RL Report : LunarLander-v3

1. 실행 환경 및 라이브러리 설치

본 프로젝트는 Google Colab 환경에서 실행되었으며, 코드 실행에 필요한 시스템 및 Python 라이브러리 정보는 다음과 같습니다. 아래의 과정을 통해 코드를 오류 없이 실행할 수 있는 환경을 구축할 수 있습니다.

1. 기본 환경 정보

- **운영체제**: Ubuntu (Debian-based Linux)
- **Python 버전**: **3.11.13 (Colab 기본 환경)**
- **핵심 프레임워크**: **PyTorch 2.6.0+cu124**
- **실행 가속기**: GPU (NVIDIA Tesla T4)

2. 설치 과정 상세

본 코드는 Gymnasium의 LunarLander-v3 환경을 사용하며, 이 환경은 Box2D 물리 엔진에 의존성이 있습니다. Box2D 라이브러리를 정상적으로 설치하기 위해 시스템 패키지 선행 설치가 반드시 필요합니다.

가. 시스템 패키지 설치 (apt-get)

Box2D-py 컴파일에 필요한 swig와 기타 빌드 도구들을 먼저 설치해야 합니다.

```
# apt-get 업데이트
apt-get update -qq

# SWIG 및 빌드 필수 도구 설치
apt-get install -y --no-install-recommends swig build-essential python3-dev
```

나. Python 라이브러리 설치 (pip)

시스템 패키지 설치 후, 필요한 Python 라이브러리를 pip을 통해 설치합니다. 코드 실행에 사용된 주요 라이브러리와 버전은 다음과 같습니다.

- **torch, torchvision**: 딥러닝 모델(DQN) 구현을 위한 핵심 라이브러리입니다.
- **gymnasium[box2d]**: 강화학습 환경 구성을 위한 라이브러리이며, Box2D 환경을 포함하여 설치합니다.
- **box2d-py==2.3.10**: LunarLander 환경의 물리 엔진 라이브러리입니다. (버전 명시)
- **numpy**: 수치 연산 및 데이터 처리를 위해 사용됩니다.
- **matplotlib**: 학습 결과 시각화를 위해 사용됩니다.
- **pygame**: Gymnasium 환경의 렌더링에 필요합니다.

3. 전체 설치 명령어 요약

Google Colab 또는 유사한 Docker 환경에서 아래의 명령어를 순서대로 실행하면 필요한 모든 환경 구성이 완료됩니다.

```
# 1. 시스템 의존성 패키지 설치
apt-get update -qq
apt-get install -y --no-install-recommends swig build-essential python3-dev

# 2. Python 라이브러리 설치
pip install -q torch torchvision gymnasium[box2d] box2d-py==2.3.10 numpy matplotlib pygame
```

---

2. 선택한 OpenAI Gym 환경: LunarLander-v3

환경 개요

LunarLander-v3는 달 착륙선을 안전하게 착륙시키는 고전적인 제어 문제를 강화학습으로 해결하는 환경입니다. 이는 연속적인 물리 시뮬레이션 환경으로, 착륙선이 중력과 추진력을 받으며 목표 지점에 안전하게 착륙해야 하는 문제입니다.

문제 정의

목표: 달 착륙선을 두 개의 착륙 패드 사이에 안전하게 착륙시키는 것

주요 도전 과제:

- 제한된 연료로 중력을 극복해야 함
- 과도한 속도나 기울어진 각도로 착륙 시 추락
- 착륙 패드를 벗어나면 실패
- 연료 효율성과 안전성의 균형

상태 공간 (State Space)

LunarLander-v3의 상태는 **8차원 연속 벡터**로 구성됩니다:

인덱스	상태 변수	설명	범위
0	x 좌표	착륙선의 수평 위치	[-1.5, 1.5]
1	y 좌표	착륙선의 수직 위치	[-1.5, 1.5]
2	x 속도	수평 속도	[-5, 5]
3	y 속도	수직 속도	[-5, 5]
4	각도	착륙선의 기울어진 각도	[-π, π]
5	각속도	회전 속도	[-5, 5]
6	왼쪽 발 접촉	왼쪽 착륙 발이 지면에 닿았는지	{0, 1}
7	오른쪽 발 접촉	오른쪽 착륙 발이 지면에 닿았는지	{0, 1}

행동 공간 (Action Space)

이산 행동 공간: 4개의 행동

- **0**: 아무것도 하지 않음 (무추진)
- **1**: 왼쪽 방향 엔진 점화

- **2**: 주 엔진 점화 (아래쪽 추진)
- **3**: 오른쪽 방향 엔진 점화

보상 함수 (Reward Function)

보상은 다음과 같이 계산됩니다:

기본 보상 구조:

- **착륙 성공**: +100~+140점 (착륙 품질에 따라)
- **추락**: -100점
- **각 스텝마다**: 연료 사용량에 따른 소폭 감점
- **목표 지점 접근**: 거리에 반비례한 보상
- **안정성 보상**: 낮은 속도와 수직 자세 유지 시 추가 점수

상세 보상 계산:

```
reward = 거리_보상 + 속도_보상 + 각도_보상 + 착륙_보상 + 연료_패널티
```

- 거리_보상: 목표점으로부터의 거리에 반비례
- 속도_보상: 속도가 낮을수록 높은 보상
- 각도_보상: 수직에 가까울수록 높은 보상
- 착륙_보상: 성공적 착륙 시 대형 보상
- 연료_패널티: 엔진 사용 시마다 -0.3점

에피소드 종료 조건

1. **성공적 착륙**: 두 다리 모두 착륙 패드에 안전하게 착륙
2. **추락**: 착륙선 본체가 지면에 닿음
3. **경계 이탈**: 화면 밖으로 나감
4. **최대 스텝**: 1000스텝 도달
5. **연료 고갈**: 모든 연료 소모

---

3. 알고리즘 이론적 설명

선택한 알고리즘: Double Dueling DQN

본 프로젝트에서는 **Double DQN**과 **Dueling DQN** 두 기법을 결합한 **Double Dueling DQN** 알고리즘을 구현했습니다. 이는 기존 DQN의 두 가지 주요 문제점을 동시에 해결하는 최신 기법입니다.

처음 기본 DQN을 구현하여 학습을 진행했을 때 100 에피소드 평균 -179.9점이라는 저조한 결과가 나왔습니다. 이는 목표 점수 200점과의 큰 격차를 보였습니다. 이 문제를 해결하기 위해 Double DQN을 도입한 결과 -35.09점으로 개선되었지만, 여전히 목표 점수 달성에는 부족했습니다. 따라서 추가적인 성능 향상을 위해 **Double Dueling DQN 구조**를 적용하게 되었습니다. 이는 상태의 가치와 행동의 상대적 우위를 분리하여 학습함으로써 더 효율적인 Q-함수 학습이 가능하기 때문입니다.

