Міністерство освіти і науки України

Харківський національний університет імені В. Н. Каразіна

Факультет комп’ютерних наук

Кафедра теоретичної та прикладної системотехніки

«Затверджую»
Зав. кафедри теоретичної та
прикладної системотехніки
_____ д.т.н., проф. С. І. Шматков
«___» грудня 2023 р.

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «**КОМП’ЮТЕРНА МОДЕЛЬ ПАРАЛЕЛЬНОЇ РЕАЛІЗАЦІЇ МЕТОДУ ГАУСА З ЕЛЕМЕНТАМИ ШТУЧНОГО ІНТЕЛЕКТУ**»

Захищено на засіданні
Атестаційної комісії № 42
протокол № __ від __.12.2023 р.
Оцінка _____ / _____
Голова Атестаційної комісії
_____ **СКОБ Ю. О.**.
(підпис) (прізвище та ініціали)

Виконав:
студент групи КУ– 61
Галузь знань: 15 – Автоматизація та приладобудування
Спеціальність: 151 – «Автоматизація та комп'ютерно-інтегровані технології»
ВОСТРИШЕВ Ілля Сергійович

Керівник: д.т.н., професор кафедри теоретичної та прикладної системотехніки
ТОЛСТОЛУЗЬКА Олена Геннадіївна

Рецензент:

Харків – 2023

АНОТАЦІЯ

Пояснювальна записка до магістерської кваліфікаційної роботи складається зі вступу, трьох розділів, висновків, списку використаних джерел і чотирьох додатків. Загальний обсяг роботи складає 92 сторінки, із яких 60 сторінок основної частини з 31 рисунком, 3 таблицями, 30 найменувань списку використаних джерел, та чотирма додатками.

В сучасному світі інформаційних технологій є актуальною проблема необхідності оптимізації обчислювальних процесів та підвищення їхньої ефективності в умовах зростаючого обсягу даних та складних обчислювальних завдань. Тому метою роботи було підвищення ефективності процесу обчислень методу Гауса шляхом використання паралельного програмування та елементів штучного інтелекту.

При виконанні кваліфікаційної роботи були використані паралельні обчислення, методи штучного інтелекту та програмні засоби на основі мови програмування C#, такі як Task Parallel Library (TPL), TensorFlow.NET та інші.

У ході виконання роботи було отримано результати, які свідчать про високий рівень ефективності запропонованого підходу до оптимізації методу Гауса та модель паралельної реалізації методу Гауса з елементами штучного інтелекту.

Головною областю використання розробленої моделі є оптимізація обчислень в галузях, де важливо швидко та ефективно розв'язувати великі обчислювальні завдання, такі як розв'язання систем лінійних рівнянь, чисельні симуляції та математичні моделі.

Ключові слова: TensorFlow.NET, метод Гауса, системи лінійні рівнянь, паралельні обчислення, Task Parallel Library, нейронні мережі, ефективність.

ABSTRACT

The explanatory note to the master's thesis consists of an introduction, three chapters, conclusions, a list of references and four appendices. The total volume of the work is 92 pages, including 60 pages of the main part with 31 figures, 3 tables, 30 references, and four appendices.

In the modern world of information technology, there is a pressing problem of the need to optimize computing processes and increase their efficiency in the face of growing data and complex computing tasks. Therefore, the aim of the work was to increase the efficiency of the Gaussian method computation process by using parallel programming and elements of artificial intelligence.

Parallel computing, artificial intelligence methods, and software tools based on the C# programming language, such as Task Parallel Library (TPL), TensorFlow.NET, and others, were used in the qualification work.

In the course of the work, results were obtained that indicate a high level of efficiency of the proposed approach to optimizing the Gaussian method and a model for parallel implementation of the Gaussian method with elements of artificial intelligence.

The main area of application of the developed model is the optimization of computations in areas where it is important to solve large computational problems quickly and efficiently, such as solving systems of linear equations, numerical simulations, and mathematical models.

Keywords: TensorFlow.NET, Gauss method, systems of linear equations, parallel computing, Task Parallel Library, neural networks, efficiency.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

МГ – метод Гауса.

НМ – нейронна мережа;

ReLU (Rectified Linear Unit) - функція активації, яка використовується в нейронних мережах.

RNN (Recurrent Neural Network) - рекурентна нейронна мережа.

C# - мова програмування.

.NET - це платформа для розробки та виконання різноманітних програм, яку розробляє корпорація Microsoft.

TPL (Task Parallel Library) - набір класів та методів у мові програмування C#, який надає можливість використовувати паралельні конструкції для розподіленого виконання завдань.

UML (Unified Modeling Language) - стандартний мовний засіб для візуалізації, проектування та документування систем.

ВСТУП

У сучасному світі спостерігається постійне збільшення обсягів даних. Цей ріст відбувається в багатьох галузях, включаючи бізнес, науку, медицину, інженерію тощо. Інформація, що збирається, потребує аналізу та обробки для виділення цінної інформації і прийняття обґрунтованих рішень.

Метод Гауса вже давно використовується для обчислень у різних галузях, і він є важливим інструментом для розв'язання різноманітних математичних задач. У контексті обробки великих обсягів даних, особливо важливою стає можливість паралельної реалізації методу Гауса. Це дозволить розділити обчислення на декілька паралельних завдань, що виконуються одночасно, забезпечуючи швидку та ефективну обробку даних.

Крім того, використання штучного інтелекту, зокрема нейромереж, може значно підвищити результативність обробки та аналізу даних. Нейронні мережі можуть виявляти складні залежності та патерни в даних, які не завжди можна виявити за допомогою традиційних алгоритмів. Вони широко використовуються для завдань, таких як розпізнавання образів, обробка природної мови, прогнозування та інші.

Метою дослідження є підвищення ефективності процесу обчислень методу Гауса шляхом використання паралельного програмування та елементів штучного інтелекту. Будемо розглядати в якості об’єкту дослідження обчислювальні процеси при реалізації МГ з елементами штучного інтелекту. Методами дослідження були: методи обробки та підготовки даних, використання паралельних обчислень, методи штучного інтелекту та програмні засоби на основі мови програмування C#, такі як Task Parallel Library (TPL), TensorFlow.NET та інші.

Предметом дослідження у даній роботі є комп’ютерні моделі паралельної реалізації методу Гауса з використанням елементів штучного інтелекту на основі обчислювальної системи з високою продуктивністю.

В кваліфікаційній роботі розглядається розробка та реалізація комп'ютерної моделі, яка поєднує метод Гауса, паралельні обчислення та штучні нейронні мережі для ефективної обробки великих обсягів даних, зокрема для розв’язку великих систем лінійних рівнянь. Робота включає аналіз ефективності методу Гауса в обчислювальних задачах, огляд основ паралельних обчислень та розгляд застосування нейромереж у великих обчислювальних завданнях.

У цьому контексті важливим етапом є розробка алгоритму паралельної реалізації методу Гауса, що дозволить розділити обчислення на паралельні задачі для забезпечення ефективності та швидкості обробки даних. До цього також включено інтеграцію нейронної мережі у комп'ютерну модель, вибір та побудову нейромережі, підготовку даних для навчання та налаштування процесу навчання.

Застосування такої комп'ютерної моделі може мати значущий вплив на різні галузі, де важливий аналіз та обробка великих обсягів інформації. Висновки роботи визначають ключові результати та можливі напрямки подальших досліджень.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ МЕТОДУ ГАУСА, НЕЙРОМЕРЕЖ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

1.1 Аналіз методу Гауса в обчислювальних задачах

1.1.1 Опис методу Гауса

Система лінійних алгебраїчних рівнянь - це набір двох або більше рівнянь, які містять однакові змінні. Загальний вигляд системи лінійних рівнянь виглядає наступним чином:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \quad\quad \vdots \quad\quad\quad\quad \vdots \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}, \tag{1.1}$$

де a_{mn} - це коефіцієнти системи, x_n - невідомі змінні, b_m - вільні члени. Часто коефіцієнти та невідомі є дійсними або комплексними числами, але також зустрічаються цілі та раціональні числа, а також поліноми та елементи абстрактної алгебраїчної структури [1].

Один з надзвичайно корисних поглядів полягає в тому, що кожне невідоме є вагою для вектора-стовпця в лінійній комбінації.

$$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}, \tag{1.2}$$

Це дозволяє застосувати всю мову і теорію векторних просторів (або, в більш загальному випадку, модулів). Наприклад, сукупність усіх можливих лінійних комбінацій векторів у лівій частині називається їх проміжком, і рівняння мають розв'язок, коли вектор у правій частині знаходиться в межах цього проміжку. Якщо кожен вектор у проміжку має рівно один вираз у вигляді лінійної комбінації заданих лівих векторів, то будь-який розв'язок є унікальним. У будь-якому випадку, проміжок має базис лінійно незалежних векторів, які

гарантують точно один вираз і кількість векторів у цьому базисі (його розмірність) не може бути більшою за m або n, але може бути і меншою [2]. Це важливо, тому що якщо ми маємо m незалежних векторів, то розв'язок гарантується незалежно від правої частини, а в іншому випадку не гарантується.

Векторне рівняння еквівалентне матричному рівнянню виду:

$$Ax = b \tag{1.3}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix},\ x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},\ b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}, \tag{1.4}$$

де A - матриця розміром m×n, x - вектор-стовпець з n елементами, b - вектор-стовпець з m елементами. Кількість векторів у базисі для проміжку тепер виражається як ранг матриці [3].

Розв’язком лінійної системи називається таке присвоєння значень змінним x_1, x_2, ..., x_n, при якому виконується кожне з рівнянь. Множина всіх можливих розв'язків називається множиною розв'язків. Система лінійних рівнянь може мати різні типи розв'язків:

1. Єдиний розв'язок: система має один і єдиний розв'язок для невідомих.

2. Нескінченно багато розв'язків: система може мати безліч різних рішень.

3. Немає розв'язків: система може бути нерозв'язною, коли не існує жодного набору значень змінних, який задовольняє всі рівняння.

Метод Гауса - це алгоритм для розв'язання систем лінійних рівнянь. Основна ідея методу полягає у тому, щоб використовувати елементарні перетворення над рівняннями системи, зменшуючи її до еквівалентної системи з розширеною матрицею, яка має трикутний або ступеневий вигляд. Потім розв'язок системи знаходиться шляхом зворотного підставлення. Основні етапи методу Гауса [4]:

1. Створення розширеної матриці: об'єднуємо коефіцієнти системи рівнянь з вільними членами у розширену матрицю.

2. Прямий хід (елементарні перетворення): застосовуємо елементарні перетворення (перестановка рядків, множення рядка на константу, додавання одного рядка до іншого) для отримання трикутної або ступеневої форми розширеної матриці.

3. Зворотний хід (знаходження розв'язку): здійснюємо зворотне підставлення для знаходження значень невідомих.

Цей метод особливо ефективний, коли система має велику кількість рівнянь і невідомих. Однак важливо враховувати, що МГ не завжди є оптимальним для всіх типів систем, і у деяких випадках можуть виникнути проблеми. При його використанні в обчислювальних задачах важливо враховувати кілька аспектів та потенційних проблем [5]:

1. Чисельна стійкість: при обчисленні може виникнути втрата стійкості внаслідок скінченної точності чисел з плаваючою комою. Великі числа або малий визначник матриці можуть призвести до неправильних результатів.

2. Ділення на нуль: метод Гауса може призводити до ділення на нуль, особливо при визначенні детермінанта чи при виконанні елементарних перетворень. Це може виникнути, наприклад, коли у системі є лінійно залежні рівняння.

3. Затрати пам'яті та обчислювальний час: МГ вимагає створення розширеної матриці та виконання елементарних перетворень, що може призвести до збільшення обчислювальних затрат та використання пам'яті, особливо для великих систем рівнянь.

4. Системи зі специфічною структурою: у деяких випадках, коли система має специфічну структуру (наприклад, розріджені матриці), інші методи, такі як ітераційні методи, можуть бути більш ефективними.

5. Розширення на неоднорідні системи: метод Гауса призначений для розв'язання однорідних систем рівнянь. Для неоднорідних систем існують модифікації методу, такі як метод знаходження частинного розв'язку.

1.1.2 Ефективність обчислень

Кількість арифметичних операцій, необхідних для виконання скорочення рядків, є одним із способів вимірювання обчислювальної ефективності алгоритму. Наприклад, щоб розв’язати систему з n рівнянь для n невідомих, виконуючи над матрицею операції з рядками, доки вона не набуде ешелонної форми, а потім розв’язувати для кожного невідомого у зворотному порядку, потрібно $n(n + 1)/2$ поділок, $(2n^3 + 3n^2 − 5n)/6$ множень і $(2n^3 + 3n^2 − 5n)/6$ віднімань, загалом приблизно $2n^3/3$ операцій. Таким чином, він має часову складність $O(n^3)$ [3].

Ця складність є хорошим показником часу, необхідного для всього обчислення, коли час для кожної арифметичної операції приблизно постійний. Це у випадку, коли коефіцієнти представлені числами з плаваючою комою або коли вони належать до кінцевого поля. Якщо коефіцієнти є цілими чи раціональними числами, точно представленими, проміжні записи можуть зростати експоненціально, тому бітова складність є експоненціальною. Однак існує варіант елімінації Гауса, який називається алгоритмом Барейса, який уникає цього експоненціального зростання проміжних записів і, з такою самою арифметичною складністю $O(n^3)$, має бітову складність $O(n^5)$.

Цей алгоритм можна використовувати на комп’ютері для систем із тисячами рівнянь і невідомих. Однак вартість стає непомірно високою для систем з мільйонами рівнянь. Ці великі системи зазвичай розв'язуються за допомогою ітераційних методів. Спеціальні методи існують для систем, коефіцієнти яких відповідають регулярному шаблону[3-5].

Щоб привести матрицю $n×n$ до скороченої ешелонної форми за допомогою операцій із рядками, потрібно n^3 арифметичних операцій, що приблизно на 50% більше кроків обчислення. Однією з можливих проблем є чисельна нестабільність, викликана можливістю ділення на дуже малі числа. Якщо, наприклад, провідний коефіцієнт одного з рядків дуже близький до нуля, то для скорочення рядків матриці потрібно буде поділити на це число. Це означає, що будь-яка помилка для числа, близького до нуля, буде посилена. Усунення Гауса

є чисельно стабільним для діагонально домінантних або позитивно визначених матриць. Для загальних матриць Гаусове усунення зазвичай вважається стабільним при використанні часткового повороту, навіть якщо є приклади стабільних матриць, для яких воно нестабільне.

1.2 Паралельні обчислення

1.2.1 Основні принципи, архітектури та методи паралельних обчислень

Традиційно комп'ютерне програмне забезпечення розробляється для послідовних обчислень. Для вирішення проблеми створюється алгоритм, який виконується як послідовна серія інструкцій. Ці інструкції виконуються на одному центральному процесорному блоці комп'ютера, що дозволяє одночасно виконувати лише одну інструкцію. Після завершення поточної інструкції виконується наступна. На відміну від цього, паралельні обчислення використовують декілька обчислювальних елементів одночасно для вирішення проблеми. Це досягається шляхом поділу задачі на незалежні сегменти, що дозволяє кожному обчислювальному елементу виконувати свою частину алгоритму одночасно з іншими. Елементи обробки можуть бути різними і можуть включати ресурси, такі як один комп'ютер з декількома процесорами, кілька комп'ютерів, об'єднаних в мережу, спеціалізоване обладнання або будь-яку їх комбінацію. Паралельні обчислення історично використовувалися для наукових обчислень і моделювання наукових проблем, особливо в природничих та інженерних науках, таких як метеорологія. Це призвело до створення як паралельного апаратного та програмного забезпечення, так і високопродуктивних обчислень [8].

