Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

Факультет електроніки та комп’ютерних технологій

Курсова робота
на тему
«Алгоритм оптимізації багатошарових нейронних мереж і хаос у них»

Виконав:
студент групи ФеІм - 53
Сметюх Андрій
Науковий керівник:
доц. Куньо І. М.

Львів — 2023

Зміст

Вступ 3
1. Розробка багатошарової нейронної мережі 4
1.1. Теоретичні відомості 4
1.2. Предметна область 5
1.3. Моделювання нейронної мережі 6
1.4. Зворотнє поширення помилки 7
1.5. Алгоримти оптимізації навчання нейронної мережі 8
1.6. Хаос у нейронних мережах 9
2. Розробка програмного забезпечення 11
2.1. База даних 11
2.2. Програмне забезпечення 12
2.3. Збір тренувальних даних 13
2.4. Метод градієнтного спуску 13
3. Результати роботи 15
Висновок 17
Додатки 18
Посилання 18
Лістинг програми 19

Вступ

У світі швидкого розвитку технологій та зростаючої потреби в аналізі та обробці великих обсягів даних, нейромережі стають все більш потужним інструментом для розв'язання складних завдань. Вони можуть використовуватись для розпізнавання образів, класифікації даних, прогнозування та багатьох інших завдань, які вимагають аналізу та витягування інформації з великих наборів даних.

Проєктування нейромережі є ключовим етапом у розробці імплементації інтелектуальних систем, але успішність цих систем значною мірою залежить від ефективності алгоритмів оптимізації, які використовуються для навчання та покращення нейромережі. Вибір відповідного алгоритму оптимізації може визначити якість навчання, швидкість збіжності та здатність нейромережі до уникнення перенавчання.

Ця курсова робота присвячена дослідженню проєктування нейромережі та перевірці ефективності алгоритму оптимізації. Метою роботи є аналіз алгоритму оптимізації та його впливу на результати навчання нейромережі. Основний акцент буде зроблено на алгоритмі зворотнього поширення помилки та його ефективність при роботі з великим об’ємом тренувальних даних.

У цій роботі будуть використані мови програмування та інструменти для реалізації нейромережі, проведення навчання та оцінки її ефективності. Після реалізації нейромережі будуть проведені експерименти з різними алгоритмами оптимізації для порівняння їх продуктивності та результатів навчання. Аналіз отриманих результатів дозволить оцінити ефективність та вплив різних алгоритмів оптимізації на навчання нейромережі.

Виконання цієї курсової роботи дозволить глибше розуміти процес проєктування нейромережі та вибір оптимального алгоритму оптимізації. Отримані результати можуть бути корисні для досліджників та розробників, які працюють з нейромережами і мають інтерес до покращення їхньої продуктивності та точності.

1. Розробка багатошарової нейронної мережі

1.1. Теоретичні відомості

Нейромережа — це математична модель, яка імітує структуру та функціонування біологічних нейронних мереж з метою вирішення різноманітних задач, таких як класифікація, регресія, прогнозування та генерація. В основі нейромереж лежать штучні нейрони, які об'єднуються в графові структури і передають сигнали один одному через ваги зв'язків. Завдяки процесу навчання, під час якого ваги та зміщення між нейронами оптимізуються, нейромережі стають здатними до виявлення закономірностей та залежностей у вхідних даних.

Структура нейромережі складається з трьох основних типів шарів: вхідний шар, приховані шари та вихідний шар.

- Вхідний шар приймає дані ззовні, наприклад, зображення або текст, і передає їх у наступні шари. Вхідний шар не змінює дані, а лише служить точкою входу для них.

- Приховані шари забезпечують обробку вхідних даних і передачу інформації між шарами. Вони називаються "прихованими", оскільки їхні результати не відображаються напряму на виході мережі. Кількість прихованих шарів та нейронів у них може варіюватися в залежності від складності задачі та архітектури мережі.

- Вихідний шар формує результат, який нейромережа передбачає на основі вхідних даних. Результат може бути класифікацією, числовим значенням або іншою інформацією, залежно від типу задачі.

Якщо нейромережа містить лише один прихований шар, то вона називається одношаровою. Якщо ж більше одного – то багатошаровою. У даній курсовій роботи ми будемо працювати з багатошаровою нейромережею.

Усі нейрони в шарах з'єднані між собою через ваги зв'язків. Ваги відіграють важливу роль у навчанні нейромережі, оскільки вони визначають силу впливу

одного нейрона на інший. В процесі навчання, ваги оптимізуються, щоб мінімізувати помилку передбачення мережі.

Крім ваг зв'язків, кожен нейрон має так зване зміщення (bias), яке дозволяє регулювати активацію нейрона незалежно від вхідного сигналу. Зміщення допомагає нейромережі легше адаптуватися до різних даних та виконувати більш гнучкі перетворення на вхідних даних.

Ще одним ключовим компонентом нейромережі є функція активації. Вона застосовується до кожного нейрона у прихованих та вихідних шарах, щоб визначити його активність на основі суми вхідних сигналів, помножених на відповідні ваги та додавання зміщення. Функція активації може бути лінійною або не лінійною, в залежності від типу задачі та архітектури мережі. Деякі з найпопулярніших функцій активації включають сигмоїду, гіперболічний тангенс, ReLU (Rectified Linear Unit) та Softmax.

Навчання нейромережі полягає в оптимізації ваг зв'язків та зміщень на основі навчального набору даних. Для цього зазвичай використовують метод зворотного поширення помилки (backpropagation), який базується на градієнтному спуску. Процес навчання може тривати довгий час, залежно від розміру набору даних, архітектури мережі та складності задачі.

1.2. Предметна область

Предметною областю розробки нейромережі мною було обрано прогнозування спортивних матчів із бейсболу на основі результатів ігор у минулому.

