[13주차]
Deep Reinforcement Learning

1기 강다연
1기 구미진
1기 김지수
1기 조송희

EURON

Recap

So far… Supervised Learning

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

Goal: Learn a *function* to map x -> y

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

→ Cat

Classification

Recap

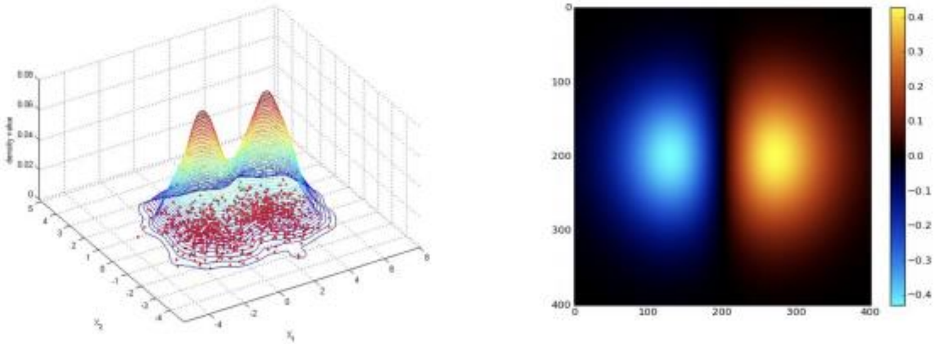
So far... Unsupervised Learning

Data: x
Just data, no labels!

Goal: Learn some underlying hidden *structure* of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

1-d density estimation



2-d density estimation

비지도 학습

지도 학습

강화 학습

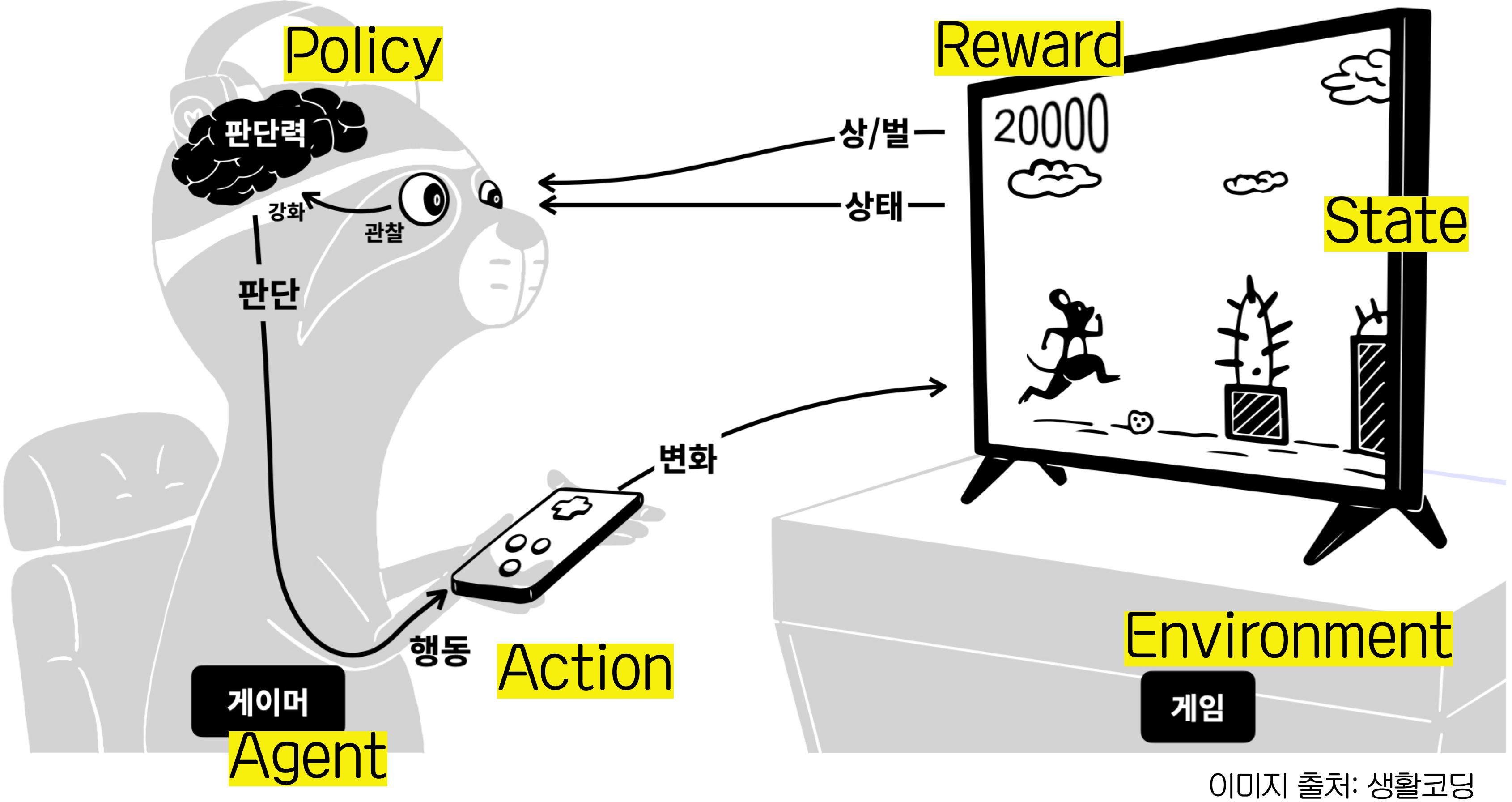
배움

지도 학습

경험

강화 학습

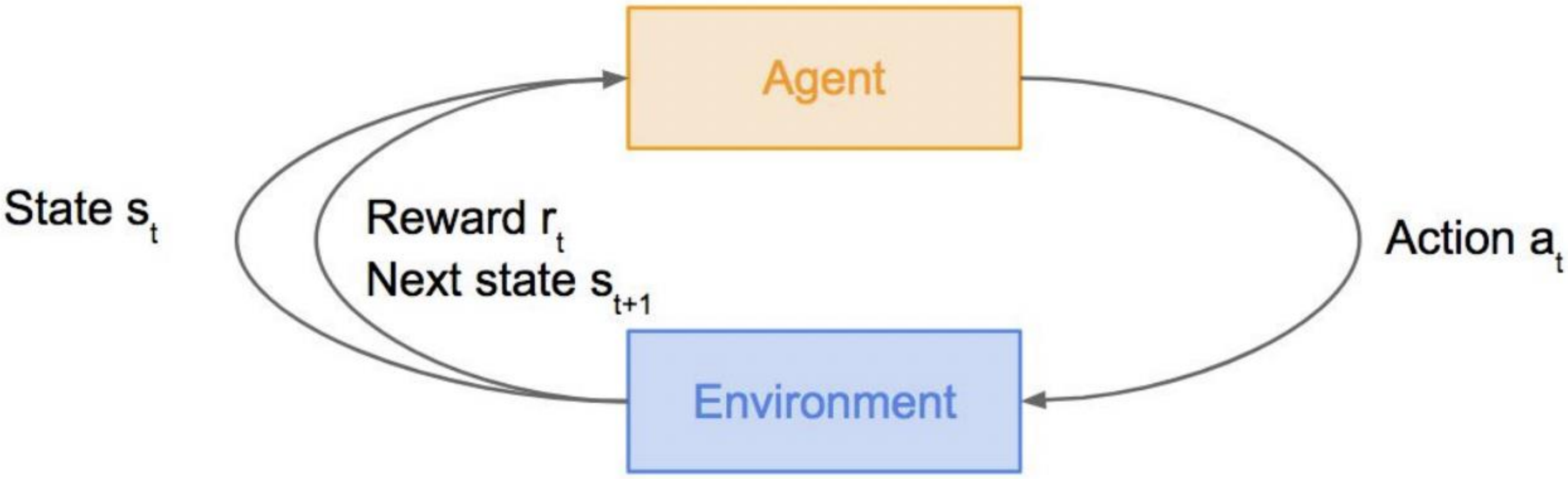
1. Reinforcement Learning



이미지 출처: 생활코딩

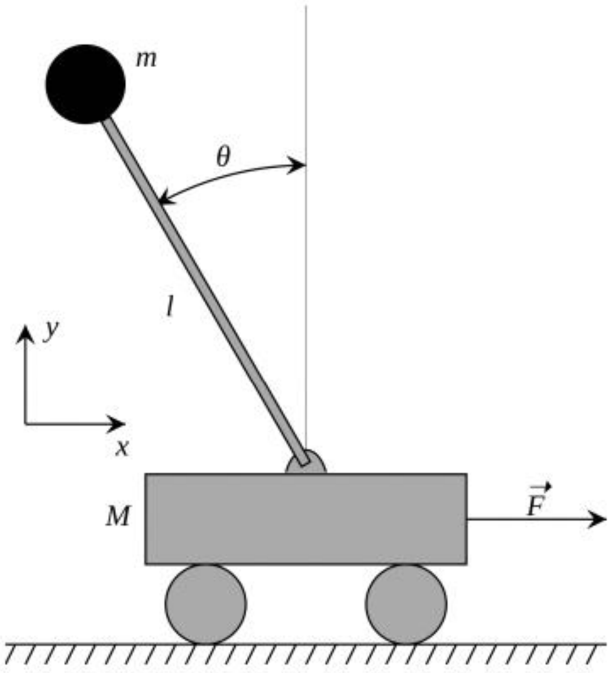
1. Reinforcement Learning

강화 학습 : 어떤 환경 안에서 정의된 에이전트가 현재의 상태를 인식하여, 선택 가능한 행동들 중 보상을 최대화하는 행동 혹은 행동 순서를 선택하는 방법



