МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних інформаційних систем
_____ Юрій КОНДРАТЕНКО
«____»_____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА

ІНТЕГРОВАНА СИСТЕМА ДЛЯ НАВЧАННЯ АГЕНТІВ UNITY ML AGENTS З ВИКОРИСТАННЯМ МОДЕЛЕЙ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ

Спеціальність 122 Комп'ютерні науки

Освітня програма «Інтелектуальні інформаційні системи»

Здобувач _____ Владислав РЕВА

«___»_____ 2024 р.

Керівник канд. фіз.-мат. наук, доцент каф. ІІС __ _____ Інесса КУЛАКОВСЬКА

«___»_____ 2024 р.

Миколаїв – 2024

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

Факультет	Комп'ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Другий (магістерський)
Освітній ступень	Магістр
Спеціальність	122 Комп'ютерні науки
Освітня програма	Інтелектуальні інформаційні системи

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Юрій КОНДРАТЕНКО

«____» _____ 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

Реви Владислава Володимировича
((прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи:«Інтегрована система для навчання агентів unity ml agents з використанням моделей глибоких нейронних мереж».
Керівник роботи: Кулаковська Інесса Василівна, доцент. каф. ІІС, канд. фіз.-мат. наук, доцент.
Затверджена наказом ЧНУ ім. Петра Могили від «03» червня 2024 р. № 140/1.
2. Строк представлення кваліфікаційної роботи«___» _____ 2024 р.
3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: середовище для тренування агентів Unity ML-Agents; базвові параметри агентів та сценарії для навчання; моделі глибоких нейронних мереж, що використовуються для навчання агентів.
Очікуваний результат: розроблені агенти, які зможуть ефективно виконувати визначені завдання, візуалізація процесу навчання агентів та аналіз їх поведінки.
4. Перелік питань, що підлягали розробці:

– розглянуто Unity ML-Agents та його інтеграцію з моделями глибокого навчання;

– проаналізовано вхідні дані для навчання агентів, їх попередню обробку;

– розроблено та налаштовано моделі глибоких нейронних мереж для тренування агентів;

– побудовано та оцінено ефективність алгоритмів підкріплювального навчання;

– описано процеси тестування та валідації навчених агентів;

– написані висновки щодо подальшого покращення роботи агентів.

5. Перелік графічних матеріалів: презентація, додатки, рисунки, таблиці.

Керівник роботи _____ Інесса КУЛАКОВСЬКА
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Владислав РЕВА
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «07» червня 2024 р.

КАЛЕНДАРНИЙ ПЛАН
кваліфікаційної роботи

Тема: Інтегрована система для навчання агентів Unity ML Agents з використанням моделей глибоких нейронних мереж

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	03.06.2024	07.06.2024	вик.
2	Аналіз предметної області та постановка задачі	10.06.2024	20.06.2024	вик.
3	Огляд літературних джерел за темою кваліфікаційної роботи, зокрема аналіз публікацій та аналогічних систем, щодо навчання агентів Unity ML Agents	21.06.2024	01.07.2024	вик.
4	Встановлення Unity, бібліотеки MLAgents та віртуального тренувального простору. Налаштування їх взаємодії	01.09.2024	25.09.2024	вик.
5	Налаштування параметрів агентів та простору для тренування.	26.09.2024	01.10.2024	вик.
6	Програмування логіки агентів	01.10.2024	21.11.2024	вик.
7	Перший попередній захист КР на засіданні комісії кафедри	22.11.2024	22.11.2024	вик.
8	Корегування роботи за результатами попереднього захисту	23.11.2024	05.12.2024	вик.
9	Другий попередній захист КР на засіданні комісії кафедри	06.12.2024	06.12.2024	вик.
10	Доробка та остаточне оформлення КР	07.12.2024	10.02.2024	вик.
11	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	16.12.2024	17.12.2024	вик.

Керівник роботи _____ Інесса КУЛАКОВСЬКА
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Владислав РЕВА
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану
«19» червня 2024 р.

АНОТАЦІЯ

до кваліфікаційної роботи

здобувача групи 601м ЧНУ ім. Петра Могили

Реви Владислава Володимировича

на тему: **“ ІНТЕГРОВАНА СИСТЕМА ДЛЯ НАВЧАННЯ АГЕНТІВ UNITY ML AGENTS З ВИКОРИСТАННЯМ МОДЕЛЕЙ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ”**

Актуальність даного дослідження полягає у швидкості розвитку штучного інтелекту та його застосування в різних галузях, навчання агентів, здатних до автономного прийняття рішень у складних динамічних середовищах, стає дедалі важливішим. Unity ML-Agents надає інструменти для розробки таких агентів у віртуальних середовищах, що широко використовуються в ігровій індустрії, симуляціях та дослідженнях. Використання глибоких нейронних мереж для навчання агентів значно підвищує їхню здатність до адаптації та оптимізації поведінки, що робить це дослідження актуальним для покращення систем автоматизації, моделювання та прийняття рішень.

Об’єктом дослідження є процес навчання агентів в середовищі Unity ML-Agents з використанням глибоких нейронних мереж.

Предметом дослідження є інтегрована система для навчання агентів у Unity ML-Agents, що включає застосування моделей глибокого навчання та алгоритмів підкріплювального навчання.

Метою дослідження є створення інтегрованої системи для ефективного навчання агентів в середовищі Unity ML-Agents, що використовує глибокі нейронні мережі, а також оцінка продуктивності агентів в різних сценаріях та задачах.

В результаті виконання роботи було досліджено архітектуру та процеси Unity ML-агентів, підкреслено інтеграцію середовищ моделювання Unity з

алгоритмами навчання з підкріпленням через надійну комунікаційну структуру. Конвеєр навчання, що використовує проксимальну оптимізацію політики, дозволив розробити інтелектуальних агентів, здатних вивчати складну поведінку через ітеративну взаємодію з навколишнім середовищем.

Дана робота складається з чотирьох розділів. Кожен розділ відповідно присвячений: аналізу предметної області; встановленню віртуального простору і його підключення до MLAgents; підготовці області навчання та налаштування базових параметрів і навчання самих агентів; аналізу отриманих результатів. Загальний обсяг роботи – 77 сторінок. Кваліфікаційна робота містить 6 додатків, 24 рисунки, 1 таблицю і 45 джерел посилання.

Ключові слова: Unity, ML-Agents, глибокі нейронні мережі, агенти, машинне навчання, тренування агентів, штучний інтелект.

ABSTRACT

to the qualification work by the student of thegroup 601m of Petro Mohyla Black Sea National University

Reva Vladyslav Volodymyrovych

" AN INTEGRATED SYSTEM FOR TRAINING UNITY ML AGENTS USING DEEP NEURAL NETWORK MODELS "

The relevance of this research lies in the speed of development of artificial intelligence and its application in various fields, the training of agents capable of autonomous decision–making in complex dynamic environments is becoming increasingly important. Unity ML-Agents provides tools for developing such agents in virtual environments widely used in the gaming industry, simulations, and research. Using deep neural networks to train agents significantly increases their ability to adapt and optimize behavior, making this research relevant for improving automation, simulation, and decision-making systems.

The object of the study is the process of training agents in the Unity ML-Agents environment using deep neural networks.

The subject of research is the development of an integrated system for training agents in Unity ML-Agents, which includes the application of deep learning models and reinforcement learning algorithms.

The purpose of the research is to create an integrated system for effective training of agents in the Unity ML-Agents environment, which uses deep neural networks, as well as to evaluate the performance of agents in various scenarios and tasks.

As a result of the work, the architecture and processes of Unity ML agents were investigated, emphasizing the integration of Unity modeling environments with reinforcement learning algorithms through a robust communication structure. The learning pipeline using proximal policy optimization allowed the development of

intelligent agents capable of learning complex behaviors through iterative interaction with the environment.

This paper consists of four chapters. Each chapter is devoted to: analyzing the subject area; setting up the virtual space and connecting it to MLAgents; preparing the training area and setting up the basic parameters and training the agents themselves; and analyzing the results. The total volume of the work is 77 pages. The qualification work contains 6 appendices, 22 figures, 1 table and 45 references.

Keywords: Unity, ML-Agents, deep neural networks, agents, machine learning, agent training, artificial intelligence.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 4
ВСТУП 5
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ 6
1.1 Огляд застосування ML 6
1.2 Аналіз інструментарію Unity ML-Agents Toolkit 8
1.3 Порівняння програмних аналогів для навчання агентів у віртуальних середовищах 11
1.4 Опис архітектури системи 13
1.5 Порівняння підкріплювального та імітаційного методів навчання 14
Висновоки до розділу 1 17
2 ВСТАНОВЛЕННЯ ВІРТУАЛЬНОГО ПРОСТОРУ ДЛЯ НАВЧАННЯ АГЕНТІВ UNITY 18
2.1 Основна бібліотека mlagents і mlagents_envs 18
2.2 Бібліотека PyTorch для нейромережевих обчислень 22
2.3 Бібліотека numpy для чисельних обчислень 24
2.4 Бібліотека Pillow для обробки зображень 26
2.5 Бібліотека gym для розробки та порівняння алгоритмів навчання 27
2.6 Бібліотека protobuf 29
2.7 Бібліотека tensorboard 31
Висновоки до розділу 2 33
3 ПРОЄКТУВАННЯ, РОЗРОБКА ТА ТЕСТУВАННЯ СИСТЕМИ 34
3.1 Основні налаштування простуру і агентів 34
3.2 Структура інтелектуальної системи 42
3.3 Опис агентів і конфігурації 48

3.4 Програмна реалізація 52
Висновоки до розділу 3 60
4 ДОСЛІДЖЕННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ 61
4.1 Аналіз перших отриманих результатів 61
4.2 Фінальна версія моделі 63
4.3 Складності реалізації 65
Висновоки до розділу 4 67
ВИСНОВКИ 68
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 69
ДОДАТОК А EnvironmentalAwareness 74
ДОДАТОК Б DeerAgent 76
ДОДАТОК В WolfAlphaAgent 80
ДОДАТОК Г WolfPackMateAgent 85
ДОДАТОК Ґ Hunting.yaml 91
ДОДАТОК Ґ Апробація роботи 93

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ШІ(AI) – Штучний Інтелект

ML – Machine Learning

API – Application Programming Interface

PPO – Proximal Policy Optimization

RL – Reinforcement Learning

ВСТУП

У сучасному світі стрімкий розвиток технологій штучного інтелекту відкриває нові можливості для автоматизації процесів і створення складних систем, здатних виконувати завдання, що раніше потребували людського втручання. Однією з таких технологій є навчання агентів у віртуальних середовищах, які можуть використовувати алгоритми глибокого навчання для оптимізації своєї поведінки та взаємодії з навколишнім середовищем.

Unity ML-Agents є потужною платформою, що дозволяє поєднувати середовище Unity з сучасними методами машинного навчання. Вона надає можливість створювати агентів, які можуть навчатися різноманітних завдань за допомогою алгоритмів підкріплювального навчання, і є ідеальним інструментом для дослідження та реалізації складних моделей поведінки у віртуальних середовищах. Використання глибоких нейронних мереж дає змогу агентам навчатися складних патернів та приймати оптимальні рішення в непередбачуваних умовах.

Метою даної кваліфікаційної роботи є розробка інтегрованої системи для навчання агентів Unity ML-Agents з використанням глибоких нейронних мереж. У роботі буде проведений аналіз методів підкріплювального навчання, розроблені моделі глибоких нейронних мереж, налаштовані та протестовані агенти в різних сценаріях. Це дослідження також спрямоване на оцінку ефективності таких систем, а також визначення можливостей для їх подальшого вдосконалення та розширення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Огляд застосування ML

Штучний інтелект та машинне навчання стали ключовими у трансформації ландшафту розробки ігор, пропонуючи динамічний та адаптивний досвід, який раніше був недосяжним за допомогою традиційних методів програмування. Ці технології дозволяють іграм пропонувати більш персоналізований, чутливий та захоплюючий досвід, дозволяючи персонажам, оточенню та ігровій механіці розумно реагувати на дії гравця.

Використання штучного інтелекту в цих творчих системах виходить за рамки простого розпізнавання об'єктів. Він включає в себе складну обробку зображень і методи машинного навчання, щоб забезпечити плавну інтеграцію створеного гравцем контенту з наявними в грі ресурсами. Наприклад, гравець може намалювати просту хмару, яку гра розпізнає, інтерпретує та перетворює на візуально цілісну хмару, що вписується в естетичні та функціональні рамки гри. Завдання полягає в тому, щоб навчити моделі ШІ точно інтерпретувати широкий спектр вхідних даних гравця, від простих начерків до складніших проектів, гарантуючи при цьому, що отримані в результаті ігрові об'єкти поводитимуться так, як очікується в ігровому середовищі [1].

Окрім підвищення творчого потенціалу, системи зі штучним інтелектом в іграх також сприяють більш захоплюючому та динамічному ігровому процесу. Використовуючи машинне навчання, ці системи можуть вчитися на взаємодії гравців, адаптуючись з часом, щоб забезпечити більш персоналізований і складний досвід. Наприклад, ШІ може навчитися розпізнавати стиль малювання або вподобання гравця, що дозволить грі пропонувати підказки або адаптувати рівень складності певних завдань відповідно до здібностей гравця.

Крім того, ШІ та ML дозволяють створювати більш складних неігрових персонажів (NPC) і внутрішньоігрові екосистеми, які взаємодіють з контентом,

створеним гравцями. Наприклад, у грі на виживання тварини або супротивники, керовані ШІ, можуть реагувати по-різному залежно від об'єктів або середовища, які створюють гравці. Це додає ще один рівень складності до ігрового процесу, оскільки гравці повинні враховувати, як їхні творіння вплинуть на ширший ігровий світ.

Незважаючи на ці досягнення, інтеграція ШІ та ML у творчі ігрові системи пов'язана з певними труднощами. Однією з головних проблем є забезпечення того, щоб ШІ міг точно інтерпретувати і реагувати на різноманітні вхідні дані, зберігаючи при цьому послідовний і приємний ігровий процес [2]. Це вимагає тривалого навчання моделей машинного навчання, надійних алгоритмів розпізнавання зображень і ретельного проектування поведінки ШІ, щоб уникнути непередбачуваних наслідків або розчарування гравців.

Інший виклик – обчислювальні вимоги до ШІ в режимі реального часу, що може призвести до навантаження на системні ресурси, особливо в ресурсоємних іграх. Розробники повинні знайти баланс між складністю систем, керованих ШІ, і необхідністю підтримувати плавний і чуйний ігровий процес. Це часто передбачає оптимізацію алгоритмів, використання попередньо навчених моделей і застосування ефективних методів обробки даних.

Таким чином, інтеграція ШІ та ML у творчі ігрові системи – це значний стрибок уперед у сфері інтерактивності та занурення у відеоігри. Дозволяючи гравцям впливати на ігровий світ шляхом малювання або створення об'єктів, ці технології відкривають нові можливості для творчості та персоналізації в іграх. Однак успішне впровадження цих систем вимагає ретельного розгляду технічних проблем, зокрема, навчання моделей ШІ, обробки даних у реальному часі та збалансованості ігрового процесу.

1.2 Аналіз інструментарію Unity ML-Agents Toolkit

Unity ML-Agents – це потужний інструментарій, який дозволяє розробникам впроваджувати поведінку, керовану машинним навчанням, в іграх, пропонуючи універсальну основу для створення інтелектуальних агентів, які можуть навчатися та адаптуватися до різних ігрових сценаріїв [3]. Цей інструментарій особливо цінний для розробників, які хочуть інтегрувати у свої ігри штучний інтелект, що взаємодіє з контентом, створеним гравцями, наприклад, з об'єктами, намальованими гравцем.

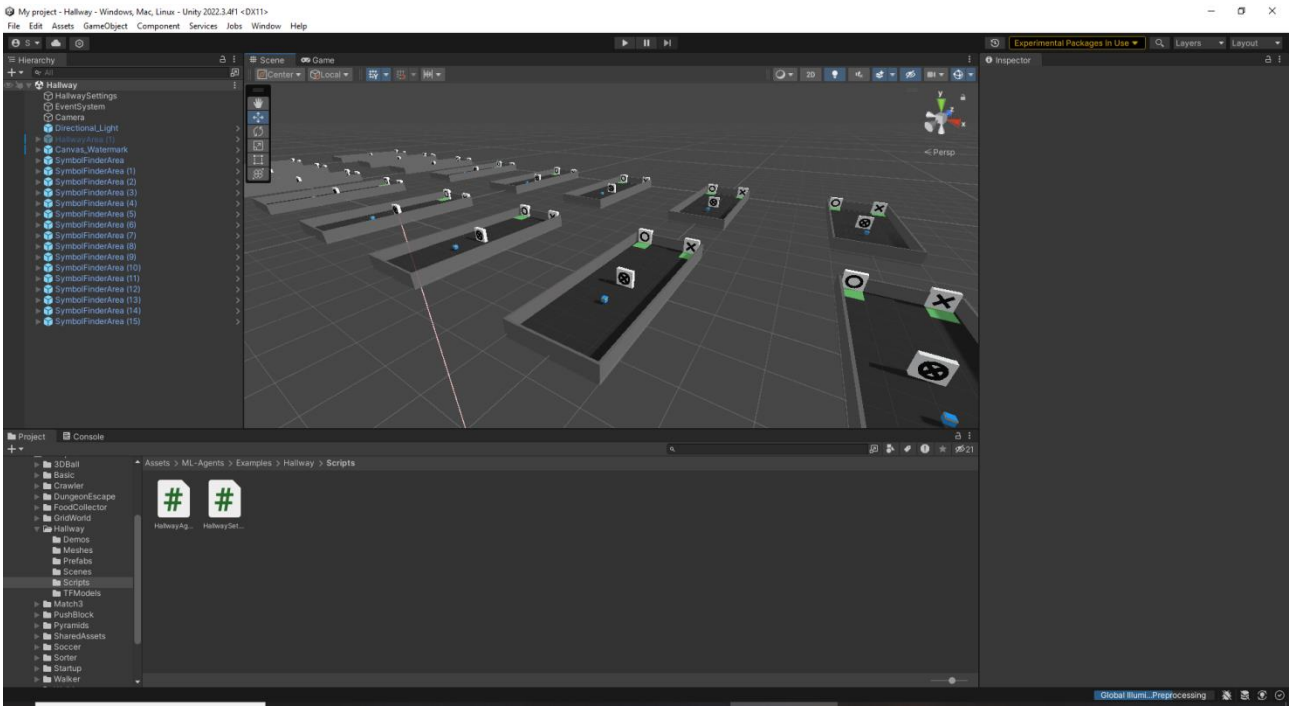


Рисунок 1.1 – Інтерфейс Unity

Інструментарій Unity ML-Agents (див. рис. 1.1) надає ряд інструментів і функцій, призначених для полегшення розробки ШІ-агентів, які можуть навчатися у своєму середовищі і з часом покращувати свою продуктивність [4]. Він використовує навчання з підкріпленням, імітаційне навчання та інші методи машинного навчання, щоб дозволити агентам виконувати складні

завдання, такі як навігація в середовищі, взаємодія з об'єктами та прийняття рішень на основі дій гравця.

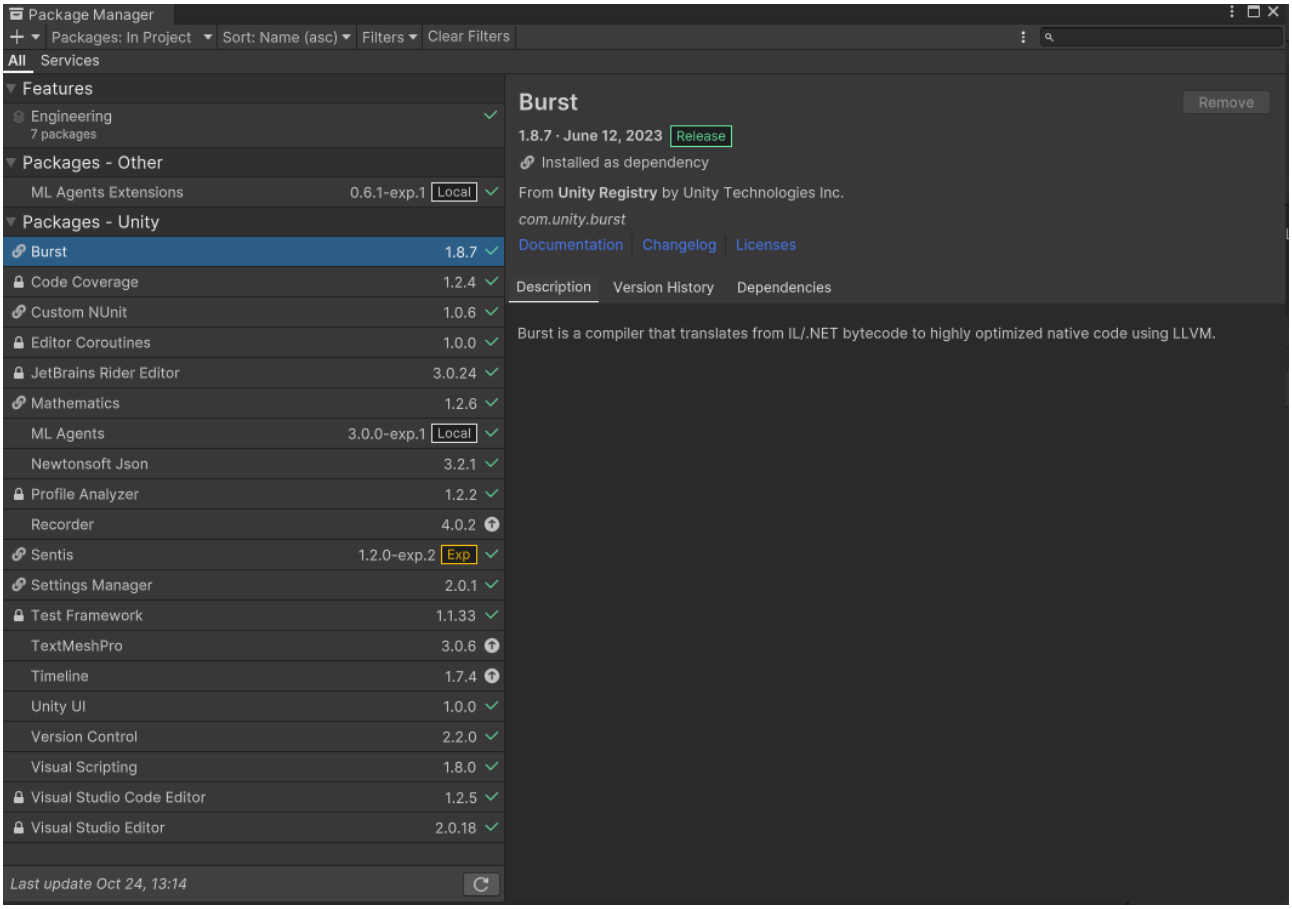


Рисунок 1.2 – Встановлені пакети для тренування агентів в Unity

Однією з ключових можливостей Unity ML-Agents є здатність створювати ШІ-агентів, які можуть динамічно реагувати на контент, створений гравцями. Наприклад, у грі, де гравці можуть малювати об'єкти, які впливають на ігровий світ, ML-агенти можна використовувати для навчання ШІ-персонажів або сутностей розпізнавати ці об'єкти та взаємодіяти з ними. Це може включати в себе навчання ШІ-персонажа пересуванню або використанню намальованих гравцем об'єктів, таких як міст або зброя, або реагування на зміни в навколишньому середовищі, спричинені творіннями гравця [5].

