Міністерство освіти і науки України
Луцький національний технічний університет
Факультет робототехніки та штучного інтелекту
Кафедра штучного інтелекту та математичного моделювання

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ
«БАКАЛАВР»

РОЗРОБКА ТА ДОСЛІДЖЕННЯ АВТОНОМНОГО ІНТЕЛЕКТУАЛЬНОГО АГЕНТА У ВІРТУАЛЬНОМУ СЕРЕДОВИЩІ МЕТОДАМИ ГЛИБОКОГО НАВЧАННЯ З ПІДКРІПЛЕННЯМ

DEVELOPMENT AND RESEARCH OF AN AUTONOMOUS INTELLIGENT AGENT IN A VIRTUAL ENVIRONMENT USING DEEP REINFORCEMENT LEARNING

Спеціальність 113 Прикладна математика
(шифр і назва спеціальності)

освітня програма «Штучний інтелект та аналіз масивів даних»
(назва освітньої програми)

Виконав: здобувач вищої освіти
Групи ПРМ-41
Крупський Олександр Віталійович
_____
(підпис)

Керівник:
к.т.н., доцент
Приходько Олексій Сергійович
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 20__ р.
к.т.н., доцент
Гарант освітньої програми:
Приходько Олексій Сергійович

_____
(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *архітектури, будівництва та дизайну*
Кафедра *прикладної математики та механіки*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *11 Математика і статистика*
Спеціальність *113 Прикладна математика*
Освітня програма «*Штучний інтелект та аналіз масивів даних»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Мікуліч О.А.
«___» _____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Крупський Олександр Віталійович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи
Розробка та дослідження автономного інтелектуального агента у віртуальному середовищі методами глибокого навчання з підкріпленням
Development and research of an autonomous intelligent agent in a virtual environment using deep reinforcement learning
Керівник роботи: *Приходько Олексій Сергійович*
затверджені наказом закладу вищої освіти від «31» грудня 2025 р. № 557/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи
«___»_____ 2026 р.

3. Вихідні дані до роботи *профільні публікації з навчання з підкріпленням, глибокого навчання та комп'ютерного зору. Власний набір даних, математичні моделі MDP, Q-навчання та CNN. Документація використаних бібліотек та методичні вказівки до кваліфікаційної роботи бакалавра*_____

4. Зміст пояснювальної записки (перелік питань, що потрібно розробити):
Аналіз предметної області
Постановка задачі та вибір методів
Практична реалізація
Отримання та аналіз результатів

5. Перелік графічного (ілюстративного) матеріалу:
Презентація роботи (слайди): об'єкт, предмет, мета та завдання дослідження; аналіз предметної області; архітектура агента (ADB, CNN, DQN); математичні та алгоритмічні моделі; результати експериментів; висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
1 розділ	*Приходько О.С., доцент кафедри*		
2 розділ	*Приходько О.С., доцент кафедри*		
3 розділ	*Приходько О.С., доцент кафедри*		
4 розділ	*Приходько О.С., доцент кафедри*		
Висновки	*Приходько О.С., доцент кафедри*		

7. Дата видачі завдання «___» _____ 202__ р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи магістра	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми*	*до 01.03.2026*	
2.	*Перший розділ*	*до 5.03.2026*	
3.	*Другий розділ*	*до 01.04.2026*	
4.	*Третій розділ*	*до 10.04.2026*	
5.	*Четвертий розділ*	*до 20.04.2026*	
6.	*Висновки*	*до 28.04.2026*	
7.	*Формування списку використаних джерел*	*до 05.05.2026*	
8.	*Оформлення ілюстративного матеріалу*	*до 09.05.2026*	
9.	*Нормоконтроль*	*до 20.05.2026*	
10.	*Інструментальна перевірка на академічний плагіат*	*до 02.062026*	*Показник запозичень тексту ____ %*
11.	*Представлення кваліфікаційної роботи бакалавра до захисту*	*до 04.06.2026*	

Здобувач вищої освіти _____ (Крупський О. В.)
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ (Приходько. О. С.)
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Крупський О. В. Розробка та дослідження автономного інтелектуального агента у віртуальному середовищі методами глибокого навчання з підкріпленням. Рукопис.

Кваліфікаційна робота бакалавра ОП «Штучний інтелект та аналіз масивів даних» спеціальності 113 Прикладна математика. Луцький національний технічний університет. Луцьк, 2026.

Робота присвячена дослідженню та програмній реалізації автономного агента для мобільної головоломки Block Blast у віртуальному середовищі Android-емулятора. Проведено аналіз предметної області та формалізовано задачу керування у термінах марковського процесу прийняття рішень. Обґрунтовано декомпозицію системи на модулі сприйняття, прийняття рішень і виконання дій. Розроблено програму annotator.py для збору навчальної вибірки пар «зображення фігури – бінарна матриця форми». Побудовано та навчено згорткову нейронну мережу block_blast_recognizer для класифікації 40 геометричних форм блоків. Реалізовано агента Deep Q-Network з маскуванням легальних дій, буфером повторів і цільовою-мережею; інтегровано зчитування стану гри та керування через Android Debug Bridge (BlueStacks, hd-adb). Програмна реалізація виконана мовою Python з використанням TensorFlow/Keras, OpenCV та NumPy.

Експериментальне дослідження підтвердило працездатність запропонованої архітектури. На власному наборі з 503 зображень (після балансування та HSV-аугментації – близько 7040 зразків) згортковий класифікатор досяг точності на валідаційній вибірці близько 100% (на тренувальній – близько 99%). Онлайн-навчання DQN у live-режимі емулятора забезпечило наповнення буферу повторів, стабільне оновлення Q-мережі та збереження контрольних точок blockblast_dqn.pkl. Отримані результати демонструють можливість побудови інтелектуального агента для комерційної мобільної гри без доступу до внутрішнього API додатку. Розроблене рішення може бути використане як основа для подальшого дослідження методів навчання з підкріпленням у віртуальних ігрових та інтерактивних середовищах.

Ключові слова: навчання з підкріпленням, Deep Q-Network, згорткові нейронні мережі, Block Blast, марковський процес прийняття рішень, комп'ютерний зір, Android Debug Bridge, машинне навчання, TensorFlow, емулятор, автономний агент.

ABSTRACT

Krupskyi O. V. Development and research of an autonomous intelligent agent in a virtual environment using deep reinforcement learning methods. Manuscript.

Bachelor's qualification thesis, Educational Programme «Artificial Intelligence and Data Analysis», specialty 113 Applied Mathematics. Lutsk National Technical University. Lutsk, 2026.

The thesis is devoted to the research and software implementation of an autonomous agent for the mobile puzzle game Block Blast in a virtual Android emulator environment. The subject domain has been analyzed and the control problem has been formalized in terms of a Markov decision process. The decomposition of the system into perception, decision-making, and action execution modules has been justified. The annotator.py program has been developed for collecting a training dataset of «piece image – binary shape matrix» pairs. A convolutional neural network block_blast_recognizer has been built and trained to classify 40 geometric block shapes. A Deep Q-Network agent has been implemented with legal-action masking, a replay buffer, and a target network; game state acquisition and control have been integrated via Android Debug Bridge (BlueStacks, hd-adb). The software implementation was carried out in Python using TensorFlow/Keras, OpenCV, and NumPy.

The experimental study confirmed the feasibility of the proposed architecture. On a custom dataset of 503 images (after class balancing and HSV augmentation – approximately 7040 samples), the convolutional classifier achieved validation accuracy of approximately 100% (on the training set – approximately 99%). Online DQN training in the emulator's live mode ensured replay buffer population, stable Q-network updates, and saving of checkpoints blockblast_dqn.pkl. The obtained results demonstrate the possibility of building an end-to-end agent for a commercial mobile game without access to the application's internal API. The developed solution may serve as a basis for further research on reinforcement learning methods in virtual gaming and interactive environments.

Keywords: reinforcement learning, Deep Q-Network, convolutional neural networks, Block Blast, Markov decision process, computer vision, Android Debug Bridge, machine learning, TensorFlow, emulator, autonomous agent.

ЗМІСТ

ВСТУП

Сучасні мобільні ігри формують великий клас віртуальних середовищ, у яких гравець приймає послідовність рішень під обмеженням правил, часу та простору на ігровому полі. Завдання автоматичного проходження таких ігор давно привертає увагу дослідників штучного інтелекту, оскільки воно поєднує сприйняття візуального стану, планування та навчання з досвіду. Гра Block Blast належить до класу головоломок із дискретним простором станів: на полі розміром вісім на вісім клітинок гравець розміщує блоки різної форми, а повні рядки або стовпці зникають. Такий формат зручний для формалізації в термінах марковського процесу прийняття рішень і для експериментальної перевірки алгоритмів навчання з підкріпленням.

Актуальність дослідження. Навчання з підкріпленням це сучасна тема, яка використовується у багатьох галузях таких як: робототехніка, інженерія автопілотів та тренування LLM. Розвиток методів глибокого навчання з підкріпленням дозволяє створювати автономних агентів, які без явного програмування правил гри формують стратегію через взаємодію з середовищем. Для прикладної математики це важливо з двох причин. По-перше, такі системи описуються формально через функції цінності, ймовірнісні переходи та оптимізаційні задачі, що дає можливість математично обґрунтувати архітектуру рішення. По-друге, практична реалізація агента в реальному віртуальному середовищі через емулятор Android вимагає інтеграції комп'ютерного зору, керування пристроєм і чисельного навчання, що відображає типовий профіль фахівця зі штучного інтелекту та аналізу масивів даних.

Стан розробки проблеми. Теоретичні основи навчання з підкріпленням систематизовано у класичних працях Sutton і Barto [1]. Перехід від табличних методів до функціонального наближення Q-функції з використанням глибоких нейронних мереж продемонстровано Mnih та співавт. у роботі про Deep Q-Network [2]. У задачах комп'ютерного зору для класифікації зображень широко застосовуються згорткові нейронні мережі, архітектура та навчання яких

детально описані у монографії Goodfellow, Bengio та Courville [3]. Водночас поєднання зорового розпізнавання фігур у мобільній грі з онлайн-навчанням політики в умовах шумних скріншотів і нестабільного каналу керування залишається практично складною інженерною задачею, яка потребує власного дослідження на конкретному об'єкті.

Мета роботи полягає у розробці та експериментальному дослідженні автономного інтелектуального агента для гри Block Blast у віртуальному середовищі Android-емулятора методами глибокого навчання з підкріпленням.

Для досягнення мети поставлено такі завдання: проаналізувати предметну область гри Block Blast та структуру даних для навчання; розробити програмний інструмент збору навчальної вибірки зображень фігур; побудувати та навчити згорткову нейронну мережу для розпізнавання форм блоків; формалізувати ігрове середовище та реалізувати агента Deep Q-Network для онлайн-навчання; інтегрувати сприйняття стану гри та керування через Android Debug Bridge; провести експериментальне дослідження якості розпізнавання та поведінки агента; узагальнити результати та сформулювати висновки.

