

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Створення інтелектуального агента для гри в шахи
з використанням синтезованих даних»**

Виконав студент групи ПЗм-22-1
УКРАЇНЕЦЬ Микола Олександрович

Керівник роботи:
ЄФРЕМОВ Юрій Миколайович

Рецензент:
ЧИЖМОТРЯ Олексій Володимирович

Житомир – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«05» лютого 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **УКРАЇНЕЦЬ Микола Олександрович**

Керівник роботи: **ЄФРЕМОВ Юрій Миколайович**

Тема роботи: **«Створення інтелектуального агента для гри в шахи з використанням синтезованих даних»,**

затверджена Наказом закладу вищої освіти від **«05» лютого 2024 р. № 48**

Вихідні дані для роботи: **Предметом дослідження є інтелектуальні шахові агенти на основі методів машинного навчання. Об'єктом дослідження є методи та алгоритми машинного навчання для створення інтелектуального агента для гри в шахи з використанням синтезованих навчальних даних.**

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Єфремов Ю. М.	05.02.2024р.	05.02.2023р.
2	Єфремов Ю. М.	05.02.2024р.	05.02.2024р.
3	Єфремов Ю. М.	05.02.2024р.	05.02.2024р.

РЕФЕРАТ

Кваліфікаційна робота присвячена створенню інтелектуального шахового агенту на основі синтезованих даних з використанням глибинного машинного навчання.

Кваліфікаційна робота складається з програмного комплексу, який включає в себе два шахові рушії, а також з пояснювальної записки. Пояснювальна записка складається з вступу, трьох розділів, висновків та списку використаної літератури, що містить 26 найменувань. Також в роботі наведено 29 рисунків.

Кваліфікаційна робота складається з трьох розділів. В першому розділі було визначено основні вимоги, проведено аналіз проблематики, а також досліджено методи та алгоритми для її вирішення. Також було проведено аналіз вимог, а також здійснено постановку завдання.

В другому розділі було проведено синтез даних та здійснено практичну реалізацію алгоритму.

В третьому розділі було розглянуто отримані шахові рушії, проведено їхнє тестування та визначено кращого з них.

Висновок містить в собі підсумки проведеного дослідження.

Ключові слова: ШАХИ, МАШИННЕ НАВЧАННЯ, МОДЕЛЬ, СИНТЕЗ ДАНИХ, ШАХОВИЙ РУШІЙ, НЕЙРОННІ МЕРЕЖІ.

ABSTRACT

The qualification work is devoted to the creation of an intelligent chess agent based on synthesized data using deep machine learning.

The qualification work consists of a software package that includes two chess engines, as well as an explanatory note. The explanatory note consists of an introduction, three chapters, conclusions, and a list of references containing 26 items. The paper also contains 29 figures.

The qualification work consists of three chapters. In the first chapter, the main requirements were identified, the problem was analyzed, and methods and algorithms for its solution were investigated. Also, the requirements were analyzed, and the task was set.

In the second section, we synthesized the data and implemented the algorithm in practice.

In the third section, the obtained chess engines were analyzed, tested, and the best one was determined.

The conclusion contains the results of the study.

Keywords: CHESS, MACHINE LEARNING, MODEL, DATA SYNTHESIS, CHESS ENGINE, NEURAL NETWORKS.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	8
ВСТУП.....	9
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ.....	11
1.1 Аналіз задачі, засобів та методів її вирішення	11
1.2 Методи створення інтелектуальних агентів для гри в шахи	15
1.3 Визначення підходу для створення набору даних	22
1.4 Аналіз вимог та постановка завдання	25
РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ АЛГОРИТМІВ НАВЧАННЯ НЕЙРОННИХ МОДЕЛЕЙ.....	29
2.1 Синтез даних для навчання	29
2.2 Кодування навчальних даних.....	36
2.3 Тренування моделей і практична реалізація.....	38
2.4 Результати проведеного навчання	44
РОЗДІЛ 3. АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ.....	49
3.1 Реалізація UCI-сумісних рушіїв	49
3.2 Робота шахових рушіїв в графічному шаховому інтерфейсі.....	52
ВИСНОВКИ	60
ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....	61

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ШІ – штучний інтелект

ГН – глибинне навчання

ГП – генетичне програмування

НП – навчання з підкріпленням

MDP – Markov's Decision Process

ВСТУП

В сучасному інформаційному суспільстві, де технології швидко розвиваються, методи машинного навчання стають важливим і ефективним інструментом для вирішення різноманітних завдань.

Сьогодні методи машинного навчання широко та успішно використовуються в різних сферах, забезпечуючи значний прогрес у вирішенні складних завдань та оптимізації процесів.

Завдяки можливостям, які надає машинне навчання та штучний інтелект, людство полегшує життя у багатьох сферах. Ці технології допомагають вирішувати складні завдання за невелику кількість часу. Для покращення моделей штучного інтелекту і визначення їхньої потужності часто використовують різні складні задачі. Однією з таких задач є гра в шахи. Шахи – це стародавня настільна інтелектуальна гра, яка походить з Індії. Шахи визначаються великою стратегічною глибиною та великою кількістю можливих комбінацій. Завдяки цьому, гра привертає увагу як новачків, так і досвідчених гравців.

Актуальність теми полягає в тому, що розробка моделей, які можуть ефективно грати в шахи дають великий поштовх в розвитку штучного інтелекту. Це стимулює дослідників шукати нові підходи та методи в розв'язанні такої нетривіальної задачі, які в майбутньому можуть бути адаптовані і використані в інших сферах діяльності. Розвиток шахових алгоритмів машинного навчання дозволяє вивчати та оптимізувати стратегії гри в умовах обмежених ресурсів, що може бути важливо для подальшого вдосконалення алгоритмів в інших галузях ІІІ.

Подальшого дослідження потребує тренування нейронних моделей на основі синтетично згенерованого набору даних.

Наукова новизна роботи полягає в тому, що було запропоновано методи синтезу навчальних даних з використанням існуючих шахових рушіїв для розробки інтелектуального шахового агента на основі нейронних мереж. Суть першого методу полягає в використанні такого підходу як самотійна гра, коли рушій грає сам з собою, для отримання шахових партій, які в подальшому будуть

використанні як навчальні дані. Другий метод полягає в застосуванні двох шахових рушіїв для проведення партій між ними та формуванні навчальних даних на основі цих партій. В результаті виконання роботи було описано кожен з методів, а також визначено той, який дає кращі результати для розробки інтелектуального шахового агента.

Метою цієї роботи є ретельний аналіз моделей та методів створення інтелектуальних шахових агентів, а також розробка методів синтезу даних для ефективного тренування цих моделей. Для досягнення цієї мети слід виконати наступні основні завдання:

- визначити основні вимоги та бажані кінцеві результати;
- провести аналіз існуючих методів до створення інтелектуальних шахових агентів;
- проаналізувати методи машинного навчання, які застосовуються для вирішення задачі гри в шахи;
- вибрати підходящі підходи для реалізації методу машинного навчання для гри в шахи;
- створити синтетичний набір даних для тренування;
- зробити обґрунтований вибір інструментарію для імплементації моделі;
- провести валідацію та тестування отриманих результатів.

Предметом дослідження є інтелектуальні шахові агенти на основі методів машинного навчання.

Об'єктом дослідження є методи та алгоритми машинного навчання для створення інтелектуального агента для гри в шахи з використанням синтезованих навчальних даних.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1 Аналіз задачі, засобів та методів її вирішення

На сьогоднішній день методи машинного навчання активно та ефективно використовуються у різних галузях, приносячи значний прогрес у розв'язанні складних завдань та оптимізації різноманітних процесів.

В економіці та фінансах, алгоритми машинного навчання застосовуються для прогнозування ринкових тенденцій, оптимізації інвестиційних портфелів та для керування ризиками. У сфері транспорту та логістики, системи автопілотів та маршрутизації оптимізують рух транспорту, забезпечуючи ефективне використання ресурсів. Навчання з підкріпленням використовується в робототехніці для навчання роботів виконувати різноманітні завдання, а в області природних мовних процесорів – для розпізнавання та генерації текстів. Ці лише кілька прикладів вказують на важливість та широкий спектр застосування машинного навчання в різних галузях, що вказує на великі можливості для подальших досягнень та інновацій.

Шахи – це інтелектуальна гра на спеціальній дошці (шахівниці), яку поділено на 64 світлі та темні клітини (їх ще називають полями). Кожна сторона («білі» і «чорні») має комплект з 16 фігур і пішаків. Гра здійснюється за встановленими правилами. Ця гра походить від стародавньої індійської гри чатуранга, яка, крім того, є ймовірним предком східних стратегічних ігор сянци, чангі і сьогі. Шахи поєднують в собі елементи мистецтва, науки і спорту. Гра в шахи вимагає стратегічного мислення, передбачення ходів супротивника та вміння керувати позицією на дошці. Шахи використовуються не лише як розвага, але і як інтелектуальний спорт, де гравці можуть конкурувати на високому рівні.

Структура та характер гри в шахи пов'язані із кількома галузями математики. Протягом сотень років вивчалися численні аспекти комбінаторики та топології, пов'язані з грою в шахи.

Було зроблено оцінки, що кількість легальних позицій у шахах становить приблизно 10^{43} . Складність дерева гри оцінюється приблизно в 10^{123} . Американський математик Клод Шеннон першим розрахував цю складність для шахів і встановив, що вона дорівнює 10^{120} , це число відоме як число Шеннона. Зазвичай, позиція має від 30 до 40 можливих ходів, проте це число може дорівнювати нулю (у випадку мату або пату) або сягати до 218 у певних позиціях.

Одним з найважливіших математичних викликів пов'язаний з грою в шахи є створення алгоритмів, які можуть грати в шахи. Перші спроби зробити шахову машину датуються ще 18 століттям. Близько 1769 року з'явився автомат під назвою «Механічний турок», який став доволі відомим, перш було розкрито, що це містифікація.

Однак все кардинально змінилось з появою цифрових комп'ютерів в 1950-х роках. В 1950 році той же Клод Шеннон опублікував новаторську статтю, яка мала назву «Програмування комп'ютера для гри в шахи». В ній він піднімає проблеми створення обчислювальної процедури чи "програми" для сучасного універсального комп'ютера, яка дозволить йому грати у шахи. Зокрема він описав чому комп'ютери дуже добре підходять для реалізації шахової машини: «Шахова машина є ідеальним об'єктом для початку, оскільки: проблема чітко визначена як у дозволених операціях (ходах), так і в кінцевій меті (мат); вона не настільки проста, щоб бути тривіальною, але й не занадто складна для задовільного розв'язання; шахи загалом вважаються грою, що вимагає «мислення» для вправної гри; розв'язання цієї проблеми змусить нас визнати можливість механізованого мислення або додатково обмежити наше уявлення про «мислення»; дискретна структура шахів гармонійно вписується в цифровий характер сучасних комп'ютерів.» [1]

Його твердження виявилось коректним і комп'ютери досягли великої майстерності в цій непростій грі. З часом можливості комп'ютерних шахових рушіїв посилювалися і згодом вони досягли такого рівня, що немає сенсу проводити змагання людей та рушіїв, більш актуальними є змагання, коли рушії змагаються з іншими рушіями.

Процес гри в шахи полягає в тому, що гравці по черзі переміщують одну фігуру (за винятком спеціального ходу рокіровки, коли тура і король змінюють свої позиції) на певну, дозволена правилами клітинку, аби досягти мату – тобто оголосивши шах, від якого король суперника ніяким чином не може захиститися. Тому в процесі роботи, шаховий рушій має обирати оптимальний хід. Загалом можна виділити два основних підходи, які використовуються для розв’язання задачі вибору ходу в комп’ютерних шахах:

- Заснований на пошуку (мінімакс/альфа-бета, дерево пошуку методом Монте-Карло);
- Заснований на оцінюванні (машинне навчання, нейронні мережі, генетичні алгоритми, градієнтний спуск, навчання з підкріпленням).

Клод Шеннон у своїй статті запропонував розглядати шахову партію як дерево – цифрову структуру даних виборів (гілок), що відповідає ходам. Вузли дерева представляли собою позиції на дошці, що виникали в результаті виборів ходу. Неможливість відобразити всю шахову партію шляхом побудови дерева від першого ходу до останнього є очевидною: у шахах є в середньому 36 ходів на одну позицію, а середня партія триває приблизно 35 ходів. Є 400 можливих позицій після першого ходу кожного гравця, приблизно 200 000 після двох ходів кожного та майже 120 мільйонів лише після 3 ходів кожного. Велика кількість ймовірних легальних позицій вказує на те, що підхід з використанням пошуку не можна реалізувати для всієї глибини дерева, адже обрахунок такої кількості комбінацій займе дуже великий час. Тому шахові рушії, які базуються на пошуку, проводять цей самий пошук оптимального ходу з обмеженою глибиною обходу дерева. вони перевіряють усі ходи, потім усі контрходи до цих ходів, потім усі ходи, що протидіють їм, і так далі. Це виконання триває до досягнення певної максимальної глибини пошуку або програма визначить, що було досягнуто остаточної позиції. Недоліком цього підходу є те, що оптимальний хід отриманий після пошуку може бути оптимальним в цій позиції, проте не таким оптимальним з точки зору з стратегічної точки зору в довгостроковій перспективі.

Нідерландський майстер шахів та психолог Адріан де Гроот інтерв'ював кількох шахістів різних рівнів майстерності і прийшов до висновку, що як майстри, так і початківці розглядають приблизно сорок-п'ятдесят позицій перед ходом. Те, що робить майстрів кращими гравцями, полягає в тому, що вони використовують навички розпізнавання шаблонів, набуті з досвіду. Це дозволяє їм детальніше розглядати деякі лінії, просто не враховуючи ходи, які вони вважають слабкими. Ще одним доказом цього є те, що хороші гравці знаходять набагато легше відтворювати позиції з реальних шахових партій, розбиваючи їх на невелику кількість впізнаваних підпозицій, ніж для повністю випадкових розташувань тих самих фігур. [2]

Тобто можна зробити висновок, що не обов'язково перебирати всі можливі комбінації ходів для визначення найкращого. В експерименті де Гроота і сильніші, і слабші гравці перебирали приблизно однакову кількість ходів, однак їхня якість відрізнялася. Завдяки досвіду сильніші гравці вибирали більш якісні ходи.

На противагу пошуковим алгоритмам, існують алгоритми засновані на оцінюванні. Для більшості шахових позицій комп'ютери не можуть передбачити всі можливі кінцеві позиції. Замість цього вони дивляться вперед на кілька кроків і порівнювати можливі позиції. Алгоритм, який оцінює позицію, називається «функцією оцінки», і ці алгоритми часто сильно відрізняються між різними шаховими програмами.

Загалом функції оцінювання беруть до уваги цінність матеріалу (фігур), які знаходяться на дошці на рівні з іншими факторами, які визначають перевагу тієї чи іншої сторони. Під час підрахунку матеріалу для кожної сторони типові значення фігур становлять 1 очко для пішака, 3 очка для коня чи слона, 5 очок для тури та 9 очок для ферзя. Королю іноді дають довільно високе значення, наприклад 200 очок, щоб гарантувати, що мат переважає всі інші фактори. Окрім очок за фігури, більшість функцій оцінки враховують багато факторів, таких як структура пішаків, той факт, що пара слонів зазвичай коштує більше, централізовані фігури коштують більше тощо.

