

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка

до дипломного проєкту (роботи)

магістра

(ступінь вищої освіти)

на тему ІНТЕЛЕКТУАЛЬНА СИСТЕМА ОПТИМІЗАЦІЇ
РОЗВ'ЯЗАННЯ ІГРОВИХ ЗАДАЧ

Виконав: студент 2 курсу, групи КНТ-514м
спеціальності _____

123 Комп'ютерна інженерія

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Комп'ютерні системи та мережі

ТАРОВИК М.Є.

(ПРИЗВИЩЕ та ініціали)

Керівник КУДЕРМЕТОВ Р.К.

(ПРИЗВИЩЕ та ініціали)

Рецензент ХАРИТОНОВ О.Б.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій
Кафедра «Комп'ютерні системи та мережі»
Ступінь вищої освіти магістерський
Спеціальність 123 Комп'ютерна інженерія
(код і найменування)
Освітня програма (спеціалізація) Комп'ютерні системи та мережі
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ
Зав. кафедри Кудерметов Р.К.
«15» жовтня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

ТАРОВИКА Микити Євгеновича

(ПРИЗВИЩЕ, ім'я, по батькові)

- Тема проєкту (роботи) Інтелектуальна система оптимізації розв'язання ігрових задач
керівник проєкту (роботи) кан. техн. наук, доцент, КУДЕРМЕТОВ Равіль Камілович
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)
затверджені наказом вищого навчального закладу від «30» жовтня 2025 року № 491
- Строк подання студентом проєкту (роботи) 23 грудня 2025 року
- Вихідні дані до проєкту (роботи) наявні методи та алгоритми створення та реалізації нейронної мережі для виконання простих ігрових задач швидше людини
- Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
 - Аналіз предметної області;
 - Методи штучного інтелекту для розробки гри;
 - Реалізація гри;
- Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
 - Тестування гри;
 - Таблиця результатів кожного сімейства;
 - Таблиця інформації про навчання НМ;
 - Таблиця роботи програми;

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	КУДЕРМЕТОВ Р.К., доцент		
нормоконтроль	ЩЕРБАК Н.В., ст. викл.		

7. Дата видачі завдання 01.10.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз наявних методів та алгоритмів пошуку об'єктів, перегляд отриманих результатів, виділення недоліків та переваг	15.10.2025 р.	
2	Розробка структури системи	30.10.2025 р.	
3	Розробка алгоритму роботи системи	10.11.2025 р.	
4	Реалізація системи уточненого розпізнавання	25.11.2025 р.	
5	Дослідження системи уточненого розпізнавання	30.11.2025 р.	
6	Оформлення отриманих результатів у ПЗ	05.12.2025 р.	
7	Оформлення графічного матеріалу	10.12.2025 р.	
8	Оформлення допоміжного матеріалу	15.12.2025 р.	

Студент Микита Таровик
(підпис) (ім'я, ПРИЗВИЩЕ)

Керівник проєкту (роботи) Равіль КУДЕРМЕТОВ
(підпис) (ім'я, ПРИЗВИЩЕ)

РЕФЕРАТ

ПЗ: 63 стр., 5 рис., 14 джерел.

PYTHON, ГЕНЕТИЧНИЙ АЛГОРИТМ, КОМП'ЮТЕРНА ГРА, НЕЙРОННА МЕРЕЖА, ОПТИМІЗАЦІЯ ПРОЦЕСІВ, ШТУЧНИЙ ІНТЕЛЕКТ

Об'єкт розробки – інтелектуальна система оптимізації розв'язання ігрових задач.

Метою роботи є розробка ефективної системи навчання гри «Змійка» за допомогою нейронних мереж, оптимізованих генетичними алгоритмами та використанням пам'яті.

Проект складається з трьох основних розділів, кожен з яких розкриває ключові аспекти дослідження.

У першому розділі проведено аналіз теоретичних основ, що стосуються розробки автономних інтелектуальних агентів.

Другий розділ присвячений проектуванню архітектури та алгоритмів створеної системи. Центральне місце займає опис власної архітектури MemoryPerceptron - вдосконаленого перцептрона з механізмом короткострокової пам'яті, що дозволяє агенту враховувати послідовність станів.

У третьому розділі наведено результати практичної реалізації, експериментальних досліджень та їх аналіз. Детально описано технологічний стек (Python, NumPy, Pygame), структуру проектних файлів та особливості коду. Представлено планомірну методологію експериментів, спрямованих на дослідження впливу ключових гіперпараметрів: ймовірності мутації, розміру популяції, конфігурації нейромережі та наявності механізму пам'яті.

ABSTRACT

EN: 63 pages, 5 figures, 14 sources.

PYTHON, ARTIFICIAL INTELLIGENCE, COMPUTER GAME, GENETIC ALGORITHM, NEURAL NETWORK, PROCESS OPTIMIZATION

The object of development is an intelligent optimization system for solving game problems.

The aim of the work is to develop an effective system for learning the game "Snake" using neural networks optimized by genetic algorithms and using memory.

The project consists of three main sections, each of which reveals key aspects of the research.

The first section analyzes the theoretical foundations related to the development of autonomous intelligent agents. The main attention is paid to two key components: neural networks as a universal tool for decision-making in complex environments and genetic algorithms as a method of global optimization without the need for a gradient. The principles of operation of the multilayer perceptron and its architectural variations are considered, as well as specialized mechanisms for working with temporal dependencies, such as recurrent networks and contextual memory mechanisms, which are critically important for dynamic tasks. The role of game environments as ideal testbeds for AI algorithms due to their controllability, speed, and clarity is separately investigated. Analysis of the problems of training an agent for the game "Snake" revealed key difficulties, such as the lack of a ready-made training sample and the need for long-term planning, which justified the choice of a hybrid approach that combines the advantages of the two methodologies.

ЗМІСТ

Перелік умовних позначень	8
Вступ.....	9
1 Аналіз предметної області	10
1.1 Ігрові середовища як тестові стенди для алгоритмів штучного інтелекту	11
1.2 Нейронні мережі в задачах керування та прийняття рішень	13
1.3. Генетичні алгоритми як метод оптимізації без градієнта	14
1.4 Проблеми навчання агента для гри	15
1.5 Аналіз подібних робіт та існуючих рішень	17
2 Методи штучного інтелекту для розробки гри	18
2.1 Загальна архітектура програмного комплексу	20
2.2 Модель ігрового середовища та система сенсорів	22
2.3 Архітектура нейронної мережі з механізмом пам'яті MemoryPerceptron	23
2.4 Представлення індивіда та ініціалізація популяції	25
2.5 Функція пристосованості	26
2.6 Відбір	27
2.7 Схрещування та мутація	28
2.8 Стратегія сімейств та епох	29
2.9 Система візуалізації та моніторингу процесу навчання	30
2.10 Візуалізація ігрового процесу	31
2.11 Інформаційні панелі та таблиці	32
2.12 Графік прогресу навчання	32
2.13 Система збереження та відтворення результатів	34
3 Реалізація гри	35
3.1 Реалізація програмного комплексу та технічні деталі	36
3.2 Залежності та середовище виконання	37
3.3 Структура проектних файлів	38
3.4 Обробка подій та керування симуляцією	38

3.5 Оптимізація продуктивності	39
3.6 План та методологія експериментальних досліджень	39
3.7 Контрольна конфігурація	40
3.8 Змінні параметри для дослідження	41
3.9 Метрики оцінки	42
3.10 Проведення експериментів та отримання даних	42
3.11 Результати базового експерименту	43
3.12 Експеримент 1: Вплив ймовірності мутації	44
3.13 Експеримент 2: Вплив розміру популяції	44
3.14 Експеримент 3: Вплив механізму пам'яті	45
3.15 Аналіз отриманих результатів та обговорення	46
3.16 Обмеження та проблеми, що виявились	46
3.17 Якісний аналіз поведінки навчених агентів	47
3.18 Порівняння з альтернативними підходами	47
Висновки	49
Перелік джерел посилання	51
Додаток А	52
Додаток Б	61

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

НМ	-	нейронна мережа;
БП	-	багатошаровий перцептрон;
Мутація	-	процес випадкової зміни значень генів в хромосомах індивідуума;
ГА	-	генетичний алгоритм;
Селекція	-	процес відбору найкращих індивідуумів з популяції
Міні-батч	-	метод розбиття навчального набору даних на невеликі підмножини
Агент	-	програма або алгоритм, який взаємодіє з ігровим середовищем, приймаючи рішення на основі вхідних даних

ВСТУП

Сучасний розвиток інформаційних технологій призвів до активного використання методів штучного інтелекту у різних прикладних галузях. Особливе місце серед них займають задачі моделювання поведінки автономних агентів, здатних приймати рішення в умовах обмеженої інформації та змінного середовища. Подібні задачі виникають у робототехніці, системах керування, ігровій індустрії та навчальних симуляціях.

Ігрові середовища є зручним інструментом для дослідження інтелектуальних алгоритмів, оскільки дозволяють створити контрольовані умови експерименту, чітко визначити правила взаємодії та отримувати кількісні результати роботи агентів. Однією з таких класичних задач є гра «Змійка», яка, незважаючи на простоту правил, вимагає від агента здатності до планування, адаптації та прогнозування наслідків власних дій.

Основною складністю задачі є те, що агент не має повної інформації про майбутній стан гри та повинен приймати рішення в реальному часі. Це робить використання жорстко заданих алгоритмів малоефективним і обґрунтовує застосування методів машинного навчання та еволюційних алгоритмів.

Метою даної роботи є розробка ефективної системи навчання гри «Змійка» за допомогою нейронних мереж, оптимізованих генетичними алгоритмами та використання пам'яті. Програма має реалізувати інтелектуального агента для гри «Змійка» на основі нейронної мережі та генетичного алгоритму навчання. У рамках роботи досліджується вплив архітектури мережі, механізмів пам'яті та параметрів еволюційного відбору на поведінку агента.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Штучний інтелект як сучасний науковий напрям сягає своїм корінням у першу половину минулого століття. Одна з найперших концептуальних моделей, яка намагалася формалізувати принципи мислення, з'явилася 1943 року завдяки роботам Уоррена Маккалоха та Волтера Піттса [1]. Вони запропонували абстрактну схему штучного нейрона - базового елемента, здатного виконувати прості логічні операції. Ця модель стала теоретичною основою для всіх подальших розробок у галузі штучних нейронних мереж.

Знаковим проривом стало створення Френком Розенблаттом у 1958 році перцептрона - першої практичної нейромережі, яку можна було навчати [1]. Його архітектура обмежувалася одним шаром, що дозволяло вирішувати лише задачі з лінійною роздільністю. Нездатність перцептрона оперувати з нелінійними залежностями призвела до певного занепаду інтересу до цього напрямку в 1970-х роках, відомого як «зима штучного інтелекту».

Відродження та потужний поштовх у розвитку нейромереж відбулися в 1980-х. Важливу роль у цьому зіграли дослідження Джона Хопфілда, який запропонував модель мережі зі зворотними зв'язками, та групи на чолі з Джеффри Гінтоном, що розробила універсальний алгоритм зворотного поширення помилки [2]. Цей алгоритм нарешті відкрив шлях до ефективного навчання мереж із кількома прихованими шарами, давши старт новій епосі в дослідженнях ШІ.

На сьогодні нейронні мережі трансформувалися з академічної цікавості у ключову технологію, що лежить в основі систем розпізнавання зображень і мови, медичних діагностичних комплексів, автономного транспорту та безлічі інших прикладних розробок. Глибоке навчання, засноване на складних багатошарових архітектурах, дозволяє аналізувати колосальні масиви даних, виявляючи в них складні, неочевидні закономірності.

Паралельно з еволюцією штучного інтелекту розвивалася і культова гра «Змійка».

