Антон Андрійович БУТЕНКО[1],
здобувач другого (магістерського) рівня вищої освіти, кафедра інформаційних та комунікаційних технологій, ORCID: https://orcid.org/0009-0009-4236-5853
Олексій Дмитрович ГОЛУБЕНКО[2],
здобувач першого (бакалаврського) рівня вищої освіти, факультет фізики, астрономії та прикладної інформатики, ORCID: https://orcid.org/0009-0004-0822-7178
Олександр Анатолійович КЛИМЕНКО[1],
викладач, кафедра інформаційних та комунікаційних технологій, ORCID: https://orcid.org/0009-0009-4142-8840
Дмитро Володимирович ШАНДИБА[1],
викладач, кафедра інформаційних та комунікаційних технологій, ORCID: https://orcid.org/0000-0002-2729-3179

[1] *Заклад вищої освіти «Міжнародний науково-технічний університет імені академіка Юрія Бугая»*
[2] *Ягеллонський університет*

Прийняття: 03/03/2026
Рецензія: 20/03/2026
Публікація: 09/07/2026

DOI: https://doi.org/10.53920/ITS-2026-1-1

УДК
004.853

ISSN 2786-7226 (print)
ISSN 2786-7234 (online)

ПОРІВНЯННЯ МЕТОДІВ МАШИННОГО НАВЧАННЯ ТА ТРАДИЦІЙНИХ АЛГОРИТМІВ У ЗАДАЧАХ НАВІГАЦІЇ ТА ПОВЕДІНКИ NPC

В межах даного дослідження здійснено апробацію інструментарію Unity ML-Agents, що реалізує парадигму навчання з підкріпленням (Reinforcement Learning), як методологічної основи для моделювання автономної поведінки неігрових персонажів (NPC). Зазначений підхід розглядається як концептуально відмінна та адаптивна альтернатива класичним архітектурам керування, зокрема деревам поведінки (Behavior Trees) та скінченним автоматам (State Machines), оскільки він забезпечує підвищений рівень нелінійності, варіативності та когнітивної складності агентної поведінки.

Емпірична частина роботи реалізована у середовищі Unity на прикладі гри «Змійка», що виступає тестовим полігоном для порівняльного аналізу. У дослідженні здійснено зіставлення результатів функціонування NPC, керованих традиційними алгоритмами, із результатами, отриманими за допомогою ML-Agents, що використовували алгоритми Proximal Policy Optimization (PPO) та його модифікацію з рекурентними мережами довготривалої пам'яті (PPO+LSTM). Порівняльні таблиці результатів дозволяють здійснити кількісну та якісну оцінку ефективності

запропонованого підходу, а також окреслити його потенціал у контексті подальших досліджень у сфері інтелектуальних агентних систем.

Наукова новизна дослідження полягає у верифікації гіпотези про доцільність використання методів навчання з підкріпленням як альтернативи традиційним архітектурам керування, що підтверджено емпіричними результатами, отриманими у тестовому середовищі гри «Змійка». Практична значущість роботи визначається тим, що запропоновані рекомендації та результати порівняльного аналізу можуть бути безпосередньо інтегровані у процес розробки сучасних ігрових продуктів, сприяючи підвищенню рівня автономності, адаптивності та інтелектуальної складності поведінки NPC.

У висновках роботи сформульовано комплекс практичних рекомендацій, адресованих розробникам ігор та гейм-дизайнерам, які перебувають у пошуку альтернативних підходів до організації та керування поведінкою неігрових персонажів у власних проєктах. Окрім цього, здійснено систематизований порівняльний аналіз інтеграційних можливостей та обмежень різних архітектурних рішень, зокрема інструментарію Unity ML-Agents та дерев поведінки (Behavior Trees), що дозволяє не лише окреслити їхні функціональні переваги й недоліки у контексті практичного застосування, але й визначити перспективні напрями подальшої оптимізації агентних систем у сфері інтерактивних цифрових середовищ.

Таким чином, проведене дослідження не лише розширює теоретичні уявлення про можливості застосування алгоритмів оптимізації політики у сфері інтерактивних симуляцій, але й створює підґрунтя для подальших робіт, спрямованих на розробку масштабованих моделей агентної поведінки, здатних забезпечити якісно новий рівень когнітивної адаптивності у цифрових середовищах.

Ключові слова: *Керування поведінкою, ML-Agents, ігри, NPC, Reinforcement Learning, RL, Unity, PPO, LSTM, Behavior Trees, State Machines, Глибоке навчання, Навчання з підкріпленням.*

Anton BUTENKO, Oleksii HOLUBENKO, Oleksandr KLYMENKO, Dmytro SHANDYBA

COMPARISON OF MACHINE LEARNING METHODS AND TRADITIONAL ALGORITHMS IN NPC NAVIGATION AND BEHAVIOR TASKS

As part of this study, the Unity ML-Agents toolkit, which implements the reinforcement learning paradigm, was tested as a methodological foundation for modeling the autonomous behavior of non-player characters (NPCs). This approach is considered a conceptually distinct and adaptive alternative to classical control

architectures, particularly behavior trees and state machines, as it provides a higher level of nonlinearity, variability, and cognitive complexity in agent behavior.

The empirical part of the study was implemented in the Unity environment using the game "Snake" as a test case for comparative analysis. The study compares the performance of NPCs controlled by traditional algorithms with the results obtained using ML-Agents that employed Proximal Policy Optimization (PPO) and its modification with Long Short-Term Memory (LSTM) networks (PPO+LSTM). Comparative tables of results allow for a quantitative and qualitative assessment of the effectiveness of the proposed approach, as well as an outline of its potential in the context of further research in the field of intelligent agent systems.

The scientific novelty of this study lies in the verification of the hypothesis regarding the feasibility of using reinforcement learning methods as an alternative to traditional control architectures, as confirmed by empirical results obtained in the test environment of the "Snake" game. The practical significance of this work lies in the fact that the proposed recommendations and the results of the comparative analysis can be directly integrated into the development process of modern game products, contributing to an increase in the level of autonomy, adaptability, and intellectual complexity of NPC behavior.

