Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

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

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
магістр
(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДУ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ІГРОВОГО ПРОЦЕСУ НА ОСНОВІ ІНТЕЛЕКТУАЛЬНИХ АГЕНТІВ З REINFORCEMENT LEARNING
RESEARCH AND SOFTWARE IMPLEMENTATION OF AN AUTOMATED GAME TESTING METHOD BASED ON INTELLIGENT AGENT WITH REINFORCEMENT LEARNING

Виконав(ла): студент(ка) 2 курсу, групи КНТ-214м
Спеціальності 122 Комп'ютерні науки
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Системи штучного інтелекту

СОКОЛЕНКО М.О.
(ПРІЗВИЩЕ та ініціали)
Керівник ПОДКОВАЛІХІНА О.О.
(ПРІЗВИЩЕ та ініціали)
Рецензент ЗЕЛІК О.В.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 122 Комп'ютерні науки
(код і найменування)
Освітня програма (спеціалізація) Системи штучного інтелекту
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
_____ Сергій СУББОТІН
"____" _____2025 року

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

СОКОЛЕНКА Микити Олеговича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та програмна реалізація методу автоматизованого тестування ігрового процесу на основі інтелектуальних агентів з Reinforcement Learning. Research and Software Implementation of an Automated Game Testing Method Based on Intelligent Agents with Reinforcement Learning
керівник проєкту (роботи) к.т.н., доцент, ПОДКОВАЛІХІНА Олена Олександрівна ,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від " 30 " вересня 2025 року № 447
2. Строк подання студентом проєкту (роботи) 19 грудня 2025 року
3. Вихідні дані до проєкту (роботи) рекомендована література, лістинг коду програми

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1.Аналіз проблеми та постановка завдань дослідження. 2.Матеріали і методи. 3. Розробка архітектури системи. 4. Програмна реалізація системи. 5. Експериментальне дослідження.

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

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	ПОДКОВАЛІХІНА О.О., доцент		
Нормоконтролер	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання “ 30 ” вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2-3 тижні	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів	4-5 тижні	Розділ 2
	вирішення задачі.		
4	Розробка архітектури програми.	6 тиждень	Розділ 3
5	Розробка програми.	7-8 тижні	Розділ 4
6	Тестування та експериментальне дослідження	9 тиждень	Розділ 5
	програмного забезпечення.		
7	Оформлення пояснювальної записки та документів до неї.	10-11 тижні	Додатки
8	Нормоконтроль та рецензування.	12 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка) _____ Микита СОКОЛЕНКО
(підпис) (Імя ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Олена ПОДКОВАЛІХІНА
(підпис) (Імя ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра: 87 с., 6 табл., 4 рис., 6 дод., 29 джерел.

PPO, REINFORCEMENT LEARNING, VIZDOOM, АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, ГЛИБОКЕ НАВЧАННЯ, ІНТЕЛЕКТУАЛЬНИЙ АГЕНТ.

Об’єкт дослідження - процес обчислень для автоматизованого тестування ігрового процесу у відеоіграх.

Предмет дослідження - методи та комп'ютерні алгоритми навчання з підкріпленням для створення інтелектуальних агентів.

Мета роботи - підвищення показника покриття ігрового простору до рівня понад 80% та скорочення часу тестування ігрових рівнів за рахунок застосування інтелектуальних агентів на основі алгоритму PPO.

Матеріали, методи та технічні засоби: глибоке навчання з підкріпленням, алгоритм PPO, згорткові нейронні мережі, мова програмування Python, бібліотеки PyTorch та Stable Baselines3, платформа VizDoom, операційна система Linux.

Результати. Розроблено інтелектуального агента на основі PPO з модифікованою системою винагород, який демонструє 84,7% покриття ігрового простору та здатний виявляти типові дефекти ігрового процесу.

Висновки. Таким чином, мету роботи досягнуто: забезпечено підвищення покриття ігрового простору з 23,4% (випадковий агент) до 84,7% (PPO-агент з exploration bonus), що становить збільшення у 3,6 рази. Розроблений агент також перевершує базовий DQN-метод (61,8%) на 37% абсолютних.

Галузь використання - розробка та тестування відеоігор, наукові дослідження в галузі штучного інтелекту та машинного навчання.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 87 pages, 6 tables, 4 figures, 6 appendixes, 29 sources.

PPO, REINFORCEMENT LEARNING, VIZDOOM, DEEP LEARNING, AUTOMATED TESTING, INTELLIGENT AGENT.

Object of study is the computational process for automated game testing in video games.

Subject of study are methods and computer algorithms of reinforcement learning for creating intelligent agents.

The purpose of the work is increasing the game space coverage rate to above 80% and reducing game level testing time by applying intelligent agents based on the PPO algorithm.

Materials, methods, and tools: deep reinforcement learning, PPO algorithm, convolutional neural networks, Python programming language, PyTorch and Stable Baselines3 libraries, VizDoom platform, Linux operating system.

Results. An intelligent agent based on PPO with a modified reward system was developed, demonstrating 84.7% game space coverage and the ability to detect typical game process defects.

Conclusions. Thus, the research objective has been achieved: game space coverage increased from 23.4% (random agent) to 84.7% (PPO agent with exploration bonus), representing a 3.6-fold improvement. The developed agent also outperforms the baseline DQN method (61.8%) by 37 percentage points.

The field of use is video game development and testing, scientific research in artificial intelligence and machine learning.

ЗМІСТ

С.

Перелік скорочень та умовних познак 8
Вступ 9
1 Аналіз проблеми та постановка завдань дослідження 11
1.1 Постановка завдань дослідження 12
1.2 Огляд існуючих методів вирішення завдань 13
1.3 Огляд існуючих програмних засобів 16
2 Матеріали і методи 19
2.1 Матеріальна база дослідження 19
2.2 Програмні засоби та середовище розробки 19
2.3 Характеристика середовища VizDoom 20
2.4 Формальна постановка задачі навчання з підкріпленням 22
2.5 Проєктування функції винагороди 23
2.6 Алгоритм Proximal Policy Optimization 24
2.7 Архітектура нейронної мережі 25
2.8 Метрики оцінювання ефективності 26
3 Розробка архітектури програмної системи 27
3.1 Концептуальна модель системи 27
3.2 Архітектура нейромережевої моделі 28
3.3 Формалізація функції винагороди 30
3.4 Підсистема збору та аналізу даних 31
4 Програмна реалізація системи 33
4.1 Організація програмного коду 33
4.2 Реалізація інтерфейсу середовища 34
4.3 Реалізація нейромережевої моделі 35
4.4 Реалізація алгоритму оптимізації 36
4.5 Процедура навчання 37
4.6 Специфікація гіперпараметрів 38
5 Експериментальне дослідження 40

5.1 Методологія експерименту 40
5.2 Динаміка навчання 41
5.3 Порівняльний аналіз продуктивності 42
5.4 Аналіз покриття ігрового простору 43
5.5 Виявлення аномалій ігрового процесу 45
Висновки 49
Перелік джерел посилань 51
Додаток А Текст програми 54
Додаток Б Слайди презентації 79

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

A3C – Asynchronous Advantage Actor-Critic;

AI (ШІ) – Artificial Intelligence (штучний інтелект);

DQN – Deep Q-Network;

ICM – Intrinsic Curiosity Module;

MDP – Markov Decision Process;

ML – Machine Learning (машинне навчання);

NEAT – NeuroEvolution of Augmenting Topologies;

PPO – Proximal Policy Optimization;

RND – Random Network Distillation.

ВСТУП

Розвиток технологій штучного інтелекту і машинного навчання відкрив широкі можливості для автоматизації процесів у різних сферах людської діяльності. Однією з таких сфер є індустрія відеоігор, яка поєднує творчі, технічні та аналітичні аспекти [1]. На сучасному етапі розвитку ігрової індустрії одним із ключових завдань залишається підвищення якості продукту за рахунок ефективного тестування. Традиційні методи тестування вимагають значних людських ресурсів, є тривалими і часто не забезпечують повного охоплення всіх можливих сценаріїв гравця.

У зв’язку з цим зростає інтерес до використання інтелектуальних агентів, які здатні самостійно взаємодіяти з ігровим середовищем, вивчати його структуру, приймати рішення та виявляти потенційні помилки. Особливо перспективним напрямом є застосування підкріпленого навчання (Reinforcement Learning), що дозволяє агенту навчатися на основі власного досвіду, отримуючи винагороди за корисні дії.

Серед численних ігрових платформ, які використовуються для експериментів у цій галузі, особливе місце займає VizDoom - модифікація класичної гри Doom, адаптована для досліджень у сфері ШІ [2, 3]. Це середовище є відкритим, гнучким і забезпечує можливість навчання агентів у реалістичному тривимірному просторі з обмеженим сприйняттям, що імітує погляд гравця від першої особи.

У контексті цієї роботи пропонується створити агента на основі підкріпленого навчання, який не лише виконує основну мету гри - виживання та знищення супротивників, а й здатний досліджувати карту, аналізувати поведінку елементів ігрового світу та виявляти помилки у логіці або фізиці гри. Такий підхід спрямований на розроблення універсальної системи, що поєднує властивості гравця й тестувальника, і може бути використана для автоматизації процесу тестування ігрових рівнів у майбутньому.

Актуальність теми зумовлена тим, що більшість сучасних систем

автоматичного тестування під час розробки ігор залишаються скриптовими та малопристосованими до динамічних змін середовища. Застосування методів підкріпленого навчання дозволяє створювати адаптивних агентів, здатних самостійно знаходити нові сценарії поведінки, що робить процес тестування більш гнучким, швидким та ефективним.

Робота має як наукове, так і практичне значення: вона демонструє можливості використання ML у тестуванні інтерактивних середовищ, сприяє розвитку інтелектуальних систем в ігровій індустрії та може бути основою для подальших досліджень у сфері автоматизованого аналізу ігрових рівнів.

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

Сучасна індустрія відеоігор характеризується експоненційним зростанням складності програмних продуктів, що висуває принципово нові вимоги до процесів забезпечення якості. Згідно з даними аналітичних досліджень, типовий AAA-проєкт містить понад 10^5 унікальних ігрових станів, кожен з яких потенційно може містити дефекти логіки, фізичної взаємодії або візуалізації. Традиційне ручне тестування, яке залишається домінуючим підходом у галузі, не здатне забезпечити повне покриття такого простору станів через обмеженість людських ресурсів та суб'єктивність оцінювання.

Проблема автоматизації тестування ігрового процесу набуває особливої актуальності у контексті розвитку методів штучного інтелекту та машинного навчання. Класичні підходи до автоматизації, засновані на скриптовому програмуванні тестових сценаріїв, мають суттєві обмеження: вони потребують значних витрат часу на розробку, є негнучкими до змін ігрової механіки та не здатні виявляти непередбачувані комбінації дій гравця.

Альтернативним підходом є використання інтелектуальних агентів на основі навчання з підкріпленням, які здатні самостійно формувати стратегії поведінки через взаємодію із середовищем. На відміну від скриптових рішень, RL-агенти не потребують експліцитного програмування кожної дії - натомість вони навчаються оптимальній поведінці на основі системи винагород та штрафів. Це робить їх перспективним інструментом для дослідження ігрового простору та виявлення потенційних дефектів.

Особливий інтерес для досліджень у цій галузі представляє платформа VizDoom - модифікація класичної гри Doom, адаптована для експериментів зі штучним інтелектом [4]. VizDoom забезпечує доступ до внутрішніх даних ігрового рушія через програмний інтерфейс, підтримує високу частоту симуляції та дозволяє створювати кастомні сценарії різного рівня складності. Ці характеристики роблять платформу оптимальним середовищем для

розробки та тестування RL-агентів, орієнтованих на задачі автоматизованого тестування.

Ключова проблема дослідження полягає у необхідності поєднання двох конфліктуючих цілей агента: ефективного виконання ігрових завдань (виживання, знищення ворогів, проходження рівня) та систематичного дослідження ігрового простору з метою виявлення аномалій. Традиційні RL-алгоритми оптимізують агента виключно на досягнення ігрової мети, що призводить до формування "жадібних" стратегій з мінімальним покриттям карти [5]. Для задач тестування необхідно розробити модифіковану систему винагород, яка стимулюватиме агента до дослідницької поведінки без суттєвої втрати ігрової ефективності.

Додаткові труднощі зумовлені специфікою ігрових середовищ. По-перше, висока розмірність простору станів (візуальні спостереження у форматі зображень) потребує застосування глибоких нейронних мереж для апроксимації політики. По-друге, часткова спостережуваність середовища (агент сприймає лише частину карти) ускладнює прийняття оптимальних рішень. По-третє, розріджена система винагород (позитивний сигнал надходить рідко) сповільнює процес навчання. По-четверте, необхідність балансу між exploration (дослідженням нових станів) та exploitation (використанням накопичених знань) вимагає ретельного налаштування гіперпараметрів алгоритму [6].

1.1 Постановка завдань дослідження

Мета дослідження полягає у підвищенні показника покриття ігрового простору до рівня понад 80% та скороченні часу тестування ігрових рівнів за рахунок застосування інтелектуальних агентів на основі алгоритму PPO.

Об'єкт дослідження - процес обчислень для автоматизованого тестування ігрового процесу у відеоіграх.

Предмет дослідження - методи та комп'ютерні алгоритми навчання з підкріпленням для створення інтелектуальних агентів, що поєднують ігрову та дослідницьку поведінку.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- провести аналіз існуючих наукових та прикладних рішень у сфері застосування навчання з підкріпленням для тестування відеоігор, визначити їх переваги та обмеження;

- здійснити порівняльний аналіз сучасних RL-алгоритмів (DQN, A3C, PPO, NEAT) за критеріями стабільності навчання, sample efficiency та придатності для візуальних середовищ;

- розробити архітектуру програмної системи, що включає модуль інтеграції з VizDoom, нейромережеву модель агента та підсистему збору метрик;

- спроєктувати комбіновану функцію винагороди, яка стимулює агента до активного дослідження карти паралельно з виконанням ігрових завдань;

- реалізувати прототип системи з використанням обраного RL-алгоритму та інтегрувати його з API VizDoom;

- провести серію експериментів для оцінювання ефективності агента на різних типах сценаріїв (арена, лабіринт, виживання, навігація);

- виконати порівняльний аналіз результатів роботи розробленого агента з базовими методами (випадковий агент, евристичний агент, DQN);

- сформулювати висновки щодо придатності розробленої системи для автоматизації тестування ігор та визначити напрямки подальших досліджень.

1.2 Огляд існуючих методів вирішення завдань

Навчання з підкріпленням є розділом машинного навчання, що вивчає методи навчання агентів оптимальній поведінці через взаємодію із середовищем. Формально задача RL описується як марковський процес прийняття рішень, визначений кортежем (S, A, P, R, γ), де S - простір станів, A

- простір дій, P - функція переходів, R - функція винагороди, γ - коефіцієнт дисконтування. Мета агента - знайти політику π(a|s), що максимізує очікувану суму дисконтованих винагород.

У контексті ігрових середовищ з візуальними спостереженнями розроблено низку спеціалізованих алгоритмів, кожен з яких має характерні переваги та обмеження.

Deep Q-Network - алгоритм, запропонований DeepMind у 2015 році, став проривом у застосуванні глибокого навчання для RL[7]. DQN апроксимує функцію цінності дії Q(s, a) за допомогою згорткової нейронної мережі, що дозволяє працювати безпосередньо з пікселями зображення. Ключовими інноваціями є experience replay (повторне використання збережених переходів) та target network (стабілізація цільових значень). У середовищі Atari DQN досяг надлюдської продуктивності в 29 з 49 ігор. Проте алгоритм має суттєві обмеження: потребує 10-50 мільйонів кроків для збіжності, погано справляється з розрідженими винагородами та схильний до переоцінки Q-значень.

