Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

(позначення документа)

Дослідження використання штучних нейронних мереж в іграх

(тема)

Виконав: студент 2 курсу, групи СШІм-18-3
спеціальності 122 – Комп'ютерні науки
(код і повна назва спеціальності)
освітньо-наукова програма Системи штучного інтелекту (СШІ)
(повна назва освітньої програми)
Харченко М. О.
(прізвище, ініціали)
Керівник
(посада, прізвище, ініціали)

Допускається до захисту

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

2020 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 – Комп'ютерні науки
(код і повна назва)

Тип програми _____
(освітньо-проіфесійна або освітньо-наукова)

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

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

Студентові Харченка Михайла Олександровича
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження використання штучних нейронних мереж в іграх
_____
_____

затверджена наказом університету від ___ _____ 20 __ р. №_____

2. Термін подання студентом роботи до екзаменаційної комісії ____ _____ 20__ р.

3. Вихідні дані до роботи _____
_____
_____
_____
_____
_____
_____
_____
_____

4. Перелік питань, що потрібно опрацювати в роботі _____
_____
_____
_____
_____
_____
_____
_____

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

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

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка

Дата видачі завдання _____ _____ 20 __ р.

Студент _____
(підпис)

Керівник роботи _____ _____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до атестаційної роботи магістра містить: 85 стор., 21 рис., 1 додаток, 43 джерел.

Об'єктом дослідження стали стратегічні ігри та застосування генетичних алгоритмів для навчання нейронних мереж в іграх.

Метою роботи було створення прототипу битви з стратегій та навчання нейронної мережі вигравати битву.

Методи розробки базуються на інструментах розробки ігор на платформі .NET, движку Unity 3D, використовуючи середовище розробки Microsoft Visual Studio 2019.

В результаті роботи здійснена програмна реалізація прототипу битви, та нейрона мережа з навченими вагами за допомогою Unity 3D.

ДОДАТОК, UNITY 3D, ГРА, РОЗРОБКА, EVOLUTIONARY ALGORITHM, GENETIC ALGORITHM, EVOLUTION STRATEGY, EVOLUTION PROGRAMMING, .NET.

ABSTRACT

Explanatory note to master’s qualification work: 85 pages, 21 figures, 43 sources, 1 applications.

The object of the study was strategic games and the use of genetic algorithms for learning neural networks in games.

The aim of the work was to create a prototype battle strategy and teach the neural network to win the battle.

Development methods are based on game development tools on the .NET platform, Unity 3D engine, using the development environment Microsoft Visual Studio 2019.

As a result, the software implementation of the battle prototype and the neural network with trained scales using Unity 3D.

APPLICATION, UNITY 3D, GAME, DEVELOPMENT, .NET, GENETIC ALGORITHM, EVOLUTION STRATEGY, EVOLUTION PROGRAMMING.

ЗМІСТ

ВСТУП

У традиційних дослідженнях в галузі ШІ метою є створення справжнього інтелекту, хоча і штучними засобами. З точки зору ігор справжній ШІ далеко виходить за рамки вимог розважального програмного проекту. В іграх така міць не потрібна. Ігровий ШІ не повинен бути наділений почуттями і самосвідомістю (чесно кажучи, і дуже добре, що це саме так!). Йому немає необхідності навчатися чогось за межами рамок ігрового процесу. Справжня мета ШІ в іграх полягає в імітації розумної поведінки і в наданні гравцеві переконливо, правдоподібно завдання.

Основним принципом, що лежить в основі роботи ШІ, є прийняття рішень. При прийнятті рішень система повинна впливати на об'єкти за допомогою ШІ. При цьому такий вплив може бути організовано у вигляді «мовлення ШІ» або «звернень об'єктів».

У системах з «мовленням ШІ» система ШІ зазвичай ізольована у вигляді окремого елемента ігрової архітектури. Така стратегія часто приймає форму окремого потоку або декількох потоків, в яких ШІ обчислює найкраще рішення для заданих параметрів гри. Коли ШІ приймає рішення, воно передається усім учасникам об'єктів. Такий підхід найкраще працює в стратегіях реального часу, де ШІ аналізує загальний хід подій у всій грі.

Системи з «зверненнями об'єктів» краще підходять для ігор з простими об'єктами. В таких іграх об'єкти звертаються до системи ШІ кожен раз, коли об'єкт «думає» або оновлює себе. Такий підхід відмінно підходить для систем з великою кількістю об'єктів, яким не потрібно «думати» занадто часто, наприклад в шутерах. Така система також може скористатися перевагами багатопотокової архітектури, але для неї потрібно більш складне планування.

У цій роботі буде проаналізовано як штучні нейронні мережі можна використовувати в іграх.

1 АНАЛІЗ ПРОБЛЕМНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

Відеогра — це електронна гра, в ігровому процесі якої гравець використовує інтерфейс користувача, щоб отримати зворотну інформацію з відеопристрою. Електронні пристрої, які використовуються для того щоб грати, називаються ігровими платформами. Наприклад, до таких платформ належать персональний комп'ютер та гральна консоль. Пристрій введення, який використовується для керування грою, називається ігровим контролером. Це може бути, наприклад, джойстик, клавіатура та мишка, геймпад або сенсорний екран.

У 2011 році відеоігри були офіційно визнані видом мистецтва урядом США та Національним фондом мистецтв США. Однак, загальноосвітове визнання їх мистецтвом лишається дискутивним питанням.

Відеоігри та пов'язані продукти й заходи приносять великі прибутки і мають значний вплив на інші види розваг. Світовий ринок відеоігор зростає з кожним роком, у 2017 році він був оцінений від $58 млрд до $78,6 млрд. Кількість гравців по всьому світу склала 1,8 млрд чоловік. Найбільшими регіональнимим ринками на 2015 були Північноамериканський ($23,6 млрд), Азіатський ($23,1 млрд), Європейський ($22 млрд) і, зі значним відривом, Південноамериканський ($4,5 млрд). Найбільші прибутки отримувалися від мобільних ігор ($22,3 млрд), роздрібні продажі принесли $19,7 млрд, free-to-play MMO — $8,7 млрд, ігри в соціальних мережах — $7,9 млрд, завантажувані доповнення до вже випущених відеоігор — $7,5 млрд, електронні консолі — $3,1 млрд, pay-to-play MMO — $2,7 млрд, пов'язані

відео — $1,5 млрд, кіберспорт — $612 млн і віртуальна реальність — $225 млн. Розробка відеогри на Заході зазвичай вимагає від $5 млн до $100 млн.

Розробка відеогри має низку послідовних етапів, загалом їх є три: розробка програмного (джерельного) коду, розробка контенту (малюнки, моделі, музика) та розробка ігрових механік. Їм передує проектування (пре-продакшну) — генерування геймдизайнером ідей щодо майбутньої гри, вибір жанру, тематики, особливостей ігрового процесу, розробка сценарію та образів персонажів з оточенням. Менеджер координує дії різних людей, залучених до розробки, складає план їхньої роботи, встановлює терміни її виконання, планує витрати. Готова гра в свою чергу має пройти низку етапів, в ході яких потрапляє до гравців і підтримує інтерес до себе. Індустрія відеоігор включає у себе багато людей з різними професіями та ролями: програмістів, які відповідають за технічні можливості гри, художників, моделювальників та аніматорів, які створюють графічний контент, композиторів та звукорежисерів, які створюють звукове оформлення та музичний супровід, який нерідко видається окремим накладом. За успішне завершення роботи над проектом відповідають продюсери. Відеоігри, які розробляються незалежними розробниками чи аматорами називаються інді-іграми. Такі ігри нерідко створюються за допомогою спеціальних програм, які можуть не вимагати окремо розробки коду або графіки, наприклад, як RPG Maker.

Джерельний код є основою будь-якої відеогри і відповідає за її технічні можливості, від яких залежить контент та ігровий процес. Сучасні ігри здебільшого засновані на готових програмних модулях — ігрових рушіях, де вже реалізовані базові функції, здатні зв'язувати воєдино графіку, звук, об'єкти і їх рухи. Однак програмісти все-одно мусять писати код, щоб налаштувати рушій і сповна реалізувати задуману гру. Деякі розробники створюють власні рушії для конкретної гри. Існують як вільні ігрові рушії, доступні будь-кому, так і ті, що вимагають отримання ліцензії на їх

використання. Останні як правило володіють ширшим функціоналом і використовують передові технології.

Відеоігри, так само як і музичні чи літературні твори, можна категоризувати у жанри. Щодо відеоігор жанри виділяються на основі спільних характеристик ігрового процесу або його цілей. Часто у одній відеогрі поєднуються декілька жанрів (наприклад, більшість сучасних рольових ігор мають елементи екшн). Єдино визначеної класифікації жанрів відеоігор не існує, проте в більшості класифікацій виділяються основні:

- Пригодницькі — де дія відбувається в рамках визначеної історії та передбачає детальне дослідження ігрового світу. Пригодницькі ігри менш покладаються на візуальні образи і більшою мірою на переживання сюжету. Нерідко передбачають вирішення головоломок.
- Екшн — ігри, де гравцеві слід в більшості покладатися на швидкість реакції. Дія гри, як правило, зосереджена на різного роду боях, в ході якої гравець повинен вчасно виконувати потрібні дії. Небойові завдання можуть включати в себе уникнення пасток, проходження місць за визначений час і т. д.
- Гоночні — цей жанр включає в себе всі ігри, в яких участь у перегонах різного роду є основою ігрового процесу. Зазвичай гоночні ігри використовують автомобілі та інші транспортні засоби.
- Рольові — рольові відеоігри походять від настільних рольових ігор, звідки взяли ігрову механіку. Гравець «грає роль», певного персонажа, який з часом і в міру виконання поставлених завдань розвивається.
- Стратегічні — в широкому сенсі стратегічним відеоіграми є ті, де запорукою перемоги є розв'язання проблем шляхом попереднього обдумування та планування. У більш вузькому, стратегічні ігри відтворюють збройні конфлікти, де гравець керує арміями чи країнами, що вимагає стратегічного мислення.

- Симулятори — різною мірою реалістично відтворюють якийсь з аспектів реального життя. Наприклад, існують симулятори побачень чи управління літаком.
- Навчальні — служать для навчання гравця в якійсь області. Зазвичай призначені для дітей, проте існують і навчальні ігри для дорослих.
- Спортивні — ігри, які відтворюють реальні (футбол, хокей) чи вигадані (квідич) види спорту.

Також відеоігри розрізняються за тематикою: фентезійні, детективні, жахи і т. д. За перспективою: від першої особи, від третьої особи, ізометричні, з видом збоку/згори.

Ігровий штучний інтелект (англ. Game artificial intelligence) - набір програмних методик, які використовуються в комп'ютерних іграх для створення ілюзії інтелекту в поведінці персонажів, керованих комп'ютером. Ігровий ШІ, крім методів традиційного штучного інтелекту, включає також алгоритми теорії управління, робототехніки, комп'ютерної графіки та інформатики в цілому.

Реалізація ШІ сильно впливає на геймплей, системні вимоги і бюджет гри, і розробники балансують між цими вимогами, намагаючись зробити цікавий і невимогливий до ресурсів ШІ малою ціною. Тому підхід до ігрового ШІ серйозно відрізняється від підходу до традиційного ШІ - широко застосовуються різного роду спрощення, обмани і емуляції. Наприклад: з одного боку, в шутерах від першого особи безпомилкове рух і миттєве прицілювання, властиве роботам, не залишає жодного шансу людині, так що ці здібності штучно знижуються. З іншого боку - боти повинні робити засідки, діяти командою і т. д., для цього застосовуються «милиці» у вигляді контрольних точок, розставлених на рівні.

Персонажів комп'ютерних ігор, керованих ігровим штучним інтелектом, ділять на:

- неігрові персонажі - як правило, ці персонажі є дружніми або нейтральними до людського гравцеві.
- боти - ворожі до гравця персонажі, що наближаються за можливостями до ігрового персонажу; проти гравця в будь-який конкретний момент борються невелика кількість спамерських пошукових роботів. Боти найбільш складні в програмуванні.
- моби (англ. Mob) - ворожі до гравця «нізкоінтеллектуальних» персонажі. Моби вбиваються гравцями в великих кількостях заради очок досвіду, артефактів або проходження території.

1.2 Виявлення проблем та актуалізація рішень

Сьогодні в жанрі стратегій присутня явна проблема неякісного штучного інтелекту. Я обожнюю стратегічні ігри, в багато грав, і в результаті безлічі ігор з комп'ютерним суперником, здається, розкрив таємницю штучного інтелекту.

Таємниця полягає в тому, що комп'ютер не намагається грати в гру і перемогти, а його завдання - розважати гравця. Свого роду ігровий майстер.

Ігровий майстер (англ. Gamemaster) — людина, що проводить модулі рольових ігор за вигаданим чи запозиченим сюжетом, визначає правила взаємодії персонажу зі світом і вплив світу на дії персонажа. Майстер не повинен без попереднього оголошення та узгодження із гравцями прагнути їхнього програшу. В настільних рольових іграх цю функцію виконує одна людина, у відеоіграх — спеціальна програма, в польових чи подібних — майстерська група.

Такий підхід написання штучного інтелекту відмінно підходить для ролевіх ігор, наприклад Відьмак. Ти прийшов на потрібну локацію, спрацьовує скрипт і гра тебе розважає. Але це абсолютно не підходить для стратегій, тому що стратегії - це в першу чергу гри, де люди змагаються в своєму інтелекті.

Наприклад в грі Цивілізація, ти бачиш, що поруч з тобою комп'ютерний суперник, у якого немає армії і грошей, щоб їй купити, ти вирішуєш напасти. Тут же спрацьовує скрипт і у суперника армія порівнянна з твоєї за чисельністю і юніти, які мають бонуси проти твоїх юнітів. Наприклад ти нападаєш вершниками, тоді у комп'ютера з'являться копейщики. А рівень складності регулюється тільки тим, які бонуси отримує комп'ютер. На рисунку 1.1 ви бачите бонуси, які отримує комп’ютер на останньому рівні складності.

