인제고 공학일반

연구보고서

저사양 하드웨어에서의 강화학습 성능에 관한 고찰

학번 31013

이름 이준혁

목 차

Ⅰ. 서론

LLM(GPT, Gamini)나 CNN(자율주행자동차, 알파고)이 등장함에 따라 과거보다 인공지능이 우리 삶에 밀접하게 영향을 주고 있다. 이러한 인공지능이 상용화되기 위해선 비용문제가 해결되어야 한다. 예를들어 GPT나 Gamini등은 클라우드 시스템을 활용하여 아무리 저사양 하드웨어더라도 인터넷과 브라우저만 실행이 가능하다면 고성능 인공지능을 사용할 수 있도록 하였다.

하지만 휴머노이드 로봇을 만든다거나 인공지능을 우리의 삶에 더 가까이 다다가게 하기 위해선 저사양 하드웨어 환경에서도 인공지능 모델이 작동할 수 있어야 한다. 또한 선행연구들을 조사해보면 저사양 하드웨어에서 강화학습 모델의 최적화 연구는(적어도 국내에선) 활발히 진행되고 있지 않다.

나는 미래의 AI기술의 핵심은 AI가 스스로 생각할 수 있도록 유도하는 강화학습이라고 생각한다. 따라서 강화학습 모델의 최적화는 반드시 해결되어야할 과제이다. 따라서 이번 연구에선 간단한 강화학습 모델과 저사양 하드웨어(라즈베리파이나 리소스를 많이 사용하고 있는 노트북 등)을 이용해 실제로 모델을 구축해 유의미한 최적화가 구현 가능한지 탐구해보고자 한다.

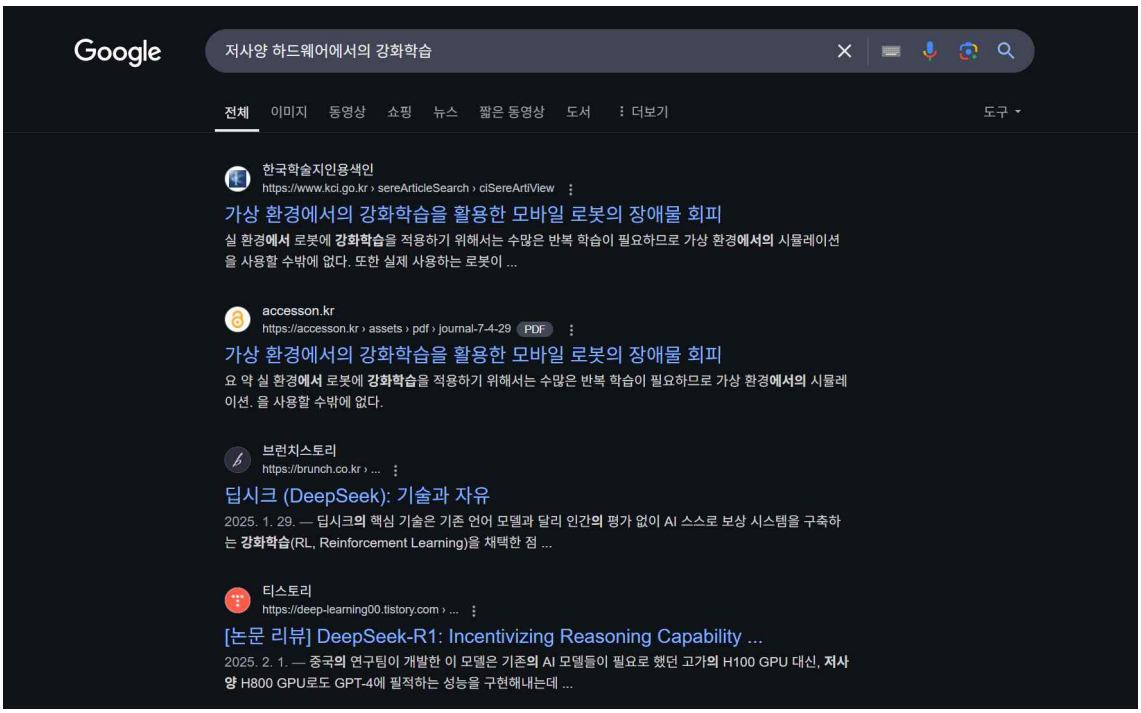
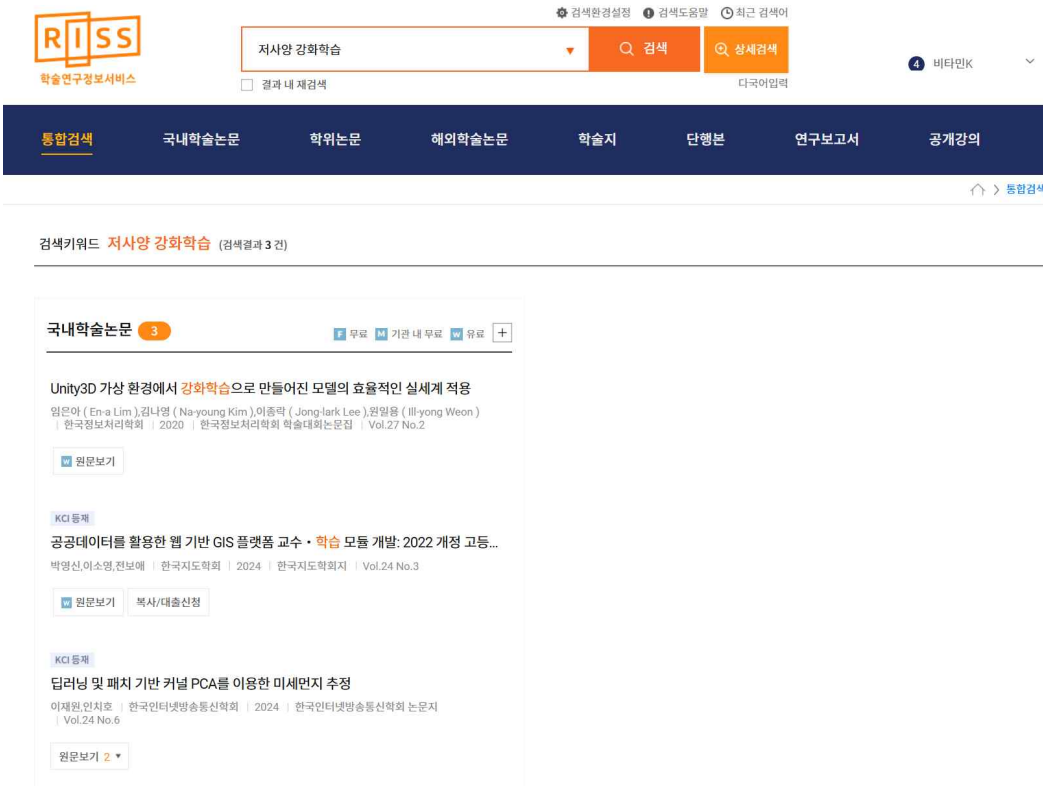


사진1,2 : 저사양 하드웨어에서 강화학습 모델의 최적화에 관한 연구는 거의 없다.


RISS
학술연구정보서비스
검색환경설정
검색도움말
최근 검색어
저사양 강화학습
검색
상세검색
결과 내 재검색
다국어입력
4 비타민K
통합검색
국내학술논문
학위논문
해외학술논문
학술지
단행본
연구보고서
공개강의
통합검색
검색키워드 저사양 강화학습 (검색결과 3 건)
국내학술논문 3
F 무료 M 기관 내 무료 W 유료
Unity3D 가상 환경에서 강화학습으로 만들어진 모델의 효율적인 실세계 적용
임은아 (En-a Lim),김나영 (Na-young Kim),이종락 (Jong-lark Lee),원일용 (Ill-yong Weon)
한국정보처리학회 2020 한국정보처리학회 학술대회논문집 Vol.27 No.2
W 원문보기
KCI 등재
공공데이터를 활용한 웹 기반 GIS 플랫폼 교수・학습 모듈 개발: 2022 개정 고등...
박영신,이소영,전보애 한국지도학회 2024 한국지도학회지 Vol.24 No.3
W 원문보기
복사/대출신청
KCI 등재
딥러닝 및 패치 기반 커널 PCA를 이용한 미세먼지 추정
이재원,인치호 한국인터넷방송통신학회 2024 한국인터넷방송통신학회 논문지
Vol.24 No.6
원문보기 2

Ⅱ-Ⅰ. 머신러닝과 딥러닝

A. 머신러닝

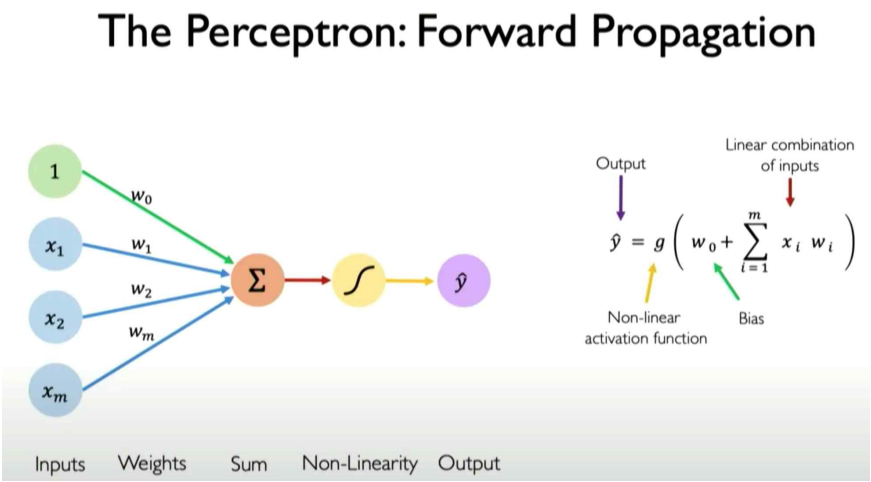
강화학습은 크게보면 머신러닝의 하위집합이다. 인공지능의 하위집합이 머신러닝이고 머신러닝 안에 딥러닝과 강화학습, 지도학습, 비지도학습이 있다. 머신러닝이란 데이터를 수치화하고 그 값들을 바탕으로 타겟 데이터를 예측하거나 데이터들을 분류 하는 등 기존에 간단한 알고리즘들로는 수행할 수 없었던 작업들을 수행할 수 있도록 해주는 컴퓨팅 기술이다.

