Reinforcement Learning Term Project: Cart Pole Problem

Kichang Lee*
Department of Integrated Technology
College of Computing
Yonsei University
kichang.lee@yonsei.ac.kr

1 Introduction

이번 Term project에서는 OpenAI에서 제공하는 gym library를 활용하여 Cart Pole 문제를 해결하는 것을 다룬다. Cart Pole 문제를 해결하기 위해 사용하는 알고리즘은 크게 'Q-learning', 'Deep Q-Network', 'Policy gradient method' 총 3가지이다. 해당 보고서는 Cart Pole 문제를 해결하기 위해 사용하는 앞서 언급한 세 가지 알고리즘에 대한 이론적인 설명과 구현을 다루는 Section 3, 4, 5와 그 결과를 분석하는 Section 6으로 구성되어 있다.

Table 1에는 이번 과제에 사용한 하드웨어 환경과 언어, 라이브러리 등에 대한 정보를 명시해두었다. **모든 실험의 경우 재현가능성을 보장하기 위하여 random seed를 본인의 학번인 2022314416으로 통일 및 고정하여 진행하였다.**

	Specification
OS	Microsoft Windows 10 Education
OS Version	10.0.19042
CPU	Intel(R) Core(TM) i9-10850K CPU 3.60GHz, 3600Mhz
RAM	64GB
GPU	NVIDIA RTX 3090
Language	Python(=3.11.1)
Library	numpy(=1.23.5), gym(=0.26.2), torch(=2.0.0+cu118)

Table 1: Environment Specification

2 Cart Pole Problem

(a) Balanced state (b) Unbalanced state

Figure 1: Cart Pole system

Cart Pole 문제는 assignment 3에서 다룬 Mountain car 문제와 같이 OpenAI에서 제공하는 classical control 문제의 일종으로, 마찰이 없는 표면에 있는 막대(pole)의 한 쪽 끝이 연결되어 있는 cart가 있으며, 해당 막대는 올바르게 새워진 상태로 시작된다. 해당 문제의 목표는 막대가 넘어지지 않도록 cart를 좌우로 움직이는 것이며, 특정 각도 이상으로 막대가 넘어지면 종료된다.

Figure 1는 gym 라이브러리를 통해서 Cart Pole 문제를 실행 및 시각화 한 것으로 좌측과 같이 pole이 balanced state를 유지하도록 검은색 cart를 좌우로 움직이게 된다. Cart Pole의

*Embedded Intelligent System Laboratory(EIS LAB)

state는 4가지의 값으로 구성되며, Kumar [2]의 reference에서는 이를 $\{x, \dot{x}, \theta, \dot{\theta}\}$로 표기하였으며, x는 cart position, $\dot{x}$는 cart의 velocity, θ는 pole의 각도, $\dot{\theta}$는 pole의 angular velocity를 의미한다. 이러한 state의 observation state는 다음과 같이 정의 된다.

$$
\begin{aligned}
x &\in [-4.8, 4.8] \\
\dot{x} &\in (-\inf, \inf) \\
\theta &\in [-0.418\ rad\ (-24^\circ), 0.418\ rad\ (24^\circ)] \\
\dot{\theta} &\in (-\inf, \inf)
\end{aligned}
$$

이러한 환경에서 pole의 각도가 $[-12^\circ, 12^\circ]$ 넘어가는 경우, cart의 position이 $[-2.4, 2.4]$를 벗어나는 경우, 한 episode 내에서 500 step 이상 진행되는 경우 (truncation) episode가 종료되게 된다. 해당 환경에서 취할 수 있는 action은 좌측으로 이동, 우측으로 이동 두 가지가 존재하며, episode가 종료되지 않은 경우 한 step당 +1의 reward를 받게 된다.

이번 프로젝트에서 제공된 reference [2]의 경우 Cart Pole의 version이 달라 200 step이 truncation의 기준점으로 보이며, 수렴에 대한 기준 또한 100 episode를 연달아 195 이상의 reward를 얻는 경우 수렴이 된 것으로 판단하였다. 하지만 현재 gym library에서 제공하는 최신 version의 경우 500 step을 기준으로 하고 있으며, 475 이상의 reward를 받은 경우를 기준으로 잡고 있다. [2]의 기준의 경우 후자를 만족시키는 경우 자연스럽게 만족시키게 되는 조건으로 조금 더 어렵고 최신 version을 기준으로 100 episode를 연달아 475 이상의 reward를 받은 경우 수렴된 것으로 판별하였다. 마지막으로 수렴이 안되는 경우를 고려하여 1000 episode를 초과하는 경우 학습을 종료하였다.

