Playing Atari with Deep Reinforcement Learning

김진수

wlstn5376@gmail.com

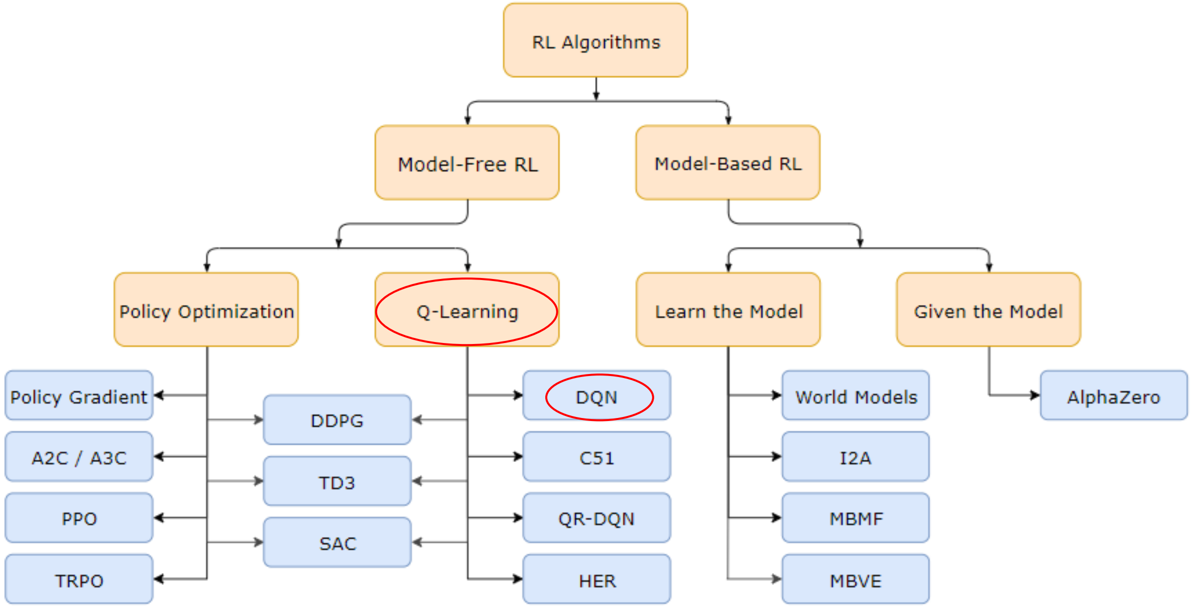
Table of Content

- ❑ Introduction
  - Fundamental of Reinforcement Learning
  - Markov Process
  - Q Learning
  - Fitted Q Learning
- ❑ Deep Reinforcement Learning
  - Deep Q Learning
  - Replay Memory
  - Non-stationary target
  - Algorithm
- ❑ Experiment and Result
  - Atari
  - Cart Pole
- ❑ Extra
  - Double DQN
  - Dueling Network Architecture
  - Policy Gradient Method

Introduction

- Fundamental of Reinforcement Learning

[1]



1. An Introduction to Deep Reinforcement Learning, Vincent-Francois, page 18, Fig 3.2

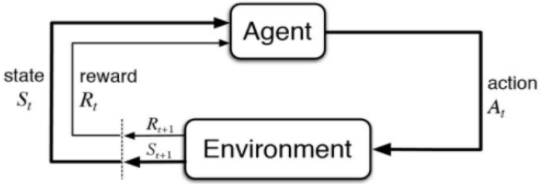
Introduction

❏ Fundamental of Reinforcement Learning

- Definition

: ML technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experience → **Learn a optimal policy which maximizes reward**

- Elements



(1) Agent : object that takes decisions based on the rewards and punishment

(2) Environment : Physical world in which the agent interacts

(3) State : Current situation of the agent

(4) Reward : Feedback from the environment

(5) Action : mechanism by which the agent transitions between states of the environment

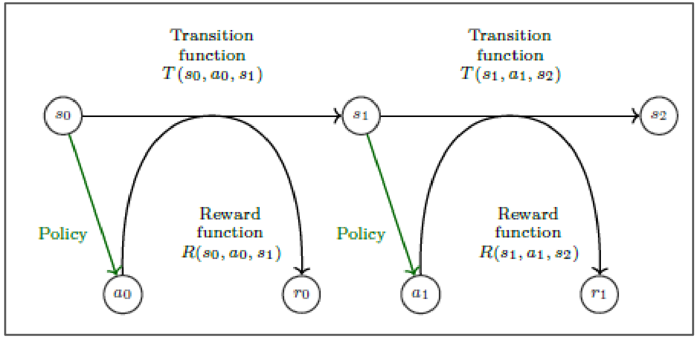
(6) value : Future reward that an agent would receive by taking an action in a particular state