Об'єкт дослідження – процес прийняття рішень автономним інтелектуальним агентом у віртуальному ігровому середовищі.

Предмет дослідження – методи глибокого навчання з підкріпленням і згорткові нейронні мережі для сприйняття стану та вибору дій у дискретному просторі Block Blast.

Методику дослідження у роботі реалізовано через експериментальне моделювання, порівняння класичного підходу комп'ютерного зору з навчанням з учителем, чисельну оптимізацію параметрів нейронних мереж, а також аналіз і синтез програмних модулів збору даних, сприйняття та керування.

Інформаційну базу сформовано з монографій і наукових статей з навчання з підкріпленням та глибокого навчання [1–5], офіційної документації програмних бібліотек TensorFlow, OpenCV, scikit-learn [6–9], методичних вказівок ЛНТУ до оформлення кваліфікаційних робіт [10], а також власного масиву даних, зібраного за допомогою розробленої програми annotator. З точки зору прикладної математики Block Blast цікавий тим, що поєднує комбінаторну

структуру розміщення поліміно на решітці з необхідністю оцінювати довгострокові наслідки локальних рішень. Навіть коли поточний хід формально допустимий, він може створити фрагментацію вільного простору і прискорити програш. Тому недостатньо реалізувати greedy-евристику «поставити фігуру в першу доступну клітинку» – потрібен механізм, здатний накопичувати досвід через багато епізодів гри.

У контексті освітньої програми «Штучний інтелект та аналіз масивів даних» дана робота демонструє повний цикл розробки на основі даних: від формування власного масиву даних до побудови моделей, що працюють у реальному середовищі. Особливу увагу приділено якості даних, оскільки помилка анотації однієї фігури безпосередньо впливає на здатність CNN [21] правильно відновити форму блоку, а отже – на коректність множини легальних дій для DQN.

Практична мотивація також полягає у тому, що мобільні ігри часто не надають програмного API для зчитування внутрішнього стану. Єдиним доступним каналом залишається візуальний інтерфейс користувача. Саме тому в роботі було прийнято рішення будувати агента як систему «комп'ютерний зір + навчання з підкріпленням + емуляція дотиків», а не як модуль, інтегрований у пам'ять гри.

У процесі підготовки роботи опрацьовано літературу з табличного Q-learning, experience replay, згорткових мереж, загальних підходів до інтелектуальних агентів [17] та сучасних практик Deep RL [1–3, 20]. Це дозволило обґрунтувати вибір DQN як базового, але достатньо потужного алгоритму для дискретного простору дій помірної розмірності з високорозмірним спостереженням.

Наукова новизна роботи полягає не у запропонованні нового теоретичного алгоритму, а в комплексній адаптації відомих методів до специфічного поєднання умов: мобільна гра, емулятор, шумні скріншоти, необхідність власного набору даних і онлайн-навчання політики. Для бакалаврської кваліфікаційної роботи саме така інтеграція є достатньою і водночас показовою з практичної точки зору.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ДАНИХ

1.1. Характеристика гри Block Blast та її ігрового простору

Block Blast є мобільною головоломкою, у якій основна дія полягає у розміщенні на прямокутному полі блоків тетроміно-подібних форм. У досліджуваній версії гри поле має розмір вісім на вісім клітинок, а гравцеві одночасно доступні три фігури, з яких він обирає одну для кожного ходу. Після успішного розміщення фігура зникає з нижньої панелі, а на полі залишаються заповнені клітинки. Якщо рядок або стовпець повністю заповнені, відповідна лінія очищується, що створює додатковий простір для наступних ходів. Гра завершується, коли жодну з доступних фігур неможливо розмістити на дошці без перетину з уже зайнятими клітинками.

У математичному описі стан гри можна представити бінарною матрицею $\boldsymbol{B} \in \{0,1\}^{8\times 8}$, де одиниця означає зайняту клітинку, а нуль – вільну. Кожна фігура описується компактною бінарною підматрицею $\boldsymbol{P} \in \{0,1\}^{h\times w}$, де h та w не перевищують п'яти, оскільки у практичній реалізації анотації використовувалася сітка п'ять на п'ять з подальшим обрізанням нульових рядків і стовпців. Таким чином, на кожному кроці агент спостерігає кортеж $(\boldsymbol{B}, \boldsymbol{P}_1, \boldsymbol{P}_2, \boldsymbol{P}_3)$, де деякі $\boldsymbol{P}_i$ можуть бути порожніми, якщо відповідний слот на панелі фігур не містить блоку.

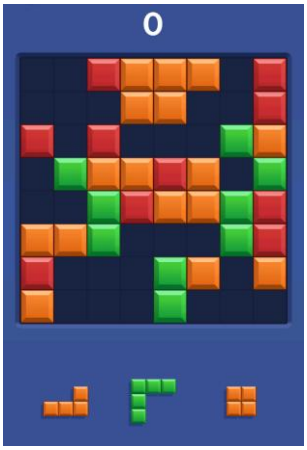

Рисунок 1.1 – Ігровий інтерфейс Block Blast у емуляторі BlueStacks

Правила розміщення формулюються через операцію накладання фігури на підматрицю дошки. Дія (i, r, c) означає вибір i-ї фігури та її розміщення так, щоб лівий верхній кут фігури відповідав клітинці (r, c) дошки. Хід допустимий тоді і лише тоді, коли для всіх клітинок фігури виконується $\boldsymbol{B}[r+u, c+v] + \boldsymbol{P}[u, v] \leq 1$. Після розміщення оновлюється матриця дошки, після чого перевіряється наявність повніх ліній. Такий опис дозволяє однозначно генерувати множину легальних дій на кожному кроці, що є критично важливим для навчання з підкріпленням, оскільки агент не повинен витрачати дослідницький бюджет на заведомо неможливі ходи.

1.2. Віртуальне середовище та канал взаємодії з грою

Для відтворення мобільного середовища у роботі було використано емулятор BlueStacks, який запускає Android-додаток Block Blast на персональному комп'ютері. Взаємодія з емулятором організована через Android Debug Bridge (ADB) – інструмент командного рядка, що дозволяє отримувати скріншот екрана та імітувати дотики і жести перетягування. У практичній конфігурації застосовувалась утиліта hd-adb, яка постачається разом із BlueStacks і підключається до локального порту емулятора, зазвичай 127.0.0.1:5555.

Скріншот отримують командою exec-out screencap -p, після чого байтовий потік декодується засобами OpenCV у масив NumPy формату BGR. Команда input swipe використовується для перетягування фігури з панелі на цільову позицію дошки. Тут важливо врахувати особливість фізики гри: блок у Block Blast рухається швидше, ніж кінцева точка пальцевого жеста, тому у програмній реалізації застосовано коефіцієнт swipe_gain, який скорочує довжину свайпу відносно геометричної відстані між центром фігури та цільовою клітинкою.

Рисунок 1.2 – Схема взаємодії програмного агента з емулятором через ADB

Оскільки роздільність вікна емулятора може змінюватися, у роботі реалізовано автоматичну калібрацію координат. Еталонні прямокутники обрізки поля та панелі фігур, отримані для розміру кадру приблизно 720×1280, масштабуються пропорційно до поточної ширини та висоти скріншота. Це зменшує потребу в ручному підборі піксельних координат при зміні налаштувань дисплея.

1.3. Теоретичні основи навчання з підкріпленням

Задачу керування агентом у грі формалізують як марковський процес прийняття рішень (MDP). Кортеж процесу задається у вигляді

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma) \quad (1.1)$$

де $\mathcal{S}$ – множина станів, $\mathcal{A}$ – множина дій, $P(s'|s,a)$ – ймовірність переходу до стану s' зі стану s після дії a, $R(s,a,s')$ – функція нагороди, $\gamma \in [0,1)$ – коефіцієнт дисконтування майбутніх нагород.

де $\mathcal{S}$ – множина станів; $\mathcal{A}$ – множина дій; $P(s'|s,a)$ – ймовірність переходу; $R(s,a,s')$ – функція миттєвої нагороди; γ – коефіцієнт дисконтування.

Мета агента – максимізувати очікувану сумарну дисконтовану нагороду. Для дискретного простору дій і тривалого горизонту використовують функцію цінності дії

$$Q(s,a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a] \quad (1.2)$$

де $Q(s,a)$ – очікувана сумарна нагорода при виконанні дії a у стані s; r_t – нагорода на кроці t; γ – коефіцієнт дисконтування.

Оптимальна Q-функція задовольняє рівняння Беллмана

$$Q^*(s,a) = \mathbb{E}_{s'}\left[R + \gamma \max_{a'} Q^*(s',a')\right] \quad (1.3)$$

де Q^* – оптимальна функція цінності дії; s' – наступний стан; a' – наступна дія.

Алгоритм Q-learning наближує Q ітераційним оновленням на основі спостережуваних переходів. У методі Deep Q-Network функція $Q(s,a;\theta)$ апроксимується нейронною мережею з параметрами θ, що дозволяє працювати з високорозмірними вхідними станами, зокрема зображеннями або їх внутрішніми представленнями [2].

1.4. Аналіз зібраного масиву даних для навчання зорового модуля

Початковий етап дослідження показав, що класичні евристики сегментації кольору та k-means не забезпечують стабільного відновлення форми фігур у реальних скріншотах через варіативність освітлення, градієнтів і фону панелі. Тому для навчання згорткової мережі було сформовано власний датасет. Програма annotator, описана детально у розділі 3, зберігає для кожного прикладу пару файлів: зображення фрагмента панелі фігур у форматі PNG та відповідну бінарну матрицю форми у форматі NPY.

У фінальній конфігурації експерименту завантажено 503 зображення, які відповідають 40 унікальним геометричним формам блоків. Кожна форма кодується матрицею з елементами нуль і один, де один позначає заповнену клітинку шаблону. Перед навчанням матриці нормалізуються за розміром шляхом обрізання нульових рядків і стовпців, що відповідає природній симетрії блоків у грі.

Таблиця 1.1 – Основні характеристики навчального набору даних

Показник	Значення
Кількість зображень	503
Кількість класів (форм)	40
Розмір зображення після нормалізації	176×200 px
Формат мітки	бінарна матриця .npy
Після балансування та аугментації	близько 7040 зразків

Дисбаланс класів у сирих даних усувався шляхом додаткової генерації аугментованих копій для рідкісних форм. Тут було використано перетворення у колірному просторі HSV з випадковим зсувом відтінку та яскравості, що імітує зміни кольору блоків між партіями гри без зміни геометрії.

1.5. Програмні засоби та бібліотеки, застосовані у роботі

Робоче середовище побудовано на мові Python 3, оскільки для неї існує розвинена екосистема бібліотек машинного навчання та обробки зображень [11].

Python використано як уніфікуючу платформу для збору даних, навчання моделей, експериментів і керування емулятором. *NumPy* [12] забезпечує ефективне представлення матриць дошки, фігур і пакетів зображень. *OpenCV* [7] застосовано для декодування скріншотів, зміни розмірів зображень, перетворень HSV та морфологічної обробки при детекції екрана завершення гри.

