Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Ігровий штучний інтелект з використанням генетичних алгоритмів
(тема)

Виконав:
студент 2 курсу, групи СШІм-21-1
Гриб Д.В.
(прізвище, ініціали)

Спеціальність 122 Комп’ютерні науки
(код і повна назва спеціальності)

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

Освітня програма Системи штучного інтелекту
(повна назва спеціалізації)

Керівник проф. Сніжко Д.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ________________ В.О. Філатов
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп’ютерні науки
(код і повна назва)

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

Освітня програма Системи штучного інтелекту (СШІ)
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)
«______» ___________ 20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Грибу Дмитру Володимировичу
(прізвище, ім’я, по батькові)

1. Тема роботи Ігровий штучний інтелект з використанням генетичних алгоритмів

затверджена наказом університету від 31 березня 20 23 р. № 306Ст

2. Термін подання студентом роботи до екзаменаційної комісії 17 травня 20 23 р.

3. Вихідні дані до роботи Науково-технічні публікації, відкриті джерела в мережі Інтернет, відкриті набори даних, середовище розробки JetBrains Rider 2022.2, ігровий рушій Unity, інструмент для візуального проектування App Diagrams

4. Перелік питань, що потрібно опрацювати в роботі

1) Постановка задачі
2) Аналіз предметної області
3) Розробка компонентів програмного забезпечення

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)________________

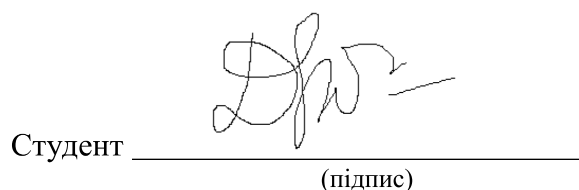
6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Вивчення предметної області дослідження	05.04.2023	виконано
2	Дослідження літератури	08.04.2023	виконано
3	Вивчення проблеми, що потребує рішення	11.04.2023	виконано
4	Формування постановки задачі дослідження	12.04.2023	виконано
5	Аналіз існуючих підходів	15.04.2023	виконано
6	Концептуальне проектування методу рішення задачі	17.04.2023	виконано
7	Проектування компонентів ПЗ	20.04.2023	виконано
8	Розробка компонентів ПЗ	23.04.2023	виконано
9	Тестування працездатності розробленого ПЗ	28.04.2023	виконано
10	Оформлення пояснювальної записки	05.05.2023	виконано
11	Попередній захист	12.05.2023	виконано
12	Захист перед ЕК	17.05.2023	

Дата видачі завдання _3_ _квітня_ 20_23_ р.



Студент ________________________________
(підпис)

Керівник роботи ____________________________ ________проф. Сніжко Д.В.________
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 75 с., 18 рис., 2 дод., 29 джерела.

ГЕНЕТИЧНІ АЛГОРИТМИ, МАШИННЕ НАВЧАННЯ, НЕЙРОМЕРЕЖІ, ОПТИМІЗАЦІЯ, РОЗРОБКА ІГОР, ШТУЧНИЙ ІНТЕЛЕКТ.

Об'єкт дослідження – штучний інтелект та його використання в комп'ютерних іграх в жанрі гонки.

Предмет дослідження – застосування генетичних алгоритмів для розв'язання задачі оптимізації параметрів штучного інтелекту в комп'ютерних іграх в жанрі гонки.

Мета роботи – дослідити можливості використання генетичних алгоритмів для оптимізації параметрів штучного інтелекту в комп'ютерних іграх в жанрі гонки з метою покращення поведінки інтелектуальних агентів та підвищення рівня геймплею.

Методи дослідження – аналіз існуючих підходів до використання штучного інтелекту в комп'ютерних іграх, проектування та розробка генетичного алгоритму для оптимізації параметрів інтелектуальних агентів в гонках, проведення комп'ютерних експериментів з використанням розробленого алгоритму.

Результати дослідження – було розроблено генетичний алгоритм для оптимізації параметрів штучного інтелекту в гонках, який використовується для автоматичного визначення оптимальних значень параметрів, що підвищує рівень ігрового досвіду. Було проведено комп'ютерний експеримент з використанням розробленого алгоритму, який показав покращення поведінки інтелектуальних агентів в гонках, що забезпечило більш реалістичний та цікавий геймплей.

ABSTRACT

Explanatory note: 75 p., 18 fig., 2 ann., 29 sources.

ARTIFICIAL INTELLIGENCE, GAME DEVELOPMENT, GENETIC ALGORITHMS, MACHINE LEARNING, NEURAL NETWORKS, OPTIMIZATION

Research object – artificial intelligence and its use in racing computer games.

Research subject – application of genetic algorithms to optimize the parameters of artificial intelligence in racing computer games.

The purpose of the study is to investigate the possibilities of using genetic algorithms to optimize the parameters of artificial intelligence in racing computer games in order to improve the behavior of intelligent agents and enhance the gameplay experience.

Research methods – analysis of existing approaches to the use of artificial intelligence in computer games, design and development of a genetic algorithm to optimize the parameters of intelligent agents in racing, conducting computer experiments using the developed algorithm.

Research results – a genetic algorithm was developed to optimize the parameters of artificial intelligence in racing, which is used to automatically determine optimal parameter values, increasing the level of gameplay experience. A computer experiment was conducted using the developed algorithm, which showed an improvement in the behavior of intelligent agents in racing, providing a more realistic and engaging gameplay experience.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень та термінів...... 7
Вступ.. 8
1 Постановка задачі.. 10
1.1 Актуальність теми... 10
1.2 Мета дослідження 10
1.3 Завдання дослідження... 11
2 Аналіз предметної області... 12
2.1 Огляд жанру гонок в комп'ютерних іграх 12
2.2 Використання штучного інтелекту в комп'ютерних іграх................. 12
2.3 Штучні нейронні мережі 12
2.4 Мережа перцептронів 13
2.5 Генетичні алгоритми.. 14
2.6 Приклади використання генетичних методів ШІ в іграх.................. 15
2.7 Функція придатності. Вимірювання ефективності ШІ 17
2.8 Мутації генетичного коду агента 19
2.9 Використання генетичних алгоритмів для оптимізації параметрів інтелектуальних агентів комп’ютерній грі.. 20
2.10 Визначення типу навчання та структури нейромережі.................... 22
2.11 Навчання БНМ. Метод зворотного поширення помилки................. 31
3 Розробка компонентів програмного забезпечення 37
3.1 Визначення стеку технологій... 37
3.2 Визначення середовища ігрових агентів 38
3.3 Проектування програмного забезпечення... 41
3.4 Загальна структура програмного забезпечення 44
Висновки 56
Перелік джерел посилання.. 57
Додаток А Код програми... 60
Додаток Б Відомість кваліфікаційної роботи.. 75

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

AI – Artificial Intelligence – штучний інтелект;

ШНМ – Штучні нейронні мережі;

ГА – Генетичні алгоритми;

RNN – Recurrent neural network – рекурентна нейронна мережа;

CNN – Convolutional neural network – згорткова нейронні мережа;

BNN – Binarized neural network – бінаризована нейронна мережа;

DNN – Deep neural network – глибинна нейронна мережа;

GNN – Graph neural network – графова нейронна мережа;

Unity – a cross-platform game engine developed by Unity Technologies

Gradient descent – an iterative first-order optimisation algorithm used to find a local minimum/maximum of a given function;

Backpropagation – a widely used algorithm for training feedforward artificial neural networks or other parameterized networks with differentiable nodes.

ВСТУП

Штучний інтелект (ШІ) в комп'ютерних іграх стає все більш популярним, оскільки дозволяє розширити можливості геймплею і створити нові виклики для гравців. У жанрі гонок штучний інтелект може бути особливо важливим, оскільки залежно від поведінки інтелектуальних агентів геймплей може бути більш чи менш складним і цікавим. Оптимізація параметрів інтелектуальних агентів в гонках з метою покращення їх поведінки може забезпечити високий рівень геймплею та забезпечити велику кількість задоволення гравців.

Один з підходів до оптимізації параметрів інтелектуальних агентів в гонках – використання генетичних алгоритмів. Генетичні алгоритми є ефективним засобом знаходження оптимальних рішень у великих просторах параметрів, що може бути важливим у випадку складних ігрових сценаріїв.

Метою цього дослідження є дослідження можливостей використання генетичних алгоритмів для оптимізації параметрів інтелектуальних агентів в комп'ютерних іграх в жанрі гонок. Робота спрямована на покращення поведінки інтелектуальних агентів та підвищення рівня геймплею в цьому жанрі. Для досягнення цієї мети будуть проаналізовані існуючі підходи до використання штучного інтелекту в комп'ютерних іграх, буде розроблений генетичний алгоритм для оптимізації параметрів інтелектуальних агентів в гонках та будуть проведені комп'ютерні експерименти для оцінки ефективності розробленого алгоритму.

Дослідження використання штучного інтелекту в комп'ютерних іграх має великий потенціал у забезпеченні реалістичної та цікавої геймплею для користувачів. В жанрі гонок, поведінка інтелектуальних агентів, таких як супротивники на трасі, є одним з ключових елементів геймплею. Відповідно, покращення їхньої поведінки може призвести до покращення рівня геймплею та задоволення користувачів.

Для досягнення поставленої мети, буде проведений аналіз існуючих підходів до використання штучного інтелекту в комп'ютерних іграх, зокрема, підходів, що використовують генетичні алгоритми. Буде розроблений генетичний алгоритм для оптимізації параметрів інтелектуальних агентів в гонках. Для перевірки ефективності алгоритму будуть проведені комп'ютерні експерименти на основі відомої гонки в жанрі комп'ютерних ігор. Результати експериментів будуть проаналізовані, і буде зроблений висновок про ефективність розробленого генетичного алгоритму для оптимізації поведінки інтелектуальних агентів в комп'ютерних іграх в жанрі гонок.

Гонки є одним із найпопулярніших жанрів в ігровій індустрії, що постійно розвивається і вдосконалюється. Для того, щоб забезпечити більш ефективну інтелектуальну поведінку ігрових персонажів, штучний інтелект (ШІ) може бути застосований в комп'ютерних іграх. ШІ може допомогти в розв'язанні задач, які не можуть бути ефективно вирішені людиною, а також забезпечити більш непередбачуваний та цікавий геймплей для гравців.

У цьому дослідженні буде використаний генетичний алгоритм для оптимізації параметрів ШІ в жанрі гонок. Генетичні алгоритми є потужним інструментом оптимізації, який заснований на еволюційному принципі відбору найкращих рішень від попереднього покоління. Використання генетичних алгоритмів для оптимізації параметрів ШІ може допомогти покращити поведінку ігрових персонажів та забезпечити більш непередбачуваний та цікавий геймплей для гравців.

У рамках дослідження будуть проведені комп'ютерні експерименти з використанням розробленого генетичного алгоритму для оптимізації параметрів ШІ в гонках. Експерименти дозволять оцінити ефективність запропонованого підходу та порівняти його з існуючими методами оптимізації параметрів ШІ.

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Актуальність теми

Сучасні технології штучного інтелекту (ШІ) та генетичні алгоритми дозволяють створювати різноманітні ігрові середовища, де комп'ютерні програми можуть взаємодіяти з гравцями на рівні, що наближається до людського. Один із напрямів застосування ШІ – ігрова індустрія, де ШІ використовують для створення гравців-ботів або опонентів, що дозволяє покращувати рівень геймплею та розширювати можливості ігрового процесу.

Зважаючи на те, що комп'ютерні ігри є дедалі більш популярними, застосування штучного інтелекту в цих іграх стає все більш важливим для покращення геймплею. Використання генетичних алгоритмів для оптимізації параметрів інтелектуальних агентів в комп'ютерних іграх є перспективним напрямом досліджень, оскільки вони можуть покращити поведінку інтелектуальних агентів та підвищити рівень геймплею в різних жанрах, зокрема в жанрі гонок. Таким чином, тема ШІ в комп'ютерних іграх має велику актуальність у сучасному світі технологій та є важливим напрямком для дослідження та розробки нових методів оптимізації інтелектуальних агентів в комп'ютерних іграх.

У гонках, як і в багатьох інших жанрах ігор, ШІ може бути використана для створення реалістичних опонентів, які можуть взаємодіяти з гравцем на рівні, що наближається до людського. Це дозволяє покращувати рівень геймплею та створювати більш складні та захоплюючі ігрові середовища.

1.2 Мета дослідження

Метою дослідження є створення ШІ для комп’ютерної гри на основі генетичних алгоритмів, яка зможе навчитися вести автомобіль по трасі. Крім

того, дослідження має на меті дослідити можливості використання штучних нейронних мереж для оптимізації руху автомобіля та покращення поведінки ботів на трасі. Також метою дослідження є розробка та експериментальне дослідження генетичного алгоритму для оптимізації параметрів інтелектуальних агентів в комп'ютерних іграх. Основним завданням дослідження є визначення можливостей використання генетичних алгоритмів для оптимізації параметрів інтелектуальних агентів та підвищення рівня геймплею в жанрі гонок.

1.3 Завдання дослідження

Для досягнення поставленої мети необхідно розв'язати наступні завдання:

Дослідити існуючі методи та алгоритми ШІ в гонках та вибрати найбільш ефективні та відповідні для застосування в контексті даної роботи.

Розробити математичну модель гонки з використанням ШІ на основі генетичних алгоритмів, яка дозволить створити інноваційну та ефективну систему ШІ для гонок.

Реалізувати розроблену модель ШІ у вигляді комп'ютерної програми, що дозволить провести експериментальне дослідження та перевірити її ефективність та функціональні можливості.

Провести експериментальне дослідження розробленої моделі на різних типах гонок, визначити її потенціал, виявити можливість покращення та оптимізації за рахунок внесення змін у параметри алгоритму ШІ.

2 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

2.1 Огляд жанру гонок в комп'ютерних іграх

Жанр гонок є одним з найпопулярніших в комп'ютерних іграх. Він характеризується головними елементами, такими як швидкість, керування автомобілем та боротьба з суперниками за перемогу. Жанр гонок може бути поділений на різні піджанри, такі як гоночні симулятори, аркадні гонки та ралі-ігри.

2.2 Використання штучного інтелекту в комп'ютерних іграх

Застосування штучного інтелекту в комп'ютерних іграх дозволяє створювати інтелектуальних агентів, які можуть пристосовуватися до різних ситуацій в грі та навчатися на основі досвіду. Штучний інтелект може бути використаний для покращення поведінки інтелектуальних агентів та для створення більш складних ігрових сценаріїв.

2.3 Штучні нейронні мережі

Штучні нейронні мережі (ШНМ) є однією з найпопулярніших математичних моделей, використовуваних в галузі ШІ. Для руху автомобіля також можна використовувати ШНМ.

Штучна нейронна мережа – це математична модель, яка імітує роботу людського мозку. Вона складається з нейронів, які пов'язані між собою в залежності від вагових коефіцієнтів, та з функції активації, яка визначає, як нейрон буде реагувати на вхідні дані.

ШНМ можна використовувати для прогнозування поведінки автомобіля на дорозі, наприклад, для визначення швидкості, кута повороту та траєкторії руху автомобіля. Для цього потрібно спочатку навчити ШНМ

за допомогою навчального набору даних, який включає інформацію про рух автомобіля та параметри дороги. Навчальний набір даних містить вхідні дані (наприклад, швидкість автомобіля, радіус повороту, нахил дороги тощо) та очікувані вихідні дані (наприклад, кут повороту, швидкість, прискорення тощо).

ШНМ для руху автомобіля може бути побудована з декількох шарів нейронів, з яких кожен шар містить різну кількість нейронів. Перший шар вводить вхідні дані, останній – виводить результати. Проміжні шари, які називаються «прихованими», обробляють інформацію та забезпечують зв'язок між вхідним та вихідним шарами [21].

2.4 Мережа перцептронів

Ще одним прикладом математичної моделі, яка може бути використана в генетичних алгоритмах для ШІ в іграх, є мережа перцептронів. Це математична модель, яка імітує роботу нейронних мереж, тобто систем, що працюють на основі взаємозв'язків між «нейронами», які обчислюють значення відповідно до вхідних даних.

Мережа перцептронів складається зі шарів нейронів, які мають зв'язки між собою з різними вагами. Кожен нейрон отримує вхідні дані з попереднього шару, обчислює значення за допомогою функції активації і передає результат наступному шару. Останній шар мережі видає результат.