Goal : 더 많은 보상을 받을 수 있는 정책(policy)

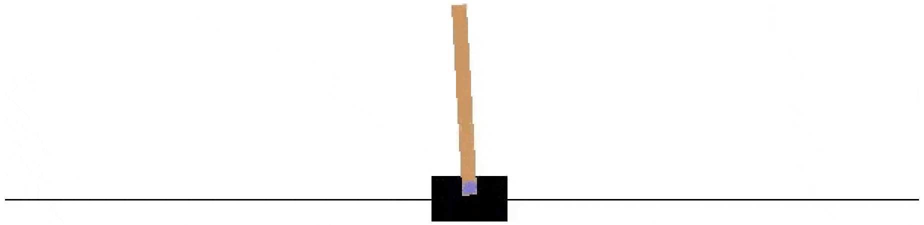
Cart-Pole Problem

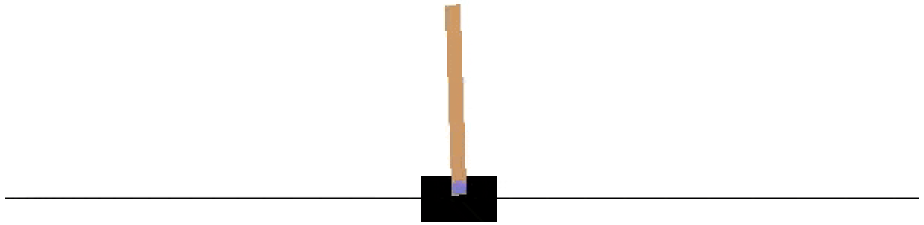


Objective: Balance a pole on top of a movable cart

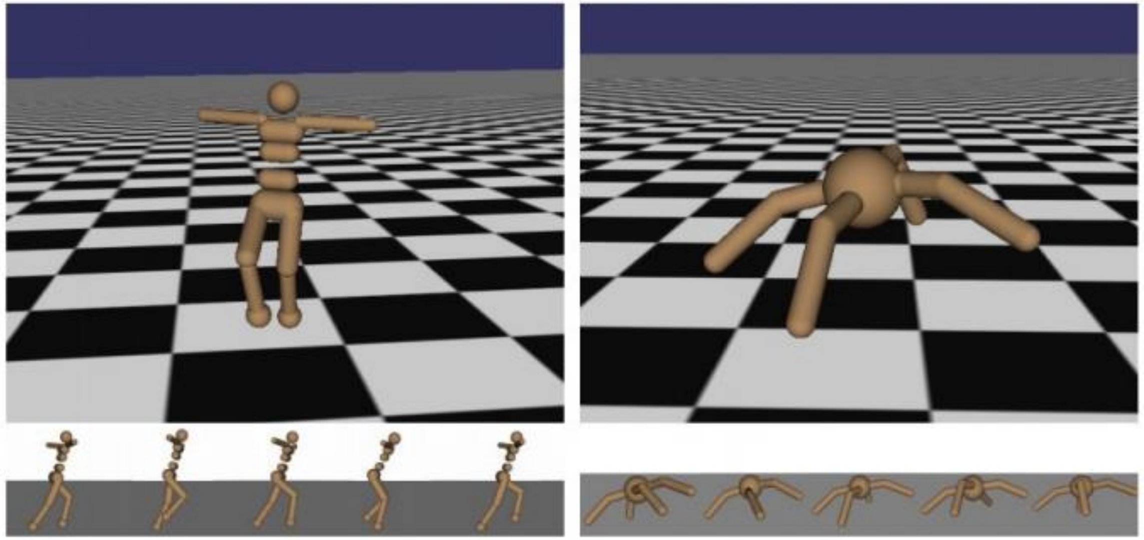
State: angle, angular speed, position, horizontal velocity
Action: horizontal force applied on the cart
Reward: 1 at each time step if the pole is upright

Cart-Pole Problem





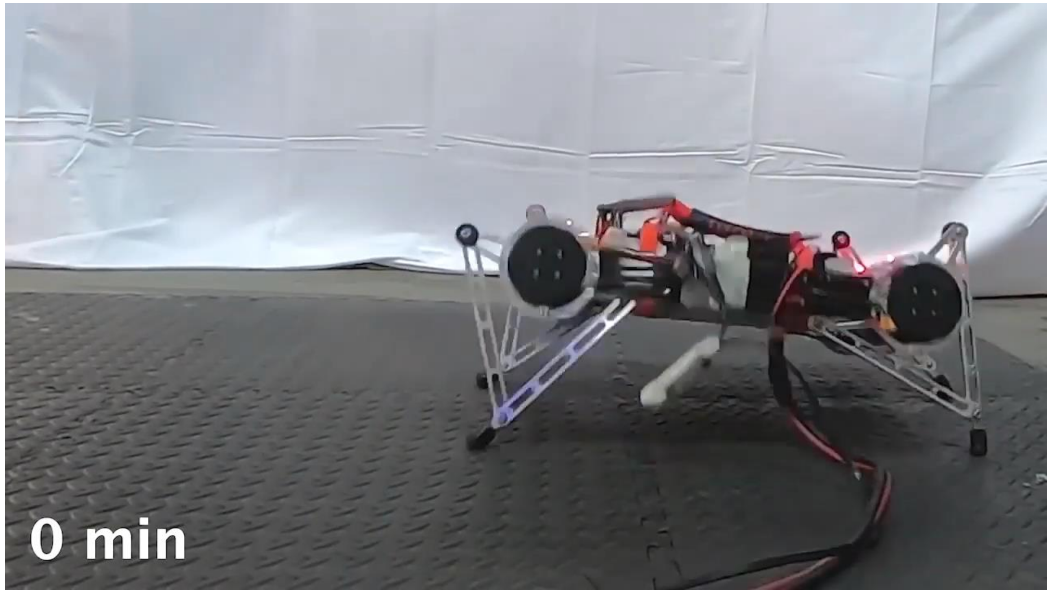
Robot Locomotion

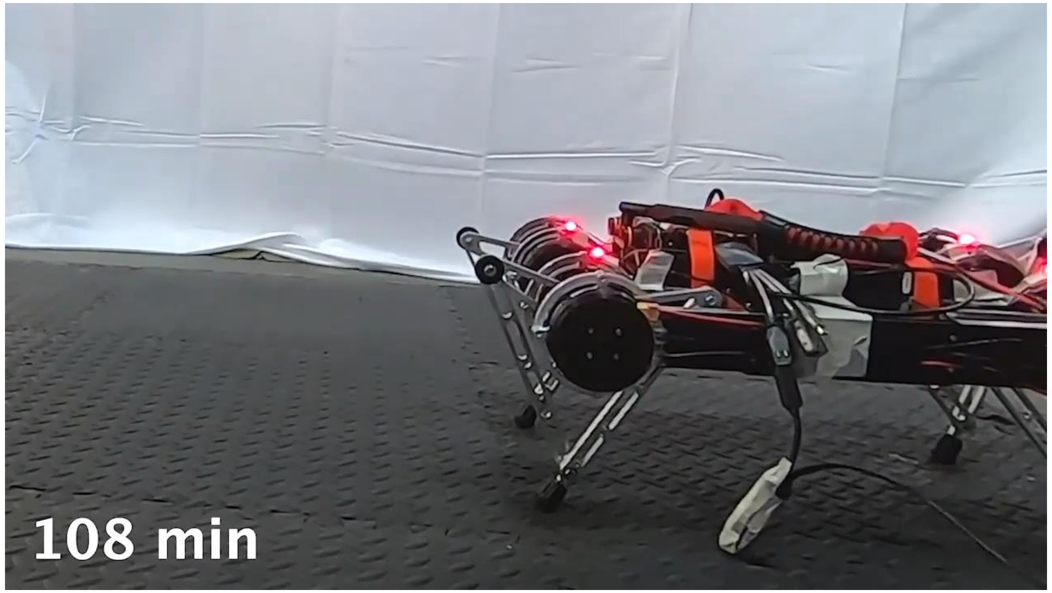


Objective: Make the robot move forward

State: Angle and position of the joints
Action: Torques applied on joints
Reward: 1 at each time step upright + forward movement

