

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту

«Технології паралельних обчислень. Курсова робота»

Тема: Алгоритм пошуку оптимальних значень методом градієнтного спуску та його параллельна реалізація з використанням мови Python

|  |  |
| --- | --- |
| **Керівник**:  проф. Стеценко Інна Вячеславівна  «Допущено до захисту»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2024 р.  Захищено з оцінкою  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Члени комісії:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Виконавець**:  Тихонов Федір Сергійович  студент групи ІП-11  залікова книжка № 11-30  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «23» травня 2024 р.  Антон ДИФУЧИН |

**Київ – 2024**

ЗМІСТ

[ЗАВДАННЯ 4](#_Toc164967540)

[АНОТАЦІЯ 5](#_Toc164967541)

[ВСТУП 6](#_Toc164967542)

[1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ 7](#_Toc164967543)

[1.1 Послідовний градієнтний спуск для однієї вхідної змінної 7](#_Toc164967544)

[1.2 Градієнтний спуск для декількох вхідних змінних 10](#_Toc164967545)

[1.3 Пакетний градієнтний спуск для декількох вхідних змінних 11](#_Toc164967546)

[1.4 Поліноміальний пакетний градієнтний спуск 12](#_Toc164967547)

[2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ 16](#_Toc164967548)

[2.1 Проектування послідовного алгоритму 16](#_Toc164967549)

[2.2 Реалізація послідовного алгоритму 16](#_Toc164967550)

[2.3 Аналіз швидкодії послідовного алгоритму 19](#_Toc164967551)

[2.4 Висновок 23](#_Toc164967552)

[3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС 24](#_Toc164967553)

[4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ 25](#_Toc164967554)

[4.1 Проектування паралельного алгоритму 25](#_Toc164967555)

[4.2 Реалізація паралельного алгоритму 25](#_Toc164967556)

[4.3 Тестування алгоритму 27](#_Toc164967557)

[5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ 29](#_Toc164967558)

[5.1 Дослідження оптимальної кількості створених підпроцесів 29](#_Toc164967559)

[5.2 Дослідження оптимального розміру міні-пакета 31](#_Toc164967560)

[5.3 Дослідження порівняння часової ефективності послідовного і паралелізованого алгоритмів 32](#_Toc164967561)

[ВИСНОВКИ 35](#_Toc164967562)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 37](#_Toc164967563)

[ДОДАТКИ 38](#_Toc164967564)

[Додаток А. Код Matrix.py 38](#_Toc164967565)

[Додаток Б. Код файлу SeqMiniBatchGradientDescent.py 46](#_Toc164967566)

[Додаток В. Код файлу ParMiniBatchGradientDescent 49](#_Toc164967567)

[Додаток Г 52](#_Toc164967568)

# 

# ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму пошуку оптимальних значень методом градієнтного спуску, послідовних та паралельних, з відповідними посиланнями на джерела інформації (статті, книги, електронні ресурси). Зробити висновок про актуальність дослідження.
2. Виконати розробку послідовного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Опис алгоритму забезпечити у вигляді псевдокоду. Провести тестування алгоритму та зробити висновок про коректність розробленого алгоритму. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.
3. Виконати розробку паралельного алгоритму у відповідності до обраного завдання та обраного програмного забезпечення для реалізації. Опис алгоритму забезпечити у вигляді псевдокоду. Забезпечити ініціалізацію даних при будь-якому великому заданому параметрі кількості даних.
4. Виконати тестування алгоритму, що доводить коректність результатів обчислень. Тестування алгоритму обов’язково проводити на великій кількості даних. Коректність перевіряти порівнянням з результатами послідовного алгоритму.
5. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.
6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення не менше 1,2.
7. Дослідити вплив параметрів паралельного алгоритму на отримуване прискорення. Один з таких параметрів – це кількість підзадач, на які поділена задача при розпаралелюванні її виконання.
8. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

# АНОТАЦІЯ

Пояснювальна записка до курсової роботи: 50 сторінок, 31 рисунок, 4 таблиці.

Об’єкт дослідження: пошук оптимальних значень методом градієнтного спуску

Мета роботи: теоретично дослідити паралельні методи пошуку оптимальних значень методом градієнтного спуску; переглянути відомі їхні реалізації; спроектувати, реалізувати, протестувати послідовний та паралельний алгоритми; дослідити ефективність паралелізації програми;

Виконана програмна реалізація паралельного та послідовного алгоритму пошуку шляху у графі, проведено аналіз їх ефективності.

Ключові слова: ПОШУК ОПТИМАЛЬНИХ ЗНАЧЕНЬ, ГРАДІЄНТНИЙ СПУСК, ПАКЕТИ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ.

# ВСТУП

У сучасному світі машинного навчання та штучного інтелекту, оптимізація алгоритмів відіграє ключову роль у вирішенні складних задач, таких як навчання нейронних мереж. Градієнтний спуск (GD) [1] є одним з найбільш широко використовуваних алгоритмів оптимізації, який застосовується для мінімізації функції втрат та знаходження оптимальних параметрів моделі. Використання GD є актуальним у багатьох сферах, включаючи комп'ютерний зір, обробку природної мови та рекомендаційні системи.

Зі зростанням обсягів даних та складності моделей, виникає потреба в розробці ефективних паралельних алгоритмів для прискорення процесу навчання. У цьому контексті, дослідження можливостей паралельної реалізації алгоритму градієнтного спуску за допомогою мови програмування Python [2] набуває особливої актуальності.

Об'єктом даної курсової роботи є вивчення та аналіз підходів до паралельної реалізації алгоритму GD для моделей різної складності та розмірності. Враховуючи ітеративну природу GD, основним завданням є розробка ефективної паралельної реалізації, яка дозволить прискорити процес оптимізації. Також в рамках роботи буде розглянуто та проаналізовано ефективність різних методів паралельного виконання алгоритму, таких як паралельний стохастичний градієнтний спуск (SGD) [3] та міні-пакетний градієнтний спуск (Mini-Batch GD) [4].

Окрім цього, планується розробка та аналіз нового підходу до паралельної реалізації GD, який дозволить ефективно навчати моделі на великих обсягах даних та складних архітектурах нейронних мереж. Отримані результати та висновки матимуть важливе значення для розвитку та оптимізації алгоритмів машинного навчання в умовах сучасних обчислювальних ресурсів та потреб у швидкому навчанні моделей.

# 1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Поки розпараллелювання алгоритму градієнтного спуску не є найскладнішою задачею, розуміння алгоритму потребує розуміння математичної логіки та розуміння того, як взагалі математично можна знайти деякі оптимальні коефіцієнти для деякої функції.

## 1.1 Послідовний градієнтний спуск для однієї вхідної змінної

Для початку, сформулюємо функцію і загальну область використання градієнтного спуску. Однією з головних задач машинного навчання є лінійна регресія, тобто знаходження деякої лінійної залежності між вхідними (Х) і вихідними (у) даними.