B. 딥러닝

강화학습에 관한 연구에서 딥러닝을 언급하는 이유는 여러 강화학습 기법들은 거의 다 딥러닝에 기반을 두고있거나 최소한 딥러닝을 이용하고 있다. 이번 연구에서 주로 사용할 DQN만 생각해도 Deep Q-learning Network의 약자로 딥러닝에 기반을 둔 알고리즘이다.

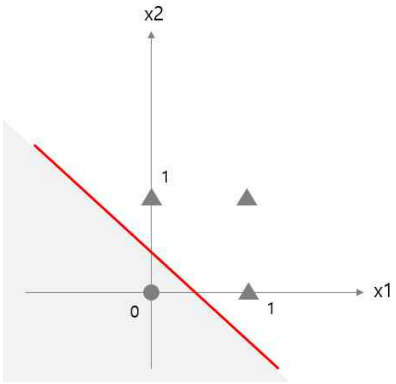
1. 퍼셉트론(perceptron)

딥러닝 기술의 가장 기본적인 구성요소다. Perceptron 또는 뉴런(셀)이라고 부르는 이것은 데이터를 받으면 데이터에 가중치를 곱하고 편향을 더한다. 그리고 결괏값을 활성화함수(activation function)에 대입하는데 활성화함수의 역할은 밑에서 서술하겠다. $\overline{Y} = weight \times X_t + bias$, $Y_{t+1} = activation(\overline{Y})$로 표현된다. 이는 우리 뇌를 모방한 기술로 우리 뇌 안에 있는 뉴런들이 데이터를 전기신호로 전달하고 서로 그 값들을 조작하면서 데이터를 해석하는 과정을 모방한 것이다. 딥러닝 모델은 이 weight와 bias를 최적으로 만들기 위해서 학습하는데 이 weight나 bias등을 파라미터(parameter)라고 한다. 그리고 모델을 학습시키는 사람이 조정하는 활성화함수나 여러 상수 값들이 있는데 이는 모델이 학습하지 않는 즉 모델을 만드는 사람이 조정하는 상수라서 하이퍼 파라미터라고 부른다. 강화학습을 예로들면 할인률등이 하이퍼 파라미터이다.

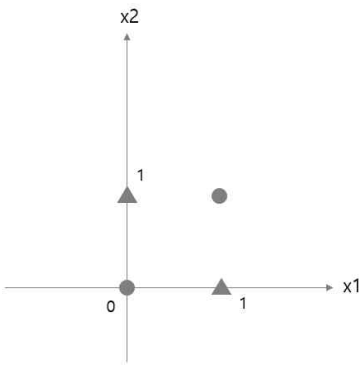


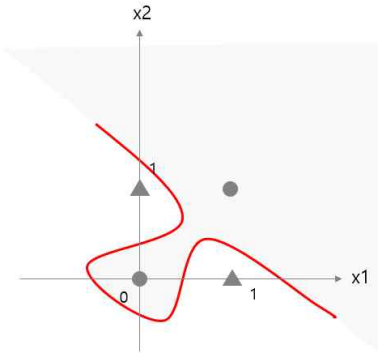
2. 활성화 함수(activation function)

딥러닝은 데이터들의 관계를 비선형적으로 학습해야 한다. 유명한 문제인 XOR문제로 예시를 들어보자. 아래 그래프에선 ▲모양을 분류하기 위해선 선형적인 관계로도 충분히 분류할 수 있다. (다항함수는 선형적인 관계(y=ax^n+⋯+b)꼴로 표현 가능). 때문에 activation function이 크게 중요하지 않을 것이다.



아래 그래프에선 우리가 위에서 사용한 직선으로만 해당 데이터들을 분류하는것엔 무리가 있을 것이다. 그러나 직선이 아닌 곡선을 이용한다면 위 그래프의 데이터를 충분히 분류할 수 있을 것이다. activation function이 하는 일은 이러한 데이터들을 비선형적 관계로 분류할 수 있도록 만들어주는 것이다. (non-activation perceptron은 y=ax+b의 1차원 형태밖에 표현 못함)

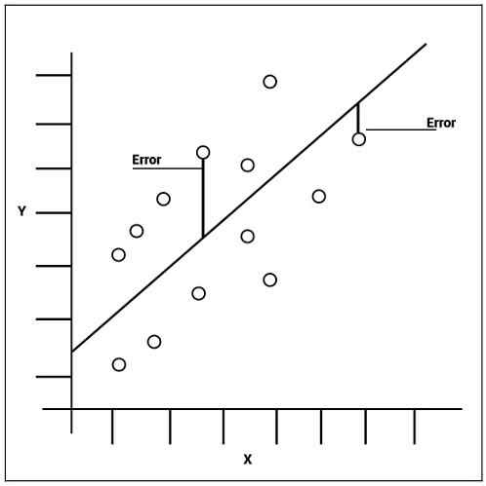




3. 손실함수(loss function)

손실함수는 딥러닝 뿐만 아니라 머신러닝에서 가장 중요한 과정인데, 모델이 수행한 행동(데이터를 얼마나 정확하게 예측했는지, 데이터를 얼마나 잘 분류했는지, 과적합이나 과소적합이 일어나진 않았는지 등등)을 평가해주는 함수이다. loss function은 프로젝트에 따라 각기 다른 형태를 취할 수 있는데 주로 사용하는 MSE를 예시로 들어 설명하겠다.

$$L(y, \hat{y}) = MSE = \frac{1}{n}\sum_{i=1}^{n}(y-\hat{y})^2$$



위 그래프에서 동그라미는 실제 데이터이고 직선이 모델이 예측한 값이다. 실제 데이터에 대응되는 점과의 오차가 가장 적어야 하므로 직선과 각 점들과의 거리의 합을 계산하고 그 값을 error 즉 손실이라고 하는 것이다. 왜냐하면 이 값이 작으면 작을수록 모델이 데이터를 더 잘 예측함을 의미하기 때문이다. (물론 모델이 너무 학습데이터에 편향되면 과대적합등의 문제가 일어날 수 있다.) 근데 굳이 제곱을 해주는 이유는 우린 y(실제값) - $\hat{y}$(예측값)을 계산하는데 이 값이 음수인지 양수인지 알 수 없기 때문이다. 우리는 오류의 정도가 필요한거지 그 오류의 방향이나 부호는 중요치 않다. 따라서 절댓값을 취하거나 제곱을 하는데, 컴퓨터 특성상 주로 제곱을 한다.

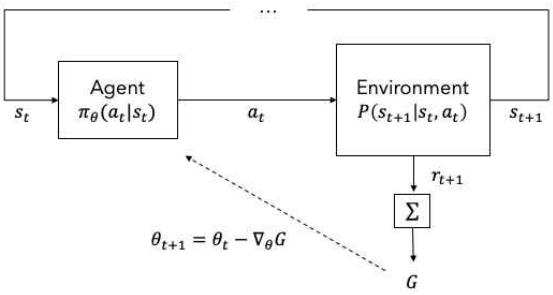
4. 옵티마이저(Optimizer)

Optimizer는 loss function을 최소로 하는 파라미터 값을 찾아주는 알고리즘이다. Optimizer의 대표적인 예시로는 세특 주제로 매우 자주 쓰이는 경사하강법(Gradient Descent)가 있다. 경사하강법은 너무 널리 알려지기도 했고 Optimizer는 이번 프로젝트에서 중요한 부분이 아닐뿐더러 한번 서술하기 시작하면 너무 오래걸려서 생략하도록 하겠다.

Ⅱ-Ⅱ. 강화학습 모델

A. 강화학습 원리

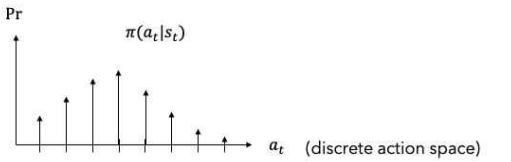
강화학습(Reinforcement Learning)은 여러 인공지능 학습 방식 중에서도, 가장 직관적으로 인간의 학습 과정을 모방한 방법이다. 모델(Agent)이 특정 행동(action)을 수행하면, 이에 대한 보상(reward)을 받고, 이 보상을 최대화하는 방향으로 점차 더 나은 행동을 학습해간다.
(https://tutorials.pytorch.kr/_images/cartpole.gif 참고)

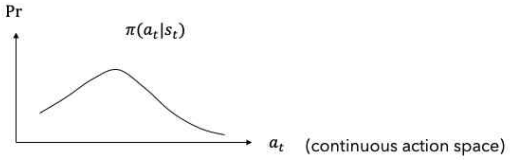


위 그림은 가장 간단한 강화학습 모델을 도식화한 것이다. Agent가 모델이고 Environment가 모델이 움직이는 환경(스타크래프트 AI면 맵, 종족, 미네랄 수 등 현재 상태)이다.
s_t는 Agent에게 환경이 변화할 때마다 변한 환경을 전달해주는 매개고 a_t는 환경에 영향을 미치는 모델의 행동(action)이다. r_{t+1}은 행동 a_t에 대한 보상이고 G는 보상의 전체 합이다. $\theta_{t+1} = \theta_t - \nabla G$은 경사하강법을 나타낸 식으로 이번 프로젝트에서 별로 중요하지 않으니 넘어가겠다. $\pi_\theta(a_t|s_t)$는 상태 s_t일 때 행동 a_t를 할 확률이고 $P(s_{t+1}|s_t, a_t)$는 환경 s_t일 때 행동 a_t이 환경 s_{t+1}를 만들 확률이다. 밑에서 자세히 설명하도록 하겠다.

B. 에이전트와 정책(Agent and Policy)

정책이란 어떤 상태에서 어떤 행동을 할지 에이전트에게 알려주는 전략이다. 확률적 정책이 있고 결정적 정책이 있다. 확률적 정책은 이 상황에선 1번 행동을 0.6의 확률로 시행하고 2번 행동을 0.3, 3번 행동을 0.1 이런식으로 에이전트가 행할 행동을 확률로 알려주는거고 결정적 정책은 어떤 상태에선 이것만 해라 라는 정책이 된다. 보통 어떤 상태에서 어떤 행동이 가장 최선인지 알기 어려우므로 확률적 정책을 많이 사용한다. 앞으로 π가 나오면 정책(주로 확률적 정책)이라고 생각해주면 된다. 이건 MDP를 이해해야 더 잘 이해할 수 있으므로 밑에서 추가적으로 설명하겠다.





