

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ІНТЕГРАЦІЯ ГЕНЕТИЧНОГО АЛГОРИТМУ В
НЕЙРОМЕРЕЖУ З НАВЧАННЯМ З ПІДКРІПЛЕННЯМ
INTEGRATION OF A GENETIC ALGORITHM INTO
A NEURAL NETWORK WITH REINFORCEMENT LEARNING

Виконав(ла): студент(ка) 4 курсу, групи КНТ-110
Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)
Інженерія програмного забезпечення

ТРІНКАЛЬ М.Є.

(ПРИЗВИЩЕ та ініціали)

Керівник ГЛАДКОВА О.М.

(ПРИЗВИЩЕ та ініціали)

Рецензент ЗЕЛЕНЬОВА І.Я

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2024 року

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

ТРІНКАЛЬ Максима Євгеновича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Інтеграція генетичного алгоритму в нейромережу з навчанням з підкріпленням. Integration of a Genetic Algorithm into a Neural Network with Reinforcement Learning

керівник проєкту (роботи) к.т.н., ГЛАДКОВА Ольга Миколаївна,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 24 ” квітня 2024 року № 168

2. Строк подання студентом проєкту (роботи) 10 червня 2024 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Постановка завдання дослідження. 2. Огляд існуючих рішень. 3. Інструменти та алгоритми. 4. Опис програмної реалізації. 5. Всесвіт та тестування. 6. Робота користувача з бібліотекою моделі.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	ГЛАДКОВА О.М., доцент		
Нормоконтролер	БЄЛОВА А.В., асистент		

7. Дата видачі завдання “ 15 ” квітня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів вирішення задачі.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 5, 6
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Максим ТРІНКАЛЬ
(Імя ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Ольга ГЛАДКОВА
(Імя ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 130 с., 35 рис., 2 дод., 17 джерел.

НЕЙРОМЕРЕЖІ, НАВЧАННЯ З ПІДКРІПЛЕННЯМ, ГЕНЕТИЧНІ АЛГОРИТМИ, ОПТИМІЗАЦІЯ, ГЛИБОКЕ НАВЧАННЯ.

Об'єкт дослідження – моделі нейронних мереж та еволюційні алгоритми.

Предмет дослідження – модель нейронної мережі з навчанням з підкріпленням та генетичний алгоритм.

Мета роботи – розробка та дослідження моделі, що інтегрує генетичний алгоритм в нейронні мережі з навчанням з підкріпленням, з метою оптимізації процесу навчання та підвищення загальної ефективності алгоритмів у вирішенні складних задач.

Матеріали, методи та технічні засоби: структурне та об'єктно-орієнтоване програмування, генетичний алгоритм, моделі навчання нейронних мереж, алгоритми ініціалізації нейронів, активації нейронів, навчання нейронних мереж, мова програмування C#, персональний комп'ютер з процесором Intel Core i5 під управлінням операційної системи Microsoft Windows 10.

Результати. Створено модель нейронної мережі з підкріпленням та генетичним алгоритмом. Інтеграція генетичного алгоритму в нейромережі з навчанням з підкріпленням дає змогу підвищити ефективність навчання за рахунок ефективнішого пошуку оптимальних параметрів і структури мережі.

Висновки. Розроблено модель нейронної мережі з навчанням з підкріпленням, яка автоматизує та прискорює процес навчання агента.

Галузь використання – дослідження штучного життя, робототехніка, банківська справа.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 130 pages, 35 figures, 2 appendixes, 17 sources.

NEURAL NETWORKS, REINFORCEMENT LEARNING, GENETIC ALGORITHMS, OPTIMIZATION, DEEP LEARNING.

Object of study is a neural network models and evolutionary algorithms.

Subject of study is a neural network model with reinforcement learning and a genetic algorithm.

The purpose of the work is the development and research of a model that integrates a genetic algorithm into neural networks with reinforcement learning in order to optimize the learning process and increase the overall efficiency of algorithms in solving complex problems.

Materials, methods and tools: structural and object-oriented programming, genetic algorithm, neural network learning models, neuron initialization algorithms, neuron activation, neural network training, C# programming language, personal computer with an Intel Core i5 processor under control Microsoft Windows 10 operating system.

The results of the study show that the integration of the genetic algorithm into neural networks with reinforcement learning makes it possible to increase the effectiveness of learning due to a more effective search for optimal parameters and network structure. This approach can be used to solve a wide range of problems, where it is necessary to achieve high accuracy and adaptability of models.

Conclusions. A neural network model with reinforcement learning was developed, which automates and accelerates the learning process of the agent.

Field of use – artificial life research, robotics, banking.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	10
Вступ.....	11
1 Постановка завдання дослідження.....	12
1.1 Постановка завдань дослідження.....	12
1.2 Висновки до розділу	13
2 Огляд існуючих рішень	14
2.1 Глибоке навчання з підкріпленням із застосуванням генетичного алгоритму для оптимізації параметрів.....	14
2.2 Автоматична оптимізація гіперпараметрів за допомогою генетичного алгоритму в глибокому навчанні з підкріпленням для робототехнічних маніпуляцій.....	15
2.3 Нур-RL: Оптимізація гіперпараметрів за допомогою навчання з підкріпленням.....	16
2.4 Висновки до розділу	17
3 Інструменти та алгоритми.....	19
3.1 Середовище розробки Microsoft Visual Studio та C#.....	20
3.1.1 Microsoft Visual Studio.....	20
3.1.2 C#	20
3.2 Double Deep Q-Network	21
3.3 Алгоритм генетичного програмування.....	23
3.4 Temporal Difference Learning.....	26
3.5 Система обчислення нагороди.....	27
3.6 Нормалізація MinMax та Tanh	28
3.6.1 MinMax Нормалізація.....	29
3.6.2 Tanh Нормалізація.....	29
3.7 Stochastic Gradient Descent з Momentum.....	29
3.8 Регуляризатори Dropout, L2, Weights Noise	30

3.8.1 Dropout	31
3.8.2 L2 Регуляризація (Ridge Regression)	31
3.8.3 Weights Noise (Шум в Вагах)	31
3.8.4 Поєднання методів регуляризації	32
3.9 Функція активізації Swish	32
3.10 Ініціалізатор He	33
3.11 Epsilon-greedy стратегія дослідження	33
3.12 Remind Similar Expierence алгоритм	34
3.13 Підхід до навчання	35
3.13.1 Online Learning	35
3.13.2 Replay Learning (Experience Replay)	36
3.13.3 Гібридний підхід	36
3.14 Висновок до розділу	37
4 Опис програмної реалізації	39
4.1 DDQNwithGA бібліотека	39
4.1.1 DDQNwithGAModel (class)	40
4.1.1.1 Конструктор DDQNwithGAModel	41
4.1.1.2 Методи DDQNwithGAModel для управління Memory	41
4.1.1.3 Методи DDQNwithGAModel для навчання моделі	41
4.1.1.4 Методи DDQNwithGAModel для вибору дії	41
4.1.1.5 Методи DDQNwithGAModel для оновлення мережі	42
4.1.1.6 Інші методи	42
4.1.1.7 Поля і властивості DDQNwithGAModel	42
4.1.1.7.1 Поля та властивості для даних та стану	42
4.1.1.7.2 Поля для нейронної мережі	43
4.1.1.7.3 Поля для кастомізації поведінки	44
4.1.2 DDQNwithGACreator	44
4.1.2.1 Методи створення моделі	45
4.1.2.2 Методи клонування моделі	45
4.1.2.3 Внутрішні методи DDQNwithGACreator	45

4.1.3 Ресурси нейроної мережі.....	46
4.1.3.1 NNInference.....	46
4.1.3.1.1 Конструктор NNInference.....	46
4.1.3.1.2 Методи для прямого розповсюдження.....	46
4.1.3.1.3 Вибір відповіді	47
4.1.3.2 NNLayers	47
4.1.3.2.1 Поля та властивості	47
4.1.3.2.2 Конструктор.....	48
4.1.3.2.3 Методи	48
4.1.3.3 NNTrainWithSGDM.....	48
4.1.3.3.1 Поля.....	49
4.1.3.3.2 Конструктор.....	49
4.1.3.3.3 Методи	49
4.1.3.4 DDQNMemory	50
4.1.3.4.1 Поля.....	50
4.1.3.4.2 Конструктор.....	51
4.1.3.4.3 Методи	51
4.1.4 HyperparameterGen	51
4.1.4.1 Поля та константи	52
4.1.4.2 Конструктори	52
4.1.4.3 Методи	52
4.1.4.4 Enum GenHyperparameter	53
4.1.5 Інтерфейси	54
4.1.5.1 IDDQNwithGACritic.....	55
4.1.5.2 IDDQNwithGACustomRemindExperiencesDefinder.....	55
4.1.5.3 IDDQNwithGACustomRewardCalculator	56
4.1.6 Алгоритми для оптимізації навчання	57
4.1.6.1 RSEMethod.....	57
4.1.6.2 TDLearningMethod	58
4.1.6.3 Activation.....	59

4.1.6.4 Normalizer	60
4.1.6.5 Regularization	61
4.2 Симуляція.....	62
4.2.1 SimulationHandler	63
4.2.1.1 Поля.....	63
4.2.1.2 Методи	63
4.2.2 WorldResources	64
4.2.2.1 Cell	65
4.2.2.1.1 CellModel	65
4.2.2.1.2 CellActionHandler	67
4.2.2.2 Meteor	69
4.2.2.2.1 MeteorBlock	69
4.2.2.2.2 MeteorModel.....	69
4.2.2.3 WorldModel	70
4.2.2.3.1 Основні компоненти та функціонал	70
4.2.2.3.2 Управління часом та подіями	71
4.2.2.3.3 Взаємодія з клітинами	71
4.2.2.3.4 Візуалізація.....	71
4.2.2.4 WorldArea.....	71
4.2.2.4.1 Основні компоненти та функціонал	72
4.2.2.4.2 Ініціалізація та налаштування	72
4.2.2.4.3 Управління елементами на ігровому полі.....	72
4.2.2.4.4 Управління клітинами	73
4.2.2.4.5 Допоміжні функції.....	73
4.2.3 Constants	73
4.2.4 StatDatabase	74
4.2.4.1 Поля	74
4.2.4.2 Методи	74
4.2.5 Renderer	75
4.2.5.1 Основні компоненти та методи	75

	9
4.2.5.2 Синхронізація та візуалізація	76
4.3 Програма для тестування	76
4.3.1 ErrorStat Form	76
4.3.2 ErrorStatModel.....	77
5 Всесвіт та тестування.....	79
5.1 Всесвіт	79
5.1.1 Легенда візуалізації.....	79
5.1.2 Клітини.....	80
5.1.3 Час доби	83
5.1.4 Метеорити.....	83
5.1.5 Отруєні області.....	84
5.1.6 Приклад симуляції	84
5.2 Тестування	88
5.2.1 Порівняння алгоритмів.....	88
5.3 Висновок до розділу	93
6 Робота користувача з бібліотекою моделі	94
6.1 Вимоги до користувача.....	94
6.2 Інструкція для користувача.....	95
6.3 Висновок до розділу	98
Висновки	99
Перелік джерел посилань	100
Додаток А Текст програми.....	102
Додаток Б Слайди презентації	120

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

- DDPG – глибокому детермінованому політичному градієнті;
DDQN – подвійна глибока Q мережа;
GA – генетичний алгоритм;
HER – відтворенні досвіду з огляду;
RL – навчання з підкріпленням;
TDL – метод часових різниць.

ВСТУП

У сучасному світі швидкого розвитку технологій та збільшення обсягів обробки даних з'являється все більше викликів, які пов'язанні з оптимізацією процесів обчислень, зокрема, й у галузі штучного інтелекту та дослідження штучного життя. Генетичний алгоритм в поєднанні з нейронними мережами з навчанням з підкріпленням не лише відкриває нові можливості для розв'язання складних задач у сфері машинного навчання, а й прокладає шлях до глибших досліджень у галузі штучного життя та еволюції.

Ця робота присвячена розробці та аналізу моделі, яка інтегрує генетичний алгоритм у нейронні мережі з навчанням з підкріпленням для покращення процесу навчання та підвищення ефективності рішень, та вивченню перспектив моделювання процесів штучного життя й еволюції. Перший розділ присвячений визначенню актуальності та формулюванні умов задачі дослідження, включно з деталями дослідження штучного життя та еволюції. Задачею другого розділу є огляд літератури та аналіз існуючих рішень у галузі нейронних мереж із навчанням із підкріпленням, генетичних алгоритмів та їх застосування в моделюванні штучного життя. У третьому розділі описано теоретичні основи використаних методів, включно з, детальним описом розробленого алгоритму й механізмів навчання нейронних мереж з підкріпленням, а також їх роль у дослідженні еволюційних процесів і штучного життя. Четвертий розділ характеризує програмну реалізацію розробленої системи, зокрема, алгоритми, архітектуру, інтерфейс користувача й можливості моделювання еволюційних процесів. У п'ятому розділі представлений аналіз результатів тестування системи, в тому числі, оцінка ефективності інтеграції генетичного алгоритму в процесі навчання та його вплив на моделювання штучного життя й еволюційних процесів. У висновках основні результати дослідження та пропозиції з приводу подальшого використання та розвитку розробленої системи, а й перспективи її застосування у сфері штучного інтелекту, дослідження штучного життя та вивчення еволюційних алгоритмів.

1 ПОСТАНОВКА ЗАВДАННЯ ДОСЛІДЖЕННЯ

1.1 Постановка завдань дослідження

Мета даної дипломної роботи - розробка та дослідження моделі, що інтегрує генетичний алгоритм у нейронні мережі з навчанням із підкріпленням задля оптимізації процесу навчання й підвищення загальної ефективності алгоритмів у вирішенні складних задач.

Для досягнення поставленої мети були визначені наступні завдання:

- розробка програмної архітектури, що забезпечує інтеграцію генетичного алгоритму з нейронними мережами з навчанням із підкріпленням;
- створення ефективного генетичного алгоритму з метою оптимізації параметрів і структури нейронної мережі;
- реалізація системи навчання з підкріпленням, яка дозволяє нейронній мережі вирішувати задачі, ефективно взаємодіючи з середовищем;
- оцінка ефективності розробленої системи на зразку задачі, яка вимагає адаптації стратегій поведінки в динамічних умовах;
- аналіз можливостей моделювання процесів еволюції та штучного життя за допомогою розробленої системи.

Система має забезпечувати наступні можливості:

- ефективне навчання нейронної мережі для вирішення складних задач за допомогою генетичного алгоритму;
- візуалізація процесу навчання й розвитку нейронної мережі в реальному часі;
- моделювання процесів штучного життя та еволюційних стратегій у розв'язуванні задач;
- аналіз і порівняння ефективності різних параметрів генетичного алгоритму й конфігурацій нейронної мережі.

1.2 Висновки до розділу

В результаті було сформульовано умови задачі розробки та аналізу моделі з інтеграцією генетичного алгоритму в нейронні мережі з навчанням із підкріпленням, що дозволяє не тільки підвищити ефективність процесу навчання, а й відкриває можливості досліджувати штучне життя та еволюційні процеси. Ця модель зможе бути використана задля вирішення різних задач, в яких потрібен високий рівень адаптивності та оптимізації рішень в динамічних умовах, і також для подальших досліджень у галузі штучного інтелекту, штучного життя й еволюційних алгоритмів.

2 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

У цьому розділі розглядаються три наукові роботи, які пов'язані з оптимізацією гіперпараметрів за допомогою генетичних алгоритмів та глибокого навчання з підкріпленням у контексті робототехнічних маніпуляцій та інших різних задач.

2.1 Глибоке навчання з підкріпленням із застосуванням генетичного алгоритму для оптимізації параметрів

У роботі [1] розглядається інноваційний підхід до оптимізації параметрів у системах глибокого навчання з підкріпленням. Основною метою їх дослідження було підвищення ефективності навчання штучних інтелектуальних систем за допомогою поліпшення процесу підбору оптимальних параметрів.

Автори здійснюють пошук оптимальних рішень шляхом мутації, генерації та комбінування існуючих параметрів, імітуючи процеси еволюції й природного відбору. Пропонуючи ефективні рішення для складних проблем, цей метод показує високу адаптивність до різноманітних завдань глибокого навчання.

Їх дослідження продемонструвало значне поліпшення продуктивності навчальних алгоритмів за рахунок точного налаштування параметрів, що дозволяє досягнути кращих результатів у більш короткі строки. З'ясувалося, що застосування генетичного алгоритму є особливо зручним у задачах, де традиційні методи оптимізації не забезпечують необхідної ефективності через велику кількість параметрів та їхні складні взаємозв'язки [2].

Важлива перевага описаного їми підходу - його універсальність та перспектива використання в різноманітних галузях, включно з робототехнікою, фінансовою аналітикою, медичним діагностуванням та іншими областями, в яких застосовуються комплексні системи штучного інтелекту. Результати їх дослідження відкривають нові шляхи для

розроблення більш дієвих і адаптивних систем глибокого навчання, що будуть здатні самостійно оптимізувати свої параметри, щоб досягнути найкращих результатів у різних задачах.

2.2 Автоматична оптимізація гіперпараметрів за допомогою генетичного алгоритму в глибокому навчанні з підкріпленням для робототехнічних маніпуляцій

У роботі [2] дослідники запропонували метод, що базується на глибокому детермінованому політичному градієнті (DDPG) та відтворенні досвіду з огляду (HER), що застосовує генетичний алгоритм (GA) задля точного налаштування значень гіперпараметрів. Аналіз їх результатів показав помітне підвищення ефективності, продуктивності та зменшення часу навчання. Також, завдяки порівнянням дослідники надали докази того, що GA+DDPG+HER є кращим за існуючі методи.

В дослідженні навчальні агенти можуть застосовувати навчання з підкріпленням (RL) задля вибору своїх дій, користуючись функцією винагороди. Однак, процес навчання має значний вплив на вибір значень гіперпараметрів, які використовують в алгоритмі навчання. Із цього випливає необхідність покращення алгоритму RL. Хоча алгоритми можуть бути корисними для вирішення певних проблем, вони не дуже зручні в складних ситуаціях, як-от роботичні завдання, які використовують RL через відсутність зрозумілого зв'язку між зміною налаштувань гіперпараметрів і змінами продуктивності.

У цій роботі генетичний алгоритм (GA) застосовується для оптимізації гіперпараметрів, що може бути використано задля оптимізації гіперпараметрів у порівняних типах задач. GA здатний налаштовувати гіперпараметри лише один раз, а потім користуватися ними нескінченно в подальших навчаннях RL, зберігаючи обчислювальні ресурси, які інакше витрачалися б кожного разу, як RL застосовують до агента.

Основний вклад цього дослідження включає розробку нового алгоритму, щоб автоматизувати налаштування гіперпараметрів. Ця робота робить великий внесок у дослідження оптимізації в галузі глибокого навчання з підкріпленням, зокрема, використання генетичного алгоритму, щоб підвищити ефективність та зменшити кількість часу навчання в завданнях роботичної маніпуляції.

2.3 Нур-RL: Оптимізація гіперпараметрів за допомогою навчання з підкріпленням

У роботі[3] дослідники вивчають проблему оптимізації гіперпараметрів як послідовну задачу прийняття рішень про те, який гіперпараметр тестувати далі, і звертаються до неї за допомогою навчання з підкріпленням. Експерименти на великому наборі з 50 наборів даних показують, що наш метод має перевагу над сучасними підходами до навчання гіперпараметрів.

Оптимізація гіперпараметрів грає велику роль в основній продуктивності моделей машинного навчання й може бути головним чинником, що визначає, чи навчена модель виявиться прогресивною або просто помірною. Розміри наборів даних та складності моделей, що постійно зростають, спричиняють збільшення часу навчання й вимагають більше ресурсів. Це робить оптимізацію гіперпараметрів ще більш складним завданням. При автоматизації налаштування гіперпараметрів практичні рішення мають бути перевіреними, масштабованими та зберігати низький обчислювальний бюджет, а також паралельно забезпечувати високу кінцеву продуктивність [3].

Дослідники моделюють проблему оптимізації гіперпараметрів як упорядковану задачу прийняття рішень про те, який гіперпараметр тестувати далі, і підходять до неї з боку навчання з підкріпленням. Це відкриває їх моделі можливість не покладатися на евристичну функцію придбання, як у

випадку з послідовною модельно-базованою байєсовою оптимізацією (SMBO), а вчитися, які гіперпараметри слід тестувати далі, на основі того, яке зменшення втрати валідації вони зрештою спричинять. Їх спосіб показує помітне підвищення продуктивності та зменшення кількості часу навчання на широкому спектрі завдань маніпуляції з роботами. Більш того, вони, порівнюючи, наводять докази, що їх підхід (GA+DDPG+HER) найкращий серед існуючих методів [3].

2.4 Висновки до розділу

У цьому розділі розглядаються три наукові роботи, що присвячені оптимізації гіперпараметрів у контексті глибокого навчання з підкріпленням і застосування генетичних алгоритмів. Аналіз цих досліджень продемонстрував великий потенціал інтеграції генетичних алгоритмів у процеси навчання з підкріпленням для ефективного вибору оптимальних параметрів систем штучного інтелекту.

Перша праця показує, як саме генетичний алгоритм можна використовувати, щоб автоматизувати та оптимізувати вибір гіперпараметрів, що значно збільшує ефективність навчання штучних інтелектуальних систем. Друге дослідження підтверджує, що використання генетичного алгоритму разом із методом глибокого детермінованого політичного градієнту та відтворенням досвіду з огляду значно покращує продуктивність та зменшує кількість часу навчання в завданнях роботичної маніпуляції. Третя стаття є інновативною, так як формулює оптимізацію гіперпараметрів як задачу навчання з підкріпленням. Це дозволяє системі вчитися обирати гіперпараметри, що найкраще впливають на зниження втрати валідації.

Узагальнюючи, ці дослідження описують потенціал застосування генетичних алгоритмів і навчання з підкріпленням для покращення процесів автоматичного налаштування гіперпараметрів у системах штучного інтелекту. Такі методи можуть мати широке застосування в різних галузях,

враховуючи, обробку мови робототехніку, розпізнавання образів та інші сфери, в яких значну роль відіграє високий рівень адаптивності.

3 ІНСТРУМЕНТИ ТА АЛГОРИТМИ

Програмний застосунок, написаний на C# в операційній системі Windows. В якості середовища розробки було вибрано Microsoft Visual Studio. Для тестування я використав MS SQL для збереження покрокових результатів та розроблений мною додаток на Win Forms для відображення цих даних на графіці.

В реалізованій мною моделі нейронної мережі я використав такі моделі, алгоритми та методи:

Double Deep Q-Network (DDQN) та Temporal Difference Learning (TD Learning) - для визначення та оновлення стратегії вибору дій на основі нагороди та оновлення цільової мережі [4].

Метод Reinforcement Similar Experiences (RSE) - для визначення найбільш подібних досвідів та їх нагадування задля покращення процесу навчання [5].

Стратегія Epsilon-greedy - для балансу між використанням вже відомих стратегій та дослідженням нових дій [6].

Система обчислення нагороди - використовується власний калькулятор нагород для оптимізації рішень на основі отриманих нагород.

Алгоритми генетичного програмування - для оптимізації гіперпараметрів мережі та методів навчання, відбору агентів з успішними рішеннями та схрещення згадок(Memory).

Формули нормалізації - використовуються для обробки нагород та інших значень з метою покращення процесу навчання, а саме MinMax та Tanh.

Оптимізатор – Stochastic Gradient Descent with Momentum для прискорення та стабілізації навчання.

Регуляризатори – Dropout, L2 та Weights Noise використовуються для запобігання перенавчанню.

Функція активації – Swish додає необхідну нелінійність до моделі, що дозволяє нейронній мережі навчатися складнішим та більш абстрактним відносинам між вхідними даними та виходом.

3.1 Середовище розробки Microsoft Visual Studio та C#

3.1.1 Microsoft Visual Studio

Microsoft Visual Studio є простим, але ефективним редактором коду, який підтримує різні мови програмування та інструменти, зокрема, C#. Завдяки своїй гнучкості та великому вибору розширень, Microsoft Visual Studio дає можливість легко налаштовувати середовище під конкретні вимоги проекту [5].

Інтеграція із системами контролю версій, наприклад, Git дозволяє легко керувати версіями коду безпосередньо з редактора, включно з комітами, пушами, пулами й переглядом змін. Також наявність вбудованих терміналів і, розширений для роботи з кодом, інструментарій роблять Microsoft Visual Studio хорошим вибором задля розробки алгоритмів штучного інтелекту. За допомогою підтримки великої кількості розширень та інструментів, які призначені для особливих потреб розробників ІІІ, таких як інтеграція з системами машинного навчання та підтримка контейнерів Docker, використання Microsoft Visual Studio як середовища розробки надає додаткову гнучкість.

Крім того, Microsoft Visual Studio є безкоштовним для використання та в той час залишається нескладним і швидким, забезпечуючи високу продуктивність.

3.1.2 C#

C# оптимізований для високої продуктивності. Це є критично важливим для обробки великих обсягів даних та виконання складних обчислень, що часто зустрічаються в проектах зі штучним інтелектом. Строга

типізація та сучасні функції мови C# допомагають написати безпечний та надійний код, при цьому зменшуючи ймовірність помилок та недоліків.

C# має велику екосистему й спільноту. Саме це забезпечує доступ до широкого спектра інструментів, фреймворків та бібліотек. Таким чином, ви маєте змогу швидко знайти готові рішення або бібліотеки для реалізації специфічних аспектів ваших проєктів, наприклад, обробки даних, математичних операцій, а також спеціалізовані бібліотеки для роботи з нейронними мережами та генетичними алгоритмами [6].

Одна з головних причин використання C# - ця мова ефективно підтримує багатопоточність, що є дуже важливим для розробки високопродуктивних додатків у області штучного інтелекту. Багатопоточність дозволяє більш ефективно використовувати ресурси процесора під час тренування моделей або обробки великих масивів даних.

C# та .NET надають розробникам потужні засоби для створення безпечного коду, в тому числі, за допомогою механізмів управління доступом, шифрування та безпечного управління пам'яттю. Це дає змогу мінімізувати ризики, які пов'язані з виконанням коду й обробкою даних.

C# та .NET оптимізовані для високої продуктивності, що є важливим при розробці обчислювально-інтенсивних додатків, якими є системи на основі нейронних мереж і генетичних алгоритмів. Результативне управління пам'яттю й оптимізовані алгоритми в .NET дозволяють досягати високої швидкодії [6].

