Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи забезпечення адаптивності ігрового штучного інтелекту

(тема)

Виконав:

студент II курсу, групи СПм-20-1

Супруненко М.Ю.
(прізвище, ініціали)

Спеціальність

123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма

Системне програмування
(повна назва освітньої програми)

Керівник: доц. Іващенко Г.С.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ **20__ р.**

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Супруненку Максиму Юрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Методи забезпечення адаптивності ігрового штучного інтелекту

затверджена наказом по університету від “ 5 ” листопада 2021 р. № 1657 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 13 грудня 2021 р.

3. Вхідні дані до роботи

1) документація для створення ігрового агенту на основі навчання з підкріпленням
2) мова програмування Python
3) бібліотека PyTorch
4) програмний засіб HandyRL

4. Перелік питань, що потрібно опрацювати у роботі

1) Дослідження методів забезпечення адаптивності ігрового штучного інтелекту та створення агенту
2) Створення ігрового середовища для ігри «Змійка»
3) Порівняння розробленого адаптивного агенту з існуючими реалізаціями ігрового штучного інтелекту у рамках змагання на платформі Kaggle

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Слайд-презентація – 13 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Вибір та обгрунтування методики дослідження	09.11.21-18.11.21	
2	Вибір інструментальних засобів	19.11.21-22.11.21	
3	Проведення експериментів	23.11.21-29.11.21	
4	Оформлення матеріалів кваліфікаційної роботи	29.11.21-30.11.21	
5	Подання кваліфікаційної роботи керівникові та	01.12.21-02.12.21	
	її попередній захист	.	
6	Подання кваліфікаційної роботи на рецензу-	03.12.21-04.12.21	
	вання		

Дата видачі завдання 8 листопада 2021 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Іващенко Г.С.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 76 с., 25 рис., 5 табл., 28 джерел.

REINFORCEMENT LEARNING, SNAKE GAME, DEEP Q-LEARNING, MONTE-CARLO SEARCH TREE, ARTIFICIAL NEURAL NETWORKS, PYTHON.

Метою роботи є дослідження агентів в ході якого потрібно створити ігровий штучний інтелект для гри «Змійка», побудований на основі адаптивного ігрового штучного інтелекту використовуючи навчання з підкріпленням та глибині нейронні мережі. Агент повинен вчитися на основі попереднього досвіду і демонструвати результат під час гри с користувачем.

У ході виконання кваліфікаційної роботи створено адаптивний ігровий агент на основі штучної нейронної мережі для ігри «Змійка», створено середовище виконання агенту для ігри в «Змійку», оцінено результати тренування та валідації агенту при виконанні дослідження та проведено ігрову партію створеного агенту з конкурентами, і проведено широкий аналіз роботи створеного агенту порівняно з конкурентами і надано кроки для покрашення агенту, створеного в рамках кваліфікаційної роботи.

ABSTRACT

Master’s thesis: 76 pages, 20 figures, 5 tables, 28 sources.

REINFORCEMENT LEARNING, SNAKE, DEEP Q-LEARNING, MONTE-CARLO SEARCH TREE, NEURAL NETWORKS, PYTHON.

The aim of the work is to study the work of agents in the course of which it is necessary to create a game artificial intelligence for the game "Snake", built on adaptive game artificial intelligence, using training with reinforcement and depth of the neural network. The agent must learn from previous experience and demonstrate the result while playing with the user. As the number of players played with the agent grows, the difficulty of advancing to the next game should increase.

In the course of the qualification work, an adaptive game agent based on SNM for the game "Snake" was created, an environment for the agent to play the game "Snake" was created, the results of training and confirmation of the agent during the research were evaluated. extensive analysis of the work of the agent created interaction with competitors and the steps taken to improve the agent created in the qualification work.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШІ – штучний інтелект

ШНМ – штучна нейронна мережа

МС – алгоритм Монте Карло (англ., Monte-Carlo)

MCST – дерево пошуку Монте-Карло (англ., Monte-Carlo Search Tree)

RL – навчання з підкріпленням (англ., Reinforcement learning)

ВСТУП

Ігровий штучний інтелект – набір алгоритмів з теорії управління, робототехніки, комп’ютерної графіки та інформатики, які виконують автоматичні обчислення на основі наборів вхідних даних та генерують обмежений набір відповідей [1, 5].

У відеоіграх штучний інтелект використовується для генерування адаптивної або інтелектуальної поведінки неігрових персонажів (NPC), подібних до людського інтелекту. Штучний інтелект був невід’ємною частиною відеоігор з моменту їх появи в 1950-х роках. Він служить для покращення досвіду гравця при ігровій сесії.

Реалізація ігрового штучного інтелекту являється нетривіальною проблемою, так як потребує досвіду в розробці ігрових алгоритмів та значних обчислювальних ресурсів. З 1950-х років розробники досліджували агенти ігрового штучного інтелекту, яким була дана можливість вчитися на основі отриманого ігрового досвіду, замість того, щоб додержуватися тактики чи стратегії, яка була статично задана геймдизайнером чи розробником відеоігор. Існують традиційні підходи, такі як динамічне програмування, які могли б вирішити проблему, але вони є ефективними тільки на невеликій виборці даних, що зібрані під час дій агенту на ігровому полі.

В останнє десятиріччя почався розвиток підходів розробки агентів ігрового штучного інтелекту, що засновані на використанні навчання з підкріпленням [21-24].

За останні кілька років Reinforcement learning (RL) став все більш популярним завдяки ефективності при вирішенні складних проблем. Деякі з досягнень обумовлені поєднанням RL з методами глибокого навчання [5]. Ця комбінація, яка називається глибоким RL, є найбільш корисною в задачах з багатокомпонентною матрицею станів великої розмірності.

Традиційні RL підходи були малоефективними з точки зори дизайну на швидкості роботи, але RL, який побудований на основі нейронних мереж, успішний при вирішенні складних завдань, в яких необхідно враховувати попередній досвід, що дозволяло системі вивчати різні рівні абстракції даних. Наприклад, глибокий RL-агент може успішно навчатися з візуальних перцептивних вхідних даних, що складаються з тисяч пікселів [5]. Це відкриває можливість імітувати деякі людські можливості вирішення проблем з великою кількістю даних. Кілька помітних робіт з використанням глибокого RL в іграх виділилися досягненням надлюдського рівня в іграх Atari [4], оволодіння Go [19].

Deep RL також має потенціал для таких сфер, як робототехніка, самокеровані автомобілі, інтелектуальні мережі та ігри. В застосуванні глибоких алгоритмів RL виникає кілька проблем, серед яких слід визначити нездатність ефективно досліджувати навколишнє середовище або здатність до узагальнення. Таким чином, було запропоновано великий набір алгоритмів для Deep RL, залежно від різноманітних налаштувань та послідовних прийняття рішень [25].

Типовою грою для дослідження підходів забезпечення адаптивності ігрового штучного інтелекту є традиційна гра «Змійка» [26]. Використання для неї штучних нейронних мереж (ШНМ) та Reinforcement learning дозволяє враховувати геймплейний досвід гравця.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Обґрунтування актуальності роботи

Основною стратегією розробників із створенню ігрового штучного інтелекту (ШІ) являлося створення статичних правил, що є самим простим рішенням для поліпшення проблеми створення ігрового ШІ. Недоліком статичного штучного інтелекту є відсутність гнучкості стратегії, що унеможливлює адаптування ШІ, особливо помітне при наявності великої кількості гравців на одному полі [5].

Ігрові ШІ на основі навчання з підкріпленням є досить популярними, через що до сих пір досліджується і виявляють різні способи покращення існуючих підходів. Одним із самих популярних напрямків являється навчання традиційних ігор, таких як «Змійка» [5].

«Змійка» – це класична гра, в якій гравець керує змією, щоб максимізувати кількість зібраної їжі (фруктів), яка появляється в випадкових місцях [7]. Їжа з’являється на полі гри в будь-який час. Змія росте в довжину на одну клітинку якщо з’їсть фрукт. Гра закінчується коли змія зіштовхується з перешкодою, чи з іншою змією, чи сама з собою. В «Змійку» можуть грати декілька гравців, що може ускладнити гру, особливо, при використанні ботів – змійок під керуванням ігрових ШІ.

Ігровий штучний інтелект може бути корисний для покращення геймплею для фінального користувача при добавлені нових ШІ в ігрову систему. Це може забезпечити кращий загальний досвід користувача, так як агент не є передбачуваним і може навчатися в реальному часі [7].

Створення агентів ігрового штучного інтелекту є популярним завданням серед розробників на платформі Kaggle [8]. Kaggle – це платформа для проведення змагань із машинного навчання, де кожен із учасників може приєднатися до змагання і виконати завдання, встановлене власником

змагання. Традиційним завдання є створення моделі машинного навчання яка показає результат згідно метрики, заданої власником змагання. Виграє той учасник, в якого метрика показує найкращий результат серед всіх гравців, які виконали роботу. Платформою володіє компанія Google, що представила змагання під назвою «Hungry Geese» [9, 11].

Метою змагання є створення ігрового штучного інтелекту на основі алгоритмів машинного навчання для ігри «Змійка» та створення ігрового агенту, який здатний вижити протягом найбільшого часу. Також важливим критерієм перемоги є кількість зібраних фруктів.

Актуальність дослідження ігрового ШІ обумовлена можливістю використання цих підходів не тільки у ігрових застосунках, але й при програмуванні автономних апаратно-програмних комплексів, таких як побутові роботи. Також актуальності сприяє поширення у користувачів спеціалізованих обчислювальних ресурсів (таких, як тензорні ядра), що дозволяє використовувати ШНМ для недетермінованої поведінки та враховувати досвід [13].

1.2 Різновиди ігрового штучного інтелекту

Комп’ютерні ігри можна розглядати як стенд для вивчення штучного інтелекту. У минулому велика частина академічної роботи над штучним інтелектом та іграми була зосереджена на традиційних настільних іграх, таких як шашки та шахи. На цих іграх аналізувалася ефективність методів ШІ. Серед засобів ШІ слід зазначити множину методів, побудованих по аналогії з процесами та явищами реального світу – такі методи, як еволюційні алгоритми, штучні нейронні мережі та нечітка логіка . Ці методи можна застосовувати також при розвитку ігор, у різних аспектах. Наприклад, інтелектуальна чи адаптивна поведінка NPC, нелінійна розповідь або більш насичений зміст гри (у формі, наприклад, рівнів, карти, місцевості, музики чи навіть самих правил гри) [5]. Крім того, з точки зору галузі, штучний

інтелект починають розглядати як механізм покращення процесу розробки ігор, а також інструмент для розширення комерційного життя ігор.

Методи штучного інтелекту, застосовані для досягнення мети в одній конкретній грі, можуть бути застосовані до подібної гри, що належить до того ж жанру, що надає можливість зменшити витрати на розробку нових ігор.

Існують два типу ігрового штучного інтелекту: статично заданий штучний інтелект та адаптивний (динамічний) штучний інтелект [1].

Статичний інтелект задається набором правил, які є константними на протязі всього ігрового процесу. Його головною особливістю являється прогнозованість дій в заданій системі. Наприклад, у грі «Змійка» це проявляється у вигляді того, що змійка прагне уникнути колізії зі своїм хвостом та виконує невелике прогнозування на наступні клітинки вперед. Прогнозування є однією із самих слабких частин статично заданого алгоритму, побудованого на правилах. Комплексна система прогнозування ускладнює розширення компонентів в алгоритмі, що в результаті може показати неспроможність до покращення штучного інтелекту в цілому. Складність розробки значно збільшується при прогнозуванні на 10 кроків вперед.

В адаптивному ігровому штучному інтелекті відсутні недоліки, які має статичний ігровий інтелект. Динамічний штучний інтелект враховує ігровий досвід, отриманий під час ігрової сесії. Однією із важливих переваг являється врахування всіх симуляцій даної ігри. Новітні процесори та графічні відеокарти надають можливість використати достатні ресурси для симуляції великої кількості ігор і збирати отриманий досвід для навчання ігрового штучного інтелекту перед грою з реальним гравцем.

Наприклад, ігрові агенти, які використовують новітні архітектури нейронних мереж, здатні зберігати та акумулювати досвід у вигляді вагових коефіцієнтів в нейронних шарах. Даний досвід можливо поновлювати шляхом врахування нових симуляцій в довільний момент часу.

Перевагою динамічного ігрового штучного інтелекту є дослідження неявних способів досягання результату (перемоги у ігровій партії). RL дозволяє вказати коефіцієнт дослідження, який показує можливість виконання кроку, відмінного від отриманого досвіду. При виконанні великої кількості симуляцій, система може дослідити нові шляхи, незважаючи на попередній досвід, і здатна удосконалити поточний ігровий штучний інтелект автоматично.

У контексті розробки агентів для гри «Змійка» можна виділити такі переваги використання статичного штучного інтелекту:

- працює на основі заданих правил;
- легко інтерпретується, так як правила задані людиною;
- варіанти реалізацій статичного ШІ є достатньо дослідженими;
- відрізняється простотою початкової реалізації.

Недоліками є:

- нелінійне зростання складності системи із збільшенням кількості правил;
- проблема урахування в правилах одночасної присутності декількох агентів на одному ігровому полі. У випадку «Змійки» це призводить до зіткнень;
- складність створення декількох агентів в рамках однієї системи.

Серед переваг використання динамічного штучного інтелекту, який базується на основі попереднього ігрового досвіду (адаптивного ігрового ШІ) слід зазначити:

- ШІ навчається на основі попереднього ігрового досвіду, що отриманий під час великої кількості зіграних ігор підряд;
- можливість запобігти зіткненням з іншими агентами на протязі тривалої ігрової сесії;
- можливість виконання дослідження нових способів дії ігрового штучного інтелекту за допомогою задання коефіцієнту дослідження.

Недоліками є:

- штучний інтелект досить складно інтерпретувати, ігровий досвід зберігається у формі ваг в агенті нейронної мережі;
- потребує значних ресурсів відеокарти для навчання нейронної мережі;
- вимагає значну кількість симуляцій для отримання результату.

1.3 Існуючі реалізації адаптивного ігрового штучного інтелекту

Досліджено існуючі реалізації ігрового штучного інтелекту для ігри «Змійка». Можливо виділити загальні характеристики, що притаманні розглянутим реалізаціям [7, 13, 26]:

- модель для навчання ігрового адаптивного агенту;
- баланс між дослідженням та використанням;
- використовувана евристика;
- аналіз планованих дій с точки зору їх прийнятності;
- функція оцінки;
- використання засобів програмування.