Гнучкість інструментарію дозволяє розробникам створювати власні навчальні середовища, в яких AI-агенти можуть вивчати певну поведінку, пов'язану з контентом, створеним гравцем. Наприклад, розробники можуть створювати сценарії, в яких агенти ШІ повинні навчитися розпізнавати різні типи об'єктів, намальованих гравцем, і відповідно адаптувати свою поведінку. Це може включати розпізнавання різниці між намальованою хмарою, яка представляє нешкідливий елемент навколишнього середовища, і намальованою грозовою хмарою, яка може викликати більш обережну поведінку персонажів, керованих ШІ.

Однак, хоча Unity ML-Agents забезпечує надійний фреймворк для розробки інтелектуальних агентів, існують обмеження, які слід враховувати при впровадженні ШІ, що взаємодіє з контентом, створеним гравцем. Однією з проблем є складність навчання ШІ-агентів, щоб вони точно інтерпретували та реагували на широкий спектр малюнків гравців [6]. Це вимагає великих навчальних даних і ретельного налаштування моделей машинного навчання, щоб ШІ міг узагальнювати навчальні приклади на основі різноманітних і непередбачуваних вхідних даних, які можуть надавати гравці.

Ще одним обмеженням є потенційні обчислювальні витрати, пов'язані з обробкою ШІ в реальному часі. Навчання та запуск моделей машинного навчання в реальному часі може бути ресурсномістким, особливо в складних ігрових середовищах, де кілька ШІ-агентів одночасно взаємодіють з контентом, створеним гравцем [7]. Розробники повинні оптимізувати використання ML-агентів, щоб збалансувати вимоги до обробки ШІ в реальному часі з необхідністю безперебійної роботи ігрового процесу.

Крім того, успіх інтеграції агентів Unity ML з контентом, створеним гравцями, залежить від якості навчальних даних та ефективності використовуваних моделей машинного навчання. Розробники повинні ретельно проектувати навчальні середовища та обирати відповідні алгоритми навчання,

щоб гарантувати, що агенти ШІ зможуть ефективно та надійно засвоїти бажану поведінку.

Незважаючи на ці виклики, Unity ML-Agents пропонує потужний набір інструментів(див. рис. 1.2) для розробників, які прагнуть розширити межі інтерактивного та динамічного ігрового процесу. Дозволяючи агентам ШІ взаємодіяти з контентом, створеним гравцями, інструментарій відкриває нові можливості для створення ігор, які є більш чуйними, персоналізованими та захоплюючими [8]. Оскільки машинне навчання продовжує розвиватися, можливості Unity ML-Agents, ймовірно, розширюватимуться, пропонуючи ще більше можливостей для інновацій у розробці ігор.

1.3 Порівняння програмних аналогів для навчання агентів у віртуальних середовищах

У процесі аналізу існуючих інструментів для навчання агентів у віртуальних середовищах було проведено порівняння кількох популярних платформ, які можуть бути використані як альтернатива Unity ML-Agents. Таблиця демонструє ключові характеристики таких інструментів, як OpenAI Gym, AirSim, CARLA, DeepMind Lab та MuJoCo (у табл. 1.1).

Unity ML-Agents вирізняється серед конкурентів завдяки інтеграції з потужною графічною платформою Unity, що дозволяє створювати агентів у реалістичних 3D середовищах [9]. Це робить його універсальним інструментом для навчання агентів різним типам завдань, від простої навігації до складної взаємодії з оточенням.

Інші інструменти, такі як OpenAI Gym, фокусуються на двовимірних симуляціях і мають простіші середовища, але пропонують широкий вибір класичних задач підкріплювального навчання. AirSim і CARLA, у свою чергу,

спеціалізуються на симуляціях автопілотів і транспорту, забезпечуючи високу реалістичність, але мають обмежене застосування для інших типів задач.

Таким чином, Unity ML-Agents є ідеальним вибором для проєктів, де важлива інтеграція складних моделей поведінки агентів у різноманітних ігрових або симуляційних середовищах, забезпечуючи гнучкість і візуальну привабливість.

Таблиця 1.1 – Порівняння програмних аналогів

Назва інструменту	**Основні можливості**	**Типи середовищ**	**Алгоритми навчання**	**Переваги**	**Недоліки**
Unity ML-Agents	Інтеграція агентів у 3D–середовищах Unity	3D середовища Unity	Підкріплювальне, імітаційне навчання	Потужна графіка, велика гнучкість, візуалізація	Велике навантаження на ресурси
OpenAI Gym	Широкий набір RL середовищ	2D симуляції, класичні RL задачі	Підкріплювальне навчання, навчання політики	Стандартизований API, велика кількість середовищ	Базова візуалізація, немає повної 3D підтримки
AirSim (Microsoft)	Симуляція автопілота дронів і автомобілів	Реалістичні симуляції транспорту	Підкріплювальне, імітаційне навчання	Реалістичні фізичні моделі, підтримка зовнішніх сенсорів	Високі вимоги до обладнання
CARLA	Симуляція автономного керування автомобілями	Реалістичні міські середовища	Підкріплювальне навчання	Висока реалістичність, детальні карти	Орієнтований на вузькі завдання (автокерування)
DeepMind Lab	Навчання агентів у 3D середовищах	3D лабіринти, навчальні сценарії	Підкріплювальне навчання	Оптимізовані середовища для дослідження AI	Менш гнучкий у порівнянні з Unity ML-Agents
MuJoCo	Фізично точна симуляція роботів	2D/3D середовища, роботи	Підкріплювальне навчання	Точна симуляція фізичних об'єктів	Вузька спеціалізація на робототехніці

1.4 Опис архітектури системи

Архітектура системи, що використовує Unity ML-Agents, базується на взаємодії кількох компонентів, які забезпечують ефективне навчання агентів у віртуальних середовищах. Основні компоненти системи включають середовище Unity, ML-агенти, бібліотеки для машинного навчання (наприклад, PyTorch або TensorFlow), а також алгоритми підкріплювального навчання [10].

Середовище Unity Unity — це графічна платформа, яка використовується для створення реалістичних 2D та 3D середовищ, де агенти взаємодіють з об’єктами, навчаються виконувати різні завдання та адаптуються до змінних умов. В Unity створюються сценарії з необхідними параметрами, а також налаштовуються об'єкти, з якими агент може взаємодіяти.

ML-агенти Unity ML-Agents Toolkit забезпечує міст між середовищем Unity та алгоритмами машинного навчання. Він надає API, який дозволяє агентам отримувати інформацію про середовище та надсилати свої дії у відповідь. ML-агенти використовують підкріплювальне навчання, де агенти отримують винагороду за досягнення певних цілей і на основі цього навчаються покращувати свою поведінку.

Алгоритми підкріплювального навчання Для навчання агентів використовуються алгоритми підкріплювального навчання (наприклад, Proximal Policy Optimization (PPO) або Soft Actor-Critic (SAC)). Ці алгоритми дозволяють агентам поступово вдосконалювати свою поведінку на основі досвіду, отриманого в результаті взаємодії з середовищем.

Python API та бібліотеки для навчання Python API надає інтерфейс між Unity і платформами машинного навчання, такими як PyTorch або TensorFlow. Ці бібліотеки відповідають за обчислення та навчання нейронних мереж, які використовуються агентами для прийняття рішень. Python API отримує дані від

Unity, обробляє їх за допомогою моделей нейронних мереж, та повертає відповідні дії агентам [11].

Взаємодія між компонентами Архітектура системи працює за наступним циклом:

– агент у середовищі Unity отримує інформацію про стан середовища через сенсори (наприклад, позицію об'єктів);

– ці дані передаються через Python API до моделі машинного навчання;

– модель нейронної мережі виводить рішення, яке дію має виконати агент;

– дія агента повертається назад у середовище Unity, де агент виконує цю дію і отримує оновлену винагороду або нові умови;

– процес повторюється доти, доки агент не навчиться оптимальній стратегії для досягнення цілі.

Завдяки такій архітектурі система дозволяє створювати агентів, які можуть ефективно навчатися та вдосконалювати свої навички в різноманітних ігрових та симуляційних середовищах.

1.5 Порівняння підкріплювального та імітаційного методів навчання

Unity ML-Agents використовується кілька методів навчання агентів, серед яких найпоширенішими є підкріплювальне навчання та імітаційне навчання. Обидва ці методи мають свої переваги і недоліки, а також різні сфери застосування, що визначає їх ефективність у певних завданнях [12].

Підкріплювальне навчання (Reinforcement Learning, RL) — це метод, у якому агент навчається шляхом взаємодії зі своїм середовищем. Він отримує винагороди або покарання за свої дії і намагається максимізувати свою загальну винагороду за допомогою випробувань та помилок. Це дозволяє агенту

досліджувати середовище і знаходити оптимальні стратегії для досягнення певних цілей.

Переваги:

– гнучкість: RL підходить для широкого спектра задач, оскільки агент навчається взаємодіяти зі складними і динамічними середовищами;

– автономність: агент самостійно досліджує середовище без необхідності заздалегідь підготовлених даних;

– ефективність у довготривалих завданнях: RL дозволяє агентам ухвалювати послідовні рішення, що робить його корисним для завдань з довготривалою стратегією.

Недоліки:

– високі обчислювальні витрати: процес навчання через підкріплення може займати багато часу і потребує великих обчислювальних ресурсів.

– нестабільність навчання: через залежність від випадкових взаємодій зі середовищем агент може зіткнутися з проблемами, такими як надмірне чи недостатнє навчання.

Імітаційне навчання (Imitation Learning, IL) – передбачає навчання агента на основі демонстрацій від експерта. Агент повторює дії експерта, спостерігаючи за тим, як експерт виконує завдання, і намагається їх відтворити. Це дозволяє швидко навчити агента виконувати складні завдання, якщо є доступ до великої кількості прикладів правильних дій [13].

Переваги:

– швидке навчання: агенту не потрібно самостійно досліджувати середовище, тому він може навчитися швидше, маючи доступ до якісних демонстрацій.

– стабільність: навчання на базі експертних даних знижує ймовірність помилок і випадкових дій агента.

Недоліки:

– залежність від демонстрацій: якість навчання агента повністю залежить від якості і кількості експертних прикладів. Якщо демонстрації не охоплюють всі можливі ситуації, агент може не навчитися правильній поведінці.

– обмежена адаптивність: агент може погано справлятися з новими, незнайомими ситуаціями, яких не було у демонстраціях.

Підкріплювальне навчання краще підходить для завдань, де агенту потрібно самостійно досліджувати складне середовище і адаптуватися до нових умов. Воно ефективне в ситуаціях, коли немає готових експертних даних, і агент повинен знайти оптимальну стратегію через випробування і помилки.

Імітаційне навчання корисне, коли є доступ до даних експертних демонстрацій, що дозволяє швидко навчити агента. Однак його застосування обмежене у випадках, коли середовище змінюється або з’являються нові ситуації, до яких агент не був підготовлений.

У реальних проєктах найчастіше використовують комбінацію обох підходів, де агент спочатку навчається через імітацію, а потім удосконалює свою поведінку через підкріплювальне навчання [14]. Така гібридна модель дозволяє досягти швидкого старту навчання і високої адаптивності в складних середовищах.

Висновки до розділу 1

Застосування машинного навчання та штучного інтелекту в іграх відкриває нові горизонти для створення динамічних і адаптивних ігрових систем. Ці технології дозволяють розробникам інтегрувати складні, інтелектуальні рішення в ігровий процес, що робить взаємодію гравців із грою більш персоналізованою та захоплюючою. Огляд можливостей Unity ML-Agents показує, що цей інструментарій є потужним засобом для впровадження поведінки, керованої машинним навчанням, у ігри, що взаємодіють із контентом, створеним гравцями.

Попри значні переваги, такі як здатність створювати агентів, які навчаються та адаптуються, існують і виклики, пов'язані з точним інтерпретуванням вхідних даних гравця та значними обчислювальними витратами. Однак, оптимізуючи процеси та ретельно підходячи до навчання моделей, розробники можуть ефективно вирішити ці проблеми. У результаті, інтеграція ШІ в ігри за допомогою таких інструментів, як Unity ML-Agents, відкриває нові можливості для інновацій у сфері розробки ігор, роблячи ігровий процес більш інтерактивним та унікальним для кожного гравця.

2 ВСТАНОВЛЕННЯ ВІРТУАЛЬНОГО ПРОСТОРУ ДЛЯ НАВЧАННЯ АГЕНТІВ UNITY

2.1 Основна бібліотека mlagents і mlagents_envs

ML-Agents та mlagents_envs є частиною набору інструментів Unity для інтеграції машинного навчання в середовище Unity [15].

ML-Agents (Machine Learning Agents) – це платформа з відкритим кодом, розроблена Unity Technologies, яка дозволяє навчати і тестувати інтелектуальні агенти на основі машинного навчання в ігрових середовищах Unity. Вона є потужним інструментом для дослідження штучного інтелекту (ШІ) та ігрового дизайну, а також для створення симуляційних моделей реальних фізичних або логічних систем.

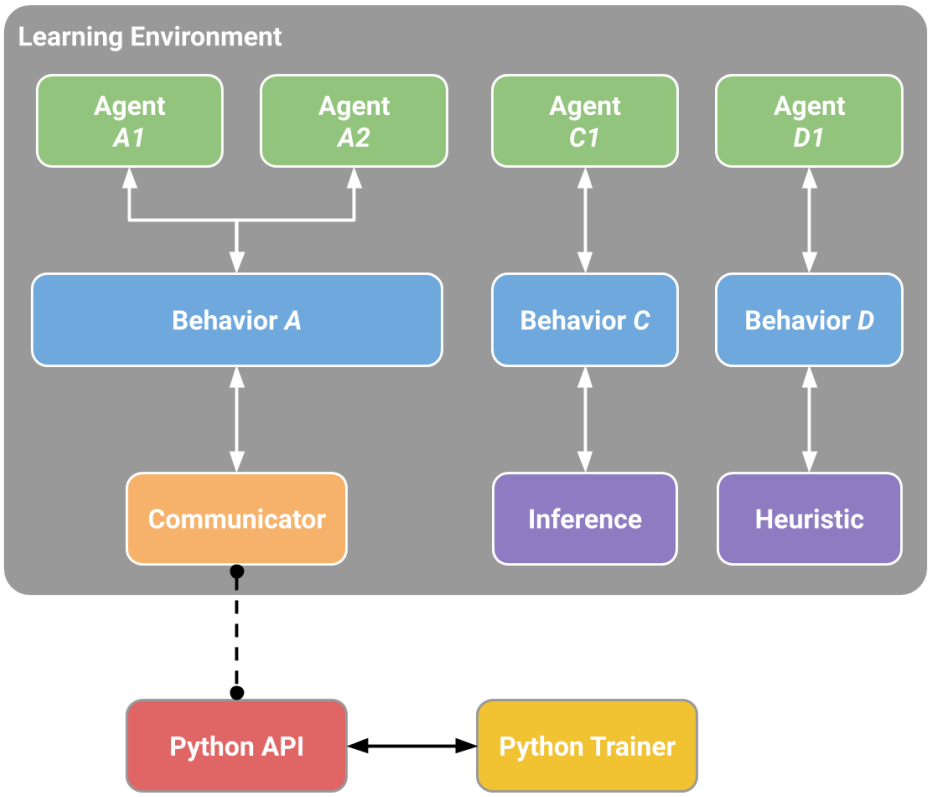


Рисунок 2.1 – Приклад блок-схеми ML-Agents Toolkit

Основні компоненти ML-Agents (див. рис. 2.1).

Unity SDK – це набір засобів, що дозволяє інтегрувати поведінку агентів у ігрові об’єкти Unity. Він включає в себе скрипти для взаємодії з середовищем, агентів та моделі спостереження.

ML-Agents Python API – дозволяє взаємодіяти з Unity середовищем через Python, забезпечуючи навчання агентів за допомогою алгоритмів машинного навчання. Python API працює як міст між Unity і середовищем машинного навчання.

Мережі для навчання – ML-Agents використовують TensorFlow або PyTorch для побудови і тренування моделей. Також надаються попередньо налаштовані моделі, які можна використовувати для швидкого старту.

ML-Agents Toolkit підтримує різні алгоритми навчання з підкріпленням, включаючи Proximal Policy Optimization (PPO) та Soft Actor-Critic (SAC). Надає API Python для навчання та взаємодії з середовищем Unity. Це включає спілкування з середовищем, надсилання дій та отримання зауважень і винагород [16]. Також пропонує інструменти для візуалізації поведінки агента та прогресу навчання.

mlagents_envs – це пакет Python, який надає основну функціональність для взаємодії з середовищами Unity. Він є частиною ML-Agents Toolkit і включає:

– взаємодіє з середовищами Unity, керуючи комунікацією між Python та Unity. Сюди входить надсилання дій агентам та отримання зауважень і винагород;

– надає API для налаштування та керування середовищами, збору даних та обробки спостережень;

– дозволяє конфігурувати середовища, включаючи налаштування параметрів і керування епізодами.

Основні функції бібліотеки ML-Agents:

– підтримка кількох агентів: в одному середовищі можна запускати багато агентів одночасно, що дозволяє тренувати кооперативну або конкурентну поведінку;

– підтримка кількох мозків: можна визначити кілька різних мозків (brain) для різних агентів, які будуть використовувати різні методи навчання або моделі;

– підтримка кількох дій та спостережень: ML-Agents дозволяють створювати як дискретні, так і неперервні простори дій та спостережень, що дозволяє моделювати різноманітні ігрові ситуації;

– алгоритми навчання: ML-Agents підтримують кілька алгоритмів навчання, включаючи підкріплювальне навчання (PPO, SAC), імітаційне навчання та інші сучасні техніки навчання;

– візуалізація навчання: Unity дозволяє візуалізувати процес навчання агентів у реальному часі, що дуже корисно для розробки ігор та моделювання;

– додаткові інструменти: Unity ML-Agents надає різноманітні інструменти для спрощення процесу навчання агентів, включаючи засоби для налаштування гіперпараметрів, моніторингу навчання та запису результатів.

Бібліотека mlagents_envs.

mlagents_envs – це важлива частина ML-Agents, яка відповідає за взаємодію з середовищами, в яких працюють агенти. Це своєрідний місток між Python API і Unity середовищем [17]. Бібліотека дозволяє створювати і взаємодіяти зі середовищами, які можуть бути досить складними з точки зору ігрової логіки та фізики.

Основні функції mlagents_envs включають:

– створення середовищ: бібліотека дозволяє створювати нові середовища Unity та взаємодіяти з ними з Python. Середовище може мати багато агентів з різними наборами спостережень і дій;

– взаємодія з агентами: за допомогою mlagents_envs користувач може отримувати спостереження агентів, передавати їм дії і контролювати, як вони навчаються у відповідь на різні стимули;

– інтерфейс Gym: бібліотека підтримує стандарт OpenAI Gym, що дозволяє легко підключати ML-Agents до інших середовищ машинного навчання;

– гнучкість: mlagents_envs дозволяє налаштовувати середовище, додаючи нові спостереження та дії, змінювати фізичні параметри світу та поведінку агентів.

Практичне використання ML-Agents і mlagents_envs.

Для роботи з ML-Agents необхідно пройти кілька етапів:

– налаштування середовища Unity: розробник створює або налаштовує існуюче ігрове середовище в Unity. Середовище повинно містити агентів – об'єкти, які будуть навчатися або виконувати певні дії;

– додавання компонентів ML-Agents: у кожного агента додається скрипт, що описує його поведінку, спостереження та дії. Також агенту необхідно надати "brain" – компонент, що відповідає за прийняття рішень [18];

– налаштування Python API: на стороні Python користувач налаштовує скрипти для навчання агентів. Використовуючи бібліотеку mlagents_envs, створюється взаємодія з Unity середовищем. Python API може використовувати різноманітні алгоритми навчання, такі як підкріплювальне навчання або імітаційне навчання;

– навчання агентів: агенти починають взаємодіяти зі своїм середовищем, збирати спостереження і отримувати винагороди або покарання. Залежно від налаштувань, агенти можуть навчатися на основі своїх помилок, оптимізуючи свою поведінку для досягнення найкращих результатів;

– оцінка результатів: після завершення навчання агенти можуть бути протестовані на нових завданнях або в нових середовищах. Розробники можуть

також використовувати візуалізацію та інші засоби Unity для аналізу результатів навчання.

Переваги ML-Agents:

– простота використання: ML-Agents дуже добре інтегрований з Unity, що дозволяє легко почати навчання агентів;

– масштабованість: ML-Agents підтримує великі симуляції з багатьма агентами та складними середовищами;

– гнучкість: ML-Agents може використовувати різні алгоритми машинного навчання для тренування агентів і створювати різноманітні типи взаємодії між агентами [19];

– велика спільнота: ML-Agents підтримується активною спільнотою, що надає можливість отримати швидку допомогу та використовувати багато ресурсів.

