Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет електроніки та комп'ютерних технологій
Кафедра системного проектування

ГЕНЕТИЧНІ АЛГОРИТМИ В НАВЧАННІ КЕРУВАННЯ РУХОМ КОСМІЧНОГО КОРАБЛЯ В СПРОЩЕНІЙ МОДЕЛІ СОНЯЧНОЇ СИСТЕМИ

Курсова робота

Виконавець
студент 1-го курсу магістратури
групи ФЕІм-12
Прецель В. О.
Науковий керівник
доц. Шувар Р. Я.

Львів – 2023

АНОТАЦІЯ

Курсова робота на тему: “Генетичні алгоритми в навчанні керування рухом космічного корабля в спрощеній моделі Сонячної системи”. В роботі описано процес створення двовимірної симуляції планетної системи разом із реалізацією навчання нейромережі за допомогою генетичних алгоритмів. Наведено теоретичне обґрунтування використаного підходу в навчанні. В роботі розглянуто та проаналізовано результати навчання нейромережі.

ANNOTATION

Course work "Genetic algorithms in learning to control the movement of a spaceship in a simplified model of the solar system". The paper describes the process of creating a two-dimensional simulation of the planetary system along with the implementation of genetic algorithms used for training a neural network. The theoretical justification of the used approach is presented. The work includes analysis of the results of neural network learning.

ЗМІСТ

АНОТАЦІЯ 2
ВСТУП 5
1. ТЕОРЕТИЧНІ ВІДОМОСТІ 6
1.1. Генетичні алгоритми та нейронні мережі 6
1.1.1. Генетичні алгоритми 6
1.1.2. Представлення хромосом 7
1.1.3. Пропорційний відбір 7
1.1.4. Оператори схрещування 8
1.1.5. Оператори мутації 10
1.1.6. Нейронні мережі 11
1.1.7. Навчання нейронної мережі в генетичних алгоритмах 12
1.2. Використані технології 12
1.2.1. Unity 12
1.2.2. C# 13
2. ПОСТАНОВКА ЗАДАЧІ 14
2.1. Загальний опис задачі 14
2.2. Симуляція спрощеної Сонячної системи 14
2.3. Генетичні алгоритми в тренуванні нейромережі 16
3. РЕАЛІЗАЦІЯ ПРОЕКТУ 18
3.1. Створення симуляції планетної системи 18
3.1.1. Гравітація 18
3.1.2. Рух ракети 20
3.1.3. Сцена та камери 21
3.1.4. Параметри об’єктів та перевірка симуляції 22
3.2. Реалізація нейронної мережі 23
3.3. Реалізація генетичних алгоритмів 25
3.4. Реалізація навчання нейромережі 28
4. ПРОВЕДЕННЯ ДОСЛІДІВ 30

4.1. Навчання польоту при однакових стартових положеннях 30
4.1.1. Опис задачі 30
4.1.2. Налаштування навчання 30
4.1.3. Результати навчання 31
4.2. Навчання польоту із декількома симуляціями на особину 32
4.2.1. Опис задачі 32
4.2.2. Налаштування навчання 33
4.2.3. Результати навчання 34
4.3. Навчання польоту зі змінними межами мутації 35
4.3.1. Опис задачі 35
4.3.2. Налаштування навчання 35
4.3.3. Результати навчання 36
ВИСНОВКИ 38
ВИКОРИСТАНІ ДЖЕРЕЛА 41
ДОДАТКИ 42

ВСТУП

Генетичні алгоритми – це один із видів еволюційних алгоритмів, особливістю яких є використання оператора схрещення як основного механізму пошуку. Такі алгоритми можуть бути використанні для багатьох задач, такі як оптимізація функцій, симуляція штучного життя, а також навчання нейронних мереж.

За мету даної роботи поставлено вивчення засобів та підходів до навчання нейронної мережі за допомогою еволюційних алгоритмів на прикладній задачі, яка відображає явища в реальному світі, хоч і в спрощеному вигляді. Такою задачею було вибрано керування польотом космічного корабля у планетній системі під постійною дією гравітації планет.

Завдання роботи є комплексним і навчанню нейромережі передує створення симуляції руху планет та інших тіл у сонячній системі. Принциповим моментом тут є реалізація руху тіл за законом всесвітнього тяжіння та дотримання принаймні приблизно реалістичних пропорцій у відстанях та масах тіл, аби симуляція була наближена до реальності. Вже в створеній симуляції відбуватиметься навчання руху космічного корабля, для якого будуть визначені точки початку руху та його місця призначення.

Після проведення навчань нейромережі з різними вхідними даними проаналізовано результати та зроблено висновки щодо можливостей вибраного підходу у розв'язанні вибраної задачі.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. Генетичні алгоритми та нейронні мережі

1.1.1. Генетичні алгоритми

Генетичний алгоритм — це еволюційний алгоритм пошуку, що натхненний теорією природної еволюції. Цей алгоритм відображає процес природного відбору, коли для розмноження відбираються найпристосованіші особини з метою отримання якнайкращих нащадків наступного покоління. В генетичних алгоритмах можна виділити декілька етапів.

1) *Початкова популяція*. Першим кроком є генерування індивідів, які разом називають популяцією. Кожна особина – це рішення проблеми, яка досліджується. Індивід характеризується набором параметрів (змінних), відомих як гени. Гени об’єднуються в рядок, який називають хромосомою.
2) *Відбір*. Відбір полягає в тому, щоб відібрати найбільш пристосованих особин і дати їм можливість передати свої гени наступному поколінню. Робиться це за допомогою функції пристосованості, що відображає досліджувану проблему. Більші значення функції пристосованості означають кращу придатність індивіда, а відповідно і більші шанси передати його гени нащадкам.
3) *Схрещування (кросинговер)*. Під час схрещування відібрані індивіди обмінюються випадковим чином генами між собою для утворення нових хромосом. Схрещені хромосоми стануть основою для утворення нових нащадків.
4) *Мутація*. Отримані в попередньому кроці хромосоми підлягають випадковим змінам у генах, що дозволяє підтримувати різноманітність популяції а відповідно і розглядати більше рішень для проблеми. Нарешті отримані нащадки стають повноцінними особинами та входять до нового покоління, яке вже стає основою для нового відбору.

1.1.2. Представлення хромосом

На практиці зазвичай існує два типи представлення геному:

- двійковий, зі значенням 0 або 1;
- дійсні числа.

Канонічний генетичний алгоритм використовує двійкове представлення, оскільки він більше натхненний біологією, де еволюція відбувається в ДНК шляхом зміни хімічних сполук, наприклад перевертання бітів для мутації або кросинговеру геному. Однак двійкове представлення більше не використовується масово через проблему кодування та декодування чисел з плаваючою комою в двійкові.

З іншого боку, представлення з плаваючою комою залишає числа такими, як вони є, таким чином економлячи час, не кодуючи чи декодуючи в двійкове представлення. На практиці представлення з плаваючою комою може працювати так само добре, а іноді й краще, ніж стандартне двійкове кодування, тому немає причин турбуватися про якість алгоритму, якщо використовується представлення з плаваючою комою.

1.1.3. Пропорційний відбір

Пропорційний відбір, або рулеточний відбір (roulette-wheel selection) — оператор відбору, що вибирає індивідів за допомогою N випадкових виборів, де N – розмір популяції, а випадковий вибір індивіда робиться з ймовірністю пропорційною до пристосованості цього нащадка.

На початку рулеточного відбору здійснюють обрахунок відносного значення пристосованості індивідів, що буде ймовірністю його відбору до наступної популяції:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j},$$

де f_i – пристосованість індивіда, N – кількість особин у популяції. Після цього здійснюється N “запусків” колеса, де кожна секція відповідає окремому індивіду,

розмір кожної секції на колесі є більшим в залежності від відносного значення пристосованості індивіда. Таким чином, індивіди з кращою пристосованістю мають більшу ймовірність бути відібраними.

1.1.4. Оператори схрещування

Оператори кросинговеру можна класифікувати на три типи:

- нестатеві, де нащадки утворюються лише від одного з батьків;
- статеві, де у схрещування беруть участь хромосоми двох батьків для утворення одного або декількох нащадків;
- мультирекомбінаційні, із участю більше ніж двох батьківських хромосом.

Крім того, для відібраних батьків для створення нащадків враховується ймовірність виконання кросинговеру, щоб не втратити генетичну інформацію батьків через постійний кросинговер. Якщо схрещування не застосовується, то нащадок є копією найбільш пристосованого батька. Крім того, якщо народжується одне потомство, воно може замінити батьківську особину з найгіршою пристосованістю.

Для бінарного представлення хромосом існує три основні техніки кросинговеру.

1) *Одноточковий оператор*. Вибирається випадкова точка, яка ділить хромосоми обох батьків на дві секції. Хромосоми обох батьків обмінюються цими секціями, утворюючи нові хромосоми нащадків.
2) *Багатоточковий оператор.* Вибирається декілька точок для поділу хромосом на декілька секцій, і аналогічно із одноточковим оператором відбувається обмін секціями.
3) *Рівномірний оператор.* На відміну від попередніх операторів, ніяких поділів на секції не відбувається, а натомість для кожного гена є певна ймовірність, що відбудуться його обмін в хромосомах.

У випадку представлення з плаваючою комою існує два основних типи методи схрещування.

В першому методі хромосоми використовуються як вектори, для яких обчислюється середній вектор від батьківських особин. Потім до знайденого середнього вектора додається певне значення дисперсії у напрямку до одного з батьків, щоб отримати нащадків. Таким чином, нащадки є певним усередненням батьківських особин. Для обрахунку середнього між двома батьківськими хромосомами можна використовувати наступну формулу:

$$\tilde{x}_{ij} = (1-\gamma)x_{1j} + \gamma x_{2j},$$

$$\gamma \in [0,1], \gamma \sim N(0.5, 0.15) \text{ або } \gamma \sim U(0,1),$$

де x_1 та x_2 – батьківські особини для і-го нащадка $\tilde{x}_i$, де j – позиція гена в хромосомах.

Якщо взяти гамму рівну 0.5, лінійна комбінація хромосом є середнім арифметичним між двома батьками. Загальноприйнято або використовувати випадкове значення гамми з нормальним розподілом із центром в 0.5 зі стандартним відхиленням 0.15, аби значення гамми могло коливатись від 0.05 до 0.95, або ж використовувати випадкове значення з рівномірним розподілом від 0 до 1.

Даний тип схрещування можна розширити для кількох батьків, де кожне батьківське значення має власне гамма-значення, де сума гамма-значень дорівнює одиниці:

$$\tilde{x}_{ij} = \sum \gamma_p x_{pj}, \sum \gamma_p = 1.$$

Проте при використанні кількох батьків може бути складно вибрати значення γ. Якщо всі значення γ рівні, забезпечується хороша різноманітність популяції, однак можна відбирати батьків на основі їх значень пристосованості та зробити так, щоб батьківська особина з найкращим значенням мала найбільше значення γ, аби нащадки збігались саме до цього батька, проте, якщо значення γ занадто високе, це може призвести до передчасної конвергенції. На практиці для

схрещування з декількома батьківськими особинами зазвичай використовується середнє арифметичне.

В другому методі схрещування хромосома нащадка утворюється через обмін генами, тобто чисел з плаваючою комою, через що значення самих генів не змінюється на відміну від попереднього методу. Цей тип кросинговеру має лише дискретну кількість можливих значень, оскільки при створенні потомства немає жодної варіації, лише змінні значення, наявні в батьків. Це може бути проблемою, оскільки немає способу представити новий генетичний матеріал, однак ця проблема може бути розв'язана за рахунок мутації.

Для двох батьків формула знаходження хромосоми нащадка має такий вигляд:

$$\tilde{x}_{ij} = x_{1j} \text{ або } x_{2j},$$

де один із батьківських генів x_{1j} та x_{2j} вибирається випадковим чином для його присвоєння $\tilde{x}_{ij}$.

Як і в попередньому методі, можна розширити схрещування до використання кількох батьківських особин:

$$\tilde{x}_{ij} = x_{pj},$$

де x_{pj} – випадково обраний ген одного із батьків. У такому підході питання полягає в тому, ген якого з батьків має успадкувати нащадок, це називається стратегією сканування. Ймовірність успадкування гена від одного з батьків може бути рівномірною або ґрунтуватися на їхніх значеннях пристосованості за допомогою методу пропорційного відбору.

1.1.5. Оператори мутації

Для введення у популяцію нового генетичного матеріалу використовують мутації. Для двійкового представлення зазвичай використовують два основних способи виконання мутації: через рівномірну мутацію, коли кожен біт має однакові шанси бути перевернутим, або лише випадковим перевертанням одного з бітів.

Для представлення хромосоми з плаваючою комою існує багато різних методів мутації. Основний метод мутації полягає у додаванні випадкового значення з рівномірним розподілом до кожного гена нащадка в межах області значень цього гена. Якщо ці межі широкі, це сприятиме дослідженню – знаходження тих індивідів, які найкраще підходять для розв’язку завдання, а якщо вона мала, то це сприятиме експлуатації – покращенню результатів для знаходження найкращого можливого рішення. Вибір максимального значення мутації залежить від проблеми, але зазвичай воно становить близько 1% від загальної області значень для заданого гена.

1.1.6. Нейронні мережі

Нейронні мережі — є обчислювальними системами, що натхненні реальними біологічними скупченнями нейронів, що утворюють мозок тварин. Нейромережі утворені із сукупності з’єднаних вузлів, які називаються нейронами, що приблизно моделюють нейрони мозку. Кожне з’єднання, як синапси в мозку, може передавати сигнал до інших нейронів. Нейрон отримує сигнали, потім обробляє їх і може сигналізувати підключеним до нього нейронам. «Сигнал» у з’єднанні є дійсним числом, а вихід кожного нейрона обчислюється за допомогою деякої нелінійної функції суми його входів. Нейрони та з’єднання зазвичай мають вагу, яка коригується в міру навчання. Вага збільшує або зменшує силу сигналу під час підключення. Нейрони можуть мати такий поріг, що сигнал надсилається лише тоді, коли сукупний сигнал перетинає цей поріг.

Як правило, нейрони утворюють шари. Різні шари можуть виконувати різні перетворення своїх вхідних даних. Сигнали проходять від першого шару (вхідний рівень) до останнього (вихідний рівень).