Robot Locomotion



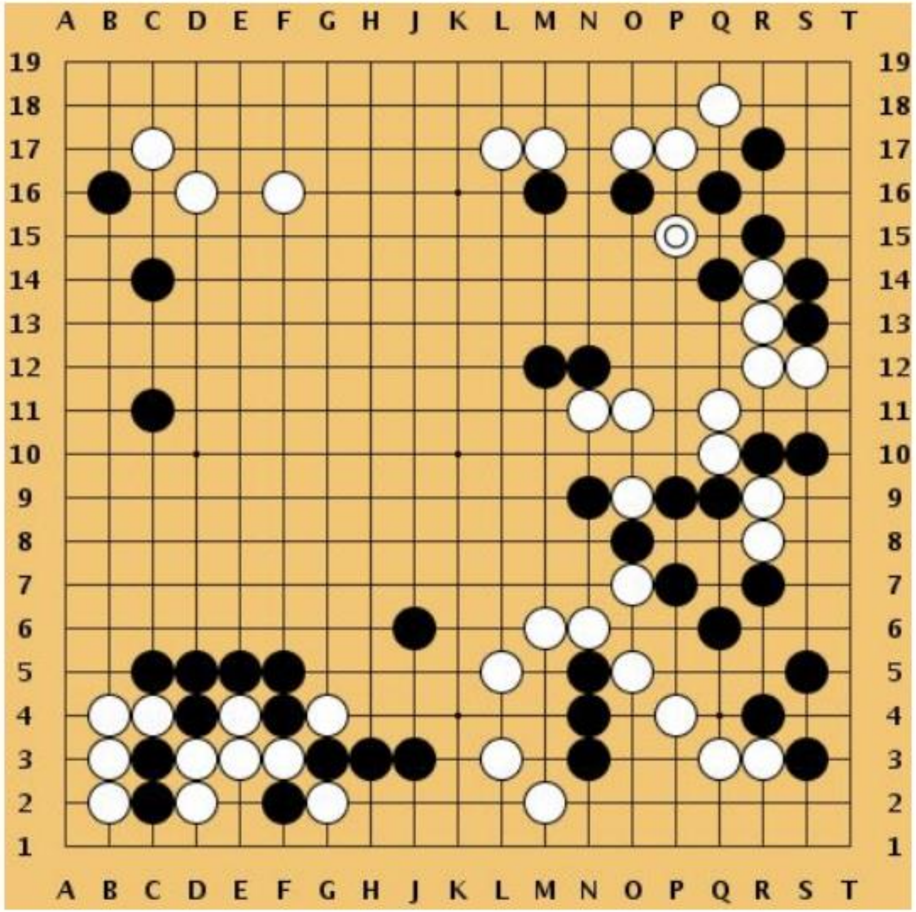


Atari Games

Objective: Complete the game with the highest score

State: Raw pixel inputs of the game state
Action: Game controls e.g. Left, Right, Up, Down
Reward: Score increase/decrease at each time step

Go



Objective: Win the game!

State: Position of all pieces
Action: Where to put the next piece down
Reward: 1 if win at the end of the game, 0 otherwise

Then How can we mathematically formalize the RL problem?

By Markov Decision Process
MDP

MDP

Markov Decision Process

- At time step t=0, environment samples initial state $s_0 \sim p(s_0)$
- Then, for t=0 until done:
  - Agent selects action a_t
  - Environment samples reward $r_t \sim R(. \mid s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
  - Agent receives reward r_t and next state s_{t+1}

Policy π

: 어떤 상태(state) A를 입력받아 취할 행동(action)을 output하는 함수로, 에이전트가 행동을 결정하기 위해 사용하는 알고리즘

강화학습의 목표 : 최적의 policy π*를 찾는것 = 누적된 보상액이 최대가 되게끔 하는 것

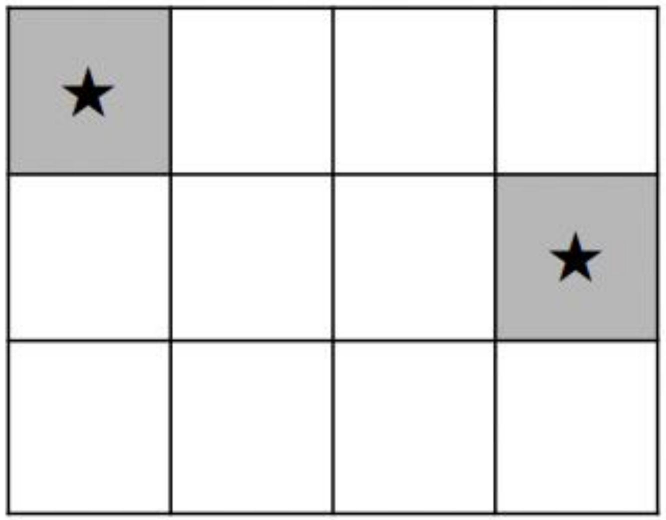
MDP

actions = {

1. right
2. left
3. up
4. down

}

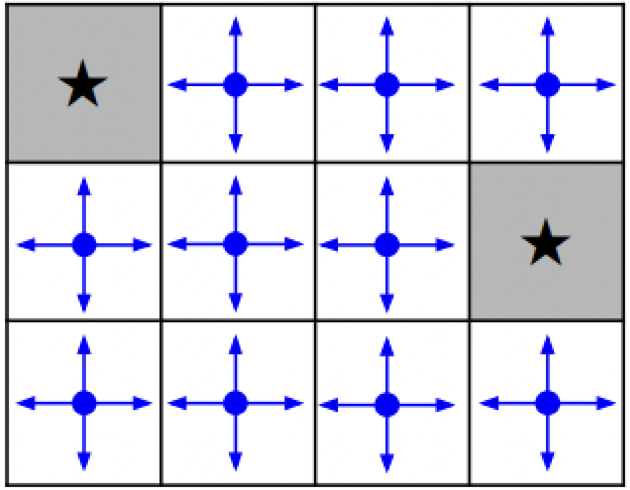
states



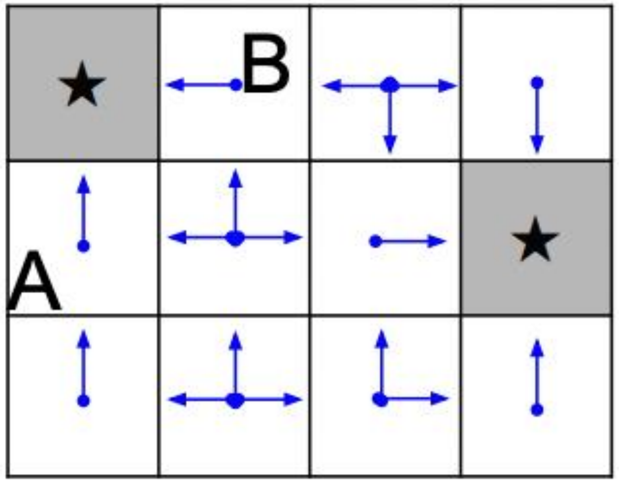
Set a negative "reward" for each transition (e.g. $r = -1$)

Objective: reach one of terminal states (greyed out) in least number of actions

MDP



Random Policy



Optimal Policy

Optimal Policy π^*

Reward의 합을 최대화

$$: \pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi\right] \text{ with } s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$$

Value Function

특정 상태에서 어떤 행동을 선택할지 기준

How good is a state?

The **value function** at state s, is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

Q-Value Function

특정 상태 s에서 특정 행동 a를 취했을 때 받을 반환값에 대한 기댓값

How good is a state-action pair?

The **Q-value function** at state s and action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Bellman Equation

벨만 방정식 :

현재 상태의 가치함수 Q*(s,a)와 다음 상태의 가치함수Q*(s'a') 사이의 관계

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

Q-learning

- 최적의 (상태) 가치 함수 $$V^*(s) = \max_{\pi} V^{\pi}(s).$$

- 벨만 방정식 $$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s').$$

- 최적의 정책 $$\pi^*(s) = \arg\max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s').$$

$$V^*(s) = V^{\pi^*}(s) \geq V^{\pi}(s).$$

Q-learning

최적의 정책은 어떻게 구할 수 있을까?

\+ Policy iteration

Value iteration

1. 모든 상태 가치를 0으로 초기화한다. V(s)=0, for all s.
2. 수렴할 때(V*를 구할 때)까지 value iteration 알고리즘을 반복한다.

$$V(s) = R(s) + \max_{a \in A} \gamma \sum_{a'} P_{sa}(s')V(s'), \quad \text{for all } s.$$

3. Optimal policy를 구할 수 있다.

$$\pi^*(s) = \arg\max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s')V^*(s').$$

