

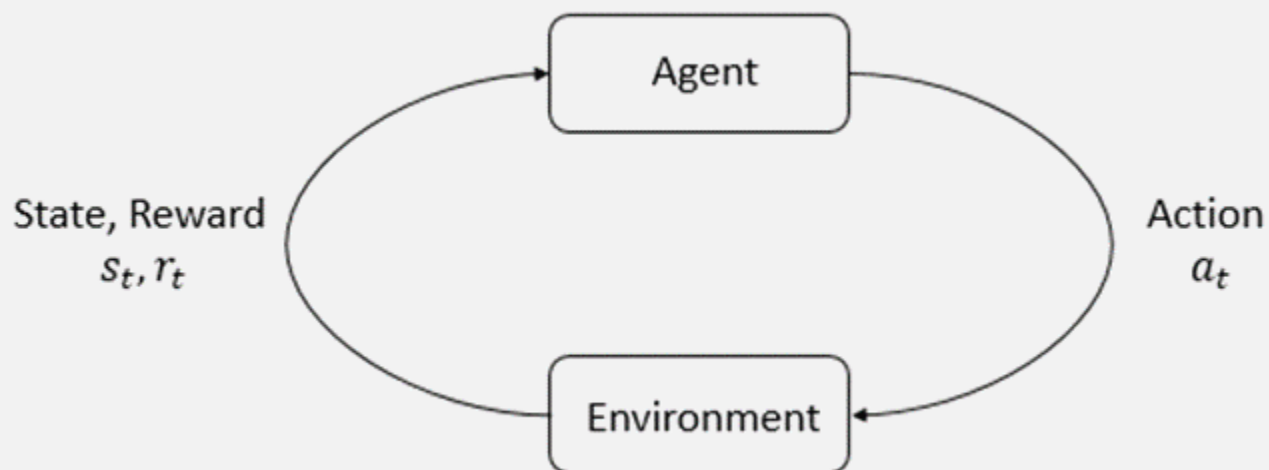
Q-learning

201810800 이혜인

1

Q-learning

- Model이 없이(Model-Free) 학습하는 **강화학습 알고리즘**
- 강화 학습이란 어떤 환경 안에서 정의된 Agent가 현재의 상태를 인식하여, 선택 가능한 Action들 중 Reward를 최대화하는 Action 혹은 Action 순서를 선택하는 방법



- Agent가 행동(action)을 선택했을 때 얻는 보상(reward)의 기대 값(Q-value)을 예측하는 Q 함수의 학습을 통해 최적의 정책(policy)을 학습
- Agent가 특정 상황에서 특정 행동을 하라는 최적의 Policy를 배우는 것
- 즉, Reward를 통해 현재 State에서 어떤 Action을 취하는게 좋은지 학습하는 과정으로 이해
 - 정책은 주어진 상태(state)에서 의사결정자가 어떤 행동을 선택할지 나타내는 규칙

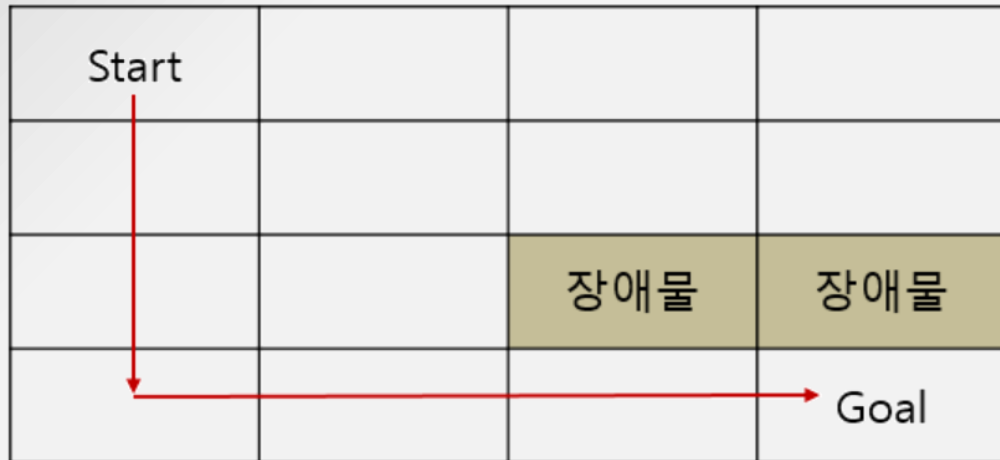
- Q-learning은 현재 State로부터 시작하여 연속적인 State들을 모두 거쳤을 때 얻게 되는 전체 Reward의 예측 값을 극대화
- 즉, 어떤 action을 취하는게 좋은지에 대해서는 각 action마다 그 action이 취했을 때 얼마나 좋은지를 측정하는 값(Q-value) 학습하고, Q-value가 높으면 그 action을 선택했을 때 더 좋은 reward를 받는다고 이해할 수 있죠. 즉, 선택은 action에 Q-value 중 가장 높은 Q-value를 가지는 action을 취하는 것

1

Q-learning – Q-value

- Q-value는 어떤 시간 t 에서 전략(policy)에 따라 어떤 Action a 를 했을 때, 미래 보상의 총합의 기대 값
- Q-Learning에서는 어떤 State S 에서 어떤 Action A 를 했을 때, 그 행동이 가지는 Value를 계산하는 $Q(s, a)$ 함수를 사용

- Q-learning



Start			
		장애물	장애물
			Goal

- Start에서 취할 수 있는 행동은 위, 아래, 오른쪽, 왼쪽
- 행동 : Action, 위치 : State
- Start에서 왼쪽이나 위로 가는 Action은 좋은 행동이 아님 → 이를 판단할 근거(Reward) 필요

1

Q-learning

- Reward 설정

Start			
		장애물	장애물
			Goal

- Reward 설정 : 이동했을 때 벽에 부딪히면 -2, 아니면 -1, 장애물이 있다면 -10이라 가정
- Start에서는 아래 or 오른쪽이 좋다고 학습
- 즉, 각 State마다 최악의 Action을 제외한 나머지 Action을 선택하도록 학습 → 지금 당장은 좋은 Action이지만 먼 미래에도 좋은 Action인가? NO

1

Q-learning

- Reward 설정

Start			
		↓	↓
		장애물	장애물
			Goal

- 먼 미래에 좋은 결과를 내는 Action을 선택하도록 Discount factor 도입
- 파란색 화살표처럼 장애물에 도달할 확률이 있다는 것을 feedback으로 주는 것
- 각 State가 받는 Reward : 현재 받을 수 있는 Reward + 미래 보상의 합