Introduction

❑ Markov Process

“The Future is independent of the past given the present”

$$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_1, \ldots, S_t]$$



Definition 3.1. A discrete time stochastic control process is Markovian (i.e., it has the Markov property) if

- $\mathbb{P}(\omega_{t+1} \mid \omega_t, a_t) = \mathbb{P}(\omega_{t+1} \mid \omega_t, a_t, \ldots, , \omega_0, a_0)$, and
- $\mathbb{P}(r_t \mid \omega_t, a_t) = \mathbb{P}(r_t \mid \omega_t, a_t, \ldots, , \omega_0, a_0)$.

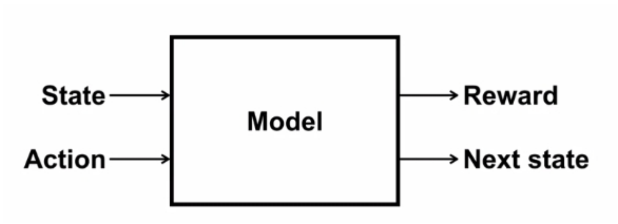
Definition 3.2. An MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$ where:

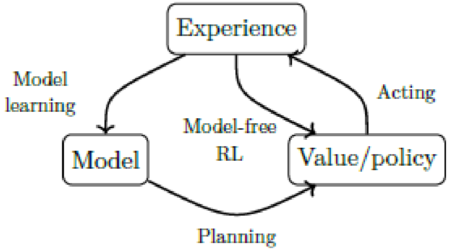
- $\mathcal{S}$ is the state space,
- $\mathcal{A}$ is the action space,
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition function (set of conditional transition probabilities between states),
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R}$ is the reward function, where $\mathcal{R}$ is a continuous set of possible rewards in a range $R_{\max} \in \mathbb{R}^+$ (e.g., $[0, R_{\max}]$),
- $\gamma \in [0, 1)$ is the discount factor.

Introduction

❏ Q-Learning

- Model - Free Algorithm
  - ❖ model
    - (1) transition model
    - (2) returns probability of being next state given current state and action
  - ❖ model - free
    - (1) unknown environment
    - (2) return observations manually
    - (3) Policy Optimization(on-policy) vs **Q Learning**(off-policy)





Introduction

❑ Q-Learning

- Value Function

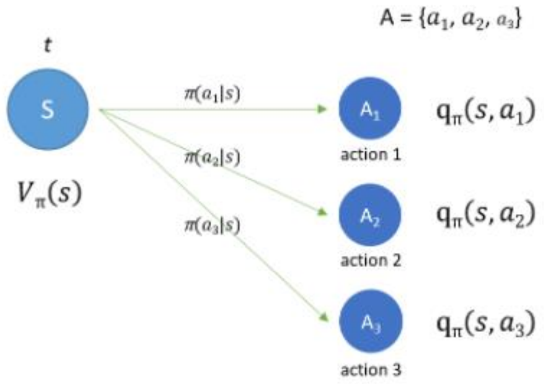
  ❖ State - value function

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

  ❖ Action - value function

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

$$\boxed{\begin{aligned} V^*(s) &= \max_{\pi \in \Pi} V^{\pi}(s). \\ Q^*(s,a) &= \max_{\pi \in \Pi} Q^{\pi}(s,a). \\ \pi^*(s) &= \operatorname*{argmax}_{a \in \mathcal{A}} Q^*(s,a). \end{aligned}}$$



* Policy : mapping func. (s → a)

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) * q_\pi(s,a)$$

Introduction

- Q-Learning
  - Bellman Equation

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

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

    - ★ Bellman Operator

$$(\mathcal{B}K)(s,a) = \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a' \in \mathcal{A}} K(s',a')\right)$$

for any policy π any initial vector v,

$$\lim_{k\to\infty} (\tau^{\pi})^k = v_{\pi}, \quad \lim_{k\to\infty} (\tau^{*})^k = v_{*}$$

where v_π is the value of policy π
and v_* is the value of an optimal policy π_*

[1]The Bellman operator is a contraction mapping because it can be shown that for any pair of bounded functions $K, K' : S \times A \to \mathbb{R}$, the following condition is respected:

$$\|TK - TK'\|_{\infty} \leq \gamma \|K - K'\|_{\infty}.$$

Introduction

- Q-Learning
  - Algorithm
    - Iterative Method
      (1) Initialize Q(s,a) with arbitrary fixed values
      (2) Select current action and state
      (3) Observe reward and new state
      (4) From Bellman Equation, get a new target value and update Q(s,a)
      (5) Check if Q(s,a) is converged.
      (6) If not, repeat the process (2)