MLB (Major League Baseball, «Головна бейсбольна ліга») – це ліга професійного бейсболу в США, яка включає в свій склад 30 команд. Заснована ще 1876 році, ліга практично ніколи не змінювала правил гри. Також у кожному сезоні кожна з команд станом на 2023 рік проводить по 162 матчі, що загалом становить 2430 ігор щороку. Тобто, загальна кількість матчів, яка була зіграна за останні понад 100 років, становить близько 200 000 матчів. Це є дуже велике значення в контексті навчання нейронної мережі.

1.3. Моделювання нейронної мережі

У процесі дослідження було визначено дані, які будуть сприйматися як ознаки для вхіжного шару. Для зручності будемо позначати «Команда 1» - це команда, яка грає вдома; «Команда 2» - це команда, яка грає на виїзді. До цих ознак належать:

1) Частка перемог Команди 1 перед матчем;
2) Середня кількість набраних пробіжок Командою 1;
3) Середня кількість пропущених пробіжок Командою 1;
4) Частка перемог Команди 1 в домашніх матчах;
5) Середня кількість набраних пробіжок Командою 1 в домашніх матчах;
6) Середня кількість пропущених пробіжок Командою 1 в домашніх матчах;
7) Частка перемог Команди 2 перед матчем;
8) Середня кількість набраних пробіжок Командою 2;
9) Середня кількість пропущених пробіжок Командою 2;
10) Частка перемог Команди 2 у виїзних матчах;
11) Середня кількість набраних пробіжок Командою 2 у виїзних матчах;
12) Середня кількість пропущених пробіжок Командою 2 у виїзних матчах;

Вихідний шар складається лише з одного виводу, який може набувати значення r = [0; 1], де 0 – прогнозована перемога Команди 2; 1 – прогнозована перемога Команди 1.

Також було обрано 3 приховані шари, в кожному з яких є по 12 нейронів.

Отже, загальний вигляд нейронної мережі для виконання даної задачі, зображено на рис. 1

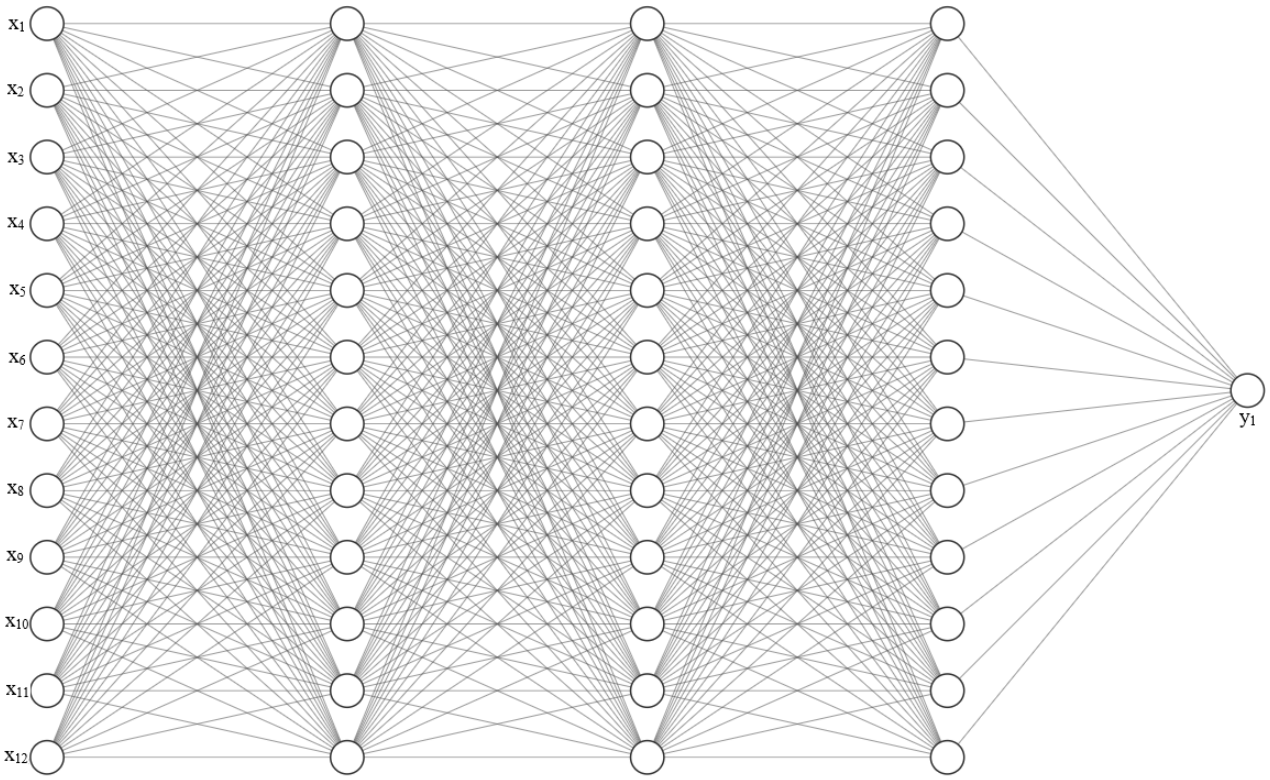

Рис. 1. Схема нейронної мережі для розв’язання задачі

1.4. Зворотнє поширення помилки

Важливим алгоритмом начання нейромережі є алгоритм зворотнього поширення помилки (backpropagation). Він є ключовим алгоритмом для навчання багатошарових нейронних мереж. Його основна ідея полягає в обчисленні та розповсюдженні градієнтів помилки назад по мережі з метою оновлення ваг та зсувів шарів.

Процес зворотнього поширення помилки зазвичай складається з таких кроків:

1) Прямий прохід (forward pass): Вхідні дані передаються через мережу, і активації шарів обчислюються по шляху від початкового шару до вихідного шару.