1

Q-learning

- Reward 설정



- optimal policy : 모든 state에서 Q-value를 최대로 하는 action 선택

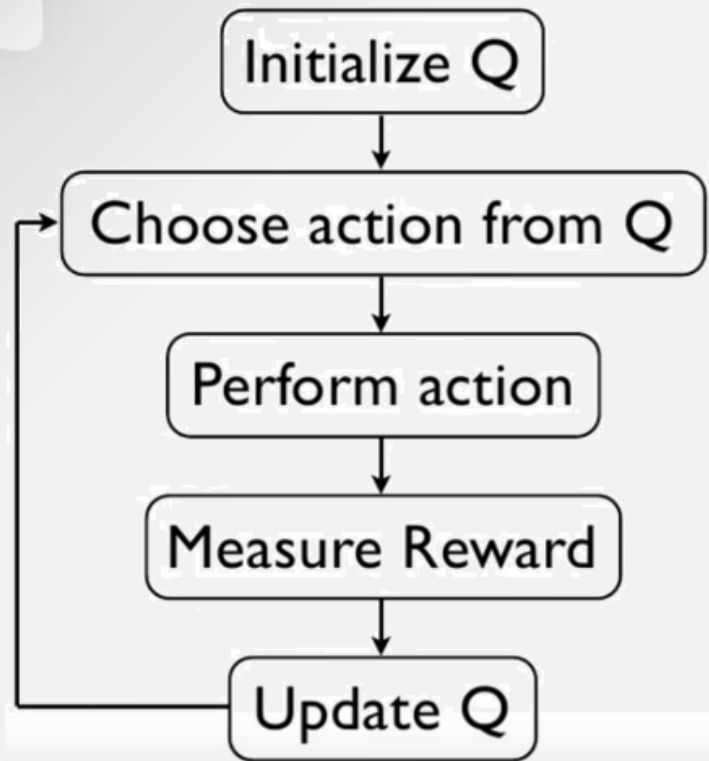
1

Q-learning

- 다음 식은 $Q(\text{state}_t, \text{action}_k)$ 를 갱신하는 공식
- 현재 State의 action에 대한 Q-value값은 현재 받는 reward와 다음 state에서의 maximum Q-value값 * discount factor로 update
- 처음에는 random한 action을 취하게 되지만, 이 과정을반복하게 된다면, optimum policy를 얻을 수 있게 되는 것

$$Q(\text{state}_t, \text{action}_k) = (1 - \eta) Q(\text{state}_t, \text{action}_k) + \eta (R(\text{state}_t, \text{action}_k) + \max Q(\text{state}_{t+1}, \text{action}_u))$$

Q-learning



- 알고리즘이 시작되기 전에 Q 함수는 고정된 임의의 값을 가짐
- 매 time-step(t)마다 Agent는 Action선택
- Reward를 받으며 새로운 State로 전이하고, Q 값이 갱신
- 이전의 값과 새로운 정보의 weighted sum을 이용하는 Value Iteration Update 기법

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

The equation shows the Q-learning update rule. The left side is the new Q value for state s_t and action a_t . The right side is the weighted sum of the old Q value and the immediate reward plus the discounted maximum future Q value. The terms are annotated with their roles: 'old value', 'learning rate', 'reward', 'discount factor', and 'estimate of optimal future value'.

- Q-learning에서 State마다 Action을 취하게 되는데, Action 을 취할 때 “가끔” random하게 선택함
- 이유는 가보지 않을 곳을 탐험 하면서 새로운 좋은 경로를 찾을 수 있게끔 하는 것인데, 이를 Exploration이라고 정의하고, E-greedy 방법을 통해 수행

- 즉, 0~1사이로 random하게 난수를 추출해서 그게 특정 threshold보다 낮으면 random하게 action을 취하고, 이 threshold는 episode가 반복되면서(학습이 진행되면서) 점점(greedy)하게 낮춤
- 학습이 계속 진행되면 threshold값은 거의 0에 수렴
- Action을 취하고 Reward를 받고 다음 State를 받고 그리고 현재 State와 Action에 대한 Q-value를 update하는 과정을 무수히 반복

- Exploration
- Policy의 action 이외에 다른 action을 선택하는 것이 더 큰 이득을 얻을 수 있는 가능성 존재

$$Q(\text{state}, \text{action})$$

$$= R(\text{state}, \text{action}) + \text{Gamma} * \text{Max}[Q(\text{next state}, \text{all actions})]$$

- Exploitation
- Policy대로만 action을 선택

Q-learning

Frozen_lake – v0

1

Q-learning - FrozenLake-v0

- Structure : 4 X 4 grid
- State : Start / Goal / Safe-Frozen / hole (4가지)
- Reward : Goal state 를 지날 때 1, 그 외의 step은 0



1

Q-learning - FrozenLake-v0

- Objective : agent가 시작 블록에서 목표 블록으로 구멍을 피해서 이동하는 것을 학습
- Catch : 해당 grid world에는 바람이 불고 있어 Agent가 수행하는 Action에 방해가 되는 상황



Q-learning - FrozenLake-v0

- 환경이 stochastic(or non-deterministic)
- discount rate : 0.98
 - learning rate를 어떤 값으로 주어도 정확도가 0.5를 넘지 못함
- discount rate : 0.99
 - learning rate : 0.85 → 정확도 : 0.6895(가장 높은 정확도)
 - learning rate에 따라 차이 보임

$$\hat{Q}(s, a) \leftarrow (1 - \alpha)\hat{Q}(s, a) + \alpha[r + \gamma \max_{a'} \hat{Q}(s', a')]$$

Q-learning - FrozenLake-v0

```
# 할 수 있는 Action과 총 State 수
N_ACTIONS = 4
N_STATES = 16

# LEARNING_RATE - 기존 Q값에 비중을 두는 정도
LEARNING_RATE = .6 # 0.85: fast learning
# discount factor : 미래에 받을 보상에 대한 신뢰도(0 ~ 1)
DISCOUNT_RATE = .98

# 몇 번 시도를 할 것인가 (에피소드)
N_EPISODES = 2000

def main():
    """Main"""
    frozen_lake_env = gym.make("FrozenLake-v0")

    # 모든 0으로 테이블 초기화
    Q = np.zeros([N_STATES, N_ACTIONS])

    # learning parameters 설정

    # 총 rewards 및 episode별 단계를 포함하는 list 생성
    rewards = [] # reward 저장하는 list
```

- Action, State 정의
 - Action은 상하좌우로 4개,
State는 4*4 Grid로 16개
- Learning Rate, Discount Factor 및 Episode 값 설정
- 환경 : FrozenLake-v0 불러오기
- 초기화 및 Q-Table 초기화
- List 생성

Q-learning - FrozenLake-v0

```
# 해당 episode동안
for i in range(N_EPISODES):
    # 환경을 재설정하고 새 observation을 가져옵니다.
    state = frozone_lake_env.reset()
    episode_reward = 0 # reward 총합
    done = False

    # Q-Table 학습 알고리즘
    while not done:
        # Q-Table에서 원하는 방식으로(argmax) 작업을 선택합니다.
        action = np.argmax(Q[state, :])

        # 환경으로부터 새로운 state 및 reward 받기
        # new_state : 다음 상태
        new_state, reward, done, _ = frozone_lake_env.step(action)

        reward = -1 if done and reward < 1 else reward
```

