환경에 따른 알고리즘 비교 및 향상

상명대학교 휴먼지능정보공학과 Team : 하노이
201810822 한승훈
201910798 노희재
202010799 이재준

개요

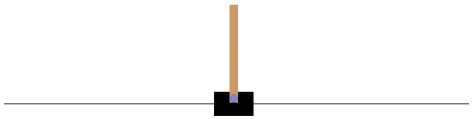
- 난이도에 따라 초급 환경, 중급 환경, 고급 환경을 선택하고 알고리즘 비교 분석 진행
  - 초급 환경: CartPole-v1에서 DQN, REINFORCE, PPO, Actor-Critic 비교.
  - 중급 환경: MountainCar-v0 에서 정책 기반(REINFORCE)과 값 기반(DQN) 비교.
  - 고급 환경: BipedalWalker-v3에서 PPO와 Actor-Critic 비교.

초급 환경

초기 테스트를 통해 각 알고리즘(DQN, REINFORCE, Actor_Critic, PPO)의 간단한 개요에 대해 알아봄.

비교적 간단한 환경에서 실행을 통해 확인함.

초급 환경– Cart Pole



Goal – 막대(pole)가 떨어지지 않도록 수레(cart)를 왼쪽 또는 오른쪽으로 움직이는 것.

State

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

Action

Num	Action
0	Push cart to the left
1	Push cart to the right

Reward – 매 스텝마다 +1.

초급 환경– DQN

- **IDEA: Action-Value 함수 Q(s,a)를 신경망으로 근사**

- **중요 구성: Experience Replay, Target Network**
  - **Experience Replay(Replay) – temporal correlations between samples 해결**
  - **Target Network – non-stationary target 해결**

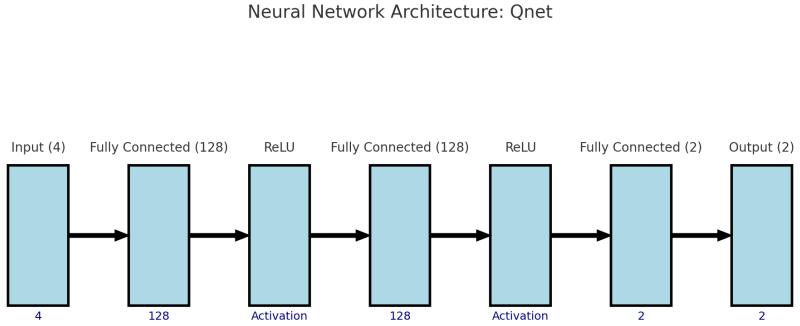
- **Discrete Action Space에 적합**

초급 환경– DQN

- **Parameters**
  - **Learning_rate – 0.001**
  - **Gamma – 0.98**
  - **Buffer_limit = 50000**
  - **Batch_size = 64**

```
learning_rate = 0.001
gamma = 0.98
buffer_limit = 50000
batch_size = 64
```

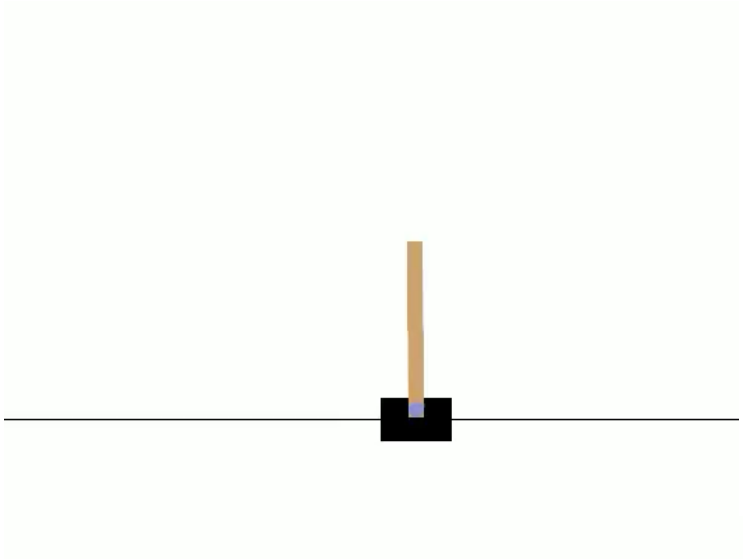
- **Q-Network구조**

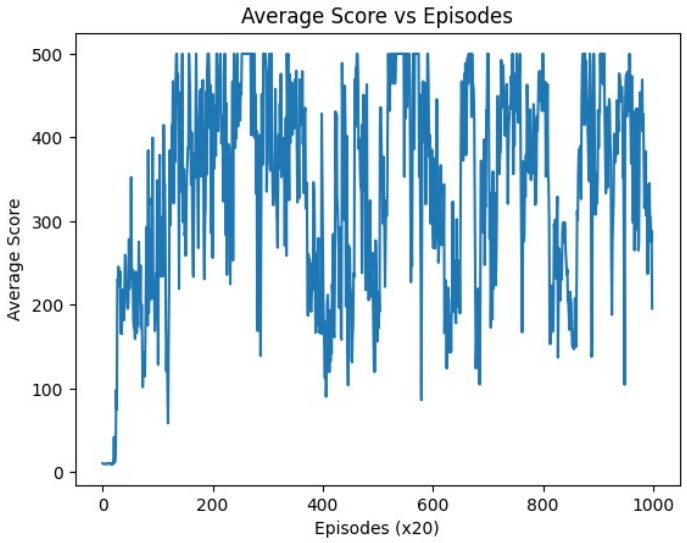


초급 환경– DQN

- 10,000 에피소드 학습 결과
  - 10번에 한 번씩 평균 score 출력
  - 초기에 최적값에 도달함.
  - 엡실론과 신경망 구조, 환경의 랜덤 요소로 인해 점수의 변동 폭이 커진 것으로 추정.