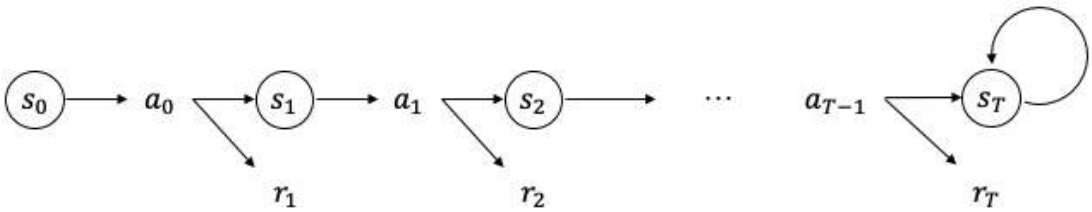
위 그래프는 확률적 정책을 이산적으로 분포시킨 그래프와 연속적으로 분포시킨 그래프이다. 이산적인 행동 확률 분포의 예로는 테트리스(시계방향 회전, 반시계방향 회전) 게임이 있을 수 있다. 하지만 현실에서의 행동은 거의 연속적인 분포를 가지기 때문에 주로 연속확률분포를 이용해서 모델의 행동을 결정한다.

$$\sum_{a_t \in A} \pi(a_t|s_t) = 1 \qquad \text{(for discrete action space)}$$

$$\int_{a_t \in A} \pi(a_t|s_t) da_t = 1 \qquad \text{(for continuous action space)}$$

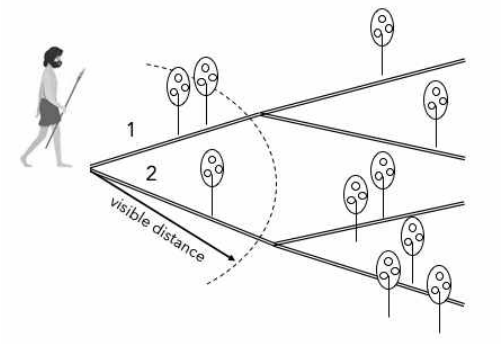
위 식은 행동을 할 확률들의 합은 반드시 1이여야함을 이야기한다.

C. 보상(reward)



위 그림을 보면 상태 s_0에서 출발하여 행동 $a_{0,1,2\cdots}$을 수행하며 보상 $r_{0,1,2\cdots}$을 얻는 그림이다. 별로 어려운 내용은 아니니 설명은 생략하겠다.

$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ 이 식은 모든 보상들의 합, 즉 모델의 그동안의 행동들에 대한 보상들의 합이다. 여기서 모델은 r_n이 최대가 되로록 하는 것이 아닌 G_t가 최대가 되도록 행동을 수행하는데, 이 이유는 아래 그림을 보면 알 수 있다.

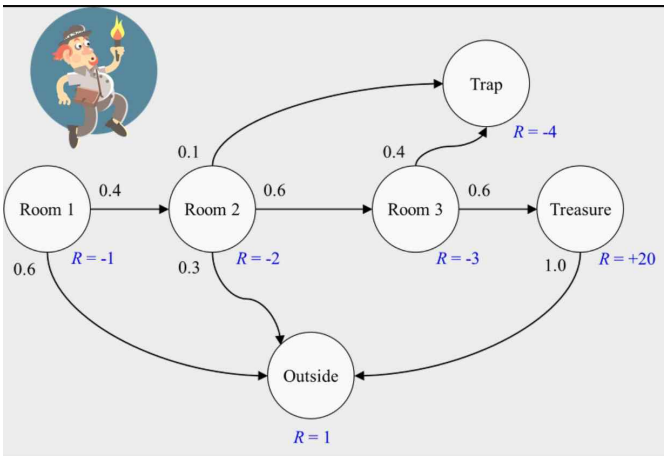


당연히 우리는 2번 경로를 택해야 함을 알지만 원시인의 시야에선 1번 선택이 최선으로 보인다. 눈앞의

보상에만 눈이 멀면 최종 보상의 합이 줄어들기 때문에 눈앞의 보상도 좋지만 나중 보상을 위해서 행동을 수행함도 중요하다. 따라서 위 식에서의 γ를 할인계수라고 하고 0~1사이의 값을 가진다. 할인계수가 0에 가까워 질수록 현재의 보상이 중요하고 할인계수가 1에 가까워질수록 미래의 보상이 중요해진다.

D. 마르코프 결정 과정 (Markov Decision Process) --- MDP

마르코프 결정 과정(MDP)은 강화학습 모델을 개발할 때, 환경을 구성하는 기본적인 수학적 틀이라고 할 수 있다. 이를 제대로 이해하기 위해서는 먼저 마르코프 보상 과정(MRP, Markov Reward Process)에 대한 이해가 필요하다. 이는 상태 전이와 보상이 있는 환경을 의미하며, action은 포함되지 않는다. 강화학습의 보상 구조와 유사한 개념이다.
이러한 내용을 보다 직관적으로 이해하기 위해, 유적을 탐사하는 탐험가를 예시로 들어 설명해보겠다.



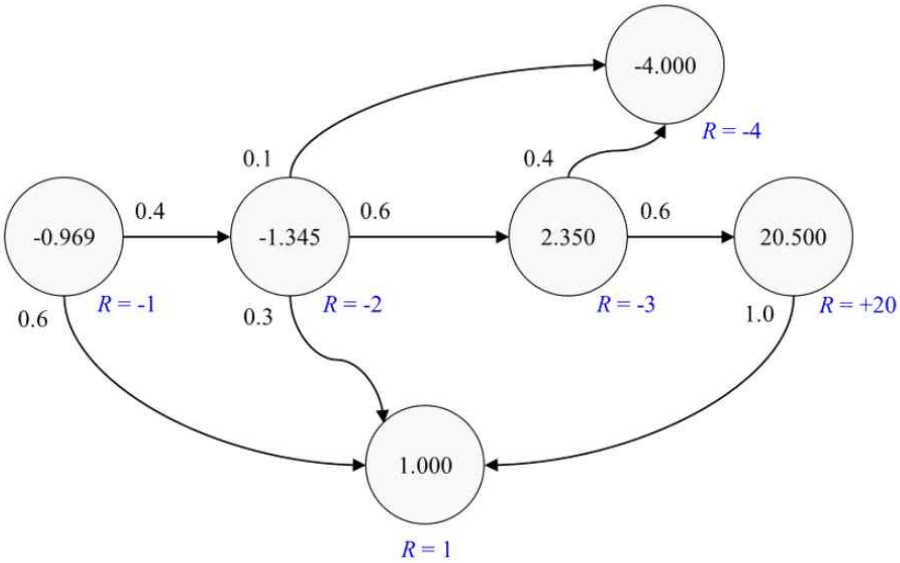
위 사진에서 탐험가는 Room과 trap등 각 state에서 파란색으로 쓰여있는 R이라는 reward를 받는다.

$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ 이 식에 따라 room1에서부터 room1 → room2 → outside의 경로를 거치는 경우의 G는 할인율 γ를 1이라고 가정하면 G = -1 + (-2) + 1 = -2가 된다. 다만 일반적으로는
$G = r_0 + \gamma \cdot r_1 + \gamma^2 \cdot r_2 + \cdots$ 의 형태로 계산한다.

일일이 계산해서 직관적으로 경로를 찾을 수도 있지만 현실은 저 상황보다 더 복잡하고 변수가 많다. 따라서 최적의 값을 찾기 위해 state-value function과 action-value function이란 개념이 도입되었다.

1) state-value function (V(s))

$V(s) = E[G_t | S_t = s]$ V(s)는 각 state에서 현재 정책을 따라갔을 때 얻을 수 있는 reward의 기댓값(expected return)을 계산하는 함수이다. 쉽게 말하면, 현재 상태가 궁극적인 목표를 얻기 위해 얼마나 좋은 상태인지 판단하는 기준이다. 이는 각 상태를 시작점으로 했을 때 가능한 여러 행동 경로의 G 값을 평균낸 것이라고 볼 수 있다.



위 그림에서 v(s)를 계산하여 표현하면 위 그림과 같이 구해질 수 있다.

2) Markov decision process (MDP)

MDP는 MRP에 행동(action) 개념을 포함한 구조이다. 이렇게 되면 우리의 목적은 MDP에 의해 정의된 문제상황에 따라, 각 상태(state)에서 최적의 G값을 얻게 해주는 행동(action) 이 무엇인지 찾는 것이다. 이때 행동을 선택하는 전략이 정책(Policy, π)이며, 이는 $\pi(a|s)$: 상태 s에서 행동 a를 선택할 확률과 같은 조건부 확률로 표현된다.

3) Action-value function($q_\pi(s,a)$)

Q(s, a)는 상태 s에서 행동 a를 했을 때 얼마나 가치 있는지를 나타내는 함수이다. state-value function이 상태 자체의 가치를 나타낸다면, action-value function은 특정 상태에서 특정 행동을 했을 때의 기대 보상(G) 을 계산한다. 이 값은 Q-value 또는 Q값이라고도 부른다.

결론적으로 마르코프 결정 과정(MDP)에 따라 최적의 state-value function, action-value function을 찾는 것이 강화학습 모델의 최종 목표라고 할 수 있다.

$V_*(s) = \max_\pi V_\pi(s)$ 왼쪽 식은 각각의 함수의 optimization식이다.

$q_*(s,a) = \max_\pi q_\pi(s,a)$

E. Q-learning

1. q-learning: Q-learning은 가장 단순한 형태의 강화학습 알고리즘으로, Q-table을 만들어서 상태-행동 쌍의 가치를 학습한다. 이때 사전학습된 환경 모델이 필요 없기 때문에 model-free 알고리즘이라고 부른다. Q-table에는 각 (state, action) 쌍의 Q값이 저장되며, 이 값은 Bellman Equation을 이용해 점진적으로 업데이트된다.

- 벨만 방정식(Bellman Equation)

강화학습에서 가장 핵심적인 수식 중 하나가 벨만 방정식(Bellman Equation)이다. 이 식은 에이전트가 어떤 상태에 있을 때, 그 상태에서 받을 수 있는 보상의 기대값(V(s)또는 Q(s, a))을 미래의 보상과 현재의 보상의 관계로 나타내주는 식이다. 쉽게 말하면, 현재 상태의 가치는 '당장의 보상 + 다음 상태의 가치'의 합이라는 개념이다.

좀 더 구체적으로 말하자면, 상태 가치 함수 V(s)는 어떤 상태 s에서 시작했을 때, 앞으로 정책 π를 따르며 얻게 될 보상의 기대값이다. 벨만 방정식은 이 기대값을 계산하기 위해 아래와 같은 식을 사용한다.

