DQN 계열 알고리즘과 보상 설계를 통한 LunarLander 강화학습 성능 향상 보고서

팀: 개인

팀원: 20211014 김한성
github: https://github.com/rlagkstjdtt-cmyk/RL_project

요약 (Abstract)

본 프로젝트는 Gymnasium의 LunarLander-v3 환경에서 DQN(Deep Q-Network) 계열 알고리즘의 성능을 향상시키는 것을 목표로 한다. 기본적인 DQN 구조를 기반으로 Double DQN, Dueling DQN 등 확장 알고리즘을 비교하고, FuelSaving, SafeLanding과 같은 보상 설계(Reward Shaping) 기법의 효과를 검증하였다. 또한, 타깃 업데이트 방식, 리플레이 버퍼 크기, ε-decay 속도 등 핵심 구성요소들의 변화에 따른 성능 영향을 분석하였다. 모든 결과를 종합하여 성능이 우수한 최종 에이전트를 설계하고 그 성능을 평가한다. 학습 안정성과 결과 재현성을 확보하기 위해 Target Network, Replay Buffer와 같은 안정화 기법을 적용하였으며, 실험은 seed=0과 seed=1 두 번의 독립 학습을 통해 진행하였다.

1. 서론 (Introduction)

1.1. 프로젝트 배경 및 목표

LunarLander-v3 환경은 강화학습 알고리즘의 기초 성능 및 안정성을 평가하기에 적합한 대표적인 이산 행동 제어 문제이다. 본 프로젝트는 이 환경에서 DQN 알고리즘의 최적화 가능성을 탐색하고, 안정적인 착륙 정책을 학습하기 위한 효율적인 방법을 모색하는 것을 목표로 한다.

1.2. 주요 실험 내용

LunarLander 환경에서 DQN 기반 에이전트의 성능에 영향을 미치는 주요 요소를 분석하기 위해 다음과 같은 세 가지 축으로 실험을 진행하였다.

1. 알고리즘 구조 비교: DQN / Double DQN / Dueling DQN
2. 보상 설계 비교 **(Reward Shaping)**: Base / FuelSaving / SafeLanding
3. 구성요소 비교 **(Ablation Study)**:
    - 타깃 업데이트 방식 (Hard vs Soft)
    - Replay Buffer 크기 (Small vs Middle vs Big)

- ε-decay 속도 (Fast vs Middle vs Slow)

위 결과를 종합하여 성능이 좋은 최종 에이전트 설계 및 평가하였다.

2. 환경 및 전처리 (Environment and preprocessing)

2.1. 환경 설명 (Gymnasium LunarLander-v3)

다음은 Gymnasium LunarLander-v3 환경에 대한 설명입니다.

- 상태(state)

$$\text{State} = (x, y, v_x, v_y, \theta, \dot{\theta}, L_{contact}, R_{contact})$$

요소	의미
(x, y)	착륙선의 위치 (좌표)
(v_x, v_y)	착륙선의 속도
θ	착륙선의 각도 (기울어짐)
$\dot{\theta}$	착륙선의 각속도
$L_{contact}$	왼쪽 착륙 다리 지면 접촉 여부 (Boolean $\rightarrow$ 0.0 또는 1.0 변환)
$R_{contact}$	오른쪽 착륙 다리 지면 접촉 여부 (Boolean $\rightarrow$ 0.0 또는 1.0 변환)

- 행동(action)

Action Index	행동 내용
0	아무 것도 안 함 (None)
1	왼쪽 엔진 (Side Engine) 점화
2	메인(중앙) 엔진 (Main Engine) 점화
3	오른쪽 엔진 (Side Engine) 점화

- 기본 보상(환경 제공)
    - 착륙 성공: 위치/속도 조건을 만족하면 +100~+200 수준의 보상
    - 추락/화면 이탈: 큰 음의 보상
    - 다리 접촉 시 +10 보상
    - 매 step마다 작은 음의 보상 및 엔진 사용 페널티가 포함
      → 자연스럽게 짧은 시간 안에, 연료를 적게 쓰고, 안정적으로 착륙하는 정책을 유도

2.2. 전처리

- 관측 상태: 환경에서 제공하는 8차원 실수 벡터를 그대로 사용.
- 형 변환: 필요 시 np.float32로 형 변환 후 PyTorch 텐서로 변환.
- 스케일링: 별도의 정규화/표준화는 적용하지 않았으며, 네트워크가 비선형 함수를 통해 직접 스케일을 학습하도록 설계.

3. 실험 알고리즘 및 하이퍼파라미터 (Algorithms and Hyperparameters)

3.1. 기본 DQN

- Q-network 구조
    - 입력 차원: 8 (상태 벡터)
    - 은닉층: Fully-Connected 2층 MLP
        - 예: 128 → 128, 활성함수 ReLU
    - 출력 차원: 4 (각 행동에 대한 Q(s,a) 값)
- 손실 함수
    - 경험 재플레이 버퍼에서 샘플링한 배치에 대해

$$y = r + \gamma \max_{a'} Q_{\theta^-}(s', a')$$

$$L(\theta) = \mathbb{E}[(y - Q_\theta(s, a))^2]$$

- **Replay Buffer**: 과거 transition을 저장해 두고 미니배치로 랜덤 샘플링하여 학습에 사용. 샘플 상관을 줄이고 학습 분포를 안정화하는 역할.
- **Target Network**: 일정 주기마다 타깃 Q 네트워크를 온라인 Q 네트워크로 복사(Hard Update)하여, 타깃 값이 너무 빠르게 변하는 것을 방지하고 학습 안정성을 증가.

