Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет Комп'ютерні технології та системи
Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка

до кваліфікаційної роботи
ОС магістр

на тему: «Дослідження методів навчання нейронних мереж при моделюванні поведінки ігрових агентів»

за освітньою програмою Інженерія програмного забезпечення
зі спеціальності: 121 Інженерія програмного забезпечення

Виконав: студент групи ПЗ2321: / Кирило ЯРОВИЙ /

Керівник: / Вадим ГОРЯЧКІН /

Нормоконтролер: / Світлана ВОЛКОВА /

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

Студент

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems
Department Computer information technology

Explanatory Note

to Master's Thesis

on the topic: «Research on methods of neural network training in modeling the behavior of game agents»

according to educational curriculum **Software engineering**
in the Speciality: **121 software engineering**

Done by the student of the group PZ2321: _____ / Kyrylo YAROVYI /

Scientific Supervisor: _____ / Vadim HORYACHKIN /

Normative controller: _____ / Svitlana VOLKOVA /

Dnipro – 2025

Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет: Комп'ютерних технологій і систем

Кафедра: Комп’ютерні інформаційні технології

Рівень вищої освіти: магістр

Освітня програма: **12** Інженерія програмного забезпечення

Спеціальність: **121** Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ____КІТ
_____ Вадим ГОРЯЧКІН
_____ _____ 202__ р.

ЗАВДАННЯ

На кваліфікаційну роботу ____Магістр_____

студенту Яровому Кирилу Вікторовичу

1. Тема дипломної роботи: Дослідження методів навчання нейронних мереж при моделюванні поведінки ігрових агентів.
   Керівник роботи: Горячкін Вадим Миколайович
   затверджені наказом 1256 ст від 07.10.2024 року
2. Строк подання студентом роботи 08.01.2025 року
3. Вихідні дані до дипломної роботи: програмний продукт.
4. Зміст пояснювальної записки (перелік питань до розробки):
   4.1. Аналітична частина: огляд предметної галузі;
   4.2. Основна частина: опис моделі, опис агентів, опис дослідження;
   4.3. Експерименти та висновки.
5. Перелік демонстраційного матеріалу:
   5.1. презентація;

5.2. демонстраційне відео.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ	16.05.2024	
2	Аналіз сучасного стану дослідження за науковими літературними статтями та дослідженнями	02.06.2024	
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує інтеграції в середовище для вирішення поставленої задачі	02.08.2024	
4	Постановка задачі та технічне завдання	01.11.2024	30%
5	Техніко-економічні показники	02.11.2024	
6	Розробка інструментальних засобів дослідження	13.11.2024	
7	Виконання досліджень	02.12.2024	60%
8	Оформлення тез доповідей	14.12.2024	
9	Оформлення статті у фаховий журнал	18.12.2024	
10	Оформлення пояснювальної записки	26.12.2024	
11	Розробка демонстраційних матеріалів	19.12.2024	
12	Подання кваліфікаційної роботи до кафедри	08.01.2025	100%
13	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	22.01.2025	

Студент _____ / Кирило ЯРОВИЙ /

Керівник роботи _____ /Вадим ГОРЯКІН/

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра:

105 с., 54 рис., 4 додатки, 12 джерел.

Об’єктом дослідження є два підходи у нейронних мережах, які використовуються для навчання моделей приймати рішення.

Метою дослідження є знаходження найбільш оптимального підходу навчання для створення ігрового агента, який буде приймати рішення.

Методи дослідження: навчання нейронних мереж за допомогою двох підходів під час створення ігрового бота.

Пояснювальна записка включає вступ, три розділи, висновки, бібліографічний список та додатки.

У вступі викладено суть, мету та актуальність роботи (2 сторінок).

Перший розділ присвячений аналізу сучасного стану предметної області (4 сторінки).

Другий розділ охоплює етапи проектування (14 сторінок).

Третій розділ описує процес тестування та відлагодження (14 сторінок).

Додатки включають листи затвердження, технічне завдання, текст програми, керівництво користувача та тези.

Ключові слова: модель, нейронна мережа, нейрон, ваги, вхідний/вихідний шар, Supervised Learning, Reinforcement Learning.

ЗМІСТ

ВСТУП

Актуальність роботи. В сучасному світі нейронні мережі [1] почали стрімко завойовувати технологічний світ, перш за все, завдяки своїй унікальній здатності адаптуватися та навчатися. Інтерес також стимулюється доступністю потужного обладнання, хмарних платформ і відкритих бібліотек, що робить їх використання більш доступним для розробників та компаній. Такі технології дозволили відійти від написання складних алгоритмів та надати можливість нейронним мережам проявити себе в розпізнаванні образів.

Моделювання поведінки ігрових агентів є важливим напрямом досліджень у галузі штучного інтелекту. Завдяки прогресу в технологіях машинного навчання, стало можливим створення агентів, здатних демонструвати складну адаптивну поведінку у віртуальних середовищах. Створенні моделі знаходять застосування у різних сферах: від розробки відеоігор до тестування складних систем та вирішення реальних задач.

Одним із ключових аспектів розробки таких систем є вибір ефективних методів навчання нейронних мереж. Вони визначають здатність агента адаптуватися до змінюваних умов, приймати оптимальні рішення в умовах невизначеності та вчитися на власних помилках.

Мета роботи. Дослідження та аналіз сучасних методів навчання нейронних мереж для моделювання поведінки ігрових агентів, з акцентом на вивчення їх ефективності, адаптивності та застосування. Робота спрямована на визначення оптимальних підходів до навчання агентів, які забезпечують високу якість прийняття рішень, здатність до самонавчання та адаптації до динамічних умов.

Експлуатаційне призначення. Результати дослідження можуть бути використані в різних сферах, пов'язаних із розробкою та впровадженням інтелектуальних агентів. Це допоможе створювати більш реалістичних і адаптивних NPC, які зможуть підвищити занурення гравців. Розробляти ігрових ботів для тестування ігор та автоматизації ігрових сценаріїв. Використання ігрових агентів у навчальних симуляціях, де моделюються складні сценарії для

підготовки фахівців. Перенесення моделей поведінки ігрових агентів на фізичних роботів для виконання реальних задач, таких як навігація, співпраця чи взаємодія з людьми. Моделювання процесів, які вимагають прийняття рішень в умовах невизначеності. Таким чином, експлуатаційне призначення дослідження спрямоване на створення гнучких, інтелектуальних систем, здатних вирішувати складні задачі в ігрових, симуляційних та реальних середовищах.

1. Огляд предметної галузі

Нейронні мережі є надзвичайно популярною, перспективною та цікавою галуззю штучного інтелекту, яка була натхненна будовою та принципом функціонування людського мозку. Ці складні математичні моделі складаються з взаємопов'язаних штучних нейронів, організованих у шари, які здатні навчатися та розпізнавати складні взаємозв’язки в даних.

Основна архітектура нейронних мереж включає вхідний шар, один або кілька прихованих шарів та вихідний шар. Кожен нейрон отримує вхідні сигнали, обробляє їх за допомогою активаційних функцій та передає результат далі. Процес навчання відбувається через коригування вагових коефіцієнтів зв'язків між нейронами, що дозволяє мережі поступово покращувати точність передбачень.

Сьогодні нейронні мережі застосовуються в багатьох галузях, а саме: від розпізнавання зображень і мови до медичної діагностики, фінансового прогнозування та автономного керування транспортними засобами. Вони стали потужним інструментом розв'язання дуже складних завдань, які ще донедавна здавалися неможливими.

У сфері нейронних мереж лідерами є такі компанії як Google DeepMind, OpenAI, NVIDIA та Meta AI. Кожна з цих компаній займається вдосконаленням нейронних мереж від апаратного рівня, де створюють потужну інфраструктуру для навчання мереж, до комп’ютерного зору з подальшим розвиненням практичного застосування технології. Провідні технологічні компанії світу продовжують інвестувати величезні ресурси в розвиток цієї технології, що свідчить про її визначальну роль у формуванні майбутнього штучного інтелекту.

1.1 Огляд сучасних моделей

Машинне навчання має при собі різні підходи до розв'язання завдань. Одними з найпопулярніших є методи навчання Supervised Learning [2] та Reinforcement Learning [3]. Ці підходи спрямовані на створення моделей, які

можуть приймати рішення та робити прогнози, але вони мають принципово різні підходи під час навчання.

Supervised Learning – це один з найпопулярніших методів навчання нейронної мережі. Його основна ідея в тому, що модель навчається на відібраних даних. Тобто, їй надають вхідні дані та вихідні дані, які очікують від неї отримати. У CNN [4] (Convolutional neural network) навчання нейромережі відбувається за допомогою згорткових шарів, які знаходять унікальні ознаки на об’єкті. Після чого через алгоритм зворотного поширення помилки здійснюється налаштування вагових коефіцієнтів. Це процес відбується до тих пір, поки модель не перестає навчатися.

Reinforcement Learning – це також один з найпопулярніших методів навчання нейронної мережі. Особливість його в тому, що він взаємодіє з середовищем з яким і буде працювати. Тобто, він отримує позитивні бали за правильні дії та негативні бали за неправильні дії. Задача перед RL стоїть доволі зрозуміла, а саме, у максимізація винагороди у довгостроковій сумі шляхом пошуку оптимальної поведінки. Тому, така модель навчання добре підходить до задач де потрібна послідовність дій: управління роботом чи при створенні агента для будь-якої гри. Основними алгоритмами є Q-learning [5], SARSA [6] та методи на основі глибокого навчання, такі як Deep Q-Networks [7] (DQN). Унікальною особливістю такого навчання є те, що модель не отримує негайно правильних відповідей, а повинна навчатися шляхом проб і помилок, балансуючи між дослідженням і використанням набутих знань.

1.2 Аналіз ефективності

Ефективність навчання за допомогою Supervised Learning визначається в здатності моделі до точного узагальнення та передбачення. Через те, що вхідні та вихідні дані є розміченими, алгоритми можуть побудувати чітку математичну модель, яка зможе встановлювати закономірності між вхідними та цільовими даними. Це особливо корисно в таких сферах, як медична діагностика, фінансове прогнозування, розпізнавання зображень та автоматизація процесів прийняття

рішень. Supervised Learning є дуже потужним інструментом машинного навчання, який має високий потенціал точності та передбачуваності. Ефективність такого методу ґрунтується на здатності навчатися на розмічених даних, встановлювати складні залежності та робити точні передбачення. Однак, для того, щоб модель успішно навчалась потрібно ретельно підготувати вхідні/вихідні дані, вибрати відповідний алгоритм, налаштування моделі та спостерігати, як вона навчається.

Основна ефективність Reinforcement Learning полягає у здатності моделі до адаптивного навчання та прийнятті оптимальних рішень у різних середовищах. Модель послідовно вивчає простір можливих дій, отримуючи бали за успішні та штрафи за неуспішні рішення. Цей ітеративний процес дозволяє моделі поступово виробляти стратегію, яка максимізує сумарну винагороду. Яскравими прикладами успіху є системи штучного інтелекту, що перевершують людей у складних іграх, таких як Go, шахи та відеоігри, а також у складних задачах управління роботизованими системами та оптимізації промислових процесів. Незважаючи на технічні виклики, цей підхід демонструє вражаючий потенціал у розв'язанні складних задач, які раніше здавалися нерозв'язними для штучного інтелекту.

1.3 Обмеження та перешкоди

Однією з головних проблем нейронних мереж є ефект «чорної скриньки» коли внутрішні процеси прийняття рішень нейронною мережею є практично непрозорими та незрозумілими. Це означає, що навіть досвідчені фахівці не завжди можуть чітко пояснити, чому штучний інтелект прийняв саме таке рішення, що створює серйозні проблеми у критично важливих сферах. А також, обмежена здатність нейронних мереж до адаптації в умовах швидких змін вхідних даних. На відміну від людського мозку, штучні нейронні мережі потребують повторного навчання або значної модифікації для ефективної роботи при суттєвих змінах контексту.

Тому, незважаючи на чудові можливості Supervised Learning має свої певні та обмеження та виклики. Одна з найголовніших проблем це необхідність великого обсягу якісних даних. Все це займає великої кількості людських ресурсів для підготовки навчальної вибірки. Також, точність моделі безпосередньо залежить від якості та репрезентативності навчальних даних. Неправильно підготовлені або мала кількість даних можуть призвести до суттєвих похибок у передбаченнях. Ще однією важливою проблемою є ризик перенавчання, коли модель занадто точно підлаштовується під навчальні дані, втрачаючи здатність до узагальнення на нові, невідомі раніше приклади.

Reinforcement Learning також має свої вади, які впливають на її ефективність. Одним з найскладніших аспектів є баланс між дослідженням та експлуатацією з робочим середовищем. Тобто, модель повинна постійно балансувати між пошуком нових стратегій та використанням вже відомих ефективних рішень. Занадто агресивне дослідження може призвести до марнування ресурсів, а надмірна експлуатація – до застряганням моделі в локальних оптимумах. А також, процес навчання часто вимагає величезної кількості ітерацій і може бути обчислювально складним, особливо в умовах великої кількості можливих станів та дій.

1.4 Висновок до розділу 1

Нейронні мережі є дуже потужним інструментом в сучасному світі. Вони вже допомагають людині вирішувати надскладні задачі, які раніше не мали рішення в світі програмного забезпечення. Головним фактором успіху в цій сфері є правильне вибір, проектування та навчання нейронної мережі. Тому, вибір правильної архітектури та способу навчання є ключом для вирішення проблеми такої проблеми як: побудова ігрового агента, але і будь-якої іншої задачі де можна по справжньому використати потенціал цієї технології. Кожен день нейронні мережі та методи їх навчання вдосконалюються через вклад провідних компаній, що підтверджує важливу роль нейронних мереж у сучасному світі.

2. Створення моделей та ігрових агентів

2.1 Загальні положення

Серед різних типів штучних нейронних мереж особливе місце займають згорткові (CNN), які спеціалізуються на обробці візуальної інформації. Завдяки своїй архітектурі, ці мережі чудово справляються з такими завданнями як ідентифікація предметів, розпізнавання людських облич та аналіз мовлення через зображення. У сфері настільних ігор CNN демонструють значний потенціал, допомагаючи комп'ютеру розуміти ігрове поле, оцінювати позиції та діяти як розумний опонент для гравця. Ключовою особливістю CNN є використання згорткових шарів, які автоматично виявляють важливі ознаки в зображеннях – від простих ліній та кутів до складних патернів та текстур. Така здатність до ієрархічного навчання робить їх незамінними в сучасних системах комп'ютерного зору.

Представлення дошки є головним фактор для досягненні успіху в тренуванні нейронних мереж. У програмному форматі можна представити дошку як варіанти чисел де (0) – пуста клітинка, (1) – символ хрестик та (-1) – символ нулика.

Документування коду програми виконано за допомогою веб-додатку Jupyter Notebook [8] та мови Python [9].

```
[7]: from tictactoe.board import Board
     board = Board()
     print(repr(board))

     Board(board=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], endgame=False)
```

Рисунок 2.1 – Пуста дошка у програмному форматі

```
[6]: from tictactoe.board import Board
     board = Board()
     print(repr(board))
     board.set_move(4, Board.CROSS)
     board.random_move(Board.NOUGHT)
     print(repr(board))

     Board(board=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], endgame=False)
     Board(board=[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], endgame=False)
```

Рисунок 2.2 – Дошка з хрестиком і нуликом у програмному форматі

Для користувача розроблене графічне представлення дошки за допомогою фреймворка Flask [10] та графічних елементів JavaScript.

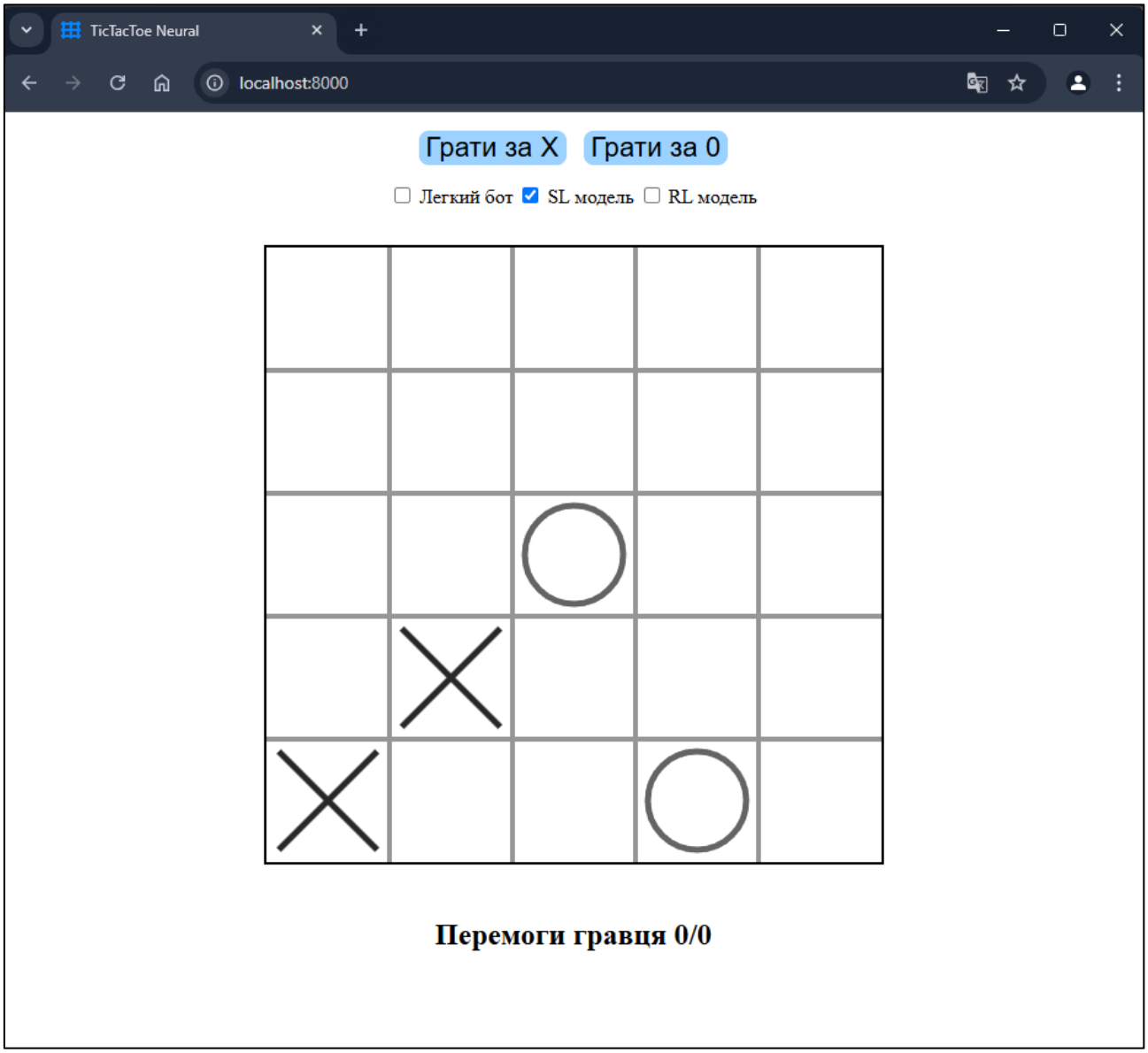


Рисунок 2.3 – Графічне представлення дошки

Натомість, таке представлення зручне для користувача, але для нейронної мережі потрібно представити це зображення у більш коректному форматі. Тому, розробимо приклад на якому буде три фігури хрестика та дві нулика:

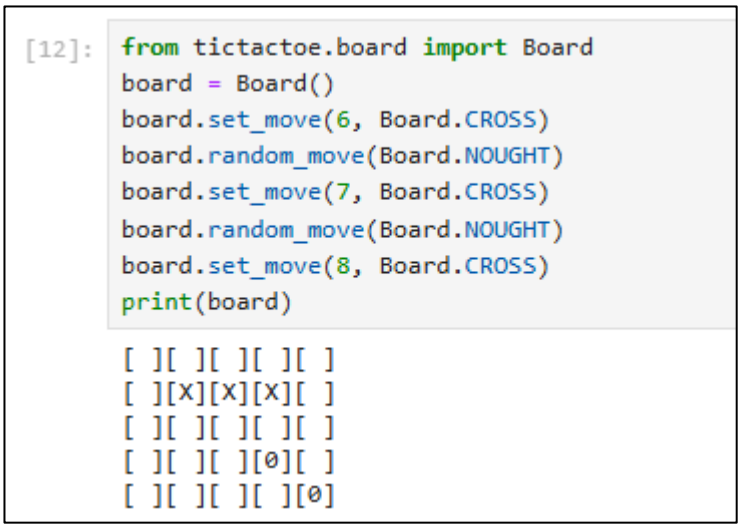


Рисунок 2.4 – Приклад для перетворення дошки

Даний стан дошки може бути перетворений у тензор з розміром (2, 5, 5). Де 2 – кількість шарів, 5 – висота зображення у пікселях та 5 – ширина зображення у пікселях.