Q-learning

Q-Value iteration

i -> ∞, Qi 는 Q*로 수렴

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

문제: NOT scalable

-> 반복적인 업데이트를 위해서는 모든 Q(s, a)를 계산해야 하는데 전체 state 공간은 매우 크므로 계산하는 것이 불가능

Optimal policy

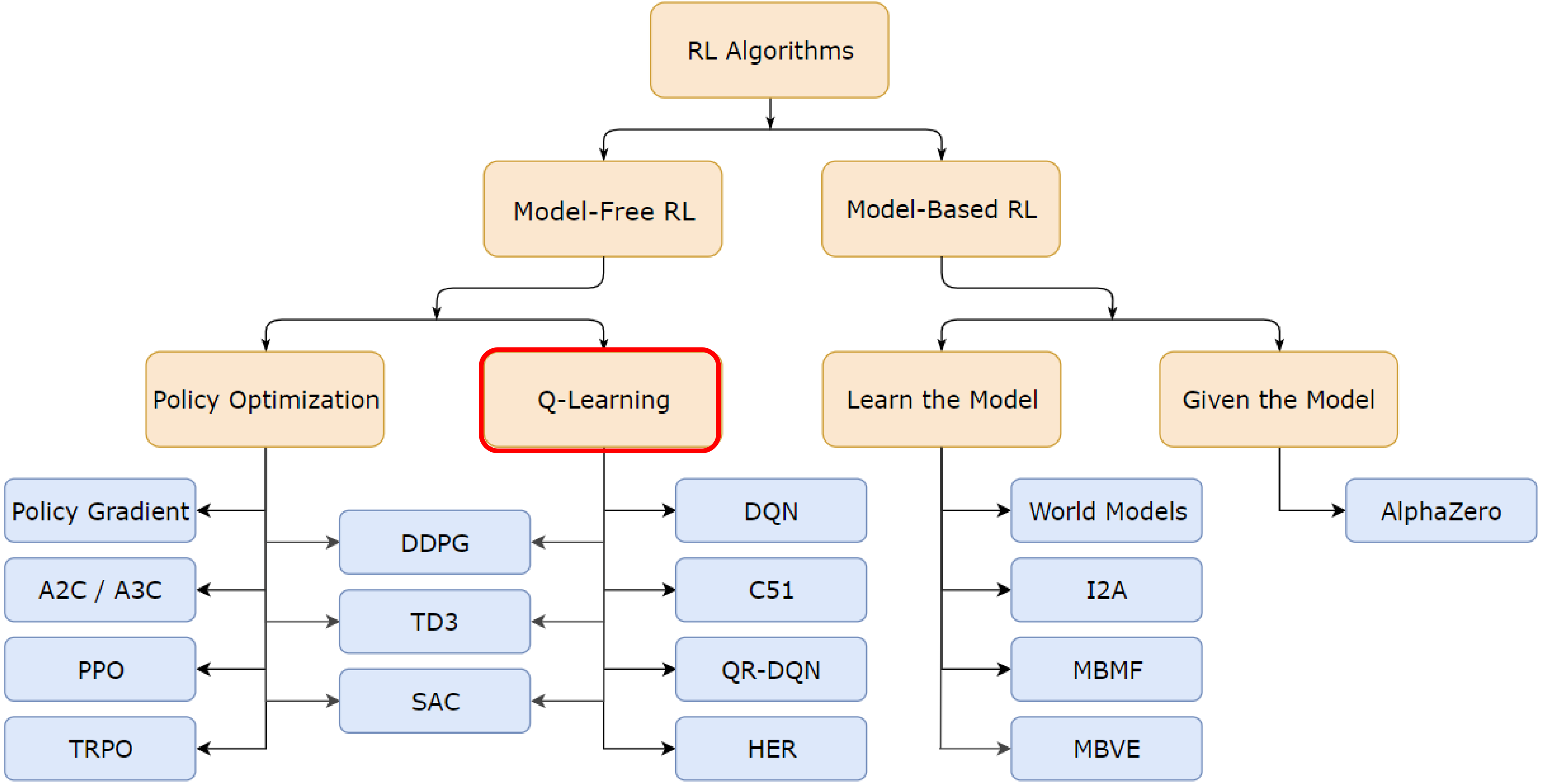
$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q^*(s,a)$$

해결: Q(s, a)를 근사하여 추정

Ex. Neural network

Q-learning

강화학습 알고리즘 분류



Q-learning

Q-learning

- 특정 상태에서 어떤 행동을 하는 것이 미래 보상을 가장 높여줄 것인지에 대한 정책을 지속적으로 업데이트하는 알고리즘
- MDP의 전이 확률과 보상을 초기에 알지 못함. 대표적인 Model-free RL
- 상태와 행동이 많은 MDP에 적용하기 어려움 -> NOT scalable
- Q러닝의 Q함수 업데이트 식

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} \right)$$

Q-learning

Q-learning의 문제

문제: NOT scalable

-> 반복적인 업데이트를 위해서는
모든 Q(s, a)를 계산해야 하는데
전체 state 공간은 매우 크므로
계산하는 것이 불가능

해결: Q(s, a)를 근사하여 추정

Approximate Q-Learning 근사 Q러닝,
Deep Q-Learning 심층 Q러닝
(Neural network를 이용하여
근사시키는 방법)

$$Q(s,a;\theta) \approx Q^*(s,a)$$

function parameters (weights)

Deep Q-learning

Action-value function 추정을 위해
Deep neural network를 이용하여 함수를 근사(=deep Q-Learning)

$$Q(s, a; \theta) \approx Q^*(s, a)$$

function parameters (weights)

최적의 Q-value 함수는 벨만 방정식을 만족하므로

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a\right]$$

Deep Q-learning

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) | s,a\right]$

target Q value

Backward Pass

Gradient update (with respect to Q-function parameters θ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

Deep Q-learning

Q-network를 학습시킬 때 발생하는 문제

하나의 batch 내에서 연속적인 샘플들로 학습하면,

1. 모든 샘플들이 상관관계(correlation)를 가져 비효율적
2. 파라미터가 다음 샘플까지 결정하는 bad feedback loops 발생

Experience replay

1. 연속적인 샘플 대신 replay memory에서 랜덤하게 샘플링 된 미니배치를 사용하여 Q-network를 학습시킴 -> 상관관계 문제 해결
2. 하나의 샘플이 여러 번 뽑혀 multiple weight update -> 데이터의 효율 증가

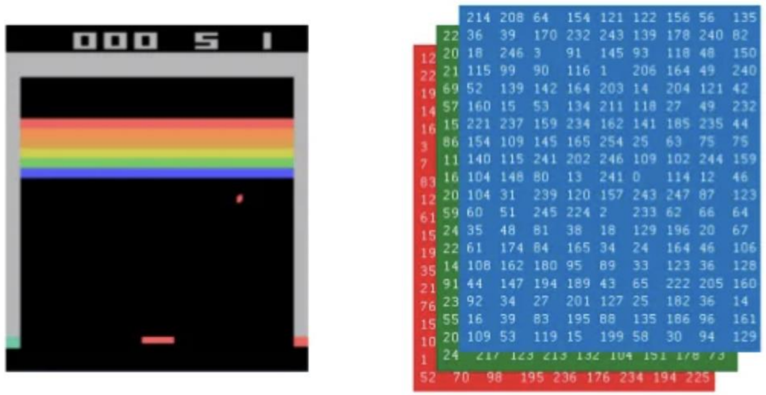
Deep Q-learning

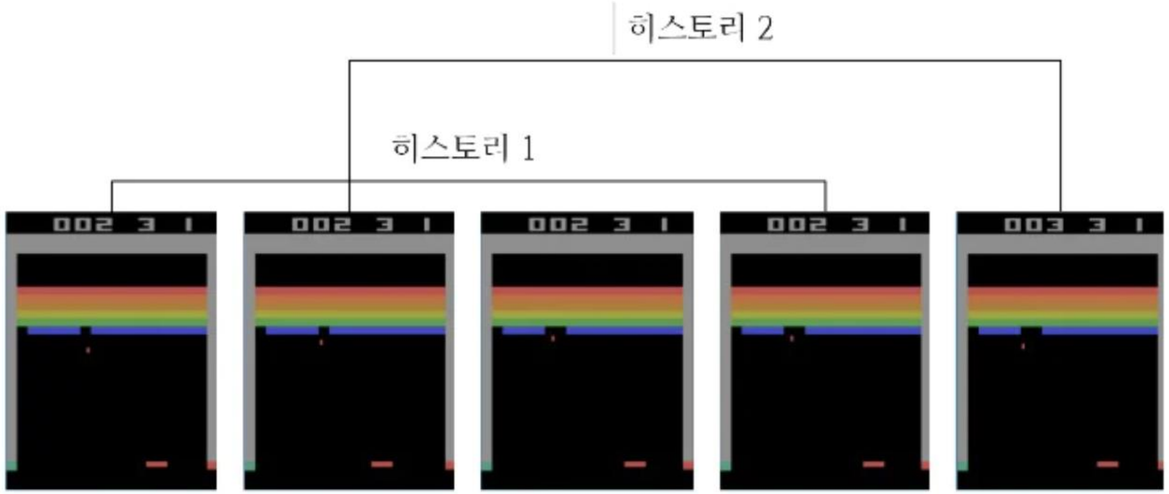
Atari games

- 목표: 게임에서 높은 점수 받기
- State: raw pixel inputs
- Action: 상/하/좌/우 움직임
- Reward: 벽돌을 깰 때마다 점수를 얻으며,
  위 층의 벽돌을 깰수록 더 큰 점수를 얻음

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Deep Q-learning





- State: raw pixel inputs

-> 게임 화면의 RGB 이미지

모든 상태의 Q함수를 저장하고
업데이트하는 방식으로는 불가능!

- State: raw pixel inputs

-> RGB 이미지 4장으로 이루어진 히스토리

Q함수를 인공신경망으로 근사하자!

Deep Q-learning

Q-network Architecture

$Q(s, a; \theta)$:
neural network
with weights θ

Output vector는 action에 대한
Q-value 값으로,
게임에서의 action이 4가지이므로
output도 4차원 FC-4

FC-4 (Q-values)

FC-256

32 4x4 conv, stride 2

16 8x8 conv, stride 4

Input으로 state를 넣으면,
모든 Q-value를 한 번의
Forward pass로 계산가능

Current state s_t: 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

[Mnih et al. NIPS Workshop 2013; Nature 2015]

DQN

We refer to convolutional networks trained with our approach as Deep Q-Networks (DQN).

DQN은 Deep Q-Network로, CNN을 이용하여 learning하는 방법이다.

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.