Нехай ми маємо деяку вхідну характеристику х, та вихідну характеристику у, і деякі точки, між якими потрібно знайти залежність:

A graph paper with blue x and x

Description automatically generated

Рис. 1 – Набір точок на площині xOy.

Якщо ми хочемо знайти таку пряму, яка максимально наближувала б себе до цього набору точок, то за функцію, яку ми хочемо мінімізувати, є залишкова сума квадратів(англ. Residual Sum Of Squares or RSS), яку можна описати наступним чином:

*,* (1.1)

де, – це кількість точок у наборі даних, – це передбачуване значення вихідних даних, – це дійсне значення вихідних даних, – це дійсне значення вхідних даних.

Залишок – це відстань від дійсної точки у наборі даних до точки на передбачуваній прямій з відповідним аргументом. Залишки візуалізовані на рис.2.

A graph of a line graph

Description automatically generated

Рис. 2 – Візуалізація залишків

Тоді функція передбачуваних значень матиме наступний вигляд:

(1.2)

Де w – це кутовий коефіцієнт х, а b – це зміщення. Ці значення – це ті значення, які нам потрібно оптимізувати.

Дещо важливо зауважити, що в цьому підрозділі наводиться приклад використання градієнтного спуску тільки на лінійній функції, але сам градієнтний спуск може знаходити параметри для будь-якої функції.

Функцію RSS потрібно мінімізувати, оскільки скорочуючи залишки ми скорочуємо відстань між передбаченим значенням та дійсним значенням. Якщо представити функцію RSS у залежності від x та у, то ми отримаємо деяку опуклу функцію, яка виглядає таким чином:

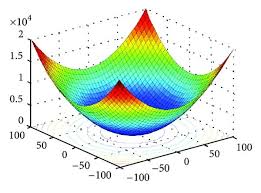


Рис. 3 – Опукла функція для пошуку оптимальних значень

Дана функція має тільки один локальний мінімум, що підштовхує до ідеї, що якщо представити похідну цієї функції, то локальний мінімум і є мінімумом цієї функції. Мінімум функції означає максимальне наближення шуканих коефіцієнтів до оптимального значення. Проте даний випадок зустрічається нечасто, і на практиці зустрічаються випадки, коли є більше одного локального мінімуму.

Похідну RSS можна порахувати за допомогою відомих правил знаходження похідних, і результат виглядає наступним чином:

(1.3)

Відтепер задача полягає у тому, щоби знайти мінімальне значення похідної, що можна зробити за допомогою градієнтів. Градієнт – це значення часткової похідної функції. Його користь полягає у тому, що він визначає напрямок найвищого зростання функції у заданій точці. З цього можна дізнатися напрямок функції, при якому вона зменшується.

Тоді значення градієнта відносно кутового коєфіцієнта вираховується за формулою:

(1.4)

Значення градієнта відносно зміщення вираховується за формулою:

(1.5)

Після вираховування градієнтів змінюємо значення кутового коєфіцієнта і зміщення:

(1.6)

(1.7)

Де learning\_rate – це деяке значення, яке потрібно налаштувати вручну для максимальної ефективності, і варієються від значень 0.5 до 0.0001.

Процес вираховування градієнтів та зміни коефіцієнтів повторюється деяку кількість разів, або епох, або поки не буде досягнуто деякого цільового значення[1].

## 1.2 Градієнтний спуск для декількох вхідних змінних

На щастя, для реалізації градієнтного спуску з деякою m-ною кількістю вхідних змінних, потрібно внести відносно небагато змін у початковий алгоритм.

Головною зміною від градієнтного спуску з однією змінною є те, що рівняння прямої (1.1) тепер виглядає як

(1.8)

Тобто тепер замість одного кутового коефіцієнта у нас є вектор кутових коефіцієнтів:

(1.9)

Тоді функція RSS (1.1) повертає на виході не скаляр, а вектор значень довжиною m. Від цього трохи змінюється процес пошуку похідної, оскільки тепер нам потрібно шукати m похідних, відносно кожної невідомої, тобто m разів.

З цього випливає, що тепер замість скалярного добутку, у формулі пошуку градієнтів для кутових коефіцієнтів (1.4) використовуватимемо алгоритм множення матриць:

(1.10),

Де x – це вхідна одиниця даних, y – це вихідна одиниця даних, n – об’єм набору даних, m – кількість вхідних змінних

(1.11)

## 1.3 Пакетний градієнтний спуск для декількох вхідних змінних

Хоча й попередня імплементація алгоритму є дуже корисною, з нею є дуже багато проблем. По-перше, цей алгоритм оброблює увесь набір даних одним «махом». Тобто, якщо цей алгоритм отримає на вхід декілька ГБ даних, то можуть виникнути певні труднощі. Цей алгоритм також має дуже високу складність[2].

У реалізації алгоритму пошуку оптимальних значень градієнтного спуску, я обрав варіацію алгоритму, що використовує імплементацію навчання за допомогою пакетів. На початку набір даних розбивається на деяку кількість пакетів, розміри яких варіюються у межах від кількох десятків до кількох сотень. Градієнти цих пакетів потім вираховуються паралельно та потім агрегуються. Функція пошуку градієнтів залишається.

Псевдокод:

func optimize (X, y, w, b, epochs, learning\_rate){

for(int epoch = 0; epoch < epochs; epoch++){

w\_grads = [];

b\_grads = [];

parallelize {

for X\_batch, y\_batch in zip(X, y){

w\_grad, b\_grad = compute\_gradients(X\_batch, y\_batch, y, w, b); // Параллельний розрахунок коефіцієнтів пакетів

w\_grads.add(w\_grad); // Додання коефіцієнтів

b\_grads.add(b\_grad); // у списки

}

}

w\_grad = sum(w\_grads) / len(w\_grads) // Агрегація коефіцієнтів

b\_grad = sum(b\_grads) / len(b\_grads)

w -= w\_grad \* learning\_rate; // Зміна кутових й //коефіцієнтів (1.6)

b -= b\_grad \* learning\_rate; // Зміна зміщення (1.7)

}

return w, b;

## 1.4 Поліноміальний пакетний градієнтний спуск

У підрозділах раніше було показано варіації градієнтного спуску лише для лінійної функції. Але градієнтний спуск може знаходити оптимальні значення для будь-яких функцій. В цьому розділі наведено загальний алгоритм для пошуку оптимальних значень коефіцієнтів поліноміальних функцій методом градієнтного спуску[3].

Загальний вигляд поліному:

По-перше, коефіцієнти для поліному, де степінь більше нуля, і де є декілька вхідних функцій, тепер виглядає таким чином:

Де m – максимальний степінь поліному, n – кількість вхідних змінних

Коефіцієнт b не змінюється.

Процес пошуку оптимальних коефіцієнтів поліноміальної функції не змінюється, проте змінюється процес обчислення градієнтів w, а саме, змінюється процес вираховування передбачуваних значень , та процес власне обчислення градієнтів.