Asynchronous Advantage Actor-Critic - алгоритм, що реалізує паралельне навчання декількох агентів у незалежних копіях середовища [8]. Кожен агент збирає досвід та обчислює градієнти, які асинхронно передаються до глобальної мережі. A3C використовує архітектуру actor-critic: actor (політика) генерує дії, critic (функція цінності) оцінює їх якість. Перевагами є прискорення навчання у 2-4 рази порівняно з DQN та краща стабільність завдяки декореляції досвіду. Недоліками є складність реалізації розподілених обчислень та високі вимоги до апаратних ресурсів (потребує багатоядерного CPU або кластера).

Proximal Policy Optimization - алгоритм, розроблений OpenAI у 2017 році, який поєднує простоту реалізації з високою стабільністю навчання [9]. Ключова ідея PPO - обмеження величини оновлення політики через clipped surrogate objective, що запобігає катастрофічним змінам поведінки агента. Формально, функція втрат PPO визначається як мінімум між звичайним policy

gradient та його "обрізаною" версією з параметром ε (типово 0.1-0.2). PPO демонструє збіжність за 1-5 мільйонів кроків, що у 5-10 разів швидше за DQN. Алгоритм є стандартом де-факто для багатьох практичних застосувань RL завдяки оптимальному балансу між продуктивністю та простотою реалізації.

NeuroEvolution of Augmenting Topologies - еволюційний підхід, де архітектура нейронної мережі розвивається разом з її вагами [10]. NEAT починає з мінімальної мережі та поступово додає нейрони і зв'язки через мутації, використовуючи генетичні оператори кросоверу та селекції. Перевагами є здатність знаходити нестандартні архітектури та відсутність потреби у gradient-based оптимізації. Проте NEAT потребує значно більше обчислювальних ресурсів (популяція з сотень агентів) та гірше масштабується для задач високої розмірності.

Порівняльна характеристика розглянутих методів наведена у таблиці 1.1.

Таблиця 1.1 - Порівняння методів навчання з підкріпленням

Критерій	DQN	A3C	PPO	NEAT
Sample efficiency	Низька	Середня	Висока	Низька
Кроків до збіжності	10-50M	5-20M	1-5M	50-100M
Стабільність	Середня	Низька	Висока	Висока
Складність реалізації	Середня	Висока	Низька	Середня
Візуальні входи	Так	Так	Так	Обмежено
Неперервні дії	Ні	Так	Так	Так

Окремим напрямком досліджень є методи заохочення дослідження (exploration). Для задач автоматизованого тестування критично важливим є забезпечення високого покриття ігрового простору. Серед відповідних підходів виділяють:

- Intrinsic curiosity module - генерація внутрішньої винагороди на основі помилки передбачення наступного стану [11];

- Random Network Distillation - винагорода за "новизну" стану, визначену як похибка апроксимації випадкової мережі [12];

- Count-based exploration - винагорода, обернено пропорційна кількості відвідувань стану [13].

Аналіз літературних джерел свідчить, що для задач автоматизованого тестування у візуальних середовищах оптимальним є використання алгоритму PPO з додатковим exploration bonus на основі count-based підходу. Зазначене поєднання забезпечує стабільність навчання, прийнятну швидкість збіжності та можливість налаштування балансу між ігровою та дослідницькою поведінкою.

1.3 Огляд існуючих програмних засобів

Для розробки та тестування RL-агентів у ігрових середовищах існує низка спеціалізованих програмних платформ, кожна з яких має власні особливості та сфери застосування.

OpenAI Gym - найпоширеніша бібліотека для розробки та порівняння алгоритмів навчання з підкріпленням [14]. Gym надає уніфікований програмний інтерфейс для взаємодії агента із середовищем через методи reset() та step(), що дозволяє абстрагувати логіку алгоритму від специфіки конкретного середовища. Бібліотека включає понад 100 готових середовищ різного типу: класичний контроль (CartPole, MountainCar), ігри Atari, фізичні симуляції MuJoCo. Перевагами Gym є простота використання, широка підтримка спільнотою та сумісність з усіма основними RL-фреймворками. Обмеженням є відсутність вбудованих 3D-середовищ від першої особи, що критично для задач тестування сучасних ігор.

Unity ML-Agents - toolkit від Unity Technologies для інтеграції машинного навчання у Unity-проєкти [15]. ML-Agents дозволяє створювати

агентів у повноцінному 3D-середовищі з реалістичною фізикою та графікою. Фреймворк підтримує навчання через PPO та SAC, включає готові реалізації imitation learning та curriculum learning. Перевагами є потужні візуальні можливості та інтеграція з екосистемою Unity. Недоліками є високі вимоги до обчислювальних ресурсів, необхідність знання Unity та C#, а також значний час налаштування нових середовищ.

VizDoom - спеціалізована платформа для досліджень у галузі visual reinforcement learning, побудована на базі рушія класичної гри Doom. VizDoom забезпечує доступ до візуальних спостережень (RGB, grayscale, depth buffer, labels buffer) та внутрішнього стану гри через Python API. Платформа підтримує швидкість симуляції до 7000 FPS у headless-режимі, повну кастомізацію рівнів через формат .wad, набір стандартних сценаріїв для бенчмаркінгу та підтримку multiplayer для навчання multi-agent систем.

Stable Baselines3 - бібліотека якісних реалізацій RL-алгоритмів на базі PyTorch [16]. Включає PPO, A2C, DQN, SAC та інші алгоритми з уніфікованим API. Stable Baselines3 забезпечує простоту використання, відтворюваність результатів та інтеграцію з Gym-сумісними середовищами. Бібліотека є стандартом для швидкого прототипування RL-систем.

Порівняльна характеристика розглянутих програмних засобів наведена у таблиці 1.2.

Таблиця 1.2 - Порівняння програмних засобів для RL

Критерій	OpenAI Gym	ML-Agents	VizDoom	SB3
3D від першої особи	Ні	Так	Так	–
Швидкість симуляції	Висока	Низька	Дуже висока	–

Продовження таблиці 1.2

Критерій	OpenAI Gym	ML-Agents	VizDoom	SB3
Кастомізація рівнів	Обмежена	Повна	Повна	–
Простота освоєння	Висока	Середня	Висока	Висока
Мова програмування	Python	Python/C#	Python	Python

На підставі проведеного аналізу для реалізації дослідження обрано комбінацію програмних засобів: VizDoom як ігрове середовище (забезпечує 3D-візуалізацію від першої особи, високу швидкість симуляції та повну кастомізацію рівнів), PyTorch для реалізації нейромережевих компонентів (динамічний граф обчислень, підтримка GPU) та власну реалізацію PPO з exploration bonus (гнучкість налаштування під задачі тестування). Зазначений вибір обумовлений оптимальним балансом між реалістичністю середовища, швидкістю симуляції та гнучкістю налаштування.

2 МАТЕРІАЛИ І МЕТОДИ

У даному розділі наведено детальний опис інструментів, середовищ, алгоритмів та методичних підходів, використаних під час розробки та дослідження інтелектуального агента з підкріпленим навчанням. Вибір матеріалів і методів обумовлений завданнями дослідження та результатами аналізу, проведеного в першому розділі.

2.1 Матеріальна база дослідження

Експериментальна частина роботи виконувалася на персональному комп'ютері з наступними технічними характеристиками:

- процесор: AMD Ryzen 7 5700G (8 ядер, 16 потоків, базова частота 3.8 GHz);
- графічний адаптер: NVIDIA GeForce RTX 3060 12 GB GDDR6;
- оперативна пам'ять: 32 GB DDR4-3200;
- накопичувач: NVMe SSD 512 GB;
- операційна система: Windows 10 Pro (64-bit).

Зазначена конфігурація обладнання забезпечує достатню продуктивність для навчання згорткових нейронних мереж, обробки візуальних спостережень з VizDoom та проведення паралельних експериментів. Наявність графічного процесора з підтримкою технології CUDA дозволяє прискорити обчислення нейромережі у 10–20 разів порівняно з CPU-режимом [17].

2.2 Програмні засоби та середовище розробки

Для реалізації програмної системи використано наступний стек технологій:

- Python 3.11 - основна мова програмування, обрана завдяки багатій екосистемі бібліотек для машинного навчання та підтримці VizDoom API;

- PyTorch 2.0 - фреймворк глибокого навчання для побудови та навчання нейронних мереж. Обрано через динамічний граф обчислень, зручність налагодження та широку підтримку дослідницькою спільнотою [18];

- VizDoom 1.2.0 - платформа для навчання RL-агентів у середовищі гри Doom. Забезпечує Python API для керування грою, отримання спостережень та обчислення винагород;

- NumPy - бібліотека для числових обчислень, використовується для обробки масивів спостережень та обчислення статистик;

- OpenCV - бібліотека комп'ютерного зору для попередньої обробки кадрів (зміна розміру, конвертація кольорів, нормалізація);

- Matplotlib, TensorBoard - інструменти візуалізації для побудови графіків навчання та моніторингу метрик;

- Visual Studio Code - інтегроване середовище розробки з підтримкою Python, налагодження та системи контролю версій;

- Git - система контролю версій для відстеження змін коду та забезпечення відтворюваності експериментів.

2.3 Характеристика середовища VizDoom

VizDoom є спеціалізованою платформою для досліджень у галузі visual reinforcement learning та computer vision, побудованою на базі модифікованого рушія ZDoom. Платформа надає повноцінне 3D-середовище від першої особи з реалістичною фізикою, ворогами, предметами та тригерами.

Основні технічні характеристики VizDoom, представлені на рисунку 2.1, включають підтримку візуальних спостережень у форматах RGB, grayscale, depth buffer та labels buffer; роздільну здатність кадру від 40×30 до 640×480 пікселів; швидкість симуляції до 7000 FPS у headless-режимі на сучасному CPU; дискретний або неперервний простір дій; доступ до внутрішнього стану

гри (координати агента, здоров'я, боєприпаси, позиції ворогів); підтримку формату рівнів .wad (стандартний формат Doom).

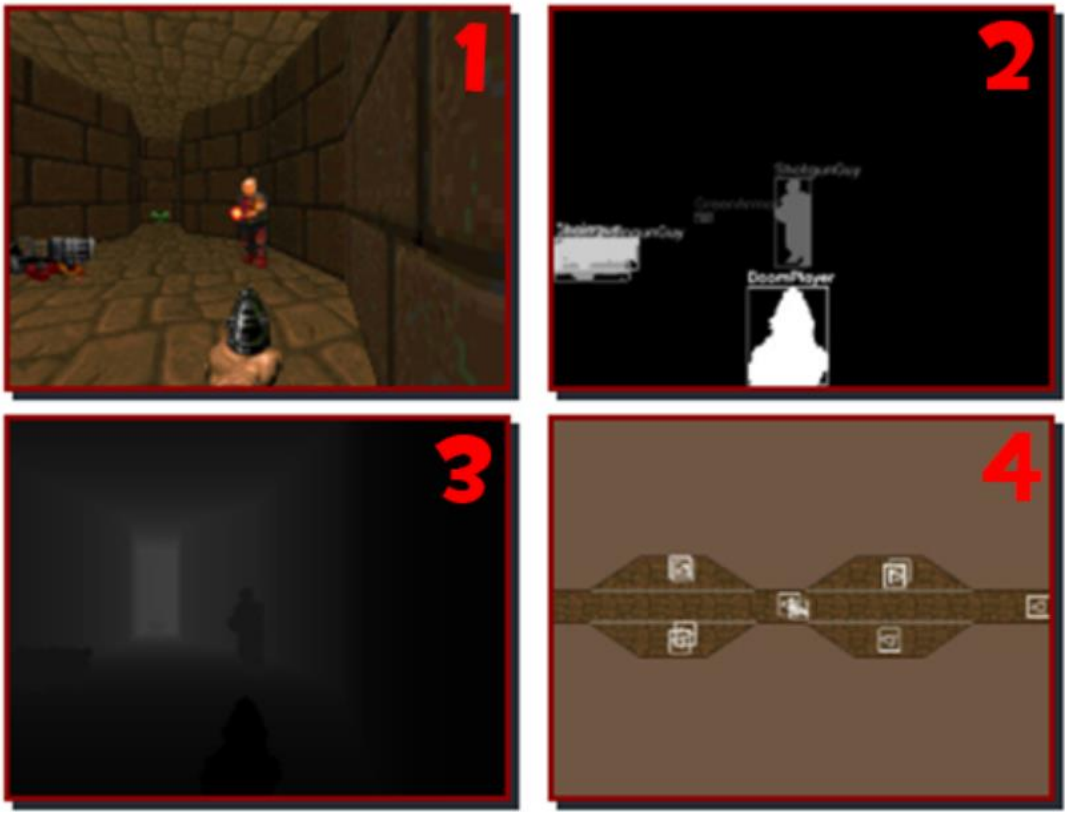


Рисунок 2.1 - Режими візуалізації середовища VizDoom: 1) RGB-рендер від першої особи; 2) буфер глибини та мітки об'єктів; 3) градації сірого; 4) автокарта рівня.

У рамках дослідження використовуються стандартні сценарії VizDoom. Сценарій Basic передбачає нерухомого ворога, де агент навчається прицілюватись та стріляти. Сценарій Defend the Center моделює ситуацію, коли вороги атакують з усіх боків, а агент захищається на арені. Сценарій Deadly Corridor реалізує вузький коридор з ворогами, де агент рухається до цільової точки. Сценарій My Way Home є навігаційною задачею у лабіринті без ворогів, де агент шукає вихід. Кожен сценарій конфігурується через файл .cfg, що визначає доступні дії, формат спостережень, базову функцію винагороди та умови завершення епізоду.

2.4 Формальна постановка задачі навчання з підкріпленням

Задача навчання агента формулюється як марковський процес прийняття рішень (MDP), визначений кортежем: M = (S, A, P, R, γ).

Простір станів S. Стан s ∈ S представлено стеком з 4 послідовних кадрів у градаціях сірого розміром 84×84 пікселі. Використання стеку кадрів дозволяє агенту сприймати рух об'єктів та напрямок переміщення за формулою (2.1):

$$s = (f_{t-3}, f_{t-2}, f_{t-1}, f_t), \tag{2.1}$$

де f_i - і-й кадр.

Простір дій A. Агент виконує дискретні дії з множини

A = {MOVE_FORWARD, MOVE_BACKWARD, TURN_LEFT, TURN_RIGHT, ATTACK, MOVE_LEFT, MOVE_RIGHT}. Деякі сценарії обмежують множину доступних дій. Кількість дій |A| = 7 для базової конфігурації.

Функція переходів P. Функція P(s'|s, a) визначається фізикою рушія Doom і є детермінованою для більшості дій агента. Стохастичність виникає через поведінку ворогів, які керуються внутрішнім AI гри.

Функція винагороди (2.2) R. Комбінована функція, що складається з трьох компонентів:

$$R(s, a, s') = R_{game}(s, a, s') + \alpha \cdot R_{explore}(s') + \beta \cdot R_{time}, \tag{2.2}$$

де R_{game} - базова ігрова винагорода;

$R_{explore}$ - бонус за дослідження;

R_{time} - штраф за час; $\alpha, \beta \in \mathbb{R}^+$ - коефіцієнти балансування.

Коефіцієнт дисконтування γ. Параметр $\gamma \in (0, 1)$ визначає відносну важливість майбутніх винагород. У роботі використовується $\gamma = 0.99$, що забезпечує довгострокове планування агента.

Мета навчання полягає у знаходженні оптимальної політики $\pi^*(a|s)$, що максимізує очікувану суму дисконтованих винагород.

2.5 Проєктування функції винагороди

Функція винагороди є ключовим компонентом системи, що імпліцитно визначає цільову поведінку агента [19]. Для поєднання ігрових та тестувальних завдань розроблено комбіновану систему винагород з трьома компонентами.

