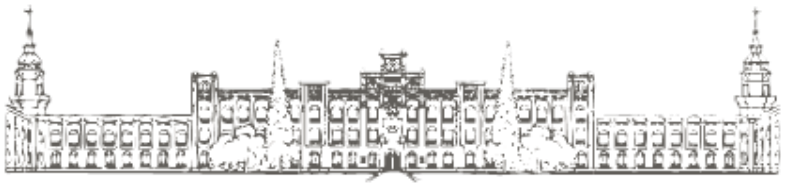

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту
«Технології паралельних обчислень. Курсова робота»
Тема: Алгоритм навчання нейромережі та його паралельна реалізація в Python

Керівник:
Професорка Стеценко Інна В’ячеславівна

«Допущено до захисту»
_____

«___» _____ 2024 р.

Захищено з оцінкою
_____

Члени комісії:

_____

_____

Виконавець:
Лошак Віктор Іванович
студент групи ІП-11
залікова книжка № ____
_____

«23» травня 2024 р.

Інна СТЕЦЕНКО

Київ – 2024

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму, 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

Структура та обсяг роботи. Пояснювальна записка курсової роботи складається з 5 розділів, містить 19 рисунків, 7 таблиць, 16 джерел.

Мета. Метою роботи є:

- Розробка алгоритму послідовного навчання нейронної мережі на мові програмування Python.
- Розробка алгоритму послідовного навчання нейронної мережі на мові програмування Python.
- Для паралельної реалізації досягти прискорення не менше 1.2 при достатньо великих розмірах вхідних даних.

У розділі опису алгоритму було надано загальний опис нейронної мережі, її призначення, введено основні поняття що необхідні для розуміння процесу тренування, представлено існуючі види паралелізації що застосовуються для алгоритмів тренування нейромереж.

У розділі розробки послідовного алгоритму було наведено опис послідовного алгоритму тренування, описано та обґрунтовано архітектурні рішення нейромережі.

У розділі вибору програмного забезпечення було описано програмні засоби що можуть бути використані для розробки паралельного алгоритму та обґрунтовано використання застосованих в розробці інструментів.

У розділі розробки паралельного алгоритму було описано процес проектування, імплементації, тестування та аналізу паралельного алгоритму.

У розділі дослідження ефективності було проведено порівняльний аналіз показників послідовного та паралельного алгоритмів.

КЛЮЧОВІ СЛОВА: ПАРАЛЕЛІЗАЦІЯ ЗА ДАНИМИ, НЕЙРОННА МЕРЕЖА, NN, ПЕРЦЕПТРОН, БАГАТОШАРОВИЙ ПЕРЦЕПТРОН, MLP, АЛГОРИТМ НАВЧАННЯ, ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ, ПРИСКОРЕННЯ, ПРОЕКТУВАННЯ, АНАЛІЗ.

ЗМІСТ

ВСТУП

У сучасному світі штучний інтелект і, зокрема, нейронні мережі стають все більш розповсюдженими. Широке застосування нейронних мереж від автоматизації процесів до розпізнавання зображень та аналізу даних підкреслює їх значення у розвитку інформаційних технологій. Проте, ефективність навчання таких мереж часто обмежена обчислювальними можливостями сучасної техніки, особливо коли мова йде про великі обсяги даних і складні моделі.

Паралельне програмування відіграє ключову роль у оптимізації процесів навчання нейронних мереж, оскільки воно дозволяє розподілити обчислювальне навантаження між кількома процесорами або машинами. Ця техніка не тільки знижує час навчання, але й збільшує ефективність використання ресурсів, що є критично важливим для обробки великих наборів даних, що використовуються для тренування.

Python це найбільш розповсюджена мова програмування у сфері аналізу даних і машинного навчання, має багатий набір бібліотек та інструментів для розробки нейронних мереж та виконання математичних операцій. Також ця мова підтримує паралельне програмування, що дозволяє реалізовувати алгоритми що використовують паралелізм.

В цій курсовій буде імплементовано алгоритм навчання нейронної мережі з використанням мови Python, а саме її бібліотеки Multiprocessing. За її допомогою буде створена паралельна реалізація алгоритму, що пришвидшить процес тренування мережі на доступних даних.

1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

1.1 Опис нейронної мережі, існуючих алгоритмів її навчання, та способів її використання.

Нейронна мережа — це модель машинного навчання, створена для імітації функцій і структури людського мозку.

Найпростішою нейронною мережею є Перцептрон, також відомий як нейрон, одношарова нейронна мережа, яка складається з вхідного та вихідного рівня. Це бінарний класифікатор, який приймає рішення шляхом обчислення зваженої суми вхідних характеристик і передачі її через функцію активації, яка додає нелінійність в модель.[1] Коефіцієнти цієї зваженої суми називають вагами. Вихідне значення отримане після застосування функції активації називають активацією нейрона. Можливості перцептрона обмежені; він може розв’язувати лише лінійно розділювані функції. Для подолання цієї проблеми було створено багатошаровий перцептрон(MLP) — нейронну мережу що складається з шарів щільно з’єднаних нейронів(Рисунок 1.1)

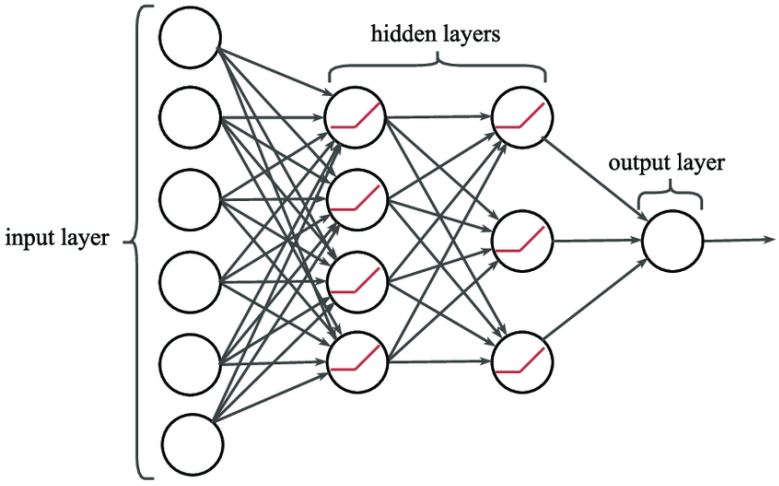

Рисунок. 1.1. – Ілюстрація тришарового MLP[2]

Функція нейронної мережі найкраще описується Теоремою Цибенка, також відомою як універсальна теорема апроксимації[3]. Ця теорема стверджує, що нейронна мережа з одним прихованим шаром, що містить скінченну кількість нейронів, може апроксимувати будь-яку неперервну функцію на компактній

підмножині n-вимірного простору. Ця теоретична гарантія доводить що нейронні мережі можуть бути використані для моделювання будь-якої функції.

Для навчання нейронної мережі використовується алгоритм, відомий як зворотне поширення(backpropagation)[4], який був популяризований у 1980-х роках. Зворотне розповсюдження — це метод ітераційної оптимізації, який регулює ваги мережі, щоб мінімізувати похибку між прогнозованими результатами та фактичними цілями. Він складається з двох основних фаз: прямого поширення та зворотного поширення. На етапі зворотного поширення алгоритм обчислює градієнт функції втрат щодо кожної ваги мережі за допомогою ланцюгового правила обчислення похідних. Ці градієнти вказують напрямок і величину коригувань, що мають бути застосовані до ваг, щоб зменшити помилку.

Зворотне поширення має кілька недоліків: воно потребує великих наборів даних для ефективного навчання, залежить від правильного вибору гіперпараметрів (наприклад, швидкості навчання та розміру міні-батчів), і схильне до перенавчання, якщо не використовувати відповідні методи регуляризації, а також має проблему зникнення градієнта в глибоких мережах.

1.2 Опис та реалізація послідовного алгоритму

В даній роботі ми використовуватимемо MLP з двома прихованим шарами, оскільки така мережа відповідає вимогам UAT. Вибір спрощеної архітектури дозволяє мінімізувати вплив недоліків нейронних мереж на результати дослідження швидкодії послідовного та паралельного алгоритмів.

Набір усіх даних доступних для тренування моделі називається епохою. Коли ми говоримо що мережа була натренована на одній епосі це означає що мережа “побачила” весь тренувальний набір даних один раз. Ми можемо тренувати модель на одних і тих самих даних декілька разів, що може призвести до покращення її ефективності.

При тренуванні мережі епоха що подаються на вхід зазвичай поділена на блоки, так звані міні-батчі(mini-batches). Після обробки кожного такого блоку

алгоритм навчання мережі оновлює параметри моделі, а саме ваги та відступи. Розмір таких блоків визначається гіперпараметром. Розмір блоку в найпростішій реалізації градієнтного спуску дорівнює одиниці, тобто ваги моделі оновлюються після кожного прикладу що подається на вхід. В такому випадку вийнятки, що можуть бути присутні в даних, будуть сильно впливати на параметри моделі що може негативно відобразитись на її результативності. Щоб запобігти цьому розмір міні-батчу встановлюється на значення більші одиниці, і для оновлення ваг мережі використовується середній градієнт обрахований для всіх зразків всередині міні-батчу.