Недоліки ML-Agents:

– обмежена підтримка інших мов: хоча Python – одна з найпопулярніших мов для машинного навчання, обмежена підтримка інших мов може стати проблемою для деяких розробників;

– вимоги до ресурсів: навчання агентів у складних середовищах може вимагати значних обчислювальних ресурсів.

2.2 Бібліотека PyTorch для нейромережевих обчислень

PyTorch – це потужна, гнучка та популярна бібліотека для глибокого навчання, яка підтримує динамічні обчислювальні графіки, ефективні тензорні операції та повний набір інструментів для машинного навчання [20]. Вона добре інтегрується з Unity ML-Agents для розробки інтелектуальних агентів і моделей в середовищах Unity.

PyTorch використовує динамічні графіки обчислень (eager execution), що означає, що граф будується «на льоту» по мірі виконання операцій. Це полегшує налагодження та експерименти у порівнянні зі статичними графами обчислень, що використовуються у деяких інших фреймворках [21].

PyTorch надає потужну тензорну бібліотеку, подібну до NumPy, але з прискоренням на GPU. Тензори – це багатовимірні масиви, які можна обробляти на CPU або GPU.

PyTorch містить бібліотеку автоматичного диференціювання autograd, яка підтримує оптимізацію на основі градієнта і дозволяє легко обчислювати градієнти для зворотного розповсюдження та надає багатий набір готових нейромережевих модулів і шарів, які можна використовувати для побудови різних архітектур, таких як CNN, RNN і трансформатори.

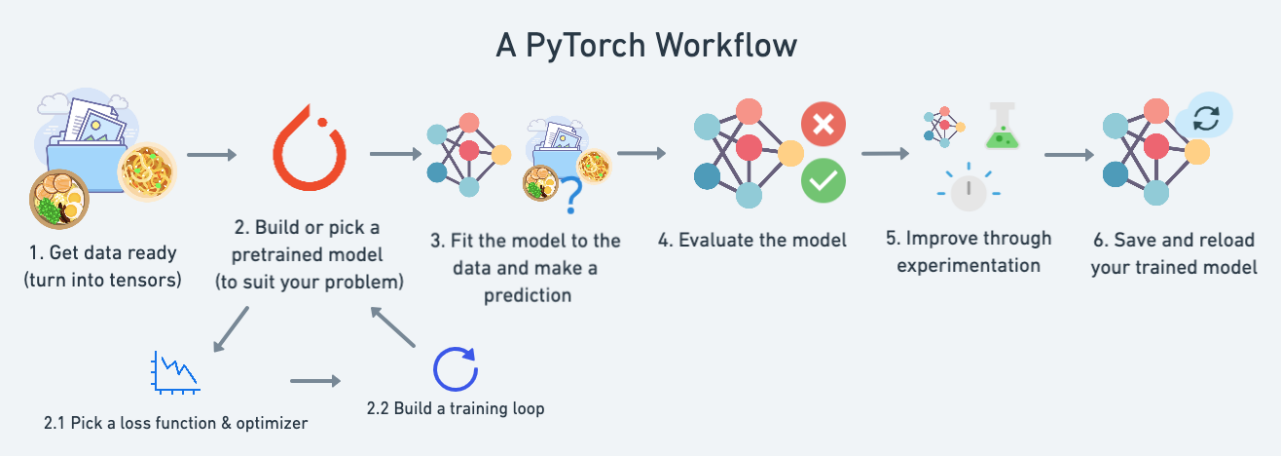


Рисунок 2.2 – Основа робочого процесу PyTorch

Бібліотека містить реалізації різних алгоритмів оптимізації, таких як SGD, Adam та RMSprop, які необхідні для навчання нейронних мереж. PyTorch має утиліти для ефективної обробки та завантаження даних, включаючи інструменти для доповнення даних та паралельної обробки.

Моделі PyTorch можна експортувати та розгортати у виробництві за допомогою TorchScript, який забезпечує спосіб серіалізації та оптимізації моделей для розгортання у виробничому середовищі [22].

Я використав PyTorch разом з Unity ML-Agents Toolkit для навчання та оцінки моделей машинного навчання в середовищах Unity (див. рис. 2.2). Хоча ML-Agents переважно використовував Python для процесу навчання та інтеграції, гнучкість PyTorch і його підтримка динамічних обчислювальних графів робили його популярним вибором для розробки та навчання моделей.

2.3 Бібліотека numpy для чисельних обчислень

NumPy (Numerical Python) – це бібліотека для чисельних обчислень мовою Python. Вона забезпечує підтримку великих багатовимірних масивів і матриць, а також набір математичних функцій для роботи з цими масивами. Вона широко використовується в наукових обчисленнях, аналізі даних та машинному навчанні.

Основною особливістю NumPy є об'єкт ndarray, потужний N-вимірний масив, який підтримує широкий спектр математичних операцій. Цей об'єкт масиву дозволяє ефективно зберігати та маніпулювати великими наборами даних.

NumPy містить багатий набір математичних функцій для роботи з масивами, таких як лінійна алгебра, статистика та перетворення Фур'є [23]. Ці функції оптимізовані для продуктивності і можуть ефективно обробляти складні обчислення.

Трансляція – це потужна функція, яка дозволяє NumPy виконувати операції над масивами різної форми інтуїтивно зрозумілим та ефективним способом. Це дозволяє уникнути необхідності в явних циклах і робить код чистішим і швидшим.

NumPy часто використовується разом з іншими бібліотеками, такими як SciPy, Pandas та Matplotlib. Вона слугує базовою бібліотекою для цих інструментів, забезпечуючи спільну структуру даних та операцій.

NumPy реалізовано на мові C, що дозволяє йому досягати високої продуктивності для чисельних обчислень. Він використовує оптимізовані процедури для роботи з масивами, що робить його придатним для аналізу великих обсягів даних та математичних обчислень.

NumPy надає функції для зміни форми, нарізки та індексації масивів, які є важливими для попередньої обробки даних та маніпуляцій з ними [24]. Бібліотека містить інструменти для генерації випадкових чисел, що корисно для таких завдань, як ініціалізація ваг у нейронних мережах або створення синтетичних даних.

В контексті Unity ML-Agents NumPy відіграє допоміжну роль в аналізі даних та маніпуляціях з ними під час навчання та оцінки моделей машинного навчання. NumPy можна використовувати для попередньої обробки даних, зібраних з середовищ Unity, перед тим, як завантажувати їх у моделі машинного навчання. Це включає нормалізацію, перетворення та аналіз даних. При взаємодії з середовищами Unity, спостереження та винагороди можна обробляти та аналізувати за допомогою NumPy. Це допомагає зрозуміти дані і внести корективи в навчальний процес [25].

Масиви NumPy використовується з такими бібліотеками, як Matplotlib, для візуалізації метрик навчання, статистики продуктивності або будь-яких інших відповідних даних.

Оскільки PyTorch і NumPy часто використовуються разом, масиви NumPy можно конвертувати в тензори PyTorch і навпаки. Це корисно при виконанні маніпуляцій з даними в NumPy і навчанні моделей в PyTorch.

2.4 Бібліотека Pillow для обробки зображень

Pillow – популярна бібліотека Python для обробки зображень. Це сучасний форк бібліотеки зображень Python Imaging Library (PIL), що надає прості у використанні методи для відкриття, маніпулювання та збереження різних форматів зображень. Pillow широко використовується в таких задачах, як редагування зображень, перетворення форматів і попередня обробка даних.

Pillow підтримує різні формати зображень, включаючи JPEG, PNG, GIF, BMP і TIFF. Це дозволяє відкривати зображення з файлів або URL-адрес і зберігати їх у різних форматах.

Може змінювати розмір зображень до різних розмірів зі збереженням якості. Обрізання областей, що цікавлять, із зображень. Обертання та перевертання зображень [26]. Застосування фільтрів, таких як розмиття, підвищення різкості та покращення країв.

Pillow надає можливості для малювання фігур, тексту та інших графічних елементів на зображеннях, включаючи додавання анотацій, рамок і користувацької графіки. Налаштування властивостей кольору, конвертація між колірними режимами (наприклад, RGB, відтінки сірого) та виконання таких операцій, як регулювання яскравості та контрастності.

Інструменти Pillow також дозволяють покращувати якість зображень, зокрема регулювати різкість, яскравість і контрастність. Перетворення зображень між різними форматами та режимами, наприклад, RGB у відтінки сірого, також є частиною функціоналу.

У контексті Unity ML-Agents Pillow є корисним для попередньої обробки та маніпулювання зображеннями, отриманими з середовищ Unity.

Перед завантаженням зображень у моделі машинного навчання може знадобитися їх попередня обробка, яка включає зміну розміру, обрізання або

перетворення в відтінки сірого [27]. Pillow надає інструменти для ефективного виконання цих завдань.

Pillow також може бути використаний для доповнення зображень для навчання, включаючи застосування таких перетворень, як обертання, перевертання та коригування кольору для створення варіацій навчальних даних.

Крім того, Pillow дозволяє створювати візуалізації навчальних даних, спостережень або результатів моделювання, що сприяє розумінню продуктивності моделей машинного навчання та виявленню проблем з налагодженням.

У випадку, коли зображення з Unity мають формат, який не підтримується безпосередньо іншими бібліотеками, Pillow може бути використаний для перетворення їх у відповідний формат для подальшої обробки.

2.5 Бібліотека gym для розробки та порівняння алгоритмів навчання

Gym – це інструментарій з відкритим вихідним кодом для розробки та порівняння алгоритмів навчання з підкріпленням (RL). Він надає набір середовищ для тестування агентів RL і широко використовується в дослідницькій спільноті для бенчмаркінгу та експериментів [28]. Gym був розроблений OpenAI і покликаний полегшити розробку алгоритмів RL шляхом надання стандартизованого інтерфейсу для різних середовищ.

Gym надає узгоджений API для різних середовищ. Ця стандартизація полегшує розробку алгоритмів RL та порівняння їхньої продуктивності в різних середовищах. Дозволяє створювати власні середовища, якщо вбудовані не відповідають вашим потребам. Для цього потрібно створити підкласи класу Env та реалізувати необхідні методи.

Gym можна використовувати з популярними бібліотеками RL, такими як Stable Baselines3, RLlib та TensorFlow Agents. Він також добре інтегрується з інструментами та бібліотеками візуалізації.

Надаючи стандартизований набір середовищ і тестів, Gym полегшує оцінку і порівняння різних алгоритмів RL. Це допомагає зрозуміти, які алгоритми працюють найкраще за певних умов.

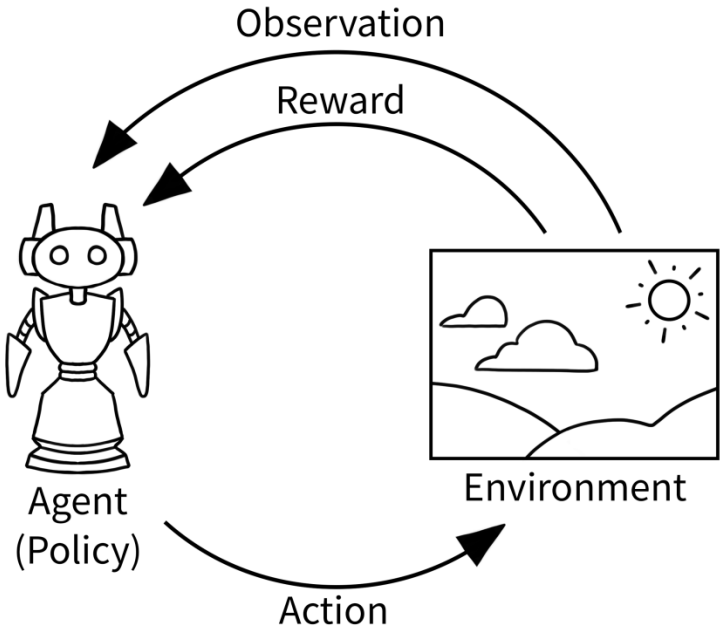


Рисунок 2.3 – Взаємодія з навколишнім середовищем

Gym і Unity ML-Agents служать для схожих цілей, але в різних контекстах. Unity ML-Agents фокусується на інтеграції машинного навчання з середовищами Unity, в той час як Gym надає ширший спектр середовищ для RL-досліджень [29].

Створення середовищ Unity, сумісних з API Gym, можливе через реалізацію обгортки, сумісної з Gym (див. рис. 2.3). Це дозволяє використовувати інструменти та бібліотеки Gym для навчання RL-агентів у середовищах Unity.

Алгоритми RL, які були протестовані в середовищах Gym, можуть бути розширені або адаптовані для роботи з середовищами Unity ML-Agents. Це забезпечує можливість роботи з більш складними або реалістичними сценаріями.

Моделі, навчені в середовищі Gym, можуть бути адаптовані або перенесені в середовище Unity ML-Agents. Для цього необхідно переконатися, що модель може обробляти спостереження та дії, які надає середовище Unity.

2.6 Бібліотека protobuf

Protocol Buffers (Protobuf) – це формат серіалізації даних з мовною діагностикою, розроблений компанією Google. Він розроблений, щоб бути ефективним як з точки зору швидкості, так і з точки зору розміру, що робить його популярним вибором для зв'язку між сервісами та для зберігання даних у компактному форматі. Protobuf широко використовується в різних додатках, включаючи мережевий зв'язок, конфігураційні файли та зберігання даних [30].

Protobuf забезпечує компактний двійковий формат для кодування даних. Це робить його швидшим та ефективнішим з точки зору зберігання та передачі даних у порівнянні з текстовими форматами, такими як JSON або XML. Структури даних визначаються за допомогою схеми, записаної у .proto-файлі. Ця схема визначає поля та їхні типи, що гарантує відповідність даних попередньо визначеній структурі.

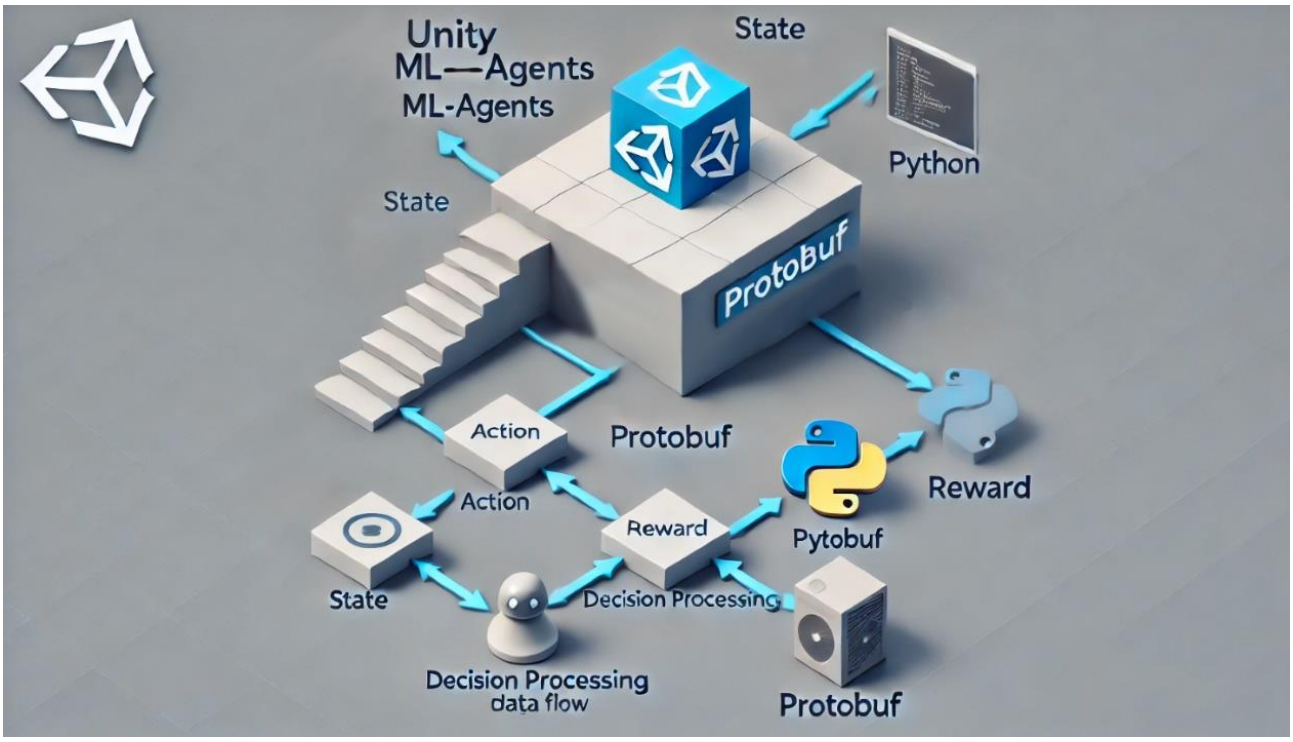


Рисунок 2.4 – Візуалізації процесу комунікації між ML-Agents та Python з використанням protobuf

Protobuf підтримує широкий спектр мов програмування, включаючи C++, Java, Python, Go та інші. Це забезпечує безперешкодну інтеграцію між різними платформами та мовами.

Protobuf дозволяє розвивати структури даних з часом. Нові поля можна додавати без порушення зворотної сумісності, а старі поля можна застарівати або видаляти, зберігаючи сумісність з новими версіями [31]. Використовує схему для генерації вихідного коду на потрібних мовах програмування. Цей згенерований код включає класи для кодування та декодування повідомлень, а також для обробки валідації повідомлень.

Protobuf можна використовувати для визначення та керування конфігураціями для середовищ Unity або моделей ML (див. рис. 2.4). Це може включати визначення складних структур даних у компактному форматі. Для обміну даними між Unity та іншими сервісами або компонентами (наприклад,

сервером або іншим додатком) Protobuf надає ефективний та надійний спосіб серіалізації та десеріалізації даних.

При інтеграції Unity з іншими системами або платформами, що використовують Protobuf, такими як внутрішні сервіси або зовнішні API, Protobuf забезпечує ефективний обмін та обробку даних [32].

2.7 Бібліотека tensorboard

Рисунок 2.5 – Використання TensorBoard для спостереження за навчанням

TensorBoard – це потужний інструмент візуалізації, який постачається разом з TensorFlow, щоб допомогти розробникам зрозуміти, налагодити та оптимізувати свої моделі машинного навчання (див. рис. 2.5). Він надає набір інтерактивних візуалізацій та метрик для відстеження продуктивності та поведінки ваших моделей під час навчання та оцінювання [33].

Ключові особливості TensorBoard:

– візуалізує скалярні показники, такі як втрати та точність з часом. Це допомагає відстежувати прогрес навчання та діагностувати такі проблеми, як надмірне або недостатнє налаштуванняl;

– відображає обчислювальний графік вашої моделі, що показує структуру нейронної мережі. Це допомагає зрозуміти і налагодити архітектуру моделі;

– будує гістограми тензорів, таких як ваги та зсуви, щоб спостерігати за їх розподілом у часі. Це корисно для аналізу того, як тензори змінюються під час навчання;

– візуалізує розподіл значень тензорів у часі. Це може допомогти зрозуміти поведінку ваг і градієнтів під час навчання;

– відображає зображення та інші візуальні дані. Це корисно для візуалізації входів, виходів і проміжних активацій даних;

– дозволяє візуалізувати текстові дані та інші типи інформації. Це може бути корисно для реєстрації та візуалізації текстових даних;

– візуалізує багатовимірні вбудовування за допомогою таких методів, як t-SNE або PCA. Це допомагає зрозуміти, як різні точки даних згруповані у просторі ознак;

– забезпечує 3D-візуалізацію високорозмірних даних, таких як вбудовування, для дослідження взаємозв'язків і кластерів;

– дозволяє створювати власні скалярні та графічні візуалізації, які можна адаптувати до конкретних потреб.

Висновки до розділу 2

Використання різних бібліотек, таких як ML-Agents, PyTorch, NumPy, Pillow, Gym, Protobuf та TensorBoard, значно спрощує процес навчання агентів в Unity ML Agents. Ці інструменти забезпечують розробникам гнучкість у налаштуванні та оптимізації моделей, що дозволяє створювати більш реалістичні та ефективні середовища для навчання з підкріпленням. Комбінація цих технологій дозволяє досягти високої продуктивності та точності при розробці інтелектуальних агентів, що взаємодіють у середовищах Unity.

3 ПРОЄКТУВАННЯ, РОЗРОБКА ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Основні налаштування простуру і агентів

Agent — це ключовий компонент у Unity ML-Agents, який представляє собою розумного агента, що навчається взаємодіяти з оточенням. Агент є основним "активним" елементом у системі машинного навчання і відповідає за прийняття рішень, спостереження за середовищем і виконання дій [34]. Завдяки агентам реалізується взаємодія між середовищем і моделлю навчання.

Агент збирає дані про оточення через сенсори (або безпосередньо з навколишніх об'єктів) і перетворює їх у вхідні дані для нейронної мережі. Ці дані можуть включати:

- позицію, швидкість чи напрямок руху об'єктів;
- інформацію з Ray Perception Sensor, камер чи інших джерел;
- значення із середовища, які передаються агенту вручну.

На основі отриманих спостережень агент виконує дії. Дії визначають, як агент взаємодіє з середовищем:

- дискретні дії (наприклад, рух уперед або обертання вліво);
- неперервні дії (наприклад, поворот на певний кут чи зміна швидкості).

Ці дії задаються моделлю на основі поточного стану агента.

Щоб навчити агента досягати цілей, йому призначаються нагороди. Позитивні нагороди за бажану поведінку (наприклад, досягнення цілі). Негативні нагороди за небажані дії (наприклад, зіткнення з перешкодою). Нагороди дозволяють агенту навчатися через методи підкріплення (reinforcement learning).

Агент проходить кілька ключових етапів у процесі роботи:

- Initialize – початкова ініціалізація агента;
- CollectObservations – збір даних про середовище;

– OnActionReceived – отримання дій від моделі та їх виконання;

– Heuristic – реалізація дій вручну (наприклад, для тестування або навчання);

– EndEpisode – завершення епізоду (наприклад, якщо агент досяг мети чи зазнав невдачі).

Агент працює в межах середовища Unity, яке виступає в ролі симуляції для навчання. Всі об'єкти середовища можуть взаємодіяти з агентом через фізику, колізії чи пряме спостереження [35]. Щоб додати агента до сцени, необхідно призначити компонент Agent до об'єкта Unity. Цей компонент є основою для створення користувацької логіки агента.