$V(s) = E[r + \gamma \times V(s')]$

여기서

r은 현재 상태에서 받은 보상,

γ는 할인율(0~1 사이의 값),

s'는 다음 상태,

E[...]는 기대값을 의미한다.

즉, 현재 상태의 가치는 "지금 보상 + 미래 상태의 기대 가치"의 합이다. 이 개념은 모든 상태에 대해 재귀적으로 적용되며, 그렇게 함으로써 전체 환경에서 각 상태의 가치가 어떻게 결정되는지를 수학적으로 정의할 수 있게 된다.

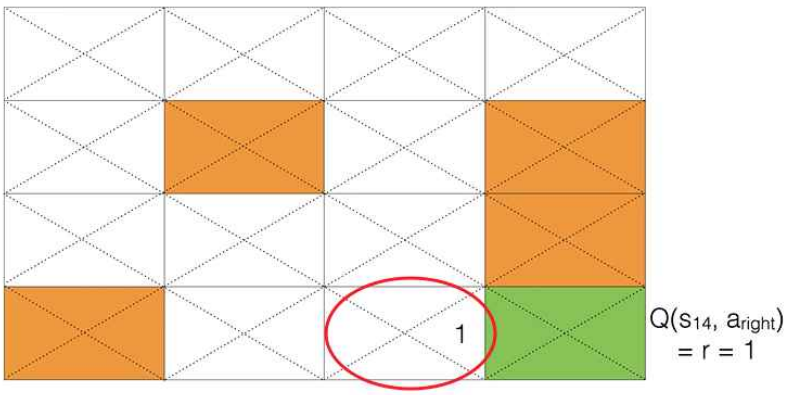
비슷하게, 행동 가치 함수 Q(s, a)에 대한 벨만 방정식도 존재한다. 이 경우는 특정 상태에서 특정 행동을 했을 때 얻는 보상의 기대값을 나타내며, 식은 다음과 같다: $Q(s, a) = E[r + \gamma \times Q(s', a')]$

이때 a'은 다음 상태에서 선택될 다음 행동이고, 정책 π에 따라 선택된다. 이 식 역시 현재 상태-행동 쌍의 가치를 "당장의 보상 + 다음 상태에서의 기대 보상"으로 표현한 것이다. 이 구조를 바탕으로 Q-learning, DQN 등의 알고리즘은 Q값을 반복적으로 업데이트하며 최적의 정책을 학습하게 된다.

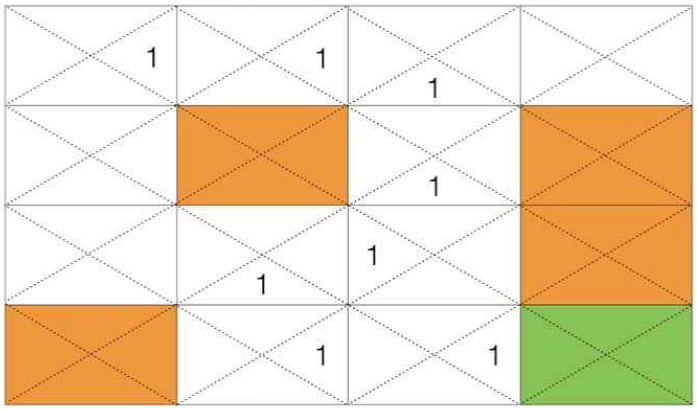
결국 벨만 방정식은 강화학습이 작동하는 기본 원리를 표현한 수학적 틀이며, 에이전트가 현재의 선택이 미래에 어떤 영향을 주는지를 수치적으로 계산할 수 있도록 해주는 가장 중요한 공식이라 할 수 있다.

- Q-table

아래 그림을 통해 Q-table이 생성되는 과정을 설명하겠다.



위 그림에서 초록색 (4, 4)를 목적지라고 가정하면 빨간색 동그라미 칸에서 오른쪽으로 가는 것(action)을 1의 보상을 기대할 수 있다고 할 수 있다.



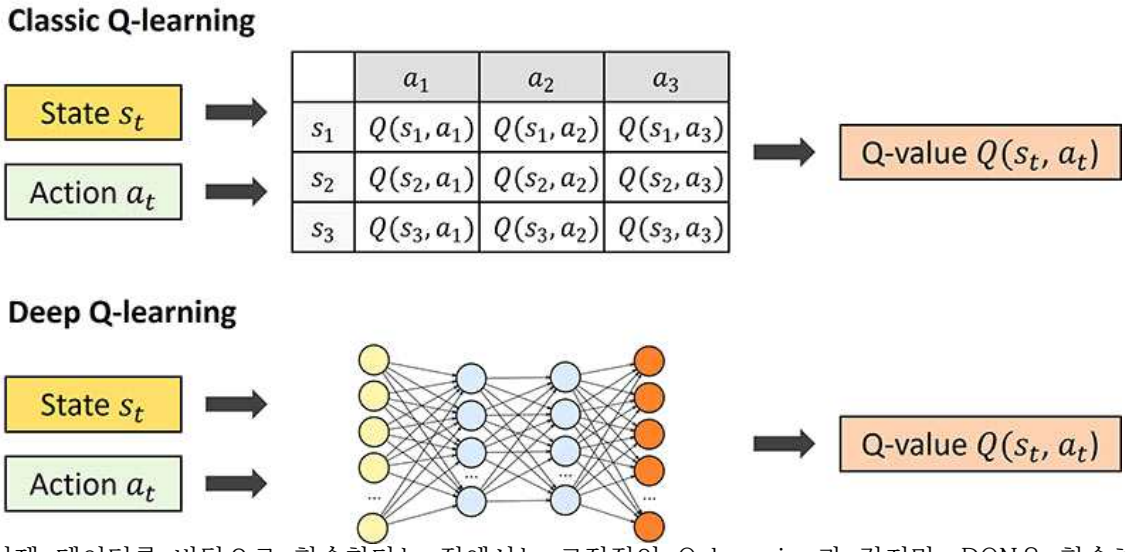
이 기대값들은 모두 Q-value function을 기반으로 계산되며, 반복적으로 업데이트되면서 최적의 정책을 찾아간다. 최종적으론 벨만 방정식을 통해 Q-table값들이 점진적으로 업데이트된다.

상태(State)	행동 1	행동 2	행동 3	행동 4
S1	0.5	0.2	-0.1	0.0
S2	0.0	0.8	0.3	-0.5
S3	-0.3	0.7	0.5	0.2

위 그림은 q-table을 표현한다.

2. deep q-learning: Deep Q-learning은 Q-learning에 딥러닝을 결합한 형태로, Q-table 대신 신경망을 이용하여 Q값을 근사한다. 이때 (state, action, reward, next_state, done)의 transition 데이터를 입력으로 사용하여, 주어진 상태에서 행동을 했을 때의 Q값을 예측한다. 신경망의 파라미터는 일반적으로 θ로 표현된다.

위 그림은 Classic Q-Learning과 Deep Q-Network(DQN)를 비교한 것이다. DQN은 Q-table을 신경망으로 대체하여, 각 상태에서 가능한 행동을 했을 때의 기대 보상(Q-value)을 예측한다. 즉, Q-table의 값을 일일이 계산해 저장하는 방식이 아니라, 신경망을 통해 Q 함수를 근사적으로 학습한다.



실제 데이터를 바탕으로 학습한다는 점에서는 고전적인 Q-learning과 같지만, DQN은 학습하지 않은 상태(state)에 대해서도 일반화된 판단을 할 수 있다는 장점이 있다. 따라서 모든 상태-행동을 일일이 경험하지 않아도 되기 때문에 훨씬 효율적인 학습이 가능하며, 이는 모델을 최적화하는 것이 목표인 이번 프로젝트와 잘 어울린다.

DQN(Deep Q-Network)은 2015년 구글 딥마인드가 Nature에 발표한 논문에서 처음 제안된 알고리즘으로, 기존 Q-learning을 딥러닝과 결합한 방식이다. DQN의 핵심 요소로는 ① CNN 아키텍처, ② Experience Replay, ③ Target Network 세 가지가 꼽힌다. 이번 프로젝트는 CNN아키텍처를 사용하지 않았기에 Experience Replay를 가능하게 해주는 replay buffer과정과 target network에 대해 설명하겠다.

-Replay Buffer

Replay Buffer는 에이전트가 환경과 상호작용하면서 얻은 transition 데이터를 저장하는 공간으로, (state, action, reward, next_state, done) 형태로 저장된다. 이후 이 데이터를 무작위로 샘플링하여 학습에 사용하는데, 이때 추출된 묶음을 mini-batch라고 부른다.

1. 시간에 따른 데이터 의존성 감소 (Decorrelation):

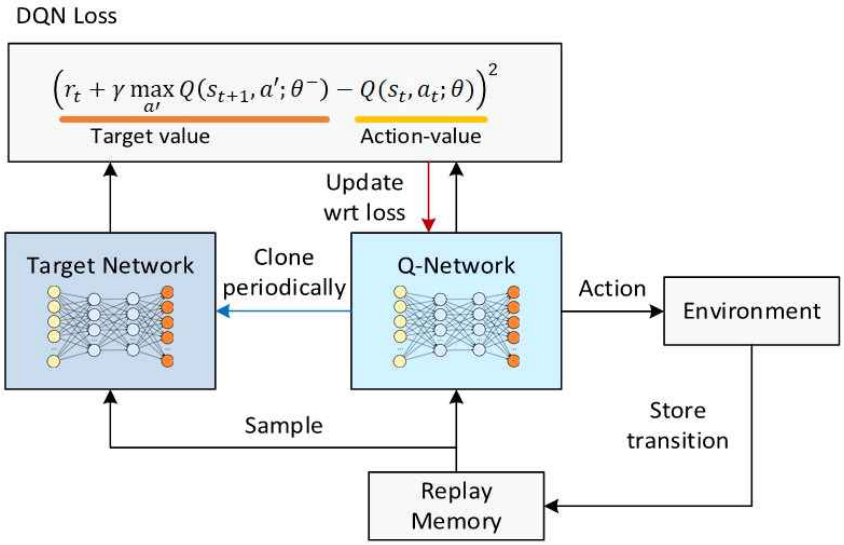
강화학습 데이터는 시간 순서대로 수집되기 때문에, transition 간에 강한 의존성이 존재한다. 그러나 대부분의 학습 알고리즘은 데이터를 서로 독립적으로 샘플링했다는 전제를 깔고 있다. 따라서 Replay Buffer에서 무작위로 데이터를 추출함으로써, 순차적인 데이터 간 의존성을 줄이고 안정적인 학습이 가능하게 만든다.