3.2. 비교 알고리즘

알고리즘 비교 실험에서는 다음 세 가지를 구현·비교하였다.

1. **DQN_basic**
    - 기본 DQN
2. **DoubleDQN**
    - 타깃 계산을
      $$a^* = \arg\max_a Q_\theta(s', a), \quad y = r + \gamma Q_{\theta^-}(s', a^*)$$
    - 로 변경하여 overestimation 완화
3. **DoubleDQN_Dueling**
    - Double DQN에 Dueling 구조를 결합
    - Q(s,a)를 상태가치 V(s)와 이점함수 A(s,a)로 분해하는 head를 사용

→ 알고리즘 비교 실험 후, 최종 일반화 단계에서는
plain DQN 구조**(use_double_dqn=False, use_dueling=False)** 를 선택하였다.
(실험 결과 및 구현 복잡도 등을 고려한 선택)

3.3. 보상 설계 (Reward Shaping)

보상 설계 실험에서는 환경 기본 보상에 추가로 다음 항목들을 적용해 보았다.

- **BaseReward**: 보상 셰이핑 없음 (환경 기본 보상 그대로 사용)
- **FuelSaving**: 연료 절약형 보상

$$r' = r - \alpha \cdot \text{engine\_usage}$$

-엔진을 사용할 때마다 α>0\alpha > 0α>0만큼 추가 패널티

-연료 사용량을 줄이는 정책 유도

- **SafeLanding**: 안정 착륙형 보상

$$r' = r - \beta_\theta |\theta| - \beta_v \|(v_x, v_y)\|$$

각도 및 속도를 줄이는 방향으로 추가 패널티

3.4. RL 하이퍼파라미터 (초기 공통 BASE 설정 및 선택 이유)

다음은 초기 BASE 설정 값과 선택 이유를 요약한 표입니다.

항목	초기 BASE 설정 값	선택 이유 요약
할인율 (gamma)	0.99	장기적인 착륙 성공 보상을 고려하기 위해 널리 쓰이는 값을 채택. 0.9보다 커야 하며, 0.999보다는 안정적.
학습률 (lr)	1e-3	PyTorch 기반 DQN 구현에서 자주 사용되는 값. 초기 실험에서 발산 없이 안정적인 수렴 확인.
배치 크기	64	Gradient 분산 안정화(32보다 큼)와 메모리/시간 효율(128보다 작음)의 타협점.
Replay Buffer 크기	100,000	다양한 궤적을 충분히 저장할 수 있는 표준적인 크기.

항목	초기 BASE 설정 값	선택 이유 요약
최소 리플레이 크기	5,000	버퍼가 비어있는 상태에서 학습하는 것을 방지하여 안정적인 업데이트 시작점을 확보. 5,000 스텝까지 워밍업 후 학습 시작.
타깃 업데이트 방식	Hard update	구현 단순성을 위해 채택.
타깃 업데이트 주기	1000 스텝마다	안정성과 수렴 속도 측면에서 무난한 값으로 판단.
초기 탐험률 (eps_start)	1.0	초기 단계에서 완전 랜덤 정책으로 다양한 상태를 탐험.
최종 탐험률 (eps_end)	0.05	학습 후반에 소량의 탐험을 남겨두어 local optimum에 갇히는 것을 방지.
eps_decay_steps	60000	탐험과 수렴 사이의 적절한 타협점. 최종 일반화 설정에서 60,000 스텝 사용.
총 에피소드 수	800 (이후 1000으로 조정)	초기 설정은 800이었으나, 수렴 부족으로 후반 실험에서는 1000 에피소드로 늘려서 진행.

1. 네트워크 구조 관련

- 은닉층 크기: `hidden_dim = 128` (2층 MLP)
    - 설정 값: 입력 8차원 → 128 → 128 → 출력 4차원
    - 이유:
        - LunarLander의 상태 차원(8)이 크지 않고, 환경 자체가 너무 복잡한 고차원 문제가 아니다.
        - 64보다 작은 은닉 크기는 표현력이 부족할 수 있고, 256 이상은 파라미터 수가 불필요하게 커져 학습 시간과 불안정성이 증가할 수 있다.

        - 따라서 실무/튜토리얼에서 자주 사용되는 128 차원의 2층 MLP를 선택하여, 표현력과 연산 비용 사이의 균형을 맞추었다.

- 활성함수: ReLU
    - 이유:
        - 구현이 간단하고 계산량이 적으며,
        - Sigmoid/Tanh에 비해 기울기 소실 문제(vanishing gradient)에 덜 민감하다.
        - DQN류 구현에서 사실상 “기본값”으로 사용되는 함수이므로 채택하였다.

