TDDE13 LE3 HT2023

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

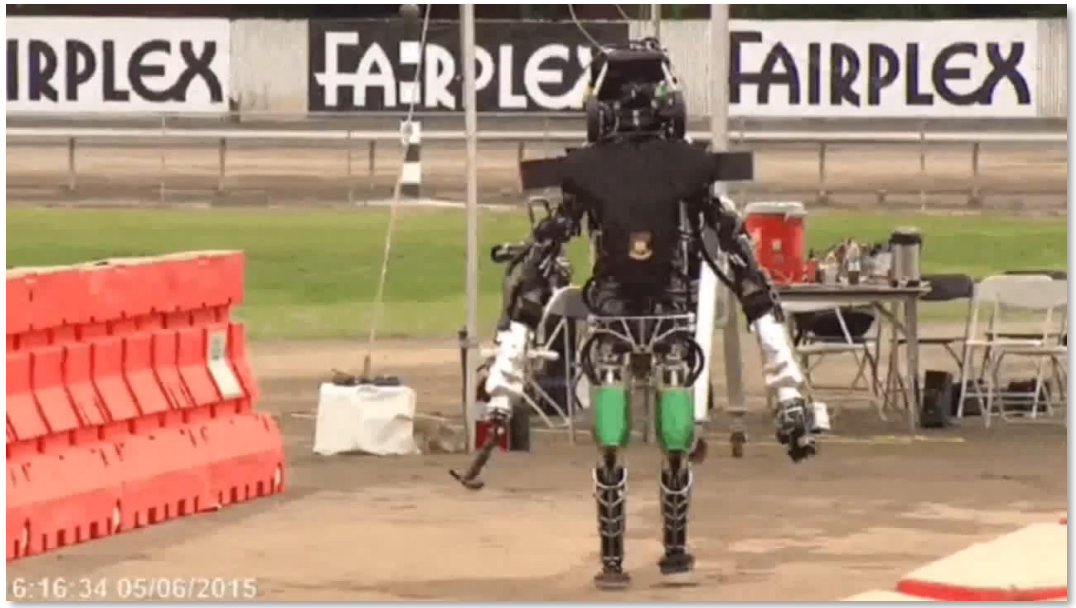
Data: state-action pairs

Goal: Maximize future rewards over many time steps

Apple example:

Eat this thing because it will keep you alive.

From Supervised to Reinforcement Learning - Learning How to Act



Humorous reminder from IEEE Spectrum: The DARPA 2015 Humanoid Challenge “Fail Compilation”
To be fair, this is the state of the art:
https://youtu.be/NR32ULxbjYc

- Can we use supervised learning to **learn** how to act?
- **E.g. engineering** robot behavior can be **fragile** and **time consuming**
  - Things humans do without thinking require **extremely detailed instructions** for a robot. Even robust locomotion is hard.

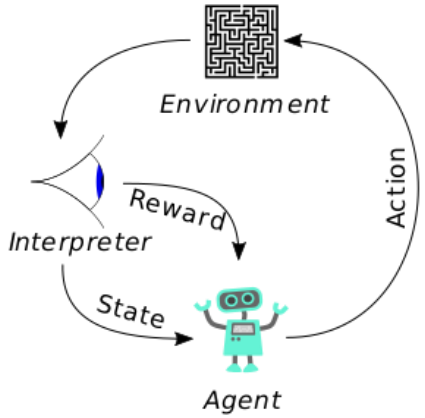
Learning How to Act

- Yes, one can learn a mapping from problem state (e.g. position) to action
  - As in all supervised learning, this requires a teacher
  - Sometimes called ”imitation learning”
- However, **supervised learning** with robots can get tedious as providing examples of correct behaviour is difficult to automate
- Can we remove the human from the loop?
  1. An **automated teacher** like a **planning or optimal control** algorithm can generate supervised examples **if it has a model of the environment**
     - Mordatch et al, https://www.youtube.com/watch?v=IxrnT0JOs40
     - LiU’s research with real nano-quadcopters (deep ANN on-board the microcontroller)
  2. Reinforcement learning attempts to generalize this to learning from scratch in completely **unknown environments**

Reinforcement Learning Basic Concept

- *Reinforcement Learning is learning what to do – how to map situations to actions – so as to maximum a numerical reward.*

  Reinforcement Learning: An introduction
  Sutton & Barto



- Rather than learning from explicit training data, or discovering patterns in static data, reinforcement learning discovers the best option (highest reward) from trial and error.
- Inverse Reinforcement Learning
  - Learn reward function by observing an expert
  - “Apprenticeship learning“
  - E.g. Abbeel et al. *Autonomous Helicopter Aerobatics through Apprenticeship Learning*

Reinforcement Learning: Key Concepts

Agent: takes actions.

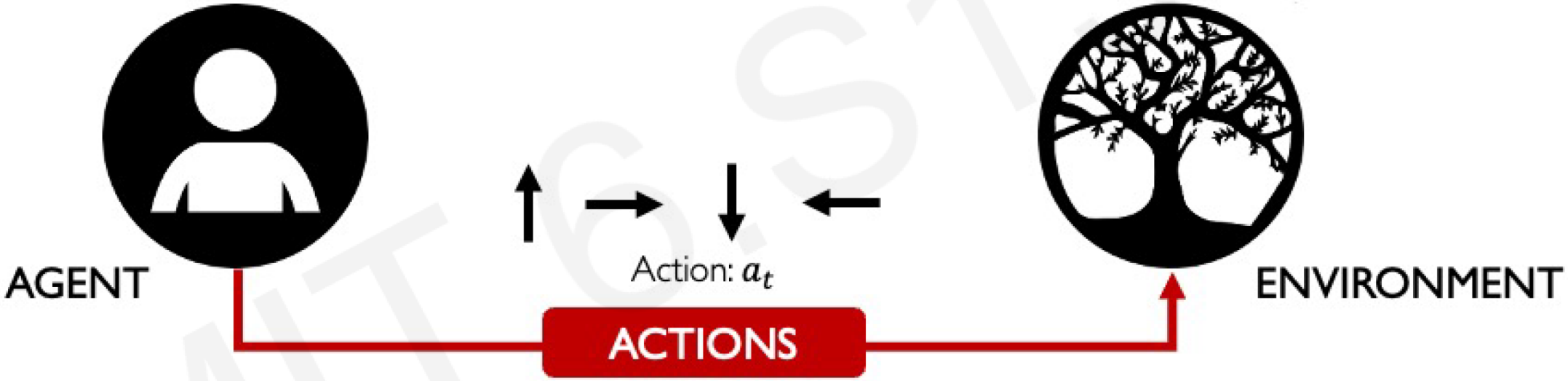
Reinforcement Learning: Key Concepts

AGENT

ENVIRONMENT

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

Reinforcement Learning: Key Concepts



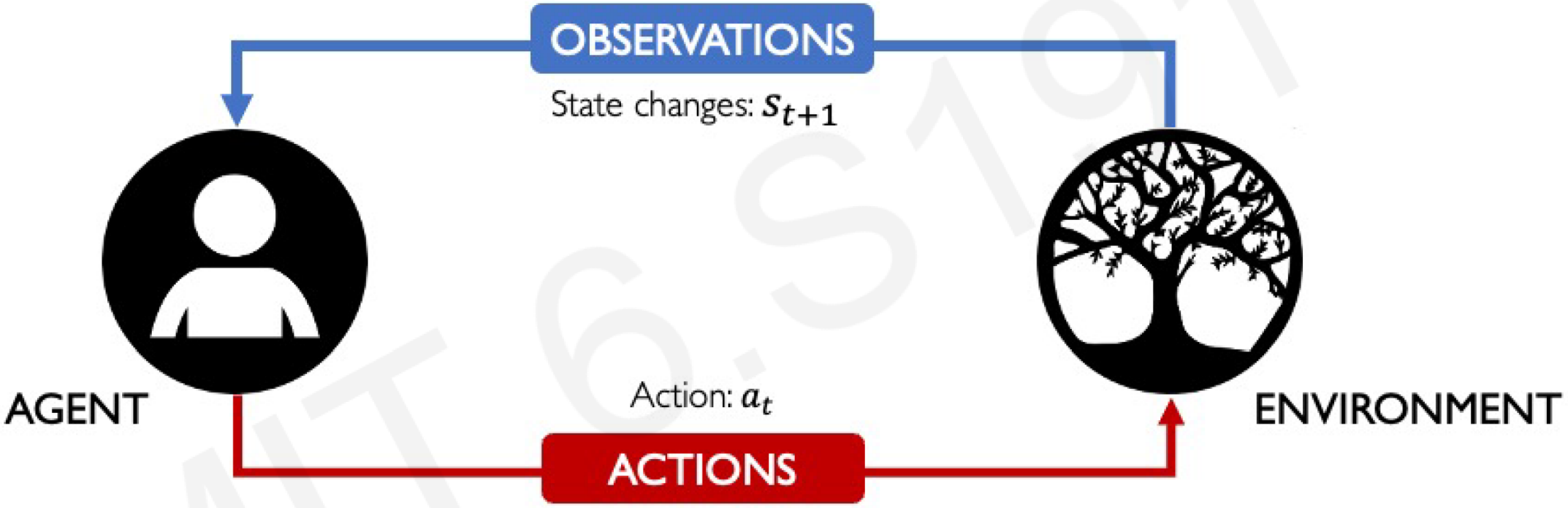
Action: a move the agent can make in the environment.

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.

Reinforcement Learning: Key Concepts

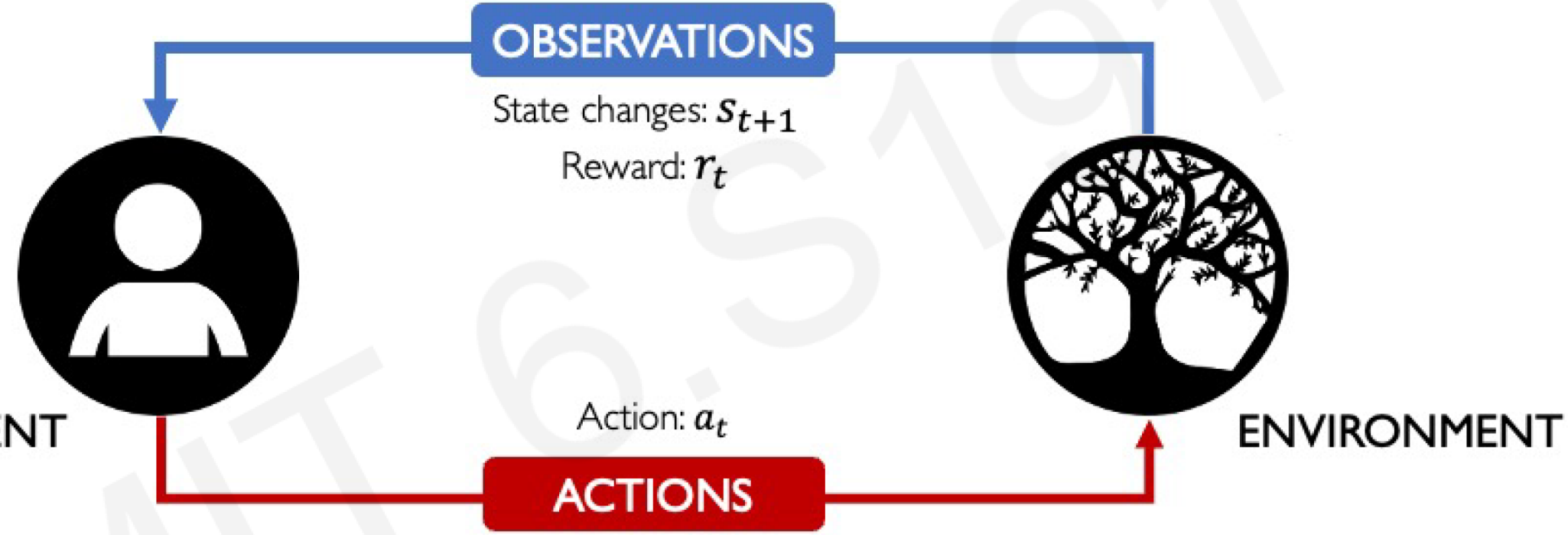


State: a situation which the agent perceives.

Reinforcement Learning: Key Concepts

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

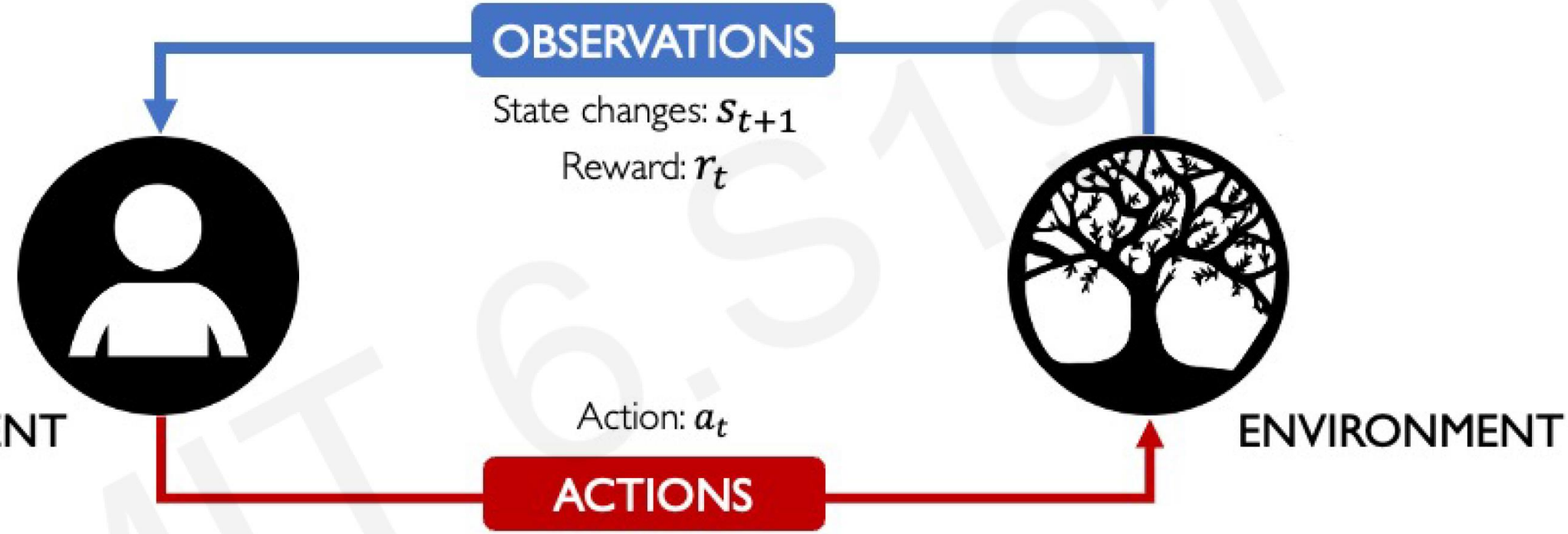
Reinforcement Learning: Key Concepts



Total Reward (Return)

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

Reinforcement Learning: Key Concepts



Total Reward (Return)

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

Reinforcement Learning: Key Concepts

Discounted Total Reward (Return)

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

Reinforcement Learning: Key Concepts

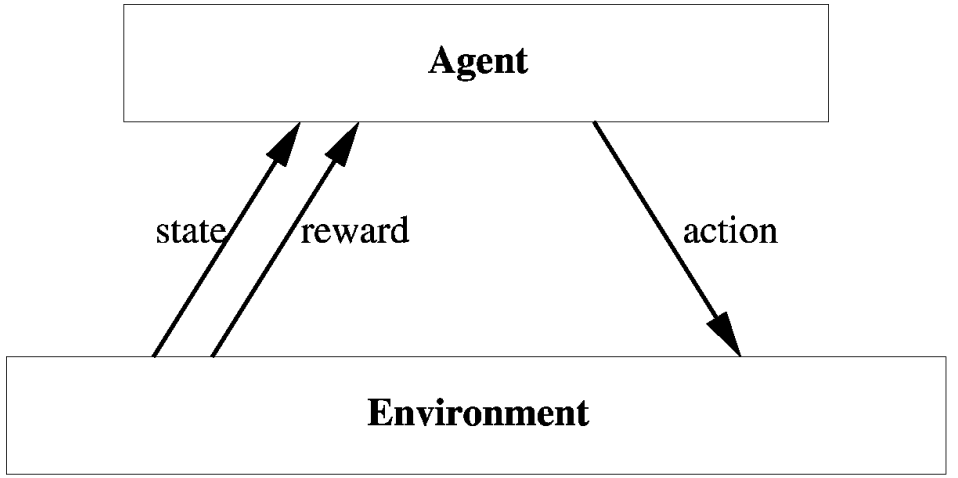
Discounted Total Reward (Return)

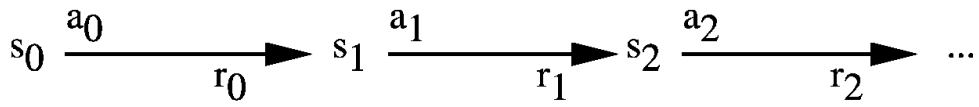
$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i = \gamma^t r_t + \gamma^{t+1} r_{t+1} \ldots + \gamma^{t+n} r_{t+n} + \cdots$$

γ: discount factor; $0 < \gamma < 1$

A Reinforcement Learning Problem

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



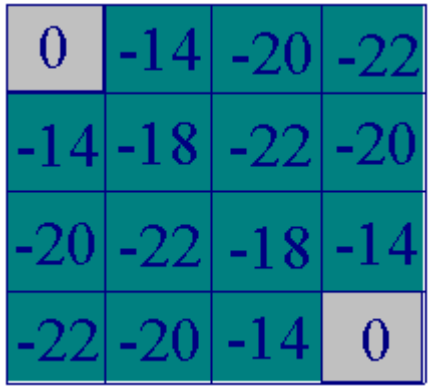


Goal: Learn to choose actions that maximize

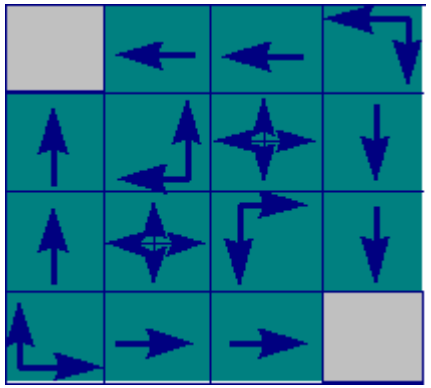
$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$, where $0<\gamma<1$