Мережі перцептронів можуть бути використані для розв'язання різних завдань у гральній індустрії, таких як розпізнавання образів, класифікація даних, передбачення поведінки гравців, і т. д. Вони також можуть бути використані для навчання гравців, наприклад, для розпізнавання імовірних місць, де може з'явитися ворог, або для передбачення траєкторії руху гравця.

Формула для роботи нейрона в мережі перцептронів:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

де x – сума добутків вхідних даних на ваги зв'язків між нейронами.

Крім того, можуть бути використані інші математичні моделі, такі як марківські моделі, моделі з деревом пошуку, генетичні програми та інші, залежно від завдання та специфіки конкретної гри. Важливо вибрати правильну модель та налаштувати параметри генетичного алгоритму.

2.5 Генетичні алгоритми

Штучний інтелект (ШІ) в комп'ютерних іграх стає все більш розповсюдженим і складнішим з кожним роком. Один зі способів реалізації ШІ в іграх – це використання генетичних алгоритмів (ГА).

Генетичні алгоритми є методом оптимізації, який використовує еволюційні принципи для пошуку оптимального розв'язку. Генетичні алгоритми включають у себе оператори селекції, кросоверу та мутації, які дозволяють створювати нові комбінації генів та покращувати якість розв'язку.

Генетичний алгоритм – це евристичний алгоритм пошуку, що моделює природний процес еволюції. ГА складається з популяції рішень, кожне з яких представляється виглядом генотипу. Генотипи потім оцінюються за допомогою функції придатності, і ті, що мають кращу придатність, копіюються у нову популяцію. У новій популяції виконуються мутації та скрещування, які створюють нові рішення, які знову оцінюються за допомогою функції придатності. Цей процес повторюється доки не буде досягнуто прийнятне рішення [4].

У випадку з комп'ютерними іграми, генотипи можуть представляти характеристики ігрових персонажів, такі як швидкість руху, силу атаки та витривалість. Функція придатності може бути визначена за кількістю очок,

які здобуває персонаж під час гри, чи за тим, наскільки швидко він проходить рівні.

Інший підхід до використання генетичних алгоритмів в комп'ютерних іграх – це навчання штучної нейронної мережі за допомогою ГА. В цьому випадку генотипи представляють параметри нейронної мережі, такі як кількість шарів, кількість нейронів в кожному шарі та вагові коефіцієнти. Функція придатності може бути визначена за точність прогнозування результатів штучної нейронної мережі на тестових наборах даних.

Використання генетичних алгоритмів для розробки ШІ в комп'ютерних іграх має декілька переваг. По-перше, ГА можуть знайти оптимальні параметри для ШІ, які можуть бути складними для налагодження вручну. По-друге, використання ГА дозволяє створювати ШІ, які можуть адаптуватися до зміни вимог гри, забезпечуючи більш гнучкий ігровий процес.

Проте, використання генетичних алгоритмів для розробки ШІ також має деякі недоліки. По-перше, ГА можуть вимагати значної кількості обчислювальних ресурсів, особливо якщо популяція має великий розмір. По-друге, залежність від функції придатності може призвести до певних обмежень відносно того, які характеристики можуть бути вдосконалені ШІ.

У підсумку, генетичні алгоритми є ефективними інструментами для розробки штучного інтелекту в комп'ютерних іграх. Використання цих алгоритмів дозволяє створювати ШІ, які можуть адаптуватися до змін у грі, забезпечуючи більш гнучкий ігровий процес. Однак, важливо розуміти, що використання генетичних алгоритмів не є універсальним рішенням, і кращий метод для розробки ШІ може залежати від специфіки гри та її вимог.

2.6 Приклади використання генетичних методів ШІ в іграх

Генетичні методи ШІ в іграх використовують принцип еволюції, де штучні інтелектуальні агенти проходять через процес відбору, схрещування

та мутації, щоб створити нову популяцію, яка є кращою у виконанні певної задачі, такої як перемога в іграх. Генетичні алгоритми широко використовуються в розробці ігор, таких як стратегії в реальному часі, рольові ігри та ігри з головоломками.

Один з прикладів використання генетичних алгоритмів в ШІ – це гра «Flappy Bird». У цій грі мета гравця полягає в тому, щоб уникати перешкод, рухаючись вгору і вниз на певній висоті. У цьому прикладі, генетичні алгоритми використовуються для того, щоб створити оптимальні параметри для ШІ, які допоможуть гравцеві максимально довго зберігатися в живих. Починаючи з початкової популяції ШІ, агенти з найкращою функцією придатності, яка вимірює, як добре ШІ грає в гру, зберігаються та використовуються для створення нової популяції, яка потім проходить через той самий процес відбору, схрещування та мутації.

Інший приклад використання генетичних алгоритмів в ШІ – це гра «Pac-Man». У цій грі мета гравця полягає в тому, щоб зібрати якомога більше їжі, уникати привидів та збирати бонуси для отримання додаткових очок. У цьому прикладі, генетичні алгоритми використовуються для створення оптимальної стратегії для ШІ, щоб максимально зібрати їжу та отримати якомога більше очок. Починаючи з початкової популяції ШІ, агенти з найкращою функцією придатності, яка вимірює, як добре ШІ грає в гру, зберігаються та використовуються для створення нової популяції. У цьому випадку, функція придатності вимірюється за кількістю зібраної їжі та очок.

Ще одним прикладом використання генетичних алгоритмів в ШІ є гра «StarCraft II», де генетичні алгоритми використовуються для створення оптимальної стратегії в реальному часі. У цій грі мета полягає в тому, щоб виконувати різні завдання, такі як збір ресурсів та виконання військових операцій. ШІ повинні приймати швидкі рішення та змінювати свою стратегію в залежності від дій гравців та розвитку гри. Генетичні алгоритми використовуються для створення оптимальних стратегій для різних типів ситуацій у грі.

Усі ці приклади використання генетичних алгоритмів в ШІ демонструють, як еволюційний підхід може допомогти створити оптимальні стратегії та допомогти ШІ стати більш ефективними в іграх. Завдяки використанню математичних моделей та генетичних алгоритмів, можливо створити більш складні та вимогливі ігри, де ШІ повинні пристосовуватися до різних ситуацій та шукати оптимальні рішення для досягнення максимальних результатів.

2.7 Функція придатності. Вимірювання ефективності ШІ

Один з прикладів математичної моделі, яку можна використовувати в генетичних алгоритмах для ШІ, – це функція придатності, яка використовується для вимірювання ефективності ШІ в іграх. Функція придатності – це функція, яка визначає, наскільки добре даний агент справляється зі своєю задачею. У грі на збирання їжі, функція придатності може бути визначена наступним чином:

$$\mathrm{f} = x * n + y * n \tag{2.2}$$

де «x» – це кількість зібраної їжі, а «y» – це кількість набраних очок агентом. У цій формулі, чим більше їжі збирається та більше очок набирається агентом, тим вище його функція придатності.

Функція придатності, також відома як фітнес-функція, є ключовим елементом генетичного алгоритму. Ця функція визначає, наскільки «хорошим» є поточний варіант рішення, який описується набором генів. У контексті ШІ в іграх, функція придатності визначає, наскільки ефективною є стратегія гравця в грі.

Функція придатності повинна бути такою, щоб вона можна була ефективно обчислювати та відображати різницю між «хорошими» та «поганими» рішеннями. Для цього вона може бути визначена як числова

оцінка, яка відображає рівень успіху генетичного рішення в задачі. Ця оцінка зазвичай повинна бути максимізована під час еволюції рішень, оскільки метою є знайти найкраще можливе рішення. [16]

Наприклад, у випадку гри в Го, функція придатності може бути визначена як кількість території, яку зайняв генетичний агент. Оскільки метою гри є зайняття якомога більшої території, функція придатності повинна бути високою для генетичних агентів, які займають більше території. В інших випадках, функція придатності може бути більш складною та включати багато факторів, які впливають на успіх стратегії в грі.

Функція придатності може бути вибрана залежно від задачі, яку потрібно вирішити. Наприклад, використання функції придатності може бути використане для оптимізації параметрів алгоритмів, які використовуються для прийняття рішень у ШІ. У цьому випадку, функція придатності може відображати якість рішення, яке було прийняте з використанням конкретних параметрів. Наприклад, якщо ми використовуємо генетичний алгоритм для оптимізації параметрів нейромережі, функція придатності може відображати точність моделі на валідаційному наборі даних. Ми можемо обчислити цю точність за допомогою метрик, таких як середньоквадратичне відхилення або коефіцієнт детермінації.

Для визначення функції придатності, також можна використовувати інші методи, такі як машинне навчання або статистичний аналіз. Наприклад, у грі в шахи, функція придатності може визначатися за допомогою класифікатора, який навчається розрізняти переможні та програшні позиції. Таким чином, генетичний алгоритм може збирати гіпотези щодо найбільш ефективних стратегій гри та відшукувати їх, використовуючи функцію придатності та інші методи аналізу даних.

Взагалі, функція придатності є важливим компонентом генетичного алгоритму, що дозволяє ШІ вирішувати складні задачі та шукати оптимальні

рішення в іграх. Вона дозволяє оцінювати якість рішень та використовувати цю інформацію для того, щоб вибрати найбільш ефективні стратегії та навчити ШІ виконувати складні завдання в іграх [19].

2.8 Мутації генетичного коду агента

Інший приклад математичної моделі, який можна використовувати в генетичних алгоритмах для ШІ – це формула, яка визначає імовірність мутації генетичного коду агента. Мутації – це процес зміни генетичного коду агента, який може допомогти агенту адаптуватися до змінних умов гри. Формула, яка визначає ймовірність мутації, може мати вигляд:

$$y = n * (1 - \sqrt[P]{\sum_{k=1}^{m} (x_{ik} - x_{jk})^P}) \qquad (2.3)$$

де «n» – це константа, яка визначає частоту мутацій. У цій формулі, чим менше подібності між двома агентами, тим більша ймовірність мутації їх генетичного коду.

Мутації генетичного коду агента є однією з ключових характеристик, яка впливає на його поведінку в комп'ютерних іграх. За допомогою мутацій генетичного коду можна створювати нові та унікальні властивості для агента, що дозволяє підвищити ефективність гри.

У процесі мутацій генетичного коду агента, зазвичай використовуються різні методи оптимізації, такі як генетичні алгоритми, щоб створювати нові комбінації генетичних властивостей. Ці методи дозволяють враховувати наявність різних факторів

Мутації генетичного коду агента стали предметом багатьох досліджень в галузі комп'ютерних ігор. Штучний інтелект, що використовується в сучасних іграх, може бути програмований з допомогою

генетичних алгоритмів, які змінюють генетичний код агента з кожним наступним поколінням.

Мутації генетичного коду агента зазвичай здійснюються шляхом зміни випадкового гену або групи генів. Це дозволяє отримати нові комбінації генів, які можуть призвести до поліпшення поведінки агента у грі [1].

Інший приклад використання генетичних алгоритмів у ШІ в іграх – це створення нейромережі, яка може вивчати оптимальну стратегію гри. Нейромережі – це алгоритми машинного навчання, які можуть використовуватися для вивчення оптимальної стратегії в іграх. Генетичні алгоритми можуть використовуватися для оптимізації параметрів нейромережі та вибору найкращої нейромережі з популяції.

Отже, генетичні алгоритми можуть бути потужним інструментом у розробці ШІ в іграх. Вони можуть використовуватися для створення оптимальних стратегій, оптимізації параметрів ШІ та створення нейромереж, які можуть вивчати оптимальну стратегію гри.

2.9 Використання генетичних алгоритмів для оптимізації параметрів інтелектуальних агентів комп’ютерній грі

Використання генетичних алгоритмів для оптимізації параметрів інтелектуальних агентів в гонках є перспективним напрямом досліджень. Генетичні алгоритми є потужним інструментом для оптимізації параметрів інтелектуальних агентів в комп'ютерних іграх, зокрема в гонках.

Генетичні алгоритми можуть бути дуже ефективними для оптимізації параметрів інтелектуальних агентів в комп'ютерній грі гонки. Основна ідея полягає в тому, щоб створити популяцію індивідів, кожен з яких представляє собою набір параметрів, що контролюють поведінку агента в грі. Наприклад, ці параметри можуть включати швидкість, кут повороту, реакцію на перешкоди та інші характеристики, які впливають на результат гонки.

Потім за допомогою вибірки, кросоверів і мутацій, здійснюється еволюція популяції, щоб знайти оптимальні параметри для агента, який може вибрати найкращий шлях у грі та досягти перемоги. Одним з важливих аспектів при використанні генетичних алгоритмів є визначення критерію фітнесу, який буде використовуватися для оцінки кожного індивіда в популяції. Наприклад, цим критерієм може бути час проходження траси, кількість стикань з перешкодами, загальний рахунок тощо [18].

Генетичні алгоритми можуть знайти оптимальні параметри для агента, який може показувати кращі результати, ніж агенти з фіксованими параметрами, або навіть людські гравці. Однак, важливо знати, що процес оптимізації може зайняти багато часу і потребує багато обчислювальних ресурсів, особливо при великих популяціях і складних гральних середовищах.

Таким чином, генетичні алгоритми можуть бути потужним інструментом для оптимізації параметрів інтелектуальних агентів у комп'ютерних іграх гонок, але, важливо також розглядати і інші підходи до розв'язання проблеми оптимізації параметрів агента, такі як навчання з підсиленням або еволюційні стратегії.

Навчання з підсиленням є альтернативним методом оптимізації параметрів інтелектуального агента, що полягає в тому, щоб навчити агента поводитися у грі шляхом посилення його поведінки через нагороди або покарання. В цьому методі, агент пропонує дії, а середовище повертає нагороду або штраф в залежності від того, наскільки ефективні були дії агента. Під час навчання з підсиленням, агент змінює свої параметри таким чином, щоб максимізувати отримані нагороди.

Інший підхід, який можна використовувати в оптимізації параметрів агента, – це еволюційні стратегії. Цей підхід також ґрунтується на ідеї еволюції, але відрізняється від генетичних алгоритмів. Замість того, щоб використовувати випадкові мутації, еволюційні стратегії використовують диференційні мутації, які залежать від поточного стану агента.

Отже, для оптимізації параметрів інтелектуальних агентів у комп'ютерних іграх гонок, можна використовувати генетичні алгоритми, навчання з підсиленням або еволюційні стратегії. Вибір конкретного підходу залежить від складності гри, кількості параметрів, які потрібно оптимізувати, та доступних обчислювальних ресурсів.

2.10 Визначення типу навчання та структури нейромережі

У сучасних комп'ютерних іграх віртуальні агенти відіграють важливу роль, створюючи реалістичне імітацію дійсності та покращуючи ігровий досвід користувачів. Для досягнення цього були створені різні типи нейромереж, призначені для навчання віртуальних агентів.

Штучні нейронні мережі можна класифікувати за різними ознаками, такими як тип вхідної інформації, кількість шарів, характер навчання, характер налаштування синапсів, час передачі сигналу та характер зв'язків.

Класифікація за типом вхідної інформації:

– прямий доступ (feedforward neural networks);

– зворотний зв'язок (feedback neural networks).

Класифікація за кількістю шарів:

– одношарові (single-layer neural networks);

– багатошарові (multilayer neural networks).

Класифікація за характером навчання:

– навчання з вчителем (supervised learning);

– навчання без вчителя (unsupervised learning);

– підсилююче навчання (reinforcement learning).

Класифікація за характером налаштування синапсів:

– жорстке налаштування (hard-coded weights);

– м'яке налаштування (soft-coded weights).

Класифікація за часом передачі сигналу:

– синхронні (synchronous neural networks);

– асинхронні (asynchronous neural networks).

Класифікація за характером зв'язків:

– повнозв'язні (fully connected neural networks);

– зв'язки з обмеженою кількістю;

– зв'язки з локальними зв'язками.

Кожен тип архітектури штучної нейронної мережі має свої особливості та застосування в різних областях. Наприклад, багатошарові нейронні мережі звичайно використовуються для обробки зображень та розпізнавання мови, тоді як рекурентні нейронні мережі використовуються для обробки послідовностей даних, таких як мовлення або текст [11].

Одним з найбільш поширених типів нейромереж для навчання віртуальних агентів є зворотньо-поширтовані нейронні мережі (backpropagation neural networks). Цей тип нейромережі використовується для навчання віртуальних агентів на основі великої кількості даних, які надаються вхідними сигналами. Він використовується для розв'язання різних задач віртуального середовища, таких як розпізнавання образів, навігація та управління рухом [17].

