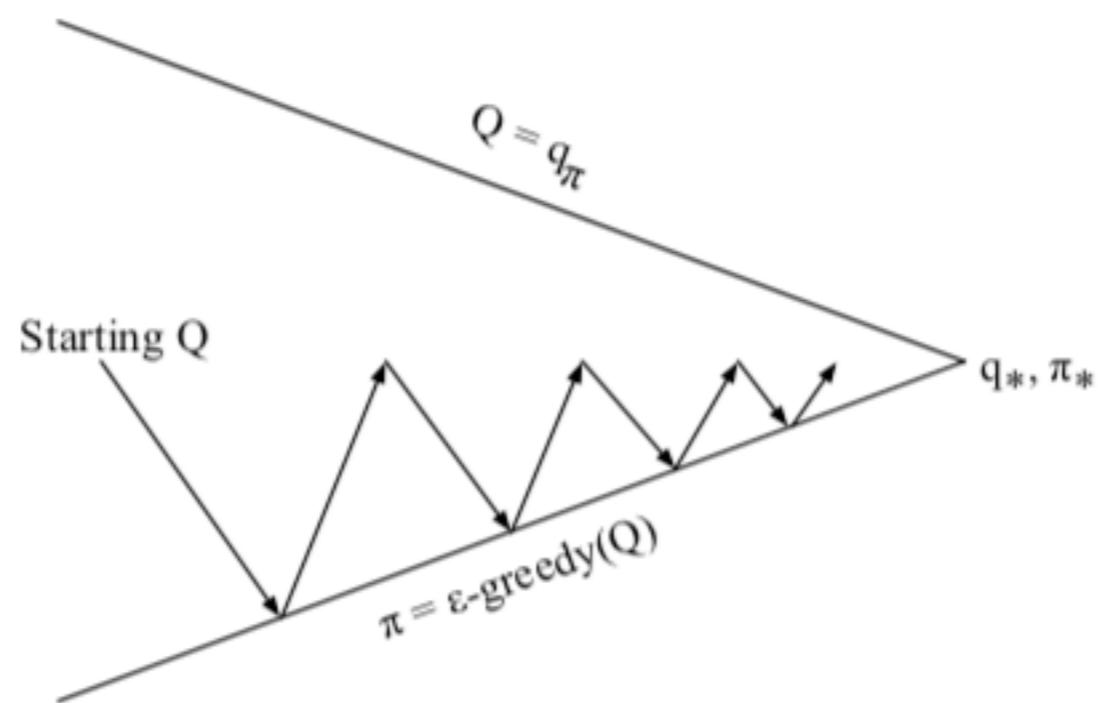
$A^* \leftarrow \arg \max_a Q(S_t, a)$  (with ties broken arbitrarily)

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

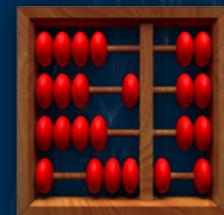
$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

# One last problem

Q-values of what policy do we learn?

