МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра прикладної математики та моделювання складних систем

«До захисту допущено»

Завідувач кафедри

__________ Ігор КОПЛИК

___ __________ 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавр

зі спеціальності 113 Прикладна математика

освітньо-професійної програми «Наука про дані та моделювання складних систем»

на тему: «Використання нейромереж та штучного інтелекту для оптимізації поведінки безпілотних систем в емуляторі бойових дій в покроковому режимі.»

Здобувача групи ПМ-01 Владислава ШУЛЬЖЕНКА

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

__________ Владислав ШУЛЬЖЕНКО

Керівник доцент, кандидат наук, Аліна ДВОРНИЧЕНКО __________

Суми – 2024

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Факультет **електроніки та інформаційних технологій**

Кафедра **прикладної математики та моделювання складних систем**

Рівень вищої освіти **перший (бакалаврський)**

Галузь знань **11 «Математика та статистика»**

Спеціальність **113 «Прикладна математика»**

Освітня програма **освітньо-професійна** «Наука про дані та моделювання складних систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри ПМтаМСС
Ігор КОПЛИК ____________
«___»__________ 2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧЕВІ ВИЩОЇ ОСВІТИ

Шульженку Владиславу Вікторовичу

1. Тема роботи: Використання нейромереж та штучного інтелекту для оптимізації поведінки безпілотних систем в емуляторі бойових дій в покроковому режимі.
   Керівник роботи Дворниченко Аліна Василівна, доцент, кандидат фіз.-мат.наук, затверджено наказом по факультету ЕлІТ від «05» квітня 2024 р. № 0349-VI
2. Термін подання роботи здобувачем «31» травня 2024 р.
3. Вихідні дані по роботі: Модель штучного інтелекту, що побудована з використанням нейронної мережі.
4. Зміст розрахунково-пояснювальної записки (перелік питань, для розроблення): Емулятор Rage Of Mechs, WebSocket з’єднання, обробка даних, нейронні мережі, глибоке Q-навчання, навчання нейронної мережі, аналіз результатів.

5. Перелік графічного матеріалу: Емулятор Rage Of Mechs, програмний код реалізації з'єднання з сервером, середовища та нейронної мережі, ігрова статистика під час навчання нейронної мережі
6. Консультанти проєкту (роботи) із зазначенням розділів проєкту, що їх стосується:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «08» квітня 2024р.

КАЛЕНДАРНИЙ ПЛАН

№	**Вид робіт**	**Термін виконання**	**Примітка**
1	Аналіз середовища ігрового емулятора Rage Of Mechs, створення з'єднання з ігровим сервером для обміну інформацією в режимі реального часу	08.04.2024 р. – 13.04.2024 р.	
2	Літературний аналіз та порівняння алгоритмів, що використовують нейронні мережі з вибором найкращого для поставленої задачі	14.04.2024 р. – 24.04.2024 р.	
3	Побудова та оптимізація моделі, що використовує алгоритм глибокого Q-навчання	25.04.2024 р. – 30.05.2024 р.	

Здобувач вищої освіти ________________ Владислав ШУЛЬЖЕНКО

Керівник роботи ________________ Аліна ДВОРНИЧЕНКО

Анотація

Кваліфікаційна робота: 68 сторінок, 15 рисунків, 17 джерел, 12 формул, 1 таблиця.

Мета роботи: Створення моделі штучного інтелекту для оптимізації поведінки безпілотної системи в емуляторі бойових дій в покроковому режимі Rage Of Mechs.

Об'єкт дослідження: Емулятор бойових дій в покроковому режимі Rage Of Mechs.

Предмет дослідження: Алгоритми для створення штучного інтелекту з використанням нейронних мереж.

Методи дослідження: Літературний аналіз, аналіз архітектури середовища емулятора Rage Of Mechs, емпіричний аналіз роботи моделі штучного інтелекту, апробація результатів.

Ключові слова: МАШИННЕ НАВЧАННЯ, ШТУЧНИЙ ІНТЕЛЕКТ, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, НЕЙРОННА МЕРЕЖА, ГЛИБОКЕ Q-НАВЧАННЯ.

ЗМІСТ

ВСТУП....7

РОЗДІЛ 1. АНАЛІЗ СЕРЕДОВИЩА ЕМУЛЯТОРА RAGE OF MECHS....8

Емулятор Rage Of Mechs....8

З’єднання сервера з клієнтом....10

Попередня обробка даних....12

РОЗДІЛ 2. ОГЛЯД АЛГОРИТМІВ ДЛЯ СТВОРЕННЯ ШТУЧНОГО ІНТЕЛЕКТУ З ВИКОРИСТАННЯМ НЕЙРОННОЇ МЕРЕЖІ....14

Вимоги до алгоритму....14

Дерево пошуку Монте-Карло....15

Метод Актор-Критик....16

Глибоке Q-навчання....18

Порівняння та вибір алгоритму....20

РОЗДІЛ 3. ПОБУДОВА МОДЕЛІ ШТУЧНОГО ІНТЕЛЕКТУ З ВИКОРИСТАННЯМ ГЛИБОКОГО Q-НАВЧАННЯ....23

Підготовка середовища ігрового емулятора до роботи з алгоритмом глибокого Q-навчання....23

Вибір архітектури та побудова нейронної мережі....27

Оптимізація роботи нейронної мережі....28

Створення системи нагород....29

Навчання агентів та аналіз результатів....31

ВИСНОВОК....41

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ....42

ДОДАТОК А....44

ДОДАТОК Б....47

ДОДАТОК В..51
ДОДАТОК Г 54
ДОДАТОК Ґ 61
ДОДАТОК Д 65

ВСТУП

Сучасні технології дедалі частіше покладаються на використання штучного інтелекту. Основою великої кількості таких алгоритмів є нейронні мережі – математичні моделі, що намагаються імітувати роботу людського мозку. Такі моделі складаються з простих обчислювальних одиниць, з'єднаних між собою та організованих у шари, між якими передається інформація.

Нейронні мережі застосовуються в різних сферах, від розпізнавання зображень до створення чат-ботів та управління безпілотними системами. Для різних задач використовуються різні алгоритми, які можуть значно відрізнятися по суті, але мати спільні риси, що дозволяють їх віднести до цього типу математичних моделей.

Точна робота нейронної мережі вимагає правильного підбору та форматування вхідних даних, підбору значень гіперпараметрів, що впливають на процес навчання та обробки вихідних даних цієї моделі. Під час своєї роботи вона завдяки відгукам, які постійно їй надаються, корегує ваги нейронів, що впливають на отримані результати.

У ході цієї роботи буде створено нейронну мережу для керування мехом, тобто безпілотною системою в грі-емуляторі Rage Of Mechs, ігровий процес в якому відбувається в покроковому режимі. Будуть розглянуті технології, які використовуватимуться у процесі навчання та буде здійснено огляд різних алгоритмів для побудови штучного інтелекту в цьому середовищі. Разом з цим вхідні дані будуть оброблені та підготовлені до роботи з графічними процесорами, які підтримують CUDA, що дозволяють значно пришвидшити обчислювальні процеси.

РОЗДІЛ 1. АНАЛІЗ СЕРЕДОВИЩА ЕМУЛЯТОРА RAGE OF MECHS

Емулятор Rage Of Mechs

Починати цю роботу варто з підготовки середовища покрокового емулятора Rage Of Mechs до побудови штучного інтелекту з використанням нейронної мережі. Перший крок – це огляд та розуміння основних ігрових механік емулятора.

Rage Of Mechs являє собою покроковий емулятор бойових дій, в яких беруть участь мехи, тобто керовані роботи. Він надає широкі можливості зі створення одного або декількох мехів для ведення боїв. Користувачу надається вибір з великої кількості різних механічних модулів для корпусу меха та ще більшої кількості модулів озброєння.

Рисунок 1.1.1 – Зовнішній вигляд ангара емулятора Rage Of Mechs

Створення власних мехів відбувається в ангарі «див. рис. 1.1.1», де також відображається кількість ігрових ресурсів, що використовуються для купівлі модулів

та ремонту з поповненням боєзапасу. Тут теж можна побачити характеристики побудованих бойових машин та інформацію про їх пілотів.

Рисунок 1.1.2 – Зовнішній вигляд емулятора Rage Of Mechs у ході бою

З ангара користувач може почати бій. Після пошуку суперника у вигляді іншого гравця чи штучного інтелекту перед ним постає певна мапа, що складається з клітинок, на яких знаходяться його і ворожі механізми та інші ігрові об’єкти «див. рис. 1.1.2».

Дії суперники вчиняють у покроковому режимі по черзі під час своїх ходів. На один хід дається обмежена кількість часу. Під час нього користувач може робити різні дії, такі як пересування, постріли та захоплення ігрових об’єктів, що дають ігрові ресурси. Кожен мех має певну кількість очок для вчинення дій, що витрачаються на них та не може нічого робити після їх витрати.

Проаналізувавши ігровий процес емулятора можна зробити висновки, що для навчання моделі штучного інтелекту необхідна буде інформація про мапу, ворожих і союзних мехів, їхні модулі та ігрові об’єкти. Результатом її роботи повинна бути

команда на вчинення певної дії, такої як переміщення, стрільба з модулів та евакуація або підбір ігрового об'єкта.

З'єднання сервера з клієнтом

Після аналізу ігрового середовища та визначення необхідної для роботи та навчання моделі інформації необхідно встановити спосіб отримувати ці дані. Задля цього необхідно буде встановити з'єднання між ігровим сервером та клієнтом, на якому будуть відбуватися всі обчислювальні процеси. Воно повинне бути неперервним та забезпечувати обмін інформацією в режимі реального часу. Ця робота не вимагатиме створення або модифікації ігрового сервера, оскільки він вже був створений розробниками ігрового емулятора та задовольняє всі вимоги з отримання інформації для навчання.

Розглянемо процес встановлення з'єднання з клієнтською стороною, щоб забезпечити безперервний обмін даними. До нього будуть дві основні вимоги: робота в режимі реального часу та швидкодія. Їх виконання забезпечить швидке та ефективне навчання моделі штучного інтелекту.

Обмін даних буде відбуватися з використанням формату серіалізації Protocol Buffers. Розроблений компанією Google, він довгий час використовувався у її внутрішньому середовищі та проєктах. З тверджень компанії, ця технологія зменшує обсяги інформації, що передається та збільшує швидкість обробки даних, якщо порівнювати її роботу з альтернативою, тобто XML. Сервер емулятора вже налаштований для використання Protocol Buffers, тому будемо використовувати саме такий формат серіалізації.

Задля забезпечення роботи в режимі реального часу використаємо технологію WebSocket, яка є протоколом для обміну даними. Вона забезпечує двосторонній зв'язок між клієнтом, що виконує ненадійний код в контрольованому середовищі, і віддаленим хостом, який погодився на зв'язок з цим кодом. Протокол складається з початкового рукостискання, за яким слідує базове обрамлення повідомлень, що накладається на TCP. Мета цієї технології – надати браузерним додаткам, які потребують двостороннього зв'язку з серверами, механізм, який не покладається на

відкриття декількох HTTP-з'єднань. WebSocket спроєктовано для втілення у веббраузерах та вебсерверах, але він може бути використаним будь-яким клієнт-серверним застосунком. Ця технологія чудово підходить для використання у цій роботі.

При створенні клієнтської частини роботи буде використана високорівнева мова програмування Python. В поєднанні з вбудованими модулями та іншими бібліотеками вона дозволить створити компактний та ефективний код для виконання поставлених завдань.

Задля реалізації необхідного функціоналу створимо клас гравця з усією необхідною інформацією та деякі допоміжні методи для зв’язку з сервером у майбутньому:

```
class Player:
    def __init__(self, url, email, password):
        self.url = url
        self.email = email
        self.password = password
        self.ws_connect = None
        self.sign_up_resp = None
        self.log_in_resp = None
        self.me_resp = None
        self.in_queue = asyncio.Queue(COMMAND_QUEUE_SIZE)
```

Аутентифікація клієнтського застосунку на сервері буде відбуватися без використання Websocket з’єднання до моменту успішної авторизації користувача. Програмний код для цього буде виглядати наступним чином:

```
import asyncio
from google.protobuf.json_format import MessageToDict, ParseDict

async def auth(player):
    sign_up_response = player.sign_up()
    log_in_response = player.log_in()

    auth_token = log_in_response['data']['access_token']
    player.set_token(auth_token)

    await player.ws_init()

    try:
        await player.send_auth_message(auth_token)
    except Exception as e:
        print(f"Failed to send authentication message: {e}")

    await asyncio.sleep(1)
```

```
    message = await player.get_message(False)
    assert MessageToDict(message)['type'] == 'Auth'
```

Розглянемо створення Websocket з’єднання для обміну повідомленнями між сервером та клієнтом:

```
async def ws_init(self):
    if not self.get_token():
        raise ValueError("access token does not exist")

    ws_url = 'ws' + self.url[4:] + '/ws'
    self.ws_connect = await websockets.connect(ws_url)
    asyncio.create_task(self.handle_read())
```

У випадку успішного виконання наведеного програмного коду на сервері буде зареєстрований користувач, відбудеться його авторизація та створення з’єднання в режимі реального часу. Якщо такий користувач вже існує, то крок з реєстрацією буде пропущений. [1][2]

Попередня обробка даних

Виконавши огляд основних ігрових механік емулятора, можна визначити частки інформації, що будуть потрібні для навчання. Оскільки на кожному кроці результатом роботи штучного інтелекту буде вказівник на дію, яку найкраще вчинити в даній ігровій ситуації, то залишимо лише ті дані, що будуть необхідними для її вибору.

Користувач отримує інформацію про мапу, на якій буде відбуватися бій на його початку та про поточну ігрову ситуацію після кожної дії. Для цього необхідно зробити схожі запити на сервер та отримати на них відповідь з інформацією. Навчання та перевірка моделі буде відбуватися на одній мапі, через що можемо не використовувати інформацію про неї, оскільки незмінні дані не будуть відігравати значної ролі при визначенні того, яку дію необхідно вчинити. Розглянемо запит на отримання даних про ігрову ситуацію наразі:

```
import base64

from src.messages import turn_response_pb2
from google.protobuf.json_format import MessageToDict, ParseDict

async def get_turn_info(player):
```

```
    await player.send_turn_message()

    msg = await player.get_message(True)
    print(MessageToDict(msg)['type'])
    if MessageToDict(msg)['type'] == "Turn":
        turn_response =
turn_response_pb2.TurnResponse.FromString(base64.b64decode(MessageToDict(msg)['data']))
        return turn_response
    return None
```

Відкидання зайвих даних це лише перший крок в оптимізації роботи нейронної мережі. При створенні моделі будемо використовувати бібліотеку Pytorch, що надає широкий функціонал для створення та налаштування моделей штучного інтелекту. Одним з методів оптимізації є використання технології CUDA, що дозволяє значно пришвидшити обчислювальні процеси завдяки використанню графічних процесорів компанії Nvidia. Бібліотека Pytorch робить її використання можливим та простим, але при цьому вимагатиме зберігання даних у спеціальних структурах – тензорах. Тензор – це багатовимірний масив. Більш формально, тензор N-го порядку – це елемент тензорного добутку N векторних просторів, кожен з яких має власну систему координат.