Рисунок 1.1 — Бонуси ШІ на остнньому рівні складності.

Це читерство з боку комп'ютера призводить до того, що на початку гри гравцеві просто доводиться виживати. Однак через якийсь час, здійснюючи правильні вчинки, гравець все-одно з легкістю перемагає комп'ютер.

Тому зараз всі гравці прагнуть грати по мережі з справжніми людьми. Але по мережі так само є свої проблеми - ти стаєш прив'язаний до інших людей. З'являється людський фактор. Інший гравець може розчаруватися і покинути гру, замість нього почне грати комп'ютер. Але багатогодинна гра вже буде зіпсована, бо сусіди комп'ютера будуть в більш тепличних умовах,

ніж ті гравці, у яких в сусідах справжні люди. Тому відмінним рішенням для стратегій було б - штучний інтелект, який грає чесно і прагне перемогти в грі.

Так само такий штучний інтелект допоможе гейм-дизайнерам збалансувати гру. Якщо ми бачимо, що комп'ютер надає перевагу завжди використовувати одну стратегію - означає гра збалансована погано.

Баланс дуже важливий для стратегій. Існує 5 основних стратегій:

- нескінченне розповзання
- висока захисна імперія
- широка захисна імперія
- широка агресивна імперія
- висока агресивна імперія

Нескінченне розповзання - це історично найперша стратегія. Коли в іграх не було штрафів за велику кількість міст, найкращою стратегією було нескінченно ставити нові міста, поки вся карта не буде в містах.

Далі з появою штрафів за кількість міст, з'явився поділ на високі і широкі імперії. Висока імперія має на увазі невелику кількість міст, зате з великим числом жителів. Широка імперія передбачає велику кількість міст з невеликою кількістю жителів. Так само стратегії поділяються по агресивності. Граючи в стратегію ти завжди вибираєш - розвиватися або будувати армію. Захисна стратегія має на увазі те, що ти будуєш менше армії, тільки для захисту. Розраховувати на розвиток тільки своїми силами. Агресивна стратегія має на увазі те, що ти будуєш армію і захоплюєш сусідів. Твій розвиток відбувається завдяки захопленню міст.

В оптимальній стратегії ти можеш перемогти використовуючи будь-яку з цих стратегій.

1.3 Постановка задачі

У атестаційній роботі необхідно дослідити способи навчання штучного інтелекту в стратегіях. Для цього ми створимо прототип найпростішого бою в стратегіях і постараємося навчити комп'ютер грамотно пересувати і атакувати юнитами.

Прототип бою напишемо використовую ігровий движок Unity. Битися один з одним будуть 4 типи юнітів. Мечники, копейщики, лучники і вершники. Буде спроектована своя нейронна мережа для кожного виду юнітів. І будуть навчені з використанням генетичних алгоритмів. Результатом роботи будуть збережені ваги кращих нейронних мереж.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОГО ПРОТОТИПУ

2.1 Концепт ігрового прототипу

Ігровий прототип повинен відповідати наступним вимогам щодо технологій:

- маленька карта 6*10
- стандартне розташування юнітів;
- 4 види юнітів;
- Юніти можуть або рухатися, або атакувати;

Грають 2 гравців, кожен по черзі управляє виділеним юнітом. Це типовий бій як у грі "Герої" (рисунок 2.1).

Рисунок 2.1 — Приклад бою з «Героїв 5».

Баланс юнітів зроблений по типу "камінь, ножиці, папір". Копейщики кращі проти вершників, мечники кращі проти копейщиков, а вершники хороші проти мечников. Лучники - це юніт далекого бою. Всі юніти можуть рухатися на 2 клітини, а вершники на 4. Юніти ближнього бою можуть атакувати юніта в сусідній клітці, а далекого бою в радіусі 2-ух клітин.

2.2 Вимоги ігрового бота

Ігровий бот повинен прийняти рішеніє - куди походити або кого проатаковати. Перемога в бою - це повне знищення юнітів супротивника.

Для кожного типу юнітів потрібно створити свою нейронну мережу. На вхід нейронної мережі надходить інформація про карту і про те, де хто знаходиться. На вихід ми отримуємо інформацію про те, яку дію зробити.

3 ДОСЛІДЖЕННЯ ЕВОЛЮЦІЙНИХ АЛГОРИТМІВ

3.1 Загальні відомості про еволюційні алгоритми

У штучному інтелекті (AI) еволюційний алгоритм (EA) - це підмножина еволюційних обчислень [1], загальний алгоритм оптимізації метагевристичної популяції на основі населення. EA використовує механізми, натхненні біологічною еволюцією, такі як відтворення, мутація, рекомбінація та селекція. Кандидатські рішення проблеми оптимізації відіграють роль людей у популяції, а функція придатності визначає якість рішень (див. Також функцію втрат). Потім еволюція сукупності відбувається після неодноразового застосування вищевказаних операторів. [1].

Еволюційні алгоритми часто добре наближаються до вирішення всіх типів проблем, оскільки в ідеалі вони не дають жодних припущень про базовий фітнес-ландшафт. Методи еволюційних алгоритмів, застосованих для моделювання біологічної еволюції, як правило, обмежуються дослідженнями мікроеволюційних процесів та моделей планування, заснованих на клітинних процесах. У більшості реальних застосувань EA обчислювальна складність є забороняючим фактором. Фактично, ця обчислювальна складність обумовлена оцінкою функцій фітнесу. Наближення фітнесу - одне з рішень для подолання цієї складності. Однак, здавалося б, простий EA може вирішити часто складні проблеми; тому може не бути прямого зв’язку між складністю алгоритму та складністю проблеми.

Моделювання еволюції можна розділити на дві категорі:

1. Системи, які використовують лише еволюційні принципи. Вони успішно використовувалися для завдань виду функціональної оптимізації і

можуть легко бути описані на математичній мові. До них належать еволюційні алгоритми, такі як еволюційне програмування, генетичні алгоритми, еволюційні стратегії [2].

2. Системи, які є біологічно реалістичніші, але які не виявилися корисними в прикладному сенсі. Вони більше схожі на біологічні системи і менш направлені на вирішення технічних завдань. Вони володіють складною і цікавою поведінкою, і, мабуть, незабаром отримають практичне вживання. До цих систем відносять так зване штучне життя [3].

3. Еволюційні алгоритми, в сучасному вигляді, з'явились наприкінці 1960-х на початку 1970-х (існують посилання на раніші дослідження). Еволюційні алгоритми можна поділити на три групи:

4. Еволюційне програмування: фокусується більше на адаптації індивідів, аніж на еволюції генетичної інформації. Зазвичай, еволюційне програмування застосовує безстатеве розмноження та мутації, тобто, внесення невеликих змін в поточний розв'язок та методи селекції основані на прямій конкуренції.

5. Еволюційні стратегії (ЕС): Важливою особливістю еволюційних стратегій є використання само-адаптивних механізмів для контролю процесу мутації. Ці механізми зосереджені не лише на еволюції шуканих розв'язків, а й на еволюції параметрів мутації.

6. Генетичний алгоритм (ГА): Основною особливістю генетичних алгоритмів є використання оператора рекомбінації (схрещення) як основного механізму пошуку. Це ґрунтується на припущенні, що частини оптимального розв'язку можуть бути знайдені незалежно та рекомбіновані для отримання кращого розв'язку [4].

Можливим обмеженням [відповідно до кого?] Багатьох еволюційних алгоритмів є відсутність чіткого розрізнення генотипу та фенотипу. У природі запліднена яєчна клітина проходить складний процес, відомий як ембріогенез, щоб стати зрілим фенотипом. Вважається, що це непряме

кодування робить генетичний пошук більш надійним (тобто знижує ймовірність смертельних мутацій), а також може покращити еволюційність організму. Такі непрямі (генеральні або розвиваючі) кодування також дозволяють еволюції використовувати закономірність у навколишньому середовищі. Останні роботи в галузі штучного ембріогенезу або систем штучного розвитку прагнуть вирішити ці проблеми. І програмування експресії генів успішно досліджує систему генотипів-фенотипів, де генотип складається з лінійних мультигенних хромосом фіксованої довжини, а фенотип складається з декількох дерев експресії або комп'ютерних програм різного розміру та форми.

До інших не класичних задач, для розв'язання яких застосовано еволюційні алгоритми, належать планування, складання розкладів, обчислення маршрутів, задачі розташування та транспортування. Також еволюційні алгоритми використовують для оптимізації структур та електронних схем, в медицині та в економіці.

Можливість використання еволюційних алгоритмів у галузі музики активно досліджується насамперед у Австрії, а саме при спробі моделювання та відтворення гри на музичних інструментах видатними особистостями різних епох [5].

У штучному інтелекті і машинному навчанні еволюційні алгоритми - це розділ еволюційних обчислень, в яких використовуються моделі процесів природного відбору (розмноження, мутація, рекомбінація і відбір) і принципи природної еволюції для вирішення завдань оптимізації [6].

Рішення оптимізаційної задачі, розглядаються як особи популяції. Якість кожного рішення оцінюється за допомогою спеціальної функції придатності (fitness function - фітнес-функція), після чого відбувається еволюція популяції. Таким чином, еволюційний алгоритм містить наступні кроки:

- Формується початкова популяція методом випадкового відбору (перше покоління).
- Оцінюється придатність кожного члена популяції за допомогою фітнес-функції.
- Повторюються наступні дії (еволюція):
  - відбір - вибір найбільш пристосованих особин для розмноження (батьки);
  - розмноження - формування нових особин шляхом схрещування і мутації, і оцінка їх придатності;
  - рекомбінація - найменш пристосовані особини попереднього покоління замінюються найбільш пристосованими особинами нового покоління.

Еволюційні алгоритми мають наступні переваги:

- застосовні до широкого класу задач оптимізації;
- легко комбінуються з іншими методами;
- дозволяють отримувати добре інтерпретуються результати;
- мають інтерактивністю - можна включити в популяцію запропоновані користувачем рішення;
- розглядають велику кількість альтернативних рішень.

До недоліками еволюційних алгоритмів можна віднести:

- відсутність гарантії знаходження оптимального рішення за кінцеве час - алгоритм реалізує локальну оптимізацію;
- слабка теоретична база - алгоритм є евристичним, тобто точність і строгість постановки приносяться в жертву можливості бути реалізованим;
- складність у використанні - може знадобитися налаштування параметрів моделі, що оптимізується за допомогою еволюційного алгоритму;
- високі обчислювальні витрати - алгоритм не є масштабованим.

Еволюція - ітеративний процес. Ми використовуємо корутіну, щоб забезпечити продовження такого циклу. Якщо коротко, то ми почнемо з певного геному і створимо кілька мутованих копій. Для кожної копії ми створимо екземпляр істоти і протестуємо його роботу в симуляції. Потім ми візьмемо геном найбільш успішного істоти і повторимо ітерацію[7].

3.2 Еволюційне программування

Еволюційне програмування - одна з чотирьох основних парадигм еволюційного алгоритму. Це аналогічно генетичному програмуванню, але оптимізована структура програми, що оптимізується, при цьому її числові параметри дозволяють еволюціонувати [8].

Його вперше застосував Лоуренс Дж. Фогель у США в 1960 році для того, щоб використовувати імітаційну еволюцію як навчальний процес, спрямований на генерування штучного інтелекту. Фогель використовував кінцеві машини як предиктори і розвивав їх. В даний час еволюційне програмування - це широкий еволюційний діалект обчислень без фіксованої структури або (подання) на відміну від деяких інших діалектів. Це стає важче відрізнити від еволюційних стратегій.

Основним його оператором варіації є мутація; члени популяції розглядаються як частина конкретного виду, а не представники одного виду, тому кожен батько породжує потомство, використовуючи (μ + μ) [потрібне додаткове пояснення] відбір виживців.

Пошук залежності цільових змінних від інших проводиться у формі функцій якогось певного виду. Наприклад, в одному з найвдаліших алгоритмів цього типу — методі групового урахування аргументів (МГУА) залежність шукають у формі поліномів. Причому складні поліноми

заміняються декількома простими, враховують лише деякі ознаки (групи аргументів). Зазвичай використовуються попарні об'єднання ознак. Цей метод не має великих переваг в порівнянні з нейронними мережами з готовим набором стандартних нелінійних функцій, але, отримані формули залежності, в принципі, піддаються аналізу і інтерпретації (хоча на практиці це все-таки складно) [9].

У штучному інтелекті генетичне програмування (GP) - це техніка еволюціонуючих програм, починаючи з групи непридатних (як правило, випадкових) програм, придатних для певного завдання, застосовуючи операції, аналогічні природним генетичним процесам до популяції програм. Це, по суті, евристична техніка пошуку, яку часто характеризують як «сходження на гору», тобто пошук оптимальної або принаймні відповідної програми серед простору всіх програм. [10].

Операції: вибір найкращих програм для відтворення (кросовер) та мутації відповідно до заздалегідь визначеної міри фітнесу, як правило, володіння потрібним завданням. Операція кросовера включає обмінювання випадкових частин вибраних пар (батьків) для отримання нового і іншого потомства, що стає частиною програм нового покоління. Мутація включає заміну деякої випадкової частини програми на якусь іншу випадкову частину програми. Деякі програми, не вибрані для відтворення, копіюються з поточного в нове покоління. Потім відбір та інші операції рекурсивно застосовуються до програм нового покоління. [11].

СЗазвичай члени кожного нового покоління в середньому більше підходять, ніж члени попереднього покоління, і програма найкращого покоління часто краще, ніж програми найкращого покоління попередніх поколінь. Припинення рекурсії відбувається тоді, коли якась окрема програма досягає заздалегідь визначеного рівня кваліфікації чи придатності. [12].