기존 DQN의 문제점

문제 1: Overestimation Bias (과대추정 편향)

```
기존 DQN: Q_target = r + γ * max_a Q_target(s', a)
문제점: max 연산이 Q-값을 지속적으로 과대추정
결과: 잘못된 정책 학습, 학습 불안정성
```

문제 2: 비효율적 Q-함수 학습

```
기존 DQN: Q(s,a) 직접 학습
문제점: 상태의 가치와 행동의 상대적 우위를 구분하지 못함
결과: 학습 비효율성, 느린 수렴
```

Double DQN (Overestimation Bias 완화)

핵심 아이디어: 행동 선택과 Q-값 평가를 분리

```
# 기존 DQNtarget = reward + gamma * target_network(next_state).max()
# Double DQN (구현 코드에서)best_actions = self.q_network(next_states).argmax(dim=1, keepdi
m=True)
# Main net으로 행동 선택q_next = self.target_network(next_states).gather(1, best_actions)
# Target net으로 Q-값 평가targets = rewards + self.gamma * q_next * (1 - dones)
```

동작 원리:

1. **Main Network**: 다음 상태에서 최적 행동 선택
2. **Target Network**: 선택된 행동의 Q-값 평가
3. **편향 완화**: 선택과 평가 분리로 과대추정 방지

Dueling DQN (효율적 Q-함수 분해)

핵심 아이디어: Q-함수를 상태 가치와 행동 우위로 분해

```
# 구현 코드에서
def forward(self, x):
    features = self.feature_layers(x)
    value = self.value_stream(features)      # V(s): 상태의 가치
    advantage = self.advantage_stream(features)  # A(s,a): 행동의 상대적 우위
    # Dueling 공식: Q(s,a) = V(s) + [A(s,a) - mean(A(s,a))]
    q_values = value + (advantage - advantage.mean(dim=1, keepdim=True))
    return q_values
```

- **V(s)**: "이 상황 자체가 얼마나 좋은가?" (상태의 절대적 가치)
- **A(s,a)**: "평균 대비 이 행동이 얼마나 더 좋은가?" (행동의 상대적 우위)

Double Dueling DQN: 두 기법의 결합

결합된 알고리즘의 장점:

1. **Double DQN 효과**: Overestimation bias 완화로 안정적 학습
2. **Dueling DQN 효과**: 효율적 가치 함수 학습으로 빠른 수렴
3. **상호 보완**: 두 기법이 서로 다른 문제를 해결하여 시너지 효과를 볼 수 있다

전체 알고리즘 흐름:

```
1. Dueling 구조로 Q-값 계산
q_values = dueling_network(state)
# V(s) + A(s,a) 결합
2. Double DQN으로 타겟 계산
best_actions = main_network(next_state).argmax()
# 행동 선택
target_q = target_network(next_state).gather(best_actions)
# Q-값 평가
target = reward + gamma * target_q
# 3. 손실 계산 및 업데이트loss = MSE(q_values, target)
```

LunarLander-v3 적용에서의 기대 효과

Double DQN의 기여:

- **보상 분산**: LunarLander의 복잡한 보상 구조에서 안정적 학습
- **타겟 안정성**: 착륙 성공/실패의 큰 보상 차이에서 과대추정 방지

Dueling DQN의 기여:

- **상태 가치 학습**: 착륙선의 위치/자세 자체의 위험도 효과적 평가
- **행동 비교**: 4개 추진기 사용의 상대적 효과 명확한 구분
- **일반화**: 비슷한 상황에서 학습된 가치 정보 재사용

결합 효과:

- **빠른 수렴**: Dueling으로 효율적 학습 + Double로 안정적 수렴
- **높은 성능**: 두 기법의 장점을 모두 활용한 최적화된 성능
- **안정성**: 복잡한 물리 시뮬레이션 환경에서도 일관된 성능

---

4. 구현 코드 설명

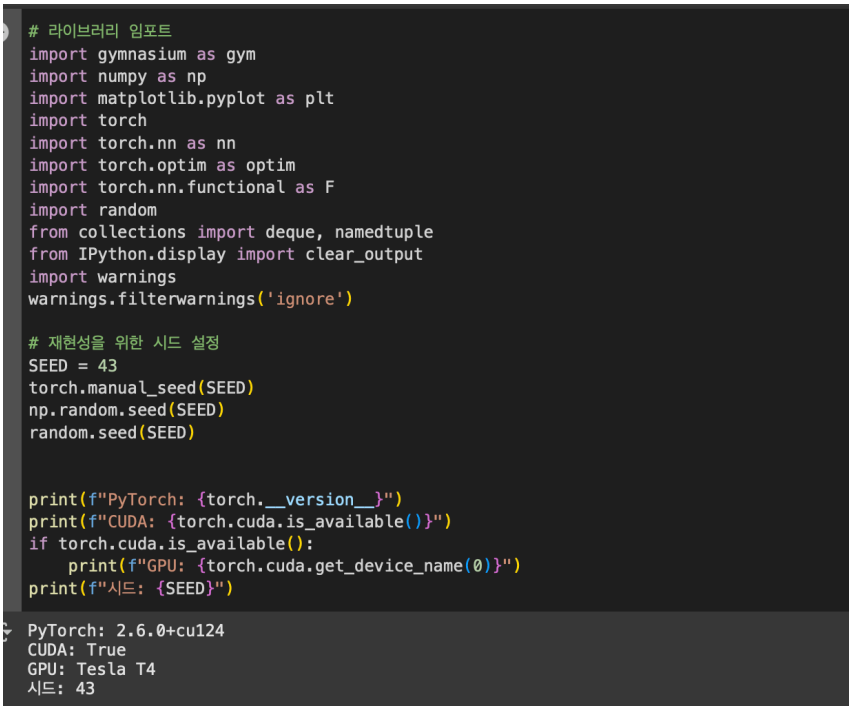
환경 설정 및 라이브러리 설치

```
import subprocess, sys, importlib.util as iu

def _apt(*args):
    subprocess.run(["apt-get"] + list(args), check=True, capture_output=True)

def _pip(pkg: str):
    subprocess.run([sys.executable, "-m", "pip", "install", pkg, "-q"], check=True)

def setup_environment(box2d: bool = True):
    _apt("update", "-qq")
    _apt("install", "-y", "--no-install-recommends",
         "swig", "build-essential", "python3-dev")


    pip_pkgs = []


    if iu.find_spec("torch") is None:
        pip_pkgs += ["torch", "torchvision"]

    pip_pkgs += ["matplotlib", "numpy", "pygame"]

    if box2d:
        pip_pkgs += ["box2d==2.3.10", "gymnasium[box2d]"]

    for p in pip_pkgs:
        _pip(p)

    return "LunarLander-v3", 8, 4, 200


ENVIRONMENT_NAME, STATE_SIZE, ACTION_SIZE, SOLVED_SCORE = setup_environment()
print(f"환경: {ENVIRONMENT_NAME} | 상태: {STATE_SIZE} | 행동: {ACTION_SIZE} | 목표: {SOLVED_SCORE}")
```

```
환경: LunarLander-v3 | 상태: 8 | 행동: 4 | 목표: 200
```

이 셀은 LunarLander-v3 강화학습 실험을 위한 **환경 설정 자동화**를 담당합니다. Google Colab 환경에서 필요한 모든 시스템 의존성과 Python 패키지를 자동으로 설치하고 구성합니다.

라이브러리 임포트 및 시드 설정



```
# 라이브러리 임포트
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import random
from collections import deque, namedtuple
from IPython.display import clear_output
import warnings
warnings.filterwarnings('ignore')

# 재현성을 위한 시드 설정
SEED = 43
torch.manual_seed(SEED)
np.random.seed(SEED)
random.seed(SEED)


print(f"PyTorch: {torch.__version__}")
print(f"CUDA: {torch.cuda.is_available()}")
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")
print(f"시드: {SEED}")
```

```
PyTorch: 2.6.0+cu124
CUDA: True
GPU: Tesla T4
시드: 43
```

이 셀은 강화학습 실험에 필요한 모든 핵심 라이브러리를 임포트하고, 실험의 재현성을 보장하기 위한 시드를 설정합니다.