Псевдокод для функції обчислення градієнтів:

func compute\_gradients(double[][] X, double[] y, double[][]w, double b, max\_degree){  
 N = y.shape[0];  
 y\_pred = List(N);

for(int i = 0; i < N, i++ ){

y\_pred.add(0.0) // Ініціалізація масиву передбачуваних значень, //наповнення 0

}  
 for(int degree = 0; degree < max\_degree; degree++){  
 added = X.pow(degree + 1).dot(w.get\_column(degree));//Вираховування   
 y\_pred = y\_pred + added; // передбачуаних значень

}  
 y\_pred = y\_pred + b;  
 error = y\_pred – y; // Вираховування похибки

w\_grad = List(N);

for(int i = 0; i < N, i++ ){

w\_grad[i] = List(max\_degree – 1);

for(int j = 0; j < max\_degree – 1; j++){ // Ініціалізація параметрів w

w\_grad[i][j] = 0;

}

}  
 for(int degree = 0; degree < max\_degree – 1; degree++){  
 w\_grad.set\_column(degree, X.pow(degree + 1).T.dot(error) \* (2 / N));

} // Вираховування градієнтів коефіцієнтів w  
 b\_grad = error.sum\_elem() \* (2 / N); // Вираховування градієнту b;  
 return w\_grad, b\_grad;

}

Як можна побачити, наголовніші зміни полягають у обчисленні передбачуваних значеннях, оскільки кожен раз коли ми проходимо коефіцієнти наступного ступеня, ми вводимо вхідні дані у деяку степінь і множимо на відповідний коефіцієнт, та у обчисленні градієнтів w, оскільки ми вираховуємо градієнти для кожного ступеня окремо.

До алгоритму пошуку оптимальних значень також було додано ранньої зупинки, якщо знайдені градієнти мають деяке дуже мале значення:

func optimize (X, y, w, b, epochs, learning\_rate, max\_degree, early\_stopping){

for(int epoch = 0; epoch < epochs; epoch++){

w\_grads = [];

b\_grads = [];

for X\_batch, y\_batch in zip(X, y){

w\_grad, b\_grad = compute\_gradients(X\_batch, y\_batch, y, w, b, max\_degree); // розрахунок коефіцієнтів пакетів

w\_grads.add(w\_grad); // Додання коефіцієнтів

b\_grads.add(b\_grad); // у списки

}

w\_grad = sum(w\_grads) / len(w\_grads) // Агрегація коефіцієнтів

b\_grad = sum(b\_grads) / len(b\_grads)

w -= w\_grad \* learning\_rate;

b -= b\_grad \* learning\_rate;

if (avg(w\_grad) < early\_stopping && b\_grad < early\_stopping){

break;

}

return w, b;

}

Для паралельної реалізації алгоритму:

func optimize (X, y, w, b, epochs, learning\_rate, max\_degree, early\_stopping){

for(int epoch = 0; epoch < epochs; epoch++){

w\_grads = List(len(X));

b\_grads = List(len(X));

parallelize{

for X\_batch, y\_batch in zip(X, y){

w\_grad, b\_grad = compute\_gradients(X\_batch, y\_batch, y, w, b, max\_degree); // розрахунок //градієнтів коефіцієнтів пакетів

w\_grads.add(w\_grad); // Додання коефіцієнтів

b\_grads.add(b\_grad); // у списки

}

}

w\_grad = sum(w\_grads) / len(w\_grads) // Агрегація коефіцієнтів

b\_grad = sum(b\_grads) / len(b\_grads)

w -= w\_grad \* learning\_rate;

b -= b\_grad \* learning\_rate;

if (avg(w\_grad) < early\_stopping && b\_grad < early\_stopping){

break;

}

return w, b;

}

# 2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

У цій частині роботи проводиться ґрунтовний аналіз процесу розробки послідовного алгоритму градієнтного спуску. Визначаються ключові кроки та етапи, необхідні для створення функціонального та ефективного алгоритмічного рішення.

Основний фокус розділу зосереджений на дослідженні та порівнянні часових та просторових характеристик розробленого послідовного алгоритму градієнтного спуску. Це дозволяє оцінити його ефективність та ідентифікувати потенційні напрямки для подальшої оптимізації.

## 2.1 Проектування послідовного алгоритму

Відповідно до минулого розділу, було розроблено послідовну імплементацію алгоритму пошуку оптимальних значень методом градієнтного спуску. Я вирішив використати архітектуру класу для реалізації градієнтного спуску, оскільки це значно полегшить інтерфейс для користувача.

## 2.2 Реалізація послідовного алгоритму

Як було наведено у минулому розділі, більшість математичних операцій, які використовуються для алгоритму в тому чи іншому вигляді включають в себе математичні операції над матрицями, будь це додавання, віднімання, множення матриць на число чи на іншу матрицю.

Тому було розроблено власний клас Matrix:

A screen shot of a computer program

Description automatically generated

Рис. 4 – Конструктор матриці

У перелік функціональних можливостей класу включено транспонування і розділення на пакети(для параллелізації):

A screenshot of a computer

Description automatically generated

Рис. 5 – Транспонація матриці і розділення на пакети

Нормалізацію:

A computer screen with white text

Description automatically generated

Рис. 6 – Нормалізація матриці

Додавання, віднімання на числа чи іншу матрицю:

A screen shot of a computer

Description automatically generated

Рис. 7 – Додавання і віднімання матриць з числами або з іншими матрицями

Множення і ділення матриць(поелементно):

A screenshot of a computer program

Description automatically generated

Рис. 8 – Поелементне множення та ділення матриць на числа чи інші матриці

Множення матриць і знаходження суми елементів матриці:

A screenshot of a computer

Description automatically generated

Рис. 9 – Сума елементів матриці і множення матриць

Цього функціоналу достатньо для покриття всіх потреб послідовного та паралельного градієнтного спуску. Отже, програмна реалізація послідовного градієнтного спуску полягає у класі SeqMiniBatchGradientDescent, конструктор якого наведено нижче:

A computer screen shot of a program code

Description automatically generated

Рис. 10 – Конструктор конструктора класу SeqMiniBatchGradientDescent

Функція для пошуку градієнтів:

A computer screen with text

Description automatically generated

Рис. 11 – Функція для обчислення градієнтів

Даний метод має абсолютно ту саму поведінку як функція для пошуку градієнтів у псевдокоді минулого розділу.

Метод для оптимізації коефіцієнтів:

A computer screen shot of a program code

Description automatically generated

Рис. 12 - Метод для оптимізації коефіцієнтів

Цей метод дещо відрізняється від функції у псевдокоді, але це пов’язано з форматом виведення процесу оптимізації градієнтів. Окрім того, щоб порахувати коефіцієнти, ця функція також засікає час, за який градієнти оптимізуються і показує шкалу прогресу під час оптимізації.