3 Q-learning

3.1 Background

Q-learning이란 Model-Free Control 알고리즘의 일종으로 다음과 같은 Q table을 update하여 최적의 policy π를 찾는 과정을 수행한다.

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + |(s_t, a_t)]$$

s_t와 a_t는 각각 t 시점의 state와 action으로 각각의 state와 action에 대하여 discount factor γ가 적용된 return의 기댓값을 최대화 하는 과정이라고 설명할 수 있다. Q-learning의 알고리즘의 경우 (s_t, a_t)에서 가장 큰 return을 얻을 수 있는 a'를 찾으며 이는 다음과 같이 수식화 할 수 있다.

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

일반적으로 local minimum 문제를 해결하기 위하여 ϵ-greedy 방식을 통해 searching을 수행한다. [2]에서는 ϵ 값을 1에서 점진적으로 줄여나가는 방식을 채택한다. 최종적으로 Q-table $Q(s, a)$을 update하는 과정은 다음과 같은 수식을 따르게 된다 [5].

$$Q(s, a) = Q(s, a) + \alpha[R(s, a) + \gamma \max_a Q(s', a) - Q(s, a)]$$

이때 α는 learning rate에 대응되는 개념으로 [2]에서는 1.0에서 0.1로 값을 줄여나가는 방식을 활용하였다. 위의 update 과정은 agent가 수렴할 때 까지 반복하게 된다. 위 수식에서 $R(s, a)$는 (s, a)에서의 reward를 의미하며, 이전 assignemnt에서 사용한 *SARSA* 알고리즘과 달리 다음 state (s')에 대한 search를 포함한다는 점에서 차이가 있다.

3.2 Implementation

보고서의 가독성을 위해 핵심적인 코드를 제외한 나머지 코드와 코드를 실행하기 위한 command line등은 `https://github.com/leekichang/IIT6051_RL/tree/main/Term_Project/src`에서 확인할 수 있도록 하였다. Q-learning은 뒤에서 사용할 방식들과 달리 Q table을 기반으로 동작하는 알고리즘이므로 state와 action에 대한 table을 구성해야 한다. 이때 Cart Pole의 환경은 연속적인 환경이기에 이를 discrete하게 변환할 필요가 있다. 해당 과정의 경우 [2]의 `bucketize()` 함수를 활용하였다.

Code 1는 `bucketize()` 함수를 구현한 것으로, 입력된 state의 space에 대해서 bucket index를 두고 그 간격을 일정하게 나눠 index를 배분하는 방식으로 작동한다. 이때 $\dot{x}$와 $\dot{\theta}$의 경우

Listing 1 Bucketize function

```
def bucketize(self, state):
    bucket_indice = []
    for i in range(len(state)):
        if state[i] <= self.STATE_BOUNDS[i][0]:
            bucket_index = 0
        elif state[i] >= self.STATE_BOUNDS[i][1]:
            bucket_index = self.NUM_BUCKETS[i] - 1
        else:
            bound_width = self.STATE_BOUNDS[i][1] - self.STATE_BOUNDS[i][0]
            offset = (self.NUM_BUCKETS[i]-1)*\\
                      self.STATE_BOUNDS[i][0]/bound_width
            scaling = (self.NUM_BUCKETS[i]-1)/bound_width
            bucket_index = int(round(scaling*state[i] - offset))
        bucket_indice.append(bucket_index)
    return tuple(bucket_indice)
```

주어진 범위가 $[-\inf, \inf]$ 이므로 이를 적당한 구간으로 제한할 필요가 있다. 이는 [2]와 같이들을 각각 $\dot{x} \in [-0.5, 0.5]$와 $\dot{\theta} \in [-50°, 50°]$로 제한하였다. 이를 통해 discrete한 Q-table을 얻을 수 있다. Bucket의 size와 같은 파라미터를 포함한 언급하지 않는 파라미터들의 경우 기본적으로 [2]의 값을 참조하였다.