Для навчання будуть використані лише дані про мехів та їх стан і про ігрові предмети. Створимо програмний код для зберігання отриманої інформації про ігрову ситуацію у вигляді тензорів:

```
def create_state_tensor(turn_info):
    drop_tensor = create_drop_tensor(turn_info)
    player_mech_tensor = create_player_mech_tensor(turn_info)
    enemy_mech_tensor = create_enemy_mech_tensor(turn_info)

    concatenated_tensor = torch.cat((drop_tensor, player_mech_tensor, enemy_mech_tensor))

    return concatenated_tensor.float()
```

Збережені дані мають невеликий обсяг та зберігаються у вигляді тензорів, що значно пришвидшить роботу та навчання нейронної мережі. [3][4][5]

РОЗДІЛ 2. ОГЛЯД АЛГОРИТМІВ ДЛЯ СТВОРЕННЯ ШТУЧНОГО ІНТЕЛЕКТУ З ВИКОРИСТАННЯМ НЕЙРОННОЇ МЕРЕЖІ

Вимоги до алгоритму

Як вже раніше було зазначено, алгоритм для побудови штучного інтелекту повинен задовольняти наступні вимоги: швидкодія, можливість генерувати оцінку для великої кількості дій з невеликої кількості наявної інформації, здатність ефективно навчатися та оптимізувати свою поведінку та при його побудові повинні бути використані нейронні мережі. Перша вимога буде частково виконуватися завдяки оптимізаційним заходам, вжитим раніше. Наступним кроком буде літературний аналіз, метою якого є визначення алгоритмів, що підходять під зазначені параметри.

Для розв’язання такої задачі необхідно використовувати алгоритм машинного навчання з підкріпленням. Він імітує процес навчання методом спроб і помилок, який люди використовують для досягнення своїх цілей. Дії моделі, які працюють на досягнення кінцевої мети, підкріплюються, тоді як дії, які відвертають від мети, ігноруються. Алгоритми навчання з підкріпленням використовують парадигму заохочення і покарання при обробці даних. Вони вчаться на основі зворотного зв'язку від кожної дії й самостійно знаходять найкращі шляхи обробки для досягнення кінцевого результату. Алгоритми також здатні на відкладене винагородження. Найкраща загальна стратегія може вимагати короткострокових жертв, тому найкращий підхід, який вони знаходять, може включати певні покарання або відступ на цьому шляху.

Після невеликого дослідження було виявлено три основні алгоритми, що можуть бути використані для створення моделі штучного інтелекту. Першим є дерево пошуку Монте-Карло, який можна модифікувати задля використання нейронної мережі в цілях оптимізації. Другим є метод Актор-Критик (A2C), що складається з

двох агентів, перша з яких відповідає за вибір дій, а друга за оцінку її роботи. Третім є алгоритм глибокого Q-навчання, що оцінює кожну з можливих дій з використанням нейронної мережі для генерації Q-значень, які є оцінками цінності цих дій.

Використання будь-якого з цих методів може бути дієвим для ігрового середовища емулятора. Попри це, вони вимагають детального огляду та порівняння задля вибору оптимального у плані швидкодії, простоти реалізації та точності результатів алгоритму. [6][7][8][9][11]

Дерево пошуку Монте-Карло

Дерево пошуку Монте-Карло (далі MCTS) – це евристичний алгоритм пошуку, який використовується в галузі штучного інтелекту. В ньому комбінуються класичні реалізації дерева пошуку з алгоритмами машинного навчання, такими як підкріплене навчання та нейронні мережі.

MCTS поєднує в собі стандарти стратегій Монте-Карло, які покладаються на випадкову вибірку та статистичну оцінку з методами пошуку на основі дерев. На відміну від традиційних алгоритмів пошуку, які покладаються на дослідження всієї області пошуку, він спеціалізується на вибірці та дослідженні лише перспективних ділянок області пошуку. MCTS виконує випадкову вибірку у вигляді симуляцій і зберігає статистику дій, щоб зробити більш обґрунтований вибір на кожній наступній ітерації.

Вибір дочірніх вузлів поточного вузла відбувається, як правило, з використанням алгоритму верхньої довірчої межі, що використовує формулу:

$$UCB = Q_i + C\sqrt{\frac{\ln N}{n_i}}, \tag{2.2.1}$$

де Q_i – оціночне значення дочірнього вузла, N – кількість відвідувань батьківського вузла, n_i – кількість відвідувань дочірнього вузла, C – константа, яка контролює компроміс між дослідженням та використанням вже відомих шляхів.

MCTS є інтуїтивно зрозумілим і легким для впровадження. Він вчиться з симуляцій, що дозволяє йому адаптуватися та вдосконалюватися. При цьому всьому

з ростом дерева збільшується потреба в оперативній пам’яті та існує ризик, що алгоритм може не відвідати достатньо разів кожен вузол, щоб зрозуміти його результат. Також для ефективного визначення найкращого шляху потрібно багато ітерацій, що може бути повільним.

Нейронні мережі можуть бути використані для покращення рішень, знайдених за допомогою MCTS, шляхом навчання на оптимальних стратегіях та коригування рішень з урахуванням вартості транзакцій. Нейронна мережа тренується на приблизних рішеннях, отриманих з дерева пошуку, щоб знайти глобальний оптимум. [8][9][10]

Метод Актор-Критик

Алгоритм Актор-Критик для навчання з підкріпленням використовує дві мережі: актора, який вирішує, яку дію виконати, та критика, який оцінює дію актора, вказуючи, наскільки добре вона була виконана. Актор отримує на вході стан, а на виході повертає найкращу дію для цього стану. Він контролює поведінку агента, вивчаючи оптимальну поведінку. Критик, з іншого боку, оцінює дію, обчислюючи функцію цінності цієї дії. Ці дві моделі беруть участь у грі, де вони обидві стають кращими у своїй ролі з плином часу. В результаті загальна архітектура навчиться грати в гру більш ефективно, ніж ці два методи окремо.

Навчання обох мереж відбувається окремо і використовує градієнтне сходження для оновлення ваг. В основі алгоритму лежить ідея, що градієнт функції дає напрямок найкрутішого підйому. Ітеративно рухаючись по цій траєкторії, алгоритм намагається знайти найвищу точку на поверхні функції, яка відповідає її максимальному значенню. Вираз градієнта політики для алгоритму REINFORCE має наступний вигляд:

$$\nabla J(\theta) = \nabla E_{\pi_\theta}[R(\tau)] = E_{\pi_\theta}\left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)\right), \quad (2.3.1)$$

де E_{π_θ} – математичне сподівання за розподілом ймовірностей траєкторії, $R(\tau)$ – результат траєкторії τ, який є сумою отриманих нагород.

Результат траєкторії τ можна записати наступним чином:

$$R(\tau) = \sum_{t=0}^{T-1} r_t, \quad (2.3.2)$$

де r_t – отримана нагорода.

Після заміни значення траєкторії на записане раніше значення «див. формулу 2.3.2» та додавання базової функції, вираз градієнта «див. формулу 2.3.1» буде мати наступний вигляд:

$$\nabla J(\theta) = E_{\pi_\theta}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t=0}^{T-1} r_t' - b(s_t)\right)], \quad (2.3.3)$$

де $b(s_t)$ – значення базової функції.

Ми можемо назвати винагороду та базове значення функцією переваги. Її можна записати наступним чином:

$$\mathrm{A}(s_t, a_t) = \sum_{t=0}^{T-1} r_t' - b(s_t) \quad (2.3.4)$$

При використанні методу Актор-Критик базове значення буде функцією значення поточного стану, тому запишемо модифіковану версію функції переваги «див. формулу 2.3.4»:

$$\mathrm{A}(s_t, a_t) = r(s_t, a_t) + V_{\pi_\theta}(s_t + 1) - V_{\pi_\theta}(s_t), \quad (2.3.5)$$

де s_t – стан в поточний момент навчання, a_t – дія, обрана агентом в поточний момент навчання, $V_{\pi_\theta}(s_t)$ – функція значення стану.

Замість того, щоб змушувати критика вивчати Q-значення, ми змушуємо його вивчати значення переваг. Таким чином, оцінка дії ґрунтується не лише на тому, наскільки вона хороша, але й на тому, наскільки вона може бути кращою. Перевага функції переваги полягає в тому, що вона зменшує високу дисперсію мереж і стабілізує модель.

Алгоритм може бути видозмінений задля того, щоб працювати в асинхронному режимі, що значно пришвидшує його роботу. Агенти навчаються паралельно і періодично оновлюють глобальну мережу, в якій зберігаються спільні параметри. Оновлення відбуваються не одночасно, і саме звідси походить асинхронність. Після кожного оновлення агенти скидають свої параметри до параметрів глобальної мережі й продовжують незалежне дослідження та навчання, поки не оновлять себе знову. Основним недоліком асинхронності є те, що деякі агенти будуть грати зі старою

версією параметрів. Звичайно, оновлення може відбуватися не асинхронно, а одночасно. В такому випадку ми маємо покращену версію звичайного алгоритму з декількома агентами замість одного, який зачекає, поки всі агенти завершать свій сегмент, а потім оновить глобальні ваги мережі й перезавантажить всіх агентів.

Використовуючи дві моделі для взаємодії, метод значно покращує процес навчання, забезпечуючи краще узгодження стратегій та оцінок. Проте така складність може ускладнювати реалізацію порівняно з простішими алгоритмами, що є однією з його слабких сторін. Додатково, необхідність частого оновлення ваг додає до складності процесу навчання, що може вимагати більше ресурсів та часу. Попри це, комбінований підхід демонструє високу ефективність, роблячи його гарним вибором для складних завдань. [7][12]

Глибоке Q-навчання

Q-навчання є одним з основних підходів до навчання без моделі з підкріпленням. Воно зосереджується на визначенні того, наскільки корисна певна дія для отримання майбутньої винагороди в стані за певною політикою. Процес Q-навчання включає зберігання та оновлення Q-значень у Q-таблиці, яка має розміри, що відповідають кількості дій та станів у середовищі.

Початковий вибір дії агентом є випадковим, але з часом, коли Q-таблиця починає оновлюватися, вибір дії базується на максимальному Q-значенні для стану. Ці значення оновлюються за допомогою рівняння Беллмана, яке враховує різницю між поточним Q-значенням для пари стан-дія та оцінкою оптимального майбутнього Q-значення:

$$V(s) = \max_a (R(s,a) + \gamma \sum_{s'} P(s'|s,a) V(s')), \tag{2.4.1}$$

де $V(s)$ – це функція цінності стану, яка вимірює очікувану максимальну суму винагород, починаючи з цього стану, $R(s,a)$ – нагорода, яку агент отримує при виконанні дії a в стані s, γ – це коефіцієнт, який визначає важливість майбутніх

винагород порівняно з негайними винагородами, $P(s'|s,a)$ – ймовірність переходу в стан s' зі стану s при вчиненні дії a.

Для вибору дії, що вчинить агент часто використовують епсилон-жадібну стратегію. Вона спрямована на балансування між дослідженням і експлуатацією вже відомих стратегій, де агент з імовірністю ε обирає випадкову дію, а з імовірністю $1-\varepsilon$ обирає дію, яка максимізує Q-значення.

Функція трат використовується при навчанні агента для мінімізації різниці між поточним і цільовим Q-значенням. Ця різниця може обчислюватися за допомогою середньоквадратичної похибки. В такому випадку функція втрат визначається з використанням рівняння Беллмана «див. формулу 2.4.1» наступним чином:

$$MSE = \frac{1}{n}\sum_{y=1}^{n}(\max_{a}(R(s,a)+\gamma Q(s',a')) - Q(s,a))^2, \quad (2.4.2)$$

де $\max_{a}(R(s,a)+\gamma Q(s',a'))$ – цільове Q-значення, $Q(s,a)$ – поточне Q-значення.

При використанні алгоритму Q-навчання для розв'язання задач машинного навчання у середовищі з великою кількістю станів та можливих дій виникатиме очевидна проблема. Ускладнення роботи агента буде викликане тим, що він повинен оцінювати Q-значення для кожної можливої дії в кожному стані. Це призводить до повільного навчання та великих розмірів Q-таблиці, що унеможливлює або робить її зберігання дуже складним.

Глибоке Q-навчання є одним з фундаментальних підходів у сфері навчання з підкріпленням. Цей алгоритм побудований на основі його звичайної версії з ключовою відмінністю, а саме з додаванням нейронної мережі, яка генерує Q-значення для пари стан-дія. Замість зберігання цих значень у таблиці мережа навчається визначати найкращу дію для ігрової ситуації, базуючись на вже отриманому досвіді.

Оновлення ваг нейронної мережі під час глибокого Q-навчання так само відбувається за допомогою рівняння Беллмана «див. формулу 2.4.1» з використанням техніки досвіду повторного відтворення, яка допомагає стабілізувати навчання та підвищити ефективність алгоритму. Агент збирає досвід під час взаємодії з

середовищем. Кожна взаємодія агента зі середовищем записується як перехід у вигляді чотирьох елементів: поточний стан, дія, винагорода та наступний стан. Усі ці переходи зберігаються у великій пам'яті, яка називається буфером повторного відтворення. Цей буфер має обмежений розмір, і коли він заповнюється, старіші переходи замінюються новішими. Під час навчання замість використання останнього переходу агент випадково вибирає мінінабір збережених переходів з буфера. Це дозволяє алгоритму краще узагальнювати та знижує кореляцію між послідовними переходами.

З метою оптимізації при роботі глибокого Q-навчання створюють дві нейронні мережі, а саме основну та цільову. Використання двох мереж допомагає уникнути нестабільності, яка може виникнути через постійні зміни в оцінках Q-значень. Основна мережа використовується для вибору дій та оновлення ваг під час навчання. Вона безпосередньо взаємодіє з середовищем та оновлюється кожні кілька кроків. Цільова мережа використовується для оцінки Q-значень під час оновлення основної мережі. Її ваги оновлюються менш часто, що забезпечує стабільність цільових значень. Цільова мережа допомагає відокремити цільові Q-значення від оцінок, які використовуються основною мережею, що зменшує кореляцію між цільовими та прогнозованими Q-значеннями.

Глибоке Q-навчання дозволяє обробляти великі простори станів та дій. Використання методів оптимізації забезпечує стабільність алгоритму. Він є простим у реалізації, але вимагатиме точного налаштування гіперпараметрів та може мати труднощі з завданнями, де потрібно розуміння довготривалих залежностей. [6][13]

Порівняння та вибір алгоритму

Для розв’язання цієї задачі необхідно обрати лише один алгоритм, тому порівняємо слабкі та сильні сторони кожного з них і розглянемо можливості їх практичного застосування.

Дерево пошуку Монте-Карло з використанням нейронних мереж має наступні переваги: може бути ефективним для завдань, де потрібне глибоке довгострокове

планування, оскільки він базується на пошуку за допомогою дерев рішень, присутня можливість використання різних функцій оцінки для кращого апроксимування вартості вузлів дерева. При цьому присутні наступні недоліки: велика обчислювальна складність, особливо при роботі з великими деревами пошуку та обмежена ефективність у великих та неперервних просторах станів, де складність зростає експоненційно.

Метод Актор-Критик ефективний у навчанні стратегій великих просторів дій та його асинхронна версія, завдяки можливості паралельного навчання декількох агентів одночасно, збільшує швидкість навчання. Слабкими сторонами цього алгоритму є велика чутливість до гіперпараметрів та у випадку з асинхронною версією складність в реалізації та великі вимоги до обчислювальних ресурсів.

Глибоке Q-навчання приваблює відносною простотою його реалізації та надає можливість ефективно навчати стратегії, які базуються на довгострокових наслідках. До деяких проблем можна віднести проблеми зі стійкістю в навчанні та неоптимальність у виборі дій у великих просторах.

Проаналізувавши середовище та всі три алгоритми я вирішив реалізувати алгоритм глибокого Q-навчання з наступних причин:

- Цей алгоритм є гарним вибором, оскільки він побудований на базі одного з основних підходів до навчання без моделі з підкріпленням та буде слугувати гарним вибором для демонстрації можливостей нейронних мереж у розв’язанні таких задач.
- Простота в реалізації цього алгоритму дозволить витратити більше часу на заходи з оптимізації його роботи.
- Недоліки алгоритму вирішуються його модифікуванням, таким як додавання цільової мережі та буферу повторного відтворення.
- Алгоритм надає гарні результати при побудові довгострокових стратегій, що є важливим в ігровому середовищі емулятора.