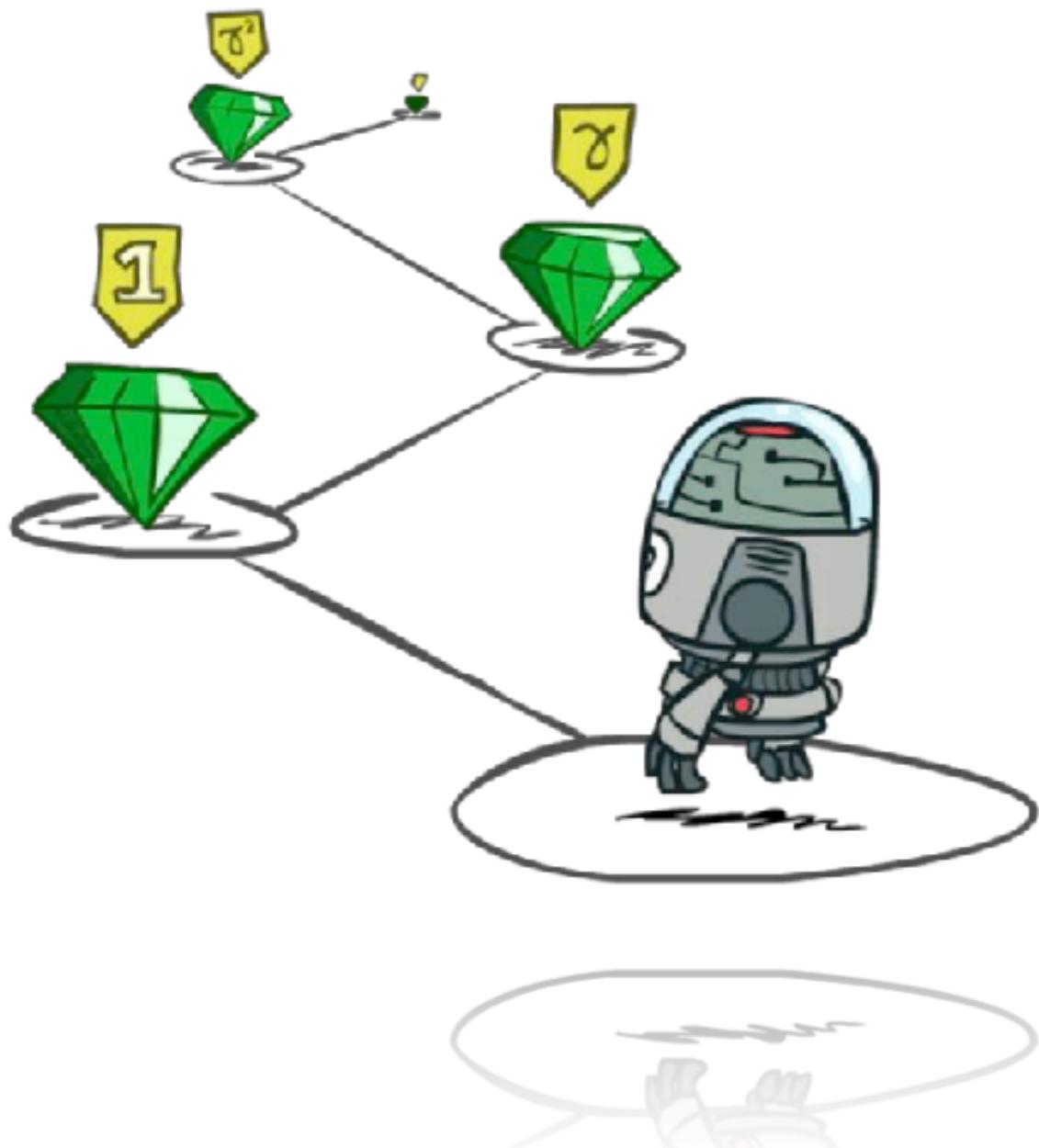


# Policy Evaluation (differently)



# (Recap) Value function

$$v_{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s] = \mathbb{E}_{\pi}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots] = \mathbb{E}_{\pi}[R_{t+1} + \gamma v_{\pi}(S_{t+1})]$$



Value function estimates how good it is for the agent to be in a given state. The notion of “how good” here is defined in terms of **future rewards that can be expected**, or to be precise, in terms of **expected return from a given state following agent’s policy**.

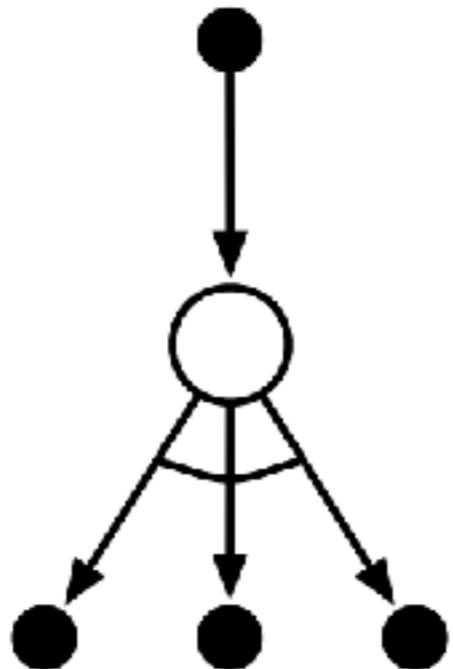
# Temporal-Difference Learning

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



# Q-Learning

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left( \underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\substack{\text{learned value} \\ \text{estimate of optimal future value}}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$



1. Rollout a given policy to gather experience.
2. Use Temporal-Difference Learning at each transition.
3. As more transitions are observed, the average will converge to the **optimal Q-function!** It's called off-policy learning.