Інший параметр що впливає на швидкість навчання моделі є learning rate. Його значення визначає з якою “швидкістю” ваги моделі змінюють своє значення в протилежному напрямку до градієнта. Занадто високий learning rate може призвести до того що алгоритм не зможе “збігтись” до оптимального значення, тоді як занадто низький може сповільнити процес навчання.

1.3 Опис та реалізація паралельного алгоритму

Розглянемо існуючі методи що застосовуються для паралелізації алгоритмів навчання нейронних мереж.

● Паралелізація за даними (Data Parallelism):

Процес навчання проходить наступним чином: спершу тренувальні дані поділяються на так звані міні-батчі, котрі далі розподіляються серед декількох обчислювальних одиниць якими в цій курсовій виступають процеси, але також це можуть бути GPU, TPU. Кожен такий вузол самостійно виконує обчислення градієнтів для призначеного йому міні-батчу. Потім, градієнти з усіх вузлів агрегуються - вони можуть бути або усереднені, або сумовані. Параметри моделі оновлюються на основі отриманого узагальненого градієнту. Оновлення параметрів відбувається синхронно. Цей метод був застосований у написанні курсової роботи[5]. (Рисунок 1.2)

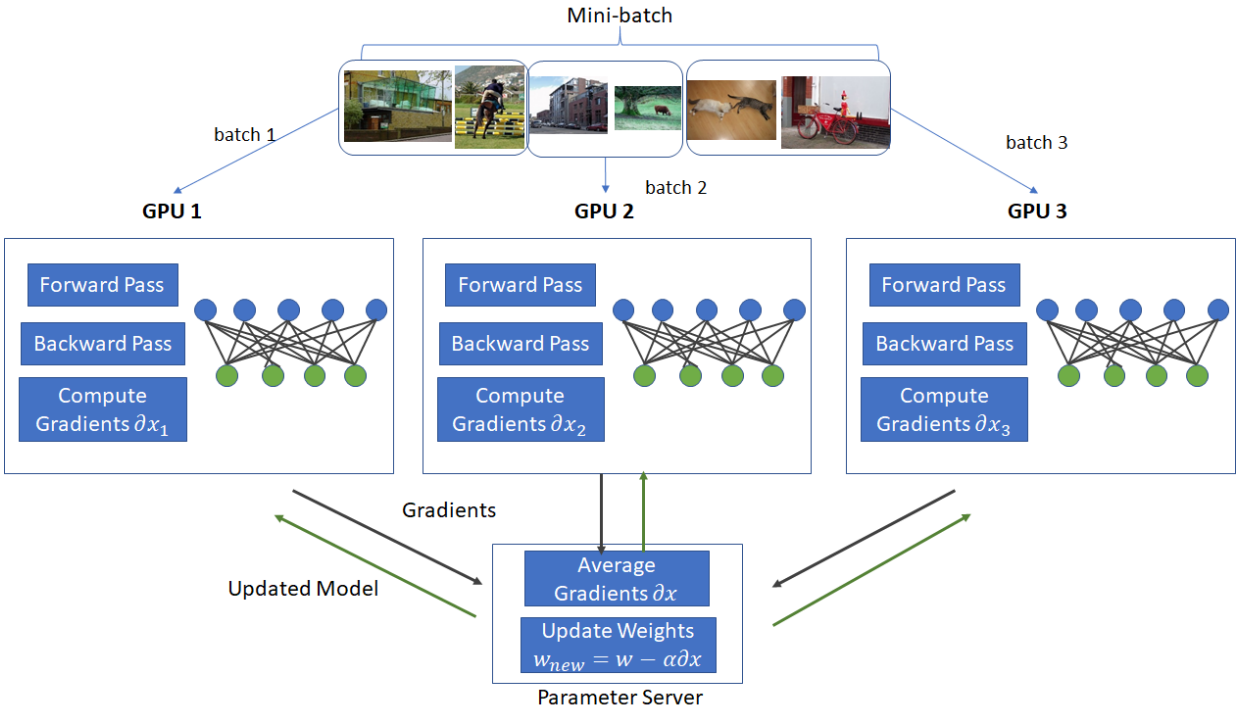

Рисунок 1.2 — Ілюстрація паралелізму за даними.[6]

- Паралелізація по шарам (Pipeline Parallelism): це один з видів паралелізації за моделлю. В цьому методі послідовні шари моделі розподіляються між кількома обчислювальними одиницями(наприклад GPU). Кожен GPU зберігає лише свій шар моделі, завдяки чому та сама модель споживає пропорційно менше пам'яті на кожному GPU. Розділити велику модель на послідовні блоки шарів відносно просто. Проте, через послідовну залежність між вхідними та вихідними даними шарів, проста реалізація може призвести до значних простоїв, коли обчислювальний вузол чекає на дані від попереднього вузла, які використовуються як вхідні дані. Ці періоди очікування називаються "бульбашками", що веде до недовикористання обчислювальних ресурсів вузла.[5][7]

- Паралелізація за тензорами (Tensor parallelism): це один з видів паралелізації за моделлю. Паралелізація по шарам ділить модель "вертикально" за шарами. Також можливе "горизонтальне" розбиття певних операцій в межах шару, що зазвичай називають тренуванням з паралелізмом за тензорами. Для багатьох сучасних моделей, вузьким місцем у тренуванні є множення великих матриць. Результат множення матриць можна уявити як набір скалярних

добутків між парами рядків та стовпців. Для розпаралелювання цієї операції ми можемо виконувати незалежні скалярні добутки на різних GPU, або обчислювати частини кожного скалярного добутку на різних GPU та підсумовувати результати. Це дозволяє обробляти більші тензори та моделі, які не вміщуються в пам’ять одного пристрою.[5] [8]

- Mixture-of-Experts (MoE):

В цьому підході лише частина мережі використовується для обчислення результату для вхідних даних. Такої поведінки можна досягти з використанням деількох наборів ваг, з яких мережа може вибрати потрібний за допомогою механізму gating на етапі inference. Це дозволяє використовувати в моделі більше параметрів без збільшення обчислювальних витрат. Кожен набір ваг називається "експертом" у надії, що мережа буде тренувати цих «експертів» для виконання специфічних завдань. Різні експерти можуть бути розміщені на різних обчислювальних одиницях.[5][9]

Методи паралелізації наведені вище можна розділити на дві категорії: ті що використовуються у випадку коли модель надто велика щоб бути натренованою на одному обчислювальному вузлі, та ті основною метою якихє пришвидшення тренування моделі. Паралелізація за тензорами та MoE відносяться до першої категорії. Паралелізація по шарам може бути використана для прискорення, але лише якщо вона реалізована разом з паралелізацією за даними. Отже в даній курсовій ми використовуватимемо паралелізацію за даними.

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

2.1 Проектування послідовного алгоритму

Для демонстрації роботи мережі обрано задачу інтерполяції складної нелінійної функції. Мережа буде використана щоб передбачити значення функції в заданій точці. Таким чином буде продемонстровано здатність нейронних мереж до апроксимації функцій.

Для моделювання використаємо функцію Растрігіна[10]. Функція Растрігіна — це неопукла функція, яка часто використовується для оцінки алгоритмів оптимізації. Вона є типовим прикладом нелінійної функції з багатьма локальними мінімумами, що робить її особливо складною для оптимізаційних алгоритмів, які прагнуть знайти глобальний мінімум. В даному випадку ми не будемо шукати мінімум а просто використаємо її для моделювання.

Математичне формулювання функції наведено у формулі (2.1):

$$f(x) = An + \sum_{i=1}^{n}\left(x_i^2 - A\cos(2\pi x_i)\right) \quad (2.1)$$

де:

$x = (x_1, \dots, x_n)$ — вектор вхідних змінних

n — кількість вимірів (змінних) функції;

A — константа, зазвичай вибрана як 10

Для полегшення задачі візуалізації результатів тренування мережі використовуватимемо функцію Растрігіна в двох вимірах. Вхідними даними для мережі буде точка з координатами (x_1, x_2) а виходом— значення функції в цій точці. Зображення згенерованих даних для тренування моделі наведене на рисунку 2.1

