TDDC17 LE11 HT2024

Machine Learning III

Fredrik Heintz

Dept. of Computer Science
Linköping University

fredrik.heintz@liu.se

@FredrikHeintz

Outline:

- Reinforcement learning
- Deep reinforcement learning
- Multi-objective reinforcement learning

LINKÖPING UNIVERSITY

Classes of Learning Problems

Supervised Learning

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

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

Apple example:

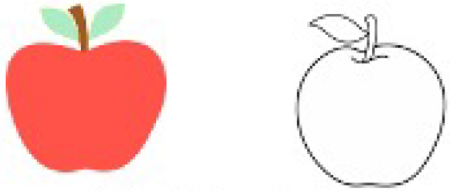
This thing is an apple.

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.

Reinforcement Learning: Key Concepts

Agent: takes actions.

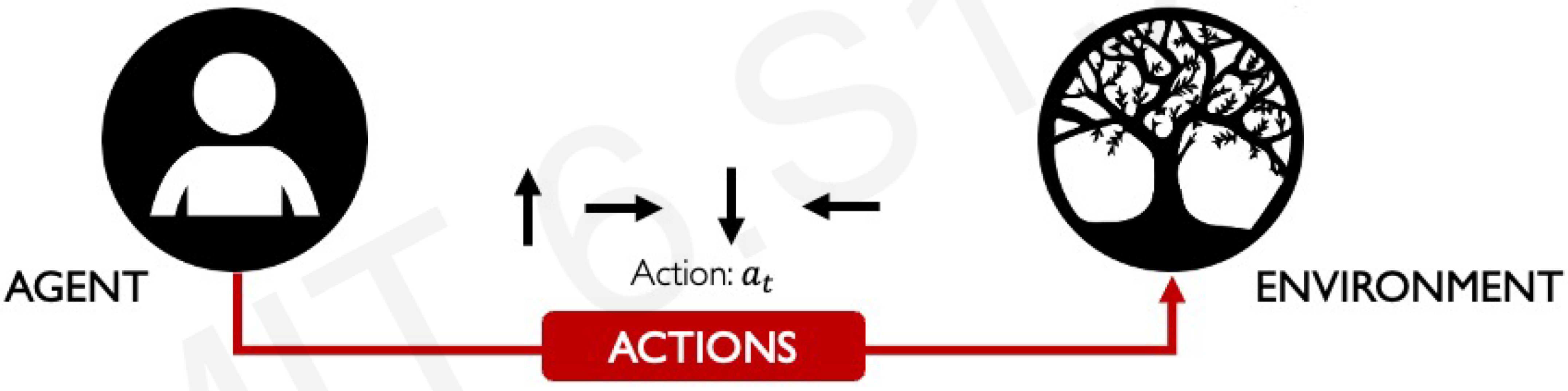
Reinforcement Learning: Key Concepts

AGENT

ENVIRONMENT

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

Reinforcement Learning: Key Concepts



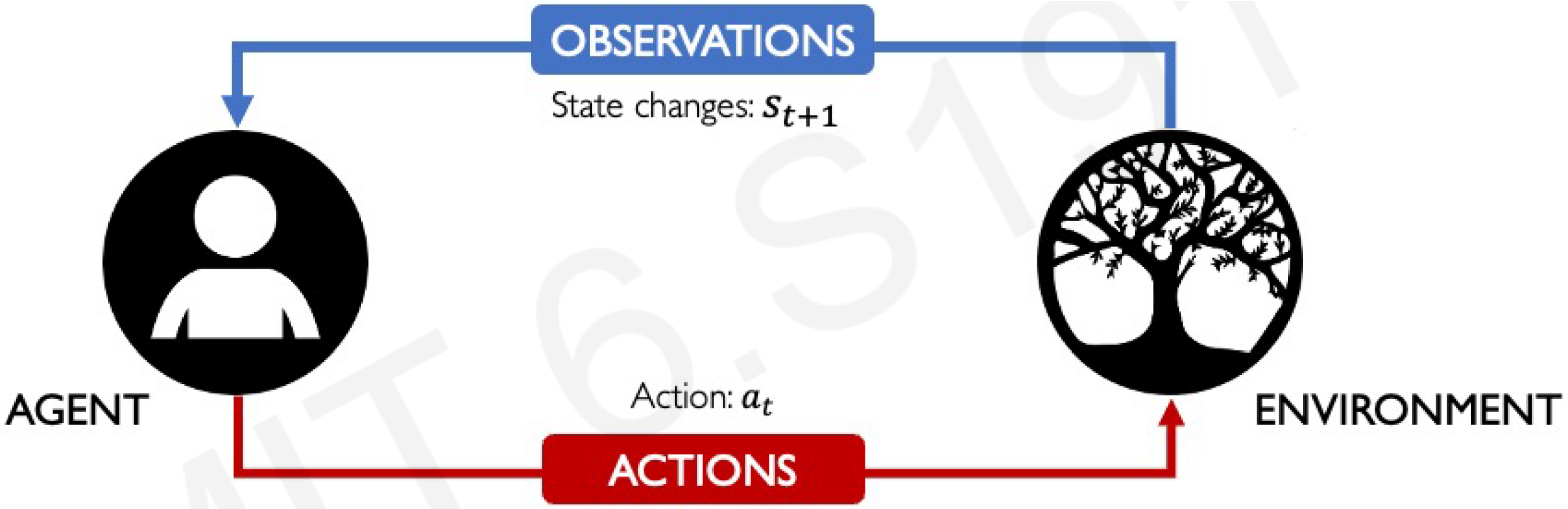
Action: a move the agent can make in the environment.

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

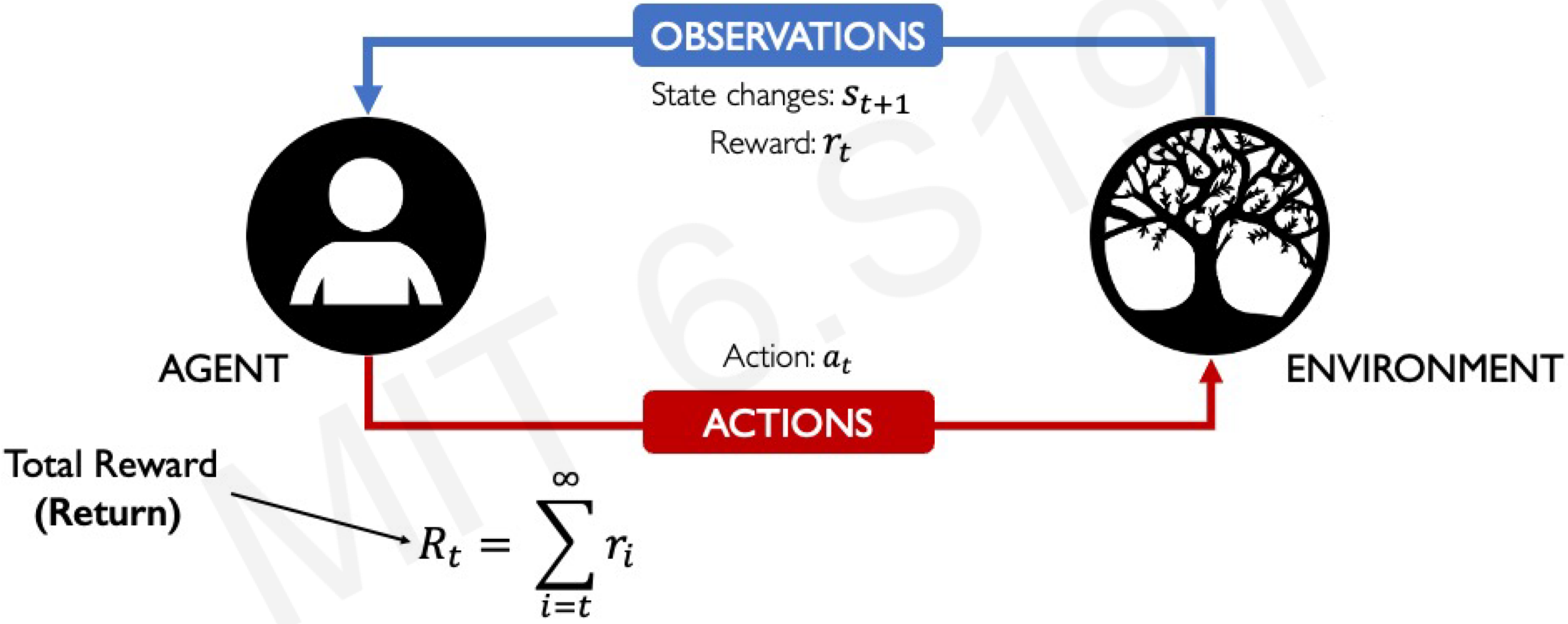


State: a situation which the agent perceives.

Reinforcement Learning: Key Concepts

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

Reinforcement Learning: Key Concepts



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

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

Reinforcement Learning: Key Concepts

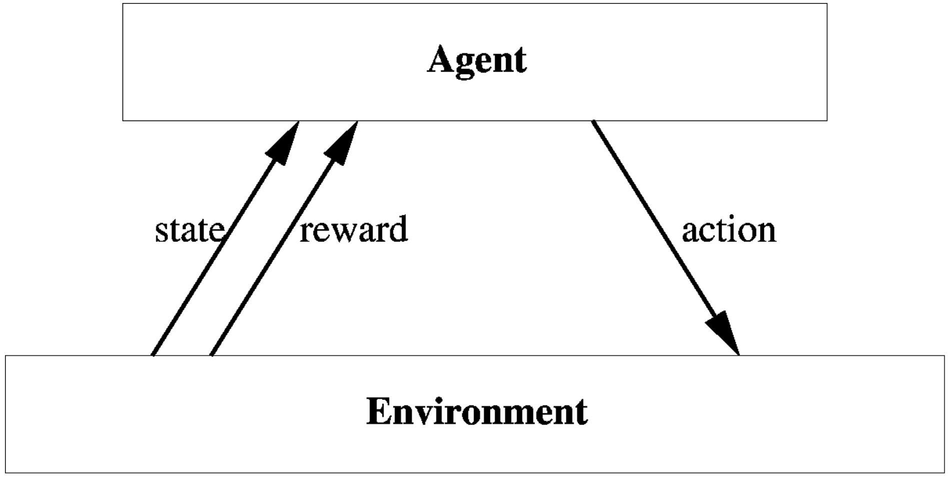
Discounted Total Reward (Return)

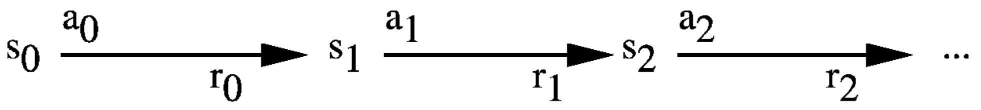
$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i = \gamma^t r_t + \gamma^{t+1} r_{t+1} \ldots + \gamma^{t+n} r_{t+n} + \cdots$$

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 $\pi: S \to A$
  - 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

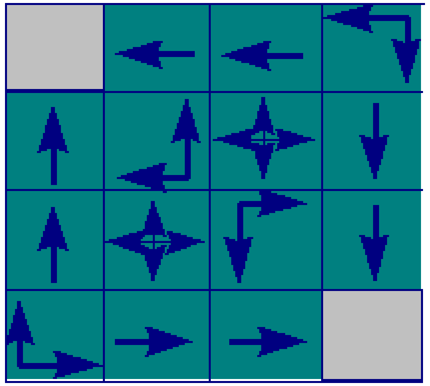
$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$, where $0<\gamma<1$

RL Value Function - Example

A minimum time to goal world

0	-14	-20	-22
-14	-18	-22	-20
-20	-22	-18	-14
-22	-20	-14	0

Value function for random movement



Optimal policy

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

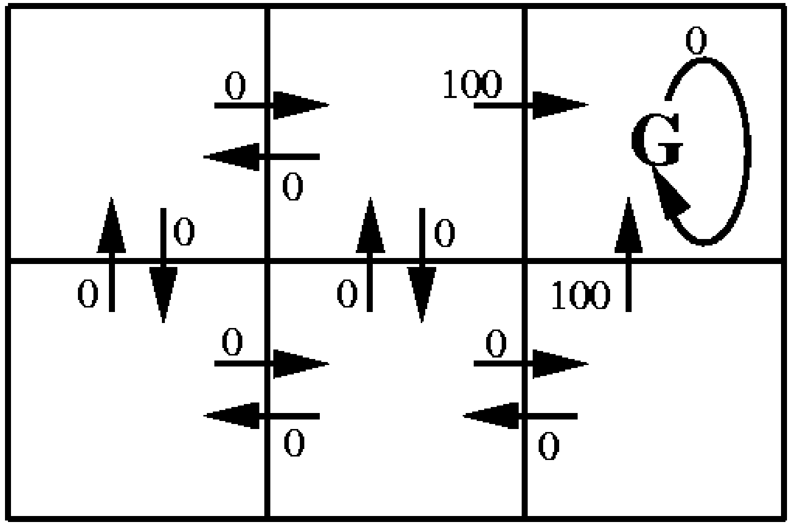
Optimal value function

Markov Decision Processes

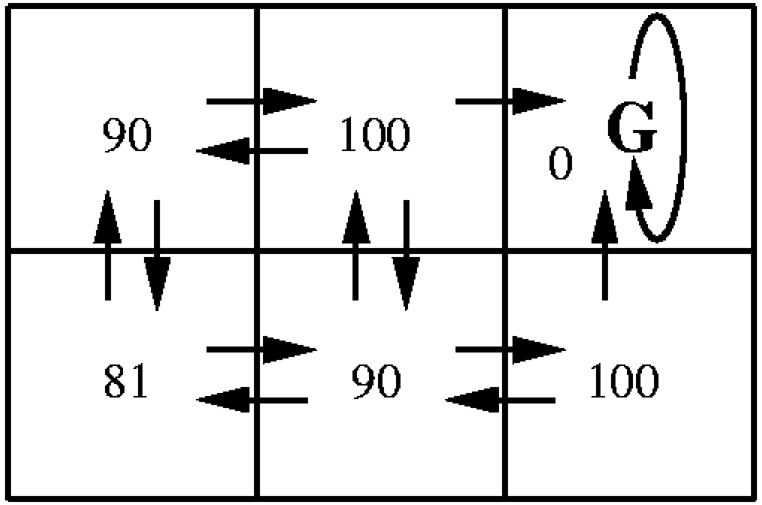
Assume:

- 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

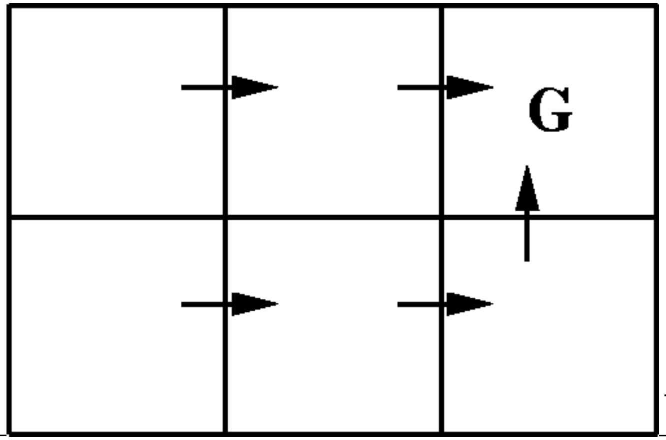
MDP Example



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

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

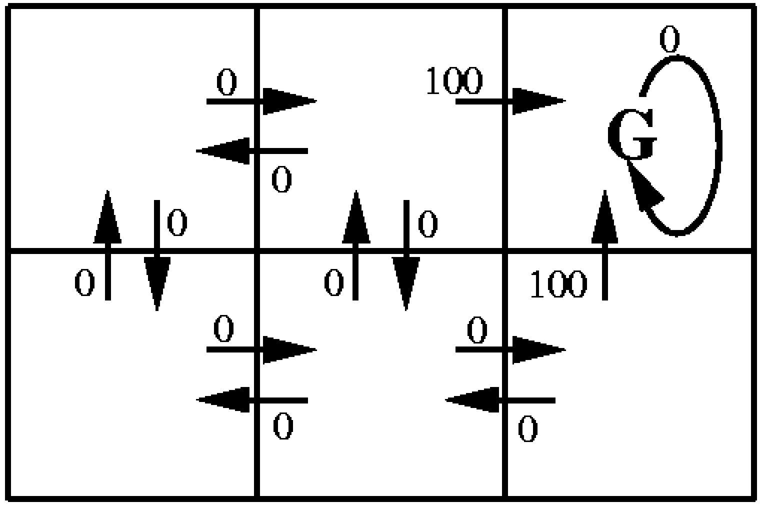
The Q-Function

Optimal policy:

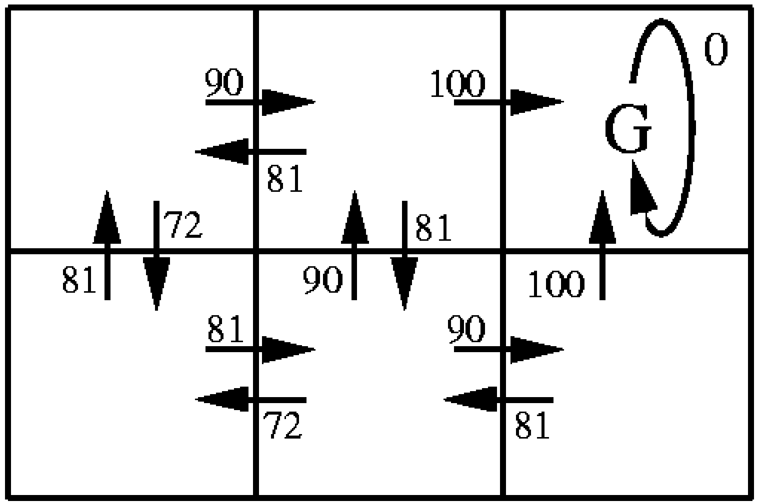
- $\pi^*(s) = \text{argmax}_a[r(s,a) + \gamma V^*(\delta(s,a))]$
- Doesn't work if we don't know r and δ.