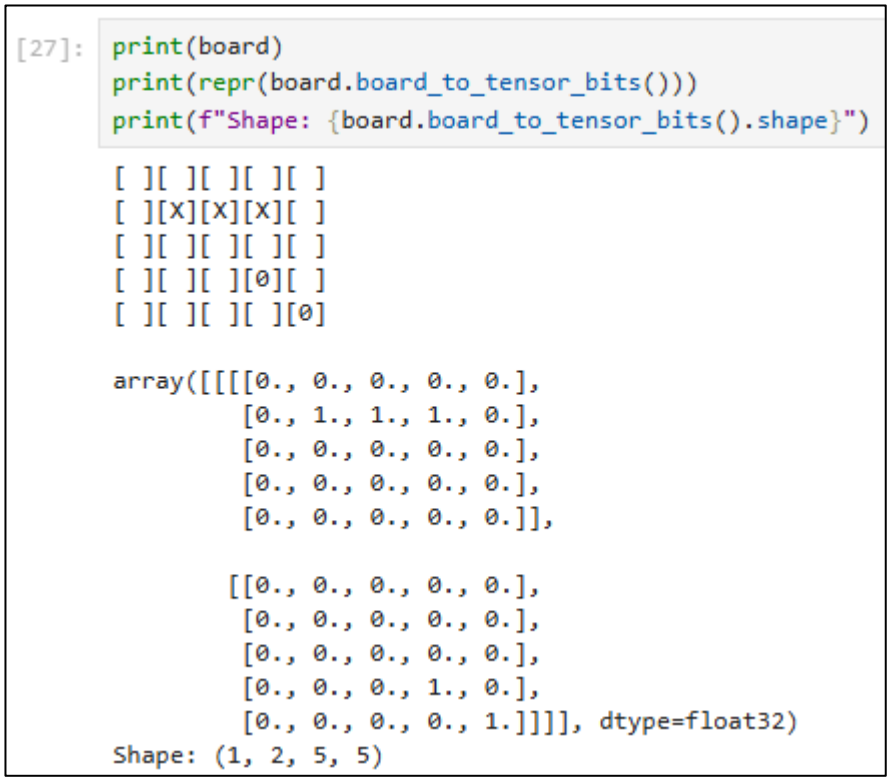


Рисунок 2.5 – Приклад представлення дошки у бітовому форматі

У цьому випадку дошка, яка зберігалась у програмі у форматі (25,), тобто масив з 25 елементів кожен з яких може бути (0, 1 або -1), перетворюється на дві матриці у загальний розмір (2, 5, 5). Перша матриця це представлення хрестика

де позиція хрестика позначається як (1), а всі інші фігури як (0). А також, друга матриця де елементи нулика позначаються як (1), а всі інші фігури так само, як в хрестика – (0).

2.2 Особливості моделі Supervised

Оскільки, модель Supervised навчається на мічених даних, тобто, вона отримує вхідний шар, після чого проводить розрахунки, перевіряє з очікуваним результатом, наданими вчителем, та робить коригування в моделі. Модель такої нейронної мережі буде виглядати наступним чином:

- Вхідний шар:
    - Розмір батча;
    - 2 – Кількість вхідних шарів;
    - 5 – Висота (піксель);
    - 5 – Ширина (піксель).
- Згорткові шари:
    - Кількість шарів та фільтрів під час тестування буде різна;
    - Розмір ядра 3;
    - Функція активації ReLU [11].
- Flatten (Перетворення карт ознак у одновимірний масив);
- Перший повністю зв’язаний шар:
    - Вхідний (Розмір батча, кількість шарів * 5 * 5);
    - Вихідний (Розмір батча, кількість шарів * 5 * 5).
- Другий повністю зв’язаний шар:
    - Вхідний (Розмір батча, кількість шарів * 5 * 5);
    - Вихідний (Розмір батча, 1).
- Функція активації tanh [12] (Гіперболічний тангенс).
- Вихідний шар:
    - Розмір батча;
    - 1 – Кількість нейронів.

2.3 Особливості Supervised агента

Для того, щоб побудувати навчання Supervised агента, потрібно створити або завантажити відповідний датасет. Де у якості вхідного шару буде стан дошки, а у вихідного – результат гри.

Оскільки «Хрестики-нулики» з розміром дошки 5 на 5 елементів є не стандартною грою, тому, для цього було створене окреме програмне забезпечення, яке запускає грати двох простих ботів, після чого записує результат гри.

Створені ігри записуються у список, після чого зберігаються у форматі .csv для подальшого використання.

```
[1]: %run -m tictactoe.supervised.dataset

tictactoe/supervised/data/train.csv games: [150] x_wins: [150] o_wins: [150] draws: [150]
tictactoe/supervised/data/validate.csv games: [75] x_wins: [75] o_wins: [75] draws: [75]
```

Рисунок 2.6 – Створення датасету. Параметри за замовчуванням (150 тренувальних та 75 валідаційних ігор)

```
[1]: %run -m tictactoe.supervised.dataset 1000 500

tictactoe/supervised/data/train.csv games: [1000] x_wins: [1000] o_wins: [1000] draws: [1000]
tictactoe/supervised/data/validate.csv games: [500] x_wins: [500] o_wins: [500] draws: [500]
```

Рисунок 2.7 – Створення датасету. Використання унікальних параметрів (1000 тренувальний та 500 валідаційних ігор)

Створені ігри можна переглянути у файлі tictactoe/supervised/data/train.csv та tictactoe/supervised/data/validate.csv.

```
PS D:\Project\MyProjects\TicBot\tictactoe\supervised\data> type train.csv | more
0,-1,1,0,-1,-1,0,1,1,1,1,0,1,0,-1,-1,1,1,0,0,-1,-1,1,-1,0,1
0,1,-1,1,-1,-1,-1,-1,1,1,-1,0,1,-1,0,1,1,1,-1,0,1,1,-1,1,-1,1
-1,-1,-1,1,-1,1,1,0,1,1,-1,1,-1,-1,1,-1,1,1,-1,0,-1,1,1,-1,1,1
0,1,1,1,-1,0,1,-1,1,-1,0,1,-1,1,-1,0,-1,-1,1,1,-1,1,1,-1,-1,1
0,1,1,0,-1,-1,1,1,-1,0,-1,-1,-1,1,1,0,0,1,0,1,0,0,-1,-1,1,1
```

Рисунок 2.8 – Тренувальний датасет

```
PS D:\Project\MyProjects\TicBot\tictactoe\supervised\data> type validate.csv | more
-1,-1,1,1,-1,-1,1,1,0,0,1,1,-1,0,-1,1,-1,0,1,-1,-1,-1,1,1,1,1
1,1,0,1,-1,-1,1,-1,1,-1,-1,0,1,1,-1,1,-1,-1,1,1,-1,-1,1,-1,1,1
0,0,-1,0,1,-1,-1,1,0,-1,0,1,0,-1,-1,-1,1,1,1,1,0,1,-1,1,0,1
-1,0,-1,1,1,-1,0,-1,1,1,0,0,0,1,0,1,0,-1,1,1,-1,-1,1,0,-1,1
1,0,-1,0,1,0,1,-1,0,0,0,0,1,0,-1,0,-1,-1,1,-1,-1,1,1,0,1,1
```

Рисунок 2.9 – Валідаційний датасет

Як можна було почати зі скріншотів кількість чисел у рядку дорівнює 26, хоча, розмір дошки – 25. Це через те, що останнє 26-е число відповідає за переможця у грі. Тобто, якщо в грі переміг хрестик, то число дорівнює (1), якщо нулик – (-1) та нічия – (0).

Розглянемо приклад з файлу train.csv, де стан дошки (0, -1, -1, 0, 0, 1, 1, -1, -1, 0, -1, 1, 1, 0, -1, 0, 0, 1, 0, 1, -1, -1, 1, 1) та переможець 1 (хрестик).

```
[1]: from tictactoe.board import Board

     board = Board([0,-1,-1,0,0,1,1,-1,-1,0,-1,1,1,0,-1,0,0,1,0,1,-1,-1,1,1,1])

     print(board)
     print(board.get_winner())
     print(f"Figure winner: {board.get_figure_winner()}")

     [ ][0][0][ ][ ]
     [X][X][0][0][ ]
     [0][X][X][ ][0]
     [ ][ ][X][ ][X]
     [0][0][X][X][X]

     (True, ((1, 0), (2, 1), (3, 2), (4, 3)))
     Figure winner: 1
```

Рисунок 2.10 – Результат гри з переможцем хрестиком

Будь-який рядок можна перетворити на об’єкт класу Board та перевірити стан дошки та переможця. В даному випадку перемогу отримав хрестик.

Наступним етапом підготовки даних є перетворення, а саме, нормалізація датасету .csv файлу з числами (-1, 0 та 1) у формат, який краще розуміє нейронна мережа: дві матриці з числами від 0 до 1. Для реалізації цієї процедури було створено модуль tictactoe/supervised/batch. Цей фунціонал окремо зчитує ігри з двох файлів: train.csv та validate.csv та перетворює їх 4-х вимірні масиви з

розміром (Кількість ігор, кількісті шарів, висота, ширина) та зберігає у форматі .npz.

```
[1]: %run -m tictactoe.supervised.batch

     File "tictactoe/supervised/data/train.npz" batch_in: (3000, 2, 5, 5) batch_out: (3000, 1)
     File "tictactoe/supervised/data/validate.npz" batch_in: (1500, 2, 5, 5) batch_out: (1500, 1)
```

Рисунок 2.11 – Перетворення ігор зі списку у 4-х вимірний масив

Після створення датасету та його нормалізації, можна переходити до навчання моделі. Час який модель буде потребувати на навчання буде залежатиме від кількості даних та складності моделі.

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.supervised.sl_model
0 0.6726245880126953
10 0.08345572650432587
20 0.07205353677272797
30 0.03283166512846947
40 0.064274944436502457
50 0.006901496555656195
```

Рисунок 2.12 – Процес навчання Supervised моделі

Перше число у рядку позначає епоху навчання, тобто, скільки ітерацій зробила модель під час навчання. Друге – похибку за епоху навчання чим менше це число, тим краще, адже модель може краще прогнозувати стан дошки.

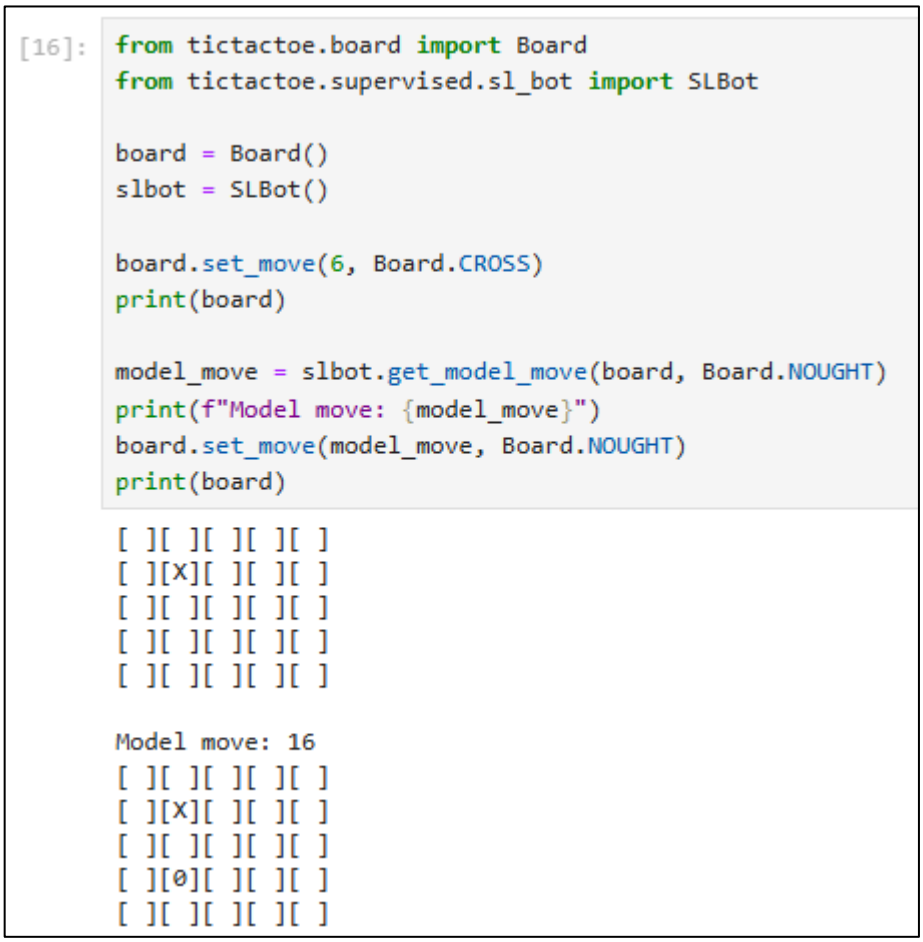
Оскільки, створена модель має CNN архітектуру й може розпізнавати образи, то її можна протестувати на валідаційному датасеті та перевірити успішність моделі.

```
[1]: %run -m tictactoe.supervised.validate

     Data length: 500 validation loss: 0.2518, accuracy: 83.20%
```

Рисунок 2.13 – Валідація моделі

Останнім етапом Supervised агента є передача в якості параметра стану дошки та фігури за яку агент грає. Модель робить аналіз всіх можливих ходів та робить той, який має найбільшу вірогідність для перемоги.



```
[16]: from tictactoe.board import Board
      from tictactoe.supervised.sl_bot import SLBot

      board = Board()
      slbot = SLBot()

      board.set_move(6, Board.CROSS)
      print(board)

      model_move = slbot.get_model_move(board, Board.NOUGHT)
      print(f"Model move: {model_move}")
      board.set_move(model_move, Board.NOUGHT)
      print(board)

[ ][ ][ ][ ][ ]
[ ][X][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]

Model move: 16
[ ][ ][ ][ ][ ]
[ ][X][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][0][ ][ ][ ]
[ ][ ][ ][ ][ ]
```

Рисунок 2.14 – Використання моделі Supervised за сторону нулик

```
[3]: from tictactoe.board import Board
     from tictactoe.supervised.sl_bot import SLBot

     board = Board()
     slbot = SLBot()

     model_move = slbot.get_model_move(board, Board.CROSS)
     print(f"SLBot move: {model_move}")
     board.set_move(model_move, Board.CROSS)
     print(board)

     board.set_move(14, Board.NOUGHT)
     print(board)

     SLBot move: 10
     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]
     [X][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]

     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]
     [X][ ][ ][ ][0]
     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]
```

Рисунок 2.15 – Використання моделі Supervised за сторону хрестик

2.4 Особливості моделі Reinforcement

Модель Reinforcement навчається на своїх діях, тому її архітектура, хоча і схожа, але відрізняється від Supervised кількістю вихідних нейронів та останньою функцією активації. Така мережа виглядає наступним чином:

- Вхідний шар:
  - Розмір батча;
  - 2 – Кількість вхідних шарів;
  - 5 – Висота (піксель);
  - 5 – Ширина (піксель).
- Згорткові шари:
  - Кількість шарів та фільтрів під час тестування буде різна;
  - Розмір ядра 3;
  - Функція активації ReLU.

- Flatten (Перетворення карт ознак у одновимірний масив);
- Перший повністю зв’язаний шар:
    - Вхідний (Розмір батча, кількість шарів * 5 * 5);
    - Вихідний (Розмір батча, кількість шарів * 5 * 5).
- Другий повністю зв’язаний шар:
    - Вхідний (Розмір батча, кількість шарів * 5 * 5);
    - Вихідний (Розмір батча, 25).
- Функція активації відсутня.
- Вихідний шар:
    - Розмір батча;
    - 25 – Кількість нейронів.

2.5 Особливості Reinforcement агента

У свою чергу Reinforcement агент має більш інтерактивний спосіб навчання. В цьому випадку модель такого агента має свої особливості, а саме, вихідний шар з 25 нейронів, замість 1 у Supervised моделі. Така архітектура дозволяє проводити покрокове навчання моделі під час гри.

Було створено два агента на основі однієї моделі. Один агент призначений для гри за сторону хрестик, другий – нулик. Саме тому, для успішної реалізації цих агентів було обрано навчання з підкріпленням з алгоритмом Deep Q-Network (DQN). Такий підхід дозволяє агенту безпосередньо взаємодіяти з ігровим середовищем, бачити стани та робити дії. Головними ключами до успіху реалізації цих агентів є наявність:

- Системи пам’яті. Агент зберігає свій попередній досвід у форматі (Стан дошки, вибрана позиція, нагорода, наступний стан дошки, стан гри). Отриманий досвід буде в подальшому використовувати для пошуку найоптимальних дій;
- Системи дій. Модель з підкріпленням особлива тим, що перед навчанням вона досліджує середовище в якому знаходиться. Тому, вона має змінну епсилон = 1, яка з кількістю ходів агента поступово зменшує своє значення.

Така змінна допомагає агенту перші свої ходи робити випадково та набиратися знань. З плином ходів, значення епсилон зменшується і модель переходить у режим використання отриманих знань.

- Системи винагород. Після кожної дії агент отримує систему нагород: позитивні бали за перемогу, від'ємні бали за поразку, нічию та продовження гри. Це все сприяє на його дії, тому агент намагається набрати найбільшу кількість балів за найменшу кількість дій;
- Функції Q. Центральна концепція в навчанні з підкріпленням, зокрема в алгоритмах, таких як Q-learning та DQN. Робити оцінку на «якість» певної дії та визначати чи принесло це успіх, чи ні. Формула алгоритму: «нагорода + коефіцієнт дисконтування * максимальне Q-значення * (1 – стан гри)».

Для реалізації навчання двох агентів застосовано спосіб гри один з одним. Тобто, перший агент починає гру за хрестик, інший – нулик. Такий підхід має ряд переваг:

- Мотивація до навчання;
- Ефективне використання ресурсів;
- Можливість самонавчання;
- Виявлення слабких місць.

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.reinforcement.rl_model
0 19 cross: [-5.0, 1.0000, 10] nought: [0.5, 1.0000, 10]
10 14 cross: [1.5, 0.8508, 115] nought: [-3.5, 0.8508, 111]
20 24 cross: [-8.0, 0.6951, 209] nought: [-6.0, 0.6951, 199]
30 15 cross: [-4.0, 0.5680, 296] nought: [1.5, 0.5680, 280]
40 13 cross: [-3.5, 0.4641, 394] nought: [2.0, 0.4641, 376]
```

Рисунок 2.16 – Процес навчання Reinforcement моделі

Процес навчання частково схожий на навчання моделі з учителем, але має своє особливості, для опису було взято другий рядок навчання:

- 10 – Позначення епохи навчання;
- 14 – Позначення кількості ходів у ігровій партії;
- Cross:

- 1.5 – Кількість балів отриманих першим агентом за партію, позитивне значення означає його перемогу;
- 0.8508 – Поточний ступінь епсилон. Значення зменшується з кожною ітерацією, це означає, що агент поступово переходить у режим використання своїх знань. Поточний проміжок виставлено на рівні 1 <= epsilon <= 0.05 (Може бути змінено);
- 115 – Кількість станів гри у пам’яті агента. Поточне значення виставлено до 1000 станів (Може бути збільшено).

- Nought:
  - -3.5 – Кількість балів отриманих другим агентом за партію, негативне значення означає його поразку;
  - 0.8508 – Поточний ступінь епсилон. Значення зменшується з кожною ітерацією, це означає, що агент поступово переходить у режим використання своїх знань. Поточний проміжок виставлено на рівні 1 <= epsilon <= 0.05 (Може бути змінено);
  - 111 – Кількість станів гри у пам’яті агента. Поточне значення виставлено до 1000 станів (Може бути збільшено).