Серед моделей для навчання ігрового адаптивного агенту для ігри «Змійка» найбільш популярним підходом є використання нейронних мереж, які побудовані на основі згорткових нейронних шарів, наприклад, з інтеграцією торусної згортки. Ця модель здатна показувати кращий результат по загальній метриці власника змагань порівняно з іншими аналогами які використовують згорткові ШНМ [26].

При виборі політики ігрового штучного інтелекту існуючі реалізації використовують наступні підходи [26]: жадібний (greedy), жадібний зі схильністю використовувати ризик (risk averse greedy), прямий агент, прямий агент зі схильністю не ризикувати.

Поширеною практикою при розробці агентів є використання різноманітних евристик. Наприклад, провести гру власного ігрового агенту зі трьома іншими, що змусить власного агенту уникати зіткнень з іншими

агентами одночасно. Під час гри декількох сторонніх агентів, основний агент здатний взаємодіяти з іншими та вчитися з урахуванням їх поведінки. Також особливістю евристики являється переймання досвіду для головного агенту від других агентів з точки зору максимізації кількості набраних балів під час ігрової сесії.

Наприклад, якщо сторонні агенти мають схильність на ризик (у балансі між дослідженням та використанням) і здатні швидко максимізувати кількість балів шляхом швидкого збирання фруктів, головний агент також буде переймати отриманий досвід. Головним недоліком даної евристики являється використання значних комп’ютерних ресурсів для навчання головного агенту, так як переймання досвіду від інших агентів потребує значної кількості зіграних ігор [26]. Ще однією евристикою є використання алгоритму заливки (Flood fill) [26]. Це алгоритм, який враховує, наскільки далеко предмети чи гравці находяться від ігрового агенту.

При виборі варіантів майбутніх дій агенту, доцільним є прогнозування на декілька кроків уперед, з метою уникнення дій, які можуть призвести до погіршення загальної метрики під час виконання оптимізації та перенавчання. Щоб уникнути цього необхідно виконати фільтрацію небажаних кроків за допомогою алгоритму Monte-Carlo Search Tree [2].

У якості функції оцінки використана кількість балів, відповідно до числа зібраної їжі та часу присутності агенту на ігровому полі.

Більшість з розглянутих реалізацій використовують мову програмування Python для створення агентів адаптивного ігрового штучного інтелекту. Але реалізація Monte-Carlo Search Tree на мові програмування Python являється ускладненою, тому особливістю розглянутих рішень є реалізація цього алгоритму на мові програмування C++.

Розглянуті реалізації ігрового штучного інтелекту на основі таких підходів:

- на основі генетичних алгоритмів;
- на основі жадібної політики та згорткових ШНМ;

- на основі жадібної політики зі схильністю використовувати ризик та нейронної мережі на основі згорткових шарів;
- на основі жадібної політики та ШНМ на основі згорткових шарів у вигляді тору та використання Flood fill алгоритму.

Кожен із аналогів представлений як рішення до задачі, яка описана в змаганні на платформі Kaggle (таблиця 1.1).

Таблиця 1.1 – Порівняльна таблиця існуючих рішень по створенню штучного інтелекту для ігри «Змійка»

Назва рішення	Переваги	Недоліки
Ігровий штучний інтелект на основі генетичних алгоритмів	Простота реалізації. Ефективність роботи при невеликій кількості об’єктів	Невелика тривалість життя
Ігровий штучний інтелект на основі жадної політики та ШНМ на основі згорткових шарів	Агент здатен уникати колізій на протязі більшого часу, у порівнянні з аналогами. Алгоритм здатний збирати фрукти та не закінчувати гру на протязі 200 кроків	Великий час тренування моделі. Середня здатність збирання фруктів під час ігрової сесії порівняно із аналогами
Ігровий штучний інтелект на основі жадібної політики зі схильністю використовувати ризик та ШНМ на основі згорткових шарів	Ефективний алгоритм, що дозволяє швидко збирати фрукти, порівняно з аналогами. Це надає можливість мати високий рахунок на початку гри	Невелика тривалість життя агенту

Продовження таблиці 1.1

Ігровий штучний інтелект на основі жадібної політики та нейронної мережі на основі згорткових шарів у вигліді торуса та використання алгориму Flood fill	Агент здатний уникати колізій на протязі великого проміжку часу порівняно з аналогами. Алгоритм здатний збирати фрукти та не закінчувати гру на протязі 230 кроків Має більшу схильність до збору фруктів, ніж жадібний алгоритм	Значний час тренування моделі.

Найбільш поширеними недоліками аналогів є великий час тренування та дисбаланс між пошуком фруктів та часом життя «Змійки». Щоб зменшити дисбаланс вибору рішення, доречно використання ансамбль моделей: штучний інтелект на основі жадної політики та згорткової штучної нейронної мережі та моделі на основі жадної політики та штучної нейронної мережі на основі згорткових шарів у вигляді торуса та використання алгоритму Flood fill.

1.4 Постановка задачі

Метою та завданням кваліфікаційної роботи є розробка демонстраційної комп’ютерної програми, яка містить в собі гру «Змійка» та агентів, побудованих на основі динамічного ігрового штучного інтелекту використовуючи навчання з підкріпленням та глибині нейронні мережі. Агент повинен вчитися на основі попереднього досвіду і демонструвати результат під час гри с користувачем.

Досягнення поставленої мети можливе завдяки вирішенню наступних задач:

Аналіз предметної області. Потрібно дослідити базові принципи побудови ігор та створення для них штучного інтелекту. Також необхідно дослідити сучасні алгоритми динамічного навчання агентів. Це можливо виконати за допомогою Deep Q-Learning моделей та Monte-Carlo Search Tree. Потрібно виконати дослідження використання функції по пеналізацізації на основі видалення нейронів за допомогою гаусового підходу [20].

Створення ігрового середовища для ігри «Змійка». Середовище повинне мати поле, яке може містити до 5 агентів одночасно, виключаючи головного гравця. Ігрове середовище повинне повністю повторювати правила, які використовуються в традиційній ігри. Середовище повинне бути використане з платформи Kaggle [8].

Агент повинен використати підхід до адаптації, побудований на основі навчання з підкріпленням. Агент повинен використовувати Deep Q-Learning та Monte-Carlo search Tree [17].

Виконати тренування ігрового штучного інтелекту та налаштування кількості та видів нейронів для тренування. Штучний інтелект повинен виконувати прогнозування дій в реальному часі. Кількість шарів нейронної мережі та їх види повинні бути задані самостійно.

Вибір найефективнішого агенту серед всіх можливих на основі функції винагороди.

Інтеграція штучного інтелекту в ігрове середовище. Середовище повинне мати підтримку роботи багатьох агентів одночасно і слідувати за правилами ігри «Змійка».

Демонстрація роботи багатьох агентів під час однієї ігрової сесії. Результатом роботи повинен стати прототип гри з ігровим середовищем та багатьма агентами, які можуть грати одночасно в полі з другими агентами. Агенти повинні не закінчувати гру в короткий час, а забезпечити можливість того, щоб одна партія продовжувалась не менше 30 секунд.

2 ТЕХНОЛОГІЇ ТА МЕТОДИ ВИРІШЕННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

2.1 Опис методології CRISP-DM

Якість виконання проекту залежить від методології обраного процесу. При розробці ігрового агенту використовується методологію CRISP-DM. CRISP-DM [27] – це міжгалузевий процес аналізу даних. Методика CRISP-DM забезпечує структурований підхід до планування проекту. Переваги цієї методики в представленні чіткої послідовності дій, необхідних для створення успішного проекту.

Методологія CRISP-DM описується з точки зору ієрархічної моделі процесу, що включає чотири рівні абстракції (від загального до конкретного): фази (phases), загальні завдання (generic tasks), спеціалізовані завдання (specialized tasks) та екземпляри процесу (process instances).

На найвищому рівні процес передачі даних організований у невелику кількість фаз. Кожна фаза складається з декількох загальних завдань другого рівня. Загальні завдання необхідно створювати таким чином, щоб вони були завершеними і стабільними. Завершеними завданнями вважаються завдання, які покривають процес аналізу даних та всі можливі дані. Стабільними вважаються завданнями, при яких модель буде валідною при інтеграції нових видів моделей в ансамбль.

Життєвий цикл проекту складається з таких компонентів (рисунок 2.1):

- дослідження бізнес-області замовника;
- дослідження наявних даних;
- підготовка даних;
- створення моделей машинного навчання;
- оцінка результатів моделі на основі кросс-валідації та тестового датасету;
- публікація створеного алгоритму до програмного середовища.

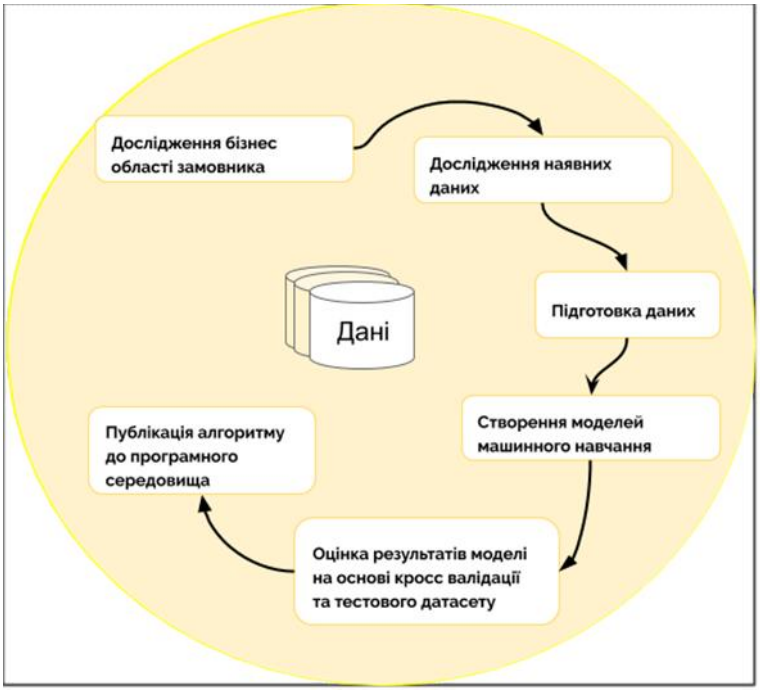

Рисунок 2.1 – Життєвий цикл розробки проекту при використанні методології CRISP-DM

Розглянуто детальний опис кожного кроку в життєвому циклі розробки при використанні методології CRISP-DM (рисунок 2.1).

Дослідження бізнес-області замовника. Це початковий етап, що спрямований на розуміння цілей та вимог проекту з точки зору бізнесу і використання цих знань у визначення проблем аналізу даних та створення відповідного плану до проекту.

Дослідження наявних даних. Етап дослідження наявних даних починається зі збору даних та супроводжується такими процесами, як ознайомлення з даними, виявлення проблем з якістю даних, виконання початкового аналізу даних або виявлення фактів для формування гіпотез щодо прихованої інформації. Для формулювання проблеми аналізу даних та плану проекту необхідно обов’язково проводити дослідження наявних даних.

Підготовка даних. Етап підготовки даних охоплює створення кінцевого набору даних (дані, які будуть передані для тренування моделей, ансамблей, проведення оцінки ефективності моделей на основі кросс-валідації,

проведення оцінки ефективності яка показує контрольний результат ефективності моделі) з необроблених даних. Завдання з підготовки даних, виконується декілька разів, але не обов’язково в установленому порядку.

Завданнями підготовки даних є:

- вибір датасетів (наборів даних) для обробки даних;
- трансформація табличних значень;
- очищення даних від порожніх рядків, колонок, аномалій;
- створення нових атрибутів на основі бізнес аналізу області використання даних замовника;
- трансформація даних для моделей, які будуть використовуватися. Для лінійних моделей та нейронних мереж необхідно конвертувати дані за допомогою нормалізації;
- створення моделей машинного навчання;
- оцінка результатів моделі на основі кросс-валідації та тестового датасету;
- публікація створеного алгоритму до програмного середовища;
- отримані результати потрібно буде організувати та представити таким чином, щоб замовник міг ним скористатися.

2.2 Reinforcement learning

Reinforcement learning (RL) – це область машинного навчання, яка визначає, як агенти повинні сприймати дії в середовищі, щоб максимально розширити уявлення про сукупну винагороду [5].

Reinforcement learning передбачає навчання без вчителя з діями, щоб максимізувати винагороду. Це є проблемою замкнутого циклу, оскільки система навчання впливає на наступні результати. Агенту не повідомляється, які дії він має вчинити, як у поширених формах машинного навчання. Агент повинен виконати дослідження, які дії приносять найбільшу винагороду, випробовуючи їх. В складних випадках дії можуть вплинути не тільки на

винагороду, а й на наступну ситуацію і, відповідно, на всі наступні винагороди. Агент немає прямих вказівок щодо того, які дії слід вжити, і які наслідки цих дій, включаючи розмір винагороди, відтвореного протягом тривалого періоду часу.

Одна з проблем, що виникає у процесі reinforcement learning – це компроміс між дослідженням та експлуатацією [6]. Щоб отримати велику винагороду, підкріплюючий навчальний агент повинен віддавати перевагу діям, які він пробував в минулому і виявив свою ефективність в отриманні винагороди. Але щоб виявити такі дії, він повинен спробувати дії, які він не вибрав раніше. Агент повинен використовувати те, що він уже знає, щоб отримати винагороду, але він також має досліджувати, щоб краще вибирати дії на майбутнє. Дилема полягає в тому, що ні розвідки, ні експлуатації не може бути виконано виключно без помилок у виконанні завдання. Агент повинен спробувати різні дії і поступово віддавати перевагу тим, які здаються найкращими. В стохастичному завданні, кожну дію потрібно виконати багато разів, щоб отримати достовірну оцінку. Дилема «розвідка-експлуатація» вивчалася математиками протягом багатьох десятиліть. Проблема балансування розвідки та експлуатації не виникає в supervise і unsupervised навчанні [6].

Ще однією ключовою особливістю навчання з підкріпленням є те, що враховується вся проблема взаємодії цільового агенту з невизначеним середовищем. Розроблені теорії планування з загальної цілі, але без урахування ролі планування у прийнятті рішень у реальному часі, або питання про те, у яких випадках прогнозувальні моделі необхідні для планування. Хоча ці підходи дали багато корисних результатів, їх зосередження на окремих підпроблемах є суттєвим обмеженням [10].