# Full algorithm

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  $R, S'$

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

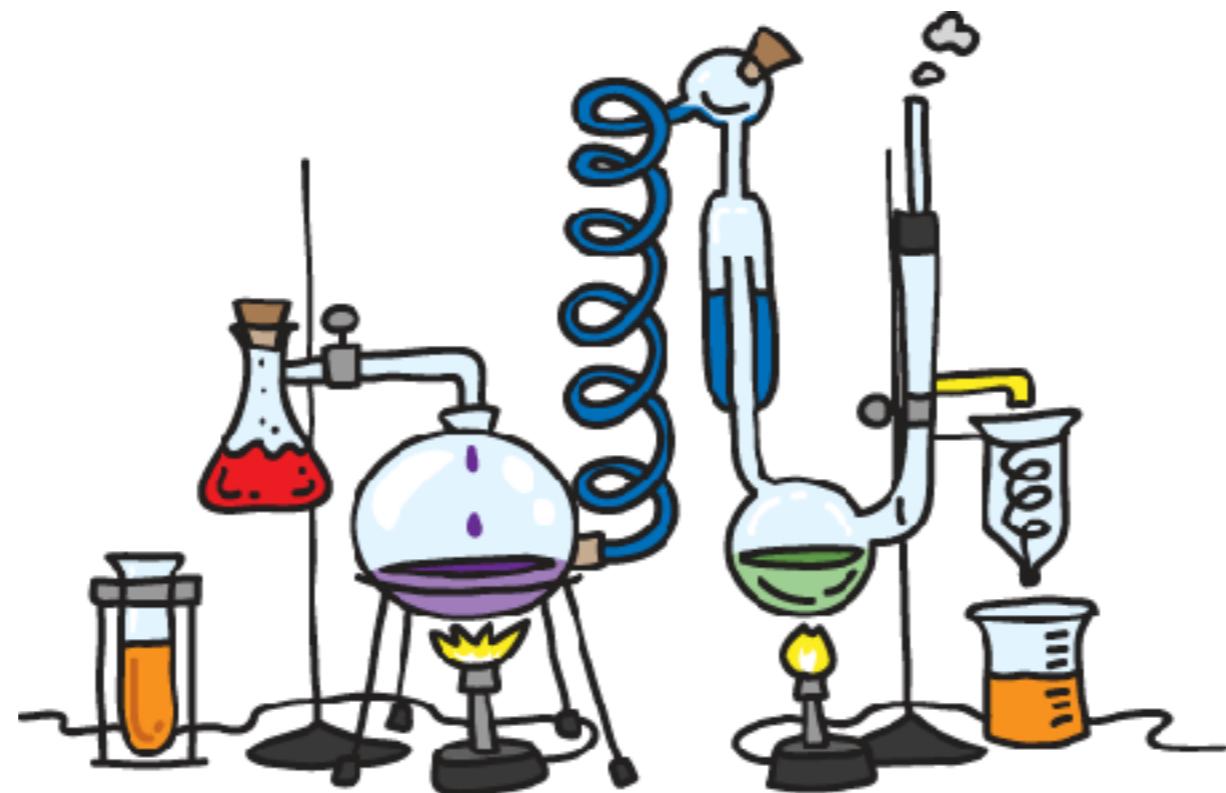
$$S \leftarrow S'$$

    until  $S$  is terminal

# On-policy vs. Off-policy

The policy being learned about is called the target policy, and the policy used to generate behaviour is called the behaviour policy.

- In Q-Learning, the agent learns optimal policy using absolute greedy policy and behaves using other policies such as e-greedy policy. Because the target policy is different to the behaviour policy, so Q-Learning is off-policy algorithm.
- In MC-control, the agent learns optimal policy and behaves using the same policy such as e-greedy policy. Because the target policy is the same to the behaviour policy, so MC-control is on-policy algorithm.