Її прообраз вийшов у 1976 році під назвою Blockade як аркадна гра від студії Gremlin. Механіка була простою та геніальною: гравець керував зростаючою лінією на екрані, намагаючись збирати предмети та уникати зіткнень.

Справжню світову популярність гра здобула через два десятиліття, коли була вбудована як стандартна заставка в мобільні телефони Nokia. Саме ця версія перетворила «Змійку» на глобальний феномен, знайомий мільярдам людей, і закріпила її статус відеоігрової класики. З того часу гра неодноразово перевидавалася та інтерпретувалася на різних платформах, не втрачаючи своєї актуальності.

У сучасних умовах дві ці лінії - розвиток нейронних мереж та культова гра - тісно переплітаються в інноваційних проєктах. Нейромережі використовуються як для створення адаптивних ігрових світів, що підлаштовуються під стиль користувача, так і для розробки автономних агентів, здатних самостійно освоювати ігрову механіку. Таким чином, застосування алгоритмів штучного інтелекту для навчання агента гри «Змійка» є не лише наочним навчальним прикладом, але й практичною демонстрацією потенціалу цих технологій для створення нового покоління інтерактивних інтелектуальних систем.

1.1 Ігрові середовища як тестові стенди для алгоритмів штучного інтелекту

Розвиток штучного інтелекту нерозривно пов'язаний із використанням ігор як інструменту дослідження. Від легендарного тесту Тюрінга, що пропонував імітувати людську розмову, до сучасних складних стратегій, таких як Dota 2 або StarCraft II, ігри завжди були вимірювальним інструментом для оцінки розумності машини.

Переваги ігрових середовищ для досліджень ШІ [14] заключаються у контрольованості та детермінованості. Правила гри чітко визначені, а стан

системи може бути повністю описаний. Це дозволяє точно відтворювати умови експерименту та ізолювати вплив зовнішніх факторів.

Масштабованість та швидкість. Ігрові симуляції можуть працювати в тисячі разів швидше за реальний час. За одну годину можна провести мільйони ігрових партій, що недосяжно для експериментів у фізичному світі.

Безпека. Помилки агента в грі не мають реальних негативних наслідків, що дає змогу безстрашно тестувати радикальні та дослідницькі стратегії.

Чіткі метрики успіху. Перемога/поразка, набрані очки, час виживання - усі ці показники є кількісними та об'єктивними критеріями для оцінки якості роботи алгоритму.

Поступове ускладнення. Складність гри можна регулювати (наприклад, збільшуючи розмір поля або швидкість), створюючи плавний градієнт складності для навчання агента.

Гра «Змійка» є яскравим прикладом такого середовища. Її правила надзвичайно прості: агент керує змійкою, яка рухається по дискретній сітці, збирає з'являючуся їжу (яблука), від чого збільшується в довжину, і програє при зіткненні з межами поля або з власним тілом. Незважаючи на цю простоту, гра породжує ряд складних завдань, наприклад планування шляху - знаходження маршруту до яблука в умовах, що постійно змінюються через зростання власного тіла.

Уникнення пасток - агент повинен передбачати, чи не опиниться він в глухому куті через декілька ходів.

Компроміс між ризиком та винагородою - іноді найкоротший шлях до їжі несе загрозу, тому потрібно обирати безпечніший, але довший маршрут.

Таким чином, «Змійка» є відмінним мікросвітом для дослідження алгоритмів, що поєднують реактивне керування (миттєва реакція на перешкоди) з елементарним стратегічним плануванням.

1.2 Нейронні мережі в задачах керування та прийняття рішень

Нейронна мережа (НМ) - це математична модель, натхненна будовою біологічних нейронних мереж. Її основна сила полягає в здатності навчатися складним нелінійним залежностям між вхідними та вихідними даними без необхідності їх явного програмування.

Багатошаровий перцептрон (Multilayer Perceptron, MLP) - це найпоширеніший тип НМ, що складається з вхідного, одного або кількох прихованих шарів та вихідного шару. Кожен шар складається з нейронів, які приймають сигнал від нейронів попереднього шару, зважують їх, підсумовують та пропускають через нелінійну функцію активації (наприклад, ReLU – Rectified Linear Unit). Саме завдяки прихованим шарам та нелінійності MLP може моделювати довільні складні функції.

У контексті гри «Змійка» нейронна мережа виконує функцію політики (policy). Вона отримує на вхід вектор, що описує стан середовища, і видає на виході розподіл ймовірностей або оцінку переваги для кожної з можливих дій (рух вгору, вниз, вліво, вправо). Агент вибирає дію з найвищою оцінкою.

Ключова проблема в архітектурі для динамічних ігор: класичний MLP обробляє кожен вхідний вектор незалежно. Для гри, де важлива послідовність подій (наприклад, щойно зібране яблуко або циклічний рух для уникнення), такої амнезії може бути недостатньо. Тому в сучасних підходах часто використовують мережі з пам'яттю, такі як рекурентні нейронні мережі (RNN) або їх розвинені версії (LSTM, GRU). Альтернативним, більш простим підходом є введення механізму контексту шляхом додавання до вхідного вектора спеціального стану, що оновлюється з кожним кроком і несе інформацію про недавні події. Саме такий підхід було обрано для реалізації в даній роботі через його простоту та ефективність для задач із порівняно короткою часовою залежністю.

1.3 Генетичні алгоритми як метод оптимізації без градієнта

Навчання нейронної мережі в задачах з підкріпленням часто ускладнене тим, що функція винагороди (наприклад, кількість очок в грі) є розрідженою, шумною та недиференційованою за параметрами мережі. Це робить неможливим безпосереднє застосування класичних методів на основі градієнтного спуску, таких як алгоритм зворотного поширення помилки.

Генетичні алгоритми (ГА) належать до класу еволюційних алгоритмів і натхненні принципами природної еволюції: відбором, спадковістю та мутацією. Вони є потужним інструментом для глобальної оптимізації в умовах відсутності інформації про градієнт цільової функції [14].

Основний цикл генетичного алгоритму починається з ініціалізації популяції. Створюється початковий набір індивідів (хромосом). У нашому випадку кожен індивід - це повний набір ваг однієї нейронної мережі. Ініціалізація зазвичай випадкова.

Оцінка пристосованості кожен індивід «програє» одну або кілька ігрових сесій. Його пристосованість обчислюється на основі результатів гри (очки, тривалість). Це найбільш обчислювально-витратний етап.

Відбір індивіди з вищою пристосованістю мають більший шанс бути обраними як «батьки» для наступного покоління. Це імітує природний відбор, де сильніші особини отримують більше шансів на розмноження.

Схрещуванні гени (у нашому випадку - ваги) двох батьківських індивідів об'єднуються для створення одного або двох нащадків. Це основний механізм дослідження простору рішень і поєднання успішних ознак.

Мутація з невеликою ймовірністю у генах нащадків вносяться випадкові зміни. Це запобігає передчасній збіжності алгоритму до локального оптимуму та підтримує генетичну різноманітність популяції.

Формування нового покоління. Нащадки замінюють старе покоління (повністю або частково), і цикл повторюється з кроку 2.

Перевага ГА для навчання гри «Змійка» полягає в тому, що алгоритму не потрібно «розуміти», чому один набір ваг кращий за інший. Він лише постійно відбирає та комбінує ті варіанти, які випадково призвели до кращого результату. За сотні та тисячі поколінь такий поступовий, ітеративний пошук може призвести до появи справді ефективних стратегій.

1.4 Проблеми навчання агента для гри

Застосування генетичних алгоритмів для навчання ігрових агентів, попри свою ефективність, пов'язане з низкою викликів та технічних обмежень, які потребують уваги дослідника.

Складність пошуку в широкому просторі параметрів є одним з основних викликів. Кожен агент описується великою кількістю ваг нейронної мережі, що створює експоненційно великий простір можливих рішень. Ефективне дослідження такого простору вимагає або значної обчислювальної потужності, або вдалого обмеження самої задачі. У даній роботі це обмеження було враховане через вибір відносно простої гри «Змійка» з чітко визначеним полем та дискретними діями, що дозволило зконцентруватися на відпрацюванні механізмів навчання, а не на подоланні непомірної складності середовища.

Високі обчислювальні витрати безпосередньо впливають з попередньої проблеми. Оцінка пристосованості кожного індивіда вимагає повної ігрової сесії, яка може тривати сотні ходів. Оскільки популяція налічує сотні або тисячі особин, а процес еволюції повторюється протягом багатьох епох, сумарний обсяг симуляцій стає колосальним. Це накладає практичні обмеження на глибину досліджень та необхідність оптимізації коду, особливо критичних ділянок, таких як цикл гри та розрахунки нейронної мережі.

Ризик втрати корисних адаптацій є внутрішньою властивістю стохастичних операцій генетичного алгоритму. Процеси схрещування та мутації, хоча і

спрямовані на покращення популяції, носять ймовірнісний характер. Внаслідок цього успішна комбінація параметрів, що виникла в одному поколінні, може бути «розібрана» або переписана при створенні нащадків. Для пом'якшення цього ефекту в практичній реалізації часто застосовують стратегію елітизму, яка гарантовано зберігає найкращі рішення, а також ретельно підбирають ймовірності операцій.

Чутливість до гіперпараметрів ускладнює процес налаштування. Ефективність алгоритму суттєво залежить від таких налаштувань, як розмір популяції, інтенсивність мутацій та схеми відбору. Неоптимальний вибір цих значень може призвести до передчасної збіжності на посередніх результатах або, навпаки, до повного відсутності прогресу. Пошук балансу часто є ітеративним процесом, що базується на емпіричних даних, отриманих у попередніх експериментах, що підкреслює важливість системи моніторингу та збереження проміжних результатів.

Незважаючи на перелічені труднощі, генетичні алгоритми залишаються потужним інструментом саме для завдань, де простір параметрів великий, а цільова функція недиференційована або має значний шум, як це часто буває в ігрових середовищах. Їхня сила полягає в здатності до паралельного дослідження широкого фронту рішень і поступового накопичення покращень через механізми, натхненні біологічною еволюцією. Ключем до успіху є не усунення цих проблем повністю, а їхнє усвідомлення та адаптація методології до конкретних умов задачі, що включає обґрунтований вибір моделі середовища, ретельне проектування архітектури агента та проведення планомірних експериментів для калібрування алгоритму.

1.5 Аналіз подібних робіт та існуючих рішень

Задача навчання агента для гри «Змійка» є досить популярним бенчмарком в академічних колах та серед ентузіастів машинного навчання. Існує кілька типових підходів до її вирішення:

Алгоритми пошуку шляху. Ці детерміновані алгоритми можуть знаходити оптимальний шлях до яблука, але вони не вміють адаптуватися до довгострокових наслідків (зростання тіла) і можуть легко потрапити в пастку.

Глибоке зміцнене навчання. Це сучасний стандарт для багатьох ігрових завдань. Агент навчає оцінку якості для кожної пари «стан-дія». Недоліком є висока обчислювальна складність, необхідність у складному представленні стану (часто як зображення екрана) та нестабільність навчання, що вимагає ретельного підбору гіперпараметрів.

Чисті генетичні алгоритми. Часто використовуються для прямої оптимізації ваг невеликої нейронної мережі. Цей підхід, обраний у даній роботі, є більш простим для реалізації та розуміння, стійким до шуму в функції винагороди, але може вимагати більшої кількості ігрових симуляцій для збіжності.

Вибір гібридного підходу (нейромережа + ГА) у даній роботі обґрунтований балансом між складністю реалізації та потенційною ефективністю. Це дозволяє сконцентруватися на дослідженні впливу архітектури мережі (зокрема, механізму пам'яті) та параметрів еволюції на кінцевий результат, а не на тонкощах стабілізації градієнтного навчання.

