МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет

імені Петра Могили

Факультет комп'ютерних наук

Кафедра комп'ютерна інженерія

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри комп'ютерної інженерії_____ І. М. Журавська
підпис

«17» лютого 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

ІМПЛЕМЕНТАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ СИМУЛЯЦІЇ БІОПОПУЛЯЦІЇ НА ОСНОВІ НЕЙРОННОЇ МЕРЕЖІ ТА ІГРОВОГО РУШІЯ У БАГАТОЯДЕРНІ ПРИСТРОЇ

Спеціальність «Комп'ютерна інженерія»

123 –КРМ – 605.21710902

Студент _____ Є. С.Беззуб
підпис
«17» лютого 2023 р.

Керівник канд. тех. наук, доцент _____ А. П. Бойко
підпис
«17» лютого 2023 р.

Миколаїв – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра комп'ютерної інженерії

ЗАТВЕРДЖУЮ

Зав. кафедри Журавська І. М

«4» листопада 2022 р.

ЗАВДАННЯ

на виконання кваліфікаційної магістерської роботи

Видано студенту групи 605 факультету комп'ютерних наук

Беззубу Євгенію Сергійовичу

(прізвище, ім'я, по батькові студента)

1. Тема кваліфікаційної роботи

Імплементація програмного забезпечення для симуляції біопопуляції на основі нейронної мережі та ігрового рушія у багатоядерні пристрої

Затверджена наказом по ЧНУ від «3» листопада 2023 р. № 201

2. Строк представлення кваліфікаційної роботи «27» лютого 2023 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні

Очікуваним результатом є вбудована симуляція популяції в багатоядерні пристрої.

4. Перелік питань, що підлягають розробці

Аналіз предметної сфери, дослідження аналогів; моделювання та проєктування; розробка нейроної мережі; розробка застосунку – симулятору та імплементація в багатоядерні пристрої; тестування

5. Перелік графічних матеріалів

Слайди презентацій

6. Завдання до спеціальної частини

Оцінка умов праці на робочому місці

7. Консультанти:

Консультант	Кафедра (організація)	Частина роботи
Л. І.Григор'єва	Еколгія	Спеціальна частина з охорони праці

Керівник роботи __канд. тех. наук, доцент Бойко Анжела Петрівна

(посада, прізвище, ім'я, по батькові)

_____

(підпис)

Завдання прийнято до виконання. Здобувач вищої освіти

_Беззуб Євгеній Сергійович

(прізвище, ім'я, по батькові студента)

_____

(підпис)

Дата видачі завдання «4» листопада 2023 р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: Імплементація програмного забезпечення для симуляції біопопуляції на основі нейронної мережі та ігрового рушія у багатоядерні пристрої.

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розробка та затвердження завдання на виконання КРМ	21.07.2022	10.08.2022	Виконано
2.	Огляд літератури за темою роботи	12.08.2022	20.09.2022	Виконано
3.	Аналіз предметної області	22.10.2022	28.10.2022	Виконано
4.	Аналіз технології Нейронної мережі	28.10.2022	01.11.2022	Виконано
5.	Моделювання об’єкту та предмету дослідження	01.11.2022	05.11.2022	Виконано
6.	Розробка застосунку	06.11.2022	30.12.2022	Виконано
7.	Оптимізація проекту та навчання нейронної мережі	22.11.2022	05.01.2023	Виконано
8.	Тестування застосунку на різних пристроях	05.01.2023	02.02.2023	Виконано
9.	Розробка спеціальної частини з охорони праці	03.02.2023	05.02.2023	Виконано
10.	Аналіз отриманих результатів	05.02.2023	10.02.2023	Виконано
11.	Оформлення КРМ	12.08.2022	12.02.2023	Виконано
12.	Оформлення презентації	12.02.2023	13.02.2023	Виконано
13.	Рецензування	07.02.2023	07.02.2023	Виконано
14.	Попередній захист	15.02.2023	15.02.2023	Виконано
15.	Завершення оформлення КРМ та презентації	13.02.2023	17.02.2023	Виконано
16.	Відгук керівника КРМ	17.02.2023	17.02.2023	Виконано
17.	Захист кваліфікаційної роботи	27.02.2023	27.02.2023	Виконано

Розробив студент _____Беззуб Є. С._____ _____
(*прізвище, ім’я, по батькові)* *(підпис)*
«__» _____ 20__ р.

Керівник роботи _к.т.н., доцент Бойко Є. С._____ _____
(*посада, прізвище, ім’я, по батькові)* *(підпис)*
«__» _____ 20__ р.

АНОТАЦІЯ

до кваліфікаційної роботи магістра

«ІМПЛЕМЕНТАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ СИМУЛЯЦІЇ БІОПОПУЛЯЦІЇ НА ОСНОВІ НЕЙРОННОЇ МЕРЕЖІ ТА ІГРОВОГО РУШІЯ У БАГАТОЯДЕРНІ ПРИСТРОЇ»

Студент 605 гр.: Беззуб Євгеній Сергійович

Керівник: канд. техн. наук, доцент Бойко А.П

Кваліфікаційна робота викладена на ___сторінки, вона містить ___ розділи, ___ілюстрацій, ___ таблиці, ___ джерел в переліку посилань.

У магістерській роботі досліджується можливість створення симуляції популяції живих організмів за допомогою Unity3d - потужного інструменту для створення ігор та різних візуалізації в 2D та 3D просторі.

Створення програмного забезпечення, що дозволяє передбачити результати процесів життєдіяльності живих організмів у критичних ситуаціях, згенерувавши проблему в цифровому простору, та імплементація його в багатоядерні пристрої є **актуальною задачею.**

Мета: Покращення методів тестування різноманітних природніх та штучних явищ за рахунок створення програмного забезпечення з використанням нейронних мереж на базі платформи розробки Unity3D.

Об'єкт: процес імплементації програмного забезпечення у багатоядерні пристрої.

Предмет: симуляція біопопуляції на основі нейронної мережі та ігрового рушія.

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

– розглянути методи моделювання живих організмів, а також дослідити різні алгоритми та методи управління популяцією;

– дослідити принципи роботи нейронної мережі;

– проаналізувати існуючі застосунки та визначити їх основні переваги та недоліки;

– здійснити аналіз редакторів для розробки застосунків та обрати середовище для розробки застосунку;

– створити функціональну та абстрактну моделі системи;

– провести аналіз багатоядерних пристроїв, які зможуть бути використані у тестувані;

– створити симуляцію популяції на основі нейронної мережі використовуючи обрані методи та засоби розробки.

– тестування створенного застосунку.

Методи та засоби дослідження – для створення застосунку для симуляції різних життєвих ситуацій було обрано ігровий рушій Unity з використанням об’єктно-орієнтованої мови C#.

Практичне значення отриманих результатів даної роботи полягає у тому, що представлена модель може підвищити рівень безпеки людства. Проведення тестів глобальних проблем на неіснуючих істотах не несе ніяких втрат серед населення Землі. Саме завдяки технології заснованої на нейронних мережах, дає змогу симулювати поведінку людей наближено до реальної, і в свою чергу це допомже уникнути поганих наслідків в майбутньому

У першому розділі роботи проводиться системний аналіз обраної предметної області та, на його основі, формулюється постановка задачі та специфікація вимог до програмного забезпечення.

У другому розділі розглядається основна технологія розробки застосунку. В розділ входить опис різних типів нейронних мереж та генетичного алгоритма. Наведені переваги та недоліки симуляцій засновані на нейронній мережі.

У третьому розділі відбувається програмна реалізація та розробка застосунку

У четвертому розділі відбувається тестування апаратно – програмного комплексу різними методами.

Ключові слова: комп’ютерна симуляція, нейронна мережа, Unity.

ASTRACT

of the Master`s Thesis

" SOFTWARE IMPLEMENTATION FOR
NEURON-BASED BIOPOPULATION SIMULATIONS
NETWORK AND GAME ENGINE IN MULTI-CORE DEVICES "

Student of group 605: Bezzub Evgeniy Serhiyovych

Supervisor: Ph.D. tech. Sciences, Associate Professor Boyko A. P.

Thesis is laid out on ___ pages, it contains ___ sections, ___ illustrations, ___ tables, ___ sources in the list of links.

The master's thesis examines the possibility of creating a simulation of a population of living organisms using Unity3d - a powerful tool for creating games and various visualizations in 2D and 3D space.

The creation of software that allows predicting the results of life processes of living organisms in critical situations, generating a problem in the digital space, and its implementation in multi-core devices is an urgent task.

Purpose: Improvement of methods of testing various natural and artificial phenomena due to the creation of software using neural networks based on the Unity3D development platform.

Object: the process of implementing software in multi-core devices.

Subject: biopopulation simulation based on neural network and game engine.

To achieve the goal, the following tasks must be solved:

– consider methods of modeling living organisms, as well as research various algorithms and methods of population management;

– investigate the principles of neural network operation;

– analyze existing applications and determine their main advantages and disadvantages;

– analyze editors for application development and choose an environment for application development;

– create functional and abstract system models;

– analyze multi-core devices that can be used in testing;

– create a population simulation based on a neural network using the selected development methods and tools.

– testing of the created application.

Research methods and tools - to create an application for simulating various life situations, the Unity game engine using the object-oriented C# language was chosen.

The practical significance of the obtained results of this work is that the presented model can increase the level of security of mankind. Conducting tests of global problems on non-existent creatures does not cause any losses among the population of the Earth. Thanks to the technology based on neural networks, it is possible to simulate the behavior of people close to the real one, and in turn this will help to avoid bad consequences in the future

In the first section of the work, a systematic analysis of the selected subject area is carried out and, on its basis, the problem statement and the specification of software requirements are formulated.

The second chapter deals with the basic technology of application development. The chapter includes a description of different types of neural networks and the genetic algorithm. The following advantages and disadvantages of simulations are based on a neural network.

In the third section, software implementation and development of the application takes place

In the fourth section, the hardware and software complex is tested using various methods.

Keywords: computer simulation, neural network, Unity.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ЕОМ	–	Електронна обчислювальна машина;
НМ	–	Нейрона мережа;
ОС	–	операційна система;
ПК	–	персональний комп’ютер;
ШІ	–	штучний інтелект;
API	–	інтерфейс програмування застосунків (application programming interface);
FPS	–	Кадрів на секунду (Frame Per Second).
ML	–	Машинне навчання (Machine learning);
UI	–	інтерфейс користувача (user interface);
UML	–	уніфікована мова моделювання (unified modeling language);
WEBGL	–	кроссплатформенний API для 3D-графіки в браузері;

ВСТУП

Весь час свого існування людство стикається з проблемами глобального масштабу. Особливо це відчутно в останнє століття: було пережито ядерні катаклізми [15], смертоносні віруси та кровопролитні війни. Для розуміння поведінки живих істот є сенс симулювати процес життєдіяльності у різноманітних критичних ситуаціях.

Симуляція життя живих організмів – це ключовий аспект дослідження біологічних процесів, який може допомогти у розумінні різних аспектів життєдіяльності організмів. Для повного розуміння поведінки живих істот, без шкоди навколишньому середовищу, можливо лише за умовами, які були створені в цифровому всесвіті.

У магістерській роботі досліджується можливість створення симуляції популяції живих організмів за допомогою Unity3d - потужного інструменту для створення ігор та різних візуалізації в 2D та 3D просторі.

Створення програмного забезпечення, що дозволяє передбачити результати процесів життєдіяльності живих організмів у критичних ситуаціях, згенерувавши проблему в цифровому простору, та імплементація його в багатоядерні пристрої є **актуальною задачею.**

Мета: Покращення методів тестування різноманітних природніх та штучних явищ за рахунок створення програмного забезпечення з використанням нейронних мереж на базі платформи розробки Unity3D.

Об’єкт: процес імплементації програмного забезпечення у багатоядерні пристрої.

Предмет: симуляція біопопуляції на основі нейронної мережі та ігрового рушія.

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

– розглянути методи моделювання живих організмів, а також дослідити різні алгоритми та методи управління популяцією;

– дослідити принципи роботи нейронної мережі;

– проаналізувати існуючі застосунки та визначити їх основні переваги та недоліки;

– здійснити аналіз редакторів для розробки застосунків та обрати середовище для розробки застосунку;

– створити функціональну та абстрактну моделі системи;

– провести аналіз багатоядерних пристроїв, які зможуть бути використані у тестувані;

– створити симуляцію популяції на основі нейронної мережі використовуючи обрані методи та засоби розробки.

– тестування створенного застосунку.

Методи та засоби дослідження – для створення застосунку для симуляції різних життєвих ситуацій було обрано ігровий рушій Unity з використанням об’єктно-орієнтованої мови C#.

Практичне значення отриманих результатів даної роботи полягає у тому, що представлена модель може підвищити рівень безпеки людства. Проведення тестів глобальних проблем на неіснуючих істотах не несе ніяких втрат серед населення Землі. Саме завдяки технології заснованої на нейронних мережах, дає змогу симулювати поведінку людей наближено до реальної, і в свою чергу це допомже уникнути поганих наслідків в майбутньому.

Результати роботи доповідалися на Всеукраїнській науково-практичній конференції молодих вчених, аспірантів і студентів «Інформаційні технології та інженерія» [1].

1 АНАЛІЗ ПРЕДМЕТНОЇ СФЕРИ, ОБ'ЄКТУ ТА ПРЕДМЕТУ ДОСЛІДЖЕННЯ

1.1 Нейронна мережа

Нейронні мережі – це математичні моделі, які можуть імітувати роботу нейронів у мозку людини. Вони складаються з безлічі нейронів, кожен з яких може приймати вхідні дані та видавати вихідні дані. Нейронні мережі можуть використовуватися для вирішення різних завдань, таких як класифікація, регресія та генерація.

Одним із найбільш популярних типів нейронних мереж є нейронна мережа із зворотним поширенням помилки (backpropagation). Цей тип мережі використовує алгоритм навчання, який дозволяє мережі автоматично налаштовувати ваги нейронів для мінімізації помилки між вихідними даними мережі та бажаними вихідними даними.

Нейронні мережі можуть також використовуватися для обробки зображень, голосове розпізнавання, автоматичне перекладання та ігри з ШІ. Наприклад, нейронні мережі можуть використовуватися для розпізнавання зображень в цілому та класифікації об'єктів на зображенні, або для генерації природної мови за допомогою мереж генеративно-змагальних нейронних мереж [2].

Створення нейронної мережі зазвичай відбувається в декілька кроків.

Першим кроком є визначення того, яке завдання необхідно вирішувати за допомогою нейронної мережі. Далі необхідно підготувати дані, які будуть використовуватися для навчання та тестування нейронної мережі.

Другим кроком є вибір архітектури нейронної мережі, яка буде використовуватися для вирішення задачі. Це може бути лінійна модель, повнозв'язкова нейронна мережа, згорткова нейронна мережа тощо.

Третім кроком є навчання мережі з використанням підготовлених даних.

Четвертий крок починається після навчання нейронної мережі. Необхідно протестувати на нових, невикористаних раніше даних. Це дозволить оцінити точність та ефективність моделі та виявити будь-які проблеми, які можуть бути виправлені надалі.

П’ятий, після тестування, можна проводити додаткові налаштування та покращення моди

Нейронні мережі також сприяють розвитку нових технологій, таких як ШІ та машинне навчання, які мають значний потенціал для зміни способу життя та бізнесу. У той же час необхідно розуміти, що нейронні мережі - це потужний інструмент, який може використовуватися в різних цілях, тому важливо вживати відповідних заходів для контролю та регулювання використання нейронних мереж. Таким чином, необхідно забезпечити транспарентність, відкритість та відповідальність у використанні нейронних мереж, щоб переконатися, що вони використовуються з метою, яка приносить користь суспільству в цілому.

Серед найвідоміших проектів можна виділити наступні:

AlphaGo – комп'ютерна програма, розроблена Google DeepMind, яка використовує нейронні мережі для гри в гру Go. AlphaGo став першою комп'ютерною програмою, яка перемогла чемпіона світу у грі Go.

OpenAI GPT - генеративна нейронна мережа, яка здатна генерувати текст природною мовою з високим ступенем подібності з текстом, написаним людиною.

ImageNet – нейронна мережа, яка використовується для розпізнавання зображень. Вона була розроблена та навчена на великій кількості зображень, щоб визначати об'єкти, особи та інші сутності на зображенні.

U-Net – це нейронна мережа, яка використовується для задач сегментації зображень. Вона була розроблена для завдань медичної діагностики, але також може використовуватися

Зрештою, нейронні мережі - це потужний інструмент для автоматизації та оптимізації безлічі завдань, і їх використання зростатиме та розвиватиметься в майбутньому. Важливо, щоб суспільство було усвідомленим та вживало заходів для управління та контролю їх використання, щоб забезпечити правильний напрямок та процвітання у майбутньому.

1.2 Нейронна мережа в іграх та додатках

Нейронні мережі (НМ) та штучний інтелект (ШІ) є важливими технологіями для розвитку ігор та програм. Нейронні мережі можуть використовуватися для реалізації різних функцій в іграх та програмах. Нейронні мережі можуть використовуватися для створення AI для персонажів в іграх, які можуть діяти адаптивно та навчатися від своїх дій та взаємодії з іншими персонажами та оточенням. Відноситься це також до комп’ютерного зору [18]. Мережа може використовуватися для розпізнавання об'єктів і жестів в іграх та програмах, дозволяючи користувачеві керувати грою або програмою за допомогою рухів тіла або жестів. Крім цього, нейронні мережі можуть використовуватись для аналізу ігрового процесу та адаптивного регулювання рівня складності залежно від навичок гравця. Для дизайну, нейронні зв’язки можуть використовуватися для створення контенту в іграх, наприклад, для створення нових персонажів, об'єктів і локацій.

В цілому, нейронні мережі та ШІ надають багато переваг для розвитку ігор та програм, дозволяючи створювати більш реалістичні, захоплюючі та інтелектуальні ігрові світи та програми. Ці технології дозволяють створювати більш складні ШІ, які можуть діяти адаптивно та навчатися, забезпечуючи більш глибокий та інтерактивний ігровий досвід. Нейронні мережі та ШІ також можуть використовуватися для аналізу ігрового процесу та адаптивного регулювання рівня складності, що дозволяє створювати кращий ігровий досвід для гравців [22].

