Smart Mobile Platform

Deep Reinforcement Learning (DRL)

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

Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

Inverse Reinforcement Learning and Imitation Learning

- **<u>Introduction and Applications</u>**
- Dynamic Programming
- Q-Learning and Markov Decision Process

Deep Learning Revolution is Real

Geoffrey E Hinton

Yoshua Bengio

Yann LeCun

FATHERS OF THE DEEP LEARNING REVOLUTION RECEIVE ACM A.M. TURING AWARD

Bengio, Hinton, and LeCun Ushered in Major Breakthroughs in Artificial Intelligence

ACM named Yoshua Bengio, Geoffrey Hinton, and Yann LeCun recipients of the 2018 ACM A.M. Turing Award for conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing. Bengio is Professor at the University of Montreal and Scientific Director at Mila, Quebec's Artificial Intelligence Institute; Hinton is VP and Engineering Fellow of Google, Chief Scientific Adviser of The Vector Institute, and University Professor Emeritus at the University of Toronto; and LeCun is Professor at New York University and VP and Chief AI Scientist at Facebook.

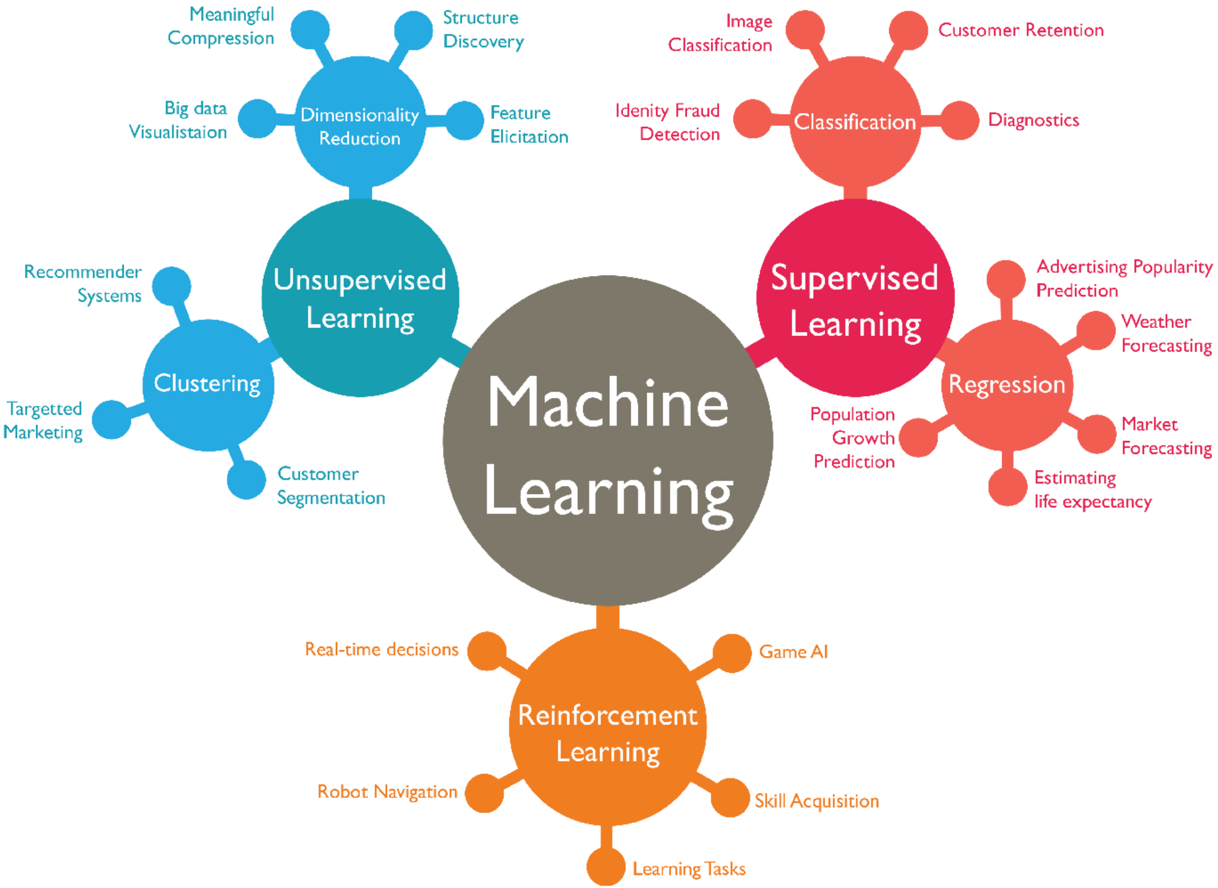
Working independently and together, Hinton, LeCun and Bengio developed conceptual foundations for the field, identified surprising phenomena through experiments, and contributed engineering advances that demonstrated the practical advantages of deep neural networks. In recent years, deep learning methods have been responsible for astonishing breakthroughs in computer vision, speech recognition, natural language processing, and robotics—among other applications.

While the use of artificial neural networks as a tool to help computers recognize patterns and simulate human intelligence had been introduced in the 1980s, by the early 2000s, LeCun, Hinton and Bengio were among a small group who remained committed to this approach. Though their efforts to rekindle the AI community's interest in neural networks were initially met with skepticism, their ideas recently resulted in major technological advances, and their methodology is now the dominant paradigm in the field.

The ACM A.M. Turing Award, often referred to as the "Nobel Prize

Alan Turing (1912-1954)
Father of Computer Science

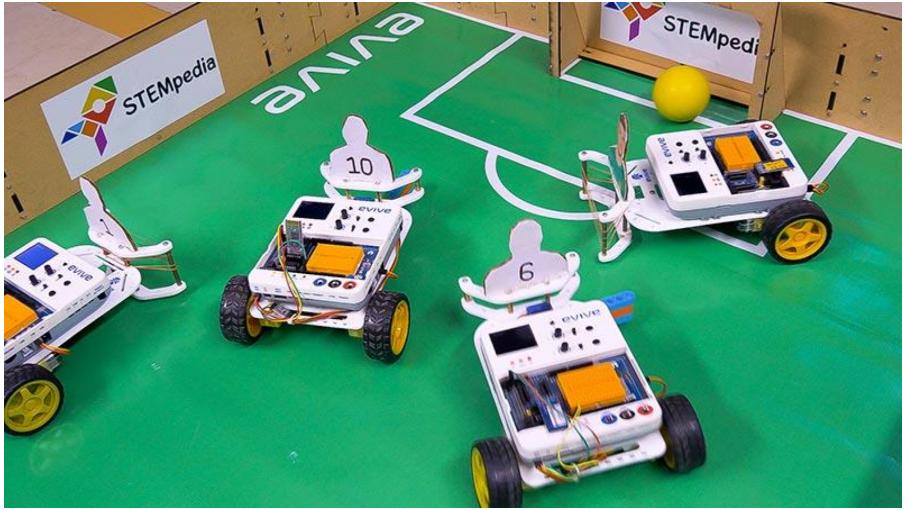
https://amturing.acm.org/


Meaningful Compression
Structure Discovery
Big data Visualistaion
Dimensionality Reduction
Feature Elicitation
Image Classification
Customer Retention
Idenity Fraud Detection
Classification
Diagnostics
Recommender Systems
Unsupervised Learning
Supervised Learning
Advertising Popularity Prediction
Weather Forecasting
Clustering
Regression
Targetted Marketing
Machine Learning
Population Growth Prediction
Market Forecasting
Customer Segmentation
Estimating life expectancy
Real-time decisions
Game AI
Reinforcement Learning
Robot Navigation
Skill Acquisition
Learning Tasks

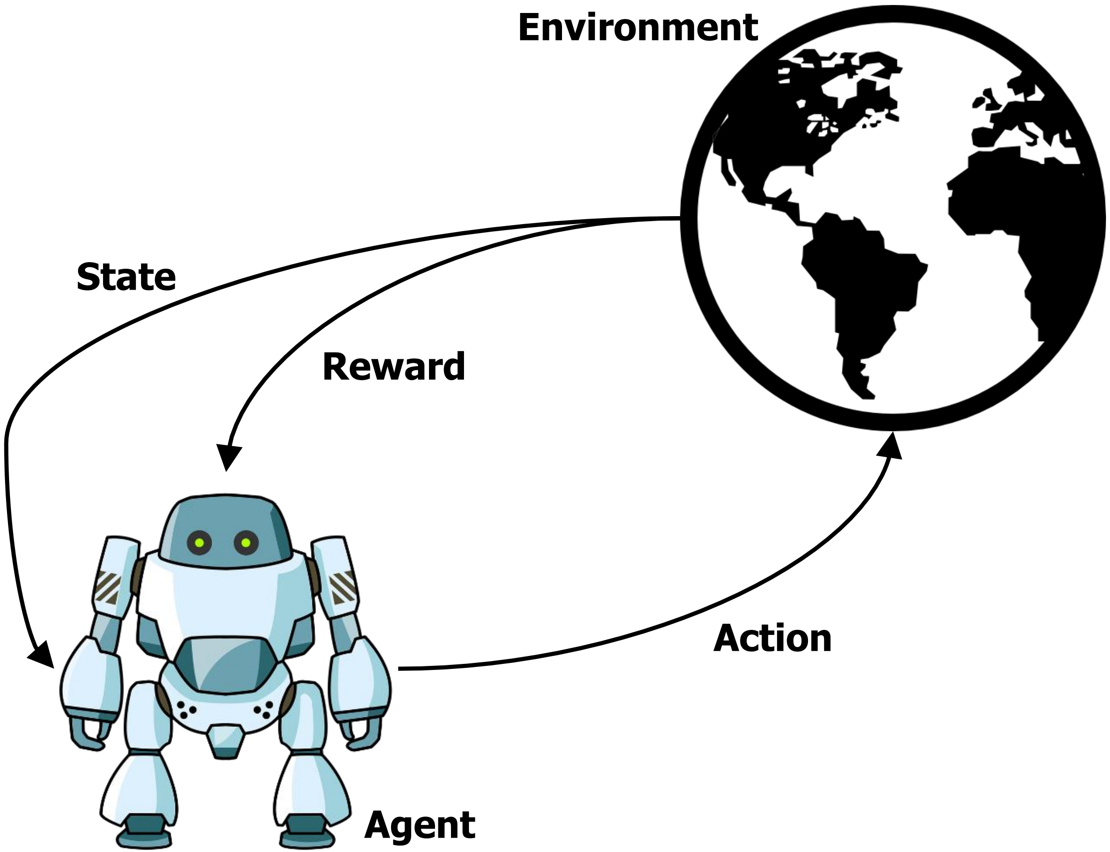
- Brief History and Successes
  - Minsky's PhD thesis (1954): Stochastic Neural-Analog **Reinforcement** Computer
  - Analogies with animal learning and psychology
  - Job-shop scheduling for NASA space missions (Zhang and Dietterich, 1997)
  - Robotic soccer (Stone and Veloso, 1998) – part of the world-champion approach

- When RL can be used?
  - Find the (approximated) **optimal action sequence** for **expected reward maximization (not for single optimal solution)**
  - Define **actions** and **rewards**. These are all we need to do.

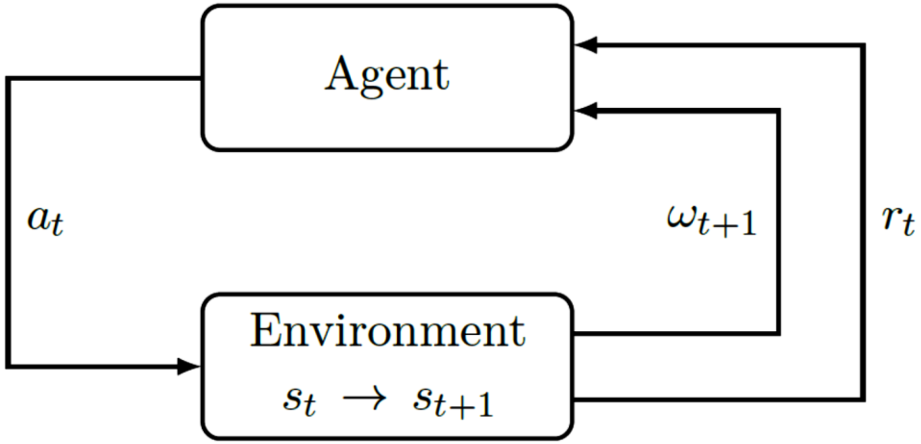
- Action Sequence (also called **Policy**, later in this presentation)!



Introduction to RL



- **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 $\omega_{t+1} \in \Omega$



Introduction and Preliminaries

Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

Inverse Reinforcement Learning and Imitation Learning

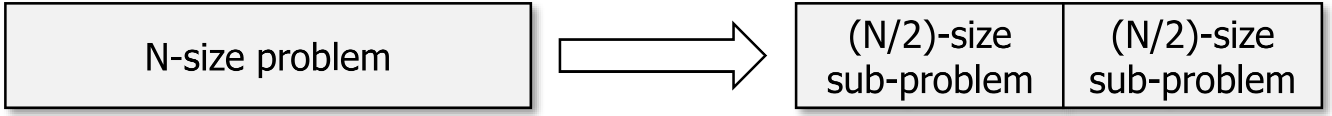
- Introduction and Applications
- **Dynamic Programming**
- Q-Learning and Markov Decision Process

Outline

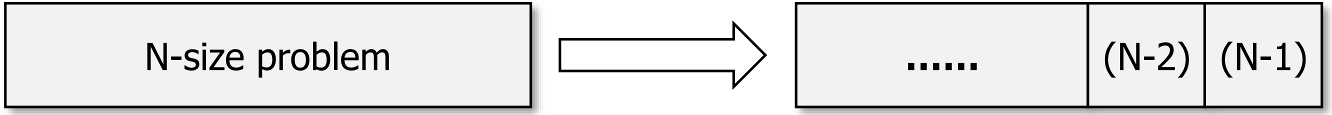
- **<u>Introduction</u>**
- Applications
  - Fibonacci Number
  - Pascal's Triangle
  - Knapsack Problem

Introduction

- 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



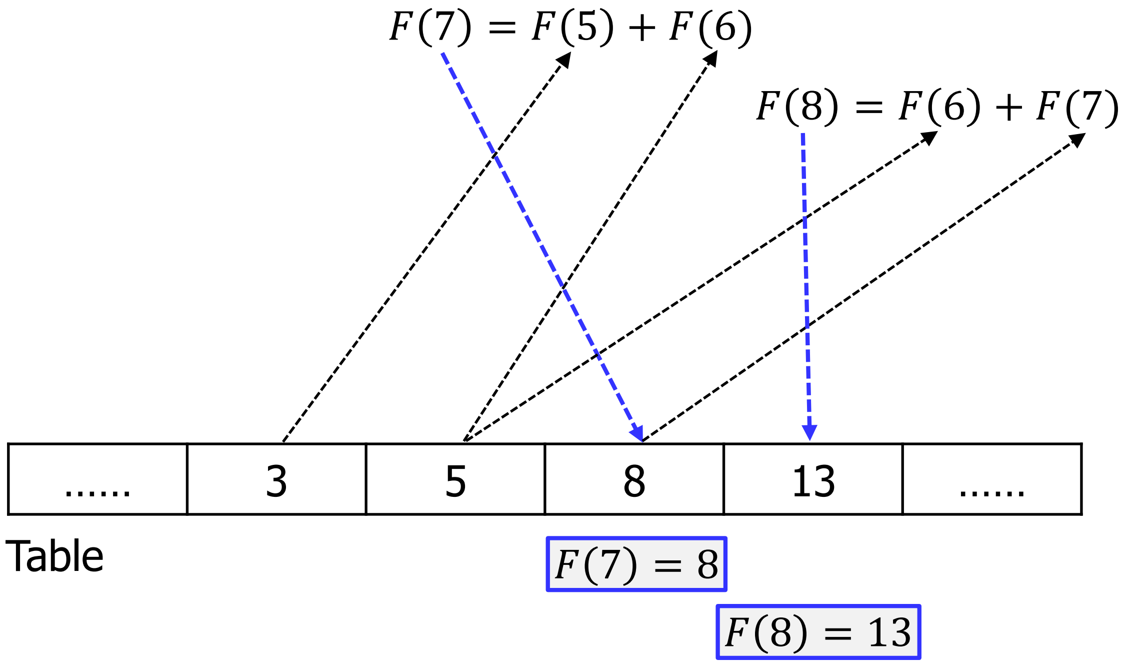
    - Dynamic programming



Outline

- 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!)