2) Обчислення помилки (error calculation): Порівнюються вихідні значення, обчислені на прямому проході, з бажаними вихідними значеннями. На основі цієї різниці визначається помилка мережі.

3) Зворотнє поширення помилки (backward pass): Починаючи з вихідного шару, градієнт помилки обчислюється для кожного шару, використовуючи ланцюжкове правило диференціювання. Градієнти помилки передаються назад по мережі.

4) Оновлення ваг та зсувів (weight and bias update): За допомогою обчислених градієнтів помилки ваги та зсуви кожного шару оновлюються з використанням певного оптимізаційного алгоритму, такого як стохастичний градієнтний спуск.

Цей процес повторюється для кожного навчального прикладу у тренувальному наборі з метою мінімізації помилки мережі та покращення її прогнозувальної здатності. Зворотнє поширення помилки є ефективним способом навчання багатошарових нейронних мереж та відіграє важливу роль у їх успішному функціонуванні.

1.5. Алгоритми оптимізації начання нейронної мережі

Існує кілька алгоритмів оптимізації, які використовуються для навчання та покращення нейромереж. Ось кілька з них[1]:

1) Алгоритм зворотнього поширення помилки (Backpropagation): Це найпоширеніший алгоритм оптимізації для навчання нейромереж. Він базується на обчисленні градієнтів функції втрати від виходів мережі до входів і використанні цих градієнтів для оновлення ваг та зсувів мережі. Процес повторюється на кожному навчальному прикладі для кожної ітерації навчання.

$$\theta = \theta - \eta \nabla_\theta J(\theta)$$

На кожному кроці рахується градієнт функції втрат, який множиться на параметр навчання η, після чого ці параметри змінюються на відповідне значення. Основним недоліком цього алгоритму є можливість потрапити в локальний мінімум, але не дістатися до глообального.

2) Метод стохастичного градієнтного спуску (Stochastic Gradient Descent, SGD): Цей метод є варіацією алгоритму зворотнього поширення помилки, де оновлення ваг та зсувів здійснюється після кожного окремого навчального прикладу, а не після всієї навчальної вибірки. Це дозволяє швидше знаходити локальні мінімуми, але може бути шумним і приводити до меншої стійкості

навчання. Також цей алгоритм дозволяє краще оминати локальні мінімуми, а знаходити глобальні.

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

3) Міні-пакетний градієнтний спуск. Цей алгоритм суміщує в собі переваги обидвох попередніх методів, оскільки нейромережа швидше навчається а також з більшою ймовірністю проходить локальні мінімуми.

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

4) Варіації градієнтного спуску: Існує кілька варіацій градієнтного спуску, таких як метод моменту (Momentum), метод Адама (Adam), адаптивний градієнтний спуск (AdaGrad), RMSprop і т.д. Ці методи використовують різні стратегії адаптації кроку навчання або розподілу градієнтів для ефективнішого навчання та збільшення швидкості збіжності.

5) Еволюційні алгоритми: Деякі варіанти нейроеволюції використовують генетичні алгоритми або еволюційні стратегії для пошуку оптимальних наборів ваг та зсувів. Ці алгоритми використовують елементи випадковості та вибірковості для знаходження кращих рішень.

6) Регуляризація: Це техніка, яка допомагає запобігти перенавчанню (overfitting) шляхом додаткових обмежень на ваги нейронів. Популярні методи регуляризації включають L1 та L2 регуляризацію, відомі також як Lasso і Ridge відповідно.

Ці алгоритми є лише деякими з багатьох доступних методів оптимізації нейромереж. Вибір конкретного алгоритму залежить від конкретної задачі, розміру та складності нейромережі, а також від експериментів з результатами навчання.

1.6. Хаос у нейронних мережах

Хаос у нейронних мережах відноситься до небажаного стану, коли модель стає непередбачуваною, нестабільною або неспроможною ефективно вирішувати задачу. В хаотичному стані нейронна мережа може вести себе неочікувано та втратити свою здатність до зваженого та точного розпізнавання патернів у вхідних даних.

Хаос може виникати з різних причин у нейронних мережах:

1) Недостатній обсяг навчальних даних: Якщо модель навчається на обмеженому обсязі даних або даних, що не представляють реальну різноманітність, то вона може не мати достатньої інформації для коректного узагальнення та передбачення на нових вхідних даних.

2) Неправильне налаштування гіперпараметрів: Гіперпараметри, такі як швидкість навчання, розмір шарів, кількість прихованих одиниць тощо, потребують належного налаштування. Якщо ці параметри встановлені неправильно, може виникнути хаос, і модель може навчитися погано або навіть не навчитися взагалі.

3) Перенавчання (overfitting): Це стан, коли модель навчається дуже докладно "запам'ятовуючи" тренувальні дані, але втрачає здатність узагальнювати на нові дані. Це може призвести до хаотичних результатів на невідомих даних.

4) Недонавчання (underfitting): Це стан, коли модель недостатньо навчається на тренувальних даних та не може ефективно розпізнавати або передбачати патерни.

Для запобігання хаосу в нейронних мережах можна використовувати різні стратегії, такі як оптимальний вибір гіперпараметрів, регуляризація, використання більш репрезентативних та різноманітних навчальних даних, а також методи ранньої зупинки навчання.

2. Розробка програмного забезпечення

2.1. База даних

У процесі розробки ПЗ було також спроєктована база даних. Вона складається з трьох таблиць:

1) Seasons – містить інформацію про сезони ліги MLB;
2) Teams – містить інформацію про команди MLB;
3) Games – містить інформацію про кожен матч (дата, результат, усі дані, що використовуються як вхідні дані для нейромережі);

Діаграму бази даних можна побачити на рис. 2.1.