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:

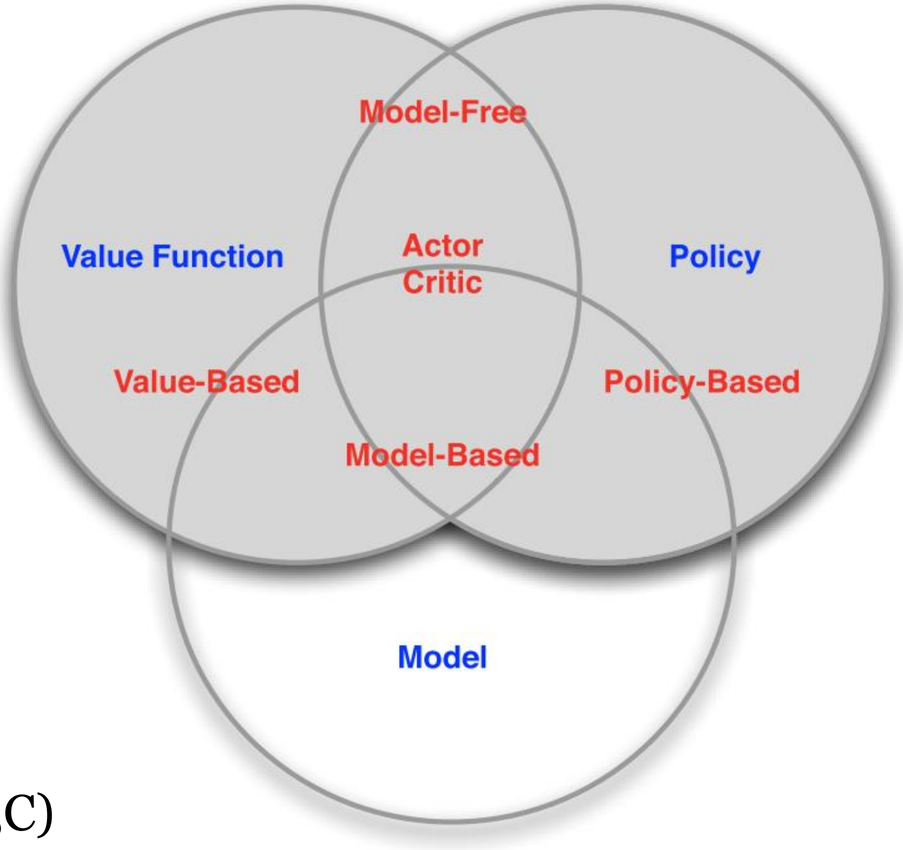
  $Q(s_t,a_t) := r(s_t,a_t) + \gamma V^*(\delta(s_t,a_t)) =$

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

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

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 $Q(s, a)$

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

Policy Learning

Find $\pi(s)$

Sample $a \sim \pi(s)$

Q-Learning for Deterministic Worlds

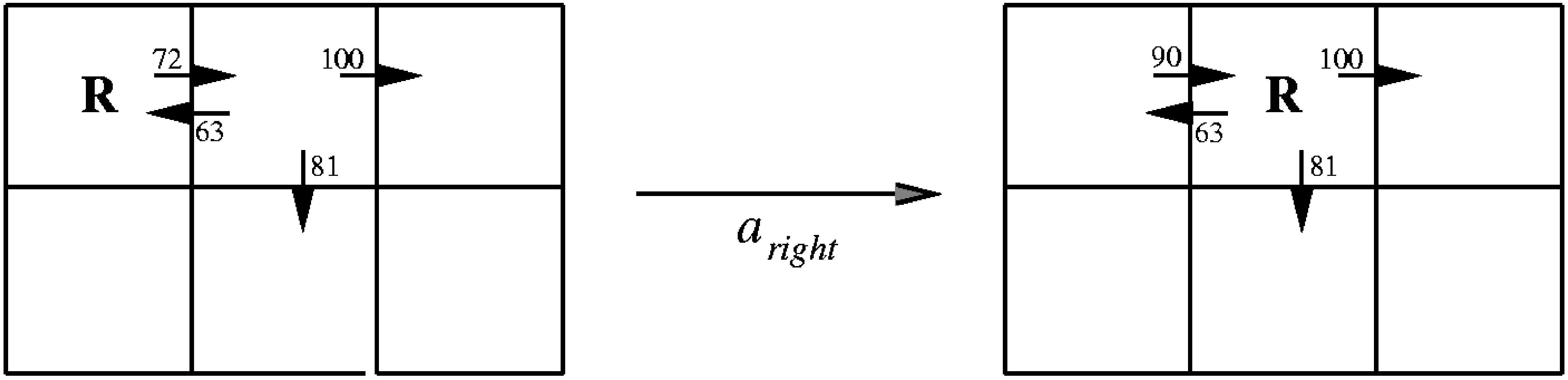
For each s, a initialize table entry $\hat{Q}(s,a) := 0$.

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



Initial state: s_1 Next state: s_2

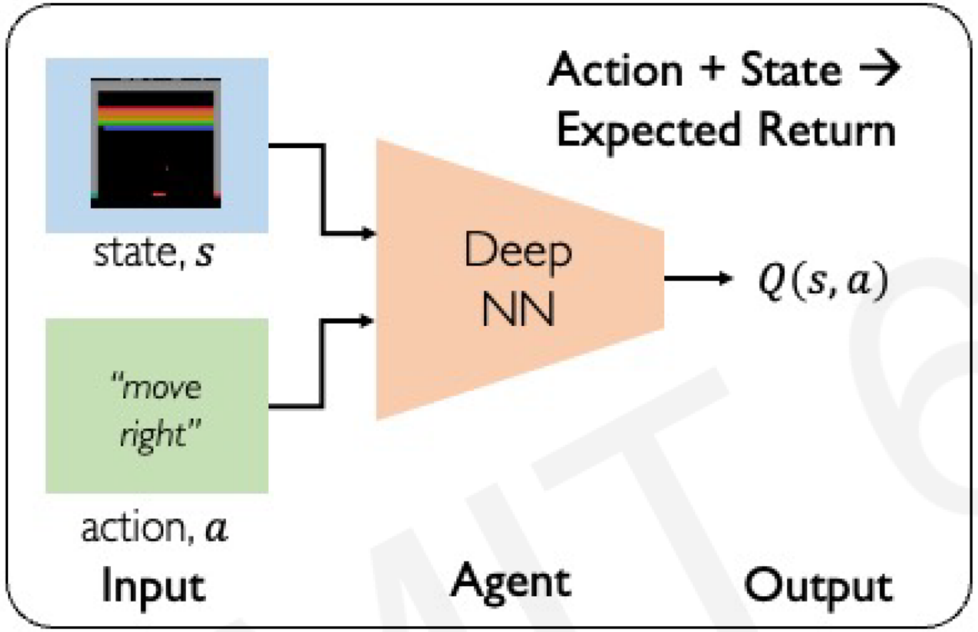
$$\hat{Q}(s_1, a_{right}) := r + \gamma \max_{a'} \hat{Q}(s_2, a')$$
$$:= 0 + 0.9 \max\{63, 81, 100\}$$
$$:= 90$$

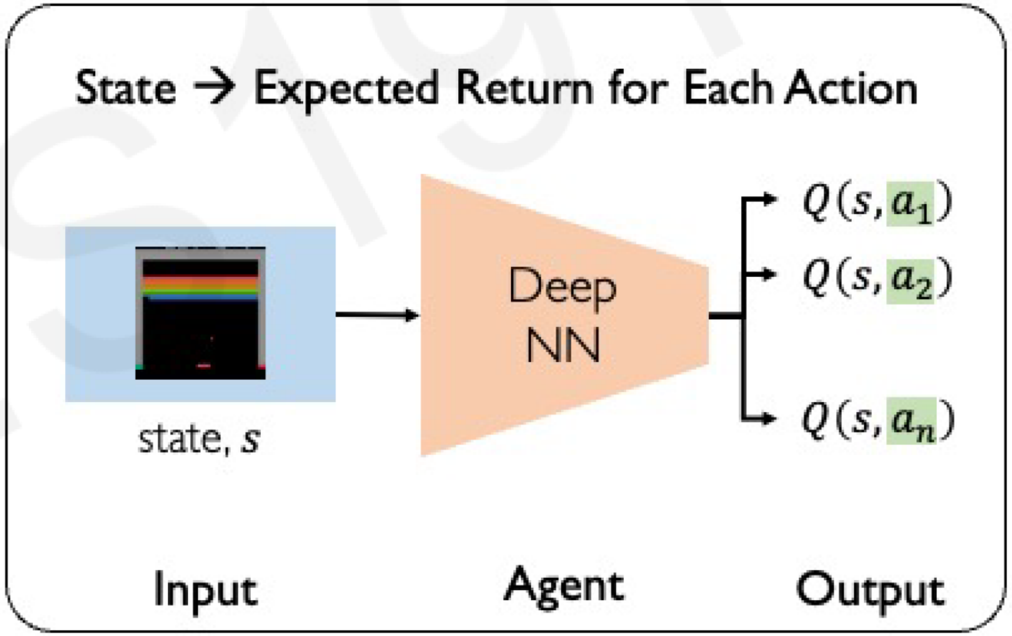
Q-Learning Continued

- 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)

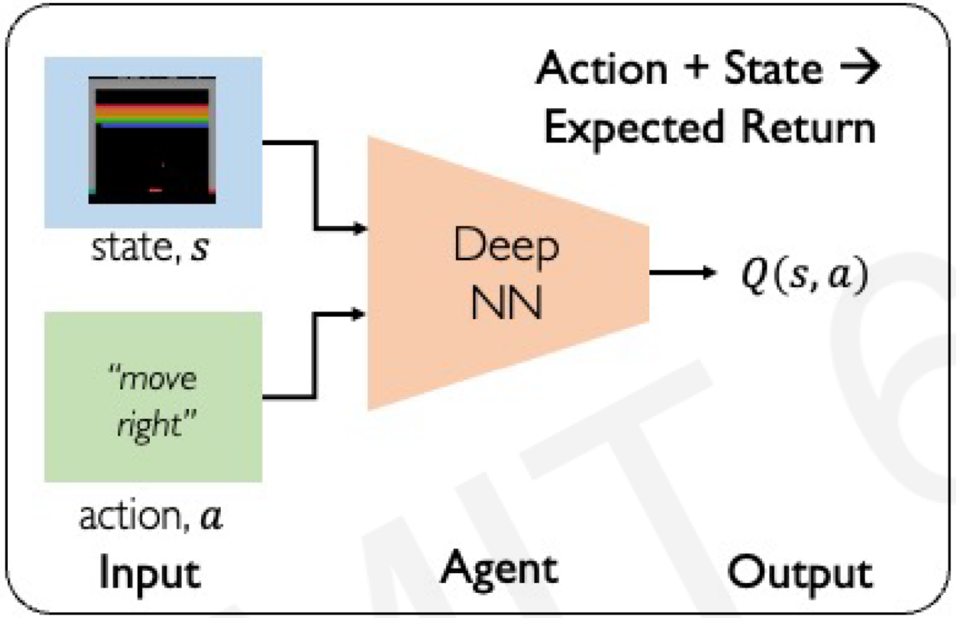
How can we use deep neural networks to model Q-functions?

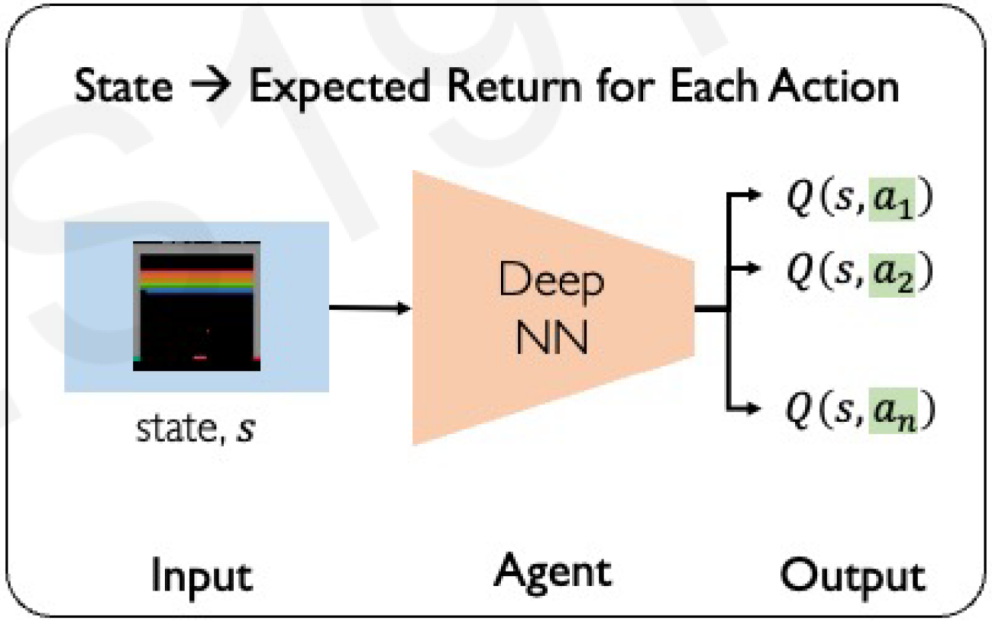




Deep Q Networks (DQN): Training

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

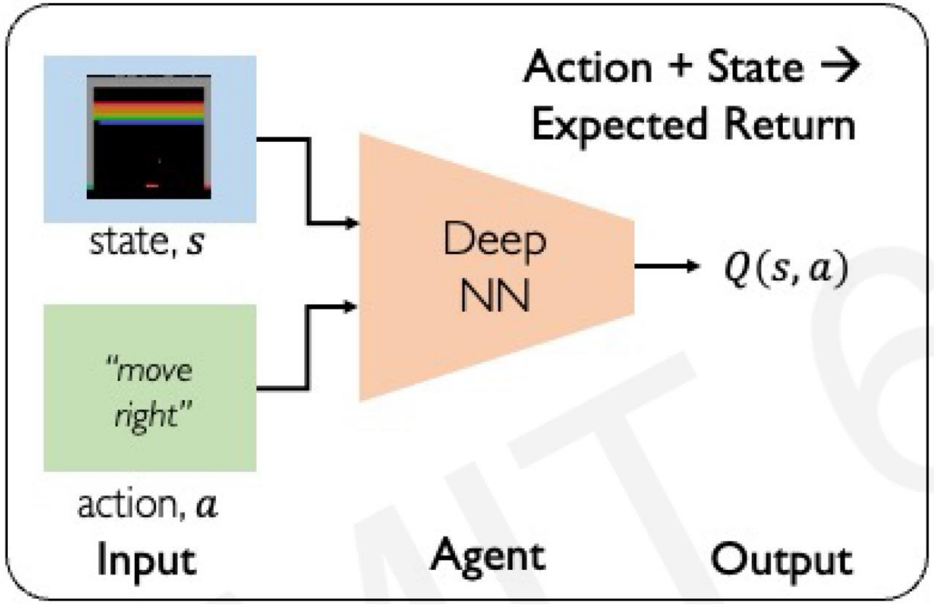


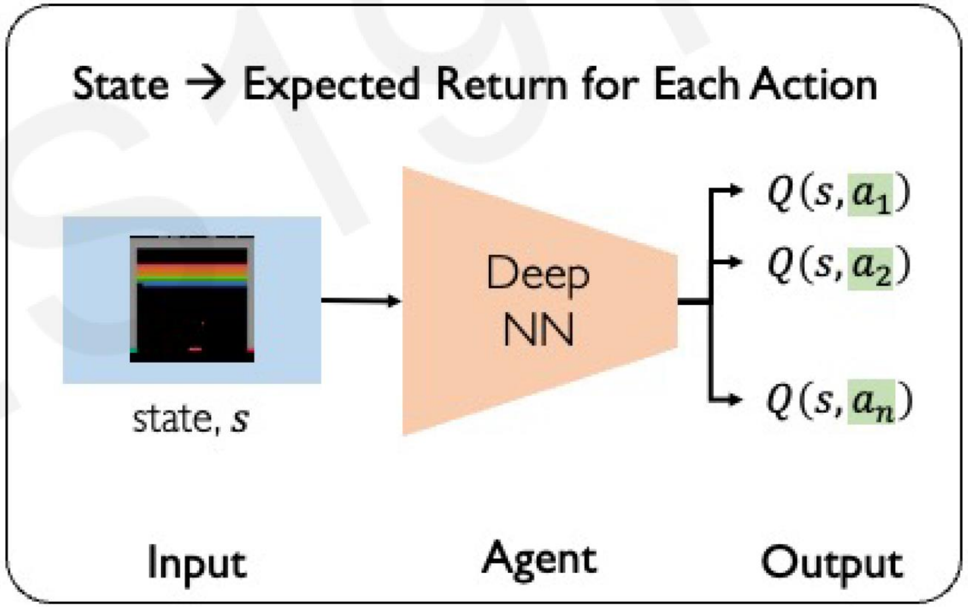


What happens if we take all the best actions?
Maximize target return → train the agent

Deep Q Networks (DQN): Training

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



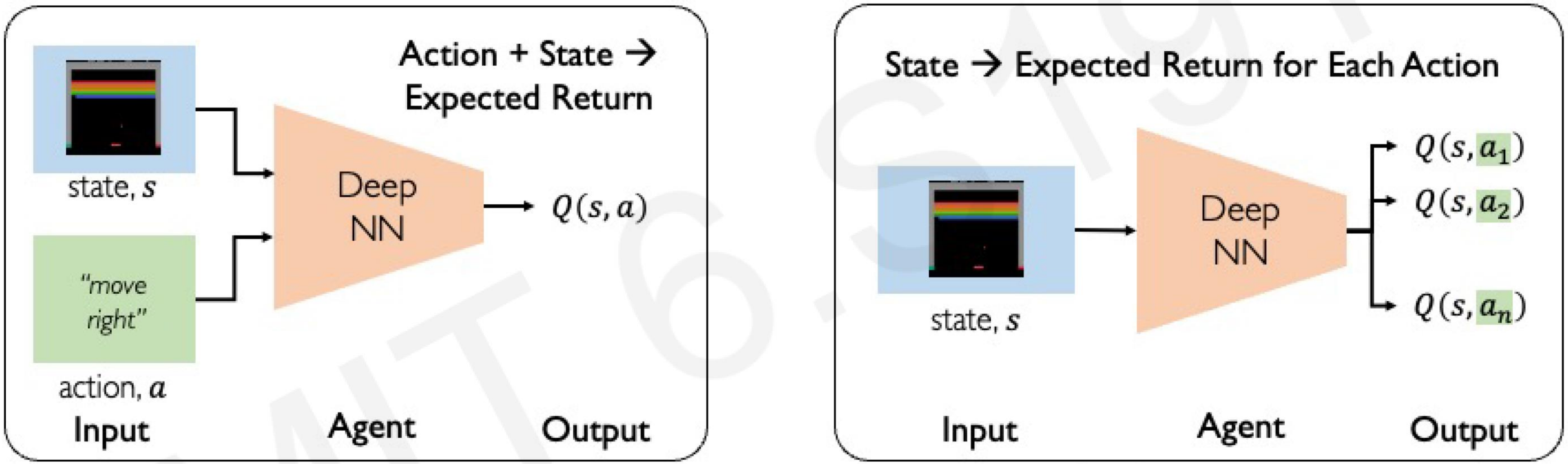


$$\overbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}^{\text{target}}$$

Take all the best actions → target return

Deep Q Networks (DQN): Training

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

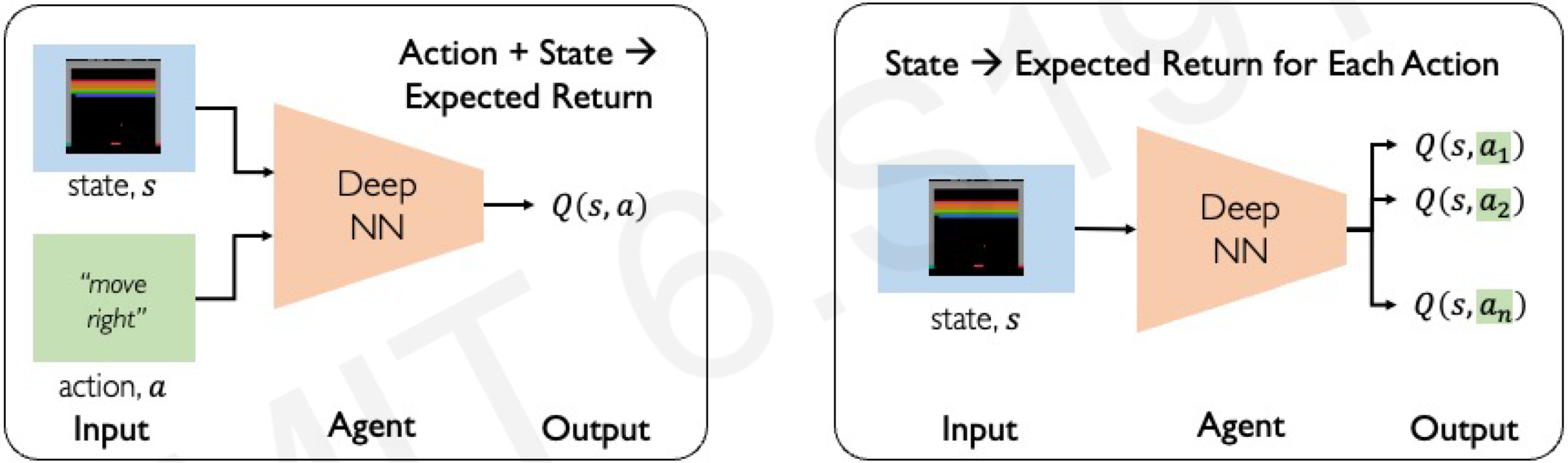


$$\underbrace{\left(r + \gamma \max_{a'} Q(s',a')\right)}_{\text{target}} \quad \underbrace{Q(s,a)}_{\text{predicted}}$$

