МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 30.00.00.000 ПЗ

Група ІПм-23-3

Семків Ілля

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Семків Ілля Віталійович

(прізвище, ім'я, по батькові)

УДК 004.032.26
(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі та методи налаштування параметрів нейромереж для задач навчання

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Семків І.В.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Зікратий Сергій Вікторович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітньо-кваліфікаційний рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” Вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Семківу Іллі Віталійовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи **“**Моделі та методи налаштування параметрів нейромереж для задач навчання **”**
керівник проекту (роботи) Зікратий Сергій Вікторович, к.т.н., доцент
затверджені наказом закладу вищої освіти від **“** 22 ” листопада 2024 р. № 781\7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Дані для навчання нейромережі, алгоритми, архітектура, набір параметрів, особливості тестування.

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Теоретичні відомості про нейромережі, їх компоненти, класифікацію, моделі та застосування
2. Парадигми і алгоритми навчання нейромереж, алгоритми оптимізації, функції втрат, механізми поліпшення навчання, глибоке навчання
3. Алгоритм навчання моделі штучної нейронної мережі
4. Розробка нейромережі для навчання, опис архітектури, збір даних та тренування моделі

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Зображення схеми одношарової ШІ (рис. 1.1, ст. 12)
2. Зображення штучного нейрона (рис. 1.2, ст. 15)
3. Зображення схеми мережі трансормера (рис. 1.8, ст. 24)
4. Процес тренування моделі (рис. 3.2, ст. 82)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Перевірка на плагіат	доц. к.т.н. Вовк Р. Б.		

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури	08.11.2024	виконано
2	Аналіз алгоритмів і їх параметрів	15.11.2024	виконано
3	Пошук оптимізаторів	12.11.2024	виконано
4	Підбір даних для машинного навчання	15.11.20234	виконано
5	Формулювання вимог та алгоритмів функціонування системи	26.11.2024	виконано
6	Програмна реалізація рішення	05.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 85 с., 21 рис., 40 джерел.

Тема: Моделі та методи налаштування параметрів нейромереж для задач навчання

Об’єкт дослідження: моделі, алгоритми та параметри для навчання нейромережі гри в шахи.

Мета роботи: дослідження впливу параметрів навчання, кількості і якості даних та алгоритмів на самостійне навчання штучної нейронної мережі.

Предмет дослідження: штучні нейронні мережі, їх компоненти, моделі та застосування, оптимізатори Adam і Lookahead, функції втрат.

Результати дослідження:

Досліджено вплив параметрів навчання, кількості даних та алгоритмів на самостійне навчання штучної нейронної мережі.

Висновок:

В результаті досліджень було визначено вплив кількості даних і параметрів на якість навчання моделей нейромережі, отримано кілька моделей штучної нейронної мережі, що навчались на різній кількості вхідних даних, з різними параметрами і використанням різних оптимізаторів навчання та протестовано кожну з моделей.

ШТУЧНА НЕЙРОННА МЕРЕЖА, АЛГОРИТМИ НАВЧАННЯ, ЗАСТОСУВАННЯ НЕЙРОМЕРЕЖ, АЛГОРИТМИ ОПТИМІЗАЦІЙ, ФУНКЦІЇ ВТРАТ, ПАРАМЕТРИ НАВЧАННЯ, ADAM, LOOKAHEAD.

ANNOTATION

Master's work: 85 pages, 21 figures, 40 references.

Topic: models and Methods for Tuning Neural Network Parameters for Learning Tasks.

Object of research: models, algorithms, and parameters for training a chess-playing neural network.

Purpose: to investigate the impact of training parameters, the quantity and quality of data, and algorithms on the autonomous learning of an artificial neural network.

Subject of research: artificial neural networks, their components, models and applications, Adam and Lookahead optimizers, loss functions.

Research results:

The influence of training parameters, data quantity, and algorithms on the autonomous learning of an artificial neural network was investigated.

Conclusion:

As a result of the research, the impact of data quantity and training parameters on the quality of neural network model learning was analyzed. Several artificial neural network models were created, trained on varying amounts of input data with different parameters and using different training optimizers. Each of these models was tested.

ARTIFICIAL NEURAL NETWORK, LEARNING ALGORITHMS, NEURAL NETWORK APPLICATIONS, OPTIMIZATION ALGORITHMS, LOSS FUNCTIONS, TRAINING PARAMETERS, ADAM, LOOKAHEAD.

ЗМІСТ

Стр.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШНМ – штучна нейронна мережа

ШІ – штучний інтелект

МН – машинне навчання

ЗНМ – згорткові нейронні мережі

КНМ - конволюційні нейронні мережі

РНМ – рекурентні нейронні мережі

СКМ - Середньоквадратична похибка

САМ - Середня абсолютна похибка

ВСТУП

Актуальність роботи

Штучні нейронні мережі (ШНМ) стали невід'ємною частиною сучасної науки, техніки та медицини, демонструючи свою ефективність у випадках, коли традиційні методи не здатні забезпечити належну точність чи швидкість. Їх застосування охоплює широкий спектр задач, таких як класифікація й кластеризація образів, прогнозування, оптимізація, управління та створення інтелектуальних інформаційних систем. Саме це робить тему дослідження штучних нейронних мереж актуальною й надзвичайно важливою для подальшого розвитку сучасних технологій.

Мета і задачі дослідження

Метою представленої магістерської роботи є — дослідити принципи роботи нейронних мереж, механізми навчання, залежність результатів від якості даних, а також ключові алгоритми оптимізації та параметри, що впливають на ефективність і швидкість тренування моделі. Важливим елементом дослідження є практичне застосування цих знань для розробки моделі нейромережі, здатної грати в шахи.

Задачею виконання роботи включало задачу створення унікальної моделі, яка здатна навчитися гри в шахи, базуючись виключно на даних про зіграні партії гравців високого рівня. Особливістю підходу є те, що модель не має попередніх знань про гру. Навчання відбувається через аналіз даних, прогнозування можливих ходів і вибір оптимального. Такий підхід підкреслює важливість адаптивності моделі та її здатності до узагальнення.

Об'єктом дослідження є моделі та методи налаштувань нейромереж для потреб навчання.

Предметом дослідження є моделі і алгоритми нейромереж, для обробки задач навчання, параметри моделей та оптимізаторів.

Методи дослідження для оцінки здатності моделі до узагальнення використовувались різні тестові дані а налаштування гіперпараметрів оптимізаторів

Наукова новизна одержаних результатів

Проведено аналіз існуючих мереж і принципів їх навчання, наслідком якого було отримано модель, метод навчання якої раніше не використовувався

Практичне значення оержаних результатів

На основі проведеного дослідження було виконано аналіз якості тренування моделей нейромережі в залежності від об’єму навчальних даних, навчено моделі, які можуть продовжувати розвиватись і покращуватись.

Особистий внесок

1. Проаналізовано засоби реалізації моделей нейромереж.
2. Проведено на практиці порівняння якості навчання моделей.

Структура магістерської роботи

Магістерська робота викладена на 85 сторінках другованого тексту, який складається з вступу, трьох розділів, висновків, списку використаних джерел (40 найменувань). Робота містить 21 рисунок.

РОЗДІЛ 1

ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО НЕЙРОМЕРЕЖІ, ЇХ КОМПОНЕНТИ, КЛАСИФІКАЦІЮ, МОДЕЛІ ТА ЗАСТОСУВАННЯ.

1.1. Поняття нейромережі

Штучні нейронні мережі (ШІ) — це моделі обчислень, натхненні структурою та функціонуванням біологічними нейронними зв’язками, як у людському мозку. Вони складаються з великої кількості взаємопов'язаних одиничних елементів, які називаються нейронами, і здатні навчатися з даних, робити прогнози чи класифікації. Кожне з'єднання, як і синапси в біологічному мозку, може передавати сигнал до інших нейронів. Штучний нейрон отримує сигнали, потім обробляє їх і може сигналізувати нейронам, з якими його з'єднано. Сигнал у з'єднанні це дійсне число, а вихід кожного нейрона обчислюється деякою нелінійною функцією суми його входів. З'єднання називають ребрами. Нейрони та ребра зазвичай мають вагу, яка підлаштовується в процесі навчання. Вага збільшує або зменшує силу сигналу на з'єднанні. Нейрони можуть мати такий поріг, що сигнал надсилається лише тоді, коли сукупний сигнал перевищує цей поріг [13].

Будь-яка нейронна ШНМ – це набір нейронів і зв’язків, що їх пов’язує. В такому контексті нейрон є простою функцією з великою кількістю

Нейронні мережі навчаються (або, їх тренують) шляхом обробки прикладів, кожен з яких містить відомий «вхід» та «результат», утворюючи ймовірнісно зважені асоціації між ними, які зберігаються в структурі даних самої мережі. Тренування нейронної мережі заданим прикладом зазвичай здійснюють шляхом визначення різниці між обробленим виходом мережі (часто, передбаченням) і цільовим виходом. Ця різниця є похибкою. Потім мережа підлаштовує свої зважені асоціації відповідно до правила навчання і з використанням цього значення похибки. Послідовні підлаштовування

призведуть до виробляння нейронною мережею результатів, усе більше схожих на цільові. Після достатньої кількості цих підлаштовувань, тренування можливо припинити на основі певного критерію. Це форма керованого навчання.

Такі системи навчаються виконувати завдання, розглядаючи приклади, як правило, без програмування правил для конкретних завдань. Наприклад, у розпізнаванні зображень вони можуть навчитися встановлювати зображення, на яких зображені коти, аналізуючи приклади зображень, мічені вручну як «кіт» та «не кіт», і використовуючи результати для ідентифікування котів на інших зображеннях. Вони роблять це без будь-якого знання про котів, наприклад, що вони мають хутро, хвости, вуса та котоподібні морди. Натомість, вони автоматично породжують ідентифікаційні характеристики з прикладів, які оброблюють.

На рисунку 1.1 представлено просту візуальну схему одношарової штучної нейронної мережі.

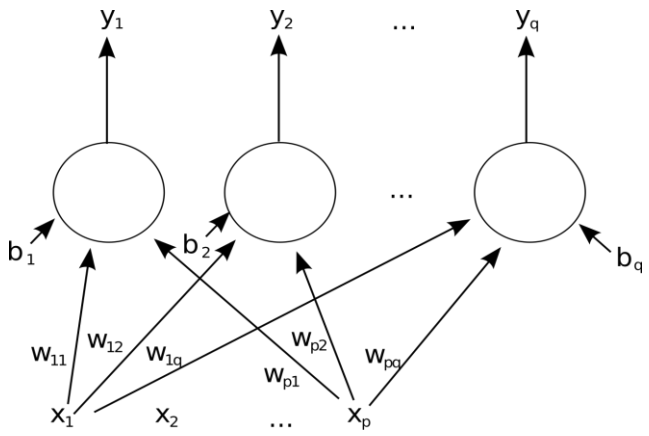

Рис. 1.1. Зображення схеми одношарової ШІ

1.2. Розвиток нейронних мереж

Ідея нейронних мереж виникла в середині XX століття, коли вчені почали шукати способи моделювати роботу людського мозку. На початку 40-

х років Уоррен МакКаллох і Уолтер Піттс запропонували першу математичну модель нейрона — "логічний нейрон". Ця модель представляла нейрон як простий пристрій, здатний виконувати базові логічні операції, такі як "і", "або" та "не" [1].

Історія нейронних мереж, як основи штучного інтелекту починається у 1943 році, коли Ернст Ізінг і Вільгельм Ленц створити і проаналізували модель Ізінга. Це модель статистичної системи, яка по суті є штучною рекурентною нейронною мережею (RNN), але без елементу навчання.

Першу найпростішу модель нейромережі – перцептрон, запропонував американський психолог Френк Розенблат. Винайдений пізніше перцептрон Румельхарта [1] відрізнявся тим, що мав метод зворотнього поширення помилки, який використовувався для зменшення помилок роботи перцептрону. Наступним етапом було створення першого БШП глибокого навчання, опублікуваного Олексієм Григоровичем Івахненком та Валентином Лапою у 1965 році під назвою «метод групового урахування аргументів» [8]. Метод заснований на селективному відборі моделей, де параметр точності моделювання збільшується на кожному кроці.

У 1969 році Марвін Мінський і Сеймур Паперт у книзі "Перцептрони" [9] показали обмеження одношарових нейронних мереж. Вони довели, що такі мережі не можуть розв'язувати деякі прості задачі, наприклад, задачу XOR. Це призвело до зниження інтересу до нейромереж, і дослідження в цій галузі було значно скорочено.

Інтерес до нейронних мереж відродився у 1980-х роках завдяки появі алгоритму зворотного поширення помилки (backpropagation). Цей алгоритм, запропонований Джеффрі Гінтоном, Румельхартом і Вільямсом, дозволив навчати багатошарові нейронні мережі. Завдяки зворотному поширенню помилки стало можливим ефективно налаштовувати ваги мережі, що значно розширило їх функціональні можливості.

У цей період також з'явилися нові архітектури нейронних мереж, такі як мережі Хопфілда та самоорганізуючі карти Кохонена. Ці моделі

продемонстрували потенціал нейронних мереж для зберігання інформації та кластеризації даних.

Також в 1980 році Куніхіко Фукусіма запропонував архітектуру згорткової нейронної мережі зі згортковими шарами та шарами пониження дискретизації, яку назвав неокогнітроном [29-30]. 1969 року він також запропонував передавальну функцію ReLU. Цей випрямляч став найпопулярнішою передавальною функцією для ЗНМ та глибоких нейронних мереж загалом. ЗНМ стали важливим інструментом комп'ютерного бачення. Також у 1980-х роках метод зворотного поширення помилок, запропонований Румельхартом і Гінтоном, став основою для навчання багатошарових нейронних мереж. Це дало можливість працювати зі складними задачами класифікації та прогнозування. З 2010-х років завдяки збільшенню обчислювальних потужностей і появі великих обсягів даних (Big Data) нейромережі стали основою багатьох технологій штучного інтелекту, таких як розпізнавання мови, зображень і тексту [10].

У 2012 році модель AlexNet виграла конкурс ImageNet, продемонструвавши значне поліпшення якості класифікації зображень. Це стало початком ери глибокого навчання. З того часу архітектури глибоких мереж, такі як ResNet, VGG та Inception, стали основними інструментами для вирішення складних задач.

Сучасний трансформер запропонував Ашиш Васвані у праці 2017 року «Увага – це все, що вам треба». Ця нейромережа не має рекурентних вузлів, натомість базується на механізмі багатоповерхової уваги, тому вимагає менше часу на тренування ніж попередні рекурентні нейромережі. Прикладом сучаної нейромережі-трансормера є мовні моделі ChatGPT, GPT-4 та BERT [31].

Сьогодні нейронні мережі застосовуються у багатьох галузях: від медицини до автоматизації бізнесу. Вони використовуються для створення самокерованих автомобілів, автоматичного перекладу, генерації зображень та

звуку. Розробка нових архітектур, таких як GPT та DALL-E, відкрила нові горизонти для творчого застосування нейромереж[31].

1.3. Основні компоненти нейромереж

Штучні нейрони – вузол ШІ, математично представлений як деяка нелінійна функція від єдиного аргументу. Ця функція називається функцією активації або функцією спрацьовування.

На рисунку 1.2 зображено схему штучного нейрона, де 1 – нейрони, які передають сигнал на вхід даного нейрона, 2 – суматор вхідних сигналів, 3 – обчислювач передавальної функції, 4 – нейрони, на вхід яких подається сигнал даного нейрону.

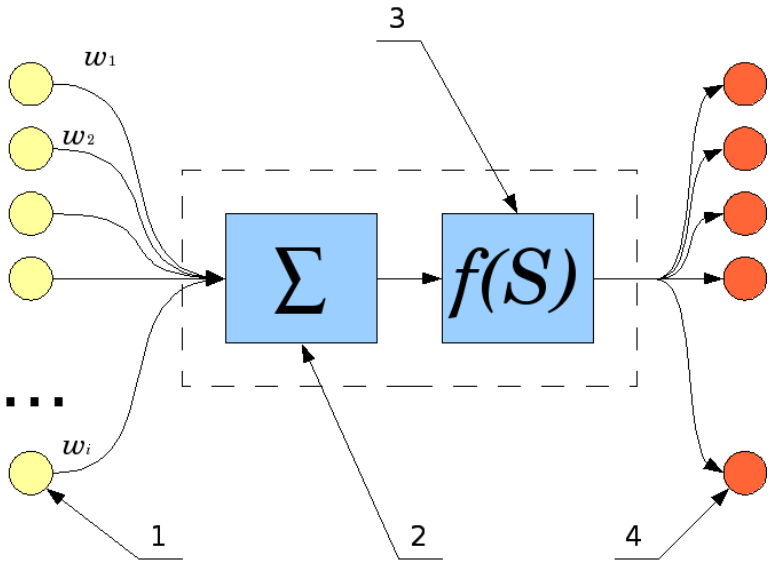

Рис. 1.2. Схема штучного нейрона.

Штучні нейрони формуються у шари, які виконую певні функії. Зазвичай нейромережі складаються з декількох шарів:

- вхідний шар – це шар нейронів, який отримує вхідні дані;
- приховані шари – шари, розміщені між вхідним і вихідним шаром, які обробляють і трансформують дані;

- вихідний шар – нейрони вихідного шару видають результат після обробки даних.

На рисунку 1.3 представлено просту візуальну схему шарів нейромережі і зв’язків між ними.

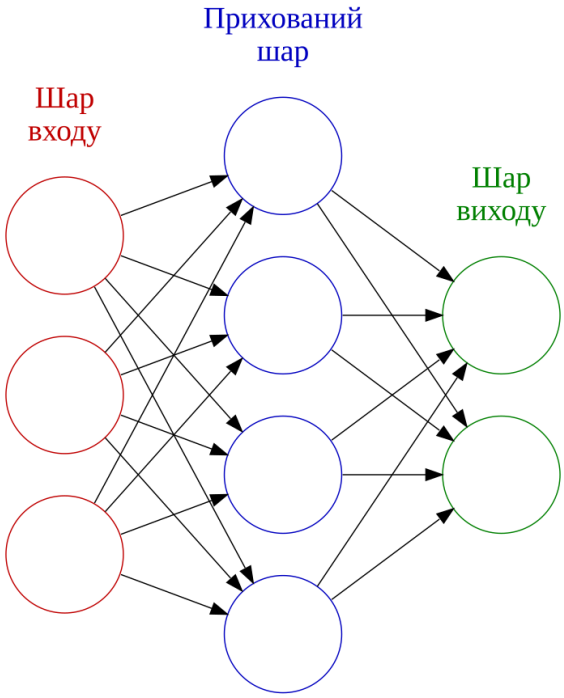

Рис. 1.3. Зображення схеми шарів нейромережі

Ваги та зсуви є ключовими параметрами нейронної мережі, які визначають, як вона обробляє дані та навчається. Вони працюють разом, щоб дозволити моделі адаптуватися до різноманітних даних і знаходити складні залежності між ними.