1.1.7. Навчання нейронної мережі в генетичних алгоритмах

Існує велика кількість способів навчання нейронних мереж, де значення ваг змінюється за рахунок обчислення похибки і її врахування для обчислення нового значення ваг. У випадку тренування нейромережі з генетичними алгоритмами, ваги змінюватимуться в ході еволюції. Ваги найромережі записуються в гени хромосом. Таким чином, при відборі найпристосованіших хромосом отримуватимемо популяцію індивідів із нейромережами, що найкраще підходитимуть для розв’язання досліджуваної проблеми.

1.2. Використані технології

1.2.1. Unity

Unity – це кросплатформний ігровий рушій, що може бути використаний для створення тривимірних і двовимірних ігор, а також для інтерактивного моделювання, симуляцій тощо. Рушій використовується й іншими галузями окрім відеоігор, такими як кіно, автомобілебудування, архітектура, інженерія, будівництво та багато інших.

Unity дає користувачам можливість створювати ігри та симуляції як у двох вимірах, так і в трьох вимірах, також рушій надає основний API сценаріїв у C# з використанням Mono як для редактора Unity у формі плагінів, так і для самих ігор.

У 2D-іграх Unity дозволяє імпортувати спрайти та надає розширені можливості для рендерингу 2D-світу. Для 3D-ігор Unity дозволяє налаштовувати стиснення текстур, MIP-мапи та роздільну здатність для кожної платформи, яка підтримує ігровий рушій, а також забезпечує підтримку відображення рельєфу, віддзеркалень та паралаксу, динамічні тіні та інші можливості.

1.2.2. C#

C# – об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи .NET. Останньою стабільною версією є C# 11.0, що вийшов разом з релізом .NET 7.

Синтаксис C# близький до C++ і Java. Мова має строгу статичну типізацію, підтримує поліморфізм, перевантаження операторів, вказівники на функції-члени класів, атрибути, події, властивості, винятки, коментарі у форматі XML. C# перейняв багато від своїх попередників: мов C++, Object Pascal і Smalltalk. Спираючись на практику їхнього використання, C# виключає деякі концепції, що зарекомендували себе як проблематичні при розробці програмних систем, наприклад, мова C#, на відміну від C++, не передбачає множинне успадкування класів.

В Unity C# використовується як мова для написання скриптів, що потім прикріплюються до об’єктів на сцені для опису їх поведінки. Для того щоб запустити саму симуляцію не потрібно писати жодного коду, оскільки Unity бере виконання програми на себе.

2. ПОСТАНОВКА ЗАДАЧІ

2.1. Загальний опис задачі

Задачу можна сформулювати таким чином: створити проект, в якому за допомогою генетичних алгоритмів відбуватиметься навчання нейромережі, що керуватиме польотом космічного корабля у спрощеній симуляції планетної системи, та проаналізувати отримані результати. Відповідно, завдання є комплексним і потребує створення симуляції планетної системи з врахуванням законів фізики, яким підпорядковується рух тіл в цій системі, а також реалізації нейромережі та генетичних алгоритмів для керування рухом космічного корабля. В поставленій задачі будемо розглядати навчання руху ракети від Землі до Місяця.

2.2. Симуляція спрощеної Сонячної системи

Першим етапом роботи є створення симуляції, яка відображатиме реальне середовище для прикладної задачі, яка розглядається. Відповідно, необхідно створити симуляцію руху тіл у планетній системі.

Реальний світ є тривимірним, проте можна звернути увагу, що планети рухаються навколо Сонця по еліптичних орбітах і перебувають в одній площині. Тому для спрощення задачі доречно розглядати ці тіла у тривимірному просторі.

Рух тіл у планетній системі відбувається під дією сили гравітації: планети рухаються по еліптичних орбітах, а об'єкти, що пролітають повз, змінюють траєкторію руху. Тому для створення реалістичної симуляції необхідно використати закон всесвітнього тяжіння:

$$F_{12} = G\frac{m_1 m_2}{r_{12}^2},$$

де F_{12} – сила, що діє на тіло, G – гравітаційна стала, m_1, m_2 – маси тіл, r_{12} – відстань між тілами. Для значення гравітаційної сталої доречно взяти одиницю, як це робиться у природній системі одиниць для спрощення математичних розрахунків.

У поставленій задачі розглядається рух ракети від Землі до Місяця, тому для реалістичності симуляції варто використати величини пропорційні до їх реальних величин.

Маса Землі	$5{,}9737\times10^{24}$ кг
Маса Місяця	$7{,}3477\times10^{22}$ кг
Середній радіус Землі	6371,3 км
Середній радіус Місяця	1737,1 км
Відстань між центрами Землі та Місяця в апогеї	406700 км
Відстань між центрами Землі та Місяця в перигеї	356500 км
Швидкість Місяця в апогеї	0.97 км/с
Швидкість Місяця в перигеї	1.082 км/с

Таблиця 1. Деякі фізичні характеристики Землі та Місяця

У рушію Unity немає поняття одиниць вимірювання, тому прийнято вважати, що одиниця є тим, що вважає розробник симуляції. Це спрощує задачу для задання відстані та маси тіл, проте необхідно з’ясувати як задати Місяцю рух навколо Землі. У Unity для застосування сили до тіла існує спеціальний метод “AddForce(Vector3 force, ForceMode mode)” із режимом “Impulse”. Саме цей метод і буде використовуватись для задання початкового імпульсу, лишилось обчислити значення параметра “force”. Сила F може бути обчислена за другим законом Ньютона:

$$F = m\,\frac{dv}{dt} = ma,$$

де a – прискорення, що застосоване до тіла масою m, щоб змінити його швидкість на dv за проміжок часу dt. Значення dt візьмемо рівне часу за який оновлюються кадри в Unity – 0.02 с. Необхідно пам'ятати про масштаб, який використовуємо в Unity, тому додатково розділимо на відстань між Землею та Місяцем, а також визначено, що необхідно домножити результат на 10^5. Крім того необхідно звернути увагу, що орбіта Місяця еліптична та має різні значення швидкості в апогеї та перигеї (найвіддаленіша та найближча точки орбіти

відповідно). Остаточно при прийнятому масштабі 1 одиниця маси = 10^{21} кг, 1 одиниця відстані = 10^{7} м, отримуємо таке значення сили, яке потрібно застосувати в програмі:

$$\frac{73.477 \times 0.97 \times 10^{-4}}{0.02 \times (40.67 + 0.68 + 0.17)} \times 10^{5} = 858.29.$$

Для космічного корабля, представленим ракетою, візьмемо мінімально можливі значення для маси та розміру. Доречно було б взяти значення, які були б пропорційні реальним прототипам ракет, але Unity тут накладає певні обмеження. Рухом ракети можна керувати додаючи йому прискорення за рахунок додавання імпульсу з використанням того ж методу “AddForce(Vector3 force, ForceMode mode)” із режимом “Impulse”. Ракета змінюватиме швидкість і буде маневрувати в залежності від вектора напрямлення сили “force”.

2.3. Генетичні алгоритми в тренуванні нейромережі

Нейромережа виконуватиме функцію мозку, що керуватиме ракетою на основі вхідних параметрів. Значеннями, що подаватимуться на вхід нейрона, будуть координати та вектори швидкості ракети та пункту призначення, тобто Місяця. Якби в планетній системі було більше тіл, їх також варто було б подавати на вхід мережі, оскільки вони б виконували роль перешкод для ракети, яка та мала б навчитись оминати. На виході ж отримуємо вектор та величину прискорення, які ракета використає для зміни швидкості та напрямку руху. Вхідні значення нейромережі нормалізуватимемо до одиниці, а вихідні будемо навпаки перетворювати з одиниці на необхідні значення.

Оскільки нейромережа працює з координатами, які можуть однаково набувати як від’ємних так і додатних значень, як функцію активації вирішено взяти функцію гіперболічного тангенсу, яка є непарною:

$$th(x) = \frac{sh(x)}{ch(x)} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}.$$

Кожен індивід в популяції буде мати свою нейромережу, ваги якої записуватимемо в хромосоми. Саме вони будуть еволюціонувати в процесі

роботи генетичного алгоритму, і як наслідок змінюватиметься поведінка ракет, що в результаті має привести до поступового покращення результатів.

Завданням ракети є долетіти до Місяця, тому функцію пристосованості f визначимо так, щоб більші значення отримувались при меншій відстані між Місяцем та ракетою, коли вона була в найближчій до Місяця точці:

$$f(x) = \frac{1}{x},$$

де x – відстань від ракети до Місяця у момент, коли ракета була до Місяця найближче.

3. РЕАЛІЗАЦІЯ ПРОЕКТУ

3.1. Створення симуляції планетної системи

3.1.1. Гравітація

Основним для симуляції дії гравітації буде скрипт “GravityHandler.cs” (див. лістинг 1), що оброблюватиме усі об’єкти на сцені зі скриптом “GravityTarget.cs”. Тут реалізовано метод “AddGravityForce()”, що обраховує значення сили, яка діє на тіла, із врахуванням напрямку сили. Для застосування цієї сили на об’єкт використовується метод “AddForce(Vector3 force, ForceMode mode)” із режимом “Force”. Функція “AddGravityForce()” запускається при кожному новому кадрі для постійної симуляції дії гравітації.

```
public class GravityHandler : MonoBehaviour
{
    [SerializeField] float g = 1;
    static float G;
    public static List<GravityTarget> attractors = new List<GravityTarget>();
    public static List<GravityTarget> attractees = new List<GravityTarget>();

    void FixedUpdate()
    {
            G = g;
            SimulateGravities();
    }

    public static void SimulateGravities()
    {
        foreach (GravityTarget attractor in attractors)
            foreach (GravityTarget attractee in attractees)
                if (attractor != attractee)
                        AddGravityForce(attractor.rigidBody, attractee.rigidBody);
    }

    public static void AddGravityForce(Rigidbody2D attractor, Rigidbody2D target)
    {
        double massProduct = attractor.mass * target.mass;
        Vector3 difference = attractor.position - target.position;
        float distance = difference.magnitude;

        double unScaledforceMagnitude = massProduct / Math.Pow(distance, 2);
        double forceMagnitude = G * unScaledforceMagnitude;

        Vector3 forceDirection = difference.normalized;
        Vector3 forceVector = forceDirection * (float)forceMagnitude;

        target.AddForce(forceVector);
    }
}
```

Лістинг 1. Клас GravityHandler

Хоча в реальному світі всі об’єкти беруть участь в гравітації, але в скрипті “GravityTarget.cs” додано властивості і декілька допоміжних методів для того, аби можна було маніпулювати тим, чи об’єкт притягує інші об’єкти, і чи може притягуватись сам. Також скрипт містить метод “ApplyVelocity()” (див. лістинг 2), для задання початкової швидкості тіла, і обробник колізій, який необхідний для відстеження зіткнень ракети з Місяцем чи Землею.

```
public class GravityTarget : MonoBehaviour
{
    void Start()
    {
        if (applyInitialVelocityOnStart)
        {
            ApplyVelocity(initialVelocity);
        }
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        var collidedRigidbody = collision.gameObject.GetComponent<Rigidbody2D>();

        if (handleCollidingGravity)
        {
            isColliding = true;
            SetAsAttractee(false);
        }

        if (handleCollidingVelocity)
        {
            rigidBody.velocity = Vector3.zero;
            rigidBody.angularVelocity = 0;
            collidedRigidbody.velocity =
                collision.gameObject.GetComponent<GravityTarget>().savedVelocity;
        }
    }

    void OnCollisionExit2D(Collision2D collision)
    {
        isColliding = false;

        if (handleCollidingGravity)
            SetAsAttractee(isAttractee);
    }

    void ApplyVelocity(Vector3 velocity)
    {
        rigidBody.AddForce(initialVelocity, ForceMode2D.Impulse);
    }
}
```

Лістинг 2. Клас GravityTarget, функції ApplyVelocity() та OnCollission()

3.1.2. Рух ракети

Для зміни швидкості та напрямку руху ракети скрипт “RocketTarget.cs” (див. лістинг 3), містить функцію “ChangeRocketVelocity()”. Для відображення напрямку руху створено функцію “ChangeRocketDirection()”, а для відображення величини прискорення – “ChangeFireParticle()”.

```
public class RocketTarget : MonoBehaviour
{
    void ChangeRocketVelocity()
    {
        if (!moveVector.Equals(Vector3.zero))
        {
            rigidBody.AddForce(updateVelocityValue * moveVector, ForceMode2D.Impulse);
        }
    }

    void ChangeFireParticle()
    {
        var main = particleSys.main;
        if (!moveVector.Equals(Vector3.zero))
        {
            main.startLifetime =
                (updateVelocityValue / maxUpdateVelocityValue) * 0.5f;

            float particleAngle =
                Mathf.Atan2(moveVector.y, moveVector.x) * Mathf.Rad2Deg;
            var quat1 = Quaternion.AngleAxis(particleAngle, Vector3.right);

            if (moveVector.y < 0)
                moveVector.y *= -1;

            particleAngle = Mathf.Atan2(moveVector.x, moveVector.y) * Mathf.Rad2Deg;
            var quat2 = Quaternion.AngleAxis(particleAngle, Vector3.down);

            particle.transform.rotation = quat1 * quat2;
        }
        else
        {
            main.startLifetime = 0f;
        }
    }

    void ChangeRocketDirection()
    {
        Vector2 moveDirection = rigidBody.velocity;
        if (moveDirection != Vector2.zero)
        {
            float angle = Mathf.Atan2(-moveDirection.x, moveDirection.y)
                * Mathf.Rad2Deg;
            rigidBody.MoveRotation(Quaternion.AngleAxis(angle, Vector3.forward));
        }
    }
}
```

Лістинг 3. Клас RocketTarget, функції ChangeRocketVelocity(), ChangeRocketDirection() та ChangeFireParticle()

3.1.3. Сцена та камери

Оскільки симуляція відтворює реальні пропорції, то якщо просто відобразити об’єкти на сцені з однією основною камерою, одні об’єкти будуть непомітні на фоні інших, а велика відстань між об’єктами не дозволить охопити всі тіла так, щоб ті були видимими при їх малих відносно цієї відстані розмірах. Тому для того, аби зробити об’єкти помітними для великих відстаней, створено об’єкти-мітки з розмірами, значно більшими за реальні. Ці об’єкти будуть видимі лише на основній камері, що буде сильно віддалена від сцени для охоплення всіх об’єктів на ній. А для відображення реальних розмірів об’єктів додамо другу камеру, яка крім того буде змінювати позицію разом з об’єктом, до якого прикріплена. Це дозволить стежити за тим об’єктом, який нас цікавить.