- 모든 환경 Reset
- Q-learning 적용
 - Action : argmax로 선택
 - 다음 state, reward 등 받기 - Update

Q-learning - FrozenLake-v0

```
# learning rate를 사용하여 Q-Table에 새로운 지식 update
# 수식 사용
Q[state, action] = (
    1 - LEARNING_RATE) * Q[state, action] + LEARNING_RATE * (
        reward + DISCOUNT_RATE * np.max(Q[new_state, :]))

episode_reward += reward
state = new_state

rewards.append(episode_reward)

# Reward / episode 출력
print("Score over time: " + str(sum(rewards) / N_EPISODES))
print("Final Q-Table Values")
```

- Q(s, a) 수식 사용해서
update – learning rate 및
discount factor 사용
- Reward 및 state update,
print

Q-learning - FrozenLake-v0

```
for i in range(10): # 10간격으로 test & 출력
    # 환경을 재설정하고 새 observation을 가져옵니다.
    state = frozone_lake_env.reset()
    episode_reward = 0
    done = False

    # Q-Table 학습 알고리즘
    while not done:
        # Q-Table에서 원하는 방식으로(argmax) 작업을 선택합니다.
        action = np.argmax(Q[state, :])

        # 환경으로부터 새로운 state 및 reward 받기
        new_state, reward, done, _ = frozone_lake_env.step(action)
        frozone_lake_env.render()
        time.sleep(.1)

        episode_reward += reward
        state = new_state
```

- test하고 print
- 앞에 방식과 동일하게 Q-learning 적용

Q-learning - FrozenLake-v0

```
if done:
    # episode_reward : 총 reward합
    print("Episode Reward: {}".format(episode_reward))
    print("GAME OVER")
    print("episode reward: ", episode_reward)
```

```
rewards.append(episode_reward)
print("Success rate: " + str(sum(rewards) / N_EPISODES))
print("Final Q-Table Values")
print(Q)
plt.bar(range(len(rewards)), rewards, color="blue")
plt.show()
```

```
frozone_lake_env.close()
```

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

- 총 결과를 print

Q-learning - FrozenLake-v0

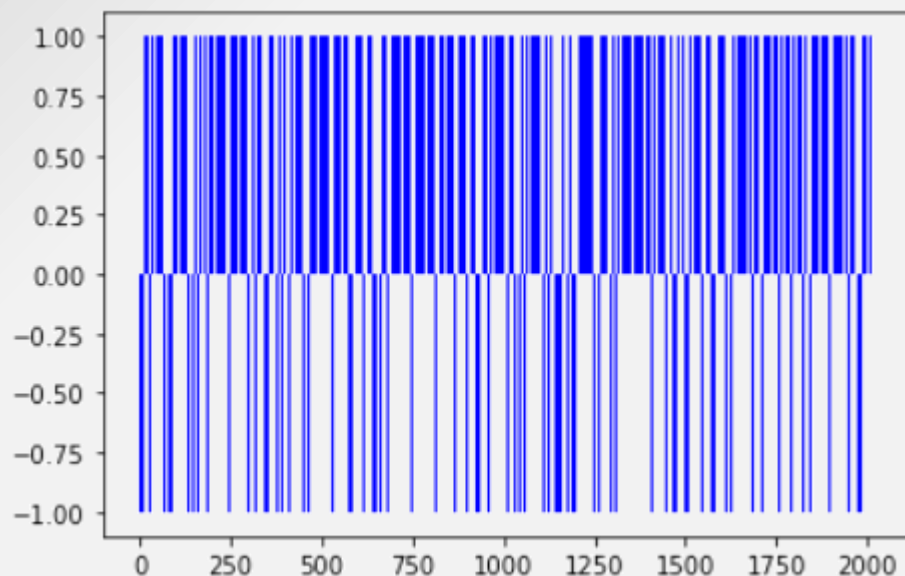
```

Episode Reward: 1.0
GAME OVER
episode reward: 1.0
Success rate: 0.428
Final Q-Table Values
[[ 0.12070919 -0.50499607 -0.4687401  -0.46903535]
 [-0.74750625 -0.60147728 -0.64831474  0.18501279]
 [-0.0234789  -0.43196242 -0.41595143 -0.38758879]
 [-0.75        -0.71544957 -0.75        -0.92912496]
 [ 0.19498059 -0.75        -0.73612735 -0.70262422]
 [ 0.          0.          0.          0.          ]
 [-0.98335547 -0.97995936 -0.89676682 -0.98824479]
 [ 0.          0.          0.          0.          ]
 [-0.66836325 -0.72859517 -0.69365099  0.3298838 ]
 [-0.8273901   0.53125401 -0.68640071 -0.74758248]
 [-0.11222815 -0.91067255 -0.8744815  -0.9059375 ]
 [ 0.          0.          0.          0.          ]
 [ 0.          0.          0.          0.          ]
 [-0.5         -0.5         0.85522722 -0.27135696]
 [-0.06125625  0.94849171  0.          0.          ]
 [ 0.          0.          0.          0.          ]]

```

- Learning rate : 0.5, Discount factor : 0.99, 정확도 : 약 42%
- Q-Table : 전체 Q-value값 반환 → 해당 Table에서 가장 높은 Q-value를 선택하는 것이 optimal policy를 선택하는 방법
- Episode Reward : 전체 보상의 총합
- Success rate : 정확도

Q-learning - FrozenLake-v0



- 그래프 : 총 횟수 당 Reward 값
→ 평균 : 정확도
- 오른쪽 그림 : 시도하는 방향을
그림으로 보여줌(상하좌우)

(Up)
SFFF
FHFH
F**F**HH
HFFG

(Down)
SFFF
FHFH
FFFH
H**F**FG

(Right)
SFFF
FHFH
F**F**HH
HFFG

(Down)
SFFF
FHFH
FF**F**H
HFFG

(Left)
SFFF
FHFH
FFFH
HFF**G**

(Down)
SFFF
FHFH
FFFH
HFF**G**

DQN

FrozenLake-v0

CartPole-v1

1

DQN Frozen_lake – v0

- FrozenLake-v0
- Start 지점에서 시작,
검은 부분이 Hole,
Goal 지점이 도착



1

DQN - FrozenLake-v0

```
env = gym.make('FrozenLake-v0')

# Env를 기준으로 한 입력 및 출력 크기
input_size = env.observation_space.n
output_size = env.action_space.n
learning_rate = 0.1

# feed-forward part of the network 설정
# action 선택
X = tf.placeholder(shape=[1, input_size], dtype=tf.float32) # state input
W = tf.Variable(tf.random.uniform(
    [input_size, output_size], 0, 0.01)) # weight

Qpred = tf.matmul(X, W) # Out Q prediction
Y = tf.placeholder(shape=[1, output_size], dtype=tf.float32) # Y label

loss = tf.reduce_sum(tf.square(Y - Qpred))
train = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate).minimize(loss)

optimizer = tf.keras.optimizers.Adam()
```