Лістинг 1.1 - Створення ігрового поля

```
class Pole:
    def __init__(self, size, cell, win_h, win_v):
        self.size = size
        self.cell = cell
        self.win_h = win_h
        self.win_v = win_v
```

```

self.otstup = 10
self.otstup_block = 10
self.pole_size = self.cell*2 + self.size*self.cell

self.BLACK = (0, 0, 0)
self.WHITE = (255, 255, 255)
self.RED = (255, 0, 0)
self.GREEN = (0, 128, 0)
self.GREY = (36,36,36)
self.L_WHITE = (100,100,100)

self.text_font = pygame.font.SysFont("Arial", 19)
self.text_font_sm = pygame.font.SysFont("Arial", 16)
self.text_font_smm = pygame.font.SysFont("Arial", 14)
self.text_col = (255, 255, 255)
self.text_col_yellow = (250, 249, 170)

self.img_stat =
pygame.image.load('snake_plot_clean.png')

```

2 МЕТОДИ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ РОЗРОБКИ ГРИ

Генетичні алгоритми є потужним методом оптимізації, концепція якого запозичена з механізмів біологічної еволюції. Їхня робота імітує такі природні процеси, як природний відбір, схрещування та випадкові зміни [6]. Цей підхід довів свою ефективність для розв'язання складних завдань оптимізації в різних науково-технічних дисциплінах, де класичні методи, що потребують градієнта або аналітичного опису, виявляються малопридатними.

Алгоритм працює циклічно, поки не буде виконана задана умова зупинки - досягнення потрібної якості рішення, вичерпання обчислювального ресурсу або відсутність прогресу протягом певної кількості ітерацій. Індивід, що демонструє

найкращу пристосованість в останній популяції, приймається за знайдений розв'язок. Сила генетичних алгоритмів полягає в їхній здатності паралельно досліджувати величезні простори параметрів і поступово концентрувати пошук в районах із високою якістю рішення [7].

Як частина більш широкого класу еволюційних алгоритмів, генетичні алгоритми базуються на принципах спадковості та відбору для пошуку оптимальних або близьких до них рішень складних проблем. Окрім базового циклу, існує низка розширених технік і модифікацій, спрямованих на підвищення ефективності та стійкості алгоритму.

Елітизм - це одна з фундаментальних технік, що запобігає втраті найкращих знахідок. У класичній схемі внаслідок стохастичності операцій схрещування та мутації найуспішніші особини можуть випадково не потрапити до наступного покоління. Стратегія елітизму гарантовано зберігає незмінними кілька найкращих індивідів, забезпечуючи монотонне не погіршення якості популяції з часом [6].

Адаптивні генетичні алгоритми покращують процес пошуку за рахунок динамічної зміни власних параметрів. Наприклад, якщо популяція демонструє ознаки застою (збіжності до локального оптимуму), система може автоматично підвищити ймовірність мутації, посиливши дослідницьку складову. І навпаки, при стабільному прогресі параметри можуть закріплюватися для закріплення успішних ознак [7].

Багатоцільова оптимізація - це напрямок для завдань, де потрібно враховувати одночасно кілька критеріїв якості, які часто суперечать один одному. Спеціальні багатоцільові генетичні алгоритми, використовуючи концепцію оптимальності за Парето, формують не єдине рішення, а цілий набір компромісних варіантів, надаючи аналітику вибір найкращого балансу [6].

Популяційні стратегії та коеволюція передбачають використання не однієї, а кількох ізольованих або взаємодіючих популяцій. Такі підходи, включаючи конкурентну коеволюцію, де групи конкурують між собою, дозволяють краще охопити простір пошуку та запобігти ранній збіжності всієї системи до одного, не обов'язково найкращого, рішення.

Використання апріорних знань про предметну область може різко підвищити ефективність пошуку. Наприклад, при оптимізації інженерної конструкції включення до алгоритму знань про фізичні обмеження (міцність, вага) дозволяє відсіяти заздалегідь нежиттєздатні варіанти, звуживши область пошуку та прискоривши отримання корисного результату [6].

Гібридизація з іншими методами відкриває шлях до створення високопродуктивних систем. Поєднання генетичного алгоритму з методами локального пошуку (наприклад, градієнтними) або алгоритмами рою часток дозволяє скористатися перевагами кожного підходу: глобальним дослідженням першого та швидким уточненням рішення другим.

Сфера застосування генетичних алгоритмів надзвичайно широка. У робототехніці їх використовують для планування траєкторій, синтезу систем керування та навігації складних автономних апаратів. В енергетиці вони допомагають знаходити оптимальні режими розподілу навантажень та проектування енергосистем. У фінансовій галузі алгоритми застосовуються для задач управління інвестиційними портфелями та прогнозування ринкових трендів.

2.1 Загальна архітектура програмного комплексу

Система розроблена з урахуванням принципів модульності та низької зв'язності. Це дозволяє незалежно модифікувати, тестувати та покращувати окремі компоненти. Весь комплекс написаний на мові програмування Python з використанням бібліотек NumPy для ефективних матричних обчислень та Pygame для візуалізації.

Основні модулі системи та їх призначення:

- snake_matrix.py (Модуль світу);
- snake_ai.py (Модуль інтелекту);
- snake_draw.py (Модуль візуалізації);

- snake_main.py (Головний контролер);
- md_snakes.py (Модуль генерації популяції);
- plot_to_img.py (Утиліта побудови графіків).

Модуль світу відповідає за логіку гри. Містить клас `Mir`, який зберігає стан ігрового поля (розташування змійки, яблука, стіни), генерує нові яблука, перевіряє колізії та, що найважливіше, обчислює вектор сенсорів для нейронної мережі на основі поточного стану.

Допоміжний модуль для створення великої вихідної популяції з випадковими вагами, що може бути розпаралелено для прискорення початкового етапу.

Цей модуль є диригентом, що об'єднує всі інші компоненти. Він містить головний ігровий цикл, управляє процесом еволюції (цикли по сімействам та епохам), викликає функції відбору, зберігає статистику та ваги у файли. Тут же реалізовано клас `Sesion_snake` для структурованого зберігання всієї інформації про хід експерименту.

Відповідає за графічний інтерфейс користувача. Клас `Role` має методи для малювання ігрового поля, змійки, яблука, а також численних інформаційних панелей: таблиць поточної статистики, історії сімейств, загальної таблиці рекордів та графіку прогресу навчання. Модуль забезпечує інтерактивність (керування швидкістю, пауза).

Серце системи. Містить клас `MemoryPerceptron`, який реалізує архітектуру нейронної мережі з механізмом пам'яті. Також у цьому модулі знаходяться всі функції для роботи з генетичним алгоритмом: ініціалізація ваг `rand_w()`, схрещування `shuff()` та створення нової популяції `w_suffle()`.

Окремий скрипт, який за даними з CSV-файлу будує графік прогресу сімейств і зберігає його як зображення для подальшого відображення в головному вікні.

Взаємодія модулів відбувається за наступним принципом: `snake_main.py` створює екземпляри `Mir` та `MemoryPerceptron`. На кожному кроці гри він запитує у `Mir` вектор сенсорів, передає його в `MemoryPerceptron` для отримання рішення

(напрямку руху) і відправляє це рішення назад у Mir для оновлення стану. Паралельно snake_main.py передає дані про стан у snake_draw.py для відображення. Після завершення ігрової сесії для всієї популяції запускається логіка генетичного алгоритму з модуля snake_ai.py.

Лістинг 2.1 - Створення випадкових ваг

```

self.w_in = np.random.uniform(-1, 1, (self.total_in,
self.lay_1_q))
self.w_lay = np.random.uniform(-1, 1, (self.lay_1_q,
self.lay_2_q))
self.w_out = np.random.uniform(-1, 1, (self.lay_2_q,
self.out_1_q))

```

2.2 Модель ігрового середовища та система сенсорів

Клас Mir імітує світ гри. Ігрове поле представлене двовимірною матрицею arr розміром (size + 2) x (size + 2). Додаткові два рядки та стовпці відведені під непрохідні стіни (значення 11), що обмежують поле. В середині знаходяться вільні клітини (0), клітини, зайняті тілом змійки (1), та клітина з яблуком (7).

Ключовою функцією модуля є sensors(). Її завдання - перетворити графічний стан поля на числовий вектор, зрозумілий нейронній мережі. Замість того, щоб подавати на вхід сиру матрицю або координати, була реалізована система вісімкових дальномірів.

Алгоритм роботи sensors() - визначається позиція голови змійки.

Задається вісім напрямків: вгору, вниз, вліво, вправо та чотири діагональні. Для кожного напрямку запускається віртуальний промінь, який крок за кроком рухається від голови. На кожному кроці променя перевіряється вміст клітини:

Якщо зустрілася стіна - фіксується нормована відстань до неї (1 / step) і пошук у цьому напрямку припиняється. Якщо зустрілося тіло змійки (крім голови)

- фіксується відстань до тіла. Якщо зустрілося яблуко - фіксується відстань до яблука.

Таким чином, для кожного з 8 напрямків ми отримуємо 3 значення (відстань до стіни, тіла, яблука), що у сумі дає 24-мірний вектор стану.

Такий підхід є інваріантним до абсолютного положення змійки на полі і концентрується на відносній геометрії оточення, що значно прискорює навчання.

Лістинг 2.2 - Фрагмент що демонструє логіку одного променя for d in directions:

```
dist_wall = dist_body = dist_apple = 0
step = 1
while True:
    pos = (head[0] + d[0]*step, head[1] + d[1]*step)
    cell_value = self.arr[pos]
    if cell_value == 11: # Стіна
        dist_wall = 1 / step
        break
    elif cell_value == 1 and pos != head: # Тіло змії
        dist_body = 1 / step
    elif cell_value == 7: # Яблуко
        dist_apple = 1 / step
    step += 1
self.snake_sensors.extend([dist_wall, dist_body,
dist_apple])
```

2.3 Архітектура нейронної мережі з механізмом пам'яті MemoryPerceptron

Клас MemoryPerceptron реалізує власне мозок агента. Його архітектура є розвиненням класичного перцептрона.

Основні компоненти:

- вхідний шар;
- перший прихований шар;
- другий прихований шар;
- вихідний шар ;
- вектор пам'яті.

Вхідний шар приймає об'єднаний вектор `inp_total`, що складається з поточних сенсорів `inp_1` (24 значення) та вектора пам'яті `memory` (ще 24 значення). Разом - 48 входів.

Для першого прихованого шару (`lay_1`) кількість нейронів задається параметром `lay_1_q` (наприклад, 16). Застосовується функція активації ReLU для введення нелінійності.

Для другого прихованого шару (`lay_2`) - кількість нейронів `lay_2_q` (наприклад, 12). Також з активацією ReLU. Наявність двох шарів дозволяє мережі моделювати більш складні функції.

Для вихідного шару (`out_1`) - чотири нейрони без функції активації (лінійні), що відповідають оцінці якості кожного з чотирьох напрямків руху (вгору, вниз, вліво, вправо).

Вектор пам'яті це масив фіксованої довжини (рівної розмірності сенсорів), який оновлюється на кожному кроці за формулою: $memory = memory_decay * memory + inp_1$. Параметр `memory_decay` (наприклад, 0.7) визначає швидкість «забування»: значення, близьке до 1, зберігає пам'ять довше, близьке до 0 - робить агента більш реактивним.

Процес прямого поширення оновлення пам'яті: `self.memory = self.memory_decay * self.memory + inp_1` об'єднання векторів: `inp_total = np.concatenate([inp_1, self.memory])` обчислення стану першого прихованого шару: `self.lay_1 = self.relu(np.dot(inp_total, self.w_in))` обчислення стану другого прихованого шару: `self.lay_2 = self.relu(np.dot(self.lay_1, self.w_lay))` обчислення вихідних оцінок: `self.out_1 = np.dot(self.lay_2, self.w_out)` вибір дії: за допомогою `np.argmax(self.out_1)` визначається індекс напрямку з найвищою оцінкою.

