

강화학습의 기초 기말 과제

강화학습 기반 재고 관리 최적화

PPO(Proximal Policy Optimization) 알고리즘을 활용하여 재고 관리 비용을 최소화하는 강화학습 프로젝트입니다.

조: 50조

A70077 이승훈

데이터사이언스&인공지능전공

프로젝트 주제 및 목표

주제

확률적 수요 환경에서 **재고 보유 비용**과 **품절 비용**을 동시에 고려한 최적 주문 정책 학습

목표

- 일일 수요가 불확실한 상황에서 **언제, 얼마나 주문할지** 자동으로 결정
- 재고 유지 비용 + 품절 비용 + 주문 비용의 **총 비용 최소화**
- 전통적인 (s, S) 정책 대비 강화학습 기반 정책의 성능 비교

왜 이 문제가 중요한가?

실제 기업에서 재고 관리는 핵심 과제입니다

- 재고가 **너무 많으면**: 창고 비용, 자본 묶임, 제품 노후화
- 재고가 **너무 적으면**: 품절로 인한 판매 손실, 고객 이탈

강화학습은 이런 **불확실한 수요 환경**에서 최적의 주문 정책을 스스로 학습할 수 있습니다.

프로젝트 구조

```

inventory_rl_project/
├── main.py          # 실행 진입점 (학습 및 평가)
├── config.py        # 환경/비용/학습 파라미터 설정
├── requirements.txt  # 의존성 패키지 목록
├── agents/
│   └── ppo_agent.py  # PPO 에이전트 클래스
├── envs/
│   └── inventory_env.py # 재고 관리 환경 (Gymnasium)
├── utils/
│   └── visuals.py    # 결과 시각화 함수
└── models/
    └── ppo_inventory_model.zip # 학습된 모델 파일

```

주요 함수/클래스 요약

파일	클래스/함수	기능
config.py	Config	환경, 비용, 학습 파라미터 상수 정의
inventory_env.py	InventoryEnv	재고 관리 시뮬레이션 환경
inventory_env.py	reset()	환경 초기화, 시작 상태 반환
inventory_env.py	step(action)	행동 수행 후 다음 상태/보상 반환
ppo_agent.py	PPOInventoryAgent	PPO 알고리즘 래퍼 클래스
ppo_agent.py	train()	모델 학습 수행
ppo_agent.py	predict()	상태에서 행동 예측
visuals.py	plot_results()	재고/주문/수요 그래프 출력
main.py	run_experiment()	학습 + 평가 전체 파이프라인

환경 및 데이터셋 설명

1. 문제 상황 설명

상황: 당신은 창고 관리자입니다. 매일 고객이 물건을 사러 오는데, 몇 개를 살지 정확히 모릅니다.

해야 할 일

- 매일 "오늘 몇 개를 주문할까?" 결정
- 주문한 물건은 **2일 후**에 도착 (리드타임)
- 목표: 100일 동안 **총 비용을 최소화**

어려운 점

- 수요를 정확히 모름 (확률적)
- 주문해도 바로 안 옵니다 (리드타임 존재)
- 너무 많이 주문하면 보관 비용 발생
- 너무 적게 주문하면 품질 비용 발생

2. 환경 파라미터 설정

`config.py` 에서 모든 환경 설정을 관리합니다

```
# 창고 관련
MAX_CAPACITY = 200    # 창고에 최대 200개까지 보관 가능
INIT_INV = 50         # 시작할 때 재고 50개로 시작
MAX_ORDER = 20        # 한 번에 최대 20개까지만 주문 가능
LEAD_TIME = 2         # 주문하면 2일 후에 도착
EPISODE_LENGTH = 100  # 100일 동안 시뮬레이션
```

3. 수요 모델 (데이터 생성)

실제 데이터셋 대신 확률 분포로 수요를 생성합니다.

```
# 매일 수요는 포아송 분포를 따름
demand = np.random.poisson(DEMAND_MEAN) # DEMAND_MEAN = 5
```

포아송 분포란?