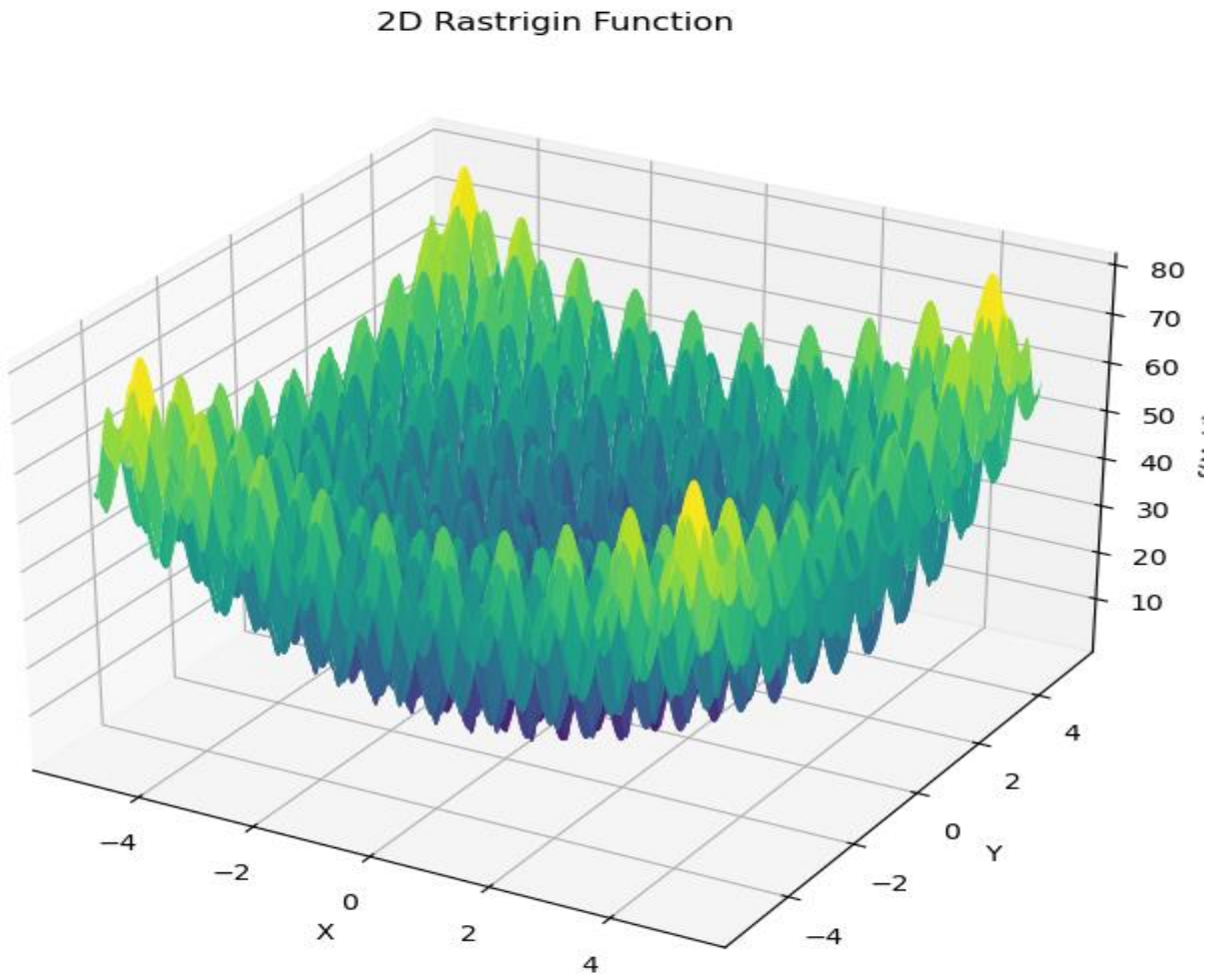

Рисунок 2.1— двовимірна функція Растрігіна на проміжку від -2π до 2π.

Архітектура мережі була визначена експериментальним шляхом. Оптимальна кількість шарів для моделювання обраної функції 3. Кожен прихований шар містить 128 нейронів. Обидва шари використовують функцію активації Сигмоїд. Швидкість навчання встановлено як 0.05. Оптимальний Розмір міні-батчу було визначено за допомогою експерименту і встановлено як 256.

2.2 Реалізація послідовного алгоритму

```
Function rastrigin(X)
    A = 10
    return 20 + (X[0]^2 - A * cos(2 * π * X[0])) + (X[1]^2 - A * cos(2 * π * X[1]))

Function generate_data(n_samples, n_start, n_end)
    x = Generate linear space from n_start to n_end with n_samples steps
    y = Generate linear space from n_start to n_end with n_samples steps
    Create meshgrid from x and y
    Flatten and transpose meshgrid to create X_grid
    z = rastrigin(X_grid)
    return X_grid, z

Function sigmoid(z)
    return 1 / (1 + e^(-z))

Function sigmoid_derivative(z)
    return z * (1 - z)

Function mse_loss(y_true, y_pred)
    return Average of (y_true - y_pred)^2

Function forward(X, W1, b1, W2, b2, W3, b3)
    z1 = X * W1 + b1
    a1 = sigmoid(z1)
    z2 = a1 * W2 + b2
    a2 = sigmoid(z2)
    z3 = a2 * W3 + b3
    return z3, a1, a2

Function backward(X, y, output, a1, a2, W1, W2, W3)
    m = Number of examples in y
    Calculate gradient for output layer
    Calculate gradient for layer 2
    Calculate gradient for layer 1
    Return gradients for weights and biases

Function update_parameters(W1, b1, W2, b2, W3, b3, gradients, learning_rate)
    Update each weight and bias using its corresponding gradient and learning rate
    return Updated weights and biases

Function process_batch(batch, W1, b1, W2, b2, W3, b3)
    Unpack batch into X_batch and y_batch
    output, a1, a2 = forward(X_batch, W1, b1, W2, b2, W3, b3)
    gradients = backward(X_batch, y_batch, output, a1, a2, W1, W2, W3)
    return gradients
```

Рисунок 2.2—Псевдокод допоміжних функцій

```
Function fit_batch(X, y, epochs, batch_size, W1, b1, W2, b2, W3, b3, learning_rate, batch_group_size):
    for epoch in range(epochs):
        permutation = np.random.permutation(X.shape[0])
        X_shuffled = X[permutation]
        y_shuffled = y[permutation]

        batches = [(X_shuffled[i:i + batch_size], y_shuffled[i:i + batch_size])
                   for i in range(0, X.shape[0], batch_size)]

        batch_groups = [batches[i:i + batch_group_size]
                        for i in range(0, len(batches), batch_group_size)]

        for i, batch_group in enumerate(batch_groups):
            group_gradients = empty array

            for batch in batch_group:
                gradients = process_batch(batch, W1, b1, W2, b2, W3, b3)
                group_gradients.append(gradients)

            avg_gradients = list of gradients averaged over the batch_group
            W1, b1, W2, b2, W3, b3 = update_parameters(
                W1, b1, W2, b2, W3, b3, avg_gradients, learning_rate)
```

Рисунок 2.3—Псевдокод послідовного алгоритму тренування нейромережі

Алгоритм навчання включає наступні кроки:

1) Перемішування вхідних даних для заданої епохи
2) Розбиття вхідних даних на порції розміру batch_group_size.
3) Для кожної групи розрахувати градієнт. Це виконується шляхом запуску process_batch що виконує пряме та зворотне поширення.
4) Знайти усереднений градієнт для групи. Це потрібно для згладження значення градієнтів, щоб зменшити вплив винятків в даних на процес навчання. З іншого боку якщо збільшити розмір групи на якій виконується узагальнення ми ризикуємо надто сповільнити процес навчання що приведе до поганих результатів роботи мережі.
5) Оновити параметри моделі(ваги та зміщення)
6) Повторюємо кроки 1-5 для заданої кількості епох або поки не досягнута задовільна точність мережі.

Розрахуємо часову складність алгоритму навчання. Часова складність дорівнює кількості елементарних операцій що виконуються алгоритмом. В випадку послідовного алгоритму кількість цих операцій дорівнює кількості даних що проходять через мережу за час тренування помножена на кількість ваг що присутні в мережі. Отже часова складність дорівнює: $O(e \cdot n \cdot w)$, де $w =$

$(i \cdot h + h \cdot h + h \cdot o)$ за умови що i — кількість вхідних нейронів, h — розмір прихованого шару, o— кількість вихідних нейронів.

Лістинг коду наведений у додатку А.

2.3 Тестування послідовного алгоритму

Щоб провести тестування алгоритму використаємо метрику R2[12]. На рисунках 2.5, 2.6 та 2.7 нижче також видно значення середньоквадратичної помилки для мережі натренованої на трьохсот епохах. Створимо таблицю 2.1 що демонструє взаємозв'язок якості моделі(R2) від кількості даних що надаються їй для тренування. Загальну кількість даних що доступні моделі можна обчислити помноживши кількість епох на кількість даних що містяться в одній епосі.

Таблиця 2.1 — Залежність метрики R^2 від кількості даних для тренування паралельного алгоритму.