RL Value Function - Example

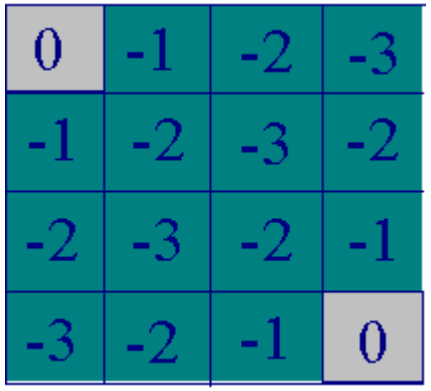
A minimum time to goal world



Value function for random movement



Optimal policy



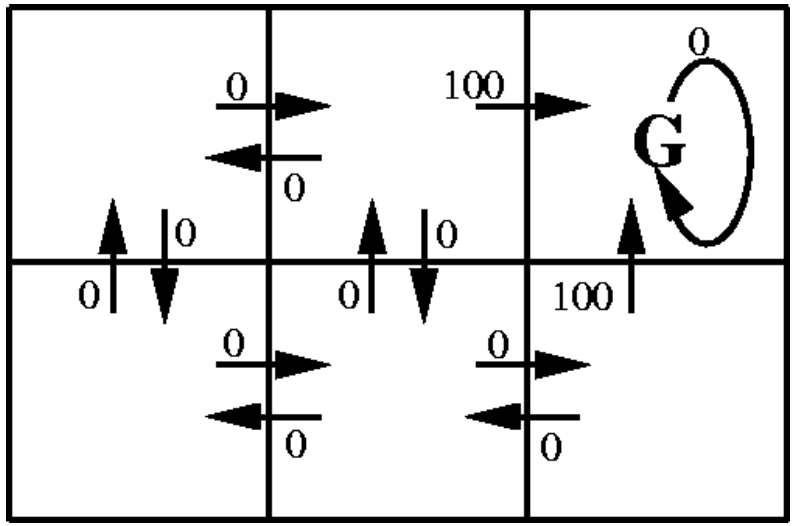
Optimal value function

Markov Decision Processes

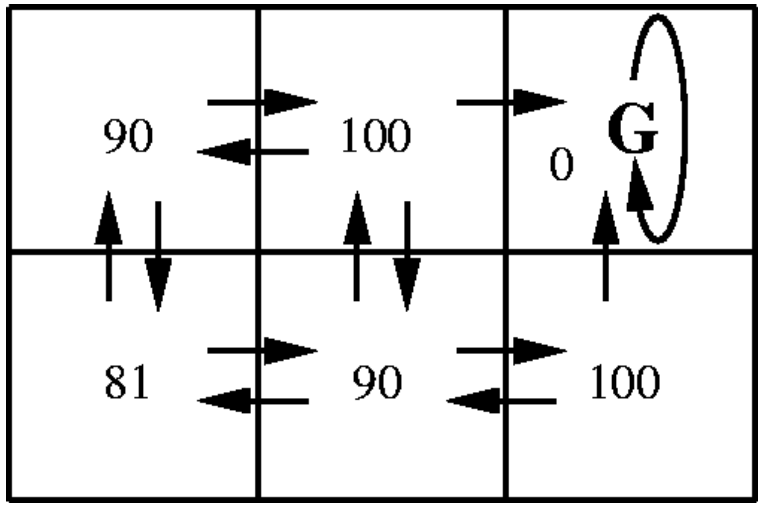
Assume:

- 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 δ and r may be non-deterministic
  - functions δ and r not necessarily known to the agent

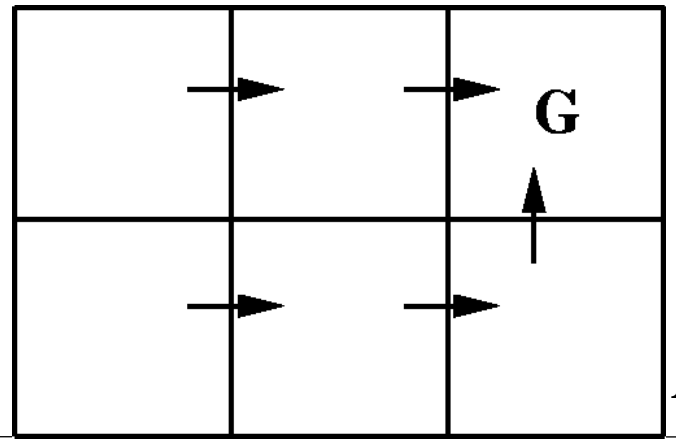
MDP Example



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

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

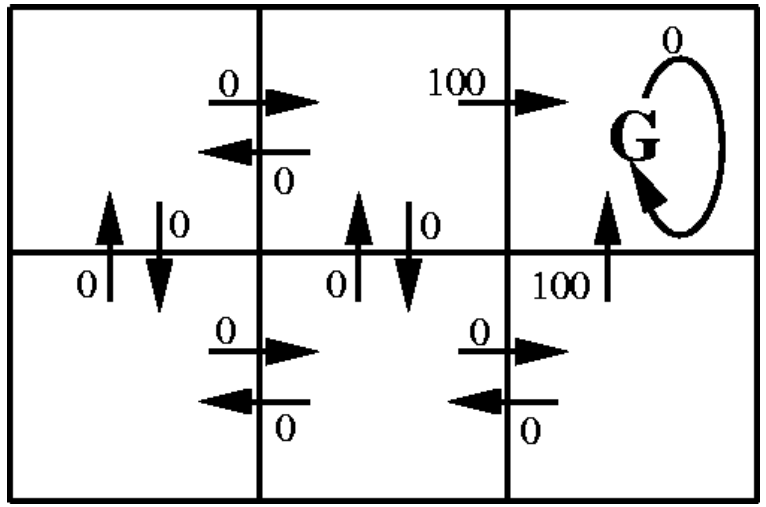
The Q-Function

Optimal policy:

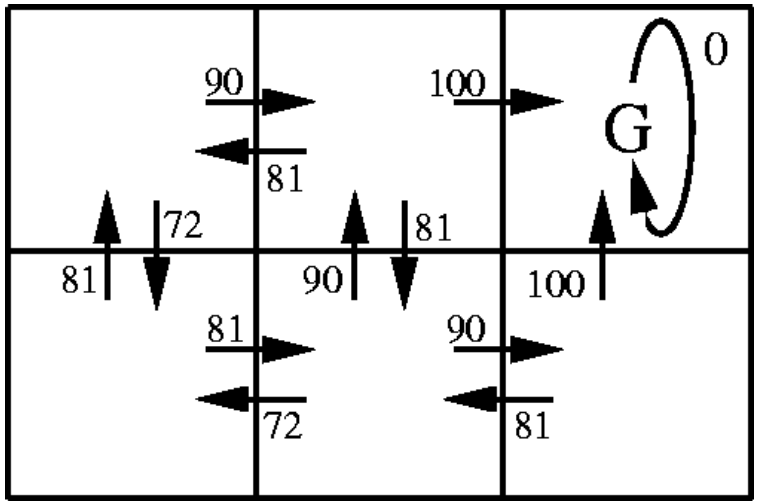
- $\pi^*(s) = \text{argmax}_a[r(s,a) + \gamma V^*(\delta(s,a))]$
- Doesn't work if we don't know r and δ.

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

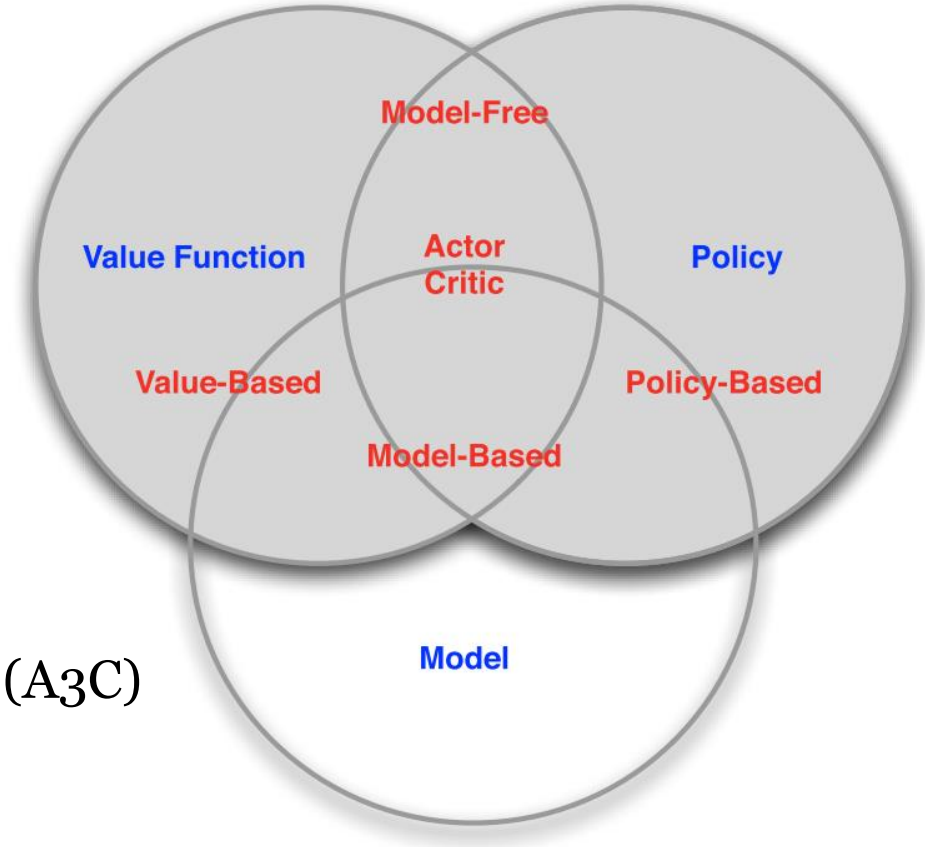
- Note Q and V* closely related:
  $V^*(s) = \max_{a'} Q(s,a')$
- Therefore Q can be written as:

  $$Q(s_t,a_t) := r(s_t,a_t) + \gamma V^*(\delta(s_t,a_t)) =$$
  $$r(s_t,a_t) + \gamma \max_{a'} Q(s_{t+1},a')$$
- 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 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)



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

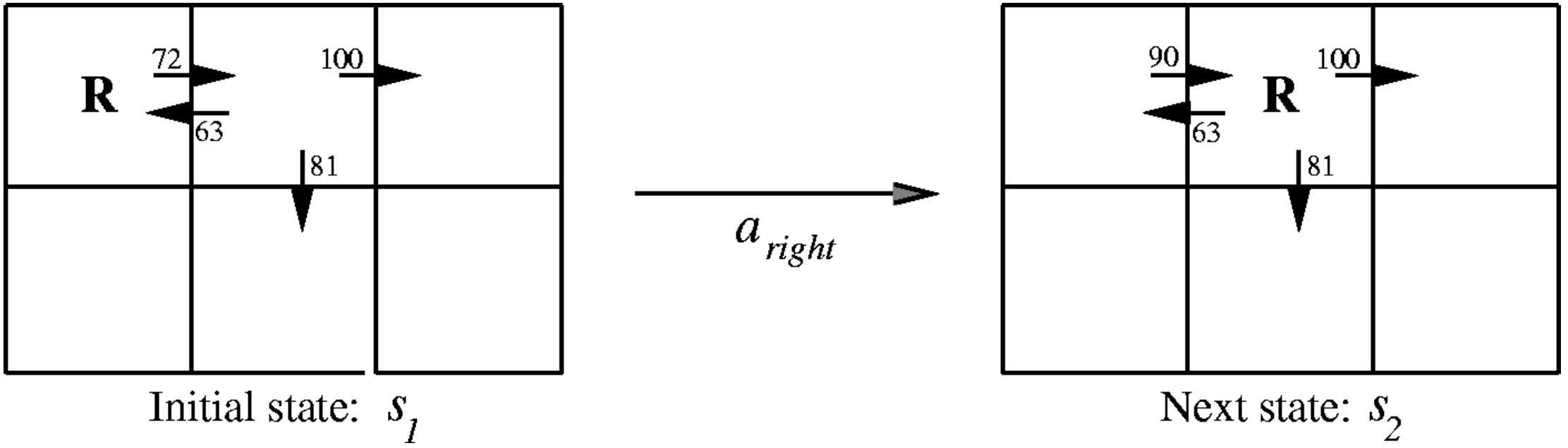
For each s, a initialize table entry $\hat{Q}(s,a) := 0$.

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



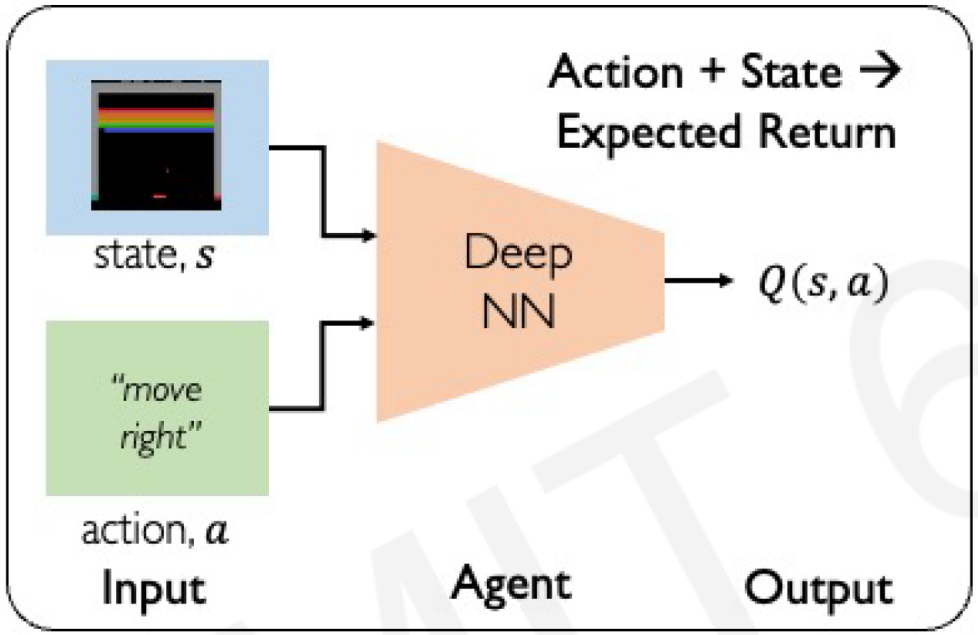
$$Q\hat{}(s_1, a_{right}) := r + \gamma \max_{a'} Q\hat{}(s_2, a')$$
$$:= 0 + 0.9 \max\{63, 81, 100\}$$
$$:= 90$$

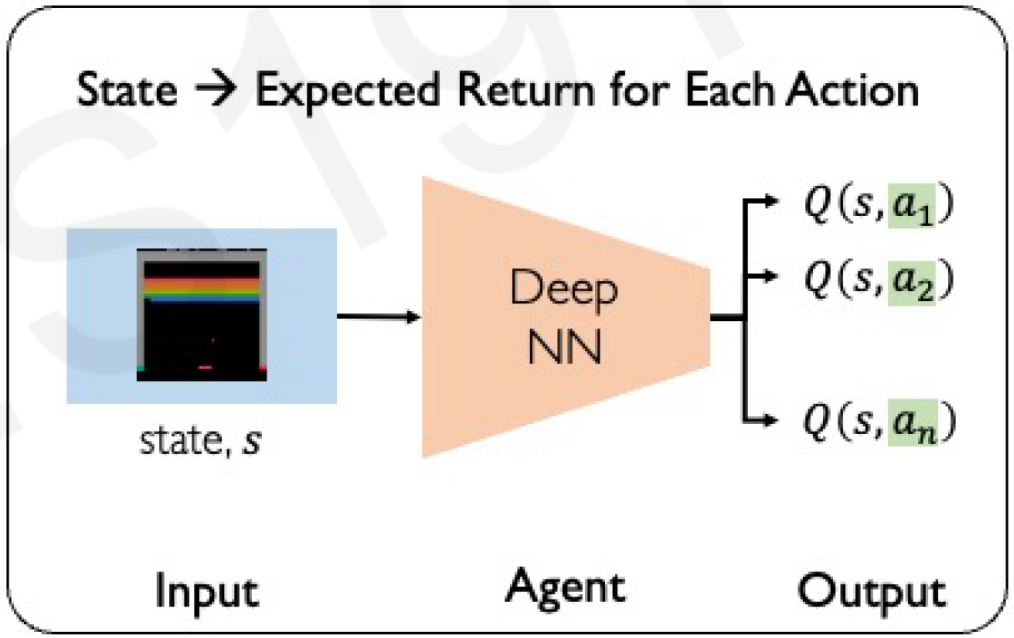
Q-Learning Continued

- 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)

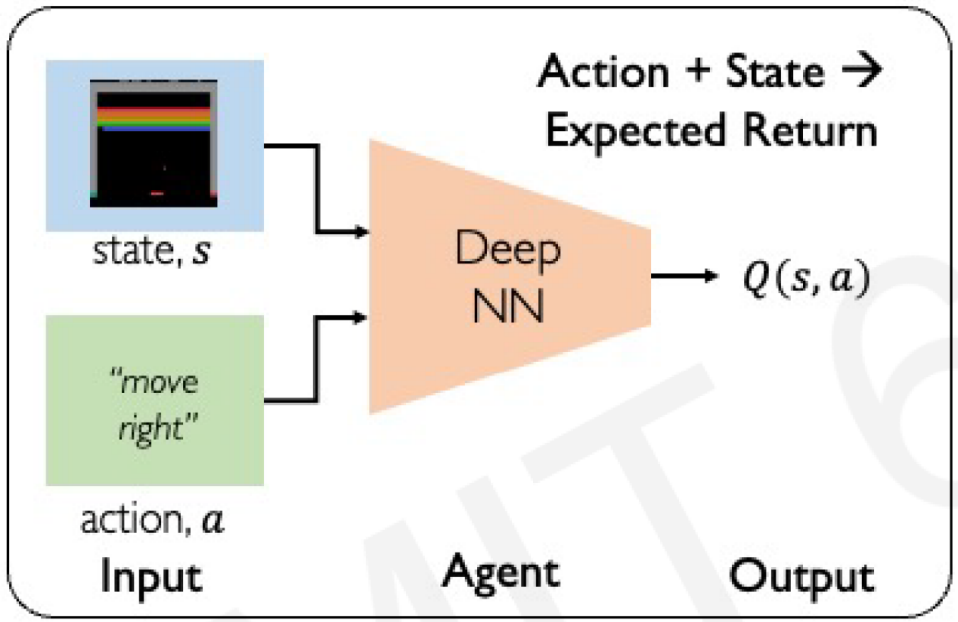
How can we use deep neural networks to model Q-functions?