У компоненті Agent можна налаштувати Behavior Parameters, який визначає, як агент навчається (дискретні чи неперервні дії, кількість спостережень тощо), Max Step задає максимальну кількість кроків у кожному епізоді та Decision Requester задає частоту прийняття рішень агентом.

Agent є центральною частиною Unity ML-Agents, яка поєднує механіку Unity з можливостями навчання підкріплення. Він дозволяє створювати розумних персонажів і автоматизовані системи, які можуть адаптуватися до змін у середовищі.

Rigidbody і Box Collider – це компоненти фізики в Unity, які дозволяють створювати реалістичні рух і взаємодії між об'єктами. Вони часто використовуються разом для налаштування фізичної поведінки об'єктів у 3D–просторі.

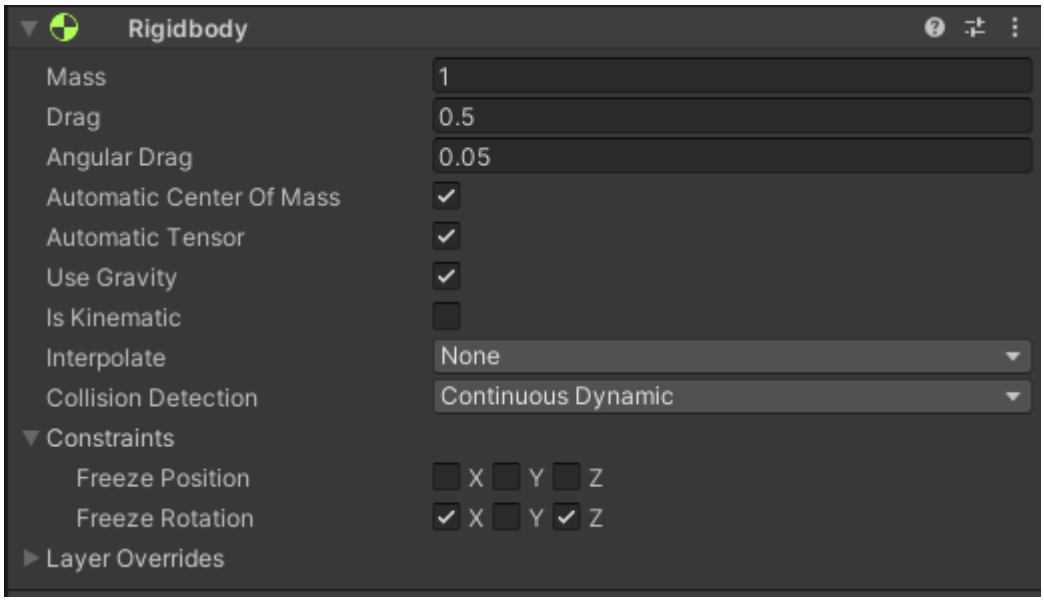


Рисунок 3.1 – Параметри агентів Rigidbody

Rigidbody додає об'єкту фізику, дозволяючи йому взаємодіяти із силами, такими як гравітація, тертя чи зіткнення (див. рис. 3.1).

Основні властивості.

Mass (Маса) – визначає масу об'єкта, що впливає на силу, необхідну для його переміщення. Чим більша маса, тим важче об'єкт зрушити.

Drag (Сила опору) – задає опір руху в просторі. Використовується для зменшення швидкості об'єкта.

Angular Drag (Кутовий опір) – контролює опір до обертання.

Use Gravity (Використовувати гравітацію) – якщо увімкнено, на об'єкт діє сила гравітації.

Is Kinematic (Кінематичний) – якщо увімкнено, Rigidbody перестає підкорятися фізичним силам, але його все ще можна переміщувати скриптом [36].

Constraints (Обмеження) – дозволяє заморожувати переміщення або обертання об'єкта по певних осях.

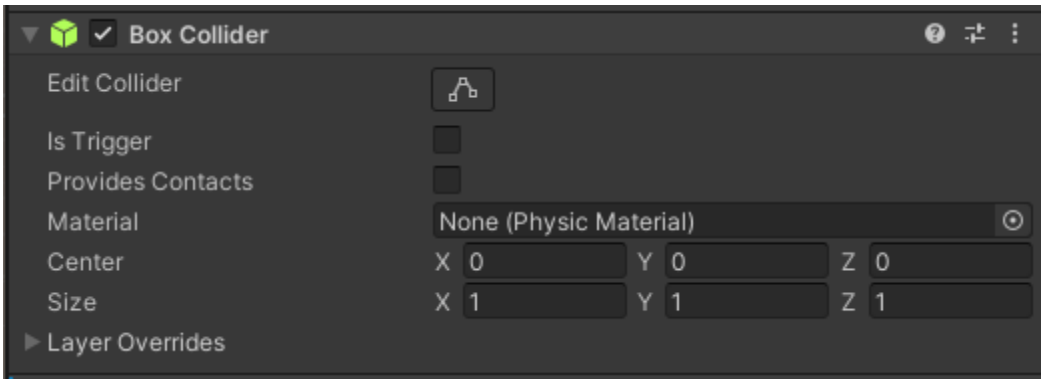


Рисунок 3.2 – Параметри агентів BoxCollider

Box Collider додає об'єкту невидиму "коробку", яка використовується для визначення зіткнень (див. рис. 3.2). Об'єкти з Box Collider можуть взаємодіяти один з одним, якщо у них також є Rigidbody.

Основні властивості.

Center – визначає центр колайдера відносно об'єкта.

Size – розмір колайдера по осях X, Y, Z.

Is Trigger (Тригер) – якщо увімкнено, об'єкт не має фізичної взаємодії, але реагує на події OnTriggerEnter, OnTriggerStay і OnTriggerExit.

Разом вони надають об'єкту фізичні властивості: вагу, рух, обертання. Та забезпечують коректні зіткнення або тригерну взаємодію з іншими об'єктами.

Behaviour Parameters – це ключовий компонент у Unity ML-Agents, що визначає, як агент взаємодіє з навколишнім середовищем і як він навчається. Behaviour Parameters дозволяють налаштувати спосіб навчання агента, тип його дій і обробку спостережень [37]. Вони задаються через інспектор Unity і мають низку важливих параметрів.

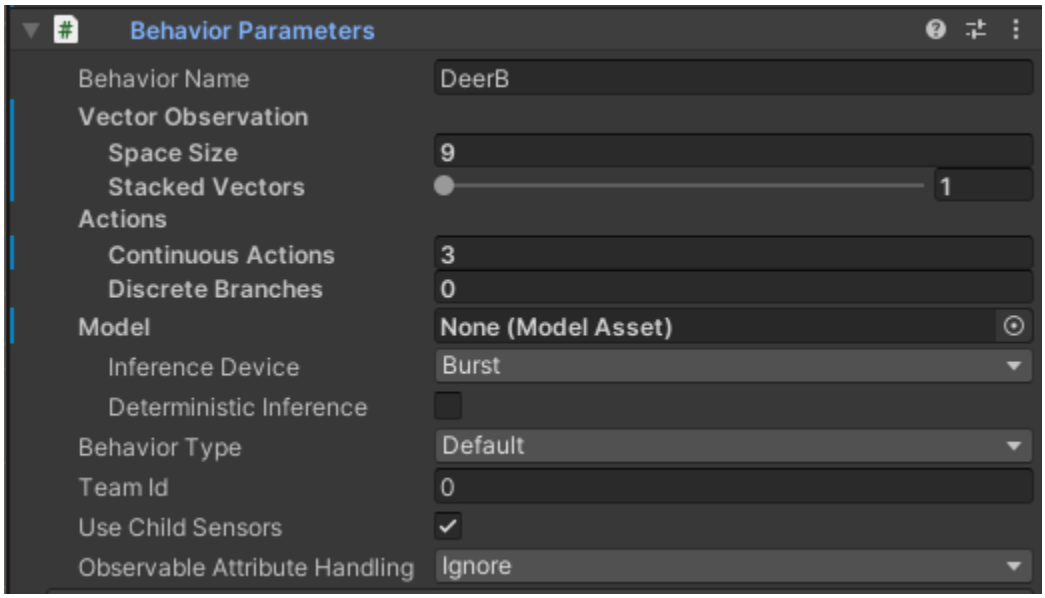


Рисунок 3.3 – Параметри агентів Behaviour

BehaviorName – це унікальне ім’я, яке визначає агент і прив'язує його до конкретної політики навчання (див. рис. 3.3). Наприклад, якщо ви використовуєте кілька агентів, кожен із них може мати своє ім’я для ідентифікації під час навчання [38].

VectorObservation – векторні спостереження визначають кількість числових значень, які агент отримує від середовища. Параметри:

Space Size – розмір векторного простору (кількість спостережень).

Stacked Vectors – кількість повторюваних станів, що передаються до нейронної мережі, що дозволяє агенту враховувати часовий контекст.

Actions (тип і простір дій агента)

Branch Type може бути Discrete (дискретний) або Continuous (неперервний).

Continuous Actions – використовується для керування агентами, що виконують плавні рухи. Наприклад, водіння автомобіля.

Discrete Actions – икористовується для вибору з кількох чітко визначених дій, наприклад, "йти вперед", "повернути наліво". Кількість гілок і розмір кожної гілки налаштовуються для точного визначення простору дій.

Model – поле для завантаження попередньо натренованої моделі (файл .onnx). Це дозволяє використовувати вже навчений агент у середовищі без необхідності подальшого навчання.

InferenceDevice (визначає, на якому пристрої виконуватиметься обчислення)

CPU – стандартний варіант, якщо немає доступу до GPU.

GPU – забезпечує швидші обчислення, особливо для складних моделей.

UseChildSensors – увімкнення цього параметра дозволяє використовувати сенсори, додані до дочірніх об'єктів агента (наприклад, Ray Perception Sensor).

Normalization – ця опція дозволяє нормалізувати спостереження агента, масштабуючи їх до певного діапазону. Це сприяє стабільності навчання.

Goals (Цілі) – використовується для визначення кінцевої мети агента, наприклад, досягти певної точки або зловити інший об'єкт.

Behaviour Parameters забезпечують гнучкість у налаштуванні агента, дозволяючи адаптувати його поведінку до конкретного завдання. Це основний механізм для інтеграції навчання з підкріпленням у Unity ML-Agents.

Model Overrider – це функція в Unity ML-Agents, яка дозволяє задавати індивідуальні моделі для конкретних агентів у середовищі, навіть якщо вони мають однаковий Behavior Name. Ця функція забезпечує гнучкість і дозволяє використовувати різні стратегії або політики для кожного агента без створення додаткових наборів параметрів поведінки [39].

Model Overrider дозволяє агентам з однаковим ім’ям поведінки використовувати різні моделі. Це корисно, якщо потрібно протестувати кілька варіантів політик в одному середовищі. У Unity для кожного агента можна вручну призначити окрему модель через компонент Model Overrider у редакторі. Це робиться шляхом завантаження іншого файлу .onnx у полі Model Overrider. Можливо призначити моделі, які відрізняються структурою чи параметрами, що дозволяє виконувати експерименти з різними типами агентів у

рамках однієї сцени. Використання Model Overrider не конфліктує з тренуванням агента. Якщо модель призначена через Model Overrider, вона використовується для інференції (прогнозування дій) без необхідності змінювати Behavior Parameters [40].

Model Overrider дозволяє легко модифікувати і перевіряти поведінку агентів у складних багатокористувацьких середовищах, що робить процес налагодження і тестування більш зручним і ефективним.

Decision Requester – це компонент у Unity ML-Agents, який відповідає за управління частотою прийняття рішень агентом (див. рис. 3.4). Цей компонент автоматизує запити на рішення від нейронної мережі або іншої системи прийняття рішень, забезпечуючи гнучкість і зручність у налаштуванні поведінки агентів у середовищі.

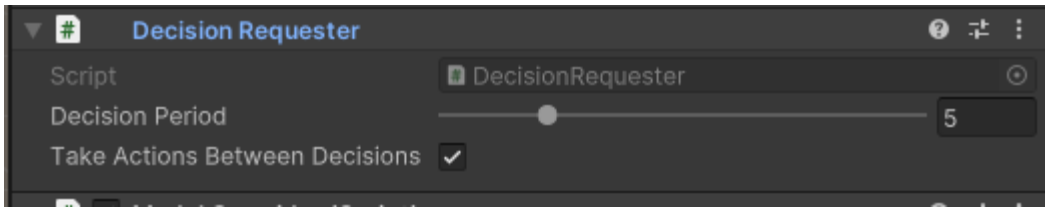


Рисунок 3.4 – Параметри агентів DecisionRequester

Decision Requester дозволяє визначити, як часто агент запитує нове рішення від моделі. Це регулюється параметром Decision Period, значення 1 означає, що агент запитує рішення на кожному кроці симуляції. Більші значення змушують агента використовувати одне і те ж рішення протягом кількох кроків, що може бути корисно для зменшення навантаження на систему або для моделювання поведінки, яка не потребує частих оновлень [41].

Разом із Decision Period можна налаштувати Repeat Action — кількість кроків, протягом яких агент повторює одну і ту ж дію між запитами рішення.

Decision Requester забезпечує інтеграцію з основним середовищем ML-Agents, оптимізуючи процес взаємодії агентів з їхнім оточенням. Наприклад в

іграх або симуляціях, де кожен крок дорогий з точки зору обчислень, частоту рішень можна зменшити, а у швидких середовищах, де потрібно миттєве реагування, можна встановити частий запит рішень.

Ray Perception Sensor 3D — це компонент Unity ML-Agents, який використовується для надання агенту можливості сприймати оточення за допомогою віртуальних "променів" (див. рис. 3.5). Цей сенсор імітує роботу датчиків, схожих на ті, що використовуються в робототехніці (наприклад, LIDAR), і дозволяє агенту збирати інформацію про об'єкти навколо нього.

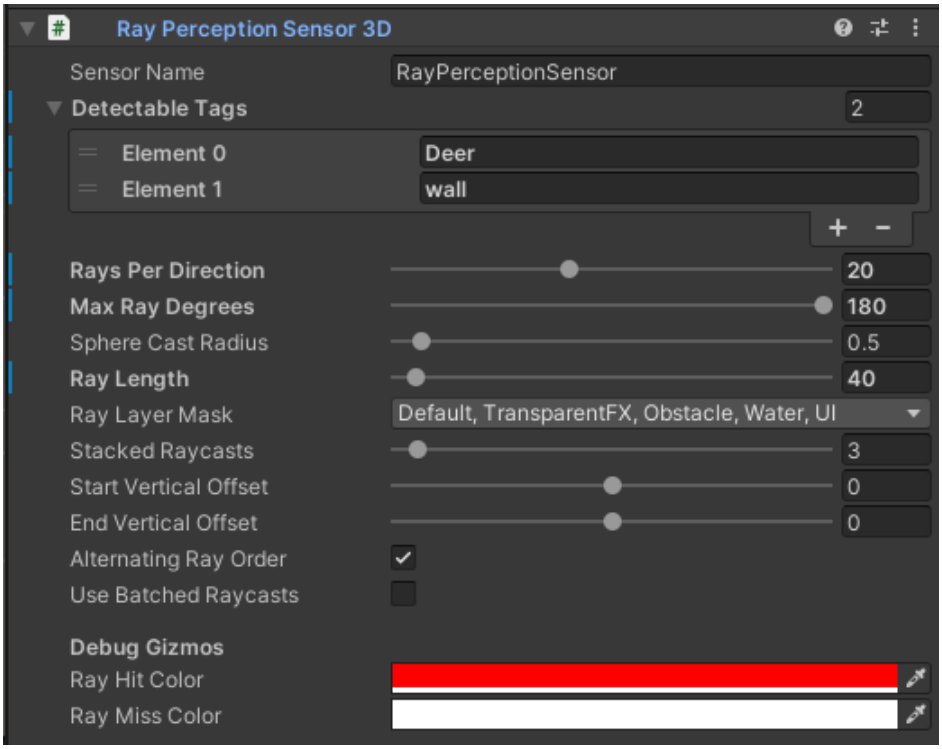


Рисунок 3.5 – Параметри агентів Ray Perception Sensor 3D

Сенсор випускає набір променів у тривимірному просторі відносно агента. Кожен промінь перевіряє наявність перешкод у заданому напрямку, визначаючи тип об'єкта, який промінь "побачив" (через налаштування тегів або шарів у Unity) [42].

Ray Layer Mask – визначає, які шари об'єктів сенсор буде враховувати.

Detectable Tags – список тегів, які агент може "розпізнати". Наприклад, "Wall", "Target", "Obstacle".

Ray Length – максимальна довжина променя (як далеко агент "бачить").

Angles – визначають напрямки, під якими випускаються промені.

Ray Perception Sensor 3D дозволяє охоплювати не лише площину, але й тривимірний простір. Промені можуть випускатися у різних вертикальних і горизонтальних напрямках, щоб агент міг сприймати об'єкти як знизу, так і зверху.

Зібрана сенсором інформація передається до нейронної мережі як частина спостережень агента. Для кожного променя передаються дані: наявність перешкоди, відстань до перешкоди (нормалізована), тег або тип об'єкта, ефективність.

Параметри:

- Sensor Shape: визначає форму сенсора (наприклад, сектор);
- Number of Rays: задає кількість променів;
- Max Ray Degrees: задає кут між крайніми променями;
- Ray Length: задає максимальну довжину променів;
- Detectable Tags: додає теги для розпізнаваних об'єктів;
- Layer Mask: задає шари, на які реагуватимуть промені.

3.2 Структура інтелектуальної системи

Загальна архітектура.

Система побудована на базі Unity ML-Agents та моделей глибокого навчання, інтегрованих через Python API. Основна мета — створення агентів, здатних адаптивно взаємодіяти з віртуальним середовищем, приймаючи ефективні рішення на основі спостережень та винагород.

Основні компоненти.

1. Unity Environment – симуляційне середовище для агентів, що включає фізичні взаємодії, перешкоди та об'єкти (див. рис. 3.9).

2. ML-Agents Toolkit – бібліотека для інтеграції навчання підкріплення з Unity.

3. Python API – забезпечує взаємодію між середовищем Unity і платформами для навчання нейронних мереж, такими як PyTorch або TensorFlow (див. рис. 3.10).

4. Нейронна мережа – модель, що приймає рішення агента, навчаючись на даних зі спостережень.

Ключові компоненти.

1. Агент – це автономний об'єкт у середовищі Unity, який навчається взаємодіяти з оточенням.

Спостереження: дані, які агент отримує через сенсори (позиція, швидкість, відстань до об'єктів).

Дії: рух, обертання чи інші взаємодії з об'єктами середовища. Вони можуть бути:

– дискретними (наприклад, "рух вперед", "повернути ліворуч");

– неперервними (наприклад, "обертання на певний кут");

– нагороди: мотиваційна система, що дає позитивні винагороди за бажану поведінку (наприклад, досягнення мети) та штрафи за небажані дії (зіткнення з перешкодами).

2. Сенсори забезпечують збір даних про середовище. Основний сенсор у системі – Ray Perception Sensor 3D.

Ray Perception створює "промені", що визначають наявність об'єктів у просторі.

Налаштування:

– Detectable Tags – теги, які агент розпізнає (наприклад, "Ціль", "Стіна");

– Ray Length – максимальна відстань, яку охоплює сенсор;

– Layer Mask – шари, з яких сенсор отримує інформацію.

3. Decision Requester – цей компонент визначає частоту прийняття рішень.

Decision Period: скільки кроків симуляції виконується перед кожним рішенням агента.

Repeat Action: кількість повторень однієї й тієї ж дії.

4. Нейронна мережа – модель нейронної мережі має наступну архітектуру:

– приховані шари: обробляють вхідні дані;

– вихідний шар: генерує дії агента;

– параметри навчання;

– Hidden Units: кількість нейронів у кожному прихованому шарі;

– Num Layers: кількість шарів;

– Learning Rate: швидкість навчання.

Процес взаємодії компонентів:

– спостереження: агент отримує інформацію про середовище через сенсори;

– обробка: Python API передає ці дані у модель нейронної мережі;

– рішення: нейронна мережа генерує дії агента на основі спостережень;

– дії: агент взаємодіє з середовищем, виконуючи обчислені дії;

– навчання: на основі отриманих винагород модель оновлює свої ваги через алгоритми підкріплювального навчання.

Конфігурація навчання

Навчання агентів базується на файлі .yaml, який містить гіперпараметри:

– Batch Size – кількість досвідів для одного оновлення;

– Buffer Size – обсяг пам'яті для збереження досвідів;

– Gamma – коефіцієнт знижки для довгострокових винагород.

Сигнали винагороди: Extrinsic Reward – зовнішні винагороди за досягнення мети.

Архітектура тренування агентів.

1. На початку епізоду Агент у середовищі Unity ініціалізує епізод за допомогою методу OnEpisodeBegin(). Він повертається у початковий стан (наприклад, на стартову позицію), і всі змінні стану скидаються. Unity також генерує початкові умови середовища.

2. Агент у Unity викликає метод CollectObservations(), щоб зібрати інформацію про стан середовища (наприклад, координати агента, відстань до цілей, положення перешкод). Ці спостереження формуються як вектор чисел і передаються через протокол gRPC у Python.

3. У Python, отримані спостереження подаються до нейронної мережі (моделі). Ця модель була ініціалізована раніше і навчається на основі алгоритму Proximal Policy Optimization. Модель обчислює відповідні дії агента у вигляді чисел (наприклад, напрямок руху чи швидкість). Результати обчислень надсилаються назад до Unity.

4. Unity отримує дії з Python і застосовує їх до агента через метод OnActionReceived(). На основі цих дій агент змінює свій стан у середовищі (рухається, обертається тощо). Після виконання дій Unity оновлює фізику та візуалізацію середовища.

5. Unity оцінює результати дій агента через систему винагород. Метод AddReward() нараховує позитивну або негативну винагороду залежно від успіху агента (наприклад, наближення до цілі чи уникнення перешкод). Ця винагорода разом із новими спостереженнями передається в Python.

6. На стороні Python алгоритм навчання обчислює, як змінити параметри нейронної мережі для покращення поведінки агента. Використовуючи метод зворотного поширення помилки, модель оновлюється на основі дій агента, винагороди та нового стану середовища.