3.2 Double Deep Q-Network

DDQN модель, або подвійна глибока Q-мережа (Double Deep Q-Network), є значним удосконаленням традиційної глибокої Q-мережі (DQN), що використовується в галузі глибокого навчання з підкріпленням. Головною інновацією DDQN є розробка механізму, що сприяє зменшенню проблем, пов'язаних із переоцінкою дій, які доволі часто спостерігаються в DQN. У традиційних DQN-архітектурах одна й та ж нейронна мережа здійснює дві

функції, а саме: вибір дії та оцінка цінності цієї дії. Такий підхід призводить до занадто оптимістичних оцінок цінності дій, що у свою чергу може мати негативний вплив на процес навчання [4].

DDQN вирішує цю проблему, використовуючи дві окремі мережі: мережа політики (Policy Network) - обирає дію, яка повинна бути виконана, мережа цінності (Value Network) - оцінює цінність цієї дії, застосовуючи відмінну від мережі політики вагову матрицю.

Цей спосіб дозволяє DDQN зменшити ризик переоцінки дій, оскільки оцінка цінності дії ґрунтується не на тих же вагах, що й вибір дії. Отже, DDQN забезпечує більш точну та об'єктивну оцінку дій, що, в свою чергу, сприяє більш стабільному й дієвому процесу навчання.

До переваг можна віднести: зменшення переоцінки дій: алгоритм зменшує загрозу переоцінки дій, а це є ключовим для оптимізації стратегій поведінки агента; більш стабільне навчання: розділення процесу вибору дії та оцінки її цінності дає можливість досягнути більшої стабільності процесу навчання; покращена збіжність: DDQN часто досягає ліпшої збіжності в порівнянні з традиційними DQN моделями, що й призводить до вищого рівня ефективності вирішення задач [4].

DDQN особливо продуктивний у динамічних та непередбачуваних середовищах, де для успішного навчання критичне значення має точність оцінки дій. Це може включати оптимізацію стратегій у фінансових моделях, симуляції реального світу для робототехніки або складні ігрові середовища. Використання DDQN може привести до покращення розуміння динаміки середовища та досягнення вищих результатів, ніж із традиційними методами.

В якості бази для створення DDQNwithGA була обрана саме ця модель через її універсальність і ефективність. У розробленій мною моделі цільова нейромережа оновлюється вже в нових поколіннях агентів. Це означає, що агент протягом свого життя навчає онлайн-мережу і відбирається за рахунок генетичного алгоритму як найбільш успішний екземпляр. Тоді його нащадки переносять його досвід, набуті навички й здійснюють апгрейд цільової мережі на основі цієї онлайн-мережі. Крім того, при схрещенні змішуються

згадки(Memory) агента батька й агента матері, завдяки чому під час навчання з використанням replay learning результати завжди будуть хоч трохи відрізнятися. Це робить алгоритм дієвішим і допомагає зменшити ризик потрапляння в локальні мінімуми [5].

3.3 Алгоритм генетичного програмування

Генетичний алгоритм — це метод оптимізації, створений під впливом процесів природного відбору, які відбуваються в біології. Цей метод використовується для автоматичного поліпшення нейронних мереж із плином часу, імітуючи процес еволюції [6].

Генетичні алгоритми (рис.3.1) працюють за допомогою створення популяції кандидатів (у цьому контексті — нейронних мереж), і потім відбору, схрещування й мутації серед цих кандидатів задля створення нових поколінь. Ціллю є знаходження найкращої можливої конфігурації нейронної мережі для даної задачі. Основні кроки процесу [6]:

- ініціалізація: створення випадкової початкової популяції нейронних мереж (або індивідів);
- оцінка: кожен індивід (нейронна мережа) у популяції оцінюється відповідно до функції пристосованості, яка визначає, наскільки добре мережа розв’язує поставлену задачу;
- відбір: найкращі індивіди обираються як основа для створення наступного покоління, вибір часто ґрунтується на принципах природного відбору, згідно з якими "сильніші" індивіди мають більше шансів на репродукцію;
- схрещування: комбінування вибраних індивідів, щоб створювати нащадків, це може включати обмін частинами їхньої "генетичної" структури (наприклад, параметрами нейронної мережі);
- мутація: з малою ймовірністю в нащадках можуть відбуватися випадкові зміни (мутації) для забезпечення генетичного різноманіття;

– нове покоління: створене нове покоління замінює старе, і цикл повторюється з кроку 2.

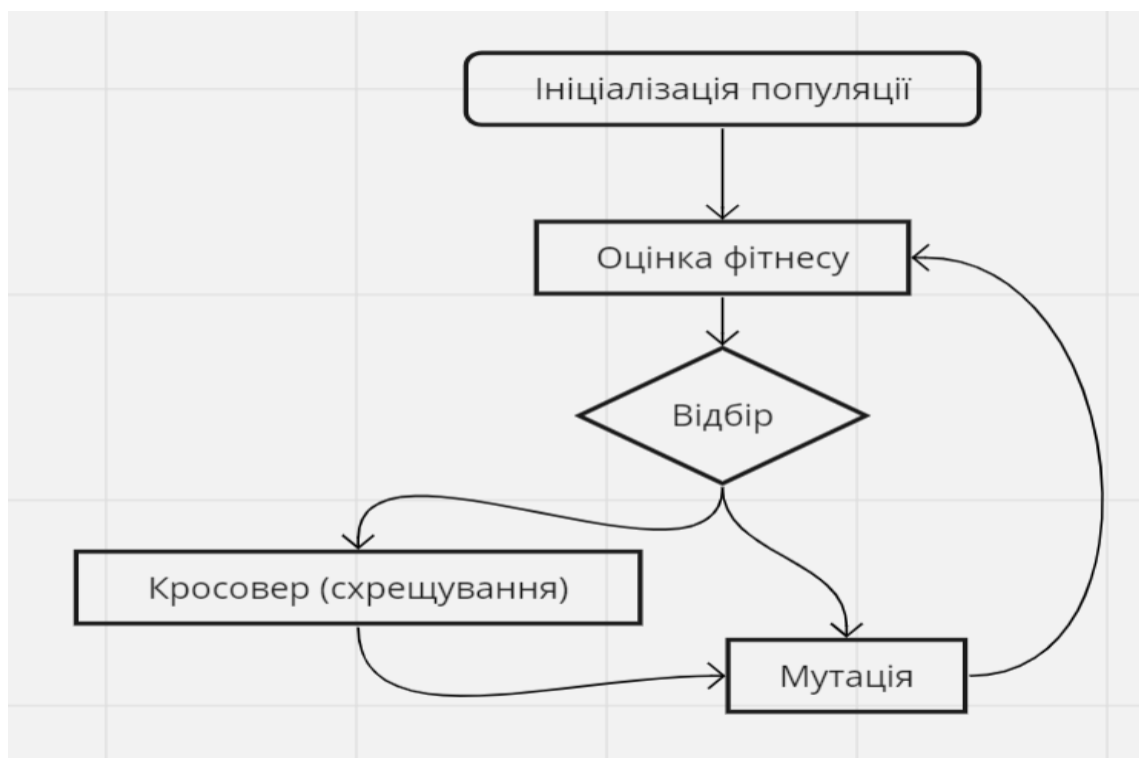


Рисунок 3.1 – Архітектура генетичного алгоритму

У розробленому варіанті цей алгоритм трохи відрізняється. Агент має змогу обирати, хоче він розвиватись, використовуючи тільки мутацію, чи спочатку схреститись із іншим агентом. Таким чином, у випадках, де мала популяція агентів, для деяких завдань, де потрібне детальне знання про область проблеми, де кросовер може призвести до створення нежиттєздатних рішень, які не мають сенсу в контексті завдання, у випадках, коли область пошуку є дуже вузькою або коли оптимальне рішення знаходиться в дуже конкретній частині простору пошуку, використання кросовера може розсіювати пошук і відволікати від знаходження оптимального рішення, агент має альтернативу кросоверу і обирає більш вигідну стратегію еволюції за рахунок відбору. Крім того, користувач бібліотеки з моделю за власним бажанням може обмежувати розвиток тільки тим чи іншим шляхом, або поєднувати їх у якомусь іншому вигляді.

Цей процес повторюється впродовж багатьох поколінь, із кожним новим, потенційно, покращуючи здатність вирішувати цільову задачу до тих пір, поки не буде досягнуто задовільного рішення або не буде вичерпано час чи ресурси. За замовчуванням агент буде прагнути виконати дії, які не тільки сприяють досягненню цілей епізоду, але й здійснюються з мінімальною кількістю похибок і високою ефективністю, щоб максимізувати загальну нагороду. Це стимулює агента до постійного вдосконалення стратегій поведінки та адаптації до змінних умов середовища.

Хоч генетичні алгоритми і є універсальними та можуть застосовуватися до широкого спектру оптимізаційних задач, вони можуть бути не найпродуктивнішими в особливих обставинах. У динамічних середовищах, особливо, де випадкові ситуації мають чималий вплив на агентів, генетичний алгоритм підбирає найбільш оптимальне, проте не найдієвіше рішення. Агент, навчений лише за рахунок генетичного алгоритму, не може аналізувати середовище, а тільки реалізовує дію, що задана поточним геном. Тому генетичний алгоритм, у розробленій мною моделі, не застосовується для вибору рішення. Він використовується для підбору гіперпараметрів, де не вигідні зміни не зберігаються за рахунок відбору кращих варіантів. Це зменшує ефективність навчання на початку та збільшує коливання, однак, набагато полегшується підбір архітектури, і з'являється можливість ускладнити нейронну мережу, що робить її більш автономною. У перспективі рішення агентів більш варіативні, і в деяких задачах з ризиком великої кількості локальних мінімумів динамічна зміна гіперпараметрів дозволяє значно швидше їх подолати.

Крім того, при схрещуванні, крім гіперпараметрів, схрещуються згадки(Memory). Згадки(Memory) використовуються для навчання на отриманому агентом досвіді. Схрещування дає змогу швидко підвищити різноманітність учбових даних і дати кожному нащадку різну базу для навчання, за рахунок чого їх стратегії трохи відрізняються, особливо це стосується задач, де можливо зберігати дуже обмежений набір даних. Це надає

можливість націлити кожного агента на дослідження трохи різних стратегій, що в перспективі прискорює розвиток популяції загалом.

3.4 Temporal Difference Learning

Temporal Difference Learning (TD Learning) є фундаментальною концепцією у сфері машинного навчання з підкріпленням, яка зосереджується на вивченні того, як передбачати майбутню винагороду, ґрунтуючись на поточних оцінках, без необхідності очікувати на фінальний результат. Цей метод використовує різницю між послідовними прогнозами (тобто тимчасову різницю), щоб коригувати оцінки вартості станів або дій. Це дозволяє агенту поступово поліпшувати своє розуміння середовища, навіть під час перебування в процесі виконання завдань [7].

Використання TD Learning має значні переваги, бо інформація про винагороду може стати доступною не відразу, а розподіляється в часі. Такі обставини часто можна зустріти в комплексних середовищах, в яких дії агента на початкових етапах можуть впливати на винагороди, які він отримає в майбутньому. TD Learning дозволяє агенту навчатися з кожним кроком, адаптуватися та оптимізувати свою стратегію в реальному часі, і це значно підвищує ефективність навчання [7].

Однією з головних переваг TD Learning є його вміння вчитися "на льоту", що робить його особливо вигідним задля розробки автономних систем, які мають швидко адаптуватися до змін у середовищі. Цей метод дозволяє агенту краще усвідомлювати довгострокові наслідки своїх дій, використовуючи оцінку винагороди, що покращує стратегічне планування та прийняття рішень.

Включення TD Learning підходить для розв'язування важких завдань, де потрібно робити прогнозування та адаптацію в реальному часі, і є важливим рухом до створення більш глибоких та дієвих систем машинного навчання.

Проблема цього алгоритму - складність визначення оптимального гіперпараметру `discount`, що відповідальний за передбачувану нагороду. Через неадекватну оцінку своїх дій надмірно велике або навпаки занадто мале значення можуть тільки заважати агенту при вирішенні поставленої задачі. За рахунок використання генетичного алгоритму у визначенні гіперпараметрів значення `discount` підбирається динамічно протягом розвитку агенту й гнучко підлаштовуються під визначену задачу. Також, завдяки відбору різних успішних агентів із різними значеннями цього гіперпараметру кількість різноманітних стратегій збільшується. Це помітно покращує якість результату навчання [8].

3.5 Система обчислення нагороди

Механізм нагород є ключовим елементом для навчання агента в рамках парадигми підкріплювального навчання. Основна думка полягає в тому, що агент отримує нагороди або покарання за виконання дій в залежності від їх впливу на досягнення мети. Такий спосіб дозволяє агенту навчатися на власному досвіді й вибудовувати власну стратегію поведінки, яка максимально збільшить сумарну винагороду [9].

Агент навчається в середовищі, де нагороди отримуються за досягнення певних цілей, при цьому максимізуючи продуктивність своїх дій. Якщо епізод завершений (`done == true`), то загальна нагорода (`reward`) розраховується як середнє значення нагород за всі дії в епізоді (`TotalReward / TotalActionsNum`). Якщо ця середня нагорода є позитивною, вона коригується з урахуванням кількості вдалих дій в епізоді (`episodeSuccessValue`), застосовуючи гіперпараметри `GenHyperparameter.genDoneBonusA` і `Gen.Hyperparameter.genDoneBonusB` для розрахунку бонусу за завершення. Додатково до загальної нагороди додається бонус (`bonus`) [9].

Завдяки використанню гіперпараметрів агент може точніше визначати стратегію розвитку й уникати локальних мінімумів. Це підштовхує агента до

різноманітніших стратегій. Наприклад, у створеній мною симуляції, за рахунок цих гіперпараметрів агенти поділялись на кілька різновидів: деякі намагались зробити не багато дій, але з високим результатом. Якщо їм це не вдавалося, вони оновлювали спробу, а інші - навпаки отримували більшу нагороду за накопиченні бали, здійснюючи велику кількість дій, і потім помножували їх, завершуючи епізод.

Якщо епізод не завершений, то поточна нагорода розраховується як різниця між поточною цінністю цілі (`targetValue`) і цінністю цілі перед дією (`TargetValueBeforeAction`). Якщо дія була хибною (`IsActionError == true`), із нагороди віднімається штраф, в інакшому випадку – додається бонус за правильну дію. Обидва параметри теж залежать від гіперпараметрів (`errorFine`, `correctBonus`).

`IsActionError` визначається завдяки критику, який за бажанням може створити за допомогою інтерфейсу (`IDDQNwithGACritic`) користувач для своїх задач, якщо ж інтерфейс не реалізован, то цей параметр ігнорується. Так само за допомогою інтерфейсу (`IDDQNwithGACustomRewardCalculator`) користувач може змінити цілі навчання агента.

За замовчуванням агент буде прагнути до виконання дій, що не лише сприяють досягненню мети епізоду, але й здійснюються з мінімальною кількістю помилок і високою дієвістю, задля максимізування всієї нагороди. Це змушує агента постійн вдосконалювати стратегії поведінки та пристосування до непостійних умов середовища.

3.6 Нормалізація MinMax та Tanh

Нормалізація даних - важливий крок у передобробці даних, особливо у машинному навчанні та аналізу даних. Вона допомагає збільшити швидкість та стабільність навчання алгоритмів і також зменшує вирогідність того, що деякі характеристики (ознаки) можуть бути недооцінені через їх масштаб. Два популярні методи нормалізації — це нормалізація за допомогою гіперболічного тангенса (`Tanh`) та `MinMax` нормалізація [10].

3.6.1 MinMax Нормалізація

MinMax нормалізація — це метод масштабування, який трансформує данні, зміщуючи мінімальне значення кожної з характеристик до 0, а максимальне — до 1. Формула для MinMax нормалізації виглядає наступним чином: $\text{min_max_norm} = (X - X_{\min}) / (X_{\max} - X_{\min})$ [11].

Метод використовується для масштабування числових даних, коли потрібно зберегти точні нульові значення та розподілити характеристики між 0 та 1. Це особливо корисно в умовах, коли алгоритми навчання вразливі до масштабу характеристик, наприклад, в нейронних мережах, алгоритмах, що використовують метрики відстані, тощо [11].

3.6.2 Tanh Нормалізація

Нормалізація за допомогою гіперболічного тангенса (Tanh) — це метод, який використовує гіперболічну функцію тангенса для масштабування значень характеристик таким чином, що результати потрапляють в діапазон від -1 до 1. Формула має вигляд: $\text{tanh_norm} = (\exp(a \cdot X) - \exp(-a \cdot X)) / (\exp(a \cdot X) + \exp(-a \cdot X))$ [12].

Такий метод використовують для масштабування даних, коли потрібно, щоб середнє значення було близьким до 0, що може сприяти більш швидкій конвергенції під час навчання нейронних мереж. Tanh нормалізація особливо доцільна, коли дані мають приблизно симетричний розподіл навколо нуля.

3.7 Stochastic Gradient Descent з Momentum

Stochastic Gradient Descent (SGD) з Momentum є поширеним варіантом алгоритму стохастичного градієнтного спуску, який застосовується, задля оптимізування в машинному навчанні та глибокому навчанні. Momentum — це гіперпараметр, який дозволяє алгоритму швидше збігатися до мінімуму

функції втрат, враховуючи попередні рухи для визначення напрямку наступного кроку. Це дозволяє "перескакувати" локальні мінімуми [13].

Stochastic Gradient Descent (SGD) - це метод оптимізації, що використовується для зведення до мінімуму функції втрат шляхом ітеративного оновлення ваг моделі в напрямку найбільшого спаду функції втрат. Він робить це, використовуючи тільки малу вибірку даних (міні-батч), що робить алгоритм ефективнішим за пам'яттю та більш швидким за класичний градієнтний спуск [13].

Momentum є модифікацією SGD, яка додає поняття "інерції" до процесу навчання, використовуючи експоненційно згладжений середній попередніх оновлень ваг. Це дозволяє зменшити коливання в напрямках зі змінним градієнтом алгоритму та пришвидшитися в напрямках із постійним градієнтом. З використанням цього оптимізатора агенти навчаються швидше, що дуже важливо у великих та складних моделях. За рахунок поліпшення вміння агента дієво оптимізувати свою політику поведінки в складних середовищах збільшуються його шанси на успіх. Метод ефективен у фазах, де градієнт функції втрат дає змогу швидко рухатися до глобального мінімуму. Це робить його вигідним для використання в симуляторах еволюції, в яких швидкість збіжності є критичною [13].

Основною проблемою цього алгоритму є додатковий гіперпараметр Momentum, оскільки він потребує додаткової логіки динамічної зміни в залежності від етапу розвитку агенту, тому зазвичай використовуються універсальніші та легші в налаштуванні оптимізатори, такі як Adam. Проте, завдяки генетичному алгоритму, який за рахунок відбору відокремлює неактуальні значення від актуальних ця проблема має менший вплив на складність роботи моделі [13].

3.8 Регуляризатори Dropout, L2, Weights Noise

Регуляризація є головною деталлю в ході тренування нейронних мереж, що сприяє запобіганню перенавчанню та покращує загальну здатність

моделі генералізувати на нових даних. Серед численних методів регуляризації особливо популярними є Dropout, L2 регуляризація та внесення шуму в ваги (Weights Noise). Усі ці методи мають свої характерні риси та області примінення.

3.8.1 Dropout

Dropout — це техніка регуляризації, що полягає у випадковому відключенні (обнуленні) частини нейронів мережі впродовж тренування. Це запобігає умовам, коли нейрони пристосовуються до дуже своєрідних особливостей тренувального набору даних, спонукаючи мережу розподіляти вагу інформації між різними нейронами, і, тим самим, поліпшуючи загальне вміння моделі до генералізації. Dropout зазвичай застосовують у глибоких нейронних мережах і великих архітектурах [14].

3.8.2 L2 Регуляризація (Ridge Regression)

L2 регуляризація, також відома як регуляризація Тихонова або Ridge Regression, полягає у додаванні до функції втрат квадратів ваг моделі, помножених на регуляризаційний коефіцієнт. Цей метод штрафує великі ваги та сприяє тренуванню більш гладких моделей, ваги яких змінюються не дуже різко. L2 регуляризація може робити модель менш чутливою до невеликих флуктуацій у тренувальних даних і запобігати перенавчанню [14].

3.8.3 Weights Noise (Шум в Вагах)

Внесення шуму у ваги (Weights Noise) — це метод, суть якого полягає у додаванні випадкового шуму до ваг нейронної мережі під час тренування. Цей спосіб може допомогти моделі краще генералізувати, змушуючи її вчитися на різноманітних варіаціях тренувальних даних. Шум може бути

доданий як до вхідних даних, так і безпосередньо до ваг. Таким чином, модель стане більш стійкою до невеликих змін або шуму в даних [14].

3.8.4 Поєднання методів регуляризації

Одночасне застосування різних методів регуляризації може бути дуже результативним для покращення загальної здатності моделі до генералізації та її стійкості до шуму в даних. Однак, це також вимагає ретельного налаштування гіперпараметрів та може збільшити обчислювальні витрати. Важливо знайти певний баланс між різними методами регуляризації, щоб максимально збільшити ефективність тренування та уникнути потенційних пасток, таких як недостатнє навчання або надмірно великі обчислювальні витрати. У розробленій мною моделі цю проблему вирішує генетичний алгоритм, тому реалізація такого поєднання регуляризаторів несе менше загроз і дієво виконує свою задачу. Варіативність регуляризаторів дає можливість запобігти перенавчанню агента й навчити його аналізувати схожий досвід, що дуже важливо в системах з постійним навчанням.

3.9 Функція активізації Swish

Функція активізації Swish є відносно новою і була запропонована командою дослідників Google. Вона є гладкою, постійною і не монотонною функцією, яка в останні роки стала відомою в глибинному навчанні. Функція Swish визначається як: $\text{Swish}(x) = x * \text{sigmoid}(\beta * x)$; β є параметром, що може бути як константою, так і навчальним параметром. β може бути адаптовано під час процесу навчання, що потенційно поліпшує продуктивність моделі. β надає додаткову гнучкість, але в той самий час потребує додаткового налаштування, що може ускладнити хід тренування [15].

Серед переваг можна відокремити такі риси, як: гладкість - Swish є гладкою функцією, тому вона є добре підходящою для оптимізації, оскільки градієнти можуть бути ефективно розповсюджені назад через мережу;

немонотонність - на відміну від багатьох традиційних функцій активації, Swish не є монотонною, і саме це робить її корисною в певних архітектурах нейронних мереж при захопленні складніших залежностей у даних.

3.10 Ініціалізатор He

Ваги ініціалізуються випадково з Гаусівського (нормального) розподілу із середнім 0 і дисперсією. Шляхом забезпечення кращого початкового розподілу ваг, He initialization має змогу прискорити конвергенцію тренування, дозволяючи мережі швидше досягнути оптимальніших рішень. Стартовий розподіл ваг, який є ні надто великим, ні надто малим дозволяє уникнути перенавчання на перших етапах тренування.

Ініціалізатор не широко використовується в сучасних глибоких нейронних мережах, зокрема тих, що користуються ReLU або схожими до нього активаційними функціями. Його ефективність у забезпеченні швидкого і стабільного навчання сприяє його популярності серед дослідників і розробників у галузі машинного навчання [16].

3.11 Epsilon-greedy стратегія дослідження

Epsilon-greedy (ϵ -greedy) стратегія є одним з найлегших, але водночас дієвих підходів до балансування між дослідженням (exploration) та використанням (exploitation) у ході навчання алгоритмів машинного навчання, особливо в задачах, пов'язаних із підсилювальним навчанням [17].

Epsilon-greedy стратегія працює шляхом вибору між дослідженням нових дій та використанням вже відомих з певною випадковістю. Параметр ϵ (epsilon), що як правило є невеликим значенням між 0 і 1, визначає ймовірність, з якою алгоритм обере дослідження над використанням. Отже, з ймовірністю ϵ алгоритм випадково обере будь-яку можливу дію, а з ймовірністю $1-\epsilon$ він вибере найкращу дію на основі поточних знань. Ця стратегія дозволяє агенту встановлювати баланс між вивченням нових дій,

які можуть принести більший виграш у майбутньому, та використанням вже відомих дій, які забезпечують найкращий виграш за поточним станом.

Зміною параметра ϵ можна просто контролювати баланс між дослідженням нових дій та використанням уже наявних. Знаходження ідеального значення ϵ може бути важким завданням, оскільки воно значно залежить від конкретної задачі та її динаміки. Це питання вирішується динамічною зміною за допомогою еволюції генів, і під кінець навчання ϵ , зазвичай, починає зменшуватись. Таким чином, популяція агентів сама регулює зацікавленість у дослідженні й не вимагає додаткової логіки.

3.12 Remind Similar Experience алгоритм

Алгоритм "Remind Similar Experience" (RSE) використовує попередні спроби для поліпшення процесу навчання. Концепція "Remind Similar Experience" заснована на ідеї, що агенти або моделі можуть покращити своє навчання та прийняття рішень, згадуючи та аналізуючи аналогічні ситуації з минулого. Це дозволяє використовувати накопичений досвід для оптимізації дій у подібних майбутніх сценаріях. У моєму коді реалізован варіант цього алгоритму з назвою Кейс-базове Міркування (Case-based Reasoning) [17].

Кейс-базове Міркування (Case-based Reasoning) – це використання бази даних із минулими випадками задля вирішення нових задач шляхом визначення й аналізу схожих випадків. Використання схожих досвідів може допомогти моделям швидше пристосуватися до нових ситуацій, застосовуючи вже накопичені вміння. Регулярний перегляд попереднього досвіду може допомогти утримувати важливу інформацію та зменшити ефект забування.

В розробленому коді RSE використовується під час обранні кожної з дій. Нагадування спрацьовує тільки тоді, коли кількість прикладів перевищує задану користувачем з певною ймовірністю. Ця вірогідність є гіперпараметром та обирає випадкову згадку з тих, що відсотково найбільш схожі. Відсоток, з яким згадка обирається випадково теж є гіперпараметром.

Визначення продуктивної метрики схожості може бути складним, оскільки вона має адекватно демонструвати сутність подібності в контексті даної задачі. Оцінка схожості між поточним станом і станами в пам'яті відтворення дозволяє ідентифікувати найбільш релевантні досвіди. Це може бути виконано за допомогою різних метрик, але за замовчування це розраховується як Евклідова відстань між векторами станів. Користувачі бібліотеки можуть реалізувати інтерфейс (IDDQNwithGACustomRemindExperiencesDefinder) і користуватися власним способом розрахунку схожого стану.

3.13 Підхід до навчання

Online learning та replay learning (часто згадується як experience replay) є двома підходами до навчання в контексті машинного навчання та підсилювального навчання. Обидва підходи мають свої характерні риси та застосовуються в залежності від конкретних умов задачі й обмежень [12].

3.13.1 Online Learning

Online learning відноситься до процесу навчання, де модель оновлюється крок за кроком в реальному часі або після кожного окремого спостереження/дії. Це означає, що система вчиться з кожним новим зразком даних без потреби перенавчати модель з нуля. Цей спосіб є результативним у сценаріях, де дані знаходяться послідовно або де необхідно швидко реагувати на зміни у вхідних даних [12].

