Playing Atari with Deep Reinforcement Learning

백 상 원

Abstract

- 강화학습에 딥러닝을 적용한 첫 모델
- 고차원 Sensory input(Vision or Speech)을 직접학습
- Q러닝 + CNN 결합
- Raw Sensory data에서 스스로 feature를 추출

Reinforcement Learning?

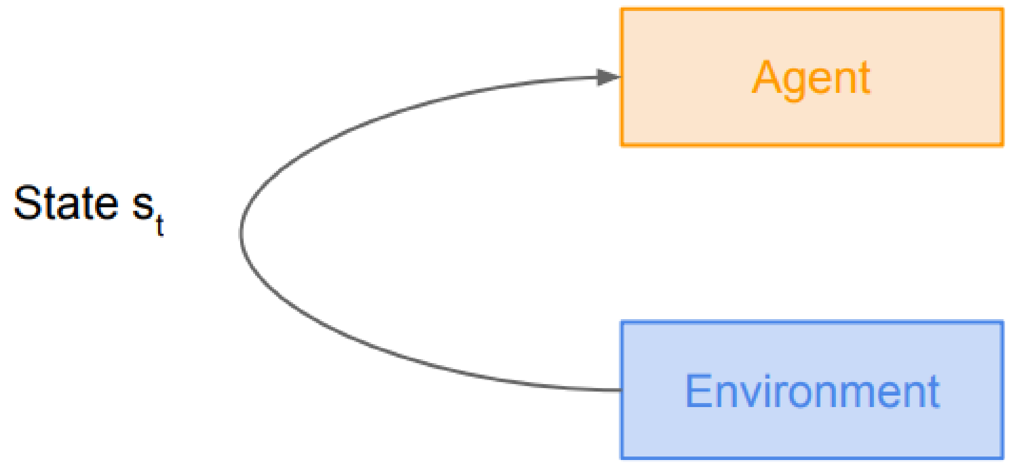
- 강화학습이란, 주어진 상황(state)에서 학습의 주체(agent)가 보상(reward)을 최대화 할 수 있는 행동(action)을 학습