- 시연 영상





초급 환경– REINFORCE

- **IDEA: Monte Carlo 방식으로 Policy Gradient 계산(π (a | s))**

$$\text{Objective function: } J(\theta) = \mathbb{E}_{\pi_\theta}[r(\tau)] = \int p(\tau;\theta)\, r(\tau)\, d\tau$$

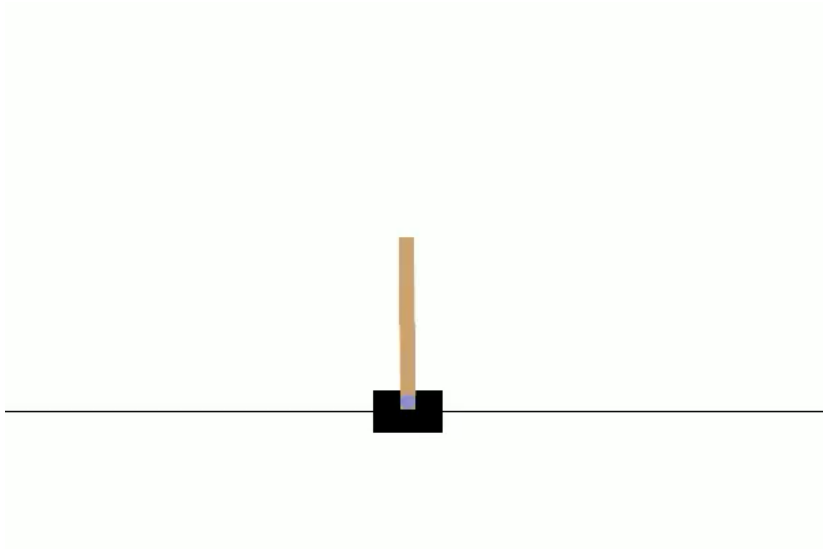
$$\text{Policy gradient update: } \theta := \theta + \alpha \nabla_\theta J(\theta)$$

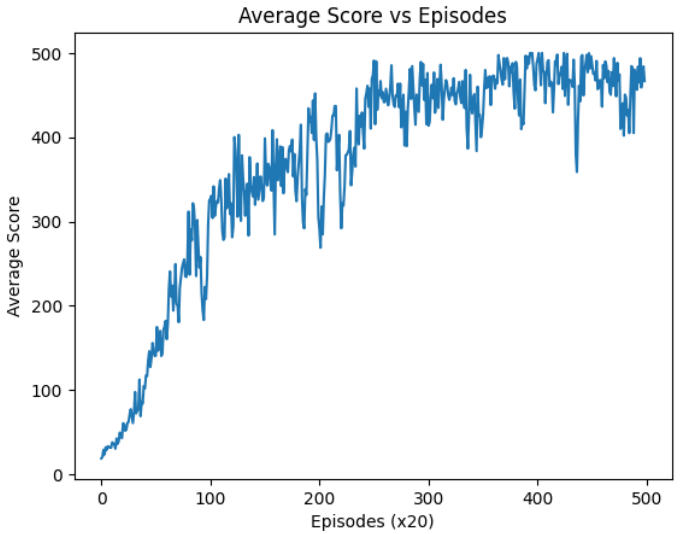
1. **환경과 상호작용 – Trajectory**
2. **G_t 계산, J(θ) 구함**
3. **Policy Gradient update**
4. **반복**

초급 환경– REINFORCE

- 10,000 에피소드 학습 결과
  - 20번에 한 번씩 평균 score 출력
  - DQN에 비해 더 많은 에피소드 후에
  - 최적값에 도달하지만 이후에는 더 안정된 결과를 보임.