---

2. 학습 관련 하이퍼파라미터

- 할인율: `gamma = 0.99`
    - 이유:
        - LunarLander는 한 에피소드 내에서 수십~수백 스텝의 연속적인 행동이 필요하며,
          착륙 성공/실패 보상은 에피소드 끝에 주어지는 경우가 많다.
        - γ가 너무 작으면(예: 0.9) 먼 미래 보상을 거의 고려하지 않아 “장기적인 착륙 성공” 대신 단기 보상에만 집착하는 정책이 학습될 수 있다.
        - 반대로 0.999처럼 지나치게 크면 학습이 느려지고 값이 불안정해질 수 있어,
          DQN에서 널리 쓰이는 0.99를 사용하였다.
- 학습률: `lr = 1e-3`
    - 이유:
        - 너무 큰 학습률(예: 1e-2)은 Q값이 발산하거나 진동할 위험이 크고,
        - 너무 작은 학습률(예: 1e-4 이하)은 수렴 속도가 매우 느려진다.
        - PyTorch 기반 DQN 구현에서 자주 사용되는 1e-3을 기본값으로 두고, 초기 실험에서 발산 없이 안정적으로 수렴하는 것을 확인하여 그대로

사용하였다.

- 배치 크기: `batch_size = 64`
    - 이유:
        - 32보다 작은 배치는 gradient 분산이 커져 학습이 불안정할 수 있고,
        - 128 이상은 메모리/시간 비용이 커진다.
        - LunarLander 수준의 문제에서는 32~64가 자주 쓰이므로, 안정성과 속도 사이의 타협점으로 64를 선택하였다.

---

3. 리플레이 버퍼 및 워밍업 설정

- 리플레이 버퍼 크기: `buffer_size = 100000` (초기 공통 설정)
    - 이유:
        - 너무 작은 버퍼는 과거 경험을 빨리 잃어버려 데이터 다양성이 떨어지고, 최근 경험에 과도하게 과적합될 수 있다.
        - LunarLander는 한 에피소드 길이가 그리 길지 않으므로, 10만 개의 transition이면 다양한 궤적을 충분히 저장할 수 있다.
        - 따라서 메모리 부담이 크지 않은 선에서, 표준적인 DQN 구현에서 많이 쓰이는 100,000을 기본값으로 사용했다.
        - 이후 구성요소 실험에서 50,000 등 중간 크기를 비교해 보고, 최종 일반화 설정에서는 50,000을 채택하였다.
- 최소 리플레이 크기(워밍업): `min_replay_size = 5000`
    - 이유:
        - 버퍼가 너무 비어 있는 상태에서 학습을 시작하면, 샘플이 거의 같은 전이들로만 구성되어 있어 학습이 매우 불안정하다.
        - 경험적으로 LunarLander에서 5,000개 정도의 transition을 먼저 쌓은 뒤 학습을 시작하면,

다양한 상태/행동이 포함되어 안정적인 업데이트가 가능했다.

- 따라서 5,000 스텝까지는 버퍼만 채우고, 그 이후부터 학습을 시작하도록 설정하였다.

---

4. 타깃 네트워크 관련

- 타깃 네트워크 사용: `use_target_net = True`
  - 이유:
    - 타깃 없이 온라인 Q 네트워크만 사용하는 경우, TD 타깃이 매 업데이트마다 크게 변해 학습이 쉽게 발산한다.
    - DQN 논문에서 이미 타깃 네트워크가 필수적인 안정화 요소로 소개되어 있으며, LunarLander 같은 환경에서도 마찬가지다.
    - 따라서 모든 실험에서 타깃 네트워크를 사용하는 것을 기본으로 했다.
- 업데이트 방식: Hard update, `target_update_freq = 1000`, `use_soft_update = False`, `tau = 0.0`
  - 이유:
    - Hard update는 구현이 간단하고, 일정 스텝마다 `target_q_net ← q_net`를 그대로 복사하는 방식이다.
    - 너무 자주 업데이트하면 타깃이 불안정해지고, 너무 드물게 업데이트하면 학습이 느려질 수 있다.
    - LunarLander에서 여러 실험을 해 본 결과, 1000 스텝마다 복사하는 것이 안정성과 수렴 속도 측면에서 무난한 선택이었다.
    - Soft update(τ 사용)는 추가 실험에서 비교만 수행하고, 최종 일반화 설정에서는 구현 단순성을 위해 Hard update를 채택하였다.

---

5. 탐험(Exploration) 관련 하이퍼파라미터

초기 공통 설정(알고리즘/리워드/구성요소 실험에서 사용한 기본값 기준):