Незважаючи на все раніше сказане, алгоритм вимагатиме великої кількості оптимізаційних заходів та точного підбору гіперпараметрів для ефективного навчання та роботи. [6][7][8][9][12][13]

РОЗДІЛ 3. ПОБУДОВА МОДЕЛІ ШТУЧНОГО ІНТЕЛЕКТУ З ВИКОРИСТАННЯМ ГЛИБОКОГО Q-НАВЧАННЯ

Підготовка середовища ігрового емулятора до роботи з алгоритмом глибокого Q-навчання

Раніше в цій роботі вже був розглянутий ігровий процес емулятора Rage Of Mechs та технології, що дозволяють встановити з'єднання з сервером для обміну інформацією з клієнтом. Наступним кроком у підготовці роботи моделі штучного інтелекту буде створення засобів взаємодії з ігровим середовищем для безперервного навчання. Серед вимог до нього варто зазначити можливість одночасної гри двох агентів один проти одного, можливість визначати можливі дії для кожного з агентів та можливість почергово вчиняти дії під час свого ходу для кожного з агентів.

В першу чергу створимо клас для симуляції ігрового середовища:

```
class GameEnvironment:
    def __init__(self, player1, player2, device):
        self.player1 = player1
        self.player2 = player2
        self.device = device
        self.steps = 0
        self.is_player1_turn = None

        self.turn_info1 = None
        self.state_tensor1 = None

        self.turn_info2 = None
        self.state_tensor2 = None

        self.actions_dictionary1 = None
        self.actions_dictionary2 = None
```

У цьому фрагменті програмного коду наведено ініціалізацію класу ігрового середовища з необхідними параметрами: гравці, графічний процесор для використання CUDA, кількість кроків під час кожного бою, вказівник для визначення того, чий зараз відбувається хід, інформація про ігрову ситуацію у вигляді об'єкта, що повертає сервер та у вигляді тензора для навчання нейронної мережі й словники, що відображають можливість вчинення певних дій. Також його можна доповнити

деякими параметрами для збору ігрової статистики та аналізу процесу навчання. Для тензорів з інформацією про ігровий стан залишимо наступну інформацію: позиція та прогрес евакуації ігрового об'єкта, положення, очки дій (далі ОД) та стан модулів меха гравця і положення та стан модулів ворожого меха, що в результаті збільшує його розмір до 52.

Визначимо, яку максимальну кількість дій може вчиняти агент в середовищі. Для цього проаналізуємо інформацію, що надає сервер. Максимальна кількість ОД у меха становить 20, це число обмежує його дії протягом одного ходу. З інформації про мапу можна дізнатися, що мінімальна вартість пересування на одну клітинку становить 2 ОД, що зрештою дозволяє визначити кількість можливих рухів меха, яка становить 440. Оскільки він може рухатися на 10 клітинок в кожну сторону та гра дозволяє вчиняти рух діагонально, то це число визначається як площа квадрата зі сторонами розміром 21 за виключенням центральної клітинки, на якій вже знаходиться мех. З обраною мною конфігурацією меха йому доступна лише дія для евакуації ігрового об'єкта та він має 21 модуль. По кожному з них можна здійснити прицільний постріл з гармати і здійснити два різних постріли по ворожому меху: постріл чергою з гармати та ракетний постріл. Це дає можливість зробити ще 24 дії, що зрештою обмежує їх максимальну кількість числом 464.

Створимо словник, що буде вказувати чи може агент виконати певну дію в даному ігровому стані:

```
    def create_actions_dictionary(self, turn_info):
        movement_actions = {(dx, dy): 0 for dx in range(-10, 11) for dy in range(-10, 11) if (dx != 0
or dy != 0)}

        attack_actions = {action: 0 for action in ['rocket_shot', 'burst_shot']}

        module_ids = [module.moduleId for module in turn_info.playerMechs[0].modules]
        aimed_actions = {f'aimed_{module_id}': 0 for module_id in module_ids}

        interaction_actions = {action: 0 for action in ['evacuate_drop']}

        actions_dictionary = {**movement_actions, **attack_actions, **aimed_actions,
**interaction_actions}
        actions_dictionary = {k: torch.tensor(v, device=self.device) for k, v in
actions_dictionary.items()}

        return actions_dictionary
```

Для кожної дії створюється пара ключ-значення, де ключ зберігає інформацію про дію, а значення вказує на можливість її виконання. Значення цієї пари буде визначатися наступним чином:

```
    def check_valid(self, turn_info, actions_dictionary):
        player_mech_position = (turn_info.playerMechs[0].position.X if
hasattr(turn_info.playerMechs[0].position, 'X') else 0,
                                turn_info.playerMechs[0].position.Y if
hasattr(turn_info.playerMechs[0].position, 'Y') else 0)
        turn_points = turn_info.playerMechs[0].turnPoints

        for move in turn_info.turn.moves:
            move_position = (move.position.X if hasattr(move.position, 'X') else 0,
                             move.position.Y if hasattr(move.position, 'Y') else 0)
            delta_x = move_position[0] - player_mech_position[0]
            delta_y = move_position[1] - player_mech_position[1]

            if (delta_x, delta_y) in actions_dictionary and turn_points >= move.cost:
                actions_dictionary[(delta_x, delta_y)] = torch.tensor(1, device=self.device)
```

Наведений фрагмент програмного коду визначає можливість пересування на певну відстань. Значення delta_x та delta_y визначають не кінцеві координати, а зміщення меха відносно його початкових координат. Інформація про це береться з даних, що надаються сервером та у випадку можливості виконання цієї дії, їй присвоюється значення 1 у словнику. Аналогічний програмний код був створений для пострілів та евакуації ігрового об’єкта.

Середовище повинно мати можливість оновлюватися перед початком бою та виконувати дії. Створимо метод, який підготує мехів до бою та оновить середовище:

```
    async def reset(self):
        await asyncio.sleep(0.5)

        await self.player1.send_ammo_load_all_message()
        await self.player2.send_ammo_load_all_message()

        await self.player1.send_repair_all_message()
        await self.player2.send_repair_all_message()

        await asyncio.sleep(0.5)

        await start_battle(self.player1, self.player2)
```

```
        await self.update_states(True)

        self.steps = 0
        self.is_player1_turn = self.turn_info1.isPlayerTurn
```

Цей метод виконує підготовчі заходи та починає битву, після чого оновлює інформацію про стан та визначає чий зараз хід.

Створимо метод, що дозволить агенту вчиняти обрану дію та переводити ігрове середовище в наступний стан:

```
    async def step(self, player, action_key):
        ep_done1, ep_done2 = False, False
        state1 = self.turn_info1
        state2 = self.turn_info2

        if player == self.player1:
            await self.make_action(self.turn_info1, player, action_key)
            self.steps += 1
            ep_done1, ep_done2 = await self.update_states(True)
        else:
            await self.make_action(self.turn_info2, player, action_key)
            self.steps += 1
            ep_done1, ep_done2 = await self.update_states(True)

        if self.steps >= 300:
            await player.send_battle_end()
            await self.update_states(True)
            ep_done1, ep_done2 = [True, 1], [True, 1]

        reward1 = self.calculate_reward(ep_done1, state1, self.turn_info1)
        reward2 = self.calculate_reward(ep_done2, state2, self.turn_info2)

        return reward1, reward2, ep_done1, ep_done2
```

Цей метод вчиняє обрану дію для обраного гравця. Далі він оновлює стан середовища та перевіряє чи закінчила обрана дія битва. На основі отриманого стану розраховується нагорода. Те, яким чином розраховується нагорода та виконується вибір дії буде розглянуто при огляді структури та методів оптимізації побудованої моделі. Для того, щоб надіслати запит на вчинення необхідної дії в методі make_action відбувається перевірка наданого ключа та вибір відповідного запиту. Фрагмент програмного коду цього методу для переміщення меха виглядає наступним чином:

```
    async def make_action(self, turn_info, player, action_key):
        player_mech = turn_info.playerMechs[0]

        if isinstance(action_key, tuple):
            await player.send_turn_move_message(player_mech.id,
```

```
                                                        (player_mech.position.X if
hasattr(player_mech.position, 'X') else 0) + action_key[0],
                                                        (player_mech.position.Y if
hasattr(player_mech.position, 'Y') else 0) + action_key[1])
```

Вибір архітектури та побудова нейронної мережі

При побудові нейронної мережі будуть використані три повноз’єднані шари, тобто такі, де кожен нейрон або вузол з попереднього шару з'єднаний з кожним нейроном поточного шару. Вони є універсальними апроксиматорами, тобто можуть апроксимувати будь-яку неперервну функцію за наявності достатньої кількості нейронів і шарів. Ця властивість робить їх придатними для апроксимації складної функції Q-значення в навчанні з підкріпленням. Повноз’єднані шари прості в реалізації й добре зрозумілі в контексті нейронних мереж. Ця простота робить їх гарною відправною точкою для розв’язання багатьох проблем.

Шари нейронної мережі вимагають активаційну функцію, яка визначає вихідний сигнал нейрона. В цій роботі буде використаний зрізаний лінійний вузол, який повертає 0, якщо отримує від'ємне значення на вході, але для будь-якого додатного значення повертає це значення назад. Його можна описати наступною формулою:

$$f(x) = \max(0, x) \tag{3.2.1}$$

Створимо програмний код для реалізації нейронної мережі з використанням бібліотеки Pytorch:

```
class DQN(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Використання повноз’єднаних шарів зі зрізаним лінійним вузлом у ролі активаційної функції є базовим та ефективним підходом до розв’язання задач

глибокого Q-навчання у зв'язку з його простотою та універсальністю. При побудові моделі штучного інтелекту створимо три шари, оскільки ігрове середовище емулятора має велику кількість станів та можливих дій і це вимагатиме більшої комплексності нейронної мережі для ефективного дослідження та опрацювання наявних дій для наявних станів. Три повноз'єднаних шари зберігають баланс між ефективністю та швидкодією роботи мережі. [14][15][16]

Оптимізація роботи нейронної мережі

Для оптимізації нейронної мережі будемо використовувати наступні методи та технології:

- **Буфер відтворення**

З використанням двобічної черги, тобто структури даних, елементи якої можуть додаватися як на початок, так і в кінець, реалізуємо буфер відтворення. Він буде мати фіксований розмір та зберігати досвіди у вигляді набору: стан, дія, нагорода, наступний стан. При навчанні для оптимізації роботи нейронної мережі замість поточного набору для оновлення ваг буде випадковим чином обиратися пакет досвіду, тобто декілька наборів, що є важливим для розриву кореляції між послідовними досвідами та стабілізації навчання.

- **Оптимізатор AdamW з функцією втрат середньоквадратичної похибки**

Функція втрат середньоквадратичної похибки використовується для обчислення градієнтів, які вказують на напрямок, у якому необхідно оновлювати ваги моделі для зменшення помилки. Оптимізатор використовує ці градієнти для оновлення ваг мережі. Розглянемо роботу оптимізатора Adam:

$$x(t) = x(t-1) - \alpha \frac{m(t)}{\sqrt{v(t)} + \varepsilon}, \quad (3.3.1)$$

де $x(t)$ – це ваги мережі, α – швидкість навчання нейронної мережі, $m(t)$ – експоненційно згладжене середнє градієнтів, $v(t)$ – експоненційно згладжене середнє квадратів градієнтів, ε – дуже маленьке число, щоб уникнути ділення на нуль.

Adam використовується, бо він ефективний у багатьох випадках і зазвичай швидше збігається до оптимального рішення, ніж інші алгоритми, хоч і часто показує гіршу узагальненість. Вдосконалена версія Adam, відома як AdamW, виправляє цю проблему, використовуючи ваговий розпад після контролю розміру кроку для кожного параметра, що призводить до кращої узагальненості моделей:

$$x(t) = x(t-1) - \alpha \frac{m(t)}{\sqrt{v(t)+\varepsilon}} - \alpha w' x(t-1), \tag{3.3.2}$$

де w' – швидкість зменшення ваг нейронної мережі. [17]

- **Епсилон-жадібна політика**

Епсилон-жадібна політика буде використана у цій роботі для оптимізації поведінки моделі та балансування між дослідженням і використанням вже відомих стратегій. На початку навчання агенти матимуть високу ймовірність вибору випадкової дії. З часом ця ймовірність буде зменшуватися для переходу від дослідження до розвитку та оптимізації вже відомо вдалих стратегій, що підвищує кінцеву продуктивність навченої моделі.

- **Цільова мережа**

Основна ідея використання цільової мережі полягає у тому, щоб уникнути ситуації, коли поточна Q-мережа намагається передбачити свої ж власні зміни. Це може призвести до великої нестабільності і коливань в процесі навчання. Цільова мережа допомагає цьому запобігти, оскільки її параметри оновлюються рідше. Параметри цільової мережі копіюються з основної Q-мережі періодично через певну кількість кроків навчання. Це допомагає зберегти стабільність цільових значень на деякий час і дозволяє основній мережі краще зближуватися до стабільного рішення.

Створення системи нагород

Оскільки алгоритм глибокого Q-навчання належить до категорії алгоритмів навчання з підкріпленням, то для оптимізації власних дій агенти вимагатимуть певного відгуку від середовища. Побудуємо систему нагород для того, щоб надавати

цей відгук. Значення нагород відносні один до одного, що забезпечує збалансованість у грі.

В першу чергу варто визначитися з нагородою, яку модель отримує за завершення бою. В цій роботі будемо розглядати 4 різних варіанти завершення бою: перемога першого агента, перемога першого агента, нічия та штучне завершення бою. Для будь-якого з перших двох випадків один мех повинен знищити іншого. Для третього випадку у них повинні закінчитися боєприпаси або у них повинні бути пошкоджені модулі для стрільби, що унеможливить заподіяння шкоди один одному. Для останнього випадку створимо штучне обмеження по тривалості бою у вигляді 400 дій для обох мехів, після чого двом агентам буде зарахована поразка, що спонукатиме їх шукати шляхи для швидшого закінчення бою. Перемога заслуговує на значну нагороду, яка буде становити 80 одиниць, щоб спонукати до пошуку та розвитку переможних стратегій. Поразка має великий штраф у розмірі -120 одиниць, щоб не підштовхувати агента до програшів. Нічия в свою чергу теж не є бажаною, хоч і не є найгіршим результатом, тому для неї призначимо менший штраф, а саме -40 одиниць.

Створимо додаткові нагороди та штрафи для ігрових дій, щоб спонукати агентів швидше отримувати переможні результати. Ігровий об’єкт, що містить цінні ресурси не є умовою для перемоги, але його підбір також повинен нагороджуватися. За кожен хід, під час якого виконується евакуація цього об’єкту агент отримує нагороду в розмірі 2 одиниць. Водночас супротивник отримує аналогічний штраф, тобто -2 одиниці. Разом з цим необхідно спонукати мехів наближатися до об’єкта. Для цього введемо наступний штраф для агента:

$$R_d = \begin{cases} -1, & d > 12 \\ -1 + (12 - d), & 1 < d < 12, \\ 0, & d \leq 1 \end{cases} \tag{3.4.1}$$

де d – це відстань від меха до ігрового об’єкта.

Щоб агенти не уникали бою будемо надавати невелику нагороду у розмірі 1.5 одиниць, коли ворожий мех знаходиться у полі зору. За заподіяння шкоди також передбачена нагорода, а саме 10 одиниць, оскільки лише ці дії напряму приведуть до

перемоги. За отримання шкоди додамо штраф у розмірі -5 одиниць, щоб агенти намагалися уникати додаткових пошкоджень, бо це може призвести до поразки.