Масштабування частоти було основним фактором підвищення продуктивності комп'ютера. Тривалість виконання програми еквівалентна кількості інструкцій, помноженій на середній час, необхідний для виконання однієї інструкції. При незмінності інших змінних підвищення тактової частоти зменшує середній час виконання однієї інструкції, а отже, зменшує час

виконання всіх комп'ютерних програм. Крім того, підвищення частоти збільшує енергоспоживання процесора. Виробники розробили енергоефективні процесори з декількома ядрами, щоб вирішити проблеми енергоспоживання та перегріву звичайних процесорів. У багатоядерних процесорах кожне ядро працює незалежно і може одночасно звертатися до однієї і тієї ж пам'яті, що дозволяє впроваджувати паралельні обчислення на настільних комп'ютерах. Як наслідок, розпаралелювання послідовних програм стало найважливішим завданням програмування.

Операційна система дозволяє паралельне виконання декількох завдань і користувацьких програм на доступних ядрах. Проте, для того, щоб послідовна програма повною мірою використовувала переваги багатоядерної архітектури, програміст повинен реструктурувати та розпаралелити код. Для покращення часу виконання прикладного програмного забезпечення масштабування частоти більше не буде достатнім; натомість програмісти повинні розпаралелювати свій програмний код, щоб використовувати зростаючу обчислювальну потужність багатоядерних архітектур [9].

Як послідовні, так і паралельні комп'ютери працюють на основі потоку інструкцій, відомих як алгоритми. Цей набір інструкцій інструктує комп’ютер про те, що він має робити на кожному кроці. Залежно від потоку інструкцій і потоку даних комп’ютери можна класифікувати на чотири категорії.

Комп'ютери SISD складаються з одного блоку керування, одного обчислювального блоку і одного блоку пам'яті. У цьому типі архітектури процесор отримує єдиний потік інструкцій від блоку керування і працює з єдиним потоком даних з блоку пам'яті [10]. Під час обробки процесор отримує одну інструкцію від блоку керування і працює з одним елементом даних, отриманим з блоку пам'яті.

Рис. 1.1 - Архітектура SISD

Комп'ютери SIMD мають один блок керування, кілька обчислювальних блоків і спільну пам'ять або мережу з'єднання. Єдиний блок керування передає інструкції всім обчислювальним процесорам у комп'ютерах SIMD. На кожному кроці обчислень процесори отримують один набір інструкцій від блоку керування і обробляють окремі набори даних з блоку пам'яті. Кожен процесор має свій блок локальної пам'яті для зберігання даних та інструкцій. Для того, щоб процесори могли спілкуватися між собою, вони використовують спільну пам'ять або мережу зв'язку. Поки одні процесори виконують інструкції, інші чекають на наступний набір інструкцій. Інструкції блоку керування визначають, який процесор буде активним (виконувати інструкції), а який - неактивним (чекати на наступний набір інструкцій) [10.11].

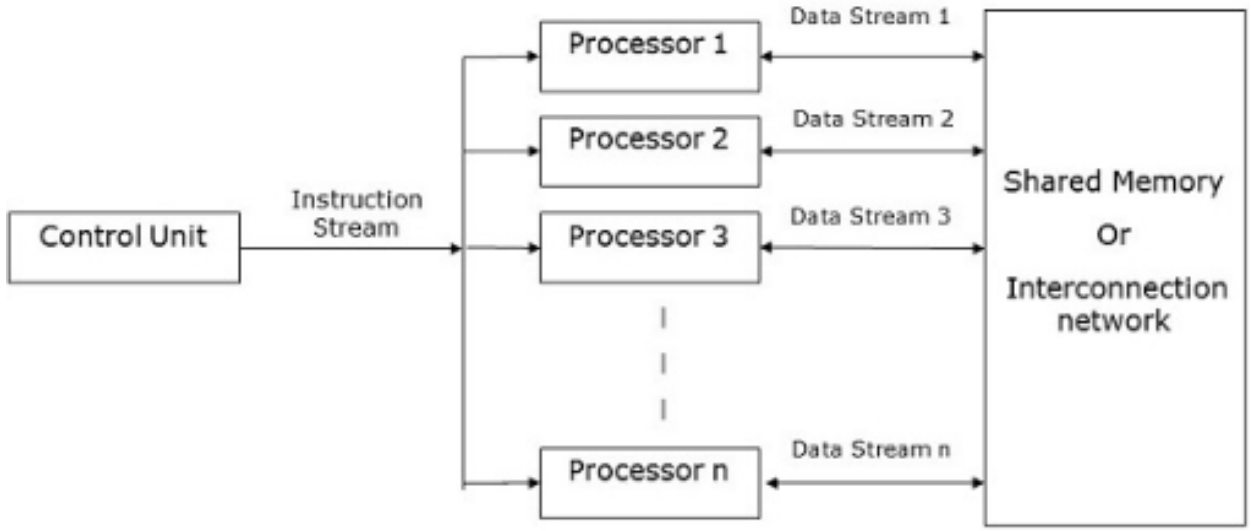


Рис. 1.2 - Архітектура SIMD

Комп'ютери MISD мають кілька блоків керування, процесорів та один блок пам'яті. Кожен процесор має свій власний блок керування, і вони разом використовують загальний блок пам'яті. Кожен процесор отримує інструкції індивідуально від свого блоку керування і працює з окремим потоком даних відповідно до вказівок своїх блоків управління. Всі процесори працюють одночасно.

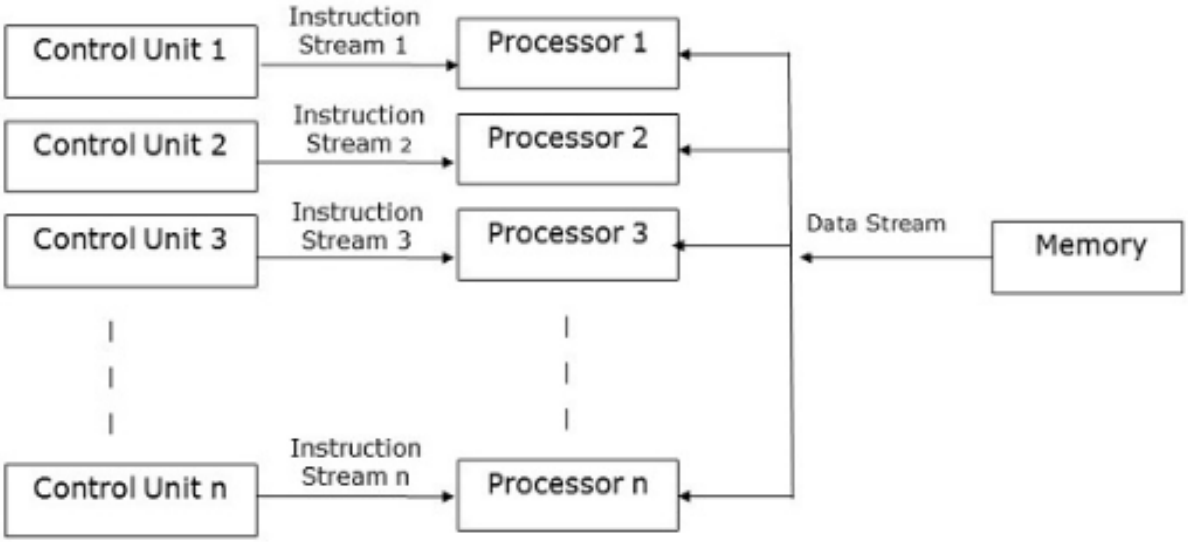


Рис. 1.3 - Архітектура MISD

Комп'ютери MIMD мають кілька блоків керування та обробки, а також спільну пам'ять або мережу зв'язку. Кожен процесор має власний блок керування та локальної пам'яті, а також арифметико-логічний блок. Вони отримують різні набори інструкцій від відповідних блоків керування і працюють з різними наборами даних.

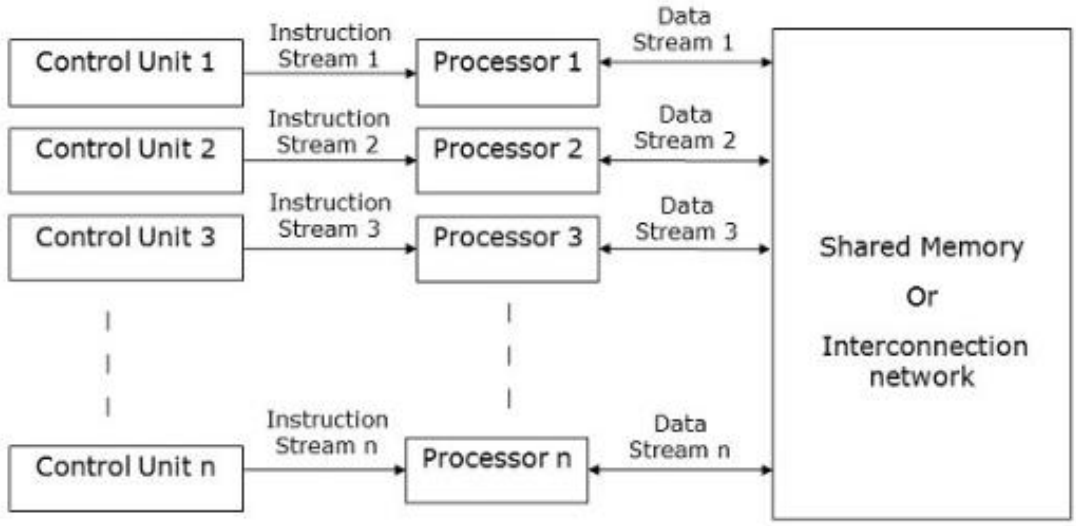


Рис. 1.4 - Архітектура MIMD

Загалом існує чотири типи паралельних обчислень:

1. Паралелізм на рівні бітів - форма паралельних обчислень, в якій кожна задача залежить від розміру комп’ютерного слова — кількості інформації, якою

процесор може маніпулювати за один цикл. З точки зору виконання завдання над великими обсягами даних, це зменшує кількість інструкцій, які повинен виконати процесор. Виникає необхідність розбити операцію на серії інструкцій. Наприклад, є 8-розрядний процесор, а ви хочете виконати операцію над 16-розрядними числами. Спочатку він повинен оперувати 8 молодшими бітами, а потім 8 старшими бітами. Отже, для виконання операції потрібно дві інструкції. Операція може бути виконана однією інструкцією, тобто 16-розрядним процесором [11].

2. Паралелізм на рівні інструкцій - в одному такті процесора процесор вирішує, скільки інструкцій виконується одночасно на рівні інструкцій. Для кожної фази тактового циклу процесор в паралелізмі на рівні інструкцій може мати можливість адресувати менше, ніж одну інструкцію. Програмний підхід до паралелізму на рівні інструкцій функціонує на основі статичного паралелізму, коли комп'ютер вирішує, які інструкції виконувати одночасно.

3. Паралелізм даних - це розпаралелювання на декількох процесорах у паралельних обчислювальних середовищах. Він зосереджується на розподілі даних між різними вузлами, які працюють з даними паралельно. Його можна застосовувати до звичайних структур даних, таких як масиви і матриці, працюючи над кожним елементом паралельно. Він протиставляється паралелізму завдань як іншій формі паралелізму. Завдання з паралельної обробки масиву з n елементів можна розділити порівну між усіма процесорами. Припустимо, що ми хочемо підсумувати всі елементи заданого масиву і час однієї операції додавання становить T одиниць часу. У випадку послідовного виконання, час, який займає процес, буде n × T одиниць часу, оскільки він підсумовує всі елементи масиву. З іншого боку, якщо ми виконаємо цю роботу як паралельну роботу з даними на 4 процесорах, час, що витрачається, зменшиться до (n/4) × T + об'єднання накладних одиниць часу. Паралельне виконання дає прискорення в 4 рази порівняно з послідовним виконанням. Важливо зазначити, що локальність посилань на дані відіграє важливу роль в оцінці продуктивності моделі паралельного програмування даних. Локальність

даних залежить від кількості звернень до пам'яті, що виконуються програмою, а також від розміру кешу [12].

4. Паралелізм завдань, також відомий як паралелізм функцій і паралелізм керування, - це техніка розпаралелювання комп'ютерного коду на багатьох процесорах у середовищах для паралельних обчислень. Його мета - розподілити завдання, які виконуються одночасно потоками або процесами, між різними процесорами. На відміну від паралелізму даних, при якому одна й та сама задача виконується на різних компонентах даних, паралелізм завдань передбачає одночасне виконання багатьох різних завдань на одних і тих самих даних. Конвеєризація є поширеною формою паралелізму завдань, що складається з послідовного переміщення одного набору даних через серію окремих завдань, де кожне завдання може виконуватися незалежно від інших [13].

Архітектура паралельного комп’ютера існує в різноманітних комп’ютерах, класифікованих відповідно до рівня, на якому апаратне забезпечення підтримує паралелізм. Архітектура паралельного комп’ютера та методи програмування працюють разом, щоб ефективно використовувати ці машини. Існують різні класи паралельних комп’ютерних архітектур, а саме [14]:

1. Багатоядерні обчислення - інтегральна схема комп'ютерного процесора, що містить два або більше окремих обчислювальних ядер, відома як багатоядерний процесор, який має можливість виконувати програмні інструкції одночасно. Ядра можуть реалізовувати такі архітектури, як VLIW, суперскалярна, багатопотокова або векторна, і можуть бути інтегровані на одному кристалі інтегральної схеми або на декількох кристалах в одному корпусі мікросхеми. Багатоядерні архітектури класифікуються як гетерогенні, тобто такі, що складаються з ядер, які не є ідентичними, або як гомогенні, тобто такі, що складаються лише з ідентичних ядер.

2. Симетрична багатопроцесорність - багатопроцесорна комп’ютерна апаратна та програмна архітектура, в якій два або більше незалежних, однорідних процесорів контролюються одним екземпляром операційної системи, яка однаково обробляє всі процесори та підключена до єдиної спільної

основної пам’яті з повним доступом до всіх загальних ресурсів та пристроїв. Кожен процесор має приватну кеш-пам’ять, може бути під’єднаний за допомогою сітчастих мереж на кристалі та може працювати над будь-яким завданням, незалежно від того, де в пам’яті знаходяться дані для цього завдання.

3. Розподілені обчислення - на різних комп'ютерах, об'єднаних в мережу, розміщуються компоненти розподіленої системи. Ці мережеві комп'ютери координують свої дії за допомогою спілкування через HTTP, RPC-подібні черги повідомлень та з'єднувачі. Паралельність роботи компонентів і незалежна відмова компонентів є характеристиками розподілених систем. Зазвичай розподілене програмування класифікують у вигляді однорангових, клієнт-серверних, трирівневих або n-рівневих архітектур. Іноді терміни "паралельні обчислення" і "розподілені обчислення" використовуються як взаємозамінні, оскільки вони багато в чому перетинаються.

4. Масово паралельні обчислення - це стосується використання багатьох комп’ютерів або комп’ютерних процесорів для одночасного виконання набору паралельних обчислень. Один підхід передбачає групування кількох процесорів у тісно структурований централізований комп’ютерний кластер. Іншим підходом є грід-обчислення, у якому багато широко розповсюджених комп’ютерів працюють разом і спілкуються через Інтернет для вирішення певної проблеми [13-15].

1.2.2 Переваги та обмеження паралельних обчислень

Переваги паралельних обчислень [14]:

- Прискорення: паралельні обчислення можуть значно скоротити час, необхідний для вирішення складної проблеми, розбиваючи її на більш дрібні частини, які можна вирішити одночасно.
- Масштабованість: архітектури паралельних обчислень можна легко масштабувати для обробки більших наборів даних або складніших обчислень.