- 환경과 상호작용을 통해 얻은 보상으로부터 학습

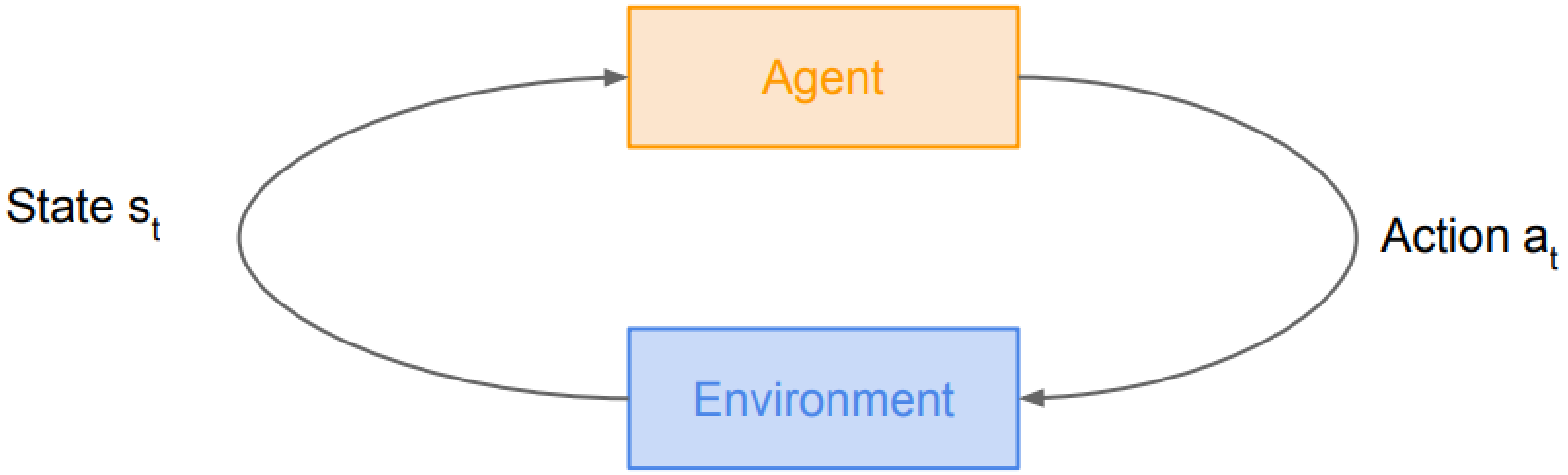
- 현재 선택한 행동이 미래의 보상에 영향

Reinforcement Learning

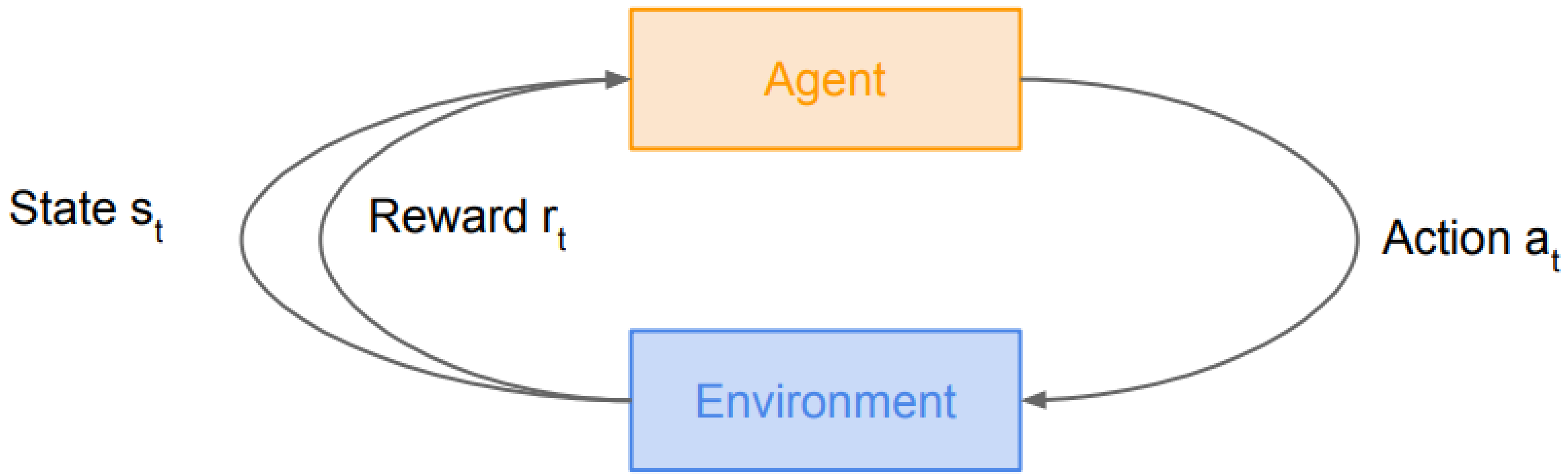
Reinforcement Learning



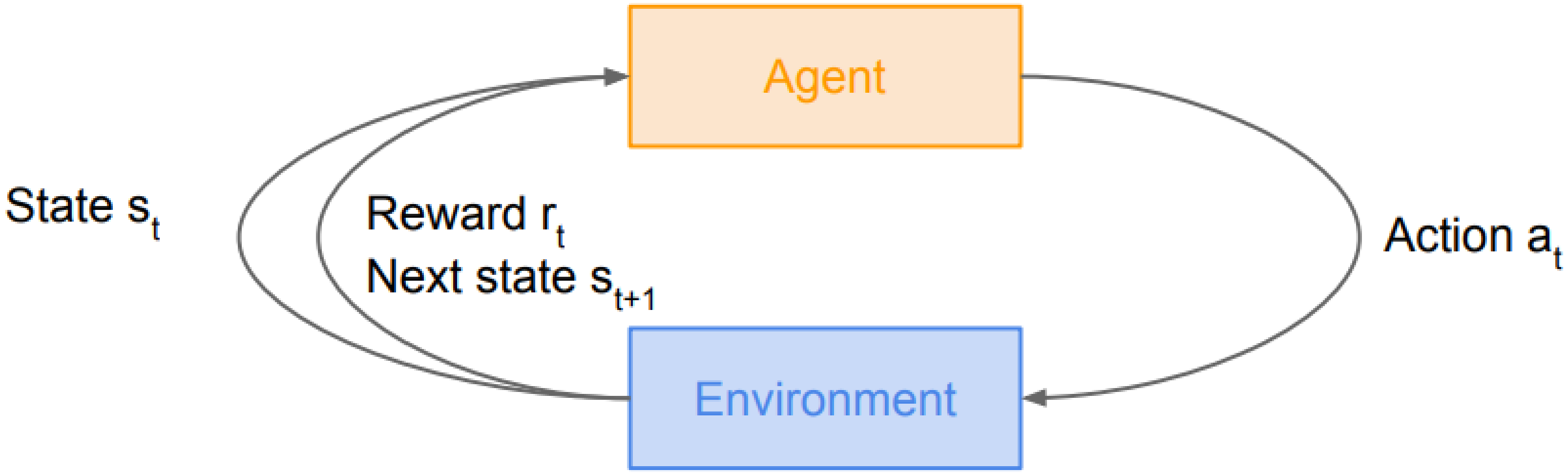
Reinforcement Learning



Reinforcement Learning



Reinforcement Learning



Markov Decision Process

- 상태(S) : 상태(state)의 집합
- 행동(A) : 행동(action)의 집합
- 전이 확률(P) : 현재 상태(state)에서 행동(action)하였을 때 다음 상태(state)로 이동할 확률
- 보상(R) : 현재 상태(state)에서 행동(action)하였을 때 얻을 수 있는 기대 보상
- 할인계수(γ) : 즉각적으로 얻는 보상(reward)와 미래에 얻을 수 있는 보상(reward)간의 중요도 조절
- 정책(π) : 각 상태에서의 행동 규칙확률

Q-Learning

$$Q^{\pi}(s,a) = E[R_t | s_t = s, a_t = a, \pi]$$

$$= E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, a_t = a, \pi]$$

정책(π)에 따라 현재 상태(s)에서

행동(a)하였을 때 기대되는 누적 보상을 $Q^{\pi}(s,a)$로 정의