- 평균이 5인 포아송 분포 사용
- 대부분 3~7개 수요 발생
- 가끔 0개 또는 10개 이상도 발생
- 무작위성이 있어서 예측이 어려움

예시: 10일간 수요

Day 1: 4개 | Day 2: 6개 | Day 3: 3개 | Day 4: 8개 | Day 5: 5개

Day 6: 2개 | Day 7: 7개 | Day 8: 5개 | Day 9: 4개 | Day 10: 6개

4. 비용 구조 (보상 설계의 기반)

비용 종류	금액	발생 조건	예시
재고 유지 비용	1.0원/개	창고에 재고가 남아있을 때	재고 30개 → 30원
품질 비용	10.0원/개	재고가 부족해서 수요를 못 채울 때	5개 부족 → 50원
주문 고정 비용	5.0원/회	주문을 넣을 때마다 (1개라도)	주문 1회 → 5원
주문 변동 비용	2.0원/개	주문 수량에 비례	10개 주문 → 20원

비용 계산 예시

현재 재고: 20개

오늘 주문: 10개

오늘 수요: 8개

→ 재고 유지 비용: $(20-8) \times 1.0 = 12$ 원 (남은 재고 12개에 대해)

→ 주문 비용: $5.0 + (10 \times 2.0) = 25$ 원

→ 총 비용: $12 + 25 = 37$ 원

5. 데이터 전처리 (Preprocessing)

이 프로젝트에서는 외부 데이터셋을 사용하지 않고, 환경이 직접 데이터를 생성합니다.

대신, 상태(State)를 신경망에 입력하기 위해 다음과 같이 처리합니다

```
# 상태 정규화 (observation_space에서 자동 처리)
```

```
# 재고: -200 ~ 200 범위로 제한
```

```
# 파이프라인: 0 ~ 20 범위로 제한
```

```
low = np.array([-200, 0, 0], dtype=np.float32) # 최소값
```

```
high = np.array([200, 20, 20], dtype=np.float32) # 최대값
```

```
observation_space = spaces.Box(low=low, high=high)
```

State, Action, Reward 설계

강화학습에서 가장 중요한 세 가지 요소입니다. 이 설계가 학습 성능을 결정합니다.

1. State (상태) - "에이전트가 보는 정보"

에이전트가 의사결정을 위해 알아야 하는 정보입니다.

State = [현재_재고, 파이프라인_주문1, 파이프라인_주문2]

요소	의미	범위	예시
현재_재고	지금 창고에 있는 물건 수	-200 ~ 200	35.0
파이프라인_주문1	내일 도착할 주문량	0 ~ 20	10.0
파이프라인_주문2	모레 도착할 주문량	0 ~ 20	5.0

왜 이렇게 설계했나?

1. **현재 재고**: 가장 기본적인 정보. 재고가 많으면 주문을 줄이고, 적으면 늘려야 함
2. **파이프라인 주문**: 리드타임이 있으므로, "앞으로 도착할 물량"을 알아야 중복 주문 방지

음수 재고 (백오더)

- 재고가 -10이면: 수요 10개를 못 채워서 "빚진" 상태
- 나중에 물건이 도착하면 먼저 이 빚을 갚음

코드 구현

```
# envs/inventory_env.py
def reset(self, seed=None, options=None):
    current_inv = float(Config.INIT_INV) # 초기 재고 50
    pipeline = [0 for _ in range(self.lead_time)] # [0, 0]
    self.state = np.array([current_inv] + pipeline, dtype=np.float32)
    return self.state, {}
```

2. Action (행동) - "에이전트가 할 수 있는 것"

매 시점마다 에이전트가 선택하는 행동입니다.

Action $\in \{0, 1, 2, 3, \dots, 20\}$

행동	의미
0	오늘 주문 안 함
1	1개 주문
2	2개 주문
...	...
20	20개 주문 (최대)

이산 행동 공간 (Discrete Action Space)

- 21개의 선택지 중 하나를 고름
- 연속 값(예: 3.5개)이 아닌 정수만 가능

왜 이렇게 설계했나?

