Artificial Intelligence and Mobility Lab

Reinforcement Learning Algorithms and Applications

Prof. Joongheon Kim
Korea University, School of Electrical Engineering
Artificial Intelligence and Mobility Laboratory
https://joongheon.github.io
joongheon@korea.ac.kr

Introduction and Preliminaries

Deep Reinforcement Learning Theory and Implementation

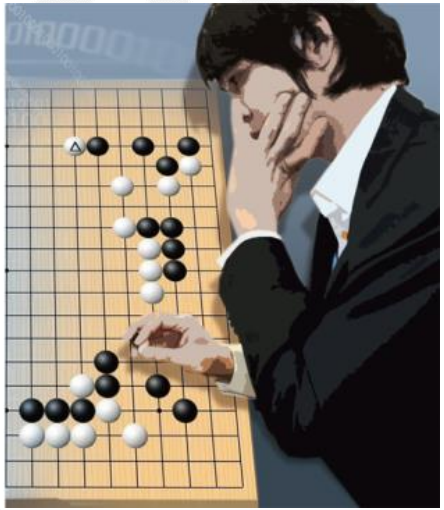
Policy Gradient

Imitation Learning

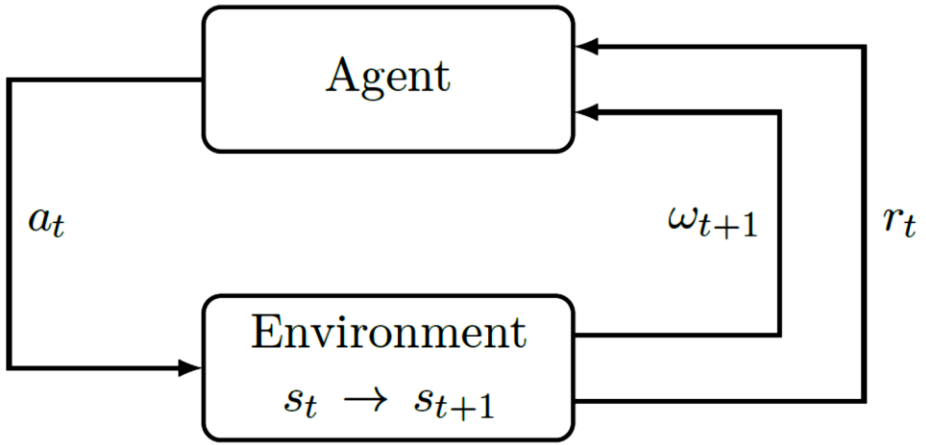
Autonomous Mobility Applications

- **Introduction and Applications**
- Dynamic Programming
- Markov Decision Process

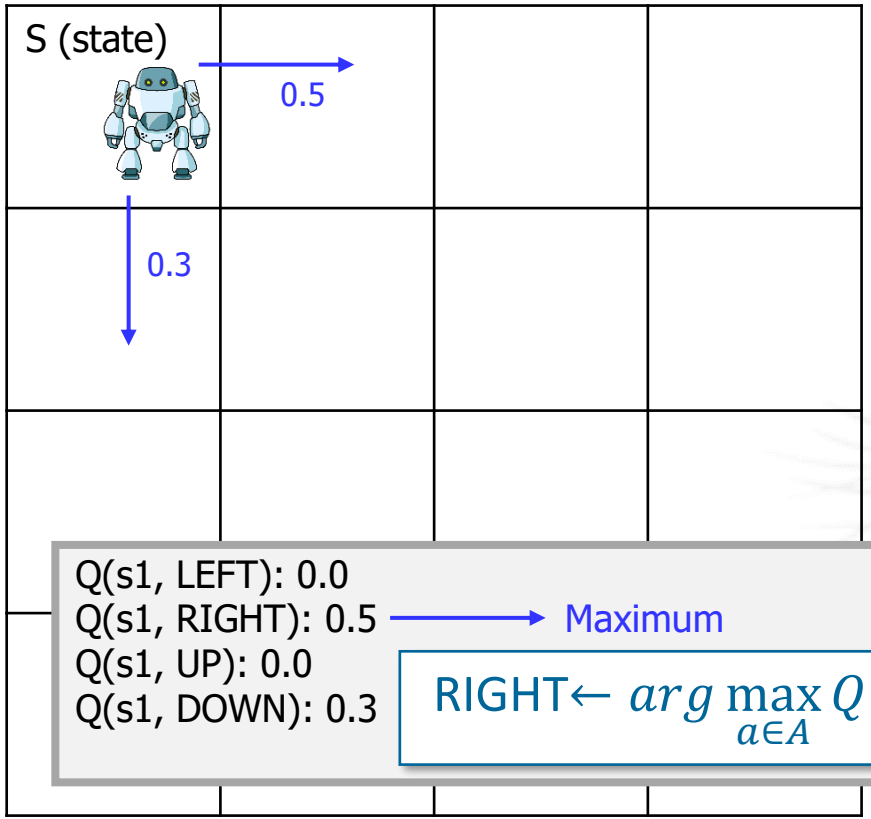
- Action Sequence (also called **Policy**, later in this presentation)!



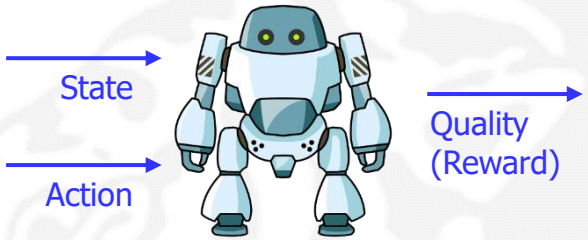
- **RL Setting**
  - The general RL problem is formalized as a **discrete time stochastic control process** where **an agent interacts with its environment** as follows:
    1. The agent starts
       in a given state within its environment $s_0 \in S$
       by gathering an initial observation $\omega_0 \in \Omega$.
    2. At each time step t,
       The agent has to take an action $a_t \in A$.
       It follows three consequences:
       1) Obtains a reward $r_t \in R$
       2) State transitions to $s_{t+1} \in S$
       3) Obtains an observation $\omega_{t+1} \in \Omega$



Q-Learning



- Q-Function (State-action value)

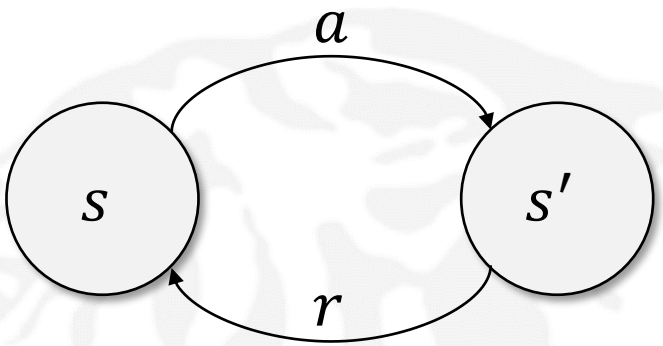


Q (state, action)

Optimal Policy π and Max Q

- Max Q = $\max_{a'} Q(s, a')$
- $\pi^*(s) = arg\max_{a} Q(s, a)$

- My condition
  - I am now in state s
  - When I do action a, I will go to s'.
  - When I do action a, I will get reward r
  - Q in s', it means $Q(s', a')$ exists.



- How can we express $Q(s, a)$ using $Q(s', a')$?

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

Recurrence (e.g., factorial)

```
F(x){
        if (x != 1){ x * F(x-1) }
        if (x == 1){ F(x) = 1 }
        }
}
```

3! = F(3) = 3 * F(2)
= 3 * 2 * F(1)
= 3 * 2* 1 = 6

Introduction and Preliminaries

Deep Reinforcement Learning Theory and Implementation

Policy Gradient

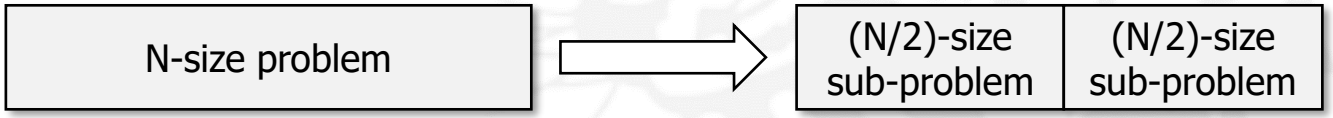
Imitation Learning

Autonomous Mobility Applications

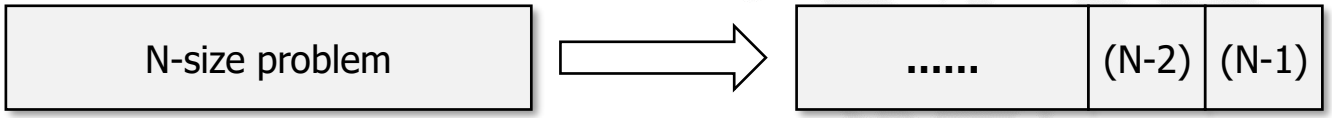
- Introduction and Applications
- **Dynamic Programming**
- Markov Decision Process

- **Introduction**
- Applications
  - Fibonacci Number
  - Pascal's Triangle
  - Knapsack Problem

- Dynamic Programming
  - The term “programming” stands for “planning”.
  - Usually used for optimization problems
  - In order to solve large-scale problems, (i) divide the problems into several sub-problems, (ii) solve the sub-problems, and (iii) obtain the solution of the original problem based on the solutions of the sub-problems, recursively
  - Difference from divide-and-conquer
    - Divide-and-conquer

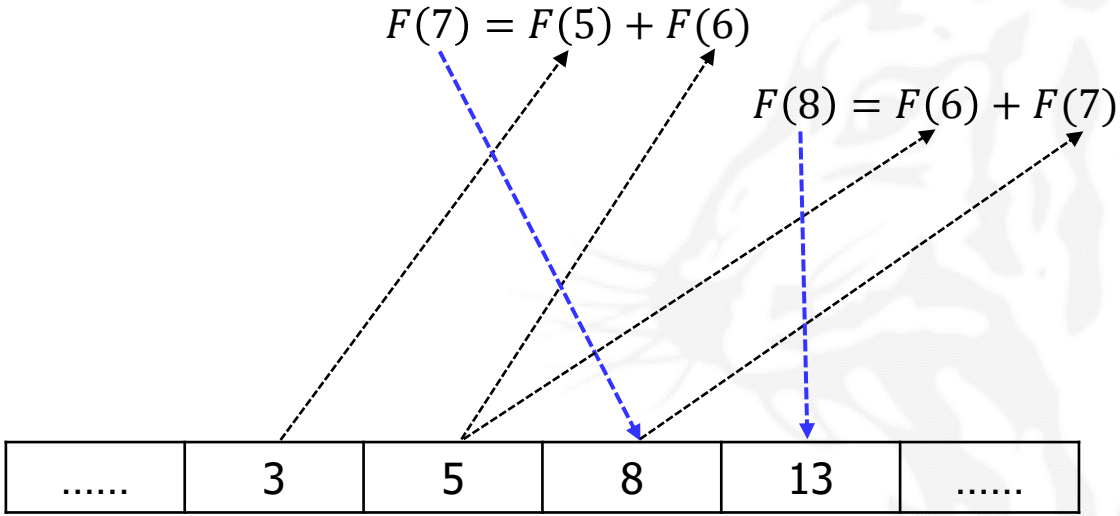


    - Dynamic programming



- Introduction
- Applications
  - **Fibonacci Number**
  - Pascal's Triangle
  - Knapsack Problem

- Fibonacci Number
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ……
  - $F(N) = F(N-2) + F(N-1)$ where $F(1) = 0$ and $F(2) = 1$ (Recursive!)

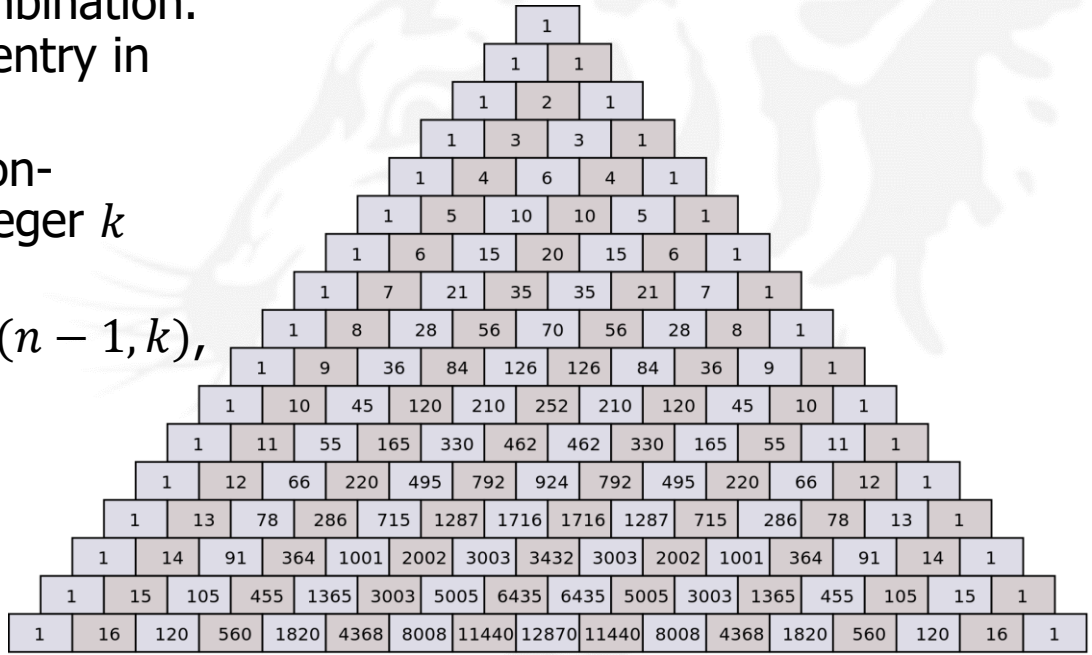


- Introduction
- Applications
  - Fibonacci Number
  - **Pascal's Triangle**
  - Knapsack Problem

- Pascal's Triangle
  - The entry in the n-th row and k-th column of Pascal's triangle is denoted $C(n,k)$ where C stands for combination. Note that the unique nonzero entry in the topmost row is $C(0,0) = 1$.
  - General Formulation for any non-negative integer n and any integer k between 0 and n:

    $$C(n,k) = C(n-1,k-1) + C(n-1,k),$$

  - Recursive!



Introduction and Preliminaries

- Introduction and Applications
- Dynamic Programming
- **Markov Decision Process**

Deep Reinforcement Learning Theory and Implementation

Policy Gradient

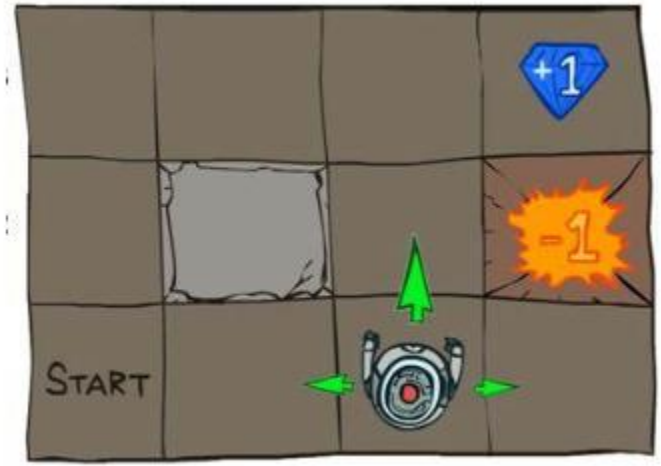
Imitation Learning

Autonomous Mobility Applications

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - S: Set of states
  - A: Set of actions
  - R: Reward function
  - T: Transition function
  - γ: Discount factor

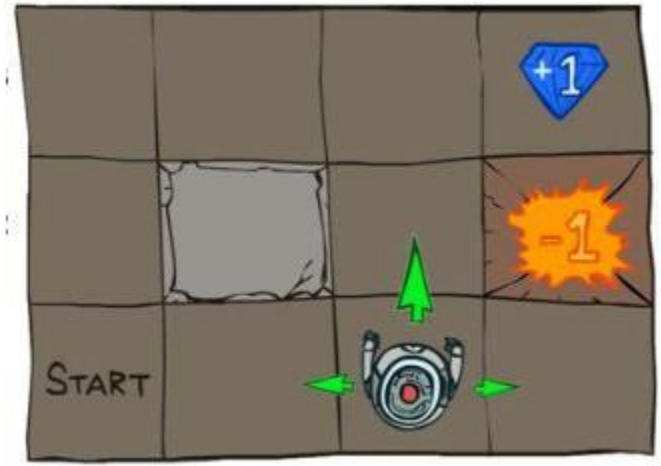
How can we use MDP to model agent in a maze?

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - **S: Set of states**
  - A: Set of actions
  - R: Reward function
  - T: Transition function
  - γ: Discount factor



S: location (x, y) if the maze is a 2D grid
- s_0: starting state
- s: current state
- s': next state
- s_t: state at time t

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - S: Set of states
  - **A: Set of actions**
  - R: Reward function
  - T: Transition function
  - γ: Discount factor

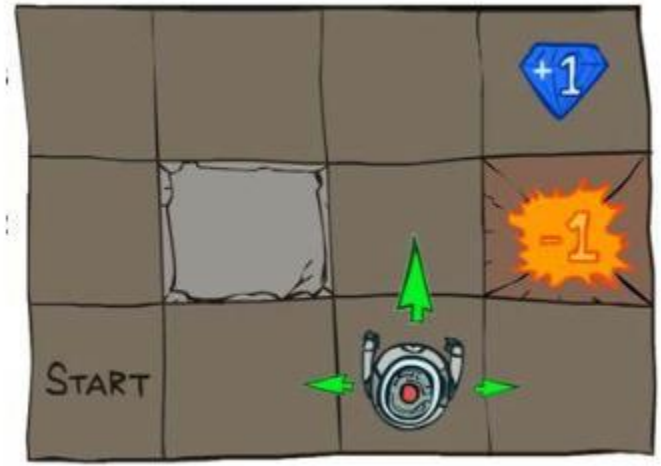


S: location (x, y) if the maze is a 2D grid

A: move up, down, left, or right

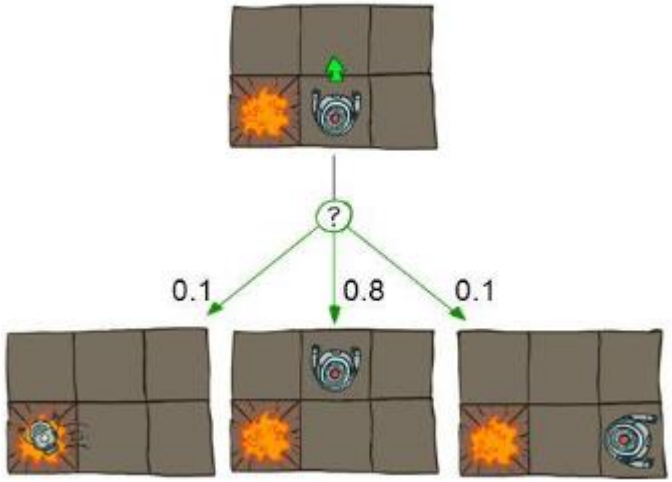
- $s \rightarrow s'$

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - S: Set of states
  - A: Set of actions
  - **R: Reward function**
  - T: Transition function
  - γ: Discount factor



S: location (x, y) if the maze is a 2D grid
A: move up, down, left, or right
R: how good was the chosen action?
- $r = R(s, a, s')$
- -1 for moving (battery used)
- +1 for jewel? +100 for exit?