Кількість епох	Розмір епохи	R^2
100	40,000	0.5388
100	90,000	0.5522
100	160,000	0.6733
200	40,000	0.5824
200	90,000	0.6750
200	160,000	0.7927
300	40,000	0.5877
300	90,000	0.7606
300	160,000	0.9549

Для встановлення якості результатів що надає модель, побудуємо графік що показує розбіжності між реальними значенням функції Растрігіна для тестових даних та значеннями функції що передбачила натренована модель.(Рисунок 2.4)

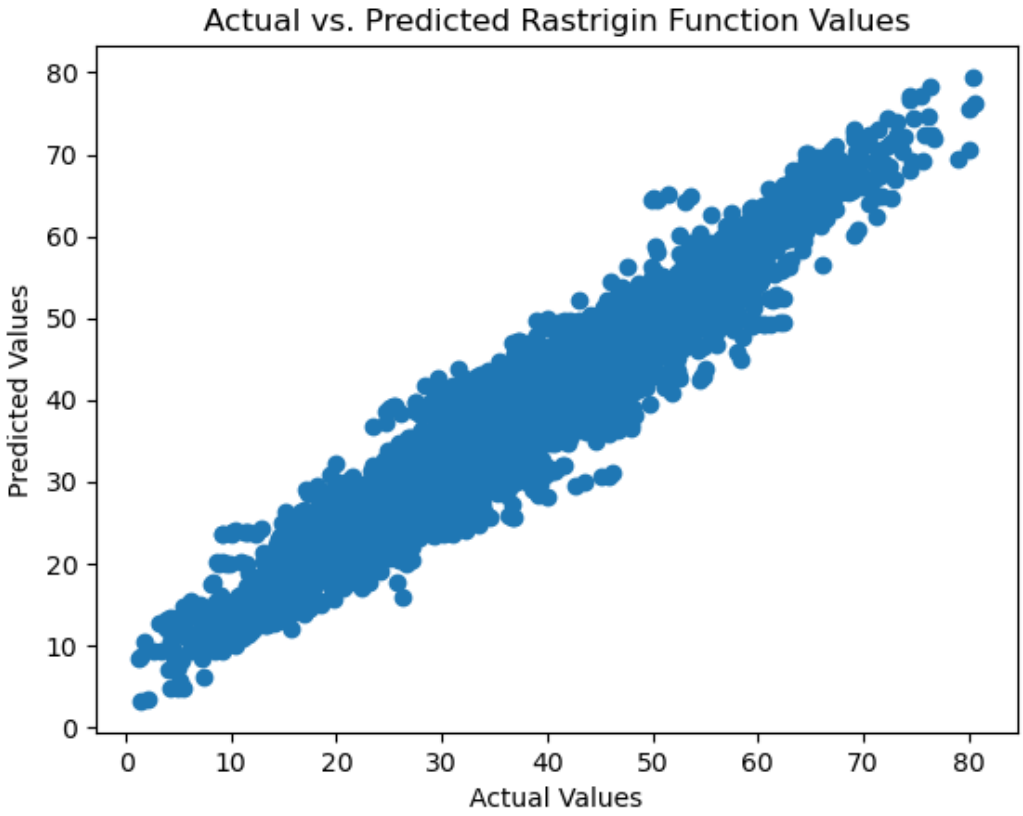

Рисунок 2.4—Графік реальних і передбачених значень функції Растрігіна

Як видно з результатів, графік виглядає як бісектриса першого квартиля системи координат, це свідчить що більшість передбачених значень співпадають з реальними.

Розглянемо також тренд спадання середньоквадратичної помилки(MSE) зі збільшенням кількості епох на яких тренується модель.

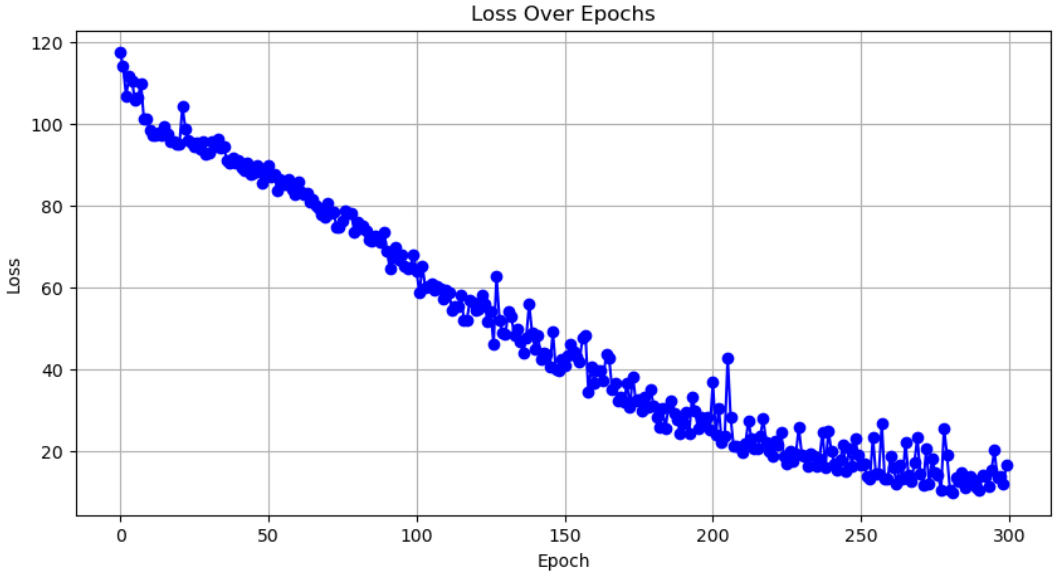

Рисунок 2.5— Графік MSE зі зростанням кількості епох для тренування.

2.4 Аналіз швидкодії послідовного алгоритму

Для аналізу швидкодії будемо тренувати модель на вхідних даних різного розміру. Для отримання точних значень часу виконання епохи, додамо код що буде розраховувати середній час тренування однієї епохи, для кількості епох що буде задана в конфігурації алгоритму. Значення кількості епох не буде змінюватись протягом усього тестування, щоб забезпечити стабільність процесу усереднення. На рисунках 2.6-2.8 можна побачити лог часу тренування епох.

```
Epoch 297/300, Loss: 85.95401163808457, Time: 0.20100116729736328
Epoch 298/300, Loss: 85.30270413540102, Time: 0.20430779457092285
Epoch 299/300, Loss: 85.29586973916425, Time: 0.19199872016906738
Epoch 300/300, Loss: 85.00522913099937, Time: 0.19600152969360352
```

Рисунок 2.6— Час тренування однієї епохи послідовного алгоритму для 40000 вхідних даних.

```
Epoch 297/300, Loss: 46.087061591339356, Time: 0.43000125885009766
Epoch 298/300, Loss: 51.576274548047614, Time: 0.4790005683898926
Epoch 299/300, Loss: 52.012756572501154, Time: 0.49347949028015137
Epoch 300/300, Loss: 51.2326465252086, Time: 0.447035551071167
```

Рисунок 2.7— Час тренування однієї епохи послідовного алгоритму для 90000 вхідних даних.

```
Epoch 297/300, Loss: 8.081158084085025, Time: 0.7703375816345215
Epoch 298/300, Loss: 9.96933693398054, Time: 0.7644093036651611
Epoch 299/300, Loss: 7.232563690534621, Time: 0.7590219974517822
Epoch 300/300, Loss: 9.096050569255096, Time: 0.7591795921325684
```

Рисунок 2.8— Час тренування однієї епохи послідовного алгоритму для 160000 вхідних даних.

Середній час тренування послідовного алгоритму на одній епосі наведено у таблиці 2.2

Таблиця 2.2— Середній час тренування епохи.

Кількість елементів в одній епосі	Середній час тренування однієї епохи в секундах
2,500	0.01234
10,000	0.05033
40,000	0.19556
90,000	0.45111
160,000	0.77502
250,000	1.17206
360,000	1.77066

Щоб краще зрозуміти залежність між цими двома показниками побудуємо графік. На рисунку 2.9 бачимо лінійну залежність, що має сенс, оскільки зі зростанням розміру епохи зростає кількість разів що ми оновлюємо ваги моделі, обчислюючи градієнт.

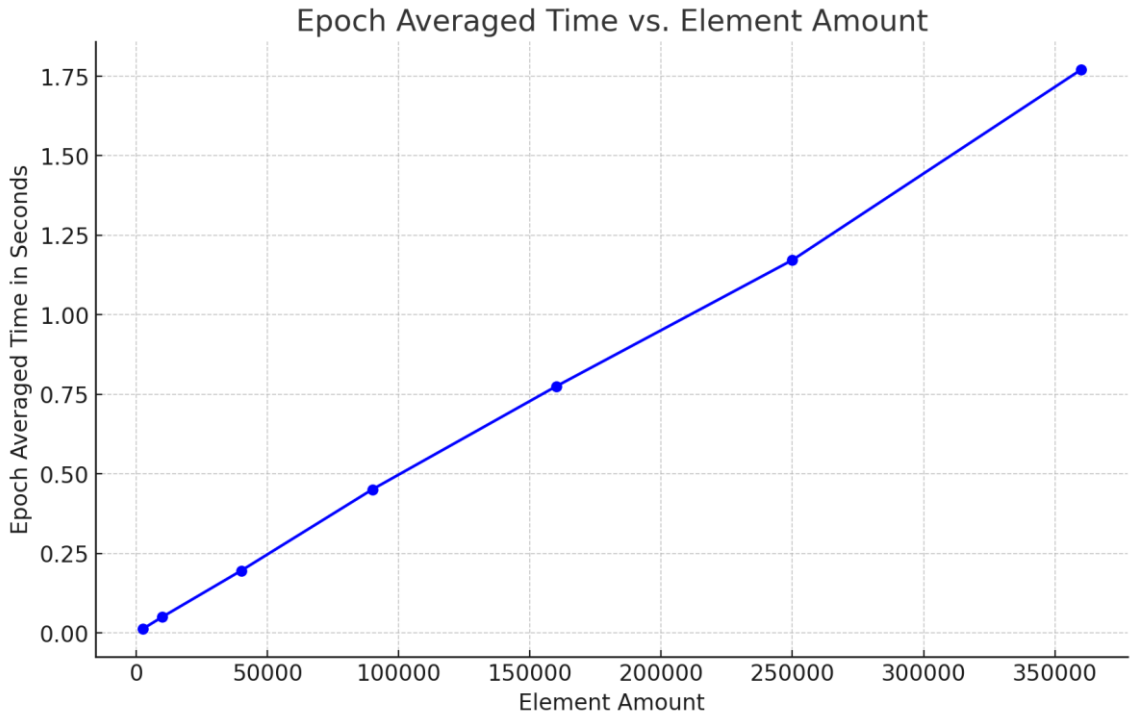

Рисунок 2.9— Середній час тренування епохи в секундах в залежності від кількості елементів в одній епосі.