Listing 2 Q-learning algorithm

```
def fit(self):
    learning_rate = self.get_learning_rate(0)
    explore_rate  = self.get_explore_rate(0)
    discount_factor = 0.99
    num_streaks = 0
    for episode in tqdm(range(self.NUM_EPISODES)):
        obv     = self.env.reset()
        state_0 = self.state_to_bucket(obv[0])
        for t in range(self.MAX_T):
            action = self.select_action(state_0, explore_rate)
            obv, reward, done, _, _ = self.env.step(action)
            state = self.state_to_bucket(obv)
            best_q = np.amax(self.q_table[state])
            self.q_table[state_0 + (action,)] += learning_rate*\\
            (reward + discount_factor*(best_q) \\
            - self.q_table[state_0 + (action,)])
            state_0 = state
            if done:
                if (t >= self.SOLVED_T):
                    num_streaks += 1
                else:
                    num_streaks = 0
                break
        if num_streaks > self.STREAK_TO_END:
            break
        explore_rate  = self.get_explore_rate(episode)
        learning_rate = self.get_learning_rate(episode)
```

Code 2은 가장 Q learning에 있어서 가장 핵심적인 학습 부분이다. 앞서 언급한 것과 같이 동적으로 episode가 변화함에 따라 learning rate와 ϵ 값을 변화시킬 수 있도록 설정하였으며, greedy하게 Q 값 $\max_a Q(s', a)$를 찾아 update하는 방식으로 구현하였음을 확인 할 수 있다.

4 Deep Q-Network

4.1 Background

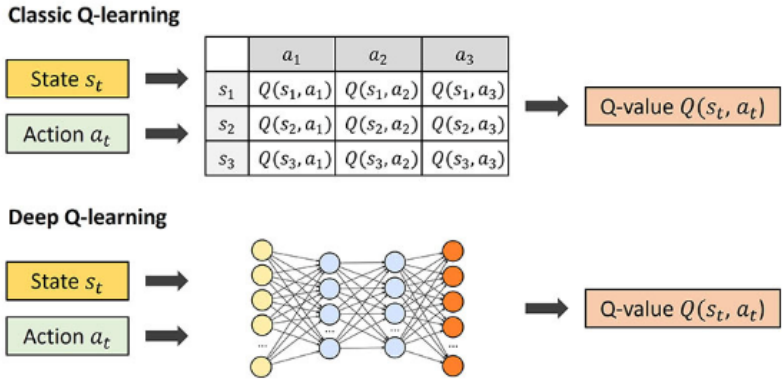


Figure 2: Cart Pole system

Deep Q-Network (DQN)는 Q-table을 사용하는 classic Q-learning과 달리 state와 action 값을 입력으로 받아 새로운 Q value를 출력하는 신경망(neural network)를 학습하는 방식으로 강화학습에 딥러닝 방식을 적용하는 방식이라고 이해할 수 있다. Figure 2 [1]은 그 차이를 나타낸 그림으로 DQN의 경우 이산적인 Q table을 사용하지 않기 때문에 연속적인 state와 action에 대한 값을 입력으로 처리할 수 있다는 장점이 있다. 뿐만 아니라, state space ($\mathcal{S}$), action space ($\mathcal{A}$)가 큰 환경에서 Q table을 사용하는 경우 많은 memory와 exploration time을 요구되는데, DQN의 경우 neural network을 통해 이러한 문제도 완화할 수 있다.

해당 프로젝트에서는 Minh et al. [3]이 제안한 DQN 학습 방식을 기반으로 agent를 학습시켰다. 핵심적인 요소는 (1) agent network와 target network의 구분, (2) experience replay 기법이다. Minh et al.에서는 Atari game을 학습시키기 위해 게임 화면을 convolutional neural network를 통과시키는 과정을 거쳤지만, Cart Pole의 경우 주어진 observation space가 상대적으로 간단하므로 해당 부분은 제외하였다.

Minh et al. 이전의 연구들에서는 강화학습의 특성상 명확한 ground truth가 없으므로 특정 policy에서 얻은 return 등을 기준으로 학습을 진행해왔는데, 이러한 접근은 agent가 학습되어 가며 계속하여 target이 바뀌게 되므로 학습의 불안정성을 키웠다. Agent network와 target network의 분리를 통해 agent의 network이 고정된 target network를 target으로 학습하여 적절하게 학습을 할 수 있다.