- Markov Decision Process (MDP) Components: $<S, A, R, T, \gamma>$
  - S: Set of states
  - A: Set of actions
  - R: Reward function
  - **T: Transition function**
  - γ: Discount factor



Stochastic Transition

S: location (x, y) if the maze is a 2D grid
A: move up, down, left, or right
R: how good was the chosen action?
T: where is the robot's new location?
- $T = P(s'|s, a)$

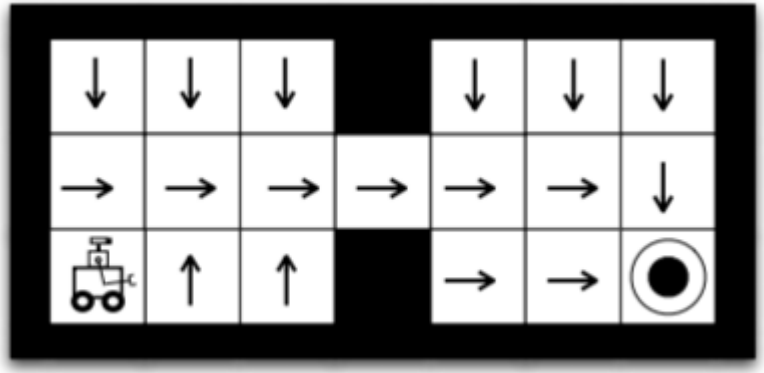
- Markov Decision Process (MDP) Components: <S, A, R, T, γ>
  - S: Set of states
  - A: Set of actions
  - R: Reward function
  - T: Transition function
  - **γ: Discount factor**

S: location (x, y) if the maze is a 2D grid
A: move up, down, left, or right
R: how good was the chosen action?
T: where is the robot's new location?
γ: how much does future reward worth?

- $0 \leq \gamma \leq 1$, [$\gamma \approx 0$: future reward is near 0 (immediate action is preferred)]

- Policy
  - $\pi: S \rightarrow A$
  - Maps states to actions
  - Gives an action for every state
- Return
  -

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$



Our goal:
Find π that maximizes expected return!

- Action Value Function (Q)

$$Q^{\pi}(s,a) = E_{\pi}(R_t|s_t = s, a_t = a) = E_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,\middle|\, s_t = s, a_t = a\right)$$

  - Expected return of starting at state s, taking action a, and then following policy π
  - How much return do I expect starting from state s and taking action a?

- Our goal is to find the optimal policy

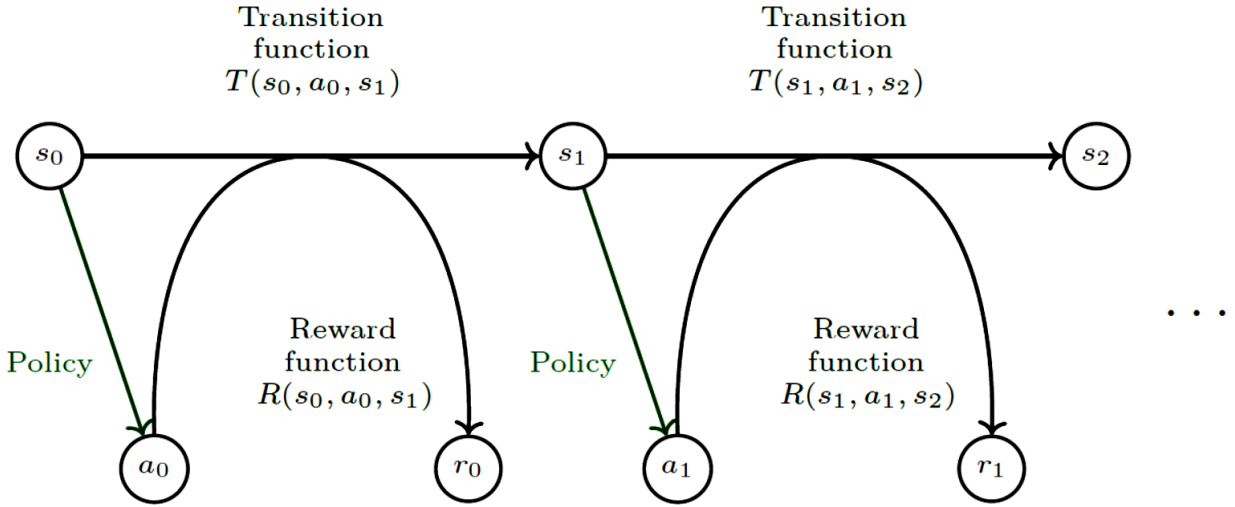
$$\pi^*(s) = \max_{\pi} R^{\pi}(s)$$

  - If $T(s'|s,a)$ and $R(s,a,s')$ are known, this is a **planning** problem.
  - We can use **dynamic programming** to find the optimal policy.

- **Markov Property**
  - [Definition (Markovian)] A discrete time stochastic control process is Markovian (i.e., it has the Markov property) if
    - $P(\omega_{t+1}|\omega_t, a_t) = P(\omega_{t+1}|\omega_t, a_t, \cdots, \omega_0, a_0)$, and
    - $P(r_t|\omega_t, a_t) = P(r_t|\omega_t, a_t, \cdots, \omega_0, a_0)$
  - The Markov property means that the future of the process only depends on the current observation, and the agent has no interest in looking at the full history.

- **Markov Property**
  - [Definition (MDP)] A Markov Decision Process (MDP) is a discrete time stochastic control process defined as follows. An MDP is a 5-tuple (S, A, T, R, γ) where:
    - S is the state space,
    - A is the action space,
    - $T: S \times A \times S \to [0,1]$ is the transition function (set of conditional transition probabilities between states),
    - $R: S \times A \times S \to R$ is the reward function, where R is a continuous set of possible rewards in a range $R_{\max} \in R^{+}$ (e.g., $[0, R_{\max}]$),
    - $\gamma \in [0,1)$ is the discount factor.

- **Markov Property**
  - The system in [Definition (MDP)] is fully observable in an MDP, which means that the observation is the same as the state of the environment: $\omega_t = s_t$.
  - At each time step t,
    - The probability of moving to s_{t+1} is given by the state transition function $T(s_t, a_t, s_{t+1})$ and the reward is given by a bounded reward function $R(s_t, a_t, s_{t+1}) \in R$.



Introduction and Preliminaries

Deep Reinforcement Learning Theory and Implementation

- **Deep Neural Network Summary**
- Deep Q-Network (DQN)
- Q-Learning Implementation

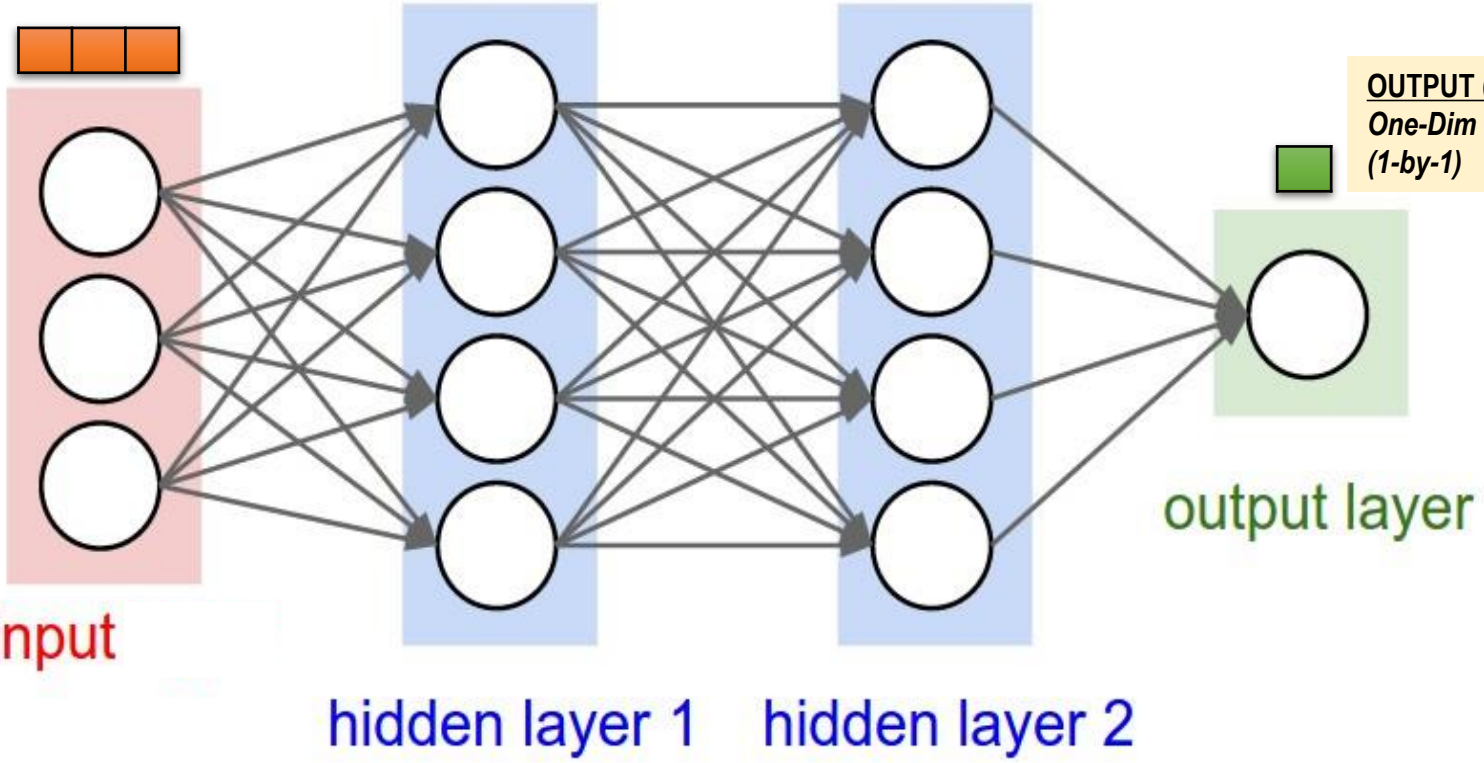
Policy Gradient

Imitation Learning

Autonomous Mobility Applications

- Toy Model

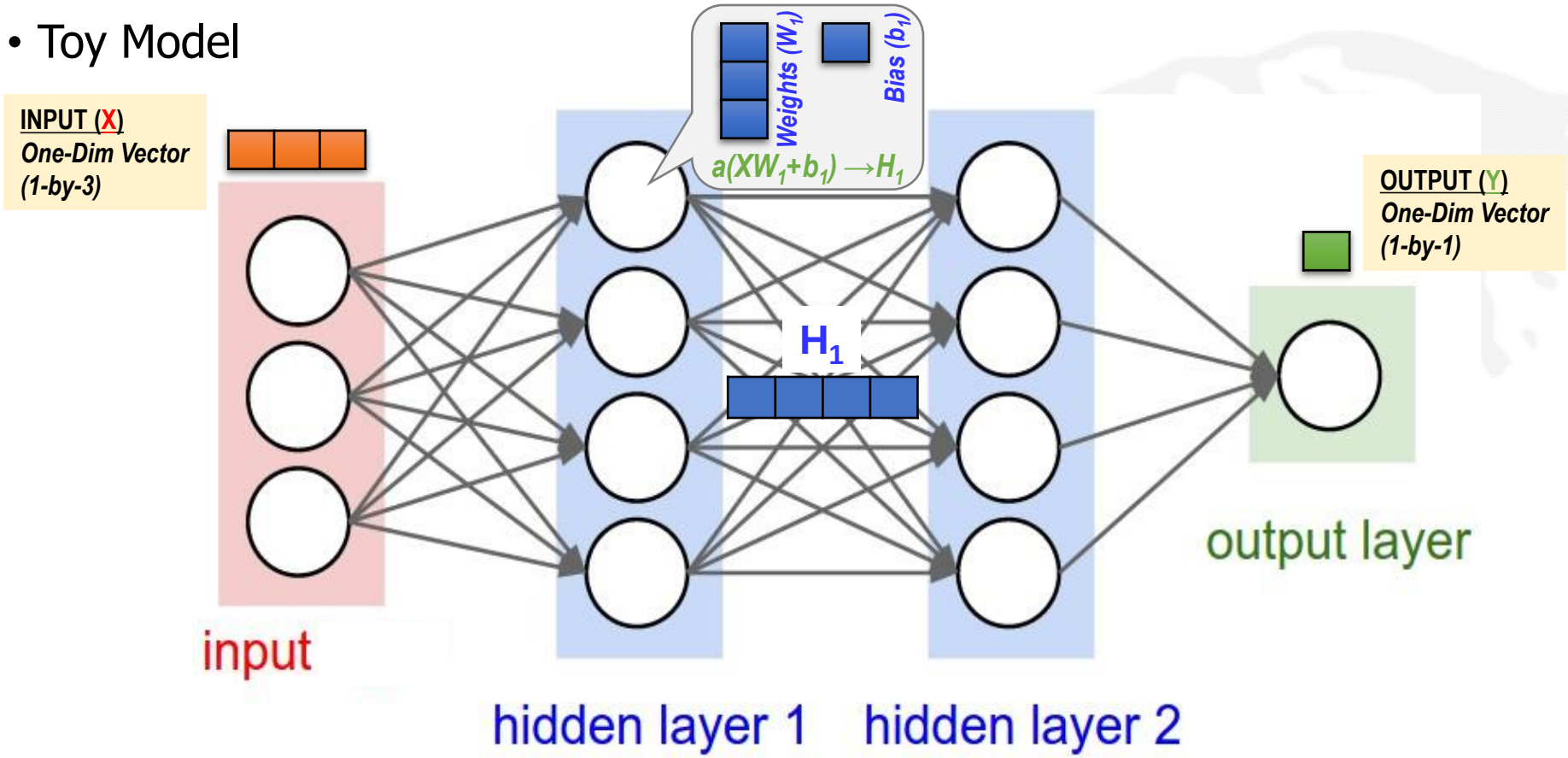
INPUT (X)
One-Dim Vector (1-by-3)



OUTPUT (Y)
One-Dim Vector (1-by-1)

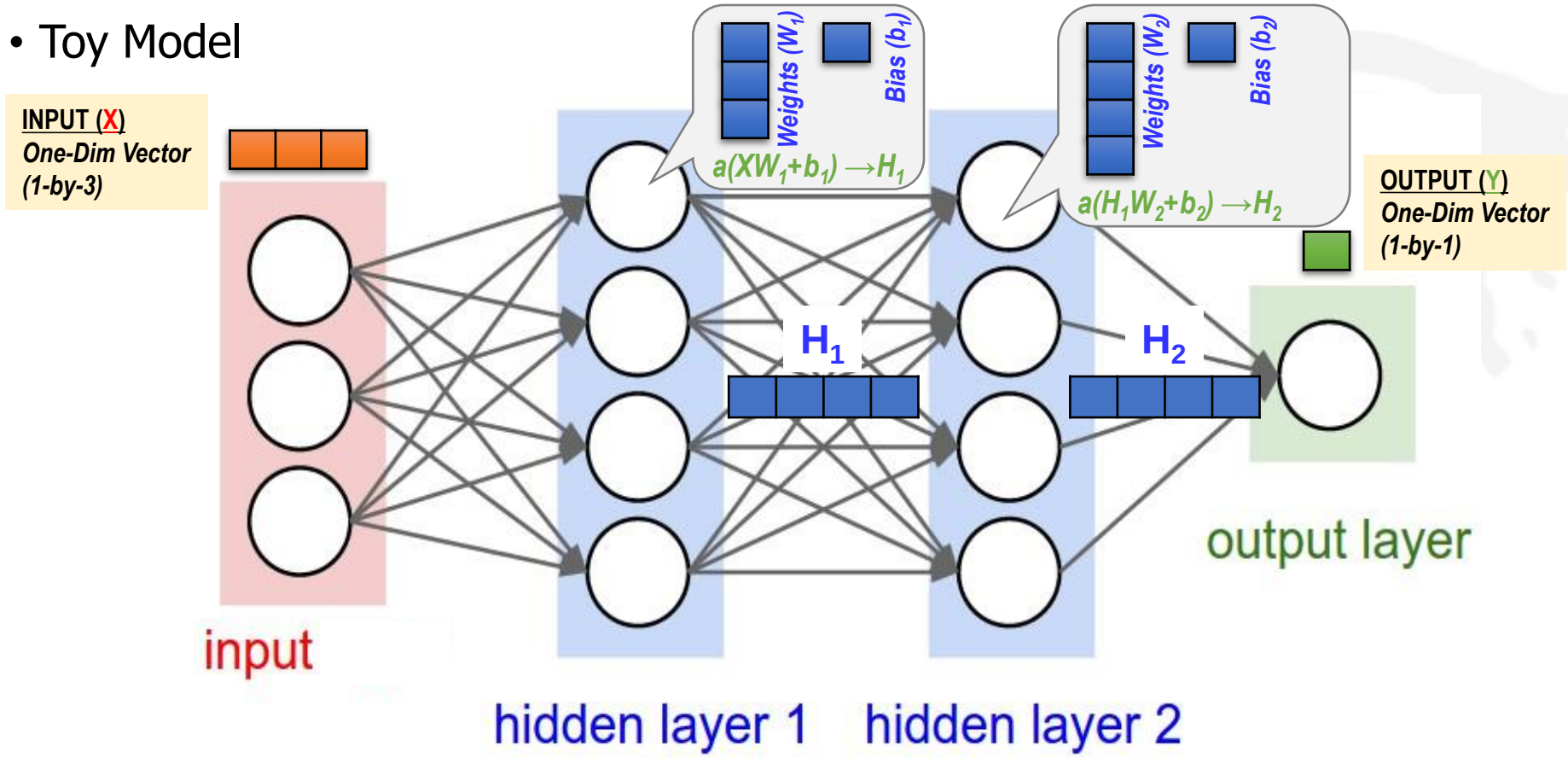
Conventional Deep Neural Network Training and Inference

- Toy Model



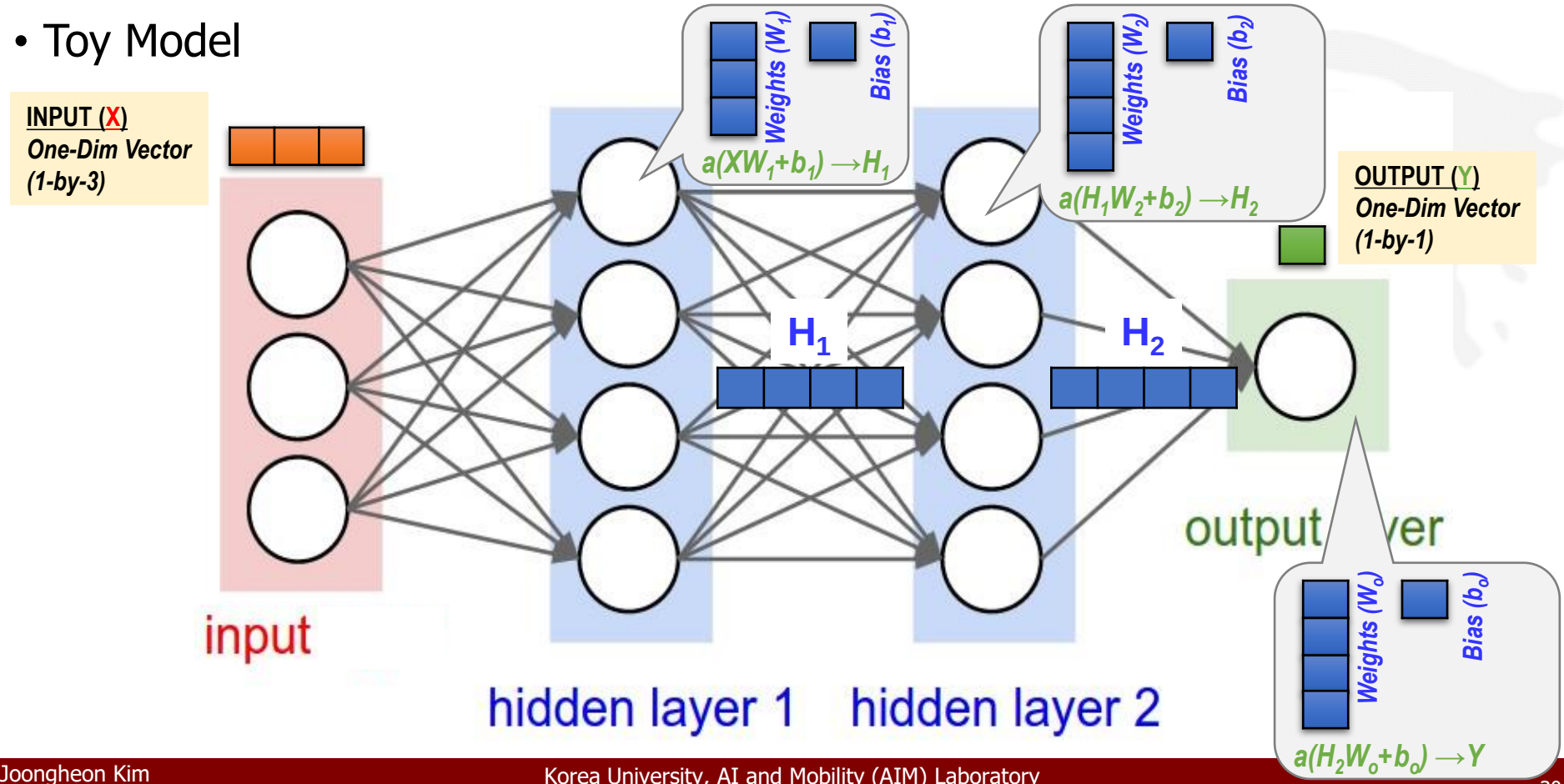
Conventional Deep Neural Network Training and Inference