Може, і часто трапляється, що певний запуск алгоритму призводить до передчасного зближення до деякого локального максимуму, що не є

глобальним оптимальним або навіть хорошим рішенням. Для отримання дуже хорошого результату зазвичай необхідні кілька пробігів (десятки до сотні). Також може знадобитися збільшити розмір і мінливість особин, щоб уникнути патологій [13].

Перший запис пропозиції щодо розвитку програм - це, мабуть, запис Алана Тьюрінга в 1950 році. Перед публікацією Джона Холланда «Адаптація в природних і штучних системах» було розрив за 25 років, що виклав теоретичні та емпіричні основи науки. У 1981 році Річард Форсайт продемонстрував успішну еволюцію невеликих програм, представлених як дерева, для проведення класифікації доказів місця злочину для внутрішнього офісу Великобританії. [14].

Хоча ідея еволюціонуючих програм, спочатку на комп'ютерній мові Лісп, була актуальною серед студентів Джона Холланда, тільки тоді, коли вони організували першу конференцію з генетичних алгоритмів у Пітсбурзі, Ніхайл Креймер опублікував розвинені програми у двох спеціально розроблених мови, які включали перше твердження сучасного генетичного програмування на основі «дерев» (тобто процедурні мови, організовані в структурах на основі дерев і керовані відповідними GA-операторами). У 1988 році Джон Коза (також аспірант Джона Холланда) запатентував свій винахід GA для еволюції програми. Далі було опубліковано Міжнародну спільну конференцію зі штучного інтелекту IJCAI-89.

Коза слідував за цим у 205 публікаціях на тему "Генетичне програмування" (GP), ім'я якого придумав Девід Голдберг, також доктор наук Джона Холланда. Однак, серія з 4-х книг Коза, починаючи з 1992 року із супровідними відеозаписами, справді створила GP. Згодом відбулося величезне розширення кількості публікацій із Бібліографією генетичного програмування, що перевищило 10 000 записів. У 2010 році Коза перерахував 77 результатів, коли генетичне програмування було конкурентоспроможним для людини [15].

У 1996 році Коза розпочав щорічну конференцію з генетичного програмування, після якої у 1998 р. Відбулася щорічна конференція EuroGP та перша книга у серії GP, яку редагував Коза. У 1998 році побачив перший підручник із загальної практики. Рок продовжував процвітати, що призвело до першого спеціалізованого журналу про загальну лікарську практику, а через три роки (2003 р.) Рік Ріоло створив щорічний семінар теорії та практики генетичного програмування (GPTP). Документи з генетичного програмування продовжують публікуватися на різноманітних конференціях та пов'язаних журналах. Сьогодні існує дев'ятнадцять книг загальної практики, в тому числі кілька для студентів.

Рання робота, яка створює підґрунтя для поточних тем та застосувань для генетичного програмування, різноманітна і включає синтез та ремонт програмного забезпечення, прогнозне моделювання, обмін даними, фінансове моделювання, м'які датчики, дизайн, та обробка зображень. Застосування в деяких областях, таких як дизайн, часто використовують проміжні представлення, наприклад, клітинне кодування Фреда Груу. Промисловий процес був значним у кількох сферах, включаючи фінанси, хімічну промисловість, біоінформатику та металургійну промисловість.

GP розвиває комп’ютерні програми, традиційно представлені в пам'яті у вигляді деревних структур. Дерева можна легко оцінити рекурсивно. Кожен вузол дерева має функцію оператора, а кожен термінальний вузол має операнд, що робить математичні вирази легко розвиватися та оцінювати. Таким чином, традиційно GP надає перевагу використанню мов програмування, які природно втілюють структури дерева (наприклад, Lisp; інші функціональні мови програмування також підходять).

Недеревні уявлення були запропоновані та успішно реалізовані, наприклад, лінійне генетичне програмування, яке відповідає більш традиційним імперативним мовам [див., Наприклад, Banzhaf et al. (1998)]. Комерційне програмне забезпечення GP Discipulus використовує автоматичну індукцію двійкового машинного коду ("AIM") для досягнення

кращих показників. μGP використовує спрямовані мультиграфи для створення програм, які повністю використовують синтаксис заданої мови складання. Інші програмні представлення, на яких були проведені значні дослідження та розробки, включають програми для віртуальних машин на основі стека та послідовності цілих чисел, які відображаються у довільних мовах програмування за допомогою граматик. Картезіанське генетичне програмування - це ще одна форма GP, яка використовує графічне подання замість звичайного представлення на основі дерева для кодування комп'ютерних програм.

Більшість представлень мають структурно неефективний код (інтрони). Такі гени, що не кодують, можуть здатися марними, оскільки вони не впливають на продуктивність жодної людини. Однак вони змінюють ймовірність породження різних нащадків за допомогою операторів варіацій, і, таким чином, змінюють варіаційні властивості індивіда. Експерименти, схоже, демонструють більш швидку конвергенцію при використанні програмних уявлень, що дозволяють такі не кодують гени, порівняно з програмними уявленнями, які не мають жодних некодуючих генів.

Відбір - це процес, при якому певні особи вибираються з поточного покоління, які послужать батьками для наступного покоління. Люди вибираються ймовірнісно так, що більш якісні особи мають більший шанс отримати вибір. Найпоширенішим методом відбору в GP є турнірний відбір, хоча було показано, що інші методи, такі як пропорційний відбір фітнесу, вибір лексикази та інші, мають більшу ефективність для багатьох проблем з ГП.

Елітаризм, який передбачає висівання наступного покоління з найкращого індивіда (або кращих російських особин) з нинішнього покоління, - це техніка, яка іноді використовується для уникнення регресії.

До особи, відібраних на етапі відбору, описаному вище, застосовуються різні генетичні оператори (тобто кросовер та мутація) для

розведення нових особин. Швидкість, з якою застосовуються ці оператори, визначає різноманітність у сукупності.

Відверніть один або кілька біт від попереднього потомства, щоб створити нову дитину чи покоління.

GP успішно використовується як автоматичний інструмент програмування, інструмент машинного навчання та механізм автоматичного вирішення проблем. GP особливо корисний у областях, де точна форма рішення не відома заздалегідь або є приблизним рішенням (можливо, тому що знайти точне рішення дуже важко). Деякі із застосувань GP - це встановлення кривих, моделювання даних, символічна регресія, вибір особливостей, класифікація тощо. Джон Р. Коза згадує 76 випадків, коли генетичне програмування могло дати результати, які є конкурентоспроможними за результатами, отриманими людиною (звані людиною -конкурентні результати). Починаючи з 2004 року, щорічна конференція з генетичних та еволюційних обчислень (GECCO) проводить змагання за конкурентоспроможність людини (називається Humies), де грошові нагороди вручаються за результати конкурентоспроможності людини, отримані за будь-якої форми генетичних та еволюційних обчислень. За цей рік GP виграв багато нагород у цьому конкурсі.

Метагенетичне програмування - це запропонована методика навчання мета розвитку генетичної системи програмування з використанням самого генетичного програмування. Це дозволяє припустити, що самі хромосоми, кросовер та мутація еволюціонували, тому, як і їхнім реальним колегам, потрібно дозволити змінюватися самостійно, а не визначатись програмою людини. Meta-GP був офіційно запропонований Юргеном Шмідхубером у 1987 р. Еуріско Дуга Лената - це попереднє зусилля, яке може бути тією ж технікою. Це рекурсивний, але закінчуючий алгоритм, що дозволяє йому уникнути нескінченної рекурсії. У підході "автоконструктивного розвитку" до метагенетичного програмування методи виробництва та зміни потомства

закодовані в межах самих розвиваються програм, а програми виконуються для створення нових програм, що додаються до населення.

Критики цієї ідеї часто кажуть, що цей підхід є надзвичайно широким за обсягом. Однак, можливо, можна обмежити критерій придатності до загального класу результатів, і таким чином отримати розвинуту загальну групу, яка б більш ефективно давала результати для підкласів. Це може мати форму мета-розвинутого GP для створення алгоритмів ходьби людини, який потім використовується для розвитку бігу, стрибків і т.д.

3.3 Еволюційні стратегії.

В інформатиці стратегія еволюції (ES) - це технологія оптимізації, заснована на ідеях еволюції. Він належить до загального класу методологій еволюційного обчислення чи штучної еволюції.

Техніка оптимізації "стратегії еволюції" була створена на початку 1960-х, а далі розвинута в 1970-х, а пізніше Інго Рехенберг, Ганс-Пол Швефель та їхні колеги. [16].

Еволюційні стратегії використовують в якості операторів пошуку природні залежні від проблеми уявлення, і в першу чергу мутацію та вибір. Як і еволюційні алгоритми, оператори застосовуються в циклі. Ітерація циклу називається генерацією. Послідовність поколінь продовжується до тих пір, поки не буде виконано критерій припинення. [17].

Для пошукових просторів у реальному значенні мутація виконується шляхом додавання нормально розподіленого випадкового вектора. Розмір кроку або сила мутації (тобто стандартне відхилення нормального розподілу) часто регулюється самоадаптацією (див. Вікно еволюції). Індивідуальні розміри кроків для кожної координати або кореляції між координатами, які

по суті визначаються базовою коваріаційною матрицею, на практиці керуються або самоадаптацією, або адаптацією матриці коваріації (CMA-ES). Коли крок мутації виводиться з багатоваріантного нормального розподілу з використанням еволюціонуючої коваріаційної матриці, було висунуто гіпотезу, що ця адаптована матриця наближає обернену гессіану до ландшафту пошуку. Ця гіпотеза доведена для статичної моделі, що спирається на квадратичне наближення [18].

Вибір (екологічний) стратегій еволюції є детермінованим і базується лише на рейтингу фітнесу, а не на фактичних значеннях фітнесу. Отже, отриманий алгоритм є інваріантним щодо монотонних перетворень цільової функції. Найпростіша еволюційна стратегія діє на сукупність двох розмірів: поточну точку (батьківську) та результат її мутації. Тільки якщо придатність мутанта принаймні така ж хороша, як і батьківська, вона стає батьком наступного покоління. В іншому випадку мутант не враховується. Це (1 + 1) -ES. Більш загально, λ мутанти можуть генеруватися та конкурувати з батьківським, що називається (1 + λ) -ES. У (1, λ) -ES кращий мутант стає батьком наступного покоління, тоді як поточний батьків завжди не враховується. Для деяких із цих варіантів докази лінійної конвергенції (у стохастичному сенсі) отримані на одномодальних об'єктивних функціях.

Сучасні похідні стратегії еволюції часто використовують сукупність μ батьків та рекомбінацію як додатковий оператор, який називається (μ / ρ +, λ) -ES. Це робить їх менш схильними до поселення в місцевих оптимах. [19].

При пошуку рішення в еволюційної стратегії спочатку відбувається мутація і схрещування особин для отримання нащадків, потім відбувається детермінований відбір без повторень кращих особин із загального покоління батьків і нащадків. Як мутації часто використовується додавання нормально розподіленої випадкової величини до кожної компоненті хромосоми.

3.4 Генетичний алгоритм

У галузі інформатики та досліджень операцій генетичний алгоритм (GA) - це метагеврист, натхненний процесом природного відбору, що належить до більшого класу еволюційних алгоритмів (EA). Генетичні алгоритми зазвичай використовуються для створення високоякісних рішень для оптимізації та пошуку проблем, спираючись на біологічно натхненні оператори, такі як мутація, кросовер та вибір. [1] Джон Голланд представив генетичні алгоритми в 1960 р. На основі концепції теорії еволюції Дарвіна; його студент Девід Е. Голдберг ще більше продовжив GA в 1989 році. [21].

У генетичному алгоритмі сукупність кандидатських рішень (які називаються індивідами, істотами або фенотипами) проблеми оптимізації розвивається до кращих рішень. Кожен кандидатський розчин має набір властивостей (його хромосоми або генотип), які можна мутувати і змінювати; традиційно рішення представлені у двійковій формі як рядки 0s та 1s, але можливі й інші кодування.

Еволюція, як правило, починається з популяції випадково генерованих особин і є ітераційним процесом, причому популяція в кожній ітерації називається поколінням. У кожному поколінні оцінюється придатність кожної особи в популяції; придатність зазвичай є значенням цільової функції в задачі оптимізації, що вирішується. Більш придатні особини вибираються стохастично з поточної сукупності, і геном кожної особи видозмінюється (рекомбінований і, можливо, випадково мутований), щоб утворити нове покоління. Нове покоління кандидатських рішень потім використовується при наступній ітерації алгоритму. Зазвичай алгоритм припиняється, коли було вироблено максимальну кількість поколінь або досягнуто задовільного рівня придатності для населення.

Стандартне представлення кожного рішення-кандидата - це масив бітів. [3] Масиви інших типів та структур можна використовувати по суті таким же чином. Основна властивість, яка робить ці генетичні уявлення зручними,

полягає в тому, що їх частини легко вирівнюються завдяки фіксованому розміру, що полегшує прості операції кросовера. Також можуть бути використані представлення змінної довжини, але в цьому випадку виконання кросовера є складнішим. Деревоподібні уявлення досліджуються в генетичному програмуванні, а графічні подання досліджуються в еволюційному програмуванні; в програмуванні експресії генів досліджується поєднання як лінійних хромосом, так і дерев [22].

Після того, як визначено генетичне представлення та фітнес-функція, GA переходить до ініціалізації популяції розчинів, а потім для вдосконалення її шляхом повторного застосування операторів мутації, кросовера, інверсії та селекції.

Кількість населення залежить від характеру проблеми, але зазвичай містить кілька сотень чи тисяч можливих рішень. Часто початкова сукупність генерується випадковим чином, дозволяючи весь спектр можливих рішень (простір пошуку). Іноді розчини можуть бути «засіяні» на ділянках, де можливі оптимальні рішення.