Іншим важливим типом нейромереж є рекурентні нейронні мережі (recurrent neural networks), які використовуються для вивчення послідовностей даних, таких як мовленнєві тексти та вхідні дії користувача в ігровому середовищі. Цей тип нейромережі може використовуватися для передбачення наступного кроку в діяльності віртуального агента на основі попередніх дій та поведінки [4].

Також для навчання віртуальних агентів використовуються еволюційні нейронні мережі (evolutionary neural networks), які використовують генетичні алгоритми для визначення найкращих параметрів нейромережі. Цей тип нейромережі зазвичай використовується для розв'язання складних задач, що потребують великої кількості навчальних даних.

Існують також глибинні нейронні мережі (deep neural networks, DNN), які мають більш складну структуру та зазвичай використовуються для

розв'язування складних задач, таких як обробка зображень або мовленнєва розпізнавання.

До інших типів нейромереж, які використовуються для навчання віртуальних агентів для комп'ютерних ігор, належать рекурентні нейронні мережі (recurrent neural networks, RNN), які зазвичай використовуються для роботи з послідовністю даних, таких як текст або звук. Іншим важливим типом є сверхкрупні нейронні мережі (convolutional neural networks, CNN), які зазвичай використовуються для обробки зображень [9].

Також можна виділити генеративні нейронні мережі (generative neural networks, GNN), які здатні генерувати нові дані на основі навчальних даних. Вони часто використовуються для створення віртуальних світів та персонажів для комп'ютерних ігор [13].

Нейромережі також можуть бути поділеними на залежні та незалежні від контексту (context-dependent та context-independent). Контекстно-залежні нейромережі використовують контекст, такий як попередній вхід або контекстну інформацію, для зміни внутрішнього стану мережі та прийняття рішень.

Таким чином, для навчання віртуальних агентів для комп'ютерних ігор можуть бути використані різні типи нейромереж, залежно від конкретних вимог та задач, які потрібно розв'язати. Кожен тип мережі має свої переваги та обмеження, тому вибір конкретної нейромережі залежить від конкретного завдання, що потрібно вирішити.

Для вирішення поставленої задачі було обрано ГА – генетичні алгоритми. Генетичні алгоритми – це один із підходів до розв'язання задач оптимізації, який базується на принципах природної еволюції. Використання генетичних алгоритмів в різних областях, таких як машинне навчання, розпізнавання образів та робототехніка, здобуває все більшу популярність [12].

Однією з переваг генетичних алгоритмів є їх здатність до роботи з комплексними задачами оптимізації, які можуть мати багато різних варіантів

рішення. Генетичний алгоритм може допомогти знайти глобальний максимум або мінімум у великому просторі можливих рішень, забезпечуючи збільшення швидкості та точності пошуку [15].

Іншою важливою перевагою генетичних алгоритмів є їх здатність до роботи зі складними системами, для яких не завжди можна отримати точні аналітичні рішення. Генетичні алгоритми можуть допомогти вирішити такі задачі шляхом створення та оцінювання багатьох варіантів рішень.

Крім того, генетичні алгоритми можуть бути легко адаптовані для вирішення різних задач оптимізації, таких як навчання нейромереж, вибір оптимального маршруту для робота або оптимізація виробничих процесів.

Також генетичні алгоритми дозволяють використовувати багато різних видів функцій мети, зокрема, необхідних для задач мультикритеріальної оптимізації.

Ці алгоритми є особливо корисними в складних задачах, де традиційні методи оптимізації можуть не працювати ефективно або взагалі не застосовуватися. До переваг генетичних алгоритмів можна віднести:

Робота зі складними, нелінійними функціями. Генетичні алгоритми можуть оптимізувати функції, що мають багато мінімумів та максимумів, а також функції, що містять плато.

Пошук оптимального рішення в широкому просторі параметрів. Генетичні алгоритми можуть працювати зі значним числом параметрів, що дає змогу знайти оптимальне рішення в багатовимірному просторі.

Гнучкість та адаптивність. Генетичні алгоритми можуть працювати з різними типами даних та функцій. Вони можуть бути налаштовані для різних типів задач та відрізнятися за підходом до оптимізації.

Пошук глобального мінімуму. Генетичні алгоритми можуть знайти глобальний мінімум функції, що дає можливість знайти найкраще рішення для задачі.

Ефективність. Генетичні алгоритми можуть працювати з великою кількістю рішень та ефективно збільшувати шанси на знаходження оптимального рішення [3].

Зазначимо, що генетичні алгоритми мають деякі недоліки, такі як необхідність налаштування параметрів, можливість застрягання у локальному мінімумі та обмежена швидкість оптимізації. Однак, при правильному застосуванні, генетичні алгоритми можуть бути ефективним та потужним інструментом для вирішення різних задач оптимізації. Для досягнення кращих результатів, важливо правильно налаштувати параметри алгоритму, включаючи розмір популяції, ймовірність мутації та ймовірність схрещування [14].

Однією з головних переваг генетичних алгоритмів є їх здатність працювати з великими об'ємами даних та великою кількістю параметрів. Вони також можуть працювати з дуже складними функціями та нелінійними відношеннями між змінними, що робить їх ефективним інструментом для розв'язання задач в складних системах.

Ще однією перевагою генетичних алгоритмів є їх здатність знаходити оптимальні рішення в багатовимірних просторах швидше, ніж традиційні методи оптимізації. Вони можуть виявляти глобальні максимуми та мінімуми з функціями, що мають багато локальних екстремумів.

Крім того, генетичні алгоритми можуть працювати з різними типами даних, включаючи категоріальні, бінарні та неперервні дані. Це дає можливість їх використання в широкому спектрі дисциплін, таких як фінанси, економіка, маркетинг, технічні науки та інші.

Було розглянуто наступні види нейронних мереж, які можуть бути використані для штучного інтелеку в комп’ютерній грі:

Одношарові персептрони – це прості нейронні мережі, які можуть використовуватися для класифікації об'єктів у комп'ютерних іграх. Вони складаються зі входів, ваг і вихідного шару, де застосовується функція

активації. Однак, вони обмежені в своїй здатності розрізняти складні патерни [7].

Багатошарові персептрони складніші за одношарові персептрони і можуть розрізняти більш складні патерни (рисунок 2.1). Вони складаються з багатьох шарів, кожен з яких містить набір нейронів (рисунок 2.2). Кожен шар оброблює вхідні дані, зіставляє їх з вагами та використовує функції активації для передачі відповіді на наступний шар. Багатошарові персептрони можуть використовуватися для класифікації об'єктів, передбачення результатів, а також для генерації контенту у комп'ютерних іграх [8].

Рекурентні нейронні мережі мають зв'язки, які дозволяють попередній вихідний стан нейронів впливати на подальші результати. Це забезпечує мережам здатність до розуміння послідовностей, що робить їх корисними для здійснення вибору дій в комп'ютерних іграх [6].

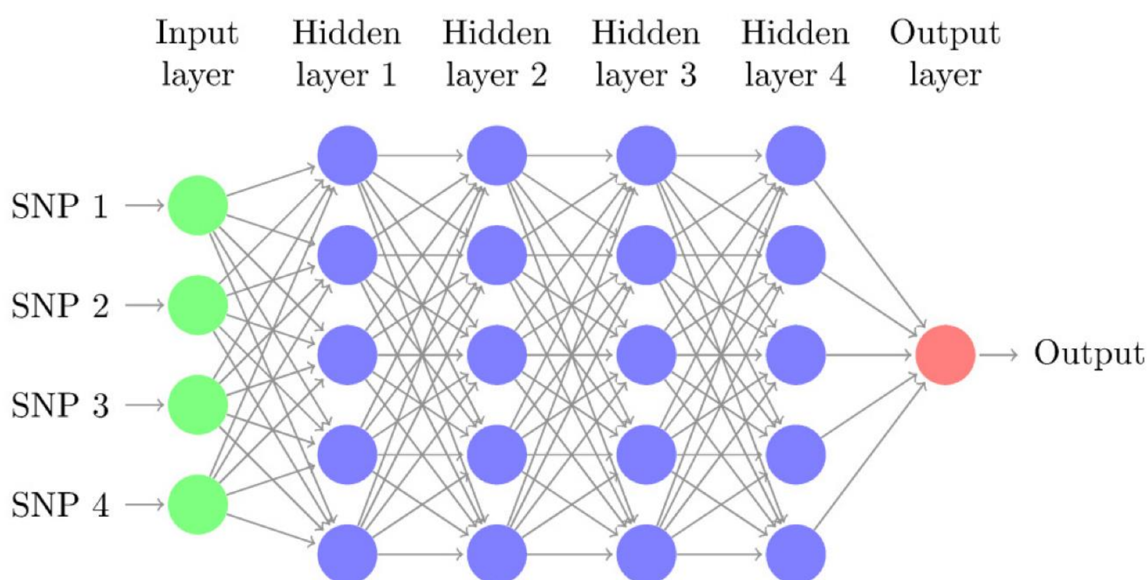


Рисунок 2.1 – Діаграма багатошарового персептрона (MLP) із чотирма прихованими шарами та набором однонуклеотидних поліморфізмів (SNP) як вхідні дані та ілюструє базовий «нейрон» із n входами

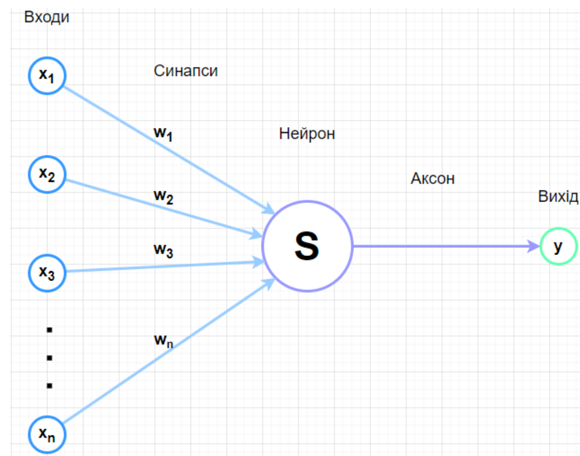


Рисунок 2.2 – Структурна діаграма нейрона

Для вирішення поставленої задачі, було обрано багатошарову, повновз'язну нейронну мережу (багатошаровий персептрон), тому що в ігровому середовищі існує багато факторів які впливають на вибір штучного інтелекту в детермінований момент часу. Багатошарова нейронна мережа (БНМ) прямого поширення (багатошаровий персептрон) складається з формальних нейронів і характеризується наступними параметрами і властивостями: M – кількість шарів мережі, Nμ – кількість нейронів у μ-му шарі, зв'язки між нейронами у шарі відсутні. Виходи нейронів μ-го шару, μ=1,2,..., M–1, надходять на входи нейронів тільки наступного μ+1-го шару. Зовнішній векторний сигнал x надходить на входи нейронів тільки першого шару, виходи нейронів останнього M-го шару утворюють вектор виходів мережі y (M).

Кожен i-й нейрон (μ, i)-го шару ((μ, i)-й нейрон) перетворює вхідний вектор x(μ, i) у вихідну скалярну величину y(μ, i). Це

$$\varphi^{(\mu,i)}(w^{(\mu,i)}, x^{(\mu,i)}) \quad (2.4)$$

яка далі перетворюється на вихідну величину

$$y^{(\mu,i)} = \psi^{(\mu,i)}(\varphi^{(\mu,i)}(w^{(\mu,i)}, x^{(\mu,i)})), \tag{2.5}$$

$$w^{(\mu,i)} = (w_0^{(\mu,i)}, w_1^{(\mu,i)}, \ldots, w_{N\mu}^{(\mu,i)})^T, \tag{2.6}$$

де $w^{(\mu,i)}$ – вектор вагових коефіцієнтів нейрона;

$x^{(\mu,i)}$ – j-та компонента N_μ – вимірного вхідного вектора $x^{(\mu,i)}$.

Якість рішення, одержуваного БНМ, буде істотно залежати від кількості шарів, кількості нейронів у кожному шарі і кількості зв'язків між шарами. Загальна кількість ваг і порогів у БНМ визначається формулою:

$$N_w = \sum_{\mu=1}^{M} N_\mu \left(N_\mu + 1\right), N_0 = N, \tag{2.7}$$

Для різних класів задач, що вирішуються за допомогою НМ, запропоновані евристичні оцінки кількості шарів і нейронів. Для забезпечення узагальнювальних властивостей необхідно, щоб кількість ваг і порогів БНМ

$$N_w \ll NS, \tag{2.8}$$

де N – розмірність вхідного сигналу (кількість ознак);

S – кількість екземплярів у навчальній вибірці.

Виходячи для простоти з того, що розмірність виходу БНМ дорівнює одному (тобто мережа має на останньому шарі тільки один нейрон), а кількість шарів може становити два або три (менше двох не дозволить вирішувати нелінійні задачі, а більше трьох шарів непотрібно, бо відомо, що тришарова БНМ здатна апроксимувати будь-яку обчислювану функцію), а також приймаючи $N \approx S$ отримаємо:

– для двошарової БНМ:

$$N_w = N_1(N+1) + (N_1+1) = N_1(N+2) + 1 \ll NS,$$
$$N_1 \ll \frac{NS-1}{N+2} \Rightarrow N_1 \ll N; \qquad (2.9)$$

– для тришарової БНМ:

$$N_w = N_1(N+1) + N_2(N_1+1) + (N_2+1) =$$
$$= N_1(N+1) + N_2N_1 + 2N_2 + 1 \ll NS. \qquad (2.10)$$

Приймемо $N_2 = 0{,}5N_1$ а $N_1N = N_1^2$ тоді для тришарової БНМ отримаємо:

$$1{,}5N_1N_1 + 2N_1 \ll NN - 1 \Rightarrow N_1 \ll N. \qquad (2.11)$$

Відомі й інші підходи. Наприклад, для оцінки кількості нейронів у прихованих шарах однорідних двошарових БНМ з сігмоїдними функціями активації можна використовувати формулу для оцінки необхідного числа синаптичних ваг N_w:

$$\frac{N_M S}{1+\log_2 S} \le N_w \le N_M\left(\frac{S}{N_M}+1\right)(N+N_M+1) + N_M \qquad (2.12)$$

де N_M – розмірність вихідного сигналу.

Оцінивши необхідну кількість ваг, можна розрахувати N_H – кількість нейронів у прихованих шарах. Наприклад, для двошарової мережі вона складе:

$$N_H = \frac{N_w}{(N+\ N_M)} \qquad (2.13)$$

Відомі й інші формули для оцінки, наприклад:

$$2(N + N_H + N_M) \leq S \leq 10(N + N_H + N_M);$$
$$0{,}1S - N - N_M \leq N_H \leq 0{,}5S - N - N_M. \quad (2.14)$$

2.11 Навчання БНМ. Метод зворотного поширення помилки

Процес навчання БНМ здійснюється у результаті мінімізації цільової функції – певного критерію якості $E(Q_{(\varepsilon,s)})$, що характеризує інтегральну міру близькості виходів мережі $y^{(M)}$ і вказівок вчителя

$$y^{*} = \{y^{S^{*}}\}, s = 1,2, \dots, S, \quad (2.15)$$

де s – номер поточного екземпляра навчальної вибірки;

S – кількість екземплярів у навчальної вибірки;

$Q_{(\varepsilon,s)}$ – миттєвий критерій якості, що залежить від вектора помилки мережі для i-ї вихідної змінної s-го екземпляра

$$\varepsilon(s, i, w) = y^{(M,i)} - y_i^{S^{*}}, i = 1,2, \dots, N_M, \quad (2.16)$$

де w – множина ваг мережі;

$y^{(M,i)}$ та $y_i^{S^{*}}$ – відповідно, розраховане мережею і бажане значення i-ї вихідної змінної для s-го екземпляра навчаючої вибірки.

Як миттєвий критерій якості найбільш часто використовують суму квадратів помилки

$$Q_{(\varepsilon,s)} = \sum_{i=1}^{N_M} \varepsilon(s, i, w)^2, \quad (2.17)$$

або суму модулів помилки для s-го екземпляра вибірки

$$Q_{(\varepsilon,s)} = \sum_{i=1}^{N_M} |\varepsilon(s,i,w)|, \qquad (2.18)$$

Інтегральну міру близькості визначають за формулою:

$$E = \frac{1}{\upsilon} \sum_{s=1}^{S} Q_{(\varepsilon,s)}, \qquad (2.19)$$

де υ – деякий коефіцієнт (якщо $\upsilon=1$, то E визначає сумарну миттєву помилку; якщо $\upsilon = S$, то E визначає середню миттєву помилку).