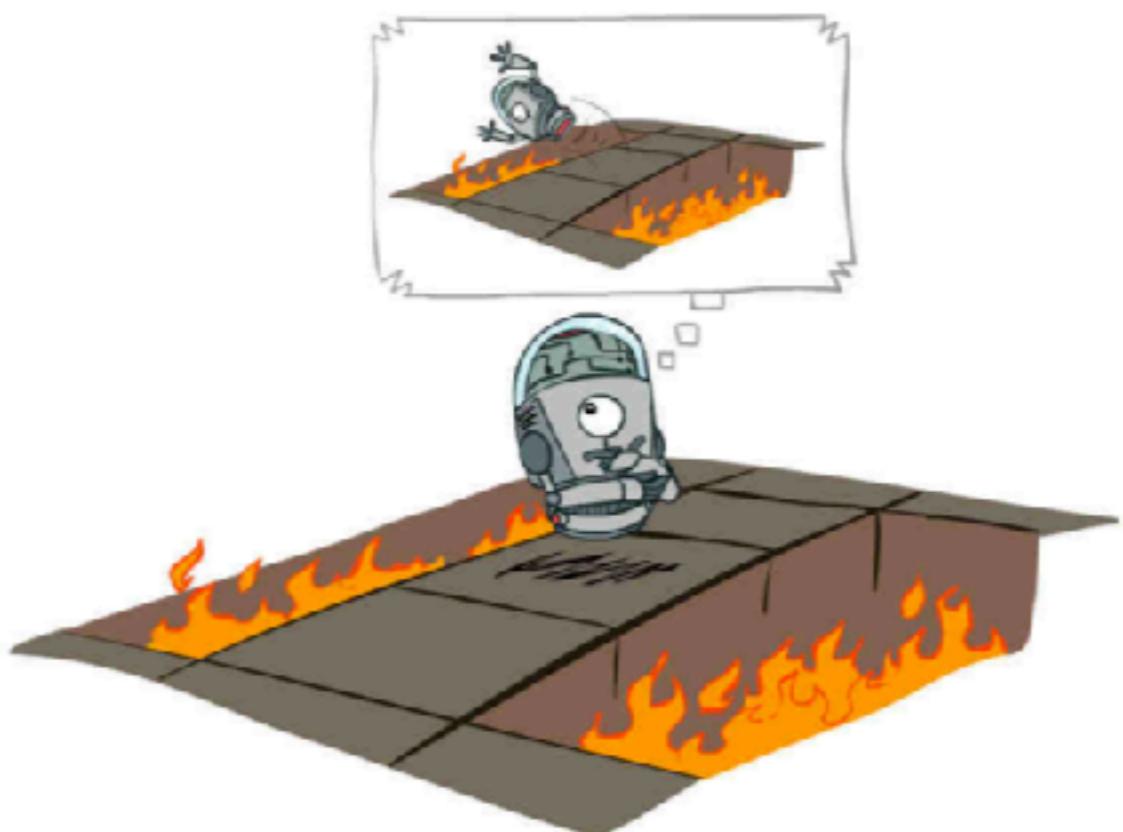


**Do it yourself!**

# Simulation-based search



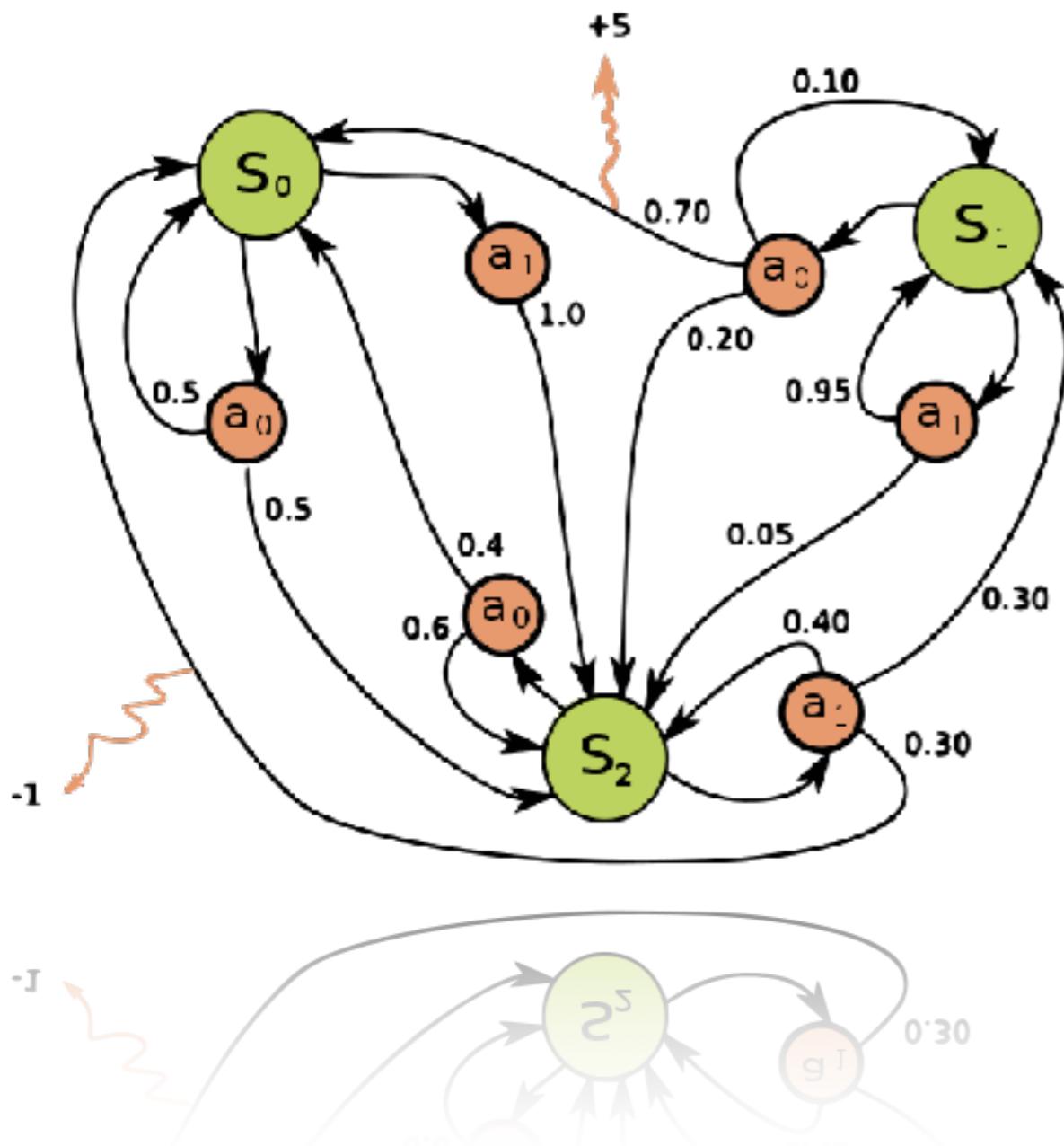
# Planning vs. learning



# Temporality



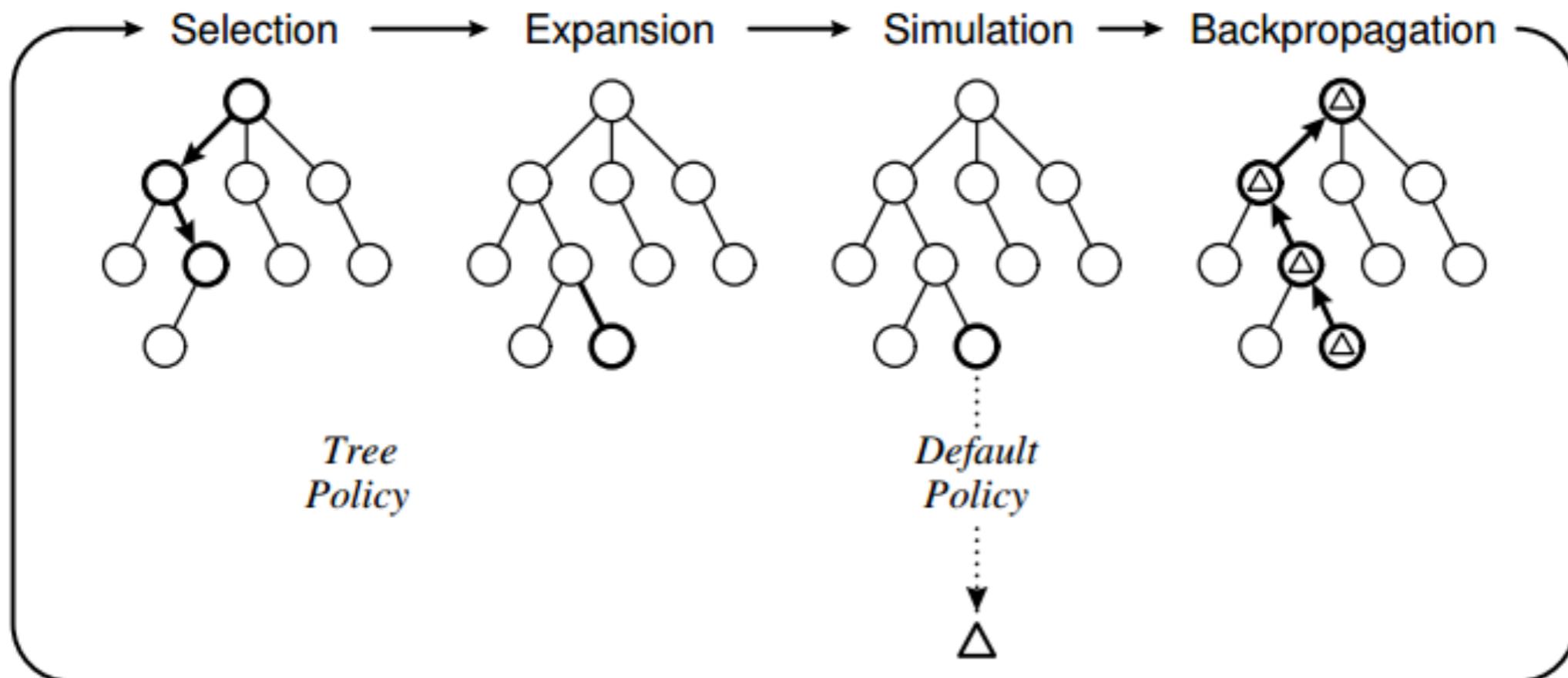
# sub-MDP



- Set of states:  $S$  with initial  $s_t$
- Set of actions:  $A$
- The dynamics of the MDP or transition probabilities:  
$$P_{ss'}^a = \Pr(S_{t+1} = s' | S_t = s, A_t = a)$$
- Reward function:  
$$R_{ss'}^a = R(s, a, s')$$
- Policy:  $\pi(s, a) = \Pr(A_t = a | S_t = s)$
- Discounting factor:  $\gamma$