Outline

- 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!

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
1 12 66 220 495 792 924 792 495 220 66 12 1
1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1
1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1
1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1
1 16 120 560 1820 4368 8008 11440 12870 11440 8008 4368 1820 560 120 16 1

Outline

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

- Knapsack Problem
  - Suppose that we have N items ($I_i, i \in \{1, \cdots, N\}$) and each item has its own value (v_i for I_i where $i \in \{1, \cdots, N\}$) and weight (w_i for I_i where $i \in \{1, \cdots, N\}$). Now, we want to put items into knapsack where the capacity of knapsack is C. For the item allocation into the knapsack, we want to maximize the summation of values of allocated items.
  - Formulation:

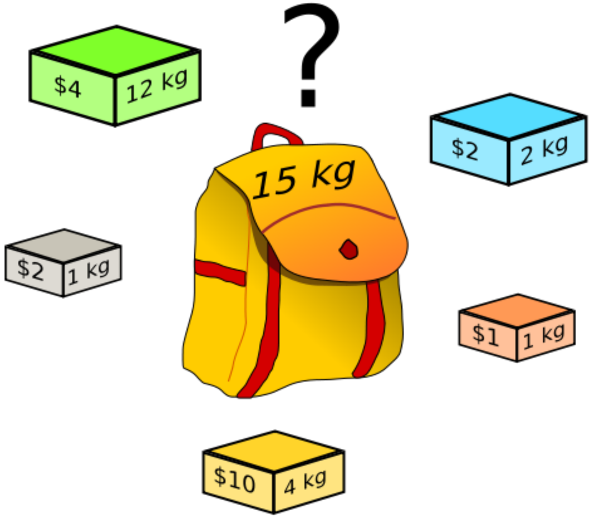
$$max: \sum_{i=1}^{N} v_i \cdot x_i$$

subject to

$$\sum_{i=1}^{N} w_i \cdot x_i \leq C$$

$$x_i \in \{0,1\}, i \in \{1, \cdots, N\}$$

and x_i is decision variable which defines the selection of I_i



- Formulation
  - **OPT(i,w):** Maximum sum value when
    - Items: $1, 2, 3, \cdots, i$
    - Residual capacity in the knapsack: w
  - Two possible cases
    - Case 1) When item i is NOT selected: **OPT(i,w)** $\leftarrow$ **OPT($i-1$,w)**
    - Case 2) When item i is selected: **OPT(i,w)** $\leftarrow$ **OPT($i-1$,$w-w_i$)+v_i**
      - v_i: The value of item i
      - w_i: The weight of item i
      - Note) This holds only when $w_i \leq w$.
  - Final Form

OPT(i,w) $\leftarrow$ **max{OPT($i-1$,w), OPT($i-1$,$w-w_i$)+v_i}**

OPT(i,w) $\leftarrow$ 0 // when $i = 0$

Application: Knapsack Problem

Product	A	B	C	D	E
Weight	3	4	7	8	9
Value	4	5	10	11	13

Product A Only

Capacity	3	4	5	6	7
Value	4	4	4	8	8
Product	A	A	A	AA	AA

Product A and Product B

Capacity	3	4	5	6	7
Value	4	5	5	8	9
Product	A	B	B	AA	AB

Lecture Roadmap

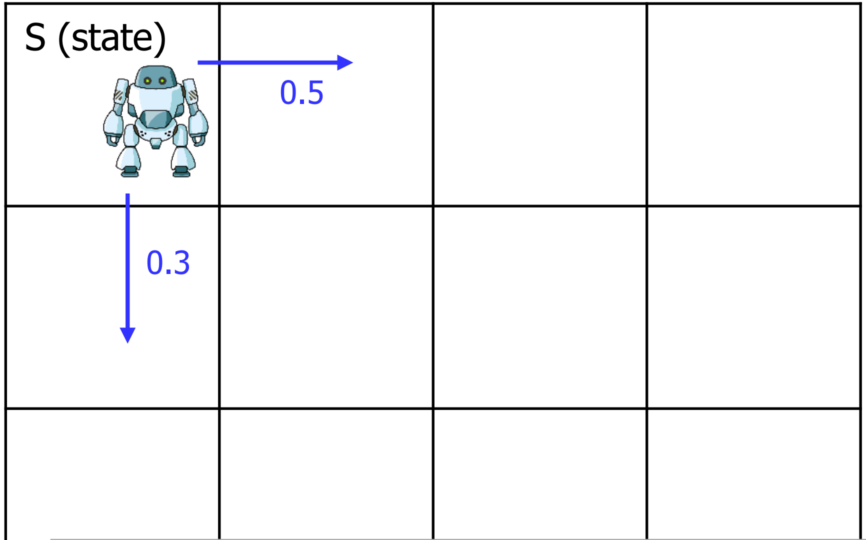
Introduction and Preliminaries

Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

Inverse Reinforcement Learning and Imitation Learning

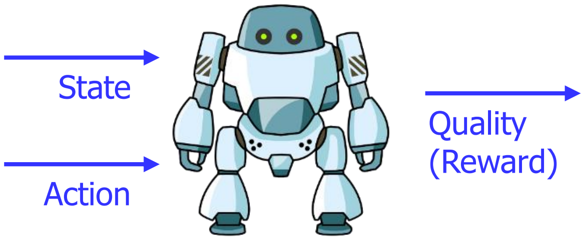
- Introduction and Applications
- Dynamic Programming
- **<u>Q-Learning and Markov Decision Process</u>**



Q(s1, LEFT): 0.0
Q(s1, RIGHT): 0.5 → Maximum
Q(s1, UP): 0.0
Q(s1, DOWN): 0.3

$$\text{RIGHT} \leftarrow arg \max_{a \in A} Q(s_1, a)$$

- Q-Function (State-action value)

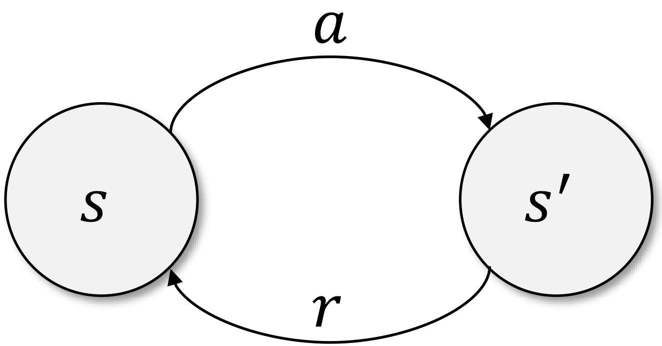


Q (state, action)

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 $Q(s', a')$?

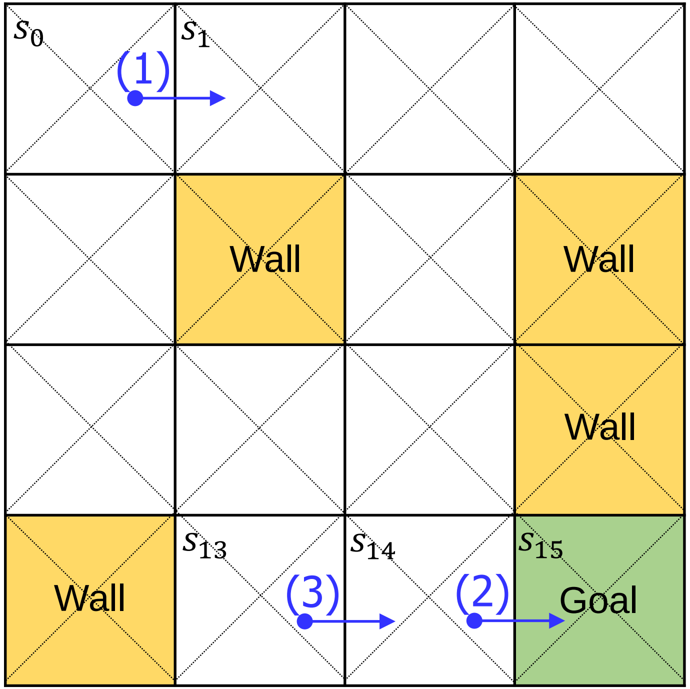
$$Q(s, a) = r + \max_{a'} Q(s', a')$$

Recurrence (e.g., factorial)

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

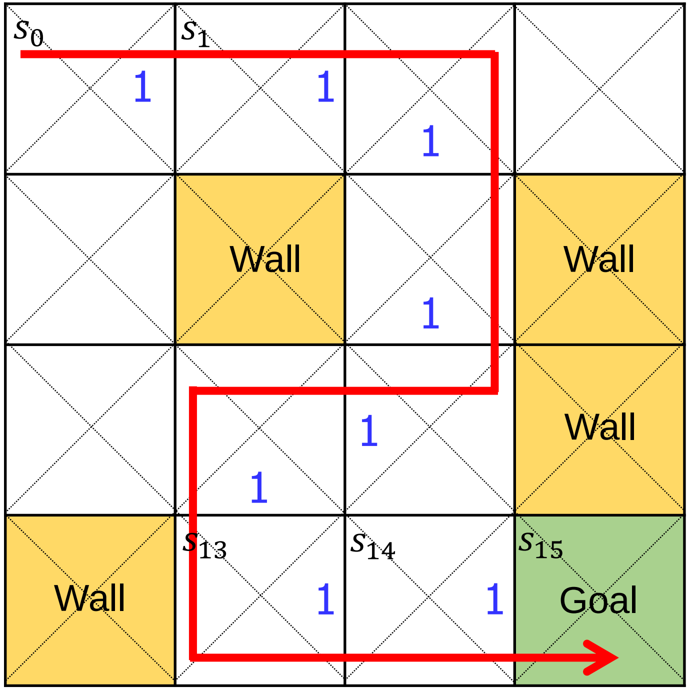
3! = F(3) = 3 * F(2)
= 3 * 2 * F(1)
= 3 * 2* 1 = 6

- 16 states and 4 actions (U, D, L, R)

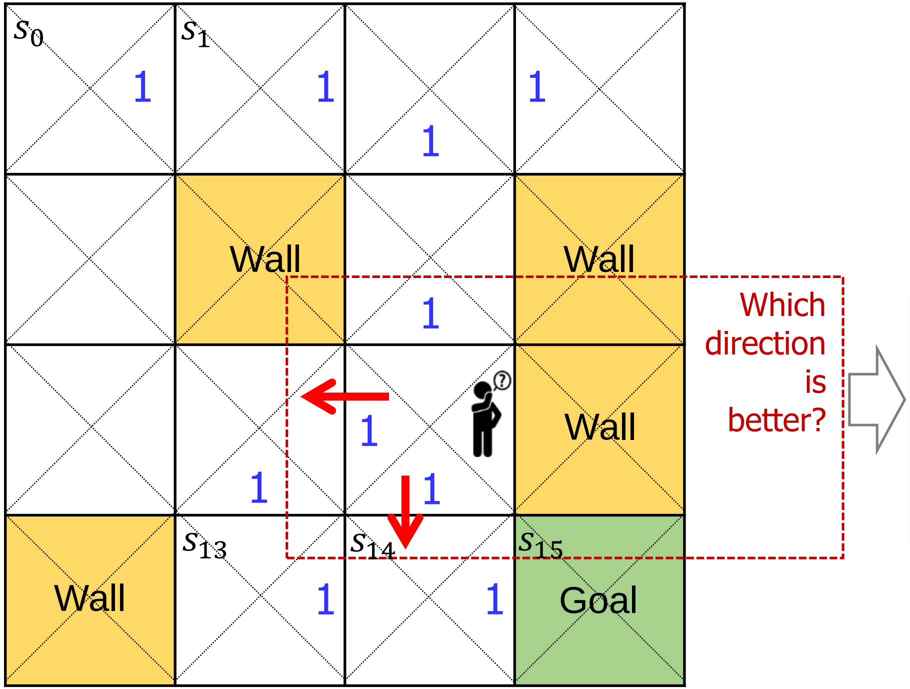


- Initial Status
  - All 64 Q values are 0,
  - Reward are all zero except $r_{s_{15},L} = 1$
- For (1), from s_0 to s_1
  - $Q(s_0, a_R) = r + \max_a Q(s_1, a) = 0 + \max\{0,0,0,0\} = 0$
- For (2), from s_{14} to s_{15} (goal)
  - $Q(s_{14}, a_R) = r + \max_a Q(s_{15}, a) = 1 + \max\{0,0,0,0\} = 1$
- For (3), from s_{13} to s_{14}
  - $Q(s_{13}, a_R) = r + \max_a Q(s_{14}, a) = 0 + \max\{0,0,1,0\} = 1$

- 16 states and 4 actions (U, D, L, R)



- 16 states and 4 actions (U, D, L, R)



Learning $Q(s,a)$ with Discounted Reward

$$Q(s,a) = r + \boxed{\gamma} \cdot arg \max_a Q(s',a')$$

$$0 < \gamma \leq 1$$

- For each s, a, initialize table entry $Q(s,a) \leftarrow 0$
- Observe current state s
- Do forever
  - Select an action a and execute it
  - Receive immediate reward r
  - Observe the new state s'
  - Update the table entry for $Q(s,a)$ as follows:

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

  - $s \leftarrow s'$

Q-Learning with Exploit and Exploration: ε-Greedy

Finding the Best Restaurant

- Try the best one during weekdays.
- Try new ones during weekends.

