# TDDE13 LE3 HT2024

## Multi-agent Learning

**Fredrik Heintz**

**Dept. of Computer Science**
**Linköping University**

**fredrik.heintz@liu.se**
**@FredrikHeintz**

**Outline:**

- **Reinforcement learning**
- **Deep reinforcement learning**
- **Multi-agent reinforcement learning**

LINKÖPING UNIVERSITY

# Classes of Learning Problems

## Supervised Learning

**Data:** $(x, y)$
$x$ is data, $y$ is label

**Goal:** Learn function to map
$x \rightarrow y$

**Apple example:**

This thing is an apple.

## Unsupervised Learning

**Data:** $x$
$x$ is data, no labels!

**Goal:** Learn underlying structure

**Apple example:**

This thing is like the other thing.

## Reinforcement Learning

**Data:** state-action pairs

**Goal:** Maximize future rewards over many time steps

**Apple example:**

Eat this thing because it will keep you alive.

MIT 6.S191

# Reinforcement Learning: Key Concepts


**Agent**: takes actions.

MIT 6.S191

# Reinforcement Learning: Key Concepts

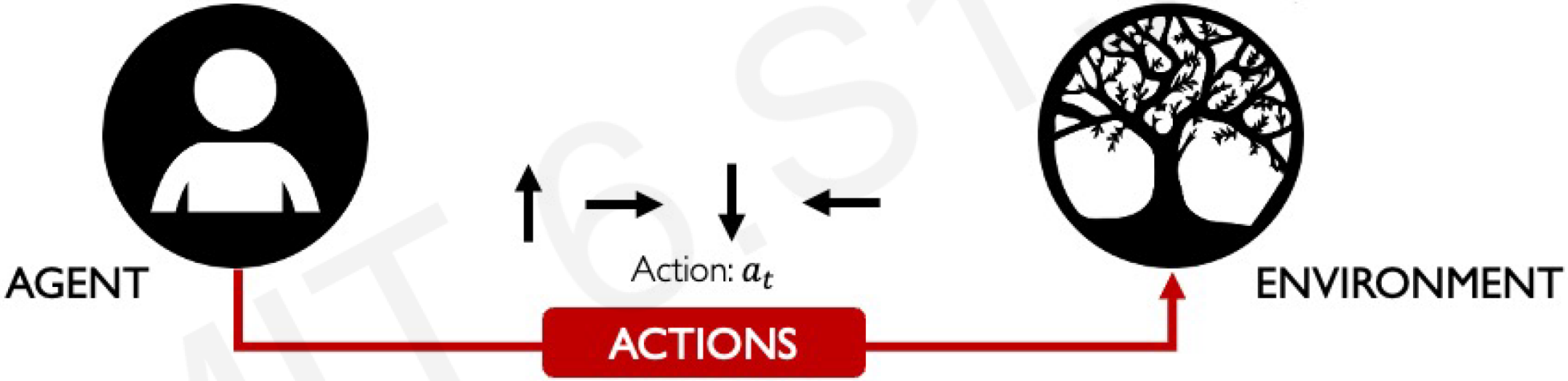
AGENT

ENVIRONMENT

**Environment**: the world in which the agent exists and operates.

MIT 6.S191

# Reinforcement Learning: Key Concepts




**Action**: a move the agent can make in the environment.

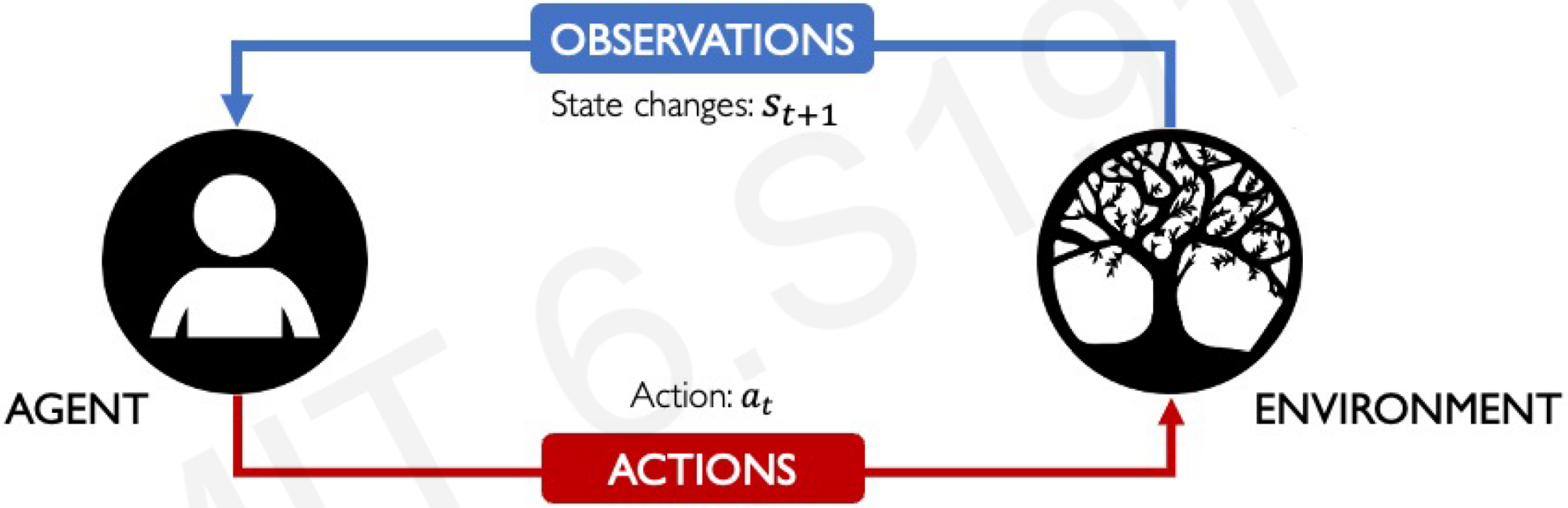
**Action space** $A$: the set of possible actions an agent can make in the environment

# Reinforcement Learning: Key Concepts


**Observations**: of the environment after taking actions.

MIT 6.S191

# Reinforcement Learning: Key Concepts




**State**: a situation which the agent perceives.

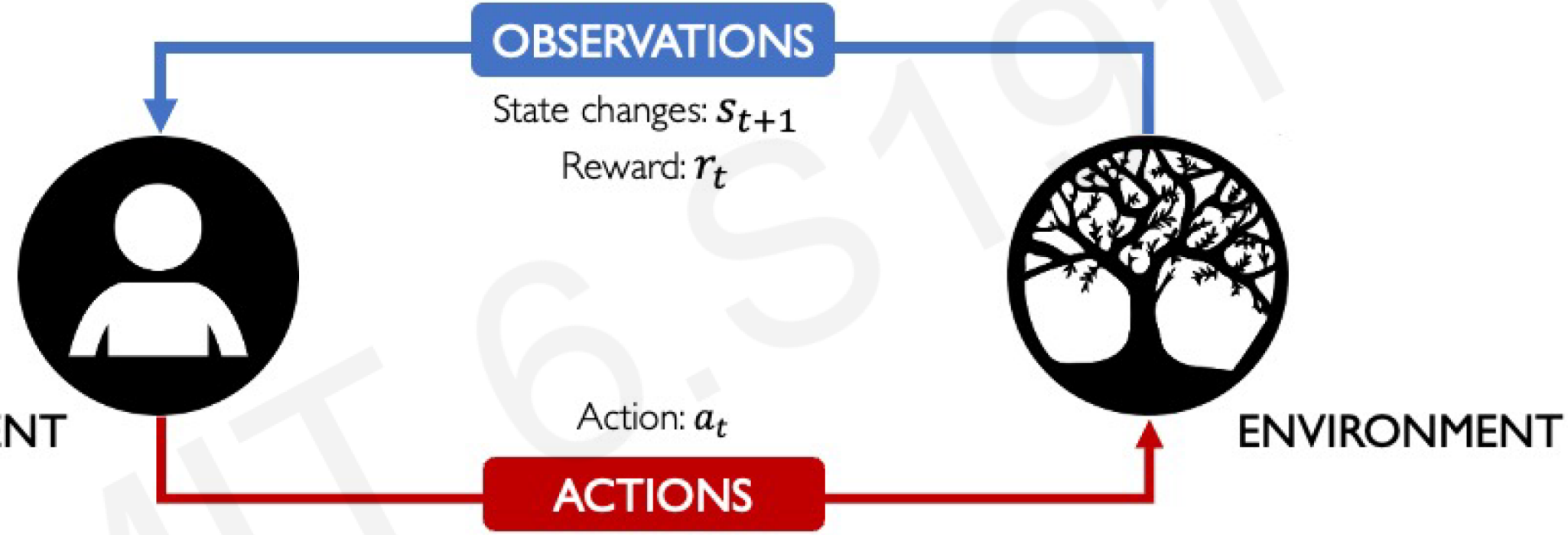
MIT 6.S191

# Reinforcement Learning: Key Concepts


**Reward**: feedback that measures the success or failure of the agent's action.

MIT 6.S191

# Reinforcement Learning: Key Concepts

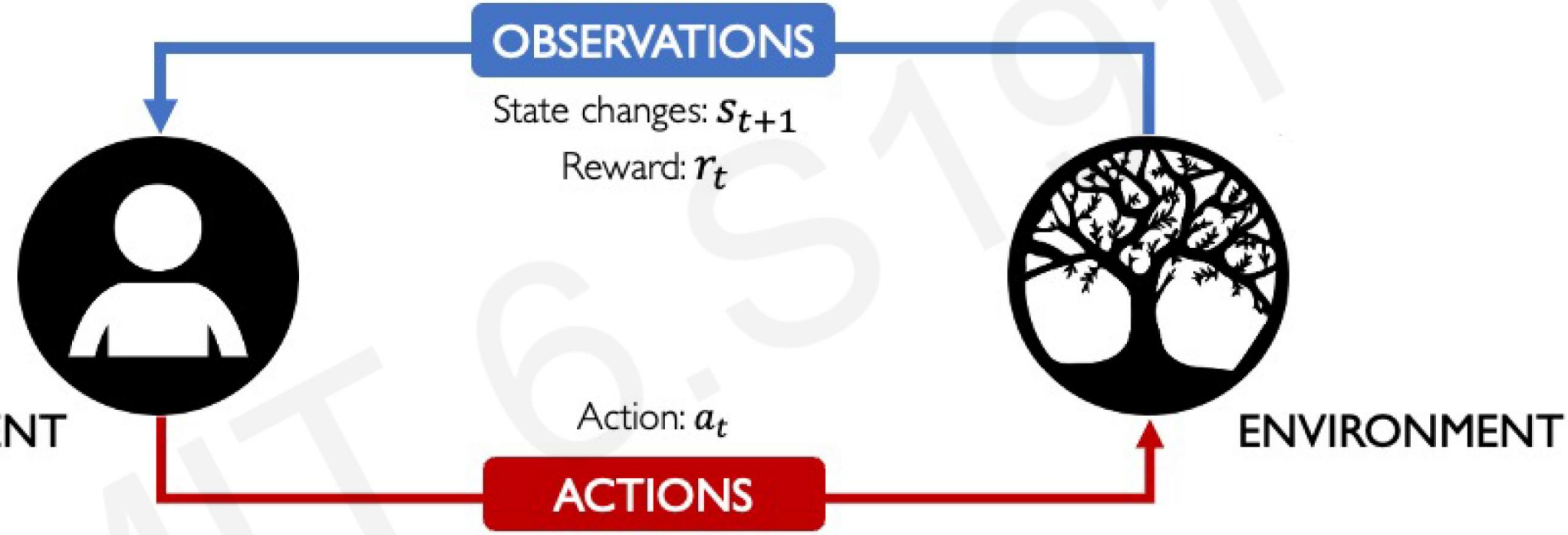



**Total Reward (Return)**

$$R_t = \sum_{i=t}^{\infty} r_i$$

MIT 6.S191

# Reinforcement Learning: Key Concepts




**Total Reward (Return)**

$$R_t = \sum_{i=t}^{\infty} r_i = r_t + r_{t+1} \ldots + r_{t+n} + \cdots$$

MIT 6.S191

# Reinforcement Learning: Key Concepts


Discounted Total Reward (Return)

$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i$$

MIT 6.S191

# Reinforcement Learning: Key Concepts


Discounted Total Reward (Return)

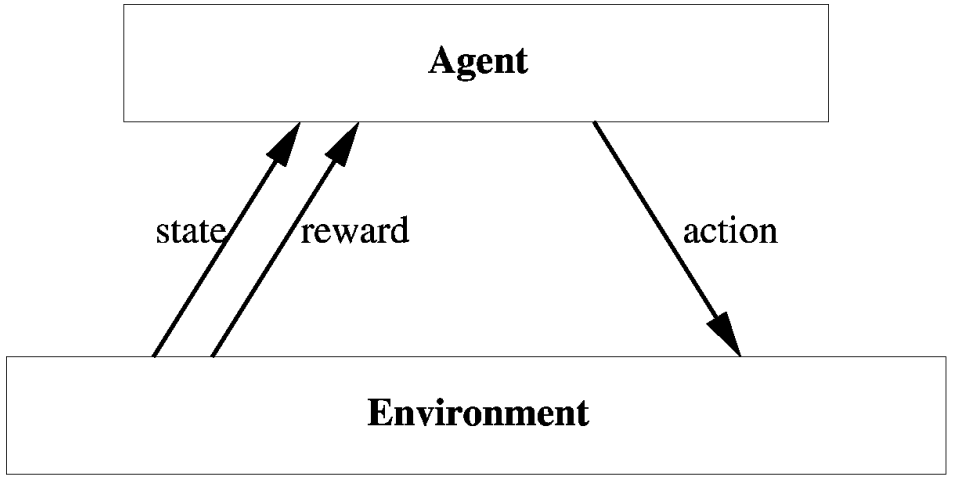
$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i = \gamma^t r_t + \gamma^{t+1} r_{t+1} \ldots + \gamma^{t+n} r_{t+n} + \cdots$$

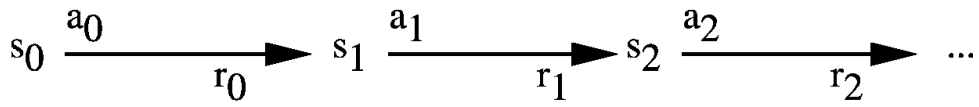
$\gamma$: discount factor; $0 < \gamma < 1$

MIT 6.S191

# A Reinforcement Learning Problem

- The environment
- The reward function *r(s,a)*
  - Pure delay reward and avoidance problems
  - Minimum time to goal
  - Games
- The value function *V(s)*
  - Policy $\pi$: $S \rightarrow A$
  - Value $V^{\pi}(s) := \Sigma_i \gamma^i r_{t+i}$
- Find the optimal policy $\pi^*$ that maximizes $V^{\pi^*}(s)$ for all states *s*.



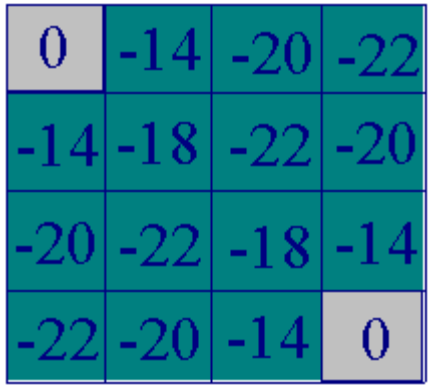




Goal: Learn to choose actions that maximize

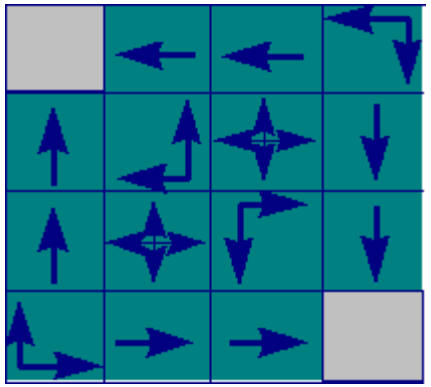
$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$ , where $0<\gamma<1$

# RL Value Function - Example

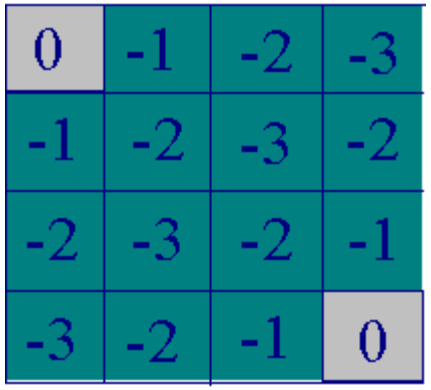
## A minimum time to goal world




Value function for random movement



Optimal policy



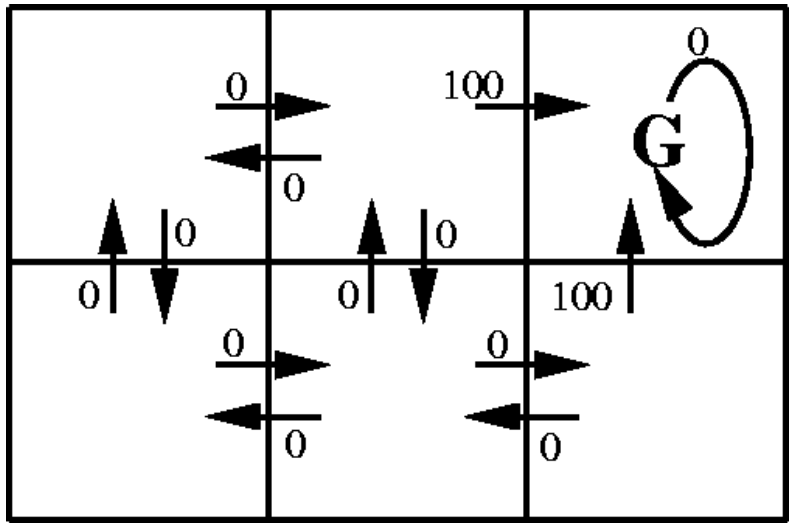

Optimal value function

# Markov Decision Processes

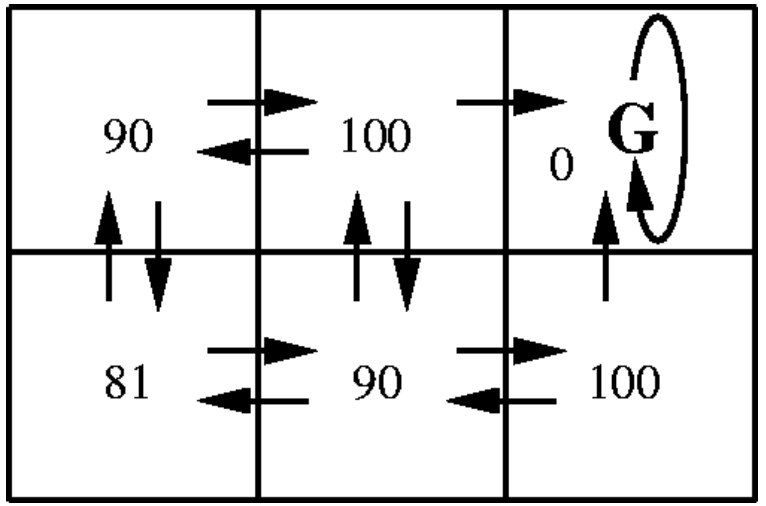
Assume:

- finite set of states $S$, finite set of actions $A$
- at each discrete time agent observes state $s_t \in S$ and chooses action $a_t \in A$
- then receives immediate reward $r_t$
- and state changes to $s_{t+1}$
- Markov assumption: $s_{t+1} = \delta(s_t, a_t)$ and $r_t = r(s_t, a_t)$
  - i.e. $r_t$ and $s_{t+1}$ depend only on current state and action
  - functions $\delta$ and $r$ may be non-deterministic
  - functions $\delta$ and $r$ not necessarily known to the agent

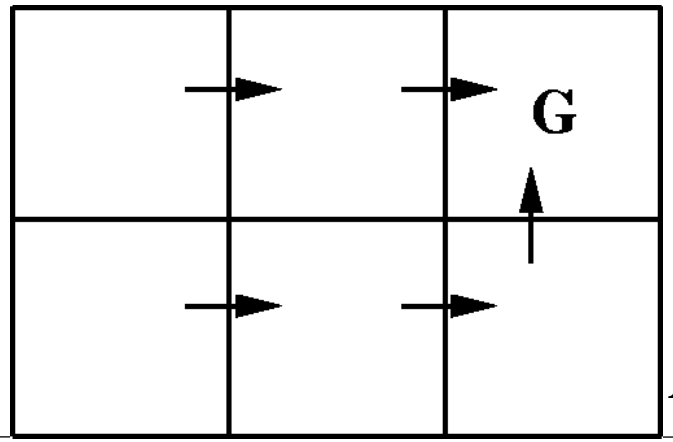
# MDP Example




*r(s,a)*




*V*(s)*




An optimal policy

# Defining the Q-Function

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

Total reward, $R_t$, is the discounted sum of all rewards obtained from time $t$

$$Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$$

The Q-function captures the **expected total future reward** an agent in state, $s$, can receive by executing a certain action, $a$

# How to Take Actions Given a Q-Function

$$Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$$

(state, action)

Ultimately, the agent needs a **policy $\pi(s)$**, to infer the **best action to take** at its state, s

**Strategy:** the policy should choose an action that maximizes future reward

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

MIT 6.S191

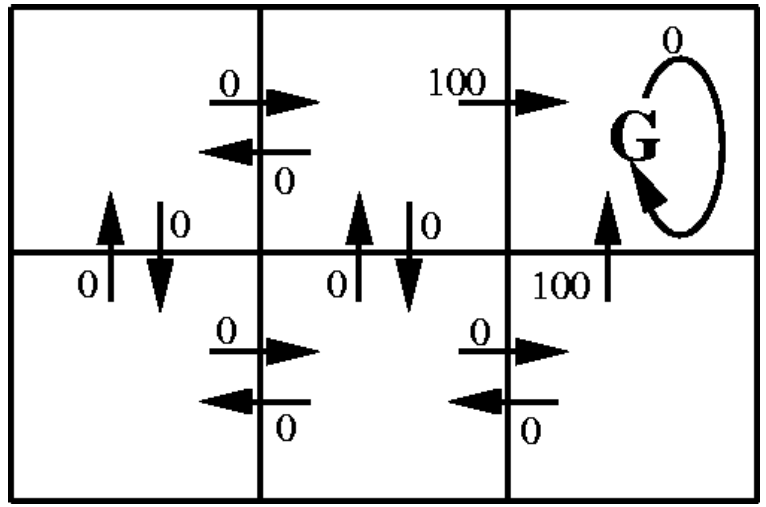
# The Q-Function

Optimal policy:

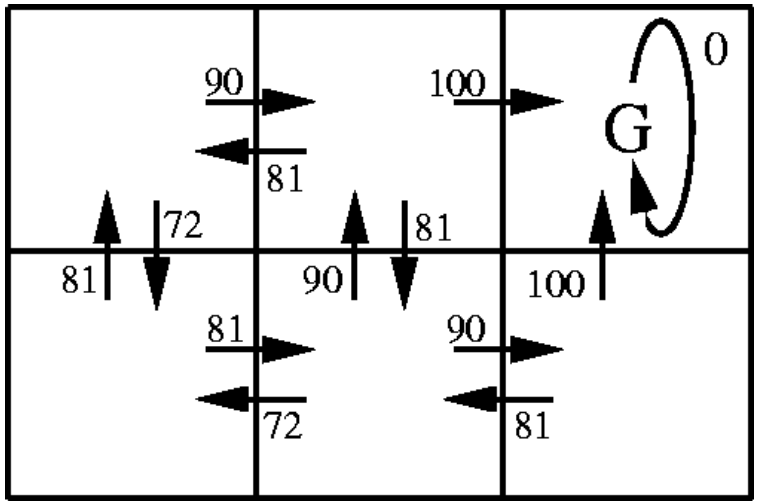
- $\pi^*(s) = \text{argmax}_a[r(s,a) + \gamma V^*(\delta(s,a))]$
- Doesn't work if we don't know $r$ and $\delta$.