Таким чином, на кожному кроці мережа приймає рішення не лише на основі того, що бачить зараз, але й з урахуванням того, що вона бачила та робила протягом останніх кількох ходів. Це дозволяє уникнути безглузвих циклічних рухів та покращити стратегічне планування.

Лістинг 2.3 - Генерація наступного покоління з використанням пам'яті

```
new_pop = []
for p in parents:
    child = MemoryPerceptron(p.in_l_q, p.lay_1_q,
p.lay_2_q, p.out_l_q)
    child.get_w(p.w_in.copy(), p.w_lay.copy(),
p.w_out.copy())
    new_pop.append(child)
return new_pop
```

2.4 Представлення індивіда та ініціалізація популяції

Кожна змійка в популяції керується своєю власною нейронною мережею, архітектура якої фіксована, а унікальність визначається ваговими матрицями: w_{in} , w_{lay} та w_{out} . Саме ці матриці є аналогом хромосоми в генетичному алгоритмі. Розміри матриць залежать від конфігурації мережі:

- w_{in} : розмірність ($total_in$, lay_1_q), де $total_in = in_l_q * 2$ (сенсори + пам'ять);
- w_{lay} : розмірність (lay_1_q , lay_2_q);
- w_{out} : розмірність (lay_2_q , out_l_q).

Таким чином, індивід кодується не як простий рядок чисел, а як список цих трьох двовимірних масивів. Така структура зберігає семантичну інформацію про призначення кожного параметра.

Початкова популяція генерується випадковим чином. Для кожної нової змійки викликається метод `rand_w()` класу `MemoryPerceptron`, заповнює всі вагові матриці випадковими числами з рівномірного розподілу в діапазоні $[-1.0, 1.0]$.

Лістинг 2.4 - ініціалізація випадкових ваг `rand_w(self)`

```

        self.w_in    =    np.random.uniform(-1,    1,    (self.total_in,
self.lay_1_q))
        self.w_lay    =    np.random.uniform(-1,    1,    (self.lay_1_q,
self.lay_2_q))
        self.w_out    =    np.random.uniform(-1,    1,    (self.lay_2_q,
self.out_1_q))

```

Розмір початкової популяції (`pop_size`) є критичним гіперпараметром. Велика популяція забезпечує високу генетичну різноманітність і зменшує ризик потрапляння в локальний оптимум, але значно збільшує час обчислень для однієї епохи. Мала популяція швидше збігається, але може «застрягти» на посередньому рішенні.

2.5 Функція пристосованості

Після того як усі змійки в популяції провели свою ігрову сесію (до смерті або до перевищення максимальної кількості ходів), необхідно кількісно оцінити успішність кожної з них. Ця оцінка називається пристосованістю (`fitness`) і є основним критерієм природного відбору.

У найпростішому випадку пристосованістю може бути кількість зібраних яблук (`score`). Однак такий підхід не розрізняє змійку, яка збрала 10 яблук за 1000 ходів, і ту, що збрала 10 яблук за 100 ходів. Друга є очевидно ефективнішою. Тому в даній реалізації використовується складена фітнес-функція.

Індивіди сортуються за двома ключами: спочатку за кількістю яблук за спаданням ($-i[0]$), а потім - за загальною кількістю зроблених ходів за зростанням ($i[1]$). Таким чином, змійка з більшою кількістю яблук завжди отримує пріоритет, а серед змійок з однаковим рахунком кращою вважається та, що досягла результату швидше (зробила менше ходів).

Лістинг 2.5 - Сортування за фітнесом

```
self.era_snakes_list.sort(key=lambda i: (-i[0], i[1]))
```

Такий підхід сприяє еволюції не просто виживаючих, а ефективно діючих агентів.

2.6 Відбір

Метою етапу відбору є визначення тих індивідів (батьків), чий гени будуть використані для створення наступного покоління. У цій роботі реалізовано простий, але ефективний метод елітарного відбору (elitist selection).

Після сортування всієї популяції за фітнесом, відбирається фіксована кількість `parents_quantity` (наприклад, 15) найкращих індивідів. Ця група і утворює батьківський пул. Елітарний відбор гарантує, що найкращі знайдені рішення ніколи не будуть втрачені в процесі еволюції (принцип елітизму).

Лістинг 2.6 - Відбір кращих індивідів

```
ses.select_the_best(family_number, era, parents_quantity)
    ses.deti_list = ai.w_suffle(ses.era_weights_list,
probability_a, probability_b, probability_mutation)
    pole.now_top(ses.df_now(family_number, era),
family_number, era, len(ses.deti_list))
    pole.top_table(ses.df_all())
```

```

ses.save_best_weights()
if era in ses.df_cols:
    ses.update_pivot_scores(family_number, era)
    pole.pivot_score(ses.df_pivot_scores)
pole.show_plot()
pygame.display.update()

```

2.7 Схрещування та мутація

Ці два етапи часто поєднуються і відповідають за створення нових, генетично різноманітних особин. У даній реалізації вони інтегровані в одну операцію `shuff()`, яка є основою методу `w_shuffle()`.

Операція `shuff()` для двох батьківських вагових матриць:

Обидві матриці `w1` і `w2` «розгортаються» в одномірні вектори `a` і `b` за допомогою методу `ravel()`.

Для кожного елемента `i` у цих векторах створюється нове значення `c[i]` шляхом ймовірнісного вибору з ймовірністю `probability_a` (наприклад, 70%) береться ген від першого батька (`a[i]`), з ймовірністю `probability_b` (30%) - ген від другого батька (`b[i]`), з ймовірністю `k` (ймовірність мутації, наприклад, 5%) ген ігнорується, і на його місце генерується абсолютно нове випадкове значення в діапазоні `[-1, 1]`. Отриманий одномірний вектор `c` перетворюється назад у вихідну двовимірну форму матриці батьків.

Лістинг 2.7 - Операція перемішування з мутацією

```

def shuff(self, w1, w2, probability_a=70, probability_b=30, k=0):
    a = w1.ravel()
    b = w2.ravel()
    c = []
    for i in range(len(a)):

```

```

c.append(random.choices([a[i], b[i], random.uniform(-1,
1)],

weights=[probability_a,
probability_b, k])[0])
return np.array(c).reshape(wl.shape)

```

Операція `w_shuffle()` для створення нової популяції - ця функція приймає список ваг найкращих батьків (`wl`). Для створення нового покоління вона послідовно схрещує кожного батька з кожним (включаючи схрещування з самим собою). Для кожної пари (i, j) генеруються два нащадки перший нащадок створюється з ненульовою ймовірністю мутації k , другий нащадок створюється з нульовою ймовірністю мутації ($k=0$), тобто є чистим схрещуванням.

Такий підхід забезпечує експоненційне зростання популяції після схрещування. Щоб уникнути експоненційного вибуху, наступне покоління формується не з усіх нащадків, а знову проходить через етап відбору після оцінки фітнесу. Проте, активне схрещування всіх з усіма значно розширює простір пошуку та сприяє комбінуванню різних корисних ознак.

2.8 Стратегія сімейств та епох

Навчання в програмі організоване в ієрархії вищого порядку. Сімейство - окрема незалежна еволюційна лінія, що починається зі своєї власної випадково ініціалізованої популяції. Мета - дослідити, як різні початкові умови впливають на кінцевий результат. Епоха - один повний цикл генетичного алгоритму всередині сімейства: оцінка $\rightarrow$ відбір $\rightarrow$ схрещування/мутація $\rightarrow$ формування нового покоління.

Моделювання декількох сімейств паралельно дозволяє зменшити ймовірність того, що весь експеримент «застрягне» в невдалому локальному

оптимумі, порівняти ефективність різних траєкторій еволюції, зібрати більш надійну статистику про поведінку алгоритму.

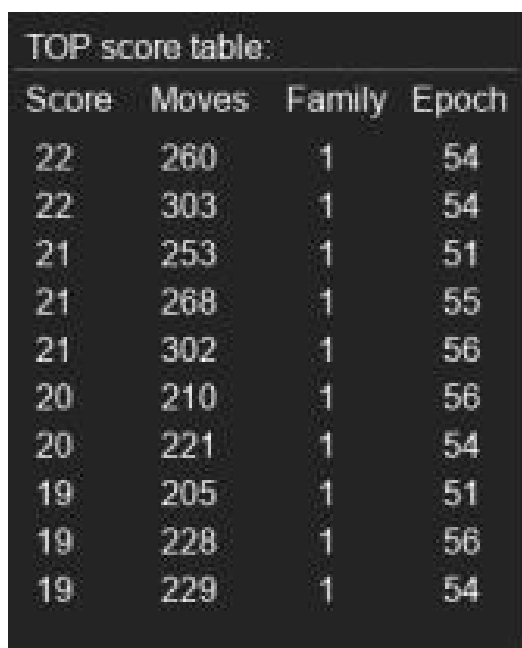
Кожне сімейство проходить фіксовану кількість епох (наприклад, 100), після чого його найкращі результати фіксуються. Весь цей процес керується головним контролером у `snake_main.py`.

Family: 1 Epoch: 58	
Score	Moves
19	239
16	195
16	273
15	162
15	181
14	152
14	157
14	166
14	173
14	179
Population: 465	

Рисунок 2.1 - Таблиця результатів поточного сімейства

2.9 Система візуалізації та моніторингу процесу навчання

Однією з ключових складових успіху проекту стала розробка зручного інструменту для спостереження за процесом навчання в реальному часі. Пасивне збереження лог-файлів не дозволяє оперативно оцінити прогрес або виявити аномалії. Тому був створений складний графічний інтерфейс на базі бібліотеки Pygame.



Score	Moves	Family	Epoch
22	280	1	54
22	303	1	54
21	253	1	51
21	268	1	55
21	302	1	56
20	210	1	56
20	221	1	54
19	205	1	51
19	228	1	56
19	229	1	54

Рисунок 2.2 - Таблиця для спостереження результатів у реальному часі

2.10 Візуалізація ігрового процесу

Основним елементом інтерфейсу є ігрове поле, яке відображає поточний стан симуляції для однієї обраної змійки. Користувач може вмикати та вимикати анімацію, регулювати її швидкість клавішами, що дозволяє детально вивчати поведінку агентів на різних стадіях навчання:

- стіни позначаються темно-сірим кольором;
- тіло змійки - білими квадратами;
- голова змійки - може виділятися відтінком;
- яблуко - малиновим або зеленим квадратом.

Під полем у реальному часі відображається лічильник зібраних яблук (Score) та загальна кількість ходів (Moves) для поточної змійки.

2.11 Інформаційні панелі та таблиці

Навколо ігрового поля розташовано низку панелей, що відображають статистику. Панель параметрів гри. Фіксує основні налаштування експерименту: унікальний ідентифікатор сесії, розмір поля, кількість сенсорів, конфігурацію нейромережі, кількість сімейств, епох та розмір початкової популяції.

Таблиця поточної епохи (`now_top`). Невелика таблиця, що показує ТОП-10 найкращих змійок поточного сімейства в поточній епосі. Для кожної відображається кількість яблук (`Score`) та ходів (`Moves`). Ця панель дозволяє відстежувати динаміку покращення всередині однієї епохи.

Зведена таблиця успішності сімейств (`pivot_scores`). Велика таблиця, рядки якої - номери сімейств, а стовпці - певні епохи (наприклад, кожна 5-та). У клітинці записується максимальний рахунок, досягнутий найкращою змійкою даного сімейства в цю епоху. Ця таблиця є основним інструментом порівняльного аналізу: вона наочно показує, яке сімейство навчається швидше, яке вийшло на плато, а яке раптово покращило результати.