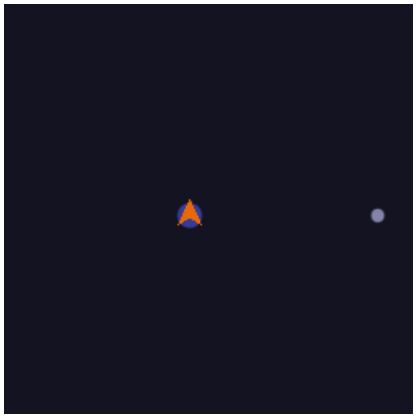
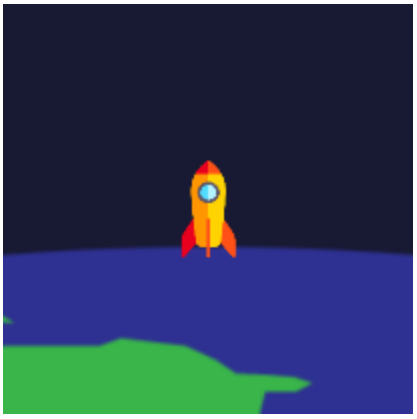
а)  б) 

Рис. 1. а) Зображення з основної камери: ракета (оранжева стріла) на поверхні Землі (синя точка) та Місяць (сіра точка); б) зображення з допоміжної камери: ракета на поверхні Землі

З метою відображення траєкторій руху об’єктів, кожному об’єкту додано компоненту “TrailParticle”, що залишає слід за шляхом, який пройшов об’єкт. Ці компоненти відображатимуться тільки на основній камері.

Допоміжна камера не обертатиметься разом з прикріпленим об’єктом аби можна було зрозуміти, куди він напрямлений.

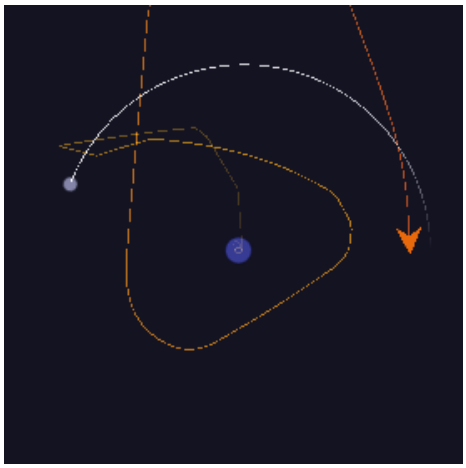
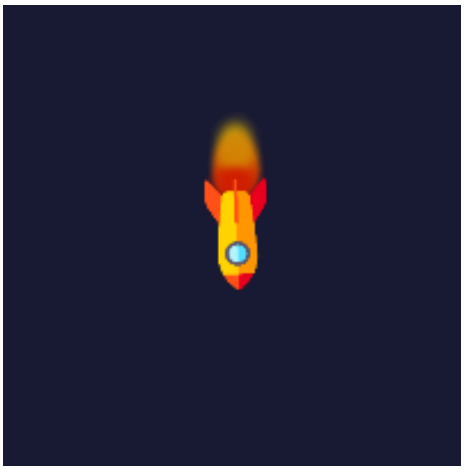
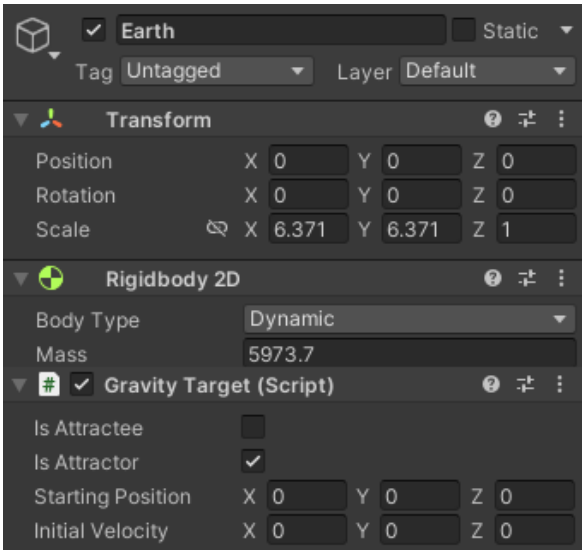
а)  б) 

Рис. 2. а) Зображення з основної камери: ракета залишає при русі оранжевий слід та Місяць описує еліптичну дугу сірим кольором; б) зображення з допоміжної камери: ракета летить у напрямку вниз

3.1.4. Параметри об’єктів та перевірка симуляції

Параметри об’єктам на сцені задаємо пропорційно реальним фізичним характеристикам із таблиці 1. Будемо вважати, що центром системи буде Земля, тому зробимо її нерухомою, для цього знімемо прапорець із властивості “IsAtractee” в “GravityTarget”. Основні властивості об’єктів, які відображають фізичні параметри Землі та Місяця на сцені представлено на рис. 3.

а) 

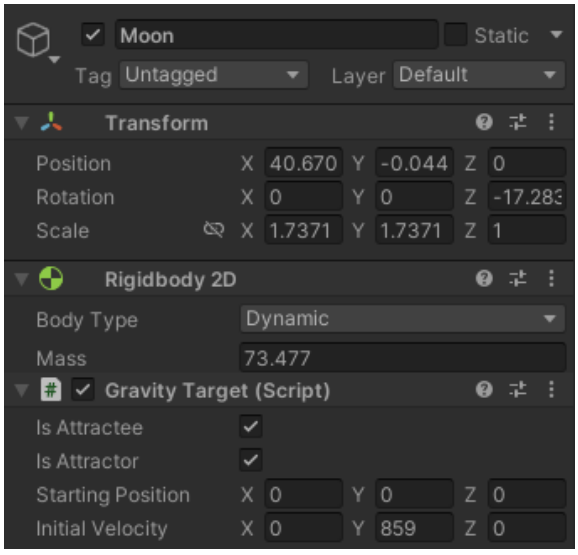
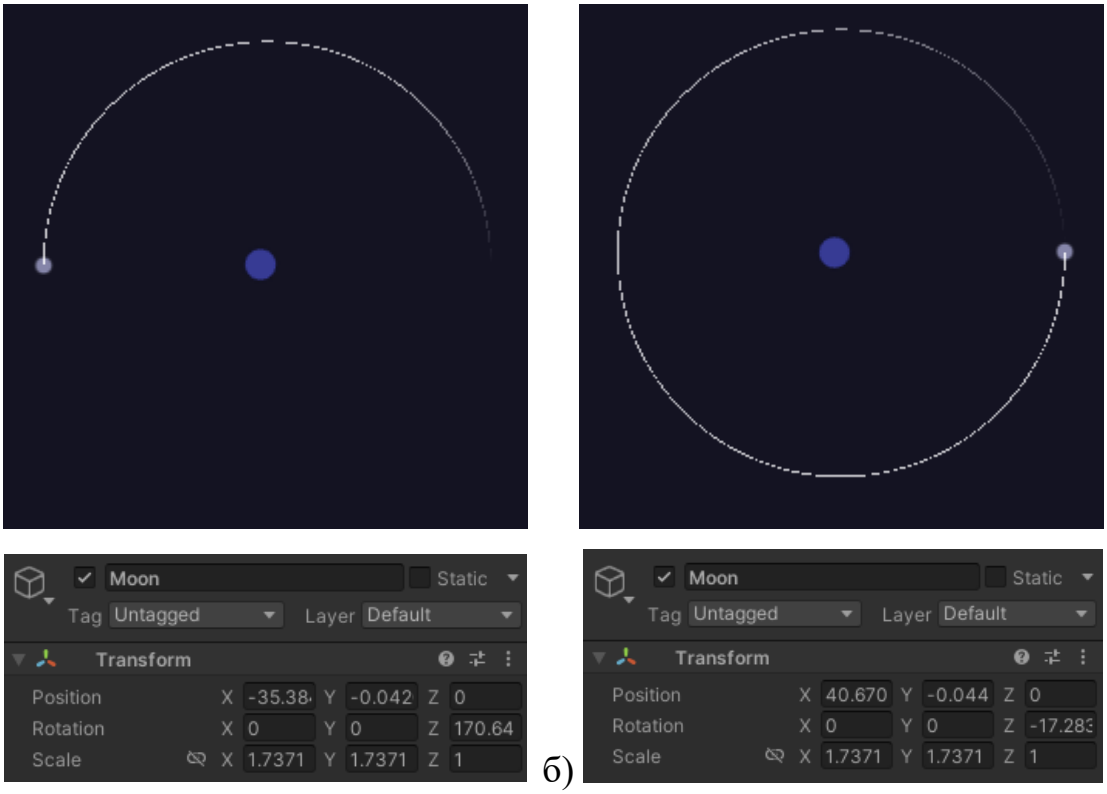
б) 

Рис. 3. а) Деякі властивості об’єкта “Earth”; б) деякі властивості об’єкта “Moon”

Для перевірки того, чи дійсно Місяць рухається навколо Землі правильно, звіримо параметри його орбіти та порівняємо їх із реальними (див. рис. 4).



а) б)

Рис. 4. а) Положення Місяця в перигеї; б) положення Місяця в апогеї

Можемо бачити, що координати Місяця в перигеї та апогеї приблизно збігаються із реальними значеннями, записаними в таблиці 1. Можемо сказати, що симуляція із задовільною точністю відтворює реальний рух Місяця навколо Сонця.

3.2. Реалізація нейронної мережі

Нейронну мережу реалізовано в групі класів, кожен з яких відображає структурний елемент в нейромережах. Це класи “Neuron”, “Dendrite”, “Layer”, “NeuralNetwork”. Нейрони з’єднуються з іншими нейронами для передачі сигнали через дендрити і об’єднуються в шари, які в свою чергу об’єднані в нейронну мережу. На клас “Neural Network” варто звернути особливу увагу. У класі реалізовано функцію “Run()” для проведення обчислень в мережі на основі вхідних даних. Як результат отримуємо значення на вихідних нейронах, що

використовуватимуться надалі в керуванні ракетою. Як функцію активації використано гіперболічний тангенс.

```
public class NeuralNetwork
{
    double Tanh(double x)
    {
        return System.Math.Tanh(x);
    }

    public double[] Run(List<double> input)
    {
        if (input.Count != this.layers[0].neurons.Count)
        {
            return null;
        }

        for (int l = 0; l < layers.Count; l++)
        {
            Layer currentLayer = layers[l];

            for (int n = 0; n < currentLayer.neurons.Count; n++)
            {
                Neuron neuron = currentLayer.neurons[n];
                if (l == 0)
                {
                    neuron.value = input[n];
                }
                else
                {
                    neuron.value = 0;
                    for (int lastNeuron = 0; lastNeuron
                        < this.layers[l - 1].neurons.Count; lastNeuron++)
                    {
                        neuron.value += this.layers[l - 1].neurons[lastNeuron].value
                            * neuron.dendrites[lastNeuron].weight;
                    }
                    neuron.value = Tanh(neuron.value + neuron.bias);
                }
            }
        }

        Layer lastLayer = this.layers[this.layers.Count - 1];
        int numOutput = lastLayer.neurons.Count;
        double[] output = new double[numOutput];
        for (int i = 0; i < numOutput; i++)
        {
            output[i] = lastLayer.neurons[i].value;
        }
        return output;
    }
}
```

Лістинг 4. Клас NeuralNetwork, функція Run()

Створена нейромережа використовується в класі “AIRocketController.cs”. При кожному оновленню кадру визначаються параметри об’єктів на сцені, що подаються на вхід нейромережі в нормалізованому вигляді. Дані на виході

подаються як параметри у функцію керування ракетою через клас “RocketTarget”.

```
public class AIRocketController : MonoBehaviour
{
    void FixedUpdate()
    {
        if (alive)
        {
            List<double> input = new List<double>();

            var normCoef = rocket.awayDistance;

            input.Add(ToNormValue(destination.rigidBody.position.x, normCoef));
            input.Add(ToNormValue(destination.rigidBody.position.y, normCoef));
            input.Add(ToNormValue(destination.rigidBody.velocity.x, 20));
            input.Add(ToNormValue(destination.rigidBody.velocity.y, 20));

            input.Add(ToNormValue(rocket.rigidBody.position.x, normCoef));
            input.Add(ToNormValue(rocket.rigidBody.position.y, normCoef));
            input.Add(ToNormValue(rocket.rigidBody.velocity.x, 1000));
            input.Add(ToNormValue(rocket.rigidBody.velocity.y, 1000));

            double[] output = network.Run(input);

            rocket.updateVelocityValue = ToUseValue((float)output[0],
                rocket.maxUpdateVelocityValue, 0, rocket.maxUpdateVelocityValue);
            rocket.moveVector = new Vector3((float)output[1], (float)output[2], 0);

            CalculateFitness();
        }
    }
}
```

Лістинг 5. Клас AIRocketController, керування ракетою за допомогою нейромережі

3.3. Реалізація генетичних алгоритмів

Хромосоми для ваг нейромереж будуть мати представлення у вигляді списку чисел із плаваючою комою. Для того щоб отримати хромосоми з нейромережі та навпаки, щоб використати хромосому для зміни значень ваг нейромережі у класі “NeuralNetwork” реалізовано методи “Encode()” та “Decode()”.