DQN은 Atari game의 raw pixel들을 인풋으로 입력 받고, CNN을 function approximator로 이용하여 value function 을 output으로 출력한다.

Playing Atari with Deep Reinforcement Learning(2013)

Q-network to Deep Q-network

1. CNN(Convolution Neural Network)을 적용해서
   화면의 pixel을 input data로 입력 받는다.

2. Replay memory에 경험한 Transition pair들을 저장하고
   재사용 한다(Experience Replay).

3. Q-value를 계산하는데 있어서, Target Network를 따로 구성하여 학습한다.

Playing Atari with Deep Reinforcement Learning(2013)

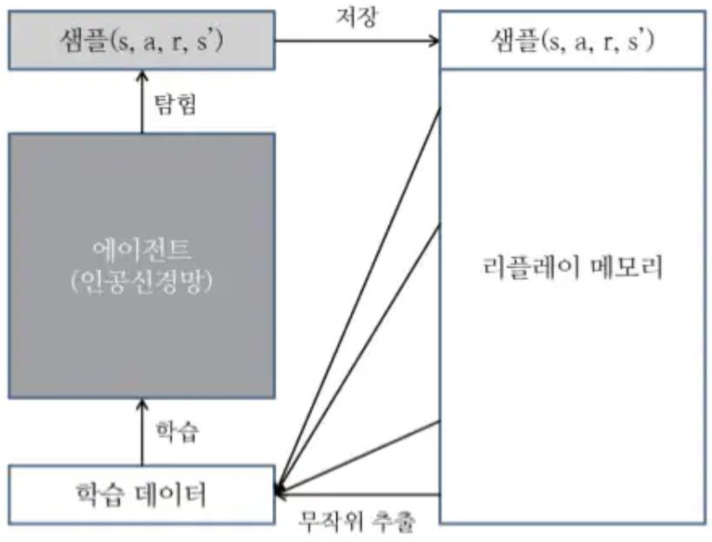
DQN

- CNN

- raw pixel을 directly input으로 사용하기위해 CNN network를 사용

- Experience replay

- agent의 experience를 FIFO로 data set에 저장해두고, update시에 radomly draw하여 mini-batch로 구성한 뒤에 parameter를 update



- Target network

- 분리된 target network를 둠으로써 parameter를 고정시킴
-> parameter가 다음 샘플까지 결정하는 bad feedback loops를 방지

DQN

Deep Q-Learning 알고리즘

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
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 $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    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

---

1) replay memory capacity인 N을 정하고,
Q-network weight를 임의로 초기화한다.

2) 총 M번의 에피소드를 진행한다.

3) timestep T만큼 학습을 수행한다.

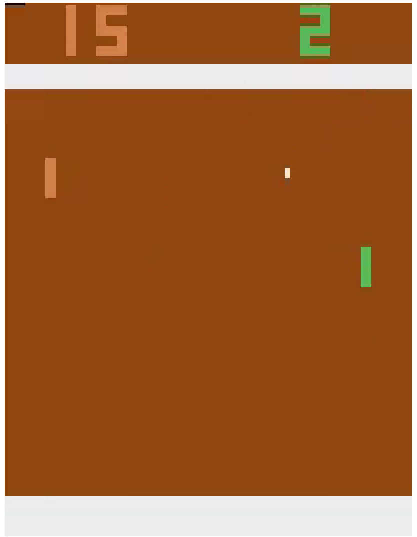
4) 대부분 정책에 따라 행동을 취하고,
일부는 임의로 행동을 취한다.

5) Transition을 replay memory에 저장한다.

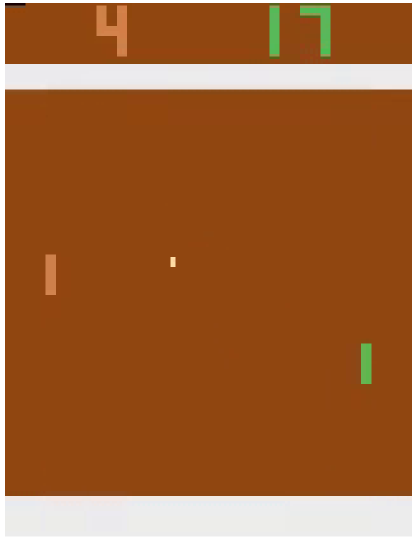
6) 연속적인 샘플을 사용하는 대신,
replay memory에서 임의의 미니배치를 샘플
링하여 학습에 이용한다.

[Mnih et al. NIPS Workshop 2013; Nature 2015]

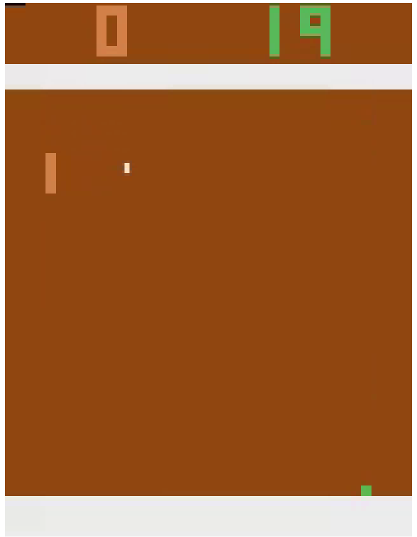
Pong Game: Atari Pong game using DQN



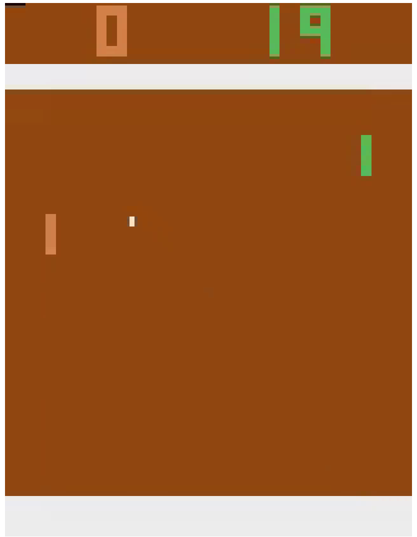
random



DQN
5260000 steps



DQN
8090000 steps



DQN
9500000 steps

Policy Gradients

Why Policy Gradient?

In high-dimension state, it is hard to find optimal policy for each state-action pair through Q-function!

Objective Function of Policy Gradient

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right]$$

➡ What to be found: optimal policy that maximize objective function

$$\theta^* = \arg\max_\theta J(\theta)$$, θ: weight of neural network

$$= \arg\max_\theta E_{\tau\sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)\right]$$

➡ **Gradient Ascent**

Policy Gradients

Gradient Ascent

to find argmax of objective function

$$\theta^{\star} = \arg\max_{\theta} \underbrace{E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]}_{J(\theta)}$$

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)} [\underbrace{r(\tau)}_{\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)}] = \int \pi_\theta(\tau) r(\tau) d\tau$$

$$\nabla_\theta J(\theta) = \int_\tau r(\tau) \nabla_\theta p(\tau; \theta) \mathrm{d}\tau$$

Intractable! Gradient of an expectation is problematic when p depends on θ

$$\nabla_\theta J(\theta) = \int \underline{\nabla_\theta \pi_\theta(\tau)} r(\tau) d\tau = \int \underline{\pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau)} r(\tau) d\tau = E_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta(\tau) r(\tau)]$$

Policy Gradients

Intractable $\nabla_\theta J(\theta) = \int_\tau r(\tau)\nabla_\theta p(\tau;\theta)\mathrm{d}\tau$

Intractable! Gradient of an expectation is problematic when p depends on θ

From a computational complexity stance, **intractable problems** are problems for which there exist no <u>efficient</u> algorithms to solve them.

Use this trick! $\nabla_\theta p(\tau;\theta) = p(\tau;\theta)\frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta)\nabla_\theta \log p(\tau;\theta)$

$$\boxed{\nabla_\theta J(\theta)} = \int_\tau \left(r(\tau)\nabla_\theta \log p(\tau;\theta)\right) p(\tau;\theta)\mathrm{d}\tau$$
$$= \boxed{\mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\nabla_\theta \log p(\tau;\theta)\right]}$$

Expectation can be estimated with **Monte Carlo sampling**

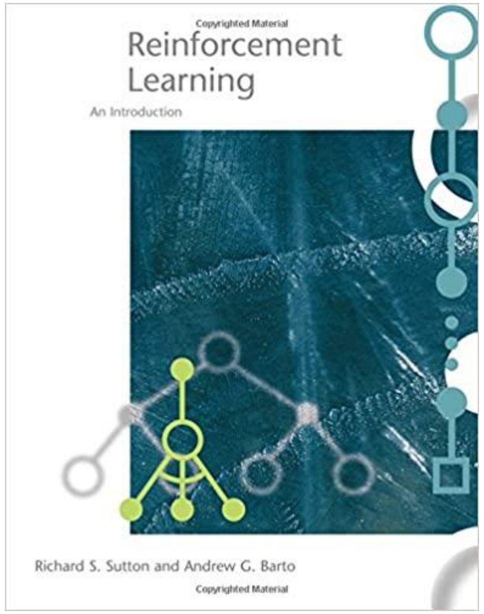
Policy Gradients

Monte Carlo? The term "Monte Carlo" is often used more broadly for any estimation method whose operation involves a significant random component. Here we use it specifically for **methods based on averaging complete returns.**

- Monte-Carlo policy evaluation uses *empirical mean* return instead of *expected* return

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta}), \forall a \in \mathcal{A}, s \in \mathcal{S}, \boldsymbol{\theta} \in \mathbb{R}^n$
Initialize policy weights $\boldsymbol{\theta}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G_t \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta})$



Policy Gradients

Gradient Estimator $$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

r(τ) ↑: push **up** the probabilities of actions

r(τ) ↓: push **down** the probabilities of actions

But it' s problematic!

