Факультет комп’ютерних наук і технологій

Кафедра програмних засобів

До захисту допущено: Завідувач кафедри

\_

«\_ » 2024 р.

Дипломна робота

**на здобуття ступеня бакалавра спеціальності 121 «Інженерія програмного забезпечення»**

**освітня програма «Інженерія програмного забезпечення »**

**на тему: «Інтеграція генетичного алгоритму в нейронну мережу з навчанням з підкріпленням»**

Виконав:

Студент IV курсу, групи КНТ-110

Трінкаль Максим Євгенович

Керівник:

Рецензент:

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

Запоріжжя – 2024 року

Факультет комп’ютерних наук і технологій

Кафедра програмних засобів

Рівень вищої освіти перший рівень

Напрям підготовки 121 Програмна інженерія

|  |  |  |  |
| --- | --- | --- | --- |
| ЗАТВЕРДЖУЮ | | | |
| Завідувач кафедри | | | |
|  |  |  |  |
| (підпис) | | | |
|  |  | “ “ | 2024 р. |
|  |  | **ЗАВДАННЯ** |  |
| **на дипломну роботу студенту** | | | |
|  | | | |
|  |  | (прізвище, ім'я, по–батькові) |  |
| 1. Тема роботи | *Інтеграція генетичного алгоритму в нейронну мережу* | | |
| *з навчанням з підкріпленням* | | | |
| керівник роботи |  | | |
| (прізвище, ім'я, по–батькові, науковий ступінь, вчене звання) | | | |
| затверджені наказом по університету від " " 2024р. № \_ | | | |
| 2. Термін подання студентом роботи | | | *„ ” 2024 р.* |
| 3. Вихідні дані до роботи | |  | |
| 4. Зміст роботи |  |  |  |
|  | | | |
|  | | | |
|  | | | |
|  | | | |
| 5. Перелік ілюстративного матеріалу ( із зазначенням плакатів, презентації тощо) | | | |

|  |  |
| --- | --- |
|  | |
| Дата видачі завдання | " " р. |

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № з/п | Назва етапів виконання дипломної роботи | Термін виконання етапів роботи | Примітка |
| *1* |  |  |  |
| *2* |  |  |  |
| *3* |  |  |  |
|  |  |  |  |
| *4* |  |  |  |
|  |  |  |  |
| *5* |  |  |  |
| *6* |  |  |  |
| *7* |  |  |  |
| *8* |  |  |  |
| *9* |  |  |  |
| *10* |  |  |  |

|  |  |  |
| --- | --- | --- |
| **Студент** |  | Максим ТРІНКАЛЬ |
| (підпис) |  | (Власне ім’я, ПРІЗВИЩЕ) |
| **Керівник роботи** |  |  |
| (підпис) |  | (Власне ім’я, ПРІЗВИЩЕ) |

## АНОТАЦІЯ

Ця дипломна робота присвячена дослідженню та розробці методу інтеграції генетичного алгоритму в нейронні мережі з навчанням з підкріпленням. В роботі аналізуються основні принципи навчання з підкріпленням, особливості генетичних алгоритмів та їх застосування для оптимізації параметрів нейронних мереж. Розглядається можливість поєднання цих двох підходів з метою підвищення ефективності навчання моделей глибокого навчання.

У теоретичній частині роботи описано теоретичні основи нейронних мереж, навчання з підкріпленням та генетичних алгоритмів, а також розглянуто існуючі дослідження та розробки в даній області. В експериментальній частині представлено розроблену систему, яка використовує генетичний алгоритм для оптимізації структури та параметрів нейронної мережі, що навчається з підкріпленням. Аналізуються результати застосування розробленого методу на задачах навчання з підкріпленням, включаючи управління агентами в середовищах з високим рівнем складності.

Результати дослідження показують, що інтеграція генетичного алгоритму в нейронні мережі з навчанням з підкріпленням дозволяє підвищити ефективність навчання за рахунок більш ефективного пошуку оптимальних параметрів та структури мережі. Такий підхід може бути використаний для розв'язання широкого спектра задач, де необхідно досягти високої точності та адаптивності моделей.

Ключові слова: нейронні мережі, навчання з підкріпленням, генетичні алгоритми, оптимізація, глибоке навчання.

## ABSTRACT

This thesis is devoted to the research and development of a method for integrating a genetic algorithm into neural networks with reinforcement learning. The paper analyzes the main principles of reinforcement learning, features of genetic algorithms and their application to optimize the parameters of neural networks. The possibility of combining these two approaches is considered in order to increase the effectiveness of learning deep learning models.

The theoretical part of the work describes the theoretical foundations of neural networks, reinforcement learning, and genetic algorithms, as well as reviews existing research and development in this area. The experimental part presents a developed system that uses a genetic algorithm to optimize the structure and parameters of a reinforcement learning neural network. The results of applying the developed method to reinforcement learning task, including managing agents in environments with a high level of complexity, are analyzed.

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.

Keywords: neural networks, reinforcement learning, genetic algorithms, optimization, deep learning.

## ЗМІСТ

[АНОТАЦІЯ 4](#_Toc160103475)

[ABSTRACT 5](#_Toc160103476)

[ЗМІСТ 6](#_Toc160103477)

[ВСТУП 8](#_Toc160103478)

[1 ПОСТАНОВКА ЗАДАЧІ 9](#_Toc160103479)

[2 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ 11](#_Toc160103480)

[2.1 Глибоке навчання з підкріпленням із застосуванням генетичного алгоритму для оптимізації параметрів 11](#_Toc160103481)

[2.2 Автоматична оптимізація гіперпараметрів за допомогою генетичного алгоритму у глибокому навчанні з підкріпленням для робототехнічних маніпуляцій 13](#_Toc160103482)

[2.3 Hyp-RL: Оптимізація гіперпараметрів за допомогою навчання з підкріпленням 14](#_Toc160103483)

[Висновки до розділу 16](#_Toc160103484)

[3 ІНСТУМЕНТИ ТА АЛГОРИТМИ 17](#_Toc160103485)

[3.1 Середовище розробки Microsoft Visual Studio та C# 17](#_Toc160103486)

[3.1.1 Microsoft Visual Studio 18](#_Toc160103487)

[3.1.2 C# 18](#_Toc160103488)

[3.2 Double Deep Q-Network 19](#_Toc160103489)

[3.3 Алгоритм генетичного програмування 20](#_Toc160103490)

[3.4 Temporal Difference Learning 23](#_Toc160103491)

[3.5 Система обчислення нагороди 24](#_Toc160103492)

[3.6 Нормалізація MinMax та Tanh 26](#_Toc160103493)

[3.6.1 MinMax Нормалізація 26](#_Toc160103494)

[3.6.2 Tanh Нормалізація 26](#_Toc160103495)

[3.7 Stochastic Gradient Descent з Momentum 27](#_Toc160103496)

[3.8 Регулярізатори Dropout, L2, Weights Noise 28](#_Toc160103497)

[3.8.1 Dropout 28](#_Toc160103498)

[3.8.2 L2 Регулярізація (Ridge Regression) 28](#_Toc160103499)

[3.8.3 Weights Noise (Шум в Вагах) 29](#_Toc160103500)

[3.8.4 Поєднання методів регулярізації 29](#_Toc160103501)

[3.9 Функція активізації Swish 29](#_Toc160103502)

[3.10 Ініціалізатор He 30](#_Toc160103503)

[3.11 Epsilon-greedy стратегія дослідження 31](#_Toc160103504)

[3.12 Remind Similar Expierence алгоритм 31](#_Toc160103505)

[3.13 Підхід до навчання 32](#_Toc160103506)

[3.13.1 Online Learning 32](#_Toc160103507)

[3.13.2 Replay Learning (Experience Replay) 33](#_Toc160103508)

[3.13.3 Гібридний підхід 33](#_Toc160103509)

[3.14 Висновок до розділу 34](#_Toc160103510)

[4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ 36](#_Toc160103511)

[4.1 DDQNwithGA бібліотека 36](#_Toc160103512)

[4.1.1 DDQNwithGAModel (class) 37](#_Toc160103513)

[4.1.1.1 Конструктор DDQNwithGAModel 37](#_Toc160103514)

[4.1.1.2 Методи DDQNwithGAModel для управління Memory 38](#_Toc160103515)

[4.1.1.3 Методи DDQNwithGAModel для навчання моделі 38](#_Toc160103516)

[4.1.1.4 Методи DDQNwithGAModel для вибору дії 38](#_Toc160103517)

[4.1.1.5 Методи DDQNwithGAModel для оновлення мережі 38](#_Toc160103518)

[4.1.1.6 Інші методи 38](#_Toc160103519)

[4.1.1.7 Поля і властивості DDQNwithGAModel 39](#_Toc160103520)

[4.1.1.7.1 Поля та властивості DDQNwithGAModel для даних та стану 39](#_Toc160103521)

[4.1.1.7.2 Поля для нейронної мережі 40](#_Toc160103522)

[4.1.1.7.3 Поля для кастомізації поведінки 40](#_Toc160103523)

[4.1.2 DDQNwithGACreator 41](#_Toc160103524)

[4.1.2.1 Методи створення моделі 41](#_Toc160103525)

[4.1.2.2 Методи клонування моделі 41](#_Toc160103526)

[4.1.1 Внутрішні методи DDQNwithGACreator 42](#_Toc160103527)

[4.1.2 Ресурси нейроної мережі 42](#_Toc160103528)

[4.1.2.1 NNInference 42](#_Toc160103529)

[4.1.3.1.1 Конструктор NNInference 42](#_Toc160103530)

[4.1.3.1.2 Методи для прямого розповсюдження 43](#_Toc160103531)

[4.1.3.1.3 Вибір відповіді 43](#_Toc160103532)

[4.1.2.2 NNLayers 43](#_Toc160103533)

[4.1.3.2.1 Поля та властивості 44](#_Toc160103534)

[4.1.3.2.2 Конструктор 44](#_Toc160103535)

[4.1.3.2.3 Методи 44](#_Toc160103536)

[4.1.2.3 NNTrainWithSGDM 44](#_Toc160103537)

[4.1.3.4.1 Поля 45](#_Toc160103538)

[4.1.3.4.2 Конструктор 45](#_Toc160103539)

[4.1.3.4.3 Методи 45](#_Toc160103540)

[4.1.2.4 DDQNMemory 46](#_Toc160103541)

[4.1.2.4.1 Поля 46](#_Toc160103542)

[4.1.2.4.2 Конструктор 46](#_Toc160103543)

[4.1.2.4.3 Методи 47](#_Toc160103544)

[4.1.3 HyperparameterGen 47](#_Toc160103545)

[4.1.3.1 Поля та Константи 47](#_Toc160103546)

[4.1.3.2 Конструктори 48](#_Toc160103547)

[4.1.3.3 Методи 48](#_Toc160103548)

[4.1.3.4 Enum GenHyperparameter 48](#_Toc160103549)

[4.1.4 Інтерфейси 50](#_Toc160103550)

[4.1.4.1 IDDQNwithGACritic 50](#_Toc160103551)

[4.1.5.1.1. Методи 50](#_Toc160103552)

[4.1.4.2 IDDQNwithGACustomRemindExperiencesDefinder 51](#_Toc160103553)

[4.1.4.2.1 Методи 51](#_Toc160103554)

[4.1.4.3 IDDQNwithGACustomRewardCalculator 51](#_Toc160103555)

[4.1.4.3.1 Методи 52](#_Toc160103556)

[4.1.5 Алгоритми для оптимізації навчання 52](#_Toc160103557)

[4.1.5.3 Activation 55](#_Toc160103558)

[4.1.5.4 Normalizer 55](#_Toc160103559)

[4.1.5.5 Regularization 57](#_Toc160103560)

[4.2 Симуляція 58](#_Toc160103561)

[4.2.1 SimulationHandler 58](#_Toc160103562)

[4.2.2 WorldResources 59](#_Toc160103563)

[4.2.2.1 Cell 61](#_Toc160103564)

[4.2.2.1.1 CellModel 61](#_Toc160103565)

[4.2.2.1.2 CellActionHandler 62](#_Toc160103566)

[4.2.2.1.3 Класи для налаштування моделі DDQNwithGA 63](#_Toc160103567)

[4.2.2.2 Meteor 64](#_Toc160103568)

[4.2.2.3 WorldModel 65](#_Toc160103569)

[4.2.2.4 WorldArea 66](#_Toc160103570)

[4.2.3 Constants 68](#_Toc160103571)

[4.2.4 StatDatabase 68](#_Toc160103572)

[4.2.4.1 Поля 69](#_Toc160103573)

[4.2.4.2 Методи 69](#_Toc160103574)

[4.2.5 Renderer 69](#_Toc160103575)

[4.2.5.1 Основні компоненти та методи: 70](#_Toc160103576)

[4.2.5.2 Синхронізація та візуалізація 70](#_Toc160103577)

[4.3 Програма для тестування 70](#_Toc160103578)

[4.3.1 ErrorStat Form 70](#_Toc160103579)

[4.3.2 ErrorStatModel 71](#_Toc160103580)

[5 ВСЕСВІТ ТА ТЕСТУВАННЯ 73](#_Toc160103581)

[5.1 Всесвіт 73](#_Toc160103582)

[5.1.1 Легенда візуалізації 73](#_Toc160103583)

[5.1.2 Клітини 74](#_Toc160103584)

[5.1.3 Час доби 77](#_Toc160103585)

[5.1.4 Метеорити 77](#_Toc160103586)

[5.1.5 Отруєні області 78](#_Toc160103587)

[5.1.6 Приклад симуляції 78](#_Toc160103588)

[5.2 Тестування 82](#_Toc160103589)

[6 РОБОТА КОРИСТУВАЧА З БІБЛІОТЕКОЮ МОДЕЛІ 89](#_Toc160103590)

[ВИСНОВКИ 94](#_Toc160103591)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 95](#_Toc160103592)

[Додаток А 97](#_Toc160103593)

[Додаток Б 108](#_Toc160103594)

## ВСТУП

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

Ця дипломна робота присвячена розробці та аналізу моделі, яка інтегрує генетичний алгоритм в нейронні мережі з навчанням з підкріпленням, щоб покращити процес навчання та підвищити ефективність рішень, а також вивченню потенціалу моделювання процесів штучного життя та еволюції.

Перший розділ визначає актуальність та постановку задачі дослідження, включаючи аспекти дослідження штучного життя та еволюції. Другий розділ присвячений огляду літератури та аналізу існуючих рішень у сфері нейронних мереж з навчанням з підкріпленням, генетичних алгоритмів та їх застосування у моделюванні штучного життя. У третьому розділі представлено теоретичні основи використаних методів, зокрема, детальний опис розробленого алгоритму, механізмів навчання нейронних мереж з підкріпленням та їх роль у дослідженні штучного життя та еволюційних процесів. Четвертий розділ описує програмну реалізацію розробленої системи, зокрема архітектуру, алгоритми, інтерфейс користувача та можливості моделювання еволюційних процесів. У п'ятому розділі аналізуються результати тестування системи, зокрема оцінка ефективності інтеграції генетичного алгоритму в процесі навчання та його вплив на моделювання штучного життя й еволюційних процесів. У висновках подаються основні результати дослідження та пропозиції щодо подальшого використання й розвитку розробленої системи, а також перспективи її застосування у галузі штучного інтелекту, дослідження штучного життя та вивчення еволюційних алгоритмів.