Функцію пристосованості визначено у класі “AIRocketController()”, яка викликається при кожному відмальовуванню нового фрейму. У цій функції значення придатності обраховується як обернено-пропорційне до відстані між

Місяцем та ракетою. Для генетичного алгоритму буде враховане лише значення придатності в момент часу, коли ракета була до Місяця найближче.

```
public class NeuralNetwork
{
    public List<double> Encode()
    {
        List<double> chromosome = new List<double>();
        for (int i = 1; i < layers.Count; i++)
        {
            for (int j = 0; j < layers[i].neurons.Count; j++)
            {
                chromosome.Add(layers[i].neurons[j].bias);
                for (int k = 0; k < layers[i].neurons[j].NumDendrites(); k++)
                    chromosome.Add(layers[i].neurons[j].dendrites[k].weight);
            }
        }
        return chromosome;
    }

    public void Decode(List<double> chromosome)
    {
        int geneIndex = 0;
        for (int i = 1; i < layers.Count; i++)
        {
            for (int j = 0; j < layers[i].neurons.Count; j++)
            {
                layers[i].neurons[j].bias = chromosome[geneIndex];
                geneIndex++;
                for (int k = 0; k < layers[i].neurons[j].NumDendrites(); k++)
                {
                    layers[i].neurons[j].dendrites[k].weight = chromosome[geneIndex];
                    geneIndex++;
                }
            }
        }
    }
}
```

Лістинг 6. Клас NeuralNetwork, функції Encode() та Decode()

```
public class AIRocketController : MonoBehaviour
{
    private void CalculateFitness()
    {
        float distance = Vector3.Distance(
            rocket.rigidBody.position, destination.rigidBody.position);

        float fitnessValue = 1 / distance;

        fitness = fitnessValue;
        if (fitness > bestFitness)
        {
            bestFitness = fitness;
        }
    }
}
```

Лістинг 7. Клас AIRocketController, функція CalculateFitness()

Генетичний процес здійснюється у класі “GeneticController”. Тут реалізовано функції для відбору, схрещування, мутації хромосом і як наслідок отримання нового покоління на основі індивідів поточного покоління.

Відбір здійснюється з пропорційною селекцією но основі значень функції пристосованості. Оператором кросинговеру є випадковий вибір гена від одного з батьків для кожної позиції хромосоми. Мутація відбувається за рахунок додавання випадкового рівномірно розподіленого значення у межах 0.1% від області значень мутуючого гена, але ці межі можуть бути зміненими через параметри класу “GeneticController”, так само як можуть бути змінені ймовірності мутації та схрещування.

```
public class GeneticController
{
    public void Crossover(List<double> mother, List<double> father)
    {
        List<double> tempM = new List<double>();
        List<double> tempF = new List<double>();

        for (int i = 0; i < mother.Count; i++)
        {
            if (crossoverRate > UnityEngine.Random.Range(0, 1f))
            {
                tempM.Add(father[i]);
                tempF.Add(mother[i]);
            }
            else
            {
                tempM.Add(mother[i]);
                tempF.Add(father[i]);
            }
        }

        mother.RemoveRange(0, mother.Count);
        father.RemoveRange(0, mother.Count);

        mother.InsertRange(0, tempF);
        father.InsertRange(0, tempM);
    }
}
```

Лістинг 8. Клас GeneticController, функція Crossover()

```
public class GeneticController
{
    public void Mutate(NeuralNetwork creature)
    {
        List<double> chromosome = creature.Encode();
        for (int i = 0; i < chromosome.Count; i++)
        {
            if (this.mutationRate > UnityEngine.Random.Range(0f, 1f))
            {
                chromosome[i] += mutationChange * UnityEngine.Random.Range(-1f, 1f);
                if (chromosome[i] > 1) chromosome[i] = 1;
                else if (chromosome[i] < -1) chromosome[i] = -1;
            }
        }

        creature.Decode(chromosome);
    }
}
```

Лістинг 9. Клас GeneticController, функція Mutate()

3.4. Реалізація навчання нейромережі

Навчання за допомогою реалізованого класу генетичних алгоритмів здійснюється в скрипті “Academy.cs”, що прикріплюється до об'єкту на сцені та керує всім процесом навчанні. Його роботу можна описати наступними кроками:

1. *Створення початкової популяції.* Створюється масив об'єктів “species” класу “GeneticController”, кількість яких рівна заданому розміру популяції “numGenomes”. Кожен об'єкт ініціалізує свою нейромережу з випадковими значенням ваг для подальшого тренування. Крім того, за параметром “numSimulate” можна задати, скільки ракет буде перебувати на симуляції одночасно. Саме на створених ракетах відбуватиметься робота нейромереж.
2. *Виконання симуляції.* Створені ракети, керовані нейромеражами, виконують політ. Політ припиняється, якщо виконується одна із умов припинення симуляції: виліт за межі симуляції, симуляція триває занадто довго або ракета влучила в об’єкт, який є пунктом призначення. Найкращі значення функції пристосованості записуються в об’єкти “species”. У симуляції передбачена можливість запускати один індивід декілька разів, робиться це через параметр “numExperiments”. Таким чином можна протестувати, як буде поводити себе ракета при різних

початкових положеннях Місяця відносно Землі. Значення пристосованості індивіда тоді визначатиметься як середнє пристосованостей для всіх експериментів.

3. *Отримання нового покоління.* Після того, як всі індивіди “species” були протестовані в симуляції, виконується генетичний процес і на основі отриманих значень пристосованості індивідів отримуємо нове покоління, яке візьме участь в наступній симуляції з подальшою еволюцією.

Для стеження за перебігом навчання на екрані виводяться показники про поточне та найкраще покоління, разом із найкращим та середнім показниками пристосованості та числом потраплянь ракети на Місяць. Симуляція із запущеним процесом навчання зображена на рис. 5.

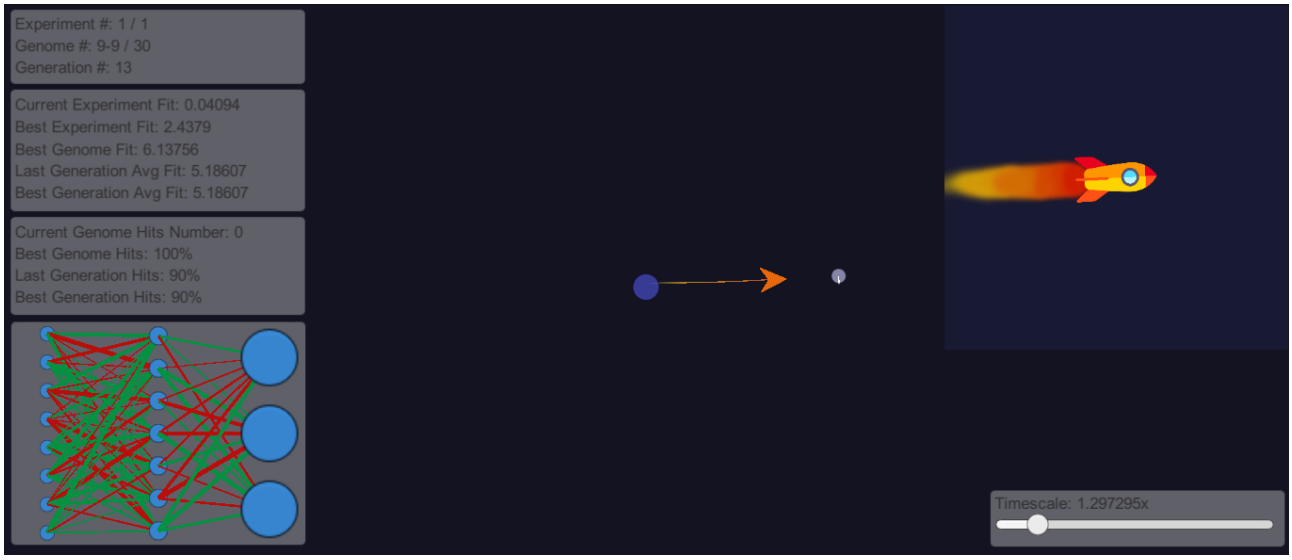


Рис. 5. Перебіг навчання керування польотом космічного корабля

4. ПРОВЕДЕННЯ ДОСЛІДІВ

4.1. Навчання польоту при однакових стартових положеннях

4.1.1. Опис задачі

Об’єкти в симуляції перебувають в постійному русі, тому в навчанні є принциповим положення об’єктів одне відносно одного в момент початку виконання симуляції польоту для кожного індивіда в популяції. Якщо це не врахувати, то може виявитись, що індивіди перебуватимуть в різних умовах, таким чином забираючи можливість правильно визначити хто із них є кращим. Саме тому вирішено при кожній новій симуляції польоту для індивіда повертати всі об’єкти в їх початкове положення. Це гарантуватиме, що індивіди отримуватимуть однакові дані на вхід їх нейромереж, і тому можна буде правильно зробити висновок про пристосованість.

4.1.2. Налаштування навчання

Для того щоб ракета навчилась здійснювати політ було вибрано нейронну мережу зі структурою, зображеною на рис. 6. Як було згадано в попередніх

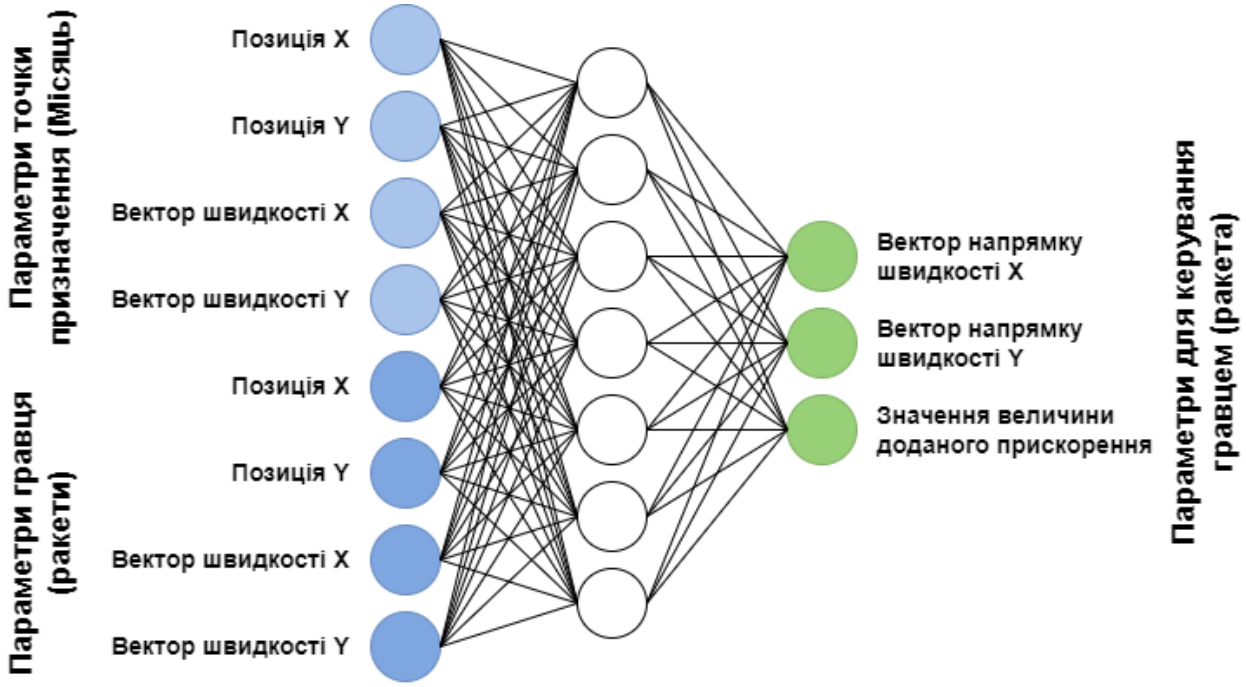


Рис. 6. Структура нейромережі із одним прихованим шаром із 7 нейронів

розділах, на вхід нейронна мережа приймає параметри про положення та швидкість об’єктів, а саме ракети та Місяця, а на виході видає параметри, необхідні для керування ракетою. Таким чином, отримуємо 8 нейронів на вхідному шарі та 3 нейрони на вихідному шарі. Крім того, наявний один прихований шар із 7 нейронами.

Параметри для проведення генетичного процесу висвітлено на рис. 7. Для кожного індивіда проводимо по одній симуляції. Кожного разу, як починає відбуватись симуляція для наступного індивіда, всі об'єкти повертаються в початкове положення. Для максимального значення мутації вибрано значення 0.1% від максимального значення гена, оскільки при більшому значенні спостерігалась недостатня точність нейромережі для потрапляння ракети на поверхню Місяця.

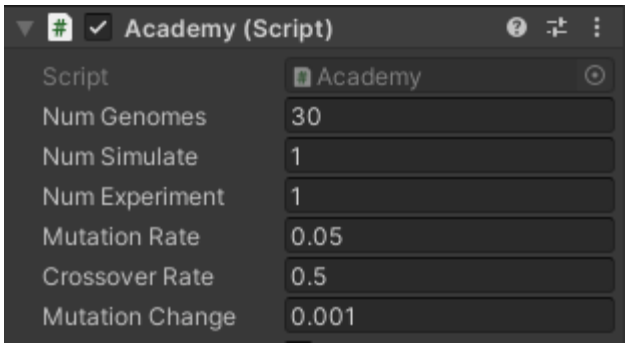


Рис. 7. Об’єкт “Academy”, основні параметри для проведення генетичного алгоритму в першому досліді

4.1.3. Результати навчання

Вибрана структура нейромережі добре справляється з розв’язком даної задачі, і вже на 15 поколінні можемо спостерігати що більшість космічних кораблів змогли долетіти до Місяця. Процес навчання можемо спостерігати на графіках на рис. 8. Вже після 5-го покоління відбувається стрімке зростання в значеннях функції пристосованості, а на 8-у поколінні отримуємо перше

потрапляння ракети на Місяць. Вже після 15-го покоління 100% індивідів справляється із керуванням польоту і має високі показники пристосованості.

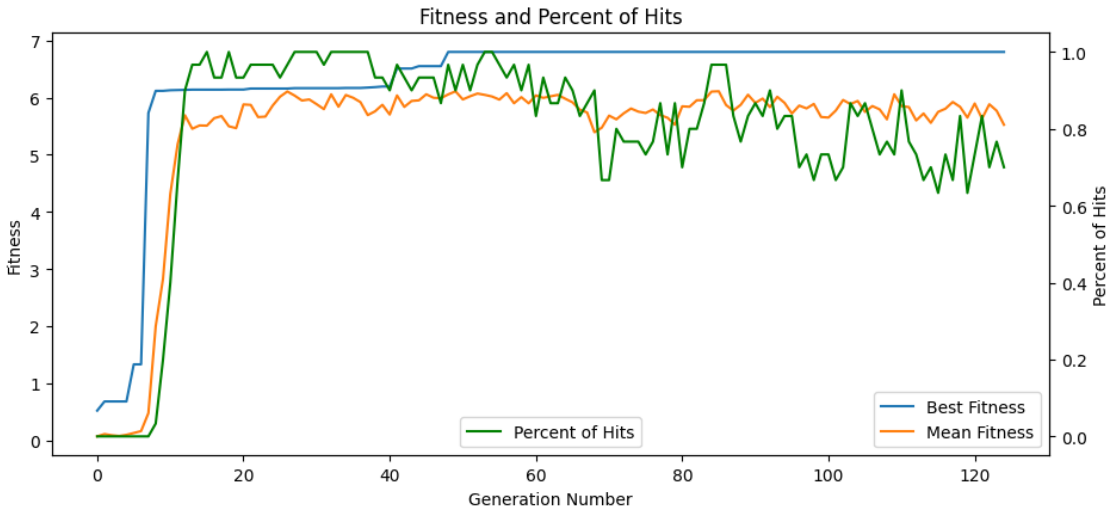


Рис. 8. Графік перебігу навчання при однакових стартових положеннях індивідів у популяції

Після 40-го покоління кількість успішних польотів дещо спадає, що може бути пов’язане з тим, що модель обчислює пристосованість тільки із відстані між Місяцем та ракетою, без врахування того, чи відбулось зіткнення між ними.