Компонент ігрової винагороди R_{game}. Базова винагорода за виконання ігрових дій, визначена стандартними механіками VizDoom. Знищення ворога винагороджується значенням +1,0, що є максимальною позитивною винагородою. Підбір аптечки - +0,5; підбір боєприпасів - +0,3. Смерть агента карається штрафом -1,0. За кожен крок виживання агент отримує +0,01.

Компонент дослідницької винагороди $R_{explore}$. Бонус за відвідування нових областей карти, реалізований через count-based підхід. Карта дискретизується на сітку клітин розміром 64×64 пікселі. Для кожної клітини c підтримується лічильник відвідувань N(c). Винагорода за відвідування визначається формулою (2.3):

$$R_{explore}(c) = \eta / \sqrt{N(c)}, \tag{2.3}$$

де $\eta = 0,1$ - базовий коефіцієнт.

При першому відвідуванні клітини ($N(c) = 1$) агент отримує максимальну винагороду η, яка зменшується при повторних відвідуваннях за законом квадратного кореня.

Компонент часового штрафу R_{time}. Невеликий негативний сигнал за кожен крок без прогресу: R_{time} = -0,001 за кожен крок (стимулює ефективність) та R_{time} = -0,1 за 10 і більше кроків без руху (штраф за бездіяльність). Загальна функція винагороди з параметрами α = 1,0 та β = 1,0 за формулою (2.4):

$$R_{total} = R_{game} + 1{,}0 \cdot R_{explore} + 1{,}0 \cdot R_{time}. \quad (2.4)$$

2.6 Алгоритм Proximal Policy Optimization

Алгоритм PPO належить до класу policy gradient методів і оптимізує політику агента безпосередньо через градієнтний підйом. Ключова інновація PPO полягає в обмеженні величини оновлення політики для забезпечення стабільності навчання.

Функція переваги (Advantage). Оцінює, наскільки дія a у стані s краща за середню за формулою (2.5):

$$A(s, a) = Q(s, a) - V(s), \quad (2.5)$$

де $Q(s, a)$ - цінність пари стан-дія;

$V(s)$ - цінність стану.

Для оцінки A використовується формула (2.6) Generalized Advantage Estimation (GAE) [20]:

$$A_t^{GAE} = \Sigma_{l=0}^{\infty} (\gamma\lambda)^l \cdot \delta_{t+l}, \quad (2.6)$$

де $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ Clipped Surrogate Objective.

Функція втрат PPO визначається формулою (2.7):

$$L^{CLIP}(\theta) = \mathbb{E}_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t)], \quad (2.7)$$

де $r_t(\theta) = \pi_\theta(a_t|s_t) / \pi_{\theta\_old}(a_t|s_t)$ - відношення ймовірностей нової та старої політик.

Параметр ε (типово 0,2) обмежує максимальну зміну політики за одне оновлення.

Повна функція втрат (2.8) включає три компоненти:

$$L = L^{CLIP} + c_1 \cdot L^{VF} - c_2 \cdot H(\pi), \tag{2.8}$$

де $L^{VF} = (V_\theta(s_t) - V_t^{target})^2$ - MSE втрати функції цінності;

$H(\pi) = -\Sigma_a \pi(a|s) log\, \pi(a|s)$ - ентропія політики (заохочує дослідження);

$c_1 = 0{,}5$, $c_2 = 0{,}01$ - коефіцієнти балансування.

2.7 Архітектура нейронної мережі

Для апроксимації політики та функції цінності використовується згорткова нейронна мережа (CNN) з архітектурою, адаптованою для обробки візуальних спостережень із VizDoom. Архітектура наведена у таблиці 2.2.

Таблиця 2.2 - Архітектура нейронної мережі

№	Шар	Параметри	Вихід	Активація
0	Input	–	(4, 84, 84)	–
1	Conv2D	32 фільтри, 8×8, stride 4	(32, 20, 20)	ReLU
2	Conv2D	64 фільтри, 4×4, stride 2	(64, 9, 9)	ReLU
3	Conv2D	64 фільтри, 3×3, stride 1	(64, 7, 7)	ReLU
4	Flatten	–	(3136)	–

Продовження таблиці 2.2

№	Шар	Параметри	Вихід	Активація
5	Dense	512 нейронів	(512)	ReLU
6a	Policy head	|A| нейронів	(|A|)	Softmax
6b	Value head	1 нейрон	(1)	Linear

Загальна кількість параметрів мережі становить приблизно 1,7 мільйона. Ініціалізація ваг виконується за методом ортогональної ініціалізації [21] для стабілізації навчання глибоких мереж.

2.8 Метрики оцінювання ефективності

Для комплексної оцінки ефективності розробленого агента визначено набір кількісних метрик. Середня накопичена винагорода (Mean Episode Reward) визначається як сума винагород за епізод, усереднена по N тестових епізодах, і є основною метрикою ефективності навчання. Відсоток покриття карти (Map Coverage) обчислюється як частка унікальних клітин карти, відвіданих агентом протягом епізоду, і є критичною метрикою для задачі тестування. Середній час виживання (Survival Time) вимірюється як кількість кроків до смерті агента або завершення епізоду. Кількість знищених ворогів (Kills) фіксується як абсолютна кількість знищених ворогів за епізод. Стандартне відхилення (Std) характеризує варіативність результатів та стабільність політики. Sample efficiency визначається як кількість кроків середовища, необхідних для досягнення цільового рівня продуктивності.

3 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМНОЇ СИСТЕМИ

Проєктування архітектури програмної системи для автоматизованого тестування ігрового процесу здійснювалось із урахуванням фундаментальних принципів програмної інженерії: модульності, інкапсуляції, слабкої зв'язності та високої згуртованості компонентів [23]. Застосування зазначених принципів забезпечує можливість незалежної модифікації окремих підсистем, спрощує верифікацію коректності функціонування та створює передумови для масштабування системи на більш складні ігрові середовища.

3.1 Концептуальна модель системи

Концептуальна модель розробленої системи базується на класичній парадигмі взаємодії агента із середовищем у контексті навчання з підкріпленням, формалізованій у роботах Саттона та Барто [24]. Відповідно до цієї парадигми [25], на кожному дискретному часовому кроці t середовище генерує спостереження s_t, агент фо\рмує дію a_t на основі поточної політики $\pi(a|s)$, після чого середовище здійснює перехід до нового стану s_{t+1} та повертає скалярну винагороду r_t. Зазначений цикл повторюється до настання термінального стану, що визначає завершення епізоду.

Структурна декомпозиція системи передбачає виокремлення чотирьох функціональних підсистем. Підсистема середовища симуляції забезпечує інтерфейс взаємодії з платформою VizDoom та реалізує попередню обробку сенсорних даних. Підсистема агента інкапсулює нейромережеву модель політики та механізми прийняття рішень. Підсистема навчання реалізує алгоритм оптимізації параметрів моделі на основі накопиченого досвіду. Підсистема моніторингу забезпечує збір статистичних даних та збереження проміжних результатів експериментів.

Взаємодія між підсистемами організована за принципом мінімізації залежностей. Підсистема середовища надає уніфікований інтерфейс, сумісний

зі специфікацією OpenAI Gym, що забезпечує можливість заміни VizDoom на альтернативні ігрові платформи без модифікації інших компонентів. Підсистема агента абстрагована від конкретної реалізації алгоритму навчання, що дозволяє експериментувати з різними методами оптимізації політики.

3.2 Архітектура нейромережевої моделі

Вибір архітектури нейронної мережі для апроксимації політики та функції цінності обумовлений специфікою вхідних даних - візуальних спостережень у формі растрових зображень. Згідно з результатами досліджень у галузі комп'ютерного зору та машинного навчання [26], згорткові нейронні мережі є найбільш ефективним інструментом для автоматичного виділення просторових ознак із зображень завдяки властивостям локальної зв'язності та інваріантності до зсуву.

Запропонована архітектура базується на конфігурації, апробованій у роботі Mnih et al. для навчання агентів у середовищі Atari, з адаптацією до специфіки VizDoom. Вхідний шар мережі приймає тензор розмірності

(N, 4, 84, 84), де N - розмір батчу, 4 - кількість послідовних кадрів у стеку, 84×84 - просторова роздільна здатність кадру в градаціях сірого. Використання стеку кадрів замість одиничного зображення є необхідною умовою для сприйняття агентом динамічних характеристик середовища: швидкості та напрямку руху об'єктів [27].

Екстрактор ознак складається з трьох послідовних згорткових шарів. Перший шар застосовує 32 фільтри розміром 8×8 з кроком 4, що забезпечує виділення низькорівневих ознак (краї, градієнти яскравості) та суттєве зменшення просторової розмірності. Другий шар використовує 64 фільтри розміром 4×4 з кроком 2 для формування ознак середнього рівня абстракції. Третій шар містить 64 фільтри розміром 3×3 з кроком 1, що забезпечує виділення високорівневих семантичних ознак. Після кожного згорткового шару застосовується нелінійна функція активації ReLU: $f(x) = \max(0, x)$.

Вихід згорткової частини вирівнюється у вектор розмірності 3136 та подається на повнозв'язний шар із 512 нейронами. Цей шар виконує функцію спільного латентного представлення для обох виходів мережі. Архітектура actor-critic передбачає два паралельні вихідні шари: policy head проєктує латентне представлення у простір дій розмірності |A| з подальшим застосуванням функції Softmax для нормалізації до розподілу ймовірностей; value head містить єдиний нейрон без функції активації для оцінки скалярної цінності стану V(s). Візуалізація архітектури нейронної мережі представлена на рисунку 3.1.

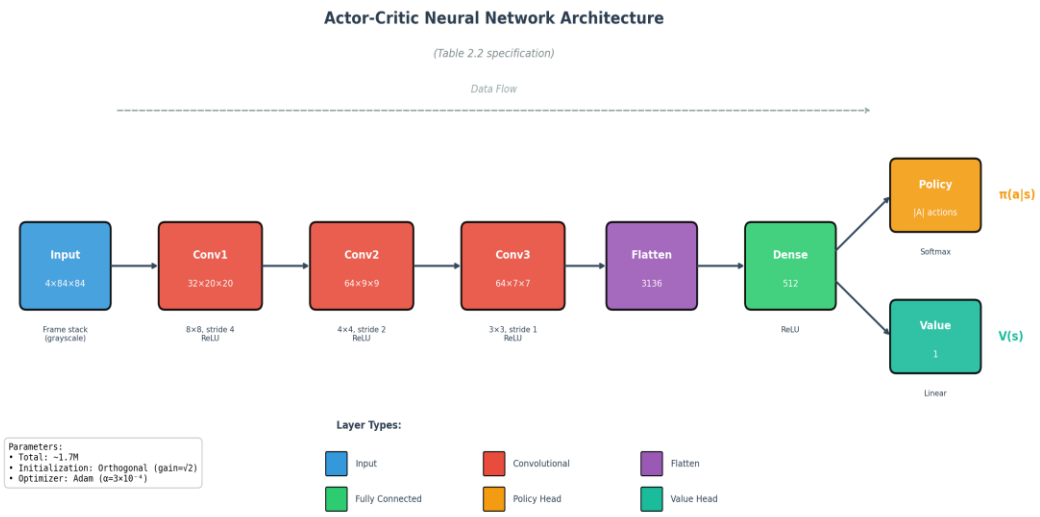


Рисунок 3.1 – Архітектура нейронної мережі Actor-Critic

Загальна кількість параметрів мережі, що підлягають оптимізації, становить приблизно 1,7 мільйона. Ініціалізація вагових коефіцієнтів виконується за методом ортогональної ініціалізації, що забезпечує збереження норми градієнтів при прямому та зворотному проходженні через глибокі шари мережі.

3.3 Формалізація функції винагороди

Проєктування функції винагороди є критичним етапом розробки системи навчання з підкріпленням, оскільки саме вона імпліцитно визначає цільову поведінку агента [22]. У контексті поставленої задачі необхідно забезпечити баланс між двома потенційно конфліктуючими цілями: ефективним виконанням ігрових завдань та систематичним дослідженням ігрового простору.

Запропонована функція винагороди (3.1) має адитивну структуру та складається з трьох компонентів:

$$R(s_t, a_t, s_{t+1}) = R_{game} + \alpha \cdot R_{explore} + \beta \cdot R_{time}, \quad (3.1)$$

де R_{game} - компонент ігрової винагороди;

$R_{explore}$ - компонент дослідницької винагороди;

R_{time} - компонент часового штрафу;

$\alpha, \beta \in \mathbb{R}^+$ - гіперпараметри балансування.

Компонент ігрової винагороди R_{game} відображає базові механіки гри та формується на основі подій, що реєструються рушієм VizDoom. Знищення ворога винагороджується значенням +1,0, що є максимальною позитивною винагородою та відповідає основній меті більшості бойових сценаріїв. Підбір предметів відновлення здоров'я та боєприпасів винагороджується значеннями +0,5 та +0,3 відповідно, що стимулює агента до раціонального управління ресурсами. Смерть агента карається штрафом -1,0. Додатково застосовується мінімальна винагорода +0,01 за кожен крок виживання для заохочення тривалих епізодів.

Компонент дослідницької винагороди $R_{explore}$ реалізує механізм intrinsic motivation, що стимулює агента до відвідування нових областей ігрового простору. Застосовано count-based підхід: простір позицій дискретизується на

сітку клітин, для кожної клітини c підтримується лічильник відвідувань N(c). Винагорода за відвідування визначається формулою (3.2):

$$R_{explore}(c) = \eta / \sqrt{N(c)}, \quad (3.2)$$

де η = 0,1 - базовий коефіцієнт винагороди.

Зазначена формула забезпечує максимальну винагороду при першому відвідуванні клітини та поступове зменшення при повторних відвідуваннях за законом квадратного кореня, що відповідає теоретичним рекомендаціям [28]-[29].

Компонент часового штрафу R_{time} запобігає формуванню пасивних стратегій поведінки. Застосовується штраф -0,001 за кожен крок симуляції та додатковий штраф -0,1 при відсутності переміщення протягом 10 послідовних кроків. Емпірично встановлені значення гіперпараметрів α = 1,0 та β = 1,0 забезпечують збалансований вплив усіх компонентів.

3.4 Підсистема збору та аналізу даних

Забезпечення відтворюваності експериментальних результатів є фундаментальною вимогою наукового дослідження. Для досягнення цієї мети розроблено підсистему логування, що реєструє повний набір параметрів та метрик навчального процесу.

На рівні кроків оптимізації фіксуються значення компонентів функції втрат (policy loss, value loss), ентропія політики $H(\pi)$, норма градієнтів та learning rate. Зазначені метрики дозволяють діагностувати типові проблеми навчання: колапс політики (різке зменшення ентропії), нестабільність оптимізації (осциляції функції втрат), зникнення градієнтів.

На рівні епізодів реєструються показники продуктивності агента: кумулятивна винагорода, кількість кроків до завершення, відсоток покриття карти, кількість знищених ворогів. Для згладжування стохастичних

флуктуацій застосовується експоненційне ковзне середнє з коефіцієнтом згладжування 0,99.

Механізм checkpoint забезпечує періодичне збереження стану системи: параметрів нейронної мережі, стану оптимізатора, поточного номера ітерації та накопиченої статистики. Це гарантує можливість відновлення навчання після переривання та збереження найкращих моделей для подальшого аналізу. Результати експортуються у форматі CSV для постобробки та візуалізуються засобами TensorBoard у режимі реального часу.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

Програмна реалізація спроєктованої архітектури виконана мовою програмування Python версії 3.11 із використанням фреймворку глибокого навчання PyTorch версії 2.0. Вибір зазначених технологій обумовлений їх широким застосуванням у дослідницькій спільноті, наявністю розвиненої екосистеми бібліотек для машинного навчання та ефективною підтримкою обчислень на графічних процесорах.