Під час кожного наступного покоління частина існуючої популяції відбирається для розведення нового покоління. Індивідуальні рішення вибираються за допомогою процесу, заснованого на фітнесі, де зазвичай більше шансів на вибір фітнес-рішень (як вимірюють функції фітнесу). Певні методи відбору оцінюють придатність кожного рішення та переважно вибирають найкращі рішення. Інші методи оцінюють лише випадкову вибірку сукупності, оскільки колишній процес може бути дуже трудомістким. (рисунок 3.1)

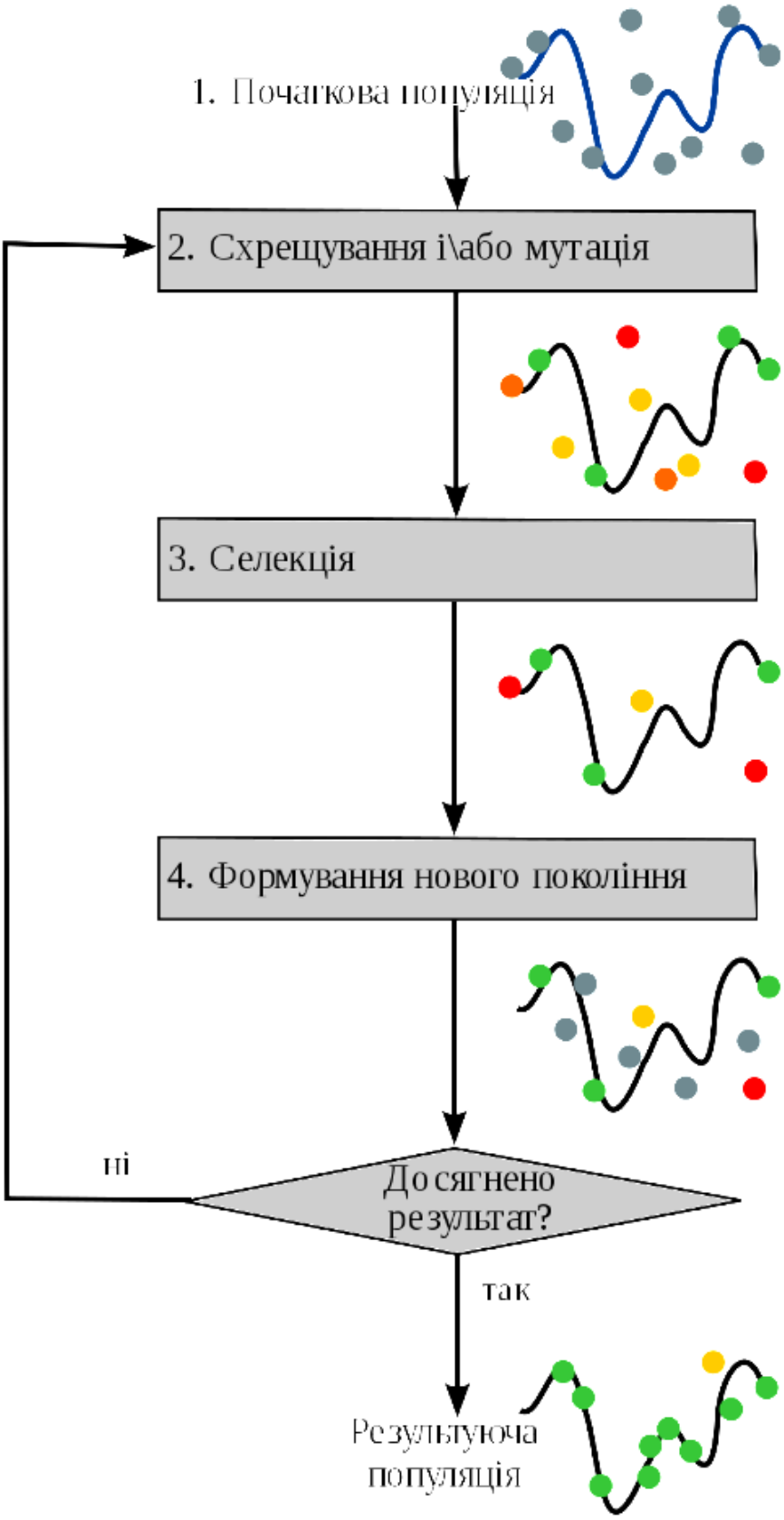

Рисунок 3.1 — Генетичний алгоритм.

Функція фітнесу визначається генетичним представленням і вимірює якість представленого рішення. Функція фітнесу завжди залежить від проблем. Наприклад, у проблемі з рюкзаком хочеться максимально збільшити загальне значення об'єктів, які можна помістити в рюкзак певної фіксованої ємності. Представлення рішення може представляти собою масив бітів, де кожен біт являє собою інший об'єкт, а значення біта (0 або 1) являє собою, чи є об'єкт у рюкзаку чи ні. Не кожне таке подання є дійсним, оскільки розмір предметів може перевищувати ємність рюкзака. Придатність рішення - це сума значень усіх об'єктів у рюкзаку, якщо представлення дійсне, або 0 в іншому випадку. [23]

У деяких проблемах визначити фітнес-вираз важко або навіть неможливо; у цих випадках може бути використане моделювання для визначення значення функції придатності фенотипу (наприклад, обчислювальна динаміка рідини використовується для визначення опору повітря транспортного засобу, форма якого кодується як фенотип), або навіть використовуються інтерактивні генетичні алгоритми.

Наступним кроком є генерування сукупності рішень другого покоління з тих, що відбираються за допомогою комбінації генетичних операторів: кросовер (також званий рекомбінацією) та мутації.

Для кожного нового розчину, що випускається, для виведення з пулу, обраного раніше, вибирається пара «батьківських» розчинів. Виробляючи «дочірнє» рішення з використанням вищевказаних методів кросовер та мутації, створюється нове рішення, яке, як правило, поділяє багато характеристик своїх «батьків». Для кожної нової дитини вибираються нові батьки, і процес триває до тих пір, поки не сформується нова сукупність рішень відповідного розміру. Хоча методи розмноження, засновані на використанні двох батьків, є більш «натхненними біологією», деякі дослідження припускають, що більше двох «батьків» генерують хромосоми вищої якості. [24].

Ці процеси в кінцевому рахунку призводять до того, що хромосоми наступного покоління відрізняються від початкових поколінь. Взагалі середня придатність збільшиться за цією процедурою для населення, оскільки для розмноження відбираються лише найкращі організми першого покоління, а також невелика частка менш придатних розчинів. Ці менш придатні рішення забезпечують генетичне різноманіття в генетичному пулі батьків і, отже, забезпечують генетичне різноманіття наступного покоління дітей.

Хоча кросовер та мутація відомі як основні генетичні оператори, в генетичних алгоритмах можливе використання інших операторів, таких як перегрупування, колонізація-вимирання або міграція.

Варто налаштувати параметри, такі як ймовірність мутації, ймовірність перехрестя та розмір сукупності, щоб знайти розумні параметри для класу проблем, над яким працює. Дуже невеликий коефіцієнт мутації може призвести до генетичного дрейфу (який має неергодичний характер). Занадто висока швидкість рекомбінації може призвести до передчасного зближення генетичного алгоритму. Занадто висока швидкість мутації може призвести до втрати хороших розчинів, якщо не буде застосовано елітарний вибір [25].

На додаток до основних операторів, наведених вище, для еволюції обчислення швидше або більш надійно може бути використана інша евристика. Евристика специфікації карає схрещування між занадто подібними рішеннями; це заохочує різноманітність населення та допомагає запобігти передчасній конвергенції до менш оптимального рішення [26].

Цей процес покоління повторюється до досягнення умови припинення. Загальні умови припинення:

- Знайдено рішення, яке відповідає мінімальним критеріям
- Фіксована кількість досягнутих поколінь
- Досягнуто виділений бюджет (час обчислення / гроші)

- Придатність вищого рейтингу досягає або досягла такого плато, що послідовні ітерації більше не дають кращих результатів

- Ручний огляд

- Поєднання перерахованого.

Найпростіший алгоритм представляє кожну хромосому як біт-рядок. Зазвичай числові параметри можуть бути представлені цілими числами, хоча можна використовувати подання з плаваючою комою. Представлення з плаваючою комою є природним для еволюційних стратегій та еволюційного програмування. Поняття про генетичні алгоритми в реальному значенні було запропоновано, але насправді є неправильним, оскільки воно насправді не представляє теорії будівельних блоків, запропонованої Джоном Генрі Голланом у 1970-х. Ця теорія не позбавлена підтримки, базуючись на теоретичних та експериментальних результатах (див. Нижче). Основний алгоритм виконує кросовер та мутацію на бітовому рівні. Інші варіанти розглядають хромосому як перелік чисел, які є індексами в таблиці інструкцій, вузлами у зв’язаному списку, хешах, об'єктах або будь-якій іншій структурі даних, яку можна уявити. Кросовер та мутація виконуються так, щоб поважати межі елементів даних. Для більшості типів даних можуть бути спроектовані конкретні оператори варіацій. Здається, що різні типи хромосомних даних працюють краще або гірше для різних конкретних проблемних областей.

Коли використовуються бітові рядкові представлення цілих чисел, часто використовується кодування сірого кольору. Таким чином, невеликі зміни цілого числа можуть бути легко вплинуті за допомогою мутацій або перехресних схем. Це виявило, що допомагає запобігти передчасній конвергенції на так званих стінках Хеммінга, в яких має відбутися занадто багато одночасних мутацій (або кросовер), щоб змінити хромосому на краще рішення.

Інші підходи передбачають використання масивів дійсних значень чисел замість бітових рядків для представлення хромосом. Результати теорії

схем говорять про те, що в цілому чим менший алфавіт, тим краща ефективність, але спочатку дослідників це здивувало, що хороші результати були отримані завдяки використанню реальних цінних хромосом. Це було пояснено як набір реальних значень у кінцевій популяції хромосом як формування віртуального алфавіту (коли добір та рекомбінація є домінуючим) зі значно меншою кардинальністю, ніж можна було б очікувати від подання з плаваючою точкою [27].

Розширення доступної для генетичного алгоритму проблемної області може бути отримано за допомогою більш складного кодування пулів розчинів шляхом об'єднання декількох типів гетерогенно кодованих генів в одну хромосому. [18] Цей конкретний підхід дозволяє вирішити проблеми оптимізації, які вимагають сильно розрізнених областей визначення параметрів проблеми. Наприклад, у проблемах каскадної настройки регулятора внутрішня структура контролера циклу може належати звичайному регулятору з трьох параметрів, тоді як зовнішній цикл може реалізовувати лінгвістичний контролер (наприклад, нечітка система), який має суттєво інший опис. Ця особлива форма кодування вимагає спеціалізованого кросовер-механізму, який рекомбінує хромосому за розрізом, і це корисний інструмент для моделювання та моделювання складних адаптивних систем, особливо процесів еволюції.

Паралельна реалізація генетичних алгоритмів відбувається у двох варіантах. Грубозернисті паралельні генетичні алгоритми передбачають популяцію на кожному з комп'ютерних вузлів та міграцію особин між вузлами. Дрібнозернисті паралельні генетичні алгоритми передбачають, що індивід на кожному вузлі процесора, який діє з сусідніми особами для відбору та відтворення. Інші варіанти, як генетичні алгоритми для проблем з оптимізацією в Інтернеті, вводять залежність від часу або шум у функції фітнесу.

Генетичні алгоритми з адаптаційними параметрами (адаптивні генетичні алгоритми, АГА) - ще один важливий та перспективний варіант

генетичних алгоритмів. Ймовірність схрещування (pc) та мутації (pm) значно визначає ступінь точності розчину та швидкість конвергенції, які генетичні алгоритми можуть отримати. Замість використання фіксованих значень pc та pm, AGA використовують інформацію про населення у кожному поколінні та адаптивно коригують pc та pm, щоб підтримувати різноманітність населення, а також підтримувати здатність до конвергенції. У AGA (адаптивний генетичний алгоритм) коригування pc та pm залежить від значень придатності розчинів. У CAGA (адаптивний генетичний алгоритм на основі кластеризації) через використання кластерного аналізу для судження про стан оптимізації популяції, коригування ПК та pm залежить від цих станів оптимізації. Комбінувати GA з іншими методами оптимізації може бути досить ефективно. GA, як правило, є досить хорошим у пошуку загальних глобальних рішень, але досить неефективним у пошуку кількох останніх мутацій, щоб знайти абсолютний оптимум. Інші прийоми (такі як просте сходження на гірку) досить ефективні для пошуку абсолютного оптимального в обмеженому регіоні. Чергування GA та альпіністського сходження може підвищити ефективність [необхідне цитування], подолавши відсутність стійкості підйому на гірські сходи.

Це означає, що правила генетичної варіації можуть мати різний зміст у природному випадку. Наприклад, за умови, що етапи зберігаються в послідовному порядку, перетинання може підсумовувати кілька кроків від материнської ДНК, додаючи кількість кроків від батьківської ДНК тощо. Це як додавання векторів, які, ймовірно, можуть слідувати хребтом у фенотипічному ландшафті. Таким чином, ефективність процесу може бути підвищена на багато порядків. Більше того, оператор інверсії має можливість розміщувати кроки в послідовному порядку або будь-якому іншому підходящому порядку на користь виживання або ефективності.

Варіація, коли популяція загалом розвивається, а не окремі її члени, відома як рекомбінація генофонду.

Було розроблено ряд варіацій для спроби покращити продуктивність ГА щодо проблем з високим ступенем епістазу придатності, тобто, коли придатність рішення складається з взаємодіючих підмножин його змінних. Такі алгоритми мають на меті вивчити (перш ніж використовувати) ці корисні фенотипічні взаємодії. Таким чином, вони узгоджуються з гіпотезою будівельного блоку в адаптивному зменшенні руйнівної рекомбінації. Видатними прикладами такого підходу є mGA, GEMGA та LLGA.