4.2. Навчання польоту із декількома симуляціями на особину

4.2.1. Опис задачі

Хоча в попередньому досліді і було досягнуто хороших результатів, проте присутній один недолік: через те що всі симуляції проводились при однакових положеннях Місяця, не вийде сказати, що нейромережа навчилась керувати кораблем саме до Місяця, а не просто в одному напрямку. Тому змінимо підхід і для кожного індивіда будемо проводити декілька польотів без затирання положення Місяця. Після того як особина завершить всі симуляції, здійснюється перехід до наступного індивіда і положення всіх об’єктів знову повернеться в початковий стан, для того щоб зберегти рівність умов для всіх індивідів. Значення пристосованості буде визначатись як її середнє для всіх експериментів особини.

4.2.2. Налаштування навчання

Оскільки задача в даному досліді є складнішою і необхідно визначити більше закономірностей в даних, то необхідно ускладнити структуру нейромережі. Вирішено взяти нейромережу із 2 прихованими шарами по 11 нейронів в кожному (див. рис. 9). Кількість вхідних та вихідних нейронів залишилась тією самою.

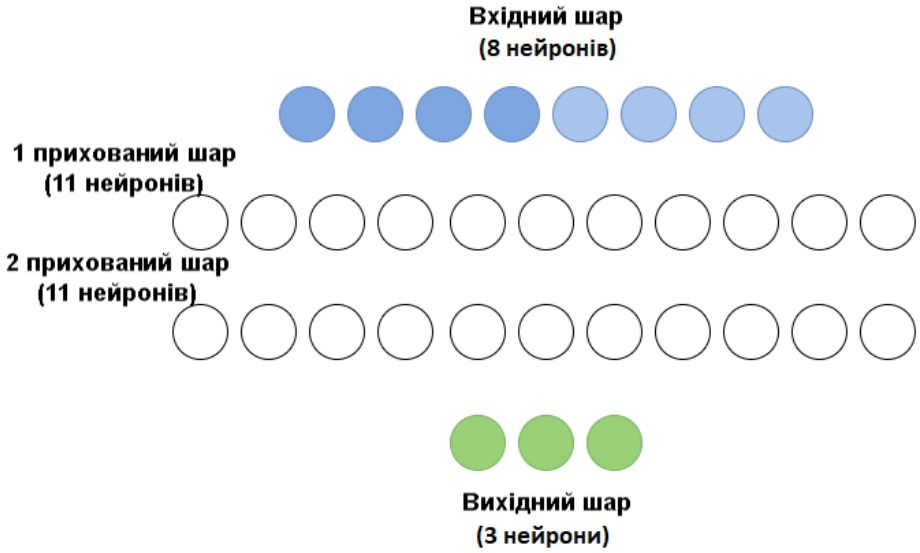


Рис. 9. Структура нейромережі із 2 прихованими шарами по 11 нейронів

Більшість параметрів для виконання генетичного алгоритму залишимо такими ж, як і в попередньому досліді, за винятком тих, які визначають кількість експериментів на кожну особину. Тому змінимо значення поля “numExperiment” до 10 (див. рис. 10).

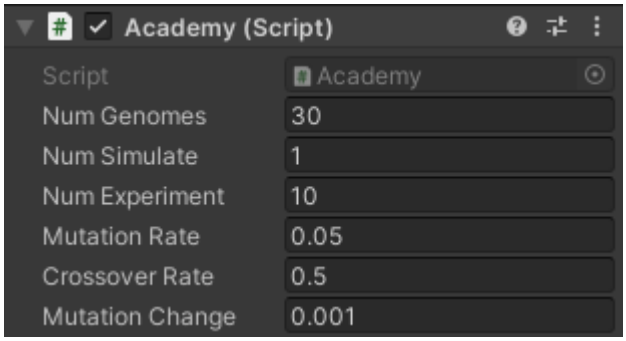


Рис. 10. Об’єкт “Academy”, основні параметри для проведення генетичного алгоритму в другому досліді

4.2.3. Результати навчання

Через складність нейронної мережі необхідною є більша кількість поколінь, проте навіть притаких умовах вдалось досягти лише задовільних результатів. Найкращі особини впорались із завданням доставки ракети на Місяць лише в 30% проведених симуляцій, тобто в 3 з 10 симуляцій, а найкращі показники пристосованості досягали приблизно 2,5. Для порівняння, в попередньому досліді це значення майже досягало 7. Звернувшись до графіку на рис. 11 бачимо, що стрімке зростання у значеннях пристосованості та числі потраплянь починає відбуватись від 15-го покоління, після 40-го покоління бачимо лише плавне покращення. На рис. 12 зображено показники для того самого досліду для більшої кількості поколінь. Спостерігаємо поступове погіршення значень пристосованості при продовженні навчання. Можна звернути увагу, що середній показник пристосованості на покоління завжди залишається значно меншим за його найкраще значення на покоління.

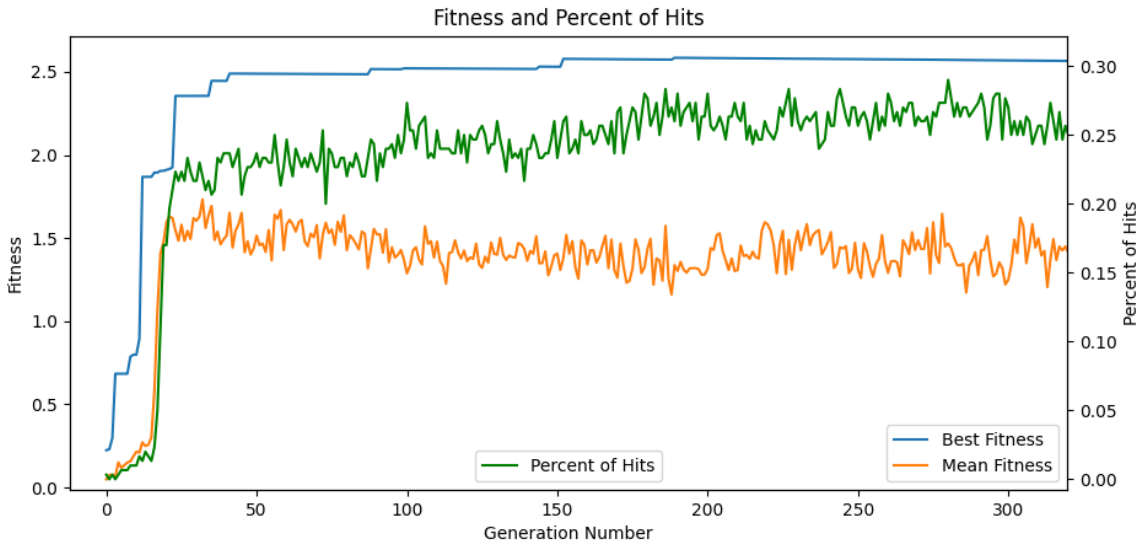


Рис. 11. Графік перебігу навчання при проведенні кількох симуляцій на особину

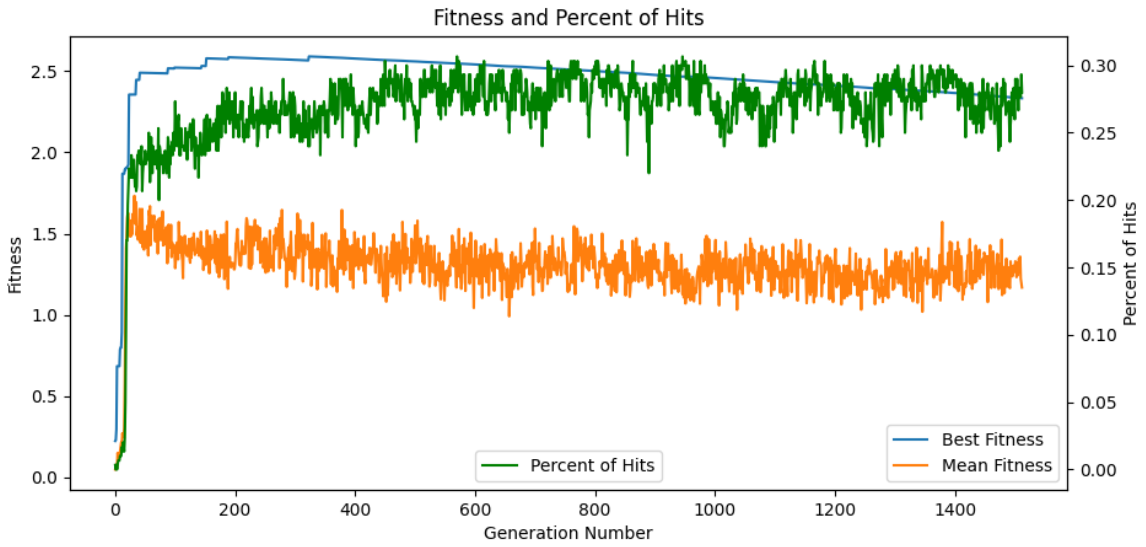


Рис. 12. Графік перебігу навчання при проведенні кількох симуляцій на особину зі спостеріганням спадання показників пристосованості

4.3. Навчання польоту зі змінними межами мутації

4.3.1. Опис задачі

Ускладнимо попередню задачу та будемо проводити 50 симуляцій на одну особину. Ключовим в цьому досліді буде зміна меж максимального значення мутації. В попердніх дослідах цей показник залишався сталим та був рівний 0.1%. В цьому ж досліді декілька разів буде здійснено зміна цього значення від більшого до меншого, тим самим забезпечуючи перехід від “exploration”, тобто знаходження найкращих рішень для розв’язання проблеми, до “exploitation” – покращення точності моделі. Варто зазначити, що програма не передбачає можливості для автоматичного підбору нових значень меж мутації, тому змінюватимемо їх самостійно, коли спостерігатимемо спад в прирості значення функції пристосованості.

4.3.2. Налаштування навчання

Збільшимо кількість нейронів в прихованих шарах: в кожному прихованому шарі знаходитиметься 15 нейронів. Розмір популяції збільшимо до 100. Значення параметра “mutationChange”, який визначає максимально

можливе значення мутації, на початку виконання виставляємо рівним 1%. Параметр “numExperiment” робимо рівним 50, це дозволить перевірити політ ракети при багатьох різних положеннях Місяця відносно Землі.

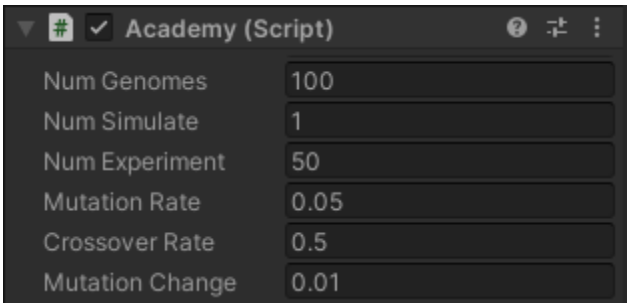


Рис. 13. Об’єкт “Academy”, основні параметри для проведення генетичного алгоритму в третьому досліді

4.3.3. Результати навчання

Можемо спостерігати схожі із попереднім дослідом результати навчання. Хоч модель тренується достатньо для того, аби навчитись визначати напрямок для польоту ракети, але бракує точності в керуванні щоб доставити її на Місяць.

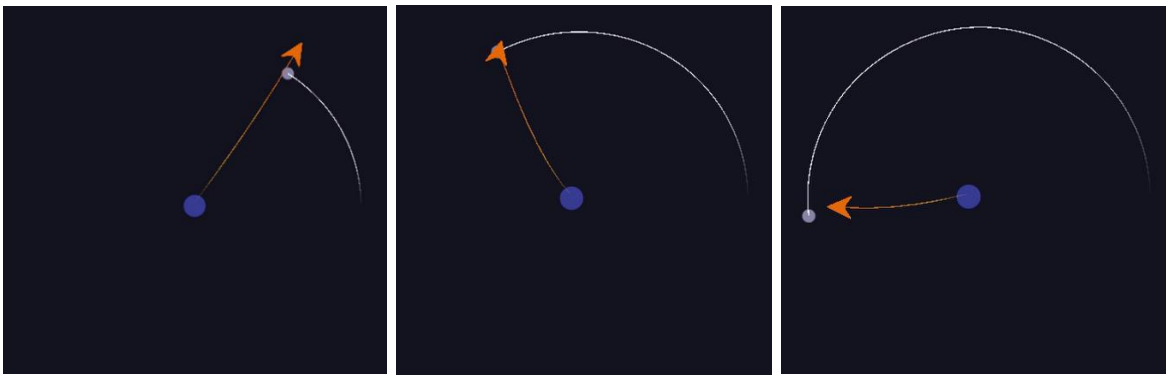


Рис. 14. Траєкторія руху ракети при різних положеннях Місяця для моделі, навченої до 260 поколінь в третьому досліді

На графіку на рис. 15 можемо бачити, як покращувались показники пристосованості при зменшенні максимально можливого значення мутації:

- для поколінь 1-115 максимальне значення мутації 1%;
- для поколінь 116-170 максимальне значення мутації 0,1%;
- для поколінь 171-260 максимальне значення мутації 0,01%.