Було обрано клас для реалізації алгоритму через те, що функціональна реалізація включала б у себе значно більшу кількість вхідних параметрів, повертаємих значень і тд. Дана реалізація значно спрощує сприйняття коду для користувача.

## 2.3 Аналіз швидкодії послідовного алгоритму

Перед тим, як перевіряти швидкодію алгоритму, важливо зазначити, на якому пристрої відбувалися експерименти:

A screenshot of a computer

Description automatically generated

Рис. 13 - Пристрій для експериментів

По-перше, проведемо експеримент на точність алгоритму. У цій задачі, градієнтний спуск має знаходити нелінійний зв’язок між величинами, тому і дані я підібрав такі, які залежать нелінійно.

A computer screen with numbers and symbols

Description automatically generated

Рис. 14 – Тренувальний набір даних

Даний набір даних складається з 5,000 елементів, де вхідні дані складаються з 3 різних колонок. Також проводиться нормалізація, оскільки це дуже сильно полегшує алгоритму пошук нелінійних залежностей, тому нормалізація є прийнятною практикою при навчанні моделей нейронних мереж.

Проведемо оптимізацію алгоритму:

A black screen with white text

Description automatically generated

Рис. 15 – Оптимізація послідовним алгоритмом

Як можна побачити, я обрав дуже велике число епох для алгоритму. Це зроблено через те, що пошук нелінійних залежностей є дуже складною задачею для алгоритму, тому і кількість ітерацій є дуже великою. Також я обрав параметр stopping, який відповідає за ранню зупинку, і він дорівнює значенню , що є мінімальним допустимим значенням для градієнту. Можна помітити, що алгоритм зупинився на 1365 ітерації.

Набір даних обсягом у 5,000 елементів було оброблено успішно, перевіримо результати метрикою MSE, або середньоквадратичною похибкою:

A screenshot of a computer program

Description automatically generated

Рис. 16 – Середньоквадратична похибка для послідовного алгоритму

Середньоквадратична похибка для нашого алгоритму складає , що є дуже точним результатом. Можливо це здається занадто точним, але насправді це є доволі середній резлуьтат, враховуючи те, що наші дані нормалізовані.

Наведемо приклад результату, що виводить алгоритм. Вхідні дані були нормалізовані, тому проведемо денормалізацію даних, щоб більш-менш точно показати результати роботи алгоритму:

A screenshot of a computer

Description automatically generated

Рис. 17 – Результати оптимізації алгоритму

Проведемо серію досліджень для тренувальних наборів даних змінної величини на лінійних функціях з наступним патерном для вхідних даних:

A screenshot of a computer

Description automatically generated

Рис. 18 – Вхідні дані для алгоритму

|  |  |  |
| --- | --- | --- |
| *Кількість вибірок (5 вхідних колонок)* | *Час виконання, секунди* | *Кількість ітерацій до ранньої зупинки* |
| 5000 | 4.068210 | 93 |
| 10000 | 8.701283 | 94 |
| 15000 | 13.021464 | 94 |
| 20000 | 18.997838 | 94 |
| 25000 | 24.450192 | 95 |
| 30000 | 23.511114 | 95 |
| 35000 | 28.802749 | 95 |
| 40000 | 36.990829 | 95 |
| 45000 | 36.763637 | 88 |
| 50000 | 38.365674 | 89 |

Таблиця 1 – Дослідження швидкодії послідовного алгоритму

Дані результати було б також краще продемонструвати на графіку:

A graph with a line going up

Description automatically generated

Рис. 19 – Графік залежності часу виконання послідовного алгоритму від кількості вибірок

Бачимо, що різниця у часі з 5,000 вибірок до 50,000 зросла майже у 10 разів. Сучасні великомасштабні нейронні мережі мають інколи значно більшу кількість вхідних даних і колонок для вхідних даних, тому подальша оптимізація є дуже бажаною.

## 2.4 Висновок

У цьому розділі було реалізовано послідовний алгоритм пошуку оптимальних значень методом градієнтного спуску на мові Python. Для цього також було виконано клас Matrix для спрощених дій над матрицями результати оптимізації знаходяться у об’єкті класу, тому можна без зайвих проблем провести діагностику оптимізації.

# 3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

Для реалізації паралельного міні-пакетного градієнтного спуску (Mini-Batch Gradient Descent) було використано кілька бібліотек мови програмування Python, які дозволяють ефективно розпаралелювати обчислення та прискорити процес навчання моделі.

Одна з ключових бібліотек, які використовуються в даному алгоритмі, - це `concurrent.futures`. Ця бібліотека надає високорівневий інтерфейс для асинхронного виконання коду та управління паралельними задачами. Зокрема, клас `ProcessPoolExecutor` з цієї бібліотеки дозволяє створювати пул процесів, які можуть одночасно виконувати обчислення градієнтів для різних міні-пакетів даних. Це дозволяє ефективно розподілити навантаження між доступними процесорами та прискорити процес навчання.

Крім того, для роботи з матрицями та векторами використовується власна реалізація класу `Matrix`, який забезпечує основні операції лінійної алгебри, такі як множення матриць та транспонування. Цей клас дозволяє зручно працювати з даними та спрощує реалізацію обчислень градієнтів.

Для вимірювання часу виконання алгоритму та відображення прогресу навчання використовуються вбудовані бібліотеки `time` та `math`. Бібліотека `time` дозволяє визначити час початку та закінчення роботи алгоритму, що дає змогу оцінити швидкодію та ефективність розпаралелювання. Бібліотека `math` використовується для математичних обчислень, таких як округлення та ділення.

Використання цих бібліотек у поєднанні з продуманою архітектурою класу `ParMiniBatchGradientDescent` дозволяє досягти ефективної паралельної реалізації міні-пакетного градієнтного спуску, що значно прискорює процес навчання моделей машинного навчання.

# 4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

У рамках даного розділу проведемо огляд процесу розробки паралельного алгоритму пошуку оптимальних значень методом градієнтного спуску. Виконаємо дії аналогічно 2 розділу. Також розглянемо розпаралелювання, та як його краще використати для нашого алгоритму.

## 4.1 Проектування паралельного алгоритму

Для паралелізації послідовного алгоритму, я вирішив паралелізувати обчислення градієнтів для пакетів. Як було наведено у 1 розділі, суть алгоритму полягає у розпаралелюванні обробки міні-пакетів.

## 4.2 Реалізація паралельного алгоритму

Для паралелізації обчислення градієнтів міні-пакетів використаємо пул процесів від бібліотеки concurrent. Ми не використовуємо пул потоків через те, що Python Global Interpreter Lock дозволяє виконувати байт-код мови тільки одному потоку за раз, навіть якщо було ініціалізовано декілька потоків у середовищі.