# Monte-Carlo Tree Search

in policy iteration framework



Each node keeps track of:

$N(s)$  – a total count for the state

$N(s, a)$  – a total count for the action in the state

$Q(s, a)$  – value of the action in the state

# What about exploration?

Tree policy in Monte-Carlo Tree Search (MCTS)

$$\pi_{tree}(s) \doteq \operatorname{argmax}_a Q(s, a)$$

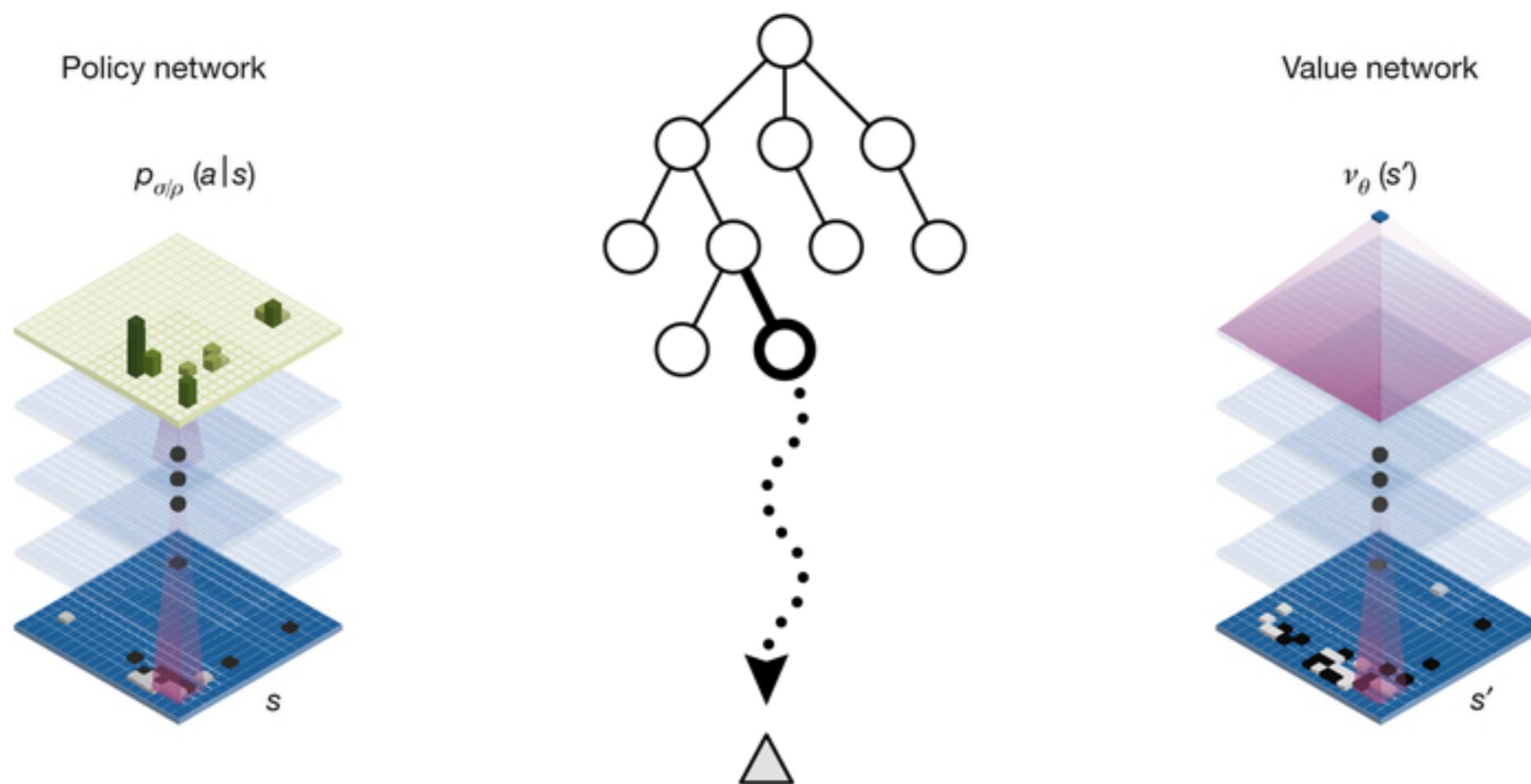
Tree policy in Upper Confidence Tree (UCT)

$$\pi_{tree}(s) \doteq \operatorname{argmax}_a \left[ Q(s, a) + c \sqrt{\frac{\ln N(s)}{N(s, a)}} \right]$$