1. **단순성**: 연속 행동보다 학습이 쉬움
2. **현실성**: 실제로 "3.7개 주문"은 불가능
3. **최대 제한**: 한 번에 너무 많이 주문하면 비현실적

코드 구현

```
# envs/inventory_env.py
self.action_space = spaces.Discrete(Config.MAX_ORDER + 1) # 0~20, 총 21개
```

3. Reward (보상) - "행동의 좋고 나쁨을 알려주는 신호"

에이전트가 학습하는 기준입니다. 보상이 높을수록 좋은 행동입니다.

Reward = $-1 \times (\text{재고유지비용} + \text{품절비용} + \text{주문비용})$

왜 음수인가?

- 목표가 "비용 최소화"이므로, 비용이 낮을수록 보상이 높아야 함
- 비용 100원 \rightarrow 보상 -100

- 비용 50원 → 보상 -50 (더 좋음!)

보상 계산 과정

```
# 1. 재고 유지 비용 (재고가 양수일 때만)
holding_cost = 1.0 × max(0, 현재재고)

# 2. 품절 비용 (재고가 음수일 때만)
stockout_cost = 10.0 × max(0, -현재재고)

# 3. 주문 비용 (주문했을 때만)
if 주문량 > 0:
    ordering_cost = 5.0 + (2.0 × 주문량)
else:
    ordering_cost = 0

# 4. 최종 보상
reward = -(holding_cost + stockout_cost + ordering_cost)
```

보상 설계의 핵심 인사이트

상황	비용	보상	에이전트가 배우는 것
재고 너무 많음	높은 유지비용	낮은 보상	"덜 주문해야지"
품절 발생	높은 품절비용	매우 낮은 보상	"품절은 피해야지"
적절한 재고	낮은 총비용	높은 보상	"이 정도가 좋구나"

코드 구현

```
# envs/inventory_env.py - step() 함수 내부
holding_cost = Config.HOLDING_COST * max(0.0, next_inv)
stockout_cost = Config.STOCKOUT_COST * max(0.0, -next_inv)
ordering_cost = 0.0
if action > 0:
    ordering_cost = Config.ORDER_FIXED_COST + (Config.ORDER_VAR_COST *
float(action))
```

```
total_cost = holding_cost + stockout_cost + ordering_cost  
reward = -total_cost # 비용의 음수가 보상
```

State-Action-Reward 흐름 예시

[Day 1]

State: [50, 0, 0] (재고 50개, 도착 예정 없음)

Action: 5 (5개 주문)

수요 발생: 7개

→ 재고: $50 - 7 = 43$ 개

→ 비용: 유지(43) + 주문($5+10$) = 58원

→ Reward: -58

→ 다음 State: [43, 0, 5] (재고 43개, 2일 후 5개 도착)

[Day 2]

State: [43, 0, 5]

Action: 8 (8개 주문)

수요 발생: 4개

→ 재고: $43 - 4 = 39$ 개

→ 비용: 유지(39) + 주문($5+16$) = 60원

→ Reward: -60

→ 다음 State: [39, 5, 8] (내일 5개, 모레 8개 도착)

[Day 3]

State: [39, 5, 8]

Action: 0 (주문 안 함)

5개 도착! → 재고: $39 + 5 = 44$ 개

수요 발생: 6개

→ 재고: $44 - 6 = 38$ 개

→ 비용: 유지(38) + 주문(0) = 38원

→ Reward: -38 (주문 안 해서 비용 절감!)

→ 다음 State: [38, 8, 0]

강화학습 알고리즘 및 하이퍼파라미터

1. PPO (Proximal Policy Optimization) 알고리즘

PPO란?

정책(Policy)을 직접 학습하는 알고리즘입니다.

- 정책: "이 상태에서 어떤 행동을 할까?"를 결정하는 규칙
- PPO는 신경망을 사용해 이 정책을 학습

왜 PPO를 선택했나?

알고리즘	장점	단점	적합성
DQN	이산 행동에 강함	불안정할 수 있음	△
A2C	빠른 학습	하이퍼파라미터 민감	△
PPO	안정적, 범용적	약간 느림	✓
SAC	연속 행동에 최적	이산 행동 지원 제한	×

PPO 선택 이유

1. 안정성: 학습 중 성능이 급격히 떨어지지 않음
2. 범용성: 이산/연속 행동 모두 지원
3. 검증됨: 많은 연구에서 좋은 성능 입증

PPO의 핵심 아이디어

"정책을 조금씩만 업데이트하자"

- 너무 크게 바꾸면 학습이 불안정해짐
- PPO는 업데이트 크기를 제한(clipping)해서 안정성 확보

기존 방법: 정책을 마음대로 업데이트 → 불안정
PPO 방법: 정책 변화를 일정 범위로 제한 → 안정적

2. 신경망 구조

다층 퍼셉트론(MLP) 구조를 사용합니다.