Проблеми, які, як видається, є особливо підходящими для вирішення генетичними алгоритмами, включають проблеми з розкладом та плануванням, і багато програмних пакетів для планування базуються на ГА [потрібне цитування]. GA також застосовуються до машинобудування. Генетичні алгоритми часто застосовуються як підхід до вирішення проблем глобальної оптимізації.

Як правило, генетичні алгоритми великого пальця можуть бути корисні в проблемних областях, які мають складний ландшафт фітнесу як змішування, тобто мутація в поєднанні з кросовером призначена для відсторонення населення від місцевої оптими, що традиційний алгоритм сходження на гірку може застрягнути п. Зауважте, що широко використовувані кросовер оператори не можуть змінити будь-яку єдину сукупність. Мутація сама по собі може забезпечити ергодичність загального процесу генетичного алгоритму (розглядається як ланцюг Маркова).

Приклади проблем, що вирішуються генетичними алгоритмами, включають: дзеркала, призначені для перенаправлення сонячного світла до сонячного колектора, антени, призначені для отримання радіосигналів у космосі, методи ходьби для комп'ютерних фігур, оптимальне проектування аеродинамічних тіл у складні потоки.

У 1950 році Алан Тьюрінг запропонував "навчальну машину", яка би паралельна принципам еволюції. Комп'ютерне моделювання еволюції розпочалося ще в 1954 р. З роботи Нілса Алла Баррічеллі, який використовував комп'ютер в Інституті підвищення кваліфікації в Принстоні,

штат Нью-Джерсі. Його публікація 1954 року не була широко помічена. Починаючи з 1957 р. австралійський кількісний генетик Алекс Фрейзер опублікував серію праць про моделювання штучного відбору організмів з декількома локусами, що контролюють вимірювану ознаку. З цього початку комп'ютерне моделювання еволюції біологами стало більш поширеним на початку 1960-х років, а методи були описані в книгах Фрейзера та Бернелла (1970) та Кросбі (1973). Моделювання Фрейзера включало всі істотні елементи сучасних генетичних алгоритмів. Крім того, у 60-х роках Ганс-Йоахім Бремерман опублікував серію статей, в яких також було прийнято рішення вирішення проблем оптимізації, що зазнали рекомбінації, мутації та відбору. Дослідження Бремермана також включали елементи сучасних генетичних алгоритмів. Інші примітні ранні піонери - Річард Фрідберг, Джордж Фрідман та Майкл Конрад. Багато ранніх робіт передруковано Фогелем (1998).

Хоча Баррічеллі в роботі, яку він повідомив у 1963 році, імітував еволюцію здатності грати у просту гру, штучна еволюція стала лише широко відомим методом оптимізації в результаті роботи Інго Рехенберга та Ганса-Пола Швефеля в 1960-ті та початку 1970-х років - група Рехенберга змогла вирішити складні інженерні проблеми за допомогою еволюційних стратегій.Іншим підходом стала методика еволюційного програмування Лоуренса Дж. Фогеля, яка була запропонована для генерування штучного інтелекту. Еволюційне програмування спочатку використовувало машини з кінцевим станом для прогнозування середовищ і використовувало варіації та вибір для оптимізації логіки прогнозування. Зокрема, генетичні алгоритми стали популярними завдяки роботі Джона Голланда на початку 1970-х, зокрема, його книзі «Адаптація в природних та штучних системах» (1975). Його робота виникла з досліджень стільникових автоматів, проведених Голландією та його студентами Мічиганського університету. Голландія запровадила формалізовану основу для прогнозування якості наступного покоління, відому як теорема схеми Голландії. Дослідження в GA

залишалися в основному теоретичними до середини 1980-х, коли в Піттсбургу, штат Пенсильванія, відбулася Перша міжнародна конференція з генетичних алгоритмів.

В кінці 1980-х General Electric почав продавати перший у світі генетичний алгоритм, набір інструментальних інструментів на основі мейнфреймів, призначених для промислових процесів. У 1989 році компанія Axcelis, Inc. випустила перший у світі комерційний продукт GA для настільних комп'ютерів Evolver. Письменник з технологій New York Times Джон Маркофф писав про Evolver в 1990 році, і це був єдиний інтерактивний комерційний генетичний алгоритм до 1995 року. Evolver був проданий Palisade у 1997 році, перекладений декількома мовами, і наразі він перебуває у 6-й версії. Починаючи з 1990-х, MATLAB створив три евристичні алгоритми оптимізації без похідних (імітаційний відпал, оптимізація рою частинок, генетичний алгоритм) та два алгоритми прямого пошуку (симплексний пошук, пошук шаблонів) .

3.5 Нейроеволюція та система класифікатора навчання

Нейроеволюція, або нейро-еволюція, - це форма штучного інтелекту, яка використовує еволюційні алгоритми для створення штучних нейронних мереж (ANN), параметрів, топології та правил. Найчастіше застосовується в штучному житті, загальній грі та еволюційній робототехніці. Основна перевага полягає в тому, що нейроеволюцію можна застосовувати ширше, ніж керовані алгоритми навчання, для яких потрібна програма правильних пар вхід-вихід. Навпаки, нейроеволюція вимагає лише міри ефективності мережі при виконанні завдання. Наприклад, результат гри (тобто виграв чи програв один гравець) можна легко виміряти, не надаючи помічені приклади

бажаних стратегій. Нейроеволюція зазвичай використовується як частина парадигми навчального підкріплення, і вона може протиставлятися звичайним методам глибокого навчання, що використовують градієнтний спуск по нейронній мережі з фіксованою топологією. [28].

Визначено багато алгоритмів нейроеволюції. Одне загальне розмежування - між алгоритмами, які змінюють лише силу ваги зв'язку для фіксованої мережевої топології (іноді їх називають звичайною нейроеволюцією), на відміну від тих, що розвиваються як топологією мережі, так і її вагами (званими TWEANN, для топології та ваги Розвиваючі алгоритми штучної нейронної мережі) [29].

Більшість нейронних мереж використовують градієнтне спускання, а не нейроеволюцію. Однак приблизно в 2017 році дослідники Uber заявили, що виявили, що прості алгоритми структурної нейроеволюції є конкурентоспроможними сучасними галузевими стандартними алгоритмами глибокого вивчення градієнта, частково тому, що виявлено, що нейроеволюція рідше застряє в локальних мінімумах. У науці журналіст Меттью Хатсон припускав, що частина причин нейроеволюції є успішною там, де вона раніше не вдалася, через збільшення обчислювальної потужності, наявної в 2010-х [30].

Еволюційні алгоритми працюють на популяції генотипів (їх також називають геномами). У нейроеволюції генотип відображається у фенотипі нейронної мережі, який оцінюється за певним завданням, щоб отримати його придатність [31].

У схемах прямого кодування генотип безпосередньо співпадає з фенотипом. Тобто кожен нейрон і з'єднання в нейронній мережі конкретизуються безпосередньо і явно в генотипі. Навпаки, у схемах непрямого кодування генотип побічно визначає, як повинна формуватися ця мережа [32].

Традиційно опосередковані кодування, які використовують штучну ембріогенію (також відому як штучна розробка), класифікують за граматичним підходом та підходом до клітинної хімії. Перший розробляє набори правил у вигляді граматичних систем переписування. Останній намагається імітувати те, як фізичні структури виникають у біології за допомогою експресії генів. Системи непрямого кодування часто використовують аспекти обох підходів. [33].

Стенлі та Мійкукулайн пропонують систематику ембріогенних систем, яка має відображати їх основні властивості [34] [35] [36]. Таксономія визначає п'ять безперервних вимірів, уздовж яких можна розмістити будь-яку ембріогенну систему:

- Доля клітини (нейрона): кінцеві характеристики та роль клітини у зрілому фенотипі. Цей вимір підраховує кількість методів, які використовуються для визначення долі клітини.

- Націлювання: метод, за допомогою якого з'єднання спрямовуються від вихідних комірок до цільових комірок. Це варіюється від конкретного націлювання (джерело та ціль явно визначені) до відносного націлювання (наприклад, на основі розташування комірок відносно один одного).

- Гетерохронія: терміни та впорядкування подій під час ембріогенії. Підраховує кількість механізмів зміни часу проведення подій.

- Каналізація: наскільки толерантний геном до мутацій (крихкості). Варіюється від того, щоб вимагати точних генотипних інструкцій, до високої толерантності неточної мутації.

- Комплексизація: здатність системи (включаючи еволюційний алгоритм та генотип для відображення фенотипу) дозволяти комплексувати геном (а отже, і фенотип) з часом. Варіюється від дозволу лише геномів фіксованого розміру до дозволу геномів дуже змінної довжини..

Системи класифікаторів навчання або LCS - це парадигма методів машинного навчання, заснованих на правилах, що поєднують компонент

виявлення (наприклад, як правило, генетичний алгоритм) з компонентом навчання (виконуючи або кероване навчання, або посилене навчання, або непідконтрольне навчання). Системи класифікаторів навчання прагнуть визначити набір контекстно-залежних правил, які колективно зберігають та застосовують знання окремо, щоб робити прогнози (наприклад, моделювання поведінки, класифікація, обробка даних, регресія, наближення функції, або ігрова стратегія). Цей підхід дозволяє розбити складні простори рішення на більш дрібні, прості частини [37].

Концепції, що лежать в основі систем класифікації класифікаторів, походять від спроб моделювання складних адаптивних систем, використовуючи агенти, засновані на правилах, для формування штучної когнітивної системи (тобто штучного інтелекту).

Архітектура та компоненти даної системи класифікаторів навчання можуть бути досить різними. Корисно мислити LCS як машину, що складається з декількох взаємодіючих компонентів. Компоненти можуть бути додані або видалені, або існуючі компоненти модифіковані / обмінені відповідно до потреб заданої доменної області (наприклад, алгоритмічних будівельних блоків) або зробити алгоритм достатньо гнучким для функціонування в багатьох різних проблемних областях. В результаті парадигма LCS може бути гнучко застосована до багатьох проблемних областей, які вимагають машинного навчання. Основні підрозділи серед впровадження LCS такі: (1) Мічиганська архітектура проти піттсбурзької архітектури [10], (2) підсилення навчання проти контрольованого навчання, (3) додаткове навчання порівняно з пакетним навчанням, (4) онлайн-навчання та офлайн-навчання, (5) фітнес на основі сили та фітнес на основі точності та (6) повне відображення дій у порівнянні з найкращими картами дій. Ці підрозділи не обов'язково є взаємовиключними. Наприклад, XCS, найвідоміший і найкраще вивчений алгоритм LCS, який є в Мічигані, був розроблений для навчання підкріплення, але також може виконувати контрольне навчання, застосовує додаткове навчання, яке може бути як в

Інтернеті, так і в автономному режимі, застосовується фітнес на основі точності , і прагне створити повне відображення дій.

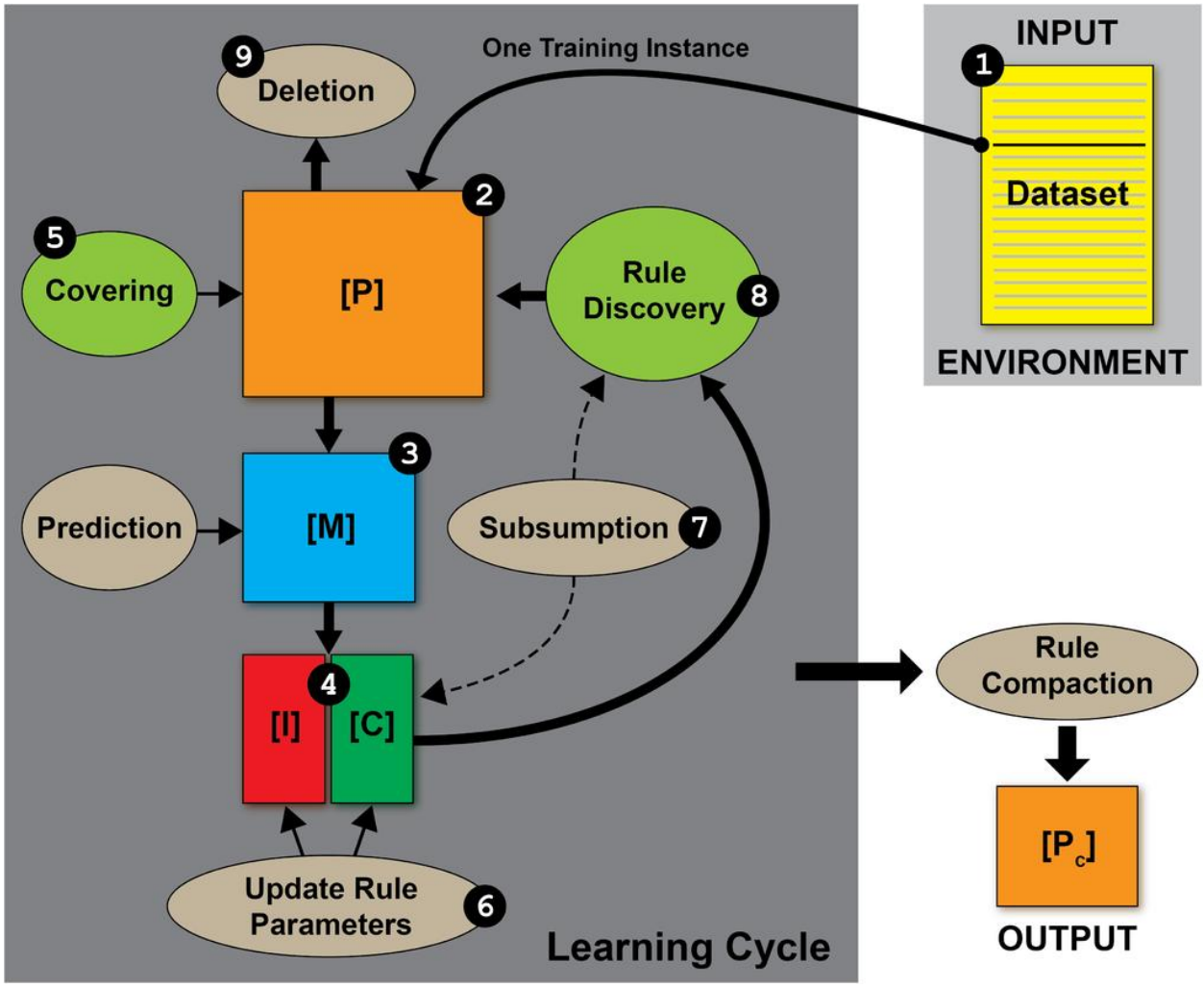

Рисунок 3.2 — Системи класифікаторів навчання.