- 초기 탐험률: `eps_start = 1.0`
    - 이유:
        - 학습 초반에는 환경에 대한 정보가 전혀 없으므로,
          거의 무작위에 가까운 행동을 통해 다양한 상태를 방문하는 것이 중요하다.
        - ε=1.0이면 완전 랜덤 정책이므로,
          버퍼를 채우는 초기 단계에서 다양한 경험을 수집하기에 적절하다.
- 최종 탐험률: `eps_end ≈ 0.05` (대략 5% 수준)
    - 이유:
        - 학습 후반에는 정책이 어느 정도 괜찮은 수준에 도달했더라도,
          완전히 탐험을 끊어버리면 새로운 상황에 대해 적응하기 어렵다.
        - 0.0까지 떨어뜨리는 대신 **소량의 탐험(5%)**을 남겨두어,
          극단적으로 나쁜 local optimum에 갇히는 것을 방지하고자 했다.
- ε 감소 스텝 수: `eps_decay_steps = 50000` (초기), 이후 60000으로 조정
    - 이유:
        - ε가 너무 빨리 줄어들면 초반에 충분한 탐험 없이 탐욕적 정책으로 고착될 위험이 있다.
        - 반대로 너무 천천히 줄어들면 수렴이 늦어지고 학습 곡선에 잡음이 많아진다.
        - LunarLander에서 여러 실험을 해 보았을 때,
          5만~6만 스텝 정도의 선형 감소가 “탐험 ↔ 수렴” 사이의 적절한 타협점이라고 판단하였다.
        - 최종 일반화 설정에서는 실험 결과를 바탕으로 60,000 스텝을 사용하였다.

4. 실험 방법 및 평가 (Experimental Methods and Evaluation)

4.1. 실험 환경 및 설정

본 실험은 강화 학습 알고리즘의 성능을 평가하기 위해 아래와 같은 환경 및 설정을 사용하였습니다.

항목	내용
하드웨어	Google Colab 환경 (GPU - T4)
소프트웨어	Python 3.x, PyTorch, NumPy, Gymnasium (LunarLander-v3), Matplotlib, tqdm 등
전처리	상태의 정규화/표준화는 미적용. 환경 반환 상태를 `np.float32`/`torch.float32`로 형 변환하는 최소한의 전처리만 수행.

4.2. 평가 지표 (Evaluation Metric)

각 설정(알고리즘 / 보상 설계 / 구성요소 조합)에 대해 다음 절차로 평가를 수행하였다.

1. 해당 설정과 seed(0 또는 1)에 대해 지정된 에피소드 수(예: 1000 에피소드) 동안 학습을 진행한다.
2. 학습이 끝난 후에는 탐험 없이**(epsilon = 0)** 완전히 탐욕적인 정책으로 고정한다.
3. 이 정책으로 일정 횟수(예: 20 에피소드)의 에피소드를 실행하여, 각 에피소드 리턴(총 보상)을 기록한다.
4. 한 번의 학습(run)에 대한 성능 지표로, 해당 run에서 얻은 에피소드 리턴들의 평균을 사용한다.

5. 같은 설정에 대해 **seed = 0, 1** 두 번의 학습을 수행하여

$$R_1,\ R_2$$

두 개의 평가 리턴을 얻는다.

이때 최종적으로 사용하는 평가지표는 다음과 같다.

- **평균 성능**: 두 seed에 대한 리턴의 평균

$$\bar{R} = \frac{R_1 + R_2}{2}$$

- **표본 표준편차**:

$$s = \sqrt{\frac{(R_1 - \bar{R})^2 + (R_2 - \bar{R})^2}{n-1}}, \quad n = 2$$

- **(선택적으로) 성공률**: 일정 기준(예: 리턴 ≥ 200)을 만족한 에피소드 비율

표본 수가 매우 작기 때문에, 정규분포 대신 자유도 1인 t-분포를 사용하여
형식적인 95% 신뢰구간을 다음과 같이 계산하였다.

$$\bar{R} \pm t_{1,0.975} \cdot \frac{s}{\sqrt{n}}, \quad t_{1,0.975} \approx 12.706$$

즉, $n = 2$일 때 95% 신뢰구간은

$$\bar{R} \pm 12.706 \cdot \frac{s}{\sqrt{2}}$$

가 된다.

4.3. 결과의 신뢰도

표본 수가 2개(n=2)**에 불과하기 때문에,

- t-값이 매우 크고
- 신뢰구간 폭도 과도하게 넓어지며
- 통계적으로 “정교한 95% 신뢰구간”이라고 보기 어렵다는 한계가 존재한다.

따라서 본 프로젝트에서는 이 신뢰구간을

“해당 설정에서 seed를 바꿔서 두 번 실험했을 때, 결과가 어느 정도 범위에서 변동할 수 있는지 보여주는 거친 지표”로 해석하였다.