Високі нагороди та штрафи за кінцевий результат епізоду створюють довгострокову мотивацію, тоді як менші нагороди та штрафи за проміжні дії забезпечують короткострокове регулювання поведінки агента. Вони розподіляються так, щоб агент максимально прагнув досягнення цілей, мінімізації втрат та ефективної взаємодії з ворогами.

Навчання агентів та аналіз результатів

В процесі навчання будемо тренувати двох агентів один проти одного. При використанні такого підходу, рівень складності природно збільшується, оскільки кожен агент покращує свою стратегію у відповідь на вдосконалення іншого агента. Це призводить до еволюційного процесу, де агенти стають все сильнішими з часом. Процес навчання вимагатиме правильного та точного підбору гіперпараметрів «див. Додаток А». Оберемо декілька наборів цих параметрів та проведемо процеси навчання для кожного з них «див. табл. 3.5.1». Розміри шарів нейронної мережі, кількість епох, частота оновлення цільової мережі й параметри буфера відтворення завжди будуть залишатися незмінними: вхідний шар розміром 52, приховані шари з розмірами 128 та 256 і вихідний шар для 464 дій, 100 епох, оновлення мережі раз на 25 дій, буфер відтворення розміром 10000 з використанням мініпакету, що містить 32 набори даних.

Таблиця 3.5.1

Гіперпараметр	Набір 1	Набір 2	Набір 3
lr	0.005	0.01	0.002
gamma	0.99	0.99	0.99
epsilon	0.7	0.9	1.0
epsilon_decay	0.995	0.999	0.999
epsilon_min	0.01	0.01	0.01

weight_decay	0.01	0.05	0.01

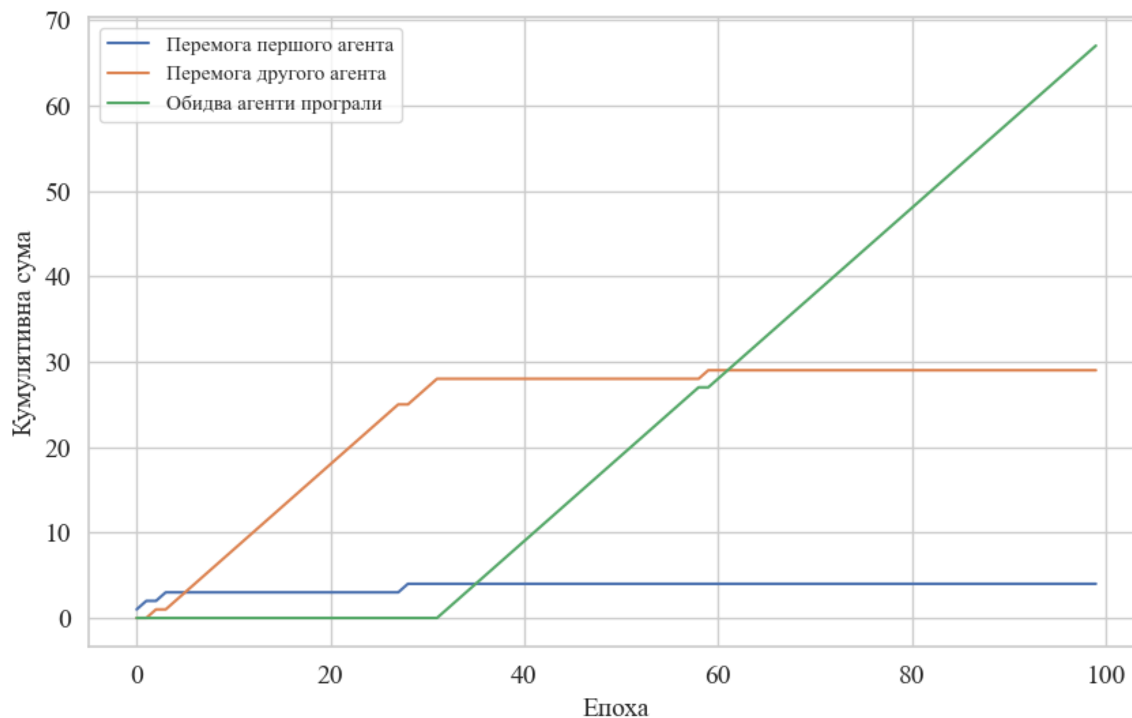

Рисунок 3.5.1 – Результати боїв для першого набору параметрів

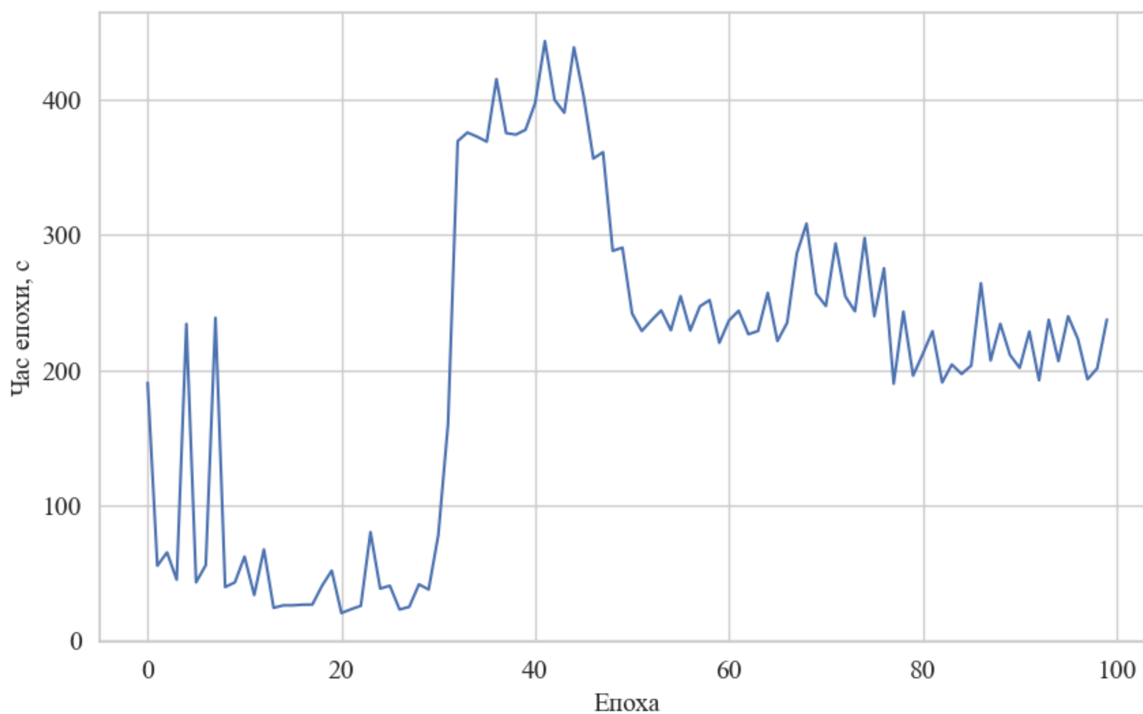

Рисунок 3.5.2 – Час боїв для першого набору параметрів

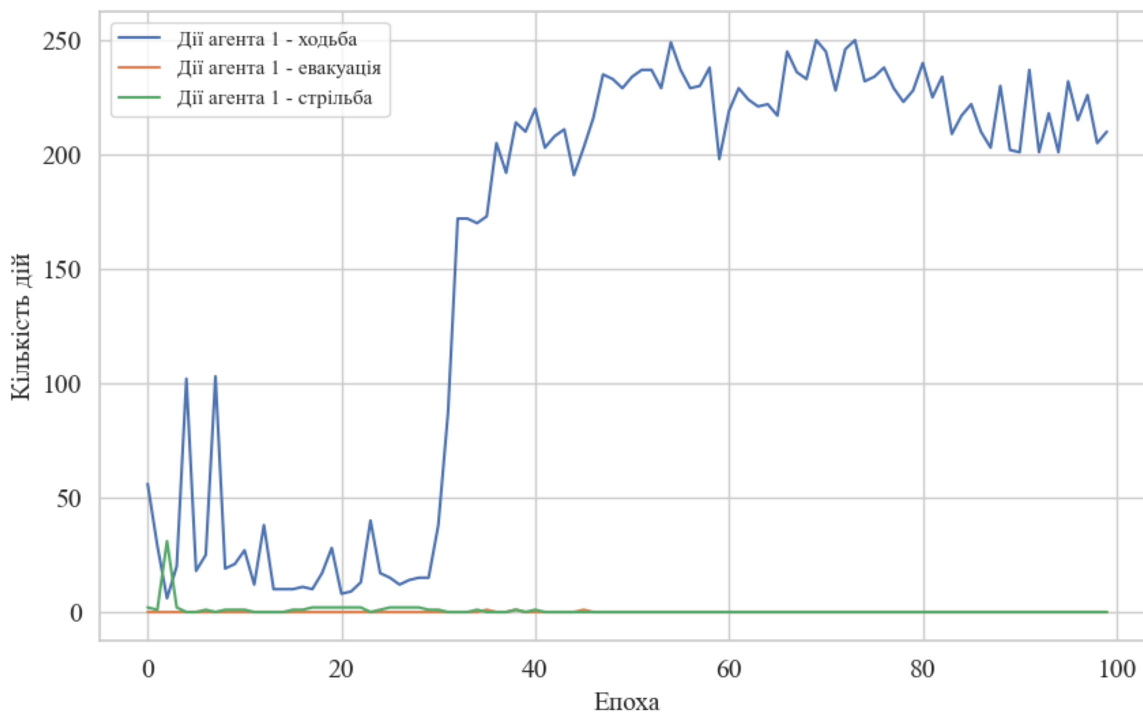

Рисунок 3.5.3 – Дії першого агента для першого набору параметрів

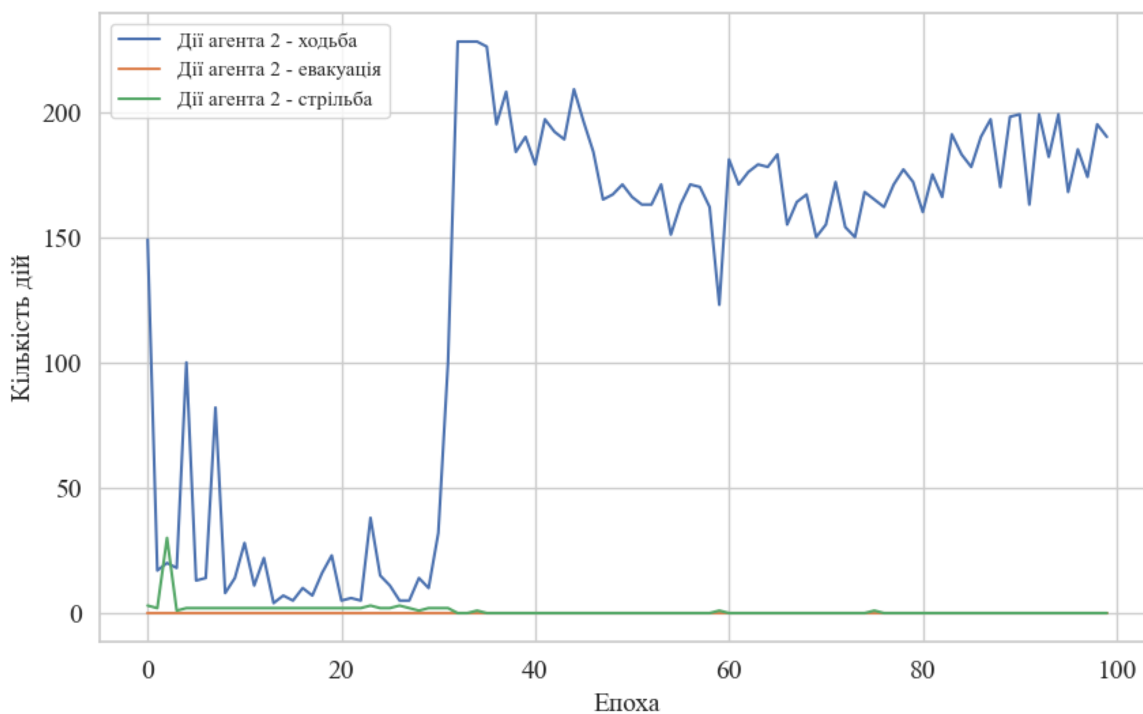

Рисунок 3.5.4 – Дії другого агента для першого набору параметрів

З отриманих графіків можна зробити висновок, що обраний набір гіперпараметрів не підходить для навчання моделі штучного інтелекту. У зв'язку з низьким значенням ε агенти швидко припиняють досліджувати середовище та зациклюються на локальних оптимальних стратегіях. На перших епохах другий агент домінує над першим, оскільки його локальний оптимум виявився краще за оптимум опонента, але з часом вони обидва змінюють свою стратегію та перестають атакувати одне одного «див. рис. 3.5.1». Через такі зміни в поведінці агентів збільшується час, який витрачається на кожну епоху «див. рис. 3.5.2».

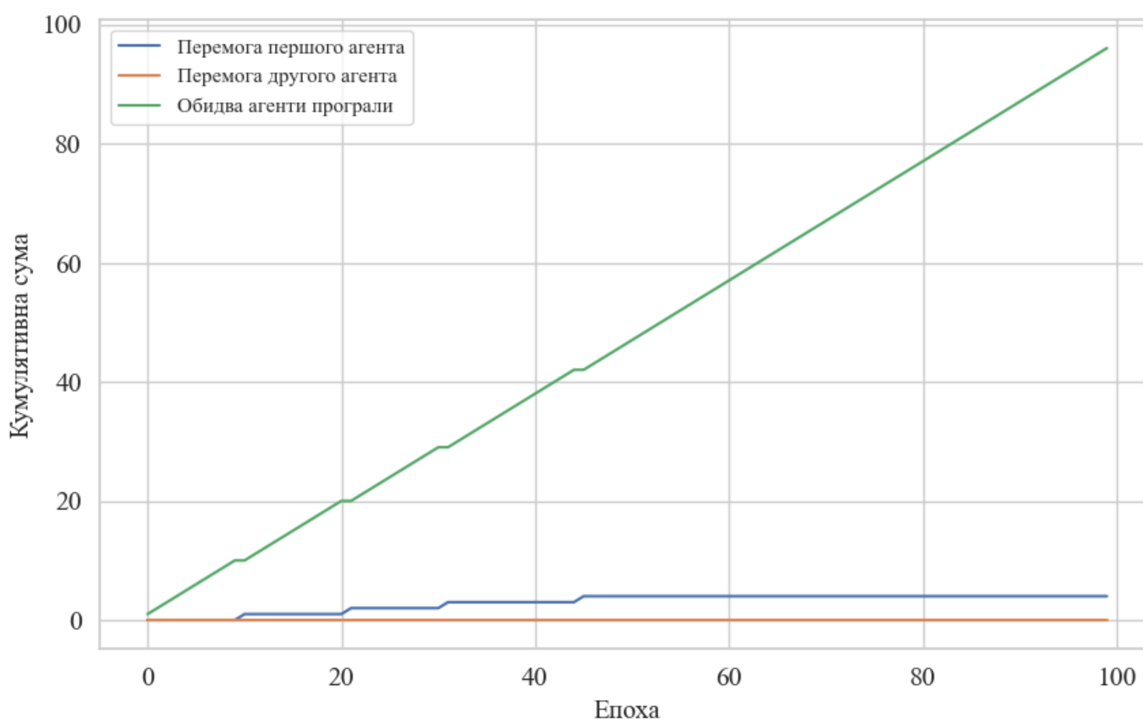

Рисунок 3.5.5 – Результати боїв для другого набору параметрів

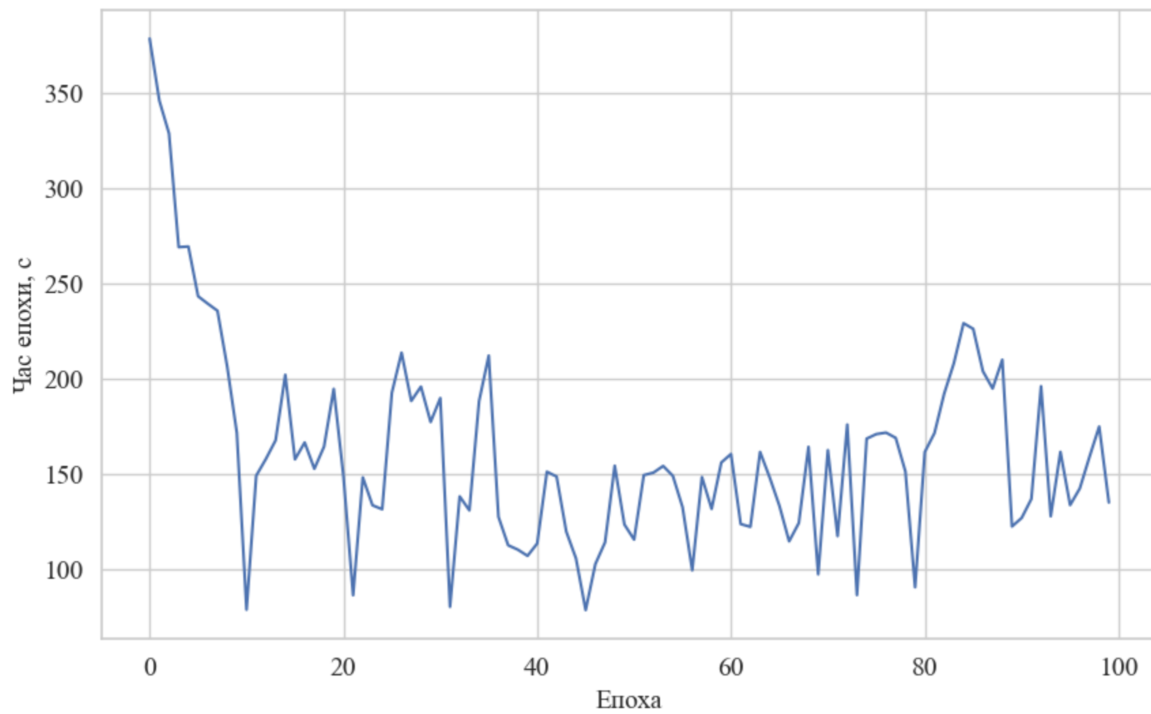

Рисунок 3.5.6 – Час боїв для другого набору параметрів

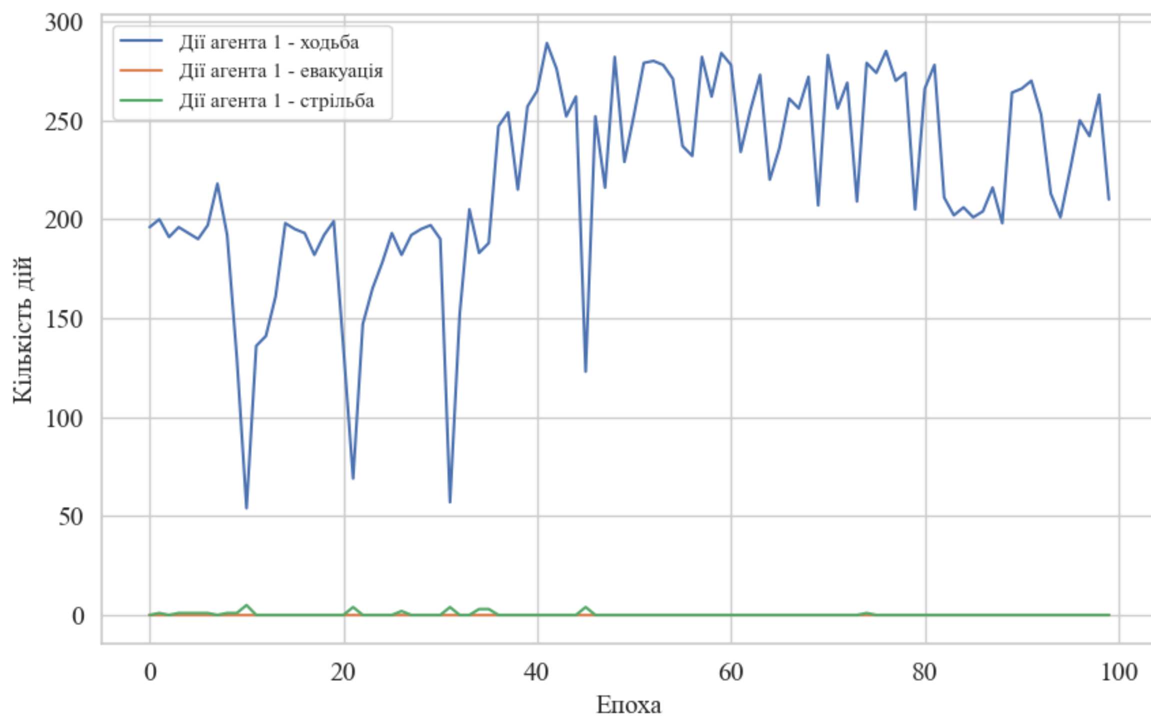

Рисунок 3.5.7 – Дії першого агента для другого набору параметрів

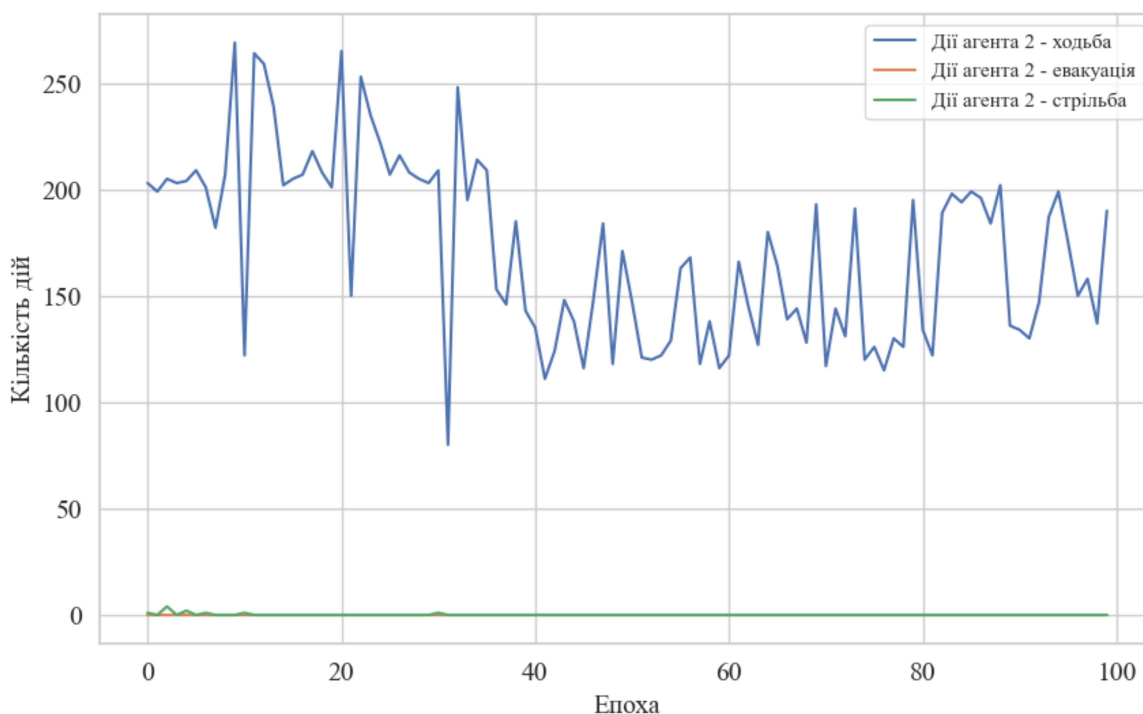

Рисунок 3.5.8 – Дії другого агента для другого набору параметрів

Для другого набору параметрів можна зробити схожі висновки, але з інших причин. Коефіцієнт навчання може бути занадто високим для такої моделі. Висока швидкість навчання може призвести до того, що модель перевищить мінімуми, що призведе до нестабільного навчання та поганої конвергенції «див. рис. 3.5.5». Також можна помітити, що агенти змінюють співвідношення кількості дій, що вони вчиняють з часом, таким чином перший агент спочатку ходить менше за другого «див. рис. 3.5.7», а потім ситуація стає протилежною «див. рис. 3.5.8».

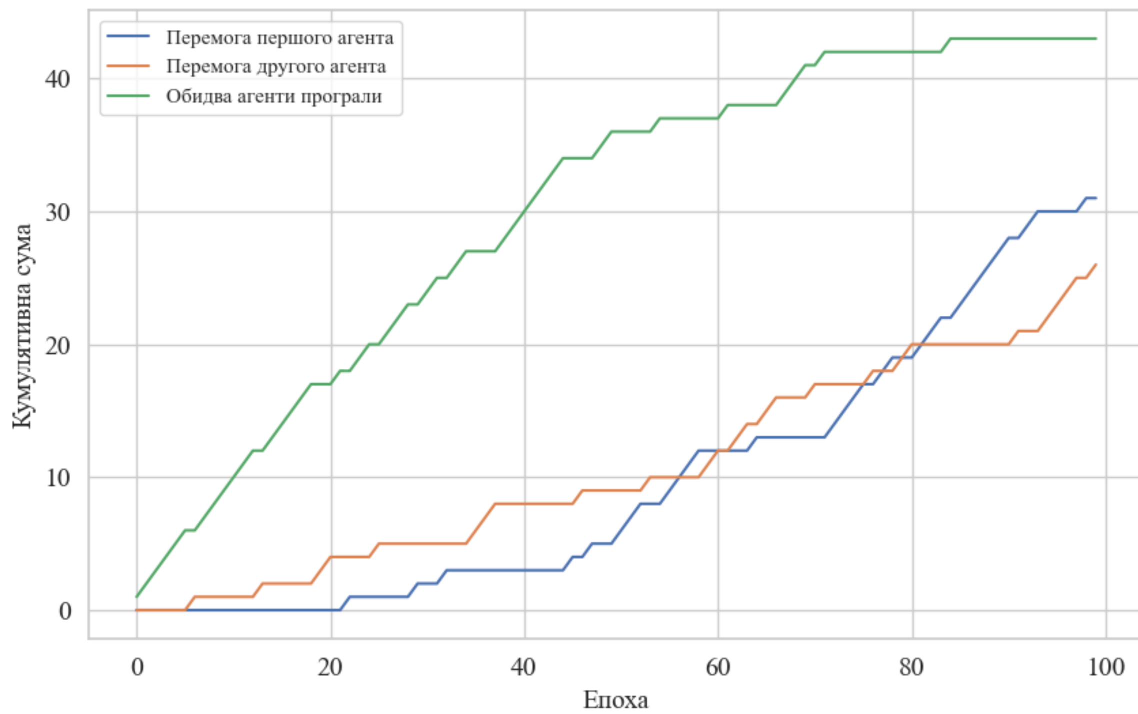

Рисунок 3.5.9 – Результати боїв для третього набору параметрів

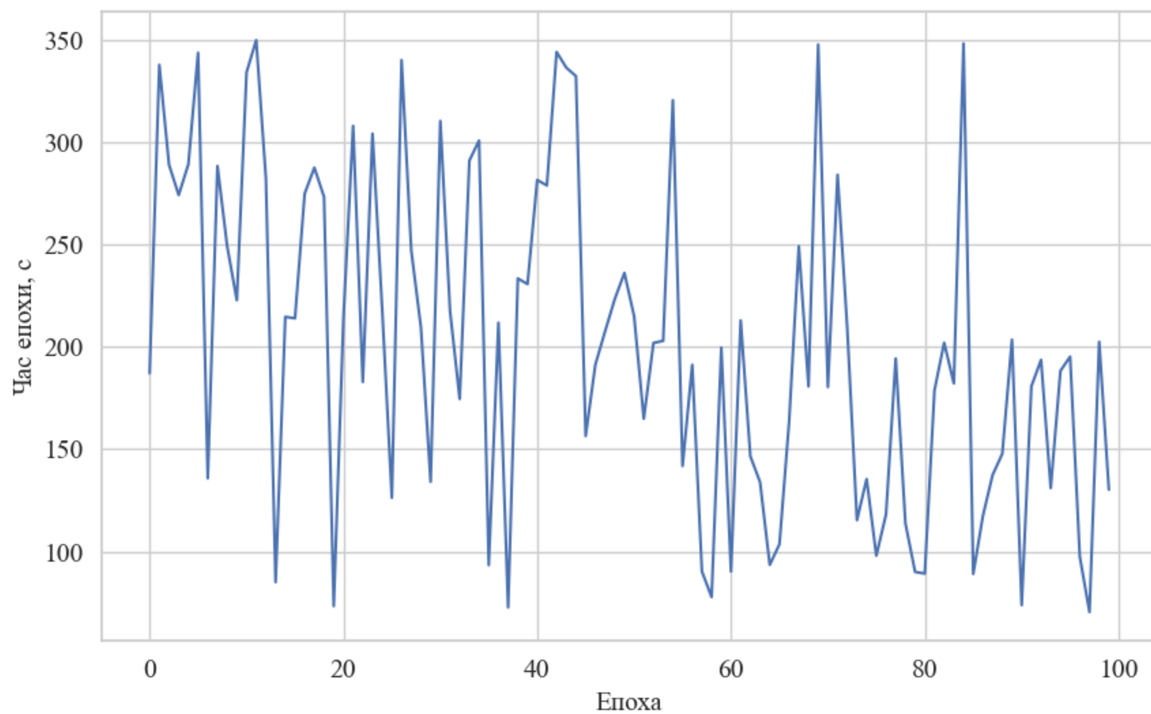

Рисунок 3.5.10 – Час боїв для третього набору параметрів

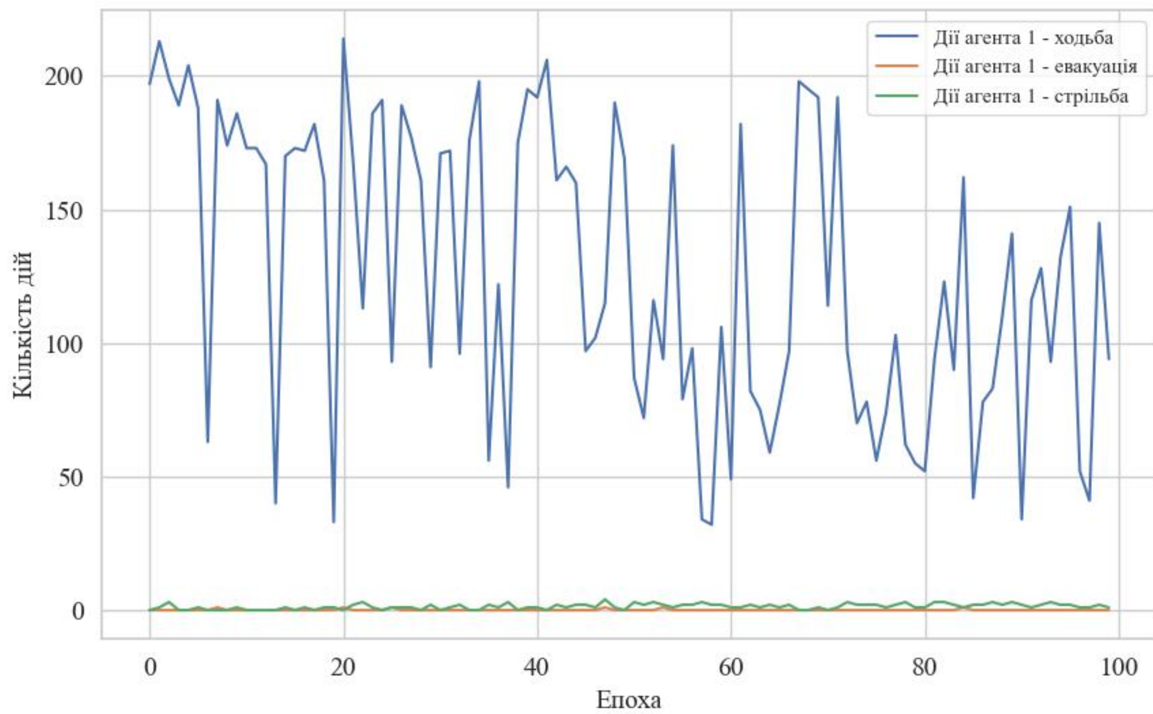

Рисунок 3.5.11 – Дії першого агента для третього набору параметрів

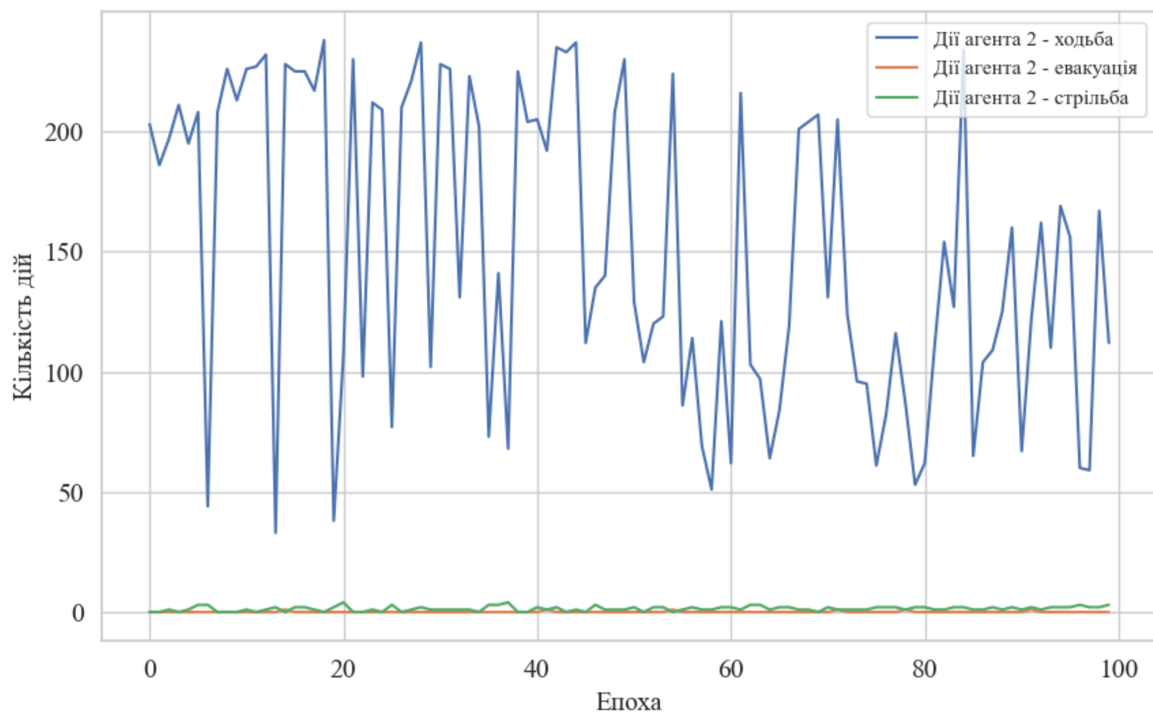

Рисунок 3.5.12 – Дії другого агента для третього набору параметрів

Для обраного набору параметрів можна прослідкувати наступне: навчання було вдалим, оскільки в процесі навчання агенти були рівні по силах одне одному та з часом збільшували частку своїх перемог відносно загальної кількості епох «див. рис. 3.5.9». Ближче до кінця навчального процесу майже кожна битва закінчувалася перемогою одного з гравців, що вказує на оптимізацію їхньої поведінки. Також з графіків, що відображають час боїв «див. рис. 3.5.10» та поведінку першого «див. рис. 3.5.11» і другого «див. рис. 3.5.12» агентів можна побачити залежність між їх діями та тривалістю бою. Побудуємо кореляційну матрицю.

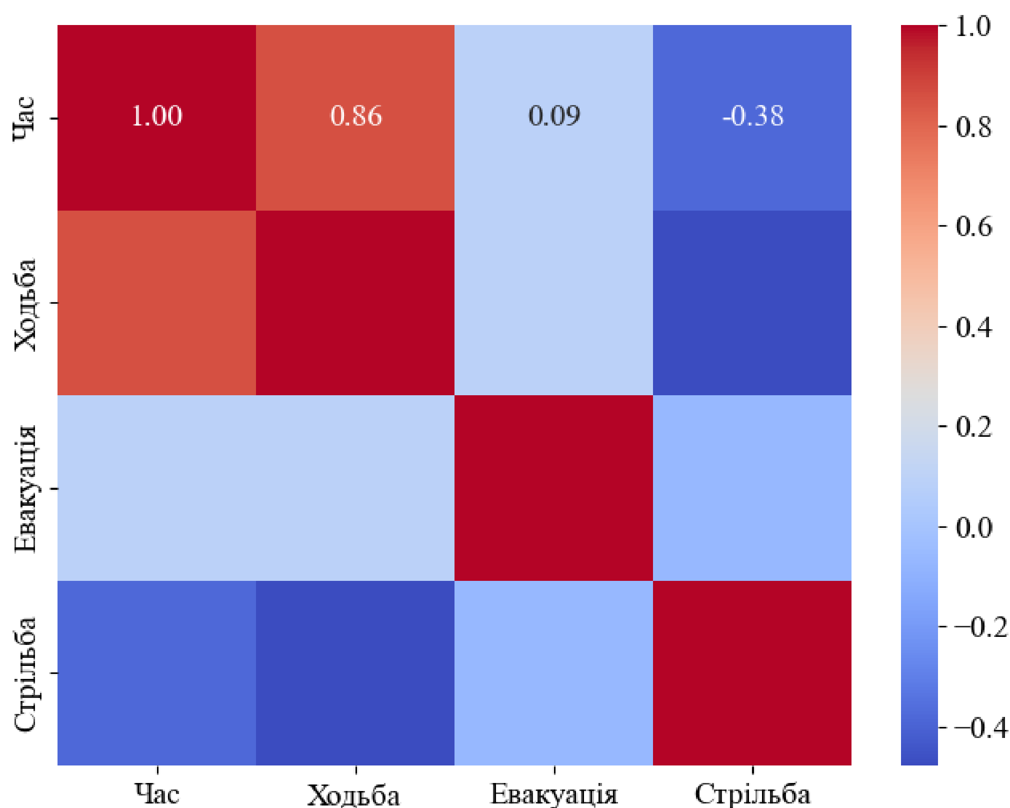

Рисунок 3.5.13 – Кореляційна матриця, що відображає залежність між діями агентів та тривалістю бою для третього набору параметрів

З побудованої матриці «див. рис. 3.5.13» можна побачити те, що час бою має велику кореляцію з кількістю дій, що спрямовані на переміщення мехів. Так само можна зробити висновок, що дії, пов’язані зі стрільбою, які спрямовані на знищення

ворога, мають невелику обернену кореляцію з діями на переміщення та тривалістю бою. Це означає, що такі дії пришвидшують завершення битви.

Для подальшої оптимізації процесу навчання варто спробувати збільшити швидкість навчання та додати коефіцієнт, що буде зменшувати її з часом. Це потенційно може пришвидшити процес тренування агентів. Для кращих результатів можна модифікувати алгоритм глибокого Q-навчання, наприклад реалізувати алгоритм подвійного глибокого Q-навчання.

ВИСНОВОК

У даній роботі було проведено всебічний аналіз та реалізацію штучного інтелекту для ігрового емулятора Rage Of Mechs з використанням алгоритму глибокого Q-навчання. Процес дослідження і реалізації складався з кількох ключових етапів, що охоплюють аналіз середовища емулятора, огляд та порівняння різних алгоритмів штучного інтелекту, а також безпосереднє створення моделі на основі вибраного алгоритму.

На першому етапі було розглянуто середовище емулятора Rage Of Mechs, описано процес з'єднання сервера з клієнтом та проведено попередню обробку даних, необхідних для навчання моделі штучного інтелекту. Цей аналіз дозволив зрозуміти специфіку середовища та визначити ключові параметри для подальшої роботи.