Маючи на увазі, що LCS є парадигмою для генетичного машинного навчання, а не конкретного методу, нижче викладаються ключові елементи загального, сучасного (тобто після XCS) алгоритму LCS. Для простоти зосередимось на архітектурі в стилі Мічигана з контрольованим навчанням. Дивіться ілюстрації праворуч, де викладаються послідовні кроки, що беруть участь у цьому типі загальних LCS.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1 Розробка ігрового прототипу

Для початку створимо карту і дві команди. Карта розміром 6 * 10. У кожній команді по 4 юніта. Лучник, мечник, копейщик і вершник. Кожен юніт ходить по черзі (рисунок 4.1).

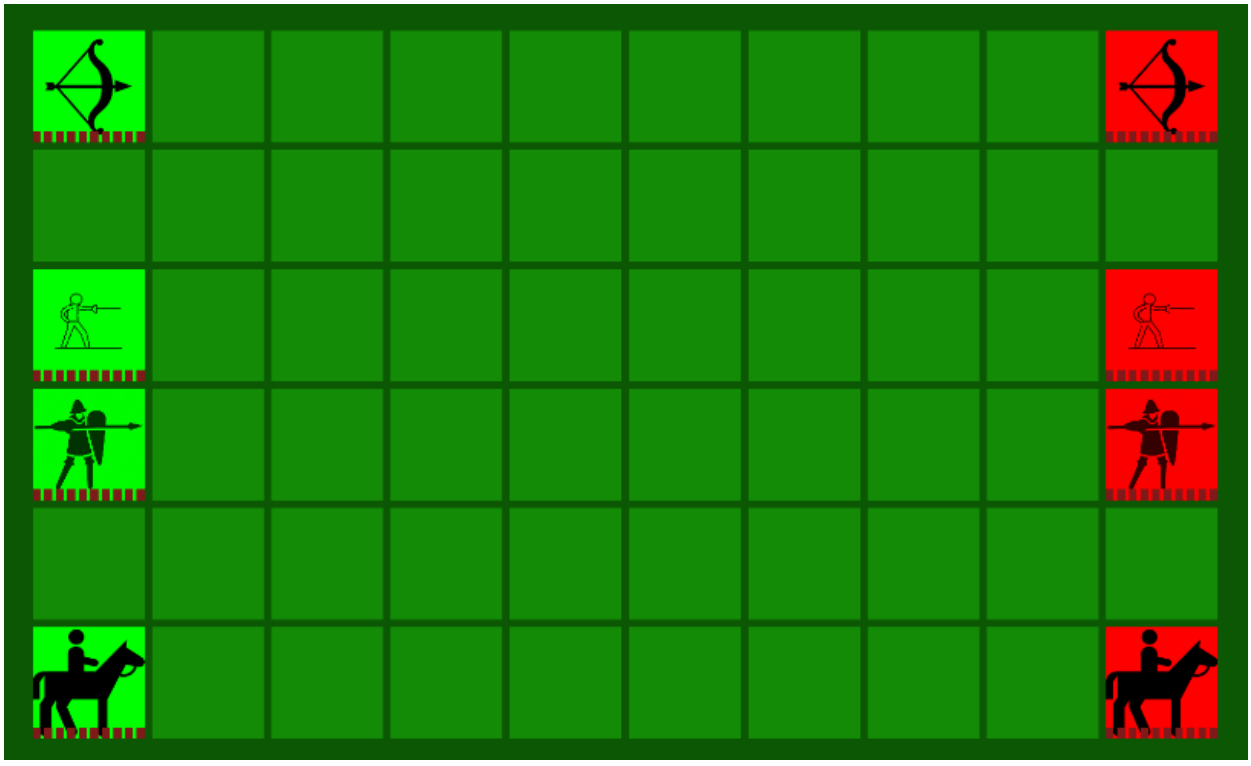

Рисунок 4.1 — Карта бою.

Для створення прототипу використовувався движок Unity. Для початку ми створили клас GodManager, который управляє тим, що відбувається на сцені [38].

На старті гри ми створюємо карту. Потім додаємо на карту юнітів. Потім запускаємо карутину, яка буде рухати наших юнітів по черзі (рисунок 4.2).

```
// Start is called before the first frame update
Event function
void Start()
{
    Map = new MapTile[10, 6];
    for (int j = 0; j < 6; j++)
    {
        for (int i = 0; i < 10; i++)
        {
            var go :GameObject = Instantiate(CellPrefab, Grid.transform);
            var cell = go.GetComponent<MapTile>();
            Map[i, j] = cell;
        }
    }

    SetUnits();
    StartCoroutine(nameof(Mooving));
     index = 0;
```

Рисунок 4.2 — Старт гри.

Для тесту ми запустили карутину кожну секунду. Зробили індексатор, який проходив за списком юнітів і викликав їх рух (рисуонк 4.3).

```
1 usage
private IEnumerator Mooving()
{
    while (true)
    {
        yield return new WaitForSeconds(1);
        _index %= Units.Count;
        var unit = Units[_index];
        UnitMove(unit, unit.Network.Execute( input: MapInputs()));

        _index++;
    }
}
```

Рисунок 4.3 — Рух юнітів.

При додаванні юнітів на карту в першу чергу необхідно створити дві команди. А потім створити юнітів, прив'язати їх до команди і розмістити на карті (рисунок 4.4).

```
1 usage
public void SetUnits()
{
    Units = new List<Unit>();
    _myTeam = new Team();
    _enemyTeam = new Team();

    Units.Add( item: new Unit( x: 0,  y: 0, UnitType.Archer, _myTeam));
    Units.Add( item: new Unit( x: 0,  y: 2, UnitType.Swordsman, _myTeam));
    Units.Add( item: new Unit( x: 0,  y: 3, UnitType.Spearman, _myTeam));
    Units.Add( item: new Unit( x: 0,  y: 5, UnitType.Rider, _myTeam));

    Units.Add( item: new Unit( x: 9,  y: 0, UnitType.Archer, _enemyTeam));
    Units.Add( item: new Unit( x: 9,  y: 2, UnitType.Swordsman, _enemyTeam));
    Units.Add( item: new Unit( x: 9,  y: 3, UnitType.Spearman, _enemyTeam));
    Units.Add( item: new Unit( x: 9,  y: 5, UnitType.Rider, _enemyTeam));
}
```

Рисунок 4.4 — Додавання юнітів.

Сам клас Unit складається з таких полів як: тип, позиція на карті, команда і здоров'я. При створенні нового юніта в конструкторі відразу ж ініціалізується нейронна мережа, параметри якої залежать від типу юніта. Вся справа в вихідних параметрах. Кожна нейронна мережа отримує на вхід дані з карти. А на виході - дія, яку повинен зробити юніт. Так як юніти різні і у них можуть бути різні дії, то кількість нейронів у вихідному шарі відрізняється. Всі юніти за один крок можуть або атакувати, або переміщатися. Вершник здатний переміщатися на 4 клітини, а атакувати на одну клітку. Мечник і копейщик переміщаються на 2 клітини і атакують на одну клітку. А лучник переміщається на 2 клітини, а атакує на одну. Так само лучник є юнітом далекого бою, а значить після атаки противник не може нанести контрудар (рисунок 4.5). Контру удар в два рази слабкіший.

```
public UnitType Type;

public int X;

public int Y;

public Team Team;

public Network Network;

public int Health;

8 usages
public Unit(int x, int y, UnitType type, Team team)
{
    X = x;
    Y = y;
    Type = type;
    Team = team;
    Network = SetNetworkByType(type);
    Health = 10;
}
```

Рисунок 4.5 — Класс юніта.

Так як на вхід кожна нейронна мережа отримує завжди повну інформацію про карту, то ми зробили для цього спеціальний метод.

Кожна клітина карти може бути: порожньою, кліткою свого юніта, кліткою противника і одним з типів юнітів, або ніяким. І того у кожної клітини 8 станів, а значить для кожного стану потрібен свій вхідний нейрон. Так як карта у нас 6 * 10 = 60, і станів 8 * 60 = 480. І того у вхідному шарі повинно бути 480 нейронів (рисунок 4.6).

```
1 usage
private List<double> MapInputs()
{
    var result = new List<double>();
    for (var i = 0; i < Map.GetLength( dimension: 0); i++)
    {
        for (var j = 0; j < Map.GetLength( dimension: 1); j++)
        {
            var mapTile = Map[i, j];
            result.Add(mapTile.MapType == MapType.None ? 1 : 0);
            result.Add(mapTile.MapType == MapType.Archer ? 1 : 0);
            result.Add(mapTile.MapType == MapType.Swordsman ? 1 : 0);
            result.Add(mapTile.MapType == MapType.Spearman ? 1 : 0);
            result.Add(mapTile.MapType == MapType.Rider ? 1 : 0);
            result.Add(mapTile.EnemyType == EnemyType.None ? 1 : 0);
            result.Add(mapTile.EnemyType == EnemyType.Mine ? 1 : 0);
            result.Add(mapTile.EnemyType == EnemyType.Enemy ? 1 : 0);
        }
    }

    return result;
}
```

Рисунок 4.6 — Інформація з карти для входного шару.

Кожен кадр ми оновлюємо карту. Спочатку ми очищаємо карту, а потім додаємо на неї юнітів. Так само ми перевіряємо, якщо у юніта немає здоров'я, то ми видаляємо його зі списку юнітів.

Ми перевіряємо до якої команди належить юніт, кількість його здоров'я, тип юніта. І передаємо всю цю інформацію до спеціального класу, що відповідає за поведінку конкретної комірки.

Осередок так само кожен кадр оновлює свій стан. Вона малює осередок в потрібний колір в залежності від того, якій команді належить юніт на цьому осередку, і чи є він там взагалі.

Так само ми відображаємо шкалу здоров'я, в залежності від здоров'я юніта на комірці. Якщо на клітинці немає юніта, то ми приховуємо шкалу здоров'я.

Ми додаємо потрібну картинку в залежності від того, який юніт розташований на комірці. Це відбувається кожен кадр, що дозволяє нам завжди бачити актуальну інформацію (рисунок 4.7, рисунок 4.8).

```
void Update()
{
    for (var j = 0; j < 6; j++)
    {
        for (var i = 0; i < 10; i++)
        {
            var mapTile = Map[i, j];
            mapTile.EnemyType = EnemyType.None;
            mapTile.MapType = MapType.None;
        }
    }

    var unitsForDelete = new List<Unit>();

    foreach (var unit in Units)
    {
        var mapTile = Map[unit.X, unit.Y];
        mapTile.EnemyType = unit.Team == _myTeam ? EnemyType.Mine : EnemyType.Enemy;

        mapTile.UnitHealth = unit.Health;

        if (unit.Health <= 0)
            unitsForDelete.Add(unit);

        switch (unit.Type)
        {
            case UnitType.Archer:
                mapTile.MapType = MapType.Archer;
                break;
            case UnitType.Swordsman:
                mapTile.MapType = MapType.Swordsman;
                break;
            case UnitType.Spearman:
                mapTile.MapType = MapType.Spearman;
                break;
            case UnitType.Rider:
                mapTile.MapType = MapType.Rider;
                break;
        }
    }

    foreach (var unit in unitsForDelete)
    {
        Units.Remove(unit);
    }
}
```

Рисунок 4.7 — Оновлення гри кожен кадр.

```
void Update()
{
    foreach (var image in Health)
        image.gameObject.SetActive(true);

    switch (MapType)
    {
        case MapType.None:
            Image.sprite = null;
            foreach (var image in Health)
                image.gameObject.SetActive(false);
            break;
        case MapType.Archer:
            Image.sprite = Archer;
            break;
        case MapType.Swordsman:
            Image.sprite = Swordsman;
            break;
        case MapType.Spearman:
            Image.sprite = Spearman;
            break;
        case MapType.Rider:
            Image.sprite = Rider;
            break;
    }

    switch (EnemyType)
    {
        case EnemyType.None:
            Image.color = new Color( r: 0.074f,  g: 0.547f , b: 0.023f);
            break;
        case EnemyType.Mine:
            Image.color = Color.green;
            break;
        case EnemyType.Enemy:
            Image.color = Color.red;
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }

    for (var index = 0; index < Health.Length; index++)
        Health[index].color = UnitHealth > index ? new Color( r: 0.5f,  g: 0.1f,  b: 0.1f) : new Color( r: 0,  g: 0,  b: 0,  a: 0);
}
```

Рисунок 4.8 — Оновлення осередку кожен кадр.

Для кожного типу юнітів є свій набір дій, які він може зробити, а значить і свій набір вихідних нейронів.

Для лучника ми можемо бути походити на 12 клітин і проатаковать 12 клітин. Разом 24.

Для вершника у нас є 40 клітин, куди можна походити і 4 клітини для атаки. Разом 44.