TensorFlow та високорівневий API *Keras* [6] використано для побудови та навчання двох нейронних моделей: згорткового класифікатора фігур і Deep Q-Network. *scikit-learn* [9] застосовано для стратифікованого розділення вибірки та one-hot кодування міток класів. *Matplotlib* [13] використано для візуалізації калібрування, кривих навчання та динаміки нагород агента.

Для ручного збору наборі даних було використано *Tkinter* та *Pillow* у скрипті annotator: перша бібліотека створює графічний інтерфейс із сітками анотації, друга – відображає скріншоти фігур. *Android Debug Bridge* [14]

виступає зовнішнім інструментом керування емулятором і не є Python-бібліотекою, але саме через нього реалізовано замкнений контур «спостереження – дія» у фізичному віртуальному середовищі.

У таблиці 1.2 зведено роль кожного програмного компонента у загальній архітектурі рішення.

Таблиця 1.2 – Призначення основних програмних засобів

Засіб	Роль у системі
annotator	збір пар «зображення – матриця форми»
TensorFlow/Keras	навчання CNN і DQN
OpenCV	обробка скріншотів і аугментація
ADB	скріншот і жести в емуляторі
Програмний модуль агента	інтеграція модулів і запуск агента

TensorFlow [6] є відкритою платформою для чисельного обчислення та машинного навчання. У роботі використано високорівневий API Keras [8], який дозволяє декларативно описувати Sequential-моделі CNN і програмно будувати функціональну DQN. Keras абстрагує деталі зворотного поширення помилки і автоматично формує обчислювальний граф для операцій Conv2D, Dense, MaxPooling. Оптимізатор Adam [19] поєднує переваги імпульсного методу та адаптивної швидкості навчання, що зручно для задач класифікації зображень без ручного підбору розкладу швидкості навчання на ранніх етапах. Для CNN model.compile задає categorical cross-entropy як функцію втрат, що відповідає мультикласовій класифікації з one-hot мітками. Збереження моделі у форматі .keras включає архітектуру, ваги та конфігурацію компіляції, що спрощує подальше розгортання навченого класифікатора у програмному модулі агента. DQN реалізовано через підклас tf.keras.Model (DQNet), оскільки цикл навчання власний через GradientTape; target-мережа – окремий екземпляр DQNet з копією ваг policy-мережі. Huber loss зменшує величину градієнта при великих помилках target Q порівняно з чистою MSE, що стабілізує навчання при різких стрибках нагороди від штрафу за завершення гри (game over).

OpenCV [7] використано на кожному етапі зорового ланцюжка: imdecode для PNG-байтів з ADB, resize для приведення ROI фігур до 176×200, cvtColor для HSV-аугментації, inRange і морфологічні операції для детекції кнопки перезапуску гри. Вибір INTER_AREA при зменшенні розміру важливий, бо альтернатива INTER_LINEAR може створювати «подвоєні» краї блоків, що погіршує класифікацію тонких форм. Простір HSV зручніший за RGB для зміни відтінку, оскільки відтінок (hue) відокремлений від яскравості і насиченості.

NumPy [12] забезпечує представлення ndarray для матриць дошки, масок фігур і пакетів зображень. Операції np.unique, булева індексація, clip використовуються у analyze_board_state, augment_image і перевірках легальних ходів. scikit-learn [9] застосовано для train_test_split з stratify=y_encoded, що гарантує представленість рідкісних класів у валідаційній вибірці, та для OneHotEncoder. Matplotlib [13] використано для дослідницького аналізу гістограм класів, кривих навчання та накладення прямокутників калібрування – для відтворюваності експериментів і рисунків до роботи.

Tkinter – стандартна GUI-бібліотека Python; у annotator вона створює LabelFrame для кожної фігури, сітку кнопок 5×5 і обробники подій toggle_button. Pillow (PIL Fork) [15] конвертує numpy RGB-масиви у PhotoImage для відображення після зменшення зображення. ADB [14] викликається через subprocess, а не через окремий обгортки python-adb, щоб мінімізувати залежності; недолік – необхідність власної логіки перепідключення при збоях device offline. Практичні аспекти побудови моделей на Python узагальнено у посібнику Géron [18]. З ігрової механіки Block Blast випливає, що кількість можливих конфігурацій дошки велика, хоча формально вона обмежена $\sum_{k=0}^{64}\binom{64}{k}$, на практиці ж досяжні лише ті стани, які виникають послідовністю легальних розміщень. Це зменшує фактичну складність порівняно з довільною бінарною матрицею, але все одно залишає простір достатньо багатим для RL.

Важливою особливістю є те, що гравець не обирає порядок появи фігур – він реагує на трійку фігур, яку генерує гра. Отже, агент працює в умовах

часткової спостережуваності, якщо розглядати повний стан гри ширше, ніж поточна дошка і три фігури. Проте для практичної реалізації прийнято наближення, що трійка фігур разом із дошкою є достатньою статистикою для прийняття рішення на короткому планувальному горизонті.

Щодо віртуального середовища, BlueStacks надає зручний спосіб відтворити Android-додаток на ПК без модифікації APK-файлу гри. Це етично і технологічно простіше, ніж зворотна інженерія внутрішніх структур даних гри. Недоліком є залежність від стабільності ADB-з'єднання і від затримок рендерингу, які не контролюються агентом напряму.

У теоретичному блоці варто наголосити, що рівняння Беллмана (1.3) задає не просто рекурсію для обчислення Q, а принцип оптимальності: оптимальна цінність поточного рішення дорівнює негайній нагороді плюс дисконтована оптимальна цінність наступного стану. Саме ця структура є основою цільового значення для DQN у формулі (2.7).

Аналіз набору даних показав, що 40 класів відображають різноманітність форм – від одиничного блоку до складніших L-подібних і прямокутних композицій. Кожен клас ідентифікується не лише візуально, а й матрицею форми, що дозволяє однозначно перевірити правильність передбачення CNN не лише за точністю, а й за геометричною відповідністю.

Щодо програмних засобів, Python обрано через швидкість прототипування. TensorFlow/Keras забезпечують декларативний опис CNN і DQN, OpenCV – перевірені операції над зображеннями, scikit-learn – стратифіковане розділення вибірки без ручної реалізації. Tkinter у annotator не потребує окремого веб-стеку і працює локально, що зручно під час тривалої ручної анотації.

Окремо слід зазначити, що ADB не є частиною Python-екосистеми, але документований Android Developers [14] і фактично виступає стандартом для дослідницьких прототипів у мобільних іграх. У BlueStacks використовується варіант hd-adb, який підключається до локального порту емулятора; цю особливість враховано у функції ensure_adb_device. Окремої уваги заслуговує питання, чому саме Block Blast обрано як полігон для RL-агента. На відміну від класичних ігор Atari, де дії часто зводяться до натискання дискретних кнопок,

тут основна дія реалізується жестом перетягування, який у програмній реалізації дискретизується через вибір цільової клітинки на дошці. На відміну від шахів або Go, горизонт гри не обмежений наперед відомою глибиною дерева пошуку, а залежить від генерації нових трійок фігур. Це наближує задачу до практичних сценаріїв керування, де спостереження зашумлене, а виконання дії лише наближено відтворюється.

Поле вісім на вісім формально задає великий простір бінарних конфігурацій, проте на практиці досяжні лише ті стани, які реально виникають у процесі гри. Наявність одночасно трьох фігур створює коефіцієнт розгалуження до 192 легальних дій на крок, що залишається прийнятним для DQN без окремого дерева пошуку. Якщо легальних дій мало, гра наближається до завершення; якщо їх немає взагалі – настав екран завершення гри або сталася помилка зорового модуля.

Емулятор BlueStacks дозволяє фіксувати розмір вікна для відтворюваних координат обрізки. У роботі еталонний кадр 720×1280 пікселів отримано емпірично на етапі попередніх експериментальних запусків системи. Лінійне масштабування в auto_calibrate_from_screenshot припускає однаковий коефіцієнт по осях, що коректно для вікна зі сталим співвідношенням сторін.

Затримка ADB на один скріншот і свайп зазвичай становить сотні мілісекунд. Сумарний крок агента доповнено паузами step_delay_sec та post_placement_check_delay_sec, оскільки при занадто швидкому зчитуванні стану зростала частота штрафів за хибний свайп. З теорії навчання з підкріпленням коефіцієнт дисконтування $\gamma = 0{,}99$ задає ефективний горизонт розподілу кредиту нагороди порядку сотні кроків, що достатньо для типових епізодів Block Blast.

Набір даних із 503 зображень сформовано за кілька сесій ручної анотації. Інструмент annotator оптимізовано під швидке відтворення трьох фігур одночасно, поки на екрані не зміниться наступний стан гри. Словник class_dictionary.json зберігає відповідність між числовим ідентифікатором класу та матрицею форми у форматі, зручному для перевірки людиною та для версіонування разом із кодом.

РОЗДІЛ 2
ПОСТАНОВКА ЗАДАЧІ ТА ВИБІР МЕТОДІВ РОЗВ'ЯЗАННЯ

2.1. Формалізація задачі керування агентом

Автономний агент у дослідженні працює в дискретному часі $t = 0,1,2,\ldots$. На кожному кроці він отримує спостереження поточного стану гри, обирає дію і отримує числову нагороду. Стан s_t формується на основі візуального аналізу екрана і включає матрицю дошки $\boldsymbol{B}_t$ та набір із трьох матриць фігур. Оскільки безпосередньо використовувати сирі пікселі всього кадру неефективно, у роботі застосовано стисле представлення, яке описано у розділі 3.

Простір дій складається з усіх допустимих triple (i, r, c), де $i \in \{0,1,2\}$ – індекс фігури на панелі, (r, c) – координати лівого верхнього кута розміщення на дошці. Теоретична верхня межа кількості дій дорівнює

$$|\mathcal{A}| = 3 \times 8 \times 8 = 192 \tag{2.1}$$

де $|\mathcal{A}|$ – розмірність простору дій; множник 3 відповідає кількості слотів фігур; множники 8 – розміру дошки за рядком і стовпцем.

На практиці на кожному кроці доступна лише підмножина $\mathcal{A}_t \subset \mathcal{A}$ легальних дій, яка визначається правилами гри та поточним вмістом дошки. Перехід між станами у реальному середовищі стохастичний через можливі помилки свайпу, затримки анімації і шум зорового модуля, однак у навчальному циклі DQN використовують детерміноване спостереження наступного стислого стану після паузи, достатньої для стабілізації кадру.

Функція нагороди $r_t = R(s_t, a_t, s_{t+1})$ задається явно і включає як позитивні стимули за успішне розміщення та очищення ліній, так і штрафи за хибні жести, утворення «замкнених дірок» і завершення гри. Коефіцієнт дисконтування у експериментах прийнято $\gamma = 0.99$, що відповідає помірному пріоритету довгострокового виживання на полі.

