

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук та технологій
(повне найменування факультету)

Кафедра комп'ютерних систем та мереж
(повне найменування кафедри)

Пояснювальна записка
до дипломного проєкту (роботи)
бакалавра
(ступінь вищої освіти (освітній ступінь))

на тему РОЗРОБКА КОМП'ЮТЕРНОЇ ГРИ З ВИКОРИСТАННЯМ
ШТУЧНОГО ІНТЕЛЕКТУ

Виконав: студент 4 курсу групи КНТ 511сп
спеціальності _____

123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація) _____
«Комп'ютерна інженерія»

ТАРОВИК М.Є.
(ПРИЗВИЩЕ та ініціали)

Керівник _____
ТЯГУНОВА М.Ю.
(ПРИЗВИЩЕ та ініціали)

Рецензент _____
МАЛИЙ О.Ю.
(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування вищого навчального закладу)

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

ЗАТВЕРДЖУЮ
Завідувач кафедри КУДЕРМЕТОВ Р.К.
“ ” 2024 року

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

Таровик Микита Євгенович
(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка комп'ютерної гри з використанням штучного інтелекту

керівник проєкту (роботи) к. т. н., доцент, Тягунова Марія Юріївна

(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом вищого навчального закладу від “ 1 ” квітня 2024 року № 105

2. Строк подання студентом проєкту (роботи) 25 травня 2024 року

3. Вихідні дані до проєкту (роботи) ігрові механіки, методи штучного інтелекту, такі як генетичний алгоритм та нейронні мережі

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) _____

1. Аналіз предметної області

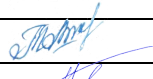
2. Методи штучного інтелекту для розробки гри

3. Реалізація гри

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Тестування гри, Графік результатів кожного сімейства, Таблиця інформації про навчання НМ, Таблиця результатів роботи програми

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	ТЯГУНОВА М.Ю., к.т.н., доцент		
нормоконтроль	ЩЕРБАК Н.В., ст. викл.		

7. Дата видачі завдання 01.03.2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Ознайомлення з завданням	01.03.2024 р.	
2	Збір матеріалу для проєкту	03.03.2024 р.	
3	Вивчення варіантів реалізації та вибір варіанту для розробки на мові Python	12.04.2024 р.	
4	Збір всіх комплектуючих для розробки гри та нейронної мережі	23.04.2024 р.	
5	Розробка гри та нейронної мережі	25.04.2024 р.	
6	Тестування гри та нейронної мережі	25.05.2024 р.	
7	Оформлення графічного матеріалу	28.05.2024 р.	
8	Оформлення допуску до захисту	30.05.2024 р.	

Студент (ка)


(підпис) Микита ТАРОВИК
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)


(підпис) Марія ТЯГУНОВА
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

ПЗ: 61 стр., 10 рис., 15 джерел.

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

Об'єкт розробки – гра на мові Python з використанням нейронної мережі.

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

Проект складається з трьох розділів.

У першому розділі представлено аналіз предметної області, де розглянуто архітектури нейронних мереж, багатошаровий перцептрон, алгоритми навчання нейронних мереж та проблеми навчання агента для гри. Особливу увагу приділено використанню нейронних мереж у іграх, що є актуальним напрямом досліджень. Ця частина роботи розкриває сучасні методи та технології, що використовуються у сфері штучного інтелекту.

Другий розділ присвячено методам штучного інтелекту для розробки гри. У цьому розділі детально розглянуто процеси ініціалізації популяції, фітнес-функції, операції селекції, схрещування та мутації, а також еволюцію популяції.

У третьому розділі описано реалізацію гри. Тут детально розглянуто процес створення гри «Змійка», проєктування нейронної мережі для цієї гри, реалізацію алгоритму та навчання і тестування моделі.

ЗМІСТ

Вступ.....	12
1 Аналіз предметної області.....	12
1.1 Архітектури нейронних мереж.....	13
1.2 Багатошаровий перцептрон.....	15
1.3. Алгоритми навчання нейронних мереж	16
1.4 Проблеми навчання агента для гри	18
1.5 Використання нейронних мереж у іграх	19
2 Методи штучного інтелекту для розробки гри	21
2.1 Ініціалізація популяції	26
2.2 Фітнес-функція	26
2.3 Операції селекції, схрещування та мутації.....	29
2.4 Еволюція популяції.....	33
3 Реалізація гри.....	35
3.1 Створення гри «Змійка»	37
3.2 Проектування нейронної мережі для гри «Змійка».....	40
3.3 Реалізація алгоритму.....	42
3.4 Навчання та тестування моделі	47
Висновки	55
Перелік джерел посилання	56
Додаток А.....	57

**ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ,
СКОРОЧЕНЬ І ТЕРМІНІВ**

НМ	-	Нейронна мережа;
БП	-	Багатошаровий перцептрон;
ГА	-	Генетичний алгоритм;
ГНК	-	Генеративна конкурентна мережа;
CNN	-	Англ. Convolution Neural Networks;
RNN	-	Англ. Recurrent Neural Networks;
LSTM	-	Англ. Long Short-Term Memory;
ADAM	-	Англ. Adaptive Moment Estimation;
RMSP	-	Англ. Root Mean Square Propagation;
NPC	-	Англ. Non-Playable Character.

ВСТУП

Штучний інтелект в сучасному світі та займає одне з провідних місць серед наукових досліджень та інноваційних впроваджень. Використання нейронних мереж та генетичних алгоритмів у різних сферах, від медицини до ігрового бізнесу, підтверджують їхній високий потенціал та ефективність у вирішенні складних завдань. При тому можливість вирішення проблеми складністю нашої змійки її можливості використовувати алгоритми машинного навчання та оптимізацію. В грі зі змійкою користувачеві необхідно швидко приймати рішення, добре орієнтуватися та ефективно управляти віртуальною постаттю у посуваючійся ігровій зоні. Тут будуть функціонувати нейронні мережі, що навчалися на вхідних даних, а також генетичні алгоритми, що оптимізують параметри мережі для досягнення кращих результатів.

Метою роботи - розробка ефективної системи навчання гри «Змійка» за допомогою нейронних мереж, оптимізованих генетичними алгоритмами. Для досягнення поставленої мети необхідно виконати такі завдання:

- проаналізувати предметну область;
- розробити модель нейронної мережі;
- розробити генетичний алгоритм для оптимізації параметрів нейронної мережі з метою підвищення її продуктивності;
- реалізувати запропоновані методи у грі;
- проаналізувати результати.

Методи використані у ході роботи:

- аналіз наукової літератури: для вивчення теоретичних основ нейронних мереж та генетичних алгоритмів, їхнього застосування у різних галузях, зокрема в ігровій індустрії;

- розробка алгоритмів: створення та тестування алгоритмів для навчання нейронної мережі гри "Змійка" та їх оптимізації за допомогою генетичних алгоритмів;
- програмування: реалізація розроблених алгоритмів у програмному середовищі з використанням мов програмування та спеціалізованих бібліотек;
- статистичний аналіз: обробка та інтерпретація результатів експериментів, порівняння з існуючими методами та моделями.

Таким чином, ця робота спрямована на вивчення можливостей покращення проходження гри «Змійка» та знаходження оптимального рішення за допомогою штучного інтелекту. Результати цієї роботи можуть бути корисними для подальшого розвитку інтелектуальних систем управління, не тільки у сфері комп'ютерних ігор, але й у ширшому контексті.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Нейронні мережі, основа сучасних методів штучного інтелекту, виникли в середині 20 століття. Перша спроба змодельовати процеси мислення за допомогою математичних структур була зроблена в 1943 році Уорреном Маккалохом і Волтером Піттсом [1]. Вони запропонували модель штучного нейрона, здатного виконувати логічні операції. Ця модель заклала основу для подальших досліджень у галузі штучного інтелекту.

У 1958 році Френк Розенблатт розробив перцептрон, першу нейронну мережу, яка могла навчатися. Перцептрон складався з одного шару нейронів і використовувався для вирішення простих задач класифікації. Однак її обмеження, такі як нездатність вирішувати нелінійні проблеми, призвели до тимчасового зниження інтересу до нейронних мереж у 1970-х роках [1].

У 1980-х роках інтерес до нейронних мереж відродився завдяки роботам Джона Хопфілда та Джеффри Гінтона. Хопфілд запропонував модель рекурентної нейронної мережі, а Гінтон і його колеги розробили алгоритм зворотного поширення помилок, який дозволив навчати багат шарові нейронні мережі [2]. Ці досягнення започаткували нову еру в дослідженнях штучного інтелекту.

Сьогодні нейронні мережі використовуються в багатьох сферах: від розпізнавання образів і мови до медичної діагностики та автономного водіння. Глибокі нейронні мережі (deep learning) дозволяють обробляти великі обсяги даних і знаходити складні закономірності, що сприяє розвитку технологій штучного інтелекту.

Не менш цікава історія гри «Змійка». Вперше вона з'явилася в 1976 році як аркадна гра Blockade, розроблена Gremlin. Гра передбачала управління змією, яка рухалася по екрану, збільшуючись після збору їжі, при цьому гравець повинен був уникати зіткнень зі стінами та власним тілом.

У 1997 році «Змійка» набула всесвітньої популярності завдяки включенню в мобільні телефони Nokia. Ця версія гри стала справжнім хітом, закріпивши за собою статус класики в історії відеоігор. З тих пір «Змійка» неодноразово переосмислювалася і відтворювалася на різних платформах, залишаючись популярною серед гравців різних поколінь.

Сучасні нейронні мережі та гра «Змійка» продовжують розвиватися, взаємодіючи в нових, інноваційних проектах. Нейронні мережі використовуються для створення більш складних ігор, які адаптуються до стилю гравця, і для розробки систем, які дозволяють комп'ютеру вчитися грати в ігри автономно. Використання нейронних мереж у грі «Змійка» є яскравим прикладом використання штучного інтелекту для автоматизації ігрового процесу, демонструючи потенціал таких технологій у розробці інтерактивних та адаптивних розважальних систем.

1.1 Архітектури нейронних мереж

Розглянемо основні типи архітектур нейронних мереж:

- одношаровий персептрон;
- багатошаровий персептрон;
- згорткові нейронні мережі;
- повторювані нейронні мережі;
- генеративна конкурентна мережа;
- transformer.

Найпростіша форма нейронної мережі, що складається з одного шару нейронів [3]. Кожен нейрон отримує певну кількість сигналів як вхідні дані та зважує та підсумовує ці сигнали. Вихідний сигнал визначається на основі

підсумованого сигналу та функції активації. Одношарові персептрони можуть вирішувати лише задачі з лінійним розділенням, що обмежує їх застосування.

Спеціалізуються на обробці просторово структурованих даних, таких як зображення та відео. CNN складається зі згорткових шарів, сумацийних шарів (об'єднуючих шарів) і повністю зв'язаних шарів. Згорткові шари використовують фільтри для виділення об'єктів на різних рівнях абстракції, що дозволяє ефективно ідентифікувати об'єкти на зображеннях. CNN широко використовується в комп'ютерному зорі, медичній діагностиці та інших галузях [1].

Мають зворотний зв'язок і можуть обробляти послідовні дані, такі як текст, аудіо або часові ряди. RNN можуть зберігати інформацію про попередні стани, що робить їх ефективними в задачах прогнозування та аналізу послідовності [2]. Однак стандартні RNN мають труднощі з обробкою довгострокових залежностей. Для вирішення цієї проблеми були розроблені моделі LSTM.

Складається з двох нейронних мереж: генератора та дискримінатора, які конкурують між собою. Генератор створює нові вибірки даних, а дискримінатор намагається відрізнити реальні вибірки від згенерованих. ГГН використовуються для створення реалістичних зображень, відео, тексту та інших матеріалів [3]. Вони мають потенційне застосування в творчих галузях, дизайні та розробці нових продуктів.

Складається з кількох шарів нейронів: вхідного шару, одного або кількох прихованих шарів і вихідного шару [3]. Приховані шари дозволяють моделювати складні нелінійні залежності. Алгоритм зворотного поширення помилок, запропонований у 1986 році, дозволяє ефективно навчати багатошарові персептрони. БП використовується для розпізнавання образів, класифікації та інших завдань, які вимагають високої точності [1].