Network prediction

Deep Q Networks (DQN): Training

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

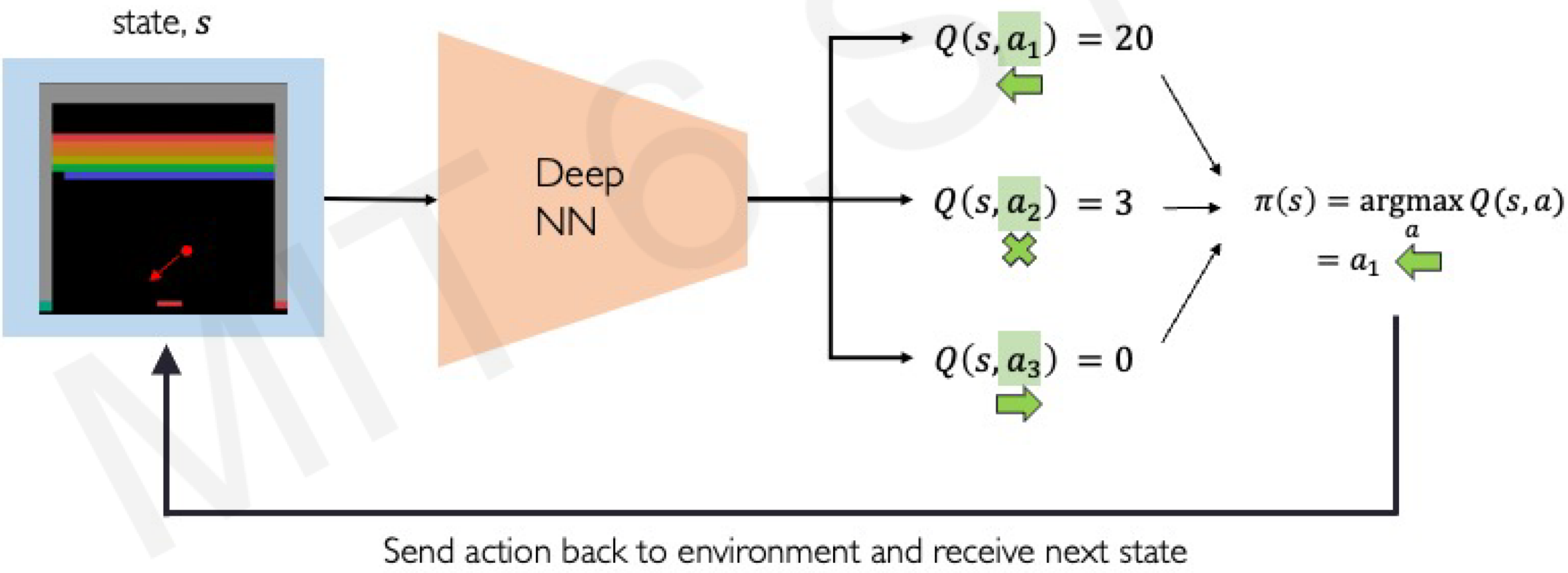


$$\mathcal{L} = \mathbb{E}\left[\left\|\overbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}^{\text{target}} - \overbrace{Q(s, a)}^{\text{predicted}}\right\|^2\right]$$

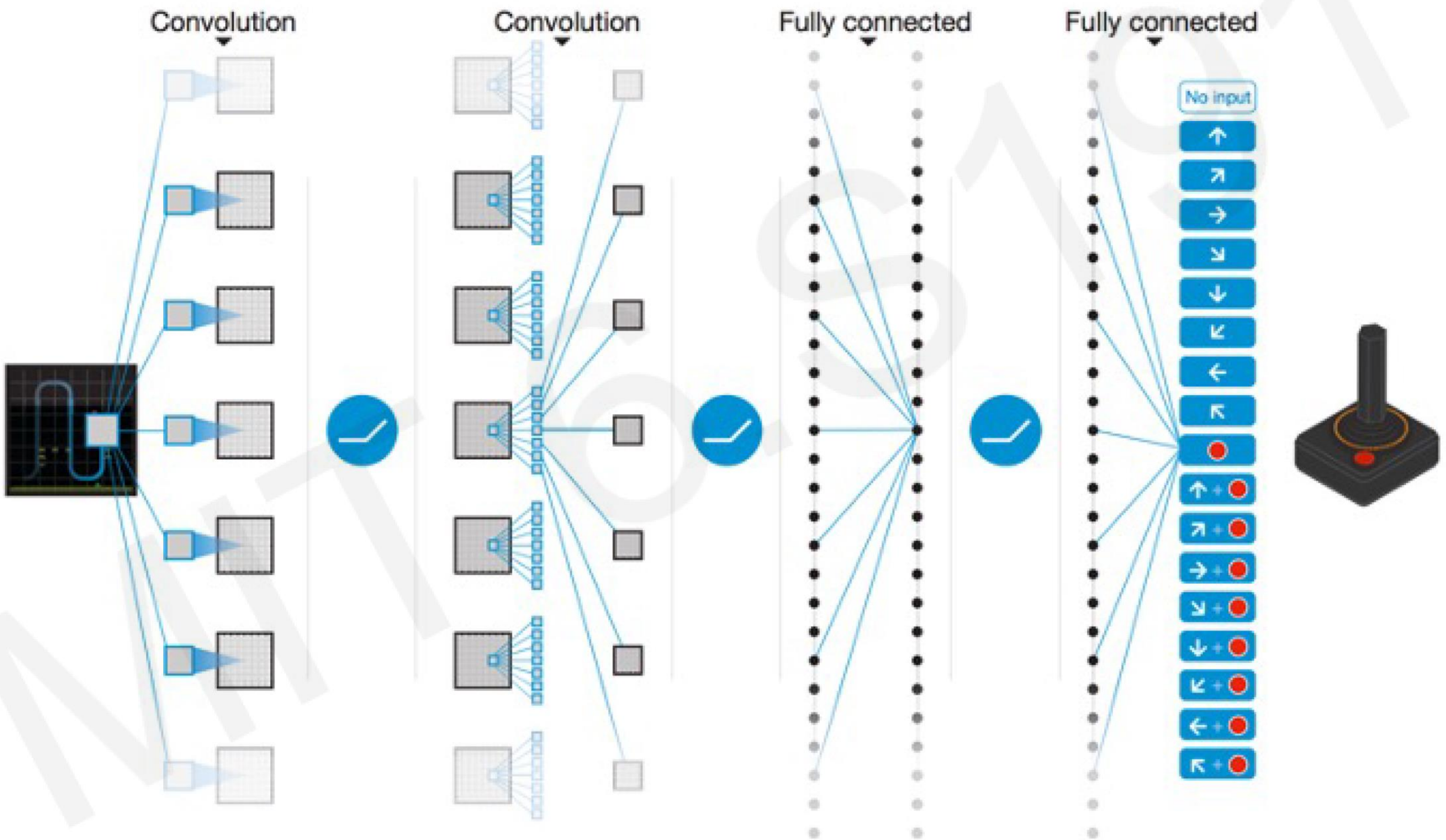
Q-Loss

Deep Q Network Summary

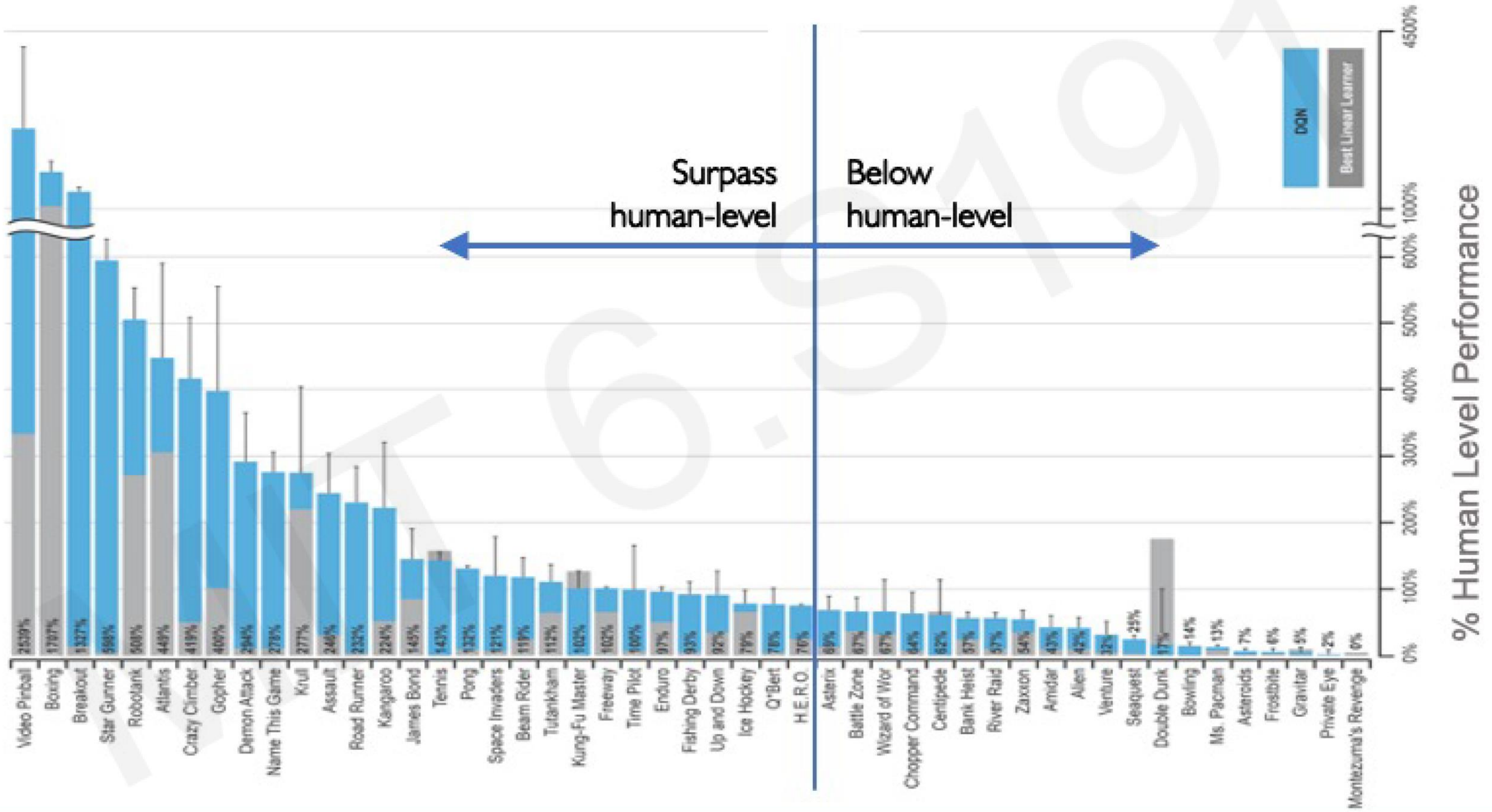
Use NN to learn Q-function and then use to infer the optimal policy, $\pi(s)$



DQN Atari Results



DQN Atari Results



Rainbow DQN

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

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

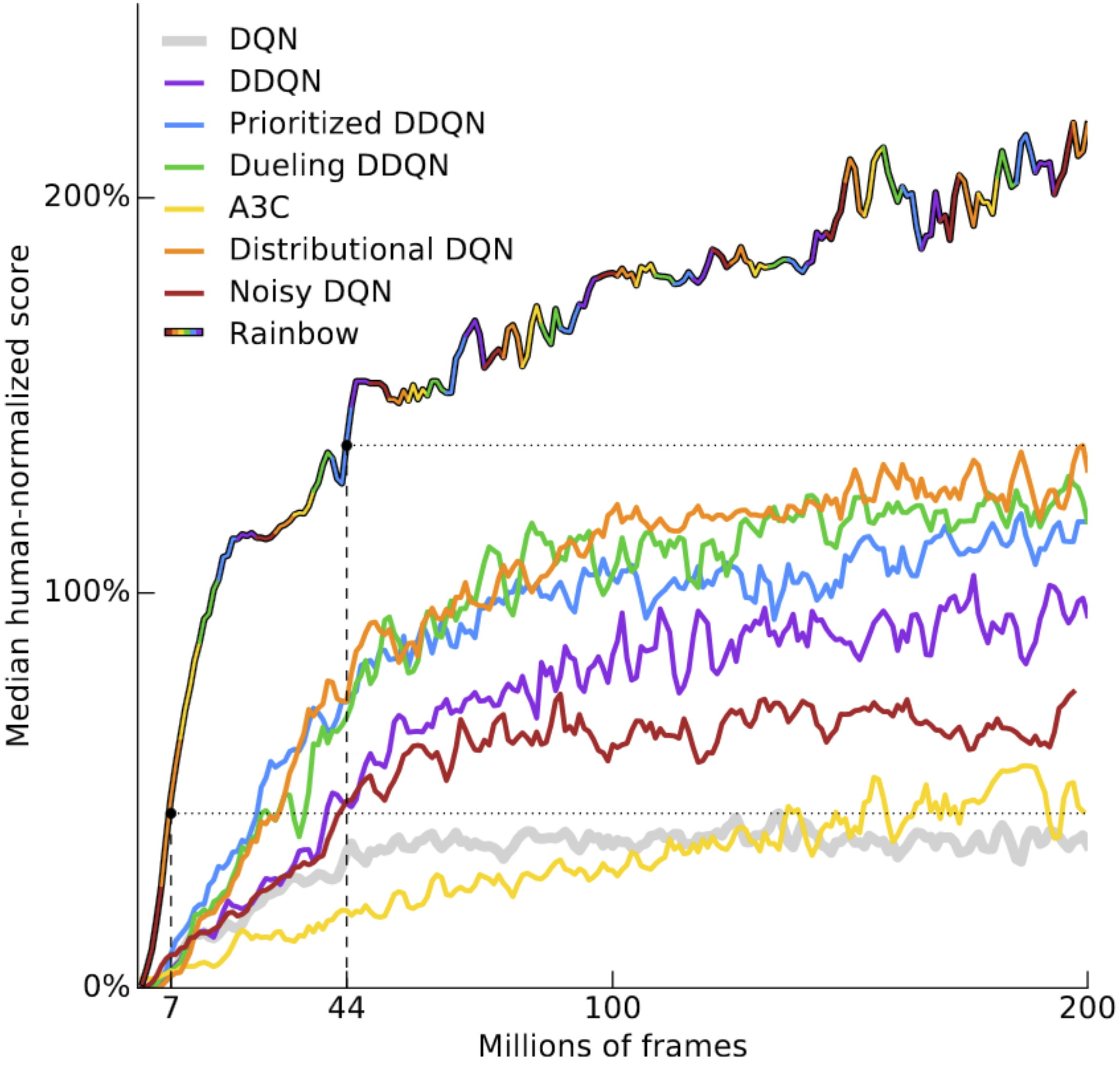
- Prioritized experience

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

- 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

Reinforcement Learning Algorithms

Value Learning

Find $Q(s,a)$

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

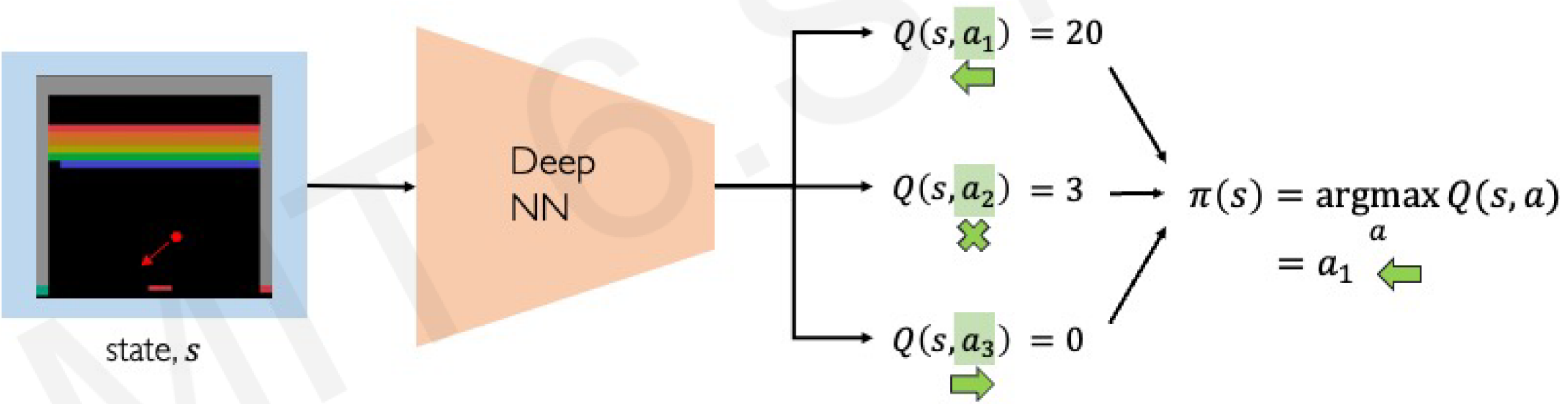
Policy Learning

Find $\pi(s)$

Sample $a \sim \pi(s)$

Deep Q Networks

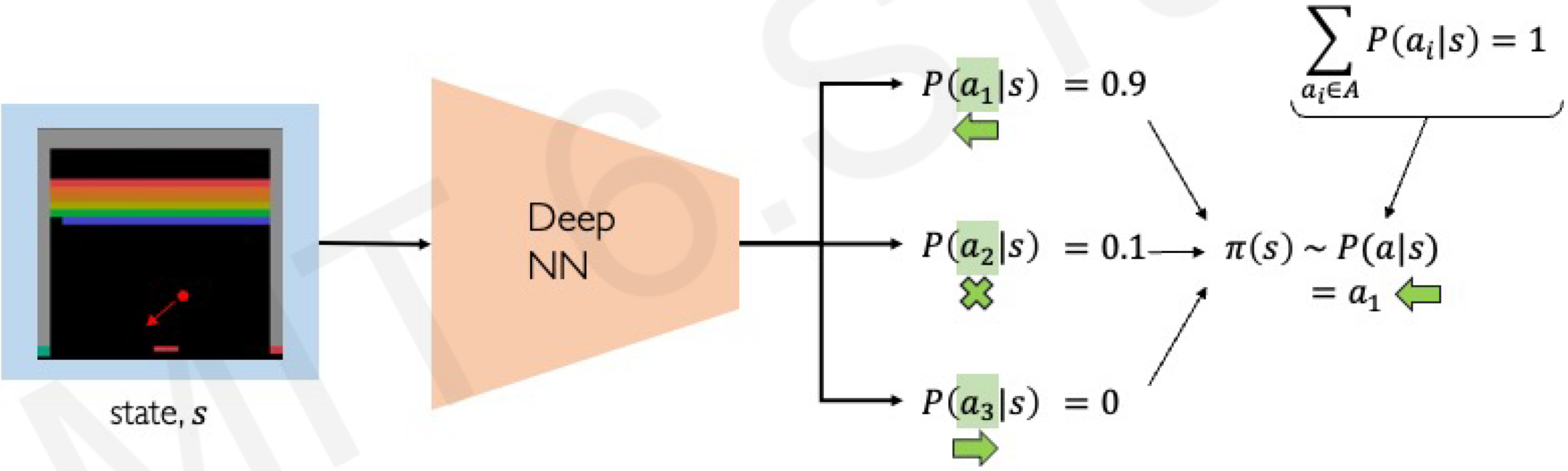
DQN: Approximate Q-function and use to infer the optimal policy, $\pi(s)$