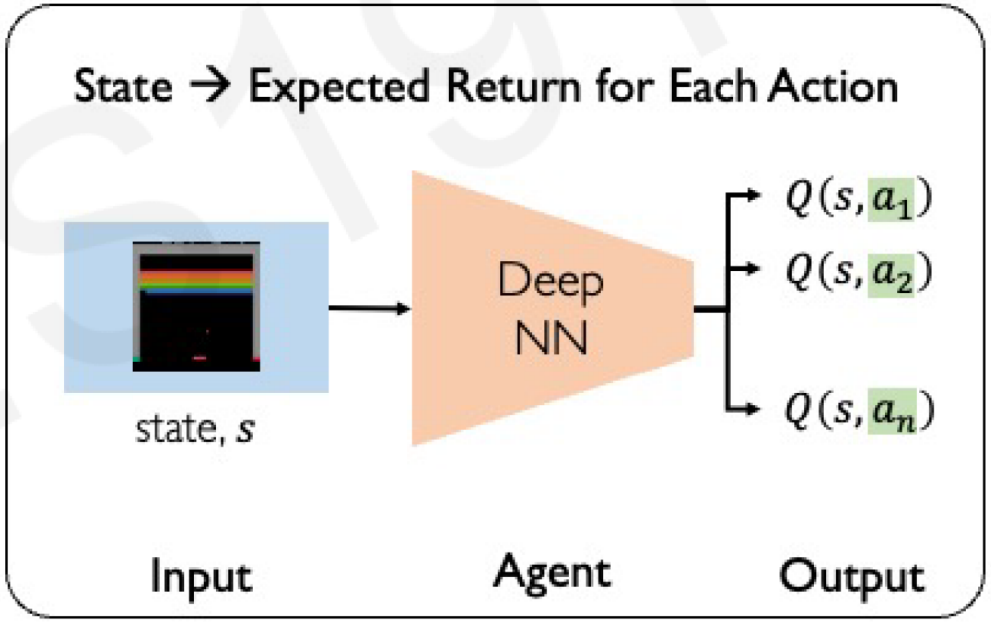




Deep Q Networks (DQN): Training

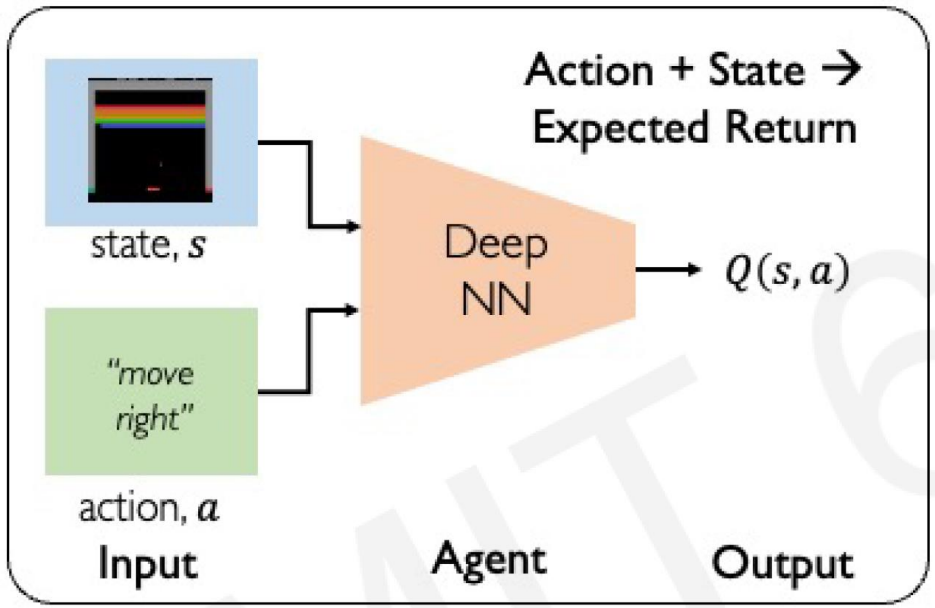
How can we use deep neural networks to model Q-functions?

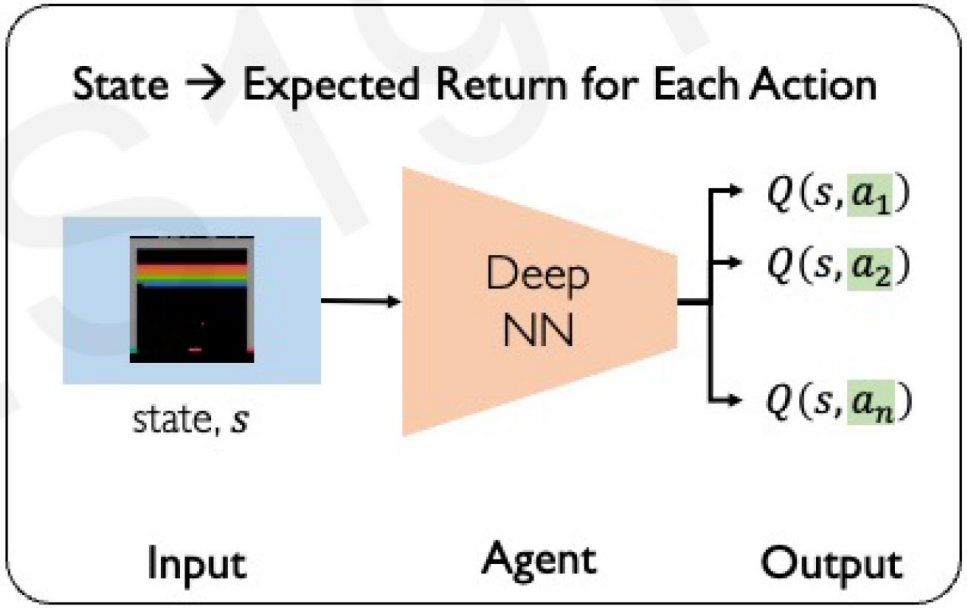




Deep Q Networks (DQN): Training

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



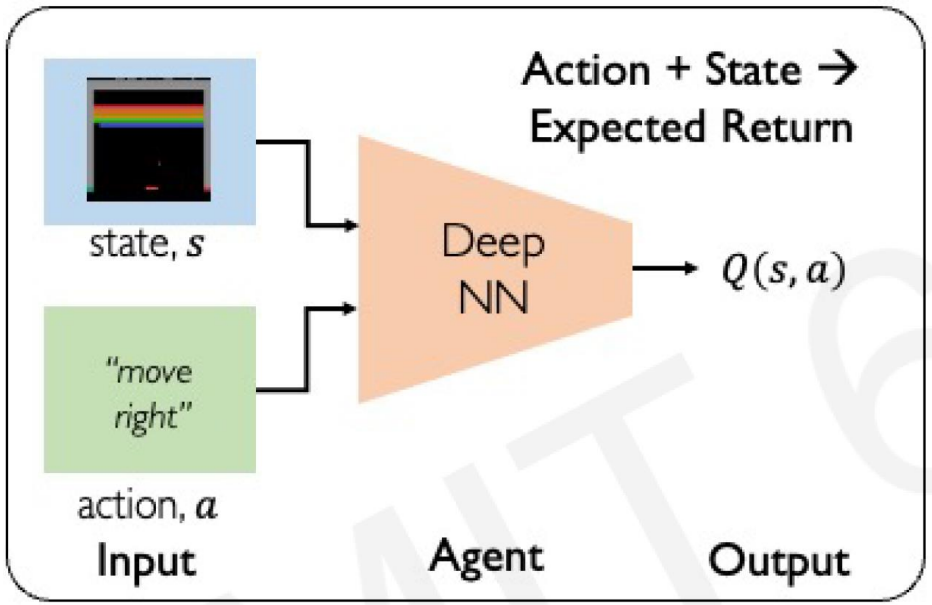


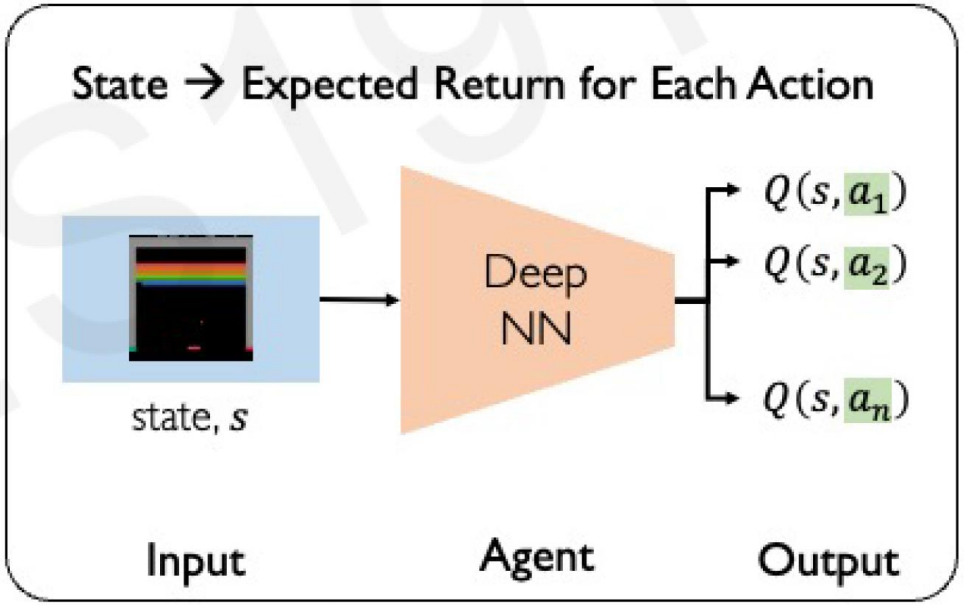
$$\underbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}_{\text{target}}$$

Take all the best actions → target return

Deep Q Networks (DQN): Training

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

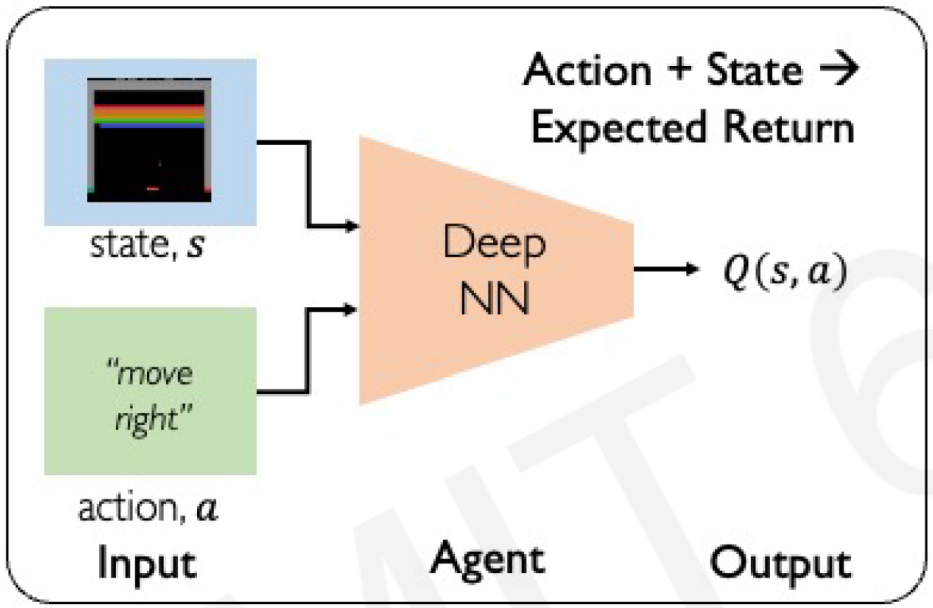


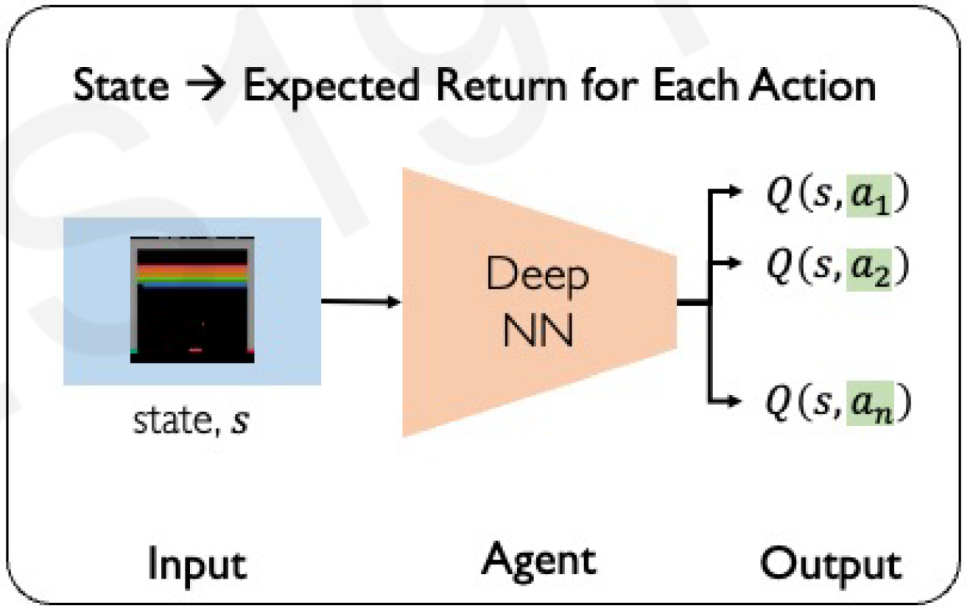


$$\underbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}_{\text{target}} \quad \underbrace{Q(s,a)}_{\text{predicted}}$$