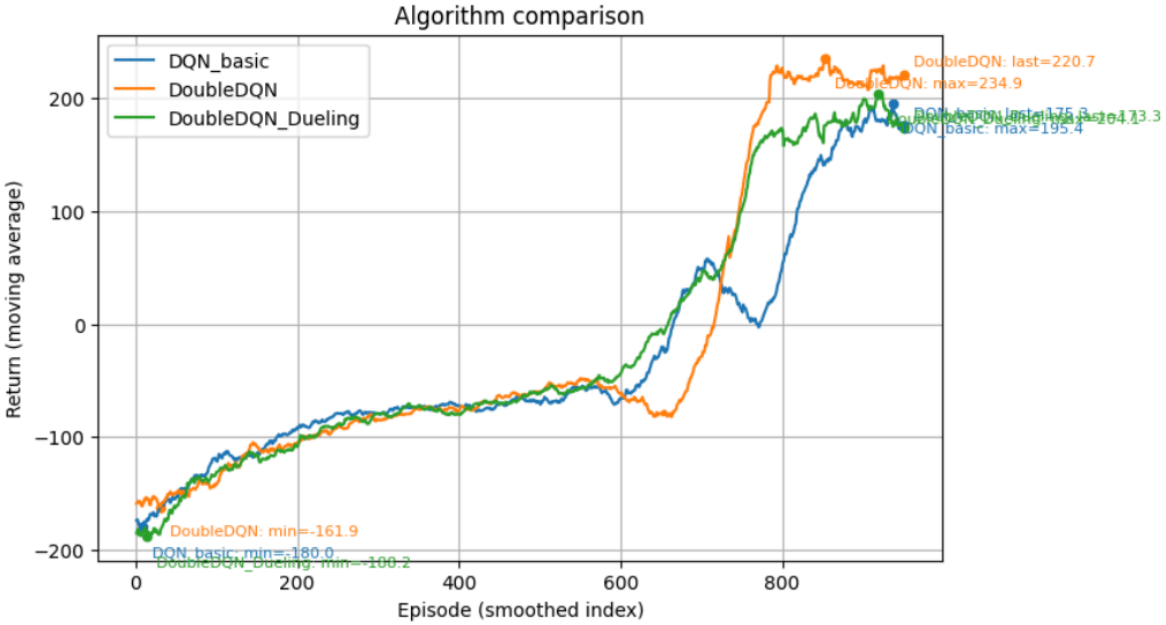
시간 및 자원 제약 때문에 모든 설정에서 많은 seed를 사용하지는 못했지만,
다음과 같은 방식으로 결과의 신뢰도를 보완하고자 하였다.

- 동일한 설정에 대해 seed=0,1 두 번의 독립 학습을 수행하고 평균/변동 범위를 함께 제시
- 알고리즘 비교, 보상 설계, 구성요소(ablation) 실험을 각각 따로 수행하여,
  한 가지 실험에 특화된 결과가 아니라 여러 축에 걸친 일관된 경향을 확인

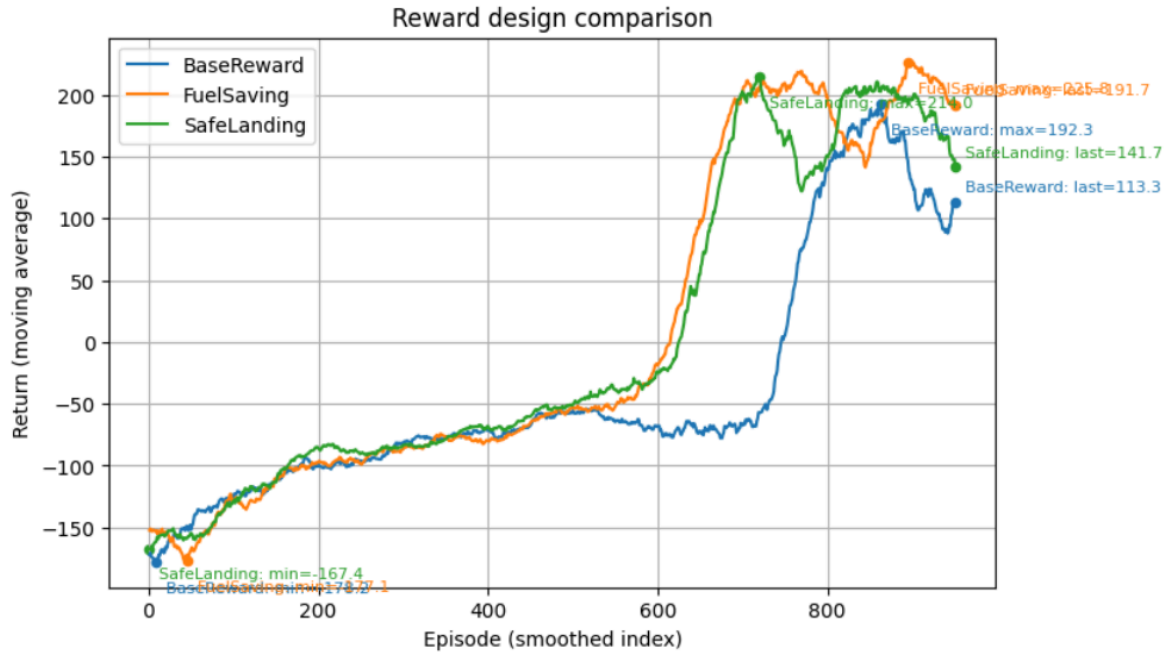
이를 통해 “seed 하나에서 우연히 잘 나온 결과”를 피하고,
비록 seed 개수는 2개로 제한적이지만,
여러 설정과 실험축을 통틀어 전반적인 성능 경향과 상대적인 비교를 수행하는 것을 목표로 하였다.