- Відмовостійкість: паралельні обчислення можуть бути відмовостійкими, що означає, що якщо один процесор або вузол виходить з ладу, інші можуть продовжувати працювати.
- Рентабельність: завдяки використанню кількох процесорів або вузлів паралельні обчислення можуть бути економічно ефективнішими, ніж використання одного високопродуктивного процесора.
- Підвищена продуктивність: паралельні обчислення можуть підвищити продуктивність певних типів обчислень, як моделювання, аналіз даних і машинне навчання. Крім того, паралельні обчислення підходять для апаратного забезпечення, оскільки при послідовних обчисленнях потенційна обчислювальна потужність витрачається даремно.

Обмеження паралельних обчислень [16]:

- Складність: архітектури паралельних обчислень можуть бути складними та важкими для програмування, що може вимагати спеціальних навичок і знань.
- Витрати на зв’язок: зв’язок між процесорами або вузлами може бути повільним, що може обмежити підвищення продуктивності паралельних обчислень.
- Проблеми синхронізації: коли кілька процесорів або вузлів працюють разом, можуть виникнути проблеми синхронізації, що може призвести до помилок або зниження продуктивності.
- Обмежена масштабованість: деякі архітектури паралельних обчислень можуть мати обмежену масштабованість, що може обмежити їхню корисність для певних типів обчислень.
- Обмеження апаратного забезпечення: для архітектури паралельних обчислень може знадобитися спеціальне обладнання, яке може бути дорогим і важкодоступним.

1.3 Штучні нейронні мережі

1.3.1 Основи нейромереж

Штучні нейронні мережі містять штучні нейрони, які називаються одиницями. Ці одиниці розташовані в серії шарів, які разом складають всю штучну нейронну мережу в системі. Шар може мати лише десяток одиниць або мільйони одиниць, оскільки це залежить від того, як складна нейронна мережа повинна вивчати приховані закономірності в наборі даних. Зазвичай штучна нейронна мережа має вхідний шар, вихідний шар, а також приховані шари. Вхідний шар отримує дані із зовнішнього світу, які нейронна мережа повинна проаналізувати або вивчити. Потім ці дані проходять через один або кілька прихованих шарів, які перетворюють вхідні дані на дані, що є цінними для вихідного шару. Нарешті, вихідний шар забезпечує вихід у вигляді реакції штучної нейронної мережі на надані вхідні дані.

У більшості нейронних мереж одиниці з'єднані між собою від одного шару до іншого. Кожен з цих зв'язків має вагу, яка визначає вплив одного елемента на інший. Коли дані передаються від одного елемента до іншого, нейронна мережа дізнається все більше і більше про дані, що врешті-решт призводить до виходу з вихідного шару.

Нейронні мережі натхненні структурою мозку, але не обов'язково є його точною моделлю. Ми ще багато чого не знаємо про мозок і про те, як він працює, але він слугує джерелом натхнення в багатьох наукових галузях завдяки своїй здатності розвивати інтелект. І хоча існують нейронні мережі, створені з єдиною метою - зрозуміти, як працює мозок, глибинне навчання, яким ми його знаємо сьогодні, не має на меті повторити роботу мозку. Натомість, глибинне навчання зосереджується на створенні систем, які вивчають багаторівневу композицію шаблонів [17].

На початку 1940-х років Воррен Маккалох, нейрофізіолог, об'єднався з логіком Волтером Піттсом, щоб створити модель роботи мозку. Це була проста лінійна модель, яка давала позитивний або негативний результат залежно від набору вхідних сигналів і ваг. Тобто кожен вхідний сигнал (x_1, x_2, ..., x_n) має свою

вагу (w_1, w_2, ..., w_n), що відображає його важливість для роботи нейрону. Нейрон обчислює зважену суму вхідних сигналів і їх ваг. Якщо значення суми більше або рівне порогу, нейрон видає вихідний сигнал 1, а в іншому випадку, вихід буде 0. Ця модель обчислень була навмисно названа нейронною, тому що вона намагалася імітувати роботу основного блоку мозку. Подібно до того, як нейрони мозку отримують електричні сигнали, нейрон Маккалоха і Піттса отримував вхідні сигнали і, якщо вони були достатньо сильними, передавав їх іншим нейронам. Перше застосування нейрона відтворювало логічний вентиль, де у вас є один або два бінарних входи і булева функція, яка активується лише при правильних входах і вагах [18].

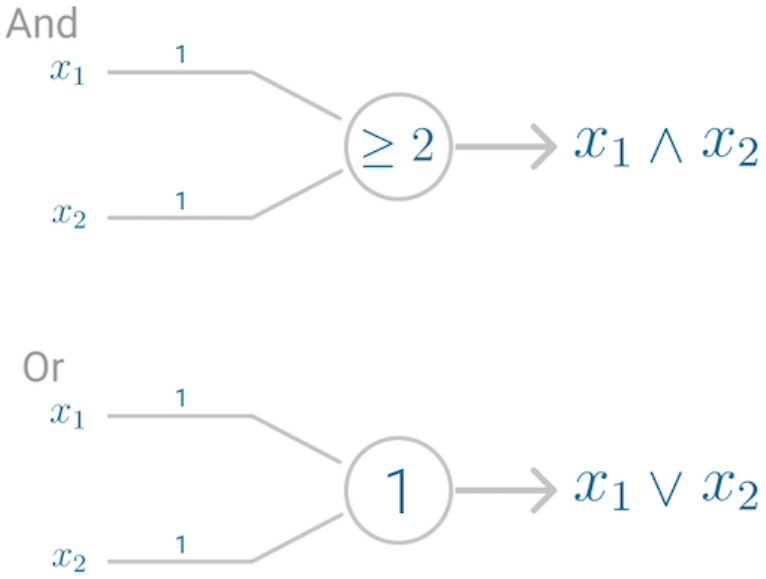


Рис. 1.5 - Приклад логічних елементів І та АБО

Однак у цієї моделі була проблема. Вона не могла навчатися, як мозок. Єдиний спосіб отримати бажаний результат - це заздалегідь встановити ваги, які працюють як каталізатори в моделі. Лише через деякий час Френк Розенблат розширив цю модель і створив алгоритм, який міг вивчати ваги, щоб генерувати вихідні дані. Спираючись на нейрон Маккалоха та Піттса, Розенблат розробив персептрон.

Персептрон Розенблата спирався на базову одиницю обчислень - нейрон. Як і в попередніх моделях, кожен нейрон має комірку, яка отримує серію пар

входів і ваг. Основна відмінність моделі Розенблата полягає в тому, що входи об'єднуються у зважену суму, і якщо зважена сума перевищує заздалегідь визначений поріг, нейрон спрацьовує і видає вихідний сигнал.

Математична модель нейрону взагалі може бути представлена як відображення вхідних сигналів у вихідний сигнал за допомогою функції активації. Функція активації призначена для моделювання активності нейрону. Вона визначає чи буде нейрон активований [19]. Для цього обчислюється зважена сума та додатково додається зміщення, щоб отримати результат. Загальна формула для вихідного сигналу y може бути записана як:

$$y = f\left(\sum_{i=1}^{n} x_i\, w_i + \ b\right), \tag{1.5}$$

де f – функція активації, x_i – вхідні сигнали, w_i – ваги, b - зсув. Значення y може бути від 0 до 1 або від -1 до 1, залежно від вибору функції активації. Це значення y є вихідним сигналом нейрону і може використовуватися як вхід для наступних шарів в нейромережі або як вихід самого нейрону, якщо це останній шар.

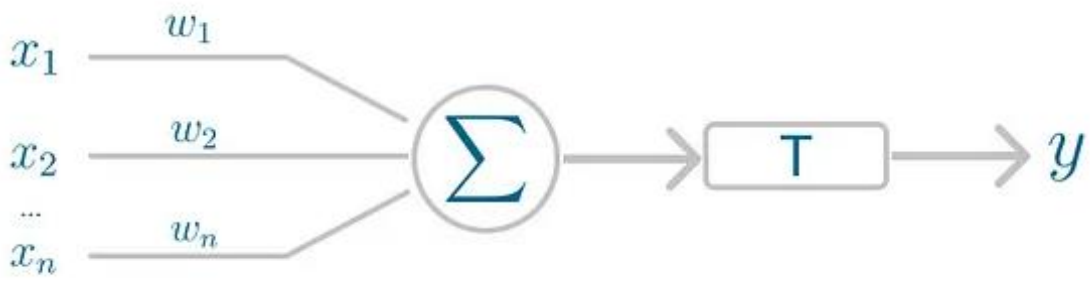


Рис. 1.6 - Одношаровий персептрон

Поріг T являє собою функцію активації. Ступінчаста (порогова) функція активації, також відома як функція Гевісайда, є простою лінійною функцією, яка визначає вихід нейрону на основі порогового значення. Тобто, якщо вхідний сигнал перевищує порогове значення, то функція активації повертає значення 1, а якщо вхідний сигнал менше порогового значення, то функція активації повертає значення 0.

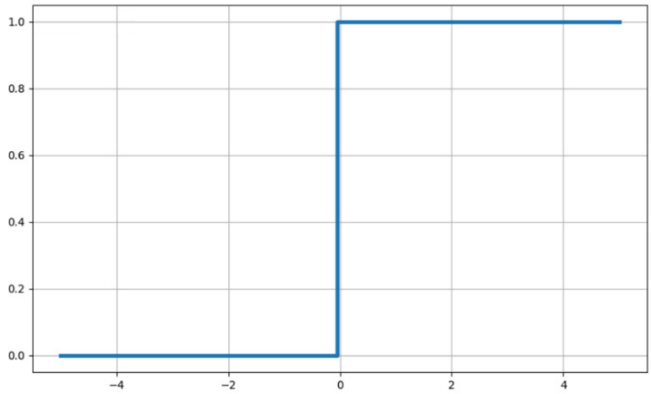


Рис. 1.7 - Ступінчаста (порогова) функція активації

З таким дискретним виходом, керованим функцією активації, персептрон можна використовувати як модель бінарної класифікації, визначаючи лінійну межу рішення. Він знаходить розділову гіперплощину, яка мінімізує відстань між неправильно класифікованими точками та межею рішення.

Функція втрат персептрона:

$$D(w, c) = -\sum_{i \in M} y_i (x_i w_i + c)\,, \tag{1.6}$$

де D – відстань, y – вихід, M - неправильно класифіковані спостереження.

Щоб мінімізувати цю відстань, персептрон використовує функцію оптимізації стохастичний градієнтний спуск. Якщо дані є лінійно відокремлюваними, гарантується, що стохастичний градієнтний спуск зійдеться за скінченну кількість кроків. Останній елемент, який потрібен персептрону, - це функція активації, функція, яка визначає, чи буде нейрон працювати чи ні. Початкові моделі персептрона використовували сигмоїдну функцію. Вона зіставляє будь-який реальний вхідний сигнал зі значенням 0 або 1 і кодує нелінійну функцію. Нейрон може отримувати на вхід від'ємні числа, і він все одно буде здатний видавати на виході або 0, або 1 [20]. Формула сигмоїдної функції:

$$F(x) = \frac{1}{1+exp(-1)} \tag{1.7}$$

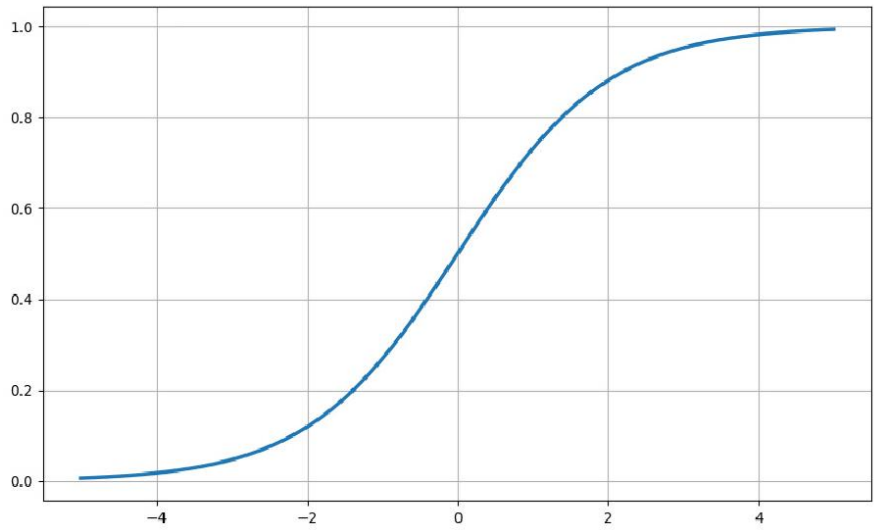


Рис. 1.8 – Сигмоїдна функція активації

Алгоритми останнього десятиліття здебільшого використовують Rectified Linear Unit (ReLU) як функцію активації нейрона. Формула ReLU:

$$F(x) = max(0, x) \tag{1.8}$$

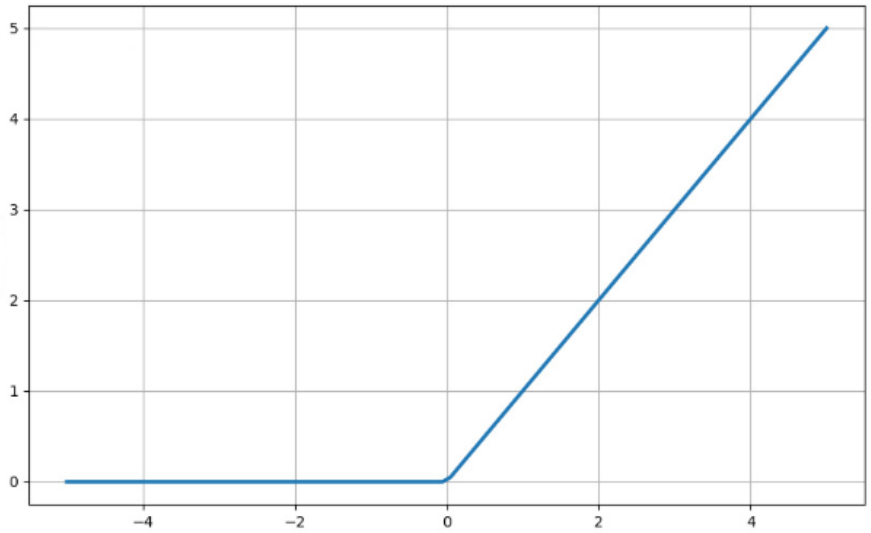


Рис. 1.9 - Функція активації ReLU

Причина, чому ReLU стала більш прийнятною, полягає в тому, що вона дозволяє краще оптимізувати за допомогою стохастичного градієнтного спуску, більш ефективно обчислювати і є масштабовано-інваріантною, тобто її характеристики не залежать від масштабу вхідних даних. Нейрон отримує вхідні

дані і випадковим чином вибирає початковий набір ваг. Вони об'єднуються у зважену суму, а потім ReLU, функція активації, визначає значення виходу.

Багатошаровий персептрон має вхідні та вихідні шари, а також один або кілька прихованих шарів з великою кількістю нейронів, об'єднаних разом. І хоча в персептроні нейрон повинен мати функцію активації, яка встановлює поріг, наприклад, ReLU або сигмоїд, нейрони в багатошаровому персептроні можуть використовувати будь-яку довільну функцію активації [21].