Q-Learning

$$Q^*(s,a) = max_\pi E[R_t | s_t = s, a_t = a, \pi]$$

$$= max_\pi E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, a_t = a, \pi]$$

$Q(s,a)$를 최대화하는 최적의 정책(π^*)을 찾는 것이 목표

$$Q^*(s,a) = E_{s_{t+1} \sim \varepsilon}\left[r_t + \gamma max_{a_{t+1}} Q^*(s_{t+1,} a_{t+1}) | s_t = s, a_t = a\right]$$

Bellman equation

Q-Learning

$$Q_{i+1}(s,a) = E_{s_{t+1}\sim\varepsilon}\left[r_t + \gamma max_{a_{t+1}} Q_i\,(s_{t+1,}a_{t+1}) | s_t = s, a_t = a\right]$$

Bellman equation을 반복적으로 업데이트

$$i \;\rightarrow\; \infty \qquad\qquad Q_i \;\rightarrow\; Q^*$$

만약, 무한대로 반복 　　　　 최적의 값으로 수렴

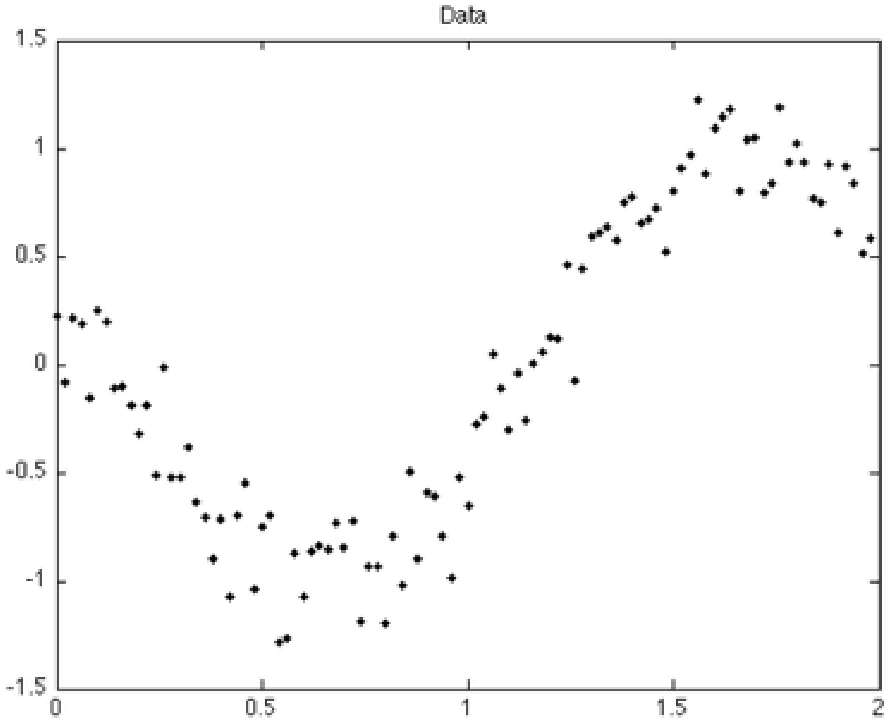
$$Q^*(s,a) = E_{s_{t+1}\sim\varepsilon}\left[r_t + \gamma max_{a_{t+1}} Q^*(s_{t+1,}a_{t+1}) | s_t = s, a_t = a\right]$$