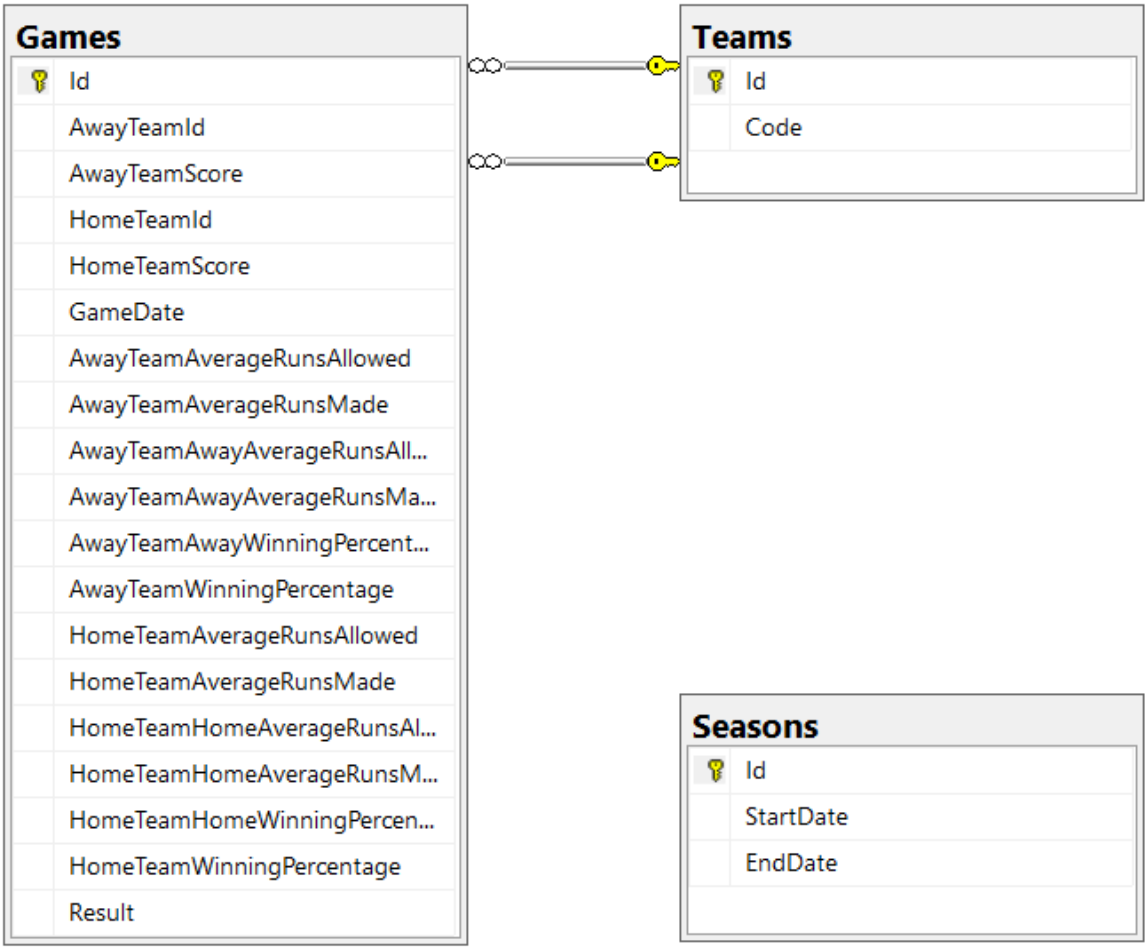

Рис. 2.1. Діаграма бази даних

2.2. Програмне забезпечення

Програмне забезпечення для реалізації роботи нейромережі написане на мові C# як консольна програма за допомогою програми Microsoft Visual Studio 2022.