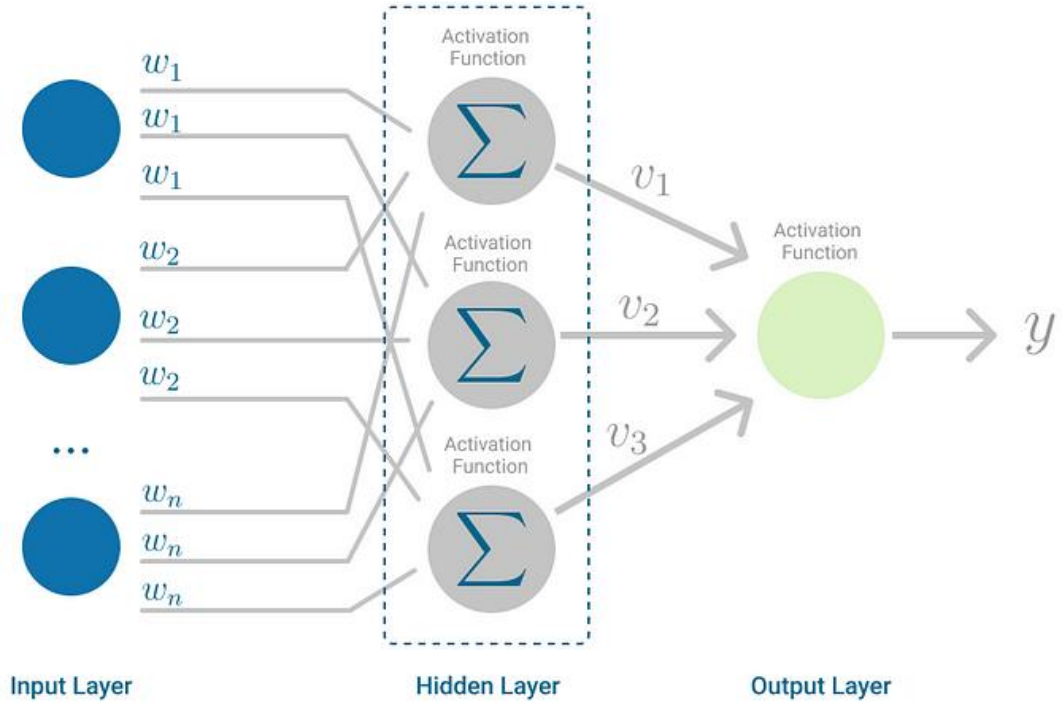


Рис. 1.10 - Багатошаровий персептрон

Багатошаровий персептрон підпадає під категорію алгоритмів прямого поширення, тому що входи об'єднуються з початковими вагами у зважену суму і піддаються функції активації, так само як і в персептроні. Але різниця в тому, що кожна лінійна комбінація поширюється на наступний шар. Кожен шар передає наступному результат своїх обчислень, своє внутрішнє представлення даних. Так відбувається весь шлях через приховані шари до вихідного шару. Якби алгоритм обчислював тільки зважені суми в кожному нейроні, передавав результати у вихідний шар і зупинявся на цьому, він не зміг би дізнатися ваги, які мінімізують функцію вартості. І якщо б алгоритм обчислював лише одну

ітерацію, фактичного навчання не було б. Тому тут потрібно використовувати зворотне поширення помилки [19,22].

Зворотне поширення - це алгоритм навчання, який дозволяє багатошаровому персептрону ітеративно коригувати ваги в мережі з метою мінімізації помилки. Існує одна жорстка вимога для правильної роботи зворотнього поширення. Функція, яка поєднує входи та ваги нейрона, наприклад, зважена сума, та функція активації, наприклад, ReLU, повинні бути диференційованими. Ці функції повинні мати обмежену похідну, оскільки градієнтний спуск зазвичай є функцією оптимізації, що використовується в багатошаровому персептроні.

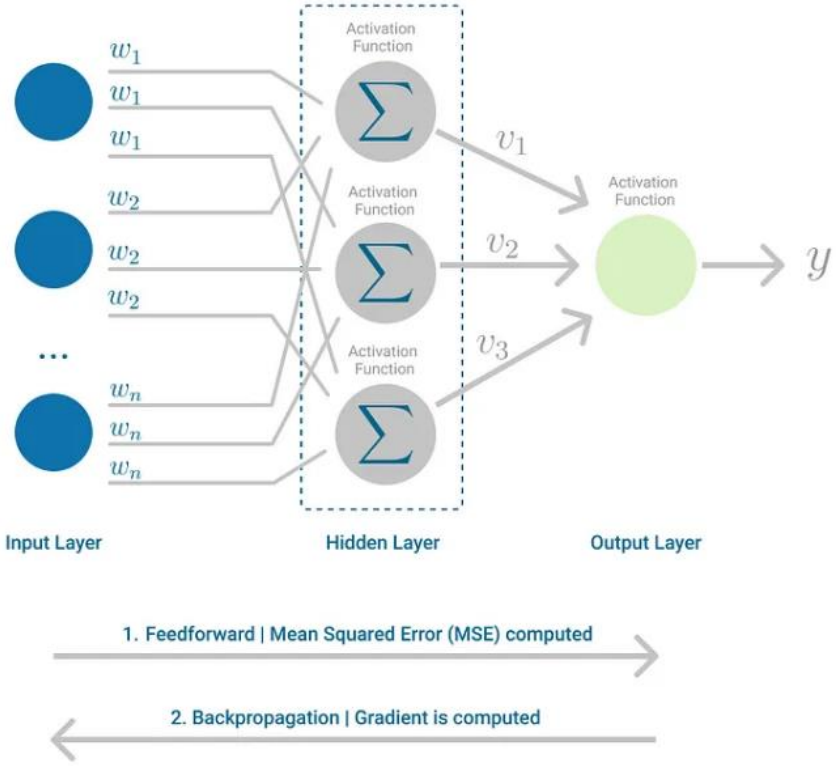


Рис. 1.11 - Багатошаровий персептрон із виділенням етапів прямого поширення та зворотного поширення

На кожній ітерації, після того, як зважені суми пересилаються через усі шари, обчислюється градієнт середньоквадратичної помилки для всіх вхідних і вихідних пар. Потім, щоб поширити його назад, ваги першого прихованого шару

оновлюються значенням градієнта. Таким чином ваги повертаються до початкової точки нейронної мережі.

$$\Delta_w(t) = -\varepsilon \frac{dE}{dw_{(t)}} + \alpha \Delta_{w(t-1)}\,, \tag{1.9}$$

де $\Delta_w(t)$ - градієнт поточної ітерації, ε - зсув, $dw_{(t)}$ - вектор ваги, α - швидкість навчання, $\Delta_{w(t-1)}$ - градієнт попередньої ітерації, dE – помилка.

Цей процес триває до тих пір, поки градієнт для кожної пари вхід-вихід не зійдеться, тобто новий обчислений градієнт не зміниться більше, ніж на заданий поріг збіжності, порівняно з попередньою ітерацією.

1.3.2 Основні типи нейронних мереж та доцільність їх застосування

Нейронна мережа прямого поширення - це тип штучної нейронної мережі, яка працює з інформацією, що тече тільки в одному напрямку, від вхідних вузлів, через приховані вузли, до вихідних вузлів. У конструкції цієї мережі відсутні цикли або петлі. Нейронні мережі прямого поширення були початковою формою винайдених штучних нейронних мереж, які є простішими порівняно з рекурентними нейронними мережами.

Архітектура нейронної мережі прямого поширення складається з трьох типів шарів: вхідний шар, приховані шари і вихідний шар. Кожен шар складається з одиниць, відомих як нейрони, які пов'язані між собою вагами. Вхідний шар ініціює обчислення мережі, приймаючи вхідні значення і передаючи їх на обробку прихованим шарам. Вхідний шар складається з нейронів, які отримують вхідні дані і передають їх наступному шару. Кількість нейронів у вхідному шарі визначається розмірами вхідних даних. Приховані шари, які можна вважати обчислювальним двигуном нейронної мережі, не видно ні на вході, ні на виході. Нейрони кожного прихованого шару обчислюють зважену суму виходів попереднього шару, застосовують функцію активації і передають результат наступному шару. Нейронна мережа може містити різну кількість прихованих шарів, від жодного до декількох. Останній шар, відомий як

вихідний, генерує вихід мережі у відповідь на вхідні дані [23]. Кількість нейронів у вихідному шарі змінюється відповідно до кількості потенційних виходів, які мережа налаштована генерувати. У повністю пов'язаній мережі кожен нейрон в одному шарі пов'язаний з кожним нейроном в наступному шарі. Сила зв'язку між нейронами представлена вагами, а навчання в нейронній мережі полягає в оновленні цих ваг на основі помилки на виході.

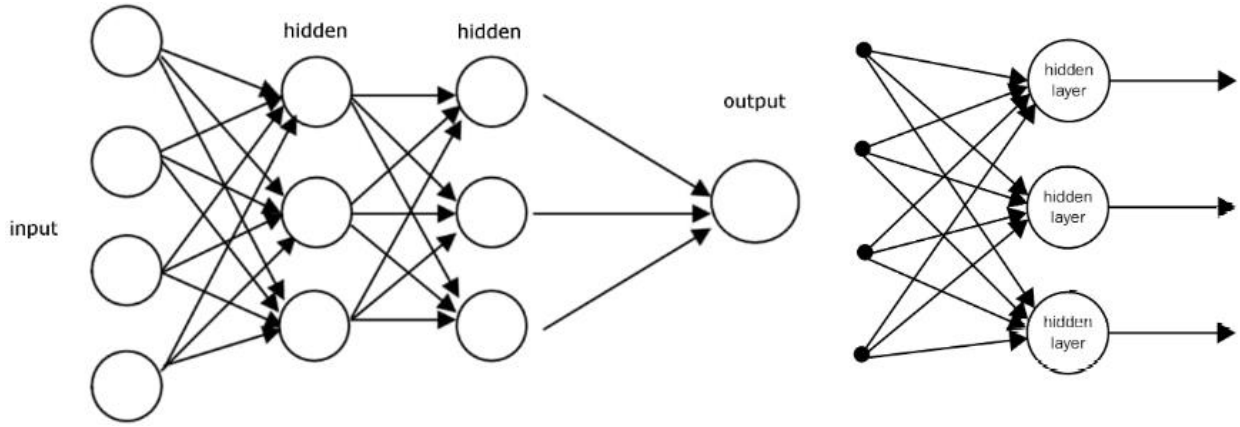


Рис. 1.12 – Архітектура нейронної мережі прямого поширення

Функція нейронної мережі прямого поширення складається з двох фаз: фази прямого поширення і фази зворотного поширення. Вхідні дані подаються в мережу у фазі прямого поширення і поширюються через мережу вперед. На кожному прихованому шарі обчислюється сума зважених вхідних даних і пропускається через активаційну функцію, яка вносить нелінійність у модель. Цей процес триває до тих пір, поки не буде досягнутий вихідний шар, і не буде згенеровано прогноз. На етапі зворотного поширення обчислюється похибка, яка є різницею між прогнозованим виходом і фактичним виходом, а потім поширюється назад через мережу для коригування ваг з метою мінімізації цієї похибки. Коригування ваг зазвичай здійснюється за допомогою алгоритму оптимізації градієнтного спуску. Нейронні мережі прямого поширення володіють значними можливостями. Однак вони демонструють свій власний набір проблем і обмежень. Вибір кількості прихованих шарів і нейронів у кожному шарі є однією з основних проблем, яка може суттєво вплинути на продуктивність мережі. Ще однією поширеною проблемою є перенавчання, коли

мережа отримує повне уявлення про навчальні дані, які містять шум, і, отже, погано працює, коли їй пред'являють невидимі дані [24].

Рекурентні нейронні мережі (RNN) - це категорія штучних нейронних мереж, які моделюють зв'язки між вузлами у вигляді орієнтованого графа в часовій послідовності, що відображає динамічну поведінку в часі. На відміну від нейронних мереж прямого поширення, RNN використовують свою внутрішню пам'ять (стан) для обробки вхідних послідовностей. Рекурентні нейронні мережі є дуже цінними в задачах, які вимагають врахування контексту або послідовного характеру даних, включаючи прогнозування часових рядів, обробку природної мови, розпізнавання мови і навіть підписи до зображень.

Фундаментальною концепцією RNN є поняття персистентності - інформація циркулює в мережі, що дозволяє приймати рішення під впливом не лише поточних вхідних даних, але й послідовності попередніх вхідних даних. Наприклад, при передбаченні наступного слова в реченні може бути корисним розпізнавання попередніх слів. Рекурентні нейронні мережі досягають цієї безперервності шляхом включення циклів у саму мережу. Під час обробки вхідних даних частина вихідних даних зберігається у внутрішньому стані мережі і згодом використовується як додатковий контекст для обробки майбутніх вхідних даних. Цей процес триває, поки RNN обробляє кожен елемент вхідної послідовності. Це дозволяє мережі побудувати представлення всієї послідовності у своїй пам'яті [25].

Архітектура найпростішої RNN включає в себе один шар, що зациклюється. Цей цикл представляє часовий аспект мережі, так що на кожному часовому кроці шар отримує вхід не тільки від попереднього шару, але й від власного виходу з попереднього часового кроку. Цей циклічний зв'язок надає мережі функцію пам'яті, яка дозволяє їй зберігати інформацію між кроками обробки.

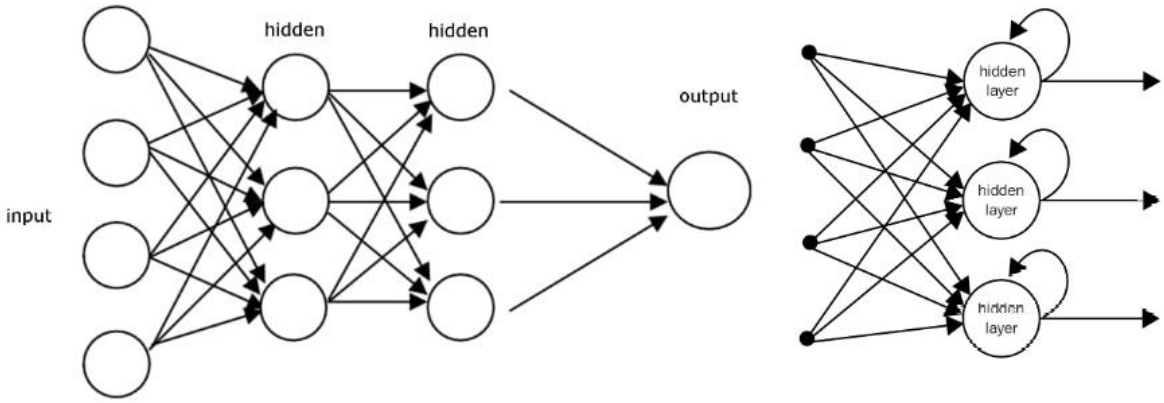


Рис. 1.13 – Архітектура рекурентної нейронної мережі

RNN часто будуються з великою кількістю шарів, а також існують більш складні моделі, такі як мережі з довгою короткочасною пам'яттю (LSTM) та рекурентні блоки з закритими шарами (GRU), які призначені для більш точного відображення довготривалих залежностей та вирішення таких проблем, як проблема зникаючого градієнта.

Навчання RNN схоже на навчання будь-якої нейронної мережі, але з додаванням часового виміру. Найпоширеніший алгоритм навчання для RNN - це поширення в часі (Backpropagation Through Time) [26]. BPTT створює копію RNN на кожному часовому кроці, ефективно розгортаючи її в часі, а потім застосовує стандартний алгоритм зворотного поширення для навчання мережі. Однак BPTT може бути дорогим в обчислювальному плані і може страждати від зникаючих або вибухових градієнтів, особливо коли має справу з довгими послідовностями.