Даний модуль може бути налаштований різною кількістю параметрів агентів під час тестування програмного забезпечення. Це надасть можливість з пошуку найоптимальнішої моделі та структури для виконання поставленого завдання.

```
[7]: from tictactoe.board import Board
     from tictactoe.reinforcement.rl_bot import RLBotNought

     board = Board()
     rlbot_nought = RLBotNought()

     board.set_move(0, Board.CROSS)
     print(board)

     model_move = rlbot_nought.get_move(board)
     print(f"RLBotNought move: {model_move}")
     board.set_move(model_move, Board.NOUGHT)
     print(board)

[X][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]

RLBotNought move: 19
[X][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][0]
[ ][ ][ ][ ][ ]
```

Рисунок 2.17 – Використання RL агента за сторону нулик

```
[5]: from tictactoe.board import Board
     from tictactoe.reinforcement.rl_bot import RLBotCross

     board = Board()
     rlbot_cross = RLBotCross()

     model_move = rlbot_cross.get_move(board)
     print(f"RLBotCross move: {model_move}")
     board.set_move(model_move, Board.CROSS)
     print(board)

     board.set_move(4, Board.NOUGHT)
     print(board)

     RLBotCross move: 2
     [ ][ ][X][ ][ ]
     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]

     [ ][ ][X][ ][0]
     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]
     [ ][ ][ ][ ][ ]
```

Рисунок 2.18 – Використання RL агента за сторону хрестик

2.6 Висновок до розділу 2

У розділі 2 можна більше детально ознайомитися з відмінностями Supervised та Reinforcement методів навчання. Обидва використовують згорткові нейронні мережі (CNN), є підмножинами машинного навчання, але мають суттєві відмінності в підході до навчання, хоча побудовані для отримання схожого результату. RL є набагато більш самостійним підходом, хоча є доволі складним у структурі побудові агента та може мати наступні проблеми: навчання може бути дуже тривалим та потребувати великої кількості спроб, важко підібрати правильну функцію винагороди, а також, може бути нестабільним та схильним до локальних оптимумів. У свою чергу Supervised Learning потребує дуже багато праці у підготовці даних і може так статися, що модель навчається лише на тих прикладах, які були в навчальному наборі та буде погано справлятися з новими, нетиповими для себе ситуаціями.

3. Тестування агентів та моделей

3.1 Загальні положення

Під час тестування головною метою є визначення залежності складності моделі від кількості даних та її успіху, а також, визначення найкращого методу навчання для поставленої задачі. Це допоможе зрозуміти, які плюси та мінуси мають більш складні моделі та різні методи навчання. Для того, щоб створити оптимальне середовище для двох моделей були введенні наступні обмеження на схожі їх особливості:

- Обидві моделі мають однаковий вхідний шар під час усіх тестів;
- Кількість згорткових шарів та фільтрів повинна буди однакова під час різних тестів;
- Кількість повністю зв’язаних шарів повинна буди однакова під час різних тестів;
- Значення темпу навчання зафіксовано на значенні 0.001;
- Кількість епох навчання зафіксовано на значенні 100 ітерацій;
- Розмір батча 32 зразків за один прохід;
- Кількість ігор моделі проти простого бота зафіксовано на позначці 250;
- Кількість ігор моделей одна проти одної зафіксовано на позначці 250;
- Ігровий агент вибирає 1 з 4 найкращих варіантів для ходу.

Оскільки, дві моделі відрізняються підходами до навчання, наступні обмеження є унікальними для обох агентів:

- Supervised Learning:
    - Кількість датасету обмежена на позначці 100 тренувальних та 100 валідаційних ігор.
- Reinforcement Learning:
    - Пам’ять моделі зберігає до 1000 станів;
    - Нагороди:
        - 5 балів за перемогу у партії;
        - -5 балів за поразку у партії;

- -2 бали за нічию у партії;
- -0.5 балів за продовження гри.

- Коефіцієнт дисконтування дорівнює 1;
- Баланс між дослідженням та використанням:
  - epsilon = 1.0;
  - epsilon_decay = 0.98;
  - epsilon_min = 0.05.

3.3 Тестування найменшої моделі

3.2.1. Результати SL моделі та агента

```
SLModel(
  (a1): Conv2d(2, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=25, out_features=1, bias=True)
)
Кількість параметрів: 45
```

Рисунок 3.1 – Архітектура найменшої SL моделі

```
[3]: %run -m tictactoe.supervised.validate

     Data length: 100 validation loss: 0.8927, accuracy: 8.00%
```

Рисунок 3.2 – Валідація найменшої SL моделі

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.supervised.simulate 250
SL vs Easy (Side: X, N=250) W=168 (67.20%), L=62 (24.80%), D=20 (8.00%),  Length=19.34
SL vs Easy (Side: 0, N=250) W=161 (64.40%), L=80 (32.00%), D=9 (3.60%),  Length=17.16
```

Рисунок 3.3 – Результати ігор найменшої SL моделі проти простого бота

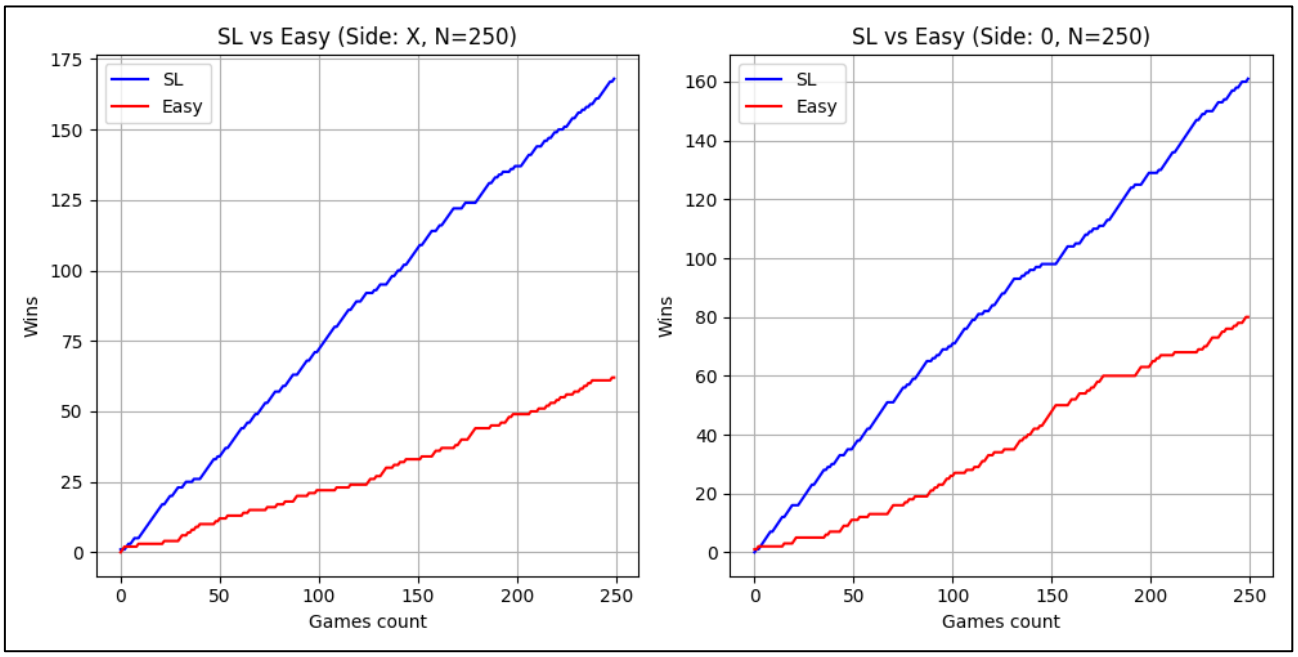


Рисунок 3.4 – Графічне представлення результатів ігор найменшої SL моделі

Найменша SL модель показала доволі непогані результат у грі проти простого бота, не дивлячись на її розмір. Із мінусів: 8% розпізнавання станів дошки є доволі низьким результат, а також, довжина гри є доволі довгою.

3.2.3 Результати RL моделі та агента

```
RLModel(
  (a1): Conv2d(2, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=25, out_features=25, bias=True)
)
Кількість параметрів: 669
```

Рисунок 3.5 – Архітектура найменшої RL моделі

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.reinforcement.simulate 250
RL vs Easy (Side: X, N=250) W=144 (57.60%), L=93 (37.20%), D=13 (5.20%),  Length=18.61
RL vs Easy (Side: 0, N=250) W=116 (46.40%), L=115 (46.00%), D=19 (7.60%),  Length=18.99
```

Рисунок 3.6 – Результати ігор найменшої RL моделі проти простого бота

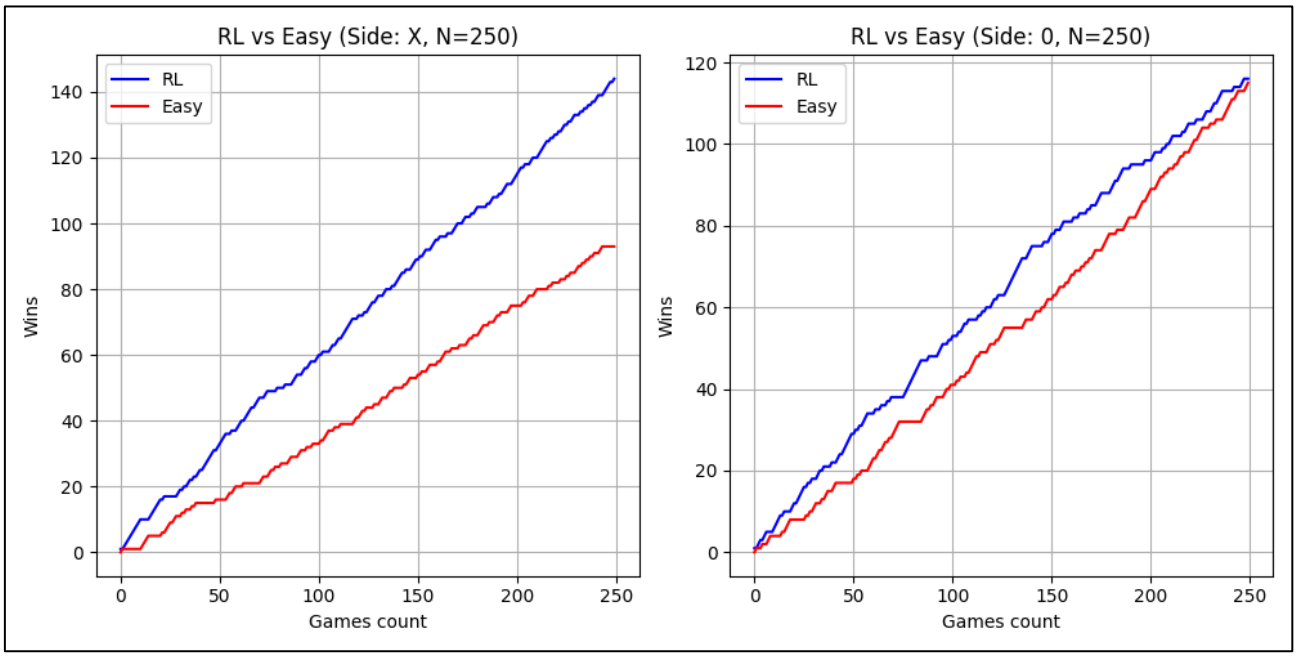


Рисунок 3.7 – Графічне представлення результатів ігор найменшої RL моделі

Із результатів можна зрозуміти, що найменша модель для RL агента є слабким місцем. Перемоги даються важко, особливо для агента, який грає за нулик. Низький результат перемог можна пояснити тим, що модель часто програвала агенту «хрестик» під час навчання та не встигла знайти свою стратегію.

3.2.3 Результати SL проти RL агента

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.simulate 250
SL vs RL (Side: X, N=250) W=162 (64.80%), L=66 (26.40%), D=22 (8.80%),  Length=18.25
SL vs RL (Side: 0, N=250) W=177 (70.80%), L=54 (21.60%), D=19 (7.60%),  Length=17.84
```

Рисунок 3.8 – Результати SL проти RL моделі

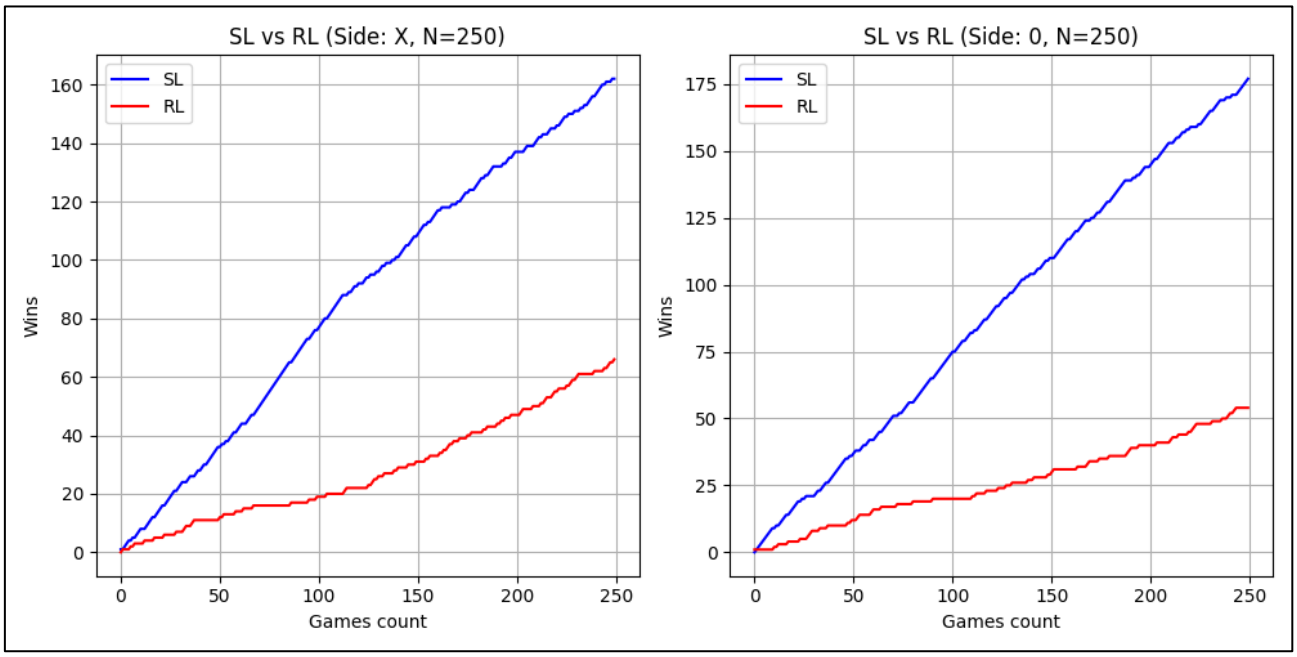


Рисунок 3.9 – Графічне представлення результатів ігор SL проти RL моделі

В ході тестування найменших моделей виявлено: що підхід Supervised Learning, незважаючи на доволі жахливий результат з розпізнавання стану дошки у 8%, має перевагу над Reinforcement Learning як в протистоянні простому боту, так і віч-на-віч. Такий результат можна пояснити тим, що SL модель навчається на вже готових даних, в той час як RL сама їх для себе підготовує і просто не може знайти чіткі взаємозв’язки у середовищі гри через простоту моделі.

3.3. Тестування малої моделі

3.3.1. Результати SL моделі та агента

```
SLModel(
  (a1): Conv2d(2, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=100, out_features=1, bias=True)
)
Кількість параметрів: 177
```

Рисунок 3.10 – Архітектура малої SL моделі

```
[2]: %run -m tictactoe.supervised.validate

     Data length: 100 validation loss: 0.6731, accuracy: 33.00%
```

Рисунок 3.11 – Валідація малої SL моделі

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.supervised.simulate 250
SL vs Easy (Side: X, N=250) W=196 (78.40%), L=47 (18.80%), D=7 (2.80%),  Length=15.72
SL vs Easy (Side: 0, N=250) W=139 (55.60%), L=100 (40.00%), D=11 (4.40%),  Length=18.19
```

Рисунок 3.12 – Результати ігор малої SL моделі

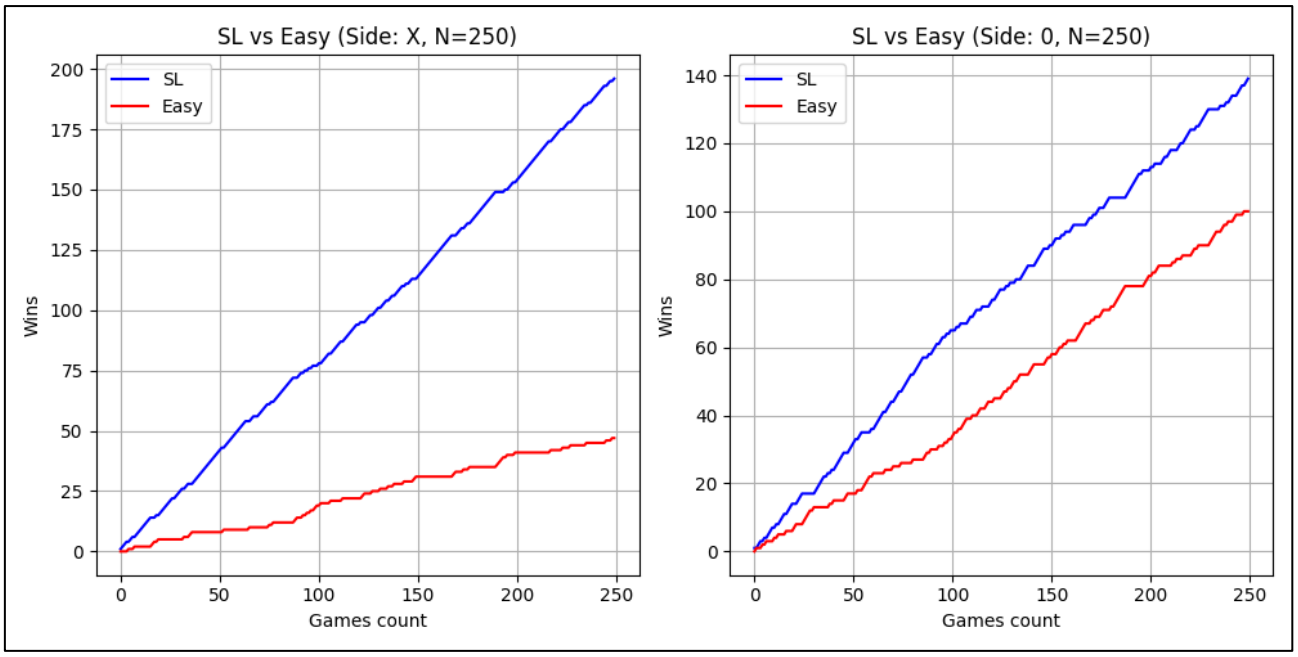


Рисунок 3.13 – Графічне представлення результатів ігор малої SL моделі

В ході цього тестування було отримано один з найкращих результатів для сторони «хрестик» як у відсотках перемог, так і у продовжені гри. Також, збільшення згорткових шарів дало суттєве збільшення розпізнавання стану дошки. Нажаль, результат для «нулика» трішки погіршився, посприяти цьому могла система з вибору 1 з 4 найкращих варіантів, агент міг обирати не найкращі варіанти, що і посприяло результату.