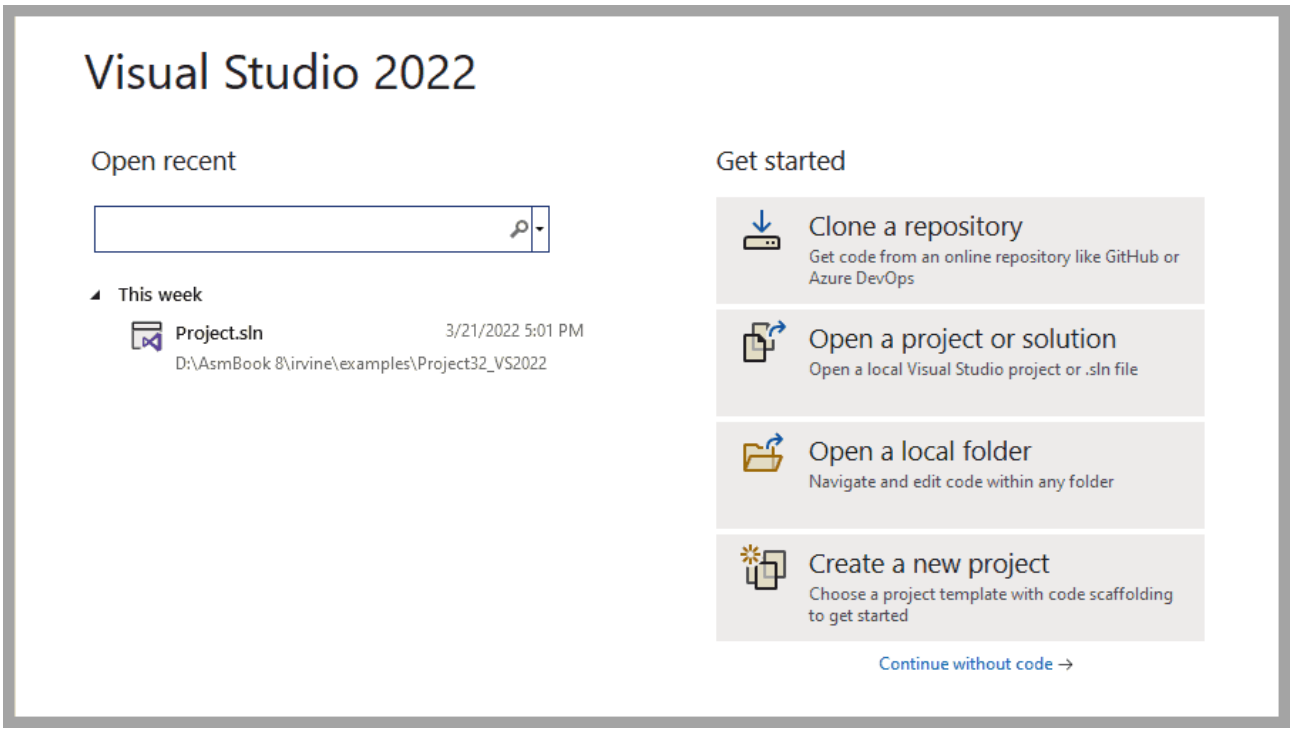

Рис. 2.2. Початкове вікно Visual Studio 2022

Visual Studio 2022 дозволяє створювати програмні продукти на мовах C, C++, C#, F# та багатьох інших. Це є потужний інструмент для розробки та відлагодження програм.

Для роботи з базою даних було обрано фреймворк Entity Framework.

Entity Framework (EF) - це технологія доступу до даних для платформи .NET, яка надає рівень абстракції над базами даних і дозволяє зосередитись на роботі з об'єктами даних замість написання SQL-запитів. Використання Entity Framework у курсовій роботі може бути корисним для реалізації доступу до бази даних та роботи з даними в зручній об'єктно-орієнтованій парадигмі.

Основні переваги Entity Framework:

1) Об'єктно-орієнтована модель: Entity Framework дозволяє працювати з базою даних, використовуючи об'єкти класів, що спрощує розробку та підтримку коду.

2) Автоматичне відображення: Entity Framework автоматично створює відображення між базою даних і класами об'єктів, що дозволяє автоматично генерувати SQL-запити на основі взаємодії з об'єктами.

3) Міграції бази даних: EF надає можливість автоматичного оновлення структури бази даних на основі змін в моделі даних, що полегшує роботу з еволюцією схеми бази даних.

4) Підтримка різних провайдерів баз даних: Entity Framework підтримує різні провайдери баз даних, такі як SQL Server, MySQL, Oracle, SQLite та інші, що дає можливість легко переходити між різними системами управління базами даних.

5) Запити LINQ: Entity Framework дозволяє використовувати мову запитів LINQ (Language Integrated Query) для створення потужних та зручних запитів до бази даних без прямої роботи з SQL.

2.3. Збір тренувальних даних

В якості тренувальних даних було зібрано всю статистику про матчі ліги MLB за 1871 по 2022 роки[4]. Також була написана спеціальна програма, яка зчитує ці дані та зберігає в базі даних. У результаті було отримано інформацію про 192 335 бейсбольних матчів. Усі вони були обрані як тренувальні, оскільки за понад 150-річну історію практично не відбувалося змін правил гри або її вадливих компонентів, тому усі отримані результати є значимими.

2.4. Метод градієнтного спуску

Для прикладу було обрано алгоритм градієнтного спуску.

Метод градієнтного спуску є одним з основних алгоритмів оптимізації, який використовується при навчанні нейромереж. Він дозволяє знайти локальний мінімум (або максимум) функції втрат шляхом ітеративного оновлення параметрів нейромережі в напрямку найшвидшого спаду функції втрат.

Основна ідея методу градієнтного спуску полягає в обчисленні градієнта функції втрат відносно ваг та зсувів нейромережі і використанні цього градієнта для керування оновленням параметрів. Градієнт показує напрямок найшвидшого зростання функції втрат, тому для знаходження мінімуму необхідно рухатись в протилежному напрямку.

Алгоритм градієнтного спуску можна описати наступними кроками:

1) Ініціалізувати ваги та зсуви нейромережі випадковими значеннями або за допомогою попередньо обчислених значень.

2) Обчислити значення вихідного шару нейромережі, прямо поширюючи вхідні дані через шари нейронів.

3) Обчислити значення функції втрат, яка відображає різницю між очікуваними та фактичними виходами нейромережі.

4) Обчислити градієнти функції втрат відносно ваг та зсувів шляхом зворотного поширення помилки в нейромережі.

5) Оновити ваги та зсуви, використовуючи градієнти та швидкість навчання (learning rate). Швидкість навчання визначає, наскільки швидко параметри нейромережі оновлюються на кожній ітерації.

6) Повторити кроки 2-5 до досягнення критерію зупинки, наприклад, заданої кількості ітерацій або досягнення деякого епсілон (мінімальна зміна функції втрат).

Цей процес повторюється на кожній ітерації навчання для знаходження оптимальних значень ваг та зсувів, що мінімізують функцію втрат. Чим більш точно вдається оновлювати параметри нейромережі, тим більш точнішим стає її вихід та здатність розв'язувати задачу.

Проте, важливо враховувати, що метод градієнтного спуску може застрягати в локальних мінімумах, що може унеможливити досягнення глобального оптимуму. Для подолання цього, існують різні модифікації методу, такі як стохастичний градієнтний спуск та методи з моментом.

Узагальнюючи, метод градієнтного спуску є потужним інструментом для оптимізації нейромереж, проте його ефективність може залежати від вибору правильної швидкості навчання, архітектури нейромережі та прийнятих припущень.