З іншої сторони, пул процесів уникає це обмеження, оскільки він ініціює окремі процеси, для яких обмежень щодо виконання коду, таких як Python GIL, не існує. Це може надати дуже серйозне прискорення, порівняно з послідовним алгоритмом. Реалізація паралельного алгоритму, так само як і реалізація послідовного алгоритму, використовує клас для виконання усіх операцій під назвою ParMiniBatchGradientDescent:

A computer screen with white text

Description automatically generated

Рис. 20 – Конструктор класу

Перша зміна полягає у додаткових параметрі num\_processеs. Цей параметр вказує кількість підпроцесів, які створюються.

Алгоритм пошуку градієнтів програмно не змінюється відносно послідовного алгоритм.

A computer screen shot of a program code

Description automatically generated

Рис. 21 – Алгоритм обчислення градієнтів для паралелізованого алгоритму

Процес оптимізації, з іншої сторони, змінюється доволі сильно на відміну від функції обчислення градієнтів. Внутрішній цикл для обробки міні-пакетів паралелізується пулом процесів. Для пришвидшення було використано генератор списків (list comprehension) для внутрішнього циклу і його обробки пулом процесів. Також було додано шкалу прогресу і виведення часу після закінчення роботи алгоритму:

A computer screen shot of a program code

Description automatically generated

Рис. 22 – Паралелізований алгоритм градієнтного спуску

## 4.3 Тестування алгоритму

Проведемо експеримент аналогічний тому, що було проведено з послідовним алгоритмом. Дослідження щодо часової ефективності паралельного алгоритму буде виконано у наступному розділі.

Створюємо тренувальний набір даних з 5,000 вибірок, аналогічний тому, що було виконано на послідовному алгоритмі.

A screenshot of a computer program

Description automatically generated

Рис. 23 – Створюваний тренувальний набір даних

Створюємо об’єкт класу ParMiniBatchGradientDescent, та починаємо процес оптимізації:

A screen shot of a computer program

Description automatically generated

Рис. 24 – Оптимізація паралельного алгоритму

Як можна побачити, набір у 5,000 вибірок паралельний алгоритм за 31.1 секунду, але про це детальніше у наступному розділі. Порівняємо метрику MSE:

A computer screen with text and numbers

Description automatically generated

Рис. 25 – оцінка MSE для паралельного алгоритму

Очевидно, що дана похибка є дуже близькою до 0, тому можна вважати реалізацію алгоритму успішною.

Виведемо практичні результати обробки даних:

A screenshot of a computer

Description automatically generated

Рис. 26 – результат від паралелізованого градієнтного спуску

Результат ідентичний тому, що був на послідовному алгоритмі. Це очікувано, оскільки було взято одні й ті самі гіперпараметри.

# 5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Ми упевнилися у здатності алгоритму знаходити оптимальні значення, тому можемо перейти до наступного етапу – це підбір оптимальних параметрів створюємих підпроцесів та розміру пакетів.

## 5.1 Дослідження оптимальної кількості створених підпроцесів

Усі експерименти виконані на тому самому пристрої, що і для послідовного алгоритму, нижче наведено кількість ядер на пристрої:

A screenshot of a computer

Description automatically generated

Рис. 27 – Кількість ядер на пристрої

Отже, на пристрої є загалом 8 ядер, 4 ядра на комплексні задачі та 4 енергоефективних ядра.

Подивимося, як різна кількість генеруємих підпроцесів впливає на прискорення алгоритму на тренувальному наборі даних у 50,000 вибірок. Функцію обрано лінійну, оскільки для поліноміальної функції треба у рази більше ітерацій.

У наступному експерименті дізнаємося, яка кількість підпроцесів працює найкраще. Обсяг створюємих підпроцесів варіюється від 1 до 16.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Кількість створених підпроцесів* | *Час виконання для 1 заміру (у секундах)* | *Час виконання для 2 заміру (у секундах)* | *Час виконання для 3 заміру (у секундах) заміру)* | ***Середнє значення серед замірів(у секундах)*** |
| 1 | 44.735188 | 43.783946 | 39.271311 | **42.596815** |
| 2 | 23.169742 | 23.826998 | 20.814192 | **22.603644** |
| 3 | 16.401435 | 19.939893 | 14.839453 | **17.060260** |
| 4 | 19.303956 | 14.490517 | 12.040704 | **15.278392** |
| 5 | 14.057023 | 13.687116 | 11.374417 | **13.039519** |
| 6 | 13.052068 | 13.479485 | 10.915975 | **12.482509** |
| 7 | 11.481244 | 12.478308 | 10.565728 | **11.508427** |
| 8 | 14.011223 | 11.352478 | 10.416532 | **11.926744** |
| 9 | 14.932922 | 11.411429 | 10.394200 | **12.246184** |
| 10 | 15.126027 | 11.338340 | 10.463190 | **12.309186** |
| 11 | 12.060377 | 11.512666 | 10.305280 | **11.292774** |
| 12 | 12.265940 | 11.379978 | 10.434307 | **11.360075** |
| 13 | 12.163645 | 10.846929 | 10.522866 | **11.177813** |
| 14 | 11.833435 | 10.441365 | 10.450305 | **10.908369** |
| 15 | 12.096543 | 10.384644 | 10.670744 | **11.050644** |
| 16 | 11.926364 | 10.498398 | 10.402440 | **10.942401** |

Таблиця 2 – Час виконання в залежності від кількості підпроцесів

Щоб більш наглядно продемонструвати ці дані, нижче наведено графік:

A graph with a line

Description automatically generated

Рис. 28 Графік часу виконання від кількості підпроцесів

Як можна побачити, найкраще алгоритм працює на 14 та 16 створюємих підпроцесах.

## 5.2 Дослідження оптимального розміру міні-пакета

Наступний експеримент – порівняємо швидкодію алгоритму в залежності від розміру міні-пакетів на тій самій вибірці у 50,000 елементів. Розглянуті значення включають в себе наступні значення: {8, 16, 32, 64, 128, 256, 512, 1024, 2048}. Також виведемо значення MSE, якого досягатиме алгоритм при цих значеннях.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Розмір | Час на 1 замірі | Час на 2 замірі | Час на 3 замірі | **Час у середньому** | **MSE у середньому** |
| 8 | 76.246442 | 77.010089 | 77.235718 | **76.830750** | **1.126982e-17** |
| 16 | 60.477854 | 42.866919 | 36.815521 | **46.720098** | **1.126982e-17** |
| 32 | 25.069484 | 24.151656 | 27.898976 | **25.706705** | **1.168949e-17** |
| 64 | 18.053743 | 16.444316 | 17.340935 | **17.279665** | **1.257746e-17** |
| 128 | 12.400642 | 12.362712 | 12.826942 | **12.530099** | **1.257553e-17** |
| 256 | 10.273869 | 10.147541 | 11.530241 | **10.650550** | **1.108988e-17** |
| 512 | 10.707288 | 9.660846 | 11.148822 | **10.505652** | **1.106446e-17** |
| 1024 | 9.606693 | 9.790903 | 11.039120 | **10.145572** | **1.101386e-17** |
| 2048 | 11.661306 | 11.245413 | 12.434126 | **11.780282** | **1.141518e-17** |

Таблиця 3 – Залежність часу виконання і MSE від розміру пакета

Також візуалізуємо ці значення на графіку:

A graph with a line

Description automatically generated

Рис. 29 – Середня залежність часу виконання від розміру міні-пакета

A graph with lines on it

Description automatically generated

Рис. 30 – Середнє залежність MSE від часу пакета

Тож можемо обрати у якості параметру для розміру пакета 1024, оскільки це параметр, при якому час виконання і похибка найменші.