3.3.3. Результати RL моделі та агента

```
RLModel(
  (a1): Conv2d(2, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=100, out_features=25, bias=True)
)
Кількість параметрів: 2601
```

Рисунок 3.14 – Архітектура малої RL моделі

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.reinforcement.simulate 250
RL vs Easy (Side: X, N=250) W=191 (76.40%), L=45 (18.00%), D=14 (5.60%),  Length=16.43
RL vs Easy (Side: 0, N=250) W=126 (50.40%), L=108 (43.20%), D=16 (6.40%),  Length=18.96
```

Рисунок 3.15 – Результати ігор малої RL моделі

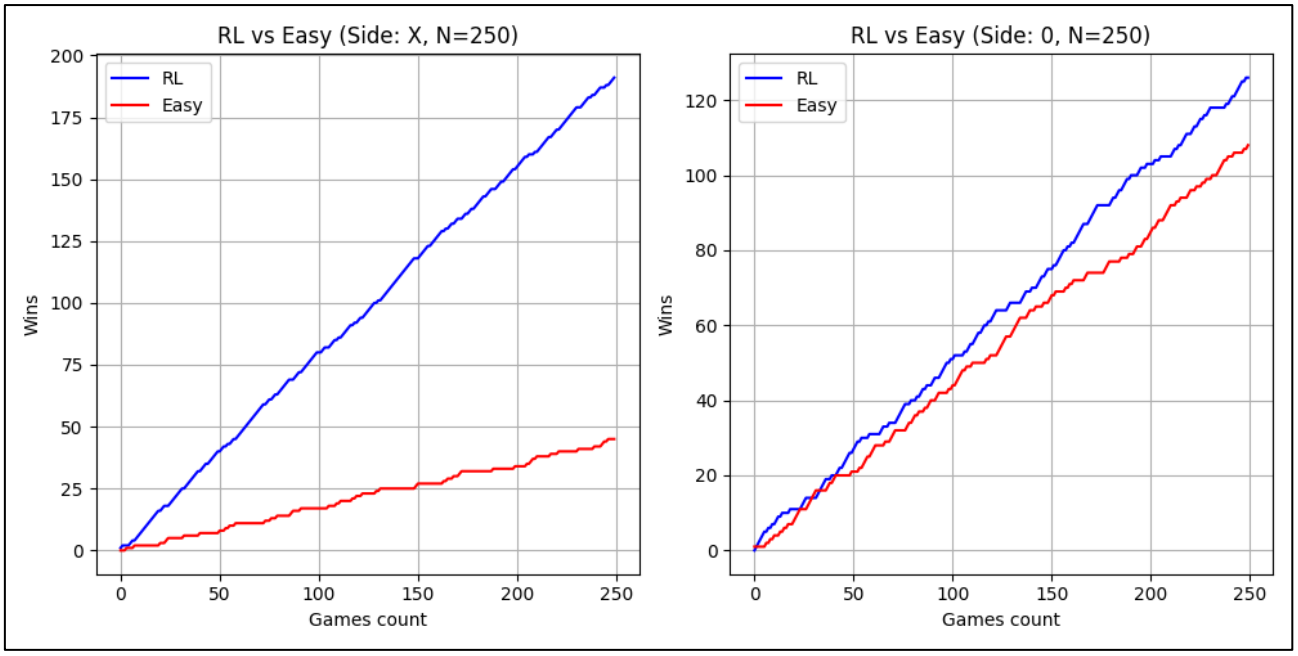


Рисунок 3.16 – Графічне представлення результатів ігор малої RL моделі

Як і у випадку SL моделі, RL модель показала один з найкращих результатів для сторони «хрестик». Результат «нулика» також покращився у порівнянні з найменшою моделлю.

3.3.3. Результати SL проти RL агента

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.simulate 250
SL vs RL (Side: X, N=250) W=188 (75.20%), L=59 (23.60%), D=3 (1.20%),  Length=15.03
SL vs RL (Side: 0, N=250) W=52 (20.80%), L=183 (73.20%), D=15 (6.00%),  Length=15.78
```

Рисунок 3.17 – Результати ігор SL проти RL моделі

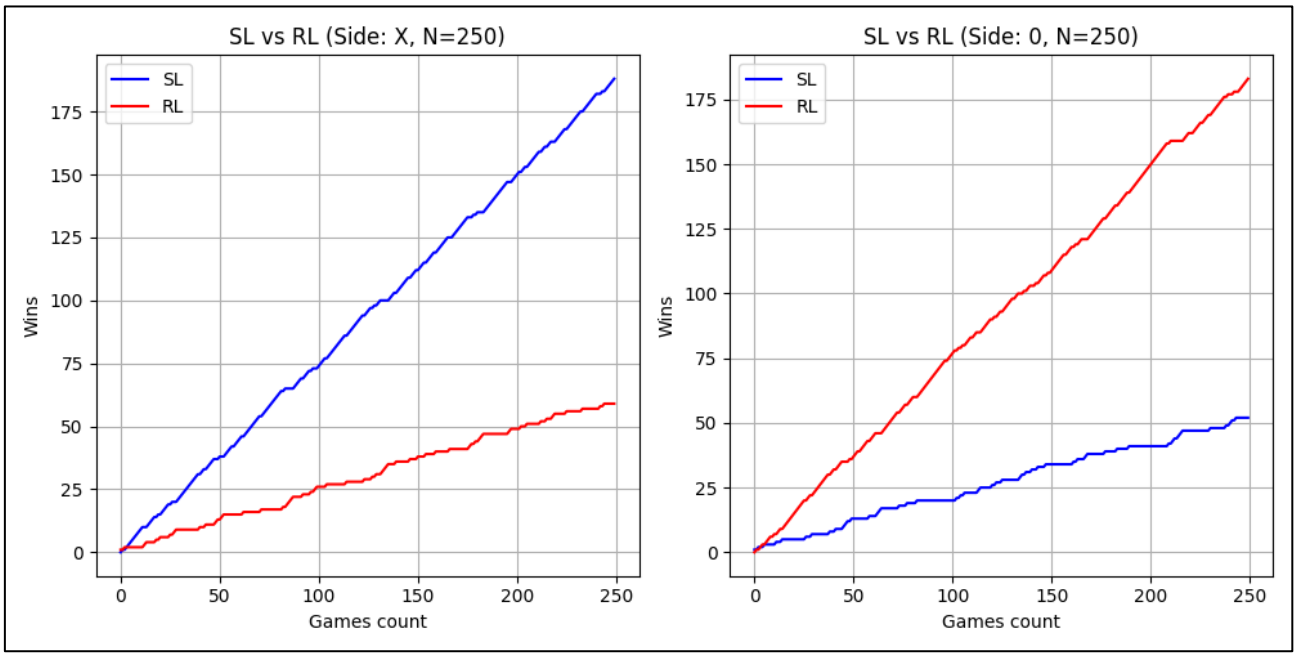


Рисунок 3.18 – Графічне представлення результатів ігор SL проти RL моделі

В ході тестування малих моделей можна дійти висновку, що загальні результати кожної моделі покращились, порівнюючи з попередньою моделлю. Supervised Learning показала 33% розпізнавання результатів гри, а модель RL змогла перемогти SL за сторону нулик. Також, моделі показали чудові результати довжини гри.

3.4 Тестування середньої моделі

3.4.1 Результати SL моделі та агента

```
SLModel(
  (a1): Conv2d(2, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=100, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=1, bias=True)
)
Кількість параметрів: 10277
```

Рисунок 3.19 – Архітектура середньої SL моделі

```
[2]: %run -m tictactoe.supervised.validate

     Data length: 100 validation loss: 1.0902, accuracy: 52.00%
```

Рисунок 3.20 – Валідація середньої SL моделі

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.supervised.simulate 250
SL vs Easy (Side: X, N=250) W=172 (68.80%), L=57 (22.80%), D=21 (8.40%),  Length=17.77
SL vs Easy (Side: 0, N=250) W=129 (51.60%), L=104 (41.60%), D=17 (6.80%),  Length=19.15
```

Рисунок 3.21 – Результати ігор середньої SL моделі

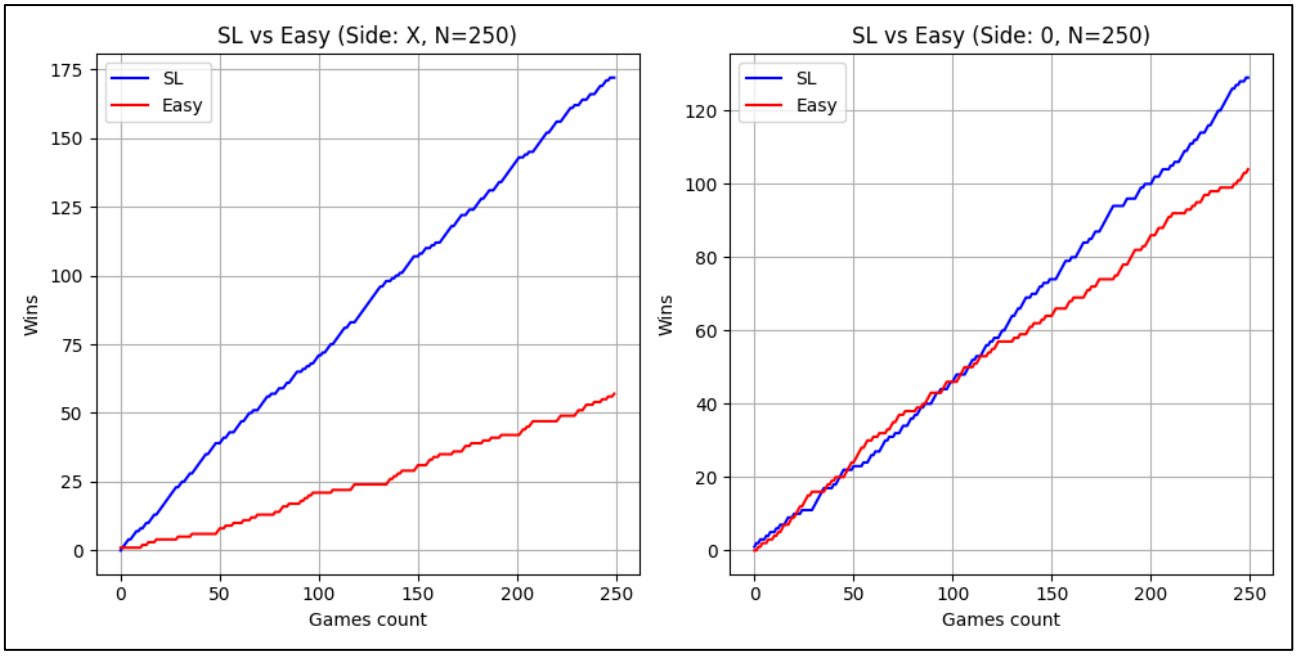


Рисунок 3.22 – Графічне представлення результатів ігор середньої SL моделі

В ході тестування стало зрозуміло, що додавання ще одного повнозв'язного шару у модель не дає суттєвих позитивних покращень, окрім, значного приросту у розпізнаванні стану дошки.

3.4.2 Результати RL моделі та агента

```
RLModel(
  (a1): Conv2d(2, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=100, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=25, bias=True)
)
Кількість параметрів: 12701
```

Рисунок 3.23 – Архітектура середньої RL моделі

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.reinforcement.simulate 250
RL vs Easy (Side: X, N=250) W=137 (54.80%), L=107 (42.80%), D=6 (2.40%),  Length=19.18
RL vs Easy (Side: 0, N=250) W=178 (71.20%), L=64 (25.60%), D=8 (3.20%),  Length=14.72
```

Рисунок 3.24 – Результати ігор середньої RL моделі

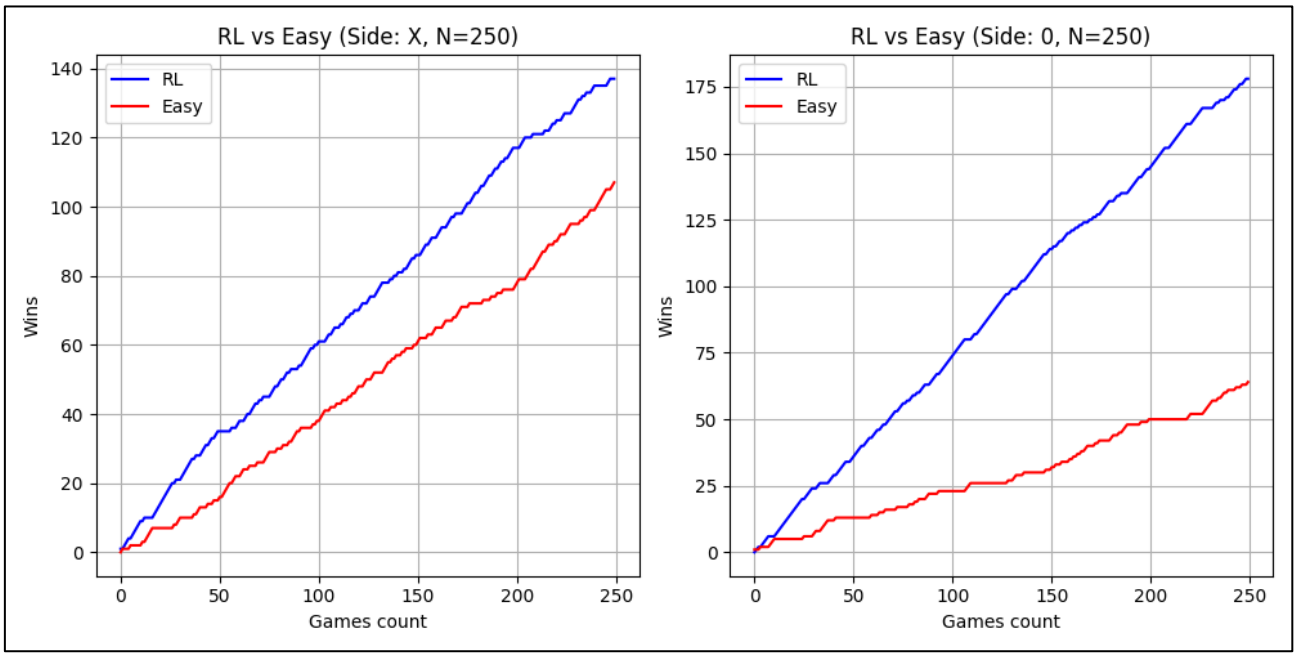


Рисунок 3.25 – Графічне представлення результатів ігор середньої RL моделі

Тестування середньої RL моделі показало найкращий результат за тривалістю гри, а також, одну з найбільших по кількісті перемог сторони «нулик». Більша кількість перемог цієї сторони означає: що під час навчання агент, який відповідав за навчання «нулика» отримував більшу кількість перемог та розвив кращу стратегію ніж агент «хрестик».

3.4.3. Результати SL проти RL агента

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.simulate 250
SL vs RL (Side: X, N=250) W=144 (57.60%), L=105 (42.00%), D=1 (0.40%),  Length=14.66
SL vs RL (Side: 0, N=250) W=112 (44.80%), L=125 (50.00%), D=13 (5.20%),  Length=19.13
```

Рисунок 3.26 – Результати ігор SL проти RL моделі

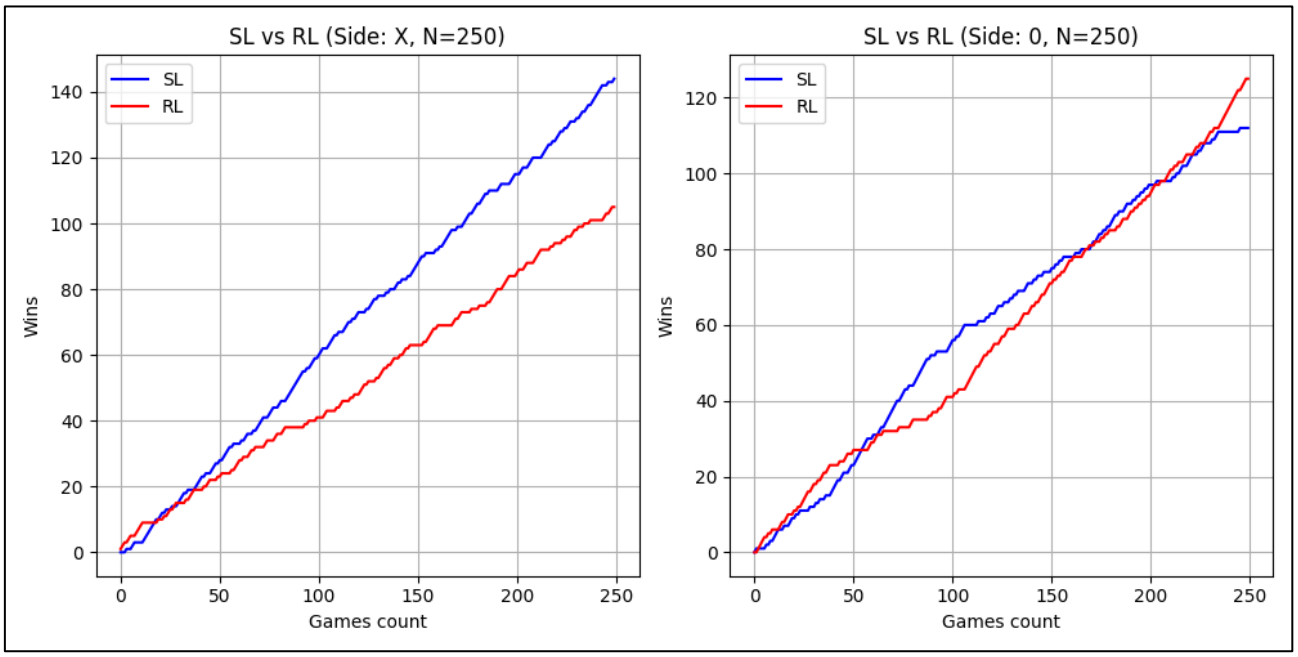


Рисунок 3.27 – Графічне представлення результатів ігор SL проти RL моделі

В ході тестування середньої моделі були виявлено, що збільшення складності моделей позитивно на результати, окрім, валідації SL та деяких випадків перестало впливати. Обидві моделі віч-на-віч показують приблизно однакові результати.

3.5 Тестування великої моделі

3.5.1 Результати SL моделі та агента

```
SLModel(
  (a1): Conv2d(2, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (a2): Conv2d(4, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (a3): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=400, out_features=400, bias=True)
  (fc2): Linear(in_features=400, out_features=1, bias=True)
)
Кількість параметрів: 162341
```

Рисунок 3.28 – Архітектура великої SL моделі

```
[2]: %run -m tictactoe.supervised.validate

     Data length: 100 validation loss: 0.8143, accuracy: 65.00%
```

Рисунок 3.29 – Валідація великої SL моделі

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.supervised.simulate 250
SL vs Easy (Side: X, N=250) W=188 (75.20%), L=52 (20.80%), D=10 (4.00%),  Length=16.26
SL vs Easy (Side: 0, N=250) W=178 (71.20%), L=65 (26.00%), D=7 (2.80%),  Length=16.50
```

Рисунок 3.30 – Результати ігор великої SL моделі

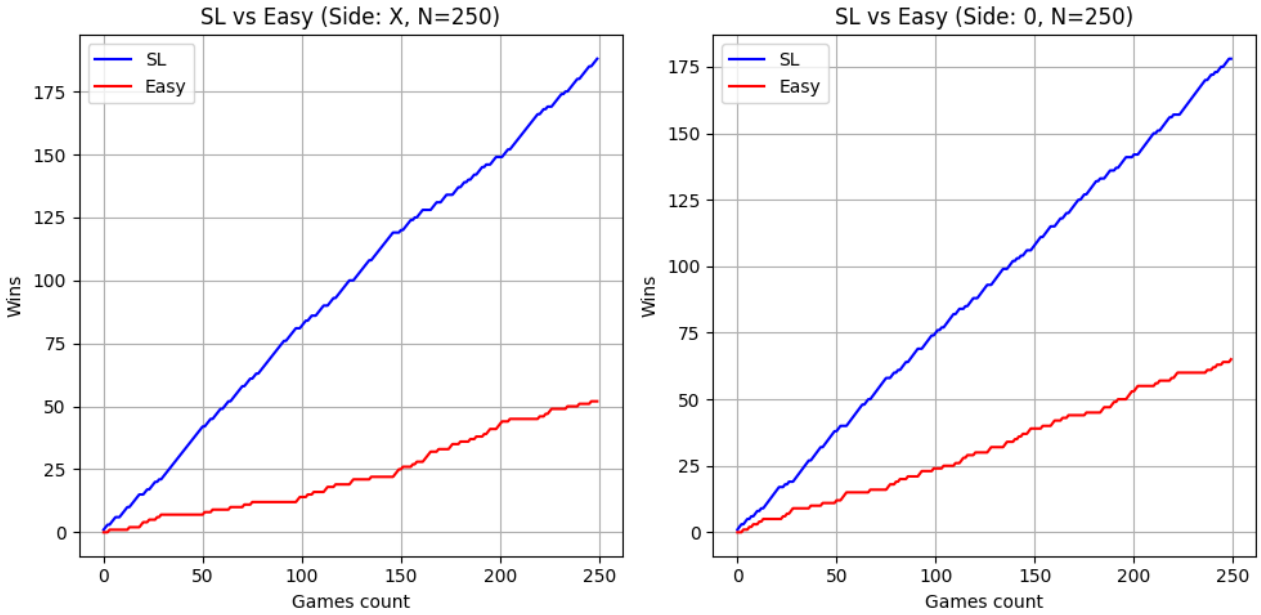


Рисунок 3.31 – Графічне представлення результатів ігор великої SL моделі

Велика модель для SL агента показала одні з найкращих результатів серед перемог, тривалістю гри та найкращий з точності розпізнавання стану дошки. Складна модель явно дала агенту можливість краще розпізнавати переможні комбінації серед навчального датасету, що і посприяло на результат.