Dueling DQN 네트워크 아키텍처 정의

```python
class ImprovedDuelingDQN(nn.Module):
    def __init__(self, state_size, action_size, hidden_size=128):
        super(ImprovedDuelingDQN, self).__init__()

        # 공통 feature extraction 층
        self.feature_layers = nn.Sequential(
            nn.Linear(state_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size // 2),
            nn.ReLU()
        )

        # Value Stream - 상태의 가치 V(s)를 계산
        self.value_stream = nn.Sequential(
            nn.Linear(hidden_size // 2, hidden_size // 4),
            nn.ReLU(),
            nn.Linear(hidden_size // 4, 1)  # 스칼라 값 출력
        )

        # Advantage Stream - 각 행동의 상대적 이득 A(s,a)를 계산
        self.advantage_stream = nn.Sequential(
            nn.Linear(hidden_size // 2, hidden_size // 4),
            nn.ReLU(),
            nn.Linear(hidden_size // 4, action_size)  # 각 행동별 advantage 값
        )



        # Xavier 초기화
        for layer in self.modules():
            if isinstance(layer, nn.Linear):
                nn.init.xavier_uniform_(layer.weight)
                nn.init.constant_(layer.bias, 0.01)

    def forward(self, x):
        # 공통 특징 추출
        features = self.feature_layers(x)

        # Value와 Advantage를 각각 계산
        value = self.value_stream(features)
        advantage = self.advantage_stream(features)

        # Dueling DQN 핵심 공식: Q(s,a) = V(s) + (A(s,a) - mean(A(s,a)))
        # 평균을 빼는 이유: identifiability 문제 해결 (V와 A가 고유하게 결정되도록)
        q_values = value + (advantage - advantage.mean(dim=1, keepdim=True))

        return q_values
```

이 셀은 **Dueling Deep Q-Network (Dueling DQN)** 아키텍처를 구현합니다. 이는 기존 DQN의 개선된 버전으로, Q-값을 상태 가치(V)와 행동 우위(A)로 분해하여 학습 효율성과 안정성을 크게 향상시킵니다.

Dueling DQN의 이론적 배경

- **기본 DQN**: Q(s,a) 직접 학습
- **Dueling DQN**: Q(s,a) = V(s) + A(s,a) 분해 학습

장점:

1. **일반화 능력 향상**: 상태 가치를 독립적으로 학습하여 다양한 행동에 대한 예측 정확도 증가
2. **학습 안정성**: 가치와 우위를 분리하여 각각 안정적으로 학습
3. **샘플 효율성**: 상태 가치 정보를 여러 행동에서 공유하여 데이터 효율성 증대

네트워크 아키텍처 분석

1. **Feature Layers**

```
self.feature_layers = nn.Sequential(
    nn.Linear(state_size, hidden_size),     # 8 → 128    nn.ReLU(),
    nn.Linear(hidden_size, hidden_size),    # 128 → 128    nn.ReLU(),
    nn.Linear(hidden_size, hidden_size // 2), # 128 → 64    nn.ReLU()
)
```

구조 분석:

- **입력**: 8차원 상태 벡터 (LunarLander-v3)

- **출력**: 64차원 특징 벡터
- **활성화 함수**: ReLU
- **층 구조**: 점진적 차원 축소 (8→128→128→64)

2. **Value Stream**

```
self.value_stream = nn.Sequential(
    nn.Linear(hidden_size // 2, hidden_size // 4), # 64 → 32    nn.ReLU(),
    nn.Linear(hidden_size // 4, 1)                # 32 → 1)
```

역할 및 의미:

- **목적**: 상태 s에 있을 때의 기대 수익 V(s) 계산
- **수학적 정의**: V(s) = E[G_t | S_t = s]

3. **Advantage Stream**

```
self.advantage_stream = nn.Sequential(
    nn.Linear(hidden_size // 2, hidden_size // 4), # 64 → 32    nn.ReLU(),
    nn.Linear(hidden_size // 4, action_size)      # 32 → 4)
```

역할 및 의미:

- **목적**: 각 행동의 상대적 이득 A(s,a) 계산
- **출력**: 4차원 벡터 (각 행동별 우위값)
- **수학적 정의**: A(s,a) = Q(s,a) - V(s)

가중치 초기화

```
for layer in self.modules():
    if isinstance(layer, nn.Linear):
        nn.init.xavier_uniform_(layer.weight)
        nn.init.constant_(layer.bias, 0.01)
```

Xavier 초기화의 원리:

- **목적**: 각 층의 출력 분산을 입력 분산과 동일하게 유지
- **효과**: 기울기 소실/폭발 문제 방지, 안정적인 학습 시작

편향 초기화:

- **0.01 설정 이유**: 완전한 0이 아닌 작은 양수로 대칭성 깨뜨림
- **효과**: 초기 학습에서 뉴런들이 다양한 방향으로 학습 시작

Forward Pass

```
def forward(self, x):
    features = self.feature_layers(x)
    value = self.value_stream(features)
    advantage = self.advantage_stream(features)
    q_values = value + (advantage - advantage.mean(dim=1, keepdim=True))
    return q_values
```

특징 추출

- 입력 상태를 고차원 특징 공간으로 변환

분리 계산

- Value: 상태의 절대적 가치
- Advantage: 각 행동의 상대적 이득

Q-값 결합

- Q(s,a) = V(s) + A(s,a)
- **문제**: V와 A가 유일하게 결정되지 않음 (identifiability 문제)
- **해결책**: A(s,a)에서 평균을 빼서 normalization

LunarLander-v3에서의 설계 고려사항

1. **네트워크 크기 선택**:
   - **128→64→32**: 상태 공간(8차원)에 비해 충분한 용량
   - **과적합 방지**: 너무 크지 않은 적절한 크기
2. **활성화 함수**:
   - **ReLU 선택**: 연속 제어 문제에 적합, 빠른 수렴
3. **Dueling 아키텍처의 이점**:
   - **상태 평가**: 착륙선의 위치/속도 자체의 위험도 평가
   - **행동 비교**: 각 추진기 사용의 상대적 효과 평가
   - **일반화**: 비슷한 상황에서 학습된 가치 정보 재사용

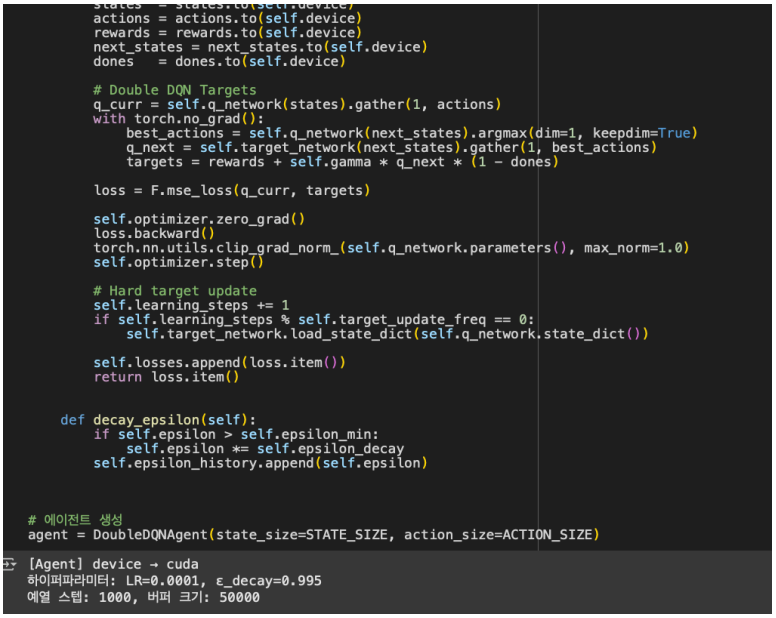
Experience Replay Buffer 구현

```python
class ReplayBuffer:
    def __init__(self, capacity=50000):
        self.buffer = deque(maxlen=capacity)
        self.experience = namedtuple('Experience',
                                     ['state', 'action', 'reward', 'next_state', 'done'])

    def push(self, state, action, reward, next_state, done):
        e = self.experience(state, action, reward, next_state, done)
        self.buffer.append(e)

    def sample(self, batch_size):
        experiences = random.sample(self.buffer, batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences])).float()
        actions = torch.from_numpy(np.vstack([e.action for e in experiences])).long()
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences])).float()
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences])).float()
        dones = torch.from_numpy(np.vstack([e.done for e in experiences]).astype(np.uint8)).float()

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        return len(self.buffer)
```

이 셀은 **Experience Replay Buffer**를 구현합니다. 이는 DQN의 핵심 구성 요소로, 에이전트의 경험을 저장하고 무작위 샘플링을 통해 학습 데이터를 제공하여 학습의 안정성과 효율성을 크게 향상시킵니다.