Другий розділ роботи був присвячений огляду алгоритмів для створення штучного інтелекту з використанням нейронної мережі. Було розглянуто вимоги до алгоритму, а також проведено детальний аналіз алгоритмів, таких як дерево пошуку Монте-Карло, метод Актор-Критик та глибоке Q-навчання. На основі порівняння алгоритмів було обрано глибоке Q-навчання, яке виявилось найбільш відповідним для поставлених задач.

У третьому розділі було здійснено побудову моделі штучного інтелекту з використанням алгоритму глибокого Q-навчання. Проведено підготовку середовища ігрового емулятора, обрано архітектуру та побудовано нейронну мережу. Також була виконана оптимізація роботи нейронної мережі, створено систему нагород та проведено навчання агентів. Результати навчання було детально проаналізовано.

В результаті виконаної роботи було створено ефективну модель штучного інтелекту для емулятора гри Rage Of Mechs, яка продемонструвала високу здатність до навчання та адаптації в ігровому середовищі. Отримані результати підтверджують ефективність використання алгоритму глибокого Q-навчання для розв’язання задач у сфері створення штучного інтелекту в ігрових емуляторах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Google LLC. Overview. Protocol Buffers Documentation. URL: https://protobuf.dev/overview/ (date of access: 10.04.2024).
2. Melnikov A., Fette I. RFC 6455: the websocket protocol. IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc6455 (date of access: 10.04.2024).
3. PyTorch Contributors. PyTorch 2.3 documentation. PyTorch. URL: https://pytorch.org/docs/ (date of access: 12.04.2024).
4. Kolda T. G., Bader B. W. Tensor decompositions and applications. SIAM journal on applied mathematics. 2009. Vol. 51, no. 3. P. 455–500.
5. NVIDIA Corporation & affiliates. Introduction – CUDA C Programming Guide. NVIDIA Documentation Hub – NVIDIA Docs. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (date of access: 12.04.2024).
6. Chandrakant K. Reinforcement Learning with Neural Network. Baeldung on Computer Science. URL: https://www.baeldung.com/cs/reinforcement-learning-neural-network (date of access: 14.04.2024).
7. Karunakaran D. The actor-Critic Reinforcement Learning algorithm. Medium. URL: https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14 (date of access: 15.04.2024).
8. Mulvey J. M., Li X., Aydınhan A. O. Solving Multi-Period Financial Planning Models: Combining Monte Carlo Tree Search and Neural Networks. Department of Operations Research and Financial Engineering, Princeton University. 2022 (date of access: 16.04.2024).
9. Roy R. ML | Monte Carlo Tree Search (MCTS). GeeksforGeeks. URL: https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/ (date of access: 16.04.2024).

10. Ranjan A. Implementation of Upper Confidence Bound Algorithm. Medium. URL: https://medium.com/analytics-vidhya/implementation-of-upper-confidence-bound-algorithm-ce0651b63c15 (date of access: 17.04.2024).
11. What is Reinforcement Learning? - Reinforcement Learning Explained. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is/reinforcement-learning (date of access: 19.04.2024).
12. Karagiannakos S. The idea behind Actor-Critics and how A2C and A3C improve them. AI Summer. URL: https://theaisummer.com/Actor_critics/ (date of access: 19.04.2024).
13. Wang M. Deep Q-Learning Tutorial: minDQN. Towards Data Science. URL: https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc (date of access: 20.04.2024).
14. Rastogi V. Fully Connected Layer. Medium. URL: https://medium.com/@vaibhav1403/fully-connected-layer-f13275337c7c (date of access: 26.04.2024).
15. Gandhi M. Fully Connected Layer vs Convolutional Layer. GeeksforGeeks. URL: https://www.geeksforgeeks.org/fully-connected-layer-vs-convolutional-layer/ (date of access: 26.04.2024).
16. Becker D. Rectified Linear Units (ReLU) in Deep Learning. Kaggle: Your Machine Learning and Data Science Community. URL: https://www.kaggle.com/code/dansbecker/rectified-linear-units-relu-in-deep-learning (date of access: 26.04.2024).
17. Graetz F. M. Why AdamW matters. Towards Data Science. URL: https://towardsdatascience.com/why-adamw-matters-736223f31b5d (дата звернення: 27.04.2024).

ДОДАТОК А

Позначення в програмному коді гіперпараметрів, що використовуються для навчання моделі штучного інтелекту разом з їх назвою та призначенням.

Таблиця А1

Позначення в програмному коді	Назва	Призначення
input_size	Розмір вхідного шару	Визначає кількість вхідних нейронів у першому шарі мережі, що відповідає кількості ознак у вхідних даних
hidden_size1	Розмір першого прихованого шару	Визначає кількість нейронів у першому прихованому шарі мережі
hidden_size2	Розмір другого прихованого шару	Визначає кількість нейронів у другому прихованому шарі мережі
output_size	Розмір вихідного шару	Визначає кількість нейронів у вихідному шарі, що відповідає кількості можливих дій, які може виконувати агент
lr	Швидкість навчання	Визначає крок оновлення ваг під час навчання моделі
gamma	Коефіцієнт дисконтування	Визначає, наскільки агент враховує майбутні

		нагороди порівняно з поточними
epsilon	Значення ε для епсилон-жадібної політики	Початкове значення ε для епсилон-жадібної політики, що визначає ймовірність вибору випадкової дії замість найкращої відомої дії на початку навчання
epsilon_decay	Коефіцієнт зменшення ε	Визначає швидкість зменшення ε протягом навчання
epsilon_min	Мінімальне значення ε	Мінімальне значення для ε, нижче якого воно не зменшується
weight_decay	Коефіцієнт регуляризації ваг	Використовується для регуляризації ваг моделі, запобігаючи перенавчанню шляхом додавання штрафу за великі значення ваг
target_update	Частота оновлення цільової мережі	Визначає, як часто оновлюється цільова мережа, що використовується для обчислення цільових значень під час навчання
buffer_capacity	Розмір буфера відтворення	Максимальна кількість досвідів, що зберігаються

		в буфері відтворення для вибіркового навчання моделі
batch_size	Розмір міні-пакету	Кількість досвідів, вибраних з буфера відтворення для навчання моделі під час однієї ітерації
epochs	Кількість епох	Кількість проведених боїв для навчання моделі

ДОДАТОК Б

Програмний код, що відповідає за ініціалізацію та навчання агентів:

```
import asyncio
import torch
import traceback
import tracemalloc
import os
import logging
import time

from src.player import Player
from src.model.environment import GameEnvironment
from src.model.model import DQNAgent
from src.utils import auth, save_epoch_data
from dotenv import load_dotenv

async def main():
    tracemalloc.start()

    logging.basicConfig(level=logging.INFO)
    logger = logging.getLogger(__name__)

    load_dotenv()

    email1 = os.getenv('EMAIL1')
    password1 = os.getenv('PASSWORD1')
    email2 = os.getenv('EMAIL2')
    password2 = os.getenv('PASSWORD2')
    url = os.getenv('URL')

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    player1 = Player(url, email1, password1)
    player2 = Player(url, email2, password2)

    await auth(player1)
    await auth(player2)

    env = GameEnvironment(player1, player2, device)

    input_size = 52
    hidden_size1 = 128
    hidden_size2 = 256
    output_size = 464
    lr = 0.002
    gamma = 0.99
    epsilon = 1.0
    epsilon_decay = 0.999
    epsilon_min = 0.01
    weight_decay = 0.01
    target_update = 25
```

```
    buffer_capacity = 10000
    batch_size = 32
    epochs = 100

    agent1 = DQNAgent(input_size, hidden_size1, hidden_size2, output_size, device, lr, gamma, epsilon,
                      epsilon_decay, epsilon_min, weight_decay, target_update, buffer_capacity,
batch_size)
    agent2 = DQNAgent(input_size, hidden_size1, hidden_size2, output_size, device, lr, gamma, epsilon,
                      epsilon_decay, epsilon_min, weight_decay, target_update, buffer_capacity,
batch_size)

    weights_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'weights')
    os.makedirs(weights_dir, exist_ok=True)

    agent1_weights_path = os.path.join(weights_dir, 'agent1_weights.pth')
    agent2_weights_path = os.path.join(weights_dir, 'agent2_weights.pth')
    epoch_data_path = os.path.join(weights_dir, 'epoch_data.json')

    if os.path.isfile(agent1_weights_path):
        agent1.model.load_state_dict(torch.load(agent1_weights_path))
    if os.path.isfile(agent2_weights_path):
        agent2.model.load_state_dict(torch.load(agent2_weights_path))

    for epoch in range(epochs):
        start_time = time.time()

        player1_actions = {'walk': 0, 'evacuate': 0, 'shoot': 0}
        player2_actions = {'walk': 0, 'evacuate': 0, 'shoot': 0}
        result1, result2 = 0, 0

        try:
            await env.reset()
            done = False
            while not done:
                if env.is_player1_turn:
                    env.actions_dictionary1 = env.check_valid(env.turn_info1,
env.create_actions_dictionary(env.turn_info1))
                    state1 = env.state_tensor1.to(device)
                    state2 = env.state_tensor2.to(device)
                    action = agent1.select_action(state1, env.actions_dictionary1)
                    if action:
                        reward1, reward2, done1, done2 = await env.step(env.player1, action)
                        next_state1 = env.state_tensor1.to(device)
                        next_state2 = env.state_tensor2.to(device)
                        if done2:
                            agent2.optimize_model(state2, action, next_state2, reward2, done2,
env.actions_dictionary2, env.steps)
                        agent1.optimize_model(state1, action, next_state1, reward1, done1,
env.actions_dictionary1, env.steps)
                        if isinstance(action, tuple):
                            player1_actions['walk'] += 1
                        elif action == "evacuate_drop":
                            player1_actions['evacuate'] += 1
                        else:
```

```
                            player1_actions['shoot'] += 1
                    else:
                        await env.player1.send_turn_end_message()
                        await env.update_states(True)
                else:
                    env.actions_dictionary2 = env.check_valid(env.turn_info2,
env.create_actions_dictionary(env.turn_info2))
                    state1 = env.state_tensor1.to(device)
                    state2 = env.state_tensor2.to(device)
                    action = agent2.select_action(state2, env.actions_dictionary2)
                    if action:
                        reward1, reward2, done1, done2 = await env.step(env.player2, action)
                        next_state1 = env.state_tensor1.to(device)
                        next_state2 = env.state_tensor2.to(device)
                        if done1:
                            agent1.optimize_model(state1, action, next_state1, reward1, done1,
env.actions_dictionary1, env.steps)
                        agent2.optimize_model(state2, action, next_state2, reward2, done2,
env.actions_dictionary2, env.steps)
                        if isinstance(action, tuple):
                            player2_actions['walk'] += 1
                        elif action == "evacuate_drop":
                            player2_actions['evacuate'] += 1
                        else:
                            player2_actions['shoot'] += 1
                    else:
                        await env.player2.send_turn_end_message()
                        await env.update_states(True)

                if done1 or done2:
                    done = True
                    if done1[1] == 0:
                        result1 = 1
                    elif done1[1] == 2:
                        result1 = 0
                    else:
                        result1 = -1

                    if done2[1] == 0:
                        result2 = 1
                    elif done2[1] == 2:
                        result2 = 0
                    else:
                        result2 = -1

                    epoch_time = time.time() - start_time
                    save_epoch_data(epoch, epoch_time, player1_actions, player2_actions, result1,
result2, epoch_data_path)

                    torch.save(agent1.model.state_dict(), agent1_weights_path)
                    torch.save(agent2.model.state_dict(), agent2_weights_path)

        except Exception:
            logger.error(traceback.format_exc())
```

```
            await player1.send_battle_end()
            await env.update_states(True)
        finally:
            logger.info(f"Vlad Move: {player1_actions['walk']} times")
            logger.info(f"Vlad Evacuate: {player1_actions['evacuate']} times")
            logger.info(f"Vlad Shoot: {player1_actions['shoot']} times")
            logger.info(f"Bag Move: {player2_actions['walk']} times")
            logger.info(f"Bag Evacuate: {player2_actions['evacuate']} times")
            logger.info(f"Bag Shoot: {player2_actions['shoot']} times")
            logger.info(f"Result: {result1, result2}")
            logger.info(f"Epoch: {epoch}")

if __name__ == '__main__':
    asyncio.run(main())
```

ДОДАТОК В

Програмний код, що відповідає за реалізацію моделі глибокого Q-навчання:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import random

from collections import deque

class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)
        self.action_map = {}
        self.index_map = {}
        self.index_counter = 0

    def add(self, experience):
        self.buffer.append(experience)

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, next_states, rewards, dones, action_dicts = zip(*batch)

        actions = [self.action_map[action] if action in self.action_map else self.add_action(action)
for action in actions]

        dones = [int(done[0]) if isinstance(done, list) else int(done) for done in dones]

        return (torch.stack(states),
                torch.tensor(actions, dtype=torch.long),
                torch.stack(next_states),
                torch.tensor(rewards, dtype=torch.float32),
                torch.tensor(dones, dtype=torch.bool),
                action_dicts)

    def size(self):
        return len(self.buffer)

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

    def add_action(self, action):
        index = self.index_counter
        self.action_map[action] = index
        self.index_map[index] = action
        self.index_counter += 1
        return index

class DQN(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
```

```
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, output_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class DQNAgent:
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size, device, lr, gamma,
epsilon,
                 epsilon_decay, epsilon_min, weight_decay, target_update, buffer_capacity,
batch_size):
        self.device = device
        self.model = DQN(input_size, hidden_size1, hidden_size2, output_size).to(device)
        self.target_model = DQN(input_size, hidden_size1, hidden_size2, output_size).to(device)
        self.target_model.load_state_dict(self.model.state_dict())
        self.target_model.eval()

        self.optimizer = optim.AdamW(self.model.parameters(), lr=lr, weight_decay=weight_decay)
        self.criterion = nn.MSELoss()
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min
        self.target_update = target_update
        self.replay_buffer = ReplayBuffer(buffer_capacity)
        self.batch_size = batch_size

    def select_action(self, state, action_dict):
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)
        valid_actions = [i for i, action in enumerate(action_dict.values()) if action.item() == 1]

        if not valid_actions:
            return None

        if random.random() < self.epsilon:
            action = random.choice(valid_actions)
        else:
            with torch.no_grad():
                q_values = self.model(state)
                valid_q_values = q_values[valid_actions]
                max_q_index = torch.argmax(valid_q_values).item()
                action = valid_actions[max_q_index]

        return list(action_dict.keys())[action]

    def optimize_model(self, state, action, next_state, reward, done, action_dict, steps):
        state = state.float()
        next_state = next_state.float()
```

```
        state_action_values = self.model(state.unsqueeze(0)).squeeze(0)
        state_action_value = state_action_values[list(action_dict.keys()).index(action)]

        self.replay_buffer.add((state, action, next_state, reward, done, action_dict))

        if self.replay_buffer.size() < self.batch_size:
            return

        batch = self.replay_buffer.sample(self.batch_size)
        states, actions, next_states, rewards, dones, action_dicts = batch

        states = states.to(self.device)
        actions = actions.to(self.device)
        next_states = next_states.to(self.device)
        rewards = rewards.to(self.device)
        dones = dones.to(self.device)

        state_action_values = self.model(states).gather(1, actions.unsqueeze(1)).squeeze(1)
        next_state_values = self.target_model(next_states).max(1)[0].detach()
        expected_state_action_values = rewards + (self.gamma * next_state_values * (1 -
dones.float()))

        loss = self.criterion(state_action_values, expected_state_action_values)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        if steps % self.target_update == 0:
            self.target_model.load_state_dict(self.model.state_dict())
```

ДОДАТОК Г

Програмний код для реалізації класу ігрового середовища:

```
import torch
import torch.nn.functional as F
import base64
import asyncio

from google.protobuf.json_format import MessageToDict, ParseDict
from src.messages import turn_response_pb2, battle_end_response_pb2
from src.utils import (
    start_battle,
    calculate_distance,
    create_state_tensor
)

class GameEnvironment:
    def __init__(self, player1, player2, device):
        self.player1 = player1
        self.player2 = player2
        self.device = device
        self.steps = 0
        self.is_player1_turn = None

        self.turn_info1 = None
        self.state_tensor1 = None

        self.turn_info2 = None
        self.state_tensor2 = None

        self.actions_dictionary1 = None
        self.actions_dictionary2 = None

    async def reset(self):
        await asyncio.sleep(1)

        await self.player1.send_ammo_load_all_message()
        await self.player2.send_ammo_load_all_message()

        await self.player1.send_repair_all_message()
        await self.player2.send_repair_all_message()

        await asyncio.sleep(0.5)

        await start_battle(self.player1, self.player2)

        await self.update_states(True)

        self.steps = 0
        self.is_player1_turn = self.turn_info1.isPlayerTurn

    async def step(self, player, action_key):
```

```
        ep_done1, ep_done2 = False, False
        state1 = self.turn_info1
        state2 = self.turn_info2

        if player == self.player1:
            await self.make_action(self.turn_info1, player, action_key)
            self.steps += 1
            ep_done1, ep_done2 = await self.update_states(True)
        else:
            await self.make_action(self.turn_info2, player, action_key)
            self.steps += 1
            ep_done1, ep_done2 = await self.update_states(True)

        if self.steps >= 400:
            await player.send_battle_end()
            await self.update_states(True)
            ep_done1, ep_done2 = [True, 1], [True, 1]

        reward1 = self.calculate_reward(ep_done1, state1, self.turn_info1)
        reward2 = self.calculate_reward(ep_done2, state2, self.turn_info2)

        return reward1, reward2, ep_done1, ep_done2

    async def update_states(self, wait):
        ep_done1, ep_done2 = False, False

        async def get_non_chat_message(player):
            msg = await player.get_message(True)
            while MessageToDict(msg)['type'] == "Chat":
                msg = await player.get_message(False)
            return msg

        msg = await get_non_chat_message(self.player1)
        if MessageToDict(msg)['type'] == "Turn":
            self.turn_info1 =
turn_response_pb2.TurnResponse.FromString(base64.b64decode(MessageToDict(msg)['data']))
            self.state_tensor1 = create_state_tensor(self.turn_info1)

        elif MessageToDict(msg)['type'] == "BattleEnd":
            battle_end =
battle_end_response_pb2.BattleEndResponse.FromString(base64.b64decode(MessageToDict(msg)['data']))
            ep_done1 = [True, battle_end.status]

            await asyncio.sleep(0.5)

            msg = await self.player1.get_message(False)
            if msg and MessageToDict(msg)['type'] == "Turn":
                self.turn_info1 =
turn_response_pb2.TurnResponse.FromString(base64.b64decode(MessageToDict(msg)['data']))
                self.state_tensor1 = create_state_tensor(self.turn_info1)

        msg = await get_non_chat_message(self.player2)
        if MessageToDict(msg)['type'] == "Turn":
```

```
            self.turn_info2 =
turn_response_pb2.TurnResponse.FromString(base64.b64decode(MessageToDict(msg)['data']))
            self.state_tensor2 = create_state_tensor(self.turn_info2)

        elif MessageToDict(msg)['type'] == "BattleEnd":
            battle_end =
battle_end_response_pb2.BattleEndResponse.FromString(base64.b64decode(MessageToDict(msg)['data']))
            ep_done2 = [True, battle_end.status]

            await asyncio.sleep(0.5)

            msg = await self.player2.get_message(False)
            if msg and MessageToDict(msg)['type'] == "Turn":
                self.turn_info2 =
turn_response_pb2.TurnResponse.FromString(base64.b64decode(MessageToDict(msg)['data']))
                self.state_tensor2 = create_state_tensor(self.turn_info2)

        self.is_player1_turn = self.turn_info1.isPlayerTurn

        return ep_done1, ep_done2

    def create_actions_dictionary(self, turn_info):
        movement_actions = {(dx, dy): 0 for dx in range(-10, 11) for dy in range(-10, 11) if (dx != 0
or dy != 0)}

        attack_actions = {action: 0 for action in ['rocket_shot', 'burst_shot']}

        module_ids = [module.moduleId for module in turn_info.playerMechs[0].modules]
        aimed_actions = {f'aimed_{module_id}': 0 for module_id in module_ids}

        interaction_actions = {action: 0 for action in ['evacuate_drop']}

        actions_dictionary = {**movement_actions, **attack_actions, **aimed_actions,
**interaction_actions}
        actions_dictionary = {k: torch.tensor(v, device=self.device) for k, v in
actions_dictionary.items()}

        return actions_dictionary

    def check_valid(self, turn_info, actions_dictionary):
        player_mech_position = (turn_info.playerMechs[0].position.X if
hasattr(turn_info.playerMechs[0].position, 'X') else 0,
                         turn_info.playerMechs[0].position.Y if
hasattr(turn_info.playerMechs[0].position, 'Y') else 0)
        turn_points = turn_info.playerMechs[0].turnPoints

        for move in turn_info.turn.moves:
            move_position = (move.position.X if hasattr(move.position, 'X') else 0,
                             move.position.Y if hasattr(move.position, 'Y') else 0)
            delta_x = move_position[0] - player_mech_position[0]
            delta_y = move_position[1] - player_mech_position[1]

            if (delta_x, delta_y) in actions_dictionary and turn_points >= move.cost:
                actions_dictionary[(delta_x, delta_y)] = torch.tensor(1, device=self.device)
```

```
        if hasattr(turn_info, "enemyMechs") and (len(turn_info.enemyMechs) >= 1):
            enemy_mech_position = (turn_info.enemyMechs[0].position.X if
hasattr(turn_info.enemyMechs[0].position, 'X') else 0,
                                   turn_info.enemyMechs[0].position.Y if
hasattr(turn_info.enemyMechs[0].position, 'Y') else 0)
        else: enemy_mech_position = False
        if hasattr(turn_info, "dropItems") and (len(turn_info.dropItems) >= 1):
            drop_position = (turn_info.dropItems[0].position.X if
hasattr(turn_info.dropItems[0].position, 'X') else 0,
                             turn_info.dropItems[0].position.Y if
hasattr(turn_info.dropItems[0].position, 'Y') else 0)
        else: drop_position = False

        for module in turn_info.turn.modules:
            if module.effect.effectType.code in ['damage_kinetic', 'damage_heat', 'damage_energy',
'aim_shoot',
                                                 'damage_aimed_kinetic', 'damage_aimed_energy',
'damage_aimed_heat']:
                for tile in module.moduleEffectTiles:
                    tile_position = (tile.position.X if hasattr(tile.position, 'X') else 0,
                                     tile.position.Y if hasattr(tile.position, 'Y') else 0)
                    if enemy_mech_position and (tile_position == enemy_mech_position) and (tile.chance
> 0):
                        if module.effect.name == "rocket_150mm_shot" and turn_points >= tile.cost:
                            actions_dictionary["rocket_shot"] = torch.tensor(1, device=self.device)

                        if module.effect.name == "doomsday_armor_piercing_burst" and turn_points >=
tile.cost:
                            actions_dictionary["burst_shot"] = torch.tensor(1, device=self.device)

                    if enemy_mech_position and (tile_position == enemy_mech_position) and
hasattr(tile, "aimedModuleEffects"):
                        for aimed in tile.aimedModuleEffects:
                            aimed_action = f'aimed_{aimed.mechModule.moduleId}'
                            if aimed.chance > 0 and turn_points >= 14:
                                enemy_module = next((m for m in turn_info.enemyMechs[0].modules if
m.moduleId == aimed.mechModule.moduleId), None)
                                if enemy_module and enemy_module.hpCurrent > 0:
                                    actions_dictionary[aimed_action] = torch.tensor(1,
device=self.device)

            if drop_position and module.effect.effectType.code == 'drop_evacuate':
                tile = module.moduleEffectTiles[0]
                tile_position = (tile.position.X if hasattr(tile.position, 'X') else 0,
                                 tile.position.Y if hasattr(tile.position, 'Y') else 0)
                if turn_points >= tile.cost and (tile_position == drop_position):
                    actions_dictionary['evacuate_drop'] = torch.tensor(1, device=self.device)

        return actions_dictionary

    async def make_action(self, turn_info, player, action_key):
        player_mech = turn_info.playerMechs[0]
```

```
        rocket, gun = 0, 0
        for module in turn_info.turn.modules:
            if module.effect.name == "rocket_150mm_shot":
                rocket = module.moduleId
            if module.effect.name == "doomsday_armor_piercing_burst":
                gun = module.moduleId
        player_modules = [rocket, gun]

        if hasattr(turn_info, 'enemyMechs') and (len(turn_info.enemyMechs) > 0):
            enemy_mech = turn_info.enemyMechs[0]
            enemy_modules = {module.moduleId: module.id for module in enemy_mech.modules}

        if isinstance(action_key, tuple):
            await player.send_turn_move_message(player_mech.id,
                                                (player_mech.position.X if
hasattr(player_mech.position, 'X') else 0) + action_key[0],
                                                (player_mech.position.Y if
hasattr(player_mech.position, 'Y') else 0) + action_key[1])
            print(action_key, player.email)

        elif action_key == "evacuate_drop":
            drop_item = turn_info.dropItems[0]
            await player.send_turn_drop_message(drop_item.id,
                                                2,
                                                drop_item.position.X if hasattr(drop_item.position,
'X') else 0,
                                                drop_item.position.Y if hasattr(drop_item.position,
'Y') else 0)
            print(action_key, player.email)
            await asyncio.sleep(0.5)

        elif action_key == "rocket_shot":
            await player.send_turn_shoot_message(player_modules[0],
                                                 22,
                                                 enemy_mech.position.X if hasattr(enemy_mech.position,
'X') else 0,
                                                 enemy_mech.position.Y if hasattr(enemy_mech.position,
'Y') else 0)
            print(action_key, player.email)
            await asyncio.sleep(0.5)

        elif action_key == "burst_shot":
            await player.send_turn_shoot_message(player_modules[1],
                                                 49,
                                                 enemy_mech.position.X if hasattr(enemy_mech.position,
'X') else 0,
                                                 enemy_mech.position.Y if hasattr(enemy_mech.position,
'Y') else 0)
            print(action_key, player.email)
            await asyncio.sleep(0.5)

        elif action_key.startswith("aimed_"):
            module_id = int(action_key.split('_')[1])
            if module_id in enemy_modules:
```

```
            await player.send_turn_shoot_message(player_modules[1],
                                                 63,
                                                 enemy_mech.position.X if
hasattr(enemy_mech.position, 'X') else 0,
                                                 enemy_mech.position.Y if
hasattr(enemy_mech.position, 'Y') else 0,
                                                 enemy_modules[module_id])
            print(action_key, player.email)
            await asyncio.sleep(0.5)

    async def repair_and_ammo(self):
        await self.player.send_ammo_load_all_message()
        await self.player.get_message(True)
        await self.player.send_repair_all_message()
        await self.player.get_message(True)

    def calculate_reward(self, ep_done, state, next_state):
        reward = 0
        if isinstance(ep_done, (list, tuple)):
            if ep_done[1] == 0:
                reward += 80
                print("WON")
            if ep_done[1] == 1:
                reward += -120
                print("LOST")
            if ep_done[1] == 2:
                reward += -40
                print("DRAW")
            return reward

        distance_to_drop = calculate_distance(next_state)

        if distance_to_drop > 12:
            reward += -1
        elif distance_to_drop <= 12 and distance_to_drop > 1:
            reward += -1 + (12 - distance_to_drop) / 1
        elif distance_to_drop <= 1:
            reward += 0

        if hasattr(next_state, "dropItems") and (next_state.dropItems[0].status == "Evacuation"):
            if 1 <= next_state.dropItems[0].evacuationProgress.progressEnemy <= 4:
                reward += -2
            elif 1 <= next_state.dropItems[0].evacuationProgress.progressPlayer <= 4:
                reward += 2

        if hasattr(next_state, "enemyMechs") and (len(next_state.enemyMechs) > 0):
            reward += 1.5

            if hasattr (state, "enemyMechs") and (len(state.enemyMechs) > 0):
                current_enemy_mech_hp = sum([module.hpCurrent for module in
next_state.enemyMechs[0].modules])
                previous_enemy_mech_hp = sum([module.hpCurrent for module in
state.enemyMechs[0].modules])
```

```
                if current_enemy_mech_hp < previous_enemy_mech_hp:
                    reward += 10

            current_player_mech_hp = sum([module.hpCurrent for module in
next_state.playerMechs[0].modules])
            previous_player_mech_hp = sum([module.hpCurrent for module in
state.playerMechs[0].modules])

            if current_player_mech_hp < previous_player_mech_hp:
                reward -= 5

        reward = torch.tensor([reward], device=self.device, dtype=torch.float32).clone().detach()

        return reward
```