Навчання з підкріпленням пропонує адаптивний агент. Усі агенти навчання з підкріпленням мають явні цілі, які можуть використовувати аспекти свого оточення та обирати дії, щоб впливати на своє оточення. Агент повинен діяти, незважаючи на значну невизначеність середовища, з якою він

стикається. Коли навчання з підкріпленням передбачає планування, воно має бути спрямоване на взаємодію між плануванням і вибором дій у реальному часі та врахуванням як вдосконалюються модель середовища. Коли навчання з підкріпленням передбачає supervised learning (навчання з вчителем), це відбувається з визначенням того, які можливості є критичними.

Навчання з підкріпленням також є частиною сучасної тенденції у штучному інтелекту до використання простих загальних принципів [12]. Слід прийняти, що інтелект натомість пояснюється володінням величезною кількістю процедур та евристик спеціального призначення. Методи, засновані на загальних принципах, такі як пошук або навчання, характеризувалися «слабкими методами», тоді як ті, що базуються на конкретних знаннях, називалися «сильними методами». Сучасний AI тепер включає багато досліджень, які шукають загальні принципи навчання, пошуку та прийняття рішень, а також намагаються включити величезну кількість знань у цій галузі.

2.3 Q-learning

Q-learning – це алгоритм навчання з model-free підкріплення. «Q» у Q-learning означає якість, яка показує, наскільки корисна дана дія для отримання певної майбутньої винагороди.

Q-learning є алгоритмом value-based навчання [14]. Алгоритми на value-based оновлюють функцію значення на основі рівняння (зокрема рівняння Беллмана). Алгоритми policy-based оцінюють значення функції за допомогою жадібної політики, отриманої з останнього оновлення політики.

Q-learning – це off-policy алгоритм. Це означає, що він дізнається значення оптимальної політики незалежно від дій агенту. З іншого боку, on-policy learner дізнається значення політики, яку виконує агент, включаючи етапи дослідження, і він знайде політику, яка є оптимальною, враховуючи дослідження, притаманне цій політиці [15, 15, 22].

Q*(s, a) – очікувана вартість (сукупна дисконтована винагорода) виконання дії a в стані s і подальшого дотримання оптимальної політики.

Q-learning Temporal Differences (TD) для оцінки значення Q*(s, a). Temporal Differences визначає, що агент навчається з середовища через епізоди без попереднього знання середовища.

Q-функція використовує рівняння Беллмана і приймає два вхідні характеристики: стан (s) і дію (a) (рисунок 2.2).

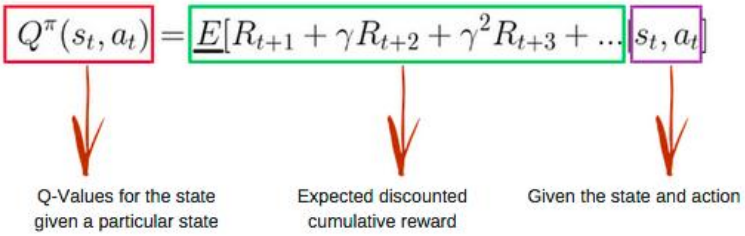

Рисунок 2.2 – Рівняння Белмана

Алгоритм Q-learning у загальному випадку містить наступні етапи (рисунок 2.3).

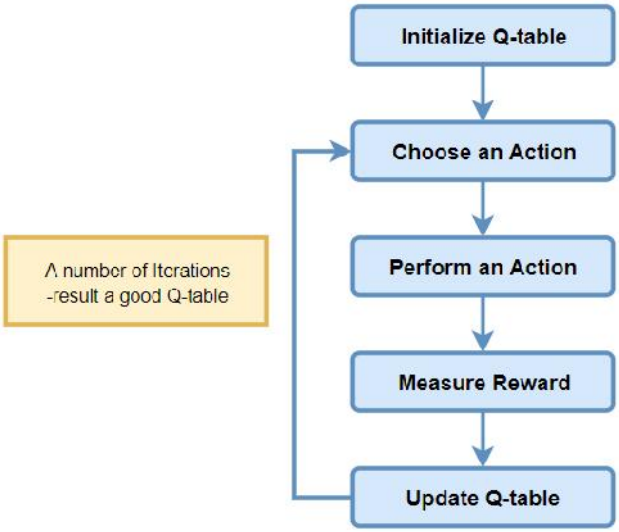

Рисунок 2.3 – Алгоритм роботи Q-Learning

Перший крок полягає у побудові Q-таблиці. Існує n колонок, де n дорівнює кількості дій. Існує m рядків, де m – кількість станів (рисунок 2.4).

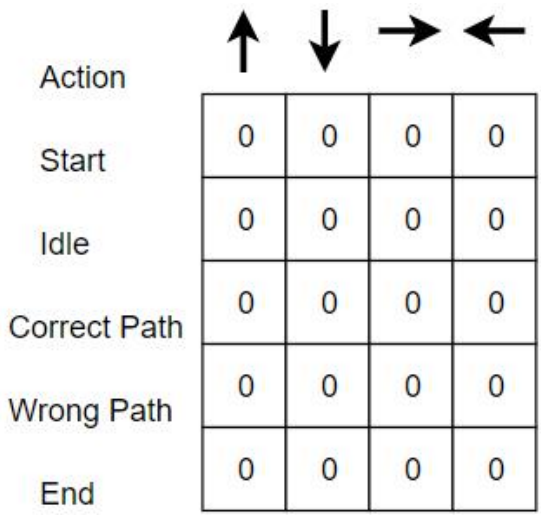

Рисунок 2.4 – Q-table, де колонка – це дія, рядок – це стан

Агент підтримує таблицю Q[S, A], де S – набір станів, а A – набір дій.

Q[s, a] представляє його поточну оцінку Q*(s,a).

Q-Table – це структура даних, яка використовується для обчислення максимальної очікуваної винагороди за дію в кожному стані. Таким чином, ця таблиця направляє агента до найкращих дій при кожному кроці. Для вивчення кожного значення Q-таблиці використовується алгоритм Q-Learning.

Другим кроком є вибір дії. Спочатку на основі Q-таблиці вибирається дія (a) у стані (s). Коли епізод починається спочатку, кожне значення Q має бути 0. Потім оновлюється Q-значення, використовуючи рівняння Беллмана.

Для використання жадібної стратегії пошуку використовуються алгоритм Epsilon. На початку значення Epsilon будуть високими. Агент досліджуватиме навколишнє середовище і випадковим чином вибирає дії, оскільки агент нічого не знає про середовище. У міру того, як агент досліджує середовище, швидкість алгоритму Epsilon зменшується, і агент починає експлуатувати середовище [15, 16].

Четвертим кроком є вимір винагороди.

П’ятим кроком є обновлення значень для функції $Q(s,a)$.

Цей процес повторюється знову і знову, поки навчання не буде завершено.

Таким чином Q-Таблиця оновлюється, а функція Q максимізується. Тут Q(стан, дія) повертає очікувану майбутню винагороду за цю дію в цьому стані (рисунок 2.5).

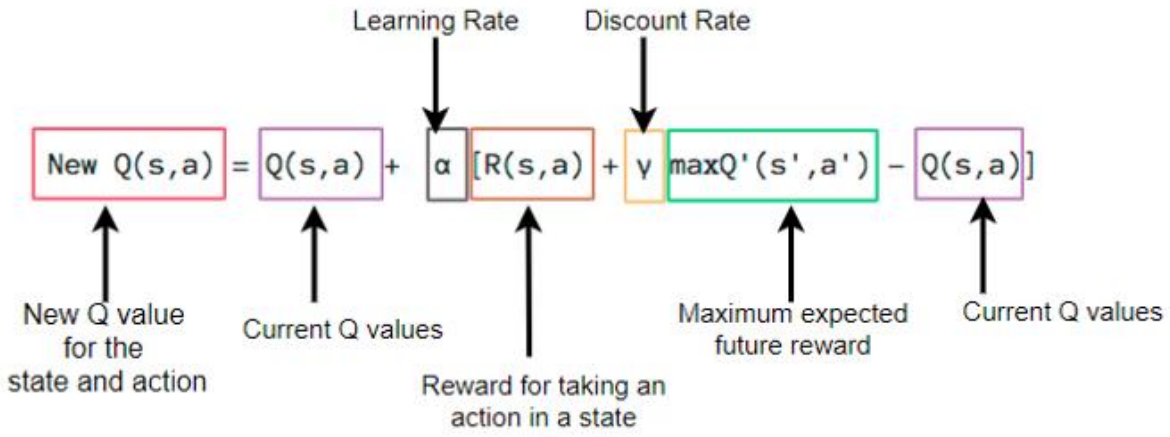

Рисунок 2.5 – Оновлення Q-таблиці

2.4 Deep Q-learning

В Deep Q-learning [14] використовується нейронна мережа для наближення функції Q-значення. Стан подається як вхід, а Q-значення всіх можливих дій генерується як вихід. Етапи навчання з підкріпленням за допомогою мереж глибокого Q-навчання (DQN) наступні.

Весь минулий досвід зберігається користувачем в пам’яті.

Наступна дія визначається максимальним виходом Deep Q-learning network. Функція втрат тут є середньоквадратичною помилкою прогнозованого Q-значення та цільового Q-значення – Q*. Наведено порівняння між Q-learning та Deep Q-learning (рисунок 2.6).

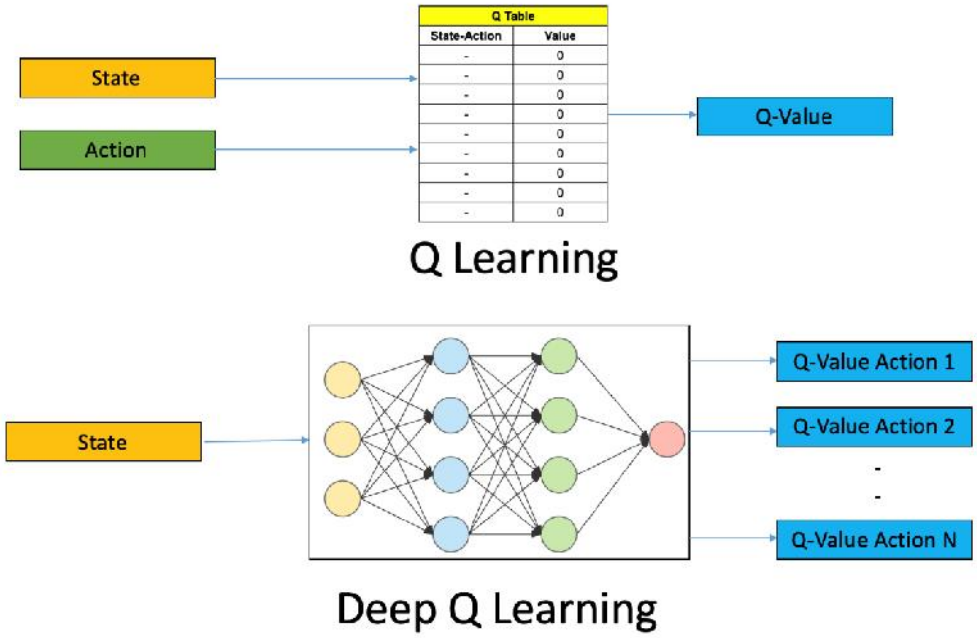

Рисунок 2.6 – Порівняння принципів роботи Q-Learning та Deep Q-Learning

Для розрахунку функції витрат використаємо рівняння оновлення значення Q, отриманого з рівняння Беллмана (2.1):

$$Q(S_t, A_t) = Q(S_t, A_t) + a[R_{t+1} + y \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (2.1)$$

Функція прогнозує власне значення, але оскільки R є винагородою, мережа оновлює свій градієнт, використовуючи backprogation, щоб наблизитися до прогнозованого результату.

2.5 PyTorch

Фреймворки глибокого навчання часто зосереджені на зручності використання або швидкості. PyTorch [8] – це бібліотека машинного навчання, яка забезпечує імперативний і Pythonic стиль програмування.

Бібліотека підтримує принцип «код як модель», налагодження та узгоджується з іншими популярними обчислювальними бібліотеками,

залишаючись ефективним засобом. Підтримує апаратні прискорювачі, такі як графічні процесори. Pytorch використовує автоматичне диференціювання, що дозволило повністю автоматизувати обчислювання похідних. Це значно полегшує дослідження різних підходів машинного навчання, при цьому дозволяє виконувати ефективну оптимізацію на основі градієнта. Пакет autograd популяризував використання цієї техніки для масивів NumPy [11].

З появою вільного програмного забезпечення наукова спільнота відійшла від закритого програмного забезпечення, таке як Matlab і перейшла на екосистему Python з відкритим вихідним кодом з такими пакетами, як NumPy, SciPy та Pandas. Це дозволило користуватися перевагами величезного сховища бібліотек, виконувати обробку великого набору даних, статистичний аналіз, будувати графіки, тощо. Відкритість, сумісність та гнучкість вільного програмного забезпечення сприяли розвитку спільнот, які може швидко задовольнити нові потреби, розширивши існуючу функціональність бібліотеки (рисунок 2.7).

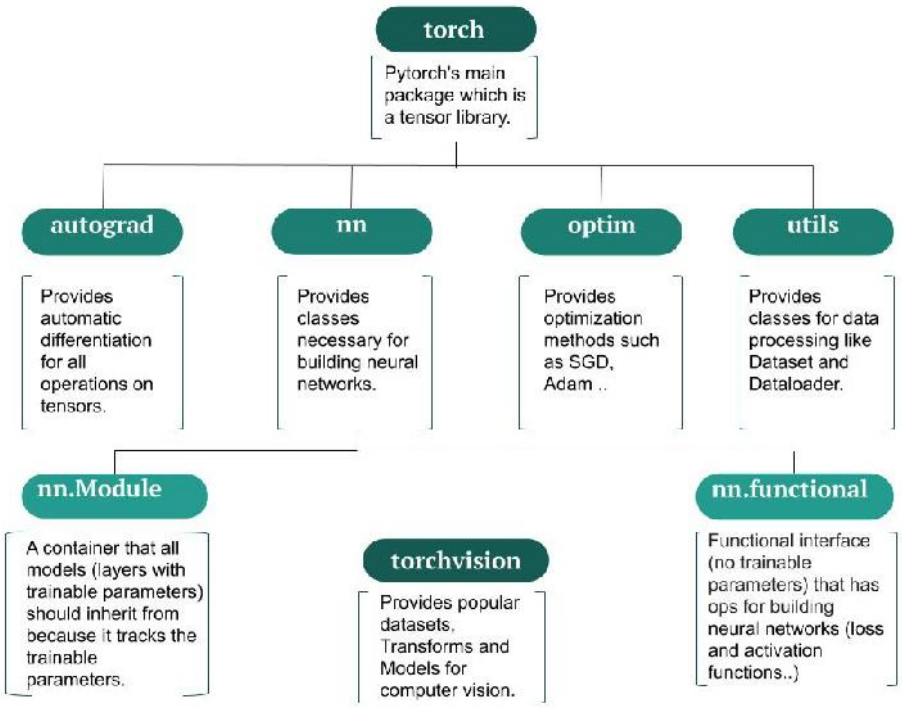

Рисунок 2.7 – Компоненти бібліотеки PyTorch

2.6 Monte-Carlo tree search

MCTS [18, 20] використовує симуляцію Монте-Карло для накопичення оцінок вартості, щоб направляти агенту до ефективних траєкторій у дереві пошуку. MCTS складається з повторюваних ітерацій (обмежених обчисленням часу та ресурсів) і складається з 4 кроків: вибір, розширення, моделювання та резервне копіювання [18].