1.3 Комп’ютерні симуляції та комп`ютерна модель

Комп'ютерна симуляція – це метод, який використовується для моделювання реальних систем та процесів на комп'ютері. Цей метод може використовуватися в різних галузях, таких як наука, інженерія, медицина та бізнес. Однією з головних причин використання комп'ютерної симуляції є те, що вона дозволяє вивчати системи та процеси, які складно чи неможливо дослідити безпосередньо.

Однією з головних областей, де комп'ютерна симуляція широко використовується, є інженерна справа. Інженери використовують симуляцію для проектування та оптимізації різних компонентів та систем, таких як автомобілі, літальні апарати та медичне обладнання.

Комп'ютерне моделювання – метод уявлення об'єкта, який відрізняється від реального, але дуже наближений до дійсності. За допомогою такого моделювання, теоретично, можливо вивчати фізико-хімічні, біологічні, хімічні закони та експерименти, які неможливо провести в реальному житті.

Один із прикладів комп'ютерної симуляції – це моделювання та симуляція авіакомпаній та аеропортів. За допомогою комп'ютерної симуляції можна моделювати роботу аеропорту, систем керування польотами, планування маршрутів літаків та багато іншого. Це дозволяє авіакомпаніям оцінювати продуктивність та планування, а також шукати оптимальні рішення для підвищення ефективності та зниження витрат. Інший приклад може бути моделювання процесів у хімічній індустрії, де можна симулювати та оцінювати різні сценарії реакцій та процесів, допомагаючи розробникам знайти найбільш оптимальні рішення для виробництва та зменшення забруднення.

Переваги комп'ютерного моделювання:

– вільне та доступне у використанні;

– можна розраховувати та створювати такі об'єкти, які у реальних умовах неможливі;

– з допомогою комп'ютерного моделювання можна як спостерігати, а й прогнозувати результати експериментів;

– знаходити оптимальну форму та конструкцію не створюючи пробних деталей;

– експерименти без ризику для здоров'я людини і не загрожує природі;

– можливість огляду об'єкта з усіх боків.

Недоліки комп'ютерного моделювання:

– помилкою є те що моделювання може якісно виявляти нові явища. т.к. має бути підтвердження у реальних умовах та у реальних експериментах;

– модельний аналіз зменшує можливі пояснення. з об'єкта моделювання можна «вичавити» лише те, що входить у рамки моделі;

– комп'ютерне моделювання - це моделювання, яке досліджує моделі в теорії, не має підтвердження та доказів природних явищ у реальних умовах. Тобто разом з теорією має йти і практика, щоб дослідник міг порівнювати результати одразу у двох напрямках та оцінювати якість моделі.

1.4 Аналіз застосунків, що використовують комп’ютерну симуляцію

Існує багато цікавих рішень у вигляді онлайн додатку, або програм для проведення тестів та аналізу різних ситуацій. Майже всі симулятори можна привести в приклад, як аналогічні застосунки. Але найбільш приближено схожими є онлайн сервіс-мапа Nukemap, ігри-симулятори Plague inc та інші.

Nukemap це інтерактивна карта (рис. 1), яка дозволяє користувачам візуалізувати потенційні наслідки вибуху ядерної боєголовки у різних містах світу [10]. Користувачі можуть вибрати тип боєголовки, потужність вибуху та місце, де він відбудеться, і переглянути охоплення ураження, включаючи області виявлення, опіків та радіаційного опромінення. Це інструмент освіти, який допомагає людям зрозуміти потенційні наслідки ядерної війни.

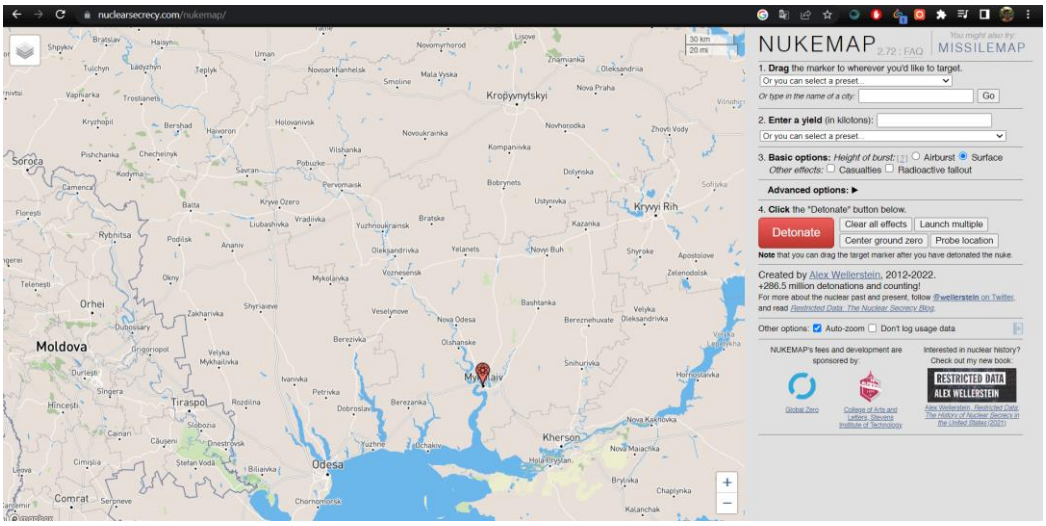


Рисунок 1.1 – Онлайн симулятор Nukemap

Nukemap має кілька плюсів:

– дозволяє людям візуалізувати потенційні наслідки ядерного вибуху, що може допомогти краще зрозуміти ризики та наслідки ядерної війни;

– користувачі можуть змінювати параметри вибуху, щоб дослідити різні сценарії та зрозуміти, як вони можуть вплинути на різні міста та регіони;

– nukemap може бути використаний як освітній інструмент для студентів та науковців, щоб допомогти їм краще зрозуміти ризики та наслідки ядерної війни.

Plague Inc. це гра-симулятор (рис. 1.2), в якій гравці використовують різні способи, щоб поширити інфекцію та викликати глобальне зараження. Гравець обирає расу вірусу і починає гру з однієї людини, зараженої в одній країні. Потім він використовує різні способи, щоб поширити інфекцію та зробити її глобальною, при цьому долаючи медичні та політичні перепони.

Рисунок 1.2 – Plague inc гра-симулятор

Гравець може розвивати та покращувати свій вірус, додаючи нові симптоми та способи поширення. Гравець має бути обережним, оскільки люди та уряди можуть вживати заходів для боротьби з інфекцією. Мета гри - знищити людство або якнайбільше вбити людей, при цьому уникаючи діагностики та контролю.

Plague Inc. Крім не дуже веселого сенсу гри може бути корисним застосунком:

– гра може допомогти людям краще зрозуміти механізми поширення інфекції та заходи протидії, які можуть бути вжиті для боротьби з ними;

– гравцям потрібно приймати рішення та адаптуватися до змін у ситуації, що допомагає розвивати їхнє критичне мислення.

Однак, необхідно мати на увазі, що гра імітує реальні наслідки зараження і не повинна бути прийнятою поведінкою в житті, створюючи або продовжуючи реальні інфекції.

1.5 Вибір середовища для створення симуляції

Для роботи з Математичними моделями та ШІ зазвичай вистачає звичайного компілятору. Але в такому разі буде втрачено потенціал у створені зручного додатку для користувачів. Для цієї задачі підходить середовище для розробки ігор Unity3D.

Unity - це провідна платформа у світі для створення та управління інтерактивним 3D-вмістом у режимі реального часу, що забезпечує інструменти для створення дивовижних ігор та їх публікації на широкому діапазоні пристроїв (рис. 1.3).

Рисунок 1.3 – Логотип рушія для створення ігор

Unity3d є чудовим застосунком для створення ігор та віртуальної реальності, але його потужні інструменти та гнучкість роблять його придатним для створення не лише ігор, а й інших видів програм та проектів .

Unity3d використовується для створення віртуальних турів будівлями та іншими спорудами, дозволяючи користувачам досліджувати проект до того, як він буде збудований. Крім цього, рушій може використовуватись для створення навчальних та тренувальних додатків, допомагаючи студентам візуалізувати та краще розуміти складні концепції. Unity використовується в медичній галузі для створення віртуальних тренажерів та моделей органів, допомагаючи у навчанні медичного персоналу та плануванні операцій. Крім

технічного плану, наприклад, кінокомпанії вдаються до ігрових рушіїв, для створення контекту для фільмів завдяки можливостям цього додатку.

1.6 Постановка задачі

Ідея данного застосунку полягає в тому, щоб користувачі а з будь якої точки світу змогли випробувати цей симулятор на будь якому пристрої. Як було згадано раніше, життя приносить багато різних ситуацій, до яких кожен з нас має бути готовим. Тому, створення симулятору з можливістю відтворення різних природних і не природних ситуацій для проведення тестування на нейронній мережі буде великою перевагою серед інших симуляторів. Адже нейронні звязки це природне явище для людини. Основуючись на цьому, поведінка штучних істот з людською поведінкою, дає змогу провести статистику наближену до реальних цифр.

Для створення якісного застосунку, потрібно дотримуватись наступного плану.

Дослідження та аналіз вимог: вивчення вимог до додатку, аналіз аудиторії та цільової групи, вивчення конкурентів та визначення функціональності.

Проектування та планування: створення концепції та дизайну програми, складання технічного завдання та планування розробки.

Розробка: написання коду, реалізація функціональності, тестування та налагодження.

Тестування: тестування функціональності, випробування на сумісність, випробування на продуктивність, виявлення та усунення помилок.

Висновки до розділу 1

В даному розділі було розглянуто різні об’єкти, які служитимуть в створені комп’ютерної моделі симуляції популяції організмів на основі нейронних мереж та ігрового рушія Unity3D.

Отже можна сказати, що нейронні мережі є потужним інструментом на вирішення різних завдань машинного навчання, зокрема комп'ютерної візуалізації. Виконання завдання включатиме дослідження та застосування нейронних мереж для вирішення конкретної задачі в галузі комп'ютерної візуалізації, наприклад створення реалістичної поведінки в різних іграх чи симулятора.

2 ХАРАКТЕРИСТИКА ДОСЛІДЖУВАНОГО ОБ'ЄКТА

2.1 Принцип роботи нейронів

Нейрон – це обчислювальна одиниця, яка отримує інформацію, здійснює над нею прості обчислення і передає її далі. Вони діляться на три основні типи: вхідний, прихований та вихідний (рис. 2.1).

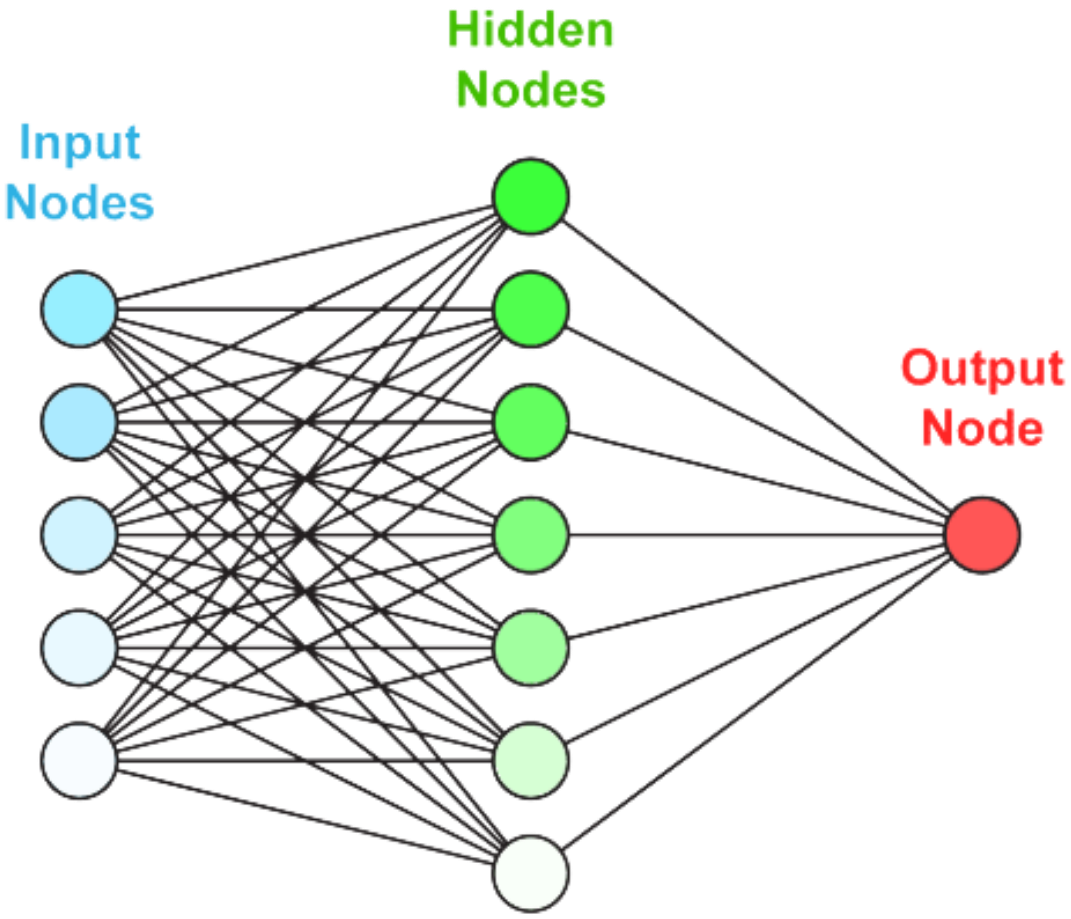


Рисунок 2.1 – Схема з'єднання нейронів

Вхідний рівень: вхідний рівень отримує вхідні дані, наприклад зображення, аудіо або текст.

Прихований(і) шар(и): прихований(і) шар(и) обробляє вхідні дані, використовуючи вагові коефіцієнти та зміщення для генерації виходу для кожного нейрона.

Вихідний рівень: вихідний рівень генерує кінцевий результат на основі виходів із прихованого шару(ів).

Попереднє розповсюдження: вхідні дані обробляються через мережу шляхом передачі їх із вхідного рівня на вихідний.

Функція втрат: результат порівнюється з очікуваним результатом за допомогою функції втрат. Мета полягає в тому, щоб мінімізувати втрати, які вимірюють різницю між прогнозованими та фактичними результатами.

Зворотне розповсюдження: потім помилка поширюється мережею назад, а ваги та зміщення коригуються, щоб зменшити втрати на наступній ітерації.

Повторення: цей процес повторюється кілька разів, доки втрати не будуть мінімізовані до прийнятного рівня.

Прогноз: після навчання моделі її можна використовувати для прогнозування нових вхідних даних.

2.2 Опис технології нейронної мережі

Для навчання нейронних мереж використовується алгоритм зворотного поширення помилки, що дозволяє знаходити оптимальні ваги кожного нейрона. Залежно від завдання та архітектури нейронної мережі можуть використовуватись різні типи шарів, такі як повнозв'язкові, згорткові або рекурентні шари. Нейронні мережі використовуються в багатьох програмах, таких як аналіз зображень, розпізнавання мовлення, генерація тексту та багато інших.

В даний час нейронні мережі є одними з найпотужніших інструментів машинного навчання, і їх застосування зростає у багатьох галузях, таких як медицина, фінанси, маркетинг та багато інших [12]. Однією з основних переваг нейронних мереж є їх здатність автоматично вивчати складні закономірності даних, без необхідності ручного програмування правил. Однак, разом з цим нейронні мережі також можуть бути складними для розуміння та налагодження, і їх моделі можуть бути вразливими до

перенавчання. Важливо вибирати відповідну архітектуру та методи навчання для кожного конкретного завдання [4].

Загалом існує 9 типів нейромереж:

– персептрон (Perceptron);
– нейронна мережа прямої подачі (Feed Forward Neural Network);
– багатошаровий персептрон (Multilayer Perceptron);
– згорткова нейронна мережа (Convolutional Neural Network);
– радіальна базова функціональна нейронна мережа (Radial Basis Functional Neural Network);
– рекурентна нейронна мережа (LSTM – Long Short-Term Memory);
– моделі «послідовність до послідовності» (Sequence to Sequence Models);
– модульна нейронна мережа (Modular Neural Network).

2.2.1 Тип персептрон

Модель персептрона, запропонована Мінським-Пейпертом, є однією з найпростіших і найстаріших моделей нейрона. Це найменша одиниця нейронної мережі, яка виконує певні обчислення для виявлення особливостей або бізнес-аналітики у вхідних даних. Він приймає зважені вхідні дані та застосовує функцію активації, щоб отримати результат як кінцевий результат. Персептрон також відомий як TLU (порогова логічна одиниця).

Perceptron — це алгоритм навчання під наглядом, який класифікує дані за двома категоріями, отже, це двійковий класифікатор. Персептрон розділяє вхідний простір на дві категорії гіперплощиною.

Переваги Perceptron

Персептрони можуть реалізувати такі логічні ворота, як AND, OR або NAND.