До переваг Online Learning можна віднести вміння швидко адаптуватися до нових даних. Однак така модель дуже чутлива до шуму в даних та має великий ризик забути старий досвід через постійне оновлення моделі [12].

3.13.2 Replay Learning (Experience Replay)

Experience replay є підходом, що заснований на запам'ятовуванні попередніх досвідів агента (наприклад, у вигляді пар стан-дія-винагорода) та їх застосуванні в процесі навчання. Цей метод дозволяє ефективно використовувати обмежені досвіди, повторно використовуючи їх задля тренувань, що покращує стабільність та продуктивність навчання [12].

Серед переваг можна виокремити зниження кореляції між послідовними тренувальними прикладами, що покращує стабільність навчання; можливість ефективнішого використання обмежених досвідів. Проте є також і недолік - метод потребує великий об'єм пам'яті [12].

3.13.3 Гібридний підхід

У розробленому алгоритмі використовується гібридний підхід. Об'єм даних для початку навчання задається користувачем, а перед тим агент навчається за допомогою Online Learning. Після того, як потрібна кількість згадок накопичилась, агент з певною періодичністю починає навчатися за допомогою replay learning на заданій користувачем кількості даних, одночасно продовжуючи вчитися після кожного рішення.

Початкове використання online learning дозволяє моделі швидко пристосовуватися до нових даних, що є особливо корисним на ранніх стадіях навчання. Перехід до replay learning, за наявності достатньої суми даних, забезпечує глибше навчання на підставі ширшого спектру досвіду. Використання replay learning дає змогу багаторазово використовувати цінний досвід, який, забезпечуючи дієвіше використання відомих даних, може бути рідкісним або особливо актуальним для навчання. Replay learning допомагає розбавити послідовність тренувальних даних, шляхом зменшення кореляції між ними та поліпшення загальної стабільності та ефективності навчання. Цей спосіб надає можливість гнучко налаштовувати баланс між швидкістю адаптації до нових даних та глибиною навчання на основі існуючих даних,

таким чином забезпечуючи масштабованість під різноманітні умови навчання. Періодичний перехід від online до replay learning сприяє збалансуванню між потребою швидко реагувати на нові дані та необхідністю уникнути перенавчання на обмеженому або неоднорідному наборі даних.

Цей гібридний підхід використовує сильні сторони кожного з методів і забезпечує більш ефективний процес навчання. Насамперед доцільно його застосовувати в динамічних середовищах, де обставини швидко міняються, а також у проблематичних задачах, для яких важливо раціонально використовувати кожен досвід задля покращення продуктивності алгоритму.

3.14 Висновок до розділу

Отже, у розробленій DDQNwithGA системі використовується поєднання подвійної глибокої Q-мержі (DDQN) та генетичного алгоритму, де генетичний алгоритм відповідальний за налаштування гіперпараметрів та підбір навчальних даних.

Використання генетичного алгоритму для динамічного формування гіперпараметрів у подвійних глибоких Q-мережах (DDQN) має низку переваг, які підвищують загальну продуктивність і ефективність навчання. Нижче перелічені деякі ключові плюси:

- автоматизація відбору гіперпараметрів: генетичний алгоритм автоматизує процес пошуку оптимальних гіперпараметрів, при цьому зменшуючи потребу в ручному налаштуванні і тестуванні, що може бути часомістким і вимагає немалих зусиль;
- покращення продуктивності: уміння генетичного алгоритму дієво досліджувати простір гіперпараметрів може стати причиною знаходження ефективніших конфігурацій, які підвищують рівень продуктивності DDQN мережі, особливо у складних середовищах;
- адаптація до змінних умов: генетичні алгоритми можуть підлаштувати гіперпараметри відповідно до змін у даних або середовищі,

дозволяючи DDQN мережам продовжувати бути ефективними навіть при непостійних умовах;

- пошук глобального оптимуму: генетичні алгоритми здатні до глобального пошуку в просторі гіперпараметрів, уникаючи локальних мінімумів, які можуть спричинити інші методи оптимізації;

- збалансування дослідження і використання: генетичний алгоритм може допомогти визначити баланс між дослідженням нових стратегій і використанням перевірених способів, оптимізуючи політику дій DDQN задля досягнення кращих результатів;

- паралельні обчислення: генетичні алгоритми добре піддаються синхронним обчисленням, що дозволяє значно пришвидшити процес пошуку оптимальних гіперпараметрів за рахунок використання багатоядерних процесорів;

- гнучкість і універсальність: генетичний алгоритм може бути застосований до широкого діапазону задач і не залежить від специфіки проблеми, і це робить його універсальним інструментом для оптимізації гіперпараметрів у різних сценаріях використання DDQN.

Ці переваги роблять генетичний алгоритм потужним знаряддям для оптимізації гіперпараметрів у DDQN, що може значно поліпшити їх продуктивність і ефективність у різноманітних задачах. В свою чергу наслідування різних згадок(Memory) від різних батьків у контексті генетичних алгоритмів та нейронних мереж, сприяє різноманітності стратегій агентів. Це різноманіття може призвести до знаходження креативніших і оригінальніших розв'язань задач, підвищуючи шанси на виявлення оптимальних стратегій поведінки або конфігурацій, а також може допомогти моделі оминати застрягання в локальних мінімумах, збільшуючи шанси на визначення глобального максимуму або мінімуму.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Ця робота складається з таких частин: бібліотека з реалізацією моделі нейронної мережі; симуляція (всесвіт в якому розвивається агент); програма для тестування.

4.1 DDQNwithGA бібліотека

Для зручного використання моделі була розроблена бібліотека, яка містить в собі реалізацію алгоритму DDQNwithGA та допоміжні алгоритми (рис.4.1). Бібліотека базується на парадигмі ООП і не використовує сторонніх бібліотек з реалізацією методів для створення нейронних мереж.

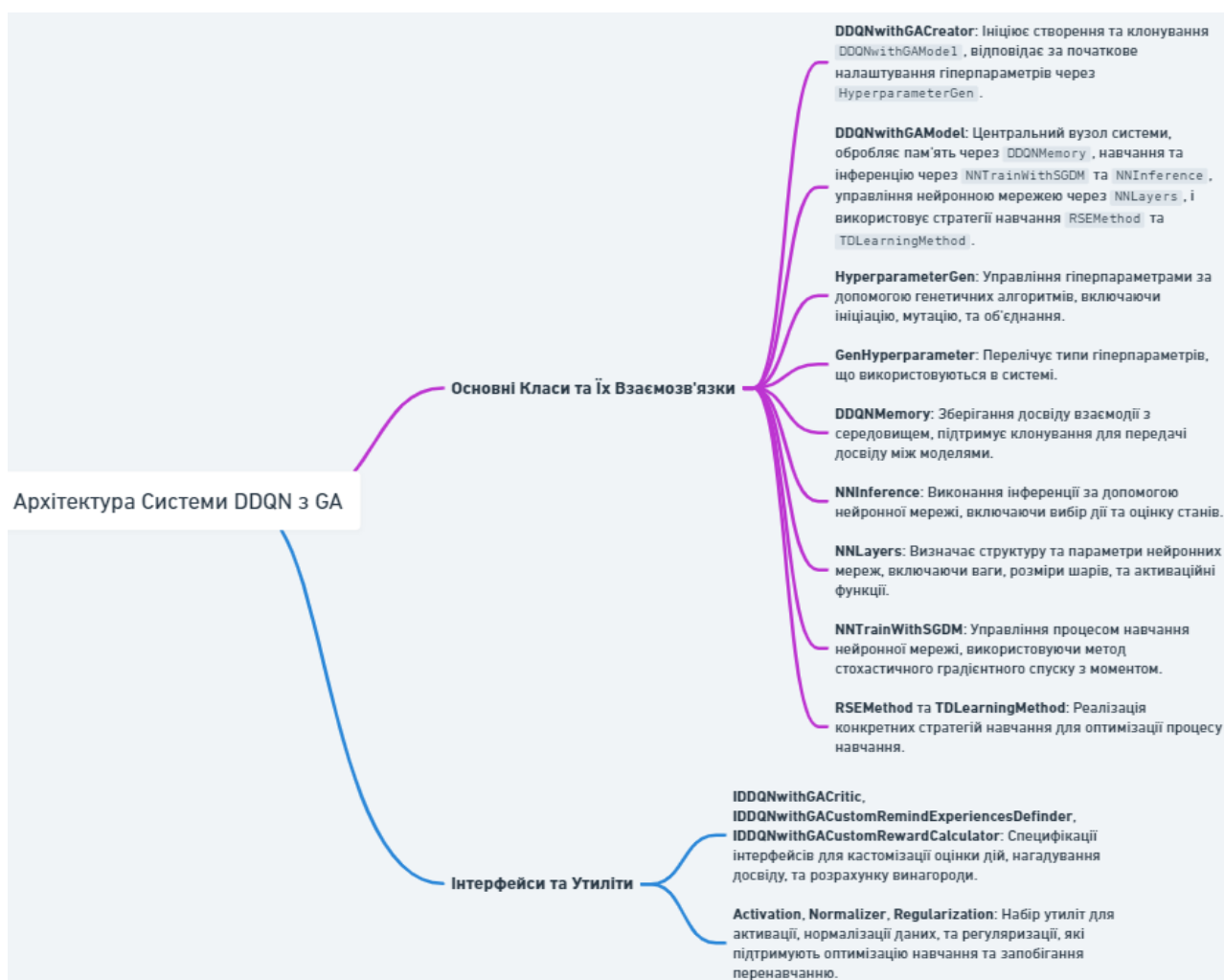


Рисунок 4.1 – Архітектура системи DDQNwithGA

4.1.1 DDQNwithGAModel (class)

Цей клас DDQNwithGAModel (рис.4.2) є програмною реалізацією алгоритму подвійної глибокої Q-навчання (Double Deep Q-Network, DDQN) з інтеграцією генетичних алгоритмів (GA) для оптимізації гіперпараметрів. Він призначений для використання в контексті навчання з підкріпленням, де агент намагається вивчити оптимальну стратегію дій, базуючись на спостереженнях про середовище і отриманих винагородах. Додаток А містить повний код класу.

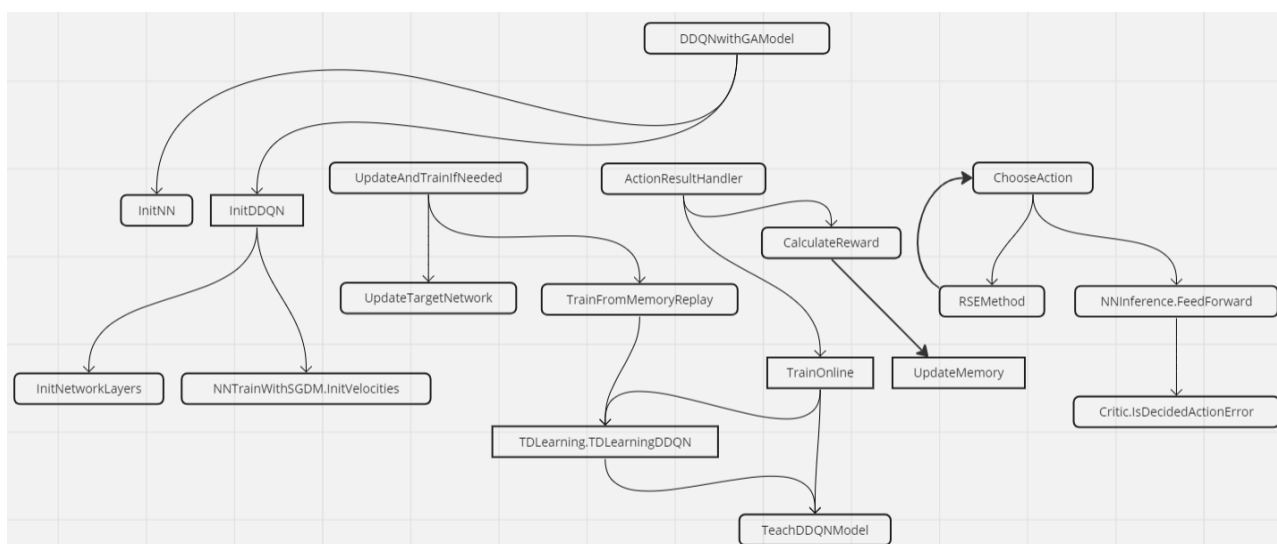


Рисунок 4.2 – Взаємозв’язок методів в класі DDQNwithGAModel

На рисунку зображено спрощений взаємозв’язок методів в класі. Ці методи містять реалізовані алгоритми і дозволяють користувачу бібліотеки з DDQNwithGA застосовувати модель в своєму коді не занурюючись в реалізацію - за допомогою декількох відкритих методів, а більш продвинутому користувачу надає можливість підлаштувати алгоритм під його задачі не занурюючись в реалізацію більш глибоких методів.

Опис кожного методу, поля та властивості та його призначення наведено нижче.

4.1.1.1 Конструктор DDQNwithGAModel

Конструктор (DDQNwithGAModel) ініціалізує основні параметри моделі, такі як розміри шарів нейронної мережі, максимальну ємність пам'яті, розмір пакету для пакетного навчання та мінімальну кількість спогадів для початку навчання з відтворенням пам'яті.

4.1.1.2 Методи DDQNwithGAModel для управління Memory

UpdateMemory (UpdateMemory) додає новий досвід до пам'яті агента і видаляє найстаріший, якщо загальна кількість перевищує максимальну ємність.

IsMemoryEnough (IsMemoryEnough) перевіряє, чи накопичено достатньо досвіду в пам'яті для початку навчання.

4.1.1.3 Методи DDQNwithGAModel для навчання моделі

UpdateAndTrainIfNeeded (UpdateAndTrainIfNeeded) оновлює цільову мережу та проводить навчання з відтворенням пам'яті, якщо накопичено достатньо досвіду.

TrainFromMemoryReplay (TrainFromMemoryReplay) проводить навчання моделі на випадково вибраних прикладах з пам'яті.

TrainOnline (TrainOnline) проводить онлайн-навчання моделі безпосередньо після кожної взаємодії з середовищем.

4.1.1.4 Методи DDQNwithGAModel для вибору дії

ChooseAction (ChooseAction) вибирає дію для виконання в середовищі, використовуючи або стратегію випадкового дослідження, або жадібний вибір на основі прогнозів нейронної мережі.

4.1.1.5 Методи DDQNwithGAModel для оновлення мережі

UpdateTargetNetwork (UpdateTargetNetwork) оновлює ваги цільової нейронної мережі, копіюючи ваги з онлайн-мережі.

4.1.1.6 Інші методи

ActionResultHandler (ActionResultHandler) обробляє результат виконання дії, оновлюючи стан середовища і розраховуючи винагороду.

CalculateReward (CalculateReward) розраховує винагороду за виконану дію на основі заданих критеріїв.

InitNN (InitNN), InitDDQN (InitDDQN), InitNetworkLayers (InitNetworkLayers) ініціалізують нейронну мережу, параметри DDQN та шари мережі відповідно.

TeachDDQNModel (TeachDDQNModel) застосовує алгоритм зворотного поширення помилки для оновлення ваг нейронної мережі.

4.1.1.7 Поля і властивості DDQNwithGAModel

У класі DDQNwithGAModel визначено низку властивостей та полів, кожне з яких відіграє ключову роль у реалізації алгоритму подвійної глибокої Q-навчання з інтеграцією генетичних алгоритмів. Давайте розглянемо кожне з них:

4.1.1.7.1 Поля та властивості для даних та стану

random (Random): Генератор випадкових чисел, що використовується для вибору дій та вибірки досвідів з пам'яті.

Memory (List<DDQNMemory>): Список, що зберігає історію досвідів агента. Кожен досвід включає стан до дії, саму дію, отриману винагороду, стан після дії та прапор завершення епізоду.

`MaxMemoryCapacity (uint)`: Максимальна кількість досвідів, яку може зберігати пам'ять.

`MinMemoryToStartMemoryReplayLearning (uint)`: Мінімальна кількість досвідів у пам'яті, необхідна для початку процесу навчання з відтворенням пам'яті.

`IsMemoryEnough (bool)`: Прапор, що вказує, чи накопичено достатньо досвідів для початку навчання.

`BatchMemorySize (uint)`: Розмір пакету для пакетного навчання з відтворенням пам'яті.

`**AfterActionState (double[])`, `BeforeActionState (double[])`: Масиви, що зберігають стан середовища до та після виконання дії.

`TargetValueBeforeAction (double)`: Цільове значення до виконання дії.

`Action (int)`: Остання вибрана дія.

`TotalReward (double)`, `TotalActionsNum (double)`: Загальна сума винагород та кількість дій за епізод для розрахунку середньої винагороди.

`IsActionError (bool)`: Прапор, що вказує на те, чи була остання дія помилковою з точки зору критика.

4.1.1.7.2 Поля для нейронної мережі

`OnlineLayers (NNLayers[])`, `TargetLayers (NNLayers[])`: Масиви шарів нейронної мережі для онлайн і цільової мережі відповідно.

`LayersSizes (int[])`: Масив, що зберігає розміри кожного шару нейронної мережі.

`Gen (HyperparameterGen)`: Інстанція для генерації та оптимізації гіперпараметрів за допомогою генетичних алгоритмів.

`NNInference`, `NNTrainWithSGDM`: Компоненти для виконання висновку та навчання нейронної мережі з використанням методу стохастичного градієнтного спуску з моментом.

4.1.1.7.3 Поля для кастомізації поведінки

Critic (IDDQNwithGACritic): Інтерфейс для критика, який може оцінювати, чи була вибрана дія помилковою.

RewardCalculator (IDDQNwithGACustomRewardCalculator): Інтерфейс для калькулятора винагород, що дозволяє кастомізувати розрахунок винагород.

RemindExperiencesDefinder (IDDQNwithGACustomRemindExperiencesDefinder): Інтерфейс для визначення логіки нагадування певних досвідів, що може бути корисним для покращення стратегії навчання.

Ці поля та властивості разом формують систему, призначену для навчання агента в середовищі навчання з підкріпленням. Вони дозволяють агенту зберігати історію свого досвіду, адаптуватися до середовища через процес навчання, оптимізувати свої стратегії дій на основі отриманих винагород та керувати своїми нейронними мережами для вдосконалення прогнозування та вибору дій.

4.1.2 DDQNwithGACreator

Клас DDQNwithGACreator використовується для створення та клонування екземплярів моделі DDQNwithGAModel, яка інтегрує подвійну глибоку Q-мережу (DDQN) з генетичними алгоритмами (GA) для оптимізації гіперпараметрів. Цей клас є фабрикою для створення, копіювання та схрещування моделей DDQN з GA, що дозволяє легко експериментувати з різними конфігураціями мереж та гіперпараметрів. Цей клас дозволяє ефективно управляти життєвим циклом моделей DDQN з GA, спрощуючи експерименти з навчанням за допомогою різноманітних архітектур нейронних мереж і налаштувань гіперпараметрів. Повний код у додатку Б.

Далі наведено детальний опис його методів.

4.1.2.1 Методи створення моделі

CreateDDQNwithGA - методи перевантажені для створення нового екземпляра DDQNwithGAModel з заданими параметрами. Перший варіант методу приймає словник початкової похибки гіперпараметрів (startHyperparameterScatterDict), а другий - одне значення похибки для всіх гіперпараметрів (startHyperparameterScatter). Це дозволяє гнучко налаштовувати ініціалізацію генетичного алгоритму.

4.1.2.2 Методи клонування моделі

CloneDDQNwithGA - методи для клонування моделей. Перший варіант створює точну копію переданої моделі, в той час як другий варіант генерує нову модель шляхом злиття двох батьківських моделей, використовуючи випадковий вибір атрибутів від кожного з батьків.

4.1.2.3 Внутрішні методи DDQNwithGACreator

ValidateLayerSizes - перевіряє, що передано достатню кількість шарів для моделі (мінімум вхідний та вихідний).

CopyFrom - створює точну копію оригінальної моделі, включаючи всі її параметри, пам'ять, швидкості оновлення та шари нейронної мережі.

MergeFrom - створює нову модель шляхом комбінування атрибутів двох батьківських моделей, вибираючи кожен атрибут випадковим чином від одного з батьків.

InheritVelocities, InheritNetworkLayers, InheritMemory - допоміжні методи, що використовуються для копіювання або злиття відповідних частин моделі, таких як ваги та зсуви нейронної мережі, її пам'ять і параметри оновлення градієнтного спуску.

4.1.3 Ресурси нейронної мережі

Ресурс нейронної мережі відповідають за навчання моделі і її використання, а також надають архітектуру для збереження даних, створення та оновлення мережі.

4.1.3.1 NNInference

Клас NNInference призначений для виконання прогнозування або інференсу в контексті нейронної мережі, що використовується в алгоритмі DDQN з генетичним алгоритмом для оптимізації гіперпараметрів. Цей клас є ключовим компонентом для роботи нейронної мережі, яка визначає, як вхідні дані перетворюються на вихідні сигнали через серію взаємопов'язаних шарів. Цей клас є важливою складовою системи навчання з підкріпленням, де ефективність інференсу нейронної мережі безпосередньо впливає на здатність агента вивчати та адаптуватися до середовища.

Ось детальний опис його методів:

4.1.3.1.1 Конструктор NNInference

NNInference - конструктор приймає два параметри: екземпляр HyperparameterGen, що містить гіперпараметри нейронної мережі, і масив layersSizes, який визначає розміри кожного шару мережі. Ці параметри використовуються для подальшого конфігурування та виконання нейронної мережі.

4.1.3.1.2 Методи для прямого розповсюдження

FeedForward - метод для прямого розповсюдження вхідних даних через нейронну мережу без введення шуму. Він послідовно обчислює вихід

кожного шару, використовуючи ваги, зсуви та активаційну функцію (Swish), параметризовану гіперпараметром `beta`.

`FeedForwardWithNoise` - альтернативний метод для прямого розповсюдження, який додатково вводить шум до результатів активаційної функції кожного нейрона. Шум генерується з урахуванням інтенсивності шуму, визначеної в гіперпараметрах. Також застосовується метод `Dropout` для кожного шару на основі заданої ймовірності випадання, що допомагає уникнути перенавчання мережі.

4.1.3.1.3 Вибір відповіді

`FindMaxIndexForFindAction` - метод визначає індекс максимального елемента в масиві вихідних значень нейронної мережі, що відповідає вибору дії з найвищим очікуваним винагородженням. У разі, коли вхідний масив є результатом методу `FeedForward` або `FeedForwardWithNoise`, цей індекс представляє дію, яку слід вибрати.

4.1.3.2 NNLayers

Структура `NNLayers` представляє собою шар нейронної мережі в контексті системи подвійного глибокого Q-навчання (DDQN) з інтеграцією генетичних алгоритмів. Ця структура використовується для зберігання та обробки інформації, пов'язаної з кожним шаром нейронної мережі, включаючи розміри шарів, нейрони, зміщення (biases) та ваги (weights). Використання цієї структури дозволяє ефективно моделювати та маніпулювати різними шарами нейронної мережі, спрощуючи процес навчання. Далі неведо детальний опис її компонентів.

4.1.3.2.1 Поля та властивості

`Size` - властивість, яка вказує кількість нейронів у поточному шарі.

`NextSize` - властивість, яка вказує кількість нейронів у наступному шарі, необхідна для визначення розмірів масиву ваг.

`Neurons` - масив, який містить значення активації нейронів поточного шару.

`Biases` - масив, який містить зсуви (біаси) для кожного нейрона в шарі.

`Weights` - двовимірний масив, який зберігає ваги між нейронами поточного та наступного шарів.

4.1.3.2.2 Конструктор

`NNLayers(int size, int nextSize)` - конструктор, який ініціалізує шар нейронної мережі з заданими розмірами. Він створює масиви для нейронів, зміщень та ваг, а також викликає метод ініціалізації ваг за допомогою методу `He`.

4.1.3.2.3 Методи

`InitializeWeightsHe()` - метод для ініціалізації ваг шару з використанням методу ініціалізації `He`, який є популярним вибором для глибоких нейронних мереж, оскільки він допомагає запобігти проблемам з градієнтами в глибоких мережах, нормалізуючи ваги за розміром шару. Ініціалізація `He` використовує нормальний розподіл зі стандартним відхиленням, обрахованим як корінь квадратний з 2, діленого на кількість нейронів у шарі, що дозволяє вагам мати оптимальні початкові значення для активації типу `ReLU` або її варіантів, таких як `Swish`.

4.1.3.3 NNTrainWithSGDM

Клас `NNTrainWithSGDM` відповідає за навчання нейронної мережі в контексті подвійного глибокого Q-навчання (`DDQN`) з використанням стохастичного градієнтного спуску з імпульсом (`Stochastic Gradient Descent`

with Momentum, SGDM). Цей метод навчання поєднує переваги стохастичного градієнтного спуску з прискоренням конвергенції через використання імпульсу, який допомагає "прискорити" градієнти в правильному напрямку. Клас є ключовим для реалізації процесу навчання нейронної мережі у системі DDQN з GA, забезпечуючи ефективне та швидке оновлення ваг з метою оптимізації політики дій агента в середовищі навчання з підкріпленням. Далі наведено детальний опис його компонентів.

4.1.3.3.1 Поля

`VelocitiesWeights` - тривимірний масив, який зберігає швидкості зміни ваг для кожного з'єднання між нейронами в мережі. Це необхідно для реалізації механізму імпульсу.

`VelocitiesBiases` - двовимірний масив, який зберігає швидкості зміни зміщень (biases) для кожного нейрона.

`gen` - екземпляр класу `HyperparameterGen`, який містить гіперпараметри, використовувані для навчання мережі, включаючи швидкість навчання, коефіцієнт імпульсу, коефіцієнт L2-регуляризації тощо.

4.1.3.3.2 Конструктор

`NNTrainWithSGDM(HyperparameterGen gen)` - конструктор класу, що ініціалізує клас з об'єктом `HyperparameterGen`, який містить необхідні гіперпараметри.

4.1.3.3.3 Методи

`InitVelocities(NNLayers[] layers)` - ініціалізує швидкість зміни ваг та зміщення для кожного шару мережі, готуючи їх до процесу навчання з використанням SGDM.

`BackPropagationSGDM(double[] predicted, double[] targets, NNLayers[] layers)` - виконує алгоритм зворотного поширення помилки з імпульсом для оновлення ваг та зміщень мережі. Метод адаптує ваги та зміщення, використовуючи розрахункові градієнти з урахуванням вихідних даних мережі та цільових значень. Використання імпульсу допомагає підвищити ефективність і швидкість навчання, а L2-регуляризація допомагає запобігти перенавчанню шляхом додавання штрафу за великі ваги.