3. Результати роботи

Для перевірки роботи було обрано 15 матчів, які відбулися 26 травня 2023 року[5]. Результати передбачень нейронної мережі порівнювалися з результатами, отриманими в результаті навчання лінійною регресією. Результати можна побачити на табл. 1

Команда 1	Команда 2	Прогноз лінійної регресії	Прогноз нейромережі	Результат
Чикаго Кабс	Цинцинатті Редс	П1 (-)	П1 (-)	П2 (0 - 9)
Детройт Тайгерс	Чикаго Уайт Сокс	П1 (-)	П2 (+)	П2 (3 - 12)
Тампа Бей Рейс	Лос Анджелес Доджерс	П1 (+)	П1 (+)	П1 (9 - 3)
Балтимор Оріолс	Техас Рейнджерс	П1 (-)	П1 (-)	П2 (2 - 12)
Нью-Йорк Янкіс	Сан Дієго Падрес	П1 (-)	П2 (+)	П2 (1 - 5)
Атланта Брейвс	Філадельфія Філліс	П1 (-)	П1 (-)	П2 (4 - 6)
Канзас Сіті Роялс	Вашингтон Нешналс	П1 (-)	П2 (+)	П2 (10 - 12)
Мілуокі Брюерс	Сан Франциско Джаянтс	П1 (-)	П1 (-)	П2 (1 - 15)
Міннесота Твінс	Торонто Блу Джейс	П1 (-)	П1 (-)	П2 (1 – 3)
Колорадо Рокіс	Нью-Йорк Метс	П2 (+)	П2 (+)	П2 (2 - 5)
Лос Анджелес Енджелс	Маямі Марлінс	П1 (-)	П1 (-)	П2 (2 - 6)
Окленд Атлетікс	Х’юстон Астрос	П2 (+)	П2 (+)	П2 (2 – 5)
Клівленд Ґардіанс	Сент Луїс Кардіналс	П1 (+)	П1 (+)	П1 (4 – 3)
Аризона Даймондбекс	Бостон Ред Сокс	П1 (-)	П1 (-)	П2 (2 – 7)
Сієтл Маррінерс	Піттсбург Пайретс	П1 (-)	П1 (-)	П2 (6 – 11)

Табл. 1. Порівняльна таблиця прогнозів та реальних результатів матчів

Висновок

У цій курсовій роботі було проведено дослідження проєктування нейромережі та перевірки ефективності алгоритмів оптимізації. Основними цілями були аналіз різних алгоритмів оптимізації та вивчення їх впливу на результати навчання нейромережі.

У процесі роботи були використані мови програмування та інструменти для реалізації нейромережі, проведення навчання та оцінки її ефективності. Були реалізовані алгоритми зворотнього поширення помилки та стохастичного градієнтного спуску.

Експерименти були проведені з різними алгоритмами оптимізації, аналізувалися їх продуктивність та результати навчання нейромережі. Було з'ясовано, що вибір правильного алгоритму оптимізації має суттєвий вплив на якість навчання та швидкість збіжності нейромережі.

Аналіз отриманих результатів показав, що алгоритм зворотнього поширення помилки є ефективним методом оптимізації для нейромережі. Він дозволяє досягнути заданої точності та швидко збігається до оптимальних значень ваг та зсувів. Стохастичний градієнтний спуск також показав гарні результати, але може бути менш стійким та вимагати більше обчислювальних ресурсів.

Ця робота підкреслила важливість правильного вибору алгоритму оптимізації для досягнення найкращих результатів навчання нейромережі. Для подальших досліджень можна розглянути інші алгоритми оптимізації та їх вплив на різні типи нейромереж та завдань. Отримані в цій роботі знання та результати можуть бути використані в подальшій розробці та вдосконаленні інтелектуальних систем, що базуються на нейромережах.

Розроблена в даній курсовій роботі нейромережа змогла вірно спрогнозувати 7 результатів із 15. У той час звичайна лінійна регресія змогла спрогнозувати лише 4 правильні результати. Тому гіпотеза, що можна легко прогнозувати різні спортивні події, є не зовсім вірною. Можливо надалі необхідно враховувати більше факторів (наприклад, форму команди тощо) або змінити гіперпараметри.

Додатки

Посилання

1. An overview of gradient descent optimization algorithms, Sebastian Ruder Insight Centre for Data Analytics, NUI Galway Aylien Ltd., Dublin
2. Хаотичні стани багатошарової нейронної мережі, С. Свелеба, І. Катеринчук, І. Куньо, І. Карпа, О. Семотюк, Я. Шмигельський, Н. Свелеба, В. Куньо, ISSN 2224-087X. Електроніка та інформаційні технології. 2021. Випуск 16. С. 20–35

3. Specifics of the learning error dependence of multilayered neural networks from the activation function during the process of printed digits, S. Sveleba , I. Katerynchuk , I. Kuno , O. Semotiuk , Ya. Shmyhelskyy and N. Sveleba, ISSN 2224-087X. Електроніка та інформаційні технології. 2022. Випуск 17. С. 36–53
4. https://www.retrosheet.org/gamelogs/gl1871_2022.zip
5. https://www.mlb.com/scores/2023-05-26