Вага – це числовий коефіцієнт, який визначає силу вхідного сигналу на нейрон. Кожне з’єднання між нейронами має свої вагу, яка визначає важливість вхідного сигналу для обчиснення вихідного значення нейрона. Під час навчання нейромережа коригує ваги зв’язків, щоб зменшити різницю між передбаченим і реальним виходом.

Зсув — це додатковий параметр, який додається до зваженої суми вхідних сигналів перед застосуванням активаційної функції. Він дозволяє моделі краще адаптуватись до даних.

Активаційні функції — це математичні функції, які визначають вихід нейрона після обробки його вхідних сигналів. Вони додають нелінійність у модель, що дозволяє нейромережі розв'язувати складні задачі, зокрема класифікацію, регресію, виявлення ознак тощо. Без цих функцій нейромережа могла б виконувати лінійні перетворення і була б не здатною до моделювання складних залежностей. Також ці функції впливають на активність нейронів, і залежить чи будуть його вихідні дані впливати на наступні шари [11].

Основні активаційні функції:

a) Лінійна функція (Linear Activation)

Математична формула лінійної функції має такий вигляд:

$$f(x) = x \quad , \tag{1.1}$$

де x дорівнює значенню входу помноженому на вагу і додано зсув. Перевагою даної функції є її простота. Недоліки полягають у неможливості використання для багатошарових моделей нейромереж та лінійності.

b) Сигмоїдна функція (Sigmoid)

Математична формула сигмоїдальної функції має такий вигляд:

$$f(x) = \frac{1}{1 + e^{-x}} \quad , \tag{1.2}$$

де діапазон *x*: (0, 1).

Перевагою функції є якісне виконання задач з класифікації. Недоліки полягають в тому, що при великих або малих значеннях градієнт стає дуже малим. Також обчислення відбуваються досить повільно.

c) Гіперболічний тангенс (Tanh)

Математична формула функції гіперболічного тангенсу має такий вигляд:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad (1.3)$$

де діапазон *x*: (-1, 1).

Перевагою даної функції є те що вона центрується до нуля, що зручніше для навчання. Недоліками є згасання градієнтів, як у попередній функції.

d) Функція ReLU (Rectified Linear Unit)

Математична формула функції ReLU має такий вигляд:

$$f(x) = \max(0, x), \quad (1.4)$$

де діапазон *x*: [0, ∞].

Перевагою даної функції є простота у використанні, швидкість обчислень і ефективність у вирішенні проблеми згасаючих градієнтів. Недоілком є проблема "мертвих нейронів", коли вхідні значення x<0x завжди дають f(x)=0, і нейрон перестає вчитися.

e) Leaky ReLU

Математична формула функції Leaky ReLU має такий вигляд:

$$f(x) = \begin{cases} x, & \text{якщо } x > 0 \\ \alpha x, & \text{якщо } x \leq 0 \end{cases}, \quad (1.5)$$

де діапазон *x*: (-∞, ∞), параметр α: Малий додатний коефіцієнт, наприклад, 0.01.

Перевагою є усунення проблеми мертвих нейронів.Недоліком є ускладнення моделі через додавання додаткових параметрів.

f) Exponential Linear Unit (ELU)

Математична формула функції Exponential Linear Unit має такий вигляд:

$$f(x) = \begin{cases} x, & \text{якщо } x > 0 \\ \alpha(e^x - 1), & \text{якщо } x \leq 0 \end{cases}, \quad (1.6)$$

де діапазон *x*: (-α, ∞).

Перевагою є вирішення проблеми мертвих нейронів і покращену оптимізацію. Недоліком є швидкість обчислення.

g) Swish

Математична формула функції Swish має такий вигляд:

$$f(x) = x \cdot \text{sigmoid}(x), \quad (1.7)$$

де діапазон *x*: (-∞, ∞).

Схожа до функції ReLU, але дає кращу точність за рахунок складніших обчислень.

h) Softmax

Математична формула функції Softmax має такий вигляд:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \quad (1.8)$$

де діапазон *x*: (0, 1), сума виходів = 1.

Дана функція використовується у вихідному шарі для задач багатокласової класифікації. Вона перетворює виходи у ймовірності. Може бути чутлива до великих значень, що спричиняє нестабільності.

1.4. Класифікація нейромереж та їх застосування

Існує кілька видів ШНМ, на яких базуються моделі, створені для виконання певних завдань. Далі наведені види з поясненням, коли вони використовуються:

Мережі прямого поширення (Freedforward Neural Network) – це перша і найпростіша модель нейромережі, інформація в якій буде рухатись без циклів, а лише від вхідного шару, крізь внутрішні шари і до вихідного. Найпростішою нейромережею цього типу є перцептрон (Perceptron). Використовуються для завдань класифікації, в ситуаціях де дані структуровані і мають прямі зв’язки. Мережі прямого поширення є важливими для всіх інших типів нейромереж. На рисунку 1.4 відображено схему мережі прямого поширення.

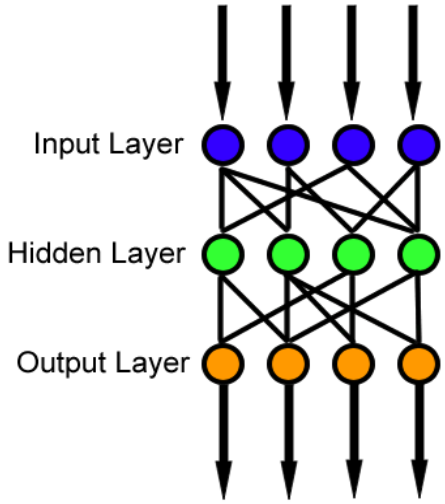

Рис. 1.4. Зображення схеми мережі прямого поширення

Конволюційні нейромережі (Convolutional Neural Network) спеціалізуються на обробці відео та зображень. Вони складаються з одного або багатьох згорткових шарів. Використовуються для розпізнавання зображень, об’єктів на них, а також для аналізу відео. Завдяки своїй здібності вираховувати просторові площини, вони досить ефективні у виконанні візуальних завдань. На рисунку 1.4 відображено схему конволюційної мережі.

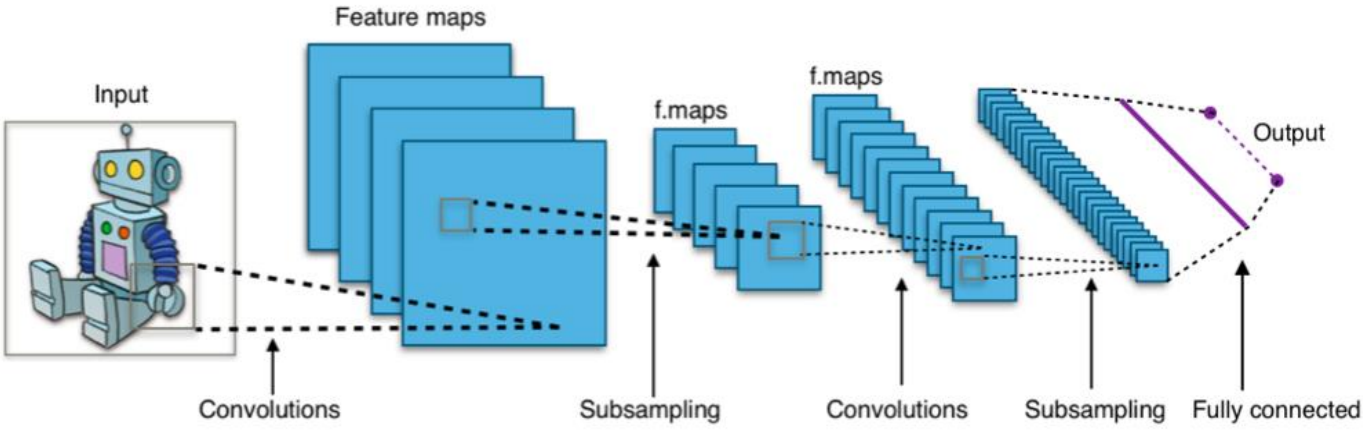

Рис. 1.5. Зображення типової схеми конволюційної мережі

Рекурентні нейромережі (Recurent Neural Network) схожі до мереж прямого поширення, але на відміну від них мають циклічні з’єднання, що дозволяє їм зберігати пам’ять про попередні етапи і враховувати їх при обробці поточних даних.

Основною архітектурою рекурентної мережі є повнорекурентна, принципом якої є з’єднання нейроподібних вузлів з кожним іншим вузлом. Вузли поділяютьс на кілька вхідних, кілька вихідних і приховані. Для завдань навчання з підкріпленням, де на мережу не здійснюється зовнішнього впливу, може застосовуватись функція нагороди, суть якої полягає у максимальному збільшенні вигоди.

Мережі рекурентного типу використовуються у обробці тексту, як генерація і переклад, розпізнання мовлення. На рисунку 1.4 відображено схему рекурентної нейромережі.

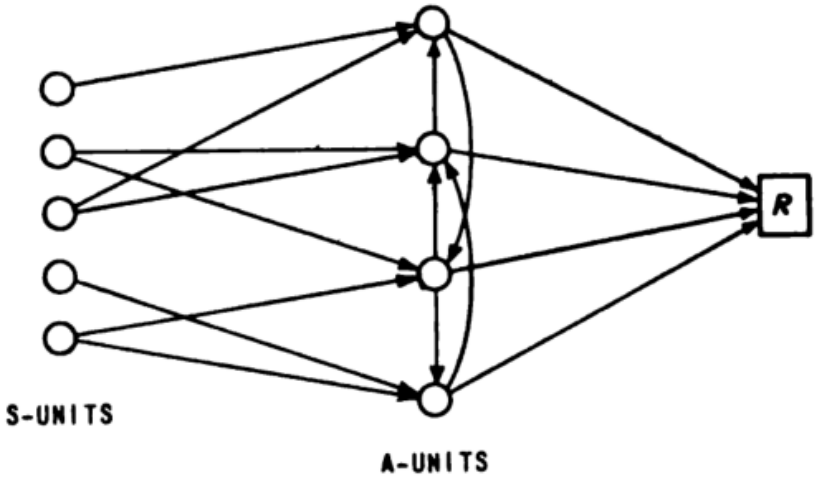

Рис. 1.6. Зображення схеми рекурентної нейромережі

Довгі короткочасні пам'яті (LSTM) створені для вирішення проблеми забуття даних, яка притаманна звичайним RNM. Це досягається за допомогою воріт, що контролюють потік інформації. Використовуються для перекладу, а також для прогнозування, завдяки можливості вловлювати залежності і послідовності. Моделі добре підходдять для виконання завданб класифікації, обробки і передбачення часових рядів, коли між подіями є часові затримки. На рисунку 1.7 приведено зображення схеми мережі довгої короткочасної пам’яті.

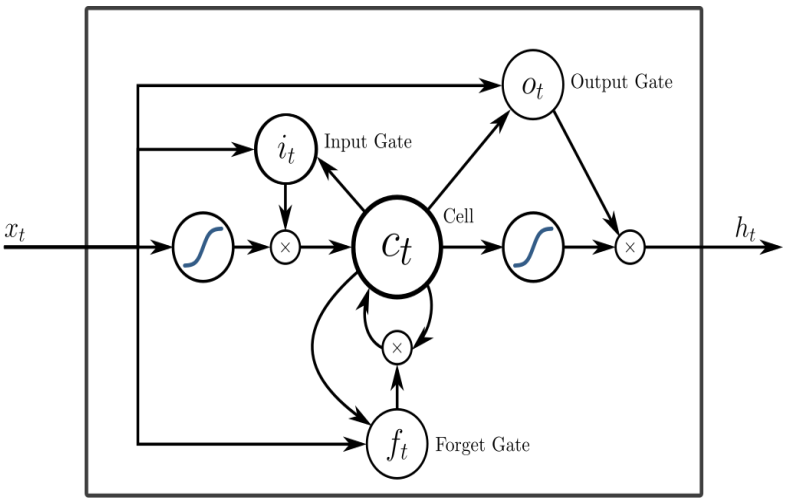

Рис. 1.7. Зображення схеми мережі довгої короткочасної пам’яті

Трансформери – це нейромережі, які покладаються на механізм уваги, для обробки вхідних даних. Вони є основою для моделей, таких як BERT, GPT, та інших великих мовних моделей. Даний тип нейромереж досяг великого успіху в обробці природної мови, що застосовується в задачах машинного перекладу. Також трансформери добре виконують завдання пов'язані з написанням тексту, розпізнавання відео, написання комп'ютерного коду та інше.

Трансформери зазвичай працюють в парадигмі самонавчання. Процес навчання складається з двох етапів. Першим етапом є попереднє тренування, яке відбувається за допомогою алгоритму некерованого навчання на базі великої кількості даних. На другому етапі відбувається донавчання, або тонке налаштування. Цей процес відбувається за допомогою алгоритму керованого, або контрольованого навчання.

ChatGPT (Gnerative Pre-trained Transformer) – це нейромережа трансформер створена компанією OpenAI. Вона має в основі мовну модель, оптимізована для розмов пригодними мовами, здатна генерувати відповіді на питання в різних галузях, при цьому враховуючи контекст розмови. На рисунку 1.8 приведено схему роботи мережі трансформера. Сучасна модель ChatGPT пройшла через кілька стадій оновлення і донавчання. Версія GPT-4 залишила позаду попередню версію моделі отримавши вищу оцінку під час проходження тестування. Також розробники багато працювали над безпекою користування моделі. Таким чином ймовірність відповіді на заборонений, шкідливий або небезпечний контент на 82% нижча, також на 40% зросла ймовірність генерації фактичних відповідей [38].

На рисунку 1.8 зображено схему роботи нейронної мереші трансформера.

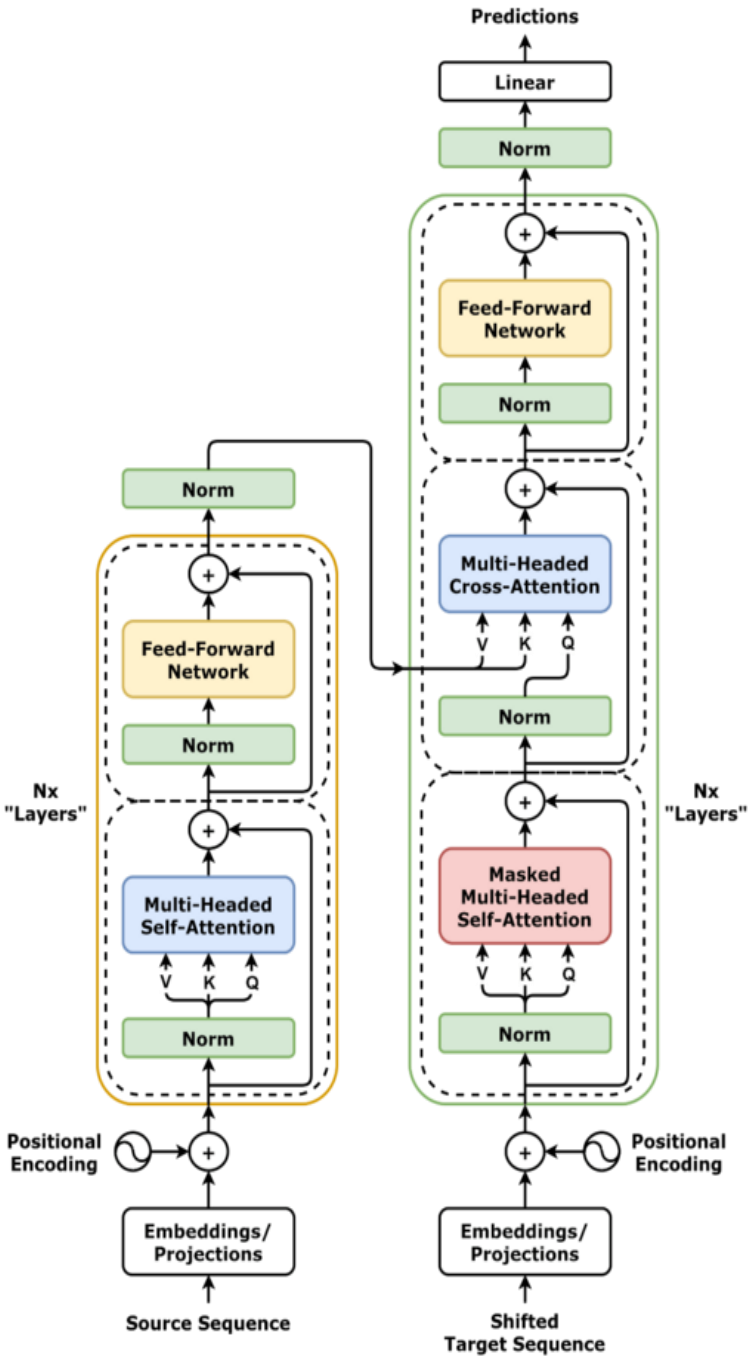

Рис. 1.8. Зображення схеми мережі трансормера

1. Генеративно-змагальні мережі (GAN) – моделі, що складаються з двох нейромереж – генератора і дискримінатора, які змагаються один з одним, створюючи нові дані, схожі на реальні (наприклад, для створення фальшивих зображень). Автокодери – моделі для навчання на стислих представленнях даних, які можуть використовуватись для стиснення, а також

для генерації нових даних. На рисунку 1.9 зображно схему генеративно-змагальної мережі.

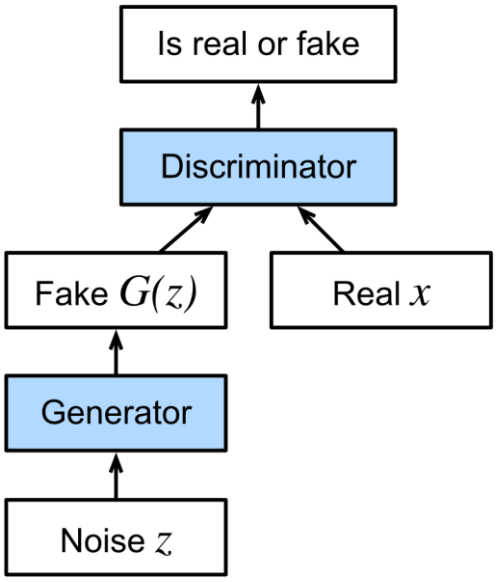

Рис. 1.9. Зображення схеми генеративно-змагальної мережі

1.5. Висновки до розділу

Під час виконання роботи У цьому розділі розглянуто основні концепції, які формують основу сучасної теорії нейронних мереж, їх архітектури та застосування. Було проаналізовано компоненти нейронних мереж, такі як ваги, зсуви та активаційні функції, які визначають їх здатність моделювати складні залежності між вхідними та вихідними даними. Важливе значення приділено активаційним функціям, які забезпечують нелінійність моделей і дозволяють нейромережам ефективно працювати з багатошаровими структурами.