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

Для розробки паралелізованого алгоритму навчання нейронної мережі було обрано мову Python —динамічно типізовану мову високого рівня. вона має простий синтаксис та забезпечує велику кількість бібліотек для проведення паралельних(parallel) та одночасних(concurrent) обчислень.

Python дозволяє створювати паралельні обчислення двома основними методами: за допомогою багатопоточності (multithreading) та багатопроцесорності (multiprocessing). Оскільки в даній курсовій ми працюємо з CPU інтенсивною задачею, в подальшому розглядатимемо для реалізації бібліотеки та модулі що забезпечують multiprocessing.

Стандартний інтерпретатор Python (CPython), який ми використовуємо в цій курсовій, має певні обмеження, такі як Global Interpreter Lock (GIL)[13], який ускладнює виконання багатопроцесорних обчислень з істинною паралельністю на одному інтерпретаторі. Це необхідно, тому що внутрішня пам'ять Python не є безпечною для використання в багатопоточних програмах. Для обходу цього обмеження Python надає наступні засоби:

- *Multiprocessing*: ця бібліотека дозволяє створювати окремі процеси, кожен з яких має свій власний екземпляр інтерпретатора Python з власним GIL. Це робить можливим паралельне використання декількох ядер процесора, але також створює мінуси, як от те, що кожен процес займає більше пам’яті і потребує тривалого часу для розгортання. Бібліотека дозволяє використовувати багатопроцесорність за допомогою класів Process, Pool (для створення пулу процесів), Queue та Pipe (для обміну даними між процесами) [14]
- *concurrent.futures*: цей модуль є обгорткою навколо Multiprocessing і надає високорівневий інтерфейс для асинхронного виконання завдань. Він дозволяє використовувати як багатопоточність, так і багатопроцесорність через класи ThreadPoolExecutor та ProcessPoolExecutor[15]. Недоліком простоти є втрата контролю. Одним

з прикладів того, що можна зробити в multiprocessing, але не можна легко реалізувати в concurrent.futures, є використання спільної пам'яті між процесами. В multiprocessing є модулі Value та Array, які дозволяють створювати спільні змінні та масиви, які можуть використовувати кілька процесів для обміну даними.

- *mpi4py*: є бібліотекою для Python, яка забезпечує підтримку стандарту MPI (Message Passing Interface) для паралельних і розподілених обчислень. Вона використовує мережеві протоколи для обміну даними між процесами, які можуть знаходитися на різних фізичних вузлах.[16]

Оскільки ми хочемо мати повний контроль над процесом виконання, в даній курсовій ми використаємо бібліотеку Multiprocessing для розробки паралельного алгоритму. Було обрано саме її а не mpi4py, тому що multiprocessing доступна без додаткових налаштувань середовища, має простіший інтерфейс і є достатньою для виконання вимог цієї роботи. Ця бібліотека дозволяє ефективно паралелізувати обчислення, необхідні для обробки великих обсягів даних та великих наборів параметрів у нейронних мережах.

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

4.1 Проектування паралельного алгоритму

Архітектура мережі для паралельного алгоритму навчання залишається незмінною, в порівнянні з тією що наведена в пункті 2.1 курсової. Мережа приймає на вхід точку з незалежними координатами x, y на проміжку від -2π до 2π та повертає значення функції Растрігіна в цій точці[10]. Оптимальна кількість шарів для моделювання обраної функції 3. Кожен прихований шар містить 128 нейронів. Обидва шари використовують функцію активації Сигмоїд. Швидкість навчання встановлено як 0.05. Оптимальний Розмір міні-батчу було визначено за допомогою експерименту і встановлено як 256. Для тренування мережі використовується алгоритм стохастичного градієнтного спуску.

Окремо варто згадати про параметр batch_group_size, що присутній в послідовній реалізації алгоритму. Значення цього параметру визначає після обробки скількох міні-батчів даних алгоритм оновить параметри моделі, використовуючи середнє значення отриманих в міні-батчах градієнтів. В паралельній реалізації цей параметр має додаткове значення: він вказує на максимальний розмір партії міні-батчів, для яких обчислення будуть виконуватись в паралельному режимі. Значення параметру регулює швидкість навчання мережі, оскільки впливає напряму на те, скільки разів параметри мережі будуть оновлені при тренуванні на даних розміром n. Оптимальне значення цього параметру буде встановлене експериментальним шляхом у розділі 5 курсової.

Опис основних кроків алгоритму навчання нейронних мереж що використовує паралелізацію за даними:

1) Ініціалізація ваг та зміщень мережі. Ваги ініціалізуються вибіркою з нормального розподілу— це необхідно щоб уникнути проблеми зникаючого градієнту[11]. Генерація або завантаження даних для тренування моделі.

2) Перемішування вхідних даних для заданої епохи. Необхідно щоб забезпечити рівномірний розподіл даних всередині міні-батчів.
3) Розбиття даних на міні-батчі. Розрахунок середнього градієнту міні-батча замість градієнтів індивідуальних прикладів дозволяє зменшити вплив вийнятків в даних на тренування моделі.
4) Розбиття вхідних даних на порції розміру batch_group_size. Кожна група батчів буде виконуватись паралельно.
5) Паралельне обчислення градієнтів: Для кожного міні-батча в групі:
    a. Виконується прямий прохід через нейронну мережу на даних для отримання значення функції втрат, в даному випадку середньоквадратичної похибки.
    b. За допомогою значення похибки розраховуються градієнти ваг та зміщень мережі. Цей крок має назву зворотне поширення помилки(backpropagation).
6) Знайти усереднений градієнт для групи.
7) Оновити параметри моделі(ваги та зміщення)
8) Повторюємо кроки 1-5 для заданої кількості епох або поки не досягнута задовільна точність мережі.

4.2 Реалізація паралельного алгоритму

```
Function rastrigin(X)
    A = 10
    return 20 + (X[0]^2 - A * cos(2 * π * X[0])) + (X[1]^2 - A * cos(2 * π * X[1]))

Function generate_data(n_samples, n_start, n_end)
    x = Generate linear space from n_start to n_end with n_samples steps
    y = Generate linear space from n_start to n_end with n_samples steps
    Create meshgrid from x and y
    Flatten and transpose meshgrid to create X_grid
    z = rastrigin(X_grid)
    return X_grid, z

Function sigmoid(z)
    return 1 / (1 + e^(-z))

Function sigmoid_derivative(z)
    return z * (1 - z)

Function mse_loss(y_true, y_pred)
    return Average of (y_true - y_pred)^2

Function forward(X, W1, b1, W2, b2, W3, b3)
    z1 = X * W1 + b1
    a1 = sigmoid(z1)
    z2 = a1 * W2 + b2
    a2 = sigmoid(z2)
    z3 = a2 * W3 + b3
    return z3, a1, a2

Function backward(X, y, output, a1, a2, W1, W2, W3)
    m = Number of examples in y
    Calculate gradient for output layer
    Calculate gradient for layer 2
    Calculate gradient for layer 1
    Return gradients for weights and biases

Function update_parameters(W1, b1, W2, b2, W3, b3, gradients, learning_rate)
    Update each weight and bias using its corresponding gradient and learning rate
    return Updated weights and biases

Function process_batch(batch, W1, b1, W2, b2, W3, b3)
    Unpack batch into X_batch and y_batch
    output, a1, a2 = forward(X_batch, W1, b1, W2, b2, W3, b3)
    gradients = backward(X_batch, y_batch, output, a1, a2, W1, W2, W3)
    return gradients
```

Рисунок 4.1—Псевдокод допоміжних функцій

```
Function parallel_fit_batch(X, y, epoch_count, batch_size, W1, b1, W2, b2, W3, b3, learning_rate, batch_group_size):
    for epoch in range(epoch_count):
        permutation = np.random.permutation(X.shape[0])
        X_shuffled = X[permutation]
        y_shuffled = y[permutation]

        batches = [(X_shuffled[i:i + batch_size], y_shuffled[i:i + batch_size])
                   for i in range(0, X.shape[0], batch_size)]

        batch_groups = [batches[i:i + batch_group_size]
                        for i in range(0, len(batches), batch_group_size)]

        for i, batch_group in enumerate(batch_groups):
            sending process_batch with [(batch, W1, b1, W2, b2, W3, b3) for batch in batch_group] as parameters to worker processes
            gradients = result returned by worker processes


            avg_gradients = list of gradients averaged over the batch_group
            W1, b1, W2, b2, W3, b3 = update_parameters(
                W1, b1, W2, b2, W3, b3, avg_gradients, learning_rate)
```

Рисунок 4.2—Псевдокод паралельного алгоритму тренування нейромережі

Розрахуємо часову складність алгоритму навчання. Часова складність дорівнює кількості елементарних операцій що виконуються алгоритмом. Для нейромережі кількість цих операцій дорівнює кількості даних що проходить через мережу за час тренування помножена на кількість ваг що присутні в мережі. Оскільки ми розглядаємо паралельний алгоритм, то побудуємо формулу для розрахунку часової складності на одному процесі “працівнику”. Для такого процесу, кількість тренувальних даних буде зменшена в l разів де $l = min(p, s)$. Тут p це кількість доступних програмі процесів та s — кількість батчів для яких розраховується градієнт перш ніж його усереднити(batch_group_size в коді алгоритму).