ε-Greedy

```
e=0.1
IF (random < e)
        a = random;
ELSE
        a = argmax(Q(s,a));
```

Decaying ε-Greedy

```
for i in range (1000); e=0.1 / (i+1);
IF (random < e)
        a = random;
ELSE
        a = argmax(Q(s,a));
```

Q-Learning with Exploit and Exploration: Add Random Noise

Finding the Best Restaurant

McDonald's	KFC	Burger King	Lotteria
43	35	41	31

Add Random Noise

52	**56**	53	32

Add Random Noise

a = argmax(Q(s,a) + **random_values**);

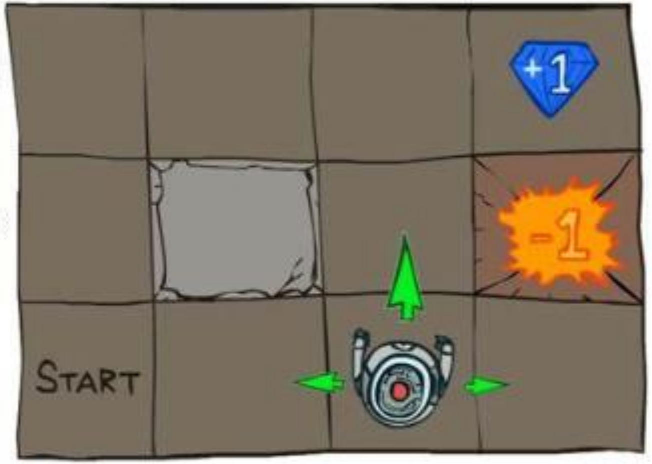
Add Decaying Random Noise

for i in range (1000);
a = argmax(Q(s,a) + random**/(i+1)**);

- 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

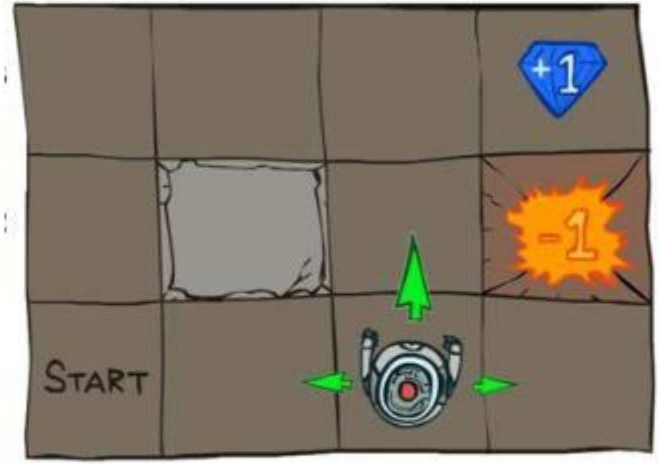
How can we use MDP to model agent in a maze?

- 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, γ>
  - S: Set of states
  - **A: Set of actions**
  - R: Reward function
  - T: Transition function
  - γ: Discount factor

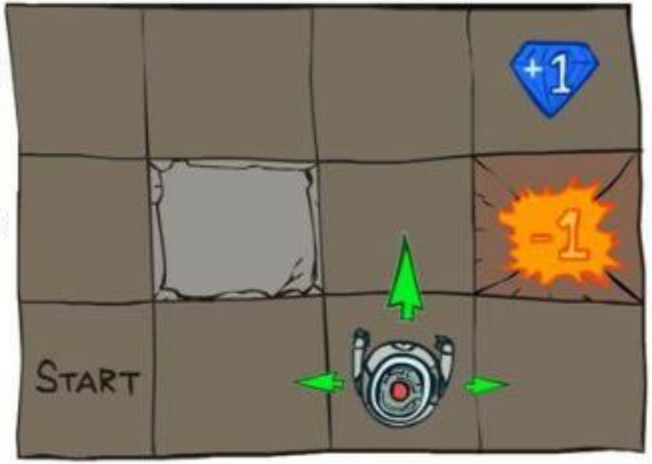


S: location (x, y) if the maze is a 2D grid

A: move up, down, left, or right

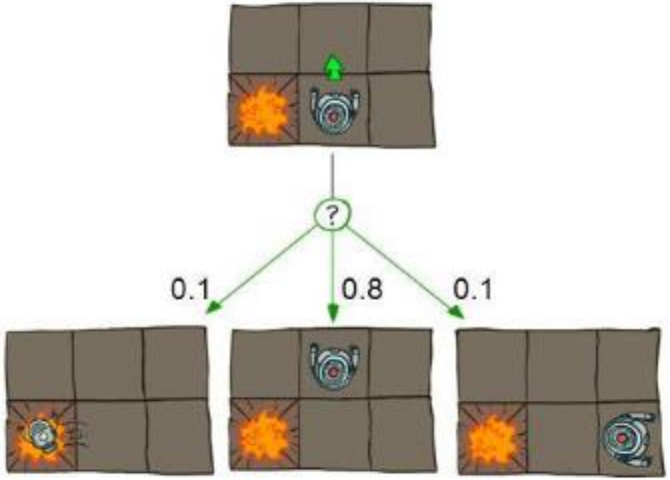
- $s \rightarrow s'$

- 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

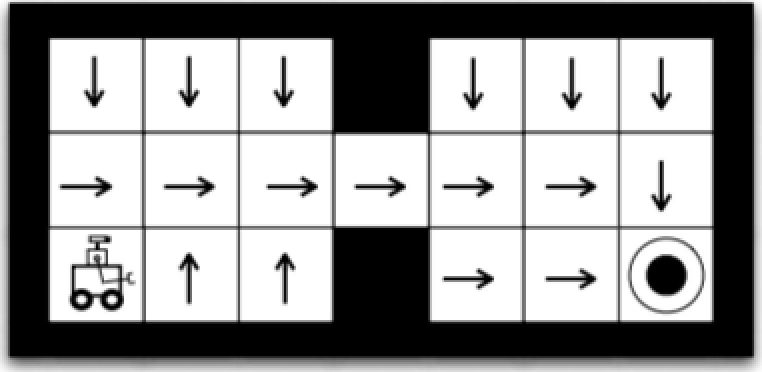
S: location (x, y) if the maze is a 2D grid
A: move up, down, left, or right
R: how good was the chosen action?
T: where is the robot's new location?
- $T = P(s'|s, a)$

- 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)]

Markov Decision Process (MDP)

- 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!

- State Value Function (V)

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

  - Expected return of starting at state s and following policy π
  - How much return do I expect starting from state s?

- 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

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

  - 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.

- Notes
  - Bellman Equation (Value Iteration)

$$\boxed{\forall s \in S\colon V^*(s) = \max_{a} \sum_{s'} \{R(s,a,s') \cdot T(s,a,s') + \gamma V^*(s')\}}$$

- **Intro to Reinforcement Learning**
  - **Prologue**
  - Formal Framework

arXiv:1811.12560v2 [cs.LG] 3 Dec 2018

An Introduction to Deep Reinforcement Learning

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare and Joelle Pineau (2018), "An Introduction to Deep Reinforcement Learning", Foundations and Trends in Machine Learning: Vol. 11, No. 3-4. DOI: 10.1561/2200000071.

Vincent François-Lavet
McGill University
vincent.francois-lavet@mcgill.ca

Peter Henderson
McGill University
peter.henderson@mail.mcgill.ca

Riashat Islam
McGill University
riashat.islam@mail.mcgill.ca

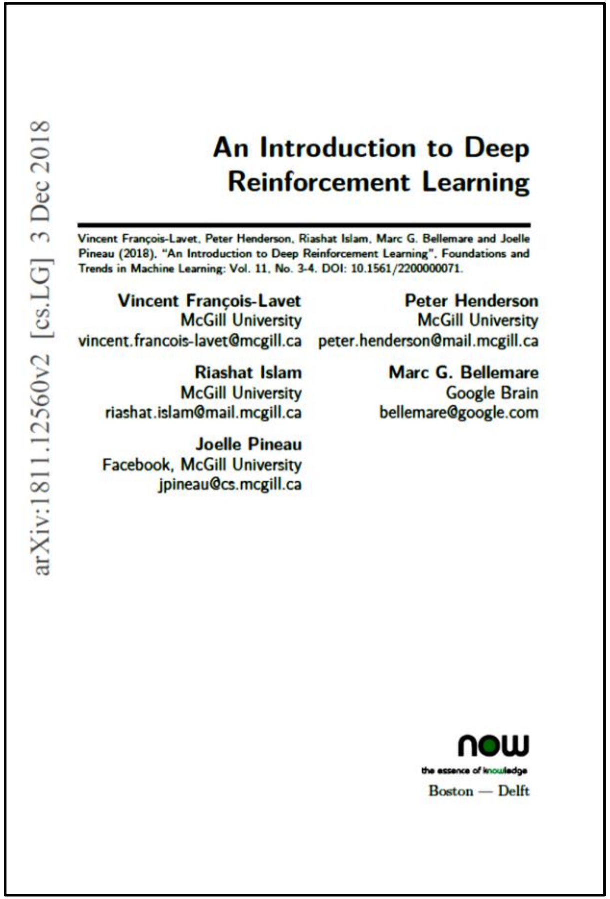
Marc G. Bellemare
Google Brain
bellemare@google.com

Joelle Pineau
Facebook, McGill University
jpineau@cs.mcgill.ca

now
the essence of knowledge
Boston — Delft

- Reinforcement Learning (RL)
  - RL is the area of machine learning that deals with **sequential decision-making**.
  - RL problem can be formalized as an **agent** that has to make decisions in an environment to **optimize a given notion of cumulative rewards**.
  - Key aspects of RL
    - An agent **learns** a good behavior.
      - It modifies or acquires new behaviors and skills incrementally.
    - It uses trial-and-error **experience**.
      - An RL agent does not require complete knowledge or control of the environment.
      - It only needs to be able to interact with the environment and collect information.

- **Intro to Reinforcement Learning**
  - Prologue
  - **Formal Framework**



arXiv:1811.12560v2 [cs.LG] 3 Dec 2018

An Introduction to Deep Reinforcement Learning

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare and Joelle Pineau (2018), "An Introduction to Deep Reinforcement Learning", Foundations and Trends in Machine Learning: Vol. 11, No. 3-4. DOI: 10.1561/2200000071.

Vincent François-Lavet
McGill University
vincent.francois-lavet@mcgill.ca

Peter Henderson
McGill University
peter.henderson@mail.mcgill.ca

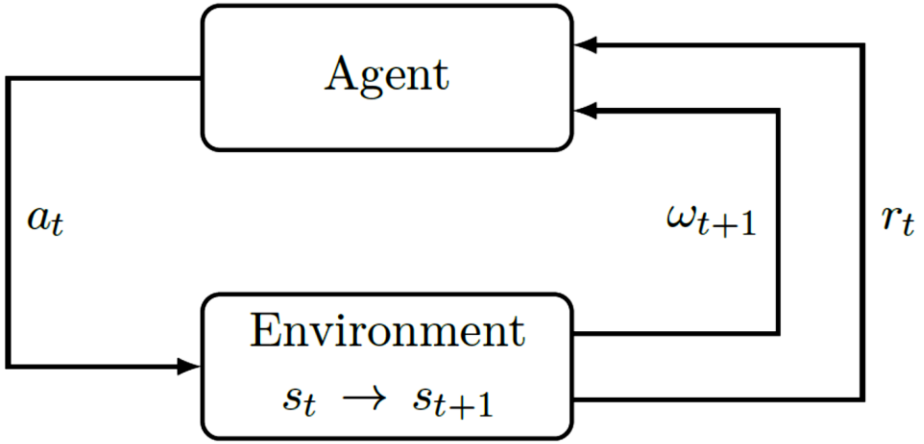
Riashat Islam
McGill University
riashat.islam@mail.mcgill.ca

Marc G. Bellemare
Google Brain
bellemare@google.com

Joelle Pineau
Facebook, McGill University
jpineau@cs.mcgill.ca

now
the essence of knowledge
Boston — Delft

- **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 $\omega_{t+1} \in \Omega$

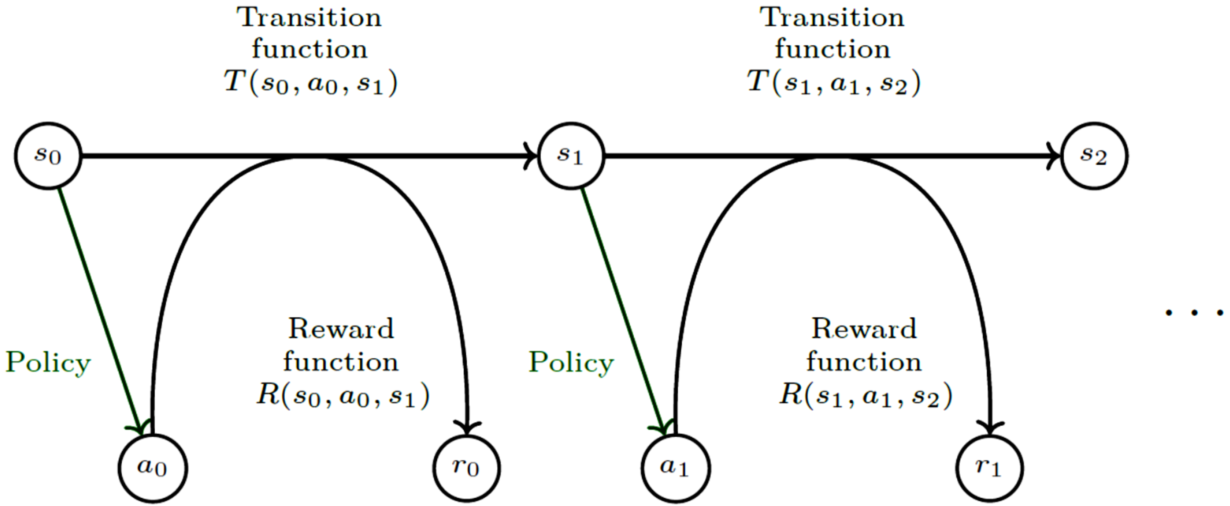


- **Markov Property**
  - [Definition (Markovian)] A discrete time stochastic control process is Markovian (i.e., it has the Markov property) if
    - $P(\omega_{t+1}|\omega_t, a_t) = P(\omega_{t+1}|\omega_t, a_t, \cdots, \omega_0, a_0)$, and
    - $P(r_t|\omega_t, a_t) = P(r_t|\omega_t, a_t, \cdots, \omega_0, a_0)$

  - 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 $R(s_t, a_t, s_{t+1}) \in R$.



- **Different Categories of Policies**
    - A policy defines how an agent selects actions.
    - Policies can also be categorized under a second criterion of being either deterministic or stochastic:
        - In the deterministic case, the policy is described by $\pi(s): S \rightarrow A$.
        - In the stochastic case, the policy is described by $\pi(s, a): S \times A \rightarrow [0,1]$ where $\pi(s, a)$ denotes the probability that action a may be chosen in state s.