Для мечника і списоносця у нас є 12 клітин, куди можна походити і 4 клітини для атаки. Разом 16 (рисунок 4.9).

```
1 usage
private Network SetNetworkByType(UnitType type)
{
    Network result;
    switch (type)
    {
        case UnitType.Archer:
            result = new Network( settings: new[] {480, 120, 24}, ActivationFunctions.Sigmoid);
            break;
        case UnitType.Swordsman:
            result = new Network( settings: new[] {480, 96, 16}, ActivationFunctions.Sigmoid);
            break;
        case UnitType.Spearman:
            result = new Network( settings: new[] {480, 96, 16}, ActivationFunctions.Sigmoid);
            break;
        case UnitType.Rider:
            result = new Network( settings: new[] {480, 132, 44}, ActivationFunctions.Sigmoid);
            break;
        default:
            throw new ArgumentOutOfRangeException(nameof(type), type,  message: null);
    }

    return result;
}
```

Рисунок 4.9 — Створення нейронної мережі для кожного типу юнітів.

Так само у кожного юніта є своя сила атаки. Для цього ми додали метод Attack в клас юніта. Ми передаємо тип юніта, якого атакуємо, тому що в залежності від типу змінюється значення атаки. Стандартне значення атаки 2. Але вершник краще в атаці з мечниками і тому його атака 3, а мечник навпаки і у нього атака 1. Копейщики кращий проти вершників, а мечник кращий проти копейщиков.

Стандартний принцип «Камінь, ножиці, папір», який використовується практично у всіх стратегічних іграх (рисунок 4.10).

```
2 usages
public int Attack(UnitType type)
{
    switch (Type)
    {
        case UnitType.Archer:
            switch (type)
            {
                case UnitType.Archer:
                    return 2;
                case UnitType.Swordsman:
                    return 2;
                case UnitType.Spearman:
                    return 2;
                case UnitType.Rider:
                    return 2;
            }

            break;
        case UnitType.Swordsman:
            switch (type)
            {
                case UnitType.Archer:
                    return 2;
                case UnitType.Swordsman:
                    return 2;
                case UnitType.Spearman:
                    return 3;
                case UnitType.Rider:
                    return 1;
            }
```

Рисунок 4.10 — Перша частина методу атаки.

Продовження логіки атаки ви можете побачити на малюнку (рисунок 4.11).

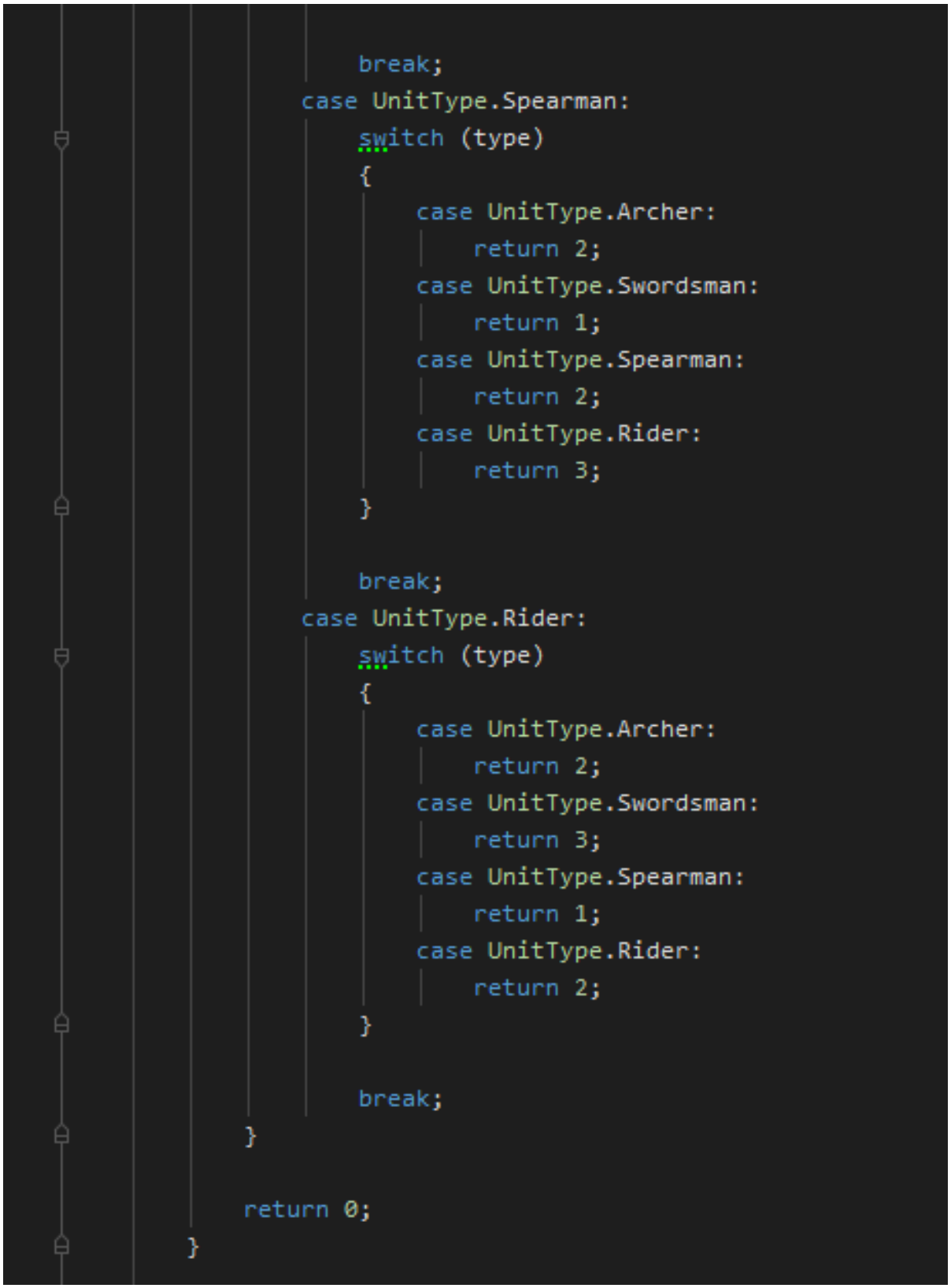

Рисунок 4.11 — Друга частина методу атаки.

Перші нейрони вихідного шару відповідають за рух, останні - за атаку. Так як кількість нейронів у вихідному шарі у кожного типу різний, то ми

створили метод IsAttack, який за індексом нейрона вихідного шару і типу юніта визначить - юніт планує атакувати чи ні (рисунок 4.12).

```
2 usages
public bool IsAttack(int key)
{
    switch (Type)
    {
        case UnitType.Archer:
            return key >= 12;
        case UnitType.Swordsman:
            return key >= 12;
        case UnitType.Spearman:
            return key >= 12;
        case UnitType.Rider:
            return key >= 40;
    }

    return false;
}
```

Рисунок 4.12 — Виявлення типу дії.

Після того, як ми отримали список вихідних нейронів. Ми сортируємо їх від більшого до меншого. І в залежності від індексу нейрона ми отримаємо позицію, куди повинен переміститися юніт, або яку повинен проатакувати. Для цього ми створили спеціальний метод GetPosition, який за індексом і

типу юніта повертає вектор, на який треба зробити зміщення щодо поточної позиції юніта (рисунок 4.13).

```
1 usage
public Vector2 GetPosition(int key)
{
    switch (Type)
    {
        case UnitType.Archer:
            return GetArcherPosition(key);
        case UnitType.Swordsman:
            return GetSwordsmanPosition(key);
        case UnitType.Spearman:
            return GetSpearmanPosition(key);
        case UnitType.Rider:
            return GetRiderPosition(key);
        default:
            throw new ArgumentOutOfRangeException();
    }

    return new Vector2();
}
```

Рисунок 4.13 — Метод визначення позиції.

4.2 Навчання ігрового бота

Для того щоб нам було легше працювати з нейронними мережами, ми представимо нейронну мережу в об'єктно-орієнтованому вигляді. Основним компонентом нейронної мережі є нейрон. Він містить в собі функцію активації та списки ваг, для вхідних значень в цей нейрон (рисунок 4.14).

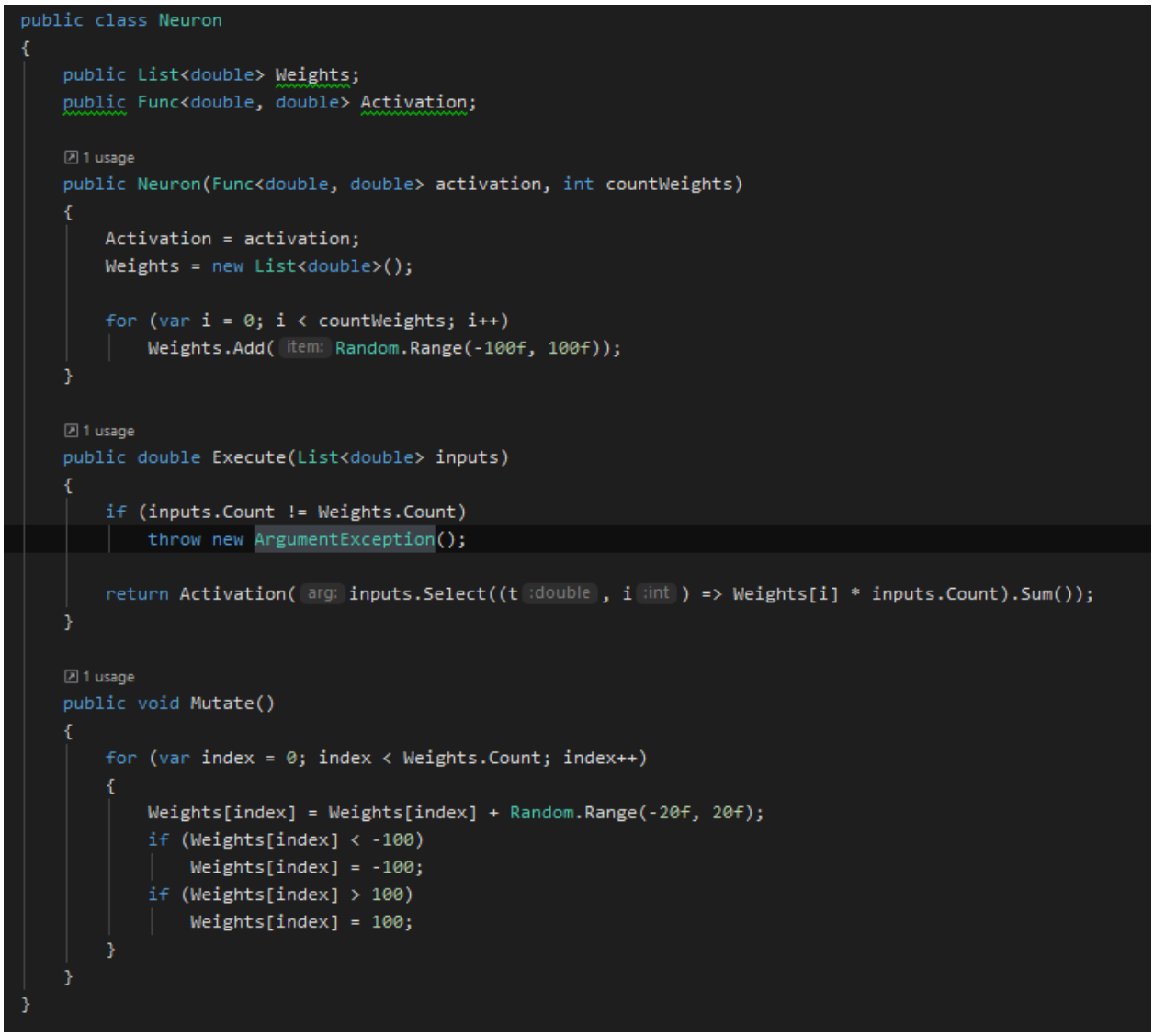

Рисунок 4.14 — Класс нейрона.

При створенні нейрона йому передається кількість нейронів з попереднього шару. Він створює список ваг цієї кількості і ініціалізує їх випадковими значеннями. При активації нейрона викликається метод Execute. Йому передається вихід попереднього шару. Ми застосовуємо ваги до входу, викликаємо функцію суматора, а потім отримане значення передаємо в функцію активації.

Так само у нейрона є метод Mutate, який додає випадкову мутацію нейронної мережі [39].

Наступним важливим об'єктом нейронної мережі є шар Layer. Він містить в собі список нейронів. Суть методу активації полягає у виклику

методу активації кожного нейрона в шарі і повернення списку з виходів кожного нейрона [40].

Метод мутації викликає метод мутації кожного нейрона шару (рисунок 4.15).

```
3 usages
public class Layer
{
    public List<Neuron> Neurons;

    1 usage
    public Layer(int currentLayer, int previousLayer, Func<double, double> activationFunction)
    {
        Neurons = new List<Neuron>();

        for (var i = 0; i < currentLayer; i++)
            Neurons.Add( item: new Neuron(activationFunction, previousLayer));
    }

    1 usage
    public List<double> Excute(List<double> input)
    {
        return Neurons.Select(item :Neuron => item.Execute(input)).ToList();
    }

    1 usage
    public void Mutate()
    {
        foreach (var neuron in Neurons)
            neuron.Mutate();
    }
}
```

Рисунок 4.15 — Класс шару.