Отже часова складність дорівнює: $O(e \cdot n/l \cdot w)$, де $w = (i \cdot h + h \cdot h + h \cdot o)$ за умови що i — кількість вхідних нейронів, h — розмір прихованого шару(обидва приховані шари мають однаковий розмір), o— кількість вихідних нейронів.

Лістинг коду наведений у додатку А.

4.3 Тестування паралельного алгоритму

Щоб провести тестування алгоритму використаємо метрику R2[12]. Створимо таблицю 4.1, що демонструє взаємозв’язок якості моделі(R2) від кількості даних що надаються їй для тренування. Загальну кількість доступних

моделі даних можна обчислити помноживши кількість епох на кількість даних що містяться в одній епосі.

Таблиця 4.1 — Залежність метрики R^2 від кількості даних для тренування паралельного алгоритму.

Кількість епох	Розмір епохи	R^2
100	40,000	0.5312
100	90,000	0.5612
100	160,000	0.6572
200	40,000	0.5723
200	90,000	0.6470
200	160,000	0.7316
300	40,000	0.6068
300	90,000	0.7141
300	160,000	0.9183

Для встановлення якості результатів що надає модель, побудуємо графік що показує розбіжності між реальними значенням функції Растрігіна для тестових даних та значеннями функції що передбачила натренована модель.(Рисунок 4.3)

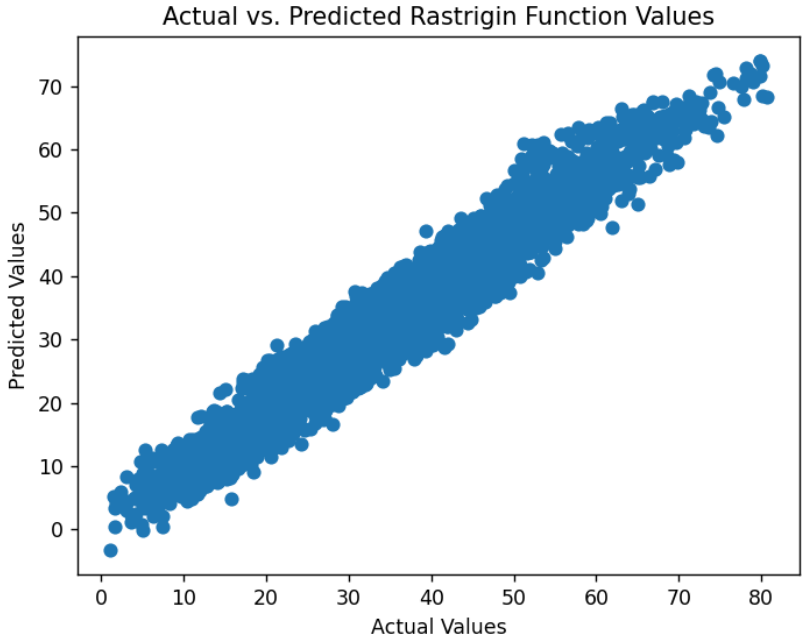

Рисунок 4.3—Графік реальних і передбачених значень функції Растрігіна

Як видно з результатів, графік виглядає як бісектриса першого квартиля системи координат, це свідчить що більшість передбачених значень співпадають з реальними.

Розглянемо також тренд спадання середньоквадратичної помилки(MSE) зі збільшенням кількості епох на яких тренується модель.

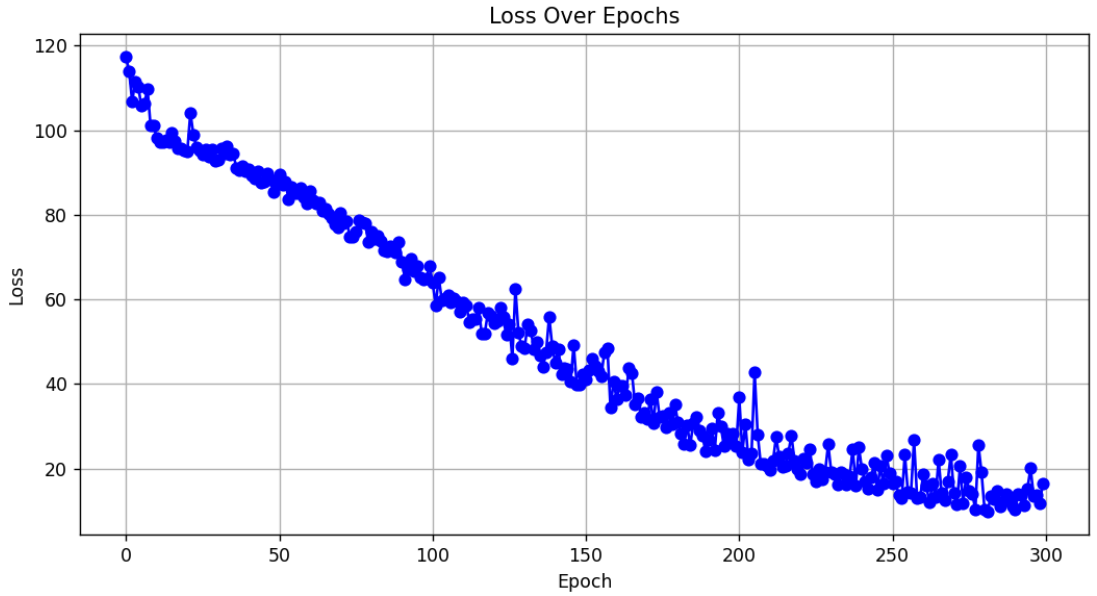

Рисунок 2.5— Графік MSE зі зростанням кількості епох для тренування.

4.4 Аналіз швидкодії паралельного алгоритму

Для аналізу швидкодії алгоритму проаналізуємо його структуру. Весь процес тренування складається з епох. Отже щоб дослідити швидкодію ми можемо знайти усереднений час тренування мережі на одній епосі і зробити висновок про швидкодію всього алгоритму. Для цього зафіксуємо кількість епох на яких тренуватиметься модель, розраховувати середній час тренування однієї епохи, для кількості епох що буде задана в конфігурації алгоритму. На рисунках 2.5-2.6 можна побачити лог часу тренування епох.

```
Epoch 20/300, Loss: 99.60755544626039, Time: 0.12599945068359375
Epoch 21/300, Loss: 100.18011982985439, Time: 0.1110072135925293
Epoch 22/300, Loss: 100.40714689831997, Time: 0.11800360679626465
Epoch 23/300, Loss: 100.3660248771242, Time: 0.12200069427490234
Epoch 24/300, Loss: 99.03649442877125, Time: 0.11700248718261719
Epoch 25/300, Loss: 100.83087761089415, Time: 0.23700284957885742
```

Рисунок 2.5— Час тренування однієї епохи послідовного алгоритму для 40000 вхідних даних.

```
Epoch 15/300, Loss: 98.76642891633773, Time: 0.2610013484954834
Epoch 16/300, Loss: 100.2364469652211, Time: 0.43700122833251953
Epoch 17/300, Loss: 98.14421958690089, Time: 0.31200098991394043
Epoch 18/300, Loss: 113.5944530737512, Time: 0.32200026512145996
Epoch 19/300, Loss: 109.883955442698, Time: 0.2650010585784912
Epoch 20/300, Loss: 101.81667896648884, Time: 0.3209714889526367
```

Рисунок 2.6— Час тренування однієї епохи послідовного алгоритму для 90000 вхідних даних.

```
Epoch 295/300, Loss: 15.28575145582765, Time: 0.5411269664764404
Epoch 296/300, Loss: 20.076749431045187, Time: 0.5510027408599854
Epoch 297/300, Loss: 13.497200789392036, Time: 0.5740005970001221
Epoch 298/300, Loss: 13.687687541222077, Time: 0.5790016651153564
Epoch 299/300, Loss: 11.78792514704331, Time: 0.5450003147125244
Epoch 300/300, Loss: 16.475085377589036, Time: 0.5769975185394287
```

Рисунок 2.7— Час тренування однієї епохи послідовного алгоритму для 160000 вхідних даних.