## 1 ПОСТАНОВКА ЗАДАЧІ

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 задачі:

- Розробка програмної архітектури, яка забезпечує інтеграцію генетичного алгоритму з нейронними мережами з навчанням з підкріпленням.

- Створення ефективного генетичного алгоритму для оптимізації параметрів та структури нейронної мережі.

- Реалізація системи навчання з підкріпленням, що дозволяє нейронній мережі ефективно взаємодіяти з середовищем для вирішення задач.

- Оцінка ефективності розробленої системи на прикладі задачі, що вимагає адаптації стратегій поведінки в динамічних умовах.

- Аналіз можливостей моделювання процесів штучного життя та еволюції за допомогою розробленої системи.

Система повинна забезпечувати наступні можливості:

- Ефективне навчання нейронної мережі за допомогою генетичного алгоритму для вирішення складних задач.

- Візуалізація процесу навчання та еволюції нейронної мережі в реальному часі.

- Моделювання процесів штучного життя та еволюційних стратегій у вирішенні задач.

- Аналіз та порівняння ефективності різних конфігурацій нейронної мережі та параметрів генетичного алгоритму.

Висновки до розділу:

В результаті була поставлена задача розробки та аналізу моделі з інтеграцією генетичного алгоритму в нейронні мережі з навчанням з підкріпленням, що дозволяє не лише підвищити ефективність процесу навчання, але й відкриває можливості для дослідження штучного життя та еволюційних процесів. Ця модель зможе бути використана для вирішення різноманітних задач, де потрібна висока адаптивність та оптимізація рішень в динамічних умовах, а також для подальших досліджень у сфері штучного інтелекту, штучного життя та еволюційних алгоритмів.

## 2 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

У цьому розділі розглядаються три наукові роботи, що стосуються оптимізації гіперпараметрів за допомогою генетичних алгоритмів та глибокого навчання з підкріпленням у контексті робототехнічних маніпуляцій та інших задач.

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

Автори: Адарш Сегал, Хунг Ман Ла, Сушіл Дж. Луїс, Хай Нгуєн

Стаття "Глибоке навчання з підкріпленням із застосуванням генетичного алгоритму для оптимізації параметрів" розглядає інноваційний підхід до оптимізації параметрів у системах глибокого навчання з підкріпленням. Розробка цього методу була ініційована командою дослідників, до якої входять Адарш Сегал, Хунг Ман Ла, Сушіл Дж. Луїс та Хай Нгуєн. Основною метою було підвищення ефективності навчання штучних інтелектуальних систем шляхом удосконалення процесу підбору оптимальних параметрів.

В основі дослідження лежить інтеграція генетичного алгоритму у процес глибокого навчання з підкріпленням. Це дозволило автоматизувати процес вибору параметрів, які найкраще впливають на продуктивність навчання. Генетичний алгоритм, імітуючи процеси природного відбору та еволюції, проводить пошук оптимальних рішень шляхом генерації, комбінування та мутації існуючих параметрів. Цей метод демонструє високу адаптивність до різних завдань глибокого навчання, пропонуючи ефективні рішення для складних проблем.

Дослідження показало значне підвищення продуктивності навчальних алгоритмів за рахунок точної настройки параметрів, що дозволяє досягти кращих результатів у коротші терміни. Використання генетичного алгоритму виявилося особливо корисним у задачах, де традиційні методи оптимізації не забезпечують достатньої ефективності через велику кількість параметрів та їх складні взаємозв'язки.

Важливою перевагою представленого підходу є його універсальність та можливість застосування в різних областях, включаючи робототехніку, фінансовий аналіз, медичне діагностування та інші сфери, де використовуються комплексні системи штучного інтелекту. Результати дослідження відкривають нові перспективи для розробки більш ефективних і адаптивних систем глибокого навчання, здатних самостійно оптимізувати свої параметри для досягнення найкращих результатів у різноманітних задачах.

Окрім технічних аспектів, стаття також акцентує увагу на потенційних застосуваннях розробленого методу, демонструючи його значний вплив на прогрес у сфері штучного інтелекту. Дослідження становить значний внесок у розвиток глибокого навчання з підкріпленням, пропонуючи ефективний інструмент для вдосконалення алгоритмів і підвищення їхньої адаптивності та продуктивності.

### Автоматична оптимізація гіперпараметрів за допомогою генетичного алгоритму у глибокому навчанні з підкріпленням для робототехнічних маніпуляцій

Автори: Хунг Ла, Адарш Сегал, Ніколас Вард, Сушіл Луїс

Ця робота запропонувала метод, базований на глибокому детермінованому політичному градієнті (DDPG) та відтворенні досвіду з огляду (HER), який використовує генетичний алгоритм (GA) для точного налаштування значень гіперпараметрів. Метод (GA+DDPG+HER) було експериментально перевірено на шести завданнях роботичної маніпуляції: FetchReach; FetchSlide; FetchPush; FetchPick&Place; Відкриття дверей; і AuboReach. Аналіз цих результатів продемонстрував значне підвищення продуктивності та зменшення часу навчання. Також порівнювали та надали докази того, що GA+DDPG+HER кращий за існуючі методи.

Вступ. Навчальні агенти можуть використовувати навчання з підкріпленням (RL), щоб вирішувати свої дії, використовуючи функцію винагороди. Однак процес навчання значно впливає на вибір значень гіперпараметрів, що використовуються в алгоритмі навчання. Із цього випливає натхнення для покращення алгоритму RL. Хоча алгоритми, як-от підйомники на пагорбах, можуть бути корисними в певних ситуаціях, вони не дуже корисні в складних ситуаціях, як-от роботичні завдання, що використовують RL, через відсутність ясного зв'язку між змінами продуктивності та зміною налаштувань гіперпараметрів.

Генетичний алгоритм (GA) використовується для оптимізації гіперпараметрів, що може бути використано для оптимізації гіперпараметрів у порівнянних типах проблем. GA має здатність налаштовувати гіперпараметри один раз і використовувати їх безкінечно в подальших навчаннях RL, зберігаючи обчислювальні ресурси, які інакше витрачалися б кожного разу, коли RL застосовується до агента.

Основні внески цього дослідження включають розробку нового алгоритму для автоматичного налаштування гіперпараметрів, перевірку алгоритму на шести симульованих та одному реальному завданні, оцінку процесу навчання на основі різних гіперпараметрів та порівняння ефективності DDPG+HER із використанням гіперпараметрів, знайдених GA+DDPG+HER, в як симульованих, так і реальних завданнях маніпуляції.

Дослідження демонструє, що GA+DDPG+HER перевершує інші підходи, зокрема, у завданні FetchReach, GA+DDPG+HER показав кращі результати. Відкритий вихідний код доступний за посиланням <https://github.com/aralab-unr/ga-drl-aubo-ara-lab>.

У вступі також розглядаються додаткові застосування навчання з підкріпленням (RL) у різноманітних додатках, включаючи роботичний настільний теніс, планування хірургічних роботів, швидке планування руху в бімануальному перехопленні голки та навігацію в акваторії. Ці застосування використовують RL як мотивуючу альтернативу автоматизації ручної праці.

Дослідження підкреслює значення налаштування гіперпараметрів, особливо при використанні машинного навчання, що може мати значний вплив на продуктивність алгоритму. Із цього виникло натхнення для покращення алгоритму RL, що призвело до розробки нового підходу до автоматичного налаштування гіперпараметрів, який було застосовано до DDPG+HER.

Ця робота вносить важливий вклад у дослідження оптимізації в області глибокого навчання з підкріпленням, зокрема, застосування генетичного алгоритму для підвищення ефективності та скорочення часу навчання в завданнях роботичної маніпуляції.

.

### Hyp-RL: Оптимізація гіперпараметрів за допомогою навчання з підкріпленням

Автори: Хаді С. Джомаа, Джосіф Грабоцка, Ларс Шмідт-Тіме

У цій роботі ми розглядаємо проблему оптимізації гіперпараметрів як послідовну задачу прийняття рішень про те, який гіперпараметр тестувати далі, і звертаємося до неї за допомогою навчання з підкріпленням. Таким чином, наша модель не повинна покладатися на евристичну функцію придбання, як SMBO, а може вчитися, які гіперпараметри тестувати далі, на основі подальшого зниження втрати валідації, яке вони зрештою призведуть, або тому, що вони дають хороші моделі самі по собі, або тому, що вони дозволяють політиці вибору гіперпараметрів побудувати кращу наближену модель, яка здатна обирати кращі гіперпараметри пізніше. Експерименти на великому наборі з 50 наборів даних демонструють, що наш метод перевершує сучасні підходи до навчання гіперпараметрів.

Оптимізація гіперпараметрів грає значну роль у загальній продуктивності моделей машинного навчання і може бути головним фактором, який визначає, чи навчена модель виявиться передовою або просто помірною. Зростаючі розміри наборів даних та складності моделей призводять до збільшення часу навчання та потребують більше ресурсів, що робить оптимізацію гіперпараметрів ще складнішим завданням. При автоматизації налаштування гіперпараметрів практичні рішення повинні бути надійними, масштабованими та зберігати низький обчислювальний бюджет, одночасно забезпечуючи сильну кінцеву продуктивність. Було запропоновано кілька рішень для налаштування гіперпараметрів протягом років, які відрізняються за обчислювальною складністю та масштабованістю. Деякі більш традиційні підходи включають ручний пошук, пошук по сітці та випадковий пошук, тоді як інші, такі як методи байєсової оптимізації, покладаються на ймовірнісні моделі, найчастіше гауссові процеси, для оптимізації очікуваного покращення наближеної моделі, розглядаючи функцію відгуку як чорний ящик.

Ми моделюємо проблему оптимізації гіперпараметрів як послідовну задачу прийняття рішень про те, який гіперпараметр тестувати далі, і підходимо до неї з точки зору навчання з підкріпленням. Це дозволяє нашій моделі не покладатися на евристичну функцію придбання, як у випадку з послідовною модельно-базованою байєсовою оптимізацією (SMBO), але натомість вчитися, які гіперпараметри слід тестувати далі, на основі того, яке зменшення втрати валідації вони зрештою призведуть. Такий підхід демонструє значне покращення продуктивності та скорочення часу навчання на широкому спектрі завдань маніпуляції з роботами. Також ми порівнюємо і надаємо докази, що наш підхід (GA+DDPG+HER) кращий, ніж існуючі методи.

### Висновки до розділу

У даному розділі було розглянуто три наукові роботи, присвячені оптимізації гіперпараметрів у контексті глибокого навчання з підкріпленням і застосування генетичних алгоритмів. Аналіз цих досліджень виявив значний потенціал інтеграції генетичних алгоритмів у процеси навчання з підкріпленням для ефективного вибору оптимальних параметрів систем штучного інтелекту.

Перша стаття демонструє, як генетичний алгоритм може бути використаний для автоматизації та оптимізації вибору гіперпараметрів, що підвищує ефективність навчання штучних інтелектуальних систем. Друге дослідження підтверджує, що застосування генетичного алгоритму разом із методом глибокого детермінованого політичного градієнту та відтворенням досвіду з огляду значно підвищує продуктивність та скорочує час навчання в завданнях роботичної маніпуляції. Третя стаття вносить інновацію, формулюючи оптимізацію гіперпараметрів як задачу навчання з підкріпленням, що дозволяє системі вчитися вибирати гіперпараметри, які найкраще сприяють зниженню втрати валідації.

Разом узяте, ці дослідження розкривають потенціал застосування генетичних алгоритмів і навчання з підкріпленням для покращення процесів автоматичного налаштування гіперпараметрів у системах штучного інтелекту. Такі підходи можуть мати широке застосування у різноманітних сферах, включаючи робототехніку, обробку мови, розпізнавання образів та інші області, де важлива висока адаптивність та ефективність навчальних алгоритмів.

## ІНСТУМЕНТИ ТА АЛГОРИТМИ

Програмний застосунок написаний на C# в операційній системі Windows. В якості середовища розробки було обрано Microsoft Visual Studio. Для тестуваня 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.

Метод Reinforcement Similar Experiences (RSE) - для визначення найбільш подібних досвідів та їх нагадування з метою покращення процесу навчання.

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

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, a 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 вона додає необхідну нелінійність до моделі, що дозволяє нейронній мережі навчатися складнішим та більш абстрактним відносинам між вхідними даними та виходом.

### Середовище розробки Microsoft Visual Studio та C#

### 3.1.1 Microsoft Visual Studio

Microsoft Visual Studio є легким, але потужним редактором коду, що підтримує різноманітні мови програмування та інструменти, включаючи C#. Завдяки своїй гнучкості та великому вибору розширень, Microsoft Visual Studio дозволяє легко налаштовувати середовище під конкретні потреби проекту.

Інтеграція з системами контролю версій, такими як Git дозволяє легко керувати версіями коду безпосередньо з редактора, включаючи коміти, пуши, пули, і перегляд змін. Також наявність вбудованих терміналів і розширений інструментарій для роботи з кодом роблять Microsoft Visual Studio відмінним вибором для розробки алгоритмів штучного інтелекту. Використання Microsoft Visual Studio як середовища розробки надає додаткову гнучкість завдяки підтримці великої кількості розширень та інструментів, призначених для специфічних потреб розробників ШІ, таких як інтеграція з системами машинного навчання, підтримка контейнерів Docker та інше.

Крім того Microsoft Visual Studio є повністю безкоштовним для використання та в той час Microsoft Visual Studio залишається легким і швидким, забезпечуючи високу продуктивність навіть на менш потужних комп'ютерах.

### 3.1.2 C#

C# оптимізований для високої продуктивності, що є критично важливим для обробки великих обсягів даних та виконання складних обчислень, які часто зустрічаються в проектах зі штучним інтелектом. Строга типізація та сучасні функції мови C# сприяють написанню надійного та безпечного коду, зменшуючи ймовірність помилок та вразливостей.

C# має велику екосистему і спільноту, що забезпечує доступ до широкого спектра бібліотек, фреймворків та інструментів. Це означає, що ви можете легко знайти готові рішення або бібліотеки для реалізації специфічних аспектів ваших проектів, таких як обробка даних, математичні операції, а також спеціалізовані бібліотеки для роботи з нейронними мережами та генетичними алгоритмами.

Одною з головних причин було те що C# ефективно підтримує багатопоточність, що є важливим для розробки високопродуктивних додатків у сфері штучного інтелекту. Багатопоточність дозволяє ефективніше використовувати ресурси процесора при тренуванні моделей або обробці великих масивів даних.