Недоліки Perceptron

Персептрони можуть вивчати лише задачі з лінійним відокремленням, такі як проблема логічного AND. Для нелінійних задач, таких як логічна проблема XOR, це не працює (рис. 2.2)

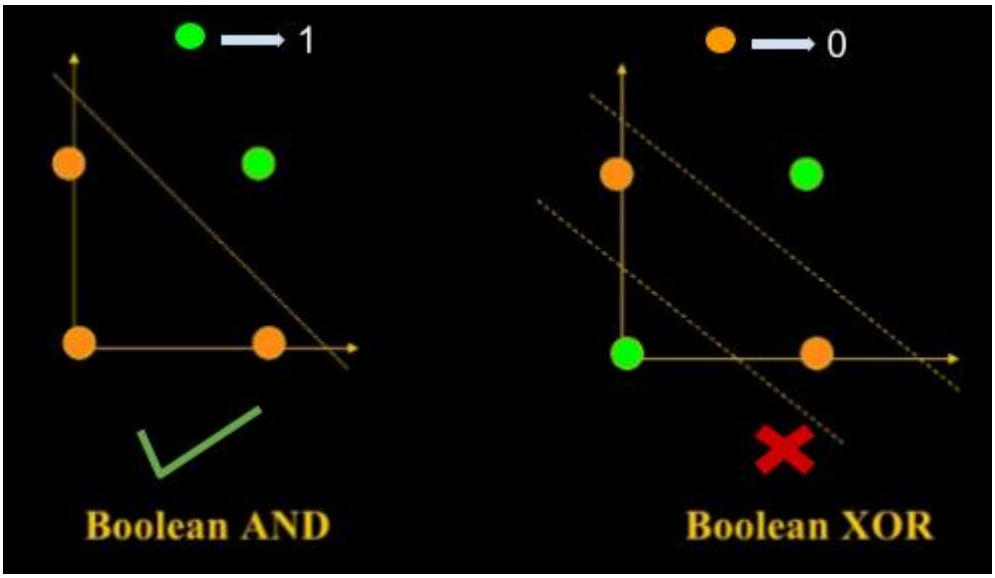


Рисунок 2.2 – Недолік Perceptron

2.2.2 Тип прямої подачі

Застосування нейронних мережах прямої подачі:

- проста класифікація (де традиційні алгоритми класифікації на основі машинного навчання мають обмеження);
- розпізнавання обличчя (проста пряма обробка зображень);
- комп’ютерний зір (де цільові класи важко класифікувати);
- розпізнавання мови.

Найпростіша форма нейронних мереж, де вхідні дані рухаються лише в одному напрямку, проходячи через штучні нейронні вузли та виходячи через вихідні вузли (рис. 2.3) Там, де приховані шари можуть або не можуть бути присутніми, присутні вхідний і вихідний шари. Виходячи з цього, їх можна

далі класифікувати як одношарові або багаторівневі нейронні мережі прямого зв’язку.

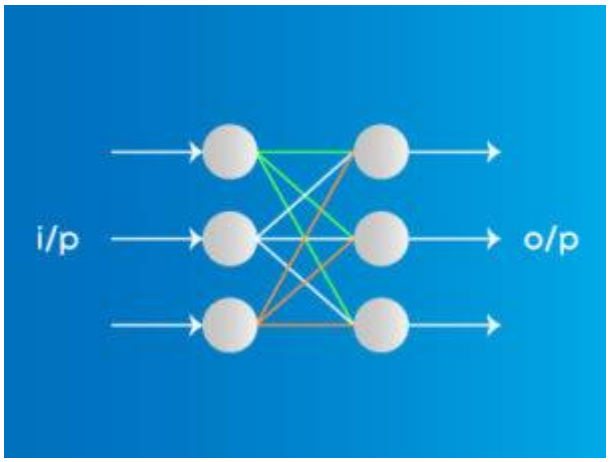


Рисунок 2.3– Схема дії нейронів в Прямій подачі

Кількість шарів залежить від складності функції. Тип має односпрямоване пряме поширення, але не має зворотного. Ваги тут статичні. Функція активації подається вхідними даними, які помножуються на ваги. Для цього використовується класифікуюча функція активації або функція покрокової активації. Наприклад: нейрон активується, якщо він перевищує порогове значення (зазвичай 0), і нейрон видає 1 як вихід. Нейрон не активується, якщо він нижче порогового значення (зазвичай 0), яке вважається -1. Вони досить прості в обслуговуванні та обладнані для роботи з даними, які містять багато шуму.

Переваги прямої нейронної мережі:

- менш складний, простий у проектуванні та обслуговуванні;
- швидкий і швидкий (одностороннє розповсюдження);
- висока чутливість до зашумлених даних.

Недоліки прямої нейронної мережі:

Не можна використовувати для глибокого навчання (через відсутність щільних шарів і зворотного поширення)

2.2.3 Тип багатошаровий персептрон

Застосунки на багатошаровому персептроні:

– розпізнавання мови;

– машинний переклад;

– комплексна класифікація.

Точка входу до складних нейронних мереж, де вхідні дані проходять через різні шари штучних нейронів. Кожен окремий вузол з’єднаний з усіма нейронами на наступному рівні, що робить його повністю пов’язаною нейронною мережею. Присутні вхідні та вихідні шари з кількома прихованими шарами, тобто загалом щонайменше три або більше шарів. Він має двонаправлене поширення, тобто пряме поширення та зворотне поширення (рис. 2.4).

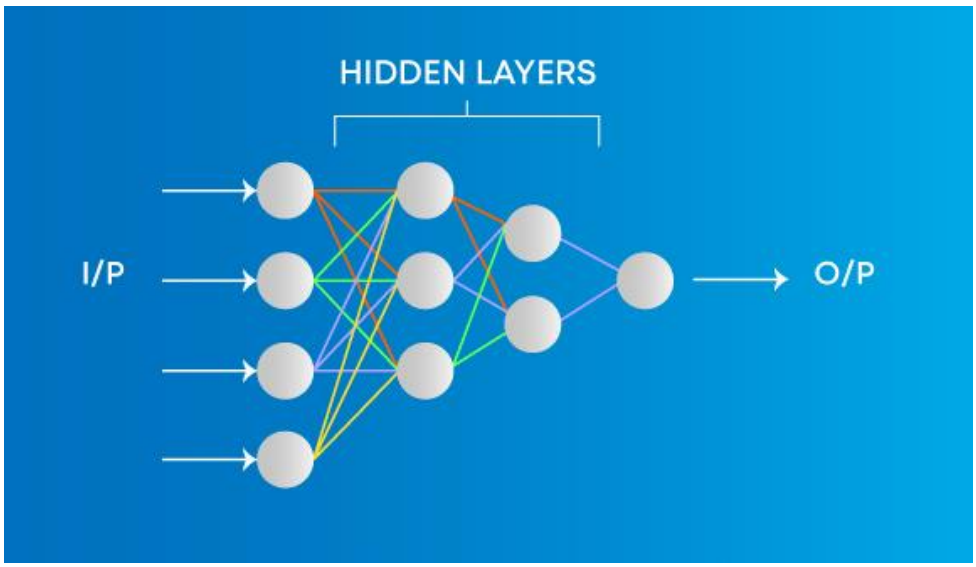


Рисунок 2.4 – Схема дії нейронів в Багатошаровому персептроні

Вхідні дані множаться на вагові коефіцієнти та передаються до функції активації, а при зворотному поширенні вони змінюються, щоб зменшити втрати. Простими словами, ваги - це значення, отримані машиною з нейронних мереж. Вони самоналаштовуються залежно від різниці між прогнозованими результатами та навчальними входами. Використовуються нелінійні функції активації, а потім softmax як функція активації вихідного рівня.

Переваги багатошарового персептрона полягають у використанні для глибокого навчання (через наявність щільних повністю з'єднаних шарів і зворотного поширення).

Недоліки багатошарового перцептрона:

– порівняно складний у проектуванні та обслуговуванні;
– порівняно повільно (залежить від кількості прихованих шарів).

2.2.4 Тип згорткової нейронної мережі

Програми на згортковій нейронній мережі:

– обробка зображення;
– комп'ютерний зір;
– розпізнавання мови;
– машинний переклад.

Згорткова нейронна мережа містить тривимірне розташування нейронів замість стандартного двовимірного масиву. Перший шар називається згортковим. Кожен нейрон згорткового шару обробляє лише інформацію з невеликої частини поля зору. Вхідні функції беруться пакетно, як фільтр. Мережа розуміє зображення по частинах і може обчислювати ці операції кілька разів, щоб завершити повну обробку зображення. Обробка передбачає перетворення зображення з шкали RGB або HSI на градацію сірого. Подальші зміни в значенні пікселя допоможуть виявити краї, і зображення можна класифікувати за різними категоріями (рис. 2.5).

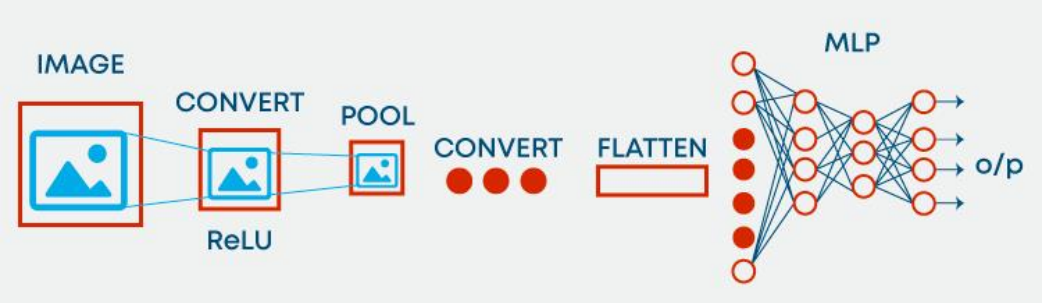


Рисунок 2.5 – Схема роботи нейронної мережі за Згортковим типом

Розповсюдження є односпрямованим, де CNN містить один або кілька згорткових шарів з подальшим об’єднанням, і двонаправленим, де вихід згорткового шару надходить до повністю зв’язаної нейронної мережі для класифікації зображень, як показано на схемі вище. Фільтри використовуються для виділення певних частин зображення. У MLP вхідні дані множаться на ваги та передаються до функції активації. Convolution використовує RELU, а MLP використовує нелінійну функцію активації, за якою слідує softmax. Згорткові нейронні мережі показують дуже ефективні результати в розпізнаванні зображень і відео, семантичному розборі та виявленні перефразів.

Переваги згорткової нейронної мережі:

– використовується для глибокого навчання з невеликою кількістю параметрів;

– менше параметрів для вивчення порівняно з повністю підключеним рівнем.

Недоліки згорткової нейронної мережі:

– порівняно складний у проектуванні та обслуговуванні;

– порівняно повільно (залежить від кількості прихованих шарів).

2.2.5 Тип радіально базової мережі

Мережа радіальних базових функцій складається з вхідного вектора, за яким слідує шар нейронів RBF і вихідного рівня з одним вузлом на категорію. Класифікація виконується шляхом вимірювання подібності вхідних даних до точок даних із навчального набору, де кожен нейрон зберігає прототип. Це буде один із прикладів із навчального набору.

Коли новий вхідний вектор (n-вимірний вектор, який ви намагаєтеся класифікувати) потребує класифікації, кожен нейрон обчислює евклідову відстань між вхідним сигналом і його прототипом. Наприклад, якщо у нас є два класи, тобто клас A та клас B, тоді новий вхід, який потрібно

класифікувати, є більш близьким до прототипів класу A, ніж до прототипів класу B. Отже, його можна позначити або класифікувати як клас A.

Кожен нейрон RBF порівнює вхідний вектор із своїм прототипом і виводить значення в діапазоні, яке є мірою подібності від 0 до 1. Оскільки вхідний сигнал дорівнює прототипу, вихід цього нейрона RBF буде 1, а відстань між вхід і прототип відповідь експоненціально падає до 0. Крива, згенерована відповіддю нейрона, прагне до типової дзвоноподібної кривої. Вихідний рівень складається з набору нейронів (по одному на категорію) (рис. 2.6)

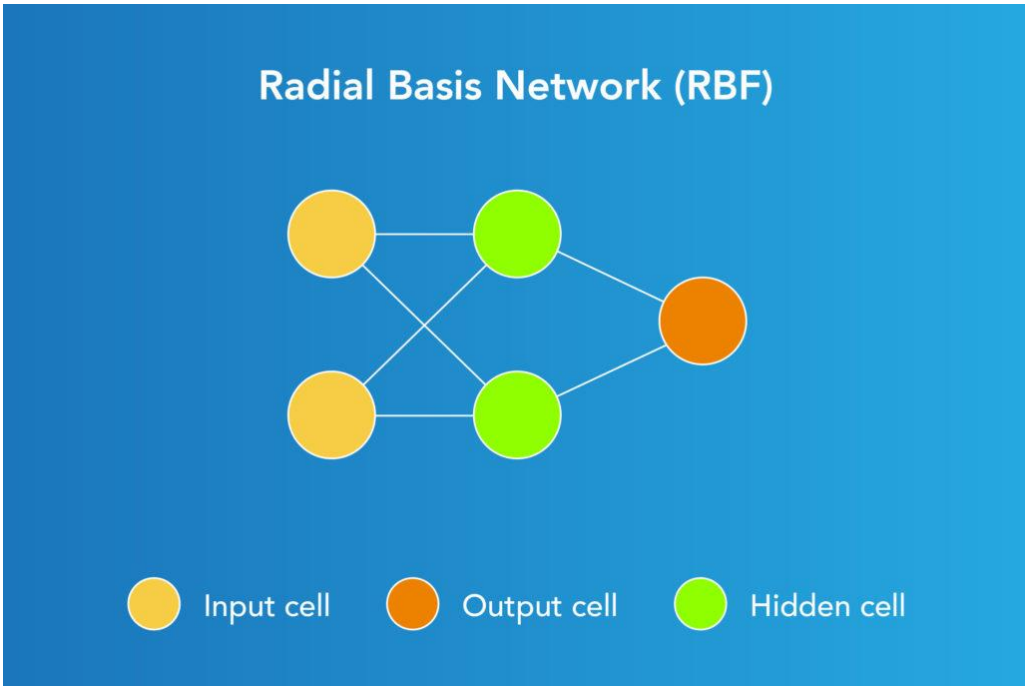


Рисунок 2.6 – Схема роботи Радіальних базових функцій

2.2.6 Тип рекуретної мережі

Застосування рекурентних нейронних мереж:

– обробка тексту, наприклад автоматичні пропозиції, перевірка раматики тощо;
– перетворення тексту в мовлення;
– тег для зображень;
– аналіз настроїв;
– переклад.

Мережа розроблена для збереження виходу шару, повторювана нейронна мережа повертається на вхід, щоб допомогти передбачити результат шару. Перший рівень, як правило, є прямою нейронною мережею, за якою слідує повторюваний рівень нейронної мережі, де деяка інформація, яку він мав на попередньому часовому кроці, запам’ятовується функцією пам’яті. У цьому випадку реалізовано пряме поширення. Він зберігає інформацію, необхідну для подальшого використання. Якщо прогноз неправильний, для внесення невеликих змін використовується швидкість навчання. Отже, поступово збільшуючи його, щоб зробити правильний прогноз під час зворотного поширення (рис. 2.7).

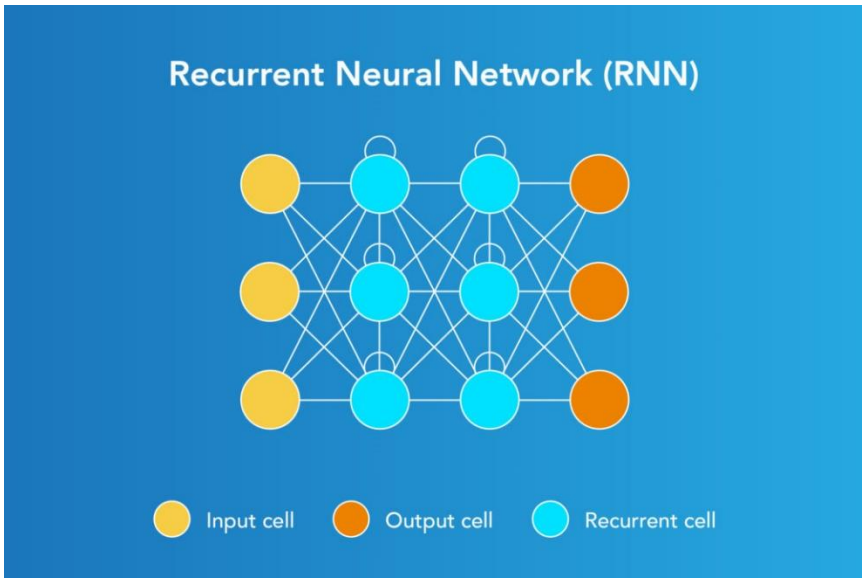


Рисунок 2.7– Схема роботи нейронів рекуретної мережі

Переваги рекурентних нейронних мереж:

- модель послідовних даних, де кожна вибірка може вважатися залежною від історичних, є однією з переваг;
- використовується з шарами згортки для підвищення ефективності пікселів.

Недоліки рекурентних нейронних мереж:

- проблеми зникнення та вибуху градієнта;
- навчання рекурентних нейронних мереж може бути складним завданням;
- важко обробляти довгі послідовні дані за допомогою relu як функції активації.

2.2.7 Тип моделі послідовності

Модель послідовності складається з двох рекурентних нейронних мереж. Тут існує кодер, який обробляє вхід, і декодер, який обробляє вихід. Кодер і декодер працюють одночасно – або використовуючи той самий параметр, або різні. Ця модель, на відміну від реальної RNN, особливо застосовна в тих випадках, коли довжина вхідних даних дорівнює довжині вихідних даних. Хоча вони мають ті ж переваги та обмеження, що й RNN, ці моделі зазвичай застосовуються в основному в чат-ботах, машинних перекладах і системах відповідей на запитання (рис. 2.8).

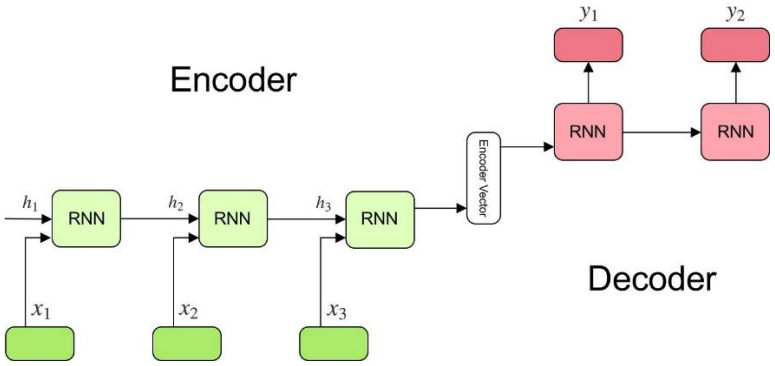


Рисунок 2.8– Схема роботи нейронних мереж

2.2.8 Тип модульної нейронної мережі

Застосування модульної нейронної мережі:

- Системи прогнозування фондового ринку;
- Адаптивний MNN для розпізнавання символів;
- Стиснення вхідних даних високого рівня.