Був запропонований у 2017 році та став революційною архітектурою для обробки послідовностей даних. Вони використовують механізм уваги, який дозволяє враховувати залежності між елементами послідовності незалежно від їх відстані. Transformer продемонстрував високу продуктивність у таких завданнях природної мови, як машинний переклад, аналіз тексту та генерування мовлення.

1.2 Багатошаровий перцептрон

Багатошаровий перцептрон (рис. 1.1) одна з найбільш ранніх моделей штучних нейронних мереж, розроблених для вирішення проблем класифікації. Модель виникла на початку 1960-х як спроба математичного моделювання людського сприйняття та розпізнавання образів [2]. Архітектура перцептрона складається з набору вхідних нейронів (або датчиків (рис. 1.2)), підключених до вихідного нейрона (або класифікатора).

Основна ідея багатошарового перцептрона полягає в тому, щоб мати можливість ідентифікувати складні нелінійні залежності в даних. Він складається з кількох шарів нейронів, кожен шар з'єднаний з наступним шаром за допомогою ваг. Вхідні дані надходять на перший рівень, де кожен нейрон обчислює зважену суму вхідних значень, а потім передає її через функцію активації [4].

Цей процес триває на кожному рівні мережі, і кінцевим результатом є остаточне призначення або класифікація. Навчання багатошарових перцептронів зазвичай виконується за допомогою алгоритму зворотного поширення, який передбачає коригування вагових коефіцієнтів мережі на основі розміру помилки у вихідних даних.

Важливим аспектом багатошарового перцептрона є його здатність автоматично вивчати складні візерунки в матеріалах, що робить його потужним інструментом для завдань класифікації та прогнозування [3]. Однак він також має деякі обмеження, наприклад, інші архітектури нейронної мережі можуть бути більш ефективними при обробці неструктурованих даних або великих обсягів даних.

Таким чином, багатошарові перцептрони є важливим кроком у розвитку штучних нейронних мереж, що дозволяє моделювати складні залежності з високою точністю та ефективністю в багатьох сучасних задачах обробки інформації.

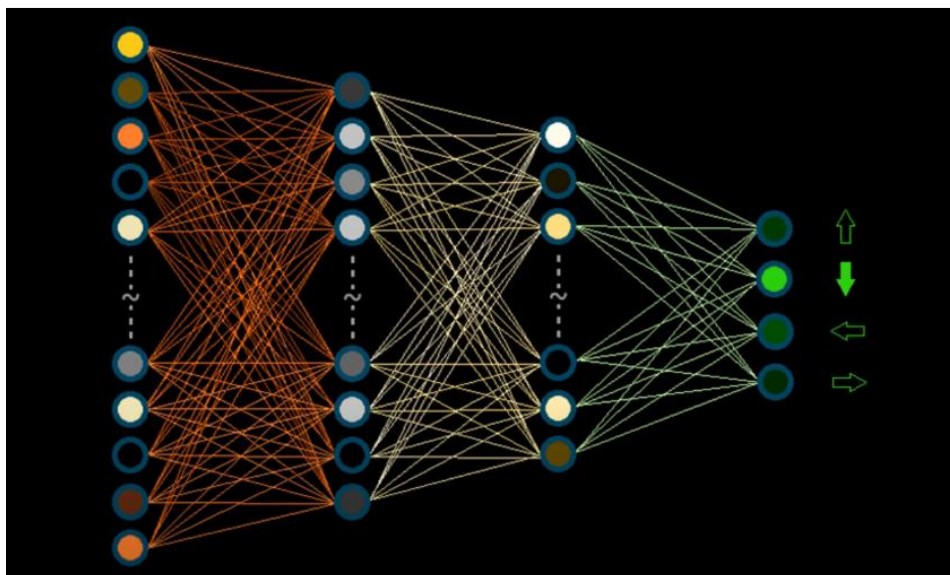


Рисунок 1.1 – Багатошаровий перцептрон з 2 прихованими шарами

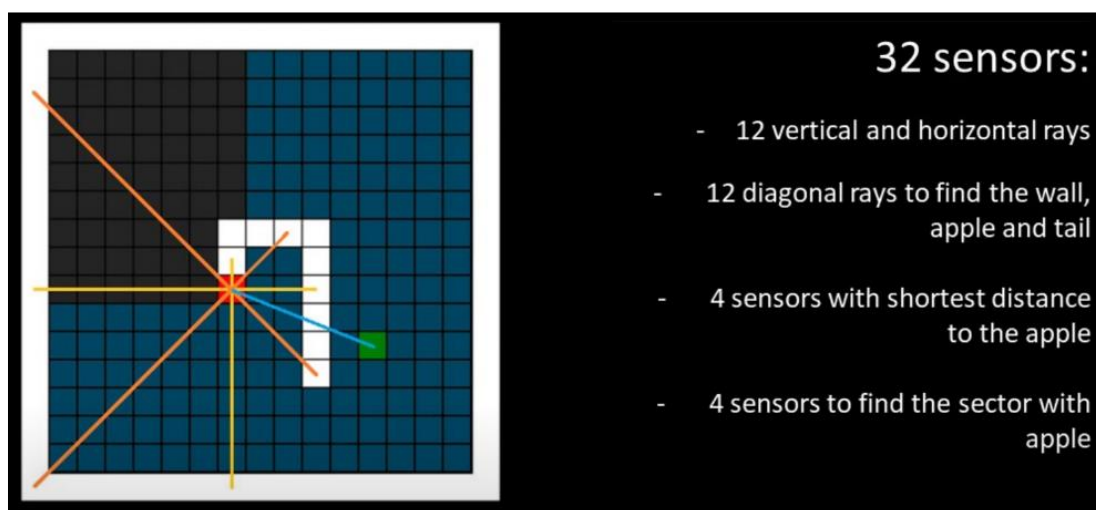


Рисунок 1.2 – Відображення прихованих векторів

1.3 Алгоритми навчання нейронних мереж

Алгоритми навчання нейронних мереж є ключовим елементом їх успішної роботи, включаючи різні методи оптимізації параметрів мережі для досягнення високої точності вирішення завдань.

Базові алгоритми навчання:

- алгоритм зворотного поширення помилок;
- стохастичний градієнтний спуск;
- метод моментів;
- адаптивні методи оптимізації;
- методи регуляризації;
- генетичний алгоритм.

Це базовий метод оптимізації, який використовується в поєднанні з алгоритмом зворотного поширення помилок [5]. Він використовує невеликі підвибірки даних для оцінки градієнта функції втрат і оновлення параметрів мережі.

Цей алгоритм найчастіше використовується для навчання багаторівневих нейронних мереж. Він заснований на ідеї використання градієнтного спуску для оновлення ваг нейронів у мережі [4]. Цей процес передбачає обчислення похідних втрат на основі вихідних значень мережі, а потім зворотне розповсюдження цих похідних через мережу для коригування ваги кожного рівня.

Цей метод використовується для прискорення процесу оптимізації. Він накопичує історію оновлень параметрів і використовує її для розрахунку наступного кроку, забезпечуючи швидшу конвергенцію оптимального рішення [5].

Включаючи такі алгоритми, як ADAM (Adaptive Moment Estimation), RMSP (Root Mean Square Propagation) та інші алгоритми, які регулюють швидкість навчання кожного параметра на основі історії градієнта [2], [4]. Для складних завдань навчання нейронної мережі ці методи часто ефективніші, ніж традиційні SGD.

Такі як регуляризація L1 і L2, випадання та пакетна нормалізація, щоб запобігти перетренуванню. Їх призначення — контролювати розмір параметрів мережі та підвищувати її масштабованість [3].

Використовує принципи природного відбору та еволюції для навчання нейронних мереж [6]. Спочатку створюється початкова популяція індивідумів (наприклад, набір параметрів мережі). Кожна особа оцінюється за допомогою

функції фітнесу, яка представляє її ефективність на основі даних тренування. Наступні кроки включають вибір найкращої особини, схрещування її параметрів для створення нових особин і зміну параметрів для забезпечення різноманітності. Процес повторюється для кількох ітерацій або до досягнення певного критерію зупинки [7]. Генетичні алгоритми дозволяють ефективно досліджувати великі простори параметрів і знаходити оптимальні конфігурації нейронних мереж, особливо в складних задачах, де інші методи оптимізації можуть бути неефективними.

1.4 Проблеми навчання агента для гри

Під час навчання агента за допомогою генетичних алгоритмів може виникнути кілька проблем:

- проблема розміру простору параметрів;
- високі вимоги до обчислень;
- складність передачі знань;
- необхідно налаштувати параметри алгоритму.

Простори параметрів можуть бути великими та складними, особливо в складних ігрових середовищах. Генетичні алгоритми можуть потребувати значних обчислювальних ресурсів для ефективного дослідження різних комбінацій параметрів. Для вирішення цієї задачі ми обрали просту гру «Змійка».

Оцінка функції придатності (міри продуктивності агента) може бути дуже дорогою з точки зору часу та обчислювальних ресурсів, особливо якщо гра займає багато часу кожен раунд.

Генетичні алгоритми вимагають відповідного налаштування таких параметрів, як розмір популяції, схрещування та ймовірність мутації, що може бути складним завданням і потребує досвіду в оптимізації алгоритмів.

Після перерахування можливих проблем я хотів би звернути увагу на той факт, що, незважаючи на низку недоліків, генетичні алгоритми ефективні для навчання агентів в ігровому середовищі, особливо коли потрібно досліджувати великі простори параметрів. Для досягнення успішних результатів важливо враховувати специфіку конкретного завдання та адаптувати алгоритм до його потреб.

У генетичних алгоритмах передача знань між поколіннями людей може бути складною. Покоління з корисними адаптаціями можуть втратити ці корисні ознаки під час гібридизації та мутації.

1.5 Використання нейронних мереж у іграх

Використання нейронних мереж у комп'ютерних іграх не рідкість, і існує безліч програм, які можуть покращити ігровий процес.

У цьому розділі розглядаються основні методи використання нейронних мереж у комп'ютерних іграх:

- штучний інтелект NPC;
- створення контенту;
- аналіз поведінки гравців;
- покращення візуальної та реалістичної графіки;
- автоматизація тестування гри;
- персоналізація ігрового процесу.

Нейронні мережі використовуються для моделювання поведінки комп'ютерних персонажів. Вони дозволяють NPC адаптуватися до мінливих умов гри, розуміти стратегії гравців і приймати обґрунтовані рішення на основі отриманих даних [8]. Наприклад, NPC можуть використовувати нейронні мережі для визначення найкращого шляху досягнення цілей, аналізу бойових ситуацій і прийняття рішень щодо використання зброї чи захисту.

Нейронні мережі використовуються для створення нових рівнів, карт, персонажів і навіть внутрішньоігрових історій [8]. Вони можуть створювати різноманітний контент, щоб надавати гравцям унікальний ігровий досвід щоразу, коли вони грають.

Нейронні мережі використовуються для аналізу та прогнозування поведінки гравців у багатокористувацьких іграх [8]. Вони можуть розрізняти стилі гри, визначати відмінності в підходах і допомагати розробникам покращити баланс ігрових систем.

Нейронні мережі використовуються для поліпшення якості графіки в іграх. Вони можуть покращити текстуру, освітлення та рендеринг сцени, забезпечуючи більш реалістичну графіку.

Нейронні мережі використовуються для автоматизації тестування гри шляхом моделювання та аналізу поведінки гравців і NPC [8]. Вони дозволяють вам знаходити помилки у вашій грі та виправляти їх до випуску.

Нейронні мережі використовуються для персоналізації ігрового досвіду, адаптації ігрового процесу до особистих уподобань і стилю гравця [8].

У цій роботі більше уваги приділяється тому, як штучний інтелект вплине на тестування гри. Після завершення розробки гри потрібно тестувати за допомогою гравців, але на відміну від НМ, можливості людини обмежені, що не дозволяє провести швидко та всестороннє тестувати.