- 환경 : FrozenLake-v0 불러오기
- Learning Rate 값 설정
- Action, State 정의 – State Input과 Weight
 - X : input, W : Weight
- Qpred : 예측값(Q-prediction)
- Loss Function : $Y - Qpred$
- Optimizer : Adam 사용

DQN - FrozenLake-v0

```
# Q-learning related parameters
# discount factor
dis = .99
# 몇 번 시도를 할 것인가 (에피소드)
num_episodes = 2000

# 에피소드마다 총 리워드의 합을 저장하는 리스트
rList = []

def one_hot(x):
    return np.identity(16)[x:x + 1]

init = tf.global_variables_initializer()

with tf.compat.v1.Session() as sess:
    sess.run(init)
    for i in range(num_episodes):
        # Reset
        s = env.reset()
        e = 1. / ((i / 50) + 10)
        rAll = 0
        done = False
        local_loss = []
```

- Discount Factor : 0.99
- Episode : 2000
- One_hot : Hypothesis따라 실제 Neural Network를 실행시키는 부분

DQN - FrozenLake-v0

```
# Q-Network training
while not done:
    # greedy (임의의 행동을 할 가능성이 있는) action을 선택
    # from the Q-network
    Qs = sess.run(Qpred, feed_dict={X: one_hot(s)})
    if np.random.rand(1) < e:
        a = env.action_space.sample()
    else:
        a = np.argmax(Qs)

    # environment로부터 새로운 상태 및 보상 받기
    s1, reward, done, _ = env.step(a)
    if done:
        # Update Q, and no Qs+1 (terminal state)
        Qs[0, a] = reward
    else:
        # 새로운 Q_s1 얻기 (new state - network를 통해)
        Qs1 = sess.run(Qpred, feed_dict={X: one_hot(s1)})
        # Update Q
        Qs[0, a] = reward + dis * np.max(Qs1)

    # 목표(Y) 및 예측 Q(Qpred) 값을 사용하여 네트워크 학습, run
    sess.run(train, feed_dict={X: one_hot(s), Y: Qs})

    rAll += reward
    s = s1
    rList.append(rAll)
```

- Qs : Network 예측값 저장
- sess.run : TensorFlow에서 Hypothesis에 따라 실제 Neural Network를 실행
- Action : E-Greedy에 따라 선택
 - Qs에 의해 선택되는 Action : Qpred의 weigh에 따라 결정된 가장 큰 argument 선택
- Step(), Qs[0, a]로 Qs얻고, Update

DQN - FrozenLake-v0

```
# Q-Network training
while not done:
    # greedy (임의의 행동을 할 가능성이 있는) action을 선택
    # from the Q-network
    Qs = sess.run(Qpred, feed_dict={X: one_hot(s)})
    if np.random.rand(1) < e:
        a = env.action_space.sample()
    else:
        a = np.argmax(Qs)

    # environment로부터 새로운 상태 및 보상 받기
    s1, reward, done, _ = env.step(a)
    if done:
        # Update Q, and no Qs+1 (terminal state)
        Qs[0, a] = reward
    else:
        # 새로운 Q_s1 얻기 (new state - network를 통해)
        Qs1 = sess.run(Qpred, feed_dict={X: one_hot(s1)})
        # Update Q
        Qs[0, a] = reward + dis * np.max(Qs1)

    # 목표(Y) 및 예측 Q(Qpred) 값을 사용하여 네트워크 학습, run
    sess.run(train, feed_dict={X: one_hot(s), Y: Qs})

    rAll += reward
    s = s1
rList.append(rAll)
```

- X(이전 State s)에 대한 Y(label)을 Qs가 가지고 있음 → 이를 이용하여 Network를 train하고, Weight Update
- 현재 State인 s를 새로운 State인 s1으로 update

1

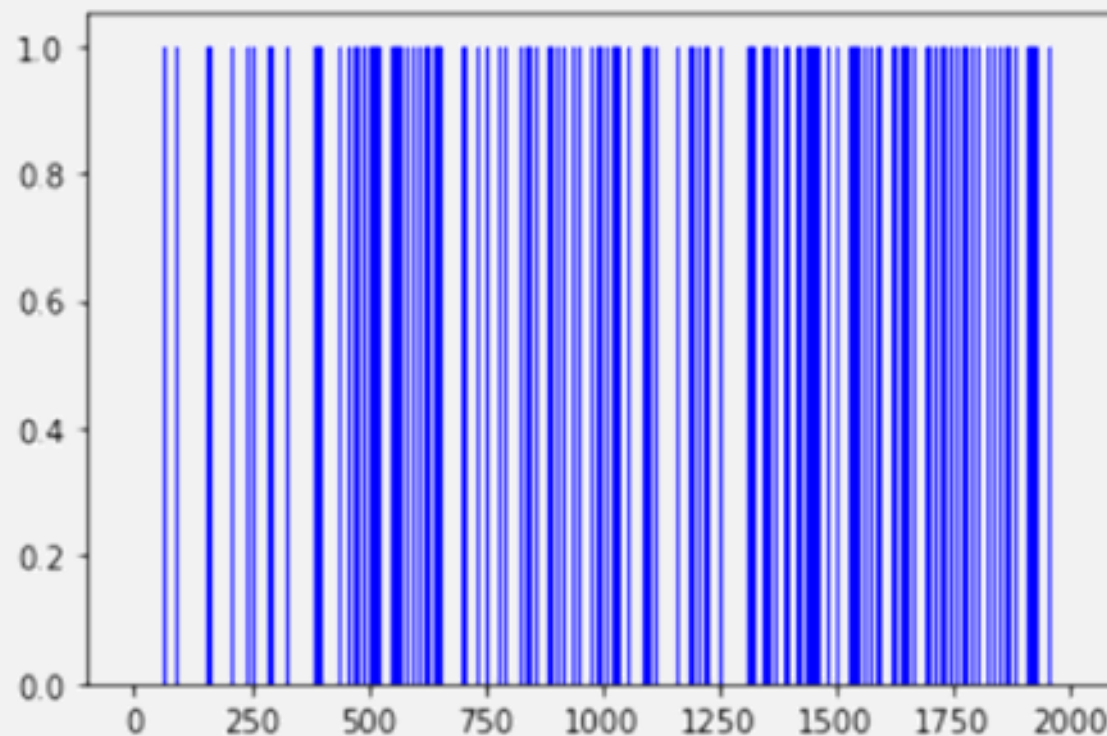
DQN - FrozenLake-v0

```
print("Percent of successful episodes: " +  
      str(sum(rList) / num_episodes) + "%")  
plt.bar(range(len(rList)), rList, color="blue")  
plt.show()
```