Більшість функцій оцінювання використовують в своїй основі нейронні мережі. Це дозволяє тренувати їх таким чином, що вони оцінюватимуть ситуації на шахівниці не лише з точки зору оптимальності, але і враховуватимуть паттерни з тих ігор, на яких вони навчалися, тобто виділяти стратегії, а не лише робити виважені ходи. Для створення нейронних мереж використовують набори даних, які дають мережі уявлення про те, що їй слід робити. Чим якісніші і об'ємніші навчальні дані, тим потужнішою і кращою виходить кінцева модель. Враховуючи це, подальше дослідження націлене на тренування моделей для гри в шахи використовуючи синтезовані набори даних.

1.2 Методи створення інтелектуальних агентів для гри в шахи

Наразі велика увага приділяється розробці та вдосконаленню методів машинного навчання. Існує декілька основних підходів, які використовуються для створення інтелектуальних шахових рушіїв. Вони включають застосування глибокого навчання та нейронних мереж для моделювання складних залежностей в шахових позиціях. Системи самонавчання, такі як ті, що базуються на методах навчання з підкріпленням, і ті, які використовують самостійну гру, також показують високу ефективність в шахах.

Загалом пропонується проаналізувати наступні методи, які застосовуються для розробки інтелектуальних шахових агентів:

- Глибинне навчання (Deep Learning);
- Навчання з підкріпленням (Reinforcement Learning);
- Методи генетичного програмування (Genetic programming)

Розглянемо кожен з цих підходів більш детально на основі досліджень інших науковців.

Глибинне навчання — це техніка машинного навчання, яка вчить комп'ютери робити те, що є природним для людини: вчитися на прикладі. В глибинному навчанні комп'ютерна модель вчиться виконувати класифікаційні за-

вдання на пряму з зображень, тексту або звуку. Моделі глибинного навчання можуть бути дуже високоточними і навіть перевершувати ефективність людини в виконанні таких завдань. Моделі натреновані з використанням великих наборів даних і архітектури нейронних мереж з багатьма шарами.

Більшість глибинних методів навчання використовують архітектуру нейронних мереж, які часто називають глибинними нейронними мережами. На відміну від традиційних нейронних мереж, які зазвичай містять 2-3 прихованих шари, глибинні нейронні мережі можуть містити до 150 прихованих шарів. [3]

У одному із своїх досліджень Thomas Petrick, Касу Lombard та інші дослідники використали методи класичного глибинного навчання для створення інтелектуального агента. Їхньою метою було створити шаховий інтелект, який буде робити «найкращий» хід, враховуючи поточний стан дошки. Очікується, що оцінювальна функція буде працювати з точністю понад 70%, а загальний шаховий інтелект зможе грати на проміжному рівні приблизно 1200 ELO. [4]

Рейтингова система Ело - це метод обчислення відносного рівня кваліфікації гравців у іграх з нульовою сумою, таких як шахи. Система рейтингу ELO для гравців представлена числом, яке може змінюватися в залежності від результатів рейтингових ігор. Після кожного поєдинку переможець отримує рейтингові очки від програвшого. Важливо зауважити, що рейтинги ELO є порівняльними та визнаються лише в рамках конкретного рейтингового пулу, в якому вони були розраховані, і не є абсолютною мірою силового рівня гравця. [5]

У своєму дослідженні Thomas Petrick з колегами тренували нейронну мережу за допомогою навчання без нагляду та під наглядом для оцінки сили позиції. Для цього вони використовували файли у форматі PGN, які призначені для представлення шахових партій. Розбираючи ці ігри певним чином, було створено набір даних, який складався з 200 000 випадкових позицій з понад 20 000 ігор гротмейстерів з сайту pgnmentor.com. У висновку, їм вдалося отримати шаховий інтелект, який грав на рівні 600 ELO. Варто відзначити, що команда зіткнулася з певними апаратними обмеженнями під час тренування моделі, адже потужності

їхніх персональних комп'ютерів було недостатньо аби проводити складні розрахунки, тому вони орендували GPU для збільшення обчислювальної здатності. Ця проблема виникла під час виконання скрипту для виділення необхідних даних для тренування. Після цього вони також збільшили кількість тренувальних даних та кількість епох для тренування. Загалом, їм вдалося досягти половину від бажаного рейтингу ELO і отримати доволі гарні результати в плані отриманої моделі.

Варто відмітити, що метод з використанням глибинних нейронних мереж вимагає великої кількості якісних, а також великої обчислювальної потужності. Це призвело до потреби в оренді GPU для виконання програми, що збільшило витрати та затримало роботу. Також навіть після другого етапу тренування, точність моделі не досягла високого рівня, і шаховий інтелект поки не є достатньо сильним для перемоги над членами команди. Є потреба в подальших удосконаленнях і оптимізаціях.

Ще одним методом розробки інтелектуальних агентів є навчання з підкріпленням. У методології навчання з підкріпленням розробники використовують систему нагород і покарань для програмування агенту. Цей підхід передбачає присвоєння позитивних значень бажаним діям з метою стимулювання їх використання та негативних значень для уникнення небажаних дій. Важливо визначити довгострокові цілі, які допомагають агентові уникнути затримок на менш важливих завданнях. За допомогою цього підходу агент навчається максимізувати довгострокові нагороди та уникати покарань, приводячи до оптимальних рішень. Цей метод широко використовується в штучному інтелекті для направлення навчання машин за допомогою нагород і покарань. [6]. На рисунку 1.1 схематично зображено цикл навчання з підкріпленням.

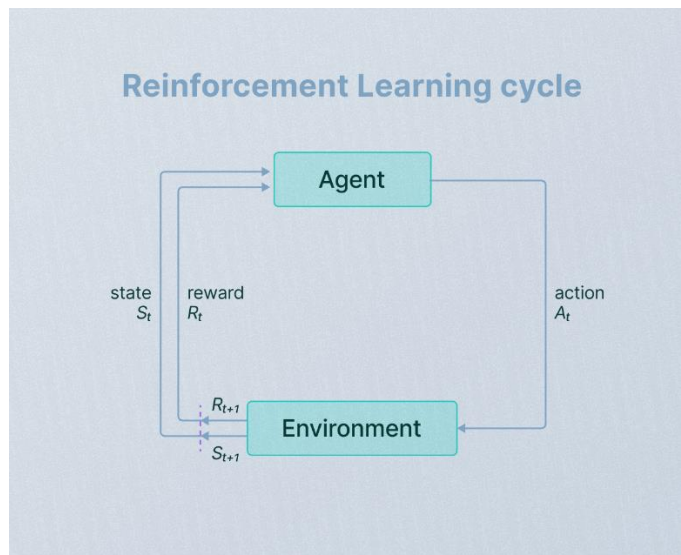


Рис. 1.1. Цикл навчання з підкріпленням

Навчання з підкріпленням має в основі Markov Decision Process. Markov Decision Process (MDP) є математичною моделлю в теорії прийняття рішень, яка використовується для формалізації ситуацій з вибором та невизначеністю, де приймаються рішення в умовах випадковості. MDP широко використовується в різних областях, включаючи теорію керування, штучний інтелект, оптимізацію та робототехніку. Основні компоненти MDP:

- Стани (States): Множина можливих ситуацій, в яких може опинитися система або агент. Стани представляють можливі конфігурації середовища;
- Дії (Actions): Множина можливих дій чи рішень, які агент може приймати в кожному стані. Дії впливають на перехід системи з одного стану в інший;
- Функція переходу (Transition Probability Function): Ця функція визначає ймовірність переходу системи з одного стану в інший після виконання певної дії;
- Функція нагороди (Reward Function): Для кожної пари стан-дія визначається нагорода, яку отримує агент. Мета полягає в тому, щоб максимізувати суму нагород протягом часу;
- Функція політики (Policy): Політика визначає стратегію агента, яка вказує, які дії слід вибирати в кожному стані.

Основним принципом MDP є властивість Маркова, що означає, що ймовірність переходу в новий стан залежить тільки від поточного стану та дії, а не від попередньої історії станів та дій.

Мета агента в MDP полягає в тому, щоб знайти оптимальну стратегію (політику), яка максимізує очікувану суму нагород протягом часу. Для цього можна використовувати алгоритми, такі як методи динамічного програмування, метод Монте-Карло та методи навчання з підсиленням. MDP є важливим інструментом в розв'язанні задач прийняття рішень в умовах невизначеності та випадковості. [6]

У своєму дослідженні Marco Block, Maro Bader, Ernesto Tapia та інші працювали над використанням Reinforcement Learning для створення шахових агентів. Для цього вони видозмінювали метод підкріпленого навчання під назвою Temporal Differences і комбінували його з іншими алгоритмами навчання.

У процесі дослідження автори визначили ряд ключових висновків стосовно отриманого шахового рушія. Під час тестування на людських гравцях у 50 іграх з різними часовими налаштуваннями, рушій показав силу на рівні близько 1800 ± 50 , але після 119 ігор його продуктивність зросла до 2016. Важливим було використання векторів для різних типів позицій, що дозволило агенту адаптуватися до оцінювання позицій та їх впливу на гру. Проте, були виявлені проблеми з деякими класами позицій, і не всі з них були вивчені належним чином. Оптимізація вручну встановлених значень та зміни в протоколі UCI виявилися корисними для покращення якості гри. Автори вбачають можливості подальших вдосконалень, зокрема, у виборі типів позицій та необхідності більшого обсягу тренувальних ігор для досягнення високої продуктивності. [8]

Ще варто відзначити розробку компанії DeepMind (дочірня компанія Google, яка спеціалізується на ШІ) під назвою AlphaZero. Традиційні шахові рушії, включаючи чемпіона світу серед комп'ютерних шахів Stockfish і революційний Deep Blue від IBM, ґрунтуються на тисячах правил і евристик сильних гравців, які намагаються врахувати кожен можливий варіант в грі.

AlphaZero використовує абсолютно інший підхід, замінюючи ці вручну створені правила глибокою нейронною мережею і універсальними алгоритмами, які нічого не знають про гру, крім основних правил.

Особливістю AlphaZero є те, що для того аби навчитися грати в певну гру, ненатренована нейронна мережа грає мільйони ігор сама з собою роблячи різні спроби і помилки. В основі цього підходу лежить навчання з підкріпленням. Спочатку вона грає абсолютно випадково, але з часом система вивчається з перемог, поразок та нічиїх, щоб налаштовувати параметри нейронної мережі, зробивши ймовірнішим вибір вигідних ходів у майбутньому. Кількість тренувань, які потрібні мережі, залежить від стилю та складності гри, займаючи приблизно 9 годин для шахів, 12 годин для шогі та 13 днів для го.

Так, час необхідний на тренування досить великий, але його результат дійсно вражаючий, так як AlphaZero може грати на рівні приблизно 4000 ELO. [9]

Також варто відмітити шаховий рушій Stockfish (з ним порівнюють рушій AlphaZero), який використовує підхід, заснований на пошуку. Рушій Stockfish складається з трьох частин: представлення дошки, евристичний пошук по дереву та оцінка дошки. Представлення дошки полягає в ефективному кодуванні стану шахової дошки, щоб її можна було ефективно зберігати і шукати по ній. Евристичний пошук по дереву наближений до пошуку по мінімаксному дереву, який, в свою чергу, був би занадто повільним у виконанні. Оцінка дошки використовує представлення дошки і дає їй оцінку того, наскільки "хороша" дошка (тобто оцінює шанси на перемогу)

У грудні 2017 року Stockfish 8 використовували як бенчмарк для тестування AlphaZero підрозділу Google DeepMind, причому кожен рушій підтримувався різним апаратним забезпеченням. AlphaZero тренувався шляхом самостійної гри протягом дев'яти годин, і досяг рівня Stockfish всього за чотири. У 100 партіях зі звичайної стартової позиції AlphaZero виграв 25 партій білими, виграв 3 партії чорними, а решту 72 зіграв внічию, зазнавши 0 поразок. AlphaZero також зіграв дванадцять 100-партійних матчів проти Stockfish, починаючи з дванадцяти популярних дебютів, з остаточним рахунком 290 перемог, 886 нічиїх і 24 поразки, з очковим рахунком 733:467. [9]

Ще одним підходом для створення шахових рушіїв є генетичне програмування. Генетичне програмування може бути використане для створення шахових

рушіїв з метою оптимізації їхньої ефективності та придатності. У генетичному програмуванні починається з випадкового створення тисяч комп'ютерних програм, які утворюють популяцію. Ця популяція поступово еволюціонує протягом кількох поколінь. Еволюційний пошук використовує принцип Дарвіна про природний відбір (виживання найкращих) та аналогії різних природних операцій, таких як кросовер, мутація, подвоєння генів, видалення генів.

Генетичне програмування іноді використовує процеси розвитку, при яких ембріон зростає в повноцінний організм. Застосування цього методу може включати в себе створення рушіїв, які адаптуються та розвиваються протягом еволюційного процесу.

Стисло говорячи, ГП застосовує принципи еволюції для створення оптимальних комп'ютерних програм. В даному контексті вказується, що ГП може автоматично створювати загальні (параметризовані) рішення проблем у вигляді графічної структури, де вузли або ребра представляють компоненти, а значення параметрів цих компонентів задаються математичними виразами із вільними змінними. Таким чином, ГП може автоматично створювати загальне рішення проблеми у вигляді параметризованої топології. [11]

У існуючих дослідженнях [12] і [13] було доведено успішне застосування генетичних алгоритмів для покращення шахових рушіїв. Зокрема у першому дослідженні розподілений генетичний алгоритм дозволив покращити гравецькі можливості існуючого шахового рушія, для гри білими та чорними фігурами. Зазначається, що оптимізація евристик для гри чорними виявилася менш ефективною, ніж для гри білими. Автори також вказують на певні обмеження у попередніх експериментах та планують подальше вдосконалення генетичного алгоритму, зокрема, планують оптимізувати всі оцінювальні коефіцієнти програми індивідуально та використовувати більшу популяцію для оптимізації.

У другому дослідженні показано, що генетичні алгоритми можуть успішно використовуватися для визначення оптимальних значень параметрів у числовій оцінці шахової позиції. Автор приходить до висновку, що отримані значення параметрів у роботі можна розглядати як оптимальні для вибраних параметрів та

розробленої реалізації. Параметри для фігур збіглися з очікуваними значеннями і, ймовірно, вони будуть успішно використовуватися у поєднанні з більш складним алгоритмом пошуку. Автор вважає, що вони не повинні раптово змінювати якість гри при використанні в іншій функції оцінювання. Однак інші параметри можуть бути неправильними, оскільки їх налаштування може бути порушено. Таким чином, оптимальність таких тонких параметрів, як, наприклад, «сусідній пішак», повинна базуватися на відмінній техніці пошуку.

Використання генетичних алгоритмів для створення шахових рушіїв має свої переваги. Однією з головних переваг є здатність алгоритмів адаптуватися та оптимізувати рушії на основі результатів гри. Генетичні алгоритми можуть ефективно враховувати різноманітні параметри та стратегії, що може призвести до створення гнучких та добре пристосованих рушіїв.

Ще однією перевагою є можливість автоматизованого покращення рушіїв без прямого втручання людини. Генетичні алгоритми можуть вести пошук оптимальних стратегій та параметрів, що робить їх ефективним інструментом для розвитку шахових рушіїв.