Модульна нейронна мережа має ряд різних мереж, які функціонують незалежно та виконують підзавдання. Різні мережі насправді не взаємодіють і не сигналізують одна одній під час процесу обчислень. Вони самостійно працюють над досягненням результату.

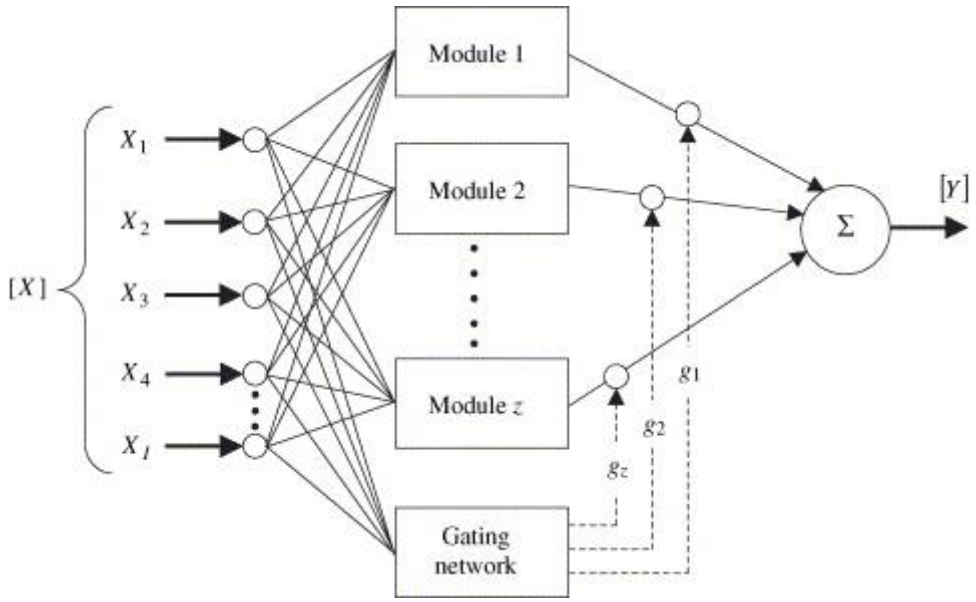


Рисунок 2.9– Модульна нейронна мережа

У результаті великий і складний обчислювальний процес виконується значно швидше, розбиваючи його на незалежні компоненти. Швидкість обчислень збільшується, оскільки мережі не взаємодіють і навіть не з’єднані одна з одною.

Переваги модульної нейронної мережі:

– ефективний;
– самостійне навчання;
– міцність.

Недоліки модульної нейронної мережі

– проблеми з рухомою ціллю.

2.3 Синапсис нейронів

Синапсис – це зв'язки між нейронами у нейронній мережі. Кожен синапс має вагу, що визначає силу зв'язку між двома нейронами. Ваги синапсів можуть бути оптимізовані за допомогою методів машинного навчання, таких як зворотне поширення помилки, щоб поліпшити якість передбачення або класифікації мережі.

У нейронній мережі синапси є тонкими лініями, які з'єднують два нейрони. Вхідні дані подаються в нейрони вхідного шару, а вихідні дані виходять із нейронів вихідного шару (рис. 2.10). Нейрони прихованих шарів є елементами, які обробляють інформацію, отриману від вхідних нейронів і передають її у вихідні нейрони.

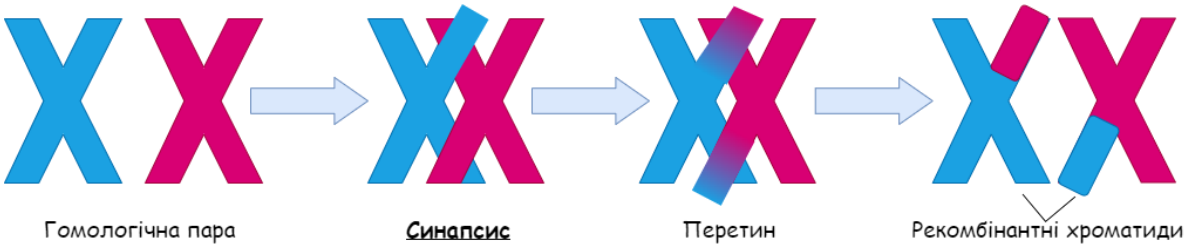


Рисунок 2.10 – Послідовний процес обміну даними

Ваги синапсів можуть бути позитивними або негативними, вони визначають, наскільки сильно дані, передані одним нейроном іншому, впливають на його вихід. Якщо вага синапсу позитивна, це означає, що вхідні дані посилюють сигнал, переданий у вихідний нейрон. Якщо ж вага синапсу

негативна, це означає, що вхідні дані згасають сигнал, переданий у вихідний нейрон.

Загалом, синапси відіграють ключову роль у формуванні вихідних сигналів нейронної мережі та є одним із найважливіших компонентів у механізмі навчання мережі.

2.4 Опис генетичного алгоритму

Генетичні алгоритми нейронних мереж (GA-NNs) - це метод оптимізації, який використовує принципи еволюції для пошуку оптимальних архітектур нейронних мереж та ваг синаптичних зв'язків. Вони є поєднанням генетичного алгоритму і нейронних мереж [13].

У GA-NNs, кожен індивід у популяції є нейронною мережею з певними архітектурою і вагами зв'язків. Ваги зв'язків можуть бути представлені у вигляді числових значень або векторів. Генетичний алгоритм використовується для вибору кращих індивідів, їх схрещування та мутації, щоб отримати нову популяцію індивідів. Цей процес повторюється багаторазово, доки буде досягнуто певний критерій зупинки.

GA-NNs можуть бути застосовані у широкому спектрі завдань, таких як класифікація, регресія, навчання з підкріпленням та пошук оптимуму. Вони добре підходять для складних завдань, таких як налаштування гіперпараметрів, в яких важко передбачити оптимальні значення ваги.

Однією з переваг GA-NNs є те, що вони можуть автоматично досліджувати широкий спектр архітектур і вагів, вибираючи найпридатніші для вирішення конкретної задачі. Це дозволяє уникнути труднощів з вибором початкових ваг та архітектури, що є однією з проблем, пов'язаних із традиційним навчанням нейронних мереж [13].

Нейроеволюція доповнювальних топологій (NEAT) — це нейроеволюційний підхід до штучного інтелекту та машинного навчання [7]. Це форма генетичного алгоритму, який використовує еволюційні принципи

для створення нейронних мереж для вирішення проблем. NEAT був розроблений Кеннетом О. Стенлі та Рісто Мійккулайненом у 2002 році і особливо добре підходить для проблем, у яких структура мережі невідома або може змінюватися, наприклад, навчання з підкріпленням і проблеми адаптивного керування.

NEAT використовує генетичний алгоритм для розробки як вагів, так і топології (тобто структури) нейронних мереж [12]. Це на відміну від традиційних алгоритмів навчання нейронної мережі, які зазвичай змінюють лише ваги мережі. У NEAT кожна мережа представлена як геном, який кодує як її топологію, так і її ваги, а генетичний алгоритм використовується для розвитку цих геномів протягом поколінь.

Однією з ключових особливостей NEAT є його здатність зберігати корисні структури успішних мереж через покоління, щоб процес еволюції спирався на попередній успіх. Це дозволяє NEAT з часом розвивати більш складні та ефективні мережі. Крім того, NEAT використовує видоутворення, яке групує подібні геноми разом і запобігає втраті генетичного різноманіття з часом, дозволяючи еволюції продовжувати досліджувати нові рішення та уникати застрягання в локальних мінімумах.

NEAT застосовувався для вирішення різноманітних завдань у таких сферах, як робототехніка, комп’ютерні ігри та автономні транспортні засоби, і показав багатообіцяючі результати в багатьох із цих областей. NEAT забезпечує гнучку та потужну структуру для розвитку нейронних мереж і може використовуватися для вирішення широкого кола проблем.

Крім цього, NEAT може вирішувати різні задачі завдяки різноманітним доповненними методами:

- RT-NEAT;
- HyperNEAT;
- CG-NEAT;
- OD-NEAT.

RT-NEAT (Real-Time NeuroEvolution of Augmenting Topologies) це метод нейроеволюції, який використовується для навчання нейронних мереж у реальному часі. RT-NEAT заснований на NEAT (NeuroEvolution of Augmenting Topologies), але ефективніший у термінах швидкості та ефективності. Він слідує гібридний підхід, комбінуючи зумовлені архітектури з генетичним пошуком для підвищення продуктивності.

RT-NEAT використовується в різних областях, таких як управління роботами, ігри, а також для оптимізації та покращення ефективності нейронних мереж.

HyperNEAT - це метод нейроеволюції, який використовується для автоматичної побудови архітектур нейронних мереж. HyperNEAT заснований на NEAT (NeuroEvolution of Augmenting Topologies), але використовує складний генетичний алгоритм, який дозволяє автоматично збудувати складні архітектури нейронних мереж.

HyperNEAT створює архітектуру нейронної мережі у вигляді зовнішньої структури, яка називається гіперкубом, який моделює зв'язки між нейронами. Цей підхід дозволяє створювати ефективніші та ефективніші архітектури нейронних мереж, ніж у звичайних методах нейроеволюції.

HyperNEAT використовується в різних галузях, таких як штучний інтелект, дослідження в галузі нейронних мереж та в іграх для навчання ШІ.

Content-Generating NEAT — це метод генерації контенту за допомогою нейроеволюції, форми машинного навчання, яка використовує генетичні алгоритми для оптимізації штучних нейронних мереж.

Навчаючи систему на основі NEAT на великому наборі даних існуючого вмісту, система може вивчати шаблони та зв’язки в даних, а потім генерувати новий оригінальний вміст на основі цих шаблонів. Це може бути корисним для таких завдань, як створення творчого тексту, створення музики або створення нових творів мистецтва.

Важливо зазначити, що хоча контент, створений за допомогою NEAT, може бути новим і несподіваним, він не завжди має високу якість або підходить для певного завдання. Він також може містити упередження або інші неточності, які присутні в навчальних даних. Як і в будь-якій системі машинного навчання, важливо ретельно оцінити результат, створений NEAT, перш ніж використовувати його в реальних програмах.

odNEAT (онлайновий домен NEAT) — це варіант алгоритму NEAT (нейроеволюція доповнювальних топологій), який спеціально розроблений для онлайн-навчання, яке є формою машинного навчання, де модель адаптується до нових даних у режимі реального часу.

odNEAT особливо корисний для додатків, які включають динамічні та мінливі середовища, такі як робототехніка, онлайн-ігри та автономні системи. У цих програмах алгоритм NEAT можна використовувати для розвитку структури та ваг нейронних мереж у відповідь на нову інформацію, дозволяючи системі навчатися та адаптуватися з часом.

Однією з ключових відмінностей між odNEAT і традиційним NEAT є те, що odNEAT змінює структуру нейронної мережі на льоту, у міру появи нових даних. Це дозволяє системі постійно покращувати свою продуктивність і реагувати на зміни умов у реальному часі.

odNEAT використовувався в різноманітних програмах, включаючи керування роботами, відеоігри та оптимізацію складних інженерних проблем. Однак, як і будь-який алгоритм машинного навчання, важливо ретельно оцінити результат, створений odNEAT, і переконатися, що він підходить для поставленого завдання.

2.5 Переваги та недоліки симуляцій на нейронній основі

Симуляції на нейронній мережі можуть використовуватись для побудови прогностичних моделей для широкого спектру застосувань, таких як прогнозування фондових ринків, прогнозування погоди, прогнозування

спалахів захворювань. Мережу можна використовувати для підтримки прийняття рішень щодо охорони здоров’я, таких як діагностика та планування лікування.

Іншою сферою, де нейронні мережі можуть мати значний вплив, є моніторинг навколишнього середовища та стійкість. Нейронні мережі можна використовувати для аналізу великої кількості даних від датчиків та інших джерел для моніторингу та прогнозування змін навколишнього середовища та впливу людської діяльності на навколишнє середовище. Це може допомогти сформулювати екологічну політику та прийняти рішення, а також дати можливість окремим особам і організаціям вживати заходів, щоб зменшити свій вплив на навколишнє середовище та сприяти сталому розвитку.

У сфері робототехніки нейронні мережі можна використовувати для розробки передових робототехнічних систем, які можуть навчатися на досвіді та адаптуватися до мінливого середовища та завдань. Ці системи можуть бути використані в широкому діапазоні застосувань, таких як виробництво, будівництво, розвідка та пошуково-рятувальні операції.

У сфері спорту нейронні мережі можна використовувати для аналізу великої кількості даних про продуктивність, щоб допомогти спортсменам і тренерам приймати рішення щодо тренувань, харчування та інших факторів, які впливають на продуктивність. Нейронні мережі також можна використовувати для розробки нових технологій для покращення спортивних результатів, таких як переносні пристрої та системи тренувань.

Це лише кілька прикладів багатьох можливостей моделювання на основі нейронної мережі. Універсальність нейронних мереж дозволяє застосовувати їх для широкого кола завдань, що робить їх цінним інструментом для багатьох галузей.

Як і будь яка технологія та процесс, нейроні мережі мають велику кількість аспектів.

Переваги:

- висока точність: нейронні мережі здатні фіксувати складні зв’язки між входами та виходами, що забезпечує високу точність моделювання.
- гнучкість. нейронні мережі можна легко навчити на нових даних і вони можуть обробляти різні типи введення та виведення, що робить їх дуже гнучкими.
- узагальнення: нейронні мережі можуть узагальнювати нові приклади, навіть ті, які не є частиною навчальних даних, що робить їх корисними для моделювання в реальних сценаріях.
- масштабованість. нейронні мережі можна легко навчити на великих наборах даних і масштабувати для роботи з великомасштабним моделюванням.

Недоліки:

- витрати на обчислення: навчання нейронних мереж може бути дорогим з обчислювальної точки зору, вимагаючи значної обчислювальної потужності та часу.
- переобладнання. нейронні мережі можуть легко переобладнати навчальні дані, що призводить до низької продуктивності нових, невидимих даних.
- непрозорість. нейронні мережі може бути складно інтерпретувати, тому важко зрозуміти, чому симуляція робить певний прогноз.
- вимоги до даних: для ефективного навчання нейронним мережам потрібні великі обсяги даних, які може бути важко отримати для деяких симуляцій.

Висновки до розділу 2

В другому розділі було розглянуто принцип роботи нейронної мережі. Проаналізовано всі існуючі типи мереж, їх недоліки та переваги. Для роботи над проектом було обрано генетичний алгоритм під назвою NEAT. У простих контрольних завданнях алгоритм NEAT часто досягає ефективних мереж швидше, ніж інші сучасні нейроеволюційні методи та методи навчання з підкріпленням.

Загалом моделювання на основі нейронної мережі може забезпечити дуже точні та гнучкі результати, але воно також має певні обмеження та проблеми. Важливо ретельно розглянути ці фактори, вирішуючи, чи використовувати нейронну мережу для конкретного моделювання.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА РОЗРОБКА МОДЕЛІ

3.1 Вибір компонентів апаратно-програмного застосунку

Програмний застосунок отримав можливість працювати на багатьох доступних пристроях. Для роботи з нейронними мережами потрібна ЕОМ чи мала ЕОМ з високою продуктивністю, особливо щодо обчислень плаваючою комою. Це може бути як настільний комп'ютер із високопродуктивним процесором та відеокартою, так і мікрокомп’ютер з додаткоим відеопроцесором (GPU). Мікрокомпютер набагато менше та легше, ніж більшість ПК, і споживає значно менше енергії. Це робить його ідеальним рішенням для проектів у галузі штучного інтелекту чи комп'ютерного зору, які потребують мобільності чи незалежності від електромережі.

Серед існуючих рішень, які зараз існують на ринку, чудово підійдуть мікрокомп’ютери Raspberry Pi 4 (рис. 3.1) та Jetson Nano (рис. 3.2). Для того, щоб зробити правильний вибір серед цих варіантів, потрібно порівняти їх [23].