Класифікація нейронних мереж охоплює широкий спектр моделей, починаючи від простих одношарових перцептронів до складних глибоких мереж. Зокрема, згадані згорткові нейромережі (CNN), що використовуються

для аналізу зображень, рекурентні мережі (RNN) для роботи з послідовними даними, та трансформери, які стали основою сучасних систем обробки природної мови.

Практичні застосування нейромереж охоплюють численні галузі, включаючи комп'ютерне бачення, обробку текстів, робототехніку, медицину та фінанси. Їх впровадження дозволяє автоматизувати складні процеси, покращувати точність прогнозів і створювати інноваційні рішення.

У підсумку, нейронні мережі є потужним інструментом сучасного штучного інтелекту, який продовжує активно розвиватися. Завдяки гнучкості та універсальності ці моделі стають невід'ємною частиною багатьох наукових і промислових проектів, відкриваючи нові горизонти для досліджень і практичного застосування.

РОЗДІЛ 2

ПАРАДИГМИ І АЛГОРИТМИ НАВЧАННЯ НЕЙРОМЕРЕЖ, АЛГОРИТМИ ОПТИМІЗАЦІЇ, ФУНКЦІЇ ВТРАТ, МЕХАНІЗМИ ПОЛІПШЕННЯ НАВЧАННЯ, ГЛИБОКЕ НАВЧАННЯ.

2.1. Парадигми і алгоритми навчання нейромереж

Машинне навчання поділяють на три основні парадигми, кожна з яких відповідає певному навчальному завданню.

Кероване, або контрольоване навчання (supervised learning) парадигма машинного навчання, в якій модель тренують набором входів і бажаних значень виходів. Алгоритм навчання змінює значення ваг, щоб зменшити різницю між передбаченими результатами і бажаними. Кероване навчання підходить для задач на розпізнавання образів, класифікацію і знаходження патернів[14].

У некерованому навчанні (unsupervised learning) вхідними даними є інформація разом з функцією витрат, функцією даних і виходу. Вибір функції витрат залежить від завдання, для якого розробляється нейромережа. Натреновані некерованим навчанням ШНМ виконують завдання оцінювання, кластерування, статистичних розподілів імовірностей, стискання інформації та її фільтрування.

Навчання з підкріпленням (reinforcement learning) це підхід, в якому актор, або діяч, вчиться приймати рішення через взаємодію з середовищем. В процесі навчання актор виконує дії, результат яких впливає на середовище і повертає діячу винагороду. Базуючись на винагороді актор змінює свою політику, задля отримання більшої винагороди, що повинно покращувати майбутні його дії.

Алгоритмом роботи навчання з підкріпленням є:

1.5. агент отримує інформацію про поточний стан середовища;

1.6. агент вибирає дію згідно своєї поточної політики;

1.7. середовище оновлює свій стан і повертає дані агенту у вигляді нагороди;

1.8. агент оновлює свою політику на основі отриманих даних, щоб покращити майбутні дії.

Самонавчання – парадигма, що була винайдена з створенням нейронної мережі, що називалась поперечним адаптивним масивом і була здатною до самонавчання. В даній парадигмі є єдиний вхід, ситуація, та єдиний вихід. Дана парадигма не має віходу для зовнішніх порад чи підкріплення з боку системи і керується виключно взаємодією між пізнанням і емоціями.

Алгоритмом самонавчання є:

- у ситуації sвиконати дію a;
- отримати наслідок ситуації s';
- обчислити емоцію перебування в наслідковій ситуації v(s');
- уточнити поперечну пам’ять w'(a,s) = w(a,s) + v(s').

2.2 Алгоритми оптимізацій

Процес оптимізації в нейромережах - це процес, під час якого відбувається налаштування ваг ШНМ для мінімізації втрат, що покращує якість передбачень моделі. Оптимізація гіперпараметрів це завдання машинного навчання з вибору множини оптимальних гіперпараметрів для алгоритму навчання ШНМ. Гіперпараметр є параметром, значення якого, на відміну від інших параметрів, використовується для керування процесом навчання. Однакові види моделей можуть мати різні обмеження, ваги або швидкості навчання для різних даних. Це і є гіперпараметри, що дозволяють

налаштовувати процес навчання моделей нейромережі. Є кілька підходів, що використовуються для навчання і оптимізації.

Пошук по ґратці – це один з підходів, суть якого в тому, що він робить повний перебір по заданій вручну підмножині простору гіперпараметрів. Цей спосіб повинен бути певною мірою продуктивним. Для порівняння використовуєтья перехресне затвердження на тренувальних даних, або оцінюванням фіксованого перевіркового набору. Пошук по ґратці страждає від прокляття розмірності, але часто легко розпаралелюється, оскільки зазвичай гіперпараметричні величини, з якими алгоритм працює, не залежать одна від одного. Алгоритм пошуку по ґратці зображено на рисунку 2.1.

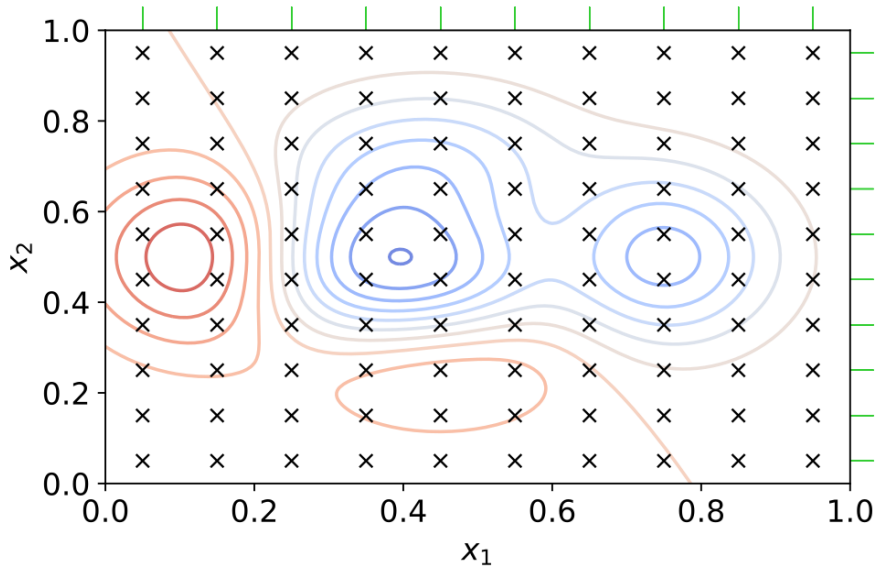

Рис. 2.1. Алгоритм пошуку по ґратці

Випадковий пошук - підхід, що замість повного перебору всіх даних, обирає випадкові дані для пошуку. метод можна узагальнити на неперервні та змішані простори. Випадковий пошук може перевершити пошук по ґратці, особливо, якщо лише мала кількість гіперпараметрів впливає на продуктивність алгоритму машинного навчання. У цьому випадку кажуть, що завдання оптимізації має низьку внутрішню розмірність. Випадковий пошук також легко паралелізується і, крім того, можливе використання

попередніх даних через вибір розподілу для вибірки випадкових параметрів. Алгоритм випадкового пошуку відображено на рисунку 2.2.

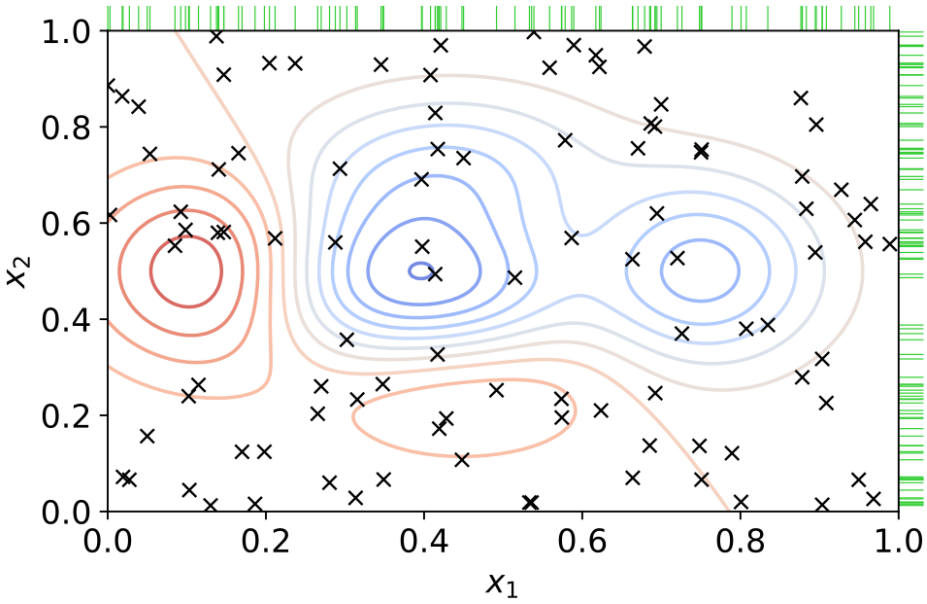

Рисунок 2.2. Алгоритм випадкового пошуку

Баєсова оптимізація — це підхід до оптимізації функцій, які є дорогими або складними для обчислення. Вона ґрунтується на побудові статистичної моделі цієї функції, найчастіше використовуючи гаусівські процеси (Gaussian Processes, GP). Замість прямого обчислення значень функції, метод намагається передбачити її поведінку та знаходить оптимальні параметри з мінімальною кількістю запитів.

Принцип роботи**:**

1. Спочатку будується сурогатна модель, яка наближає цільову функцію.
2. Використовується функція набуття (acquisition function), щоб визначити, які точки досліджувати далі. Ця функція збалансовує дослідження (exploration) нових областей і використання (exploitation) вже відомих.

3. Модель оновлюється після кожного нового обчислення цільової функції, покращуючи свою точність.

Баєсова оптимізація часто застосовується для налаштування гіперпараметрів моделей машинного навчання, особливо якщо їхнє навчання є тривалим процесом. Вона дозволяє ефективно знайти оптимальні параметри навіть за обмеженої кількості спроб.

В практичному виконанні байєсівська оптимізація показує кращі результати з меншими обчисленнями в порівнянні з пошуком по ґратці та випадковим пошуком завдяки можливості судження про якість експериментів ще до їх виконання. На рисунку 2.2 зображено алгоритм байєсової оптимізації.

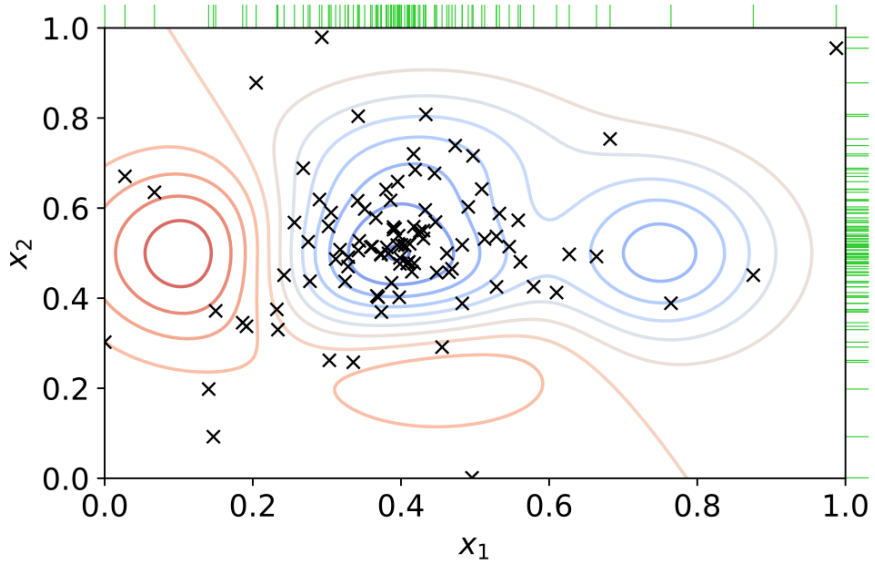

Рис. 2.2. Алгоритм баєсової оптимізації

2.3. Функції втрат

Функції втрат (loss functions) це математичні вирази, що оцінюють точність передбачень ШНМ в зрівнянні з реальними значеннями. Такі функції використовують для мінімізації помилок моделі шляхом оновлення

ваг її з’єднань під час процесу навчання. Основні функції втрат можна поділити на кілька категорій в залежності від типу виконуваної задачі.

2.3.1. Функції втрат для задач регресії

Середньоквадратична похибка (Mean squared error) — це функція, що обчислює середнє квадратичне відхилення між передбачуваними і реальними значеннями. Вона чутлива до великих помилок, оскільки, через її особливості, вплив таких помилок квадратично зростає. Майже завжди дана функція є додатною, що виходить завдяки випадковості, абе через те, що оцінювач не враховує інформацію, що могла б давати точнішу оцінку.

Середньоквадратична похибка використовується для оцінювання якості функції передбачення, яка відображає довільні входи до вибірки значень випадкової величини, або функції оцінювання, що відображає вибірку даних до оцінки.

Формула функції mean squared error (MSE):

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 , \qquad (2.1)$$

де Y_i - справжнє значення цільової змінної для і-го зразка, $\hat{Y}_i$ - передбачене значення моделі, N - кількість зразків у вибірці.

Середня абсолютна похибка (mean absolute error) (MAE) функція, що розраховує середнє абсолютне відхилення між передбачуваними і реальними значеннями. Дана функція, завдяки своїй структурі менше піддається впливу аномальних значень в порівнянні з MSE.

Формула функції mean absolute error (MAE):

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{Y}_i - Y_i| , \qquad (2.2)$$

де Y_i - справжнє значення цільової змінної для і-го зразка, $\hat{Y}_i$ - передбачене значення моделі, N - кількість зразків у вибірці.

Функція Huber loss запропонована шведським статистом Петером Джостом Хубером. Вона поєднує в собі властивості попередніх функцій разом зі зменшенням впливу помилок на вихідні значення. Для малих значень функція набуває вигляду квадратичної функції, а для великих — лінійної, з однаковими значеннями різних секцій. Завдяки цьому добре працює з даними, в яких присутні шуми.

Формула функції huber loss:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \le \delta, \\ \delta \cdot \left(|y - f(x)| - \frac{1}{2}\delta\right), & \text{otherwise.} \end{cases} \quad , \qquad (2.3)$$

де δ — поріг, що визначає перехід між квадратичною та лінійною областями, Y_i - справжнє значення цільової змінної для і-го зразка, $\hat{Y}_i$ - передбачене значення моделі, N - кількість зразків у вибірці.

2.3.2. Функції для задач класифікації

Binary cross-entropy (log loss) функція що використовується для бінарної класифікації даних. Вона оцінює різницю між передбаченими ймовірностями і фактичними класами.

Формула функції huber loss:

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i)) \quad , \qquad (2.4)$$

де $y_i \in \{0,1\}$ - справжня мітка класу (0 або 1), p(yi) - передбачена ймовірність для класу 1, 1-p(yi) – передбачена ймовірність для класу 0.

Categorical cross-entropy використовується для багатокласової класифікації. Принцип роботи полягає в обчисленні відстані між реальним і передбаченим розподілом ймовірностей.

Формула функції categorical cross-entropy:

$$L(y, \hat{y}) = \sum_{i=1}^{c} y_i \log(\hat{y}_i) \quad , \qquad (2.5)$$

де $y_i \in \{0,1\}$ - справжня мітка класу (0 або 1), p(yi) - передбачена ймовірність для класу 1, 1-p(yi) – передбачена ймовірність для класу 0.

Hinge loss використовується в задачах класифікації даних для моделей що підтримують векторний тип даних. Принцип роботи полягає в тому, що вона шукає відстані між класифікатором і найближчими точками.

Формула функції hinge loss:

$$\ell(y) = \max(0, 1 - t \cdot y) \quad , \qquad (2.6)$$

де $t = \pm 1$, y – оцінка класифікатора.

2.3.3. Спеціалізовані функції втрат

Kullback-leibler divergence функція, що застосовується для вимірювання різниці мож двома розподілами ймовірностей. Це несиметрична функція, яка вимірює відносну ентропію або відмінності в інформації, пердставленої в двох різних джерелах. Дана функція використовується в задачах моделювання розподілів, у варіаційних автоенкодерах.

Формула функції kullback-leibler divergence:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x)\ \log\left(\frac{P(x)}{Q(x)}\right).$$, (2.7)

Custom loss functions - це функції, які створюється під конкретні задачі, як наприклад функція F1-score, що створена для задач оптимізації метрик.

Triplet loss – це функція, що використовується в задачах навчання із зразками, як наприклад зображення, задля розпізнавання осіб. Принцип роботиполягає в стимулюванні моделі до скорочення відстані між подібними зразками та збільшенням відстані між різними.

Отже вибір функції втрат для нейромережі залежить від завдання, яке вона повинна буде виконати, типів даних, з якими вона буде працювати і цільової метрики. Для прикладу у розв’язанні задач регресії часто застосовується функція втрат MSE, а для класифікації — cross-entropy. Для складніших завдань створюються спеціалізовані функції втрат.

2.4. Механізм глибокого навчання

Глибоке навчання - це підгалузь машинного навчання, яка використовує штучні нейронні мережі для моделювання складних закономірностей у даних. Відмінною рисою глибокого навчання є багатошарові структури, де кожен шар обробляє інформацію й передає її на наступний. Ця методологія дозволяє нейронним мережам вивчати як прості, так і складні функції, починаючи від розпізнавання об’єктів на зображеннях і закінчуючи мовними моделями.

Глибина ШНМ визначається кількістю внутрішніх шарів, які обробляють вхідні дані.

– Мілка мережа (2-3 шари) добре працює на простих завданнях.

– Глибока мережа (10+ шарів) може розпізнавати складні об'єкти чи закономірності у великих даних.

Глибокі нейронні мережі складаються з таких основних компонентів:

– Вхідний шар: отримує сирі дані, наприклад, пікселі зображень у вигляді матриці чисел.

