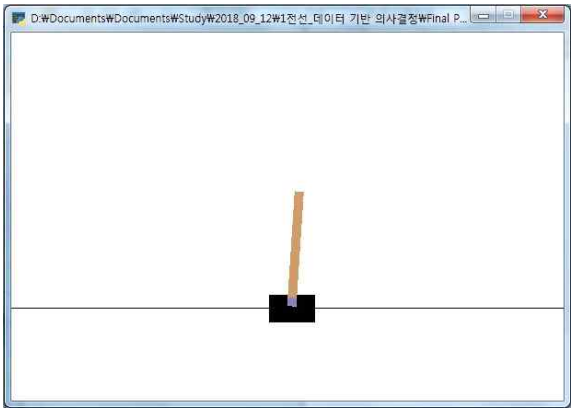
Cart Pole

- Gym Library -

20120450 박수호

1. Problem Define



마찰이 없는 바닥을 움직이는 검은색 차(cart)에 막대가 붙어있다. 강화학습을 통해 차를 좌우로 적절히 움직여 막대가 넘어지지 않도록 하는 것을 목표로 한다.

이 문제를 해결할 경우 상태에 따라 적절한 Action을 취해야 하는 다른 형태의 문제에도 같은 알고리즘을 적용할 수 있을 것으로 예상된다.

2. Mathematical Expression

A. State

1) 화면 전체 pixel을 state로 인식할 수 있음

2) 화면 전체 pixel을 그대로 state로 인식할 경우 state space가 커질 수 있어 시간에 따른 화면의 변화(=pixel2 – pixel1)를 state로 인식할 수 있음

3) 화면 전체 pixel을 이용하는 것은 Cart Pole 문제 뿐 아니라 화면을 볼 수 있는 모든 종류의 문제에 적용이 가능하나 효율이 떨어질 수 있다.

따라서 Cart Pole 문제에 특화된 State를 다음과 같이 정의하고 활용한다.

State = [차(cart)의 위치 / 차의 속도 / 막대와 차의 각도 / 막대 끝(위쪽)의 속도]

B. Action

좌우로 차를 움직이는 행동을 할 수 있다.

Action = [Left / Right]

C. Reward

1) 막대가 ㄱ) 15도 이상 기울어지나 ㄴ) 화면 중앙에서 2.4 units 떨어질 경우 **Episode 종료 및 reward = -1**.

2) 그 외의 경우 **reward = +1.**

3. Required Data - 시뮬레이션 상황이 아니라 가정

A. 차에 위치 센서를 부착하여 실시간 위치 수집

B. 차의 실시간 위치를 이용하여 속도 계산

C. 막대와 차 사이 각도를 실시간으로 수집

D. 막대와 차 사이 실시간 각도를 이용하여 막대 끝의 속도 계산

4. Used Decision Making Algorithm

DQN(Deep Q Network)과 DDQN(Double DQN)을 활용하여 카트와 막대의 상태에 따라 적절한 행동을 취할 수 있는 Agent를 만든다.

A. Code Basic Structure

Replay Memory
position : Memory List 내 최신 데이터 업로드 위치 capacity : Memory 용량 / 초과할 경우 가장 과거 데이터를 지운 후 업데이트 memory : 데이터가 저장된 Memory
push (State, Action, Reward, NextState) : Memory에 데이터 삽입 sample (Sample Size) : Sample Size만큼 Memory에서 데이터를 불어옴

Q Net
n_layers : 모든 layer의 크기를 저장한 List layers : Neural Network의 Parameters를 저장한 attribute
forward (State) : Action(Left, Right)에 대한 Estimated Q value를 계산 save (function_name) : 주어진 name으로 현재 Q network 저장

Agent
env : Agent가 활동하는 Environment 저장 qf : Agent가 활용하는 Neural Network(=Q Net) 저장 RM : Replay Memory n_episode : 총 episode 수 total_step : 총 step 수 u_check : Batch Update에 따라 진행한 Neural Network Update 횟수 train_reward : 시간에 따라 reward 저장 ▶ Training 종료 후 그래프를 그릴 때 Y축 train_step : 시간에 따라 step 수 저장 ▶ Training 종료 후 그래프를 그릴 때 X축 gamma : Discount Factor optimizer : NN Update에 이용하는 optimizer lr_scheduler : optimizer의 learning rate를 원하는 대로 조절해주는 attribute
make_network () : NN 생성 dimS () : State Space dimA () : Action Space train (function name) : Training 진행 (= Data 생성) action_choice (state) : Epsilon Greedy Action Choice Batch_Update (batch size) : Replay Memory를 이용하여 Update 진행 TD_update_Q (state, action, reward, next_state) : Q net Parameter Update train_result () : Training Result 출력 save (function_name) : Train 후 Q net 저장