- **Expected Return**
  - Basic Assumption: Consider the case of an RL agent whose goal is to find a policy $\pi(s, a) \in \Pi$, so as to optimize an **expected return** $V^{\pi}(s): S \rightarrow R$ (also called V-value function) such that

$$V^{\pi}(s) = E\left[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k} \,|s_t = s, \pi\right]$$

    where
    - $r_t = E_{a \sim \pi(s_t, \cdot)}\{R(s_t, a, s_{t+1})\}$
    - $P(s_{t+1}|s_t, a_t) = T(s_t, a_t, s_{t+1})$ with $a \sim \pi(s_t, \cdot)$

  - From the definition of the expected return, the optimal expected return is as:

$$V^{*}(s) = \max_{\pi \in \Pi} V^{\pi}(s)$$

- **Expected Return**
  - In addition, the Q-value function $Q^{\pi}(s,a): S \times A \rightarrow R$ is defined as follows:

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

  - This can be rewritten recursively in the case of an MDP using Bellman's equation:

$$Q^{\pi}(s,a) = \sum_{s\prime \in S} T(s,a,s')\{R(s,a,s') + \gamma Q^{\pi}(s', a = \pi(s'))\}$$

  - Similar to the V-value function, the optimal Q-value function $Q^{*}(s,a)$ is as:

$$Q^{*}(s,a) = \max_{\pi \in \Pi} Q^{\pi}(s,a)$$

- **Expected Return**
  - The **optimal policy** can be obtained directly from $Q^*(s,a) = \max_{\pi \in \Pi} Q^\pi(s,a)$:

$$\boxed{\pi^*(s) = arg \max_{a \in A} Q^*(s,a)}$$

Introduction and Preliminaries

Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

Inverse Reinforcement Learning and Imitation Learning

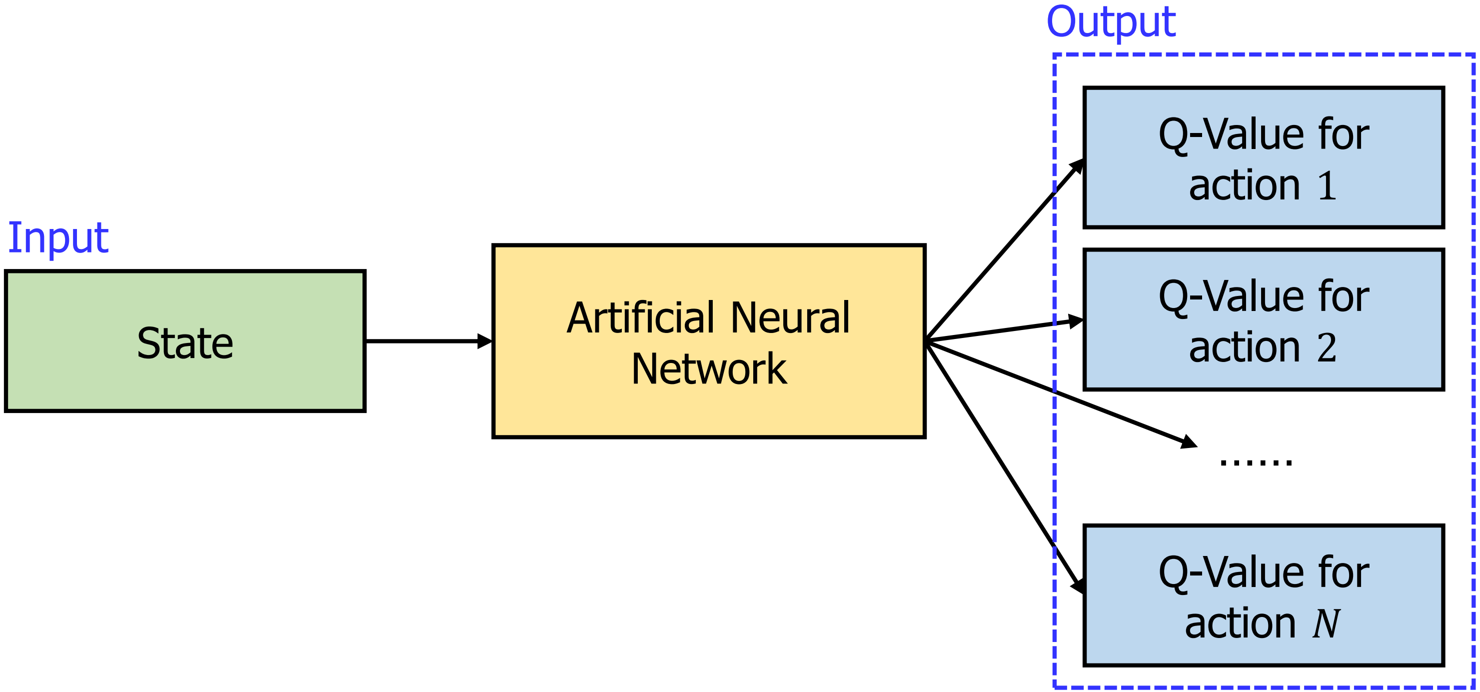
- **Introduction and Motivation**
- Deep Neural Network Summary
- Deep Q-Network (DQN)
- Performance Improvement on DQN

- Small-Scale Q-Values

$$
Q=\begin{array}{c} \\ \text{State}\left\{\begin{array}{c}0\\1\\2\\3\\4\\5\end{array}\right.\end{array}
\begin{array}{c}
\overbrace{\begin{array}{cccccc}0&1&2&3&4&5\end{array}}^{\text{Action}}\\
\begin{bmatrix}
0&0&0&0&0&0\\
0&0&0&0&0&100\\
0&0&0&0&0&0\\
0&80&0&0&0&0\\
0&0&0&0&0&0\\
0&0&0&0&0&0
\end{bmatrix}
\end{array}
\Rightarrow
Q=\begin{array}{c} \\ \text{State}\left\{\begin{array}{c}0\\1\\2\\3\\4\\5\end{array}\right.\end{array}
\begin{array}{c}
\overbrace{\begin{array}{cccccc}0&1&2&3&4&5\end{array}}^{\text{Action}}\\
\begin{bmatrix}
0&0&0&0&80&0\\
0&0&0&64&0&100\\
0&0&0&64&0&0\\
0&80&51&0&80&0\\
64&0&0&64&0&100\\
0&80&0&0&80&100
\end{bmatrix}
\end{array}
$$

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**.



Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

- Introduction and Motivation
- **Deep Neural Network Summary**
- Deep Q-Network (DQN)
- Performance Improvement on DQN

Deep Reinforcement Learning Implementation

DDPG-based Vehicular Caching

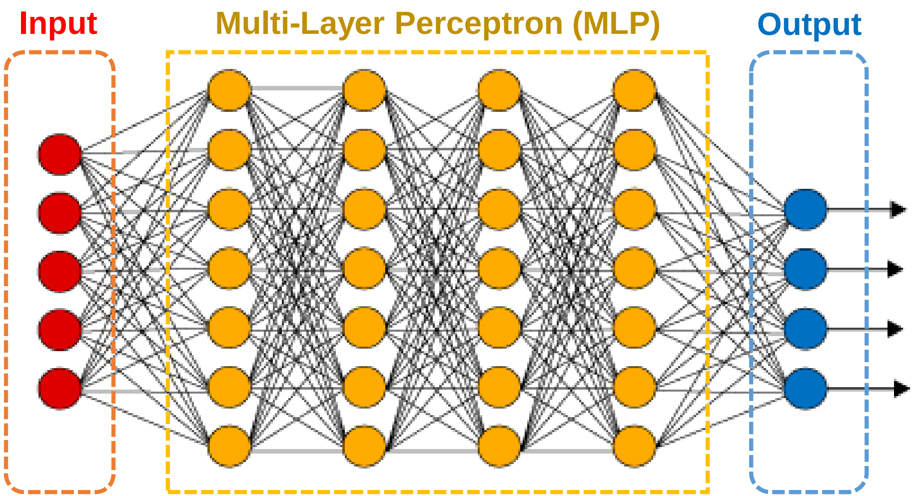
Imitation Learning and Autonomous Driving

- How Deep Learning Works?
  - Deep Learning Computation Procedure

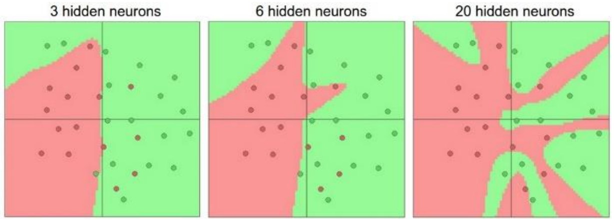
Deep Learning Model Setup
- MLP, CNN, RNN, GAN, or Customized
- # Hidden Layers, # Units, Input/Output, …
- Cost Function / Optimizer Selection

Training (with Large-Scale Dataset)
- Input: Data, Output: Labels
- Learning → Weights Updates for Cost Function Minimization

Inference / Testing (Real-Word Execution)
- Input: Real-World Input Data
- Output: Interference Results based on Updated Weights in Deep Neural Networks



Non-Linear Training (Weights Updates) for Cost Minimization: GD, SGD, Adam, etc.



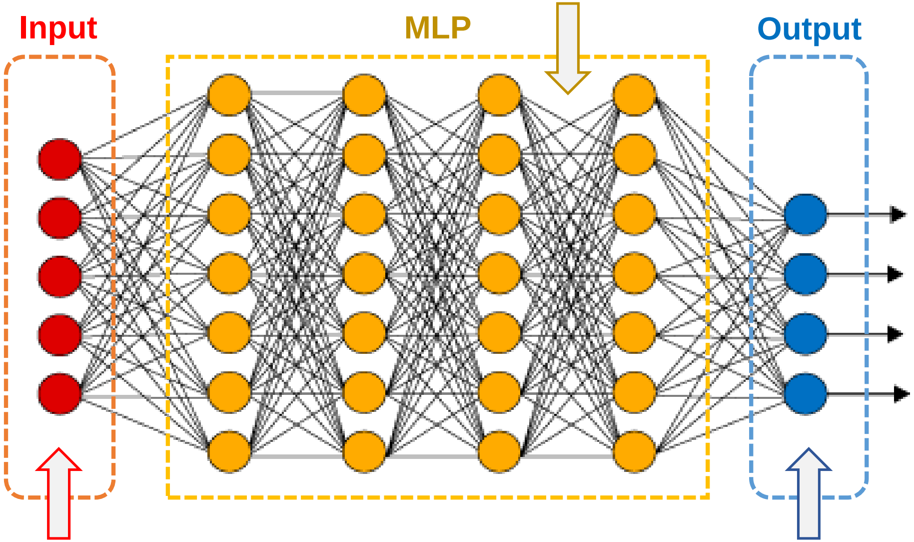
- How Deep Learning Works?
  - Deep Learning Computation Procedure

Deep Learning Model Setup
- MLP, CNN, RNN, GAN, or Customized
- # Hidden Layers, # Units, Input/Output, …
- Cost Function / Optimizer Selection

Training (with Large-Scale Dataset)
- Input: Data, Output: Labels
- Learning → Weights Updates for Cost Function Minimization

Inference / Testing (Real-Word Execution)
- Input: Real-World Input Data
- Output: Interference Results based on Updated Weights in Deep Neural Networks

All weights in units are trained/set (under cost minimization)



INPUT: Data
- One-Dimension Vector

OUTPUT: Labels
- One-Hot Encoding

We need a lot of training data for generality (otherwise, we will suffer from overfitting problem).

Introduction

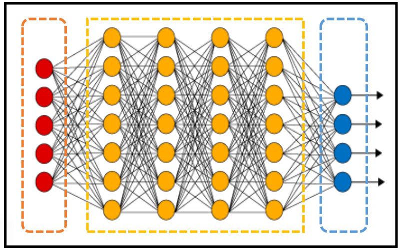
- How Deep Learning Works?
  - Deep Learning Computation Procedure

Deep Learning Model Setup
- MLP, CNN, RNN, GAN, or Customized
- # Hidden Layers, # Units, Input/Output, …
- Cost Function / Optimizer Selection

Training (with Large-Scale Dataset)
- Input: Data, Output: Labels
- Learning → Weights Updates for Cost Function Minimization

Inference / Testing (Real-Word Execution)
- Input: Real-World Input Data
- Output: Interference Results based on Updated Weights in Deep Neural Networks

Trained Model



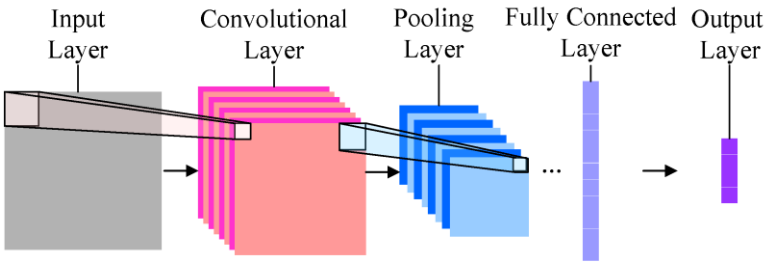
Intelligent Surveillance Platforms

INPUT: Real-Time Arrivals

OUTPUT: Inference
- Computation Results based on (i) INPUT and (ii) trained weights in units (trained model).

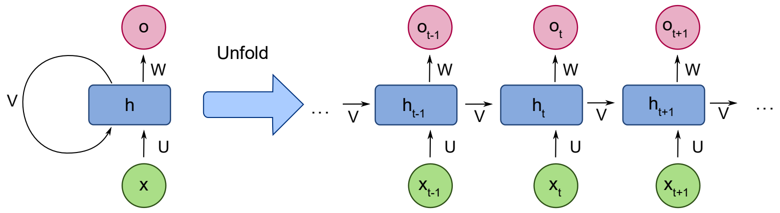
- Two Major Deep Learning Models → CNN vs. RNN

Convolutional Neural Network (CNN)



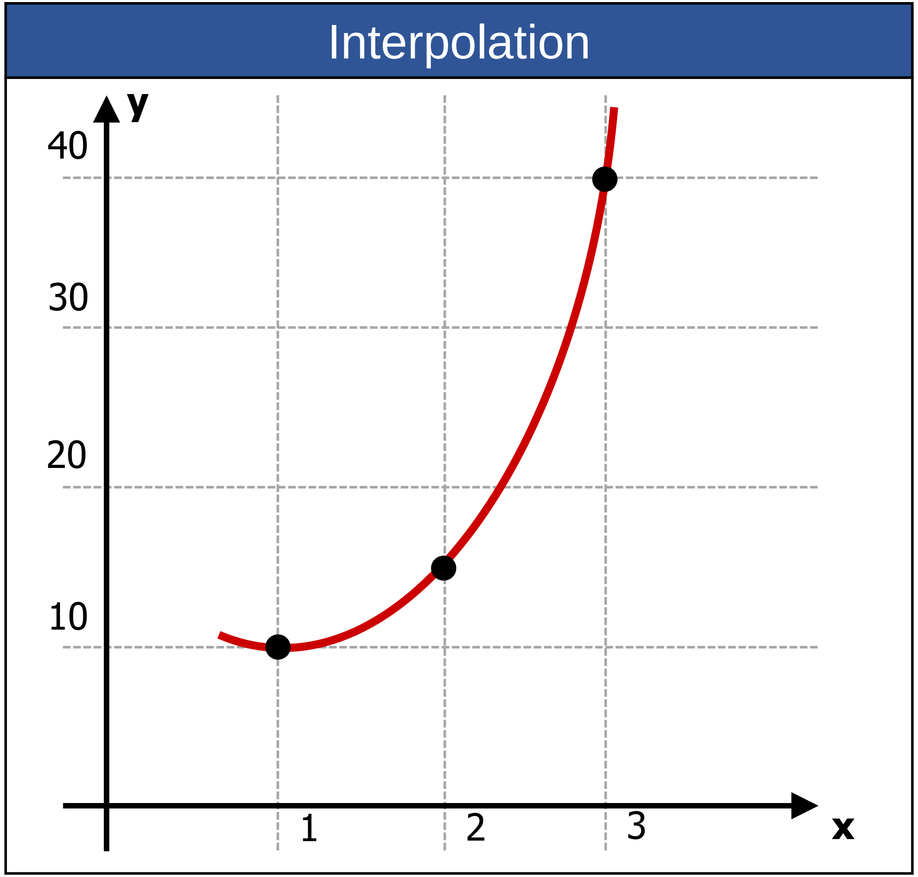
- In conventional neural network architectures, the input should be one-dimensional vector.
- In many applications, the input should be multi-dimensional (e.g., 2D for images). Thus, we need architectures in order to recognize the features in high-dimensional data.
- Mainly used for **visual information learning**