Experience replay 기법의 경우 sequential한 experience를 가지는 강화학습의 특성으로 인해 하나의 episode를 순차적으로 학습하게 되면 이에 대한 편향으로 인해 학습의 variance가 커져 학습이 어려워진다는 문제를 해결하기 위해 도입된 방식이다. 뿐만 아니라 한 번 사용했던 experience를 memory에 저장해두었다가 다시 사용할 수 있으므로 효율적인 학습 또한 가능해진다.

4.2 Implementation

우선 구현에 있어서 딥러닝 프레임워크인 pytorch를 사용하여 network를 학습하였으며, network의 구조는 [2], [3]에서 제안한 것과 같이 세개의 linear layer를 통해 (4-24-24-2)개의 node를 가지도록 하였으며, 활성화함수는 relu를 사용하였다. Optimizer의 경우 Adam을 사용하였으며 기본적으로 learning rate는 0.001로 설정하여 진행하였다.

Code 3은 DQN agent를 학습시키는 과정에 사용 과정을 간략하게 정리한 pseudocode이다 (실제 사용한 코드된 코드와 차이가 있으며, 해당 코드는 앞서 명시한 url에 첨부되어 있다). 앞선 Section에서 언급한 바와 같이 agent의 main network와 ground truth 역할을 하는 target network를 구분하여 업데이트를 진행하는 것을 확인할 수 있다.

Code 4의 경우 agent가 exploration을 통해 모델을 학습하는 전체 과정을 보여준다. `remember()` 함수를 통해 데이터를 저장하고 `train_model()`에서 이를 batch 단위로 random sampling하여 사용한다.

Listing 3 DQN train

```
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  = [sample[0][0] for sample in mini_batch])
    actions = [sample[1] for sample in mini_batch]
    rewards = [sample[2] for sample in mini_batch]
    next_states = [sample[3][0] for sample in mini_batch]
    dones = [sample[4] for sample in mini_batch]
    self.model.train(); self.target_model.eval()
    predicts = self.model(states)
    one_hot_action = F.one_hot(actions, num_classes=self.action_size)
    predicts = torch.sum(one_hot_action*predicts, -1)
    with torch.no_grad():
        target_predict = self.target_model(next_states)
    max_q = torch.amax(target_predict, dim=-1)
    targets = (rewards + (1 - dones) * self.discount_factor * max_q)
    loss = self.criterion(predicts, targets)
    loss.backward()
    self.optimizer.step()
    self.optimizer.zero_grad()
```

Listing 4 DQN Exploration

```
def explore(self):
    scores, episodes = [], []
    num_episode = 1000
    converge = False
    num_streak = 0
    e = 0
    while not converge and e < num_episode:
        done = False
        state = self.env.reset()
        state = state[0].reshape(1, -1)
        score = 0
        while not done:
            action = self.agent.choose_action(state)
            next_state, reward, done, trunc, info = self.env.step(action)
            next_state = next_state.reshape(1, -1)
            score += reward
            reward = reward if not done else -100
            self.agent.remember(state, action, reward, next_state, done)
            if len(self.agent.memory) >= self.agent.train_start:
                self.agent.train_model()
            state = next_state
            if done or trunc:
                self.agent.update_target_model()
```

5 Policy Gradient Method

5.1 Background

Q-value를 기반으로 하는 강화학습 알고리즘과 달리 policy gradient 기반의 알고리즘은 policy π 대상으로 학습하여 각 action에 대한 확률을 출력하는 형태로 학습을 진행한다. 어떤 state와 action에 대해 $\pi(s_t, a_t) = \mathbb{P}[A_t = a|S_t = t]$를 얻으며 이는 각 action에 대한 확률을 얻는 형태이며 특정 state에서 특정 'action의 value'를 계산하는 것이 아닌, 특정 state에서 '어떤

action을 해야 하는지'를 학습하므로 입력의 형태를 (s_t, a_t)가 아닌 (s_t)로 하며 출력되는 결과 또한 $Q(s_t, a_t)$가 아닌 $\pi_\theta(a|s_t)$로 차이가 있다.