3.5.2 Результати RL моделі та агента

```
RLModel(
  (a1): Conv2d(2, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (a2): Conv2d(4, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (a3): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=400, out_features=400, bias=True)
  (fc2): Linear(in_features=400, out_features=25, bias=True)
)
Кількість параметрів: 171965
```

Рисунок 3.32 – Архітектура великої RL моделі

```
Model: RLModel(Nought), Parameters: 171965
RL vs Easy (Side: X, N=250) W=127 (50.80%), L=112 (44.80%), D=11 (4.40%),  Length=19.38
RL vs Easy (Side: 0, N=250) W=148 (59.20%), L=87 (34.80%), D=15 (6.00%),  Length=18.08
```

Рисунок 3.33 – Результати ігор великої RL моделі

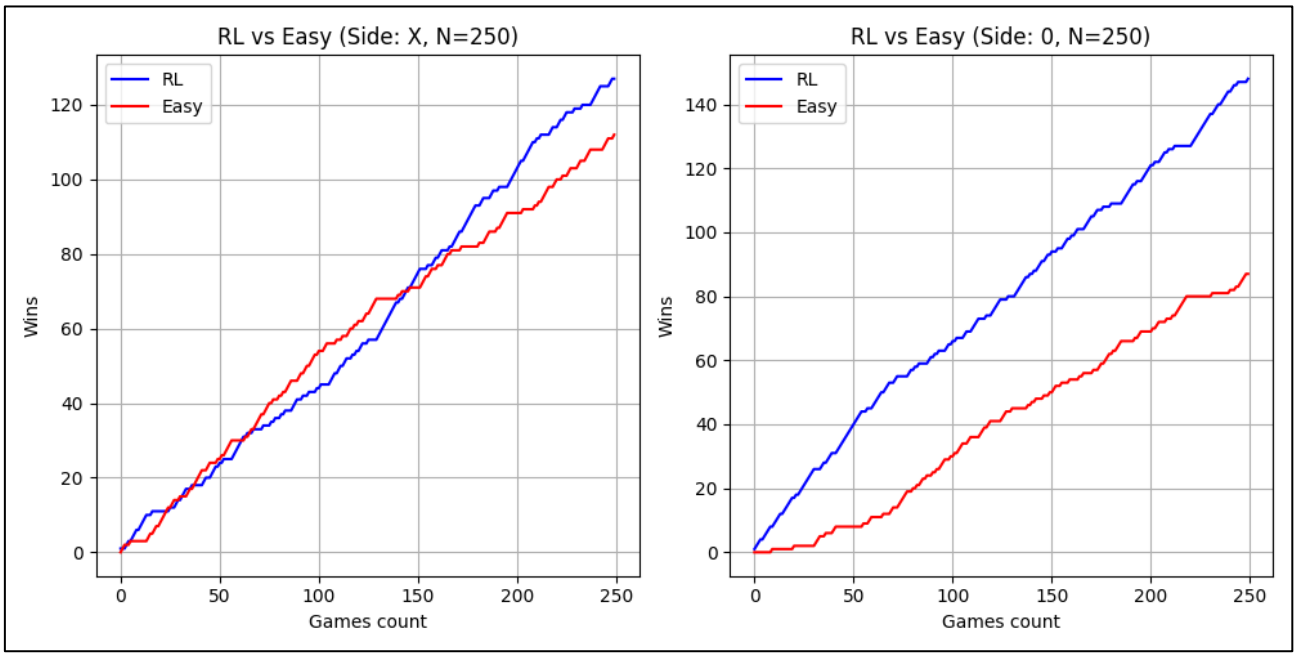


Рисунок 3.34 – Графічне представлення результатів ігор великої RL моделі

Вже чітко зрозуміло, що посилення моделі для RL агента потрібне у тому випадку, якщо архітектура агента буда ускладнюватися, тобто: збільшення кількісті ігор, видозміни системи нагород та функції навчання. В іншому випадку це буде тільки негативно впливати на результат.

3.5.3. Результати SL проти RL агента

```
PS D:\Project\MyProjects\TicBot> python -m tictactoe.simulate 250
SL vs RL (Side: X, N=250) W=107 (42.80%), L=80 (32.00%), D=63 (25.20%),  Length=19.90
SL vs RL (Side: 0, N=250) W=165 (66.00%), L=76 (30.40%), D=9 (3.60%),  Length=17.63
```

Рисунок 3.35 – Результати ігор SL проти RL моделі

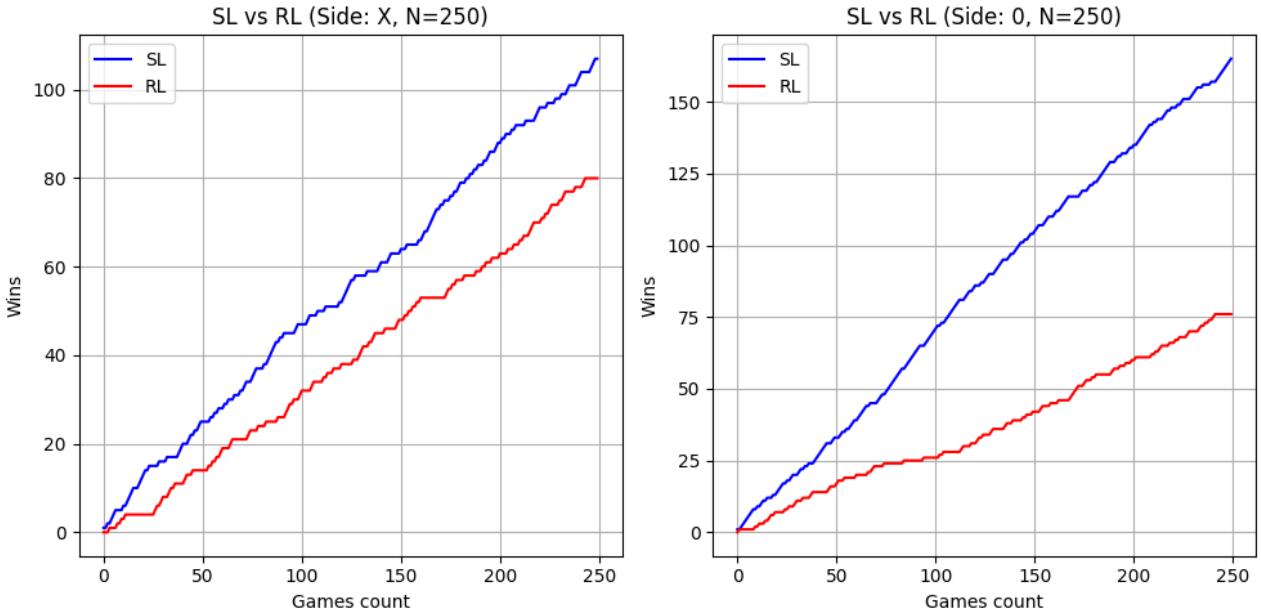


Рисунок 3.36 – Графічне представлення результатів ігор SL проти RL моделі

В ході тестування стало зрозуміло, що ще підвищення складності моделі дає мінімальний приріст у результаті для SL моделі і, навіть, негативний для RL моделі. Для такої складності моделі потрібно видозмінювати гіперпараметри для RL моделі.

3.6 Висновок до розділу 3

Під час тестування моделей у розділі 3 стало зрозуміло, що для поставленої задачі обидві моделі цілком підходять. Оскільки, успішність моделі залежить від дуже й дуже багатьох факторів, але, використовуючи обмеження, які були описані на початку тестування, можна зробити висновок, що суттєве збільшення складності моделі дає позитивні результати, але успішність моделі збільшується непропорціонально її складності. Щодо RL моделі, тестування показало цікавий результат, а саме, складна модель може навпаки, негативно впливати на результати агента. Це може бути спричинено тим, що такий агент через поставленні обмеження перед тестуванням не встигає навчитися та побудувати свою стратегію, або взагалі, складна модель нівелює отриманні знання. Налаштування моделей є дуже кропітким та складним процесом, який потребує глибокого розуміння принципу роботи нейронних мереж.

Висновок

У сучасному світі нейронні мережі стали невід'ємною частиною нашого життя. Вони присутні навіть там, де не очікуєш їх побачити. З плином часу нейронні мережі доказали свою важливість у існуванні серед людства. Маючи вже доволі довгу історію, вони зайняли нішу, яку раніше ніхто не міг зайняти: вирішення проблем, які мають у собі величезні обсяги даних. Нейронні моделі допомогли людині опрацьовувати великі обсяги даних, виявляти серед них закономірності та вирішувати завдання, які раніше здавалися дуже складними як для людини, так і для комп'ютера. Це дало змогу мережам охопити широкий спектр галузей, що додало їм віри в успішність цієї технології серед людства. Тому, відбулося саме те, що можна було очікувати: нейронні мережі отримали потужний поштовх у розвитку і це стосувалося як з точки програмної частини, так і фізичної, адже без розвитку паралельних обчислень компаніями, які займаються створенням графічних чипів ця технологія залишалась у далекому майбутньому.

Нейронна мережа є дуже обширним розумінням, яке може не конкретизувати особливості окремої моделі, і взагалі, розуміння однієї моделі не дає гарантії на розуміння іншої. Кожна задача потребує особливого підбору моделі, налаштування параметрів та гіперпараметрів. Для вирішення таких проблем, спільнота програмістів та великих компаній створила велику кількість програмного забезпечення для збору даних, їх нормалізації, а також, бібліотек, які допомагають створювати багаторівневі абстракції для того, що займатися проектуванням нейронної мережі, а не створенням її з нуля. Програмування моделей є відміним процесом від побудови звичайного програмного забезпечення. Адже, процес розробки є не лінійним, а ітеративним, а також, пояснюваність результату є низькою, що є відомою проблемою всіх нейронних мереж під назвою «чорна скринька», тобто, розробник, іноді, не може пояснити чому отриманий результат є таким. Створення моделі потребує якісного проектування, великих обчислювальних потужностей та вільного часу самого розробника.

Методи навчання нейронних мереж є дуже важливою сукупністю підходів та технік. Вони надають різні алгоритми за якими мережа починає своє навчання. Ці методи бувають різними, адже від того, які дані має компанія чи окремий вільний розробник залежить архітектура. Наприклад, підхід Supervised Learning (Навчання з вчителем) припускає, що в розробника є велика кількість даних на яких він може побудувати нейронну мережу, яка потім буде робити схожі припущення на валідаційних та робочих даних, яким навчилась під час тренування. Зовсім іншим підходом є Reinforcement Learning (Навчання з підкріпленням), що припускає відсутність даних, але можливість побудувати інтерактивну взаємодію моделі з потрібним середовищем.

В ході магістерської дипломної роботи було використано два методи навчання нейронних мереж: навчання з учителем та підкріпленням. Під час тестування було виявлено те саме, що очікувалось при проектуванні: обидві моделі підходять для створення ігрового агенту. Кожна з них має свої плюси та мінуси. Було виявлено, що складність моделі не завжди позитивно впливає на її успішність, а також, правильне налаштування параметрів та гіперпараметрів є ключом до успіху моделі. Під час тестування можна зробити хибний висновок, ніби SL агент набагато краще впорався із завданням, і це буде тільки на половину вірно. Таке враження складається через те, що в моделі «навчанням з учителем» більш проста архітектура і готові дані дають половину успіху, але така модель майже вичерпує свій потенціал при зростанні складності та зробити надуспішною не вдасться. У свою чергу RL модель не встигає розкрити увесь свій потенціал, хоча показує доволі достойні результати. Архітектура моделі «навчання з підкріпленням» якраз і розрахована на довге, але детальне вивчення структури середовища, що дасть їй перевагу над SL у майбутній перпективі. З цього можна зробити висновок, що якщо середовище просте, має невелику кількість станів і наявність даних – Supervised Learning є гарним вибором. У свою чергу, відсутність даних, складне середовище та велику кількість станів є ознакою для використання Reinforcement Learning методу навчання.

літературу оформизи згідно вимог

Бібліографічний список

1. Нейронні мережі [Електронний ресурс] // Neural networks. – Режим доступу: URL: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414 – Назва з екрана. – Дата звернення: 20 жовтня 2024.
2. Supervised Learning [Електронний ресурс] // Supervised Machine Learning. – Режим доступу: URL: https://www.javatpoint.com/supervised-machine-learning – Назва з екрана. – Дата звернення: 20 жовтня 2024.
3. Reinforcement learning [Електронний ресурс] // Reinforcement learning. – Режим доступу: URL: https://www.ibm.com/topics/reinforcement-learning – Назва з екрана. – Дата звернення: 21 жовтня 2024.
4. CNN [Електронний ресурс] // Convolutional Neural Network. – Режим доступу: URL: https://datascientest.com/en/convolutional-neural-network-everything-you-need-to-know – Назва з екрана. – Дата звернення: 22 жовтня 2024.
5. Q-Learning [Електронний ресурс] // Q-Learning. – Режим доступу: URL: https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-q-learning – Назва з екрана. – Дата звернення: 23 жовтня 2024.
6. SARSA [Електронний ресурс] // SARSA Reinforcement Learning. – Режим доступу: URL: https://builtin.com/machine-learning/sarsa – Назва з екрана. – Дата звернення: 24 жовтня 2024.
7. Deep Q-Networks [Електронний ресурс] // Deep Q-Network. – Режим доступу: URL: https://www.sciencedirect.com/topics/computer-science/deep-q-network – Назва з екрана. – Дата звернення: 26 жовтня 2024
8. Jupyter Notebook [Електронний ресурс] // Jupyter Notebook. – Режим доступу: URL: https://foxminded.ua/jupyter-notebook/ – Назва з екрана. – Дата звернення: 8 січня 2025.
9. Python [Електронний ресурс] // Python. – Режим доступу: URL: https://www.techtarget.com/whatis/definition/Python – Назва з екрана. – Дата звернення: 8 січня 2025.
10. Flask [Електронний ресурс] // Flask. – Режим доступу: URL:

https://careerfoundry.com/en/blog/web-development/what-is-flask/ – Назва з екрана. – Дата звернення: 9 січня 2025.

11. ReLU [Електронний ресурс] // ReLU. – Режим доступу: URL: https://www.deepchecks.com/glossary/rectified-linear-unit-relu/ – Назва з екрана. – Дата звернення: 10 січня 2025.
12. Tanh [Електронний ресурс] // Tanh. – Режим доступу: URL: https://www.ml-science.com/tanh-activation-function – Назва з екрана. – Дата звернення: 10 січня 2025.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ
Проректор Українського
державного університету
науки і технологій
Анатолій РАДКЕВИЧ
08.01.25

«ДОСЛІДЖЕННЯ МЕТОДІВ НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ ПРИ МОДЕЛЮВАННІ ПОВЕДІНКИ ІГРОВИХ АГЕНТІВ»

Технічне завдання
ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.01428-01-ЛЗ

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
08.01.25

Керівник розробки
_____Вадим ГОРЯЧКІН
08.01.25

Виконавець
_____Кирило ЯРОВИЙ
08.01.25

Нормоконтролер
_____Світлана ВОЛКОВА
08.01.25

2025

ДОДАТОК А

Технічне завдання

ЗАТВЕРДЖЕНО
44165850.01428-01-ЛЗ

«ДОСЛІДЖЕННЯ МЕТОДІВ НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ ПРИ МОДЕЛЮВАННІ ПОВЕДІНКИ ІГРОВИХ АГЕНТІВ»

Технічне завдання
44165850.01428-01

Листів 12

2025

в кожном додатку нумерація сторінок починається з 1

ЗМІСТ

1 ВВЕДЕННЯ

В сучасному світі, дослідження методів навчання нейронних мереж при моделюванні поведінки ігрових агентів стало надзвичайно актуальним напрямком сучасної комп'ютерної науки та штучного інтелекту. Нейронні мережі виступають дуже потужним інструментом для вирішення цих задач, оскільки дозволяють моделювати складні алгоритми прийняття рішень на основі навчання з досвідом.

Методи машинного навчання, а особливо, глибоке навчання та навчання з підкріпленням, створюють нові горизонти для створення цих сам ігрових агентів. Ці методи дають змогу штучному інтелекту не просто слідувати заздалегідь заданим алгоритмам, але й самостійно навчатися, аналізувати складні ігрові середовища, прогнозувати наслідки своїх дій та обирати оптимальні стратегії.

Метою представленого дослідження є комплексний аналіз та порівняння сучасних методів навчання нейронних мереж у контексті моделювання поведінки ігрових агентів. Передбачається детальне вивчення архітектур нейронних мереж, алгоритмів навчання, методик оцінки ефективності та адаптивності штучного інтелекту.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ від 07.12.22 №1209ст ректора Українського державного університету науки і технологій «Про призначення наукових керівників та затвердження тем бакалаврських робіт» за спеціальністю 121 «Інженерія програмного забезпечення» факультету «Комп'ютерних технологій і систем» по кафедрі «Комп'ютерні інформаційні технології».

Тема дипломної роботи – «Дослідження методів навчання нейронних мереж при моделюванні поведінки ігрових агентів». Керівник – зав. Кафедри КІТ – Горячкін В.М.

3 ПРИЗНАЧЕННЯ РОЗРОБКИ

Функціональне призначення – створенні адаптивних алгоритмів машинного навчання, які дозволяють штучному інтелекту вивчати стратегії, оптимізувати власну поведінку та ефективно взаємодіяти з динамічними ігровими просторами.

Експлуатаційне призначення – полягає в практичному застосуванні розроблених методів у різних галузях, які включають комп'ютерні ігри. Результат розробки буде здатен демонструвати складну поведінку, адаптуватися до змін середовища та конкурувати на рівні з людиною.

4. ВИМОГИ ДЛЯ ПРОГРАМНОГО ПРОДУКТУ

4.1. Вимоги до функціональних характеристик

Система повинна забезпечувати ефективне опрацювання вхідних даних про стан середовища, використовувати алгоритми навчання для формування стратегій поведінки, демонструвати здатність до генералізації отриманого досвіду та швидкої адаптації до змінних умов гри.

4.2. Вимоги до надійності

Повинен виконуватися контроль вхідних і вихідних нейронів для забезпечення надійної роботи нейронної мережі.

4.3. Вимоги до експлуатації

Вимоги до кліматичних умов: температура – 10-28 С, відносна вологість 20-70%. Обслуговування не потрібне.

Зі створеним програмним забезпеченням може будь-хто, хто має базовий рівень користування персональним комп’ютером.

4.4. Вимоги до складу та параметрів технічних засобів

Для коректної роботи програмного забезпечення потрібно мати процесор від 2 ядр з тактової чистотою від 3.5 ГГЦ. 4 Гб оперативної пам’яті, наявність відеокарти моделі NVIDIA з наявністю CUDA та пристрій введення/виведення інформації.

4.5. Вимоги до інформаційної та програмної сумісності

Програмне забезпечення може функціонувати на операційній системі Windows/Linux, з версією Python 3.12 та вище. А також, додаткових бібліотек NumPy (версії 2.1.3 і вище) та PyTorch (версії 2.5.1 та вище).

4.6. Вимоги до маркування і упаковки

Упаковка програмного продукту, включаючи документацію повинна бути захищена від пошкоджень різного роду (механічних, кліматичних).

На упаковці повинно бути вказана назва продукту, номер версії, мінімальні системні вимоги.

На зворотній стороні упаковки вказується розробник та його юридична адреса.

4.7 Вимоги до транспортування та зберігання

Транспортування повинно проводитись в упаковці та забезпечувати цілісність продукту.

5 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

До складу документації мають входити:

- специфікація;
- текст програми.

Програмна документація повинна відповідати вимогам ДСТУ [1].

6 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Таблиця А.1 – Стадії та етапи розробки

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вступ	16.05.2024	
2	Аналіз сучасного стану дослідження за науковими літературними статтями та дослідженнями	02.06.2024	
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує інтеграції в середовище для вирішення поставленої задачі	02.08.2024	
4	Постановка задачі та технічне завдання	01.11.2024	30%
5	Техніко-економічні показники	02.11.2024	
6	Розробка інструментальних засобів дослідження	13.11.2024	
7	Виконання досліджень	02.12.2024	60%
8	Оформлення тез доповідей	14.12.2024	
9	Оформлення статті у фаховий журнал	18.12.2024	
10	Оформлення пояснювальної записки	26.12.2024	
11	Розробка демонстраційних матеріалів	19.12.2024	
12	Подання кваліфікаційної роботи до кафедри	08.01.2025	100%
13	хист кваліфікаційної роботи на засіданні Екзаменаційної комісії	22.01.2025	

7. ПОРЯДОК І КОНТРОЛЬ ПРИЙМАННЯ

Контроль виконання здійснює керівник розробки Горячкін В. М. Прийом здійснюється уповноваженою комісією.

8. БІБЛІОГРАФІЧНИЙ СПИСОК

1. Івченко, Ю.М. Основи стандартизації програмних систем [Текст]: методичні вказівки до дипломного проектування та лабораторних робіт / уклад.: Ю. М. Івченко, В. І. Шинкаренко, В. Г. Івченко; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Д.: Вид-во Дніпропетр. нац. ун-ту залізн. трансп. ім. акад. В. Лазаряна, 2009. - 38 с.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ
Проректор Українського
державного університету
науки і технологій
Анатолій РАДКЕВИЧ
08.01.25

«ДОСЛІДЖЕННЯ МЕТОДІВ НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ ПРИ МОДЕЛЮВАННІ ПОВЕДІНКИ ІГРОВИХ АГЕНТІВ»

Текст програми
ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.01428-01 12 01-ЛЗ

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
08.01.25

Керівник розробки
_____Вадим ГОРЯЧКІН
08.01.25

Виконавець
_____Кирило ЯРОВИЙ
08.01.25

Нормоконтролер
_____Світлана ВОЛКОВА
08.02.25

2025

ДОДАТОК Б

Текст програми

ЗАТВЕРДЖЕНО
44165850.01428-01 12 01-ЛЗ

«ДОСЛІДЖЕННЯ МЕТОДІВ НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ ПРИ МОДЕЛЮВАННІ ПОВЕДІНКИ ІГРОВИХ АГЕНТІВ»

Текст програми
44165850.01428-01 12-01
Листів 36

2025

АНТОТАЦІЯ

в кожном додатку нумерація починається з 1

Документ 44165850.01428-01 12-01 «Дослідження методів навчання нейронних мереж при моделюванні поведінки ігрових агентів» входить до складу програмної документації.