При виконанні вибору використовується tree policy для побудови шляху від кореня до найбільш перспективного вузла листя (рисунок 2.8).

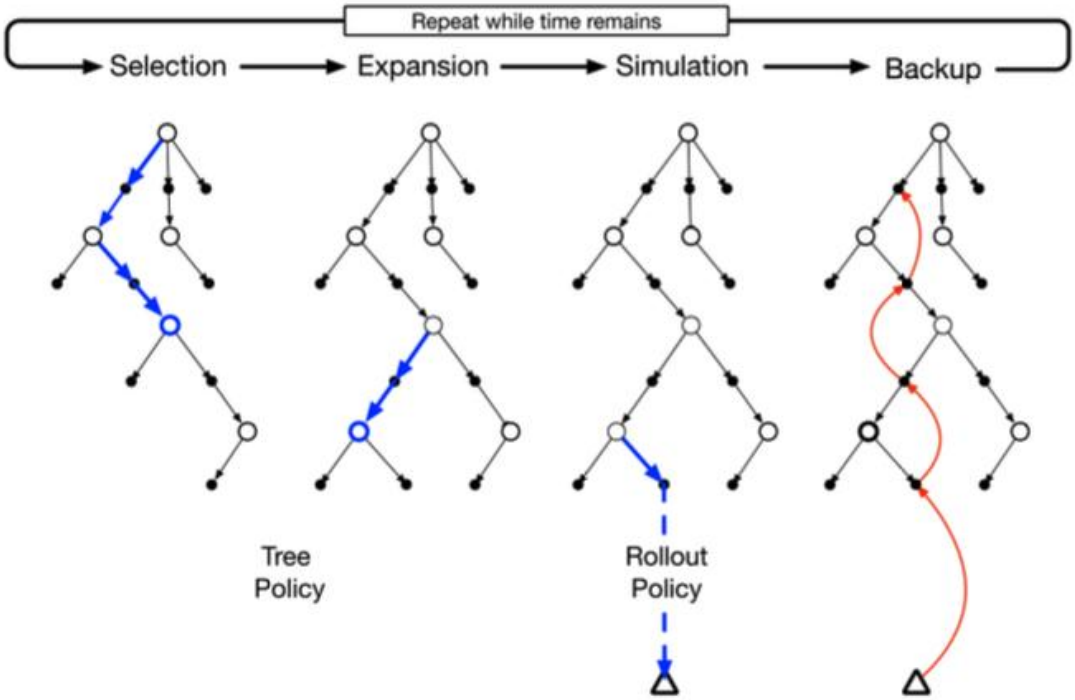

Рисунок 2.8 – Принцип роботи алгоритму Monte-Carlo tree search

Tree policy – це політика, яка використовується для вибору дії (вузла) у вершині (досліджена частина ігрового дерева на відміну від величезної невивченої нижньої частини). Одним з важливих особливостей використання такої політики є баланс дослідження та експлуатації (exploration vs exploitation). Така політика є досить популярною в Reinforcement learning.

Наприклад, в алгоритмі AlphaGo використовується політика на основі Upper Confidence Bound (UCB) [18]. Кожен вузол має пов’язане значення UCB, і під час вибору завжди вибирається дочірній вузол з найвищим значенням UCB (2.2).

$$USB(node_i) = x_i + c\sqrt{logN/n_i} \tag{2.2}$$

де x_i – середнє значення вузла, n_i – кількість візитів до ноди, N – кількість візитів до попереднього вузла. Чим більше відвідується вузол i, тим меншим стає друга частина в UCB, отже, зменшується його ймовірність повторного вибору.

Тому алгоритм UCB має вбудовану функцію дослідження та експлуатації. Алгоритм буде верхню досліджувати частину дерева, поки не буде досягнутий листовий вузол.

Наступним кроком є розширення. На етапі розширення випадковим чином вибирається недосліджений вузол листового вузла.

Наступним кроком є симуляція. Під час цього кроку розгортається одна або декілька моделей із винагородою, накопиченою за кожну симуляцію. Політика розгортання зазвичай є простою, щоб її можна було швидко виконувати. Наприклад, перемога може призвести до винагороди +1, нічия – 0 і програш – 1.

Наступним кроком є резервне копіювання. Використовуються накопичені нагороди моделювання для резервного копіювання та оновлення значень вузлів у верхній частині дерева. Значення вузлів не обновлюються під час кроків розгортання. Основною причиною виконання переднього кроку є зосередження на верхніх кореневих вузла, на основі яких приймається рішення про наступний крок. Тоді як значення за межами верхньої частини дерева не мають значення для такого рішення.

2.7 Stacking

Stacking – це процедура, спрямована на покращення прогнозування шляхом змішування або комбінування прогнозів декількох моделей машинного навчання. Існує безліч методів для комбінування, від простих, таких як voting, або находження середнього значення прогнозів, до побудови складних моделей навчання (логістичні регресії, градіїєтний бустинг).

Stacking прогнозів моделей машинного навчання дуже часто допомагає отримати кращі результати і широко використовується для перемоги в змаганнях з Kaggle [19].

2.8 HandyRL

HandyRL [3] – це зручний і простий фреймворк, заснований на Python і PyTorch для розподіленого навчання з підкріпленням, який можна застосувати до власного середовища. Для широкомасштабного навчання HandyRL забезпечує підтримку паралельності відповідно до середовища.

HandyRL в основному надає алгоритм «Policy gradient algorithm with off-policy correction» та забезпечує можливість використовувати деякі off-policy варіанти методів оновлення від традиційних (Монте-Карло, TD(λ)) до нових (V-Trace, UPGO). HandyRL використовує архітектуру стилю learner-worker, як IMPALA [12]. Учень – це головний елемент навчання, який оновлює модель та контролює потоки. Потоки виконують дві ролі. Вони асинхронно генерують епізоди (траєкторії) та оцінюють навчені моделі. При генерації епізодів самовідтворення здійснюється за замовчуванням.

2.9 Кросс-валідація

Кросс-валідація (cross-validation, CV) – процедура емпіричного оцінювання узагальнюючої здатності алгоритмів, яких навчають по

прецедентах [28]. Фіксується розбиття вихідної вибірки на дві підвибірки: навчальну і контрольну. Для кожного розбиття виконується налаштування алгоритму за навчальною підвибіркою, потім оцінюється його середня помилка на об’єктах контрольної підвибірки. Оцінкою кросс-валідації називається середня по всім розбиттям величина помилки на контрольних підвибірках.

Кросс-валідація є стандартною методикою тестування і порівняння алгоритмів класифікації, регресії і прогнозування. Можуть бути різні варіанти змінного контролю, що відрізняються способами розбиття вибірки (таблиця 2.2).

Таблиця 2.2 – Описання методів кросс-валідації

Назва	Описання
Hold-out CV	Оцінка крос-валідації будується по одному випадковому розбиттю.
Leave One Out (LOO)	Leave One Out (або LOO) – працює наступним чином. Кожна вибірка створюється шляхом взяття всіх крім одного. Цей один елемент з вибірки буде тестовим елементом. Таким чином, маємо різні навчальні набори та різні набори тестових вибірок.
K-Fold	Fold ділить усі вибірки на підвибірки, які називаються фолдами рівних розмірів (якщо можливо). Функцію прогнозування вивчають за допомогою фолдів всіх фолдів крім одного, а відкинутий фолд використовують для тестування.

Якщо вибірка незалежна, то середня помилка кросс-валідації дає не зміщену оцінку ймовірності помилки. Це вигідно відрізняє її від середньої помилки на навчальній вибірці, яка може виявитися зміщеною (оптимістично заниженою) оцінкою ймовірності помилки, що пов’язано з явищем перенавчання.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Загальна структура проекту

Основним завданням проєкта є реалізація методів для забезпечення адаптивності ігрового агенту для ігри «Змійка». Щоб виконати це завдання потрібно слідувати за наступним алгоритмом:

- створення адаптивний ігровий агент на основі ШНМ;
- реалізація тестового середовища для гри «Змійка»;
- оцінка результатів, що отримані при тренуванні ігрового агента;
- аналіз роботи створеного агенту та конкурентів на тестовому ігровому середовищу.

Кожен етап можливо додатково розділити на допоміжні етапи і з врахуванням цього створено архітектуру файлів вихідного коду проекту (рисунок 3.1). Дана архітектура слідує методології CRISP-DM [27].

Створення ігрового агенту потребує виконання комплексних кроків по реалізації попередньої обробки даних, використання нейронної мережі, та інтерпретації результатів для користувачів. Для ефективної реалізації адаптивного ігрового агенту на основі штучної нейронної мережі виконано наступні кроки:

- побудова архітектури ШНМ;
- створення методів препроцесингу даних перед поданням їх для тренування нейроної мережі;
- налаштування ШНМ;
- налаштування конфігурації використання Monte-Carlo Search Tree для фільтрації малоефективних кроків алгоритму;
- реалізація базових методів візуалізації ефективності роботи нейронної мережі;
- оцінка ефективності роботи застосунку;

- реалізація можливості зберігання налаштувань моделі при виконанні кожних 100 епох в ШНМ в папці models, щоб відновити роботу тренування моделі, якщо память відеокарти закінчиться.

Після успішного виконання тренування результати агенту зберігаються папці game records.

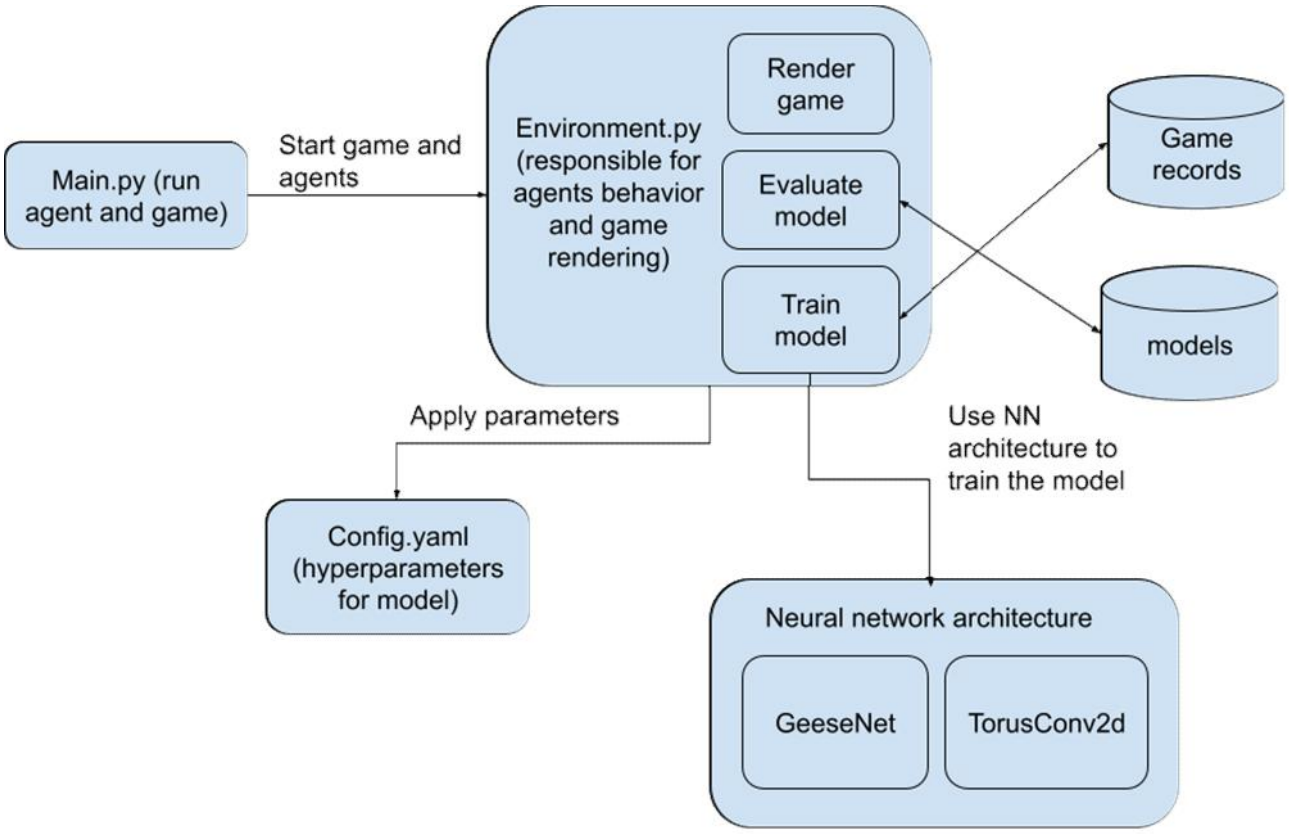

Рисунок 3.1 – Функція запуску ігрового середовища та агентів

Щоб запустити агент потрібно створити для нього середовище виконання. Середовище представлене бібліотекою kaggle_envinroments, яка спрощує створення базових агентів та ігрового поля для клієнтів.

Бібліотека kaggle_envinroments має колекцію вже реалізованих різних агентів для ігри в «Змійку» з простим статичним штучним інтелектом (рисунок 3.2). Наприклад, агент GreedyAgent використовує алгоритм находження найкоротшого шляху до фруктів, але є слабким до взаємодії з другими змійками.

Важливим компонентом бібліотеки є модуль відображення ігри, що надає простий API по створенню ігрового поля та заданню базових налаштувань, таких як розмір ігрового поля. Також компонент дозволяє оновлювати координати агентів (змійок) в залежності від результатів, що отримані від нейронної мережі створеного агенту.