Problem 1

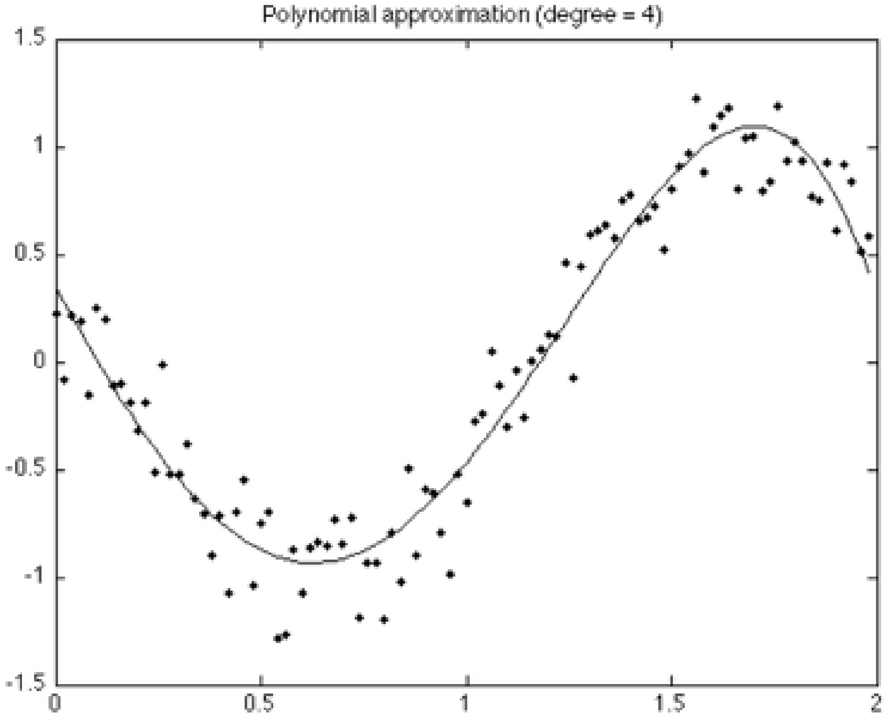
- 방대한 양의 상태와 행동이 존재하면 적용 불가능

- 일반화 성능 문제

Problem 1



Data
1.5
1
0.5
0
-0.5
-1
-1.5
0
0.5
1
1.5
2



Polynomial approximation (degree = 4)
1.5
1
0.5
0
-0.5
-1
-1.5
0
0.5
1
1.5
2

Deep Q-Learning(DQN)

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

Q-network는 Weight가 θ 인 Neural Network 근사 함수를 Q-fuction에 적용한 것

$$Q_{i+1}(s,a) = E_{s_{t+1}\sim\varepsilon}\left[r_t + \gamma max_{a_{t+1}} Q_i\,(s_{t+1,}a_{t+1})|s_t = s, a_t = a\right]$$