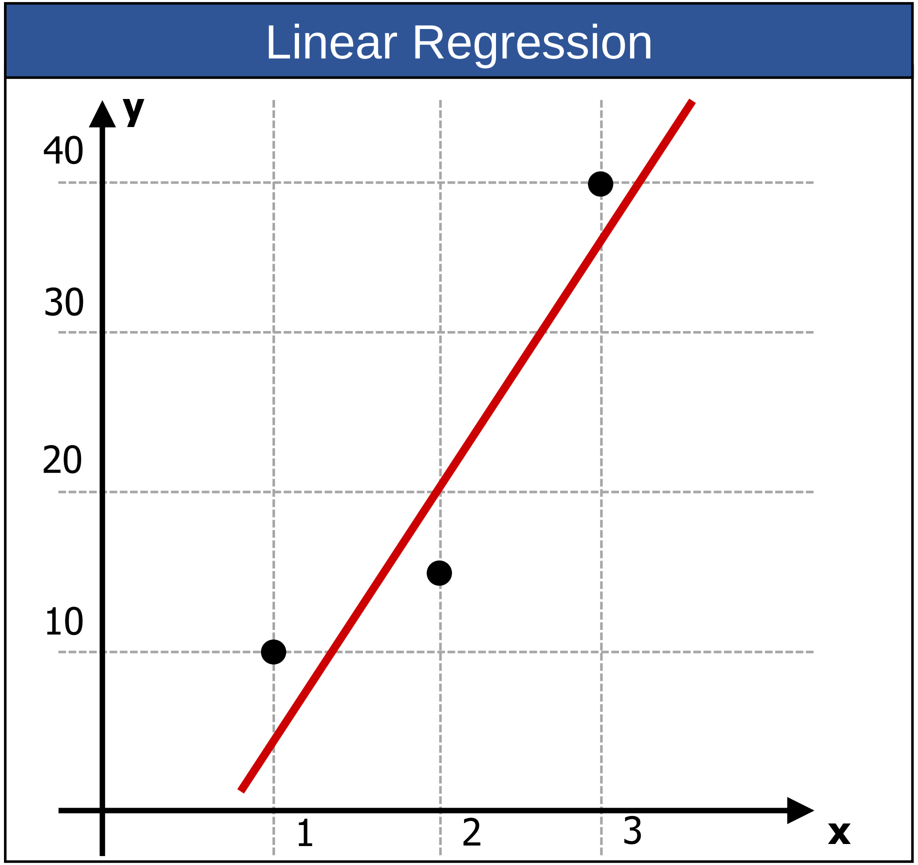
Recurrent Neural Network (RNN)



- In conventional neural network architectures, there is no way to introduce the concept of time.
- The time index can be represented as the chain of neural network models.
- The representative models are LSTM and GRU.
- Mainly used for **time-series information learning**

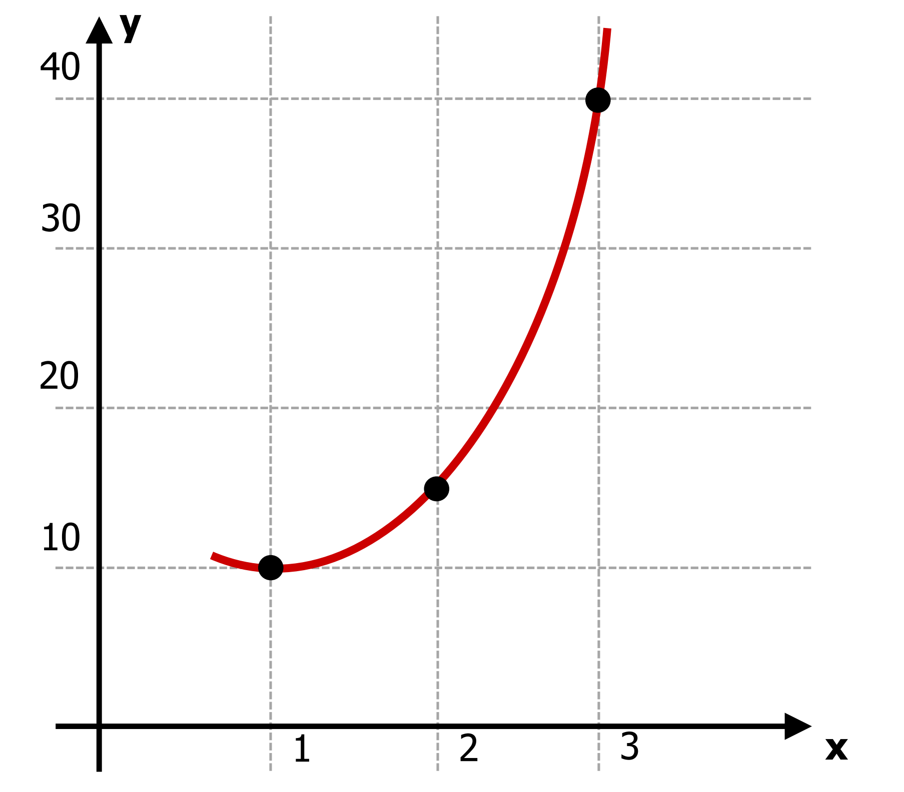
Interpolation vs. Linear Regression





Interpolation vs. Linear Regression

Interpolation

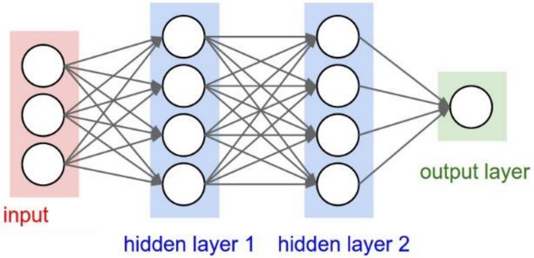


Interpolation with Polynomials

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

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

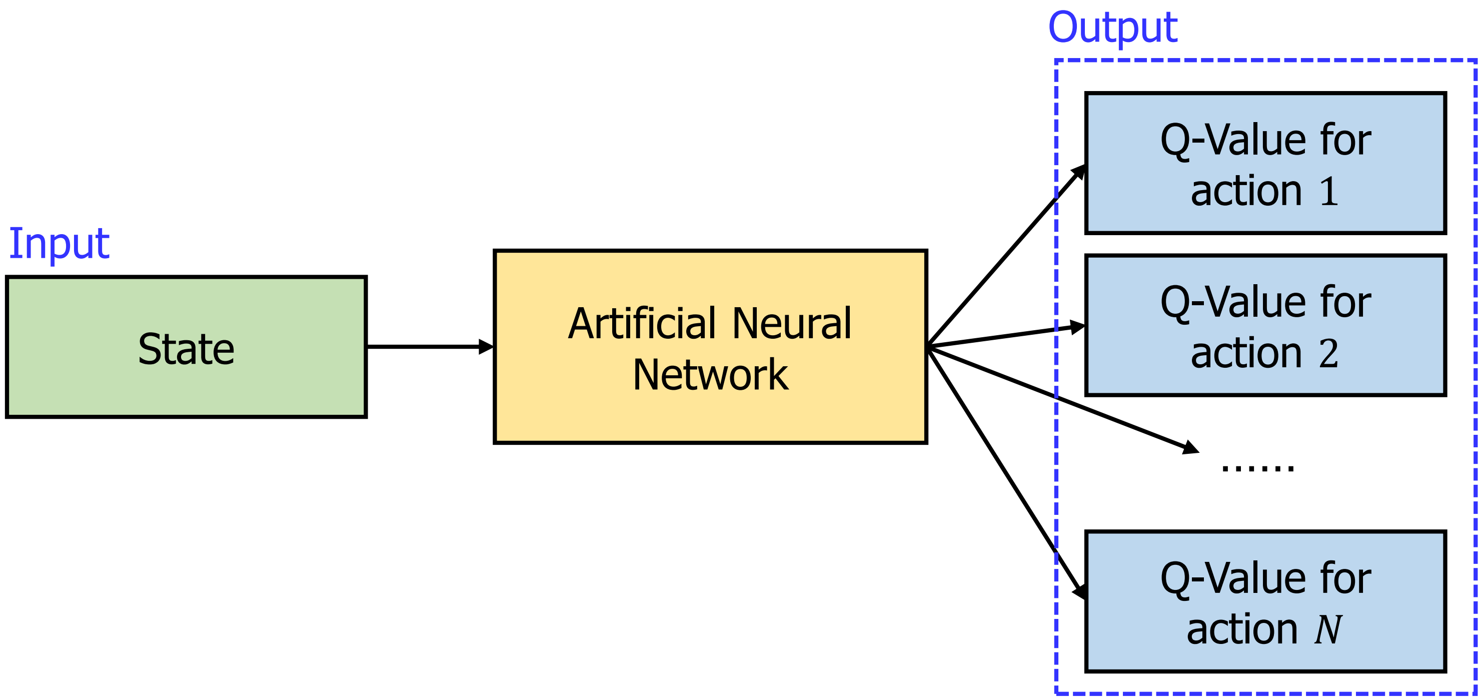
Is this related to **Neural Network Training**?



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

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**

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



Lecture Roadmap

Introduction and Preliminaries

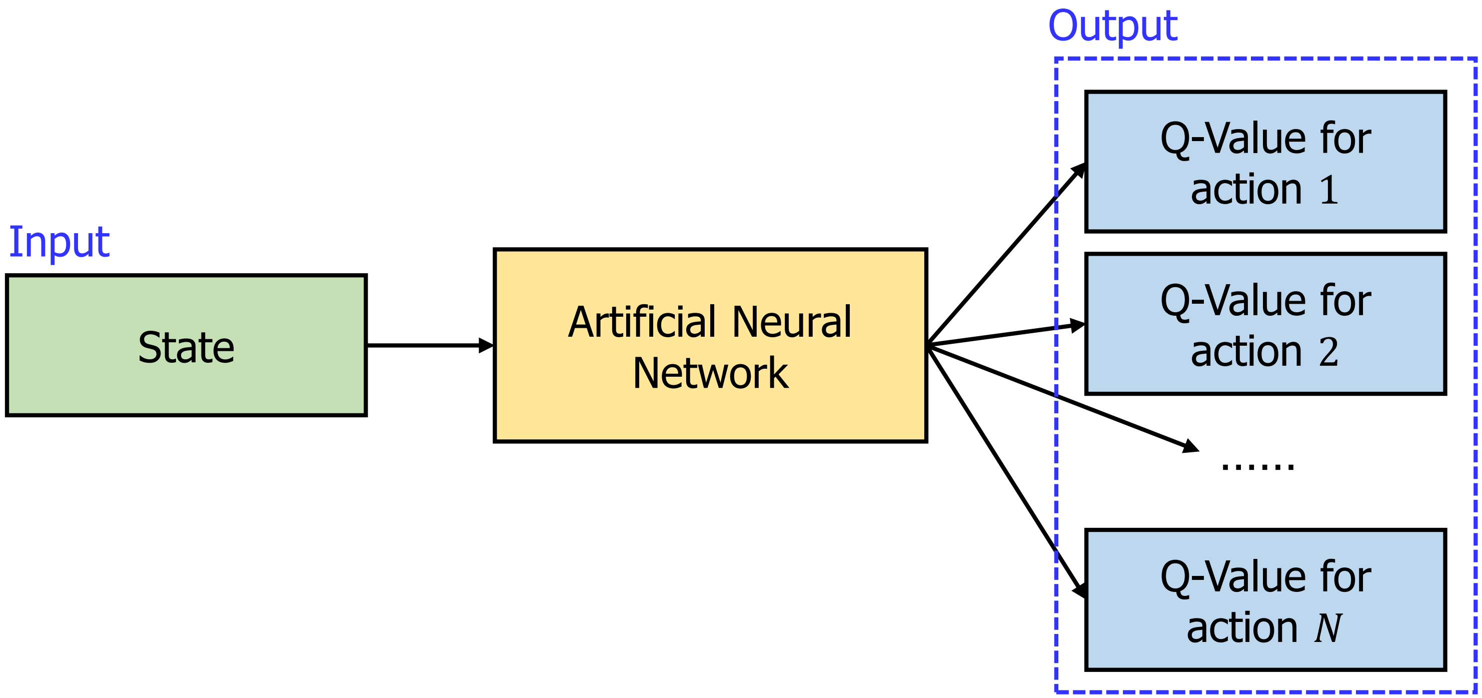
Deep Reinforcement Learning Theory

- Introduction and Motivation
- Deep Neural Network Summary
- **<u>Deep Q-Network (DQN)</u>**
- Performance Improvement on DQN

Deep Reinforcement Learning Implementation

Inverse Reinforcement Learning and Imitation Learning

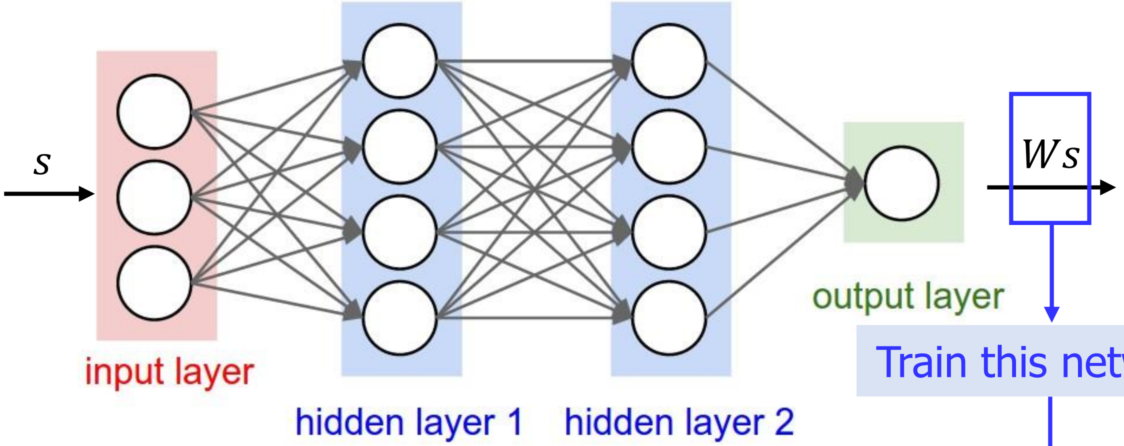
- Large-Scale Q-Values
  - It is inefficient to make the Q-table for each state-action pair.
    → ANN is used to **approximate the Q-function**.