Але з іншого боку, використання генетичних алгоритмів для створення шахових рушіїв вносить кілька потенційних недоліків у процес оптимізації. По-перше, такі алгоритми можуть бути обмеженими у просторі рішень, не враховуючи всі можливі варіанти та стратегії гри. Великі обчислювальні ресурси можуть бути необхідними для ефективної роботи генетичних алгоритмів, особливо при оптимізації складних функцій оцінювання. Залежність від початкових умов і параметрів може призвести до невизначеності результатів, а визначення правильної фітнес-функції для шахової гри може бути нетривіальним завданням.

1.3 Визначення підходу для створення набору даних

На даний час найбільше уваги приділяється глибинному навчанню та навчанню з підкріпленням. Алгоритми ГН, особливо нейронні мережі, процвітають на великих обсягах даних. Зважаючи на вибух даних у різних сферах, від зобра-

жень і тексту до структурованих даних, моделі можуть використовувати ці величезні набори даних для навчання, дозволяючи їм вивчати складні патерни і уявлення.

Для реалізації шахового агента буде використано глибинне навчання. Використання методів ГН для розробки шахових рушіїв має безліч переваг, які революціонізують в сфері комп'ютерних шахів. По-перше, ГН дозволяє створювати високоадаптивні та динамічні шахові рушії. Традиційні рушії часто покладаються на функції оцінювання та алгоритми пошуку, які можуть бути обмеженими у своїй здатності обробляти складні позиції. На відміну від них, моделі глибокого навчання можуть автоматично вивчати складні закономірності та особливості з величезних обсягів ігрових даних, що дозволяє їм приймати більш обґрунтовані та нюансовані рішення в різних позиціях.

Крім того, методи ГН мають потенціал для поглиблення розуміння шахів на фундаментальному рівні. Аналізуючи величезні масиви даних партій, і використовуючи передові архітектури нейронних мереж, моделі ГН можуть відкрити глибше розуміння основних принципів і стратегічних нюансів гри. Це нове розуміння може не лише покращити продуктивність шахових рушіїв, але й зробити внесок у ширшу сферу досліджень штучного інтелекту, проливаючи світло на складні процеси прийняття рішень та розпізнавання шаблонів у стратегічних сферах. Загалом, інтеграція глибинного навчання в розробку шахових рушіїв є багатообіцяючим шляхом для розвитку як можливостей комп'ютерних шахів, так і розуміння самої гри.

Доступність фреймворків ГН з відкритим кодом, таких як TensorFlow і PyTorch, демократизувала впровадження цих технологій. Ці фреймворки надають дослідникам і розробникам доступні інструменти для експериментів і впровадження ГН і НП моделей.

Створення шахового рушія на основі машинного навчання передбачає навчання моделі на наборі даних шахових позицій і ходів. Існує кілька способів отримання наборів даних для цієї мети:

- Публічні набори даних;

- Онлайн-платформи;
- Проекти з відкритим кодом;
- Генерація власних даних.

Створення наборів даних для моделей машинного навчання передбачає різні підходи, кожен з яких має свої переваги та недоліки. Використання публічних наборів даних пропонує велику і різноманітну колекцію прикладів з реального світу, що забезпечує широкий спектр сценаріїв для навчання моделей. Однак до недоліків можна віднести потенційні варіації в якості даних і брак специфічності для певних навчальних цілей. Онлайн-платформи, що пропонують API для доступу до даних, створених користувачами, дають уявлення про поточні тенденції та реальний ігровий процес. Тим не менш, можуть виникати обмеження, пов'язані з умовами надання послуг і ступенем контролю за якістю даних. Проекти з відкритим вихідним кодом надають дані експертного рівня та структуровані ресурси для експериментів, але ці набори даних можуть бути меншими за розміром.

Нарешті, створення власних даних дозволяє повністю контролювати і налаштовувати їх, але це може бути ресурсномістким і вносити упередження, що відображають ваш стиль гри. При виборі підходу часто оптимальною може бути гібридна стратегія, що поєднує сильні сторони різних методів, досягаючи балансу між різноманітністю, якістю та контролем при створенні наборів даних.

Для проведення дослідження пропонується створити власний набір даних використовуючи існуючі потужні шахові рушії. Зокрема пропонується використати процес гри одного рушія з самим собою як джерело партій, а також використати гру двох потужних шахових рушіїв між собою. Такий підхід дозволить повністю контролювати і налаштовувати їх. Також таким чином можна отримати набір даних, який виключає потенційні варіації в якості даних і брак специфічності для певних навчальних цілей. Такий підхід дозволить «зіткнути» обидва

існуючі підходи (заснований на пошуку і на оцінюванні), що дозволить відобразити переваги в наборі даних, що в свою чергу, відобразиться на якості гри інтелектуального шахового агента.

1.4 Аналіз вимог та постановка завдання

Шахи — це гра, яка вимагає від гравця не лише реакції на поточні обставини, але й здатності до глибокого стратегічного планування. Інтелектуальний агент повинен мати відмінне розуміння основних стратегій та здатність пристосовувати свою стратегію в залежності від обставин та ходу гри.

Для реалізації інтелектуального агента для гри в шахи слід перш за все формувати набір вимог, яким він має відповідати.

Інтелектуальний агент для гри в шахи повинен вміти аналізувати стан гри та планувати свої наступні кроки. Це означає, що агент повинен мати здатність аналізувати стан гри, включаючи позиції фігур на дошці, і здійснювати розрахунки, щоб зрозуміти, які можливості є у нього наступним ходом.

Агент повинен мати здатність оцінювати поточний стан гри та приймати рішення на основі цієї оцінки. Наприклад, якщо агент визначає, що його король знаходиться під загрозою, то він повинен зробити хід, щоб захистити короля. Тобто поведінка має бути націлена на пошук оптимальних і виважених ходів, а не випадковою, такою що лише відповідає правилам гри. У рушія повинне бути тактичне бачення гри. Наприклад, агент може використовувати тактику “вилки”, щоб одночасно загрожувати двом фігурам суперника, або стратегію “розвитку”, щоб швидко розгорнути свої фігури на дошці.

Гра в шахи вимагає раціонального використання ресурсів, таких як час та фігури на дошці. Інтелектуальний агент повинен оптимізувати свої рухи, максимізуючи використання кожної фігури та стратегічно розподіляючи час на обміркування кожного кроку.

У грі в шахи ситуації можуть раптово змінюватися, і інтелектуальний агент повинен бути здатний адаптуватися до нових обставин. Застосування методів ма-

шинного навчання та адаптивних алгоритмів дозволить агенту швидко відгукатися на зміни стратегії супротивника та підлаштовувати свої ходи для досягнення максимальної ефективності.

Таким чином, вимоги до інтелектуального агента для гри в шахи обумовлені необхідністю поєднання глибокого стратегічного розуміння, раціонального управління ресурсами, тактичної ефективності та здатності до адаптації, що в сукупності робить його спроможним конкурувати та розвиватися в умовах динамічного геймплею гри в шахи.

Розробка інтелектуального агента для гри в шахи вимагає комплексного підходу та врахування різноманітних аспектів. Основні вимоги до такого агента включають розуміння гри, ефективне управління ресурсами, тактичну ефективність та здатність до адаптації, а також задовільний рівень гри.

Провівши аналіз вимог, яким має відповідати шаховий рушій, можна сформулювати перелік завдань, які необхідно виконати, щоб отримати шаховий рушій, який їх задовольнятиме. Він складається з наступних пунктів:

- 1) Реалізація механізму проведення ігор між різними шаховими рушіями.
- 2) Створення алгоритму збору і запису синтезованих партій для подальшого формування навчального набору.
- 3) Вибір архітектури нейронної мережі, на якій базуватиметься інтелектуальний агент.
- 4) Приведення зібраних даних до виду необхідного для подачі на вхід нейронній мережі.
- 5) Реалізація архітектури нейронної мережі і її тренування.
- 6) Здійснення оптимізації параметрів нейронної моделі.
- 7) Забезпечення можливості взаємодії з агентом та його коректної роботи під час гри.
- 8) Проведення тестів для оцінки ефективності та правильності роботи.
- 9) Підбиття підсумків дослідження, визначення потенційних напрямків покращення.

Основний фокус зусиль буде спрямовано на синтез даних для навчання та їхню подальшу обробку. Також одним з найважливіших етапів є реалізація архітектури нейронної моделі та її навчання на отриманому наборі даних.

Для реалізації синтезу навчальних даних буде використано мову програмування Python. Ця мова містить велику кількість бібліотек, які спеціалізуються на шахах. Наприклад, бібліотека chess надає зручний API для презентації дошки, взаємодії з нею і декодування в різні формати. Також існуючі шахові рушії мають свої бібліотеки для роботи з ними використовуючи мову Python. Що ще робить цю мову програмування дуже привабливою для проведення дослідження, це те, що величезна кількість інструментів та бібліотек для роботи з нейронними мережами написана саме для цієї мови. Для проведення реалізації і тренування нейронної мережі можна використати фреймворк PyTorch. API PyTorch вважається більш інтуїтивно зрозумілим, що полегшує користувачам розуміння та написання коду. Синтаксис чистий і нагадує NumPy, що робить його доступним для дослідників і розробників, які вже знайомі з Python і науковими обчисленнями.

Висновки до розділу 1

В даному розділі було розглянуто основну проблематику кваліфікаційної роботи. На основі цього було визначено напрямки подальшого дослідження можливих варіантів вирішення поставленої проблеми.

Потенційним найкращим рішенням може бути використання методів машинного навчання таких як глибинне навчання. Ці підходи показали хороші результати застосування для вирішення проблеми створення шахових рушіїв. Також було розглянуто і визначено спосіб отримання навчальних даних, який полегатиме в генерації партій використовуючи існуючі потужні шахові інтелектуальні агенти. Було проаналізовано інші способи отримання навчальних наборів, описано їхні переваги та недоліки.

Було ретельно вивчено цілі, обмеження та функціональні можливості шахового рушія, ми отримали чітке розуміння того, чого потрібно досягти в результаті проведення дослідження. Сформовано перелік вимог, яким має відповідати шаховий інтелект.

Також було створено перелік етапів, які слід реалізувати для задоволення вимог. Основний виклик створення шахового агенту полягає в синтезі набору даних для навчання та вибір і реалізація нейронної мережі.

Було обрано основний інструментарій для розробки. Цей розділ дає важливий фундамент для подальшого проведення дослідження і безпосередньої реалізації шахового рушія.

РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ АЛГОРИТМІВ НАВЧАННЯ НЕЙРОННИХ МОДЕЛЕЙ

2.1 Синтез даних для навчання

Для отримання набору даних для тренування моделі, яка гратиме в шахи пропонується реалізувати наступні підходи. Перший полягатиме в тому, щоб взяти готовий шаховий рушій заснований на пошуку, зробити його суперником інший рушій, заснований на оцінюванні, провести певну кількість партій, де вони гратимуть між собою, обираючи один із найкращих, на їхню думку ходів. Кожен хід і кожную позицію на дошці записувати в файл. Другий підхід, який пропонується використати для синтезу даних, полягатиме в тому, що буде взято існуючий рушій і змусити його грати з самим собою.

Обидва підходи дають можливість отримати потрібну кількість партій, в якій ходи будуть не випадковими і виключатимуть ймовірність відверто хибних ходів, які притаманні людям. Також реалізувавши обидва з них буде проведено порівняння результатів тренування власного інтелектуального шахового агента.

Для реалізації цього підходу генерації даних на основі поєдинку двох шахових рушіїв використаємо два найпотужніші публічно доступних шахові рушії: Stockfish і LeelaChess Zero. Stockfish – це один з найпотужніших та найпопулярніших шахових двигунів, який використовується для аналізу шахових позицій, гри та розробки комп'ютерних шахових програм. Stockfish є відкритим програмним забезпеченням і доступний для використання безкоштовно.

Він володіє потужною силою гри і часто вважається одним із найсильніших комп'ютерних шахових двигунів. Він використовує високо оптимізовані алгоритми та постійно вдосконалюється. У своїй основі він використовує комбінацію різних алгоритмів для визначення кращих ходів у шахах. Він включає у себе алгоритми пошуку (такі як алгоритм альфа-бета), оцінки позиції, глибокі нейронні мережі. Stockfish використовується не лише як самостійний шаховий двигун, але і як основа для різноманітних шахових проектів та досліджень у сфері штучного інтелекту та гри в шахи.

Його суперником зробимо шаховий рушій під назвою Leela Chess Zero. Leela Chess Zero (LCZero або Lc0) – це шаховий рушій з відкритим вихідним кодом, який здобув популярність завдяки своєму унікальному підходу до шахового програмування. На відміну від традиційних шахових рушіїв, які покладаються на алгоритми, засновані на правилах, і ретельно розроблені функції оцінювання, LCZero використовує нейронні мережі та методи навчання з підкріпленням для гри в шахи.

LCZero використовує архітектуру нейронної мережі, зокрема, згорткову нейронну мережу (CNN), для оцінки позицій і прогнозування ходів. Нейронна мережа навчається за допомогою навчання з підкріпленням, що дозволяє їй покращувати свою ігрову силу з часом завдяки самотійній грі.

Процес навчання включає в себе самотійну гру, де движок грає проти самого себе і вчиться на результатах. Нейронна мережа оновлюється на основі результатів цих ігор з метою оптимізації її оціночної функції та покращення можливостей прийняття рішень.

LCZero випускається як проект з відкритим вихідним кодом, що дозволяє шаховій спільноті зробити свій внесок у його розвиток. Відкритий характер проекту сприяє співпраці та інноваціям, що призводить до постійного вдосконалення ігрової сили рушія. Це дозволяє безперешкодно використовувати його для дослідження, адже рушій можна без проблем завантажити і модифікувати потрібним чином.

Для реалізації генерації даних на основі поєдинку двох рушіїв слід вирішити наступні завдання:

- Створити середовище для гри;
- Сконфігурувати шахові рушії (за необхідності);
- Підібрати параметри поєдинків;
- Визначити умови завершення гри;
- Забезпечити запис всіх зроблених ходів;

Така процедура є доволі поширеною у розробників і дослідників різного роду шахових рушіїв. Вони часто стикаються з потребою визначення покращення чи погіршення результатів попередньої та наступної версії рушія або ж визначення, який з рушіїв є сильнішим. Тому для цього існують готові програмні рішення, які дозволяють реалізувати вище згадані сценарії. Інструмент який буде використано для проведення генерації даних на основі поєдинків між двома рушіями має назву Cute Chess. Програма Cute Chess є графічним шаховим інтерфейсом. Її можна запустити як з командного рядка, так і з панелі запуску додатків у вашому робочому середовищі. Використаємо версію для командного рядка, так як графічний інтерфейс потребує встановлення додаткових залежностей, але весь необхідний функціонал доступний і в консольному інтерфейсі.

Далі слід завантажити відповідні версії шахових рушіїв з офіційних сайтів та ознайомитися з їхньою документацією. З рушієм Stockfish в цьому плані все простіше, так як його можна завантажити та відразу використовувати. З рушієм LCZero все складніше. Так як він використовує в своїй основі нейронні мережі, то для різних цілей та конфігурацій апаратної частини є різні версії файлів ваги для його нейронної мережі. У машинному навчанні файлом ваги зазвичай називають файл, який містить вивчені параметри моделі, що навчається. Ці параметри, часто представлені у вигляді ваг і зсувів, відображають взаємозв'язок між вхідними характеристиками і вихідними прогнозами, зробленими моделлю.