Для кожного вхідного вектора x s з навчальної множини повинен бути визначений вектор бажаних виходів мережі y^S. Якщо навчальна БНМ використовується в якості класифікатора, то зазвичай бажані виходи мають низький рівень (0 або менше 0,1), крім виходу вузла, відповідного класу, до якого належить x; цей вихід у даному випадку має високий рівень (1 або більше 0,9). Для початку навчання ваги БНМ заданої структури встановлюють рівними випадковим числам. Для мінімізації цільової функції навчання вирішують завдання багатомірної нелінійної оптимізації, для чого традиційно використовують градієнтні методи. Ці методи носять ітеративний характер, оскільки компоненти градієнта виявляються нелінійними функціями. Градієнтні методи засновані на ітераційній процедурі корекції значень ваг, реалізованої у відповідності з формулою:

$$w_{k+1} = w_k + a_k p(w_k), k = 0,1,2, \ldots, \qquad (2.20)$$

де w_k, w_{k+1} – поточне і нове наближення значень ваг і порогів НМ до оптимального рішення, відповідно;

α_k – крок збіжності;

$p(w_k)$ напрям пошуку в багатовимірному просторі ваг.

Спосіб визначення $p(w_k)$ та a_k на кожній ітерації залежить від особливостей конкретного методу. Оскільки градієнтні методи вимагають

обчислення похідних цільових функції для визначення напрямку пошуку, для розрахунку частинних похідних цільової функції за вагами мережі використовують метод зворотного поширення помилки (error backpropagation method), який був розроблений як узагальнення методу Уідроу-Хоффа для БНМ з нелінійними диференційовними функціями активації. Метод зворотного поширення помилки для настроювання ваг НМ використовує градієнт функції помилки таким чином, щоб мінімізувати помилку на навчальній вибірці. Для цього мережі послідовно надаються вхідні вектори (екземпляри) з навчальної вибірки, і, починаючи з останнього шару, обчислюються градієнти функції помилки для кожного нейрона:

$$\frac{\partial E}{\partial w_j^{(\mu,i)}} = \frac{\partial E}{\partial \psi^{(\mu,i)}} \frac{\partial \psi^{(\mu,i)}}{\partial \varphi^{(\mu,i)}} \frac{\partial \varphi^{(\mu,i)}}{\partial w_j^{(\mu,i)}}, \tag{2.21}$$

$$\frac{\partial \psi^{(\mu,i)}}{\partial \varphi^{(\mu,i)}} = \psi^{,(\mu,i)}(\varphi^{(\mu,i)}). \tag{2.22}$$

Якщо в якості дискримінантної функції нейронів мережі використовується зважена сума, то

$$\frac{\partial \varphi^{(\mu,i)}}{\partial w_j^{(\mu,i)}} = x_j^{(\mu,i)} = \psi^{(\mu-1,j)}. \tag{2.23}$$

Для нейронів вихідного шару:

$$\frac{\partial E}{\partial \psi^{(M,i)}} = y_i^s - \psi^{(M,i)}. \tag{2.24}$$

Для нейронів інших шарів:

$$\frac{\partial E}{\partial \psi^{(M,i)}} = \sum_j \frac{\partial E}{\partial \psi^{(\mu+1,j)}} \frac{\partial \psi^{(\mu+1,j)}}{\partial \varphi^{(\mu+1,j)}} \frac{\partial \varphi^{(\mu+1,j)}}{\partial \psi^{(\mu,j)}}$$

$$\frac{\partial \varphi^{(\mu+1,j)}}{\partial \psi^{(\mu,j)}} = w_j^{(\mu,i)}. \quad (2.25)$$

У методі зворотного поширення помилки ваги і пороги змінюються у напрямку, протилежному градієнту. Тому базовий метод зворотного поширення часто називають методом градієнтного спуску (gradient descent). Базовий метод зворотного поширення помилки складається з таких кроків.

Крок 1. Задати навчальну вибірку екземплярів x і зіставлених ним значень цільового параметру y^*. Ініціалізувати всі ваги і пороги мережі. Задати значення прийнятного рівня помилки ξ. Встановити покажчик поточного екземпляра вибірки s = 1.

Крок 2. Подати на вхід БНМ s-й екземпляр з навчальної вибірки даних $x^S = \{x_i^S\}$, а на вихід – зіставлений йому бажаний вихідний вектор

$$y^{S^*} = \{y_j^{S^*}\}, s = 1,2,\dots,S; i = 1,2,\dots,N; j = 1,2,\dots,N_M \quad (2.26)$$

Крок 3. Рухаючись від першого шару до останнього, розрахувати значення на виходах нейронів і фактичний вихід мережі

$$y^{(M)} = \{y^{(M,i)}\}, i = 1,2,\dots,N_M \quad (2.27)$$

при подачі на вхід мережі екземпляра x^s.

Крок 4. Розрахувати помилку мережі *E*. Якщо *E* >ξ (умова закінчення навчання для s-го екземпляра не виконується), тоді перейти на крок 5; в іншому випадку – перейти на крок 6.

Крок 5. Рухаючись від вихідного шару до першого шару мережі провести настроювання ваг і порогів за формулою

$$w_{k+1} = w_k + a_k p_{(w_k)}, \tag{2.28}$$

визначивши напрямок пошуку в багатовимірному просторі ваг за формулою:

$$p\left(w_{j,k}^{(\mu,i)}\right) = -g_{j,k}^{(\mu,i)}, \tag{2.29}$$

де $g_{j,k}^{(\mu,i)} = \frac{\partial E}{\partial w_j^{(\mu,i)}}$ – градієнт помилки за j-ю вагою i-го нейрона μ-го шару БНМ на k-й ітерації навчання, який визначається за формулою:

$$g_{j,k}^{(M,i)} = \psi_k^{,(M,i)}(\varphi_k^{(M,i)})(y_i^S - \psi_k^{(M,i)}(\varphi_k^{(M,i)}))\psi_k^{(M-1,i)}(\varphi_k^{(M-1,i)}), \tag{2.30}$$

а для нейронів інших шарів за рекурентною формулою:

$$g_{j,k}^{(M,i)} = \psi_k^{,(M,i)}(\varphi_k^{(M,i)})(\sum_{j=1}^{N_{\mu+1}} g_{i,k}^{(\mu+1,j)} w_{i,k}^{(\mu+1,j)})x_{p,k}^{(\mu,i)}$$
$$\mu = M-1,\ldots,1; p = 0,\ldots,N_{\mu-1}. \tag{2.31}$$

Якщо у мережі використовуються нейрони з сигмоїдною функцією активації, то для них:

$$\psi^{,(\mu,i)} = \left(\frac{1}{1+e^{-\varphi^{(\mu,i)}(w^{(\mu,i)},x^{(\mu,i)})}}\right)' = \frac{e^{-\varphi^{(\mu,i)}(w^{(\mu,i)},x^{(\mu,i)})}}{(1+e^{-\varphi^{(\mu,i)}(w^{(\mu,i)},x^{(\mu,i)})})^2} =$$
$$\frac{1}{1+e^{-\varphi^{(\mu,i)}(w^{(\mu,i)},x^{(\mu,i)})}}\left(1-\frac{1}{1+e^{-\varphi^{(\mu,i)}(w^{(\mu,i)},x^{(\mu,i)})}}\right) = \psi^{(\mu,i)}(1-\psi^{(\mu,i)}) \tag{2.32}$$

Перейти на крок 4.

Крок 6. Встановити: *s = s +1*. Якщо s > S, то перейти на крок 7, упротилежному випадку – перейти на крок 2.

Крок 7. Визначити помилку мережі E. Якщо вона прийнятна ($E \leq \xi$), то завершити пошук і видати як результат роботи значення ваг і порогів w, у протилежному випадку – встановити s = 1 та перейти до кроку 2. Як додаткові критерії завершення пошуку у методі зворотного поширення помилки можуть використовуватися досягнення ліміту часу або ліміту кількості циклів корекції ваг (епох навчання) або досягнення заданого мінімального значення градієнта цільової функції. Стандартний метод зворотного поширення часто збігається дуже повільно, рухаючись вздовж плоских ділянок поверхні помилки. Тому на практиці його часто використовують лише для розрахунку приватних похідних цільової функції помилки за вагами НМ, а пошук в просторі ваг НМ роблять на основі більш швидких градієнтних методів. Основним недоліком традиційно використовуваних градієнтних методів навчання БНМ є обумовлена ітераційною корекцію ваг низька швидкість збіжності навчання, яка серйозно обмежує практичне застосування нейронного керування. Іншим не менш важливим недоліком даних методів є невизначеність у виборі початкової точки пошуку, параметрів архітектури та топології мережі [1].

3 РОЗРОБКА КОМПОНЕНТІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Визначення стеку технологій

Для виконяння даного завдання було обрано рушій Unity. Unity – це інтегроване середовище розробки (IDE) та ігровий рушій для створення багатоплатформних ігор та інших інтерактивних додатків. Він підтримує розробку ігор для різних платформ, таких як мобільні пристрої, настільні комп'ютери, ігрові консолі та віртуальні та доповнені реальності. Unity була розроблена компанією Unity Technologies та була випущена в 2005 році. На даний момент цей ігровий рушій є одним з найпопулярніших та найвикористовуваніших у світі. За допомогою Unity можна розробляти ігри, віртуальні тури, симуляції та інші інтерактивні додатки. Unity має простий та інтуїтивно зрозумілий інтерфейс, що дозволяє розробникам швидко створювати різноманітні ігри та додатки. Він також має потужні можливості, які дозволяють розробникам створювати складні ігри з високоякісною 3D-графікою та фізикою. Unity має вбудовану систему скриптів, яка дозволяє програмістам використовувати мови програмування, такі як C# та JavaScript, для створення логіки гри та розширення функціональності. Unity також має велику спільноту розробників, яка надає безліч додатків, ресурсів та плагінів для розширення функціональності та полегшення процесу розробки. Завдяки своїй потужності, простоті використання та широкій підтримці, Unity став дуже популярним серед розробників ігор та інших інтерактивних додатків.

C# (C Sharp) – це об'єктно-орієнтована мова програмування, розроблена компанією Microsoft в 2000 році як частина платформи .NET. Мова C# має синтаксис, схожий на C++, проте має більш просту та зручну систему типів даних.

C# є однією з найпопулярніших мов програмування, використовуваних для розробки Windows-додатків, веб-додатків, сервісів та ігор. Мова має декілька переваг, які сприяють її популярності:

Широкий спектр функціональних можливостей, що дозволяють програмістам розробляти додатки різних розмірів та складності;

Інтеграція з платформою .NET, що дозволяє розробникам використовувати стандартні бібліотеки, які значно спрощують процес програмування;

Можливість використовувати мову C# як для компіляції в бінарний код, так і для виконання на платформі .NET у вигляді інтерпретованої мови;

Широкі можливості для розробки веб-додатків, включаючи підтримку технологій ASP.NET та Razor Pages.

Основними принципами мови C# є наступні:

- об'єктно-орієнтованість;
- безпека типів;
- керування пам'яттю з допомогою збирача сміття;
- підтримка делегатів та лямбда-виразів;
- розширення за допомогою розширюваних методів.

У загальному, мова C# є потужним інструментом для розробки різноманітних додатків, що дозволяє програмістам створювати швидкі та надійні рішення для різних платформ та пристроїв.

3.2 Визначення середовища ігрових агентів

Проектування середовища для ігрових агентів – це складний процес, який передбачає розробку віртуального світу, який би забезпечував зручні умови для взаємодії між гравцем та ігровими агентами. Для цього зазвичай використовують спеціалізовані інструменти, такі як Unity, які надають можливості для швидкої розробки та налаштування ігрового середовища.

Середовище, яке створюється для ігрових агентів, повинне відповідати вимогам реалістичності, тому в процесі розробки враховуються базові закони фізики, такі як поверхневе тертя, тяжіння, інерція та інші. Ці закони фізики моделюються в середовищі за допомогою спеціальних алгоритмів та фізичних рушіїв, що дозволяє створити відчуття реалістичності та природності поведінки ігрових об'єктів. Зокрема, поверхневе тертя залежить від матеріалу поверхні та сили натиску, що створюється на неї. Так, м'яч, який котиться по гладкій поверхні, буде рухатися без відчутного опору, тоді як тяжкий предмет на тій же поверхні може зупинитися через декілька метрів. Тяжіння, з свого боку, моделюється як сила, яка притягує тіла до Землі. Інерція, в свою чергу, відображає тенденцію об'єкта продовжувати рух з тією ж самою швидкістю та напрямом, як у попередній момент, якщо на нього не діє жодна зовнішня сила (рисунок 3.1).

Рисунок 3.1 – Середовище віртуальних агентів

Одним з ключових аспектів середовища є взаємодія об'єктів з ігровими агентами, яка повинна бути реалістичною і відповідати базовим законам фізики. Наприклад, об'єкти повинні мати правильну поведінку відносно поверхневого тертя, тяжіння, пружності та інших законів фізики. Це дозволяє створювати досить складні сценарії для гравців, де кожен рух агента впливає на його подальшу поведінку в середовищі. Важливою складовою проектування середовища для ігрових агентів є також дизайн взаємодії агентів з об'єктами. Наприклад, важливо, щоб агенти могли взаємодіяти з об'єктами, які знаходяться на різних висотах або розміщені в просторі. Для цього можна використовувати спеціальні техніки, такі як трасування променів або колізії, що дозволяють агентам взаємодіяти з об'єктами вірно. Крім того, середовище повинно мати реалістичну графіку та звукове супроводження, що допомагає створити цілісний іммерсивний досвід гравця. Для цього використовуються різноманітні техніки, такі як реалістичне моделювання світла, тіней та текстур, а також звукові ефекти та музика.

Ще одним важливим аспектом проектування середовища є створення штучного інтелекту для ігрових агентів. Інтелект повинен бути настільки розвиненим, щоб агенти могли приймати рішення залежно від ситуації в грі. Іншими словами, вони повинні мати можливість вчитися і адаптуватися до нових обставин. Для цього можна використовувати нейронні мережі, генетичні алгоритми або інші методи машинного навчання

Отже, проектування середовища для ігрових агентів є складним завданням, яке вимагає комплексного підходу та використання різноманітних технологій. Від правильного дизайну середовища залежить успіх гри, тому необхідно приділяти цьому аспекту належну увагу.

3.3 Проектування програмного забезпечення

Першим кроком проектування є створення структури програмного забезпечення за допомогою діаграм класів. Діаграми класів є важливим інструментом в області об'єктно-орієнтованого програмування. Вони дозволяють представити класи, їх атрибути та методи, зв'язки між класами та іншу інформацію про систему.

Основні переваги використання діаграм класів включають:

− візуалізація структури системи: діаграми класів допомагають розуміти, як класи взаємодіють між собою та як вони впливають на функціонування системи в цілому;

− виявлення проблем: діаграми класів можуть допомогти виявити проблеми з дизайном системи, такі як зайві залежності між класами або невідповідність між інтерфейсами класів;

− покращення комунікації: діаграми класів дозволяють розробникам та іншим зацікавленим сторонам легко обговорювати та розуміти структуру системи;

− підтримка генерації коду: діаграми класів можуть бути використані для автоматичної генерації коду, що може значно зменшити час розробки та знизити ризик виникнення помилок.

Узагалі, діаграми класів допомагають зрозуміти та відобразити складну систему у вигляді простої моделі, що полегшує процес розробки, аналізу та управління проектом.

На рисунку 3.2 зображена схема взаємодії компоеннів розробленої системи. Всього вона складається з 8 компонентів. VirtualEnvironment – це компонент який відповідає за віртуальне оточнення для агентів. Модуль агента всередині має під-модуль навігації який взаємодіє з віртуальним оточненням.

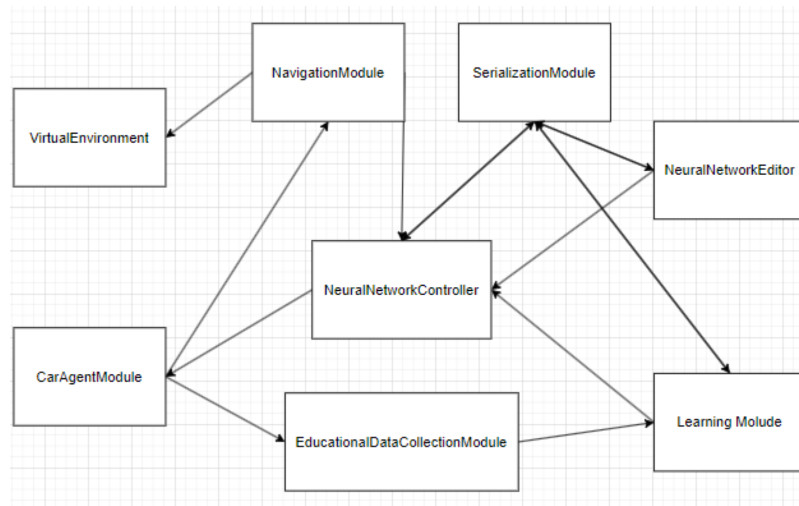


Рисунок 3.2 – Діаграма зв’язків між модулями розробленої системи

Система навчання нейронних мереж представленя двома модулями: LearningModule – призначений для запуску навчання на основі зібраних нвчальних даних із модуля – EducationalDataCollectionModule. Також окремим моделуем є редактор в якому можна створювати/редагувати моделі нейромереж. Модуль Serialization потрібен для переводу моделі нейромержі в формат JSON і навпаки (рисунок 3.3–3.6).