Загальна таблиця рекордів (`top_table`). Відображає абсолютні рекорди за всю історію симуляції (усі сімейства, усі епохи). Для кожного рекорду вказується рахунок, кількість ходів, номер сімейства та епохи, де цей рекорд був встановлений. Ця панель дає відповідь на головне питання: Якого найкращого результату нам вдалося досягти?.

2.12 Графік прогресу навчання

Найважливішим інструментом аналізу є графік, що будується в окремому процесі за допомогою бібліотеки `Matplotlib`. Скрипт `plot_to_img.py` читає

збережений CSV-файл зі статистикою, фільтрує дані за поточною міткою часу `time_label` та будує графік залежності рахунку від номера епохи.

Ключові особливості графіка - кожне сімейство представлено окремою лінією, для згладжування шуму та кращої візуалізації тренду застосовується ковзне середнє з вікном у 3 епохи.

Графік автоматично зберігається як зображення `snake_plot.png` і завантажується в головне вікно програми, де оновлюється після завершення кожного сімейства.

Цей графік дозволяє оцінити не лише кінцевий результат, але й динаміку навчання: чи є воно плавним, чи стрибкоподібним, на якій епосі відбувається найбільший прогрес, чи спостерігається перенавчання або забуття.

Лістинг 2.8 - Фрагмент логіки побудови графіка

```

pivot_table = data.pivot_table(index='epoch', columns='family',
values='score', aggfunc='max')
for family in pivot_table.columns:
    smoothed_scores =
pivot_table[family].rolling(window=3).mean()
    plt.plot(pivot_table.index, smoothed_scores, label=f'Family
{family}')
    plt.xlabel('Епоха')
    plt.ylabel('Рахунок (згладжений)')
    plt.title('Прогрес навчання по сімействам')
    plt.legend()
    plt.grid(True)
    plt.savefig('snake_plot.png')

```

2.13 Система збереження та відтворення результатів

Для забезпечення наукової цінності експериментів необхідно мати можливість їх відтворення та подальшого аналізу. У програмі реалізовано два

рівні збереження даних - збереження статистики у текстовий формат (CSV).

Після завершення кожного сімейства клас `Sesion_snake` викликає метод `save_game_stat()`. Усі результати, накопичені в `DataFrame snake_statistics`, записуються в файл з іменем `snake_statistics_<time_label>.csv`. Кожен рядок файлу містить повну інформацію про одну змійку в одну епоху:

- `family, epoch` - ідентифікатори;
- `score, moves` - результати гри;
- `desk_size, sensors, lay_1, lay_2` - параметри середовища та архітектури;
- `time_label` - унікальна мітка сесії для групування експериментів.

Серіалізація ваг найкращих мереж метод `save_best_weights()` зберігає не лише результати, а й самі «мозки» агентів. Список `snake_weights_data`, що містить вагові матриці кращих змій кожного сімейства в ключові епохи, серіалізується за допомогою модуля `pickle` у файл `snake_best_weights_<time_label>.bin`. Це дозволяє:

Відтворити та продемонструвати поведінку чемпіонів без необхідності повторного навчання. Продовжити навчання з цієї точки, використавши збережені ваги як початкову популяцію для нового експерименту. Перенести знання: використати ваги з успішного експерименту як основу для навчання в іншому, більш складному середовищі (техніка *transfer learning*). Таким чином, система забезпечує повну відтворюваність досліджень та створює базу знань для майбутніх експериментів.

3 РЕАЛІЗАЦІЯ ГРИ

Розробка комп'ютерних ігор із застосуванням нейронних мереж та генетичних алгоритмів є комплексним та багатоетапним процесом, який поєднує принципи штучного інтелекту та еволюційного програмування.

Реалізація гри «Змійка» на мові програмування Python з використанням бібліотеки `Rugame` [8, 12] включає послідовне виконання низки технічних завдань.

Основні переваги обраного способу реалізації це простота та зручність - бібліотека Pygame значно спрощує розробку графічного інтерфейсу, надаючи зрозумілі інструменти для малювання, обробки вводу користувача та керування часом. Це дозволяє швидко створювати функціональні прототипи та концентруватися на алгоритмічній частині проекту.

Гнучкість - мова Python та її екосистема дозволяють легко інтегрувати в проект потужні сторонні бібліотеки, такі як NumPy для обчислень у нейронних мережах, що є критично важливим для реалізації складної AI-логіки поруч із графічним інтерфейсом.

Серед недоліків обраного підходу можна виділити продуктивність - у порівнянні з низькорівневими мовами, Python може демонструвати меншу швидкість виконання, особливо в обчислювально-навантажених задачах, таких як масова симуляція ігрових сесій для генетичного алгоритму.

Обмежені графічні можливості - pygame є легким фреймворком і не надає такого розвиненого інструментарію для створення графіки, як повноцінні ігрові рушії. Однак для дослідницького проекту, де візуалізація слугує для наочності, а не є основним продуктом, цього достатньо.

Ініціалізація гри (лістинг 3.1) перш за все, необхідно ініціалізувати Pygame та налаштувати основні параметри гри, такі як розмір вікна, кольори та швидкість гри.

Лістинг 3.1 – Найпростіший приклад ініціалізації гри «Змійка»

```
import pygame
import time
import random

pygame.init()

white = (255, 255, 255)
yellow = (255, 255, 102)
```

```
black = (0, 0, 0)
red = (213, 50, 80)
green = (0, 255, 0)
blue = (50, 153, 213)
dis_width = 800
dis_height = 600
dis = pygame.display.set_mode((dis_width, dis_height))
pygame.display.set_caption('Snake Game by Edureka')
clock = pygame.time.Clock()
snake_block = 10
snake_speed = 15
```

Ми ініціалізуємо Pygame (див. лістинг 3.1), визначаємо кольори, налаштовуємо розмір вікна гри та створюємо його, а також встановлюємо швидкість гри.

3.1 Реалізація програмного комплексу та технічні деталі

Програмна реалізація системи була виконана з дотриманням принципів об'єктно-орієнтованого програмування (ООП) для забезпечення чіткої структури, повторного використання коду та легкості модифікації.

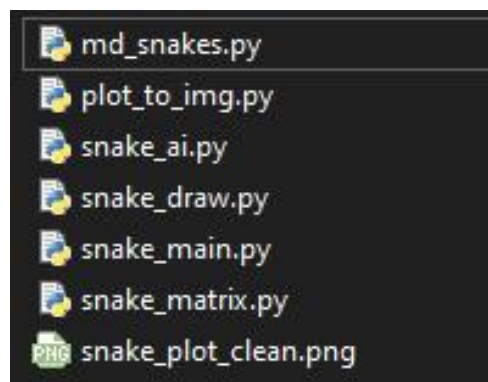


Рисунок 3.1 - Файли програми що знаходяться в 1 просторі

3.2 Залежності та середовище виконання

Система розроблена на мові Python 3.8+. Цей вибір обумовлений широким спектром наукових бібліотек, простотою синтаксису та крос-платформовістю. Використані зовнішні бібліотеки NumPy - для ефективних векторно-матричних обчислень, необхідних для роботи нейронної мережі. Операції з точковим добутком (`np.dot`) та універсальними функціями (`np.maximum` для ReLU) реалізовані на рівні оптимізованих бінарних модулів на C, що забезпечує високу швидкість.

Pygame - для створення графічного інтерфейсу користувача, візуалізації ігрового процесу та інформаційних панелей.

Pandas - для роботи з табличними даними статистики у модулі побудови графіків (`plot_to_img.py`). Використовується для читання CSV-файлів, фільтрації та агрегації даних.

Matplotlib - побудова графіків прогресу навчання.

Всі залежності описані у файлі `requirements.txt`, що дозволяє швидко відтворити робоче середовище командою `pip install -r requirements.txt`.

3.3 Структура проектних файлів

Проект має чітку модульну структуру:

- `snake_main.py` # Головний скрипт, точка входу;
- `snake_ai.py` # Клас нейронної мережі та ГА;
- `snake_matrix.py` # Модель ігрового світу;
- `snake_draw.py` # Візуалізація (Pygame);
- `md_snakes.py` # Генератор початкової популяції;
- `plot_to_img.py` # Скрипт побудови графіків;

- snake_plot_clean.png # Заглушка для графіка;
- requirements.txt # Список залежностей;
- snake_statistics_*.csv # Результати експериментів (генерується);
- snake_best_weights_*.bin # Збережені ваги мереж (генерується);

Така організація спрощує навігацію, тестування та внесення змін. Наприклад, для зміни методу відбору в ГА достатньо модифікувати лише snake_ai.py, не торкаючись інших модулів.

3.4 Обробка подій та керування симуляцією

Головний цикл у snake_main.py інтегрує всі компоненти. Важливою його частиною є обробка введення користувача для інтерактивного керування процесом клавіші Р/О - вмикання/вимикання анімації. Коли анімація вимкнена, симуляції виконуються на максимальній швидкості, що значно прискорює навчання.

Клавіші S/F - збільшення/зменшення швидкості анімації для детального спостереження. Клавіша Q - аварійне завершення програми з коректним збереженням поточних результатів.

Ця функціональність робить систему зручним інструментом як для автономного запуску довгих експериментів, так і для навчальної демонстрації.

3.5 Оптимізація продуктивності

Найбільш обчислювально-витратною частиною є оцінка фітнесу - прогон тисяч ігрових сесій. Для оптимізації було впроваджено кілька прийомів.

Векторизація обчислень у NumPy. Усі операції в нейронній мережі (множення матриць, активації) виконуються через NumPy, що на порядки швидше за чисті цикли Python.

Мінімізація графічних операцій. У режимі швидкого навчання (анімація вимкнена) візуалізація не виконується, що знімає значне навантаження з CPU/GPU.

Ефективна робота з пам'яттю. Ваги мереж зберігаються як масиви NumPy, а не списки Python, що зменшує накладні витрати.

Профілювання програми показало, що при вимкненій графіці основним «вузьким місцем» є ігрова логіка (перевірка колізій, пошук шляху променями сенсорів). Для подальшого прискорення можна розглянути переписування цих частин на Cython або використання JIT-компіляції (Numba).

3.6 План та методологія експериментальних досліджень

Мета експериментів - емпірично дослідити, як різні фактори впливають на ефективність навчання агента. Було розроблено план, що включає серію контрольованих експериментів, де змінювався один параметр за інших незмінних умов.

Лістинг 3.2 - Фрагмент коду що забезпечує зручні зміни в роботі створення умов для експериментів

```
razmer_pole = 15
    razmer_cell = 22
    win_h = 1400
    win_v = 680
    os.environ['SDL_VIDEO_WINDOW_POS'] = '%i,%i' % (40, 40)
    input_layer_quantity = 32
    neurons_layer_quantity_1 = 16
```

```

neurons_layer_quantity_2 = 12
otput_layer_quantity = 4
number_of_families = 10
epochs = 100
parents_quantity = 15
parents_quantity_s_fam = 15
random_snakes_quantity = 50000
probability_a = 50
probability_b = 50
probability_mutation = 5
probability_mutation_s_fam = 1
speed_snake = 0.02

```

3.7 Контрольна конфігурація

Для порівняння було обрано наступні базові параметри:

- розмір поля: 15x15 клітин (без урахування стін);
- архітектура НМ;
- параметри ГА;
- структура навчання 10 сімейств по 100 епох кожне.

Архітектура НМ: 24 входи (сенсори) + 24 (пам'ять) = 48; lay_1_q = 16; lay_2_q = 12; out_1_q = 4. memory_decay = 0.7

Параметри ГА: Розмір початкової популяції (pop_size) = 200; кількість батьків для відбору (parents_quantity) = 15; ймовірності при схрещуванні: probability_a=70%, probability_b=30%; ймовірність мутації (k) = 5%.