Наприклад: був проведений експеримент, у якому НМ помістили в ігрове середовище, розділивши їх на 2 групи - першій групі потрібно було приховатися, другій групі потрібно було знайти першу, щоб їх не знайшли, перша група повинна була приховати всі предмети в грі, щоб команда дослідників не могла проникнути в їх приміщення, якщо б люди грали за другу команду, вони були б дуже розчаровані, тому що розробники гри самі знали, що це була остання ситуація, з якої не було виходу, але НМ все одно грав за і після сотень невдалих ігор, він навчився використовувати недоліки гри а саме проникати через стіни та перемагати, таким чином можна зробити висновок, що НМ знаходять нові нестандартні рішення, яких інші не можуть досягти.

2 МЕТОДИ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ РОЗРОБКИ ГРИ

Генетичні алгоритми — це метод оптимізації, натхненний природними еволюційними процесами, такими як відбір, кросинговер і мутація. Він широко використовується для вирішення складних задач оптимізації в різних галузях науки і техніки [6].

Процес генетичного алгоритму припиняється, коли досягається умова зупинки. Найкращий індивід в кінцевій популяції вважається найкращим вирішенням проблеми.

Генетичний алгоритм потужний інструментом для вирішення складних задач оптимізації, коли традиційні методи можуть бути неефективними [7]. Це дозволяє ефективно досліджувати велику кількість параметрів і знаходити глобальний оптимум для еволюційних операцій, таких як відбір, кросинговер і мутація.

Генетичні алгоритми - частина ширшого еволюційного алгоритму, який використовує принципи природного відбору та генетики для пошуку оптимальних або майже оптимальних рішень складних проблем [7]. Окрім уже описаних основних функцій, існують інші аспекти та методи, які розширюють можливості Га та підвищують її ефективність розглянемо нижче:

- елітизм;
- адаптивні генетичні алгоритми;
- оптимізація за Парето;
- спільні генетичні алгоритми;
- доповнення інформацією;
- змішування.

Це техніка, яка забезпечує передачу найкращих рішень із покоління в покоління. Класичні генетичні алгоритми ризикують втратити найкращих особин під час процесів відбору, гібридизації та мутації [6]. Елітарність уникає цієї

проблеми, гарантуючи, що кілька найкращих людей залишаються незмінними в наступному поколінні.

Можуть динамічно змінювати свої параметри, такі як ймовірність мутації та схрещування, залежно від прогресу оптимізації. Наприклад, якщо алгоритм починає зближуватися до локального максимуму, ймовірність мутації може бути збільшена для дослідження нових областей простору рішень [7].

У багатьох практичних задачах необхідно оптимізувати декілька параметрів одночасно. Багатоцільові генетичні алгоритми дозволяють розглядати кілька цілей за допомогою таких методів, як оптимальність за Парето. Це призводить до низки компромісів, які утворюють так званий фронт Парето [6].

Використовують кілька популяцій, які взаємодіють одна з одною, щоб покращити процес оптимізації. Конкурентні алгоритми можуть включати аспекти конкуренції між групами, у яких кращі групи переважають гірші. Це допомагає краще вивчити простір рішень і уникнути передчасної конвергенції.

Генетичні алгоритми можуть бути доповнені інформацією про завдання, що може значно підвищити їх ефективність. Наприклад, під час оптимізації механічної конструкції можна використовувати знання про фізичні властивості та структурні обмеження матеріалу, що може зменшити розмір простору пошуку та швидше знайти кращі рішення [6].

Змішування генетичного алгоритму з іншими методами оптимізації, такими як градієнтні методи або алгоритми роїв частинок, може покращити його продуктивність. Гібридні алгоритми використовують кожен метод для досягнення швидшої та точнішої оптимізації.

Генетичні алгоритми використовуються в багатьох великих галузях, а саме: робототехніці, енергетиці та фінансах, обидві з яких мають позитивний вплив на обробку та аналіз геномів і прогнозування результатів. Генетичні алгоритми використовуються в робототехніці для створення алгоритмів, які оптимізують траєкторії руху або розробки та налаштування циклів керування та навігації автономних роботів.

2.1 Ініціалізація популяції

Ініціалізація популяції є одним з перших і найважливіших кроків у роботі генетичного алгоритму, оскільки від правильного вибору вихідного рішення може залежати швидкість і ефективність пошуку оптимального рішення. Детально розглянемо різні аспекти та методи ініціалізації популяції в генетичних алгоритмах.

Базові знання ініціалізації групи:

- популяція;
- чисельність населення.

Популяція в генетичному алгоритмі складається з особин, кожна з яких представляє можливе вирішення проблеми. Індивідууми кодуються у вигляді хромосом, які є послідовностями символів (двійкових, дійсних, порядкових тощо), які відповідають параметрам задачі [6].

Вибір розміру групи важливий для продуктивності ГА. Занадто малий розмір популяції може призвести до недостатньої різноманітності та збільшити ризик передчасної конвергенції до локального оптимуму. Більші розміри популяції збільшують різноманітність, але з більшими обчислювальними витратами [6].

Метод ініціалізації заповнення:

- випадкова ініціалізація;
- ініціалізація сітки;
- ліверізація;
- ініціалізація на основі знань;
- евристика;
- використання попередніх рішень;
- змішаний метод ініціалізації.

Найпростіший і найчастіше використовуваний метод ініціалізації групи [7]. Особи створюються випадковим чином у межах, дозволених параметрами задачі.

Він передбачає велику різноманітність, але не враховує специфіку завдання, що позначається на ефективності пошуку.

Цей метод ділить простір параметрів на однакові категорії та генерує індивідуумів у кожній точці результуючої сітки [5]. Ініціалізація сітки забезпечує рівномірний розподіл вихідних рішень по всьому простору параметрів, але може бути обчислювально дорогим для великих просторів.

Узагальнення ініціалізації багатовимірної просторової сітки. Це забезпечує рівномірне покриття простору параметрів, надаючи окремих осіб у кожному гіперкубі [5];

Іноді ви можете використовувати попередні знання про проблему, щоб створити більш продуктивну початкову сукупність [7].

Може бути застосована для отримання індивідуумів з вищими показниками фізичної підготовки. Це може значно прискорити процес конвергенції [6].

Популяцію можна ініціалізувати на основі рішень, отриманих за допомогою інших методів оптимізації, або використовуючи результати попередніх запусків ГА. [6] Це дозволяє почати пошуки в більш ефективній області простору рішення.

Поєднує випадкову ініціалізацію з евристичними або іншими алгоритмами оптимізації для створення більш ефективних початкових популяцій. Наприклад, одну частину сукупності можна ініціалізувати випадковим чином, а іншу частину можна ініціалізувати за допомогою припущень або попередніх знань [6].

Забезпечення різноманітності - різноманітність популяції є важливим фактором успіху генетичних алгоритмів. Якщо всі особини в початковій популяції дуже схожі, алгоритм може швидко сходитися до локального оптимуму.

Для забезпечення різноманітності можна використовувати такі методи:

- віддалені початкові рішення;
- ранні випадкові мутації.

Індивідууми моделюються таким чином, щоб забезпечити максимальну різноманітність у просторі рішень. Цього можна досягти за допомогою спеціального методу вибору початкової точки [7].

Випадкові мутації додаються до хромосом на ранніх стадіях для збільшення різноманітності [7].

Проблеми і виклики ініціалізації:

- чисельність населення;
- вибір методу ініціалізації.

Вибір відповідного методу ініціалізації залежить від конкретного завдання. Випадкова ініціалізація підходить для загальних випадків, але може бути недостатньою для складних задач із великим простором параметрів.

Надто великий розмір популяції призводить до вищих обчислювальних витрат, тоді як надто малий розмір популяції обмежує різноманітність і збільшує ризик передчасної конвергенції.

Приклади групової ініціалізації для різних функцій:

- ініціалізація задачі оптимізації шляху (TSP);
- розпочати завдання з проектування інженерних систем

Для проблеми комівояжера (TSP) популяцію можна ініціалізувати на основі випадкової перестановки списку міст. Можна також використовувати евристики, такі як найближчий сусід, щоб створити деякі початкові рішення [6].

Під час оптимізації інженерних конструкцій популяцію можна ініціалізувати на основі параметрів або емпіричних даних, отриманих із попередніх проектів, дозволяючи врахувати фізичні властивості матеріалу та проектні обмеження [6].

Ініціалізація популяції є важливим етапом у роботі генетичних алгоритмів, від якого значною мірою залежить ефективність і продуктивність пошуку оптимальних рішень. Використання правильного методу ініціалізації, забезпечення достатньої різноманітності та адаптація до специфіки проблеми можуть значно підвищити продуктивність генетичного алгоритму.

2.2 Фітнес-функція

Фітнес-функція є одним із ключових компонентів генетичних алгоритмів. Він визначає здатність людини розв'язувати конкретне завдання. Правильне визначення та використання функцій пристосованості є вирішальними для успіху генетичних алгоритмів. Давайте детальніше розглянемо аспекти, пов'язані з функціями фітнесу.

Базові знання фітнес-функцій - функція відповідності оцінює кожного індивіда в популяції та присвоює йому числове значення (бал), яке відображає якість цього індивіду щодо проблеми оптимізації. Залежно від характеру проблеми метою фітнес-функції може бути мінімізація або максимізація [5].

Вимоги до фітнес-функції:

- чіткість і недвозначність;
- ефективність обчислення;
- безперервність;
- різноманітність.

Робота має бути чітко визначена та недвозначна для всіх потенційних співробітників.

Функція має бути швидкою для обчислення, оскільки вона буде викликатися кілька разів для оцінки кожної особи в кожному поколінні.

Функція повинна мати плавний ландшафт без різких стрибків, що допомагає алгоритму уникнути передчасної конвергенції.

Завдання повинні заохочувати генетичне різноманіття, щоб уникнути потрапляння в локальні оптимуми.

Приклади фітнес-завдань:

- одновимірні функції;
- для задачі мінімізації;
- багатовимірні функції;
- задачі з обмеженнями;

- нормалізація працездатності;
- завдання відносної та абсолютної придатності.

Для задач оптимізації, де потрібно знайти мінімум або максимум одновимірної функції, фітнес-функція може бути простою.

$$f(x) = x^2, \quad (2.1)$$

$$f(x) = x^2 \quad f(x) = x^2 \quad f(x) = x^2, \quad (2.2)$$

індивіди з меншими значеннями x будуть мати вищий фітнес.

Для більш складних задач, де рішення представлені в багатовимірному просторі.

$$f(x, y, z) = x^2 + y^2 + z^2, \quad (2.3)$$

У задачах з обмеженнями фітнес-функція може включати штрафи за порушення обмежень.

$$f(x) = g(x) + \text{penalty}(x), \quad (2.4)$$

де $g(x)$ - основна функція.

Узагальнення функції пристосованості зазвичай використовується для забезпечення стійкості генетичних алгоритмів. Це може включати масштабування або переміщення значень фітнес-функції, щоб уникнути надто великих або занадто малих значень [7].

Існує два основних методи визначення функції пристосованості: відносні та абсолютні методи.

Повні фітнес-функції:

- функція абсолютної придатності;
- функція відносної пристосованості.

Оцінює кожну людину незалежно від інших осіб. Це класичний підхід, коли кожне рішення оцінюється функцією.

Оцінює, наскільки добре індивід працює порівняно з іншими членами популяції [7]. Наприклад, під час відбору на турніри гравці ранжуються на основі їх результатів у очних матчах.

Проблеми та виклики:

- локальний оптимум;
- обчислювальна складність;
- шум у фітнес-функції.

Функція відповідності може мати кілька локальних оптимальних значень, тому важко знайти глобальне оптимальне значення. Це одна з головних проблем генетичних алгоритмів, яка вирішується такими методами, як адаптивна ймовірність мутації.

У деяких задачах обчислення відповідної функції може зайняти багато часу. Це вимагає ефективних алгоритмів або наближень для оцінки придатності.

Може призвести до нестабільності генетичного алгоритму. Методи усереднення або фільтрації можуть допомогти зменшити вплив шуму.

Приклади застосування:

- оптимізація інженерної конструкції;
- біоінформатика;
- робототехніка;

У задачах вирівнювання послідовності ДНК функції відповідності можуть враховувати подібність послідовності, наявність розривів та інші біологічні фактори.