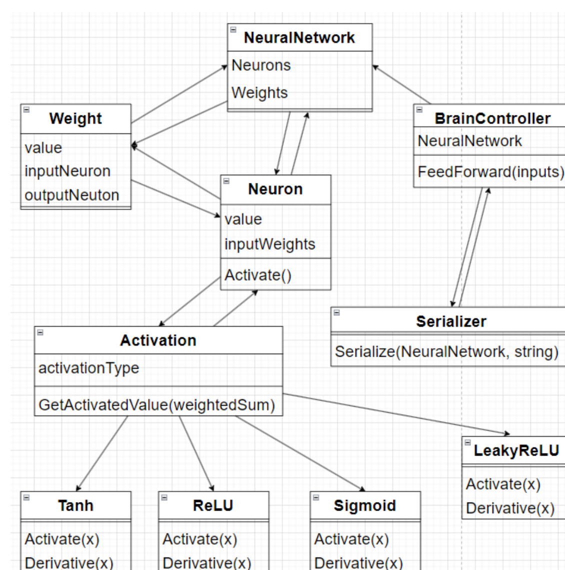


Рисунок 3.3 – Діаграма класів розробленої моделі нейронної мережі

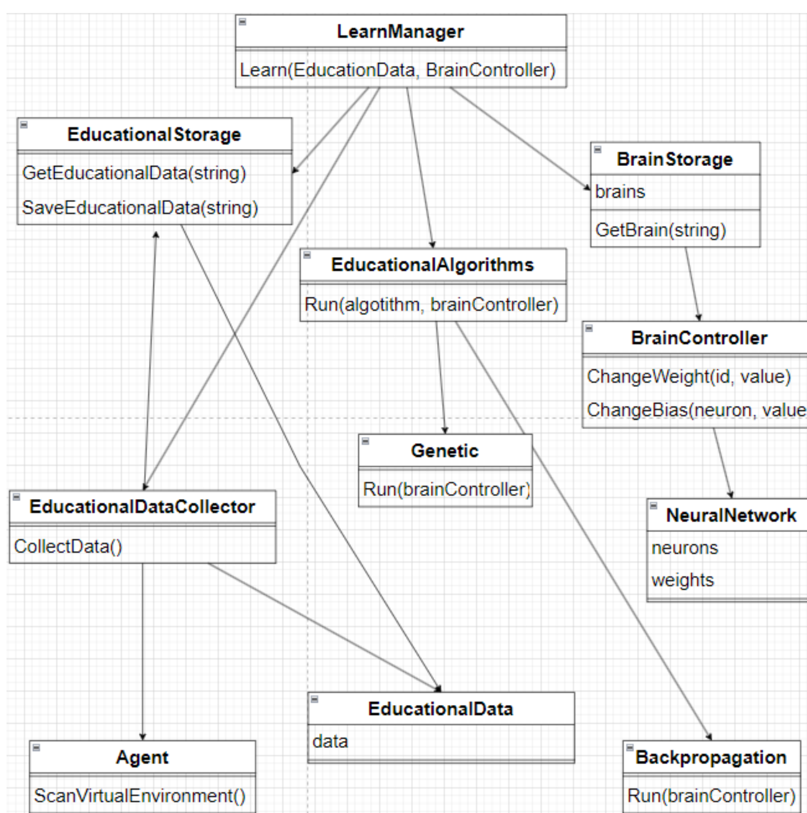


Рисунок 3.4 – Діаграма класів модуля навчання нейронної мережі

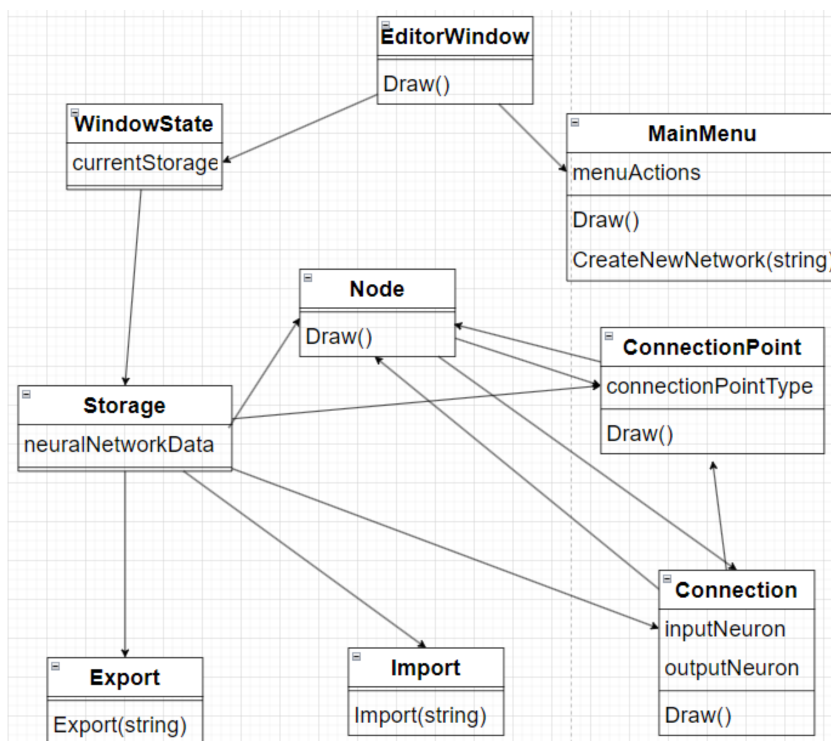


Рисунок 3.5– Діаграма класів редактора нейронних мереж

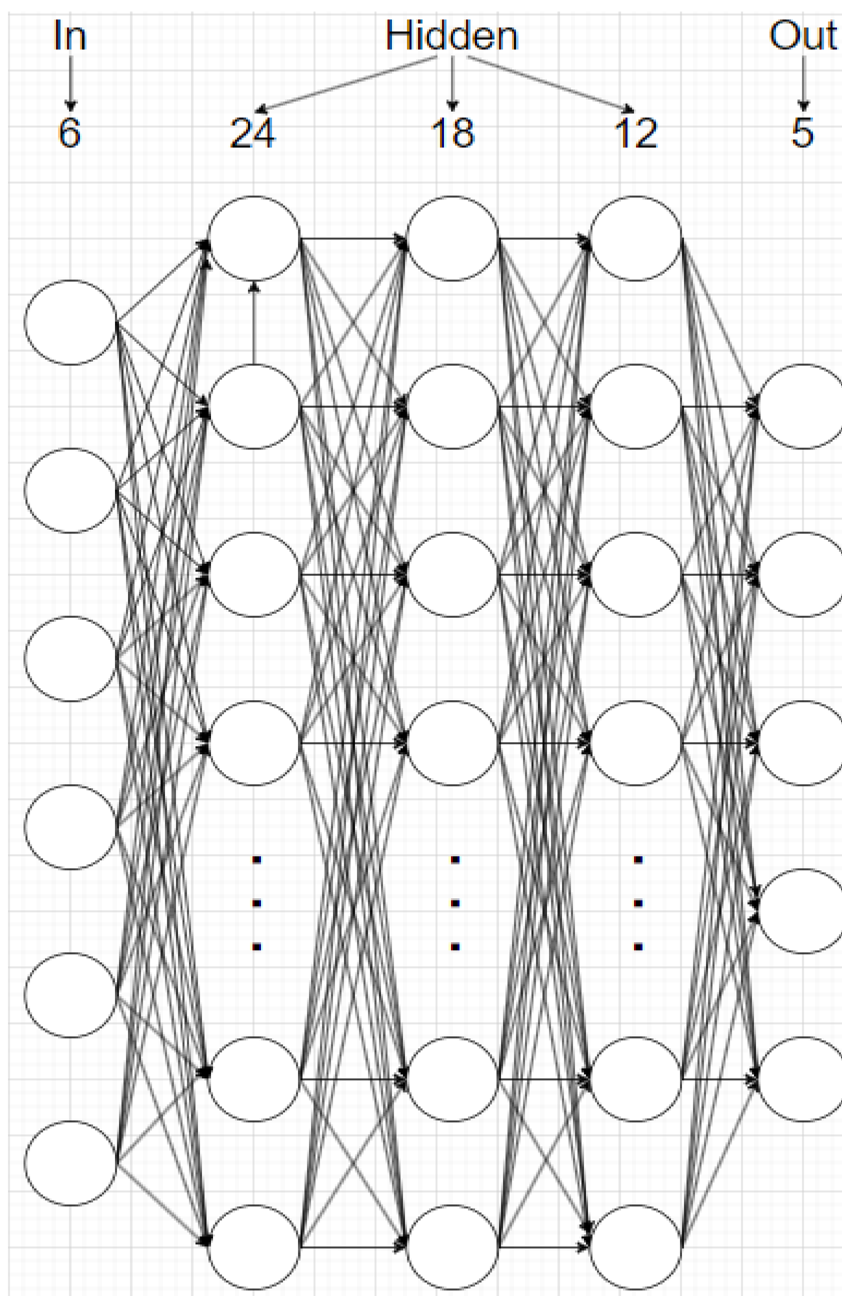


Рисунок 3.6 – Структурна діаграма розробленої нейронної мережі

3.4 Загальна структура програмного забезпечення

Для застосування розробленої нейронної мережі було розроблено середовище для віртуальних агентів. Середовище являє собою комп'ютерну гру в жанрі гонки, з використанням фізичного рушія. Програмне забезпечення включає такі частини:

– редактор нейромереж;

– модель нейромержі;

– контроллер моделі нейромережі;

– модуль навчання;

– середовище для віртуальних агентів;

– модуль віртуального агента;

– модуль імпорту/експорту моделей в формат JSON.

Редактор нейромереж реалізовано у вигляді звичайого редактора вузлів. Кожен вузол – окремий нейрон. Вузли можна поєднувати зв’язками через які передається сигнал. Редактор нейромереж забезпечує зручний та інтуїтивно зрозумілий інтерфейс для користувачів, що дозволяє легко створювати, навчати та тестувати моделі нейромереж. Крім того, він забезпечує можливість інтеграції з іншими програмними засобами, що дозволяє розробникам ефективно використовувати нейромережі в різних сферах діяльності. В редакторі реалізована функція експорту в формат JSON, що в поєднанні з незалежними від середовища Unity моделлю та контроллером нейромережі, дає змогу використовувати створені моделі нейромереж в будь-яких інших проектах з використанням мови C#.

В редакторі реалізований функціонал автоматичної генерації повнозв’язних нейронних мереж, з можливістю налаштування кількості входів/виходів та прихованих шарів (рисунок 3.7).

Також є можливість налаштування функції активації для кожного шару мережі. Додатково можна задавати поріг генерації випадкових значень для зв’язків між нейронами. Окрім цього згенеровану модель можна редагувати, наприклад, додавати нові шари, нейрони, зв’язки тощо.

Кожному окремому нейрону, за потреби можна індивідуально налаштувати функцію активації. Також є опція корегування значеннь зв’язків між нейронами (рисунок 3.8–3.9).

Рисунок 3.7 – Вікно параметрів для генерації нейромережі

Загалом у нейрона є 4 параметри які можна налаштовувати:

– значення активації;

– тип активації;

– bias;

– шар до якого належить нейрон.

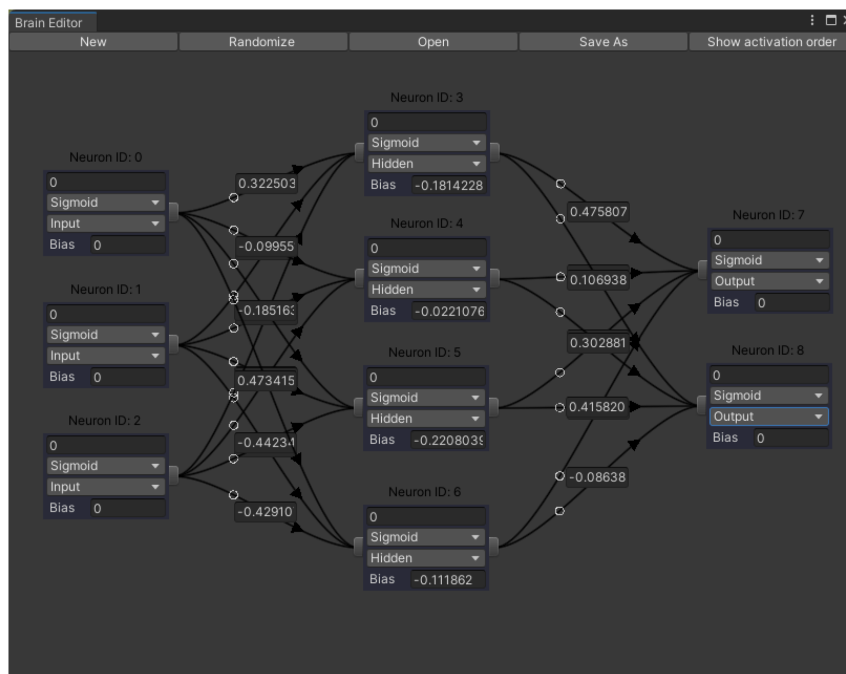


Рисунок 3.8 – Приклад згенерованої моделі повнозв’язного персепртрона

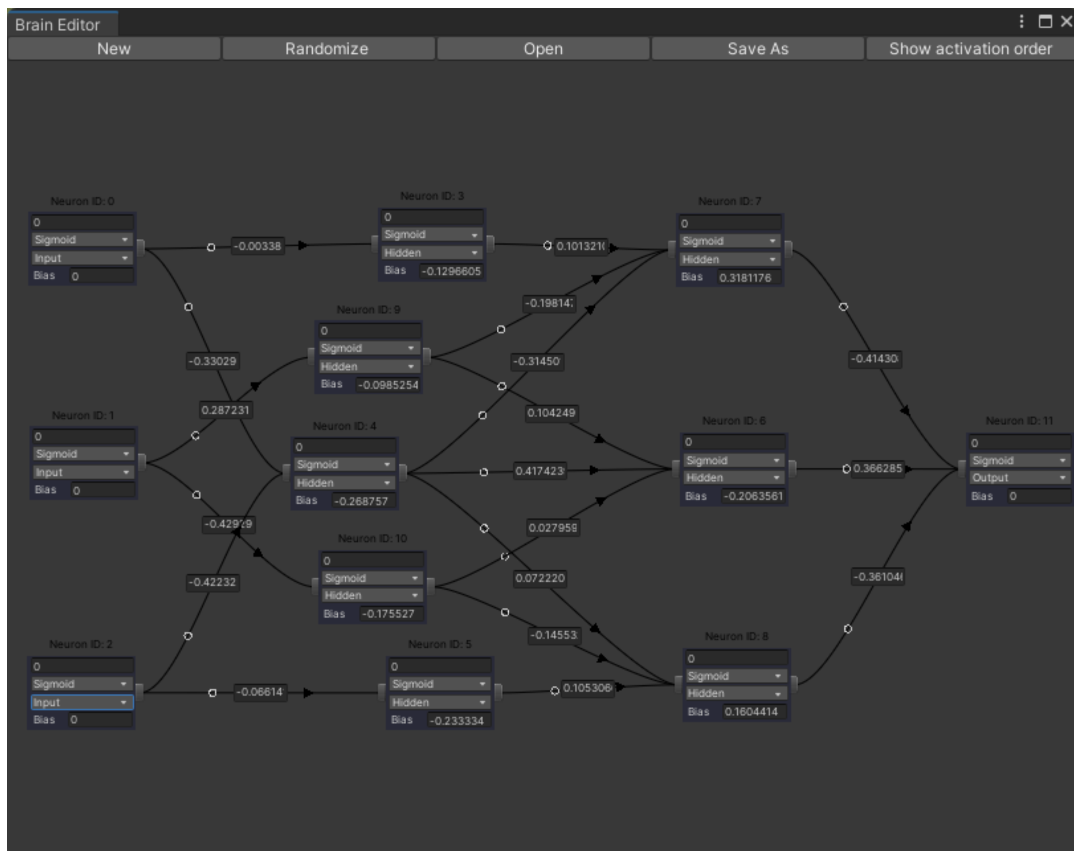


Рисунок 3.9 – Приклад створення довільної моделі нейромережі

Редактор підтримує наступні функції активації:

- sigmoid;
- relu;
- tanh;
- leaky relu.