- Q-Network Training (Linear Regression)

H(x)=Wx

Cost(W)= $\frac{1}{m}\sum_{i=1}^{m}\left(Wx^{i} - y^{i}\right)^{2}$



Train this network to approximate optimal Q, i.e., Q^*

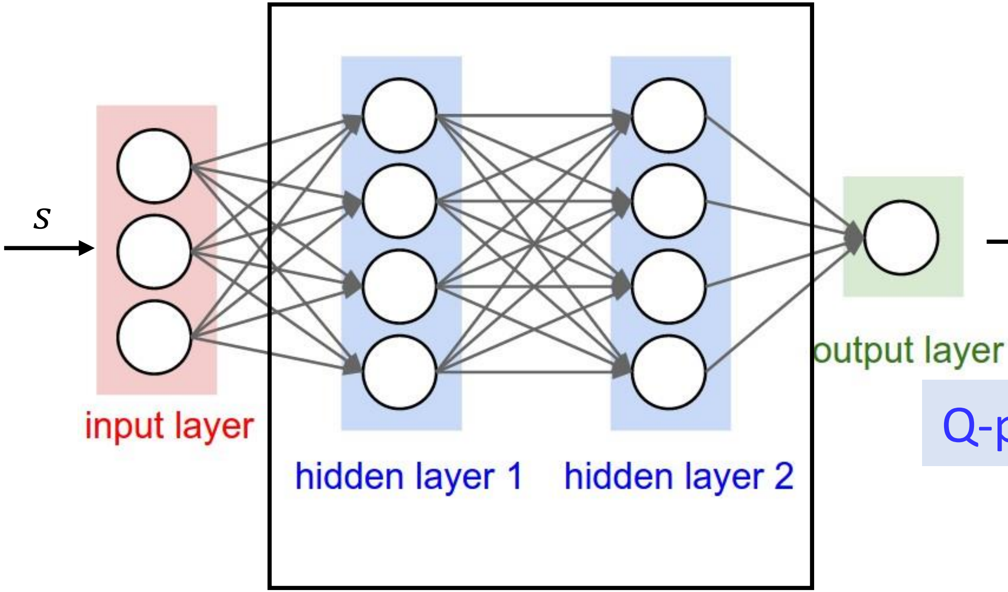
$$Ws \approx Q^*$$

- Q-Network Training (Linear Regression)

$$\text{Cost}(W) = (Ws - y)^2$$

$$y = r + \gamma \max Q(s')$$

Q^*



Ws

Q-predict: $\hat{Q}$

Weights, θ

Approximating Q^* using θ

$$Ws = \hat{Q}(s, a|\theta) \approx Q^*(s, a)$$

- Q-Network Training (Linear Regression)

$$\boxed{Ws = \hat{Q}(s,a|\theta) \approx Q^*(s,a)}$$

Approximating Q^* using θ

$$\min_{\theta} \sum_{t=0}^{T} \Big[\underbrace{\hat{Q}(s_t, a_t|\theta)}_{\hat{Q}(s,a|\theta)} - \underbrace{\Big(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'|\theta)\Big)}_{Q^*}\Big]^2$$

Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

- Introduction and Motivation
- Deep Neural Network Summary
- Deep Q-Network (DQN)
- **Performance Improvement on DQN**

Deep Reinforcement Learning Implementation

Inverse Reinforcement Learning and Imitation Learning

Algorithm 1 Deep Q-learning

Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

If preprocessing is not needed, $\emptyset(s) = s$

ε-greedy

Learning

Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for

Play Atari with Deep Reinforcement Learning

Q-Network

y-label

$$\text{Set } \boxed{y_j} = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$$

Perform a gradient descent step on $\boxed{\left(y_j - Q(\phi_j, a_j; \theta)\right)^2}$ according to equation 3

Cost function

Target/Label $\rightarrow$ Q^*

Ws $\rightarrow$ $\hat{Q}(s, a|\theta)$

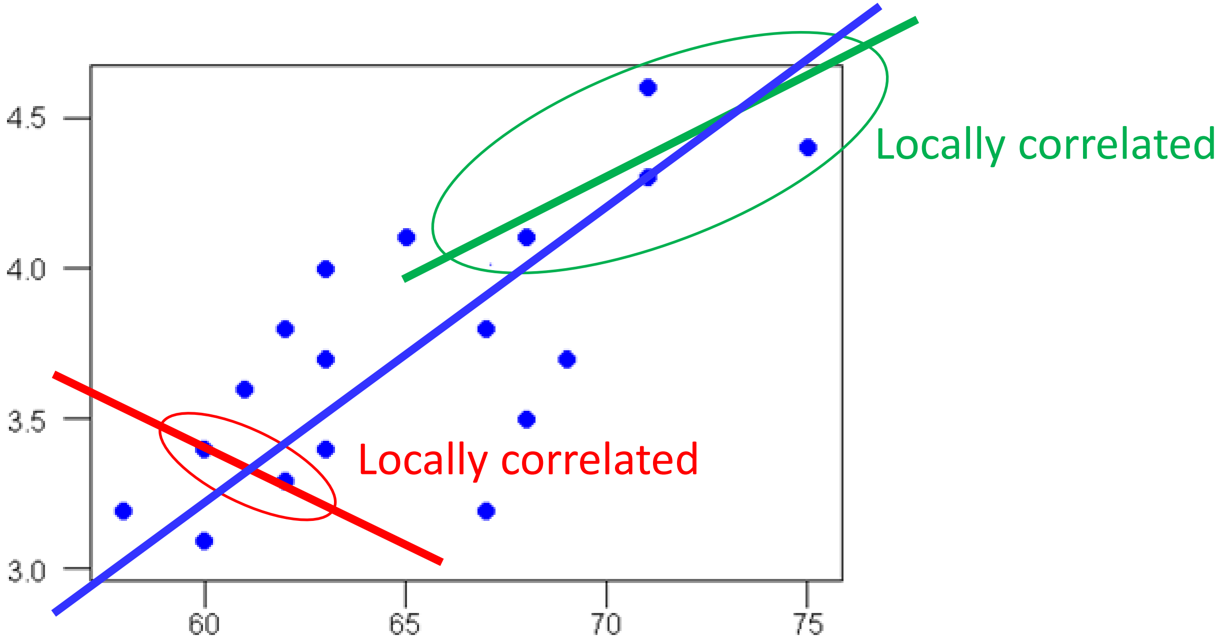
Deep Q-Network (DQN)

$$\min_{\theta} \sum_{t=0}^{T} \left[\hat{Q}(s_t, a_t|\theta) - \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'|\theta) \right) \right]^2$$

- Converges to Q^* using table lookup representation
- However, **diverges** using neural networks due to
  - Correlations between samples → [Issue #1]
  - Non-stationary targets → [Issue #2]

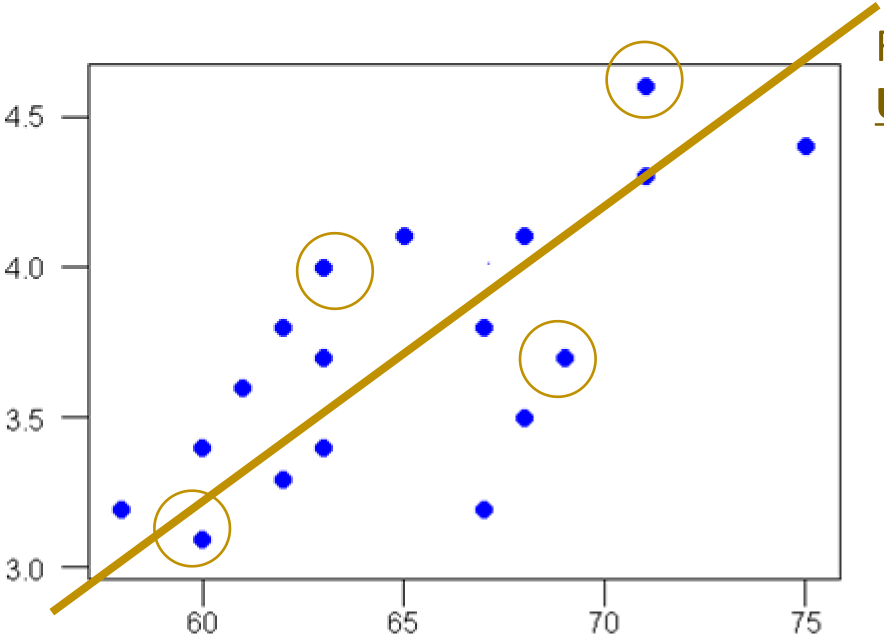
Tutorial by Google DeepMind: Deep Reinforcement Learning

- [Issue #1] Correlations between Samples



- **Solution) Capture and Replay**
    - Store learning states in buffers → random sampling and learning

- [Issue #1] Correlations between Samples
  - **Capture and Replay →** Experience Replay
    - Store learning states in buffers → random sampling and learning



Random Sampling Results are **Uniformed Distributed**.

- [Issue #2] Non-Stationary Targets

Target

$$\min_{\theta} \sum_{t=0}^{T} \left[\hat{Q}(s_t, a_t | \theta) - \underbrace{\left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a' | \theta) \right)}_{\text{Target}} \right]^2$$

- Both sides uses same network θ.
  Thus, if our Q_predict is trained, our target is consequently updated.
  → **Non-stationary targets**.
- **Solution) Separate Networks** → create a target network

- [Issue #2] Non-Stationary Targets

Target

$$\min_{\theta} \sum_{t=0}^{T} \left[\hat{Q}(s_t, a_t|\theta) - \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'|\theta) \right) \right]^2$$

$$\min_{\theta} \sum_{t=0}^{T} \left[\hat{Q}(s_t, a_t|\theta) - \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'|\bar{\theta}) \right) \right]^2$$

And periodic update!

References

- V. Mnih, *et. al.*, "**Playing Atari with Deep Reinforcement Learning**," NIPS Deep Learning Workshop (2013).
  - https://arxiv.org/abs/1312.5602
  - Citation: 2561+ (as of today)

- V. Mnih, *et. al.*, "**Human-Level Control through Deep Reinforcement Learning**," Nature (2015).
  - https://www.nature.com/articles/nature14236
  - Citation: 6066+ (as of today)

Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

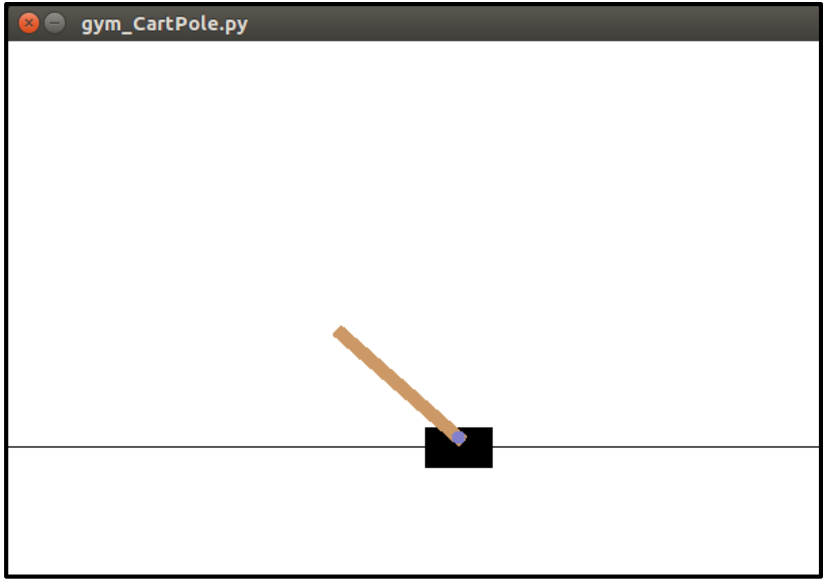
Deep Reinforcement Learning Implementation

- **Basics**
- Q-Learning Implementation
- DQN Implementation

Inverse Reinforcement Learning and Imitation Learning

Basics, Hello World: CartPole

```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)
```



Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

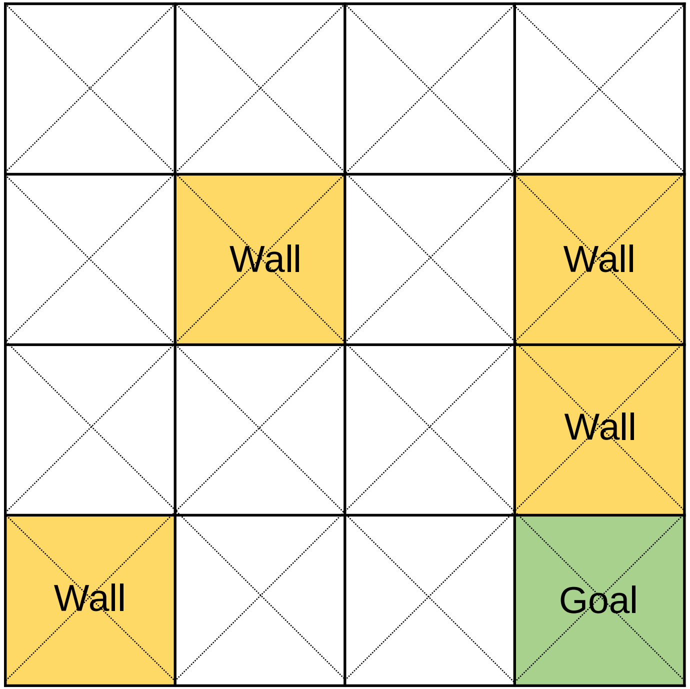
Deep Reinforcement Learning Implementation

- Basics
- **Q-Learning Implementation**
- DQN Implementation

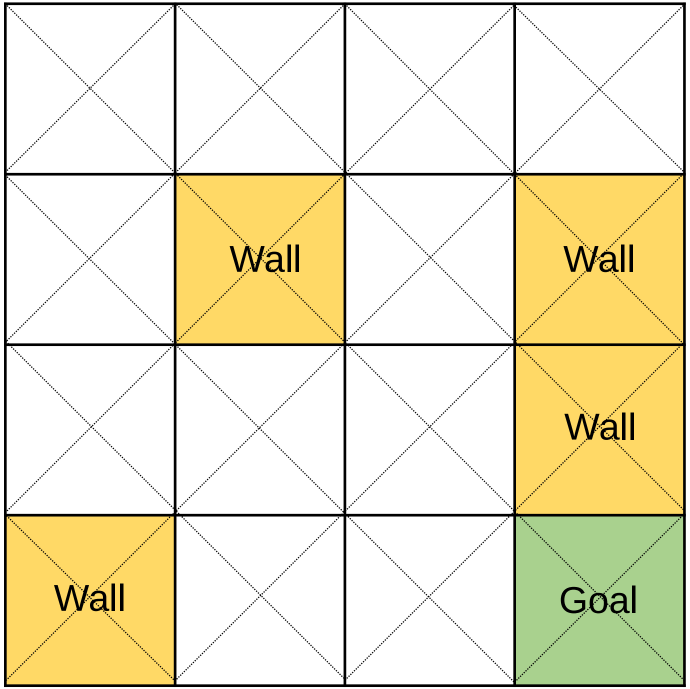
Inverse Reinforcement Learning and Imitation Learning

Outline

- Q-Learning Implementation
  - **<u>Q-Learning (Basics)</u>**
  - Q-Learning (Exploit and Exploration)




```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")
```




```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)

```
# 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
```

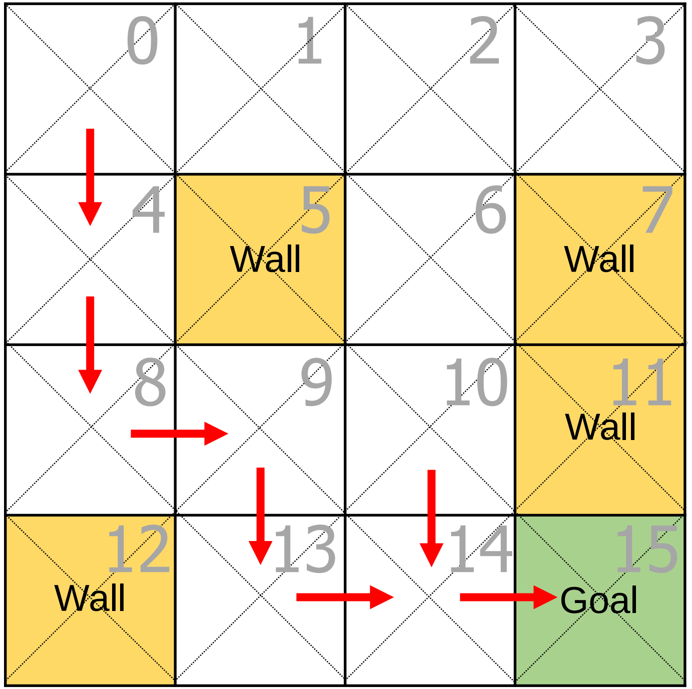
- Randomly pick one when multiple argmax values exist

```
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

[L, D, R, U]

```
'''
Final Q-Table
[[0. 1. 0. 0.]   0 (D)
 [0. 0. 0. 0.]   1
 [0. 0. 0. 0.]   2
 [0. 0. 0. 0.]   3
 [0. 1. 0. 0.]   4 (D)
 [0. 0. 0. 0.]   5
 [0. 0. 0. 0.]   6
 [0. 0. 0. 0.]   7
 [0. 0. 1. 0.]   8 (R)
 [0. 1. 0. 0.]   9 (D)
 [0. 1. 0. 0.]   10 (D)
 [0. 0. 0. 0.]   11
 [0. 0. 0. 0.]   12
 [0. 0. 1. 0.]   13 (R)
 [0. 0. 1. 0.]   14 (R)
 [0. 0. 0. 0.]]  15
Success Rate :  0.903
'''
```