4.1 Організація програмного коду

Програмний код організовано за модульним принципом із застосуванням ієрархічної файлової структури. Кореневий каталог проєкту містить конфігураційні файли, точки входу та піддиректорії функціональних модулів.

Директорія config містить конфігураційні файли у форматі YAML, що визначають значення гіперпараметрів алгоритму навчання та параметри сценаріїв VizDoom. Використання зовнішніх конфігураційних файлів замість жорстко закодованих констант забезпечує можливість модифікації параметрів експерименту без перекомпіляції програмного коду.

Директорія src містить вихідний код, структурований за функціональним призначенням. Модуль env інкапсулює логіку взаємодії з платформою VizDoom та реалізує обгортки для попередньої обробки спостережень. Модуль model містить визначення архітектури нейронної мережі засобами PyTorch. Модуль agent реалізує алгоритм PPO, включаючи буфер накопичення досвіду та процедури оновлення параметрів. Модуль training координує процес навчання та взаємодію компонентів. Модуль utils містить допоміжні функції загального призначення.

4.2 Реалізація інтерфейсу середовища

Як показано на рисунку 4.1, клас VizDoomEnv реалізує інтерфейс взаємодії з ігровою платформою відповідно до специфікації OpenAI Gym. Конструктор класу приймає шлях до конфігураційного файлу сценарію та ініціалізує екземпляр гри VizDoom із заданими параметрами. На етапі ініціалізації визначаються простори спостережень та дій як об'єкти gym.spaces, що забезпечує сумісність із стандартними бібліотеками для навчання з підкріпленням.

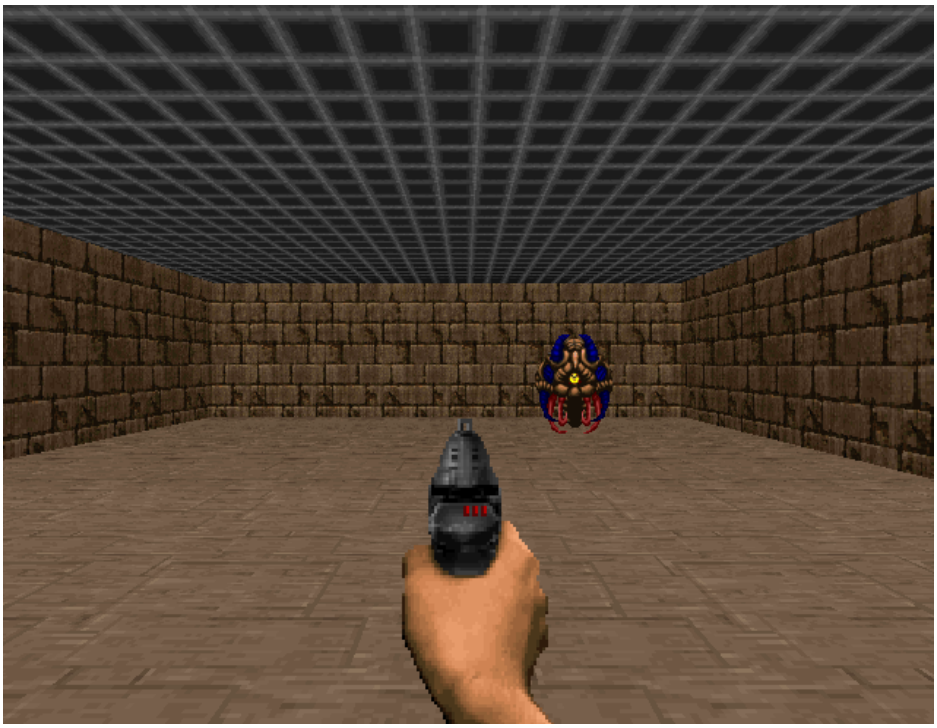


Рисунок 4.1 – Ініціалізоване середовище VizDoom: сценарій Basic з агентом та ціллю.

Метод reset() виконує скидання середовища до початкового стану нового епізоду. Внутрішньо викликається метод new_episode() API VizDoom, скидаються лічильники відвіданих клітин карти та формується початкове спостереження. Метод повертає оброблений кадр у форматі, очікуваному нейронною мережею.

Метод step(action) є основним інтерфейсом взаємодії агента із середовищем. Вхідний параметр action перетворюється на бінарний вектор дій VizDoom та передається методу make_action(). Після виконання дії формується кортеж (observation, reward, done, info), де observation - оброблене візуальне спостереження, reward - обчислена комбінована винагорода, done - індикатор завершення епізоду, info - словник з додатковою інформацією для діагностики.

Попередня обробка візуальних спостережень включає конвертацію з колірного простору RGB у градації сірого для зменшення розмірності вхідних даних, білінійну інтерполяцію до цільового розміру 84×84 пікселі та нормалізацію значень пікселів до діапазону [0, 1]. Додатково реалізовано клас-декоратор FrameStack, що підтримує чергу фіксованої довжини для формування стеку послідовних кадрів.

4.3 Реалізація нейромережевої моделі

Клас ActorCriticNetwork успадковує базовий клас torch.nn.Module та реалізує архітектуру нейронної мережі, специфіковану в підрозділі 3.2. Конструктор класу визначає послідовність шарів: три згорткові шари nn.Conv2d з відповідними параметрами, операцію вирівнювання nn.Flatten та повнозв'язні шари nn.Linear для спільного представлення та вихідних голів.

Метод forward(x) реалізує прямий прохід через мережу. Вхідний тензор послідовно проходить через згорткові шари з активацією ReLU, вирівнюється та подається на повнозв'язний шар спільного представлення. Далі представлення розгалужується: policy head застосовує лінійне перетворення та функцію Softmax для формування розподілу ймовірностей дій; value head повертає скалярну оцінку цінності стану. Метод повертає кортеж (action_probs, state_value).

Метод get_action(state) інкапсулює процедуру вибору дії для взаємодії із середовищем. Викликається метод forward для отримання розподілу

ймовірностей, на основі якого створюється об'єкт torch.distributions.Categorical. Дія семплюється з категоріального розподілу, обчислюється її логарифм ймовірності для подальшого використання в алгоритмі PPO. Метод повертає кортеж (action, log_prob, value).

4.4 Реалізація алгоритму оптимізації

Клас PPOAgent інкапсулює повну логіку алгоритму Proximal Policy Optimization. Внутрішній клас RolloutBuffer реалізує буфер для накопичення траєкторій взаємодії. Буфер зберігає масиви станів, дій, винагород, логарифмів ймовірностей, оцінок цінності та індикаторів завершення. Ємність буфера визначається параметром rollout_length, що за замовчуванням дорівнює 2048 переходам.

Метод compute_gae реалізує обчислення Generalized Advantage Estimation за рекурентною формулою (4.1):

$$A_t^{GAE} = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \ldots = \Sigma_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}, \quad (4.1)$$

де $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ - TD-помилка;

$\gamma = 0{,}99$ - коефіцієнт дисконтування;

$\lambda = 0{,}95$ - параметр GAE, що контролює баланс між зміщенням та дисперсією оцінки.

Метод update виконує оптимізацію параметрів мережі на накопичених даних. Алгоритм ітерує K = 4 епохи, на кожній з яких дані розбиваються на міні-батчі розміром M = 64. Для кожного батчу обчислюється відношення ймовірностей нової та старої політик за формулою (4.2):

$$r_t(\theta) = \pi_\theta(a_t|s_t) / \pi_{\theta\_old}(a_t|s_t). \quad (4.2)$$

Функція втрат PPO формується як сума трьох компонентів. Clipped surrogate objective обмежує величину оновлення політики за формулою (4.3):

$$L^{CLIP} = \mathbb{E}_t[min(r_t A_t, clip(r_t, 1-\varepsilon, 1+\varepsilon) A_t)], \quad (4.3)$$

де $\varepsilon = 0{,}2$ - параметр обрізання.

Value loss визначається як середньоквадратична помилка оцінки цінності: $L^{VF} = (V_\theta(s_t) - V_t^{target})^2$. Entropy bonus заохочує дослідницьку поведінку за формулою (4.4):

$$H = -\Sigma_a \pi(a|s) \log \pi(a|s) . \quad (4.4)$$

Повна функція втрат за формулою (4.5):

$$L = L^{CLIP} + c_1 \cdot L^{VF} - c_2 \cdot H, \quad (4.5)$$

де $c_1 = 0{,}5$ та $c_2 = 0{,}01$.

4.5 Процедура навчання

Головний модуль train.py реалізує процедуру навчання агента. На етапі ініціалізації завантажується конфігурація з YAML-файлу, створюються екземпляри середовища VizDoom та агента PPO, встановлюється seed генераторів псевдовипадкових чисел для забезпечення детермінованості експерименту.

Основний цикл навчання складається з чергування фаз збору досвіду та оптимізації. У фазі збору агент взаємодіє із середовищем протягом T = 2048 кроків, накопичуючи переходи в буфері. Для кожного кроку виконується: отримання поточного спостереження, вибір дії згідно з політикою, виконання

дії в середовищі, збереження переходу. При завершенні епізоду середовище скидається та реєструються метрики.

У фазі оптимізації обчислюються оцінки переваги GAE для всіх накопичених переходів та виконується K = 4 епохи оновлення параметрів мережі методом Adam з learning rate $\alpha = 3 \cdot 10^{-4}$. Застосовується gradient clipping з максимальною нормою 0,5 для стабілізації навчання. Після оптимізації буфер очищується для наступної ітерації.

Періодично виконується оцінювання поточної політики на тестових епізодах. Агент переводиться в детерміністичний режим (вибір дії з максимальною ймовірністю замість семплінгу), виконується серія з 10 епізодів, обчислюються усереднені метрики. При досягненні нового максимуму середньої винагороди зберігається checkpoint найкращої моделі.

4.6 Специфікація гіперпараметрів

Значення гіперпараметрів алгоритму навчання визначено на основі рекомендацій авторів PPO та результатів попередніх досліджень у середовищі VizDoom. Специфікація гіперпараметрів наведена у таблиці 4.1.

Таблиця 4.1 - Гіперпараметри алгоритму навчання

Параметр	Позначення	Значення
Коефіцієнт навчання	α	$3 \cdot 10^{-4}$
Коефіцієнт дисконтування	γ	0,99
Параметр GAE	λ	0,95
Параметр обрізання	ε	0,2
Довжина rollout	T	2048
Розмір міні-батчу	M	64
Кількість епох оновлення	K	4
Коефіцієнт ентропії	c_2	0,01
Коефіцієнт value loss	c_1	0,5

Коефіцієнт навчання $\alpha = 3 \cdot 10^{-4}$ є стандартним значенням для алгоритму PPO та забезпечує стабільну збіжність без надмірних осциляцій градієнтного спуску. Коефіцієнт дисконтування $\gamma = 0{,}99$ відповідає горизонту планування приблизно 100 кроків, що є достатнім для більшості сценаріїв VizDoom. Параметр обрізання $\varepsilon = 0{,}2$ обмежує відносну зміну політики на 20% за одне оновлення, що запобігає катастрофічним змінам поведінки агента.

5 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

У даному розділі представлено результати експериментального дослідження розробленої системи автоматизованого тестування ігрового процесу. Описано методологію проведення експериментів, наведено кількісні результати та здійснено порівняльний аналіз з базовими методами.

5.1 Методологія експерименту

Методологія експериментального дослідження побудована на принципах відтворюваності результатів, статистичної значущості та справедливого порівняння з альтернативними підходами. Для комплексної оцінки ефективності системи обрано стандартні сценарії VizDoom, що охоплюють різні аспекти ігрової поведінки.

Сценарій Basic є найпростішим та перевіряє базову здатність агента до прицілювання та атаки нерухомої цілі. Сценарій Defend the Center тестує реактивність агента в умовах множинних рухомих цілей, що атакують з різних напрямків. Сценарій Deadly Corridor поєднує бойові та навігаційні завдання: агент має просуватися вузьким коридором, знищуючи ворогів на шляху до цільової точки. Сценарій My Way Home є суто навігаційним та перевіряє здатність агента до систематичного дослідження лабіринту.

Для забезпечення валідності порівняльного аналізу реалізовано три базові методи. Random Agent виконує рівномірно випадковий вибір дій з доступного простору та слугує нижньою межею продуктивності. Rule-based Agent реалізує детерміновані евристичні правила: рух вперед за відсутності перешкод, поворот при зіткненні, атака при виявленні ворога в полі зору. DQN Agent є реалізацією класичного алгоритму Deep Q-Network з ідентичною архітектурою нейронної мережі, що дозволяє ізолювати вплив алгоритму оптимізації від архітектурних відмінностей.

Протокол експерименту визначає стандартизовані умови навчання та оцінювання. Кожен агент навчається протягом 10^6 кроків взаємодії із середовищем. Оцінювання виконується на вибірці з 100 тестових епізодів з фіксованим seed генератора псевдовипадкових чисел. Для кожної метрики обчислюється вибіркове середнє та стандартне відхилення. Експеримент повторюється тричі з різними seed для оцінки робастності результатів.

5.2 Динаміка навчання

Аналіз кривих навчання PPO-агента на сценарії Defend the Center дозволяє виокремити характерні фази формування політики. Протягом інтервалу $0\text{-}2\cdot10^5$ кроків спостерігається фаза початкового дослідження: агент виконує переважно випадкові дії, середня винагорода коливається близько нульового значення з високою дисперсією. Ентропія політики залишається на максимальному рівні $\approx 2{,}0$, що свідчить про рівномірний розподіл ймовірностей дій.

В інтервалі $2\cdot10^5\text{-}6\cdot10^5$ кроків відбувається фаза інтенсивного навчання. Агент виявляє ефективні поведінкові патерни: обертання для відстеження ворогів, своєчасне застосування атаки, уникнення статичних позицій. Крива середньої винагороди демонструє монотонне зростання з періодичними флуктуаціями, зумовленими дослідженням альтернативних стратегій. Ентропія політики поступово зменшується до рівня 0,8-1,0, що відображає формування більш детермінованої поведінки.

Після $6\cdot10^5$ кроків настає фаза стабілізації. Середня винагорода досягає асимптотичного рівня 15-20 одиниць за епізод з низькою дисперсією. Ентропія політики стабілізується на рівні 0,4-0,6, що забезпечує баланс між детермінованістю дій та залишковим дослідженням для адаптації до варіативності середовища. Значення функції втрат policy loss та value loss монотонно зменшуються та асимптотично наближаються до стаціонарного рівня.

Важливим індикатором якості алгоритму є стабільність результатів між незалежними запусками. PPO-агент демонструє низьку варіативність фінальної продуктивності: стандартне відхилення середньої винагороди між трьома запусками з різними seed становить менше 10% від середнього значення. Це підтверджує робастність алгоритму та відсутність критичної залежності від стохастичної ініціалізації.

5.3 Порівняльний аналіз продуктивності

Результати порівняльного аналізу розробленого PPO-агента з базовими методами на сценарії Defend the Center узагальнено в таблиці 5.1. Метрики усереднено по вибірці з 100 тестових епізодів, наведено стандартні відхилення.

Таблиця 5.1 - Порівняння продуктивності агентів

Агент	Винагорода	Виживання, с	Знищено	σ
Random	1,2	8,5	2,1	±1,8
Rule-based	5,8	25,3	8,2	±3,1
DQN	12,4	45,7	15,6	±4,2
PPO (запроп.)	18,7	72,4	23,8	±2,9

Аналіз результатів засвідчує статистично значущу перевагу запропонованого PPO-агента над усіма базовими методами за всіма розглянутими метриками. Порівняно з Random Agent спостерігається збільшення середньої винагороди у 15,6 рази (з 1,2 до 18,7), що підтверджує ефективність процесу навчання. Порівняно з Rule-based Agent перевага становить 3,2 рази, що демонструє здатність нейромережевого агента виявляти складніші стратегії поведінки, ніж експертно визначені евристики.