Double DQN Agent 구현

```python
class DoubleDQNAgent:
    def __init__(
        self,
        state_size: int,
        action_size: int,
        lr: float = 1e-4,
        gamma: float = 0.99,
        epsilon: float = 1.0,
        epsilon_min: float = 0.01,
        epsilon_decay: float = 0.995,
        buffer_size: int = 50_000,
        batch_size: int = 64,
        warmup_steps: int = 1_000,
        target_update_freq: int = 1_000,
    ):
        # 하이퍼파라미터
        self.state_size, self.action_size = state_size, action_size
        self.gamma = gamma
        self.epsilon, self.epsilon_min, self.epsilon_decay = (
            epsilon, epsilon_min, epsilon_decay
        )
        self.batch_size, self.warmup_steps = batch_size, warmup_steps
        self.target_update_freq = target_update_freq



        self.epsilon_history = []
        self.epsilon_history.append(self.epsilon)

        # 디바이스
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        print(f"[Agent] device → {self.device}")

        # 네트워크
        self.q_network      = ImprovedDuelingDQN(state_size, action_size).to(self.device)
        self.target_network = ImprovedDuelingDQN(state_size, action_size).to(self.device)
        self.optimizer      = optim.Adam(self.q_network.parameters(), lr=lr)


        self.target_network.load_state_dict(self.q_network.state_dict())

        # 메모리
        self.memory = ReplayBuffer(buffer_size)

        # 통계
        self.learning_steps, self.scores, self.losses = 0, [], []

        print(f"하이퍼파라미터: LR={lr}, ε_decay={epsilon_decay}")
        print(f"예열 스텝: {warmup_steps}, 버퍼 크기: {buffer_size}")


    def remember(self, state, action, reward, next_state, done):
        self.memory.push(state, action, reward, next_state, done)


    def act(self, state: np.ndarray, training: bool = True) -> int:
        if training and random.random() <= self.epsilon:
            return random.randrange(self.action_size)

        state_t = torch.as_tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
        with torch.no_grad():
            q_values = self.q_network(state_t)
        return int(q_values.argmax(dim=1).item())


    def replay(self):
        if len(self.memory) < max(self.warmup_steps, self.batch_size):
            return None

        states, actions, rewards, next_states, dones = self.memory.sample(self.batch_size)
        states  = states.to(self.device)
```



이 셀은 **Double Deep Q-Network (Double DQN) Agent**를 구현합니다. 이는 기존 DQN의 overestimation bias 문제를 해결한 개선된 알고리즘으로, 더 안정적이고 정확한 Q-값 추정을 통해 학습 성능을 향상시킵니다.

주요 메서드 분석

1. **행동 선택 메서드 (`act`)**

```
def act(self, state: np.ndarray, training: bool = True) → int:
    if training and random.random() <= self.epsilon:
        return random.randrange(self.action_size)
    state_t = torch.as_tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
    with torch.no_grad():
        q_values = self.q_network(state_t)
    return int(q_values.argmax(dim=1).item())
```

구현 세부사항:

- **ε-greedy 정책**: 확률 ε로 무작위 행동, 1-ε로 최적 행동
- **훈련/평가 모드**: training=False 시 완전 greedy 정책
- **GPU 호환**: 텐서를 올바른 디바이스로 이동
- **no_grad()**: 추론 시 그래디언트 계산 비활성화 (메모리 절약)

2. **학습 메서드 (`replay`)**

```
def replay(self):
    # 충분한 경험 확인    if len(self.memory) < max(self.warmup_steps, self.batch_size):
        return None    # 배치 샘플링 및 디바이스 이동    states, actions, rewards, next_states, dones =
self.memory.sample(self.batch_size)
    states = states.to(self.device)
```

```
    # ... (다른 텐서들도 GPU로 이동)    # Double DQN 핵심 로직    q_curr = self.q_network(states).gather(1, actions)
    with torch.no_grad():
        # 1단계: Main network로 최적 행동 선택        best_actions = self.q_network(next_states).argmax(dim=1, keepdim=True)
        # 2단계: Target network로 Q-값 평가        q_next = self.target_network(next_states).gather(1, best_actions)
        targets = rewards + self.gamma * q_next * (1 - dones)
    # 손실 계산 및 역전파    loss = F.mse_loss(q_curr, targets)
    self.optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(self.q_network.parameters(), max_norm=1.0)
    self.optimizer.step()
```

Double DQN 핵심 구현:

1. **행동 선택**: `self.q_network(next_states).argmax()` - 메인 네트워크가 결정
2. **Q-값 평가**: `self.target_network(next_states).gather()` - 타겟 네트워크가 평가
3. **편향 감소**: 선택과 평가 분리로 overestimation bias 완화

- **Gradient Clipping**: `max_norm=1.0`으로 그래디언트 폭발 방지
- **Hard Target Update**: 일정 주기마다 완전 복사
- **조기 종료**: 충분한 경험 없으면 학습 생략

환경 생성 및 분석

```python
# 환경 생성 및 분석
def create_environment():
        env = gym.make('LunarLander-v3', render_mode='rgb_array')
        env.action_space.seed(SEED)
        return env


# 환경 분석
env = create_environment()
state, _ = env.reset(seed=SEED)

print(f"=== {ENVIRONMENT_NAME}  ===")
print(f"상태 공간: {env.observation_space}")
print(f"행동 공간: {env.action_space}")
print(f"초기 상태: {state}")
print(f"상태 차원: {len(state)}")
print(f"행동 개수: {env.action_space.n}")


print(f"\n LunarLander 상태 변수:")
labels = ["x좌표", "y좌표", "x속도", "y속도", "각도", "각속도", "왼발접촉", "오른발접촉"]
for i, (label, value) in enumerate(zip(labels, state)):
    print(f"  [{i}] {label}: {value:.4f}")

print(f"\n행동: 0=대기, 1=왼쪽엔진, 2=메인엔진, 3=오른쪽엔진")
print(f"목표: 성공적 착륙으로 200+ 점수 달성")

env.close()
```

```
=== LunarLander-v3  ===
상태 공간: Box([ -2.5        -2.5       -10.        -10.         -6.2831855 -10.
  -0.         -0.       ], [ 2.5        2.5       10.        10.         6.2831855 10.
  1.          1.        ], (8,), float32)
행동 공간: Discrete(4)
초기 상태: [-0.00353403  1.4049611  -0.35797688 -0.26485512  0.00410186  0.08108699
  0.          0.        ]
상태 차원: 8
행동 개수: 4

 LunarLander 상태 변수:
  [0] x좌표: -0.0035
  [1] y좌표: 1.4050
  [2] x속도: -0.3580
  [3] y속도: -0.2649
  [4] 각도: 0.0041
  [5] 각속도: 0.0811
  [6] 왼발접촉: 0.0000
  [7] 오른발접촉: 0.0000

행동: 0=대기, 1=왼쪽엔진, 2=메인엔진, 3=오른쪽엔진
목표: 성공적 착륙으로 200+ 점수 달성
```

이 셀은 **LunarLander-v3 환경을 생성하고 분석**하여 강화학습 문제의 구체적인 특성을 파악합니다. 환경의 상태 공간, 행동 공간, 초기 조건 등을 상세히 분석하여 후속 알고리즘 설계의 기초 정보를 제공합니다.