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

Introduction

- Q-Learning

$$Q(s,a) := r(s,a) + \gamma \max_a Q(s',a) \longrightarrow Q(s,a) = r(s,a)$$

Game Board:

Current state (s): 000 010

Q Table: $\gamma = 0.95$

	000 100	000 010	000 001	100 000	010 000	001 000
↑	0.2	0.3	1.0	-0.22	-0.3	0.0
↓	-0.5	-0.4	-0.2	-0.04	-0.02	0.0
→	0.21	0.4	-0.3	0.5	1.0	0.0
←	-0.6	-0.1	-0.1	-0.31	-0.01	0.0

Introduction

❑ Fitted Q Learning

Q - Learning(Bellman Equation for action-value function)

Define target value as $Y_k^Q = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k)$ ← $Q(s,a) := r(s,a) + \gamma \max_a Q(s', a)$

The Q values are parameterized with a network Q(s,a,θ) where parameters θ are updated by stochastic gradient descent by minimizing the objective functions L

$$L_{obj} = \left(Q(s,a;\theta_k) - Y_k^Q\right)^2$$

The parameters θ are updated as below

$$\theta_{k+1} = \theta_k + \alpha \left(Y_k^Q - Q(s,a;\theta_k)\right) \nabla_{\theta_k} Q(s,a;\theta_k)$$

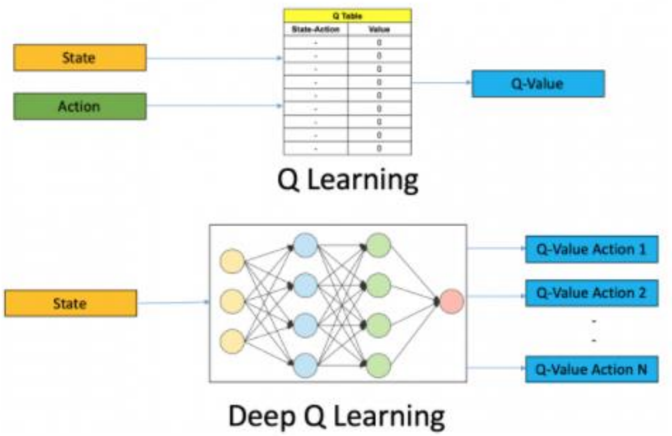
Problem

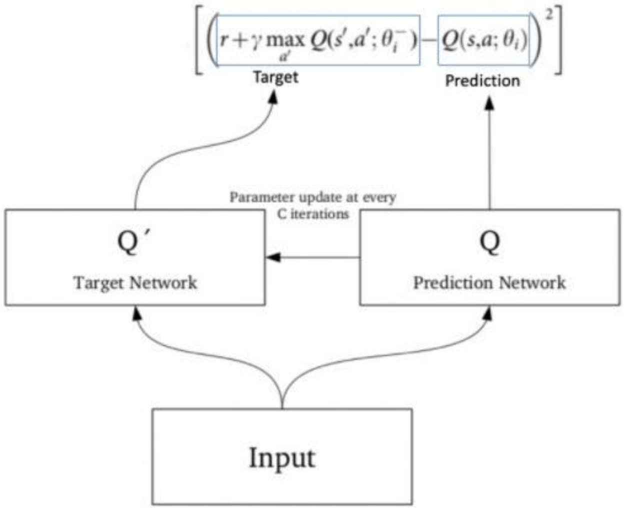
1. Bellman Operator is not enough to guarantee convergence due to the change of target for each update process
2. Q values tend to be overestimated due to the max operator => DDQN

Deep Reinforcement Learning

❑ Deep Q Learning

- Neural Network as a function approximator





Deep Reinforcement Learning

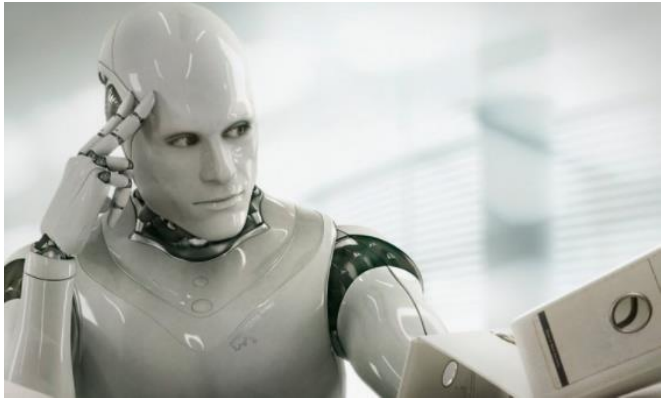
❏ Deep Q Learning

- Issue

Q-Learning Loss Term diverges due to….