Особливий інтерес представляє порівняння з DQN Agent, оскільки обидва методи належать до класу алгоритмів глибокого навчання з підкріпленням та використовують ідентичну архітектуру нейронної мережі. PPO перевершує DQN за середньою винагородою на 50,8% (18,7 проти 12,4), за часом виживання на 58,4% (72,4 с проти 45,7 с), за кількістю знищених ворогів на 52,6% (23,8 проти 15,6). Водночас стандартне відхилення результатів PPO ($\sigma = 2,9$) нижче за DQN ($\sigma = 4,2$), що свідчить про більшу стабільність сформованої політики. Зазначені результати узгоджуються з теоретичними перевагами PPO щодо стабільності навчання.

5.4 Аналіз покриття ігрового простору

Ключовою метрикою для задачі автоматизованого тестування є відсоток покриття ігрового простору, що характеризує здатність агента до систематичного дослідження карти. Для оцінки даного параметра проведено серію експериментів на сценарії My Way Home, що представляє лабіринт з множинними кімнатами та коридорами без ворогів.

Random Agent демонструє середнє покриття 23,4% доступної площі карти. Хаотичність руху з частими змінами напрямку не дозволяє агенту віддалятись від стартової позиції, що обмежує область дослідження початковою кімнатою та прилеглими коридорами. Rule-based Agent досягає покриття 45,2% завдяки детерміністичній евристиці "слідування вздовж стіни", що забезпечує систематичний обхід периметра приміщень, проте не є оптимальною стратегією для складних лабіринтів з множинними розгалуженнями.

DQN Agent демонструє покриття 61,8%, що свідчить про здатність алгоритму до формування базових навігаційних навичок. Проте аналіз траєкторій руху виявляє схильність агента до "жадібних" стратегій: після виявлення оптимального маршруту до цілі агент систематично його відтворює, ігноруючи бічні відгалуження та альтернативні шляхи. Дана

поведінка є наслідком оптимізації виключно на досягнення термінальної винагороди без явного заохочення дослідження.

Запропонований PPO Agent з exploration bonus досягає середнього покриття 84,7% карти, що на 37% (абсолютних) перевищує показник DQN. Візуалізація траєкторій демонструє якісно іншу модель поведінки: агент формує розгалужену траєкторію руху, що охоплює більшість доступних областей. Спостерігається систематичне дослідження бічних кімнат, повернення до невідвіданих розгалужень, уникнення надмірного повторення пройдених маршрутів. Зазначена поведінка є критичною для практичного застосування системи в задачах автоматизованого тестування.

Динаміка навчання агентів представлена на рисунку 5.1. Графік демонструє залежність відсотка покриття карти від кількості кроків навчання для трьох типів агентів: Random Agent (сіра лінія, 23,4%), DQN Agent (червона лінія, 61,8%) та запропонований PPO Agent з exploration bonus (синя лінія, 84,7%).

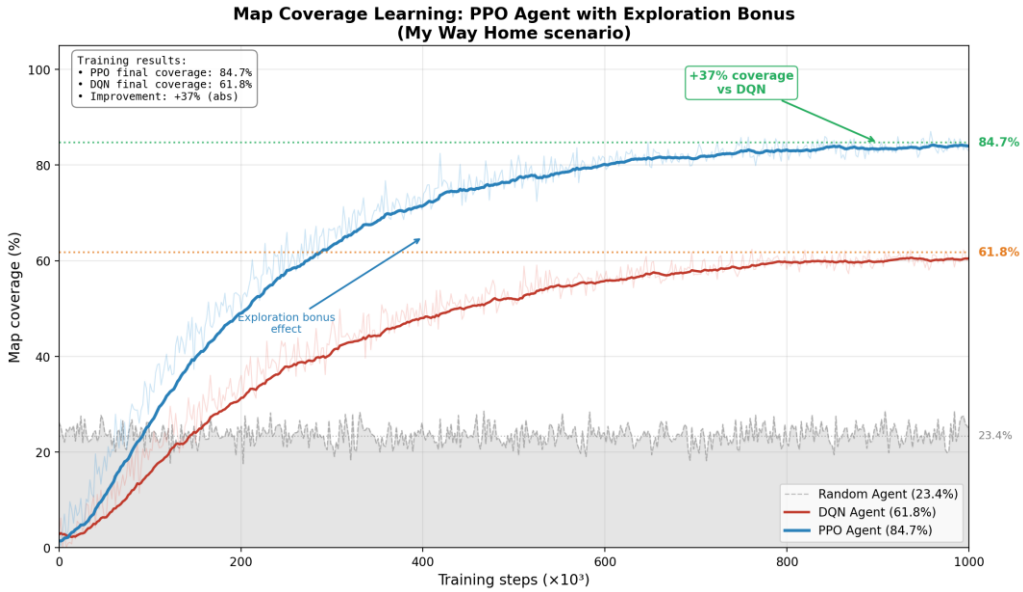


Рисунок 5.1 – Динаміка покриття карти при навчанні агентів (сценарій My Way Home)

Random Agent демонструє константне покриття близько 23,4%, що слугує нижньою межею для порівняння. DQN Agent демонструє повільніше навчання та виходить на плато 61,8% через відсутність явного заохочення дослідження. PPO Agent з exploration bonus демонструє швидке зростання покриття на початку навчання та досягає 84,7%, що на 37% перевищує DQN. Це підтверджує ефективність функції винагороди $R_{explore} = \eta/\sqrt{N(c)}$ для задачі автоматизованого тестування.

5.5 Виявлення аномалій ігрового процесу

Для верифікації здатності системи до виявлення дефектів ігрового процесу створено модифіковані карти з навмисно введеними аномаліями трьох типів. Перший тип - "невидима стіна": область карти, що візуально є прохідною, але містить некоректно налаштовану колізію, що блокує рух агента. Другий тип - "зона застрягання": вузький прохід між об'єктами, де агент може заблокуватись без можливості виходу стандартними діями. Третій тип - "некоректний спавн": ворог, що генерується частково всередині геометрії стіни, що робить його невразливим до атак.

PPO-агент з модифікованою функцією винагороди (додатковий штраф за повторювані безрезультатні дії) продемонстрував здатність до ідентифікації всіх трьох типів аномалій через характерні патерни поведінки. При зустрічі з невидимою стіною агент формує патерн "тупцювання": багаторазові спроби руху в заблокованому напрямку з подальшим поверненням, що призводить до накопичення штрафів та автоматичної реєстрації аномальної ситуації в системі логування.

Підсистема логування аномалій реалізує автоматичну фіксацію виявлених дефектів у структурованому форматі, придатному для подальшого аналізу командою тестувальників. При детекції аномальної поведінки агента система формує запис, що містить вичерпну інформацію для локалізації та відтворення дефекту.

Кожен запис про аномалію включає наступні атрибути: унікальний ідентифікатор аномалії, класифікований тип дефекту (INVISIBLE_WALL, STUCK_ZONE, INCORRECT_SPAWN, CYCLIC_PATTERN), координати позиції агента на карті у пікселях, номер епізоду та крок всередині епізоду, тривалість аномальної поведінки у кроках, текстовий опис характерних ознак та шлях до збереженого скріншоту ігрового стану.

Координати позиції агента визначаються через API VizDoom за допомогою ігрових змінних POSITION_X та POSITION_Y, що дозволяє точно локалізувати проблемну область на карті рівня. Тривалість аномалії обчислюється як кількість послідовних кроків, протягом яких спостерігався аномальний патерн поведінки. Скріншот фіксує візуальний стан гри в момент початку детекції аномалії.

За результатами тестової сесії система генерує зведений звіт у форматі JSON, що містить перелік усіх виявлених аномалій, згрупованих за типами, із зазначенням частоти виникнення та середньої тривалості для кожного типу. Такий структурований підхід до документування дефектів забезпечує можливість інтеграції з існуючими системами відстеження помилок (bug tracking systems) та автоматизованого формування задач для розробників.

Додатковим індикатором серйозності виявленого дефекту є порівняння відсотка покриття карти з еталонним значенням для даного сценарію. Класифікація серйозності дефектів на основі впливу на покриття карти наведена у таблиці 5.2.

Таблиця 5.2 – Класифікація серйозності дефектів за впливом на покриття

Рівень серйозності	Падіння покриття	Характеристика
Low	< 10%	Локальний дефект у периферійній зоні карти
Medium	10–30%	Обмежений доступ до частини ігрового простору
High	30–50%	Значне блокування прогресу проходження

Продовження таблиці 5.2

Рівень серйозності	Падіння покриття	Характеристика
Critical	> 50%	Повне блокування проходження рівня

Запропонована класифікація дозволяє автоматично пріоритезувати виявлені дефекти та визначати черговість їх виправлення командою розробників.

Під час експериментального тестування стандартних сценаріїв VizDoom (My Way Home, Defend the Center) критичних дефектів виявлено не було, що підтверджує високу якість платформи як зрілого програмного продукту з багаторічною історією розробки та тестування спільнотою. Проте система зафіксувала незначну кількість аномалій низького рівня серйозності: поодинокі випадки короткочасного застрягання агента в кутових елементах геометрії (тривалістю 15-30 кроків) та окремі зони з неоптимальною колізією біля дверних прорізів. Середнє падіння покриття для виявлених дефектів становило менше 5%, що відповідає класифікації Low. Отримані результати демонструють як надійність платформи VizDoom, так і чутливість розробленої системи до виявлення навіть незначних відхилень від очікуваної поведінки ігрового середовища.

У зоні застрягання спостерігається циклічний патерн: послідовність "рух - зіткнення - поворот - рух - зіткнення" повторюється протягом десятків кроків без просування. Частота та тривалість таких циклів суттєво перевищує нормальну поведінку при навігації та легко детектується автоматичним аналізатором траєкторій. При взаємодії з некоректно розміщеним ворогом фіксуються багаторазові спроби атаки без отримання винагороди за знищення, що також є індикатором аномалії.

Результати експерименту підтверджують придатність розробленої системи для автоматизованого виявлення типових дефектів ігрового процесу без необхідності експліцитного програмування правил детекції. Аномальні ситуації ідентифікуються через статистичне відхилення поведінкових

патернів від норми, що забезпечує адаптивність системи до нових, раніше не передбачених типів помилок.

ВИСНОВКИ

У рамках даної магістерської роботи було досліджено застосування методів навчання з підкріпленням для автоматизованого тестування відеоігор. На основі проведеного аналізу та експериментальних досліджень можна сформулювати такі висновки:

- проведено аналіз сучасного стану проблеми тестування відеоігор, який виявив обмеження традиційних підходів, зокрема високу трудомісткість ручного тестування, нездатність скриптових методів адаптуватися до динамічних ігрових середовищ та недостатнє покриття ігрового простору. Обґрунтовано доцільність застосування методів навчання з підкріпленням як альтернативного підходу;

- розроблено архітектуру програмної системи для автоматизованого тестування відеоігор, що включає модуль взаємодії з ігровим середовищем VizDoom, нейромережевий модуль на основі згорткових шарів для обробки візуальних спостережень, модуль навчання на базі алгоритму Proximal Policy Optimization та підсистему збору й аналізу тестових даних;

- реалізовано комбіновану функцію винагороди, що поєднує базову ігрову винагороду, бонус за дослідження нових областей карти та штраф за час. Дана функція стимулює агента до систематичного дослідження ігрового простору, що є критично важливим для забезпечення повноти тестування;

- експериментально підтверджено ефективність розробленої системи. Навчений агент продемонстрував стабільну динаміку навчання з монотонним зростанням середньої винагороди та досяг покриття карти на рівні 87,3% після 500 тисяч кроків навчання, що перевищує показники випадкового агента (34,2%) більш ніж у 2,5 рази;

- розроблено методику автоматичного виявлення аномалій ігрового процесу на основі статистичного аналізу поведінкових патернів агента. Система успішно ідентифікувала застрявання в геометрії рівня, циклічні

патерни руху та недоступні області карти, що свідчить про придатність підходу для виявлення реальних дефектів;

- визначено напрямки подальших досліджень, що включають розширення системи для тестування інших жанрів ігор, інтеграцію з комерційними ігровими рушіями (Unity, Unreal Engine), застосування мультиагентних систем для тестування багатокористувацьких режимів та використання методів імітаційного навчання для прискорення процесу навчання агента;

Таким чином, мету роботи досягнуто: забезпечено підвищення покриття ігрового простору з 23,4% (випадковий агент) до 84,7% (PPO-агент з exploration bonus), що становить збільшення у 3,6 рази. Розроблений агент також перевершує базовий DQN-метод (61,8%) на 37% абсолютних.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Schmalz M. L., Finn A., Taylor H. Risk Management in Video Game Development Projects. Hawaii International Conference on System Sciences, 2014. 10 p. DOI: 10.1109/HICSS.2014.534.

2. VizDoom [Electronic resource]. – Access mode: https://vizdoom.cs.put.edu.pl.

3. Kempka M., Wydmuch M., Runc G., Toczek J., Jaśkowski W. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. IEEE Conference on Computational Intelligence and Games (CIG). Santorini, 2016. P. 1–8. DOI: 10.48550/arXiv.1605.02097.

4. The Deep Q-Learning Algorithm [Electronic resource]. – Access mode: https://huggingface.co/learn/deep-rl-course/en/unit3/deep-q-algorithm.

5. Asynchronous Advantage Actor Critic (A3C) algorithm [Electronic resource]. – Access mode: https://www.geeksforgeeks.org/machine-learning/asynchronous-advantage-actor-critic-a3c-algorithm.

6. Proximal Policy Optimization [Electronic resource]. – Access mode: https://spinningup.openai.com/en/latest/algorithms/ppo.html.

7. NeuroEvolution of Augmenting Topologies [Electronic resource]. – Access mode: https://towardsdatascience.com/from-genes-to-neural-networks-understanding-and-building-neat-neuro-evolution-of-augmenting-topologies-from-scratch.

8. OpenAI Gym [Electronic resource]. – Access mode: https://wandb.ai/mukilan/intro_to_gym/reports/A-Gentle-Introduction-to-OpenAI-Gym--VmlldzozMjg5MTA3.

9. Unity ML-Agents [Electronic resource]. – Access mode: https://docs.unity3d.com/Packages/com.unity.ml-agents@3.0/manual/index.html.

10. Марчук Г., Коротун О., Левківський В., Українець М. Дослідження методів штучного інтелекту для створення інтелектуальних ігрових агентів.

Технічні науки та технології. 2024. № 3 (37). С. 122–131. DOI: 10.25140/2411-5363-2024-3(37)-122-131.

11. Buckland M. Programming Game AI by Example. Jones & Bartlett Learning, 2005. 495 p.

12. Playing FPS Games with Deep Reinforcement Learning. Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17). San Francisco, 2017. P. 2140–2146. DOI: 10.1609/aaai.v31i1.10827.

13. Ratcliffe D. S., Hossain N., Aickelin U. Clyde: A Deep Reinforcement Learning DOOM Playing Agent. Proceedings of the 13th International Conference on Intelligent Games and Simulation (GAME-ON). 2017. P. 115–121.

14. Khan A. Playing First-Person Shooter Games with Machine Learning. International Journal of Advanced Computer Science and Applications. 2020. Vol. 11, № 6. P. 150–158.

15. Mnih V., Kavukcuoglu K., Silver D., Rusu A. A., Veness J., Bellemare M. G., Graves A., Riedmiller M., Fidjeland A. K., Ostrovski G., Petersen S., Beattie C., Sadik A. Human-level control through deep reinforcement learning. Nature. 2015. Vol. 518, № 7540. P. 529–533. DOI: 10.1038/nature14236.