Фітнес-функції можуть оцінювати міцність, вагу та вартість конструкції, забезпечуючи збалансований підхід до проектування.

Фітнес-функція може оцінювати енергоефективність траєкторії робота, точність його рухів і виконання завдань.

Фітнес-функція є важливою частиною генетичного алгоритму, яка визначає якість розуміння та напрямок пошуку. Він має бути чітко визначеним, обчислювально ефективним і враховувати специфіку завдання. Використання відповідних методів нормалізації, адаптивних методів і обробки обмежень може значно підвищити ефективність генетичних алгоритмів і забезпечити досягнення глобальних оптимальних рішень.

2.3 Операції селекції, схрещування та мутації

Генетичний алгоритм - потужний інструмент оптимізації, заснований на принципі еволюції шляхом природного відбору. Основними операціями, що забезпечують еволюцію популяцій, є добір, кросинговер і мутація. Кожна з цих операцій відіграє важливу роль у генетичних алгоритмах і має свої особливості та методи реалізації [6].

Операція селекції - визначає, які особини з поточної популяції будуть використані для створення наступного покоління. Мета селекції полягає в тому, щоб забезпечити більші шанси на відтворення більш підготовлених особин.

Відбір рулетки (лістинг 2.1) шанси індивіда бути обраним залежать від його фітнесу. Особи з вищою фізичною підготовкою мають вищі шанси бути обраними.

Лістинг 2.1 – Алгоритм роботи рулеткового відбору

```
total_fitness = sum(fitness(individual) for individual in
population)
pick = random.uniform(0, total_fitness)
current = 0
for individual in population:
    current += fitness(individual)
```

```

if current > pick:
    return individual

```

Турнірний відбір (лістинг 2.2) група індивідів випадково вибирається з популяції, і кращий з них обирається для відтворення.

Лістинг 2.2 – Алгоритм роботи турнірного відбору

```

tournament = random.sample(population, k=tournament_size)
best_individual = max(tournament, key=fitness)
return best_individual

```

Ранговий відбір (лістинг 2.3) індивіди сортуються за фітнесом, і ймовірність відбору визначається їхнім рангом.

Лістинг 2.3 – Алгоритм роботи рангового відбору

```

sorted_population = sorted(population, key=fitness)
total_rank = sum(range(1, len(population) + 1))
pick = random.uniform(0, total_rank)
current = 0
for i, individual in enumerate(sorted_population):
    current += (i + 1)
    if current > pick:
        return individual

```

Стабілізуючий відбір - застосовується для збереження різноманітності в популяції, відбираючи індивідів із середнім фітнесом.

Операції схрещування комбінує гени двох батьківських індивідів для створення одного або більше нащадків. Мета кросоверу – забезпечити рекомбінацію генетичної інформації, що сприяє пошуку нових рішень.

Одноточкове схрещування (лістинг 2.4) одна точка розділу обирається випадково, і частини хромосом обох батьків комбінуються.

Лістинг 2.4 – Алгоритм роботи одноточкового схрещування

```

crossover_point = random.randint(1, len(parent1) - 1)
offspring1      =      parent1[:crossover_point]      +
parent2[crossover_point:]
offspring2      =      parent2[:crossover_point]      +
parent1[crossover_point:]

```

Двоточкове схрещування (лістинг 2.5) обираються дві точки розділу, і середні частини хромосом батьків обмінюються.

Лістинг 2.5 – Алгоритм роботи двоточкового схрещування

```

point1, point2 = sorted(random.sample(range(len(parent1)), 2))
offspring1     =  parent1[:point1]   +  parent2[point1:point2]   +
parent1[point2:]
offspring2     =  parent2[:point1]   +  parent1[point1:point2]   +
parent2[point2:]

```

Рівномірне схрещування (лістинг 2.6) кожен ген індивіда випадково обирається з одного з батьків.

Лістинг 2.6 – Алгоритм роботи рівномірного схрещування

```

offspring = []
for i in range(len(parent1)):
    if random.random() < 0.5:
        offspring.append(parent1[i])
    else:
        offspring.append(parent2[i])

```

Арифметичне схрещування (лістинг 2.7) використовується для числових хромосом, комбінуючи гени зваженим середнім.

Лістинг 2.7 – Алгоритм роботи арифметичного схрещування

```
alpha = random.uniform(0, 1)
offspring1 = alpha * parent1 + (1 - alpha) * parent2
offspring2 = alpha * parent2 + (1 - alpha) * parent1
```

Операції мутації. Мутація змінює деякі гени індивіда випадковим чином. Мета мутації – підтримувати генетичну різноманітність у популяції і допомагати уникнути передчасного збігання до локальних оптимумів.

Бітова мутація (лістинг 2.8) застосовується до двійкових хромосом, змінюючи значення бітів.

Лістинг 2.8 – Алгоритм роботи бітової мутації

```
for i in range(len(individual)):
    if random.random() < mutation_rate:
        individual[i] = 1 - individual[i]
```

Гаусова мутація (лістинг 2.9) використовується для числових хромосом, додаючи випадкове значення з нормального розподілу.

Лістинг 2.9 – Алгоритм роботи гаусової мутації

```
for i in range(len(individual)):
    if random.random() < mutation_rate:
        individual[i] += random.gauss(0, sigma)
```

Перестановочна мутація (лістинг 2.10) застосовується до порядкових хромосом, змінюючи порядок елементів

Лістинг 2.10 – Алгоритм роботи перестановочної мутації

```
if random.random() < mutation_rate:
    idx1, idx2 = random.sample(range(len(individual)), 2)
    individual[idx1], individual[idx2] = individual[idx2],
individual[idx1]
```

2.4 Еволюція популяції

Групова еволюція в генетичних алгоритмах імітує процес природного відбору, мутації та рекомбінації, і її метою є поступове покращення якості прийняття рішень групою [6]. Генетичні алгоритми використовують ідеї біологічної еволюції для пошуку оптимальних або майже оптимальних рішень складних проблем. У цьому розділі ми детально розглянемо процес еволюції популяції в генетичних алгоритмах.

Процес еволюції популяції:

- ініціалізація популяції;
- оцінка фітнесу;
- селекція;
- схрещування;
- мутація;
- формування нового покоління;
- роль різноманітності.

Пара батьківських особин вибирається з популяції, і їхні гени об'єднуються для отримання одного або кількох нащадків. Цей процес дозволяє змішувати генетичну інформацію батьків і допомагає створювати нові рішення [7].

На першому кроці створюється початкова популяція індивідумів, яка представляє можливі рішення проблеми. Початкова популяція може бути згенерована випадковим чином або з використанням деяких наближень, що призводить до більш продуктивного початкового рішення [5].

Кожен індивід у популяції оцінюється за допомогою функції відповідності, яка визначає якість рішення, яке представляє індивіда. Функція придатності присвоює кожному індивідууму числове значення, яке відображає його придатність для вирішення проблеми [6].

На основі показників фізичної підготовки відбираються люди для створення наступного покоління. Мета селекції полягає в тому, щоб дозволити особинам з

вищою пристосованістю мати більше шансів на відбір і таким чином передати свої гени майбутнім поколінням [7].

Мутації відбуваються в наступних поколіннях, вносячи нові гени в популяцію та зберігаючи генетичну різноманітність. Мутації випадковим чином змінюють деякі гени індивіда, що допомагає уникнути застою в локальному оптимумі [7].

Після селекції, гібридизації, мутації та інших операцій формується нове покоління особин. На зміну старому приходить нове покоління, і процес повторюється знову.

Генетичне різноманіття популяції є важливим фактором ефективності генетичних алгоритмів. Низька різноманітність може призвести до передчасної конвергенції до локального оптимуму, тоді як висока різноманітність корисна для пошуку глобального оптимуму.

Приклади еволюції популяції:

- оптимізація розкладів;
- еволюція стратегій в іграх.

Для задач оптимізації планування можуть бути представлені різні варіанти розподілу завдань, а функція відповідності може враховувати такі фактори, як час виконання, кількість конфліктів, ефективність використання ресурсів тощо.

У завданнях, пов'язаних з формулюванням ігрових стратегій, індивіди можуть представляти різні стратегії, а фітнес-функції можуть оцінювати успішність цих стратегій в ігровій ситуації.

Еволюція популяції в генетичних алгоритмах — це процес, який поєднує відбір, схрещування та мутацію для поступового покращення рішень. Правильна настройка цих операцій і збереження генетичного різноманіття є важливими факторами успіху генетичних алгоритмів.

3 РЕАЛІЗАЦІЯ ГРИ

Розробка комп'ютерних ігор з використанням нейронних мереж і генетичних алгоритмів є складним і творчим процесом, який об'єднує принципи штучного інтелекту та еволюційного програмування.

Реалізація гри «Змійка» на мові програмування Python [8] з використанням бібліотеки Pygame [12] – це складний процес, який включає декілька етапів.

Переваги обраного способу реалізації:

- простота та зручність;
- гнучкість;
- зрозумілий код.

Використання бібліотеки Pygame робить розробку гри простою та зручною, завдяки чому можна легко створювати графічний інтерфейс та обробляти події.

Реалізація на Python дозволяє легко інтегрувати інші модулі та бібліотеки, зокрема для роботи з нейронними мережами та генетичними алгоритмами.

Код гри легко читається та модифікується, що дозволяє швидко вносити зміни та покращення.

Недоліки обраного способу реалізації:

- продуктивність;
- обмежена графіка;
- відсутність вбудованих інструментів для AI.

Використання Python може бути не найефективнішим рішенням з точки зору продуктивності, особливо для великих та складних ігор.

Pygame не надає таких широких можливостей для створення графіки, як інші більш потужні ігрові рушії, такі як Unity або Unreal Engine;

Pygame не має вбудованих інструментів для роботи з нейронними мережами та генетичними алгоритмами, тому необхідно використовувати додаткові бібліотеки та модулі.

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 Створення гри «Змійка»

Гра «Змійка» займає особливе місце в історії комп'ютерних ігор, ставши однією з найвідоміших та найпопулярніших ігор, яка досі привертає увагу гравців різного віку. У своїй дипломній роботі я зосередився на використанні сучасних методів штучного інтелекту, включаючи нейронні мережі та генетичні алгоритми, для створення інтелектуальної версії гри. У цьому розділі я детально розгляну основні етапи створення гри «Змійка», реалізації її функціональних можливостей.

Ініціалізація гри та налаштування середовища (лістинг 3.2) на початковому етапі розробки гри «Змійка» я визначив основні компоненти гри та їхні взаємодії. Було прийнято рішення використовувати мову програмування Python завдяки її простоті, універсальності та великій кількості бібліотек для розробки ігор. Для створення графічного інтерфейсу я обрав бібліотеку Pygame, яка забезпечує потужні засоби для розробки ігор.

Лістинг 3.2 – Створення ігрового поля

```
import pygame
import random

pygame.init()
screen = pygame.display.set_mode((600, 600))
pygame.display.set_caption('Змійка')

# Кольори
WHITE = (255, 255, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)

clock = pygame.time.Clock()
block_size = 20
```

Створення класу для змійки(додаток А.1) наступним кроком було створення класу для змійки, який зберігає інформацію про її поточний стан та методи для переміщення і росту. Клас «Snake» містить такі методи, як `__init__` для ініціалізації змійки, `move` для її переміщення, `turn` для зміни напрямку руху, `reset` для скидання стану гри, та `draw` для відображення змійки на екрані.

Створення їжі для змійки (лістинг 3.3) їжа є важливим елементом гри, оскільки вона сприяє росту змійки та збільшенню її очок. Було створено клас «Food», який генерує їжу у випадкових місцях на ігровому полі та забезпечує її відображення.