2. 급격한 데이터 분포 변화 완화:

환경에서 수집되는 데이터는 상황에 따라 갑자기 분포가 변할 수 있다. 이 경우 모델이 그 변화에 과도하게 반응하여 학습이 불안정해질 수 있다. Replay Buffer는 과거의 다양한 데이터 분포를 섞어 사용함으로써, 모델이 보다 일반화된 방식으로 학습하게 도와준다. 특히 초기에는 다양한 랜덤 데이터를 학습함으로써, 이후의 변화에도 쉽게 흔들리지 않는 모델을 만들 수 있다.

-target network

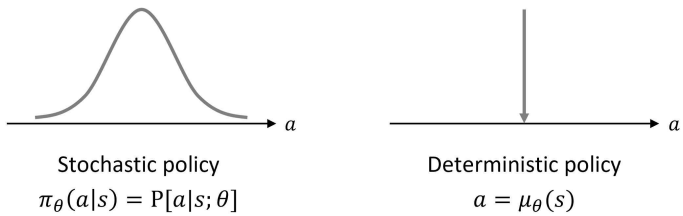
DQN에서는 학습 중인 Q-network와는 별도로 target network를 두어, 일정 주기마다만 파라미터를 복사해 업데이트한다. 이렇게 하면 예측 대상인 Q값이 너무 자주 바뀌는 것을 막아 학습의 안정성을 높일 수 있다.



위 그림은 target network와 DQN, replay buffer(replay memory)가 환경과 상호작용하는 과정을 도식화한 것이다.

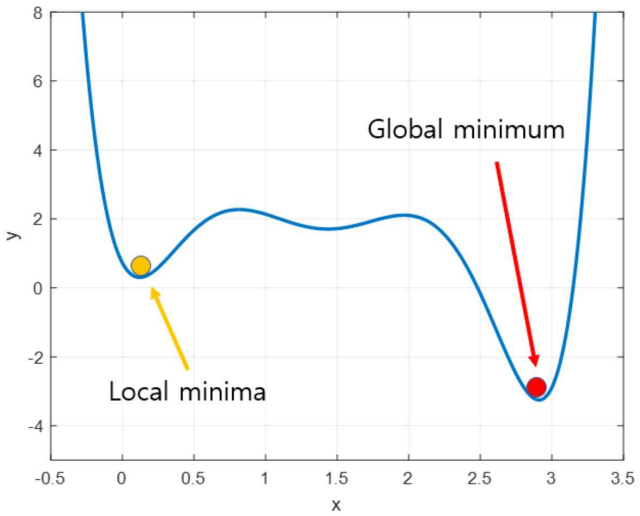
F. DPG (Deterministic Policy Gradient)

지금부터 기본적인 알고리즘에서 벗어나 점점 더 파생적이고 깊은 강화학습 알고리즘을 다룰 것이다.



위 그림은 확률적 정책(Stochastic policy)와 결정론적 정책(Deterministic policy)를 나타낸 그림이다. 위에서 정책을 설명할 때 잠깐 언급했던거다. 확률적 정책은 하나의 상태에서 에이전트가 수행해도 괜찮은 행동들을 확률로써 여러게 출력하는 반면 결정론적 정책은 반드시 하나에 상태엔 하나의 행동만이 대응되는 정책이다. DPG는 이름 그대로 결정론적 정책의 gradient를 계산해 경사하강법을 수행하는 방식이다.

이제부터 수학이라서 조금 어려운데 천천히 이해해보자.



경사 하강법은 경사가 작아지는쪽으로 갈수록 손실함수는 작아지므로 손실함수의 최솟값을 손실함수가 모든 값이 어떤 값에 대응되는지 몰라도 알 수 있다는 아이디어에서 출발한 방법론이다. 물론 기울기가 0 이라고해서 그 점이 극소(local minima)인지 최소(global minimum)인지 알 수는 없다. 이를 해결하기 위해 adam등 여러 변형된 Optimizer들이 사용되지만 일단 지금은 별로 중요치 않다.

정책함수를 $\mu_\theta(s)$라 하자. 그러면 이 정책함수는 θ를 업데이트 해가면서 모든 상태에서 Q값의 평균이 가장 커지는 쪽으로 업데이트 되어야 한다. $J(\mu_\theta(s))$를 목적함수 즉 정책함수의 결괏값 action에 대해 얼마만큼의 보상이 들어왔는지 계산하는 함수를 다음과 같이 정의하면 DPG식은 다음과 같아진다.

$\frac{dJ}{d\theta} = \frac{dJ}{da}\frac{da}{d\theta}$ 이는 연쇄 법칙을 이용하여 계산할 수 있다.

$\nabla J_\theta(\mu(s)) = E_{s\ldots D}[\nabla_a Q(s,a)|_{a=\mu_\theta(s)} \times \nabla_\theta \mu_\theta(s)]$ 따라서 다음과 같은 식으로 정책함수가 어느방향으로 업데이트 되어야 최선의 결과를 낼 수 있을지 알아내기 위한 기울기를 계산할 수 있다.

DPG를 사용하는 이유는 다음과 같다.

1. 연속적인 action space에 잘 맞음: 예를 들어, 로봇 제어처럼 행동이 이산값이 아니라 연속값일 때 확률적으로 샘플링하기가 까다로운데, DPG는 그냥 연속값 하나를 직접 출력하면 되니까 훨씬 간편하다.
2. 효율적임: stochastic policy보다 variance가 낮아서 학습이 더 빠르고 안정적이다.

가령, 로봇 팔이 어떤 목표 지점을 향해 움직이려면, 각 관절의 회전 각도를 정해야 하는데, 이걸 확률로 뽑으면 로봇이 갈팡질팡할 수 있다. 그러나 deterministic policy는 상태 s(현재 위치)에서 각도를 딱 정해서 행동하니까 좀 더 명확하게 학습시킬 수 있다.

이러한 이유로 기반이 되는 알고리즘이 많다:

DDPG (Deep Deterministic Policy Gradient): DPG를 딥러닝으로 확장

TD3: DDPG를 개선한 버전 (overestimation 문제 해결)

SAC: 기본은 stochastic이지만, deterministic 버전도 있음

G. DDPG (Deep Deterministic Policy Gradient)

DDPG는 구글 딥마인드에서 2016년도 ICLR에 발표한 논문에서 등장하였다. 이전 DPG (Deterministic Policy Gradient) 논문에서 DQN을 결합하여 발전시킨 알고리즘이다.

H. PPO(Proximal Policy Optimization)

PPO(Proximal Policy Optimization)는 에이전트가 환경에서 더 좋은 행동을 선택하도록 학습시키는 강화학습 알고리즘이다. 에이전트는 현재 상태를 보고 어떤 행동을 할지 결정하고, 그 행동에 따라 보상을 받는다. PPO는 이 보상을 바탕으로 정책(policy)을 업데이트하는데, 너무 크게 바꾸지 않고 조금씩 조심스럽게 바꿔가면서 안정적으로 학습하는 게 특징이다. 이전 정책과 새 정책 사이의 차이를 계산해서, 변화가 너무 크면 억제하고 적당하면 반영한다. 덕분에 학습이 불안정하게 튀지 않고 꾸준하게 똑똑해질 수 있다. 그래서 PPO는 로봇 제어나 게임 AI처럼 실제 적용이 중요한 상황에서도 자주 쓰이는 알고리즘이다. 쉽게 말하면, PPO는 "조금씩 천천히, 하지만 확실하게 더 나은 행동을 배우는 방식"이다.

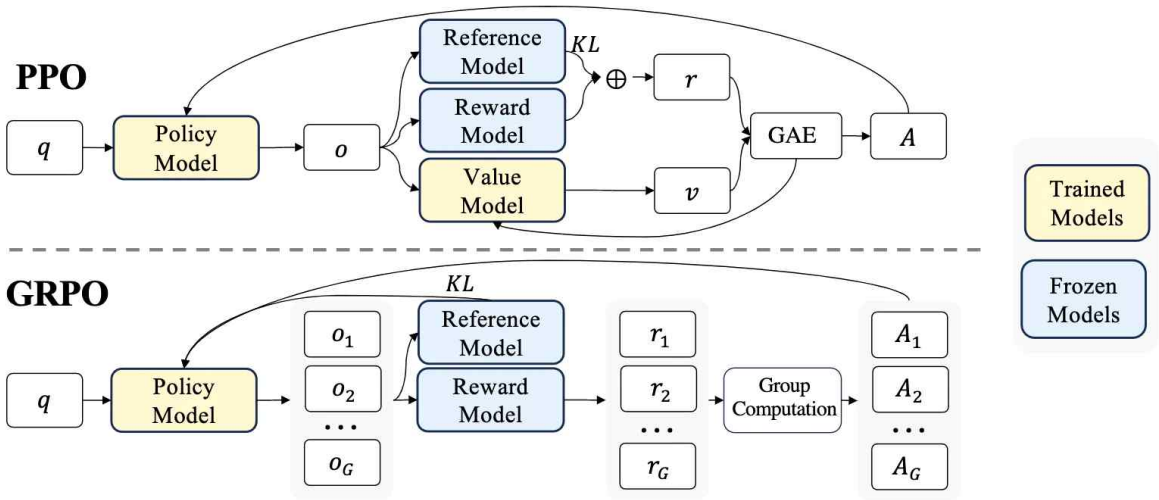


Figure 4 | Demonstration of PPO and our GRPO. GRPO foregoes the value model, instead estimating the baseline from group scores, significantly reducing training resources.

위 그림은 PPO모델의 대략적인 구조를 나타낸다.

근데 우린 이제 고3이니까 PPO의 수학적 개념을 알아야 한다. PPO는 기본적으로 Gradient decent에 기반을 둔다. 즉 "보상을 많이 받는쪽으로 천천히 움직인다"는 말이다. 수식으로는 다음과 같이 표현한다. $L(\theta) = E_t[\log \pi_\theta(a_t|s_t) \times \hat{A}_t]$ 이때 A_t는 해당 행동이 얼마나 좋은지에 대한 가치 이다. q-value와 다른 점은 다른 행동들과 비교했을 때의 상대적 가치이다. L은 손실함수를 의미하므로 이 값이 작을수록 좋다.