На офіційному сайті LCZero є набір різних нейронних мереж для різних потреб, їхній опис зображено на рисунку 2.1. Загалом вони дозволяють гнучко підлаштовуватись під потреби та можливості апаратного забезпечення.

Наприклад, для систем з невеликою кількістю обчислювальних спроможностей рекомендують використовувати менші мережі. Було проведено підбір і для генерації було залишено стандартну мережу, яка поставляється разом з клієнтом, так як вона найкраще є найкраще збалансованою для поставленого завдання.

Actual networks

[Edit on wiki](#)

In general, for game analysis and long calculation time per move, the largest network compatible with your hardware is recommended. In use cases with very low calculation time per move or slow hardware, a smaller network might be a better choice.

Network Size	Purpose	Filters	Blocks	GPU Memory Usage	File Size	Network
Large	GPU	768	15 (mish activation)	2.4 GB	160-170 MB	T82-768x15x24h-swa-7464000 (Right-click → "Save link as...")
Medium	GPU/CPU	512	15 (mish activation)	1.8 GB	140-150 MB	T1-512x15x8h-distilled-swa-3395000 (Right-click → "Save link as...")
Small	GPU/CPU	256	10 (mish activation)	1.6 GB	30-40 MB	T1-256x10-distilled-swa-2432500 (Right-click → "Save link as...")
Very Small	Sparring vs. Humans	≤128	≤10	-	≤10 MB	see below

Рис. 2.1. Доступні актуальні нейронні мережі

Після проведення конфігурації LCZero перейдемо до налаштування середувища і правил гри між рушіями. Для цього звернемося до документації рушія LCZero, який надає детальні інструкції про те, як використовувати Cute Chess разом з ним [15].

Після встановлення шахових рушіїв слід створити файл під назвою engines.js, в якому буде міститися інформація про шахові рушії і в якому режимі з ними працювати. Для потреб дослідження, цей файл має вигляд як на рисунку 2.2.

```
[
  {
    "command": "lc0-v0.30.0-windows-gpu-nvidia-cudnn\\lc0.exe",
    "name": "lc0",
    "options": [
      {
        "name": "Threads",
        "value": 2
      }
    ],
    "protocol": "uci",
    "workingDirectory": "/"
  },
  {
    "command": "stockfish-windows-x86-64\\stockfish\\stockfish-windows-x86-64.exe",
    "name": "stockfish",
    "options": [
      {
        "name": "Hash",
        "value": 256
      },
      {
        "name": "Threads",
        "value": 12
      }
    ],
    "protocol": "uci",
    "workingDirectory": "/"
  }
]
```

Рис. 2.2. Конфігурація для Cute Chess

Він містить інформацію про команди для запуску рушіїв, їхнє розташування на машині тощо.

Наступним кроком буде сформувані команду для запуску поєдинку. Cute Chess приймає велику кількість параметрів, які дозволяють гнучко налаштувати процес гри між рушіями. На рисунку 2.3 зображено параметри зі значеннями, які використовувались для проведення

```
1  cutechess-cli ·
2  · -tournament gauntlet ·
3  · -concurrency 1 ·
4  · -pgnout output_games.pgn
5  · -engine conf=stockfish tc=5.2+0.0965 ·
6  · -engine conf=lc0 tc=5.2+1 ·
7  · -draw movenumber=40 movecount=4 score=8 ·
8  · -each proto=uci -openings file=openings/openings-6ply-1000.pgn policy=round ·
9  · -repeat ·
10 · -rounds 5000 ·
11 · -games 2
```

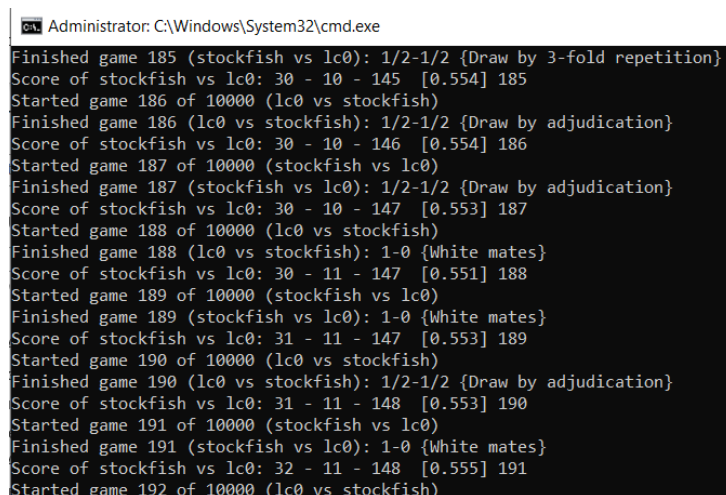
Рис. 2.3. Команда для запуску ігор між рушіями

Далі пройдемося детально по параметрам і опишемо їхні значення:

- -tournament gauntlet: ця опція вказує на те, що стиль турніру - рукавичка. Це означає, що всі інші рушії у списку будуть грати з першим рушієм;
- -concurrency 1: ця опція встановлює, що одночасно проходитиме лише одна гра. Не рекомендується ставити значення більше ніж 1, адже це може привести до того, що рушії будуть конкурувати за обчислювальні ресурси;
- -pgnout output_pgn_file.pgn: назва файлу, куди будуть зберігатися партії;
- -engine conf=stockfish tc=5.2+0.0965: цей параметр встановлює конфігурацію часу, відведеного на гру і інкремент після ходу. Варто зазначити, що для LCZero слід встановлювати більший інкремент, так як цьому рушію треба мати більше часу для визначення ходу.
- -draw movenumber=40 movecount=4 score=8: цей параметр відповідає за закінчення гри з нічиєю. Якщо пройшло більше 4 ходів, а обидва рушії мають рахунок ≤ 8 пішаків, і було зіграно принаймні 40 ходів, оголосить нічию. Захоплення або ходи пішаків обнуляють лічильник.
- -each proto=uci: обидва рушії використовують протокол UCI;

– -openings file=openings/openings-6ply-1000.pgn policy=round: цей параметр задає файл з дебютами. Цей файл важливий, так як якщо ви грати матчі без файлу дебютів, програми часто гратимуть одні й ті ж партії знову і знову, що, очевидно, проблематично. Якщо примусите їх робити унікальні дебюти для кожної пари партій або раунду, ми отримаємо набагато кращі результати.

Після цього, можна запустити процес проведення ігор між рушіями. На рисунку 2.4 зображено процес проведення ігор між рушіями.



```
Administrator: C:\Windows\System32\cmd.exe
Finished game 185 (stockfish vs lc0): 1/2-1/2 {Draw by 3-fold repetition}
Score of stockfish vs lc0: 30 - 10 - 145 [0.554] 185
Started game 186 of 10000 (lc0 vs stockfish)
Finished game 186 (lc0 vs stockfish): 1/2-1/2 {Draw by adjudication}
Score of stockfish vs lc0: 30 - 10 - 146 [0.554] 186
Started game 187 of 10000 (stockfish vs lc0)
Finished game 187 (stockfish vs lc0): 1/2-1/2 {Draw by adjudication}
Score of stockfish vs lc0: 30 - 10 - 147 [0.553] 187
Started game 188 of 10000 (lc0 vs stockfish)
Finished game 188 (lc0 vs stockfish): 1-0 {White mates}
Score of stockfish vs lc0: 30 - 11 - 147 [0.551] 188
Started game 189 of 10000 (stockfish vs lc0)
Finished game 189 (stockfish vs lc0): 1-0 {White mates}
Score of stockfish vs lc0: 31 - 11 - 147 [0.553] 189
Started game 190 of 10000 (lc0 vs stockfish)
Finished game 190 (lc0 vs stockfish): 1/2-1/2 {Draw by adjudication}
Score of stockfish vs lc0: 31 - 11 - 148 [0.553] 190
Started game 191 of 10000 (stockfish vs lc0)
Finished game 191 (stockfish vs lc0): 1-0 {White mates}
Score of stockfish vs lc0: 32 - 11 - 148 [0.555] 191
Started game 192 of 10000 (lc0 vs stockfish)
```

Рис. 2.4. Процес проведення ігор

Під час проведення ігор в консоль виводиться інформація про поточну гру, загальний рахунок, причину завершення гри.

Далі опишемо процес синтезу даних з використанням самостійної гри. Самостійна гра – це техніка, яка використовується у розвитку шахових рушіїв. Завдяки самонавчанню алгоритми грають самі з собою, генеруючи значну кількість тренувальних даних, досліджуючи різноманітні стратегії та позиції. Цей ітеративний процес дозволяє рушіям вдосконалювати свої функції оцінювання та алгоритми пошуку, поступово підвищуючи їхню ігрову силу. Крім того, самогра сприяє інтеграції навчання з підкріпленням, що дозволяє штучним інтелектам вчитися на власному досвіді та ітеративно вдосконалюватися. Цей метод слугує не лише для вдосконалення стратегії, але й для тестування та перевірки, гарантуючи, що модифікації призводять до кращої гри. Зменшуючи залежність від даних, створених людиною, самогра дозволяє рушіям розвивати свої унікальні стилі та інноваційні стратегії.

Для отримання набору даних для тренування моделі, яка гратиме в шахи пропонується зробити наступне: взяти готовий шаховий рушій, провести певну кількість партій, де він гратиме сам з собою, обираючи один із найкращих, на його думку ходів. Кожен хід і кожен позицію на дошці записувати в файл.

Таким чином ми можемо отримати потрібну кількість партій, в якій ходи будуть не випадковими і виключатимуть ймовірність хибних ходів, які притаманні людям.

Для реалізації цього підходу генерації даних використаємо потужний шаховий рушій, який має назву Stockfish, який ми використовували для генерації даних в поєдинку з іншим рушієм. Він вважається найсильнішим публічно доступним шаховим рушієм на даний момент.

Тому першим кроком в отриманні набору даних є реалізація скрипту, який взаємодіятиме з вище згаданим шаховим рушієм і зберігатиме всі його партії. Для цього потрібно завантажити сам шаховий рушій з офіційного сайту і розробити скрипт. Реалізуємо метод mineGames, який відповідатиме за проведення партій з використанням Stockfish і записуватиме всі його дії у файли з назвою movesAndPositions[n], де n – номер зіграної партії. Загалом, алгоритм роботи цього методу можна описати наступним чином:

- 1) Встановлюємо максимальний ліміт, щоб гра не тривала нескінченно;
- 2) Запускаємо задану кількість ігор і переконуємося, що як Stockfish, так і пакет шахів Chess повертаються до початкової позиції;
- 3) Отримуємо три найкращі рекомендації від Stockfish і обираємо одну з них для ходу (80% ймовірність вибору найкращого ходу, 15% - для другого найкращого ходу, 5% - для третього найкращого ходу). Вибір не завжди найкращого ходу виконується для того, щоб вибір був більш стохастичним;
- 4) Обираємо хід (забезпечуючи відсутність помилок, навіть якщо можливі менше ніж три можливих ходи), зберігаємо позицію дошки за допомогою FEN (спосіб кодування позиції на шаховій дошці), а також виконаний хід;

5) Якщо гра завершена, припиняємо цикл і зберігаємо всі позиції та виконані ходи з цих позицій. Якщо гра не завершена, продовжуємо робити ходи до завершення гри.

Повний лістинг цього методу буде наведено в додатку А.

2.2 Кодування навчальних даних

Наступний крок, який потрібно виконати для отримання навчального набору даних – це кодування. Кодування є важливим елементом у штучному інтелекті, оскільки воно дозволяє нам подавати проблеми у зрозумілому для штучного інтелекту вигляді.

Замість зображення шахової дошки або рядка, який представляє хід у шахах, наприклад, «e2e4» можна представити за допомогою масивів (списків чисел).

Так як для генерації даних використовувалось два різних підходи, то процес кодування матиме певні відмінності для них. Для збереження результатів генерації даних на основі поєдинку було використано запис в PGN файл. PGN (Portable Game Notation) слугує стандартизованим форматом для запису шахових партій, що полегшує їх обмін та аналіз на різних платформах і програмах. Він інкапсулює ключову інформацію, таку як ходи, анотації та метадані гри, у зручний для читання текстовий формат. Кожен PGN-файл зазвичай містить одну або кілька шахових партій, кожна з яких представлена послідовністю ходів у стандартній алгебраїчній нотації. Крім того, PGN-файли часто містять заголовки для забезпечення контексту, наприклад, назву змагання, імена гравців, результат та інші важливі деталі. Цей формат полегшує обмін шаховими партіями між гравцями, тренерами та ентузіастами, сприяючи поширенню та архівуванню даних, пов'язаних з шахами. [20]

Для кодування отриманих партій в формат потрібний для подачі на вхід нейронній мережі слід виконати наступні дії:

- 1) Прочитати файл з партіями;
- 2) Для кожної партії прочитати ходи;

3) Записати кожен хід і позицію після ходу в файл, для кожної гри отримаємо файл з ходами і позиціями;

4) Отримані файли з ходами і позиціями закодуємо подібним чином, як це робиться для Alpha Zero, після чого отримаємо набір файлів з закодованими позиціями і набір файлів із закодованими ходами.

Після цього отримані закодовані файли можна буде використовувати для подання на вхід для навчання нейронній моделі. Для конвертації отриманих партій з формату PGN в формат потрібний для подальшого кодування було реалізовано спеціальний скрипт, його лістинг зображено на рисунку 2.5.

```
import chess.pgn
from helpers import saveData

pgn_file = open("src/data/match/output_games.pgn")

while True:
    game = chess.pgn.read_game(pgn_file)

    if game is None:
        break

    currentGameMoves = []
    currentGamePositions = []
    board = chess.Board()

    for move in game.mainline_moves():
        # Update the board position after each move
        print("move:", move.uci(), "game number")
        print("fen position:", board.fen())
        board.push(move)

        currentGameMoves.append(move.uci())
        currentGamePositions.append(board.fen())

    saveData(currentGameMoves, currentGamePositions, "match")

pgn_file.close()
```

Рис. 2.5. Конвертер PGN партій

Для кодування ігор отриманих шляхом самотійної гри слід виконати ті ж кроки, але з тією відмінністю, що ці ігри уже будуть записані в форматі хід-позиція, тож їх можна буде відразу кодувати як описано в пункті 4. Наразі можна зробити висновок, що кодування партій отриманих шляхом самотійної гри є більш простим, так як не вимагає переходу від однієї форми запису партій до іншої. Це зумовлено тим, що для генерації гри між різними рушіями використовується Cute Chess, в якому партії записуються в PGN файл, при цьому позиції не записуються і їх слід записати самотійно на основі зроблених ходів, що вимагає додаткових дій.

2.3 Тренування моделей і практична реалізація

Після приведення даних в вигляд, потрібний для навчання нейронної моделі, можна переходити до створення архітектури і тренування. Для отримання моделі застосуємо підхід імітаційного навчання. Імітаційне навчання, також відоме як навчання на прикладах або навчання шляхом наслідування, - це парадигма машинного навчання, в якій агент вчиться виконувати завдання, спостерігаючи та імітуючи демонстрації, надані експертом. Замість того, щоб явно визначати функцію винагороди або оптимальну політику, імітаційне навчання має на меті імітувати поведінку експерта на основі спостережуваних демонстрацій.