- Toy Model



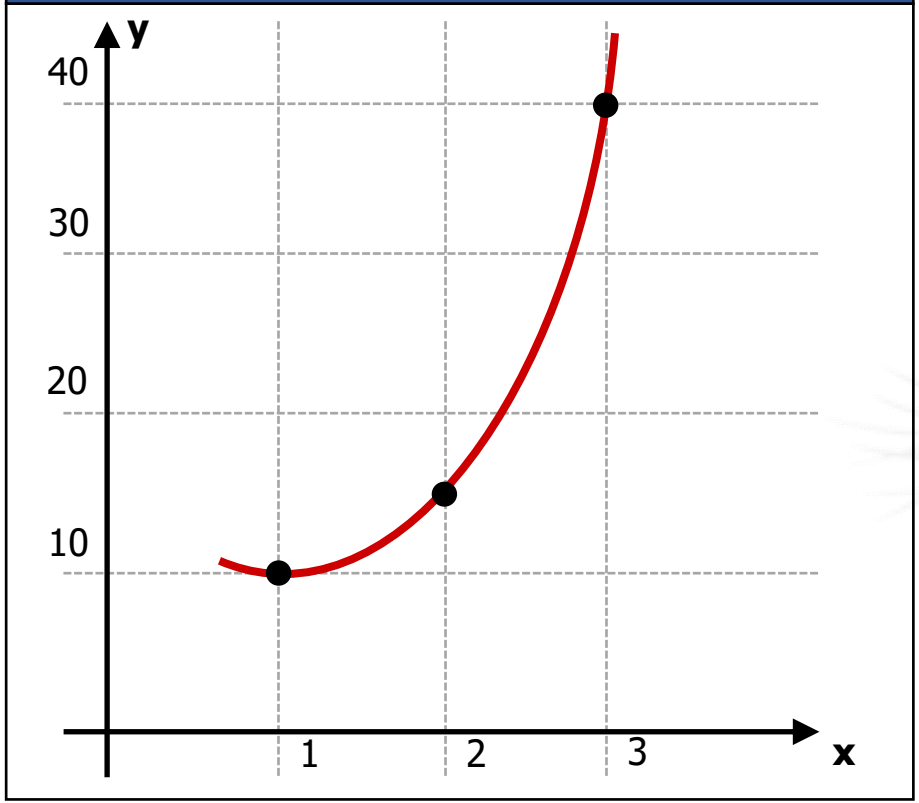
Conventional Deep Neural Network Training and Inference

- Toy Model

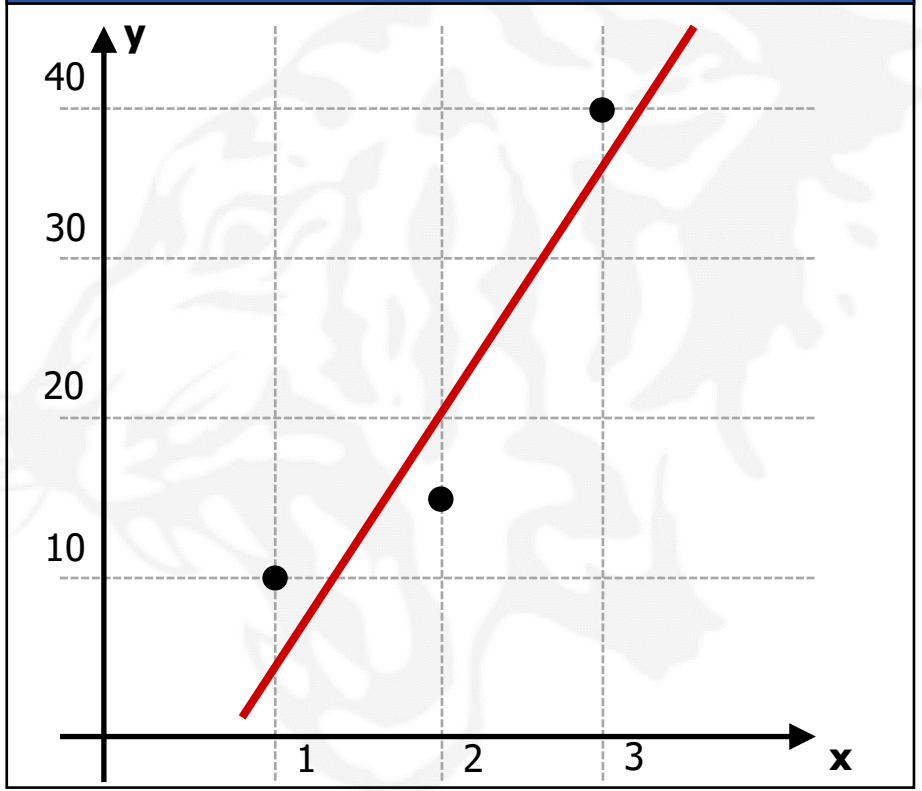


Interpolation vs. Linear Regression

Interpolation

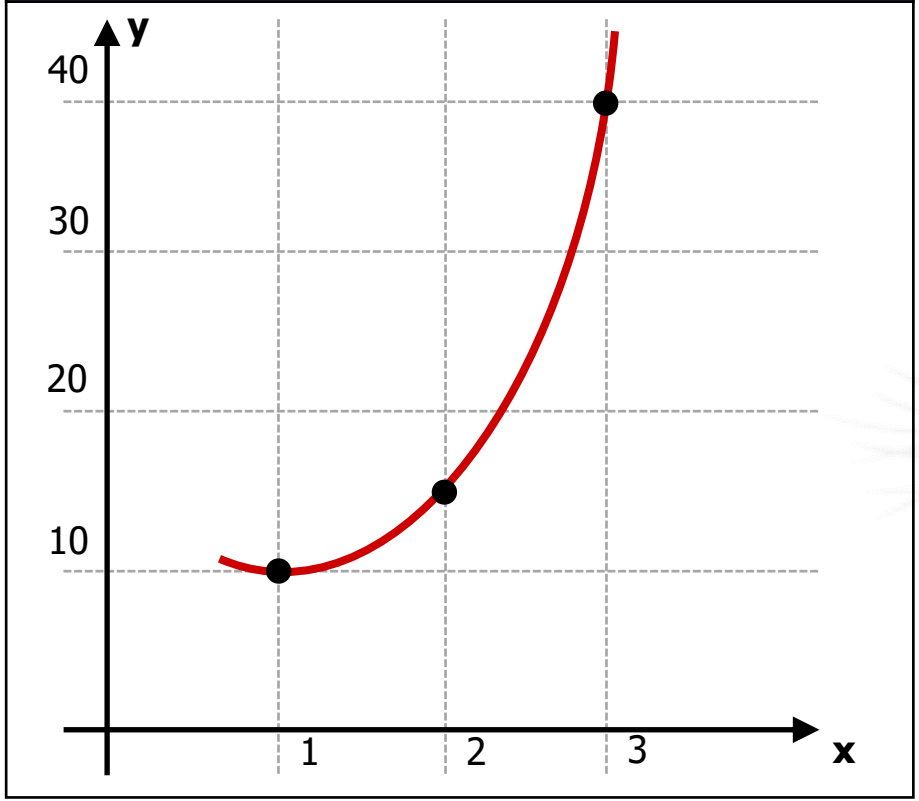


Linear Regression



Interpolation vs. Linear Regression

Interpolation

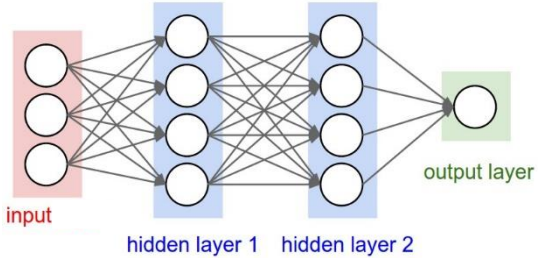


Interpolation with Polynomials

$$y = a_2 x^2 + a_1 x^1 + a_0$$

where three points are given.
→ Unique coefficients (a_0, a_1, a_2) can be calculated.

Is this related to
Neural Network Training?



$$Y = a(a(a(X \cdot W_1 + b_1) \cdot W_2 + b_2) \cdot W_o + b_o)$$

where training data/labels (X: data, Y: labels) are given.

→ Find $W_1, b_1, W_2, b_2, W_o, b_o$

→ This is the mathematical meaning of neural network training.

→ **Function Approximation**

→ The most well-known function approximation with neural network: **Deep Reinforcement Learning**

Introduction and Preliminaries

Deep Reinforcement Learning Theory and Implementation

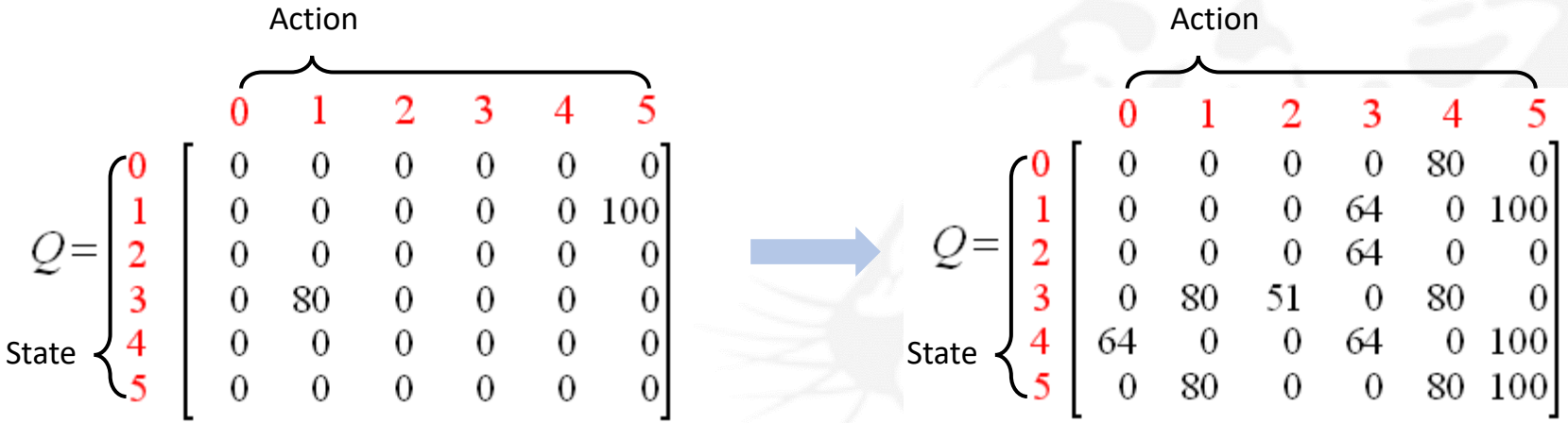
- Deep Neural Network Summary
- **Deep Q-Network (DQN)**
- Q-Learning Implementation

Policy Gradient

Imitation Learning

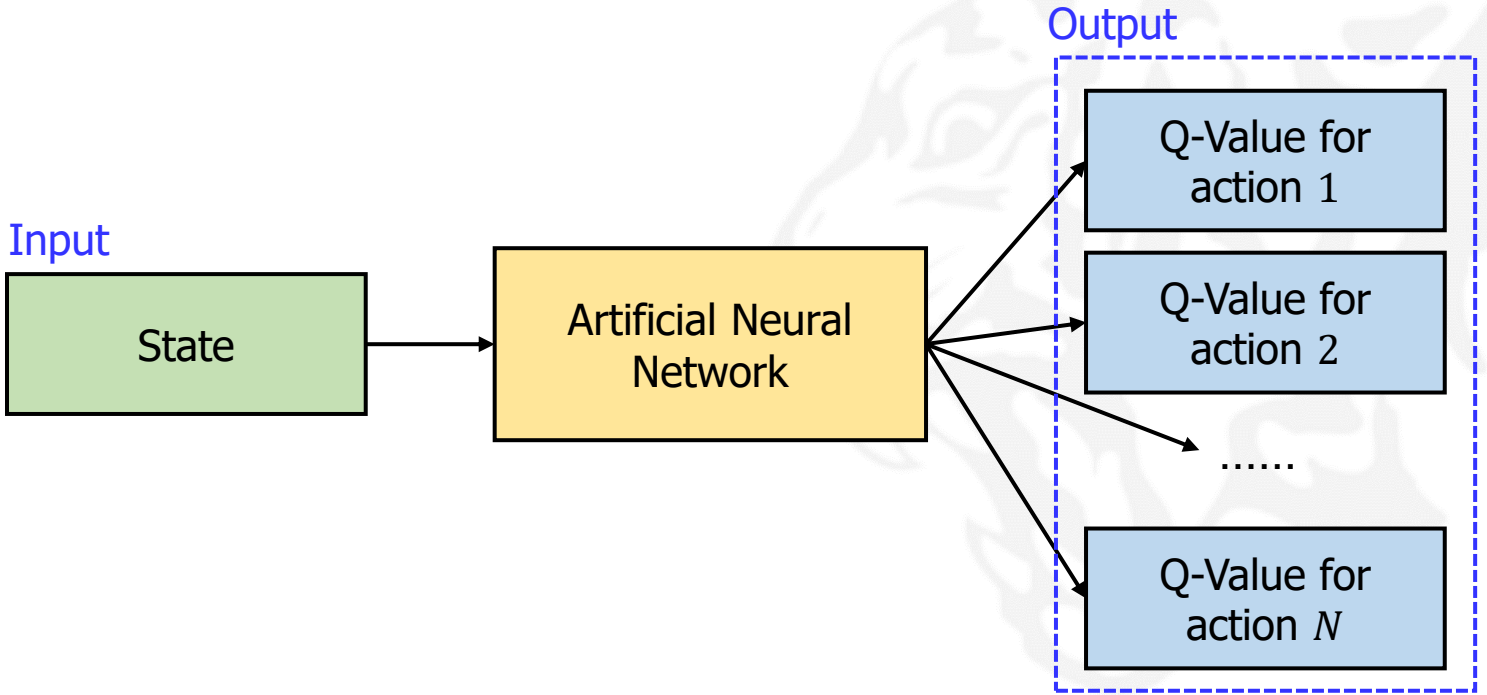
Autonomous Mobility Applications

- Small-Scale Q-Values



Q-table update example

- Large-Scale Q-Values
  - It is inefficient to make the Q-table for each state-action pair.
    → ANN is used to **approximate the Q-function**.



Introduction and Preliminaries

Deep Reinforcement Learning Theory and Implementation

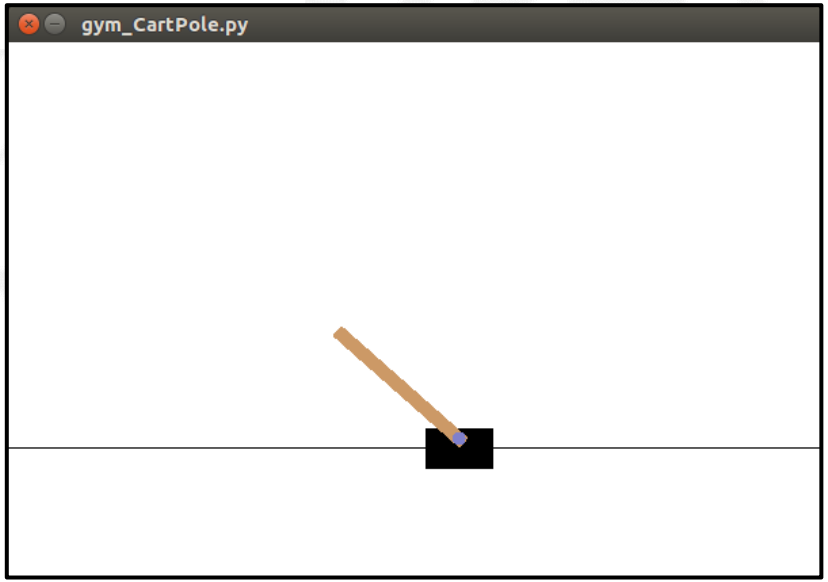
- Deep Neural Network Summary
- Deep Q-Network (DQN)
- **Q-Learning Implementation**

Policy Gradient

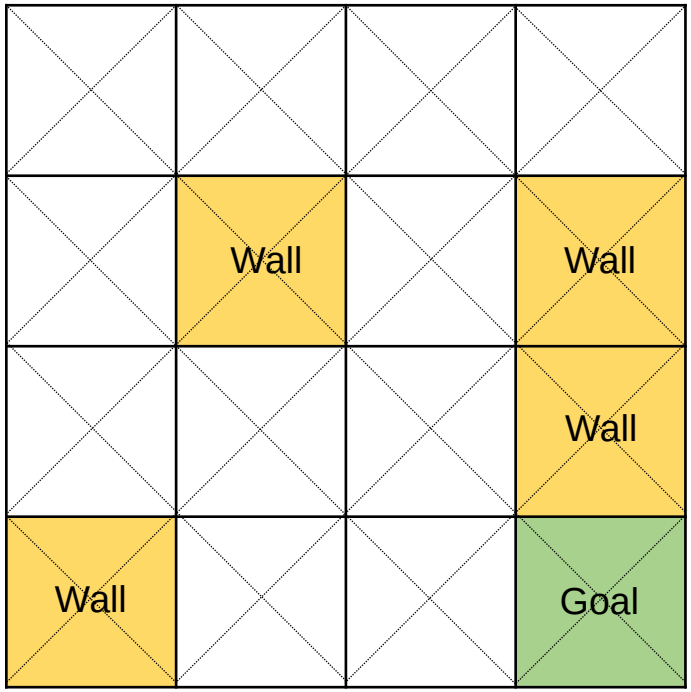
Imitation Learning

Autonomous Mobility Applications

```python
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    #env.step(action)
```



Q-Learning (Basics)




```python
import numpy as np
import matplotlib.pyplot as plt
import gym
from gym.envs.registration import register
import random

'''
Q-Table
        action |  L  |  D  |  R  |  U |
-----------------------------------------
state: 0       |     |     |     |    |
-----------------------------------------
state: 1       |     |     |     |    |
-----------------------------------------
state: 2       |     |     |     |    |
-----------------------------------------
state: ...     |     |     |     |    |
-----------------------------------------
'''

register(
    id='FrozenLake-v3',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={
        'map_name': '4x4',
        'is_slippery': False
    }
)

env = gym.make("FrozenLake-v3")
```