– Приховані шари: складаються з набору нейронів, які виконують лінійні й нелінійні перетворення над даними. Ці шари є ключовими для витягування характеристик (features) даних.

– Вихідний шар: генерує кінцевий результат, наприклад, класифікацію зображення чи передбачення числового значення.

Глибокі мережі використовують функції активації, такі як ReLU (Rectified Linear Unit), сигмоїд чи тангенс, щоб вводити нелінійність у модель і дозволяти їй знаходити складні залежності між входами і виходами.

Для прикладу взято опис аналізу зображення під час проходження даних через шари нейромережі. Вхідним шаром подається зображення у вигляді тривимірного масиву: ширина х висота х кількість каналів (наприклад, RGB-канали).

1. Перші шари (низькорівнева абстракція):
   1.1. Перші згорткові шари (Convolutional Layers) витягують прості характеристики, такі як краї, кути та текстури.
   1.2. Використовуються фільтри (kernel), які проходять по зображенню, виділяючи локальні особливості.
2. Середні шари (середньорівнева абстракція):
   2.1. Ці шари комбінують низькорівневі характеристики для створення більш складних структур, таких як форми облич чи контури об'єктів.
   2.2. Pooling-шари (максимальне чи середнє підвибірковування) зменшують розмірність даних, що знижує обчислювальні витрати і виділяє найбільш значущі характеристики.
3. Вищі шари (високорівнева абстракція):

3.1. Вони ідентифікують специфічні особливості об'єкта, наприклад, очі, ніс чи рот для розпізнавання облич.

3.2. Дані стають більш абстрактними, втрачаючи зв'язок із початковим виглядом зображення.

4. Вихідний шар:

4.1. Подає кінцевий результат, наприклад, вірогідність належності до певного класу (обличчя або не обличчя).

На рисунку 2.3 зображено приклад подання зображень в глибокому навчанні.

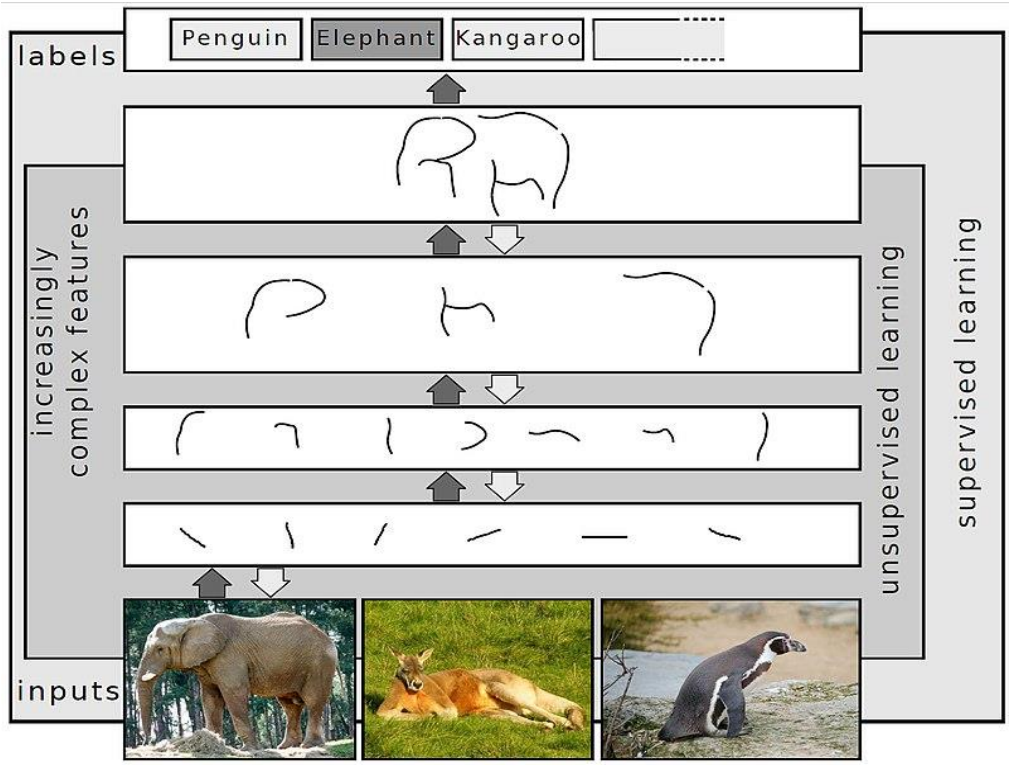

Рис. 2.3. Подання зображень на шарах абстракції в глибокому навчанні.

Глибоке навчання - це інструмент, здатний вирішувати складні задачі. Хоча воно має свої виклики, завдяки прогресу у алгоритмах, доступності даних і обчислювальних ресурсах можливий подальший розвиток алгоритмів навчання.

Для покращення навчання ШНМ варто використовувати різні набори даних. При цьому об’єм даних повинен бути значним, особливо для архітектур, які навчаються самостійно і на які не впливає тренер чи система напряму. Разом з цим важливо підібрати правильні параметри , як наприклад швидкість навчання та розмір кроку для одних моделей, або швидкість оновлення даних і розмір їх впливу на модель для інших.

2.5. Висновки до розділу

У розділі були розглянуті основні аспекти навчання нейронних мереж, що забезпечують їхню ефективність та адаптивність у вирішенні складних завдань.

Навчання нейронних мереж поділяється на три основні типи:

– Кероване навчання передбачає використання набору мічених даних. Мережа вивчає залежність між входами й відповідями, оптимізуючи свої параметри для мінімізації функції втрат.

– Некероване навчання застосовується для роботи з немічеными даними. Ця парадигма використовується для виявлення закономірностей, зокрема у задачах кластеризації, зменшення розмірності або генеративного моделювання.

– Навчання з підкріпленням характеризується тим, що модель навчається шляхом взаємодії з середовищем. Вона отримує нагороду або покарання залежно від своїх дій, оптимізуючи політику прийняття рішень.

Кожна з парадигм має специфічні сфери застосування й поєднується з іншими залежно від характеру завдання.

Описані алгоритми оптимізації, серед яких градієнтний спуск та його варіації (Adam, RMSprop, SGD), стали ключовими інструментами для налаштування параметрів моделей. Ці алгоритми спрямовані на мінімізацію функцій втрат, що є основою процесу навчання. Функції втрат, такі як

середньоквадратична помилка, крос-ентропія та Huber Loss, дозволяють кількісно оцінити, наскільки передбачення моделі відхиляються від реальних значень, та служать основою для корекції ваг.

Механізми покращення навчання, включаючи нормалізацію, регуляризацію, використання dropout і техніки ранньої зупинки, допомагають уникнути перенавчання та забезпечують стабільність і узагальнення моделі. Важливість глибокого навчання як підходу, що використовує багатошарові нейронні мережі для обробки великих обсягів даних, продемонстрована у складних завданнях, таких як комп’ютерне бачення та обробка природної мови.

У підсумку, розуміння цих аспектів дозволяє створювати ефективні нейронні мережі, оптимізувати їхню роботу та покращувати результати на практичних задачах. Сукупність цих методів і підходів визначає сучасний стан розвитку машинного навчання та відкриває нові можливості для подальших досліджень.

РОЗДІЛ 3

РОЗРОБКА НЕЙРОМЕРЕЖІ ДЛЯ НАВЧАННЯ, ОПИС АРХІТЕКТУРИ, ЗБІР ДАНИХ ТА ТРЕНУВАННЯ МОДЕЛІ.

3.1. Постановка задачі і визначення вхідних даних

В якості демонстрації навчання ШНМ було вирішено реалізувати нейромережу, яка зможе грати в шахи. Для кращої оцінки навчання моделей, було вибрано різну кількість даних для навчання, параметрів, а також різні оптимізатори. Навчені моделі будуть передбачати найкращий хід в поточний момент базуючись на опрацьованих даних.

Вхідними даними для навчання були тисячі шахових партій, зіграних людьми на шахових турнірах, які є у вільному доступі. Вибирались ігри гравців з рейтингом ELO близько двох тисяч, що вважається високим показником майстерності гравців.

Записи ходів у іграх були завантажені з сайту pgnmentor, де ходи записувались за допомогою стандартної системи, що називається шахова нотація. На рисунку 3.1 приведено приклад такого запису відкритого за допомогою додатку блокнот.

```
1.c4 e6 2.Nc3 Nf6 3.e4 d5 4.e5 Nfd7 5.d4 dxc4 6.Bxc4 c5 7.Nf3 cxd4 8.Qxd4 Nc6
9.Qe4 Qa5 10.Bf4 Bb4 11.O-O Bxc3 12.bxc3 Qxc3 13.Bd2 Qa3 14.Qg4 Qf8 15.Bb5 h5
16.Qa4 Ndb8 17.Bb4 Qg8 18.Rfd1 f6 19.exf6 gxf6 20.Bc3 Qg7 21.Qf4 O-O 22.Qh4 Qg6
23.Rd3 Kf7 24.Rad1 Ne7 25.Nd4 Qg4 26.Qxg4 hxg4 27.h3 gxh3 28.Rxh3 Rg8 29.Re1 Nd5
30.Bb2 Rxg2+ 31.Kxg2 Nf4+ 32.Kg3 Nxh3 33.Rc1 Nc6 34.Bxc6 bxc6 35.Kxh3 e5+
36.Kg3 exd4 37.Bxd4 Bd7 38.Rh1 Rg8+ 39.Kf4 Rg4+ 40.Ke3 a6 41.Rh7+ Ke6 42.Rh6 Kd5
43.Bxf6 Ra4 44.Rh5+ Ke6 45.Bc3 Rxa2 46.Ra5 Rxa5 47.Bxa5 Kd5 48.Kd3 Bf5+ 49.Ke3 Kc5
50.Bd8 Kb5 51.Be7 Be6 52.f4 c5 53.f5 Bxf5 54.Bxc5  1/2-1/2
```

Рис. 3.1. Приклад даних для навчання.

Також для виконання роботи потрібно встановити бібліотеки, за допомогою яких буде відбуватись реалізація програмного коду.

NumPy - розширення мови Python, що додає підтримку великих багатовимірних масивів і матриць, разом з великою бібліотекою високорівневих математичних функцій для операцій з цими масивами. Попередник NumPy, Numeric, був спочатку створений Jim Hugunin. NumPy — відкрите програмне забезпечення і має багато розробників. На рисунку 3.2 приведено приклад процесу встановлення бібліотеки.

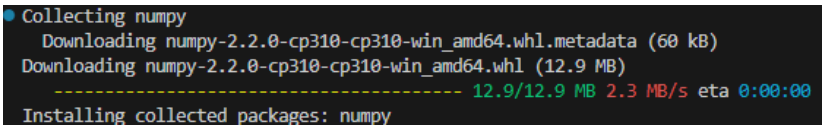

Рис. 3.2. Встановлення бібліотеки NumPy.

TensorFlow - відкрита бібліотека для машинного навчання цілій низці задач, розроблена компанією Google для задоволення її потреб у системах, здатних будувати та тренувати нейронні мережі для виявляння та розшифровування образів та кореляцій, аналогічно до навчання й розуміння, які застосовують люди. Її наразі застосовують як для досліджень, так і для розробки продуктів Google, часто замінюючи на його ролі її закритого попередника, DistBelief. TensorFlow було початково розроблено командою Google Brain для внутрішнього використання в Google, поки її не було випущено під відкритою ліцензією Apache 2.0 з 9 листопада 2015 року. На рисунку 3.3 приведено приклад процесу встановлення бібліотеки.

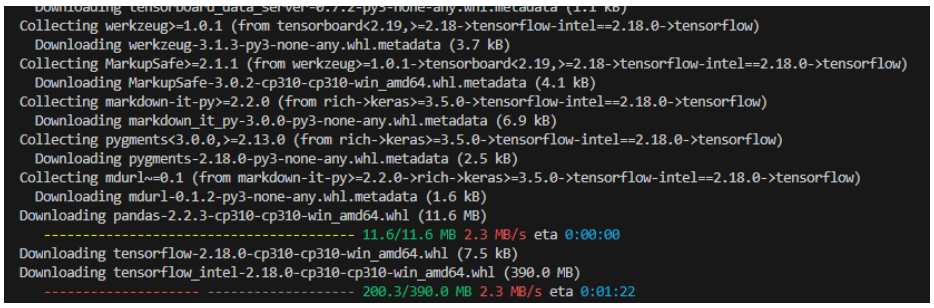

Рис. 3.3. Встановлення бібліотеки TensorFlow.

Pylint – це інструмент статичного аналізу коду для мови програмування Python. ЇЇ назва утворена з стандартного префіксу Py і посиланням на аналогічний інструмент аналізу коду lint мови програмування C. Його особливості в порівнянні з сходими інструментами в наступному:

- перевірка довжини кожного рядку коду;
- перевірка відповідності назв змінних до стандартів проєкту;
- перевірка імплементації інтерфейсів.

Інструмент відповідає стилю, рекомендованому інструкції стилів для коду Python PEP 8. На рисунку 3.4 приведено приклад процесу встановлення бібліотеки.

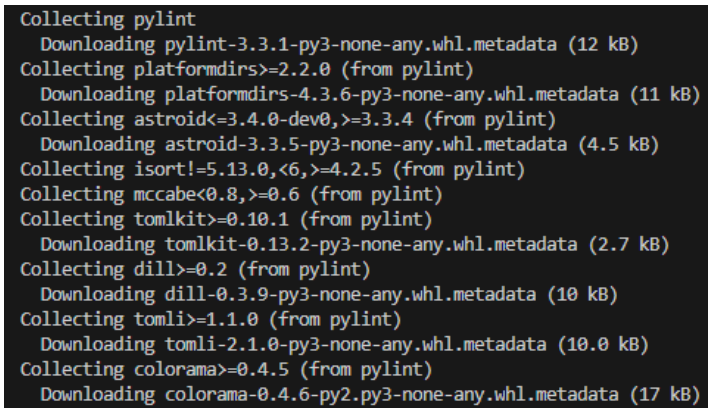

Рис. 3.4. Встановлення бібліотеки Pylint.

python-chess - програмна бібліотека для гри в шахи, створена Нікласом Фієкасом, написана мовою Python і випущена під ліцензією GPL v3. Метою було створити просту і відносно високорівневу бібліотеку, з допомогою якої полегшити реалізацію механізмів, що застосовуються для гри в шахи. На рисунку 3.5 приведено приклад процесу встановлення бібліотеки.

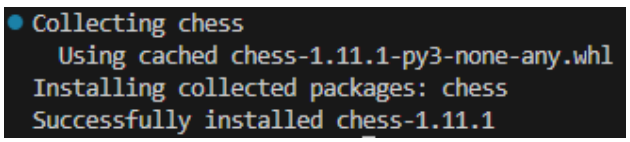

Рис. 3.5. Встановлення бібліотеки Pylint.

3.2. Підбір алгоритмів і їх параметри

Важливим етапом в тренуванні нейромереж є вибір алгоритмів, так як саме від них залежить швидкість і стабільність навчання, а також на якість кінцевого результату. Способом навчання для виконання задачі було обрано алгоритм градієнтного спуску, принципом роботи якого є мінімізація втрат шляхом поступового оновлення параметрів моделі.

Оптимізатором для моделей нейромережі було вибрано Adam, який є реалізацією алгоритму градієнтного спуску. Він комбінує ідеї адаптивних методів з методами моментів, адаптує швидкість навчання. Оптимізатор реалізовано за допомогою бібліотеки Tensorflow, що спростить роботу. В лістингу 3.1 приведено код оптимізатора.

Лістинг 3.1

```
class Adam(Optimizer):
    def __init__(self, learning_rate=0.001, beta_1=0.9, beta_2=0.999,
epsilon=1e-7, name="Adam", **kwargs):
        super(Adam, self).__init__(name, **kwargs)
        self.learning_rate = learning_rate
        self.beta_1 = beta_1
        self.beta_2 = beta_2
        self.epsilon = epsilon

    def _create_slots(self, var_list):
        for var in var_list:
            self.add_slot(var, "m")
            self.add_slot(var, "v")

    def apply_gradients(self, grads_and_vars, name=None,
experimental_aggregate_gradients=True):
        for grad, var in grads_and_vars:
            m = self.get_slot(var, "m")
            v = self.get_slot(var, "v")

            m.assign(self.beta_1 * m + (1.0 - self.beta_1) * grad)
            v.assign(self.beta_2 * v + (1.0 - self.beta_2) * tf.square(grad))

            m_hat = m / (1.0 - self.beta_1)
            v_hat = v / (1.0 - self.beta_2)

            var.assign_sub(self.learning_rate * m_hat / (tf.sqrt(v_hat) +
self.epsilon))
        return grads_and_vars
```

Швидкість навчання (learning rate) – визначає крок, на який змінюються параметри моделі після кожного оновлення. Зазвичай початкова швидкість знаходиться в межах від однієї тисячної до однієї десятитисячної.

Функція втрат (loss function), яку потрібно мінімізувати під час навчання. Тут використовується функція sparce_categorical_crossentropy, яка добре працює з задачами класифікацій. В лістингу 3.2 приведено програмний код функції.

Лістинг 3.2

```
def sparse_categorical_crossentropy(y_true, y_pred):
    loss = tf.reduce_mean(
        tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y_true,
logits=y_pred)
    )
    return loss
```

Функція метрики (metrics) визначає список метрик, які будуть використовуватись для оцінки продуктивності моделі під час навчання. Параметр accuracy це метрика точності, яка визначає відстоток правильних передбачень і вимірюється як кількість правильних передбачень поділена на загальну кількість прикладів. В лістингу 3.3 приведено програмний код функції.

Лістинг 3.3

```
def accuracy(y_true, y_pred):
    correct_predictions = tf.equal(tf.argmax(y_pred, axis=1), y_true)
    return tf.reduce_mean(tf.cast(correct_predictions, tf.float32))
```

Також в процесі роботи було прийнято рішення використати інший оптимізатор, під назвою Lookahead. Його ідея полягає у використанні швидких параметрів, які оновлюються частіше і повільних параметрів, які оновлюються повільніше, але більше впливають на остаточний стан моделі. Спочатку він оновлює швидкі дані за допомогою іншого оптимізатора, після чого швидкі параметри підтягуються до повільних.

Переметри оптимізатора Lookahead:

- внутрішній оптимізатор (inner_optimizer);
- період синхронізації (sync_period), який позначає кількість ітерацій оновлення швидких параметрів перед оновленням повільних параметрів;
- повільний крок (slow_step) визначає наскільки швидкі параметри можуть впливати на повільні.