C# та .NET надають розробникам потужні засоби для створення безпечного коду, зокрема завдяки механізмам управління доступом, шифруванню та безпечному управлінню пам'яттю. Це дозволяє мінімізувати ризики, пов'язані з виконанням коду і обробкою даних.

C# та .NET оптимізовані для високої продуктивності, що важливо при розробці обчислювально інтенсивних додатків, якими є системи на основі нейронних мереж та генетичних алгоритмів. Ефективне управління пам'яттю і оптимізовані алгоритми в .NET дозволяють досягати високої швидкодії.

### Double Deep Q-Network

DDQN модель, або подвійна глибока Q-мережа (Double Deep Q-Network), представляє собою значне удосконалення традиційної глибокої Q-мережі (DQN), яке використовується в області глибокого навчання з підкріпленням. Основною інновацією DDQN є розробка механізму, що дозволяє зменшити проблеми, пов'язані з переоцінкою дій, які часто спостерігаються в DQN. В традиційних DQN-архітектурах одна й та ж нейронна мережа виконує дві функції: вибір дії та оцінка цінності цієї дії. Такий підхід може призвести до надмірно оптимістичних оцінок цінності дій, що у свою чергу може негативно вплинути на процес навчання.

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

Цей підхід дозволяє DDQN зменшити ризик переоцінки дій, оскільки оцінка цінності дії базується не на тих же вагах, що і вибір дії. Таким чином, DDQN забезпечує більш точну і об'єктивну оцінку дій, що сприяє більш стабільному та ефективному процесу навчання.

До переваг можна віднести: зменшення переоцінки дій: Центральна перевага DDQN моделі полягає в тому, що алгоритм зменшує ризик переоцінки дій, що є ключовим для оптимізації стратегій поведінки агента, більш стабільне навчання: Розділення процесу вибору дії та оцінки її цінності дозволяє досягти більшої стабільності процесу навчання, покращена збіжність: DDQN часто досягає кращої збіжності в порівнянні з традиційними DQN моделями, що призводить до вищої ефективності вирішення задач.

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

Через універсальність та ефективність ця модель була обрана в якості бази для створеня DDQNwithGA моделі. В розробленій мною моделі цільова нейромережа оновлюєтеся вже в нових поколінях агентів. Тобто агент впродовж свого житя навчає онлаїн мережу і якщо відбирється за рахунок генетичного алгоритму, як більш успішний екземпляр. Тоді його потомки переносять його досвід та набуті навички і роблять апгрейд цільової мережі на основі цієї онлаїн мережі. Крім того при схрещені змішуються згадки(Memory) агента батька і агента матері і завдяки цьому під час навчанню з використанням replay learning результати навчання завжди будуть хоч трохи відрізнятися, що робить алгоритм більш ефективним і допомагає зменшити ризик потрапляння в локальні мінімуми.

### Алгоритм генетичного програмування

Генетичний алгоритм — це метод оптимізації, натхненний процесами природного відбору, які відбуваються в біології. Цей підхід використовується для автоматичного покращення нейронних мереж з плином часу, імітуючи процес еволюції.

Генетичні алгоритми працюють за допомогою створення популяції кандидатів (у цьому контексті — нейронних мереж), і потім відбору, схрещування та мутації серед цих кандидатів для створення нових поколінь. Ціллю є знаходження найкращої можливої конфігурації нейронної мережі для даної задачі. Ось основні кроки процесу:

Ініціалізація: Створюється випадкова початкова популяція нейронних мереж (або індивідів).

Оцінка: Кожен індивід (нейронна мережа) в популяції оцінюється згідно з функцією пристосованості, яка визначає, наскільки добре мережа вирішує поставлену задачу.

Відбір: Найкращі індивіди вибираються як основа для створення наступного покоління. Вибір часто ґрунтується на принципах природного відбору, де "сильніші" індивіди мають більше шансів на репродукцію.

Схрещування: Вибрані індивіди комбінуються для створення нащадків. Це може включати обмін частинами їхньої "генетичної" структури (наприклад, параметрами нейронної мережі).

Мутація: З невеликою 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 3.3 – Архітектура генетичного алгоритму

В моєму варіанті цей алгоритм трохи відрізняється. Агент має змогу обирати хоче він розвиватись викоистовуючи тільки мутацію, чи перед тим схреститись з іншим агентом. Таким чином у випадках: де мала популяція агентів, для деяких завдань, де потрібне детальне знання про область проблеми, де кроссовер 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 кроссовера 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 має змогу за власним бажанням обмежувати розвиток тільки тим чи іншим шляхом, або поєднувати їх в якомусь іншому вигляді.

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

Хоча генетичні алгоритми є універсальними та можуть застосовуватися до широкого спектру оптимізаційних задач, вони можуть не бути найефективнішими в специфічних ситуаціях. В динамічних середовищах особливо де випадкові обставини мають великий вплив на агентів генетичний алгоритм підбирає найоптимальніше рішення проте не найефективніше. Агент навчений тільки за рахунок генетичного алгоритму не може аналізувати середовище, а лише виконує дію задану поточним геном.

Тому генетичний алгоритм використовується в розробленій мною моделі не використовується 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, проте це набагато полегшує підбір архітектури, дозволяє ускладнити нейрону мережу і робить її більш автономною. У перспективі рішення агентів більш варіативні і в деяких задачах з ризиком великої кількості локальних мінімумів динамічна зміна гіперпараметрів дозволяє швидше їх подолати.

Крім того при схрещуванні крім гіперпараметрів схрещується згадки(Memory). Згадки(Memory) використовується для навчання на отриманому досвіді агентом. Схрещування дозволяє швидко підвищити різноманітність учбових даних і дати кожному потомку трохи різну базу для навчання за рахунок чого їх стратегії трохи відрізняються, особливо в випадках з задачами, де можливо зберігати дуже обмежений сет даних. Це дозволяє націлити кожного агента на дослідження трохи різних стратегій, що в перспективі прискорює розвиток популяції загалом.

### Temporal Difference Learning

Temporal Difference Learning (TD Learning) є фундаментальною концепцією в області машинного навчання з підкріпленням, яка зосереджується на вивченні того, як передбачати майбутню винагороду, базуючись на поточних оцінках, без необхідності очікувати на кінцевий результат. Цей метод використовує різницю між послідовними прогнозами (тобто тимчасову різницю) для коригування оцінок вартості станів або дій, що дозволяє агенту поступово покращувати своє розуміння середовища, навіть перебуваючи в процесі виконання завдань.

Використання TD Learning має значні переваги, тому що інформація про винагороду може стати доступною не відразу, але розподіляється в часі. Такі ситуації часто зустрічаються у комплексних середовищах, де дії агента на початкових етапах можуть мати вплив на винагороди, які він отримає в майбутньому. TD Learning дозволяє агенту вчитися з кожним кроком, адаптуватися та оптимізувати свою стратегію в реальному часі, що значно підвищує ефективність навчання.

Однією з ключових переваг TD Learning є його здатність до навчання "на льоту", що робить його особливо корисним для розробки автономних систем, які повинні швидко адаптуватися до змін у середовищі. Цей метод дозволяє агенту краще розуміти довгострокові наслідки своїх дій, використовуючи оцінку винагороди, що покращує прийняття рішень та стратегічне планування.

Включення TD Learning підходить для вирішення складних завдань, де потрібно здійснювати прогнозування та адаптацію в реальному часі, і є важливим кроком до створення глибших та більш ефективних систем машинного навчання.

Проблемою цього алгоритму є складність визначення оптимального гіперпараметру discount, який відповідає за передбачувану нагороду. Занадто велике або навпаки занадто мале значення можуть тільки заважати агенту при вирішенні поставленної задачі через неадекватну оцінку своїх дій. За рахунок використання генетичного алгоритму в визначенні гіперпараметрів значення discount підбирається динамічно впродовж розвитку агенту і гнучко підлаштовуються під поставлену задачу. Крім того завдяки відбору різних успішних агентів з різними значеннями цього гіперпараметру кількість різних стратегій збільшується, що покращує якість результату навчання.

### Система обчислення нагороди

Механізм нагород, який є ключовим елементом для навчання агента в рамках парадигми підкріплювального навчання. Основна ідея полягає в тому, що агент отримує нагороди або покарання за виконані дії в залежності від їх впливу на досягнення цілі. Такий підхід дозволяє агенту вчитися на власному досвіді, вибудовуючи стратегію поведінки, яка максимізує сумарну винагороду.

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

За рахунок використання гіперпараметрів агнет може більш точно визначати стратегію розвитку та уникати локальних мінімумів. Це підштовхує агента до більш різноманітних стратегій. Наприклад в створеній мною симуляцій, за рахунок цих гіперпараметрів агенти ділились на декілька різновидів – деякі намагались зробити не багато дій але з високим результатом, якщо це їм не вдавалось – вони оновлювали спробу, а інші навпаки отримували більшу нагороду за нокопиченні бали за велику кількість дій і потім помножували їх завершуючи епізод.

Якщо епізод не завершений, поточна нагорода розраховується як різниця між поточною цінністю цілі (targetValue) і цінністю цілі перед дією (TargetValueBeforeAction). Якщо дія була помилковою (IsActionError == true), з нагороди віднімається штраф, в іншому випадку – додається бонус за коректну дію, обидва параметри також залежать від гіперпараметрів (errorFine, correctBonus).

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

За замовчування агент буде прагнути до виконання дій, які не тільки сприяють досягненню цілей епізоду, але й здійснюються з мінімальною кількістю помилок і високою ефективністю, щоб максимізувати загальну нагороду. Це стимулює агента до постійного вдосконалення стратегій поведінки та адаптації до змінних умов середовища.

### Нормалізація MinMax та Tanh

Нормалізація даних є важливим кроком у передобробці даних, особливо в машинному навчанні та аналізі даних, оскільки вона допомагає підвищити швидкість та стабільність навчання алгоритмів, а також зменшує ризик, що деякі характеристики (ознаки) будуть недооцінені через їх масштаб. Два популярні методи нормалізації — це MinMax нормалізація та нормалізація за допомогою гіперболічного тангенса (Tanh).

### 3.6.1 MinMax Нормалізація

MinMax нормалізація — це метод масштабування, який трансформує данні, зміщуючи мінімальне значення кожної характеристики до 0, а максимальне — до 1. Формула для MinMax нормалізації виглядає наступним чином:

min\_max\_norm = (X - X\_min) / (X\_max - X\_min)

Метод використовується для масштабування числових даних, коли важливо зберегти точні нульові значення та розподіл характеристик між 0 та 1. Це особливо корисно, коли алгоритми навчання чутливі до масштабу характеристик, наприклад, в нейронних мережах, алгоритмах, що використовують метрики відстані, тощо.

### 3.6.2 Tanh Нормалізація

Нормалізація за допомогою гіперболічного тангенса (Tanh) — це метод, який використовує гіперболічну функцію тангенса для масштабування значень характеристик так, що результати потрапляють в діапазон від -1 до 1. Формула має вигляд:

tanh\_norm = (exp(a\*X) - exp(-a\*X)) / (exp(a\*X) + exp(-a\*X))

Такий підхід використовується для масштабування даних, коли потрібно, щоб середнє значення було близьким до 0, що може сприяти швидшій конвергенції під час навчання нейронних мереж. Tanh нормалізація особливо корисна, коли дані мають приблизно симетричний розподіл навколо нуля.

### Stochastic Gradient Descent з Momentum

Stochastic Gradient Descent (SGD) з Momentum є популярним варіантом алгоритму стохастичного градієнтного спуску, який використовується для оптимізації в машинному навчанні та глибокому навчанні. Momentum – це гіперпараметр, який допомагає алгоритму швидше збігатися до мінімуму функції втрат, враховуючи попередні кроки для визначення напрямку наступного кроку, що дозволяє "перескакувати" локальні мінімуми.

Stochastic Gradient Descent (SGD) - це метод оптимізації, що використовується для мінімізації функції втрат шляхом ітеративного оновлення ваг моделі в напрямку найбільшого спаду функції втрат. Він робить це, використовуючи лише невелику вибірку даних (міні-батч), що робить алгоритм ефективнішим за пам'яттю та швидшим за класичний градієнтний спуск.

Momentum є модифікацією SGD, яка додає поняття "інерції" до процесу навчання, використовуючи експоненційно згладжений середній попередніх оновлень ваг. Це дозволяє алгоритму прискоритися в напрямках з постійним градієнтом і зменшити коливання в напрямках зі змінним градієнтом.

SGD з Momentum особливо корисний в задачах глибинного навчання, де простір параметрів великий і складний, а ландшафт функції втрат містить багато локальних мінімумів. Це робить його ідеальним для тренування складних нейронних мереж і здебільшого використовується, в задачах комп'ютерного зору та обробки природної мови.

З використанням цього оптимізатора агенти навчаються швидше, що особливо важливо у великих та складних моделях. За рахунок покращеня здатності агента ефективно оптимізувати свою політику поведінки в складних середовищах, збільшуючи його шанси на успіх. Метод може ефективно прискорювати навчання, особливо у фазах, де градієнт функції втрат дозволяє швидко рухатися до глобального мінімуму. Це робить його вигідним для використання в симуляторах еволюції, де швидкість збіжності є критичною.

Головною проблемою цього алгоритму є додатковий гіперпараметр Momentum, тому що потребує додаткової логіки динамічної зміни в залежності від стадії розвитку агенту, тому зазвичай використовуються більш універсальні та легкі в налаштуванні оптимізатори такі як Adam. Проте завдяки генетичному алгоритму, який за рахунок відбору відокремлює неактуальні значення від актуальних – ця проблема менше впливає на складність роботи моделі.

### Регулярізатори Dropout, L2, Weights Noise

Регулярізація є ключовим елементом у процесі тренування нейронних мереж, який допомагає запобігти перенавчанню та покращує загальну здатність моделі генералізувати на нових даних. Серед численних методів регулярізації особливо популярними є Dropout, L2 регулярізація та внесення шуму в ваги (Weights Noise). Кожен з цих методів має свої особливості та сфери застосування.

### 3.8.1 Dropout

Dropout — це техніка регулярізації, що полягає у випадковому відключенні (обнуленні) частини нейронів мережі під час тренування. Це запобігає ситуації, коли нейрони адаптуються до дуже специфічних особливостей тренувального набору даних, змушуючи мережу розподіляти вагу інформації між різними нейронами і тим самим покращуючи загальну здатність моделі до генералізації. Dropout зазвичай використовують у глибоких нейронних мережах і може бути особливо ефективним у великих архітектурах.

### 3.8.2 L2 Регулярізація (Ridge Regression)

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

### 3.8.3 Weights Noise (Шум в Вагах)

Внесення шуму в ваги (Weights Noise) — це метод, який полягає у додаванні випадкового шуму до ваг нейронної мережі під час тренування. Цей підхід 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, змушуючи модель стати більш стійкою до невеликих змін або шуму в даних.