- Environment setting

Q-Learning (Basics)

```python
# Initialization with 0 in Q-table
Q = np.zeros([env.observation_space.n, env.action_space.n]) # (16,4) where 16: 4*4 map, 4: actions
num_episodes = 1000 # Number of iterations

rList = []
successRate = []

def rargmax(vector):
    m = np.amax(vector) # Return the maximum of an array or maximum along an axis (0 or 1)
    indices = np.nonzero(vector == m)[0] # np.nonzero(True/False vector) => find the maximum
    return random.choice(indices) # Random selection

for i in range(num_episodes): # Updates with num_episodes iterations
    state = env.reset() # Reset
    total_reward = 0 # Reward graph (1: success, 0: failure)
    done = None

    while not done: # The agent is not in the goal yet
        action = rargmax(Q[state, :]) # Find maximum reward among 4 actions, find next action
        new_state, reward, done, _ = env.step(action) # Result of the chosen action

        Q[state, action] = reward + np.max(Q[new_state, :]) # Q-update
        total_reward += reward
        state = new_state

    rList.append(total_reward) # Reward appending
    successRate.append(sum(rList)/(i+1)) # Success rate appending
```

Q-Learning (Basics)

```
32  # Initialization with 0 in Q-table
33  Q = np.zeros([env.observation_space.n, env.action_space.n]) # (16,4) where 16: 4*4 map, 4: actions
34  num_episodes = 1000 # Number of iterations
35
36  rList = []
37  successRate = []
38
39  def rargmax(vector):
40      m = np.amax(vector) # Return the maximum of an array or maximum along an axis (0 or 1)
41      indices = np.nonzero(vector == m)[0] # np.nonzero(True/False vector) => find the maximum
42      return random.choice(indices) # Random selection
43
44  for i in range(num_episodes): # Updates with num_episodes iterations
45      state = env.reset() # Reset
46      total_reward = 0 # Reward graph (1: success, 0: failure)
47      done = None
48
49      while not done: # The agent is not in the goal yet
50          action = rargmax(Q[state, :]) # Find maximum reward among 4 actions, find next action
51          new_state, reward, done, _ = env.step(action) # Result of the chosen action
52
53          Q[state, action] = reward + np.max(Q[new_state, :]) # Q-update
54          total_reward += reward
55          state = new_state
56
57      rList.append(total_reward) # Reward appending
58      successRate.append(sum(rList)/(i+1)) # Success rate appending
```

- Randomly pick one when multiple argmax values exist

Q-Learning (Basics)

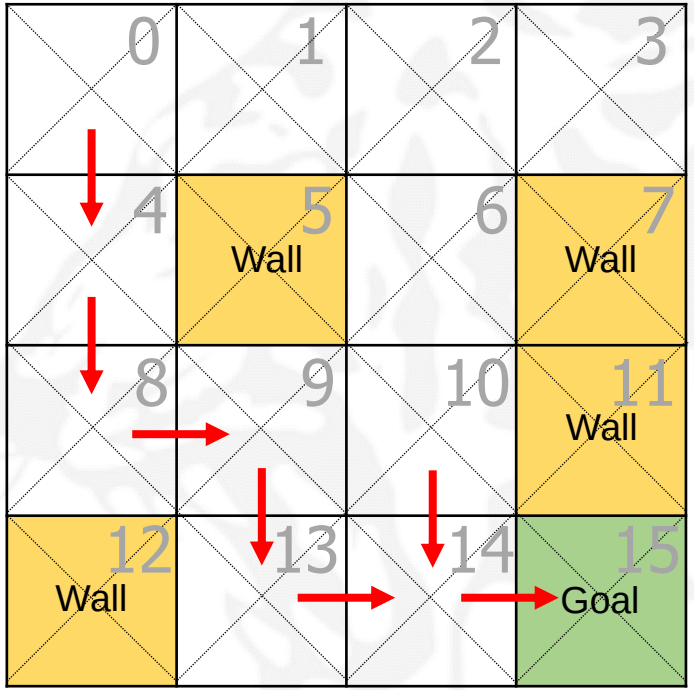

```
32  # Initialization with 0 in Q-table
33  Q = np.zeros([env.observation_space.n, env.action_space.n]) # (16,4) where 16: 4*4 map, 4: actions
34  num_episodes = 1000 # Number of iterations
35
36  rList = []
37  successRate = []
38
39  def rargmax(vector):
40      m = np.amax(vector) # Return the max
41      indices = np.nonzero(vector == m)[0]
42      return random.choice(indices) # Rand
43
44  for i in range(num_episodes): # Updates
45      state = env.reset() # Reset
46      total_reward = 0 # Reward graph (1:
47      done = None
48
49      while not done: # The agent is not in the goal yet
50          action = rargmax(Q[state, :]) # Find maximum reward among 4 actions, find next action
51          new_state, reward, done, _ = env.step(action) # Result of the chosen action
52
53          Q[state, action] = reward + np.max(Q[new_state, :]) # Q-update
54          total_reward += reward
55          state = new_state
56
57      rList.append(total_reward) # Reward appending
58      successRate.append(sum(rList)/(i+1)) # Success rate appending
```

- Iteration until the agent arrives at the goal or it cannot move anymore.
- (line 50) find the action which returns max Q value.
- (line 51) take the action which is the result of (line 50).
  - done: if the agent is at goal or cannot move anymore, done → True
- (line 53) Q-update
- (line 54) reward value accumulation
- (line 55) state value update for next iteration

```
'''
Final Q-Table
[[0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 0.]]
Success Rate :  0.903
'''
```

[L, D, R, U]

0 (D)
1
2
3
4 (D)
5
6
7
8 (R)
9 (D)
10 (D)
11
12
13 (R)
14 (R)
15



Introduction and Preliminaries

Deep Reinforcement Learning Theory and Implementation

Policy Gradient

Imitation Learning

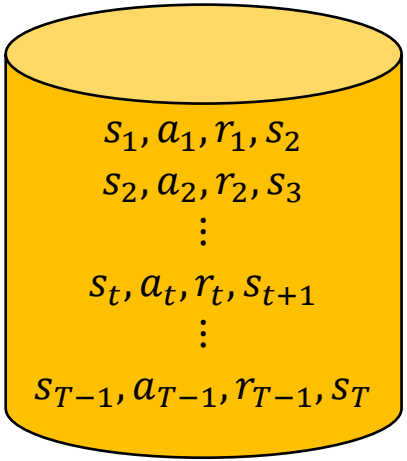
Autonomous Mobility Applications

정책 경사 = 정책 + 경사

1. 정책함수 $\pi_\theta(a|s)$의 좋고/나쁨을 판단하는 기준 $J(\theta)$을 만들고
2. 그 기준 $J(\theta)$에 대한 정책 함수의 경사 (gradient)를 계산해서
3. 경사 상승법을 활용해서 정책을 최적화 한다.

$$\max: J(\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t) \right] = V^\pi(s)$$

Experience Replay



- 목적식 [예시: REINFORCE]
  - $J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t)] = V^\pi(s_0)$

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[\nabla_\theta \ln \pi_\theta(a|s)\, Q^\pi(s,a)]$$
$$= \mathbb{E}_\pi[\nabla_\theta \ln \pi_\theta(A_t|S_t)\, G_t]$$

  - 1-step 단위 Update

$$\theta \leftarrow \theta + \alpha \nabla_\theta \ln \pi_\theta(A_t|S_t) G_t$$

  - 전체 에피소드 단위 update

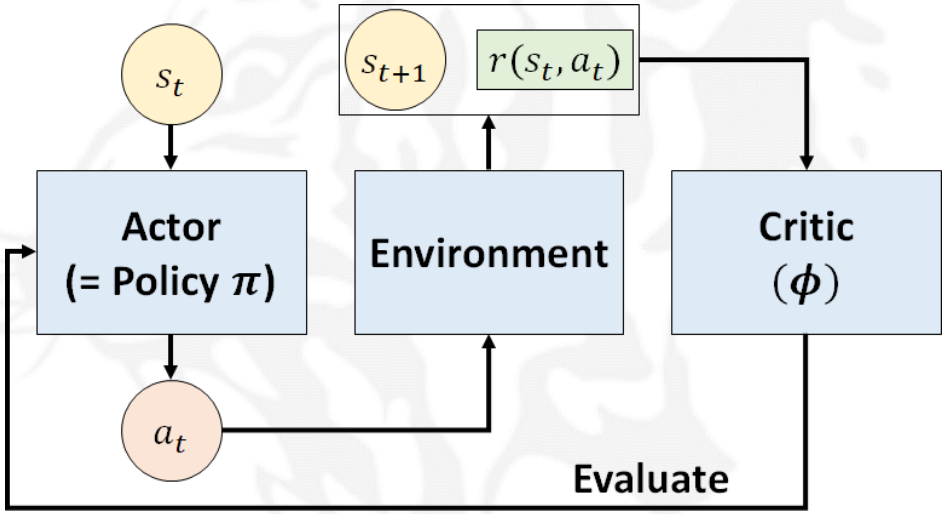
$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \nabla_\theta [\ln(\pi_\theta(A_t|S_t)) G_t]$$

- 구성: Actor 신경망 ($\pi_\theta(a|s)$)과 Critic 신경망 ($Q_\phi^\pi(s,a)$)로 구성

$$G_t \leftarrow Q_\phi^\pi(S_t, A_t)$$

$$\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t|S_t)$$

정책경사> Actor-Critic > 구조



Actor $\pi_\theta(a|s)$ **Critic** $Q^\pi_\phi(s,a)$

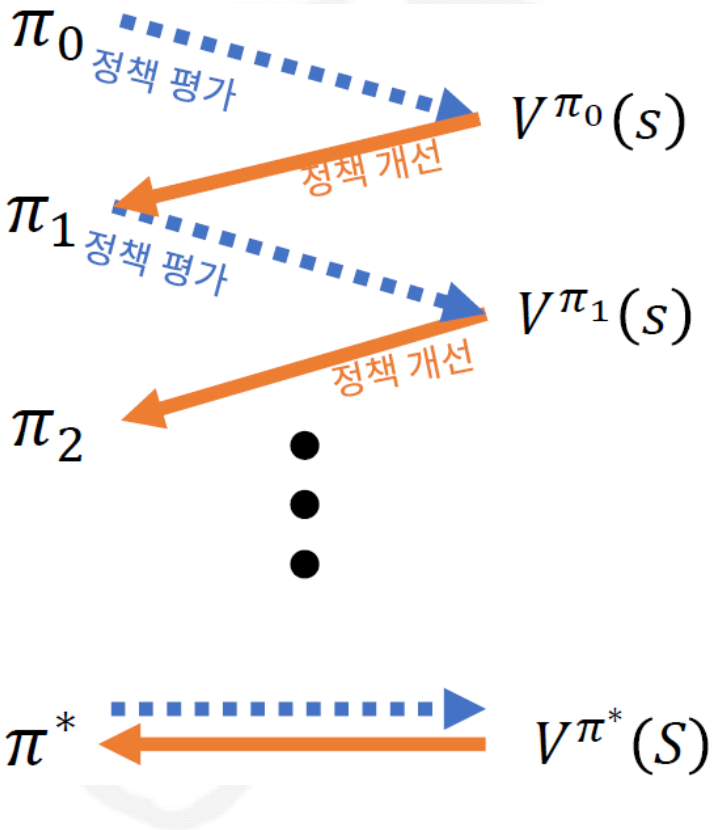
정책 평가:

주어진 정책 π에 대한 가치함수 $V^{\pi}(s)$를 계산

정책 개선:

$\pi_{i+1} \geq \pi_i$ 를 만족시키는 π_{i+1} 을 생성

(정책함수 π 간의 대소관계를 다음과 같이 정의 한다.
$\pi' \geq \pi$ 만약 $V_{\pi'}(s) \geq V_{\pi}(s), \forall\, s \in \mathcal{S}$)



- $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \Rightarrow$ 어드밴티지 함수
- 특정 상황 s에서 행동 a가 다른 행동들에 비해서 얼마나 상대적으로 좋은가?
- 구현 방법

(1) 행동 가치함수 학습: $Q^{\pi}(s,a) \cong Q_{\phi}^{\pi}(s,a)$

(2) 상태 가치함수 학습: $V^{\pi}(s) \cong V_{\psi}^{\pi}(s)$

(3) 어드밴티지 함수: $A^{\pi}(s,a) \cong Q_{\phi}^{\pi}(s,a) - V_{\psi}^{\pi}(s)$

(4) 정책 경사:$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi}[\nabla_{\theta} \ln \pi_{\theta}(a|s) A^{\pi}(s,a)]$

Actor Param: π
Critic Param: ϕ
Target Param: ψ

정책경사> DDPG > 계보

DQN

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou
Daan Wierstra Martin Riedmiller
DeepMind Technologies
{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

Deterministic Policy Gradient

Deterministic Policy Gradient Algorithms

David Silver DAVID@DEEPMIND.COM
DeepMind Technologies, London, UK
Guy Lever GUY.LEVER@UCL.AC.UK
University College London, UK
Nicolas Heess, Thomas Degris, Daan Wierstra, Martin Riedmiller *@DEEPMIND.COM
DeepMind Technologies, London, UK

Abstract

In this paper we consider *deterministic* policy gradient algorithms for reinforcement learning with continuous actions. The deterministic policy gradient has a particularly appealing form: it is the expected gradient of the action-value function. This simple form means that the deterministic policy gradient can be estimated much more efficiently than the usual stochastic policy gradient. To ensure adequate exploration, we introduce an off-policy actor-critic algorithm that learns a deterministic target policy from an exploratory behaviour policy. We demonstrate that deterministic policy gradient algorithms can significantly outperform their stochastic counterparts in high-dimensional action spaces.

case, as policy variance tends to zero, of the stochastic policy gradient.
From a practical viewpoint, there is a crucial difference between the stochastic and deterministic policy gradients. In the stochastic case, the policy gradient integrates over both state and action spaces, whereas in the deterministic case it only integrates over the state space. As a result, computing the stochastic policy gradient may require more samples, especially if the action space has many dimensions.
In order to explore the full state and action space, a stochastic policy is often necessary. To ensure that our deterministic policy gradient algorithms continue to explore satisfactorily, we introduce an off-policy learning algorithm. The basic idea is to choose actions according to a stochastic behaviour policy (to ensure adequate exploration), but to learn about a deterministic target policy (exploiting the efficiency of the deterministic policy gradient). We use the

DDPG

Published as a conference paper at ICLR 2016

CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING

Timothy P. Lillicrap; Jonathan J. Hunt; Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra
Google Deepmind
London, UK
{countzero, jjhunt, apritzel, heess, etom, tassa, davidsilver, wierstra} @ google.com

ABSTRACT

We adapt the ideas underlying the success of Deep Q-Learning to the continuous action domain. We present an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces. Using the same learning algorithm, network architecture and hyper-parameters, our algorithm robustly solves more than 20 simulated physics tasks, including classic problems such as cartpole swing-up, dexterous manipulation, legged locomotion and car driving. Our algorithm is able to find policies whose performance is competitive with those found by a planning algorithm with full access to the dynamics of the domain and its derivatives. We further demonstrate that for many of the tasks the algorithm can learn policies "end-to-end": directly from raw pixel inputs.

Dec 19, 2013

Q-Learning + Deep Policy

ICML 2014

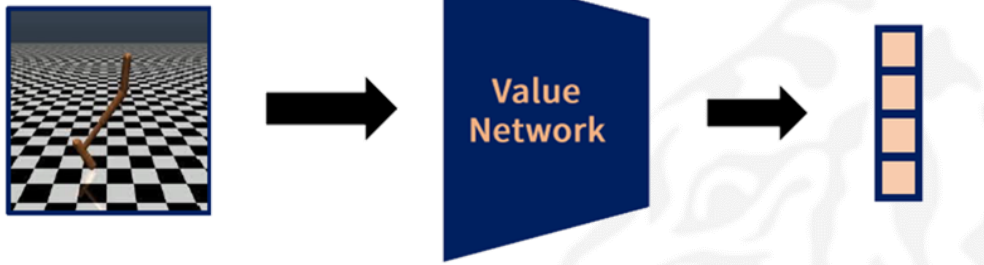
Deterministic Policy

ICLR 2016

Deep Deterministic Policy

연속적인 행동 공간에서의 Q-Learning

➔ 행동 이산화(Discretization)



행동공간이 $-1 \leq a \leq 1$ 의 범위를 가진다고 가정

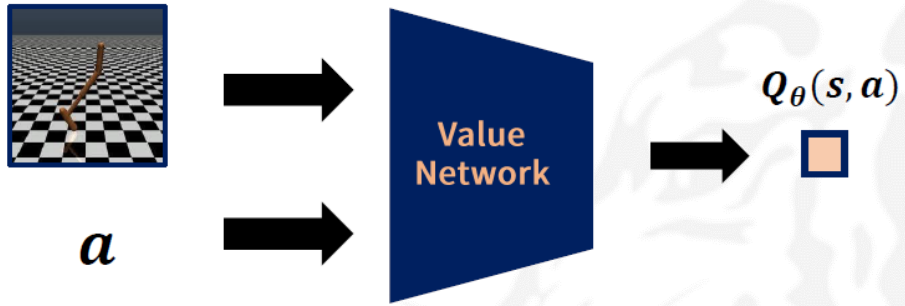
원하는 대로 (Δa = 임의의 값) 이산화

$[-1, -1 + \Delta a, -1 + 2\Delta a, .., 1]$

좋은 Δa? 너무 큰 Δa → 제어값의 정확도가 낮아짐

너무 작은 Δa → 계산/학습에 불리

연속적인 행동 공간에서의 Q-Learning
➔ 행동을 Critic의 Input으로.

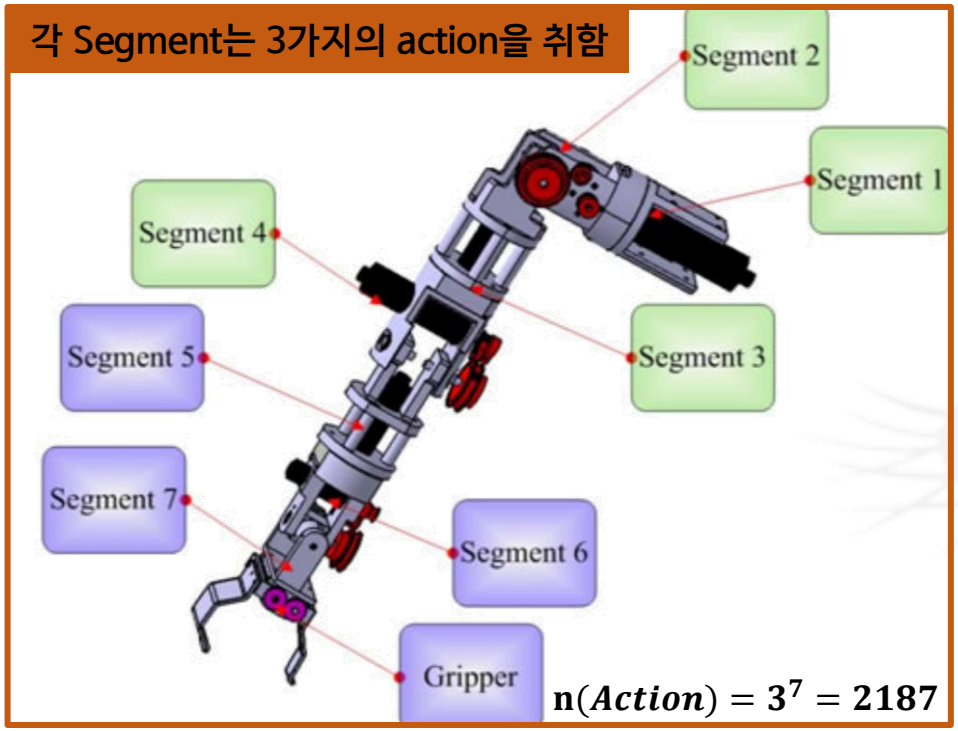


$$Q(s,a) \leftarrow Q(s,a) + \eta\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

Q-Learning target 계산에 필요한 $\max_{a'} \boldsymbol{Q_\theta}(s',a')$ 를 찾는 것이 매우 힘듦.