1. Correlations between samples
2. Non-stationary targets(update with a function approximator)

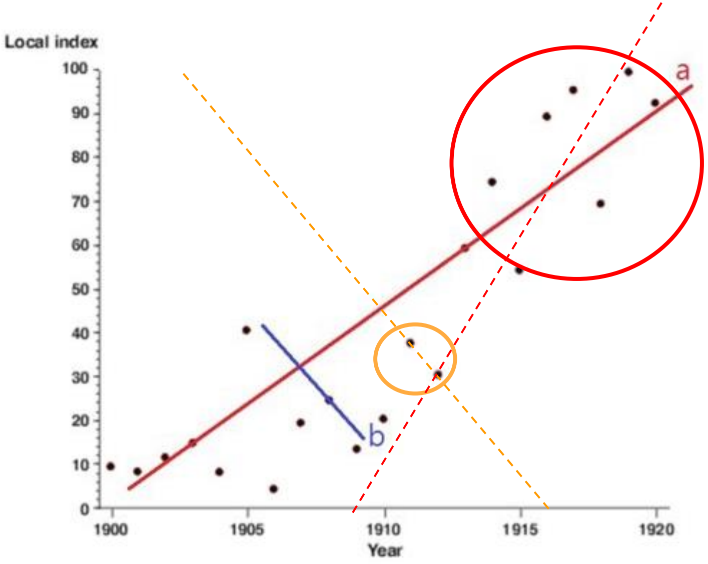
DeepMind



Deep Reinforcement Learning

❑ Replay Memory

- Generate Experience Replay Memory for each episode and extract random sample
- For each episode, store the agent’s experiences with state-action pairs (s,a,r,s) and sample a random batch
- available to get a subset within which the correlation among the samples is low and provide better sampling efficiency




```python
Transition = namedtuple(
    'Transition',
    ('state', 'action', 'next_state', 'reward')
)

# Batch 단위로 데이터를 샘플링하여 학습에 필요한 batch 데이터를 생성
# queue 구조를 이용
class ReplayMemory(object):
    def __init__(self, capacity):
        self.memory = deque([], maxlen = capacity)

    def push(self, *args):
        '''
        transition data 저장
        '''
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Deep Reinforcement Learning

❑ Non stationary targets

- Create a target Q network which is separated from a prediction Q network
- Deep Q-Learning process
  1. (1) update new value from prediction Q network
  2. (2) get target value from the fixed target Q network
  3. (3) update weights from prediction Q network
  4. (4) load weights from prediction Q network to target Q network for every n steps
  5. (5) repeat

From prediction Q network

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

From target Q network

Deep Reinforcement Learning

- Algorithm

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

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-)$$

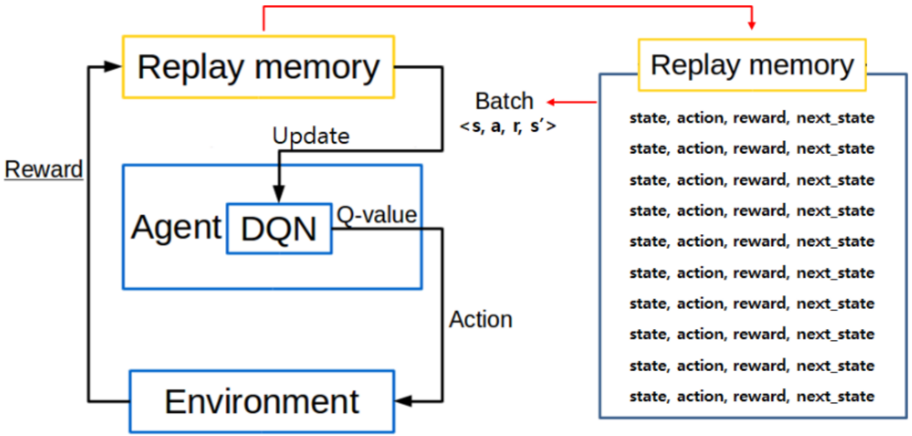
$$\mathrm{L}_{DQN} = \left(Q(s, a; \theta_k) - Y_k^Q\right)^2$$

$$\theta_{k+1} = \theta_k + \alpha \left(Y_k^Q - Q(s, a; \theta_k)\right) \nabla_{\theta_k} Q(s, a; \theta_k).$$

Deep Reinforcement Learning

❑ Algorithm