Лістинг програми

Файл «NeuralNetwork.cs»

```
public class NeuralNetwork
{
    private double[,] weights1; // weights between input layer and first hidden layer
    private double[,] weights2; // weights between first and second hidden layer
    private double[,] weights3; // weights between second and third hidden layer
    private double[,] weightsOutput; // weights between third hidden layer and output layer
    private double[] biases; // biases for all layers
    private int inputSize = 12;
    private int hiddenSize = 12;
    private int outputSize = 1;
    private double learningRate = 0.001;
```

```
public NeuralNetwork()
{
    // Initialize weights and biases with random values
    weights1 = InitializeWeights(inputSize, hiddenSize);
    weights2 = InitializeWeights(hiddenSize, hiddenSize);
    weights3 = InitializeWeights(hiddenSize, hiddenSize);
    weightsOutput = InitializeWeights(hiddenSize, outputSize);
    biases = InitializeBiases(hiddenSize, 3);
}

private double[,] InitializeWeights(int rows, int cols)
{
    var random = new Random();
    var weights = new double[rows, cols];
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            weights[i, j] = random.NextDouble();
        }
    }
    return weights;
}

private double[] InitializeBiases(int size, int layers)
{
    var random = new Random();
    var biases = new double[size * layers];
    for (int i = 0; i < size * layers; i++)
    {
        biases[i] = random.NextDouble();
    }
    return biases;
}

public double[] FeedForward(double[] inputs)
{
    // Compute activations for the first hidden layer
    var hiddenLayer1 = ComputeLayer(inputs, weights1, biases, hiddenSize);

    // Compute activations for the second hidden layer
    var hiddenLayer2 = ComputeLayer(hiddenLayer1, weights2, biases, hiddenSize);

    // Compute activations for the third hidden layer
    var hiddenLayer3 = ComputeLayer(hiddenLayer2, weights3, biases, hiddenSize);

    // Compute output
    var output = ComputeLayer(hiddenLayer3, weightsOutput, biases, outputSize);

    return output;
}

private double[] ComputeLayer(double[] inputs, double[,] weights, double[] biases, int outputSize)
{
    var output = new double[outputSize];
    for (int i = 0; i < outputSize; i++)
    {
        double sum = 0;
        for (int j = 0; j < inputs.Length; j++)
        {
            sum += inputs[j] * weights[j, i];
        }
        output[i] = Sigmoid(sum + biases[i]);
    }
    return output;
}

private double Sigmoid(double x)
{
    // Using sigmoid activation function
    return 1 / (1 + Math.Exp(-x));
}

private double SigmoidDerivative(double x)
{
    return Sigmoid(x) * (1 - Sigmoid(x));
}

public void Train(double[] inputs, double targetOutput)
{
    double[] output = FeedForward(inputs);
```

```
        double error = targetOutput - output[0];

        // Compute gradients for the output layer
        double outputGradient = SigmoidDerivative(output[0]) * error;

        // Update weights between third hidden layer and output layer
        for (int i = 0; i < hiddenSize; i++)
        {
            weightsOutput[i, 0] += learningRate * outputGradient;
        }

        var hiddenLayer1 = ComputeLayer(inputs, weights1, biases, hiddenSize);
        var hiddenLayer2 = ComputeLayer(hiddenLayer1, weights2, biases, hiddenSize);

        // Compute gradients for the third hidden layer
        double[] hiddenLayer3Gradients = new double[hiddenSize];
        for (int i = 0; i < hiddenSize; i++)
        {
            hiddenLayer3Gradients[i] = outputGradient * weightsOutput[i, 0] * SigmoidDerivative(output[0]);
        }

        // Update weights between second and third hidden layer
        UpdateWeights(hiddenLayer2, hiddenLayer3Gradients, weights3);

        // Compute gradients for the second hidden layer
        double[] hiddenLayer2Gradients = ComputeGradients(hiddenLayer2, hiddenLayer3Gradients, weights3);

        // Update weights between first and second hidden layer
        UpdateWeights(hiddenLayer1, hiddenLayer2Gradients, weights2);

        // Compute gradients for the first hidden layer
        double[] hiddenLayer1Gradients = ComputeGradients(hiddenLayer1, hiddenLayer2Gradients, weights2);

        // Update weights between input layer and first hidden layer
        UpdateWeights(inputs, hiddenLayer1Gradients, weights1);
    }

    private void UpdateWeights(double[] inputs, double[] gradients, double[,] weights)
    {
        for (int i = 0; i < inputs.Length; i++)
        {
            for (int j = 0; j < gradients.Length; j++)
            {
                weights[i, j] += learningRate * gradients[j] * inputs[i];
            }
        }
    }

    private double[] ComputeGradients(double[] outputs, double[] gradients, double[,] weights)
    {
        double[] newGradients = new double[hiddenSize];
        for (int i = 0; i < hiddenSize; i++)
        {
            double sum = 0;
            for (int j = 0; j < gradients.Length; j++)
            {
                sum += gradients[j] * weights[i, j];
            }
            newGradients[i] = SigmoidDerivative(outputs[i]) * sum;
        }
        return newGradients;
    }
}
    }
```

Файл «LinearRegression.cs»

```
public class LinearRegression
{
    public MultipleLinearRegression? Train(double[][] inputData, double[] outputData)
    {
        if(inputData.Length != outputData.Length)
        {
            Console.WriteLine("Error! Number of inputs doesn't match number of outputs");
            return null;
        }
```

```
        var ols = new OrdinaryLeastSquares()
        {
            UseIntercept = true
        };

        MultipleLinearRegression regression = ols.Learn(inputData, outputData);

        return regression;
    }
        }
```