Перевагами Lookahead є стабільність завдяки повільним оновленням параметрів і гнучкість в роботі з іншими оптимізаторами.

В лістингу 3.4 приведено приклад реалізації оптимізатора Lookahead в бібліотеці Tensorflow.

Лістинг 3.4

```
import tensorflow as tf

class Lookahead(tf.keras.optimizers.Optimizer):
    def __init__(self, inner_optimizer, sync_period=5, slow_step=0.5,
name="Lookahead", **kwargs):
        super(Lookahead, self).__init__(name, **kwargs)
        self.inner_optimizer = inner_optimizer
        self.sync_period = sync_period
        self.slow_step = slow_step
        self.counter = 0
    def apply_gradients(self, grads_and_vars, name=None,
experimental_aggregate_gradients=True):
        if not hasattr(self, 'slow_weights'):
            self.slow_weights = [tf.Variable(var, trainable=False) for _, var
in grads_and_vars]

        self.inner_optimizer.apply_gradients(grads_and_vars)
        self.counter += 1

        if self.counter % self.sync_period == 0:  # Перевірка, чи час оновити
повільні параметри
            for slow, (_, fast) in zip(self.slow_weights, grads_and_vars):
                slow.assign(slow + self.slow_step * (fast - slow))
                fast.assign(slow)

    def get_config(self):
        config = {
            "inner_optimizer":
tf.keras.optimizers.serialize(self.inner_optimizer),
            "sync_period": self.sync_period,
            "slow_step": self.slow_step,
        }
        base_config = super(Lookahead, self).get_config()
        return {**base_config, **config}
```

3.3. Опис архітектури моделей

У роботі використовуються дві різні моделі для задачі передбачення шахових ходів. Перша модель побудована на основі оптимізатора Adam і має глибоку архітектуру з декількома пов’язаними шарами. Друга модель - це донавчання першої моделі за допомогою Lookahead для підвищення стабільності та точності. Обидві моделі приймають на вхід шахову дошку у вигляді матриці розміром 8x8 і передбачають ймовірності всіх можливих ходів. Далі детальніше про кожну з мереж і їх варіанти.

Перша модель під назвою chess_model на основі оптимізатора Adam має вхідний шар, який перетворює шахову дошку на вектор з довжиною 64. Шістнадцять послідовних прихованих шарів для обробки даних, кожен з яких має 64 нейрони і функцію ReLU. Вихідний шар має 4096 нейронів і функцією активації softmax. Така кількість прихованих шарів допомагає визначати взаємозв’язки між позиціями шахових фігур на дошці, а функція активації softmax у вихідному шарі перетворює значення в ймовірності, що відповідаються усім можливим переміщенням фігур. В лістингу 3.5 приведено програмний код моделі.

Лістинг 3.5

```
def create_model():
    model = Sequential([
        Flatten(input_shape=(8, 8)),
        Dense(128, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(4096, activation="softmax")
    ])
```

Для другої моделі під назвою chess_model_lookahead було вибрано оптимізатор lookahead і вже натреновану модель на базі оптимізатора Adam. Дана модель має покращену стабільність за рахунок саме донавчання з іншим алгоритмом замість тренування від початку. Вхідні дані, як і у попередній моделі, pgn-файли з записами десятків тисяч шахових партій. Програмний код даної моделі приведено в лістингу 3.6.

Лістинг 3.6

```
def finetune_model(model_path, pgn_files):
    model = load_existing_model(model_path)
    x_data, y_data = preprocess_games(parse_pgn_files(pgn_files))

    base_optimizer = Adam(learning_rate=0.001)
    lookahead_optimizer = Lookahead(optimizer=base_optimizer, sync_period=5,
slow_step=0.5)

    model.compile(optimizer=lookahead_optimizer,
loss="sparse_categorical_crossentropy", metrics=["accuracy"])

    early_stopping = EarlyStopping(monitor="loss", patience=3)
    model.fit(x_data, y_data, batch_size=32, epochs=10,
callbacks=[early_stopping])
```

Програмний код оптимізатора lookahead приведено в лістингу 3.7

Лістинг 3.7

```
class Lookahead(Optimizer):
    def __init__(self, optimizer, learning_rate=0.001, sync_period=5,
slow_step=0.5, name="Lookahead", **kwargs):

        super(Lookahead, self).__init__(name=name,
learning_rate=learning_rate, **kwargs)
        self.optimizer = optimizer
        self.sync_period = sync_period
        self.slow_step = slow_step
        self.counter = 0
        self._slow_weights = []

    def _create_slots(self, var_list):
        self.optimizer._create_slots(var_list)
        for var in var_list:
            self._slow_weights.append(tf.Variable(var, trainable=False,
dtype=var.dtype))

    def apply_gradients(self, grads_and_vars, **kwargs):

        grads_and_vars = list(grads_and_vars)
```

```
        if not grads_and_vars or not all(len(pair) == 2 for pair in
grads_and_vars):
            raise ValueError(
                "Invalid grads_and_vars: Expected a list of (gradient,
variable) pairs.")

        self.counter += 1
        if not self._slow_weights:
            self._slow_weights = [tf.identity(w) for _, w in grads_and_vars]

        self.optimizer.apply_gradients(grads_and_vars)

        if self.counter % self.sync_period == 0:
            for (g, w), slow_w in zip(grads_and_vars, self._slow_weights):
                slow_w.assign(slow_w + self.slow_step * (w - slow_w))
                w.assign(slow_w)

    def get_config(self):
        config = {
            "optimizer": tf.keras.optimizers.serialize(self.optimizer),
            "sync_period": self.sync_period,
            "slow_step": self.slow_step,
        }
        base_config = super().get_config()
        return {**base_config, **config}
```

3.4. Розробка програмного коду і тренування моделей

Процес розробки першої моделі на оптимізаторі Adam складався з розробки архітектури моделі, розглянутої раніше у лістингу 5. Далі за допомогою бібліотеки chess було реалізовано метод для перетворення шахової дошки у матрицю з розмірами 8x8 та викликом функції кодування фігур. Результатом роботи методу є повернення матриці з значеннями, які позначають розміщення фігур на дошці. Програмний код функції перетворення шахової дошки в лістингу 3.8.

Лістинг 3.8

```
def board_to_matrix(board):
    matrix = np.zeros((8, 8), dtype=int)
    piece_map = board.piece_map()
    for square, piece in piece_map.items():
        row, col = divmod(square, 8)
        matrix[row, col] = piece_to_int(piece)
    return matrix
```

Розроблено метод кодування фігур, для позначення різних фігур різними цифрами у матриці. При цьому білі фігури позначаються додатніми числами, а чорні – від'ємними. Програмний код представлено у лістингу 3.9.

Лістинг 3.9

```
def piece_to_int(piece):
    piece_dict = {
        chess.PAWN: 1, chess.KNIGHT: 2, chess.BISHOP: 3,
        chess.ROOK: 4, chess.QUEEN: 5, chess.KING: 6
    }
    value = piece_dict[piece.piece_type]
    return value if piece.color == chess.WHITE else -value
```

Далі було розроблено функцію для кодування ходу у форматі «від, до». Програмний код представлено у лістингу 3.10.

Лістинг 3.10

```
def move_to_vector(move):
    return [move.from_square, move.to_square]
```

Описано функцію для зчитування вхідних даних з pgn-файлів, організовану за допомогою функцій бібліотеки chess. Програмний код представлено у лістингу 3.11

Лістинг 3.11

```
def parse_pgn_file(file_path):
    games = []
    with open(file_path, "r") as pgn_file:
        while True:
            game = chess.pgn.read_game(pgn_file)
            if game is None:
                break
            games.append(game)
    return games
```

За допомогою цих функцій модель розуміла розміщення фігур, які відображались у вигляді матриці, що відображено на рисунку 3.6.

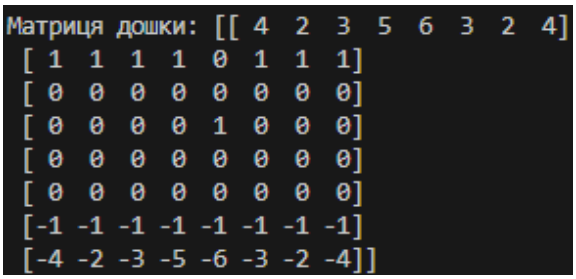

Рис. 3.6. Вигляд матриці дошки

Для обробки даних також було розроблену функцію, за допомогою якої нейромережа могла б вітворювати зчитані партії. Програмний код представлено в лістингу 3.12.

Лістинг 3.12

```
def extract_features_and_labels(games):
    features = []
    labels = []
    for game in games:
        board = game.board()
        for move in game.mainline_moves():

            position = board_to_matrix(board)
            features.append(position)

            move_vector = move_to_vector(move)
            labels.append(move_vector[1])
            board.push(move)
    return np.array(features), np.array(labels).flatten()
```

Наступною була розробка функції тренування нейромережі, яка за допомогою попередніх функцій створювала мітки, на яких пізніше ШІ навчалась і після навчання зберігала модель у файл. Програмний код приведений в лістингу 3.13.

Лістинг 3.13

```
def train_model(games):
    print(f"Loaded {len(games)} games for training.")
    features, labels = extract_features_and_labels(games)
    print("Features shape:", features.shape)
    print("Labels shape:", labels.shape)
    model = create_model()
    model.fit(features, labels, epochs=10, batch_size=32)
    model.save("chess_model_nk.keras")
    print("Model trained and saved")
```

Друга модель на базі оптимізатора lookahead має схожі функції, але відрізняється методом навчання. Було вирішено розробити механізм донавчання раніше створеної моделі, замість розробки нової на базі іншого алгоритму. Потенційно це повинно було підвищити якість навчання моделі і її стабільність. Також для донавчання було підібрано додаткові дані в об’ємі що дорівнював початковому. Програмний код навчання моделі з оптимізатором lookahead представлено в лістингу 3.14.

Лістинг 3.14

```
def finetune_model(model_path, pgn_files):

    model = load_existing_model(model_path)
    all_games = []
    for file in pgn_files:
        all_games.extend(parse_pgn_file(file))

    x_data, y_data = preprocess_games(all_games)

    print(f"Розмірність x_data: {x_data.shape}")
    print(f"Розмірність y_data: {y_data.shape}")

    base_optimizer = Adam(learning_rate=0.001)
    lookahead_optimizer = Lookahead(optimizer=base_optimizer, sync_period=5,
slow_step=0.5)

    model.compile(optimizer=lookahead_optimizer,
loss="sparse_categorical_crossentropy", metrics=["accuracy"])

    early_stopping = EarlyStopping(monitor="loss", patience=3)

    model.fit(
        x_data,
        y_data,
        batch_size=32,
        epochs=10,
        callbacks=[early_stopping],
    )

    updated_model_path = "chess_model_lookahead.keras"
    model.save(updated_model_path)
    print(f"Оновлена модель збережена у: {updated_model_path}")
```

Далі, на базі існуючих функцій, було розроблено програмний код, що дозволяв грати в шахи з моделями приймаючи їх ходи і передаючи їх на вихід, попередньо роблячи перевірку на легальність ходу. Програмний код реалізації отримання руху від моделі представлено в лістингу 3.15.

Лістинг 3.15

```
def get_ai_move(model, board):
    board_matrix = board_to_matrix(board)
    print("Матриця дошки:", board_matrix)

    prediction = model.predict(board_matrix.reshape(1, 8, 8), verbose=0)[0]
    print("Передбачення моделі:", prediction)

    legal_moves = list(board.legal_moves)
    print("Модель шукає легальний хід")

    legal_indices = []
    for move in legal_moves:
        move_vector = move_to_vector(move)
        legal_indices.append(move_vector[1])

    legal_predictions = [prediction[idx] if idx in legal_indices else -1 for
idx in range(len(prediction))]

    best_index = np.argmax(legal_predictions)
    if legal_predictions[best_index] == -1:
        print("ШІ не зміг знайти відповідний хід серед легальних!")
        return None
    print("Модель знайшла легальний хід")

    return legal_moves[legal_indices.index(best_index)]
```

В лістингу 3.16 приведено програмний код функції, що відповідала за проведення гри.

Лістинг 3.16

```
def play_with_gui(model):
    """Основна функція для гри через GUI."""
    board = chess.Board()  # Ініціалізація дошки
    selected_square = None
    running = True

    while running and not board.is_game_over():
        draw_board(board)  # Оновити дошку
        pygame.display.flip()

        if board.turn == chess.WHITE:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    exit()

                if event.type == pygame.MOUSEBUTTONDOWN:
                    x, y = event.pos
                    square = get_square_from_click(x, y)
                    if selected_square is None:
                        if board.piece_at(square) and
board.piece_at(square).color == chess.WHITE:
                            selected_square = square
                    else:
                        move = chess.Move(selected_square, square)
```

```
                        if move in board.legal_moves:
                        selected_square = None

        else:
            ai_move = get_ai_move(model, board)
            if ai_move:
                if board.piece_at(ai_move.from_square).piece_type ==
chess.PAWN and chess.square_rank(ai_move.to_square) in [0, 7]:
                    ai_move.promotion = chess.QUEEN
                board.push(ai_move)
            else:
                print("ШІ не зміг обрати хід. Гра завершується.")
                running = False

    print("Гра завершена.")
    pygame.quit()
```

Також потрібно було розробити код для правила промоції пішака, коли він досягає ворожої бази. Програмний код функції промоції пішака представлено в лістингу 3.17.

Лістинг 3.17

```
def handle_promotion_with_gui(board, move, square_size):
    promotion_window = pygame.display.set_mode((4 * square_size,
square_size))
    promotion_window.fill((200, 200, 200))
    promotion_pieces = ['q', 'r', 'b', 'n']
    images = {
        'q': pygame.image.load('images/white_queen.png'),
        'r': pygame.image.load('images/white_rook.png'),
        'b': pygame.image.load('images/white_bishop.png'),
        'n': pygame.image.load('images/white_knight.png'),
    }
    for i, piece in enumerate(promotion_pieces):
        img = pygame.transform.scale(images[piece], (square_size,
square_size))
        promotion_window.blit(img, (i * square_size, 0))
    pygame.display.flip()
    selected_piece = None
    while not selected_piece:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                exit()
            if event.type == pygame.MOUSEBUTTONDOWN:
                x, _ = event.pos
                index = x // square_size
                if 0 <= index < len(promotion_pieces):
                    selected_piece = promotion_pieces[index]
    move.promotion = {'q': chess.QUEEN, 'r': chess.ROOK, 'b': chess.BISHOP,
'n': chess.KNIGHT}[selected_piece]
    promotion_window = pygame.display.set_mode((square_size * 8, square_size
* 8))
    draw_board(board)
    pygame.display.flip()
    print("Дошку перемальовано")
```

В лістингу 3.18 приведено оновлений програмний код функції, що відповідає за проведення гри і може обробляти правило промоції пішака.

Лістинг 3.18

```
def play_with_gui(model):
    board = chess.Board()
    selected_square = None
    running = True
    while running and not board.is_game_over():
        draw_board(board)
        pygame.display.flip()
        if board.turn == chess.WHITE:  # Хід гравця
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    x, y = event.pos
                    square = get_square_from_click(x, y)

                    if selected_square is None:
                        if board.piece_at(square) and
board.piece_at(square).color == chess.WHITE:
                            selected_square = square
                            print("Фігуру вибрано")
                    else:  # Спроба зробити хід
                        piece = board.piece_at(selected_square)
                        move = chess.Move(selected_square, square)

                        if piece and piece.piece_type == chess.PAWN and
chess.square_rank(square) in [0, 7]:
                            handle_promotion_with_gui(board, move,
square_size)
                            move.promotion = chess.QUEEN
                        if move in board.legal_moves:
                            board.push(move)
                            print("Хід виконано")
                        else:
                            print("Вибрано нелегальний хід")
                        selected_square = None
        else:
            ai_move = get_ai_move(model, board)
            if ai_move:
                if board.piece_at(ai_move.from_square).piece_type ==
chess.PAWN and chess.square_rank(ai_move.to_square) in [0, 7]:
                    ai_move.promotion = chess.QUEEN
                board.push(ai_move)
            else:
                running = False
pygame.quit()
```

В лістингу 3.19 приведено програмний код файлу, у якому відбувається повне оброблення грального дошки, визначення легальних рухів, обробки вибору фігури і отримання ходів від моделей.