B. 적용 가능한 알고리즘 : SARSA / Q-Learning etc...

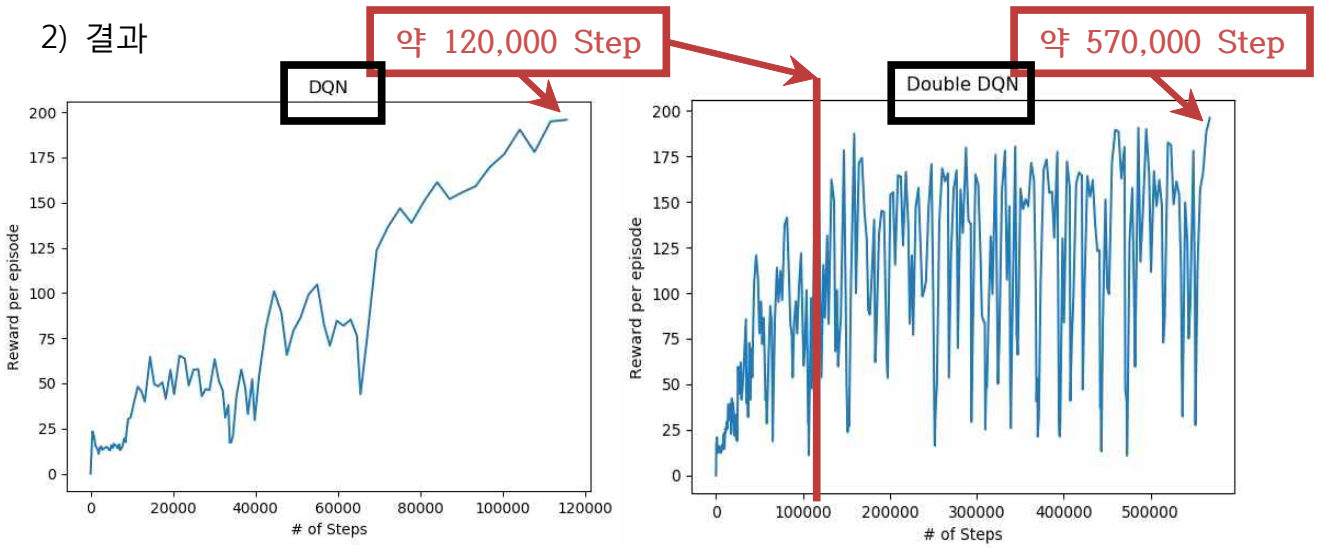
1) SARSA : On-Policy Control
   Exploration, Updating Target 모두 동일한 policy(ex) eps-greedy)를 이용
2) Q-Learning : Off-Policy Control
   Updating Target 생성 시 Greedy Action을 취하여 생성

▶ 대게 Q-Learning이 SARSA 보다 빠르게 수렴한다는 점에 착안하여 Neural Network을 기반으로 **DQN**과 **DDQN**을 적용하여 성능을 분석함.

C. DQN vs DDQN (Double DQN)

1) 평가 기준
   - 막대가 차에서 쓰러지지 않는다면 받을 수 있는 **최대 Reward(=시간) = 200**
   - 열 번의 Episode 평균 195초를 버틸 수 있기까지 필요한 Total Step 수 측정
   - **Total Step 수가 작을수록 빠르게 수렴한 것**으로 판단

2) 결과



3) 결론
   ㄱ. DQN의 단점 : Maximization Bias로 인해 Q(s,a) 값을 Overestimate할 가능성이 존재함
   ㄴ. DDQN : 위의 DQN의 단점을 보완하고자 두 개의 Q Network을 사용. DQN에서는 잘못된 Q Net을 바탕으로 Greedy Action이 선택될 수 있었던 상황에서 다른 Q Net의 Greedy Action을 사용함으로써 해결하려 함
   ㄷ. 대체로 DDQN이 DQN보다 좋은 성능을 보이는 것으로 알려져 있지만 CartPole 문제에서는 DQN이 훨씬 좋은 성능을 보임
   ㄹ. DDQN은 DQN과 달리 Q(s,a) 값을 Underestimate할 가능성이 존재함[1)]

1) 결론 ㄹ. 관련 논문 : Deep Reinforcement Learning with Double Q-learning (Hado van Hasselt and Arthur Guez and David Silver)

ㅁ. CartPole 문제에서는 DDQN에서의 Underestimating이 DQN에서의 Overestimating보다 나쁜 영향을 미치는 것으로 보임