Рекурентні нейронні мережі мають значну потужність, але також несуть в собі певні перешкоди. Однією з них є "проблема зникаючого градієнта", коли градієнти можуть стати занадто незначними під час навчання, що в кінцевому підсумку перешкоджає здатності мережі вивчати довгострокові зв'язки. І навпаки, "проблема вибухоподібного градієнта" призводить до експоненціального зростання градієнтів, що призводить до нестабільної поведінки під час навчання. Крім того, RNN можуть вимагати більше обчислювальних ресурсів, ніж мережі прямого поширення, через свою

послідовну природу і, як наслідок, повільніше навчаються. Часові залежності, присутні в RNN, створюють проблеми з розпаралелюванням їх навчання на декількох графічних процесорах або машинах.

Дослідники розробили різноманітні методи для подолання труднощів, пов'язаних з RNN. Мережі LSTM та GRU, про які ми вже згадували, спеціально розроблені для того, щоб краще фіксувати довгострокові залежності та вирішувати проблему зникаючих градієнтів. Також було впроваджено механізми уваги та трансформаторні моделі, які дозволяють мережі спрямовувати свою увагу на різні частини вхідної послідовності. Цей підхід продемонстрував чудові результати в таких завданнях, як переклад мови та узагальнення тексту.

Визначення доцільності застосування конкретного типу нейронної мережі включає в себе розгляд різних аспектів, які визначають, наскільки ефективною буде дана архітектура для вирішення конкретної задачі. Ось деякі ключові аспекти, які варто враховувати [24, 27]:

- Параметри навчальних даних: описують характеристики ваших даних, такі як обсяг, розподіл класів, наявність аномалій. Важливо мати достатньо різноманітних та представницьких даних для ефективного навчання.
- Загальні обмеження процесу навчання: включають час навчання, доступність ресурсів, обсяг доступних обчислювальних потужностей та інші обмеження, які можуть вплинути на тривалість та ефективність навчання.
- Вимоги до обчислювальних потужностей: залежать від архітектури мережі та обсягу даних. Глибокі мережі та великі обсяги даних можуть вимагати значних обчислювальних ресурсів, іноді використання графічних процесорів чи спеціалізованих пристроїв для прискорення навчання може бути доцільним.
- Вимоги до вихідної інформації: специфікації та формати очікуваних результатів від мережі, які слід враховувати під час проектування та навчання.
- Обмеження технічної реалізації нейронної мережі: технічні обмеження, такі як обсяг пам'яті, швидкість обчислень, можливість паралельності та інші фактори, що впливають на ефективність та швидкість роботи мережі.

- Сфера застосування: визначення конкретних областей, в яких планується використовувати нейронну мережу. Наприклад, для обробки зображень, обробки мовленнєвих даних, машинного перекладу тощо.

1.3.3 Застосування нейронних мереж у великих обчислювальних задачах

Нейроні мережі широко використовуються у великих обчислювальних задачах завдяки їхній здатності автоматично визначати складні залежності у великих наборах даних. Ось кілька областей, де вони знаходять широке застосування:

- Машинне навчання та глибоке навчання. Тобто класифікація розпізнавання образів, де нейронні мережі використовуються для розв'язання завдань класифікації та розпізнавання образів, таких як розпізнавання зображень, голосу, тексту тощо. Також генерація контенту - використовуються для генерації зображень, тексту, музики та інших типів контенту.
- Обробка природної мови (NLP). Тобто машинний переклад, де нейронні мережі допомагають у вдосконаленні систем машинного перекладу, роблячи їх більш точними та придатними для різних мов. Також аналіз настрою та сентименту - використовуються для визначення та аналізу настрою текстових даних, таких як відгуки та коментарі у соціальних мережах або інших платформах, сервісах.
- Медичні дослідження - нейронні мережі використовуються для аналізу медичних зображень, рентгенів та інших даних для діагностики та прогнозування захворювань.
- Фінансовий аналіз – нейронні мережі допомагають аналізувати великі обсяги фінансових даних для прогнозування рухів ринку та прийняття рішень.
- Інтернет речей (IoT) - оптимізація мережевих процесів за допомогою нейронних мереж, забезпечуючи більш ефективний обмін даними та керування пристроями.

- Автономні транспортні системи - самокерування автомобілів та дронів: нейронні мережі використовуються для розв'язання складних завдань навігації та взаємодії з оточуючим середовищем.
- Квантові обчислення - нейронні мережі можуть бути використані для оптимізації завдань, пов'язаних із квантовим обчисленням, сприяючи покращенню ефективності обчислень.
- Енергоефективність - великі підприємства та міста використовують нейромережі для ефективного управління енергоспоживанням та оптимізації систем житла.

Застосування нейромереж у великих обчислювальних задачах розширюється широким спектром галузей і вносить значний внесок у вирішення складних проблем та удосконалення різноманітних аспектів сучасного життя. Ці технології надають можливість автоматизувати та поліпшити розв'язання завдань, які раніше були важкодоступними або вимагали великих зусиль. Вони постійно розвиваються, а їх потенціал в сучасному світі ще не повністю розкритий. Це відкриває шлях для автоматизації процесів, покращення рішень у різних сферах, таких як медицина, фінанси, транспорт, інтернет речей та інші.

1.4 Постановка задачі дослідження

Об'єкт дослідження даної кваліфікаційної роботи є обчислювальні процеси при реалізації методу Гауса з елементами штучного інтелекту.

Предметом дослідження даної кваліфікаційної роботи є комп’ютерні моделі паралельної реалізації методу Гауса з використанням елементів штучного інтелекту на основі обчислювальної системи з високою продуктивністю.

Мета даної кваліфікаційної роботи — підвищення ефективності процесу обчислень методу Гауса шляхом використання паралельного програмування та елементів штучного інтелекту.

Для досягнення мети потрібно виконати наступний перелік сформованих задач, таких як: реалізація ефективного паралельного виконання методу Гауса для розв'язання лінійних систем рівнянь великих розмірностей; інтеграція

нейромереж у процес розв'язання лінійних систем методом Гауса; аналіз впливу параметрів нейромережі на швидкість розв'язання; проведення експериментів для оцінки ефективності розробленої моделі; порівняння ефективності паралельної та послідовної реалізації методу Гауса; визначення оптимальних параметрів для паралельних обчислень та елементів штучного інтелекту; огляд та аналіз отриманих результатів.

Отже дана робота присвячена розробці комп’ютерної моделі паралельної реалізації методу Гауса з елементами штучного інтелекту, яка буде мати високу ефективність при розв’язані великих систем лінійних рівнянь.

Висновки до розділу 1

В розділі було розглянуто метод Гауса, який визначається як ефективний і класичний метод розв'язання систем лінійних рівнянь. Підкреслено його важливість у численних обчислювальних завданнях та велику потребу в оптимізації для вирішення великих систем. Досліджено його алгоритм, а точніше прямий та зворотній хід, та проведена оцінка ефективності методу.

Далі були розглянуті основні принципи, архітектури та методи паралельних обчислень, які є ключовим елементом для підвищення ефективності методу Гауса , зокрема, шляхом реалізації паралельного виконання методу для розв'язання великих систем лінійних рівнянь. Досліджено основи нейронних мереж, різноманітні типи мереж та доцільність їх застосування у великих обчислювальних задачах. Надано огляд архітектури та функціоналу кожного типу мережі, а також розглянуто основні виклики та обмеження, пов'язані з їхнім застосуванням. Також була визначена постановка задачі дослідження. Були описані предмет, об’єкт, мета та задачі, які потрібно розробити для досягнення мети дослідження.

РОЗДІЛ 2
РОЗРОБКА ТА РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ МОДЕЛІ

2.1 Алгоритм паралельної реалізації методу Гауса

2.1.1 Послідовна реалізація

Метод Гауса є широко відомим прямим алгоритмом розв’язання систем лінійних рівнянь, для яких матриці коефіцієнтів є щільними. Якщо система лінійних рівнянь є невиродженою, то метод Гауса гарантує знаходження розв’язку з похибкою, яка визначається точністю машинних обчислень. Основна ідея методу полягає в приведенні матриці A за допомогою еквівалентних перетворень до трикутного вигляду. Потім значення шуканих невідомих може бути отримано безпосередньо в явному вигляді.

Метод Гауса базується на здатності здійснювати еквівалентні перетворення лінійних рівнянь, які не змінюють розв'язку системи. До цих еквівалентних перетворень відносяться [2,3]:

- множення або ділення будь-якого з рівнянь на ненульове число;
- перестановка рівнянь системи;
- віднімання або додавання до рівняння будь-якого іншого рівняння системи.

Метод Гауса передбачає виконання двох послідовних етапів. Під час першого етапу, відомого як прямий хід, початкова система лінійних рівнянь приводиться, шляхом послідовного виключення невідомих, до верхнього трикутного вигляду:

$$Ux = c, \tag{2.1}$$

де матриця коефіцієнтів системи, яка отримується, має вигляд:

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & \cdots & u_{0,n-1} \\ 0 & u_{1,1} & \cdots & u_{1,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & u_{n-1,n-1} \end{pmatrix} \tag{2.2}$$

На етапі зворотного ходу визначаються значення невідомих. Цей етап розпочинається з останнього рівняння в перетвореній системі, обчислюється значення змінної x_{n-1}, а потім, використовуючи передостаннє рівняння, отримуємо значення змінної x_{n-2} і т.д.

Процедура прямого ходу полягає в послідовному виключенні невідомих у системі лінійних рівнянь, яка розв'язується. На ітерації k, $0 \le k < n - 1$, методу проводиться виключення невідомої x_i для всіх рівнянь з номерами i, більшими за k (тобто $k < i \le n - 1$). Для цього з цих рівнянь здійснюється віднімання рядка i, помноженого на константу (a_{ik}/a_{kk}), для того, щоб результуючий коефіцієнт при невідомій x_i в рядках виявився нульовим - всі необхідні обчислення можуть бути визначені за допомогою співвідношень [4]:

$$a'_{ij} = a_{ij} - \left(\frac{a_{ik}}{a_{kk}}\right) \cdot a_{kj} \tag{2.3}$$

$$b'_i = b_i - \left(\frac{a_{ik}}{a_{kk}}\right) \cdot b_k\,, \tag{2.4}$$

де $k = 1, 2, \ldots, n - 1$ - крок виключення невідомих, який співпадає з номером рівняння в системі; $i = k + 1, \ldots, n$ - номер рівняння, з якого виключається невідома; $j = k, \ldots, n$ – номер елемента в рівнянні.

Пояснимо виконання прямого ходу на прикладі системи лінійних рівнянь вигляду:

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ 2x_0 + 7x_1 + 5x_2 &= 18 \\ x_0 + 4x_1 + 6x_2 &= 26 \end{aligned}$$

На першій ітерації проводиться виключення невідомої x_0 з другого і третього рядка. Для цього із цих рядків потрібно відняти перший рядок, помножений відповідно на 2 і 1. Після цих перетворень система рівнянь приймає вигляд:

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 + \ \ x_2 &= 16 \\ x_1 + 4x_2 &= 25 \end{aligned}$$

В результаті залишається виконати останню ітерацію і виключити невідому x_1 з третього рівняння. Для цього необхідно відняти другий рядок, і в остаточній формі система має наступний вигляд:

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 + x_2 &= 16 \\ 3x_2 &= 9 \end{aligned}$$

На рис. 2.1 представлена загальна схема стану даних на k-й ітерації прямого ходу алгоритму Гауса. Всі коефіцієнти при невідомих, розташовані нижче за головну діагональ і ліворуч стовпця j, вже є нульовими. На k-й ітерації прямого ходу здійснюється обнуління коефіцієнтів стовпця j, розташованих нижче головної діагоналі, шляхом віднімання рядка i, помноженого на потрібну ненульову константу. Після проведення подібної ітерації матриця, що визначає систему лінійних рівнянь, стає приведеною до верхнього трикутного вигляду [5,6].

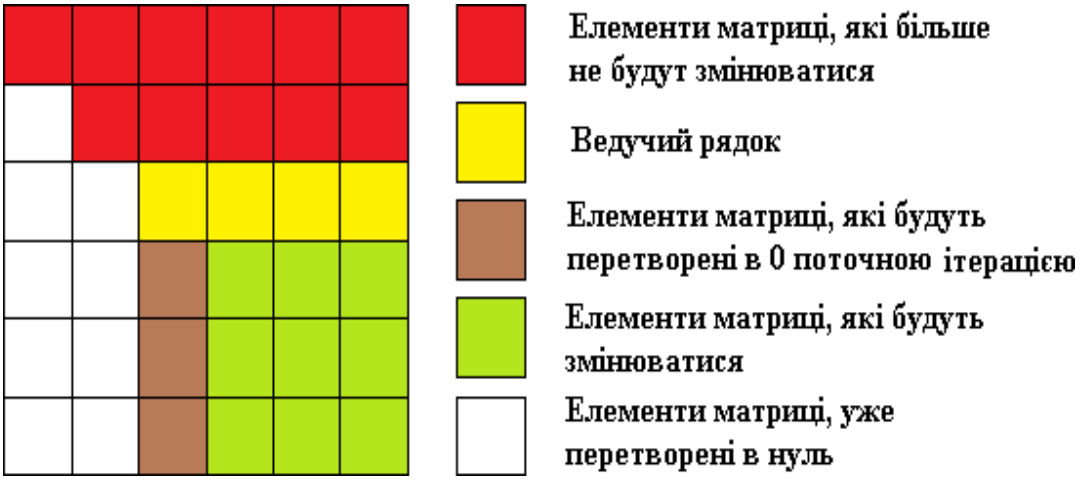


Рис. 2.1 – Ітерація прямого ходу алгоритму Гауса

Під час прямого ходу рядок, який використовується для виключення невідомих, отримує назву "ведучого", і діагональний елемент у цьому ведучому рядку називається ведучим елементом. Важливо відзначити, що обчислення можливі лише у випадку, якщо ведучий елемент не рівний нулю. Також важливо враховувати, що при невеликому значенні ведучого елемента $a_{i,j}$, операції

ділення і множення рядків на цей елемент можуть призводити до накопичення обчислювальної погрішності та зробити алгоритм менш стійким в обчисленнях .