Policy Gradient (PG): Key Idea

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

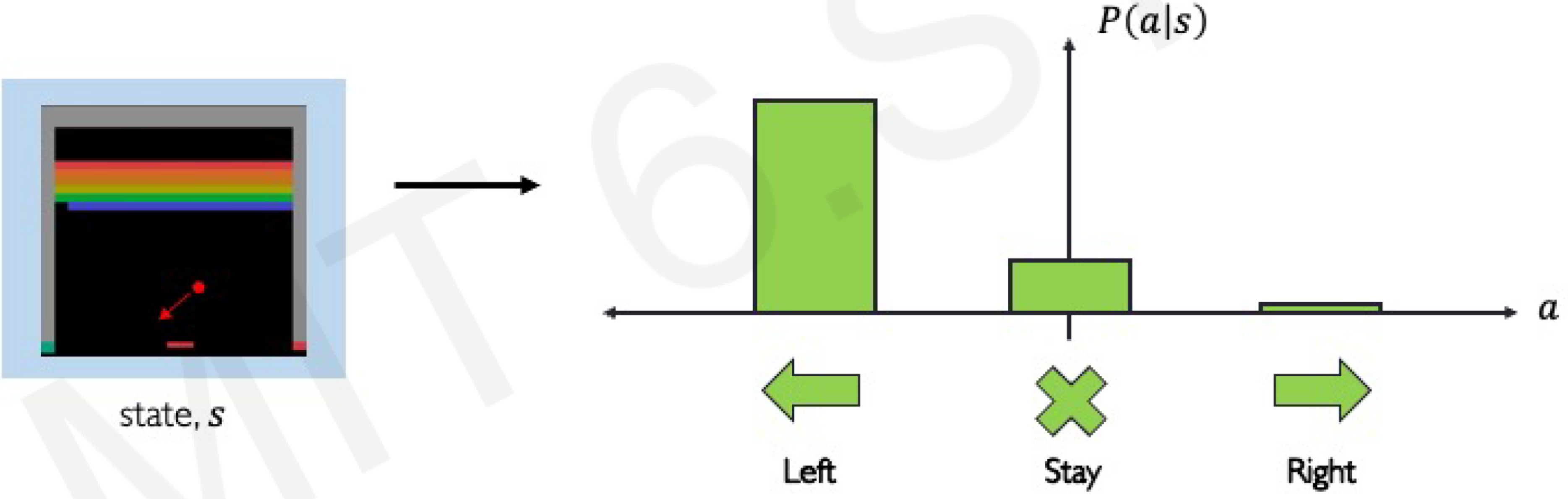
Policy Gradient: Directly optimize the policy $\pi(s)$



What are some advantages of this formulation?

Discrete vs Continuous Action Spaces

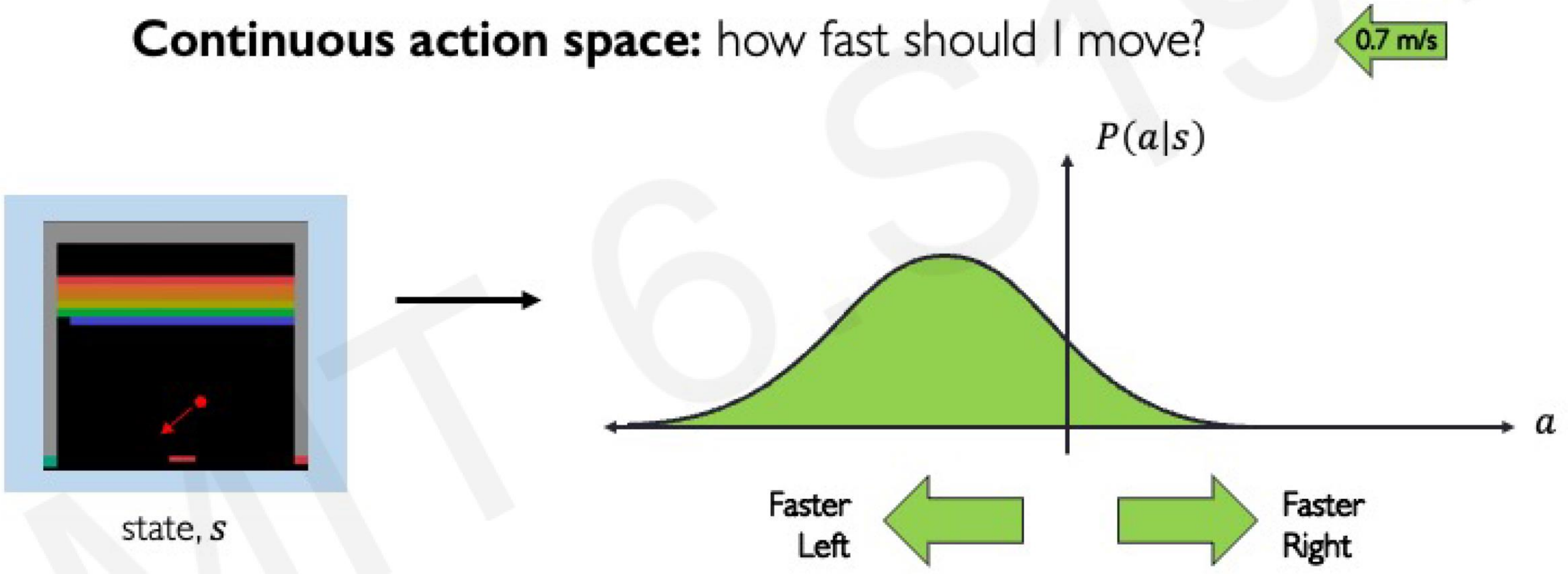
Discrete action space: which direction should I move?



Discrete vs Continuous Action Spaces

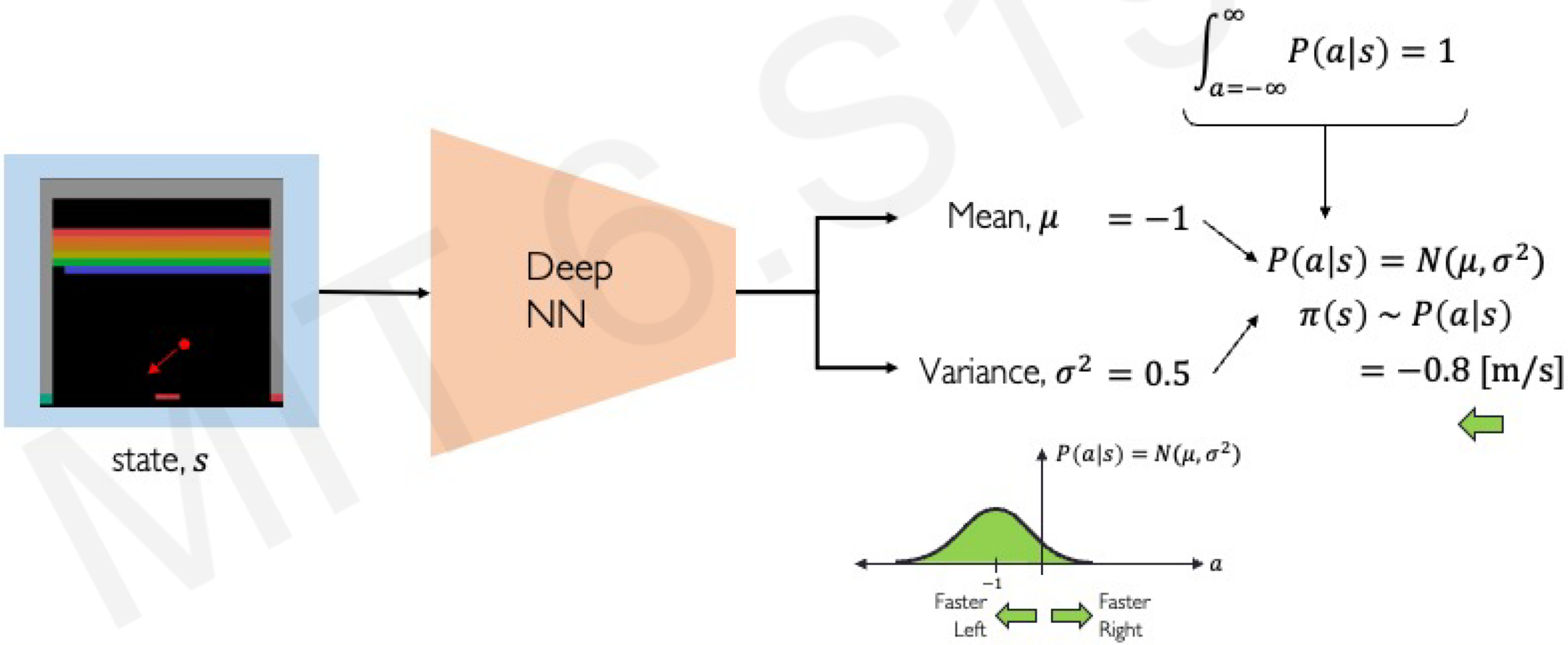
Discrete action space: which direction should I move?

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

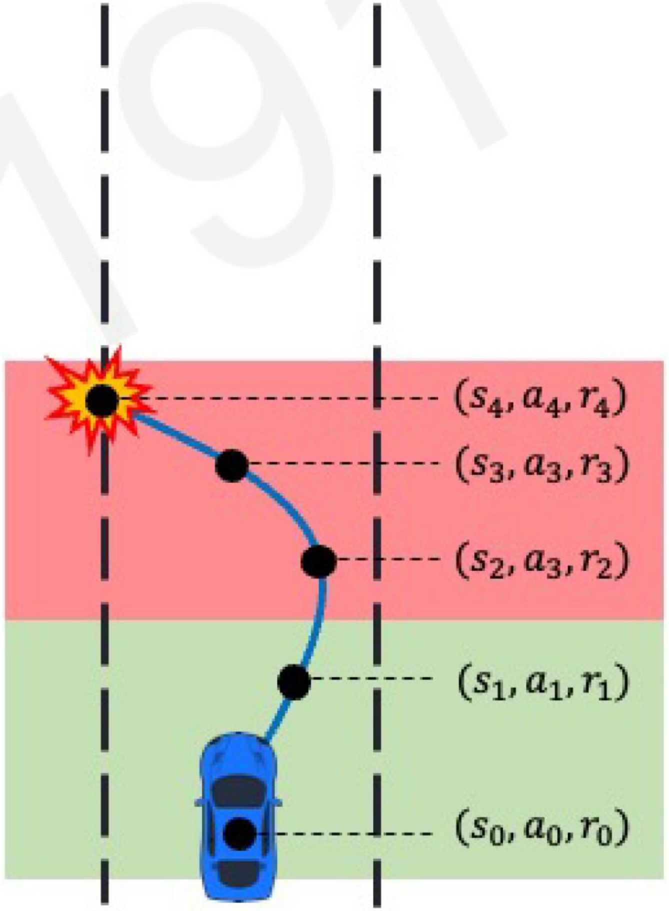
Action: steering wheel angle

Reward: distance traveled

Training Policy Gradients

Training Algorithm

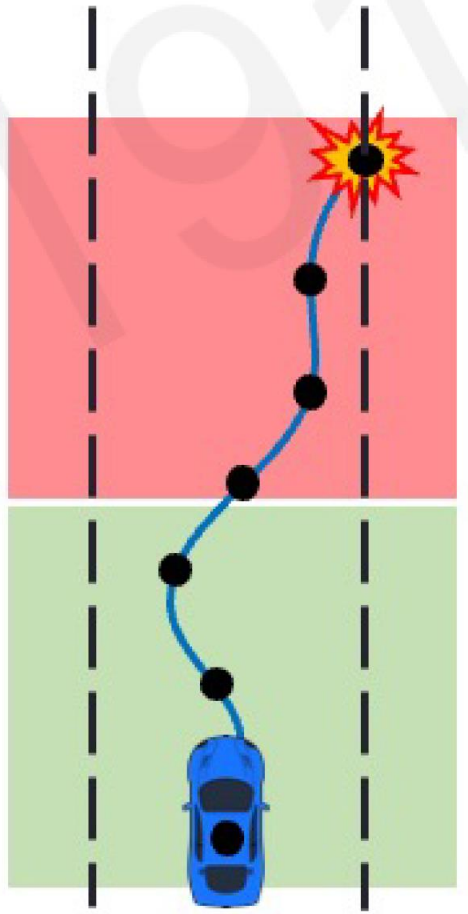
1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



Training Policy Gradients

Training Algorithm

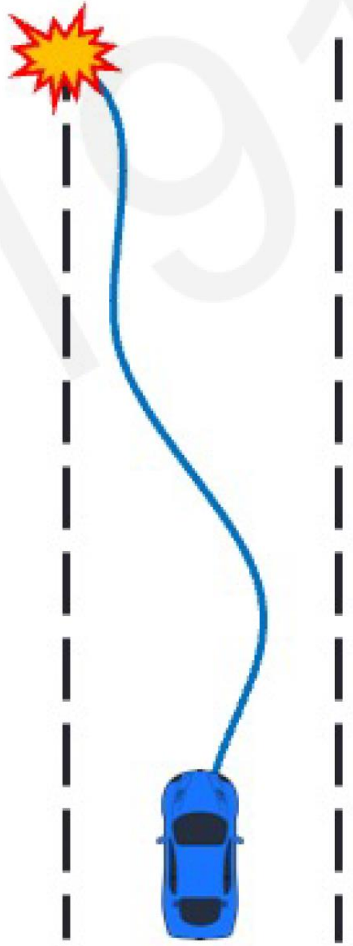
1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



Training Policy Gradients

Training Algorithm

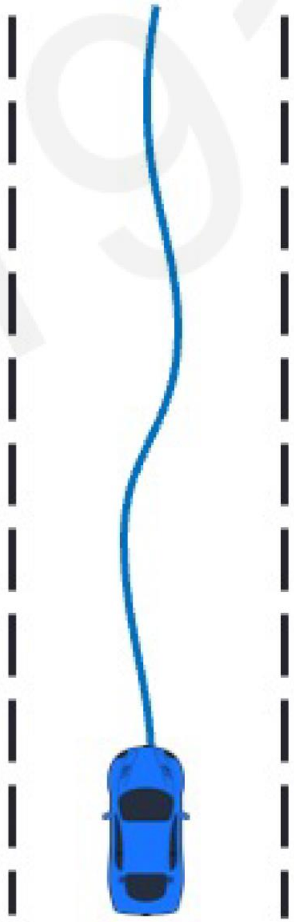
1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



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!

REINFORCE

- Take parameterized policy $\pi_{\theta 0}$
- Sample an episode τ with parameters θ_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: $\pi(s, a, \theta)$

Score function: $R(\tau)$

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

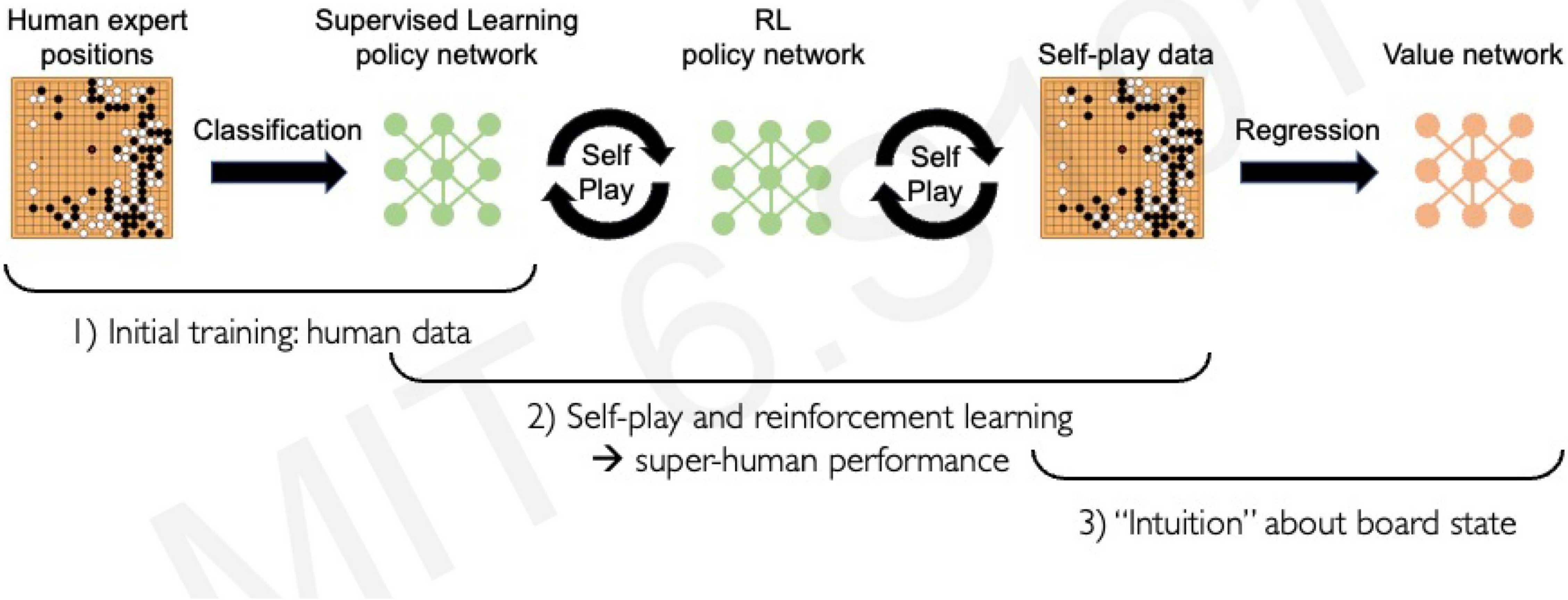
Change in parameters: $\Delta\theta$

Learning rate: α

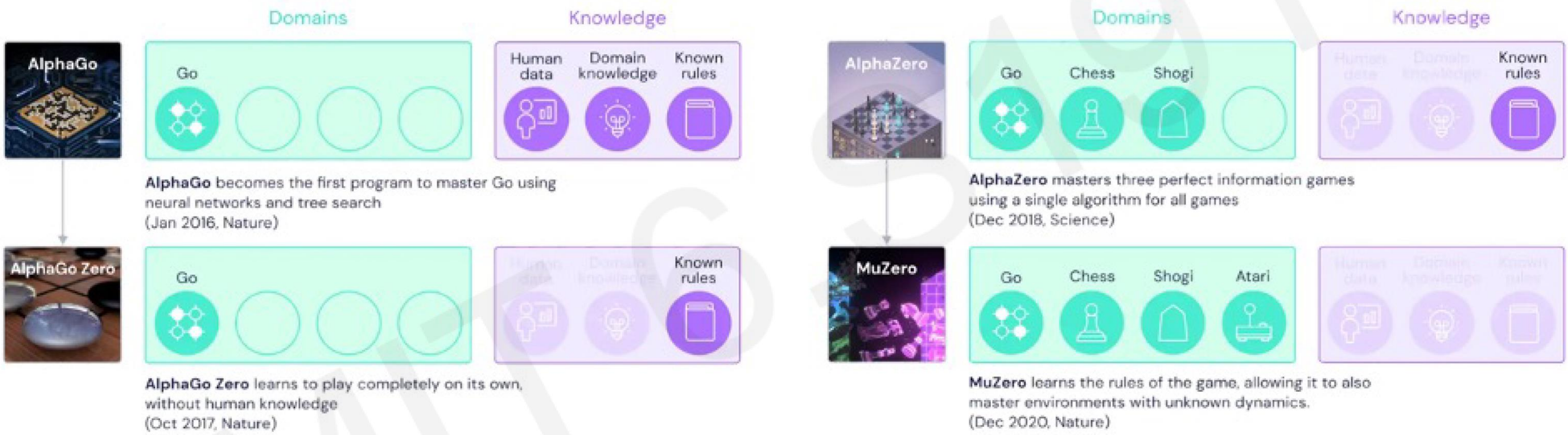
REINFORCE

function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **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 θ
end function