It suffers from **high variance** caused by the empirical returns.

Variance Reduction

Policy Gradients

Variance Reduction

1. **First idea:** Push up probabilities of an action seen, only by the cumulative future reward from that state

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} \left(\sum_{t' \geq t} r_{t'} \right) \nabla_\theta \log \pi_\theta (a_t | s_t)$$

2. **Second idea:** Use discount factor γ to ignore delayed effects

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} \right) \nabla_\theta \log \pi_\theta (a_t | s_t)$$

Policy Gradients

Variance Reduction

3. Baseline To reduce variance is **subtract a baseline b(s)** from the returns in the policy gradient.

Idea: Introduce a baseline function dependent on the state.
Concretely, estimator is now:

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} \left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

A simple baseline: constant moving average of rewards experienced so far from all trajectories

By introducing a baseline, we can **recalibrate the rewards relative to the average action.**

Reinforce algorithm

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

1.Sample the trajectories from the policy

2.Calculate the gradient

3.Update the policy

Reinforce algorithm

function REINFORCE
    Initialise θ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
        for $t = 1$ to $T-1$ **do**
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return θ
end function

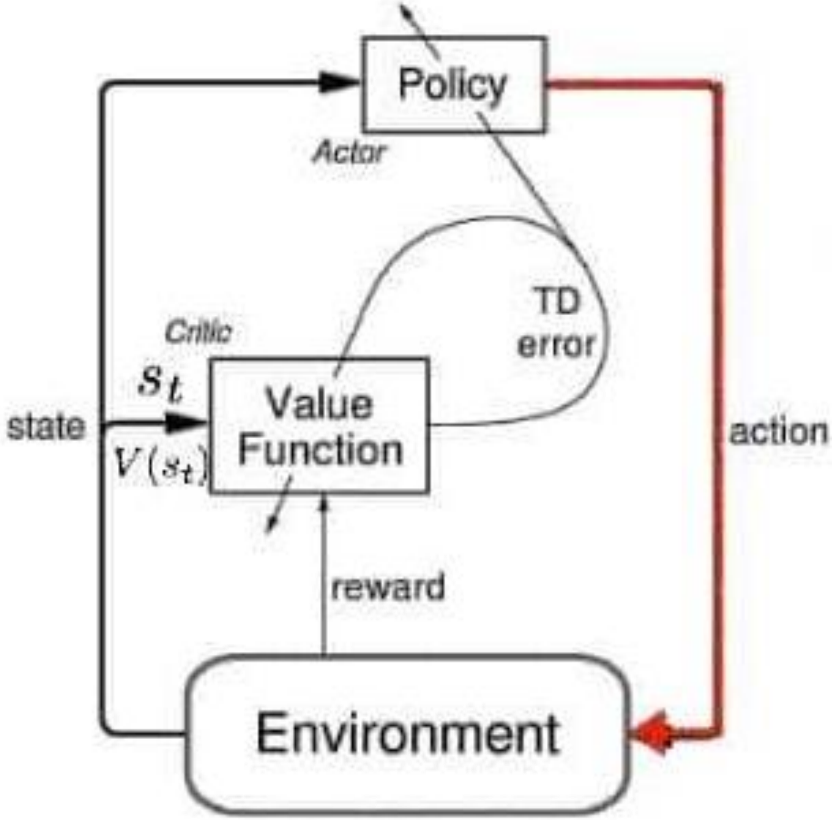
1. Perform a trajectory roll-out using the current policy
2. Store log probabilities (of policy) and reward values at each step
3. Calculate discounted cumulative future reward at each step
4. Compute policy gradient and update policy parameter
5. Repeat 1–4

Actor-Critic Algorithm

Expected value of what we should get from state

estimator

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t | s_t)$$



Actor: policy. 어떤 action을 취할 것인지 결정

Critic: Q-function. Actor에게 이 action이 좋은지, 어떻게 조정할지 알려줌

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

Actor-Critic Algorithm

Initialize policy parameters θ, critic parameters ϕ

For iteration=1, 2 … **do**

Sample m trajectories under the current policy

$\Delta\theta \leftarrow 0$

For i=1, …, m **do**

For t=1, ... , T **do**

$$A_t = \sum_{t' \geq t} \gamma^{t'-t} r_t^i - V_\phi(s_t^i)$$

Advantage function

$$\Delta\theta \leftarrow \Delta\theta + A_t \nabla_\theta \log(a_t^i | s_t^i)$$

Gradient estimator

$$\Delta\phi \leftarrow \sum_i \sum_t \nabla_\phi ||A_t^i||^2$$

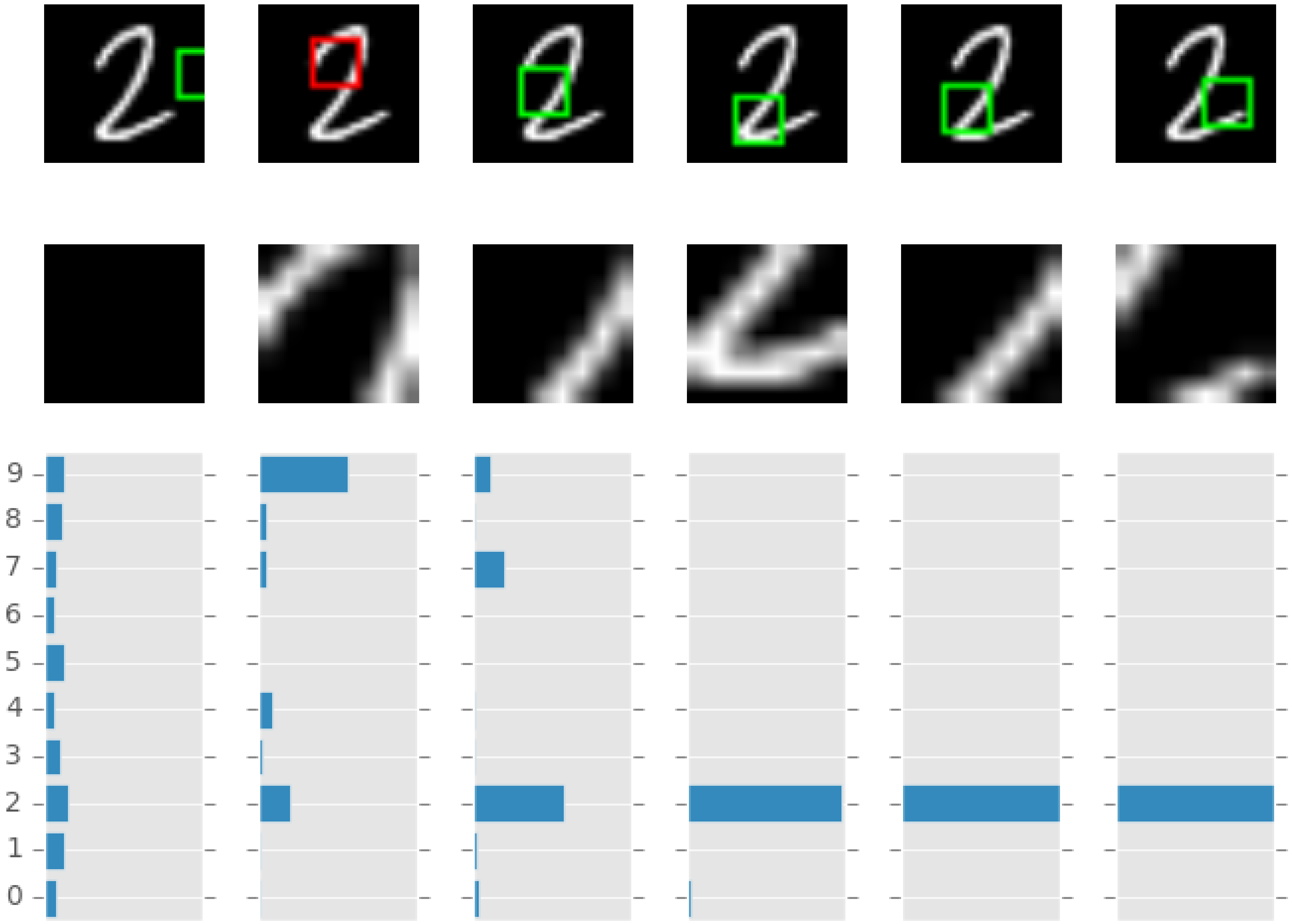
Learning, optimizing critic function

$$\theta \leftarrow \alpha\Delta\theta$$
$$\phi \leftarrow \beta\Delta\phi$$

Update

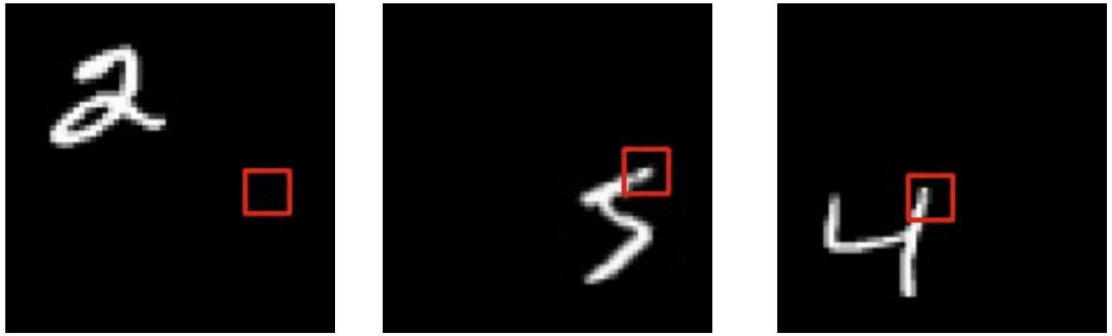
End for

RAM (Recurrent Attention Model)