Файл «TestGames.cs»

```
  public class TestGames
  {
      public double[][] TEST_GAMES_INPUTS = new double[][]
      {
          //            T1W%,  T1RM,  T1RA,  T1HW%, T1HRM, T1HRA, T2W%,  T2RM,  T2RA,  T2AW%, T2ARM, T2ARA
          new double[] { 0.449, 4.735, 4.347, 0.520, 4.720, 4.120, 0.420, 4.340, 5.120, 0.318, 3.864, 4.727 }, // Chicago Cubs - Cincinatti
Reds (0 - 9)
          new double[] { 0.479, 3.729, 4.646, 0.524, 3.905, 4.571, 0.404, 4.135, 5.173, 0.333, 3.963, 4.963 }, // Detroit Tigers - Chicago
White Sox (3 - 12)
          new double[] { 0.712, 5.962, 3.769, 0.828, 5.690, 3.379, 0.608, 5.471, 4.373, 0.519, 5.259, 5.481 }, // Tampa Bay Rays - LA
Dodgers (9 - 3)
          new double[] { 0.660, 5.140, 4.240, 0.652, 4.348, 3.826, 0.633, 6.327, 4.041, 0.600, 5.880, 3.720 }, // Baltimore Orioles - Texas
Rangers (2 - 12)
          new double[] { 0.577, 4.596, 4.077, 0.586, 4.690, 4.034, 0.460, 4.000, 4.100, 0.478, 3.957, 4.304 }, // NY Yankees - San Diego
Padres (1 - 5)
          new double[] { 0.620, 5.180, 4.020, 0.538, 4.692, 4.885, 0.460, 4.300, 4.980, 0.346, 4.038, 5.923 }, // Atlanta Braves -
Philadelphia Phillies (4 - 6)
          new double[] { 0.294, 3.882, 5.314, 0.269, 4.308, 5.808, 0.420, 4.160, 4.640, 0.455, 4.818, 4.136 }, // Kansas City Royals -
Washington Nationals (10 - 12)
          new double[] { 0.540, 4.100, 4.240, 0.600, 4.440, 3.880, 0.500, 4.300, 4.600, 0.417, 4.625, 4.833 }, // Milwaukee Brewers - San
Francisco Giants (1 - 15)
          new double[] { 0.520, 4.600, 3.700, 0.600, 5.000, 3.800, 0.510, 4.784, 4.451, 0.448, 5.172, 5.069 }, // Minnesota Twins - Toronto
Blue Jays (1 - 3)
          new double[] { 0.431, 4.451, 5.431, 0.500, 5.077, 6.231, 0.510, 4.333, 4.569, 0.467, 4.267, 4.433 }, // Colorado Rockies - NY
Mets (2 - 5)
          new double[] { 0.549, 4.941, 4.627, 0.600, 5.160, 5.000, 0.490, 3.667, 4.647, 0.423, 4.038, 4.923 }, // LA Angels - Miami Marlins
(2 - 6)
          new double[] { 0.192, 3.519, 7.058, 0.200, 3.280, 6.720, 0.571, 4.347, 3.571, 0.583, 4.333, 3.458 }, // Oakland Athletics -
Houston Astros (2 - 5)
          new double[] { 0.429, 3.449, 4.245, 0.417, 2.750, 4.000, 0.442, 5.058, 4.788, 0.462, 4.269, 4.500 }, // Cleveland Guardians - St.
Louis Cardinals (4 - 3)
          new double[] { 0.580, 5.140, 4.840, 0.583, 5.125, 4.333, 0.520, 5.240, 5.100, 0.458, 4.417, 4.667 }, // Arizona Diamondbacks -
Boston Red Sox (2 - 7)
          new double[] { 0.520, 4.500, 3.820, 0.538, 4.385, 3.500, 0.510, 4.327, 4.061, 0.542, 4.750, 3.292 }  // Seattle Mariners -
Pittsburgh Pirates (6 - 11)
      };
      public double[] TEST_GAMES_OUTPUTS = new double[]
      {
          0, // Chicago Cubs - Cincinatti Reds (0 - 9)
          0, // Detroit Tigers - Chicago White Sox (3 - 12)
          1, // Tampa Bay Rays - LA Dodgers (9 - 3)
          0, // Baltimore Orioles - Texas Rangers (2 - 12)
          0, // NY Yankees - San Diego Padres (1 - 5)
          0, // Atlanta Braves - Philadelphia Phillies (4 - 6)
          0, // Kansas City Royals - Washington Nationals (10 - 12)
          0, // Milwaukee Brewers - San Francisco Giants (1 - 15)
          0, // Minnesota Twins - Toronto Blue Jays (1 - 3)
          0, // Colorado Rockies - NY Mets (2 - 5)
          0, // LA Angels - Miami Marlins (2 - 6)
          0, // Oakland Athletics - Houston Astros (2 - 5)
          1, // Cleveland Guardians - St. Louis Cardinals (4 - 3)
          0, // Arizona Diamondbacks - Boston Red Sox (2 - 7)
          0  // Seattle Mariners - Pittsburgh Pirates (6 - 11)
      };
          }
```

Файл «Program.cs»

```
using var context = new MLBDBContext();

var allGames = context.Games.Where(g => g.GameDate.Year < 2023).Select(g => new double[]
{
    g.HomeTeamWinningPercentage,
    g.HomeTeamAverageRunsMade,
    g.HomeTeamAverageRunsAllowed,
    g.HomeTeamHomeWinningPercentage,
    g.HomeTeamHomeAverageRunsMade,
    g.HomeTeamHomeAverageRunsAllowed,
    g.AwayTeamWinningPercentage,
    g.AwayTeamAverageRunsMade,
    g.AwayTeamAverageRunsAllowed,
    g.AwayTeamAwayWinningPercentage,
    g.AwayTeamAwayAverageRunsMade,
    g.AwayTeamAwayAverageRunsAllowed,
    g.Result
}).ToArray();

double[][] inputs = allGames.Select(row => row.Take(row.Length - 1).ToArray()).ToArray();
double[] outputs = allGames.Select(row => row.Last()).ToArray();

TestGames testGames = new TestGames();

LinearRegression linearRegression = new LinearRegression();
var regression = linearRegression.Train(inputs, outputs);

var correctLinearRegressionCount = 0;

for(int i = 0; i < testGames.TEST_GAMES_INPUTS.Count(); i++)
{
    var projectedResult = Math.Round(regression.Transform(testGames.TEST_GAMES_INPUTS[i]));

    if(projectedResult == testGames.TEST_GAMES_OUTPUTS[i])
    {
        correctLinearRegressionCount++;
    }
}

NeuralNetwork nn = new NeuralNetwork();

for(int i = 0; i < inputs.Count(); i++)
{
    nn.Train(inputs[i], outputs[i]);
}

var correctNeuralNetworkCount = 0;

for (int i = 0; i < testGames.TEST_GAMES_INPUTS.Count(); i++)
{
    var projectedResult = nn.FeedForward(testGames.TEST_GAMES_INPUTS[i]);

    if (Math.Round(projectedResult[0]) == testGames.TEST_GAMES_OUTPUTS[i])
    {
        correctNeuralNetworkCount++;
    }
        }
```