AlphaGo Beats Top Human Player (2016)



MuZero: Learning Dynamics for Planning (2020)



Deep Reinforcement Learning Summary

Foundations

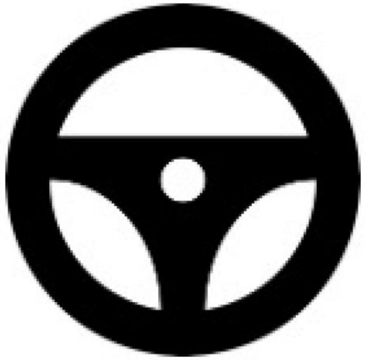
- Agents acting in environment
- State-action pairs → maximize future rewards
- Discounting

Q-Learning

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

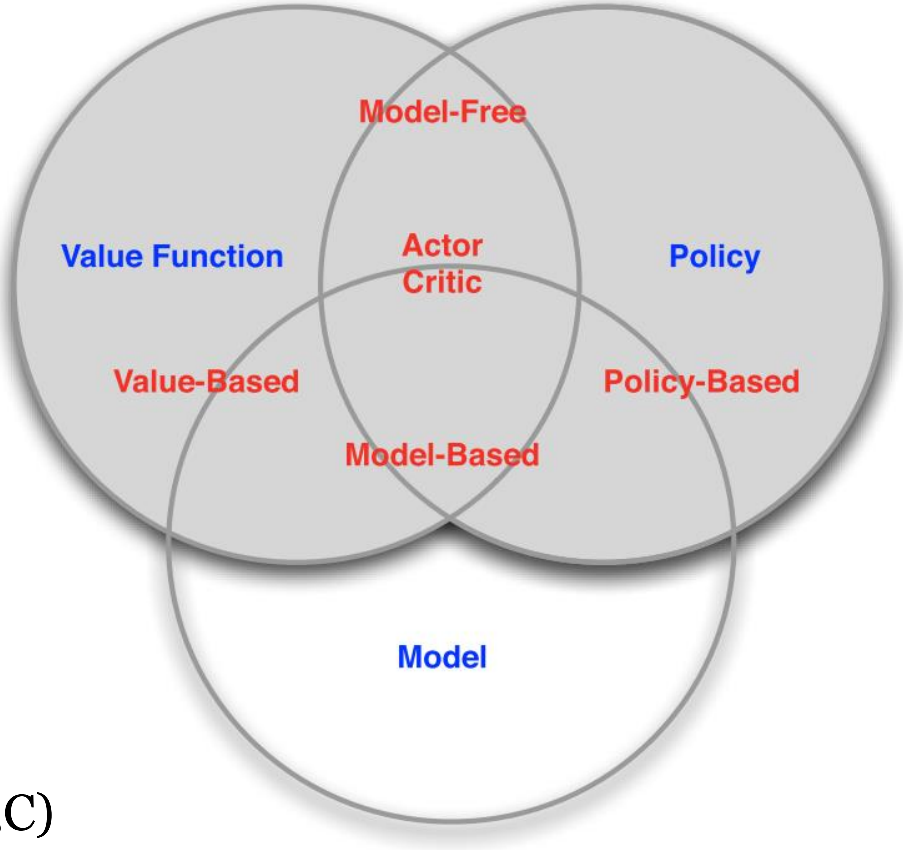
Policy Gradients

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

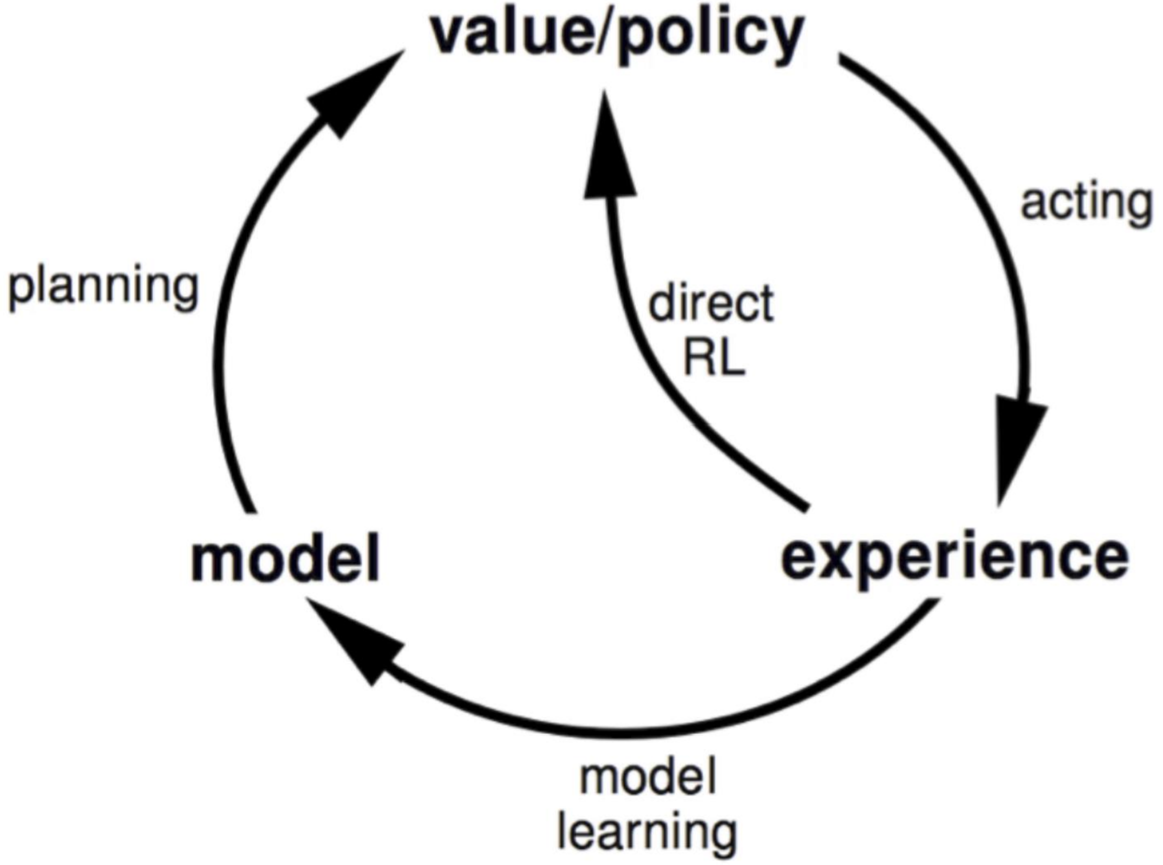


Reinforcement Learning Approaches

- 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



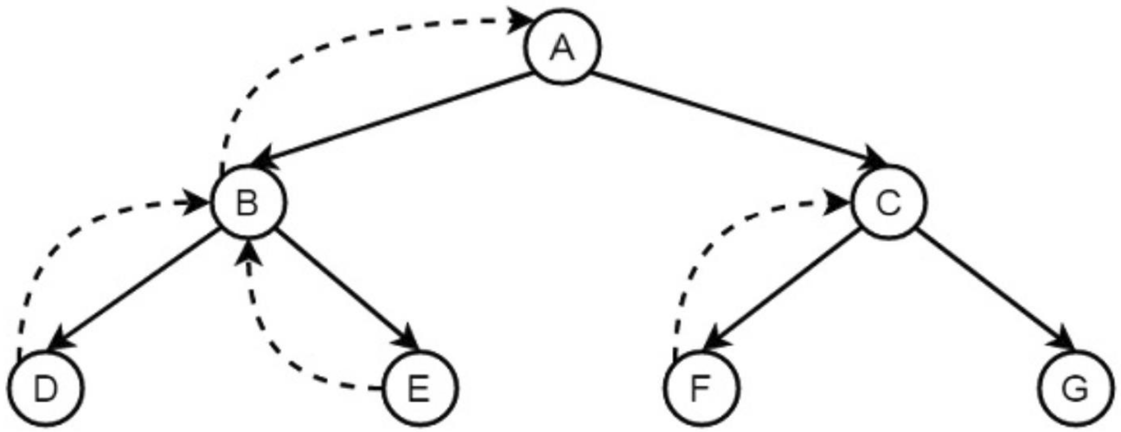
Learn *policy* direct or learn *transition* first and then policy?

Learning Policies vs Learning Transitions

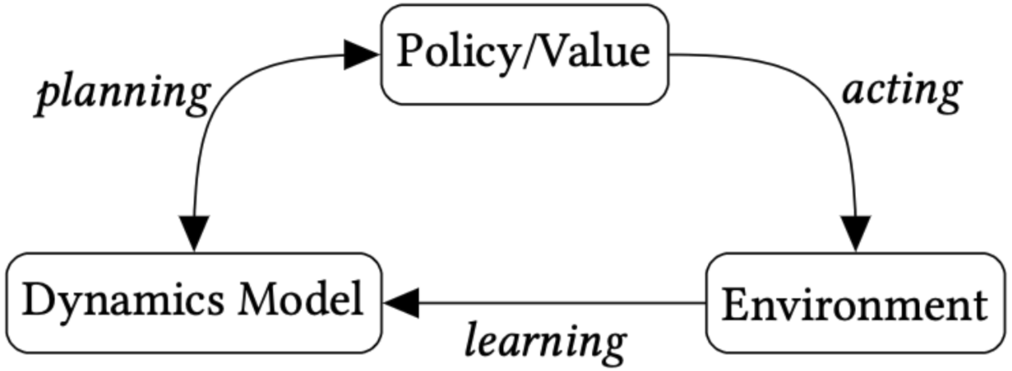
- $s \rightarrow a \rightarrow s' \rightarrow a' \rightarrow s'' \rightarrow a'' \rightarrow s''' \rightarrow a''' \rightarrow s''''$
- Learning a policy $s \rightarrow a$
  - Learning how to react in an environment
- Learning a transition $s \rightarrow a \rightarrow s'$
  - Learning how the environment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

Learning vs Planning

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



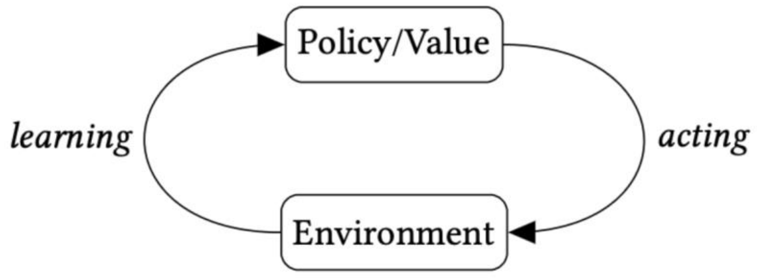
Model-Based RL



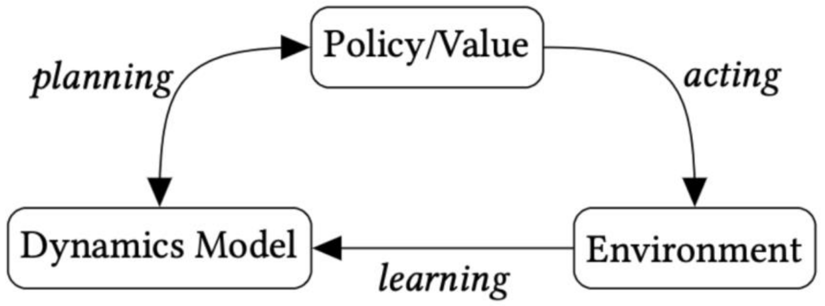
repeat
  Sample environment E to generate data $D = (s, a, r', s')$
  Use D to learn $M = T_a(s, s'), R_a(s, s')$ ▹ learning
  for $n = 1, \ldots, N$ **do**
    Use M to update policy $\pi(s, a)$ ▹ planning
  end for
until π converges

Model-Based RL

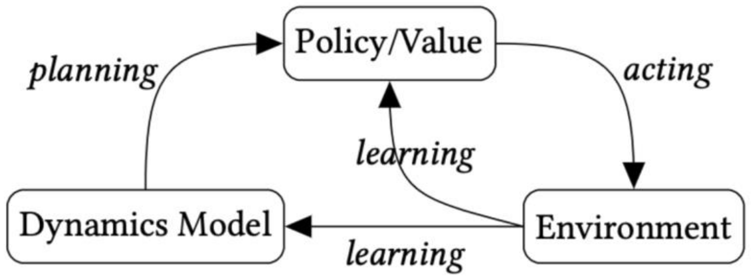
Learn policy directly



Learn model
and then plan actions



Use experience to
update both model and policy

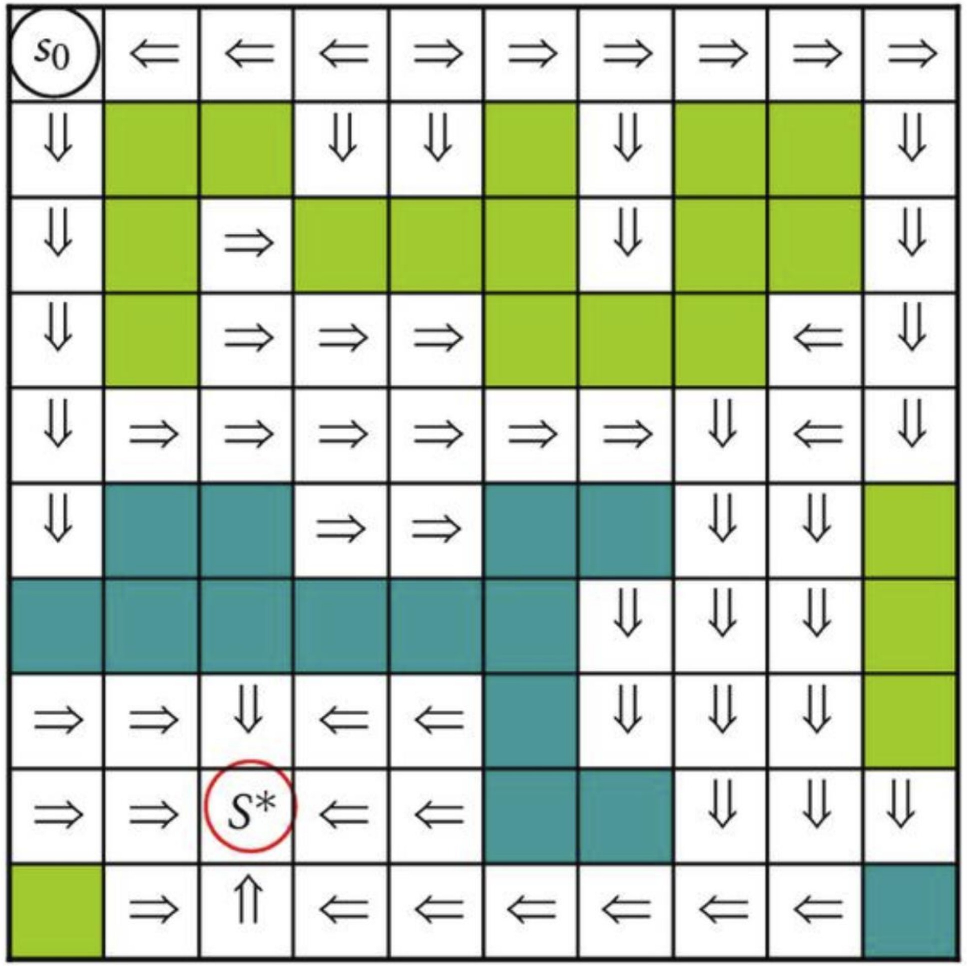


Dyna [Sutton]

- Initialize Q-function
- Repeat
  - Initialize s; a←π(s); (s',r)←Env(s,a) :: **Learn**
  - Q(s,a) ← Q(s,a)+α[r+γmax$_a$Q(s',$_a$)-Q(s,a)]
  - M(s,a) ← (s',r) :: **Model**
  - For n=1, …, N :
    - Select $\hat{s}$ and $\hat{a}$ randomly
    - (s',r) ← M($\hat{s}$,$\hat{a}$) :: **Plan for FREE!**
    - Q($\hat{s}$,$\hat{a}$) ← Q($\hat{s}$,$\hat{a}$)+α[r+γmax$_a$Q(s',$_a$)-Q($\hat{s}$,$\hat{a}$)]
- Until Q converges
- return Q

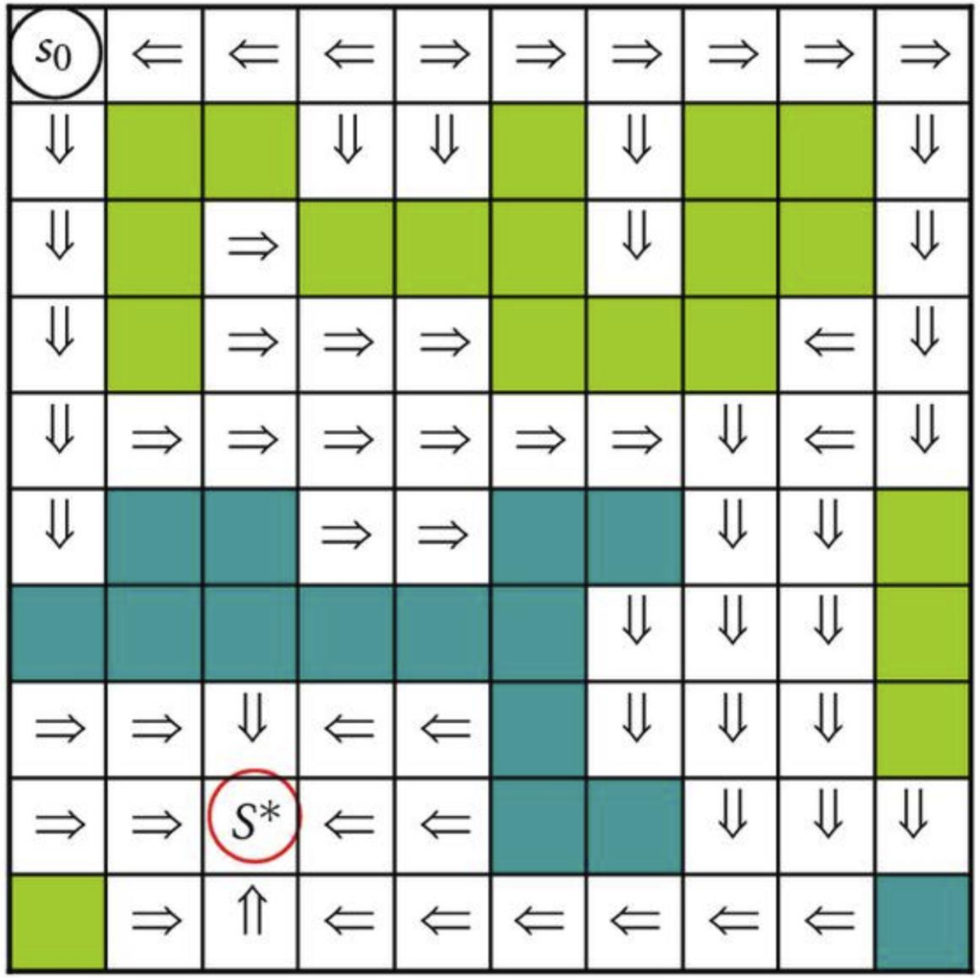
Example Model-Free RL

- Initialize Q-function
- For All Episodes:
  - Initialize s
  - For All Time Steps in this Episode:
    - Select a ϵ-greedy from Q(s)
    - Perform a in Environment giving s' and r
    - $Q(s,a) \leftarrow Q(s,a)+\alpha[r+\gamma \max_a Q(s',a)-Q(s,a)]$
    - $s \leftarrow s'$
- return Q