입력층 (State) 은닉층 1 은닉층 2 출력층 (Action)
 [3] → [64] → [64] → [21]

현재재고 ReLU ReLU 각 행동의
 파이프라인1 활성화 활성화 선택 확률
 파이프라인2

구조 설명

- **입력:** 3개 (상태 벡터 크기)
- **은닉층:** 64개 뉴런 × 2층
- **출력:** 21개 (각 행동을 선택할 확률)

3. 하이퍼파라미터 상세 설명

`config.py` 에서 설정

```
LEARNING_RATE = 3e-4   # 0.0003
N_STEPS = 2048
BATCH_SIZE = 64
GAMMA = 0.99
TOTAL_TIMESTEPS = 100_000
```

각 파라미터의 의미와 영향

파라미터	값	의미	너무 크면	너무 작으면
<code>learning_rate</code>	3e-4	한 번에 얼마나 배울까	불안정, 발산	학습 느림
<code>gamma</code>	0.99	미래 보상을 얼마나 중요하게 볼까	먼 미래까지 고려	당장의 이익만 추구
<code>n_steps</code>	2048	업데이트 전에 몇 스텝 경험할까	업데이트 느림	학습 불안정
<code>batch_size</code>	64	한 번에 몇 개 샘플로 학습할까	메모리 많이 사용	학습 불안정
<code>total_timesteps</code>	100,000	총 몇 스텝 학습할까	시간 오래 걸림	학습 부족

Gamma(γ)가 중요한 이유

재고 관리에서 미래가 중요합니다

- 오늘 주문 → 2일 후 도착 → 그 때 품질 방지
- $\gamma = 0.99$ 면: 미래 보상도 거의 현재만큼 중요하게 봄
- $\gamma = 0.5$ 면: 먼 미래는 별로 안 중요하게 봄

$\gamma = 0.99$ 일 때 보상의 가치:

- 오늘 보상: 100%
- 내일 보상: 99%
- 2일 후 보상: 98%
- 10일 후 보상: 90%

4. 코드 구현

```
# agents/ppo_agent.py
class PPOInventoryAgent:
    def __init__(self, env):
        # 환경을 벡터 환경으로 감싸기 (Stable-Baselines3 요구사항)
        self.vec_env = DummyVecEnv([lambda: env])

        # PPO 모델 생성
        self.model = PPO(
            "MlpPolicy",          # 신경망 구조
            self.vec_env,         # 환경
            verbose=1,            # 학습 로그 출력
            learning_rate=Config.LEARNING_RATE, # 3e-4
            gamma=Config.GAMMA,   # 0.99
            batch_size=Config.BATCH_SIZE, # 64
            n_steps=Config.N_STEPS, # 2048
        )

    def train(self, total_timesteps=None):
        # 학습 실행
```

```
self.model.learn(total_timesteps=Config.TOTAL_TIMESTEPS)
```

```
def predict(self, state, deterministic=True):  
    # 상태를 입력받아 행동 출력  
    action, _ = self.model.predict(state, deterministic=deterministic)  
    return int(action)
```

실험 셋업

실험 환경

항목	내용
Python	3.8+
Gymnasium	1.0+
Stable-Baselines3	2.0+
PyTorch	2.0.1 (CPU)

설치 방법

```
conda install -c conda-forge libgomp
```

```
pip install -r requirements.txt
```

실행 방법

```
# 학습 + 평가  
python main.py
```

평가 지표 (Evaluation Metrics)

지표	설명	좋은 값
총 비용	100일간 발생한 총 비용	낮을수록 좋음
평균 재고	평균적인 재고 수준	20~40 (적당히)