3.8 Змінні параметри для дослідження

Було заплановано наступні серії експериментів:

Вплив ймовірності мутації (k): $k = 0\%$ (без мутацій), 1% , 5% (база), 10% , 20% . Дуже низька ймовірність призведе до передчасної збіжності, дуже висока - до хаотичного пошуку без покращення.

Вплив розміру популяції (pop_size): $pop_size = 50, 100, 200$ (база), 500 . Гіпотеза: більша популяція покращує покриття простору пошуку та кінцевий результат, але зростає час навчання.

Вплив архітектури мережі (розмір прихованих шарів):

- варіація 1: $lay_1_q = 8, lay_2_q = 6$ (менша мережа);
- варіація 2: $lay_1_q = 32, lay_2_q = 24$ (більша мережа);
- варіація 3: Відсутність другого прихованого шару ($lay_2_q = 0$, вихід прямо з lay_1).

Вплив механізму пам'яті: порівняння MemoryPerceptron (з $memory_decay=0.7$) та звичайного перцептрона ($memory_decay=0.0$, тобто пам'ять не використовується). Гіпотеза: мережа з пам'яттю демонструватиме більш стабільну та стратегічну поведінку, уникаючи циклів.

Вплив системи сенсорів: порівняння повної системи ($8 \text{ напрямків} * 3 \text{ об'єкти} = 24 \text{ входи}$) зі спрощеною (наприклад, лише 4 основні напрямки або лише відстані до стін та яблука).

3.9 Метрики оцінки

Для кількісного порівняння експериментів використовувався максимальний досягнутий рахунок: головний показник ефективності.

Середній рахунок найкращого індивіда по епохах: показує стабільність навчання.

Епоха збіжності: номер епохи, після якої рахунок найкращого індивіда не покращується більше ніж на 5% протягом 20 наступних епох. Показує швидкість навчання.

Різноманітність популяції: обчислювалась як середнє стандартне відхилення ваг по всій популяції на останніх епохах. Низька різноманітність свідчить про передчасну збіжність.

3.10 Проведення експериментів та отримання даних

Експерименти проводились на персональному комп'ютері з процесором Intel Core i7, 32 ГБ оперативної пам'яті. Для кожного набору параметрів програма запускалась в повністю автоматичному режимі (анімація вимкнена). Час виконання одного повного експерименту (10 сімейств по 100 епох) складав від 2 до 8 годин залежно від розміру популяції та складності мережі.

Система моніторингу дозволяла спостерігати за прогресом в реальному часі через інформаційні панелі. Після завершення кожного запуску автоматично генерувались:

- файл CSV зі всією детальною статистикою;
- файл BIN з вагами найкращих мереж;
- графік прогресу сімейств у вигляді зображення PNG.

3.11 Результати базового експерименту

У контрольній конфігурації система продемонструвала здатність до ефективного навчання.

Перші 10-20 епох: швидке зростання рахунку від 0-2 до 10-15 яблук. Агенти навчались базовим навичкам: рухатись до яблука, уникати стін.

Епохи 20-60: повільніше, але стає покращення до 30-50 яблук. З'являється стратегія: обходити власне хвіст, планувати маршрут, уникати пасток.

Після 60-ї епохи: прогрес уповільнюється, багато сімейств виходять на плато. Однак, в окремих сімействах спостерігались "прориви", коли рахунок різко зростав на 10-15 очок після тривалого застою, що свідчить про ефективність мутацій у пошуку нових рішень.

Максимальний рахунок, досягнутий у базовому експерименті, склав 71 яблуко. Це відбулося в сімействі №3 на 89-й епосі. Візуальне спостереження за цією змійкою показало високорозвинену стратегію: вона рухалась "змійкою", заповнюючи поле певним патерном, що дозволяло довго уникати пасток. Графік прогресу сімейств чітко показав, що хоча траєкторії різні, більшість з них демонструє позитивний тренд.

	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
1	2	3	3	3	2	3	3	3	5	15	14	21	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Рисунок 3.2 - Таблиця для спостереження за змінами сімейств на протязі епох

3.12 Експеримент 1: вплив ймовірності мутації

Результати підтвердили гіпотезу про критичну роль мутації.

$k = 0\%$: Навчання швидко (на 20-30 епосі) виходило на низьке плато (15-25 яблук). Різноманітність популяції стрімко падала. Алгоритм швидко знаходив посереднє рішення і не міг його покращити.

$k = 1\%$: Результати були кращими за відсутність мутацій, але все ще гіршими за базовий випадок. Максимум ~45 яблук.

$k = 5\%$ (база): Найкращий баланс. Мутації дозволяли вийти з локальних оптимумів, що призводило до проривів на пізніх етапах.

$k = 10\%, 20\%$: Прогрес став більш хаотичним. Середній рахунок міг бути високим, але стабільність падала. Занадто високий рівень шуму перешкоджав збереженню набутих корисних ознак.

3.13 Експеримент 2: вплив розміру популяції

$pop_size = 50$: Навчання було найшвидшим рання збіжність, але кінцевий результат обмежений (~35 яблук). Малої різноманітності не вистачило для пошуку складних стратегій.

$pop_size = 100, 200$: показали найкращі результати. Популяція у 200 особин дала більш стабільний прогрес і вищий максимальний рахунок.

$pop_size = 500$: максимальний рахунок був аналогічним популяції у 200, але час навчання зріс більш ніж вдвічі. Зменшення приросту ефективності свідчить про те, що для даної задачі існує достатній розмір популяції, після якого подальше збільшення дає незначну вигоду.

3.14 Експеримент 3: вплив механізму пам'яті

Порівняння MemoryPerceptron та звичайного перцептрона дало однозначні результати:

Без пам'яті ($\text{memory_decay}=0.0$): Агент часто потрапляв у нескінченні цикли наприклад, рух по квадрату. Йому було важко "згадати", що він вже був у цьому місці кілька ходів тому. Максимальний рахунок обмежувався 30-40 яблуками.

З пам'яттю ($\text{memory_decay}=0.7$): Циклічна поведінка була значно рідшою. Агент демонстрував здатність "викручуватися" з тісних ситуацій, що свідчить про використання контексту. Це прямо вплинуло на максимальний рахунок (71 проти 38 у найкращому випадку без пам'яті).

python

Демонстрація різниці в коді: оновлення пам'яті є ключовим# З пам'яттю:

```
self.memory = 0.7 * self.memory + inp_1
```

```
inp_total = np.concatenate([inp_1, self.memory])
```

Без пам'яті (емульовано):# self.memory = 0.0 * self.memory + inp_1

Пам'ять завжди обнуляється# inp_total = inp_1 # Мережа бачить тільки поточний стан

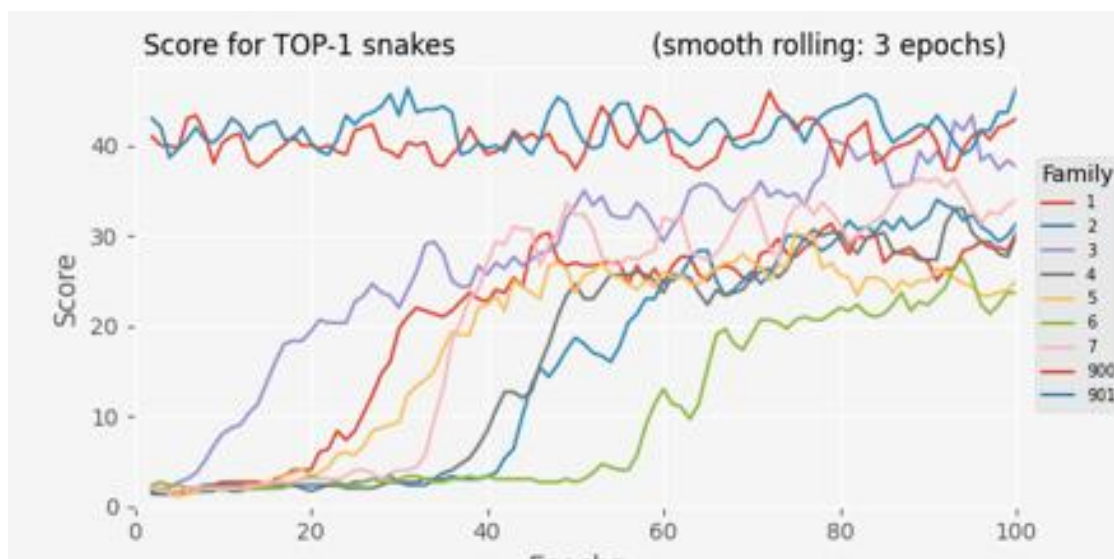


Рисунок 3.4 - Гарфік для порівняння просування навчання

3.15 Аналіз отриманих результатів та обговорення

Аналіз даних експериментів дозволяє зробити низку висновків про роботу системи та природу навчання в даній задачі.

Баланс дослідження та використання. Експеримент з мутацією наочно показав, що ідеальний алгоритм має не лише відтворювати успішні риси схрещування, але й постійно досліджувати нові можливості мутація. Значення $k=5\%$ виявилось оптимальним для цієї задачі.

Необхідність контексту. Результати експерименту з пам'яттю є, мабуть, найважливішим висновком. Проста нейромережа, яка приймає рішення лише на основі миттєвого "знімка", принципово обмежена в динамічних середовищах. Механізм пам'яті, навіть такий простий, є необхідним мінімумом для формування нетривіальної поведінки.

Достатня генетична різноманітність. Маленькі популяції швидко втрачають різноманітність і "висихають". Велика популяція - це страхівка від передчасної збіжності, хоча й за рахунок обчислювальних витрат.

3.16 Обмеження та проблеми, що виявились

"Плато" навчання. Більшість сімейств виходили на плато на рівні 40-60 яблук. Подолати його вдавалося лише окремим "щасливим" сімействам. Це говорить про те, що простір можливих стратегій дуже великий, а шлях до справді оптимальної поведінки (як у людини, що може збирати сотні яблук) вимагає або на порядки більше ітерацій, або принципово інших механізмів навчання (наприклад, ієрархічного посилення навчання).

Вплив випадковості. Різні сімейства, стартуючи з випадкових ваг, досягали різних рівнів успіху. Це підкреслює стохастичну природу генетичних алгоритмів

та необхідність проведення багаторазових запусків для отримання статистично значущих результатів.

Обчислювальна складність. Головним обмеженням для поглибленого дослідження залишається час. Експеримент з популяцією у 500 особин тривав майже 8 годин. Для дослідження впливу десятків параметрів необхідна значно потужніша обчислювальна база або розпаралелювання.

3.17 Якісний аналіз поведінки навчених агентів

Спостереження за найкращими змійками дозволило виділити типи поведінки, що еволюціонували:

- “Патрульна”;
- “Агресивна”;
- “Обережна”.

Патрульна стратегія: Змійка рухається вздовж краю поля, поступово заповнюючи його “змійкою”. Дуже ефективно для уникнення пасток. Агресивна стратегія швидкий рух до яблука найкоротшим шляхом з майстерним оминанням власного тіла в останній момент. Обережна стратегія уникання центр поля, де ймовірність потрапити в пастку вища. Агент намагається залишати собі “запасний вихід”.

Ці стратегії не були запрограмовані - вони виникли самоорганізовано в результаті еволюційного пошуку, що підтверджує потужність обраного підходу.

3.18 Порівняння з альтернативними підходами

Для оцінки ефективності розробленої системи корисним є її порівняння з іншими можливими підходами до навчання гри «Змійка».

Навчання з підкріпленням на основі градієнта Policy Gradients, A2C. Ці методи можуть бути ефективнішими за ГА в плані швидкості збіжності, оскільки напрямок оновлення ваг обчислюється аналітично. Однак вони вимагають диференційованого середовища та складнішої реалізації, схильні до проблем зі стабільністю навчання та вибору гіперпараметрів. Наша система значно простіша і стійкіша.