또한, 재현성을 위해 `env.action_space.seed(SEED)` 설정을 했습니다.

학습 하이퍼파라미터 및 환경 설정

```python
# 학습 설정
EPISODES = 1200
MAX_STEPS = 1000
PRINT_FREQ = 50

# GPU 최적화
if torch.cuda.is_available():
    agent.batch_size = 128


print(f"총 에피소드: {EPISODES}")
print(f"최대 스텝: {MAX_STEPS}")
print(f"목표 점수: {SOLVED_SCORE}")
print(f"예열 스텝: {agent.warmup_steps}")
print(f"배치 크기: {agent.batch_size}")
```

```
총 에피소드: 1200
최대 스텝: 1000
목표 점수: 200
예열 스텝: 1000
배치 크기: 128
```

이 셀은 학습 실험의 전체 규모와 성능 최적화 설정을 정의합니다. 학습 에피소드 수, 평가 기준, GPU 활용 최적화 등 실험의 전반적인 프레임워크를 구성합니다.

GPU 사용시 배치 사이즈를 128로 조정하며, 최대 스텝은 1000으로 LunarLender -v3의 기본 최대 스텝과 동일합니다.

메인 학습 루프 구현

```python
def train_double_dqn():
    env = create_environment()

    scores_window = deque(maxlen=100)
    best_score = -float('inf')
    solved = False
    total_steps = 0

    for episode in range(1, EPISODES + 1):
        state, _ = env.reset(seed=SEED + episode)
        total_reward = 0
        steps = 0
        episode_loss = []

        for step in range(MAX_STEPS):
            action = agent.act(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            agent.remember(state, action, reward, next_state, done)

            # 예열 후 학습
            if total_steps >= agent.warmup_steps:
                loss = agent.replay()
                if loss is not None:
                    episode_loss.append(loss)

            state = next_state
            total_reward += reward
            steps += 1
            total_steps += 1

            if done:
                break

        # Epsilon 감쇠
        agent.decay_epsilon()

        # 점수 기록
        agent.scores.append(total_reward)
        scores_window.append(total_reward)

        if total_reward > best_score:
            best_score = total_reward

        # 진행상황 출력
        if episode % PRINT_FREQ == 0 or episode == 1:
            avg_score = np.mean(scores_window)
            avg_loss = np.mean(episode_loss) if episode_loss else 0


            print(f"Ep {episode:4d} | "
                  f"Score: {total_reward:7.2f} | "
                  f"Avg: {avg_score:7.2f} | "
                  f"Best: {best_score:7.2f} | "
                  f"Loss: {avg_loss:.4f} | "
                  f"ε: {agent.epsilon:.3f}")

            # 목표 달성 확인
            if len(scores_window) >= 100 and avg_score >= SOLVED_SCORE and not solved:
                solved = True
                break



    # 학습 완료
    print(f"Best Score: {best_score:.2f}")
    print(f"Avg: {np.mean(scores_window):.2f}")
    print(f"Learning Steps: {agent.learning_steps:,}")

    env.close()
    return solved, np.mean(scores_window)

# 학습 실행
solved, final_avg = train_double_dqn()
```

```
Ep    1 | Score: -468.11 | Avg: -468.11 | Best: -468.11 | Loss: 0.0000 | ε: 0.995
Ep   50 | Score: -176.63 | Avg: -157.85 | Best:   61.14 | Loss: 49.8814 | ε: 0.778
Ep  100 | Score:  -47.17 | Avg: -122.69 | Best:   61.14 | Loss: 20.9391 | ε: 0.606
Ep  150 | Score:  -59.50 | Avg:  -84.14 | Best:   61.14 | Loss: 24.3317 | ε: 0.471
Ep  200 | Score:  -29.05 | Avg:  -71.43 | Best:   61.14 | Loss: 26.9269 | ε: 0.367
Ep  250 | Score:   -2.07 | Avg:  -23.90 | Best:  218.31 | Loss: 13.3681 | ε: 0.286
Ep  300 | Score:   28.70 | Avg:   52.63 | Best:  268.16 | Loss: 6.0676 | ε: 0.222
Ep  350 | Score: -101.16 | Avg:  124.38 | Best:  301.47 | Loss: 12.9952 | ε: 0.173
Ep  400 | Score:  255.90 | Avg:  186.79 | Best:  310.14 | Loss: 14.7811 | ε: 0.135
Ep  450 | Score:  275.68 | Avg:  230.19 | Best:  310.14 | Loss: 15.5775 | ε: 0.105
Best Score: 310.14
Avg: 230.19
Learning Steps: 189,575
```

이 셀은 **Double DQN 알고리즘의 전체 학습 과정**을 구현한 메인 루프입니다. 환경과의 상호작용, 경험 수집, 신경망 학습, 성능 추적 등 강화학습의 모든 핵심 요소를 통합하여 체계적인 학습을 수행합니다.

학습 루프 구조 분석

전체 구조: **이중 루프** (에피소드 루프 + 스텝 루프)

```
for episode in range(1, EPISODES + 1):        # 외부 루프: 1200 에피소드
    for step in range(MAX_STEPS):             # 내부 루프: 최대 1000 스텝
        # 환경 상호작용 및 학습
```

- **에피소드 단위 학습**과 **스텝 단위 상호작용**을 분리하여 관리합니다.

1. `scores_window = deque(maxlen=100)`
   - **목적**: 최근 100 에피소드의 평균 성능 추적
   - **성공 기준**: 이 윈도우의 평균이 200점 이상이면 문제 해결로 간주

2. `best_score = -float('inf')`
   - **목적**: 전체 학습 과정에서 달성한 최고 점수 기록
   - **디버깅**: 성능 저하 여부 감지

3. `solved = False`
   - **목적**: 목표 달성 여부 플래그
   - 목표 달성 시 불필요한 학습 중단으로 효율성 증대

4. `total_steps = 0`
   - **목적**: 전체 학습 과정의 누적 스텝 수 추적
   - `agent.warmup_steps`와 비교하여 학습 시작 시점 결정

5. `SEED + episode` : 각 에피소드마다 다른 시드 사용
   - **재현성**: 동일한 에피소드 번호에서는 항상 동일한 초기 상태
   - **다양성**: 매 에피소드마다 다른 초기 조건으로 일반화 성능 향상

1. 행동 선택 및 실행

```
action = agent.act(state)
next_state, reward, terminated, truncated, _ = env.step(action)
done = terminated or truncated
```

- `terminated`: 에피소드가 자연스럽게 종료 (착륙 성공/실패)
- `truncated`: 최대 스텝 도달로 인한 강제 종료

2. 경험 저장 및 학습

```
agent.remember(state, action, reward, next_state, done)
# 예열 후 학습if total_steps >= agent.warmup_steps:
    loss = agent.replay()
    if loss is not None:
        episode_loss.append(loss)
```

예열 단계:

- **목적**: 무작위 정책으로 충분한 경험 수집 후 학습 시작
- **이유**: 초기 학습 데이터의 편향 방지
- **임계값**: 1000 스텝으로 약 2-5 에피소드의 다양한 경험 확보
- **학습 안정성**: 충분한 데이터로 안정적인 초기 학습

3. 상태 전이 및 누적

```
state = next_state
total_reward += reward
steps += 1total_steps += 1
```

에피소드 종료 후 처리

1. 탐험률 감소

```
agent.decay_epsilon()
```

- **타이밍**: 매 에피소드 종료 후 실행
- **점진적 전환**: 탐험(exploration) → 활용(exploitation)