The Q-function:

- $Q(s,a) := r(s,a) + \gamma V^*(\delta(s,a))$
- $\pi^*(s) = \text{argmax}_a Q(s,a)$




*r(s,a)*




*Q(s,a)*

# The Q-Function

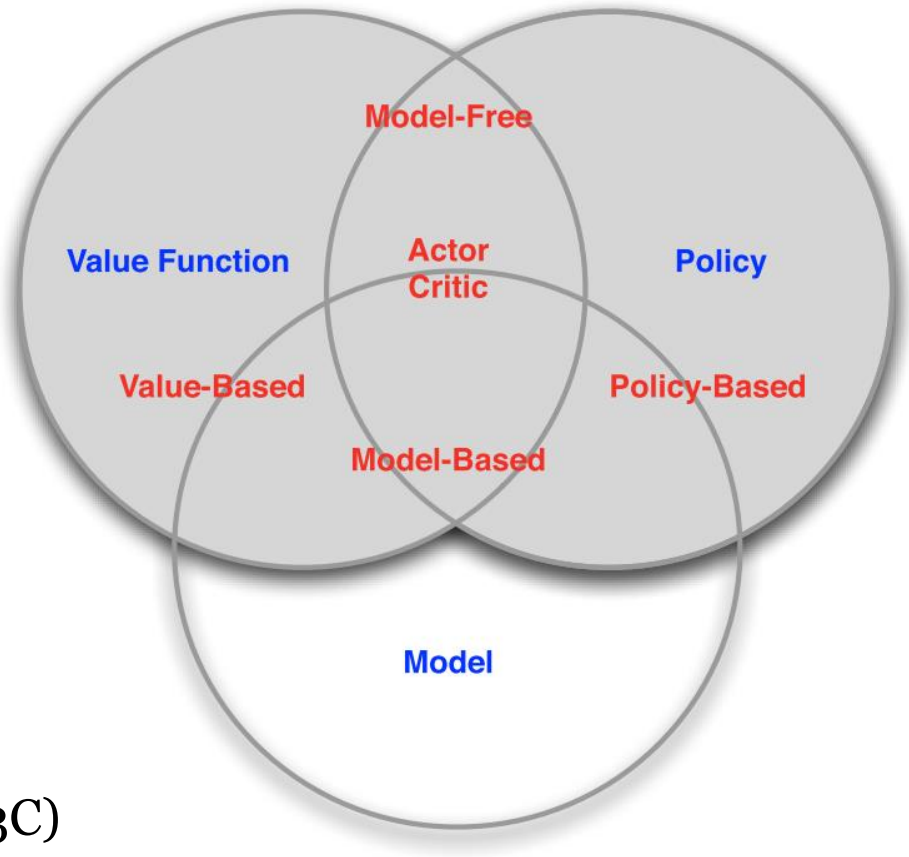
- Note Q and V* closely related:
  $V^*(s) = \max_{a'} Q(s,a')$
- Therefore Q can be written as:

  $$Q(s_t,a_t) := r(s_t,a_t) + \gamma V^*(\delta(s_t,a_t)) =$$
  $$r(s_t,a_t) + \gamma \max_{a'} Q(s_{t+1},a')$$
- If $\hat{Q}$ denote the current approximation of Q then it can be updated by:

  $\hat{Q}(s,a) := r + \gamma \max_{a'} \hat{Q}(s',a')$

# Reinforcement Learning Approaches

- Value-Based:
  - Learn value function
  - Implicit policy (e.g. greedy selection)
  - Example: Deep Q Networks (DQN)
- Policy-Based:
  - No value function
  - Learn explicit (stochastic) policy
  - Example: Stochastic Policy Gradients
- Model-Based:
  - Learn transition model
  - Implicit policy
  - Example: Dreamer
- Actor-Critic:
  - Learn value function
  - Learn policy using value function
  - Example: Asynchronous Advantage Actor Critic (A3C)



# Reinforcement Learning Algorithms

## Value Learning

Find $Q(s, a)$

$$a = \underset{a}{\operatorname{argmax}}\, Q(s, a)$$

## Policy Learning

Find $\pi(s)$

Sample $a \sim \pi(s)$

# Reinforcement Learning Algorithms

## Value Learning

Find $Q(s, a)$

$$a = \underset{a}{\text{argmax}}\, Q(s, a)$$

## Policy Learning

Find $\pi(s)$

Sample $a \sim \pi(s)$

# Q-Learning for Deterministic Worlds

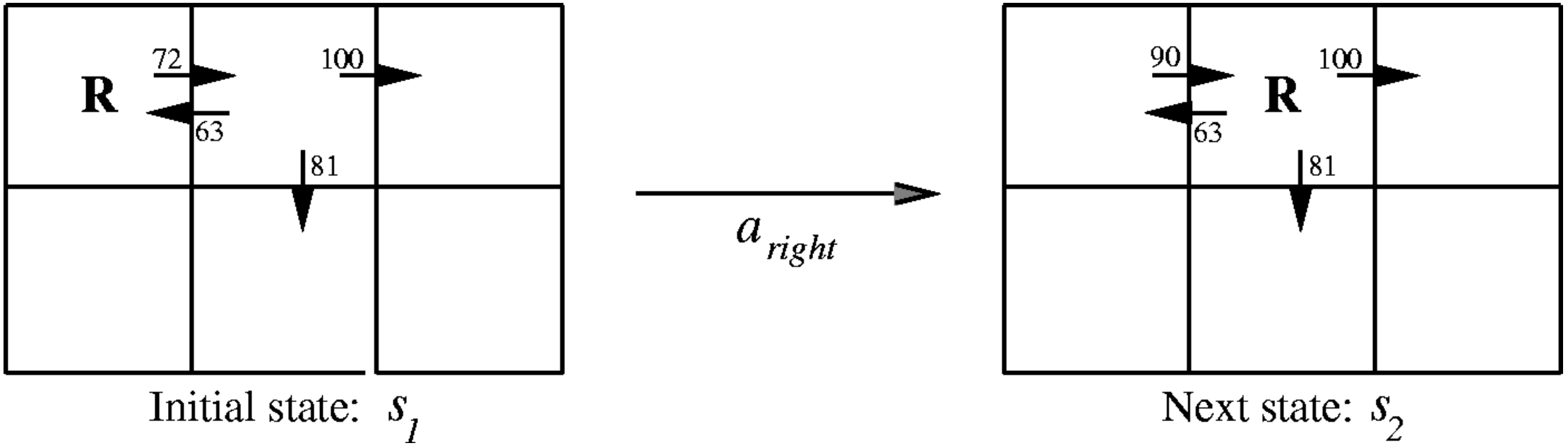
For each $s$, $a$ initialize table entry $\hat{Q}(s,a) := 0$.

Observe current state $s$.

Do forever:

1. Select an action $a$ and execute it
2. Receive immediate reward $r$
3. Observe the new state $s'$
4. Update the table entry for $\hat{Q}(s,a)$:
   $\hat{Q}(s,a) := r + \gamma \max_{a'} \hat{Q}(s',a')$
5. $s := s'$

# Q-Learning Example



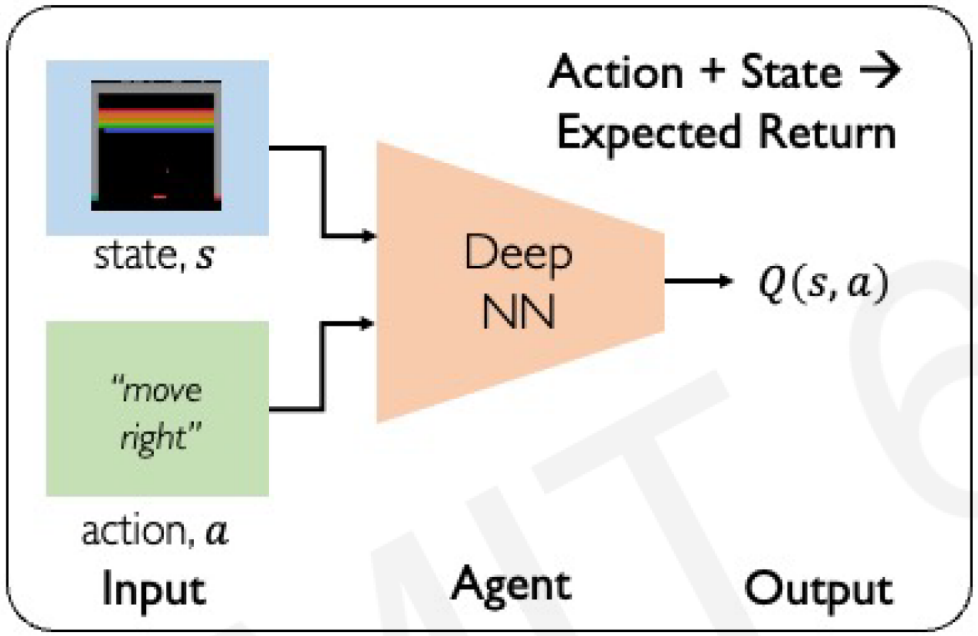

$$Q\,\hat{}\,(s_1, a_{right}) := r + \gamma \max_{a'} Q\,\hat{}\,(s_2, a')$$
$$:= 0 + 0.9 \max\{63, 81, 100\}$$
$$:= 90$$

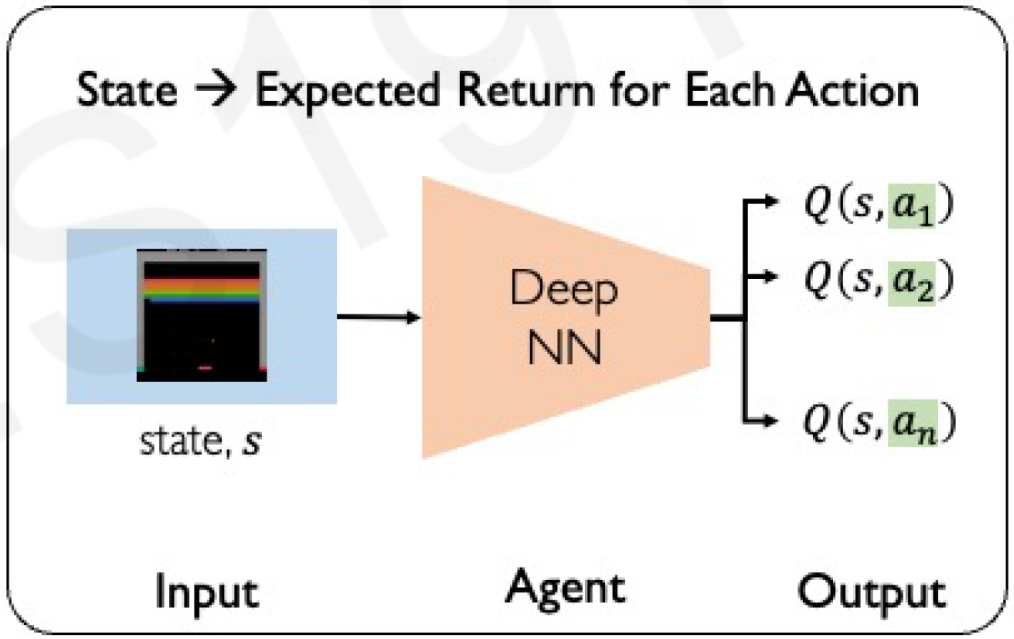
# Q-Learning Continued

- Exploration
  - Selecting the best action
  - Probabilistic choice
- Improving convergence
  - Update sequences
  - Remember old state-action transitions and their immediate reward
- Non-deterministic MDPs
- Temporal Difference Learning

# Deep Q-Learning (DQN)

How can we use deep neural networks to model Q-functions?

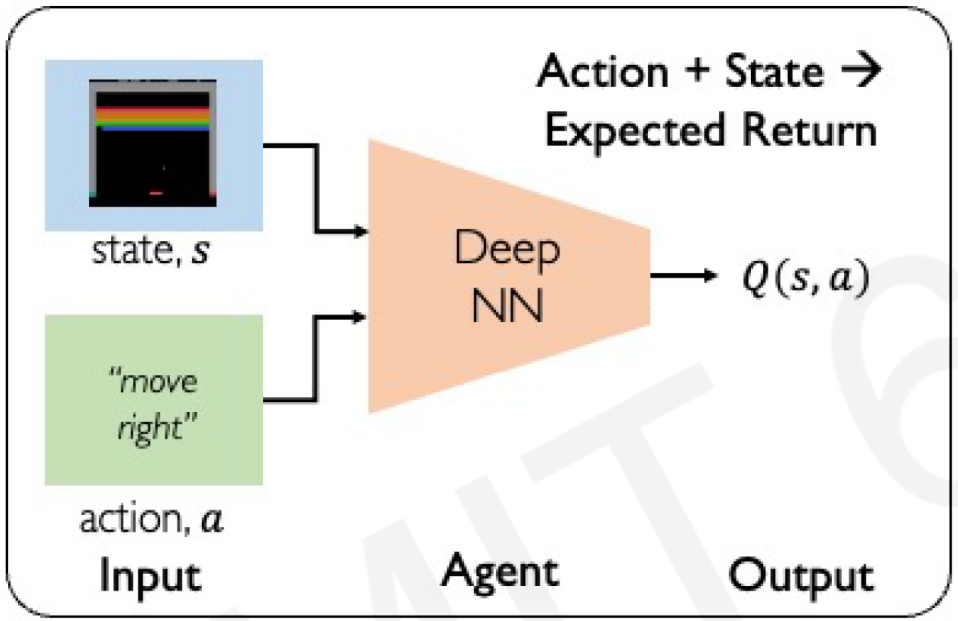


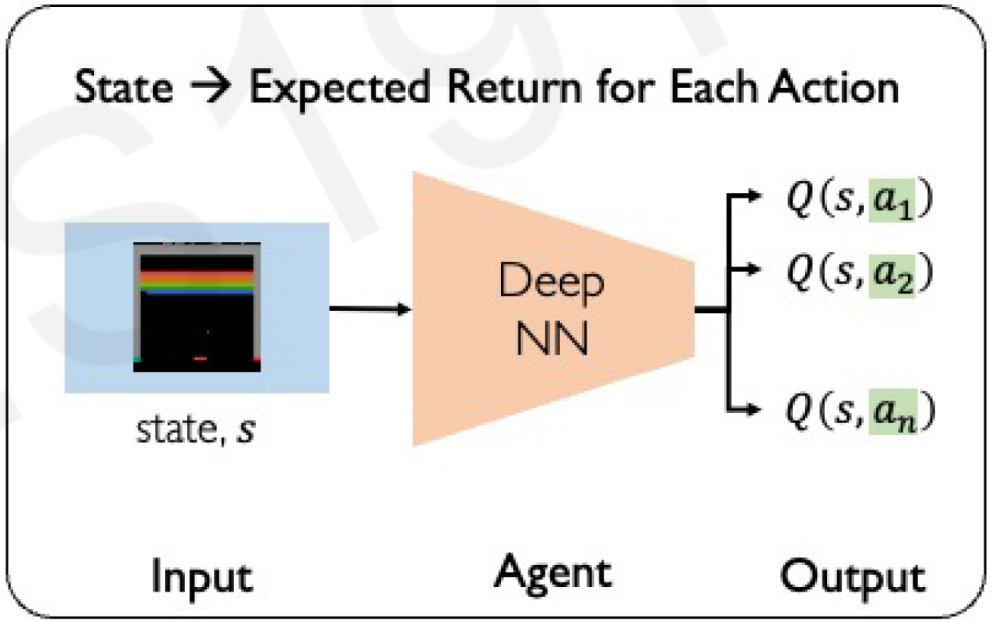




MIT 6.S191

# Deep Q Networks (DQN): Training

How can we use deep neural networks to model Q-functions?



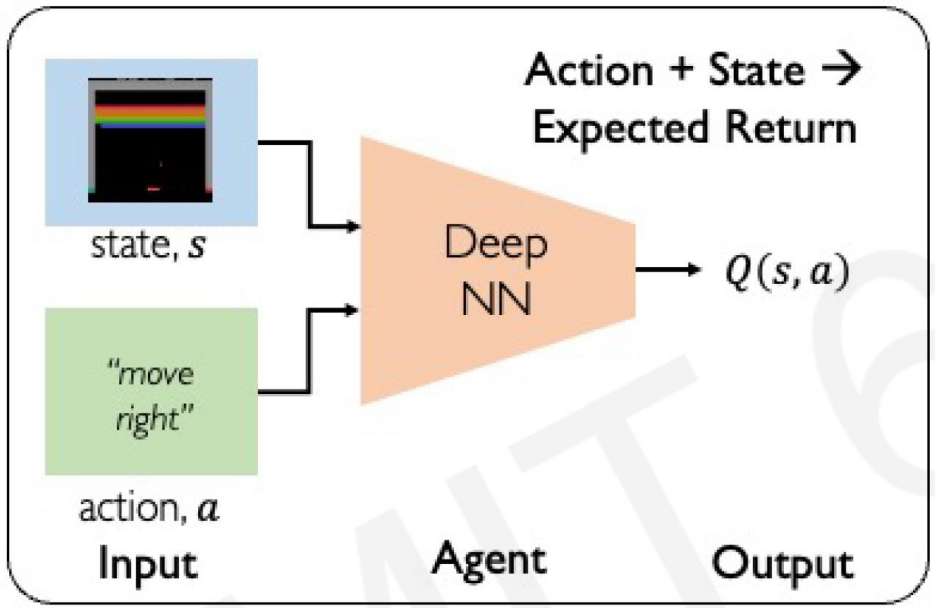



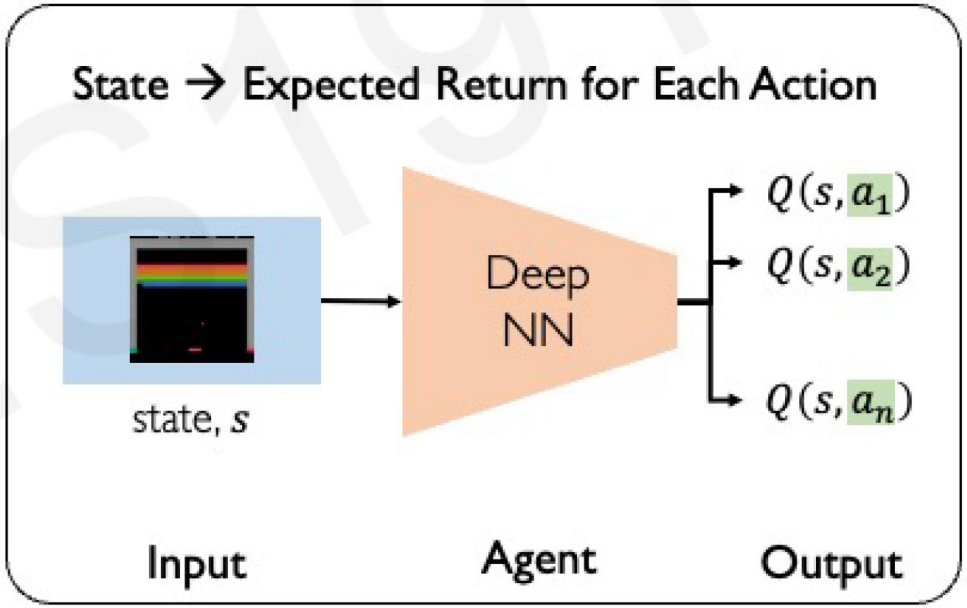

**What happens if we take all the best actions?**
***Maximize target return → train the agent***

MIT 6.S191

# Deep Q Networks (DQN): Training

How can we use deep neural networks to model Q-functions?



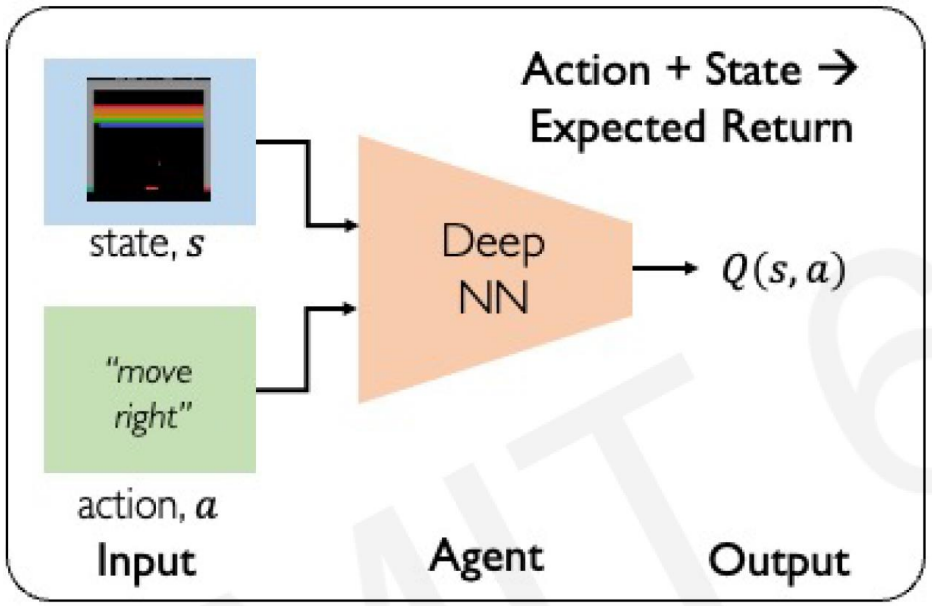

$$\underbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}_{\text{target}}$$

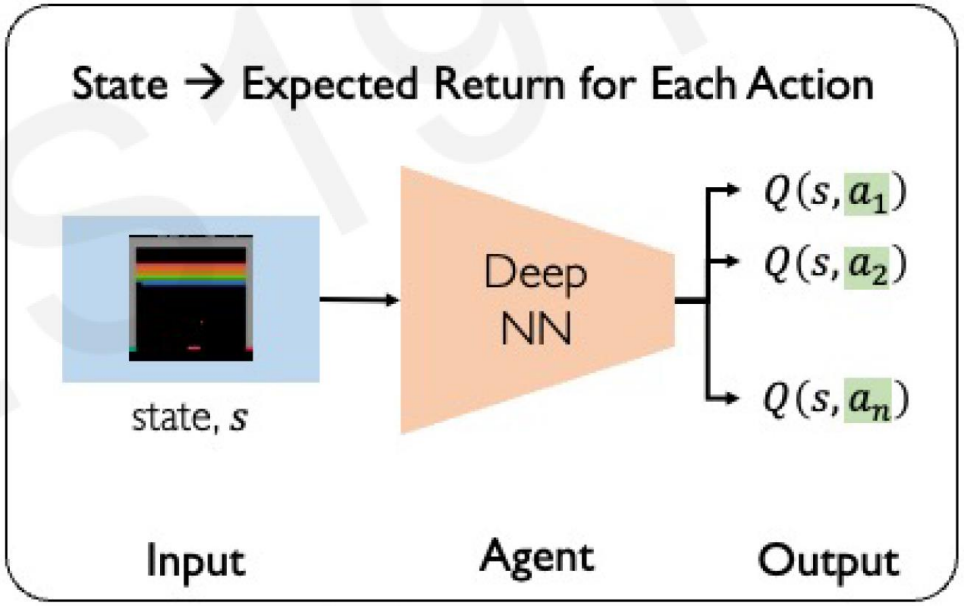
Take all the best actions → target return

MIT 6.S191

# Deep Q Networks (DQN): Training

How can we use deep neural networks to model Q-functions?

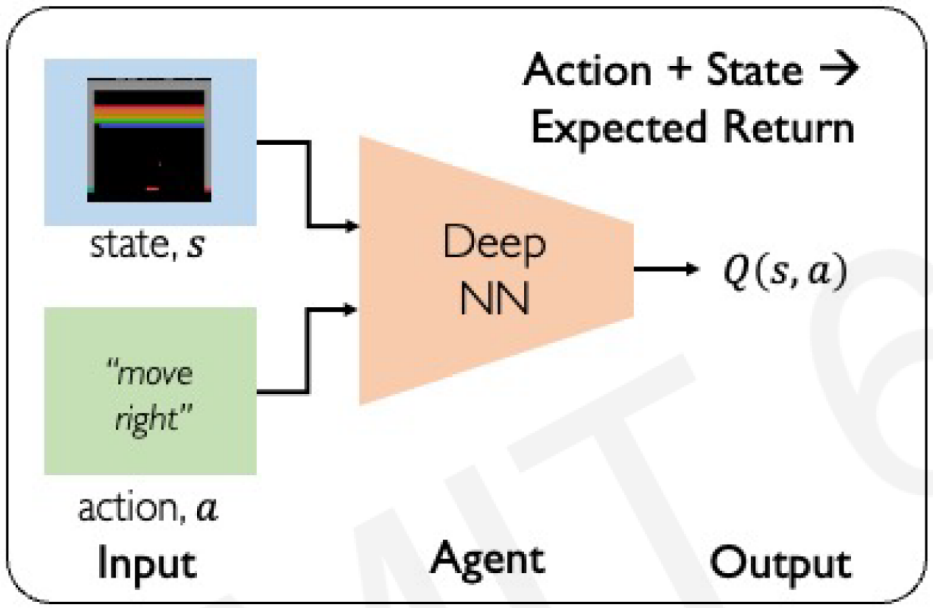


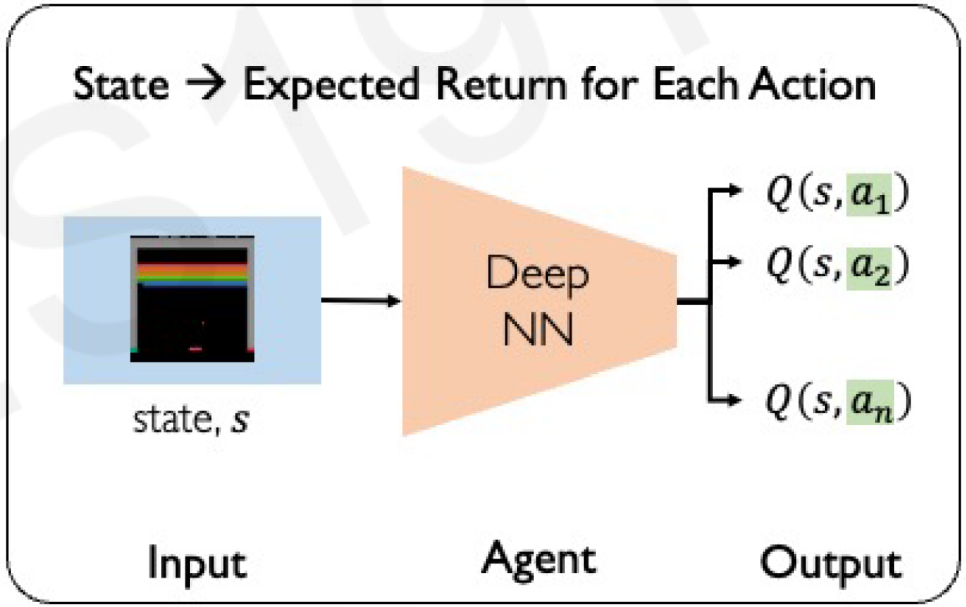




$$\underbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}_{\text{target}} \qquad \underbrace{Q(s,a)}_{\text{predicted}}$$

# Deep Q Networks (DQN): Training

How can we use deep neural networks to model Q-functions?