4.1.3.4 DDQNMemory

Клас `DDQNMemory` є ключовою частиною механізму відтворення досвіду в `DDQN`, дозволяючи агенту ефективно навчатися на основі попереднього досвіду, що сприяє покращенню стратегії поведінки в середовищі. Зберігання досвіду дозволяє агенту використовувати методи пакетного навчання для оновлення політики дій на основі великої кількості досвіду, замість того, щоб навчатися від кожного досвіду окремо, що підвищує стабільність і ефективність навчання.

`DDQNMemory` використовується для зберігання інформації про окремий досвід агента в контексті подвійного глибокого Q-навчання (`DDQN`). Цей досвід включає стан середовища до виконання дії, саму вирішену дію, отриману винагороду, стан середовища після виконання дії та прапорець, який вказує, чи завершено епізод. Ось детальний опис його компонентів та методів:

4.1.3.4.1 Поля

`BeforeActionState` - масив, що представляє стан середовища до виконання дії.

`DecidedAction` - ціле число, що вказує на дію, вирішену агентом для виконання в даному стані.

Reward - число з плаваючою комою, що представляє винагороду, отриману агентом за виконання дії.

AfterActionState - масив, що представляє стан середовища після виконання дії.

Done - булеве значення, що вказує, чи завершено епізод після виконання даної дії.

4.1.3.4.2 Конструктор

DDQNMemory(double[] beforeActionState, int decidedAction, double reward, double[] afterActionState, bool done) - конструктор, який ініціалізує новий екземпляр класу з заданими параметрами досвіду.

4.1.3.4.3 Методи

Clone() - реалізація інтерфейсу ICloneable, яка дозволяє створювати глибокі копії об'єктів DDQNMemory. Це корисно для створення незалежних копій досвіду, які можуть бути збережені в пам'яті для подальшого навчання без ризику ненавмисної модифікації оригінальних даних.

4.1.4 HyperparameterGen

Клас HyperparameterGen розроблений для управління та оптимізації гіперпараметрів у контексті генетичних алгоритмів, застосованих до подвійної глибокої Q-мережі (DDQN). Цей клас дозволяє динамічно адаптувати гіперпараметри під час процесу навчання, щоб покращити загальну продуктивність агента. Клас HyperparameterGen є фундаментальним для адаптивної настройки гіперпараметрів у системах навчання з підкріпленням, де здатність до точної настройки гіперпараметрів може значно впливати на продуктивність та ефективність навчання.

Ось детальний опис його компонентів та функціональності:

4.1.4.1 Поля та константи

Клас містить набір констант, які визначають початкові значення гіперпараметрів, таких як швидкість навчання (`learningRateStart`), коефіцієнт зниження (`discountFactorStart`), ймовірність дослідження (`explorationStart`) та інші.

Поле `HyperparameterChromosome` – словник, який зберігає поточні значення гіперпараметрів.

Поле `blockedHyperparameterGens` – список заблокованих гіперпараметрів, які не будуть змінюватися під час мутацій.

Поле `random` – екземпляр класу `Random`, використовуваний для генерації випадкових чисел.

4.1.4.2 Конструктори

`HyperparameterGen()` – базовий конструктор, який ініціалізує гіперпараметри з їх початковими значеннями.

`HyperparameterGen(HyperparameterGen original)` – конструктор копіювання, який створює новий екземпляр, копіюючи гіперпараметри з іншого екземпляра та застосовуючи випадкову мутацію.

`HyperparameterGen(HyperparameterGen mother, HyperparameterGen father)` – конструктор, що створює новий екземпляр шляхом комбінування гіперпараметрів від двох батьківських екземплярів з подальшою мутацією.

4.1.4.3 Методи

`HyperparametersInit()` – ініціалізує гіперпараметри з їх початковими значеннями.

`StartHyperparameterScatter(double scatterProc)` та `StartHyperparameterScatter(Dictionary<GenHyperparameter, double>`

scatterProcDict) – методи для ініціалізації гіперпараметрів з додаванням випадкового розкиду до їх початкових значень.

BlockHyperparameterGenChanging(GenHyperparameter hyperparameter) і UnblockHyperparameterGenChanging(GenHyperparameter hyperparameter) – блокують або розблоковують зміни конкретного гіперпараметра під час мутації.

SetHyperparameter(GenHyperparameter hyperparameter, double positiveOrZeroValue) – встановлює конкретне значення для обраного гіперпараметра.

RandomMutation() – застосовує випадкову мутацію до гіперпараметрів, які не заблоковані.

4.1.4.4 Enum GenHyperparameter

Цей enum визначає ідентифікатори для всіх гіперпараметрів, які можуть бути оптимізовані в процесі навчання. Це дозволяє легко звертатися до конкретних гіперпараметрів при їх читанні та оновленні.

У переліку GenHyperparameter визначено ключі для доступу до різних гіперпараметрів, які використовуються в генетичних алгоритмах для налаштування подвійної глибокої Q-мережі (DDQN). Кожен гіперпараметр має унікальне значення, яке впливає на різні аспекти процесу навчання або поведінки алгоритму.

hyperparameterChromosomeMutationProbability - ймовірність мутації для хромосом гіперпараметрів. Визначає, наскільки часто будуть відбуватися випадкові зміни в гіперпараметрах під час оптимізації.

errorFine - штраф за помилку. Використовується для покарання моделі за неправильні дії, допомагає уникати небажаних дій.

correctBonus - бонус за правильну дію. Мотивує модель вибирати дії, які максимізують отриману винагороду.

genDoneBonusA і genDoneBonusB - коефіцієнти для обчислення бонусу при завершенні задачі або епізоду.

`genHyperparameterPercentageChange` - відсоток зміни гіперпараметрів. Вказує на максимальний діапазон зміни гіперпараметрів під час мутації.

`learningRate` - швидкість навчання. Контролює, наскільки швидко модель адаптується до нових даних. Занадто велике значення може призвести до нестабільності, тоді як занадто мале - до повільного навчання.

`noiseIntensity` - інтенсивність шуму. Визначає рівень шуму, який додається до вихідних даних мережі для збільшення її робастності або для експлорації.

`discountFactor` - фактор дисконтування. Використовується в обчисленні майбутньої винагороди, допомагає визначити важливість майбутніх винагород.

`exploration` - рівень експлорації. Регулює баланс між дослідженням нових дій та використанням вже відомих стратегій.

`momentumCoefficient` - коефіцієнт імпульсу. Використовується для підвищення швидкості навчання і зменшення осциляцій.

`lambdaL2` - коефіцієнт L2-регуляризації. Додає штраф за великі ваги, сприяючи більш гладким і загальним моделям.

`beta` - параметр для активаційної функції Swish, що впливає на її форму та поведінку.

`dropoutRate` - рівень відмови нейронів (dropout). Вказує на частку нейронів, які випадково ігноруються під час тренування, що допомагає запобігти перенаванчання.

`percentageOfSimilarExperiences` - відсоток схожих досвідів. Використовується для вибору частки схожих досвідів під час тренування для підвищення ефективності навчання.

`remindProbability` - ймовірність нагадування. Контролює, наскільки часто модель повторює попередні досвіди для кращого запам'ятовування.

4.1.5 Інтерфейси

Інтерфейси у реалізованій мною бібліотеці дозволяють користувачу

бібліотеки гнучко підлаштовувати модель під свої потреби і оптимізувати навчання під свої задачі.

4.1.5.1 IDDQNwithGACritic

Інтерфейс `IDDQNwithGACritic` визначає контракт для компонентів-критиків у системі подвійного глибокого Q-навчання (DDQN) з інтеграцією генетичних алгоритмів. Цей інтерфейс дозволяє реалізувати механізм оцінки вибраних дій агента, допомагаючи визначити, чи є вибрана дія помилковою з урахуванням поточного стану середовища.

Метод `IsDecidedActionError(int decidedAction, double[] LastState)`: метод приймає вирішену дію (`decidedAction`) та останній стан середовища (`LastState`), повертаючи булеве значення, яке вказує, чи є ця дія помилковою в контексті даного стану. Це дозволяє системі навчання з підкріпленням адаптувати стратегії на основі оцінок правильності дій, сприяючи покращенню вибору дій агента.

4.1.5.2 IDDQNwithGACustomRemindExperiencesDefinder

Інтерфейс `IDDQNwithGACustomRemindExperiencesDefinder` визначає контракт для реалізації механізму визначення та нагадування досвідів у системі подвійного глибокого Q-навчання (DDQN) з інтеграцією генетичних алгоритмів. Цей інтерфейс дозволяє створити кастомізовані методи для вибору певних досвідів з пам'яті, які мають бути пізніше використані під час навчання, на основі поточного стану середовища, а також для оцінки схожості між станами. Далі наведено детальний опис його методів.

Метод `CustomRemindExperiencesDefinder(List<DDQNMemory> memory, double[] currentState)`: метод приймає список досвідів (`List<DDQNMemory>`) та поточний стан середовища (`double[] currentState`). Він повинен повертати список досвідів, вибраних на основі поточного стану для використання в процесі навчання. Це дозволяє агенту зосередитися на

найбільш релевантних або схожих досвідах, покращуючи ефективність навчання.

Метод `CustomCalculateSimilarityState(double[] currentState, double[] previousState)`: метод приймає поточний стан (`double[] currentState`) та попередній стан (`double[] previousState`), обчислюючи рівень їх схожості. Вихідне значення (`double`) використовується для визначення, наскільки схожі стани один на одного. Цей метод може бути застосований для оцінки, які досвіди є найбільш релевантними для поточної ситуації.

4.1.5.3 IDDQNwithGACustomRewardCalculator

Інтерфейс `IDDQNwithGACustomRewardCalculator` визначає контракт для кастомізованих калькуляторів винагороди в системі подвійного глибокого Q-навчання (DDQN) з інтеграцією генетичних алгоритмів. Цей інтерфейс дозволяє реалізувати власні методи обчислення винагороди для агента, базуючись на результаті дії агента в середовищі та інших параметрах. Далі наведено детальний опис його методу.

Метод `CalculateReward(bool done, double episodeSuccessValue, double targetValue, double bonus)`: метод для обчислення винагороди агента. Він приймає наступні параметри:

- `done`: булеве значення, що вказує, чи завершився поточний епізод (наприклад, чи досяг агент цілі або чи сталася термінальна подія);
- `episodeSuccessValue`: число, що представляє ступінь успіху або досягнення цілі в поточному епізоді;
- `targetValue`: цільове значення, яке може використовуватися для додаткової оцінки дій агента (наприклад, різниця між поточним станом і цільовим станом);
- `bonus`: додатковий бонус, який може бути доданий до винагороди за специфічні умови або досягнення.

Метод повинен повертати числове значення винагороди, яке відображає ефективність дії агента в контексті заданих параметрів.

4.1.6 Алгоритми для оптимізації навчання

Оптимізатори навчання є критично важливим елементом у процесі проектування та тренування нейронних мереж, забезпечуючи збалансований та ефективний підхід до мінімізації функції втрат і досягнення високої продуктивності моделей.

4.1.6.1 RSEMethod

Клас RSEMethod в системі подвійного глибокого Q-навчання (DDQN) з генетичними алгоритмами реалізує метод для відбору та навчання на основі найбільш схожих досвідів. Цей підхід може допомогти покращити швидкість та ефективність навчання агента, зосереджуючись на найбільш релевантних даних. Використання такого підходу дозволяє агенту зосередитися на ключових досвідах, мінімізуючи "шум" від менш підходящих даних та сприяючи формуванню точніших і ефективніших стратегій поведінки.

Далі наведено детальний опис його компонентів та методів.

Поле `mostSimilarExperiences`: список для зберігання найбільш схожих досвідів, обраних для навчання.

Метод `CalculateEuclideanDistance(double[] currentState, double[] previousState)`: обчислює Евклідову відстань між поточним та попереднім станами, що використовується для визначення схожості досвідів.

Метод `DefineMostSimilarExperiences(double[] currentState, double percentageOfSimilarExperiences, List<DDQNMemory> memory, IDDQNwithGACustomRemindExperiencesDefinder remindExperiencesDefinder = null)`: визначає найбільш схожі досвіди на основі поточного стану. Може використовувати кастомізоване визначення схожих досвідів або стандартний підхід з Евклідовою відстанню.

Метод `RemindSimilarExperiencesDDQN(NNLayers[] onlineLayers, NNLayers[] targetLayers, NNInference inference, Action<double[], double[]> teachModel, TDLearningMethod TDLearning = null, double discountFactor = 0.95)`: використовує обрані схожі досвіди для навчання моделі. Якщо задано метод `TDLearning`, використовує його для обчислення цільових Q-значень перед навчанням моделі.

4.1.6.2 TDLearningMethod

Клас `TDLearningMethod` реалізує метод навчання (Temporal Difference) спеціально для подвійної глибокої Q-мережі (Double Deep Q-Network, DDQN) в контексті системи з генетичними алгоритмами. TD навчання є одним з ключових методів в навчанні з підкріпленням, який дозволяє агенту вчитися безпосередньо від отриманого досвіду, використовуючи різницю між прогнозованими та фактичними винагородами для оновлення своїх прогнозів. Далі наведено детальний опис методу `TDLearningDDQN`.

Цей метод приймає наступні параметри:

- `discountFactor`: коефіцієнт дисконтування, що використовується для обчислення майбутньої винагороди;
- `state`: масив, що представляє поточний стан середовища;
- `done`: булеве значення, що вказує, чи завершився поточний епізод;
- `reward`: винагорода, отримана за виконання дії;
- `decidedAction`: індекс вибраної дії;
- `nextState`: масив, що представляє стан середовища після виконання дії;
- `onlineLayers` та `targetLayers`: масиви шарів нейронної мережі для онлайн і цільової Q-мереж;
- `inference`: екземпляр класу `NNInference` для виконання прямого розповсюдження в мережі.

Далі перераховано алгоритми цього класу.

Якщо епізод не завершено, використовує онлайн-мережу для отримання Q-значень наступного стану і вибору оптимальної дії. Потім отримує Q-значення для цієї дії від цільової мережі і обчислює цільове значення TD, використовуючи отриману винагороду і дисконтовану майбутню винагороду.

Якщо епізод завершено, цільове значення TD дорівнює отриманій винагороді.

Виконує пряме розповсюдження для поточного стану за допомогою онлайн-мережі, використовуючи метод `FeedForwardWithNoise`, щоб додати елемент стохастичності.

Копіює отримані Q-значення в новий масив, який буде слугувати масивом цільових Q-значень для навчання, і оновлює Q-значення для вибраної дії з розрахунковим цільовим значенням TD.

Використання онлайн і цільової мереж дозволяє уникнути проблеми переоцінки винагород, яка є типовою для традиційних DQN.

4.1.6.3 Activation

Статичний клас `Activation` визначає реалізацію активаційних функцій `Swish` та її похідної, які можуть бути використані в нейронних мережах. `Swish` є популярною активаційною функцією, запропонованою Google Brain, і часто використовується у глибокому навчанні через її здатність до допомоги у навчанні глибоких мереж з кращою конвергенцією порівняно з `ReLU`. Використання `Swish` замість традиційних активаційних функцій, може допомогти уникнути проблем з "мертвими нейронами", коли нейрони стають неактивними та не вносять вклад в адаптацію мережі через негативні вхідні значення. Далі представлено детальний опис методів.

Метод `SwishActivation(double x, double beta)`: Реалізує активаційну функцію `Swish`, яка визначається як $x * \text{sigmoid}(\beta * x)$, де $\text{sigmoid}(\beta * x)$ є сигмоїдною функцією, а β - параметр, що регулює крутизну сигмоїди.

Функція Swish вважається нелінійною та неперервною, здатною до самономалізації в глибоких мережах.

Метод `DSwishActivation(double x, double beta)`: Реалізує похідну функції Swish за x , яка необхідна для процедур зворотного розповсюдження помилки в навчанні нейронних мереж. Похідна виражається як $\text{sigmoid}(\beta * x) + \beta * x * \text{sigmoid}(\beta * x) * (1 - \text{sigmoid}(\beta * x))$, що дозволяє обчислити градієнт функції втрат відносно ваг нейронної мережі.

4.1.6.4 Normalizer

Статичний клас `Normalizer` містить методи для нормалізації та денормалізації даних, що є важливими операціями в обробці даних, особливо у роботі з нейронними мережами. Нормалізація допомагає покращити швидкість навчання та стабільність моделей шляхом приведення вхідних даних до певного стандартного діапазону.

Підготовка даних: нормалізація даних перед тренуванням моделі допомагає уникнути проблем зі збіжністю та дозволяє моделі швидше адаптуватися.

Обробка вхідних даних: при передачі даних в нейронну мережу нормалізація забезпечує, що вхідні значення мають однаковий масштаб, що полегшує оптимізацію ваг.

Візуалізація даних: нормалізовані дані можуть бути корисними для візуалізації, дозволяючи легше порівнювати різні набори даних або оцінювати зміну певних параметрів.

Ці методи є універсальними та можуть бути використані в широкому спектрі задач обробки даних, від простих програм до складних систем машинного навчання.

Далі представлено детальний опис методів, які пропонує цей клас.

Метод `MinMaxNormalize(double value, double minValue, double maxValue)`: використовує метод нормалізації мінімум-максимум для того, щоб привести значення `value` до діапазону від 0 до 1 на основі вказаних

мінімального (`minValue`) і максимального (`maxValue`) значень. Якщо `value` більше або дорівнює `maxValue`, повертається 1; якщо `value` менше або дорівнює `minValue`, повертається 0; в інших випадках виконується пропорційне масштабування.

Метод `MinMaxDenormalize(double normalizedValue, double minValue, double maxValue)`: перетворює нормалізоване значення `normalizedValue` назад у його оригінальний діапазон, використовуючи ті ж `minValue` і `maxValue`.

Метод `TanhNormalize(double value)`: використовує гіперболічний тангенс для нормалізації `value`, що автоматично приводить його до діапазону між -1 і 1. Ця функція є корисною, коли потрібно обробляти негативні значення або коли бажано мати симетричний діапазон нормалізації.

4.1.6.5 Regularization

Статичний клас `Regularization` надає набір методів для застосування регуляризації та інших технік, які спрямовані на покращення загальної продуктивності та уникнення перенавчання в нейронних мережах. Ці методи регуляризації є критично важливими для побудови ефективних нейронних мереж, та в задачах з великими даними, де ризик перенавчання високий. L2 регуляризація сприяє обмеженню розміру ваг, Dropout дозволяє мережі краще узагальнювати знання, а введення шуму може допомогти мережі адаптуватися до невеликих даних. Детальний опис його методів наводиться нижче.

Метод `ApplyL2Regularization(double pureGradient, double lambdaL2, double weight)`: застосовує L2 регуляризацію до градієнта, що допомагає уникнути перенавчання шляхом додавання штрафу за великі ваги. `pureGradient` - це градієнт без регуляризації, `lambdaL2` - коефіцієнт L2 регуляризації, а `weight` - вага, до якої застосовується регуляризація. Результатом є модифікований градієнт.

Метод `ApplyDropout(double[] activations, int layerIndex, int[] layersSizes, double dropoutRate)`: реалізує метод Dropout, який випадковим

чином "вимикає" деякі нейрони під час тренування, щоб зменшити залежність між нейронами та уникнути перенавчання. `activations` - масив активацій нейронів поточного шару, `layerIndex` - індекс поточного шару, `layersSizes` - масив розмірів шарів мережі, `dropoutRate` - ймовірність відключення кожного нейрона.

Метод `GenerateRandomNoise()`: генерує випадковий шум, використовуючи нормальний розподіл (Гауссівський розподіл).

4.2 Симуляція

Симуляція еволюції за допомогою моделі DDQNwithGA створена для тестування можливостей розробленого алгоритму (рис.4.3). Програма симулює складне та динамічне середовище з візуалізацією в консолі.

Для прискорення розвитку використовується паралелізм, кожний агент використовує свій власний потік для прийняття рішення, навчання та зміни середовища.

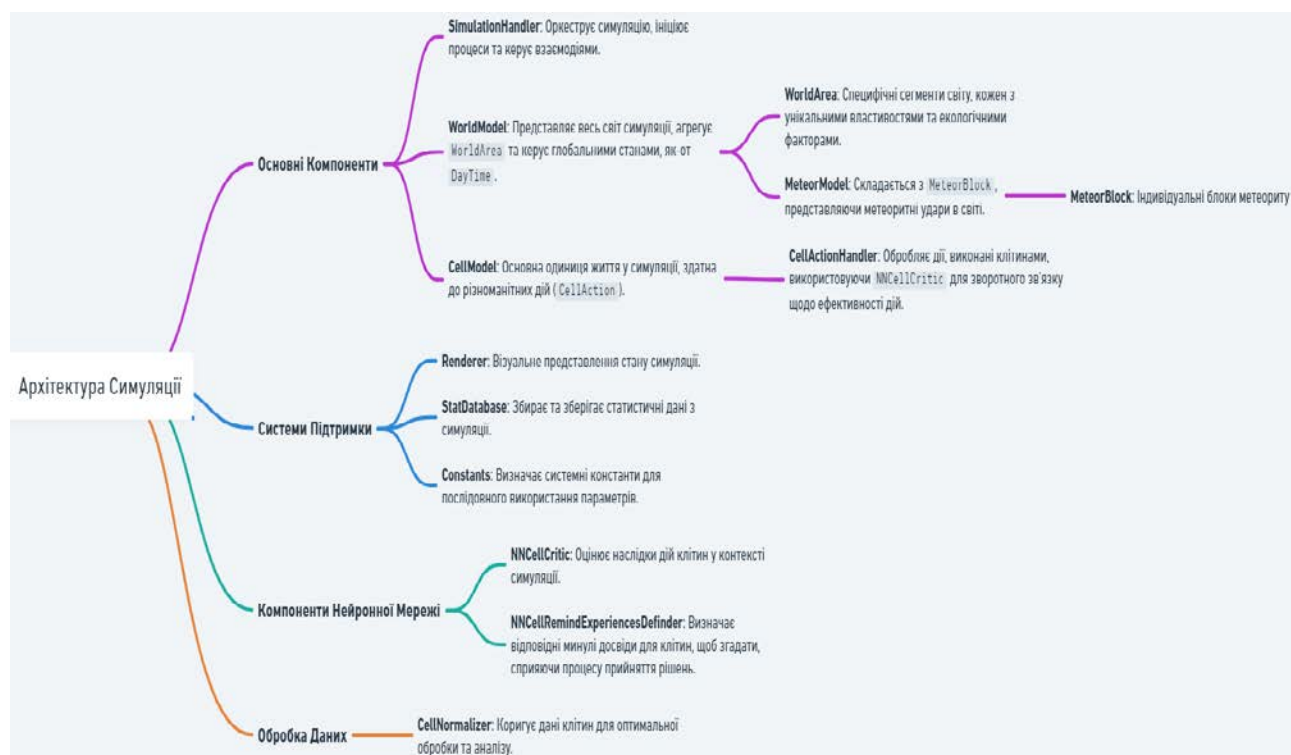


Рисунок 4.3 – Архітектура симуляції

4.2.1 SimulationHandler

Клас `SimulationHandler` відіграє ключову роль у керуванні процесом симуляції.

Його методи та поля разом формують основу для детального вивчення та аналізу процесів еволюції клітин у контрольованому середовищі. Вони забезпечують інструменти для моніторингу динаміки популяції, взаємодії клітин з середовищем, а також для майбутньої оцінки ефективності.

Розглянемо більш детально поля та методи, що визначають логіку та структуру цього класу.

4.2.1.1 Поля

`WorldModel world`: інстанція класу `WorldModel`, що містить модель світу, де відбувається симуляція.

`StatDatabase statDatabase`: інстанція класу `StatDatabase`, призначена для зберігання та аналізу статистичних даних симуляції.

`int PhotoCells, BiteCells, AbsorbCells, MineCells, ErrorCells, SlipCells`: цілочисельні змінні для підрахунку кількості клітин за типами.

`double DayErrorValue, CurrentErrorProc`: змінні для зберігання значень помилок за день та поточного відсотка помилок.

4.2.1.2 Методи

`void StartSimulation()`: головний метод для запуску симуляції. Він ініціює цикл симуляції, що продовжується до того моменту, поки в моделі світу залишається хоча б одна клітина.

`void ShowWorldInfo()`: виводить інформацію про світ, включаючи поточний хід, рік, час доби, енергію фотосинтезу тощо.

`void ShowCellsNumInfo()`: виводить загальну кількість клітин та середню енергію.

`void ShowCellsTypeInfo()`: виводить інформацію за типами клітин, включаючи відсоток помилок.

`void ShowTimeInfo(Watch stopwatchCells)`: виводить інформацію про час виконання ходу.

`void UpdateCellsTypeInfo()`: оновлює інформацію про кількість клітин за типами на основі їхнього кольору та виявлених помилок у русі.

`void UpdateDataBaseDayErrorValue()`: оновлює базу даних статистики, додаючи значення помилок за день.

4.2.2 WorldResources

Об'єкти світу формують основусвіту симуляції, кожен з них відіграє важливу роль у моделюванні динамічного віртуального світу, де агенти взаємодіють з середовищем та один з одним (рис.4.4), а також адаптуються та розвиваються відповідно до зовнішніх умов і власного досвіду.

`WorldModel`, `WorldArea`, та `MeteorModel` спільно створюють і управляють середовищем симуляції. Вони відповідають за моделювання часу доби, погодних умов, метеоритних подій та загальним станом світу, в якому існують агенти.

`CellModel` представляє індивідуальні клітини, які є агентами в цьому світі, включаючи їхні характеристики та поведінку, а `CellActionHandler` керує діями клітин, такими як рух, споживання ресурсів та взаємодія з іншими клітинами або елементами середовища.

Разом ці об'єкти створюють багатопланову симуляцію, де агенти не просто реагують на зміни в середовищі, а й активно намагаються пристосуватися та розвиватися через складні механізми навчання та взаємодії. Це дозволяє моделювати більш реалістичну екосистему, в якій можуть відбуватися складні процеси адаптації та еволюції.



Рисунок 4.4 – Взаємодія об’єктів світу

4.2.2.1 Cell

Cell(клітина) – це агент, який вчиться пристосовуватись до навколишнього середовища. В нього є 30 можливих дій і йому треба навчитись використовувати якнайкраще, щоб максимізувати нагороду, тобто отриману енергію. За зв'язок з навколишнім середовищем, нейронною мережею та обробку дій клітини відповідають наступні класи:

4.2.2.1.1 CellModel

Клас CellModel є головним компонентом у симуляційній системі, яка моделює поведінку та взаємодії агентів у віртуальному середовищі. Він інкапсулює дані та логіку, необхідні для імітації життєвого циклу клітин, включаючи їхні рухи, взаємодію з іншими клітинами та середовищем, харчування, розмноження та адаптацію до змінних умов.