Hard attention

RAM (Recurrent Attention Model)

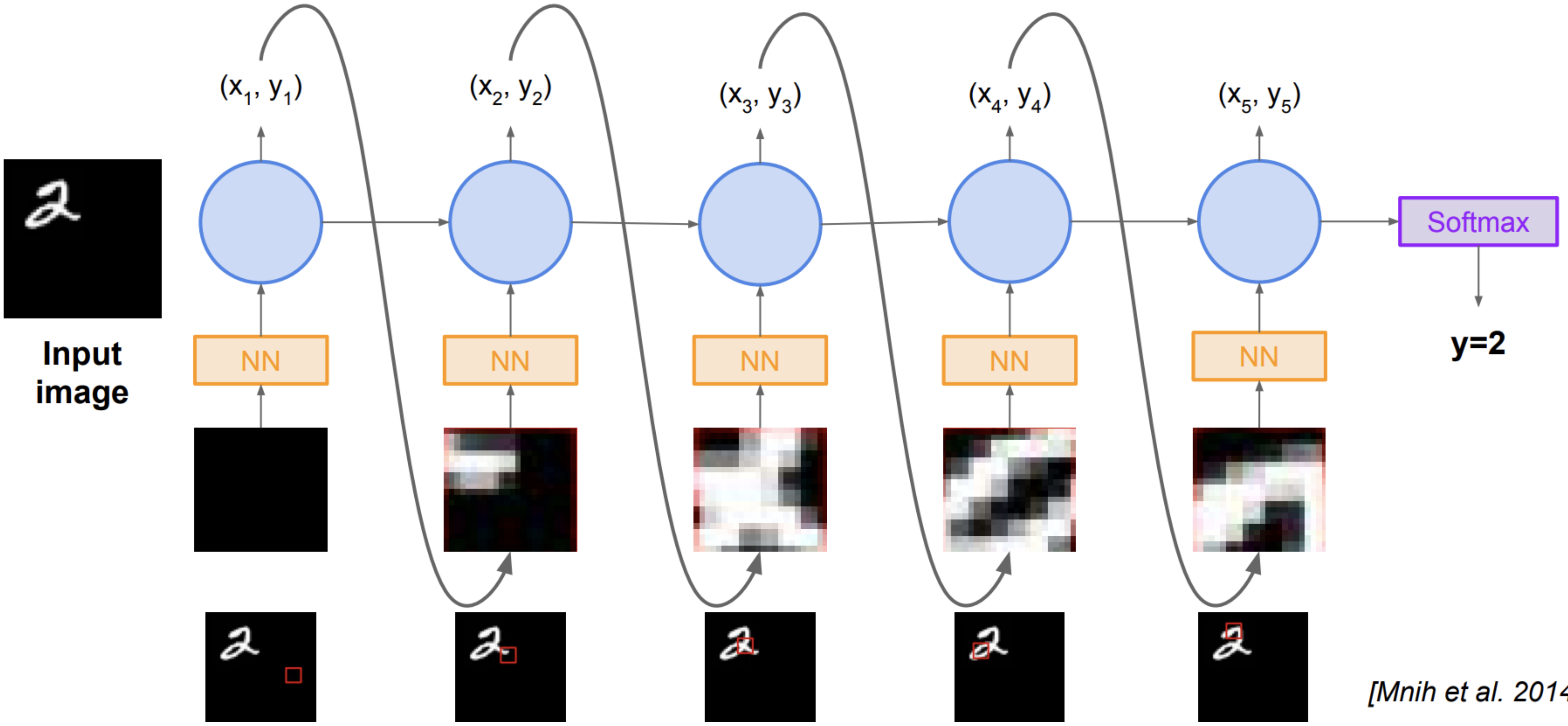


State: 지금까지 본 것
Action: 다음에 볼 box의 중심 (x, y) 좌표
Reward: final timestep에서 이미지를 옳게 분류했으면 1

틀리게 분

류했으면 0

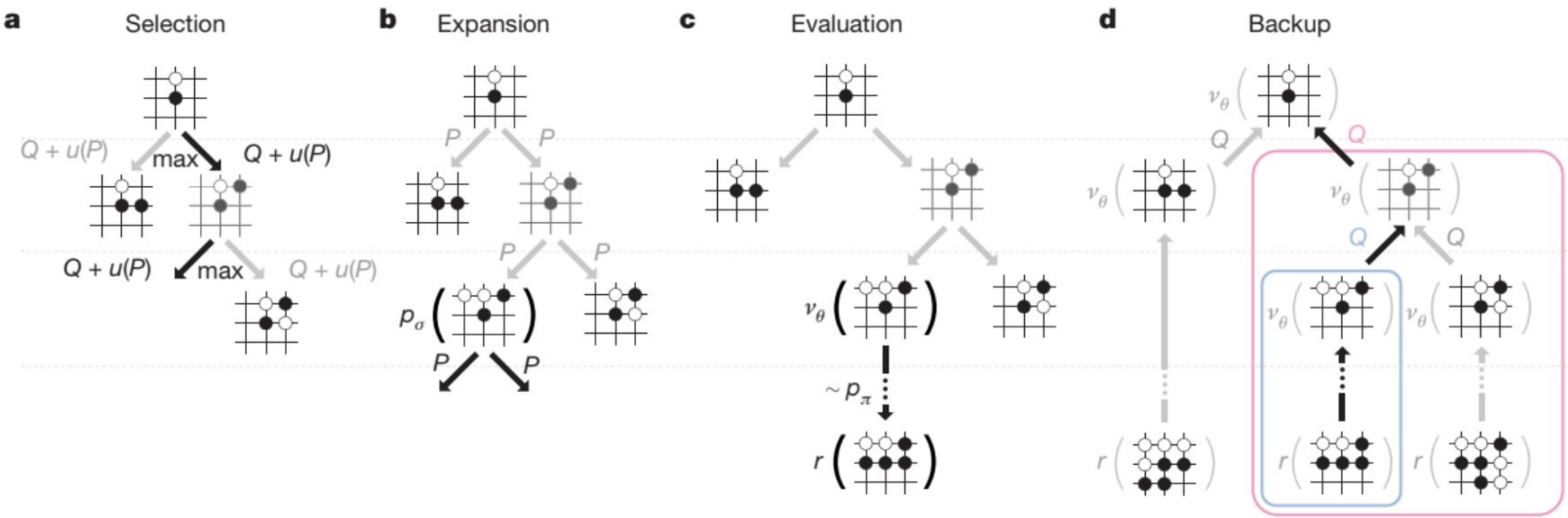
RAM (Recurrent Attention Model)



[Mnih et al. 2014]

AlphaGo

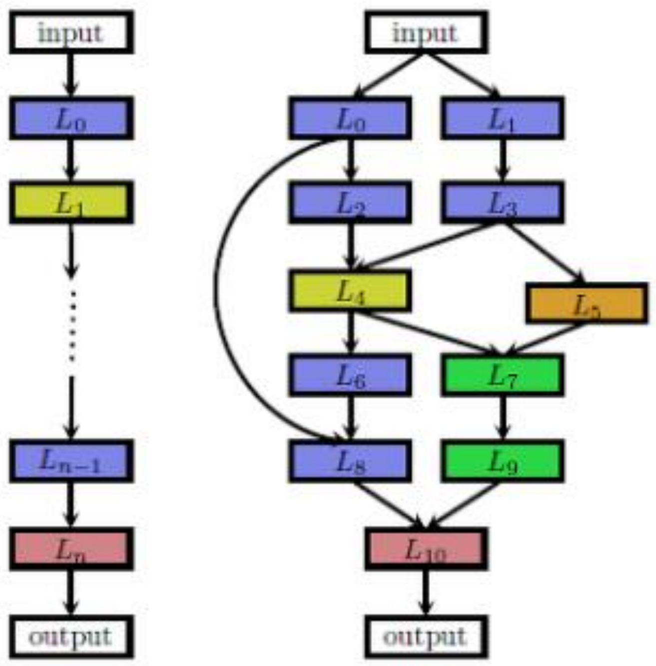
Supervised learning
+
Reinforcement learning



Neural Architecture Search (NAS)