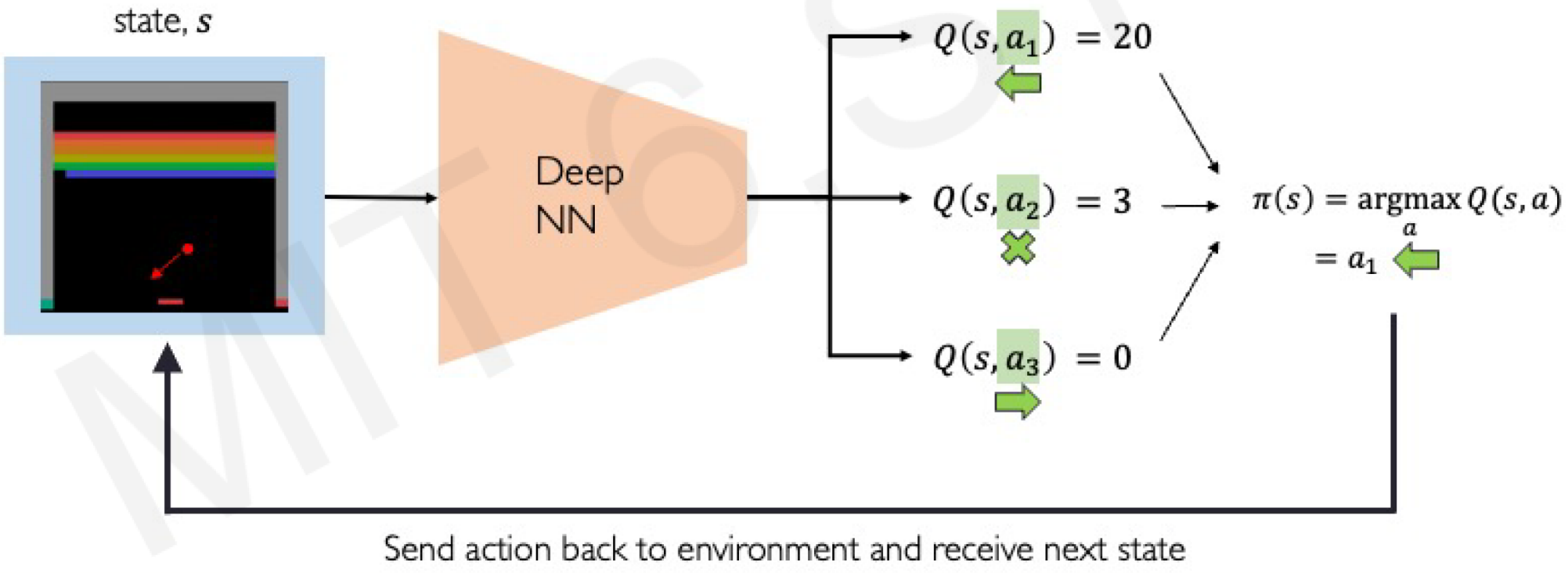




$$\mathcal{L} = \mathbb{E}\left[\left\|\overbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}^{\text{target}} - \overbrace{Q(s,a)}^{\text{predicted}}\right\|^2\right] \quad \textbf{Q-Loss}$$

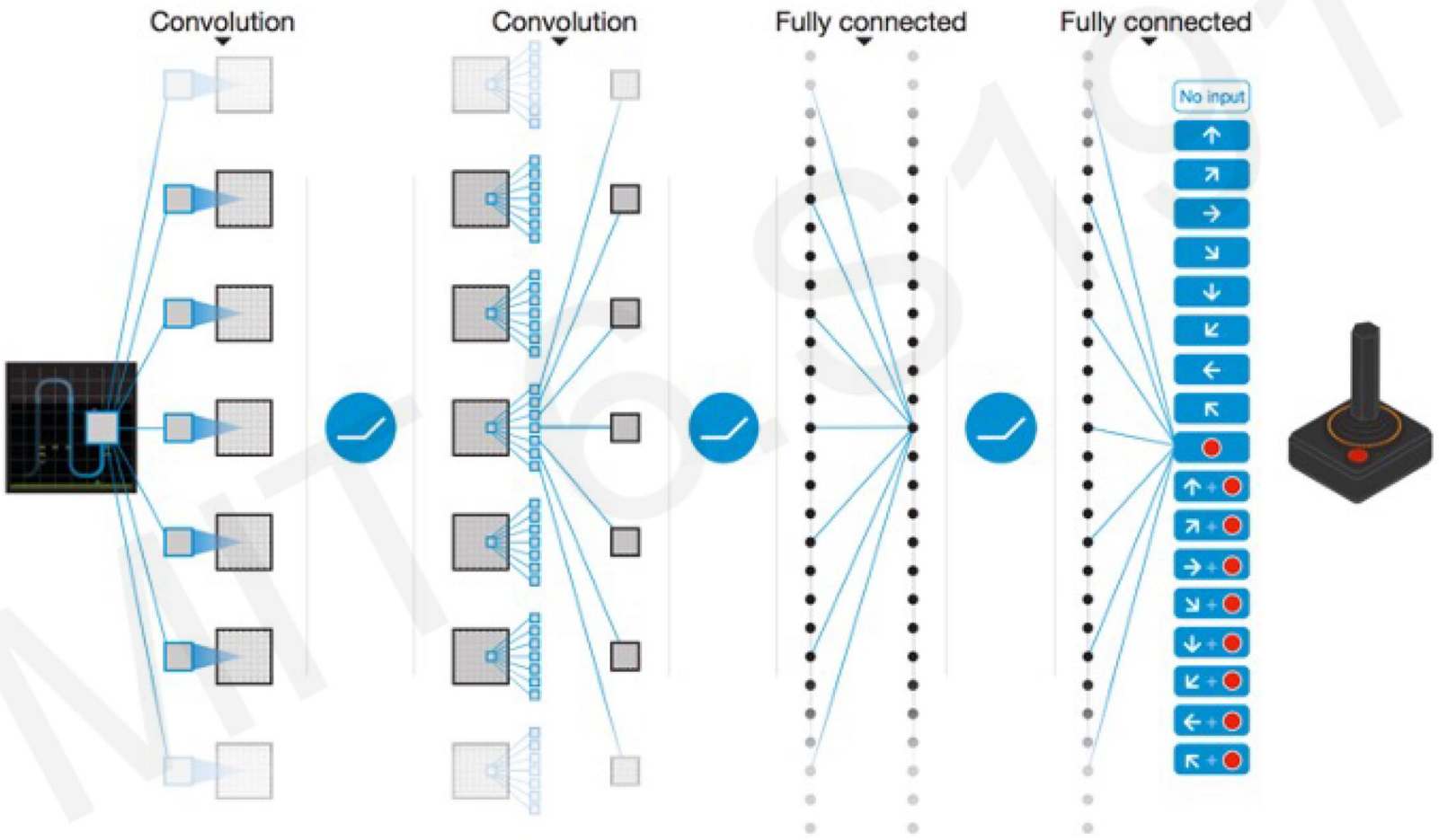
MIT 6.S191

# Deep Q Network Summary

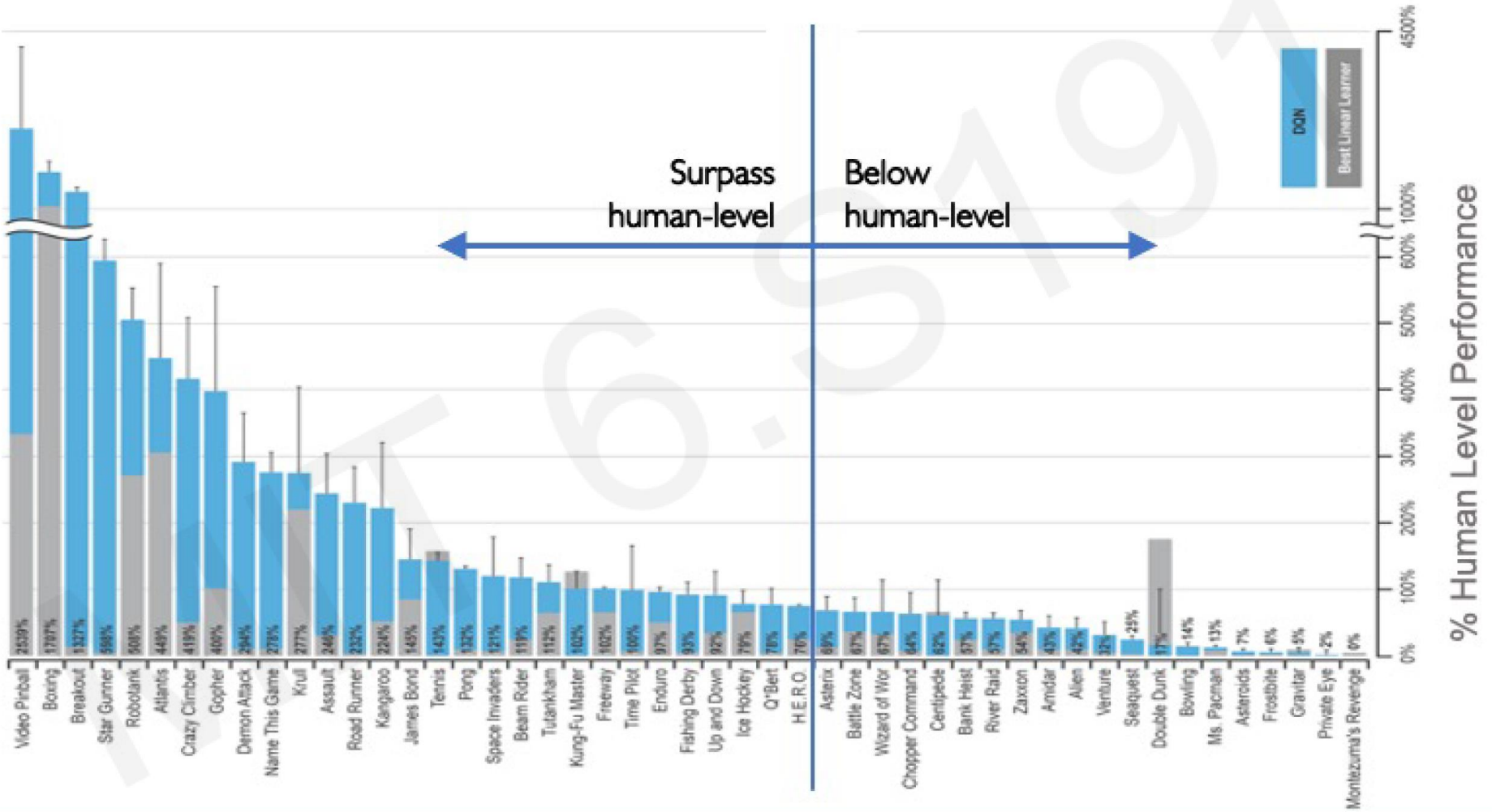
Use NN to learn Q-function and then use to infer the optimal policy, $\pi(s)$



# DQN Atari Results



# DQN Atari Results



# Rainbow DQN

- DQN - baseline
- Double DQN - de-overestimate values

$$(R_{t+1}+\gamma_{t+1}q_{\overline{\theta}}(S_{t+1}, \operatorname{argmax}_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2$$

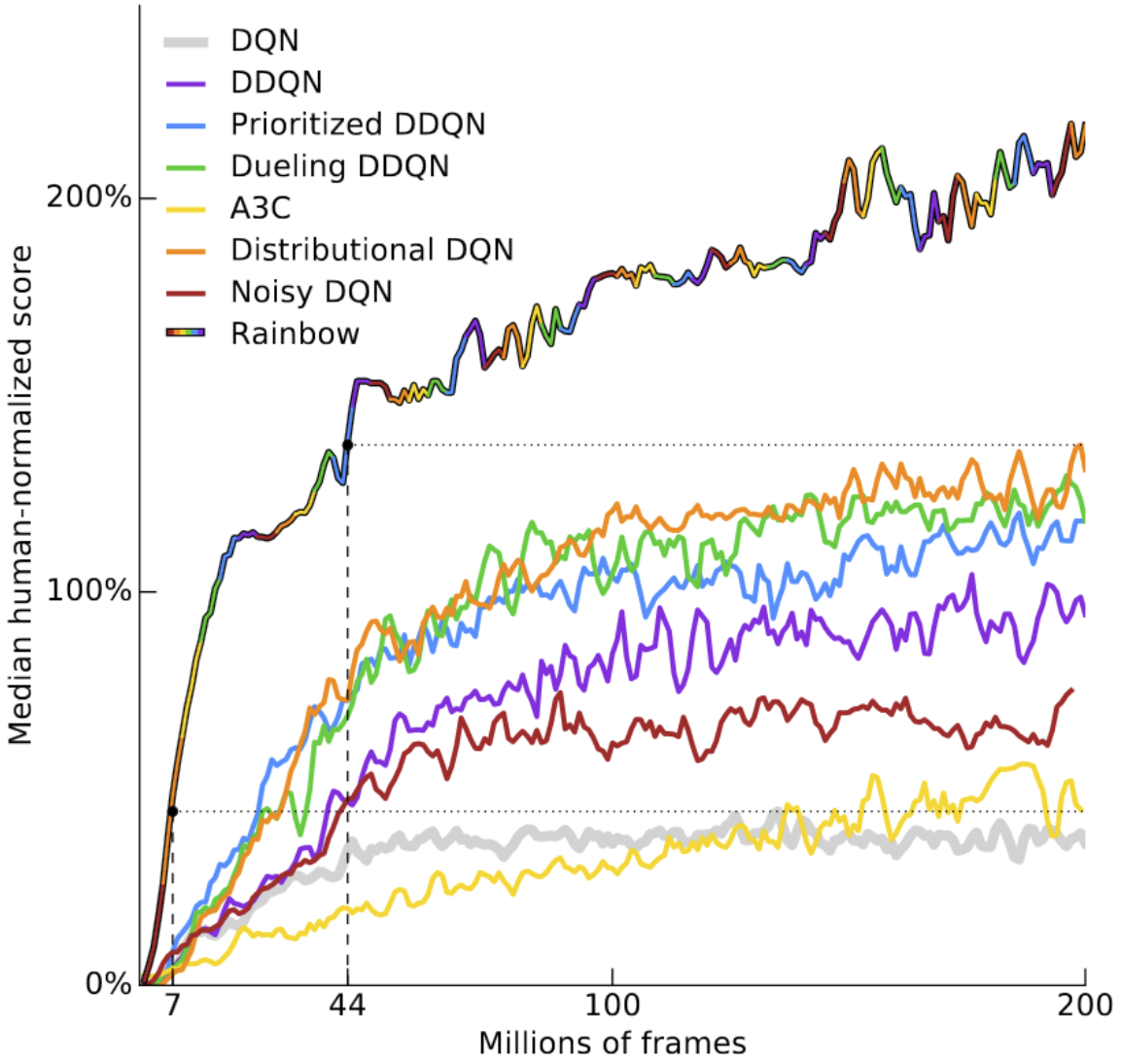
- Prioritized experience

$$p_t \propto \left| R_{t+1} + \gamma_{t+1} \max_{a'} q_{\overline{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t) \right|^\omega$$

- Dueling networks

$$q_\theta(s,a) = v_\eta(f_\xi(s)) + a_\psi(f_\xi(s), a) - \frac{\sum_{a'} a_\psi(f_\xi(s), a')}{N_{\text{actions}}}$$

- Distributional DQN - probability distribution
- Noisy DQN - parametric noise
- -> **ADDITIVE**



# Downsides of Q-Learning

**Complexity:**

- Can model scenarios where the action space is discrete and small
- Cannot handle continuous action spaces

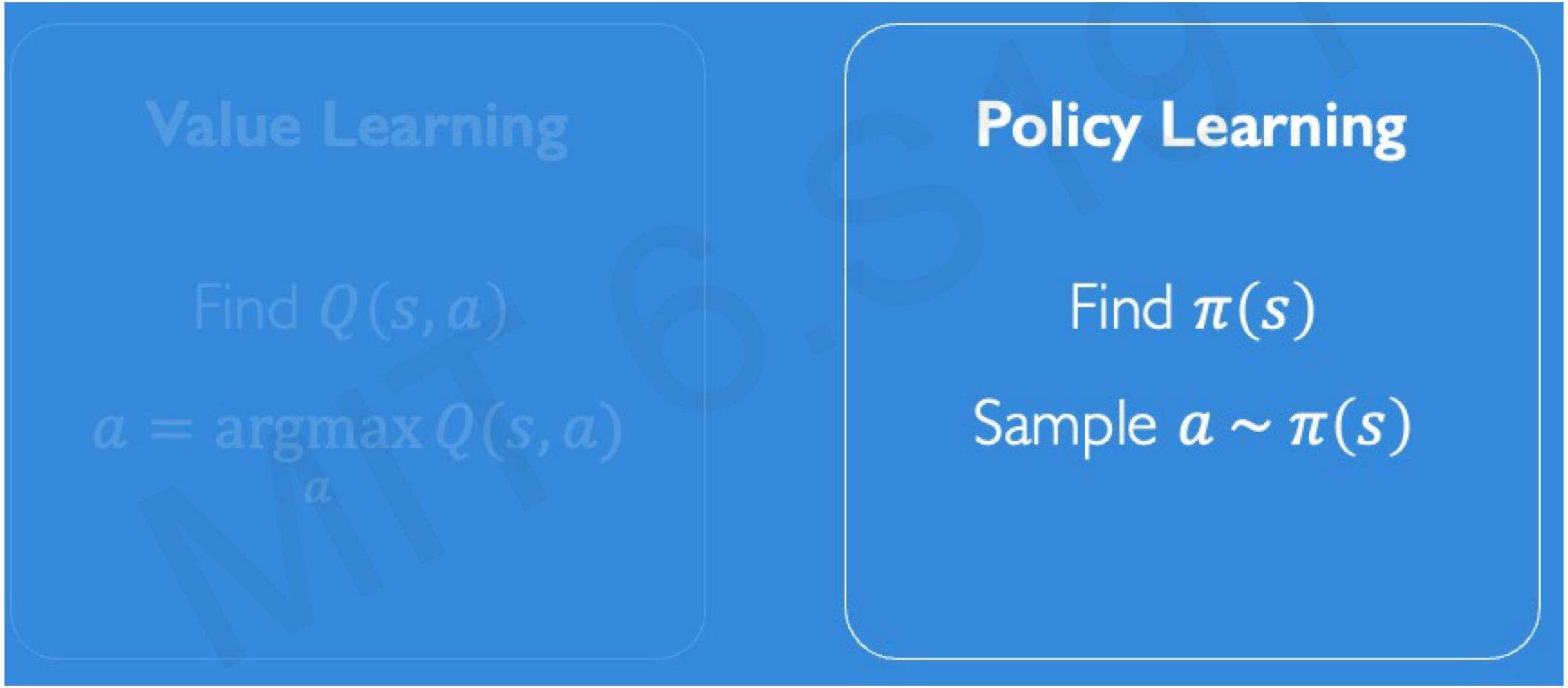
**Flexibility:**

- Policy is deterministically computed from the Q function by maximizing the reward → cannot learn stochastic policies

**To address these, consider a new class of RL training algorithms: Policy gradient methods**

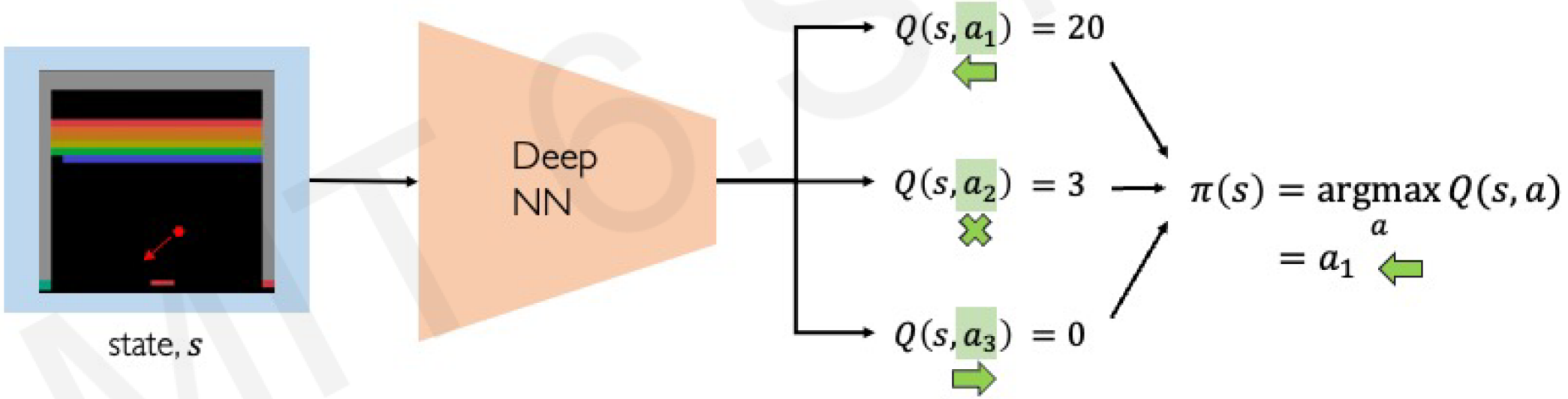
MIT 6.S191

# Reinforcement Learning Algorithms



# Deep Q Networks

**DQN:** Approximate Q-function and use to infer the optimal policy, $\pi(s)$

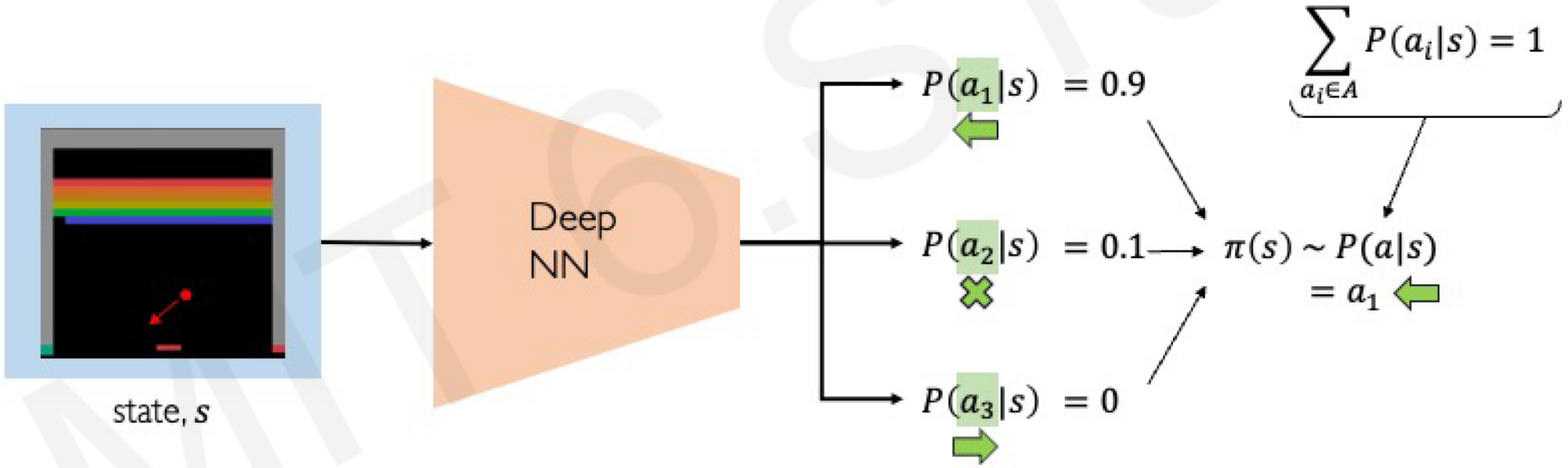



MIT 6.S191

# Policy Gradient (PG): Key Idea

**DQN:** Approximate Q-function and use to infer the optimal policy, $\pi(s)$

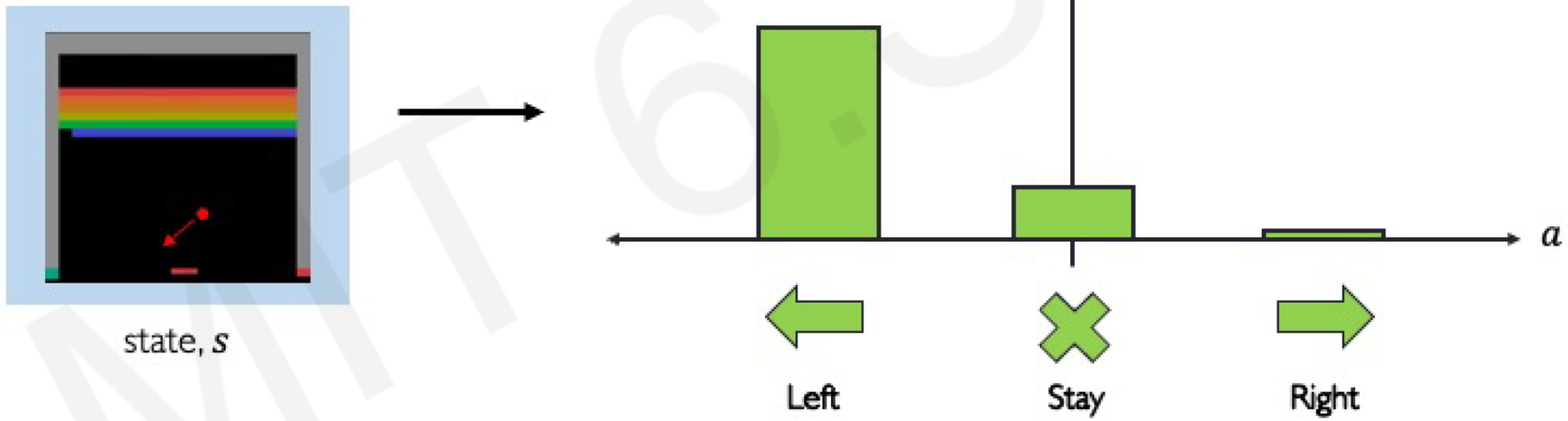
**Policy Gradient:** Directly optimize the policy $\pi(s)$




What are some advantages of this formulation?

# Discrete vs Continuous Action Spaces

**Discrete action space:** which direction should I move?

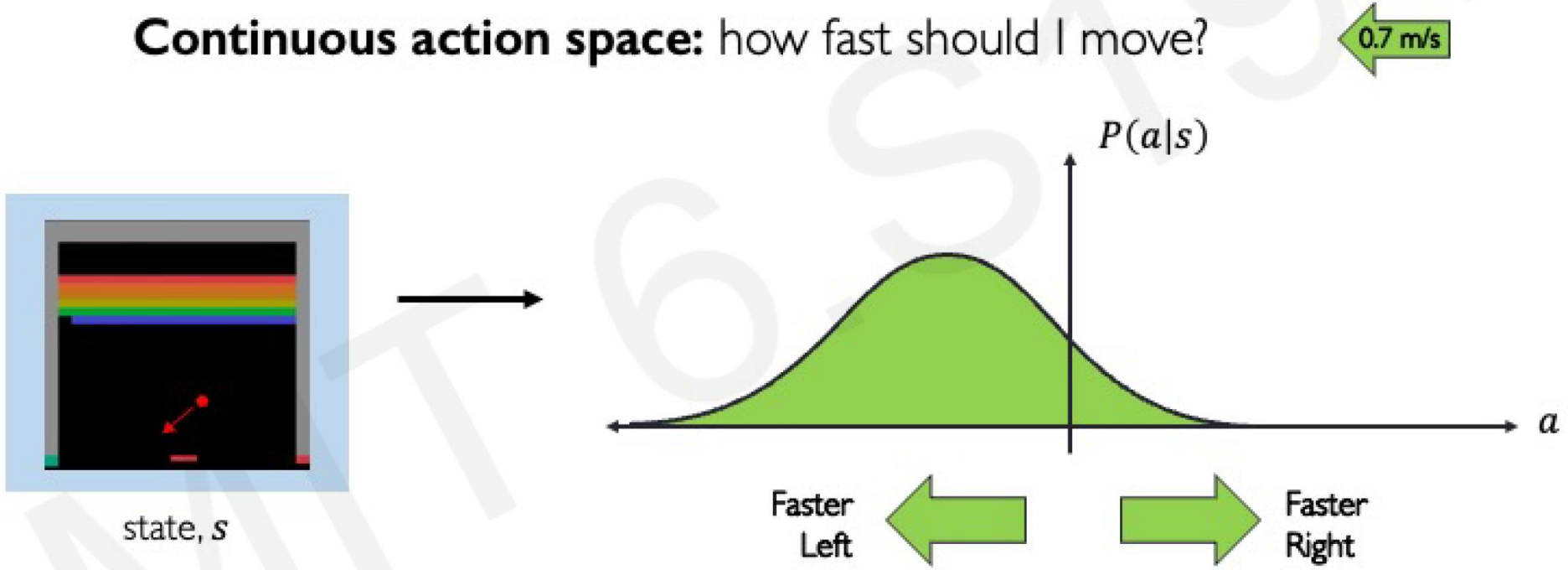



MIT 6.S191

# Discrete vs Continuous Action Spaces

**Discrete action space:** which direction should I move?

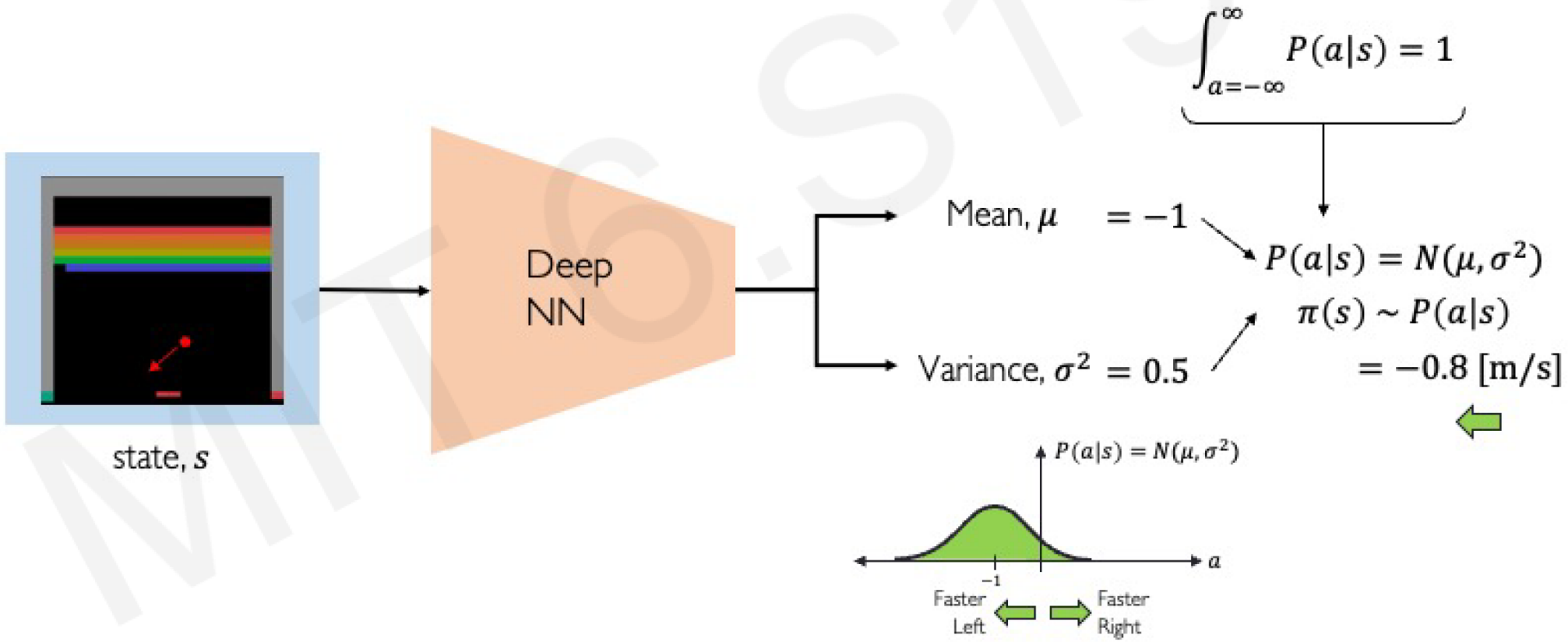
**Continuous action space:** how fast should I move?