4.2.2.1.1.1 Поля

PositionX, PositionY: координати клітини на ігровому полі.

Energy: поточний рівень енергії клітини.

CurrentAge: поточний вік клітини.

CellColor: колір клітини для візуалізації.

IsDead, IsSlip, IsHide та інші логічні поля, що відображають стан клітини.

brain: модель мозку клітини, яка використовує глибоке посилене навчання для оптимізації поведінки.

world: посилання на модель світу, в якому існує клітина.

4.2.2.1.1.2 Методи

MakeAction(): вирішує, яку дію виконати на основі поточного стану та навколишнього середовища.

MoveUp(), MoveDown(), MoveLeft(), MoveRight() та інші методи руху: переміщують клітину в різних напрямках.

JumpUp(), JumpRight(), JumpDown(), JumpLeft(): виконують "стрибки" клітини на більшу відстань у вказаному напрямку.

BiteLeftUp(), BiteUp() та інші методи "кусання": взаємодіють з іншими клітинами, "куштуючи" їх для отримання енергії.

Photosynthesis(): дозволяє клітині отримувати енергію через фотосинтез.

Absorption(): абсорбція ресурсів з навколишнього середовища.

Clone(), Reproduction(): методи, що відповідають за розмноження клітини.

Slip(), Hide(): дозволяють клітині "слизати" або ховатися для захисту.

MineTop(), MineRightSide(), MineBottom(), MineLeftSide(): методи для "видобутку" ресурсів.

4.2.2.1.3 Взаємодія з нейронною мережею

Клітина використовує нейронну мережу (DDQNwithGA) для визначення оптимальних дій на основі свого стану та стану навколишнього середовища. Мережа аналізує поточну ситуацію та вибирає найкращу дію, яку потім виконує клітина.

Цей клас є важливим елементом симуляції, оскільки він детально моделює поведінку та життєвий цикл клітин, надаючи основу для вивчення складних взаємодій та еволюційних процесів у віртуальному світі.

4.2.2.1.2 CellActionHandler

Клас CellActionHandler відповідає за управління діями клітин у симуляційному світі. Це включає рух клітин, їхнє харчування, розмноження, а також взаємодію з іншими об'єктами та середовищем, такими як отруйні зони або метеоритні блоки. Основні аспекти та функціонал класу включають:

4.2.2.1.2.1 Поля

World: посилання на екземпляр WorldModel, який представляє собою поточний стан світу симуляції.

LockObject: об'єкт для синхронізації доступу до спільних ресурсів в багатопоточному середовищі. Використання lockObject забезпечує безпечну багатопоточну роботу, дозволяючи синхронно оновлювати стан світу та уникати конфліктів при одночасному доступі до спільних ресурсів.

4.2.2.1.2.2 Методи

CellMove(int i): управління переміщенням клітини на основі її індексу в масиві клітин. Цей метод враховує стан клітини (жива/мертва) та змінює її позицію, якщо вона виконала рух.

`GetInfoFromAreaToCellBrainInput(int positionX, int positionY)`: збирає інформацію з навколишнього середовища клітини для подальшої передачі до її "мозку" (нейронної мережі). Це включає дані про енергію в сусідніх клітинах, типи об'єктів у навколишньому середовищі та інше.

`CellChangePos(CellModel cell, int lastX, int lastY)`: оновлює позицію клітини у світі після її переміщення, зберігаючи зміни у відповідних структурах даних.

`CellHunt(CellModel hunter, CellModel victim)`: управління процесом полювання клітини на іншу клітину, а також логіку передачі енергії від жертви до мисливця.

`CellAbsorb(CellModel absorber)`: виконує абсорбцію енергії з навколишнього середовища клітиною.

`CellMineTop(CellModel miner)`, `CellMineRightSide`, `CellMineBottom`, `CellMineLeftSide`: методи для "видобутку" ресурсів клітиною з метеоритних блоків у різних напрямках.

`CellStartCreatingClones()` та `CellStartReproduction()`: управління процесом розмноження клітин, включаючи клонування та створення потомства за участю двох клітин.

4.2.2.1.3 Класи для налаштування моделі DDQNwithGA

Класи `NNCellCritic` і `NNCellRemindExperiencesDefinder` відіграють ключову роль в процесі навчання та прийняття рішень клітин у системі, яка використовує глибоке посилене навчання з генетичним алгоритмом (DDQNwithGA). Вони дозволяють клітинам аналізувати свої дії відповідно поставленої задачі.

Клас `NNCellCritic` використовується для оцінки, чи є обрана дія помилковою на основі поточного стану вхідних даних клітини. Цей клас реалізує інтерфейс `IDDDQNwithGACritic`, який вимагає метод `IsDecidedActionError`, що повертає булеве значення, вказуючи на помилковість дії.

Клас `NNCellRemindExperiencesFinder` відповідає за вибір найбільш релевантних досвідів із пам'яті для підтримки процесу навчання клітини. Цей клас реалізує інтерфейс `IDDQNwithGACustomRemindExperiencesFinder`, що дозволяє кастомізувати процес вибору досвіду з пам'яті на основі поточного стану середовища. Класи `MeteorBlock` та `MeteorModel` вносять в систему додатковий рівень динаміки та викликів, надаючи клітинам можливість адаптуватися до раптових змін середовища, видобувати ресурси з метеоритних блоків та виживати в умовах "метеоритної ночі".

4.2.2.2 Meteor

Класи `MeteorBlock` та `MeteorModel` є частиною симуляційної системи, що моделює вплив метеоритів на віртуальне середовище, в якому існують клітини. Вони дозволяють імітувати події, пов'язані з падінням метеоритів, та їхній вплив на клітини та інші елементи світу.

4.2.2.2.1 MeteorBlock

Цей клас представляє окремий блок метеорита, який має визначене розташування та кількість руди (`OrbNum`), що може бути видобута клітинами. Кількість руди генерується випадково у момент створення метеоритного блоку.

4.2.2.2.2 MeteorModel

Клас `MeteorModel` управляє генерацією метеоритних подій у симуляції, включаючи розміщення метеоритних блоків на карті та визначення тривалості "ночі", спричиненої метеоритним дощем.

Нижче перераховані поля і методи цього класу.

`MeteorBlockDist`: дистанція, на якій розміщуються метеоритні блоки навколо центральної точки падіння.

PoisonAddDist: відстань, на якій може поширюватися отрута від метеоритів (не використовується безпосередньо в наведеному коді).

NightTurns: кількість ходів "ночі", що настає після падіння метеорита.

IsCreateMeteorBlocks: чи потрібно створювати метеоритні блоки.

CenterPositionX, CenterPositionY: координати центру події падіння метеорита.

MeteorBlocks: список метеоритних блоків, що були сгенеровані під час події.

MeteorModel(): конструктор, який ініціалізує модель, вибираючи тип події (великий, середній, малий метеоритний дощ) та генеруючи метеоритні блоки.

CreateMeteorBlocks(): створює метеоритні блоки навколо центральної точки падіння на заданій дистанції.

4.2.2.3 WorldModel

Клас WorldModel є ключовим для координації взаємодії між усіма елементами симуляції, управління часовими циклами та подіями у світі, а також забезпеченням механізму для їх візуалізації. Він втілює основну логіку симуляції, забезпечуючи динамічне та змінне середовище для еволюції та взаємодії клітин.

4.2.2.3.1 Основні компоненти та функціонал

WorldArea: визначає ігрове поле, де відбувається вся діяльність клітин та інших об'єктів.

cellActionHandler: об'єкт, що керує діями клітин, такими як їхній рух, харчування, репродукція.

Cells: список всіх клітин у світі, кожна з яких представлена екземпляром класу CellModel.

`worldRenderer`: об'єкт для візуалізації стану світу, який може бути включений або виключений в залежності від налаштувань симуляції.

4.2.2.3.2 Управління часом та подіями

`MakeTurn()`: основний метод, що викликається для кожного "ходу" у грі, виконуючи логіку переміщення клітин, їхнього розмноження, вплив метеоритних подій тощо.

`MeteorFalling()`: імітує падіння метеоритів, викликаючи створення метеоритних блоків на карті та можливе знищення клітин у зоні удару.

`UpdateTimeAndSeason()`: оновлює час у світі, переключаючи між днем та ніччю, а також відстежує зміну днів і років.

4.2.2.3.3 Взаємодія з клітинами

`GetCell(int x, int y)`: повертає клітину, розташовану на заданих координатах, якщо вона існує.

`PerformCellLogicParallel()`: виконує логіку дій клітин паралельно, використовуючи багатопоточність для ефективнішого оновлення стану світу.

4.2.2.3.4 Візуалізація

`InitWorldRenderer(Renderer worldRenderer)`: ініціалізує рендерер для візуалізації стану світу.

`CreateVisualIfRendererExists()` та `StartRenderIfRendererExists()`: методи для управління процесом візуалізації, залежно від наявності рендерера.

4.2.2.4 WorldArea

Клас `WorldArea` забезпечує високий рівень абстракції для управління просторовими аспектами симуляції, дозволяючи легко маніпулювати

елементами на ігровому полі та адаптувати світ до різних подій, таких як падіння метеоритів чи зміни в енергетиці середовища. Це ключовий компонент для створення динамічного та змінного світу, в якому клітини мають адаптуватися та еволюціонувати для виживання.

4.2.2.4.1 Основні компоненти та функціонал

AreaChar, AreaColor, AreaEnergy: Масиви, які зберігають символи, кольори та енергетичні значення для кожної точки на ігровому полі відповідно.

MeteorBlocks: список метеоритних блоків, що присутні на ігровому полі.

4.2.2.4.2 Ініціалізація та налаштування

WorldArea(WorldModel world): конструктор класу, який ініціалізує ігрове поле, задаючи початкові значення та створюючи необхідні структури даних.

CreateAreasParallel(): метод для паралельної ініціалізації ігрового поля, включаючи встановлення кордонів, початкового розміщення клітин та заповнення пустого простору.

4.2.2.4.3 Управління елементами на ігровому полі

FillAreaParallel(): заповнює ігрове поле енергією та визначає отруйні зони на основі рівня енергії.

CreatePoisonArea(int x, int y): створює отруйну зону на заданих координатах.

ClearAreaFromPoison(int x, int y): видаляє отруйну зону, встановлюючи звичайний стан для вказаної точки.

`CreateMeteorBlock(MeteorBlock meteorBlock)`: додає метеоритний блок до списку та оновлює ігрове поле відповідно.

`ClearMeteorBlock(MeteorBlock meteorBlock)`: видаляє метеоритний блок із списку та ігрового поля.

`GetMeteorBlock(int x, int y)`: повертає метеоритний блок за заданими координатами.

4.2.2.4.4 Управління клітинами

`ClearDeadCells()`: видаляє мертві клітини з ігрового поля, перетворюючи їх на енергію.

`ClearAreaFromDeadCell(int positionX, int positionY)`: очищує ігрове поле від мертвих клітин.

`DeadCellToAreaEnergy(int i)` та `DeadCellToAreaEnergy(CellModel cell)`: перетворює енергію мертвих клітин у вільну енергію на ігровому полі.

4.2.2.4.5 Допоміжні функції

`IsAreaPoisoned(int x, int y)`: перевіряє, чи є задана точка на ігровому полі отруйною.

`FindAllEmptyCharNearCellCoord`, `FindAllCellCharNearCellCoord`: методи для пошуку порожніх точок або точок з клітинами навколо заданої координати.

4.2.3 Constants

Клас `Constants` визначає набір констант, які використовуються у програмі для управління ключовими параметрами симуляції. Це включає значення, які впливають на поведінку клітин, умови середовища, механіки гри, та інші аспекти, що дозволяють налаштовувати та керувати симуляцією. Ці константи роблять симуляцію більш гнучкою та налаштовуваною під різні

сценарії. Константи впливають на стратегії виживання клітин, їх адаптацію до змін у середовищі та загальну динаміку еволюційного процесу в симуляційному світі.

4.2.4 StatDatabase

Клас StatDatabase в проєкті "Еволюція клітин" призначений для роботи з базою даних, дозволяючи вести статистику по помилковим діям клітин протягом симуляції. Цей клас використовує ADO.NET для взаємодії з базою даних Microsoft SQL Server, зберігаючи інформацію про кожен "день" симуляції та відповідне значення помилки.

Цей клас слугує для збору та аналізу статистики по симуляції, дозволяючи детально відстежувати помилкові дії клітин та оцінювати ефективність їхньої поведінки у різних умовах середовища. Використання бази даних для зберігання такої інформації допомагає у забезпеченні постійного доступу до історичних даних та їх аналізі.

4.2.4.1 Поля

connectionString: рядок підключення до бази даних, що вказує на джерело даних, назву каталогу, параметри шифрування та інтегровану безпеку.

tableErrorMoves: назва таблиці у базі даних, де зберігаються дані про помилкові дії.

4.2.4.2 Методи

StatDatabase(): конструктор класу, який викликає метод ініціалізації бази даних InitializeDatabase(), створюючи нову таблицю для зберігання даних або очищаючи існуючу.

`UpdateStat(int Day, double DayErrorValue)`: оновлює статистику в базі даних, вставляючи нові значення дня та відповідної помилки.

`InsertDataError(int lineNumber, double value)`: виконує вставку даних про помилку у вказаний день симуляції. Повертає `true`, якщо операція пройшла успішно, та `false` у випадку помилки, логуючи деталі помилки.

`InitializeDatabase()`: ініціалізує базу даних, спочатку видаляючи існуючу таблицю з даними про помилки, якщо така існує, а потім створюючи нову таблицю для зберігання цих даних.

4.2.5 Renderer

Клас `Renderer` є важливою частиною системи для забезпечення інтерактивності та візуальної привабливості симуляції. Він допомагає краще зрозуміти динаміку процесів, що відбуваються в симульованому світі, та надає засоби для спостереження за еволюцією та адаптацією клітин у реальному часі. Цей клас забезпечує інтерактивне візуальне представлення динаміки симуляції, роблячи процес спостереження за еволюцією клітин більш наочним та зрозумілим.

4.2.5.1 Основні компоненти та методи

`World`: посилання на екземпляр `WorldModel`, який містить поточний стан симуляції.

`CreateWorldVisual()`: метод для створення початкового візуального представлення світу. Він проходиться по всім елементам ігрового поля, відображаючи їх у консолі згідно з їх типом та станом.

`VisualChange(int x, int y, char ch, ConsoleColor fore)`: метод для оновлення візуального представлення окремих елементів на ігровому полі. Він змінює зображення елемента за заданими координатами, символом та кольором.

4.2.5.2 Синхронізація та візуалізація

Використання об'єкта `lockObject` забезпечує синхронізацію доступу до ресурсів при багатопоточній візуалізації, запобігаючи конфліктам та помилкам при оновленні стану в різних частинах ігрового поля.

Перед відображенням символів на екрані виконується перевірка `worldCreated`, щоб уникнути спроби візуалізації до завершення ініціалізації світу.

4.3 Програма для тестування

Програма, складається з двох класів `StatModelError` та `ErrorStat`, призначена для збору та візуалізації статистичних даних про помилки в процесі симуляції. Це дозволяє аналізувати ефективність розвитку клітин у віртуальному середовищі, виявляти тенденції та потенційні проблеми в алгоритмах їхньої взаємодії та адаптації.

4.3.1 ErrorStat Form

Це форма `Windows Forms`, яка використовує бібліотеку `LiveCharts` для створення інтерактивних діаграм. Клас `ErrorStat` є корисним інструментом для моніторингу ефективності та точності симуляції "Еволюції клітин", надаючи візуальне представлення динаміки зміни помилок у процесі еволюції.

`LiveCharts`, допомагає динамічно відображати зміни в даних про помилки, що дозволяє аналізувати тенденції та шаблони у поведінці симуляції. Дані представлені у вигляді лінійного графіка, де по осі *X* дні симуляції, а по осі *Y* — значення помилок

Завдяки використанню таймера дані на графіку оновлюються автоматично через заданий інтервал часу, що забезпечує актуальність

відображеної інформації без необхідності вручну перезавантажувати або оновлювати форму.

Нижче наведені поля та методи класу.

ErrorNoGenSwish: список об'єктів `StatModelError`, що зберігає дані про помилки, завантажені з бази даних.

timer: таймер для періодичного оновлення даних на графіку.

InitChart(): метод, який ініціалізує графік, встановлюючи назви осей та розташування легенди.

LoadData(): завантажує дані з бази даних та оновлює графік.

LoadStatsFromDatabase(string tableName): викликає статичний метод `LoadDataFromDatabase` класу `StatModelError` для завантаження даних з бази даних.

InitTimer(): ініціалізує таймер та встановлює його інтервал.

Timer_Tick(object sender, EventArgs e): метод, який виконується при кожному тикі таймера, оновлюючи дані на графіку.

4.3.2 ErrorStatModel

Клас `StatModelError` призначений для зберігання та завантаження статистичних даних про помилки, які відбуваються протягом симуляції "Еволюції клітин", з бази даних. Ці дані дозволяють аналізувати ефективність та адаптацію клітин протягом часу, для ідентифікації тенденцій, проблемних моментів у поведінці клітин або для оцінки ефективності різних стратегій адаптації. Через збереження даних у базі даних забезпечується легкий доступ до історичних даних, що дозволяє проводити глибокий аналіз ефективності моделі еволюції клітин.

Нижче наведені поля та методи цього класу.

Day: день симуляції, який може використовуватися як індекс для відстеження хронології подій.

ErrorPoint: величина помилки в заданий день, що може слугувати метрикою для оцінки точності або ефективності поведінки клітин.

`LoadDataFromDatabase(string tableName)`: статичний метод, який завантажує дані про помилки з вказаної таблиці бази даних. Метод використовує ADO.NET для з'єднання з базою даних SQL Server, виконує SQL-запит для вибірки даних та повертає список об'єктів `StatModelError`, заповнених даними з бази.

Процес завантаження даних:

- створення з'єднання з базою даних за допомогою рядка підключення;
- відкриття з'єднання та виконання SQL-запиту для вибірки даних з таблиці;
- читання результатів запиту через `SqlDataReader`;
- створення екземплярів `StatModelError` для кожного рядка у результаті та заповнення їх даними;
- додавання екземплярів до списку, який потім повертається методом.

5 ВСЕСВІТ ТА ТЕСТУВАННЯ

5.1 Всесвіт

Екосистема симуляції є складним і взаємопов'язаним середовищем, де клітини взаємодіють одна з одною та з різними елементами навколишнього середовища, зокрема отруєними областями та метеоритами. Кожен компонент екосистеми, від найдрібніших деталей ландшафту до глобальних циклів зміни часу доби, відіграє роль у формуванні умов життя та еволюції клітин. У процесі симуляції клітини постійно стикаються з викликами: від пошуку та використання ресурсів до уникнення хижаків та адаптації до змін у навколишньому середовищі.

Основа світу симуляції - це двовимірний масив, де кожен елемент може грати роль певної частини навколишнього середовища: бути вільною зоною, отруєною зоною або місцем падіння метеориту. Також елементи можуть бути зайняті клітинами - агентами симуляції, які взаємодіють між собою та з навколишнім середовищем.

5.1.1 Легенда візуалізації

Нова клітини - `ConsoleColor.Yellow`. Клітина хижак - `ConsoleColor.DarkRed`. Клітина, яка фотосинтезує - `ConsoleColor.Green`. Клітина поглинач - `ConsoleColor.Blue`. Клітина, яка спить - `ConsoleColor.DarkMagenta`. Клітина, яка ховається - `ConsoleColor.Magenta`. Клітина, яка копає руду - `ConsoleColor.DarkYellow`. Мертва клітина - `ConsoleColor.Gray`. Клітина, яка помилилась - `ConsoleColor.Red`.

@ - клітина.

- стіна.

! - отруєна територія.

\$ - блок метеориту.

На рисунку 5.1 колір клітини залежить від останньої дії. В момент збереження в симуляції пройшло тільки 319 діб (початок навчання).

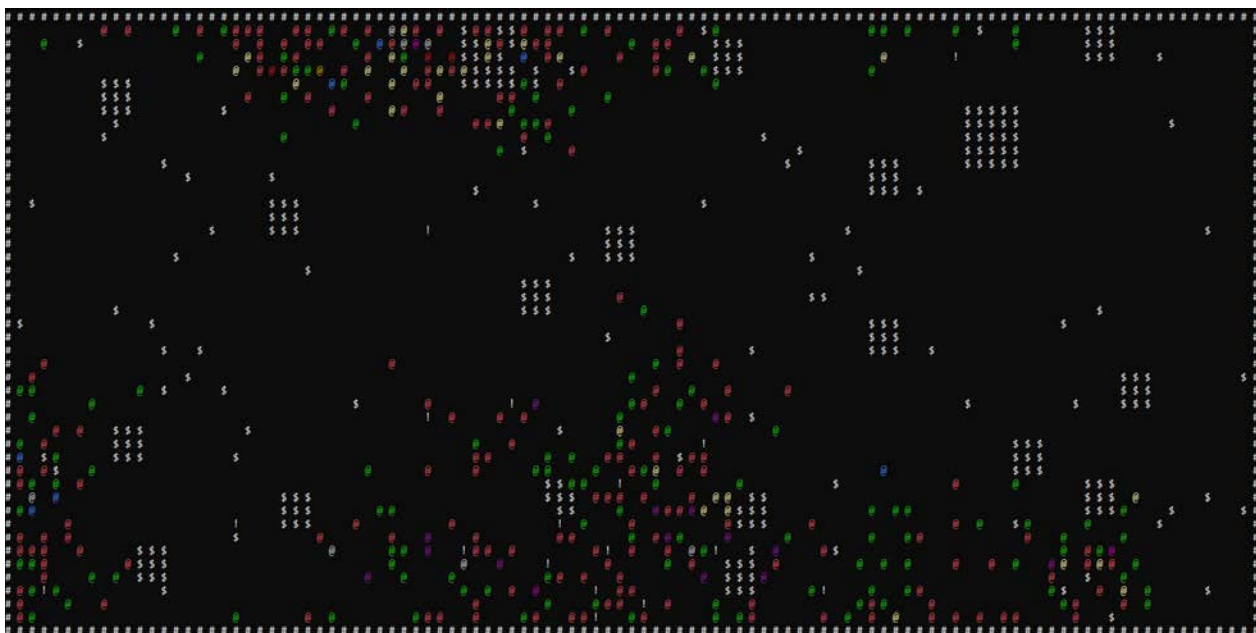


Рисунок 5.1 – Зразок візуалізації симуляції

На зображенні можна побачити приблизно 5 колоній, які створились на базі малої кількості клітин з більше ніж 1000, в яких випав найуспішніший геном і вони клонувались або схрещувались.

5.1.2 Клітини

Клітини у світі симуляції є складними агентами, які взаємодіють з навколишнім середовищем та іншими клітинами. Кожна клітина має 30 можливих дій, та певну кількість енергії, а в перспективі й унікальними стратегіями виживання, які адаптовані до умов середовища, що динамічно змінюється. Нижче наведено огляд основних дій доступних клітинам, та їх впливу на екосистему симуляції.

Переміщення. Клітини можуть рухатись в різних напрямках: вгору, вниз, ліворуч, праворуч, а також по діагоналі. Це дозволяє їм досліджувати навколишнє середовище, знаходити їжу, уникати небезпек і конкурентів.

Стрибки. Стрибки дозволяють агенту перескакувати через одну або кілька клітин, що може бути корисним для швидкого переміщення або уникнення перешкод і хижаків, або навпаки полювання.

Фотосинтез. Фотосинтез є критично важливим для клітин. Вдень вони можуть використовувати сонячне світло для виробництва енергії, що є основним джерелом для багатьох клітин. У фотосинтеза є певні правила:

Чим більше клітин у світі – тим менше максимально можливий фотосинтез і тим більше клітини втрачають енергії за спробу фотосинтезувати вночі. Агенти навчаються і адаптуються до середовища і їх стає дуже багато, тому для підсилення природного відбору зроблені умови, в яких чим агентів більше тим складніше їм виживати.

Другим правилом я надихнувся з відомого клітинного автомату «Гра в життя», який був вигаданий математиком Джоном Конвеєм у 1970 році. Це правило говорить, що якщо біля клітини 2-3 інші живі клітини то вона може отримати максимальну кількість енергії з фотосинтезу, а чим більша різниця від цих чисел – тим відповідно менше енергії вона може згенерувати.

Поглинання. Ця дія дозволяє клітинам поглинати залишки інших клітин для отримання енергії. Це важливий механізм виживання, особливо в умовах гострої конкуренції та обмежених ресурсів, крім того якщо залишків буде забагато зона стане отруєною, а за допомогою поглинання можна її «вилікувати».

Відтворення. Клітини можуть розмножуватись двома способами: через клонування себе або через розмноження з іншими клітинами. Це забезпечує генетичну різноманітність і адаптацію до умов навколишнього середовища.

Видобуток. Клітини здатні видобувати ресурси з метеоритних блоків, що надає їм додаткову енергію для виживання та розвитку.

Полювання. Клітини можуть атакувати та поглинати інші клітини для отримання їх ресурсів, що додає елемент конкуренції та хижацтва в екосистему.

Захист. Клітини можуть ховатись, якщо клітина зхована то полювання на неї неможливо.

Сон. Клітина використовує енергію для кожної дії, якщо клітина спить, то вона витрачає менше енергії за це, чим якщо б вона обрала будь яку іншу дію.

Забезпечення передачі свого генетичного матеріалу наступному поколінню є ключовою метою клітин. Це може включати як клонування, так і схрещення. Ця мета є головною, за рахунок того що ці дії закінчують епізод і клітина при навчанні отримує нагороду за дії в ньому і за рахунок генетичного алгоритму. В моєму коді реалізован природний відбір, тому клітини, які не обирали репродукцію з часом вимерли і не змогли передати накопичені знання та гени. У довгостроковій перспективі клітини прагнутимуть еволюційного розвитку, де природний відбір заохочуватиме найбільш адаптовані та успішні стратегії виживання та відтворення, тим самим формуючи майбутній напрямок еволюції клітин у симуляції.

Енергія є також важливим аспектом у житті клітини, що визначає її здатність до виконання дій та виживання в динамічному середовищі, а також оцінює успішність рішень та епох. Клітини повинні балансувати між витратами та отриманням енергії, щоб підтримувати свою життєдіяльність. Споживання енергії за кожну дію та стратегії управління енергетичними ресурсами є критично важливими для адаптації до змінних умов та для ефективної взаємодії з іншими клітинами та елементами середовища.

Також клітини будуть прагнути до розробки та реалізації стратегій, спрямованих на мінімізацію ризиків та загроз для свого існування. Це може включати уникнення хижаків, адаптацію до токсичних або отруєних областей та розвиток механізмів самозахисту.