При імітаційному навчанні агент зазвичай вивчає політику, яка зіставляє спостереження з діями, безпосередньо імітуючи дії, продемонстровані експертом. Цього можна досягти за допомогою різних методів машинного навчання, таких як контрольоване навчання або навчання з підкріпленням, коли демонстрації експерта є формою контрольованого зворотного зв'язку.

Імітаційне навчання особливо корисне в сценаріях, де складно визначити функцію винагороди або де розробка оптимальної політики є складною. Воно застосовується в робототехніці, автономному водінні, обробці природної мови та інших сферах, де людський досвід може надати цінні вказівки для вивчення складних завдань.

Однією з ключових проблем імітаційного навчання є вирішення проблеми зсуву розподілу, коли розподіл станів, з якими стикаються під час навчання, може відрізнятись від розподілу станів під час розгортання. Вирішення цієї проблеми часто включає в себе такі методи, як адаптація до предметної області, надійна оптимізація або включення невизначеності в процес навчання, щоб гарантувати, що ефективність агента добре узагальнюється в нових ситуаціях.

Далі налаштуємо середовище для тренування моделей. Так як маємо два різних підходи для отримання навчальних даних, то і в результаті навчання отримаємо дві моделі: одна навчатиметься на партіях, згенерованих в результаті самогри, інша на основі партій між двома різними шаховими рушіями.

Навчання моделей буде проводитися локально, безпосередньо на робочій машині. Так, для цього можна було б використати Google Colaboratory. Google Colaboratory, яку часто називають «Google Colab», - це безкоштовна хмарна платформа від Google, що дозволяє користувачам писати та виконувати код на Python в інтерактивному середовищі на основі браузера. Вона базується на Jupyter Notebooks і забезпечує зручний спосіб розробки та запуску коду на Python, зокрема для задач машинного навчання та аналізу даних. Проте зважаючи на те, що весь об'єм навчальних даних було генеровано локально і його перенос зайняв би велику кількість простору на хмарному диску і часу, ще й враховуючи те, що безкоштовна версія акаунту Google дає загалом 15GB простору для користувацьких файлів, то цей варіант навчання моделей було відкинуто. Також Google Colaboratory обмежує час користувацьких сесій, тому навчання довелося б поновлювати приблизно кожні 3 години, що також накладає незручності, адже на власній машині можна проводити навчання стільки скільки потрібно.

Створивши новий ноутбук локально, встановлюємо і підключаємо необхідні пакети та бібліотеки. Їхній перелік зображено на рисунку 2.6. Зокрема, нам знадобиться бібліотека gym-chess для кодування та декодування ходів і бібліотека chess для зручного проведення партій.

```
!pip3 install chess

!pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118

!pip install torchvision

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
► Launch TensorBoard Session
from torch.utils.tensorboard import SummaryWriter
from datetime import datetime
import gym
import gym_chess
import os
import chess
from tqdm import tqdm
from gym_chess.alphazero.move_encoding import utils
from pathlib import Path
from typing import Optional
```

Рис. 2.6. Встановлення і підключення потрібних пакетів

Також, аби процес тренування проходив швидше, слід переконатися, що використовуватимуться CUDA ядра виділеного графічного процесора для нашого ноутбука.

Наступним кроком буде завантажити всі дані для тренування. Для цього читаємо кожен файл з ходами і позиціями і розширюємо загальні масиви даних прочитаними даними. Після цього робимо тренувальний та валідаційний набір даних. Лістинг коду, відповідального за завантаження та створення наборів зображено на рисунку 2.7.

```
BATCH_SIZE = 4

allMoves = []
allBoards = []

files = os.listdir("./data/selfPlay/prepared")
numOfEach = len(files) // 2

for i in range(numOfEach):
    try:
        moves = np.load(f"./data/selfPlay/prepared/moves{i}.npy", allow_pickle=True)
        boards = np.load(f"./data/selfPlay/prepared/positions{i}.npy", allow_pickle=True)
        if len(moves) != len(boards):
            print("ERROR ON i = ", i, len(moves), len(boards))
        allMoves.extend(moves)
        allBoards.extend(boards)
    except:
        print("error: could not load ", i, ", but is still going")

allMoves = np.array(allMoves)[:int(len(allMoves))]
allBoards = np.array(allBoards)[:int(len(allBoards))]
assert len(allMoves) == len(allBoards), "MUST BE OF SAME LENGTH"

trainDataIdx = int(len(allMoves) * 0.8)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
allBoards = torch.from_numpy(np.asarray(allBoards)).to(device)
allMoves = torch.from_numpy(np.asarray(allMoves)).to(device)

training_set = torch.utils.data.TensorDataset(
    allBoards[:trainDataIdx], allMoves[:trainDataIdx]
)
test_set = torch.utils.data.TensorDataset(
    allBoards[trainDataIdx:], allMoves[trainDataIdx:]
)

training_loader = torch.utils.data.DataLoader(
    training_set, batch_size=BATCH_SIZE, shuffle=True
)
validation_loader = torch.utils.data.DataLoader(
    test_set, batch_size=BATCH_SIZE, shuffle=False
)
```

Рис. 2.7. Завантаження та створення наборів даних

Наступним кроком буде опис архітектури моделі, яку зображено на рисунку 2.8. Використовуватимемо модель, яка є нейронною мережею прямого поширення, розробленою для задачі класифікації, де метою є передбачення одного з декількох класів (ходів) на основі вхідного представлення позиції на шаховій дошці. Розмір вхідних даних дорівнює 896, що представляє собою одномірний масив розміром 896. Це відповідає вектору ознак, отриманому з позиції на шаховій дошці. Цей розмір вхідних даних підходить для конкретного представлення

шахової дошки, яка може бути дошкою 8x8 з 14 каналами, що представляють різні фігури та інші ознаки. Розмір вихідних даних задано у 4672, що, відповідає кількості унікальних ходів у просторі дії шахової гри. Варто зауважити, що дана архітектура залишає простір для її покращення, такі як включення згорткових шарів, зміна кількості нейронів в лінійних шарах, зміна кількості прихованих шарів і використання різних функцій активації для них, таких як Sigmoid або Tanh.

```
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.INPUT_SIZE = 896
        self.OUTPUT_SIZE = 4672

        self.activation = torch.nn.ReLU()
        self.linear1 = torch.nn.Linear(self.INPUT_SIZE, 1000)
        self.linear2 = torch.nn.Linear(1000, 1000)
        self.linear3 = torch.nn.Linear(1000, 1000)
        self.linear4 = torch.nn.Linear(1000, 200)
        self.linear5 = torch.nn.Linear(200, self.OUTPUT_SIZE)
        self.softmax = torch.nn.Softmax(1)

    def forward(self, x):
        x = x.to(torch.float32)
        x = x.reshape(x.shape[0], -1)
        x = self.linear1(x)
        x = self.activation(x)
        x = self.linear2(x)
        x = self.activation(x)
        x = self.linear3(x)
        x = self.activation(x)
        x = self.linear4(x)
        x = self.activation(x)
        x = self.linear5(x)
        return x
```

Рис. 2.8. Архітектура нейронної мережі

Також клас моделі реалізує метод `predict`, який відповідає безпосередньо за визначення оптимального ходу, базуючись на ситуації на дошці. Метод прогнозування приймає на вхід стан шахової дошки і кодує його в представлення у вигляді масиву за допомогою функції `encodeBoard`. Після зміни форми та перетворення закодованої дошки в тензор PyTorch, він пропускає його через нейронетическу модель, щоб отримати вихідні ймовірності для різних ходів. Ці ймовірності потім перетворюються на розподіл ймовірностей за допомогою активації `softmax`. Метод ітераційно перебирає ймовірності, щоб вибрати хід з найбільшою ймовірністю, декодуючи його з індексного представлення у формат UCI. Якщо обраний хід є легальним, він повертає хід, в іншому випадку цикл продовжується. У випадку помилки декодування метод пропускає проблемний хід. Якщо

не знайдено жодного легального ходу, повертається значення випадковий легальний хід, враховуючи поточний стан дошки. Лістинг методу predict зображено на рисунку 2.9.

```
def predict(self, board: chess.Board):
    """takes in a chess board and returns a chess move object."""
    with torch.no_grad():
        encodedBoard = encodeBoard(board)
        encodedBoard = encodedBoard.reshape(1, -1)
        encodedBoard = torch.from_numpy(encodedBoard)
        res = self.forward(encodedBoard)
        probs = self.softmax(res)

        probs = probs.numpy()[0]

    while len(probs) > 0:
        moveIdx = probs.argmax()
        try:
            uciMove = decodeMove(moveIdx, board)
            if uciMove is None:
                probs = np.delete(probs, moveIdx)
                continue
            move = chess.Move.from_uci(str(uciMove))
            if (move in board.legal_moves):
                return move
            except:
                pass
            probs = np.delete(
                probs, moveIdx
            )

    legal_moves = list(board.legal_moves)
    if legal_moves:
        return random.choice(legal_moves)
    else:
        return None
```

Рис. 2.9. Метод для визначення ходу

Після створення наборів даних і опису архітектури моделі, перейдемо до налаштування процесу тренування. Для цього слід описати методи для проведення тренування однієї епохи та забезпечити створення, зберігання і отримання файлу найкращої моделі. Зосередимося більше на методі для тренування однієї епохи. Метод `train_one_epoch` полегшує навчання нейромережевої моделі для однієї епохи з використанням наданого набору даних (`training_loader`). Він ітераційно проходить через партії пар вхід-вихід, ініціалізуючи градієнти нулем, пропускаючи вхідні дані через модель для отримання прогнозів, обчислюючи втрати між прогнозами і фактичними мітками, поширюючи градієнти в зворотному напрямку і оновлюючи параметри моделі за допомогою оптимізатора. Він відстежує поточні втрати, повідомляючи середні втрати за вказаний інтервал запису-

вачу TensorBoard для візуалізації та моніторингу. Метод повертає останню обчислену втрату, що представляє середню втрату на партію за всю епоху, допомагаючи оцінити прогрес навчання і продуктивність моделі. Лістинг методу зображено на рисунку 2.10.

```
def train_one_epoch(model, optimizer, loss_fn, epoch_index, tb_writer):
    running_loss = 0.0
    last_loss = 0.0

    for i, data in enumerate(training_loader):
        inputs, labels = data

        optimizer.zero_grad()

        outputs = model(inputs)

        loss = loss_fn(outputs, labels)
        loss.backward()

        optimizer.step()

        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar("Loss/train", last_loss, tb_x)
            running_loss = 0.0

    return last_loss
```

Рис. 2.10. Метод train_one_epoch

Далі опишемо код, який відповідає за проведення навчання. Цей код навчає нейронну модель за допомогою PyTorch протягом заданої кількості епох (EPOCHS). Він ініціалізує модель, функцію втрат та оптимізатор, а також налаштовує запис на TensorBoard для запису прогресу навчання. Під час навчання він проходить кожну епоху, виконуючи прямий і зворотний прохід над навчальними даними для оновлення параметрів моделі. Крім того, він оцінює модель на окремому валідаційному наборі даних, щоб відстежувати її продуктивність і запобігти надмірному пристосуванню. Найкраща модель зберігається на основі найнижчої втрати валідації, досягнутої протягом навчання. Нарешті, код виводить найкращу валідаційну втрату, отриману для всіх навчених і збережених моделей. Загалом, цей код організовує процес навчання, відстежує ефективність і зберігає найкращу модель для подальшого використання. Його лістинг буде наведено в додатку А.

2.4 Результати проведеного навчання

Першою було натреновано модель, яка навчалася на даних отриманих шляхом гри двох рушіїв, далі називатимемо її match. На рисунках 2.11 – 2.12 зображено графік мінімізації втрат під час тренування та валідації.

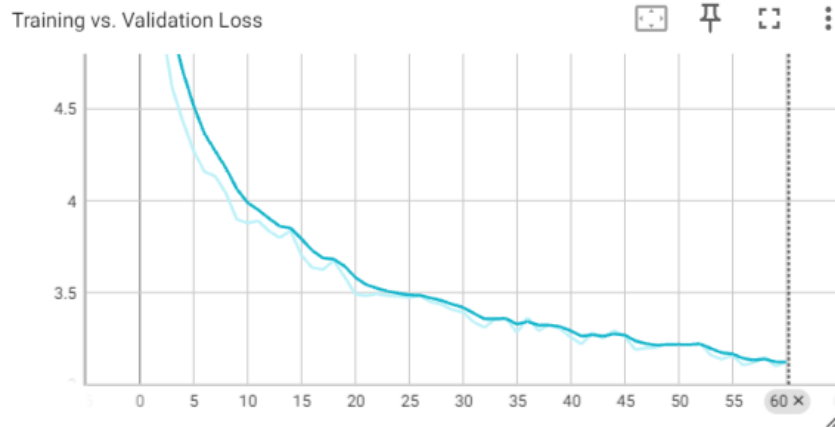


Рис. 2.11. Графік зміни втрат під час тренування match моделі

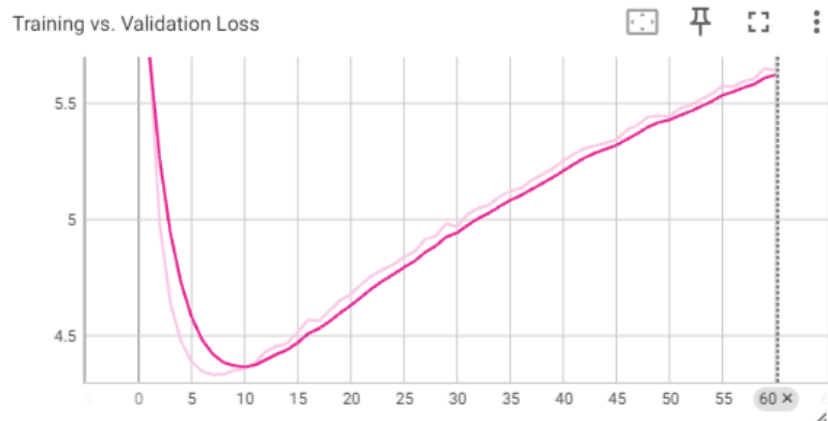


Рис. 2.12. Графік зміни втрат під час валідації match моделі

На рисунку 2.13 – 2.14 зображено графіки зміни втрат для моделі, яка навчалася на даних, отриманих шляхом самотійної гри, далі цю модель називатимемо її selfPlay.