Лістинг 3.3 – Програмний код створеного класу їжі

```
class Food:
    def __init__(self):
        self.position = (0, 0)
        self.color = RED
        self.randomize_position()

    def randomize_position(self):
        self.position = (random.randint(0, 29) * block_size,
random.randint(0, 29) * block_size)

    def draw(self, surface):
        r = pygame.Rect((self.position[0], self.position [1]),
(block_size, block_size))
        pygame.draw.rect(surface, self.color, r)
        pygame.draw.rect(surface, WHITE, r, 1)
```

Ігровий цикл (лістинг 3.4) основний ігровий цикл відповідає за оновлення стану гри та відображення її на екрані. Тут відбувається обробка подій, переміщення змійки та генерація нової їжі. Ігровий цикл виконує функції

оновлення положення змійки, перевірки на зіткнення з їжею та оновлення інтерфейсу користувача.

Лістинг 3.4 – Програмний код ігрового циклу

```
def game_loop():
    snake = Snake()
    food = Food()
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                return
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_UP:
                    snake.turn(UP)
                elif event.key == pygame.K_DOWN:
                    snake.turn(DOWN)
                elif event.key == pygame.K_LEFT:
                    snake.turn(LEFT)
                elif event.key == pygame.K_RIGHT:
                    snake.turn(RIGHT)

        snake.move()
        if snake.get_head_position() == food.position:
            snake.length += 1
            snake.score += 1
            food.randomize_position()

        screen.fill(WHITE)
        snake.draw(screen)
        food.draw(screen)
        pygame.display.update()
        clock.tick(10)

game_loop()
```

3.2 Проектування нейронної мережі для гри «Змійка»

Проектування та реалізація нейронної мережі для автоматизації стратегії гри «Змійка». Нейронна мережа буде навчатися приймати рішення щодо наступного кроку змії на основі поточного стану гри та оточуючого середовища.

Огляд методології - дослідження базується на використанні нейронних мереж для розробки алгоритмів, які керують поведінкою змії у грі «Змійка». Використано два основні підходи:

- модуль візуалізації гри;
- нейронна мережа для прийняття рішень.

У реалізації використовується модуль, який відповідає за візуальне представлення гри та її статистичних даних. Написаний код забезпечує побудову ігрового поля, відображення різноманітних таблиць і графіків, які аналізуються для визначення ефективності нейронної мережі(див. лістинг 3.1).

Нейронна мережа розроблена з використанням класу *Perceptron*, який має методи для ініціалізації ваг, випадкового генерування ваг, перемішування ваг для різних алгоритмів сімей і суперсімей, а також методи для обчислення вихідних значень мережі на основі вхідних даних(додаток А.2);

Ініціалізація ваг: Ваги нейронної мережі ініціалізуються згідно з параметрами, заданими користувачем або випадковим чином (лістинг 3.5).

Лістинг 3.5 – Створення випадкових ваг

```
def rand_w(self):
    self.w_in = np.random.uniform(-1, 1, (self.in_l_q,
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))
```

Обчислення вихідних значень: На основі поточного стану гри і переданих параметрів мережа обчислює вихідні значення, які визначають наступний крок змії (лістинг 3.6).

Лістинг 3.6 – Обчислення даних отриманих в ході гри

```
def calculation(self, inp_1):
    self.inp_1 = inp_1
    self.lay_1 = self.relu(np.dot(self.inp_1, 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)
```

Перемішування ваг для еволюційних алгоритмів: Використані методи перемішування ваг для оптимізації нейронної мережі. Це дозволяє покращити стратегії гри через генерацію нових варіантів ваг і вибір найбільш успішних з них (лістинг 3.7).

Лістинг 3.7 – Код перемішування ваг

```
def shuff(self, w1, w2, probability_a=70, probability_b=30, k=0):
    a = w1.ravel()
    b = w2.ravel()
    c = []
    for i in range(0, len(a)):
        c.append(random.choices([a[i], b[i], random.uniform(-1,
1)], weights=[probability_a, probability_b, k]))
    new_w = np.array(c)
    new_w = new_w.reshape(w1.shape)
    return new_w
```

Розроблена нейронна мережа для гри «Змійка» відображається як ефективний інструмент для автоматизації стратегій гравців у цій класичній грі. Вона відкриває можливості для подальшого розширення, оптимізації та вдосконалення алгоритмів на основі використання штучного інтелекту.

Ця робота демонструє значимість застосування нейронних мереж у вирішенні задач ігрової індустрії, зокрема в області розробки інтелектуальних агентів для комп'ютерних ігор.

3.3 Реалізація алгоритму

Генетичний алгоритм, використовуваний у цій роботі, реалізований на основі власної інтерпретації принципів еволюції. Він використовується

Ініціалізація популяції: Початкові нейронні мережі створюються з випадковими вагами. Це досягається за допомогою методу `rand_w`, який ініціалізує випадкові ваги для вхідного шару, прихованого шару і вихідного шару нейронної мережі (див. лістинг 3.5).

Оцінка пристосованості: Кожна нейронна мережа оцінюється за допомогою функції пристосованості, яка в даному випадку може бути оцінкою її продуктивності у грі "Змійка". Функція пристосованості дозволяє визначити, наскільки добре мережа справляється з поставленою задачею, наприклад, набирає бали в грі або зберігає свою змійку від зіткнень (лістинг 3.8).

Лістинг 3.8 – Код оцінки пристосованості

```
class Sesion_snake():
    def __init__(self):
        self.era_snakes_list = []
        self.snake_statistics =
pd.DataFrame(columns=['ai_format', 'family', 'epoch', 'score',
'moves', 'desk_size', 'sensors', 'lay_1', 'lay_2',
'top_parents_by_epoch', 'first_random_snakes', 'time_label'])
        self.time_label = time()

    def evaluate_fitness(self):
```

```

        for snake in self.era_snakes_list:
            fitness_score = snake.get_score() /
snake.get_moves()
            snake.set_fitness(fitness_score)

```

Селекція: Нейронні мережі з популяції вибираються для подальшого використання у селекційному процесі на основі їх функції пристосованості. Кращі мережі мають більші шанси на виживання і розмноження (лістинг 3.9).

Лістинг 3.9 – Код селекції

```

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(
        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])

```

Схрещування і мутація: Обрані нейронні мережі схрещуються, щоб створити нові потомствени мережі. Методи `w_suffle` та `w_suffle_superfam` використовуються для змішування ваг між нейронними мережами з метою створення нових потомків з новими комбінаціями ваг. Крім того, здійснюється мутація, яка дозволяє інтродукцію випадкових змін у вагах з певною ймовірністю, що дозволяє алгоритму виходити з локальних максимумів (лістинг 3.10).

Лістинг 3.10 – Код схрещування та мутації

```

def w_suffle(self, wl, probability_a, probability_b, k):
    w_in_list = []
    w_lay_list = []
    w_out_list = []
    deti_list = []

    for ves in wl:
        w_in_list.append(ves[0])
        w_lay_list.append(ves [1])
        w_out_list.append(ves [2])
        deti_list.append([ves[0], ves [1], ves [2]])

    for i in range(0, len(w_in_list)):
        for j in range(0, len(w_in_list)):
            child_in = self.shuff(w_in_list[i], w_in_list[j],
probability_a, probability_b, k)

```

```

        child_layer = self.shuff(w_layer_list[i], w_layer_list[j],
probability_a, probability_b, k)
        child_out = self.shuff(w_out_list[i], w_out_list[j],
probability_a, probability_b, k)
        deti_list.append([child_in, child_layer, child_out])

        child_in = self.shuff(w_in_list[i], w_in_list[j],
probability_a, probability_b, 0)
        child_layer = self.shuff(w_layer_list[i], w_layer_list[j],
probability_a, probability_b, 0)
        child_out = self.shuff(w_out_list[i], w_out_list[j],
probability_a, probability_b, 0)
        deti_list.append([child_in, child_layer, child_out])

    return deti_list

```

Оцінка і збереження: Після схрещування і мутації нові мережі оцінюються за їхньою пристосованістю, і кращі з них вибираються для наступної ітерації алгоритму. Цикл продовжується до досягнення заданої умови зупинки (наприклад, досягнення заданої кількості поколінь або досягнення оптимального рівня продуктивності) (лістинг 3.11).

Лістинг 3.11 – Оцінка результатів та збереження

```

class Sesion_snake():
    def select_the_best(self, fam, ep, pq):
        for i in self.era_snakes_list:

self.snake_statistics.loc[len(self.snake_statistics.index)] = {
            'ai_format': str(razmer_pole) + 'x' +
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

```

Збереження даних у файли (лістинг 3.12, рис.3.1).

Лістинг 3.12 – Код створення файлу зі статистикою гри

```

class Sesion_snake():

    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")

```

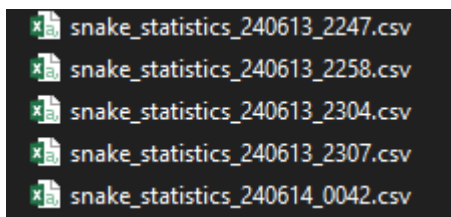


Рисунок 3.1 – Файли з збереженими результатами роботи програми

Використання методів (лістинг 3.13).

Лістинг 3.13 – Приклад коду з використанням методів

```
if __name__ == "__main__":

    ses.save_game_stat()
    ses.save_best_weights()
```

Реалізація генетичного алгоритму для оптимізації нейронної мережі в грі «Змійка» вимагає комплексного підходу до поєднання елементів штучного інтелекту та еволюційного обчислення. Використання власної реалізації методів для ініціалізації ваг, схрещування і мутації дозволяє забезпечити гнучкість і ефективність алгоритму, а також адаптувати його до конкретних умов ігрового середовища. Такий підхід дозволяє досягати значних результатів у вдосконаленні поведінки штучного інтелекту в ігровому середовищі «Змійка».

3.4 Навчання та тестування моделі

Для тестування моделі необхідно запуснути роботу коду та перевірити основні пункти або додати декілька за необхідності.

Код включає функції для відбору найкращих змій, оновлення табличних даних, форматування даних для відображення в таблицях та збереження найкращих ваг змій у файл, тобто усе необхідне для навчання НМ(додаток А.3, рис 3.2).

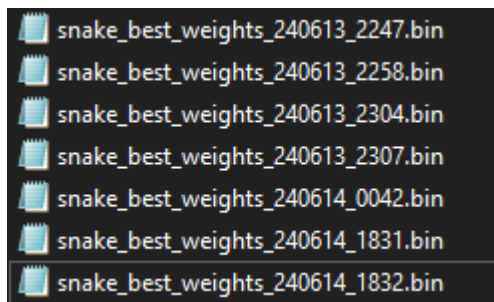


Рисунок 3.2 – Файли в яких зберігаються найкращі ваги змій

Наступний крок збір результатів усіх сімей за усі епохи – це дозволить швидко отримати інформацію про хід роботи(лістинг3.14, рис 3.3).

Лістинг 3.14 – Код створення таблиці з результатами найкращих індивідів

```
def table_scores(self, score, count_moves, count_all_moves):
    x = self.otstup + self.cell
    y = self.pole_size + 15
    dy = 25
    table_font = pygame.font.SysFont("Arial", 18)

    text = 'Score: ' + str(score) + '          Moves: ' +
str(count_all_moves)
    img = table_font.render(text, True, self.text_col)
    self.rect_scores = img.get_rect()
    self.window.blit(img, (x,y))
```

	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
1	2	2	3	2	3	9	24	21	21	35	27	28	26	26	23	27	26	28	27	30	33
2	1	2	3	3	2	3	2	3	4	18	16	20	27	25	26	28	32	33	35	32	33
3	2	4	8	16	22	25	21	25	24	28	31	33	29	33	33	35	40	35	35	45	36
4	2	2	1	2	2	2	2	5	10	16	23	26	21	21	28	33	30	28	27	31	35
5	1	1	1	3	2	5	12	20	18	22	24	24	28	27	24	36	22	23	28	24	26
6	2	2	2	2	3	3	4	3	3	3	2	4	13	23	21	21	22	26	22	22	22
7	2	2	2	3	3	5	4	16	30	26	33	28	38	28	36	30	30	34	33	32	37
900	40	43	47	40	39	47	40	35	36	44	36	39	40	41	44	43	37	40	40	40	42
901	43	42	44	41	36	40	40	41	35	37	40	46	42	44	42	40	46	37	43	42	47
902	38	41	46	46	57	59	66	67	69	68	-	-	-	-	-	-	-	-	-	-	-

Рисунок 3.3 – Таблиця з результатами усіх сімейств на протязі 100 епох

Далі аналіз результатів в реальному часу за допомогою невеликої таблиці, що дозволить зробити висновки про ефективність навчання НМ(лістинг 3.15, рис. 3.4).