- Q-Learning Implementation
  - Q-Learning (Basics)
  - **Q-Learning (Exploit and Exploration)**

Q-Learning (Exploit and Exploration)

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

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

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

Q = np.zeros([env.observation_space.n, env.action_space.n])
num_episodes = 1000

rList = []
successRate = []
e = 0.1 # exploit and exploration

mode = input(
    "Mode Selection [(1) e-greedy (2) decaying e-greedy (3) random noise (etc) original]: ")
r = 0.9 # discount factor
```

Q-Learning (Exploit and Exploration)

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

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

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

Q = np.zeros([env.observation_space.n, env.action_space.n])
num_episodes = 1000

rList = []
successRate = []
e = 0.1 # exploit and exploration

mode = input(
    "Mode Selection [(1) e-greedy (2) decaying e-greedy (3) random noise (etc) original]: ")
r = 0.9 # discount factor
```

- Parameter setting

Q-Learning (Exploit and Exploration)

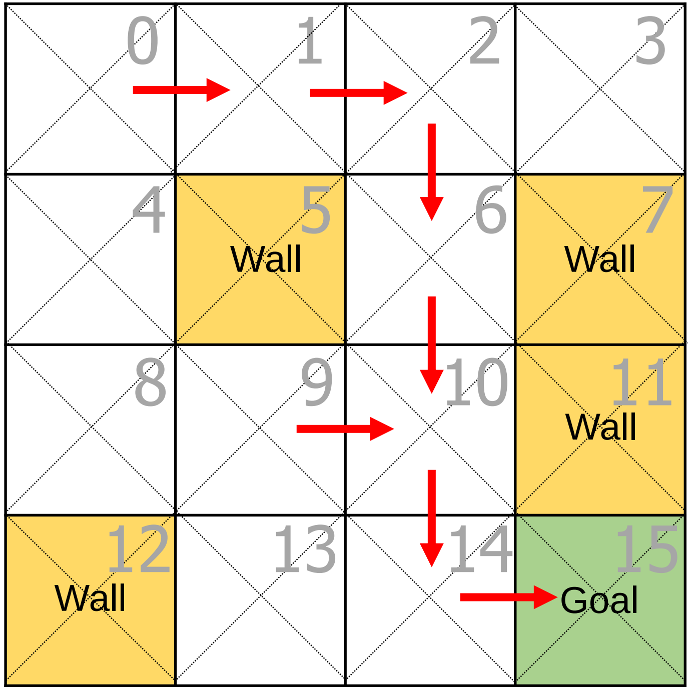
```python
def rargmax(vector):
    m = np.amax(vector)
    indices = np.nonzero(vector == m)[0]
    return random.choice(indices)

for i in range(num_episodes):
    state = env.reset()
    total_reward = 0
    done = None

    while not done:
        rand = random.random()
        # e-greedy / decaying e-greedy
        if (mode == '1' and rand < e) or (mode == '2' and (rand < e / (i + 1))):
            action = env.action_space.sample()
        # random noise
        elif mode == '3':
            action = rargmax(
                Q[state, :] + np.random.random(env.action_space.n) / (i + 1))
        # original
        else:
            action = rargmax(Q[state, :])

        new_state, reward, done, _ = env.step(action)
        Q[state, action] = reward + r * np.max(Q[new_state, :])
        total_reward += reward
        state = new_state

    rList.append(total_reward)
    successRate.append(sum(rList) / (i + 1))
```

Q-Learning (Exploit and Exploration)

```
print("Final Q-Table")
print(Q)
print("Success Rate : ", successRate[-1])
plt.plot(range(len(rList)), rList)
plt.plot(range(len(successRate)), successRate)
plt.show()
'''
Mode Selection [(1) e-greedy (2) decaying e-greedy (3) random noise (etc) original]: 2
Final Q-Table
[[0.      0.      0.59049 0.      ]
 [0.      0.      0.6561  0.      ]
 [0.59049 0.729   0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.81    0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.81    0.      ]
 [0.      0.9     0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      1.      0.      ]
 [0.      0.      0.      0.      ]]
Success Rate :  0.932
'''
```

[L, D, R, U]

```
Final Q-Table
[[0.         0.         0.59049    0.         ]  0 (R)
 [0.         0.         0.6561     0.         ]  1 (R)
 [0.59049    0.729      0.         0.         ]  2 (D)
 [0.         0.         0.         0.         ]  3
 [0.         0.         0.         0.         ]  4
 [0.         0.         0.         0.         ]  5
 [0.         0.81       0.         0.         ]  6 (D)
 [0.         0.         0.         0.         ]  7
 [0.         0.         0.         0.         ]  8
 [0.         0.         0.81       0.         ]  9 (R)
 [0.         0.9        0.         0.         ]  10 (D)
 [0.         0.         0.         0.         ]  11
 [0.         0.         0.         0.         ]  12
 [0.         0.         0.         0.         ]  13
 [0.         0.         1.         0.         ]  14 (R)
 [0.         0.         0.         0.         ]] 15
Success Rate :  0.932
```



Introduction and Preliminaries

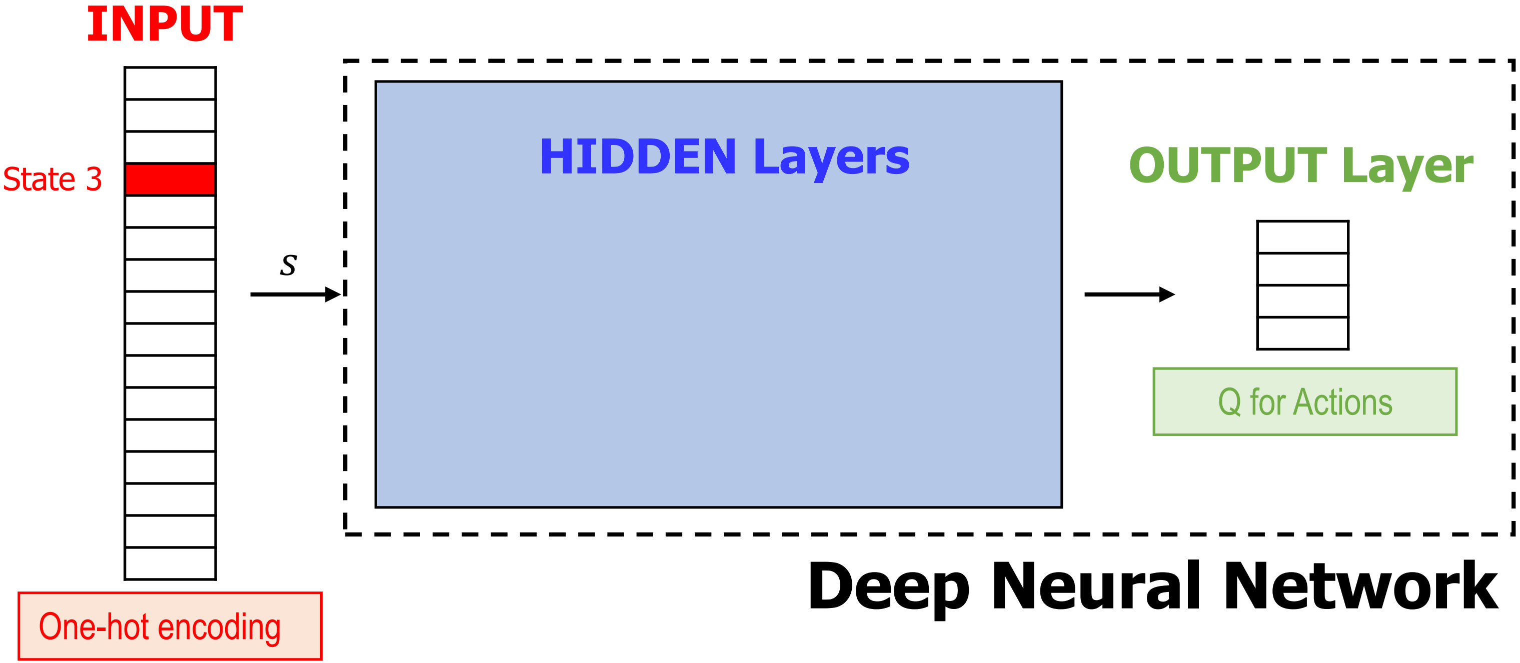
Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

- Basics
- Q-Learning Implementation
- **DQN Implementation**

Inverse Reinforcement Learning and Imitation Learning

- Frozen Lake
  - Input: States 0~15 (totally 16) → one-hot encoding
  - Output: 4 actions (totally 4) → Q-values for Up, Down, Left, and Right



Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

Imitation Learning and Autonomous Driving

- **Introduction**
- Inverse Reinforcement Learning and Imitation Learning
- Applications to Autonomous Driving

- ICML 2018 Tutorial
  - https://sites.google.com/view/icml2018-imitation-learning/

Imitation Learning Tutorial ICML 2018

- Gameplay

Pro-Gamer

Trained Agent

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

- Problems of RL

1. Reward Shaping

2. Safe Learning

3. Exploration process

Imitation Learning **handles with** these problems through the demonstration of the experts.

- Starcraft2

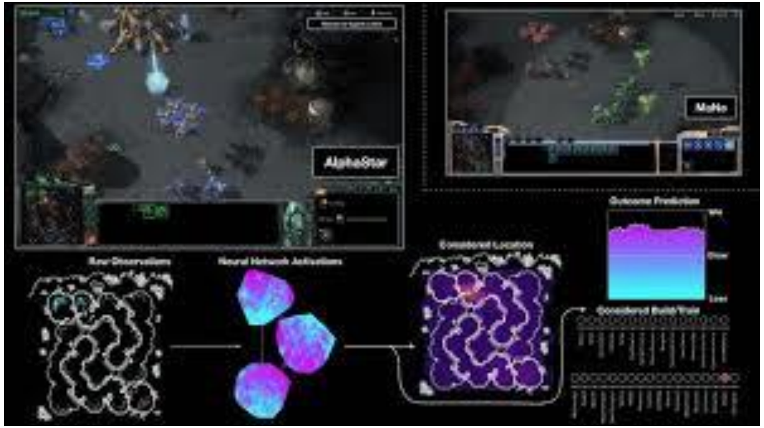
States: s = **minimap**, **screen**
Action: a = **select**, **drag**
Training set: $D = \{\tau \coloneqq (s, a)\}$ from expert
Goal: learn $\pi_\theta(s) \rightarrow$ a

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

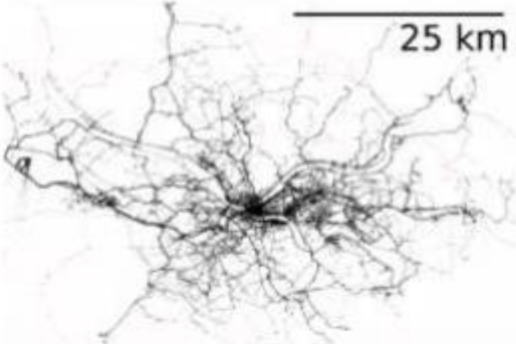
State Dynamics: P(s′|s,a)
- Typically not known to policy
- Essentially the simulator/environment

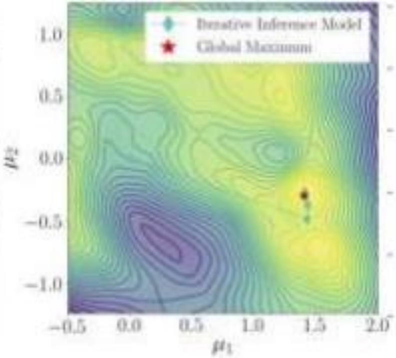
Rollout: sequentially execute $\pi_\theta(s_0)$ on initial state
- Produce trajectories $\boldsymbol{\tau}$

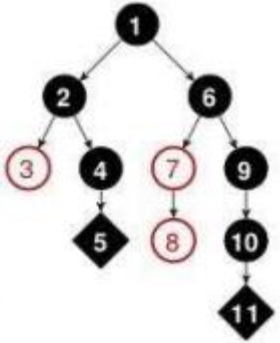
$\mathbf{P}(\boldsymbol{\tau}|\boldsymbol{\pi})$: distribution of trajectories induced by a policy
$\mathbf{P}(\mathbf{s}|\boldsymbol{\pi})$: distribution of states induced by a policy

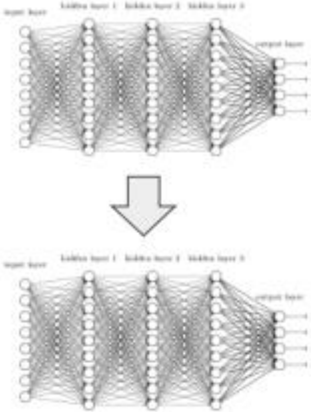


Imitation Learning Applications









Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

Deep Reinforcement Learning Implementation

Imitation Learning and Autonomous Driving

- Introduction
- **Inverse Reinforcement Learning and Imitation Learning**
- Applications to Autonomous Driving

Inverse Reinforcement Learning (IRL)

Environment Model(MDP)

RL

Reward Function R → Reinforcement Learning $\arg\max_{\pi} \mathrm{E}[\sum_{t} \gamma^{t} R(s_t)|\pi]$ → Optimal Policy π

Environment Model(MDP)

IRL

Reward Function R ← Inverse Reinforcement Learning ← Optimal Policy π

R that explains Expert Trajectories

Expert Trajectories $s_0, a_0, s_1, a_1, s_2, a_2, \cdots$

Inverse Reinforcement Learning (IRL)

RL

Environment Model(MDP)

Reward Function R → Reinforcement Learning $\arg\max_\pi \mathrm{E}[\sum_t \gamma^t R(s_t)|\pi]$ → Optimal Policy π

IRL

Environment Model(MDP)

Reward Function R ← Inverse Reinforcement Learning ← Optimal Policy π

Expert Policy

R that explains Expert Trajectories

Expert Trajectories $s_0, a_0, s_1, a_1, s_2, a_2, \cdots$

Overview: Apprenticeship Learning

- Paper
  - P. Abbeel and A.Y. Ng, "Apprenticeship Learning via Inverse Reinforcement Learning," ICML (2004).
- Abstract
  - Markov decision process
    - **Not explicitly given a reward function**
    - But **observe** an expert demonstrating the task
  - The expert → Tries to maximize a reward function
    - **Reward function → expressible as a linear combination of known features**
  - The proposed algorithm
    - Learning the task demonstrated by the expert
    - Based on using **inverse reinforcement learning** to try to recover the unknown reward function

- (Finite-State) **Markov Decision Process (MDP)**, (S, A, T, γ, D, R) where
  - S is a finite set of states
  - A is a set of actions
  - $T = \{P_{sa}\}$ is a set of state transition probabilities (P_{sa} is the state transition distribution upon taking an action a in state s)
  - $\gamma \in [0,1)$ is a discount factor
  - D is the initial-state distribution, from which the start state s_0 is drawn
  - $R: S \longmapsto A$ is the reward function (assume to be bounded in absolute value by 1)
- **MDP-wo-R**
  - MDP without a reward function, i.e., $(S, A, T, \gamma, D, \cancel{R}) \rightarrow (S, A, T, \gamma, D)$

- Given an MDP-wo-R, a feature mapping ϕ and the expert's feature expectations μ_E, find a policy whose performance is close to that of the expert's, on the unknown reward function $R^* = {w^*}^T \phi$.
- To accomplish this, we will find a policy $\tilde{\pi}$ such that ($w \in R^k$ ($\|w\|_1 \leq 1$)

$$\| \mu(\tilde{\pi}) - \mu_E \|_2 \leq \epsilon.$$

$$\left| E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \middle| \pi_E \right] - E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \middle| \tilde{\pi} \right] \right| = | w^T(\tilde{\pi}) - w^T \mu_E |$$

$|x^T y| \leq \|x\|_2 \|y\|_2$

$$\leq \|w\|_2 \|\mu(\tilde{\pi}) - \mu_E\|_2$$

$\|w\|_2 \leq \|w\|_1 \leq 1$

$$\leq 1 \cdot \epsilon = \epsilon$$

The problem is reduced to find a policy $\tilde{\pi}$ that induces feature expectations $\mu(\tilde{\pi})$ close to μ_E.