7. Епізод завершується, якщо агент досягає цілі, порушує умови (наприклад, виходить за межі середовища) або вичерпує час. Unity викликає метод OnEpisodeBegin() для запуску нового епізоду, і процес повторюється.

8. Після кількох епізодів Python зберігає навчений стан моделі у файл .onnx. Ця модель згодом може бути завантажена у Unity для тестування чи використання у грі.

Головні компоненти Unity ML-Agents (див. рис. 3.6).

Перший – це навчальний компонент (на Unity), який містить сцену Unity та елементи оточення [43].

Другий – це Python API, який містить алгоритми RL (такі як PPO та SAC). Ми використовуємо цей API для запуску навчання, тестування тощо. Він взаємодіє з навчальним середовищем через зовнішній комунікатор.

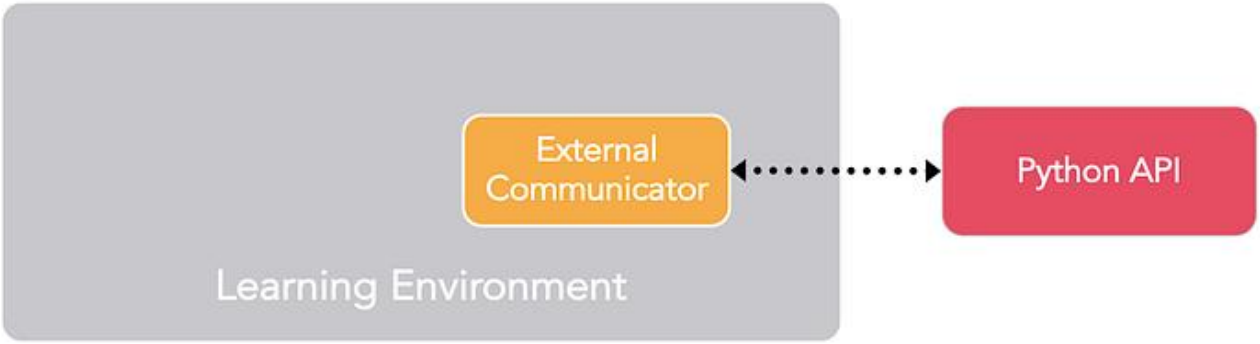


Рисунок 3.6 – Головні компоненти

У навчальному компоненті (див. рис. 3.7). перший — це Агент, актор сцени. Саме його тренеруємо, оптимізуючи його політику (яка визначатиме, яку дію виконати в кожному стані), яку називають Мозок.

І нарешті, є Академія — цей елемент оркеструє агентів та їх процес прийняття рішень. Можна уявити Академію як маестро, який керує запитами з Python API.

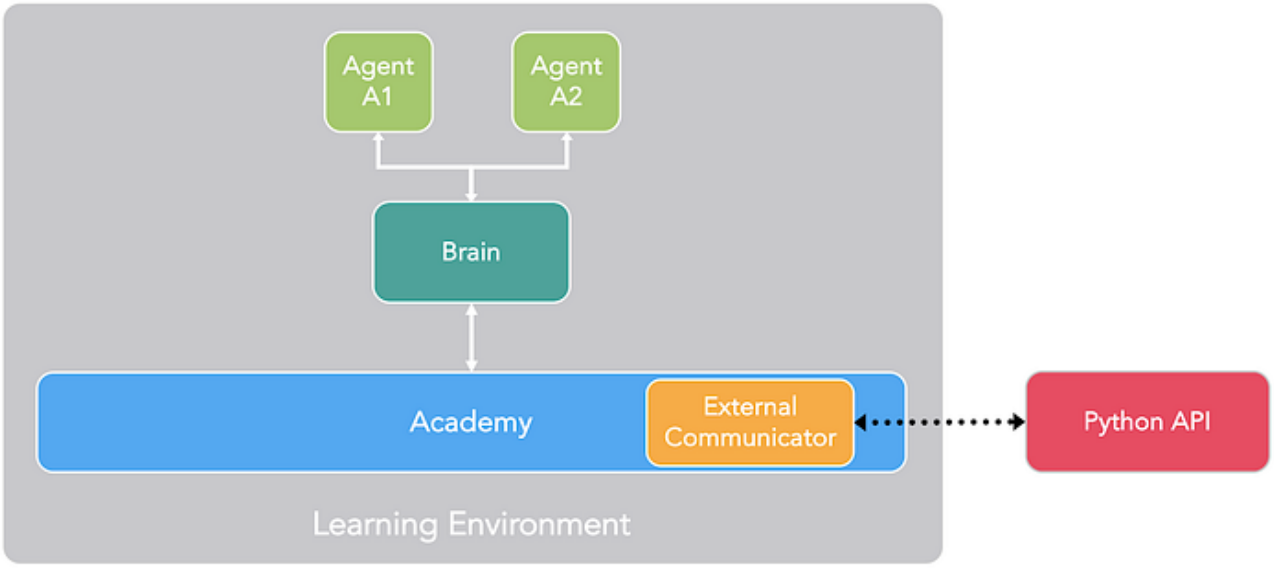


Рисунок 3.7 – Вміст навчального компоненту

Процес RL можно зобразити наступним чином (див. рис. 3.8).

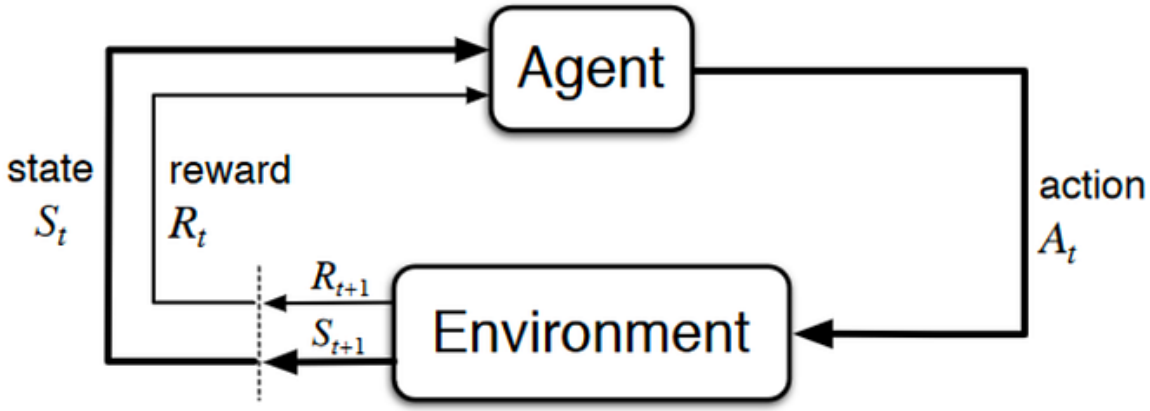


Рисунок 3.8 – Цикл RL

Агент отримує від оточення стан S0 – ми отримуємо перший кадр нашої гри (оточення).

Виходячи зі стану S0, агент виконує дію A0 – агент, наприклад, рухається вправо.

Середовище переходить у новий стан S1.

Видаємо агенту винагороду R1 – він не помер(позитивна винагорода +1).

Цей цикл RL виводить послідовність стану, дії та винагороди. Мета агента – максимізувати очікувану кумулятивну винагороду.

3.3 Опис агентів і конфігурації

WolfAlphaAgent (додаток В)

Цей скрипт представляє альфа-вовка у симуляції вовчої зграї за допомогою ML-агентів Unity [44]. Він очолює зграю і прогнозує майбутнє положення оленя, щоб оптимізувати його рух і координацію.

Ключові особливості:

– система прогнозування: реалізує прогнозування майбутнього положення оленя на основі швидкості та оновлює прогнози через певні проміжки часу для мінімізації обчислень;

– координація зграї: винагорода на основі близькості до цільового оленя та позиціонування відносно інших членів зграї;

– спостереження: збирає дані про місцезнаходження альфа-вовка, прогнозоване місцезнаходження оленя, швидкість і місцезнаходження членів зграї.

Винагороди:

– нагороджує за скорочення відстані до оленя і збереження близькості до передбачуваної позиції;

– штрафує за погані прогнози або віддалення від цілі.

Переваги:

– детальна система прогнозування для покращення полювання;

– нагороди за координацію зграї заохочують реалістичну групову динаміку.

WolfPackMateAgent (додаток Г)

Цей скрипт координує дії окремих членів зграї з альфа-вовком.

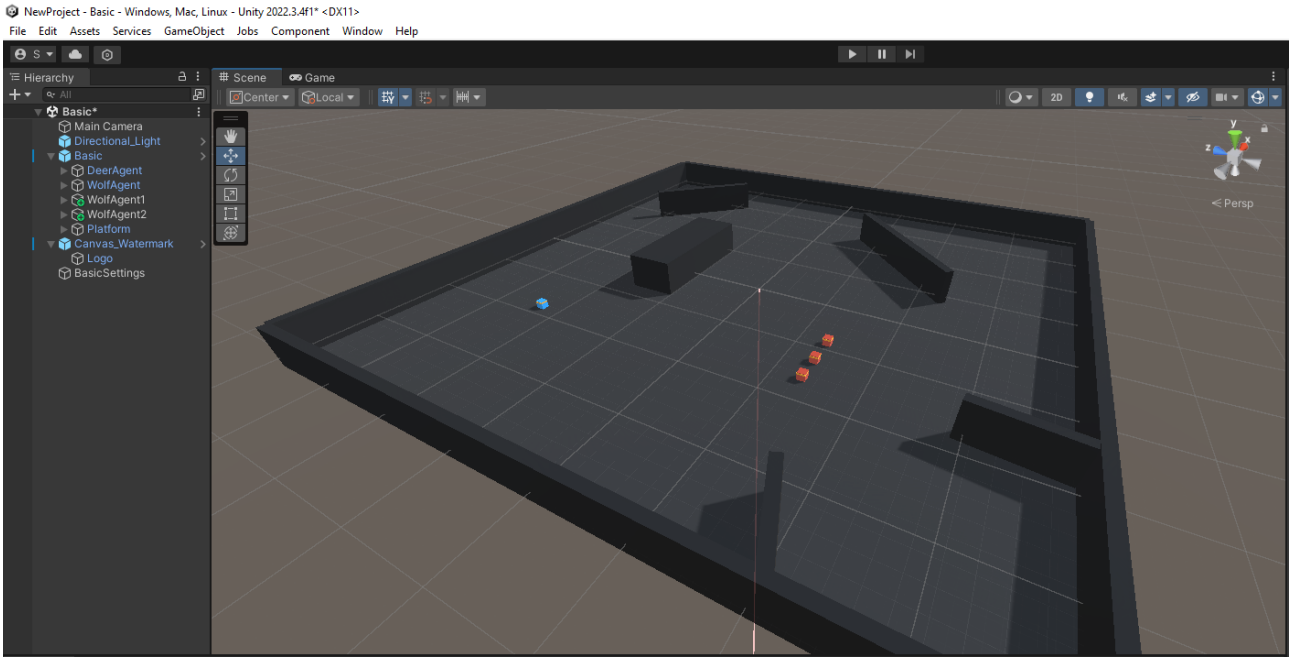


Рисунок 3.9 – Зона тренування агентів

Основні характеристики:

– призначення флангу: кожному члену зграї призначається унікальна позиція на фланзі навколо оленя на основі його індексу;

– формування зграї: винагорода за дотримання оптимальної відстані від альфа-вовка та інших членів зграї;

– спостереження: збирає дані про положення, швидкості та відстані, пов'язані з оленем, альфа-вовком та іншими членами зграї.

Сильні сторони:

– заохочує кооперативну поведінку через винагороду за флангове обходження та формування;

– окремі функції винагороди за рух, координацію та підтримання формації.

DeerAgent (додаток Б)

Цей скрипт моделює оленя, здобич у симуляції, і його основна мета – уникати вовків.

Основні характеристики:

– механізм втечі: використовує raycast для виявлення вовків і динамічної втечі;

– штраф за бездіяльність: штрафує за бездіяльність, щоб заохотити ефективний рух;

– нагороди за рух: винагорода за цілеспрямований рух та уникнення вовків.

Сильні сторони:

– чутливий механізм втечі збільшує виклик для вовків;

– логічна система покарань перешкоджає бездіяльності або неефективній поведінці.

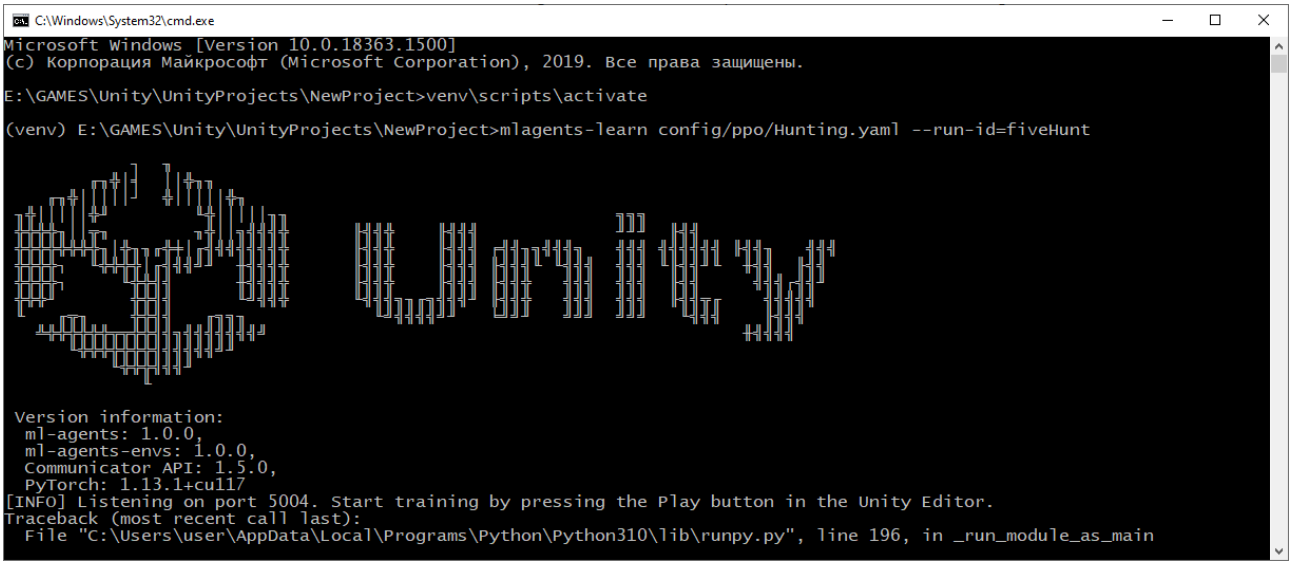


Рисунок 3.10 – Віртуальний простір

Файл .yaml (Додаток Д) для Unity ML-Agents визначає конфігурацію поведінки агентів [45]. У ньому налаштовуються параметри тренувань, гіперпараметри, мережеві архітектури та сигнали винагороди для кожного агента:

– behaviors: це головний розділ, який містить список агентів із їхніми параметрами;

– trainer_type: тип тренувальника. У даному випадку використовується ppo (Proximal Policy Optimization), який є популярним методом для навчання агентів;

– hyperparameters: налаштування для алгоритму навчання;

– batch_size: кількість досвідів, які використовуються для одного оновлення політики;

– buffer_size: розмір буфера, в якому зберігаються досвіди перед навчанням;

– learning_rate: швидкість навчання моделі;

– beta: параметр регуляризації ентропії, що допомагає агенту досліджувати більше;

– epsilon: коефіцієнт для обмеження змін у політиці;

– lambd: параметр для GAE (Generalized Advantage Estimation);

– num_epoch: кількість проходів через буфер під час навчання;

– learning_rate_schedule: графік зміни швидкості навчання (наприклад, linear);

– network_settings: архітектура нейронної мережі;

– normalize: нормалізація вхідних даних;

– hidden_units: кількість нейронів у кожному прихованому шарі;

– num_layers: кількість прихованих шарів;

– vis_encode_type: тип візуального енкодера, якщо агент працює з візуальними спостереженнями (наприклад, simple);

– reward_signals: налаштування сигналів винагороди;

– extrinsic: зовнішня винагорода;

– gamma: коефіцієнт знижки для довгострокових винагород;

– strength: вага сигналу винагороди;

– keep_checkpoints: кількість збережених контрольних точок моделі;

– max_steps: максимальна кількість кроків, яку агент може виконати під час тренувань;

– time_horizon: кількість кроків, за які обчислюється знижка винагороди;

– summary_freq: як часто зберігаються резюме тренувань для TensorBoard.

3.4 Програмна реалізація

EnvironmentalAwareness (додаток А)

Клас допомагає агенту виявляти перешкоди в заданому радіусі за допомогою променевих повідомлень, збирати детальну інформацію про навколишнє середовище, включаючи розташування та відстань до перешкод та розраховувати безпечний напрямок руху на основі бажаного напрямку руху та виявлених перешкод.

Ключові компоненти.

Структура даних про навколишнє середовище інкапсулює інформацію про оточення агента:

– obstacleDetected – чи виявлено перешкоду в заданому напрямку;

– obstacleDistances – відстань до виявлених перешкод;

– obstacleDirections – напрямки виявлених перешкод відносно агента;

– safeDirections – напрямки, де не виявлено жодних перешкод.

Конструктор ініціалізує клас за допомогою:

– agentTransform – положення та орієнтація агента у світі;

– detectionRadius – на якій відстані агент може виявити перешкоди;

– rayCount – кількість променів для сканування середовища;

– obstacleLayer – вказує, які шари вважати перешкодами.

Основні методи.

ScanEnvironment() виконує променеве сканування по колу навколо агента для виявлення перешкод. Ділить 360 градусів на рівні кути (rayCount) для кожного кута випромінює промінь у відповідному напрямку, якщо промінь потрапляє на перешкоду то позначає цей напрямок як заблокований, записує відстань до перешкоди і позначає напрямок як небезпечний. Якщо промінь не потрапляє на перешкоду то позначає напрямок як безпечний.

Вихідні дані повертають EnvironmentData, що містить інформацію про виявлені перешкоди та безпечні напрямки.

GetBestAvoidanceDirection(Vector3 desiredDirection, EnvironmentData envData) обчислює найкращий можливий напрямок руху, який дозволяє уникнути перешкод. Знаходить найближчий напрямок до desiredDirection за допомогою GetClosestDirectionIndex(), якщо найближчий напрямок безпечний (не виявлено перешкод), повертає desiredDirection. Якщо заблоковано, обчислює всі безпечні напрямки на основі вирівнювання з бажаним напрямком (за допомогою крапкового добутку) відстань до перешкод (перевага надається дальнім напрямкам) і повертає найкращий безпечний напрямок на основі цієї оцінки.

GetClosestDirectionIndex(Vector3 direction, Vector3[] directions) знаходить індекс напрямку, найбільш близького до заданого вектора (напрямку) і обчислює точковий добуток між нормалізованим напрямком і кожним вектором у напрямках та вибирає вектор з найбільшим точковим добутком (найближче вирівнювання).

DeerAgent (додаток Б)

Властивості агента.

speed, rotationSpeed – керує швидкістю руху та швидкістю повороту.

raycastDistance – визначає, наскільки далеко агент може «бачити» перед собою для виявлення вовка.

detectionRadius, rayCount – визначають обізнаність про навколишнє середовище для виявлення перешкод за допомогою спеціального класу EnvironmentalAwareness.

Керування станом.

isFleeing – відстежує, чи тікає агент наразі від хижака.

lastSeenWolfTime – відстежує, коли востаннє було виявлено вовка, щоб керувати перезавантаженнями для втечі.

idleTime – відстежує, як довго агент залишається нерухомим, щоб покарати за бездіяльність.

Основні методи.

Initialize() – готує агента, встановлюючи rigidbody, початкову позицію та обізнаність про навколишнє середовище.

OnEpisodeBegin() – скидає стан та положення агента на початку нового епізоду навчання.

CollectObservations(датчик VectorSensor) надає агенту інформацію про його оточення, таку як: положення, швидкість та напрямок руху. Наявність перешкод та відстань до них, визначені за допомогою системи EnvironmentalAwareness.

OnActionReceived(ActionBuffers actions) обробляє вибрані агентом дії. Рух (об'єднує вхідні дані про рух вперед, обстріл та обертання), уникнення перешкод (використовує дані про оточення для пошуку безпечних напрямків руху), нагороди (заохочує рух і карає за зіткнення, бездіяльність або небезпечну поведінку.

Винагороди заохочують рух, якщо він ефективно уникає перешкод або долає значні відстані.

Штрафи застосовуються за бездіяльність, зіткнення з вовками або небезпечну поведінку.

RaycastForWolves() – виявляє вовків у напрямку руху агента за допомогою променевих сигналів.

FleeFromWolf() – розраховує та виконує рух втечі від вовків при їх виявленні.

OnCollisionEnter(Зіткнення) – завершує епізод зі штрафом, якщо агент зіткнеться з вовком.

Heuristic() – дозволяє вручну керувати агентом під час тестування, зіставляючи вхідні дані користувача з діями агента.

WolfAlphaAgent (додаток В)

Властивості агента.

speed and rotationSpeed – визначає швидкість руху та обертання вовка.

rb – компонент rigidbody для застосування фізики руху.

Awareness – екземпляр EnvironmentalAwareness для виявлення та уникнення перешкод.

targetDeer – олень, якого переслідує вовк.

wolfPack – інші вовки у зграї.

predictionTime і predictionSteps – визначає, наскільки далеко вперед і на скільки кроків передбачити рух оленя.

predictedDeerPosition – рогнозоване майбутнє положення оленя.

Основні методи.

Initialize() – встановлює початковий стан агента, включаючи його rigidbody, систему обізнаності та початкові прогнози щодо положення оленя.

OnEpisodeBegin() – скидає позицію, швидкість та змінні передбачення агента на початку навчального епізоду.

UpdateDeerPrediction() – відстежує швидкість оленя та періодично оновлює прогнози.

PredictDeerPosition() – моделює майбутні позиції оленя за кілька кроків на основі його поточної швидкості (див. рис. 3.11). Оцінює кожне передбачення за допомогою EvaluatePredictionPoint(), щоб знайти найкращу позицію.