Bellman equation을 반복적으로 업데이트

$i \;\rightarrow\; \infty$ ⇨ $Q_i \;\rightarrow\; Q^*$

만약, 무한대로 반복 최적의 값으로 수렴

$$L_i(\theta_i) = E_{s_t,a_t\sim\rho(\cdot);s_{t+1}\sim\varepsilon}\left[\left(r_t + \gamma max_{a_{t+1}} Q\left(s_{t+1,}a_{t+1};\theta_{i-1}\right) - Q(s_t,a_t;\theta_i)\right)^2\right]$$

Deep Q-Learning(DQN)

$$L_i(\theta_i) = E_{s_t, a_t \sim \rho(\cdot); s_{t+1} \sim \varepsilon}\left[\left(r_t + \gamma max_{a_{t+1}} Q\left(s_{t+1,} a_{t+1}; \theta_{i-1}\right) - Q(s_t, a_t; \theta_i)\right)^2\right]$$

Gradient Desent

$$\nabla_{\theta_i} L_i(\theta_i) = E_{s_t, a_t \sim \rho(\cdot); s_{t+1} \sim \varepsilon}\left[\left(r_t + \gamma max_{a_{t+1}} Q\left(s_{t+1,} a_{t+1}; \theta_{i-1}\right) - Q(s_t, a_t; \theta_i)\right) \nabla_{\theta_i} Q(s_t, a_t; \theta_i)\right]$$

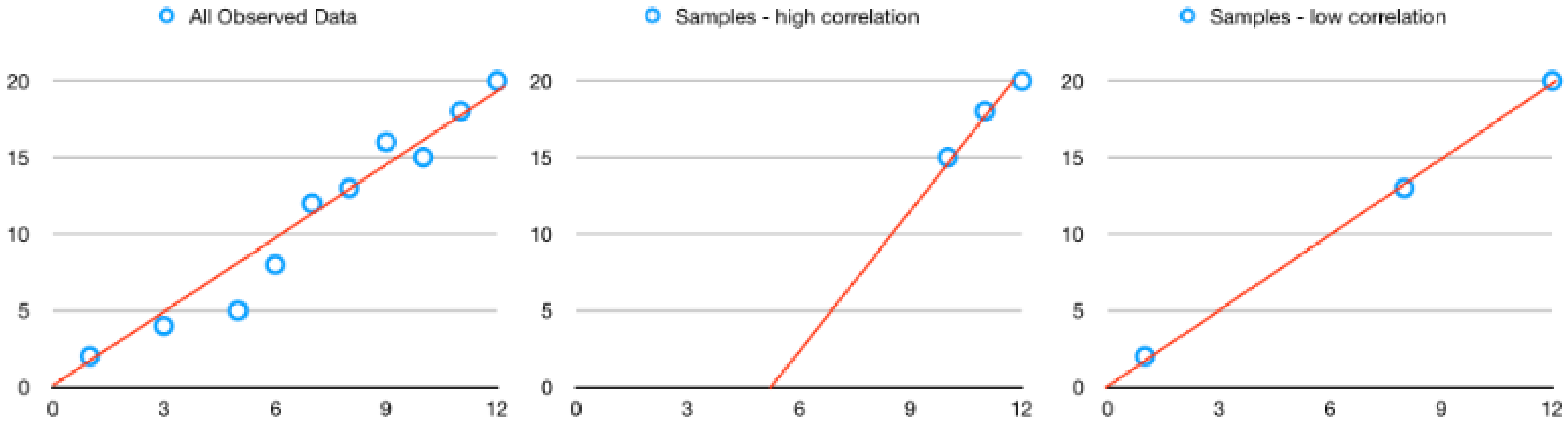
매 time-step&새로운 샘플이 들어올 때 마다 업데이트

Problem 2

- Neural Network가 잘되는 이유는 방대한 양의 데이터가 있어서 가능한데 강화 학습은 데이터 부족

- 데이터간의 높은 Correlation

Problem 2


All Observed Data
Samples - high correlation
Samples - low correlation
20
15
10
5
0
0
3
6
9
12