The conclusions of this work formulate a set of practical recommendations addressed to game developers and game designers who are seeking alternative approaches to organizing and controlling the behavior of non-player characters in their own projects. In addition, a systematic comparative analysis was conducted of the integration capabilities and limitations of various architectural solutions, specifically the Unity ML-Agents toolkit and behavior trees (Behavior Trees), which allows not only to outline their functional advantages and disadvantages in the context of practical application but also to identify promising directions for the further optimization of agent-based systems in the field of interactive digital environments.

Thus, this study not only expands theoretical understanding of the potential applications of policy optimization algorithms in the field of interactive simulations, but also lays the groundwork for future research aimed at developing scalable models of agent behavior capable of achieving a qualitatively new level of cognitive adaptability in digital environments.

Keywords: *Behavior control, ML-Agents, games, NPC, Reinforcement Learning, RL, Unity, PPO, LSTM, Behavior Trees, State Machines, Deep Learning, Reinforcement Learning.*

Постановка проблеми. У сучасному дискурсі досліджень цифрових інтерактивних систем, незалежно від конкретної апаратної платформи чи жанрової парадигми, однією з фундаментальних проблем, що постає перед розробниками, є конструювання алгоритмічних моделей, здатних

забезпечувати когерентну, раціонально обґрунтовану та контекстуально релевантну поведінку ігрових агентів. Йдеться про ті суб'єкти ігрового процесу, які не перебувають під безпосереднім контролем користувача, але водночас виконують функції супротивників, союзників або допоміжних компонентів, що реагують на дії гравця, інших агентів та на динаміку середовища.

У науковій та професійній літературі такі агенти позначаються терміном *NPC* (*Non-Playable Character*), що відображає їхню онтологічну специфіку як об'єктів, поведінка яких детермінується не волею користувача, а попередньо закладеними у програмний код правилами та логічними структурами.

Механізми прийняття рішень NPC, як правило, реалізуються у вигляді реактивних патернів, що актуалізуються у відповідь на певні тригери: отримання шкоди від гравця чи інших агентів, необхідність здійснення підтримки користувача у випадку загрози його життєвим ресурсам, супровід до визначених просторових локацій, забезпечення доступу до критично важливих артефактів, включно з лікувальними засобами та зброєю. Таким чином, NPC функціонують як інструмент підтримання ігрової динаміки, забезпечуючи баланс між інтерактивністю та системною передбачуваністю.

Звичайні підходи розробки системи прийняття рішень для NPC, які в основному використовують стеження за оточенням, стеження за діями інших NPC, аналіз поведінки гравця, реакція на дії гравця та інших NPC зазвичай базуються на давно зарекомендованих себе алгоритмах та підходах проектування систем прийняття рішень для NPC, таких як: Кінцеві автомати (Finite State Machines), Дерево рішень (Behaviour Tree), Системи аналізу користі (Utility Systems), або Планування дій (Goal-Oriented Action Planning).

Зазвичай, ці підходи працюють добре і не викликають ніяких нарікань, проте через те, що вони використовують заздалегідь задані алгоритми дій та реакцій, наприклад, якщо гравець підходить на достатню відстань до ворога - ворог може його побачити та почати переслідувати, чи атакувати, чи втікати (в залежності від рівня свого здоров'я), - це вимагає від програмістів доповнювати алгоритмічну базу реакцій і дій кожен раз, коли в гру додається нова механіка, новий тип персонажу (character), або новий тип оточення (наприклад, море).

Цю проблему потенційно можна виправити за допомогою адаптивної поведінки та інтеграції ШІ, який міг би адаптуватися до оточення, нової поведінки NPC або нових механік без необхідності дописувати або переписувати існуючі алгоритми поведінки NPC.

Також, інтеграція ШІ потенційно може значно покращити варіативність проходження гри, яке може бути досягнуто за допомогою математичного навчання та потенційно нової, заздалегідь не закладеної механіки реакції NPC на дії гравця.

Імітаційна поведінка може бути адаптована для імітації людської поведінки, що може покращити якість ботів у грі, особливо для мультиплеерних ігор, коли на початковому етапі релізу гри потрібна імітація великої кількості гравців та достатня кількість опонентів для реальних гравців.

Для подібних цілей у Unity-іграх може бути використаний ML-Agents (Unity Machine Learning Agents Toolkit) - це набір інструментів з відкритим вихідним кодом, який дозволяє розробникам створювати оточення, де ШІ-агенти можуть навчатися складним поведінкам за допомогою навчання з підкріпленням (Reinforcement Learning), ідеально підходящим для створення реалістичної поведінки NPC та симуляцій[10].

Аналіз останніх досліджень і публікацій. Архітектури прийняття рішень на основі правил історично домінували в реалізації поведінки неігрових персонажів (NPC) у цифрових іграх завдяки їхньому детермінізму, інтерпретованості та низьким обчислювальним витратам. Серед цих підходів найбільш широко впровадженими парадигмами є скінченні автомати (FSM), дерева поведінки (BT) та цілеспрямоване планування дій (GOAP).

Скінченні автомати є однією з найперших і найпростіших моделей керування поведінкою NPC. FSM представляє агента як сукупність дискретних станів із чіткими переходами, що активуються зумовленими ситуаціями. Скінченні автомати є обчислювально ефективними та легкими в реалізації; проте вони мають низьку масштабованість, оскільки кількість станів і переходів зростає комбінаторно з ускладненням поведінки. Це явище, яке часто називають «вибухом станів», суттєво обмежує застосування FSM у сучасних іграх із багатим простором взаємодії та динамічним середовищем [1].

Дерева поведінки були впроваджені як більш модульна та ієрархічна альтернатива FSM з метою покращення масштабованості та зручності підтримки. BT декомпонують поведінку на багаторазові вузли, організовані в деревоподібну структуру, що зазвичай складається з вузлів керування потоком (таких як селектори та послідовності) і листкових вузлів, що представляють дії або умови. Така архітектура дозволяє дизайнерам проєктувати та втілювати складну поведінку через композицію, зберігаючи при цьому читабельність і можливість налагодження. Дерева поведінки отримали широке розповсюдження в комерційних ігрових рушіях завдяки їхній зручності для дизайнерів та детермінованій семантиці виконання [2].

Станом на 2026 рік, дерева поведінки залишаються фундаментально системами на основі правил, що покладаються на вручну створену логіку. Як наслідок, проєкти та ігри, що використовують вищевказану модель цього типу, демонструють обмежену здатність до адаптації в непередбачуваних ситуаціях і потребують постійного ручного оновлення при впровадженні нових механік або характеристик середовища.