## 5.3 Дослідження порівняння часової ефективності послідовного і паралелізованого алгоритмів

Отже, було підібрано усі необхідні значення для максимального прискорення алгоритму, тому зараз можемо порівняти часову ефективність обох алгоритмів:

|  |  |  |  |
| --- | --- | --- | --- |
| Кількість вибірок | Cередній час виконання паралельного алгоритму | Cередній час виконання послідовного алгоритму | **Прискорення** |
| 5000 | 1.888315 | 4.068210 | **2.154413** |
| 10000 | 2.973364 | 8.701283 | **2.926410** |
| 15000 | 4.324602 | 13.021464 | **3.011020** |
| 20000 | 5.116435 | 18.997838 | **3.713101** |
| 25000 | 5.726192 | 24.450192 | **4.269887** |
| 30000 | 6.541405 | 23.511114 | **3.594199** |
| 35000 | 8.430988 | 28.802749 | **3.416296** |
| 40000 | 12.048959 | 36.990829 | **3.070044** |
| 45000 | 9.152933 | 36.763637 | **4.016596** |
| 50000 | 11.608839 | 38.365674 | **3.304867** |

Таблиця 4 – Порівняння швидкодії паралельного і послідовного алгоритмів

Щоб показати, наскільки краще працює паралельний алгоритм порівняно з послідовним, наведено графік:

A graph with lines and numbers

Description automatically generated

Рис. 31 – Порівняння часової ефективності послідовного і паралельного алгоритмів

Як можна побачити, прискорення у паралельного алгоритму порівняно з послідовним відбувається більш ніж у 4 рази на більших наборах даних, і у 2 разів на менших наборах даних.

# ВИСНОВКИ

Виконання курсової роботи дозволило нам глибоко зануритись у вивчення паралельної реалізації алгоритму градієнтного спуску, що включало теоретичний аналіз алгоритму, детальний опис ключових класів, допоміжних структур даних та їх взаємодії. Процес ретельного тестування виявив помилки, забезпечив виправлення та підтвердження ефективності розробленого рішення.

Аналіз даних, отриманих в ході дослідження, продемонстрував значні переваги застосування паралельного виконання градієнтного спуску у порівнянні з послідовним. При збільшенні кількості процесів до восьми, прискорення склало більше ніж 3 рази на великих наборах даних, що свідчить про ефективність паралелізації задачі. Це досягнення є особливо значущим, враховуючи обчислювальну складність алгоритму та великі обсяги даних, що використовуються у сучасних задачах машинного навчання.

Проте, було також зазначено, що подальше збільшення кількості процесів призвело лише до незначного покращення продуктивності, що підкреслює існування межі ефективності паралельної обробки для даної задачі. Це обмеження було пов'язано з накладними витратами на комунікацію між процесами та синхронізацію, які з часом починають переважувати над вигодами від додавання нових обчислювальних ресурсів.

Крім того, дослідження виявило важливість вибору оптимального розміру міні-пакетів для досягнення найкращої продуктивності. Занадто малі міні-пакети призводять до надмірних накладних витрат на комунікацію, тоді як занадто великі міні-пакети обмежують можливості паралелізації. Знаходження балансу між цими факторами є ключовим для ефективної реалізації паралельного градієнтного спуску.

У підсумку, дослідження підтвердило високу ефективність паралельної реалізації градієнтного спуску, особливо при оптимальній кількості процесів та розмірі міні-пакетів, що дозволяє значно скоротити час навчання моделей на великих наборах даних. Результати цієї роботи відкривають нові можливості для масштабування алгоритмів машинного навчання та прискорення процесу розробки інтелектуальних систем. Подальші дослідження можуть бути спрямовані на вдосконалення паралельних алгоритмів оптимізації та адаптацію їх до різних архітектур паралельних обчислювальних систем.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] S. Ruder, "An overview of gradient descent optimization algorithms," arXiv preprint arXiv:1609.04747, 2016.

[2] M. Zinkevich et al., "Parallelized stochastic gradient descent," Advances in neural information processing systems, 2010.

[3] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

# ДОДАТКИ

## Додаток А. Код Matrix.py

import copy  
import math  
import types  
  
  
class Matrix:  
 def \_\_init\_\_(self, matrix):  
 self.\_\_matrix = matrix  
 self.\_\_rows = len(matrix)  
 self.\_\_cols = len(matrix[0])  
 self.T = self.\_\_transpose()  
  
 def \_\_str\_\_(self):  
 return str(self.\_\_matrix)  
  
 def shape(self):  
 return self.\_\_rows, self.\_\_cols  
  
 def \_\_getitem\_\_(self, key):  
 if isinstance(key, tuple):  
 if len(key) != 2:  
 raise IndexError("Index must be a tuple of 2 integers")  
 if key[0] >= self.\_\_rows or key[1] >= self.\_\_cols:  
 raise IndexError("Index out of range")  
 return self.\_\_matrix[key[0]][key[1]]  
 elif isinstance(key, int):  
 if key >= self.\_\_rows:  
 raise IndexError("Index out of range")  
 return self.\_\_matrix[key]  
 elif isinstance(key, slice):  
 start = key.start or 0  
 stop = key.stop or self.\_\_rows  
 step = key.step or 1  
 return Matrix([row[start:stop:step] for row in self.\_\_matrix])  
 else:  
 raise TypeError("Invalid index type")  
  
 def \_\_transpose(self):  
 transposed = [[] for \_ in range(self.\_\_cols)]  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 transposed[j].append(self.\_\_matrix[i][j])  
 return transposed  
  
 def dot(self, other):  
 if self.\_\_cols != other.\_\_rows:  
 raise ValueError(f"Incorrect Shape: Matrix 1: {self.shape()}"  
 f", Matrix 2: {other.shape()}")  
 lst\_c = [[0 for \_ in range(other.\_\_cols)] for \_ in range(self.\_\_rows)]  
 for i in range(self.\_\_rows):  
 for j in range(other.\_\_cols):  
 for k in range(self.\_\_cols):  
 lst\_c[i][j] += self.\_\_matrix[i][k] \* other.\_\_matrix[k][j]  
 return Matrix(lst\_c)  
  