# AlphaZero demystified



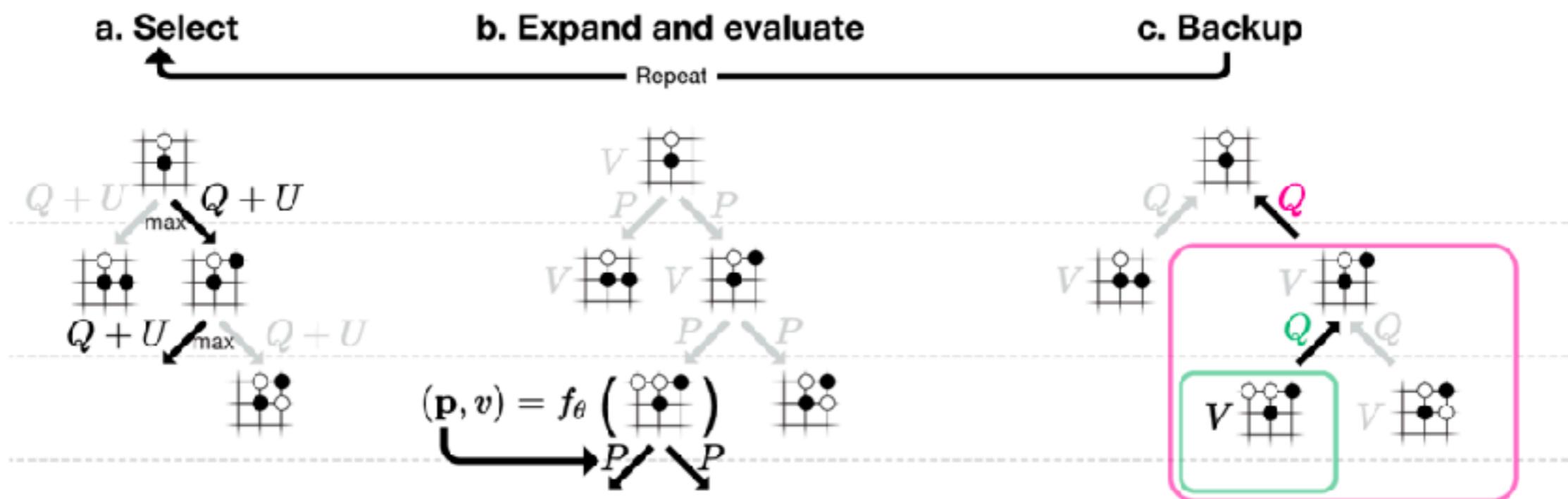
# AlphaZero architecture



# Requirements on problem kind

- **discrete** - states and actions sets form a discontinuous sequences, put simply they are discrete.
- **deterministic** - every move has one possible outcome.
- with **perfect information** - both players have perfect knowledge about current game state.
- access to **game rules** - there need to be some sort of environment simulator.

# AlphaZero Tree Search



# Efficiency Through Value Function

Games like chess  and Go  have very (in Go I mean VERY) large state space - legal board game positions.