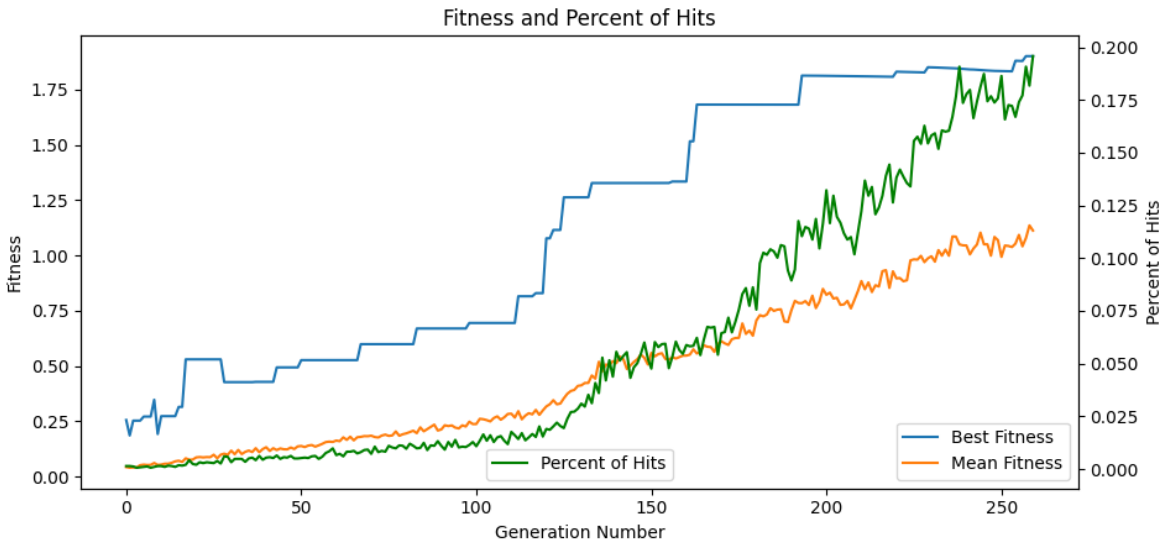


Рис. 15. Графік перебігу навчання при проведенні кількох симуляцій на особину та зменшенням максимального значення мутації під час навчання

Можна продовжити процес навчання і очікувати на покращення результатів, але він виявився досить часозатратним через великий розмір популяції, велике число симуляцій на особину та досить складну структуру нейромережі, тому після отримання результатів щодо впливу зміни меж мутації на перебіг навчання було вирішено припинити виконання алгоритму на даному етапі.

ВИСНОВКИ

В ході виконання курсової роботи було створено симуляцію спрощеної планетної системи та реалізовано навчання нейронної мережі за допомогою генетичних алгоритмів. Наприкінці було проведено ряд дослідів, що демонструють здатність вибраного підходу для розв'язання задачі у керуванні космічним кораблем від заданої початкової позиції до пункту призначення.

Значна частина роботи складалась із проектування симуляції руху тіл у планетній системі. У створеній симуляції можна виділити такі особливості:

- Рух тіл у просторі відбувається не за наперед визначеними траєкторіями, а за рахунок обрахунків, що відображають реальні закони фізики. Це не тільки додає реалістичності симуляції, а й дозволяє вводити в симуляцію нові тіла та змінювати їх параметри під час виконання симуляції.
- За рахунок дотримання реальних пропорцій відстаней, розмірів та мас Місяця та Землі вдалось отримати симуляцію, що досить точно відтворює їх реальний рух. В той же момент, через деякі обмеження рушія Unity, розміри ракети сильно віддаленні від реалістичних, тому є потреба або в зміні підходу щодо реалізації симуляції, або вибору іншого рушія.
- В симуляції присутня деяка обробка зіткнень об'єктів між собою, проте вона не досконала і більше покладається на готові рішення в Unity, ніж на реальні закони фізики, тому ця частина потребує вдосконалення. Хоч і даний недолік був виявлений, він не впливає на перебіг навчання нейромережі, оскільки відсутня потреба в аналізі поведінки космічного корабля при зіткненнях.

Реалізація генетичних алгоритмів значною мірою враховує представлення хромосом у вигляді чисел з плаваючою комою. Це виражається в особливостях використаних операторів схрещування та мутації. Хоч дана реалізація добре

справлялась з тренуванням нейромереж, бажаним є додавання інших типів операторів схрещування та мутації для варіативності проведення дослідів та знаходження оптимального рішення.

Щодо інтерфейсу в створеній симуляції бажаним є додавання більшої кількості параметрів для контролю за перебігом навчання нейромережі, а також виведення додаткової статистики, наприклад, з використанням графіків динаміки зміни значень пристосованості зі зміною поколінь. Проте останнє було компенсовано за рахунок запису даних у файл та подальшого використання зовнішніх програм.

В першому проведеному досліді вдалось отримати хороші результати тренування нейромережі і в найкращий момент перебігу навчання всі особини в популяції успішно потрапляли до пункту призначення. Тим не менш, цей дослід є досить недосконалим, адже не показує реальну здатність нейромережі керувати польотом саме до Місяця, а не просто в певному напрямку.

Тому в наступних дослідах було симульовано політ при різних початкових положеннях Місяця. В цих дослідах вдалось досягти тільки відносно задовільних результатів, адже не при всіх симуляціях особина справлялась з завданням.

Для покращення результатів навчання можна застосувати наступні рішення:

- Поекспериментувати з іншими структурами нейромережі.
- Здійснювати початковий етап навчання із більшим максимальним значенням мутації гена, так званий “exploration” підхід, для того аби можна було знайти найкраще рішення з наявними даними. Після цього поступово переходити до менших значень максимальної мутації, так званий “exploitation” підхід, для того щоб збільшити точність і досягти найкращих результатів.
- Винайти інший спосіб проведення дослідів, який дозволятиме тренувати нейромережу при різних умовах зі збереженням рівнозначності в отриманих показниках пристосованості для різних особин з популяції.

Варто також зазначити, що створена симуляція і проведені досліди мають потенціал для розширення. Додавання нових параметрів для навчання, таких як використання космічним кораблем пального та час здійснення польоту можна використати у функції пристосованості для того, щоб нейромережа навчилась їх економити, або ж додавання перешкод до планетної системи, або при польоті корабель навчився їх уникати. Це може надалі виразитись у складнішій поведінці, такій як здійснення гравітаційних маневрів, що використовуються при космічних польотах у реальності.

Досліджена тема дає змогу подивитись на генетичні алгоритми під іншим кутом та знайти рішення, які будуть релевантні саме для поставленої задачі, але реалізація потребує значної уваги до деталей та вивчення різних її аспектів.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. The Nature of Code / D. Shiffman, 2012. – 391-431 pp.
2. Genetic Algorithms in Search, Optimization, and Machine Learning / D. E. Goldberg – The University of Alabama: Adisson-Wesley Publishing Company, Inc., 1989. – 412 p.
3. A Field Guide to Genetic Programming / R. Poli, W. B. Langdon, N. F. McPhee – Lulu Enterprises, UK Ltd, 2008. – 242 p.
4. Unity in Action: Multiplatform Game Development in C# with Unity 5 / J. Hocking – Manning Publications; 1st edition, 2015. – 352 p.
5. Unit 3) Genetic Algorithms (Part 1) / Brandon Morgan – Towards Data Science, medium, 2021; Available from: https://towardsdatascience.com/unit-3-genetic-algorithms-part-1-986e3b4666d7
6. Using Neural Networks and Genetic Algorithms in C# .NET / Kory Becker – Primary Objects, 2009; Available from: https://www.primaryobjects.com/2009/05/06/using-neural-networks-and-genetic-algorithms-in-c-net/
7. What is the best game engine: is Unity right for you? / Marie Dealessandri – GamesIndustry.biz, 2016; Available from: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you
8. Coding in C# in Unity for beginners; Available from: https://unity.com/how-to/learning-c-sharp-unity-beginners

ДОДАТКИ

Додаток 1

Лістинг GravityHandler.cs

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

public class GravityHandler : MonoBehaviour
{
    [SerializeField] float g = 1;
    static float G;
    public static bool isSimulatingLive = true;
    public static List<GravityTarget> attractors = new List<GravityTarget>();
    public static List<GravityTarget> attractees = new List<GravityTarget>();

    void FixedUpdate()
    {
        G = g;
        if (isSimulatingLive)
            SimulateGravities();
    }

    public static void SimulateGravities()
    {
        foreach (GravityTarget attractor in attractors)
        {
            foreach (GravityTarget attractee in attractees)
            {
                if (attractor != attractee)
                {
                    var dist = Vector3.Distance(attractor.rigidBody.position,
attractee.rigidBody.position);
                    if (dist > attractor.GetComponent<Collider2D>().bounds.size.x &&
dist > attractor.GetComponent<Collider2D>().bounds.size.y
                        && dist > attractee.GetComponent<Collider2D>().bounds.size.x
&& dist > attractee.GetComponent<Collider2D>().bounds.size.y)
                    {
                        AddGravityForce(attractor.rigidBody, attractee.rigidBody);
                    }
                }
            }
        }
    }

    public static void AddGravityForce(Rigidbody2D attractor, Rigidbody2D target)
    {
        double massProduct = attractor.mass * target.mass;
        Vector3 difference = attractor.position - target.position;
        float distance = difference.magnitude;
        double unScaledforceMagnitude = massProduct / Math.Pow(distance, 2);
        double forceMagnitude = G * unScaledforceMagnitude;
        Vector3 forceDirection = difference.normalized;
        Vector3 forceVector = forceDirection * (float)forceMagnitude;
        target.AddForce(forceVector);
    }
}
```

Додаток 2 (частина 1)

Лістинг GravityTarget.cs

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

[RequireComponent(typeof(Rigidbody2D))]
public class GravityTarget : MonoBehaviour
{
    public Rigidbody2D rigidBody;
    public FixedJoint2D joint;
    private TrailRenderer trailRenderer;

    public bool isAttractee;
    public bool isAttractor;

    public Vector3 startingPosition;
    public Vector3 initialVelocity;
    public bool applyInitialVelocityOnStart;
    public bool handleCollidingVelocity;
    public bool handleCollidingGravity;
    public bool handleCollidingJoint;
    public bool isColliding;

    private Vector3 savedVelocity;
    private bool clearTrail = false;

    void Awake()
    {
        rigidBody = this.GetComponent<Rigidbody2D>();
        trailRenderer =
this.transform.Find("MapIcon").gameObject.GetComponent<TrailRenderer>();
    }

    void Start()
    {
        if (applyInitialVelocityOnStart)
        {
            ApplyVelocity(initialVelocity);
        }
    }

    void FixedUpdate()
    {
        savedVelocity = rigidBody.velocity;

        SetGravityState(!isColliding);
        if (joint != null)
        {
            joint.autoConfigureConnectedAnchor = false;
        }

        if (clearTrail)
        {
            clearTrail = false;
            trailRenderer.Clear();
            trailRenderer.enabled = true;
        }
    }
```

Додаток 2 (частина 2)

```
    void OnEnable()
    {
        SetGravityState(true);
    }

    void OnDisable()
    {
        SetGravityState(false);
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        var collidedRigidbody = collision.gameObject.GetComponent<Rigidbody2D>();

        if (handleCollidingGravity)
        {
            isColliding = true;
            SetAsAttractee(false);
        }
        if (handleCollidingVelocity)
        {
            rigidBody.velocity = Vector3.zero;
            rigidBody.angularVelocity = 0;

            collidedRigidbody.velocity =
collision.gameObject.GetComponent<GravityTarget>().savedVelocity;
        }
        if (handleCollidingJoint)
        {
            if (joint == null)
            {
                joint = gameObject.AddComponent<FixedJoint2D>();
                joint.connectedBody = collidedRigidbody;
                joint.breakForce = 0.1f;
                joint.enableCollision = true;
                joint.autoConfigureConnectedAnchor = true;
            }
        }
    }

    void OnCollisionExit2D(Collision2D collision)
    {
        isColliding = false;

        if (handleCollidingGravity)
        {
            SetAsAttractee(isAttractee);
        }
    }

    void ApplyVelocity(Vector3 velocity)
    {
        rigidBody.AddForce(initialVelocity, ForceMode2D.Impulse);
    }

    void SetAsAttractee(bool value)
    {
        if (value)
        {
            if (!GravityHandler.attractees.Contains(this))
            {
```

Додаток 2 (частина 3)

```
                GravityHandler.attractees.Add(this);
            }
        }
        else
        {
            GravityHandler.attractees.Remove(this);
        }
    }

    public void ResetPosition()
    {
        rigidBody.position = startingPosition;

        joint = null;
        isColliding = false;
        SetAsAttractee(isAttractee);

        rigidBody.velocity = Vector3.zero;
        ApplyVelocity(initialVelocity);

        trailRenderer.enabled = false;
        clearTrail = true;
    }

    void SetAsAttractor(bool value)
    {
        if (value)
        {
            if (!GravityHandler.attractors.Contains(this))
            {
                GravityHandler.attractors.Add(this);
            }
        }
        else
        {
            GravityHandler.attractors.Remove(this);
        }
    }

    void SetGravityState(bool isActive)
    {
        if (isActive)
        {
            SetAsAttractor(isAttractor);
            SetAsAttractee(isAttractee);
        }
        else
        {
            SetAsAttractee(false);
            SetAsAttractor(false);
        }
    }
}
```

Додаток 3 (частина 1)

Лістинг RocketTarget.cs

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

[RequireComponent(typeof(Rigidbody2D))]
public class RocketTarget : MonoBehaviour
{
    public Rigidbody2D rigidBody;
    private GravityTarget gravityTarget;
    private GameObject particle;
    private ParticleSystem particleSys;

    public float updateVelocityValue;
    public float maxUpdateVelocityValue;

    public bool handleVelocity;

    public Vector3 moveVector;

    public float idleTime;
    public float oldTime;
    public float stuckTime;
    public float collideTime;
    public float awayDistance;
    public int crushCount;

    public Vector3 centerPosition;
    public GameObject destination;

    public bool handleOld;
    public bool handleStuck;
    public bool handleCollide;
    public bool handleAway;
    public bool handleCrush;

    public bool old;
    public bool stuck;
    public bool collided;
    public bool away;
    public bool crushed;
    public bool hit;

    private float timeOldLeft;
    private float timeStuckLeft;
    private float timeCollideLeft;
    private int crushCountLeft;

    private Vector2 previousPosition;

    void Awake()
    {
        gravityTarget = this.GetComponent<GravityTarget>();

        rigidBody = this.GetComponent<Rigidbody2D>();
        particle = this.transform.Find("FlameParticles").gameObject;
        particleSys = particle.GetComponent<ParticleSystem>();
        previousPosition = rigidBody.position;
    }
```

Додаток 3 (частина 2)