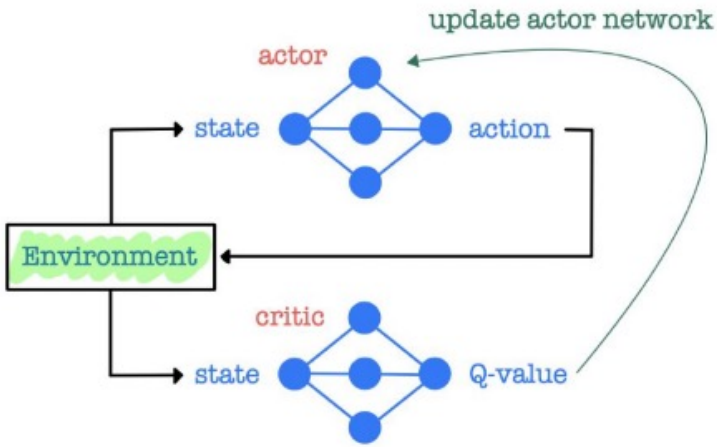
- 시연영상

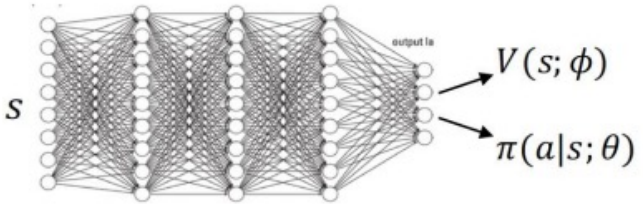




초급 환경– Actor-Critic

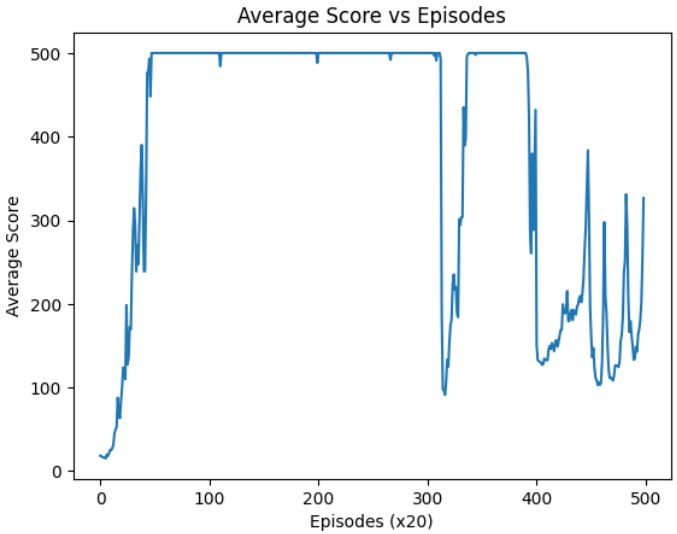
- **IDEA: Actor(policy)와 Critic(value function)을 같이 사용**
- **TD error를 사용한 TD Actor-Critic**



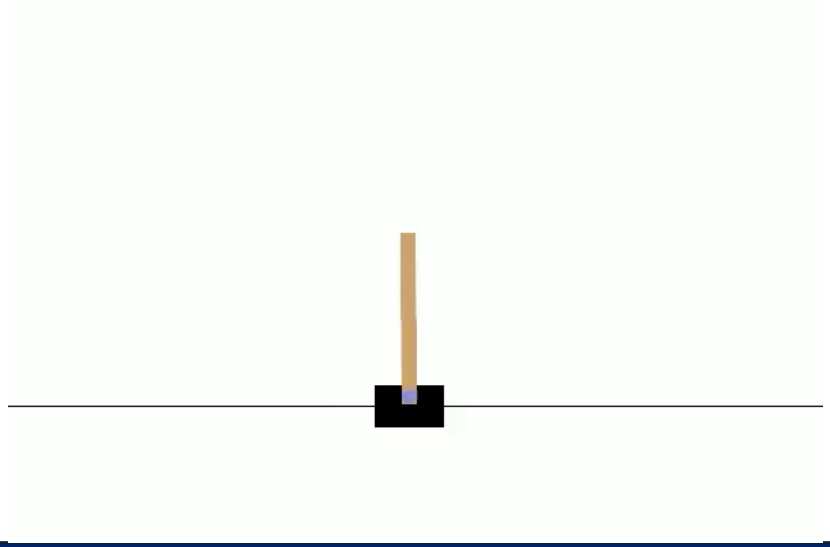


초급 환경– Actor-Critic

- 10,000번 에피소드 학습 결과
  - 20번에 한 번씩 평균 score 출력
  - 초기에 최적값에 도달하고 안정적임.
  - 중후반에 갑작스러운 성능 저하 및 변동성
  - 하이퍼파라미터 등, Critic의 학습 불안정성



- 시연영상



초급 환경– PPO

- **IDEA: 정책 갱신 시 변화 폭을 제한하여 안정적인 학습**

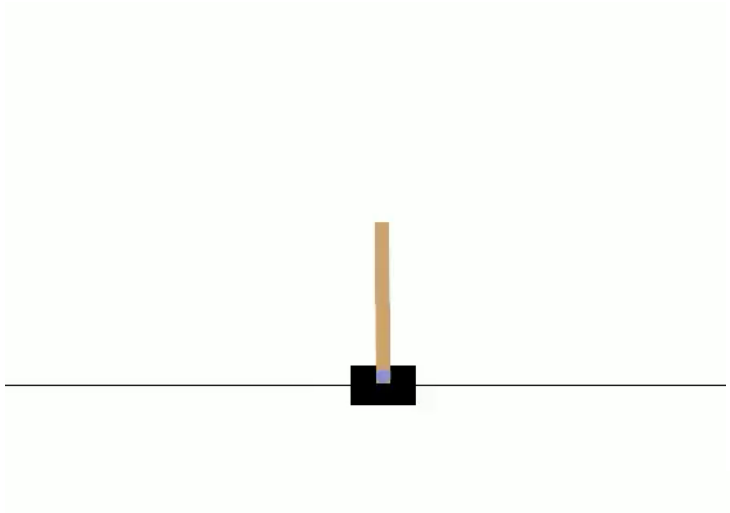
- **기존 정책과 새로운 정책의 차이를 클리핑(Clipping)이라는 기법으로 제한.**
- **정책 업데이트 시 아래의 식 사용**