### 3.8.4 Поєднання методів регулярізації

Одночасне використання різних методів регулярізації може бути дуже корисним для покращення загальної здатності моделі до генералізації та її стійкості до шуму в даних. Проте, це також вимагає ретельного налаштування гіперпараметрів та може збільшити обчислювальні витрати. Важливо знайти правильний баланс між різними методами регулярізації, щоб максимізувати ефективність тренування та уникнути потенційних пасток, таких як недостатнє навчання або занадто великі обчислювальні витрати. У розробленій мною моделі цю проблему вирішує генетичний алгоритм, тому реалізація такого поєднання регулярзаторів несе менше ризиків і ефективно виконує свою задачу. Варіативність регуляризаторів дозволяє запобігти перенавчанню агента і навчити його аналізувати схожий досвід, що особливо важливо в системах з постійним навчанням.

### Функція активізації Swish

Функція активізації Swish є відносно новою і була запропонована командою дослідників Google. Вона є гладкою, неперервною та не монотонною функцією, яка в останні роки здобула популярність у глибинному навчанні. Функція Swish визначається як: Swish(x) = x \* sigmoid(β \* x);

β є параметром, який може бути як константою, так і навчальним параметром. β може бути адаптовано під час процесу навчання, що потенційно покращує продуктивність моделі. β надає додаткову гнучкість, але також вимагає додаткового налаштування, що може ускладнити процес тренування.

Серед переваг можна відокремити такі характеристики, як: гладкість - Swish є гладкою функцією, що робить її 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, що може допомогти в певних архітектурах нейронних мереж захопити складніші залежності в даних.

### Ініціалізатор He

Ваги ініціалізуються випадковим чином з Гаусівського (нормального) розподілу з середнім 0 і дисперсією. Шляхом забезпечення кращого стартового розподілу ваг, He initialization може прискорити конвергенцію тренування, дозволяючи мережі швидше досягати більш оптимальних рішень. Початковий розподіл ваг, який є ні надто малим, ні надто великим, допомагає уникнути перенавчання на ранніх етапах тренування.

Ініціалізатор He широко використовується в сучасних глибоких нейронних мережах, особливо тих, що використовують ReLU або подібні до нього активаційні функції. Його ефективність у забезпеченні швидкого і стабільного навчання робить його популярним вибором серед дослідників і розробників у галузі машинного навчання.

### Epsilon-greedy стратегія дослідження

Epsilon-greedy (ε-greedy) стратегія є одним з найпростіших, але водночас ефективних підходів до балансування між дослідженням (exploration) та використанням (exploitation) у процесі навчання алгоритмів машинного навчання, особливо в задачах, пов'язаних з підсилювальним навчанням.

Epsilon-greedy стратегія працює, вибираючи між дослідженням нових дій та використанням вже відомих з певною випадковістю. Параметр ε (epsilon), який зазвичай є невеликим значенням між 0 і 1, визначає ймовірність, з якою алгоритм обере дослідження над використанням. Це означає, що з ймовірністю ε алгоритм випадково вибере будь-яку можливу дію, а з ймовірністю 1-ε він вибере найкращу дію на основі поточних знань. Ця стратегія дозволяє агенту знаходити 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 легко регулювати баланс між дослідженням нових дій та використанням вже відомих. Знаходження ідеального значення ε може бути непростим завданням, оскільки воно сильно залежить від конкретної задачі та її динаміки. Це питання вирішується динамічною зміною за допомогую еволюції генів і під кінець навчання ε, як правило, починає зменшуватись. Таким чином популяція агентів сама регулює зацікавленність в досліджені і не вмагає додаткової логіки.

### Remind Similar Expierence алгоритм

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

Кейс-базове Міркування (Case-based Reasoning) – це використання бази даних з минулими випадками або сценаріями для вирішення нових проблем шляхом знаходження і аналізу схожих випадків. Використання схожих досвідів може допомогти моделям швидше адаптуватися до нових ситуацій, використовуючи вже накопичені знання. Регулярний перегляд попереднього досвіду може допомогти утримувати важливу інформацію та зменшити ефект забування.

В моєму коді RSE використовується при обранні кажної дій. Нагадування спрацьовує тільки коли кількість прикладів перевищує задану користувачем з певною вірогідністю. Ця вірогідність є гіперпараметром і обирає випадкову згадку з тих що відсотково найсхожі. Відсоток з яких згадка обирається випадково теж є гіперпараметром.

Визначення ефективної метрики схожості може бути складним, оскільки вона повинна адекватно відображати 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 зроблено за допомогою різних метрик, але за замовчування це розраховується, як Евклідова відстань між векторами станів. Користувачі бібліотеки можуть реалізувати інтерфейс (IDDQNwithGACustomRemindExperiencesDefinder) і використовувати власний спосіб розрахунку схожего стану.

### 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 особливості та застосовуються в залежності від конкретних потреб задачі та обмежень.

### 3.13.1 Online Learning

Online learning відноситься до процесу навчання, де модель оновлюється послідовно в реальному часі або після кожного окремого спостереження/дії. Це означає, що система вчиться з кожним новим прикладом даних без потреби перенавчати модель з нуля. Цей підхід є корисним у сценаріях, де дані надходять послідовно або де необхідно швидко реагувати на зміни у вхідних даних.

Серед переваг Online Learning можна віднести здатність швидко адаптуватися до нових даних. Однак така модель дуже чутлива до шуму в даних та має великий ризик забуття старого досвіду, через постійне оновлення моделі.

### 3.13.2 Replay Learning (Experience Replay)

Experience replay є підходом, який заснований на зберіганні попередніх досвідів агента (наприклад, у вигляді пар стан-дія-винагорода) та їх використанні для навчання. Цей метод дозволяє ефективно використовувати обмежені досвіди шляхом повторного використання їх для тренувань, що покращує стабільність та ефективність навчання.

Серед переваг: зниження кореляції між послідовними тренувальними прикладами, що покращує стабільність навчання, можливість більш ефективного використання обмежених досвідів. Проте вимагає великий об'єм пам'яті.

### 3.13.3 Гібридний підхід

В моєму алгоритмі використовується гібридний підхід. Кількість даних для початку навчання зада'ється користувачем, а до того агент навчається за допомогою Online Learning. Після того, як необхідна кількість згадок накопичилась агент починає з певною періодичністю навчатися за допомогою replay learning на заданій користувачем кількості даних, в той самий час продовжуючи вчитися після кожного рішення.

Початкове використання online learning дозволяє моделі швидко адаптуватися до нових даних, що є особливо корисним на ранніх етапах навчання. Перехід до replay learning, коли є достатньо даних, забезпечує глибше навчання на основі ширшого спектру досвіду. Використання replay learning дозволяє багаторазово використовувати важливі досвіди, які можуть бути рідкісними або особливо важливими для навчання, забезпечуючи ефективніше використання наявних даних. Replay learning допомагає розбавити послідовність тренувальних даних, зменшуючи кореляцію між ними та покращуючи загальну стабільність та ефективність навчання. Цей підхід дозволяє гнучко налаштовувати баланс між швидкістю адаптації до нових даних та глибиною навчання на основі історичних даних, забезпечуючи масштабованість під різні умови навчання. Періодичний перехід від online до replay learning дозволяє збалансувати між потребою швидко реагувати на нові дані та необхідністю уникнути перенавчання на обмеженому або неоднорідному наборі даних.

Цей гібридний підхід використовує сильні сторони кожного методу, забезпечуючи більш ефективний процес навчання. Він особливо корисний в динамічних середовищах, де умови швидко змінюються, а також у складних задачах, де важливо ефективно використовувати кожен досвід для покращення продуктивності алгоритму.

### Висновок до розділу

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

Використання генетичного алгоритму для динамічного формування гіперпараметрів у подвійних глибоких Q-мережах (DDQN) може мати ряд переваг, які покращують загальну продуктивність і ефективність навчання. Ось деякі ключові плюси:

Автоматизація відбору гіперпараметрів: Генетичний алгоритм автоматизує процес пошуку оптимальних гіперпараметрів, зменшуючи необхідність в ручному налаштуванні і тестуванні, що може бути часомістким і потребує значних зусиль.

Покращення продуктивності: Здатність генетичного алгоритму ефективно досліджувати простір гіперпараметрів може призвести до знаходження більш ефективних конфігурацій, які покращують продуктивність DDQN мережі, особливо у складних середовищах.

Адаптація до змінних умов: Генетичні алгоритми можуть адаптувати гіперпараметри відповідно до змін у даних або середовищі, дозволяючи DDQN мережам залишатися ефективними навіть при змінних умовах.

Пошук глобального оптимуму: Генетичні алгоритми здатні до глобального пошуку в просторі гіперпараметрів, уникаючи локальних мінімумів, до яких можуть призвести інші методи оптимізації.

Збалансування дослідження і використання: Генетичний алгоритм може допомогти знайти баланс між дослідженням нових стратегій і використанням перевірених підходів, оптимізуючи політику дій DDQN для досягнення кращих результатів.

Паралельні обчислення: Генетичні алгоритми добре піддаються паралельним обчисленням, що дозволяє значно прискорити процес пошуку оптимальних гіперпараметрів за рахунок використання багатоядерних процесорів.

Гнучкість і універсальність: Генетичний алгоритм може бути застосований до широкого спектру задач і не залежить від специфіки проблеми, роблячи його універсальним інструментом для оптимізації гіперпараметрів у різних сценаріях використання DDQN.

Ці переваги роблять генетичний алгоритм потужним інструментом для оптимізації гіперпараметрів у DDQN, що може значно покращити їх продуктивність і ефективність у різноманітних задачах. А наслідування різних згадок(Memory) від різних батьків у контексті генетичних алгоритмів та нейронних мереж, сприяє різноманітності стратегій агентів. Це різноманіття може призвести до знаходження більш креативних і нестандартних рішень задач, підвищуючи шанси на виявлення оптимальних стратегій поведінки або конфігурацій, а також може допомогти моделі уникнути застрягання в локальних мінімумах, збільшуючи шанси на знаходження глобального максимуму або мінімуму.

## ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Ця робота складається з таких частин:

-Бібліотека зреалізацією моделі нейронної мережі;

-Симуляція (всесвіт в якому розвивається агент);

-Програма для тестування

### DDQNwithGA бібліотека

Для зручного використання моделі я зробив бібліотеку, яка містить в собі реалізацію алгоритму DDQNwithGA та допоміжні алгоритми. Бібліотека базується на парадигмі ООП і не використовує стороніх бібліотек з реалізацією методів для створеня нейроних мереж.

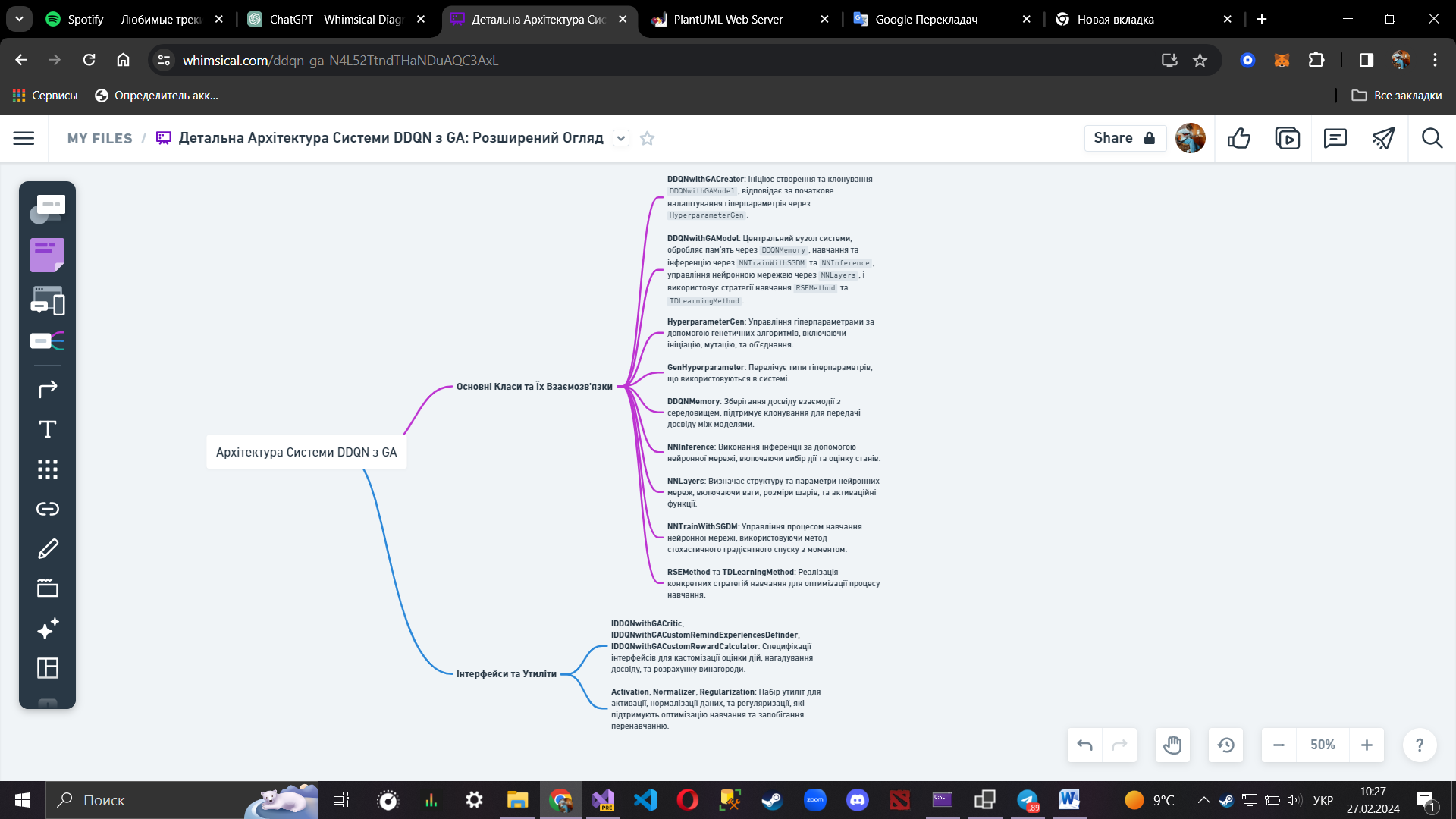


Рисунок 4.1 – Архітектура системи DDQNwithGA

### DDQNwithGAModel (class)

Цей клас DDQNwithGAModel 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.