- **Proposed Apprenticeship Learning Algorithm** (for finding a policy $\tilde{\pi}$)
  1. Randomly pick some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu\left(\pi^{(0)}\right)$, and set $i = 1$.
  2. Compute
  $$t^{(i)} = \max_{w:\|w\|_2 \leq 1} \min_{j \in \{0,\ldots,(i-1)\}} \left\{ w^T \left(\mu_E - \mu^{(j)} \right) \right\}$$
     - Let $w^{(i)}$ be the value of w that attains this maximum.
  3. If $t^{(i)} \leq \epsilon$, then terminate.
     - Upon termination, the algorithm returns $\left\{ \pi^{(i)} \middle| i = 0, \ldots, n \right\}$
  4. Using the RL algorithm, compute the optimal policy $\pi^{(i)}$ for the MDP using regards $R = \left(w^{(i)}\right)^T \phi$.
  5. Compute (or estimate) $\mu^{(i)} = \mu\left(\pi^{(i)}\right)$.
  6. Set $i = i + 1$ and go back to step 2.

- **Proposed Apprenticeship Learning Algorithm** (Explanation)
  - On iteration i, we have already found some policies $\pi^0, \ldots, \pi^{(i-1)}$.
  - Step 2 (Inverse Reinforcement Learning)
    - Tries to guess the reward function being optimized by the expert.
    - The maximization in that step can be written as:

$$\max_{t,w}: t$$

$$\text{s.t.} \quad w^T \mu_E \geq w^T \mu^{(j)} + t, \quad j = 0, \ldots, i-1$$

$$\|w\|_2 \leq 1 \quad \text{QP}$$

==

Note)

$$t^{(i)} = \max_{w:\|w\|_2 \leq 1} \min_{j \in \{0,\ldots,(i-1)\}} \{w^T(\mu_E - \mu^{(j)})\}$$

The algorithm tries to find a reward function $R = w^{(i)} \cdot \phi$ such that

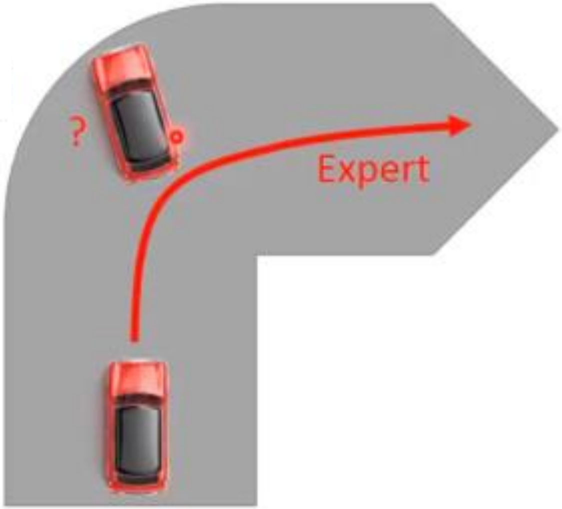
$$E_{s_0 \sim D}[V^{\pi_E}(s_0)] \geq E_{s_0 \sim D}\left[V^{\pi^{(i)}}(s_0)\right] + t$$

(a reward on which the expert does better, by a margin of t than any of the i policies we had found previously)

- **Behavior Cloning**
  - Define $P^* = P(s|\pi^*)$ (distribution of states visited by **expert**)
  - **Learning objective**

$$argmin_{\theta}\ E_{(s,a^*)\sim P^*} L(a^*, \pi_{\theta}(s))$$
$$L(a^*, \pi_{\theta}(s)) = (a^* - \pi_{\theta}(s))^2$$

  - **Discussion**
    - Works well when P^* close to the distribution of states visited by π_{θ}
    - **Minimize 1-step deviation error** along the expert trajectories



- **Interactive Expert**, **ICML 2016**
    - Can query expert at any state
    - Typically applied to rollout trajectories: $s \sim P(s|\pi)$
    - **Learning objective**

$$argmin_{\boldsymbol{\theta}} \, E_{s\sim P(s|\pi^*)} L(\pi^*(s), \pi_{\boldsymbol{\theta}}(s))$$

$$L(\pi^*(s), \pi_{\boldsymbol{\theta}}(s)) = (\pi^*(s) - \pi_{\boldsymbol{\theta}}(s))^2$$

- **Algorithm Pseudo-Code**

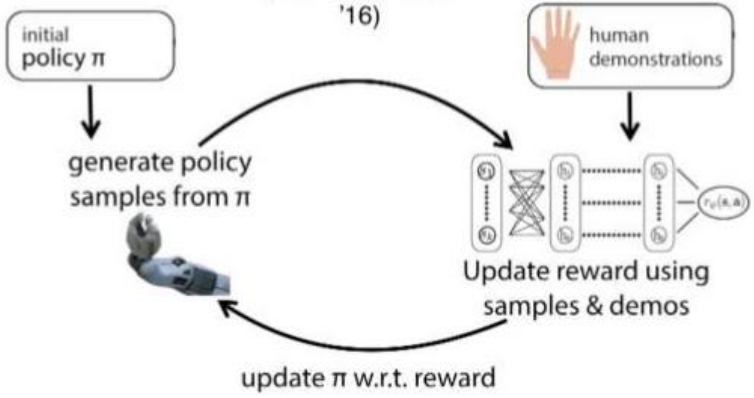
REPEAT
- Fix P, estimate π

  Solve $argmin_{\boldsymbol{\theta}} \, E_{s\sim P(s|\pi^*)} L(\pi^*(s), \pi_{\boldsymbol{\theta}}(s))$
- Fix π, estimate P

  Empirically estimate via rollout π

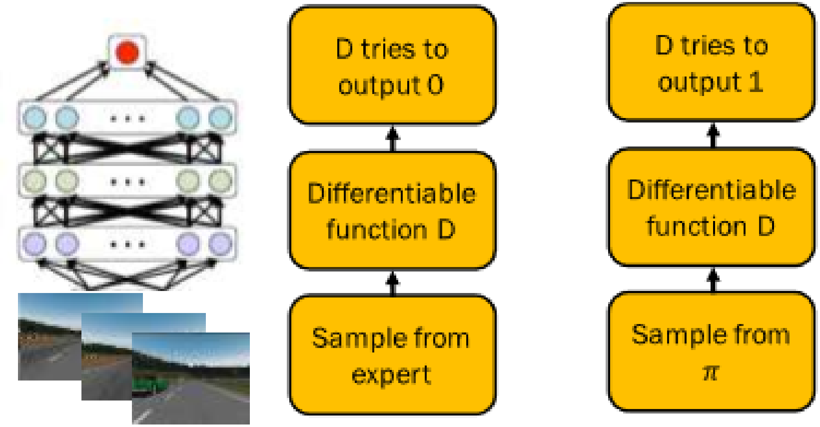
UNTIL NO CHANGE

(Finn et al. ICML '16)



- **Generative Adversarial Imitation Learning (GAIL)**, **NIPS 2016**
  - Generative adversarial imitation learning (GAIL) learns a policy that can imitate expert demonstration using **the adversarial network** from generative adversarial network (GAN).
  - **Learning Objective**

$$\boxed{argmin_{\theta}\ argmax_{\emptyset}\ E[\log(D_{\emptyset}(s,a)] + E[\log(1 - D_{\emptyset}(s,a))]}$$



Lecture Roadmap

Introduction and Preliminaries

Deep Reinforcement Learning Theory

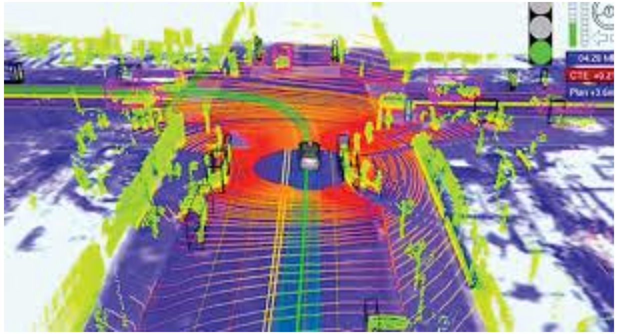
Deep Reinforcement Learning Implementation

Imitation Learning and Autonomous Driving

- Introduction
- Inverse Reinforcement Learning and Imitation Learning
- **Applications to Autonomous Driving**

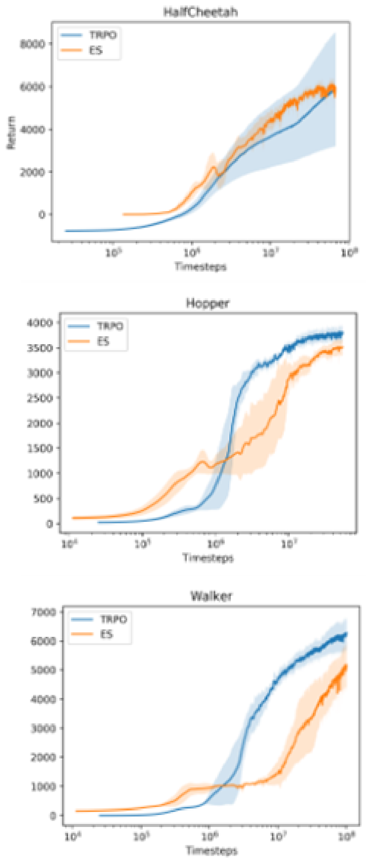
- Autonomous Driving Control

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



- **Random Search** in Parameter Space

> **Evolution Strategies as a Scalable Alternative to Reinforcement Learning**
> Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever (OpenAI)
> https://arxiv.org/abs/1703.03864
> **Simple random search provides a competitive approach to reinforcement learning**
> Horia Maria, Aurelia Guy, and Benjamin Rechet (UC-Berkeley)
> https://arxiv.org/abs/1803.07055



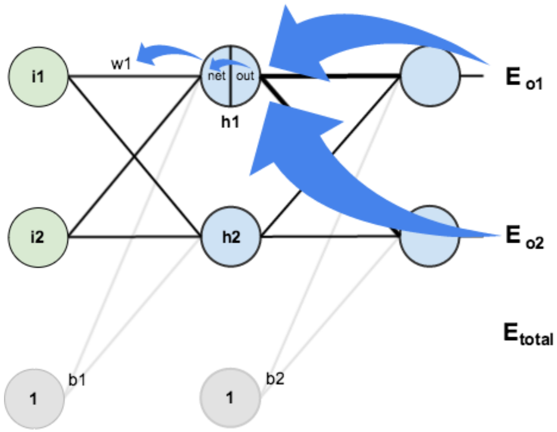
- The papers say
  - Random search is able to be so much faster is that unlike a lot of reinforcement learning algorithms that use deep learning with many hidden layers, augmented random search uses perceptrons.
  - There are fewer weights to adjust and learn, but at the same time, random search manages to get higher rewards in specific applications.

- A **derivative-free** optimization method

No Backpropagation for Training

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

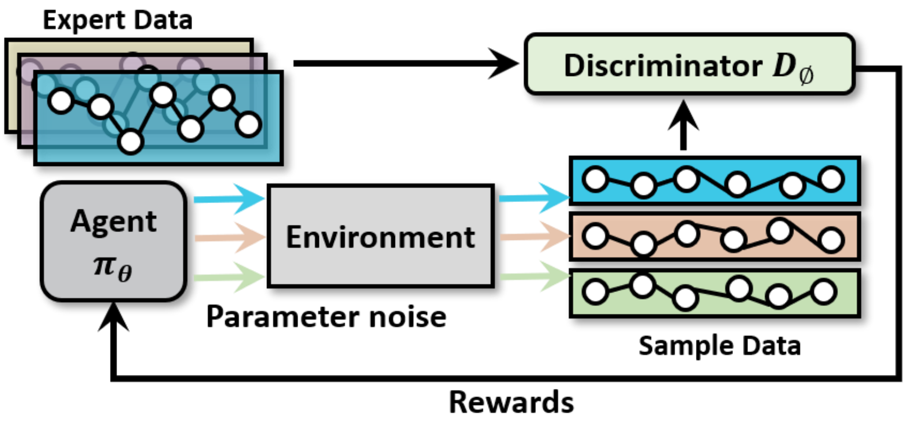
$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

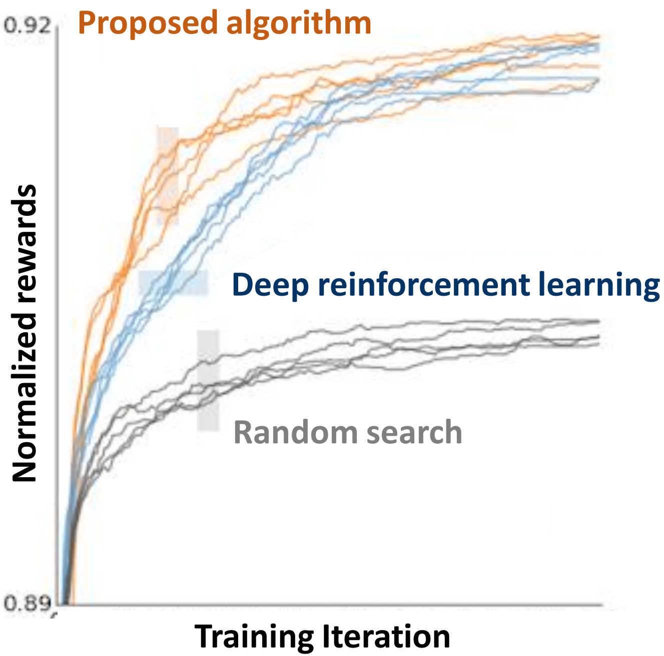


$E_{total} = E_{o1} + E_{o2}$

- **No need for backpropagation:** Random search only requires the forward pass of the policy and does not require backpropagation (or value function estimation), which makes the code shorter and between 2-3 times faster in practice.
- **Highly parallelizable:** Random search only requires workers to communicate a few scalars between each other, while in RL it is necessary to synchronize entire parameter vectors (which can be millions of numbers).

- On memory-constrained systems, it is not necessary to keep a record of the episodes for a later update.
- Optimized communication enables agents to be trained in a distributed manner easily.





M. Shin and J. Kim, "**Adversarial Imitation Learning via Random Search in Lane Change Decision-Making**," *ICML 2019 Workshop on AI for Autonomous Driving*, 2019.

M. Shin and J. Kim, "**Randomized Adversarial Imitation Learning for Autonomous Driving**," *IJCAI*, 2019., (Acceptance Rate: 850/4752=17.89%)

Generative Adversarial Network (GAN) + **Random Search**
for Autonomous Driving

Autonomous Driving with Imitation Learning