5. 실험 결과 (Results)

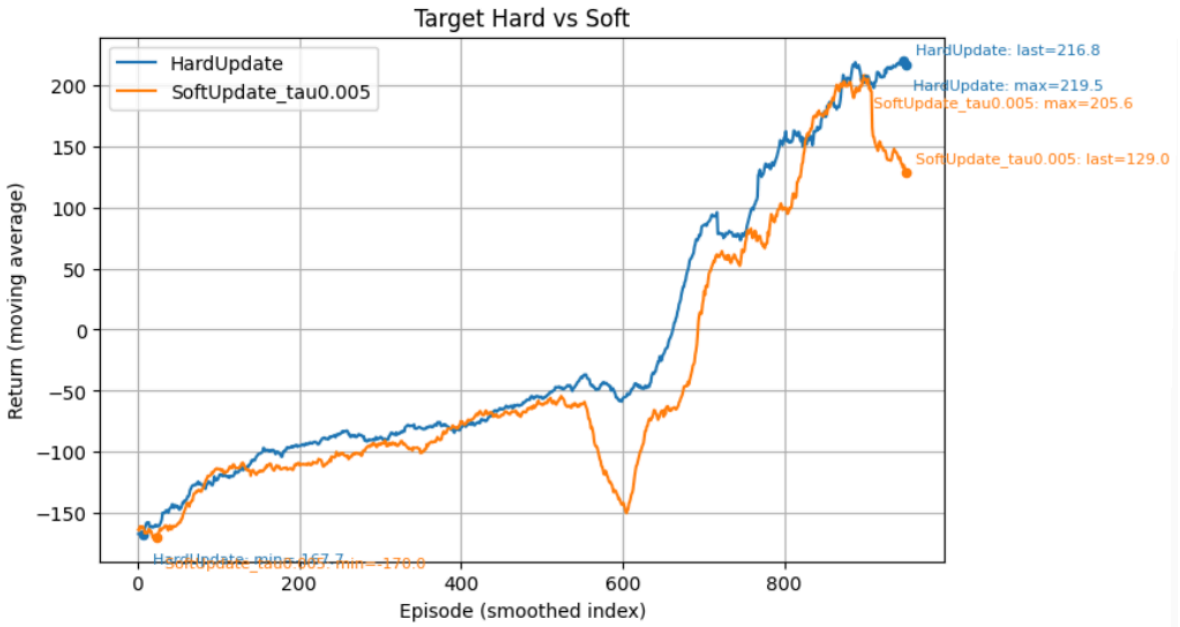
-



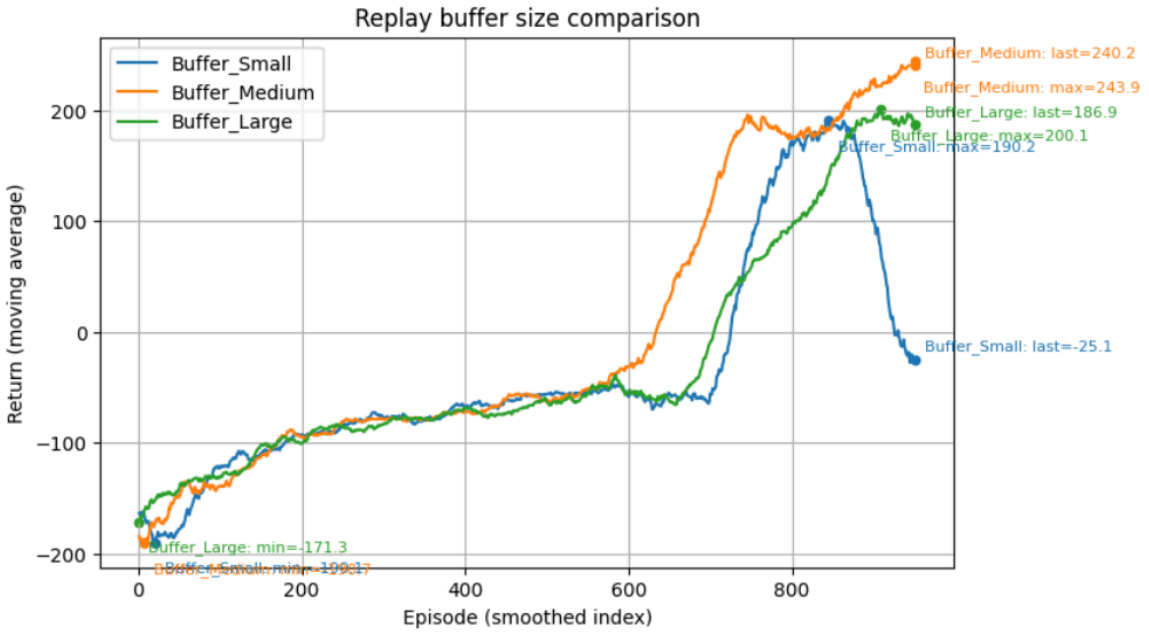
안정성 면에서 Dueling DQN이 강점을 보였다. 복잡하지 않은 환경이라 세 알고리즘 모두 나쁘지 않은 결과를 냈다.