2.2. Простір дій, легальність ходів і маскування Q-значень

Перед передачею стану у Deep Q-Network усі недопустимі дії мають бути виключені з процесу вибору. Для цього формується бінарна маска $\boldsymbol{m} \in \{0,1\}^{192}$, де $m_a = 1$ тоді і лише тоді, коли дія a легальна. Якщо $\hat{Q}(s, a; \theta)$ – вихід мережі, то під час жадібного вибору та при обчисленні цільового значення для некоректних дій Q замінюють великим від'ємним числом, практично $-\infty$, щоб argmax не обрав їх.

Така схема маскування критична для Block Blast, оскільки без неї агент у фазі використання (exploitation) міг би обирати геометрично неможливі позиції, що призводить до хибних свайпів і спотворення replay buffer. У коді легальність перевіряється через функцію can_place, яка для кожної фігури перебирає всі позиції на дошці вісім на вісім.

Маску легальних дій зберігають разом із переходом у replay buffer, що дозволяє коректно обчислювати $\max_{a'} Q(s', a')$ лише по допустимих a'. Без цієї деталі навчання стає нестабільним, оскільки target Q включав би заведомо нереалізовані дії. Маскування можна інтерпретувати як проекційний оператор на допустиму підмножину $\mathcal{A}_t$: політика фактично визначена не на повному просторі $\mathcal{A}$, а на перетині з геометрично допустимими ходами поточного стану.

Кожен індекс дії $a \in \{0, \dots, 191\}$ однозначно декодується як трійка (i, r, c), де $i = \lfloor a/64 \rfloor$ – індекс фігури, $r = \lfloor (a \bmod 64)/8 \rfloor$ і $c = a \bmod 8$ – координати лівого верхнього кута на дошці. Фіксоване бієктивне кодування спрощує налагодження: помилки в select або train_step легко простежити до конкретної фігури і клітинки.

2.3. Вибір згорткової нейронної мережі для розпізнавання фігур

На першому етапі розробки було реалізовано автоматичне виділення форми блоку через k-means у просторі кольорів і побудову бінарної маски контуру. Емпірично цей підхід давав нестабільні результати: при схожих

відтінках фону і блоку контури рвались, а розмір обмежувального прямокутника не відповідав реальній формі. Тому для модуля зору було обрано підхід з навчанням з учителем – класифікацію зображення фігури одним із 40 класів.

Нехай $\boldsymbol{x}$ – нормалізоване зображення фігури, $\boldsymbol{y}$ – one-hot вектор довжини $K = 40$. Згорткова мережа $f_{\text{CNN}}(\boldsymbol{x}; \boldsymbol{\phi})$ з параметрами $\boldsymbol{\phi}$ формує logits $\boldsymbol{z}$, а ймовірність класу k обчислюють через softmax:

$$P(y = k|\boldsymbol{x}) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.5}$$

де $P(y = k|\boldsymbol{x})$ – ймовірність класу k; z_k – k-ий logit на виході мережі; K – кількість класів.

Навчання мінімізує categorical cross-entropy:

$$\mathcal{L}_{\text{CNN}} = -\sum_{k=1}^{K} y_k \log \hat{y}_k \tag{2.6}$$

де y_k – компонента one-hot мітки; $\hat{y}_k$ – передбачена ймовірність класу k.

Після класифікації індекс найбільш ймовірного класу відображається у матрицю форми через словник class_dictionary.json. Ця згорткова нейронна мережа виконує роль percеption-модуля, який перетворює піксельний фрагмент панелі фігур у символічний опис, сумісний з логікою розміщення блоків.

2.4. Обґрунтування методу Deep Q-Network

Для стратегічного рівня гри застосовано Deep Q-Network (DQN) [2]. Політика задається через наближення $Q(s, a; \theta)$. Після виконання переходу (s, a, r, s') формується цільове значення

$$y = r + \gamma \max_{a' \in \mathcal{A}'} Q(s', a'; \theta^-) \tag{2.7}$$

де y – ціль для оновлення Q-значення; θ^- – параметри target-мережі; $\mathcal{A}'$ – множина легальних дій у стані s'.

Параметри θ оновлюють мінімізацією Huber loss між $Q(s, a; \theta)$ і y:

$$\mathcal{L}_{\text{DQN}} = \mathcal{H}(y,\ Q(s, a; \theta)) \tag{2.8}$$

де $\mathcal{H}$ – функція Huber, стійка до викидів у нагородах.

Для стабілізації навчання використано replay buffer ємністю 50 000 переходів, періодичну синхронізацію target-мережі кожні 200 кроків і ε-greedy стратегії досліджувального вибору. Розклад ε у роботі двофазний: на ранніх кроках домінує випадковий вибір легальних дій для наповнення буфера різноманітними ситуаціями, далі частка випадковості експоненційно спадає до малого значення близько 0,03.

Хоча повний піксельний стан мав би розмірність $H \times W \times 3$, практичний стан стискається до тензора $8 \times 8 \times 4$: перший канал – сітка зайнятості дошки, канали 2–4 – три маски фігур із прив'язкою до лівого верхнього кута. Це інженерний вибір між повнотою спостереження і складністю навчання: end-to-end Conv2D на такій розрідженій сітці достатньо для концептуального DQN у межах бакалаврської роботи. Альтернативне кодування могло б включати інженерно сконструювані ознаки – кількість «дірок», висоти стовпців, – але потребувало б додаткового підбору без гарантії переваги над згортковим представленням.

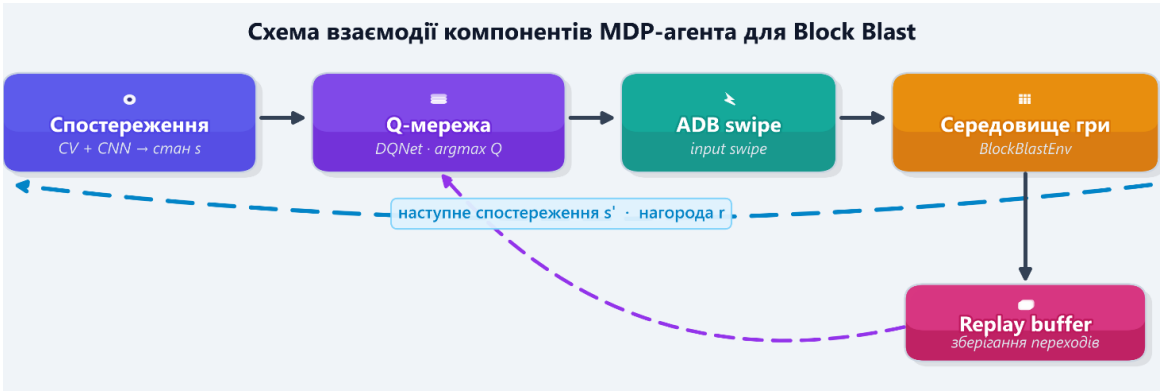

Рисунок 2.1 – Схема взаємодії компонентів MDP-агента для Block Blast

2.5. Функція нагороди та її компоненти

Якість навчання DQN суттєво залежить від формування нагороди. У роботі сумарна нагорода за крок представлена як лінійна комбінація компонент:

$$R = w_p r_{\text{place}} + w_c r_{\text{clear}} + w_m r_{\text{multi}} + w_h r_{\text{hole}} + w_f r_{\text{false}} + w_g r_{\text{over}} + w_t r_{\text{term}}, \quad (2.10)$$

де $w_p, w_c, \ldots$ – вагові коефіцієнти компонент; r_{place} – базова нагорода за підтверджене розміщення; r_{clear} – бонус за очищені лінії; r_{multi} – додатковий бонус за одночасне очищення кількох ліній; r_{hole} – штраф за нові замкнені порожні клітинки; r_{false} – штраф за свайп без появи блоків; r_{over} – штраф за завершення гри (game over); r_{term} – штраф за стан без легальних ходів.

У практичній конфігурації RewardWeights нагорода за одну лінію дорівнює 6, за розміщення – 1, штраф за хибний свайп – 8, штраф за завершення гри – 25. Такий баланс спрямований на те, щоб агент насамперед уникав технічних помилок керування, а вже потім оптимізував очищення ліній. Формалізація MDP у розділі 2.1 навмисно залишається на рівні, достатньому для практичної реалізації. Стан s_t у коді не є сирим скріншотом, а компактним тензором, побудованим після CV і CNN. Це приклад ланцюжок навчання представлень: спочатку спостереження стискається до семантично значущих ознак, потім Q-мережа працює вже з цим простором.

Маскування Q-значень можна інтерпретувати як проекційний оператор на допустиму підмножину дій. Без нього вибір argmax Q часто «випадково» потрапляв би на недопустимий індекс, що у live-режимі призводило б до серії хибних свайпів. Тому маска – не опційна оптимізація, а структурне обмеження задачі.

Порівняння k-means і CNN ілюструє типовий компроміс між класичним ланцюжком без набору даних і навчанням сприйняття на основі даних. K-means не потребує набору даних, але чутливий до зміни освітлення і фону. CNN потребує анотацій, проте після навчання стійко відновлює клас форми навіть при зміні відтінку (hue) блоків, особливо якщо аугментація під час тренування відображала ці зміни.

Математично softmax (2.5) можна трактувати як експоненційне перетворення logits у ймовірнісний симплекс Δ^{K-1}. Cross-entropy (2.6) є мінус логарифмом правдоподібності мультикласової моделі і зручна для градієнтної оптимізації Adam [19]. У DQN Huber loss зменшує чутливість до рідкісних великих відхилень target Q, що важливо при формування нагороди з різкими штрафами.

Replay buffer розриває часову кореляцію між послідовними кадрами одного епізоду, що стабілізує SGD-оновлення в Q-мережі. Target network зменшує проблему «рухомої цілі»: без θ^- мережа «ганялася» б за власними швидко змінюваними передбаченнями. Синхронізація кожні 200 кроків – емпірично обраний компроміс між свіжістю target і стабільністю.

Функція нагороди (2.10) – результат кількох ітерацій підбору. Якщо штраф за хибний свайп замалий, агент ігнорує якість жестів; якщо занадто великий – replay buffer наповнюється переважно негативними переходами і навчання сповільнюється. Аналогічно, бонус за одночасне очищення кількох ліній мотивує не просто ставити фігури, а формувати ряди, що відповідає меті самої гри. Формалізм MDP у розділі 2 залишається абстрактним, тоді як у коді перехід $P(s'|s,a)$ не моделюється явно: середовище трактують як «чорну скриньку», яка повертає наступне спостереження та нагороду. Саме тому безмодельний DQN є природним вибором, коли внутрішня фізика гри недоступна напряму.

Маска легальних дій зберігається разом із переходом у replay buffer, що дозволяє коректно обчислювати $\max_{a'} Q(s', a')$ лише по допустимих a'. Без цієї деталі навчання стає нестабільним, оскільки target Q включає заведомо нереалізовані дії.

Порівняння k-means і CNN ілюструє типовий компроміс між класичним ланцюжком без набору даних і навчанням сприйняття на основі даних. K-means не потребує анотацій, але чутливий до освітлення та кольору фону панелі фігур. CNN потребує ручної праці annotator, проте після навчання стійко відновлює

клас форми при зміні відтінку (hue) блоків, особливо якщо аугментація під час тренування відображала такі зміни.