Детерміновані алгоритми пошуку шляху. Алгоритм завжди знайде шлях до яблука, якщо він існує. Однак він не може формувати довгострокову стратегію обходу власного тіла і легко створює безвихідні ситуації. Навчена нейромережа, навпаки, володіє емерджентною стратегією уникнення пасток.

Людина-гравець. Досвідчений гравець може набирати сотні очок. Наша система (71 очко) значно поступається людині, але перевершує випадкового агента (0-2 очки) та більшість простих алгоритмічних агентів. Прірва між агентом і людиною вказує на складність формування абстрактного планування та гнучкої логіки.

Реалізована система продемонструвала свою працездатність та ефективність. Експерименти підтвердили гіпотези про важливість мутації, механізму пам'яті та достатнього розміру популяції. Система здатна автономно виявляти нетривіальні ігрові стратегії, хоча її максимальна ефективність поки що обмежена складністю задачі та обчислювальними ресурсами. Отримані результати створюють міцну основу для подальших досліджень і вдосконалень.

ВИСНОВКИ

У ході виконання даної дипломної роботи було повністю реалізовано поставлені завдання з розробки, дослідження та аналізу автономного інтелектуального агента для гри «Змійка». Проведена робота дозволила сформулювати низку ключових наукових та практичних висновків.

Успішно розроблено модульний програмний комплекс, що інтегрує модель ігрового середовища, архітектуру нейронної мережі з механізмом короткострокової пам'яті, генетичний алгоритм навчання та інтерактивну систему візуалізації. Комплекс демонструє високу ступінь завершеності та може використовуватися як навчальний стенд для вивчення основ нейронних мереж, генетичних алгоритмів та зміцненого навчання. Реалізація на мові Python з використанням бібліотек NumPy та Pygame забезпечила гарний баланс між продуктивністю, наочністю та зручністю модифікації.

Архітектура нейронної мережі MemoryPerceptron з двома прихованими шарами та контекстною пам'яттю виявилася ефективною для поставленої задачі. Експериментально доведено, що наявність механізму пам'яті ($\text{memory_decay} = 0.7$) є критично важливою для формування стійкої, нециклічної поведінки агента. Агенти без пам'яті демонстрували значно гірші результати (максимум 38 яблук проти 71 з пам'яттю), часто потрапляючи в пастки через нездатність враховувати послідовність своїх дій. Це підтверджує тезу про те, що навіть простий механізм збереження контексту значно розширює можливості нейромережевого керування в динамічних середовищах.

Реалізований генетичний алгоритм з елітарним відбором, пропорційним схрещуванням та стохастичною мутацією продемонстрував свою ефективність як метод оптимізації без градієнта. Система здатна автономно покращувати параметри мережі, не вимагаючи ніякої апіорної інформації про гру. Ключовим результатом стало визначення впливу гіперпараметрів ГА:

Ймовірність мутації: Було знайдено оптимальне значення близько 5%. Відсутність мутацій призводить до передчасної збіжності в локальний оптимум, а надто високий рівень (20%) — до нестабільного, хаотичного навчання.

Розмір популяції: Для даної задачі достатнім виявився розмір популяції в 200 особин. Збільшення до 500 особин дало незначний прирід якості при суттєвому збільшенні часу обчислень, що свідчить про досягнення точки «спадної віддачі».

Стратегія «сімейств»: Паралельне навчання декількох незалежних сімейств дозволило зменшити ризик невдалого експерименту через невдалу ініціалізацію та отримати більш надійну статистику щодо ефективності алгоритму.

Система моніторингу, візуалізації та збереження даних, розроблена в роботі, є потужним інструментом аналізу. Графіки прогресу, таблиці результатів у реальному часі, автоматичне збереження статистики у CSV та серіалізація ваг у бінарних файлах забезпечили повну відтворюваність експериментів та глибокий аналіз динаміки навчання. Це дозволило не лише оцінити кінцевий результат, але й зрозуміти процес його досягнення, виявити проблеми (наприклад, вихід на плато) і корегувати налаштування.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

- 1 Kevin P. Murphy Machine Learning: A Probabilistic Perspective, 2012. 1068 p.
- 2 Гудфеллоу, Йошуа Бенджио, Аарон Курвилль MIT Press: Deep Learning, 2016. 652 p.
- 3 Christopher M. Bishop Pattern Recognition and Machine Learning , 2006. 758 p.
- 4 Митчелл, Томас. Machine Learning, 1997. 421 p.
- 5 Дослідження принципів кодування алгоритмів
URL:<http://neuralnetworksanddeeplearning.com/> (дата звернення 29.04.2025).
- 6 David E. Goldberg Genetic Algorithms in Search, Optimization, and Machine Learning, 1989. 432 p.
- 7 Melanie Mitchell An Introduction to Genetic Algorithms, 1996. 162 p.
- 8 Дослідження та застосування нейронних мереж у відеоіграх
URL:<https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf> (дата звернення 23.04.2025).
- 9 Al Sweigart Invent Your Own Computer Games with Python, 2016. 375 p.
- 10 Alejandro Rodas de Paz Python Game Programming By Example, 2015. 429 p.
- 11 Andreas C. Muller Introduction to Machine Learning with Python, 2017. 392 p.
- 12 Al Sweigart Making Games with Python & Pygame, 2012. 365 p.
- 13 Дослідження нейронних мереж URL:<https://www.kaggle.com/> (дата звернення 16.04.2025).
- 14 Таровик М.Є., Киричек Г.Г. Методи використання генеративно-змагальних нейронних мереж. Тиждень науки–2023. Факультет комп'ютерних наук і технологій. Тези доповідей науково-практичної конференції, Запоріжжя, 24–28 квітня 2023 р. – Запоріжжя: НУ «Запорізька політехніка», 2023. – С. 49–52.
URL: <http://eir.zntu.edu.ua/handle/123456789/12415>

ДОДАТОК А

Лістинг А.1 - Сортуювання всіх змій за критеріями відбору

```
def select_the_best(self, fam, ep, pq):
    self.era_snakes_list.sort(key=lambda i: (-i[0], -i[1]))
    self.era_snakes_list = self.era_snakes_list[:pq]
    self.era_weights_list = []
    self.era_scores_list = []
    self.era_moves_list = []

    for i in self.era_snakes_list:
        self.era_weights_list.append(i[4])
        self.era_scores_list.append(i[0])
        self.era_moves_list.append(i[1])

self.snake_statistics.loc[len(self.snake_statistics.index)] = {
    'ai_format': str(razmer_pole) + 'x' +
str(razmer_pole) + '_' + str(input_layer_quantity) + 'in' +
str(neurons_layer_quantity_1) + '_'
+ str(neurons_layer_quantity_2),
    'family': i[2],
    'epoch': i[3],
    'score': i[0],
    'moves': i[1],
    'desk_size': razmer_pole,
    'sensors': input_layer_quantity,
    'lay_1': neurons_layer_quantity_1,
    'lay_2': neurons_layer_quantity_2,
    'top_parents_by_epoch': parents_quantity,
    'first_random_snakes': random_snakes_quantity,
    'time_label': self.time_label
}
```

```

        self.snake_top_weights.append([i[0], i[1], i[4]])

        self.snake_top_weights.sort(key=lambda i: (-i[0], -i[1]))
        self.snake_top_weights =
self.snake_top_weights[:self.max_best_w]
        self.snake_weights_data.append([fam, ep,
self.era_weights_list, self.era_scores_list, self.era_moves_list])
    def save_game_stat(self):
        file_name = "snake_statistics_" + str(self.time_label) +
".csv"
        try:
            self.snake_statistics.to_csv(file_name, mode='w',
index=False)
        except:
            print('Произошла ошибка при записи статистики в файл.
Данные не сохранены')
    def save_best_weights(self):
        file_name = "snake_best_weights_" + str(self.time_label)
+ ".bin"
        try:
            with open(file_name, "wb") as file:
                pickle.dump(self.snake_weights_data, file)
        except:
            print("Произошла ошибка при записи весов лучших змей
в файл")

```

Лістинг А.2 - Код нейронної мережі

```

import random
import numpy as np

class MemoryPerceptron:

```

```

def __init__(self, in_l_q, lay_1_q, lay_2_q, out_l_q,
memory_decay=0.7):
    self.in_l_q = in_l_q
    self.lay_1_q = lay_1_q
    self.lay_2_q = lay_2_q
    self.out_l_q = out_l_q
    self.memory_decay = memory_decay
    self.memory = np.zeros(in_l_q)
    self.total_in = in_l_q * 2
    self.fitness = 0

    self.new_pct()
    self.rand_w()

def relu(self, x):
    return np.maximum(x, 0)

def new_pct(self):
    """Сброс весов и памяти"""
    self.lay_1 = np.zeros(self.lay_1_q)
    self.lay_2 = np.zeros(self.lay_2_q)
    self.out_l = np.zeros(self.out_l_q)
    self.memory = np.zeros(self.in_l_q)

def rand_w(self):

    self.w_in = np.random.uniform(-1, 1, (self.total_in,
self.lay_1_q))
    self.w_lay = np.random.uniform(-1, 1, (self.lay_1_q,
self.lay_2_q))
    self.w_out = np.random.uniform(-1, 1, (self.lay_2_q,
self.out_l_q))

def get_w(self, rw_in, rw_lay, rw_out):

```

```

        self.w_in = rw_in
        self.w_lay = rw_lay
        self.w_out = rw_out

    def shuff(self, w1, w2, probability_a=70, probability_b=30,
k=0):

        a = w1.ravel()
        b = w2.ravel()
        c = []
        for i in range(len(a)):
            c.append(random.choices([a[i], b[i],
random.uniform(-1, 1)],
                                weights=[probability_a,
probability_b, k])[0])
        return np.array(c).reshape(w1.shape)

    def w_suffle(self, w1, probability_a, probability_b, k):

        deti_list = []

        def unpack_weights(item):
            if isinstance(item, list) and len(item) == 3 and
all(isinstance(x, np.ndarray) for x in item):
                return item
            elif isinstance(item, list) and len(item) == 1:
                return unpack_weights(item[0])
            else:
                raise ValueError("Неверная структура весов!")

        for i in range(len(w1)):
            for j in range(len(w1)):
                try:
                    w1l, w1l, w0l = unpack_weights(w1[i])

```

```

        wi2, wl2, wo2 = unpack_weights(wl[j])
    except ValueError:
        continue

    if w11.shape != wi2.shape or w11.shape !=
w12.shape or w01.shape != wo2.shape:
        continue

    child_in = self.shuff(wi1, wi2, probability_a,
probability_b, k)
    child_lay = self.shuff(wl1, wl2, probability_a,
probability_b, k)
    child_out = self.shuff(w01, wo2, probability_a,
probability_b, k)
    deti_list.append([child_in, child_lay,
child_out])

    child_in = self.shuff(wi1, wi2, probability_a,
probability_b, 0)
    child_lay = self.shuff(wl1, wl2, probability_a,
probability_b, 0)
    child_out = self.shuff(w01, wo2, probability_a,
probability_b, 0)
    deti_list.append([child_in, child_lay,
child_out])

    return deti_list

def calculation(self, inp_1):

    self.memory = self.memory_decay * self.memory + inp_1
    inp_total = np.concatenate([inp_1, self.memory])

```

```

self.lay_1 = self.relu(np.dot(inp_total, self.w_in))
self.lay_2 = self.relu(np.dot(self.lay_1, self.w_lay))
self.out_1 = np.dot(self.lay_2, self.w_out)

def solution(self):

    n = np.argmax(self.out_1)
    sol = [0, 0, 0, 0]
    sol[n] = 1
    return sol

@staticmethod
def next_generation(parents):

    new_pop = []
    for p in parents:
        child = MemoryPerceptron(p.in_l_q, p.lay_1_q,
p.lay_2_q, p.out_l_q)
        child.get_w(p.w_in.copy(), p.w_lay.copy(),
p.w_out.copy())
        new_pop.append(child)
    return new_pop