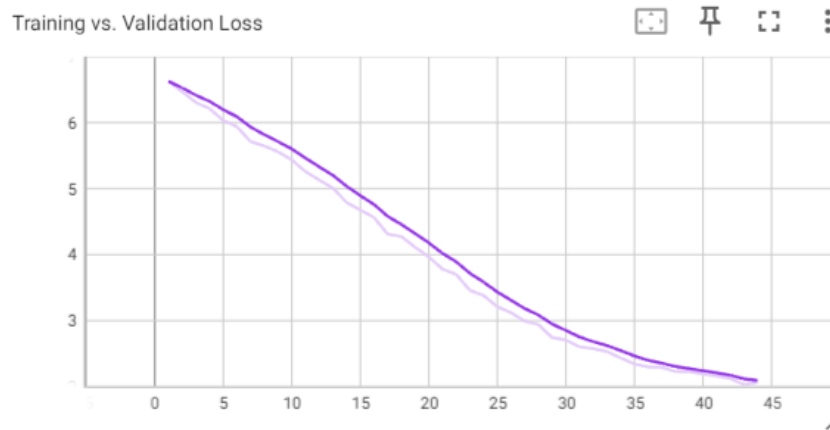


Рис. 2.13. Графік зміни втрат під час тренування selfPlay моделі

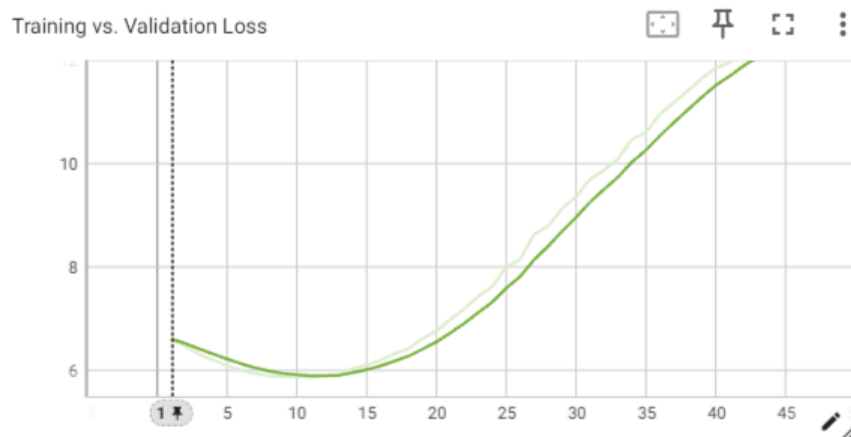


Рис. 2.14. Графік зміни втрат під час валідації selfPlay моделі

Як видно з рисунків наведених вище, приблизно після 10 епохи значення втрат почало зростати для обох моделей, що означає, що вони зіткнулися з явищем, яке відоме як «overfitting», або перенавчання, надмірне пристосування. Надмірне пристосування відбувається, коли модель машинного навчання занадто ретельно вивчає навчальні дані, фіксуючи шум і нерелевантні деталі, а не основні закономірності. Перенавчання може статися з різних причин. Зокрема, це може бути через архітектуру моделі або через наявність шуму в даних. Існують різні способи боротьби з цим явищем, такі як регуляризація, зменшення кількості ознак або рання зупинка (зупинка процесу навчання до того, як модель почне перенавчатися). Саме підхід схожий на ранню зупинку був використаний під час тренування моделей, тому найкраща отримана модель, для обох підходів була збережена.

Після навчання кожної з моделей варто перевірити їхню працездатність і провести певну кількість партій. Для цього спочатку переконаємося, що вона здатна робити коректні ходи. Щоб перевірити це, проведемо партію, де зіграємо з моделями по черзі самі. Для цього треба виконати такі дії:

- 1) Завантажити файл найкращої моделі, який було отримано при навчанні;
- 2) Створити нову дошку;
- 3) Здійснити хід (або відразу передати дошку моделі);
- 4) Передати об'єкт дошки моделі на вхід;
- 5) Відобразити хід моделі на дошці;
- 6) Повторити кроки 3-5 до закінчення гри.

Загалом це не найкращий підхід до реалізації гри з отриманими моделями, проте його достатньо для перевірки працездатності. Також це доволі гарна можливість кинути виклик власному творінню і розуміти його рівень гри в порівнянні з власним. Процес гри з однією з моделей зображено на рисунку 2.15.



Рис. 2.15. Гра з моделлю

Також можна перевірити рівень гри моделей в поєдинку з рушієм Stockfish. Зрозуміло, що отриманим моделям не вдасться обіграти цей рушій на його повній потужності, тому буде виставлено його силу гри через відповідний параметр.

Лістинг коду, відповідального за гру з рушієм Stockfish наведено на рисунку 2.16. Цей код симулює шахову партію між ШІ-агентом та рушієм Stockfish. Він ініціалізує шахову дошку та порожній список для зберігання всіх ходів, зроблених під час гри. Гра триває максимум 150 ходів, кожна ітерація представляє один хід ШІ-агента або Stockfish. Спочатку ШІ-агент робить хід, він записується на дошку, а позначення ходу додається до списку всіх ходів. Далі Stockfish оцінює поточну позицію на дошці на основі записаних ходів, робить свій хід протягом 3 секунд і оновлює дошку відповідним чином. Після завершення циклу движок Stockfish перезавантажується, і відображається остаточна позиція на дошці.

```
board = chess.Board()
allMoves = []

MAX_NUMBER_OF_MOVES = 150
for i in range(MAX_NUMBER_OF_MOVES):
    try:
        move = saved_model.predict(board)
        board.push(move)
        allMoves.append(str(move))
    except:
        print("game over. You lost")
        break

    stockfish.set_position(allMoves)
    stockfishMove = stockfish.get_best_move_time(3)
    allMoves.append(stockfishMove)
    stockfishMove = chess.Move.from_uci(stockfishMove)
    board.push(stockfishMove)

stockfish.reset_engine_parameters()

board
```

Рис. 2.16. Гра моделі з рушієм Stockfish

Вищезазначені способи перевірки моделей дають лише базове уявлення про їхні можливості. Для кращого розуміння результатів тренування слід провести більшу кількість партій і визначити приблизний ELO рейтинг для обох моделей, а також визначити, який підхід для синтезу навчальних даних дає кращі результати після навчання. Це один з найважливіших етапів дослідження. Детальний аналіз працездатності отриманих моделей буде проведено в наступному розділі.

Висновки до розділу 2

У даному розділі було досліджено різноманітні аспекти збору, синтезу та обробки даних для подальшого навчання нейронних мереж. Аналізуючи доступні джерела даних, виявлено, що для навчання можливо використовувати як реальні дані з поєдинків, так і синтетично згенеровані дані, що базуються на самостійних ігрових сценаріях.

Основна увага приділялася процесу синтезу даних, який має велике значення для створення репрезентативних та різноманітних навчальних наборів. Синтез даних на основі поєдинків та самостійних ігор одного рушія відображає реальність гри, допомагаючи моделі у вивченні різноманітних стратегій та сценаріїв.

Кодування навчальних даних виявилось ключовим етапом у підготовці даних для навчання моделей. Правильне кодування дозволяє ефективно представляти інформацію, що сприяє покращенню результатів навчання та забезпечує більшу точність прогнозування.

Після тренування моделей і практичної реалізації виявлено позитивні результати, що підтверджують ефективність обраних підходів до навчання нейронних моделей. Ретельно підготовлені дані та здійснення адаптивного тренування сприяють покращенню якості прогнозів моделі, що може мати велике значення у практичних застосуваннях. У результаті виконання цього розділу було отримано дві моделі, одна з яких була натренована на даних отриманих шляхом самостійної гри, інша ж була натренована на даних отриманих з поєдинків двох шахових рушіїв. Також було проведено базову перевірку працездатності моделей шляхом гри людини з моделлю, а також шляхом гри моделей з іншим шаховим рушієм. Наступним етапом дослідження буде визначення ефективності моделей та порівняння підходів до генерації даних.

РОЗДІЛ 3. АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

3.1 Реалізація UCI-сумісних рушіїв

Після навчання, було отримано файли моделей, які натреновані визначати наступний хід на основі поточної позиції на дошці. Проте, для того аби мати краще розуміння якості моделей і мати можливість зручної роботи з моделями в графічних шахових інтерфейсах, слід реалізувати шахові рушії, в основі яких лежатимуть отримані моделі.

Для того, щоб шаховий рушій мав можливість комунікувати з різними графічними інтерфейсами існують різні протоколи взаємодії. Один з них протокол UCI (Universal Chess Interface). Це протокол, який визначає, як шахові рушії взаємодіють з інтерфейсами користувача (UI) або графічними інтерфейсами користувача (GUI). Протокол UCI забезпечує сумісність між різними шаховими рушіями та інтерфейсами, дозволяючи їм безперешкодно працювати разом. [23]

Загалом існують різні думки про протокол UCI для шахових рушіїв, як позитивні так і негативні, проте для цілей дослідження він цілком підходить, так як він підтримується майже всіма шаховими графічними інтерфейсами, він простий в розумінні і реалізації, код для роботи з протоколом виходить простим і прямолінійним. Тому шахові рушії працюватимуть за протоколом UCI.

Для початку слід розробити клас, який надаватиме можливість зручно працювати з реалізованою моделлю. Було розроблено клас-обгортку над класом моделі, яка використовувалась для навчання.

Клас `PlayableModel` містить в собі поле `model`, яке ініціалізується екземпляром класу `Model`. В конструкторі приймається шлях до файлу моделі, далі з нього ініціалізується сама модель. Також реалізовано метод `predict`, який приймає дошку, всередині себе він звертається до моделі для того аби та зробила передбачення наступного найкращого ходу. Метод повертає список з двох елементів: хід і поточний стан дошки.

Лістинг класу-обгортки наведено на рисунку 3.1.

```

from model import Model
import torch
import chess

class PlayableModel:
    def __init__(self, path: str):
        self.model = Model()
        f = open(path, "r")
        f.readline()
        model_path = f.readline()
        f.close()
        self.model.load_state_dict(torch.load(model_path))

    def predict(self, board: chess.Board):
        move = self.model.predict(board)
        board.push(move)

    return [move, board]

```

Рис. 3.1. Клас для роботи рушія з моделлю

Наступним кроком буде реалізація класу Engine, який відповідає за обробку UCI-команд. Клас Engine представляє шаховий рушій, що реалізує протокол Universal Chess Interface (UCI). Після ініціалізації він встановлює шахову дошку і завантажує задану модель для передбачення ходів. Він обробляє команди UCI, такі як «uci», «isready», «position», «go», «ucinewgame» та «quit» за допомогою методів uci(), is_ready(), position(fen), go(), new_game() та quit(). Метод run() безперервно читає вхідні команди, обробляє їх і виконує відповідні дії. Загалом, клас надає модульний та розширюваний фреймворк для побудови та запуску шахового рушія, сумісного з інтерфейсами користувача на основі UCI або іншими системами. Лістинг методів, які реалізують UCI протокол зображено на рисунку 3.2.

```

def uci(self):
    print(f"id name {self.name.capitalize()}")
    print("id author Mykola Ukrainets")
    print("uciok")

def is_ready(self):
    print("readyok")

def position(self, fen):
    self.board.set_board_fen(fen)

def go(self):
    move, board = self.model.predict(self.board)
    self.board = board
    print("bestmove", move.uci())

def new_game(self):
    self.board = chess.Board()
    self.is_ready()

def read_move(self, move):
    self.board.push_san(move)

```

Рис. 3.2. Методи для реалізації протоколу UCI

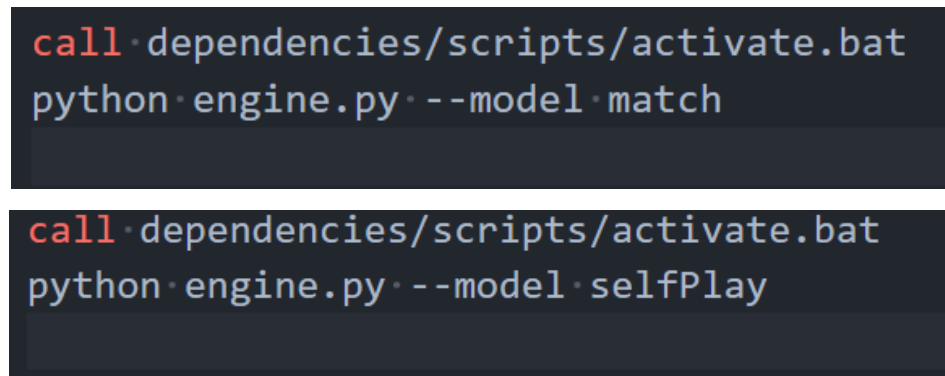
Також для зручності роботи з рушієм на різних машинах було реалізовано скрипт для встановлення всіх необхідних залежностей. Адже без низ жоден потенційний користувач не зможе використовувати шаховий рушій. Цей скрипт автоматизує створення віртуального середовища та встановлення залежностей проекту в ньому. Він визначає дві функції, `create_venv(venv_dir)` і `install_dependencies(venv_dir)`, які відповідають за створення віртуального середовища за допомогою модуля `venv` і встановлення залежностей за допомогою `pip`. Скрипт вказує директорію для віртуального середовища як `"dependencies"`, потім послідовно виконує функції створення віртуального середовища та встановлення залежностей, перелічених у файлі `requirements.txt`, разом з додатковими залежностями для `PyTorch`. Після успішного завершення він друкує інформативні повідомлення про успішне створення віртуального середовища та встановлення залежностей, забезпечуючи спрощення процесу налаштування проекту. На рисунку 3.3 зображено лістинг функції для встановлення необхідних залежностей.

```
def install_dependencies(venv_dir):
    """Install dependencies within the virtual environment."""
    try:
        subprocess.check_call(
            [
                os.path.join(venv_dir, "Scripts", "python"),
                "-m",
                "pip",
                "install",
                "-r",
                "requirements.txt",
            ]
        )
        subprocess.check_call(
            [
                os.path.join(venv_dir, "Scripts", "python"),
                "-m",
                "pip",
                "install",
                "torch",
                "torchvision",
                "torchaudio",
                "--index-url",
                "https://download.pytorch.org/whl/cu118",
            ]
        )
    except subprocess.CalledProcessError as e:
        print("Error installing dependencies:", e)
        sys.exit(1)
```

Рис. 3.3. Лістинг функції для встановлення залежностей

Після розробки основного функціоналу для реалізації рушіїв, потрібно потурбуватися про відповідні скрипти для встановлення і запуску, які використовуватимуться безпосередньо графічним шаховим інтерфейсом.

Для використання рушіями відповідної моделі під час їх запуску з командного рядку, слід передавати аргумент командного рядка з назвою `--model` та значенням «match» або «selfPlay». Значення цього аргументу визначатиме, яку саме модель завантажувати для рушія. На рисунку 3.4 зображено лістинг скриптів для запуску відповідних екземплярів рушіїв.



```
call dependencies/scripts/activate.bat
python engine.py --model match
```

```
call dependencies/scripts/activate.bat
python engine.py --model selfPlay
```

Рис. 3.4. Скрипти для запуску рушіїв

Перед запуском будь-якого з рушіїв слід в першу чергу запустити скрипт для ініціалізації віртуального середовища, в якому відбудеться встановлення всіх необхідних пакетів та бібліотек. Для цього теж було зроблено простий скрипт для командного рядка, який запускає процедуру створення середовища встановлення залежностей, яку було згадано раніше (рис. 3.3).

Після реалізації рушіїв і необхідної для інтеграції з шаховим графічним інтерфейсом інфраструктури, можна переходити безпосередньо до конфігурації рушіїв в шаховому графічному інтерфейсі та проведення дослідження ефективності.

3.2 Робота шахових рушіїв в графічному шаховому інтерфейсі

Раніше для реалізації генерації даних на основі гри двох рушіїв вже було використано версію Cute Chess для командного рядка. Проте для більшої наглядності та зручності роботи з отриманими шаховими рушіями буде використано

версію з графічним інтерфейсом. По-перше, це дасть можливість зручно проводити поодинокі ігри з рушіями в форматі «людина проти комп'ютера» та «комп'ютер проти комп'ютера», а по-друге, в графічному інтерфейсі буде зручніше налаштовувати турніри між рушіями, де вони гратимуть певну кількість ігор між собою. Це дозволить зробити висновок, який рушій виявиться сильнішим, відповідно це дасть змогу визначити кращий підхід генерації даних для тренування моделей.