Лістинг 3.19

```
import pygame
import numpy as np
import chess
from tensorflow.keras.models import load_model

# Load model
model = load_model("chess_model_114k.keras")

pygame.init()
size = 512
square_size = size // 8
screen = pygame.display.set_mode((size, size))
pygame.display.set_caption("Chess Game")

pieces = {
    'P': pygame.image.load('images/white_pawn.png').convert_alpha(),
    'N': pygame.image.load('images/white_knight.png').convert_alpha(),
    'B': pygame.image.load('images/white_bishop.png').convert_alpha(),
    'R': pygame.image.load('images/white_rook.png').convert_alpha(),
    'Q': pygame.image.load('images/white_queen.png').convert_alpha(),
    'K': pygame.image.load('images/white_king.png').convert_alpha(),
    'p': pygame.image.load('images/black_pawn.png').convert_alpha(),
    'n': pygame.image.load('images/black_knight.png').convert_alpha(),
    'b': pygame.image.load('images/black_bishop.png').convert_alpha(),
    'r': pygame.image.load('images/black_rook.png').convert_alpha(),
    'q': pygame.image.load('images/black_queen.png').convert_alpha(),
    'k': pygame.image.load('images/black_king.png').convert_alpha(),
}

for key in pieces:
    pieces[key] = pygame.transform.scale(pieces[key], (square_size,
square_size))

def draw_board(board):
    screen.fill((255, 255, 255))
    for row in range(8):
        for col in range(8):
            color = (240, 217, 181) if (row + col) % 2 == 0 else (181, 136,
99)
            pygame.draw.rect(screen, color, pygame.Rect(col * square_size,
row * square_size, square_size, square_size))
    piece_map = board.piece_map()
    for square, piece in piece_map.items():
        row, col = divmod(square, 8)
        row = 7 - row
        screen.blit(pieces[piece.symbol()], (col * square_size, row *
square_size))
    pygame.display.flip()

def get_square_from_click(x, y):
    col = x // square_size
    row = y // square_size
    row = 7 - row
    return chess.square(col, row)

def board_to_matrix(board):
    matrix = np.zeros((8, 8), dtype=int)
    for square in chess.SQUARES:
        piece = board.piece_at(square)
        row = chess.square_rank(square)
        col = chess.square_file(square)
```

```
            matrix[row, col] = piece_to_int(piece)
    return matrix

def piece_to_int(piece):
    if piece is None:
        return 0
    piece_dict = {
        chess.PAWN: 1,
        chess.KNIGHT: 2,
        chess.BISHOP: 3,
        chess.ROOK: 4,
        chess.QUEEN: 5,
        chess.KING: 6
    }
    color = 1 if piece.color == chess.WHITE else -1
    return color * piece_dict[piece.piece_type]

def handle_promotion_with_gui(board, move, square_size):
    print("Початок функції промоції")
    promotion_window = pygame.display.set_mode((4 * square_size,
square_size))
    promotion_window.fill((200, 200, 200))
    promotion_pieces = ['q', 'r', 'b', 'n']
    images = {
        'q': pygame.image.load('images/white_queen.png'),
        'r': pygame.image.load('images/white_rook.png'),
        'b': pygame.image.load('images/white_bishop.png'),
        'n': pygame.image.load('images/white_knight.png'),
    }
    for i, piece in enumerate(promotion_pieces):
        img = pygame.transform.scale(images[piece], (square_size,
square_size))
        promotion_window.blit(img, (i * square_size, 0))
    pygame.display.flip()

    selected_piece = None
    while not selected_piece:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                exit()
            if event.type == pygame.MOUSEBUTTONDOWN:
                x, _ = event.pos
                index = x // square_size
                if 0 <= index < len(promotion_pieces):
                    selected_piece = promotion_pieces[index]
    move.promotion = {'q': chess.QUEEN, 'r': chess.ROOK, 'b': chess.BISHOP,
'n': chess.KNIGHT}[selected_piece]
    print("Кінець функції промоції")
    promotion_window = pygame.display.set_mode((square_size * 8, square_size
* 8))
    draw_board(board)
    pygame.display.flip()
    print("Дошку перемальовано")

def move_to_vector(move):
    """Кодування ходу у форматі (from, to)."""
    return [move.from_square, move.to_square]

def get_ai_move(model, board):
    """Отримати хід від нейромережі для поточної позиції."""
    board_matrix = board_to_matrix(board)
    print("Матриця дошки:", board_matrix)
```

```
    prediction = model.predict(board_matrix.reshape(1, 8, 8), verbose=0)[0]
    print("Передбачення моделі:", prediction)

    legal_moves = list(board.legal_moves)
    print("Модель шукає легальний хід")
    # Перетворимо легальні ходи в індекси для моделі
    legal_indices = []
    for move in legal_moves:
        move_vector = move_to_vector(move)
        legal_indices.append(move_vector[1])

    legal_predictions = [prediction[idx] if idx in legal_indices else -1 for
idx in range(len(prediction))]

    best_index = np.argmax(legal_predictions)
    if legal_predictions[best_index] == -1:
        print("ШІ не зміг знайти відповідний хід серед легальних!")
        return None
    print("Модель знайшла легальний хід")
    return legal_moves[legal_indices.index(best_index)]

def play_with_gui(model):
    board = chess.Board()
    selected_square = None
    running = True
    while running and not board.is_game_over():
        draw_board(board)
        pygame.display.flip()
        if board.turn == chess.WHITE:  # Хід гравця
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    x, y = event.pos
                    square = get_square_from_click(x, y)

                    if selected_square is None:  # Вибір фігури
                        if board.piece_at(square) and
board.piece_at(square).color == chess.WHITE:
                            selected_square = square
                            print("Фігуру вибрано")
                    else:
                        piece = board.piece_at(selected_square)
                        move = chess.Move(selected_square, square)

                        if piece and piece.piece_type == chess.PAWN and
chess.square_rank(square) in [0, 7]:
                            handle_promotion_with_gui(board, move,
square_size)
                            move.promotion = chess.QUEEN
                        if move in board.legal_moves:
                            board.push(move)
                            print("Хід виконано")
                        else:
                            print("Вибрано нелегальний хід")

                        selected_square = None
        else:
            ai_move = get_ai_move(model, board)
            if ai_move:
```

```
                if board.piece_at(ai_move.from_square).piece_type ==
chess.PAWN and chess.square_rank(ai_move.to_square) in [0, 7]:
                    ai_move.promotion = chess.QUEEN
                board.push(ai_move)
            else:
                running = False
    pygame.quit()

play_with_gui(model)
```

За допомогою даного програмного коду і знайдених у вільному доступі даних шахових партій професійних гравців, було проведено процес тренування кожної з чотирьох моделей:

1. Базова модель на базі алгоритму Adam, тренована на чотирьох тисячах партій з налаштуванням швидкості тренування 0.001, функцією втрат sparse_categorical_crossentropy і метрикою accuracy під назвою chess_model;

```
Epoch 1/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 54s 6ms/step - accuracy: 0.0348 - loss: 4.0655
Epoch 2/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 81s 10ms/step - accuracy: 0.0455 - loss: 3.9094
Epoch 3/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 85s 10ms/step - accuracy: 0.0531 - loss: 3.8827
Epoch 4/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 79s 10ms/step - accuracy: 0.0523 - loss: 3.8810
Epoch 5/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 76s 9ms/step - accuracy: 0.0437 - loss: 3.9384
Epoch 6/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 71s 9ms/step - accuracy: 0.0572 - loss: 3.8674
Epoch 7/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 75s 9ms/step - accuracy: 0.0594 - loss: 3.8541
Epoch 8/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 69s 8ms/step - accuracy: 0.0605 - loss: 3.8436
Epoch 9/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 56s 7ms/step - accuracy: 0.0611 - loss: 3.8339
Epoch 10/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 53s 6ms/step - accuracy: 0.0609 - loss: 3.8330
```

Рис. 3.2. Процес тренування моделі chess_model

2. Покращена модель Adam тренована на дев’ятнадцяти тисячах партій з налаштуванням швидкості тренування 0.001, функцією втрат sparse_categorical_crossentropy і метрикою accuracy під назвою chess_model_19k;

```
Epoch 1/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 223s 6ms/step - accuracy: 0.0471 - loss: 3.9367
Epoch 2/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 229s 7ms/step - accuracy: 0.0796 - loss: 3.7572
Epoch 3/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 203s 6ms/step - accuracy: 0.0975 - loss: 3.6973
Epoch 4/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 193s 6ms/step - accuracy: 0.1033 - loss: 3.6691
Epoch 5/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 191s 6ms/step - accuracy: 0.1057 - loss: 3.6538
Epoch 6/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 192s 6ms/step - accuracy: 0.1049 - loss: 3.6510
Epoch 7/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 191s 6ms/step - accuracy: 0.1105 - loss: 3.6335
Epoch 8/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 194s 6ms/step - accuracy: 0.1097 - loss: 3.6358
Epoch 9/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 198s 6ms/step - accuracy: 0.1090 - loss: 3.6381
Epoch 10/10
34393/34393 ━━━━━━━━━━━━━━━━━━━━ 193s 6ms/step - accuracy: 0.1057 - loss: 3.6529
```

Рис. 3.3. Процес тренування моделі chess_model_19k

3. Модель Adam з механікою донавчання за допомогою оптимізатора lookahead і додаткових даних у вигляді одинадцяти тисяч партій під назвою chess_model_lookahead;

```
Epoch 1/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 645s 8ms/step - accuracy: 0.0525 - loss: 3.9033
Epoch 2/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 613s 7ms/step - accuracy: 0.0614 - loss: 3.8528
Epoch 3/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 543s 7ms/step - accuracy: 0.0799 - loss: 3.7720
Epoch 4/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 559s 7ms/step - accuracy: 0.0880 - loss: 3.7278
Epoch 5/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 521s 6ms/step - accuracy: 0.0862 - loss: 3.7365
Epoch 6/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 511s 6ms/step - accuracy: 0.0956 - loss: 3.7055
Epoch 7/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 522s 6ms/step - accuracy: 0.1027 - loss: 3.6797
Epoch 8/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 533s 6ms/step - accuracy: 0.1037 - loss: 3.6743
Epoch 9/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 550s 7ms/step - accuracy: 0.1064 - loss: 3.6643
Epoch 10/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 524s 6ms/step - accuracy: 0.0784 - loss: 3.7945
```

Рис. 3.4. Процес тренування моделі chess_model_lookahead

4. Просунута модель Adam тренована на ста чотирнадцяти тисячах партій під назвою chess_model_114k;

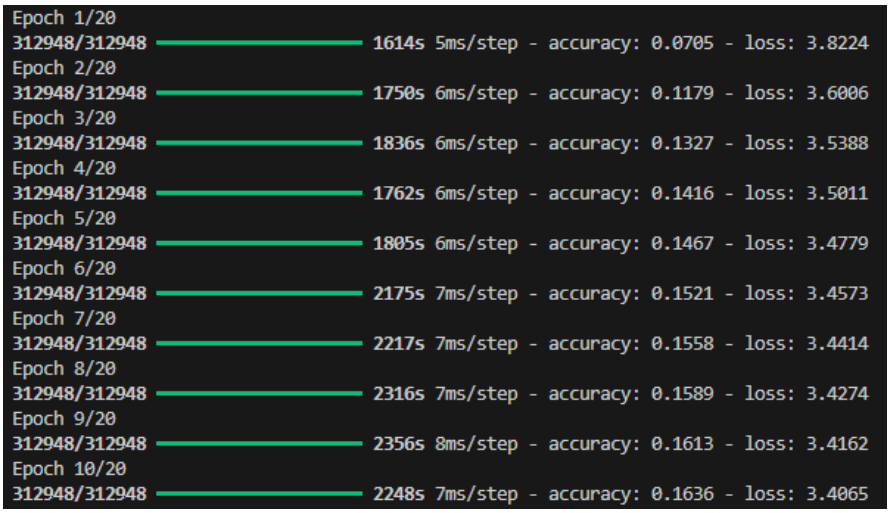

Рис. 3.5. Частина процесу тренування моделі chess_model_114k

3.5. Тестування роботи моделей

Після навчання моделей йде процес тестування, під час якого буде визначатись вплив кількості даних і налаштувань параметрів моделей на якість навчання і роботи нейромережі. Тестування нейромережі, розробленої для гри в шахи, спрямоване на оцінку її продуктивності, точності передбачень та здатності до адаптації в реальних умовах гри. Основними цілями є:

- визначення рівня гри нейромережі залежно від кількості даних для навчання;
- оцінка її здатності протидіяти людським гравцям;
- аналіз якості ходів моделі.

В процесі тестування зверталась увага на дії моделей, їхні спроби захисту і, нападу та розміну фігур. Також кільком грацям було

запропоновано провести партії з кожною моделлю. Для повнішого оцінювання моделі підбирались гравці з різними рівнями гри і вміннями.

Перша модель під назвою "chess_model_4k" навчалась на чотирьох тисячах партій професійних гравців використовуючи оптимізатор Adam з швидкісю навчання, що дорівнює 0.001, функцією втрат sparse_categorical_crossentropy і метрикою accuracy. Сумарний час навчання становив близько сімнадцяти хвилин. Метрика точності коливаючись зростала повільними темпами, що підвищувало впевненість в передбаченнях моделі в той час як функція втрат понижалась, що означало більшу ефективність машинного навчання.

Процес навчання з відлображенням параметрів до кожної епохи навчання зображено на рисунку 3.6.

```
Epoch 1/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 54s 6ms/step - accuracy: 0.0348 - loss: 4.0655
Epoch 2/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 81s 10ms/step - accuracy: 0.0455 - loss: 3.9094
Epoch 3/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 85s 10ms/step - accuracy: 0.0531 - loss: 3.8827
Epoch 4/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 79s 10ms/step - accuracy: 0.0523 - loss: 3.8810
Epoch 5/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 76s 9ms/step - accuracy: 0.0437 - loss: 3.9384
Epoch 6/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 71s 9ms/step - accuracy: 0.0572 - loss: 3.8674
Epoch 7/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 75s 9ms/step - accuracy: 0.0594 - loss: 3.8541
Epoch 8/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 69s 8ms/step - accuracy: 0.0605 - loss: 3.8436
Epoch 9/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 56s 7ms/step - accuracy: 0.0611 - loss: 3.8339
Epoch 10/10
8250/8250 ━━━━━━━━━━━━━━━━━━━━ 53s 6ms/step - accuracy: 0.0609 - loss: 3.8330
```

Рис. 3.6. Процес навчання моделі chess_model_4k

Висновок після тестування: модель робила багато поганих ходів і лише зрідка хороші, слабко захищала свої фігури і не дуже активно забирала ворожі. Це відмітили всі гравці, ось як вони відгукнулись про модель:

– Гравець 1: нейромережа майже не робила хороші ходи, коли поганих ходів було чимало. Загалом гра вийшла легкою.
– Гравець 2: Хороших ходів мало, переважали погані, легка партія.
– Гравець 3: Модель погано захищалась, пропускала можливості забирати фігури без ризику. Партія була легкою.
– Гравець 4: Було кілька хроших ходів, але загалом більше поганих.

Друга модель під назвою "chess_model_19k" навчалась на дев’ятнадцятьох тисячах партій професійних гравців використовуючи оптимізатор Adam з швидкісю навчання, що дорівнює 0.0001, функцією втрат sparse_categorical_crossentropy і метрикою accuracy. Сумарний час навчання становив близько години хвилин. Процес навчання зображено на рисунку 3.7.

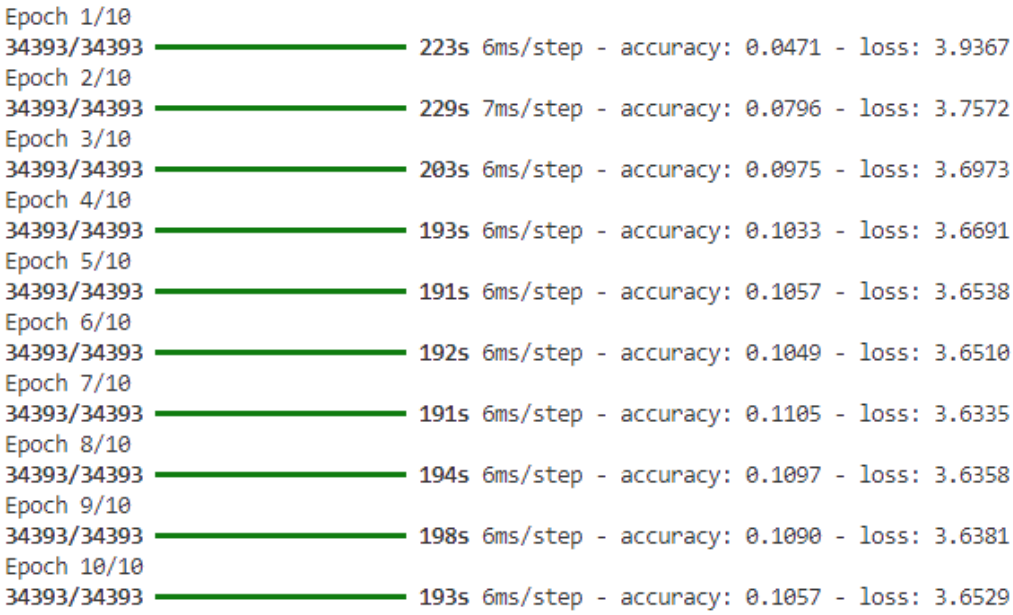

Рис. 3.7. Процес навчання моделі chess_model_19k

Висновок після тестування: кількість поганих ходів знизилась, а хороших трохи зросла. Модель почала діяти більш хаотично іноді не забираючи фігури без ризику, а іноді йдучи на погані обміни фігурами. В

ендшпілі нерідко робила повторювані ходи. Гравці відчули покращення, але воно було недостатнім для суттєвого підчищення складності:

- Гравець 1: нейромережа робила хороші ходи, в той час коли поганих ходів було менше, поганий напад. Гра вийшла складнішою, хоча не надто.
- Гравець 2: Хороші ходи присутні на рівні з поганими. Було трохи складніше.
- Гравець 3: Модель хаотично рухалась і погано захищалась. Партія була досить легкою.
- Гравець 4: Система робила хороші ходи, але не дуже добре захищалась. Поганих було менше ніж таких, що викликали питання.

Третя модель під назвою "chess_model_lookahead" була навчена на дев’ятнадцятьох тисячах партій професійних гравців. Далі вона використовуючи оптимізатор lookahead з швидкісю навчання, що дорівнює 0.001, функцією втрат sparse_categorical_crossentropy, за допомогою процесу донавчання тренувалась на додаткових двадцяти тисячах партій. Процес навчання зображено на рисунку 3.8.

```
Epoch 1/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 645s 8ms/step - accuracy: 0.0525 - loss: 3.9033
Epoch 2/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 613s 7ms/step - accuracy: 0.0614 - loss: 3.8528
Epoch 3/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 543s 7ms/step - accuracy: 0.0799 - loss: 3.7720
Epoch 4/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 559s 7ms/step - accuracy: 0.0880 - loss: 3.7278
Epoch 5/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 521s 6ms/step - accuracy: 0.0862 - loss: 3.7365
Epoch 6/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 511s 6ms/step - accuracy: 0.0956 - loss: 3.7055
Epoch 7/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 522s 6ms/step - accuracy: 0.1027 - loss: 3.6797
Epoch 8/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 533s 6ms/step - accuracy: 0.1037 - loss: 3.6743
Epoch 9/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 550s 7ms/step - accuracy: 0.1064 - loss: 3.6643
Epoch 10/10
82869/82869 ━━━━━━━━━━━━━━━━━━━━ 524s 6ms/step - accuracy: 0.0784 - loss: 3.7945
```

Рис. 3.7. Процес навчання моделі chess_model_19k

Висновок після тестування: кількість поганих ходів трохи зменшилась, зросла стабільність. Модель все ще діяла хаотично але вже менше надмірно ризикувала фігурами. В ендшпілі інколи робила повторювані, або погані ходи. На думку гравців покращення було незначним, але у випадку з слабким гравцем кількість поганих ходів зросла після опенінгу:

- Гравець 1: Важко відрізнити, поганих ходів було менше, але гра не була важкою.
- Гравець 2: Гра пройшла легко.
- Гравець 3: Модель краще захищалась, але проводила погані розміни фігур, інколи не атакувала.
- Гравець 4: Система добре грала, але під кінець почала робити дивні ходи.