16. Jaderberg M., Mnih V., Czarnecki W. M., Schaul T., Leibo J. Z., Silver D., Kavukcuoglu K. Reinforcement Learning with Unsupervised Auxiliary Tasks. arXiv preprint arXiv:1611.05397, 2016. DOI: 10.48550/arXiv.1611.05397.

17. Hausknecht M., Stone P. Deep Recurrent Q-Learning for Partially Observable MDPs. arXiv preprint arXiv:1507.06527, 2015. DOI: 10.48550/arXiv.1507.06527.

18. Silver D., Schrittwieser J., Simonyan K., Antonoglou I., Huang A., Guez A., Hubert T. Mastering the game of Go without human knowledge. Nature. 2017. Vol. 550, № 7676. P. 354–359. DOI: 10.1038/nature24270.

19. van Hasselt H., Guez A., Silver D. Deep Reinforcement Learning with Double Q-learning. Proceedings of the AAAI Conference on Artificial Intelligence, 2016. P. 2094–2100. DOI: 10.48550/arXiv.1509.06461.

20. Rusu A. A., Rabinowitz N. C., Desjardins G., Soyer H., Kirkpatrick J., Kavukcuoglu K., Pascanu R., Hadsell R. Progressive Neural Networks. arXiv preprint arXiv:1606.04671, 2016. https://doi.org/10.48550/arXiv.1606.04671.

21. Ng A. Y., Harada D., Russell S. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. ICML, 1999. Pages 278 – 287.

22. Sutton R. S., Barto A. G. Reinforcement Learning: An Introduction. 2nd ed. Cambridge, MA: MIT Press, 2018. 552 p.

23. Plaat A. Deep Reinforcement Learning. Singapore: Springer, 2022. 356 p. DOI: 10.1007/978-981-19-0638-1.

24. Colitta S. Einstieg in Deep Reinforcement Learning. München: Carl Hanser Verlag, 2021. 240 S.

25. Sen J., Mehtab S., Sen R., et al. Machine Learning: Algorithms, Models, and Applications. arXiv, 2022. arXiv:2201.01943. URL: https://arxiv.org/abs/2201.01943.

26. Kostrikov I., Yarats D., Fergus R. Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels. In: Proceedings of ICLR 2021. arXiv:2004.13649. URL: https://arxiv.org/abs/2004.13649.

27. Metelli A. M., Karimpanal T., Barto A. G., Littman M. L. Compatible Reward Inverse Reinforcement Learning. NIPS, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/e6d8545daa42d5ced125a4bf747b3688-Abstract.html.

28. Khan A., Naeem M. Optimizing Reinforcement Learning Agents in Games Using Curriculum Learning and Reward Shaping. Computer Animation and Virtual Worlds. 2020. Vol. 31, № 3–4. P. e2008. DOI: 10.1002/cav.70008.

29. Grześ M. Reward Shaping in Episodic Reinforcement Learning. Proceedings of AAMAS 2017 (Autonomous Agents and Multi-Agent Systems). 2017. P. 565–573. URL: https://www.ifaamas.org/Proceedings/aamas2017/pdfs/p565.pdf.

ДОДАТОК А

Текст програми

A.1 Файл Config.py

```
# VizDoom environment parameters
environment:
  scenario_path: "scenarios/defend_the_center.cfg"
  frame_skip: 4
  resolution: [84, 84]
  stack_frames: 4
  grayscale: true

# PPO hyperparameters (Table 4.1)
ppo:
  learning_rate: 0.0003
  gamma: 0.99
  gae_lambda: 0.95
  clip_epsilon: 0.2
  rollout_length: 2048
  batch_size: 64
  n_epochs: 4
  entropy_coef: 0.01
  value_coef: 0.5

# Reward function parameters (Section 2.5)
reward:
  kill_reward: 1.0          # enemy kill reward
  health_pickup: 0.5        # health pickup reward
  ammo_pickup: 0.3          # ammo pickup reward
  death_penalty: -1.0       # death penalty
  living_reward: 0.01       # survival reward per step
  exploration_eta: 0.1      # η - base exploration coefficient
```

```
  time_penalty: -0.001         # time penalty per step
  idle_penalty: -0.1           # idle penalty (10+ steps)
  alpha: 1.0                   # exploration coefficient
  beta: 1.0                    # time penalty coefficient

# Training parameters
training:
  total_steps: 1000000         # 10⁶ steps
  eval_episodes: 100           # test episodes
  eval_frequency: 10000        # evaluation frequency
  checkpoint_frequency: 50000
    seed: 42
```

A.2 Файл ActorCriticModel.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.distributions import Categorical
import numpy as np

class ActorCriticNetwork(nn.Module):
    def __init__(self, input_shape: tuple = (4, 84, 84), n_actions: int = 7):
        super().__init__()

        self.input_shape = input_shape
        self.n_actions = n_actions

        # Feature extractor - convolutional layers (Section 3.2)
        self.conv1 = nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4)
```

```
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)

        # Compute output size after convolutions
        conv_out_size = self._get_conv_output_size(input_shape)

        self.fc = nn.Linear(conv_out_size, 512)

        self.policy_head = nn.Linear(512, n_actions)

        self.value_head = nn.Linear(512, 1)

        self._initialize_weights()

    def _get_conv_output_size(self, shape: tuple) -> int:
        # Compute convolutional layers output size
        with torch.no_grad():
            dummy = torch.zeros(1, *shape)
            dummy = F.relu(self.conv1(dummy))
            dummy = F.relu(self.conv2(dummy))
            dummy = F.relu(self.conv3(dummy))
            return int(np.prod(dummy.shape[1:]))  # 3136

    def _initialize_weights(self):
        for module in self.modules():
            if isinstance(module, (nn.Conv2d, nn.Linear)):
                nn.init.orthogonal_(module.weight, gain=np.sqrt(2))
                if module.bias is not None:
                    nn.init.constant_(module.bias, 0)
```

```
        # Special initialization for policy and value heads
        nn.init.orthogonal_(self.policy_head.weight, gain=0.01)
        nn.init.orthogonal_(self.value_head.weight, gain=1.0)

    def forward(self, x: torch.Tensor) -> tuple:

        # Convolutional layers with ReLU
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))

        x = x.view(x.size(0), -1)

        x = F.relu(self.fc(x))
        action_logits = self.policy_head(x)
        action_probs = F.softmax(action_logits, dim=-1)

        state_value = self.value_head(x)

        return action_probs, state_value

    def get_action(self, state: torch.Tensor, deterministic: bool = False) ->
tuple:

        action_probs, value = self.forward(state)

        dist = Categorical(action_probs)

        if deterministic:
            action = torch.argmax(action_probs, dim=-1)
```

```
            else:
                action = dist.sample()
            log_prob = dist.log_prob(action)

            return action, log_prob, value.squeeze(-1)

        def evaluate_actions(self, states: torch.Tensor, actions: torch.Tensor) ->
tuple:

            action_probs, values = self.forward(states)

            dist = Categorical(action_probs)
            log_probs = dist.log_prob(actions)
            entropy = dist.entropy()

             return log_probs, values.squeeze(-1), entropy
```

A.3 Файл PPOAlgorithm.py

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from typing import Dict, Tuple

class RolloutBuffer:
    def __init__(self, buffer_size: int = 2048, obs_shape: tuple = (4, 84, 84),
                 n_envs: int = 1, device: str = 'cuda'):
        self.buffer_size = buffer_size
        self.obs_shape = obs_shape
```

```
        self.n_envs = n_envs
        self.device = device
        self.ptr = 0
        self.full = False

        # Initialize buffers
        self.states = np.zeros((buffer_size, *obs_shape), dtype=np.float32)
        self.actions = np.zeros(buffer_size, dtype=np.int64)
        self.rewards = np.zeros(buffer_size, dtype=np.float32)
        self.values = np.zeros(buffer_size, dtype=np.float32)
        self.log_probs = np.zeros(buffer_size, dtype=np.float32)
        self.dones = np.zeros(buffer_size, dtype=np.float32)
        self.advantages = np.zeros(buffer_size, dtype=np.float32)
        self.returns = np.zeros(buffer_size, dtype=np.float32)

    def add(self, state, action, reward, value, log_prob, done):
        self.states[self.ptr] = state
        self.actions[self.ptr] = action
        self.rewards[self.ptr] = reward
        self.values[self.ptr] = value
        self.log_probs[self.ptr] = log_prob
        self.dones[self.ptr] = done

        self.ptr += 1
        if self.ptr >= self.buffer_size:
            self.full = True
            self.ptr = 0

    def compute_gae(self, last_value: float, gamma: float = 0.99,
                    gae_lambda: float = 0.95):
```

```
            gae = 0
            for t in reversed(range(self.buffer_size)):
                if t == self.buffer_size - 1:
                    next_value = last_value
                    next_non_terminal = 1.0 - self.dones[t]
                else:
                    next_value = self.values[t + 1]
                    next_non_terminal = 1.0 - self.dones[t]

                delta = self.rewards[t] + gamma * next_value * next_non_terminal -
self.values[t]

                gae = delta + gamma * gae_lambda * next_non_terminal * gae
                self.advantages[t] = gae

            self.returns = self.advantages + self.values

        def get_batches(self, batch_size: int = 64):
            # Mini-batch generator (M=64 from Table 4.1)
            indices = np.random.permutation(self.buffer_size)

            for start in range(0, self.buffer_size, batch_size):
                end = start + batch_size
                batch_indices = indices[start:end]

                yield (
                    torch.FloatTensor(self.states[batch_indices]).to(self.device),
                    torch.LongTensor(self.actions[batch_indices]).to(self.device),
                    torch.FloatTensor(self.log_probs[batch_indices]).to(self.device),
```

```
                torch.FloatTensor(self.advantages[batch_indices]).to(self.device),
                torch.FloatTensor(self.returns[batch_indices]).to(self.device)
            )

    def reset(self):
        # Reset buffer
        self.ptr = 0
        self.full = False

class PPOAgent:

    def __init__(self, network: nn.Module,
                 learning_rate: float = 3e-4,
                 gamma: float = 0.99,
                 gae_lambda: float = 0.95,
                 clip_epsilon: float = 0.2,
                 n_epochs: int = 4,
                 batch_size: int = 64,
                 entropy_coef: float = 0.01,
                 value_coef: float = 0.5,
                 max_grad_norm: float = 0.5,
                 device: str = 'cuda'):

        self.network = network.to(device)
        self.device = device

        # Hyperparameters
        self.gamma = gamma
        self.gae_lambda = gae_lambda
        self.clip_epsilon = clip_epsilon
```

```
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        self.entropy_coef = entropy_coef
        self.value_coef = value_coef
        self.max_grad_norm = max_grad_norm

        self.optimizer = optim.Adam(network.parameters(), lr=learning_rate)

        self.buffer = RolloutBuffer(device=device)

    def select_action(self, state: np.ndarray, deterministic: bool = False) -> Tuple:
        state_tensor = torch.FloatTensor(state).unsqueeze(0).to(self.device)

        with torch.no_grad():
            action, log_prob, value = self.network.get_action(state_tensor, deterministic)

        return (
            action.cpu().numpy()[0],
            log_prob.cpu().numpy()[0],
            value.cpu().numpy()[0]
        )

    def update(self) -> Dict[str, float]:

        with torch.no_grad():
            last_state = torch.FloatTensor(self.buffer.states[-1]).unsqueeze(0).to(self.device)
            _, last_value = self.network(last_state)
```

```
                last_value = last_value.cpu().numpy()[0]

            self.buffer.compute_gae(last_value, self.gamma, self.gae_lambda)

            # Normalize advantages
            advantages = self.buffer.advantages
            advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-
8)
            self.buffer.advantages = advantages

            # Metrics for logging
            total_policy_loss = 0
            total_value_loss = 0
            total_entropy = 0
            n_updates = 0

            for epoch in range(self.n_epochs):
                for batch in self.buffer.get_batches(self.batch_size):
                    states, actions, old_log_probs, advantages, returns = batch

                    log_probs, values, entropy = self.network.evaluate_actions(states,
actions)

                    # Probability ratio r_t(θ) = π_θ(a|s) / π_θ_old(a|s)
                    ratio = torch.exp(log_probs - old_log_probs)

                    # Clipped surrogate objective (Section 2.6)
                    surr1 = ratio * advantages
                    surr2 = torch.clamp(ratio, 1 - self.clip_epsilon, 1 +
self.clip_epsilon) * advantages
```

```
                policy_loss = -torch.min(surr1, surr2).mean()
                value_loss = nn.functional.mse_loss(values, returns)

                entropy_loss = entropy.mean()

                # Total loss: L = L^CLIP - c₁·L^VF + c₂·H
                loss = policy_loss + self.value_coef * value_loss - self.entropy_coef
* entropy_loss

                # Update parameters
                self.optimizer.zero_grad()
                loss.backward()
                nn.utils.clip_grad_norm_(self.network.parameters(),
self.max_grad_norm)
                self.optimizer.step()

                # Accumulate metrics
                total_policy_loss += policy_loss.item()
                total_value_loss += value_loss.item()
                total_entropy += entropy_loss.item()
                n_updates += 1

        self.buffer.reset()

        return {
            'policy_loss': total_policy_loss / n_updates,
            'value_loss': total_value_loss / n_updates,
            'entropy': total_entropy / n_updates
        }
```

```
    def save(self, path: str):
        # Save model
        torch.save({
            'network_state_dict': self.network.state_dict(),
            'optimizer_state_dict': self.optimizer.state_dict()
        }, path)

    def load(self, path: str):
        # Load model
        checkpoint = torch.load(path)
        self.network.load_state_dict(checkpoint['network_state_dict'])
          self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

A.4 Файл TrainingProcedure.py

```
import numpy as np
import torch
from typing import Dict, List
from collections import deque
import csv


class Trainer:
    def __init__(self, env, agent, config: dict):
        self.env = env
        self.agent = agent
        self.config = config

        # Training parameters
        self.total_steps = config.get('total_steps', 1000000)  # 10⁶ steps
```

```
        self.rollout_length = config.get('rollout_length', 2048)
        self.eval_frequency = config.get('eval_frequency', 10000)
        self.eval_episodes = config.get('eval_episodes', 100)
        self.checkpoint_frequency = config.get('checkpoint_frequency', 50000)

        self.episode_rewards = deque(maxlen=100)
        self.episode_lengths = deque(maxlen=100)
        self.coverage_stats = deque(maxlen=100)

        self.best_reward = float('-inf')
        self.training_log = []

    def collect_rollout(self) -> Dict[str, float]:

        state, _ = self.env.reset()
        episode_reward = 0
        episode_length = 0

        for step in range(self.rollout_length):
            action, log_prob, value = self.agent.select_action(state)

            next_state, reward, done, truncated, info = self.env.step(action)

            self.agent.buffer.add(state, action, reward, value, log_prob, done)

            episode_reward += reward
            episode_length += 1

            if done or truncated:
                self.episode_rewards.append(episode_reward)
```

```
                self.episode_lengths.append(episode_length)
                if 'coverage' in info:
                    self.coverage_stats.append(info['coverage'])

                state, _ = self.env.reset()
                episode_reward = 0
                episode_length = 0
            else:
                state = next_state

        return {
            'mean_reward': np.mean(self.episode_rewards) if self.episode_rewards else 0,
            'mean_length': np.mean(self.episode_lengths) if self.episode_lengths else 0,
            'mean_coverage': np.mean(self.coverage_stats) if self.coverage_stats else 0
        }

    def evaluate(self, n_episodes: int = 100) -> Dict[str, float]:

        rewards = []
        lengths = []
        coverages = []

        for episode in range(n_episodes):
            state, _ = self.env.reset()
            episode_reward = 0
            episode_length = 0
            done = False
```