У документі представлено текст додатку. Програмне забезпечення розроблено на мові Python та JavaScript за допомогою таких бібліотек як: NumPy, PyTorch, Matplotlib, Jupyter Notebook та Flask.

ЗМІСТ

1. ТЕКСТ ПРОГРАМИ

1.1.static/

1.1.1. canvas.js

```
const CANVAS_ID = "canvas";
const CANVAS_DIV_ID = "canvas_div";
const CANVAS_WIDTH = 500;
const CANVAS_HEIGHT = 500;
const CANVAS_ROWS = 5;
const CANVAS_COLS = 5;
const CANVAS_ONE_FIFTH_WIDTH =
CANVAS_WIDTH / CANVAS_COLS;
const CANVAS_ONE_FIFTH_HEIGHT =
CANVAS_HEIGHT / CANVAS_ROWS;

const MARKING_COLOR = "rgb(150, 150,
150)";
const MARKING_WIDTH = 4;

const CROSS_COLOR = "rgb(40, 40, 40)";
const CROSS_WIDTH = 5;
const CROSS_INDENT = 10;

const NOUGHT_COLOR = "rgb(100, 100,
100)";
const NOUGHT_WIDTH = 5;
const NOUGHT_INDENT = 10;

const WINNER_COLOR = "rgb(50, 205,
50)";
const WINNER_WIDTH = 7;

function getCanvas() {
    return
document.getElementById(CANVAS_ID);
}

function createCanvas() {
    if (getCanvas() != null) {
        getCanvas().remove();
    }

    let canvas =
document.createElement('canvas');
    canvas.id = CANVAS_ID;
    canvas.width = CANVAS_WIDTH;
    canvas.height = CANVAS_HEIGHT;


document.getElementById(CANVAS_DIV_I
D).appendChild(canvas);
}

function getCoordinates(event) {
    let rect =
getCanvas().getBoundingClientRect();

    let x = event.clientX - rect.left;
    let y = event.clientY - rect.top;

    return {"x": x, "y": y};
}

function getMatrixIndex(coordinates) {
    let row = Math.floor(coordinates['y'] /
CANVAS_ONE_FIFTH_WIDTH);
    let col = Math.floor(coordinates['x'] /
CANVAS_ONE_FIFTH_HEIGHT);

    return {"row": row, "col": col};
}

function vectorToMatrixIndex(indexVector) {
    const index = indexVector['index'];

    let row = Math.floor(index /
CANVAS_ROWS);
    let col = index % CANVAS_COLS;

    return {"row": row, "col": col};
}

function matrixToVectorIndex(indexMatrix)
{
    let index = CANVAS_ROWS *
indexMatrix['row'] + indexMatrix['col'];
```

```
    return {"index": index};
}

function addCanvasClick() {
    let canvas = getCanvas();

    canvas.addEventListener('mousedown',
function(event) {
        userClick(event);
    });
}

function checkboxClick(checkboxId = 0) {
    let checkboxes =
document.querySelectorAll('input[name="my
_checkbox"]');

    checkboxes.forEach(function(checkbox) {
        checkbox.checked = false;
    });

    checkboxes[checkboxId].checked = true;
    botId = checkboxId;
}

class Draw {
    constructor(x1, y1, x2, y2, strokeStyle,
lineWidth) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;

        this.ctx = getCanvas().getContext("2d");
        this.ctx.strokeStyle = strokeStyle;
        this.ctx.lineWidth = lineWidth;

        this.currentX = x1;
        this.currentY = y1;

        this.deltaX = (x2 - x1) / 25;
        this.deltaY = (y2 - y1) / 25;
        this.step = 0;
    }

    drawLine() {
        this.ctx.beginPath();
        this.ctx.moveTo(this.x1, this.y1);
        this.ctx.lineTo(this.x2, this.y2);
        this.ctx.stroke();
    }

    /*
    drawLine() {
        if (this.step > 25) return;

        this.ctx.beginPath();
        this.ctx.moveTo(this.currentX,
this.currentY);
        this.currentX += this.deltaX;
        this.currentY += this.deltaY;
        this.ctx.lineTo(this.currentX,
this.currentY);
        this.ctx.stroke();
        this.step++;
        requestAnimationFrame(() =>
this.drawLine());
    }
    */
}

function drawMarking() {
    for (let i = 1; i <= CANVAS_COLS - 1;
i++)
        new
Draw(CANVAS_ONE_FIFTH_WIDTH * i,
0, CANVAS_ONE_FIFTH_WIDTH * i,
CANVAS_HEIGHT, MARKING_COLOR,
MARKING_WIDTH).drawLine();

    for (let i = 1; i <= CANVAS_ROWS - 1;
i++)
        new Draw(0,
CANVAS_ONE_FIFTH_HEIGHT * i,
CANVAS_WIDTH,
CANVAS_ONE_FIFTH_HEIGHT * i,
MARKING_COLOR,
MARKING_WIDTH).drawLine();
}

function drawCross(coordinatesPart) {
    const row = coordinatesPart['row'];
    const col = coordinatesPart['col'];
```

```
    let x1 = CANVAS_ONE_FIFTH_WIDTH
* col + CROSS_INDENT;
    let y1 = CANVAS_ONE_FIFTH_HEIGHT
* row + CROSS_INDENT;

    let x2 = CANVAS_ONE_FIFTH_WIDTH
* col + CANVAS_ONE_FIFTH_WIDTH -
CROSS_INDENT;
    let y2 = CANVAS_ONE_FIFTH_HEIGHT
* row + CANVAS_ONE_FIFTH_HEIGHT -
CROSS_INDENT;

    new Draw(x1, y1, x2, y2,
CROSS_COLOR,
CROSS_WIDTH).drawLine();
    new Draw(x2, y1, x1, y2,
CROSS_COLOR,
CROSS_WIDTH).drawLine();
}

function drawNought(coordinatesPart) {
    const row = coordinatesPart['row'];
    const col = coordinatesPart['col'];

    let ctx = getCanvas().getContext("2d");
    ctx.strokeStyle = NOUGHT_COLOR;
    ctx.lineWidth = NOUGHT_WIDTH;

    let centerX =
CANVAS_ONE_FIFTH_WIDTH * col +
CANVAS_ONE_FIFTH_WIDTH / 2;
    let centerY =
CANVAS_ONE_FIFTH_HEIGHT * row +
CANVAS_ONE_FIFTH_HEIGHT / 2;

    let radiusX =
CANVAS_ONE_FIFTH_WIDTH / 2 -
NOUGHT_INDENT;
    let radiusY =
CANVAS_ONE_FIFTH_HEIGHT / 2 -
NOUGHT_INDENT;

    ctx.beginPath();
    ctx.ellipse(centerX, centerY, radiusX,
radiusY, 0, 0, 2 * Math.PI);
    ctx.stroke();
}

function drawWinner() {
    let x = (positionCol) =>
CANVAS_ONE_FIFTH_WIDTH *
positionCol +
CANVAS_ONE_FIFTH_WIDTH / 2;
    let y = (positionRow) =>
CANVAS_ONE_FIFTH_HEIGHT *
positionRow +
CANVAS_ONE_FIFTH_HEIGHT / 2;

    let x1 = x(winPositions[0][1]);
    let y1 = y(winPositions[0][0])

    let x2 = x(winPositions[3][1])
    let y2 = y(winPositions[3][0])

    new Draw(x1, y1, x2, y2,
WINNER_COLOR,
WINNER_WIDTH).drawLine();
}
```

1.1.2. gameBrain.js

```
const BOARD_SIZE = 25;

const CROSS = 1;
const NOUGHT = -1;
const EMPTY = 0;

let board = null;
let playerSide = null;
let botSide = null;
let botId = 0;
let botPositionVector = null;

let isEnd = null;
let canPlayerClick = null;
let winPositions = null;
let winnerFigure = null;

function newGame(playerForSide = CROSS)
{
   createCanvas();
   addCanvasClick();
   drawMarking();
   removeResultHeader();
   scoreboard();

   board = new
Array(BOARD_SIZE).fill(EMPTY);
   playerSide = (playerForSide == CROSS ?
CROSS : NOUGHT);
   botSide = (playerForSide == CROSS ?
NOUGHT : CROSS);
   canPlayerClick = (playerSide == CROSS ?
true : false);

   isEnd = false;
   winPositions = null;
   winnerFigure = null;
   botPositionVector = null;

   if (botSide == CROSS) botMove();
}

function botMove() {
   $(document).ready(function() {
      $.ajax({
         url: "/bot_move",
         type: "POST",
         data: JSON.stringify({
            "board": board,
            "bot_side": botSide,
            "bot_id": botId
         }),
         dataType: "json",
         contentType: "application/json",
         success: function(response) {
            board = response['board'];
            isEnd = response['is_end'];
            winPositions =
response['win_positions'];
            winnerFigure =
response['winner_figure'];
            botPositionVector =
response['bot_position'];

            if (botPositionVector != null) {
               let botPositionMatrix =
vectorToMatrixIndex({"index":
botPositionVector});
               botSide == CROSS ?
drawCross(botPositionMatrix) :
drawNought(botPositionMatrix);
               canPlayerClick = true;
            }

            if (isEnd) {
               if (winPositions != null) {
                  drawWinner();
                  winnerFigure == playerSide ?
playerWins++ : botWins++;
                  scoreboard();
               }
               createResultHeader();
            }
         },
         error: function(error) {
            console.error(error);
         }
      });
   });
}
```

```
function userClick(event) {
    const coordinates = getCoordinates(event);
    const indexMatrix =
getMatrixIndex(coordinates);
    const indexVector =
matrixToVectorIndex(indexMatrix);

    canPlayerClick = false;

    if (board[indexVector['index']] != EMPTY
|| isEnd || canPlayerClick) return;

    board[indexVector['index']] = playerSide;
    playerSide == CROSS ?
drawCross(indexMatrix) :
drawNought(indexMatrix);
    botMove();
}
```

1.1.3 gameResult.js

```
const SCORE_BOARD_DIV_ID = "score_board";
const SCRORE_HEADER_ID = "score_header";

const RESULT_HEADER_DIV_ID = "after_game";
const RESULT_HEADER_ID = "game_winner";

let playerWins = 0;
let botWins = 0;

function scoreboard() {
    if (document.getElementById(SCRORE_HEADER_ID) != null)

document.getElementById(SCRORE_HEADER_ID).remove();

    let header = document.createElement('h2');
    header.id = SCRORE_HEADER_ID;
    header.innerHTML = `Перемоги гравця ${playerWins}/${botWins}`;


document.getElementById(SCORE_BOARD_DIV_ID).appendChild(header);
}

function removeResultHeader() {
    if (document.getElementById(RESULT_HEADER_ID) != null)

document.getElementById(RESULT_HEADER_ID).remove();
}

function createResultHeader() {
    let header = document.createElement('h1');
    header.id = RESULT_HEADER_ID;
    if (winnerFigure == 0) header.innerHTML = "У грі нічия";
    else header.innerHTML = `Перемога для ${winnerFigure == CROSS ? "X" : "0"}`;


document.getElementById(RESULT_HEADER_DIV_ID).appendChild(header);
}
```

1.1.4. style.css

```
div {
    margin: 10px;
    display: block;
    text-align: center;
}

input[type=button] {
    margin: 5px;
    font-size: 1.4em;
    color: rgb(0, 0, 0);
    background-color: rgb(155, 210, 255);
    border-radius: 8px;
    border: 0px solid;
}


input[type=button]:hover {
    background-color: rgb(197, 228, 255);
}

input[type=button]:active {
    background-color: rgb(224, 241, 255);
}

canvas {
    margin: 20px;
    border: 2px solid black;
}
```

1.2 templates/

1.2.1 index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>TicTacToe Neural</title>
    <link rel="shortcut icon" href="{{
url_for('static', filename='favicon.ico')}}">
    <meta name="viewport"
content="width=device-width, initial-
scale=1"/>
    <link rel="stylesheet"
href="static/style.css">
    <script
src="https://ajax.googleapis.com/ajax/libs/jqu
ery/3.7.1/jquery.min.js"></script>
    <script src="static/canvas.js"></script>
    <script
src="static/gameBrain.js"></script>
    <script
src="static/gameResult.js"></script>
  </head>

  <body>
    <div id="new_game">
      <input type="button"
id="new_game_x" value="Грати за X"
onclick="newGame(playerForSide = 1)">
      <input type="button"
id="new_game_o" value="Грати за 0"
onclick="newGame(playerForSide = -1)">
    </div>

    <div id="models">
      <input type="checkbox"
id="easy_bot" name="my_checkbox"
onclick="checkboxClick(checkboxId = 0)">
      <label for="easy_bot">Легкий
бот</label>

      <input type="checkbox" id="sl_bot"
name="my_checkbox"
onclick="checkboxClick(checkboxId = 1)">
      <label for="sl_bot">SL
модель</label>

      <input type="checkbox" id="rl_bot"
name="my_checkbox"
onclick="checkboxClick(checkboxId = 2)">
      <label for="rl_bot">RL
модель</label>
    </div>

    <div id="canvas_div"></div>

    <div id="after_game"></div>

    <div id="score_board"></div>

    <script>
      newGame();
      checkboxClick();
    </script>
  </body>
</html>
```

2 tictactoe/

2.1.1 reinforcement/

2.1.1.1 rl_bot.py

```
import torch
import torch.nn.functional as F
from tictactoe.reinforcement.rl_model import
RLModel, MODEL_CROSS_PATH,
MODEL_NOUGHT_PATH
from tictactoe.board import Board

class RLBot:
    BEST_OF = 4

    def __init__(self, model_path: str):
        device = torch.device("cpu")
        self.model = RLModel()

self.model.load_state_dict(torch.load(model_
path, weights_only=True,
map_location=device))
        self.model.eval()

    def get_move(self, board: Board):
        empty_spots = board.get_empty_spots()
        if len(empty_spots) >=
RLBot.BEST_OF: CHOICES_COUNT =
RLBot.BEST_OF
        else: CHOICES_COUNT = 1

        values =
self.model(torch.tensor(board.board_to_tenso
r_bits())).view(-1)
        valid_values = values[empty_spots]

        best_values, _ = torch.topk(valid_values,
CHOICES_COUNT)
        probabilities = F.softmax(best_values,
dim=0)
        index_probabilities =
torch.multinomial(probabilities,
num_samples=1).item()

        winner =
best_values[index_probabilities]
        index_values = torch.nonzero(values ==
winner).squeeze().item()

        return index_values

class RLBotCross(RLBot):
    def __init__(self):
        super(RLBotCross,
self).__init__(MODEL_CROSS_PATH)

class RLBotNought(RLBot):
    def __init__(self):
        super(RLBotNought,
self).__init__(MODEL_NOUGHT_PATH)
```

2.1.1.2. rl_model.py

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import random
from collections import deque
from tictactoe.board import Board

MODEL_CROSS_PATH =
"tictactoe/reinforcement/data/model_cross.pth
"
MODEL_NOUGHT_PATH =
"tictactoe/reinforcement/data/model_nought.p
th"

device = torch.device("cuda:0" if
torch.cuda.is_available() else "cpu")

class RLModel(nn.Module):
    def __init__(self):
        super(RLModel, self).__init__()
        self.a1 = nn.Conv2d(in_channels=2,
out_channels=4, kernel_size=3, padding=1)
        self.a2 = nn.Conv2d(in_channels=4,
out_channels=8, kernel_size=3, padding=1)
        self.a3 = nn.Conv2d(in_channels=8,
out_channels=16, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(in_features=16*5*5,
out_features=16*5*5)
        self.fc2 = nn.Linear(in_features=16*5*5,
out_features=25)

    def forward(self, x):
        x = F.relu(self.a1(x))
        x = F.relu(self.a2(x))
        x = F.relu(self.a3(x))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class RLAgent:
    def __init__(self, model: RLModel):
        self.model = model.to(device)
        self.target_model =
RLModel().to(device)

self.target_model.load_state_dict(model.state
_dict())

        self.optimizer =
torch.optim.Adam(self.model.parameters(),
lr=0.001)
        self.memory = deque(maxlen=1000)

        self.gamma = 1
        self.epsilon = 1.0
        self.epsilon_decay = 0.98
        self.epsilon_min = 0.05

    def remember(self, board: np.ndarray,
position: int, reward: int, board_next:
np.ndarray, endgame: bool):
        self.memory.append((board, position,
reward, board_next, endgame))

    def act(self, board: Board):
        empty_spots = board.get_empty_spots()

        if random.random() <= self.epsilon:
            return random.choice(empty_spots)

        with torch.no_grad():
            q_values =
self.model(torch.tensor(board.board_to_tenso
r_bits(), device=device))
            q_values = q_values.view(-1)
            valid_q_values =
q_values[empty_spots]

            max_value =
torch.max(valid_q_values).item()
            best_index = torch.nonzero(q_values
== max_value).squeeze().item()

            return best_index

    def replay(self, batch_size: int):
        if len(self.memory) < batch_size:
```

```
            return

        batch = random.sample(self.memory,
batch_size)

        boards = []
        targets = []

        for board_tensor, position, reward,
board_next_tensor, endgame in batch:
            q_values =
self.model(torch.tensor(board_tensor,
device=device))

            if endgame:
                max_q_value_next =
torch.tensor(0.0, device=device)
            else:
                q_values_next =
self.target_model(torch.tensor(board_next_te
nsor, device=device))
                max_q_value_next =
torch.max(q_values_next)

            target = q_values.clone().detach()
            target[0, position] = reward +
self.gamma * max_q_value_next * (1 -
int(endgame))


boards.append(board_tensor.reshape((2, 5,
5)))
            targets.append(target)

        boards_tensor =
torch.tensor(np.array(boards),
dtype=torch.float32, device=device)
        targets_tensor =
torch.stack(targets).squeeze(1).to(device)

        predictions = self.model(boards_tensor)
        loss = F.mse_loss(predictions,
targets_tensor)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def update_target_model(self):

self.target_model.load_state_dict(self.model.s
tate_dict())

def get_parameters_count(model: RLModel):
    return sum(p.numel() for p in
model.parameters() if p.requires_grad)

if __name__ == "__main__":
    model_cross = RLModel()
    agent_cross = RLAgent(model_cross)

    model_nought = RLModel()
    agent_nought = RLAgent(model_nought)

    for epoch in range(100):
        board = Board()
        current_player = Board.CROSS

        total_reward_cross = 0
        total_reward_nought = 0
        game_steps = 0

        while True:
            if current_player == Board.CROSS:
                position = agent_cross.act(board)
                before_board =
board.board_to_tensor_bits()
                reward = board.step(position,
Board.CROSS)

agent_cross.remember(before_board,
position, reward,
board.board_to_tensor_bits(),
board.endgame)
                current_player = Board.NOUGHT
                total_reward_cross += reward
                if board.endgame: break
            else:
                position = agent_nought.act(board)
                before_board =
board.board_to_tensor_bits()
```

```
            reward = board.step(position,
Board.NOUGHT)

agent_nought.remember(before_board,
position, reward,
board.board_to_tensor_bits(),
board.endgame)
            current_player = Board.CROSS
            total_reward_nought += reward
            if board.endgame: break

        game_steps += 1

    agent_cross.replay(32)
    agent_nought.replay(32)

    if epoch % 5 == 0:
        agent_cross.update_target_model()
        agent_nought.update_target_model()

    if epoch % 10 == 0:
        print(epoch, game_steps, end=" ")
        print(f"cross: [{total_reward_cross},
{(agent_cross.epsilon):.4f},
{len(agent_cross.memory)}]", end=" ")
        print(f"nought:
[{total_reward_nought},
{(agent_nought.epsilon):.4f},
{len(agent_nought.memory)}]")

  torch.save(agent_cross.model.state_dict(),
MODEL_CROSS_PATH)

torch.save(agent_nought.model.state_dict(),
MODEL_NOUGHT_PATH)

  print(f"Model: RLModel(Cross),
Parameters:
{get_parameters_count(model_cross)}")
  print(f"Model: RLModel(Nought),
Parameters:
{get_parameters_count(model_nought)}")
```