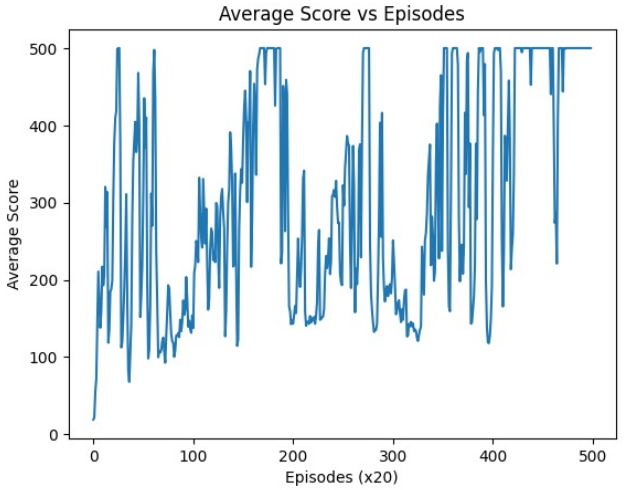
$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta) \cdot A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \cdot A_t) \right]$$

초급 환경– PPO

- 10,000번 에피소드 학습 결과
  - 20번에 한 번씩 평균 score 출력
  - 초기에 최적값에 도달
  - 이후 변동성이 큰 이유는 exploration과 exploitaion을 병행하기 때문
  - 후반에 점차 안정됨이 보이고 점수가 낮아 지는 순간은 탐험과정에서의 일시적인 성능저하로 보임

- 시연영상





초급 환경– 결론

결론

- **각 알고리즘의 간단한 개요에 대해 알아봄**
- **Cartpole 환경이 쉬운 환경이기에 별도의 튜닝 없이도 4가지 모두 최적에 도달함.**
- **이후 비교적 심화된 환경에서 정책기반, 값 기반 알고리즘의 차이를 볼 예정**

MountainCar-v0는 목표 위치에 도달하기 위해 충분한 속도를 얻는 것이 중요한 환경.

중급 환경– MountainCar-v0

Goal – : MountainCar-v0는 목표 위치에 도달하기 위해 충분한 속도를 얻는 것이 중요한 환경.

State

Num	Observation	Min	Max	Unit
0	position of the car along the x-axis	-1.2	0.6	position (m)
1	velocity of the car	-0.07	0.07	velocity (v)

Action

Action Space

There are 3 discrete deterministic actions:

- 0: Accelerate to the left
- 1: Don't accelerate
- 2: Accelerate to the right

MountainCar-v0는 목표 위치에 도달하기 위해 충분한 속도를 얻는 것이 중요한 환경.

중급 환경– MountainCar-v0

Reward – DQN

```
while not done:
    a = q.sample_action(torch.from_numpy(s).float(), epsilon)
    step_result = env.step(a)  # 반환 값을 변수로 저장
    if len(step_result) == 5:
        s_prime, r, terminated, truncated, info = step_result
    else:  # truncated가 없는 경우
        s_prime, r, terminated, info = step_result
        truncated = False

    done = terminated or truncated  # 종료 조건 업데이트

    position, velocity = s_prime
    if velocity < 0 and a == 0:
        r += 2
    elif velocity > 0 and a == 2:
        r += 2
    elif velocity < 0 and a == 2:
        r -= 1
    elif velocity > 0 and a == 0:
        r -= 1
    if position >= 0.5:
        r += 100

    r += abs(velocity) * 0.1
    r += abs(position) * 0.1
    r += max(0, 200 - t) * 0.01

    if done and position < 0.5:
        r -= 50
```

MountainCar-v0는 목표 위치에 도달하기 위해 충분한 속도를 얻는 것이 중요한 환경.