- $\boldsymbol{Q_\theta}(s',a')$ 는 (대다수의 경우) a' 에 대해 볼록 함수가 아니기 때문.
- 즉, Q-Learning 자체가 비효율적이 됨.

Robotics Domain



문제점 …

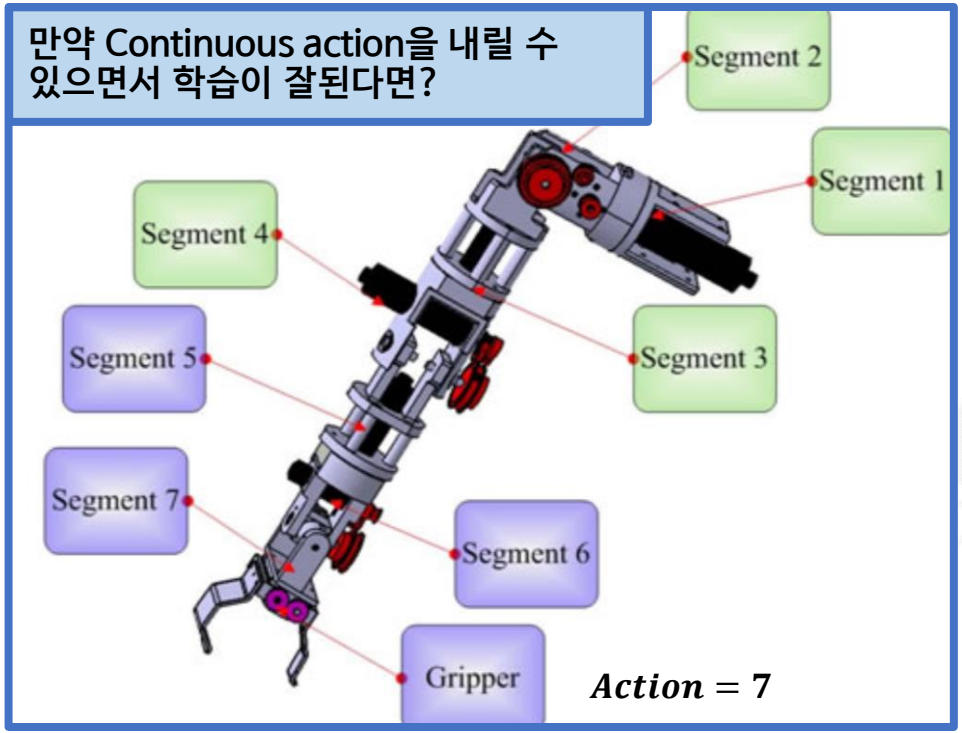
1. 확률적 정책에서의 너무 큰 행동 공간
2. 이산화(discretization)에 의한 Loss 발생

➔학습이 잘 되지 않고,

➔정교한 행동을 내리지못함

정책경사> DDPG > DDPG에서는?

만약 Continuous action을 내릴 수 있으면서 학습이 잘된다면?



목적: 연속적인 정책 함수 $\pi(a|s)$

방법: Actor-Critic을 활용

- Actor: 결정적(Deterministic) 정책 함수
- Critic: Q-Network

결론

1. 연속적인 정책 함수 구현 가능
2. 학습이 잘 됨으로써, 정교한 Action 가능

정책경사> DDPG > Deterministic Policy 가정

- 일반적인 Q-Learning은 환경 ($\boldsymbol{E}$)과 정책($\boldsymbol{\pi}$)이 $Q^{\pi}(s_t, a_t)$를 결정

➔$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim \boldsymbol{E}}[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \boldsymbol{\pi}}\ [Q^{\pi}(s_{t+1}, a_{t+1})]]$

- $Q^{\mu}(s_t, a_t)$의 기댓값 계산에 환경 ($\boldsymbol{E}$)만 영향을 미침
- 확정적 정책에 대한 표기를 $\mu(s_t)$로 하고, 행동 가치함수를 $Q^{\mu}(s_t, a_t)$ 다음과 같이 가정

➔$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim \boldsymbol{E}}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1})]$

Q_1)연속적이면서 확정적 정책을 학습시키는 방법이 무엇일까?
Q_2) 연속적이면서 확정적 정책을 학습시키기 위한 Loss function은 무엇일까?

결정적 정책의 cost 함수 $J(\mu_\theta)$:

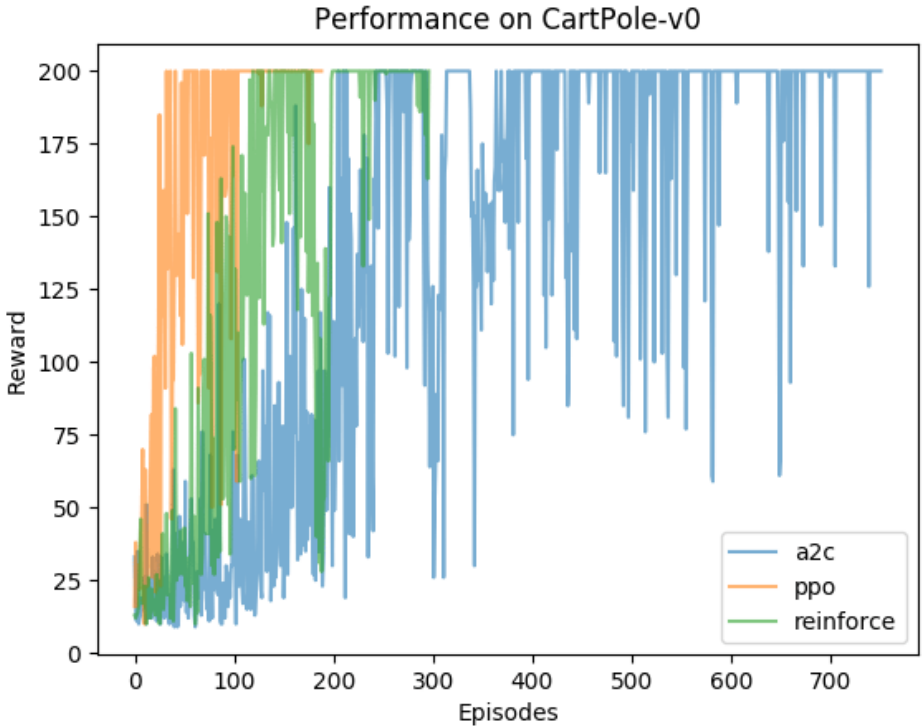
$$J(\mu_\theta) = \mathbb{E}_{s\sim p^{\pi_\theta}}[r(s,\mu_\theta(s))]$$

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s\sim p^{\pi_\theta}}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q(s,a)|_{a=\mu_\theta(s)}\right]$$

cf) 확률적 정책의 cost 함수 $J(\pi_\theta)$:

$$J(\pi_\theta) = \mathbb{E}_{s\sim p^{\pi_\theta}, a\sim\pi_\theta}[r(s,a)]$$

- DDPG는 Off-Policy Actor-Critic 모델
- Critic Loss: $\frac{1}{N}\sum_i^N (Q^\mu(s_i, a_i) - r_i + \gamma Q^\mu(s_{i+1}, \mu(s_{i+1})))^2$
- Actor Loss: $\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s\sim p^{\pi_\theta}}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q(s,a)|_{a=\mu_\theta(s)}\right]$



[Key Point]

- **받는 보상 최대**로 하면서,

  정책의 분산을 줄이고자 함.

- 두 목적함수:
  - **Min:** $\nabla_\theta \pi_\theta(s_t, a_t)$ → 새로운 목적
  - **Max:** $J(\theta)$ → **Maximize** $L(\theta)$

- 어떻게? → Surrogate 함수($L(\theta)$)
  - 이전 스텝에서 업데이트한 정책($\pi_{\theta_{OLD}}$)과 업데이트할 정책 (π_θ) 이 서로 비슷하다.
  - 이전 스텝의 목적함수보다 업데이트 할 목적함수가 항상 크다.
  - 즉, $\pi_\theta \cong \pi_{\theta_{OLD}}$, $J(\theta) - J(\theta_{OLD}) = L(\theta) > 0$을 항상 만족.

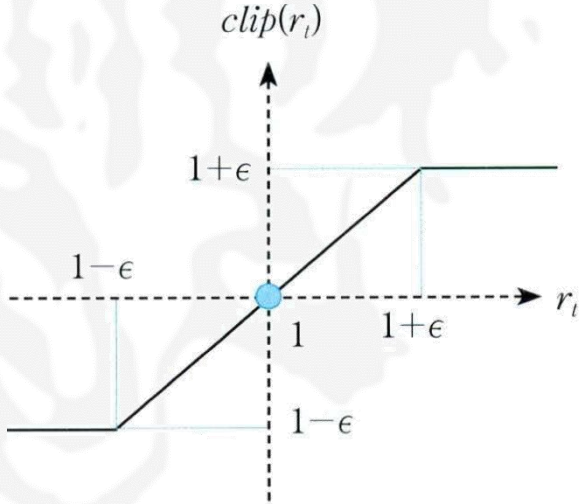
$$L(\theta) = \sum_{t=0}^{\infty} \mathbb{E}_{\tau_{s_0:a_t} \sim p_{\theta_{OLD}}(\tau_{s_0:a_t})} \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{OLD}}(a_t|s_t)} \gamma^t A^{\pi_{\theta_{OLD}}}(s_t, a_t) \right]$$

- "$\pi_\theta \cong \pi_{\theta_{OLD}}$"을 구현하기 위한 수학적 도구
  - **Clip function**

- π_θ와 $\pi_{\theta_{OLD}}$의 비율을 다음과 같이 정의하자.
- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{OLD}}(a_t|s_t)}$

- "$\pi_\theta \cong \pi_{\theta_{OLD}}$" 을 구현하기 위한 Clip 함수를 다음과 같이 정의하자.
- $clip(r_t(\theta), 1-\epsilon, 1+\epsilon) = \begin{cases} 1+\epsilon, & if\ r_t(\theta) \geq 1+\epsilon \\ 1-\epsilon, & if\ r_t(\theta) \leq 1-\epsilon \\ r_t(\theta), & otherwise. \end{cases}$

ϵ은 작은 상수임 ($\epsilon = 0.01$)



```python
import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical
learning_rate = 0.0002
gamma = 0.98
class REINFORCE(nn.Module):
    def __init__(self):
        super(REINFORCE, self).__init__()
        self.data = []
        self.fc1 = nn.Linear(4, 128) # state 의 구성요소 4개 (cart position/velocity, pole angle/angular velocity)
        self.fc2 = nn.Linear(128, 2) # action 2개에 대한 확률 (left, right)
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)  # policy parameters 의미
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.softmax(self.fc2(x), dim = 0) #확률적 정책이 나오니까 // 첫번째 차원에 대해 softmax 적용
        return x
    def put_data(self, item): # return 계산 위한 샘플 추가
        self.data.append(item)
    def train_net(self):
        R = 0 # 초기 셋팅0
        #print(self.state_dict()['fc1.weight'])
        self.optimizer.zero_grad() #  init gradient
        for r, prob in self.data[::-1]: # 연산의 편의성을 위해 뒤에서 부터 계산
            R = r + gamma * R
            loss = -torch.log(prob) * R
            loss.backward() # back propagation
        self.optimizer.step() # weight update
        #print(self.state_dict()['fc1.weight'])
        self.data = []
```

```python
def main():
    env = gym.make('CartPole-v1')
    pi = REINFORCE()
    score = 0.0
    print_interval = 20

    for n_epi in range(10000): #episode 10000번
        s = env.reset()
        done = False
        while not done: # CartPole-v1 forced to terminates at 500 step.하나의 episode 의 step size
            prob = pi(torch.from_numpy(s).float()) # s: observation 구성 요소가 4개 # phi_theta(a|s)
            # print(prob) 하면 2개의 액션에 대한 확률이 나옴. ex) tensor[0.5389, 0.4611]
            #print(prob)
            m = Categorical(prob)
            # softmax 통해 나온 각 행동에 대한 확률을 확률밀도함수로 만들어 줌
            a = m.sample()
            s_prime, r, done, info = env.step(a.item())  # pytorch tensor 의 값만 뽑아내려고, o or 1
            pi.put_data((r,prob[a]))
            s = s_prime
            score += r # 1 for every step taken
        pi.train_net()  # 에피소드 한번마다 신경망 파라미터 업데이트
        # 그럼 forward x 에 들어가는 데이터가 ()
        if n_epi%print_interval==0 and n_epi!=0:
                print("# of episode :{}, avg score : {}".format(n_epi, score/print_interval))
                score = 0.0
    env.close()

if __name__ == '__main__':
    main()
```

Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory and Implementation

Policy Gradient

Imitation Learning

Autonomous Mobility Applications

- Gameplay

Pro-Gamer

Trained Agent

The goal of Imitation Learning is to train a policy to mimic **the expert's demonstrations**

- Starcraft2

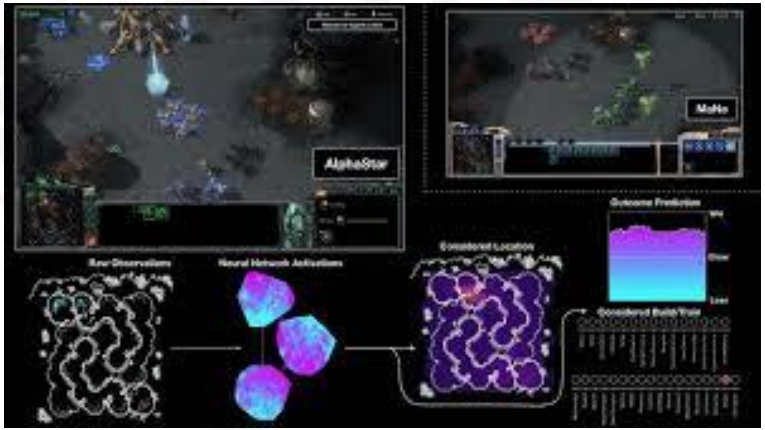
States: s = **minimap**, **screen**
Action: a **= select**, **drag**
Training set: $D = \{\tau := (s, a)\}$ from expert
Goal: learn $\pi_\theta(s) \rightarrow a$

States: s
Action: a
Policy: π_θ
- Policy maps states to actions : $\pi_\theta(s) \rightarrow a$
- Distributions over actions : $\pi_\theta(s) \rightarrow P(a)$

State Dynamics: $P(s'|s,a)$
- Typically not known to policy
- Essentially the simulator/environment

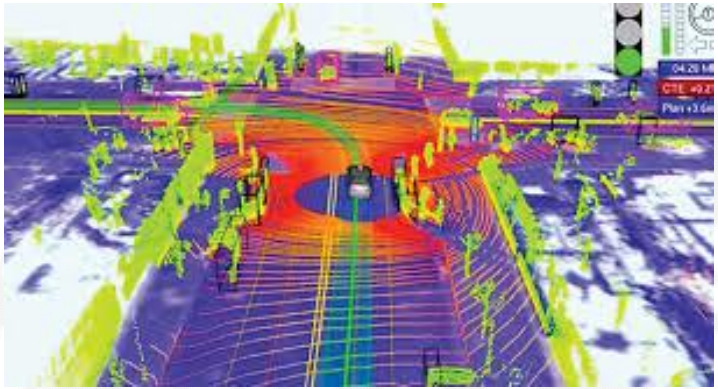
Rollout: sequentially execute $\pi_\theta(s_0)$ on initial state
- Produce trajectories $\boldsymbol{\tau}$

$\mathbf{P}(\boldsymbol{\tau}|\boldsymbol{\pi})$**:** distribution of trajectories induced by a policy
$\mathbf{P}(\mathbf{s}|\boldsymbol{\pi})$**:** distribution of states induced by a policy



- Autonomous Driving Control

States: s = **sensors**
Action: a = **steering wheel**, **brake**, …
Training set: $D = \{\tau := (s, a)\}$ from expert
Goal: learn $\pi_\theta(s) \rightarrow a$



- PPF/RFTN Injection Control in Medicine

States: s = **BIS**, **BP**, …
Action: a **= PPF**, **RFTN**, …
Training set: $D = \{\tau := (s, a)\}$ from expert
Goal: learn $\pi_\theta(\mathrm{s}) \rightarrow \mathrm{a}$

Introduction and Preliminaries

Deep Reinforcement Learning Theory and Implementation

Policy Gradient

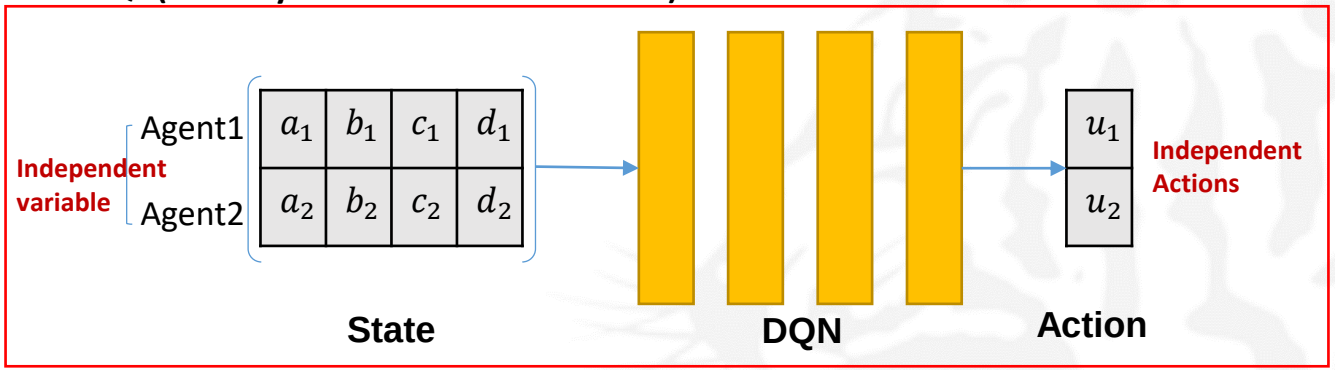
Imitation Learning

Autonomous Mobility Applications

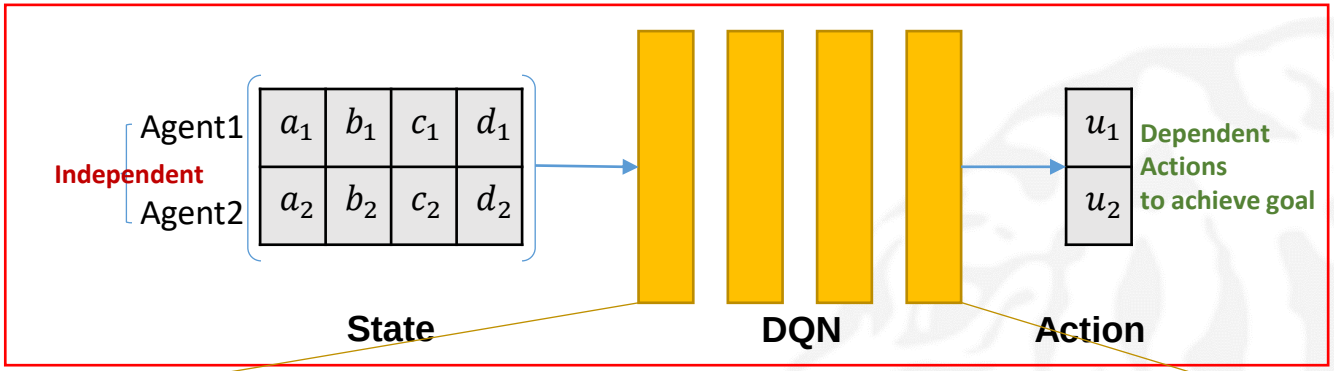
- **MADRL**
- Applications

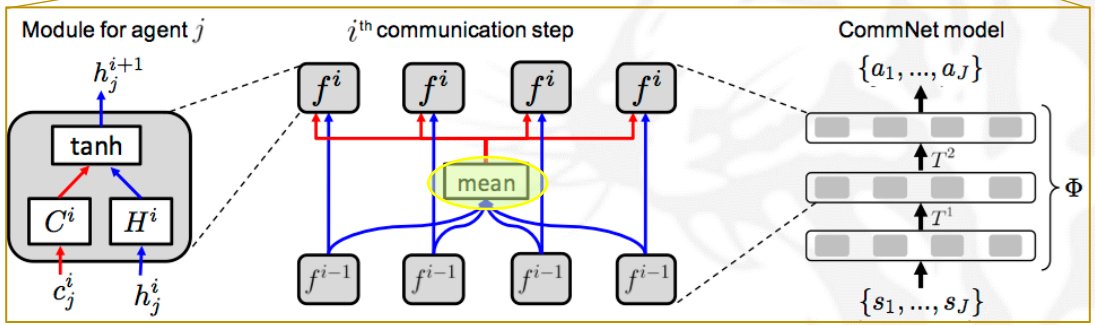
With Previous method

With DQN(Partially Observable Environment)