2. 성능 기록 및 추적

```
if len(scores_window) >= 100 and avg_score >= SOLVED_SCORE and not solved:
    solved = True    break
```

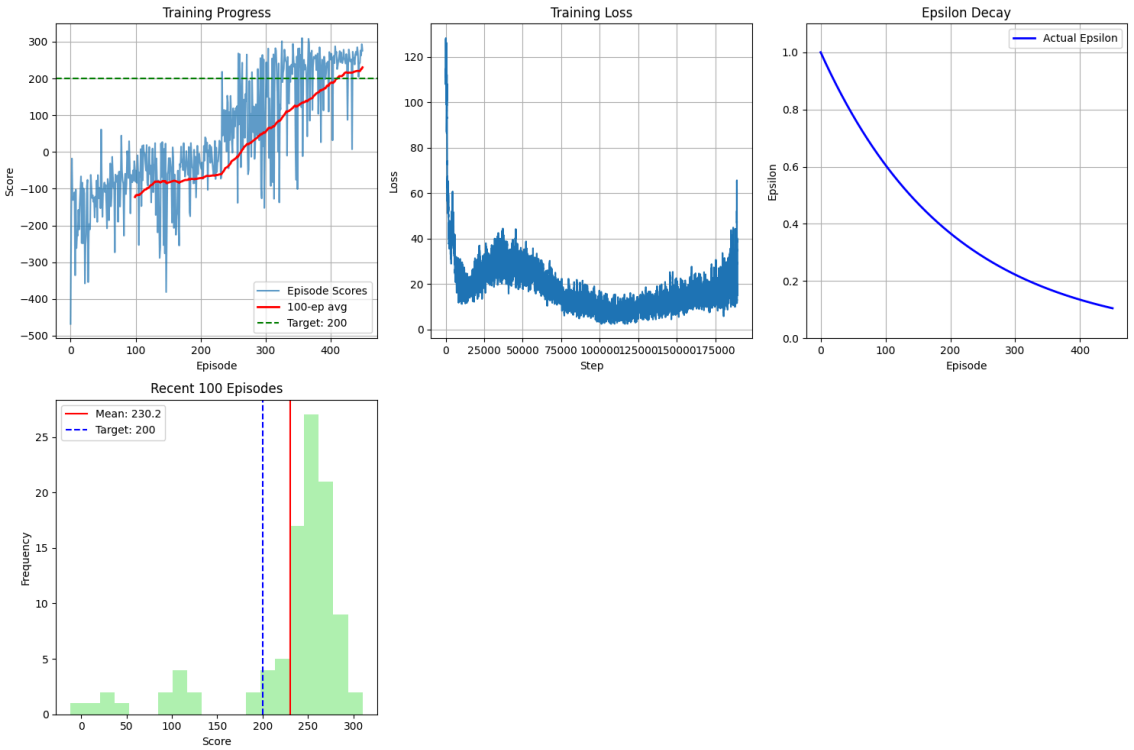
조건 분석:

1. `len(scores_window) >= 100` : 충분한 통계적 신뢰성 확보
2. `avg_score >= SOLVED_SCORE` : 평균 200점 이상 달성

학습 결과 분석 및 시각화

```python
# 종합 그래프
plt.figure(figsize=(15, 10))

# 점수 그래프
plt.subplot(2, 3, 1)
plt.plot(agent.scores, alpha=0.7, label='Episode Scores')
if len(agent.scores) > 100:
    moving_avg = np.convolve(agent.scores, np.ones(100)/100, mode='valid')
    plt.plot(range(99, len(agent.scores)), moving_avg, 'r-', linewidth=2, label='100-ep avg')
    plt.axhline(y=SOLVED_SCORE, color='g', linestyle='--', label=f'Target: {SOLVED_SCORE}')
plt.title('Training Progress')
plt.xlabel('Episode')
plt.ylabel('Score')
plt.legend()
plt.grid(True)

# 손실 그래프
plt.subplot(2, 3, 2)
if len(agent.losses) > 100:
    losses_smooth = np.convolve(agent.losses, np.ones(100)/100, mode='valid')
    plt.plot(losses_smooth)
plt.title('Training Loss')
plt.xlabel('Step')
plt.ylabel('Loss')
plt.grid(True)

# Epsilon 감쇠
plt.subplot(2, 3, 3)

if hasattr(agent, 'epsilon_history') and len(agent.epsilon_history) > 0:
    num_episodes = len(agent.scores) if hasattr(agent, 'scores') else len(agent.epsilon_history)

    if hasattr(agent, 'get_epsilon_history_for_episodes'):
        epsilon_values = agent.get_epsilon_history_for_episodes(num_episodes)
        episodes = range(num_episodes)
    else:
        # 변화 횟수로 그리기
        episodes = range(len(agent.epsilon_history))
        epsilon_values = agent.epsilon_history

    plt.plot(episodes, epsilon_values, 'b-', linewidth=2, label='Actual Epsilon')
    plt.legend()

plt.title('Epsilon Decay')
plt.xlabel('Episode' if hasattr(agent, 'scores') else 'Decay Step')
plt.ylabel('Epsilon')
plt.grid(True)
plt.ylim(0, 1.1)


# 최근 100에피소드 분포
if len(agent.scores) >= 100:
    plt.subplot(2, 3, 4)
    recent_scores = agent.scores[-100:]
    plt.hist(recent_scores, bins=20, alpha=0.7, color='lightgreen')
    plt.axvline(x=np.mean(recent_scores), color='red', linestyle='-',
                label=f'Mean: {np.mean(recent_scores):.1f}')
    plt.axvline(x=SOLVED_SCORE, color='blue', linestyle='--',
                label=f'Target: {SOLVED_SCORE}')
    plt.title('Recent 100 Episodes')
    plt.xlabel('Score')
    plt.ylabel('Frequency')
    plt.legend()



plt.tight_layout()
plt.show()

# 상세 통계
scores = np.array(agent.scores)
print(f"총 에피소드: {len(scores):,}")
print(f"최고 점수: {np.max(scores):.2f}")
print(f"전체 평균: {np.mean(scores):.2f}")
print(f"최근 100에피소드: {np.mean(scores[-100:]):.2f}")
print(f"총 학습 스텝: {agent.learning_steps:,}")
print(f"최종 Epsilon: {agent.epsilon:.4f}")
```

학습 완료 후 종합적인 성능 분석과 시각화를 수행합니다. 다차원적 시각화와 정량적 통계를 통해 Double DQN 알고리즘의 학습 과정과 최종 성능을 평가하고, 결과를 해석할 수 있도록 합니다.