Значення ваг у функції нагороди (2.10) підбиралися ітеративно під час розробки модуля керування агентом. Якщо штраф за хибний свайп замалий, агент ігнорує якість жестів; якщо занадто великий – replay buffer насичується переважно негативними переходами. Аналогічно бонус за одночасне очищення кількох ліній мотивує не просто заповнювати поле, а формувати повні ряди, що відповідає меті самої гри.

Розклад ε -greedy з фазою досліджувального вибору і подальшим експоненційним спадом дозволяє спочатку наповнити buffer різноманітними ситуаціями, а потім поступово покладатися на Q-мережу. Це типова інженерна практика для онлайн RL у середовищах з рідкісними позитивними нагородами.

РОЗДІЛ 3
РОЗРОБКА ТА ІМПЛЕМЕНТАЦІЯ РІШЕННЯ

3.1. Загальна архітектура програмної системи

Розроблена система складається з чотирьох логічних шарів. Перший шар – збір даних і підготовка навчальної вибірки за допомогою annotator. Другий шар – модуль сприйняття, який на основі скріншота відновлює матрицю дошки вісім на вісім і три матриці фігур; для фігур тут було використано CNN. Третій шар – модуль прийняття рішень на базі DQN, який формує тензор стану розміром вісім на вісім на чотири канали і обирає одну з легальних дій. Четвертий шар – модуль керування емулятором через ADB, який виконує свайп і обробляє перезапуск при завершення гри (game over).

Такий поділ дозволяє незалежно тренувати зоровий класифікатор offline, а політику DQN – online безпосередньо під час гри. Артефакти block_blast_recognizer.keras і class_dictionary.json стають стабільним інтерфейсом між двома етапами.

3.2. Програма annotator для збору навчальних даних

Оскільки готових публічних датасетів фігур Block Blast не існує, у роботі було розроблено власну програму анотації. У цьому коді реалізовано графічний інтерфейс на Tkinter, який одночасно показує три фрагменти панелі фігур і три сітки п'ять на п'ять для ручного відтворення форми кожного блоку.

Після натискання кнопки «Зробити скріншот» програма викликає функцію get_bluestacks_screenshot(), яка через ADB отримує повний кадр емулятора. Панель фігур обрізається за фіксованими координатами MAIN_PANEL_COORDS і ділиться на три зони однакової ширини. Користувач позначає заповнені клітинки у сітці синім кольором, після чого натискає «Зберегти дані».

Ключовою операцією підготовки міток є функція _trim_zeros, яка видаляє нульові рядки і стовпці по краях матриці п'ять на п'ять. Це відповідає тому, що у грі форма блоку задається мінімальним обмежувальним прямокутником без зайвих нульових контурів. Для кожної непорожньої фігури зберігаються два файли з однаковою базовою назвою {timestamp}_fig{i}: PNG-зображення та NPY-матриця.

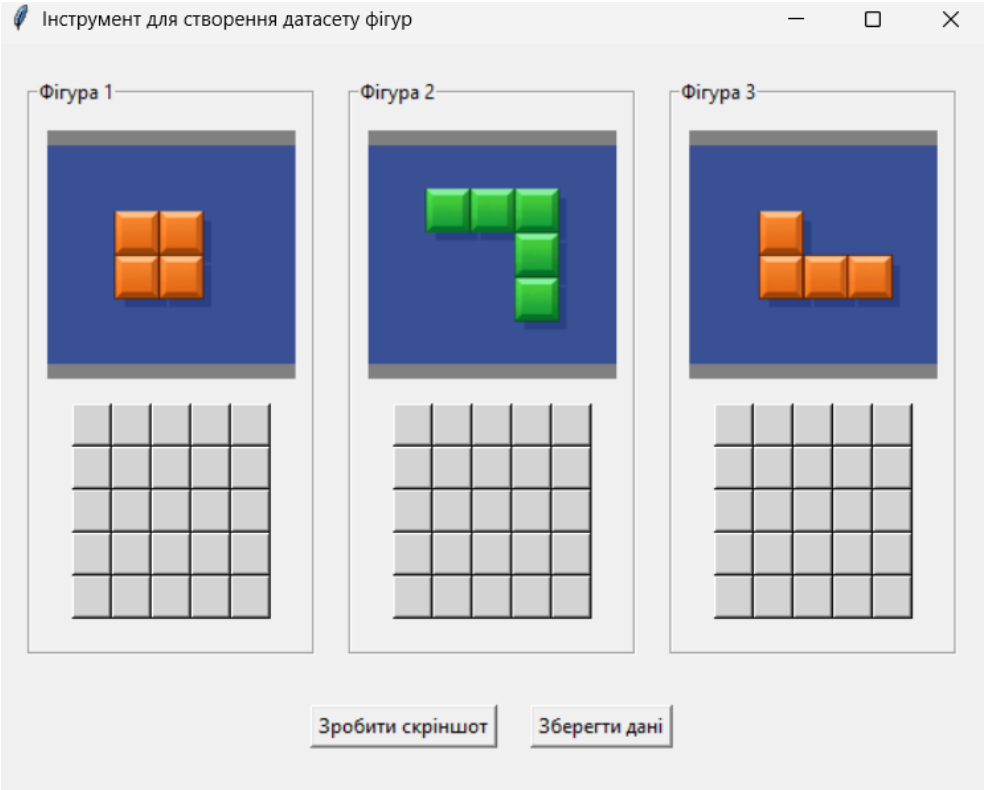

Рисунок 3.1 – Інтерфейс програми annotator для збору навчальної вибірки

Лістинг 3.1 Збереження пари «зображення – матриця» у annotator

```
def save_data(self):
    base_filename = int(time.time())
    for i in range(3):
        matrix_5x5 = np.zeros((GRID_SIZE, GRID_SIZE), dtype=int)
        for r in range(GRID_SIZE):
            for c in range(GRID_SIZE):
                if self.annotation_grids[i][r][c].cget("bg") ==
"blue":
                    matrix_5x5[r, c] = 1
        trimmed_matrix = self._trim_zeros(matrix_5x5)
        cv2.imwrite(img_path, cv2.cvtColor(self.piece_images[i],
cv2.COLOR_RGB2BGR))
        np.save(matrix_path, trimmed_matrix)
```

кінець лістингу 3.1

Практичний досвід показав, що якість набору даних напряму залежить від дисципліни анотації: навіть невелике зміщення форми в сітці створює новий «псевдоклас», тому під час збору даних доводилось дотримуватись єдиного правила вирівнювання блоків у шаблоні п'ять на п'ять.

Файли набору даних мають вигляд {unix_timestamp}_fig{i}.png і {unix_timestamp}_fig{i}.npy, де $i \in \{1,2,3\}$ відповідає слоту на панелі фігур. Групування за базовим ім'ям у load_data гарантує, що зображення і мітка не роз'єднуються. Пошкоджені PNG пропускаються з попередженням – це важливо під час тривалих сесій анотації, коли іноді можливий збій знімка екрана. Мітка часу у назві файлу забезпечує унікальність без окремих UUID-бібліотек.

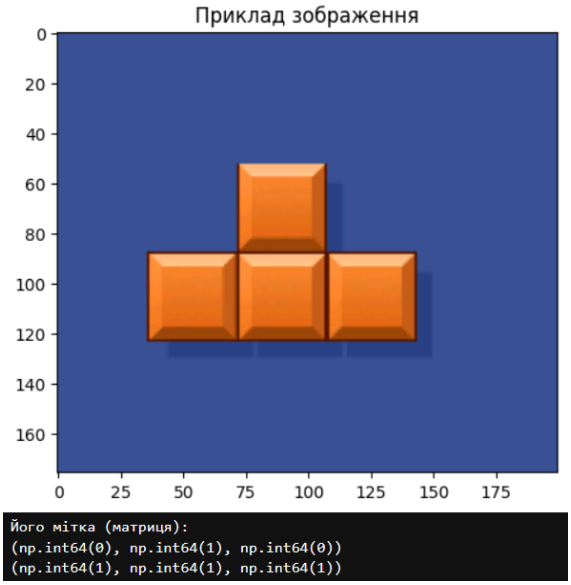

Рисунок 3.2 – Приклад пари «зображення фігури – бінарна матриця форми»

3.3. Попередня обробка даних та аугментація

Після завантаження набору даних усі зображення приводяться до розміру 176×200 пікселів методом cv2.resize з інтерполяцією INTER_AREA, що зменшує зубчастість при зменшенні масштабу. Піксельні значення нормалізуються діленням на 255, отримуючи діапазон [0,1].

Аугментація виконується у просторі HSV. Нехай (H, V) – канали відтінку і яскравості після перетворення з BGR. Тоді для кожної копії зображення генерують

$$H' = (H + \Delta h) \bmod 180, \quad V' = \text{clip}(V + \Delta v,\ 0,\ 255) \tag{3.1}$$

де $\Delta h \sim \mathcal{U}(-15{,}15)$ – випадковий зсув відтінку; $\Delta v \sim \mathcal{U}(-20{,}20)$ – випадковий зсув яскравості.

Спочатку рідкісні класи доповнюють копіями до частоти найчастішого класу, потім увесь збалансований набір збільшують у чотири рази шляхом повторної аугментації. У результаті обсяг даних зростає приблизно до 7040 зображень, що покращує стійкість CNN до варіацій кольору блоків.

Лістинг 3.2 HSV-аугментація та балансування класів перед навчанням CNN

```
def augment_image(image):
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    hsv[:, :, 0] = (hsv[:, :, 0].astype(int) + random.randint(-15,
15)) % 180
    hsv[:, :, 0] = hsv[:, :, 0].astype(np.uint8)
    v = np.clip(hsv[:, :, 2].astype(int) + random.randint(-20, 20),
0, 255)
    hsv[:, :, 2] = v.astype(np.uint8)
    return cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)

max_count = max(Counter(labels).values())
for label, original_imgs in class_images.items():
    balanced_images.extend(original_imgs)
    for _ in range(max_count - len(original_imgs)):

balanced_images.append(augment_image(random.choice(original_imgs)))
```

кінець лістингу 3.2

3.4. Архітектура та навчання CNN block_blast_recognizer

Ця згорткова нейронна мережа виконує роль класифікатора форм фігур. Архітектура побудована як Sequential-модель Keras із трьома блоками Conv2D–MaxPooling, повнозв'язним шаром на 128 нейронів, Dropout 0,5 і softmax-виходом на 40 класів. Загальна кількість параметрів становить близько 7,6 млн.