```python
for t in count():
    state = state.to(device)
    action = select_action(state, policy_net, device)
    _, reward, done, _ = env.step(action.item())

    reward = torch.tensor([reward], device = device)
    last_screen = current_screen
    current_screen = get_screen(env)

    if not done:
        next_state = current_screen - last_screen

    else:
        next_state = None

    # memory에 transition 저장
    memory.push(state, action, next_state, reward)

    state = next_state

    # policy_net -> optimize
    optimize_model()

    if done:
        episode_durations.append(t+1)
        break

if i_episode % TARGET_UPDATE == 0:
    target_net.load_state_dict(policy_net.state_dict())
```



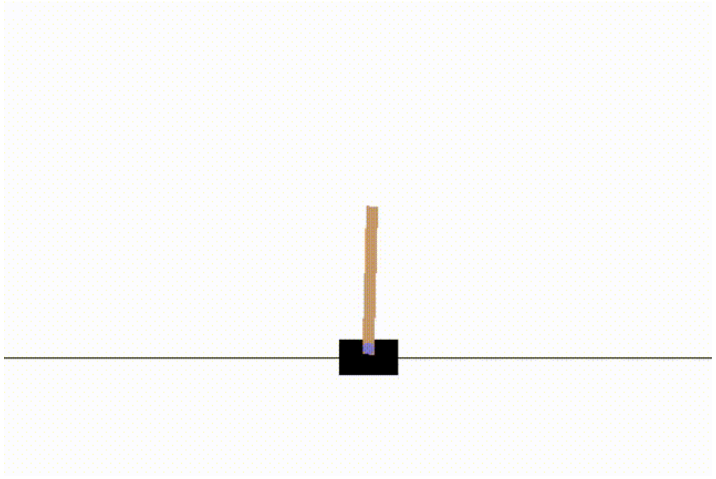

```python
# Q(s_t, a) computation
# tensor -> gather(axis = 1, action_batch) -> tensor에서 각 행별 인덱스에 대응되는 값 호출
state_action_values = policy_net(state_batch).gather(1, action_batch)

next_state_values = torch.zeros(BATCH_SIZE, device = device)

next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()