ДОДАТОК Ґ

Програмний код для реалізації класу гравця:

```
import asyncio
import requests
import websockets
import logging

from google.protobuf.json_format import MessageToDict, ParseDict
from src.messages import (
    message_pb2,
    battle_start_request_pb2,
    turn_mech_request_pb2,
    turn_move_request_pb2,
    turn_drop_request_pb2,
    turn_shoot_request_pb2,
)


logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

COMMAND_QUEUE_SIZE = 50

class Player:
    def __init__(self, url, email, password):
        self.url = url
        self.email = email
        self.password = password
        self.ws_connect = None
        self.sign_up_resp = None
        self.log_in_resp = None
        self.me_resp = None
        self.in_queue = asyncio.Queue(COMMAND_QUEUE_SIZE)

    async def ws_init(self):
        if not self.get_token():
            raise ValueError("access token does not exist")

        ws_url = 'ws' + self.url[4:] + '/ws'
        self.ws_connect = await websockets.connect(ws_url)
        asyncio.create_task(self.handle_read())

    async def handle_read(self):
        while True:
            try:
                message = await self.ws_connect.recv()
                m = message_pb2.Message.FromString(message)
                await self.in_queue.put(m)
            except Exception as e:
                logger.error(f"error reading message: {e}")
                continue
```

```
    async def send_auth_message(self, token):
        return await self.send_message(message_pb2.Message.MessageType.Auth, token.encode())

    async def send_join_message(self):
        return await self.send_message(message_pb2.Message.MessageType.Join, b'')

    async def send_battle_end(self):
        return await self.send_message(message_pb2.Message.MessageType.BattleEnd, b'')

    async def send_turn_message(self):
        return await self.send_message(message_pb2.Message.MessageType.Turn, b'')

    async def send_turn_end_message(self):
        return await self.send_message(message_pb2.Message.MessageType.TurnEnd, b'')

    async def send_scene_message(self):
        return await self.send_message(message_pb2.Message.MessageType.Scene, b'')

    async def send_hangar_message(self):
        return None

    async def send_ammo_load_all_message(self):
        await self.send_message(message_pb2.Message.MessageType.MechsAmmoLoadAll, b'')

        msg = await self.get_message(True)
        if MessageToDict(msg)['type'] != 'Mechs':
            raise ValueError("type is not equal")

    async def send_repair_all_message(self):
        await self.send_message(message_pb2.Message.MessageType.MechsRepairAll, b'')

        msg = await self.get_message(True)
        if MessageToDict(msg)['type'] != 'Mechs':
            raise ValueError("type is not equal")

    async def send_hangar_end_message(self):
        await self.send_message(message_pb2.Message.MessageType.HangarEnd, b'')

        msg = await self.get_message(True)
        if MessageToDict(msg)['type'] != 'HangarEnd':
            raise ValueError("type is not equal")

    async def send_battle_start_message(self, state, map, day_time):
        message = battle_start_request_pb2.BattleStartRequest()
        message.state = 0 if state else 1
        message.map = map
        message.dayTime = day_time

        return await self.send_message(message_pb2.Message.MessageType.BattleStart,
message.SerializeToString())

    async def send_map_message(self):
        return await self.send_message(message_pb2.Message.MessageType.Map, b'')
```

```
    async def send_turn_mech_message(self, mech_id):
            message = turn_mech_request_pb2.TurnMechRequest()
            message.mechId = mech_id

            return await self.send_message(message_pb2.Message.MessageType.TurnMech,
message.SerializeToString())

    async def send_turn_move_message(self, mech_id, x, y):
        message = turn_move_request_pb2.TurnMoveRequest()

        message.mechId = mech_id
        message.position.X = x
        message.position.Y = y

        return await self.send_message(message_pb2.Message.MessageType.TurnMove,
message.SerializeToString())

    async def send_turn_shoot_message(self, module_id, effect_id, x, y, aimed_module_id=None):
        message = turn_shoot_request_pb2.TurnShootRequest()

        message.moduleId = module_id
        message.effectId = effect_id
        if aimed_module_id:
            message.aimedModuleId = aimed_module_id
        message.position.X = x
        message.position.Y = y

        return await self.send_message(message_pb2.Message.MessageType.TurnShoot,
message.SerializeToString())

    async def send_turn_drop_message(self, drop_id, action_type, x, y):
        message = turn_drop_request_pb2.TurnDropRequest()

        message.position.X = x
        message.position.Y = y
        message.dropItemId = drop_id
        message.type = action_type

        return await self.send_message(message_pb2.Message.MessageType.Drop,
message.SerializeToString())

    async def send_message(self, message_type, data):
        if not self.ws_connect:
            raise ValueError("could not connect to WebSocket")

        message = message_pb2.Message()
        message.type = message_type
        message.data = data
        await self.ws_connect.send(message.SerializeToString())

    async def get_message(self, wait):
        if wait:
            message = await self.in_queue.get()
            return message
```

```
        return self.in_queue.get_nowait() if not self.in_queue.empty() else None

    def sign_up(self):
        data = {"email": self.email, "password": self.password}
        response = requests.post(f"{self.url}/api/users/sign_up", json=data)
        self.sign_up_resp = response.json()
        return self.sign_up_resp

    def log_in(self):
        data = {"email": self.email, "password": self.password}
        response = requests.post(f"{self.url}/api/users/sign_in", json=data)
        self.log_in_resp = response.json()
        return self.log_in_resp

    def set_token(self, auth_token):
        self.auth_token = auth_token

    def get_token(self):
        return self.auth_token
```

ДОДАТОК Д

Програмний код для реалізації допоміжних функцій:

- **Авторизація**

```
import asyncio
from google.protobuf.json_format import MessageToDict, ParseDict

async def auth(player):
    sign_up_response = player.sign_up()
    log_in_response = player.log_in()

    auth_token = log_in_response['data']['access_token']
    player.set_token(auth_token)

    await player.ws_init()

    try:
        await player.send_auth_message(auth_token)
    except Exception as e:
        print(f"Failed to send authentication message: {e}")

    await asyncio.sleep(1)

    message = await player.get_message(False)
    assert MessageToDict(message)['type'] == 'Auth'
```

- **Відстань між мехом та ігровим об'єктом**

```
import torch
from torch import linalg as LA

def calculate_distance(next_state):
    if hasattr(next_state, "playerMechs"):
        mech_position = torch.tensor([next_state.playerMechs[0].position.X if
hasattr(next_state.playerMechs[0].position, 'X') else 0,
                                        next_state.playerMechs[0].position.Y if
hasattr(next_state.playerMechs[0].position, 'Y') else 0])
    else: return 0

    if hasattr(next_state, "dropItems"):
        drop_position = torch.tensor([next_state.dropItems[0].position.X if
hasattr(next_state.dropItems[0].position, 'X') else 0,
                                        next_state.dropItems[0].position.Y if
hasattr(next_state.dropItems[0].position, 'Y') else 0])
    else: return 0

    return LA.norm(mech_position.float() - drop_position.float())
```

- **Створення тензорів з інформацією про стан**

```
import torch

def create_drop_tensor(turn_info):
    if hasattr(turn_info, "dropItems"):
        drop_info = []
```

```
        for drop_item in turn_info.dropItems:
            drop_info.extend([
                turn_info.evacuationProgress.progressPlayer if hasattr(turn_info.evacuationProgress,
'progressPlayer') else 0,
                turn_info.evacuationProgress.progressEnemy if hasattr(turn_info.evacuationProgress,
'progressEnemy') else 0,
                turn_info.evacuationProgress.progressTarget if hasattr(turn_info.evacuationProgress,
'progressTarget') else 0,
                drop_item.position.X if hasattr(drop_item.position, 'X') else 0,
                drop_item.position.Y if hasattr(drop_item.position, 'Y') else 0,
            ])

        return torch.tensor(drop_info)
    else:
        return torch.zeros(5)

def create_player_mech_tensor(turn_info):
    if hasattr(turn_info, "playerMechs") and (len(turn_info.playerMechs) > 0):
        player_mech_info = []
        player_mech = turn_info.playerMechs[0]

        player_mech_info.append(turn_info.turn.turnPoints)

        player_mech_info.extend([
            player_mech.position.X if hasattr(player_mech.position, 'X') else 0,
            player_mech.position.Y if hasattr(player_mech.position, 'Y') else 0,
        ])

        for module in player_mech.modules:
            player_mech_info.append(module.hpCurrent)

        return torch.tensor(player_mech_info)
    else:
        return torch.zeros(24)

def create_enemy_mech_tensor(turn_info):
    if hasattr(turn_info, "enemyMechs") and (len(turn_info.enemyMechs) > 0):
        enemy_mech_info = []
        enemy_mech = turn_info.enemyMechs[0]

        enemy_mech_info.extend([
            enemy_mech.position.X if hasattr(enemy_mech.position, 'X') else 0,
            enemy_mech.position.Y if hasattr(enemy_mech.position, 'Y') else 0,
        ])

        for module in enemy_mech.modules:
            enemy_mech_info.append(module.hpCurrent)

        return torch.tensor(enemy_mech_info)
    else:
        return torch.zeros(23)

def create_state_tensor(turn_info):
    drop_tensor = create_drop_tensor(turn_info)
```

```
    player_mech_tensor = create_player_mech_tensor(turn_info)
    enemy_mech_tensor = create_enemy_mech_tensor(turn_info)

    concatenated_tensor = torch.cat((drop_tensor, player_mech_tensor, enemy_mech_tensor))

    return concatenated_tensor.float()
```

- **Збереження інформації про навчання**

```
import os
import json

def save_epoch_data(epoch, epoch_time, player1_actions, player2_actions, result1, result2, filename):
    epoch_data = {
        "epoch": epoch,
        "epoch_time": epoch_time,
        "agent1": {
            "walk_actions": player1_actions['walk'],
            "evacuate_actions": player1_actions['evacuate'],
            "shoot_actions": player1_actions['shoot'],
            "result_of_epoch": result1
        },
        "agent2": {
            "walk_actions": player2_actions['walk'],
            "evacuate_actions": player2_actions['evacuate'],
            "shoot_actions": player2_actions['shoot'],
            "result_of_epoch": result2
        }
    }

    if os.path.isfile(filename):
        with open(filename, 'r') as file:
            data = json.load(file)
        data.append(epoch_data)
    else:
        data = [epoch_data]

    with open(filename, 'w') as file:
        json.dump(data, file, indent=4)
```

- **Початок битви**

```
from google.protobuf.json_format import MessageToDict, ParseDict

async def start_battle(player1, player2):

    await player1.send_hangar_end_message()
    await player2.send_hangar_end_message()

    await player1.send_battle_start_message(True, "pripyat_x2", "day")
    await player2.send_battle_start_message(True, "pripyat_x2", "day")

    message = await player1.get_message(True)
    assert MessageToDict(message)['type'] == 'SearchEnemy'

    message = await player2.get_message(True)
    assert MessageToDict(message)['type'] == 'SearchEnemy'
```

```
message = await player1.get_message(True)
assert MessageToDict(message)['type'] == 'WaitingEnded'

message = await player2.get_message(True)
assert MessageToDict(message)['type'] == 'WaitingEnded'

message = await player1.get_message(True)
assert MessageToDict(message)['type'] == 'SearchEnd'

message = await player2.get_message(True)
assert MessageToDict(message)['type'] == 'SearchEnd'

await player1.send_join_message()
await player2.send_join_message()
```