Deep Q Networks (DQN): Training

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

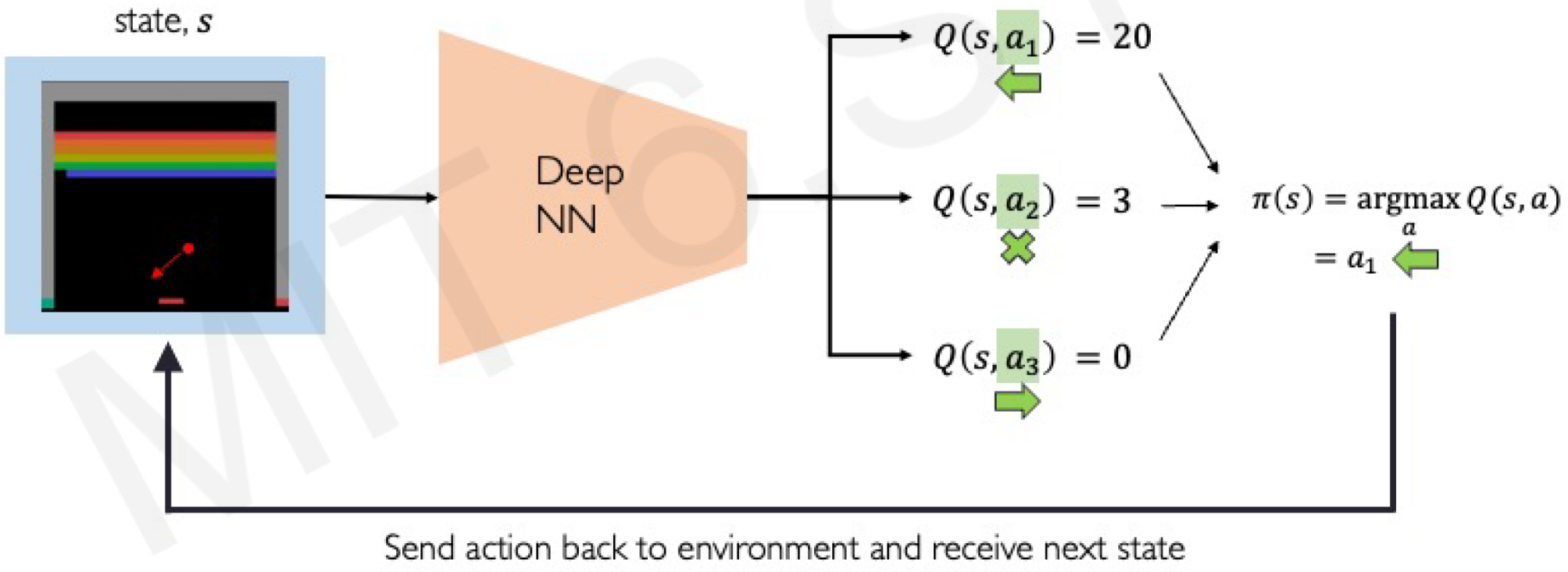




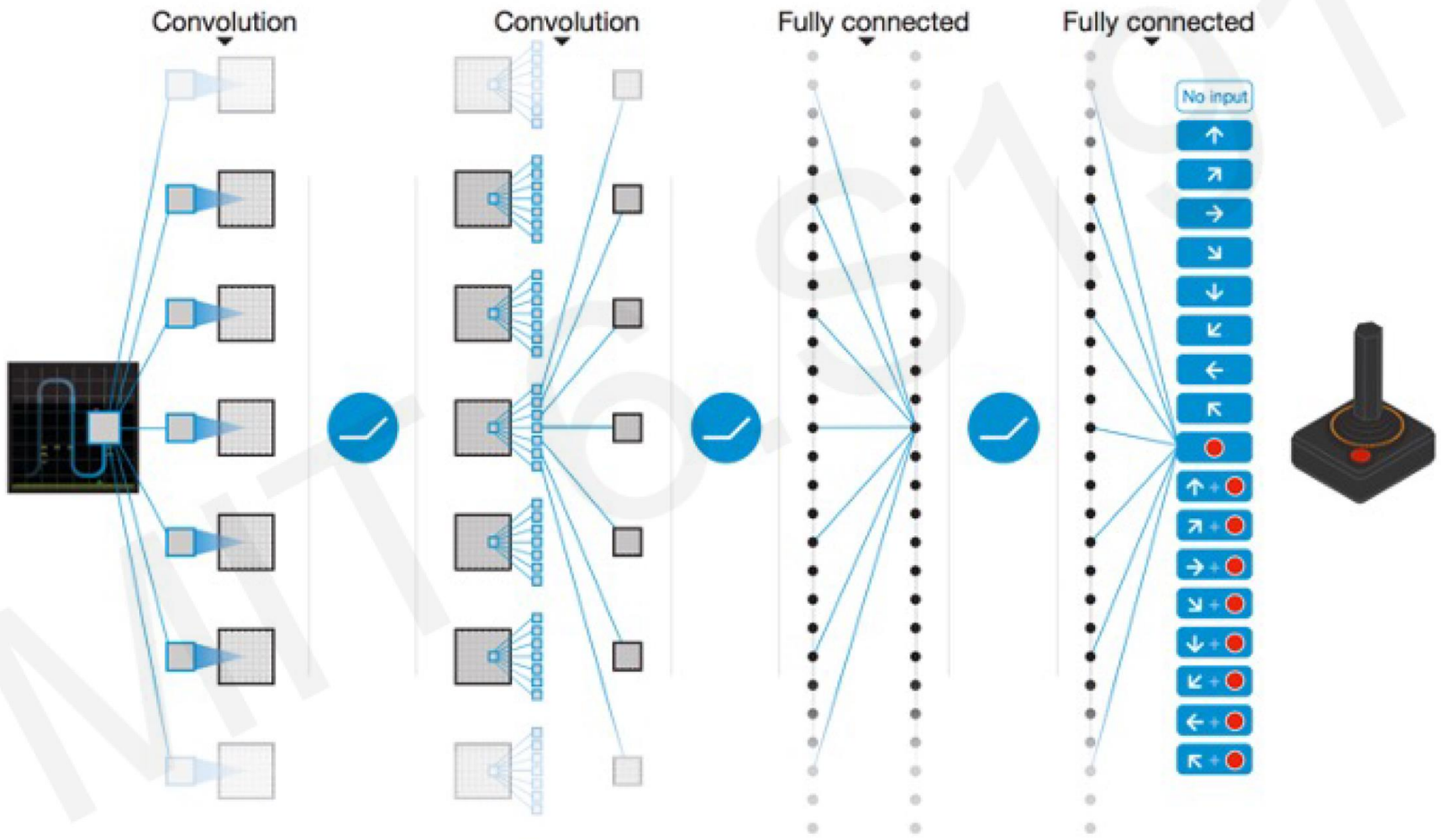
$$\mathcal{L} = \mathbb{E}\left[\left\|\overbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}^{\text{target}} - \overbrace{Q(s,a)}^{\text{predicted}}\right\|^2\right] \quad \textbf{Q-Loss}$$

Deep Q Network Summary

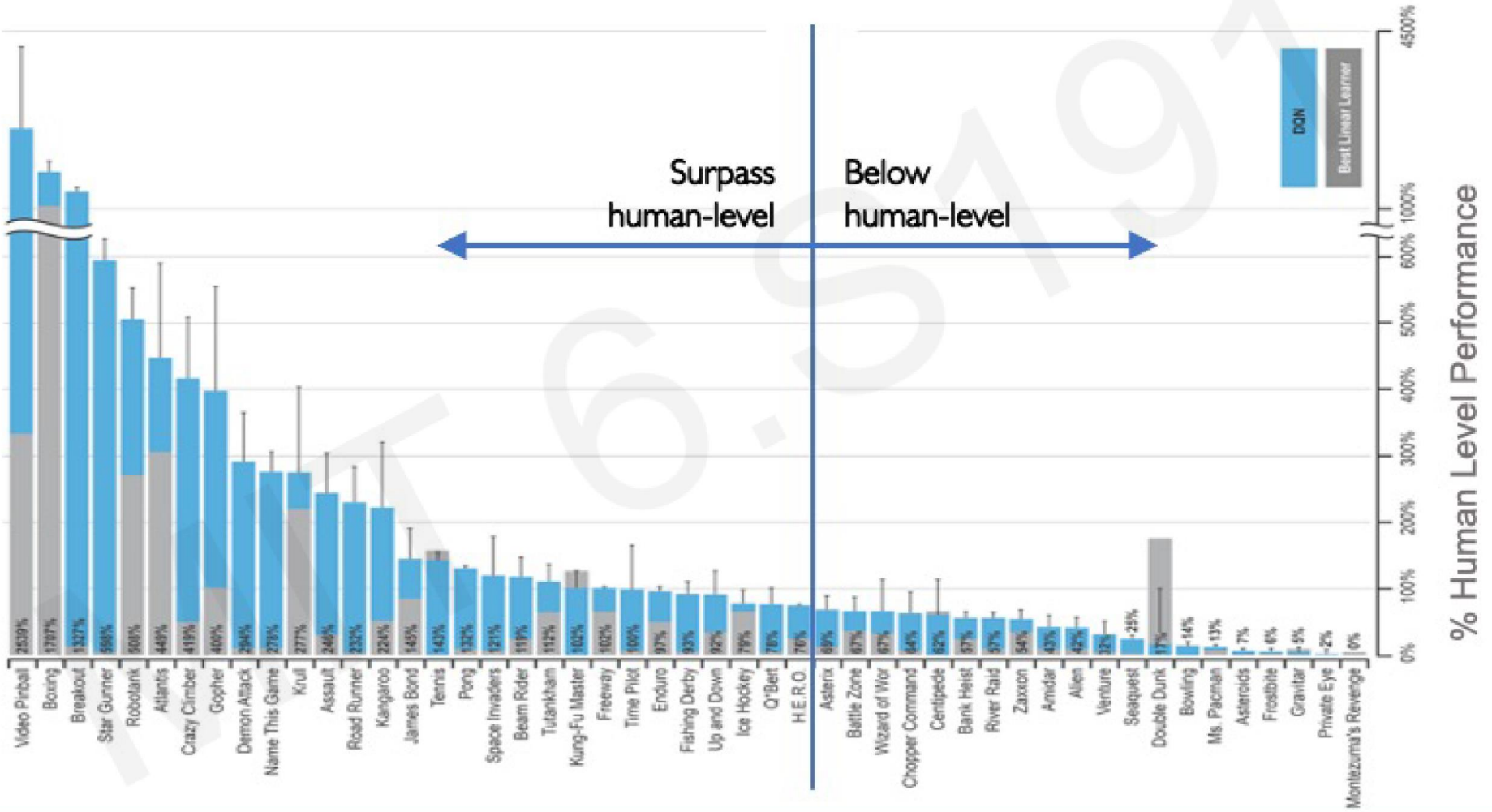
Use NN to learn Q-function and then use to infer the optimal policy, $\pi(s)$



DQN Atari Results



DQN Atari Results



Downsides of Q-Learning

Complexity:

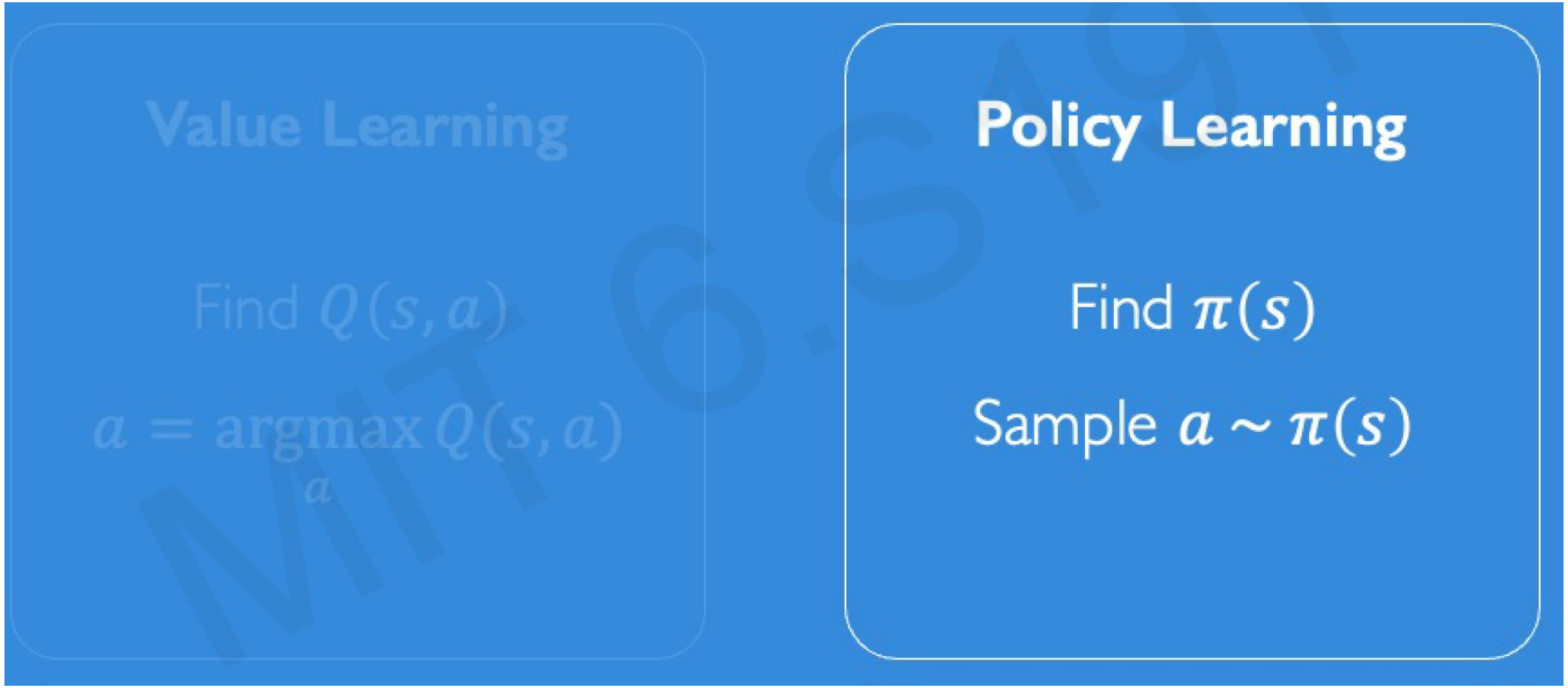
- Can model scenarios where the action space is discrete and small
- Cannot handle continuous action spaces

Flexibility:

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

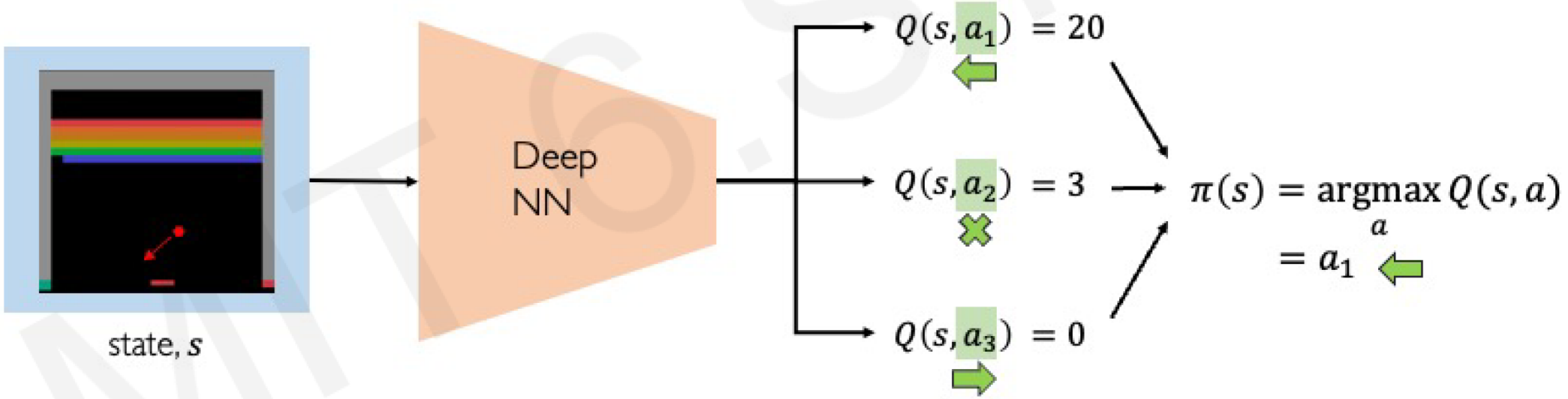
To address these, consider a new class of RL training algorithms: Policy gradient methods

Reinforcement Learning Algorithms



Deep Q Networks

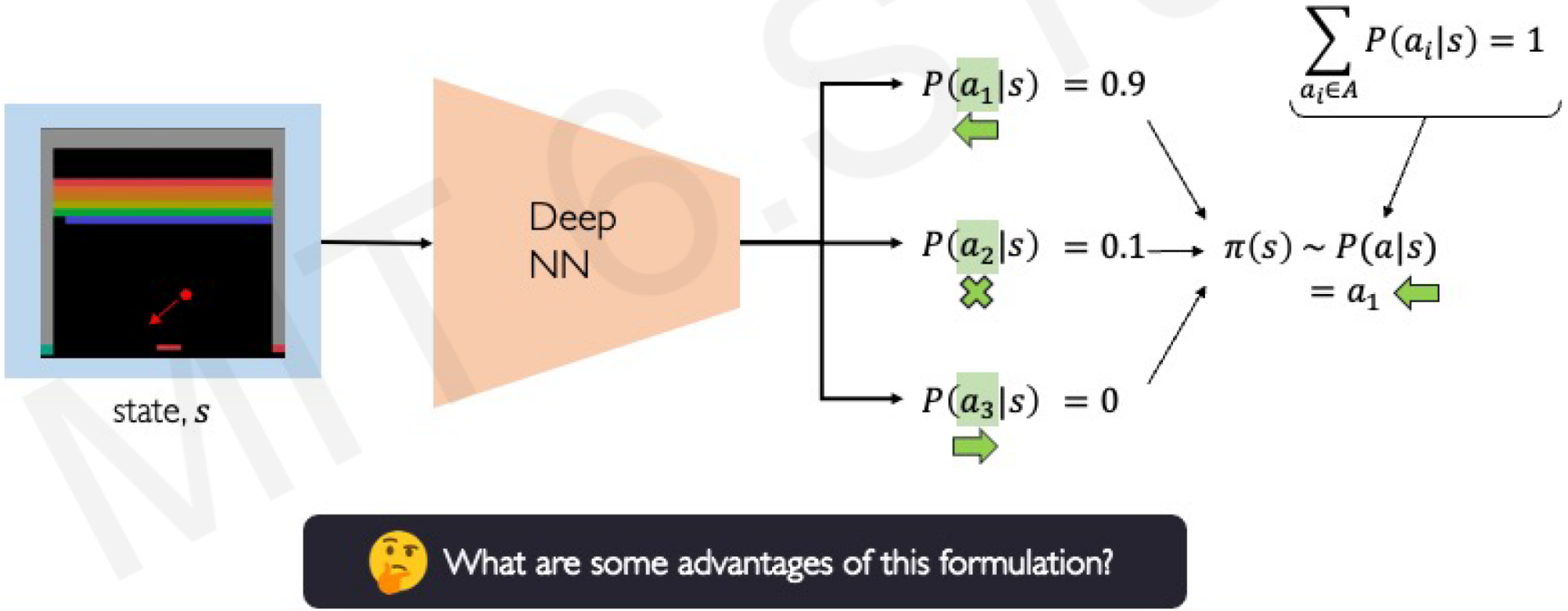
DQN: Approximate Q-function and use to infer the optimal policy, $\pi(s)$



Policy Gradient (PG): Key Idea

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

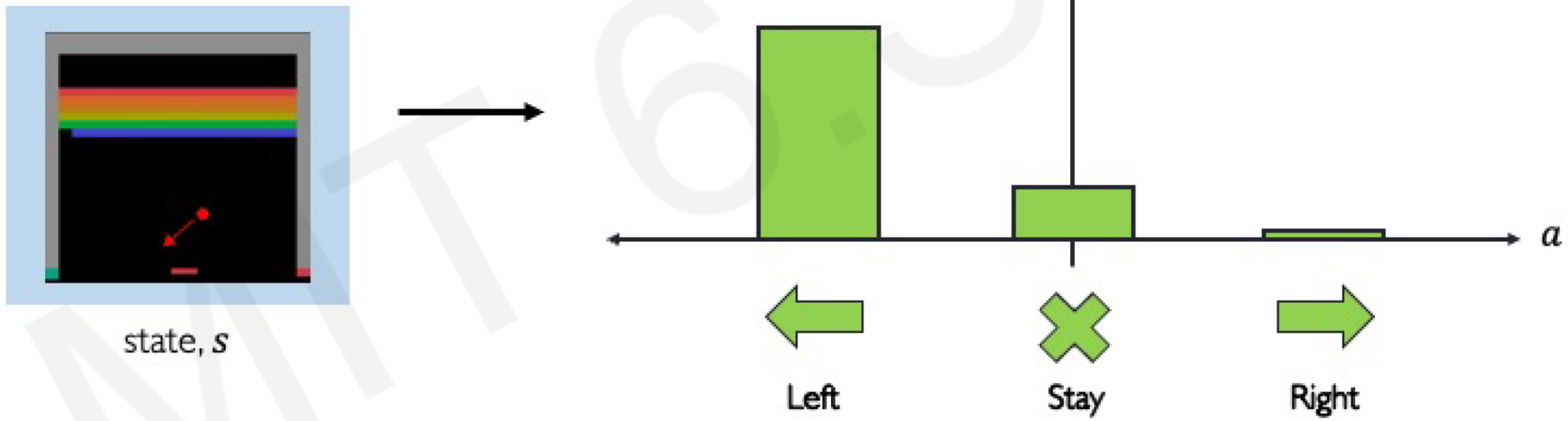
Policy Gradient: Directly optimize the policy $\pi(s)$



What are some advantages of this formulation?