expected_state_action_values = (next_state_values * GAMMA) + reward_batch

criterion = nn.SmoothL1Loss() # Huber Loss
loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))
```

Deep Reinforcement Learning

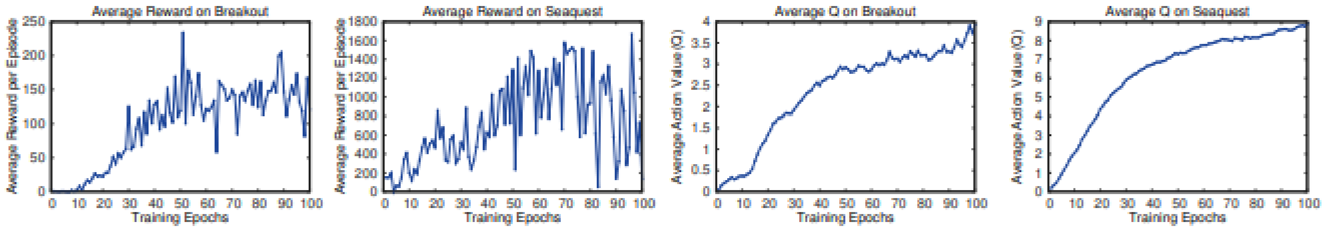
- Model Architecture

Layer (type)	Input Shape	Param #	Tr. Param #
Conv2d-1	[1, 3, 40, 90]	1,216	1,216
BatchNorm2d-2	[1, 16, 18, 43]	32	32
Conv2d-3	[1, 16, 18, 43]	12,832	12,832
BatchNorm2d-4	[1, 32, 7, 20]	64	64
Conv2d-5	[1, 32, 7, 20]	25,632	25,632
BatchNorm2d-6	[1, 32, 2, 8]	64	64
Linear-7	[1, 512]	1,026	1,026

Total params: 40,866
Trainable params: 40,866
Non-trainable params: 0



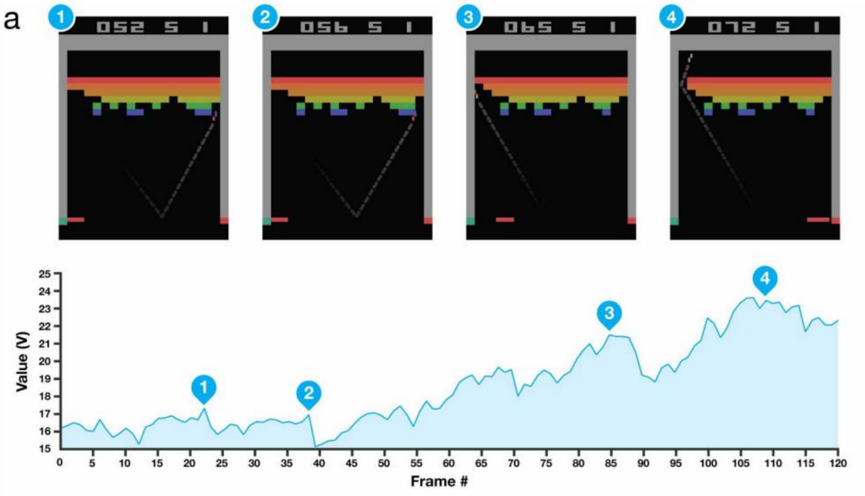
Experiment and Result

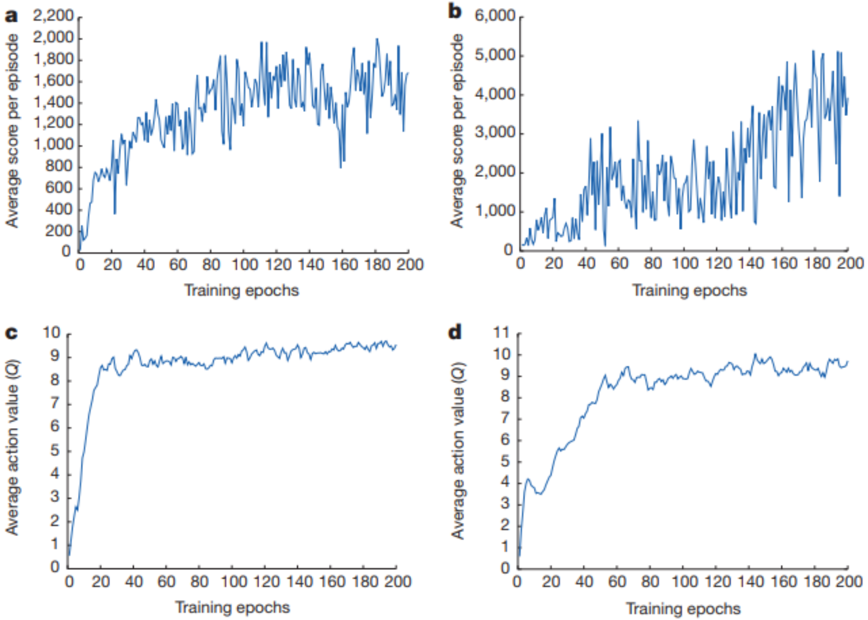


	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Experiment and Result

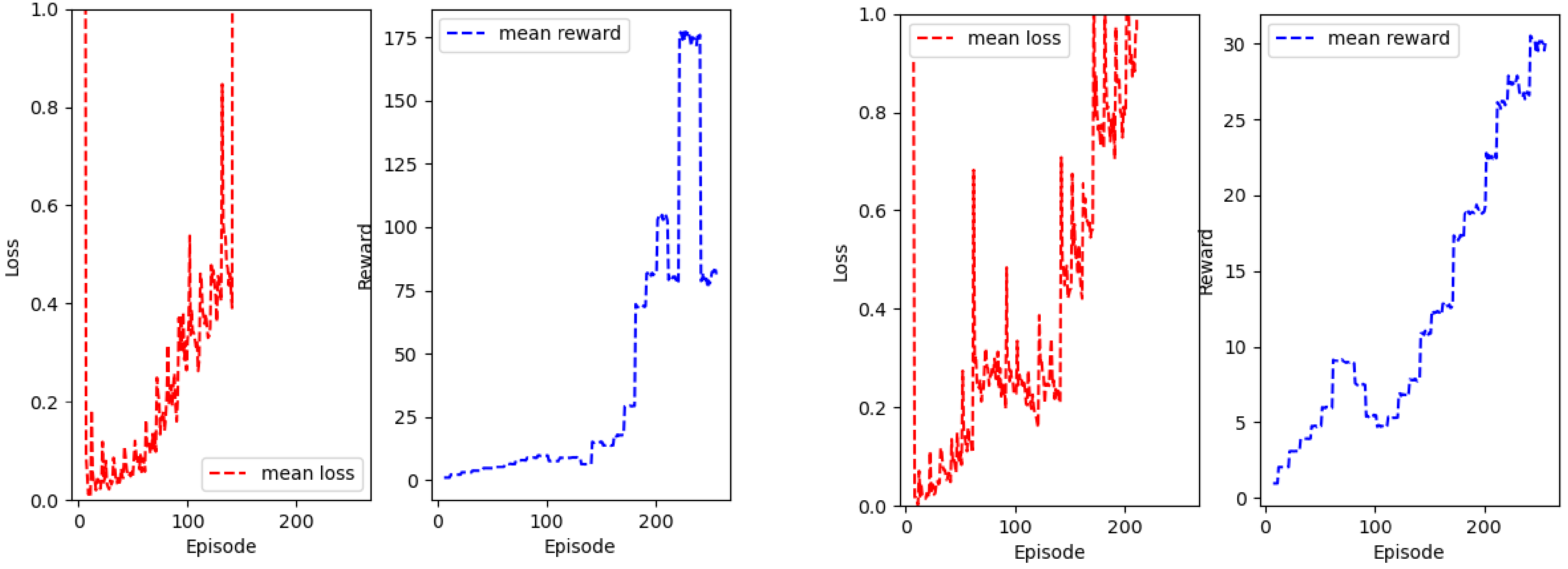
(+) Human-level control through deep reinforcement learning(nature, 2015)





Experiment and Result

(+) Custom(OpenAI gym cart-pole v0), left : DQN, right : DDQN



Extra

❑ Double DQN

- Decoupling the action selection from the action evaluation

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

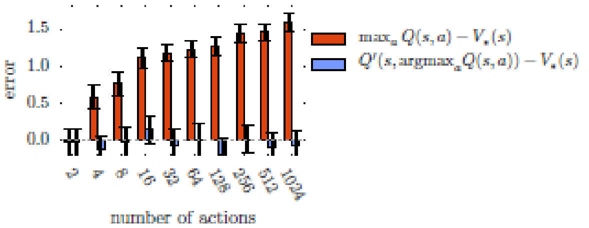
$$\downarrow$$

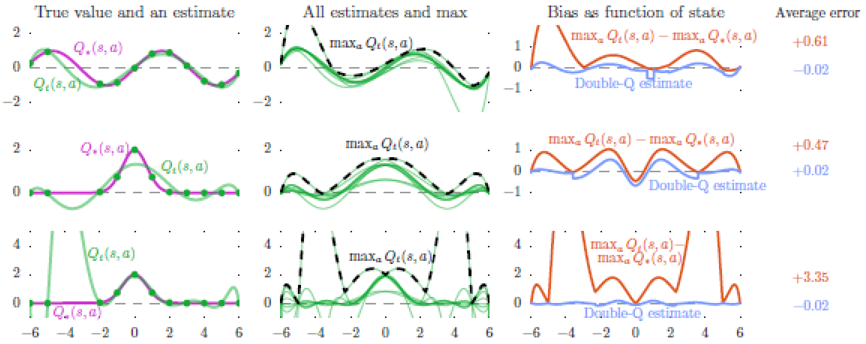
$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)$$