 def \_\_add\_\_(self, other):  
 if isinstance(other, int):  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] += other  
 return Matrix(new\_matrix)  
 if isinstance(other, float):  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] += other  
 return Matrix(new\_matrix)  
 if isinstance(other, Matrix):  
 if self.shape() != other.shape():  
 raise IndexError(f"Invalid Shape: Matrix 1: {self.shape()}, Matrix 2: {other.shape()}")  
 else:  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] += other[i][j]  
 return Matrix(new\_matrix)  
  
 def \_\_sub\_\_(self, other):  
 if isinstance(other, int):  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] -= other  
 return Matrix(new\_matrix)  
 if isinstance(other, float):  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] -= other  
 return Matrix(new\_matrix)  
 if isinstance(other, Matrix):  
 if self.shape() != other.shape():  
 raise IndexError(f"Invalid Shape: Matrix 1:"  
 f" {self.shape()}, Matrix 2: {other.shape()}")  
 else:  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] -= other[i][j]  
 return Matrix(new\_matrix)  
  
 def \_\_mul\_\_(self, other):  
 if isinstance(other, int):  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] \*= other  
 return Matrix(new\_matrix)  
 if isinstance(other, float):  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] \*= other  
 return Matrix(new\_matrix)  
 if isinstance(other, Matrix):  
 if self.shape() != other.shape():  
 raise IndexError(f"Invalid Shape: Matrix 1:"  
 f" {self.shape()}, Matrix 2: {other.shape()}")  
 else:  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] \*= other[i][j]  
 return Matrix(new\_matrix)  
  
 def \_\_truediv\_\_(self, other):  
 if isinstance(other, int):  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] /= other  
 return Matrix(new\_matrix)  
 if isinstance(other, float):  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] /= other  
 return Matrix(new\_matrix)  
 if isinstance(other, Matrix):  
 if self.shape() != other.shape():  
 raise IndexError(f"Invalid Shape: Matrix 1: "  
 f"{self.shape()}, Matrix 2: {other.shape()}")  
 else:  
 new\_matrix = copy.deepcopy(self.\_\_matrix)  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 new\_matrix[i][j] /= other[i][j]  
 return Matrix(new\_matrix)  
  
 def \_\_iter\_\_(self):  
 return iter(self.\_\_matrix)  
  
 def sum\_elem(self):  
 scalar = 0  
 for i in range(self.\_\_rows):  
 for j in range(self.\_\_cols):  
 scalar += self.\_\_matrix[i][j]  
 return scalar  
  
 def normalise(self, min\_val=None, max\_val=None):  
 if isinstance(min\_val, types.NoneType) is True and isinstance(max\_val, types.NoneType) is True:  
 min\_val = [min(feature) for feature in self.T]  
 max\_val = [max(feature) for feature in self.T]  
 normalised\_data = []  
 for data\_point in self.\_\_matrix:  
 normalised\_point = [(data\_point[i] - min\_val[i]) /  
 (max\_val[i] - min\_val[i]) for i in range  
 (len(data\_point))]  
 normalised\_data.append(normalised\_point)  
 normalised\_data = Matrix(normalised\_data)  
 return normalised\_data, min\_val, max\_val  
  
 def denormalise(self, min\_val, max\_val):  
 denormalised\_data = []  
 for data\_point in self.\_\_matrix:  
 denormalised\_point = [data\_point[i] \* (max\_val[i] - min\_val[i]) + min\_val[i]  
 for i in range(len(data\_point))]  
 denormalised\_data.append(denormalised\_point)  
 denormalised\_data = Matrix(denormalised\_data)  
 return denormalised\_data  
  
 def split\_into\_batches(self, batch\_size):  
 num\_batches = math.ceil(self.\_\_rows / batch\_size)  
 batched\_dataset = [[] for \_ in range(num\_batches)]  
 for num\_sample in range(self.\_\_rows):  
 batched\_dataset[num\_sample // batch\_size].append(self[num\_sample])  
 batched\_dataset = [Matrix(batch) for batch in batched\_dataset]  
 return batched\_dataset  
  
 def pow(self, degree):  
 raised\_matrix = Matrix([[elem \*\* degree for elem in row] for row in self.\_\_matrix])  
 return raised\_matrix  
  
 def get\_column(self, index):  
 if index < 0 or index >= self.\_\_cols:  
 raise IndexError("Column index out of range")  
 column = [[row[index]] for row in self.\_\_matrix]  
 return Matrix(column)  
  
 def set\_column(self, index, new\_column):  
 if index < 0 or index >= self.\_\_cols:  
 raise IndexError("Column index out of range")  
 if isinstance(new\_column, list):  
 if len(new\_column) != self.\_\_rows:  
 raise ValueError("New column must be the same length as other columns")  
 elif isinstance(new\_column, Matrix):  
 if new\_column.\_\_rows != self.\_\_rows:  
 raise ValueError("New column must be the same length as other columns")  
 for i in range(self.\_\_rows):  
 self.\_\_matrix[i][index] = new\_column[i][0]  
  
def matrix\_sum(matrix\_lst):  
 sum\_matrixes = Matrix([[0 for \_ in range(matrix\_lst[0].shape()[1])] for \_ in range(matrix\_lst[0].shape()[0])])  
 for matrix in matrix\_lst:  
 sum\_matrixes = sum\_matrixes + matrix  
 return sum\_matrixes

## Додаток Б. Код файлу SeqMiniBatchGradientDescent.py

import types  
from parallelized\_functions.Matrix import Matrix, matrix\_sum  
import math  
import time  
import numpy as np  
  
  
def compute\_gradients(X, y, w, b, max\_degree):  
 N = y.shape()[0]  
 y\_pred = Matrix([[0] for \_ in range(N)])  
 for degree in range(max\_degree):  
 added = X.pow(degree + 1).dot(w.get\_column(degree))  
 y\_pred = y\_pred + added  
 y\_pred = y\_pred + b  
 error = y\_pred - y  
 w\_grad = Matrix([[0] \* w.shape()[1] for \_ in range(X.shape()[1])])  
 for degree in range(max\_degree):  
 w\_grad.set\_column(degree, Matrix(X.pow(degree + 1).T).dot(error) \* (2 / N))  
 b\_grad = error.sum\_elem() \* (2 / N)  
 return w\_grad, b\_grad  
  
  
class SeqMiniBatchGradientDescent:  
 def \_\_init\_\_(self, X, y, learning\_rate, num\_iterations, batch\_size, stopping, max\_degree):  
 self.\_\_batched\_X = X.split\_into\_batches(batch\_size)  
 self.\_\_degree = max\_degree  
 self.\_\_X = X  
 self.\_\_y = y  
 self.\_\_stopping = stopping  
 self.\_\_batched\_y = y.split\_into\_batches(batch\_size)  
 self.\_\_batch\_size = batch\_size  
 self.\_\_num\_batches = math.ceil(X.shape()[0] / batch\_size)  
 self.\_\_epochs = num\_iterations  
 self.\_\_learning\_rate = learning\_rate  
 self.\_\_w = Matrix([[0] \* max\_degree for \_ in range(X.shape()[1])])  
 self.\_\_b = 0  
  