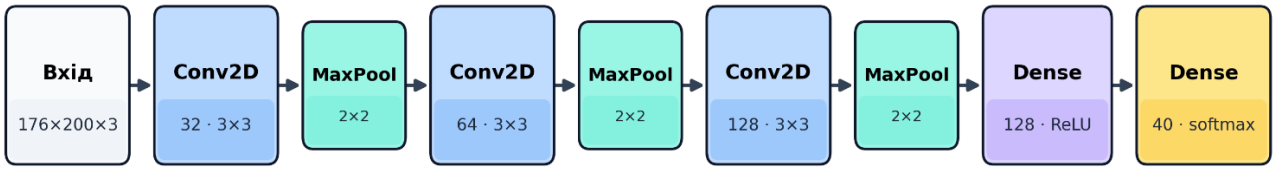

Рисунок 3.3 – Структурна схема згорткової мережі block_blast_recognizer

Лістинг 3.3 Архітектура згорткової мережі block_blast_recognizer

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu',
           input_shape=(IMG_HEIGHT, IMG_WIDTH, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax'),
])
```

кінець лістингу 3.3

Лістинг 3.4 Компіляція та навчання моделі розпізнавання фігур

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=25, batch_size=32,
                    validation_data=(X_val, y_val))
```

кінець лістингу 3.4

Навчання проводилося 25 епох з розміром пакета 32 і оптимізатором Adam. Стратифіковане розділення 80/20 зберіг пропорції класів у валідаційній вибірці: 5632 навчальних / 1408 валідаційних прикладів. Після навчання модель зберігають у форматі .keras, а словник int_to_class – у JSON для подальшого відновлення матриць фігур під час застосування моделі.

3.5. Розпізнавання стану дошки методами комп'ютерного зору

На відміну від фігур, стан дошки аналізується без нейронної мережі. Кадр обрізається за BOARD_CROP, після чого для кожної клітинки вісім на вісім беруть колір центрального пікселя. Найтемніший унікальний колір серед 64 зразків трактують як фон, а всі інші – як зайняті клітинки. Такий підхід простий і достатньо стійкий для Block Blast, де контраст між фоном решітки і блоками помітний.

Лістинг 3.5 Формування бінарної матриці дошки

```
def analyze_board_state(full_screenshot):
    b = full_screenshot[BOARD_CROP['y1']:BOARD_CROP['y2'],
                        BOARD_CROP['x1']:BOARD_CROP['x2']]
    # ... sampling center pixels ...
    bg = min(unique, key=lambda c: int(np.sum(c.astype('int32'))))
    for r, c, color in samples:
        if not np.array_equal(color, bg):
            m[r, c] = 1
    return m, b
```

кінець лістингу 3.5

Для зменшення артефактів анімації після ходу застосовано двокадрове підтвердження: два скріншоти з інтервалом 0,12 с порівнюються, і якщо дошка змінилась, беруть останній кадр.

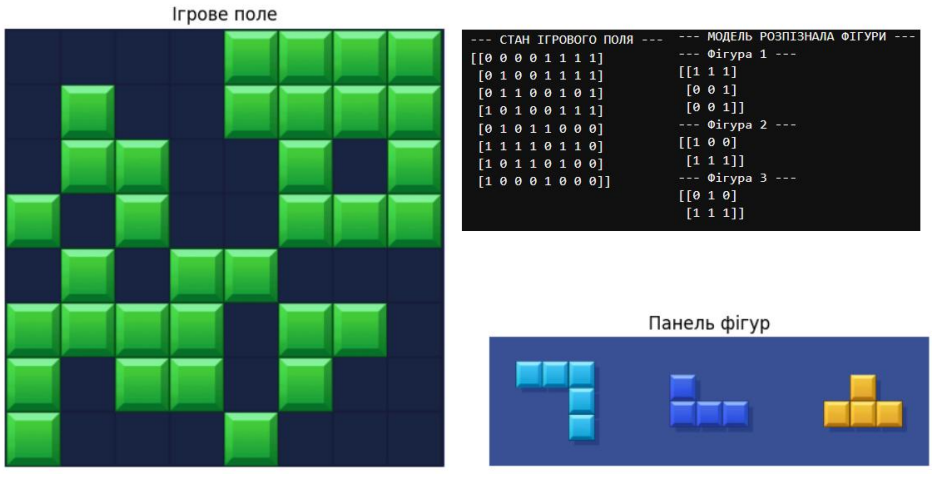

Рисунок 3.4 – Приклад розпізнаної дошки 8×8 та панелі фігур

3.6. Реалізація середовища BlockBlastEnv та DQN-агента

Клас BlockBlastEnv інкапсулює цикл «спостереження – дія – нагорода». Метод get_state() повертає дошку і три фігури або None при невдалому зчитуванні. Метод step(action) виконує свайп через ADB, чекає завершення анімації, знову зчитує стан і обчислює нагороду через compute_reward.

Вхід DQN формує функція state_to_input: канал 0 – дошка, канали 1–3 – три фігури, розміщені у верхньому лівому куті своїх каналів. Архітектура DQNet містить два згорткові шари (32 і 64 фільтри), шар Dense на 512 нейронів і лінійний вихід довжини 192.

Лістинг 3.6 Формування тензора стану для Q-мережі

```
def state_to_input(board, pieces):
    ch = np.zeros((4, 8, 8), dtype=np.float32)
    ch[0] = board
    for i in range(3):
        p = _norm_piece(pieces[i])
        h, w = p.shape
        ch[i + 1, :h, :w] = p
    return np.transpose(ch, (1, 2, 0))
```

кінець лістингу 3.6

Лістинг 3.7 Архітектура Deep Q-Network (клас DQNet)

```
class DQNet(tf.keras.Model):
    def __init__(self, action_dim):
        super().__init__()
        self.c1 = tf.keras.layers.Conv2D(32, 3, padding='same',
activation='relu')
        self.c2 = tf.keras.layers.Conv2D(64, 3, padding='same',
activation='relu')
        self.f = tf.keras.layers.Flatten()
        self.d1 = tf.keras.layers.Dense(512, activation='relu')
        self.o = tf.keras.layers.Dense(action_dim)

    def call(self, x, training=False):
        x = self.c1(x)
        x = self.c2(x)
        x = self.f(x)
        x = self.d1(x)
        return self.o(x)
```

кінець лістингу 3.7

Функція compute_reward реалізує формулу (2.10): нагорода за розміщення, очищення ліній, штрафи за хибний свайп, «замкнені дірки» та завершення гри.

Лістинг 3.8 Обчислення функції нагороди за крок

```
def compute_reward(before_board, after_board, lines_cleared, done,
                   missed_placement=False, w=RewardWeights()):
    feedback = {}
    if missed_placement:
        feedback['false_swipe_penalty'] = -w.false_swipe_penalty
    else:
        feedback['placement_reward'] = w.placement_reward
        if lines_cleared > 0:
            feedback['line_clear_reward'] = w.line_clear_reward *
lines_cleared
            if lines_cleared > 1:
                feedback['multi_line_bonus'] = w.multi_line_bonus
* (lines_cleared - 1)
        new_holes = max(0, count_enclosed_empty_cells(after_board)
                        -  count_enclosed_empty_cells(before_board))
        if new_holes > 0:
            feedback['enclosed_hole_penalty'] = -
w.enclosed_hole_penalty * new_holes
    if done:
        feedback['terminal_penalty'] = -w.terminal_penalty
    return float(sum(feedback.values())), feedback
```

кінець лістингу 3.8

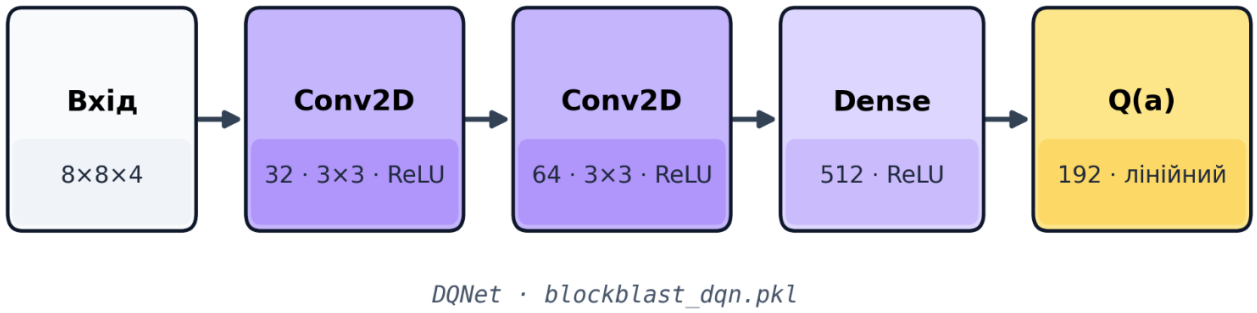

Рисунок 3.5 – Архітектура Deep Q-Network для Block Blast

Лістинг 3.9 Вибір дії з маскуванням легальних ходів

```
def select(self, s, legal_actions):
    m = self.legal_mask(legal_actions)
    ids = np.where(m > 0)[0]
    if random.random() < self.eps():
        return self.i2a(int(np.random.choice(ids))), m
    q = self.policy(s[None, ...], training=False).numpy()[0]
    q[m == 0] = -1e9
    return self.i2a(int(np.argmax(q))), m
```

кінець лістингу 3.9

Лістинг 3.10 Оновлення Q-мережі методом DQN (функція train_step)

```
def train_step(self):
    if len(self.replay) < max(self.warmup, self.batch_size):
        return None
    b = self.replay.sample(self.batch_size)
    s = np.stack([x.state for x in b]).astype(np.float32)
    a = np.array([x.action for x in b], dtype=np.int32)
    r = np.array([x.reward for x in b], dtype=np.float32)
    ns = np.stack([x.next_state for x in b]).astype(np.float32)
    d = np.array([x.done for x in b], dtype=np.float32)
    nm = np.stack([x.next_mask for x in b]).astype(np.float32)
    nq = self.target(ns, training=False).numpy()
    nq[nm == 0] = -1e9
    nv = np.max(nq, axis=1)
    y = r + (1.0 - d) * self.gamma * nv
    with tf.GradientTape() as tape:
        q = self.policy(s, training=True)
        idx = tf.stack([tf.range(tf.shape(q)[0]), a], axis=1)
        q_sel = tf.gather_nd(q, idx)
        loss = tf.keras.losses.Huber()(y, q_sel)
    g = tape.gradient(loss, self.policy.trainable_variables)
    g, _ = tf.clip_by_global_norm(g, 5.0)
    self.opt.apply_gradients(zip(g,
self.policy.trainable_variables))
    if self.step_n % self.target_sync == 0:
        self.target.set_weights(self.policy.get_weights())
    return float(loss.numpy())
```

кінець лістингу 3.10

Replay buffer зберігає кортежі (state, action, reward, next_state, done, next_mask). Навчання стартує після прогріву 500 кроків, розмір пакета (batch size) дорівнює 64, градієнти обрізаються за global norm 5.

Життєвий цикл одного кроку агента у live-режимі включає послідовність: get_state виконує подвійний скріншот і формує (board, pieces); get_legal_actions перебирає до 192 кандидатів, хоча фактична кількість легальних ходів зазвичай менша; state_to_input збирає тензор float32; agent.select з імовірністю ε обирає випадкову легальну дію, інакше – argmax Q по маскованих значеннях. Далі place_piece_via_swipe виконує ADB swipe, після затримок знову зчитується стан, detect_cleared_lines порівнює дошки, compute_reward підсумовує зважені компоненти. Перехід додається у replay buffer, train_step за потреби оновлює ваги, target-мережа синхронізується кожні 200 кроків. Цикл повторюється до завершення епізоду або досягнення ліміту дій.