```

Лістинг А.3 - Функціонування передачі векторів до нейронної мережі

```

import numpy as np
import random

class Mir:

    def __init__(self, size):
        self.size = size # размер игрового поля (кол-во клеток
без стен)

        self.reset()

```

```

def reset(self):
    self.arr = np.zeros((self.size + 2, self.size + 2),
dtype=int)
    self.score = 0
    self.count_moves = 0
    self.count_all_moves = 0
    self.snake = []
    self.snake_dir = (0, 1) # self.generate_walls()
    self.generate_snake()
    self.generate_apple()
    self.snake_sensors = []

def new(self):

    self.reset()

def generate_walls(self):

    self.arr[:, 0] = 11
    self.arr[:, -1] = 11
    self.arr[0, :] = 11
    self.arr[-1, :] = 11

def generate_snake(self):

    mid = self.size // 2 + 1
    self.snake = [(mid, mid)]
    self.arr[mid, mid] = 1

def generate_apple(self):

    empty_cells = [(i, j) for i in range(1, self.size + 1)
                    for j in range(1, self.size + 1)
                    if self.arr[i, j] == 0]

```

```

        if empty_cells:
            self.apple = random.choice(empty_cells)
            self.arr[self.apple] = 7

    def sensors(self):

        head = self.snake[-1]
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1), # ↑ ↓ ←
→
                                (-1, -1), (-1, 1), (1, -1), (1, 1)] # ↖
↗ ↘ ↙

        self.snake_sensors = []

        for d in directions:
            dist_wall = dist_body = dist_apple = 0
            step = 1
            while True:
                pos = (head[0] + d[0]*step, head[1] + d[1]*step)
                cell_value = self.arr[pos]
                if cell_value == 11:
                    dist_wall = 1 / step
                    break
                elif cell_value == 1 and pos != head:
                    dist_body = 1 / step
                elif cell_value == 7:
                    dist_apple = 1 / step
                step += 1
            self.snake_sensors.extend([dist_wall, dist_body,
dist_apple])

    def move(self, direction):

        moves_map = {0: (-1, 0), 1: (1, 0), 2: (0, -1), 3: (0,
1)}

        idx = np.argmax(direction)

```

```

        self.snake_dir = moves_map[idx]
        head = self.snake[-1]
        new_head = (head[0] + self.snake_dir[0], head[1] +
self.snake_dir[1])

```

```

        if self.arr[new_head] in [1, 11]:
            return False

```

```

        if self.arr[new_head] == 7:
            self.score += 1
            self.count_moves = 0
            self.count_all_moves += 1
            self.snake.append(new_head)
            self.arr[new_head] = 1
            self.generate_apple()

```

```

        else:

```

```

            self.count_moves += 1
            self.count_all_moves += 1
            tail = self.snake.pop(0)
            self.arr[tail] = 0
            self.snake.append(new_head)
            self.arr[new_head] = 1

```

```

        return True

```

```

@property

```

```

def snake_head(self):

```

```

    return self.snake[-1]

```

ДОДАТОК Б



Рисунок Б.1 - Привітальний слайд

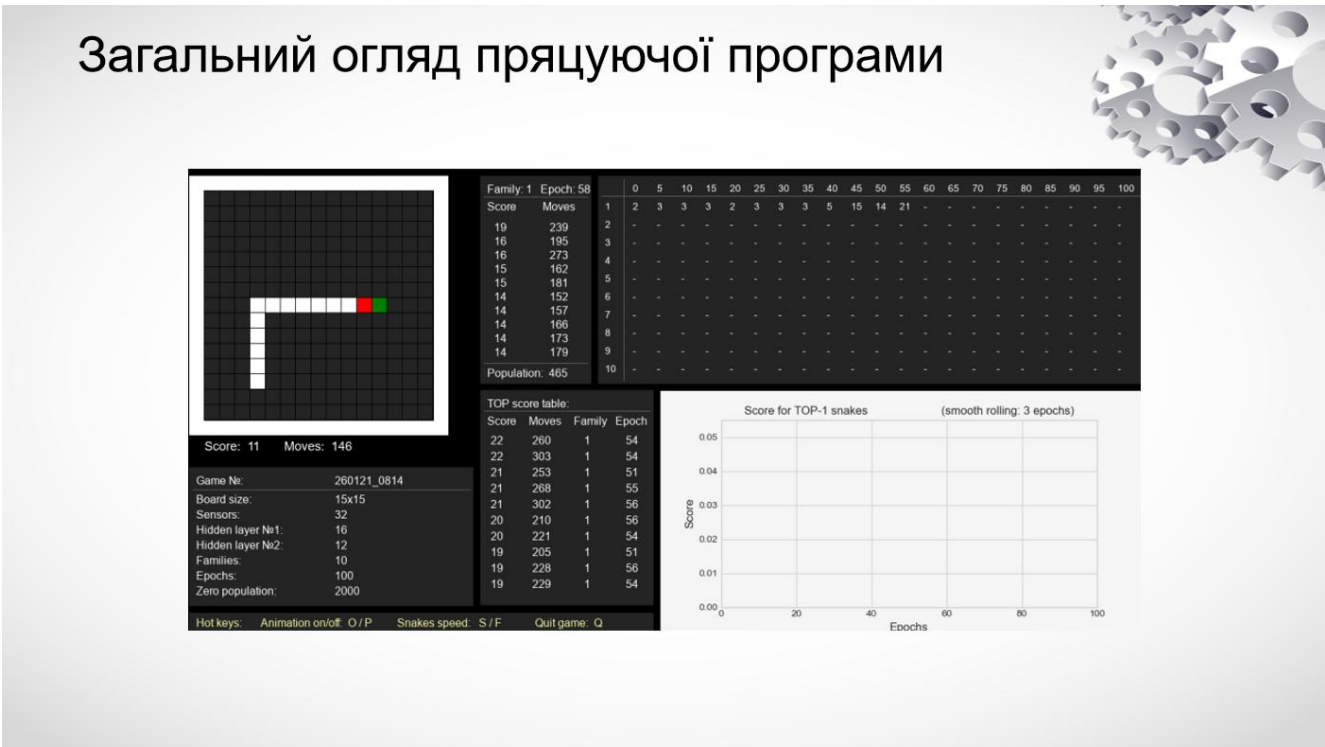


Рисунок Б.2 - Увесь візуальний функціонал програми

Пояснення роботи векторів

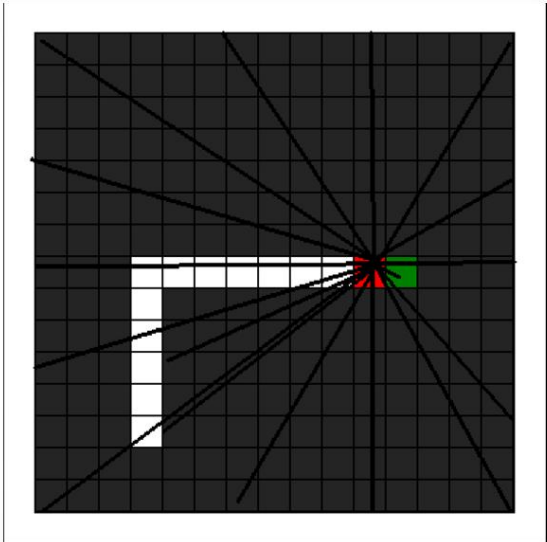


Рисунок Б.3 - Пояснення роботи векторів

Пояснення інформації для контролю навчання

Family: 1 Epoch: 58		0	5
Score	Moves	1	2
19	239	2	-
16	195	3	-
16	273	4	-
15	162	5	-
15	181	6	-
14	152	7	-
14	157	8	-
14	166	9	-
14	173	10	-
14	179		
Population: 465			
TOP score table:			
Score	Moves	Family	Epoch
22	260	1	54
22	303	1	54
21	253	1	51
21	268	1	55
21	302	1	56
20	210	1	56
20	221	1	54
19	205	1	51
19	228	1	56
19	229	1	54

Game №:	260121_0814
Board size:	15x15
Sensors:	32
Hidden layer №1:	16
Hidden layer №2:	12
Families:	10
Epochs:	100
Zero population:	2000

Рисунок Б.4 - Візуалізація інформації навчання

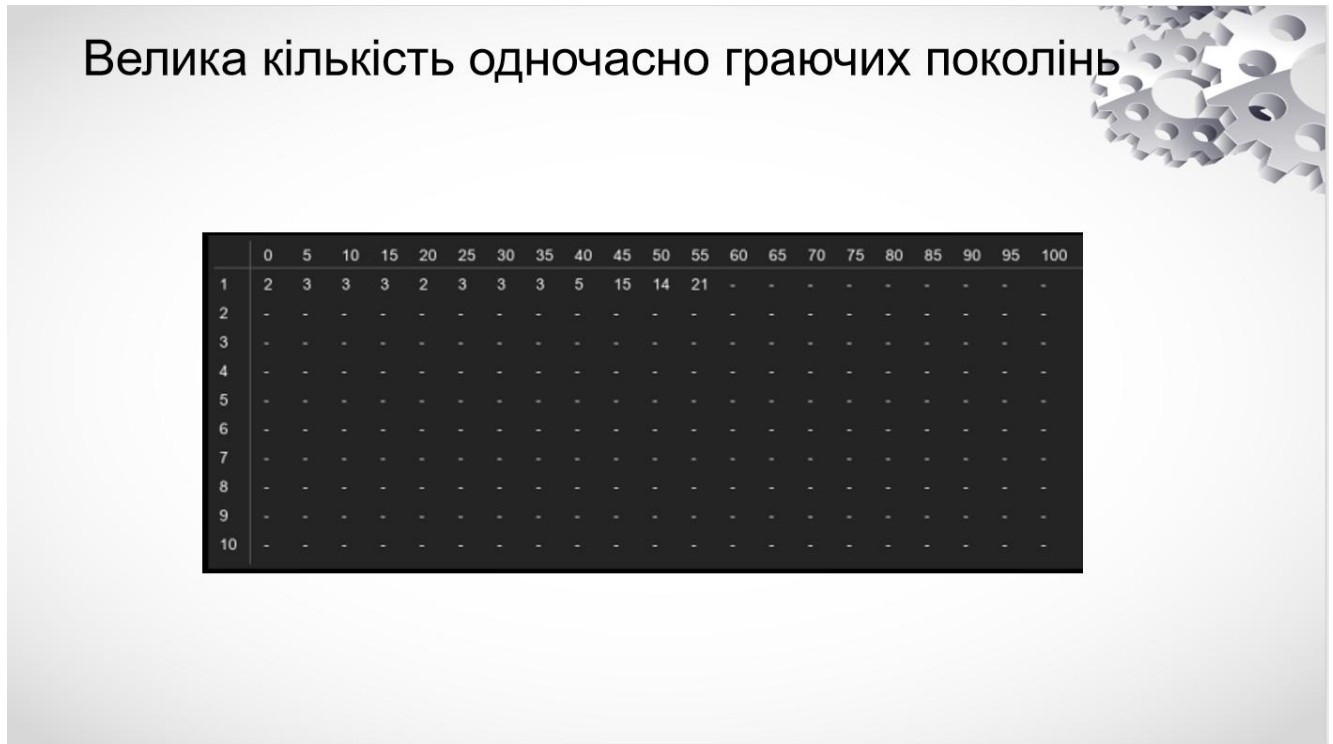


Рисунок Б.5 - Роз'яснення процесу одночасних поколінь



Рисунок Б.6 - Дякую за увагу