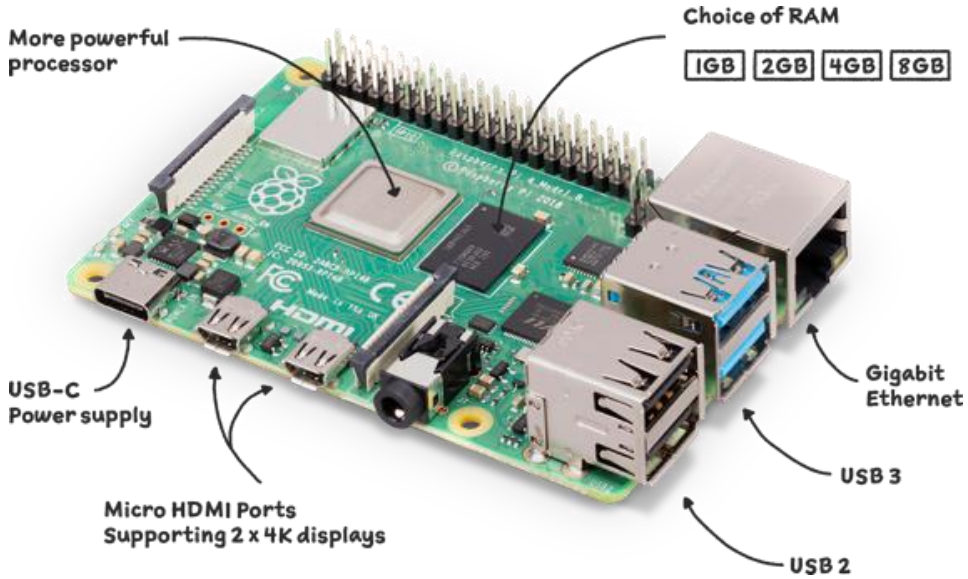


Рисунок 3.1 – Raspberry Pi 4

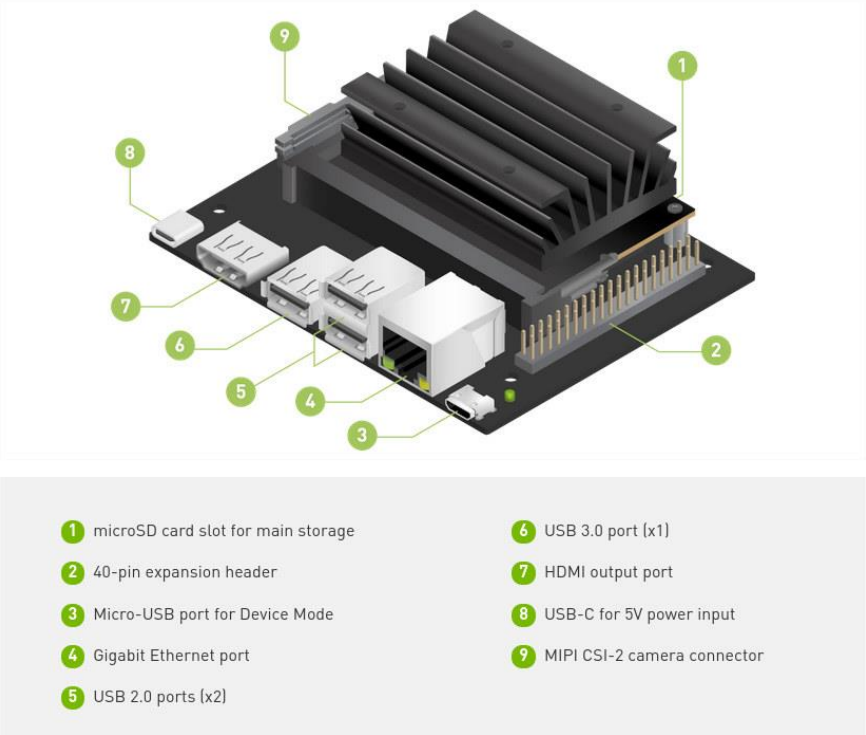


Рисунок 3.2 – Jetson Nano

Таблиця 3.1 – Характеристики Raspberry Pi 4 та Jetson Nano

Пристрій	Процесор	Пам’ять	Дисплей	Ввід/вивід	Ethernet	ОС	Ціна
Raspberry Pi 4	Система на чіпі Broadcom BCM2711 з 4 ядерним 64-bit процесор ARM Cortex-A72 1,5 ГГц	Має чотири версії оперативної пам’яті: 1 ГБ, 2 ГБ, 4 ГБ і 8 ГБ LPDDR4-3200 SDRAM	2x Micro-HDMI Raspberry Pi 4 4K 60 кадрів/с із подвійним екраном.	2X USB 3.0, 2X USB 2.0, USB C (для живлення), 3.5 мм Analog Audio/Video, 2X MicroHDMI, CSI, 40 pin GPIO, DSI	Gigabit Ethernet (Wi-Fi, Blutetooth)	ОС Raspberry PI, Ubuntu, OSMC, RetroPie та інші.	$125 – 2гб Версія $150 – 4гб Версія

Кінець таблиці 3.1

Jetson Nano	4-ядерний процесор ARM Cortex-A57 64-bit 1,43 ГГц.	LPDDR4 4 ГБ або альтернатива 2 ГБ	HDMI 2.0 і Display Port (eDP 1.4). Версія 2 ГБ підтримує лише HDMI 2.0	4гб версія: 4X USB 3.0, USB 2.0 Micro-B, 2X MIPI CSI-2, 40 pin GPIO, HDMI 2.0, DisplayPort 2гб версія: USB 3.0, 2X USB 2.0, USB Micro-b, MIPI CSI-2, 40 pin GPIO, HDMI 2.0	4гб версія: Gigabit Ethernet, M2 Key E; 2гб версія: Gigabit Ethernet, USB 802.11ac (залежно від регіону)	Linux4Tegra на базі Ubuntu 18.04 (запуск з накопичувача)	$280 – 2гб Версія $295 – 4гб Версія

Raspberry Pi 4 має достатні ресурси для запуску маленьких або середніх моделей нейронних мереж, але може бути недостатньо потужним для більш складних або великих моделей.

Jetson Nano, навпаки, має високу продуктивність, особливо щодо обчислень у плаваючій точці, і може використовуватися для більш складних моделей нейронних мереж.

Для запуску симулятору буде достатньо використати Raspberry Pi 4. Крім значно меншої ціни, Raspberry має більшу та активну спільноту розробників, ніж Jetson Nano, що означає більше можливостей для навчання та співпраці. Також Raspberry Pi може використовуватися для більш широкого спектру завдань, включаючи веб-сервери, медіа-сервери та проекти Інтернету речей, тоді як Jetson Nano більш спеціалізований для штучного інтелекту та комп'ютерного зору. Хоч і Jetson nano має кращий процесор та відеокарту, для реалізації проекту буде достатньо потужності Raspberry Pi 4.

3.2 Розробка архітектури апаратно-програмного застосунку

Для реалізації нейронної мережі, було створено 3 головних скрипта (рис. 3.3), завдяки яким, нейронна мережа створюється та починає розвиватись.

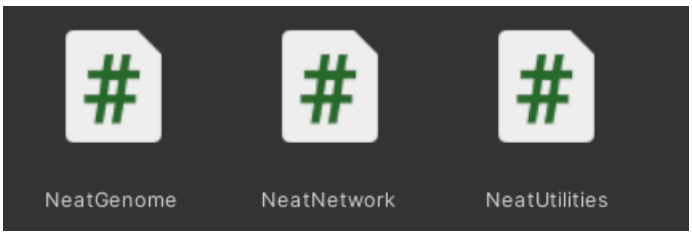


Рисунок 3.3 – Скрипти для роботи з нейронною мережею

Файл NeatGenome свого роду є оболонкою для геномів. В NeatNetwork утворюється мережа. NeatUtilities допомогає в роботі з функціями запису даних в файл, подальше зчитування з них [5][21].

Для створення мережі використано функції:

– створення мережевої структури: вузли (рис. 3.4);
– створення мережевої структури: ребра (рис. 3.5);
– створення мережевої структури: сусідні вузли (рис. 3.6).

```
private void CreateNetwork()
{
    foreach(NodeGene nodeGene in MyGenome.nodeGenes)
    {
        Node newNode = new Node(nodeGene.id);
        nodes.Add(newNode);
        if (nodeGene.type == NodeGene.TYPE.Input)
        {
            inputNodes.Add(newNode);
        }
        else if(nodeGene.type == NodeGene.TYPE.Hidden)
        {
            hiddenNodes.Add(newNode);
        }
        else if(nodeGene.type == NodeGene.TYPE.Output)
        {
            outputNodes.Add(newNode);
        }
    }
```

Рисунок 3.4 – Створення вузлів

```
foreach (ConGene conGene in MyGenome.conGenes)
{
    if(!conGene.isActive == true)
    {
        Connection newCon = new Connection(conGene.inputNode, conGene.outputNode, conGene.weight, conGene.isActive);
        connections.Add(newCon);
    }

}
```

Рисунок 3.5 – Створення ребр

```
foreach(Node node in nodes)
{
    foreach(Connection con in connections)
    {
        if (con.InputNode == node.id)
        {
            node.outputConnection.Add(con);
        }
        else if(con.OutputNode == node.id)
        {
            node.inputConnection.Add(con);
        }
    }
}
```

Рисунок 3.6 – Робота з сусідніми вузлами

Під час взаємодії нейронів між собою, вхідні дані мають розподілятись між шарами в правильних напрямках [8]. Для цього функція FeedForwardNetwork приймає вхідні дані і направляє їх в вірну позицію (рис. 3.7).

```
Ссылок: 0
public float[] FeedForwardNetwork(float[] inputs)
{
    float[] outputs = new float[outputNodes.Count];
    for(int i = 0; i < outputNodes.Count; i++)
    {
        inputNodes[i].SetInputNodeValue(inputs[i]);
        inputNodes[i].FeedForwardValue();
        inputNodes[i].value = 0;
    }
    for (int i = 0; i < hiddenNodes.Count; i++)
    {
        hiddenNodes[i].SetHiddenNodeValue();
        hiddenNodes[i].FeedForwardValue();
        hiddenNodes[i].value = 0;
    }
    for (int i = 0; i < outputNodes.Count; i++)
    {
        outputNodes[i].SetOutputNodeValue();
        outputs[i] = outputNodes[i].value;
        outputNodes[i].value = 0;
    }
    return outputs;
}
```

Рисунок 3.7 – Функція розподілення даних

Після першочергового налаштування створенної мережі, можемо перейти до підключення мережі до агенту [9]. Для того, щоб майбутній організм міг розрізняти предмети та небезпеку, потрібно застосувати датчики-сенсори. Unity має в собі вбудовану функцію Raycast [24].

Raycast в Unity — це функція фізики, яка проектує промінь у сцену, повертаючи логічне значення, якщо мета була успішно вражена. Коли це відбувається, інформація про попадання, наприклад відстань, позиція або посилання на трансформацію об’єкта, може бути збережена в змінній Raycast Hit для подальшого використання [16].

Оскільки агент має отримувати дані про навколишнє середовище, проміні мають бути розташовані навколо нього. Хоча це з одного боку легкий варіант, але він дуже ресурсомісткий і неправдоподібний для данного завдання. Тому для відтворення сценарію живої істоти, промінь має проектуватись за поворотом голови або тіла агента (рис. 3.8).

```
Ссылок: 1
private void InputSensors()
{
    Ray rayCast = new Ray(transform.position, transform.forward);
    RaycastHit hit;
    if (Physics.Raycast(rayCast,out hit, rayDistance))
    {
        if(hit.transform.tag=="Wall")
        {
            sensors[0] = hit.distance /hitDivider;
            Debug.DrawLine(rayCast.origin, hit.point, Color.red);
        }
    }
    rayCast.direction = (transform.forward + transform.right);
    if (Physics.Raycast(rayCast, out hit, rayDistance))
    {
        if (hit.transform.tag == "Wall")
        {
            sensors[0] = hit.distance / hitDivider;
            Debug.DrawLine(rayCast.origin, hit.point, Color.red);
        }
    }
    rayCast.direction = (transform.forward - transform.right) ;
    if (Physics.Raycast(rayCast, out hit, rayDistance))
    {
        if (hit.transform.tag == "Wall")
        {
            sensors[0] = hit.distance / hitDivider;
            Debug.DrawLine(rayCast.origin, hit.point, Color.red);
        }
    }
}
```

Рисунок 3.8– Функція зчитування інформації через промінь

Було створено скрипт з функцією збереження інформації про геном наших нейронних мереж, щоб у майбутньому була можливість завантажити її назад у симуляцію. Скрипт виконує шифрування геному в Json файл (рис. 3.9)

```
Ссылок: 0
public static void SaveGenome(NeatGenome genome)
{
    NeatGenomeJson genomeJson = new NeatGenomeJson();
    foreach (NodeGene node in genome.nodeGenes)
    {
        NodeGeneJson nodeJson = new NodeGeneJson();
        nodeJson.id = node.id;
        nodeJson.type = (NodeGeneJson.TYPE)node.type;
        genomeJson.nodeGenes.Add(nodeJson);
    }
    foreach (ConGene con in genome.conGenes)
    {
        ConGeneJson conJson = new ConGeneJson();
        conJson.inputNode = con.inputNode;
        conJson.outputNode = con.outputNode;
        conJson.weight = con.weight;
        conJson.isActive = con.isActive;
        conJson.innovNum = con.innovNum;
        genomeJson.conGenes.Add(conJson);
    }

    string json = JsonUtility.ToJson(genomeJson);
    File.WriteAllText(Application.dataPath + "/save.txt", json);
    print(json);
}
```

Рисунок 3.9 – Функція збереження в файл

Для завантаження збереження функція LoadGenome розшифровує файл і розподіляє всю інформацію між іншими скриптами (рис. 3.10)

```
Ссылок: 0
public static NeatGenome LoadGenome()
{
    string genomeString = File.ReadAllText(Application.dataPath + "/save.txt");
    NeatGenomeJson savedGenome = JsonUtility.FromJson<NeatGenomeJson>(genomeString);
    NeatGenome loadedGenome = new NeatGenome();
    foreach (NodeGeneJson savedNode in savedGenome.nodeGenes)
    {
        NodeGene newNode = new NodeGene(savedNode.id, (NodeGene.TYPE)savedNode.type);
        loadedGenome.nodeGenes.Add(newNode);
    }
    foreach (ConGeneJson savedCon in savedGenome.conGenes)
    {
        ConGene newCon = new ConGene(savedCon.inputNode, savedCon.outputNode, savedCon.weight, savedCon.isActive, savedCon.innovNum);
        loadedGenome.conGenes.Add(newCon);
    }

    return loadedGenome;
}
```

Рисунок 3.10 – Функція зчитування файлу

Після створення мережі та прив’язки до агенту, можемо запустити та почати навчати агента (рис. 3.11). Приблизно 100 000 кроків потрібно для того, щоб досягти першого успіху в вірному вирішення завдань агентом в симуляції.

Рисунок 3.11 – Перші кроки нерйонної мережі

3.3 Підготовка проекту до тестування

Після завершення написання програмної частини, проект має пройти тестування. Для запуску застосунку потрібно спочатку налаштувати пристрій на вірну роботу.

Перш за все, встановимо відповідне програмне забезпечення для запуску застосунку. Raspberry Pi 4 має можливість встановлення в пам’ять мікрокомп’ютера різні операційні системи: Raspberry PI, Ubuntu, OSMC, RetroPie та інші [4].

Встановлюємо Ubuntu (рис. 3.12)

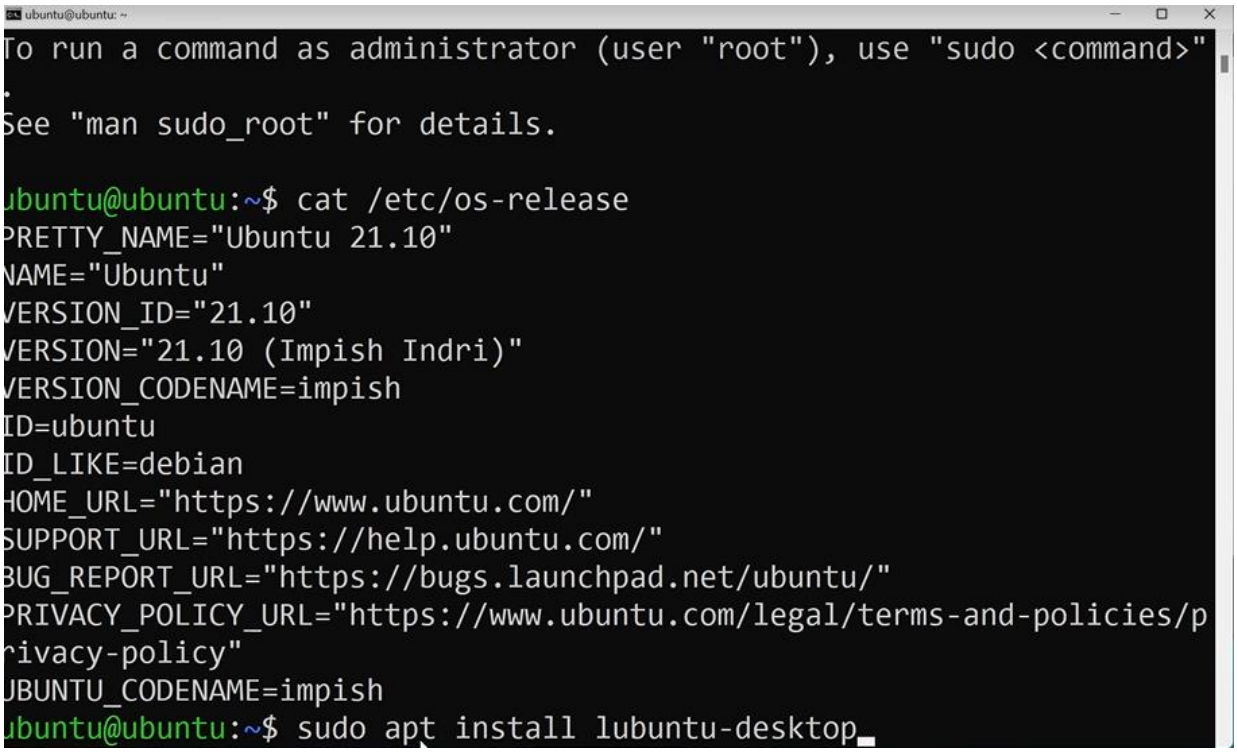


Рисунок 3.12 – Вставлення операційної системи

Наведені вище системи базуються на Linux, тому правильним рішенням є створення дистрибутиву для відповідної системи. Unity має можливість компілювати програму для різних ОС [18]. Обираємо Linux (рис. 3.13).

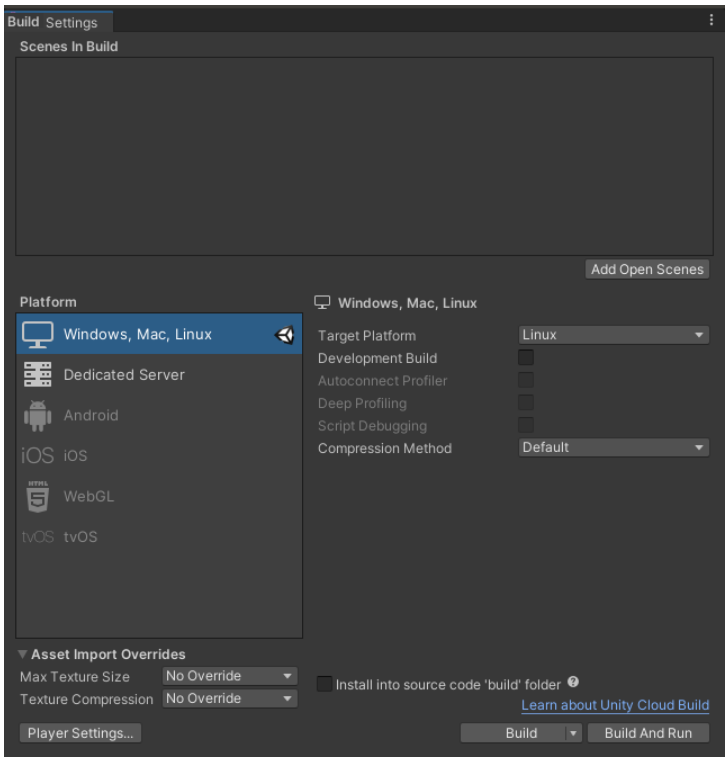


Рисунок 3.13 – Налаштування під збірку для Linux

В результаті збірки проекту отримуємо файл формату .x86, що є характерним для ОС Linux.