3.7. ADB-автоматизація та калібрування координат

Функція place_piece_via_swipe обчислює початок свайпу в центрі слота фігури і кінець з урахуванням swipe_gain та геометричного центру фігури відносно цільової клітинки. Якщо підряд фіксується серія невдалих розміщень, failsafe-механізм додає випадковий jitter до координат і невеликий шум до gain, що допомагає вийти з локальних неточностей калібрування.

Режим dry_run=True пропускає лише виконання ADB-команд свайпу (input swipe), але зчитування стану гри через screencap залишається обов'язковим – емулятор має бути запущений з відкритою грою та доступним ADB-з'єднанням. Це дозволяє перевірити логіку вибору дії, нагород, replay buffer і train_step без ризику змінити поточну позицію на дошці. Функція ensure_adb_device перевіряє доступність hd-adb і при потребі перепідключається до порту 127.0.0.1:5555. Координати BOARD_CROP і PIECE_SLOTS масштабуються лінійно від еталонного кадру 720×1280, що дозволяє змінювати розмір вікна емулятора без повної ручної перекалібровки. Failsafe поступово збільшує jitter лише після серії невдач, щоб не руйнувати точність у нормальному режимі.

3.8. Проблеми, виявлені під час розробки

Перша суттєва проблема стосувалася автоматичного виділення фігур через k-means. При зміні палітри блоків між рівнями сегментація давала фрагментовані маски, і матриця форми не збігалася з реальністю. Перехід на CNN з власним набором даних вирішив цю проблему на рівні класифікації, хоча потребував ручної праці з annotator.

Друга група проблем пов'язана з ADB. На BlueStacks періодично виникають повідомлення error: close і device offline. У коді реалізовано _adb_reconnect() і повторні спроби shell-команд, проте в live-режимі окремі епізоди все одно перериваються технічними збоями.

Третя проблема – фізика свайпу. Без коефіцієнта gain блок часто «перелітає» цільову клітинку. Навіть після калібрування точність залежить від роздільності вікна, тому застосовано failsafe з наростаючим jitter.

Четверта проблема – часовий шум зору. Якщо робити скріншот занадто рано після свайпу, на дошці видно тимчасове зображення фігури над полем, і алгоритм помилково вважає розміщення невдалим, застосовуючи штраф за хибний свайп. Додаткова затримка post_placement_check_delay_sec зменшила частоту таких помилок.

П'ята проблема – «холодний старт» DQN. На початку онлайн-навчання агент має випадкову політику і часто швидко отримує завершення гри (game over) з великим негативним значенням нагороди. Це очікувана поведінка для фази досліджувального вибору, але вимагає тривалого навчання для помітного покращення. Архітектура системи (рис. 3.1) свідомо модульна: annotator і CNN можна розробляти офлайн, DQN – онлайн. Такий поділ полегшує налагодження – якщо агент робить нелегальні розміщення, спочатку перевіряють CNN і class_dictionary.json, потім генератор легальних дій, і лише наприкінці Q-значення.

Annotator реалізовано як однофайловий GUI-додаток без зайвих залежностей. Клас AnnotationApp інкапсулює три паралельні панелі анотації,

що відповідають інтерфейсу гри: гравець одночасно бачить три фігури і має швидко їх відтворити. Мітка часу у імені файлу гарантує унікальність без UUID-бібліотек.

Операція _trim_zeros має просту математику: для матриці M знаходять множини рядків і стовпців $R = \{i \mid \exists j: M_{ij} = 1\}$, $C = \{j \mid \exists i: M_{ij} = 1\}$ і повертають підматрицю $M[R, C]$. Це канонізація форми, що усуває артефакти доповнення анотації.

У CNN кожен блок Conv2D+MaxPooling зменшує просторову роздільність і збільшує глибину ознак. Після трьох блоків карта ознак перетворюється на вектор довжини 128 через Dense+Dropout, а softmax дає розподіл по 40 класах. Dropout 0,5 зменшує спільну адаптацію нейронів під час тренування.

Розпізнавання дошки свідомо залишено простим, бо контраст блок/фон на полі вісім на вісім стабільніший, ніж на панелі фігур з градієнтами. Центральний піксель клітинки – компроміс між стійкістю до шуму і простотою; альтернатива медіани по фрагменту потребувала б додаткового підбору.

BlockBlastEnv.step – найскладніша процедурно частина системи: вона поєднує виконання дії, часову затримку, повторне зчитування стану, детекцію очищених ліній і формування нагороди. Очищення ліній визначають не симуляцією «як би могло бути», а порівнянням дошок до і після ходу через detect_cleared_lines, що відповідає реальній фізиці гри.

Вхід DQN 8×8×4 можна уявити як багатоканальне зображення, де канали 1–3 – бінарні маски фігур із прив’язкою до лівого верхнього кута. Це кодування не інваріантне до повороту, але для Block Blast орієнтація фігур фіксована інтерфейсом, тому достатньо.

ADB failsafe поступово збільшує jitter лише після серії невдач, щоб не руйнувати точність у нормальному режимі. У коді dry_run реалізовано як пропуск shell-команд свайпу в _adb_shell, тоді як get_bluestacks_screenshot викликається незалежно від цього прапорця. Тому dry_run не працює без емулятора: для кожного кроку потрібен ADB screenshot, інакше зоровий модуль не отримає стан гри.

Серед проблем розробки варто окремо згадати дисбаланс класів до аугментації: деякі рідкісні форми з’являлись у наборі даних лише кілька разів. Без балансування CNN зміщувалася до частих форм. Адаптивна аугментація практично вирівняла гістограму класів перед загальною x4-аугментацією.

Ще одна проблема – детекція завершення гри (game over) через зелену кнопку перезапуску: HSV-пороги працюють, але при зміні теми інтерфейсу (UI) можуть потребувати перекалібрування. Резервний механізм через no_legal_streak >= 2 зменшує ризик зависання агента на екрані тупику. У процесі розробки суттєвий час зайняло налаштування swipe_gain. Без нього блоки систематично «перелітають» цільову клітинку, що з точки зору RL виглядає як шум у функції переходу. Failsafe з наростаючим jitter додано як другий рівень захисту після серії невдалих розміщень.

Програмна реалізація, описана у цьому розділі, узгоджена між собою: лістинг 3.1 відображає ключову логіку збереження даних у annotator, лістинги 3.2–3.4 – підготовку та навчання CNN, лістинг 3.5 – розпізнавання дошки, лістинги 3.6–3.10 – формування стану, нагороди, архітектуру DQN, вибір дії та крок навчання Q-мережі.

РОЗДІЛ 4
ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1. Налаштування обчислювальних експериментів

Експериментальне дослідження поділено на два блоки: офлайн-навчання CNN і онлайн-навчання DQN безпосередньо в емуляторі. Обчислення виконувалися на типовому ПК без виділеного GPU: для CNN і DQN у конфігурації роботи цього достатньо. Тренування CNN займає десятки хвилин залежно від CPU, тоді як онлайн DQN може тривати години при десятках live-епізодів поспіль.

Для CNN використано стратифіковане розділення train/validation 80/20 з random_state=42, 25 епох, batch 32, оптимізатор Adam. Фіксація random_state забезпечує відтворюваність розділення вибірки при повторному навчанні; 25 епох обрано емпірично, оскільки криві навчання стабілізувалися до 20–25 епох без явного розриву перенавчання на валідації. Для DQN зафіксовано $\gamma = 0.99$, швидкість навчання 10^{-4}, replay capacity 50 000, прогрів 500 кроків, синхронізацію target-мережі кожні 200 кроків, максимум 250 дій на епізод. Checkpoint кожні 5 епізодів зберігає прогрес навіть при збої ADB.

Перед live-запуском виконується quick_start(): завантаження CNN, автокалібрація координат, перевірка ADB і підрахунок легальних дій у поточному стані. Якщо check_bot_status() повертає BOT STATUS: WORKING, дозволено переходити до run_online_training(dry_run=False). Спочатку рекомендується dry_run для перевірки логіки BlockBlastEnv, replay buffer і розкладу ε без ризику некоректних жестів.

4.2. Результати навчання згорткової мережі

Після 25 епох навчання точність на тренувальній вибірці сягала близько 99%, а на валідаційній – до 100%. Такий результат свідчить про те, що при достатній аугментації задача розпізнавання 40 форм у фіксованому ROI панелі фігур є добре умовленою. Водночас варто зазначити ризик перенавчання до кольорових патернів конкретного емулятора, тому доцільно розширити набір даних новими скріншотами.

Таблиця 4.1 – Метрики якості CNN block_blast_recognizer

Метрика	Навчання	Валідація
Точність (accuracy)	~99%	~100%
Функція втрат (cross-entropy)	низька, спадна	стабільна
Кількість класів	40	40
Розмір вибірки	5632	1408

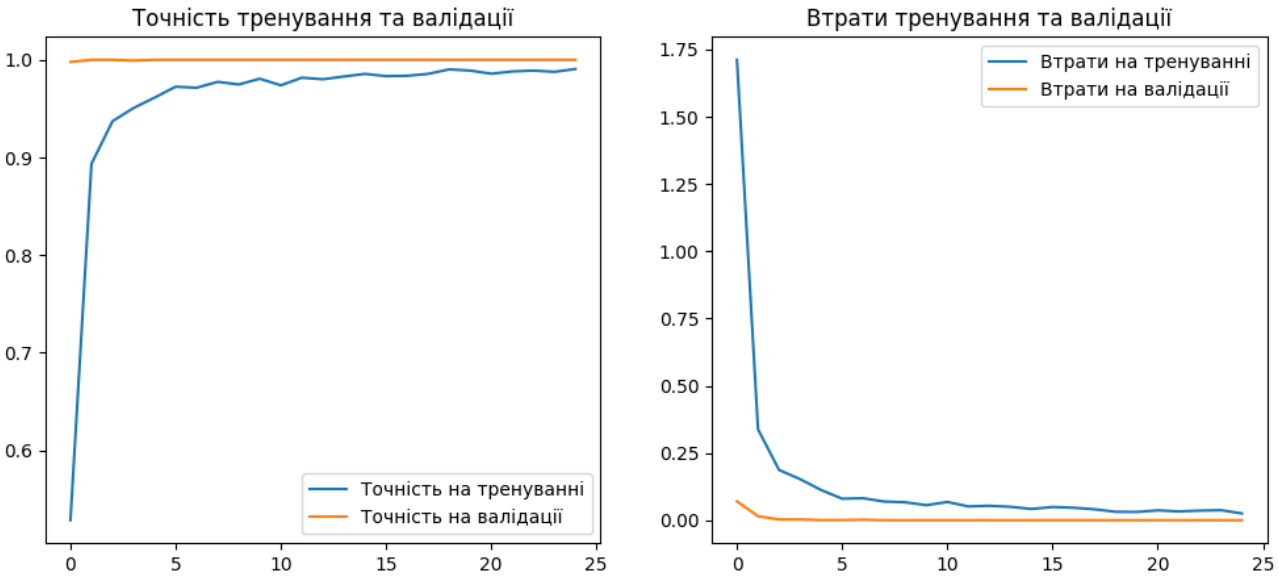

Рисунок 4.1 – Криві точності та функції втрат під час навчання CNN

На рисунку має бути зображено: два графіки точності (accuracy) та функції втрат (loss) для навчальної та валідаційної вибірок по епохах.