Цілеспрямоване планування дій (GOAP) розширює підходи на основі правил, дозволяючи агентам динамічно генерувати послідовності дій для досягнення чітко визначених цілей. GOAP моделює поведінку NPC як завдання типу планування, де дії визначаються передумовами та ефектами, а агент шукає послідовність дій, яка трансформує поточний стан світу в цільовий. Порівняно з FSM та BT, GOAP пропонує більшу гнучкість і реактивність, особливо в середовищах із кількома конкуруючими цілями. Однак системи GOAP все ще залежать від авторських визначень дій та символічних репрезентацій станів, що обмежує їхню здатність до генералізації поза межами попередньо визначеної області планування [3].

Попри практичний успіх, усі три підходи мають спільне обмеження: вони кодують інтелект експліцитно через правила, визначені дизайнером, а не шляхом вивчення оптимальної поведінки на основі досвіду. Це обмеження зумовило зростання інтересу до методів моделювання поведінки NPC, що базуються на даних та навчанні.

Отже, навчання з підкріпленням (RL) пропонує фундаментально іншу парадигму проєктування поведінки агентів, розглядаючи прийняття рішень як завдання оптимізації довгострокової сукупної винагороди. Замість того, щоб покладатися на створені вручну правила, RL-агент вивчає стратегію через взаємодію із середовищем, отримуючи зворотний зв'язок у формі скалярних винагород. Ця парадигма природно узгоджується з ігровими середовищами, які зазвичай мають чітко визначені простори станів, дії та сигнали винагороди.

Ранні застосування RL в іграх зосереджувалися на невеликих, повністю спостережуваних середовищах з обмеженим простором станів, таких як настільні ігри та прості аркадні завдання. Відродження інтересу до RL в іграх значною мірою було зумовлене успіхами глибокого навчання (deep learning), що дозволило агентам апроксимувати функції цінності та стратегії безпосередньо з високовимірних сенсорних вхідних даних. Визначним внеском у цій галузі стала розробка глибоких Q-мереж (DQN), які продемонстрували результативність на рівні людини в багатьох іграх Atari 2600, використовуючи необроблені піксельні дані [5]. Подальші роботи розширили ці ідеї на складніші домени, включаючи стратегії в реальному часі та багатокористувацькі середовища.

Таким чином, теорія навчання з підкріпленням пропонує нормативне обґрунтування того, як інтелектуальні агенти можуть оптимізувати стратегії управління середовищем. Цей підхід має глибоке коріння у психологічних та нейробіологічних теоріях поведінки тварин.

Однак успішне застосування RL у сценаріях, що за складністю наближаються до реального світу, ставить перед агентами складне завдання: самостійно формувати ефективні репрезентації середовища на основі високовимірних сенсорних даних. Це необхідно для того, щоб екстраполювати минулий досвід на нові, раніше невідомі ситуації.

Примітно, що люди та інші живі організми розв'язують цю проблему завдяки гармонійному поєднанню механізмів навчання з підкріпленням та ієрархічних систем обробки сенсорної інформації. Підтвердженням цього є значний обсяг нейрофізіологічних даних, які демонструють вражаючий паралелізм між фазними сигналами дофамінергічних нейронів та алгоритмами навчання за методом часових різниць (Temporal Difference Learning).

Хоча RL-агенти досягли певних успіхів у різних галузях, їх застосування тривалий час обмежувалося доменами, де корисні ознаки можна було виділити вручну, або середовищами з повністю спостережуваним низьковимірним простором станів [5].

У контексті ігрового штучного інтелекту навчання з підкріпленням досліджувалося для таких завдань, як навігація, бойова поведінка, управління ресурсами та моделювання опонента. Порівняно зі скриптовими агентами, RL-агенти можуть виявляти емерджентну поведінку, яка не була явно запрограмована, що потенційно підвищує реіграбельність і сприйняття інтелектуальності. Проте ці переваги досягаються ціною підвищеної обчислювальної складності, довших циклів розробки через вимоги до навчання та зниженої інтерпретованості вивчених стратегій.

Сучасні методи градієнта стратегії, такі як Proximal Policy Optimization (PPO), стали особливо популярними для ігрових додатків завдяки стабільності навчання та відносній робастності до вибору гіперпараметрів. PPO було широко впроваджено в ігрові фреймворки, зокрема Unity ML-Agents, оскільки цей метод забезпечує практичний баланс між продуктивністю та складністю реалізації. Розширення з використанням рекурентних нейронних мереж, таких як шари довгої короткострокової пам'яті (LSTM), додатково дозволяють RL-агентам діяти в частково спостережуваних середовищах шляхом підтримання внутрішніх представлень стану [6].

Окремо варто відзначити середовища «сіткового світу» (Grid-world), які є канонічним класом бенчмарків у дослідженнях навчання з підкріпленням. У таких середовищах агент пересувається дискретним простором на

основі сітки, взаємодіючи з перешкодами, винагородами та термінальними станами. Попри появну простоту, сіткові світи можуть демонструвати нетривіальну динаміку, особливо за наявності таких обмежень, як часткова спостережуваність, відкладена винагорода або зростання простору станів.

Гра «Змійка» може бути формально інтерпретована як середовище сіткового світу з динамічно розширюваним простором станів, де конфігурація тіла агента еволюціонує з часом і обмежує майбутні дії. Ця характеристика відрізняє «Змійку» від статичних завдань навігації в сітці та вводить довгострокові залежності між діями та результатами. Попередні роботи продемонстрували, що класичні табличні методи RL стикаються з труднощами в таких середовищах через експоненціальне зростання простору станів, що зумовлює необхідність використання методів апроксимації функцій [7].

Методи глибокого RL були успішно застосовані до сіткових ігор з використанням згорткових нейронних мереж для кодування просторової інформації та рекурентних архітектур для фіксації часових залежностей. Дослідження показали, що рекурентні стратегії перевершують архітектури прямого поширення в середовищах, де агент не може повністю спостерігати глобальний стан, що часто трапляється в практичних ігрових сценаріях [8]. Ці результати підтверджують доцільність використання PPO у поєднанні з модулями LSTM для завдань у сіткових світах із частковою спостережуваністю та відкладеними наслідками.