Середній час тренування послідовного алгоритму на одній епосі наведено у таблиці 2.2

Таблиця 2.2— Середній час тренування епохи.

Кількість елементів в одній епосі	Середній час тренування однієї епохи в секундах
2,500	0.03300
10,000	0.05582
40,000	0.12930
90,000	0.32274
160,000	0.54764
250,000	0.80272
360,000	1.11701

Щоб краще зрозуміти залежність між цими двома показниками побудуємо графік. На рисунку 2.8 бачимо лінійну залежність, що має сенс, оскільки зі зростанням розміру епохи зростає кількість разів що ми оновлюємо ваги моделі, обчислюючи градієнт.

Рисунок 2.8— Середній час тренування епохи в секундах в залежності від кількості елементів в одній епосі.

Але ця лінійна залежність не зростає настільки швидко як тренд що ми спостерігали в послідовному алгоритмі. З цього можна зробити висновок про зростання ефективності паралельного алгоритму при масштабуванні навчальних даних.

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Щоб дослідити ефективність роботи паралельного алгоритму навчання застосуємо метрику прискорення, яка обчислюється за формулою 5.1. Для виконання розрахунків необхідно обрахувати середній час виконання епохи паралельного і послідовного алгоритмів, що було виконано у попередніх розділах.

$$S = \frac{T_{seq}}{T_{par}} \qquad (5.1)$$

Тестування алгоритмів проводилося на обладнанні з наступними характеристиками:

- Назва пристрою: Acer Nitro AN515-45
- Процесор: AMD Ryzen 7 5800H with Radeon Graphics, максимальна частота 3201 МГц, 8 Ядер, 16 Логічних процесів, 64 бітна архітектура
- Операційна система: Microsoft Windows 10 Pro
- Розмір оперативної пам’яті: 16 Гб

Порівняння швидкодії послідовного і паралельного алгоритмів представлено у таблиці 5.1.

Таблиця 5.1. – Назва таблиці

Кількість елементів	Час послідовного алгоритму, мікросекунд	Час паралельного алгоритму, мікросекунд
2,500	12,340	33,000
10,000	50,330	55,820
40,000	195,560	129,300
90,000	451,110	322,740
160,000	775,020	547,640
250,000	1,172,060	802,720
360,000	1,770,660	1,117,010

Для наведених в таблиці значень розрахуємо прискорення(таблиця 5.2):

Таблиця 5.2 — Прискорення досягнуте за допомогою паралелізації алгоритму.

Кількість елементів	Прискорення
2,500	0.374
10,000	0.902
40,000	1.513
90,000	1.398
160,000	1.415
250,000	1.460
360,000	1.586

В розділі 4.1 Проектування паралельного алгоритму згадувалось про гіперпараметр паралельного алгоритму batch_group_size. Цей параметр вказує скільки батчів будуть оброблятися паралельно. Його значення напряму впливає на швидкодію алгоритму, та швидкість збігу мережі. Проведемо аналіз для знаходження оптимального значення batch_group_size. Для цього порівняємо прискорення що надає паралельний алгоритм для 2, 4 та 8 процесів. Для кожного значення кількості процесів встановимо розмір групи батчів(batch_group_size)на значення кратне кількості процесів. При збільшенні batch_group_size, швидкість збігання моделі зменшується, але прискорення що надає алгоритм буде зростати, за умови наявності достатньої кількості даних в епосі.

Кількість епох для тренування встановимо на достатньо велике значення щоб усереднити час тренування епохи. Кількість даних для тренування під час експеримент залишатиметься зафіксованою(40000 елементів). Результати наведені в таблиці 5.3:

Таблиця 5.3 — Пошук оптимальної комбінації значення розміру групи батчів та максимальної кількості доступних програмі процесів.

Кількість процесів(p)	Розмір групи батчів	Середній час тренування однієї епохи, мікросекунд	Прискорення
1	p	216999	1
1	2p	190551	1
1	4p	202878	1
2	p	210997	1,028
2	2p	191999	0,992
2	4p	175561	1,155
4	p	167008	1,299
4	2p	152996	1,245
4	4p	151190	1,341
8	p	171110	1,268
8	2p	149001	1,278
8	4p	147075	1,379

Комбінація що отримала найбільший приріст це 8 доступних процесів і 32 міні-батчі в одній групі(в теорії по 4 батчі на один процес дають узагальнений градієнт за допомогою якого оновлюється модель). Для вхідних даних розміром 40000 елементів ми отримали прискорення від 0.992 до 1,379.

Такі результати приросту як 0,992 можуть бути частково пояснені довгим часом розгортання пулу процесів, що відбувається на першій ітерації навчання. Під час тренування на даних першої епохи, об’єкт пулу процесів створює ресурси процесів працівників, проводить копіювання та кешування даних, що приводить до значної затримки. Ми нівелюємо цей ефект заміряючи середнє значення часу тренування епохи, але навіть при достатньо великій кількості епох цей фактор все одно впливає на результат.

Іншим фактором що може негативно впливати на значення прискорення є малий розмір тренувальних даних. В таблиці 5.2 найкращі показники прискорення(більше 1.4) досягаються для розмірів даних більше 160000. Для даних розміром 50000, витрата часу на розгортання та операції пулу потоків не є виправданою.

ВИСНОВКИ

У ході курсової роботи було детально вивчено особливості алгоритму навчання нейронної мережі. Досліджено способи розподілення обчислень що виконуються в алгоритмі. Спроектовано та розроблено послідовну та паралельну реалізацію алгоритму навчання мережі та проведено порівняльний аналіз результатів роботи алгоритмів.

Тестування паралельної реалізації алгоритму продемонструвало хороші результати прискорення від 1.379 на даних малого розміру до 1.586 разів на більших даних. Є потенціал для ще більшого зростання при збільшенні розміру тренувальних даних. Такий результат доводить ефективність використання паралелізму даних в нейронних мережах з архітектурою багатошаровий перцептрон.

Паралелізація алгоритму навчання за даними збільшує швидкість тренування нейронної мережі та може бути використана разом з іншими способами паралелізації для досягнення ще більшої ефективності.

Під час розробки та в процесі тестування, Python та його бібліотека multiprocessing довели свою ефективність в задачах розподілення обчислень на багатоядерній системі. Це підтверджує його статус основного інструменту в галузі штучного інтелекту та машинного навчання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Perceptrons: An Introduction to Computational Geometry M. Minsky, and S. Papert. MIT Press, Cambridge, MA, USA, (1969). Режим доступу до ресурсу: https://direct.mit.edu/books/monograph/3132/PerceptronsAn-Introduction-to-Computational
2. MLP Illustration. Режим доступу до ресурсу: https://www.researchgate.net/publication/325067027_Climate_Change_and_Power_Security_Power_Load_Prediction_for_Rural_Electrical_Microgrids_Using_Long_Short_Term_Memory_and_Artificial_Neural_Networks
3. Cybenko, G. Approximation by superpositions of a sigmoidal function. Math. Control Signal Systems 2, 303–314 (1989). https://doi.org/10.1007/BF02551274
4. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536. Режим доступу до ресурсу: https://www.nature.com/articles/323533a0
5. Techniques for training large neural networks Lilian Weng, Greg Brockman June 9, 2022. Режим доступу до ресурсу: https://openai.com/index/techniques-for-training-large-neural-networks/
6. Understanding Data Parallelism in Machine Learning December 25, 2017 Режим доступу до ресурсу: https://www.telesens.co/2017/12/25/understanding-data-parallelism-in-machine-learning/
7. On-the-Fly Pipeline Parallelism Authors: I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim SukhaAuthors Info & Claims Режим доступу до ресурсу: https://dl.acm.org/doi/10.1145/2809808
8. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM Deepak Narayanan‡★, Mohammad Shoeybi† , Jared Casper† , Patrick LeGresley† , Mostofa Patwary† , Vijay Korthikanti† , Dmitri

Vainbrand† , Prethvi Kashinkunti† , Julie Bernauer† , Bryan Catanzaro† , Amar Phanishayee∗ , Matei Zaharia‡ †NVIDIA ‡Stanford University ∗Microsoft Research Режим доступу до ресурсу: https://deepakn94.github.io/assets/papers/megatron-sc21.pdf