4.3. Результати онлайн-навчання DQN та порівняння режимів dry_run і live

Онлайн-експерименти проводилися у двох режимах: dry_run (без реальних свайпів, але зі зчитуванням стану через ADB) і live. У live-режимі типова сумарна нагорода епізоду на ранніх епізодах залишалась негативною через штрафи за завершення гри (game over), хибний свайп і нелегальні дії. Кількість очищених ліній часто дорівнювала нулю, що відповідає початковій фазі досліджувального вибору, коли агент ще не навчився цілеспрямовано очищати лінії. Негативні значення сумарної нагороди на ранній стадії не означають повну несправність ланцюжка – вони очікувані, коли ε велике і політика близька до випадкової.

Проте навіть за обмеженої кількості епізодів replay buffer наповнюється реальними переходами, а графік loss демонструє, що Q-мережа оновлюється без числових розходжень. Після кількох збережень контрольних точок (checkpoint) blockblast_dqn.pkl можливе продовження навчання з resume=True. Сумарна нагорода епізоду – сума нагород за епізод, не нормована на крок; негативні значення домінують на ранньому навчанні через штрафи за завершення гри (−25) і хибний свайп (−8). Метрика очищень – ціла кількість очищених ліній, визначених комп'ютерним зором між кадрами, а не «оракулом» симулятора.

Режим dry_run корисний для перевірки логіки середовища, replay buffer і розкладу ε без ризику некоректних жестів. Оскільки без свайпів дошка на екрані не змінюється, такий режим не відтворює повноцінну динаміку гри і слугує переважно для налагодження програмного ланцюжка RL. Остаточна оцінка агента можлива лише в live-режимі, де додатково виникають шум ADB, затримки анімації і помилки свайпу, які суттєво впливають на розподіл нагород. Саме тому failsafe і swipe_gain є частиною експериментальної установки, а не другорядними константами.

Таблиця 4.2 – Приклад статистики live-навчання DQN (фрагмент)

Епізод	Нагорода	Очищення	ε	Фаза
1–5	від −8 до −25	0–1	~0,98	дослідж. вибір
10–15	від −15 до −40	0	~0,85	дослідж. вибір
18	−192	0	~0,35	зміщення моделі

У таблиці 4.2 наведено ілюстративний фрагмент live-логів, зокрема епізод із сумарною нагородою -192, який показує накопичення штрафів при тривалому епізоді без успішних очищень. Обмеження max_actions_per_episode = 250 запобігає нескінченним циклам при збоях зору.

4.4. Узагальнення результатів, інтерпретація метрик та обмеження дослідження

Експерименти підтвердили, що поєднання CNN для зору і DQN для стратегії є працездатною архітектурою для Block Blast. CNN забезпечує надійне відновлення форм фігур, а DQN формує базу для подальшого поліпшення політики через тривале онлайн-навчання. Валідаційна точність (accuracy) близька до 100% означає, що модель майже бездоганно класифікує відкладену вибірку після аугментації; водночас узагальнення на повністю нові колірні теми не гарантоване. Матриця плутанини теоретично може виявити систематичні помилки між дзеркальними формами – її доцільно побудувати засобами sklearn.metrics.confusion_matrix при оформленні остаточної версії роботи.

Розклад ε-greedy з фазами досліджувального вибору і подальшим експоненційним спадом допомагає інтерпретувати, чи агент ще активно досліджує простір дій, чи вже покладається на Q-мережу. Головним обмеженням залишається не стільки математична модель, скільки інженерна стабільність каналу ADB і якість формування нагороди у live-середовищі.

Дослідження обмежене однією грою, одним емулятором і одним профілем роздільності екрана. Перенесення рішення на реальний смартфон потребувало б повторної калібрації координат. Обсяг онлайн-навчання DQN у експериментах

становив десятки епізодів, що недостатньо для досягнення рівня людини, але достатньо для підтвердження працездатності навчального циклу. Сітковий пошук гіперпараметрів DQN не проводився – значення підібрані емпірично в процесі розробки. Статистичну значущість результатів DQN за одним прогоном не оцінювали: дисперсія RL висока, тому для сильнішого висновку потрібні повторні запуски з різними значеннями seed.

З прикладної точки зору робота демонструє, що навіть при обмеженому обчислювальному бюджеті можна побудувати end-to-end автономного агента для комерційної мобільної гри, якщо правильно декомпозувати задачу на сприйняття, прийняття рішень і виконання дій. У контексті кафедри прикладної математики ЛНТУ особливу цінність має явний зв'язок між формулами (1.1)–(2.10) і програмною реалізацією: можна простежити шлях від рівняння Беллмана до конкретного обчислення функції втрат у методі train_step. Окремо варто наголосити, що висока точність CNN на валідації не знімає потреби перевірки на нових скріншотах з іншими кольоровими темами блоків. Для DQN важливіше стабільне виконання окремих компонентів (quick_start, застосування CNN, підрахунок легальних дій), ніж абсолютне значення сумарної нагороди на ранніх епізодах – саме ці компоненти в експериментах працювали послідовно. При оформленні остаточної версії роботи доцільно доповнити розділ 4 графіками з реальних логів навчання: кривими accuracy/loss для CNN та динамікою нагороди по епізодах для DQN. Такі ілюстрації безпосередньо підтверджують висновки про стабільність навчального циклу.

ВИСНОВКИ

У кваліфікаційній роботі виконано розробку та експериментальне дослідження автономного інтелектуального агента для гри Block Blast у віртуальному середовищі Android-емулятора. Поставлену мету досягнуто: створено цілісну систему від збору даних до онлайн-навчання політики методом Deep Q-Network.

Проаналізовано предметну область гри Block Blast, її правила та математичну модель стану у вигляді бінарної дошки вісім на вісім і трьох матриць фігур. Установлено, що простір дій до 192 комбінацій на кожному кроці можна скоротити до множини легальних ходів, що є необхідною умовою ефективного навчання з підкріпленням.

Розроблено програму annotator для формування навчальної вибірки пар «зображення – матриця форми». Зібрано 503 приклади, що відповідають 40 унікальним геометричним класам блоків, що достатньо для навчання згорткового класифікатора.

Побудовано та навчено згорткову нейронну мережу block_blast_recognizer. Після балансування класів і HSV-аугментації модель досягла високої точності на валідаційній вибірці і замінила ненадійний класичний підхід k-means у модулі розпізнавання фігур.

Реалізовано ігрове середовище BlockBlastEnv з явною функцією нагороди, двокадровим підтвердженням стану, детекцією завершення гри та failsafe-корекцією свайпів. Інтегровано керування через Android Debug Bridge з автоматичною калібрацією координат.

Реалізовано DQN-агента з replay buffer, target-мережею, маскуванням легальних дій і двофазним розкладом ε -greedy. Проведено онлайн-експерименти, які підтвердили працездатність замкненого циклу «спостереження – дія – нагорода – навчання».

Експериментально доведено, що поєднання CNN для зорового сприйняття і DQN для прийняття рішень є обґрунтованим підходом до автоматизації гри

Block Blast у умовах емулятора. Отримані результати підтверджують коректність обраних математичних моделей та програмної реалізації.

З точки зору прикладної математики робота демонструє застосування формалізму марковського процесу прийняття рішень, рівняння Беллмана, функції перехресної ентропії та Q-навчання у конкретній інженерній задачі з дискретним простором станів і дій.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

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

2. Mnih V., Kavukcuoglu K., Silver D. et al. Human-level control through deep reinforcement learning // Nature. 2015. Vol. 518. P. 529–533.

3. Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge : MIT Press, 2016. 800 p.

4. Chollet F. Deep Learning with Python. 2nd ed. Shelter Island : Manning, 2021. 504 p.

5. Silver D., Huang A., Maddison C. J. et al. Mastering the game of Go with deep neural networks and tree search // Nature. 2016. Vol. 529. P. 484–489.

6. TensorFlow Developers. TensorFlow Documentation [Електронний ресурс]. URL: https://www.tensorflow.org/api_docs (дата звернення: 24.04.2026).

7. OpenCV Team. OpenCV Documentation [Електронний ресурс]. URL: https://docs.opencv.org/ (дата звернення: 23.04.2026).

8. Keras Team. Keras Documentation [Електронний ресурс]. URL: https://keras.io/api/ (дата звернення: 23.04.2026).

9. Pedregosa F. et al. Scikit-learn: Machine Learning in Python // Journal of Machine Learning Research. 2011. Vol. 12. P. 2825–2830.

10. Приходько О. С., Вавринюк К. В. Кваліфікаційна робота: методичні вказівки до оформлення кваліфікаційних робіт для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Штучний інтелект та аналіз масивів даних». Луцьк : ЛНТУ, 2026. 40 с.

11. Van Rossum G., Drake F. L. The Python Language Reference [Електронний ресурс]. URL: https://docs.python.org/3/ (дата звернення: 22.04.2026).

12. Harris C. R. et al. Array programming with NumPy // Nature. 2020. Vol. 585. P. 357–362.

13. Hunter J. D. Matplotlib: A 2D Graphics Environment // Computing in Science & Engineering. 2007. Vol. 9. № 3. P. 90–95.

14. Android Developers. Android Debug Bridge (adb) [Електронний ресурс]. URL: https://developer.android.com/tools/adb (дата звернення: 12.05.2026).

15. Clark A. Pillow (PIL Fork) Documentation [Електронний ресурс]. URL: https://pillow.readthedocs.io/ (дата звернення: 25.04.2026).

16. BlueStacks. Enable Android Debug Bridge [Електронний ресурс]. URL: https://support.bluestacks.com/ (дата звернення: 12.05.2026).

17. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 4th ed. Hoboken : Pearson, 2021. 1136 p.

18. Géron A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. 3rd ed. Sebastopol : O'Reilly Media, 2022. 856 p.

19. Kingma D. P., Ba J. Adam: A Method for Stochastic Optimization // Proc. ICLR. 2015.

20. LeCun Y., Bengio Y., Hinton G. Deep learning // Nature. 2015. Vol. 521. P. 436–444.

21. Krizhevsky A., Sutskever I., Hinton G. E. ImageNet Classification with Deep Convolutional Neural Networks // Advances in Neural Information Processing Systems. 2012. P. 1097–1105.

22. ДСТУ 3008:2015. Документація. Звіти у сфері науки і техніки. Структура і правила оформлення. [Чинний від 2016-07-01]. Київ, 2016