Особливістю бібліотеки є процес генерування фруктів на ігровому полі. На кожні 2 агенту на ігровому полі бібліотека kaggle створює один фрукт. Якщо фрукт був підібраний однією з присутніх змійок, то елемент створить новий фрукт в довільному місці на ігровому полі.

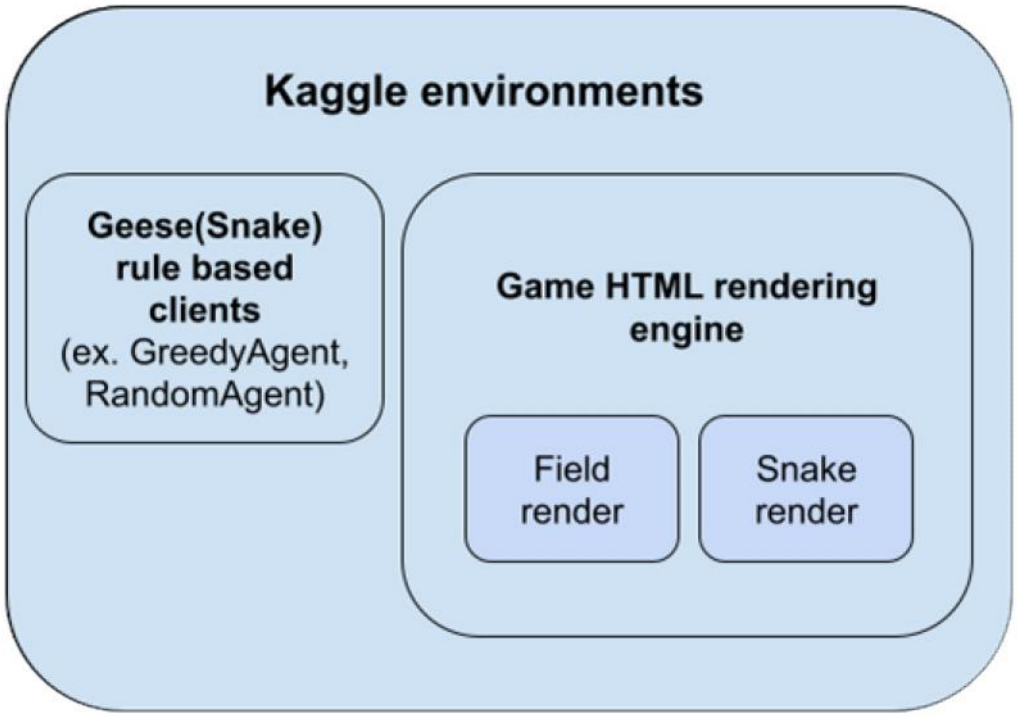

Рисунок 3.2 – Архітектура бібліотеки kaggle_enviroments

Для оцінки результатів роботи агенту проведено сумісні ігри з іншими агентами. Результати ігор з кожним з агентів було збережено в папку battles. Всі сумісні ігри виконуються на платформі kaggle так як сама платформа надає можливість ігри з агентами у вигляді зручного вікна (рисунок 3.3).

Рисунок 3.3 – Відображення ігрового процесу

Програма створена в середовищі програмування PyCharm і використовуючи мову програмування Python для реалізації логіки. Програма складається із наступник файлів :

- neural_net.py;
- torus_net.py;
- environment.py;
- main.py;
- config.yaml.

Файл neural_net.py представляє собою ШНМ, що виконує розрахунок коефіцієнтів для прийняття рішень агенту.

Файл torus_net.py відповідає за шар нейронної мережі яка складається із згорткового шару з модулем трансформації фільтра у вигляді торуса. Це основний компонент для нейронної мережі на основі якого виконується навчання та прогнозування агенту.

Файл environment.py відповідає за середовище, в якому працюють агенти. Дане середовище побудоване на основі бібліотеки HandyRL і основним завданням середовища є кодування логіки поведінки агентів, визначення кількості гравців на ігровому полі, використання нейронної мережі для виконання та прийняття дії агенту.

Файл main.py відповідальний за виконання застосунку агентів і середовища (рисунок 3.1), використовує клас від файлу environment.py для відображення ігрових агентів на одному полі і виконання агентів на ігровому полі.

Файл config.yaml – це конфігураційний файл для бібліотеки HandyRL. Дана бібліотека надає можливість конфігурації необхідних параметрів для навчання з підкріпленням. Наприклад, вона має параметри для налаштування розміру пакету для навчання чи вибору методу для оптимізації функції політики, наприклад, такої як MC.

Також проект має папки game_records, models та battles, які слугують як додаткові компоненти програми для зручності навчання та використання моделей.

Папка game_records зберігає файли, що пов’язані з штучною нейронною мережею та її вагами. Це необхідно для встановлення стану ШНМ під час тренування, якщо станеться непередбачувана ситуація, оскільки навчання ШНМ потребує значних обчислювальних ресурсів. Кожен із файлів в папці game_records має розширення .pickle та призначений для збереження статусу виконання Python-коду. Перевагою Pickle-файлів є швидкість зберігання та зчитування статусу виконання штучною нейронною мережею.

В папці Models зберігається усі можливі та створені моделі ШНМ для ігрового агенту. Програма використовує цю папку для зчитування моделі з пам’яті і прогнозування результату для кінцевого користувача. Також вона має в собі і моделі других ігрових агентів. Це необхідно, щоб проводити локально симуляції з іншими агентами та демонстрації порівняльної характеристики між ними.

Папка battles складається з ноутбуків (файлів з розширенням .ipynb), які слугують місцем для експериментів між агентами. Тут проводилися експерименти по порівнянню ефективності роботи створеного агенту в ході кваліфікаційної роботи та інших ігрових агентів.

3.2 Модуль нейронної мережі

Розглянемо модуль створення нейронної мережі для виконання дій адаптивного агенту. Ця нейронна мережа написана на мові програмування Python, яка використовує концепцію Q-learning і Deep Q Learning. Основною характеристикою нейронної мережі є розрахунок функції значення та функції політики. Функція політики (policy) – це розподіл ймовірності для кожного стану на всі можливі дії. Функція цінності (value) дозволяє оцінити якість різних політик виданих нейронною мережею.

Агент представляє собою нейронну мережу побудовану на основі декількох основних компонентів. Розглянемо один с компонентів нейронної мережі – це торусна згортка (лістинг 3.1).

Лістинг 3.1 – Клас торусної згортки

```
class TorusConv2d(nn.Module):

      def __init__(self, input_dim, output_dim, kernel_size, bn):
        super().__init__()

        self.edge_size = (kernel_size[0] // 2, kernel_size[1] // 2)
        self.conv = nn.Conv2d(input_dim, output_dim, kernel_size=kernel_size)
        self.bn = nn.BatchNorm2d(output_dim) if bn else None

    def forward(self, x):
        h =
torch.cat([x[:,:,:,-self.edge_size[1]:], x,
x[:,:,:,:self.edge_size[1]]], dim=3)
        h = torch.cat([h[:,:,-self.edge_size[0]:], h,
h[:,:,:self.edge_size[0]]], dim=2)
        h = self.conv(h)
        h = self.bn(h)
if self.bn is not None
else h
        return h
```

Клас наслідує клас nn.Module, що собою представляє оболонку з бібліотеки для будування нейронних мереж PyTorch.

Компонент TorusConv2d здатний інкапсолювати декілька шарів нейронної мережі і надати простий інтерфейс для створення загальної нейронної мережі.

Функція __init__ відповідальна за базову підготовку компоненту торусної згортки. Вона складається із наступних компонентів: двухрозмірної згортки та шару нормалізації.

Параметрами функції налаштовується кількість вхідних нейронів, кількість вихідних нейронів та розмір фільтра, що використаний згорткою для формування згорткового шару. Наступним шаром являється шар нормалізації, кількість вихідних нейронів дорівнює кількісті вихідних нейронів для згортки.

Нормалізація застосовує перетворення, яке змінює середню вихідну величину до 0, а вихідне стандартне відхилення – близько до 1. Це необхідно для швидкого навчання нейронної мережі та уникнення проблем, які можуть виникнути під час тренування.

Функція __forward__ відповідальна за тренування шарів нейронної мережі (рисунок 3.4).

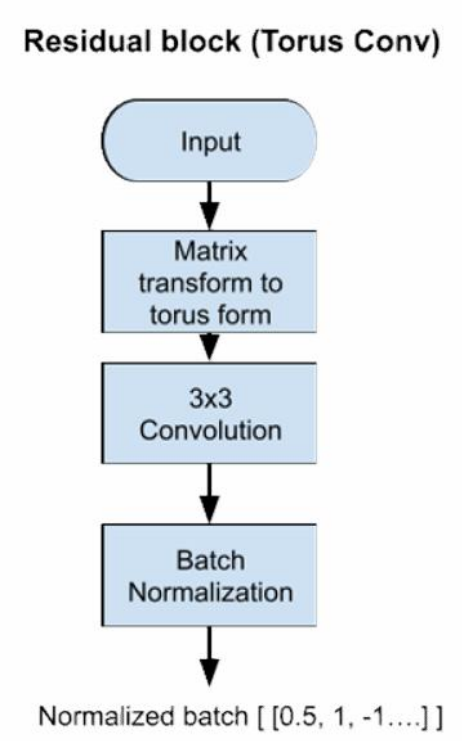

Рисунок 3.4 – Візуальне представлення торусної згортки

Розглянемо клас GeeseNet (лістинг 3.2), що представляє собою повноцінну нейронну мережу, яка складається із вхідного шару побудованих на основі торусної згортки з фільтром 3 на 3 та 12 внутрішніх блоків нейронної мережі на основі торусної згортки з фільтром 3 на 3. Виходом для нейронної мережі є значення, що необхідні для розрахунку дії агенту штучного інтелекту. Функцією активації для згорткових шарів та виходу політики є linear. Функцією активації для розрахунку функції цінності являється тангенс.

Лістинг 3.2 – Клас нейроної мережі ігрового агента

```
class GeeseNet(nn.Module):
    def __init__(self):
        super().__init__()
        layers, filters = 12, 32

        self.conv0 = TorusConv2d(17, filters, (3, 3), True)
        self.blocks = nn.ModuleList([TorusConv2d(filters, filters, (3, 3),
True) for _ in range(layers)])
        self.head_p = nn.Linear(filters, 4, bias=False)
        self.head_v = nn.Linear(filters * 2, 1, bias=False)

    def forward(self, x, _=None):
        h = F.relu_(self.conv0(x))
        for block in self.blocks:
            h = F.relu_(h + block(h))
        h_head = (h * x[:,:1]).view(h.size(0), h.size(1), -1).sum(-1)
        h_   v}
```

Особливість архітектури нейронної мережі (рисунок 3.6) полягає у тому, що шари отримання політики та значення мають комплексне отримання особливостей з шарів остатків (residuals). Вони складаються з слою head extraction і head expansion. Таким чином, цей модуль перетворює кожну клітинку, на якій знаходиться ігровий агент, в голову агенту, що дозволило нейронній мережі рахувати взаємодію між різними агентами і зменшити шанси зіткнення із іншими агентами (змійками) на ігровому полі (рисунок 3.7).

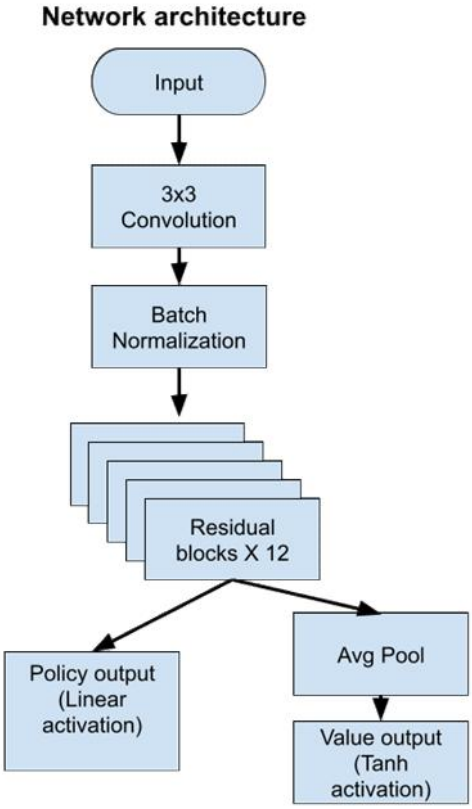

Рисунок 3.6 – Візуальне представлення повноцінної нейронної мережі для ігрового агенту

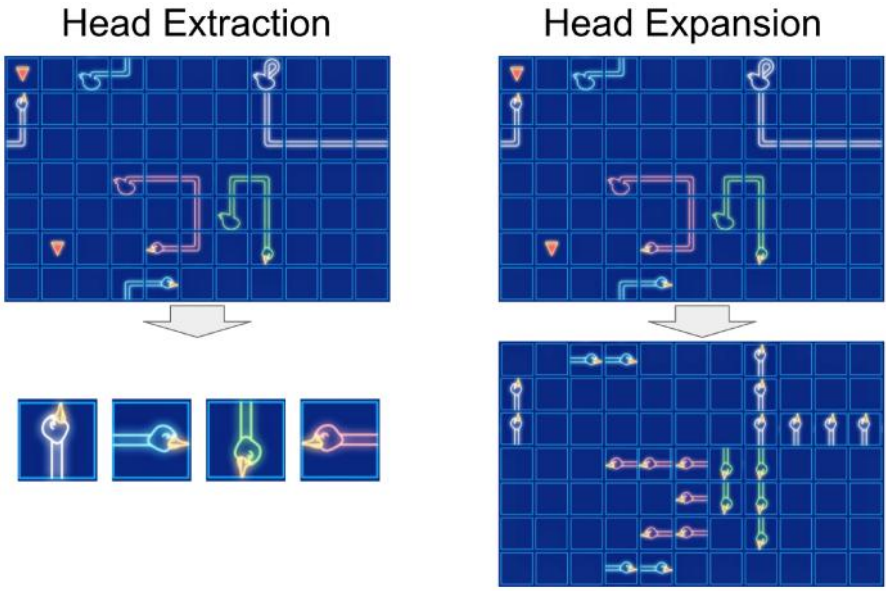

Рисунок 3.7 – Візуальне представлення head extraction і head expansion

Для налаштування параметрів нейронної мережі використовуються конфігураційний файл config.yaml (лістинг 3.3). Головними параметрами є розрахунок функції політики та функції значення за допомогою Monte Carlo Search Tree і це указано в параметрах policy_target та value_target. Параметр batch_size означає розмір пакету, тобто кількість спостережень, які будуть використані під час тренування протягом однієї епохи. Gamma і lambda параметри вказуються на коефіцієнт регуляції L1 та L2 відповідно.