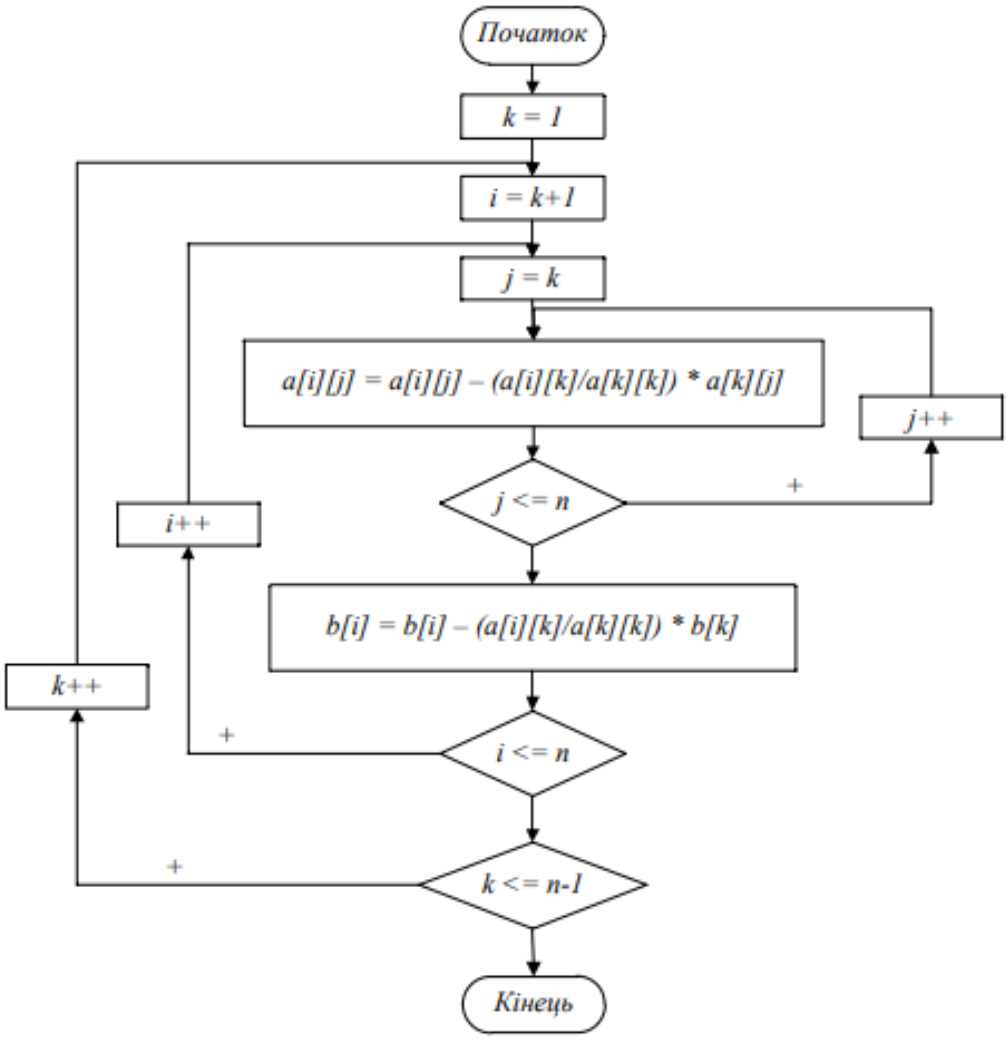


Рис. 2.2 – Блок-схема прямого ходу алгоритму Гауса

Можливий спосіб уникнути подібної проблеми може полягати в наступному: при виконанні кожної чергової ітерації прямого ходу слід визначити коефіцієнт з максимальним за модулем елемент в стовпці, в якому знаходиться невідома, що виключається, тобто

$$\underset{k \le i \le n}{y = max|a_{ik}|} \tag{2.5}$$

і вибрати у якості ведучого рядок, в якому цей коефіцієнт розташовується (дана схема вибору ведучого значення носить найменування методу головних

елементів). Обчислювальна складність прямого ходу алгоритму Гауса з вибором ведучого рядка має порядок $O(n^3)$ [3,7].

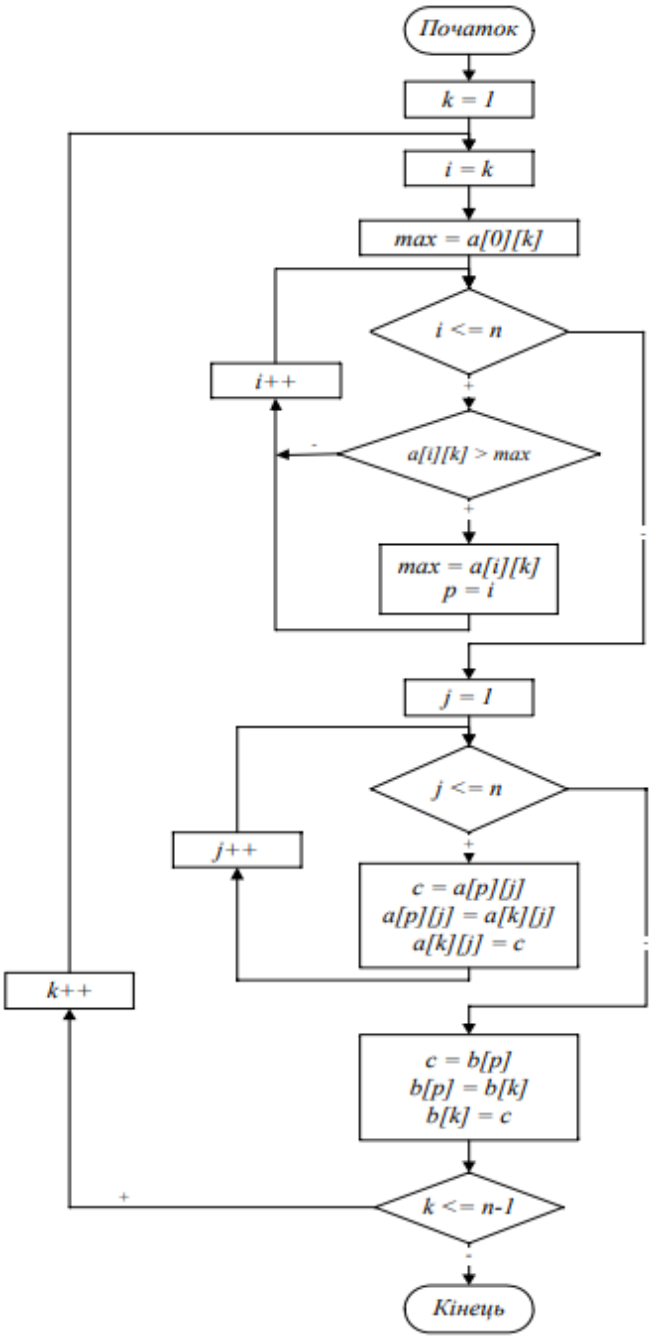


Рис. 2.3 – Блок-схема вибору головного елемента

Після того, як матриця коефіцієнтів приведена до верхнього трикутного вигляду, можна визначити значення невідомих. Починаючи з останнього рівняння у перетвореній системі, можна обчислити значення змінної x_{n-1}, після цього з передостаннього рівняння стає можливим визначення змінної x_{n-2} і т.д. В

загальному вигляді обчислення, які виконуються при зворотному ході методу Гауса можуть бути представлені за допомогою співвідношень [3-6]:

$$x_n = \frac{b_n}{a_{nn}} \tag{2.6}$$

$$x_{n-1} = \frac{b_{n-1} - a_{n-1,n} x_n}{a_{n-1,n-1}} \tag{2.7}$$

$$x_i = \frac{\left(b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j\right)}{a_{ij}}, \tag{2.8}$$

де i = n - 1, …, 1; j = i + 1, …, n.

Пояснимо, як і раніше, виконання зворотного ходу методу Гауса на прикладі системи лінійних рівнянь:

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 + x_2 &= 16 \\ 3x_2 &= 9 \end{aligned}$$

Із останнього рівняння системи можна визначити, що невідома $x_2 = 3$. В результаті стає можливим розв'язок другого рівняння і визначення значення невідомої $x_1 = 13$, тобто

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1 \\ x_1 &= 13 \\ x_2 &= 3 \end{aligned}$$

На останній ітерації зворотного ходу методу Гауса визначається значення невідомої $x_0 = -44$. З урахуванням подальшого паралельного виконання можна відзначити, що обчислення значень невідомих, що отримуються, може виконуватися відразу у всіх рівняннях системи (і ці дії можуть виконуватися в рівняннях одночасно і незалежно один від одного). Так, в даному прикладі після визначення значення невідомої x_2 система рівнянь може бути приведена до вигляду [4]:

$$\begin{aligned} x_0 + 3x_1 &= 1 \\ x_1 &= 16 \\ 3x_2 &= 9 \end{aligned}$$

Обчислювальна складність зворотного ходу алгоритму Гауса складає $O(n^2)$.

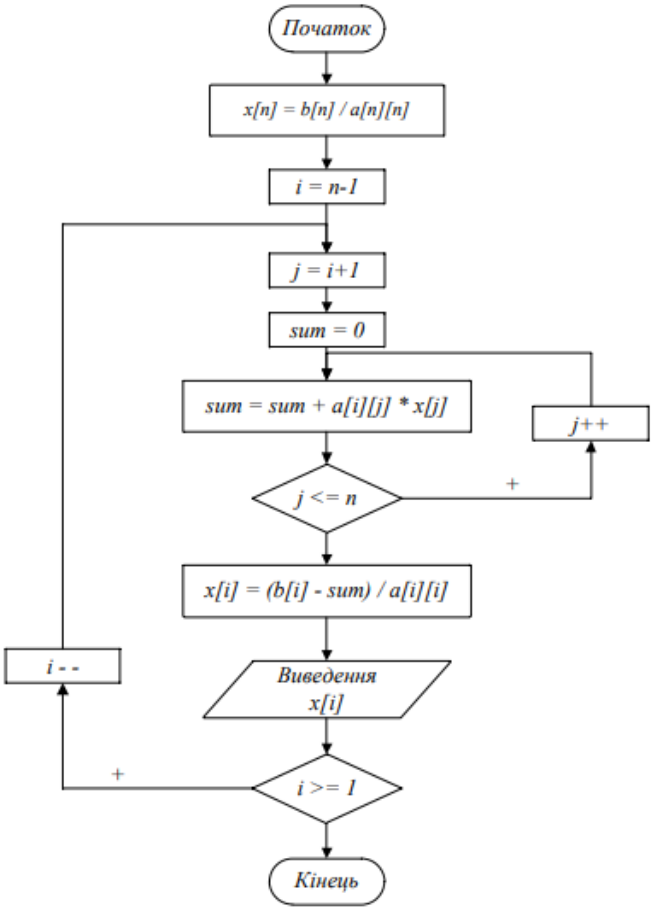


Рис. 2.4 – Блок-схема зворотного ходу алгоритму Гауса

2.1.2 Організація паралельних обчислень

При уважному розгляді методу Гауса можна помітити, що всі обчислення зводяться до однотипних обчислювальних операцій над рядками матриці коефіцієнтів системи лінійних рівнянь. Тому в основу паралельної реалізації алгоритму Гауса було покладено принцип розпаралелювання за даними. У якості базової підзадачі приймаються всі обчислення, пов'язані з обробкою одного рядка матриці A і відповідного елементу вектора b.

Розглянемо загальну схему паралельних обчислень і виникаючі при цьому інформаційні залежності між базовими підзадачами. Для виконання прямого ходу необхідно здійснити (n-1) ітерацію по виключенню невідомих для перетворення матриці коефіцієнтів A до верхнього трикутного вигляду.

Виконання ітерації i, $0 \leq i < n-1$, прямого ходу включає ряд послідовних дій. Перш за все, на самому початку ітерації обираємо ведучий рядок, який при використанні методу головних елементів визначається пошуком рядка з найбільшим по абсолютній величині значенням серед елементів стовпця i, відповідного змінній x_i, що виключається. Оскільки рядки матриці A розподілені по підзадачах, для пошуку максимального значення підзадачі з номерами k, $k<i$, обмінюються своїми елементами при змінній x_i, що виключається. Після збору всіх необхідних даних в кожній підзадачі визначається, яка з підзадач містить ведучий рядок і яке значення є провідним елементом.

Далі для продовження обчислень провідна підзадача розсилає свій рядок матриці A і відповідний елемент вектора b решті підзадач з номерами k, $k<i$. Одержавши ведучий рядок, підзадачі виконують віднімання рядків, забезпечуючи тим самим виключення відповідної невідомої x_i.

При виконанні зворотного ходу методу Гауса підзадачі виконують необхідні обчислення для знаходження значення невідомих. Як тільки яка-небудь підзадача i, $0 \leq i < n-1$, визначає значення своєї змінної x_i, це значення розсилається всім підзадачам з номерами k, $k < i$. Далі підзадачі підставляють набуте значення нової невідомої і виконують коректування значень для елементів вектора b [10-15].

Виділені базові підзадачі характеризуються однаковою обчислювальною трудомісткістю і збалансованим об'ємом даних, що передаються. У разі коли розмір матриці, що описує систему лінійних рівнянь, виявляється більшим, ніж число доступних процесорів (тобто $p<n$), базові підзадачі можна укрупнити, об'єднавши в рамках однієї підзадачі декілька рядків матриці. Проте вживання послідовної схеми розділення даних для паралельного розв’язання систем лінійних рівнянь приведе до нерівномірного обчислювального навантаження процесорів: по мірі виключення (на прямому ходу) або визначення (на зворотному ходу) невідомих в методі Гауса для все більшої частини процесорів всі необхідні обчислення будуть завершені і процесори будуть простоювати.

Можливе розв’язання проблеми балансування обчислень може полягати у використанні стрічкової циклічної схеми для розподілу даних між укрупненими підзадачами. В цьому випадку матриця A ділиться на набори (смуги) рядків вигляду [27]:

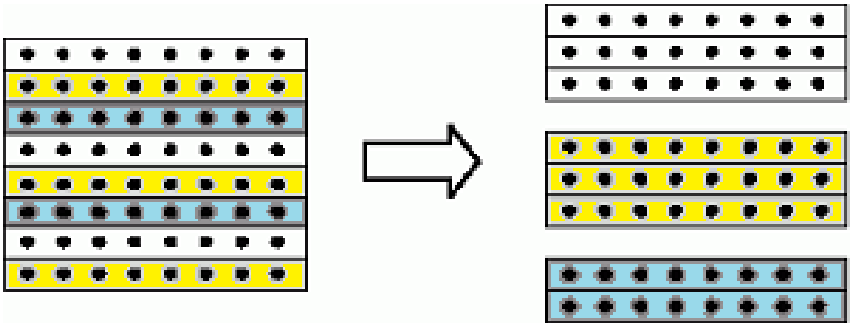


Рис. 2.5 - Приклад використання стрічкової циклічної схеми розділення рядків матриці між трьома процесорами

Зіставивши схему розділення даних і порядок виконання обчислень в методі Гауса, можна відзначити, що використання циклічного способу формування смуг дозволяє забезпечити краще балансування обчислювального навантаження між підзадачами. Розподіл підзадач між процесорами повинен враховувати характер комунікаційних операцій, які виконуються в методі Гауса. Однією з ключових операцій в області інформаційної взаємодії підзадач є передача даних від одного процесора до всіх інших процесорів у складі обчислювальної системи [16].

2.1.3 Паралельна реалізація

Task Parallel Library (TPL) - це набір загальнодоступних типів та API у просторах імен System.Threading та System.Threading.Tasks. Мета TPL - зробити розробників більш продуктивними, спростивши процес додавання паралелізму та одночасності у додатки. TPL динамічно масштабує ступінь паралелізму для найбільш ефективного використання всіх доступних процесорів. Крім того, TPL займається розбиттям роботи, плануванням потоків у пулі ThreadPool, підтримкою скасування, управлінням станами та іншими низькорівневими

деталями. Використовуючи TPL, можна максимізувати продуктивність вашого коду, зосередившись на роботі, яку покликана виконувати ваша програма.

У C# TPL є найкращим способом написання багатопотокового та паралельного коду. Однак, не весь код підходить для розпаралелювання. Наприклад, якщо цикл виконує лише невеликий обсяг роботи на кожній ітерації або виконується небагато ітерацій, то накладні витрати, пов'язані з розпаралелюванням, можуть призвести до того, що код працюватиме повільніше. Крім того, розпаралелювання, як і будь-який багатопотоковий код, додає складності виконанню вашої програми. Хоча TPL спрощує багатопотокові сценарії, ми рекомендуємо вам мати базове розуміння концепцій багатопотоковості, наприклад, блокування, тупики та умови перегонів, щоб ви могли ефективно використовувати TPL. У моєму випадку, паралельна обробка використовується для оптимізації різних етапів методу Гауса [28].

У даному коді реалізоване паралельне програмування за допомогою багатопоточності (multithreading) для розв'язання системи лінійних рівнянь методом Гауса. Моя програма використовує бібліотеку System.Threading.Tasks для створення та виконання паралельних задач.