Четверта модель під назвою "chess_model_114k" навчалась на ста чотирнадцятьох тисячах партій професійних гравців використовуючи оптимізатор Adam з швидкісю навчання, що дорівнює 0.0001, функцією втрат sparse_categorical_crossentropy і метрикою accuracy. Процес навчання зображено на рисунку 3.9.

```
Epoch 1/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 1614s 5ms/step - accuracy: 0.0705 - loss: 3.8224
Epoch 2/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 1750s 6ms/step - accuracy: 0.1179 - loss: 3.6006
Epoch 3/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 1836s 6ms/step - accuracy: 0.1327 - loss: 3.5388
Epoch 4/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 1762s 6ms/step - accuracy: 0.1416 - loss: 3.5011
Epoch 5/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 1805s 6ms/step - accuracy: 0.1467 - loss: 3.4779
Epoch 6/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 2175s 7ms/step - accuracy: 0.1521 - loss: 3.4573
Epoch 7/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 2217s 7ms/step - accuracy: 0.1558 - loss: 3.4414
Epoch 8/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 2316s 7ms/step - accuracy: 0.1589 - loss: 3.4274
Epoch 9/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 2356s 8ms/step - accuracy: 0.1613 - loss: 3.4162
Epoch 10/20
312948/312948 ━━━━━━━━━━━━━━━━━━━━ 2248s 7ms/step - accuracy: 0.1636 - loss: 3.4065
```

Рис. 3.7. Процес навчання моделі chess_model_114k

Висновок після тестування: кількість поганих ходів знизилась, а хороших - зросла. Модель відповідала на різні опенінги різними початковими ходами, але губилась в ендшпілі. Краще забирала фігури і захищала свої, хоча і тут були присутні помилки. Інколи модель не робила хороші ходи, в яких потрібно було жертвувати фігурою, а інколи йшла на погані розміни. Гравці відмітили покращення рухів моделі, хоча також підмітили погані рішення:

- Гравець 1: Нейромережа робила хороші ходи, хоча погані все ще були присутні. Грати стало важче.
- Гравець 2: Мережа добре грала, поганих ходів було не багато, іноді впускала хороші ходи.
- Гравець 3: Модель розпочала з агресивної реалізації фігур, які намагалась захищати. Ігнорувала деякі загрози.
- Гравець 4: Грати було складно, система непогано захищалась, хоча також іноді робила погані ходи, завдяки чому вийшло звести гру в нічию.

3.6. Висновки до розділу

В розділі було детально розглянуто основні етапи розробки нейронної мережі, починаючи від визначення мети й вибору архітектури до збору даних і тренування моделі. Описано, як архітектура нейромережі, зокрема кількість шарів, типи нейронів і зв’язків між ними, визначає її здатність розв'язувати конкретні задачі.

Збір даних є фундаментальним етапом, оскільки якість даних безпосередньо впливає на результати навчання. Було проаналізовано методи підготовки й очищення даних, а також створення збалансованих вибірок для уникнення зміщення.

Для демонстрації навчання моделей було вирішено вибрати для тренування два алгоритма з різними принципами роботи. Таким чином було отримано дві моделі. Перша модель побудована на основі оптимізатора Adam і має глибоку архітектуру з декількома пов’язаними шарами. Друга модель - це донавчання першої моделі за допомогою Lookahead для підвищення стабільності та точності.

Процес тренування моделі включає вибір відповідної функції втрат і алгоритму оптимізації, які спрямовують навчання в бік мінімізації помилок. Обговорено, як важливо контролювати процес навчання за допомогою механізмів регуляризації, нормалізації та ранньої зупинки для забезпечення збіжності та уникнення перенавчання.

У підсумку, ефективна розробка нейромережі вимагає комплексного підходу, що охоплює як вибір правильної архітектури та методів тренування, так і підготовку високоякісних даних. Виконання всіх цих етапів дозволяє створювати потужні моделі, здатні успішно вирішувати поставлені завдання.

ВИСНОВКИ

Навчання нейромережі це складний і довгий процес, для якого потрібно багато ресурсів навіть для досягнення низького рівня якості навчання. Процес навчання починається з аналізу теми, для якої повинна розроблятись нейромережа. Від цього залежить вибір алгоритму, що є досить важливим тому, що від нього залежить парадигма навчання моделей. Сам процес навчання моделей був досить довгий, що було пов’язано з складною архітектурою моделі та обмеженими ресурсами персонального комп’ютера, який повинен був обробляти дані.

Під час роботи було досліджено технології штучних нейронних мереж, принципів їх роботи, механізмів роботи з даними і навчання, визначення важливості кількості і якості даних для навчання, а також алгоритмів навчання і налаштовуваних параметрів, від яких залежить якість і швидкість навчання.

Практичною частиною роботи є створення моделей нейромережі для гри в шахи. Модель не повинна нічого знати про гру наперед, а вивчати її покладаючись на представлені дані зіграних партій реальних людей високого рівня майстерності. Після аналізу шахових партій вона повинна, завдяки навчанню, вміти прогнозувати можливі легальні ходи і давати їм оцінку. Базуючись на прогнозуванні модель повинна вибирати найкращий хід і виконувати його, після чого знову аналізувати ситуацію на ігровій дошці. В процесі тестування проводився аналіз якості навчання моделей, а саме відзначались хороші і погані ходи, напад і захист, розміни фігурами, використання існуючих опенінгів і реакцію моделей на них.

За представленими результатами навчання та тестування моделей можна зробити висновок, що для хорошого, якісного навчання нейромережі для гри в шахи, без попереднього їх навчання можливих ходів, стратегій і механік, потрібно багато часу, якісні дані у великій кількості і достатньо ресурсів персонального комп’ютера для обробки цих даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. "Алгоритми машинного навчання. Глибокі нейромережі в задачах механіки суцільних середовищ". Київський Національний університет імені Тараса Шевченка. М.В. Лавренюк. (2024).
2. "Штучні нейронні мережі: базові положення". Київський Політехнічний Інститут імені Ігоря Сікорського. І.А. Терейковський, Д.А. Бушуєв, Л.О. Терейковська. (2022).
3. "Штучний інтелект". Києво-Могилянська академія. Глибовець М. М., Олецький О.В. (2002).
4. "Енциклопедія кібернетики". В. Глушков. (1973).
5. "Життя 3.0.: доба штучного інтелекту". Наш формат. Макс Тегмарк. (2017).
6. "Засоби штучного інтелекту: навчальний посібник". Національний університет «Львівська політехніка». Р. О. Ткаченко, Н. О. Кустра, О. М. Павлюк, У. В. Поліщук. (2014).
7. "Системи штучного інтелекту: навчальний посібник". Національний університет «Львівська політехніка». Н. Б. Шаховська, Р. М. Камінський, О. Б. Вовк. (2018).
8. "Inductive Learning Algorithms for Complex Systems Modeling". Boca Raton: CRC Press. Madala, H.R.. Ivakhnenko, A.G. (1994).
9. "Perceptrons". Minsky, Marvin Lee. (1969).
10. "Artificial Intelligence: A Modern Approach". Pearson. Stuart J. Russell, Peter Norvig. (2015)
11. "Artificial Intelligence: An Introductory Course". Edinburgh University Press. Alan Bundy, Rod Burstall. (1984).
12. "The Quest for Artificial Intelligence". Cambridge University Press. Nils J. Nilsson. (2009).
13. "Neural Networks in Materials Science". ISIJ International. Bhadeshia H. K. D. H. (1999).

14. "Neural networks for pattern recognition". Clarendon Press. Bishop, Christopher M. (1995).
15. "Approximation by Superpositions of a Sigmoidal function". Mathematics of Control, Signals, and Systems. Cybenko, G.V. (2006).
16. "Information Theory, Inference, and Learning Algorithms". Cambridge University Press. MacKay, David, J.C. (2003).
17. "Artificial intelligence". Grey House Publishing. Wilson, Halsey (2018).
18. "Advanced methods in neural computing". Van Nostrand Reinhold. Wasserman, Philip D. (1993).
19. "Neural networks for statistical modeling". Van Nostrand Reinhold. Smith, Murray (1993).
20. "Introduction to neural networks : design, theory and applications". California Scientific Software. Lawrence, Jeanette (1994).
21. "Computational intelligence: a methodological introduction". Springer. Kruse, Rudolf; Borgelt, Christian; Klawonn, F.; Moewes, Christian; Steinbrecher, Matthias; Held, Pascal (2013).
22. "Information theory, inference, and learning algorithms". Cambridge University Press. (2003).
23. "Introduction to the theory of neural computation". Addison-Wesley. Hertz, J.; Palmer, Richard G.; Krogh, Anders S. (1991).
24. "Neural networks : a comprehensive foundation". Prentice Hall. Haykin, Simon S. (1999).
25. "An introduction to neural networks". UCL Press. Gurney, Kevin (1997).
26. "The Cascade-Correlation Learning Architecture". Fahlman, S.; Lebiere, C (1991).
27. "Wayback Machine". (2010).
28. "Common Genetic Encoding for Both Direct and Indirect Encodings of Networks". Genetic and Evolutionary Computation Conference (GECCO 2007). Yohannes Kassahun, Mark Edgington, Jan Hendrik Metzen, Gerald Sommer and Frank Kirchner.

29. "A hierarchical neural network model for selective attention". Springer-Verlag. Fukushima, Kunihiko (1987).
30. "Neocognitron". Scholarpedia. Fukushima, Kunihiko (2007).
31. "ChatGPT провіщає інтелектуальну революцію". Генрі Кіссінджер, Ерік Шмідт, Даніель Гуттенлохер.
32. Електронний ресурс. – Режим доступу: https://towardsdatascience.com/dont-look-backwards-lookahead-6bcd7ff50f93.
33. Електронний ресурс. – Режим доступу: https://computing4all.com/courses/introductory-data-science.
34. Електронний ресурс. – Режим доступу: https://www.tensorflow.org/api_docs/python/tf/keras.
35. Електронний ресурс. – Режим доступу: https://stackoverflow.com/questions/tagged/machine-learning.
36. Електронний ресурс. – Режим доступу: https://github.com.
37. Електронний ресурс. – Режим доступу: https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification.
38. Електронний ресурс. – Режим доступу: https://openai.com/index/learning-to-reason-with-llms.
39. Електронний ресурс. – Режим доступу: https://arize.com/blog-course/binary-cross-entropy-log-loss.
40. Електронний ресурс. – Режим доступу: https://paperswithcode.com/method/lookahead.