На рисунку 3.5 зображено інтерфейс програми Cute Chess.

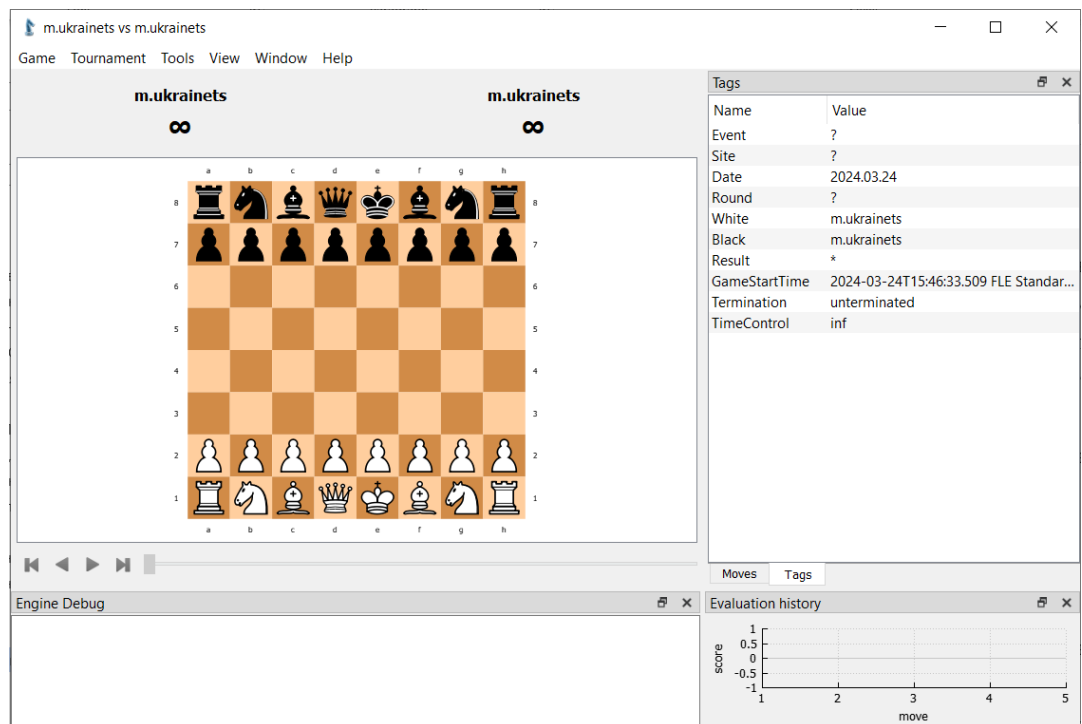


Рис. 3.5. Інтерфейс Cute Chess

Ця програма дозволяє зручно проводити ігри з шаховими рушіями, також він підтримує різні шахові протоколи, зокрема і UCI, який було реалізовано для отриманих моделей.

Для того аби мати можливість взаємодіяти з рушіями використовуючи шаховий інтерфейс, їх слід сконфігурувати. В програмі Cute Chess це робиться наступним чином. Відкриваємо вкладку Tools, обираємо пункт Settings, у вікні, що відкрилося обираємо вкладку Engines. Натискаємо кнопку Add a new engine, після чого відкриється вікно, в якому і відбувається конфігурація шахового рушія. Конфігурацію рушія selfPlay зображено на рисунку 3.6.

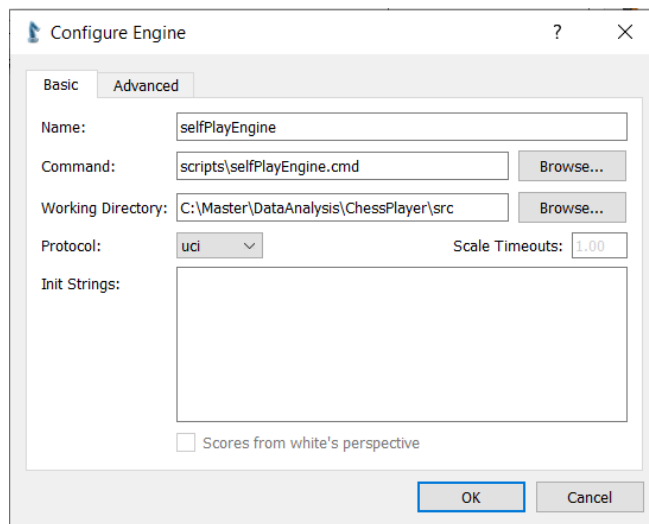


Рис. 3.6. Конфігурація рушія selfPlay в Cute Chess

Вона містить в собі назву рушія, яка буде відображатися при взаємодії з ним, шлях до команди, яка запускає рушій, а також його робочу директорію. Також слід обрати протокол, за яким працює рушій і додатково за необхідності можна задати рядки для ініціалізації.

Після конфігурації потрібно перевірити працездатність обох рушіїв в шаховому інтерфейсі, для того, щоб упевнитись, що комунікація в рамках протоколу UCI реалізована коректно. Для цього можна провести одиночні партії з кожним з рушіїв. Щоб почати гру, слід в головному вікні Cute Chess обрати вкладку Game і в ній натиснути пункт на New. Після цього відкриється вікно налаштувань партії. Основні налаштування зображено на рисунку 3.7.

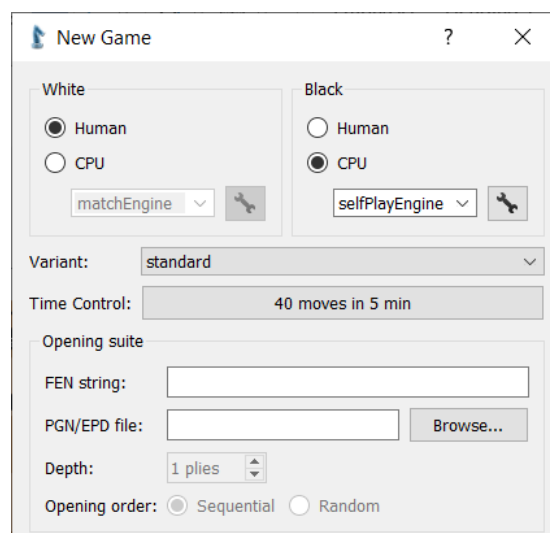


Рис. 3.7. Налаштування партії з рушієм selfPlay

З рисунку 3.7 видно, що можна задати якими фігурами гратиме людина, а якими комп'ютер, також можна обрати варіант гри (Cute Chess підтримує різні варіанти шахів, навіть такі як shogi та tictactoe), встановити контроль часу, а також задати набір дебютів. На рисунку 3.8 зображено приклад гри у форматі «людина проти комп'ютера». Варто зазначити, що варіант гри за допомогою графічного шахового інтерфейсу є набагато зручнішим ніж той, що було продемонстровано раніше, під час перевірки працездатності моделей, адже він надає велику кількість опцій для налаштування партії, через нього зручно взаємодіяти з дошкою, адже ходи робляться безпосередньо через переміщення фігур мишею, також по завершенню партії можна зручно переглянути історію ходів.



Рис. 3.8. Процес гри з рушієм selfPlay

Ще варто зауважити, що Cute Chess надає можливість для відлагодження шахових рушіїв. На рисунку 3.8 внизу можна побачити вікно Engine Debug, яке стає дуже в нагоді при зміні вихідного коду рушія і його тестуванні. В ньому виводиться те, що подається на вхід рушію і те, що видає рушій у відповідь. Це дозволяє моніторити як рушій реагує на команди протоколу UCI. Також при виникненні проблем, в коді рушія можливо додати вивід значень певних змінних і цей вивід відобразиться в цьому вікні. На рисунку 3.9 зображено комунікацію між рушієм і графічним шаховим інтерфейсом.

```
Engine Debug
>selfPlayEngine(2): position startpos moves e2e4 c7c5 g1f3 d7d6 f1c4
>selfPlayEngine(2): isready
<selfPlayEngine(2): readyok
>selfPlayEngine(2): go wtime 286605 btime 299994 movestogo 38
<selfPlayEngine(2): bestmove e7e6
>selfPlayEngine(2): position startpos moves e2e4 c7c5 g1f3 d7d6 f1c4 e7e6 f3g5
>selfPlayEngine(2): isready
<selfPlayEngine(2): readyok
>selfPlayEngine(2): go wtime -393073 btime 299992 movestogo 37
<selfPlayEngine(2): bestmove b8c6
```

Рис. 3.9. Комунікація за протоколом UCI з рушієм selfPlay

Крім можливості проводити партії в форматі «людина проти комп'ютера», Cute Chess надає можливості проведення партій в форматі «комп'ютер проти комп'ютера». Для цього при створенні нової партії слід обрати отримані рушії, визначити якими фігурами вони гратимуть. Після цього рушії запусяться і почнуть грати між собою. Результат однієї з ігор рушія match проти selfPlay зображено на рисунку 3.10.

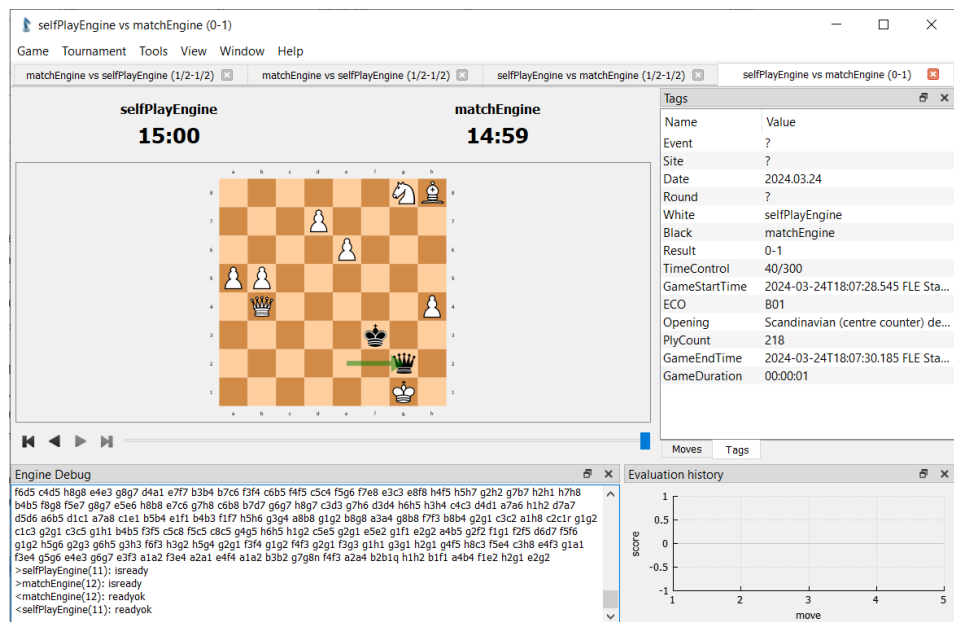


Рис. 3.10. Перемога рушія match над рушієм selfPlay

Проте робити висновки про перевагу одного рушія над іншим лише за однією чи двома партіями буде некоректно, для цього слід провести певну кількість, наприклад 1000, після якої буде можливо зібрати статистику перемог, нічий та поразок кожного рушія. На основі цієї статистики можна буде робити висновки по перевазі того чи іншого рушія.

Тому перейдемо до конфігурації турніру між рушіями. Для цього обираємо вкладку Tournament, в ній натискаємо New. Після цього з'являється вікно налаштувань турніру. Налаштування для турніру між створеними рушіями зображена на рисунку 3.11.

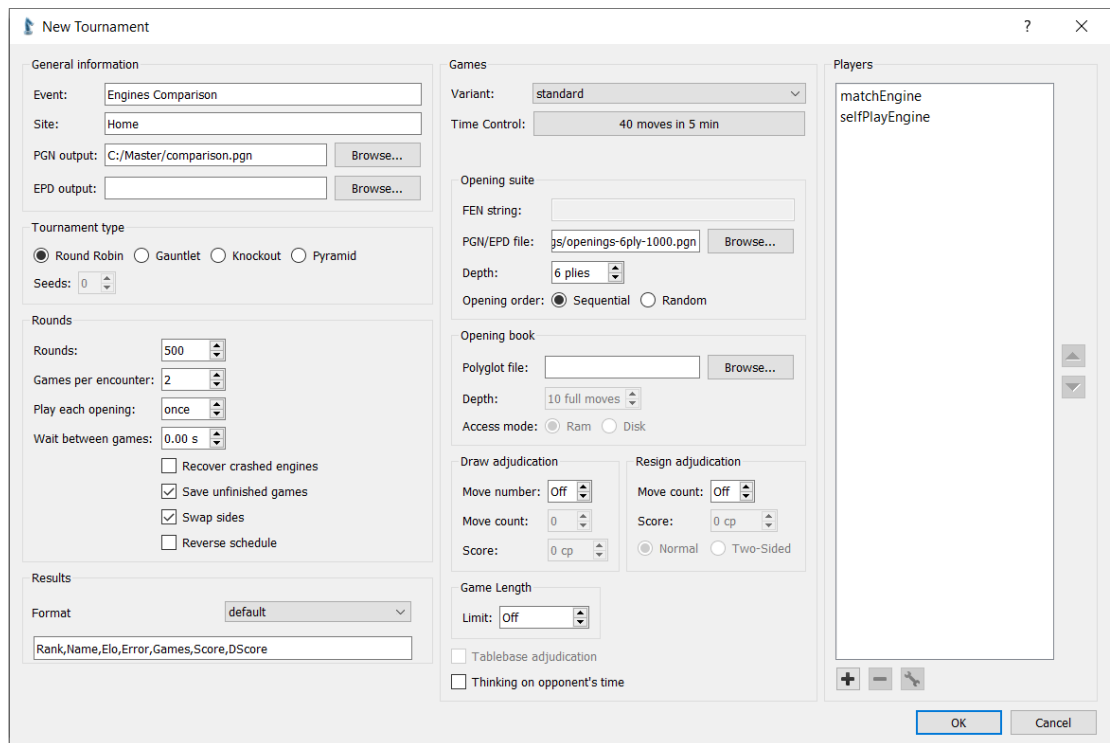
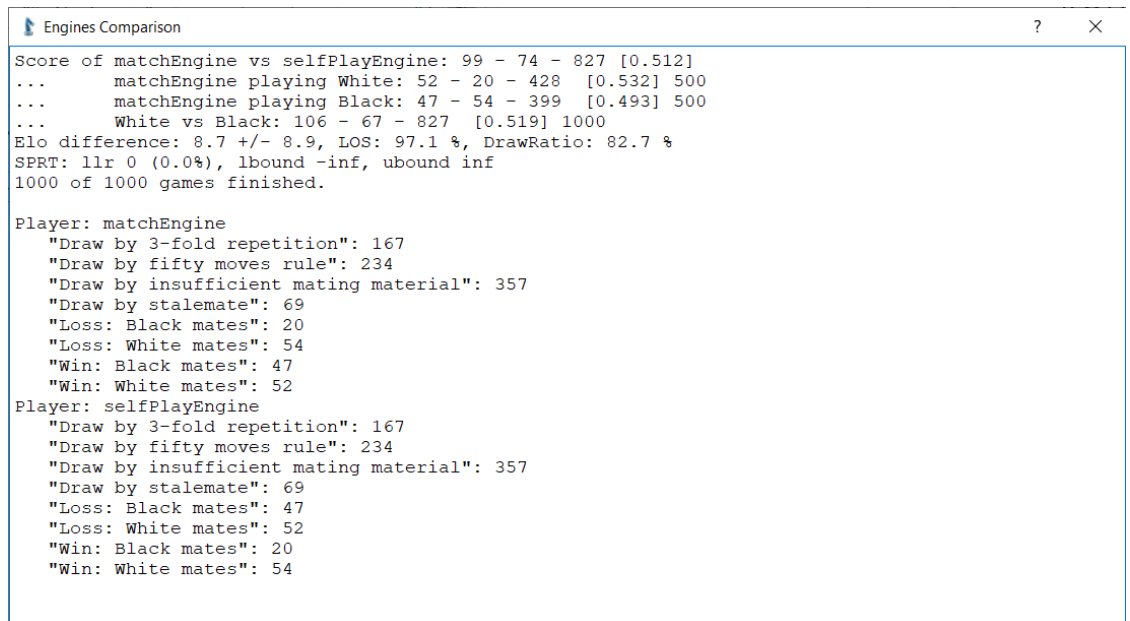


Рис. 3.11. Налаштування турніру

Розглянемо основні з них. В секції General information задаємо назву, місце проведення, а також шлях до вихідного PGN файлу, куди будуть записані усі проведенні партії. В секції Tournament type відповідно задаємо формат турніру. В цьому випадку це Round Robin, що означає, що кожен рушій гратиме з усіма іншими рушіями зі списку. Далі в секції Rounds було задано кількість раундів в 500 і по дві гри в кожному раунді, для того аби кожен з рушіїв зіграв і білими, і чорними фігурами. Доволі важливою є секція під назвою Opening suite. Вона відповідає за налаштування книги дебютів, тобто містить в собі набір ходів з початкової шахової позиції для обох гравців. Справа в тому, що якщо не використовувати книгу дебютів, то є велика ймовірність, що кожна партія між рушіями буде однаковою, адже кожен рушій завжди обиратиме один і той самий, найкращий, на його думку, хід. Книга дебютів допомагає щоразу встановлювати різну

початкову позицію, в якій кожен рушій матиме різний найкращий хід. Для нашого турніру було взято книгу дебютів одного з учасників шахової спільноти, який бере участь в розробці рушія Leela Chess Zero.