- DQN의 정확도가 낮은 이유

1. Optimal policy로 수렴하지 않기 때문
2. Network가 얇기 때문
3. 입력 Dataset 간의 correlation이 큼

Percent of successful episodes: 0.492%



1

Q-learning - FrozenLake-v0

Episode Reward: 1.0

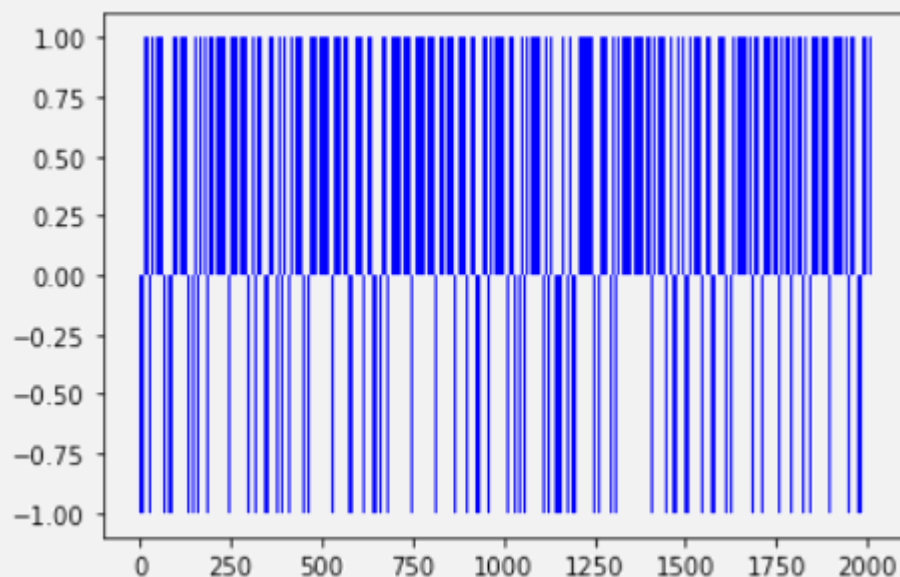
GAME OVER

episode reward: 1.0

Success rate: 0.428

Final Q-Table Values

```
[[ 0.12070919 -0.50499607 -0.4687401 -0.46903535]
 [-0.74750625 -0.60147728 -0.64831474  0.18501279]
 [-0.0234789  -0.43196242 -0.41595143 -0.38758879]
 [-0.75        -0.71544957 -0.75        -0.92912496]
 [ 0.19498059 -0.75        -0.73612735 -0.70262422]
 [ 0.          0.          0.          0.          ]
 [-0.98335547 -0.97995936 -0.89676682 -0.98824479]
 [ 0.          0.          0.          0.          ]
 [-0.66836325 -0.72859517 -0.69365099  0.3298838 ]
 [-0.8273901   0.53125401 -0.68640071 -0.74758248]
 [-0.11222815 -0.91067255 -0.8744815  -0.9059375 ]
 [ 0.          0.          0.          0.          ]
 [ 0.          0.          0.          0.          ]
 [-0.5         -0.5         0.85522722 -0.27135696]
 [-0.06125625  0.94849171  0.          0.          ]
 [ 0.          0.          0.          0.          ]]
```



(Up)

SFFF

FHFH

F[■]FH

HFFG

(Down)

SFFF

FHFH

FFFH

H[■]FG

(Right)

SFFF

FHFH

F[■]FH

HFFG

(Down)

SFFF

FHFH

FFF[■]H

HFFG

(Left)

SFFF

FHFH

FFFH

HFF[■]G

(Down)