Automatically generate model

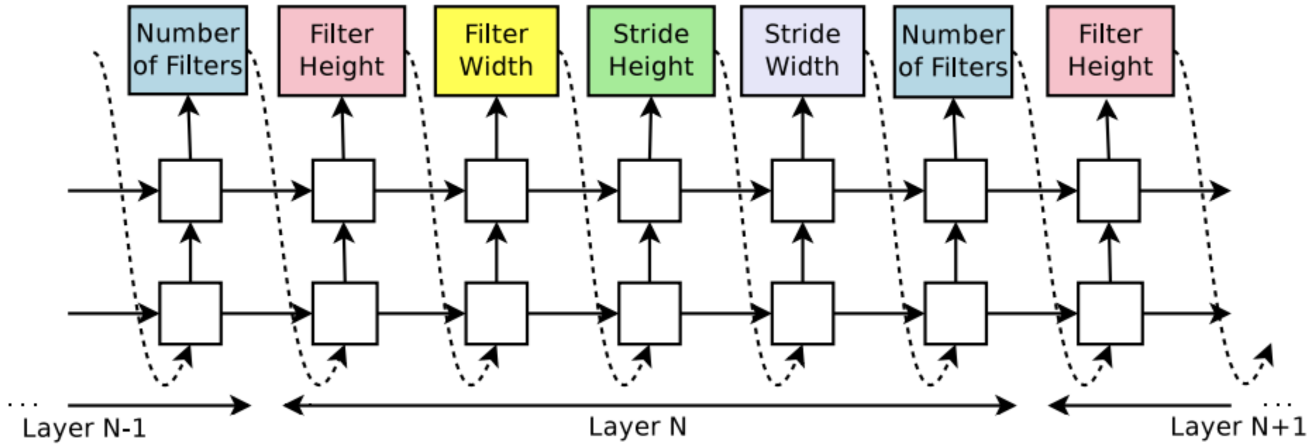
1. Search Space: neural architecture 정의
2. Search Strategy: 최대 성능의 architecture 찾기
3. Performance Estimation Strategy: 성능 평가



Neural Architecture Search (NAS)

with Reinforcement Learning

Step1 Generate model descriptions with a controller RNN

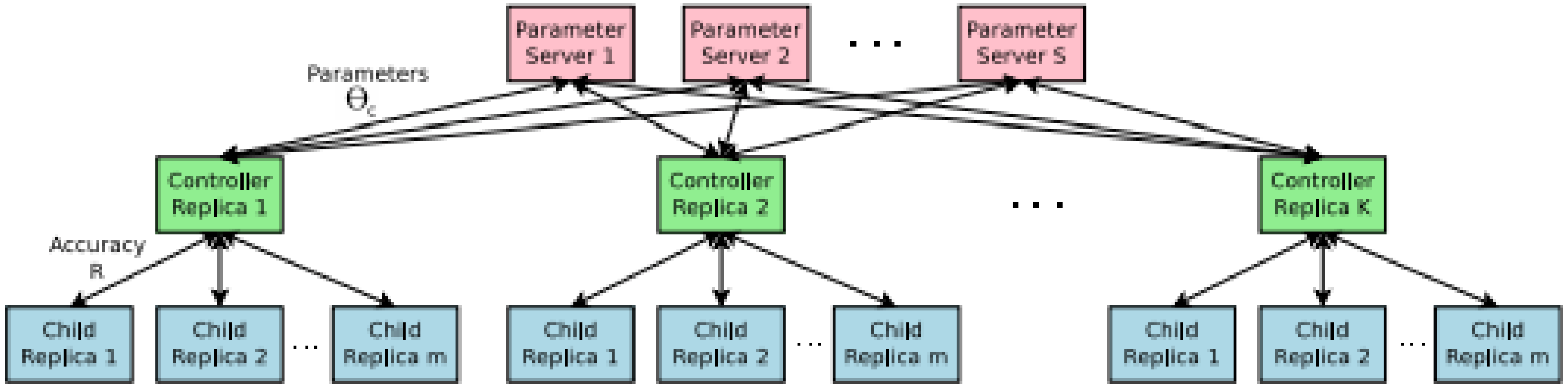


https://arxiv.org/abs/1611.01578

Neural Architecture Search (NAS)

with Reinforcement Learning

Step2 Training with reinforce

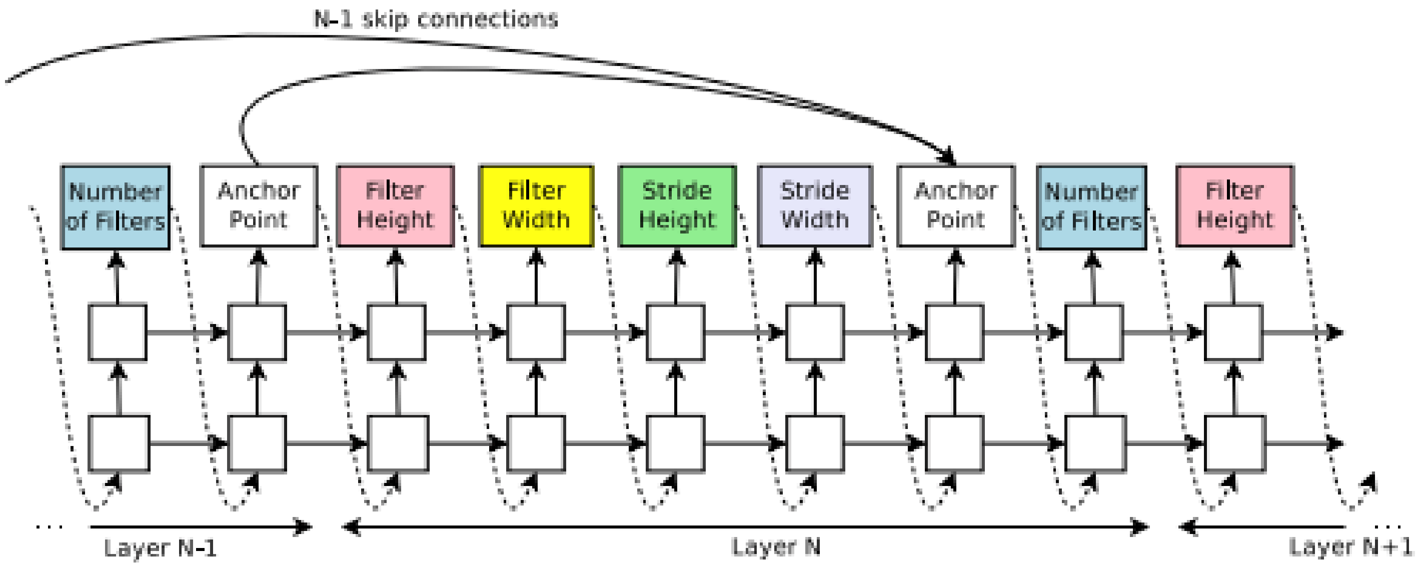


https://arxiv.org/abs/1611.01578

Neural Architecture Search (NAS)

with Reinforcement Learning

Step3 Increase architecture complexity with skip connection and other layer types

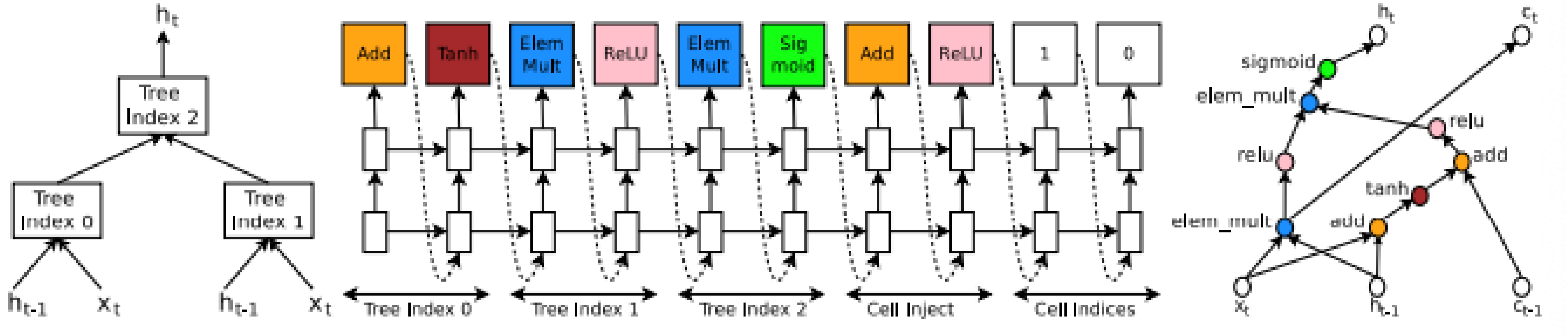


https://arxiv.org/abs/1611.01578

Neural Architecture Search (NAS)

with Reinforcement Learning

Step4 Generate recurrent cell architecture



https://arxiv.org/abs/1611.01578

AutoML

시간 소모적이고 반복적인 기계 학습 모델 개발 작업을 자동화하는 프로세스

- Data preprocessing
- Feature Engineering
- Model selection (architecture search)
- Hyperparameter optimization
- Pipeline selection
- Auto selection of evaluation metric and validation procedure
- Problem checking
- Analysis result
- Offering user interface and visualization

Meta-learning (Learning to learn)

적은 데이터로 학습하기 위한 방법

- Meta-training
- Meta-testing

→ Meta Reinforcement Learning
: Meta-learning + Reinforcement learning

감사합니다

Q&A