ДОДАТКИ

Додаток А

Програмні лістинги

```
import pygame
import numpy as np
import chess
from tensorflow.keras.models import load_model

# Load model
model = load_model("chess_model_114k.keras")

# Initialize screen
pygame.init()
size = 512
square_size = size // 8
screen = pygame.display.set_mode((size, size))
pygame.display.set_caption("Chess Game")

# Load chess piece images
pieces = {
    'P': pygame.image.load('images/white_pawn.png').convert_alpha(),
    'N': pygame.image.load('images/white_knight.png').convert_alpha(),
    'B': pygame.image.load('images/white_bishop.png').convert_alpha(),
    'R': pygame.image.load('images/white_rook.png').convert_alpha(),
    'Q': pygame.image.load('images/white_queen.png').convert_alpha(),
    'K': pygame.image.load('images/white_king.png').convert_alpha(),
    'p': pygame.image.load('images/black_pawn.png').convert_alpha(),
    'n': pygame.image.load('images/black_knight.png').convert_alpha(),
    'b': pygame.image.load('images/black_bishop.png').convert_alpha(),
    'r': pygame.image.load('images/black_rook.png').convert_alpha(),
    'q': pygame.image.load('images/black_queen.png').convert_alpha(),
    'k': pygame.image.load('images/black_king.png').convert_alpha(),
}

# Scale images
for key in pieces:
    pieces[key] = pygame.transform.scale(pieces[key], (square_size,
square_size))

# Draw the chessboard
def draw_board(board):
    screen.fill((255, 255, 255))
    for row in range(8):
        for col in range(8):
            color = (240, 217, 181) if (row + col) % 2 == 0 else (181, 136,
99)
            pygame.draw.rect(screen, color, pygame.Rect(col * square_size,
row * square_size, square_size, square_size))
    piece_map = board.piece_map()
    for square, piece in piece_map.items():
        row, col = divmod(square, 8)
        row = 7 - row  # Flip row for display
        screen.blit(pieces[piece.symbol()], (col * square_size, row *
square_size))
    pygame.display.flip()

def get_square_from_click(x, y):
    col = x // square_size
    row = y // square_size
```

```
    row = 7 - row
    return chess.square(col, row)

def board_to_matrix(board):
    matrix = np.zeros((8, 8), dtype=int)
    for square in chess.SQUARES:
        piece = board.piece_at(square)
        row = chess.square_rank(square)
        col = chess.square_file(square)
        matrix[row, col] = piece_to_int(piece)
    return matrix

def piece_to_int(piece):
    if piece is None:
        return 0
    piece_dict = {
        chess.PAWN: 1,
        chess.KNIGHT: 2,
        chess.BISHOP: 3,
        chess.ROOK: 4,
        chess.QUEEN: 5,
        chess.KING: 6
    }
    color = 1 if piece.color == chess.WHITE else -1
    return color * piece_dict[piece.piece_type]

def handle_promotion_with_gui(board, move, square_size):
    print("Початок функції промоції")
    promotion_window = pygame.display.set_mode((4 * square_size,
square_size))
    promotion_window.fill((200, 200, 200))
    promotion_pieces = ['q', 'r', 'b', 'n']
    images = {
        'q': pygame.image.load('images/white_queen.png'),
        'r': pygame.image.load('images/white_rook.png'),
        'b': pygame.image.load('images/white_bishop.png'),
        'n': pygame.image.load('images/white_knight.png'),
    }
    for i, piece in enumerate(promotion_pieces):
        img = pygame.transform.scale(images[piece], (square_size,
square_size))
        promotion_window.blit(img, (i * square_size, 0))
    pygame.display.flip()

    selected_piece = None
    while not selected_piece:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                exit()
            if event.type == pygame.MOUSEBUTTONDOWN:
                x, _ = event.pos
                index = x // square_size
                if 0 <= index < len(promotion_pieces):
                    selected_piece = promotion_pieces[index]
    move.promotion = {'q': chess.QUEEN, 'r': chess.ROOK, 'b': chess.BISHOP,
'n': chess.KNIGHT}[selected_piece]
    print("Кінець функції промоції")
    promotion_window = pygame.display.set_mode((square_size * 8, square_size
* 8))  # Повернення до розмірів дошки
    draw_board(board)  # Перемальовування дошки
    pygame.display.flip()
    print("Дошку перемальовано")
```

```
def move_to_vector(move):
    """Кодування ходу у форматі (from, to)."""
    return [move.from_square, move.to_square]

def get_ai_move(model, board):
    """Отримати хід від нейромережі для поточної позиції."""
    # Перетворення дошки на матрицю
    board_matrix = board_to_matrix(board)
    print("Матриця дошки:", board_matrix)

    # Передбачення моделі
    prediction = model.predict(board_matrix.reshape(1, 8, 8), verbose=0)[0]
    print("Передбачення моделі:", prediction)

    # Легальні ходи
    legal_moves = list(board.legal_moves)
    print("Модель шукає легальний хід")
    # Перетворимо легальні ходи в індекси для моделі
    legal_indices = []
    for move in legal_moves:
        move_vector = move_to_vector(move)
        legal_indices.append(move_vector[1])  # Індекс у форматі моделі

    # Фільтруємо передбачення лише для легальних ходів
    legal_predictions = [prediction[idx] if idx in legal_indices else -1 for
idx in range(len(prediction))]

    # Обираємо найімовірніший хід серед легальних
    best_index = np.argmax(legal_predictions)
    if legal_predictions[best_index] == -1:
        print("ШІ не зміг знайти відповідний хід серед легальних!")
        return None
    print("Модель знайшла легальний хід")
    # Повертаємо відповідний легальний хід
    return legal_moves[legal_indices.index(best_index)]

def play_with_gui(model):
    board = chess.Board()
    selected_square = None
    running = True
    while running and not board.is_game_over():
        draw_board(board)
        pygame.display.flip()
        if board.turn == chess.WHITE:  # Хід гравця
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    x, y = event.pos
                    square = get_square_from_click(x, y)

                    if selected_square is None:  # Вибір фігури
                        if board.piece_at(square) and
board.piece_at(square).color == chess.WHITE:
                            selected_square = square
                            print("Фігуру вибрано")
                    else:  # Спроба зробити хід
                        piece = board.piece_at(selected_square)
                        move = chess.Move(selected_square, square)
```

```
                        # Якщо це пішак, який досягає останнього ряду, додати
параметр промоції
                        if piece and piece.piece_type == chess.PAWN and
chess.square_rank(square) in [0, 7]:
                            print("Пішак на базу? Виклик функції промоції")
                            handle_promotion_with_gui(board, move,
square_size)
                            move.promotion = chess.QUEEN  # Тимчасово
встановити ферзя за замовчуванням

                        # Перевірка на легальний хід
                        if move in board.legal_moves:
                            board.push(move)
                            print("Хід виконано")
                        else:
                            print("Вибрано нелегальний хід")

                        selected_square = None
        else:  # Хід ШІ
            ai_move = get_ai_move(model, board)
            if ai_move:
                # Додати промоцію для пішака, якщо ШІ досягає останнього ряду
                if board.piece_at(ai_move.from_square).piece_type ==
chess.PAWN and chess.square_rank(ai_move.to_square) in [0, 7]:
                    ai_move.promotion = chess.QUEEN
                board.push(ai_move)
            else:
                running = False
    pygame.quit()

play_with_gui(model)


import pygame
import numpy as np
import chess
import tensorflow as tf
from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Optimizer
from tensorflow.keras.saving import register_keras_serializable

# Load model
model = load_model("chess_model_lookahead.keras")

# Initialize screen
pygame.init()
size = 512
square_size = size // 8
screen = pygame.display.set_mode((size, size))
pygame.display.set_caption("Chess Game")

# Load chess piece images
pieces = {
    'P': pygame.image.load('images/white_pawn.png').convert_alpha(),
    'N': pygame.image.load('images/white_knight.png').convert_alpha(),
    'B': pygame.image.load('images/white_bishop.png').convert_alpha(),
    'R': pygame.image.load('images/white_rook.png').convert_alpha(),
    'Q': pygame.image.load('images/white_queen.png').convert_alpha(),
    'K': pygame.image.load('images/white_king.png').convert_alpha(),
    'p': pygame.image.load('images/black_pawn.png').convert_alpha(),
    'n': pygame.image.load('images/black_knight.png').convert_alpha(),
```

```
    'b': pygame.image.load('images/black_bishop.png').convert_alpha(),
    'r': pygame.image.load('images/black_rook.png').convert_alpha(),
    'q': pygame.image.load('images/black_queen.png').convert_alpha(),
    'k': pygame.image.load('images/black_king.png').convert_alpha(),
}

# Scale images
for key in pieces:
    pieces[key] = pygame.transform.scale(pieces[key], (square_size,
square_size))

# Draw the chessboard
def draw_board(board):
    screen.fill((255, 255, 255))
    for row in range(8):
        for col in range(8):
            color = (240, 217, 181) if (row + col) % 2 == 0 else (181, 136,
99)
            pygame.draw.rect(screen, color, pygame.Rect(col * square_size,
row * square_size, square_size, square_size))
    piece_map = board.piece_map()
    for square, piece in piece_map.items():
        row, col = divmod(square, 8)
        row = 7 - row  # Flip row for display
        screen.blit(pieces[piece.symbol()], (col * square_size, row *
square_size))
    pygame.display.flip()

def get_square_from_click(x, y):
    col = x // square_size
    row = y // square_size
    row = 7 - row
    return chess.square(col, row)

def board_to_matrix(board):
    matrix = np.zeros((8, 8), dtype=int)
    for square in chess.SQUARES:
        piece = board.piece_at(square)
        row = chess.square_rank(square)
        col = chess.square_file(square)
        matrix[row, col] = piece_to_int(piece)
    return matrix

def piece_to_int(piece):
    if piece is None:
        return 0
    piece_dict = {
        chess.PAWN: 1,
        chess.KNIGHT: 2,
        chess.BISHOP: 3,
        chess.ROOK: 4,
        chess.QUEEN: 5,
        chess.KING: 6
    }
    color = 1 if piece.color == chess.WHITE else -1
    return color * piece_dict[piece.piece_type]

def handle_promotion_with_gui(board, move, square_size):
    print("Початок функції промоції")
    promotion_window = pygame.display.set_mode((4 * square_size,
square_size))
    promotion_window.fill((200, 200, 200))
    promotion_pieces = ['q', 'r', 'b', 'n']
```

```
    images = {
        'q': pygame.image.load('images/white_queen.png'),
        'r': pygame.image.load('images/white_rook.png'),
        'b': pygame.image.load('images/white_bishop.png'),
        'n': pygame.image.load('images/white_knight.png'),
    }
    for i, piece in enumerate(promotion_pieces):
        img = pygame.transform.scale(images[piece], (square_size,
square_size))
        promotion_window.blit(img, (i * square_size, 0))
    pygame.display.flip()

    selected_piece = None
    while not selected_piece:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                exit()
            if event.type == pygame.MOUSEBUTTONDOWN:
                x, _ = event.pos
                index = x // square_size
                if 0 <= index < len(promotion_pieces):
                    selected_piece = promotion_pieces[index]
    move.promotion = {'q': chess.QUEEN, 'r': chess.ROOK, 'b': chess.BISHOP,
'n': chess.KNIGHT}[selected_piece]
    print("Кінець функції промоції")
    promotion_window = pygame.display.set_mode((square_size * 8, square_size
* 8))  # Повернення до розмірів дошки
    draw_board(board)  # Перемальовування дошки
    pygame.display.flip()
    print("Дошку перемальовано")

def move_to_vector(move):
    """Кодування ходу у форматі (from, to)."""
    return [move.from_square, move.to_square]

@register_keras_serializable(package="Custom", name="Lookahead")
class Lookahead(Optimizer):
    def __init__(self, optimizer, learning_rate=0.001, sync_period=5,
slow_step=0.5, name="Lookahead", **kwargs):

        # Спочатку виклик базового конструктора
        super(Lookahead, self).__init__(name=name,
learning_rate=learning_rate, **kwargs)

        # Ініціалізація параметрів Lookahead
        self.optimizer = optimizer
        self.sync_period = sync_period
        self.slow_step = slow_step
        self.counter = 0
        self._slow_weights = []

    def _create_slots(self, var_list):
        self.optimizer._create_slots(var_list)
        for var in var_list:
            self._slow_weights.append(tf.Variable(var, trainable=False,
dtype=var.dtype))

    def apply_gradients(self, grads_and_vars, **kwargs):

        grads_and_vars = list(grads_and_vars)  # Конвертуємо zip-об'єкт у
список
```

```
        # Перевірка вхідних даних
        if not grads_and_vars or not all(len(pair) == 2 for pair in
grads_and_vars):
            raise ValueError(
                "Invalid grads_and_vars: Expected a list of (gradient,
variable) pairs."
            )

        # Підтримка синхронізації Lookahead
        self.counter += 1
        if not self._slow_weights:
            self._slow_weights = [tf.identity(w) for _, w in grads_and_vars]

        # Застосування градієнтів до базового оптимізатора
        self.optimizer.apply_gradients(grads_and_vars)

        # Lookahead: синхронізація ваг
        if self.counter % self.sync_period == 0:
            for (g, w), slow_w in zip(grads_and_vars, self._slow_weights):
                slow_w.assign(slow_w + self.slow_step * (w - slow_w))
                w.assign(slow_w)

    def get_config(self):
        config = {
            "optimizer": tf.keras.optimizers.serialize(self.optimizer),
            "sync_period": self.sync_period,
            "slow_step": self.slow_step,
        }
        base_config = super().get_config()
        return {**base_config, **config}

def get_ai_move(model, board):
    """Отримати хід від нейромережі для поточної позиції."""
    # Перетворення дошки на матрицю
    board_matrix = board_to_matrix(board)
    print("Матриця дошки:", board_matrix)

    # Передбачення моделі
    prediction = model.predict(board_matrix.reshape(1, 8, 8), verbose=0)[0]
    print("Передбачення моделі:", prediction)

    # Легальні ходи
    legal_moves = list(board.legal_moves)
    print("Модель шукає легальний хід")
    # Перетворимо легальні ходи в індекси для моделі
    legal_indices = []
    for move in legal_moves:
        move_vector = move_to_vector(move)
        legal_indices.append(move_vector[1])  # Індекс у форматі моделі

    # Фільтруємо передбачення лише для легальних ходів
    legal_predictions = [prediction[idx] if idx in legal_indices else -1 for
idx in range(len(prediction))]

    # Обираємо найімовірніший хід серед легальних
    best_index = np.argmax(legal_predictions)
    if legal_predictions[best_index] == -1:
        print("ШІ не зміг знайти відповідний хід серед легальних!")
        return None
    print("Модель знайшла легальний хід")
    # Повертаємо відповідний легальний хід
    return legal_moves[legal_indices.index(best_index)]
```

```
def play_with_gui(model):
    board = chess.Board()
    selected_square = None
    running = True
    while running and not board.is_game_over():
        draw_board(board)
        pygame.display.flip()
        if board.turn == chess.WHITE:  # Хід гравця
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    exit()
                if event.type == pygame.MOUSEBUTTONDOWN:
                    x, y = event.pos
                    square = get_square_from_click(x, y)

                    if selected_square is None:  # Вибір фігури
                        if board.piece_at(square) and
board.piece_at(square).color == chess.WHITE:
                            selected_square = square
                            print("Фігуру вибрано")
                    else:  # Спроба зробити хід
                        piece = board.piece_at(selected_square)
                        move = chess.Move(selected_square, square)

                        # Якщо це пішак, який досягає останнього ряду, додати
параметр промоції
                        if piece and piece.piece_type == chess.PAWN and
chess.square_rank(square) in [0, 7]:
                            print("Пішак на базу? Виклик функції промоції")
                            handle_promotion_with_gui(board, move,
square_size)
                            move.promotion = chess.QUEEN  # Тимчасово
встановити ферзя за замовчуванням

                        # Перевірка на легальний хід
                        if move in board.legal_moves:
                            board.push(move)
                            print("Хід виконано")
                        else:
                            print("Вибрано нелегальний хід")

                        selected_square = None
        else:  # Хід ШІ
            ai_move = get_ai_move(model, board)
            if ai_move:
                # Додати промоцію для пішака, якщо ШІ досягає останнього ряду
                if board.piece_at(ai_move.from_square).piece_type ==
chess.PAWN and chess.square_rank(ai_move.to_square) in [0, 7]:
                    ai_move.promotion = chess.QUEEN
                board.push(ai_move)
            else:
                running = False
    pygame.quit()

play_with_gui(model)


import chess
import chess.pgn
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
```

```
from tensorflow.keras.optimizers import Adam

# Функції для обробки шахівниці та ходів
def board_to_matrix(board):
    """Перетворює шахівницю у матрицю розміром 8x8 з кодуванням фігур."""
    matrix = np.zeros((8, 8), dtype=int)
    piece_map = board.piece_map()
    for square, piece in piece_map.items():
        row, col = divmod(square, 8)
        matrix[row, col] = piece_to_int(piece)
    return matrix

def piece_to_int(piece):
    """Кодування фігури у число."""
    piece_dict = {
        chess.PAWN: 1, chess.KNIGHT: 2, chess.BISHOP: 3,
        chess.ROOK: 4, chess.QUEEN: 5, chess.KING: 6
    }
    value = piece_dict[piece.piece_type]
    return value if piece.color == chess.WHITE else -value

def move_to_vector(move):
    """Кодування ходу у форматі (from, to)."""
    return [move.from_square, move.to_square]

# Зчитування PGN-файлу
def parse_pgn_file(file_path):
    games = []
    with open(file_path, "r") as pgn_file:
        while True:
            game = chess.pgn.read_game(pgn_file)
            if game is None:
                break
            games.append(game)
    return games

def extract_features_and_labels(games):
    features = []
    labels = []
    for game in games:
        board = game.board()
        for move in game.mainline_moves():
            # Матриця для позиції перед ходом
            position = board_to_matrix(board)
            features.append(position)

            # Хід, який потрібно передбачити
            move_vector = move_to_vector(move)
            labels.append(move_vector[1])  # `move_to_vector` повертає
(звідки, куди), беремо лише "куди".

            board.push(move)  # Зробити хід на дошці
    return np.array(features), np.array(labels).flatten()


# Створення моделі
def create_model():
    model = Sequential([
        Flatten(input_shape=(8, 8)),  # Розплющення матриці
        Dense(128, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
```

```
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),

        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),
        Dense(64, activation="relu"),

        Dense(4096, activation="softmax")  # 64 клітинок * 64 клітинок
    ])
    model.compile(optimizer=Adam(learning_rate=0.001),
loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return model

# Навчання моделі
def train_model(games):
    print(f"Loaded {len(games)} games for training.")

    # Підготувати дані
    features, labels = extract_features_and_labels(games)
    print("Features shape:", features.shape)  # Повинно бути (кількість
ходів, 8, 8)
    print("Labels shape:", labels.shape)    # Повинно бути (кількість ходів,)

    # Створити та тренувати модель
    model = create_model()
    model.fit(features, labels, epochs=10, batch_size=32)

    # Зберегти модель
    model.save("chess_model_nk_2.keras")
    print("Model trained and saved")

if __name__ == "__main__":
    pgn_files = [
        "pgn_games/Abdusattorov.pgn",
        "pgn_games/Adams.pgn",
        "pgn_games/Anand.pgn",
        "pgn_games/Andersson.pgn",
        "pgn_games/Andreikin.pgn",
        "pgn_games/Anand.pgn",
        "pgn_games/Andersson.pgn",
        "pgn_games/Andreikin.pgn",
        "pgn_games/Bacrot.pgn",
        "pgn_games/Carlsen.pgn",
        "pgn_games/Caruana.pgn",
        "pgn_games/Duda.pgn",
        "pgn_games/Ehlvest.pgn",
        "pgn_games/Eljanov.pgn",
        "pgn_games/Erigaisi.pgn",
        "pgn_games/Firouzja.pgn",
        "pgn_games/Gelfand.pgn",
        "pgn_games/Georgiev.pgn",
        "pgn_games/Giri.pgn",
        "pgn_games/Grischuk.pgn",
        "pgn_games/GurevichD.pgn",
        "pgn_games/Hort.pgn",
```

```
        "pgn_games/Ivanchuk.pgn",
        "pgn_games/Jobava.pgn",
        "pgn_games/Kamsky.pgn",
        "pgn_games/Karjakin.pgn",
        "pgn_games/Karpov.pgn",
        "pgn_games/Korchnoi.pgn",
        "pgn_games/Korobov.pgn",
        "pgn_games/Kosteniuk.pgn",
        "pgn_games/Kramnik.pgn"

    ]

    all_games = []
    for pgn_file_path in pgn_files:
        with open(pgn_file_path, "r") as pgn_file:
            while True:
                game = chess.pgn.read_game(pgn_file)
                if game is None:
                    break
                all_games.append(game)

    print(f"Загальна кількість ігор для тренування: {len(all_games)}")
    train_model(all_games)


import tensorflow as tf
from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Optimizer
import chess.pgn
import chess
import numpy as np


def parse_pgn_file(file_path):
    """Парсинг шахових партій з PGN-файлу."""
    with open(file_path, "r") as pgn_file:
        games = []
        while True:
            game = chess.pgn.read_game(pgn_file)
            if game is None:
                break
            games.append(game)
    return games

def board_to_matrix(board):
    """Перетворює шахівницю у матрицю розміром 8x8 з кодуванням фігур."""
    matrix = np.zeros((8, 8), dtype=int)
    piece_map = board.piece_map()
    for square, piece in piece_map.items():
        row, col = divmod(square, 8)
        matrix[row, col] = piece_to_int(piece)
    return matrix

def piece_to_int(piece):
    """Кодування фігури у число."""
    piece_dict = {
        chess.PAWN: 1, chess.KNIGHT: 2, chess.BISHOP: 3,
        chess.ROOK: 4, chess.QUEEN: 5, chess.KING: 6
    }
    value = piece_dict[piece.piece_type]
    return value if piece.color == chess.WHITE else -value
```

```
def move_to_vector(move):
    """Кодування ходу у форматі (from, to)."""
    return [move.from_square, move.to_square]

def extract_features_and_labels(games):
    features = []
    labels = []
    for game in games:
        board = game.board()
        for move in game.mainline_moves():
            # Матриця для позиції перед ходом
            position = board_to_matrix(board)
            features.append(position)

            # Хід, який потрібно передбачити
            move_vector = move_to_vector(move)
            labels.append(move_vector[1])  # `move_to_vector` повертає
(звідки, куди), беремо лише "куди".

            board.push(move)  # Зробити хід на дошці
    return np.array(features), np.array(labels).flatten()

def preprocess_games(all_games):
    x_train = []
    y_train = []

    for game in all_games:
        board_states, moves = extract_features_and_labels(game)
        x_train.extend(board_states)
        y_train.extend(moves)

    # Конвертуємо у numpy-матриці
    x_train = np.array(x_train).reshape(-1, 8, 8)  # 8x8 шахівниця
    y_train = np.array(y_train)

    return x_train, y_train

@register_keras_serializable(package="Custom", name="Lookahead")
class Lookahead(Optimizer):
    def __init__(self, optimizer, learning_rate=0.001, sync_period=5,
slow_step=0.5, name="Lookahead", **kwargs):
        # Спочатку виклик базового конструктора
        super(Lookahead, self).__init__(name=name,
learning_rate=learning_rate, **kwargs)

        # Ініціалізація параметрів Lookahead
        self.optimizer = optimizer
        self.sync_period = sync_period
        self.slow_step = slow_step
        self.counter = 0
        self._slow_weights = []

    def _create_slots(self, var_list):
        self.optimizer._create_slots(var_list)
        for var in var_list:
            self._slow_weights.append(tf.Variable(var, trainable=False,
dtype=var.dtype))

    def apply_gradients(self, grads_and_vars, **kwargs):

        grads_and_vars = list(grads_and_vars)  # Конвертуємо zip-об'єкт у
список
```

```
        # Перевірка вхідних даних
        if not grads_and_vars or not all(len(pair) == 2 for pair in
grads_and_vars):
            raise ValueError(
                "Invalid grads_and_vars: Expected a list of (gradient,
variable) pairs."
            )

        # Підтримка синхронізації Lookahead
        self.counter += 1
        if not self._slow_weights:
            self._slow_weights = [tf.identity(w) for _, w in grads_and_vars]

        # Застосування градієнтів до базового оптимізатора
        self.optimizer.apply_gradients(grads_and_vars)

        # Lookahead: синхронізація ваг
        if self.counter % self.sync_period == 0:
            for (g, w), slow_w in zip(grads_and_vars, self._slow_weights):
                slow_w.assign(slow_w + self.slow_step * (w - slow_w))
                w.assign(slow_w)

    def get_config(self):
        config = {
            "optimizer": tf.keras.optimizers.serialize(self.optimizer),
            "sync_period": self.sync_period,
            "slow_step": self.slow_step,
        }
        base_config = super().get_config()
        return {**base_config, **config}

def load_existing_model(model_path):
    """Завантаження існуючої моделі."""
    return load_model(model_path)

def finetune_model(model_path, pgn_files):
    """Донавчання існуючої моделі з алгоритмом Lookahead."""
    # Завантаження існуючої моделі
    model = load_existing_model(model_path)

    # Збір даних з усіх PGN-файлів
    all_games = []
    for file in pgn_files:
        all_games.extend(parse_pgn_file(file))

    # Попередня обробка даних
    x_data, y_data = preprocess_games(all_games)

    # Масиви x_data вже мають форму (n_samples, 8, 8), а y_data — одномірний
масив
    print(f"Розмірність x_data: {x_data.shape}")
    print(f"Розмірність y_data: {y_data.shape}")

    # Створення Lookahead-оптимізатора
    base_optimizer = Adam(learning_rate=0.001)
    lookahead_optimizer = Lookahead(optimizer=base_optimizer, sync_period=5,
slow_step=0.5)

    # Компіляція моделі з Lookahead
    model.compile(optimizer=lookahead_optimizer,
loss="sparse_categorical_crossentropy", metrics=["accuracy"])
```

```
    # Колбек для ранньої зупинки
    early_stopping = EarlyStopping(monitor="loss", patience=3)

    # Донавчання моделі
    model.fit(
        x_data,
        y_data,
        batch_size=32,
        epochs=10,
        callbacks=[early_stopping],
    )

    # Збереження оновленої моделі
    updated_model_path = "chess_model_lookahead.keras"
    model.save(updated_model_path)
    print(f"Оновлена модель збережена у: {updated_model_path}")

if __name__ == "__main__":
    # Шлях до існуючої моделі
    existing_model_path = "chess_model_19k.keras"
    base_optimizer = Adam(learning_rate=0.001)

    # Список файлів PGN
    pgn_files = [
        "pgn_games/Bacrot.pgn",
        "pgn_games/Carlsen.pgn",
        "pgn_games/Caruana.pgn",
        "pgn_games/Duda.pgn",
    ]

    # Виклик функції донавчання
    finetune_model(existing_model_path, pgn_files)
```