Theorem 1. *Consider a state s in which all the true optimal action values are equal at $Q_*(s,a) = V_*(s)$ for some $V_*(s)$. Let Q_t be arbitrary value estimates that are on the whole unbiased in the sense that $\sum_a (Q_t(s,a) - V_*(s)) = 0$, but that are not all zero, such that $\frac{1}{m}\sum_a (Q_t(s,a) - V_*(s))^2 = C$ for some $C > 0$, where $m \geq 2$ is the number of actions in s. Under these conditions, $\max_a Q_t(s,a) \geq V_*(s) + \sqrt{\frac{C}{m-1}}$. This lower bound is tight. Under the same conditions, the lower bound on the absolute error of the Double Q-learning estimate is zero.*

Theorem 2. *Consider a state s in which all the true optimal action values are equal at $Q_*(s,a) = V_*(s)$. Suppose that the estimation errors $Q_t(s,a) - Q_*(s,a)$ are independently distributed uniformly randomly in $[-1, 1]$. Then,*

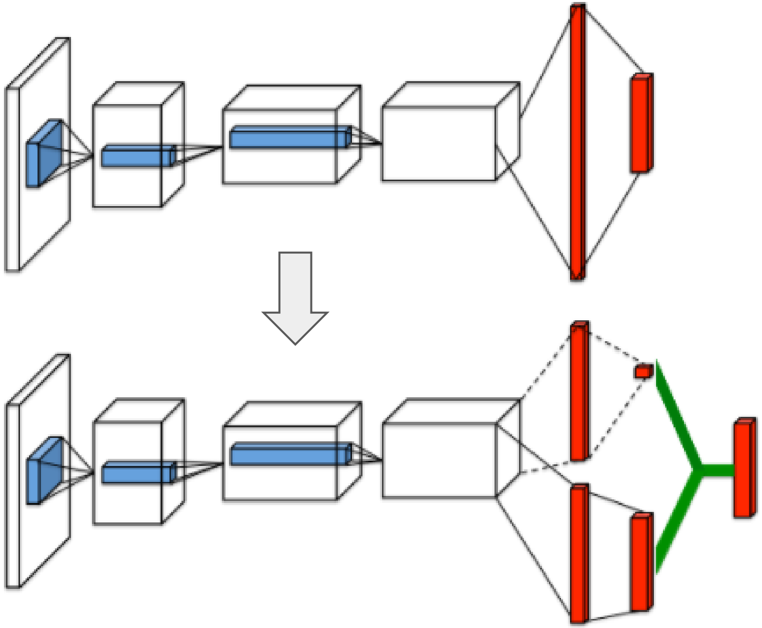
$$\mathbb{E}\left[\max_a Q_t(s,a) - V_*(s)\right] = \frac{m-1}{m+1}$$





Extra

❑ Dueling Network Architecture



- Decoupling the value and advantage function

$$Q(s,a;\theta^{(1)},\theta^{(2)},\theta^{(3)}) = V\left(s;\theta^{(1)},\theta^{(3)}\right) + \left(A\left(s,a;\theta^{(1)},\theta^{(2)}\right) - \max_{a'\in\mathcal{A}} A\left(s,a';\theta^{(1)},\theta^{(2)}\right)\right)$$

- Different approach to increase stability of the optimization

$$Q(s,a;\theta^{(1)},\theta^{(2)},\theta^{(3)}) = V\left(s;\theta^{(1)},\theta^{(3)}\right) + \left(A\left(s,a;\theta^{(1)},\theta^{(2)}\right) - \frac{1}{|\mathcal{A}|}\sum_{a'\in\mathcal{A}} A\left(s,a';\theta^{(1)},\theta^{(2)}\right)\right)$$

Profit

1. prevent overestimation of Q value function approximation