У деяких сценаріях клітини можуть розвивати здатність до соціальної взаємодії та кооперації з іншими клітинами для досягнення спільних цілей, таких як захист від хижаків, ефективніше використання фотосинтезу, захист свого виду від інших або відтворення.

Кожна група дій має свій колір, який прописан в константах, колір клітини формується, як середній колір серед дій. Таким чином клітини бачать хижаків, нові клітини, клітини які копають руду і т.д. Колір не

обмежує їх дії якоюсь групою, а лише показує найчастіший вибір, тому навіть, наприклад, зелена клітина, яка зазвичай використовує фотосинтез може полювати або поглинати енергію померлих клітин.

5.1.3 Час доби

Зміна часу доби у світі симуляції значно впливає на поведінку і стратегії виживання клітин, особливо на процес фотосинтезу.

5.1.4 Метеорити

Метеорити у світі симуляції є унікальним елементом навколишнього середовища, який вносить різноманітність в екосистему і надає клітинам додаткові ресурси для виживання. Вони не тільки змінюють глобальний стан всесвіту, а також надають ресурси для клітин, і можуть змінювати ландшафт світу симуляції, створюючи нові перешкоди, а деякі метеорити навіть викликають ніч на декілька ходів.

Генерація метеоритів відбувається випадковим чином. Параметри створення, включаючи розмір та розподіл метеоритних блоків, вираховуються на основі визначених ймовірностей та констант розміру. Ці параметри впливають на кількість метеоритних блоків, що утворюються та кількість руди в них.

Клітини можуть видобувати ресурси з метеоритних блоків, що є важливою частиною їхньої стратегії виживання. Ресурси, одержані з метеоритів, можуть бути використані клітинами для збільшення їх енергетичних запасів. Видобуток ресурсів з метеоритних блоків вимагає від клітин виконання спеціальних дій, таких як "Майнінг" (видобуток), які надають значну кількість енергії при успішному видобутку.

Метеорити, таким чином, є ключовим елементом світу симуляції, додаючи шар складності та надаючи клітинам можливості для розвитку та адаптації. Вони стимулюють різноманітність стратегій виживання серед

клітин і роблять важливий внесок у динамічність та розвиток екосистеми світу симуляції. Головною метою інтеграції їх в симуляцію було підвищення динамічності та випадковості середовища.

5.1.5 Отруєні області

Отруєні області є зонами високого ризику для клітин, оскільки перебування у таких областях може призвести до втрати енергії і навіть смерті. Клітини повинні уникати або обережно переміщатися через такі області, що вимагає від них розвитку стратегій виявлення та уникнення небезпек.

Такі зони формуються коли клітина помирає, але ще має енергію. Ця енергія потрапляє до «землі» і якщо її накопичується досить багато то ця область стає отруєною. Отруєну область можна «вилікувати» якщо клітина обере поглинання енергії, в такому випадку вона забрає цю енергію собі.

5.1.6 Приклад симуляції

Короткий приклад роботи симуляції з навчанням за моделлю DDQNwithGA. На початку ми створимо 1196 агентів з трохи різними генами та ініціалізованими вагами за допомогою вбудованого в модель ініціалізатора. Розмір всесвіту - 47 блоків за Y та 105 блоків за X.

На рисунку 5.2 можна побачити початок роботи симуляції. Кожен агент робить дію в окремому потоці, тобто симуляція працює асинхронно. На першому зображенні видно, що декілька агентів вже встигли зробити дію, але більшість ще знаходиться на стартовій позиції. Початковий колір агентів світло-жовтий, цей колір завжди означає, що ця клітина молода.

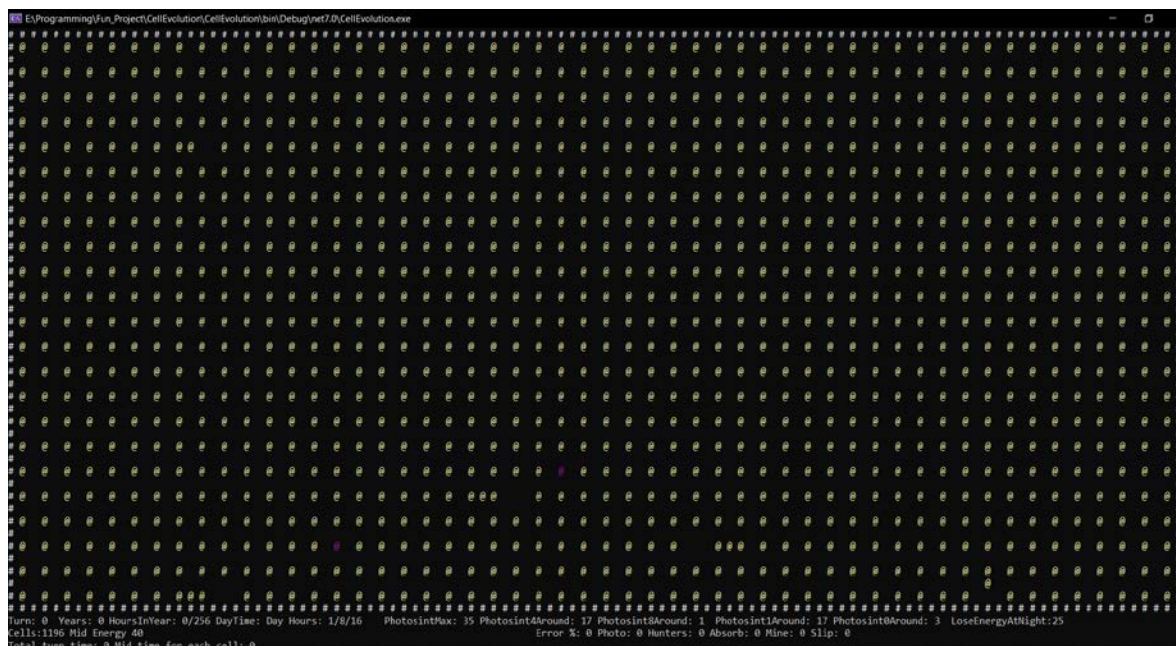


Рисунок 5.2 – Візуалізація старту симуляції

На наступному рисунку 5.3 вже пройшло 8 ходів і багато клітин, що робили дії, на які витрачали багато енергії, але не здобували її помирають. Білі клітини – мертві. Перед початком наступного ходу симуляція їх прибере з екрану.

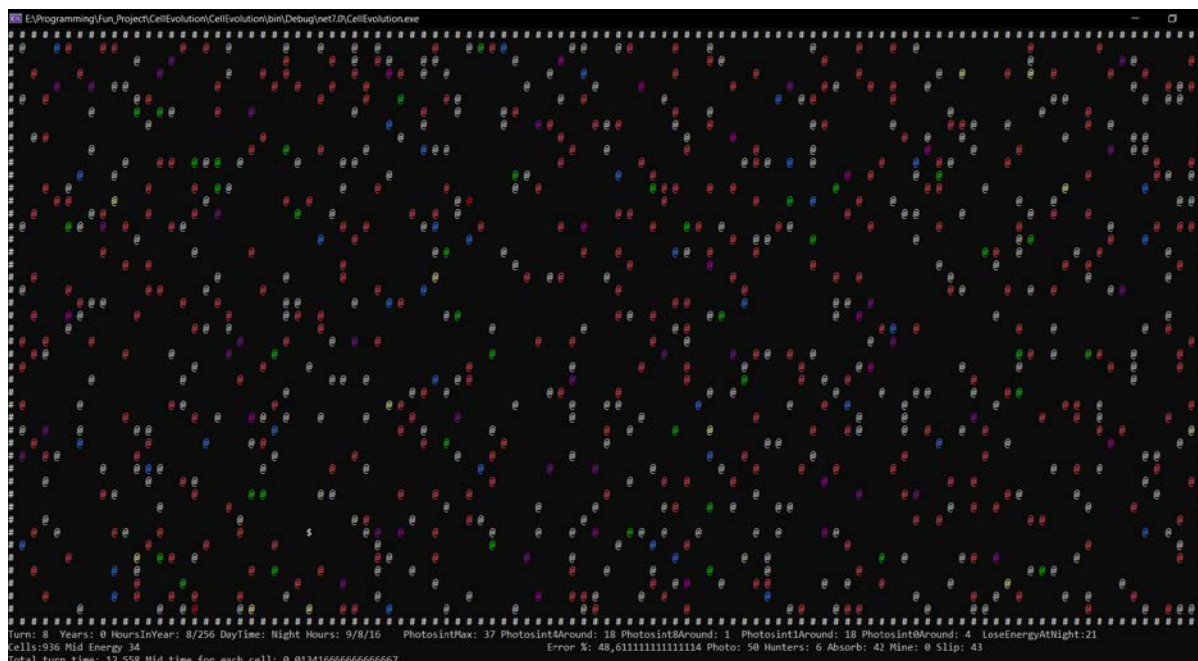


Рисунок 5.3 – Візуалізація перших етапів відбору

На рисунку 5.4 пройшло вже 33 ходи, або 2 доби. У більшості агентів помилкових дій більше ніж інших, але залишились тільки ті клітини, які вміють підлаштуватись під час доби і пережити ніч.



Рисунок 5.4 – Візуалізіція найуспішніше створених клітин

На рисунку 5.5 можна спостерігати формування колонії найуспішніших клітин, які пристосувались до екологічних умов симуляції.



Рисунок 5.5 – Візуалізація початку формування колоній

Сформовані колонії, агенти навчилися виживати в середовищі і тепер намагаються максимізувати нагороди (рис. 5.6).

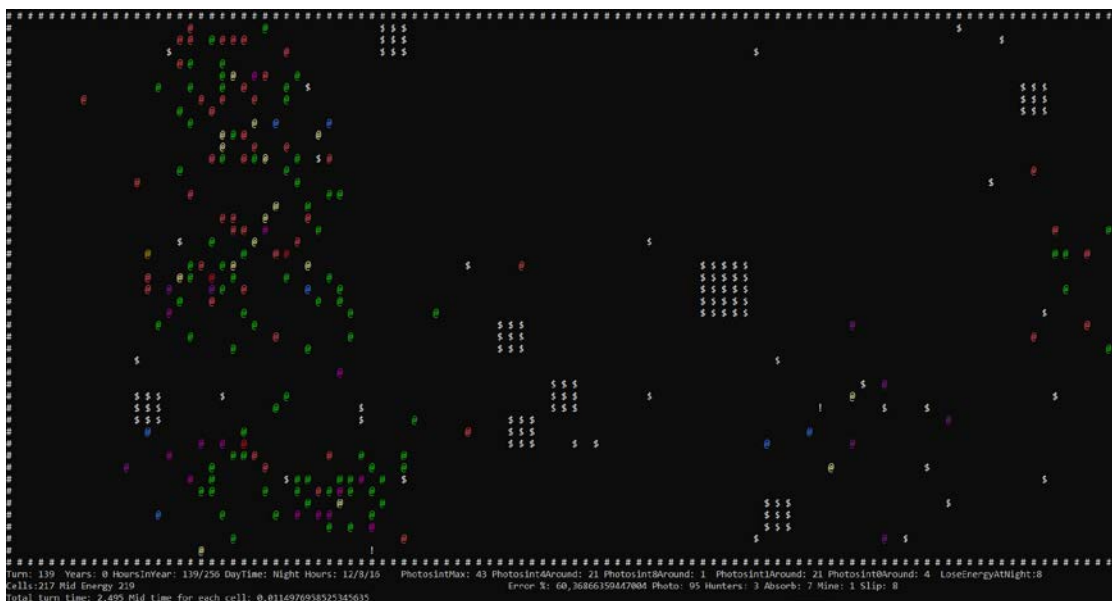


Рисунок 5.6. – Візуалізація колоній

Агенти починають розробляти певні стратегії для максимізації нагороди. На рисунку 5.7 можна побачити, як агенти вигадали стратегію формування певних форм в яких більшість агентів отримує максимально можливу кількість енергії з фотосинтезу.

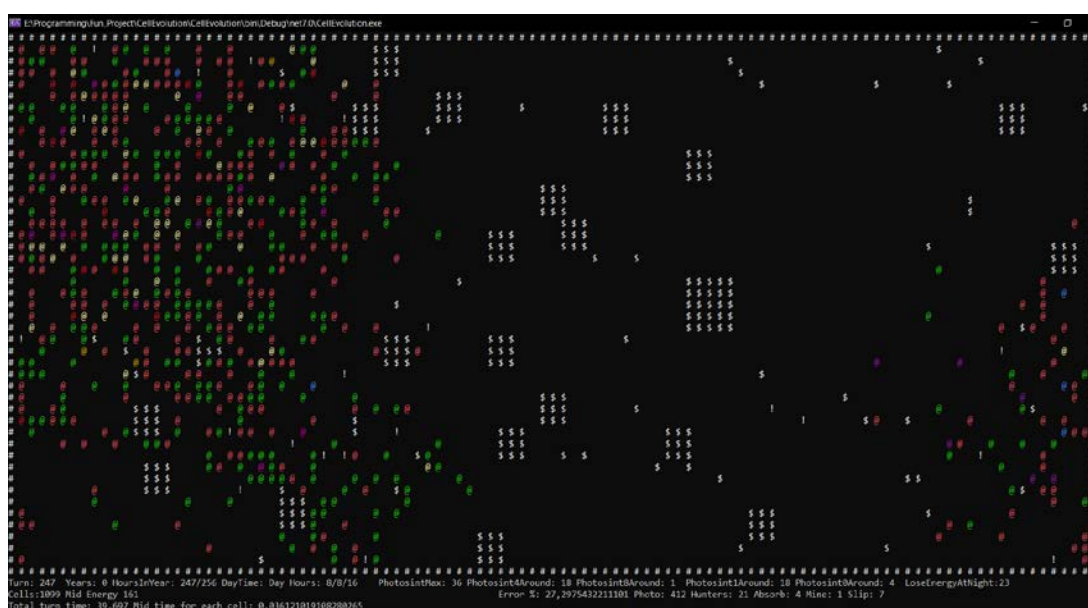


Рисунок 5.7 – Візуалізація формування агентами структур для максимізації фотосинтезу

5.2 Тестування

Тестування проходило за допомогою бази MS SQL та створенною мною програмою для створення і візуалізації графіків.

Value(Y) – відповідає за оцінку помилок. Оскільки я не знаю, яка стратегія буде вигідна через динамічність світу, я визначив якісь дії помилковими, такі як спроба походити на зайняту клітину, спроба фотосинтезувати в ночі, намагання клонуватись без достатньої кількості енергії, промазати при полюванні або видобуванні, поглинання енергії, коли її нема і т.д.

Доба(X) – це 16 ходів, де 8 з них це ніч та 8 день. За хід я отримував відсоток помилкових дій від 0 до 100 і потім всіх їх складав, тобто діапазон помилок знаходиться в межах від 0 до 1600.

При тестуванні зазвичай брався діапазон навчання з 2900 до 3400 ходів. Результати кожної доби вносились в базу і потім створювався графік з процесом та результатами відносно визначених мною помилковими діями. Проте ці оцінки не показують 100% пристосованості агентів, через дуже велику різноманітність можливих стратегій. Кожна клітина є індивідом, який формує середовище для інших клітин, тому успішність дій і стратегій залежить від стратегій і дій інших агентів.

5.2.1 Порівняння алгоритмів

Приклад генетичного алгоритму зображений на рисунку 5.8.

На представленому графіку (рис.5.8) відображено процес навчання з використанням генетичного алгоритму, де вісь Y показує "Value", показує кількість помилок, а вісь X відображає прогрес у часі за добами. Загальна тенденція демонструє, напочатку кількість помилок значно коливається, а потім спостерігається тенденція до зниження, але далі йде період стагнації та незначного зростання.

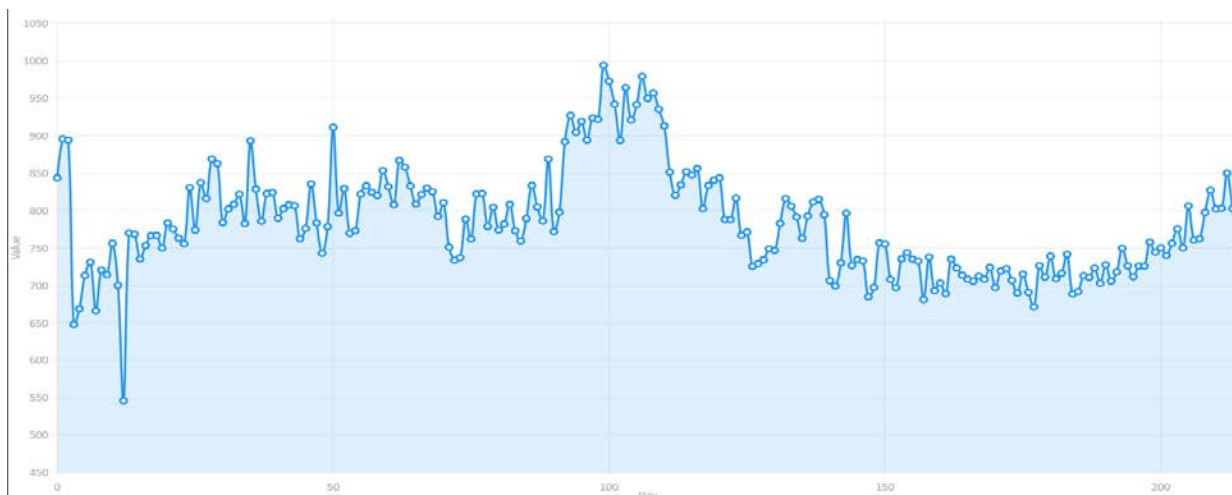


Рисунок 5.8 – Графік процесу начання за допомогою генетичного алгоритму

Це відбувається через зовелику динаміку і випадковість світу симуляції. Генетичний алгоритм знаходить оптимальне, а не найкраще рішення, тому рівень помилок залишається досить високий.

Приклад нейроеволюції зображений на рисунку 5.9. На цьому графіку ми бачимо процес навчання за допомогою нейроеволюції, де вісь Y показує кількість помилок ("Value"), а вісь X відображає прогрес у часі, виміряний днями. Подібно до генетичних алгоритмів, нейроеволюція є методом оптимізації, який використовує принципи еволюції, такі як мутації, схрещування та селекцію для підбору найкращих нейронних мереж.

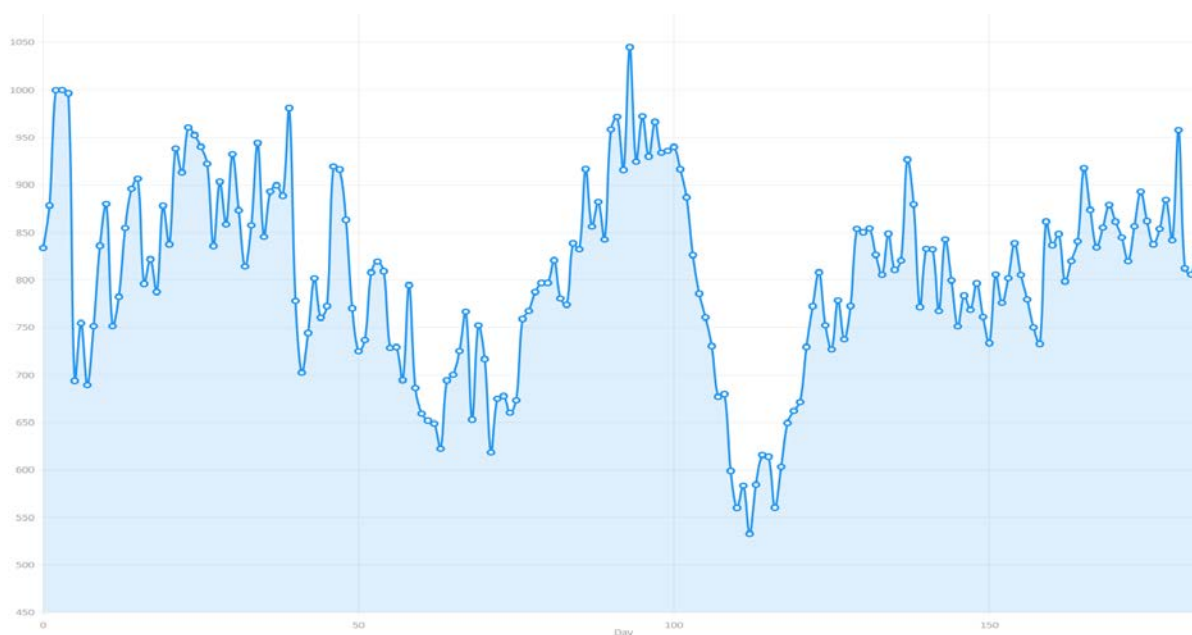


Рисунок 5.9 – Графік процесу начання за допомогою нейроеволюції

Судячи з графіку, можна помітити великі коливання, що може бути ознакою високого ступеня експлорації. Глибокий спад, який ми бачимо близько 100 доби, може вказувати на те, що було знайдено значно більш ефективну конфігурацію нейронної мережі, яка знизила кількість помилок.

Однак після цього спаду спостерігається період стагнації з незначним зростанням. Це може вказувати на насичення: алгоритм досягнув певного рівня оптимізації, і додаткові ітерації не приносять значного поліпшення. Можливо, нейронна мережа досягла локального мінімуму.

Еволюційні алгоритми можуть легко застрягти в локальних мінімумах або на плато, де дрібні зміни в архітектурі чи вагах мережі не призводять до значного покращення продуктивності, а також в багатьох випадках, використання градієнтної інформації може бути набагато ефективнішим для навчання нейронних мереж. Нейроеволюційні методи не використовують градієнтну інформацію, що може робити їх менш потужними для певних типів задач.

Приклад навчання кожного агента з використанням нейронної мережі з підкріпленням наведено на рисунку 5.10.



Рисунок 5.10 – Графік процесу навчання за допомогою навчання кожного агента з використанням нейронної мережі з підкріпленням

З графіка видно, що є великі коливання в кількості помилок на ранніх етапах навчання, що вказує на значну експлорацію стратегій або дій в алгоритмі підкріплення. Такі коливання можуть виникати через спроби мережі адаптуватися до складного або динамічного середовища.

Поступове зменшення коливань та зниження середнього рівня помилок свідчить про те, що нейронна мережа починає краще розуміти середовище і виявляти ефективніші стратегії виконання завдань. Проте, як видно з графіка, після приблизно після 100 днів, показник помилок не зменшується і навіть показує тенденцію до зростання.

Статичні гіперпараметри можуть не дозволяти адекватною мірою експлорувати середовище, особливо якщо воно змінюється чи має велику кількість станів. Мережа може перенавчитися на певних вибірках або сценаріях, втрачаючи свою здатність узагальнення нових даних. Якщо гіперпараметри є статичними, вони не можуть адаптуватися до змін в середовищі або до вимог завдання, що може перешкоджати подальшому покращенню стратегії.

Приклад графіку процесу навчання за допомогою DDQNwithGA моделі зображено на рисунку 5.11.

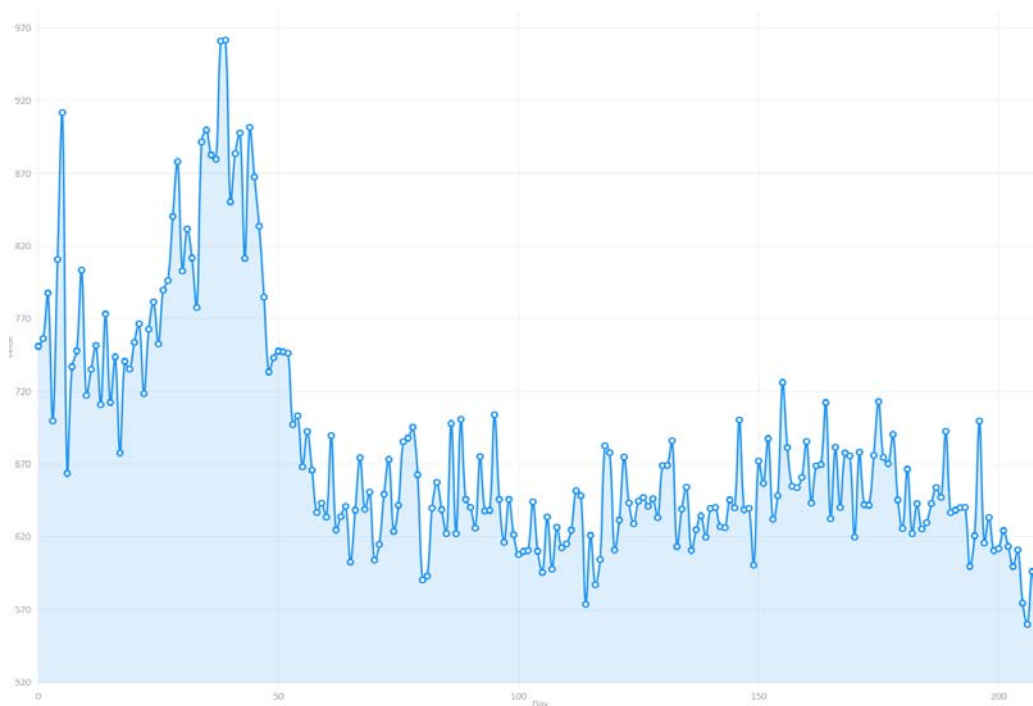


Рисунок 5.11 – Графік процесу навчання за допомогою DDQNwithGA моделі

Спостерігається значне коливання на ранніх стадіях навчання, яке згодом стабілізується. Це може означати, що DDQNwithGA ефективно використовується для пошуку оптимальної стратегії, де генетичний алгоритм допомагає оптимізувати вибір дій у середовищі з навчанням з підкріпленням. Генетичний алгоритм допомагає виявити більш ефективні гіперпараметри для DDQN, сприяючи швидшій та стабільнішій конвергенції.

На другому графіку (рис.5.12), який відображає триваліший часовий період, ми бачимо зниження помилок з часом, але з меншою волатильністю порівняно з коротшим часовим відрізком. Це вказує на те, що модель може бути більш стабільною та ефективною на довгій дистанції, хоча певні періоди коливань все ще присутні, що може свідчити про постійну експлорацію та адаптацію до можливих змін у середовищі.