$\pi_\theta(a_t|s_t)$는 행동의 확률 즉 1보다 작은 실수이다. 로그함수의 성질에 의해 로그함수값은 항상 음수이고 Adventage는 행동이 좋으면 좋을수록 큰 양수이므로 L값이 최대한 작아지려면 좋은 행동일수록 해당 행동의 확률이 1에 가까워져야 곱했을 때 그나마 L값이 작아질 수 있다. 결과적으로 Loss함수를 최적화 한다는 것은 Adventage가 높으면 높을수록 해당 행동의 확률이 커지도록하는 것 즉 우리의 최종 목표가 되는 것이다.

그런데 PPO는 정책이 갑자기 바뀌는 것을 방지하는 것이 핵심이다. 이때 이전 정책과 업데이트된 정책간의 변화된 비율을 계산하여 비율이 너무 크면 정책이 업데이트 되는 정도를 줄여 정책의 과도한 변화를 지양한다.

$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$ 비율은 다음과 같이 계산할 수 있다.

I. Actor-critic 구조

사실 강화학습 모델의 종류는 크게 정책 기반 모델과 가치 기반 모델로 나눌 수 있다. 가치 기반 알고리즘이 위에서 설명한 DQN이고 PPO는 정책 기반 알고리즘이다. 강화학습 알고리즘 중에서 Actor-Critic 구조는 정책 기반(policy-based)과 가치 기반(value-based) 방법의 장점을 결합한 형태로 널리 쓰이고 있는 방식이다. 일반적인 가치 기반 알고리즘, 예를 들어 Q-learning이나 DQN은 상태(state)와 행동(action)의 쌍에 대해 기대 보상(Q-value)을 예측하고, 그중 가장 높은 값을 가지는 행동을 선택하는 방식이다. 이 방식은 명시적으로 정책(policy)을 정의하지 않고, Q함수 자체를 통해 간접적으로 행동을 선택한다. 반면 정책 기반 알고리즘은 에이전트가 직접 정책을 학습하여, 어떤 상태에서 어떤 행동을 할지 확률적으로 또는 결정적으로 선택한다.

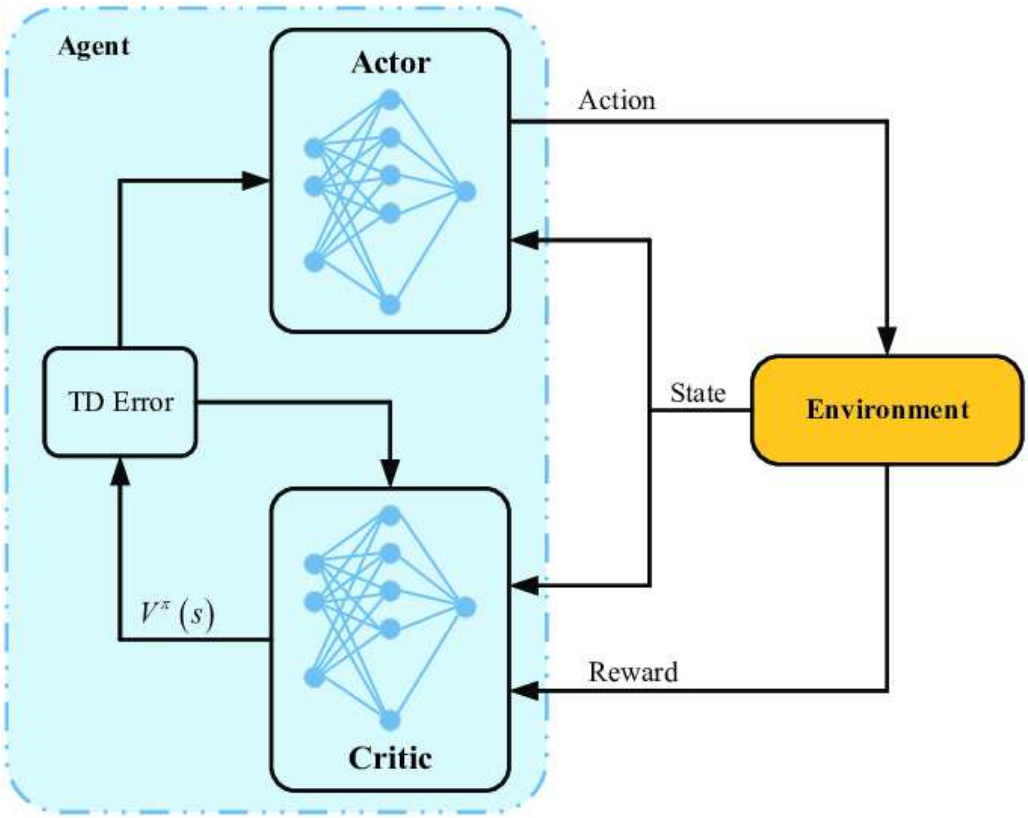
Actor-Critic 구조는 이 두 가지 접근법을 통합한다. 이름 그대로, 이 구조는 두 개의 주요 네트워크로 구성된다. Actor와 Critic이다.

Actor는 현재 상태를 받아서 어떤 행동을 할지를 결정하는 정책 함수 역할을 한다. 말하자면, Actor는 실제 '행동을 선택하는 주체'이다. 이 출력은 확률분포일 수도 있고, 연속적인 행동값일 수도 있다.

Critic은 Actor가 선택한 행동이 얼마나 잘했는지를 평가하는 가치 함수 V(s) 또는 Q(s, a) 를 추정한다. 즉, Critic은 '판사'처럼 Actor의 판단을 평가하는 역할이다.

이 구조가 효과적인 이유는, 정책 기반 학습의 단점인 높은 분산(variance) 문제를 Critic이 보완해주기 때문이다. 정책만 학습하는 경우, 단일한 시도만으로 보상에 대한 정보를 얻기 때문에 학습이 매우 불안정할 수 있다. 하지만 Critic이 상태의 가치를 추정해주면, 이 값을 기준으로 Actor는 상대적으로 안정적으로 학습할 수 있게 된다. 이때 사용되는 것이 Advantage 함수로, 이는 특정 행동이 평균적인 행동보다 얼마나 더 나았는지를 나타내는 지표다. Actor는 이 값을 크게 만들어주는 방향으로 학습된다.

학습 과정은 다음과 같이 진행된다. 에이전트는 환경에서 일정 시간 동안 상태와 행동, 보상을 수집한다. 이후 Critic은 이 데이터를 바탕으로 가치 함수를 업데이트하고, Actor는 Critic이 평가한 Advantage 값을 이용해 정책을 업데이트한다. 이때 Critic의 예측이 너무 자주 바뀌면 Actor가 불안정해질 수 있기 때문에, 일부 알고리즘에서는 Critic의 출력을 안정화시키기 위해 Target Network나 GAE(Generalized Advantage Estimation) 같은 기법을 함께 사용하기도 한다.



actor-critic 구조도

Actor-Critic 구조는 기본적인 A2C(Advantage Actor-Critic)부터 시작해서, PPO(Proximal Policy Optimization), DDPG(Deep Deterministic Policy Gradient), SAC(Soft Actor-Critic), TD3(Twin Delayed DDPG) 등 다양한 현대 강화학습 알고리즘의 기본 뼈대가 된다. 특히 연속적인 행동 공간을 다루는 문제에서는 DQN처럼 행동을 일일이 나열할 수 없기 때문에, Actor-Critic 구조는 거의 필수적으로 사용된다.

요약하자면, Actor-Critic 구조는 행동을 선택하는 네트워크(Actor)와 그 행동의 가치를 평가하는 네트워크(Critic)가 서로 협력하면서 동시에 학습해 나가는 방식이다. 이 구조는 정책 기반 강화학습의 안정성과 효율성을 크게 향상시켜 주며, 실제 로봇 제어나 게임 플레이, 자율주행 같은 고차원 문제에서 널리 활용되고 있다.

Ⅱ-Ⅲ. 모델 경량화

모델 경량화 방법엔 여러 가지 방법이 있다. 그중 이번 프로젝트에선 양자화(Quantization), 프루닝(Pruning) 두가지 방법과 tensorflow에서 지원하는 포맷인 .tflite를 사용하였다.

1. 양자화(Quantization)

양자화란 float(32bit)로 된 파라미터들을 int(8bit)로 변환해 용량을 4배 줄이고 연산을 더 빠르게 하도록 하는 방법이다. 당연히 실수일때와 비교해 정확도는 떨어지지만 많이 떨어져봐야 1~2%라서 모델 경량화가

꼭 필요할 땐 거의 필수적이다.

$q = round(\frac{r-z}{s}) \cdots q : int(8bit), r : float(32bit), z : zero\,point, s : scale$

식은 다음과 같고 z는 zero_point이다. 즉 실수 0을 int의 어느지점에 위치시킬 것인지가 결정된다. int는 -128~127까지 표현 가능하고 float32는 훨씬 많은 범위의 수를 표현하므로 zero point를 잘 결정해야 한다. scale또한 이런 맥락에서 정해진다. 예를들어 실수의 범위가 [0, 10]이라면 -128~127까지의 범위를 나타내는 int8자료형의 scale은 $\frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}$의 식에 따라 (10-0)/(127-(-127)) = 0.0392정도로 정해지게 된다.

2. 프루닝(Pruning)

프루닝이란 별로 중요하지 않은 파라미터들(예를들어 0.000001과같이 0에 근사하는 파라미터)을 0으로 바꾸어 학습하는 방법이다. 이는 모델의 학습중에 설정해야하며 모델의 성능을 크게 낮추지 않고도 연산량을 쉽게 줄일 수 있는 방법이다. 이를 "모델을 희소하게 한다"라고 하며 영어로 sparsity(희소성)으로 나타낸다. 예를들어 [0.2, -0.3, 1, 0.5, 0.1]의 벡터가 있다고 하자. 이를 프루닝 하게되면 [0, 0, 1, 0, 0]의 벡터로 변환되어 데이터의 용량을 줄이고 불필요한 복잡한 계산을 줄일 수 있다.