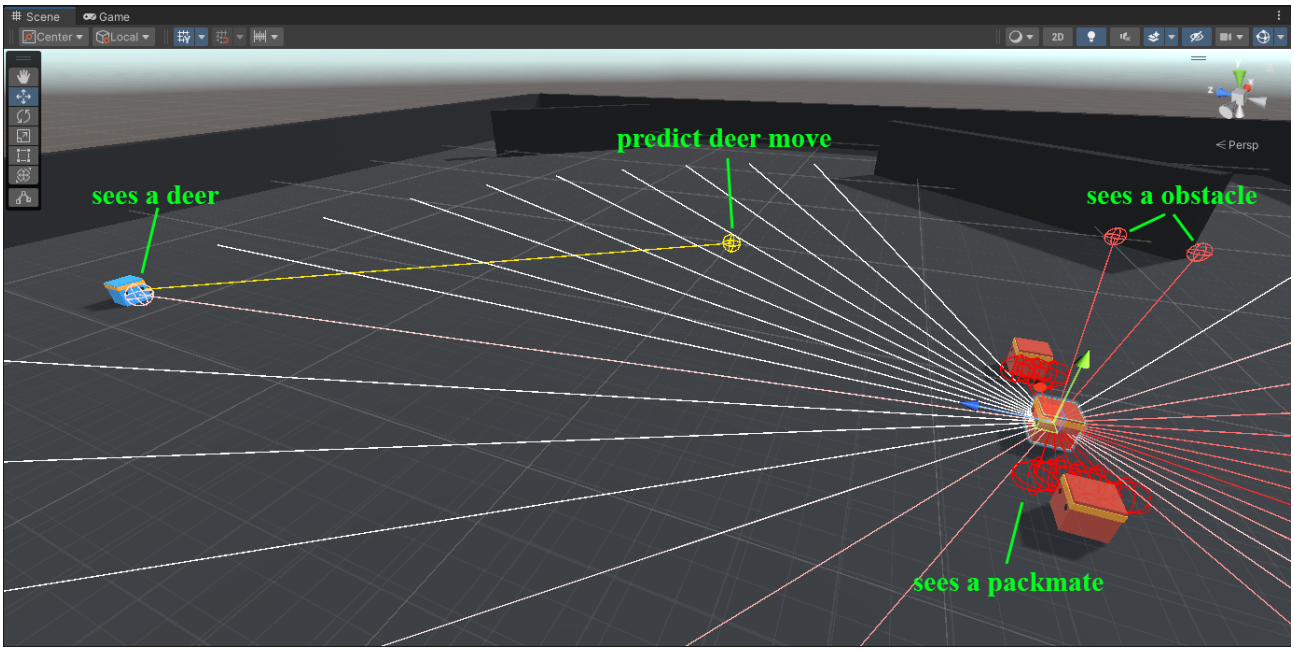


Рисунок 3.11 – AlphaWolf RayPerception

EvaluatePredictionPoint() – оцінює прогнозовану позицію на основі відстані від поточного положення оленя, близькість до перешкод і товаришів по зграї, перевіряє чи знаходиться позиція в допустимих межах.

CollectObservations() – записує дані про навколишнє середовище, в тому числі положення вовка та оленя, прогнозоване положення та швидкість оленя, позиції членів зграї та дані про перешкоди від системи EnvironmentalAwareness.

OnActionReceived() – застосовує рішення агента для руху до прогнозованого місця розташування оленя. Коригує швидкість і напрямок руху на основі даних про перешкоди з системи EnvironmentalAwareness.

Винагорода надається за зменшення відстані до оленя (RewardForProximityToDeer()), точне передбачення оленя

(RewardForPrediction()), лідерство в зграї в безпосередній близькості до оленя (RewardForLeadingPack()).

Heuristic() дозволяє ручне керування агентом з використанням клавіатури для налагодження.

OnDrawGizmosSelected() – малює прогнозовану позицію оленя у редакторі Unity з метою налагодження.

WolfPackMateAgent (додаток Г)

Властивості агента.

speed and rotationSpeed – керує швидкістю руху та швидкістю повороту.

rb – компонент rigidbody для застосування фізики руху.

Awareness – екземпляр EnvironmentalAwareness для виявлення та уникнення перешкод.

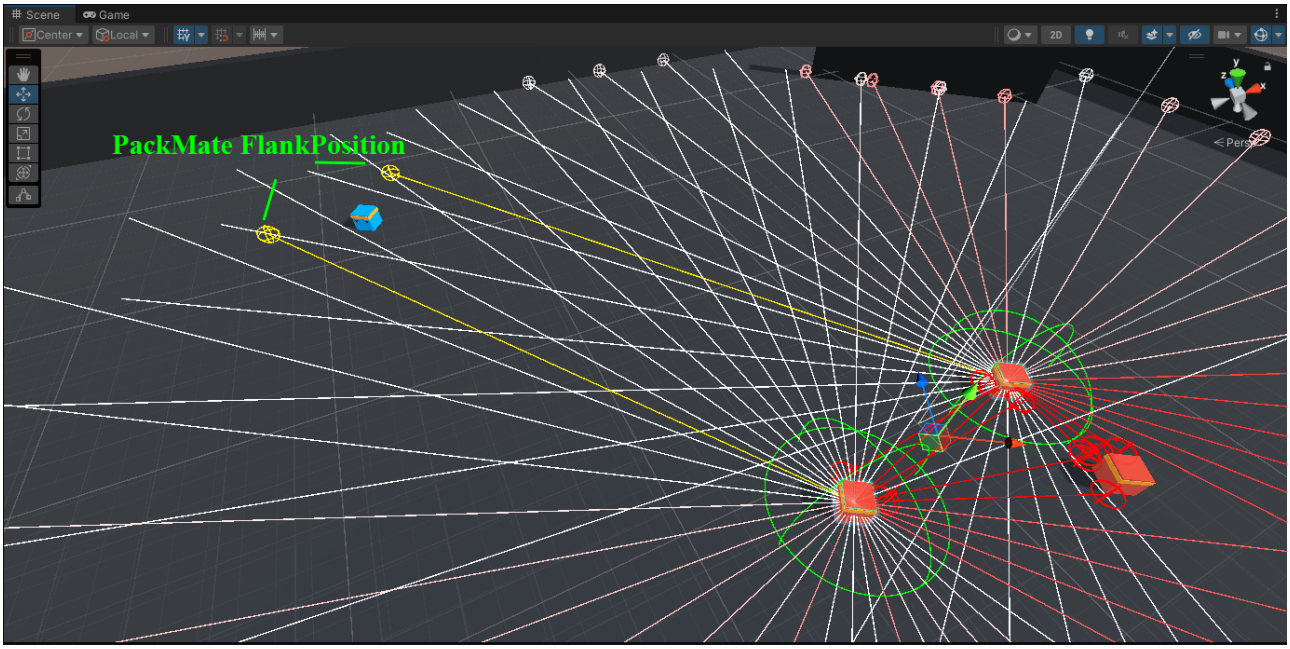


Рисунок 3.12 – PackMateWolf RayPerception

Environment References:

targetDeer – об'єкт полювання.

alphaWolf – альфа–вовк, який очолює зграю.

packMateIndex – унікальний індекс, присвоєний кожному члену зграї для розрізнення ролей.

Параметри surroundDistance, optimalPackDistance, maxPackDistance та minPackDistance визначають ідеальну форму зграї та відстань між її членами.

Основні методи.

Initialize() – встановлює rigidbody агента та його обізнаність щодо перешкод. Викликає AssignFlankPosition(), щоб визначити його роль в оточенні оленя.

OnEpisodeBegin() – скидає позицію, швидкість та флангові призначення агента на початку нового навчального епізоду.

CollectObservations() – збирає дані про:

- положення, швидкість та напрямок руху вовка;
- позиції оленя, альфа–вовка та інших членів зграї;
- присвоєне флангове положення та відстань до перешкод;
- інформація про перешкоди нормалізована для кращої ефективності тренування.

AssignFlankPosition() (див. рис. 3.12) – обчислює унікальну позицію навколо оленя на основі індексу вовка, коригує позицію, щоб уникнути перешкод, використовуючи AdjustForObstacle().

GetFlankAngleOffset() – визначає кутовий зсув для флангового обходу на основі індексу члена зграї, почергово вліво та вправо.

OnActionReceived() – обробляє безперервні дії для руху, стрейфу та обертання, інтегрує уникнення перешкод з рухом за допомогою класу EnvironmentalAwareness.

CalculateMovementReward() – заохочує скорочення відстані до оленя, зберігаючи при цьому оптимальну дистанцію до зграї.

CalculateFlankingReward() – винагороджує за наближення до призначеної флангової позиції та вирівнювання з оленем.

CalculatePackCoordinationReward() – заохочує дотримання належної відстані від альфа–вовка та інших членів зграї.

CalculateFormationReward() – винагороджує за підтримання скоординованої формації зі зграєю.

Heuristic() дозволяє ручне керування агентом з використанням клавіатури для налагодження.

OnDrawGizmosSelected() – візуалізує призначену флангову позицію та оптимальну відстань між пакетами у редакторі Unity для налагодження.

Висновки до розділу 3

Проект продемонстрував доцільність і ефективність використання навчання з підкріпленням для моделювання складної поведінки в дикій природі, наприклад, стратегії полювання у вовчих зграях. Розділивши ролі Alpha і Packmates і використовуючи набір інструментів ML-Agents для навчання вовків на основі винагород, система змогла навчитися спільної поведінки, яка імітувала тактику полювання в реальному світі.

Попри те, що процес навчання зіткнувся з технічними перешкодами, які були успішно виправлені, такими як труднощі в підтримці послідовного зв'язку між середовищами Unity і Python, проблеми сумісності версій і випадкове пошкодження файлів проєкту, окрім того, складний дизайн структур винагород вимагав ретельного калібрування, щоб запобігти конфлікту цілей і непередбачуваній поведінці, проєкт успішно впровадив систему, в якій вовки навчилися співпрацювати, а олені навчилися ухилятися демонструючи потенціал машинного навчання для моделювання поведінки в штучних середовищах.

4 ДОСЛІДЖЕННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1 Аналіз перших отриманих результатів

Кумулятивна винагорода

Результати по графіках третього етапу ("thirdHunt") свідчать про значне покращення кумулятивної винагороди у "WolfA" і "WolfP". Особливо видно стабільний тренд покращення винагороди у порівнянні з "firstHunt" і "secondHunt". "WolfB" також демонструє позитивну динаміку, хоча прогрес не такий різкий.

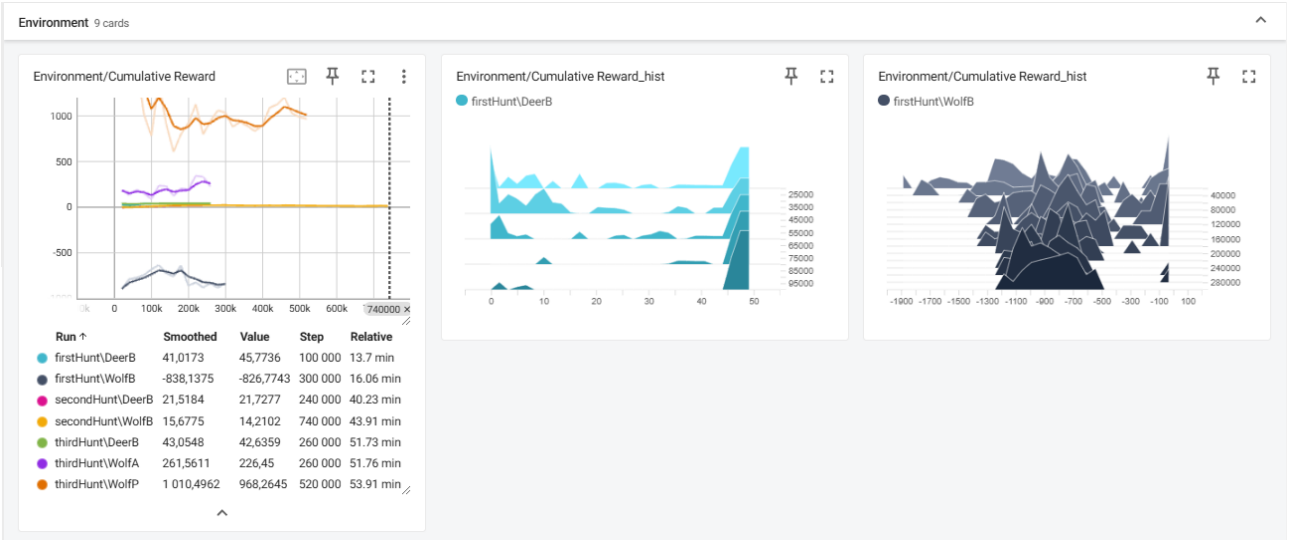


Рисунок 4.1 – Спостереження за ходом навчання 1

Довжина епізодів

Довжина епізодів ("Episode Length") значно збільшується у "thirdHunt" порівняно з "firstHunt" і "secondHunt". Це вказувати на те, що вовки стали ефективнішими у переслідуванні оленя, що дозволяє їм довше залишатися у грі.

Аналіз помилок у першому і другому етапах

У "firstHunt" і "secondHunt" спостерігається високий рівень штрафів (від'ємна винагорода) для "WolfB". Це може свідчити про те, що вовк

здійснював зайві або хаотичні повороти (відсутність згладжування), або ж стратегія переслідування не працювала ефективно.

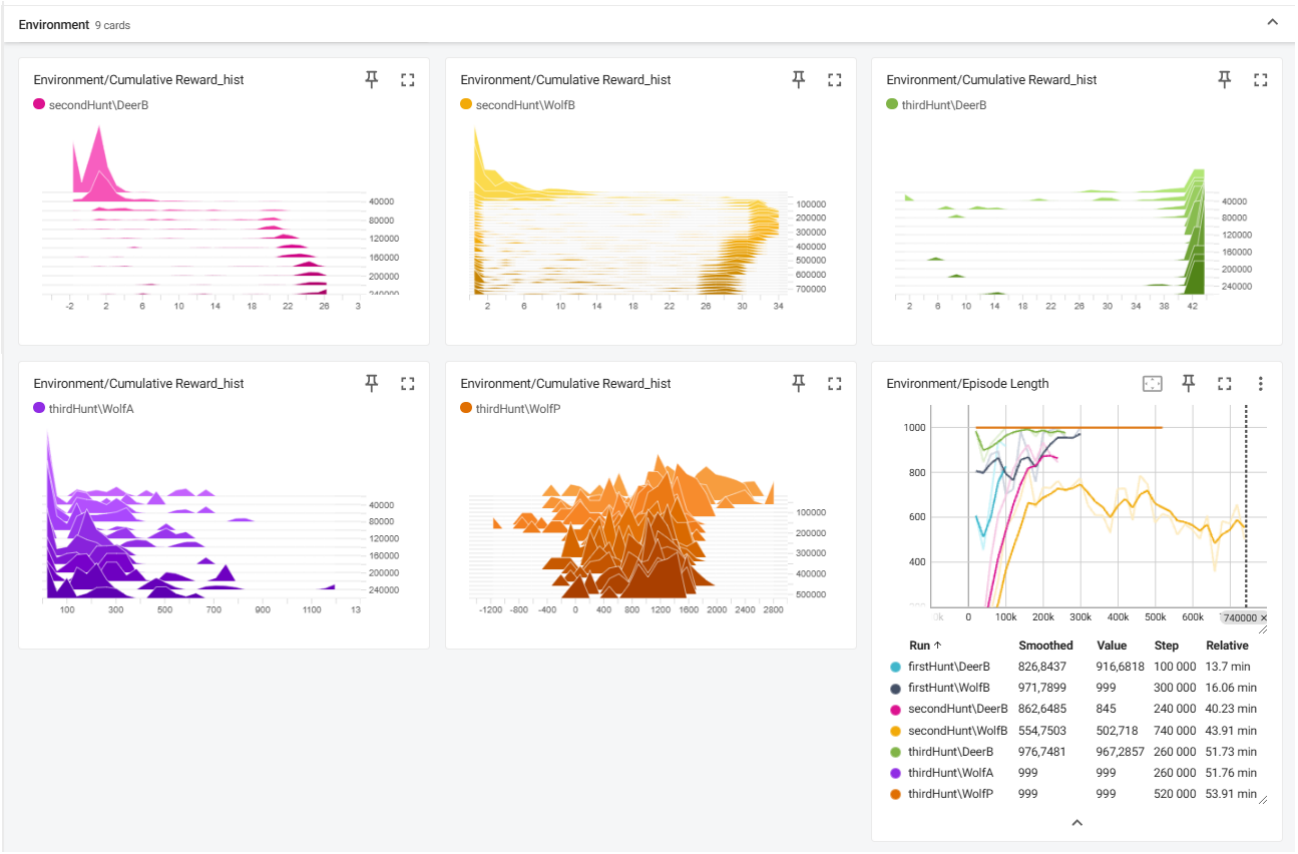


Рисунок 4.2 – Спостереження за ходом навчання 2

Покращення після оновлень

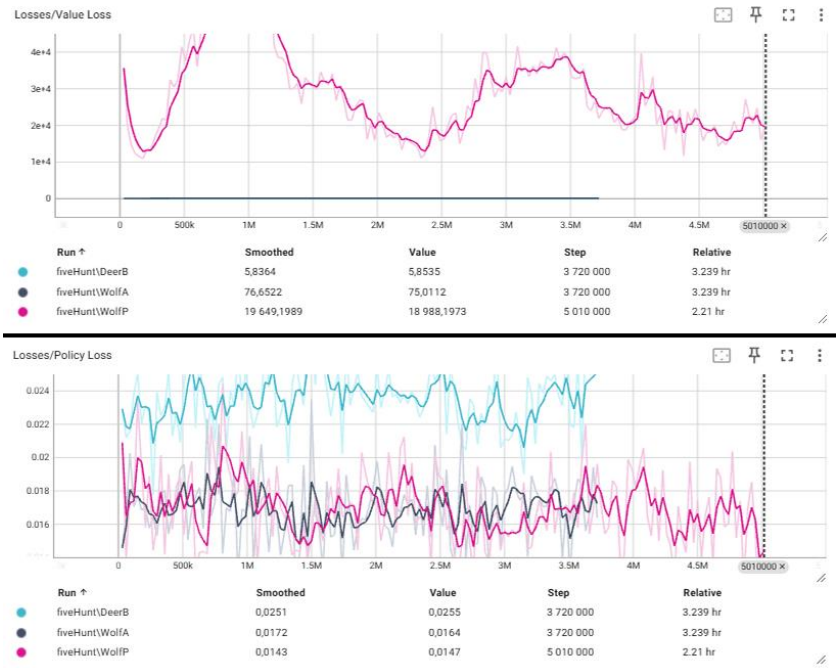
Прогнозування позицій оленя ("Predicting Deer Position") разом зі згладжуванням поворотів ("Smooth Rotations") поліпшило поведінку агентів, зокрема для "WolfA" і "WolfP". Плавний рух і зменшення хаотичних поворотів значно зменшили штрафи.

Run	Smoothed	Value	Step	Relative
firstHunt\DeerB	0,0968	0,0944	100 000	10.23 min
firstHunt\WolfB	0,1006	0,0993	100 000	3.687 min
secondHunt\DeerB	0,0996	0,0988	240 000	40.23 min
secondHunt\WolfB	0,096	0,0929	260 000	15.74 min
thirdHunt\DeerB	0,0965	0,096	260 000	51.73 min
thirdHunt\WolfA	0,0995	0,1005	260 000	51.76 min
thirdHunt\WolfP	0,0984	0,0982	520 000	53.91 min

Run	Smoothed	Value	Step	Relative
firstHunt\DeerB	0,042	0,0262	100 000	10.23 min
firstHunt\WolfB	31,5517	30,3798	100 000	3.687 min
secondHunt\DeerB	0,0196	0,0224	240 000	40.23 min
secondHunt\WolfB	0,0313	0,0348	260 000	15.74 min
thirdHunt\DeerB	0,0142	0,0159	260 000	51.73 min
thirdHunt\WolfA	118,16	121,9139	260 000	51.76 min
thirdHunt\WolfP	223,6248	234,3816	520 000	53.91 min

Рисунок 4.3 – Спостереження за ходом навчання 3

4.2 Фінальна версія моделі



Run	Smoothed	Value	Step	Relative
fiveHunt\DeerB	5,8364	5,8535	3 720 000	3.239 hr
fiveHunt\WolfA	76,6522	75,0112	3 720 000	3.239 hr
fiveHunt\WolfP	19 649,1989	18 988,1973	5 010 000	2.21 hr

Run	Smoothed	Value	Step	Relative
fiveHunt\DeerB	0,0251	0,0255	3 720 000	3.239 hr
fiveHunt\WolfA	0,0172	0,0164	3 720 000	3.239 hr
fiveHunt\WolfP	0,0143	0,0147	5 010 000	2.21 hr

Рисунок 4.4 – Спостереження за ходом навчання FiveHunt

Втрати вартості.

Відстежує втрати, пов'язані з прогнозами функції цінності. Вищі втрати цінності вказують на те, що модель намагається точно передбачити винагороду.

Для прогону «WolfP» втрати цінності дещо вищі, що свідчить про недостатню стабільність та невідповідність між складністю середовища та можливостями моделі.

Втрати політики.

Відображає оновлення політики (мережі акторів). Нижчі значення зазвичай вказують на більш стабільне навчання.

«DeerB» має дещо вищий і більш мінливий показник втрати політики порівняно з іншими, що свідчить про те, що її оновлення політики є менш стабільними.