Загалом, центральним питанням у дослідженнях ігрового ШІ є порівняльні переваги агентів на основі навчання та традиційних скриптових агентів. Скриптові підходи, включаючи FSM та BT, забезпечують передбачувану поведінку, низькі витрати часу на виконання та легкість налагодження. Ці властивості є особливо важливими в комерційній розробці ігор, де критичне значення мають надійність і контроль з боку дизайнера.

Натомість RL-агенти оптимізують поведінку через досвід і можуть адаптуватися до змін у середовищі без явного перепрограмування. Така адаптивність дозволяє RL-агентам виявляти стратегії, які могли бути не передбачені дизайнерами, що потенційно підвищує різноманітність поведінки та залученість гравців. Однак RL-агенти створюють виклики, пов'язані зі стабільністю навчання, відтворюваністю та пояснюваністю. Вивчені стратегії часто розглядаються як «чорні скриньки», що ускладнює гарантування конкретних поведінкових обмежень або діагностику випадків збоїв.

Якщо нейронна мережа приймає неправильне рішення, розробнику надзвичайно важко зрозуміти, чому саме це сталося, та як це виправити,

не зламавши інші аспекти поведінки. Сучасні дослідження в галузі пояснюваного ШІ (Explainable AI, XRL) намагаються вирішити це за допомогою векторів Шеплі (Shapley values) або дистиляції політики в дерева прийняття рішень, що дозволяє візуалізувати межі прийняття рішень агентом [9].

Емпіричні порівняння між RL та скриптовими агентами показали, що RL-агенти можуть перевершувати створені вручну системи в чітко визначених завданнях із прозорою структурою винагороди, особливо коли потрібне довгострокове планування. Проте скриптові агенти часто залишаються конкурентоспроможними або перевершують аналоги в сценаріях, де знання предметної області можуть бути ефективно закодовані і де бажана передбачувана поведінка. Як наслідок, дедалі більшу увагу привертають гібридні підходи, що поєднують логіку на основі правил із компонентами навчання. Такі системи прагнуть використовувати сильні сторони обох парадигм, застосовуючи скриптові структури для високорівневого контролю та RL для прийняття рішень низького рівня або адаптації.

Оглянута література підкреслює очевидний компроміс між традиційними архітектурами керування NPC на основі правил та підходами навчання з підкріпленням. FSM, дерева поведінки та GOAP забезпечують детерміновану та інтерпретовану поведінку, але їм бракує адаптивності та масштабованості в складних середовищах. Навчання з підкріпленням пропонує альтернативу на основі даних, здатну вивчати складну поведінку через взаємодію, зокрема в сіткових світах та середовищах із частковою спостережуваністю. Проте RL вносить додаткову складність і практичні виклики, що обмежують його пряме застосування в усіх контекстах розробки ігор. Дана робота базується на існуючих дослідженнях, емпірично порівнюючи ці парадигми в межах уніфікованої експериментальної структури, реалізованої в Unity, з особливим акцентом на сіткових середовищах та агентах навчання на основі PPO.

Метою даної роботи є дослідити можливості ML-Agents та впровадити навчену нейромережею модель поведінки NPC, описати складнощі при навчанні та порівняти її ефективність із традиційними методами алгоритмів прийняття рішень, дослідити їх ефективність у Unity-іграх та створити практичні рекомендації для розробників.

Виклад основного матеріалу дослідження. Для реалізації даної роботи був обраний рушій Unity[11], адже він має достатній обсяг документації, велике ком'юніті та інтегрований у Package Manager пакет ML-Agents.

Для виконання досліджень був розроблений проєкт — прототип гри типу "змійка" на Unity engine, за допомогою якого буде проводитися навчання та порівняння результатів досліджень. В грі реалізована механіка,

де гравець керує клітинкою (головою), яка переміщується раз на деякий час в напрямку, який вказав гравець, по полю 20 на 20, яке складається з клітинок іншого типу. Поле представляє собою двовимірний масив цілих чисел, де кожне число визначає тип клітинки (порожня, бонус, зайнята).

В грі присутні клітинки типу "бонус", за допомогою яких "змійка" подовжується на одну клітинку. Щоб підібрати бонус, гравцю потрібно перемістити "голову" змійки на клітинку бонусу. Одразу при досягненні клітинки з бонусом "змійка" збільшується на одну клітинку. Таким чином, "змійку" можна збільшити на n-кількість клітинок, де n — кількість бонусів на полі. Кожна нова додана клітинка до змійки буде повторювати послідовно свій рух за "головою", з кожним кроком "голови".

Таким чином, кожен інтервал часу — у наступній клітинці за напрямком руху буде створюватися нова клітинка, а остання, "хвіст", буде видалятися.

Якщо NPC перемістить "голову" змійки на клітинку, яка вже зайнята одним із сегментів змійки, гру буде завершено поразкою, нова гра автоматично починається, а у вікні консолі буде повідомлення про завершення гри.

Якщо агент збере усі клітинки з типом бонус, гру буде завершено перемогою.

Розташування бонусів на полі є фіксованим і однаковим для всіх ігрових епізодів: конфігурація поля не змінюється між партіями.

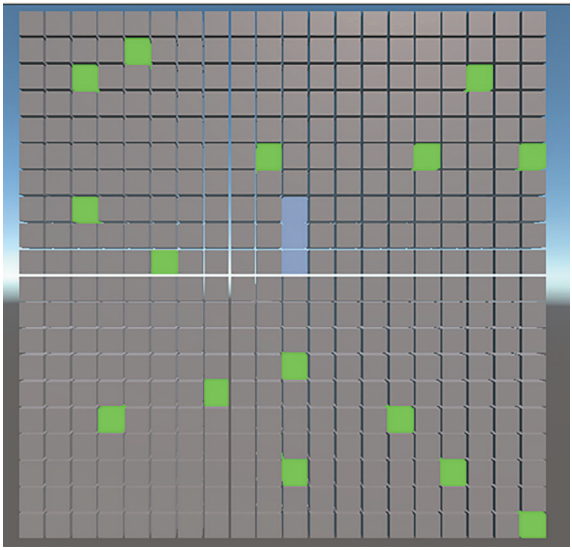

Рис. 1.
Ілюстрація проєкту гри "Змійка"
Джерело:
виконано автором

Адаптація проекта під ML-Agents

Для того, щоб зменшити час тренування моделі, в проєкті логіка та візуальна частина були розділені для можливості використання окремо. Таким чином, з'являється можливість запускати будь-яку кількість ігор, в яких вся логіка буде виконуватися, але без відображення візуальної частини. Це оптимізує використання ресурсів комп'ютера та зменшує навантаження на процесор, графічну карту та використовує менший об'єм пам'яті.