Policy gradient method의 경우 학습을 위해 objective function $J(\theta)$를 정의하며 일반적으로 $J(\theta)$는 주어진 state (s)에서의 return의 기댓값으로 정의되며 이는 다음과 같은 수식으로 쓸 수 있다.

$$\begin{aligned} J(\theta) &= v_{\pi_\theta}(s) \\ &= \mathbb{E}_{\pi_\theta}[R_{t+1} + \gamma v_{\pi_\theta}(S_{t+1})|S_t = s] \end{aligned}$$

따라서 policy gradient method의 기본적인 방향성은 $J(\theta)$를 maximize하는 θ를 찾는 것이다. 이러한 objective function을 통해 neural network를 학습시키기 위해서 일반적으로 gradient를 사용하므로 $J(\theta)$는 미분이 가능해야한다. Sutton et al.의 연구 [4]에서 $j(\theta)$는 다음과 같이 미분이 가능하다고 이야기하였다.

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_s d_{\pi_\theta}(s) \sum_a \frac{\pi_\theta(a|s)}{\pi_\theta(a|s)} \nabla_\theta \pi_\theta(a|s) q_\pi(s,a) \\ &= \sum_s d_{\pi_\theta}(s) \sum_a \pi_\theta(a|s) \nabla_\theta log \pi_\theta(a|s) q_\pi(s,a) \\ &\quad (\because \nabla_\theta log \pi_\theta(a|s) = \frac{\pi_\theta(a|s)}{\pi_\theta(a|s)}) \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(a|s) q_\pi(s,a)] \end{aligned}$$

이때 d_{π_θ}는 state s에 agent가 있을 확률을 나타내는 분포이다. 이때 기댓값은 많은 수의 샘플을 통해 근사할 수 있으므로 괄호 안의 $\nabla_\theta log \pi_\theta(a|s) q_\pi(s,a)$를 구하는 문제가 된다. 그러나 앞선 방식들과 달리 policy gradient method에서는 Q-value $(q_\pi(s,a))$에 대한 지식이 없으므로 이를 알고 있는 값으로 바꿔줄 필요가 있다. 이때 Q-value $(q_\pi(s,a))$를 return G_t로 대체하는 방식이 대표적인 policy gradient method인 REINFORCE 알고리즘이다. [4]

5.2 Implementation

Listing 5 REINFORCE algorithm training

```
def train(self):
    self.model.train()
    log_prob_actions = []
    rewards = []
    done, trunc = False, False
    episode_reward = 0
    state = self.env.reset()[0]
    while not done and not trunc:
        state = torch.FloatTensor(state).unsqueeze(0)
        action_pred = self.model(state)
        action_prob = F.softmax(action_pred, dim=-1)
        dist = distributions.Categorical(action_prob)
        action = dist.sample()
        log_prob_action = dist.log_prob(action)
        state, reward, done, trunc, _ = self.env.step(action.item())
        log_prob_actions.append(log_prob_action)
        rewards.append(reward)
        episode_reward += reward
    log_prob_actions = torch.cat(log_prob_actions)
    returns = self.calc_returns(rewards, self.gamma)
    loss = self.update_policy(returns, log_prob_actions)
    return loss, episode_reward
```

Code 5은 REINFORCE 알고리즘의 학습을 구현한 것이다. network의 출력값을 확률로 해석할 수 있도록 softmax 함수를 통과시켜 합이 1인 0과 1사이의 vector로 만든 이후, distribution

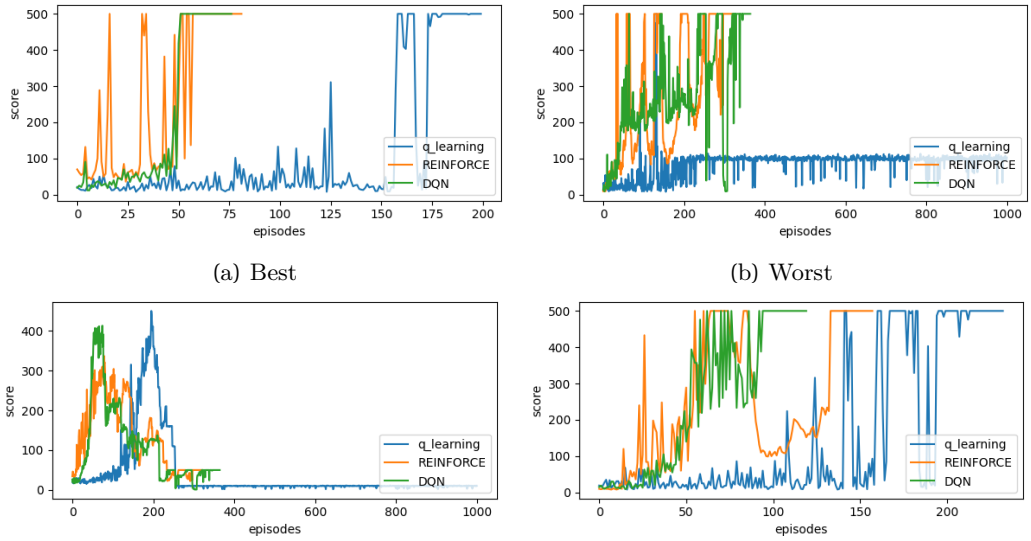
Listing 6 REINFORCE update

```
def update_policy(self, returns, log_prob_actions):
    returns = returns.detach()
    loss = - (returns * log_prob_actions).sum()
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
    return loss.item()
```