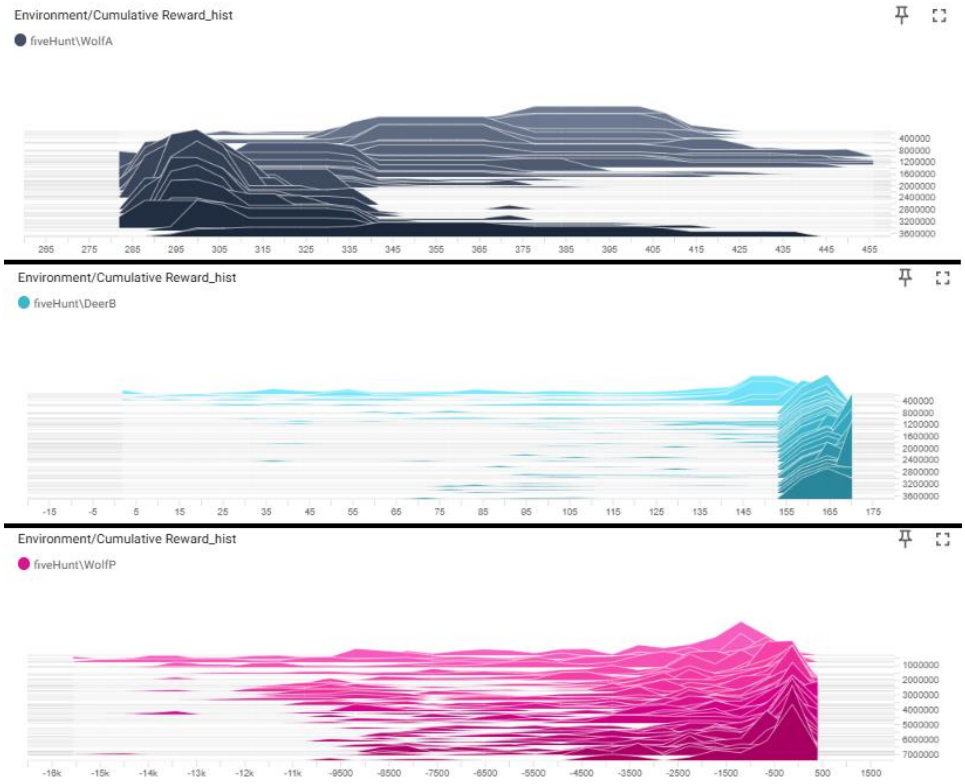


Рисунок 4.5 – Спостереження за ходом навчання FiveHunt

Навколишнє середовище/сукупна винагорода.

Відображає загальну винагороду, накопичену кожним типом агента («WolfA», «Deer», «WolfP») за епізоди.

Спостереження:

«WolfP» накопичує найбільшу винагороду з різким зростанням, але це зростання виглядає непослідовним або періодичним падінням.

«DeerB» демонструє стабільне накопичення нагород, що свідчить про більш стабільну роботу.

«WolfA» також досягає високої винагороди, але його гістограма демонструє більшу мінливість.

4.3 Складності реалізації

Unity ML-Agents – це платформа, що постійно розвивається, з деякими експериментальними функціями. Жодна з цих експериментальних функцій не задокументована належним чином і не оптимізована під час тестування. Наприклад, нові схеми винагороди або зміни в конфігураціях поведінки іноді призводять до неочікуваних результатів під час навчання, що вимагає подальшого налагодження та доопрацювання.

Однією з головних проблем було виявлення та вирішення проблем, пов'язаних із встановленням постійного зв'язку між Unity та навчанням у середовищі Python, про що також зазначалося в темі спільноти Unity (https://discussions.unity.com/t/mlagents-nothing-is-working-python/904964/3).
Проблема була чітко визначена з точки зору поведінки – жодна з команд не могла бути виконана через інтерфейс Python для продовження процесу навчання.

Причиною була невідповідність версій пакету, встановленого в Unity, Unity ML-Agents, та наявного в бібліотеці Python, Python ML-Agents, що означало несумісність протоколів. Деякі з повідомлень про помилки в цей

проміжок часу виглядали наступним чином: «Середовище не відповіло» та “Крок агента не вдалося виконати”.

Незважаючи на це, проблема, що лежить в основі, пов’язана з дуже високою складністю інтеграції Unity з інструментами Python під час розробки програми. Проблеми, пов’язані з розробкою ефективних практик керування версіями та висвітленням проблем, з якими стикаються розробники під час роботи зі швидкозмінними інструментами, також поставлені на карту в інциденті.

Unity значною мірою спирається на Visual Studio для написання скриптів і налагодження. Однак помилки в інтеграції Visual Studio можуть порушити розробку. В моєму випадку Unity міг ігнорувати існуючий Visual Studio, писав помилку, що його не існує і кожен раз треба було по новій вказувати путь до VS.

Трапився випадкок, коли файли проекту Unity несподівано пошкоджились без будь-якої причини і прийшлось створити новий проект та налатовувати по новій, переносити всі скрипти і т.д..

Розробка винагород досить складна, оскільки агентам доводилось аналізувати багато факторів. Кількість даних для обробки, відповідно, збільшила кількість різноманітних винагород, які агентам потрібні, і відповідно ризик перешкод між критеріями, що призводить до непередбачуваної поведінки. Високі винагороди повинні бути добре узгоджені з цілями, оскільки вони будуть в центрі уваги агентів, які можуть «грати» з системою.

Це також робить навчання повільним, створює багато проблем з налагодженням і ризики надмірного пристосування зменшуючи адаптивність. Але такі стратегії, як ієрархія винагороди, розріджені сигнали, ітераційне тестування та динамічне масштабування, є важливими для ефективного керування цими викликами, зберігаючи ефективне навчання.

Висновки до розділу 4

Переглянувши графіки тренувань та проаналізувавши результати, я помітив деякі чіткі тенденції та напрямки для покращення поточного стану справ.

Втрата цінності для певних агентів, зокрема «WolfP», є не високою, що свідчить про те, що функція цінності намагається точно передбачити винагороду. Низька втрата цінності може бути наслідком складної структури винагороди або недостатнього часу навчання для цього агента. З іншого боку, втрати політики значно коливаються для «DeerB», що свідчить про те, що його оновлення політики менш стабільні, можливо, через неоптимальні гіперпараметри, такі як швидкість навчання або розмір партії.

Якщо проаналізувати кумулятивну винагороду, то «WolfP» виділяється найвищою винагородою, але її прогрес має періодичні спади. Це може вказувати на проблеми із співвідношенням розвідки та видобутку або на труднощі у вивченні стабільних політик. «DeerB» демонструє постійне накопичення винагороди, що свідчить про стабільну, якщо не виняткову, ефективність. «WolfA» також демонструє високу винагороду, але з більшою мінливостью, що вказує на потенційну надмірну адаптацію або чутливість до певних станів навколишнього середовища.

Загалом, агенти досягають прогресу, але є можливість вдосконалити налаштування навчання для отримання більш надійних і послідовних результатів навчання.

ВИСНОВКИ

Підводячи підсумки, ця робота забезпечила комплексне дослідження розробки та навчання агентів з використанням Unity ML-Agents і моделей глибокого навчання, зосереджуючись на моделюванні інтелектуальної, кооперативної та змагальної поведінки у віртуальному середовищі. Проект ефективно продемонстрував застосування методів навчання з підкріпленням для відтворення складних форм поведінки, таких як скоординоване полювання у вовчих зграях та маневри ухилення оленів. Інтегруючи вдосконалене моделювання агентів, параметри поведінки та системи винагороди, це дослідження виявило величезний потенціал машинного навчання для створення адаптивних, реалістичних симуляцій.

Мета дослідження є досягнутою, проект успішно продемонстрував можливість навчання агентів, які співпрацюють і конкурують, у змодельованому середовищі, що дало багатообіцяючі результати. Однак як підвищення стабільності моделі, точніше налаштування гіперпараметрів і оптимізація систем винагороди, матиме вирішальне значення для підвищення надійності і масштабованості таких систем. Результати цієї роботи дають цінне уявлення про застосування навчання з підкріпленням і прокладають шлях для подальшого вивчення та вдосконалення агентно-орієнтованих симуляційних досліджень.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Sutton R.S., Barto A.G. Reinforcement Learning: An Introduction. MIT Press, 2018. URL: https://mitpress.mit.edu/books/reinforcement-learning-introduction (дата звернення: 10.06.2024).

2. Goodfellow I., Bengio Y., Courville A. Deep Learning. MIT Press, 2016. URL: https://www.deeplearningbook.org/ (дата звернення: 13.06.2024).

3. Chollet F. Deep Learning with Python. Manning Publications, 2018. URL: https://www.manning.com/books/deep-learning-with-python (дата звернення: 13.06.2024).

4. Szegedy C. et al. Going Deeper with Convolutions. IEEE Conference on Computer Vision and Pattern Recognition, 2015. URL: https://doi.org/10.1109/CVPR.2015.7298594 (дата звернення: 15.06.2024).

5. Schmidhuber J. Deep Learning in Neural Networks: An Overview. Neural Networks, 2015. URL: https://doi.org/10.1016/j.neunet.2014.09.003 (дата звернення: 18.06.2024).

6. He K., Zhang X., Ren S., Sun J. Deep Residual Learning for Image Recognition. IEEE Conference on Computer Vision and Pattern Recognition, 2016. URL: https://doi.org/10.1109/CVPR.2016.90 (дата звернення: 20.06.2024).

7. Unity Technologies. Unity ML-Agents Toolkit Documentation. Unity Technologies, 2024. URL: https://github.com/Unity-Technologies/ML-Agents (дата звернення: 21.06.2024).

8. Brockman G. et al. OpenAI Gym. arXiv preprint arXiv:1606.01540, 2016. URL: https://arxiv.org/abs/1606.01540 (дата звернення: 23.06.2024).

9. Paszke A. et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. NeurIPS, 2019. URL: https://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library (дата звернення: 24.06.2024).

10. Silver D. A Brief Introduction to Reinforcement Learning. Springer, 2014. URL: https://www.springer.com/gp/book/9783319142091 (дата звернення: 25.06.2024).

11. Kingma D.P., Welling M. Auto-Encoding Variational Bayes. arXiv preprint arXiv:1312.6114, 2013. URL: https://arxiv.org/abs/1312.6114 (дата звернення: 25.06.2024).

12. Vaswani A. et al. Attention Is All You Need. NeurIPS, 2017. URL: https://arxiv.org/abs/1706.03762 (дата звернення:28.06.2024).

13. Brown T.B. et al. Language Models Are Few-Shot Learners. arXiv preprint arXiv:2005.14165, 2020. URL: https://arxiv.org/abs/2005.14165 (дата звернення: 29.06.2024).

14. Mnih V. et al. Human-Level Control through Deep Reinforcement Learning. Nature, 2015. URL: https://doi.org/10.1038/nature14236 (дата звернення: 01.07.2024).

15. Dosovitskiy A. et al. An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv preprint arXiv:2010.11929, 2020. URL: https://arxiv.org/abs/2010.11929 (дата звернення: 03.09.2024).

16. Bengio Y., Simard P., Frasconi P. Learning Long-Term Dependencies with Gradient Descent Is Difficult. IEEE Transactions on Neural Networks, 1994. URL: https://doi.org/10.1109/72.279181 (дата звернення: 5.09.2024).

17. LeCun Y., Bengio Y., Hinton G. Deep Learning. Nature, 2015. URL: https://doi.org/10.1038/nature14539 (дата звернення: 6.09.2024).

18. Karpathy A. The Unreasonable Effectiveness of Recurrent Neural Networks. Blog, 2015. URL: https://karpathy.github.io/2015/05/21/rnn-effectiveness/ (дата звернення: 6.09.2024).

19. Silver D. et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. Nature, 2016. URL: https://doi.org/10.1038/nature16961 (дата звернення: 11.09.2024).

20. Radford A. et al. Learning Transferable Visual Models from Natural Language Supervision. arXiv preprint arXiv:2103.00020, 2021. URL: https://arxiv.org/abs/2103.00020 (дата звернення: 12.09.2024).

21. Ruder S. An Overview of Gradient Descent Optimization Algorithms. arXiv preprint arXiv:1609.04747, 2016. URL: https://arxiv.org/abs/1609.04747 (дата звернення: 14. 09.2024).

22. Lillicrap T.P. et al. Continuous Control with Deep Reinforcement Learning. arXiv preprint arXiv:1509.02971, 2015. URL: https://arxiv.org/abs/1509.02971 (дата звернення: 17. 09.2024).

23. Haarnoja T. et al. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv preprint arXiv:1801.01290, 2018. URL: https://arxiv.org/abs/1801.01290 (дата звернення: 19.09.2024).

24. Bellemare M.G. et al. A Distributional Perspective on Reinforcement Learning. arXiv preprint arXiv:1707.06887, 2017. URL: https://arxiv.org/abs/1707.06887 (дата звернення: 22.09.2024).

25. Krizhevsky A., Sutskever I., Hinton G.E. ImageNet Classification with Deep Convolutional Neural Networks. NeurIPS, 2012. URL: https://dl.acm.org/doi/10.1145/3065386 (дата звернення: 27.09.2024).

26. Nguyen A., Yosinski J., Clune J. Deep Neural Networks Are Easily Fooled: High Confidence Predictions for Unrecognizable Images. CVPR, 2015. URL: https://doi.org/10.1109/CVPR.2015.7298640 (дата звернення: 28.09.2024).

27. Schaul T. et al. Prioritized Experience Replay. arXiv preprint arXiv:1511.05952, 2015. URL: https://arxiv.org/abs/1511.05952 (дата звернення: 02.10.2024).

28. Schulman J. et al. Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347, 2017. URL: https://arxiv.org/abs/1707.06347 (дата звернення: 03.10.2024).

29. Finn C., Abbeel P., Levine S. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. arXiv preprint arXiv:1703.03400, 2017. URL: https://arxiv.org/abs/1703.03400 (дата звернення: 04.10.2024).

30. Foerster J. et al. Learning to Communicate with Deep Multi-Agent Reinforcement Learning. arXiv preprint arXiv:1605.06676, 2016. URL: https://arxiv.org/abs/1605.06676 (дата звернення:06.10.2024).

31. Alpaydin E. Introduction to Machine Learning. MIT Press, 2020. URL: https://mitpress.mit.edu/books/introduction-machine-learning (дата звернення: 09.10.2024).

32. Williams R.J. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. Machine Learning, 1992. URL: https://doi.org/10.1007/BF00992696 (дата звернення: 09.10.2024).

33. Mohamed S., Rezende D.J. Variational Information Maximization for Intrinsically Motivated Reinforcement Learning. NeurIPS, 2015. URL: https://arxiv.org/abs/1509.08731 (дата звернення: 10.10.2024).

34. Ranzato M. et al. Sequence Level Training with Recurrent Neural Networks. arXiv preprint arXiv:1511.06732, 2015. URL: https://arxiv.org/abs/1511.06732 (дата звернення: 14.10.2024).

35. Hessel M. et al. Rainbow: Combining Improvements in Deep Reinforcement Learning. arXiv preprint arXiv:1710.02298, 2017. URL: https://arxiv.org/abs/1710.02298 (дата звернення: 17.10.2024).

36. Lake B.M., Ullman T.D., Tenenbaum J.B., Gershman S.J. Building Machines That Learn and Think Like People. Behavioral and Brain Sciences, 2017. URL: https://doi.org/10.1017/S0140525X16001837 (дата звернення: 20.10.2024).

37. Botvinick M. et al. Reinforcement Learning, Fast and Slow. Trends in Cognitive Sciences, 2019. URL: https://doi.org/10.1016/j.tics.2019.02.006 (дата звернення: 25.10.2024).

38. Kapturowski S. et al. Recurrent Experience Replay in Distributed Reinforcement Learning. arXiv preprint arXiv:1806.04664, 2018. URL: https://arxiv.org/abs/1806.04664 (дата звернення: 29.10.2024).

39. OpenAI. Learning Dexterity. arXiv preprint arXiv:1808.00177, 2018. URL: https://arxiv.org/abs/1808.00177 (дата звернення: 01.11.2024).

40. Oh J. et al. Action-Conditional Video Prediction Using Deep Networks in Atari Games. NeurIPS, 2015. URL: https://arxiv.org/abs/1507.08750 (дата звернення: 03.11.2024).

41. Graves A. Generating Sequences with Recurrent Neural Networks. arXiv preprint arXiv:1308.0850, 2013. URL: https://arxiv.org/abs/1308.0850 (дата звернення: 06.11.2024).

42. Jaderberg M. et al. Population Based Training of Neural Networks. arXiv preprint arXiv:1711.09846, 2017. URL: https://arxiv.org/abs/1711.09846 (дата звернення: 10.11.2024).

43. Rusu A.A. et al. Progressive Neural Networks. arXiv preprint arXiv:1606.04671, 2016. URL: https://arxiv.org/abs/1606.04671 (дата звернення: 16.11.2024).

44. Tesauro G. Temporal Difference Learning and TD-Gammon. Communications of the ACM, 1995. URL: https://doi.org/10.1145/203330.203343 (дата звернення: 21.11.2024).

45. Gu S. et al. Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic. arXiv preprint arXiv:1611.02247, 2016. URL: https://arxiv.org/abs/1611.02247 (дата звернення: 21.11.2024).

ДОДАТОК А

EnvironmentalAwareness

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnvironmentalAwareness
{
   public struct EnvironmentData
   {
      public bool[] obstacleDetected;
      public float[] obstacleDistances;
      public Vector3[] obstacleDirections;
      public Vector3[] safeDirections;
   }

   private Transform agentTransform;
   private float detectionRadius;
   private int rayCount;
   private LayerMask obstacleLayer;
         public float DetectionRadius => detectionRadius;

         public EnvironmentalAwareness(Transform transform, float radius = 10f, int rays = 16)
         {
                  agentTransform = transform;
                  detectionRadius = radius;
                  rayCount = rays;
                  obstacleLayer = LayerMask.GetMask("Obstacle", "Wall", "Environment");
         }

   public EnvironmentData ScanEnvironment()
   {
      EnvironmentData data = new EnvironmentData
      {
         obstacleDetected = new bool[rayCount],
         obstacleDistances = new float[rayCount],
         obstacleDirections = new Vector3[rayCount],
         safeDirections = new Vector3[rayCount]
      };

      float angleStep = 360f / rayCount;

      for (int i = 0; i < rayCount; i++)
      {
         float angle = i * angleStep;
         Vector3 direction = Quaternion.Euler(0, angle, 0) * agentTransform.forward;

         RaycastHit hit;
         if (Physics.Raycast(agentTransform.position, direction, out hit, detectionRadius, obstacleLayer))
         {
            data.obstacleDetected[i] = true;
            data.obstacleDistances[i] = hit.distance;
            data.obstacleDirections[i] = direction;
            data.safeDirections[i] = Vector3.zero;
         }
```

```
            else
            {
                data.obstacleDetected[i] = false;
                data.obstacleDistances[i] = detectionRadius;
                data.obstacleDirections[i] = direction;
                data.safeDirections[i] = direction;
            }
        }

        return data;
    }

    public Vector3 GetBestAvoidanceDirection(Vector3 desiredDirection, EnvironmentData envData)
    {

        int closestDirectionIndex = GetClosestDirectionIndex(desiredDirection, envData.obstacleDirections);
        if (!envData.obstacleDetected[closestDirectionIndex])
        {
            return desiredDirection;
        }

        float bestScore = float.MinValue;
        Vector3 bestDirection = desiredDirection;

        for (int i = 0; i < envData.safeDirections.Length; i++)
        {
            if (envData.safeDirections[i] == Vector3.zero) continue;

            float score = Vector3.Dot(desiredDirection, envData.safeDirections[i]) *
                    (envData.obstacleDistances[i] / detectionRadius);

            if (score > bestScore)
            {
                bestScore = score;
                bestDirection = envData.safeDirections[i];
            }
        }

        return bestDirection;
    }

    private int GetClosestDirectionIndex(Vector3 direction, Vector3[] directions)
    {
        float bestDot = –1f;
        int bestIndex = 0;

        for (int i = 0; i < directions.Length; i++)
        {
            float dot = Vector3.Dot(direction.normalized, directions[i].normalized);
            if (dot > bestDot)
            {
                bestDot = dot;
                bestIndex = i;
            }
        }

        return bestIndex;
    }
}
```

ДОДАТОК Б

DeerAgent

```
using System.Collections;
using System.Collections.Generic;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;
using System.Linq;
using UnityEngine;

public class DeerAgent : Agent
{
   public float speed = 2.5f;
   public float rotationSpeed = 100f;
   public float raycastDistance = 10f;
   public float fleeCooldown = 5f;

   private Rigidbody rb;
        private EnvironmentalAwareness awareness;
   private Vector3 initialPosition;
   private bool isFleeing = false;
   private float lastSeenWolfTime;
   private Vector3 lastPosition;
   private float idleTime = 0f;
        private float episodeStartTime;

        [Header("Environmental Settings")]
        [SerializeField] private float detectionRadius = 10f;
        [SerializeField] private int rayCount = 16;

   public override void Initialize()
   {
      rb = GetComponent<Rigidbody>();
      initialPosition = transform.localPosition;
      lastPosition = initialPosition;
                awareness = new EnvironmentalAwareness(transform, detectionRadius, rayCount);
   }

   public override void OnEpisodeBegin()
   {
      base.OnEpisodeBegin();
                episodeStartTime = Time.time;
          transform.localPosition = initialPosition;
      rb.velocity = Vector3.zero;
      rb.angularVelocity = Vector3.zero;
      isFleeing = false;
      lastSeenWolfTime = Time.time;
      lastPosition = initialPosition;
      idleTime = 0f;
   }

        private void Update()
   {
                if (Time.time – episodeStartTime >= 60f)
                {
                        EndEpisode();
                }
```

```
}

public override void CollectObservations(VectorSensor sensor)
        {
                sensor.AddObservation(transform.localPosition);
                sensor.AddObservation(rb.velocity);
                sensor.AddObservation(transform.forward);

                var envdata = awareness.ScanEnvironment();
                foreach (bool detection in envdata.obstacleDetected)
                {
                        sensor.AddObservation(detection);
                }
                foreach (float distance in envdata.obstacleDistances)
                {
                        sensor.AddObservation(distance / awareness.DetectionRadius);
                }
        }

public override void OnActionReceived(ActionBuffers actions)
{
   float forward = actions.ContinuousActions[0];
   float strafe = actions.ContinuousActions[1];
   float rotation = actions.ContinuousActions[2];