DQN-based CommNet



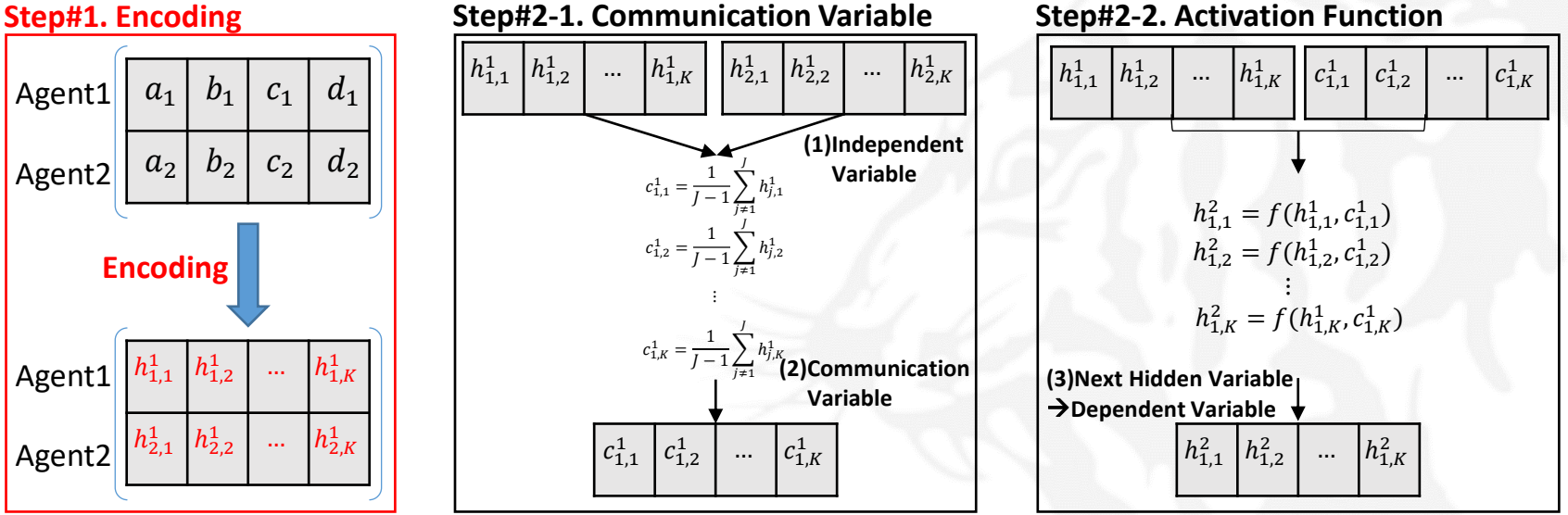


h_j^i : j-th agent's hidden state variable in i-th layer

c_j^i : j-th agent's communitive state variable in i-th layer

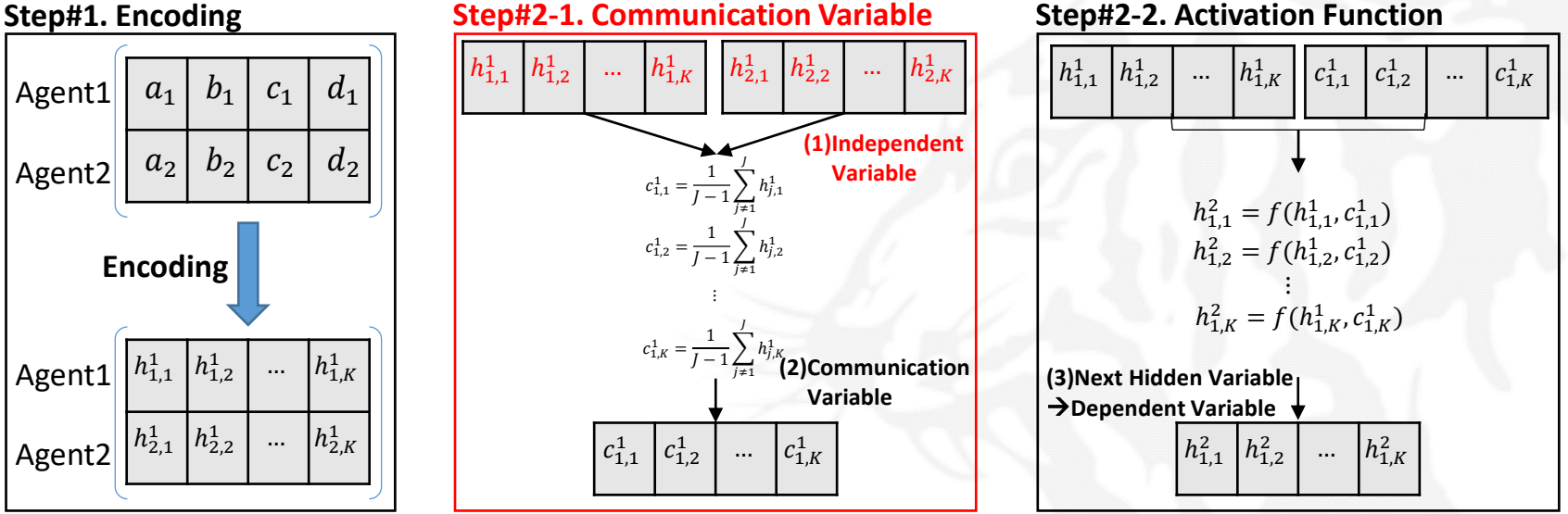
$$h_j^{i+1} = f^i(h_j^i, c_j^i)$$

CommNet Mechanism

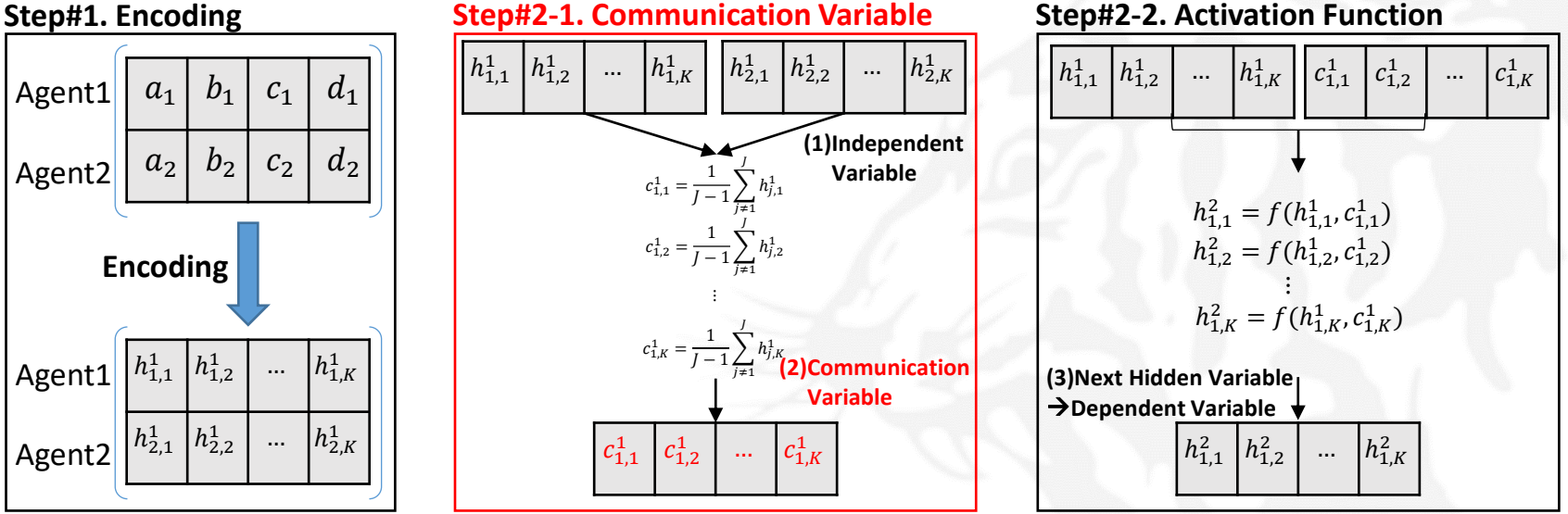


[3] S. Sukhbaatar et al., Learning Multiagent Communication with Backpropagation, *NIPS 2016*

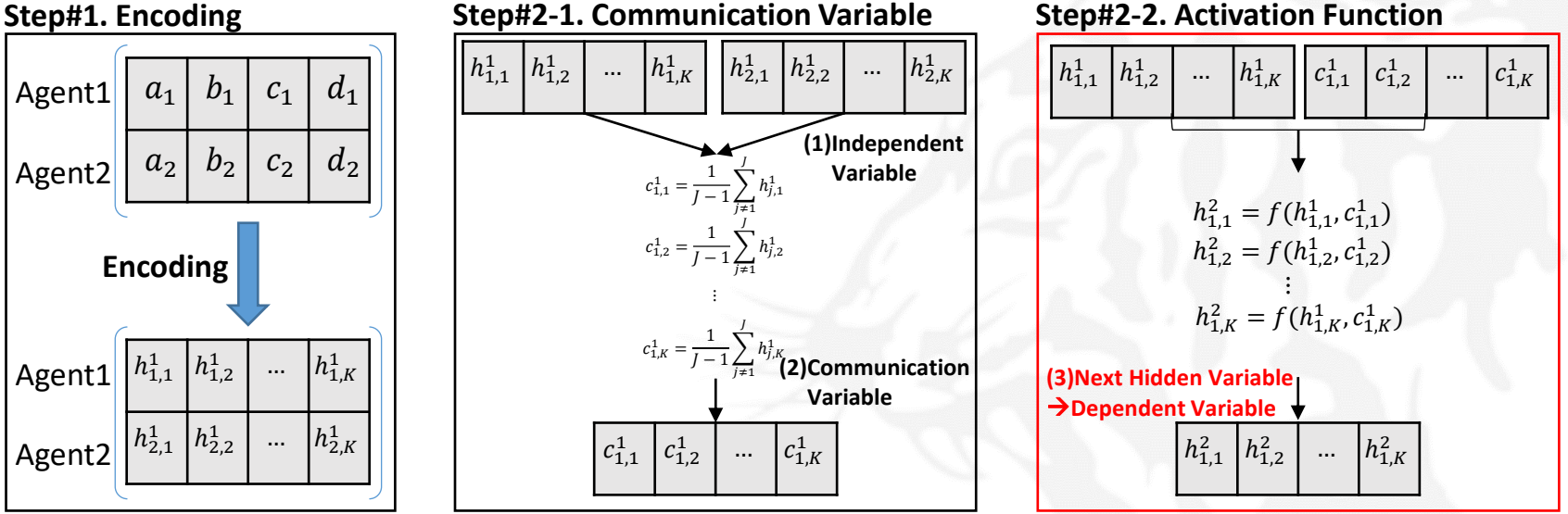
CommNet Mechanism

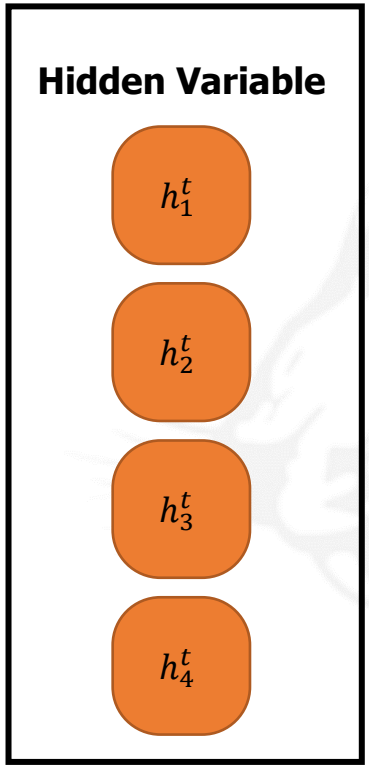


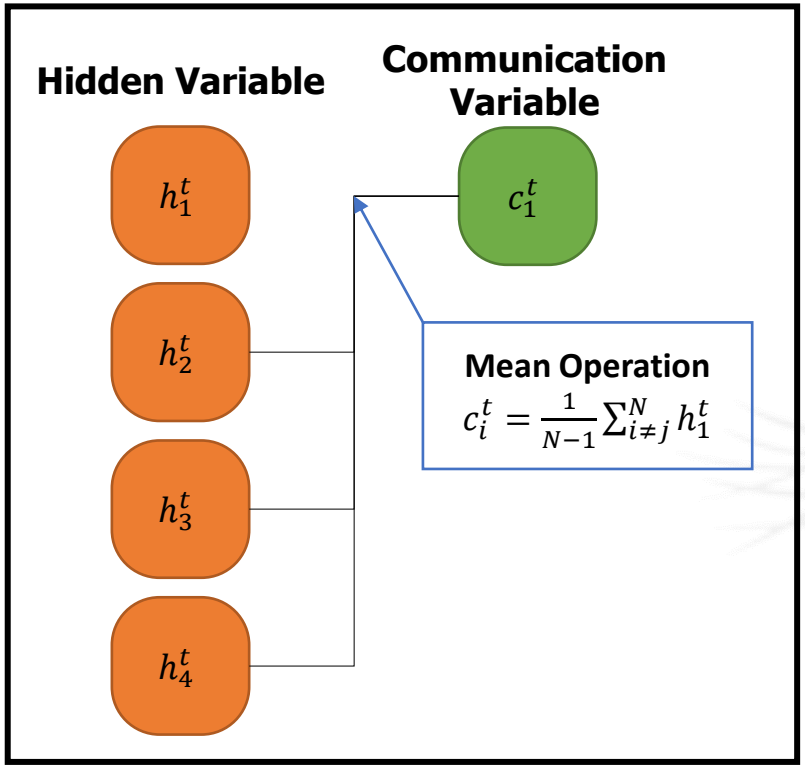
CommNet Mechanism

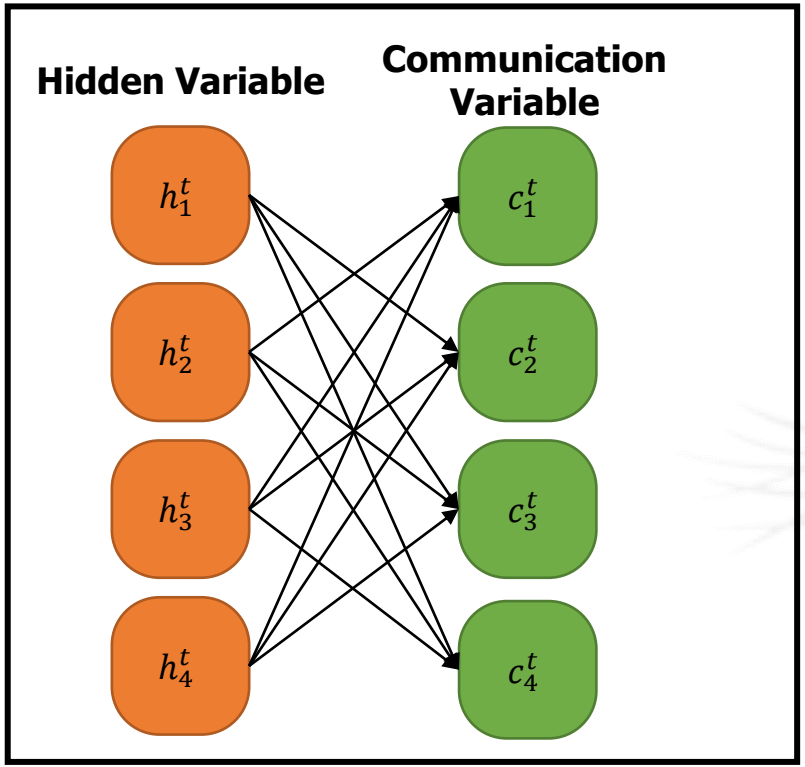


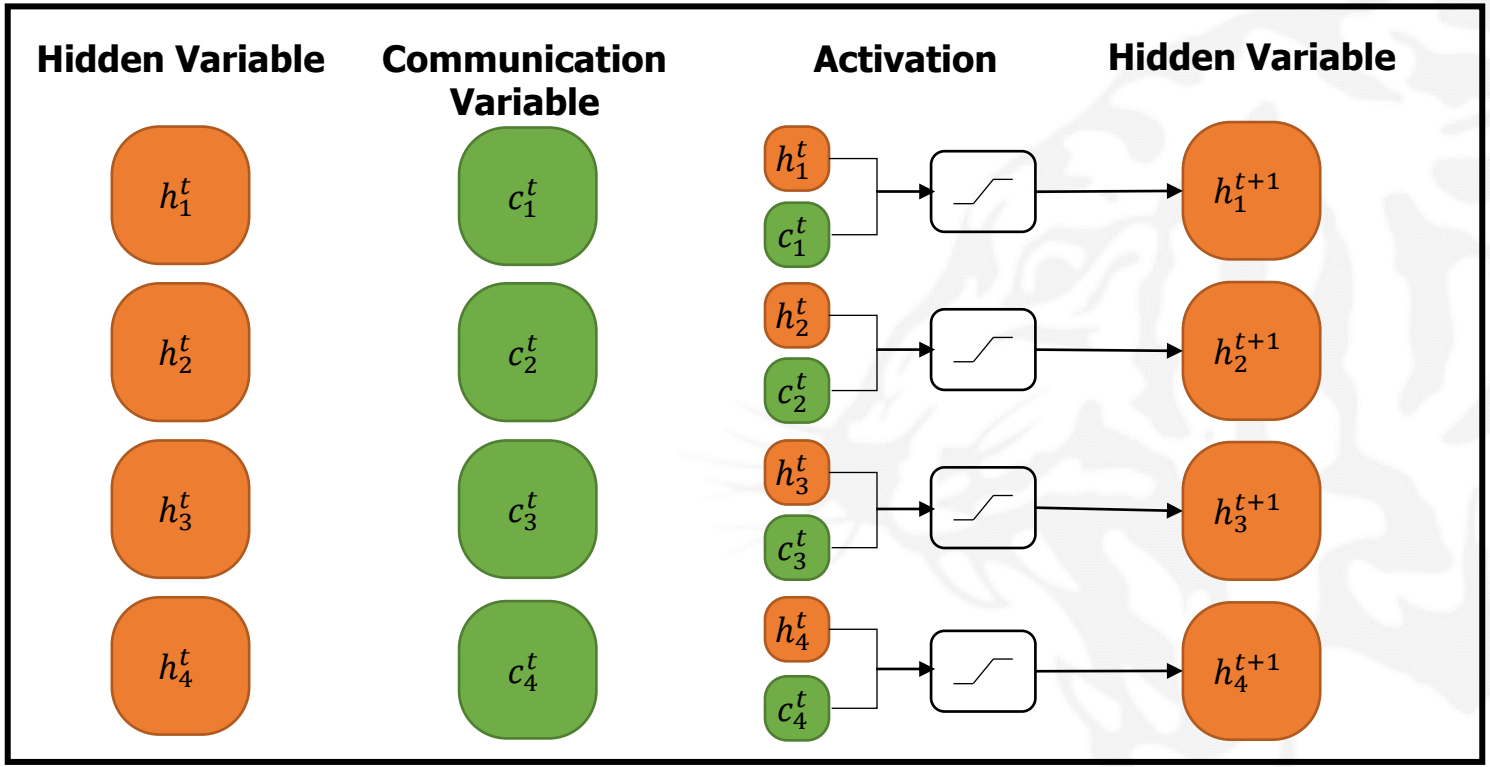
CommNet Mechanism




Hidden Variable
h_1^t
h_2^t
h_3^t
h_4^t

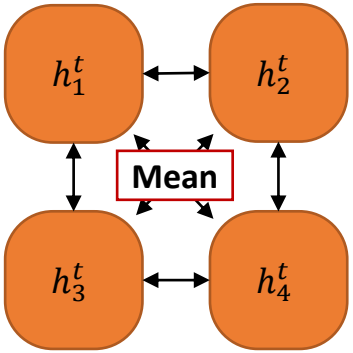

Hidden Variable
Communication Variable
h_1^t
h_2^t
h_3^t
h_4^t
c_1^t
Mean Operation
$c_i^t = \frac{1}{N-1}\sum_{i \neq j}^{N} h_1^t$


Hidden Variable
Communication Variable
h_1^t
h_2^t
h_3^t
h_4^t
c_1^t
c_2^t
c_3^t
c_4^t


Hidden Variable
Communication Variable
Activation
Hidden Variable
h_1^t
h_2^t
h_3^t
h_4^t
c_1^t
c_2^t
c_3^t
c_4^t
h_1^t
c_1^t
h_2^t
c_2^t
h_3^t
c_3^t
h_4^t
c_4^t
h_1^{t+1}
h_2^{t+1}
h_3^{t+1}
h_4^{t+1}

Now we are curious that…

CommNet



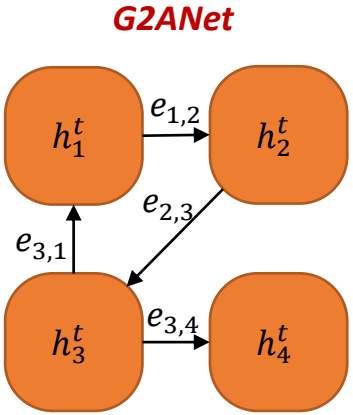
In Graph Approach.