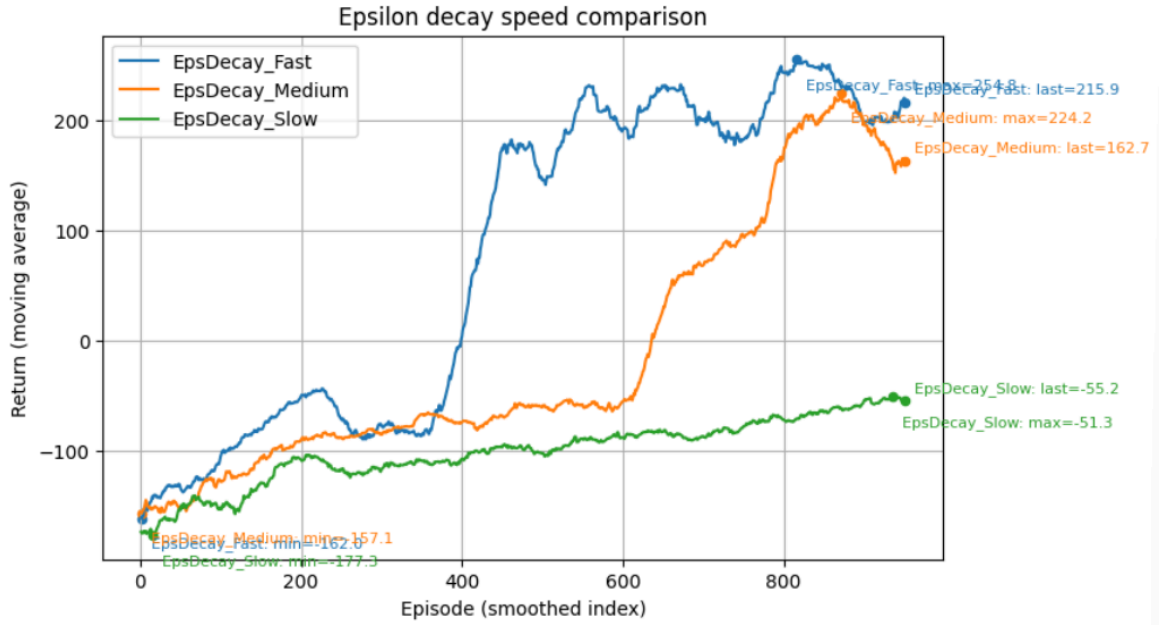
연료의 사용량에 따라 페널티를 주는 디자인이 가장 결과가 좋았다. 연료를 적게 쓸수록 에피소드가 진행 될수록 쓸데 없는 움직임이 줄어 빠르게 도착해 높은 리턴을 기록했다. 반면 수직으로 빠르게 이동하면 페널티를 주는 디자인은 안정성을 중요시 하다보니 도착시간이 늦어져 리턴값이 낮아진다.



Soft Update 학습이 너무 부드럽게 진행되다 보니, 에이전트가 "그냥 공중에 떠 있기만 하면 안 죽는다" 같은 엉성한 전략(Local Optima)에 안주하고 거기서 빠져나오지 못 한것 같다.Hard Update 주기적으로 네트워크가 확 바뀌면서 일종의 충격(Perturbation)을 준다다. 이 충격이 오히려 에이전트가 엉성한 전략을 버리고 새로운 시도를 하게 만드는 계기가 되어 더 좋은 결과를 냈다.



버퍼의 크기는 중간일 때가 가장 좋았다.버퍼 크기가 지나치게 작은 경우 샘플 간의 연관성이 높아져 과적합(Overfitting)이 발생하거나, 과거의 중요한 경험을 빠르게 잊어버리는 망각(Forgetting) 현상으로 인해 학습이 불안정해진다. 버퍼 크기가 지나치게 큰 경우 에이전트 성능이 낮았던 학습 초기의 '저품질 데이터'가 버퍼에 오랫동안 남아 학습 효율을 저하시키고 수렴 속도를 늦추는 원인이 된다. 또한, 최근에 수집한 '양질의 데이터'가 방대한 과거 데이터 속에 희석되어 배포(Sampling)될 확률이 낮아지는 문제가 발생한다. 즉, 작은 버퍼는 학습의 편향을 초래하고, 큰 버퍼는 비효율성을 유발하여 두 경우 모두 최종 성능 저하로 이어진다. 따라서 적절한 크기의 버퍼 설정이 필수적이다.