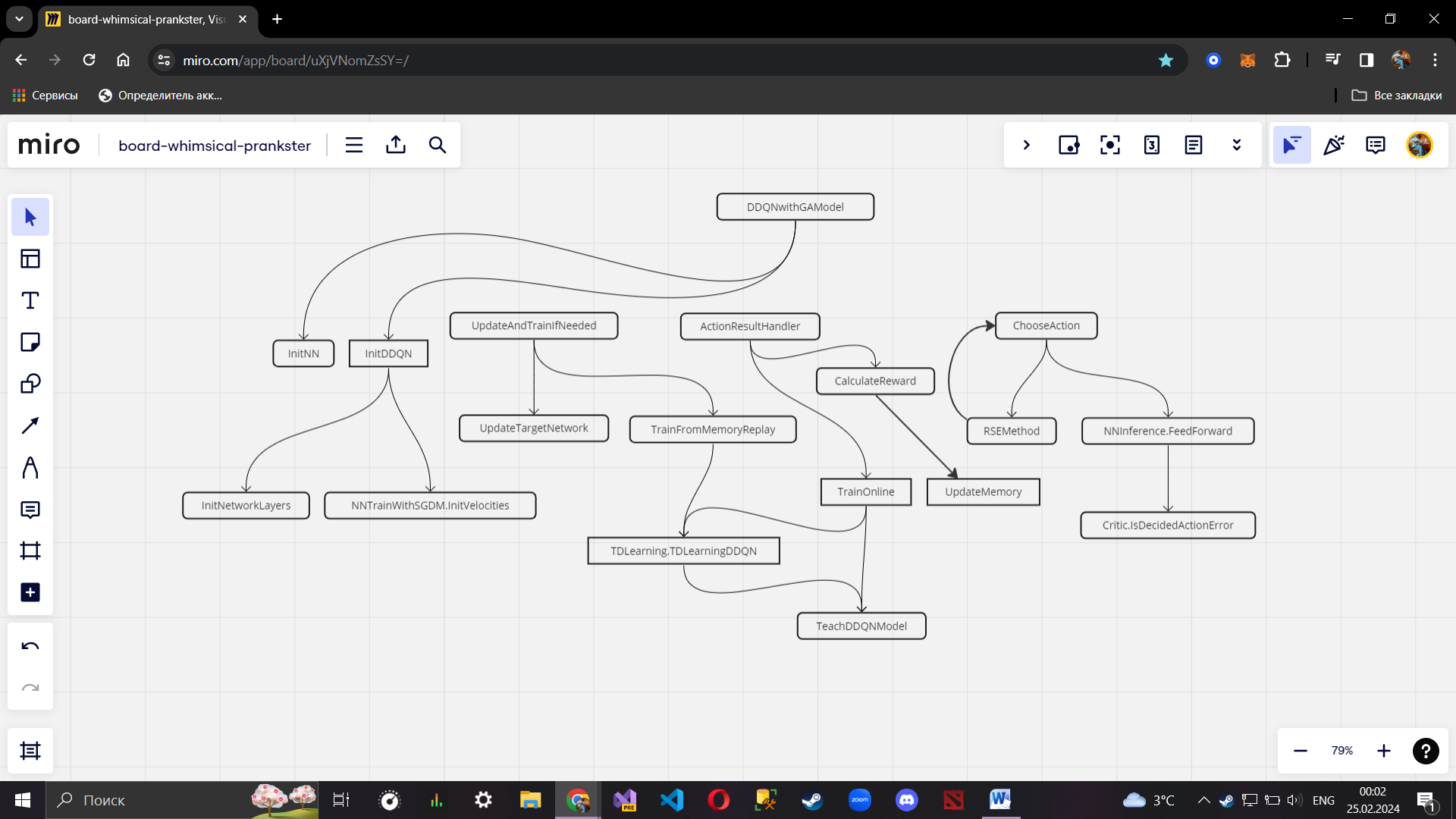


Рисунок 4.1.1 – Взаємозв’язок методів в класі DDQNwithGAModel

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

Опис кожного методу, поля та властивості та його призначення:

### 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.

### 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.

### 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.

### 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.

### Методи DDQNwithGAModel для оновлення мережі

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

### 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

.

### 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 з підкріпленням за допомогою різноманітних архітектур нейронних мереж і налаштувань гіперпараметрів. Повний код у додатку Б.

Ось 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 початкових розкидів гіперпараметрів (startHyperparameterScatterDict), а другий - одне значення розкиду 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.

### Ресурси нейроної мережі

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

.

### NNInference

Клас NNInference призначений для виконання прогнозування або інференсу в контексті нейронної мережі, що використовується в алгоритмі DDQN з генетичним алгоритмам 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. Це імітує стохастичність у процесі рішення та може допомогти підвищити загальну робастність моделі. 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.

### 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 та інференсу в рамках системи DDQN з GA.

Ось 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 Хе.

### 4.1.3.2.3 Методи

InitializeWeightsHe() - метод для ініціалізації ваг шару з використанням методу ініціалізації Хе, який є популярним вибором для глибоких нейронних мереж, оскільки він допомагає запобігти проблемам з градієнтами в глибоких мережах, нормалізуючи ваги за розміром шару. Ініціалізація Хе 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.

### 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, зменшуючи осциляцію. 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.

Ось детальний опис його компонентів:

### 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.

### Конструктор

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

### Методи

InitVelocities(NNLayers[] layers) - ініціалізує швидкості зміни ваг та зміщень 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.

### 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:

### 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.

### Конструктор

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

### 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.

### 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:

### 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.

Поля:

HyperparameterChromosome - словник, який зберігає поточні значення гіперпараметрів.

blockedHyperparameterGens - список заблокованих гіперпараметрів, які не будуть змінюватися під час мутацій.

random - екземпляр класу Random, використовуваний для генерації випадкових чисел.

### 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.

### 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.

### 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 осциляції.

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 перенавчанню.

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.

Ці гіперпараметри відіграють важливу роль у налаштуванні алгоритмів машинного навчання, дозволяючи досягати оптимальної продуктивності та адаптованості до різних задач і середовищ.

### Інтерфейси

Інтерфейси у реалізованій мною бібліотеці дозволяють користувачу бібліотеки з гнучко підлаштовувати модель під свої потреби і оптимізувати навчання під свої задачі.

### 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.

### Методи

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.

### 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 повторно використані під час навчання, на основі поточного стану середовища, а також для оцінки схожості між станами. Ось детальний опис його методів:

### Методи

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 повторного використання в процесі навчання. Це дозволяє агенту зосередитися на найбільш релевантних або схожих досвідах, покращуючи ефективність навчання.

CustomCalculateSimilarityState(double[] currentState, double[] previousState): Метод приймає поточний стан (double[] currentState) та попередній стан (double[] previousState), обчислюючи ступінь 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.

### 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. Ось детальний опис його методу:

### Методи

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.

### 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.

* + - 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 релевантних даних та сприяючи формуванню більш точних і ефективних стратегій поведінки.

Ось детальний опис його компонентів та методів:

* + - * 1. Поля

mostSimilarExperiences: Список для зберігання найбільш схожих досвідів, обраних для навчання.

* + - * 1. Методи

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

DefineMostSimilarExperiences(double[] currentState, double percentageOfSimilarExperiences, List<DDQNMemory> memory, IDDQNwithGACustomRemindExperiencesDefinder remindExperiencesDefinder = null): Визначає найбільш схожі досвіди на основі поточного стану. Може використовувати кастомізований визначник схожих досвідів, якщо такий надано, або стандартний підхід з Евклідовою відстанню.

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.

* + - 1. TDLearningMethod

Клас TDLearningMethod реалізує метод навчання з часовою різницею (Temporal Difference, TD) 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 досвіду, використовуючи різницю між прогнозованими та фактичними винагородами для оновлення своїх прогнозів.

Ось детальний опис методу TDLearningDDQN:

* + - * 1. Метод 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.

* + - * 1. Алгоритм

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. Метод TD навчання у поєднанні з DDQN підвищує стабільність та ефективність навчання, дозволяючи агенту краще адаптуватися до складних середовищ.

### 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.

Ось детальний опис методів:

* + - * 1. Методи

SwishActivation(double x, double beta): Реалізує активаційну функцію Swish, яка визначається як x \* sigmoid(beta \* x), де sigmoid(beta \* x) є сигмоїдною функцією, а beta - параметр, що регулює крутизну сигмоїди. Функція Swish вважається нелінійною та неперервною, здатною до самонормалізації в глибоких мережах.

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 sigmoid(beta \* x) + beta \* x \* sigmoid(beta \* x) \* (1 - sigmoid(beta \* x)), що дозволяє обчислити градієнт функції втрат відносно ваг нейронної мережі.

### Normalizer

Статичний клас Normalizer містить методи для нормалізації та денормалізації даних, що є важливими операціями в обробці даних, особливо при 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.

Ось детальний опис методів, які пропонує цей клас:

* + - * 1. Методи

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.

MinMaxDenormalize(double normalizedValue, double minValue, double maxValue): Здійснює обернену операцію до MinMaxNormalize, перетворюючи нормалізоване значення normalizedValue назад у його оригінальний діапазон, використовуючи ті ж minValue і maxValue.

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.

### 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

Ось детальний опис його методів:

* + - * 1. Методи

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.

ApplyDropout(double[] activations, int layerIndex, int[] layersSizes, double dropoutRate): Реалізує метод Dropout, який випадковим чином "вимикає" деякі нейрони в шарі 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.

GenerateRandomNoise(): Генерує випадковий шум, використовуючи нормальний розподіл (Гауссівський розподіл). Цей метод може бути використаний для додавання шуму до даних або ваг нейронної мережі, щоб збільшити її робастність або для експлорації різних стратегій навчання.

### Симуляція

Симуляція еволюції за допомогою моделі DDQNwithGA створена для тестування можливостей розробленого алгоритму. Проограма 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.

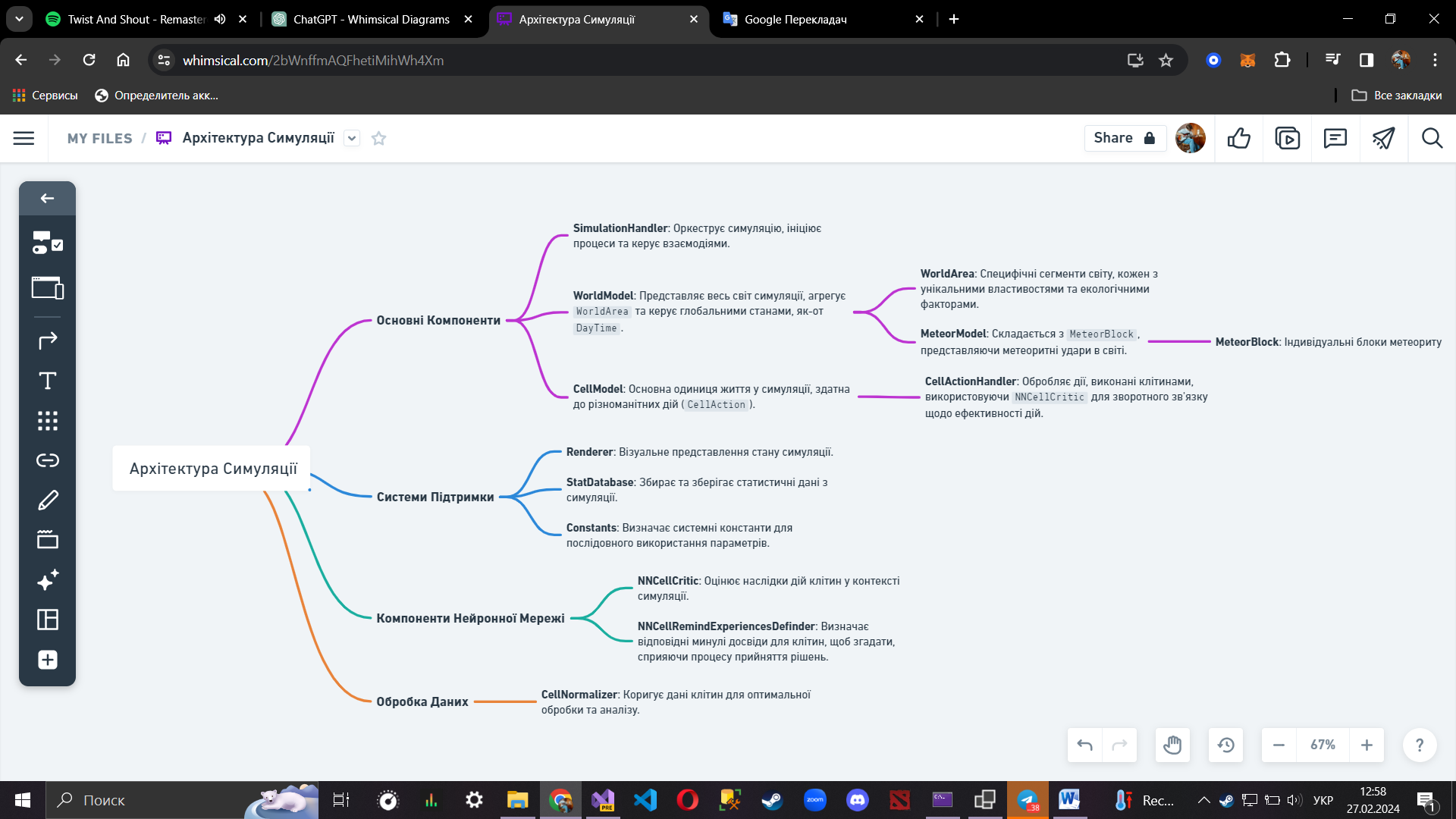


Рисунок 4.2 – Архітектура симуляції

### 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 різних стратегій виживання та адаптації.

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

* + - 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.

* + - 1. 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(Stopwatch 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.

### WorldResources

Об’єкти світу формують основу симуляційної системи, кожен з них відіграє важливу роль у моделюванні динамічного віртуального світу, де агенти взаємодіють з середовищем та одн з одним, а також адаптуються та еволюціонують відповідно до зовнішніх умов і власних досвідів.

WorldModel, WorldArea, та MeteorModel спільно створюють і управляють комплексним середовищем симуляції. Вони відповідають за моделювання часу доби, погодних умов, метеоритних подій та загального стану світу, в якому існують агенти. Це створює динамічний контекст, в якому клітини повинні виживати та адаптуватися.

CellModel представляє індивідуальні клітини, які і 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 багатошарову 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.