Sigmoid функція активації – це нелинійна функція, яка широко використовується в нейронних мережах для згладжування вхідних значень та нормалізації ваг.

Sigmoid функція має значення від 0 до 1, при цьому значення близьке до 0 вказує на низький рівень активації нейрона, а значення близьке до 1 – на високий рівень активації. Графік та формула сигмоідальної функції наведені на рисунку 3.10.

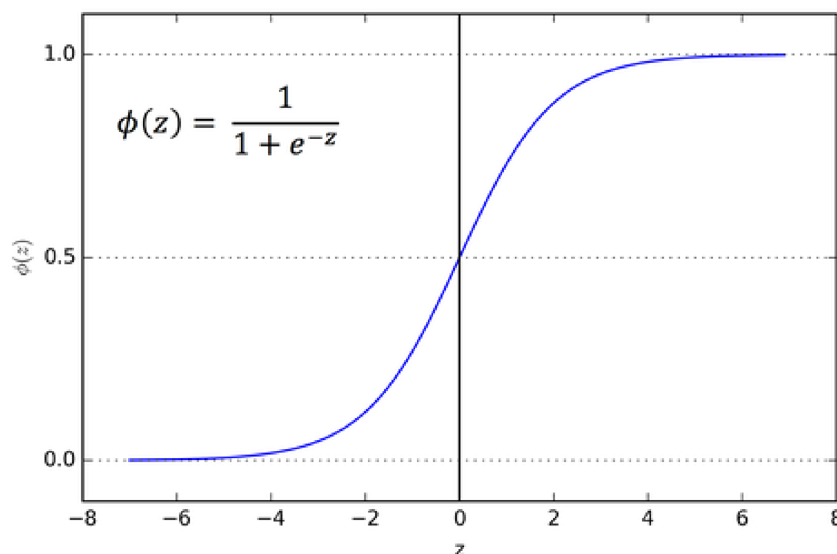


Рисунок 3.10 – Графік sigmoid функції

Основна перевага використання Sigmoid функції полягає в тому, що вона забезпечує згладження значень та дозволяє нормалізувати ваги в нейронній мережі. Крім того, Sigmoid функція є диференційованою, що дозволяє використовувати метод зворотного поширення помилок (backpropagation) для навчання нейронної мережі.

Однак, Sigmoid функція має деякі недоліки. Один з них полягає в тому, що при великих значеннях вхідних даних, функція насичується, тобто її похідна стає дуже мала, що може сповільнити процес навчання нейронної мережі. Крім того, Sigmoid функція не зберігає знак значення, тому можуть виникати проблеми з обробкою від'ємних значень. У загальному, Sigmoid функція активації є корисним інструментом для нормалізації та згладжування вхідних даних в нейронних мережах, і її використання залежить від потреби конкретного завдання [4].

Tanh функція активації – це нелінійна функція, яка широко використовується в нейронних мережах для згладжування вхідних значень та нормалізації ваг. Вона схожа до Sigmoid функції активації, але має більш симетричну форму та значення від -1 до 1 (рисунок 3.11).

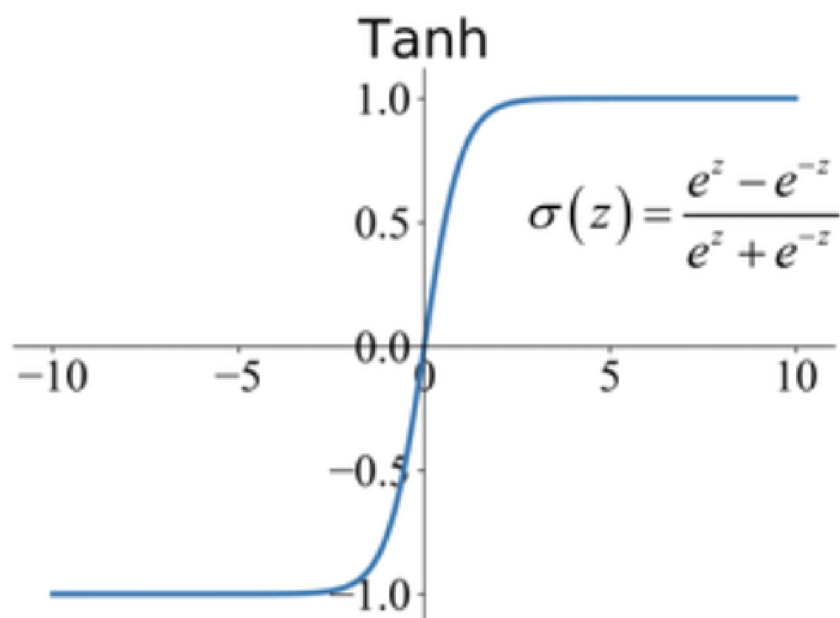


Рисунок 3.11 – Графік функції Tanh

Tanh функція має значення від -1 до 1, при цьому значення близьке до -1 вказує на негативний рівень активації нейрона, а значення близьке до 1 – на позитивний рівень активації.

Основна перевага використання Tanh функції полягає в тому, що вона забезпечує згладження значень та дозволяє нормалізувати ваги в нейронній мережі. Крім того, Tanh функція також є диференційованою, що дозволяє використовувати метод зворотного поширення помилок (backpropagation) для навчання нейронної мережі.

Недоліки Tanh функції активації аналогічні Sigmoid функції, і полягають в тому, що при великих значеннях вхідних даних, функція насичується, її похідна стає дуже мала, що може сповільнити процес навчання нейронної мережі. У загальному, Tanh функція активації є корисним інструментом для нормалізації та згладжування вхідних даних в нейронних мережах, і її використання залежить від потреби конкретного завдання.

ReLU (Rectified Linear Unit) – це нелінійна функція активації, яка використовується в нейронних мережах. Вона широко використовується в глибоких нейронних мережах, оскільки є дуже ефективною та дозволяє прискорити процес навчання (рисунок 3.12).

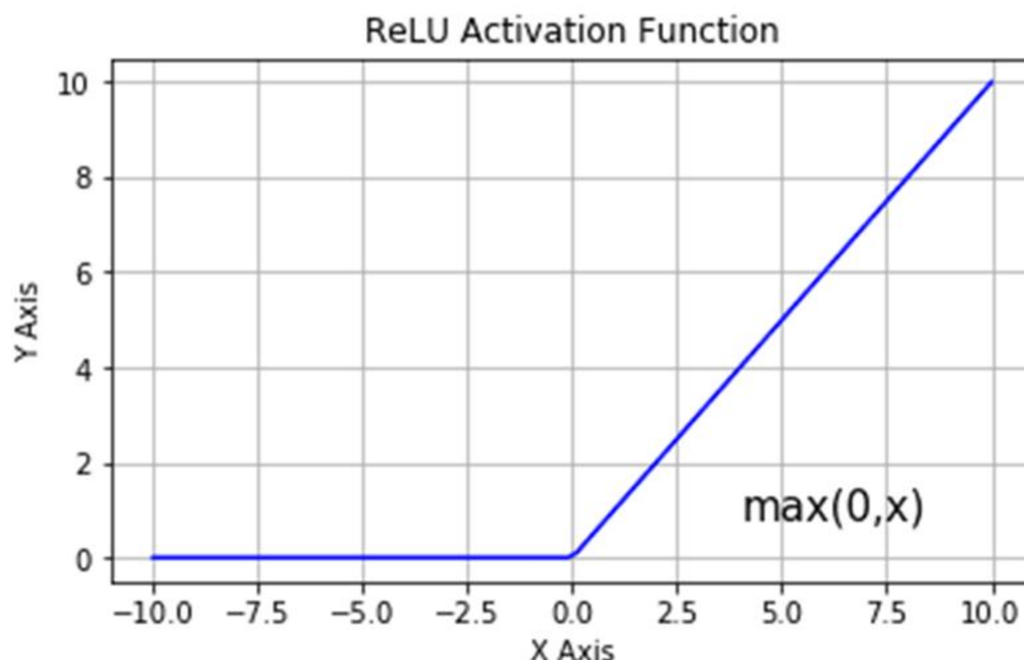


Рисунок 3.12 – Графік функції Relu

Переваги ReLU включають в себе простоту та швидкість обчислення, що дозволяє прискорити навчання нейронної мережі. Крім того, вона дозволяє уникнути проблеми вивантаження градієнту, яка може виникнути при використанні інших функцій активації.

Однак ReLU має деякі недоліки. Зокрема, вона може призвести до великої кількості «мертвих» нейронів, які неактивні і не вносять вагомий внесок у вихідний сигнал. Також, ReLU не може бути використана з безпрервними функціями втрат, такими як категоріальна крос-ентропія.

Leaky ReLU (Leaky Rectified Linear Unit) – це варіація на тему функції активації ReLU, яка вирішує деякі з її недоліків. Leaky ReLU була

запропонована з метою запобігання «мертвих» нейронів, які не активуються в процесі навчання.

Таким чином, якщо вхідний сигнал x менший за нуль, Leaky ReLU дозволяє пропускати деяку невелику частину вхідного сигналу, що дозволяє уникнути проблеми з «мертвими» нейронами (рисунок 3.13).

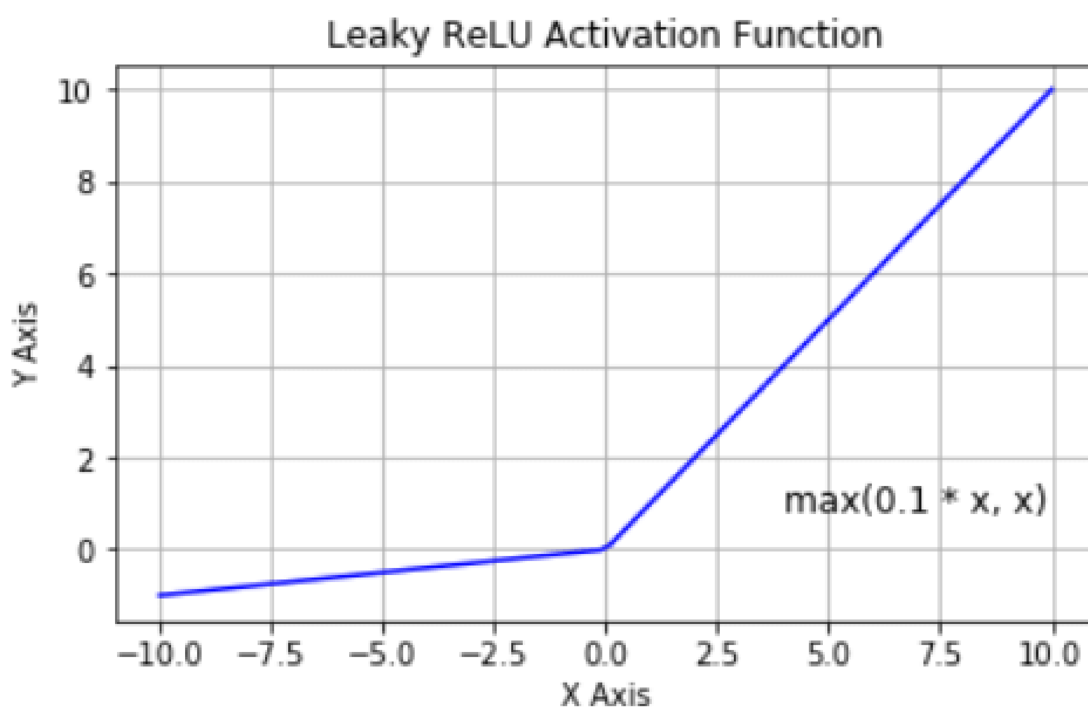


Рисунок 3.13 – Графік функції Leaky Relu

Переваги Leaky ReLU включають в себе те, що вона дозволяє уникнути «мертвих» нейронів, які можуть виникати при використанні ReLU. Крім того, Leaky ReLU має подібну швидкість обчислення, що і ReLU.

Недоліками Leaky ReLU є те, що вона може привести до проблеми з переактивністю нейронів, які можуть вести до нестабільної поведінки в процесі навчання. Також, як і у випадку з ReLU, Leaky ReLU не може бути використана з безперервними функціями втрат, такими як категоріальна крос-ентропія.

Окремою частиною редактора є модуль навчання нейромережі. Модуль підтримує наступні типи навчання:

- генетичні алгорими;
- зворотне поширення помилки.

Можна поєднати метод зворотного поширення помилки з генетичними алгоритмами, що дозволить зменшити кількість епох навчання та покращити точність моделі.

Backpropagation використовується для тренування моделі та оновлення ваг на основі вихідних даних. Це дозволяє моделі «вчитися» з прикладів та здійснювати передбачення на нових даних. Після завершення тренування можна застосувати генетичний алгоритм для оптимізації гіперпараметрів моделі, таких як кількість шарів, кількість нейронів у кожному шарі, функції активації та швидкість навчання.

Генетичний алгоритм генерує нові комбінації гіперпараметрів на основі оцінки їх ефективності на наборі валідаційних даних. Отримані комбінації гіперпараметрів можуть бути подані для наступного раунду тренування моделі, який вже буде використовувати оптимальні гіперпараметри. Ця комбінація методів дозволяє покращити результати моделі, зменшити час навчання та збільшити точність передбачень. Однак, важливо відзначити, що така оптимізація може вимагати значно більше обчислювальних ресурсів, особливо при великому обсязі даних, тому необхідно бути обережними при їх застосуванні.

Модуль навігації (рисунок 3.14) віртуального агента спрямовує промені в різних напрямках: вперед, назад, вліво, вправо і два промені вперед під кутом 45 градусів. Агент сканує навколишнє середовище за допомогою 6 променів. Кожен випущений промінь дає нам 1 вхідний параметр, який передається в нейронну мережу. Цей параметр описує дистанцію від автомобіля до стінки гоночної траси.

Редактор нейромереж також дозволяє зберігати та завантажувати навчені моделі для подальшого використання, що дозволяє ефективно використовувати їх у своїх проектах.

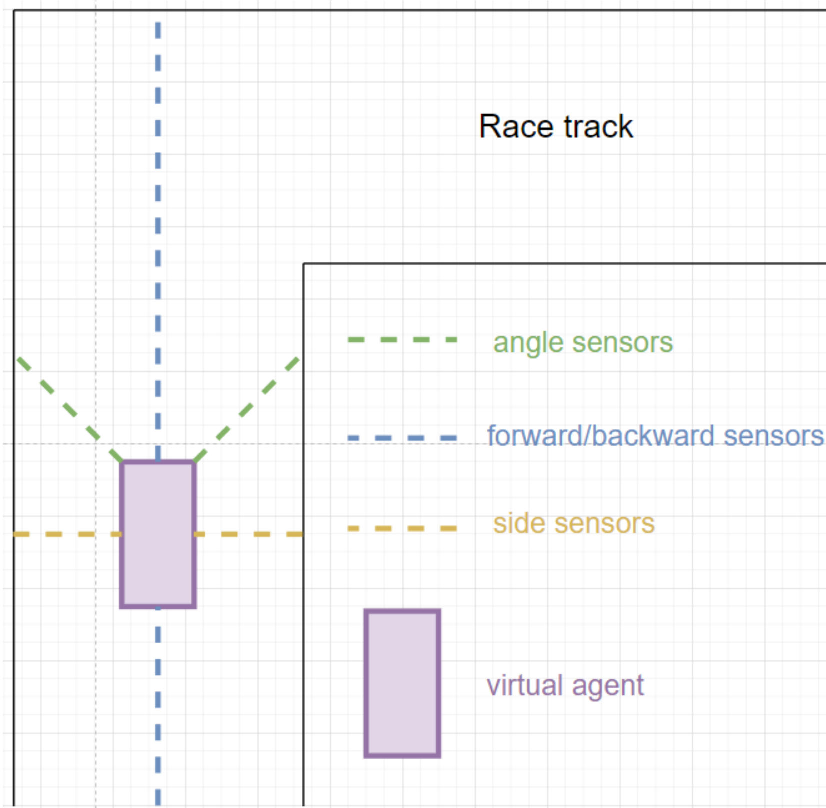


Рисунок 3.14 – Концептуальна схема принципу роботи модуля навігації

Середовище віртуальних агентів представляє собою гоночну трасу, яка має круті повороти (рисунок 3.15). Щоб ускладнити завдання для агентів трасса була змодельована таким чином, щоб напрамок повороту не повторювався. Кожен поворот на трасі має свою форму і напрямок. На основі відстаней до стінок гоночного треку нейронна мережа робить висновок. Висновок складається з 5-ти елементів.