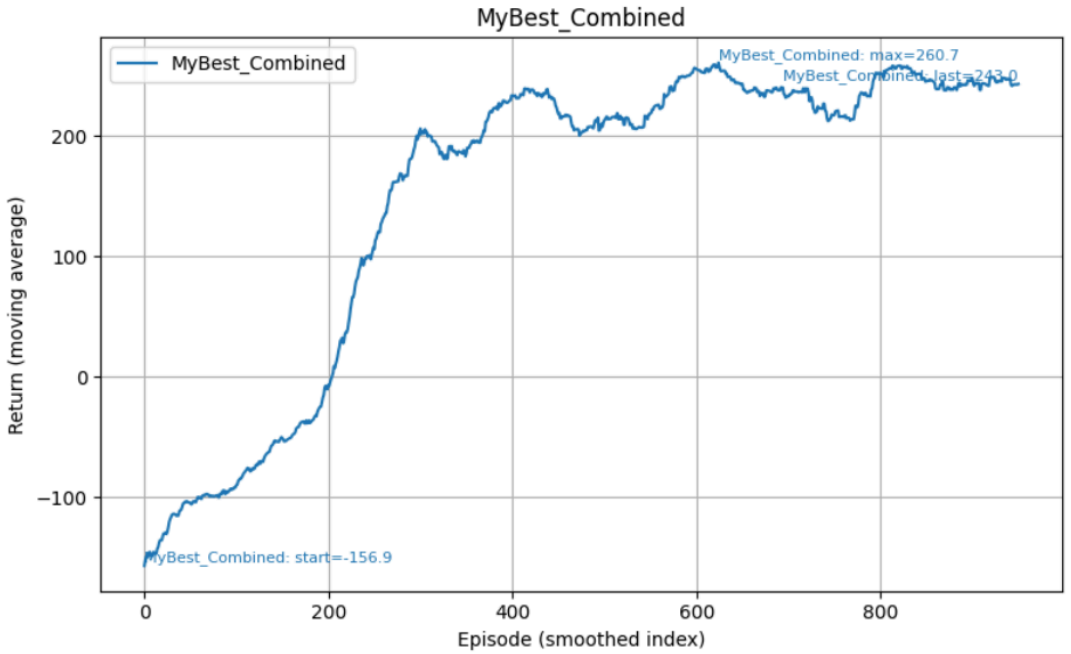
EspDecay값이 너무 느리다면 학습이 충분히 진행된 후반부에도 불필요한 무작위 행동(Random Action)이 지속된다. 이는 정책의 안정화(Stabilization)를 방해하고 수렴 속도를 현저히 지연시키며, 최종 성능의 분산(Variance)을 높이는 원인이 된다. 반면 너무

빠르다면 에이전트가 최적 정책(Global Optimum)이 아닌 국소 최적해(Local Optimum)에 고착되는 조기 수렴(Premature Convergence) 문제를 야기한다.

각 실험마다 가장 좋은 결과를 내는 파라미터와 알고리즘을 자동으로 선택하게 해서 AutoBest_Combined를 만들었다. Basic DQN, FuelSaving Reward, Hard update, Medium size Buffer, Fast esp decay이 선택되었다. 위 그래프가 그 모델의 결과이다. 위에 수많은 모델들보다 확실히 향상된 모습을 보였다.

6. 결론 및 토의 (Conclusion and Discussion)

앞으로 어떤 알고리즘을 사용할 지, 어떤 하이퍼 파라미터를 사용할 지, 또 어떻게 조합을 할지 고민이 든다면, 각각 독립적으로 비교해보고 좋은 결과를 내는 것들을 하나씩 채택해 좋은 에이전트를 만들 수 있을 것이다.



마지막으로 위에서 실험한 결과를 근거로 직접 하이퍼 파라미터와 알고리즘을 설정해서 다시 결과를 내보았다.

```
"name": "MyBest_Combined",

# 알고리즘 설정
"use_double_dqn": True,     # Double DQN 사용
"use_dueling": False,       # Dueling 은 끔
"use_target_net": True,     # target network 사용
"use_soft_update": True,    # soft update 사용
"tau": 0.05,                # soft update 계수 τ

# 보상 셰이핑
"use_reward_shaping": True,
"fuel_penalty": 0.05,
"angle_penalty": 0.3,
"vel_penalty": 0.3,

# 버퍼 / 탐험 관련
"buffer_size": 100_000,     # 리플레이 버퍼 크기
"min_replay_size": 5_000,   # 워밍업
"eps_decay_steps": 40_000, # epsilon 더 빠르게 감소 (원래보다 작게)

# 학습 길이
"num_episodes": 1000,

# 기본값 덮어쓰는 하이퍼파라미터
"gamma": 0.97,              # 할인율 (BASE_CONFIG 의 0.99 대신)
# "batch_size": 64,         # BASE_CONFIG 가 이미 64이면 생략 가능
```

위 그림은 직접 선택한 하이퍼 파라미터다
dqn보다 더 안정적인 double_dqn을 선택했다.
위 실험에서 hard update가 더 높게 나온이유가 tau값이 너무 작은 것이라 판단해서, soft update로 바꾸고 tau값을 두배정도 늘렸다.
보상 셰이핑에서는 연료 소비 페널티를 더 강하게 주면 빠르게 에피소드를 끝낼 수 있을 것이라 생각해 늘렸고. 우주선이 넘어져 큰 페널티를 받지 않게 하기위해 각도와 수직속도 페널티를 주었다. 하지만 너무 크게 주면 에피소드가 길어져 리턴이 감소할 태니 적당한 수치를 선택했다.
또한 Discount Factor인 gamma를 조금 줄여 step을 줄이고 return을 늘리도록 유도했다.

그래프를 보면 이전에 했던 모든 실험들보다 확실하게 상향되었다. 이와 같은 개별적이고 체계적인 비교 분석을 통해 우리는 각 알고리즘 및 구성요소의 이점(장점)을 명확히 파악하고, 특정 설정이 더 나은 결과를 도출하는 근본적인 이유를 이해할 수 있었습니다. 이러한 분석적 접근 방식은 단순히 성능이 좋은 조합을 찾는 것을 넘어, 향후 더욱 복잡한 환경이나 다른 강화학습 문제에 이 지식을 적용하여 에이전트의 성능과 안정성을 한층 더 발전시키는 토대가 될 것입니다.