Example Model-Based RL

- Initialize Q-function
- Repeat
  - Initialize s; a←π(s); (s',r)←Env(s,a) :: **Learn**
  - Q(s,a) ← Q(s,a)+α[r+γmax$_a$Q(s',$_a$)-Q(s,a)]
  - M(s,a) ← (s',r) :: **Model**
  - For n=1, …, N :
    - Select $\hat{s}$ and $\hat{a}$ randomly
    - (s',r) ← M($\hat{s}$,$\hat{a}$) :: **Plan for FREE!**
    - Q($\hat{s}$,$\hat{a}$) ← Q($\hat{s}$,$\hat{a}$)+α[r+γmax$_a$Q(s',$_a$)-Q($\hat{s}$,$\hat{a}$)]
- Until Q converges
- return Q



Sample Complexity

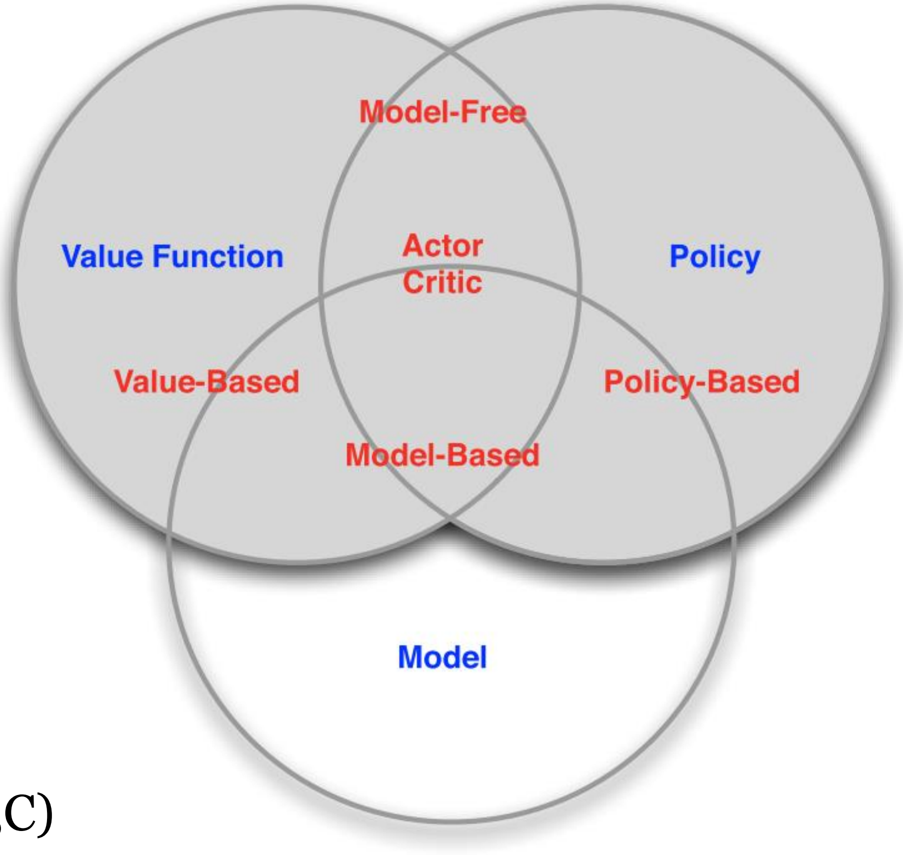
- Model-based RL reduces sample complexity.
- As soon as Model has enough transition entries, the policy can be learned from the Model, for free.
- This free learning is called planning. It does not involve environment samples, hence, “free”.

Dreamer (Latent/Traj)

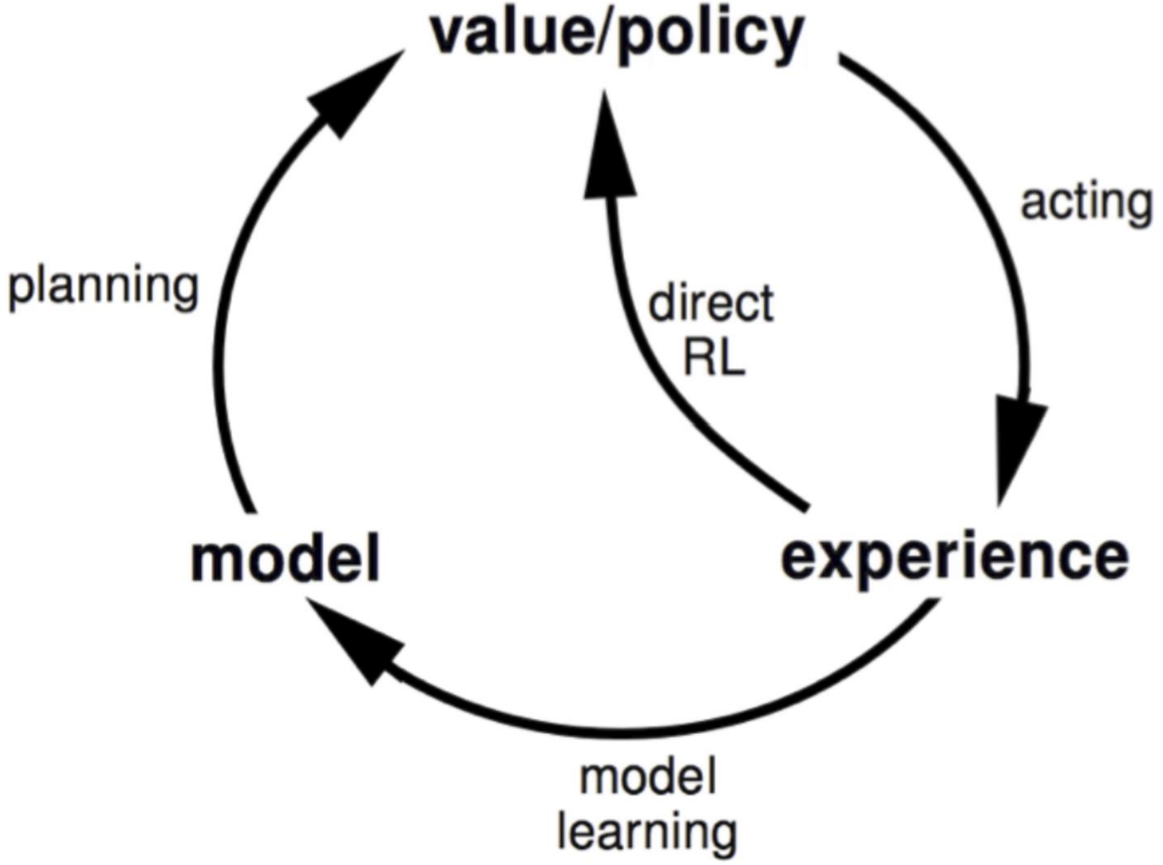
The three processes of the Dreamer agent. The world model is learned from past experience. From predictions of this model, the agent then learns a value network to predict future rewards and an actor network to select actions. The actor network is used to interact with the environment.

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

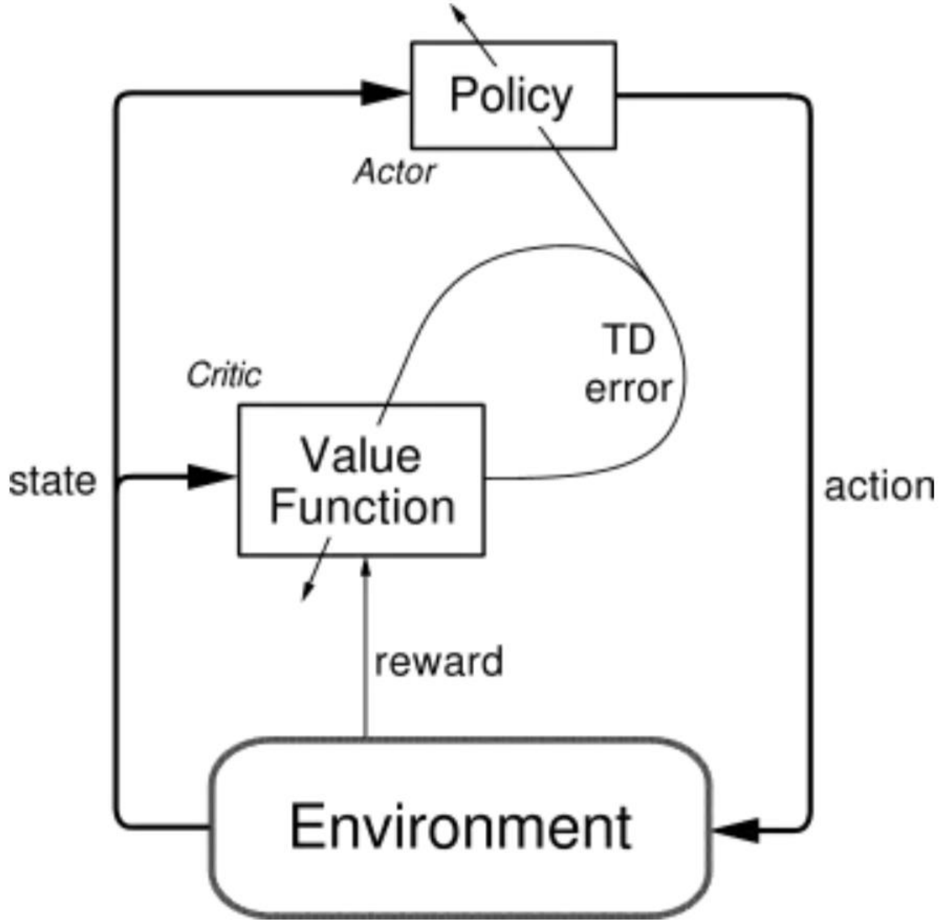


Learn *policy* direct or learn *transition* first and then policy?

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



Advantage Variants

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_0 \sim p_\theta(\tau_0)}\left[\sum_{t=0}^{n} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

- Targets:

$$\Psi_t = \hat{Q}_{MC}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^i \cdot r_i \qquad \text{Monte Carlo target}$$

$$\Psi_t = \hat{Q}_n(s_t, a_t) = \sum_{i=t}^{n-1} \gamma^i \cdot r_i + \gamma^n V_\theta(s_n) \qquad \text{bootstrap (n-step target)}$$

$$\Psi_t = \hat{A}_{MC}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^i \cdot r_i - V_\theta(s_t) \qquad \text{baseline subtraction}$$

$$\Psi_t = \hat{A}_n(s_t, a_t) = \sum_{i=t}^{n-1} \gamma^i \cdot r_i + \gamma^n V_\theta(s_n) - V_\theta(s_t) \qquad \text{baseline + bootstrap}$$

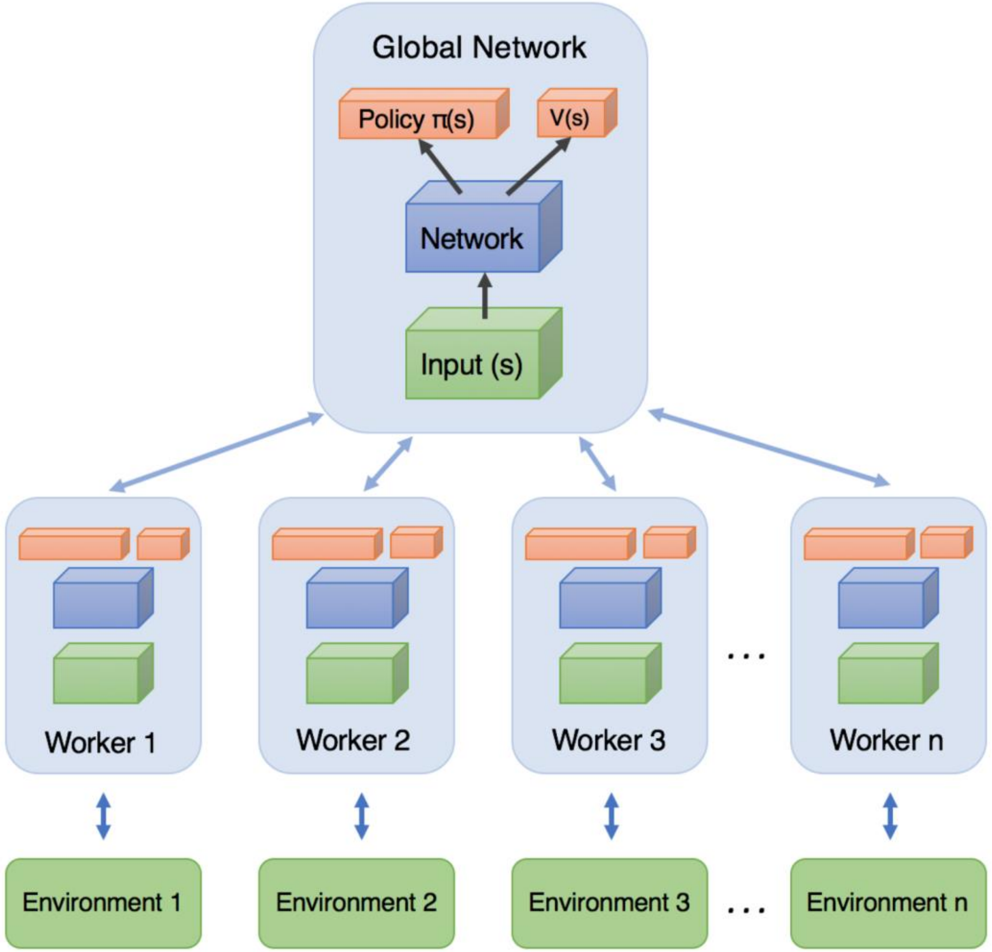
$$\Psi_t = Q_\phi(s_t, a_t) \qquad \text{Q-value approximation}$$

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 π 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

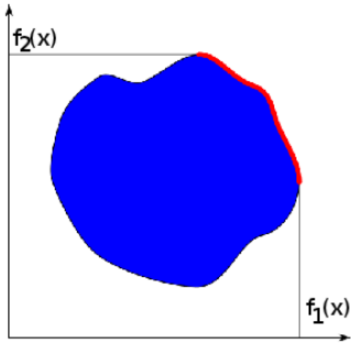


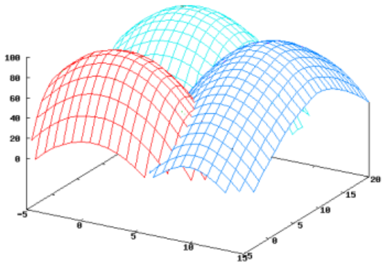
https://pylessons.com/A3C-reinforcement-learning

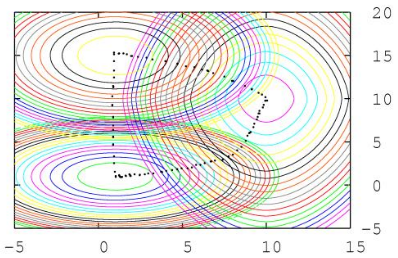
Multi-Objective Reinforcement Learning

Multi-Objective Reinforcement Learning (MORL)

- Many real-world tasks may present an agent with multiple, possibly conflicting objectives:
  - Time
  - Safety
  - Resource consumption
- Multi-Objective Reinforcement Learning allows an agent to learn how to prioritize among objectives at runtime
- Possible to create diverse populations of agents, or adapt agents to time-varying user needs, e.g. difficulty level or training session contents
- Training goals can also be considered by agents

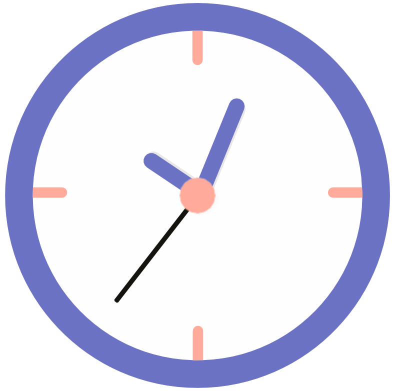






J. Källström and F.Heintz, Tunable Dynamics in Agent-Based Simulation using Multi-Objective Reinforcement Learning, AAMAS Adaptive and Learning Agents Workshop 2019.

Objectives



Self-Driving Car Objectives

DESTINATION

Reach the desired destination on time

2

SAFETY

Travel in a safe manner, without any incidents of collisions

3

COMFORT

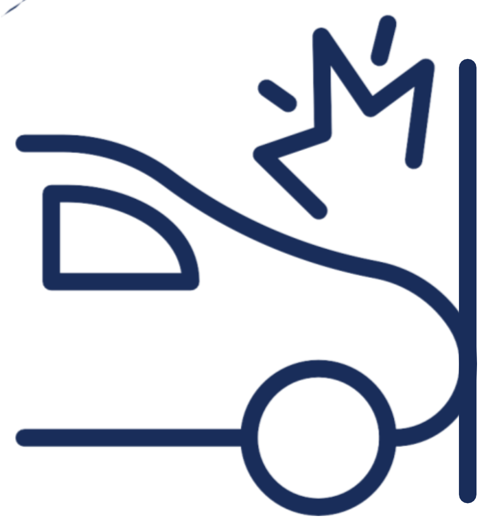
The trip should be smooth, no excessive breaking or jerking

4

CONSUMPTION

The driving style should minimise fuel comsumption

Reward Design



How to specify the reward for a collision?

- \- 10?
- \- 100?
- \- 100 000?

- This dilemma of reward scale exists for each behaviour we want to encourage:
  - avoiding collisions
  - reaching the destination on time
  - staying within speed limits
  - avoiding sudden changes in speed
  - driving within the lane
  - …

Scalar Reward Design Process

1. Design/tweak scalar reward function
2. (Re-)Train RL agent using new/updated reward function (may take hours or days)
3. Evaluate performance (and try to figure out what went wrong!)

- Repeat until the desired agent behaviour is (finally) learned
- Wasteful and time consuming process – each trained agent must be discarded if the reward function changes
- Designers implicitly bake in tradeoffs between different behaviours
- Should AI engineers make the decisions about these tradeoffs?