Кожен з елементів являє собою окрему дію, яку може зробити агент:

- рухатись вперед;
- рухатись назад;

– повернути вліво;

– повернути вправо;

– активувати закис азоту (прискорення).

Критерій за яким ведеться відбір кращих екземплярів складається з:

– середньої швидкості агента на протязі його життя;

– загальної дальності його переміщення;

– кількості пройдених точок пропуску.

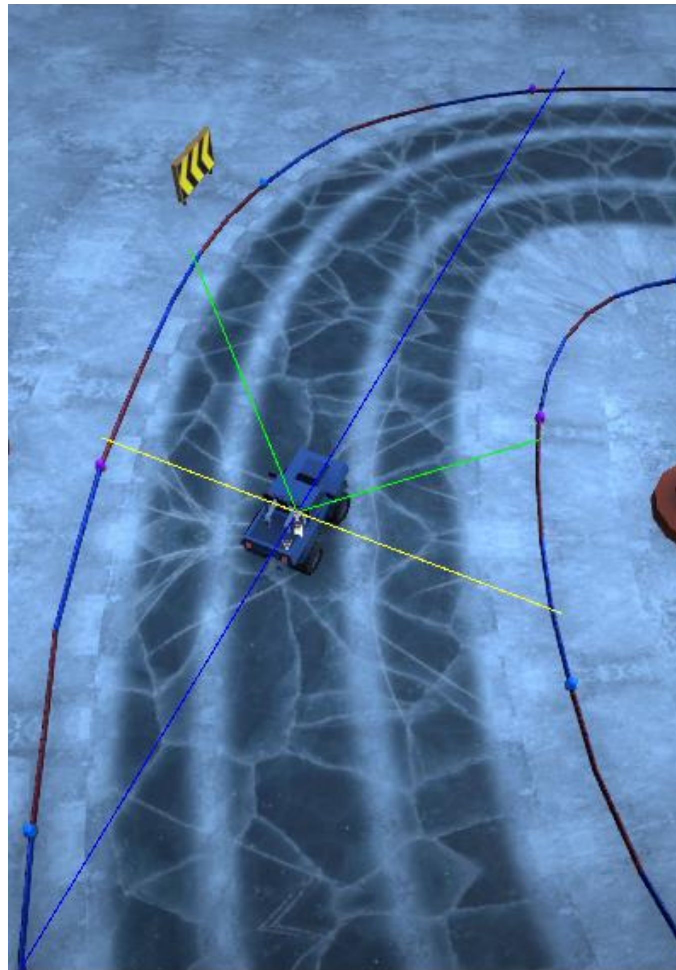


Рисунок 3.15 – Приклад сканування агентом навколишнього середовища

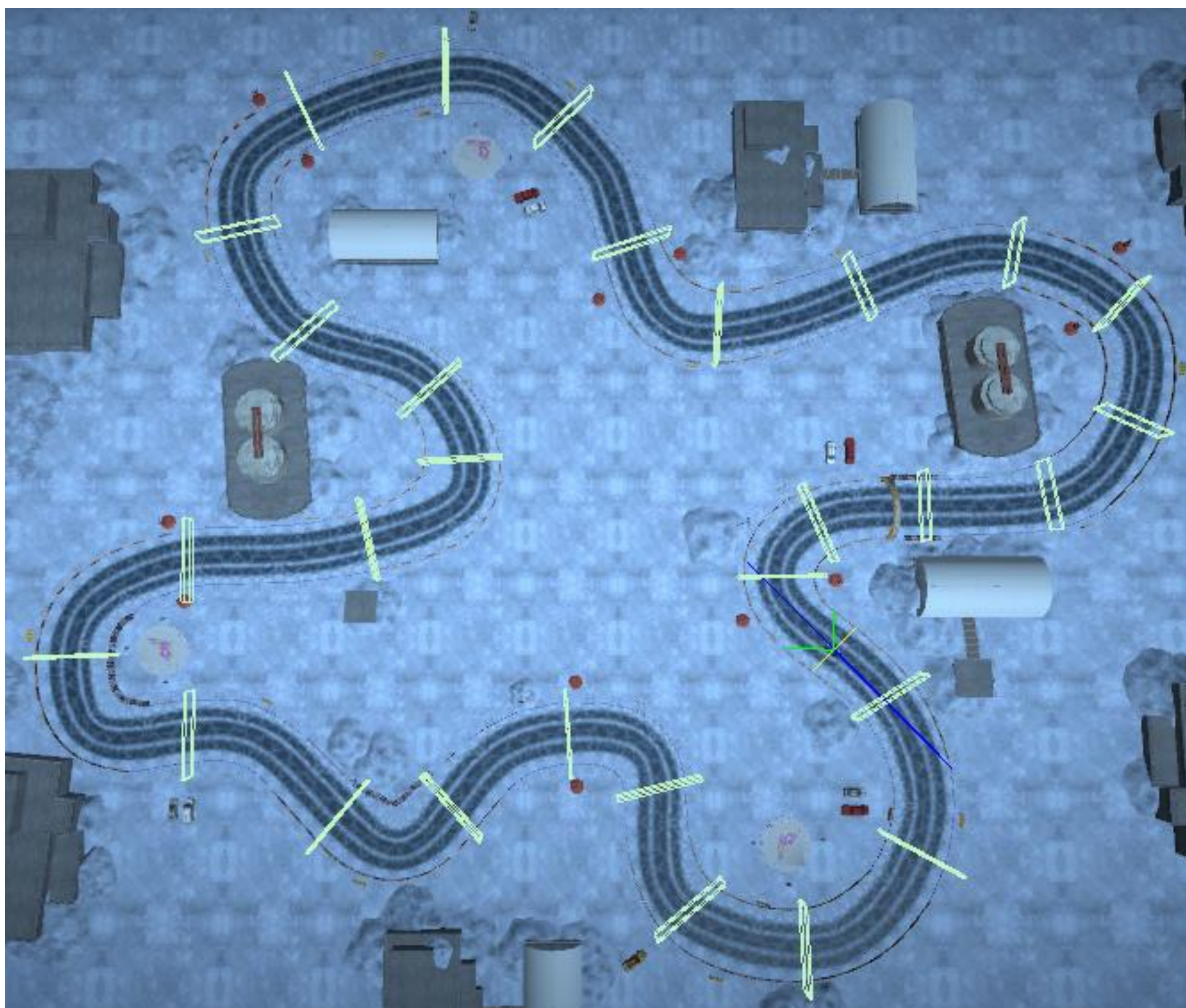


Рисунок 3.16 – Точки пропуску агентів за які нараховуються нагороди

В кожному наступному поколінні беруть участь наслідники 5-ти кращих екземплярів які були зафіксовані на протязі всього процесу навчання. Кожен екземпляр перед початком нової епохи піддається мутації. Алгоритм мутації обирає випадкові зв’язки між нейронами і змінює їх на випадкову величину.

ВИСНОВКИ

У даній кваліфікаційній роботі було досліджено теорію та практичні аспекти розробки ігрового штучного інтелекту з використанням генетичних алгоритмів. Було проведено аналіз існуючих підходів та алгоритмів розв'язання задач в галузі ігрової AI, визначено їх переваги та недоліки.

Було розроблено алгоритм генетичної еволюції, який дозволяє автоматично налаштовувати параметри гравця в ігровому середовищі. Реалізація алгоритму була проведена з використанням мов програмування C# та бібліотек написаних власноруч.

Для оцінки ефективності розробленого ігрового AI були проведені експерименти на різних ігрових середовищах з використанням різних наборів параметрів. Результати показали, що використання генетичного алгоритму дозволяє отримувати оптимальні налаштування параметрів гравця в ігровому середовищі, що призводить до покращення його результативності та ефективності.

Отже, на основі отриманих результатів можна зробити висновок про те, що розроблений ігровий AI з використанням генетичних алгоритмів є ефективним та перспективним напрямком у галузі розвитку ігрової технології та штучного інтелекту. Результати дослідження можуть бути використані для подальшого розвитку систем штучного інтелекту у різних галузях, включаючи ігрову індустрію.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Fogel, D. B., Owens, A. J., & Walsh, M. J. (2015). Artificial Intelligence through Search. Springer.

2. Rajagopalan, P., & Dasgupta, D. (2018). Genetic algorithm based optimization for game AI. Procedia Computer Science, 131, 599-606.

3. Smith-Miles, K. A., & Baatarjav, E. (2012). Optimizing opponent models for a racing game using genetic algorithms. IEEE Transactions on Computational Intelligence and AI in Games, 4(2),

4. Goldberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning. Addison-Wesley Professional.

5. Fogel, D. B. (2006). Evolutionary computation: toward a new philosophy of machine intelligence. John Wiley & Sons.

6. Yannakakis, G. N., & Togelius, J. (2018). Artificial intelligence and games. Springer.

7. Lucas, S. M. (2019). Artificial intelligence in games: A textbook. CRC Press.

8. Karafotias, G., Brownlee, A. E., & Bagnall, A. J. (2015). Evolutionary algorithms in games: A systematic review. IEEE Transactions on Computational Intelligence and AI in Games, 7(4), 321-344.

9. Challenges in benchmarking stream learning algorithms with real-world data / V. M. A. Souza et al. Data mining and knowledge discovery. 2020. Vol. 34, no. 6. P. 1805–1858. URL: https://doi.org/10.1007/s10618-020-00698-5 (date of access: 25.03.2022).

10. Domingos P., Hulten G. Mining high-speed data streams. The sixth ACM SIGKDD international conference, Boston, Massachusetts, United States, 20–23 August 2000. New York, New York, USA, 2000. URL: https://doi.org/10.1145/347090.347107 (date of access: 25.03.2022).

11. Discussion and review on evolving data streams and concept drift adapting / I. Khamassi et al. Evolving systems. 2016. Vol. 9, no. 1. P. 1–23. URL:

https://doi.org/10.1007/s12530-016-9168-2 (date of access: 25.03.2022).

12. Daniušis P., Vaitkus P. Neural network with matrix inputs. Informatica. 2008. Vol. 19, no. 4. P. 477–486. URL: https://doi.org/10.15388/informatica.2008.225 (date of access: 25.03.2022).

13. Mohamadian M., Afarideh H., Babapour F. New 2D matrix-based neural network for image processing applications. IAENG international journal of computer science. 2015. Vol. 42, no. 3. P. 265–274.

14. Gao J., Guo Y., Wang Z. Matrix neural networks. Advances in neural networks : Proceedings of the 14th International Symposium on Neural Networks (ISNN). Sapporo, Japan, 2017. P. 1–10.

15. Gao J., Guo J., Wang Z. Matrix neural networks. Advances in Neural Networks – ISNN 2017. Cham: Springer, 2017. P. 313-320.

16. Do K., Tran T., Venkatesh S. Matrix-centric neural networks. 2017. (Preprint. arXiv:1703.01454).

17. Matrix deep neural network and its rapid learning in data science tasks / I. Pliss et al. Advanced computer information technologies – ACIT 2018 : Proceedings of the conference. Ceske Budejovice, Czech Republic, 2018. P. 141–144.

18. Deep 2d-neural network and its fast learning / Y. Bodyanskiy et al. 2018 IEEE second international conference on data stream mining & processing (DSMP), Lviv, Ukraine, 21–25 August 2018. 2018. URL: https://doi.org/10.1109/dsmp.2018.8478578 (date of access: 25.03.2022).

19. Bodyanskiy Y., Boiko O., Pliss I., Kopaliani D., Volkova V. 2D-Deep Neural Network and Its Online Rapid Learning. Proceedings of the 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Metz, France, 2019, P. 304-307, 2019.

20. Kuntsevich V., Lychak M. Synthesis of optimal and adaptive control systems: the game approach. Naukova dumka. 1985.

21. Kuntsevych V. On a solving of the problem of two-dimensional discrete

filtration (synthesis of matrix filters). Automatica i telemekhanika. 1987. No. 6. P. 68–78.

22. Bodyanskiy Y., Pliss I. On a solving of the problem of a matrix object controlling under uncertainty conditions. Automatika i telemekhanika. 1990. No. 2. P. 175–178.

23. Bodyanskiy Y., Pliss I., Timofeev V. Discrete adaptive identification and extrapolation of two-dimensional fields. Pattern recognition and image analysis. 1995. Vol. 5, no. 3. P. 410–416.

24. Wolpert D. H. Stacked generalization. Neural networks. 1992. Vol. 5, no. 2. P. 241–259. URL: https://doi.org/10.1016/s0893-6080(05)80023-1 (date of access: 25.03.2022).

25. Scherer A. Neuronale Netze: Grundlagen und Anwendungen. Vieweg+Teubner Verlag, 2012. 260 p.

26. Empirical evaluation of rectified activations in convolutional network / B. Xu et al. 2015. (Preprint. arXiv:1505.00853v2).

27. Huang C. ReLU networks are universal approximators via piecewise linear or constant functions. Neural computation. 2020. Vol. 32, no. 11. P. 2249–2278. URL: https://doi.org/10.1162/neco_a_01316 (date of access: 25.03.2022).

28. Otto P., Bodyanskiy Y., Kolodyazhniy V. A new learning algorithm for a forecasting neuro-fuzzy network. Integrated Computer-Aided Engineering. 2003. Vol. 10, no. 4. P. 399–409. URL: https://doi.org/10.3233/ica-2003-10409 (date of access: 25.03.2022).

29. Bodyanskiy Y., Kolodyazhniy V., Stephan A. An adaptive learning algorithm for a neuro-fuzzy network. Computational intelligence. theory and applications. Berlin, Heidelberg, 2001. P. 68–75. URL: https://doi.org/10.1007/3-540-45493-4_11 (date of access: 25.03.2022).