2.1.1.3. rl_play.py

```
from tictactoe.reinforcement.rl_bot import
RLBotCross, RLBotNought
from tictactoe.board import Board
import sys

def player_side(board: Board, figure: int):
    print(board)
    print(board.get_empty_spots())
    position = int(input("=> "))
    board.set_move(position, figure)

    return board.get_winner()[0]

def bot_side(board: Board, figure: int, rl_bot:
RLBotCross|RLBotNought):
    position = rl_bot.get_move(board)
    board.set_move(position, figure)

    return board.get_winner()[0]

def after_game(board: Board):
    print(board)
    print(board.get_winner())

class RLPlay:
    def __init__(self):
        self.rlbot_cross = RLBotCross()
        self.rlbot_nought = RLBotNought()

    def player_x_vs_rlbot(self):
        board = Board()

        while True:
            if player_side(board, Board.CROSS):
break
            if bot_side(board, Board.NOUGHT,
self.rlbot_nought): break

        after_game(board)

    def player_o_vs_rlbot(self):
        board = Board()

        while True:
            if bot_side(board, Board.CROSS,
self.rlbot_cross): break
            if player_side(board,
Board.NOUGHT): break

        after_game(board)

    def rlbot_x_vs_randbot(self):
        board = Board()

        while True:
            if bot_side(board, Board.CROSS,
self.rlbot_cross): break


board.random_move(Board.NOUGHT)
            if board.get_winner()[0]: break

        return board.get_figure_winner(),
board.get_figures_count()

    def rlbot_o_vs_randbot(self):
        board = Board()

        while True:
            board.random_move(Board.CROSS)
            if board.get_winner()[0]: break

            if bot_side(board, Board.NOUGHT,
self.rlbot_nought): break

        return board.get_figure_winner(),
board.get_figures_count()

if __name__ == "__main__":
    rl_play = RLPlay()

    if len(sys.argv) == 1:
        rl_play.player_x_vs_rlbot()
    else:
        if sys.argv[1].lower() == "x":
            rl_play.player_x_vs_rlbot()
        elif sys.argv[1].lower() == "o":
            rl_play.player_o_vs_rlbot()
```

2.1.1.4. simulate.py

```
from tictactoe.reinforcement.rl_play import
RLPlay
from tictactoe.board import Board
import sys
from tictactoe.stats import Stats, GameStat,
Graph, get_results_str

if __name__ == "__main__":
    game_stat = GameStat()
    rl_play = RLPlay()

    if len(sys.argv) == 2:
        game_stat.games_count =
int(sys.argv[1])

    rl_stat = Stats("RL")
    rand_stat = Stats("Easy")

    for i in range(game_stat.games_count):
        result, figures_count =
rl_play.rlbot_x_vs_randbot()
        game_stat.figures_count_one +=
figures_count

        if result == Board.CROSS:
            rl_stat.x_wins += 1
            rl_stat.x_games_ids.append(i)
        elif result == Board.NOUGHT:
            rand_stat.o_wins += 1
            rand_stat.o_games_ids.append(i)


rl_stat.x_accumulated.append(rl_stat.x_wins)

rand_stat.o_accumulated.append(rand_stat.o_
wins)

    for i in range(game_stat.games_count):
        result, figures_count =
rl_play.rlbot_o_vs_randbot()
        game_stat.figures_count_two +=
figures_count

        if result == Board.CROSS:
            rand_stat.x_wins += 1
            rand_stat.x_games_ids.append(i)
        elif result == Board.NOUGHT:
            rl_stat.o_wins += 1
            rl_stat.o_games_ids.append(i)


rand_stat.x_accumulated.append(rand_stat.x_
wins)

rl_stat.o_accumulated.append(rl_stat.o_wins)

    print(get_results_str(game_stat, rl_stat,
rand_stat))
    Graph(game_stat.games_count, rl_stat,
rand_stat).draw_graph()
```

2.1.2. supervised/

2.1.2.1. batch.py

```
from tictactoe.board import Board
from tictactoe.supervised.dataset import
read_csv_file, TRAIN_CSV_PATH,
VALIDATE_CSV_PATH
import numpy as np

TRAIN_NPZ_PATH =
"tictactoe/supervised/data/train.npz"
VALIDATE_NPZ_PATH =
"tictactoe/supervised/data/validate.npz"

class Batch:
    @staticmethod
    def convert_to_batch_bits(game: list[str]):
        cross = np.zeros((Board.BOARD_SIZE),
dtype='float32')
        nought =
np.zeros((Board.BOARD_SIZE),
dtype='float32')

        for i, figure in enumerate(game):
            cross[i] = 1 if int(figure) == 1 else 0
            nought[i] = 1 if int(figure) == -1 else 0

        return np.concatenate((cross, nought),
dtype='float32').reshape((1, 2, 5, 5))

    @staticmethod
    def get_batch_in(games: list[str]):
        batch_in = np.zeros((len(games), 2, 5, 5),
dtype='float32')

        for i, game in enumerate(games):
            batch_in[i] =
Batch.convert_to_batch_bits(game[0:25])[0]

        return batch_in

    @staticmethod
    def get_batch_out(games: list[str]):
        batch_out = np.zeros((len(games), 1),
dtype='float32')

        for i, result in enumerate(games):
            batch_out[i] = np.float32(result[25])

        return batch_out

def get_results(path: str, batch_in: np.ndarray,
batch_out: np.ndarray):
    return f"File \"{path}\" batch_in:
{batch_in.shape} batch_out:
{batch_out.shape}"

if __name__ == "__main__":
    data = read_csv_file(TRAIN_CSV_PATH)
    batch_in = Batch.get_batch_in(data)
    batch_out = Batch.get_batch_out(data)
    np.savez(TRAIN_NPZ_PATH,
arr1=batch_in, arr2=batch_out)
    print(get_results(TRAIN_NPZ_PATH,
batch_in, batch_out))

    data =
read_csv_file(VALIDATE_CSV_PATH)
    batch_in = Batch.get_batch_in(data)
    batch_out = Batch.get_batch_out(data)
    np.savez(VALIDATE_NPZ_PATH,
arr1=batch_in, arr2=batch_out)

print(get_results(VALIDATE_NPZ_PATH,
batch_in, batch_out))
```

2.1.2.2. dataset.py

```
from tictactoe.board import Board
import csv
import sys

TRAIN_CSV_PATH =
"tictactoe/supervised/data/train.csv"
VALIDATE_CSV_PATH =
"tictactoe/supervised/data/validate.csv"

FILE_NEWLINE = ''
FILE_DELIMITER = ','

def write_csv_file(data: list, file_name: str):
    with open(file_name, "w",
newline=FILE_NEWLINE) as csvfile:
        csv_write = csv.writer(csvfile,
delimiter=FILE_DELIMITER)

        for row in data:
            csv_write.writerow(row)

def read_csv_file(file_name: str):
    data = []

    with open(file_name, "r",
newline=FILE_NEWLINE) as csvfile:
        reader = csv.reader(csvfile,
delimiter=FILE_DELIMITER)
        for row in reader:
            data.append(row)

    return data

def create_game():
    board = Board()

    while True:
        board.random_move(Board.CROSS)
        if board.get_winner()[0]: break

        board.random_move(Board.NOUGHT)
        if board.get_winner()[0]: break

    winner = board.get_figure_winner()
    result = board.get_board() + [winner]
    return result, winner

def random_games(count: int):
    games = []

    cross_wins = 0
    nought_wins = 0
    draws = 0

    for _ in range(count):
        result, winner = create_game()
        games.append(result)

        if winner == 1: cross_wins += 1
        elif winner == -1: nought_wins += 1
        else: draws += 1

    return games, cross_wins, nought_wins,
draws

def random_games_equal(count: int):
    games = []

    cross_wins = 0
    nought_wins = 0
    draws = 0

    def game_loop(for_who: int, wins_goal:
int):
        win_count = 0

        while True:
            game, winner = create_game()
            if winner == for_who:
                win_count += 1
                games.append(game)
            if win_count == wins_goal: break

        return win_count

    cross_wins = game_loop(1, count)
    nought_wins = game_loop(-1, count)
    #draws = game_loop(0, count)
```

```
    return games, cross_wins, nought_wins,
draws

def collect_games(count_games: int,
csv_path: str):
    games, cross_wins, nought_wins, draws =
random_games_equal(count_games)
    #games, cross_wins, nought_wins, draws =
random_games(count_games)
    write_csv_file(games, csv_path)

    print(f"{csv_path} games:
[{count_games}]", end=" ")
    print(f"x_wins: [{cross_wins}] o_wins:
[{nought_wins}] draws: [{draws}]")

if __name__ == "__main__":
    train_lenght = 150
    validate_lenght = 75

    if len(sys.argv) == 3:
        train_lenght = int(sys.argv[1])
        validate_lenght = int(sys.argv[2])

    collect_games(train_lenght,
TRAIN_CSV_PATH)
    collect_games(validate_lenght,
VALIDATE_CSV_PATH)
```

2.1.2.3. simulate.py

```
import sys
from tictactoe.supervised.sl_play import
SLPlay
from tictactoe.board import Board
from tictactoe.stats import Stats, GameStat,
Graph, get_results_str

if __name__ == "__main__":
    game_stat = GameStat()

    if len(sys.argv) == 2:
        game_stat.games_count =
int(sys.argv[1])

    play = SLPlay()

    sl_stat = Stats("SL")
    rand_stat = Stats("Easy")

    for i in range(game_stat.games_count):
        result, figures_count =
play.slbot_x_vs_randbot()
        game_stat.figures_count_one +=
figures_count

        if result == Board.CROSS:
            sl_stat.x_wins += 1
            sl_stat.x_games_ids.append(i)
        elif result == Board.NOUGHT:
            rand_stat.o_wins += 1
            rand_stat.o_games_ids.append(i)


sl_stat.x_accumulated.append(sl_stat.x_wins)

rand_stat.o_accumulated.append(rand_stat.o_
wins)

    for i in range(game_stat.games_count):
        result, figures_count =
play.slbot_o_vs_randbot()
        game_stat.figures_count_two +=
figures_count

        if result == Board.CROSS:
            rand_stat.x_wins += 1
            rand_stat.x_games_ids.append(i)
        elif result == Board.NOUGHT:
            sl_stat.o_wins += 1
            sl_stat.o_games_ids.append(i)


rand_stat.x_accumulated.append(rand_stat.x_
wins)

sl_stat.o_accumulated.append(sl_stat.o_wins)

    print(get_results_str(game_stat, sl_stat,
rand_stat))
    Graph(game_stat.games_count, sl_stat,
rand_stat).draw_graph()
```

2.1.2.4. sl_bot.py

```
from tictactoe.supervised.sl_model import
SLModel, MODEL_PATH
import torch
from tictactoe.supervised.batch import Batch
from tictactoe.board import Board
import heapq

class SLBot:
    BEST_OF = 4

    def __init__(self):
        device = torch.device("cpu")
        self.model = SLModel()

self.model.load_state_dict(torch.load(MODE
L_PATH, weights_only=True,
map_location=device))
        self.model.eval()

    def get_model_move(self, board: Board,
figure: int):
        empty_spots = board.get_empty_spots()
        variants = {}

        for position in empty_spots:
            board_int = board.get_board()
            board_int[position] = figure

            with torch.no_grad():
                inputs =
torch.from_numpy(Batch.convert_to_batch_b
its(board_int))
                variants[position] =
float(self.model(inputs))

        if len(empty_spots) >=
SLBot.BEST_OF: CHOICES_COUNT =
SLBot.BEST_OF
        else: CHOICES_COUNT = 1

        if figure == Board.CROSS:
            best_moves =
heapq.nlargest(CHOICES_COUNT, variants,
key=variants.get)
            values = torch.tensor([variants[i] for i
in best_moves], dtype=torch.float32)
            probabilities = values / values.sum()
            probabilities =
torch.clamp(probabilities, min=0)
            index =
torch.multinomial(probabilities,
num_samples=1).item()

            #return max(variants,
key=variants.get)
            return best_moves[index]
        else:
            best_moves =
heapq.nsmallest(CHOICES_COUNT,
variants, key=variants.get)

            values = torch.tensor([variants[i] for i
in best_moves], dtype=torch.float32)
            weights = 1 / torch.abs(values)
            probabilities = weights /
weights.sum()
            index =
torch.multinomial(probabilities,
num_samples=1).item()

            #return min(variants,
key=variants.get)
            return best_moves[index]

    def __call__(self, board: Board, figure: int):
        return self.get_model_move(board,
figure)
```

2.1.2.5. sl_model.py

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset,
DataLoader
from tictactoe.supervised.batch import
TRAIN_NPZ_PATH

MODEL_PATH =
"tictactoe/supervised/data/model.pth"

class MyData(Dataset):
    def __init__(self, data_npz_path):
        data = np.load(data_npz_path)
        self.inputs = data['arr1']
        self.labels = data['arr2']

    def __len__(self):
        return len(self.inputs)

    def __getitem__(self, index):
        return self.inputs[index],
self.labels[index]

class SLModel(nn.Module):
    def __init__(self):
        super(SLModel, self).__init__()
        self.a1 = nn.Conv2d(in_channels=2,
out_channels=4, kernel_size=3, padding=1)
        self.a2 = nn.Conv2d(in_channels=4,
out_channels=8, kernel_size=3, padding=1)
        self.a3 = nn.Conv2d(in_channels=8,
out_channels=16, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(in_features=16*5*5,
out_features=16*5*5)
        self.fc2 = nn.Linear(in_features=16*5*5,
out_features=1)

    def forward(self, x):
        x = F.relu(self.a1(x))
        x = F.relu(self.a2(x))
        x = F.relu(self.a3(x))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.tanh(x)

def get_parameters_count(model: SLModel):
    return sum(p.numel() for p in
model.parameters() if p.requires_grad)

if __name__ == "__main__":
    model = SLModel()
    optimizer =
torch.optim.Adam(model.parameters(),
lr=0.001)
    criterion = nn.MSELoss()
    model.train()

    device = torch.device("cuda:0" if
torch.cuda.is_available() else "cpu")
    model.to(device)

    my_data = MyData(TRAIN_NPZ_PATH)
    data_loader = DataLoader(my_data,
batch_size=32, shuffle=True)

    for epoch in range(100):
        for i, (inputs, labels) in
enumerate(data_loader):
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        print(f"{epoch} {loss.item()}")

    torch.save(model.state_dict(),
MODEL_PATH)
    print(f"Model: SLModel, Parameters:
{get_parameters_count(model)}")
```

2.1.2.6 sl_play.py

```
import sys
from tictactoe.supervised.sl_bot import SLBot
from tictactoe.board import Board

def player_side(board: Board, figure: int):
    print(board)
    print(board.get_empty_spots())
    position = int(input("=> "))
    board.set_move(position, figure)

    return board.get_winner()[0]

def bot_side(board: Board, figure: int, slbot: SLBot):
    position = slbot.get_model_move(board, figure)
    board.set_move(position, figure)

    return board.get_winner()[0]

def after_game(board: Board):
    print(board)
    print(board.get_winner())

class SLPlay:
    def __init__(self):
        self.slbot = SLBot()

    def player_x_vs_slbot(self):
        board = Board()

        while True:
            if player_side(board, Board.CROSS): break
            if bot_side(board, Board.NOUGHT, self.slbot): break

        after_game(board)

    def player_o_vs_slbot(self):
        board = Board()

        while True:
            if bot_side(board, Board.CROSS, self.slbot): break
            if player_side(board, Board.NOUGHT): break

        after_game(board)

    def slbot_x_vs_randbot(self):
        board = Board()

        while True:
            if bot_side(board, Board.CROSS, self.slbot): break


board.random_move(Board.NOUGHT)
            if board.get_winner()[0]: break

        return board.get_figure_winner(), board.get_figures_count()

    def slbot_o_vs_randbot(self):
        board = Board()

        while True:
            board.random_move(Board.CROSS)
            if board.get_winner()[0]: break

            if bot_side(board, Board.NOUGHT, self.slbot): break

        return board.get_figure_winner(), board.get_figures_count()

if __name__ == "__main__":
    play = SLPlay()

    if len(sys.argv) == 1:
        play.player_x_vs_slbot()
    else:
        if sys.argv[1].lower() == "x":
            play.player_x_vs_slbot()
        elif sys.argv[1].lower() == "o":
            play.player_o_vs_slbot()
```

2.1.2.7 validate.py

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from tictactoe.supervised.sl_model import
MyData, SLModel, MODEL_PATH
from tictactoe.supervised.batch import
VALIDATE_NPZ_PATH

if __name__ == "__main__":
    device = torch.device("cuda:0" if
torch.cuda.is_available() else "cpu")

    model = SLModel()

model.load_state_dict(torch.load(MODEL_P
ATH, weights_only=True,
map_location=device))
    model = model.to(device)
    model.eval()

    criterion = nn.MSELoss()

    my_data =
MyData(VALIDATE_NPZ_PATH)
    data_loader = DataLoader(my_data,
batch_size=1, shuffle=True)

    data_length = len(data_loader)
    validation_loss = 0.0
    accuracy = 0

    with torch.no_grad():
        for i, (inputs, labels) in
enumerate(data_loader):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            loss = criterion(outputs, labels)

            difference = (labels - outputs).abs()
            correct = (difference <=
0.5).sum().item()

            accuracy += correct
            validation_loss += loss.item()

            #print(f"label: [{float(labels):.2f}]
output: [{float(outputs):.2f}]", end=" ")

#print(f"difference:[{float(difference):.4f}]
correct: [{correct}]")

    validation_loss /= data_length
    print(f"Data length: {data_length}
validation loss: {(validation_loss):.4f},
accuracy:
{(accuracy/data_length*100):.2f}%")
```

2.1.3 board.py

```
import numpy as np
import random
import copy

class Board:
    BOARD_SIZE = 25

    CROSS = 1
    NOUGHT = -1
    EMPTY = 0

    CROSS_BIT = 1
    NOUGHT_BIT = 1
    EMPTY_BIT = 0

    CROSS_STR = "X"
    NOUGHT_STR = "0"
    EMPTY_STR = " "

    REWARD_WIN = 5
    REWARD_DEFEAT = -5
    REWARD_DRAW = -2
    REWARD_ONGOING = -0.5

    def __init__(self, board: list[int]=None):
        if board == None:
            self.__board = [self.EMPTY] *
self.BOARD_SIZE
        else:
            assert len(board) ==
self.BOARD_SIZE, "len() of board more or
less than Board.BOARD_SIZE"
            self.__board = board
        self.endgame = False

    def get_empty_spots(self):
        return [i for i, figure in
enumerate(self.__board) if figure ==
self.EMPTY]

    def set_move(self, position: int, figure: int):
        assert position >= 0 and position <
self.BOARD_SIZE, "position is out of range"
        assert self.__board[position] ==
self.EMPTY, "position is not empty"
        self.__board[position] = figure

    def step(self, position: int, figure: int):
        self.set_move(position, figure)
        self.endgame, win_positions =
self.get_winner()

        if self.endgame != True:
            return self.REWARD_ONGOING
        elif self.endgame and win_positions ==
None:
            return self.REWARD_DRAW
        else:
            winner_figure =
self.get_figure_winner()
            return self.REWARD_WIN if figure
== winner_figure else
self.REWARD_DEFEAT

    def random_move(self, figure: int):
        empty_spots = self.get_empty_spots()
        assert len(empty_spots) > 0, "there are
not empty spots for random_move()"

        position = random.choice(empty_spots)
        self.__board[position] = figure

    def get_random_move(self):
        empty_spots = self.get_empty_spots()
        assert len(empty_spots) > 0, "there are
not empty spots for random_move()"

        return random.choice(empty_spots)

    def get_winner(self):
        matrix = [self.__board[i:i + 5] for i in
range(0, self.BOARD_SIZE, 5)]
        seq = lambda seqw: all(x == seqw[0] for
x in seqw) and seqw[0] != self.EMPTY

        for row in range(5):
            for col in range(2):
                if seq(matrix[row][col:col + 4]):
```

```
                return True, tuple((row, c) for c
in range(col, col + 4))

        for col in range(5):
            for row in range(2):
                if seq([matrix[r][col] for r in
range(row, row + 4)]):
                    return True, tuple((r, col) for r in
range(row, row + 4))

        for row in range(2):
            for col in range(2):
                if seq([matrix[row + k][col + k] for
k in range(4)]):
                    return True, tuple((row + k, col +
k) for k in range(4))
                if seq([matrix[row + k][col + 3 - k]
for k in range(4)]):
                    return True, tuple((row + k, col +
3 - k) for k in range(4))

        if self.is_there_empty_spot() != True:
            return True, None
        else:
            return False, None

    @staticmethod
    def get_visualize(board: list[int]):
        assert len(board) ==
Board.BOARD_SIZE, "len() of board more or
less than Board.BOARD_SIZE"

        ret = ""
        figures_convert = {Board.CROSS:
Board.CROSS_STR, Board.NOUGHT:
Board.NOUGHT_STR, Board.EMPTY:
Board.EMPTY_STR}

        for i, figure in enumerate(board):
            ret += "[" + figures_convert[figure] +
"]"
            if (i + 1) % 5 == 0: ret += "\n"

        return ret

    def is_there_empty_spot(self):
        return self.EMPTY in self.__board

    def reset(self):
        self.__board = [self.EMPTY] *
self.BOARD_SIZE
        self.endgame = False

    def __call__(self, board: list[int]):
        assert len(board) ==
Board.BOARD_SIZE, "len() of board more or
less than Board.BOARD_SIZE"
        self.__board = board
        self.endgame = False

    def __str__(self):
        return self.get_visualize(self.__board)

    def __repr__(self):
        return f"Board(board={self.__board},
endgame={self.endgame})"

    def board_to_tensor_bits(self):
        cross = np.zeros((self.BOARD_SIZE),
dtype='float32')
        nought = np.zeros((self.BOARD_SIZE),
dtype='float32')

        for i, figure in enumerate(self.__board):
            cross[i] = self.CROSS_BIT if figure
== self.CROSS else self.EMPTY_BIT
            nought[i] = self.NOUGHT_BIT if
figure == self.NOUGHT else
self.EMPTY_BIT

        return np.concatenate((cross,
nought)).reshape((1, 2, 5, 5))

    def get_board(self):
        return copy.deepcopy(self.__board)

    def get_figure_winner(self):
        _, positions = self.get_winner()

        if positions == None:
            return self.EMPTY
        else:
            index = positions[0][0]*5 +
positions[0][1]
```

```
            return self.__board[index]

    def get_figures_count(self):
        return self.__board.count(1) +
self.__board.count(-1)
```