   Vector3 desiredMovement = (transform.forward * forward + transform.right * strafe).normalized;

   var envData = awareness.ScanEnvironment();
                Vector3 safeMovement = awareness.GetBestAvoidanceDirection(desiredMovement, envData);

   rb.velocity = safeMovement * speed;
   rb.angularVelocity = new Vector3(0, rotation * rotationSpeed, 0);

   bool nearObstacle = envData.obstacleDetected.Any(detected => detected);
                if (nearObstacle && Vector3.Distance(desiredMovement, safeMovement) > 0.1f)
                {
                        AddReward(0.05f);
                }

   if (isFleeing)
   {
      FleeFromWolf();
   }
   else
   {
      if (RaycastForWolves())
      {
         isFleeing = true;
         lastSeenWolfTime = Time.time;
      }
      else if (Time.time – lastSeenWolfTime > fleeCooldown)
      {
         AddMovementReward();
      }
   }

   if (rb.velocity.magnitude < 0.01f)
   {
      idleTime += Time.deltaTime;
      if (idleTime > 1f)
```

```
        {
            AddReward(–0.01f);
        }
    }
    else
    {
        idleTime = 0f;
    }
    AddReward(–0.001f);
}

private void AddMovementReward()
{
    float distanceMoved = Vector3.Distance(transform.localPosition, lastPosition);
    if (distanceMoved > 0.1f)
    {
        AddReward(0.01f);
    }
    lastPosition = transform.localPosition;
}

private bool RaycastForWolves()
{
    RaycastHit hit;
    if (Physics.Raycast(transform.position, transform.forward, out hit, raycastDistance))
    {
        if (hit.collider.CompareTag("Wolf"))
        {
            return true;
        }
    }
    return false;
}

private void FleeFromWolf()
{
    RaycastHit hit;
    if (Physics.Raycast(transform.position, transform.forward, out hit, raycastDistance))
    {
        if (hit.collider.CompareTag("Wolf"))
        {
            Vector3 fleeDirection = transform.position – hit.collider.transform.position;
            rb.velocity = fleeDirection.normalized * speed;
            AddReward(0.01f);
        }
    }
    else
    {
        isFleeing = false;
    }
}

private void OnCollisionEnter(Collision collision)
        {
                if (collision.gameObject.CompareTag("Wolf"))
                {
                        AddReward(–1.0f);
                        EndEpisode();
                }
        }
```

```
    public override void Heuristic(in ActionBuffers actionsOut)
    {
        var continuousActions = actionsOut.ContinuousActions;
        continuousActions[0] = Input.GetAxis("Vertical");
        continuousActions[1] = Input.GetAxis("Horizontal");
        continuousActions[2] = Input.GetAxis("Mouse X");
    }
}
```

ДОДАТОК В

WolfAlphaAgent

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;

public class WolfAlphaAgent : Agent
{
   [Header("Agent Settings")]
   public float speed = 3f;
            public float rotationSpeed = 100f;
   private Rigidbody rb;
           private EnvironmentalAwareness awareness;

   [Header("Environment References")]
   public Transform targetDeer;
   public List<WolfPackMateAgent> wolfPack;
   private const float maxDistance = 20f;

   [Header("Prediction Settings")]
   public float predictionTime = 2f;
   public int predictionSteps = 5;
   private Vector3 predictedDeerPosition;
   private Vector3 deerVelocity;
   private Vector3 lastDeerPosition;
   private float timeSinceLastPrediction;
   private const float predictionUpdateInterval = 0.2f;
           private float episodeStartTime;

   private float previousDistanceToDeer;
   private float distanceToPredictedPosition;

           [Header("Environmental Settings")]
           [SerializeField] private float detectionRadius = 10f;
           [SerializeField] private int rayCount = 16;

   public override void Initialize()
   {
      rb = GetComponent<Rigidbody>();
      lastDeerPosition = targetDeer.position;
      predictedDeerPosition = targetDeer.position;
                        awareness = new EnvironmentalAwareness(transform, detectionRadius, rayCount);
   }

   public override void OnEpisodeBegin()
   {
                        base.OnEpisodeBegin();
                        episodeStartTime = Time.time;
      rb.velocity = Vector3.zero;
      rb.angularVelocity = Vector3.zero;
      transform.localPosition = new Vector3(Random.Range(–5f, 5f), 0, Random.Range(–5f, 5f));
      previousDistanceToDeer = Vector3.Distance(transform.localPosition, targetDeer.transform.localPosition);
      lastDeerPosition = targetDeer.position;
```

```
    predictedDeerPosition = targetDeer.position;
    deerVelocity = Vector3.zero;
    timeSinceLastPrediction = 0f;
}

private void Update()
{
    UpdateDeerPrediction();
                if (Time.time – episodeStartTime >= 60f)
                {
                        EndEpisode();
                }
}

private void UpdateDeerPrediction()
{
    timeSinceLastPrediction += Time.deltaTime;

    if (timeSinceLastPrediction >= predictionUpdateInterval)
    {
        Vector3 currentDeerPosition = targetDeer.position;
        deerVelocity = (currentDeerPosition – lastDeerPosition) / predictionUpdateInterval;
        lastDeerPosition = currentDeerPosition;

        predictedDeerPosition = PredictDeerPosition();

        timeSinceLastPrediction = 0f;
    }
}

private Vector3 PredictDeerPosition()
{
    Vector3 deerPos = targetDeer.position;
    Vector3 bestPrediction = deerPos;
    float bestScore = float.MinValue;

    for (int i = 0; i < predictionSteps; i++)
    {
        float timeOffset = (i + 1) * (predictionTime / predictionSteps);
        Vector3 predictedPos = deerPos + (deerVelocity * timeOffset);

        float score = EvaluatePredictionPoint(predictedPos);

        if (score > bestScore)
        {
            bestScore = score;
            bestPrediction = predictedPos;
        }
    }

    return bestPrediction;
}

private float EvaluatePredictionPoint(Vector3 predictedPos)
{
    float score = 0f;

    float distanceFromCurrent = Vector3.Distance(targetDeer.position, predictedPos);
    score –= distanceFromCurrent * 0.5f;
```

```
    if (Mathf.Abs(predictedPos.x) > maxDistance || Mathf.Abs(predictedPos.z) > maxDistance)
    {
        score –= 1000f;
    }

    float packPositionScore = 0f;
    foreach (var packmate in wolfPack)
    {
        float distanceToPackmate = Vector3.Distance(predictedPos, packmate.transform.position);
        if (distanceToPackmate < 5f)
        {
            packPositionScore += 10f;
        }
    }
    score += packPositionScore;

                RaycastHit hit;
        if (Physics.Raycast(transform.position, predictedPos – transform.position, out hit,
            Vector3.Distance(transform.position, predictedPos)))
        {
            if (hit.collider.CompareTag("Obstacle"))
            {
                score –= 1000f;
            }
        }

    return score;
}

public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(transform.localPosition);
    sensor.AddObservation(targetDeer.localPosition);

    sensor.AddObservation(predictedDeerPosition);
    sensor.AddObservation(deerVelocity);

    sensor.AddObservation(Vector3.Distance(transform.localPosition, predictedDeerPosition));

    foreach (var packmate in wolfPack)
    {
        sensor.AddObservation(packmate.transform.localPosition);
    }

            var envdata = awareness.ScanEnvironment();
            foreach (bool detection in envdata.obstacleDetected)
            {
                sensor.AddObservation(detection);
            }
            foreach (float distance in envdata.obstacleDistances)
            {
                sensor.AddObservation(distance / awareness.DetectionRadius);
            }
}

public override void OnActionReceived(ActionBuffers actions)
{
    float forward = actions.ContinuousActions[0];
    float strafe = actions.ContinuousActions[1];
    float rotation = actions.ContinuousActions[2];
```

```
    Vector3 desiredMovement = (transform.forward * forward + transform.right * strafe).normalized;
    var envData = awareness.ScanEnvironment();
    Vector3 safeMovement = awareness.GetBestAvoidanceDirection(desiredMovement, envData);

    rb.velocity = safeMovement * speed;
    rb.angularVelocity = new Vector3(0, rotation * 10f, 0);

    RewardForProximityToDeer();
    RewardForLeadingPack();
    RewardForPrediction();

    rb.velocity = safeMovement * speed;
    rb.angularVelocity = new Vector3(0, rotation * rotationSpeed, 0);

    bool nearObstacle = envData.obstacleDetected.Any(detected => detected);
                if (nearObstacle && Vector3.Distance(desiredMovement, safeMovement) > 0.1f)
                {
                        AddReward(0.05f);
                }
}

private void RewardForProximityToDeer()
{
    float currentDistanceToDeer = Vector3.Distance(transform.localPosition, targetDeer.transform.localPosition);

    if (currentDistanceToDeer < previousDistanceToDeer)
    {
        AddReward(0.1f);
    }
    else
    {
        AddReward(–0.05f);
    }

    if (currentDistanceToDeer < 1.5f)
    {
        AddReward(1.0f);
    }

    previousDistanceToDeer = currentDistanceToDeer;
}

private void RewardForPrediction()
{
    float currentDistanceToPredicted = Vector3.Distance(transform.localPosition, predictedDeerPosition);

    if (currentDistanceToPredicted < distanceToPredictedPosition)
    {
        AddReward(0.15f);
    }

    if (Vector3.Distance(targetDeer.position, predictedDeerPosition) < 2f &&
        Vector3.Distance(transform.localPosition, predictedDeerPosition) < 3f)
    {
        AddReward(0.5f);
    }

    distanceToPredictedPosition = currentDistanceToPredicted;
}
```

```
    private void RewardForLeadingPack()
    {
        if (Vector3.Distance(transform.localPosition, targetDeer.localPosition) < 5f)
        {
            foreach (var packmate in wolfPack)
            {
                float distanceToPackmate = Vector3.Distance(transform.localPosition, packmate.transform.localPosition);
                if (distanceToPackmate < 5f)
                {
                    AddReward(0.3f);
                }
            }
        }
    }

            private void OnCollisionEnter(Collision collision)
            {
                    if (collision.gameObject.CompareTag("Deer"))
                    {
                            AddReward(10.0f);
                            EndEpisode();
                    }
            }

    public override void Heuristic(in ActionBuffers actionsOut)
    {
        var continuousActions = actionsOut.ContinuousActions;
        continuousActions[0] = Input.GetAxis("Vertical");
        continuousActions[1] = Input.GetAxis("Horizontal");
        continuousActions[2] = Input.GetAxis("Mouse X");
    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(predictedDeerPosition, 0.5f);
        Gizmos.DrawLine(targetDeer.position, predictedDeerPosition);
    }
}
```

ДОДАТОК Г

WolfPackMateAgent

```
using UnityEngine;
using System.Linq;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;
using System.Collections.Generic;

public class WolfPackMateAgent : Agent
{
   [Header("Agent Settings")]
   public float speed = 2.5f;
        public float rotationSpeed = 100f;
   private Rigidbody rb;
        private EnvironmentalAwareness awareness;
        private float episodeStartTime;

   [Header("Environment References")]
   public Transform targetDeer;
   public WolfAlphaAgent alphaWolf;
   public int packMateIndex;

   [Header("Behavior Settings")]
   [SerializeField] private float surroundDistance = 5f;
   [SerializeField] private float optimalPackDistance = 3f;
   [SerializeField] private float maxPackDistance = 8f;
   [SerializeField] private float minPackDistance = 2f;

   private Vector3 previousPosition;
   private float previousDistanceToDeer;
   private Vector3 assignedFlankPosition;
   private float timeSinceLastPositionUpdate;
   private const float positionUpdateInterval = 0.5f;

        [Header("Environmental Settings")]
        [SerializeField] private float detectionRadius = 10f;
        [SerializeField] private int rayCount = 16;

   private new void Awake()
   {
      if (packMateIndex == 0)
      {
         Debug.LogWarning($"Pack mate {gameObject.name} has default index 0. Assign unique indices to pack
members.");
      }
   }

   public override void Initialize()
   {
      rb = GetComponent<Rigidbody>();
      previousPosition = transform.position;
      AssignFlankPosition();
                awareness = new EnvironmentalAwareness(transform, detectionRadius, rayCount);
   }
```

```
public override void OnEpisodeBegin()
{
		base.OnEpisodeBegin();
		episodeStartTime = Time.time;
   rb.velocity = Vector3.zero;
   rb.angularVelocity = Vector3.zero;
   transform.localPosition = new Vector3(Random.Range(–5f, 5f), 0, Random.Range(–5f, 5f));
   previousPosition = transform.position;
   previousDistanceToDeer = Vector3.Distance(transform.position, targetDeer.position);
   AssignFlankPosition();
   timeSinceLastPositionUpdate = 0f;
}

private void Update()
{
   timeSinceLastPositionUpdate += Time.deltaTime;
   if (timeSinceLastPositionUpdate >= positionUpdateInterval)
   {
      AssignFlankPosition();
      timeSinceLastPositionUpdate = 0f;
   }
		if (Time.time – episodeStartTime >= 60f)
		{
			EndEpisode();
		}
}

public override void CollectObservations(VectorSensor sensor)
{
   sensor.AddObservation(transform.localPosition);
   sensor.AddObservation(alphaWolf.transform.localPosition);
   sensor.AddObservation(targetDeer.localPosition);

   sensor.AddObservation(assignedFlankPosition);

   sensor.AddObservation(rb.velocity);
   sensor.AddObservation(transform.forward);

   sensor.AddObservation(Vector3.Distance(transform.localPosition, targetDeer.localPosition));
   sensor.AddObservation(Vector3.Distance(transform.localPosition, alphaWolf.transform.localPosition));
   sensor.AddObservation(Vector3.Distance(transform.localPosition, assignedFlankPosition));

		var envdata = awareness.ScanEnvironment();
		foreach (bool detection in envdata.obstacleDetected)
		{
			sensor.AddObservation(detection);
		}
		foreach (float distance in envdata.obstacleDistances)
		{
			sensor.AddObservation(distance / awareness.DetectionRadius);
		}

   if (alphaWolf.wolfPack != null)
   {
      foreach (var packmate in alphaWolf.wolfPack)
      {
         if (packmate != this)
         {
            sensor.AddObservation(packmate.transform.localPosition);
            sensor.AddObservation(Vector3.Distance(transform.localPosition, packmate.transform.localPosition));
```

```
            }
        }
    }
}

    private void AssignFlankPosition()
    {
        Vector3 directionToDeer = (targetDeer.position – alphaWolf.transform.position).normalized;
        float angleOffset = GetFlankAngleOffset();

        assignedFlankPosition = targetDeer.position +
            (Quaternion.Euler(0, angleOffset, 0) * directionToDeer * surroundDistance);

        AdjustForObstacle();
    }

    private float GetFlankAngleOffset()
    {
        if (packMateIndex % 2 == 0)
        {
            return 45f + (15f * (packMateIndex / 2));
        }
        else
        {
            return –45f – (15f * (packMateIndex / 2));
        }
    }

    private void AdjustForObstacle()
    {
        RaycastHit hit;
        if (Physics.Raycast(transform.position, assignedFlankPosition – transform.position, out hit,
            Vector3.Distance(transform.position, assignedFlankPosition)))
        {
            if (hit.collider.CompareTag("Obstacle"))
            {
                var envData = awareness.ScanEnvironment();
                Vector3 direction = (assignedFlankPosition – transform.position).normalized;
                assignedFlankPosition = transform.position +
                    awareness.GetBestAvoidanceDirection(direction, envData) *
surroundDistance;
            }
        }
    }

public override void OnActionReceived(ActionBuffers actions)
{
    float forward = actions.ContinuousActions[0];
    float strafe = actions.ContinuousActions[1];
    float rotation = actions.ContinuousActions[2];

    Vector3 desiredMovement = (transform.forward * forward + transform.right * strafe).normalized;
    var envData = awareness.ScanEnvironment();
    Vector3 safeMovement = awareness.GetBestAvoidanceDirection(desiredMovement, envData);

    rb.velocity = safeMovement * speed;
    rb.angularVelocity = new Vector3(0, rotation * 10f, 0);

    CalculateMovementReward();
    CalculateFlankingReward();
```

```
    CalculatePackCoordinationReward();
    CalculateFormationReward();

    previousPosition = transform.position;
    previousDistanceToDeer = Vector3.Distance(transform.position, targetDeer.position);

    rb.velocity = safeMovement * speed;
    rb.angularVelocity = new Vector3(0, rotation * rotationSpeed, 0);

    bool nearObstacle = envData.obstacleDetected.Any(detected => detected);
                if (nearObstacle && Vector3.Distance(desiredMovement, safeMovement) > 0.1f)
                {
                        AddReward(0.05f);
                }
}

private void CalculateMovementReward()
{
    float currentDistanceToDeer = Vector3.Distance(transform.position, targetDeer.position);

    if (currentDistanceToDeer < previousDistanceToDeer && currentDistanceToDeer > minPackDistance)
    {
        AddReward(0.1f);
    }
    else if (currentDistanceToDeer < minPackDistance)
    {
        AddReward(–0.2f);
    }

    if (currentDistanceToDeer > maxPackDistance)
    {
        AddReward(–0.1f * (currentDistanceToDeer – maxPackDistance));
    }
}

private void CalculateFlankingReward()
{
    float distanceToFlank = Vector3.Distance(transform.position, assignedFlankPosition);

    if (distanceToFlank < 2f)
    {
        AddReward(0.3f);

        Vector3 directionToDeer = (targetDeer.position – transform.position).normalized;
        float angleToDeer = Vector3.Angle(transform.forward, directionToDeer);
        if (angleToDeer > 60f && angleToDeer < 120f)
        {
            AddReward(0.2f);
        }
    }
}

private void CalculatePackCoordinationReward()
{
    float distanceToAlpha = Vector3.Distance(transform.position, alphaWolf.transform.position);

    if (distanceToAlpha >= minPackDistance && distanceToAlpha <= maxPackDistance)
    {
        AddReward(0.2f);
    }
```

```
    else
    {
        AddReward(–0.1f);
    }

    foreach (var packmate in alphaWolf.wolfPack)
    {
        if (packmate != this)
        {
            float distanceToPackmate = Vector3.Distance(transform.position, packmate.transform.position);
            if (distanceToPackmate < minPackDistance)
            {
                AddReward(–0.2f);
            }
        }
    }
}

private void CalculateFormationReward()
{
    bool isInFormation = true;
    Vector3 deerToAlpha = alphaWolf.transform.position – targetDeer.position;

    foreach (var packmate in alphaWolf.wolfPack)
    {
        if (packmate != this)
        {
            Vector3 deerToPackmate = packmate.transform.position – targetDeer.position;
            float angle = Vector3.Angle(deerToAlpha, deerToPackmate);

            if (angle < 30f)
            {
                isInFormation = false;
                break;
            }
        }
    }

    if (isInFormation)
    {
        AddReward(0.5f);
    }
}

        private void OnCollisionEnter(Collision collision)
        {
                if (collision.gameObject.CompareTag("Deer"))
                {
                        AddReward(10.0f);
                        EndEpisode();
                }
        }

public override void Heuristic(in ActionBuffers actionsOut)
{
    var continuousActions = actionsOut.ContinuousActions;
    continuousActions[0] = Input.GetAxis("Vertical");
    continuousActions[1] = Input.GetAxis("Horizontal");
    continuousActions[2] = Input.GetAxis("Mouse X");
}
```

```
    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(assignedFlankPosition, 0.5f);
        Gizmos.DrawLine(transform.position, assignedFlankPosition);

        Gizmos.color = Color.green;
        Gizmos.DrawWireSphere(transform.position, optimalPackDistance);
    }

}
```

ДОДАТОК Ґ

Hunting.yaml

```
behaviors:
  WolfA:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 256
      num_layers: 3
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.995
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 5000000
    time_horizon: 500
    summary_freq: 30000

  WolfP:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
```

```
    learning_rate_schedule: linear
  network_settings:
    normalize: true
    hidden_units: 256
    num_layers: 3
    vis_encode_type: simple
  reward_signals:
    extrinsic:
      gamma: 0.995
      strength: 1.0
  keep_checkpoints: 5
  max_steps: 5000000
  time_horizon: 500
  summary_freq: 30000

DeerB:
  trainer_type: ppo
  hyperparameters:
    batch_size: 1024
    buffer_size: 10240
    learning_rate: 0.0003
    beta: 0.005
    epsilon: 0.2
    lambd: 0.95
    num_epoch: 3
    learning_rate_schedule: linear
  network_settings:
    normalize: true
    hidden_units: 128
    num_layers: 2
    vis_encode_type: simple
  reward_signals:
    extrinsic:
      gamma: 0.99
      strength: 1.0
  keep_checkpoints: 5
  max_steps: 5000000
  time_horizon: 300
  summary_freq: 30000
```

ДОДАТОК Ґ

Апробація роботи

Робота пройшла апробацію під час Всеукраїнської науково–практичної конференції молодих вчених, аспірантів і студентів, 2–4 грудня 2024 р., м. Миколаїв.

УДК 004.42

Рева В.В., Кулаковська І.В.
Чорноморський національний університет імені Петра Могили, Миколаїв, Україна

ІНТЕГРОВАНА СИСТЕМА ДЛЯ НАВЧАННЯ АГЕНТІВ UNITY ML AGENTS З ВИКОРИСТАННЯМ МОДЕЛЕЙ ГЛИБОКИХ НЕЙРОННИХ МЕРЕЖ

WolfAlphaAgent

Цей скрипт представляє альфа–вовка у симуляції вовчої зграї за допомогою ML–агентів Unity. Він очолює зграю і прогнозує майбутнє положення оленя, щоб оптимізувати його рух і координацію.

WolfPackMateAgent

Цей скрипт координує дії окремих членів зграї з альфа–вовком. Позиціонування у формації шляхом розподілу позицій навколо оленя для утримання його у центрі оточення. Автоматичне коригування розташування, враховуючи позиції альфа–вовка та інших членів зграї.

DeerAgent

Цей скрипт моделює оленя, здобич у симуляції, і його основна мета – уникати вовків. Використання raycast для збору даних про оточення для уникнення перешкод. Здатність змінювати напрямок руху для безпечного пересування.

Дослідження показали, що застосування моделей глибокого навчання суттєво підвищує ефективність агентів, особливо в сценаріях із високою складністю. Використання алгоритмів, таких як Proximal Policy Optimization (PPO), дозволило досягти стабільного навчання та високих показників винагороди в більшості тестових сценаріїв.

Інтеграція Unity ML-Agents із глибокими нейронними мережами відкриває нові можливості для розвитку інтелектуальних систем, які можуть бути використані не лише в ігровій індустрії, але й у реальних застосунках, таких як робототехніка чи управління автономними системами.