SFFF

FHFH

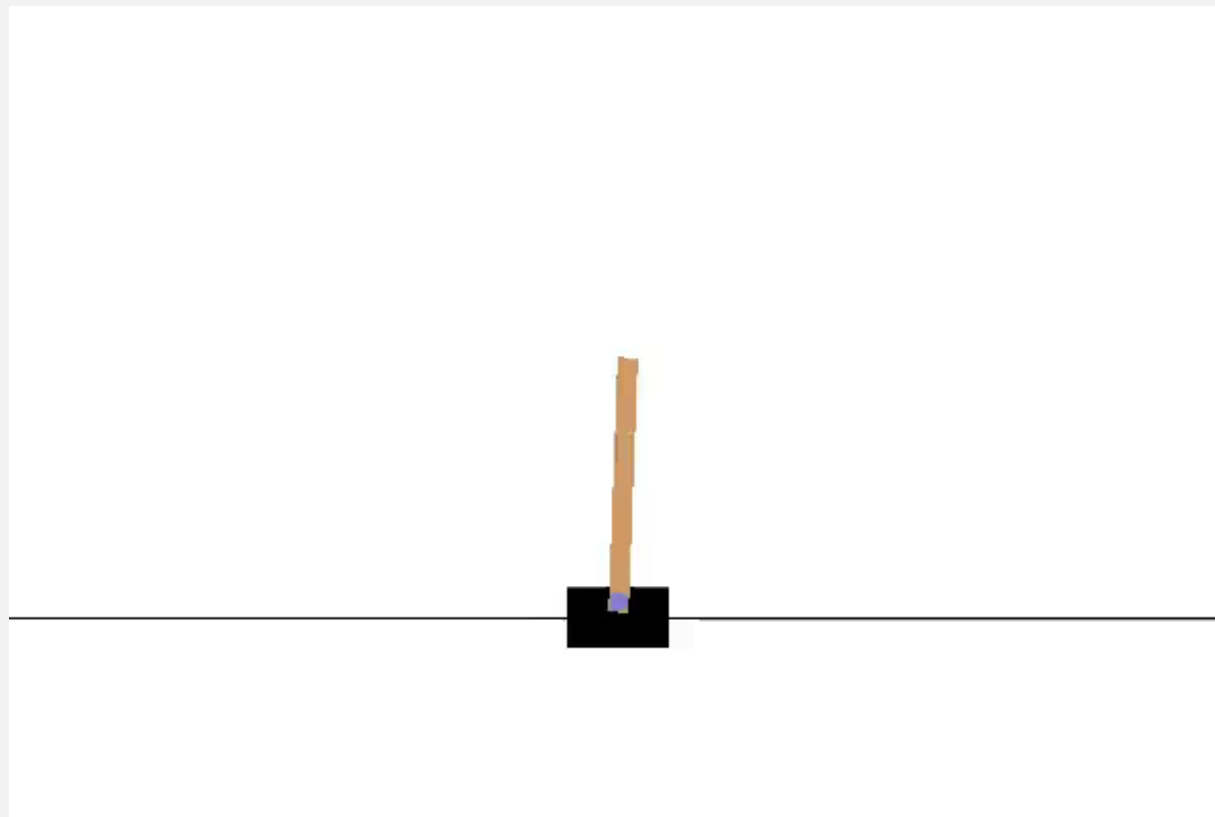
FFFH

HFF[■]G

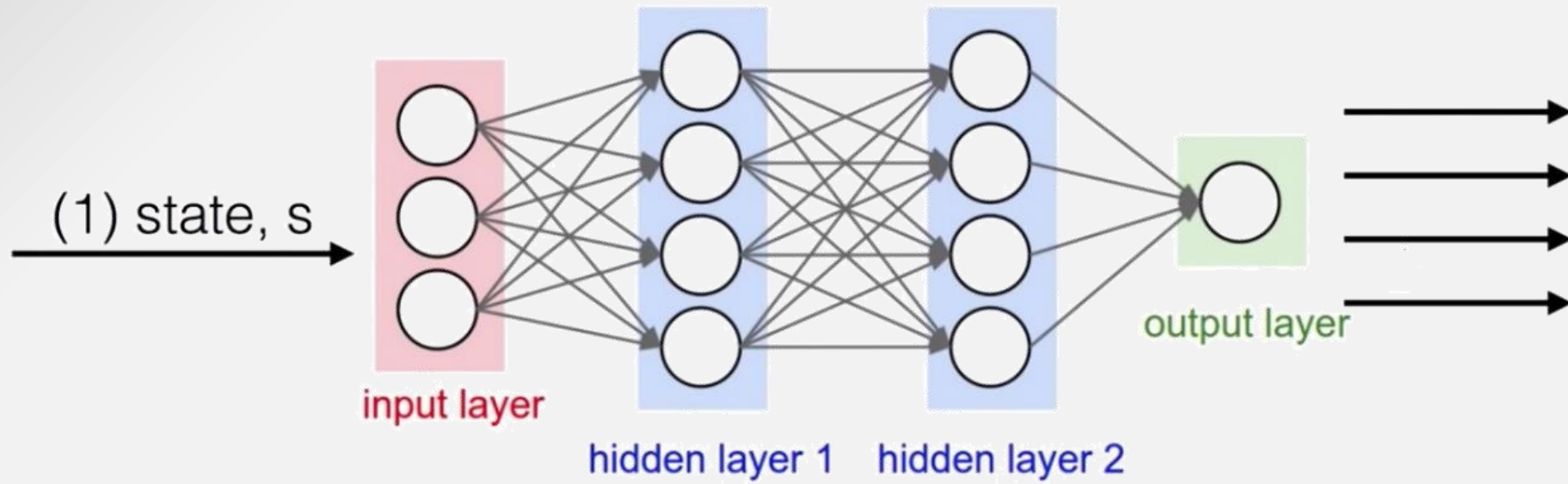
1

DQN - CartPole-v1

- CartPole-v1
- 카트 위에 막대기가 고정되어 있고 막대기는 중력에 의해 바닥을 향해 자연적으로 기울게 되는 환경을 제공
- 목적 : 카트를 좌, 우로 움직이며 막대기가 기울지 않고서 있을 수 있도록 유지



DQN - CartPole-v1



- Layer : 2개 (unit 수 : 24)
- Activation Function : ReLu

DQN - CartPole-v1

State가 입력, Q 함수가 출력인 Neural Network 생성

```
class DQN(tf.keras.Model):  
    def __init__(self, action_size):  
        super(DQN, self).__init__()  
        self.fc1 = Dense(24, activation='relu')  
        self.fc2 = Dense(24, activation='relu')  
        self.fc_out = Dense(action_size, kernel_initializer=RandomUniform(-1e-3, 1e-3))  
  
    def call(self, x):  
        x = self.fc1(x)  
        x = self.fc2(x)  
        q = self.fc_out(x)  
        return q
```

- State가 입력, Q 함수가 출력인 Neural Network 생성

DQN - CartPole-v1

```
# CartPole 환경에서 agent 역할을 하는 DQNAgent class
# CartPole 환경에서는 4가지의 State와 2가지의 Action으로 이루어짐
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        # DQN 알고리즘을 구동하기 위한 hyper-parameter 값을 설정
        self.discount_factor = 0.99
        self.learning_rate = 0.001
        self.epsilon = 1.0
        self.epsilon_decay = 0.999
        self.epsilon_min = 0.01
        self.batch_size = 64
        self.train_start = 1000
        # 리플레이 memory는 최대 크기 2000 으로 설정
        self.memory = deque(maxlen=2000)
        # model 과 target_model 두 개의 Neural Network 생성
        self.model = DQN(action_size)
        self.target_model = DQN(action_size)
        self.optimizer = Adam(lr=self.learning_rate)

        self.update_target_model()
```

- CartPole 환경에서 agent 역할을 하는 DQNAgent class
- CartPole 환경에서는 4가지의 State와 2가지의 Action(좌, 우)으로 이루어짐
- State
 - 카트의 위치
 - 카트의 속도
 - 막대기의 각도
 - 막대기의 끝부분(상단) 속도

DQN - CartPole-v1

```
# CartPole 환경에서 agent 역할을 하는 DQNAgent class
# CartPole 환경에서는 4가지의 State와 2가지의 Action으로 이루어짐
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        # DQN 알고리즘을 구동하기 위한 hyper-parameter 값을 설정
        self.discount_factor = 0.99
        self.learning_rate = 0.001
        self.epsilon = 1.0
        self.epsilon_decay = 0.999
        self.epsilon_min = 0.01
        self.batch_size = 64
        self.train_start = 1000
        # 리플레이 memory는 최대 크기 2000 으로 설정
        self.memory = deque(maxlen=2000)
        # model 과 target_model 두 개의 Neural Network 생성
        self.model = DQN(action_size)
        self.target_model = DQN(action_size)
        self.optimizer = Adam(lr=self.learning_rate)

        self.update_target_model()
```

- Discount factor : 0.99,
Learning Rate : 0.001
- Epsilon : 1.0
- model과 target_model 두 개의
Neural Network 생성
 - Q 함수 학습할 때 model의
parameter와 같이 갱신되는
것을 막기 위해 target_model
로 분리

DQN - CartPole-v1

```
# target_model의 weight를 model의 weight로 update 하는 함수
def update_target_model(self):
    self.target_model.set_weights(self.model.get_weights())

# 상태 S, 액션 A, 보상 R, 다음 상태 S', 완료 여부 done 을 저장
def remember(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))

# epsilon을 이용하여 Exploration과 Exploitation의 비율을 조정
# E-greedy 사용
def choose_action(self, state):
    return random.randrange(self.action_size) if (np.random.rand() <= self.epsilon) else np.argmax(self.model.predict(state))
```

- target_model의 weight를 model의 weight로 update 하는 함수
- 상태 S, 액션 A, 보상 R, 다음 상태 S', 완료 여부 done을 저장
- epsilon을 이용하여 Exploration과 Exploitation의 비율을 조정
 - E-greedy 사용