Лістинг 3.15 – Код постійного моніторингу та виведення результатів

```
def now_top(self, df_now, fam, epoch, population=200):
    x= self.win_h - self.otstup - 790 - 160 - self.otstup_block
    y= self.otstup
    surf_actual_top = pygame.Surface((160, 300))
    surf_actual_top.fill(self.GREY)
    pygame.draw.line(surf_actual_top, self.L_WHITE, [5, 30],
[155, 30])
    pygame.draw.line(surf_actual_top, self.L_WHITE, [5, 270],
[155, 270])
    self.window.blit(surf_actual_top, (x, y))
    txt_head = "Family: " + str(fam) + "   Epoch: " + str(epoch)
    self.window.blit(self.text_font_sm.render(txt_head, True,
self.text_col), (x + 10, y + 10))
    otstup_y = 35
    dy = 20
    dict = df_now.to_dict('list')
    self.window.blit(self.text_font_sm.render("Score", True,
self.text_col), (x + 10, y + otstup_y))
    self.window.blit(self.text_font_sm.render("Moves", True,
self.text_col), (x + 90, y + otstup_y))
    for i in range(0, len(dict[list(dict.keys())[0]])):
        self.window.blit(self.text_font_sm.render(str(dict['score'][i]),
True, self.text_col), (x + 20 , y + 10 + otstup_y + dy))
        self.window.blit(self.text_font_sm.render(str(dict['moves'][i]),
True, self.text_col), (x + 100, y + 10 + otstup_y + dy))
        y += dy

self.window.blit(self.text_font_sm.render(str("Population:      " +
str(population)), True, self.text_col),
                (x + 10, 285))
```

Family: 902 Epoch: 47	
Score	Moves
71	1793
69	2125
65	1790
65	1914
63	1453
63	1703
62	1680
62	1922
61	1932
60	1290
Population: 1300	

Рисунок 3.4 – Таблиця з найкращими результатами у реальному часі

Для зручності орієнтування та спостереження за ходом навчання створена таблиця з рекордами за увесь час, що надає змогу лишати НМ на одинці а потім просто дізнатися результати (лістинг 3.16, рис. 3.5).

Лістинг 3.16 – Формування таблиці з найкращими результатами за увесь час

```
def top_table(self, df_all):
    x= self.win_h - 700 - self.otstup_block - 250 - self.otstup
    y= self.otstup + 300 + self.otstup_block - 1
    surf_top_table = pygame.Surface((250, 310))
    surf_top_table.fill(self.GREY)
    pygame.draw.line(surf_top_table, self.L_WHITE, [5, 30],
[245, 30], 1)
    self.window.blit(surf_top_table, (x, y))
    self.window.blit(self.text_font_sm.render("TOP      score
table:", True, self.text_col), (x+self.otstup, y+self.otstup))
```

```

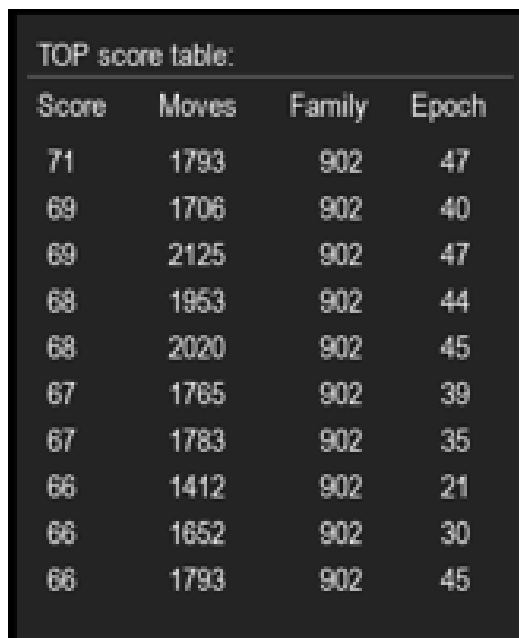
otstup_y = 30
dy = 23
dict = df_all.to_dict('list')

self.window.blit(self.text_font_sm.render("Score", True,
self.text_col), (x + 10, y + 35))
self.window.blit(self.text_font_sm.render("Moves", True,
self.text_col), (x + 70, y + 35))
self.window.blit(self.text_font_sm.render("Family", True,
self.text_col), (x + 135, y + 35))
self.window.blit(self.text_font_sm.render("Epoch", True,
self.text_col), (x + 195, y + 35))

for i in range(0, len(dict[list(dict.keys())[0]])):

self.window.blit(self.text_font_sm.render(str(dict['score'][i]),
True, self.text_col), (x + 15 , y + 10 + otstup_y + dy))

```



TOP score table:

Score	Moves	Family	Epoch
71	1793	902	47
69	1706	902	40
69	2125	902	47
68	1953	902	44
68	2020	902	45
67	1765	902	39
67	1783	902	35
66	1412	902	21
66	1652	902	30
66	1793	902	45

Рисунок 3.5 – Таблиця з найкращими результатами за увесь час

Також для зручності спостереження та можливості швидкого доступу до результатів усіх сімейств за усю ігрову сесію, а також простої можливості порівняння результатів необхідно перевірити графік (лістинг 3.17, рис. 3.6).

Лістинг 3.17 – Код створення графіку результатів

```
try:
    var = sys.argv [1]
except IndexError:
    var = ''

plt.style.use('ggplot')
file_name = "snake_statistics_" + str(var) + ".csv"
data = pd.read_csv(file_name)
data = data.query('time_label == @var')

smooth_rolling = 3
pt_mean = data.pivot_table(index='family', columns='epoch',
values='score', aggfunc='max').T
fig, ax = plt.subplots()
fig.set_size_inches(7, 3.5)
fig.set_facecolor('#F5F5F5')
ax.set_facecolor('#F5F5F5')
ax.plot(pt_mean.rolling(smooth_rolling).mean(),
label=pt_mean.columns)
ax.set_title('Score for TOP-1 snakes (smooth
rolling: 3 epochs)', fontsize=12)
ax.set_ylabel('Score', fontsize=12)
ax.set_xlabel("Epochs", fontsize=12)
ax.set_xlim(0, 100)
ax.set_ylim(0)
if len(pt_mean.count()) > 1:
    fig.legend(title='Family', loc='right', fontsize=7)

fig.savefig("snake_plot.png", dpi=100)
```

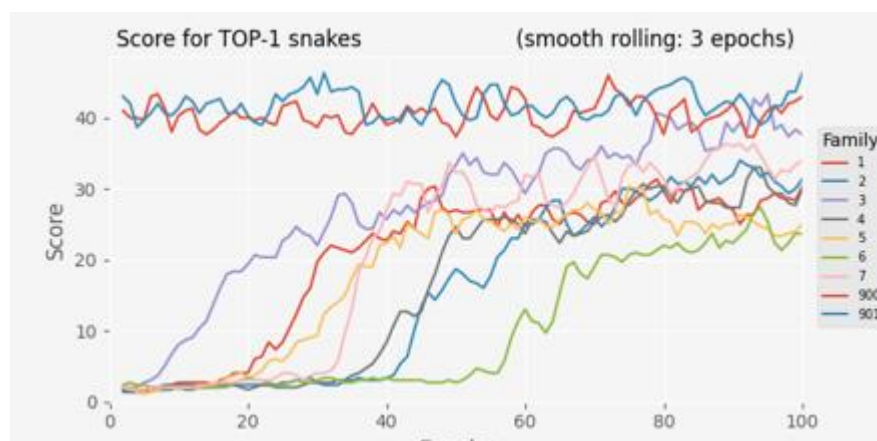


Рисунок 3.6 – Графік на якому відображено порівняння результатів усіх сімей

Під час тестування виявилось, що графік відображається невірно для її усунення доповнимо код (лістинг 3.18).

Лічтинг 3.18 – Код для коректного відображення графіку

```
def show_plot(self):
    self.img_stat = pygame.image.load('snake_plot.png')

    x= self.win_h - self.otstup - 700
    y= self.otstup + 300 + self.otstup_block
    self.window.blit(self.img_stat, (x, y))
```

По завершенню тестів необхідно перевірити базові потреби які мають бути задоволені (рис. 3.7):

- середовище гри було виконано згідно з вимогами до зручності спостереження за ходом роботи, а також керування процесом;
- НМ успішно була інтегрована в середовище гри;
- таблиці спрощують спостереження та працюють вірно;
- графік вірно відображає результати роботи що дає змогу контролювати процес навчання не відкриваючи файли з даними;
- НМ вірно зберігає та опрацьовує дані.

Також можна зробити декілька висновків щодо самого НМ:

- НМ змогла розвинути деякі навички гри в «Змійку», на момент цього тесту рекорд становить 71 яблуко;
- якщо уважно спостерігати за поведінкою НМ під час на пізніших етапах гри можна побачити деяку закономірність рухів, що свідчить про вірну роботу навчання, запису даних, зчитування даних з файлів а також інтегрування цих даних до наступних поколінь сімейства змій (рис. 3.8).

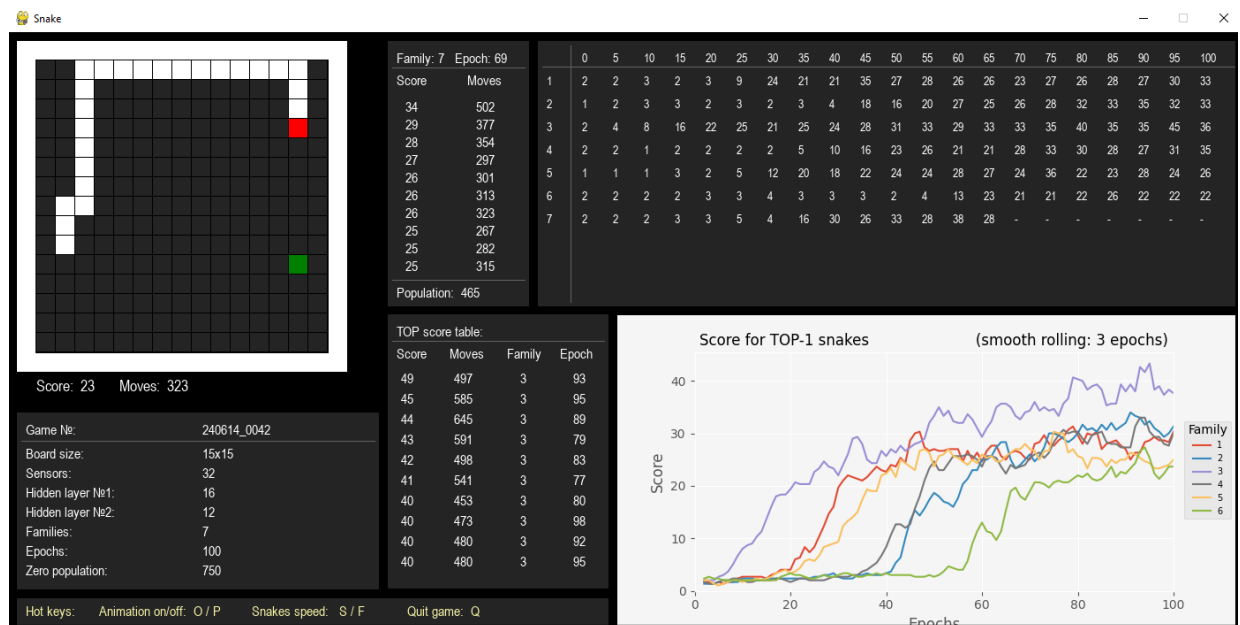


Рисунок 3.7 – Результати першого тестування

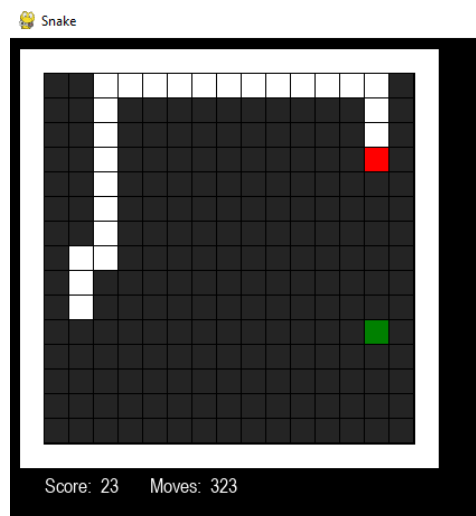


Рисунок 3.8 – Процес розвитку НМ під час гри

ВИСНОВКИ

На основі проведеної роботи по дослідженню та розробки навчального агента для гри "Змійка" з використанням нейронних мереж та генетичного алгоритму можна зробити наступні висновки.