Discrete vs Continuous Action Spaces

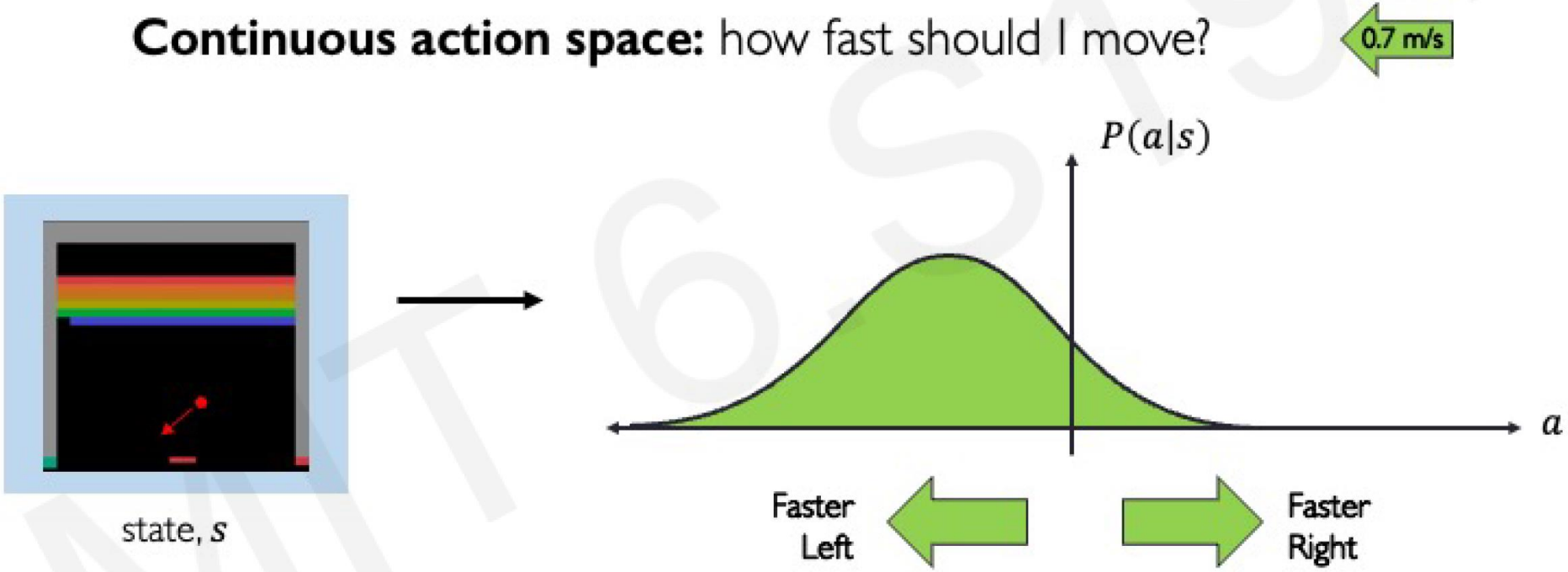
Discrete action space: which direction should I move?



Discrete vs Continuous Action Spaces

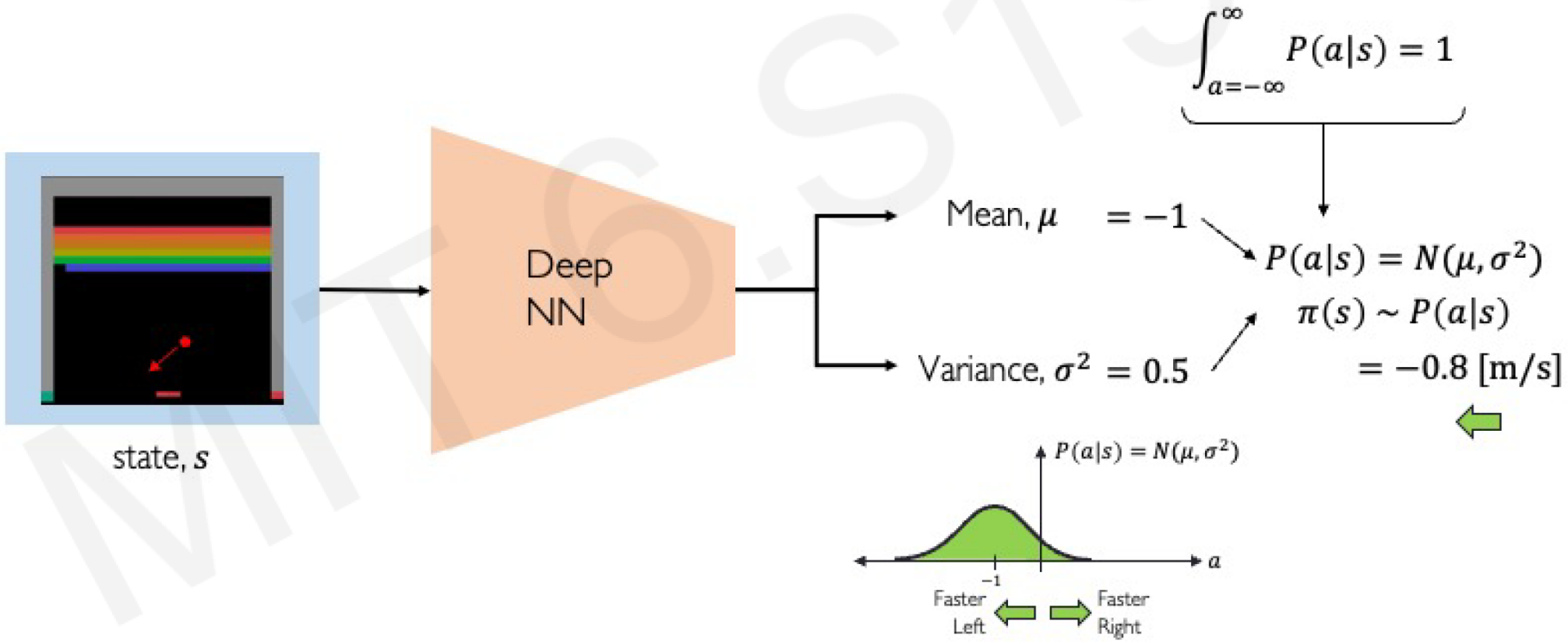
Discrete action space: which direction should I move?

Continuous action space: how fast should I move?



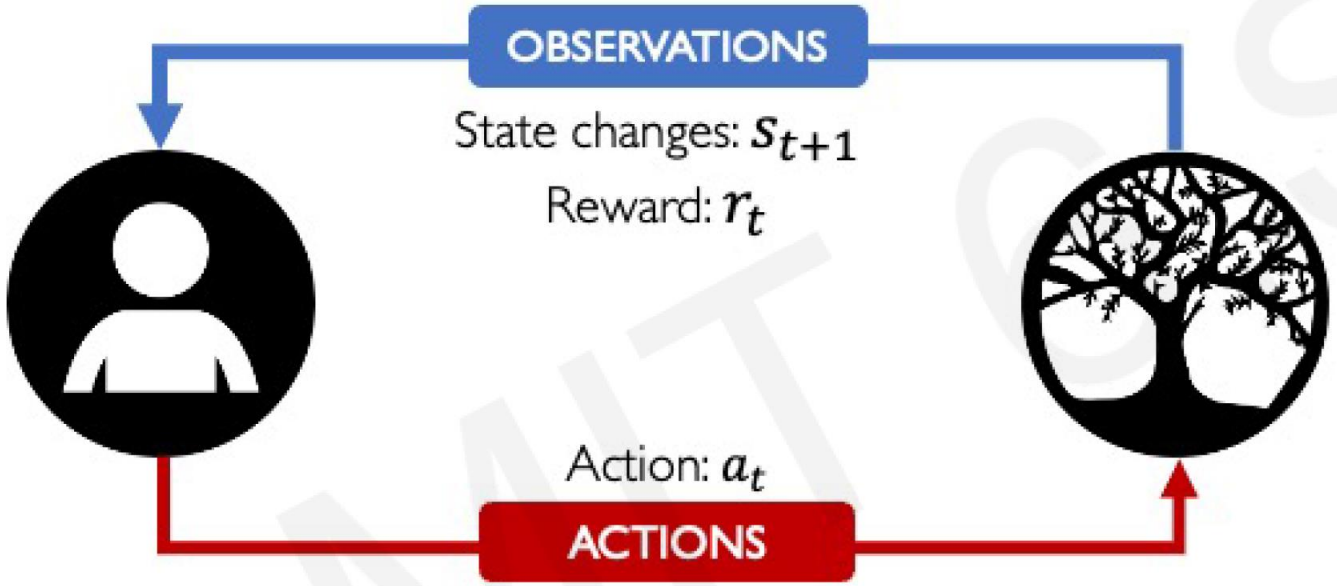
Policy Gradient (PG): Key Idea

Policy Gradient: Enables modeling of continuous action space



Training Policy Gradients: Case Study

Reinforcement Learning Loop:



Case Study – Self-Driving Cars

Agent: vehicle

State: camera, lidar, etc

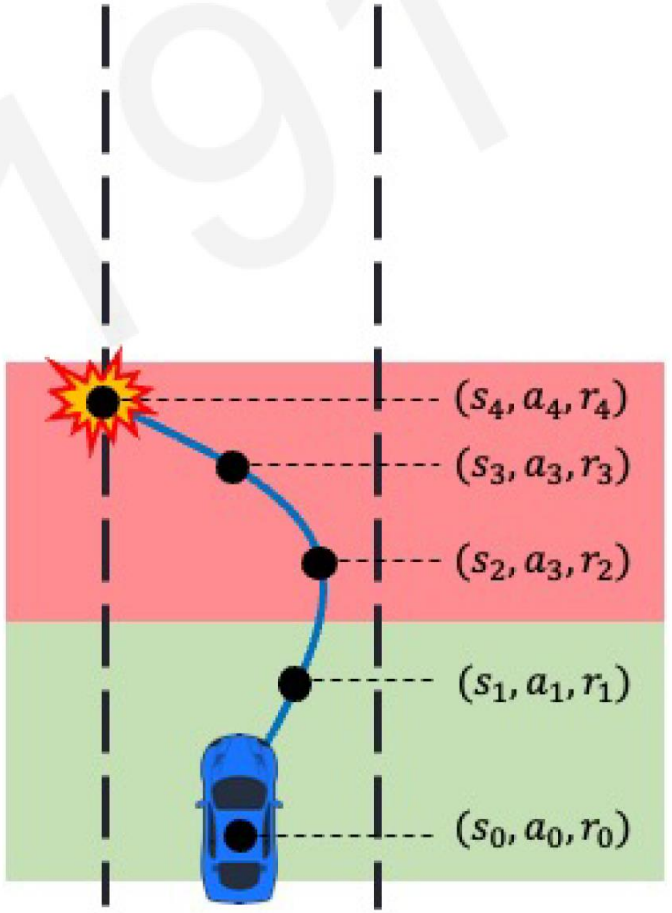
Action: steering wheel angle

Reward: distance traveled

Training Policy Gradients

Training Algorithm

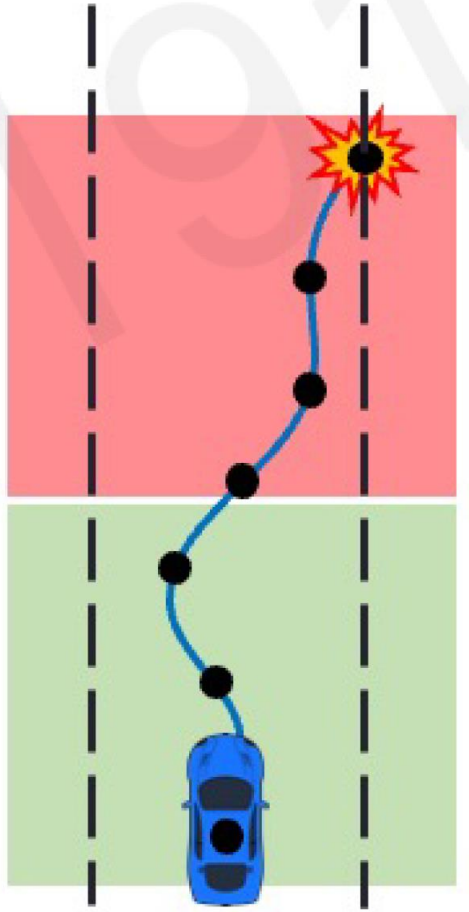
1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



Training Policy Gradients

Training Algorithm

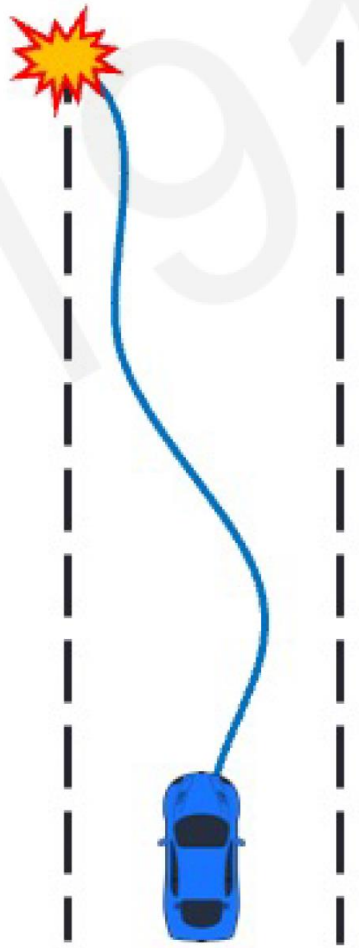
1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



Training Policy Gradients

Training Algorithm

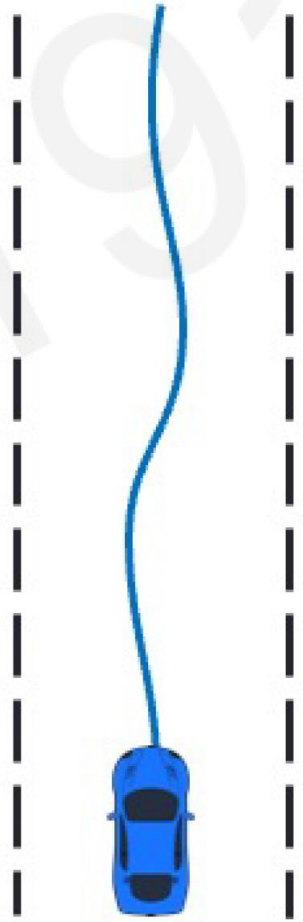
1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



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



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!

Actor-Critic Methods

- The main components in policy gradient are the policy model and the value function.
- Actor-critic methods learn the value function in addition to the policy.
  - Knowing the value function assists the policy update, such as by reducing gradient variance in vanilla policy gradients.

Actor-critic methods consist of two models, which may optionally share parameters:

- **Critic** updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(a|s)$ or state-value $V_w(s)$.
- **Actor** updates the policy parameters θ for $\pi_\theta(a|s)$, in the direction suggested by the critic.

Actor-Critic Methods

1. Initialize s, θ, w at random; sample $a \sim \pi_\theta(a|s)$.
2. For $t = 1 \dots T$:
    1. Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$;
    2. Then sample the next action $a' \sim \pi_\theta(a'|s')$;
    3. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \ln \pi_\theta(a|s)$;
    4. Compute the correction (TD error) for action-value at time t:
    $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
    and use it to update the parameters of action-value function:
    $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
    5. Update $a \leftarrow a'$ and $s \leftarrow s'$.

Two learning rates, α_θ and α_w, are predefined for policy and value function parameter updates respectively.