Лістинг 3.3 – Налаштування тренування нейронної мережі

```
train_args:
    turn_based_training: False  # always False for Hungry Geese
    observation: False
    gamma: 0.8
    forward_steps: 32
    compress_steps: 4
    entropy_regularization: 2.0e-3
    entropy_regularization_decay: 0.3
    update_episodes: 500
    batch_size: 400  # GPU memory 10GB
    minimum_episodes: 100
    maximum_episodes: 5000  # RAM 32GB
    eval_rate: 0.1
    epochs: 120
    num_batchers: 7
    worker:
        num_parallel: 6
    lambda: 0.7
    policy_target: 'MC'
    value_target: 'MC'
    seed: 0
    restart_epoch: 0
```

3.3 Реалізація тестового ігрового середовища

Реалізація тестового ігрового середовища представлена файлом environment.py і побудована на основі бібліотеки HandyRL, що спрощує інтеграцію ігрового середовища і агентів. Дане середовище представлене в класі Environment.

Клас Environment наслідує клас BaseEnvironment з бібліотеки HandyRL. BaseEnvironment надає спрощений інтерфейс представлення багатьох агентів на ігровому полі, і вказує, які методи необхідно реалізувати для

використання даного класу. BaseEnvironment також виконує налаштування параметрів нейронної мережі та особливості тренування моделі в файлі config.yaml.

Для наслідування класу BaseEnvironment реалізовано наступні методи:

- конструктор, який створює поле для ігри «Змійка» за допомогою бібліотеки kaggle_environments;

- метод rule_based_action, що відповідає за врегулювання дій до правил для агенту «Змійка»;

- метод observation, що виконує попередню обробку даних перед тренуванням нейронної мережі;

- метод step, що виконує прогнозовані дії від нейронної мережі для кожного агенту;

- метод turns, що виконує фільтрацію тільки активних агентів які спроможні виконувати наступний крок;

- метод terminal, який виконує розрахунок того, чи може агент бути присутнім на ігровому полі;

- метод outcomes, що виконує розрахунок винагороди для кожного із агентів.

На початку класу задаються константні поля (лістинг 3.4), які характеризують кількість агентів, поведінку агентів та матрицю зміщення агенту при виборі дії агенту.

ACTION відповідає за дії, що прийнята агентами після отримання результату від нейронної мережі.

Змінна DIRECTION – це матриця, де перше значення в рядку відповідає за зміщення по вертикалі, а друге значення – по горизонталі. Наприклад, при виборі дії повернути направо, тобто дії «SOUTH», агент спуститься на одну клітинку вниз.

NUM_AGENTS відповідає за кількість агентів, які будуть створені на ігровому полі.

Лістинг 3.4 – Константні поля в класі environment.py

```
ACTION = ['NORTH', 'SOUTH', 'WEST', 'EAST']
DIRECTION = [[-1, 0], [1, 0], [0, -1], [0, 1]]
NUM_AGENTS = 4
```

В конструкторі створюється ігрове середовище, що надано бібліотекою kaggle_environments (лістинг 3.5). Щоб створити ігрове середовище для агентів ігри «Змійка» передається назва «hungry_geese» в функцію make. Дане середовище дозволяє створити ігрове поле та відображення агентів на ігровому полі.

Лістинг 3.5 – Метод створення середовища

```
def __init__(self, args={}):
        super().__init__()
        self.env = make("hungry_geese")
        self.reset()
```

Дії агенту контролюються функцією rule_based_action (лістинг 3.6), тому що отримана дія від нейронної мережі може відрізнятися від тих, які прописані в правилах. Щоб уникнути даних ситуацій, використано клас GreedyAgent який на вхід отримує останній крок та дію, що надана від ШНМ, а на вихід отримується дія, яка регульована за правилам.

Лістинг 3.6 – Метод врегулювання дій агенту для ігр «Змійка» на основі прогнозування дії від нейроної мережі

```
def rule_based_action(self, player):
        from kaggle_environments.envs.hungry_geese.hungry_geese import
Observation, Configuration, Action, GreedyAgent
        action_map = {'N': Action.NORTH, 'S': Action.SOUTH, 'W': Action.WEST,
'E': Action.EAST}
        agent = GreedyAgent(Configuration({'rows': 7, 'columns': 11}))
        agent.last_action =
action_map[self.ACTION[self.last_actions[player]][0]] if player in
self.last_actions else None
        obs = {**self.obs_list[-1][0]['observation'], **self.obs_list[-
1][player]['observation']}        action = agent(Observation(obs))
return self.ACTION.index(action)
```

Функція observation (лістинг 3.7) відповідає за підготовку масиву даних для тренування нейронної мережі. Функція формує положення їжі на полі, положення голови, тіла, хвоста на ігровому полі у вигляді матриці. Сформована двомірна матриця подається у функцію net. Це сформована нейрона мережа GeeseNet на основі торусних згорток. Під час тренування HandyRL автоматично включить метод observation до запланованих дій попередньої обробки даних перед тренуванням основної нейронної мережі.

Лістинг 3.7 – Метод врегулювання дій агенту для ігри «Змійка» на основі прогнозування дії від нейронної мережі

```
def observation(self, player=None):
        if player is None:
            player = 0
        b = np.zeros((self.NUM_AGENTS * 4 + 1, 7 * 11), dtype=np.float32)
        obs = self.obs_list[-1][0]['observation']
        for p, geese in enumerate(obs['geese']):
            # head position
            for pos in geese[:1]:
                b[0 + (p - player) % self.NUM_AGENTS, pos] = 1
            # tip position
            for pos in geese[-1:]:
                b[4 + (p - player) % self.NUM_AGENTS, pos] = 1
            # whole position
            for pos in geese:
                b[8 + (p - player) % self.NUM_AGENTS, pos] = 1

        # previous head position
        if len(self.obs_list) > 1:
            obs_prev = self.obs_list[-2][0]['observation']
            for p, geese in enumerate(obs_prev['geese']):
                for pos in geese[:1]:
                    b[12 + (p - player) % self.NUM_AGENTS, pos] = 1
        # food
        for pos in obs['food']:
            b[16, pos] = 1

        return b.reshape(-1, 7, 11)
    def net(self):
        return GeeseNet
```

Метод turns (лістинг 3.8) відповідальний за те, щоб на ігровому полі відображалися тільки активні агенти. Розрахунок поточного статусу агенту виконуються в функції terminal.

Лістинг 3.8 – Методи надання можливості виконання наступного кроку

```
def turns(self):
        # players to move
        return [p for p in self.players() if self.obs_list[-1][p]['status']
== 'ACTIVE']

def terminal(self):
    # check whether terminal state or not
    for obs in self.obs_list[-1]:
        if obs['status'] == 'ACTIVE':
            return False
    return True
```

Оскільки «Змійка» є покроковою грою, кожна змійка повинна виконувати один крок вперед, і для виконання даного кроку потрібно оновити поле і це забезпечується в функції step (лістинг 3.9). Функція виконує перехід до наступної дії і відображає обновлений результат на ігровому полі.

Лістинг 3.9 – Метод обновлення ігрового поля і перехід до наступної дії

```
def step(self, actions):
        # state transition
        obs = self.env.step([self.action2str(actions.get(p, None) or 0) for p
in self.players()])
        self.update((obs, actions), False)
```

Функція outcome (лістинг 3.10) відповідає за остаточний результат і робить розрахунок набраних фруктів для кожного агенту. Перемагає агент, який зібрав найбільшу кількість фруктів. HandyRL автоматично використовує дану функцію для інтерпретації результату нейронної мережі, щоб покращити остаточні дії агенту.

Лістинг 3.10 – Метод розрахунку винаогороди для кожного агента

```
    def outcome(self):
        # return terminal outcomes
        # 1st: 1.0 2nd: 0.33 3rd: -0.33 4th: -1.00
        rewards = {o['observation']['index']: o['reward'] for o in
self.obs_list[-1]}
```

```
        outcomes = {p: 0 for p in self.players()}
        for p, r in rewards.items():
            for pp, rr in rewards.items():
                if p != pp:
                    if r > rr:
                        outcomes[p] += 1 / (self.NUM_AGENTS - 1)
                    elif r < rr:
                        outcomes[p] -= 1 / (self.NUM_AGENTS - 1)
        return outcomes
```

Останнім етапом є запуск агентів в ігровому середовищі. Запуск програми виконується в файлі main.py (лістинг 3.11). Вона створює ігрове середовище для ігри «Змійка» та стартує роботу агентів в ігровому полі.

Лістинг 3.11 – Функція запуску ігрового середовища та агентів

```
if __name__ == '__main__':
    e = Environment()
    for _ in range(200):
        e.reset()
        while not e.terminal():
            print(e)
            actions = {p: e.legal_actions(p) for p in e.turns()}
            print([[e.action2str(a, p) for a in alist] for p, alist in
actions.items()])
            e.step({p: random.choice(alist) for p, alist in actions.items()})
        print(e)
        print(e.outcome())
```

В застосунок додано клас FloodfullResult. Алгоритм Floodfill дозволяє виявити, які місця на ігровому полі є наявними для переходу в наступний стан (лістинг 3.12).

Лістинг 3.12 – Функція запуску ігрового середовища та агентів всередині

```
@dataclass
class FloodfillResult:
    field_dist: np.ndarray
    frontiers: List[List[Tuple[int, int]]]


def flood_fill(is_occupied: np.ndarray, seeds: List[Pos]) -> FloodfillResult:
    """
    Flood will start with distance 0 at seeds and only flow where
is_occupied[x,y]==0
    """
    size_x, size_y = is_occupied.shape
```

```
    field_dist = np.full(fill_value=-1, shape=(size_x, size_y))
    frontier = [(s.x, s.y) for s in seeds]
    frontiers = [frontier]
    for seed in seeds:
        field_dist[seed] = 0
    dist = 1

    while frontier:
        new_frontier: List[Tuple[int, int]] = []
        for x, y in frontier:
            for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                new_x = (x + dx) % size_x
                new_y = (y + dy) % size_y
                if is_occupied[new_x, new_y] == 0 and field_dist[new_x,
new_y] == -1:
                    field_dist[new_x, new_y] = dist
                    new_frontier.append((new_x, new_y))
        frontier = new_frontier
        frontiers.append(frontier)
        dist += 1
    return FloodfillResult(field_dist=field_dist, frontiers=frontiers)


def get_dist(
    floodfill_result: FloodfillResult, test_func: Callable[[Tuple[int, int]],
bool]
) -> Optional[int]:
    for dist, frontier in enumerate(floodfill_result.frontiers):
        for pos in frontier:
            if test_func(pos):
                return dist
    return None
```

Алгоритм визначає, якщо клітинка занята другим агентом, то поточна змійка виконує крок в іншу сторону і планує кроки до 4 навколишніх точок, пока всі клітинки не будуть дослідженими.

4 АНАЛІЗ РЕЗУЛЬТАТІВ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Порівняння статичних та адаптивних ігрових агентів

Дослідження ефективності роботи адаптивного ігрового клієнта проводилося на основі значень метрик, що зібрані під час навчання нейронної мережі. Для дослідження використовувалися статичні та адаптивні агенти штучного ігрового інтелекту.

Побудовано графік залежності кількості перемог створеного агенту з агентом побудованим на правилах (рисунок 4.1), від кількості зіграних партій, що використані для навчання агенту. Rulebase модель представляє собою найменш ефективну статичну модель. Це простий агент, який тільки націлений збір фруктів. Графік надає інформацію, що після проведення більше 20 000 ігор адаптивний агент, здатний перемагати статичний інтелект майже в кожній грі (вірогідність перемоги близько 100%).

Також побудовано графік залежності перемоги агенту, створеного з використанням генетичного алгоритму (рисунок 4.1). Shared модель представляє собою найменш ефективну адаптивну модель. Адаптивний ігровий агент, побудований з використанням генетичного алгоритму показує значно кращий результат, ніж алгоритм побудований на правилах, але програє алгоритму створеного в ході кваліфікаційної роботи. Згідно графіку створений агент має 70 відсотків шансу перемоги при проведені 40000 зіграних ігор з агентом, побудованим на основі генетичного алгоритму. 40000 є оптимальною кількістю проведених ігор з агентом, для отримання достатнього досвіду для представлення фінального результату.

Подальше навчання моделі є недоцільним з точки зору споживання комп’ютерних ресурсів. Щоб покращити шанс перемоги створеного агенту з 0.7 до 0.8 відсотків, потребувалося в 4 рази більше часу навчання моделі. Це показує, що модель досягла стабільного результату і щоб покращити шанс

перемоги моделі необхідно розглянути нові підходи до вирішення проблем створення ігрового агенту. Також необхідна додаткова зміна параметрів, для отримання нового результату.