중급 환경– MountainCar-v0

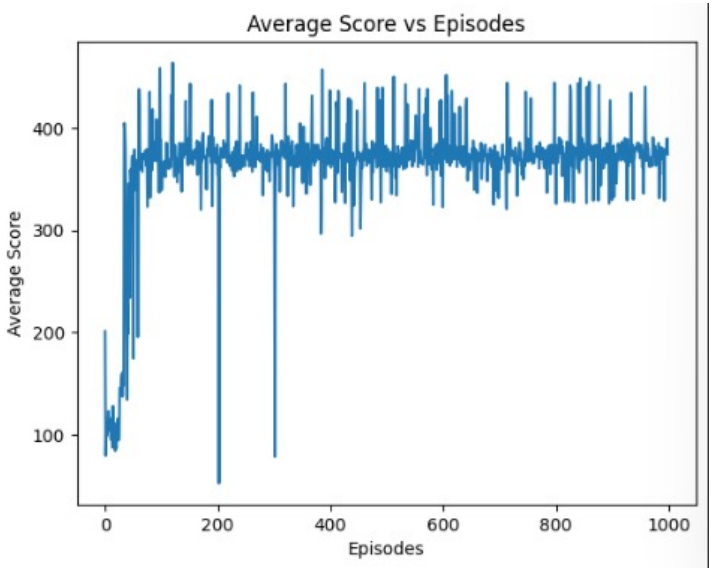
Reward - REINFORCE

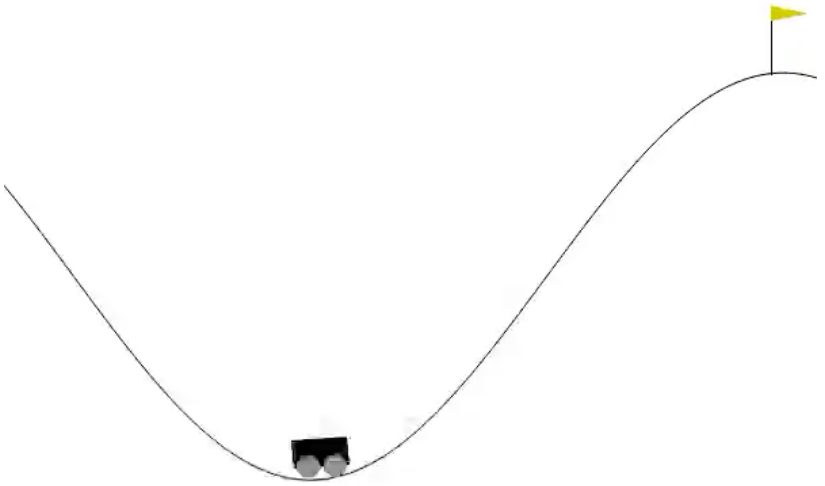
```python
def get_reward(self, state, velocity):
    # 목표 위치 도달 시 큰 보상
    if state >= 0.5:
        return 100
    # 시작점보다 오른쪽으로 이동
    elif state > -0.4:
        return (1.0 + state) ** 2 + abs(velocity) * 10
    # 시작점보다 왼쪽으로 이동
    elif state < -0.4:
        return (1.0 + state) ** 2 + abs(velocity) * 10
    # 기본 보상 (속도 기반 추가)
    else:
        return abs(velocity) * 5
```

중급 환경– DQN

- **Parameters**

```
learning_rate = 0.005
gamma = 0.98
buffer_limit = 50000
batch_size = 64
```

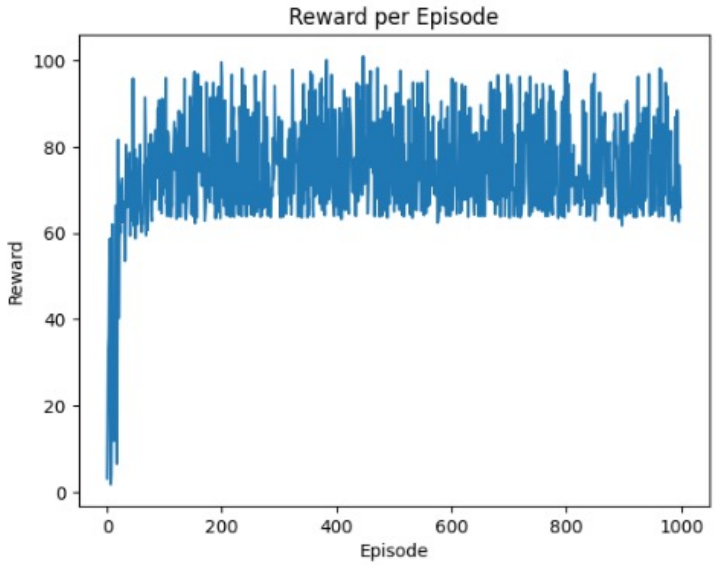




중급 환경– REINFORCE

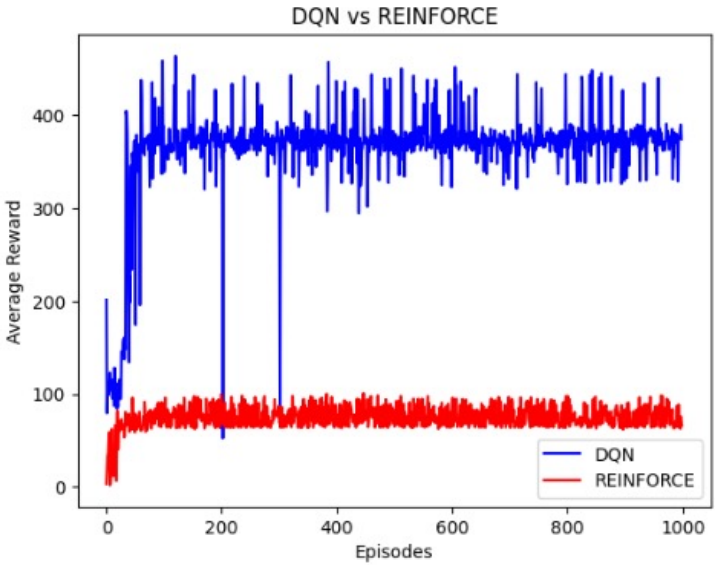
- **Parameters**

```
def main():
    # Hyperparameters
    learning_rate = 0.005
    gamma = 0.99
    hidden = 128
    episodes = 1000
```



중급 환경– DQN vs REINFORCE

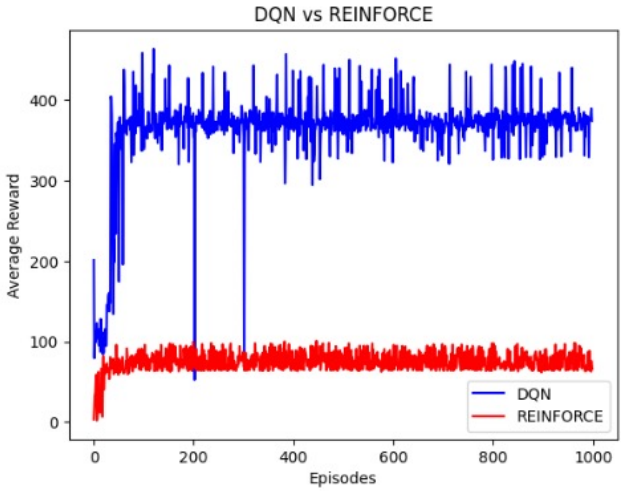
- DQN은 초기에 빠르게 학습을 하여 100 에피소드부터는 reward 300~400 사이 결과를 보임
- REINFORCE 또한 초반 100에피소드 까지는 빠르게 학습하였지만 DQN과는 달리 reward 100 이상으로는 올라가지 못함



중급 환경– DQN vs REINFORCE

- DQN의 경우 목표에 한 번이라도 도달했기 때문에 그에 따른 보상으로 이후에도 점수가 높게 나온 것으로 보았으며 REINFORCE도 목표에 도달한다면 이후 점수가 높아질 것으로 생각했다.