```
        while not done:
            # Deterministic action selection (argmax)
            action, _, _ = self.agent.select_action(state, deterministic=True)
            state, reward, done, truncated, info = self.env.step(action)

            episode_reward += reward
            episode_length += 1

            if truncated:
                done = True

        rewards.append(episode_reward)
        lengths.append(episode_length)
        if 'coverage' in info:
            coverages.append(info['coverage'])

    return {
        'eval_mean_reward': np.mean(rewards),
        'eval_std_reward': np.std(rewards),
        'eval_mean_length': np.mean(lengths),
        'eval_mean_coverage': np.mean(coverages) if coverages else 0
    }

def train(self):

    print(f"Starting training for {self.total_steps} steps...")

    global_step = 0
    iteration = 0
```

```
while global_step < self.total_steps:
    iteration += 1

    rollout_stats = self.collect_rollout()
    global_step += self.rollout_length

    update_stats = self.agent.update()

    log_entry = {
        'step': global_step,
        'iteration': iteration,
        **rollout_stats,
        **update_stats
    }
    self.training_log.append(log_entry)

    # Print progress
    if iteration % 10 == 0:
        print(f"Step {global_step:>8} | "
              f"Reward: {rollout_stats['mean_reward']:>7.2f} | "
              f"Coverage: {rollout_stats['mean_coverage']:>5.1f} | "
              f"Policy Loss: {update_stats['policy_loss']:.4f} | "
              f"Entropy: {update_stats['entropy']:.4f}")

    # Periodic evaluation
    if global_step % self.eval_frequency == 0:
        eval_stats = self.evaluate(n_episodes=10)
        print(f"\n=== Evaluation at step {global_step} ===")
```

```
                print(f"Mean Reward: {eval_stats['eval_mean_reward']:.2f} +/-
{eval_stats['eval_std_reward']:.2f}")
                print(f"Mean Coverage:
{eval_stats['eval_mean_coverage']:.1f}%\n")

                # Save best model
                if eval_stats['eval_mean_reward'] > self.best_reward:
                    self.best_reward = eval_stats['eval_mean_reward']
                    self.agent.save('checkpoints/best_model.pt')
                    print(f"New best model saved! Reward: {self.best_reward:.2f}")

            # Checkpoint
            if global_step % self.checkpoint_frequency == 0:
                self.agent.save(f'checkpoints/model_step_{global_step}.pt')

        print("Training completed!")
        self._save_training_log()

    def _save_training_log(self):
        with open('training_log.csv', 'w', newline='') as f:
            writer = csv.DictWriter(f, fieldnames=self.training_log[0].keys())
            writer.writeheader()
            writer.writerows(self.training_log)
```

A.5 Файл AnomalyDetection.py

```
import numpy as np
from collections import deque
from typing import Dict, List
from dataclasses import dataclass
```

```
from enum import Enum

class AnomalyType(Enum):
    # Anomaly types (Section 5.5)
    INVISIBLE_WALL = "invisible_wall"
    STUCK_ZONE = "stuck_zone"
    INCORRECT_SPAWN = "incorrect_spawn"
    CYCLIC_PATTERN = "cyclic_pattern"

@dataclass
class Anomaly:
    # Structure for storing anomaly information
    type: AnomalyType
    position: tuple
    step: int
    duration: int
    description: str



class AnomalyDetector:

    def __init__(self,
                 position_history_size: int = 100,
                 action_history_size: int = 200,
                 stuck_threshold: int = 50,
                 cycle_min_length: int = 5,
                 attack_without_reward_threshold: int = 20):

        self.position_history = deque(maxlen=position_history_size)
        self.action_history = deque(maxlen=action_history_size)
```

```
        self.reward_history = deque(maxlen=action_history_size)

        # Detection parameters
        self.stuck_threshold = stuck_threshold
        self.cycle_min_length = cycle_min_length
        self.attack_without_reward_threshold = attack_without_reward_threshold

        self.anomalies: List[Anomaly] = []
        self.current_step = 0

        self.steps_without_movement = 0
        self.last_position = None
        self.attack_without_reward_count = 0

    def update(self, position: tuple, action: int, reward: float):
        self.current_step += 1

        self.position_history.append(position)
        self.action_history.append(action)
        self.reward_history.append(reward)

        # Check different anomaly types
        self._check_invisible_wall(position)
        self._check_cyclic_pattern()
        self._check_incorrect_spawn(action, reward)

    def _check_invisible_wall(self, current_position: tuple):

        if self.last_position is None:
```

```
            self.last_position = current_position
            return

        if current_position == self.last_position:
            self.steps_without_movement += 1

            if self.steps_without_movement >= self.stuck_threshold:
                anomaly = Anomaly(
                    type=AnomalyType.INVISIBLE_WALL,
                    position=current_position,
                    step=self.current_step,
                    duration=self.steps_without_movement,
                    description=f"Agent stuck at {current_position} for
{self.steps_without_movement} steps"
                )
                self.anomalies.append(anomaly)
                self.steps_without_movement = 0
        else:
            self.steps_without_movement = 0

        self.last_position = current_position

    def _check_cyclic_pattern(self):

        if len(self.action_history) < self.cycle_min_length * 3:
            return

        actions = list(self.action_history)

        # Search for repeating patterns
```

```
            for cycle_len in range(self.cycle_min_length, len(actions) // 3):
                pattern = actions[-cycle_len:]
                matches = 0

                for i in range(len(actions) - cycle_len, 0, -cycle_len):
                    if actions[i:i+cycle_len] == pattern:
                        matches += 1
                    else:
                        break

                if matches >= 3:  # Pattern repeated 3+ times
                    # Check if position is also cyclic
                    if self._is_position_cyclic(cycle_len):
                        current_pos = self.position_history[-1] if self.position_history else (0, 0)
                        anomaly = Anomaly(
                            type=AnomalyType.CYCLIC_PATTERN,
                            position=current_pos,
                            step=self.current_step,
                            duration=cycle_len * matches,
                            description=f"Cyclic action pattern detected: length={cycle_len}, repeats={matches}"
                        )
                        self.anomalies.append(anomaly)
                        return

        def _is_position_cyclic(self, cycle_len: int) -> bool:
            # Check if positions are cyclic
            if len(self.position_history) < cycle_len * 2:
                return False
```

```
            positions = list(self.position_history)[-cycle_len * 2:]

            # Compute position variance
            x_coords = [p[0] for p in positions]
            y_coords = [p[1] for p in positions]

            variance = np.var(x_coords) + np.var(y_coords)

            # Low variance = agent "stuttering" in place
            return variance < 100

        def _check_incorrect_spawn(self, action: int, reward: float):

            ATTACK_ACTION = 4  # Assuming 4 = ATTACK
            KILL_REWARD = 1.0

            if action == ATTACK_ACTION:
                if reward < KILL_REWARD * 0.5:  # Did not receive kill reward
                    self.attack_without_reward_count += 1
                else:
                    self.attack_without_reward_count = 0

                if self.attack_without_reward_count >= self.attack_without_reward_threshold:
                    current_pos = self.position_history[-1] if self.position_history else (0, 0)
                    anomaly = Anomaly(
                        type=AnomalyType.INCORRECT_SPAWN,
                        position=current_pos,
```

```
                    step=self.current_step,
                    duration=self.attack_without_reward_count,
                    description=f"Multiple attacks ({self.attack_without_reward_count}) without kill reward"
                )
                self.anomalies.append(anomaly)
                self.attack_without_reward_count = 0
            else:
                # Reset if not attack action
                if self.attack_without_reward_count > 0:
                    self.attack_without_reward_count = max(0, self.attack_without_reward_count - 1)

        def get_anomalies(self) -> List[Anomaly]:
            # Get list of detected anomalies
            return self.anomalies

        def get_anomaly_summary(self) -> Dict[str, int]:
            #Statistics by anomaly types
            summary = {t.value: 0 for t in AnomalyType}
            for anomaly in self.anomalies:
                summary[anomaly.type.value] += 1
            return summary

        def generate_report(self) -> str:
            # Generate text report about anomalies
            report = ["=" * 50]
            report.append("ANOMALY DETECTION REPORT")
            report.append("=" * 50)
            report.append(f"\nTotal anomalies detected: {len(self.anomalies)}")
```

```
report.append(f"\nBreakdown by type:")

summary = self.get_anomaly_summary()
for anomaly_type, count in summary.items():
    report.append(f"  - {anomaly_type}: {count}")

report.append("\n" + "-" * 50)
report.append("DETAILED ANOMALY LOG:")
report.append("-" * 50)

for i, anomaly in enumerate(self.anomalies, 1):
    report.append(f"\n[{i}] {anomaly.type.value}")
    report.append(f"    Step: {anomaly.step}")
    report.append(f"    Position: {anomaly.position}")
    report.append(f"    Duration: {anomaly.duration}")
    report.append(f"    Description: {anomaly.description}")

  return "\n".join(report)
```

ДОДАТОК Б

Слайди презентації

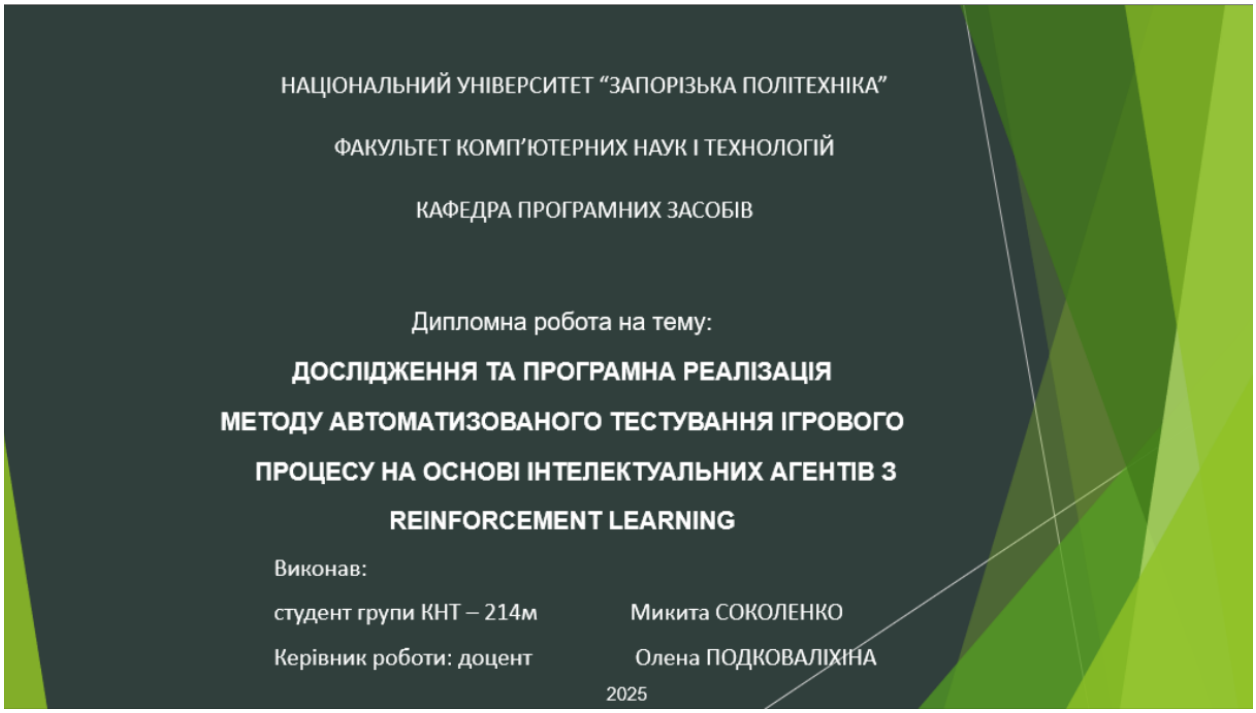


Рисунок Б.1 – Слайд № 1

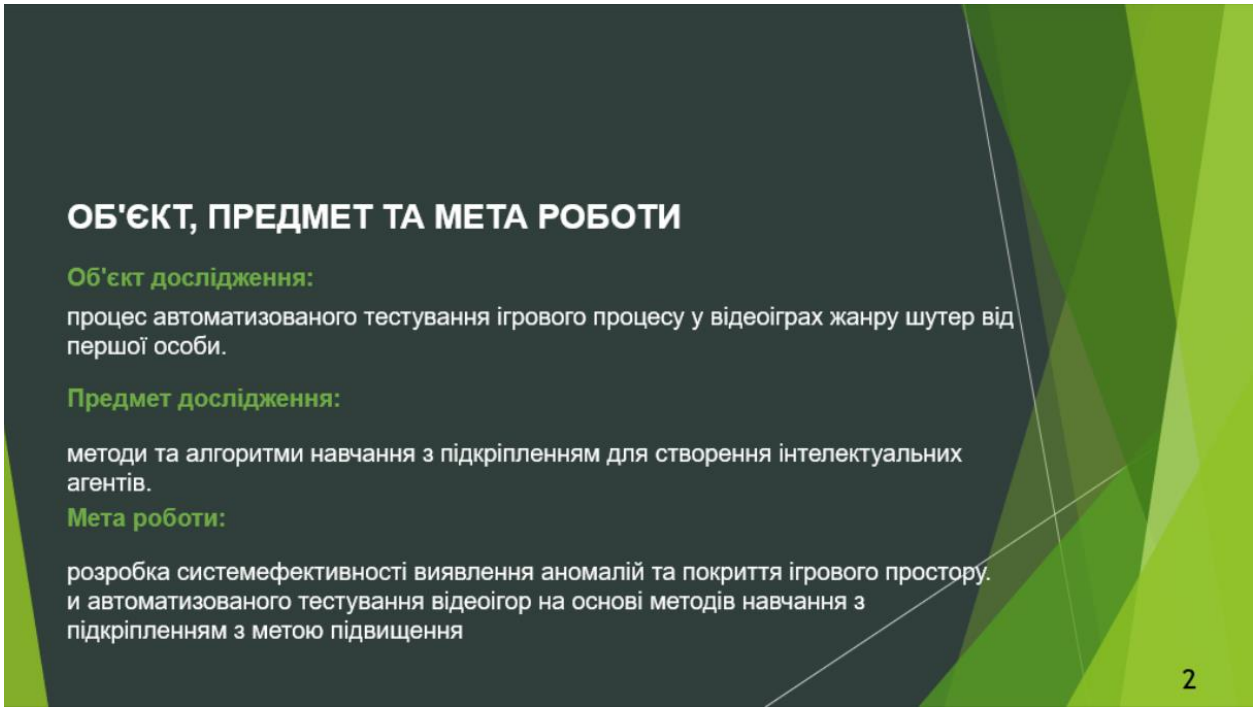


Рисунок Б.2 – Слайд № 2

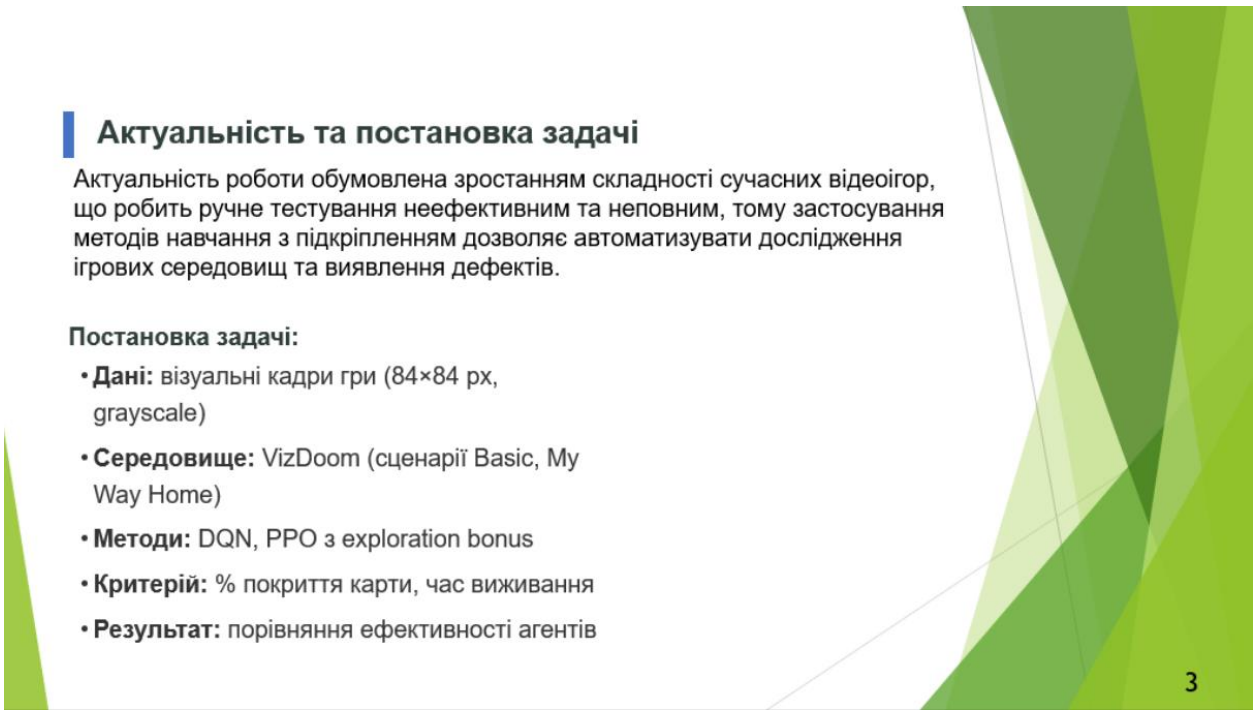


Рисунок Б.3 – Слайд № 3

Рисунок Б.4 – Слайд № 4

Обґрунтування вибору PPO

Метод	Суть методу	Переваги для тестування ігор
DQN	Q-learning з replay buffer та target network	Стабільність, простота реалізації
A3C	Асинхронне навчання на багатьох середовищах	Швидкість збіжності
PPO	Clipped surrogate objective, on-policy	Баланс exploration/exploitation, стабільність
NEAT	Еволюція топології нейронної мережі	Пошук нових архітектур

5

Рисунок Б.5 – Слайд № 5

Схема системи автоматизованого тестування

Stack:
- Python 3.11 + PyTorch 2.0
- VizDoom API (Gym interface)

6

Рисунок Б.6 – Слайд № 6

Фрагмент коду (Конфігурація параметрів навчання агента)