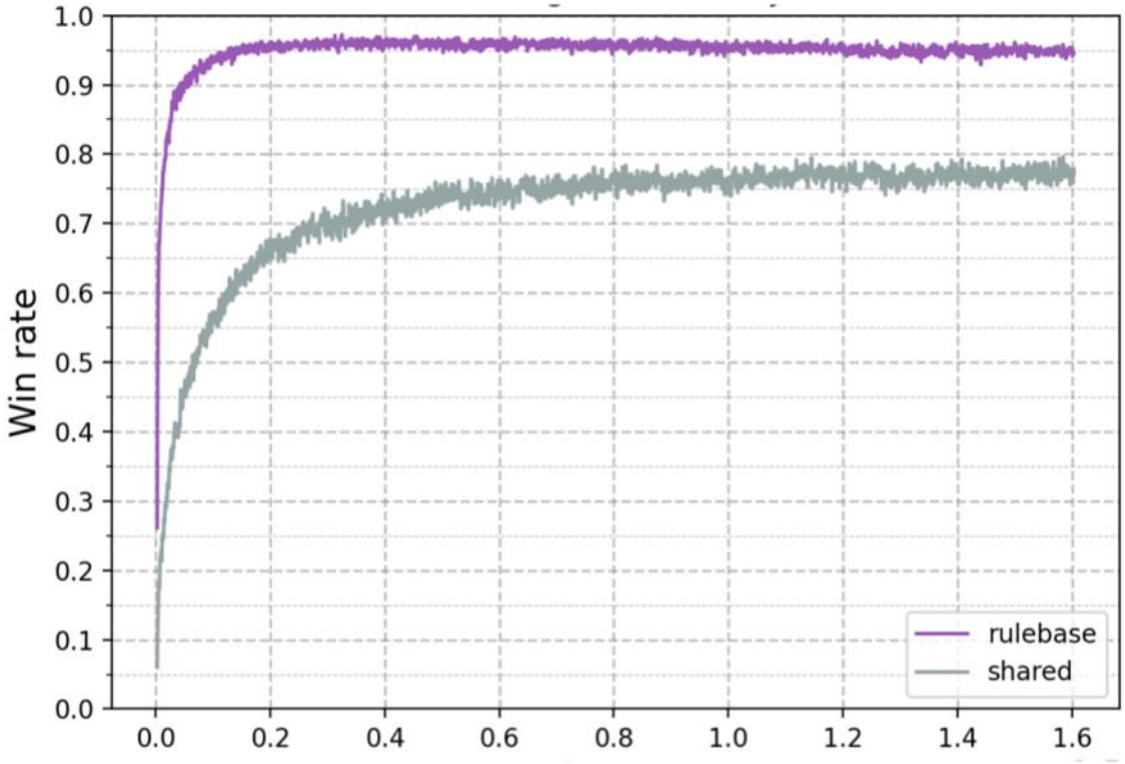

Рисунок 4.1 – Графік залежності Rulebase-агенту та Shared-агенту

Для наступного дослідження на тестове ігрове поле було добавлено 3 сторонні агенти, ігровий штучний інтелект яких побудований на основі таких підходів:

- жадібна політика та нейронна мережа на основі згорткових шарів;
- жадібна політики зі схильністю використовувати ризик та нейронна мережа на основі згорткових шарів;
- генетичний алгоритм.

Агенти, що використані у ході дослідження, представлені в ігровому середовищі на платформі Kaggle і мають можливість взаємодії між собою. У ході дослідження агенти могли виконувати до 200 кроків, і вигравав той агент, який зібрав найбільшу кількість фруктів.

Порівняння графіків залежності кількості виграних ігор від кількості зіграних партій наведено на рисунку 4.2. Для згладжування графіка було використано ковзна середня функція на 10, 25, 100 епохах. Згідно графіку ігровий агент має середній шанс перемоги приблизно 60 відсотків. Максимальний шанс перемоги був оцінений в 65 відсотків, а мінімальне значення шансу дорівнює 55 відсоткам. Також графік надає інформацію, що для досягання стабільного результату в 60 відсотків потрібно не більше 1000 епох. Це означає, що нейрона мережа досягає прийнятного результату саме при такому значенні і для покращення результату потрібно виконати додаткові дії по оптимізації алгоритму.

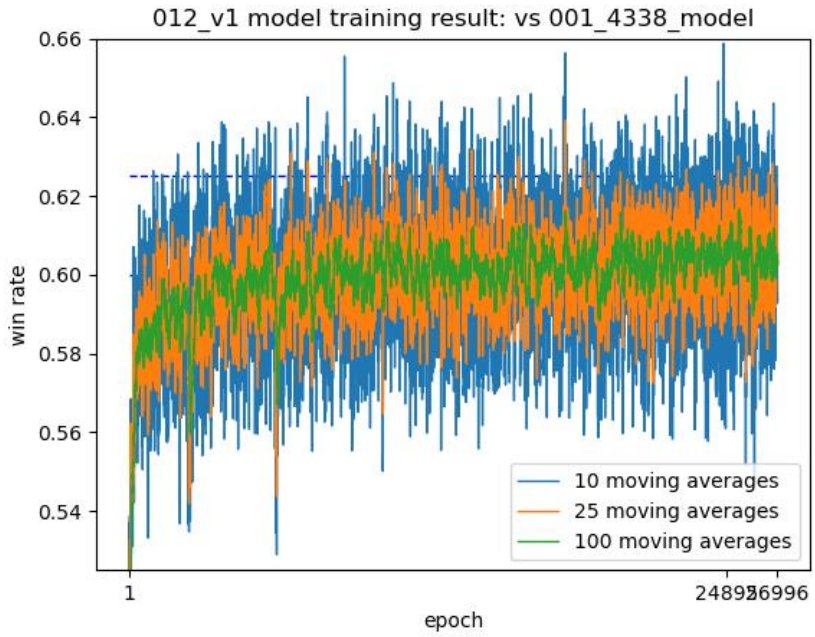

Рисунок 4.2 – Функція запуску ігрового середовища та агентів всередині

Також було проведено 2 дослідження для оцінки роботи створеного агенту. Перше дослідження – це створення 2 агентів на полі: нового агенту (Smart Geese) та найбільш популярних статичних агентів на платформі Kaggle. Розглянемо ефективність роботи у порівнянні з агентами із статичним ігровим інтелектом (таблиця 4.1).

Таблиця 4.1 – Ефективність роботи у порівнянні з агентами із статичним ігровим інтелектом

Назва суперника	Опис	Ймовірність перемоги Smart Geese
simple_toward	Простий агент, який тільки націлений збір фруктів	96%
greedy	Жадібний агент	91%
risk_averse_greedy	Жадібний агент, який не схильний до ризику (позначає всі клітини, які можуть бути небезпечними на наступному кроці, як перешкоди)	94%
simple_bfs	Великий список евристик для уникнення перешкод	95%
crazy_goose	Жадібний агент, який не схильний до ризику (позначає всі клітини, які можуть бути небезпечними на наступному кроці, як перешкоди) та має великий список евристик для уникнення перешкод	82%

Також результати проведених ігор зібрані у формі heatmap (рисунок 4.3):

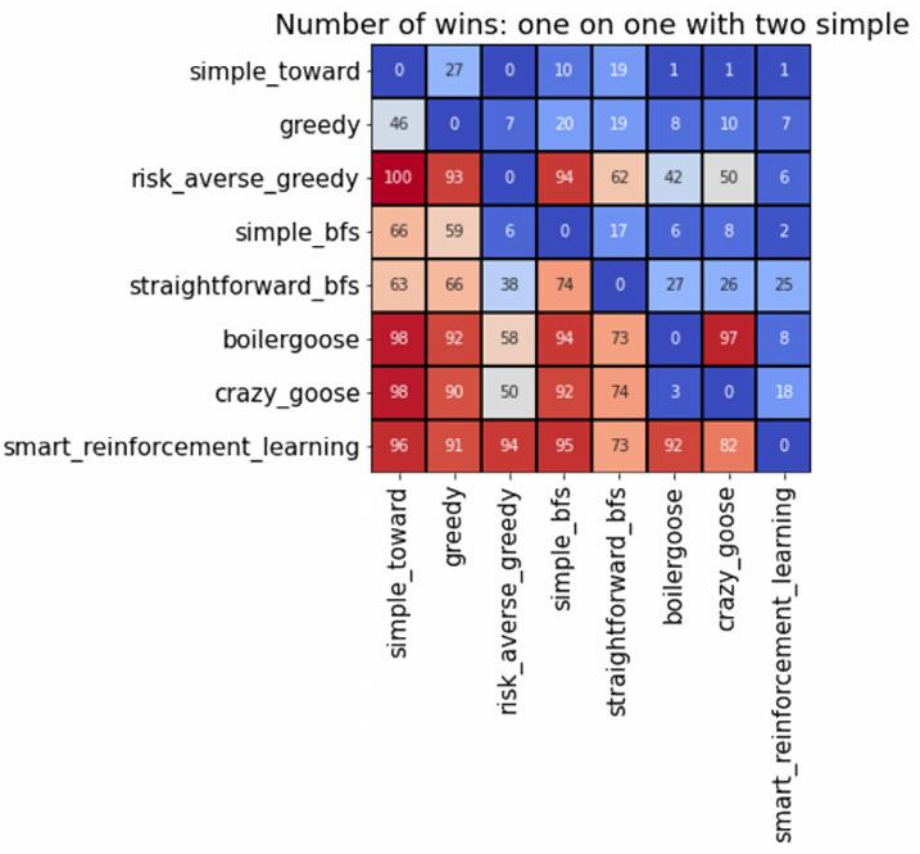

Рисунок 4.3 – Heatmap ймовірності перемоги агентів

Для ускладнення ігрового процесу, створено 3 агентів на основі статичного ігрового інтелекту проти агенту, створеного в ході кваліфікаційної роботи (таблиця 4.2).

Таблиця 4.2 – Порівняльна таблиця існуючих рішень по створенню штучного інтелекту для ігри «Змійка»

Назва суперника	Опис	Ймовірність перемоги Smart Geese
simple_toward	Простий агент який тільки націлений збір фруктів	96%
greedy	Жадібний агент	89%

Продовження таблиці 4.2

risk_averse_greedy	Жадібний агент, який не схильний до ризику (позначає всі клітини, які можуть бути небезпечними на наступному кроці, як перешкоди)	82%
simple_bfs	Великий список евристик для уникнення перешкод	98%
crazy_goose	Жадібний агент який не схильний до ризику (позначає всі клітини, які можуть бути небезпечними на наступному кроці, як перешкоди) та має великий список евристик для уникнення перешкод	79%

Також результати проведених ігор зібраних у формі heatmap (рисунок 4.4).

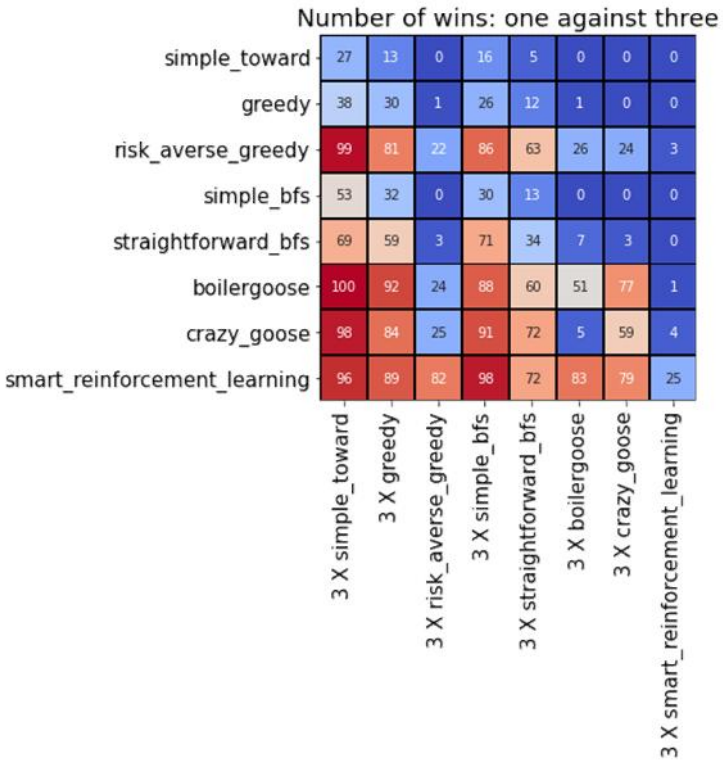

Рисунок 4.4 – Heatmap ймовірності перемоги агентів

Згідно отриманих результатів, новий агент показує незначне погіршення результатів порівняно з попереднім дослідженням, де було представлено тільки два агенти. Жадібні агенти, які не схильні до ризику, мають значно більшу кількість правил, які націлені на уникнення колізії з іншими змійками, порівняно з іншими статичними ігровими агентами, тому вони показують значно кращий результат порівняно з простим агентом, який тільки націлений збір фруктів, жадібним інтелектом та інтелектом з великою кількістю евристик. Більшисть алгоритмів, які націлені на ризик, мають значно менші шанси на виживання.

Проведено дослідження ефективності створеного агенту з динамічним ігровими інтелектом. Суперники, що отримані з платформи Kaggle, побудовані на основі адаптивного штучного інтелекту (таблиця 4.3).

Таблиця 4.3 – Ефективність роботи нового агенту порівняно з агентами із іншими популярними адаптивними ігровими інтелектами для ігри «Змійка»

Назва суперника	Опис	Ймовірність перемоги Smart Geese
genetic_agent	Ігровий штучний інтелект на основі генетичних алгоритмів.	67%
inclined_risk_agent	Ігровий штучний інтелект на основі жадібної політики зі схильністю використовувати ризик та нейронної мережі на основі згорткових шарів	56%
risk_averse_greedy	Ігровий штучний інтелект на основі жадної політики та нейронної мережі на основі згорткових шарів.	60%

4.2 Аналіз взаємодії адаптивних ігрових агентів

4.2.1 Опис обладнання та програмного забезпечення для тестування

Тестування програмного забезпечення − це процес, що призначений для виявлення інформації про якість продукту відносно контексту, в якому він має використовуватись.

Тестування ігрового штучного інтелекту проводиться на ігровому полі при взаємодії об’єктів ігри. Останньою платформою для тестування є Kaggle, де агент може зіграти із іншими агентами на одному ігровому полі і отримати оцінку в залежності від того, який агент виграв ігрову партію.

Kaggle виконує більше 50 ігрових сесій із одними і тими агентами, щоб отримати точний результат того, який агент збирає найбільшу кількість фруктів.

Програма була протестована на наступних машинах:

- Intel Core i5 2.3 GHz, 8Gb RAM, MacOS Catalina;
- AMD Ryzen 3.6 GHz, 32 Gb RAM, Windows 10, RTX 2070 Super.

4.2.2 Тестування агенту в режимі реального часу

Для демонстрації ігрового процесу було розгорнуто застосунок на платформу Kaggle. Початковий статус ігрового поля наведено на рисунку 4.5.

Ігрове поле – це матриця, 7 рядків та 11 колонок. Агенти розміщуються в довільному порядку, що згенерований платформою. Так як назва змагання, де можливо виконувати агенти, є «Hungry Geese», тому агенти мають вигляд гусаків, але по поведінці кожен агент грає як змійка. Фрукти розміщуються та появляються у випадкових клітинках на ігровому полі. Щоб зрівняти шанси кожного агенту платформа має обмеження, що фрукти не можуть знаходитися рядом. Максимальна кількість фруктів, які одночасно можуть знаходиться на ігровому полі, дорівнює 2.

На платформу було надано 4 агенту, серед яких ігровий штучний інтелект, створений в ході кваліфікаційної роботи, та агенти, що побудовані на основі таких підходів:

- жадібна політика та нейронна мережа на основі згорткових шарів;
- жадібна політика зі схильністю використовувати ризик та нейронна мережа на основі згорткових шарів;
- генетичний алгоритм.