```
    void FixedUpdate()
    {
        if (handleOld)
        {
            CheckForOld();
        }
        if (handleStuck)
        {
            CheckForStuck();
        }
        if (handleCollide)
        {
            CheckForCollided();
        }
        if (handleAway)
        {
            CheckForAway();
        }

        if (handleVelocity)
        {
            ChangeRocketVelocity();
            ChangeRocketDirection();
            ChangeFireParticle();
        }
    }

    void ChangeRocketVelocity()
    {
        if (!moveVector.Equals(Vector3.zero))
        {
            rigidBody.AddForce(updateVelocityValue * moveVector, ForceMode2D.Impulse);
        }
    }

    void CheckForOld()
    {
        if (!old)
        {
            timeOldLeft += Time.deltaTime;

            if (timeOldLeft > oldTime)
            {
                Debug.Log("Player is too Old");
                old = true;
            }
        }
    }

    void CheckForStuck()
    {
        if (!stuck && gravityTarget.joint != null)
        {
            timeStuckLeft += Time.deltaTime;
            if (timeStuckLeft > stuckTime)
            {
                Debug.Log("Player Stuck");
                stuck = true;
            }
        }
        else
```

Додаток 3 (частина 3)

```
        {
            timeStuckLeft -= Time.deltaTime;
            if (timeStuckLeft < 0)
            {
                timeStuckLeft = 0;
                stuck = false;
            }
        }
    }

    void CheckForCollided()
    {
        if (!collided && gravityTarget.isColliding)
        {
            timeCollideLeft += Time.deltaTime;
            if (timeCollideLeft > collideTime)
            {
                Debug.Log("Player Collided");
                collided = true;
            }
        }
        else
        {
            timeCollideLeft -= Time.deltaTime / 5;
            if (timeCollideLeft < 0)
            {
                timeCollideLeft = 0;
                collided = false;
            }
        }
    }

    void CheckForAway()
    {
        if (!away && Vector3.Distance(rigidBody.position, centerPosition) >
awayDistance)
        {
            Debug.Log("Player is far away");
            away = true;
        }
        else
        {
            away = false;
        }
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        if (handleCrush)
        {
            crushCountLeft += 1;
            if (crushCountLeft >= crushCount)
            {
                Debug.Log("Player Crushed");
                crushed = true;
            }
        }

        if (destination != null)
        {
            if (collision.gameObject == destination)
```

Додаток 3 (частина 4)

```
            {
                Debug.Log("Player Hit the destination");
                hit = true;
            }
        }
    }

    public void ResetRocket()
    {
        timeIdleLeft = 0;
        timeOldLeft = 0;
        timeStuckLeft = 0;
        crushCountLeft = 0;

        idle = false;
        old = false;
        stuck = false;
        collided = false;
        away = false;
        crushed = false;
        hit = false;

        transform.rotation = Quaternion.AngleAxis(0, Vector3.forward);
    }

    void ChangeFireParticle()
    {
        var main = particleSys.main;
        if (!moveVector.Equals(Vector3.zero))
        {
            main.startLifetime = (updateVelocityValue / maxUpdateVelocityValue) *
0.5f;
            float particleAngle;

            particleAngle = Mathf.Atan2(moveVector.y, moveVector.x) * Mathf.Rad2Deg;
            var quat1 = Quaternion.AngleAxis(particleAngle, Vector3.right);
            if (moveVector.y < 0)
            {
                moveVector.y *= -1;
            }
            particleAngle = Mathf.Atan2(moveVector.x, moveVector.y) * Mathf.Rad2Deg;
            var quat2 = Quaternion.AngleAxis(particleAngle, Vector3.down);
            particle.transform.rotation = quat1 * quat2;
        }
        else
        {
            main.startLifetime = 0f;
        }
    }

    void ChangeRocketDirection()
    {
        Vector2 moveDirection = rigidBody.velocity;
        if (moveDirection != Vector2.zero)
        {
            float angle = Mathf.Atan2(-moveDirection.x, moveDirection.y) *
Mathf.Rad2Deg;
            rigidBody.MoveRotation(Quaternion.AngleAxis(angle, Vector3.forward));
        }
    }
}
```

Додаток 4

Лістинг CameraTarget.cs

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

public class CameraTarget : MonoBehaviour
{
    public Transform target;
    public float height = -1f;
    void Update()
    {
        if (target != null)
            transform.position =
                new Vector3(target.position.x, target.position.y, height);
    }
}
```

Додаток 5

Лістинг Neuron.cs

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

[Serializable]
public class Neuron
{
    public List<Dendrite> dendrites;
    public double bias;
    public double delta;
    public double value;
    public int NumDendrites() { return dendrites.Count; }
    public Neuron(){
        this.bias = UnityEngine.Random.Range(-1f, 1f);
        this.dendrites = new List<Dendrite>();
    }
    public Neuron(int bias){
        this.bias = bias;
        this.dendrites = new List<Dendrite>();
    }
}
```

Додаток 6

Лістинг Dendrite.cs

```
using System;
using UnityEngine;

[Serializable]
public class Dendrite
{
    public double weight;

    public Dendrite() { this.weight = UnityEngine.Random.Range(-1f, 1f); }
}
```

Додаток 7

Лістинг Layer.cs

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

[Serializable]
public class Layer
{
    public List<Neuron> neurons;
    private int layerIndex;
    public object Neurons { get; internal set; }

    public int NumNeurons() {
        return neurons.Count;
    }

    public Layer(int numNeurons, int layerIndex){
        this.neurons = new List<Neuron>(numNeurons);
        this.layerIndex = layerIndex;

    }

    public double[,] GetWeights(NeuralNetwork network)
    {
        Layer nextLayer = network.layers[layerIndex + 1];
        double[,] weights = new double[NumNeurons(),nextLayer.NumNeurons()];
        for (int i = 0; i < NumNeurons(); i++)
        {
            for (int j = 0; j < nextLayer.neurons.Count; j++){
                weights[i, j] = nextLayer.neurons[j].dendrites[i].weight;
            }
        }
        return weights;
    }
}
```

Додаток 8 (частина 1)

Лістинг NeuralNetwork.cs

```
using UnityEngine;
using System.Collections.Generic;
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
public class NeuralNetwork
{
    public List<Layer> layers;
    public int[] layerStructure;
    public float fitness;
    public double fitnessRatio;

    public int NumLayers()
    {
        return layers.Count;
    }
```

Додаток 8 (частина 2)

```
    public NeuralNetwork(int[] layers)
    {
        if (layers.Length < 2)
        {
            return;
        }

        this.layers = new List<Layer>();
        this.layerStructure = layers;
        this.fitness = 0f;
        this.fitnessRatio = 0f;

        for (int i = 0; i < layers.Length; i++)
        {
            Layer currentLayer = new Layer(layers[i], i);
            this.layers.Add(currentLayer);

            for (int n = 0; n < layers[i]; n++)
            {
                currentLayer.neurons.Add(new Neuron());
            }

            foreach (Neuron neuron in currentLayer.neurons)
            {
                if (i == 0)
                {
                    neuron.bias = 0;
                }
                else
                {
                    for (int d = 0; d < layers[i - 1]; d++)
                    {
                        neuron.dendrites.Add(new Dendrite());
                    }
                }
            }
        }
    }

    public NeuralNetwork(String fileName)
    {
        string[] lines = File.ReadAllLines(fileName);
        string[] structure = lines[0].Split(new char[] { ',' });
        int[] numStrucutre = new int[structure.Length];

        for (int i = 0; i < structure.Length; i++)
        {
            numStrucutre[i] = System.Convert.ToInt32(structure[i]);
        }

        NeuralNetwork NN = new NeuralNetwork(numStrucutre);

        string[] element = lines[1].Split(new char[] { ',' });
        List<Double> encoded = new List<double>();

        for (int i = 0; i < element.Length; i++)
        {
            encoded.Add(Convert.ToDouble(element[i]));
            Debug.Log(encoded[i]);
        }
```

Додаток 8 (частина 3)

```
        NN.Decode(encoded);
        this.layers = NN.layers;
        this.layerStructure = NN.layerStructure;
        this.fitness = 0f;
        this.fitnessRatio = 0f;
    }

    public double Sigmoid(double x)
    {
        return 1 / (1 + Math.Exp(-x));
    }
    double Tanh(double x)
    {
        return System.Math.Tanh(x);
    }

    public List<double> Encode()
    {
        List<double> chromosome = new List<double>();

        for (int i = 1; i < layers.Count; i++)
        {
            for (int j = 0; j < layers[i].neurons.Count; j++)
            {
                chromosome.Add(layers[i].neurons[j].bias);
                for (int k = 0; k < layers[i].neurons[j].NumDendrites(); k++)
                {
                    chromosome.Add(layers[i].neurons[j].dendrites[k].weight);
                }
            }
        }
        return chromosome;
    }

    public void Decode(List<double> chromosome)
    {
        int geneIndex = 0;
        for (int i = 1; i < layers.Count; i++)
        {
            for (int j = 0; j < layers[i].neurons.Count; j++)
            {
                layers[i].neurons[j].bias = chromosome[geneIndex];
                geneIndex++;
                for (int k = 0; k < layers[i].neurons[j].NumDendrites(); k++)
                {
                    layers[i].neurons[j].dendrites[k].weight = chromosome[geneIndex];
                    geneIndex++;
                }
            }
        }
    }

    public double[] Run(List<double> input)
    {
        if (input.Count != this.layers[0].neurons.Count)
        {
            return null;
        }
        for (int l = 0; l < layers.Count; l++)
        {
            Layer currentLayer = layers[l];
```

Додаток 8 (частина 4)

```
            for (int n = 0; n < currentLayer.neurons.Count; n++)
            {
                Neuron neuron = currentLayer.neurons[n];

                if (l == 0)
                {
                    neuron.value = input[n];
                }
                else
                {
                    neuron.value = 0;
                    for (int lastNeuron = 0; lastNeuron < this.layers[l -
1].neurons.Count; lastNeuron++)
                    {
                        neuron.value += this.layers[l - 1].neurons[lastNeuron].value *
neuron.dendrites[lastNeuron].weight;
                    }
                        neuron.value = Tanh(neuron.value + neuron.bias);
                }
            }
        }

        Layer lastLayer = this.layers[this.layers.Count - 1];
        int numOutput = lastLayer.neurons.Count;
        double[] output = new double[numOutput];

        for (int i = 0; i < numOutput; i++)
        {
            output[i] = lastLayer.neurons[i].value;
        }

        return output;
    }

    public void Save(int network_number)
    {
        StreamWriter write = new StreamWriter("./Saves/NetworkSaves/nn" +
network_number + ".txt", true);

        for (int i = 0; i < layerStructure.Length - 1; i++)
        {
            write.Write(layerStructure[i] + ", ");
        }

        write.Write(layerStructure[layerStructure.Length - 1] + "\n");

        List<double> encoded = this.Encode();

        for (int i = 0; i < encoded.Count - 1; i++)
        {
            write.Write(encoded[i] + ", ");
        }

        write.Write(encoded[encoded.Count - 1]);
        write.Close();
    }
}
```

Додаток 9 (частина 1)

Лістинг GeneticController.cs

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

public class GeneticController
{
    public List<NeuralNetwork> population;
    public List<NeuralNetwork> nextGeneration;
    private double populationFitness;
    public float mutationRate;
    public float mutationChange;
    public float crossoverRate;
    public float averageFitness;
    int popSize;

    public GeneticController(int popSize, float mutationRate, float crossoverRate,
float mutationChange)
    {
        this.population = new List<NeuralNetwork>(popSize);
        this.populationFitness = 0f;
        this.mutationRate = mutationRate;
        this.crossoverRate = crossoverRate;
        this.mutationChange = mutationChange;
        this.averageFitness = 0f;
        this.popSize = popSize;

        for (int i = 0; i < popSize; i++)
        {
            this.population.Add(new NeuralNetwork(
                new int[] { 8, 7, 3 })); // 8, 11, 11, 3
        }
    }

    public void Crossover(List<double> mother, List<double> father)
    {
        List<double> tempM = new List<double>();
        List<double> tempF = new List<double>();
        for (int i = 0; i < mother.Count; i++)
        {
            if (crossoverRate > UnityEngine.Random.Range(0, 1f))
            {
                tempM.Add(father[i]);
                tempF.Add(mother[i]);
            }
            else
            {
                tempM.Add(mother[i]);
                tempF.Add(father[i]);
            }
        }

        mother.RemoveRange(0, mother.Count);
        father.RemoveRange(0, mother.Count);

        mother.InsertRange(0, tempF);
        father.InsertRange(0, tempM);
    }

    public NeuralNetwork[] Breed(NeuralNetwork mother, NeuralNetwork father)
    {
```

Додаток 9 (частина 2)

```
        NeuralNetwork child1 = new NeuralNetwork(mother.layerStructure);
        NeuralNetwork child2 = new NeuralNetwork(mother.layerStructure);

        List<double> motherChromosome = mother.Encode();
        List<double> fatherChromosome = father.Encode();

        Crossover(motherChromosome, fatherChromosome);

        child1.Decode(motherChromosome);
        child2.Decode(fatherChromosome);

        return new NeuralNetwork[] { child1, child2 };
    }

    public void Mutate(NeuralNetwork creature)
    {
        List<double> chromosome = creature.Encode();

        for (int i = 0; i < chromosome.Count; i++)
        {
            if (this.mutationRate > UnityEngine.Random.Range(0f, 1f))
            {
                chromosome[i] += mutationChange * UnityEngine.Random.Range(-1f, 1f);
                if (chromosome[i] > 1) chromosome[i] = 1;
                else if (chromosome[i] < -1) chromosome[i] = -1;
            }
        }

        creature.Decode(chromosome);
    }

    public void NextGeneration()
    {
        this.nextGeneration = new List<NeuralNetwork>();
        this.populationFitness = 0f;

        for (int i = 0; i < this.population.Count; i++)
        {
            this.populationFitness += population[i].fitness;
        }

        for (int i = 0; i < this.population.Count; i++)
        {
            population[i].fitnessRatio = population[i].fitness /
this.populationFitness;
        }

        averageFitness = (float)(this.populationFitness / this.population.Count);

        population.Sort((x, y) => y.fitnessRatio.CompareTo(x.fitnessRatio));

        nextGeneration.Add(population[0]);

        for (int i = 0; i < this.population.Count / 2; i++)
        {
            int parent1Index = -1;
            int parent2Index = -1;
            double chance = UnityEngine.Random.Range(0f, 100f) / 100;
            double chance2 = UnityEngine.Random.Range(0f, 100f) / 100;
            double range = 0;
```

Додаток 9 (частина 3)