MIT 6.S191

# Policy Gradient (PG): Key Idea

**Policy Gradient:** Enables modeling of continuous action space




MIT 6.S191

# Training Policy Gradients: Case Study

Reinforcement Learning Loop:


Case Study – Self-Driving Cars

**Agent:** vehicle

**State:** camera, lidar, etc

**Action:** steering wheel angle

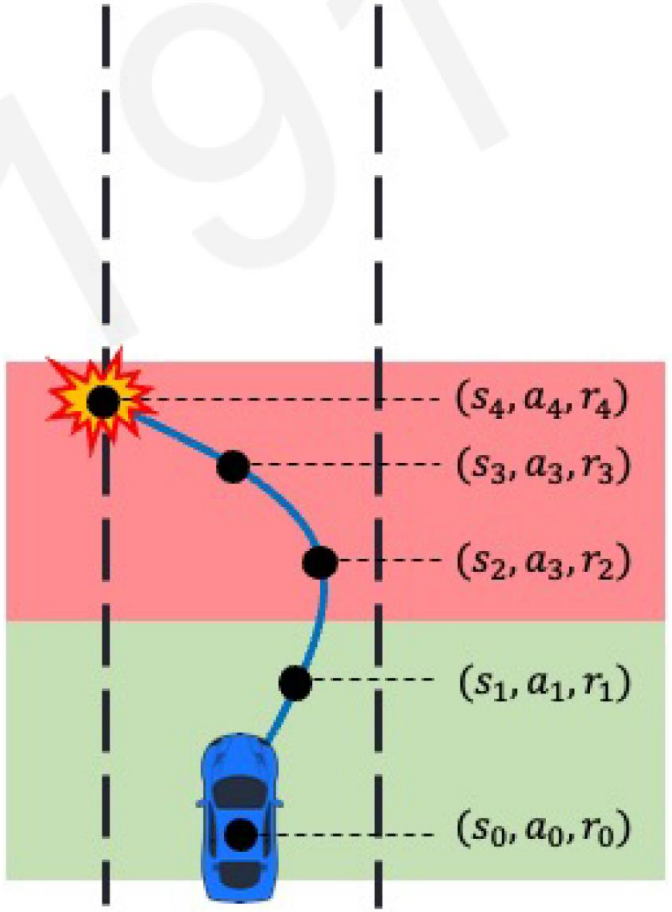
**Reward:** distance traveled

MIT 6S191

# Training Policy Gradients

**Training Algorithm**

1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward

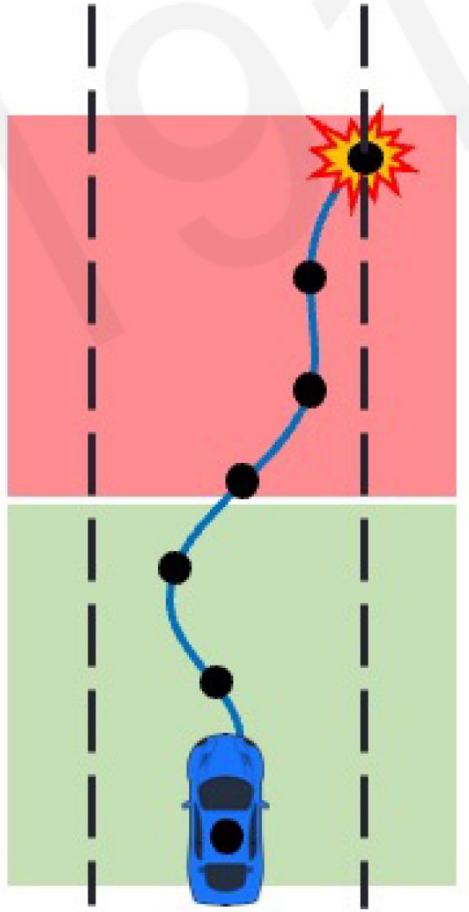



MIT 6.S191

# Training Policy Gradients

**Training Algorithm**

1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward

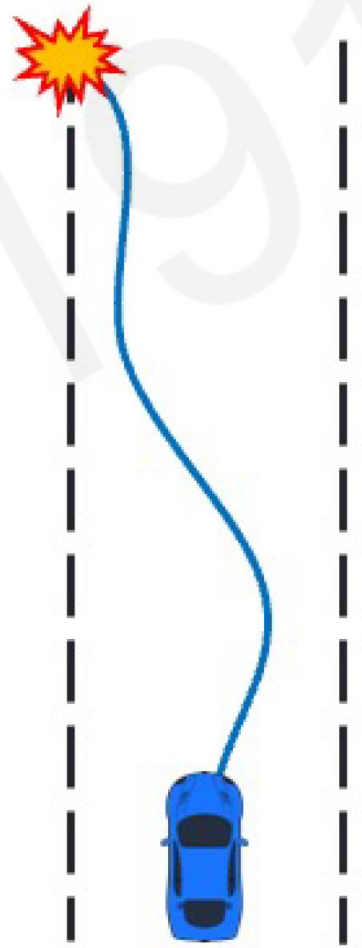


MIT 6.S191

# Training Policy Gradients

**Training Algorithm**

1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward

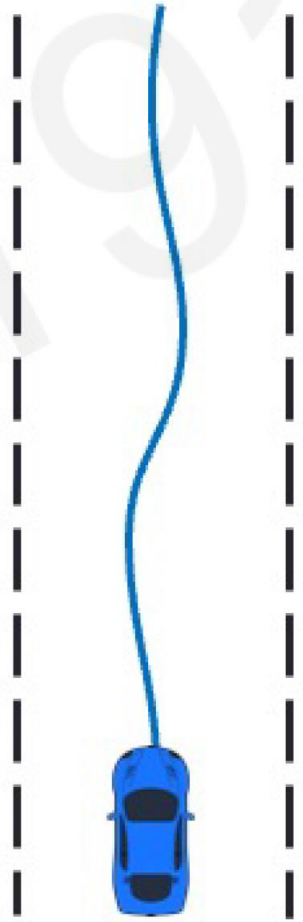


MIT 6.S191

# Training Policy Gradients

**Training Algorithm**

1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



MIT 6S191

# Training Policy Gradients

**Training Algorithm**

1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward

log-likelihood of action

$$\textbf{loss} = -\log \mathrm{P}(a_t|s_t)\, R_t$$

reward

**Gradient descent update:**

$$w' = w - \nabla \textbf{loss}$$

$$w' = w + \nabla \log \mathrm{P}(a_t|s_t)\, R_t$$

Policy gradient!

MIT 6S191

# REINFORCE

- Take parameterized policy $\pi_{\theta o}$
- Sample an episode $\tau$ with parameters $\theta_1$
- If it is better, then push parameters in that direction
- If not, then push parameters the other way
- (aka: vanilla policy gradient)

# Policy-Gradient Theorem

$$Policy\ gradient : E_\pi[\nabla_\theta(log\pi(s,a,\theta))R(\tau)]$$

Policy function — Score function

$$Update\ rule : \Delta\theta = \alpha * \nabla_\theta(log\pi(s,a,\theta))R(\tau)$$

Change in parameters — Learning rate

# REINFORCE

**function REINFORCE**

Initialise $\theta$ arbitrarily

**for** each episode $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**

**for** $t = 1$ to $T - 1$ **do**

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

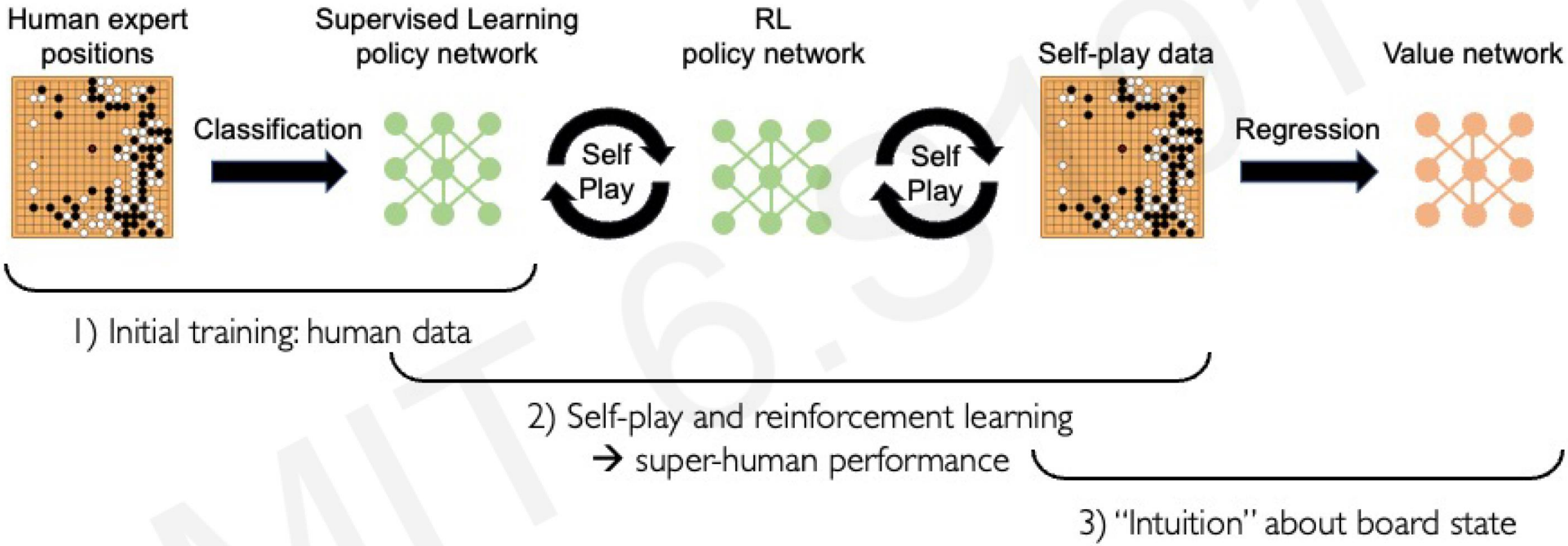
**end for**

**end for**

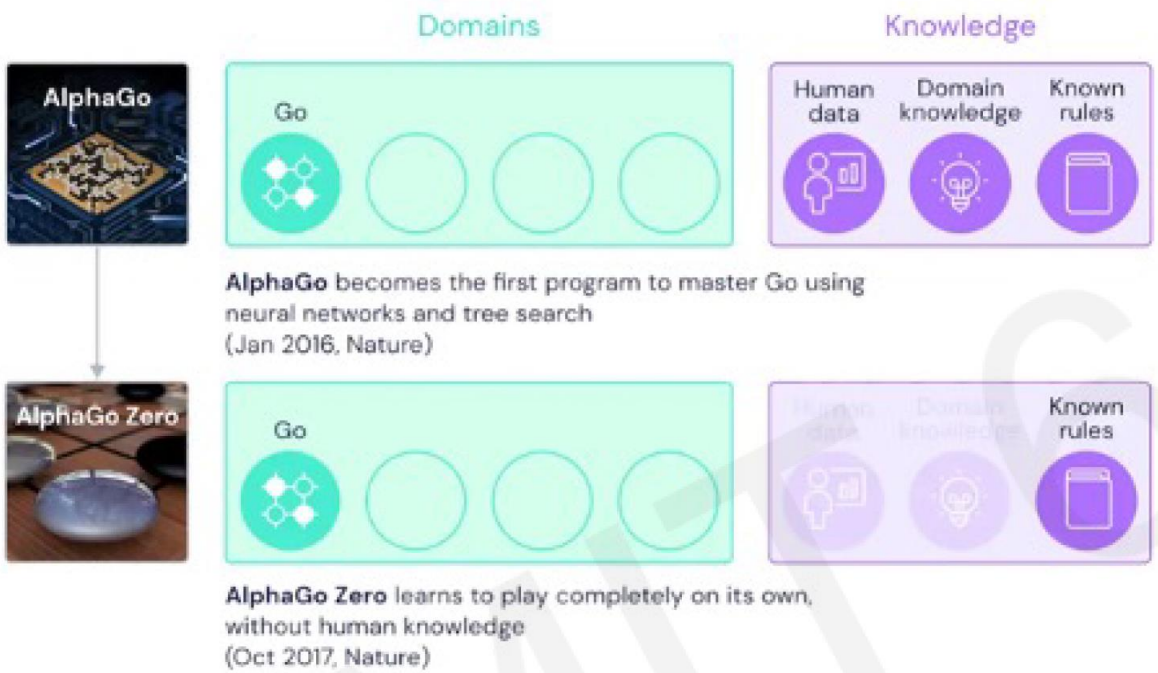
**return** $\theta$

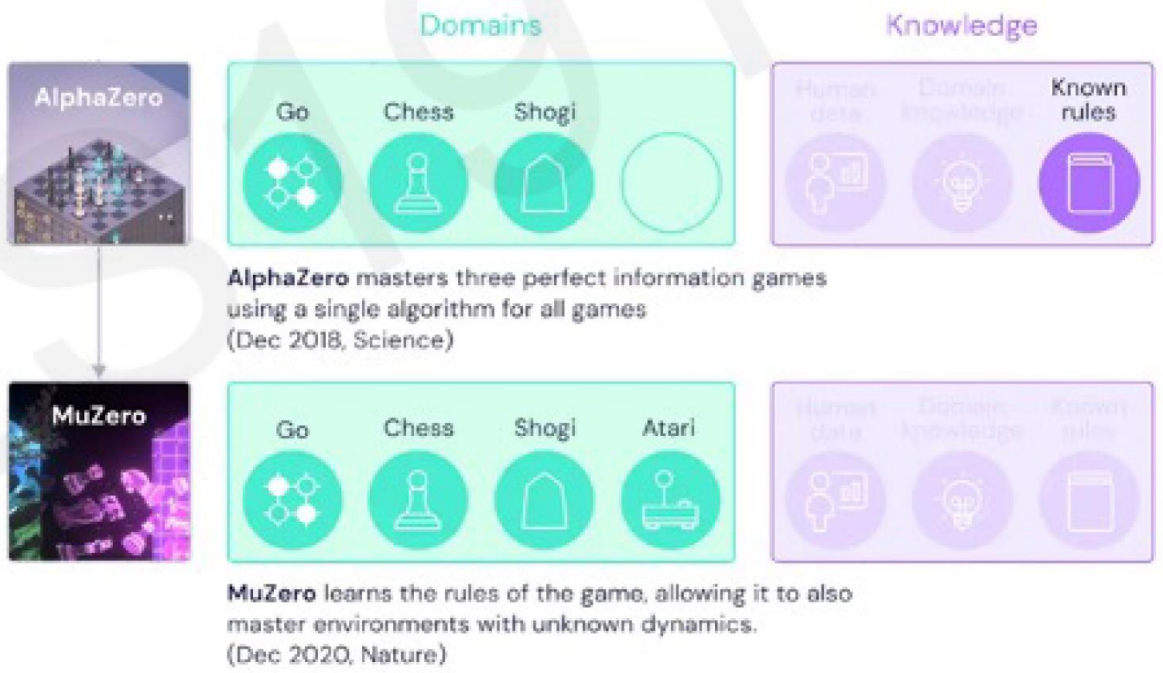
**end function**

# AlphaGo Beats Top Human Player (2016)



# MuZero: Learning Dynamics for Planning (2020)



# Deep Reinforcement Learning Summary

## Foundations

- Agents acting in environment
- State-action pairs → maximize future rewards
- Discounting

## Q-Learning

- Q function: expected total reward given **s, a**
- Policy determined by selecting action that maximizes Q function

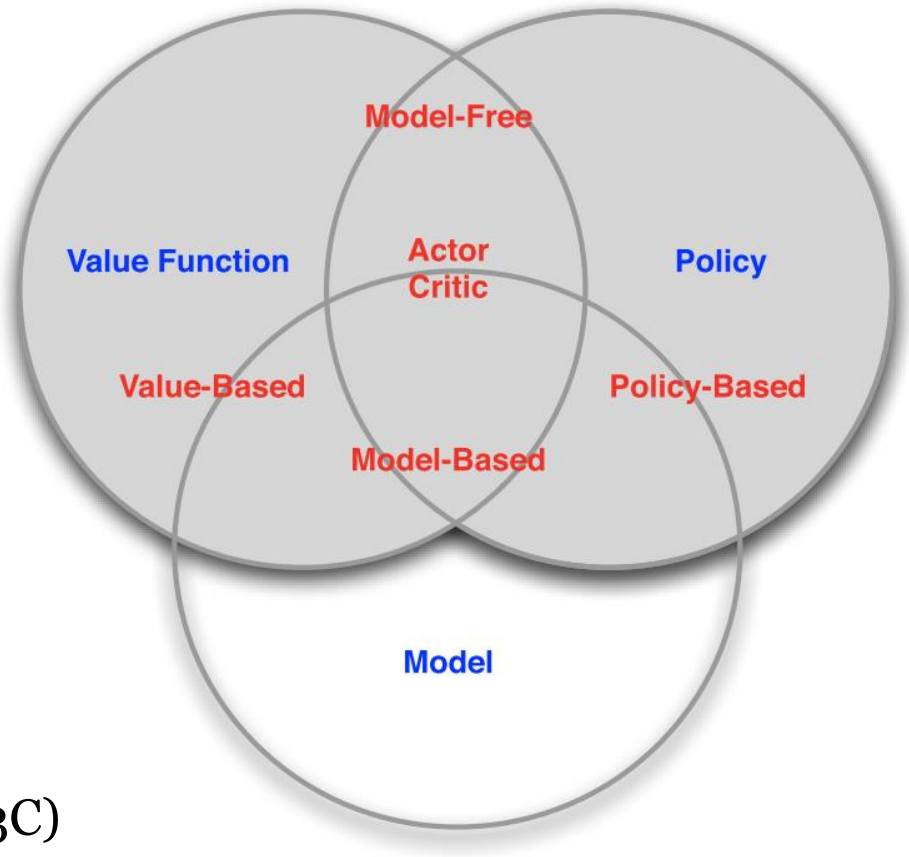
## Policy Gradients

- Learn and optimize the policy directly
- Applicable to continuous action spaces

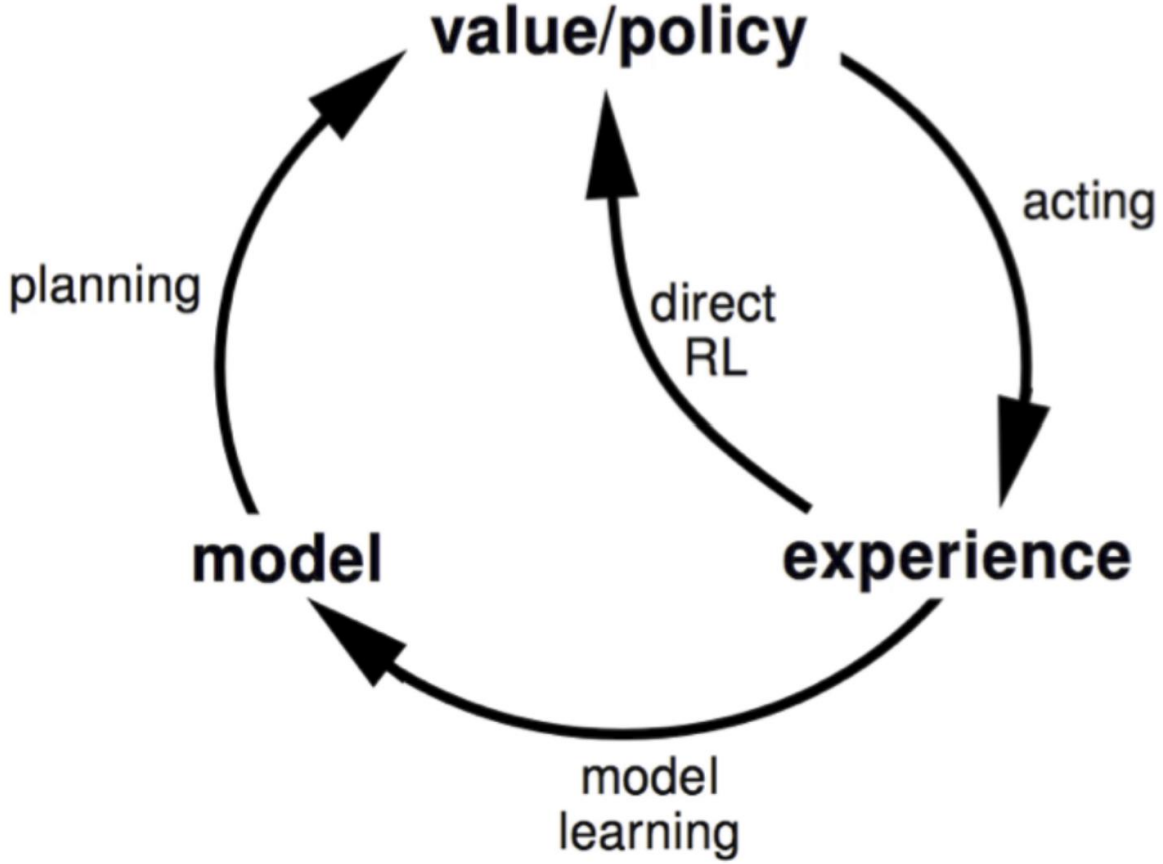
MIT 6S191

# Reinforcement Learning Approaches

- Value-Based:
  - Learn value function
  - Implicit policy (e.g. greedy selection)
  - Example: Deep Q Networks (DQN)
- Policy-Based:
  - No value function
  - Learn explicit (stochastic) policy
  - Example: Stochastic Policy Gradients
- Model-Based:
  - Learn transition model
  - Implicit policy
  - Example: Dreamer
- Actor-Critic:
  - Learn value function
  - Learn policy using value function
  - Example: Asynchronous Advantage Actor Critic (A3C)



# Model-Based vs Model-Free RL



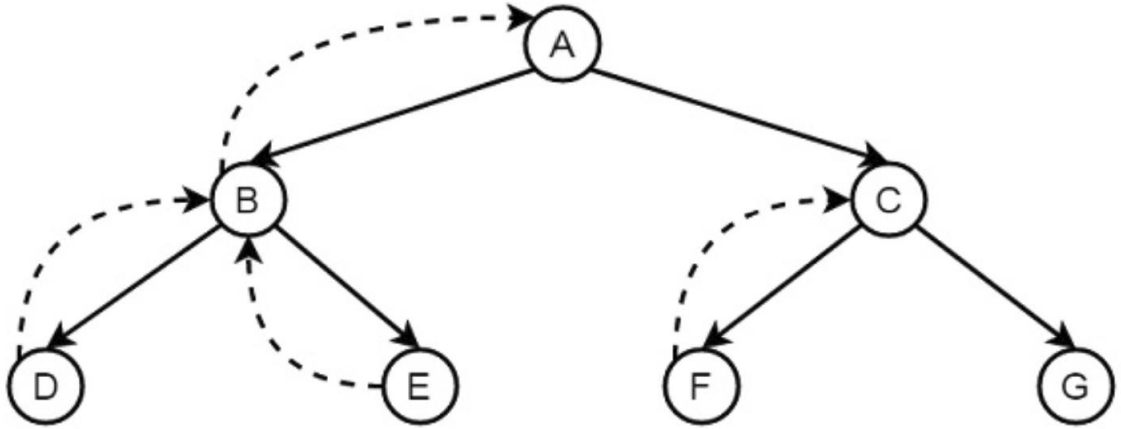

Learn *policy* direct or learn *transition* first and then policy?