Провівши 1000 ігор з отриманими шаховими рушіями, було отримано результати, зображені на рисунку 3.12.



```
Engines Comparison
Score of matchEngine vs selfPlayEngine: 99 - 74 - 827 [0.512]
...      matchEngine playing White: 52 - 20 - 428 [0.532] 500
...      matchEngine playing Black: 47 - 54 - 399 [0.493] 500
...      White vs Black: 106 - 67 - 827 [0.519] 1000
Elo difference: 8.7 +/- 8.9, LOS: 97.1 %, DrawRatio: 82.7 %
SPRT: llr 0 (0.0%), lbound -inf, ubound inf
1000 of 1000 games finished.

Player: matchEngine
  "Draw by 3-fold repetition": 167
  "Draw by fifty moves rule": 234
  "Draw by insufficient mating material": 357
  "Draw by stalemate": 69
  "Loss: Black mates": 20
  "Loss: White mates": 54
  "Win: Black mates": 47
  "Win: White mates": 52
Player: selfPlayEngine
  "Draw by 3-fold repetition": 167
  "Draw by fifty moves rule": 234
  "Draw by insufficient mating material": 357
  "Draw by stalemate": 69
  "Loss: Black mates": 47
  "Loss: White mates": 52
  "Win: Black mates": 20
  "Win: White mates": 54
```

Рис. 3.12. Результати турніру

Як видно з рисунку 3.12, рушій match має перевагу в кількості перемог. Проте також варто зауважити, що було отримано велику кількість нічиїх, що загалом властиво під час ігор шахових рушіїв. Власне більше кількість перемог рушія match вказує на те, що його модель здатна робити кращі ходи, які приводять до перемоги через мат частіше, ніж модель рушія selfPlay. Також повертаючись назад до процесу синтезу даних, варто зауважити, що він був довшим для підходу з самотійною грою, аніж для підходу з грою двох рушіїв. Це зумовлено тим, що при генерації ігор двома рушіями, була можливість задати обмеження на кількість часу на партію та інкремент після кожного ходу. Так, зрозуміло, що чим менший час на роздуми у шахового рушія, тим більше обчислювальних ресурсів він потребуватиме, але тоді і кількість зіграних партій буде більшою. Варто також додати, що скрипт, який було реалізовано для самотійної гри рушія Stockfish, не зовсім оптимальний і не має можливості працювати з дебютами,

тому потенційним покращенням підходу з самостійною грою може бути використання Cute Chess для гри одного і того ж рушія з самим собою. Це дозволить скоротити час потрібний для отримання партій, а також дасть більше контролю над ними, зокрема можливість використовувати книги дебютів. Ще одним потенційним покращенням є вдосконалення архітектури нейронної мережі, використаної для навчання. Її можна покращити додаванням згорткових шарів, аби вона брала до уваги просторові дані і помічала більше взаємозв'язків в даних.

Варто відзначити, що, в цілому, запропонований метод створення інтелектуального шахового агента на основі синтезованих даних виявився робочим. Отримані шахові рушії здатні виконувати правильні ходи, мають розуміння правил гри. Так, рівень їхньої гри виявився невисоким, проте це зумовлено відносно невеликою кількістю даних, на яких вони навчалися. Проте тут проявляється основна перевага методу створення шахових агентів на основі синтезованих даних – маючи метод генерації партій, можна створити безкінечно великий набір навчальних даних, використовуючи різні вже існуючі шахові рушії.

Висновки до розділу 3

В даному розділі було проведено реалізацію шахових рушіїв на основі отриманих нейронних моделей. Для реалізації можливості інтеграції шахових рушіїв з графічними інтерфейсами було обрано протокол UCI. Було реалізовано усю необхідну інфраструктуру для запуску та встановлення необхідних залежностей.

Було проведено тестування роботи рушіїв у форматі «людина проти комп'ютера». Також було налаштовано і проведено турнір між отриманими моделями. Його результати показали, що рушій, котрий має в своїй основі нейронну модель, натреновану на навчальних даних отриманих шляхом поєдинку має більшу силу гри, що відповідно вказує на його переваги.

ВИСНОВКИ

Під час кваліфікаційної роботи було запропоновано метод створення інтелектуальних шахових агентів з використанням нейронних мереж а синтезованих даних. Відштовхуючись від цього, було проведено дослідження та проаналізовано декілька наукових робіт за даною тематикою. В результаті було вирішено створити та порівняти власні способи синтезу навчальних даних та провести тренування нейронних моделей.

Взявши машинне навчання за основу, було вирішено, що майбутній шаховий рушій буде реалізовано на глибинній нейронній мережі.

Крім того було розроблено два способи отримання навчальних даних. Це, зокрема, підхід з використанням самостійної гри одного рушія та підхід з використанням двох рушіїв, які проводять партії між собою.

Відштовхуючись від проведеного дослідження було спроектовано та реалізовано алгоритм навчання 2 моделей відповідно на двох різних наборах даних. Також було реалізовано взаємодію отриманих шахових рушіїв з графічним шаховим інтерфейсом Cute Chess за допомогою протоколу UCI. Було продемонстровано працездатність обох рушіїв, а також проведено порівняння, за яким рушій, який використовує дані на основі партій двох рушіїв, виявився кращим. Також було відзначено переваги і недоліки, виявлені під час реалізації методу, виділено потенційні покращення.

В підсумку можна сказати, що моделі було налаштовано правильно і тепер вони виконують поставлене завдання, а запропонований метод створення інтелектуальних шахових агентів цілком працездатний. Загалом проведене дослідження та виконану роботу можна вважати вдалим, адже було отримано два методи синтезу навчальних даних, для тренування нейронної мережі, а також визначено більш ефективний з них.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. SHANNON C. XXII. Programming a Computer for Playing Chess / CLAUDE SHANNON. // Philosophical Magazine. – 1950. – №314.
2. Gobet F. ADRIAAN DE GROOT: MARRIAGE OF TWO PASSIONS / Fernand Gobet. // ICGA Journal. – 2006. – С. 239 – 240.
3. What Is Deep Learning? [Електронний ресурс] – Режим доступу до ресурсу: <https://www.mathworks.com/discovery/deep-learning.html>.
4. Building a Chess AI with Deep Learning / [Т. Petrick, К. Lombard, Р. Nayebi та ін.]. // Georgia Tech. – 2020.
5. Elo Rating System [Електронний ресурс] – Режим доступу до ресурсу: <https://www.chess.com/terms/elo-rating-chess>.
6. What is reinforcement learning? [Електронний ресурс] – Режим доступу до ресурсу: <https://www.techtargget.com/searchenterpriseai/definition/reinforcement-learning>.
7. KALLENBERG L. MARKOV DECISION PROCESSES / LODEWIJK KALLENBERG. – 2021.
8. Using Reinforcement Learning in Chess Engines / [М. Block, М. Bader, М. Ramírez та ін.] – 2008.
9. Silver D. <https://deepmind.google/discover/blog/alphazero-shedding-new-light-on-chess-shogi-and-go/> [Електронний ресурс] / D. Silver, Т. Hubert, D. Demis. – 2018. – Режим доступу до ресурсу: <https://deepmind.google/discover/blog/alphazero-shedding-new-light-on-chess-shogi-and-go/>.
10. McGourty C. DeepMind's AlphaZero crushes chess [Електронний ресурс] / Colin McGourty. – 2017. – Режим доступу до ресурсу: <https://web.archive.org/web/20240221115154/https://web.archive.org/web/20231212055739/https://chess24.com/en/read/news/deepmind-s-alphazero-crushes-chess>.

11. How Genetic Programming (GP) Works [Электронный ресурс]. – 2007. – Режим доступа до ресурсу: <https://www.genetic-programming.org/>.
12. Aksenov P. Genetic algorithms for optimising chess position scoring / Petr Aksenov. // University of Joensuu. – 2004.
13. Tomohiko M. Optimizing the performance of GNU-chess with a genetic algorithm / M. Tomohiko, L. Schmitt. // Proceedings of the 13th International Conference on Humans and Computers. – 2010. – С. 124–131.
14. Stockfish Docs [Электронный ресурс] – Режим доступа до ресурсу: <https://disservin.github.io/stockfish-docs/stockfish-wiki/Home.html>.
15. Testing guide [Электронный ресурс]. – 2020. – Режим доступа до ресурсу: <https://github.com/LeelaChessZero/lc0/wiki/Testing-guide>.
16. Technical Explanation of Leela Chess Zero [Электронный ресурс] – Режим доступа до ресурсу: <https://lczero.org/dev/wiki/technical-explanation-of-leela-chess-zero/>.
17. Klein D. Neural Networks for Chess The magic of deep and reinforcement learning revealed / Dominik Klein. – 2022. – С. 174–182.
18. Maharaj, S.; Polson, N.; Turk, A. Chess AI: Competing Paradigms for Machine Intelligence. Entropy 2022, 24, 550. <https://doi.org/10.3390/e24040550>
19. H. Panchal, S. Mishra and V. Shrivastava, "Chess Moves Prediction using Deep Learning Neural Networks," 2021 International Conference on Advances in Computing and Communications (ICACC), Kochi, Kakkanad, India, 2021, pp. 1-6
20. Portable Game Notation [Электронный ресурс] – Режим доступа до ресурсу: <https://www.chess.com/terms/chess-pgn>.
21. Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. 2017. Imitation Learning: A Survey of Learning Methods. ACM Comput. Surv. 50, 2, Article 21 (March 2018), 35 pages. <https://doi.org/10.1145/3054912>

22. Imitation Learning by Estimating Expertise of Demonstrators / [M. Beliaev, A. Shih, S. Ermon та ін.]. // Proceedings of the 39th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR. – 2022.
23. Universal Chess Interface [Електронний ресурс] – Режим доступу до ресурсу: <https://www.chessprogramming.org/UCI>.
24. UCI protocol [Електронний ресурс]. – 2004. – Режим доступу до ресурсу: <https://www.wbec-ridderkerk.nl/html/UCIProtocol.html>.
25. Czech J, Willig M, Beyer A, Kersting K and Fürnkranz J (2020) Learning to Play the Chess Variant Crazyhouse Above World Champion Level With Deep Neural Networks and Human Data. Front. Artif. Intell. 3:24. doi: 10.3389/frai.2020.00024
26. PYTORCH DOCUMENTATION [Електронний ресурс] – Режим доступу до ресурсу: <https://pytorch.org/docs/stable/index.html>.

ДОДАТКИ

Лістинг методу mineGames

```
def mineGames(numGames: int):
    """mines numGames games of moves"""
    MAX_MOVES = 250

    for i in range(numGames):
        currentGameMoves = []
        currentGamePositions = []
        board = chess.Board()
        stockfish.set_position([])

        for i in range(MAX_MOVES):
            moves = stockfish.get_top_moves(3)
            if len(moves) == 0:
                print("game is over")
                break
            elif len(moves) == 1:
                move = moves[0]["Move"]
            elif len(moves) == 2:
                move = random.choices(moves, weights=(80, 20), k=1)[0]["Move"]
            else:
                move = random.choices(moves, weights=(80, 15, 5), k=1)[0]["Move"]

            currentGamePositions.append(stockfish.get_fen_position())
            print(move)
            print(stockfish.get_fen_position())
            board.push_san(move)
            currentGameMoves.append(move)
            stockfish.set_position(currentGameMoves)
            if checkEndCondition(board):
                print("game is over")
                break

        saveData(currentGameMoves, currentGamePositions, 'selfPlay')
```

Лістинг коду для тренування моделі

```
createBestModelFile()

bestLoss, bestModelPath = retrieveBestModelInfo()

timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
writer = SummaryWriter("runs/fashion_trainer_{}".format(timestamp))
epoch_number = 0

model = Model()
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE, momentum=MOMENTUM)
```

```

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

best_vloss = 1_000_000.0

for epoch in tqdm(range(EPOCHS)):
    if epoch_number % 5 == 0:
        print("EPOCH {}".format(epoch_number + 1))

    model.train(True)
    avg_loss = train_one_epoch(model, optimizer, loss_fn, epoch_number, writer)

    running_vloss = 0.0

    model.eval()

    with torch.no_grad():
        for i, vdata in enumerate(validation_loader):
            vinputs, vlabels = vdata
            voutputs = model(vinputs)

            vloss = loss_fn(voutputs, vlabels)
            running_vloss += vloss

    avg_vloss = running_vloss / (i + 1)

    if epoch_number % 5 == 0:
        print("LOSS train {} valid {}".format(avg_loss, avg_vloss))

    writer.add_scalars(
        "Training vs. Validation Loss",
        {"Training": avg_loss, "Validation": avg_vloss},
        epoch_number + 1,
    )
    writer.flush()

    if avg_vloss < best_vloss:
        best_vloss = avg_vloss

        if bestLoss > best_vloss:
            model_path = "savedModels/selfPlay/model_{}_{}".format(
                timestamp, epoch_number
            )
            torch.save(model.state_dict(), model_path)
            saveBestModel(best_vloss, model_path)

    epoch_number += 1

print("\n\nBEST VALIDATION LOSS FOR ALL MODELS: ", bestLoss)

```