Це дає змогу запустити, наприклад, одночасно 10 ігор, в яких ML-Agents буде передавати різні Input Actions, і мати свою, окрему для кожної гри, поведінку і не витрачати зайві ресурси на створення, відображення та оновлення візуальної частини.

Проєкт "змійка" є класичною задачею, де складність середовища зростає нелінійно з кожним успішним кроком (хвіст подовжується).

Детермінований підхід

Для визначення Baseline була імплементована базова версія Behaviour Tree[12], були створені ноди для Behaviour Tree:

Селектор (Selector), або вузол пріоритетного вибору (Fallback), є одним із базових композитних вузлів (Composite Nodes) у архітектурі дерев поведінки (Behavior Trees). Його основна функція полягає у забезпеченні логічного розгалуження за принципом умовної диз'юнкції (OR).

Процес виконання (Tick) селективного вузла базується на послідовному ітеруванні дочірніх вузлів зліва направо:

Успіх (Success): Якщо будь-який дочірній вузол повертає стан Success або Running, селектор негайно припиняє подальшу ітерацію та транслює цей стан вище по дереву.

Невдача (Failure): Стан Failure від дочірнього вузла ініціює перехід до виконання наступного за пріоритетом елемента. Селектор повертає загальну невдачу лише за умови, що всі його субвузли повернули стан Failure.

Послідовність (Sequence) — це композитний вузол, який реалізує логіку AND. Він призначений для виконання серії завдань, де кожна наступна дія залежить від успішного завершення попередньої.

Виконання вузла Sequence передбачає лінійне опитування дочірніх вузлів у порядку їхнього розташування:

Успіх (Success): Вузол продовжує виконання наступного нащадка лише тоді, коли поточний повернув стан Success. Загальний стан Success повертається лише після успішного завершення всіх дочірніх вузлів у ланцюжку.

Невдача (Failure): Якщо будь-який дочірній вузол повертає Failure, Sequence негайно перериває виконання всієї гілки та транслює стан

Failure батьківському вузлу. Решта запланованих дій у цій послідовності ігнорується.

Стан виконання (Running): Якщо нащадок повертає стан Running, Sequence призупиняє ітерацію на цьому вузлі до отримання фінального результату.

MoveNode — це вузол (Node), який вибирає направлення об'єкта змійки, наприклад, коли на шляху з'явиться перешкода або границя поля, і змійка не може рухатися у заданому напрямку — спробує за допомогою Random[13] згенерувати нове направлення. Також, якщо змійка рухається в одному напрямку деякий час - нода спробує змінити напрямок через вказаний час.

Стан виконання (Running) повертається, попереду немає перешкод і змійка продовжує рух у заданому напрямку.

Стан виконання (Success) повертається, якщо була зміна напрямку руху,

IsBonusNear — вузол (Node), який визначає, чи є в межах заданої n-кількості комірок від змійки комірка з типом "бонус", і якщо є, зберігає координати комірки з найближчим у Blackboard-дереві.

Стан Success: повертається, якщо в межах заданного n є комірка з типом бонус.

Стан Failure: повертається, якщо в межах заданного радиусу n немає комірка з типом "бонус".

MoveToNearPoint — Node, який змінює направлення змійки на задану найближчу точку координат з коміркою "бонус" в Blackboard.

Стан Success повертається, якщо змійка досягає заданої точки в Blackboard.

Стан Running: повертається, якщо змійка продовжує рух до заданої у Blackboard точки координат.

Стан Failure: повертається, якщо точка координат у Blackboard стала невалідною та змійка не може продовжити рух до цієї точки.

BehaviourTree клас представляє собою root-node, який виконує всі свої нащадки безкінечно, незважаючи на повернені стани. За допомогою цих вузлів та нодів було створено циклічний алгоритм знаходження направлення руху для змійки та стратегію підбору найближчих бонусів.

Програмне забезпечення

Для реалізації проєкту було використано Unity версії 6.3 LTS, доступне на офіційному сайті Unity.com[11].

Встановлена остання на момент написання роботи версія ML Agents версії 2.0.2, встановлена через Package Manager в Unity.

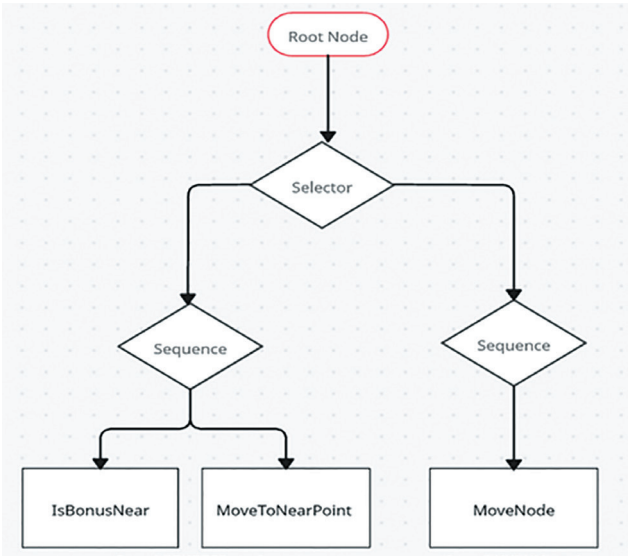

Рис. 2. Behaviour Tree – схема вибору поведінки для NPC
Джерело: виконано автором

Також, для правильної роботи потрібно розгорнути версії Python, PyTorch, Cuda суміжні між собою. Тому була обрана остання доступна на момент написання робота версія Python 3.10.

Через Python утиліту pip було інстальовано PyTorch версії 2.0.1 і всі супутні бібліотеки, за допомогою команди:

pip install torch==2.0.1 torchvision==0.15.2 torchaudio==2.0.2 --index-url https://download.pytorch.org/whl/cu118[14].

Також, для збільшення швидкості навчання, навчання проводилося на графічному процесорі (GPU), а не на центральному (CPU), була встановлена CUDA Toolkit версії 12.6.0[15].