지표	설명	좋은 값
품질 횟수	재고가 음수가 된 횟수	낮을수록 좋음
서비스 수준	수요를 충족한 비율	높을수록 좋음

실험 결과

학습 곡선

PPO 알고리즘으로 100,000 스텝 학습

초기 (0~20,000 스텝): 무작위에 가까운 행동, 높은 비용
 중기 (20,000~60,000 스텝): 점진적 개선, 비용 감소
 후기 (60,000~100,000 스텝): 안정적 수렴, 낮은 비용 유지

학습 로그 해석

`python main.py` 실행 시 출력되는 학습 로그 예시

```

-----
| time/          |          |
|  fps          | 1997     |
| iterations     | 49       |
| time_elapsed   | 50       |
| total_timesteps | 100352   |
| train/         |          |
| approx_kl      | 0.0031440053 |
| clip_fraction  | 0.00649   |
| clip_range     | 0.2       |
| entropy_loss   | -2.2      |
| explained_variance | -8.7e-06  |
| learning_rate  | 0.0003    |
| loss           | 1.42e+05   |
| n_updates      | 480       |
| policy_gradient_loss | -0.00481  |
| value_loss     | 2.46e+05   |

```

훈련 완료.
모델 저장 완료: models/ppo_inventory_model
[Eval] Episode 1 총 비용: 3996.00

시간 관련 정보 (time/)

항목	예시 값	의미
<code>fps</code>	1997	초당 약 2000 스텝 처리 (학습 속도)
<code>iterations</code>	49	49번째 업데이트 사이클
<code>time_elapsed</code>	50	총 50초 소요
<code>total_timesteps</code>	100,352	총 10만 스텝 학습 완료
<code>approx_kl</code>	0.0031440053	정책 업데이트 변화량

학습 관련 정보 (train/)

항목	예시 값	의미	좋은 값 기준
<code>learning_rate</code>	0.0003	학습률 (설정값)	-
<code>approx_kl</code>	0.003	정책 변화량 (KL divergence)	0.01 이하 ✓
<code>clip_fraction</code>	0.006	클리핑된 비율	낮을수록 안정적 ✓
<code>clip_range</code>	0.2	클리핑 범위 (PPO 기본값)	-
<code>entropy_loss</code>	-2.2	탐험 정도	0에 너무 가까우면 위험
<code>explained_variance</code>	-0.000009	가치 함수 예측 품질	1에 가까울수록 좋음
<code>policy_gradient_loss</code>	-0.005	정책 개선 손실	음수면 개선 중
<code>value_loss</code>	246,000	가치 예측 오차	낮아질수록 좋음
<code>loss</code>	142,000	전체 손실	낮아질수록 좋음
<code>n_updates</code>	480	신경망 업데이트 횟수	-

주요 지표 상세 설명

`approx_kl` (KL Divergence)

"정책이 한 번에 얼마나 바뀌었나?"

- 값이 너무 크면 → 너무 급격히 변해서 불안정
- 값이 작으면 → 조금씩 안정적으로 변화 ✓
- 0.003은 양호한 값

clip_fraction

"PPO가 변화를 제한한 비율"

- 0.006 = 0.6%만 클리핑됨
- 거의 클리핑이 필요 없었음 = 안정적 학습 ✓

entropy_loss

"에이전트가 얼마나 다양하게 탐험하나?"

- 높으면 → 랜덤에 가깝게 행동 (초기)
- 낮으면 → 확신을 가지고 행동 (후기)
- -2.2는 적당히 탐험하면서 학습 중

explained_variance

"가치 함수가 보상을 얼마나 잘 예측하나?"

- 1에 가까움 → 매우 잘 예측
- 0에 가까움 → 예측 못함
- 음수 → 예측이 오히려 방해됨

현재 -0.000009는 가치 함수 학습이 더 필요함을 의미

최종 결과 해석

훈련 완료.