У роботі був проведений всебічний аналіз технологій створення ігор та освітніх ігрових середовищ. Вивчення різноманітних архітектур нейронних мереж та алгоритмів навчання дало змогу сформуванати глибоке уявлення про методи розробки ігрових агентів, що підтримують високий рівень адаптації та ефективність у грі.

Генетичний алгоритм був обраний як ефективний метод навчання агента у варіативних умовах гри "Змійка". Детальний аналіз функціоналу і реалізація алгоритму підтвердили його здатність до еволюції стратегій та покращення результатів без потреби у вручному програмуванні.

Нейронна мережа була успішно навчена грати в "Змійку", що підтверджує можливість застосування генетичних алгоритмів для навчання ігрових агентів. Отримані результати свідчать про високий рівень адаптації мережі до геймплею та її здатність до самооптимізації в процесі гри.

Цей підхід є перспективним для подальших досліджень у сфері штучного інтелекту в ігровій індустрії. Використання генетичного алгоритму дозволяє досягати високих результатів без прямого програмування ігрових стратегій, що відкриває нові можливості для розвитку ігрових технологій і навчальних інтерактивних середовищ.

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 29.04.2024).
- 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 23.04.2024).
- 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 на тему машинного навчання
<https://machinelearningmastery.com/> (дата звернення 15.04.2024).
- 14 Дослідження нейронних мереж URL:<https://www.kaggle.com/> (дата звернення 16.04.2024).
- 15 Дослідження можливості поєднання нейронних мереж та відео ігор
URL:<https://deepmind.google/technologies/alphago/> (дата звернення 23.04.2024).

ДОДАТОК А Лістинги змійки, нейронної мережі та функції обробки

Лістинг А.1 – код класу для створення змійки

```
class Snake:
    def __init__(self):
        self.length = 1
        self.positions = [((600 // 2), (600 // 2))]
        self.direction = random.choice([UP, DOWN, LEFT, RIGHT])
        self.color = GREEN
        self.score = 0

    def get_head_position(self):
        return self.positions[0]

    def turn(self, point):
        if self.length > 1 and (point[0] * -1, point [1] * -1) ==
self.direction:
            return
        else:
            self.direction = point

    def move(self):
        cur = self.get_head_position()
        x, y = self.direction
        new = (((cur[0] + (x * block_size)) % 600), (cur [1] + (y
* block_size)) % 600)
        if len(self.positions) > 2 and new in self.positions[2:]:
            self.reset()
        else:
            self.positions.insert(0, new)
            if len(self.positions) > self.length:
                self.positions.pop()

    def reset(self):
```

```

self.length = 1
self.positions = [((600 // 2), (600 // 2))]
self.direction = random.choice([UP, DOWN, LEFT, RIGHT])
self.score = 0

def draw(self, surface):
    for p in self.positions:
        r = pygame.Rect((p[0], p[1]), (block_size,
block_size))

        pygame.draw.rect(surface, self.color, r)
        pygame.draw.rect(surface, WHITE, r, 1)

```

Лістинг А.2 – код нейронної мережі

```

import numpy as np
import random

class Perceptron:
    def __init__(self, in_l_q, lay_1_q, lay_2_q, out_l_q):
        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.w_in = None
        self.w_lay = None
        self.w_out = None
        self.inp_l = None
        self.lay_1 = None
        self.lay_2 = None
        self.out_l = None

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

    def new_pct(self):

```

```

        self.lay_1 = np.zeros(self.lay_1_q)
        self.lay_2 = np.zeros(self.lay_2_q)
        self.out_1 = np.zeros(self.out_1_q)

    def rand_w(self):
        self.w_in = np.random.uniform(-1, 1, (self.in_1_q,
self.lay_1_q))
        self.w_layer = 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))

    def get_w(self, rw_in, rw_layer, rw_out):
        self.w_in = rw_in
        self.w_layer = rw_layer
        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(0, len(a)):
            c.append(random.choices([a[i], b[i], random.uniform(-
1, 1)], weights=[probability_a, probability_b, k]))
        new_w = np.array(c)
        new_w = new_w.reshape(w1.shape)
        return new_w

    def w_shuffle(self, w1 ,probability_a ,probability_b, k):
        w_in_list = []
        w_layer_list = []
        w_out_list = []
        deti_list = []

```

```

    for ves in wl:
        w_in_list.append(ves[0])
        w_layer_list.append(ves [1])
        w_out_list.append(ves [2])
        deti_list.append([ves[0], ves [1], ves [2]])

    for i in range(0, len(w_in_list)):
        for j in range(0, len(w_in_list)):
            child_in = self.shuff(w_in_list[i], w_in_list[j],
probability_a, probability_b, k)
            child_layer = self.shuff(w_layer_list[i],
w_layer_list[j], probability_a, probability_b, k)
            child_out = self.shuff(w_out_list[i],
w_out_list[j], probability_a, probability_b, k)
            deti_list.append([child_in, child_layer,
child_out])

            child_in = self.shuff(w_in_list[i], w_in_list[j],
probability_a, probability_b, 0)
            child_layer = self.shuff(w_layer_list[i],
w_layer_list[j], probability_a, probability_b, 0)
            child_out = self.shuff(w_out_list[i],
w_out_list[j], probability_a, probability_b, 0)
            deti_list.append([child_in, child_layer,
child_out])

    return deti_list

def w_shuffle_superfam(self, wl ,probability_a ,probability_b,
k):
    w_in_list = []
    w_layer_list = []
    w_out_list = []
    deti_list = []

```

```

    for ves in wl:
        w_in_list

```

Лістинг А.3 – код функції відбору обробки та збереження ваг

```

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(
                                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 update_pivot_scores(self, fam, epoch):
        if epoch in self.df_cols:
            temp_df = (
                self.snake_statistics
                .query('family == @fam and epoch == @epoch and
time_label == @self.time_label')
                .sort_values(by='score', ascending=False)
                .head(3)
            )
            self.df_pivot_scores.at[fam, epoch] =
temp_df['score'].max()

    def df_now(self, fam, epoch, empty=False):
        max_top_lines = 10
        if empty:
            table_top_now = pd.DataFrame({'score': ['- ', '- ', '- ', '- ',
'- ', '- ', '- ', '- ', '- ', '- '],
                                          'moves': ['- ', '- ', '- ', '- ',
'- ', '- ', '- ', '- ', '- ', '- ']}))
        else:
            table_top_now = (
                self.snake_statistics
                .query('time_label == @self.time_label and family ==
@fam and epoch ==@epoch')
                .sort_values(by=['score', 'moves'], ascending=[False,
True])
                .head(max_top_lines)[['score', 'moves', 'family',
'epoch']]

```

```

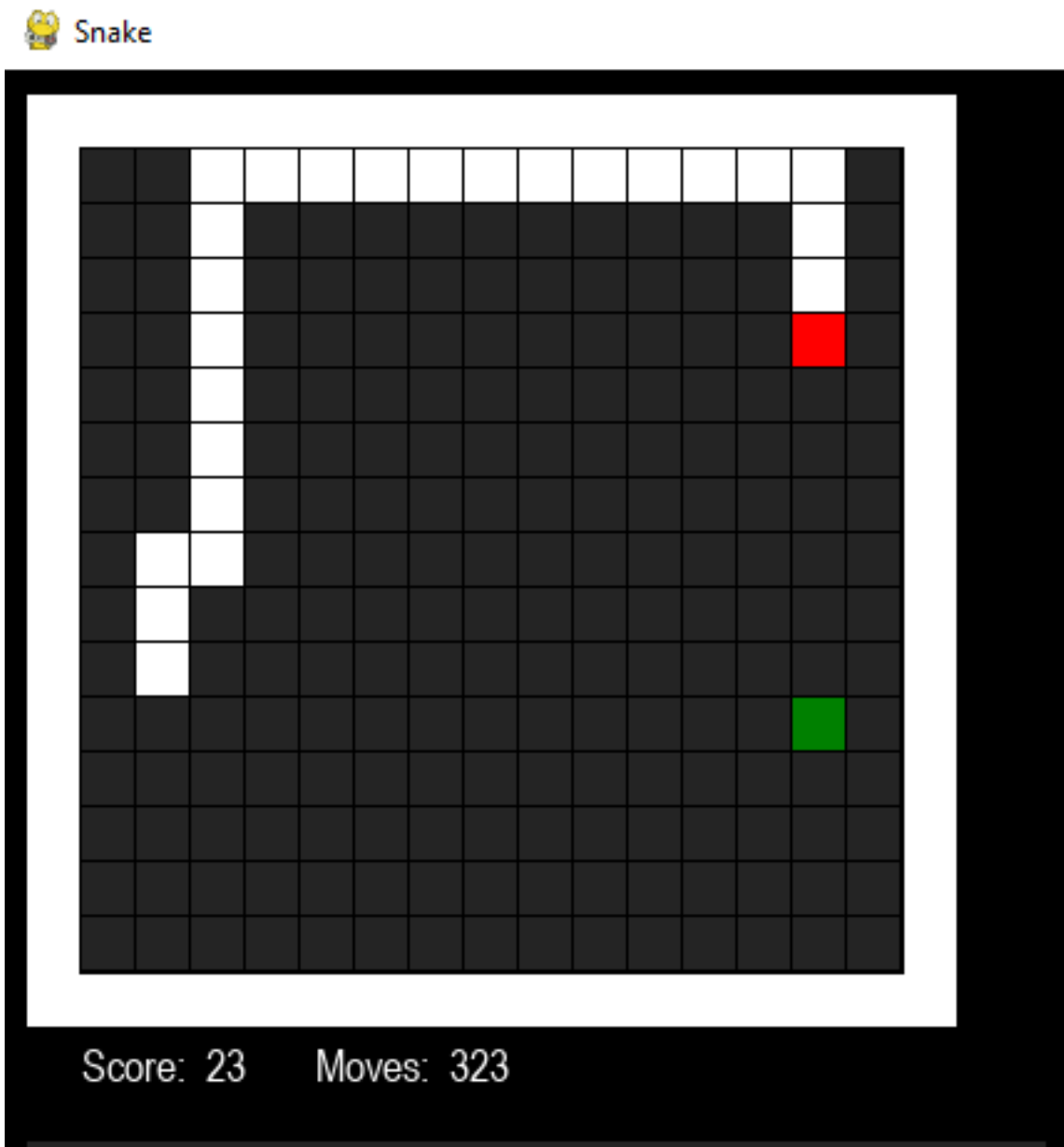
    )
    table_top_now = table_top_now[['score', 'moves']]

    return table_top_now

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")