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 θ^Q and θ^μ.
Initialize target network Q' and μ' 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:

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

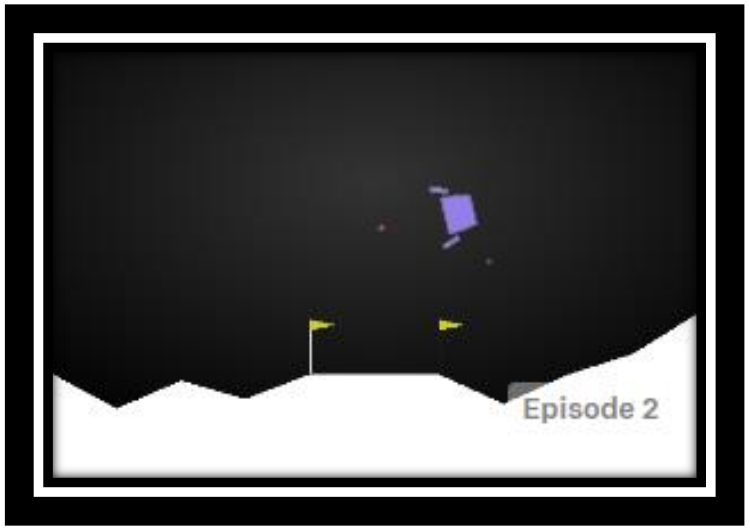
    Update the target networks:

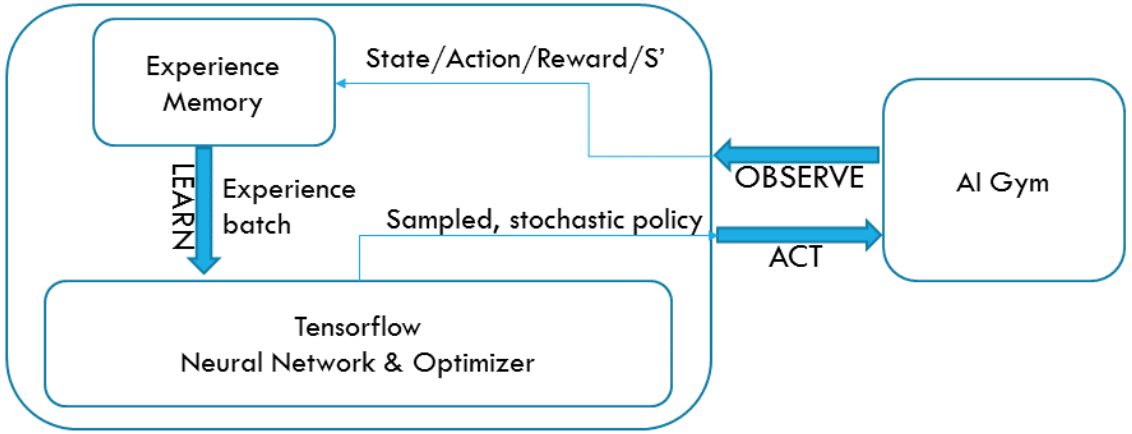
$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

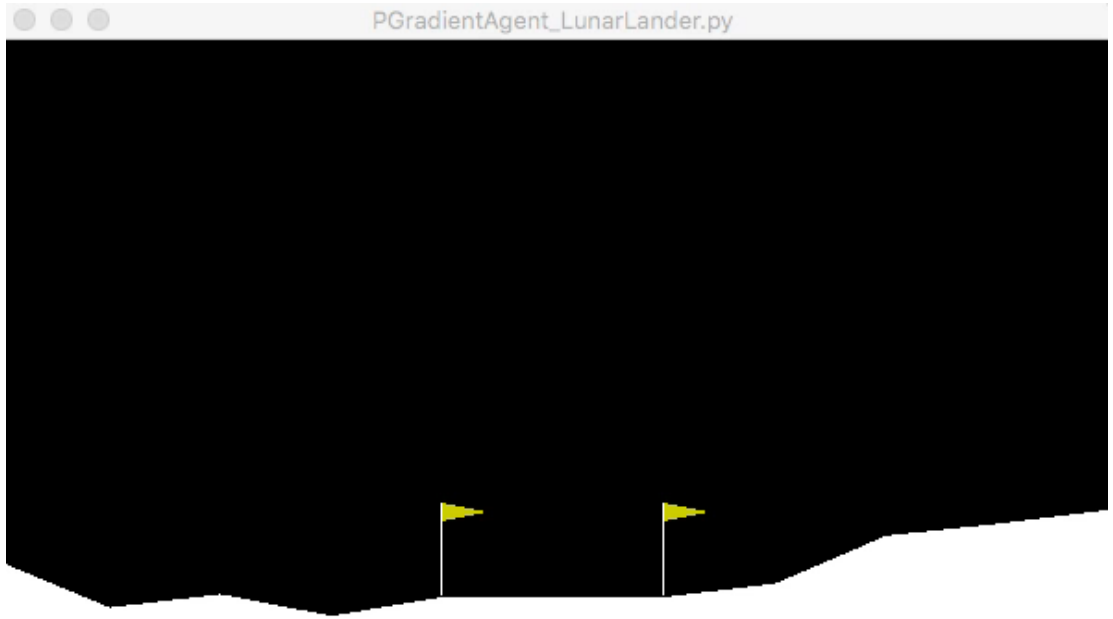
Reinforcement Learning – Neural Networks as Function Approximators

- To tackle a high-dimensional state space or continous states we can use a neural network as function approximator
- Lunar Lander experiment
  - 8 continous/discrete states
    - XY-Pos, XY-Vel, Rot, Rot-rate, Leg1/Leg2 ground contact
  - 4 discrete actions
    - Left thrust
    - Right thrust
    - Main engine thrust
    - NOP
  - Rewards
    - Move from top to bottom of the screen (+ ~100-140)
    - Land between the posts (+100)
    - Put legs on ground (+10 per leg)
  - Penalties
    - Using main engine thrust (-0.3 per frame)
    - Crashing (-100)
- Solved using Stochastic Policy Gradients

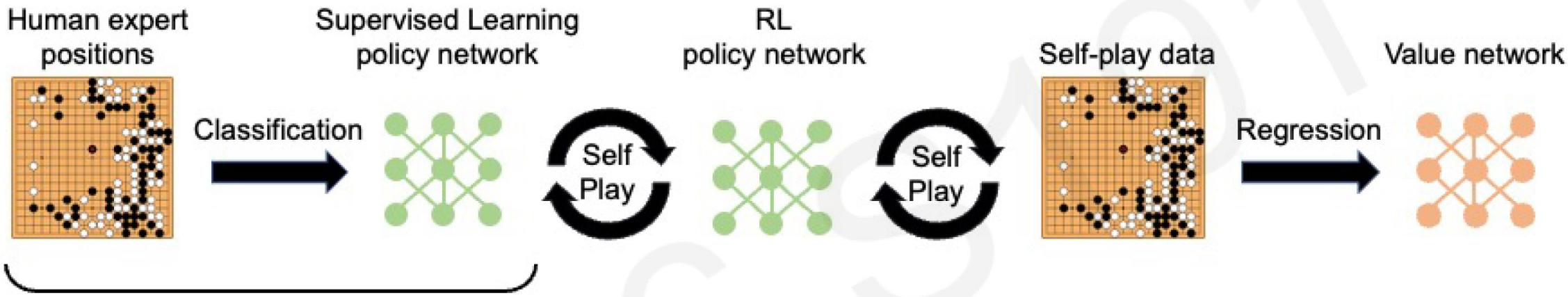




Reinforcement Learning Neural Networks as Function Approximators



AlphaGo Beats Top Human Player (2016)

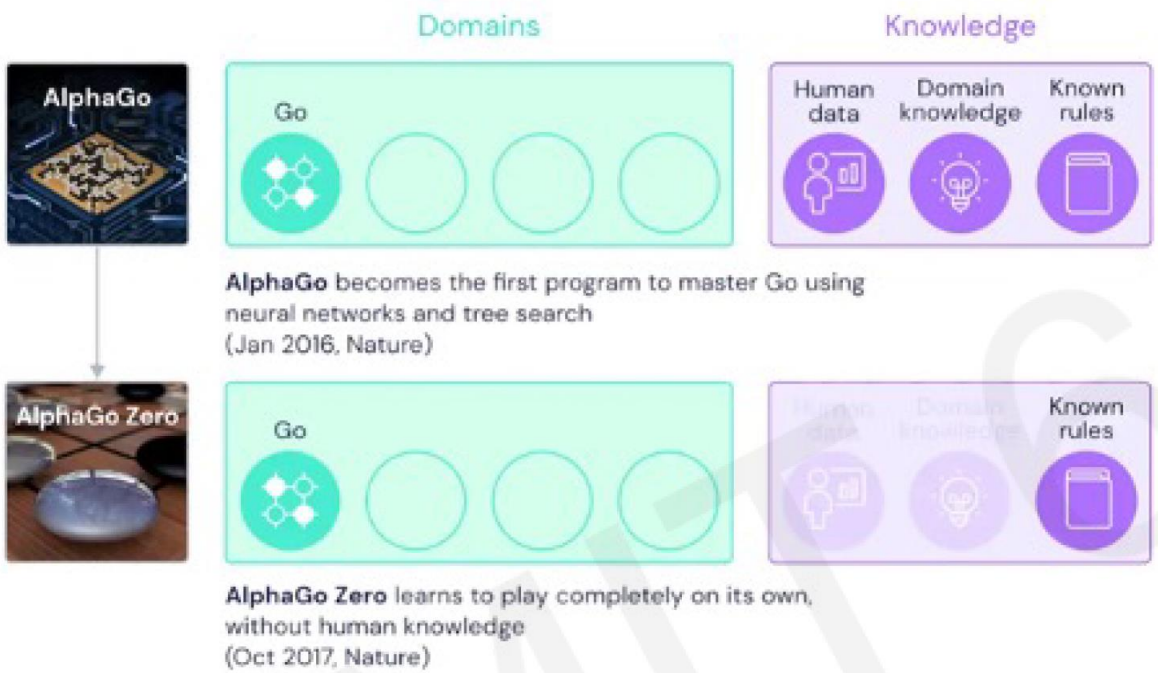


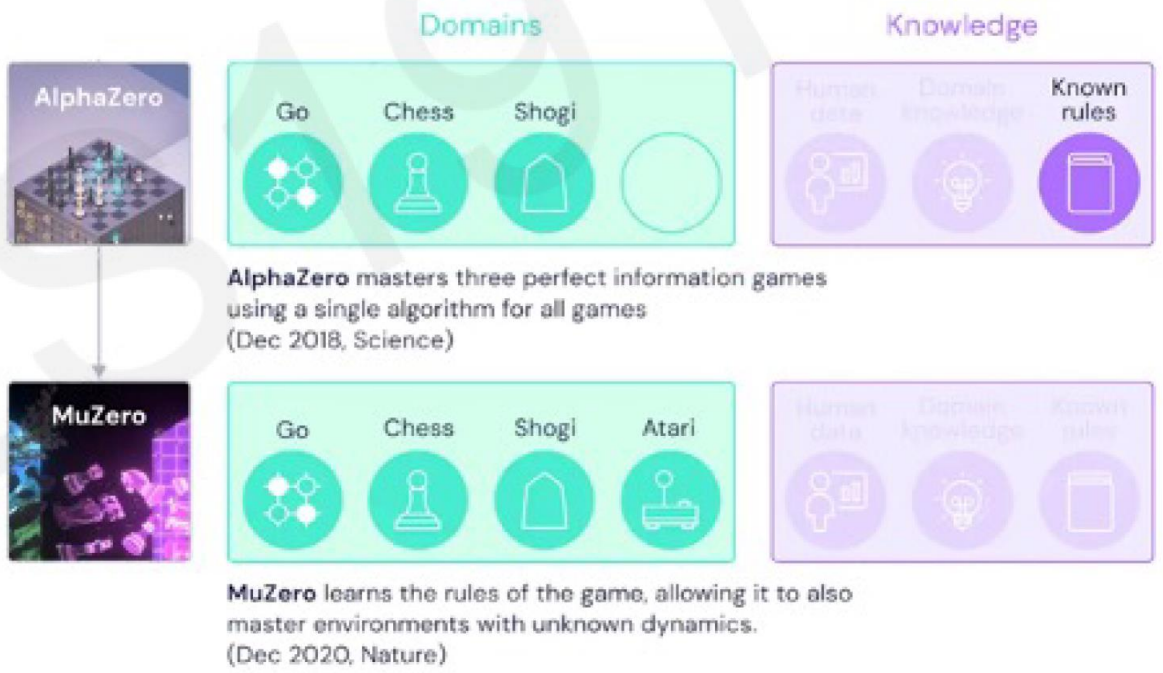
1) Initial training: human data

2) Self-play and reinforcement learning
→ super-human performance

3) "Intuition" about board state

MuZero: Learning Dynamics for Planning (2020)





Deep Reinforcement Learning Summary

Foundations

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

Q-Learning

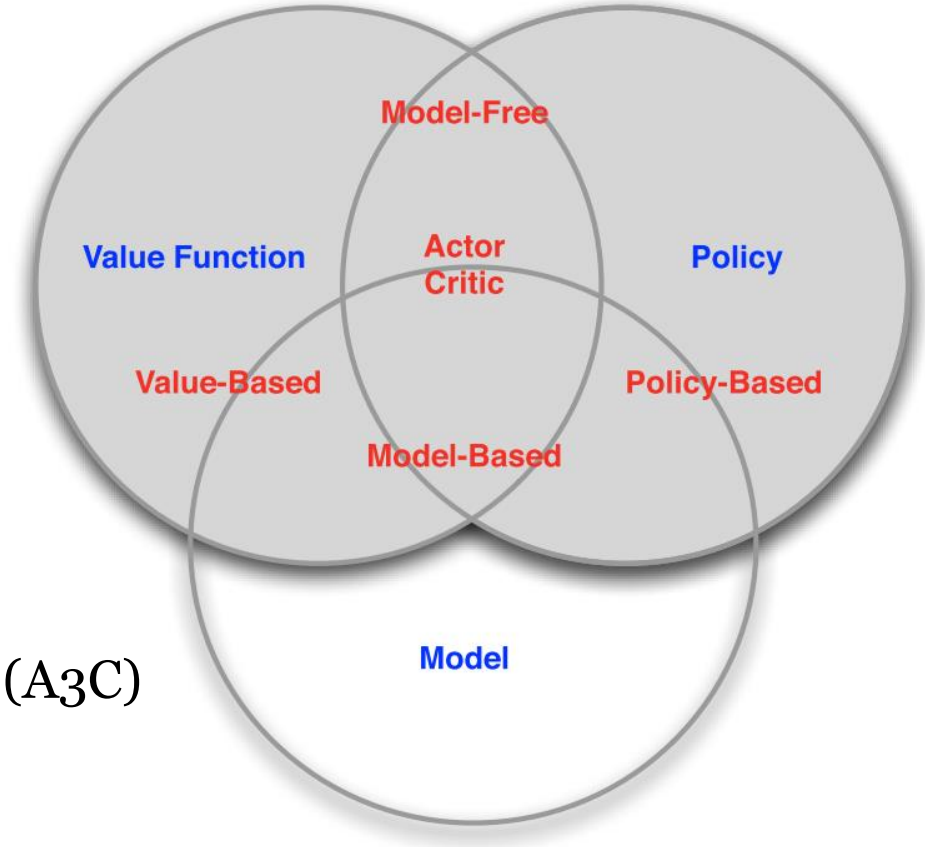
- Q function: expected total reward given **s, a**
- Policy determined by selecting action that maximizes Q function

Policy Gradients

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

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

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

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)

Model-based and Model-free Learning

- Model-based algorithms
  - Try to model the behavior of other agents
  - More specific models may improve performance, but at a loss of generality
- Model-free algorithms
  - Do not use models
  - May perform worse or take longer to converge

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)

MDP, POMDP, and Dec-POMDP

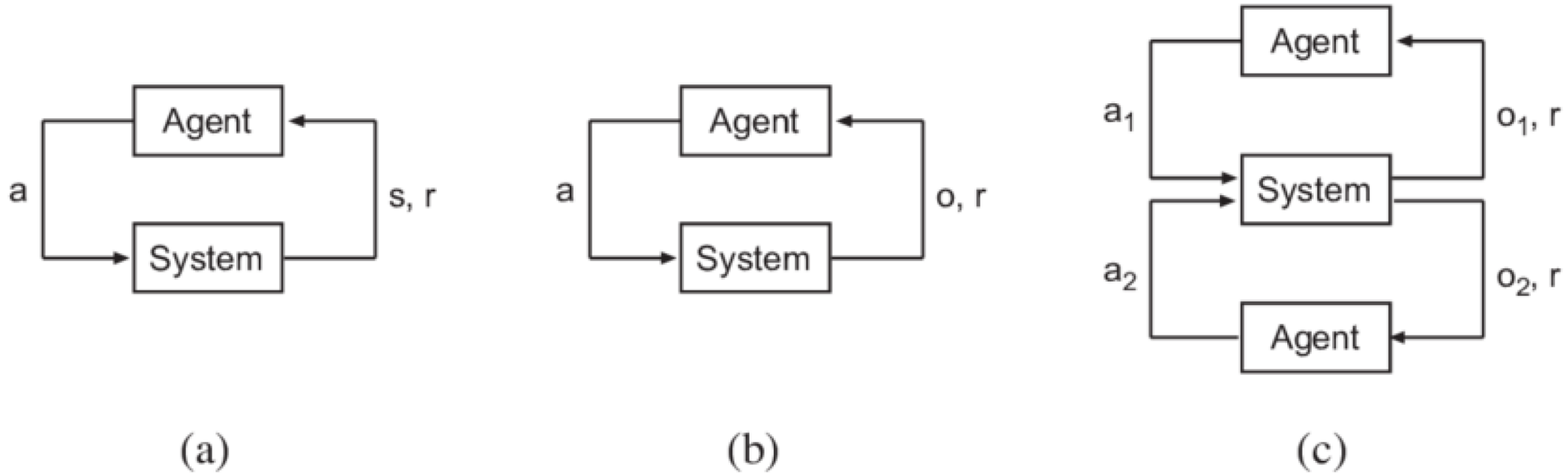
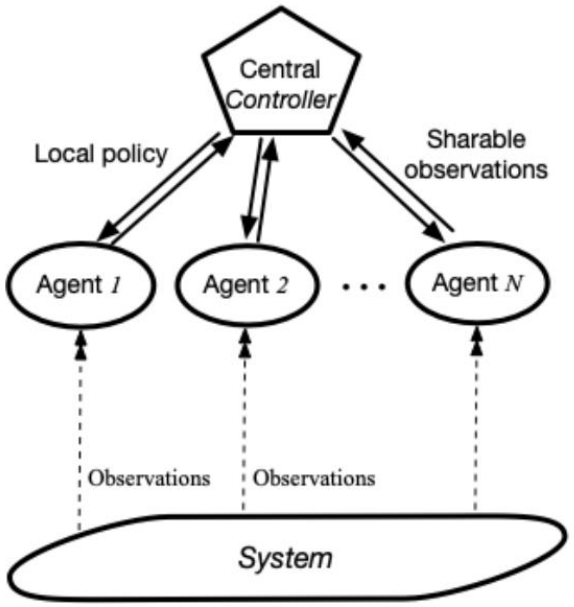
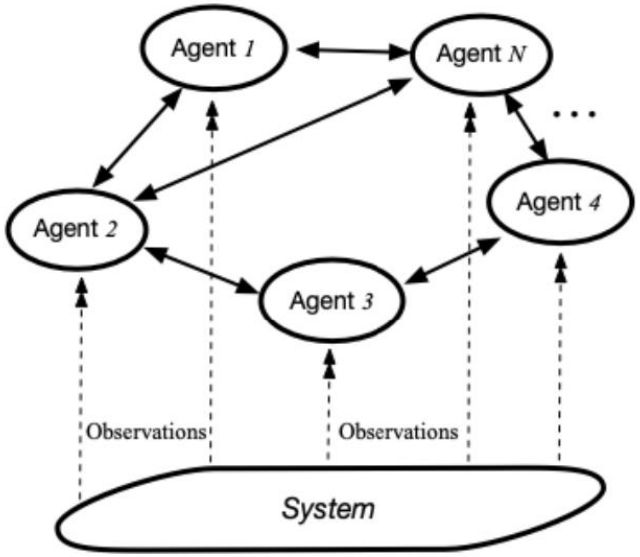