# Learning Policies vs Learning Transitions

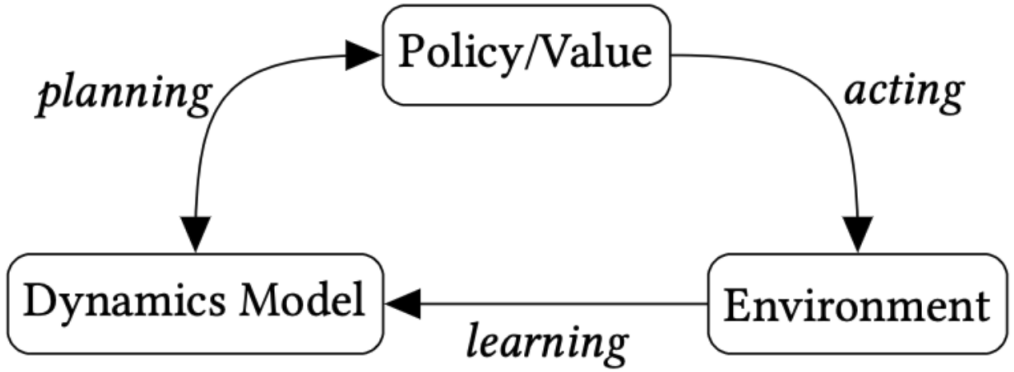
- $s \rightarrow a \rightarrow s' \rightarrow a' \rightarrow s'' \rightarrow a'' \rightarrow s''' \rightarrow a''' \rightarrow s''''$
- Learning a policy $s \rightarrow a$
  - Learning how to react in an environment
- Learning a transition $s \rightarrow a \rightarrow s'$
  - Learning how the environment reacts

# Learning vs Planning

- Learning
  - Agent changing state in the environment
  - Irreversible state change
  - Forward Path $s \rightarrow a \rightarrow s' \rightarrow a' \rightarrow s'' \rightarrow a'' \rightarrow s''' \rightarrow a''' \rightarrow s''''$
- Planning
  - Agent changing own local state
  - Reversible local state change
  - Backtracking Tree



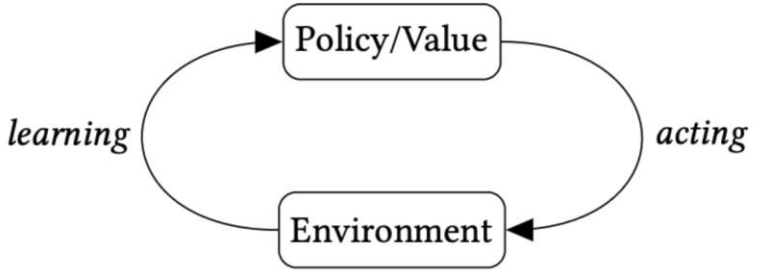
# Model-Based RL



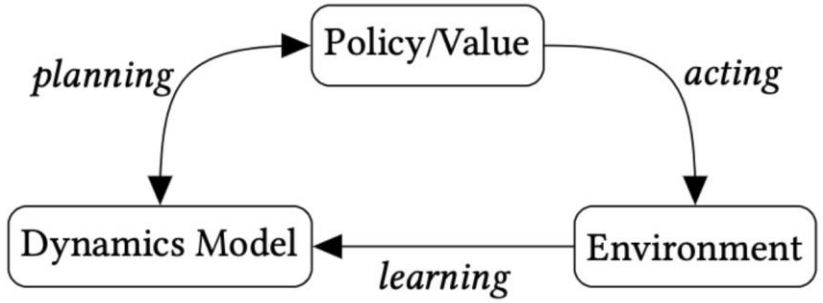

**repeat**
  Sample environment $E$ to generate data $D = (s, a, r', s')$
  Use $D$ to learn $M = T_a(s, s'), R_a(s, s')$ ▷ learning
  **for** $n = 1, \ldots, N$ **do**
    Use $M$ to update policy $\pi(s, a)$ ▷ planning
  **end for**
**until** $\pi$ converges

# Model-Based RL

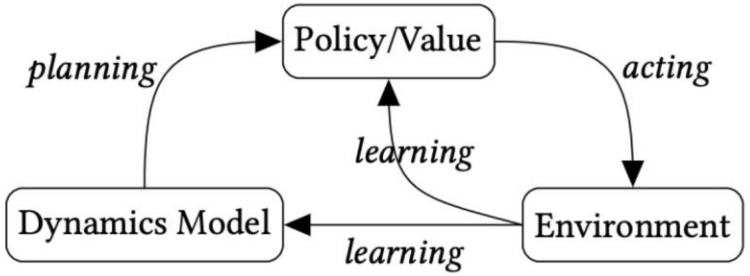
Learn policy directly




Learn model
and then plan actions

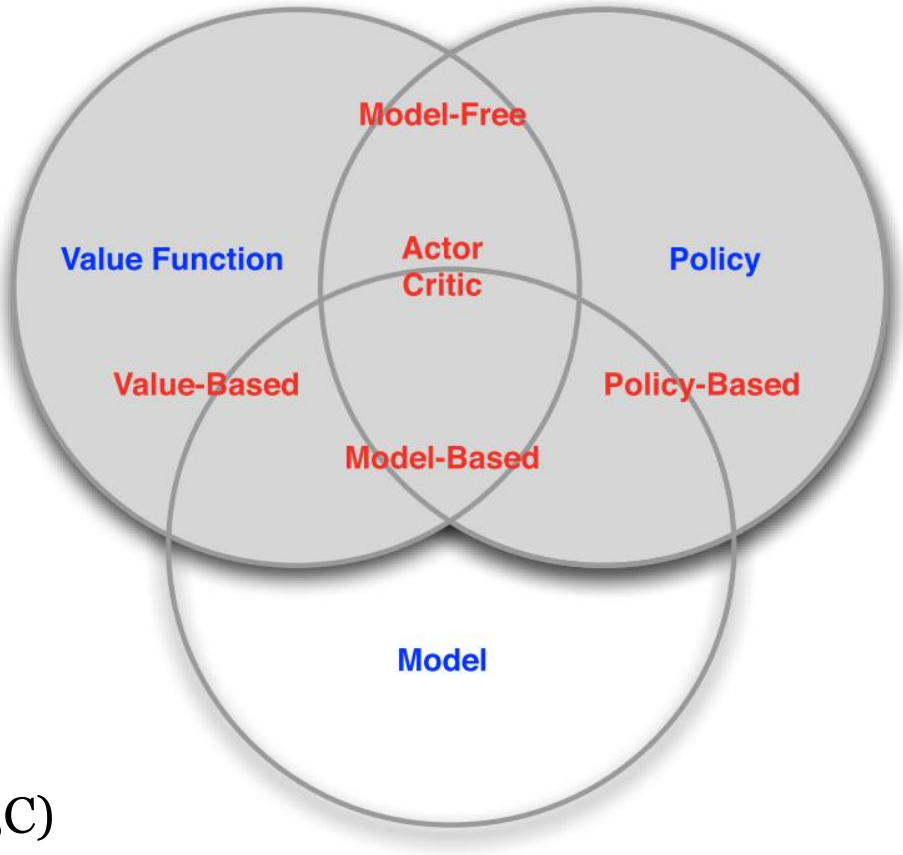



Use experience to
update both model and policy



# Reinforcement Learning Approaches

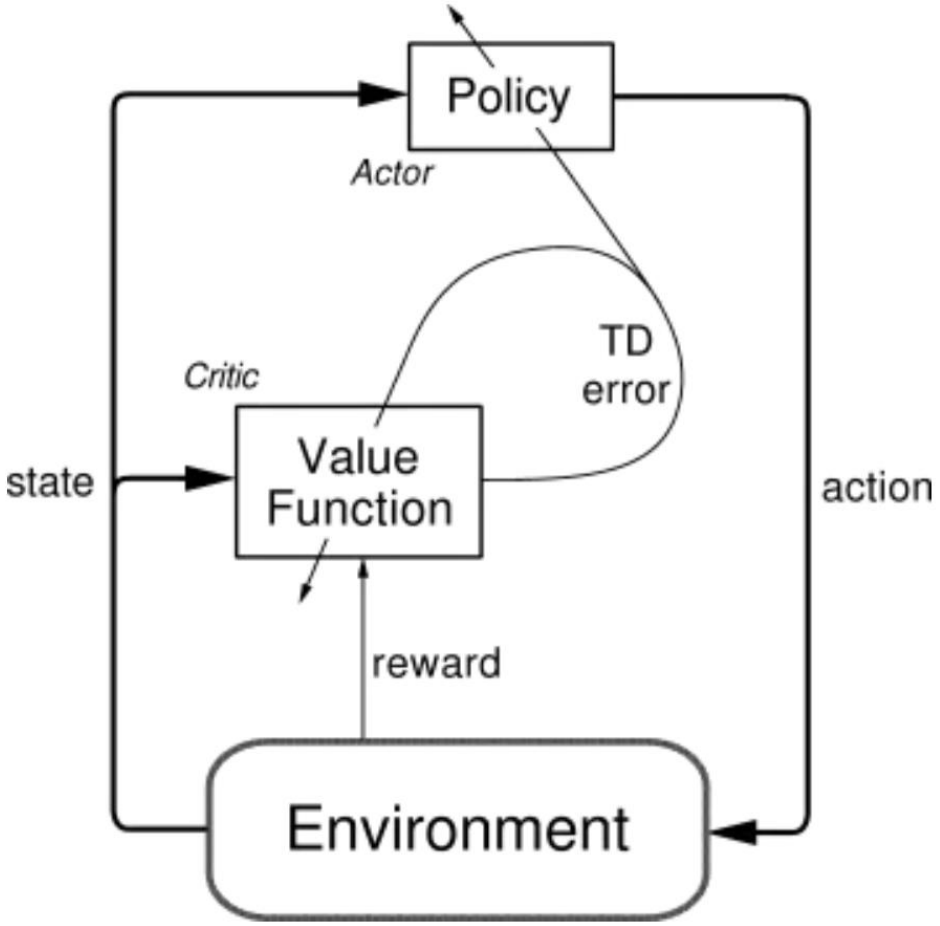
- Value-Based:
  - Learn value function
  - Implicit policy (e.g. greedy selection)
  - Example: Deep Q Networks (DQN)
- Policy-Based:
  - No value function
  - Learn explicit (stochastic) policy
  - Example: Stochastic Policy Gradients
- Model-Based:
  - Learn transition model
  - Implicit policy
  - Example: Dreamer
- Actor-Critic:
  - Learn value function
  - Learn policy using value function
  - Example: Asynchronous Advantage Actor Critic (A3C)



# Actor-Critic RL

- *An Actor* that controls **how our agent behaves** (policy-based method).
- *A Critic* that measures **how good the action taken is** (value-based method).
- Two ideas to reduce variance
  - Temporal difference bootstrapping
  - Baseline subtraction

# Actor-Critic RL

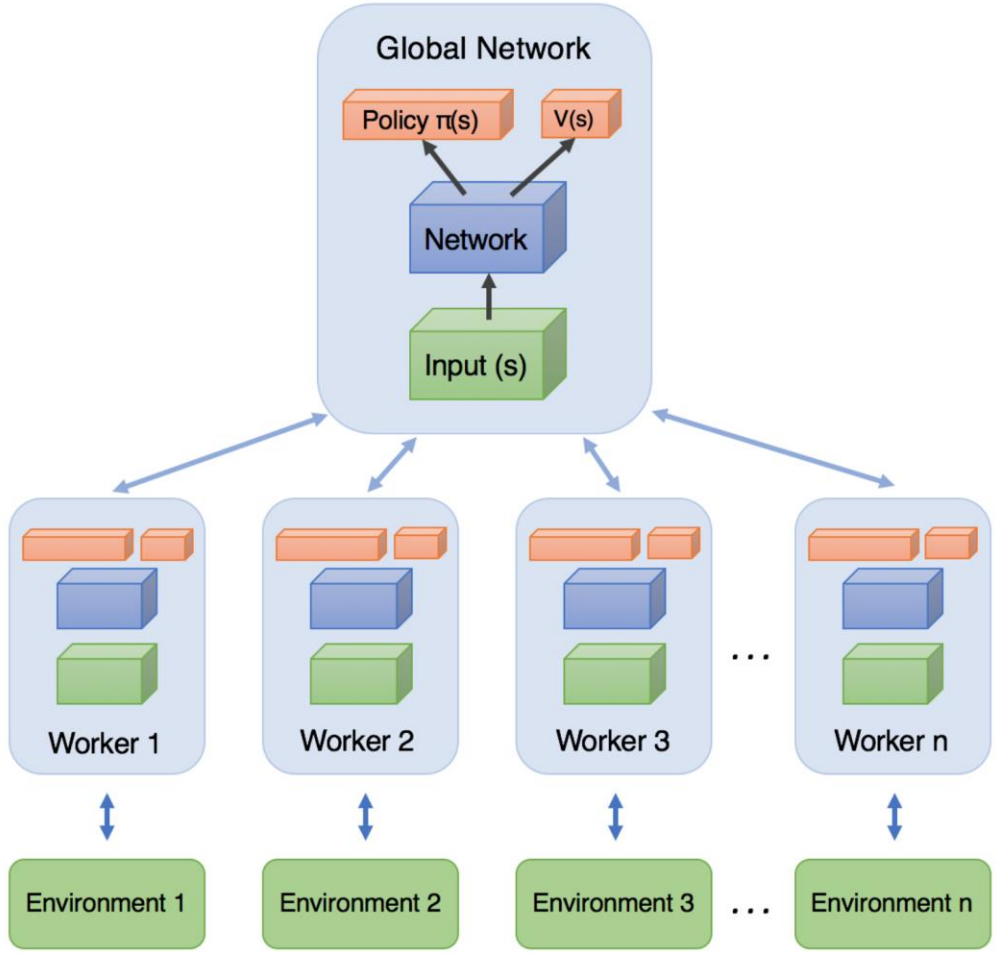


# A3C – Asynchronous Advantage Actor-Critic

- **Asynchronous**: The algorithm is an asynchronous algorithm where multiple worker agents are trained in parallel, each with their environment. This allows the algorithm to train faster as more workers are training in parallel and attain a more diverse training experience as each worker's experience is independent.
- **Advantage**: Advantage is a metric to judge how good its actions were and how they turned out. This allows the algorithm to focus on where the network's predictions were lacking. Intuitively, this will enable it to measure the advantage of taking action, following the policy $\pi$ at the given timestep.
- **Actor-Critic**: The Actor-Critic aspect of the algorithm uses an architecture that shares layers between the policy and value function.

# A3C – Asynchronous Advantage Actor-Critic

1. Fetch the global network parameters
2. Interact with the environment by following the local policy for *n* number of steps
3. Calculate value and policy loss
4. Get gradients from losses
5. Update the global network
6. Repeat




https://pylessons.com/A3C-reinforcement-learning

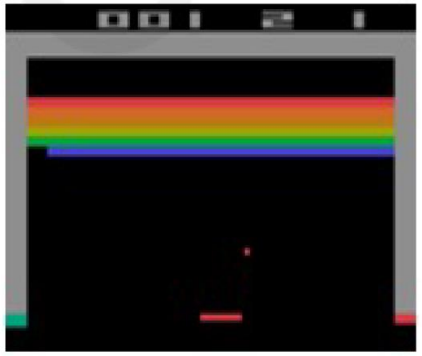
# Deep Reinforcement Learning Summary

## Foundations

- Agents acting in environment
- State-action pairs → maximize future rewards
- Discounting

## Q-Learning

- Q function: expected total reward given **s, a**
- Policy determined by selecting action that maximizes Q function



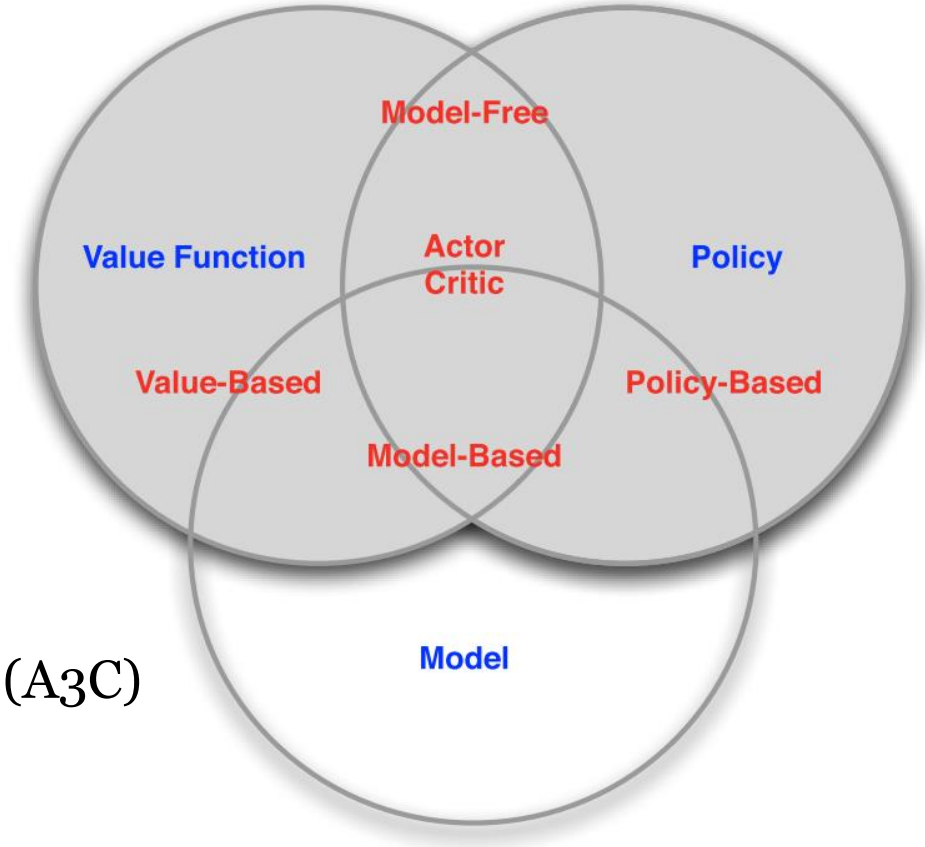
## Policy Gradients

- Learn and optimize the policy directly
- Applicable to continuous action spaces

MIT 6S191

# Reinforcement Learning Concepts

- Value-Based:
  - Learn value function
  - Implicit policy (e.g. greedy selection)
  - Example: Deep Q Networks (DQN)
- Policy-Based:
  - No value function
  - Learn explicit (stochastic) policy
  - Example: Stochastic Policy Gradients
- Actor-Critic:
  - Learn value function
  - Learn policy using value function
  - Example: Asynchronous Advantage Actor Critic (A3C)



# Multi-Agent Reinforcement Learning

# Multiagent RL (MARL) – Motivation

- In multi-agent systems it may be beneficial to learn
  - Environment dynamics
  - Reward functions
  - Other agents' strategies
- Can be useful for cooperative as well as competitive scenarios
- Finding solutions by hand may be difficult and time-consuming
  - Learning may help
  - Also lets the agent adapt to changes in the environment

# Axes of Multi-Agent RL

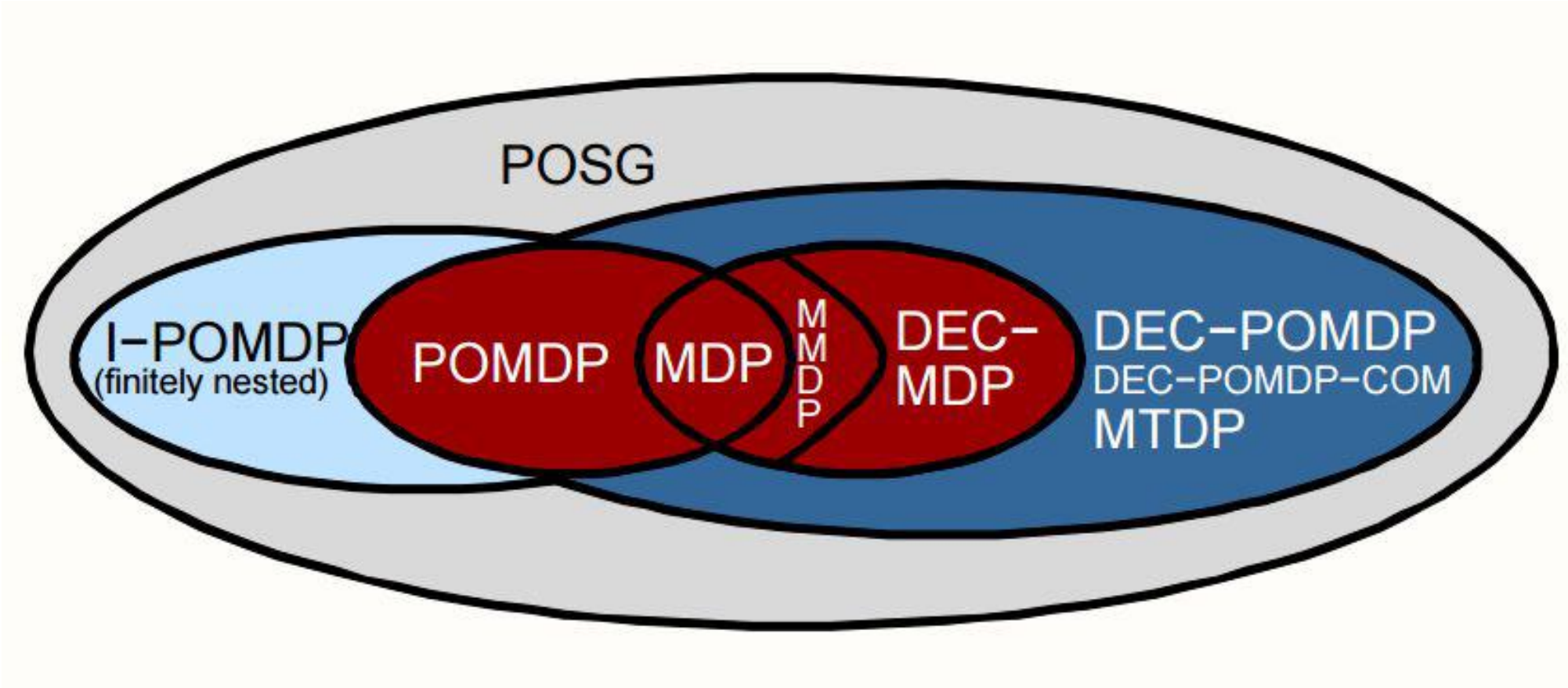
- Centralized:
  - One brain/algorithm deployed across many agents
- Prescriptive:
  - Suggests how agents should behave
- Cooperative: Agents cooperate to achieve a goal
  - Shared team reward
- Numbers of agents
  - One (single-agent)
  - Two (very common)
  - Finite
  - Infinite
- Decentralized:
  - All agents learn individually
  - Communication limitations defined by environment
- Descriptive:
  - Forecast how agent will behave
- Competitive: Agents compete against each other
  - Zero-sum games
  - Individual opposing rewards
- Neither: Agents maximize their utility which may require cooperating and/or competing
  - General-sum games

# MARL – Challenges

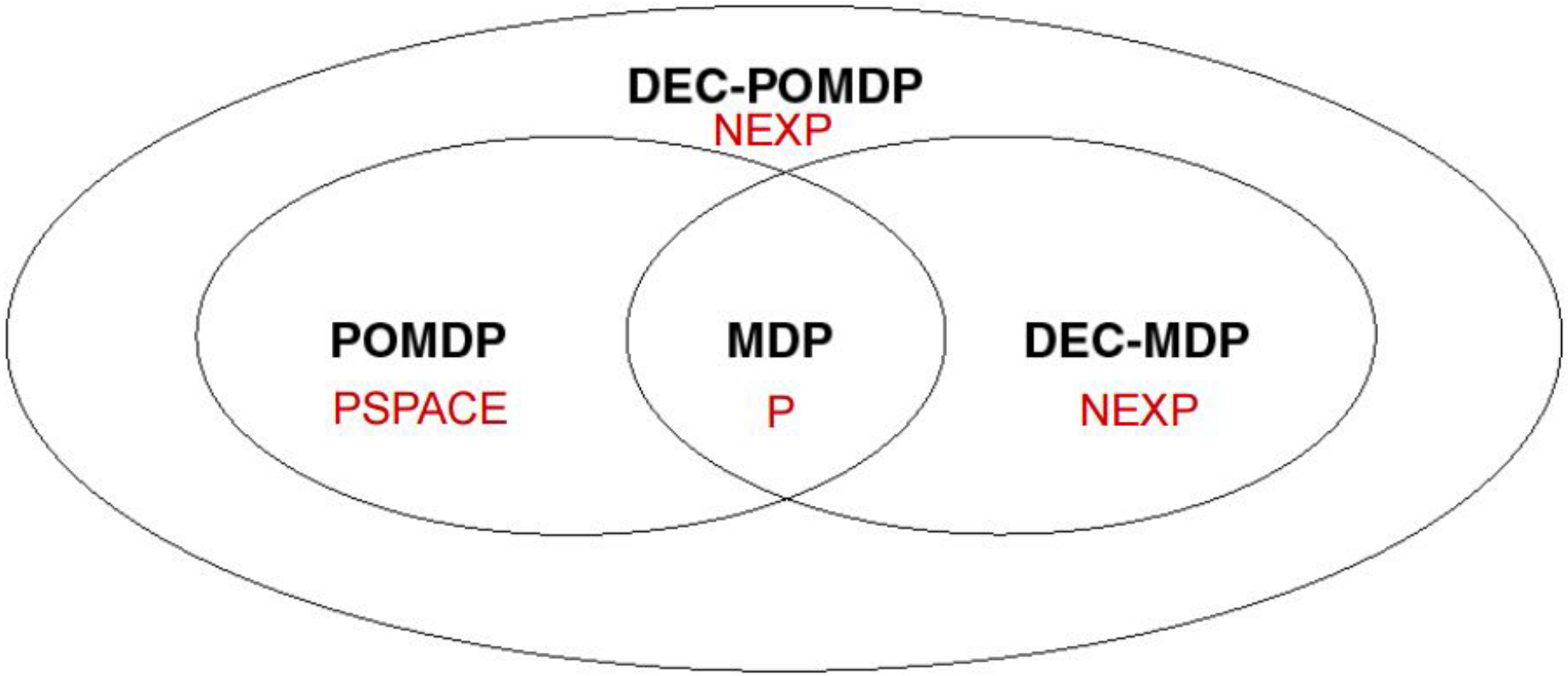
- Non-stationary environment when multiple agents learn
  - Simultaneous learning and teaching for individual agent
- Lack of observability (e.g., actions and rewards of other agents)
- Multiple Agents -> Large State Space (Esp. with simultaneous actions)

# Markov Models

| | No Agents | Single Agent | Multiple Agents |
|---|---|---|---|
| State Known | Markov Chain | Markov Decision Process (MDP) | Markov Game (a.k.a. Stochastic Game) |
| State Observed Indirectly | Hidden Markov Model (HMM) | Partially-Observable Markov Decision Process (POMDP) | Partially-Observable Stochastic Game (POSG) |


POSG
I-POMDP
(finitely nested)
POMDP
MDP
MMDP
DEC-MDP
DEC-POMDP
DEC-POMDP-COM
MTDP

# Complexity Results



# MDP, POMDP, and Dec-POMDP

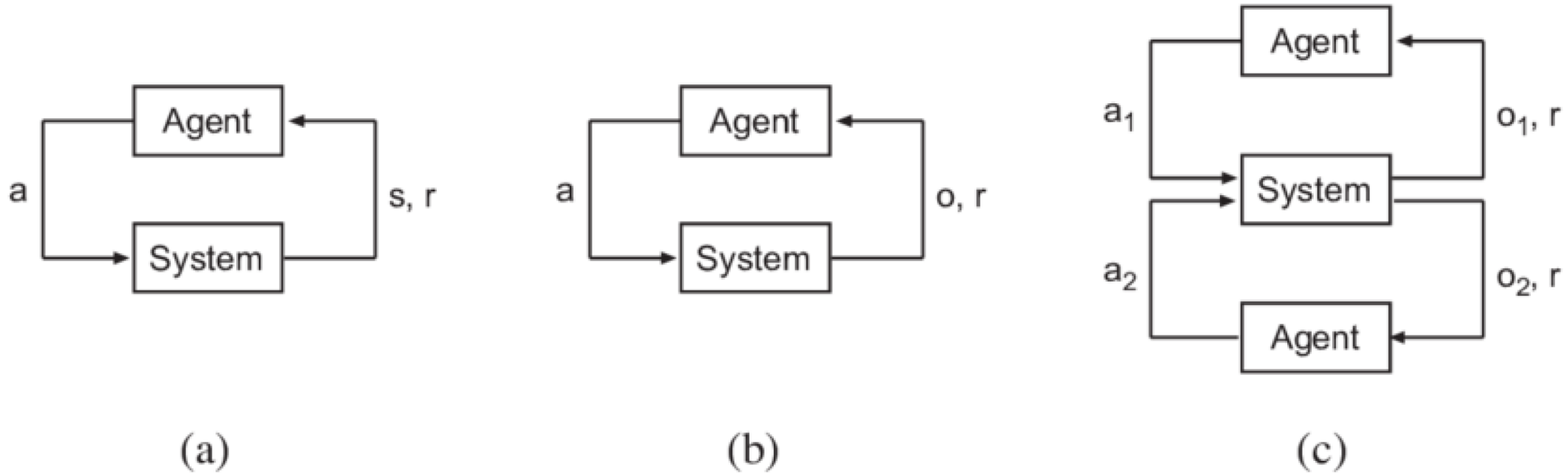



Figure: (a) Markov decision process (MDP) (b) Partially observable Markov decision process (POMDP) (c) Decentralized partially observable Markov decision process with two agents (Dec-POMDP)

# Dec-POMDP

- A Dec-POMDP can be defined with the tuple: $\langle I, S, \{A_i\}, P, \{\Omega_i\}, O, R, h \rangle$ where

- $I$ *is a finite set of agents indexed* $1, \ldots, n$.
- $S$ *is a finite set of states, with distinguished initial state* $s_0$.
- $A_i$ *is a finite set of actions available to agent* $i$, *and* $\vec{A} = \otimes_{i \in I} A_i$ *is the set of joint actions.*
- $P : S \times \vec{A} \to \Delta S$ *is a Markovian transition function.* $P(s'|s, \vec{a})$ *denotes the probability that after taking joint action* $\vec{a}$ *in state* $s$ *a transition to state* $s'$ *occurs.*
- $\Omega_i$ *is a finite set of observations available to agent* $i$, *and* $\vec{\Omega} = \otimes_{i \in I} \Omega_i$ *is the set of joint observations.*
- $O : \vec{A} \times S \to \Delta \vec{\Omega}$ *is an observation function.* $O(\vec{o}|\vec{a}, s')$ *denotes the probability of observing joint observation* $\vec{o}$ *given that joint action* $\vec{a}$ *was taken and led to state* $s'$.
- $R : \vec{A} \times S \to \mathbb{R}$ *is a reward function.* $R(\vec{a}, s')$ *denotes the reward obtained after joint action* $\vec{a}$ *was taken and a state transition to* $s'$ *occurred.*
- *If the DEC-POMDP has a finite horizon, that horizon is represented by a positive integer* $h$.