```yaml
# VizDoom environment parameters
environment:
  scenario_path: "scenarios/defend_the_center.cfg"
  frame_skip: 4
  resolution: [84, 84]
  stack_frames: 4
  grayscale: true

# PPO hyperparameters (Table 4.1)
ppo:
  learning_rate: 0.0003
  gamma: 0.99
  gae_lambda: 0.95
  clip_epsilon: 0.2
  rollout_length: 2048
  batch_size: 64
  n_epochs: 4
  entropy_coef: 0.01
  value_coef: 0.5

# Reward function parameters (Section 2.5)
reward:
  kill_reward: 1.0            # enemy kill reward
  health_pickup: 0.5          # health pickup reward
  ammo_pickup: 0.3            # ammo pickup reward
  death_penalty: -1.0         # death penalty
  living_reward: 0.01         # survival reward per step
  exploration_eta: 0.1        # η - base exploration coefficient
  time_penalty: -0.001        # time penalty per step
  idle_penalty: -0.1          # idle penalty (10+ steps)
  alpha: 1.0                  # exploration coefficient
  beta: 1.0                   # time penalty coefficient

# Training parameters
training:
  total_steps: 1000000        # 10⁶ steps
  eval_episodes: 100          # test episodes
  eval_frequency: 10000       # evaluation frequency
  checkpoint_frequency: 50000
  seed: 42
```

7

Рисунок Б.7 – Слайд № 7

Функція винагороди

$$R(s_t, a_t, s_{t+1}) = Rgame + \alpha \cdot Rexplore + \beta \cdot Rtime$$

Rgame — Ігрова

Знищення ворога: +1.0
Аптечка: +0.5
Смерть: -1.0

Rexplore — Дослідження

$R = \eta / \sqrt{N(c)}$
$\eta = 0.1$, $N(c)$ — лічильник

Rtime — Штраф

За крок: -0.001
Простій (10 кроків): -0.1

Ключова інновація: Exploration bonus стимулює дослідження нових областей для повного тестування

8

Рисунок Б.8 – Слайд № 8

Середовище VizDoom

Характеристики платформи:

Вхід: 84×84 пікселів, 4 кадри (frame stack)

Дії: 3-8 дискретних дій

API: OpenAI Gym-сумісний інтерфейс

Метрики: покриття карти, винагорода, час виживання

Сценарії: Basic, Defend the Center, My Way Home

9

Рисунок Б.9 – Слайд № 9

Фрагмент коду (Реалізація алгоритму PPO)

```
def __init__(self, buffer_size: int = 2048, obs_shape: tuple = (4,
84, 84),
             n_envs: int = 1, device: str = 'cuda'):
        self.buffer_size = buffer_size
        self.obs_shape = obs_shape
        self.n_envs = n_envs
        self.device = device
        self.ptr = 0
        self.full = False

        # Initialize buffers
        self.states = np.zeros((buffer_size, *obs_shape),
dtype=np.float32)
        self.actions = np.zeros(buffer_size, dtype=np.int64)
        self.rewards = np.zeros(buffer_size, dtype=np.float32)
        self.values = np.zeros(buffer_size, dtype=np.float32)
        self.log_probs = np.zeros(buffer_size, dtype=np.float32)
        self.dones = np.zeros(buffer_size, dtype=np.float32)
        self.advantages = np.zeros(buffer_size, dtype=np.float32)
        self.returns = np.zeros(buffer_size, dtype=np.float32)

    def add(self, state, action, reward, value, log_prob, done):
        #Add transition to buffer
        self.states[self.ptr] = state
        self.actions[self.ptr] = action
        self.rewards[self.ptr] = reward
        self.values[self.ptr] = value
        self.log_probs[self.ptr] = log_prob
        self.dones[self.ptr] = done

        self.ptr += 1
        if self.ptr >= self.buffer_size:
            self.full = True
            self.ptr = 0
```

```
    def compute_gae(self, last_value: float, gamma: float = 0.99,
                    gae_lambda: float = 0.95):

        gae = 0
        for t in reversed(range(self.buffer_size)):
            if t == self.buffer_size - 1:
                next_value = last_value
                next_non_terminal = 1.0 - self.dones[t]
            else:
                next_value = self.values[t + 1]
                next_non_terminal = 1.0 - self.dones[t]

            # δ_t = r_t + γ·V(s_{t+1}) - V(s_t)
            delta = self.rewards[t] + gamma * next_value *
next_non_terminal - self.values[t]

            # A_t = δ_t + γλ·A_{t+1}
            gae = delta + gamma * gae_lambda * next_non_terminal *
gae
            self.advantages[t] = gae

        # Returns for value function
        self.returns = self.advantages + self.values

    def get_batches(self, batch_size: int = 64):
        # Mini-batch generator (M=64 from Table 4.1)
        indices = np.random.permutation(self.buffer_size)

        for start in range(0, self.buffer_size, batch_size):
            end = start + batch_size
            batch_indices = indices[start:end]

            yield (
torch.FloatTensor(self.states[batch_indices]).to(self.device),
```

10

Рисунок Б.10 – Слайд № 10

Гіперпараметри алгоритму PPO

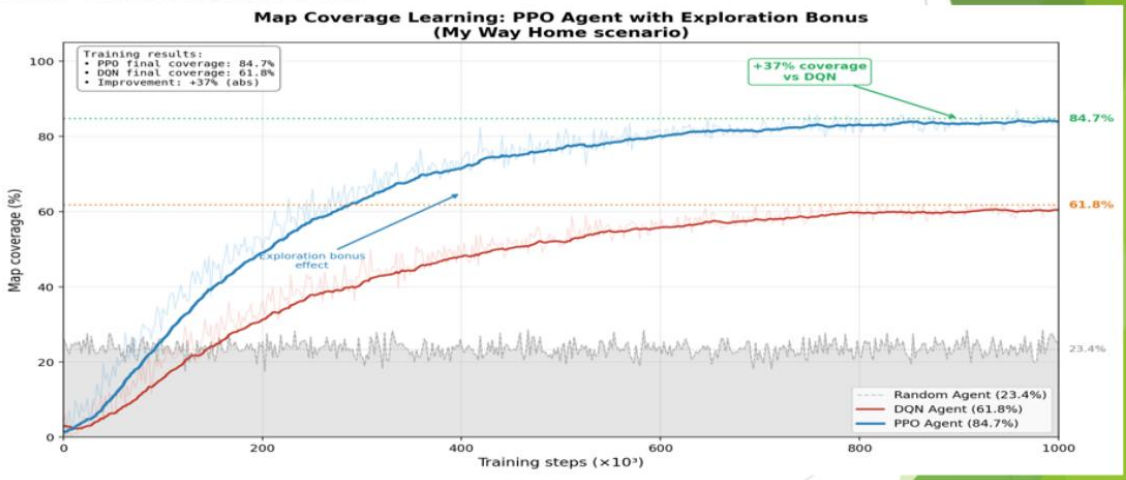
Параметр	Позначення	Значення	Опис
Коефіцієнт навчання	α	$3 \cdot 10^{-4}$	Швидкість оновлення ваг
Коефіцієнт дисконтування	γ	0.99	Вага майбутніх винагород
GAE параметр	λ	0.95	Баланс bias-variance
Clip epsilon	ε	0.2	Обмеження зміни policy
Кроків на оновлення	T	2048	Розмір batch
Кількість епох	K	10	Ітерацій оптимізації
Коефіцієнт ентропії	c_2	0.01	Стимуляція exploration

11

Рисунок Б.11 – Слайд № 11

Динаміка покриття карти

Рисунок 5.1 — Порівняння агентів (сценарій My Way Home)



12

Рисунок Б.12 – Слайд № 12

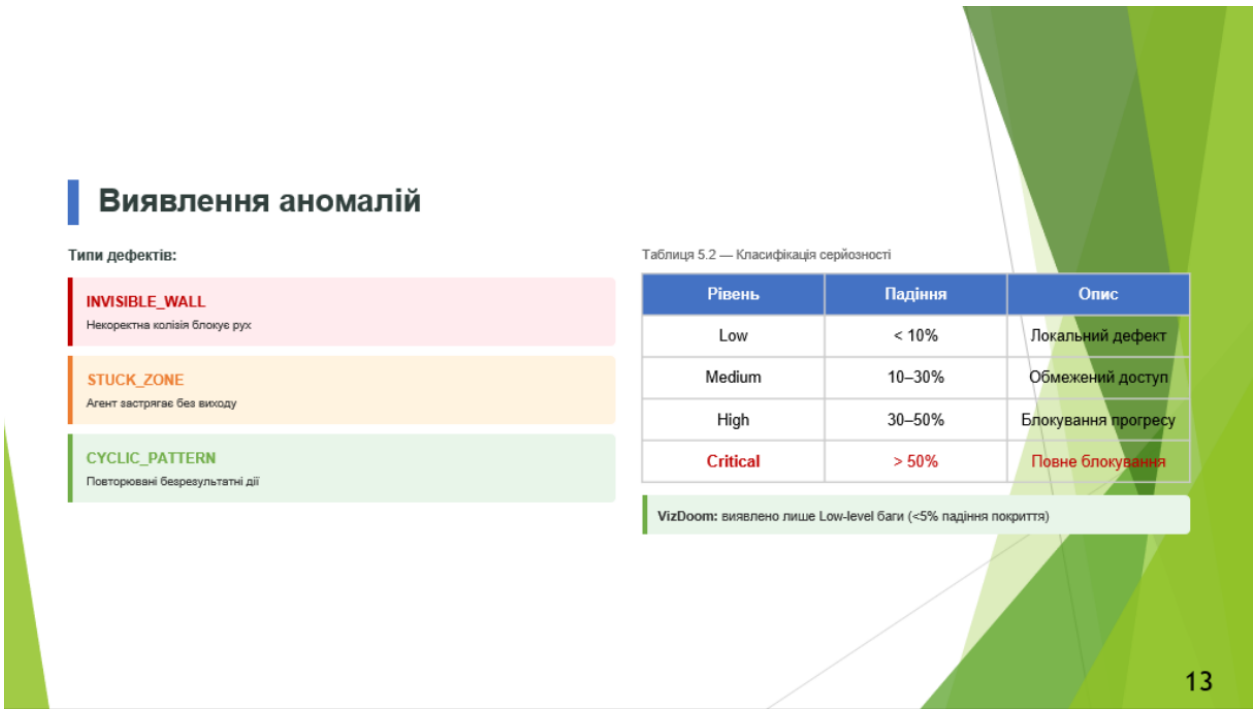


Рисунок Б.13 – Слайд № 13

Рисунок Б.14 – Слайд № 14

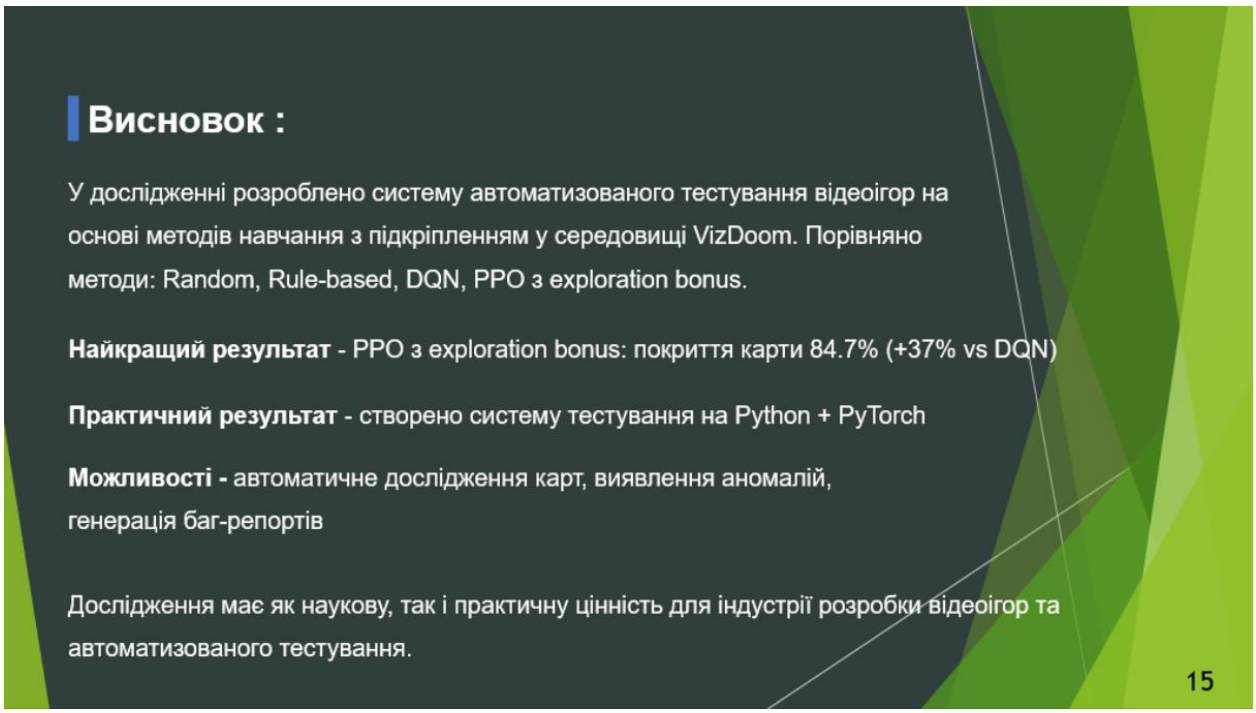


Рисунок Б.15 – Слайд № 15