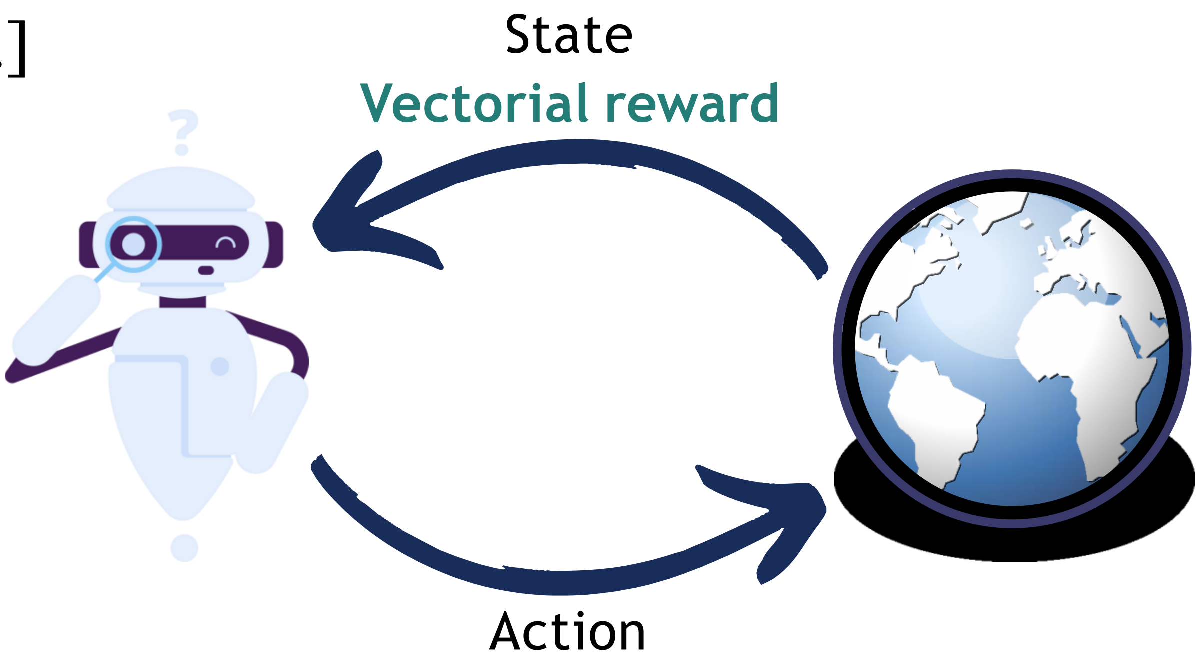
Scalar Reward Design Process

1. Design/tweak scalar reward function
2. (Re-)Train RL agent using new/updated reward function (may take hours or days)
3. Evaluate performance (and try to figure out what went wrong!)

- GT Sophy - Super Human Racing AI Agent, Sony AI
- Objectives: high precision race car control, efficient racing tactics and maneuvers, while respecting an imprecisely defined racing etiquette
- With enough time and computation, good results can be achieved:
- Could we have done better?

Multi-Objective Reinforcement Learning

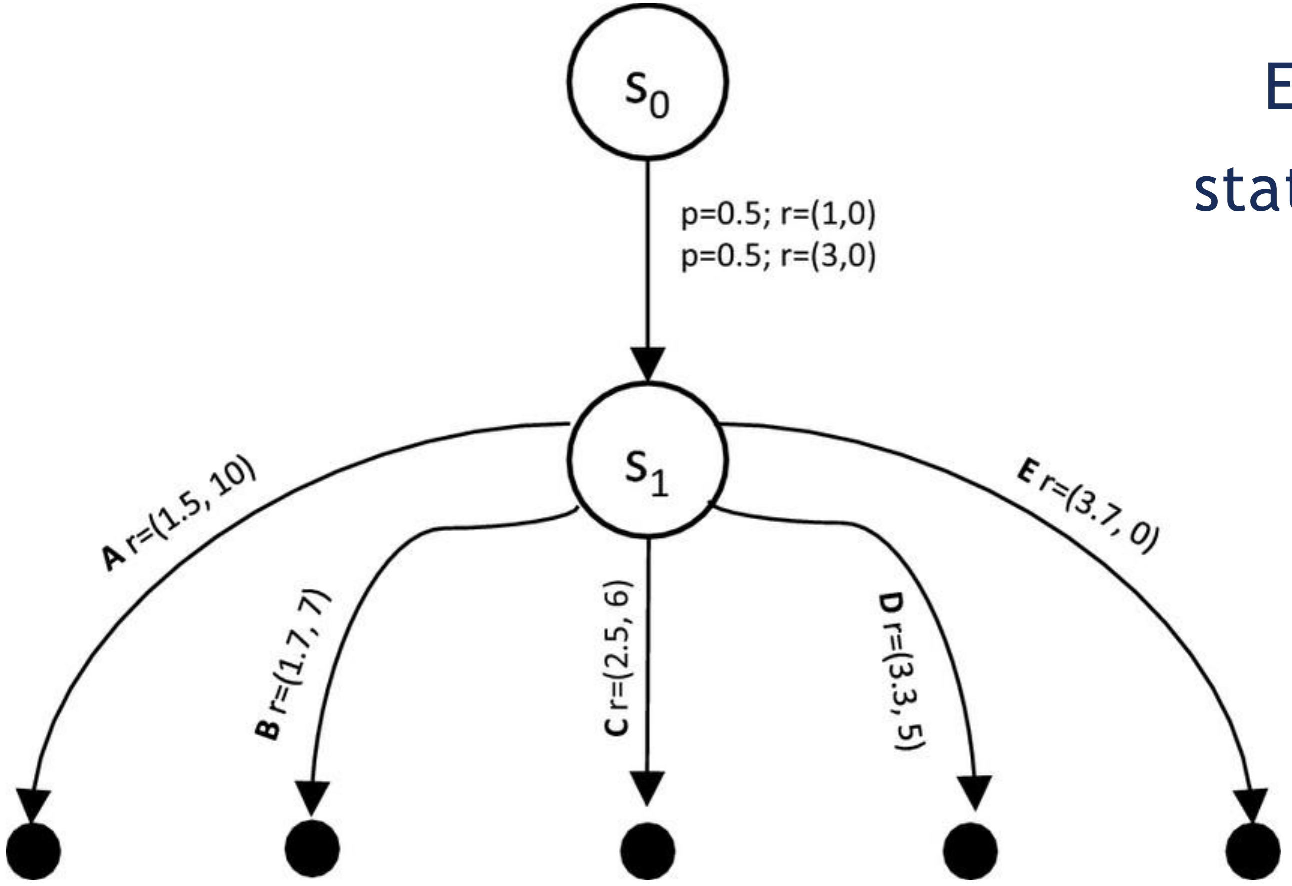
- Vector-valued reward function
  - r = [r_objective1, r_objective2, ...]
- Length of the reward vector = number of objectives



Multi-Objective Reinforcement Learning

- Multi-Objective MDP $< S, A, T, \gamma, \mathbf{R} >$
  - Set of states
  - Set of actions
  - A vectorial reward function $\mathbf{R}: S \times A \times S \to \mathbb{R}^d$
  - $d \geq 2$ objectives
  - Transition function (dynamics of the environment)
  - Discount factor $\gamma \in [0, 1]$

Multi-Objective Reinforcement Learning



Example: MOMDP with deterministic state transitions and stochastic rewards

Vamplew, P., Foale, C., & Dazeley, R. (2022). The impact of environmental stochasticity on value-based multiobjective reinforcement learning. Neural Computing and Applications, 1-17.

Value Functions and Policies

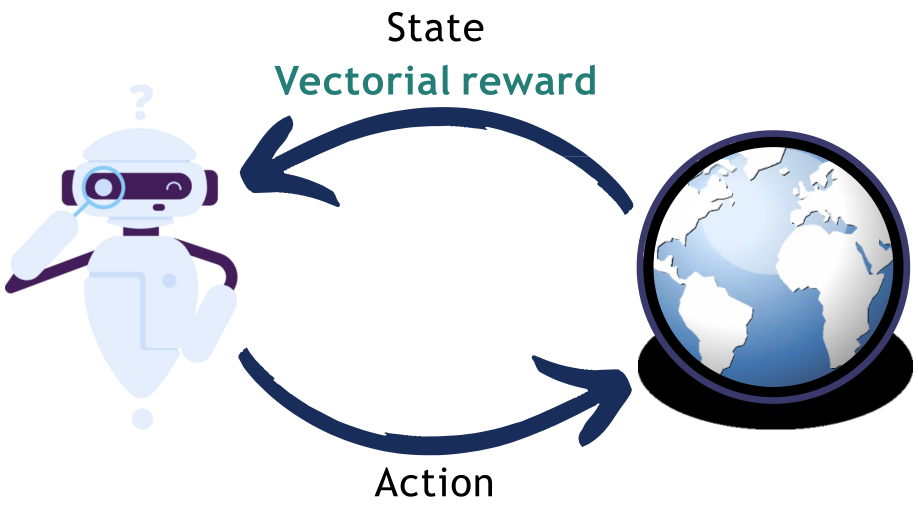
- The agent behaves according to a policy:

$$\pi : S \times A \rightarrow [0, 1]$$

- The value function of a policy in a MOMDP:

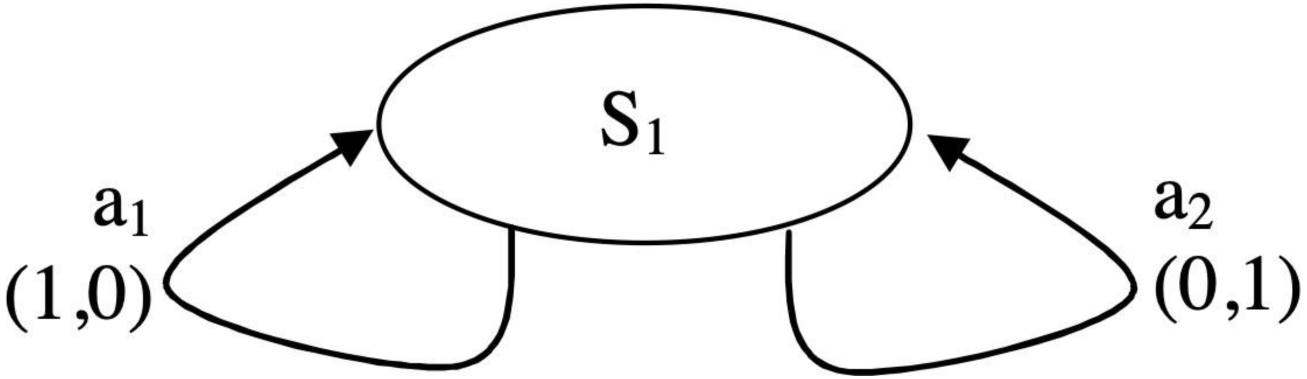
$$\mathbf{V}^{\pi} = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k \mathbf{r}_{k+1} \,|\, \pi, \mu\right]$$

where $\mathbf{r}_{k+1} = \mathbf{R}(s_k, a_k, s_{k+1})$



Deterministic vs Stochastic Policies

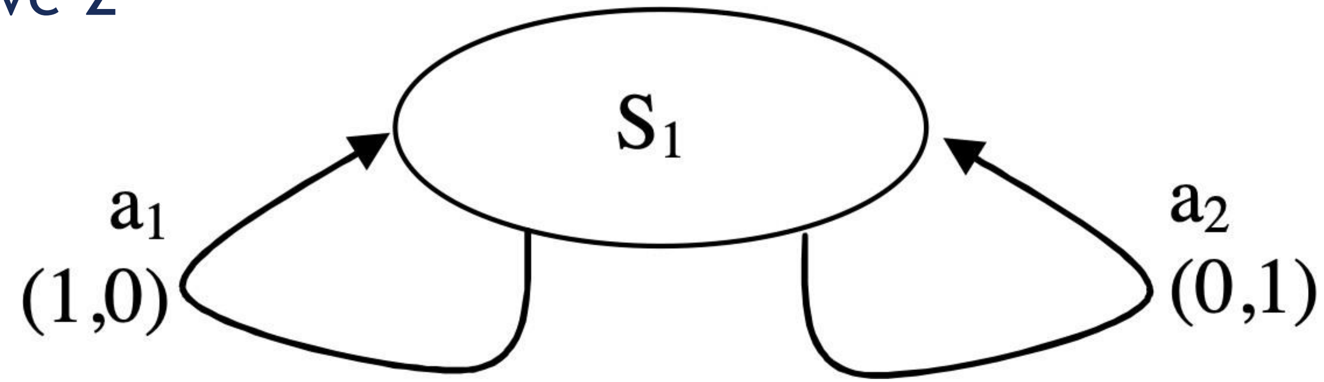
- Episodic task, 2 objectives
- 2 deterministic policies: $\pi_i = a_i$



Vamplew, P., Dazeley, R., Barker, E., & Kelarev, A. (2009). Constructing stochastic mixture policies for episodic multiobjective reinforcement learning tasks. In AI 2009: Advances in Artificial Intelligence. Proceedings 22 (pp. 340-349). Springer Berlin Heidelberg.

Deterministic vs Stochastic Policies

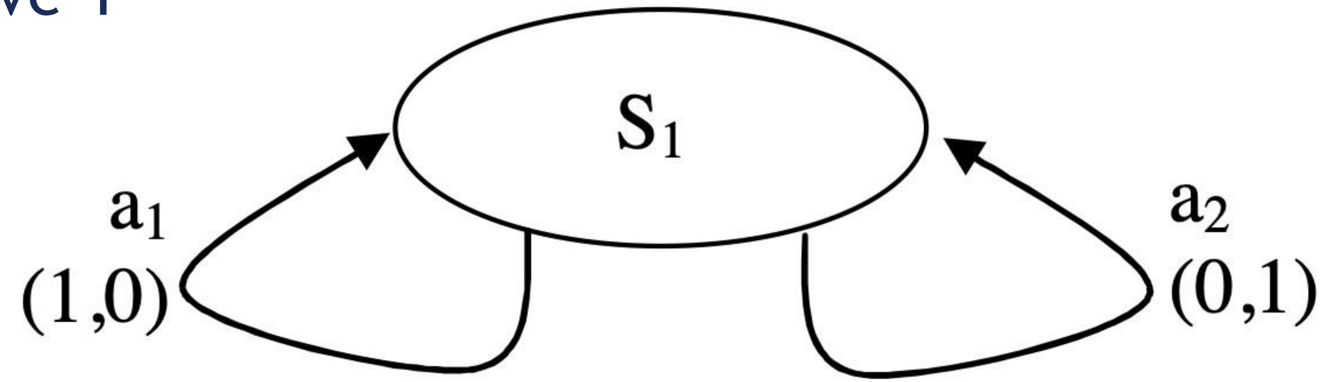
- Episodic task, 2 objectives
- 2 deterministic policies: $\pi_i = a_i$
- Always choosing action a1 will maximise the reward on objective 1, but minimise the reward for objective 2



Vamplew, P., Dazeley, R., Barker, E., & Kelarev, A. (2009). Constructing stochastic mixture policies for episodic multiobjective reinforcement learning tasks. In AI 2009: Advances in Artificial Intelligence. Proceedings 22 (pp. 340-349). Springer Berlin Heidelberg.

Deterministic vs Stochastic Policies

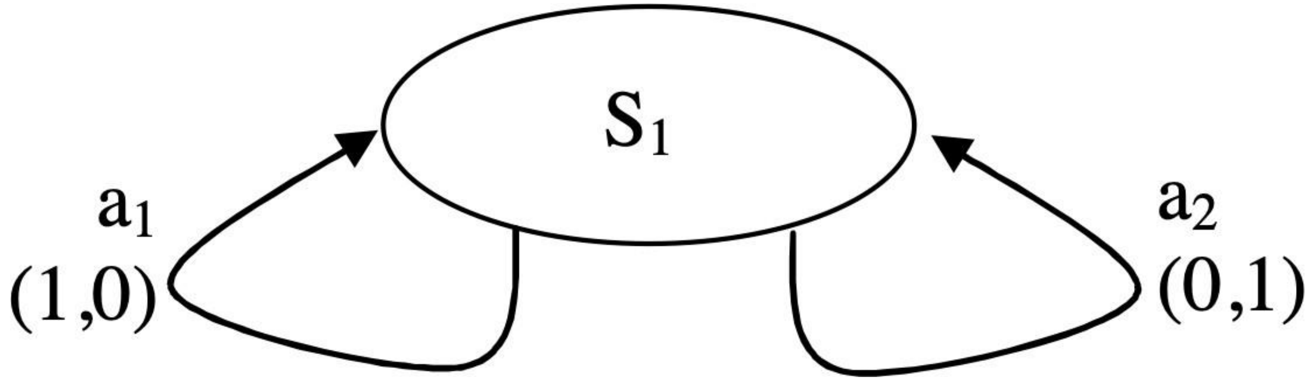
- Episodic task, 2 objectives
- 2 deterministic policies: $\pi_i = a_i$
- Always choosing action a2 will maximise the reward on objective 2, but minimise the reward for objective 1



Vamplew, P., Dazeley, R., Barker, E., & Kelarev, A. (2009). Constructing stochastic mixture policies for episodic multiobjective reinforcement learning tasks. In AI 2009: Advances in Artificial Intelligence. Proceedings 22 (pp. 340-349). Springer Berlin Heidelberg.

Deterministic vs Stochastic Policies

- Consider a stochastic policy which selects between actions a1 and a2 with probabilities p1 and (1-p1)
- The average reward received by this policy will be (p1, 1-p1)



Vamplew, P., Dazeley, R., Barker, E., & Kelarev, A. (2009). Constructing stochastic mixture policies for episodic multiobjective reinforcement learning tasks. In AI 2009: Advances in Artificial Intelligence. Proceedings 22 (pp. 340-349). Springer Berlin Heidelberg.

Value Functions and Policies

- Vectorial value functions now supply only a partial ordering, even for a given state:

$$V_i^{\pi}(s) > V_i^{\pi'}(s) \text{ but } V_j^{\pi}(s) < V_j^{\pi'}(s)$$

- We can no longer determine which values are optimal without additional information about how to prioritize the objectives

Utility Functions in Multi-Object Decision Making (MODeM)

- A utility function, u, is used to represent a user's preferences over objectives
  - Utility function maps a vector reward to a scalar utility:

$$u : \mathbb{R}^d \rightarrow \mathbb{R}$$

  - For MODeM, a utility function, u is assumed to be monotonically increasing:

$$(\forall o, V_o^\pi \geq V\pi'_o) \implies u(\mathbf{V}^\pi) \geq u\left(\mathbf{V}^{\pi'}\right)$$

Utility Functions

- **Linear utility function:**

$$u(\mathbf{V}^{\pi}) = \mathbf{w}^{\top}\mathbf{V}^{\pi}$$

- Each element w specifies how much one unit of value for the corresponding objective contributes to the scalarised value
- The elements of the weight vector are all positive real numbers and sum to 1

Utility Functions

- Examples of non-linear utility functions
  - The product utility function
    - seeks to make the objective values as balanced as possible
    - [3, 1] ? [2, 2]

$$u(\mathbf{V}^\pi) = \prod_{o=1}^{d} V_o^\pi$$

Utility Functions

- Examples of non-linear utility functions
  - The product utility function
    - seeks to make the objective values as balanced as possible
    - [3, 1] ? [2, 2]
  - The sum of squares utility function
    - tends to prioritise achieving higher values on a single objective at the expense of other objectives
    - [3, 1] ? [2, 2]

$$u(\mathbf{V}^{\pi}) = \prod_{o=1}^{d} V_o^{\pi}$$

$$u(\mathbf{V}^{\pi}) = \sum_{o=1}^{d} {V_o^{\pi}}^2$$

Utility Functions

- Examples of non-linear utility functions
  - The product utility function
    - seeks to make the objective values as balanced as possible
    - u([3, 1]) < u([2, 2])
  - The sum of squares utility function
    - tends to prioritise achieving higher values on a single objective at the expense of other objectives
    - u([3, 1]) > u([2, 2])

$$u(\mathbf{V}^{\pi}) = \prod_{o=1}^{d} V_o^{\pi}$$

$$u(\mathbf{V}^{\pi}) = \sum_{o=1}^{d} {V_o^{\pi}}^2$$

Solution Sets

- In single-objective RL problems, there exist a unique optimal value V, and there can be multiple optimal policies π that all have this value
- The goal is to learn one of these optimal policies
- In multi-objective settings there can now be multiple possibly optimal value vectors **V**

- We need to reason about **sets of possibly optimal value vectors and policies** when thinking about solutions to MORL problems

Solution Sets – Undominated Set

- The most general set of solutions: the undominated set
- The undominated set, U, is the subset of all possible policies Π and associated value vectors for which there exists a possible utility function u with a maximal scalarised value:

$$U(\Pi) = \left\{ \pi \in \Pi \;\middle|\; \exists u, \forall \pi' \in \Pi : u(\mathbf{V}^{\pi}) \geq u(\mathbf{V}^{\pi'}) \right\}$$