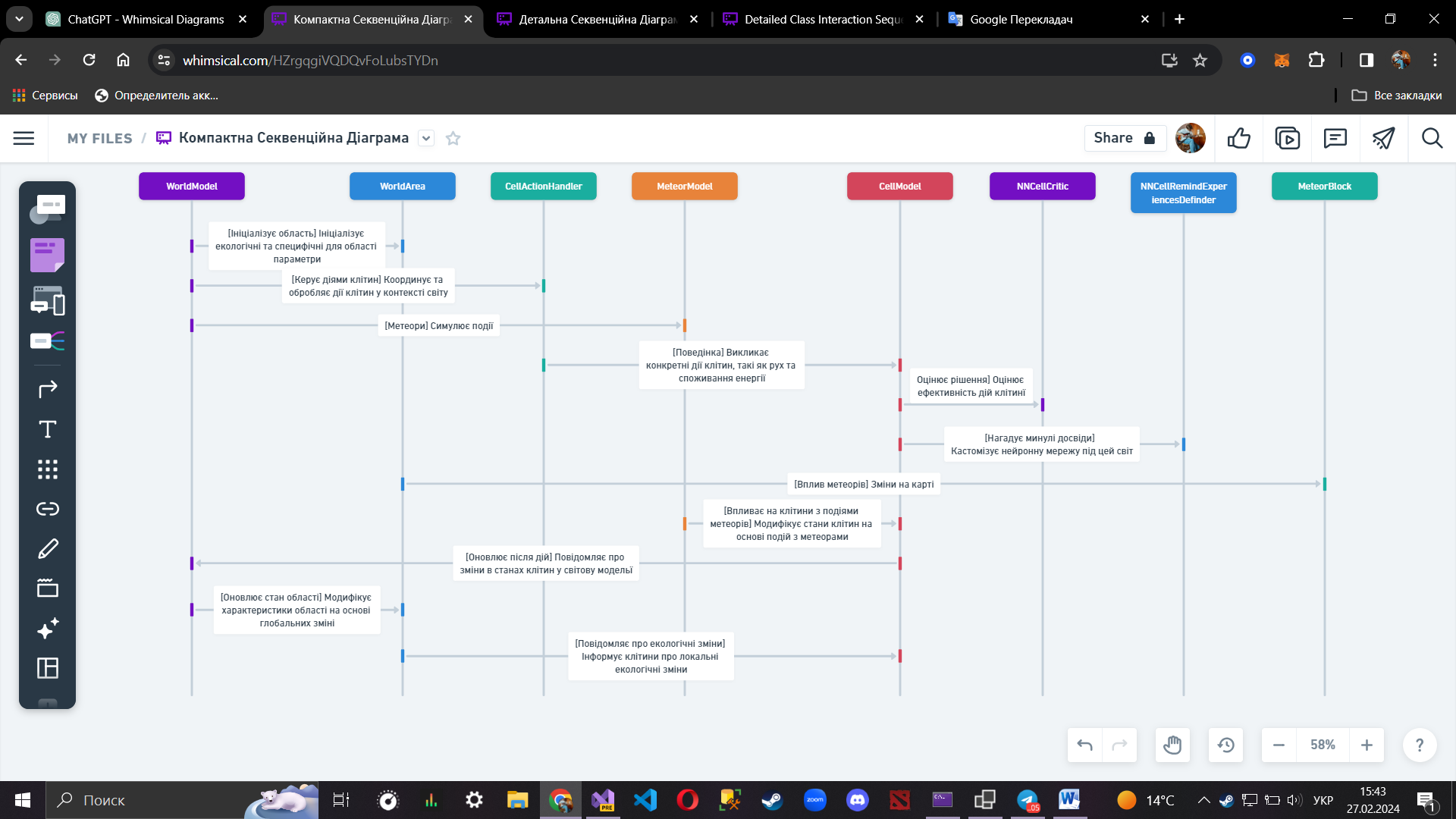


Рисунок 4.2.2 – Взаємодія об’єктів світу

### Cell

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

### CellModel

Клас CellModel є центральним компонентом у симуляційній системі, яка моделює поведінку та взаємодії клітин 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.

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.

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 "стрібки" 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.

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.

### 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:

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.

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. Включає 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.

### 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.

NNCellCritic

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

NNCellRemindExperiencesDefinder

NNCellRemindExperiencesDefinder відповідає за вибір найбільш релевантних досвідів із пам'яті для підтримки процесу навчання клітини. Цей клас реалізує інтерфейс IDDQNwithGACustomRemindExperiencesDefinder, 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".

### 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.

* + - * 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.

* + - * 1. MeteorModel

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

Поля і методи

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.

### 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.

* + - * 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.

* + - * 1. 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.

* + - * 1. 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.

* + - * 1. Візуалізація

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

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

### 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.

* + - * 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.

* + - * 1. 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.

* + - * 1. 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.

* + - * 1. 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.

* + - * 1. 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.

### 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.

### 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.

### Renderer

Клас Renderer є важливою частиною системи для забезпечення інтерактивності та візуальної привабливості симуляції. Він допомагає користувачам краще зрозуміти динаміку процесів, що відбуваються в симульованому світі, та надає засоби для нагляду 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.

### Програма для тестування

Програма, що складається з двох класів StatModelError та ErrorStat, призначена для збору та візуалізації статистичних даних про помилки в процесі симуляції "Еволюції клітин". Це дозволяє аналізувати ефективність та точність поведінки 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. Це може допомогти розробникам та дослідникам ідентифікувати потенційні проблеми або вдосконалити параметри симуляції для досягнення більш стабільних та реалістичних результатів.

Використовуючи LiveCharts, цей клас динамічно відображає зміни в даних про помилки, що дозволяє користувачам аналізувати тенденції та шаблони у поведінці симуляції. Дані представлені у вигляді лінійного графіка, де по осі X відкладено 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.

4.3.1.1 Поля та методи

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 тику 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 можуть включати день симуляції та величину помилки, асоційовану з цим днем, дозволяючи 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.

4.3.2.1 Поля та методи

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.

## ВСЕСВІТ ТА ТЕСТУВАННЯ

### Всесвіт

Екосистема симуляції є складним і взаємопов'язаним середовищем, де клітини взаємодіють одна з одною та з різними елементами навколишнього середовища, зокрема отруєні області та метеорити. Структура світу та його елементи створюють багатогранну екосистему, де взаємодія між клітинами та навколишнім середовищем лежить в основі еволюційного розвитку та адаптації видів. Кожен компонент екосистеми, від найдрібніших деталей ландшафту до глобальних циклів зміни часу доби, відіграє роль у формуванні умов життя та еволюції клітин. Таким чином, світ симуляції є складною і взаємопов'язаною системою, де постійно відбуваються процеси відбору, мутації, конкуренції та співробітництва. 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 осередок може грати роль певної частини навколишнього середовища: бути вільною зоною, отруєною зоною або місцем падіння метеориту. Також осередки можуть бути зайняті клітинами - агентами симуляції, які взаємодіють між собою та з навколишнім середовищем.

### 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;

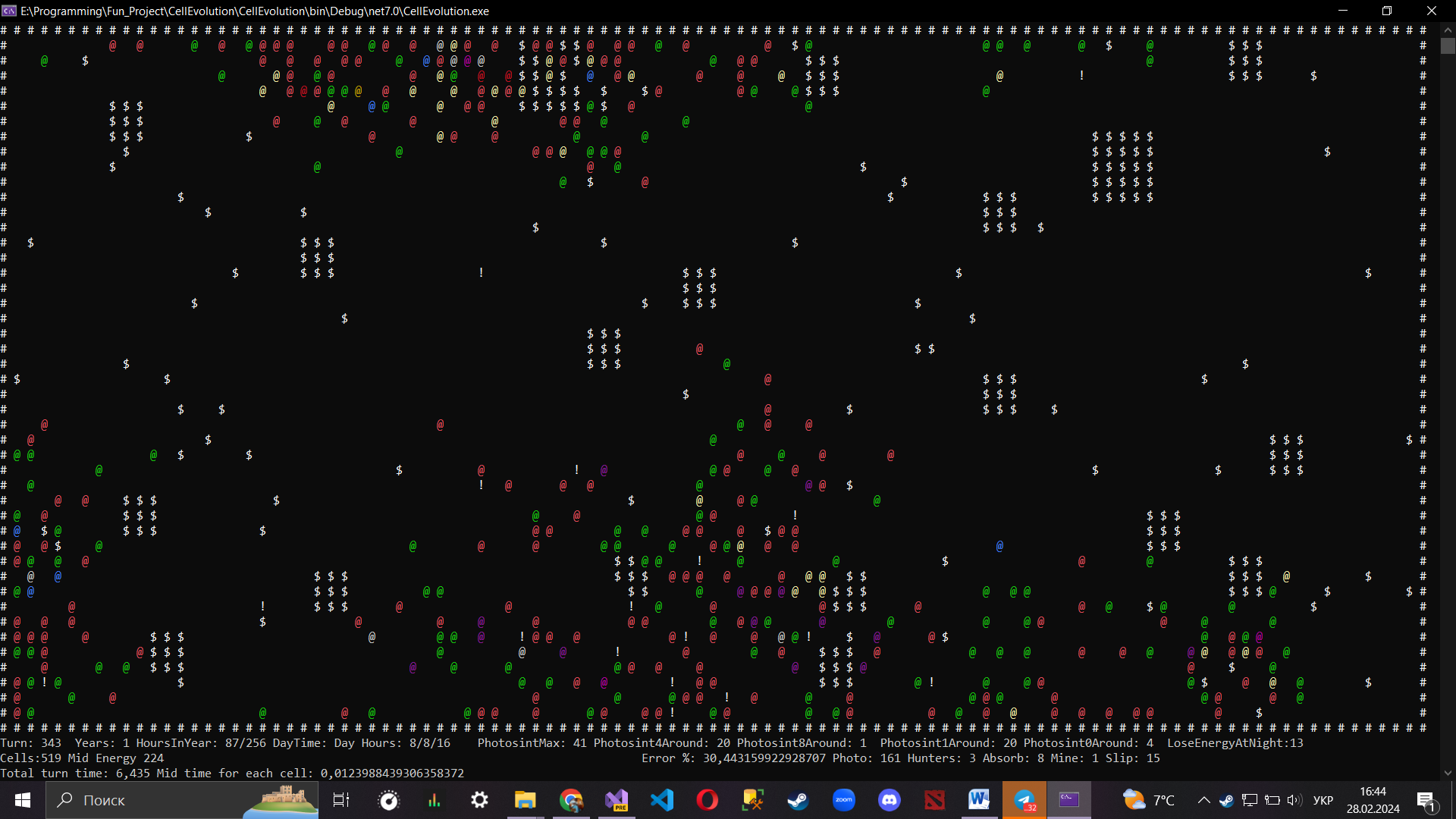


Рисунок 5.1.1 – Зразок візуалізації симуляції

На зразку[5.1.1] колір клітини залежить від останьої дії. В момент збереження в симуляції пройшло тільки 319 діб (початок навчання). 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.

### Клітини

Клітини у світі симуляції є складними агентами, оснащеними різноманітними функціями та здібностями, які дозволяють їм взаємодіяти 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.

1. Переміщення

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

2. Стрибки

Стрибки являють собою дію, що дозволяє клітині перескакувати через одну або кілька клітин простору, що може бути корисним для швидкого переміщення або уникнення перешкод і хижаків.

3. Фотосинтез

Фотосинтез є критично важливою дією для клітин для здобуття енергії. Під час дня вони можуть використовувати сонячне світло для виробництва енергії, що є основним джерелом живлення 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 середовща 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 вигаданний 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.

4. 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».

5. 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.

6. Видобуток

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

7. Полювання

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

8. Захист

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

9. 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 формирується, 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 найчастішй вибір, тому навіть, наприклад, зелена клітина, яка зазвичай використовує фотосинтез може полювати або поглинати енергію померлих клітин.

### Час доби

Зміна часу доби у світі симуляції значно впливає на поведінку і стратегії виживання клітин, особливо щодо процесу фотосинтезу. Час доби в симуляції змінюється між днем і ніччю, що безпосередньо впливає на можливість клітин проводити фотосинтез.

### 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, деякі метеорити навіть викликають ніч на декілька ходів.

Генерація метеоритів відбувається випадковим чином. Параметри створення, включаючи розмір та розподіл метеоритних блоків, визначаються 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 днамічності та випадковості середовища.

### Отруєні області

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

Такі області 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.

### Приклад симуляції

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

На першому зображенні[5.1.6.1] 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.

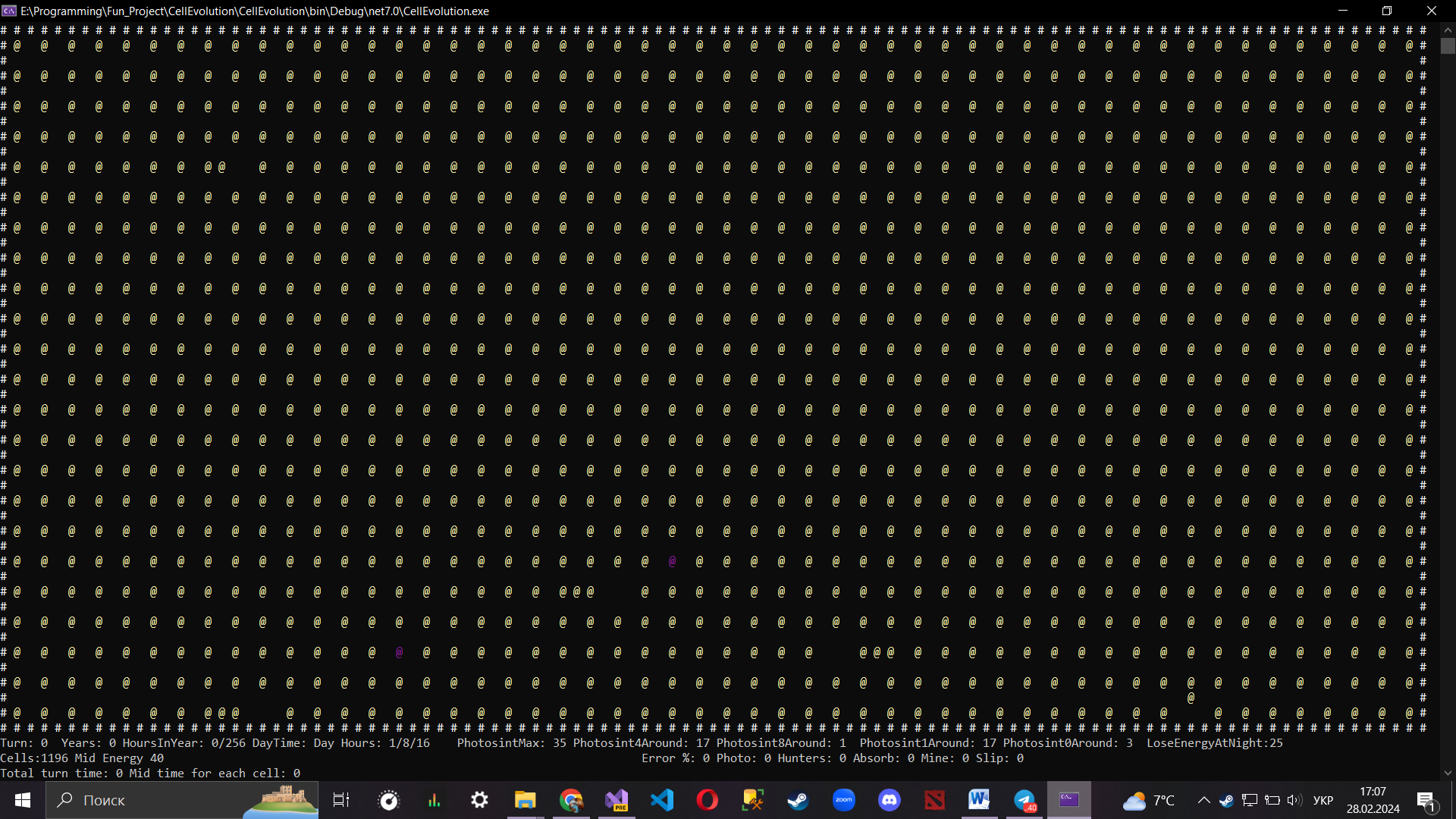


Рисунок 5.1.6.1 – Візуалізація старту симуляції

На наступному зображенні[5.1.6.2] 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 5.1.6.2 – Візуалізація перших етапів відбору

На зображенні[5.1.6.3] 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 5.1.6.3 – Візуалізіція найуспішніше створених клітин

На зображенні [5.1.6.4] можна спостерігати формування колонії найуспішніших клітин, які пристосувались до екологічних умов симуляції.