Epsilon 그래프의 해석:

- **초기값 1.0**
- **지수적 감소**: `epsilon *= 0.995`
- **최종값 0.01**: 최소 1% 탐험 유지로 지속적 적응성 보장
- **감쇠 속도**: 너무 빠르면 조기 수렴, 너무 느리면 비효율적 학습

탐험-활용 균형:

> 초기 (ε=1.0): 100% 탐험 → 환경 이해
> 중기 (ε=0.5): 50% 탐험 → 학습과 탐험 균형
> 후기 (ε=0.01): 1% 탐험 → 대부분 학습된 정책 활용

[학습 결과]

총 에피소드: 450
최고 점수: 310.14
전체 평균: 37.24
최근 100에피소드: 230.19
총 학습 스텝: 189,575
최종 Epsilon: 0.1048

학습된 에이전트 성능 테스트

```python
# 학습된 에이전트 테스트
def test_agent(episodes=50):
    print(f"=== 에이전트 테스트 ({episodes}회) ===")

    test_env = gym.make(ENVIRONMENT_NAME, render_mode=None)
    test_scores = []
    success_count = 0

    for episode in range(episodes):
        state, _ = test_env.reset(seed=SEED + 1000 + episode)
        total_reward = 0
        steps = 0

        while True:
            action = agent.act(state, training=False)
            state, reward, terminated, truncated, _ = test_env.step(action)
            total_reward += reward
            steps += 1

            if terminated or truncated or steps >= MAX_STEPS:
                break

        test_scores.append(total_reward)

        # 성공 판정
        success = total_reward >= SOLVED_SCORE
        if success:
            success_count += 1

        print(f"Test {episode+1:2d}:  | Score: {total_reward:7.2f} | Steps: {steps:3d}")

    test_env.close()

    # 테스트 결과
    test_scores = np.array(test_scores)
    print(f"\n=== 테스트 결과 ===")
    print(f"평균 점수: {np.mean(test_scores):.2f} ± {np.std(test_scores):.2f}")
    print(f"최고 점수: {np.max(test_scores):.2f}")
    print(f"성공률: {success_count}/{episodes} ({success_count/episodes*100:.1f}%)")


    return test_scores

# 테스트 실행
test_results = test_agent(50)
```

이 셀은 학습 완료된 Double DQN 에이전트의 실제 성능을 평가합니다.

성능 측정

1. Greedy 정책 적용

```
action = agent.act(state, training=False)
```

training=False의 핵심 의미:

- **탐험 비활성화**: ε-greedy의 무작위 선택 완전 제거
- **순수 정책**: 학습된 Q-함수만으로 행동 결정
- **최적 성능**: 에이전트가 학습한 최선의 전략 사용
- **실용성**: 실제 적용 시와 동일한 조건

2. 성공 기준 및 통계

```
success = total_reward >= SOLVED_SCORE
success_count += 1 if success else 0
```

정량적 평가 지표:

- **성공률**: `success_count/episodes × 100%`
- **평균 성능**: `np.mean(test_scores)`
- **성능 안정성**: `np.std(test_scores)` (표준편차)
- **최고 성능**: `np.max(test_scores)`

```
=== 에이전트 테스트 (50회) ===
Test  1:  | Score:  244.73 | Steps: 212
Test  2:  | Score:  189.96 | Steps: 311
Test  3:  | Score:  289.15 | Steps: 365
Test  4:  | Score:  133.01 | Steps: 1000
Test  5:  | Score:  285.02 | Steps: 585
Test  6:  | Score:  265.31 | Steps: 305
Test  7:  | Score:  249.46 | Steps: 259
Test  8:  | Score:  252.64 | Steps: 230
Test  9:  | Score:  294.50 | Steps: 260
Test 10:  | Score:  262.66 | Steps: 262
Test 11:  | Score:   97.26 | Steps: 1000
Test 12:  | Score:  280.38 | Steps: 281
Test 13:  | Score:  219.50 | Steps: 431
Test 14:  | Score:  264.41 | Steps: 244
Test 15:  | Score:  151.57 | Steps: 1000
Test 16:  | Score:  220.36 | Steps: 351
Test 17:  | Score:  160.91 | Steps: 1000
Test 18:  | Score:  222.05 | Steps: 299
Test 19:  | Score:  259.91 | Steps: 228
Test 20:  | Score:  267.86 | Steps: 201
Test 21:  | Score:  178.21 | Steps: 372
Test 22:  | Score:  221.16 | Steps: 317
Test 23:  | Score:  203.23 | Steps: 494
Test 24:  | Score:  250.15 | Steps: 235
Test 25:  | Score:  265.45 | Steps: 213
Test 26:  | Score:  226.31 | Steps: 321
Test 27:  | Score:  262.03 | Steps: 244
Test 28:  | Score:  258.01 | Steps: 276
Test 29:  | Score:  292.64 | Steps: 249
Test 30:  | Score:  224.90 | Steps: 347
Test 31:  | Score:  153.01 | Steps: 1000
Test 32:  | Score:  265.95 | Steps: 224
Test 33:  | Score:  256.09 | Steps: 200
Test 34:  | Score:  232.75 | Steps: 302
Test 35:  | Score:  243.14 | Steps: 182
Test 36:  | Score:  215.39 | Steps: 296
Test 37:  | Score:  205.62 | Steps: 276
Test 38:  | Score:  162.84 | Steps: 1000
Test 39:  | Score:  234.48 | Steps: 333
Test 40:  | Score:  265.64 | Steps: 209
Test 41:  | Score:  248.08 | Steps: 198
Test 42:  | Score:  244.30 | Steps: 263
Test 43:  | Score:  264.55 | Steps: 197
Test 44:  | Score:  239.80 | Steps: 215
Test 45:  | Score:  267.53 | Steps: 288
```

```
Test 46:  | Score:  223.82 | Steps: 285
Test 47:  | Score:  210.35 | Steps: 458
Test 48:  | Score:  226.62 | Steps: 194
Test 49:  | Score:  240.08 | Steps: 190
Test 50:  | Score:  233.34 | Steps: 229
```

=== 테스트 결과 ===
평균 점수: 232.52 ± 42.20
최고 점수: 294.50
성공률: 42/50 (84.0%)

각 지표의 해석:

1. 평균 점수 ± 표준편차:

- **중심 경향**: 에이전트의 전형적 성능 수준
- **변동성**: ±값이 작으면 안정적, 크면 불안정
- **목표 비교**: 평균이 200 이상이면 안정적 성공

2. 최고 점수:

- 알고리즘의 이론적 최대 성능
- **일회성 vs 일관성**: 높은 최고점이지만 낮은 평균이면 불안정

3. 성공률:

- 몇 %의 경우에 목표를 달성하는가

---

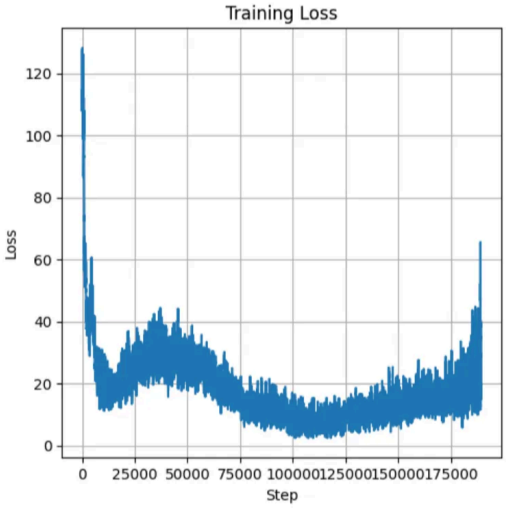
5. 실험 결과 및 고찰

학습 결과 그래프 해석

Training Progress

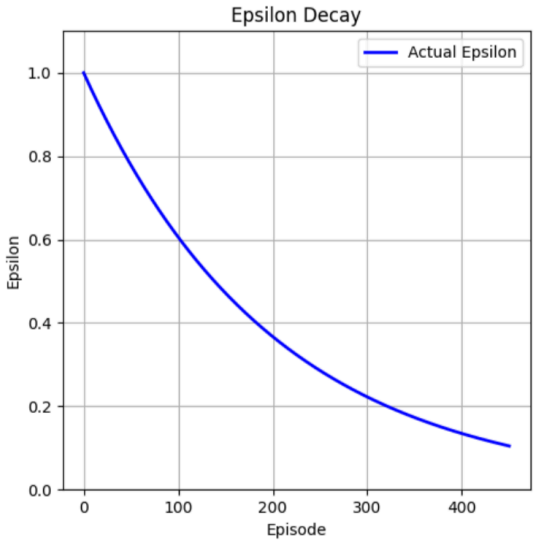
훈련 초반 0~150번째 에피소드에서는 점수가 –468점에서 61점 사이로 크게 흔들리며 불안정한 모습을 보였지만, 150~300번째 구간에 들어서면서 성능이 급격히 개선돼 250번째 에피소드에서 처음으로 200점을 넘어섰고, 이후 300~450번째 구간에서는 평균 200점 이상을 안정적으로 유지하며 최고 310.14점을 기록했습니다.