Кожен агент представлений наступними різними кольорами гусаків. Агент номер 1 представлений синім кольором, агент номер 2 представлений зеленим кольором, агент номер 3 представлений білим кольором, а ігровий штучний інтелект, створений в ході кваліфікаційної роботи, представлений червоним кольором. Завданням кожного агенту є збір фруктів та уникнення зіткнення із тілом суперника та самим собою.

Рисунок 4.5 – Функція запуску ігрового середовища та агентів всередині

Після запуску застосунку, платформа Kaggle інтегрує агентів в ігрове середовище і запускає суперників одночасно, агенти починають гру в однакових умовах. При вживанні фрукта «Змійка» росте в довжину (рисунок 4.6).

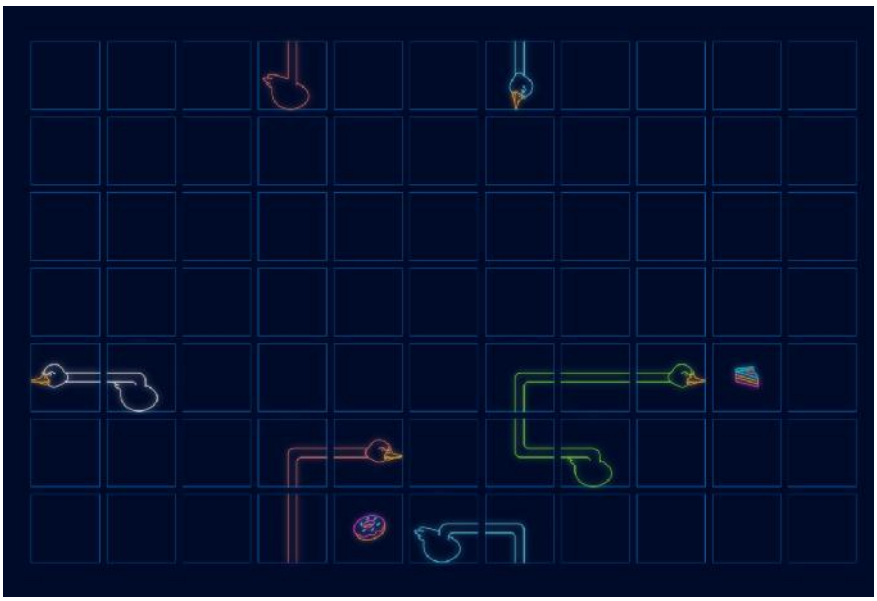

Рисунок 4.6 – Стан ігрового середовища після виконання 30 кроків

Після 30 кроків всі агенти присутні на ігровому полі і продовжують збирати фрукти. Після виконання 70 кроків червоний агент відстає на 2 фрукта від агенту, побудованого на основі генетичних алгоритмів, і займає 2 місце серед всіх змійок (рисунок 4.7).

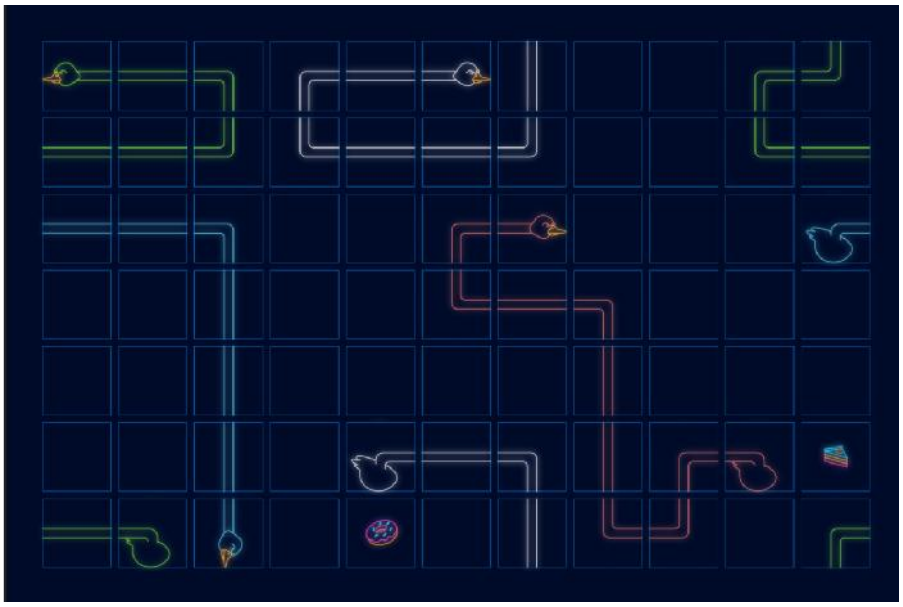

Рисунок 4.7 – Стан ігрового середовища після виконання 70 кроків

Після виконання 90 кроків більшість агентів починає стикатися з другими агентами із-за великої довжини «хвоста» змійок. На 90 кроках помирає штучний інтелект на основі генетичних алгоритмів та штучний інтелект на основі жадної політики та нейронної мережі на основі згорткових шарів. Після виконання 103 кроків агент 1 стикається із самим собою і червоний агент залишається єдиним. Так як агент залишився послідніми, то він був здатний зібрати всю необхідну їжу, щоб вирватися вперед серед всіх агентів (рисунок 4.8).

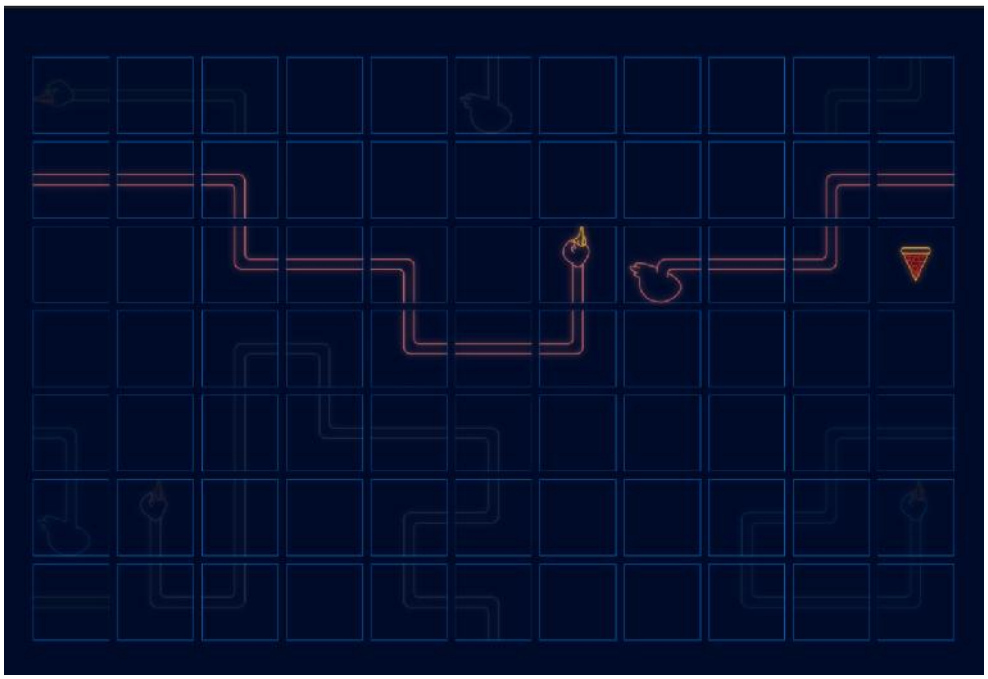

Рисунок 4.8– Стан ігрового середовища після виконання 103 кроків

Головною особливістю агенту було те, що він не мав схильності зібрати найбільшу кількість їжі за невеликий проміжок часу, а збирав фрукти послідовно балансуючи свій розмір та шанс колізії з іншими змійками.

Отже, було продемонстровано роботу створеного агенту на ігровому полі використовуючи платформу Kaggle, та інших агентів, що представлені розробниками на сайті змагання на даній платформі. Дана демонстрація представляла собою одну із зіграних партій серед 4 змійок.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи досліджені методи забезпечення адаптивності ігрового штучного інтелекту, на прикладі реалізації динамічного агенту для ігри «Змійка» на основі моделей навчання з підкріпленням (Reinforcement learning).

Проведений аналіз предметної області, створено тестове ігрове середовище для гри «Змійка» для попереднього тестування ігрових агентів, що побудовані з використання різних методів. Для побудови ігрових агентів використані Deep Q-Learning neural network та Monte-Carlo tree search, що допомогло створити базові моделі поведінки агентів.

Покращена ефективність агентів за допомогою великої кількості симуляцій, модифіковано параметри ігрового штучного інтелекту, і згідно результатів було змінено архітектуру нейронної мережі для досягнення шансу виграшу агенту більше 60 відсотків, у порівнянні з іншими агентами на платформі «Kaggle».

Для підсумкового тестування агентів було використано середовище змагання «Hungry Geese» платформи «Kaggle». Створений агент був випробуваний із іншими агентами і показав високий шанс виграшу порівняно з аналогами. Він має високу здатність до збору фруктів та досить великий час життя порівняно з аналогами, але не має схильності до збору фруктів доки всі агенти присутні на ігровому полі.

В майбутньому можливо покращити ефективність адаптивного ігрового штучного інтелекту шляхом налаштування комплексних моделей та ансамблів методів обчислювального інтелекту.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Супруненко, М. Ю. Методи забезпечення адаптивності ігрового штучного інтелекту [Текст] / М. Ю. Супруненко, Г. С. Іващенко // І Міжнародна наукова конференція «Комплексний підхід до модернізації науки: методи, моделі та мультидисциплінарність». – Вінниця, 2021. – Том 2. – С. 32–34. DOI: https://doi.org/10.36074/mcnd-19.11.2021.

2. Espeholt L. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures [Текст] / L. Espeholt, H. Soyer // Proceedings of the 35th International Conference on Machine Learning. – 2018. – P. 1–10.

3. HandyRL [Електронний ресурс] – Режим доступу : www/ URL: https://github.com/DeNA/HandyRL. – 11.11.2021 г. – Загл. з екрану.

4. Mnih V. Playing Atari with Deep Reinforcement Learning [Текст] / V. Mnih, K. Kavukcuoglu // NIPS Deep Learning Workshop 2013. – 2013. – P. 1–9.

5. Henderson P. An Introduction to Deep Reinforcement Learning [Текст] / P. Henderson, V. François-Lavet // An Introduction to Deep Reinforcement Learning. – 2018. – P. 2–5.

6. Kaiser Ł. Model Based Reinforcement Learning for Atari [Текст] / Ł. Kaiser, M. Babaeizadeh // ICLR 2020. – 2020. – P. 1–2.

7. Zhang J. Exploration of Reinforcement Learning to SNAKE [Текст] / J. Zhang, M. Tang, B. Ma // 2019 International Conference on Computational Science and Computational Intelligence (CSCI). – 2019. – P. 1–5.

8. Kaggle [Електронний ресурс] – Режим доступу : www/ URL: https://www.kaggle.com/. – 11.11.2021 г. – Загл. з екрану.

9. Hungry Geese [Електронний ресурс]– Режим доступу : www/ URL: https://www.kaggle.com/c/hungry-geese/. – 11.11.2021 г. – Загл. з екрану.

10. Paszke A. An Imperative Style, High-Performance Deep Learning

Library [Текст] / A. Paszke, S. Gross // NeurIPS. – 2019. – P. 1–12.

11. Python [Електронний ресурс] – Режим доступу : www/ URL: https://www.python.org. – 11.11.2021 г. – Загл. з екрану.

12. Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures [Електронний ресурс] – Режим доступу : www/ URL: https://github.com/deepmind/scalable_agent/tree/6c0c8a701990fab9053fb338ede9c915c18fa2b1. – 11.11.2021 г. – Загл. з екрану.

13. Bing Z. Energy-Efficient Slithering Gait Exploration for a Snake-like Robot based on Reinforcement Learning [Текст] / Z. Bing, C. Lemke // IJCAI. – 2019. – P. 1–5.

14. Fitzek D. Deep Q-learning decoder for depolarizing noise on the toric code [Текст] / D. Fitzek, M. Eliasson. – P. 1–17.

15. Swaminathan A. Off-Policy Policy Gradient with State Distribution Correction [Електронний ресурс] / A. Swaminathan, 1. Liu. – 2019. – P. 1–17.

16. Islam R. Off-Policy Policy Gradient Algorithms by Constraining the State Distribution Shift [Текст] / R. Islam, K. Teru. – 2019. – P. 1–13.

17. Chaslot G. Monte-Carlo Tree Search: A New Framework for Game AI [Текст] / G. Chaslot, S. Bakkes // AIIDE. – 2008. – P. 1–2.

18. Gelly S. Monte-Carlo tree search and rapid action value estimation in computer Go [Текст] / S. Gelly, D. Silver. – 2011. – P. 1–5.

19. Silver D. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play [Текст] / D. Silver, T. Hubert. – 2018. – P. 1–3.

20. Hendrycks D. Gaussian error linear units (GELUS) [Текст] / D. Hendrycks, K. Gimpel. – 2016. – P. 1–9.

21. Frey B. Adaptive dropout for training deep neural networks [Текст] / B. Frey, L. Ba // NIPS. – 2013. – P. 1–2.

22. N Tsitsiklis J. analysis of temporal difference learning with function approximation. [Текст] / J. N Tsitsiklis, V. Roy // NIPS. – 1996. – P. 1–2.

23. V J. analysis of temporal difference learning with function approximation [Текст] / J. V, B. Van Ro // NIPS. – 1996. – P. 1–7.

24. Sallans B. Reinforcement Learning with Factored States and Actions [Текст] / B. Sallans, G. E. Hinton // J. Mach. Learn. Res. – 2004. – P. 1–2.

25. Jen L. An application of SARSA temporal difference learning to Super Mario [Текст] / Lucas Jen – 2004. – P. 1–2.

26. Hungry Geese – Agents Comparison [Електронний ресурс] – Режим доступу : www/ URL: https://www.kaggle.com/ihelon/hungry-geese-agents-comparison. – 11.11.2021 г. – Загл. з екрану.

27. Wirth R. CRISP-DM: Towards a Standard Process Model for Data Mining [Текст] / R. Wirth, J. Hipp – 2005. – P. 1–5.

28. Cross-validation: evaluating estimator performance [Електронний ресурс] – Режим доступу : www/ URL: https://scikit-learn.org/stable/modules/cross_validation.html. – 11.11.2021 г. – Загл. з екрану.