# Dec-POMDP

- A **local policy** for each agent is a mapping from its observation sequences to actions, Ω* → A
  - State is unknown, so beneficial to remember history
- A **joint policy** is a local policy for each agent
- Goal is to maximize expected cumulative reward over a finite or infinite horizon
  - For finite-horizon cannot remember the full observation history

$$V^{\pi}(s_0) = E\Big[\sum_{t=0}^{h-1} R(\vec{a}_t, s_t)|s_0, \pi\Big]$$

  - In infinite case, a discount factor, γ, is used

$$V^{\pi}(s_0) = E\Big[\sum_{t=0}^{\infty} \gamma^t R(\vec{a}_t, s_t)|s_0, \pi\Big]$$

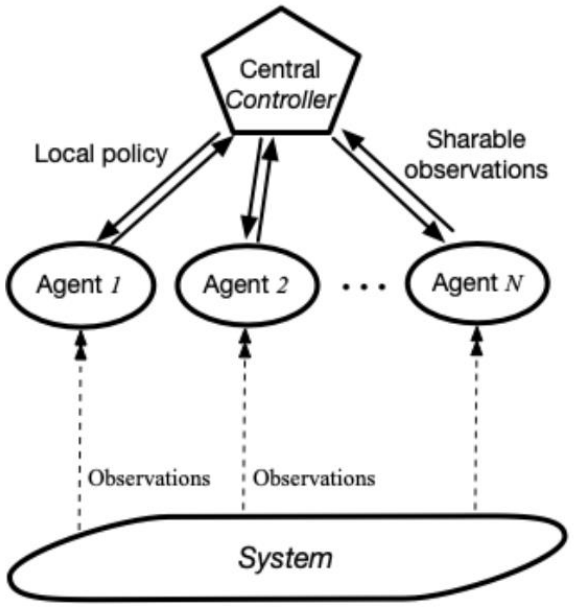
# Dec-POMDP

- Agents must consider the choices of all others in addition to the state and action uncertainty present in POMDPs.
- This makes DEC-POMDPs much harder to solve (NEXP-complete).
- No common state estimate (centralized belief state)
  - Each agent depends on the others
  - This requires a belief over the possible policies of the other agents
  - Can’t transform Dec-POMDPs into a continuous state MDP (how POMDPs are typically solved)

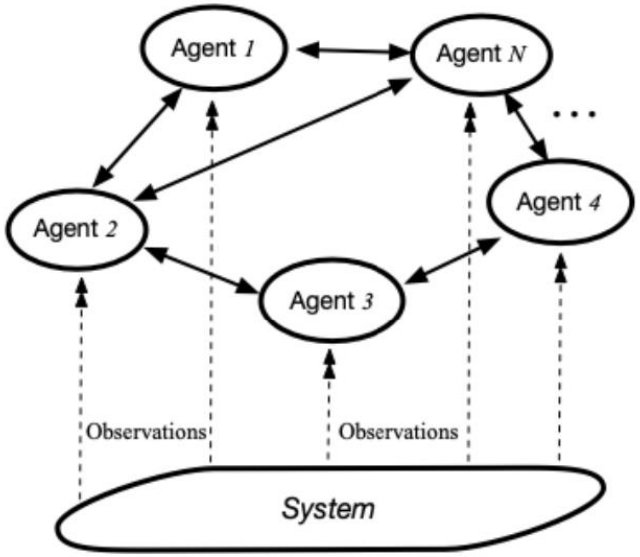
# What Problems are Dec-POMDPs Good For?

- Sequential (not “one shot” or greedy)
- Cooperative (not single agent or competitive)
- Decentralized (not centralized execution or free, instantaneous communication)
- Decision-theoretic (probabilities and values)

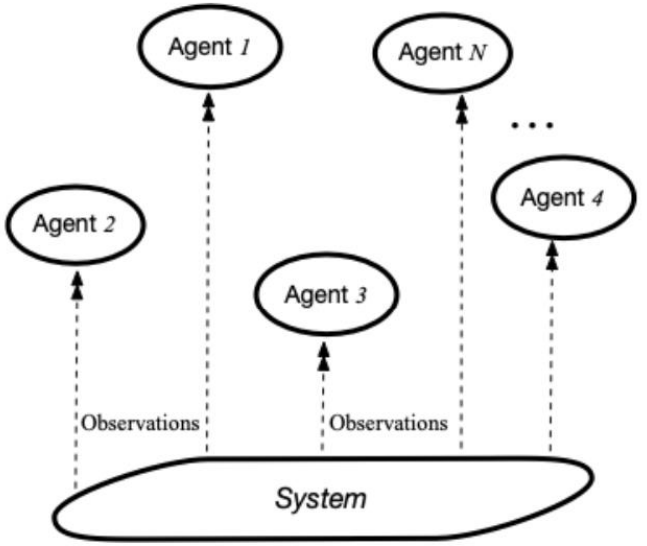
# Multi-Agent Reinforcement Learning




(a) Centralized setting

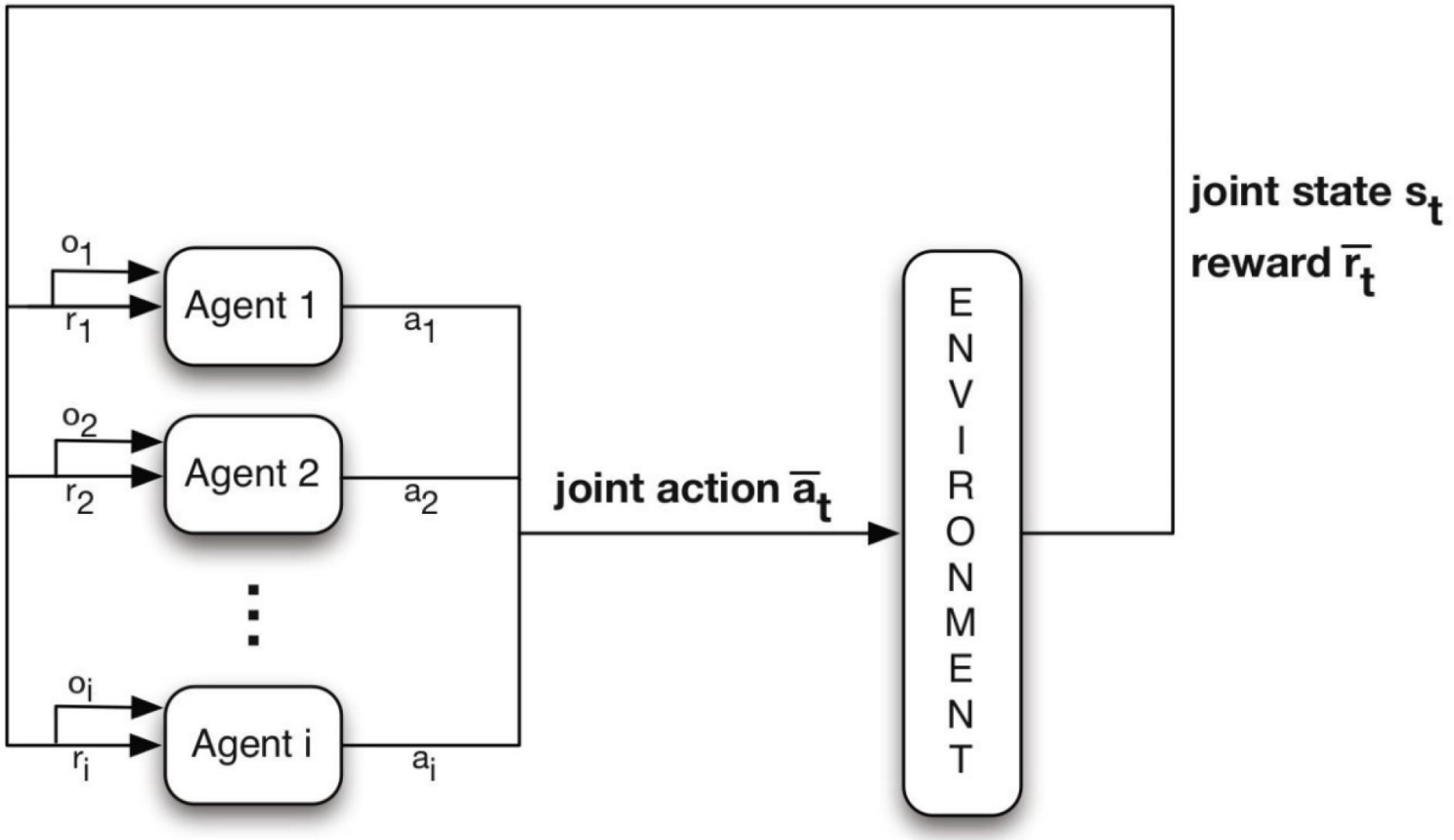



(b) Decentralized setting with networked agents



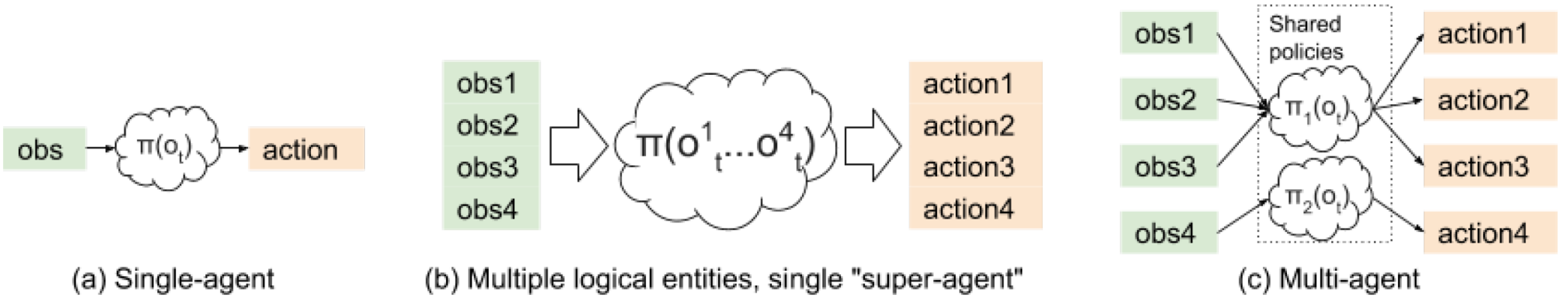

(c) Fully decentralized setting

# Multi-Agent Reinforcement Learning




Source: Nowe, Vrancx & De Hauwere 2012

# MARL Policies

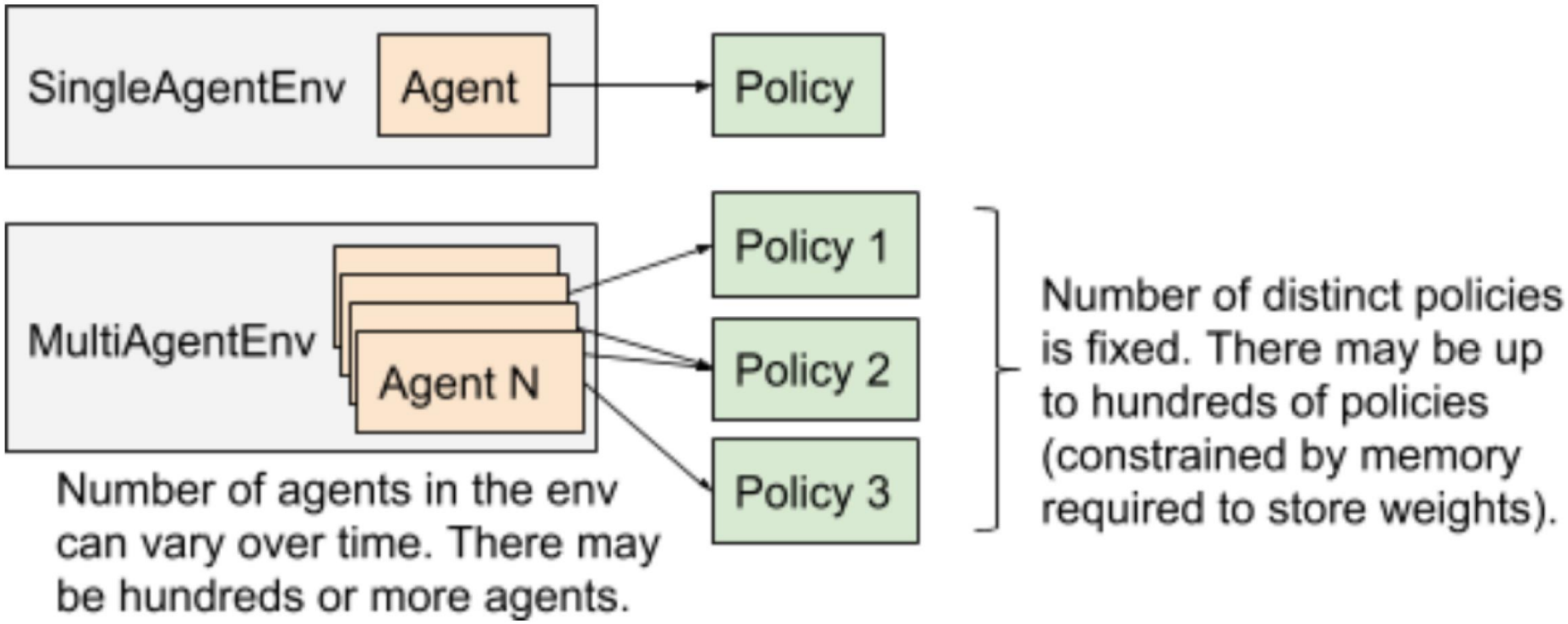



(a) Single-agent

(b) Multiple logical entities, single "super-agent"

(c) Multi-agent

# MARL Policies

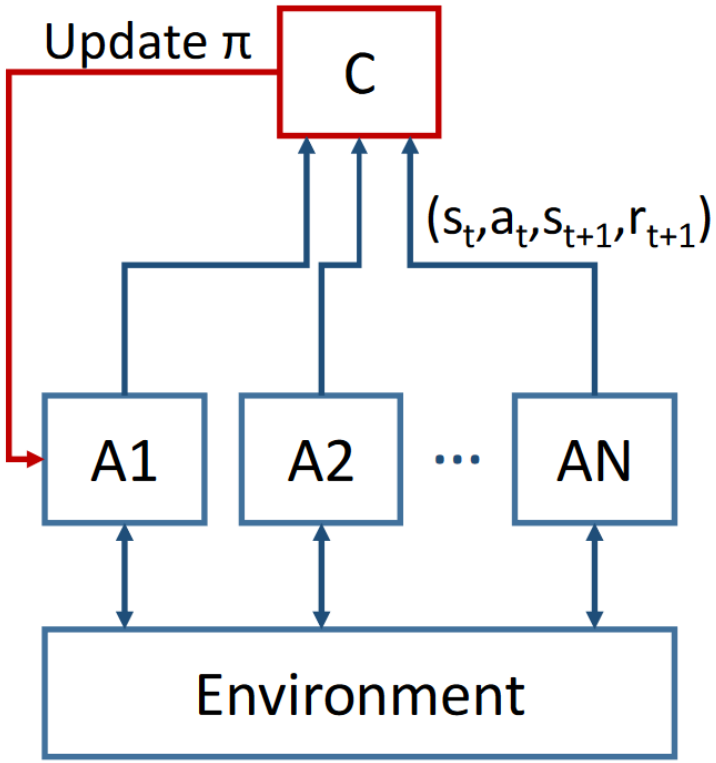


# Decentralized Multi-Agent Deep Reinforcement Learning

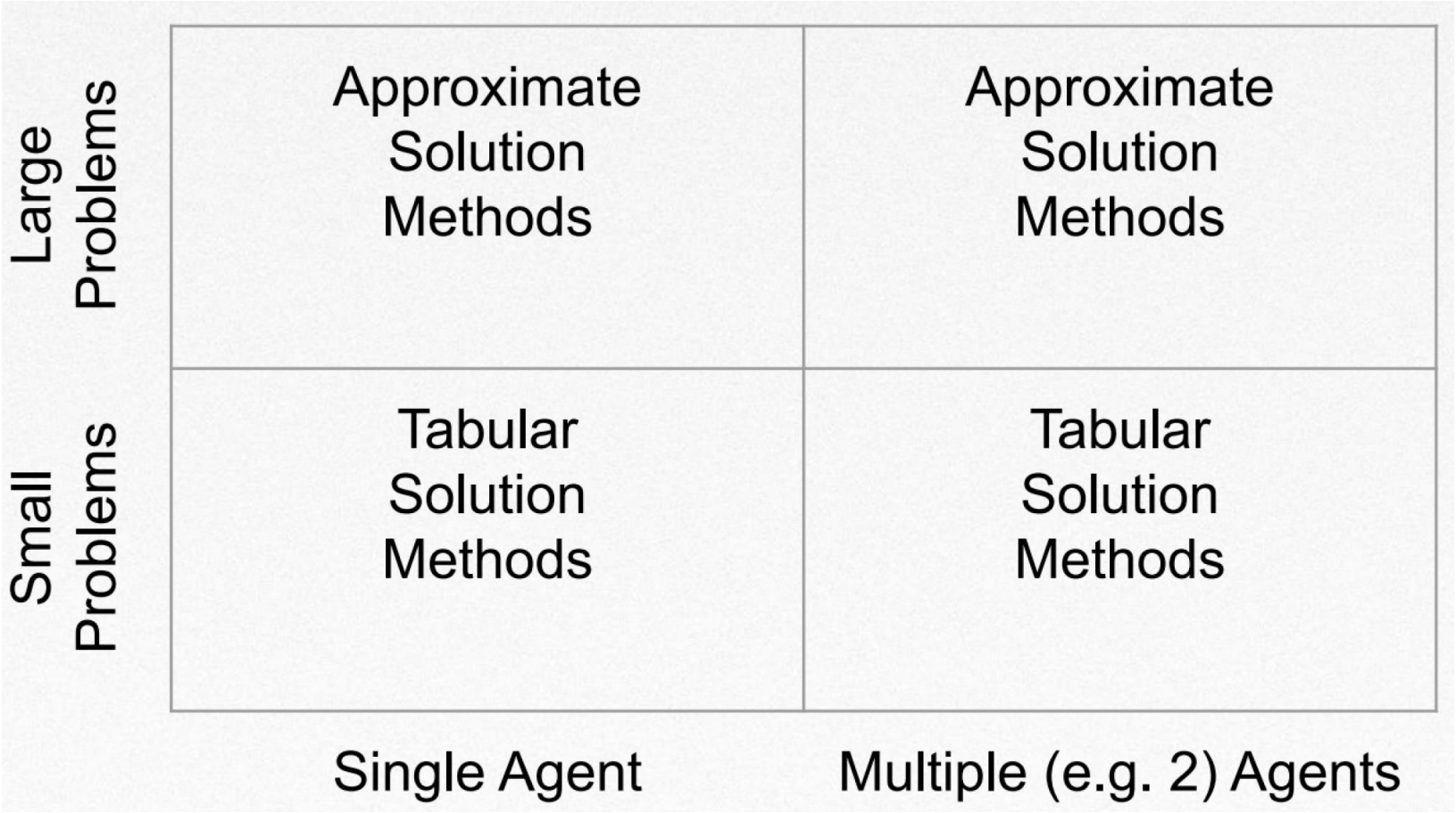
- Though no theoretical guarantees exist, single-agent algorithms may produce interesting results in multi-agent systems
- Ways to stabilize the learning process
  - Clever design of reward systems
  - Training populations of agents
    - Can allow agents to generalize

# Centralized Learning, Decentralized Execution

- Extra information is used for guidance during learning, e.g., actor-critic setup or value function decomposition
- At execution time agents act based on local observations
- Examples of algorithms
  - QMIX
  - COMA
  - MADDPG

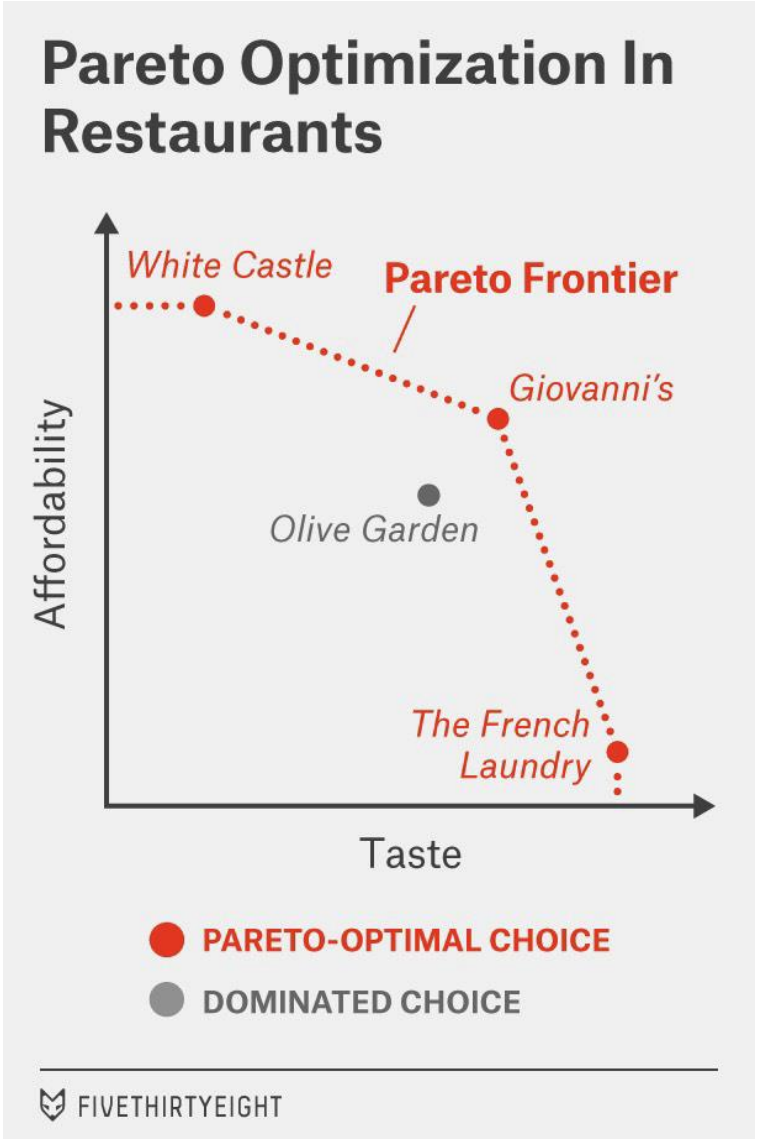


# Research in Multi-Agent RL

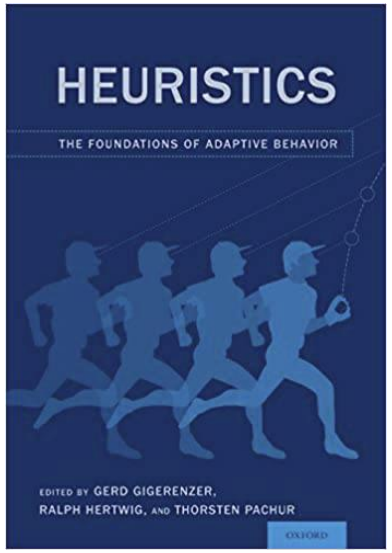


# Cooperation

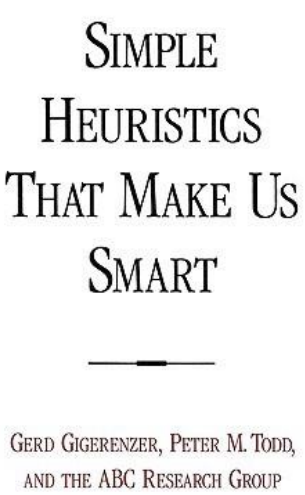
- Non zero sum; win/win
- Vilfredo Pareto
  - Pareto front is, in a cooperative setting, the combination of choices where no agent can be better off without at least making one other agent worse off
- It is the optimal cooperative strategy, the best outcome without hurting others.