Основні кроки паралельного розв'язання системи лінійних рівнянь методом Гауса виконуються у методі Solve класу LinearEquationSolver. Для цього використовуються обчислювальні задачі (Task), які виконують обчислення для певних частин матриці. Кожен потік (задача) відповідає за обчислення певного діапазону рядків матриці A. Основні етапи паралельного розв'язання:

- Перша фаза: кожен потік вибирає головний елемент у своєму діапазоні.
- Друга фаза: кожен потік виконує редукцію рядка для свого діапазону.
- Третя фаза: основний потік виконує зворотній хід методу Гауса після завершення фаз 1 та 2 всіма потоками.

Можна зосередити увагу на використанні Task.Run для створення та виконання паралельних задач в циклі.

```
public void Solve()
{
    // ... (інші частини методу залишаються незмінними)

    var tasks = new Task[Program.MAXTHREADS];
    object lockObject = new object(); // новий lockObject для поточного методу

    for (int tid = 0; tid < Program.MAXTHREADS; tid++)
    {
        tasks[tid] = Task.Run(() =>
        {
            // ... (розділення роботи між потоками)

            lock (lockObject)
            {
                // ... (критична секція коду, що взаємодіє з об'єктом equation)
                // ... (обчислення відповідної частини матриці та вектора)
            }
        });
    }

    Task.WhenAll(tasks).Wait();
```

Рис. 2.6 – Фрагмент основного блоку паралельної реалізації

Це дозволяє розділити обчислення між різними потоками, що може призвести до покращення продуктивності в системах з багатьма ядрами процесора. Однак важливо враховувати можливі проблеми з конкурентністю при звертанні до об'єктів з багатьох потоків.

Кожен потік обчислює свою частину матриці та вектора. Використовується lock для забезпечення потокобезпечності під час взаємодії з об'єктом equation. Основна робота відбувається у внутрішньому циклі, де проводяться етапи методу Гауса.

Ще я використовую блок коду Task.WhenAll(tasks).Wait() для очікування завершення всіх потоків перед продовженням виконання програми. Це дозволяє упевнитися, що всі потоки завершили свою роботу перед тим, як я переходжу до наступного етапу алгоритму.

Також, важливо зазначити, я використовую константу Program.MAXTHREADS для визначення кількості потоків, яку буду використовувати. Визначення оптимальної кількості потоків може бути складною задачею і залежить від конкретних характеристик системи та обчислювальної задачі.

2.2 Інтеграція нейронної мережі у модель

2.2.1 Вибір, побудова та інтеграція нейронної мережі

Для інтеграції в паралельну реалізацію методу Гауса було обрано нейронні мережі прямого поширення з наступних причин:

1. Вони може бути точно описані математичними формулами. Це важливо для паралельної реалізації, оскільки дозволяє легко розподілити роботу між різними процесорами або ядрами.

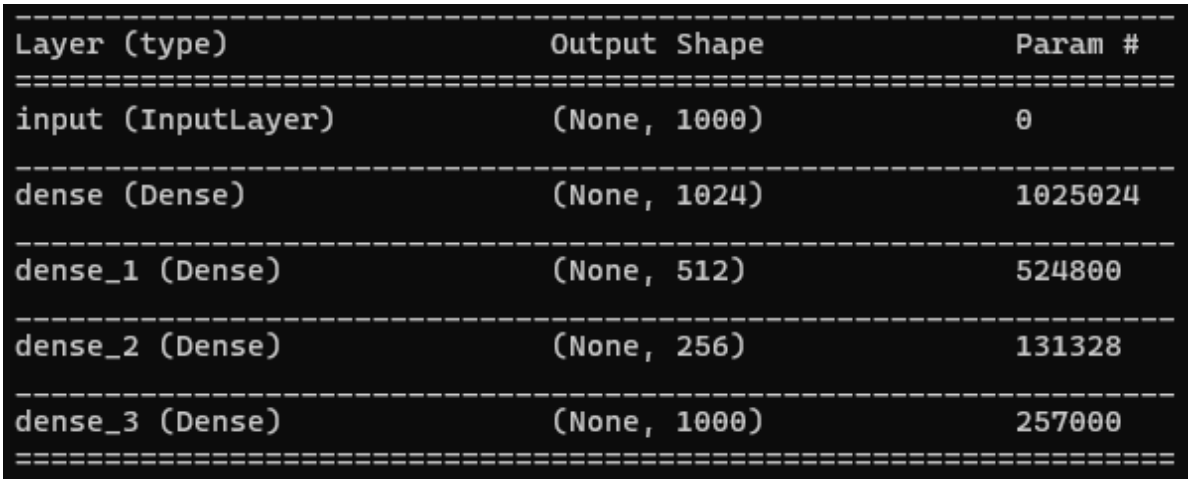
2. Такі нейронні мережі можуть бути використані для вирішення широкого спектру задач, включаючи лінійне рівняння, диференціальні рівняння та оптимізацію. Це робить її ідеальною для вирішення задач, які виникають у методі Гауса.

3. Вони може бути легко розширені для вирішення задач більшого розміру. Це важливо для паралельної реалізації, оскільки дозволяє масштабувати систему для задоволення потреб більших задач.

Для побудови нейромережі було використано TensorFlow.NET (TF.NET). Його метою є реалізація повного API Tensorflow на мові C#, що дозволяє розробникам .NET розробляти, навчати та розгортати моделі машинного навчання за допомогою крос-платформного фреймворку .NET Standard. TensorFlow.NET має вбудований високорівневий інтерфейс Keras і випускається як незалежний пакет TensorFlow.Keras. У створеній нейронній мережі використовуються Dense шари (повнозв'язні шари) [29,30]:

- Вхідний шар input: shape - кількість вхідних змінних.
- Перший прихований шар (Dense): units - 1024 (кількість нейронів у шарі), activation - "relu" (функція активації ReLU).

- Другий прихований шар (Dense): units: 512, activation: "relu".
- Третій прихований шар (Dense): units: 256. activation: "relu".
- Вихідний шар output.



Layer (type)	Output Shape	Param #
input (InputLayer)	(None, 1000)	0
dense (Dense)	(None, 1024)	1025024
dense_1 (Dense)	(None, 512)	524800
dense_2 (Dense)	(None, 256)	131328
dense_3 (Dense)	(None, 1000)	257000

Рис. 2.7 – Модель нейронної мережі

2.2.2 Процес навчання та його параметри

Спочатку підготуємо дані x_train і y_train, є вхідними та вихідними даними відповідно:

- x_train містить випадкові значення для вхідних змінних.
- y_train містить випадкові значення для відповідей (результатів рівнянь).

Набір тренувальних даних був сформований заздалегідь і дані нейронна мережа отримує з датасету за допомогою спеціально розробленого методу LoadEquationFromCSVNeural.

Процес навчання нейронної мережі відбувається під час виклику методу model.fit. Вказується кількість епох (epochs = 1000), тобто повних проходів через тренувальний набір даних. Задається розмір партії (batch size = 1024), тобто кількість вибірок, які обробляються за один раз перед оновленням ваг моделі. Для валідації моделі було використано 20% від тренувального набору.

Під час навчання, модель автоматично оцінюється на тренувальних та валідаційних даних. Під час компіляції моделі за допомогою методу model.compile, вказані різні параметри для оцінки нейронної мережі.

Оптимізатор (optimizer): визначає алгоритм оновлення ваг моделі під час навчання. У моєму випадку використовується оптимізатор Adam з вказаною швидкістю навчання (learning rate = 0.02).

Функція втрат (loss): визначає, як модель оцінює свої власні прогнози в порівнянні з правильними відповідями під час навчання. У моєму випадку використовується Mean Squared Error, тобто метрика, яка використовується для вимірювання середньої квадратичної похибки між прогнозованими значеннями і справжніми значеннями.

Метрики (metrics): показники, які використовуються для визначення того, наскільки добре модель працює під час навчання та оцінки. У вашому випадку використовуються дві метрики:

- "mae" (Mean Absolute Error) - середня абсолютна похибка.
- "accuracy" - точність.

Метрики, вказані при компіляції, виводяться для кожної епохи. Використовуються ці метрики для визначення ефективності моделі.

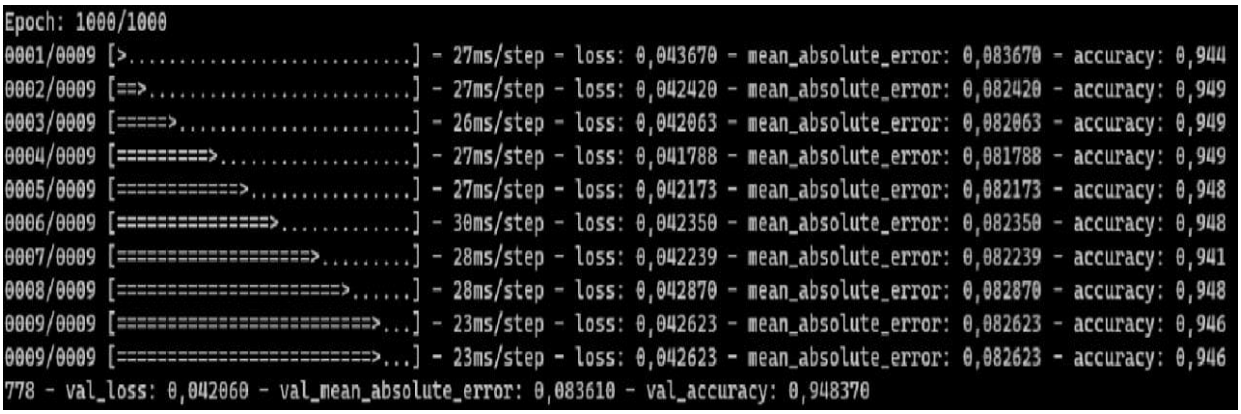


Рис. 2.8 – Процес навчання нейронної мережі

Процес навчання триває до досягнення моделлю прийнятного рівня точності. Оптимальні параметри можуть змінюватися в залежності від завдання та характеристик даних. Після завершення навчання модель може бути збережена для подальшого використання.

2.3 Архітектура комп'ютерної моделі

Тепер після реалізації головних компонентів, можна повністю розглянути нашу розроблену комп’ютерну модель для реалізації методу Гауса з елементами штучного інтелекту.

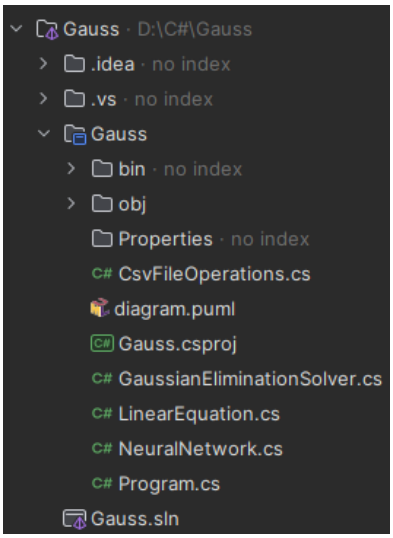


Рис. 2.9 – Структура розробленої моделі

На рис. 2.9 представлена структура каталогів та файлів моделі, написаної на C#. Вона називається "Gauss" і призначена для розв'язання систем лінійних рівнянь методом Гауса за допомогою паралельної обробки та нейронних мереж. Верхній рівень каталогу містить три підкаталоги:

1. idea - містить файли проекту, створені в середовищі розробки Rider.

2. vs - містить файли проекту, створені в середовищі розробки Visual Studio.

3. Gauss - містить файли, що безпосередньо входять до складу програми.

Каталог Gauss містить наступні файли:

CsvFileOperations.cs - містить код для читання та запису даних у файли CSV.

diagram.puml - містить діаграму UML програми.

Gauss.csproj - файл проекту програми, написаний у форматі XML.

GaussianEliminationSolver.cs - містить код для розв'язання систем лінійних рівнянь методом Гауса.

LinearEquation.cs - містить клас, що представляє систему лінійних рівнянь.

NeuralNetwork.cs - містить код для створення та навчання нейронної мережі.

Program.cs - містить основний код програми.

Каталог bin містить виконувані файли програми. Він також містить каталоги obj і Properties, які містять тимчасові файли, що створюються під час компіляції програми.

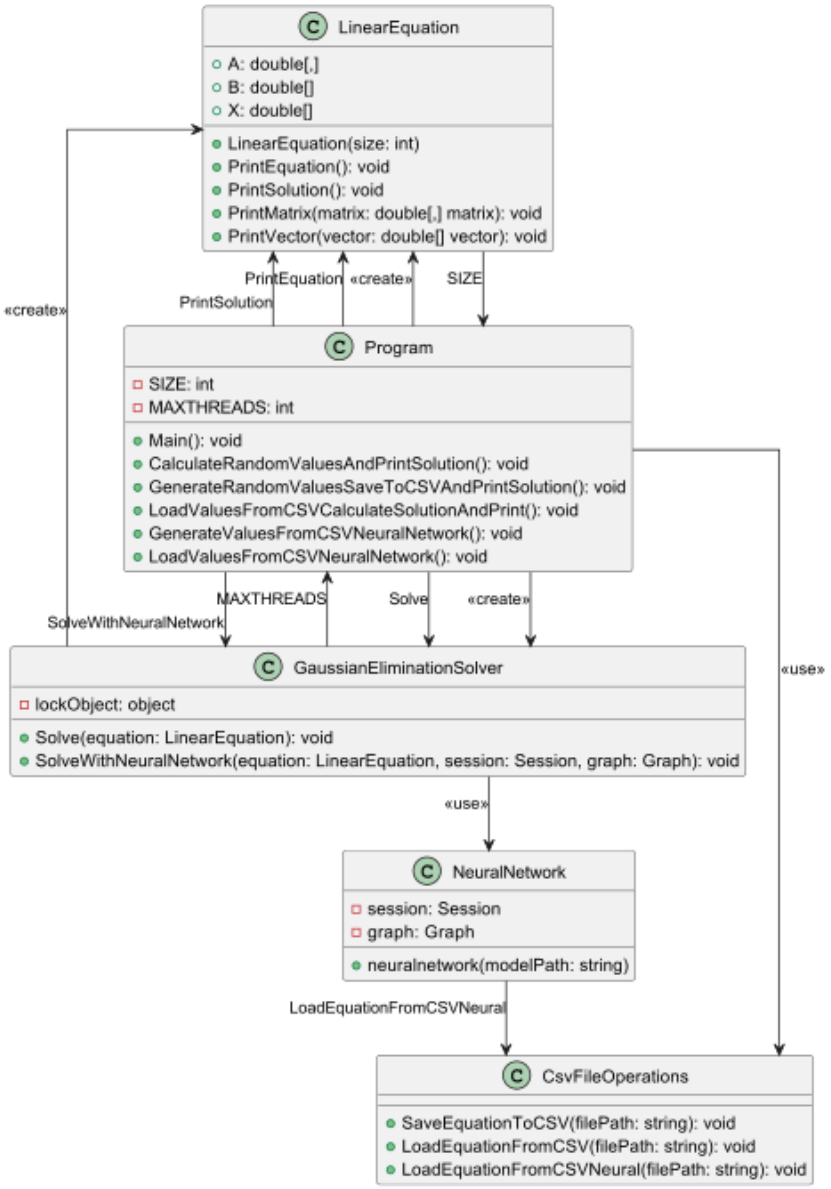


Рис. 2.10 – UML діаграма розробленої моделі

Було побудовано UML діаграму, що використовується для опису структури моделі шляхом показу класів системи, їх атрибутів, методів та відносин між класами. Ось розбивка компонентів, які видно на діаграмі:

1. Клас LinearEquation представляє лінійне рівняння. Він має масиви A, B та X, які, ймовірно, представляють коефіцієнти рівняння, константи та вектор рішення відповідно. Включає методи для друкування рівняння та рішення, а також матриці та вектора.