І звичайно ж головний об'єкт нейронної мережі - сама нейронна мережа. У конструкторі нейронної мережі передаються настройки мережі. Це масив, кількість елементів в масиві - це кількість шарів, а значення - це кількість нейронів в кожному шарі. Так само до нейронної мережі передається стандартна функція активації, яка за замовчуванням буде

встановлено до кожного нейрона мережі [41]. Але завжди можна буде її змінити (рисунок 4.16).

```
public List<Layer> Layers;

private int[] _settings;
private Func<double, double> _activationFunction;

5 usages
public Network(int[] settings, Func<double, double> activationFunction)
{
    _settings = settings;
    _activationFunction = activationFunction;

    Layers = new List<Layer>();

    for (var index = 1; index < settings.Length; index++)
        Layers.Add( item: new Layer( currentLayer: settings[index],  previousLayer: settings[index-1], activationFunction));
}

1 usage
public List<double> Execute(List<double> input)
{
    return Layers.Aggregate(input, (current :List<double> , layer) => layer.Excute(current));
}

public void Mutate()
{
    foreach (var layer in Layers)
        layer.Mutate();
}

public Network Cross(Network network)
{
    var result = new Network(_settings, _activationFunction);
    for (var i = 0; i < result.Layers.Count; i++)
    {
        var layer = result.Layers[i];
        for (var j = 0; j < layer.Neurons.Count; j++)
        {
            var layerNeuron = layer.Neurons[j];
            for (var k = 0; k < layerNeuron.Weights.Count; k++)
            {
                layerNeuron.Weights[k] = (Layers[i].Neurons[j].Weights[k] + network.Layers[i].Neurons[j].Weights[k])/2;
            }
        }
    }

    return result;
```

Рисунок 4.16 — Класс нейронної мережі.

Метод активації нейронної мережі викликає метод активації на кожному шарі по черзі. Передаючи вихід з одного шару на вхід наступного. Так само в нейронної мережі ми додали метод випадкової мутації, який викликає метод мутації на кожному шарі. І так само для навчання нашої нейронної мережі генетичними алгоритмами нам потрібен метод кроссенговера [42]. Він схрещує дві нейронні мережі та на їх основі створює нащадка.

Так само ми створили статичний клас з усіма основними функціями активації, які застосовуються в нейронних мережах (рисунок 4.17).

```
4 usages
public static class ActivationFunctions
{
    public static double Line(double x)
    {
        return x;
    }

    4 usages
    public static double Sigmoid(double x)
    {
        return 1 / (1 + Math.Exp(-x));
    }

    public static double Th(double x)
    {
        return (Math.Exp(2 * x) - 1) / (Math.Exp(2 * x) + 1);
    }
}
```

Рисунок 4.17 — Основні функції активації.

Для навчання нейронної мережі ми створили 10 команд. Однією генерацією вважається турнір, де кожна команда битиметься з кожної. По

закінченню 200 кроків команди додаються бали за кількістю здоров'я, що знищено у команди противника. Якщо команда перемогла до завершення 200 ходів, то їй так само додаються бали за кожен крок, ща залишився. А команди що програла – відіймається [43].

За результатами турніру вибирається 4 команди, які набрали більшу кількість балів. Вони переходять в наступну генерацію. Далі беремо двох переможців і в наступну генерацію додаємо їх нащадка. Так само 3 рази обираємо дві команди з чотирьох переможців і відправляємо в наступну генерацію іх нащадків. А також відправляємо двох випадкових прямих спадкоємців.

Далі для кожної нейронної мережі додаємо випадкову мутацію.

Запускаємо наш алгоритм і чекаємо, поки результат не буде нас влаштовувати. Зчасом нейронні мережі зрозуміли, що атакувати більш вірний підхід, ніж втекати, але при цьому вони перестали підставлятися під удар і спочатку обстрілювали юнітів лучниками. Підходили тільки до тих юнитів, проти яких у них є бонус до атаки.

ВИСНОВКИ

В результаті виконання атестаційної роботи був створений прототип битви в стратегічних іграх. І навчена нейронна мережа для битв. За допомогою технології C # і ігрового двигуна Unity був створений прототип. А за допомогою генетичних алгоритмів була навчена нейронна мережа для кожного з типів юнітів.

Для реалізації всього необхідного функціоналу була проаналізовано предметна область. Виявлено сучасні проблеми в стратегічних іграх. Ми з'ясували які аспекти сучасного штучного інтелекту в стратегіях не дозволяють отримати максимальне задоволення від гри.

Були досліджені сучасні способи навчання нейронних мереж в іграх. Ми розглянули всі види еволюційних алгоритмів. А також вивчили застосування генетичних алгоритмів на практиці. Ми створили перше покоління, провели турнір, визначили кращих і використовуючи генетичні алгоритми сформували наступне покоління. Перевіривши безліч генерацій ми отримали ваги для нейронної мережі кожного з типів юнітів, а також навчили їх працювати в команді.

Даний штучний інтелект дійсно прагне перемогти в грі, а не тільки розважити гравця. Можливо на даний момент ми реалізували всього лише частину логіки, а саме бій. Але бій - це одна з основних складових стратегічних ігор. І використовуючи даний підхід ми зможемо навчити штучний інтелект прагнути до перемоги в стратегічних іграх. Адже будь-яка стратегічна гра - це лише послідовне прийняття рішень.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Литвин В.В., Угрин Д.І. Методика вирішення завдань пошуку оптимальних туристичних маршрутів алгоритмами наслідування мурашиної колонії // Вестник НТУ "ХПИ". – 2016. – № 21 (1193). – С. 48 – 60.
2. Shao W., Salim F.D., Gu T., Dinh N.-T., Chan J. Traveling Officer Problem: Managing Car Parking Violations Efficiently Using Sensor Data // IEEE Internet of Things Journal. – April 2018 – Vol. 5. – Issue 2. – P. 802 – 810.
3. Meneses S., Cueva R., Tupia M., Guanira M. A genetic algorithm to solve 3D traveling salesman problem with initial population based on a GRASP algorithm // Journal of Computational Methods in Sciences and Engineering. – 2017 – vol. 17. – no. S1. – P. S1-S10.
4. Красников С.О. Алгоритм налаштування параметрів алгоритму імітаційного відпалу для розв’язання задачі комівояжера // Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки. – 2018. – Том 29. - № 5. – С. 149-153.
5. Красников С.О. Алгоритм налаштування параметрів алгоритмів стохастичного локального пошуку для розв’язання задачі комівояжера // Збірник статей за XL міжнародної науково-практичної конференції 2 частина: «Розвиток науки в XXI столітті» від 13.10.2018. – Науково-інформаційний центр «Знання». – 2018. – частина 1. – С.45-54.
6. Пападимитриу Х., Стайглиц К. Комбинаторная оптимизация: Алгоритмы и сложность // Видавництво «Мир». – 1985 . – 510 с.
7. Емец О.А., Барболина Т.Н. О свойствах линейной безусловной задачи комбинаторной оптимизации на размещениях с вероятностной неопределенностью // Кибернетика и системный анализ. – 2016. – Том 56. – №2. – С. 125-136.
8. Boland N., Hewitt M., Minh Vu D., Savelsbergh M. Solving the Traveling Salesman Problem with Time Windows through Dynamically Generated

Time-Expanded Networks // Integration of AI and OR Techniques in Constraint Programming. – 2017. – volume 10335. – P. 254-262.

9. Гуляницький Л.Ф., Мулеса О.Ю. Прикладні методи комбінаторної оптимізації // Київський університет ВПІ. – 2016. – 146 с.

10. Lawler E. L., Lenstra J. K., Rinnooy Kan A. H. G., Shmoys D. B. (1985). The Travelling Salesman Problem // John Wiley & Sons. – Chichester. – UK. – 1985. – 463 c.

11. Меламед И.И., Сергеев С.И., Сигал И.Х. Задача коммивояжера. Вопросы теории // Автоматика и телемеханика. – № 9. – 1989. – С. 3-33.

12. Сесекин А.Н., Ченцов А.А., Ченцов А.Г. Обобщенная задача курьера с функцией затрат, зависящей от списка заданий // Известия российской академии наук. теория и системы управления. – Российская академия наук. – № 2. – 2010. – С. 68-77.

13. Chisman J.A. The clustered traveling salesman problem // Computers & Operations Research. – Volume 2. – Issue 2. – September 1975. – P. 115-119.

14. Bektas T. The multiple traveling salesman problem: an overview of formulations and solution procedures // Omega. – Elsevier. – № 34. – 2006. – P. 209-219.

15. Gutin G., Punnen A. P. The traveling salesman problem and its variations // Springer. – USA. – 2006. – 830 p.

16. Current J.R., Schilling D.A. The covering salesman problem // Transportation science. – Volume 23. – № 3. – 1989. – P. 151-229.

17. Nygard K.E., Yang C.H. Genetic algorithm for the traveling salesman problem with time windows // Computer Science and Operations Research: New Developments in their Interfaces. – Elsevier. – 2014. – P. 411-423

18. Mauricio Resende G.C., Ribeiro Celso C. A short tour of combinatorial optimization and computational complexity // Optimization by GRASP. – 2016. – P. 13-39.

19. Сергиенко И.В., Гуляницкий Л.Ф., Сиренко С.И. Классификация прикладных методов комбинаторной оптимизации // Кибернетика и системный анализ. – 2009. – № 2. – С. 71-83.

20. Сергиенко И. В. Математические модели и методы решения задач дискретной оптимизации // Киев. – Наукова думка. – 1988. – 472 с.

21. Михалевич В.С., Шор Н.З., Галустова Л.А. Вычислительные методы выбора оптимальных проектных решений // Наукова думка. – Киев. – 1977. – 178 с.

22. Korte B., Vygen J. Combinatorial optimations: Theory and algorithms // Algorithms and Combinatorics. – Springer. – 2011. – Volume 21. – 659 p.

23. Ebert T., Fischer T., Belz J., Heinz T. O., Kampmann G., Nelles O. Extended Deterministic Local Search Algorithm for Maximin Latin Hypercube Designs // 2015 IEEE Symposium Series on Computational Intelligence. – 2015. – P. 375-382.

24. Kirkpatrick S., Gelatt C. D., Vecchi M. P. Optimization by Simulated Annealing // Science. – Volume 220. - № 4598. – 1983. – P.671-680.

25. Kheiri A., Özcan E., Parkes A. J. A stochastic local search algorithm with adaptive acceptance for high-school timetabling // Annals of Operations Research. – Springer. – Volume 239(1). – 2016. – P.135-151.

26. Geng X., Chen Zh., Yang W., Shi D., Zhao K. Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search // Applied Soft Computing. – Elsevier. – Volume 11. – 2011. – P.3680-3689.

27. Holland J. H. Genetic algorithms and the optimal allocation of trials // SIAM Journal on Computing. – Volume 2. - №2. – 1973. – P.88-105

28. Yu W., Li B., Jia H., Zhang M., Wang D. Application of multi-objective genetic algorithm to optimize energy efficiency and thermal comfort in building design // Energy and Buildings. – Volume 88. – 2015. – P.135-143.

29. Kadri R.L., Boctor F.F. An efficient genetic algorithm to solve the resource-constrained project scheduling problem with transfer times: The single

mode case // European Journal of Operational Research. – Volume 265. – Issue 2. – 2018. – P.454-462.

30. Merz P., Freisleben B. Memetic Algorithms for the Traveling Salesman Problem // Complex Systems. – 2001. - № 13. – P. 297-345.

31. Cattaruzza D., Absi N., Feillet D., Vidal T. A memetic algorithm for the multi trip vehicle routing problem // European Journal of Operational Research. – Volume 236. – Issue 3. – 2014. – P.833-848.

32. Colorni A., Dorigo M., Maniezzo V. Distributed optimization by ant colonies // Proceeding of ECAL91. – Elsevier Publishing. – P.134-142.

33. Liao T., Stützle T., Oca M. A. M. de, Dorigo M. A unified ant colony optimization algorithm for continuous optimization // European Journal of Operational Research. – Volume 234. – 2014. – P.597-609.

34. Wang Z., Xing H., Li T., Yang Y., Qu R., Pan Y. A modified ant colony optimization algorithm for network coding resource minimization // IEEE Transactions on Evolutionary Computation. – Volume 20. – Issue 3. – 2016. – P.325-342.

35. Гуляницький Л.Ф., Мулеса О.Ю. До класифікації метаевристик // XXI Всеукраїнська наукова конференція «Сучасні проблеми прикладної математики та інформатики». – Львів. – 2015. – С.139-142.

36. Naderi B., Ruiz R. A scatter search algorithm for the distributed permutation flowshop scheduling problem // European Journal of Operational Research. – Volume 239. – 2014. – P.323-334.

37. Sh. Chunxin, Zh. Xiaoxia, Ch. Hongyang, Y. Jiao,Wangpeng, Weiyu A Hybrid Scatter Search Algorithm for QoS Multicast Routing Problem // Chinese Control And Decision Conference. – 2018. – P.4875-4878.

38. Mahi M., Baykan Ö. K., Kodaz H. A new hybrid method based on particle swarm optimization, ant colony optimization and 3-opt algorithms for traveling salesman problem // Applied Soft Computing. – Elseveir. – May 2015. – Volume 30. – P 484-490.

39. Gutjahr W. J. Stochastic Search in Metaheuristics // Handbook of Metaheuristics. – Second Edition. – 2010. – P.573-597.

40. Mu C.H., Xie J., Liu Y., Chen F., Liu Y., Jiao L.C. Memetic algorithm with simulated annealing strategy and tightness greedy optimization for community detection in networks // Applied Soft Computing. – Volume 34. – 2015. – P.485-501.

41. Dorigo M., Blum C. Ant colony optimization theory: A survey // Theoretical computer science. – Volume 344. – 2005. – P.243-278.

42. Семенов С.С., Педан А.В., Воловиков В.С., Климов И.С. Анализ трудоемкости различных алгоритмических подходов для решения задачи коммивояжера // Системы управления, связи и безопасности. – №1. – 2017. – С. 116-131.

43. Mills K.L., Filliben J.J., Haines A.L. Determining Relative Importance and Effective Settings for Genetic Algorithm Control Parameters // Evolutionary Computation. – 2015. – MIT Press. – Volume 23. – Issue 2. – P. 309-342.