Оскільки керування застосунком засновано лише на використання UI елементів, то робити портування на інші пристрої під інше керування (наприклад: телефон), не потрібно.

Висновки до розділу 3

В даному розділі було розроблено застосунок на ігровому рушію Unity. Під час розробки симулятору було реалізовано технологію нейронної мережі. Використано метод навчання з підкріпленням та генетичний алгоритм. Після створення застосунку, його було адаптовано під мікрокомп’ютер Raspberry Pi 4. Для роботи з Raspberry, встановлено та налаштовано систему Ubuntu.

Перевагою в розробці виникла в розробці інтерфейсу; простота керування симулятором, значно заощадила час для реалізації проекту, тому що портування під інші пристрої потребує адаптації схеми керування застосунком.

4 ТЕСТУВАННЯ АПАРАТНО-ПРОГРАМНОГО ЗАСТОСУНКУ

4.1 Тестування готового застосунку

Для оцінки працездатності симулятору варто провести декілька тестів і по можливості зробити оптимізацію продукту. У Unity існує безліч різних тестових фреймворків та інструментів для написання та запуску тестів. Деякі з найпоширеніших видів тестів у Unity включають:

– unit-тести: тестування окремих блоків коду, таких як функції, методи і класи, щоб переконатися, що вони працюють правильно.

– інтеграційні тести: тестування взаємодії між кількома блоками коду або системами, щоб переконатися, що вони працюють правильно.

– acceptance-тести: тестування, яке перевіряє, чи додаток відповідає вимогам замовника та користувачів.

– ui-тести: тестування інтерфейсу користувача, щоб переконатися, що він працює правильно і відповідає заданим вимогам.

– performance-тести: тестування продуктивності програми, щоб переконатися, що вона працює досить швидко та ефективно.

– стрес-тести: тестування програми з високим навантаженням, щоб переконатися, що вона може обробляти велику кількість запитів без збоїв.

Для написання тестів у Unity можна використовувати такі інструменти як NUnit, Unity Test Runner, TestComplete та інші.

Проведемо Unit-тест, адже він найдоступніший в реалізації. Підготовимо середовище до тесту натиснувши Window – General – Test Runner (рис. 4.1).

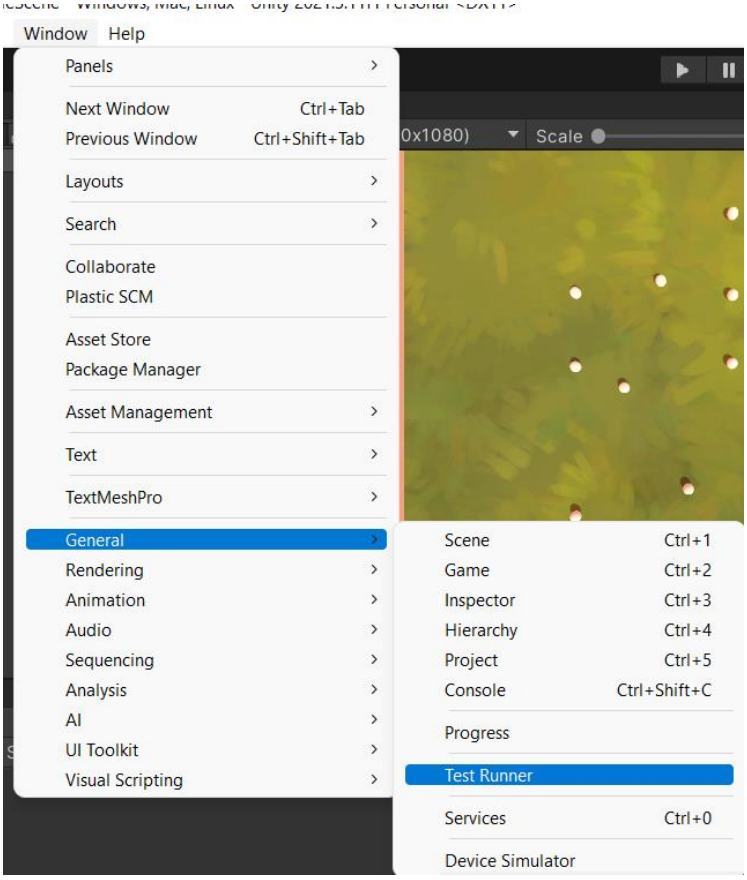


Рисунок 4.1– Виклик Test Runner’у

Після відкриття Test runner, створюється файл розширенням .asmdf (Assembly definition). В інспекторі середовища вказуємо завчасно створений файл для тесту (рис. 4.2).

Після цього можемо виконати тест на пересування агенту та його зір (Raycast) (рис. 4.3). В файлі tests.cs створимо декілька змін для перевірки на вірність дії. Атрибут [Test] означає, що саме зона (нижче атрибуту) буде позначена в інспекторі, як об’єкт тестування (рис. 4.4),(рис. 4.5).

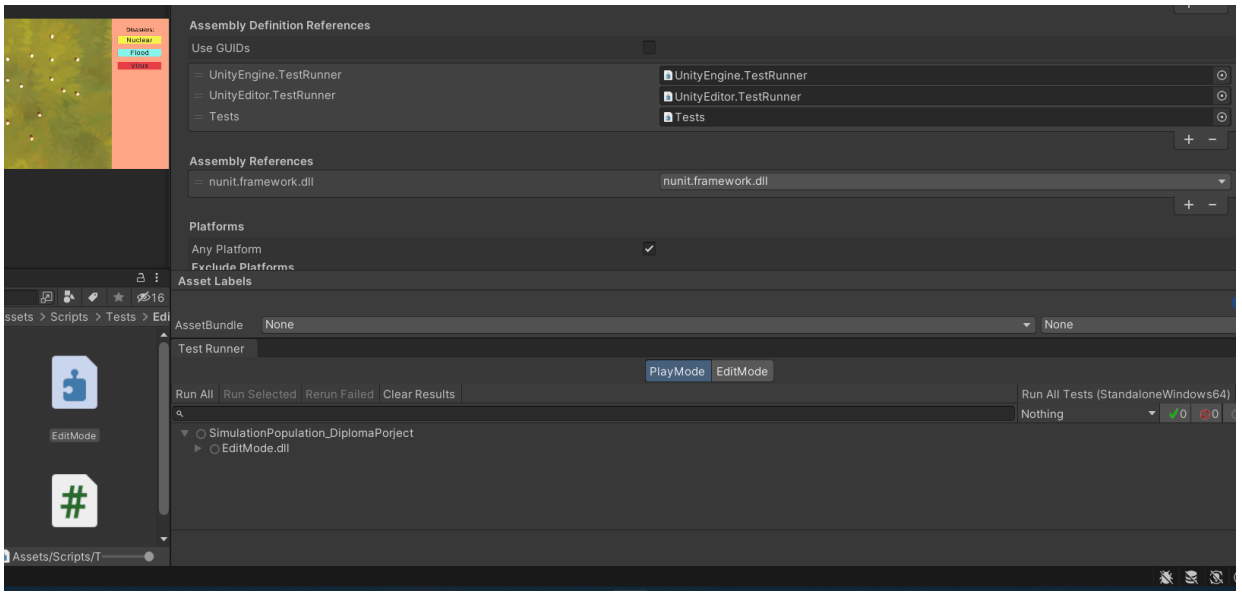


Рисунок 4.2 – Налаштування Tester Runner

```
Ссылок: 1
public void InputSensors()
{
    Ray rayCast = new Ray(transform.position, transform.right);
    RaycastHit hit;
    if (Physics.Raycast(rayCast,out hit, rayDistance))
    {
        if(hit.transform.tag=="Wall")
        {
            sensors[0] = hit.distance /hitDivider;
            Debug.DrawLine(rayCast.origin, hit.point, Color.red);
        }
    }
```

```
Ссылок: 1
public void MoveAgent(float v, float h)
{
    Vector3 input = Vector3.Lerp(Vector3.zero, new Vector3(v * 2f, 0, 0), 0.1f);
    input = transform.TransformDirection(input);

    //Movement of Agent
    transform.position += input;

    //Rotation of agent
    transform.eulerAngles += new Vector3(0, (h * 90) * 0.1f,0);
}
```

Рисунок 4.3 – Об’єкти тестування

```
Ссылок: 0
public class Tests
{
    bool isHit = true;
    bool isMoving = true;
    Vector3 MoveAgent = new Vector3(0, 0, 0);
    AgentController agent = new AgentController(new Vector3(0,0,0), true);
    // A Test behaves as an ordinary method
    [Test]
    Ссылок: 0
    public void RayCast_Test()
    {
        Assert.AreEqual(isHit, agent.hit);
    }
    [Test]
    Ссылок: 0
    public void AgentMove_Test()
    {
        if(MoveAgent.x <= agent.input.x ||
            MoveAgent.z <= agent.input.z)
        {
            Assert.IsTrue(isMoving);
        }

    }
}
```

Рисунок 4.4 - Тестування функцій методом Unit-тест (1)

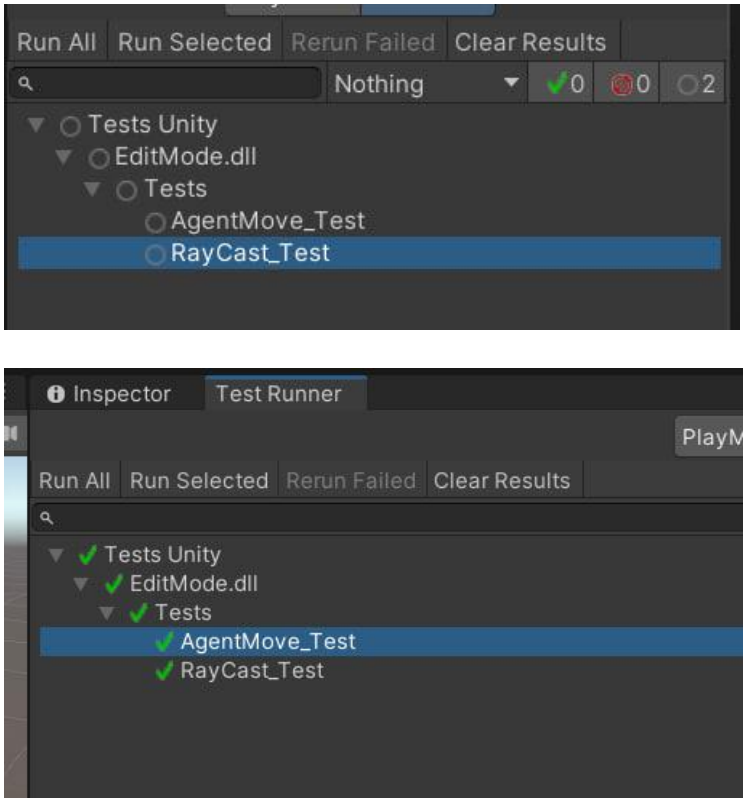


Рисунок 4.5 - Тестування функцій методом Unit-тест (2)

Як видно з результатів, тестування відбулось вдало. Обрані функції працюють корректно.

Метод стрес - тесту можливий при звичайних умовах роботи застосунку. Процес сильно навантажується, коли працює створення нових об’єктів в просторі. Агенти з’являються завдяки розмноженню на генетичному рівні, та при запуску сцени. Тому доречніше застосунок використовувати на пристроях з потужним процесором чи здатністю до обчислення великих об’ємів даних.

4.2 Оптимізація проекту

Під час роботи застосунку було помічено просідання кадрів, характерну сильному навантаженню компонентів мікрокомп’ютера. Тому, було вирішено провести оптимізацію проекту.

Для гарної роботи будь якого застосунку, варто виконати максимальну кількість варіантів оптимізації:

- код рефакторінг: перепроектування коду, переробка коду, рівносильне перетворення алгоритмів - процес зміни внутрішньої структури програми, що не зачіпає її зовнішньої поведінки і має на меті полегшити розуміння її роботи;
- налаштування сцени;
- налаштування текстур;
- налаштування освітлення;
- налаштування тіней.

У випадку рефракторінгу коду, було видалено всі стандартні методи, які створюються за замовчуванням при створені скрипту. Багато методів Unity запускаються самостійно, що викликають навантаження на процес. Тому, як наприклад методи Awake або Start запускаються відразу при ввімкнені сцени. Всі скрипти в проєкті мають за замовчуванням ці методи з самого початку [6][19][20].

На сцені можна обрати іншу якість відображення об’єктів. Це зменшить навантаження як на процесор так і на відеокарту (рис. 4.6).

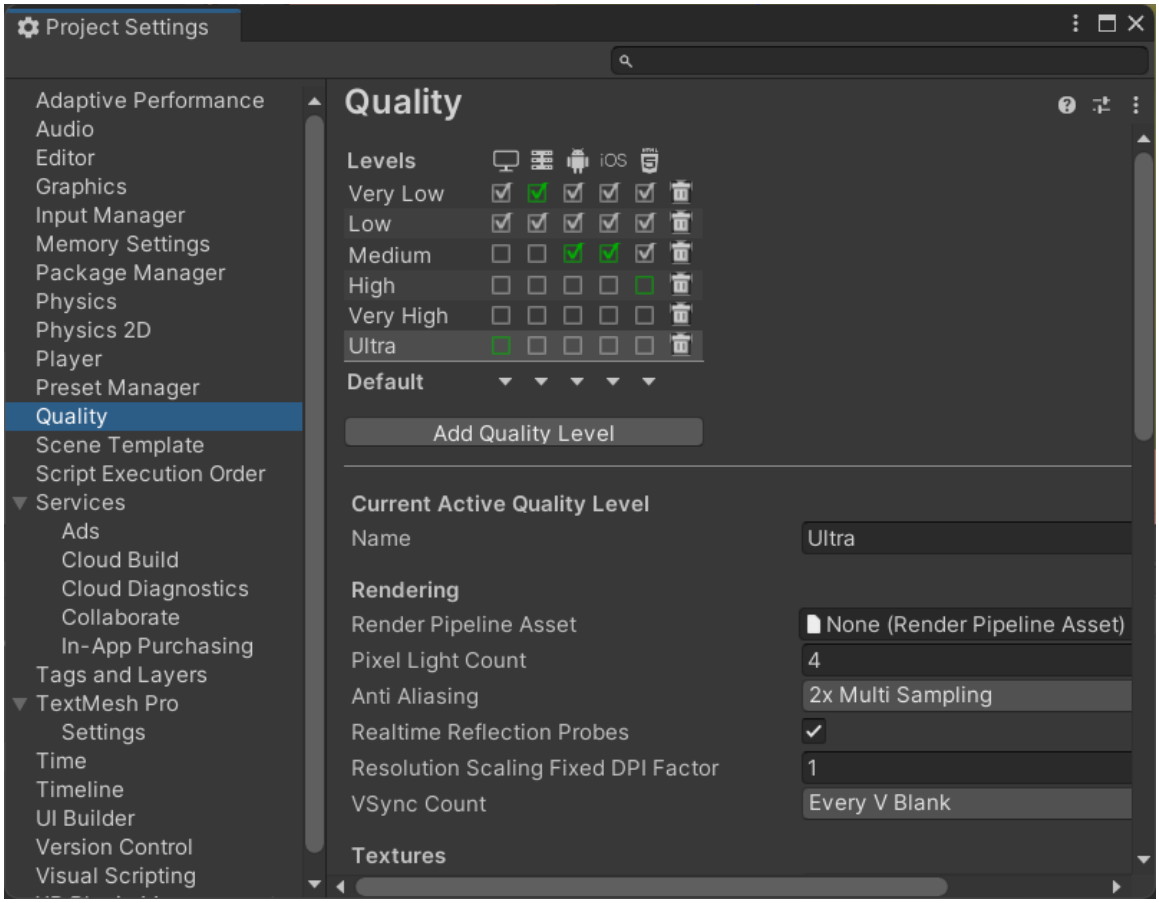


Рисунок 4.6 – Налаштування графіки на сцені

Оскільки текстури в основному відсутні в симуляції, перейдемо до наступного варіанту оптимізації.

Тіні завжди навантажують відеокарту. Агенти продовжуть створюватись надалі, а тінь від них не зникає (рис. 4.7). Світло теж відіграє велику роль у оптимізації графіки, тому було використані наступні налаштування (рис. 4.8).