```
            for (int j = 0; j < this.population.Count; j++)
            {

                range += population[j].fitnessRatio;

                if (chance > range && chance2 > range)
                {
                    continue;
                }

                if (chance <= range && parent1Index < 0)
                {
                    parent1Index = j;
                }

                if (chance2 <= range && parent2Index < 0)
                {
                    if (parent1Index == j)
                    {
                        parent2Index = (j + 1) % population.Count;
                    }
                    else
                    {
                        parent2Index = j;
                    }
                }
                if (parent1Index >= 0 && parent2Index >= 0)
                {
                    break;

                }
            }

            if (parent1Index < 0)
            {
                parent1Index = this.population.Count - 1;
            }
            if (parent2Index < 0)
            {
                parent2Index = this.population.Count - 1;
            }

            NeuralNetwork[] children = Breed(population[parent1Index],
population[parent2Index]);

            Mutate(children[0]);
            Mutate(children[1]);

            nextGeneration.Add(children[0]);
            nextGeneration.Add(children[1]);
        }

        for (int i = 1; i < popSize; i++)
        {
            population[i] = nextGeneration[i];
        }
    }
}
```

Додаток 10 (частина 1)

Лістинг AIRocketController.cs

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

public class AIRocketController : MonoBehaviour
{
    public NeuralNetwork network = null;

    public RocketTarget rocket;

    public GravityTarget rocketGravity;
    public List<GravityTarget> obstacles;
    public GravityTarget destination;
    private int numExperiment;
    private bool resetPosition;

    public float fitness;
    public float bestFitness;

    public bool alive = true;
    public bool wrong = false;

    public int hits = 0;

    void Start()
    {
        obstacles.Add(GameObject.Find("Earth").GetComponent<GravityTarget>());
        destination = GameObject.Find("Moon").GetComponent<GravityTarget>();
        rocket.destination = GameObject.Find("Moon");

        var academy = GameObject.Find("Academy").GetComponent<Academy>();
        numExperiment = academy.numExperiment;
        resetPosition = academy.resetPosition;
    }

    void FixedUpdate()
    {
        if (alive)
        {
            List<double> input = new List<double>();

            var normCoef = rocket.awayDistance;

            input.Add(ToNormValue(destination.rigidBody.position.x, normCoef));
            input.Add(ToNormValue(destination.rigidBody.position.y, normCoef));

            input.Add(ToNormValue(destination.rigidBody.velocity.x, 20));
            input.Add(ToNormValue(destination.rigidBody.velocity.y, 20));

            input.Add(ToNormValue(rocket.rigidBody.position.x, normCoef));
            input.Add(ToNormValue(rocket.rigidBody.position.y, normCoef));

            input.Add(ToNormValue(rocket.rigidBody.velocity.x, 1000));
            input.Add(ToNormValue(rocket.rigidBody.velocity.y, 1000));

            double[] output = network.Run(input);

            rocket.updateVelocityValue = ToUseValue((float)output[0],
rocket.maxUpdateVelocityValue, 0, rocket.maxUpdateVelocityValue);
```

Додаток 10 (частина 2)

```
            rocket.moveVector = new Vector3((float)output[1], (float)output[2], 0);

            CalculateFitness();

            if (rocket.hit)
            {
                hits += 1;
            }
            if (rocket.stuck || rocket.collided || rocket.away || rocket.idle ||
rocket.old || rocket.crushed || rocket.hit)
            {
                if (!rocket.hit && !rocket.away)
                {
                    wrong = true;
                    bestFitness = 0;
                }
                Stop();
            }
        }
    }

    private void CalculateFitness()
    {
        float distance = Vector3.Distance(rocket.rigidBody.position,
destination.rigidBody.position);

        float fitnessValue = 1 / distance;

        fitness = fitnessValue;
        if (fitness > bestFitness)
        {
            bestFitness = fitness;
        }
    }

    private float ToNormValue(
        float value, float coef, float? min = null, float? max = null)
    {
        var normval = value / coef;

        min ??= -coef;
        max ??= coef;
        if (normval < min)
            normval = (float)min;
        if (normval > max)
            normval = (float)max;

        return normval;
    }

    private float ToUseValue(float value, float coef, float min, float max)
    {
        var useval = min + ((value + 1) / 2) * (max - min);
        return useval;
    }

    private void Stop()
    {
        alive = false;
        wrong = false;
```

Додаток 10 (частина 3)

```
        rocket.handleVelocity = false;
        rocket.updateVelocityValue = 0;
        rocket.rigidBody.velocity = Vector3.zero;
        rocket.rigidBody.simulated = false;

        network.fitness += bestFitness / numExperiment;
    }

    public void Reset()
    {
        rocket.rigidBody.simulated = true;

        alive = true;
        wrong = false;

        rocket.handleVelocity = true;
        rocket.updateVelocityValue = 0;

        rocketGravity.ResetPosition();
        rocket.ResetRocket();

        network.fitness = 0;

        fitness = 0;
        bestFitness = 0;

        hits = 0;

        if (resetPosition)
        {
            foreach (var obstacle in obstacles)
            {
                obstacle.ResetPosition();
            }

            destination.ResetPosition();
        }
    }

    public void ResetExp()
    {
        rocket.rigidBody.simulated = true;

        alive = true;
        wrong = false;

        rocket.handleVelocity = true;
        rocket.updateVelocityValue = 0;

        fitness = 0;
        bestFitness = 0;

        rocket.ResetRocket();
        rocketGravity.ResetPosition();
    }
}
```

Додаток 11 (частина 1)

Лістинг Academy.cs

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

public class Academy : MonoBehaviour
{
    public int numGenomes;
    public int numSimulate;
    public int numExperiment;

    public float mutationRate = 0.05F;
    public float crossoverRate = 0.5F;
    public float mutationChange = 0.001F;

    public bool resetPosition;

    public GameObject rocketFab;

    GameObject[] rockets;
    AIRocketController[] rocketControllers;

    public GeneticController species;

    public CameraTarget cameraTarget;
    public AIRocketController bestRocket;

    public int currentGenome;
    public int currentExperiment;
    public int batchSimulate;
    public int currentGeneration;

    public float currentExperimentFitness;
    public float bestExperimentFitness;
    public float bestGenomeFitness;
    public float lastGenerationAverageFitness;
    public float bestGenerationAverageFitness;

    public float currentGenomeHitsNumber;
    public float bestGenomeHitsPercent;
    private float currentGenerationHitsNumber;
    public float lastGenerationHitsPercent;
    public float bestGenerationHitsPercent;

    public GameObject Network_GUI;
    private UI_Network networkUI;

    private List<GravityTarget> obstacles = new List<GravityTarget>();
    private GravityTarget destination;

    void Start()
    {
        EmptySaves();

        obstacles.Add(GameObject.Find("Earth").GetComponent<GravityTarget>());
        destination = GameObject.Find("Moon").GetComponent<GravityTarget>();
```

Додаток 11 (частина 2)

```
        species = new GeneticController(numGenomes, mutationRate, crossoverRate,
mutationChange);
        rockets = new GameObject[numSimulate];
        rocketControllers = new AIRocketController[numSimulate];

        for (int i = 0; i < numSimulate; i++)
        {
            rockets[i] = Instantiate(rocketFab,
rocketFab.GetComponent<GravityTarget>().startingPosition,
rocketFab.transform.rotation);
            rocketControllers[i] = rockets[i].GetComponent<AIRocketController>();
            rocketControllers[i].network = species.population[i];
        }

        currentGenome = numSimulate;
        batchSimulate = numSimulate;

        Network_GUI = Instantiate(Network_GUI);
        UI_Genetics genetics = Network_GUI.GetComponentInChildren<UI_Genetics>();
        genetics.academy = this;
        networkUI = Network_GUI.GetComponentInChildren<UI_Network>();
        networkUI.Display(rocketControllers[0].network);
    }

    void FixedUpdate()
    {
        bool allRocketsDead = true;
        float bestRocketFitness = 0;

        foreach (AIRocketController rocket in rocketControllers)
        {
            if (rocket.alive)
            {
                allRocketsDead = false;

                if (rocket.fitness > bestRocketFitness)
                {
                    bestRocketFitness = rocket.fitness;
                    bestRocket = rocket;

                    currentExperimentFitness = bestRocketFitness;
                    currentGenomeHitsNumber = bestRocket.hits;

                    cameraTarget.target = rocket.transform;
                    networkUI.DrawConnections(rocket.network);

                    if (rocket.fitness > bestExperimentFitness)
                        bestExperimentFitness = rocket.fitness;
                }
            }
        }

        if (allRocketsDead)
        {
            Debug.Log("All Rockets Dead");

            if (currentExperiment == numExperiment)
            {
                foreach (AIRocketController rocket in rocketControllers)
                {
                    if (rocket.network.fitness > bestGenomeFitness)
```

Додаток 11 (частина 3)

```
                    {
                        bestGenomeFitness = rocket.network.fitness;
                    }

                    float hitsPercent = (float)rocket.hits / numExperiment;
                    if (hitsPercent > bestGenomeHitsPercent)
                    {
                        bestGenomeHitsPercent = hitsPercent;
                    }
                }

                if (currentGenome == numGenomes)
                {
                    foreach (var obstacle in obstacles)
                    {
                        obstacle.ResetPosition();
                    }
                    destination.ResetPosition();

                    SaveBestNetwork();
                    SaveStats();
                    species.NextGeneration();

                    lastGenerationAverageFitness = species.averageFitness;
                    if (lastGenerationAverageFitness > bestGenerationAverageFitness)
                    {
                        bestGenerationAverageFitness = lastGenerationAverageFitness;
                    }

                    for (int i = 0; i < numSimulate; i++)
                    {
                        currentGenerationHitsNumber += rocketControllers[i].hits;
                        rocketControllers[i].network = species.population[i];
                        rocketControllers[i].Reset();
                    }

                    lastGenerationHitsPercent = currentGenerationHitsNumber /
(numGenomes * numExperiment);
                    if (lastGenerationHitsPercent > bestGenerationHitsPercent)
                    {
                        bestGenerationHitsPercent = lastGenerationHitsPercent;
                    }
                    currentGenerationHitsNumber = 0;
                    currentGeneration++;
                    currentGenome = numSimulate;
                }
                else
                {
                    if (currentGenome + numSimulate <= numGenomes)
                    {
                        Debug.Log("Full Sim");
                        batchSimulate = numSimulate;
                    }
                    else
                    {
                        Debug.Log("Partial Sim");
                        batchSimulate = numGenomes - currentGenome;
                    }

                    for (int i = 0; i < batchSimulate; i++)
                    {
```

Додаток 11 (частина 4)

```
                        currentGenerationHitsNumber += rocketControllers[i].hits;
                        rocketControllers[i].network =
                             species.population[currentGenome + i];
                        rocketControllers[i].Reset();
                    }

                    currentGenome += batchSimulate;
                }

                currentExperiment = 1;
            }
            else
            {
                for (int i = 0; i < batchSimulate; i++)
                {
                    rocketControllers[i].ResetExp();
                }

                currentExperiment += 1;
            }
        }
    }

    private void EmptySaves()
    {
        if (Directory.Exists("./Saves"))
        {
            Directory.Delete("./Saves", true);
        }
        Directory.CreateDirectory("./Saves");
        Directory.CreateDirectory("./Saves/FitnessSaves");
        Directory.CreateDirectory("./Saves/HitsSaves");
        Directory.CreateDirectory("./Saves/NetworkSaves");
    }

    private void SaveBestNetwork()
    {
        species.population.OrderByDescending(n =>
            n.fitness).FirstOrDefault().Save(currentGeneration);
    }

    private void SaveStats()
    {
        StreamWriter fitWriter = new StreamWriter("./Saves/FitnessSaves/fits.csv",
true);
        StreamWriter hitWriter = new StreamWriter("./Saves/HitsSaves/hits.csv", true);

        foreach (var individual in species.population)
        {
            fitWriter.Write(individual.fitness + ", ");
        }

        fitWriter.Write("\n");
        hitWriter.Write(lastGenerationHitsPercent + "\n");

        fitWriter.Close();
        hitWriter.Close();
    }
}
```

Додаток 12

Лістинг rocket_genetics_graph_builder.py

```
import tkinter as tk
from tkinter import filedialog
import pandas as pd
import matplotlib.pyplot as plt

print("\nChoose first csv-file with fitnesses values ")
root = tk.Tk()
root.withdraw()
file_path1 = filedialog.askopenfilename(filetypes=[("CSV Files", "*.csv")])
root.destroy()
print("csv-file with factors: ", file_path1)
fitnesses_file = pd.read_csv(file_path1)

print("\nChoose second csv-file with hits values ")
root = tk.Tk()
root.withdraw()
file_path2 = filedialog.askopenfilename(filetypes=[("CSV Files", "*.csv")])
root.destroy()
print("csv-file with experiments results: ", file_path1)
hits_file = pd.read_csv(file_path2)

fitnesses_data = fitnesses_file.values
fitnesses_data = fitnesses_data[:, :-1].astype(float)
best_fitness = fitnesses_data.max(axis=1)
mean_fitness = fitnesses_data.mean(axis=1)

hits_data = hits_file.values.flatten()

fig, ax1 = plt.subplots()

ax1.plot(range(0, len(best_fitness)), best_fitness, label='Best Fitness')
ax1.plot(range(0, len(mean_fitness)), mean_fitness, label='Mean Fitness')
ax1.set_xlabel('Generation Number')
ax1.set_ylabel('Fitness')
ax1.legend(loc='lower right')

ax2 = ax1.twinx()
ax2.plot(range(0, len(hits_data)), hits_data, 'g', label='Percent of Hits')
ax2.set_ylabel('Percent of Hits')
ax2.legend(loc='lower center')

plt.title('Fitness and Percent of Hits')
plt.show()
```