2.1.4 simulate.py

```
from tictactoe.board import Board
from tictactoe.reinforcement.rl_bot import
RLBotCross, RLBotNought
from tictactoe.supervised.sl_bot import SLBot
import sys
from tictactoe.stats import Stats, GameStat,
Graph, get_results_str

def sl_side(board: Board, figure: int, sl_bot:
SLBot):
    position = sl_bot.get_model_move(board,
figure)
    board.set_move(position, figure)

    return board.get_winner()[0]

def rl_side(board: Board, figure: int, rl_bot:
RLBotCross|RLBotNought):
    position = rl_bot.get_move(board)
    board.set_move(position, figure)

    return board.get_winner()[0]

def sl_x_vs_rl(sl_bot: SLBot, rl_bot_nought:
RLBotNought):
    board = Board()

    while True:
        if sl_side(board, Board.CROSS, sl_bot):
break
        if rl_side(board, Board.NOUGHT,
rl_bot_nought): break

    return board.get_figure_winner(),
board.get_figures_count()

def rl_x_vs_sl(rl_bot_cross: RLBotCross,
sl_bot: SLBot):
    board = Board()

    while True:
        if rl_side(board, Board.CROSS,
rl_bot_cross): break
        if sl_side(board, Board.NOUGHT,
sl_bot): break

    return board.get_figure_winner(),
board.get_figures_count()

if __name__ == "__main__":
    game_stat = GameStat()

    if len(sys.argv) == 2:
        game_stat.games_count =
int(sys.argv[1])

    sl_bot = SLBot()
    rl_bot_cross = RLBotCross()
    rl_bot_nought = RLBotNought()

    sl_stat = Stats("SL")
    rl_stat = Stats("RL")

    for i in range(game_stat.games_count):
        result, figures_count = sl_x_vs_rl(sl_bot,
rl_bot_nought)
        game_stat.figures_count_one +=
figures_count

        if result == Board.CROSS:
            sl_stat.x_wins += 1
            sl_stat.x_games_ids.append(i)
        elif result == Board.NOUGHT:
            rl_stat.o_wins += 1
            rl_stat.o_games_ids.append(i)

sl_stat.x_accumulated.append(sl_stat.x_wins)

rl_stat.o_accumulated.append(rl_stat.o_wins)

        result, figures_count =
rl_x_vs_sl(rl_bot_cross, sl_bot)
        game_stat.figures_count_two +=
figures_count

        if result == Board.CROSS:
            rl_stat.x_wins += 1
            rl_stat.x_games_ids.append(i)
        elif result == Board.NOUGHT:
            sl_stat.o_wins += 1
```

```
            sl_stat.o_games_ids.append(i)

rl_stat.x_accumulated.append(rl_stat.x_wins)

sl_stat.o_accumulated.append(sl_stat.o_wins)

    print(get_results_str(game_stat, sl_stat,
rl_stat))
    Graph(game_stat.games_count, sl_stat,
rl_stat).draw_graph()
```

2.1.5 stats.py

```
import matplotlib.pyplot as plt
from dataclasses import dataclass, field

@dataclass
class Stats:
    player_name: str
    x_wins: int = 0
    o_wins: int = 0
    x_games_ids: list[int] =
field(default_factory=list)
    o_games_ids: list[int] =
field(default_factory=list)
    x_accumulated: list[int] =
field(default_factory=list)
    o_accumulated: list[int] =
field(default_factory=list)

@dataclass
class GameStat:
    games_count: int = 100
    figures_count_one: int = 0
    figures_count_two: int = 0

def get_results_str(game_stat: GameStat,
stat_one: Stats, stat_two: Stats):
    draws_one = game_stat.games_count -
stat_one.x_wins - stat_two.o_wins
    draws_two = game_stat.games_count -
stat_one.o_wins - stat_two.x_wins

    get_names = lambda:
f"{stat_one.player_name} vs
{stat_two.player_name}"
    get_spec = lambda side: f"(Side: {side},
N={game_stat.games_count})"

    get_percent = lambda result: result /
game_stat.games_count * 100
    get_result = lambda status, result:
f"{status}={result}
({get_percent(result):.2f}%)"
    get_average = lambda figures_count: f"
Length={(figures_count /
game_stat.games_count):.2f}"
    ret = f"{get_names()} {get_spec('X')}
{get_result('W', stat_one.x_wins)},
{get_result('L', stat_two.o_wins)},
{get_result('D', draws_one)},
{get_average(game_stat.figures_count_one)}\
n"
    ret += f"{get_names()} {get_spec('0')}
{get_result('W', stat_one.o_wins)},
{get_result('L', stat_two.x_wins)},
{get_result('D', draws_two)},
{get_average(game_stat.figures_count_two)}
"

    return ret

class Graph:
    COLORS = ("blue", "red")
    LABELS_XY = ("Games count", "Wins")

    def __init__(self, games_count: int,
stat_one: Stats, stat_two: Stats):
        self.games_count = games_count
        self.x = range(games_count)

        self.stat_one = stat_one
        self.stat_two = stat_two

    def __create_plot(self, side: str, y_one:
list[int], y_two: list[int]):
        plt.title(f"{self.stat_one.player_name} vs
{self.stat_two.player_name} (Side: {side},
N={self.games_count})")
        plt.xlabel(self.LABELS_XY[0])
        plt.ylabel(self.LABELS_XY[1])
        plt.plot(self.x, y_one,
label=self.stat_one.player_name,
color=self.COLORS[0])
        plt.plot(self.x, y_two,
label=self.stat_two.player_name,
color=self.COLORS[1])
        plt.legend()
        plt.grid()

    def draw_graph(self):
        plt.figure(figsize=(10, 5))
```

```
        plt.subplot(1, 2, 1)
        self.__create_plot('X',
self.stat_one.x_accumulated,
self.stat_two.o_accumulated)

        plt.subplot(1, 2, 2)
        self.__create_plot('0',
self.stat_one.o_accumulated,
self.stat_two.x_accumulated)

        plt.tight_layout()
        plt.show()
```

2.2. app.py

```
from flask import Flask, render_template,
jsonify, request
from tictactoe.board import Board
from tictactoe.supervised.sl_bot import SLBot
from tictactoe.reinforcement.rl_bot import
RLBotCross, RLBotNought

app = Flask(__name__)

HOST = "127.0.0.1"
PORT = 8000

@app.route("/")
def index():
    return render_template("index.html")

def game_status(board: Board, is_end: bool,
win_positions: tuple[int], bot_position: int):
    response = {}

    response['board'] = board.get_board()
    response['bot_position'] = bot_position
    response['is_end'] = is_end
    response['win_positions'] = win_positions
    response['winner_figure'] =
board.get_figure_winner()

    return jsonify(response)

sl_bot = SLBot()
rl_bot_cross = RLBotCross()
rl_bot_nought = RLBotNought()

def get_bot_move(board: Board, bot_side: int,
bot_id: int):
    if bot_id == 0:
        return board.get_random_move(),
"easy_bot"
    elif bot_id == 1:
        return sl_bot.get_model_move(board,
bot_side), "sl_bot"
    else:
        return (rl_bot_cross.get_move(board) if
bot_side == Board.CROSS else
rl_bot_nought.get_move(board), "rl_bot")

@app.route("/bot_move",
methods=["POST"])
def bot_move():
    data = request.get_json()

    board = Board(data['board'])
    bot_side = data['bot_side']
    bot_id = data['bot_id']
    bot_position = None

    is_end, win_positions = board.get_winner()
    if is_end:
        return game_status(board, is_end,
win_positions, bot_position)

    bot_position, bot_name =
get_bot_move(board, bot_side, bot_id)
    board.set_move(bot_position, bot_side)

    is_end, win_positions = board.get_winner()

    return game_status(board, is_end,
win_positions, bot_position)

if __name__ == "__main__":
    app.run(host=HOST, port=PORT,
debug=True)
```

2.3 results.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


from tictactoe.board import Board
board = Board()
print(repr(board))


# In[ ]:


from tictactoe.board import Board
board = Board()
print(repr(board))
board.set_move(4, Board.CROSS)
board.random_move(Board.NOUGHT)
print(repr(board))


# In[ ]:


from tictactoe.board import Board
board = Board()
board.set_move(6, Board.CROSS)
board.random_move(Board.NOUGHT)
board.set_move(7, Board.CROSS)
board.random_move(Board.NOUGHT)
board.set_move(8, Board.CROSS)
print(board)


# In[ ]:


print(board)
print(repr(board.board_to_tensor_bits()))
print(f"Shape:
{board.board_to_tensor_bits().shape}")


# In[ ]:


get_ipython().run_line_magic('run', '-m
tictactoe.supervised.dataset 1000 500')


# In[ ]:


from tictactoe.board import Board

board = Board([0,-1,-1,0,0,1,1,-1,-1,0,-
1,1,1,0,-1,0,0,1,0,1,-1,-1,1,1,1])

print(board)
print(board.get_winner())
print(f"Figure winner:
{board.get_figure_winner()}")


# In[ ]:


get_ipython().run_line_magic('run', '-m
tictactoe.supervised.batch')


# In[ ]:


get_ipython().run_line_magic('run', '-m
tictactoe.supervised.validate')


# In[ ]:


from tictactoe.board import Board
from tictactoe.supervised.sl_bot import SLBot

board = Board()
slbot = SLBot()

board.set_move(6, Board.CROSS)
print(board)
```

```
model_move = slbot.get_model_move(board,
Board.NOUGHT)
print(f"SLBot move: {model_move}")
board.set_move(model_move,
Board.NOUGHT)
print(board)


# In[ ]:


from tictactoe.board import Board
from tictactoe.supervised.sl_bot import SLBot

board = Board()
slbot = SLBot()

model_move = slbot.get_model_move(board,
Board.CROSS)
print(f"SLBot move: {model_move}")
board.set_move(model_move,
Board.CROSS)
print(board)

board.set_move(14, Board.NOUGHT)
print(board)


# In[ ]:


from tictactoe.board import Board
from tictactoe.reinforcement.rl_bot import
RLBotNought

board = Board()
rlbot_nought = RLBotNought()

board.set_move(0, Board.CROSS)
print(board)

model_move = rlbot_nought.get_move(board)
print(f"RLBotNought move:
{model_move}")
board.set_move(model_move,
Board.NOUGHT)
print(board)


# In[ ]:


from tictactoe.board import Board
from tictactoe.reinforcement.rl_bot import
RLBotCross

board = Board()
rlbot_cross = RLBotCross()

model_move = rlbot_cross.get_move(board)
print(f"RLBotCross move: {model_move}")
board.set_move(model_move,
Board.CROSS)
print(board)

board.set_move(4, Board.NOUGHT)
print(board)


# In[ ]:


from tictactoe.supervised.sl_model import
SLModel, get_parameters_count
from tictactoe.reinforcement.rl_model import
RLModel, get_parameters_count

sl_model = SLModel()
rl_model = RLModel()

print(f"{sl_model}\nКількість параметрів:
{get_parameters_count(sl_model)}\n")
print("-" * 75, '\n')
print(f"{rl_model}\nКількість параметрів:
{get_parameters_count(rl_model)}
```

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ
Проректор Українського
державного університету
науки і технологій
Анатолій РАДКЕВИЧ
08.01.25

«ДОСЛІДЖЕННЯ МЕТОДІВ НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ ПРИ МОДЕЛЮВАННІ ПОВЕДІНКИ ІГРОВИХ АГЕНТІВ»

Керівництво користувача
ЛИСТ ЗАТВЕРДЖЕННЯ
44165850.01428-01 І3 01-ЛЗ

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
08.01.25

Керівник розробки
_____Вадим ГОРЯЧКІН
08.01.25

Виконавець
_____Кирило ЯРОВИЙ
08.01.25

Нормоконтролер
_____Світлана ВОЛКОВА
08.01.25

2025

ДОДАТОК В

Керівництво користувача

ЗАТВЕРДЖЕНО
44165850.01428-01 І3 01-ЛЗ

«ДОСЛІДЖЕННЯ МЕТОДІВ НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ ПРИ МОДЕЛЮВАННІ ПОВЕДІНКИ ІГРОВИХ АГЕНТІВ»

Керівництво користувача
44165850.01428-01 І3-01
Листів 9

2025

з 1

АНОТАЦІЯ

Документ 44165850.01428-01 І3-01 «Дослідження методів навчання нейронних мереж при моделюванні поведінки ігрових агентів» входить до складу програмної документації.

У документі представлено керівництво користувача. Програмне забезпечення розроблено на мові Python та JavaScript за допомогою таких бібліотек як: NumPy, PyTorch, Matplotlib, Jupyter Notebook та Flask.

ЗМІСТ

1. ВВЕДЕННЯ

Програмний продукт призначений для реалізації теми «Дослідження методів навчання нейронних мереж при моделюванні поведінки ігрових агентів».

Метою розробки є надання зручного функціоналу для тестування та проведення досліджень при моделюванні поведінки ігрових агентів.

Створений програмний продукт призначений як для науковців та і для звичайних користувачів. Має зручний клієнт-серверний інтерфейс, який може відображати зміни у структурі моделі, які відповідають за ігрових ботів.

2. ПІДГОТОВКА ДО ЗАСТОСУВАННЯ

Для коректної роботи програми потрібно встановити наступне програмне забезпечення та бібліотеки:

1. Завантажити та розпакувати проект;
2. Завантажити та встановити Python версії 3.12 або нижче (Завантажити можна з офіційного сайту);
3. Завантажити додаткові бібліотеки: NumPy, PyTorch, Matplotlib та Flask (Це можна зробити за допомогою системи керування пакунками pip);
4. Завантаження моделей:
   4.1. Моделі вже завантаженні:
      4.1.1. Перевірити наявність моделей у папці “tictactoe/supervised/data” та “tictactoe/reinforcement/data”. Моделі мають розширення .pth;
   4.2. Створення моделей:
      4.2.1. Supervised модель:
         4.2.1.1. Запустити модуль “tictactoe/supervised/dataset” для створення датасету;
         4.2.1.2. Запустити модуль “tictactoe/supervised/batch” для створення батчів;
         4.2.1.3. Запустити модуль “tictactoe/supervised/sl_model” після чого модель почне своє навчання.
      4.2.2. Reinforcement модель:
         4.2.2.1. Запустити модуль “tictactoe/reinforcement/rl_model” після чого модель почне своє навчання.
5. Запустити модуль “app”;
6. Після успішного запуску серверу (Може займати від 5 до 45 секунд) потрібно перейти за посиланням “localhost:8000” або “127.0.0.1:8000”.

7 ОПИС ОПЕРАЦІЙ

Після запуску локального сервера та перехід на веб-сторінку можна побачити користувацький інтерфейс, який надає наступні можливості:

- «Грати за X» – кнопка, яка надає можливість почати гру за хрестик;
- «Грати за 0» – кнопка, яка надає можливість почати гру за нулик;
- «Легкий бот» – чекбокс, який відповідає за бота, який надає свої ходи. Якщо чекбокс активний, то відповіді надає легкий бот;
- «SL модель» – чекбокс, який відповідає за бота, який надає свої ходи. Якщо чекбокс активний, то відповіді надає Supervised Learning модель;
- «RL модель» – чекбокс, який відповідає за бота, який надає свої ходи. Якщо чекбокс активний, то відповіді надає Reinforcement Learning модель;
- «Canvas» – гральний інтерфейс, який надає можливість користувачу ставити фігури на дошку.

8 АВАРІЙНІ СИТУАЦІЇ

Аварійні ситуації під час використання програмного забезпечення можуть бути спричиненні наступними причинами:

- Запуск програми не у форматі “python -m”, тобто у форматі модуля спричинить помилку;
- Відсутність однієї з бібліотек спричинить помилку;
- Відсутність папки “data/” спричинить помилку на різних етапах;
- Відсутність датасету для SL моделі спричинить помилку при її навчанні;
- Відсутність однієї з моделей спричинить помилку при запуску локального серверу.

9. РЕКОМЕНДАЦІЇ ЩОДО ЗАСТОСУВАННЯ

Для того, щоб програмне забезпечення працювало коректно на будь-яких етапах потрібно слідувати наступним порадам:

- Використовувати відеокарту від компанії NVIDIA з наявністю CUDA. Це надасть можливість виконувати тренування нейронних мереж набагато швидше;
- Використовувати потужний CPU, адже після навчання нейронних мереж моделі роблять прогнозування за допомогою центрального процесора. Це пов’язано з тим, що потреба у паралельному обчисленні зникає і CPU у такому випадку є кращим вибором.

ДОДАТОК Г

Міністерство освіти і науки України

Український державний університет науки і технологій

ТЕЗИ
XVIII Міжнародної науково-практичної конференції
«СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ І ОСВІТІ»
Присвячено пам'яті Владислава СКАЛОЗУБА

ABSTRACTS
of the XVIII International Conference
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»
Dedicated to the memory of Vladislav SKALOZUB

12.12.2024 – 13.12.2024

Дніпро

2024

КОМП’ЮТЕРНІ НАУКИ ТА ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

ПІДСЕКЦІЯ «ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА СИСТЕМИ»

Дослідження методів навчання нейронних мереж при моделюванні поведінки ігрових агентів

Яровий К.В., керівник доц. Горячкін В.М.
Український державний університет науки і технологій

В сучасному світі нейромережі отримали неабияку популярність через свою можливість використання у багатьох галузях, високу ефективність роботи з великими обсягами даних, де вони можуть знаходити закономірності, що робить їх дуже корисними для вирішення задач різних типів. Нейромережі використовуються для розпізнавання зображень, природної обробки мови, рекомендаційних систем, медичної діагностики та багатьох інших завдань. Потужні графічні процесори значно прискорили процес навчання нейромереж, зробивши їх доступними для широкого кола дослідників та компаній.

Згорткові нейронні мережі (CNN) є класом глибоких нейронних мереж, які використовуються для аналізу візуальних зображень. CNN стали потужним інструментом для розпізнавання об’єктів, облич, мови та інших завдань, які пов’язані з обробкою ображень. Також вони можуть бути дуже корисними для використання у настільних іграх для розпізнавання образів, аналізі ігрових ситуацій та створення інтелектуальних суперників.

Для ефективного тренування згорткових нейронних мереж було обрано два підходи: Supervised Learning та Reinforcement Learning. Supervised Learning - це підхід де потрібно підготувати великий набор даних, який повинен бути розподілений на тренувальний, валідаційний та тестовий набори. А також, встановити відповідні гіперпараметри, такі як розмір фільтрів, кількість шарів, розмір партії даних та швидкість навчання. Мета такого підходу полягає в тому, щоб модель навчилась передбачати правильні мітки для нових, невідомих даних,

використовуючи помилки між передбаченням та реальними мітками для коригування своїх параметрів.

Використання Reinforcement Learning є більш інтерактивним процесом. При такому підході модель навчається шляхом проб та помилок, отримуючи винагороди або покарання за свої дії. Мета моделі – максимізувати сумарну нагороду в довгостроковій перспективі. На відміну від навчання з учителем, де модель має доступ до правильних відповідей на кожному кроці, у Reinforcement Learning модель сама повинна досліджувати середовище та знаходити оптимальні дії. Це робить Reinforcement Learning більш складним, але й більш гнучким методом навчання.

В процесі дослідження створені моделі ефективного ігрового агенту для настільної гри «Хрестики-нулики» з розміром дошки п’ять у рядок та п’ять у стовпчик клітинок та переможною комбінацією у чотири символи в стовпчик, рядок або діагональ, який з методів навчає модель такого ігрового агенту швидше та з меншим використанням ресурсів. Результат роботи може бути використаний для вирішення задач, які потребують рішень за чітко встановленими правилами, а також, розширить розуміння та знання про можливості та обмеження сучасних методів машинного навчання.