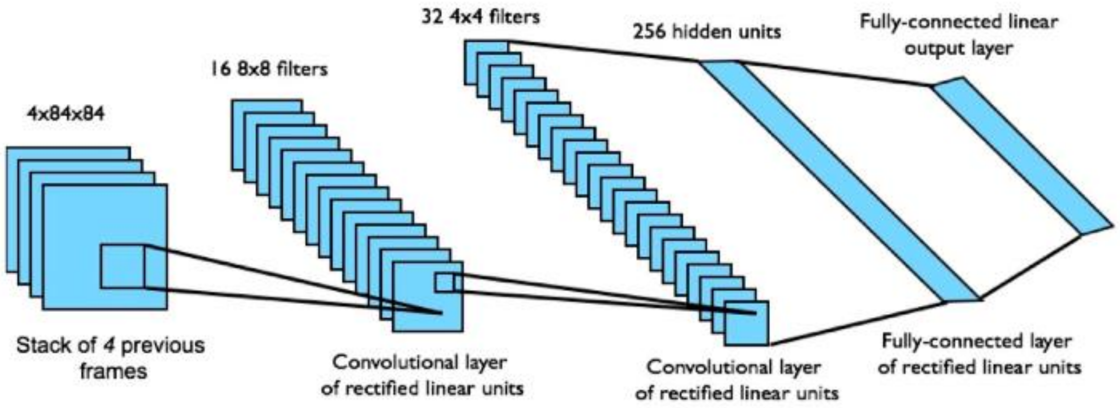
Experience Replay

- 입력데이터 간의 correlation을 줄이기 위한 방법

- Agent의 경험 $e_t = (s_t, a_t, r_t, s_{t+1})$ 를 time-step 단위로 Data set $D_t = \{e_1, \cdots, e_t\}$에 저장

- Data set에서 Random-Sampling을 통해 mini-batch단위로 데이터를 뽑아서 업데이트

Network




```python
def build_model(self):
    model = Sequential()
    model.add(Conv2D(32, (8, 8), strides=(4, 4), activation='relu',
                     input_shape=self.state_size))
    model.add(Conv2D(64, (4, 4), strides=(2, 2), activation='relu'))
    model.add(Conv2D(64, (3, 3), strides=(1, 1), activation='relu'))
    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dense(self.action_size))
    model.summary()
    return model
```

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

Reply memory
초기화 및 용량설정

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

신경망 Random Weight

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

```python
def pre_processing(observe):
    processed_observe = np.uint8(
        resize(rgb2gray(observe), (84, 84), mode='constant') * 255)
    return processed_observe
```

```python
state = pre_processing(observe)
history = np.stack((state, state, state, state), axis=2)
history = np.reshape([history], (1, 84, 84, 4))
```

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

```python
action = agent.get_action(history)

if action == 0:
    real_action = 1
elif action == 1:
    real_action = 2
else:
    real_action = 3
```

```python
def get_action(self, history):
    history = np.float32(history / 255.0)
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)
    else:
        q_value = self.model.predict(history)
        return np.argmax(q_value[0])
```

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

```
next_state = pre_processing(observe)
next_state = np.reshape([next_state], (1, 84, 84, 1))
next_history = np.append(next_state, history[:, :, :, :3], axis=3)
```

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 $\mathcal{D}$ $\longrightarrow$ `agent.append_sample(history, action, reward, next_history, dead)`
    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

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

```python
def train_model(self):
    if self.epsilon > self.epsilon_end:
        self.epsilon -= self.epsilon_decay_step

    mini_batch = random.sample(self.memory, self.batch_size)

    history = np.zeros((self.batch_size, self.state_size[0],
                        self.state_size[1], self.state_size[2]))
    next_history = np.zeros((self.batch_size, self.state_size[0],
                             self.state_size[1], self.state_size[2]))
    target = np.zeros((self.batch_size,))
    action, reward, dead = [], [], []

    for i in range(self.batch_size):
        history[i] = np.float32(mini_batch[i][0] / 255.)
        next_history[i] = np.float32(mini_batch[i][3] / 255.)
        action.append(mini_batch[i][1])
        reward.append(mini_batch[i][2])
        dead.append(mini_batch[i][4])

    target_value = self.target_model.predict(next_history)
```

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

```
for i in range(self.batch_size):
    if dead[i]:
        target[i] = reward[i]
    else:
        target[i] = reward[i] + self.discount_factor * \
                                np.amax(target_value[i])

loss = self.optimizer([history, action, target])
self.avg_loss += loss[0]
```

Conclusion

	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