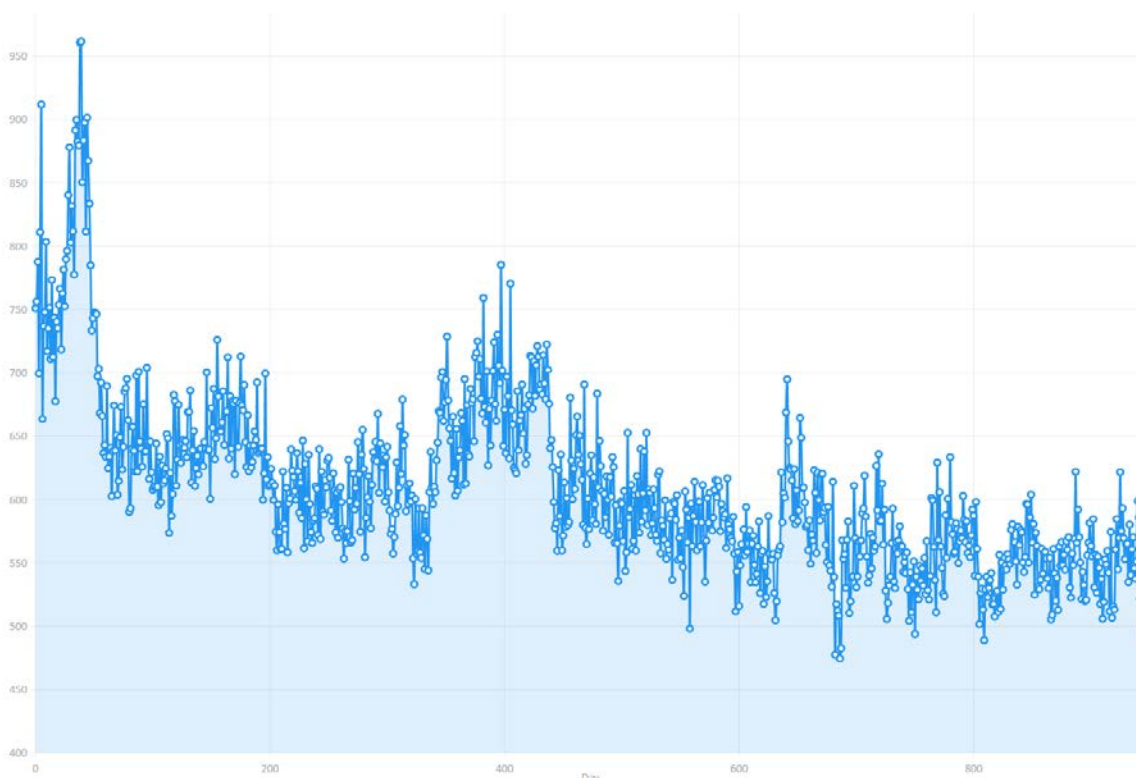


Рисунок 5.12 – Графік довшого процесу навчання за допомогою DDQNwithGA моделі

Зменшення волатильності помилок у довгостроковій перспективі може також вказувати на те, що DDQNwithGA досягла більш оптимальної політики, але досі продовжує пристосовуватися та покращувати свої

параметри для досягнення ще кращих результатів. Наявність генетичного алгоритму дозволяє зберігати рівень експлорації в процесі навчання, що є важливим для запобігання застрягання в локальних оптимумах та для підтримки здатності до узагальнення при змінних умовах середовища.

5.3 Висновок до розділу

Генетичні алгоритми та нейроеволюція демонструють потенціал для вирішення оптимізаційних задач шляхом експлорації різноманітних рішень та їх пристосування до складних умов середовища. Але, їх ефективність може бути обмеженою, коли вони стикаються з високою динамікою середовища та необхідністю швидкого реагування на зміни, що може вести до стагнації або навіть зростання помилок після певного часу навчання. Це говорить про можливу потребу в додаткових механізмах для уникнення локальних оптимумів та підтримки адаптивності алгоритму.

Інтеграція нейронних мереж з підкріпленням із генетичними алгоритмами для динамічного налаштування гіперпараметрів показує здатність до ефективного навчання та адаптації. DDQNwithGA, зокрема, демонструє до підтримки стабільної продуктивності на довгій дистанції, зменшує кількість помилок продовжуючи пошук оптимальних рішень, що може бути корисним для складних задач, де середовище постійно змінюється.

Загалом, результати вказують на те, що динамічна адаптація гіперпараметрів і продовження експлорації навіть після досягнення стабілізації є ключовими для покращення ефективності навчання алгоритмів машинного навчання, особливо в умовах, що постійно змінюються.

Крім того модель DDQNwithGA може використовуватись в задачах, де необхідне постійне навчання, бо вона вміє підбирати необхідний рівень дослідження, регуляризації і оцінки нагороди автоматично та без переналаштування, в більшості можливих ситуацій.

6 РОБОТА КОРИСТУВАЧА З БІБЛІОТЕКОЮ МОДЕЛІ

6.1 Вимоги до користувача

Для ефективної роботи з бібліотекою EvolutionNetwork.DDQNwithGA користувач повинен відповідати наступним вимогам:

а) основні знання в області програмування:

- володіння мовою C#. Бібліотека та модель написана на мові C#, тому користувач повинен мати досвід роботи з цією мовою програмування та розуміння основних концепцій, таких як класи, інтерфейси та колекції;

- знання основ роботи з .NET Framework або .NET Core. Бібліотека розроблена для платформи .NET, тому користувач повинен бути знайомий з роботою в цьому середовищі;

б) знання в області машинного навчання:

- основи навчання з підкріпленням. Користувач повинен розуміти принципи навчання з підкріпленням, зокрема, як система навчається на основі винагороди за здійснені дії;

- розуміння концепцій нейронних мереж. Базове розуміння того, як працюють нейронні мережі, включаючи передачу даних через шари та процес навчання за допомогою зворотного поширення помилки;

в) знайомство з генетичними алгоритмами. Користувач повинен мати уявлення про те, як працюють генетичні алгоритми, включаючи процеси селекції, кросоверу (злиття) та мутації, для оптимізації рішень;

г) навички роботи з бібліотекою:

- інтеграція з існуючими проєктами. Користувач повинен знати, як інтегрувати зовнішні бібліотеки в свої проєкти.

д) технічні вимоги:

- наявність розробницького середовища. Користувач повинен мати встановлене і налаштоване середовище для розробки на C#, наприклад Visual Studio або Visual Studio Code.

– комп'ютер з достатніми ресурсами. Для компіляції та виконання програм, що використовують DDQNwithGA, може знадобитися комп'ютер з достатніми обчислювальними ресурсами.

6.2 Інструкція для користувача

Ця бібліотека дозволяє вирішувати складні завдання навчання з підкріпленням із можливістю налаштування елементів, таких як повторення пам'яті, критики та розрахунок винагороди. Для роботи з бібліотекою EvolutionNetwork.DDQNwithGA вам потрібно виконати наступні кроки.

Крок 1. Створення моделі DDQNwithGA.

Основне створення моделі.

Для створення базової моделі DDQNwithGA вам потрібно визначити розміри шарів нейронної мережі, обсяги пам'яті та за бажанням гіперпараметри для генетичного алгоритму.

Розміри шарів. Визначте масив цілих чисел, що представляють розміри кожного шару у вашій нейронній мережі, включаючи вхідний і вихідний шари як мінімум.

Обсяги пам'яті. Вкажіть максимальний обсяг пам'яті для зберігання досвідів, розмір пакету для повторення пам'яті та мінімальний обсяг пам'яті, необхідний перед початком навчання з повторенням пам'яті.

Гіперпараметри. За бажанням визначте гіперпараметри для генетичного алгоритму, включаючи ймовірність мутацій і рівень дослідження.

Приклад:

```
int[] layerSizes = { inputSize, hiddenLayer1Size, outputSize };
uint maxMemoryCapacity = 10000;
uint batchMemorySize = 32;
uint minMemoryToStartMemoryReplayLearning = 1000; // Необов'язково
```

```

DDQNwithGAModel          model          =          new
DDQNwithGACreator().CreateDDQNwithGA(layerSizes,  maxMemoryCapacity,
batchMemorySize, minMemoryToStartMemoryReplayLearning);

```

Розширене створення моделі з користувацькими гіперпараметрами.

Для більшого контролю над гіперпараметрами генетичного алгоритму передайте словник GenHyperparameter до значень double при створенні моделі.

Приклад:

```

var hyperparameters = new Dictionary<GenHyperparameter, double>
{
    { GenHyperparameter.exploration, 0.1 },
    { GenHyperparameter.learningRate, 0.001 }
};

DDQNwithGAModel          model          =          new
DDQNwithGACreator().CreateDDQNwithGA(layerSizes,      hyperparameters,
maxMemoryCapacity, batchMemorySize);

```

Крок 2. Клонування моделі DDQNwithGA.

Ви можете клонувати існуючу модель DDQNwithGA безпосередньо або створити нову модель шляхом злиття двох батьківських моделей.

Пряме клонування. Створює точну копію існуючої моделі і додає мутації.

Клонування шляхом злиття. Поєднує характеристики двох батьківських моделей в нову модель і додає мутації.

Приклад:

```

DDQNwithGAModel          clone          =          new
DDQNwithGACreator().CloneDDQNwithGA(originalModel);

DDQNwithGAModel          childModel     =          new
DDQNwithGACreator().CloneDDQNwithGA(motherModel, fatherModel);

```

Крок 3. Навчання моделі.

Після створення або клонування моделі використовуйте її для виконання дій, отримання винагород і навчання з досвіду.

Вибір дії. Передайте поточний стан і цільове значення моделі, щоб вирішити про дію.

Результати дії. Оновіть модель результатами здійсненої дії, включаючи новий стан і отриману винагороду.

Приклад:

```
int action = model.ChooseAction(currentState, targetValue);
model.ActionResultHandler(newState, episodeRewardTarget,
rewardTarget, bonusReward);
```

Крок 4. Налаштування моделі.

Бібліотека підтримує налаштування критика, калькулятора винагороди та нагадування про досвід пам'яті.

Критик. Реалізує користувацьку логіку для оцінки правильності дій.

Калькулятор винагороди. Налаштовує розрахунок винагороди на основі результатів дій.

Нагадування про досвід пам'яті. Визначає, як нагадувати схожі минулі досвіди для впливу на майбутні рішення.

Для використання цих можливостей реалізуйте відповідні інтерфейси (IDDQNwithGACritic, IDDQNwithGACustomRewardCalculator, IDDQNwithGACustomRemindExperiencesDefinder) і ініціалізуйте модель з вашими реалізаціями.

Приклад:

```
model.InitCritic(new CustomCritic());
model.InitRewardCalculator(new CustomRewardCalculator());
model.InitRemindExperiencesDefinder(new
CustomRemindExperiencesDefinder());
```

6.3 Висновок до розділу

Цей посібник демонструє основні функціональні можливості бібліотеки `EvolutionNetwork.DDQNwithGA` для створення та навчання моделей DDQN з оптимізацією за допомогою генетичних алгоритмів. Налаштуйте моделі додатково, реалізувавши інтерфейси для критиків, калькуляторів винагород і нагадувань про досвід, щоб адаптувати процес навчання до вашої конкретної проблемної області.

ВИСНОВКИ

Модель демонструє здатність ефективно взаємодіяти з динамічним середовищем, пристосовувати стратегії поведінки, та оптимізувати рішення у змінних умовах. Модель показує значний потенціал для дослідження штучного життя та еволюційних процесів через моделювання. Ефективність розробленої системи була оцінена на задачі, що вимагає високої адаптивності і продемонструвала, як генетичний алгоритм допомагає виявити більш ефективні гіперпараметри для DDQN, сприяючи швидшій та стабільнішій конвергенції та оптимізації процесу навчання. Результати показують зменшення помилок у довгостроковій перспективі, що свідчить про досягнення моделлю більш оптимальної політики.

Робота відкриває нові перспективи для розробки більш ефективних систем глибокого навчання, які здатні самостійно оптимізувати свої параметри для досягнення найкращих результатів у різноманітних задачах. Важливою перевагою підходу є його універсальність та потенційне застосування в різних областях, включаючи робототехніку, фінансовий аналіз, медичне діагностування, та інші сфери, де використовуються комплексні системи штучного інтелекту. Інтеграція генетичних алгоритмів з навчанням з підкріпленням відкриває нові можливості для покращення продуктивності та адаптивності нейронних мереж.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Deep Reinforcement Learning using Genetic Algorithm for Parameter Optimization [Electronic resource]. – Access mode: <https://ar5iv.labs.arxiv.org/html/1905.04100>.
2. Automatic Hyperparameter Optimization Using Genetic Algorithm in Deep Reinforcement Learning for Robotic Manipulation Tasks [Electronic resource]. – Access mode: <https://ar5iv.labs.arxiv.org/html/2204.03656>.
3. Hyperparameter Optimization by Reinforcement Learning [Electronic resource]. – Access mode: <https://ar5iv.labs.arxiv.org/html/1906.11527>.
4. Comparison of an effectiveness of artificial neural networks for various activation functions [Electronic resource]. – Access mode: <https://ph.pollub.pl/index.php/jcsi/article/view/3069>.
5. Practical Issues in Temporal Difference Learning [Electronic resource]. – Access mode: <https://link.springer.com/article/10.1007/BF00992697>.
6. The models and structure of neural networks [Electronic resource]. – Access mode: <https://www.sciencedirect.com/science/article/abs/pii/B9780128197950000025?via%3Dihub>.
7. Exploration Strategy based on Validity of Actions in Deep Reinforcement Learning [Electronic resource]. – Access mode: <https://ieeexplore.ieee.org/document/9341014>.
8. Normalization Techniques in Training DNNs: Methodology, Analysis and Application [Electronic resource]. – Access mode: <https://ieeexplore.ieee.org/document/10056354>.
9. An Analysis of Regularization Methods in Deep Neural Networks [Electronic resource]. – Access mode: <https://ieeexplore.ieee.org/document/9342192>.
10. Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent [Electronic resource]. – Access mode: <https://epubs.siam.org/doi/10.1137/21M1453311>.

11. Multiagent Learning through Neuroevolution [Electronic resource]. – Access mode: https://link.springer.com/chapter/10.1007/978-3-642-30687-7_2.
12. Learning with genetic algorithms [Electronic resource]. – Access mode: <https://link.springer.com/article/10.1007/BF00113894>.
13. Reinforcement Learning [Electronic resource]. – Access mode: <https://ieeexplore.ieee.org/document/712192>.
14. Variations on the Game of Life [Electronic resource]. – Access mode: https://link.springer.com/chapter/10.1007/978-1-84996-217-9_13.
15. Neural Style Transfer [Electronic resource]. – Access mode: <https://ieeexplore.ieee.org/document/8732370>.
16. Reward-based training of recurrent neural networks for cognitive and value-based tasks [Electronic resource]. – Access mode: <https://elifesciences.org/articles/21492>.
17. Delving Deep into Rectifiers [Electronic resource]. – Access mode: <https://ieeexplore.ieee.org/document/7410480>.

ДОДАТОК А
Текст програми

A.1 Клас реалізації паттерну композиції, навчання нейромережі

```

using EvolutionNetwork.DDQNwithGA.DDQN;
using EvolutionNetwork.DDQNwithGA.DDQN.DQNMethods;
using EvolutionNetwork.DDQNwithGA.DDQN.NN;
using EvolutionNetwork.DDQNwithGA.Interfaces;
using EvolutionNetwork.GenAlg;
using EvolutionNetwork.StaticClasses;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static EvolutionNetwork.GenAlg.HyperparameterGen;
namespace EvolutionNetwork.DDQNwithGA
{
    public class DDQNwithGAModel
    {
        private readonly Random random = new Random();
        private TDLearningMethod TDLearning = new TDLearningMethod();
        private RSEMethod RSEMethod = new RSEMethod();
        public IDDQNwithGACritic Critic;
        public IDDQNwithGACustomRewardCalculator RewardCalculator;
        public IDDQNwithGACustomRemindExperiencesDefinder RemindExperiencesDefinder;
        //Critic
        public bool IsActionError = false;

        // DQN
        public List<DDQNMemory> Memory { get; set; } = new List<DDQNMemory>();
        public uint MaxMemoryCapacity { get; private set; }
        public uint MinMemoryToStartMemoryReplayLearning { get; private set; }
        public bool IsMemoryEnough { get; private set; }
        public uint BatchMemorySize { get; private set; }
    }
}

```

```

public double TotalReward { get; private set; } = 0;
public double TotalActionsNum { get; private set; } = 0;

public double[] AfterActionState { get; private set; }
public double[] BeforeActionState { get; private set; }
public double TargetValueBeforeAction { get; private set; }
public int Action { get; private set; }

// NN
public NNLayers[] OnlineLayers { get; private set; }
public NNLayers[] TargetLayers { get; private set; }
public int[] LayersSizes { get; private set; }

public HyperparameterGen Gen;

internal NNInference NNInference;
internal NNTrainWithSGDM NNTrainWithSGDM;

public DDQNwithGAModel(HyperparameterGen gen, int[] layersSizes, uint
maxMemoryCapacity, uint batchSize, uint minMemoryToStartMemoryReplayLearning
= 0)
{
    this.Gen = gen;
    this.LayersSizes = layersSizes;
    InitNN();
    InitDDQN(layersSizes, maxMemoryCapacity, batchSize,
minMemoryToStartMemoryReplayLearning);
}

// Auxiliary method for updating the network and training
public void UpdateAndTrainIfNeeded()
{
    UpdateTargetNetwork();
    if (Memory.Count > MinMemoryToStartMemoryReplayLearning)

```

```

    {
        TrainFromMemoryReplay();
    }
}

public void UpdateMemory(double[] beforeMoveState, int action, double reward, double[]
afterMoveState, bool done)
{
    Memory.Add(new DDQNMemory(beforeMoveState, action, reward, afterMoveState,
done));
    if (Memory.Count > MaxMemoryCapacity)
    {
        Memory.RemoveAt(0); // Remove the oldest experience to not exceed the maximum
capacity
    }
}

private void UpdateTargetNetwork()
{
    for (int i = 0; i < OnlineLayers.Length; i++)
    {
        Array.Copy(OnlineLayers[i].weights, TargetLayers[i].weights,
OnlineLayers[i].weights.Length);
        Array.Copy(OnlineLayers[i].biases, TargetLayers[i].biases,
OnlineLayers[i].biases.Length);
    }
}

public int ChooseAction(double[] currentState, double targetValue)
{
    TargetValueBeforeAction = targetValue;
    BeforeActionState = currentState;
    TotalActionsNum++;
    IsActionError = false;
}

```

```

        if (IsMemoryEnough && random.NextDouble() <
Gen.HyperparameterChromosome[GenHyperparameter.remindProbability])
    {
        RSEMethod.DefineMostSimilarExperiences(currentState,
Gen.HyperparameterChromosome[GenHyperparameter.percentageOfSimilarExperiences],
Memory, RemindExperiencesDefinder);
        RSEMethod.RemindSimilarExperiencesDDQN(OnlineLayers, TargetLayers,
NNInference, TeachDDQNModel, TDLearning,
Gen.HyperparameterChromosome[GenHyperparameter.discountFactor]);
    }

    int decidedAction = -1;
    //// Epsilon-greedy choice

    if (random.NextDouble() <
Gen.HyperparameterChromosome[GenHyperparameter.exploration]) // Exploration
    {
        double[] qValuesOutput = NNInference.FeedForward(BeforeActionState,
OnlineLayers);
        //// Epsilon-greedy choice
        if (decidedAction == -1)
        {
            int randomIndex = random.Next(LayersSizes[^1]);
            decidedAction = randomIndex;
        }
        Action = decidedAction;
    }
    else
    {
        double[] qValuesOutput = NNInference.FeedForward(BeforeActionState,
OnlineLayers);
        decidedAction = NNInference.FindMaxIndexForFindAction(qValuesOutput);
    }
    if (Critic != null)

```

```

{
    IsActionError = Critic.IsDecidedActionError(decidedAction, BeforeActionState);
}
return decidedAction;
}

public void ActionResultHandler(double[] afterActionState, double episodeRewardTarget,
double rewardTarget, double bonusReward = 0)
{
    this.AfterActionState = afterActionState;

    bool done = false;
    if (episodeRewardTarget > 0)
    {
        done = true;
    }
    double reward = 0;
    if (RewardCalculator == null)
    {
        reward += CalculateReward(done, episodeRewardTarget, rewardTarget,
bonusReward);
    }
    else
    {
        reward += RewardCalculator.CalculateReward(done, episodeRewardTarget,
rewardTarget, bonusReward);
    }
    reward = Normalizer.TanhNormalize(reward);
    if (Memory.Count > MinMemoryToStartMemoryReplayLearning)
    {
        IsMemoryEnough = true;
    }
    TrainOnline(reward, done);
    if (done)

```

```

{
    if (!IsMemoryEnough)
    {
        UpdateTargetNetwork();
    }
}

RegisterActionResult(reward, done);
}

private void RegisterActionResult(double reward, bool done)
{
    UpdateMemory(BeforeActionState, Action, reward, AfterActionState, done);
}

private double CalculateReward(bool done, double episodeSuccessValue, double
targetValue, double bonus)
{
    double reward = 0;
    if (done)
    {
        reward = TotalReward / TotalActionsNum;
        if (reward > 0)
        {
            double episodeReward = 0;
            for (int i = 0; i < episodeSuccessValue; i++)
            {
                episodeReward += reward *
Gen.HyperparameterChromosome[GenHyperparameter.genDoneBonusA] /
Math.Pow(Gen.HyperparameterChromosome[GenHyperparameter.genDoneBonusB], i);
            }
            reward = episodeReward;
        }
        else
        {
            reward = 0;
        }
    }
}

```

```

        reward += bonus;
        TotalReward = 0;
        TotalActionsNum = 0;
    }
    else
    {
        reward = targetValue - TargetValueBeforeAction;
        if (IsActionError)
        {
            reward -= Gen.HyperparameterChromosome[GenHyperparameter.errorFine];
        }
        else
        {
            reward += Gen.HyperparameterChromosome[GenHyperparameter.correctBonus];
        }
        reward += bonus;
        TotalReward += reward;
    }
    return reward;
}

```

```

private void TeachDDQNModel(double[] stateInput, double[] targetQValues)
{
    double[] predictedQValues = NNInference.FeedForward(stateInput, OnlineLayers);
    NNTrainWithSGDM.BackPropagationSGDM(predictedQValues, targetQValues,
OnlineLayers);
}
private void TrainFromMemoryReplay()
{
    var random = new Random();
    var sampledExperiences = new List<DDQNMemory>();
    for (int i = 0; i < BatchMemorySize; i++)
    {
        int index = random.Next(Memory.Count);

```

```

        sampledExperiences.Add(Memory[index]);
    }
    foreach (var experience in sampledExperiences)
    {
        double[] state = experience.BeforeActionState;
        double[] targetQValues =
TDLearning.TDLearningDDQN(Gen.HyperparameterChromosome[GenHyperparameter.discount
Factor],
        experience.BeforeActionState, experience.Done, experience.Reward,
        experience.DecidedAction, experience.AfterActionState, OnlineLayers, TargetLayers,
NNInference);
        TeachDDQNModel(state, targetQValues);
    }
}
private void TrainOnline(double reward, bool done)
{
    double[] targetQValues =
TDLearning.TDLearningDDQN(Gen.HyperparameterChromosome[GenHyperparameter.discount
Factor],
        BeforeActionState, done, reward, Action, AfterActionState, OnlineLayers,
TargetLayers, NNInference);
    TeachDDQNModel(BeforeActionState, targetQValues);
}

// Auxiliary method for initializing the model
private void InitDDQN(int[] layerSizes, uint maxMemoryCapacity, uint batchSize,
uint minMemoryToStartMemoryReplayLearning)
{
    LayersSizes = new int[layerSizes.Length];
    Array.Copy(layerSizes, LayersSizes, layerSizes.Length);
    InitNetworkLayers();
    NNTrainWithSGDM.InitVelocities(OnlineLayers);
    this.MaxMemoryCapacity = maxMemoryCapacity;
    this.BatchMemorySize = batchSize;
}

```

```
        this.MinMemoryToStartMemoryReplayLearning
minMemoryToStartMemoryReplayLearning;
    }
    private void InitNN()
    {
        NNInference = new NNInference(Gen, LayersSizes);
        NNTrainWithSGDM = new NNTrainWithSGDM(Gen);
    }
    public void InitNetworkLayers()
    {
        OnlineLayers = new NNLayers[LayersSizes.Length];
        for (int i = 0; i < LayersSizes.Length; i++)
        {
            OnlineLayers[i] = new NNLayers(LayersSizes[i], i < LayersSizes.Length - 1 ? LayersSizes[i
+ 1] : 0);
            OnlineLayers[i].InitializeWeightsHe(); // Initialization of weights for online layers
        }

        TargetLayers = new NNLayers[LayersSizes.Length];
        for (int i = 0; i < LayersSizes.Length; i++)
        {
            TargetLayers[i] = new NNLayers(LayersSizes[i], i < LayersSizes.Length - 1 ? LayersSizes[i
+ 1] : 0);
            TargetLayers[i].InitializeWeightsHe(); // Initialization of weights for target layers
        }
    }

    public void InitCritic(IDDQNwithGACritic critic)
    {
        this.Critic = critic ?? throw new ArgumentNullException(nameof(critic));
    }

    public void InitRewardCalculator(IDDQNwithGACustomRewardCalculator
rewardCalculator)
    {

```

```

        this.RewardCalculator = rewardCalculator ?? throw new
ArgumentNullException(nameof(rewardCalculator));
    }
    public void
InitRemindExperiencesDefinder(IDDQNwithGACustomRemindExperiencesDefinder
remindExperiencesDefinder)
    {
        this.RemindExperiencesDefinder = remindExperiencesDefinder ?? throw new
ArgumentNullException(nameof(remindExperiencesDefinder));
    }
}
}

```

A.2 Клас створення нейромережі та її нащадків

```

using EvolutionNetwork.DDQNwithGA.DDQN;
using EvolutionNetwork.DDQNwithGA.Interfaces;
using EvolutionNetwork.GenAlg;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static EvolutionNetwork.GenAlg.HyperparameterGen;

namespace EvolutionNetwork.DDQNwithGA
{
    public class DDQNwithGACreator
    {
        public DDQNwithGAModel CreateDDQNwithGA(int[] layerSizes,
Dictionary<GenHyperparameter, double> startHyperparameterScatterDict, uint
maxMemoryCapacity, uint batchSize, uint minMemoryToStartMemoryReplayLearning
= 0)
        {

```

```

        ValidateLayerSizes(layerSizes);
        HyperparameterGen Gen = new HyperparameterGen();
        Gen.StartHyperparameterScatter(startHyperparameterScatterDict);
        DDQNwithGAModel dDDQNwithGA = new DDQNwithGAModel(Gen, layerSizes,
maxMemoryCapacity, batchMemorySize, minMemoryToStartMemoryReplayLearning);
        return dDDQNwithGA;
    }

    public DDQNwithGAModel CreateDDQNwithGA(int[] layerSizes, uint maxMemoryCapacity,
uint batchMemorySize, uint minMemoryToStartMemoryReplayLearning = 0, double
startHyperparameterScatter = 0)
    {
        ValidateLayerSizes(layerSizes);
        HyperparameterGen Gen = new HyperparameterGen();
        Gen.StartHyperparameterScatter(startHyperparameterScatter);
        DDQNwithGAModel dDDQNwithGA = new DDQNwithGAModel(Gen, layerSizes,
maxMemoryCapacity, batchMemorySize, minMemoryToStartMemoryReplayLearning);
        return dDDQNwithGA;
    }

    public DDQNwithGAModel CloneDDQNwithGA(DDQNwithGAModel original)
    {
        return CopyFrom(original);
    }

    public DDQNwithGAModel CloneDDQNwithGA(DDQNwithGAModel mother,
DDQNwithGAModel father)
    {
        Random random = new Random();

        DDQNwithGAModel baseParent = random.Next(0, 2) == 0 ? mother : father;
        DDQNwithGAModel otherParent = baseParent == mother ? father : mother;

        return MergeFrom(baseParent, otherParent);
    }

    private void ValidateLayerSizes(int[] layerSizes)