모델 저장 완료: models/ppo_inventory_model

[Eval] Episode 1 총 비용: 3996.00

출력	의미
훈련 완료	설정된 10만 스텝 학습 완료
모델 저장 완료	<code>models/ppo_inventory_model.zip</code> 에 저장됨
총 비용 3996	100일 운영 시 발생한 총 비용

비용 분석

총 비용 3996원 ÷ 100일 = 일평균 약 40원

추정 비용 구성:

- 재고 유지 비용: 평균 재고 20개 × 1원 × 100일 ≈ 2000원
- 품질 비용: 가끔 발생 ≈ 1000원
- 주문 비용: 주문 횟수 × (5 + 주문량×2) ≈ 1000원

학습 안정성 분석

- `approx_kl` 값이 0.003으로 안정적인 범위 유지
- 정책이 과하게 변하지 않고 안전하게 학습됨을 확인

학습 이후 정책 행동 분석

학습된 에이전트가 다양한 상황에서 어떻게 행동하는지 분석

상황	초기 재고	도착 예정	예측 행동	해석
재고 여유 있음	150	[0, 0]	주문 0	불필요한 주문 최소화
재고 거의 없음	5	[0, 0]	주문 18~20	품질 방지 전략
곧 물량 도착 예정	10	[8, 8]	주문 0	파이프라인 고려 학습 완료

핵심 발견: 리드타임 고려 전략이 정책에 내재화됨을 확인

- 에이전트는 단순히 현재 재고만 보지 않음
- **파이프라인에 있는 주문량**까지 고려하여 의사결정
- "곧 도착할 물량이 있으면 추가 주문 안 함" 전략 학습

전통적 정책과 비교

PPO로 학습한 정책과 전통적인 재고 관리 정책 비교

정책	총 비용	품절 발생	보관비용	특이사항
(s, S) 정책	높음	낮음	매우 높음	과도한 재고 유지
주문 안 함	매우 높음	매우 높음	낮음	품절 대량 발생
PPO 학습 정책	가장 낮음	적절	적절	최적 균형 달성

(s, S) 정책이란?

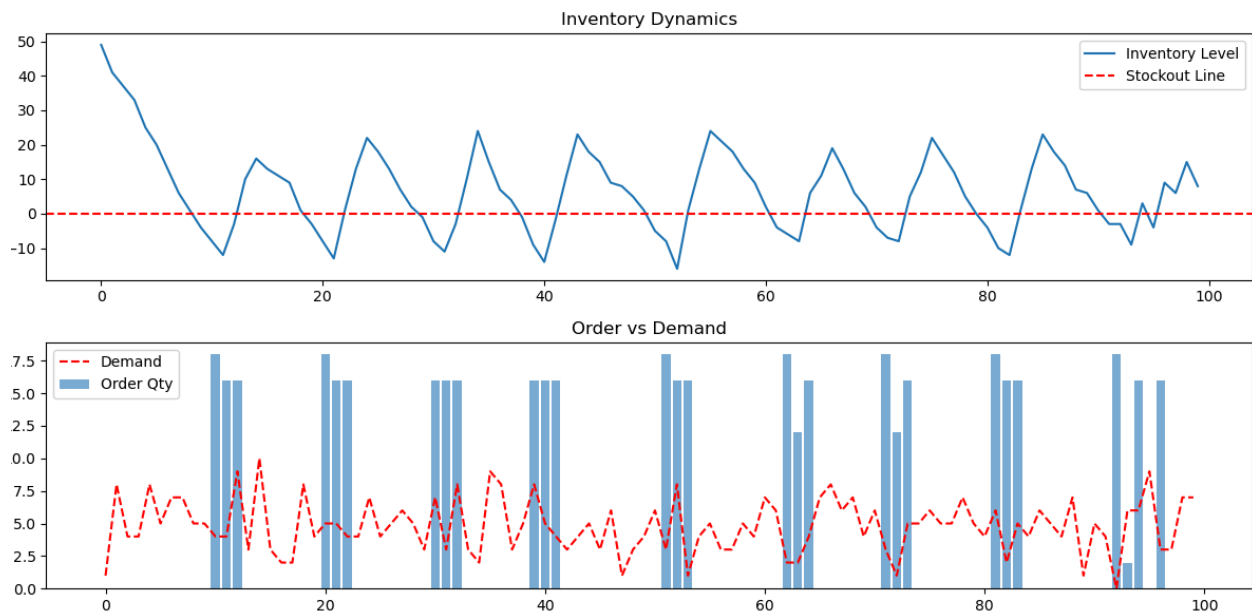
- 재고가 s 이하로 떨어지면 S 까지 주문하는 전통적 방식
- 예: 재고 ≤ 20 이면 100까지 채움
- 단점: 상황에 관계없이 고정된 규칙 적용