- 때문에 학습률 조정, 보상 조정, env._max_episode_steps조정, 에피소드 조정을 해보았지만 결과가 좋아지지는 않았다. 수정을 거듭한 끝에 현재 코드가 가장 좋은 모습을 보였고 영상을 확인해 보았지만 성공한 경우는 없었다.

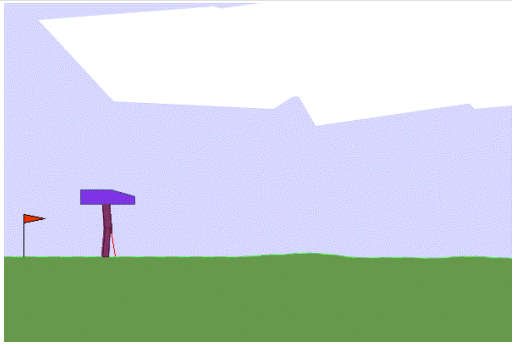


중급 환경– DQN vs REINFORCE

- **Reinforce 결과 고찰**
  - **dqn은 off-policy 이고, Reinforce 의 경우 on-policy 이다. dqn은 학습의 target이 state 상에서 받을 수 있는 가장 큰 가치를 사용하여, 항상 최고의 상태값을 받지만, Reinforce은 확률적으로 정책을 선택하므로 상황에 맞는 다른 가치를 받게 된다.**

  - **상황에 따라 다른 가치를 받아 정책을 발전시키는 것이 Policy Gradient 의 의미인데 반해, Mountain car에서 문제는 상황에 따른 것이 아닌 꾸준한 보상이 쌓여 행동을 개선시켜 나가야 되는데, REINFORCE는 보상 자체가 쌓이기 어려운 환경이라는 것**

고급 환경– Bipedal Walker



Goal – 두 발 로봇이 출발점에서 끝점까지 걷는 동안 최대한 효율적으로 이동하도록 학습

Observation Space: 각 관절의 각도 및 속도, 로봇 몸체의 위치와 속도 등 24차원의 연속적인 벡터

Action Space: 4개의 관절(hip과 knee)의 속도. 4차원의 연속적인 벡터

Reward - 로봇이 앞으로 이동한 거리만큼 보상을 부여. 과도한 에너지 사용에는 페널티 부여.

고급 환경– DQN

- 시도
  - 일반적인 방식의 DQN 적용

  - BipedalWalker-v3 환경의 행동 공간은 연속적(Box 타입)이며, 각 차원의 값은 [-1, 1] 범위
  - 해당 공간을 이산적(Discrete 타입)으로 변환하기 위해 DiscreteActionWrapper 클래스가 사용됨

  - 주요 메서드:
  - _create_action_map() : 연속적 행동 공간을 이산적 행동으로 변환하기 위해 각 차원의 값을 선형적으로 나눔
  - np.linspace(-1, 1, n_actions_per_dim)를 사용하여 각 차원의 값을 균등하게 나눔.
  - np.meshgrid와 reshape를 사용하여 가능한 모든 행동의 조합을 생성.

  - 예를 들어, 각 차원에서 5개의 이산적 값([-1.0, -0.5, 0.0, 0.5, 1.0])으로 나눈다면, 총 가능한 행동 조합의 수는 5차원 수(5^차원 수)가 됩니다. BipedalWalker-v3는 4차원 행동 공간이므로 총 5^4=625개의 행동 조합이 만들어짐

  - action(self, act)
  - 이산적 행동(act)을 받아서 사전에 생성된 action_map에서 해당하는 연속적 행동 값을 반환.

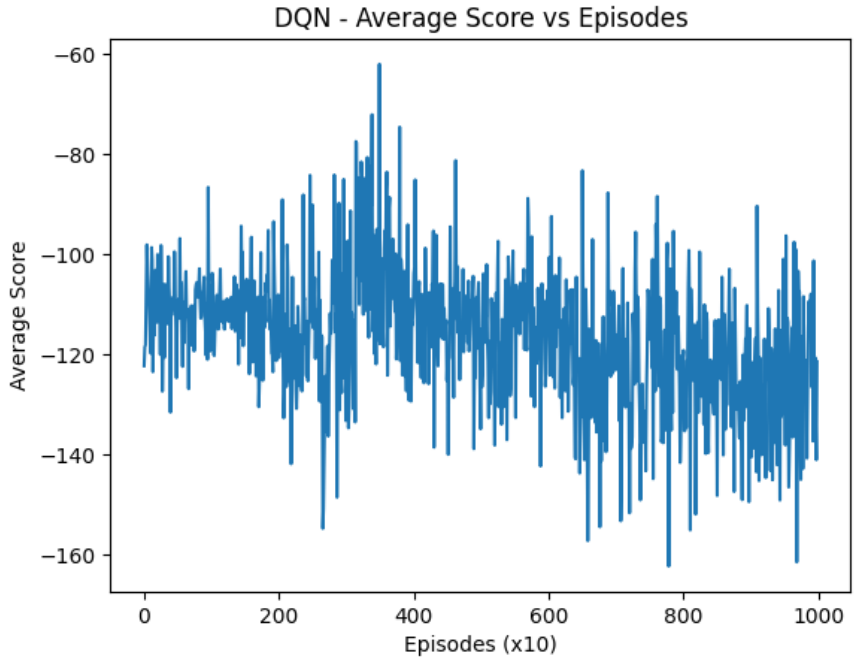
```python
# Custom discrete action wrapper
class DiscreteActionWrapper(gym.ActionWrapper):
    def __init__(self, env, n_actions_per_dim=5):
        super().__init__(env)
        self.n_actions_per_dim = n_actions_per_dim
        self.action_space = Discrete(n_actions_per_dim ** env.action_space.shape[0])
        self._create_action_map()

    def _create_action_map(self):
        """Creates a mapping from discrete actions to continuous actions."""
        n_dims = self.env.action_space.shape[0]
        action_ranges = [np.linspace(-1, 1, self.n_actions_per_dim) for _ in range(n_dims)]
        self.action_map = np.array(np.meshgrid(*action_ranges)).T.reshape(-1, n_dims)

    def action(self, act):
        return self.action_map[act]
```