DQN - CartPole-v1

```
# 경사 하강법으로 learning을 진행
def train_model(self):
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

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

    states = np.array([sample[0][0] for sample in mini_batch])
    actions = np.array([sample[1] for sample in mini_batch])
    rewards = np.array([sample[2] for sample in mini_batch])
    next_states = np.array([sample[3][0] for sample in mini_batch])
    dones = np.array([sample[4] for sample in mini_batch])

    model_params = self.model.trainable_variables
    with tf.GradientTape() as tape:
        predicts = self.model(states)
        one_hot_action = tf.one_hot(actions, self.action_size)
        predicts = tf.reduce_sum(one_hot_action * predicts, axis=1)

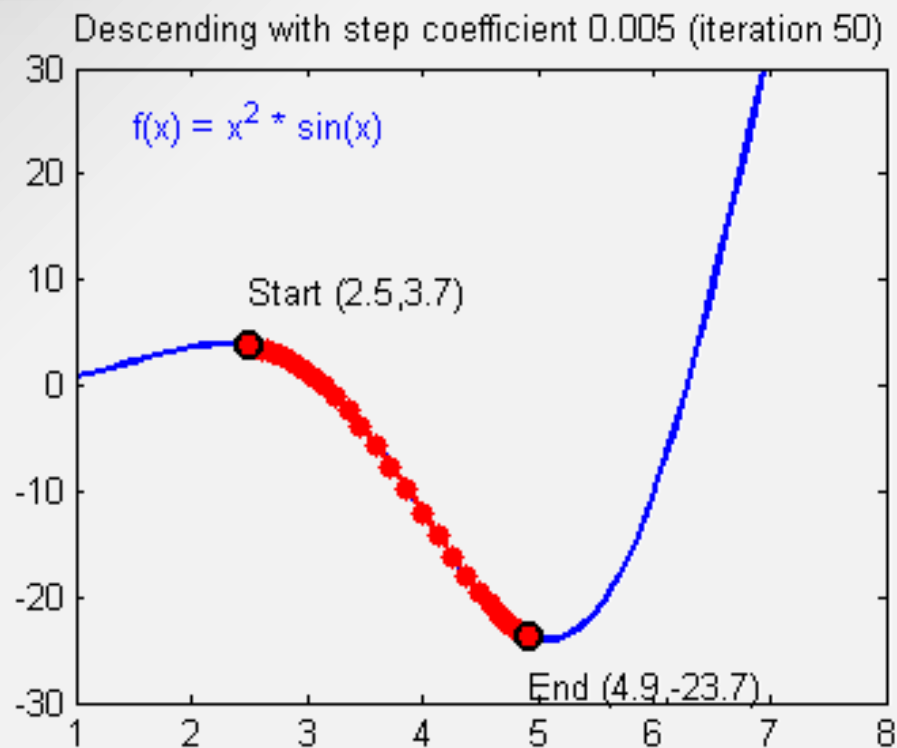
        target_predicts = self.target_model(next_states)
        target_predicts = tf.stop_gradient(target_predicts)

        max_q = np.amax(target_predicts, axis=-1)
        targets = rewards + (1 - dones) * self.discount_factor * max_q
        loss = tf.reduce_mean(tf.square(targets - predicts))

    grads = tape.gradient(loss, model_params)
    self.optimizer.apply_gradients(zip(grads, model_params))
```

- 경사 하강법으로 learning을 진행
 - Sample 간 correlation을 줄이기 위해 memory에 저장된 Data를 random하게 섞어 training에 사용할 mini-batch data 생성
 - 벨만 최적 방정식 이용 : 계산된 정답(target) - 예측값(predicts)의 차이 줄여나감

DQN - CartPole-v1



- 경사 하강법(Gradient Descent)
- Gradient Descent 방법은 1차 미분계수를 이용해 함수의 최소값을 찾아가는 iterative한 방법
- 함수 값이 낮아지는 방향으로 독립 변수 값을 변형시켜가면서 최종적으로는 최소 함수 값을 갖도록 하는 독립 변수 값을 찾는 방법

DQN - CartPole-v1

```
# CartPole-v1과 학습을 진행할 DQNAgent를 생성
if __name__ == "__main__":
    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    agent = DQNAgent(state_size, action_size)

    scores, episodes = [], []
    score_avg = 0
    num_episode = 300
    # episode 가 시작될 때마다 환경을 초기화
    for e in range(num_episode):
        done = False
        score = 0

        state = env.reset()
        state = state.reshape(1, -1)
```

- CartPole-v1과 학습을 진행할 DQNAgent를 생성
- episode 가 시작될 때마다 환경을 초기화

DQN - CartPole-v1

```
while not done:
    env.render()

    # 현재 State로 Action을 선택
    action = agent.choose_action(state)

    # 선택한 Action으로 환경에서 한 timestep 진행
    next_state, reward, done, info = env.step(action)
    next_state = next_state.reshape(1, -1)

    # timestep마다 보상 0.1, episode가 중간에 끝나면 -1 Reward
    score += reward
    reward = 0.1 if not done or score == 500 else -1

    # 리플레이 memory에 <s, a, r, s'> 저장
    agent.remember(state, action, reward, next_state, done)

    # 매 timestep마다 학습
    if len(agent.memory) >= agent.train_start:
        agent.train_model()

    state = next_state
```

- 현재 State에서 Action을 하나 선택 → 한 step 진행
- 해당 Result로 받은 Reward를 현재 State와 선택한 Action과 함께 저장
- Memory가 일정 크기 이상으로 저장시 매 step마다 학습

DQN - CartPole-v1

```
if done:
    # 각 episode마다 타겟 model을 model의 weight로 update
    agent.update_target_model()

    # episode마다 학습 결과 출력
    score_avg = 0.9 * score_avg + 0.1 * score if score_avg != 0 else score
    print('episode: {:3d} | score avg {:.2f} | memory length: {:4d} | epsilon: {:.4f}'.format(e, score_avg, len(agent.memory),
                                                                                               agent.epsilon))

    # episode마다 학습 결과 그래프로 저장
    scores.append(score_avg)
    episodes.append(e)
    plt.plot(episodes, scores, 'b')
    plt.xlabel('episode')
    plt.ylabel('average score')
    plt.savefig('cartpole_graph.png')

    # 이동 평균이 400 이상일 때 종료
    if score_avg > 400:
        agent.model.save_weights('./save_model/model', save_format='tf')
        sys.exit()
```

- 한 episode 가 완료될 때마다 target_model 을 model의 weight와 일치하도록 동기화하고, score와 model의 weight를 저장

1

DQN - CartPole-v1

```
episode: 0 | score avg 67.00 | memory length: 67 | epsilon: 1.0000
episode: 1 | score avg 61.70 | memory length: 81 | epsilon: 1.0000
episode: 2 | score avg 59.33 | memory length: 119 | epsilon: 1.0000
episode: 3 | score avg 54.50 | memory length: 130 | epsilon: 1.0000
episode: 4 | score avg 52.25 | memory length: 162 | epsilon: 1.0000
episode: 5 | score avg 48.02 | memory length: 172 | epsilon: 1.0000
episode: 6 | score avg 45.72 | memory length: 197 | epsilon: 1.0000
episode: 7 | score avg 43.15 | memory length: 217 | epsilon: 1.0000
episode: 8 | score avg 40.83 | memory length: 237 | epsilon: 1.0000
episode: 9 | score avg 37.95 | memory length: 249 | epsilon: 1.0000
episode: 10 | score avg 38.36 | memory length: 291 | epsilon: 1.0000
```