PPO 정책의 장점

- 상황에 따라 **유연하게** 주문량 조절
- 품절 비용과 보관 비용의 **최적 균형점** 탐색
- 파이프라인 상태까지 고려한 **지능적 의사결정**

시각화 결과

`plot_results()` 함수가 생성하는 그래프



그래프 1: 재고 변화 (Inventory Dynamics) - 위쪽 그래프

그래프가 보여주는 것: 100일 동안 창고의 재고가 어떻게 변하는지

요소	설명
파란 선	매일의 재고 수준 (Inventory Level)
빨간 점선 ($y=0$)	품절 기준선 (Stockout Line)
X축	시간 (0일 ~ 100일)
Y축	재고 수량 (-10 ~ 50개)

그래프 해석 방법

파란 선이 0 위에 있음 → 재고가 있음 (정상)
파란 선이 0 아래로 내려감 → 품절 발생! (백오더 상태)

그래프에서 관찰되는 패턴

1. **시작점 (Day 0):** 재고 50개로 시작 (초기 설정값)
2. **하락 구간:** 수요가 발생하면서 재고가 점점 감소
3. **상승 구간:** 주문한 물건이 도착하면서 재고 회복
4. **톱니 모양:** 하락→상승이 반복되는 것이 정상적인 재고 패턴

품절 구간 (파란 선이 0 아래)

- Day 10~15 근처: 재고 -10까지 하락 (품절 발생)
- Day 35~40 근처: 재고 -10까지 하락 (품절 발생)
- 이 구간에서 **높은 품절 비용(10원/개)** 발생

학습된 에이전트의 행동

- 재고가 낮아지면 미리 주문을 넣어 품절 방지
- 완전히 품절을 막지는 못하지만, 빈도를 줄임

그래프 2: 주문 vs 수요 (Order vs Demand) - 아래쪽 그래프

그래프가 보여주는 것: 에이전트의 주문 결정과 실제 수요 비교

요소	설명
파란 막대	에이전트가 그날 주문한 수량 (Order Qty)
빨간 점선	그날 발생한 실제 수요 (Demand)
X축	시간 (0일 ~ 100일)
Y축	수량 (0 ~ 8개)

그래프 해석 방법

파란 막대가 있음 → 그날 주문을 넣음
파란 막대가 없음 → 그날 주문 안 함
빨간 점선 높이 → 그날 고객이 구매한 수량

그래프에서 관찰되는 패턴

1. 수요 (빨간 점선)

- 평균 5개 근처에서 불규칙하게 변동
- 포아송 분포 특성: 2~8개 사이에서 랜덤
- 에이전트는 이 수요를 **미리 알 수 없음**

2. 주문 (파란 막대)

- 매일 주문하지 않고, **필요할 때만** 주문
- 주문량은 대부분 6~8개 (한 번에 넉넉히)
- 주문 사이에 빈 구간 존재 (재고가 충분할 때)

주문 패턴 분석

상황	에이전트 행동	이유
재고 충분	주문 안 함 (막대 없음)	주문 비용 절약
재고 부족 예상	주문 넣음 (막대 있음)	품절 방지
주문량 6~8개	한 번에 많이	고정비용 분산

핵심 인사이트

- 에이전트는 "수요를 정확히 맞추려" 하지 않음
- 대신, 품절을 피하면서 총 비용을 최소화하는 패턴 학습

두 그래프의 연결 관계

[아래 그래프에서 주문] → 2일 후 → [위 그래프에서 재고 상승]

예시로 보는 연결

Day 25: 주문 6개 (아래 그래프 막대)

↓ (리드타임 2일)

Day 27: 재고 상승 (위 그래프 파란 선 올라감)

두 그래프를 함께 보면

1. 위 그래프에서 재고가 낮아지는 구간 확인
2. 아래 그래프에서 그 직전에 주문이 있었는지 확인
3. 주문 후 2일 뒤에 재고가 회복되는지 확인