Figure: (a) Markov decision process (MDP) (b) Partially observable Markov decision process (POMDP) (c) Decentralized partially observable Markov decision process with two agents (Dec-POMDP)

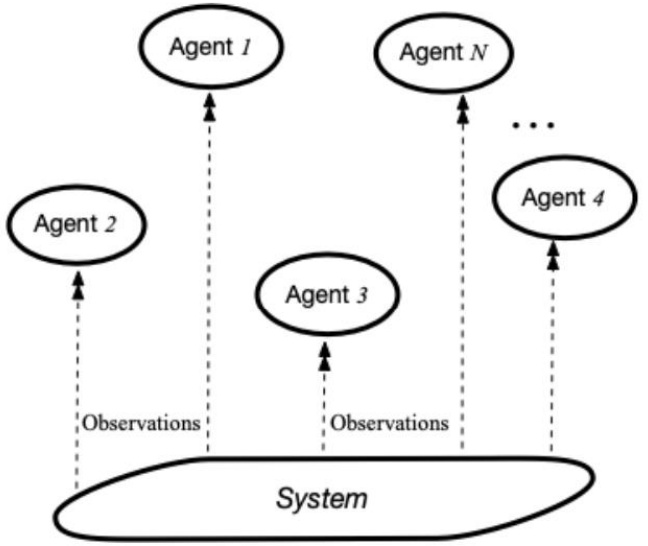
Multi-Agent Reinforcement Learning



(a) Centralized setting

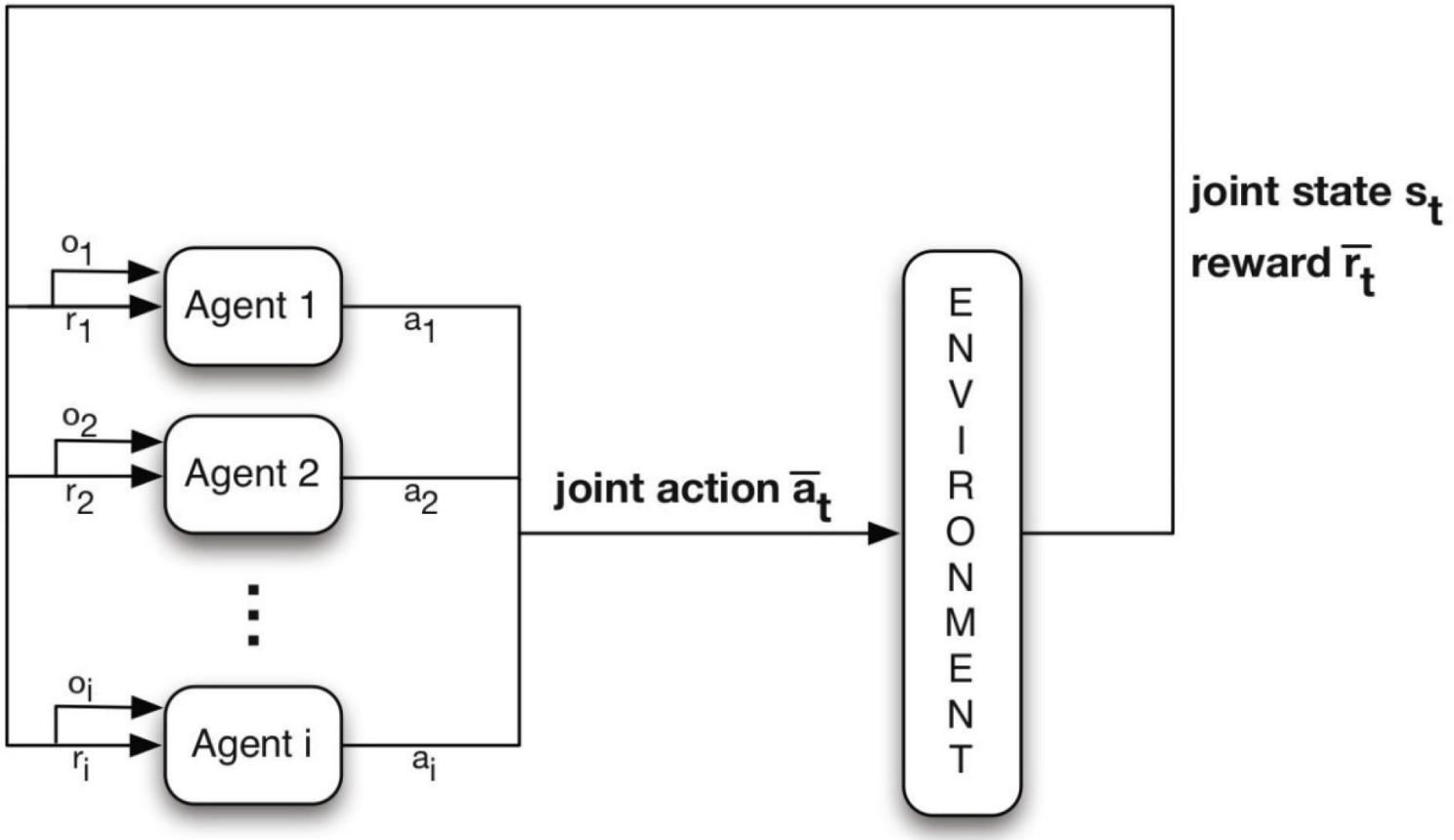


(b) Decentralized setting with networked agents



(c) Fully decentralized setting

Multi-Agent Reinforcement Learning



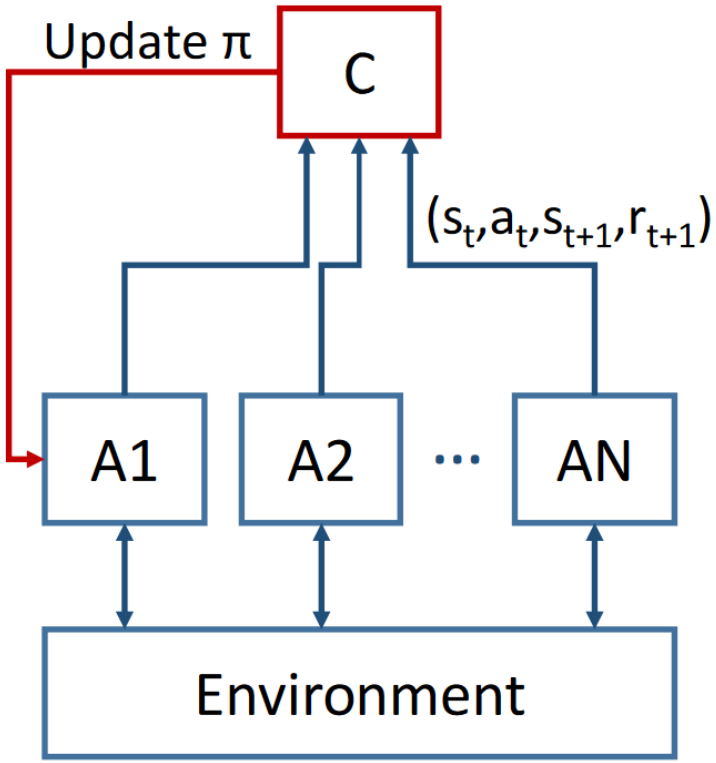
Source: Nowe, Vrancx & De Hauwere 2012

Decentralized Multi-Agent Deep Reinforcement Learning

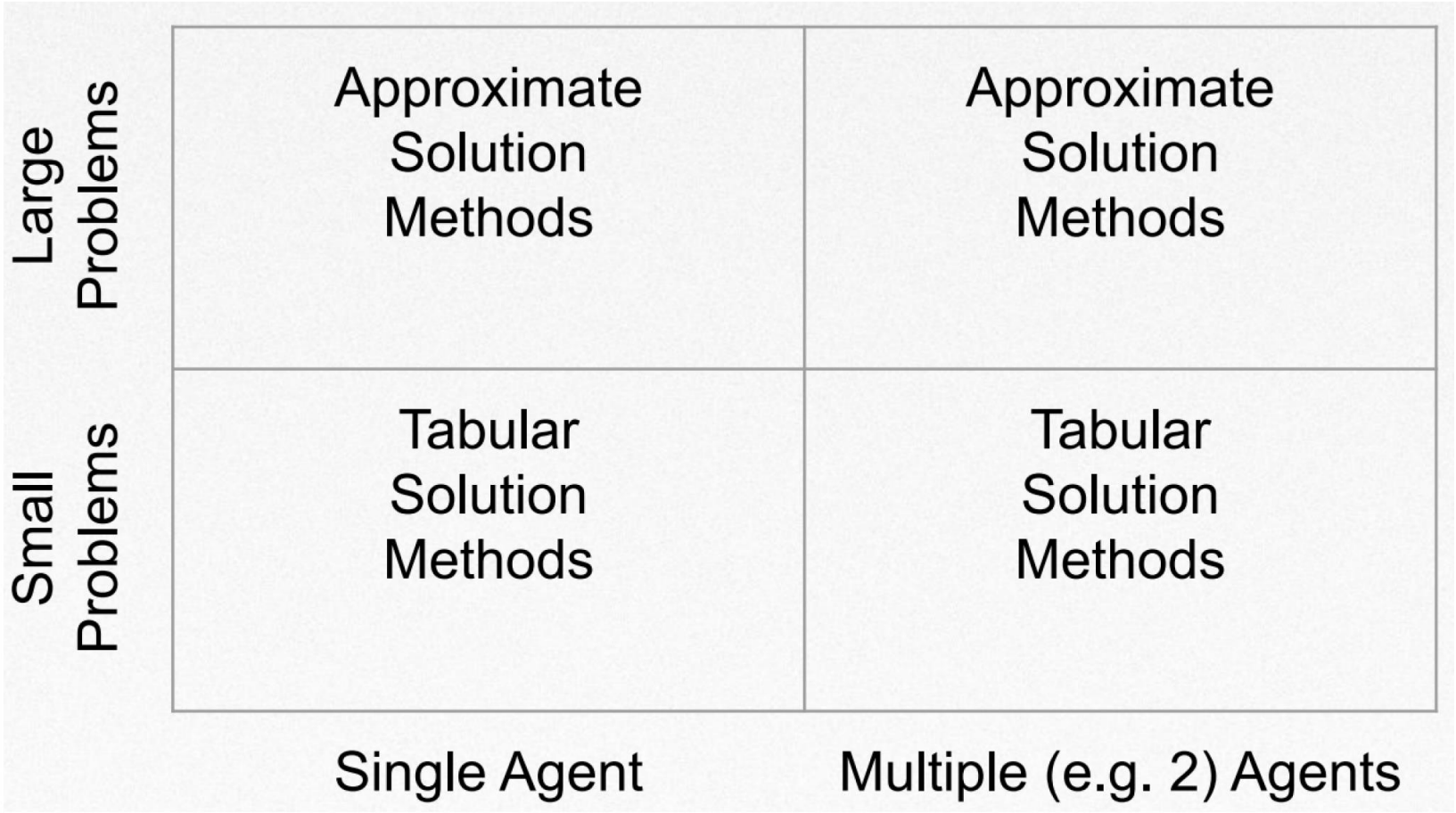
- Though no theoretical guarantees exist, single-agent algorithms may produce interesting results in multi-agent systems
- Ways to stabilize the learning process
  - Clever design of reward systems
  - Training populations of agents
    - Can allow agents to generalize

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



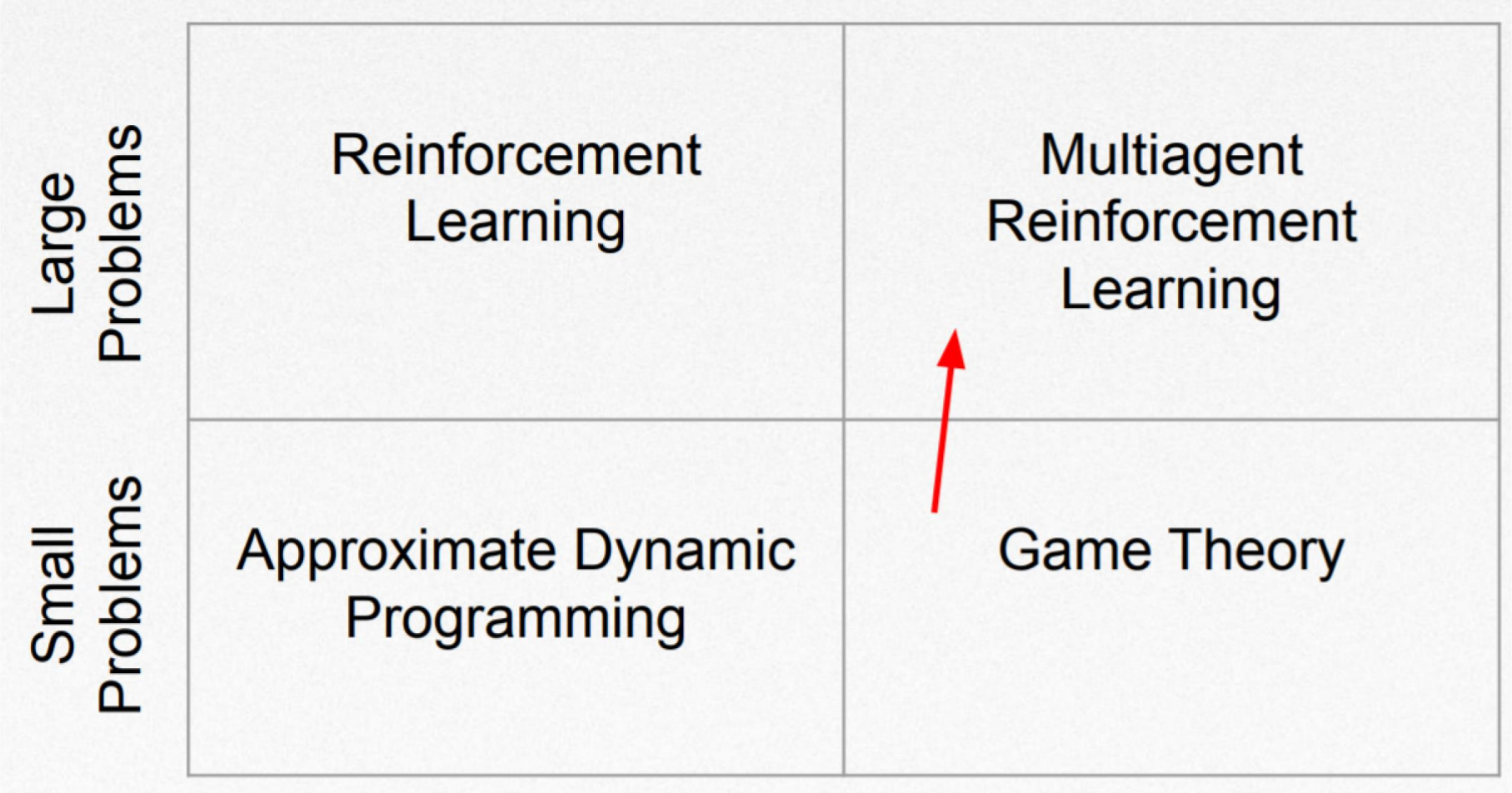
Research in Multi-Agent RL



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

Foundations of MARL



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 $\boldsymbol{\pi}$:

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

Nash Q-Learning

- In MARL, the objective of each agent is to learn an optimal policy to maximize its value function
- Optimizing the v_π^j for agent j depends on the joint policy $\boldsymbol{\pi}$ of all agents
- A Nash equilibrium is a joint policy π 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

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

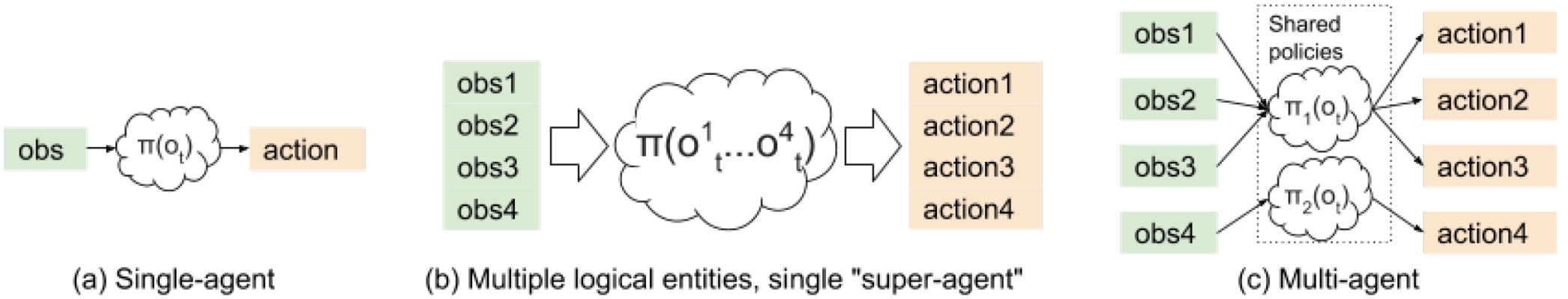
Nash Q-Learning (cont)