**Any ideas how to obtain the value function?**

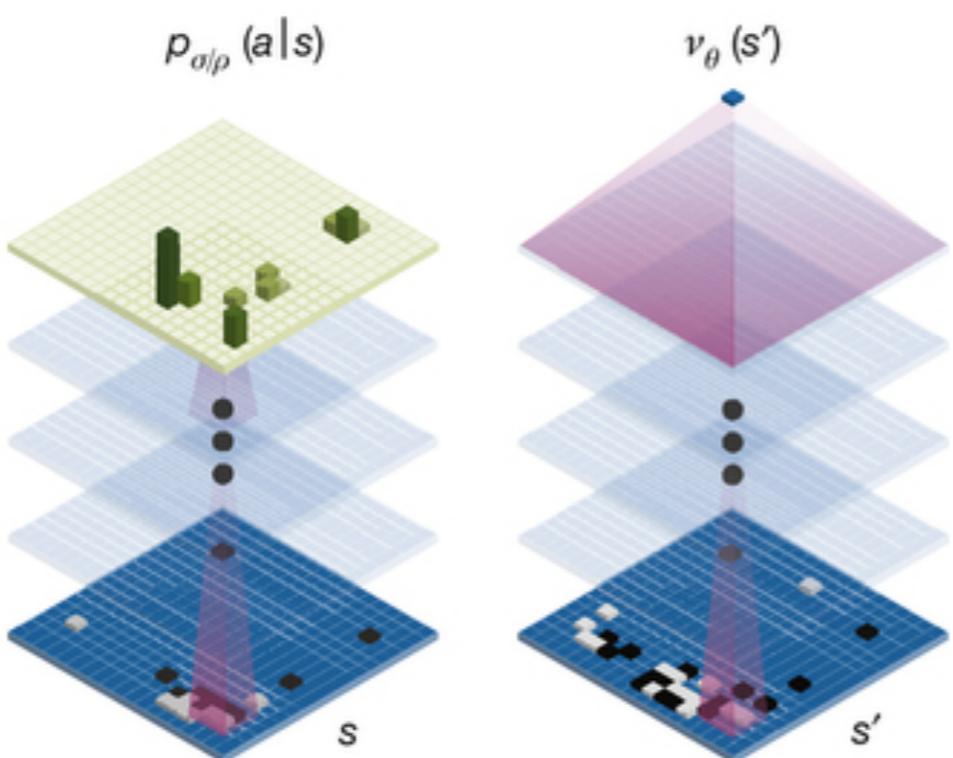
# Efficiency Through Expert Policy

Also, games like chess  and Go  have very (in Go I mean VERY) large branching factor - possible moves in each state.

$$\pi_{tree}(s) \doteq \operatorname{argmax}_a \left[ Q(s, a) + c \cdot P(s, a) \cdot \frac{\sqrt{N(s)}}{1 + N(s, a)} \right]$$

**Any ideas how to obtain the expert policy?**

# How to model the value function and the expert policy?



# Train through Policy Iteration

## SELF PLAY

Create a 'training set'

The best current player plays 25,000 games against itself

See MCTS section to understand how AlphaGo Zero selects each move

At each move, the following information is stored



The game state  
(see 'What is a Game State section')



The search probabilities  
(from the MCTS)



The winner  
(+1 if this player won, -1 if  
this player lost - added once  
the game has finished)

## RETRAIN NETWORK

Optimise the network weights

### A TRAINING LOOP

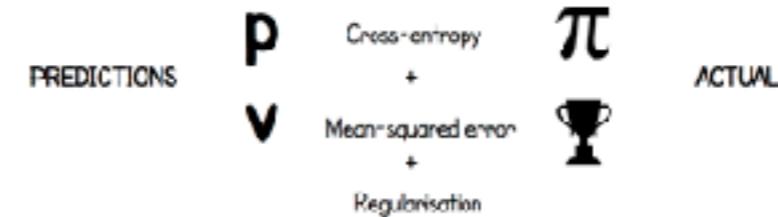
Sample a mini-batch of 2048 positions from the last 500,000 games

Retrain the current neural network on these positions

- The game states are the input (see 'Deep Neural Network Architecture')

#### Loss function

Compares predictions from the neural network with the search probabilities and actual winner



After every 1,000 training loops, evaluate the network

Full infographic in references, highly recommend to give a look

# Case Study



# Summary

- We've learned the basics of how to train an agent to make optimal decisions in an environment.
- Reinforcement Learning is mostly theoretical field because of time and space complexities of presented algorithms.
- Deep Reinforcement Learning is “more practical” thanks to advances in Deep Learning (function approximation).
- Deep Reinforcement Learning is under heavy research and development and doesn't really work yet.

# Thank you 😊

You can find this presentation and AlphaZero implementation (along other projects) on my GitHub:  
<https://github.com/piojanu>

# Sources and Good Reads

- [A Simple Alpha\(Go\) Zero Tutorial](#) by Surag Nair
- [AlphaGo Zero cheatsheet](#) by David Foster
- [AlphaGo Zero - How and Why it Works](#) by Tim Wheeler
- [Deep Reinforcement Learning: Pong from Pixels](#) by Andrej Karpathy
- [UC Berkeley AI course](#)
- [Mastering the game of Go without Human Knowledge](#) by David Silver et al.
- [UCL Course on RL](#) by David Silver
- [The Multi-Armed Bandit Problem and Its Solutions](#) by Lilian Weng
- [Deep Learning and Reinforcement Learning Summer School, Montreal 2017](#)
- Reinforcement Learning: An Introduction 2nd edition
- **AlphaGo movie on Netflix** 