으로 전환하여 해당 distribution에서 action을 sampling한 이후 return을 계산하여 gradient 기반 update를 수행하여 policy를 update한다.

Code 6는 loss 값과 로그 값을 입력으로 받아 이를 gradient ascending 방식으로 update하는 과정을 밟는다.

6 Experiment



(a) Best

(b) Worst

(c) Average

(d) Median

Figure 3: Results in 10 runs

우선 사용한 3개의 알고리즘의 성능에 대한 평가를 수행하였다. 다른 environment seed에서 총 10회 학습을 진행하였으며 Figure 3 (a)는 그 10개중 가장 빠르게 수렴 조건을 만족시킨 경우, (b)는 가장 수렴이 오래 걸린 경우, (c)는 10번의 시행에 대한 평균 score, (d)는 수렴 조건을 만족시키는데 소요된 episode 수의 중앙값 경우를 나타낸다.

Figure 3 (a)의 경우 가장 빠르게 학습이 된 경우이며 가장 빠르게 최대 score인 500d을 달성한 방식은 policy gradient method인 REINFORCE 알고리즘이었다. 하지만, REINFORCE 알고리즘의 경우 학습이 DQN에 비교하여 불안정한 모습을 보였다. Q-learning의 경우 다른 학습 방식들과 비교하여 수렴까지 훨씬 많은 episode가 요구됨을 확인할 수 있었다.

Figure 3 (b)의 경우 수렴이 가장 오래걸린 시행의 결과로, Q-learning의 경우 최종적으로 수렴 기준인 475 이상에 들어가지 못하며 1000 episode 이내에 수렴하지 못했다. REINFORCE 알고리즘과 DQN의 경우 REINFORCE 알고리즘의 학습 경향은 (a)의 경우와 유사하게 불안정했으며, DQN 또한 중간에 학습이 불안정해지는 모습을 관찰할 수 있었다.

Figure 3 (c)의 경우 모든 시행의 score의 평균을 나타내는데, 시행마다 소요된 episode의 수가 다르다는 문제를 해결하기 위해 특정 episode를 경험하지 못한 시행의 경우 해당 episode의 값을 0으로 두었다. 해당 계산 방식을 고려하면 학습이 빠르게 되어 높은 score를 구한 구간이 peak의 형태로 나타나게 된다. 해당 peak의 위치가 좌측에 있으면 있을 수록 학습이 빠르게 되는 경향이 있다는 것이며, 전체적인 곡선의 폭이 좁을수록 그 분포의 분산이 적다는 의미로

해석할 수 있다. 해당 해석을 적용할 경우 DQN과 REINFORCE 알고리즘 모두 유사하게 100 episode 근방에서 peak를 보이는 것을 확인할 수 있으며, 그 peak의 폭은 DQN이 더 좁고 높은 것을 확인할 수 있다. Q-learning의 경우 약 250 episode 이상의 상황에서는 대부분의 시행의 결과가 매우 낮은 결과를 보여주는 것을 확인할 수 있다. 이는 학습 과정에서 local minimum등의 이유로 제대로 된 학습이 이루어지지 못하고 수렴에 실패하는 경우가 있다는 것을 보여준다.

마지막으로 Figure 3 (d)의 경우 중앙값 시행으로 DQN이 가장 빠르게 수렴되었고, REINFORCE 알고리즘의 경우 중간에 score가 급격히 낮아졌다 다시 학습되는 것을 확인하였다. 이러한 결과들을 종합적으로 볼 때, Cart Pole 문제의 경우 DQN 방식이 가장 효과적이고 안정적으로 학습을 수행할 수 있으며, Q-learning의 경우 수렴의 속도나 수렴의 안정성의 측면에서 가장 나쁜 성능을 가지고 있다는 것을 확인할 수 있다. REINFORCE 알고리즘의 경우 Q-learning에 비해서는 상대적으로 안정적인 성능을 보이지만, DQN에 비하여 수렴의 안정성 측면에서 떨어지는 것으로 보인다.