- In a Nash equilibrium, each agent acts with the best response π_*^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 $\boldsymbol{r}(s,\boldsymbol{a}) = [r^1(s,\boldsymbol{a}), \cdots, r^N(s,\boldsymbol{a})]$

MARL Policies

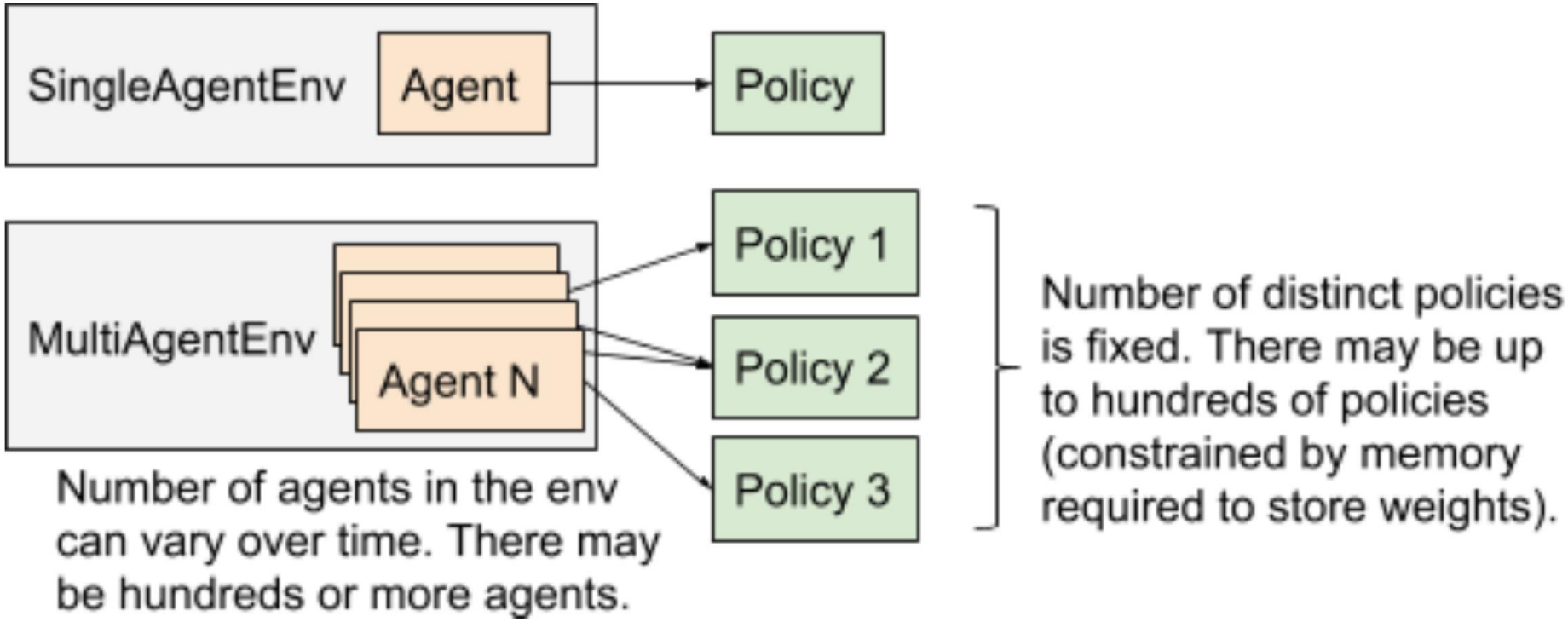


(a) Single-agent

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

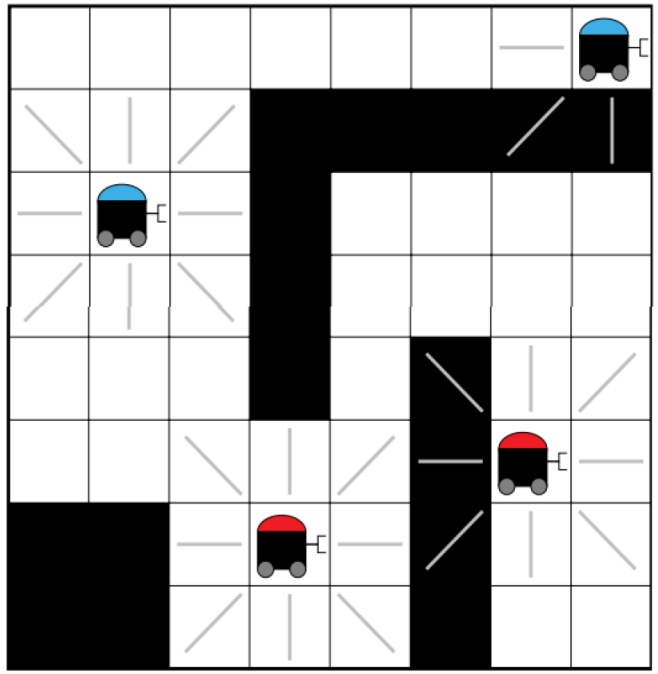
(c) Multi-agent

MARL Policies



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



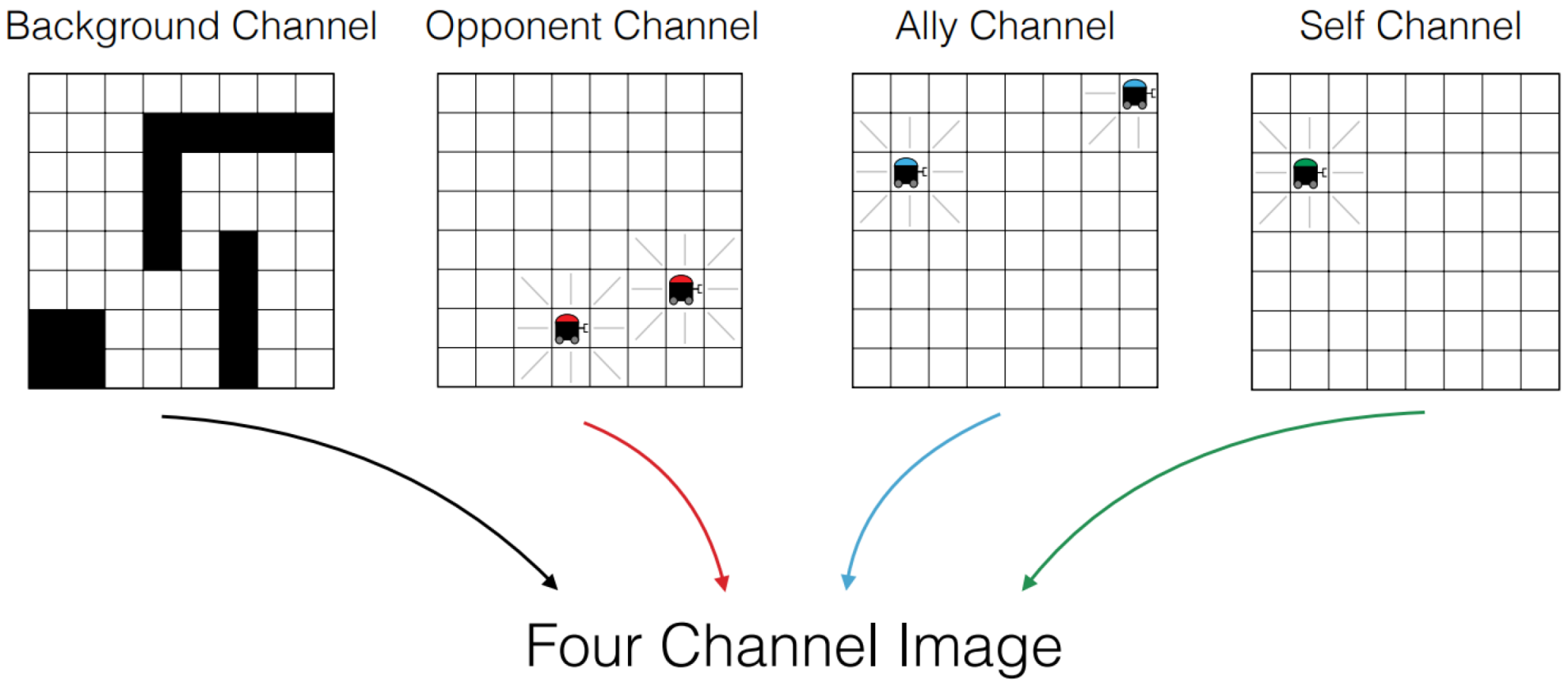
Multi-Agent Deep Q-Network (MADQN)

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):

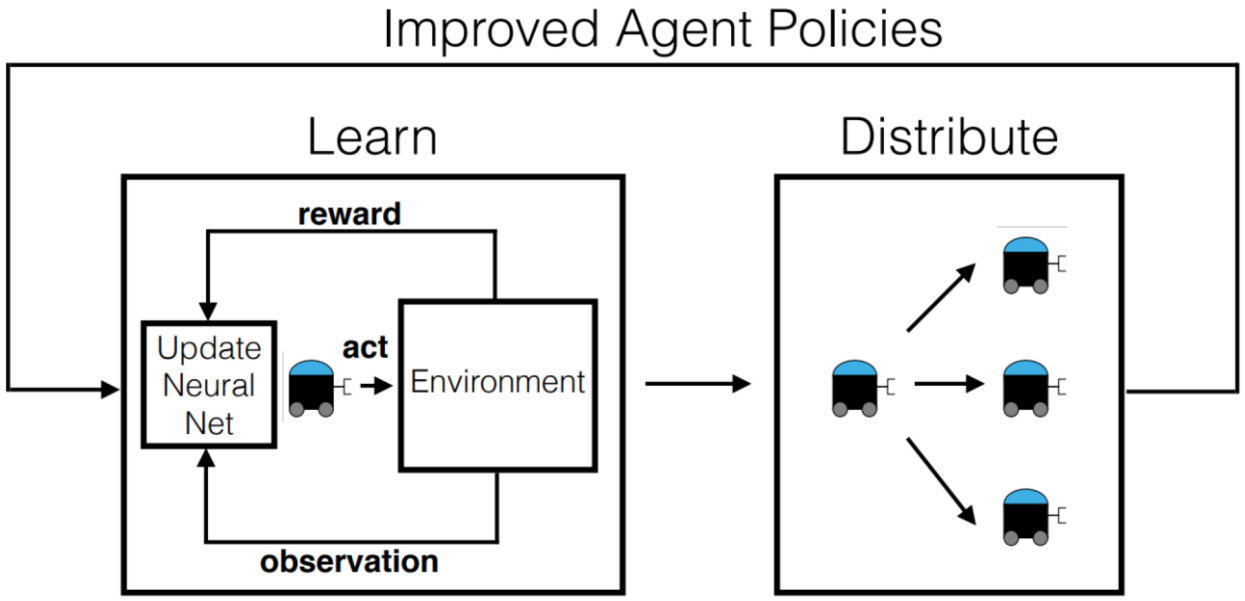
- The image tensor is of size 4× W × H, where W and H are the height and width of our two dimensional domain and four is the number of channels in the image.
- Channels:
  - **Background Channel**: contains information about any obstacles in the environment
  - **Opponent Channel:** contains information 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

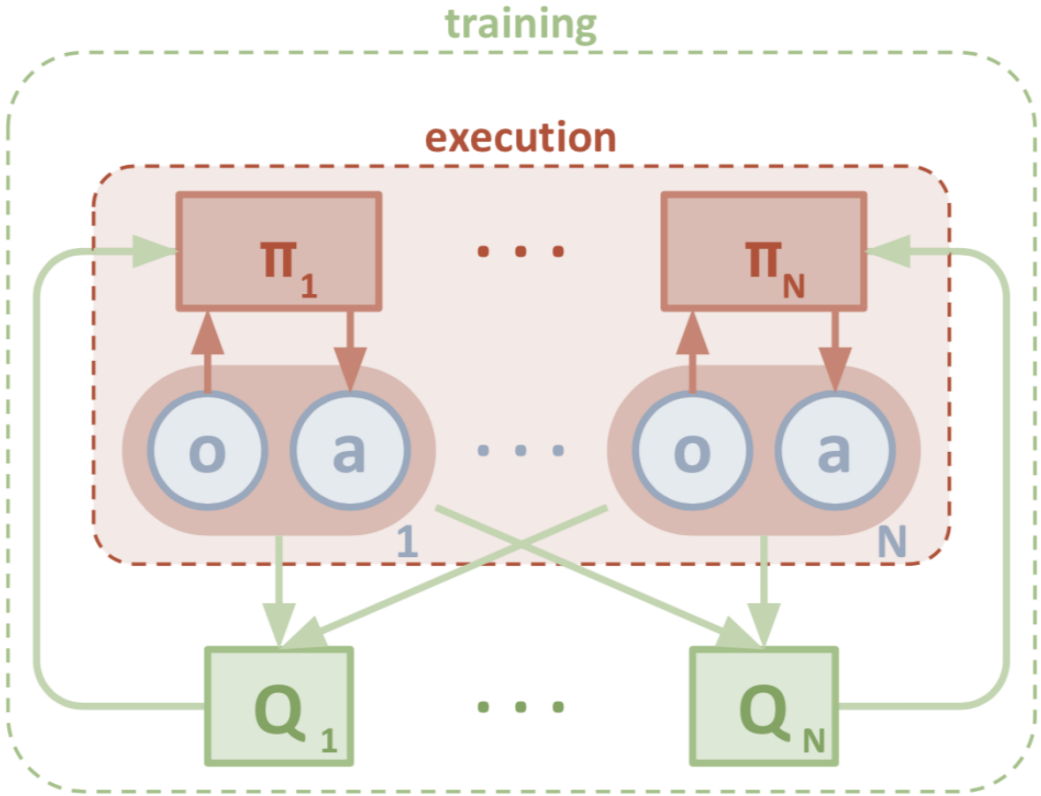


MADQN: Dealing with Ambiguity

- **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.

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:

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

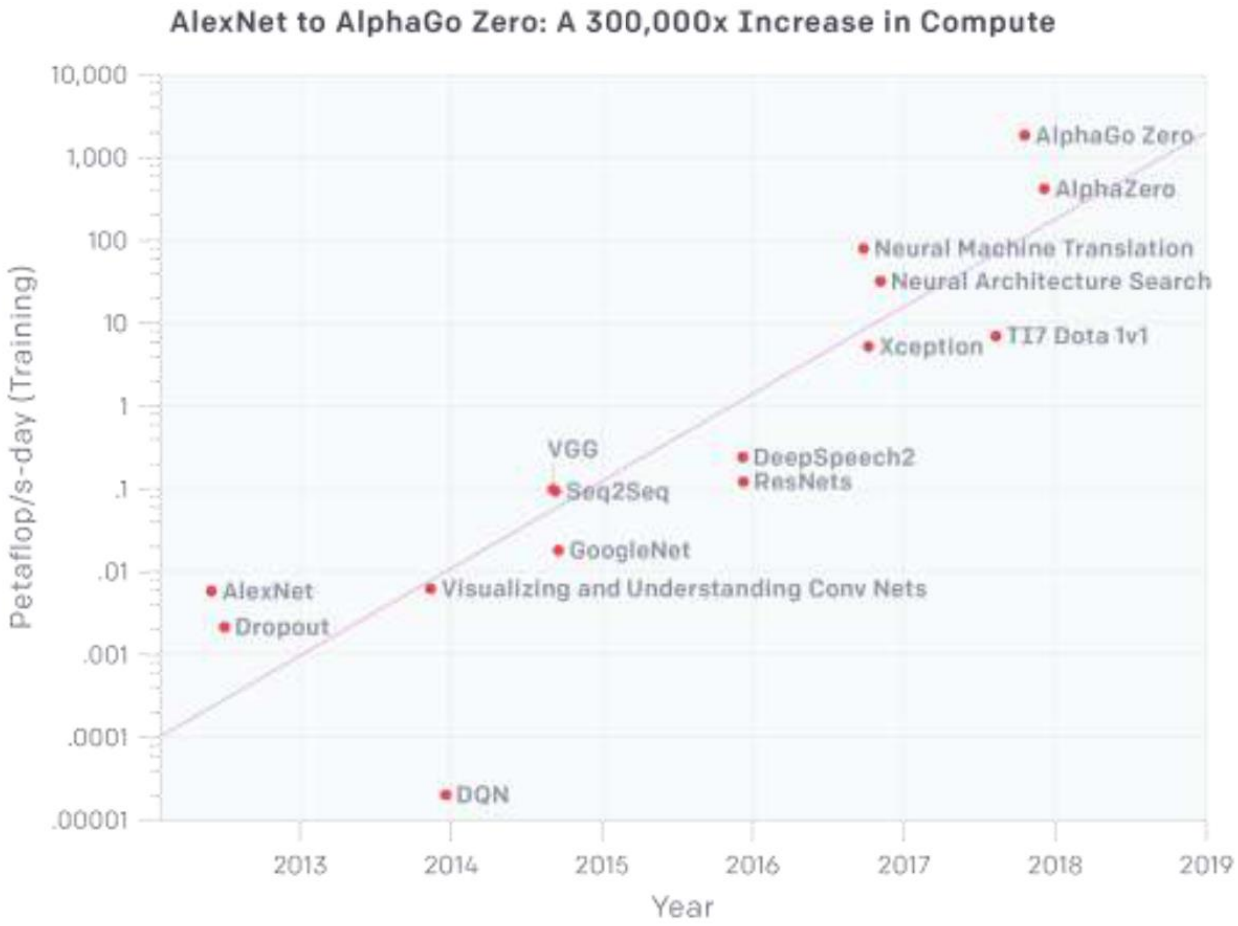
    end for
    Update target network parameters for each agent i:

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

  end for
end for

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 HT2023:
Reinforcement learning
Deep reinforcement learning
Multi-agent learning

www.ida.liu.se/~TDDE13

LINKÖPING UNIVERSITY