고급 환경– DQN

- 실패 사례 및 고찰
  - 일반적 방식의 DQN은 Bipedal Walker 환경에서 올바르게 작동하지 않았음
  - (성능, -160 ~ -60 사이)
  - 코드가 환경에서 제대로 작동하지 않는 이유에 대한 고찰
  - 행동 이산화의 한계: BipedalWalker-v3는 연속 행동이 필요한 환경이나 6단계로만 이산화 했기에 세밀한 조작을 반영하지 못해 학습이 어려워짐.
  - DQN은 일반적으로 상태-행동 쌍이 적고, 연속적이지 않은 문제에 더 적합한 알고리즘이기 때문에 예정된 문제일 수 있음
  - 연속 행동 환경에서는 DDPG, SAC, PPO와 같은 연속 제어를 지원하는 알고리즘이 더 적합할 것으로 판단



고급 환경– Actor-Critic & PPO

- **Actor-Critic**
  - **Actor(정책)과 Critic(가치 평가)를 각각 정의하여 정책을 직접 업데이트**
  - **정책 개선 단계에서 클리핑이나 제한을 사용하지 않음**
  - **간단하고 계산량이 적음.**

- **PPO**
  - **Actor-Critic에 기반하지만, 정책 업데이트에 안정성을 추가하기 위해 클리핑 기법을 사용.**
  - **정책 비율을 계산하고, 엡실론 클리핑을 적용**
  - **더 안정적이고 효율적인 학습 가능**

고급 환경– Actor-Critic & PPO

- 정책 업데이트 방식의 차이
  - Actor-Critic
    - 단순히 TD error를 이용하여 정책을 개선

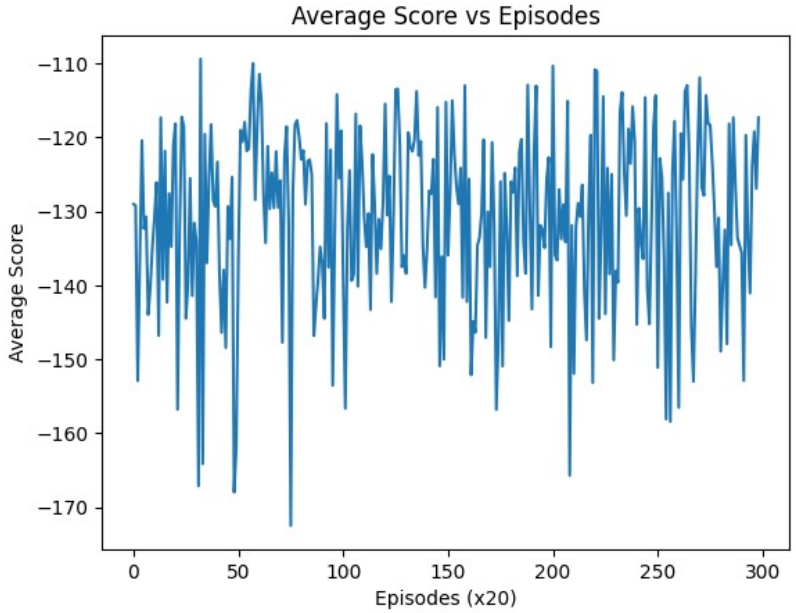
```
loss_pi = -(log_prob * delta.detach()).mean()
loss_v = F.smooth_l1_loss(self.v(s), td_target.detach())
loss = loss_pi + loss_v - 0.02 * entropy
```

  - PPO
    - Ratio: 새로운 정책과 이전 정책의 비율.
    - 클리핑(1 - eps_clip, 1 + eps_clip)으로 정책 업데이트의 변동성을 제한

```
ratio = torch.exp(log_prob - prob_a)
surr1 = ratio * advantage
surr2 = torch.clamp(ratio, 1 - eps_clip, 1 + eps_clip) * advantage
loss = -torch.min(surr1, surr2) + F.smooth_l1_loss(self.v(s), td_target.detach())
```