	Best	Worst	Median	Average	Std
Q learning	200	1000 (Did Not Converge)	224.5	303.1	233.12
DQN	**77**	365	**120.5**	**150.2**	87.44
REINFORCE	82	**322**	153.5	173.1	**68.90**

Table 2: Results in 10 runs, all results are lower the better (**Best**, Second)

Table 6는 Figure 3의 실험의 통계값들을 나타낸 것이다. 볼드체로 처리된 것은 가장 좋은 성능, 밑줄은 두번째를 의미한다. 앞선 해석과 동일하게 가장 빠르게 수렴한 경우에 DQN, REINFORCE, Q-learning의 순이었으며 소요된 episode는 각각 77, 82, 200이었다. 가장 느리게 수렴된 경우 365, 322, 1000이었으며, Q learning의 경우 수렴에 실패한 결과이다. 중앙값의 경우 120.5, 153.5, 224.5였으며[2] 평균 소요 episode는 150.2, 173.1, 303.1 순으로 DQN이 가장 빠른 수렴속도를 보이며 policy gradient 방식인 REINFORCE 알고리즘에 비해 약 20 episode 가량 빠르며 약 300 epsiode가 요구되는 policy gradient 방식보다는 두배가 빠른 속도를 보인다는 것을 확인할 수 있다. 마지막으로 표준편차의 경우 여러번 학습에 걸쳐 안정성을 간접적으로 확인할 수 있는 지표로 해당 수치가 낮다는 것은 여러 시행에 걸쳐 비슷한 성능을 보인다는 것이며 현재와 같이 어느정도 수렴이 가능한 상황에서는 낮은 표준편차가 유사한 학습의 궤적의 지표로 해석할 수 있다고 사료된다. 해당 해석을 기준으로는 REINFORCE 알고리즘이 여러 시행에 걸쳐 유사한 학습 궤적을 보인다고 해석된다.

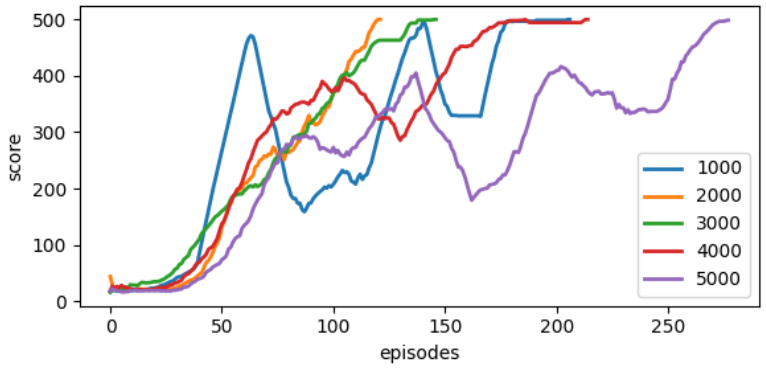


Figure 4: Results with different memory sizes

앞선 실험의 결과에서 가장 안정적인 성능을 보여준 DQN 방식에 대해서 추가적인 parmeter를 조절하는 실험을 추가로 진행하였다. 첫번째로 조절한 값은 experience replay를 위해 사용되는 memory의 크기를 조절하였다. Figure 4는 memory의 크기를 1000, 2000, 3000, 4000, 5000으로 조정하며 학습을 수행하며 마지막 25번의 episode의 score의 평균을 나타낸 것이다. 실험 결과 기본적으로 사용한 설정값인 2000이 가장 빠르게 수렴에 성공하는 모습을 확인할 수 있었으며, 3000, 4000, 1000, 5000이 그 뒤를 이었다. 이 실험 결과를 통해 memory의 크기와

[2] 짝수번의 시행으로 정확한 중앙값이 존재하지 않아 두 값의 평균을 명시하였다.

성능에는 linear한 상관관계는 존재하지 않으며 좋은 성능을 얻을 수 있는 sweet-spot이 있다는 것을 확인할 수 있다.