# Cooperative Behavior




- Dealing with non-stationarity and partial observability can be done (ignored) by separate training, no communication
- Realism can be improved with Centralized Training/Decentralized Execution -> Centralized controller, or interaction graphs
- Active field of research; overview
  - Value based: VDN, QMIX
  - Policy based: COMA, MADDPG
  - Opponent modeling: DRON, LOLA
  - Communication: Diplomacy game
  - Psychology: Heuristics



# Value Iteration – Recap

**Value iteration**

Initialize array $V$ arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
$\Delta \leftarrow 0$
For each $s \in \mathcal{S}$:
$v \leftarrow V(s)$
$V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

# Turn-Taking 2P Zero-sum Perfect Info. Games

- Player to play at s: $\tau(s)$
- Reward to player i: $r_i$
- Subset of legal actions $\textsc{LegalActions}(s)$
- Often assume episodic and $\gamma = 1$

Values of a state **to player i**: $V_i(s)$

Identities:

$$\forall s, a, s' : r_1 = -r_2, \quad V_1(s) = -V_2(s)$$

# 2P Zero-Sum Perfect Info. Value Iteration

**Value iteration**

Initialize array $V_i$ arbitrarily (e.g., $V_i(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$: *Let $i = t(s)$*
    $v \leftarrow V_i(s)$
    $V_i(s) \leftarrow \max_a \sum_{s',r_i} p(s', r_i | s, a)\left[r_i + \gamma V_i(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V_i(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that *$i = t(s)$*
  $\pi(s) = \arg\max_a \sum_{s',r_i} p(s', r_i | s, a)\left[r_i + \gamma V_i(s')\right]$

# Minimax

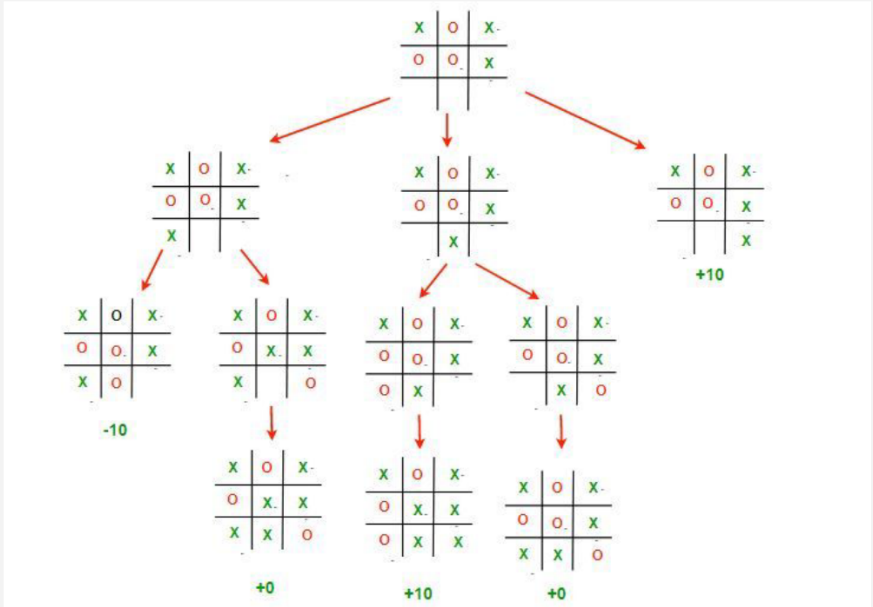
A.K.A. Alpha-Beta, Backward Induction, Retrograde Analysis, etc…

Start from search state $s$,

Compute a depth-limited approximation:

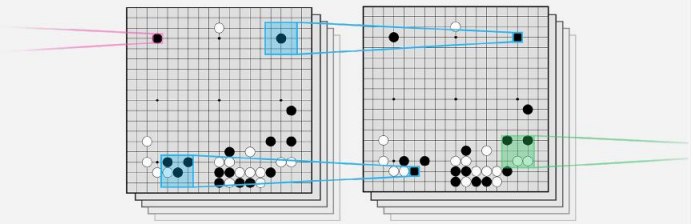
$$V_{i,d}(s) = \begin{cases} u_i(s) & \text{if } s \text{ is terminal,} \\ h_i(s) & \text{if } d = 0, \\ \sum_{s'} p(s,a,s') V_{i,d-1}(s') & \text{otherwise.} \end{cases}$$

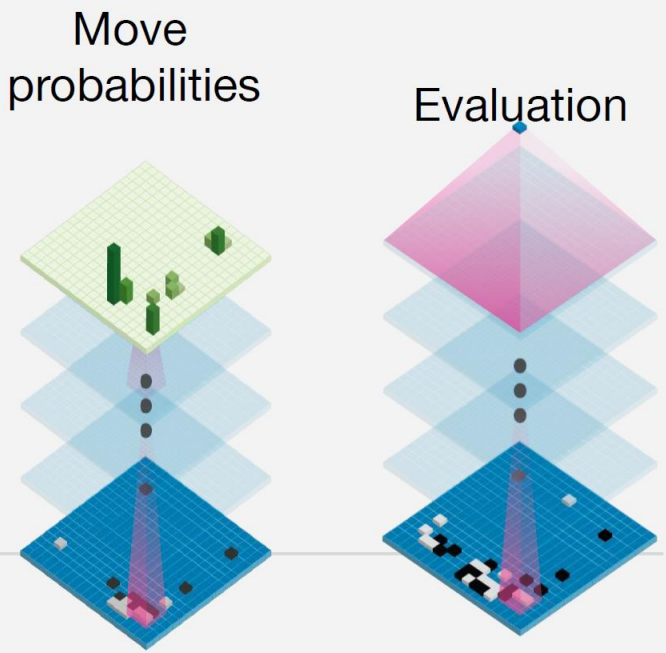
**---> Minimax Search**



# Two-Player Zero-Sum Policy Iteration

- Analogous to adaptation of value iteration
- Foundation of AlphaGo, AlphaGo Zero, AlphaZero
  - Better policy improvement via MCTS
  - Deep network func. approximation
    - Policy prior cuts down *breadth*
    - Value network cuts the *depth*






DeepMind

# 2P Zero-Sum Games with Simultaneous Moves

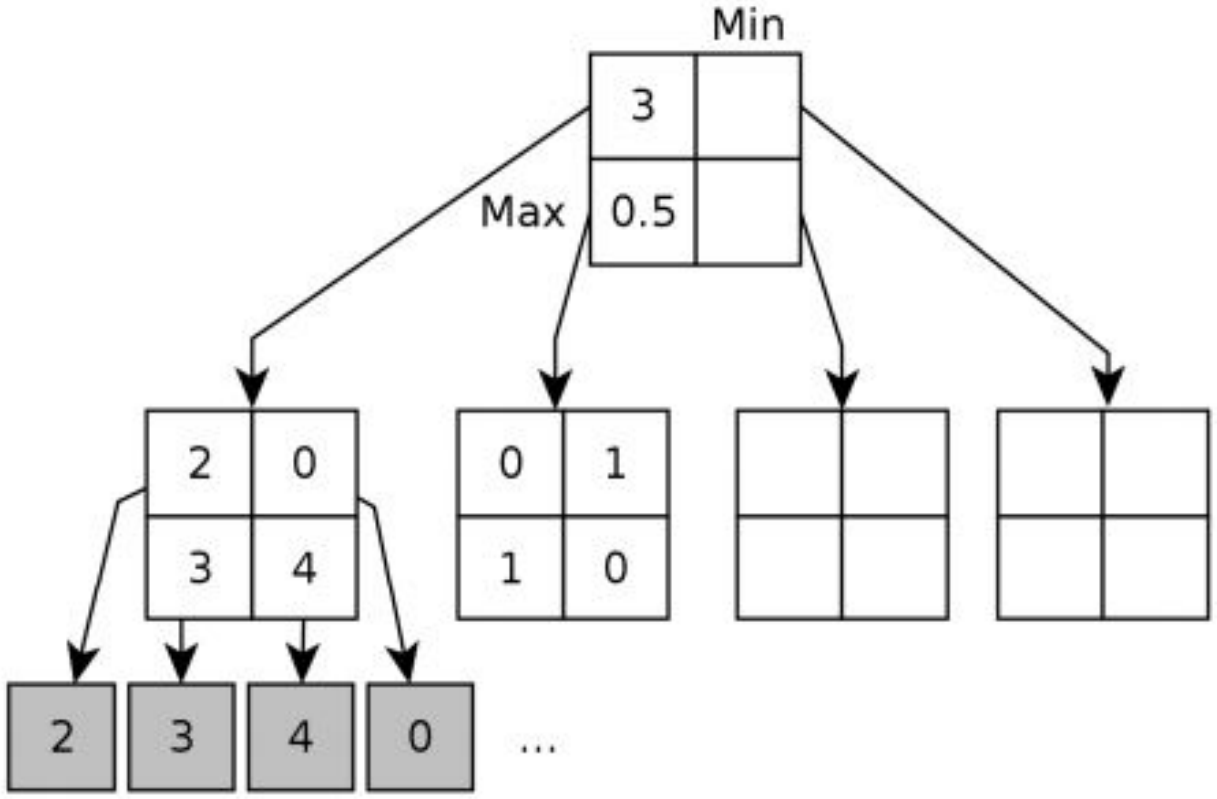



Image from Bozansky et al. 2016

# Markov Games

## “Markov Soccer”

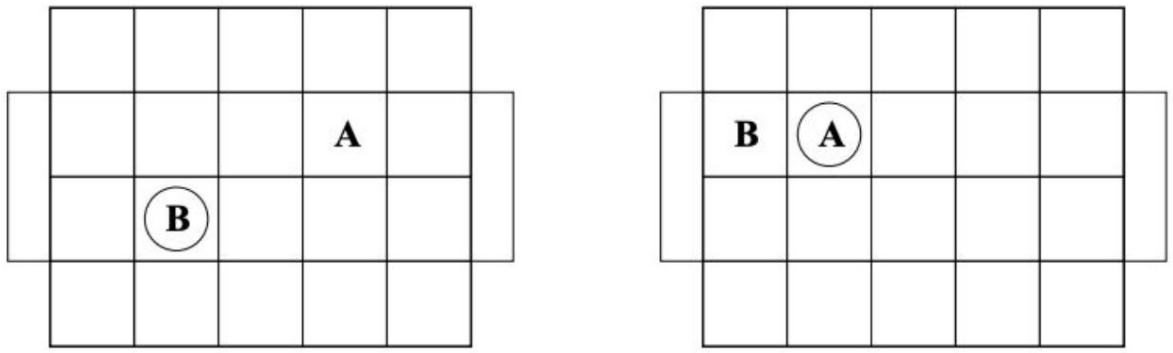
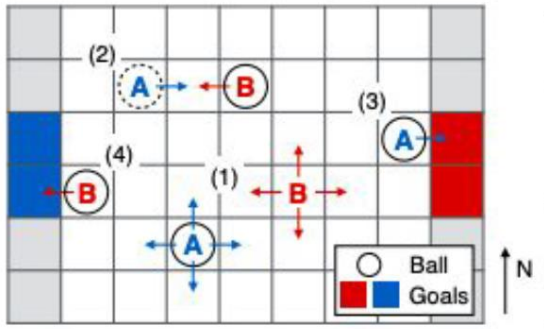



Figure 2: An initial board (left) and a situation requiring a probabilistic choice for A (right).




| | Defensive | Offensive |
|---|---|---|
| w/ ball | Avoid opponent | Advance to goal |
| w/o ball | Defend goal | Intercept the ball |

*Figure 3. Left:* Illustration of the soccer game. *Right:* Strategies of the hand-crafted rule-based agent.

Littman ‘94

He et al. ‘16

Also: Lagoudakis & Parr ‘02, Uther & Veloso ‘03, Collins ‘07

# Value Iteration for Zero-Sum Markov Games

**Value iteration**

Initialize array $V$ arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

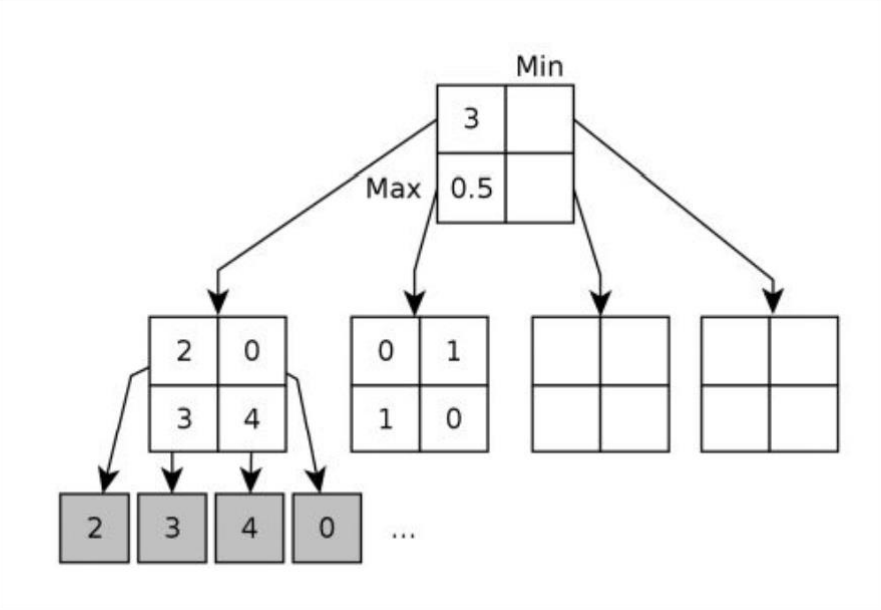
$$\min_{\pi_2(s)} \max_{\pi_1(s)} \mathbb{E}_{a \sim \pi(s), s'} [r_1(s, a, s') + \gamma V_1(s')]$$

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$

computed above

# First MARL Algorithm: Minimax-Q (Littman '94)

1. Start with arbitrary joint value functions $q(s, a, o)$




my action

opponent action

Induces a matrix of values

# First MARL Algorithm: Minimax-Q (Littman '94)

1. Start with arbitrary joint value functions $q(s, a, o)$
2. Define policy $\pi$ as in value iteration (by solving an LP)
3. Generate trajectories of tuple $(s, a, o, s')$ using behavior policy $\pi' = \epsilon \textsc{Unif}(\mathcal{A}) + (1 - \epsilon)\pi$
4. Update $q(s, a, o) = (1 - \alpha)q(s, a, o) + \alpha(r(s, a, o, s') + \gamma v(s'))$

# First MARL Algorithm: Minimax-Q (Littman '94)

Q-values are over joint actions: $Q(s, a, o)$

- s = state
- a = your action
- o = action of the opponent

Instead of playing action with highest $Q(s, a, o)$, play **MaxMin**

$$Q(s, a, o) = (1 - \alpha)Q(s, a, o) + \alpha(r + \gamma V(s'))$$

$$V(s) = \max_{\pi_s} \min_{o} \sum_{a} Q(s, a, o)\pi_s(a)$$

# MARL Formulation

- The agents choose actions according to their policies.
- For agent $j$, the corresponding policy is defined as $\pi^j : S \to \Omega(A^j)$, where $\Omega(A^j)$ is the collection of probability distributions over agent $j$'s action space $A^j$.
- Let $\boldsymbol{\pi} = [\pi^1, \cdots, \pi^N]$ - is the joint policy of all agents, then

$$v^j_{\boldsymbol{\pi}}(s) = v^j(s; \boldsymbol{\pi}) = \sum_{t=0}^{\infty} \gamma^t \mathbb{E}_{\boldsymbol{\pi}, p}[r^j_t | s_0 = s, \boldsymbol{\pi}]$$

- Q-function such that the Q-function $Q^j_{\boldsymbol{\pi}} : S \times A^1 \times \cdots \times A^N \to \mathbb{R}$ of agent $j$ under the joint policy $\boldsymbol{\pi}$:

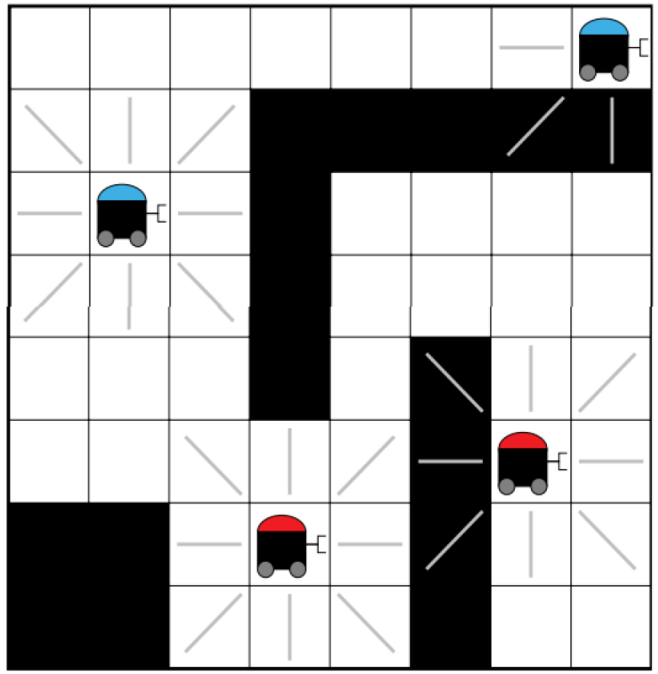
$$Q^j_{\boldsymbol{\pi}}(s, \boldsymbol{a}) = r^j(s, \boldsymbol{a}) + \gamma \mathbb{E}_{s' \sim p}[v^j_{\boldsymbol{\pi}}(s')]$$

# First Era of MARL

- Follow-ups to Minimax Q:
  - Friend-or-Foe Q-Learning (Littman ‘01)
  - Correlated Q-learning (Greenwald & Hall ‘03)
  - Nash Q-learning (Hu & Wellman ‘03)
  - Coco-Q (Sodomka et al. ‘13)
- Function approximation:
  - LSPI for Markov Games (Lagoudakis & Parr ‘02)

# Multi-Agent Deep Q-Network (MADQN)

- *n* pursuit-evasion – a set of agents (the pursuers) are attempting to chase another set of agents (the evaders)
- The agents in the problem are self-interested (or heterogeneous), i.e. they have different objectives
- The two pursuers are attempting to catch the two evaders



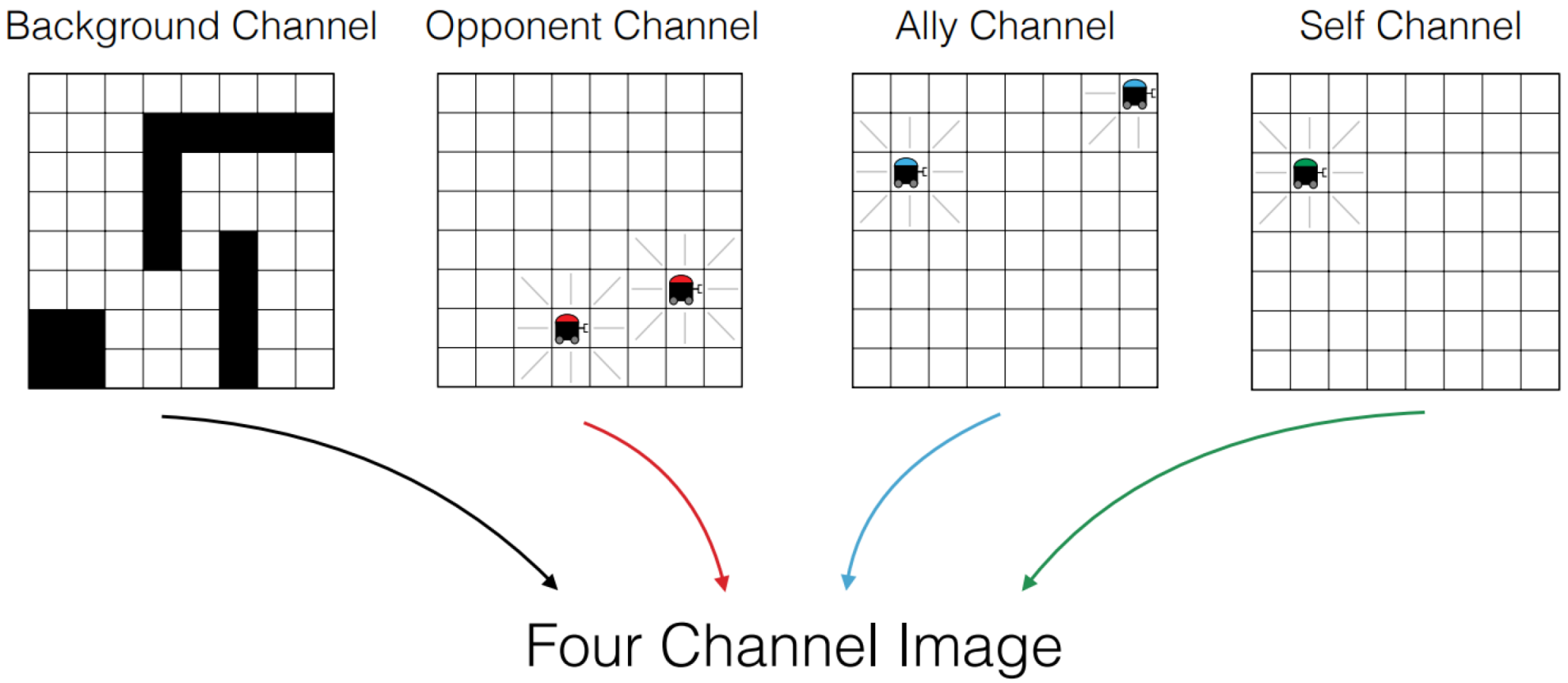
# Multi-Agent Deep Q-Network (MADQN)

**Challenge:** defining the problem in such a way that an arbitrary number of agents can be represented without changing the architecture of the deep Q-Network.

**Solution** (under some assumptions):

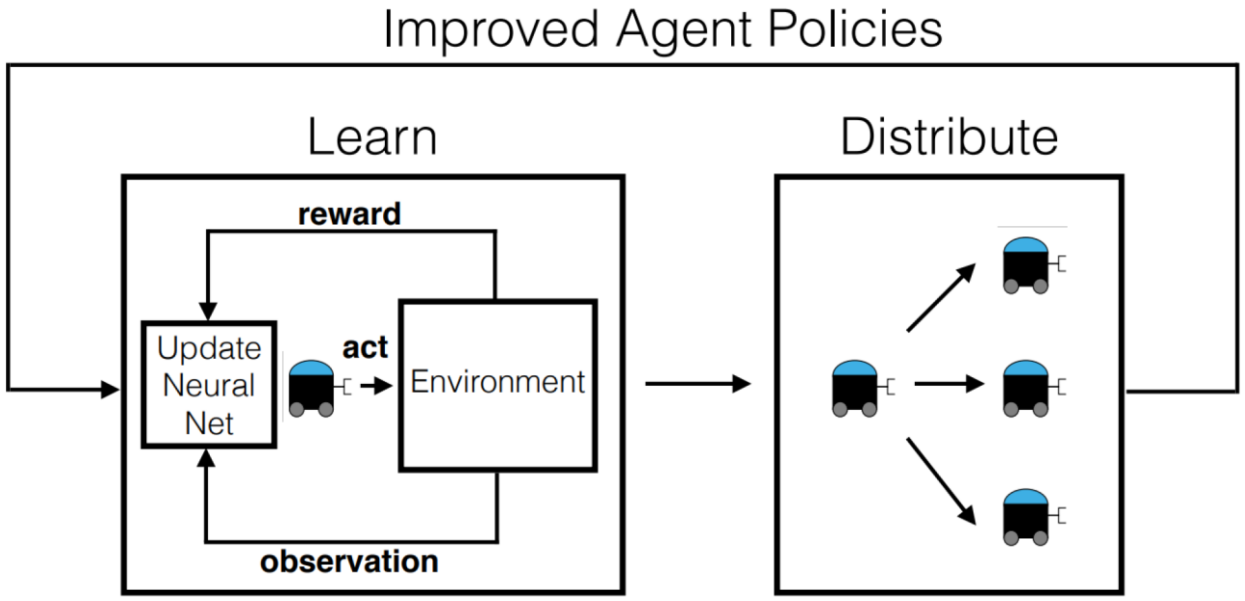
- The image tensor is of size 4$\times$ W $\times$ H, where W and H are the height and width of our two dimensional domain and four is the number of channels in the image.
- Channels:
  - **Background Channel**: contains information about any obstacles in the environment
  - **Opponent Channel:** contains information about all the opponents
  - **Ally Channel:** contains information about all the allies
  - **Self Channel:** contains information about the agent making the decision

# Multi-Agent Deep Q-Network (MADQN)



# Multi-Agent Deep Q-Network (MADQN)

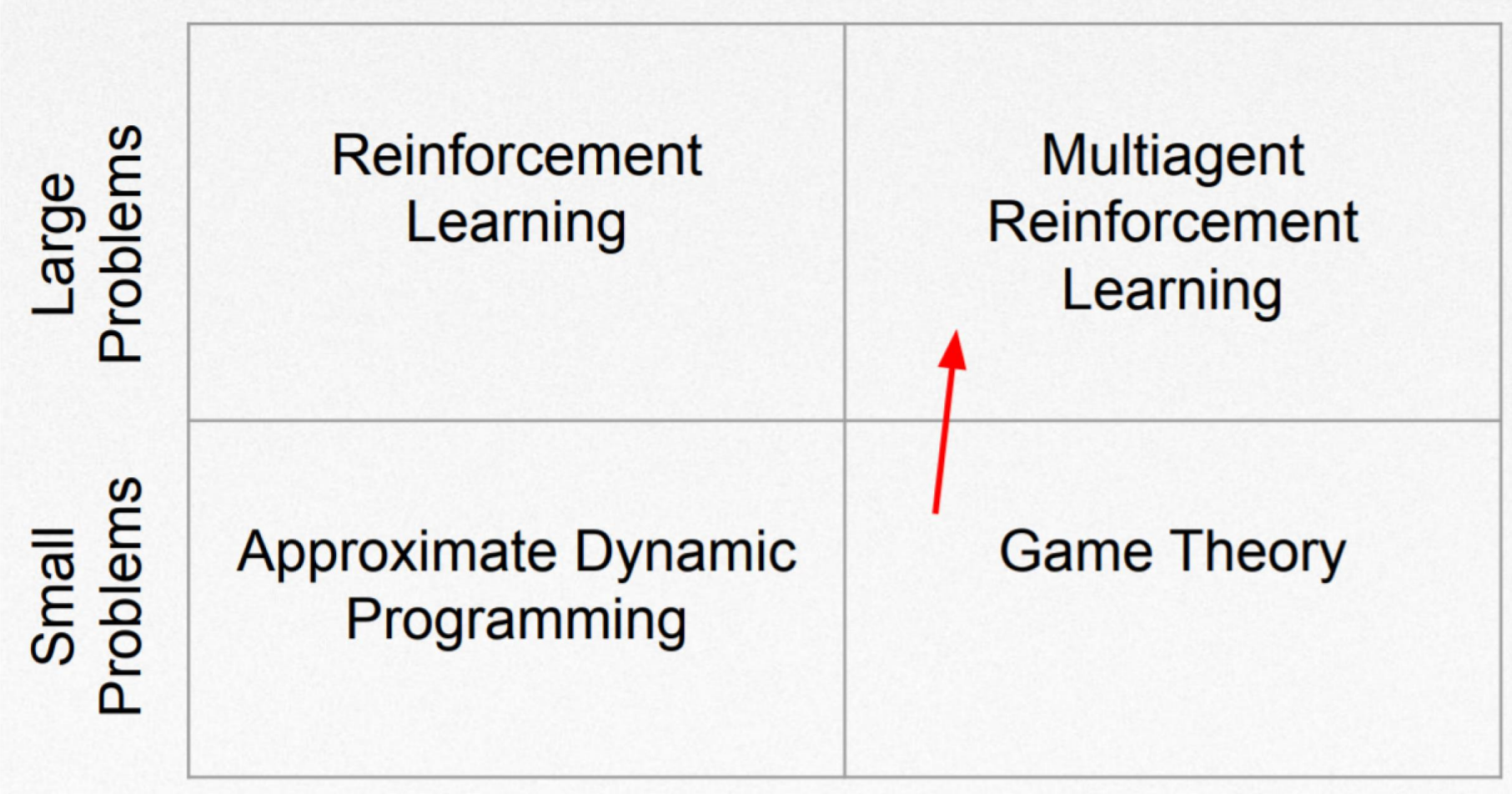
- Train one agent at a time, and fix policies of all the other agents
- After a number of iterations distribute the policy learned by the training agent to all the other agents of its type



# MADQN: Dealing with Ambiguity

- **Challenge**: When two ally agents are occupying the same position in the environment, the image-like state representation for each agent will be identical, so their policies will be exactly the same.
- **Solution**: To break this symmetry – use a stochastic policy for agents. The actions taken by the agent are drawn from a distribution derived by taking a softmax over the Q-values of the neural network. This allows allies to take different actions if they occupy the same state and break the ambiguity.