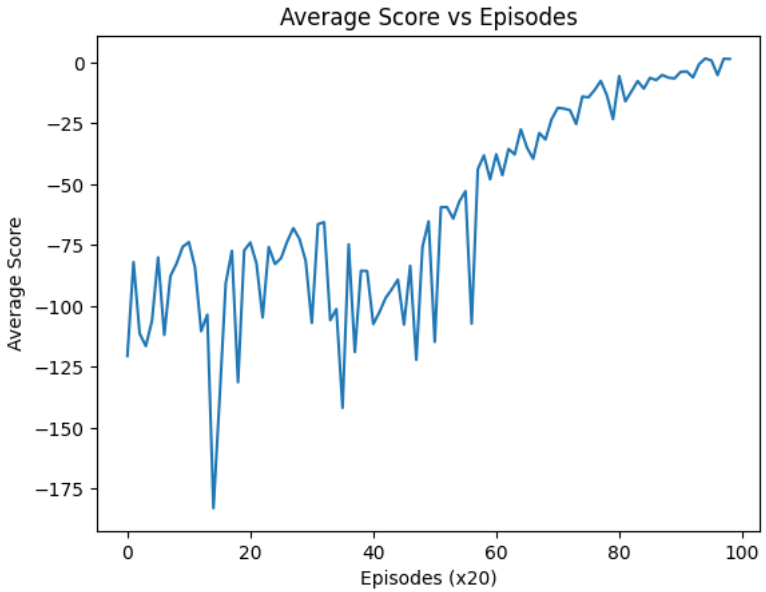
고급 환경– Actor-Critic & PPO

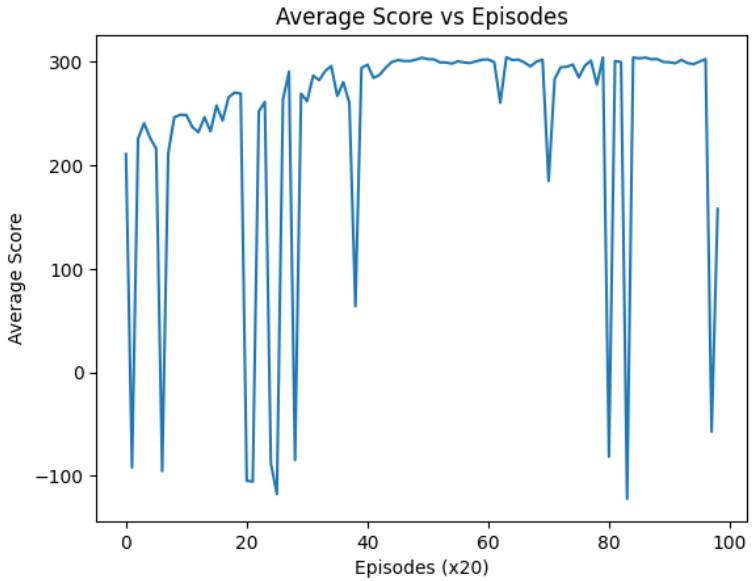
- **Actor-Critic 결과**
  - **3000번의 에피소드 동안 수렴하지 못하고 높은 reward에도 도달하지 못함.**



고급 환경– Actor-Critic & PPO

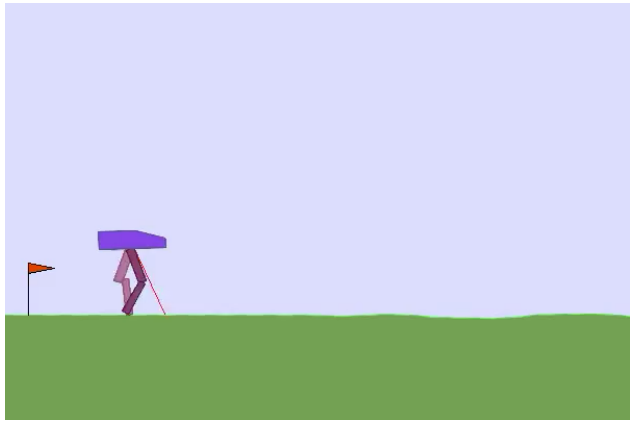
- PPO 결과
  - 총 2000번의 에피소드를 통해 최적값에 도달하였고, 이후 변동성을 보이긴 하지만 이는 탐험과 이용 문제로 추정.



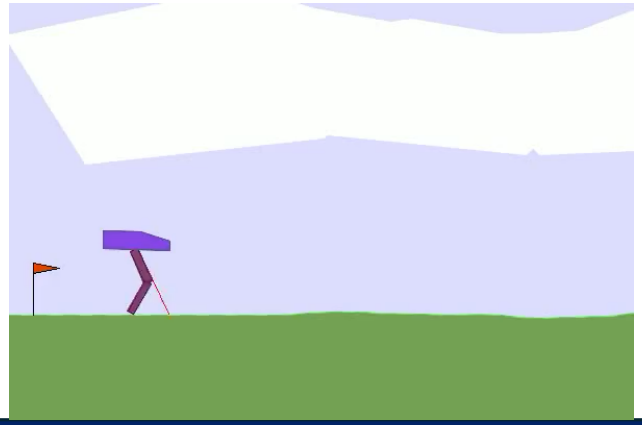


고급 환경– Actor-Critic & PPO

- **Actor-Critic 시연 영상**



- **PPO 시연 영상**



고급 환경– Actor-Critic & PPO

- 결론
  - 결과적으로 PPO의 성능이 Actor-Critic에 비해 잘 나옴.
  - Actor-Critic의 파라미터 튜닝으로 성능 향상 가능성이 있지만 PPO를 사용함.
  - PPO알고리즘을 통해 학습 안정성을 개선시켰고, 좋은 성능을 보임.

Thank you

Github link : https://github.com/gkstmdgnsgong/reinforceproj_2024_2.git