ДОДАТОК А

Код програми

```
public class Weight
   {
      public int inputNeuronID;
      public int outputNeuronID;
      public double data;

      public Neuron inputNeuron { get; private set; }
      public Neuron outputNeuron { get; private set; }

      public Weight(int inputNeuronID, int outputNeuronID, float minRange,
float maxRange)
      {
         this.inputNeuronID = inputNeuronID;
         this.outputNeuronID = outputNeuronID;
         data = UnityEngine.Random.Range(minRange, maxRange);
      }

      public void AttachNeurons(Neuron inputNeuron, Neuron outputNeuron)
      {
         this.inputNeuron = inputNeuron;
         this.outputNeuron = outputNeuron;
      }

      public Weight()
      {

      }

      public Weight Copy()
      {
         return new Weight()
         {
            inputNeuronID = inputNeuronID,
            data = data,
            outputNeuronID = outputNeuronID
         };
      }
   }
```

```
public class Neuron
   {
       public event Action<Neuron, NeuronType, NeuronType>
onChangedNeuronType;

       public int id;
       public double data;
       public double delta;
       public float bias;
       public NeuronType neuronType;
       public ActivationType activationType;
       public List<Weight> inputWeights;
       public List<Weight> outputWeights;
       public Vector2 position;

       public Neuron(int id, float minRange, float maxRange, Vector2 position)
       {
          this.id = id;
          data = default;
          bias = UnityEngine.Random.Range(minRange, maxRange);
          inputWeights = new List<Weight>();
          outputWeights = new List<Weight>();
          this.position = position;
          neuronType = NeuronType.Hidden;
          activationType = ActivationType.Sigmoid;
       }

       public Neuron()
       {

       }

       public Neuron Copy()
       {
          var weights = new List<Weight>();

          for (var i = 0; i < inputWeights.Count; i++)
          {
             weights.Add(inputWeights[i].Copy());
          }

          return new Neuron()
          {
             id = id,
```

```
                activationType = activationType,
                bias = bias,
                data = data,
                inputWeights = weights,
                neuronType = neuronType,
                onChangedNeuronType = onChangedNeuronType,
                position = position
            };
        }

        public void AddInputWeight(Weight weight)
        {
            inputWeights.Add(weight);
        }

        public void RemoveInputWeight(Weight weight)
        {
            inputWeights.Remove(weight);
        }

        public void AddOutputWeight(Weight weight)
        {
            outputWeights.Add(weight);
        }

        public void RemoveOutputWeight(Weight weight)
        {
            outputWeights.Remove(weight);
        }

        public void SetActivationType(ActivationType activationType)
        {
            this.activationType = activationType;
        }

        public void ChangeNeuronType(NeuronType newNeuronType)
        {
            var prevNeuronType = neuronType;
            neuronType = newNeuronType;
            onChangedNeuronType?.Invoke(this, prevNeuronType,
newNeuronType);
        }

        public void Activate(BrainController controller)
        {
```

```
            var weightedSum = inputWeights.Sum(w => w.data *
controller.allNeurons[w.inputNeuronID].data);
            var activatedValue = Activation.Instance.Apply(activationType,
weightedSum + bias);
            data = activatedValue;
        }

        public double Derivative()
        {
            return Activation.Instance.Derivative(activationType, data);
        }



public class BrainController
    {
        public NeuralNetworkData Data => data;
        private NeuralNetworkData data;
        public Dictionary<int, Neuron> allNeurons { get; }

        public BrainController(string brainPath, bool randomize)
        {
            data = Serializer.ReadFromJson(brainPath);
            allNeurons = new Dictionary<int, Neuron>();
            foreach (var neuron in data.inputNeurons)
            {
                allNeurons.Add(neuron.id, neuron);
            }
            foreach (var neuron in data.hiddenNeurons)
            {
                allNeurons.Add(neuron.id, neuron);
            }
            foreach (var neuron in data.outputNeurons)
            {
                allNeurons.Add(neuron.id, neuron);
            }

            if (randomize)
            {
                var hiddenNeurons = data.hiddenNeurons;
                foreach (var neuron in hiddenNeurons)
                {
                    neuron.bias += UnityEngine.Random.Range(-1f, 1f);
                    foreach (var weight in neuron.inputWeights)
                    {
```

```
                weight.data += UnityEngine.Random.Range(-1f, 1f);
            }
        }
    }
}

private BrainController(NeuralNetworkData data)
{
    this.data = data;
    allNeurons = new Dictionary<int, Neuron>();
    foreach (var neuron in data.inputNeurons)
    {
        allNeurons.Add(neuron.id, neuron);
    }
    foreach (var neuron in data.hiddenNeurons)
    {
        allNeurons.Add(neuron.id, neuron);
    }
    foreach (var neuron in data.outputNeurons)
    {
        allNeurons.Add(neuron.id, neuron);
    }
}

public BrainController Copy()
{
    var nnd = new NeuralNetworkData()
    {
        fitness = data.fitness,
        nextID = data.nextID
    };

    foreach (var neuron in data.inputNeurons)
    {
        nnd.inputNeurons.Add(neuron.Copy());
    }

    foreach (var neuron in data.hiddenNeurons)
    {
        nnd.hiddenNeurons.Add(neuron.Copy());
    }

    foreach (var neuron in data.outputNeurons)
    {
        nnd.outputNeurons.Add(neuron.Copy());
```

```
            }

            var bc = new BrainController(nnd);
            bc.Data.fitness = data.fitness;

            return bc;
        }

        public double[] FeedForward(float[] input)
        {
            if (input.Length > data.inputNeurons.Count)
                throw new Exception(
                    "Discrepancy between the number of signals given and the number
of neural network inputs");

            for (var i = 0; i < data.inputNeurons.Count; i++)
            {
                if (i < input.Length)
                {
                    data.inputNeurons[i].data = input[i];
                }
                else
                {
                    data.inputNeurons[i].data = default;
                    throw new Exception(
                        "The input neuron is assigned a default value because the number
of signals given is less than the number of inputs for the neural network");
                }
            }

            for (var i = 0; i < data.hiddenNeurons.Count; i++)
            {
                data.hiddenNeurons[i].Activate(this);
            }

            for (var i = 0; i < data.outputNeurons.Count; i++)
            {
                data.outputNeurons[i].Activate(this);
            }

            return data.outputNeurons.Select(p => p.data).ToArray();
        }

        public void Mutate(int chance, int chanceThreshold, float mutationMinVal,
float mutationMaxVal, bool mutateOutputBiases)
```

```
        {
            for (var n = 0; n < data.hiddenNeurons.Count; n++)
            {
                // mutate hidden biases
                var randValue2 = UnityEngine.Random.Range(0, chanceThreshold);
                if (randValue2 <= chance) data.hiddenNeurons[n].bias +=
UnityEngine.Random.Range(mutationMinVal, mutationMaxVal);
                // mutate hidden weights
                for (var w = 0; w < data.hiddenNeurons[n].inputWeights.Count; w++)
                {
                    var randValue = UnityEngine.Random.Range(0, chanceThreshold);
                    if (randValue <= chance)
                    {
                        data.hiddenNeurons[n].inputWeights[w].data +=
UnityEngine.Random.Range(mutationMinVal, mutationMaxVal);
                    }
                }
            }

            for (var n = 0; n < data.outputNeurons.Count; n++)
            {
                if (mutateOutputBiases)
                    // mutate output biases
                {
                    var randValue = UnityEngine.Random.Range(0, chanceThreshold);
                    if (randValue <= chance) data.hiddenNeurons[n].bias +=
UnityEngine.Random.Range(mutationMinVal, mutationMaxVal);
                }
                // mutate output weights
                for (var w = 0; w < data.outputNeurons[n].inputWeights.Count; w++)
                {
                    var randValue = UnityEngine.Random.Range(0, chanceThreshold);
                    if (randValue <= chance)
                    {
                        data.outputNeurons[n].inputWeights[w].data +=
UnityEngine.Random.Range(mutationMinVal, mutationMaxVal);
                    }
                }
            }
        }

        public void BackPropagation(float[] input, float[] expected, float
learningRate = 0.1f)
        {
            var output = FeedForward(input);
```

```
        for (var i = 0; i < output.Length; i++)
        {
            data.outputNeurons[i].delta = Math.Pow(output[i] – expected[i], 2); //
to math pow, maybe
        }

        // Change to weights
        for (var i = 0; i < data.outputNeurons.Count; i++)
        {
            var currentNeuron = data.outputNeurons[i];
            for (var w = 0; w < data.outputNeurons[i].inputWeights.Count; w++)
            {
                var currentWeight = currentNeuron.inputWeights[w];

                var deltaWeight = currentNeuron.delta * currentWeight.data *
currentNeuron.Derivative() * learningRate;
                currentWeight.data += deltaWeight;
            }
        }

        for (var n = data.hiddenNeurons.Count – 1; n >= 0; n--)
        {
            var neuron = data.hiddenNeurons[n];
            double diff = 0;
            for (var w = 0; w < neuron.outputWeights.Count; w++)
            {
                var weight = neuron.outputWeights[w];
                diff += Math.Pow(allNeurons[weight.outputNeuronID].data –
allNeurons[weight.outputNeuronID].delta, 2);
            }
            diff /= neuron.outputWeights.Count;
            neuron.delta = diff;
        }

        // Change to weights
        for (var i = data.hiddenNeurons.Count – 1; i >= 0; i--)
        {
            var currentNeuron = data.hiddenNeurons[i];
            for (var w = 0; w < data.hiddenNeurons[i].inputWeights.Count; w++)
            {
                var currentWeight = currentNeuron.inputWeights[w];

                var deltaWeight = currentNeuron.delta * currentWeight.data *
currentNeuron.Derivative() * learningRate;
```

```
                currentWeight.data += deltaWeight;
            }
        }
    }

    public void Save(string assetName)
    {
        Serializer.WriteToJson("Assets/" + assetName + ".json", data, false);
    }
}
```

```
public class NeuralNetworkData
{
    public float fitness;
    public List<Neuron> inputNeurons;
    public List<Neuron> hiddenNeurons;
    public List<Neuron> outputNeurons;

    public int nextID;

    public NeuralNetworkData()
    {
        hiddenNeurons = new List<Neuron>();
        inputNeurons = new List<Neuron>();
        outputNeurons = new List<Neuron>();
    }

    public NeuralNetworkData(int inputs,
        int outputs,
        int[] hiddenLayers,
        ActivationType outputActivationType,
        ActivationType[] hiddenActivationTypes,
        float neuronRandomRange,
        float weightRandomRange)
    {
        hiddenNeurons = new List<Neuron>();
        inputNeurons = new List<Neuron>();
        outputNeurons = new List<Neuron>();

        var horizontalSpaceLength = 200;
        var verticalSpaceLength = 100;
        var startNodesPosition = Vector2.zero;
```

```
for (var i = 0; i < inputs; i++)
{
   var newNeuron = new Neuron(nextID, 0, 0, new Vector2 (
startNodesPosition.x, startNodesPosition.y + i * verticalSpaceLength));
   newNeuron.neuronType = NeuronType.Input;
   newNeuron.onChangedNeuronType += OnChangedNeuronType;
   inputNeurons.Add(newNeuron);
   nextID++;
}

var previousHiddenLayer = new List<int>();

for (var i = 0; i < hiddenLayers.Length; i++)
{
   var nextHiddenLayer = new List<int>();

   for (var j = 0; j < hiddenLayers[i]; j++)
   {
      var newNeuron = new Neuron(nextID, -0.5f, 0.5f, new Vector2 (
startNodesPosition.x + (i + 1) * horizontalSpaceLength, startNodesPosition.y + j
* verticalSpaceLength));
      newNeuron.onChangedNeuronType += OnChangedNeuronType;
      hiddenNeurons.Add(newNeuron);
      nextID++;
      if (i == 0)
      {
         for (var w = 0; w < inputNeurons.Count; w++)
         {
            var weight = new Weight(inputNeurons[w].id, newNeuron.id, -
0.5f, 0.5f);
            newNeuron.AddInputWeight(weight);
            var neuron = inputNeurons.Find(n => n.id ==
weight.inputNeuronID);
            neuron.AddOutputWeight(weight);
         }
      }
      else
      {
         for (var w = 0; w < previousHiddenLayer.Count; w++)
         {
            var weight = new Weight(previousHiddenLayer[w],
newNeuron.id, -0.5f, 0.5f);
            newNeuron.AddInputWeight(weight);
```

```
                    var neuron = hiddenNeurons.Find(n => n.id ==
weight.inputNeuronID);
                    neuron.AddOutputWeight(weight);
                }
            }
            nextHiddenLayer.Add(newNeuron.id);
        }

        previousHiddenLayer = nextHiddenLayer;

    }

    for (var i = 0; i < outputs; i++)
    {
        var newNeuron = new Neuron(nextID, -0.5f, 0.5f, new Vector2 (
startNodesPosition.x +
                                    (hiddenLayers.Length + 1) *
horizontalSpaceLength, startNodesPosition.y + i * verticalSpaceLength));
        newNeuron.onChangedNeuronType += OnChangedNeuronType;
        outputNeurons.Add(newNeuron);
        nextID++;
        for (var w = 0; w < previousHiddenLayer.Count; w++)
        {
            var weight = new Weight(previousHiddenLayer[w], newNeuron.id,
-0.5f, 0.5f);
            newNeuron.AddInputWeight(weight);
            var neuron = hiddenNeurons.Find(n => n.id ==
weight.inputNeuronID);
            neuron.AddOutputWeight(weight);
        }
        newNeuron.neuronType = NeuronType.Output;
    }
}

public Neuron AddNeuron(Vector2 position)
{
    var newNeuron = new Neuron(nextID, -0.5f, 0.5f, position);
    newNeuron.onChangedNeuronType += OnChangedNeuronType;
    hiddenNeurons.Add(newNeuron);
    nextID++;
    return newNeuron;
}

public Weight AddWeight(Neuron fromNeuron, Neuron toNeuron)
{
```

```
            var newWeight = new Weight(fromNeuron.id, toNeuron.id, -0.5f, 0.5f);
            newWeight.AttachNeurons(fromNeuron, toNeuron);
            toNeuron.AddInputWeight(newWeight);
            fromNeuron.AddOutputWeight(newWeight);
            return newWeight;
        }

        public void RemoveNeuron(Neuron neuron)
        {
            if (hiddenNeurons.Contains(neuron)) hiddenNeurons.Remove(neuron);
            if (inputNeurons.Contains(neuron)) inputNeurons.Remove(neuron);
            if (outputNeurons.Contains(neuron)) outputNeurons.Remove(neuron);
        }

        public void RemoveWeight(Weight weight)
        {

            weight.outputNeuron.RemoveInputWeight(weight);
        }

        private void OnChangedNeuronType(Neuron neuron, NeuronType
prevNeuronType, NeuronType newNeuronType)
        {
            if (prevNeuronType == NeuronType.Hidden && newNeuronType ==
NeuronType.Input)
            {
                neuron.bias = 0;
                inputNeurons.Add(neuron);
                hiddenNeurons.Remove(neuron);
            }

            if (prevNeuronType == NeuronType.Hidden && newNeuronType ==
NeuronType.Output)
            {
                neuron.bias = 0;
                outputNeurons.Add(neuron);
                hiddenNeurons.Remove(neuron);
            }

            if (prevNeuronType == NeuronType.Input)
            {
                inputNeurons.Remove(neuron);

                switch (newNeuronType)
                {
```

```
                case NeuronType.Output:
                    neuron.bias = 0;
                    outputNeurons.Add(neuron);
                    break;
                case NeuronType.Hidden:
                    neuron.bias = UnityEngine.Random.Range(-0.5f, 0.5f);
                    hiddenNeurons.Add(neuron);
                    break;
            }
        }

        if (prevNeuronType == NeuronType.Output)
        {
            outputNeurons.Remove(neuron);
            switch (newNeuronType)
            {
                case NeuronType.Input:
                    neuron.bias = 0;
                    inputNeurons.Add(neuron);
                    break;
                case NeuronType.Hidden:
                    neuron.bias = UnityEngine.Random.Range(-0.5f, 0.5f);
                    hiddenNeurons.Add(neuron);
                    break;
            }
        }
    }
}



public static class Serializer
{
    private static void Sort(NeuralNetworkData nnd)
    {
        var sorted = nnd.hiddenNeurons.OrderByDependers(n =>
n.inputWeights.Select(w => w.inputNeuron).ToList()).ToList();
        sorted.Reverse();
        nnd.hiddenNeurons = sorted;
    }

    public static void WriteToJson(string path, NeuralNetworkData
neuralNetworkData, bool needToSort)
    {
        if (needToSort) Sort(neuralNetworkData);
```

```
            var json = JsonUtility.ToJson(neuralNetworkData);
            File.WriteAllText(path, json);
        }

        public static NeuralNetworkData ReadFromJson(string path)
        {
            var jsonString = File.ReadAllText(path);
            var networkData =
JsonUtility.FromJson<NeuralNetworkData>(jsonString);
            return networkData;
        }
    }


public class LearnManager : MonoBehaviour
    {
        public LearnDataSet LearnDataSet => learnDataSet;
        [SerializeField] private LearnDataSet learnDataSet;
        [SerializeField] private string learnedAssetName;


        [Button("Learn")]
        private void Learn()
        {
            var brain = new BrainController("Assets/Test.json", false);
            Debug.Log("Learning started");

            for (var i = 0; i < learnDataSet.data.Count; i++)
            {
                brain.BackPropagation(learnDataSet.data[i].inputParams,
learnDataSet.data[i].expectedInputs);
            }
            brain.Save(learnedAssetName);

            Debug.Log("Learning finished");
        }
    }

public abstract class NeuralBehaviour : MonoBehaviour
    {
        [SerializeField] private LayerMask layerMask;
        public event Action<NeuralBehaviour> onFailed;
        public BrainController brain { get; protected set; }
```

```
        public void SetBrain(BrainController brain)
        {
            this.brain = brain;
        }

        public void AddFitness(float fitnessValue)
        {
            brain.Data.fitness += fitnessValue;
        }

        public abstract void UseBrain();

        protected void OnFailed()
        {
            onFailed?.Invoke(this);
        }
    }



public class Sigmoid : BaseActivationController
    {
        public Sigmoid(ActivationModel model) : base(model)
        {
        }

        public override double Apply(double weightedSum)
        {
            if (weightedSum > 38.53f) return 1.0f;
            if (weightedSum < -38.53f) return 0.0f;

            var k = (float)Math.Exp(weightedSum);
            return k / (1.0f + k);
        }

        public override double Derivative(double x)
        {
            return x * (1 – x);
        }
    }
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	75 с.
	Інші документи	
2.	Презентаційні матеріали	18 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Гриб Д. В.			Ігровий штучний інтелект з використанням генетичних алгоритмів	Шифр групи	Код напр./спец.
Перевірив		Сніжко Д. В.				СШІм-21-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					