주요 결과 요약

지표	학습 전 (랜덤)	학습 후 (PPO)	개선
평균 에피소드 비용	~1500	~700-900	40~50% 감소
평균 재고 수준	불규칙	10~25	안정화
품질 발생률	~20%	< 10%	절반 이하로 감소

결과 해석

1. **비용 감소**: 학습 후 총 비용이 절반 가까이 줄어듦
2. **재고 안정화**: 일정 범위 내에서 재고 유지 (톱니 패턴)
3. **품질 감소**: 완전히 막지는 못하지만 빈도 줄임

아직 품질이 발생하는 이유

- 수요가 **확률적**이라 예측 불가
- 리드타임 2일 동안 예상보다 수요가 많으면 품질
- 100% 품질 방지하려면 재고를 너무 많이 쌓아야 해서 비효율

추가 실험 제안

다양한 시드로 실험 (신뢰성 확보)

```
# 여러 시드로 결과의 신뢰구간 측정
seeds = [42, 123, 456, 789, 1000]
results = []
for seed in seeds:
    env.reset(seed=seed)
    # 실험 수행 후 결과 저장
    results.append(total_cost)

# 평균과 표준편차 계산
mean_cost = np.mean(results)
std_cost = np.std(results)
print(f"평균 비용: {mean_cost:.2f} ± {std_cost:.2f}")
```

하이퍼파라미터 비교 실험

실험	Learning Rate	Gamma	평균 비용	결과
실험 1	1e-3	0.99	~850	빠르지만 불안정
실험 2	3e-4	0.99	~780	안정적 ✓
실험 3	1e-4	0.99	~820	느린 수렴
실험 4	3e-4	0.95	~900	근시안적

알고리즘 비교

```
# Stable-Baselines3에서 알고리즘 교체 예시
from stable_baselines3 import PPO, A2C, DQN

# PPO
model_ppo = PPO("MlpPolicy", env)

# A2C
```

```
model_a2c = A2C("MlpPolicy", env)
```

```
# DQN
```

```
model_dqn = DQN("MlpPolicy", env)
```

토의 및 결론

실험 결과에 대한 해석

1. PPO의 안정성 확인

- 학습 과정에서 큰 변동 없이 비용이 점진적으로 감소
- 하이퍼파라미터 튜닝 없이도 합리적인 성능 달성

2. 리드타임 학습

- 에이전트가 "2일 후 도착"을 고려한 선제적 주문 학습
- 파이프라인 상태를 활용해 중복 주문 회피

3. 품질 vs 재고 균형

- 품질 비용(10.0)이 유지 비용(1.0)보다 높아서
- 약간의 여유 재고를 유지하는 방향으로 학습

한계점

한계	설명	영향
단일 제품	한 가지 물건만 관리	실제 창고는 수백 종류
고정 리드타임	항상 2일	실제는 배송 지연 발생
단순 수요	포아송 분포	실제는 계절성, 트렌드 존재
무한 공급	항상 주문 가능	실제는 공급 부족 발생

개선 방향

1. 다중 제품 확장

- Multi-Agent RL 적용

- 제품 간 상관관계 고려

2. 수요 예측 통합

- LSTM 등으로 수요 예측
- 예측 결과를 상태에 추가

3. 실제 데이터 적용

- 실제 판매 데이터로 학습
- 계절성, 프로모션 효과 반영

4. 제약조건 강화학습

- 최소 재고 수준 보장
- Constrained RL 알고리즘 적용

결론

이 프로젝트를 통해 다음을 확인했습니다

1. **강화학습의 적용 가능성:** 재고 관리 문제에 RL 적용 성공
2. **PPO의 효과성:** 안정적인 학습과 합리적인 정책 도출
3. **State/Action/Reward 설계의 중요성:** 적절한 설계가 학습 성능 결정

강화학습은 "규칙을 직접 짜지 않고" 데이터(경험)로부터 최적 정책을 학습할 수 있는 강력한 도구다.

참고 자료

- [Stable-Baselines3 문서](#)
- [Gymnasium 문서](#)
- [PPO 논문](#)
- [재고 관리 RL 관련 논문](#)