1. Should the agent communicate with all agent?

2. Can we transfer only essential information between agents?

→ ***G2ANet will be the solution to the above problem.***

Now we are curious that…

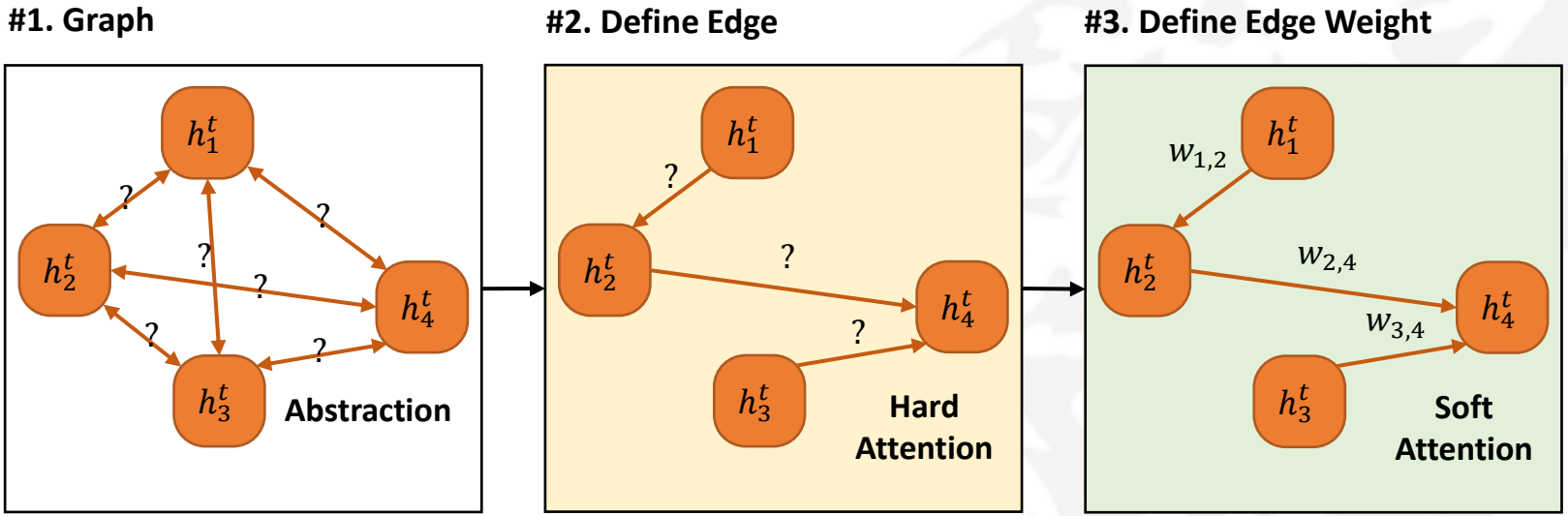


In Graph Approach.

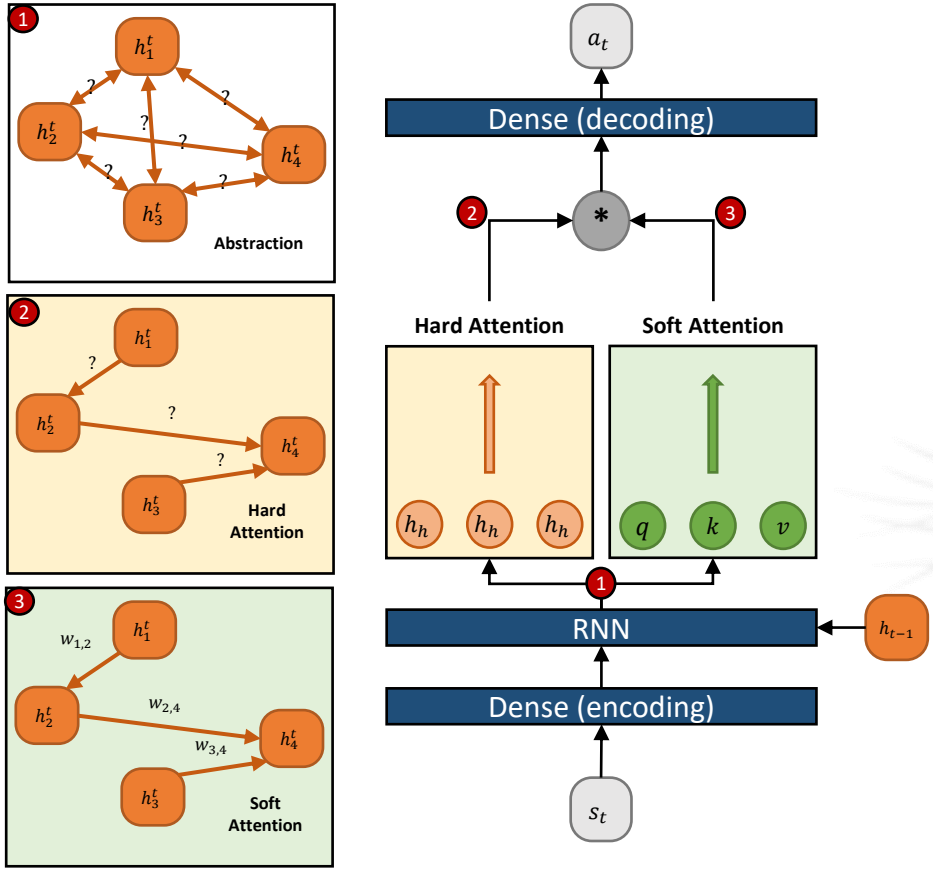
1. Should the agent communicate with all agent?
2. Can we transfer only essential information between agents?

***G2ANet* will be the solution to the above problem.**

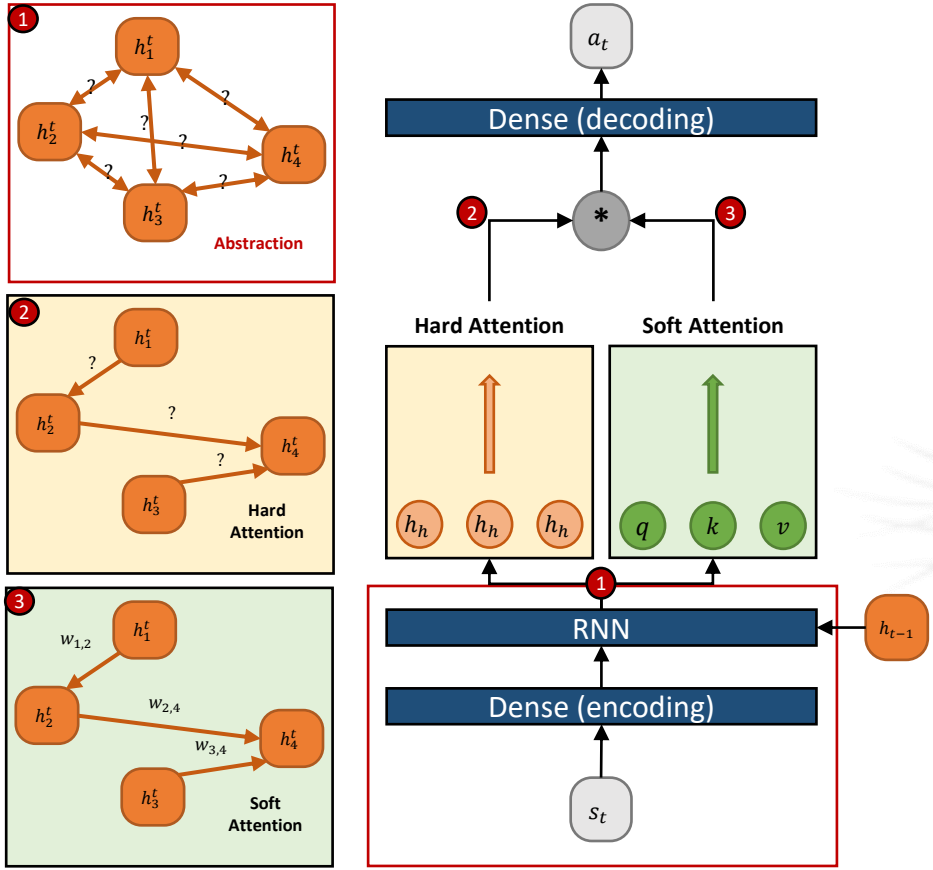
[4] Y. Liu et al., Multi-Agent Game Abstraction via Graph Attention Neural Network, *Proc. AAAI 2020*


#1. Graph
h_1^t
h_2^t
h_3^t
h_4^t
?
Abstraction
#2. Define Edge
h_1^t
h_2^t
h_3^t
h_4^t
?
Hard Attention
#3. Define Edge Weight
h_1^t
h_2^t
h_3^t
h_4^t
$w_{1,2}$
$w_{2,4}$
$w_{3,4}$
Soft Attention

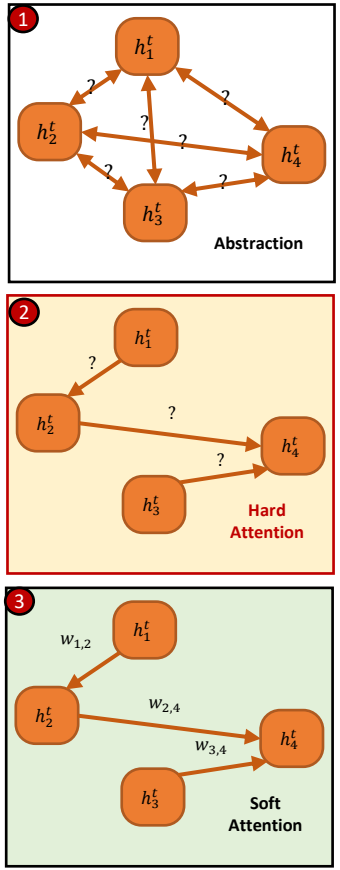
G2ANet Architecture

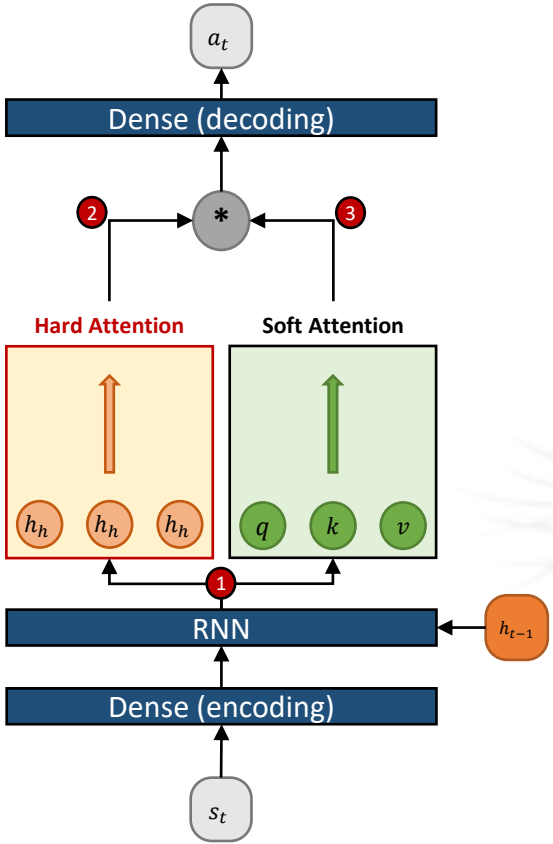


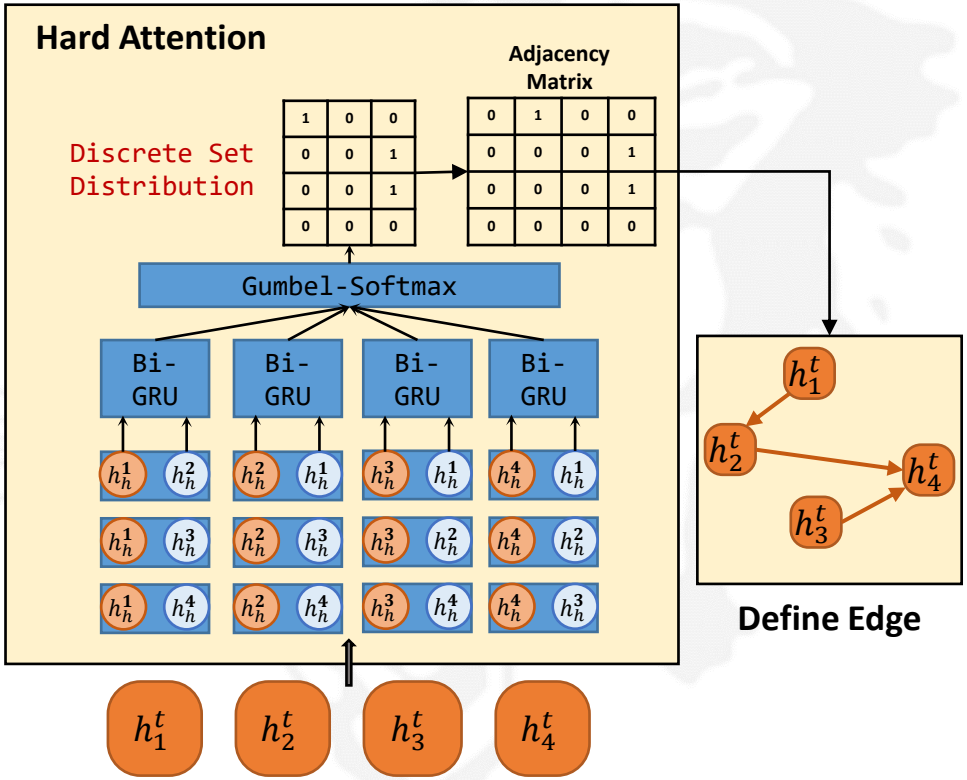
G2ANet Architecture: Abstraction



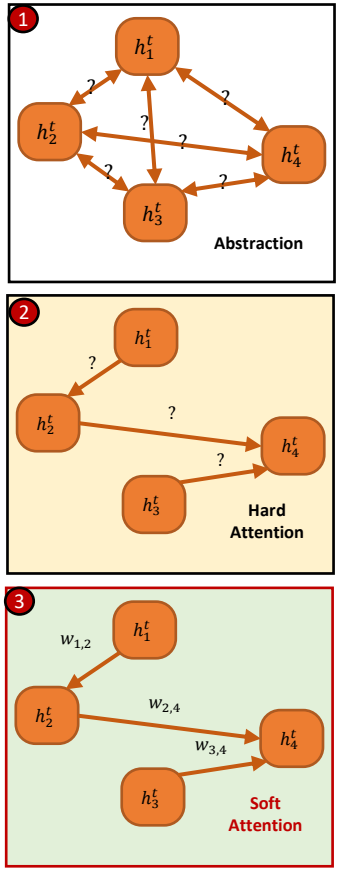
G2ANet Architecture: Hard Attention

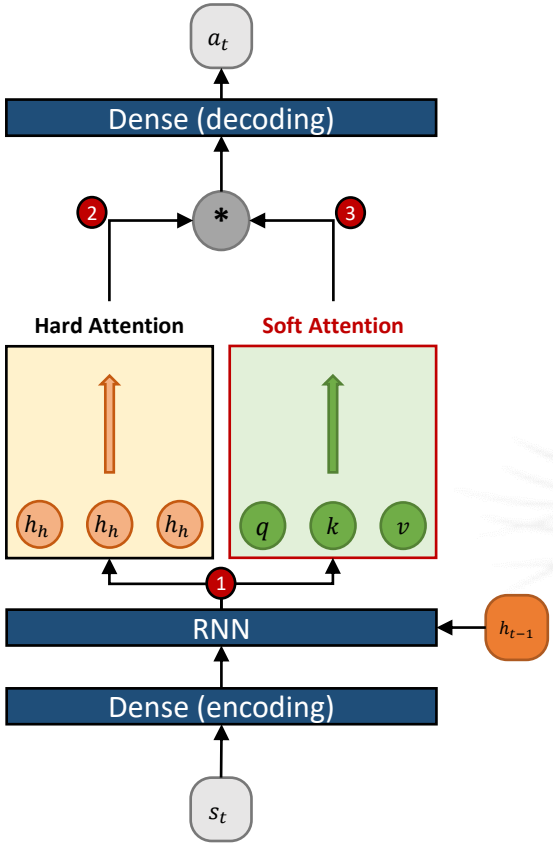


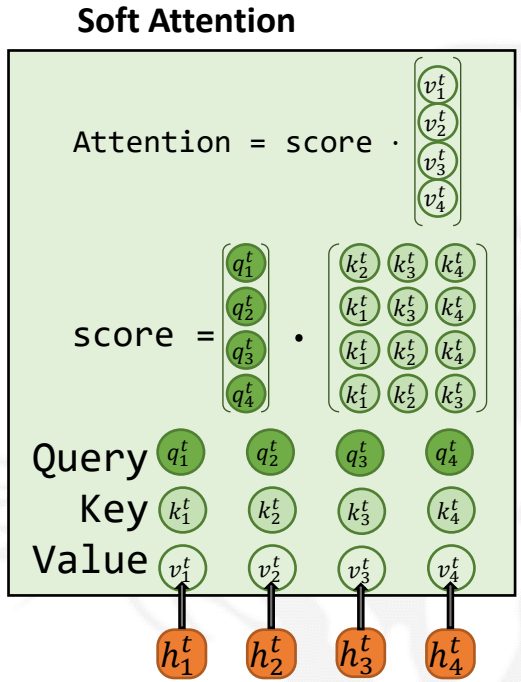




G2ANet Architecture: Soft Attention







$$Score = \boldsymbol{q} \cdot \boldsymbol{k} = \boldsymbol{q^T} * \boldsymbol{k}$$