Апаратне забезпечення

Процесор: AMD Ryzen 9 3900X 12-Core Processor, 3801 МГц, 12 ядер, 24 логічні процесори.

Відеокарта: NVIDIA GeForce RTX 2070 SUPER.

Материнська плата: ROG STRIX X570-E GAMING.

Оперативна пам’ять: 32 ГБ.

Системний диск: Samsung SSD 970 PRO 512 ГБ.

ML-Agents Toolkit

Функціонування інтелектуальних агентів у середовищі Unity ML-Agents базується на класичній парадигмі навчання з підкріпленням, де взаємодія між суб'єктом та ігровим світом реалізується через безперерв-

ний цикл обміну даними. Ключовим елементом цієї системи є простір спостережень (Observations), що формується за допомогою різноманітних сенсорів, таких як Ray Perception Sensors або Grid Sensors. Ці інструменти дозволяють агенту отримувати актуальний стан середовища у вигляді векторних або просторових даних, на основі яких нейронна мережа апроксимує найбільш вигідну стратегію поведінки.

Отримані спостереження трансформуються в простір дій (Actions), що являє собою набір дискретних і безперервних команд, які агент транслює в середовище для зміни його поточного стану. Ефективність кожної обраної дії коригується через механізм функції винагороди (Rewards).

Числовий сигнал винагороди виступає головним критерієм оптимізації: позитивні значення закріплюють цільові патерни поведінки, тоді як негативні штрафи мінімізують ймовірність повторення помилкових рішень. Таким чином, через максимізацію сумарної дисконтованої винагороди агент самостійно вибудовує складні поведінкові моделі, що є альтернативою жорстко зафіксованим алгоритмам.

Сенсори, доступні дані

Для проекту були визначені такі Observations для агента:

- Позиція змійки (X, Y).
- Направлення руху змійки (X, Y).
- Наступний тип комірки за направленням руху.
- Двомірний масив з типами комірок поля (20x20).

Вибір структури простору спостережень обумовлений необхідністю подолання проблеми інформаційної дефіцитності в умовах динамічного ігрового середовища. Для забезпечення оптимального навчання агента було застосовано гібридний підхід, що поєднує низькорозмірні векторні дані та просторові матричні репрезентації.

До складу векторних спостережень включено позицію агента (X,Y), вектор поточного напрямку руху та стан цільової комірки за траєкторією слідування.

Використання двовимірного масиву розмірністю 20×20 як вхідного тензора дозволяє реалізувати механізм просторового сприйняття. На відміну від локальних сенсорів (Ray Perception), повна топологічна карта поля надає нейронній мережі можливість ідентифікувати нелінійні патерни, такі, як створення замкнутих контурів власним тілом агента.

Reward Function

Для проекту були визначені такі Rewards для агента:

- +1 - При зборі усіх бонусів на карті та перемозі.
- -1 - При зіткненні зі своїми сегментами або виході за межі карти (поразці).

- 0.5 - При підборі бонуса на карті.
- -0.002f - Кожен крок симуляції, для того, щоб агент не зациклювався у просторі і намагався як найшвидше завершити гру.

Параметри тренування

Була задана конфігурація тренування одночасно запущених 20 копій гри змійка, одна з яких виводиться на екран монітора.

```
behaviors:
  SnakeBehavior:
    trainer_type: ppo
    hyperparameters:
      batch_size: 10
      buffer_size: 100
      learning_rate: 3.0e-4
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: constant
      epsilon_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    max_steps: 1000000
    time_horizon: 64
    summary_freq: 10000
```

Рис. 3. Конфігурація нейромережі PPO

Джерело: виконано автором

```
behaviors:
  SnakeBehavior:
    trainer_type: ppo
    hyperparameters:
      batch_size: 128
      buffer_size: 2048
      learning_rate: 3.0e-4
      beta: 5.0e-3
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2
      use_recurrent: true
      sequence_length: 64
      memory_size: 128
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    max_steps: 1000000
    time_horizon: 128
    summary_freq: 10000
```

Рис. 4. Конфігурація нейромережі PPO+LSTM

Джерело: виконано автором

Результати

Для обох моделей - PPO та PPO+LSTM - було проведено 1000000 кроків тренування.

```
[INFO] SnakeBehavior. Step: 320000. Time Elapsed: 66.390 s. Mean Reward: 6.222. Std of Reward: 1.272. Training.
[INFO] SnakeBehavior. Step: 330000. Time Elapsed: 123.033 s. Mean Reward: 6.250. Std of Reward: 1.371. Training.
[INFO] SnakeBehavior. Step: 340000. Time Elapsed: 181.571 s. Mean Reward: 6.703. Std of Reward: 1.328. Training.
[INFO] SnakeBehavior. Step: 350000. Time Elapsed: 239.936 s. Mean Reward: 6.327. Std of Reward: 1.480. Training.
[INFO] SnakeBehavior. Step: 360000. Time Elapsed: 297.027 s. Mean Reward: 7.422. Std of Reward: 1.105. Training.
[INFO] SnakeBehavior. Step: 370000. Time Elapsed: 354.556 s. Mean Reward: 6.976. Std of Reward: 1.330. Training.
[INFO] SnakeBehavior. Step: 380000. Time Elapsed: 415.048 s. Mean Reward: 5.520. Std of Reward: 2.193. Training.
[INFO] SnakeBehavior. Step: 390000. Time Elapsed: 471.856 s. Mean Reward: 3.775. Std of Reward: 0.858. Training.
[INFO] SnakeBehavior. Step: 400000. Time Elapsed: 529.471 s. Mean Reward: 6.074. Std of Reward: 1.832. Training.
[INFO] SnakeBehavior. Step: 410000. Time Elapsed: 587.343 s. Mean Reward: 6.953. Std of Reward: 1.442. Training.
[INFO] SnakeBehavior. Step: 420000. Time Elapsed: 644.940 s. Mean Reward: 7.070. Std of Reward: 1.288. Training.
[INFO] SnakeBehavior. Step: 430000. Time Elapsed: 703.978 s. Mean Reward: 7.315. Std of Reward: 1.163. Training.
[INFO] SnakeBehavior. Step: 440000. Time Elapsed: 763.063 s. Mean Reward: 6.750. Std of Reward: 1.312. Training.
```