# Foundations of MARL



# Nash Q-Learning

- In MARL, the objective of each agent is to learn an optimal policy to maximize its value function
- Optimizing the $v_\pi^j$ for agent $j$ depends on the joint policy $\boldsymbol{\pi}$ of all agents
- A Nash equilibrium is a joint policy $\pi$ such that no player has incentive to deviate unilaterally. It is represented by a particular joint policy

$$\boldsymbol{\pi}_* = [\pi_*^1, \cdots, \pi_*^N]$$

such that for all $s \in S, j \in \{1, \cdots, N\}$ it satisfies:

$$v^j(s; \boldsymbol{\pi}_*) = v^j(s; \pi_*^j, \boldsymbol{\pi}_*^{-\boldsymbol{j}}) \geq v^j(s; \pi^j, \boldsymbol{\pi}_*^{-j})$$

Here $\boldsymbol{\pi}_*^{-\boldsymbol{j}}$ is the joint policy of all agents except $j$ as

$$\boldsymbol{\pi}_*^{-j} = [\pi_*^1, \cdots, \pi_*^{j-1}, \pi_*^{j+1}, \cdots, \pi_*^N]$$

# Nash Q-Learning (cont)

- In a Nash equilibrium, each agent acts with the best response $\pi_*^j$ to others, provided that all other agents follow the policy $\boldsymbol{\pi}_*^{-j}$
- For a $N$-agent stochastic game, there is at least one Nash equilibrium with stationary policies, assuming players are rational
- Given Nash policy $\boldsymbol{\pi}_*$, the Nash value function

$$\boldsymbol{v}^{\text{Nash}} = [v_{\boldsymbol{\pi}_*}^1(s), \cdots, v_{\boldsymbol{\pi}_*}^N(s)]$$
$$\boldsymbol{Q}(s,\boldsymbol{a})^{\text{Nash}} = \mathbb{E}_{s'\sim p}[\boldsymbol{r}(s,\boldsymbol{a}) + \gamma \boldsymbol{v}^{\text{Nash}}(s')]$$

where $\boldsymbol{r}(s,\boldsymbol{a}) = [r^1(s,\boldsymbol{a}), \cdots, r^N(s,\boldsymbol{a})]$

# Counterfactual Regret Minimization (CFR)

Zinkevich et al. ‘08

- Algorithm to compute approx Nash eq. In 2P zero-sum games
- Hugely successful in Poker AI
- Size traditionally reduced apriori based on expert knowledge
- Key innovation: counterfactual values: $v_i^c(\pi, s, a)$ $v_i^c(\pi, s)$

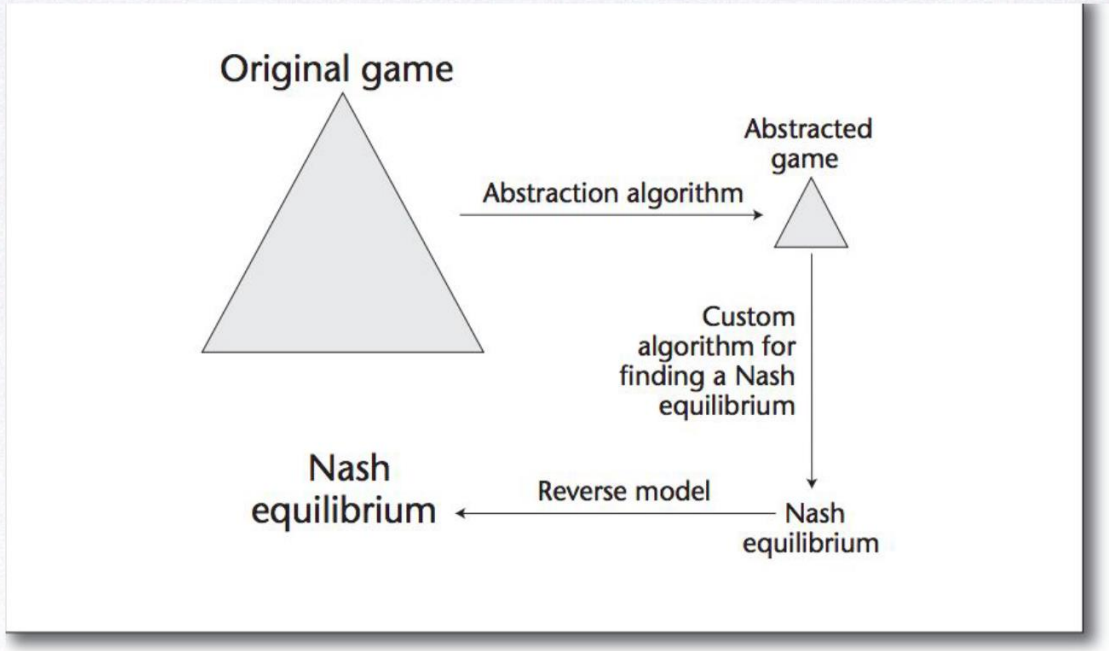



*Figure 1. Current Paradigm for Solving Large Incomplete-Information Games.*

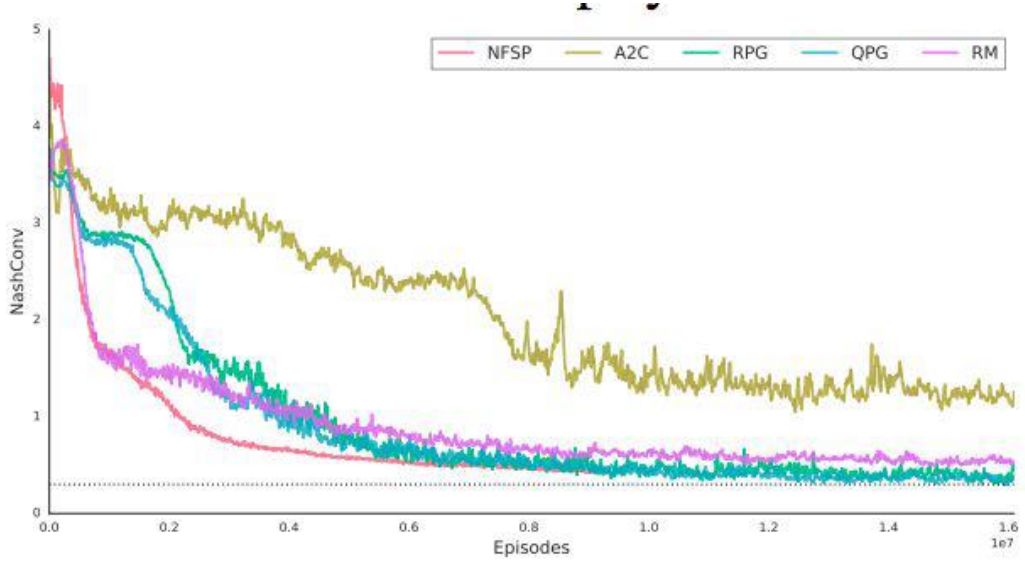
Image form Sandholm ‘10

# CFR is Policy Iteration

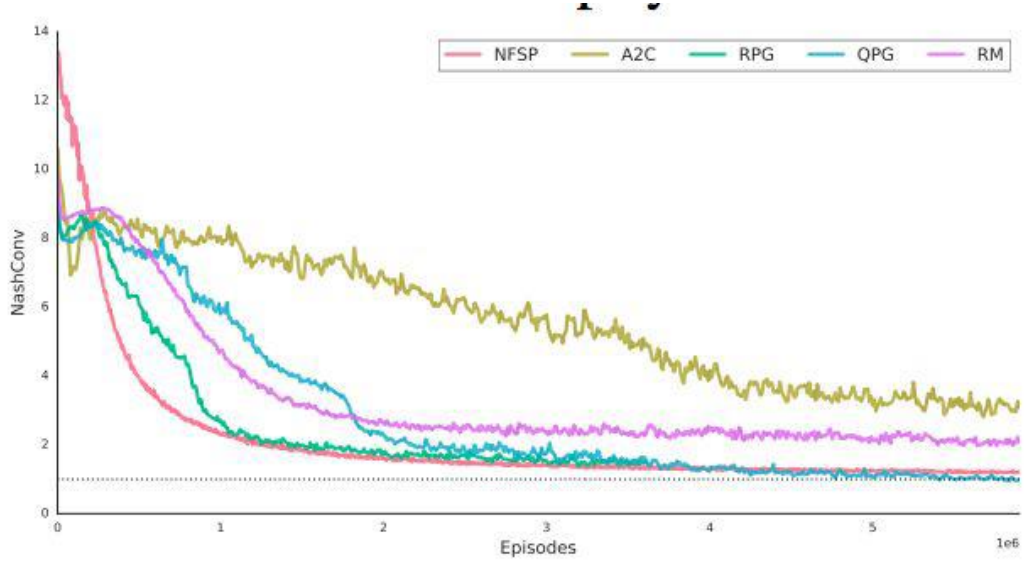
- Policy evaluation is analogous
- Policy improvement: use regret minimization algorithms
  - Average strategies converge to Nash in self-play
- Convergence guarantees are on the average policies

# Regret Policy Gradients (Srinivasan et al. '18)

- Policy gradient is doing a form of CFR minimization!
- Several new policy gradient variants inspired connection to regret




NASHCONV in 2-player Leduc




NASHCONV in 3-player Leduc

# Counterfactual Regret Minimization

- Multi-agent, partial information, competition
- Algorithm: Counterfactual regret minimization
- Minimize the regret of not having taken the right action, playing many “what-ifs” (counterfactuals)
- CFR is probabilistic multi-agent version of competitive minimax
- Works quite well in Poker
- Complicated code, see paper

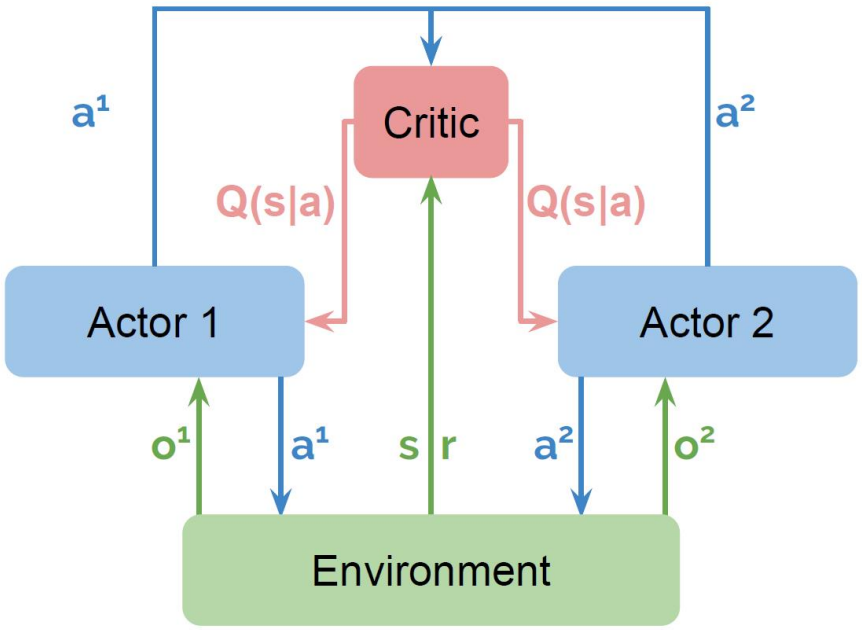
# Poker


LIBRATUS


Genome engineering on a large scale pp. 849 & 922
Monkeyflower research begins to bloom p. 854
Protecting elections from social media manipulation p. 858
Science
$15
30 AUGUST 2019
sciencemag.org
AAAS
PLYR 1
PLYR 2
PLYR 3
PLYR 4
FOLD
CALLING OUR
BLUFF
AI masters multiplayer poker
pp. 864 & 885
Pluribus

# Centralized Critic Decentralized Actor Approaches

- **Idea:** reduce nonstationarity & credit assignment issues using a central critic
- **Examples:** MADDPG [Lowe et al., 2017] & COMA [Foerster et al., 2017]
- Apply to both cooperative and competitive games




**Centralized critic trained to minimize loss:**

$$\mathcal{L}(\theta_i) = \mathbb{E}_{\mathbf{x},a,r,\mathbf{x}'}[(Q_i^{\boldsymbol{\pi}}(\mathbf{x}, a_1, \ldots, a_N) - y)^2],$$

$$y = r_i + \gamma Q_i^{\boldsymbol{\pi}'}(\mathbf{x}', a'_1, \ldots, a'_N)\big|_{a'_j = \boldsymbol{\pi}'_j(o_j)}$$

**Decentralized actors trained via policy gradient:**

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s\sim p^{\boldsymbol{\mu}}, a_i \sim \boldsymbol{\pi}_i}[\nabla_{\theta_i} \log \boldsymbol{\pi}_i(a_i|o_i) Q_i^{\boldsymbol{\pi}}(\mathbf{x}, a_1, ..., a_N)]$$

Actor

Critic

# Deep Deterministic Policy Gradient (DDPG)

- A model-free off-policy actor-critic algorithm, combining DPG and DQN.
- DQN stabilizes the learning of Q-function by experience replay and the frozen target network.
- DQN works in discrete space, and DDPG extends it to continuous space with the actor-critic framework while learning a deterministic policy.

**Algorithm 1** DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
**for** episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  **for** t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

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

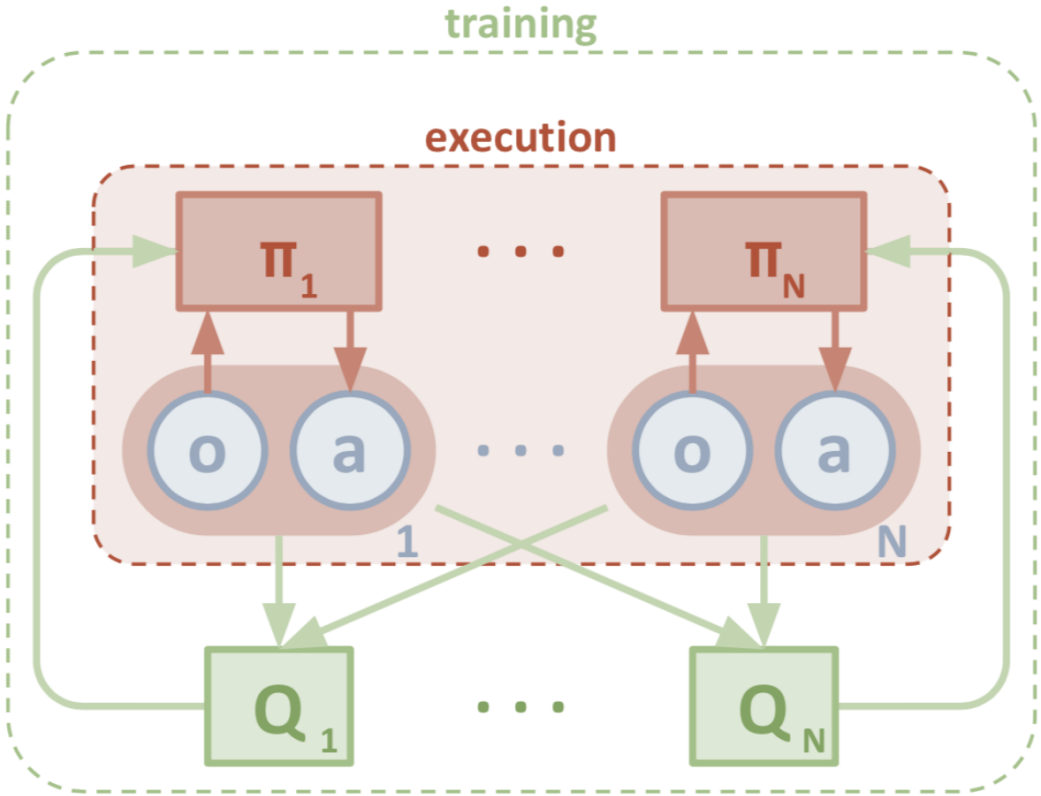
    Update the target networks:

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

  **end for**
**end for**

# Multi-Agent Deep Deterministic Policy Gradient

- **Multi-agent DDPG (MADDPG)** (Lowe et al., 2017) extends DDPG to an environment where multiple agents are coordinating to complete tasks with only local information.
- In the viewpoint of one agent, the environment is non-stationary as policies of other agents are quickly upgraded and remain unknown.
- MADDPG is an actor-critic model redesigned particularly for handling such a changing environment and interactions between agents.
  - Centralized critic + decentralized actors, changes non-stationary problems to stationary problems
  - Actors can use estimated policies of other agents for learning
  - Policy ensembles is good for reducing variance



**Algorithm 1:** Multi-Agent Deep Deterministic Policy Gradient for $N$ agents

**for** episode $= 1$ to $M$ **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  **for** $t = 1$ to max-episode-length **do**
    for each agent $i$, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward $r$ and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    **for** agent $i = 1$ to $N$ **do**
      Sample a random minibatch of $S$ samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma\, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)\big|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

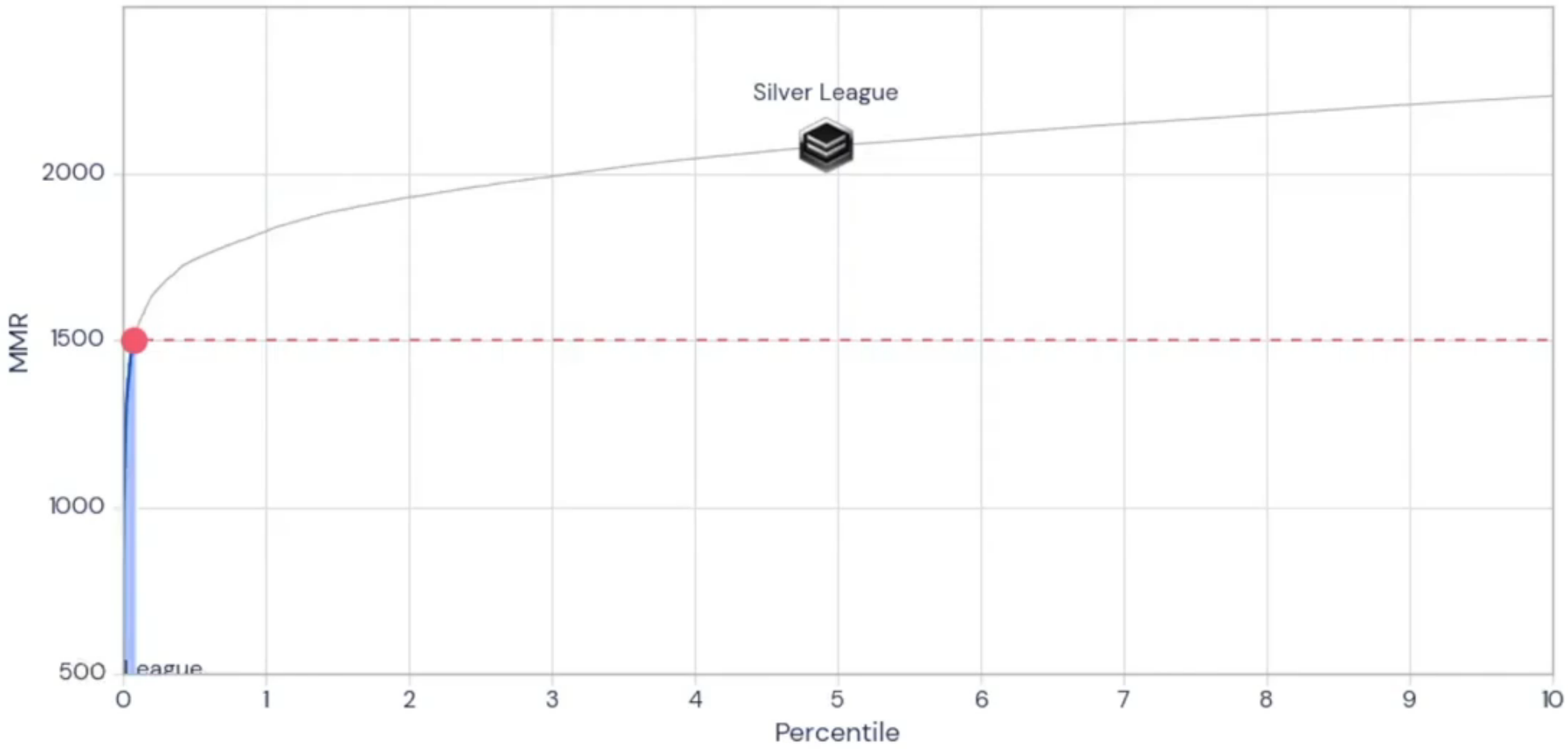
$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    **end for**
    Update target network parameters for each agent $i$:

$$\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$$

  **end for**
**end for**

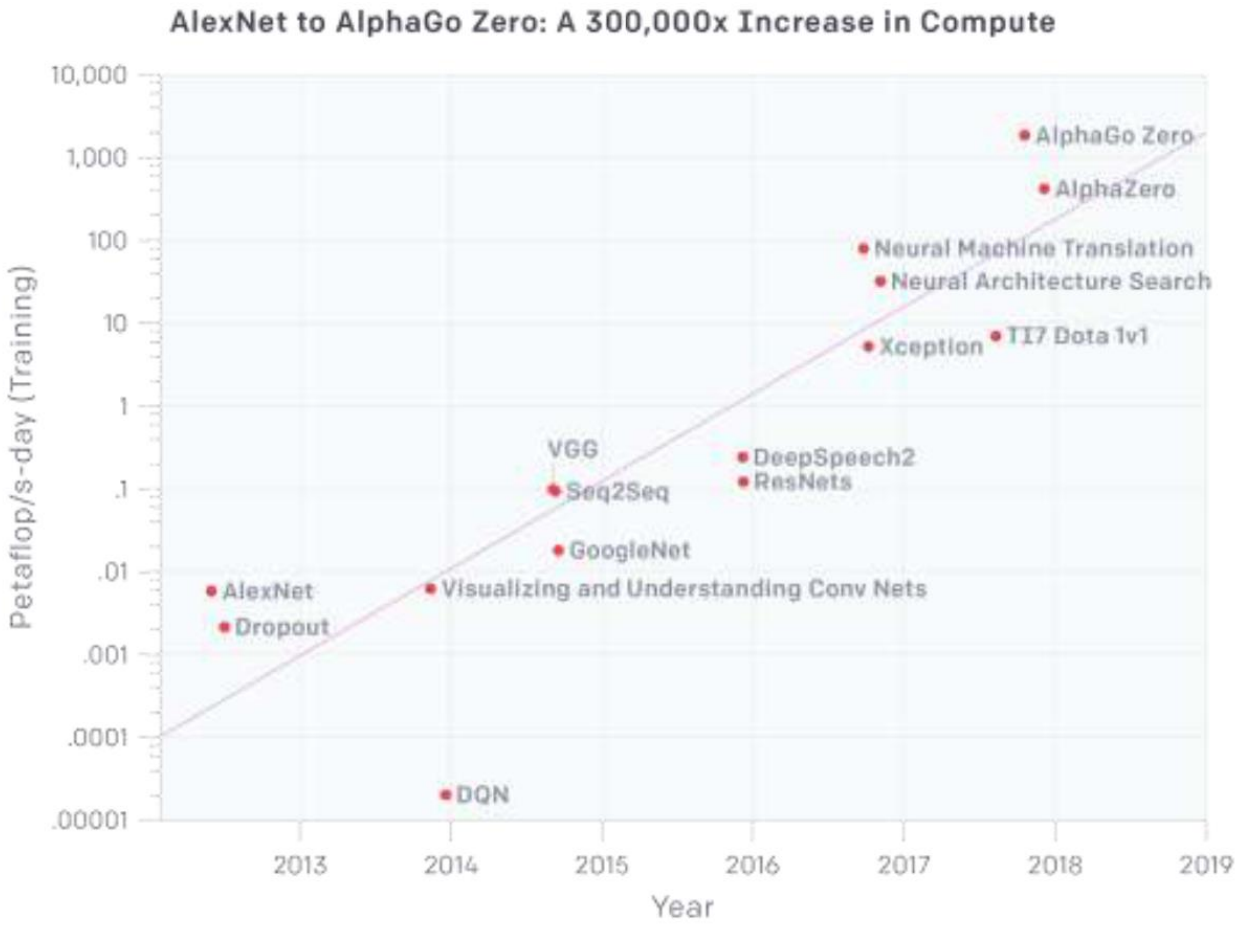
# AlphaStar




https://deepmind.google/discover/blog/alphastar-grandmaster-level-in-starcraft-ii-using-multi-agent-reinforcement-learning/

# Challenges in Multi-Agent Learning

- Computational complexity
  - AlphaGo Zero (per agent):
    - 64 GPUs & 19 CPUs
  - OpenAI Dota Five
    - 256 GPUs & 128000 CPUs
- Lack of good benchmarks
- Reproducability



**TDDE13 MAS LE3 HT2024:**
**Reinforcement learning**
**Deep reinforcement learning**
**Multi-agent learning**

# www.ida.liu.se/~TDDE13

LINKÖPING UNIVERSITY