2. increase stability of optimization

Extra

❑ Policy Gradient Method

- Policy Gradient Theorem

$$V^{\pi}(s_0) = \int_{\mathcal{S}} \rho^{\pi}(s) \int_{\mathcal{A}} \pi(s,a) R'(s,a) dads. \quad \begin{cases} R'(s,a) = \int_{s' \in \mathcal{S}} T(s,a,s') R(s,a,s') \\ \rho^{\pi}(s) = \sum_{t=0}^{\infty} \gamma^t Pr\{s_t = s | s_0, \pi\}. \end{cases}$$

$$\downarrow$$

$$\nabla_w V^{\pi_w}(s_0) = \int_{\mathcal{S}} \rho^{\pi_w}(s) \int_{\mathcal{A}} \nabla_w \pi_w(s,a) Q^{\pi_w}(s,a) dads.$$

$$\downarrow \quad \boxed{\text{REINFORCE Algorithm}}$$

$$\nabla_w V^{\pi_w}(s_0) = \mathbb{E}_{s \sim \rho^{\pi_w}, a \sim \pi_w} \left[\nabla_w \left(\log \ \pi_w(s,a) \right) Q^{\pi_w}(s,a) \right]$$

1. Initialize the policy parameter θ at random.
2. Generate one trajectory on policy π_θ: $S_1, A_1, R_2, S_2, A_2, \ldots, S_T$.
3. For t=1, 2, ... , T:
    1. Estimate the the return G_t;
    2. Update policy parameters: $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t | S_t)$

Reference

Paper

- Human-level control through deep reinforcement learning(nature, 2015)
- Playing Atari with Deep Reinforcement Learning, Mnih et al, 2013
- Dueling Network Architectures for Deep Reinforcement Learning, Wang et al, 2015
- Deep Reinforcement Learning with Double Q-learning, Hasselt et al 2015

Blog

- https://jeinalog.tistory.com/20
- https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677
- https://sumniya.tistory.com/18
- https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

End