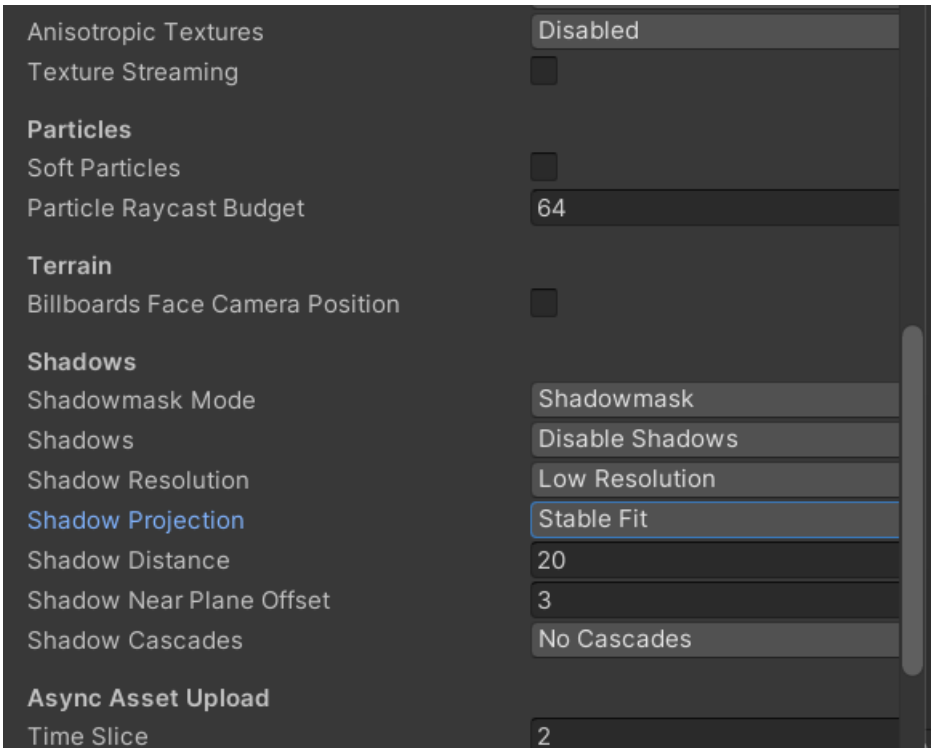


Рисунок 4.7 – Налаштування тіней

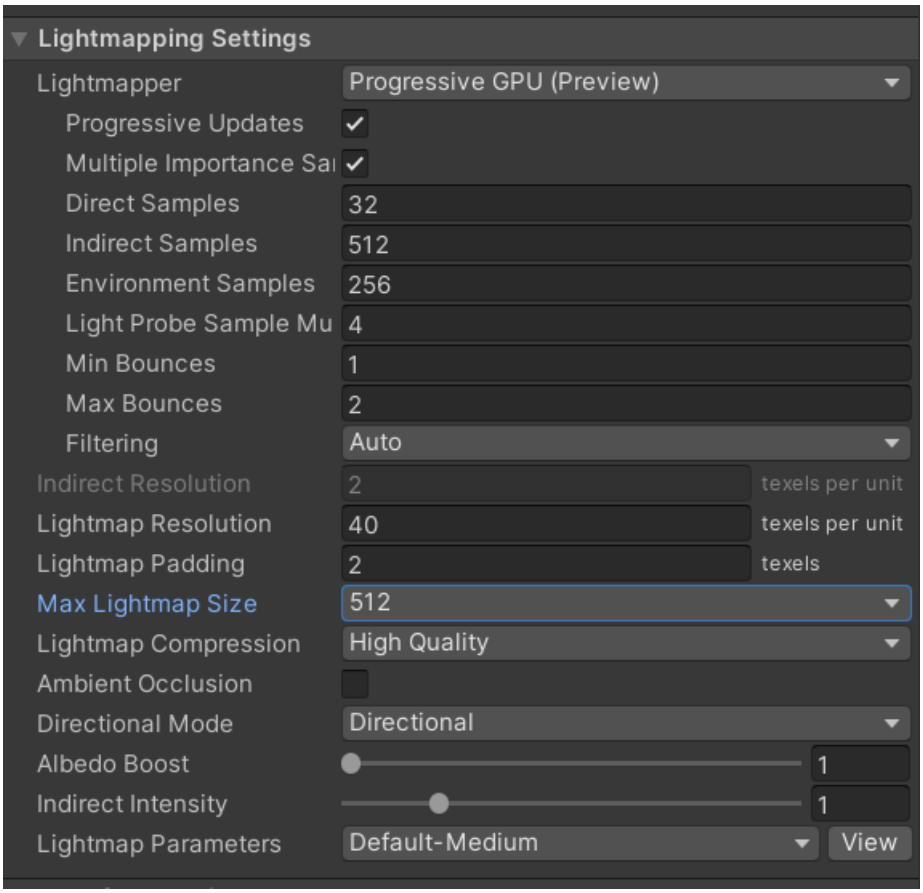


Рисунок 4.8 – Налаштування світла

Після застосуванні всіх варіантів оптимізації, було отримано хороший результат у вигляді великої кількості FPS (рис. 4.9).

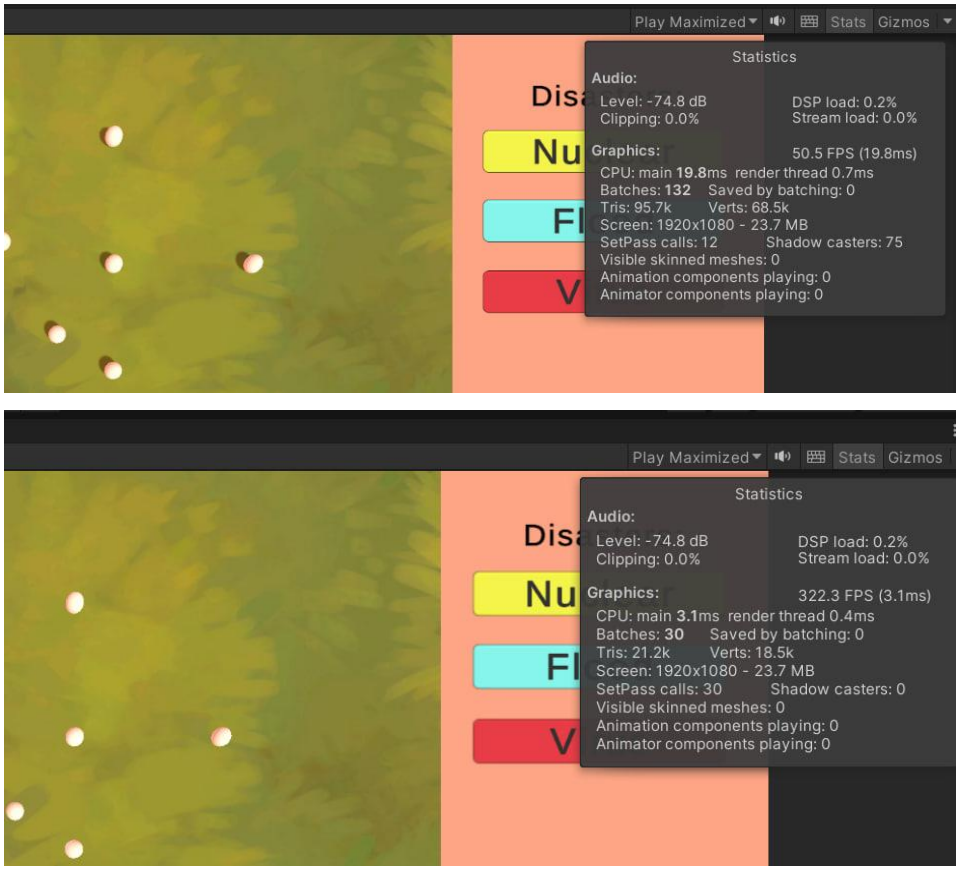


Рисунок 4.9 – Результати оптимізації застосунку

Проведення тестування на мікрокомп’ютері Raspberry Pi 4 виконується в режимі ручного тестування. Починаючи від корректного запуску програми, закінчуючи процесом розвитку подій в симуляторі.

Застосунок було протестовано на пристроях з слабкішими компонентами, але і на них він працював без нарікань.

Висновки до розділу 4

В цьому розділі були розглянути методи тестування застосунку в середовищі Unity. В якості тестування обрано метод Unit тестувань. Unit-тестування є найбільш доступним та швидким рішення для проведення тестування частин коду, функцій і методів.

Для покращення роботи, було проведено оптимізацію. Серед оптимізації, було виконано: Рефракторінг коду, налаштування сцени, налаштування графіки, світла та тіней. Тестування проводилось на різних багатоядерних пристроях різного рівня. В результаті тестування, застосунок пройшов перевірку працездатності.

ВИСНОВКИ

Кваліфікаційна робота пов’язана з розробкою програмного забезпечення для симуляції популяції живих організмів для застосування на багатоядерних пристроях. Була використана технологія нейронної мережі. Дана технологія має великий потенціал для розвитку та вже стала однією з провідних технологій нашого часу. Технологія стала невід’ємною частиною застосунків, які засновані на поведінці штучного інтелекту. Завдяки нейронним мережам можна модифікувати будь який апаратно-програмний проєкт. Мережа дає можливість відтворювати ситуації, чи вирішення завдань підвласну лише людському мозку.

У ході виконання кваліфікаційної роботи було вирішено ряд завдань, що описані нижче.

Проведено дослідження технології нейронної мережі. Нейронна мережа використовується для обробки зображень, голосове розпізнавання, симуляції поведінки, автоматичне перекладання та ігри з ШІ. Розглянуто всі типи нейронних мереж. Для реалізації технології було використано ігровий рушій та Raspberry Pi. Raspberry Pi чудово працює з великою кількістю обчислюванних даних. Враховуючи всі отримані дані, було прийнято рішення про створення застосунку, який буде застосовуватись для проведення тестів над віртуальними організмами. Це обумовлено малою кількістю програмної бази з використанням нейронних мереж для симуляції.

Визначено мету створення застосунку та вказано основні завдання при її розробці. Розроблено основні параметри системи та умови для її виконання. Обрано базові програмні засоби, що необхідні для розробки системи. В якості платформи для розробки було використано Unity. Дана платформа є безкоштовною у використанні для некомерційних проектів. Описано програмну реалізацію та наведено приклади етапів розробки застосунку. Створено застосунок та протестовано на різних апаратних засобах .

У розділі про охорону праці було досліджено основні нормативні документи, визначено норми умов праці робітника. Проаналізовано умови роботи та порівняно з нормами. Проведено розрахунок штучного освітлення робочого приміщення та надано рекомендації щодо вдосконалення робочого середовища для покращення умов праці.

Подальший розвиток розробленого застосунку полягає у розширені нейронної мережі, додавання нових об’єктів тестування. Перенесення застосунку на більш потужний мікрокомп`ютер; доробці функціональних можливостей взаємодії, збір більшої статистики.

Застосунок можна використовувати у навчальних закладах, медичних та наукових центрах для проведення тестів, які неможливо отримати за певних умов. При внесенні певних змін у роботу застосунку та за наявності експерта в різних галузей, його можна використовувати як повний симулятор людської поведінки на будь які зовнішні фактори.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Беззуб Є. С., Бойко А. П. Програмне забезпечення симуляції біопопуляції на основі нейронної мережі та ігрового рушія. Інформаційні технології та інженерія : тези доп. Всеукр. наук.-практ. конф. Миколаїв, 07–10 лют. 2023 р. Миколаїв : Чорном. нац. ун-т ім. Петра Могили, 2023. С. 92–94.

2. Штучний інтелект, машинне навчання - різниця, теорія і практика. Evergreen - web розробка і діджиталізація бізнесу за допомогою AI продуктів. URL: https://evergreens.com.ua/ua/articles/machine-learning-overview.html (дата звернення: 02.01.2023).

3. 9 types of neural networks: applications, pros, and cons. KnowledgeHut: Professional Bootcamps and Certification Courses Provider for your Future. URL: https://www.knowledgehut.com/blog/data-science/types-of-neural-networks (дата звернення: 14.01.2023).

4. Beginner's guide: how to get started with raspberry pi. PCMAG. URL: https://www.pcmag.com/how-to/beginners-guide-how-to-get-started-with-raspberry-pi (дата звернення: 14.02.2023).

5. Best unity AI tutorials – unity game development guide. GameDev Academy. URL: https://gamedevacademy.org/best-unity-ai-tutorials/ (дата звернення: 02.01.2023).

6. CG Cookie | Learn Blender, Online Tutorials and Feedback. CG cookie | learn blender, online tutorials and feedback. CG Cookie | Learn Blender, Online Tutorials and Feedback. URL: https://cgcookie.com/posts/maximizing-your-unity-games-performance (дата звернення: 14.02.2023).

7. Connor Shorten. Neuroevolution of augmenting topologies (NEAT), 2019. YouTube. URL: https://www.youtube.com/watch?v=b3D8jPmcw-g (дата звернення: 06.01.2023).

8. Constructing neural network-based models for simulating dynamical systems / C. M. Legaard та ін. ACM computing surveys. 2022. URL: https://doi.org/10.1145/3567591 (дата звернення: 14.02.2023).

9. davidrandallmiller. I programmed some creatures. they evolved., 2020. YouTube. URL: https://www.youtube.com/watch?v=N3tRFayqVtk (дата звернення: 01.02.2023).

10. Effects of nuclear weapons. atomicarchive.com: Exploring the History, Science, and Consequences of the Atomic Bomb. URL: https://www.atomicarchive.com/science/effects/index.html (дата звернення: 03.02.2023).

11. Innovative topologies and algorithms for neural networks. MDPI, 2021. URL: https://doi.org/10.3390/books978-3-0365-0285-4 (дата звернення: 14.02.2023).

12. Interfaces H. Learn how to implement NEAT AI in unity. Medium. URL: https://medium.com/@HolographicInterfaces/learn-how-to-implement-neat-ai-in-unity-157168eeae7e (дата звернення: 03.01.2023).

13. Massive computational acceleration by using neural networks to emulate mechanism-based biological models - Nature Communications. Nature. URL: https://www.nature.com/articles/s41467-019-12342-y (дата звернення: 01.01.2023).

14. Neural network models explained - seldon. Seldon. URL: https://www.seldon.io/neural-network-models-explained (дата звернення: 13.01.2023).

15. Nuclear weapon - Gun assembly, implosion, and boosting. Encyclopedia Britannica. URL: https://www.britannica.com/technology/nuclear-weapon/Gun-assembly-implosion-and-boosting (дата звернення: 11.02.2023).

16. Raycasts in unity, made easy - game dev beginner. Game Dev Beginner. URL: https://gamedevbeginner.com/raycasts-in-unity-made-easy/ (дата звернення: 10.02.2023).

17. Underpower Jet. Tutorial on programming an evolving neural network in C# w/ unity3d, 2017. YouTube. URL: https://www.youtube.com/watch?v=Yq0SfuiOVYE (дата звернення: 13.01.2023).

18. Unity - manual: linux build settings. Unity - Manual: Unity User Manual 2021.3 (LTS). URL: https://docs.unity3d.com/2022.2/Documentation/Manual/Buildsettings-linux.html (дата звернення: 08.02.2023).

19. Unity optimization tips – optimize unity game, tutorial 2023. Unity Assets & Unity Tutorials. URL: https://makaka.org/unity-tutorials/optimization (дата звернення: 14.02.2023).

20. Unity performance optimization VII: physics - UWA blog. UWA Blog. URL: https://blog.en.uwa4d.com/2022/03/01/unity-performance-optimization-VII-physics/ (дата звернення: 14.02.2023).

21. Unity - scripting API: physics.raycast. Unity - Manual: Unity User Manual 2021.3 (LTS). RL: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html (дата звернення: 07.02.2023).

22. Li M., Zhao Z., Scheidegger C. Visualizing neural networks with the grand tour. Distill. 2020. Т. 5, № 3. URL: https://doi.org/10.23915/distill.00025 (дата звернення: 14.02.2023).

23. Raspberry pi documentation - getting started. Raspberry Pi. URL: https://www.raspberrypi.com/documentation/computers/getting-started.html (дата звернення: 14.02.2023).

24. The Bibites: Digital Life. How I created an evolving neural network ecosystem, 2019. YouTube. URL: https://www.youtube.com/watch?v=myJ7YOZGkv0 (date of access: 04.02.2023).

ДОДАТОК А АПРОБАЦІЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Результати досліджень були представлені на конференції:

Інформаційні технології та інженерія: Всеукраїнська науковопрактична конференція молодих вчених, аспірантів і студентів : тези доп., 7–10 лютого 2023 р. / ЧНУ імені Петра Могили. Миколаїв, 2023 (рис. А.1), (рис. А.2).


Міністерство освіти і науки України
Чорноморський національний університет
імені Петра Могили

PETRO MOHYLA BLACK SEA NATIONAL UNIVERSITY
COMPUTER SCIENCE FACULTY
Est. 1996

«Інформаційні технології та інженерія»

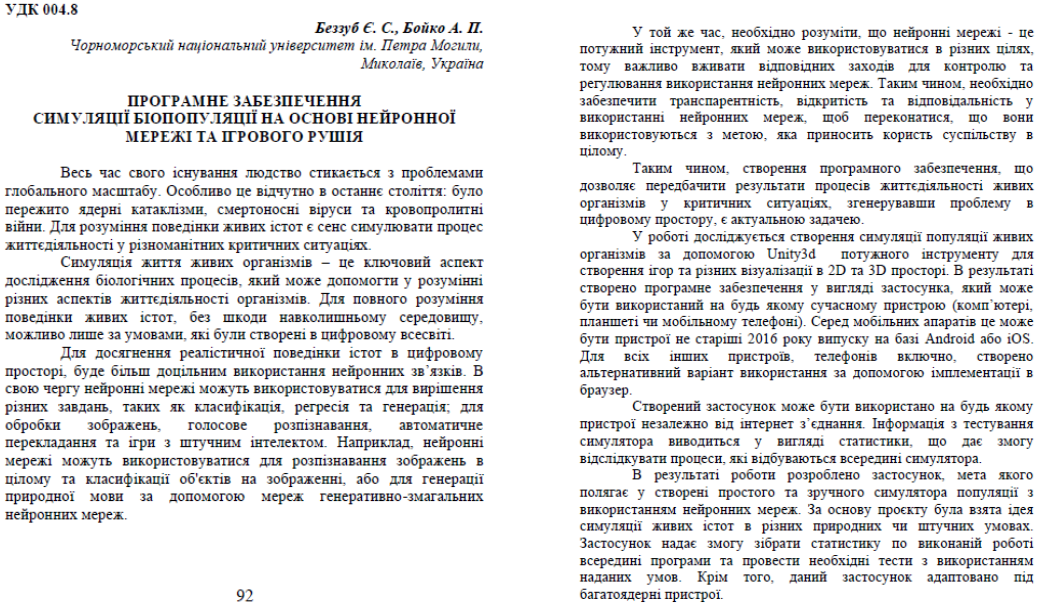
Всеукраїнська науково-практична конференція молодих вчених, аспірантів і студентів

ТЕЗИ

7–10 лютого 2023 року

Миколаїв – 2023

Рисунок А.1 – Обкладинка збірника тез конференції Інформаційні технології та інженерії – 2023



УДК 004.8

Беззуб Є. С., Бойко А. П.
Чорноморський національний університет ім. Петра Могили, Миколаїв, Україна

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИМУЛЯЦІЇ БІОПОПУЛЯЦІЇ НА ОСНОВІ НЕЙРОННОЇ МЕРЕЖІ ТА ІГРОВОГО РУШІЯ

Весь час свого існування людство стикається з проблемами глобального масштабу. Особливо це відчутно в останнє століття: було пережито ядерні катаклізми, смертоносні віруси та кровопролитні війни. Для розуміння поведінки живих істот є сенс симулювати процес життєдіяльності у різноманітних критичних ситуаціях.