5. Future Plan

A. 막대가 기울어진 방향으로 카트를 빠르게 움직여 막대를 반대 방향으로 기울도록 하고 그 방향으로 다시 이동하는 것이 좋을 전략일 것이라 추측할 수 있음 ▶ 이와 같은 **사전 지식을 Reinforcement Algorithm 상에 반영할 수 있는 방법을 조사**하여 적용 Ex) **Reward Shaping** : 사전에 Q(s,a) Value를 설정해두고, Target 형성에 이용

B. CartPole 문제의 경우 항상 화면의 중앙부에서 Episode 시작 ▶ 화면 양 끝단으로 갈수록 수집되는 데이터의 양이 적음 ▶ 초기에 Local Minimum에 빠질 경우 나오기 어려움 ▶ 화면의 양 끝단의 데이터에 대해서는 Learning Rate를 일정하게 유지하거나 매우 천천히 감소시키는 방법을 적용해볼 수 있음

6. Code

A. 알고리즘 상 핵심 함수 Code만 첨부

B. DQN 코드를 기본으로 DDQN은 DQN과 다른 부분만 첨부

```
def train(self, fname):
    reward_episode = 0
    while True:
        if self.train_reward[-1] > 195:
            print(self.train_reward)
            print(self.train_step)
            break
        self.n_episode += 1
        step = 0
        state = self.env.reset()
        done = False
        while not done:
            self.total_step += 1
            self.env.render()
            action = self.action_choice(state)
            action = np.array(action)
            next_s, reward, done, info = self.env.step(action)
            reward_episode += reward
            if done == True:
                reward = -1
            self.RM.push(state,action,reward,next_s)
            state = next_s
            if self.total_step % 100 == 0 and self.total_step >= 1000
                self.Batch_Update(100)
        if self.n_episode % 20 == 0:
            self.train_reward.append(reward_episode/20)
            self.train_step.append(self.total_step)
            reward_episode = 0
    self.save(fname)
    self.train_result()
```

1 Episode

1 Step

DQN Action Choice

```
def action_choice(self, state):
    state = torch.tensor(state,dtype=torch.float32)
    with torch.no_grad():
        q = self.qf(state)
    q = q.cpu()
    ### eps greedy action ###
    p = random.random()
    if self.u_check == 0:
        action = self.env.action_space.sample()
        return action
    else:
        eps = 1 / math.sqrt(self.u_check/2000)
        eps = min(0.5, eps)
        if p > eps:
            action = q.argmax()
        else:
            action = self.env.action_space.sample()
        return action
```

Eps Greedy

DDQN Action Choice

```
def DDQN_action_choice(self, state):
    state = torch.tensor(state,dtype=torch.float32)
    with torch.no_grad():
        q1 = self.qf1(state)
        q2 = self.qf2(state)
        q = (q1+q2)/2
    q = q.cpu()

    ### eps greedy action ###
    if self.u_check == 0:
        action = self.env.action_space.sample()
        return action
    else:
        p = random.random()
        eps = 1 / math.sqrt(self.u_check/2000)
        eps = min(0.5, eps)
        if p > eps:
            action = q.argmax()
        else:
            action = self.env.action_space.sample()
        return action
```

Based Q Calculation

공통 = Batch Update

```
def Batch_Update(self,batch_size):
    sample = self.RM.sample(batch_size)
    for i in range(len(sample)):
        s,a,r,ns = sample[i]
        self.TD_update_Q(s,a,r,ns)
```

DQN Q-Net Update

```
def TD_update_Q(self, s,a,r,ns):
    self.lr_scheduler.step()
    s = torch.tensor(s,dtype=torch.float32)
    ns = torch.tensor(ns,dtype=torch.float32)

    with torch.no_grad():
        t = r + self.gamma * self.qf(ns).max()

    q = self.qf(s)[a]
    #Loss = F.mse_loss(q, t)
    loss = F.smooth_l1_loss(q, t)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

DDQN Q-Net Update

```
def TD_update_DQ(self, s,a,r,ns):
    s = torch.tensor(s,dtype=torch.float32)
    ns = torch.tensor(ns,dtype=torch.float32)

    random_q = random.random()

    with torch.no_grad():
        if random_q < 0.5:
            t = r + self.gamma * self.qf2(ns).max()
        else:
            t = r + self.gamma * self.qf1(ns).max()

    if random_q < 0.5:
        self.lr_scheduler1.step()
        q = self.qf1(s)[a]
        #Loss = F.mse_loss(q, t)
        loss = F.smooth_l1_loss(q, t)
        self.optimizer1.zero_grad()
        loss.backward()
        self.optimizer1.step()
    else:
        self.lr_scheduler2.step()
        q = self.qf2(s)[a]
        #Loss = F.mse_loss(q, t)
        loss = F.smooth_l1_loss(q, t)
        self.optimizer2.zero_grad()
        loss.backward()
        self.optimizer2.step()
```