9. Adaptive Mixtures of Local Experts Robert A. Jacobs Michael l. Jordan Steven J. Nowlan Geoffrey E. Hinton. Режим доступу до ресурсу: https://www.cs.toronto.edu/~hinton/absps/jjnh91.pdf
10. Pohlheim, H. GEATbx Examples: Examples of Objective Functions (2005). Retrieved June 2013, Режим доступу до ресурсу: http://www.geatbx.com/download/GEATbx_ObjFunExpl_v37.pdf
11. THE VANISHING GRADIENT PROBLEM DURING LEARNING RECURRENT NEURAL NETS AND PROBLEM SOLUTIONS Sepp Hochreiter. Режим доступу до ресурсу: https://www.bioinf.jku.at/publications/older/2304.pdf
12. An R-squared measure of goodness of fit for some common nonlinear regression models. Colin Cameron a †, Frank A.G. Windmeijer. Режим доступу до ресурсу: https://doi.org/10.1016/S0304-4076(96)01818-0
13. Python 3.12.4 documentation. Режим доступу до ресурсу: https://docs.python.org/3/
14. Python 3.12.4 Multiprocessing documentation. Режим доступу до ресурсу: https://docs.python.org/3/library/multiprocessing.html
15. Python 3.12.4 documentation. Режим доступу до ресурсу: https://docs.python.org/3/library/concurrent.futures.html
16. MPI for Python. Режим доступу до ресурсу: https://mpi4py.readthedocs.io/en/stable/

ДОДАТКИ

Додаток А. Текст програмного коду

```
from time import time
import numpy as np
import matplotlib.pyplot as plt
import multiprocessing


def calculate_r2(Y_actual, Y_predicted):
    diff = Y_actual - Y_predicted
    mse = np.mean(diff**2)
    variance_Y = np.var(Y_actual)
    r2 = 1 - (mse / variance_Y)
    return r2


def rastrigin(X):
    A = 10
    return A * 2 + (X[:, 0]**2 - A * np.cos(2 * np.pi * X[:, 0])) + (X[:, 1]**2 - A
* np.cos(2 * np.pi * X[:, 1]))


def generate_data(n_samples, n_start, n_end):
    x = np.linspace(n_start, n_end, n_samples)
    y = np.linspace(n_start, n_end, n_samples)
    x, y = np.meshgrid(x, y)
    X_grid = np.vstack([x.ravel(), y.ravel()]).T

    z = rastrigin(X_grid).reshape(-1, 1)
    return X_grid, z


def sigmoid(z):
    return 1 / (1 + np.exp(-z))


def sigmoid_derivative(z):
    return z * (1 - z)


def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)


def forward(X, W1, b1, W2, b2, W3, b3):
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)
    z2 = np.dot(a1, W2) + b2
    a2 = sigmoid(z2)
    z3 = np.dot(a2, W3) + b3
```

```
    return z3, a1, a2


def backward(X, y, output, a1, a2, W1, W2, W3):
    m = y.shape[0]

    d_z3 = output - y
    d_W3 = (1/m) * np.dot(a2.T, d_z3)
    d_b3 = (1/m) * np.sum(d_z3, axis=0, keepdims=True)

    d_a2 = np.dot(d_z3, W3.T)
    d_z2 = d_a2 * sigmoid_derivative(a2)
    d_W2 = (1/m) * np.dot(a1.T, d_z2)
    d_b2 = (1/m) * np.sum(d_z2, axis=0, keepdims=True)

    d_a1 = np.dot(d_z2, W2.T)
    d_z1 = d_a1 * sigmoid_derivative(a1)
    d_W1 = (1/m) * np.dot(X.T, d_z1)
    d_b1 = (1/m) * np.sum(d_z1, axis=0, keepdims=True)

    return d_W1, d_b1, d_W2, d_b2, d_W3, d_b3


def update_parameters(W1, b1, W2, b2, W3, b3, gradients, learning_rate):
    d_W1, d_b1, d_W2, d_b2, d_W3, d_b3 = gradients
    # Update weights and biases
    W1 -= learning_rate * d_W1
    b1 -= learning_rate * d_b1
    W2 -= learning_rate * d_W2
    b2 -= learning_rate * d_b2
    W3 -= learning_rate * d_W3
    b3 -= learning_rate * d_b3
    return W1, b1, W2, b2, W3, b3


def process_batch(batch, W1, b1, W2, b2, W3, b3):
    try:
        X_batch, y_batch = batch
        output, a1, a2 = forward(X_batch, W1, b1, W2, b2, W3, b3)
        gradients = backward(X_batch, y_batch, output, a1, a2, W1, W2, W3)
        return gradients
    except Exception as e:
        print(f"Error processing batch: {e}")
        raise


def parallel_fit_batch(X, y, epochs, batch_size, W1, b1, W2, b2, W3, b3,
learning_rate, batch_group_size):
    pool = multiprocessing.Pool()
    history = []
```

```
    time_array = []
    for epoch in range(epochs):
        total_time = 0
        startTime = time()
        permutation = np.random.permutation(X.shape[0])
        X_shuffled = X[permutation]
        y_shuffled = y[permutation]

        batches = [(X_shuffled[i:i + batch_size], y_shuffled[i:i + batch_size])
                   for i in range(0, X.shape[0], batch_size)]

        batch_groups = [batches[i:i + batch_group_size]
                        for i in range(0, len(batches), batch_group_size)]

        for batch_group in batch_groups:

            gradients = pool.starmap(
                process_batch, [(batch, W1, b1, W2, b2, W3, b3) for batch in
batch_group])

            avg_gradients = [np.sum(
                [grad[i] for grad in gradients], axis=0) / batch_group_size for i
in range(6)]
            W1, b1, W2, b2, W3, b3 = update_parameters(
                W1, b1, W2, b2, W3, b3, avg_gradients, learning_rate)

        total_time = time() - startTime
        time_array.append(total_time)

        output, _, _ = forward(X_shuffled, W1, b1, W2, b2, W3, b3)
        loss = mse_loss(y_shuffled, output)
        history.append(loss)
        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss}, Time: {total_time}")

    pool.close()
    pool.join()
    return W1, b1, W2, b2, W3, b3, time_array, history


def fit_batch(X, y, epochs, batch_size, W1, b1, W2, b2, W3, b3, learning_rate,
batch_group_size):
    time_array = []
    history = []
    for epoch in range(epochs):
        total_time = 0
        startTime = time()
        permutation = np.random.permutation(X.shape[0])
        X_shuffled = X[permutation]
        y_shuffled = y[permutation]
```

```
        batches = [(X_shuffled[i:i + batch_size], y_shuffled[i:i + batch_size])
                   for i in range(0, X.shape[0], batch_size)]

        batch_groups = [batches[i:i + batch_group_size]
                        for i in range(0, len(batches), batch_group_size)]

        for i, batch_group in enumerate(batch_groups):
            group_gradients = []

            for batch in batch_group:
                gradients = process_batch(batch, W1, b1, W2, b2, W3, b3)
                group_gradients.append(gradients)

            avg_gradients = [np.sum(
                [grad[i] for grad in group_gradients], axis=0) / batch_group_size
for i in range(6)]
            W1, b1, W2, b2, W3, b3 = update_parameters(
                W1, b1, W2, b2, W3, b3, avg_gradients, learning_rate)

        total_time = time() - startTime
        time_array.append(total_time)
        output, _, _ = forward(X_shuffled, W1, b1, W2, b2, W3, b3)
        loss = mse_loss(y_shuffled, output)
        history.append(loss)
        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss}, Time: {total_time}")
        print()
    return W1, b1, W2, b2, W3, b3, time_array, history


def main():
    np.random.seed(42)

    input_size = 2
    hidden_size = 128
    output_size = 1
    W1 = np.random.randn(input_size, hidden_size)
    b1 = np.zeros((1, hidden_size))
    W2 = np.random.randn(hidden_size, hidden_size)
    b2 = np.zeros((1, hidden_size))
    W3 = np.random.randn(hidden_size, output_size)
    b3 = np.zeros((1, output_size))
    learning_rate = 0.05
    batch_group_size = 8  # multiprocessing.cpu_count()

    input_range = 5.12
    input_density = 400

    X_train, Y_train = generate_data(input_density, -input_range, input_range)
    X_test = np.random.uniform(-input_range, input_range, (5000, 2))
    Y_test = rastrigin(X_test)
```

```
    W1, b1, W2, b2, W3, b3, time_array, history = parallel_fit_batch(
        X_train, Y_train, epochs=20, batch_size=256, W1=W1, b1=b1, W2=W2, b2=b2,
W3=W3, b3=b3, learning_rate=learning_rate, batch_group_size=batch_group_size)
    # W1, b1, W2, b2, W3, b3, time_array, history = fit_batch(X_train, Y_train,
epochs=20, batch_size=256, W1=W1, b1=b1, W2=W2, b2=b2,
    #         W3=W3, b3=b3, learning_rate=learning_rate,
batch_group_size=batch_group_size)

    print("Epoch time", sum(time_array)/len(time_array))

    Y_pred, _, _ = forward(X_test, W1, b1, W2, b2, W3, b3)
    r2 = calculate_r2(Y_test, Y_pred.flatten())
    print(f"R^2 score: {r2:.4f}")

    plt.figure(figsize=(10, 5))
    plt.plot(history, marker='o', linestyle='-', color='b')
    plt.title('Loss Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.show()

    plt.scatter(Y_test, Y_pred)
    plt.xlabel('Actual Values')
    plt.ylabel('Predicted Values')
    plt.title('Actual vs. Predicted Rastrigin Function Values')
    plt.show()


if __name__ == '__main__':
    main()
```