Рис. 5. Процес навчання моделі PPO+LSTM

Джерело: виконано автором

Тренування тривало 2021 с для PPO+LSTM та 6266 с для PPO (по 1 000 000 кроків для кожної моделі). За графіками навчання (рис. 6) рекурентна модель раніше виходить на плато сумарної винагороди, тобто швидше збігається до стабільної стратегії. В результаті тренування було отримано 2 нейромережеві моделі, які було використано для запуску 2-х ігор. Третя запущена гра використовувала BT, які протягом 20 хв зберігали статистику по іграх. Якщо агент програвав або вигравав - гра починалась заново. За результатом, було отримано такі дані для порівняння:

Таблиця 1. Порівняльні результати моделей та Behaviour Tree

Параметр	Behaviour Tree	ML-Agents PPO	ML-Agents PPO+LSTM
Всього ігор	170	32	94
Перемог	26	23	94
Процент перемог	15,29412%	71,875%	100%
Загальна тривалість ігор	1234,199	1211,2	1240,699
Середня тривалість гри	7,259997	37,84999	13,19893
Найкращий час проходження гри	9,799988	16,59998	13,1
Найгірший час проходження гри	14,60001	85,89996	13,20007

Джерело: виконано автором

Примітка. «Середня тривалість гри» обчислена за всіма партіями, включно з програшами; «найкращий час проходження гри» та «найгірший час проходження гри» — лише за виграними партіями.

Рис. 6. Порівняльний графік TensorBoard сумарної винагороди при навчанні моделей PPO та PPO+LSTM

Джерело: виконано автором у програмному засобі TensorBoard

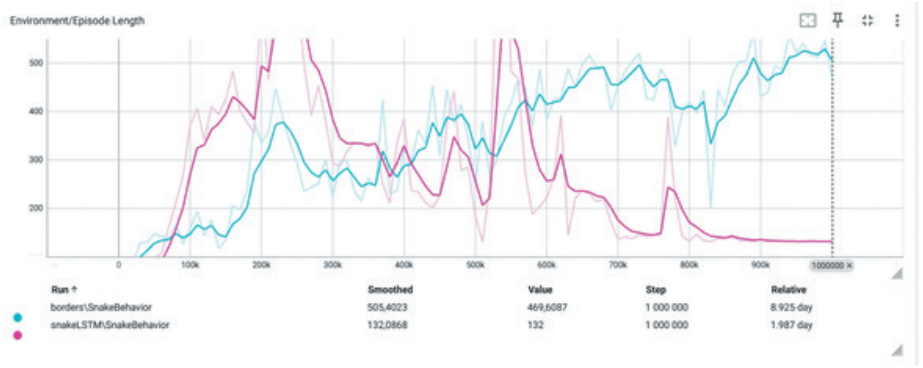

Рис. 7. Порівняльний графік TensorBoard тривалості епізоду при навчанні моделей PPO та PPO+LSTM

Джерело: виконано автором у програмному засобі TensorBoard

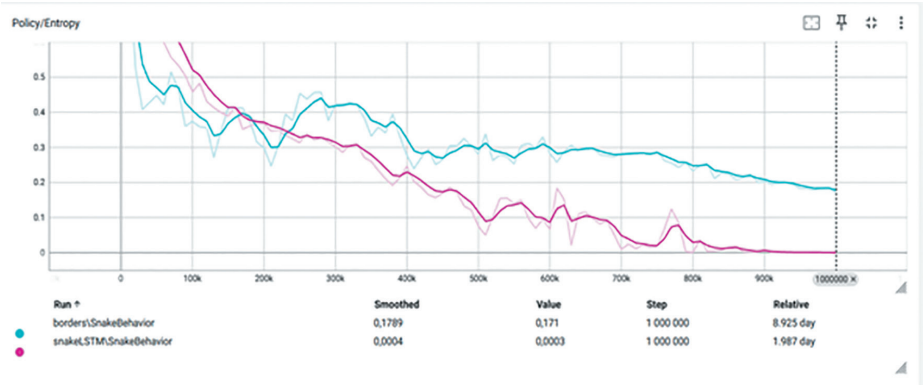

Рис. 8. Порівняльний графік TensorBoard ентропії стратегії при навчанні моделей PPO та PPO+LSTM

Джерело: виконано автором у програмному засобі TensorBoard

Висновки та пропозиції. На основі отриманих статистичних даних порівняльного аналізу трьох підходів до керування NPC - традиційних дерев поведінки (Behaviour Tree), алгоритму PPO та гібридної архітектури PPO+LSTM - можна сформулювати наступні наукові висновки:

Результати тестування демонструють значну перевагу методів машинного навчання над детермінованими алгоритмами у задачах зі складними просторовими обмеженнями. Зокрема, використання архітектури PPO+LSTM дозволило досягти 100% показника успішності (94 перемоги у

94 іграх), що свідчить про повну адаптацію моделі до заданої конфігурації поля. На противагу цьому, класичне дерево поведінки продемонструвало найнижчу ефективність (15,29%), що пояснюється складністю ручного опису всіх можливих сценаріїв самоперетину змійки в межах статичної логіки вузлів Selector та Sequence.

Слід зауважити, що отримані показники, зокрема 100% успішності PPO+LSTM, стосуються середовища з фіксованим розташуванням бонусів. За таких умов агент навчається оптимальній стратегії для конкретної статичної конфігурації поля, тобто результат демонструє здатність моделі збігатися до оптимальної політики в детермінованому середовищі, а не узагальнювати її на довільні конфігурації. Дослідження здатності до генералізації за умов випадкового розміщення бонусів є напрямом подальшої роботи.

Порівняння результатів PPO та PPO+LSTM підкреслює критичну роль пам'яті в управлінні NPC.

Агент на базі стандартного PPO продемонстрував високу варіативність результатів: хоча відсоток перемог зріс до 71,8%, різниця між найкращим (16,59 с) та найгіршим (85,89 с) часом проходження гри свідчить про нестабільність траєкторій.

Впровадження LSTM стабілізувало поведінку агента: часові показники проходження гри стали максимально згрупованими (діапазон між 13,1 с та 13,2 с), що підтверджує здатність рекурентних блоків апроксимувати оптимальну стратегію руху на основі історії попередніх станів.

Детермінований підхід (BehaviourTree) показав найменшу середню тривалість гри (7,25с), проте цей показник обумовлений не швидкістю досягнення мети, а високою частотою ранніх зіткнень (Failure). У той же час модель PPO+LSTM демонструє оптимальний баланс між швидкістю та безпекою маневрування, забезпечуючи стабільне проходження за ~13,2 секунди.