Симуляція життя живих організмів – це ключовий аспект дослідження біологічних процесів, який може допомогти у розумінні різних аспектів життєдіяльності організмів. Для повного розуміння поведінки живих істот, без шкоди навколишньому середовищу, можливо лише за умовами, які були створені в цифровому всесвіті.

Для досягнення реалістичної поведінки істот в цифровому просторі, буде більш доцільним використання нейронних зв'язків. В свою чергу нейронні мережі можуть використовуватися для вирішення різних завдань, таких як класифікація, регресія та генерація; для обробки зображень, голосове розпізнавання, автоматичне перекладання та ігри з штучним інтелектом. Наприклад, нейронні мережі можуть використовуватися для розпізнавання зображень в цілому та класифікації об'єктів на зображенні, або для генерації природної мови за допомогою мереж генеративно-змагальних нейронних мереж.

92

У той же час, необхідно розуміти, що нейронні мережі - це потужний інструмент, який може використовуватися в різних цілях, тому важливо вживати відповідних заходів для контролю та регулювання використання нейронних мереж. Таким чином, необхідно забезпечити транспарентність, відкритість та відповідальність у використанні нейронних мереж, щоб переконатися, що вони використовуються з метою, яка приносить користь суспільству в цілому.

Таким чином, створення програмного забезпечення, що дозволяє передбачити результати процесів життєдіяльності живих організмів у критичних ситуаціях, згенерувавши проблему в цифровому простору, є актуальною задачею.

У роботі досліджується створення симуляції популяції живих організмів за допомогою Unity3d потужного інструменту для створення ігор та різних візуалізацій в 2D та 3D просторі. В результаті створено програмне забезпечення у вигляді застосунка, який може бути використаний на будь якому сучасному пристрою (комп'ютері, планшеті чи мобільному телефоні). Серед мобільних апаратів це може бути пристрої не старіші 2016 року випуску на базі Android або iOS. Для всіх інших пристроїв, телефонів включно, створено альтернативний варіант використання за допомогою імплементації в браузер.

Створений застосунок може бути використано на будь якому пристрої незалежно від інтернет з'єднання. Інформація з тестування симулятора виводиться у вигляді статистики, що дає змогу відслідкувати процеси, які відбуваються всередині симулятора.

В результаті роботи розроблено застосунок, мета якого полягає у створені простого та зручного симулятора популяції з використанням нейронних мереж. За основу проєкту була взята ідея симуляції живих істот в різних природних чи штучних умовах. Застосунок надає змогу зібрати статистику по виконаній роботі всередині програми та провести необхідні тести з використанням наданих умов. Крім того, даний застосунок адаптовано під багатоядерні пристрої.

Рисунок А.2 - Тези конференції Інформаційні технології та інженерія – 2023

ДОДАТОК Б ЛІСТИНГ КОДУ ЗАСТОСУНКУ

Лістинг файлу NeatNetwork.cs

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

public class NeatNetwork
{
    public NeatGenome MyGenome;
    public List<Node> nodes;
    public List<Node> inputNodes;
    public List<Node> outputNodes;
    public List<Node> hiddenNodes;
    public List<Connection> connections;

    public NeatNetwork(int inp, int outp, int hid)
    {
        MyGenome = CreateInitialGenome(inp,outp,hid);
        nodes = new List<Node>();
        inputNodes = new List<Node>();
        outputNodes = new List<Node>();
        hiddenNodes = new List<Node>();
        connections = new List<Connection>();
        CreateNetwork();
    }

    private NeatGenome CreateInitialGenome(int inp, int outp, int hid)
    {
        List<NodeGene> newNodeGenes = new List<NodeGene>();
        List<ConGene> newConGenes = new List<ConGene>();
        int nodeId = 0;

        for (int i = 0; i < inp; i++)
        {
            NodeGene newNodeGene = new NodeGene(nodeId, NodeGene.TYPE.Input);
            newNodeGenes.Add(newNodeGene);
            nodeId += 1;
        }
        for (int i = 0; i < outp; i++)
        {
            NodeGene newNodeGene = new NodeGene(nodeId, NodeGene.TYPE.Output);
            newNodeGenes.Add(newNodeGene);
            nodeId += 1;
```

```
            }
            for (int i = 0; i < hid; i++)
            {
                NodeGene newNodeGene = new NodeGene(nodeId, NodeGene.TYPE.Hidden);
                newNodeGenes.Add(newNodeGene);
                nodeId += 1;
            }
            ConGene newConGene1 = new ConGene(Random.Range(0,3), Random.Range(3, 5),
Random.Range(0f,1f), true, 0);
            newConGenes.Add(newConGene1);
            ConGene newConGene2 = new ConGene(Random.Range(0, 3), Random.Range(3, 5),
Random.Range(0f, 1f), true, 1);
            newConGenes.Add(newConGene2);
            ConGene newConGene3 = new ConGene(Random.Range(0, 3), Random.Range(3, 5),
Random.Range(0f, 1f), true, 2);
            newConGenes.Add(newConGene3);
            NeatGenome neatGenome = new NeatGenome(newNodeGenes,newConGenes);
            return neatGenome;
        }

        private void CreateNetwork()
        {
            foreach(NodeGene nodeGene in MyGenome.nodeGenes)
            {
                Node newNode = new Node(nodeGene.id);
                nodes.Add(newNode);
                if (nodeGene.type == NodeGene.TYPE.Input)
                {
                    inputNodes.Add(newNode);
                }
                else if(nodeGene.type == NodeGene.TYPE.Hidden)
                {
                    hiddenNodes.Add(newNode);
                }
                else if(nodeGene.type == NodeGene.TYPE.Output)
                {
                    outputNodes.Add(newNode);
                }
            }
            foreach (ConGene conGene in MyGenome.conGenes)
            {
                if(!conGene.isActive == true)
                {
                    Connection newCon = new Connection(conGene.inputNode,
conGene.outputNode, conGene.weight, conGene.isActive);
```

```
                    connections.Add(newCon);
                }


            }
            foreach(Node node in nodes)
            {
                foreach(Connection con in connections)
                {
                    if (con.InputNode == node.id)
                    {
                        node.outputConnection.Add(con);
                    }
                    else if(con.OutputNode == node.id)
                    {
                        node.inputConnection.Add(con);
                    }
                }
            }
            foreach (ConGene conGene in MyGenome.conGenes)
            {
                if (conGene.isActive == true)
                {
                    Connection newCon = new Connection(conGene.inputNode,
conGene.outputNode, conGene.weight, conGene.isActive);
                    connections.Add(newCon);
                }
            }
        }
        public float[] FeedForwardNetwork(float[] inputs)
        {
            float[] outputs = new float[outputNodes.Count];
            for(int i = 0; i < outputNodes.Count; i++)
            {
                inputNodes[i].SetInputNodeValue(inputs[i]);
                inputNodes[i].FeedForwardValue();
                inputNodes[i].value = 0;
            }
            for (int i = 0; i < hiddenNodes.Count; i++)
            {
                hiddenNodes[i].SetHiddenNodeValue();
                hiddenNodes[i].FeedForwardValue();
                hiddenNodes[i].value = 0;
            }
            for (int i = 0; i < outputNodes.Count; i++)
            {
```

```
                outputNodes[i].SetOutputNodeValue();
                outputs[i] = outputNodes[i].value;
                outputNodes[i].value = 0;
            }
            return outputs;
        }
    }
    public class Node
    {
        public int id;
        public float value;

        public List<Connection> inputConnection;
        public List<Connection> outputConnection;

        public Node(int ident)
        {
            id = ident;
            inputConnection= new List<Connection>();
            outputConnection= new List<Connection>();
        }
        public void SetInputNodeValue(float val)
        {
            val = Sigmoid(val);
            value = val;
        }
        public void SetHiddenNodeValue()
        {
           float val = 0;
            foreach(Connection con in inputConnection)
            {
                val += (con.Weight * con.InputNodeValue);
            }
            value = TanH(val);
        }
        public void SetOutputNodeValue()
        {
            float val = 0;
            foreach (Connection con in inputConnection)
            {
                val += (con.Weight * con.InputNodeValue);
            }
            value = TanH(val);
        }
        public void FeedForwardValue()
```

```
        {
            foreach(Connection con in outputConnection)
            {
                con.InputNodeValue = value;
            }
        }
        public float Sigmoid(float x)
        {
            return (1 / (1 + Mathf.Exp(-x)));
        }
        public float TanH(float x)
        {
            return ((2 / (1 + Mathf.Exp(-2*x))));
        }
        private float TanHMod(float x)
        {
            return((2/(1 + Mathf.Exp(-4*x)))-1);
        }
    }
    public class Connection
    {
        public int InputNode;
        public int OutputNode;
        public float Weight;
        public bool IsActive;
        public float InputNodeValue;
        public Connection(int inputNode, int outputNode, float weight, bool isActive)
        {
            this.InputNode = inputNode;
            this.OutputNode = outputNode;
            this.Weight = weight;
            this.IsActive = isActive;
        }
    }
```

Лістинг файлу NeatGenome.cs

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

public class NeatGenome
{
    public List<NodeGene> nodeGenes;
    public List<ConGene> conGenes;
```

```
        public NeatGenome()
        {
            nodeGenes = new List<NodeGene>();
            conGenes = new List<ConGene>();
        }
        public NeatGenome(List<NodeGene> nodeGenes, List<ConGene> conGenes)
        {
            this.nodeGenes = nodeGenes;
            this.conGenes = conGenes;
        }
    }
    public class NodeGene
    {
        public int id;
        public enum TYPE
        {
            Input,
            Output,
            Hidden
        }
        public TYPE type;
        public NodeGene(int givenID, TYPE givenType)
        {
            id = givenID;
            type = givenType;
        }
    }
    public class ConGene
    {
        public int inputNode;
        public int outputNode;
        public float weight;
        public bool isActive;
        public int innovNum;
        public ConGene(int inNode, int outNode, float weig, bool active, int innov)
        {
            inputNode = inNode;
            outputNode = outNode;
            weight = weig;
            isActive = active;
            innovNum = innov;
        }
    }
```

Лістинг файлу NeatUtilities.cs

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

public class NeatUtilities : MonoBehaviour
{
    public static void SaveGenome(NeatGenome genome)
    {
        NeatGenomeJson genomeJson = new NeatGenomeJson();
        foreach (NodeGene node in genome.nodeGenes)
        {
            NodeGeneJson nodeJson = new NodeGeneJson();
            nodeJson.id = node.id;
            nodeJson.type = (NodeGeneJson.TYPE)node.type;
            genomeJson.nodeGenes.Add(nodeJson);
        }
        foreach (ConGene con in genome.conGenes)
        {
            ConGeneJson conJson = new ConGeneJson();
            conJson.inputNode = con.inputNode;
            conJson.outputNode = con.outputNode;
            conJson.weight = con.weight;
            conJson.isActive = con.isActive;
            conJson.innovNum = con.innovNum;
            genomeJson.conGenes.Add(conJson);
        }

        string json = JsonUtility.ToJson(genomeJson);
        File.WriteAllText(Application.dataPath + "/save.txt", json);
        print(json);
    }

    public static NeatGenome LoadGenome()
    {
        string genomeString = File.ReadAllText(Application.dataPath + "/save.txt");
        NeatGenomeJson savedGenome =
JsonUtility.FromJson<NeatGenomeJson>(genomeString);
        NeatGenome loadedGenome = new NeatGenome();
        foreach (NodeGeneJson savedNode in savedGenome.nodeGenes)
        {
            NodeGene newNode = new NodeGene(savedNode.id,
(NodeGene.TYPE)savedNode.type);
```

```
                loadedGenome.nodeGenes.Add(newNode);
            }
            foreach (ConGeneJson savedCon in savedGenome.conGenes)
            {
                ConGene newCon = new ConGene(savedCon.inputNode, savedCon.outputNode,
savedCon.weight, savedCon.isActive, savedCon.innovNum);
                loadedGenome.conGenes.Add(newCon);
            }

            return loadedGenome;
        }
    }

    [System.Serializable]
    public class NeatGenomeJson
    {
        public List<NodeGeneJson> nodeGenes = new List<NodeGeneJson>();
        public List<ConGeneJson> conGenes = new List<ConGeneJson>();
    }

    [System.Serializable]
    public class NodeGeneJson
    {
        public int id;
        public enum TYPE
        {
            Input, Output, Hidden
        };
        public TYPE type;
    }

    [System.Serializable]
    public class ConGeneJson
    {
        public int inputNode;
        public int outputNode;
        public float weight;
        public bool isActive;
        public int innovNum;
    }
```

Лістинг файлу NeatManager.cs

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

public class NeatGManager : MonoBehaviour
{
    public GameObject NeatAgentPrefab;
    public GameObject[] allNeatAgent;
    public NeatNetwork[] allNeatNetworks;

    public int inputNodes, outputNodes, hiddenNodes;

    [SerializeField] private int currentGeneration = 0;

    public int startingPopulation;

    public int keepBest, leaveWorst;

    public int currentAlive;
    private bool repoping = false;

    public bool spawnFromSave = false;
    public int bestTime = 100;
    public int addToBest = 50;

    void Start()
    {
        allNeatAgent = new GameObject[startingPopulation];
        allNeatNetworks = new NeatNetwork[startingPopulation];

        if (spawnFromSave == true)
        {
            StartingSavedNetwork();
        }
        else
        {
            StartingNetworks();
        }

        MutatePopulation();
        SpawnBody();
        currentGeneration += 1;
    }
```

```
void FixedUpdate()
{
    currentAlive = CurrentAlive();
    if (repoping == false && currentAlive <= 0)
    {
        repoping = true;
        // Repopulate for next generation.
        RePopulate();
        repoping = false;
    }
}

public int CurrentAlive()
{
    int alive = 0;
    for (int i = 0; i < allNeatAgent.Length; i++)
    {
        if (allNeatAgent[i].gameObject)
        {
            alive++;
        }
    }
    return alive;
}

private void RePopulate()
{
    if (spawnFromSave == true)
    {
        bestTime = bestTime + addToBest;
        StartingSavedNetwork();
    }
    else
    {
        SortPopulation();
        SetNewPopulationNetworks();
    }
    MutatePopulation();
    GameObject.FindObjectOfType<FoodManager>().DestroyFood();
    GameObject.FindObjectOfType<FoodManager>().SpawnFood();
    SpawnBody();
    currentGeneration += 1;
}
```

```
private void MutatePopulation()
{
    for (int i = keepBest; i < startingPopulation; i++)
    {
        allNeatNetworks[i].MutateNetwork();
    }
}
private void SortPopulation()
{
    for (int i = 0; i < allNeatNetworks.Length; i++)
    {
        for (int j = i; j < allNeatNetworks.Length; j++)
        {
            if (allNeatNetworks[i].fitness < allNeatNetworks[j].fitness)
            {
                NeatNetwork temp = allNeatNetworks[i];
                allNeatNetworks[i] = allNeatNetworks[j];
                allNeatNetworks[j] = temp;
            }
        }
    }
}

public void BestFound()
{
    spawnFromSave = true;

}

private void SetNewPopulationNetworks()
{
    NeatNetwork[] newPopulation = new NeatNetwork[startingPopulation];
    for (int i = 0; i < startingPopulation - leaveWorst; i++)
    {
        newPopulation[i] = allNeatNetworks[i];
    }
    for (int i = startingPopulation - leaveWorst; i < startingPopulation; i++)
    {
        newPopulation[i] = new NeatNetwork(inputNodes, outputNodes, hiddenNodes);
    }
    allNeatNetworks = newPopulation;
}

private void StartingNetworks()
{
```

```
            for (int i = 0; i < startingPopulation; i++)
            {
                allNeatNetworks[i] = new NeatNetwork(inputNodes, outputNodes, hiddenNodes);
            }
        }

        private void StartingSavedNetwork()
        {

            spawnFromSave = false;
            for (int i = 0; i < startingPopulation; i++)
            {
                allNeatNetworks[i] = new NeatNetwork(NeatUtilities.LoadGenome());
            }
        }

        private void SpawnBody()
        {
            for (int i = 0; i < startingPopulation; i++)
            {
                Vector3 pos = new Vector3(Random.value, Random.value, 20);
                pos = Camera.main.ViewportToWorldPoint(pos);

                allNeatAgent [i] = Instantiate(NeatAgentPrefab, pos, transform.rotation);
                allNeatAgent[i].gameObject.GetComponent<FishController>().myBrainIndex = i;
                allNeatAgent[i].gameObject.GetComponent<FishController>().myNetwork =
allNeatNetworks[i];
                allNeatAgent[i].gameObject.GetComponent<FishController>().inputNodes =
inputNodes;
                allNeatAgent [i].gameObject.GetComponent<FishController>().outputNodes =
outputNodes;
                allNeatAgent [i].gameObject.GetComponent<FishController>().hiddenNodes =
hiddenNodes;
            }
        }
        public void Death(float fitness, int index)
        {
            allNeatNetworks[index].fitness = fitness;
        }
    }
```