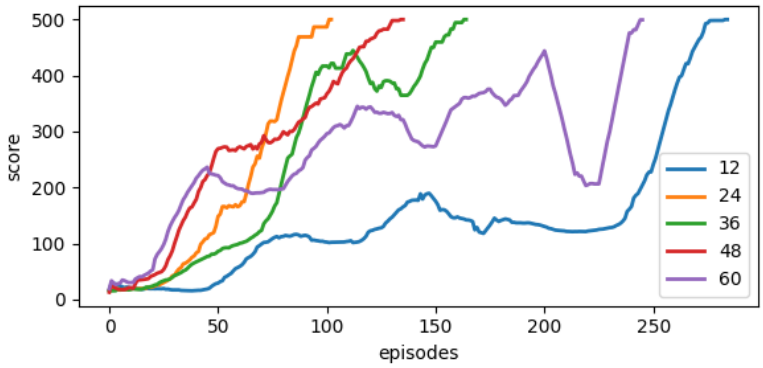


Figure 5: Results with different hidden layer size

다음은 neural network의 hidden layer size를 조정하였다. Network의 hidden layer size가 너무 큰 경우에는 가중치에 비하여 주어지는 입력이 적어 over-fitting 문제나 학습이 느려지는 문제가 발생할 수 있고, 너무 작은 경우에는 주어진 환경을 가중치가 충분히 표현하지 못하며 under-fitting가 발생할 수 있다. Figure 5은 앞서 언급한 network의 구조에서 hidden layer의 size를 12, 24, 36, 58, 60으로 바꾸며 얻은 학습 궤적을 마지막 25개의 episode에 대해 평균을 취하여 얻은 결과이다. 이 경우 hidden layer의 크기를 24로 한 경우 가장 좋은 빠른 수렴을 보였으며 48, 36, 60, 12가 뒤를 이었다. 해당 결과 또한 앞선 실험 결과와 동일하게 network의 구조 또한 수렴 속도와 linear한 상관관계를 가지지는 않으며, 적절한 값 즉 sweet spot이 존재한다는 것을 확인할 수 있었다.

	Q learning	DQN	REINFORCE
Average	5.86 msec	407.30 msec	433.21 msec
Std	0.0014	0.0624	0.0785

Table 3: Inference time

마지막으로, 물론 학습의 성능 또한 중요하지만, 실제 자율주행과 같은 real world application의 경우 inference에 소요되는 시간이 중요할 수 있다. Table 3은 각 알고리즘별 inference에 소요되는 시간을 나타낸 것으로 총 100episode를 수행하며 한 step당 소요된 시간을 측정한 것이다. 이 경우 Q learning 방식이 평균 5.86 msec/step으로 가장 빠른 속도를 보였으며 그 다음은 DQN이 407.30 msec/step, REINFORCE 알고리즘이 433.21 msec/step으로 가장 느렸다. Q learning의 경우 neural network를 사용하는 방식이 아니므로 복잡한 연산을 요구하지 않기 때문에 가장 빠른 속도를 보이는 것으로 보이며, REINFORCE 알고리즘의 경우 확률값 등을 계산하는 추가적인 연산으로 인해 차이가 발생한 것으로 생각된다. 물론 Cart Pole가 매우 간단한 문제이며 가장 느린 433.21 msec/step 또한 real-time system에 적용될 수 있을만큼 빠른 속도이지만, 결과적으로 동일한 성능을 얻을 수 있는 세 알고리즘을 다루는 상황에서 DQN과 비교하여 69.5배, REINFORECE와 비교하여 73.92배 빠른 속도를 얻을 수 있으며 추가적으로 back-propagation 또한 없으므로 실질적인 학습에 소요되는 시간이 적을 것이라는 점을 고려하면, Q learning 또한 Cart Pole와 같은 classic control 문제에서는 사용할 법한 알고리즘이라고 이야기할 수 있다.

References

[1] [RL] 강화학습 알고리즘: (1) DQN (Deep Q-Network) — ai-com.tistory.com. https://ai-com.tistory.com/entry/RL-%EA%B0%95%ED%99%94%ED%95%99%EC%8A%B5-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-1-DQN-Deep-Q-Network. [Accessed 01-Jun-2023].

[2] Swagat Kumar. Balancing a cartpole system with reinforcement learning–a tutorial. *arXiv preprint arXiv:2006.04938*, 2020.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[4] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[5] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.