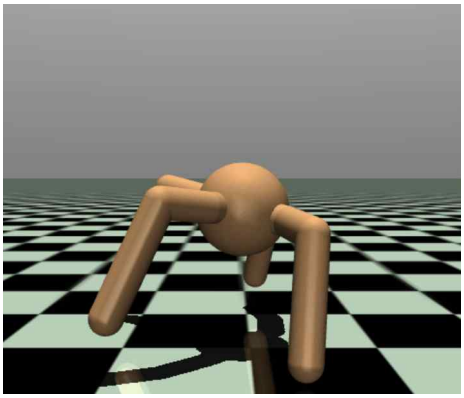
3. Tensorflow Lite

TensorFlow Lite(TFLite)는 TensorFlow로 학습한 모델을 모바일이나 라즈베리파이 같은 저사양 장치에서 빠르고 효율적으로 실행할 수 있도록 변환해주는 도구다. 기존 모델은 float 연산, 복잡한 그래프 구조, 큰 모델 크기 등을 포함하기 때문에 작은 장치에서 실행하기 어렵다. TFLite는 이 문제를 해결하기 위해 다음과 같은 특징을 갖는다.

1. 모델을 .tflite 확장자로 변환해 저장
2. 양자화(Quantization), 프루닝(Pruning) 등의 기법으로 모델을 경량화
3. 계산 그래프를 최적화해 빠르게 실행
4. FlatBuffer 포맷으로 저장하여 저용량 + 빠른 접근 가능
5. TensorFlow보다 훨씬 적은 리소스로 추론 가능 (라즈베리파이에서 유리함)

Ⅲ. 실험 수행

Python환경에서 Openai가 무료로 배포하는 gymnasium이라는 환경을 이용해 Ant-v5라는 가상 4족보행 로봇을 제어하는 강화학습 모델을 만들어서 이번 프로젝트를 수행하였다. 모델은 actor-critic구조의 기본 강화학습 모델에 신경망을 추가한 모델 하나와 PPO모델 하나를 구현했다. 결과적으론 PPO모델이 제일 성능이 좋았고 경량화는 성공적으로 수행되었다.



이렇게 생겼다.

gymnasium의 Ant-v5 환경은 3차원 평면 위에서 네 개의 다리를 가진 개미 형태의 로봇이 이동하는 강화학습 시뮬레이션이다. 이 로봇은 각 관절을 조작하여 가능한 한 멀리 앞으로 나아가는 것이 목표이다. 상태 공간은 로봇의 위치, 속도, 관절 각도 등 다양한 물리 정보를 포함하고, 행동은 각 관절에 가해지는 연속적인 토크 값으로 구성된다. 주로 PPO, SAC 같은 정책 기반 강화학습 알고리즘의 학습에 활용된다. 로봇이 균형을 유지하며 효율적으로 이동하는 전략을 스스로 학습하는 것이 이 환경의 핵심이다.

코드를 다 설명할 생각은 없었지만 고생해서 DQN모델의 파생형인 action-critic모델과 DQN모델의 발전형태인 PPO모델을 코드로 구현했는데 아무것도 설명을 안하기엔 너무 아쉬워서 그나마 쉬운 DQN모델만 설명을 해보도록 하겠다.

```
import numpy as np

class ReplayBuffer:
    def __init__(self, max_size, obs_dim, act_dim):
        self.max_size = max_size
        self.ptr = 0
        self.size = 0

        self.obs_buf = np.zeros((max_size, obs_dim), dtype=np.float32)
        self.next_obs_buf = np.zeros((max_size, obs_dim), dtype=np.float32)
        self.act_buf = np.zeros((max_size, act_dim), dtype=np.float32)
        self.rew_buf = np.zeros((max_size, 1), np.float32)
        self.done_buf = np.zeros((max_size, 1), np.float32)
```

```
    def store(self, obs, act, rew, next_obs, done):
        self.obs_buf[self.ptr] = obs
        self.act_buf[self.ptr] = act
        self.rew_buf[self.ptr] = rew
        self.next_obs_buf[self.ptr] = next_obs
        self.done_buf[self.ptr] = done

        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size + 1, self.max_size)

    def sample_batch(self, batch_size=32):
        idxs = np.random.randint(0, self.max_size, size=batch_size)

        return dict(obs = self.obs_buf[idxs],
                    act = self.act_buf[idxs],
                    rews = self.rew_buf[idxs],
                    next_obs = self.next_obs_buf[idxs],
                    done = self.done_buf[idxs])
```

-위 코드는 Replay buffer를 만든 코드이다. Replay buffer는 (state, action, reward, next_state, done)형태의 튜플 데이터를 저장해야 하므로 store함수의 인자가 self하나와 저거 5개이다. 그리고 모델이 학습을 수행할 때 batch개만큼의 데이터를 반환해야 하므로 sample_batch라는 함수의 출력값이 (state, action, reward, next_state, done)의 형태이다.

```
import tensorflow as tf                    # 딥러닝 모델 만들 때 유명한 2가지 패키지 중 하나
from tensorflow import keras
from tensorflow.keras import layers

class ActorCritic(keras.Model):
    def __init__(self, obs_dim, act_dim):
        super().__init__()

        self.actor = keras.Sequential([
            layers.Dense(256, activation='relu'),
            layers.LayerNormalization(),
            layers.Dense(256, activation='relu'),
            layers.Dense(act_dim, activation='tanh')
        ])

        self.critic = keras.Sequential([
            layers.Dense(256, activation='relu'),
            layers.LayerNormalization(),
            layers.Dense(256, activation='relu'),
            layers.Dense(1)
```

```
        ])

    def call(self, inputs):
        obs = inputs
        return self.actor(obs), self.critic(obs)
```

위 코드는 actor-critic구조에서 actor network와 critic network를 hidden layer를 2층을 쌓아 만든 코드이다.

```
class Agent:
    def __init__(self, obs_dim, act_dim):
        self.model = ActorCritic(obs_dim, act_dim)
        self.target_model = ActorCritic(obs_dim, act_dim)
        self.target_model.set_weights(self.model.get_weights())

        self.buffer = ReplayBuffer(max_size=100000, obs_dim=obs_dim, act_dim=act_dim)
        self.gamma = 0.99
        self.batch_size = 64
        self.optimizer = keras.optimizers.Adam(learning_rate=3e-4)

    def act(self, obs):
        obs = tf.convert_to_tensor(obs.reshape(1, -1), dtype=np.float32)
        action, _ = self.model(obs)
        return action.numpy()[0]

    def learn(self):
        if self.batch_size > self.buffer.size:
            return

        batch = self.buffer.sample_batch(self.batch_size)

        obs = tf.convert_to_tensor(batch['obs'], dtype=np.float32)
        act = tf.convert_to_tensor(batch['act'], dtype=np.float32)
        rews = tf.convert_to_tensor(batch['rews'], dtype=np.float32)
        next_obs = tf.convert_to_tensor(batch['next_obs'], dtype=np.float32)
        done = tf.convert_to_tensor(batch['done'], dtype=np.float32)

        with tf.GradientTape() as tape:
            _, value = self.model(obs)
            _, next_value = self.target_model(next_obs)
            target = rews + self.gamma*(1-done)*next_value
            critic_loss = tf.reduce_mean((value - target)**2)

        gradient = tape.gradient(critic_loss, self.model.trainable_variables)
```

```
        self.optimizer.apply_gradients(zip(gradient, self.model.trainable_variables))

        tau = 0.005
        new_weights = []
        target_variables = self.target_model.weights
        for i, variable in enumerate(self.model.weights):
            new_weights.append(tau * variable + (1 - tau) * target_variables[i])
        self.target_model.set_weights(new_weights)
```

위 코드는 actor-critic구조에 Agent를 합치고 agent가 replay buffer를 활용하여 학습하는 코드를 작성한 것이다.

```
import gymnasium as gym

rew_step = []

env = gym.make('Ant-v5', render_mode='human')
act_dim = env.action_space.shape[0]
obs_dim = env.observation_space.shape[0]

agent = Agent(obs_dim, act_dim)

num_episodes = 5

for ep in range(num_episodes):

    obs, info = env.reset()
    total_reward = 0
    waiting_time = 0

    for t in range(1000): # 5000frame --> step*5 = frame
        action = agent.act(obs)
        noise_scale = max(0.1, 1.0 - ep / num_episodes)  # 점점 줄이기
        noise = np.random.normal(0, noise_scale, size=action.shape)
        action += noise

        action = np.clip(action, -1, 1)
        next_obs, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated

        shaped_reward = reward

        foward_progress = abs(next_obs[0] - obs[0])
        shaped_reward += foward_progress*5
```

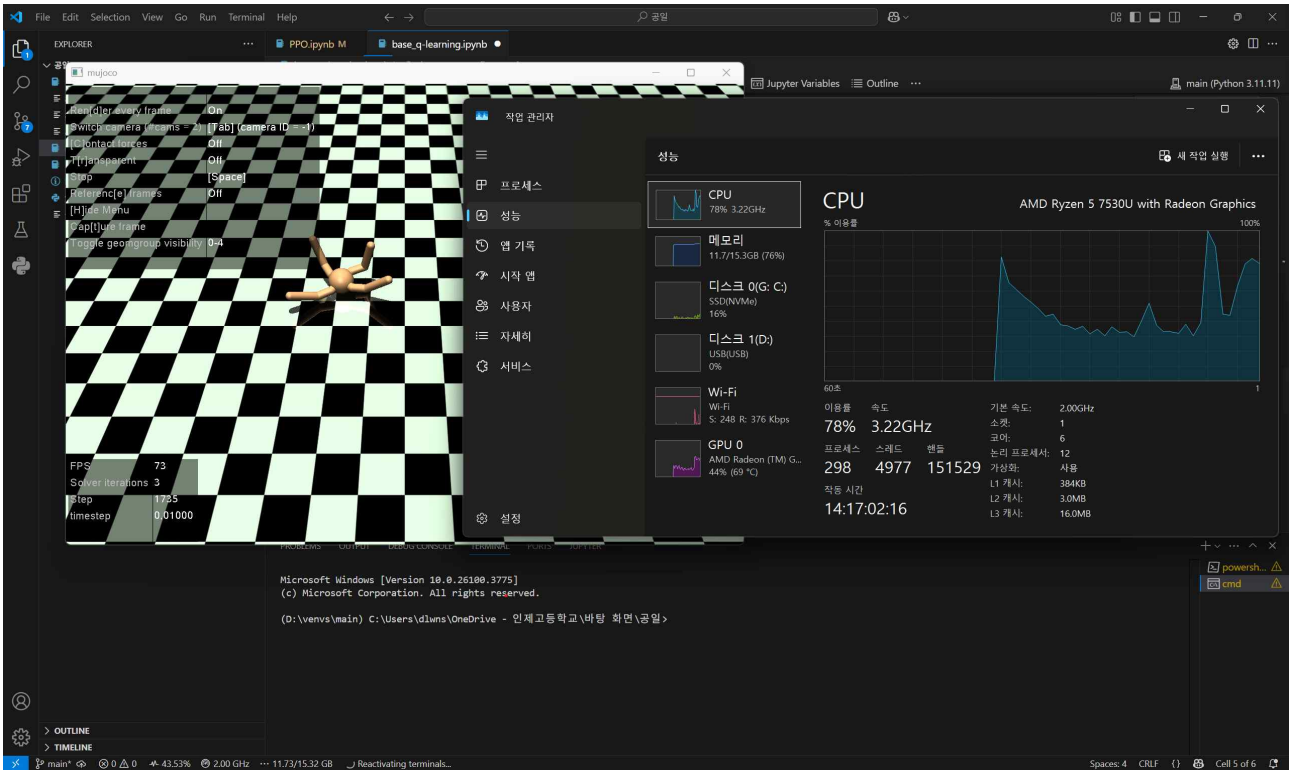
```
            pre_z_posision = obs[2]
            z_position = next_obs[2]
            waiting_penalty = 0
            if abs(pre_z_posision-z_position) + abs(next_obs[0]-obs[0]) < 0.05:
                waiting_time += 1
            else:
                waiting_time = 0

            if waiting_time > 20:
                waiting_time = 0
                waiting_penalty = 20

            shaped_reward -= waiting_penalty
            waiting_penalty = 0

            if z_position < 0.3:
                penalty = 1
            else:
                penalty = 0
            shaped_reward -= penalty

            agent.buffer.store(obs, action, shaped_reward, next_obs, done)
            agent.learn()

            obs = next_obs
            total_reward += shaped_reward

            if done:
                break

        # Actor만 따로 저장
        actor_model = agent.model.actor
        actor_model.save("actor_model.keras")


        print(f"episode: {ep}, total reward: {total_reward}")
        rew_step.append(total_reward)

input()
env.close()
```