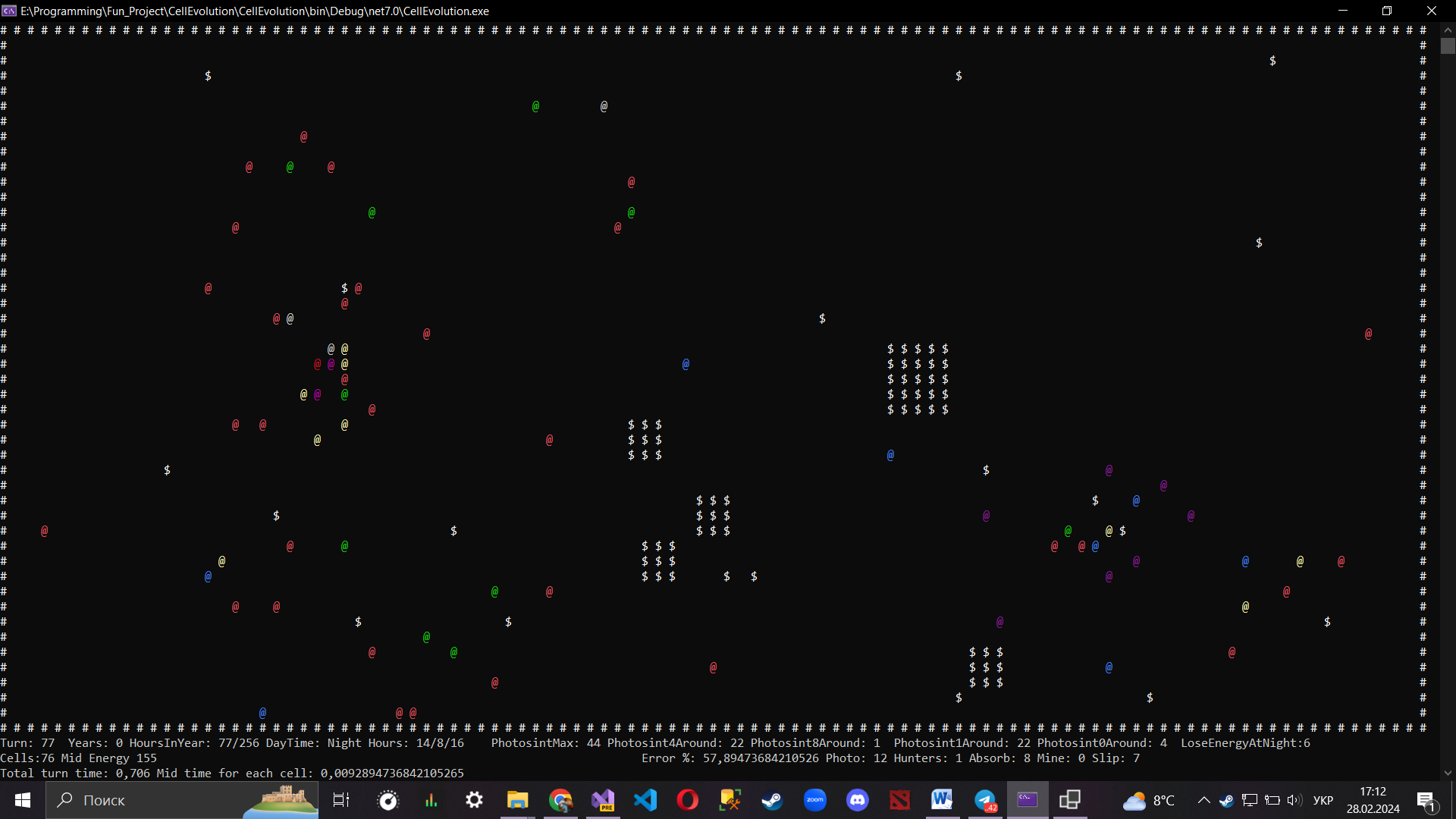


Рисунок 5.1.6.4 – Візуалізація початку формування колоній

Сформовані колонії, агенти навчились виживати в середовищі і тепер намагаються максимізувати нагороди.



Рисунок 5.1.6.4 – Візуалізація колоній

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



Рисунок 5.1.6.4 – Візуалізація формування агентами структур для максимізації фотосинтезу

### Тестування

Тестування проходило за допомогою бази MS SQL та створенною мної програми 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 енергій, 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 кожного дня вносились в базу і потім вводився графік з процесом та результатами відносно визначених мною помилковими діями. Проте ці оцінки не показують 100% пристосованності 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:

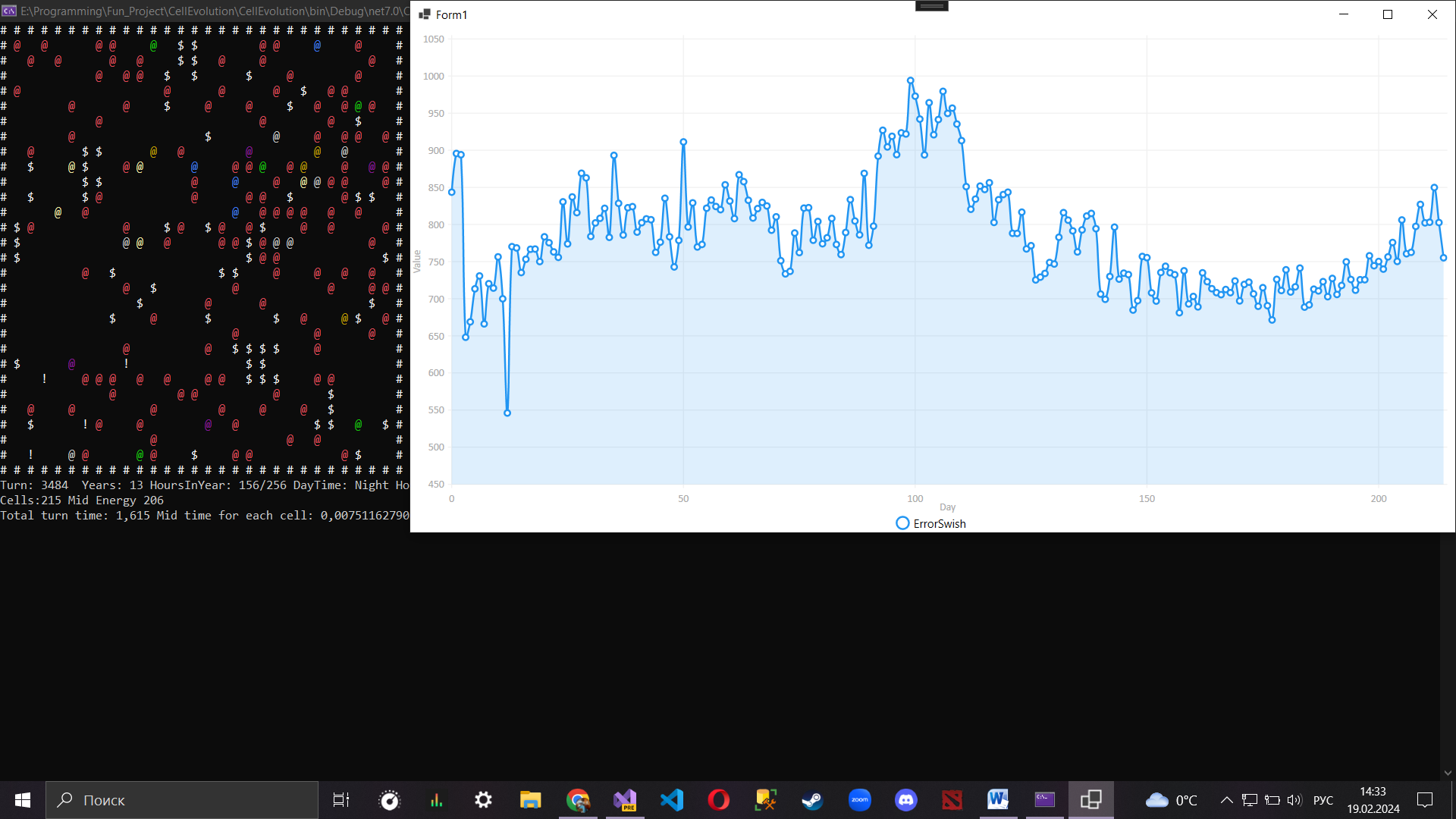


Рисунок 5.2.1.1 – Графік процессу начання за допомогою генетичного алгоритму

На представленому графіку[5.2.1.1] відображено процес навчання з використанням генетичного алгоритму, де вісь Y показує "Value", яке означає кількість помилок, а вісь X відображає прогрес у часі за добами. Загальна тенденція показує, що спочатку кількість помилок значно коливається, потім спостерігається тенденція до зниження, але далі йде період стагнації та незначного зростання.

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

Приклад нейроеволюції:

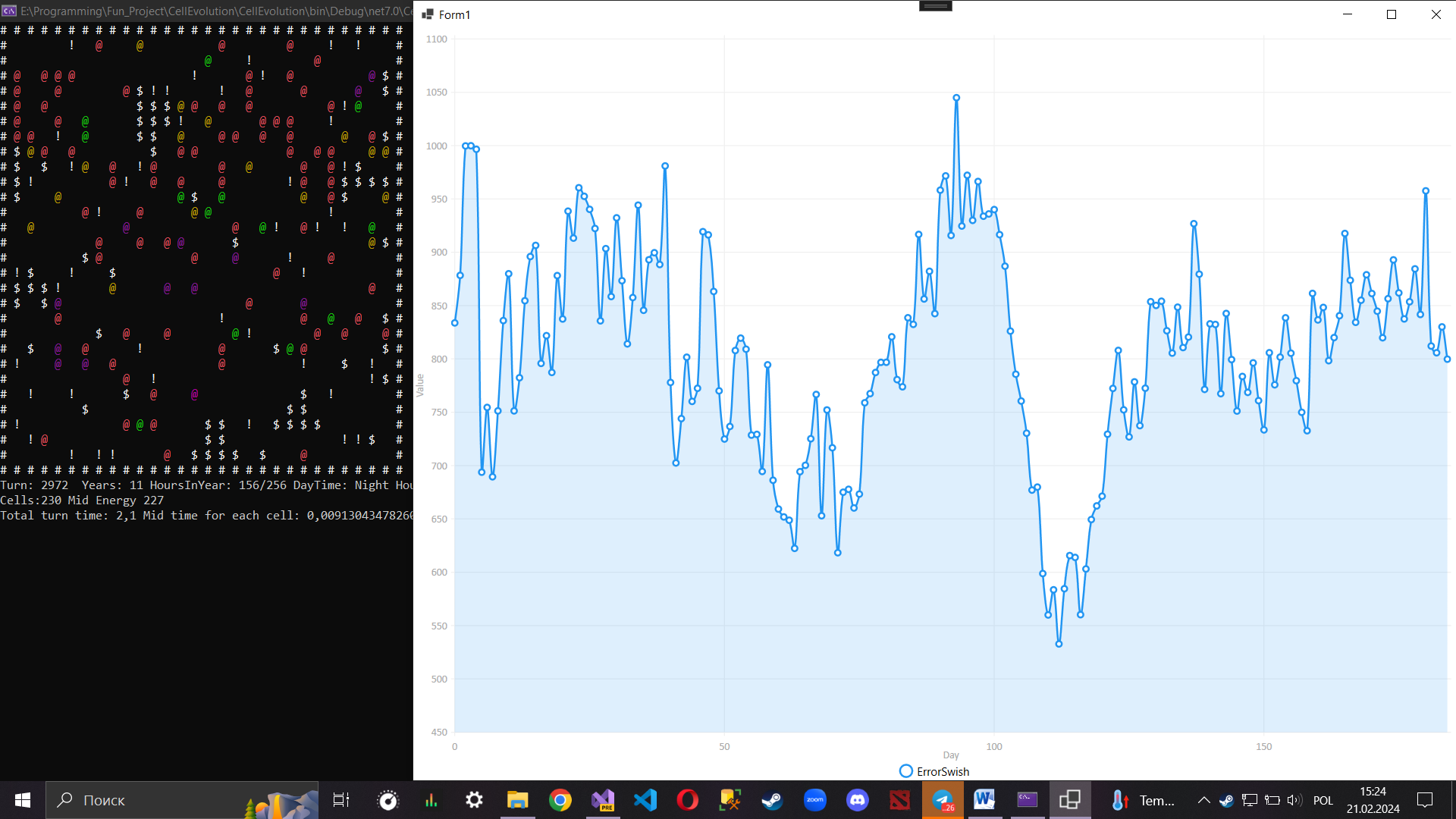


Рисунок 5.2.1.2 – Графік процессу начання за допомогою нейроеволюції

На цьому графіку[5.2.1.2] 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.

Судячи з графіку, можна помітити великі коливання, що може бути ознакою високої ступені експлорації. Глибокий спад, який ми бачимо близько 100 дня, 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 підкріпленям:



Рисунок 5.2.1.3 – Графік процессу начання за допомогою навчання кожного агента з використанням нейронної мережі з підкріпленям

З графіка[5.2.1.3] 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 стати перенавченою на певних вибірках або сценаріях, втрачаючи свою здатність узагальнення на нових даних. Якщо гіперпараметри є статичними, вони не можуть адаптуватися до змін в середовищі або до вимог завдання, що може перешкоджати дальшому навчанню або покращенню стратегії.