```

ТЕСТУВАННЯ ГРИ



```
ke_main.py | snake_matrix.py | rnd_snakes.py | snake_draw.py | snake_ai.py | plot_t
y= self.otstup
otstup_x = 35
otstup_y = 26
surf_pivot_score = pygame.Surface((790, 300))
surf_pivot_score.fill(self.GREY)
pygame.draw.line(surf_pivot_score, self.L_WHITE, [5, 29], [845, 29])
pygame.draw.line(surf_pivot_score, self.L_WHITE, [37, 5], [37, 295])
self.window.blit(surf_pivot_score, (x, y))

dx = otstup_x
for i in df.columns.values.tolist():
    self.window.blit(self.text_font_smm.render(str(i), True, self.text_col),
                    (x + 15 + dx, y + 10))
    dx += otstup_x

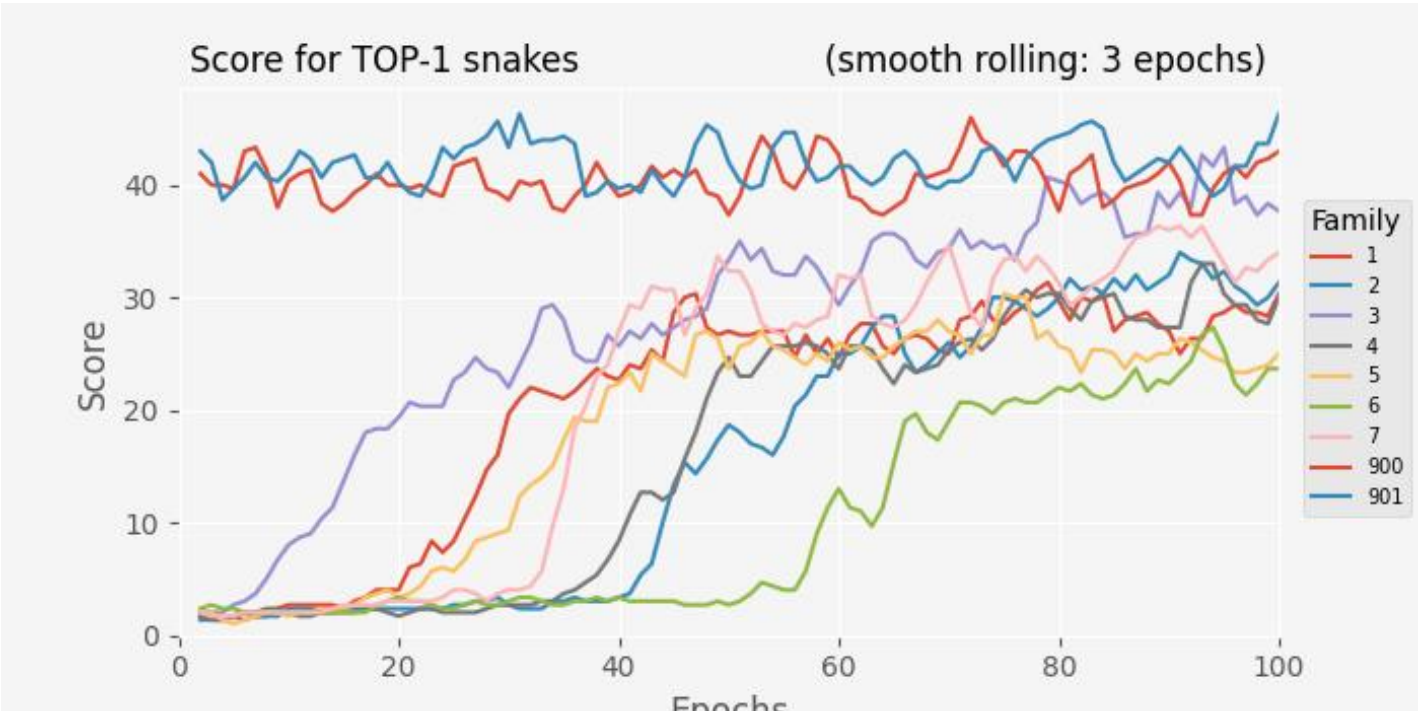
dy = otstup_y
for i in df.index.tolist():
    self.window.blit(self.text_font_smm.render(str(i), True, self.text_col),
                    (x + 10, y + 10 + dy))
    dy += otstup_y

dy = 0
for ind in df.index.tolist():
    dx = 0
    for clm in df.columns.values.tolist():
        self.window.blit(self.text_font_smm.render(str(df.loc[ind][clm]), True, self.text_col),
                        (x + 15 + otstup_x + dx, y + 10 + otstup_y + dy))
        dx += otstup_x
    dy += otstup_y

# отбраження графика
2.5.2 (SDL 2.28.3, Python 3.12.4)
```

					13.02070849.51131 ПЛ1						
					Розробка комп'ютерної гри з використанням штучного інтелекту Тестування гри	Літ.			Маса	Масштаб	
Зм	Лист	№ докум.	Підп.	Дата							
Розроб.		Таровик М.Є.									
Перевір.		Тягунова М.Ю.									
Т. контр.						Лист. 1			Листів 1		
						НУ «Запорізька політехніка» КНТ- 511сп					
Н. контр.		Щербак Н.В.									
Затв.		Кудерметов Р.К.									

ГРАФІК РЕЗУЛЬТАТІВ КОЖНОГО СІМЕЙСТВА



```
1 import sys
2 import pandas as pd
3 import matplotlib.pyplot as plt
4
5 try:
6     var = sys.argv[1]
7 except IndexError:
8     var = ''
9
10
11 plt.style.use('ggplot')
12
13 file_name = "snake_statistics_" + str(var) + ".csv"
14 data = pd.read_csv(file_name)
15 data = data.query('time_label == @var')
16
17 smooth_rolling = 3
18 pt_mean = data.pivot_table(index='family', columns='epoch', values='score', aggfunc='max').T
19
20 fig, ax = plt.subplots()
21 fig.set_size_inches(7, 3.5)
22 fig.set_facecolor('#F5F5F5')
23 ax.set_facecolor('#F5F5F5')
24
25 ax.plot(pt_mean.rolling(smooth_rolling).mean(), label=pt_mean.columns)
26
27 ax.set_title('Score for TOP-1 snakes (smooth rolling: 3 epochs)', fontsize=12)
28 ax.set_ylabel('Score', fontsize=12)
29 ax.set_xlabel("Epochs", fontsize=12)
30 ax.set_xlim(0, 100)
31 ax.set_ylim(0)
32 if len(pt_mean.count()) > 1:
33     fig.legend(title='Family', loc='right', fontsize=7)
34
35 fig.savefig("snake_plot.png", dpi=100)
```

					13.02070849.51131 ПЛ2					
					Розробка комп'ютерної гри з використанням штучного інтелекту Графік результатів кожного сімейства	Літ.		Маса	Масштаб	
Зм	Лист	№ докум.	Підп.	Дата						
Розроб.		Таровик М.Є.								
Перевір.		Тягунова М.Ю.								
Т. контр.						Лист. 1		Листів 1		
						НУ «Запорізька політехніка» КНТ-511сп				
Н. контр.		Щербак Н.В.								
Затв.		Кудерметов Р.К.								

ТАБЛИЦЯ РЕЗУЛЬТАТІВ ПРО НАВЧАННЯ НМ

```
#функция сохранения данных статистики
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')

#функция сохранения данных и весов
```

	A	B	C	D	E	F	G	H	I	J	K	L
1	ai_format,family,epoch,score,moves,desk_size,sensors,lay_1,lay_2,top_parents_by_epoch,first_random_snakes,time_label											
2	15x15_32in16_12,1,0,2,8,15,32,16,12,45,750,240614_0042											
3	15x15_32in16_12,1,0,1,16,15,32,16,12,45,750,240614_0042											
4	15x15_32in16_12,1,0,1,3,15,32,16,12,45,750,240614_0042											
5	15x15_32in16_12,1,0,1,12,15,32,16,12,45,750,240614_0042											
6	15x15_32in16_12,1,0,1,9,15,32,16,12,45,750,240614_0042											
7	15x15_32in16_12,1,0,1,10,15,32,16,12,45,750,240614_0042											
8	15x15_32in16_12,1,0,1,8,15,32,16,12,45,750,240614_0042											
9	15x15_32in16_12,1,0,1,10,15,32,16,12,45,750,240614_0042											
10	15x15_32in16_12,1,0,1,16,15,32,16,12,45,750,240614_0042											
11	15x15_32in16_12,1,0,1,10,15,32,16,12,45,750,240614_0042											
12	15x15_32in16_12,1,0,1,2,15,32,16,12,45,750,240614_0042											
13	15x15_32in16_12,1,0,1,8,15,32,16,12,45,750,240614_0042											
14	15x15_32in16_12,1,0,1,7,15,32,16,12,45,750,240614_0042											
15	15x15_32in16_12,1,0,1,4,15,32,16,12,45,750,240614_0042											
16	15x15_32in16_12,1,0,1,5,15,32,16,12,45,750,240614_0042											
17	15x15_32in16_12,1,0,1,10,15,32,16,12,45,750,240614_0042											
18	15x15_32in16_12,1,0,1,7,15,32,16,12,45,750,240614_0042											
19	15x15_32in16_12,1,0,1,8,15,32,16,12,45,750,240614_0042											
20	15x15_32in16_12,1,0,1,13,15,32,16,12,45,750,240614_0042											
21	15x15_32in16_12,1,0,1,7,15,32,16,12,45,750,240614_0042											
22	15x15_32in16_12,1,0,1,13,15,32,16,12,45,750,240614_0042											
23	15x15_32in16_12,1,0,1,7,15,32,16,12,45,750,240614_0042											
24	15x15_32in16_12,1,0,1,22,15,32,16,12,45,750,240614_0042											
25	15x15_32in16_12,1,0,1,13,15,32,16,12,45,750,240614_0042											
26	15x15_32in16_12,1,0,1,9,15,32,16,12,45,750,240614_0042											
27	15x15_32in16_12,1,0,1,22,15,32,16,12,45,750,240614_0042											
28	15x15_32in16_12,1,0,1,11,15,32,16,12,45,750,240614_0042											
29	15x15_32in16_12,1,0,1,8,15,32,16,12,45,750,240614_0042											
30	15x15_32in16_12,1,0,0,7,15,32,16,12,45,750,240614_0042											
31	15x15_32in16_12,1,0,0,9,15,32,16,12,45,750,240614_0042											
32	15x15_32in16_12,1,0,0,7,15,32,16,12,45,750,240614_0042											
33	15x15_32in16_12,1,0,0,1,15,32,16,12,45,750,240614_0042											
34	15x15_32in16_12,1,0,0,16,15,32,16,12,45,750,240614_0042											
35	15x15_32in16_12,1,0,0,7,15,32,16,12,45,750,240614_0042											
36	15x15_32in16_12,1,0,0,7,15,32,16,12,45,750,240614_0042											
37	15x15_32in16_12,1,0,0,1,15,32,16,12,45,750,240614_0042											

					13.02070849.51131 ПЛЗ					
					Розробка комп'ютерної гри з використанням штучного інтелекту Таблиця інформації про навчання НМ	Літ.		Маса	Масштаб	
Зм	Лист	№ докум.	Підп.	Дата						
Розроб.		Таровик М.Є.								
Перевір.		Тягунова М.Ю.								
Т. контр.						Лист. 1		Листів 1		
						НУ «Запорізька політехніка» КНТ-511сп				
Н. контр.		Щербак Н.В.								
Затв.		Кудерметов Р.К.								

ТАБЛИЦЯ РЕЗУЛЬТАТІВ РОБОТИ ПРОГРАМИ

```
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] # отобрали установленное количество лучших родителей

    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])
```

	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
1	2	2	3	2	3	9	24	21	21	35	27	28	26	26	23	27	26	28	27	30	33
2	1	2	3	3	2	3	2	3	4	18	16	20	27	25	26	28	32	33	35	32	33
3	2	4	8	16	22	25	21	25	24	28	31	33	29	33	33	35	40	35	35	45	36
4	2	2	1	2	2	2	2	5	10	16	23	26	21	21	28	33	30	28	27	31	35
5	1	1	1	3	2	5	12	20	18	22	24	24	28	27	24	36	22	23	28	24	26
6	2	2	2	2	3	3	4	3	3	3	2	4	13	23	21	21	22	26	22	22	22
7	2	2	2	3	3	5	4	16	30	26	33	28	38	28	-	-	-	-	-	-	-

					13.02070849.51131 ПЛ4					
					Розробка комп'ютерної гри з використанням штучного інтелекту Таблиця результатів роботи програми	Літ.		Маса	Масштаб	
Зм	Лист	№ докум.	Підп.	Дата						
Розроб.		Таровик М.Є.								
Перевір.		Тягунова М.Ю.								
Т. контр.										
						Лист. 1		Листів 1		
Н. контр.		Щербак Н.В.				НУ «Запорізька політехніка» КНТ-511сп				
Затв.		Кудерметов Р.К.								