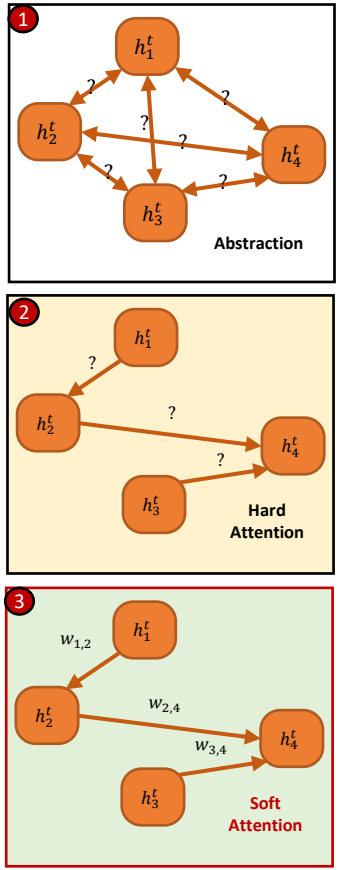
$$Score_{scaled} = \frac{Score}{\sqrt{n}}$$

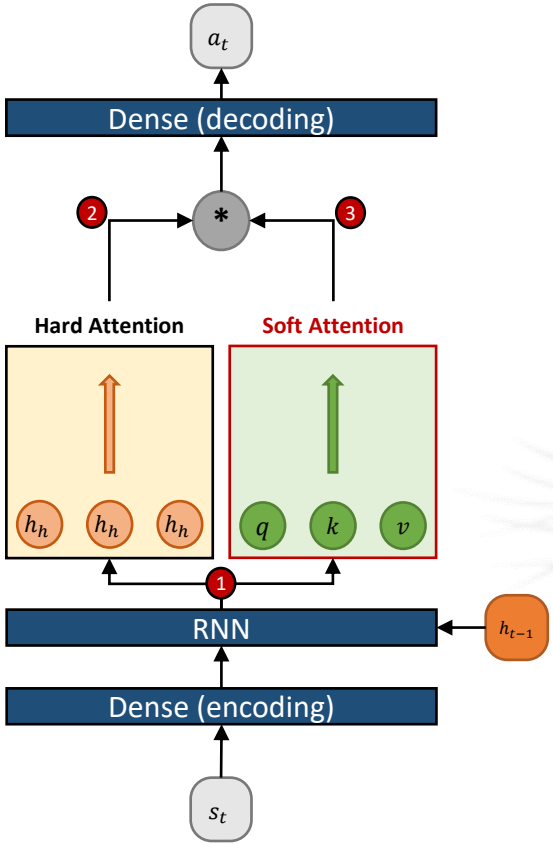
$$Attention(\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}) = Score_{scaled} * \boldsymbol{v}$$

Soft Attention Output (message)

0.11	0.84	0.4
0.1	0.18	0.72
0.34	0.38	0.28
0.16	0.14	0.70

G2ANet Architecture: Soft Attention & GNN





Hard Attention Output (connection)

1	0	0
0	0	1
0	0	1
0	0	0

*

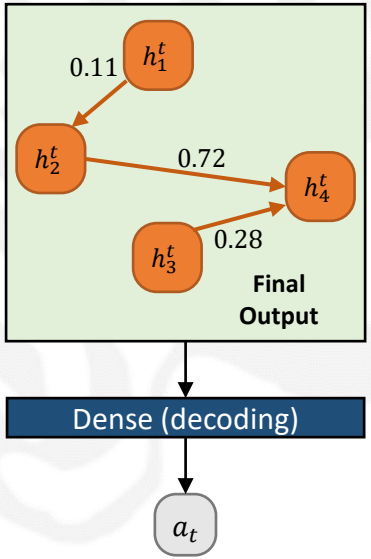
Soft Attention Output (message)

0.11	0.84	0.4
0.1	0.18	0.72
0.34	0.38	0.28
0.16	0.14	0.70

=

Final Output (connection & message)

0.11	0	0
0	0	0.72
0	0	0.28
0	0	0



Introduction and Preliminaries

Deep Reinforcement Learning Theory and Implementation

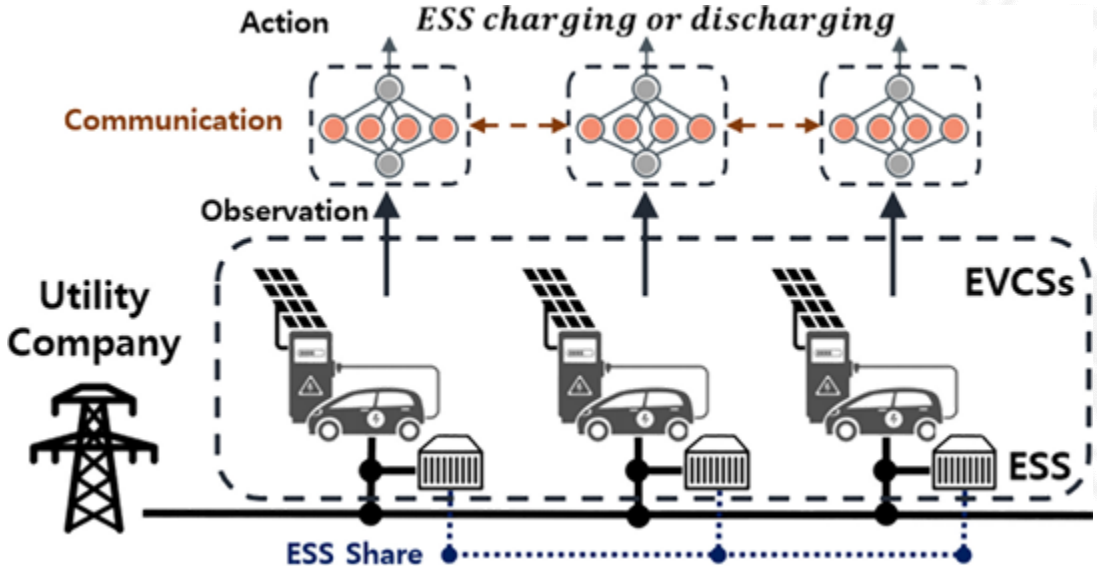
Policy Gradient

Imitation Learning

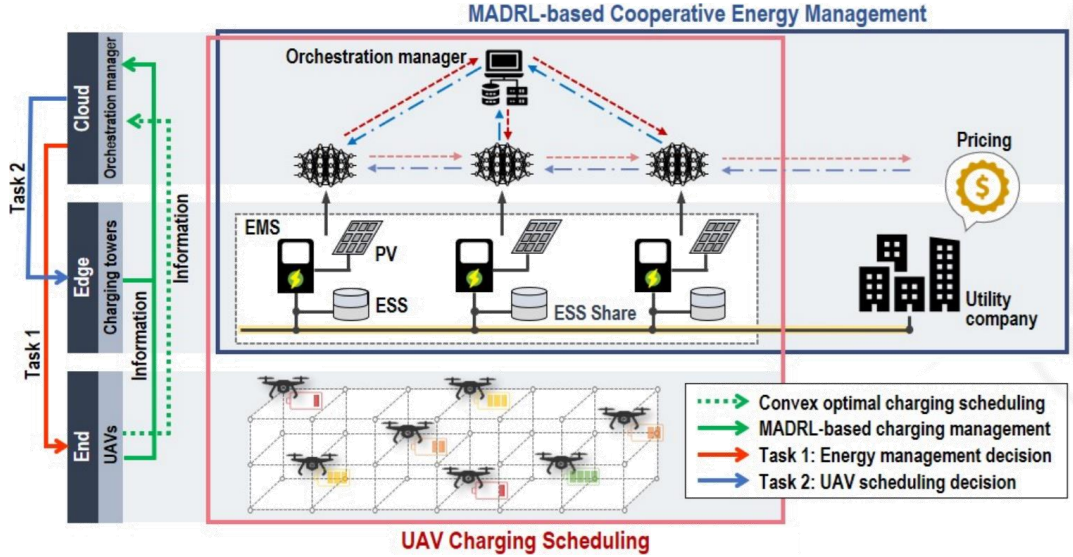
Autonomous Mobility Applications

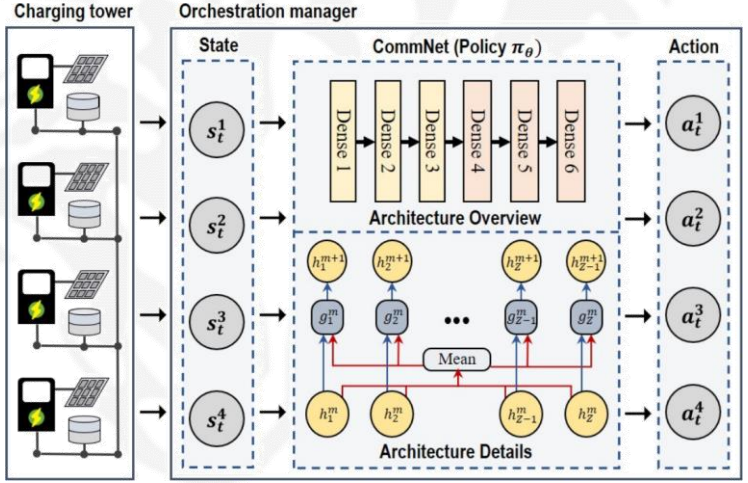
- MADRL
- **Applications**

- Electric Vehicle Charging
  - MyungJae Shin, Dae-Hyun Choi, and Joongheon Kim, "**Cooperative Management for PV/ESS-Enabled Electric-Vehicle Charging Stations: A Multiagent Deep Reinforcement Learning Approach**," IEEE Transactions on Industrial Informatics, 16(5):3493-3503, May 2020.

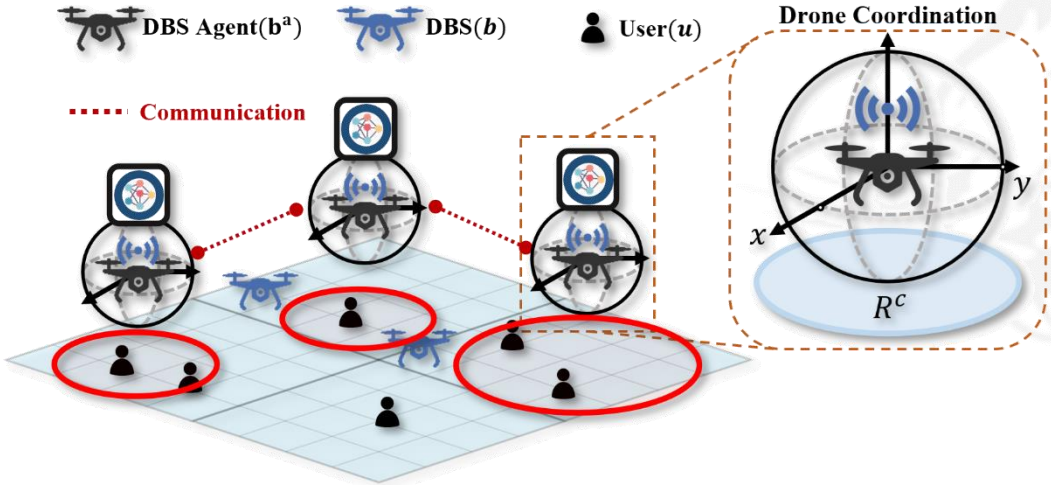


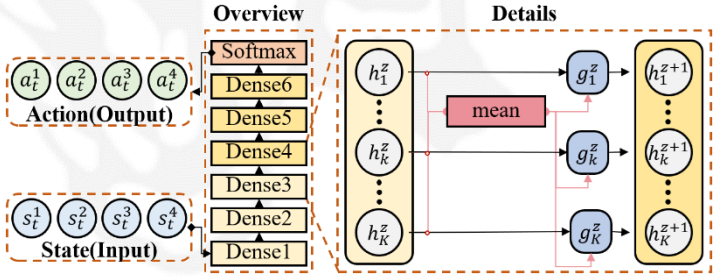
- Multi-UAV Charging
  - Soyi Jung, Won Joon Yun, MyungJae Shin, Joongheon Kim, and Jae-Hyun Kim, "**Orchestrated Scheduling and Multi-Agent Deep Reinforcement Learning for Cloud-Assisted Multi-UAV Charging Systems**," IEEE Transactions on Vehicular Technology, 70(6):5362-5377, June 2021.



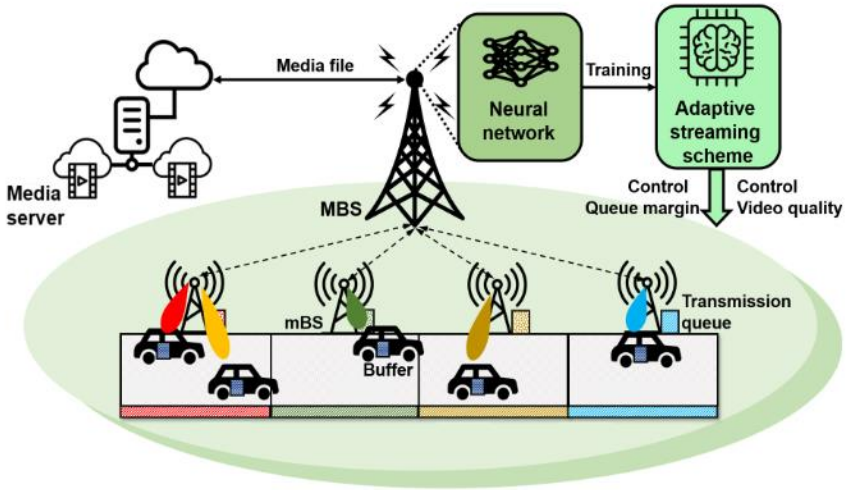


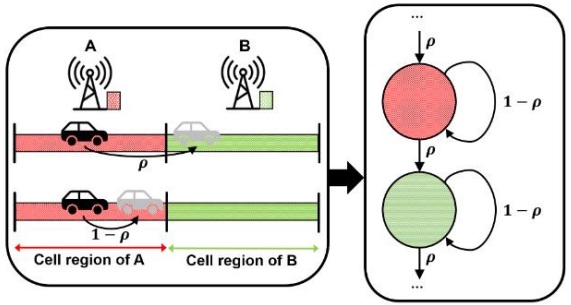
- Multi-UAV Surveillance
  - Won Joon Yun, Soohyun Park, Joongheon Kim, MyungJae Shin, Soyi Jung, David Mohaisen, and Jae-Hyun Kim, "**Cooperative Multi-Agent Deep Reinforcement Learning for Reliable Surveillance via Autonomous Multi-UAV Control**," IEEE Transactions on Industrial Informatics, (To Appear).

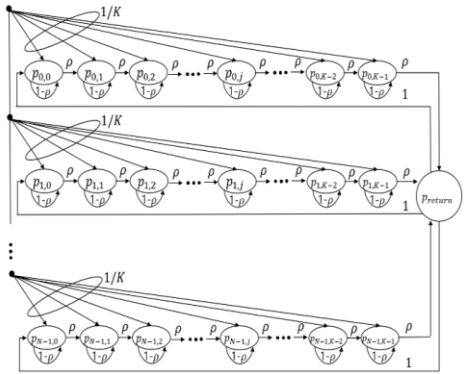




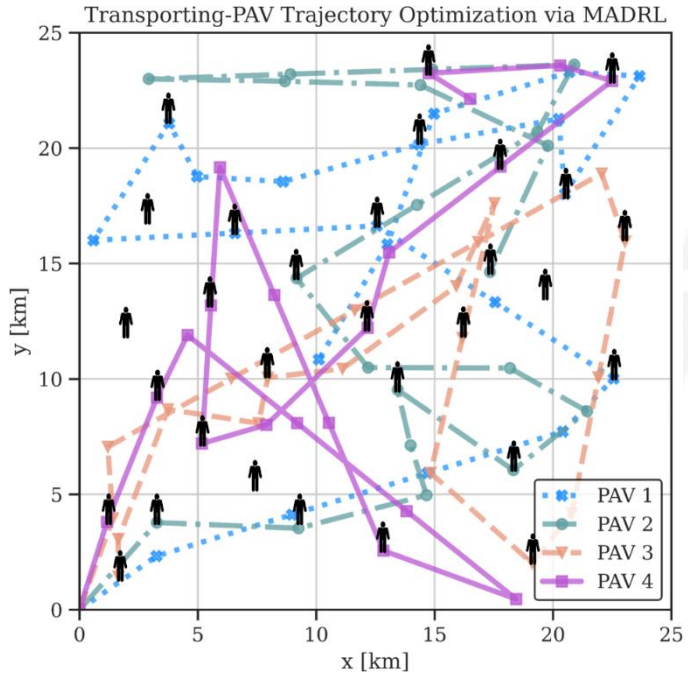
- Streaming in Connected Vehicles
  - Won Joon Yun, Dohyun Kwon, Minseok Choi, Joongheon Kim, Giuseppe Caire, and Andreas F. Molisch, "**Quality-Aware Deep Reinforcement Learning for Streaming in Infrastructure-Assisted Connected Vehicles**," IEEE Transactions on Vehicular Technology, (To Appear).

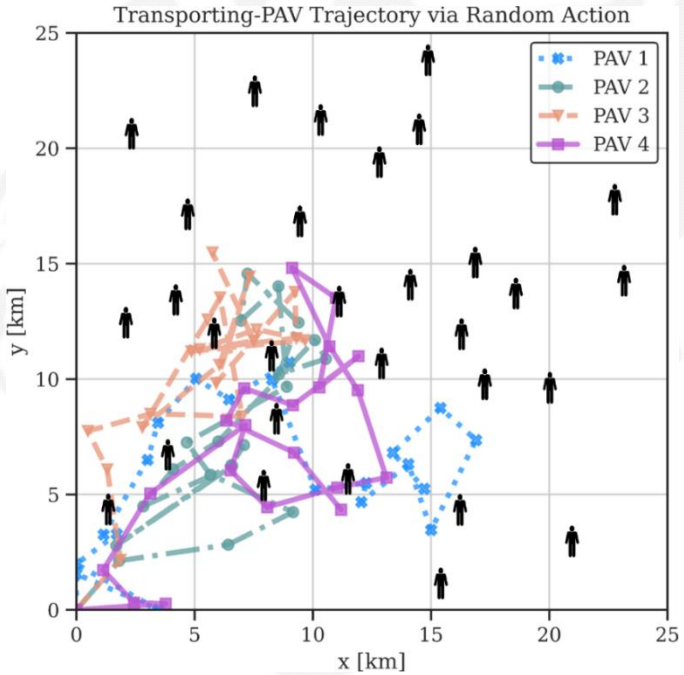






- Drone-Taxi Trajectory Learning
  - Won Joon Yun, Soyi Jung, Joongheon Kim, and Jae-Hyun Kim, "**Distributed Deep Reinforcement Learning for Autonomous Aerial eVTOL Mobility in Drone Taxi Applications**," ICT Express (Elsevier), 7(1):1-4, March 2021.





Thank you for your attention!

- More questions?
  - joongheon@korea.ac.kr