Solution Sets – Coverage Set

- The undominated set may contain excess policies
- We do not need to retain all policies to retain optimal utility

- A set CS is a ***coverage set*** if it is a subset of U and if, for every u, it contains a policy with maximal scalarised value:

$$CS(\Pi) \subseteq U(\Pi) \wedge \left(\forall u, \exists \pi \in CS(\Pi), \forall \pi' \in \Pi : u(\mathbf{V}^{\pi}) \geq u(\mathbf{V}^{\pi'}) \right)$$

Solution Sets – Pareto Front

- If the utility function u is any monotonically increasing function, then the **Pareto Front** (PF) is the undominated set:

$$PF(\Pi) = \{\pi \in \Pi \mid \nexists \pi' \in \Pi : \mathbf{V}^{\pi'} \succ_P \mathbf{V}^{\pi}\}$$

where $\succ_P$ is the Pareto dominance relationship:

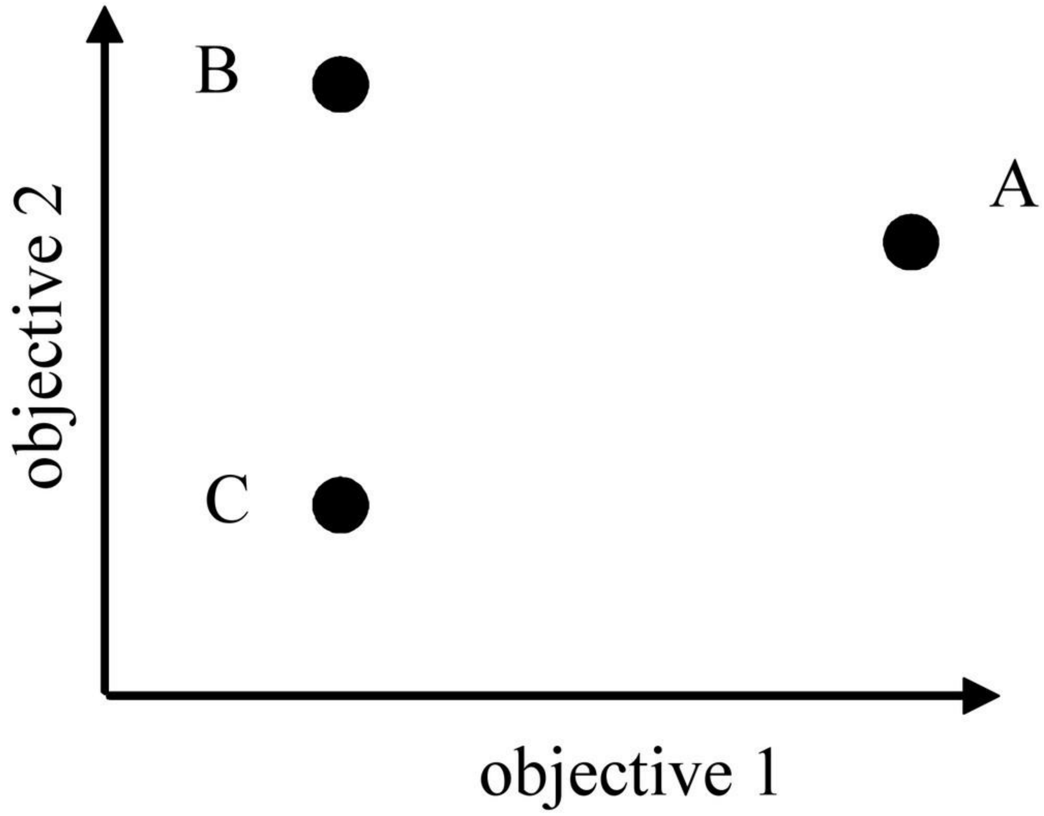
$$\mathbf{V}^{\pi} \succ_P \mathbf{V}^{\pi'} \iff (\forall i : \mathbf{V}_i^{\pi} \geq \mathbf{V}_i^{\pi'}) \wedge (\exists i : \mathbf{V}_i^{\pi} > \mathbf{V}_i^{\pi'})$$

Solution Sets – Pareto Coverage Set

- The definition of Pareto dominance corresponds exactly to the definition of monotonically increasing value functions
- Again, we can retain one of the policies that have the same value vector
- A set of policies whose value functions correspond to the PF is called a Pareto Coverage Set (PCS)

Solution Sets – Pareto Front

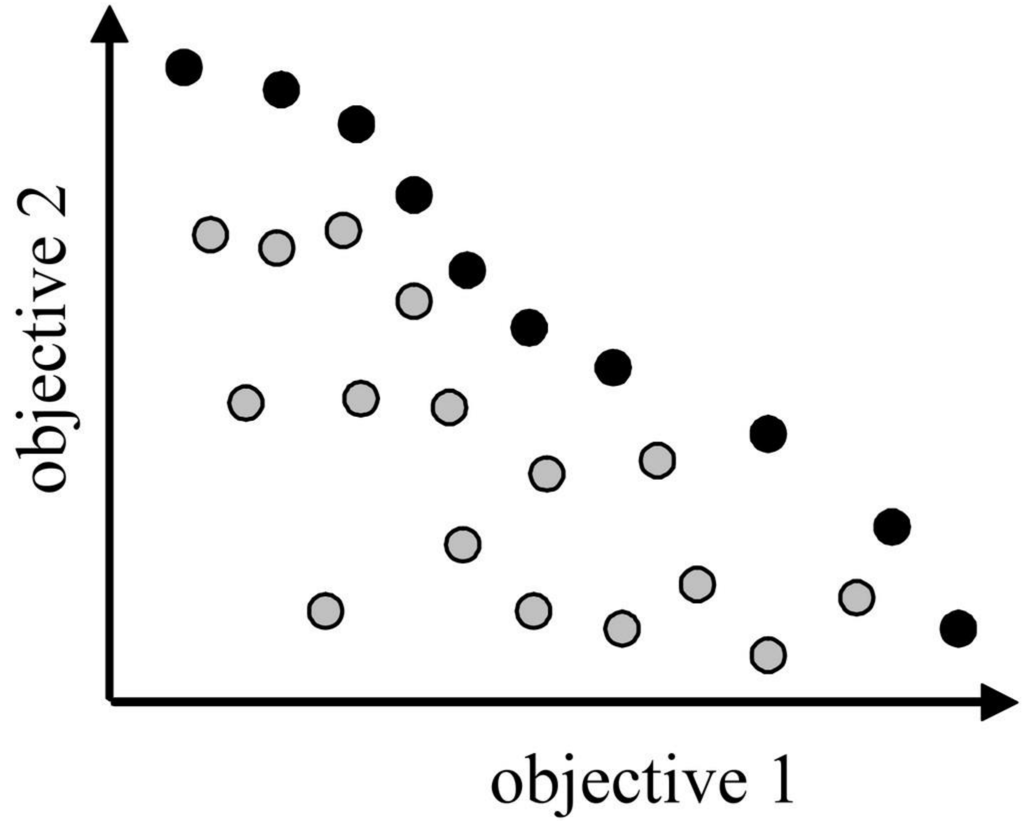
- Pareto dominance illustration, maximising objectives
- Solution A strongly dominates solution C
- Solution B weakly dominates solution C
- A and B are incomparable



Vamplew, P., Dazeley, R., Berry, A., Issabekov, R., & Dekker, E. (2011). Empirical evaluation methods for multiobjective reinforcement learning algorithms. Machine learning, 84, 51-80.

Solution Sets – Pareto Front

- Black points indicate solutions which form the Pareto front
- Grey solutions are dominated by at least one member of the Pareto front



Vamplew, P., Dazeley, R., Berry, A., Issabekov, R., & Dekker, E. (2011). Empirical evaluation methods for multiobjective reinforcement learning algorithms. Machine learning, 84, 51-80.

Solution Sets – Convex Hull

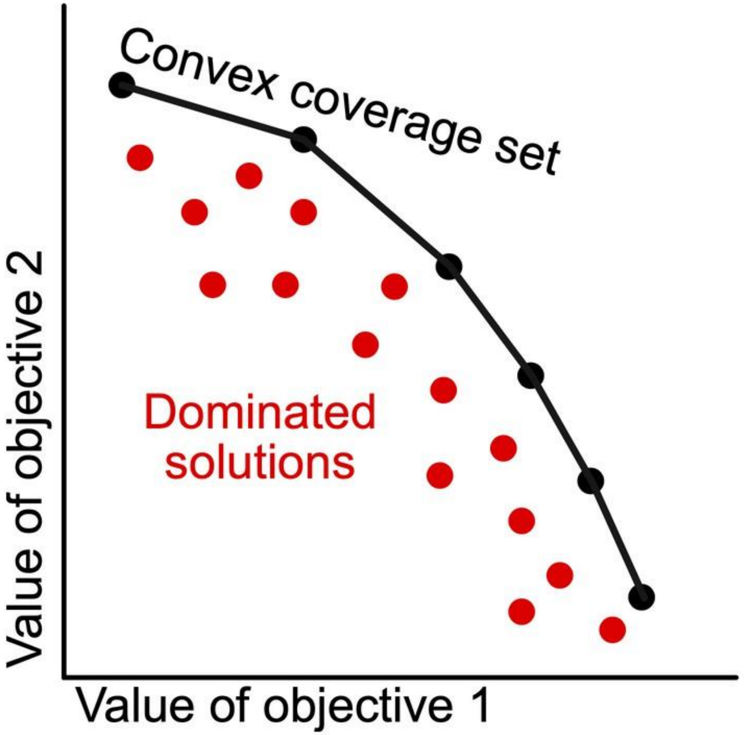
- The convex hull is the undominated set for non-decreasing linear utility functions

- The ***convex hull*** *(CH)* is the subset of Π for which there exists a **w** (for a linear *u*) for which the linearly scalarised value is maximal:

$$CH(\Pi) = \{\pi \in \Pi \mid \exists \mathbf{w}, \forall \pi' \in \Pi : \mathbf{w}^\top \mathbf{V}^\pi \geq \mathbf{w}^\top \mathbf{V}^{\pi'}\}$$

Solution Sets – Convex Coverage Set

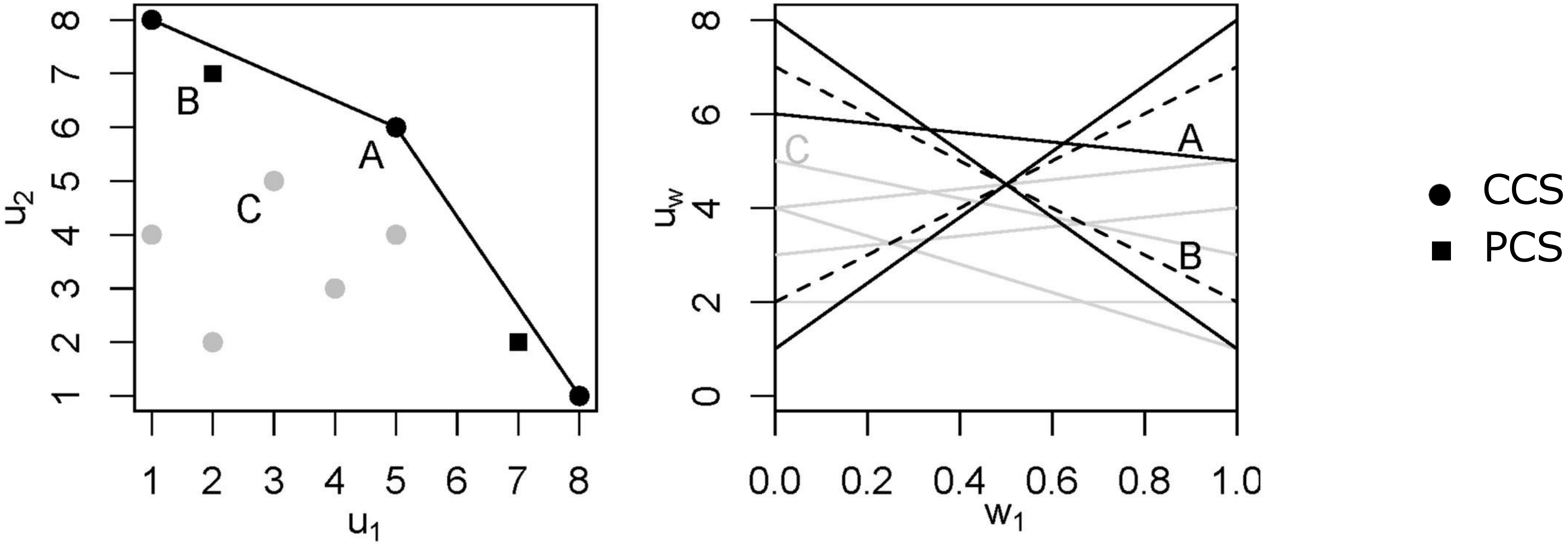
- A set $CCS(\Pi)$ is a ***convex coverage set*** if it is a subset of $CH(\Pi)$ and if for every $\mathbf{w}$ it contains a policy whose linearly scalarised value is maximal:



$$CCS(\Pi) \subseteq CH(\Pi) \wedge \left(\forall \mathbf{w}, \exists \pi \in CCS(\Pi), \forall \pi' \in \Pi : \mathbf{w}^\top \mathbf{V}^{\pi} \geq \mathbf{w}^\top \mathbf{V}^{\pi'}\right)$$

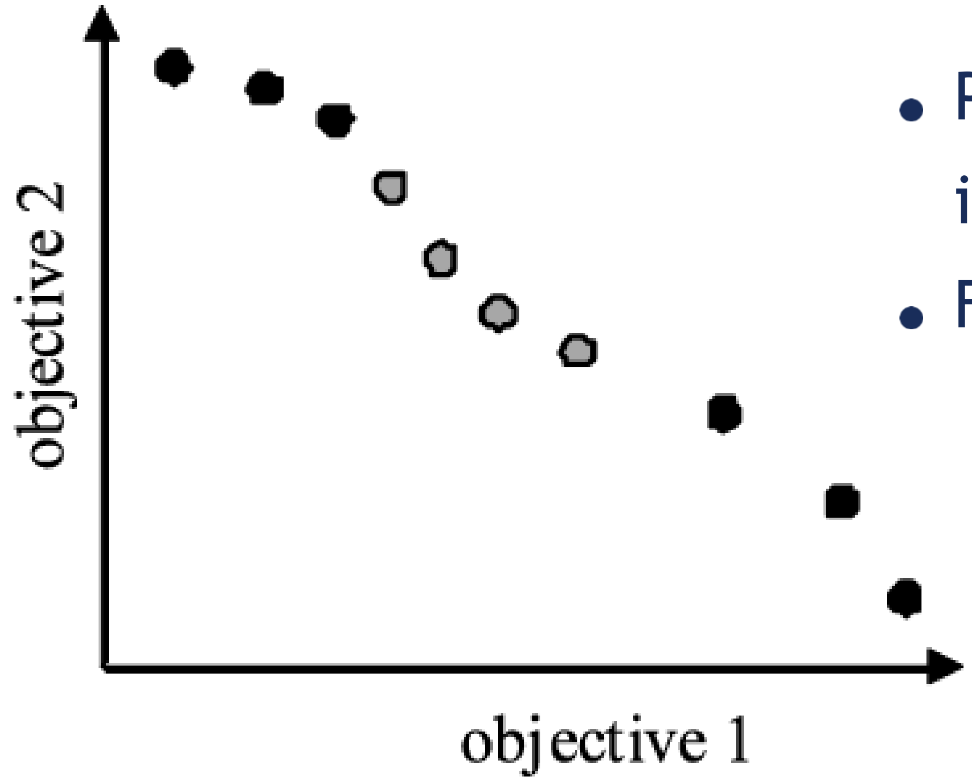
van Doorna, J., Odijkac, D., Roijersab, D. M., & de Rijkea, M. Multi-Objective Optimization for Information Retrieval.

Solutions Sets – CCS vs PCS



Roijers, D. M., Whiteson, S., & Oliehoek, F. A. (2015). Computing convex coverage sets for faster multi-objective coordination. Journal of Artificial Intelligence Research, 52, 399-443.

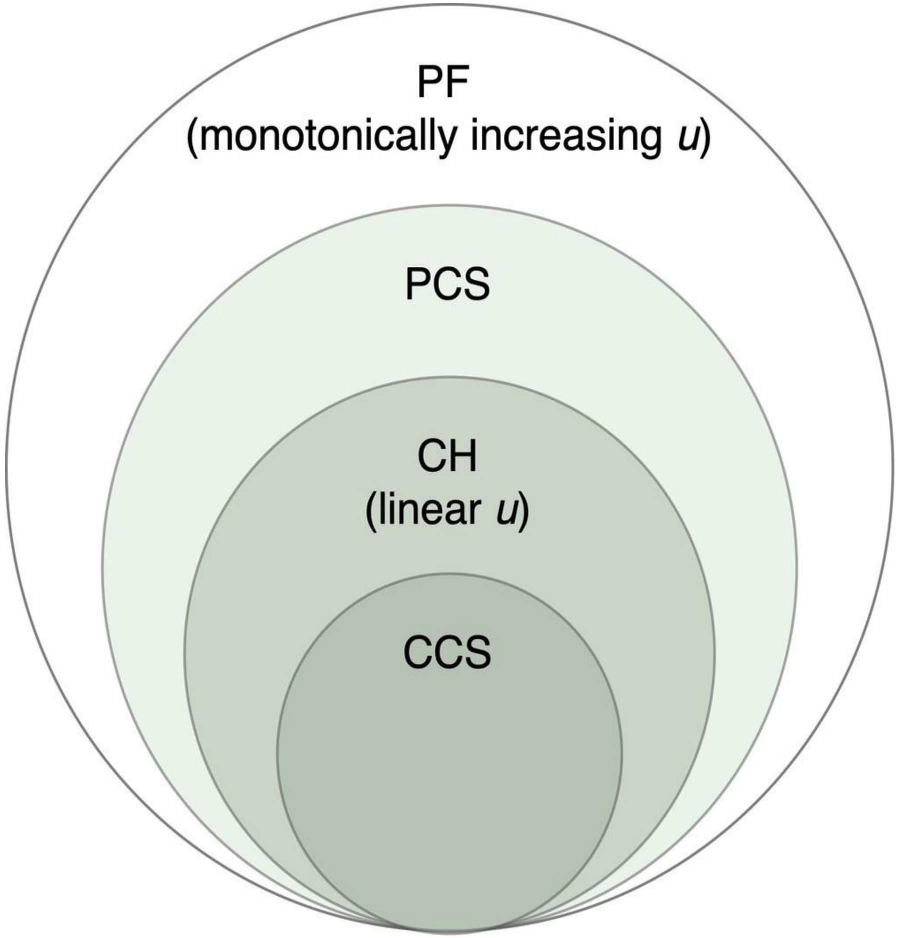
Solution Sets – Limitation of Linear U



- Pareto front containing a concave region, indicated by the grey points
- Fundamental limitation of linear scalarisation:
  - it cannot find policies which lie in non-convex regions of the Pareto front

Vamplew, P., Dazeley, R., Berry, A., Issabekov, R., & Dekker, E. (2011). Empirical evaluation methods for multiobjective reinforcement learning algorithms. Machine learning, 84, 51-80.

Solution Sets



- The choice of solution set is key to the efficiency of the algorithms used to solve multi- objective problems

TDDC17 AI LE7 HT2024:
Reinforcement learning
Deep reinforcement learning
Multi-objective reinforcement learning

www.ida.liu.se/~TDDC17

LINKÖPING UNIVERSITY