Експериментальні дані підтверджують, що впровадження методів глибокого навчання з підкріпленням, зокрема з використанням контекстно-залежних політик (Context-aware policy), є суттєво ефективнішою альтернативою традиційним методам інтелекту NPC. Це дозволяє створювати автономних агентів, здатних до безпомилкового виконання задач у динамічних середовищах, де традиційні методи (BT) виявляються обмеженими через неможливість врахування нелінійних просторових залежностей.

При проєктуванні систем на базі ML-Agents необхідно враховувати архітектурну детермінованість вхідних даних (Observables). На відміну від традиційних алгоритмів, що дозволяють оперувати динамічними структурами даних, нейромережеві моделі потребують фіксованої розмірності вхідного тензора. Це обмежує можливість використання масивів змінної

довжини, вимагаючи попереднього визначення максимального обсягу даних з нульовими значеннями або застосування методів просторової агрегації (наприклад, Grid Sensor).

Це обмеження було нівельовано шляхом переходу від векторного опису кожного сегмента хвоста до матричної репрезентації ігрового поля, що дозволило трансформувати динамічну зміну довжини об'єкта у статичний розподіл значень у межах вхідного тензора 20x20, що забезпечило стабільність навчання незалежно від поточного стану агента.

Рекомендації для застосування

Для програмістів. Перехід від детермінованого коду до навчання з підкріпленням вимагає зміни парадигми з "як робити" на "що досягти".

Оцінюйте складність простору станів, наприклад, якщо логіка NPC включає занадто багато умовних розгалужень (if-else) або потребує складного прогнозування траєкторій (як у випадку з довгим хвостом змійки), ML-Agents є ефективнішим вибором.

Використовуйте гібридні архітектури для пам'яті: для задач, де поточного спостереження недостатньо для прийняття рішення, обов'язково впроваджуйте LSTM-блоки. Це стабілізує поведінку агента та дозволяє йому "відчувати" динаміку власного тіла або прихованих об'єктів.

Враховуйте обмеження вхідних даних: Пам’ятайте, що нейромережі потребують фіксованої розмірності вхідних тензорів. Використовуйте Grid Sensor, Ray Perception Sensors, CameraSensor, RenderTextureSensor для перетворення динамічних середовищ.

Для геймдизайнерів. ML-Agents змінює процес ітерації та налаштування складності гри.

Геймдизайнер має чітко визначити, що є "добре", а що "погано", тому проєктуйте функцію винагороди як "правила гри". Уникайте занадто складних штрафів, які можуть призвести до того, що агент взагалі відмовиться рухатися, щоб не отримувати негативні бали.

Якщо фінальна задача занадто складна, розбийте її на етапи (наприклад, спочатку змійка вчиться просто їсти без хвоста, а потім — маневрувати з ним). Це значно прискорює процес розробки.

Навчені агенти часто знаходять “експлойти” в механіках, тому якщо поведінка виглядає занадто механічною або ідеальною (як показники 100% перемог), додайте невелику затримку в прийнятті рішень або обмежте точність сенсорів для створення більш природного досвіду для гравця.

Використовуйте ML-Agents для автоматизованого тестування рівнів. Агент може знайти місця, де гравець може застрягти або де складність стає непереборною, це зможе суттєво прискорити розробку проектів.

ЛІТЕРАТУРА/ REFERENCES

1. Millington I., Funge J. Artificial Intelligence for Games. Taylor & Francis Group, 2018. 872 p.

2. Liu T., Cann A., Colbert I., Saeedi M. BT+RL: Hybrid NPCs for Commercial Video Games / T. Liu, A. Cann, I. Colbert, M. Saeedi // arXiv. 2025. arXiv:2510.14154v1.

3. Addoum M. A., Mekhaemar J., Rouffet M., Jacopin E. Khaldun: GOAP for both Procedural Level generation and NPC Behaviors // Proceedings of the IEEE Conference on Games (CoG). 2024. P. 324-326.

4. Bellemare M., Veness J., Bowling M. Investigating Contingency Awareness Using Atari 2600 Games. Proceedings of the AAAI Conference on Artificial Intelligence. 2012. Vol. 26, no. 1. P. 864–871.

5. Human-level control through deep reinforcement learning / V. Mnih et al. Nature. 2015. Vol. 518, no. 7540. P. 529–533.

6. Juliani A., Berges V.-P., Vckay E., Gao Y., Henry H., Mattar M., Lange D. Unity: A General Platform for Intelligent Agents // Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE). 2018. Vol. 14, No. 1. P. 123-134.

7. Ma B., Tang M., Zhang J. Application of Reinforcement Learning in the Snake Game: A Comparative Study of Q-learning and SARSA / B. Ma, M. Tang, J. Zhang. - [S.l. : University of California, Berkeley, n.d.]. 5 p.

8. Hausknecht M., Stone P. Deep Recurrent Q-Learning for Partially Observable MDPs // Advances in Neural Information Processing Systems 28 (NIPS 2015) / ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, R. Garnett. 2015. P. 3338 - 3346.

9. Li Z., Xi-Jia Z., Altundas B., Chen L., Paleja R., Gombolay M. Towards Automated Semantic Interpretability in Reinforcement Learning via Vision-Language Models // arXiv. 2025.

10. GitHub - Unity-Technologies/ml-agents: The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents using deep reinforcement learning and imitation learning. GitHub. URL: https://github.com/Unity-Technologies/ml-agents (дата звернення: 05.02.2026).

11. Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine. Unity. URL: https://unity.com/ (дата звернення: 05.02.2026).

12. Sekhavat Y. A. Behavior Trees for Computer Games. International Journal on Artificial Intelligence Tools. 2017. Vol. 26, no. 02. P. 1730001.

13. Unity - Scripting API: Random.Range. Unity - Manual: Unity 6.3 User Manual. URL: https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Random.Range.html (дата звернення: 02.02.2026).

14. Previous PyTorch Versions. PyTorch. URL: https://pytorch.org/get-started/previous-versions/ (дата звернення: 02.02.2026).

15. CUDA Toolkit Archive. NVIDIA Developer. URL: https://developer.nvidia.com/cuda-toolkit-archive (дата звернення: 02.02.2026).