Приклад DDQNwithGA моделі:

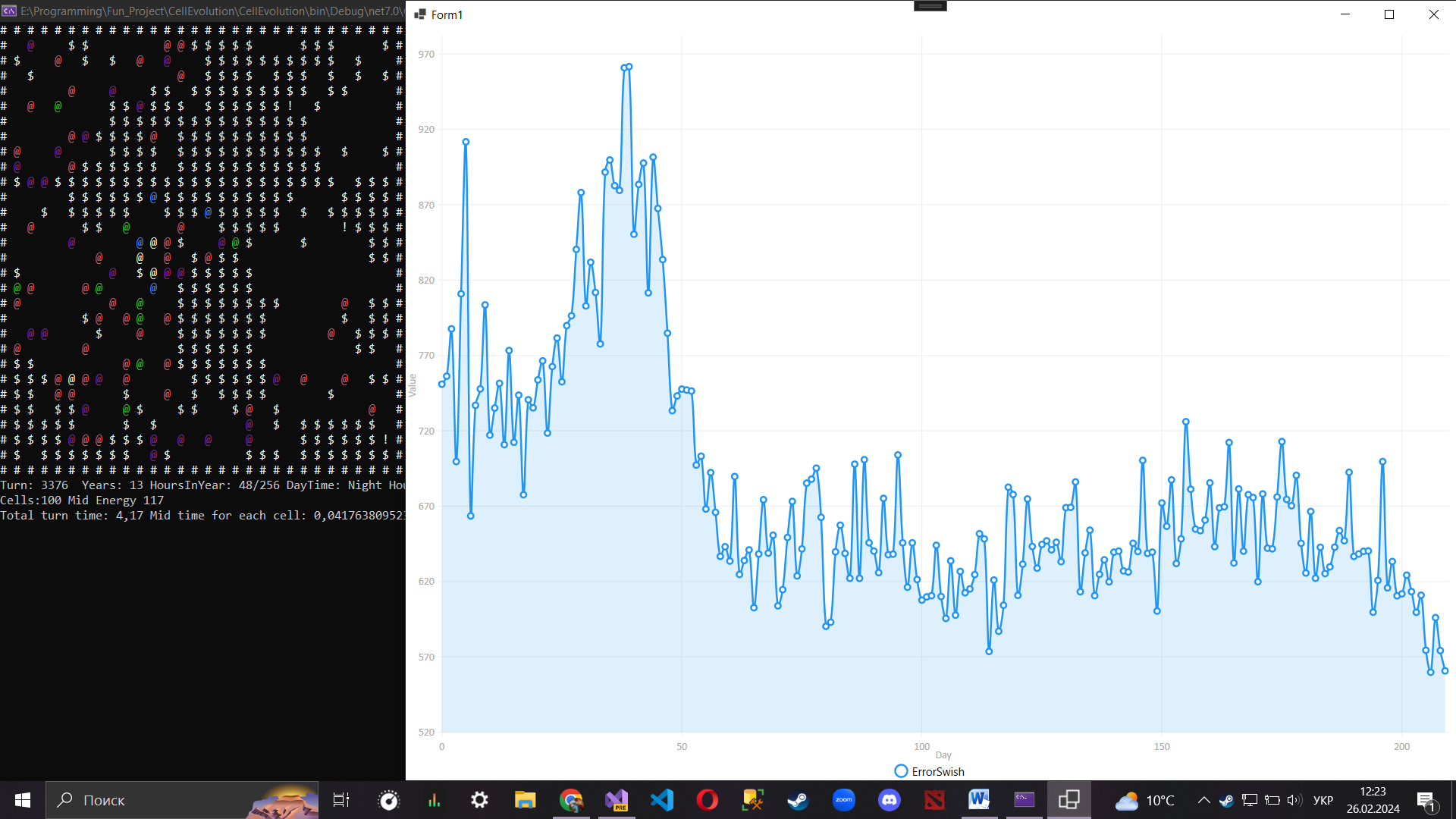


Рисунок 5.2.1.4 – Графік процесу навчання за допомогою DDQNwithGA моделі

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.

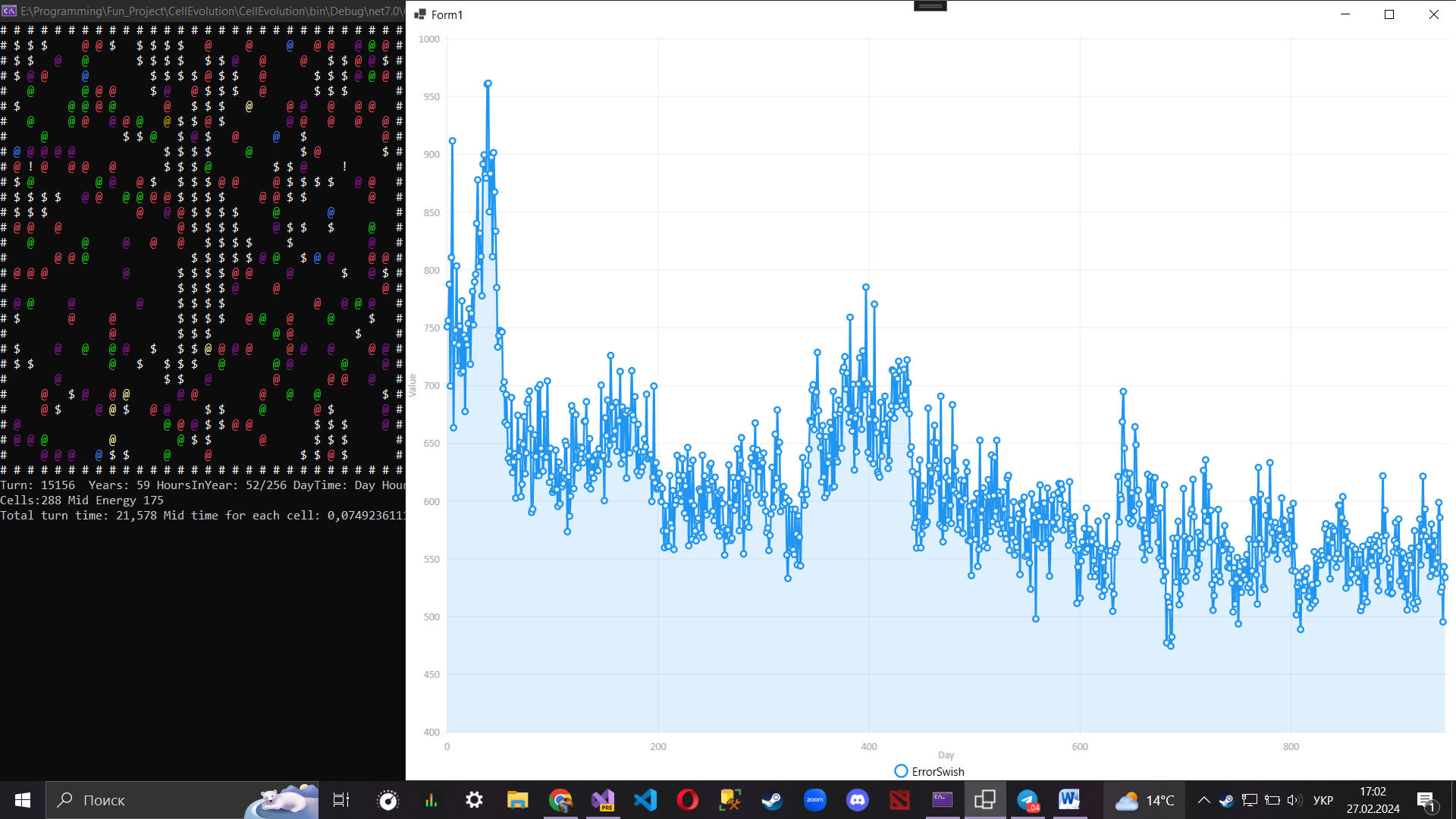


Рисунок 5.2.1.5 – Графік довшего процессу начання за допомогою DDQNwithGA моделі

На другому графіку[5.2.1.5], 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.

Зменшення волатильності помилок у довгостроковій перспективі може також вказувати на те, що DDQNwithGA досягла більш оптимальної політики, але досі продовжує адаптуватися та оптимізувати 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.

5.2.2 Висновок до розділу

Генетичні алгоритми та нейроеволюція демонструють потенціал для вирішення оптимізаційних задач шляхом експлорації різноманітних рішень та їх адаптації до складних умов середовища. Втім, 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 свідчить 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 здатність до підтримки стабільної продуктивності на довгій дистанції, зменшуючи волатильність помилок та 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.

## РОБОТА КОРИСТУВАЧА З БІБЛІОТЕКОЮ МОДЕЛІ

* + 1. Вимоги до користувача

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

1. Основні знання в області програмування:

Володіння мовою C#: Бібліотека 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.

2. 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.

3. Знайомство з генетичними алгоритмами:

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

4. Навички роботи з бібліотекою:

Інтеграція з існуючими проєктами: Користувач повинен знати, як інтегрувати зовнішні бібліотеки в свої проєкти.

5. 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.

* + 1. 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.

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);

2. Клонування моделі DDQNwithGA

Ви можете клонувати існуючу модель DDQNwithGA безпосередньо або створити нову модель шляхом злиття двох батьківьских 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);

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);

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());

Висновок

Цей посібник окреслює 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

Система демонструє здатність ефективно взаємодіяти з динамічним середовищем, адаптувати 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. Результати вказують на зменшення волатильності помилок у довгостроковій перспективі, що свідчить про досягнення моделлю більш оптимальної політики.

Робота відкриває нові перспективи для розробки більш ефективних і адаптивних 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, що є важливим кроком в напрямку створення більш інтелектуальних та самонавчальних систем.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Adarsh Sehgal, Hung Manh La, Sushil J. Louis, Hai Nguyen. - Deep Reinforcement Learning using Genetic Algorithm for Parameter Optimization <https://ar5iv.labs.arxiv.org/html/1905.04100>
2. Hung La, Adarsh Sehgal, Nicholas Ward, Sushil Louis - Automatic Hyperparameter Optimization Using Genetic Algorithm in Deep Reinforcement Learning for Robotic Manipulation Tasks

<https://ar5iv.labs.arxiv.org/html/2204.03656>

1. Hadi S. Jomaa, Josif Grabocka, Lars Schmidt-Thieme - Hyperparameter Optimization by Reinforcement Learning

<https://ar5iv.labs.arxiv.org/html/1906.11527>

1. Daniel Florek, Marek Miłosz - Comparison of an effectiveness of artificial neural networks for various activation functions

<https://ph.pollub.pl/index.php/jcsi/article/view/3069>

1. Gerald Tesauro - Practical Issues in Temporal Difference Learning

<https://link.springer.com/article/10.1007/BF00992697>

1. Licheng Jiao, Ronghua Shang, Fang Liu, Weitong Zhang - The models and structure of neural networks

<https://www.sciencedirect.com/science/article/abs/pii/B9780128197950000025?via%3Dihub>

1. Hyung-Suk Yoon, Sang-Hyun Lee, Seung-Woo Seo - Exploration Strategy based on Validity of Actions in Deep Reinforcement Learning

<https://ieeexplore.ieee.org/document/9341014>

1. Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, Ling Shao - Normalization Techniques in Training DNNs: Methodology, Analysis and Application

<https://ieeexplore.ieee.org/document/10056354>

1. Akshay Badola, Vineet Padmanabhan Nair, Rajendra Prasad Lal - An Analysis of Regularization Methods in Deep Neural Networks

<https://ieeexplore.ieee.org/document/9342192>

1. Bao Wang, Tan Nguyen, Tao Sun, Andrea L. Bertozzi, Richard G. Baraniuk, Stanley J. Osher - Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent

<https://epubs.siam.org/doi/10.1137/21M1453311>

1. Risto Miikkulainen, Eliana Feasley, Leif Johnson, Igor Karpov, Padmini Rajagopalan, Aditya Rawal, Wesley Tansey - Multiagent Learning through Neuroevolution

<https://link.springer.com/chapter/10.1007/978-3-642-30687-7_2>

1. Kenneth De Jong - Learning with genetic algorithms

<https://link.springer.com/article/10.1007/BF00113894>

1. R.S. Sutton; A.G. Barto - Reinforcement Learning

<https://ieeexplore.ieee.org/document/712192>

1. Susumu Adachi, Jia Lee - Variations on the Game of Life

<https://link.springer.com/chapter/10.1007/978-1-84996-217-9_13>

1. Yongcheng Jing, Yezhou Yang, Zunlei Feng, Jingwen Ye, Yizhou Yu, Mingli Song - Neural Style Transfer

<https://ieeexplore.ieee.org/document/8732370>

1. H Francis SongGuangyu R YangXiao-Jing Wang - Reward-based training of recurrent neural networks for cognitive and value-based tasks

<https://elifesciences.org/articles/21492>

1. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun - Delving Deep into Rectifiers

<https://ieeexplore.ieee.org/document/7410480>

## Додаток А

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 batchMemorySize, uint minMemoryToStartMemoryReplayLearning = 0)

{

this.Gen = gen;

this.LayersSizes = layersSizes;

InitNN();

InitDDQN(layersSizes, maxMemoryCapacity, batchMemorySize, 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.discountFactor],

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.discountFactor],

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 batchMemorySize, uint minMemoryToStartMemoryReplayLearning)

{

LayersSizes = new int[layerSizes.Length];

Array.Copy(layerSizes, LayersSizes, layerSizes.Length);

InitNetworkLayers();

NNTrainWithSGDM.InitVelocities(OnlineLayers);

this.MaxMemoryCapacity = maxMemoryCapacity;

this.BatchMemorySize = batchMemorySize;

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));

}

}

}

## Додаток Б

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 batchMemorySize, uint minMemoryToStartMemoryReplayLearning = 0)

{

ValidateLayerSizes(layerSizes);

HyperparameterGen Gen = new HyperparameterGen();

Gen.StartHyperparameterScatter(startHyperparameterScatterDict);

DDQNwithGAModel dDQNwithGA = new DDQNwithGAModel(Gen, layerSizes, maxMemoryCapacity, batchMemorySize, minMemoryToStartMemoryReplayLearning);

return dDQNwithGA;

}

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 dDQNwithGA = new DDQNwithGAModel(Gen, layerSizes, maxMemoryCapacity, batchMemorySize, minMemoryToStartMemoryReplayLearning);

return dDQNwithGA;

}

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 batchMemorySize = original.BatchMemorySize;

uint maxMemoryCapacity = original.MaxMemoryCapacity;

uint minMemoryToStartMemoryReplayLearning = original.MinMemoryToStartMemoryReplayLearning;

DDQNwithGAModel dDQNwithGACopy = new DDQNwithGAModel(Gen, layerSizes, maxMemoryCapacity, batchMemorySize, 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 batchMemorySize = baseParent.BatchMemorySize;

uint maxMemoryCapacity = baseParent.MaxMemoryCapacity;

uint minMemoryToStartMemoryReplayLearning = baseParent.MinMemoryToStartMemoryReplayLearning;

DDQNwithGAModel dDQNwithGACopy = new DDQNwithGAModel(Gen, layerSizes, maxMemoryCapacity, batchMemorySize, 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;

}

}

}

}