위 코드는 보상 체계와 환경을 구성한 코드이다. 보상 함수는 다음과 같이 설정했다.

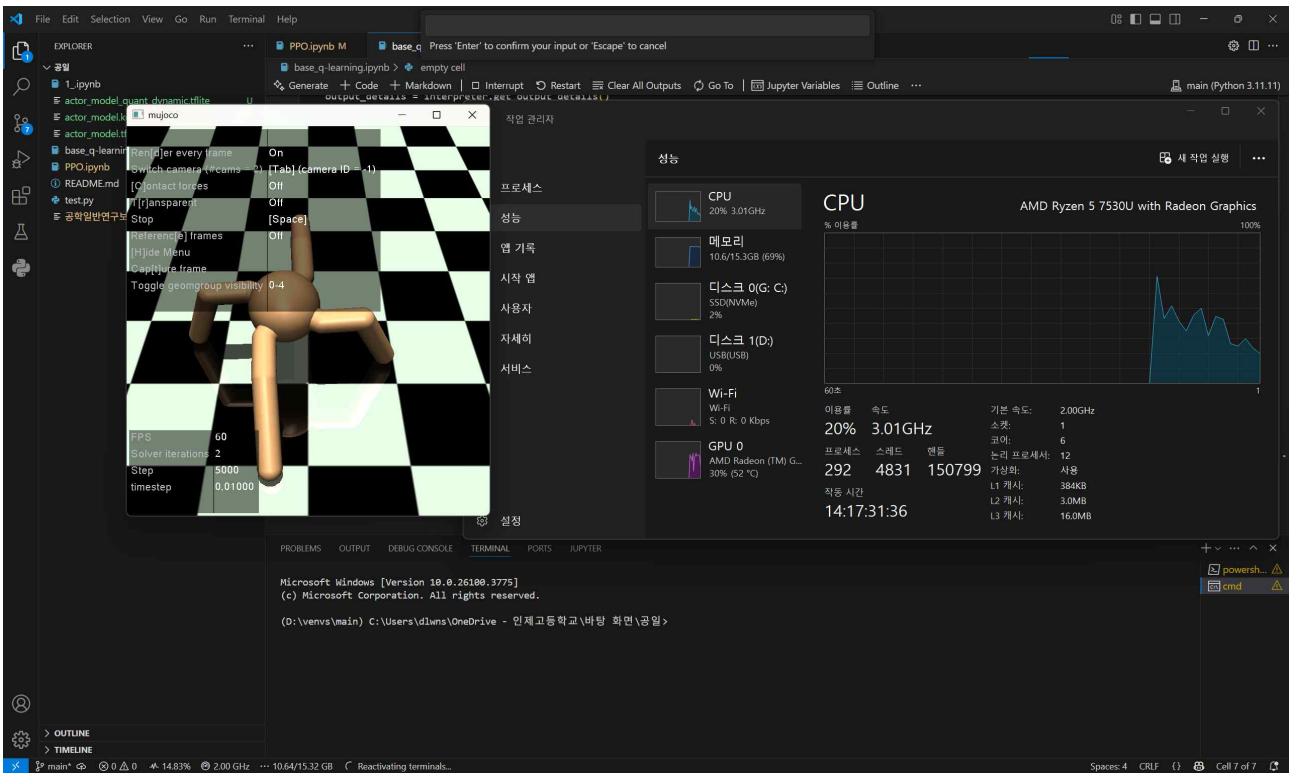
Reward

forward_reward: 로봇이 앞으로 나아간 거리에 비례하는 보상이다.
ctrl_cost: 동작 제어 신호를 크게 쓰면 패널티(벌점)가 주어진다. 로봇이 관절을 움직일 때 너무 큰 힘을 쓰지 않도록 유도하는 역할을 한다.
contact_cost: 외부와 접촉했을 때 발생하는 힘에 비례해서 추가적인 패널티가 주어진다. 특히 땅이나 다른 물체와 충돌했을 때 너무 세게 부딪히지 않도록 제어하는 데 도움을 준다.
survive_reward: 매 타임스텝(time step)마다 살아있으면 고정된 양의 보상이 주어진다. 즉, 로봇이 쓰러지거나 완전히 멈추지 않고 계속 움직이고 있으면 이 보상을 계속 받을 수 있다.
종합하면, 전체 reward는 다음과 같은 요소들을 합쳐서 계산된다:
앞으로 나아간 것에 대한 보상
제어 신호를 과하게 사용한 것에 대한 패널티
접촉에 따른 패널티
생존에 대한 보상 (가만히 서있으려고 하기도함)

Ⅳ. 탐구 결과



처음 모델을 학습시킬때는 cpu사용률이 78%까지 찍힐정도고 렌더링도 느려서 화면이 뚝뚝 끊기는 현상이 있었다.



확실히 모델 경량화를 해보니 평균 메모리 사용률도 1GB 이상 줄어들었고 cpu사용률이 평균적으로 감소

했으며 렌더링 속도가 빨라져 사람이 봤을 때 뚝뚝 끊기는 모습을 볼 순 없었다.

영상이 한글파일엔 삽입이 안돼서 링크를 걸어두었다. 링크에 들어가면 actor-critic모델과 PPO모델의 경량화 전, 후 실행 비교 영상이 있다. 아래 링크를 참고하면 결과를 쉽게 볼 수 있다.
https://github.com/ak45aaa/sub_Engineering

결과적으로 PPO모델이 확실이 가벼운 모델이라 그런지 속도도 빠르고 모델도 비교적 빠르게 움직였다. actor-critic구조는 복잡한데 성능은 성능대로 안나와서 영상을 보면 알겠지만 경량화된 모델은 아예 잘 움직이질 않는다. 이는 학습량이 부족한걸수도 있지만 모델 구조 자체가 별로 좋지 않은 이유인것같다. 그리고 또한 두 모델 다 경량화를 진행한 이후로 속도가 더 빨라지고 리소스 사용량이 줄어들었음을 볼 수 있었다.

V. 결론 및 느낀점

이번 탐구를 통해 강화학습 모델의 경량화 가능성을 직접 실험하고 검토해보는 경험을 했다. 복잡한 알고리즘 없이도 사람이 인식할 수 있을 정도의 성능을 확보할 수 있었고, 이는 저사양 환경에서도 AI 기술을 적용할 수 있다는 가능성을 시사하였다. 비록 초기에는 간단한 보상 시스템과 강화학습 구조를 사용하여 뚜렷한 성과를 얻지는 못했지만, 수학적 개념과 알고리즘에 대한 학습을 통해 연구의 방향성과 방법을 체계화할 수 있다는 자신감을 얻게 되었다.

강화학습과 인공지능의 기본 개념을 친구들에게 설명하는 과정을 통해 내가 놓치고 있었던 이론적 맥락이나 오개념들을 다시 점검할 수 있었다. 특히, 강화학습을 실습 가능한 형태로 구현하는 과정에서 오픈소스 환경인 Gymnasium을 접하게 되었고, 이를 활용해 모델을 실제로 동작시킬 수 있었다. 미분, 정규분포, 조건부 확률 등 강화학습에 필요한 수학 개념들이 등장했을 때에는 고등학교 교육과정을 바탕으로 교과서를 다시 살펴보고, 부족한 부분은 관련 논문과 온라인 강의를 참고하며 보완하였다.

이러한 과정을 통해 인공지능 기술에 대한 나의 관심이 더욱 깊어졌으며, 앞으로는 모델 경량화와 함께 하드웨어 및 소프트웨어를 통합적으로 설계하고 운용할 수 있는 인공지능 엔지니어로 성장하고 싶다는 진로 목표도 명확해졌다.

참 고 문 헌

임은아 (En-a Lim),김나영 (Na-young Kim),이종락 (Jong-lark Lee),and 원일용 (Ill-yong Weon). "Unity3D 가상 환경에서 강화학습으로 만들어진 모델의 효율적인 실세계 적용." 한국정보처리학회 학술대회논문집 27.2 (2020): 800-803.

정민구. "효과적인 강화학습을 위한 상태 텐서 설계." 국내석사학위논문 아주대학교, 2023. 경기도

2010년 10월 한국인터넷방송통신학회 논문지 제10권 제5호- 265 -

논문 2010-5-38 목표지향적 강화학습 시스템 Goal-Directed Reinforcement Learning System 이창훈* Chang-Hoon Lee

Schulman, John, et al. "Proximal policy optimization algorithms." arXiv preprint arXiv:1707.06347 (2017).

SILVER, David, et al. Deterministic policy gradient algorithms. In: International conference on machine learning. Pmlr, 2014. p. 387-395.