 def optimize(self):  
 num\_iterations = self.\_\_epochs  
 progress\_bar\_width = 50  
 print("Sequential Mini-Batch Gradient Descent Progress:")  
 print("[" + " " \* progress\_bar\_width + "]", end="\r")  
 start = time.time()  
 for epoch in range(self.\_\_epochs):  
 gradients = []  
 for batch\_X, batch\_y in zip(self.\_\_batched\_X, self.\_\_batched\_y):  
 gradients.append(compute\_gradients(batch\_X, batch\_y, self.\_\_w, self.\_\_b, self.\_\_degree))  
 w\_grad = matrix\_sum([grad[0] for grad in gradients]) / len(gradients)  
 b\_grad = sum([grad[1] for grad in gradients]) / len(gradients)  
 self.\_\_w = self.\_\_w - w\_grad \* self.\_\_learning\_rate  
 self.\_\_b = self.\_\_b - b\_grad \* self.\_\_learning\_rate  
 progress = (epoch + 1) / num\_iterations  
 filled\_width = int(progress \* progress\_bar\_width)  
 progress\_bar = "[" + "=" \* filled\_width + " " \* (progress\_bar\_width - filled\_width) + "]"  
 print(f"{progress\_bar} {progress:.0%}", end="\r")  
 if abs(w\_grad.sum\_elem() / (w\_grad.shape()[0] \* w\_grad.shape()[1])) < self.\_\_stopping and abs(b\_grad) < self.\_\_stopping:  
 print(f"\r\nEarly Stopping on iteration {epoch}")  
 break  
 print()  
 end = time.time()  
 print(f'Executed in {np.round(end - start, 3)} seconds')  
 return end - start  
  
 def predict(self, sample=None):  
 if isinstance(sample, types.NoneType) is True:  
 sample = self.\_\_X  
 y\_pred = Matrix([[0] for \_ in range(sample.shape()[0])])  
 for degree in range(self.\_\_w.shape()[1]):  
 y\_pred += sample.pow(degree + 1).dot(self.\_\_w.get\_column(degree))  
 y\_pred += self.\_\_b  
 return y\_pred  
  
 def get\_coefficients(self):  
 return self.\_\_w, self.\_\_b

## Додаток В. Код файлу ParMiniBatchGradientDescent

from parallelized\_functions.Matrix import Matrix, matrix\_sum  
import math  
from concurrent.futures import ProcessPoolExecutor  
import time  
import numpy as np  
import types  
  
  
def compute\_gradients(X, y, w, b, max\_degree):  
 N = y.shape()[0]  
 y\_pred = Matrix([[0] for \_ in range(N)])  
 for degree in range(max\_degree):  
 added = X.pow(degree + 1).dot(w.get\_column(degree))  
 y\_pred = y\_pred + added  
 y\_pred = y\_pred + b  
 error = y\_pred - y  
 w\_grad = Matrix([[0] \* w.shape()[1] for \_ in range(X.shape()[1])])  
 for degree in range(max\_degree):  
 w\_grad.set\_column(degree, Matrix(X.pow(degree + 1).T).dot(error) \* (2 / N))  
 b\_grad = error.sum\_elem() \* (2 / N)  
 return w\_grad, b\_grad  
  
  
class ParMiniBatchGradientDescent:  
 def \_\_init\_\_(self, X, y, learning\_rate, num\_iterations, batch\_size, num\_processes, stopping, max\_degree):  
 self.\_\_batched\_X = X.split\_into\_batches(batch\_size)  
 self.\_\_num\_processors = num\_processes  
 self.\_\_degree = max\_degree  
 self.\_\_X = X  
 self.\_\_stopping = stopping  
 self.\_\_y = y  
 self.\_\_batched\_y = y.split\_into\_batches(batch\_size)  
 self.\_\_batch\_size = batch\_size  
 self.\_\_num\_batches = math.ceil(X.shape()[0] / batch\_size)  
 self.\_\_epochs = num\_iterations  
 self.\_\_learning\_rate = learning\_rate  
 self.\_\_w = Matrix([[0] \* max\_degree for \_ in range(X.shape()[1])])  
 self.\_\_b = 0  
  
 def optimize(self):  
 num\_iterations = self.\_\_epochs  
 progress\_bar\_width = 50  
 print("Mini-Batch Gradient Descent Progress:")  
 print("[" + " " \* progress\_bar\_width + "]", end="\r")  
 start = time.time()  
 with ProcessPoolExecutor(max\_workers=self.\_\_num\_processors) as executor:  
 for epoch in range(self.\_\_epochs):  
 futures = [executor.submit(compute\_gradients, batch\_X, batch\_y, self.\_\_w, self.\_\_b, self.\_\_degree)  
 for batch\_X, batch\_y in zip(self.\_\_batched\_X, self.\_\_batched\_y)]  
  
 gradients = [future.result() for future in futures]  
  
 w\_grad = matrix\_sum([grad[0] for grad in gradients]) / len(gradients)  
 b\_grad = sum([grad[1] for grad in gradients]) / len(gradients)  
  
 self.\_\_w = self.\_\_w - w\_grad \* self.\_\_learning\_rate  
 self.\_\_b = self.\_\_b - b\_grad \* self.\_\_learning\_rate  
 progress = (epoch + 1) / num\_iterations  
 filled\_width = int(progress \* progress\_bar\_width)  
 progress\_bar = "[" + "=" \* filled\_width + " " \* (progress\_bar\_width - filled\_width) + "]"  
 print(f"{progress\_bar} {progress:.0%}", end="\r")  
 if abs(w\_grad.sum\_elem() / (w\_grad.shape()[0] \* w\_grad.shape()[1])) < self.\_\_stopping and abs(b\_grad) < self.\_\_stopping:  
 print(f"\r\nEarly Stopping on iteration {epoch}")  
 break  
 print()  
 end = time.time()  
 print(f'Executed in {np.round(end - start, 3)} seconds')  
 return end - start  
  
 def predict(self, sample=None):  
 if isinstance(sample, types.NoneType) is True:  
 sample = self.\_\_X  
 y\_pred = Matrix([[0] for \_ in range(sample.shape()[0])])  
 for degree in range(self.\_\_w.shape()[1]):  
 y\_pred += sample.pow(degree + 1).dot(self.\_\_w.get\_column(degree))  
 y\_pred += self.\_\_b  
 return y\_pred  
  
 def get\_coefficients(self):  
 return self.\_\_w, self.\_\_b

## Додаток Г

Посилання на ноутбуки з експериментами доступні за посиланням:

<https://github.com/FedirTikhonov/PP-CW/tree/main/notebooks>