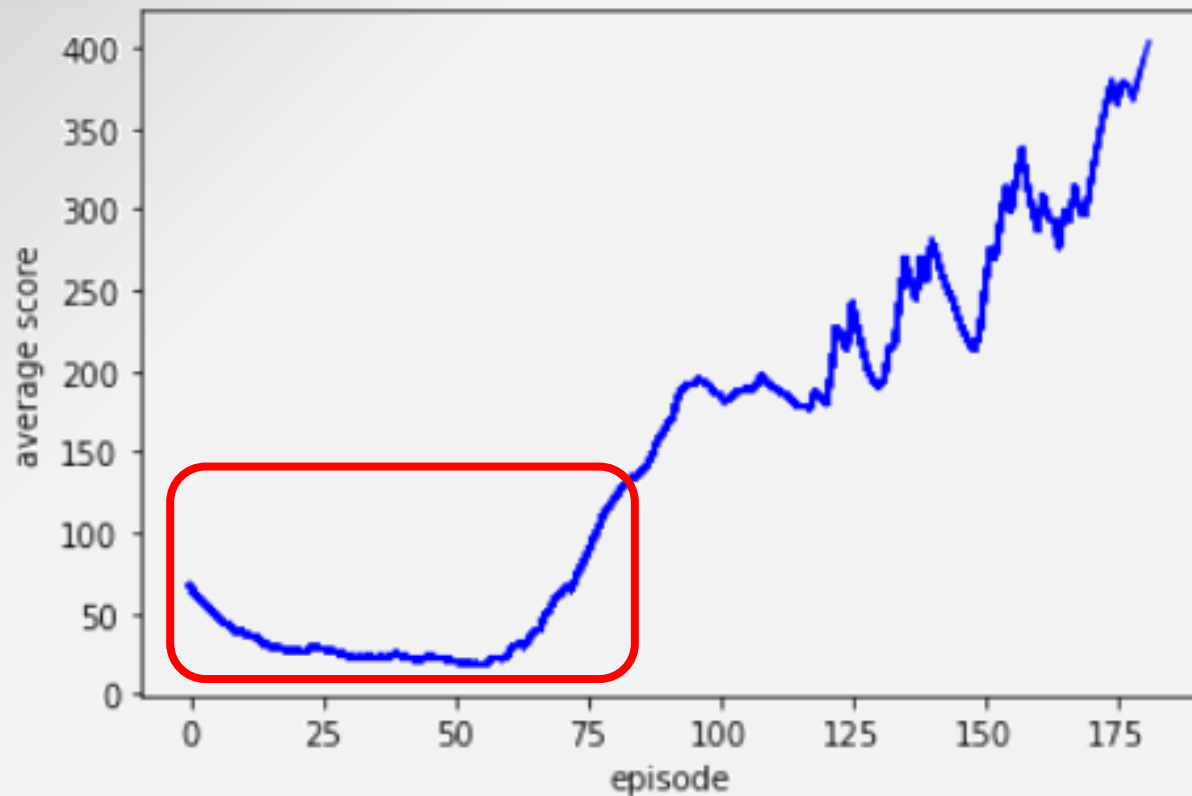
⋮

```
episode: 171 | score avg 335.10 | memory length: 2000 | epsilon: 0.0100
episode: 172 | score avg 351.59 | memory length: 2000 | epsilon: 0.0100
episode: 173 | score avg 366.43 | memory length: 2000 | epsilon: 0.0100
episode: 174 | score avg 379.79 | memory length: 2000 | epsilon: 0.0100
episode: 175 | score avg 365.11 | memory length: 2000 | epsilon: 0.0100
episode: 176 | score avg 378.60 | memory length: 2000 | epsilon: 0.0100
episode: 177 | score avg 377.34 | memory length: 2000 | epsilon: 0.0100
episode: 178 | score avg 368.11 | memory length: 2000 | epsilon: 0.0100
episode: 179 | score avg 381.30 | memory length: 2000 | epsilon: 0.0100
episode: 180 | score avg 393.17 | memory length: 2000 | epsilon: 0.0100
episode: 181 | score avg 403.85 | memory length: 2000 | epsilon: 0.0100
```

An exception has occurred, use %tb to see the full traceback.

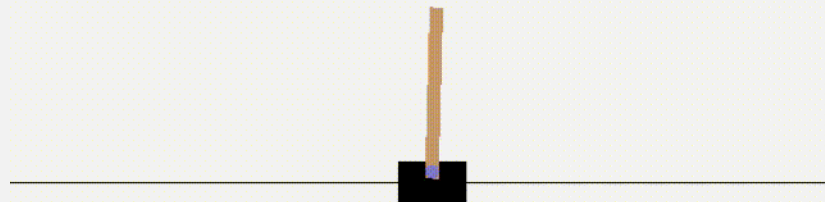
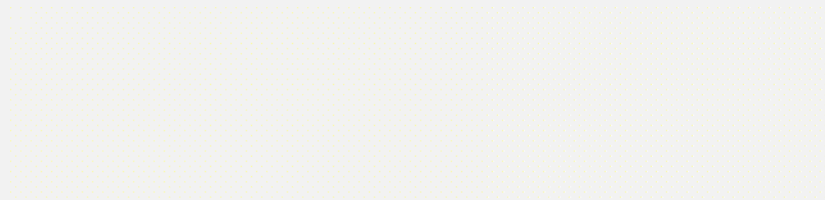
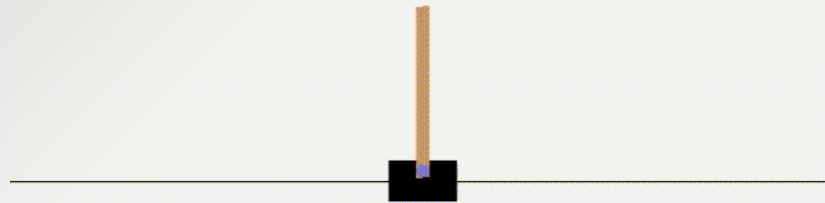
- Episode마다 현재 State를 출력
- Episode가 진행될 수록 평균 점수가 높아지는 학습 효과

DQN - CartPole-v1



- 학습이 진행되면서 저장된 Score의 변화에 대한 그래프
- 점점 Score가 높아짐

DQN - CartPole-v1



- 실행 결과
- 초반에는 불안정함
- Episode 100회 반복을 넘어가면서 점차 안정적으로 막대기를 세움

- <https://dohk.tistory.com/239>
- <https://gist.github.com/tfedohk>
- <https://hunkim.github.io/ml/>
- <https://gym.openai.com/envs/FrozenLake-v0/>
- <https://gym.openai.com/envs/CartPole-v1/>
- <https://github.com/hunkim/DeepRL-Agents>
- <https://github.com/hunkim/ReinforcementZeroToAll>
- <https://jonghyunho.github.io/reinforcement/learning/cartpole-reinforcement-learning.html>