```

```

{
    if (layerSizes.Length < 2)
    {
        throw new ArgumentException("You should have at least input and output layers");
    }
}

private DDQNwithGAModel CopyFrom(DDQNwithGAModel original)
{
    HyperparameterGen Gen = new HyperparameterGen(original.Gen);
    int[] layerSizes = (int[])original.LayersSizes.Clone();
    uint batchSize = original.BatchMemorySize;
    uint maxMemoryCapacity = original.MaxMemoryCapacity;
    uint minMemoryToStartMemoryReplayLearning =
original.MinMemoryToStartMemoryReplayLearning;

    DDQNwithGAModel dDQNwithGACopy = new DDQNwithGAModel(Gen, layerSizes,
maxMemoryCapacity, batchSize, minMemoryToStartMemoryReplayLearning);

    InheritMemory(dDQNwithGACopy, original);
    InheritVelocities(dDQNwithGACopy, original);
    InheritNetworkLayers(dDQNwithGACopy, original);

    if (original.Critic != null)
    {
        dDQNwithGACopy.InitCritic(original.Critic);
    }
    if (original.RewardCalculator != null)
    {
        dDQNwithGACopy.InitRewardCalculator(original.RewardCalculator);
    }
    if (original.RemindExperiencesDefinder != null)
    {

```

```

dDQNwithGACopy.InitRemindExperiencesDefinder(original.RemindExperiencesDefinder);
    }
    dDQNwithGACopy.UpdateAndTrainIfNeeded();
    return dDQNwithGACopy;
}

private DDQNwithGAModel MergeFrom(DDQNwithGAModel baseParent,
DDQNwithGAModel otherParent)
{
    HyperparameterGen Gen = new HyperparameterGen(baseParent.Gen,
otherParent.Gen);
    int[] layerSizes = (int[])baseParent.LayersSizes.Clone();
    uint batchSize = baseParent.BatchMemorySize;
    uint maxMemoryCapacity = baseParent.MaxMemoryCapacity;
    uint minMemoryToStartMemoryReplayLearning =
baseParent.MinMemoryToStartMemoryReplayLearning;

    DDQNwithGAModel dDQNwithGACopy = new DDQNwithGAModel(Gen, layerSizes,
maxMemoryCapacity, batchSize, minMemoryToStartMemoryReplayLearning);

    InheritMemory(dDQNwithGACopy, baseParent, otherParent);

    InheritVelocities(dDQNwithGACopy, baseParent);
    InheritNetworkLayers(dDQNwithGACopy, baseParent);
    if (baseParent.Critic != null)
    {
        dDQNwithGACopy.InitCritic(baseParent.Critic);
    }
    if (baseParent.RewardCalculator != null)
    {
        dDQNwithGACopy.InitRewardCalculator(baseParent.RewardCalculator);
    }
}

```

```

        if (baseParent.RemindExperiencesDefinder != null)
        {

dDQNwithGACopy.InitRemindExperiencesDefinder(baseParent.RemindExperiencesDefinder);

        }

        dDQNwithGACopy.UpdateAndTrainIfNeeded();
        return dDQNwithGACopy;
    }

    private void InheritVelocities(DDQNwithGAModel clone, DDQNwithGAModel original)
    {
        clone.NNTrainWithSGDM.VelocitiesWeights = new
double[original.NNTrainWithSGDM.VelocitiesWeights.Length][][];
        for (int layer = 0; layer < original.NNTrainWithSGDM.VelocitiesWeights.Length; layer++)
        {
            clone.NNTrainWithSGDM.VelocitiesWeights[layer] = new
double[original.NNTrainWithSGDM.VelocitiesWeights[layer].Length][];
            for (int neuron = 0; neuron <
original.NNTrainWithSGDM.VelocitiesWeights[layer].Length; neuron++)
            {
                clone.NNTrainWithSGDM.VelocitiesWeights[layer][neuron] = new
double[original.NNTrainWithSGDM.VelocitiesWeights[layer][neuron].Length];
                Array.Copy(original.NNTrainWithSGDM.VelocitiesWeights[layer][neuron],
clone.NNTrainWithSGDM.VelocitiesWeights[layer][neuron],
original.NNTrainWithSGDM.VelocitiesWeights[layer][neuron].Length);
            }
        }

        clone.NNTrainWithSGDM.VelocitiesBiases = new
double[original.NNTrainWithSGDM.VelocitiesBiases.Length][];
        for (int layer = 0; layer < original.NNTrainWithSGDM.VelocitiesBiases.Length; layer++)
        {
            clone.NNTrainWithSGDM.VelocitiesBiases[layer] = new
double[original.NNTrainWithSGDM.VelocitiesBiases[layer].Length];

```

```

        Array.Copy(original.NNTrainWithSGDM.VelocitiesBiases[layer],
clone.NNTrainWithSGDM.VelocitiesBiases[layer],
original.NNTrainWithSGDM.VelocitiesBiases[layer].Length);
    }
}

private void InheritNetworkLayers(DDQNwithGAModel clone, DDQNwithGAModel original)
{
    clone.InitNetworkLayers();
    CopyNNLayers(clone, original);
}

private void InheritMemory(DDQNwithGAModel clone, DDQNwithGAModel original)
{
    clone.Memory = original.Memory.Select(m => (DDQNMemory)m.Clone()).ToList();
}

private void InheritMemory(DDQNwithGAModel clone, DDQNwithGAModel mother,
DDQNwithGAModel father)
{
    Random random = new Random();
    if (mother.Memory.Count > father.Memory.Count)
    {
        clone.Memory = mother.Memory.Select(m => (DDQNMemory)m.Clone()).ToList();
        for (int i = 0; i < father.Memory.Count; i++)
        {
            if (random.Next(0, 2) == 0)
            {
                if (clone.Memory.Count < i)
                {
                    clone.Memory[i] = (DDQNMemory)father.Memory[i].Clone();
                }
                else
                {
                    DDQNMemory tamp = (DDQNMemory)father.Memory[i].Clone();
                    clone.UpdateMemory(tamp.BeforeActionState,          tamp.DecidedAction,
tamp.Reward, tamp.AfterActionState, tamp.Done);

```

```

        }
    }
}
else
{
    clone.Memory = father.Memory.Select(m => (DDQNMemory)m.Clone()).ToList();
    for (int i = 0; i < mother.Memory.Count; i++)
    {
        if (random.Next(0, 2) == 0)
        {
            if (clone.Memory.Count < i)
            {
                clone.Memory[i] = (DDQNMemory)mother.Memory[i].Clone();
            }
            else
            {
                DDQNMemory tamp = (DDQNMemory)mother.Memory[i].Clone();
                clone.UpdateMemory(tamp.BeforeActionState,          tamp.DecidedAction,
tamp.Reward, tamp.AfterActionState, tamp.Done);
            }
        }
    }
}

private void CopyNNLayers(DDQNwithGAModel clone, DDQNwithGAModel original)
{
    for (int k = 0; k < clone.OnlineLayers.Length; k++)
    {
        Array.Copy(original.OnlineLayers[k].weights,          clone.OnlineLayers[k].weights,
original.OnlineLayers[k].weights.Length);
        Array.Copy(original.OnlineLayers[k].neurons,          clone.OnlineLayers[k].neurons,
original.OnlineLayers[k].neurons.Length);
    }
}

```

```

        Array.Copy(original.OnlineLayers[k].biases,
                    clone.OnlineLayers[k].biases,
original.OnlineLayers[k].biases.Length);
        clone.OnlineLayers[k].size = original.OnlineLayers[k].size;
        clone.OnlineLayers[k].nextSize = original.OnlineLayers[k].nextSize;
    }
    for (int k = 0; k < clone.TargetLayers.Length; k++)
    {
        Array.Copy(original.TargetLayers[k].weights,
                    clone.TargetLayers[k].weights,
original.TargetLayers[k].weights.Length);
        Array.Copy(original.TargetLayers[k].neurons,
                    clone.TargetLayers[k].neurons,
original.TargetLayers[k].neurons.Length);
        Array.Copy(original.TargetLayers[k].biases,
                    clone.TargetLayers[k].biases,
original.TargetLayers[k].biases.Length);
        clone.TargetLayers[k].size = original.TargetLayers[k].size;
        clone.TargetLayers[k].nextSize = original.TargetLayers[k].nextSize;
    }
}
}
}
}

```

ДОДАТОК Б
Слайди презентації



Дипломна робота на тему:

Інтеграція генетичного алгоритму в нейронну мережу з навчанням з підкріпленням

Виконав:
студент IV курсу, групи КНТ-110
Керівник: к.т.н., доцент

Максим ТРІНКАЛЬ
Ольга ГЛАДКОВА



1

Рисунок Б.1 – Слайд 1

Мета та завдання роботи

Метою даної дипломної роботи є **розробка та дослідження моделі, що інтегрує генетичний алгоритм в нейронні мережі з навчанням з підкріпленням**, з метою оптимізації процесу навчання та підвищення загальної ефективності алгоритмів у вирішенні складних задач.

Для досягнення поставленої мети були визначені наступні **завдання**:

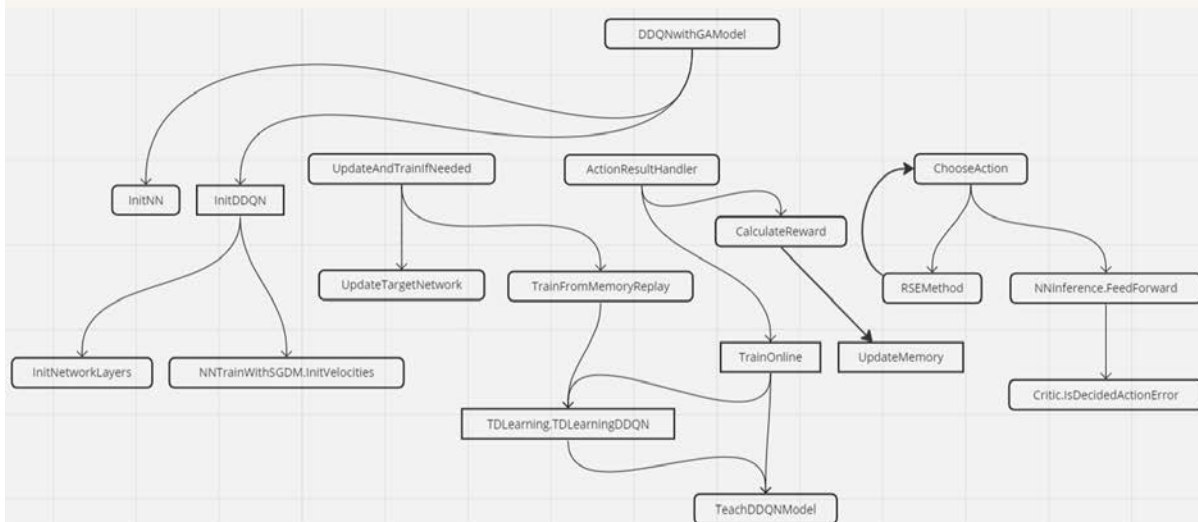
- розробка програмної архітектури системи;
- створення генетичного алгоритму з метою оптимізації параметрів і структури нейронної мережі;
- реалізація системи навчання з підкріпленням, яка дозволяє нейронній мережі вирішувати задачі, ефективно взаємодіючи з середовищем;
- оцінка ефективності розробленої системи на зразку задачі, яка вимагає адаптації стратегій поведінки в динамічних умовах;
- аналіз можливостей моделювання процесів еволюції та штучного життя за допомогою розробленої системи.



2

Рисунок Б.2– Слайд 2

Взаємозв'язок розроблених методів



3

Рисунок Б.3 – Слайд 3

Переваги поєднання DDQN та генетичного алгоритму

- ▶ Автоматизація відбору гіперпараметрів
- ▶ Збалансування дослідження і використання
- ▶ Покращення продуктивності
- ▶ Адаптація до змінних умов
- ▶ Паралельні обчислення



4

Рисунок Б.4 – Слайд 4

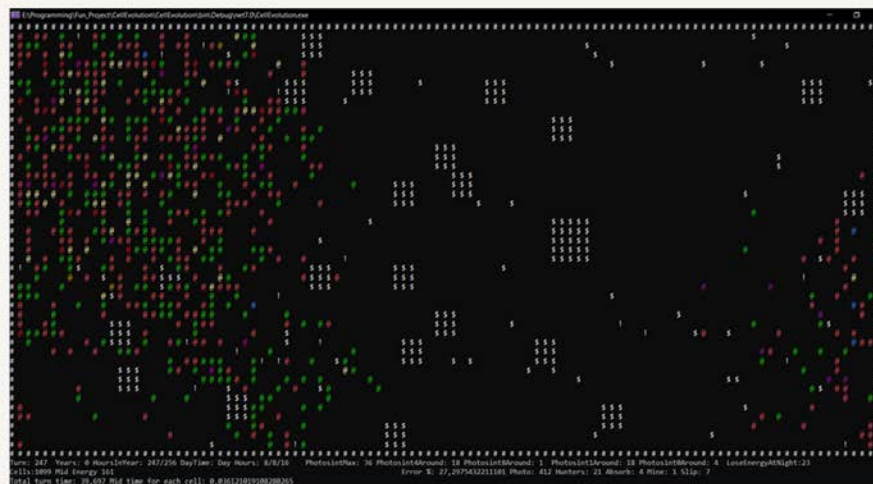
Потреба у великих обчислювальних ресурсах

DDQNwithGA вимагає значних обчислювальних ресурсів та оперативної пам'яті, що робить її не найкращім вибором для невеликих та нескладних задач.

5

Рисунок Б.5 – Слайд 5

Задача для тестування – Симуляція еволюції



6

Рисунок Б.6 – Слайд 6

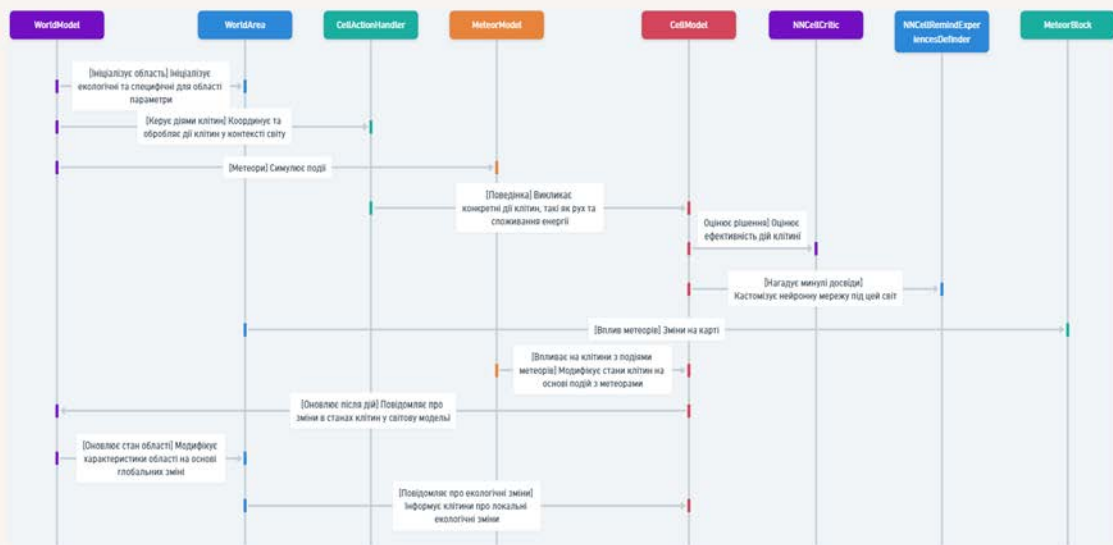
Архітектура симуляції



7

Рисунок Б.7 – Слайд 7

Взаємодія об'єктів світу



8

Рисунок Б.8 – Слайд 8

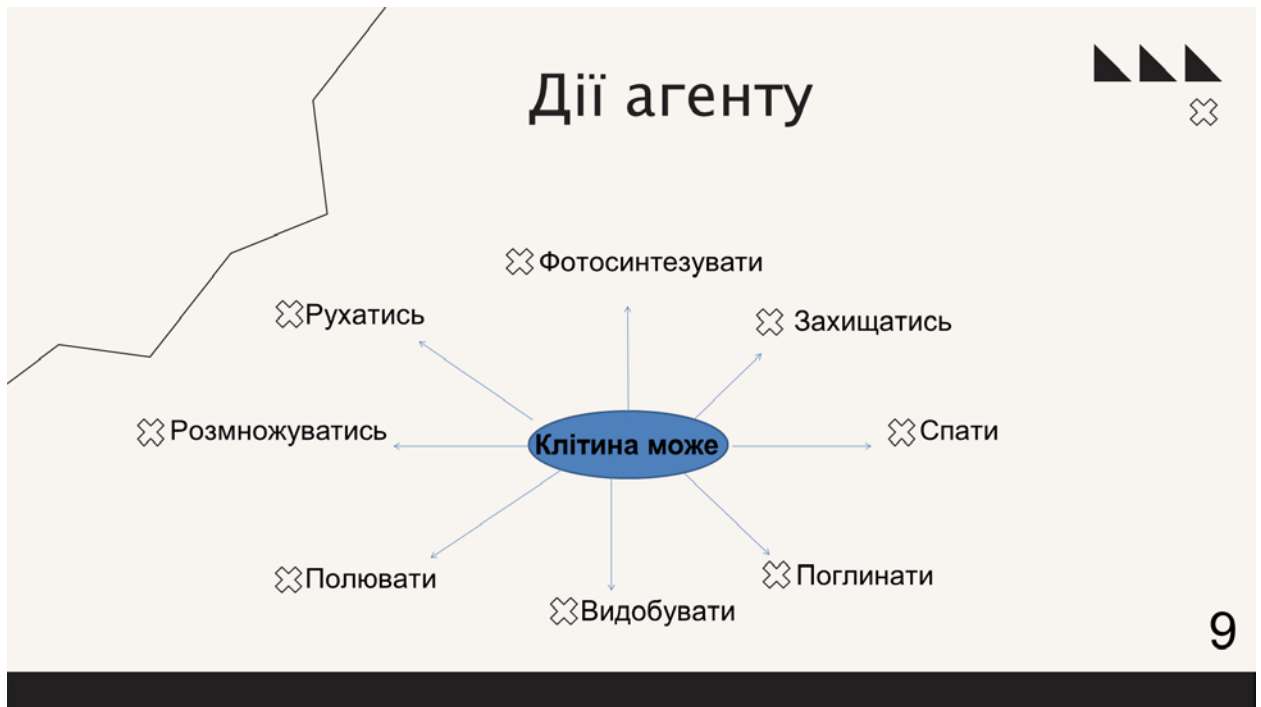


Рисунок Б.9 – Слайд 9



Рисунок Б.10 – Слайд 10



Рисунок Б.11 – Слайд 11

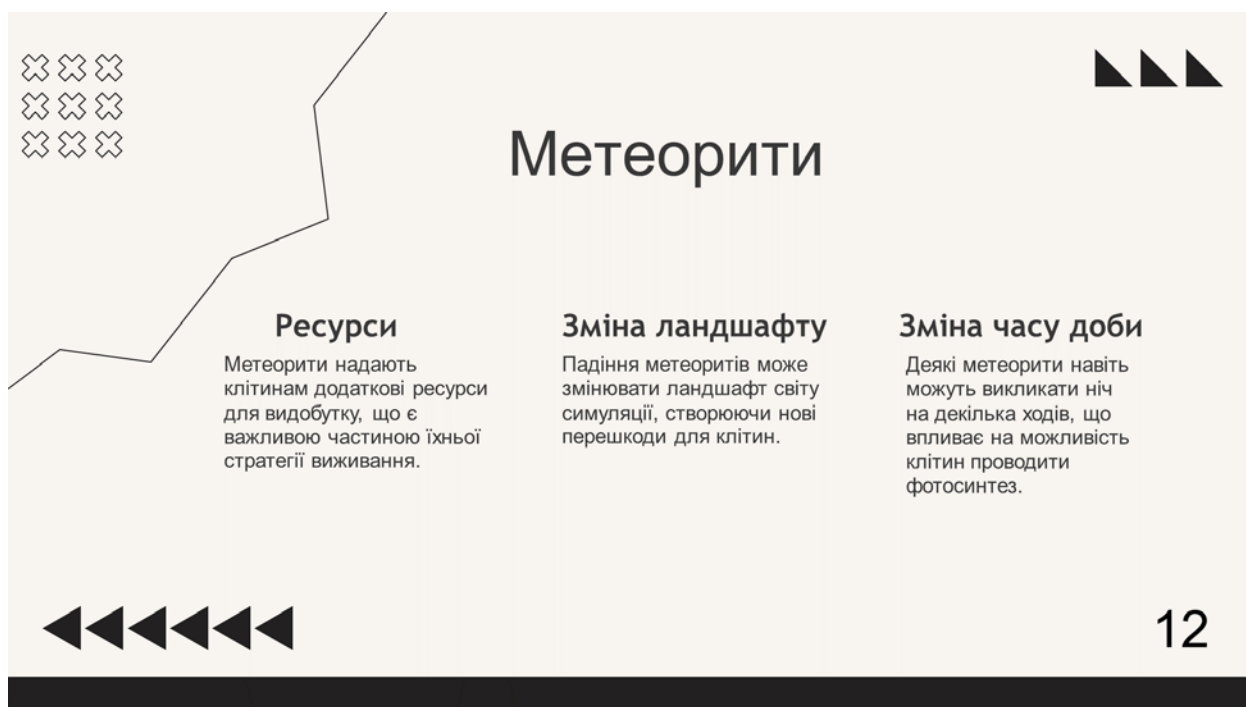
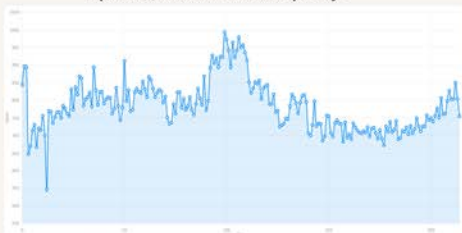


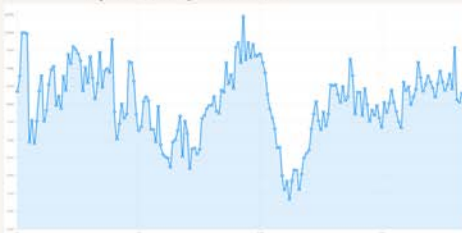
Рисунок Б.12 – Слайд 12

Порівняння результатів тестування

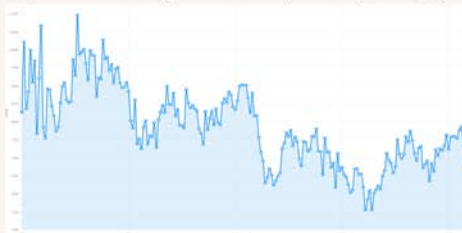
Приклад генетичного алгоритму:



Приклад нейроеволюції:



Приклад навчання з використанням нейронної мережі з підкріпленням:



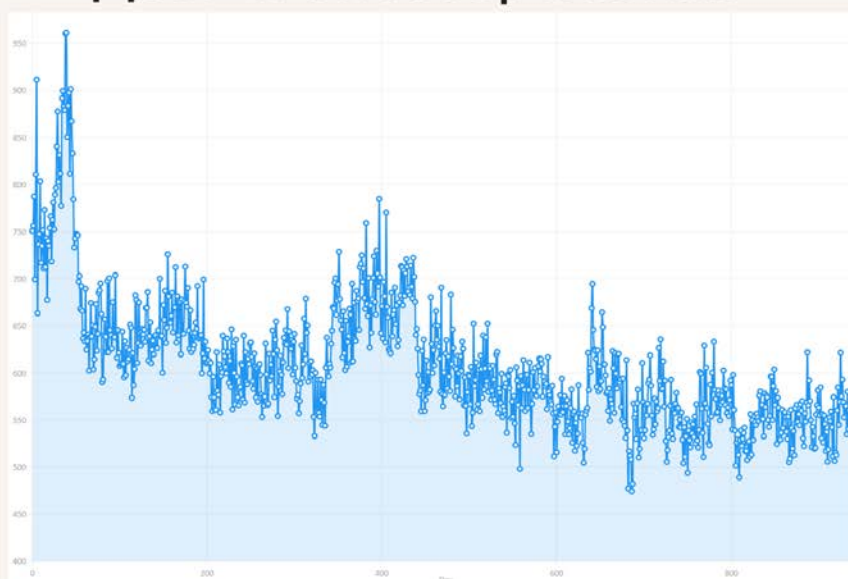
Приклад DDQNwithGA моделі:



13

Рисунок Б.13 – Слайд 13

Довше спостереження



14

Рисунок Б.14 – Слайд 14

Висновок тестування

- ✕ **Ефективне навчання та адаптація**
- ✕ **Підтримки стабільної продуктивності на довгій дистанції**
- ✕ **Зменшення волатильності помилок**

15

Рисунок Б.15 – Слайд 15

Застосування моделі

1

Робототехніка

Модель може бути застосована для навчання роботів ефективно взаємодіяти з динамічним середовищем та адаптувати свою поведінку до змінних умов.

2

Фінансовий аналіз

Модель може бути використана для прогнозування фінансових ринків та прийняття оптимальних інвестиційних рішень в умовах невизначеності.

3

Медичне діагностування

Модель може допомогти в розробці інтелектуальних систем медичної діагностики, здатних виявляти захворювання на ранніх стадіях та пропонувати ефективні методи лікування.



16

Рисунок Б.16 – Слайд 16

Висновки

Розроблена система демонструє здатність:

- ✕ Ефективно взаємодіяти з динамічним середовищем
- ✕ До адаптації стратегії поведінки
- ✕ Оптимізувати рішення у змінних умовах
- ✕ Генетичний алгоритм допомагає виявити більш ефективні гіперпараметри та стратегії рішення
- ✕ Сприяння швидшій та стабільнішій конвергенції



17

Рисунок Б.17 – Слайд 17



Дякую за увагу



Рисунок Б.18 – Слайд 18