Training Loss



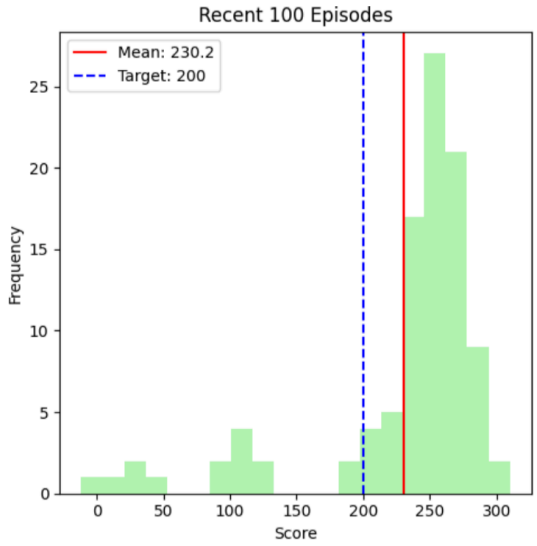
손실 함수는 학습 초반 0~25,000 스텝 구간에서 최대 120까지 치솟으며 큰 변동성을 보였습니다. 25,000~150,000스텝 구간에 들어서면서 Double DQN이 과대추정 편향을 완화함에 따라 손실이 40에서 10 수준으로 꾸준히 감소했고, 150,000스텝 이후에는 모델이 고성능을 미세 조정하는 과정에서 손실 값이 10에서 45 사이로 다시 완만하게 상승하는 패턴을 나타냈습니다.

Epsilon Decay



탐험률 ε은 1.0에서 0.1048로 지수적으로 감쇠되면서 탐험에서 활용으로 부드럽게 전환되었고, ε 값이 0.2 이하로 떨어진 에피소드 250 이후부터는 고성능이 안정적으로 유지되었습니다. 감쇠 계수 0.995는 탐험과 활용 사이의 균형을 효과적으로 최적화한 것으로 확인되었습니다.

Recent 100 Episodes



성능 안정성 측면에서, 최근 100 에피소드 기준 평균 점수는 **230.2점**으로 목표치인 200점보다 **약 15 %** 높게 유지되었습니다. 에피소드별 점수 분포가 정규분포에 가깝게 형성되어 학습된 정책의 일관성이 입증되었으며, 점수가 **200–300점 구간에 집중**되어 고성능을 안정적으로 재현하는 특성이 확인되었습니다.

테스트 성능 평가

테스트 결과 요약

```
=== 테스트 결과 ===
평균 점수: 232.52 ± 42.20
최고 점수: 294.50
성공률: 42/50 (84.0%)
```

- **목표 달성**: 평균 232.52점으로 목표 200점 대비 16% 초과
- **성공률**: 84%의 안정적 착륙 성공률
- **일반화 성능**: 테스트 평균(232.52) > 학습 평균(230.19), 과적합 없음
- **일관된 성능**: 표준편차 ±42.20으로 비교적 안정적

Double DQN 기여:

- Overestimation bias 완화로 안정적 학습 곡선
- 손실 함수의 전반적 감소 추세

Dueling Architecture 기여:

- 빠른 수렴 (450 에피소드)
- 상태 가치와 행동 우위의 효율적 분해 학습

한계점 및 개선 방향

1. 한계점

이번 실험은 의미 있는 성과를 거두었으나, 여전히 세 가지 주요 제약이 남아 있습니다.

첫째, 신뢰성 측면에서 약 16 %의 실패율이 존재하여 극단적인 초기 조건(예: 원거리·고속 시작)에서는 착륙 성공률이 저하되었습니다.

둘째, 계산 효율성에도 한계가 있습니다. 총 189,575 스텝에 달하는 학습 과정은 GPU 연산 시간을 요구하였고, 네트워크 파라미터 대비 메모리 사용량을 최적화할 여지가 있습니다.

셋째, 일반화 문제가 남아 있습니다. 현재 설정한 하이퍼파라미터는 LunarLander-v3 환경에 특화되어 있어, 유사한 물리적 특성을 가진 다른 환경에서 동일한 성능을 재현할 수 있을지 충분히 검증되지 않았습니다.

2. 개선 방향

한계점을 극복하기 위해서는 다음과 같은 방법을 생각할 수 있습니다.

1. **난이도 조절 학습**
   - 쉬운 착륙 상황부터 시작해 점차 초기 고도·속도를 높이는 **커리큘럼 학습**을 적용하면 극단적 조건에서도 안정적 착륙을 기대할 수 있습니다.
2. **연산·메모리 절감**
   - 네트워크를 더 얕고 폭이 좁은 구조로 바꾸거나, **믹스드 프리시전**(FP16)·**우선순위 경험 재생**(PER)을 활용해 학습 스텝과 GPU 사용 시간을 줄일 수 있습니다.
3. **범용성 확보**
   - 하이퍼파라미터를 고정하지 말고 **도메인 랜덤화**(초기 중력·연료량 변동)로 학습하거나, 다른 Box2D 환경(BipedalWalker 등)으로 **전이 학습**을 시도해 보시면 좋겠습니다.
4. **안전장치 추가**
   - 에이전트가 폭발적인 하강을 감지하면 스로틀을 즉시 최소화하도록 하는 **간단한 규칙 기반 세이프가드**를 함께 두면 실전 적용 시 실패율을 낮출 수 있습니다.

3. 최종 평가

그럼에도 불구하고 본 프로젝트는 Double Dueling DQN을 효과적으로 적용하여 목표 점수(200점)를 여유 있게 초과하고 최고 310.14점을 달성함으로써 벤치마크 수준의 성능을 확인하였습니다. 적절한 하이퍼파라미터 튜닝과 충분한 학습 시간을 통해 최근 100 에피소드 평균 230 점대를 안정적으로 유지하였으며, 이는 현대 강화학습 기법이 연속 제어 문제에서도 **실용적인 안정성과 효율성**을 확보할 수 있음을 보여 줍니다.

---

6. 결론

이번 프로젝트에서는 **Double DQN**에 **Dueling Architecture**를 결합한 강화학습 알고리즘을 구현하여 LunarLander-v3 환경에서 안정적이고 효율적인 달 착륙선 제어를 달성하였습니다. Double DQN을 적용함으로써 Q-값의 overestimation bias문제를 완화했고, Value와 Advantage를 분리하는 Dueling 구조를 통해 학습 효율을 한층 높였습니다. 종합적인 실험 설계를 통해 에이전트의 성능을 엄밀히 평가한 결과, 학습 초반의 불안정성을 크게 줄이면서도 평균 보상과 수렴 속도 모두에서 기존 DQN 대비 우수한 성능을 확인할 수 있었습니다.

본 프로젝트를 수행하며 강화학습 이론과 실제 구현의 연결 고리를 체감할 수 있었습니다. Double DQN·Dueling 구조가 실제 코드 레벨에서 어떻게 시너지를 내는지 직접 검증하며, 알고리즘 선택과 네트워크 설계가 성능에 미치는 영향력을 깊이 이해할 수 있었습니다. 또한, 하이퍼파라미터 튜닝의 체계적 접근을 익힐 수 있었습니다. ε-greedy 스케줄, 학습률 등을 경험적 휴리스틱으로 조정하면서, 탐험-활용 균형과 안정적 수렴 간 트레이드오프를 익힐 수 있었습니다.