2. Клас Program є основним класом програми, який містить визови методів для основних операцій, таких як обчислення та друк рішень, збереження в CSV-файли тощо. Він використовує статичні змінні для розміру та максимальної кількості потоків.

3. Клас GaussianEliminationSolver призначений для вирішення лінійних рівнянь за допомогою методу Гауса. Має методи для рішення рівнянь за допомогою паралельної обробки та нейромережі.

4. Клас NeuralNetwork містить нейрону мережу, має методи для завантаження моделі та навчання.

5. Клас CsvFileOperations містить методи для роботи з CSV-файлами, такі як збереження рівнянь у файл, завантаження рівнянь з файлу та завантаження рівнянь з файлу для нейронної мережі.

На діаграмі також зображені зв'язки між класами, які визначають, як вони взаємодіють між собою. Наприклад, зв'язок "створює" (create) вказує на те, що один клас створює екземпляри іншого класу. Зв'язок "використовує" (use) вказує на те, що один клас використовує функції або методи іншого класу.

Висновки до розділу 2

В розділі було розглянуто алгоритм паралельної реалізації методу Гауса. Починаючи з алгоритму Гауса, ми представили послідовну реалізацію, організацію паралельних обчислень та розглянули деталі паралельної реалізації, що сприяє підвищенню продуктивності обчислень.

Далі була розглянута інтеграція нейронної мережі у модель. Визначивши процес вибору, побудови та інтеграції нейронної мережі, ми також розглянули важливість вибору та підготовки даних для навчання. Процес навчання та відповідні параметри були визначені для забезпечення оптимальної ефективності нейронної мережі.

В кінці було досліджено архітектуру розробленої комп'ютерної моделі, де об'єднано різні аспекти алгоритму Гауса та нейронної мережі в єдину функціональну систему. Отримані результати та розроблені рішення становлять важливий внесок у сферу обчислювальних методів та застосування нейронних мереж у вирішенні складних завдань.

РОЗДІЛ 3
ЕКСПЕРИМЕНТИ ТА РЕЗУЛЬТАТИ

3.1 Вибір та підготовка тестових даних

Тестові дані для вирішення систем лінійних рівнянь методом Гауса були підготовлені наступним чином:

1. Визначив кількість рівнянь у системі та кількість невідомих. Нехай система має n рівнянь і m невідомих.

2. Створив коефіцієнтну матрицю A розмірністю n x m, де кожен елемент a_{ij} - це коефіцієнт при невідомій j у рівнянні i.

3. Створив вектор вільних членів b розмірністю n, де кожен елемент b_i - це вільний член у рівнянні i.

4. Записав систему рівнянь у файл за допомогою спеціально створеного методу SaveEquationToCSV. Було створено декілька наборів даних різного розміру для подальших експериментів.

5. Для того щоб витягнути дані та користуватися ними теж було створено спеціальний метод LoadEquationFromCSV, після чого можна ними користуватися для розв’язку систем лінійних рівнянь.

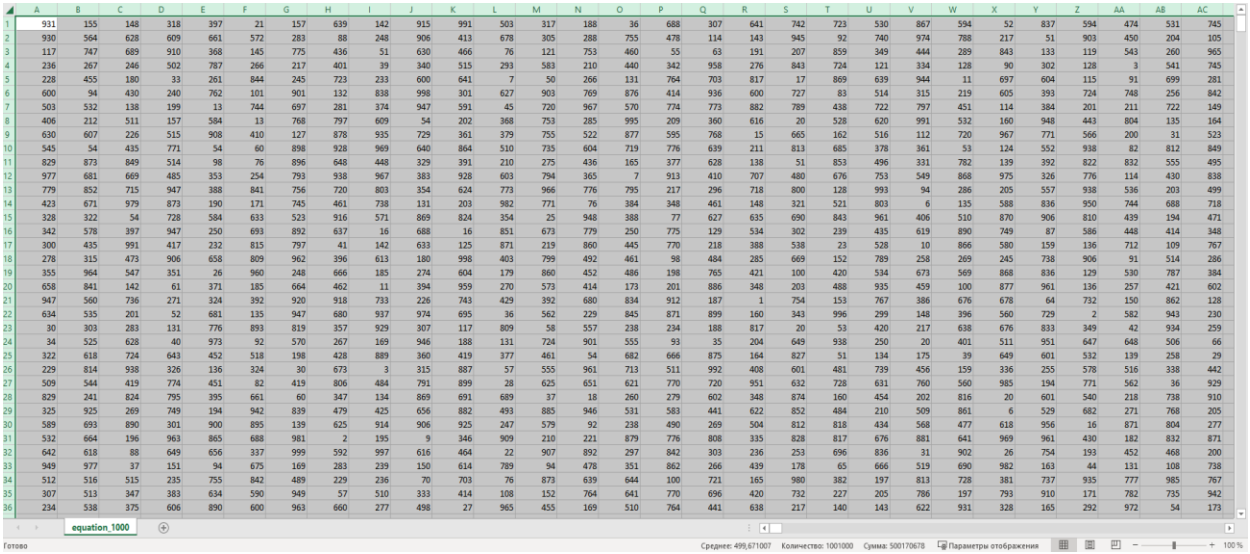


Рис. 3.1 – Приклад набору даних

3.2 Оцінка ефективності нейронної мережі

На рис. 3.2 наведено результати оцінки продуктивності моделі на валідаційному наборі даних під час навчання.

- Значення втрат на валідаційному наборі (val loss): 0,042060 - число вказує, наскільки добре прогнози моделі відповідають справжнім значенням на валідаційних даних. Втрати вимірюються у тих самих одиницях, що і квадрати різниці між прогнозованими та справжніми значеннями.
- Середня абсолютна похибка на валідаційному наборі (val_mean_absolute_error): 0,083610 – вказує на середню абсолютну величину похибки між прогнозованими і справжніми значеннями.
- Точність (val_accuracy): 0,948370 - вимірює, яка частка правильно класифікованих прикладів на валідаційних даних. приблизно 94.84% валідаційних прикладів були правильно класифіковані моделлю.

Рис. 3.2 – Оцінки продуктивності моделі

Виходячи з цих результатів, можна зробити висновок, що модель досягла досить низького рівня втрат і добре пристосувалася до валідаційних даних. Точність також є високою, що свідчить про ефективність моделі.

3.3 Результати паралельних обчислень

Порівняти розроблену паралельну реалізацію с послідовною можна використовуючи показники прискорення (S) та ефективності (E). Для розрахунку можна скористатися наступними формулами:

$$S = \frac{T_{послід.}}{T_{парал.}} \tag{3.1}$$

де $T_{послід.}$ - час виконання послідовної програми, $T_{парал.}$ - час виконання паралельної програми.

$$E = \frac{S}{P} \tag{3.2}$$

де S – прискорення, P – кількість потоків.

Для досягнення позитивного результату у паралельному обчисленні, прискорення повинно бути більше 1, оскільки ми очікуємо зменшення часу виконання програми при використанні паралельних ресурсів порівняно з послідовним виконанням. Таким чином, чим більше прискорення, тим краще. Щодо ефективності, вона також повинна бути більшою за 0. Якщо ефективність дорівнює 1, це означає, що всі використані ресурси використовуються ефективно. Значення ефективності менше 1 вказує на те, що є деяка неефективність у використанні паралельних ресурсів. Ідеальне прискорення та ефективність будуть дорівнювати кількості використаних ресурсів (якщо S = P та E = 1).

Обчислення відбувалися на процесорі Intel Core i7-9750H. Він має шість фізичних ядер, які можуть обробляти 12 одночасних потоків за допомогою Hyper-Threading. Тому було обчислення виконувалися на 2, 4, 6, 8 та 12 потоках. Об’єм оперативної пам’яті становить 32ГБ, яка працює у двуканальному режимі на частоті 2666 МГц, що прискорить доступ даних для процесора та звільняє від проблем з пам’яттю. Для порівняння було використано попередньо сформовані набори даних з різним об’ємом даних: 1000, 2000, 4000, 8000. Це для того щоб працювати з однаковими даними для порівняння результатів.

Таблиця 3.1

Результати виконання обчислень при різному об'ємі даних та кількості потоків

Об’єм даних	Час, s послідовної програми	Час, s на 2 потоках	Час, s на 4 потоках	Час, s на 6 потоках	Час, s на 8 потоках	Час, s на 12 потоках
1000	6,676	3,765	2,198	1,686	1,281	0,971
2000	54,142	30,578	19,078	15,011	10,682	8,202
4000	422,1	245,1	155,313	119,521	90,418	66,989
8000	3212,809	1946,376	1250,399	948,969	734,766	532,554

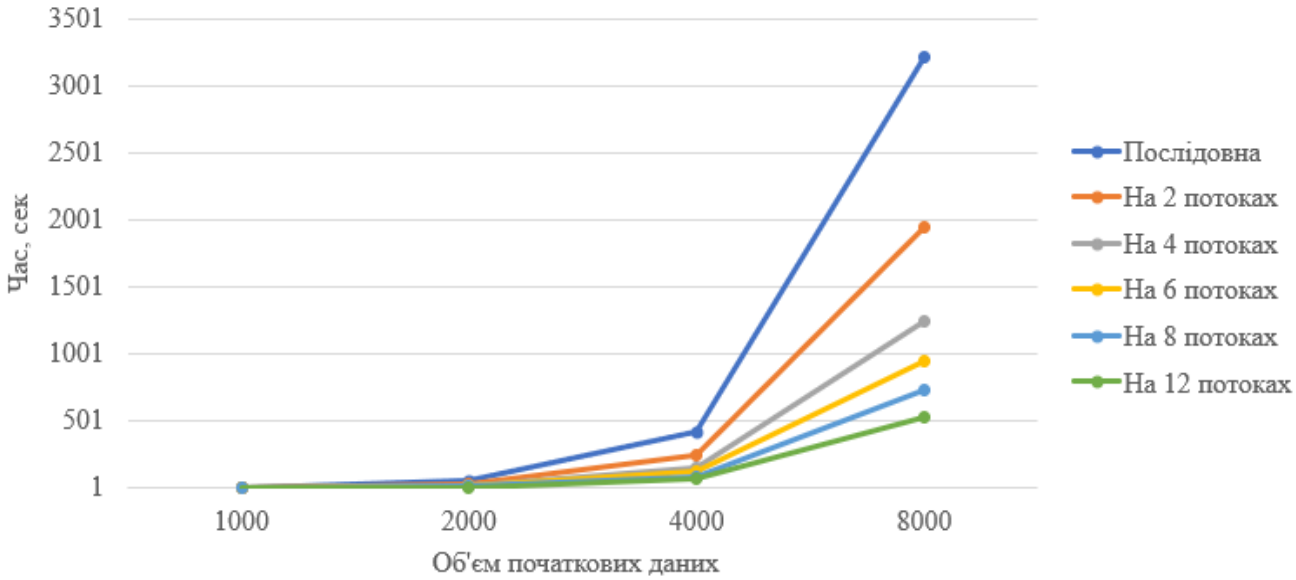


Рис. 3.3 – Графік залежності часу виконання обчислень від об’єму початкових даних для різної кількості потоків

Згідно з результатами представленими в таблиці 3.1 та графіку, можна зробити висновок, при більшій кількості потоків, час виконання обчислень зменшується. Це означає, що паралельне обчислення дозволяє виконувати завдання швидше, ніж послідовне обчислення. Тепер використовуючи формулу 3.1 порахуємо прискорення.

Таблиця 3.2

Результати прискорення обчислень при різному об'ємі даних та кількості потоків

Об’єм даних	Прискорення на 2 потоках	Прискорення на 4 потоках	Прискорення на 6 потоках	Прискорення на 8 потоках	Прискорення на 12 потоках
1000	1,7732	3,0373	3,9596	5,2115	6,8754
2000	1,7706	2,8379	3,6068	5,0685	6,6011
4000	1,7221	2,7177	3,5316	4,6683	6,301
8000	1,6506	2,5694	3,3856	4,3725	6,0328

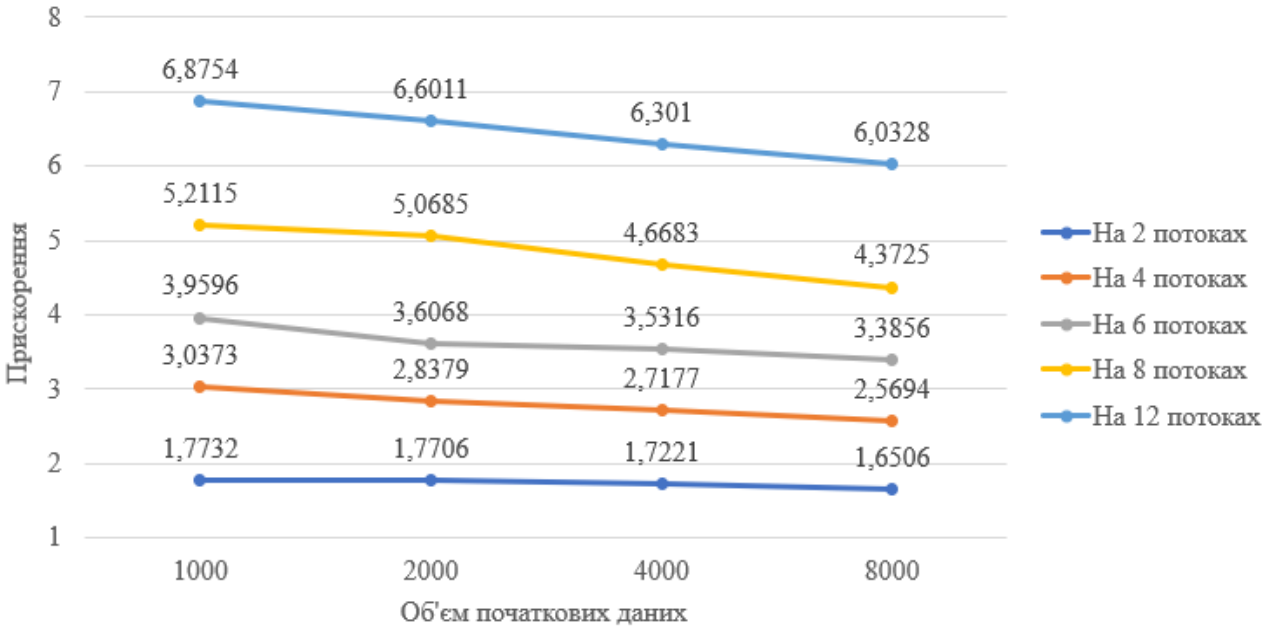


Рис. 3.4 – Графік залежності прискорення від об’єму початкових даних для різної кількості потоків

Згідно результатами представленими в таблиці 3.2 та графіку, можна зробити висновок, що прискорення обчислень зростає зі збільшенням кількості потоків, але при збільшені об’єму даних прискорення стає менше. Тепер використовуючи формулу 3.2 порахуємо ефективність.

Таблиця 3.3

Результати ефективності обчислень при різному об'ємі даних та кількості потоків

Об’єм даних	Ефективність на 2 потоках	Ефективність на 4 потоках	Ефективність на 6 потоках	Ефективність на 8 потоках	Ефективність на 12 потоках
1000	0,8866	0,76	0,66	0,6514	0,573
2000	0,8853	0,71	0,601	0,6335	0,55
4000	0,861	0,6794	0,5885	0,5835	0,525
8000	0,8253	0,6423	0,5643	0,5466	0,503

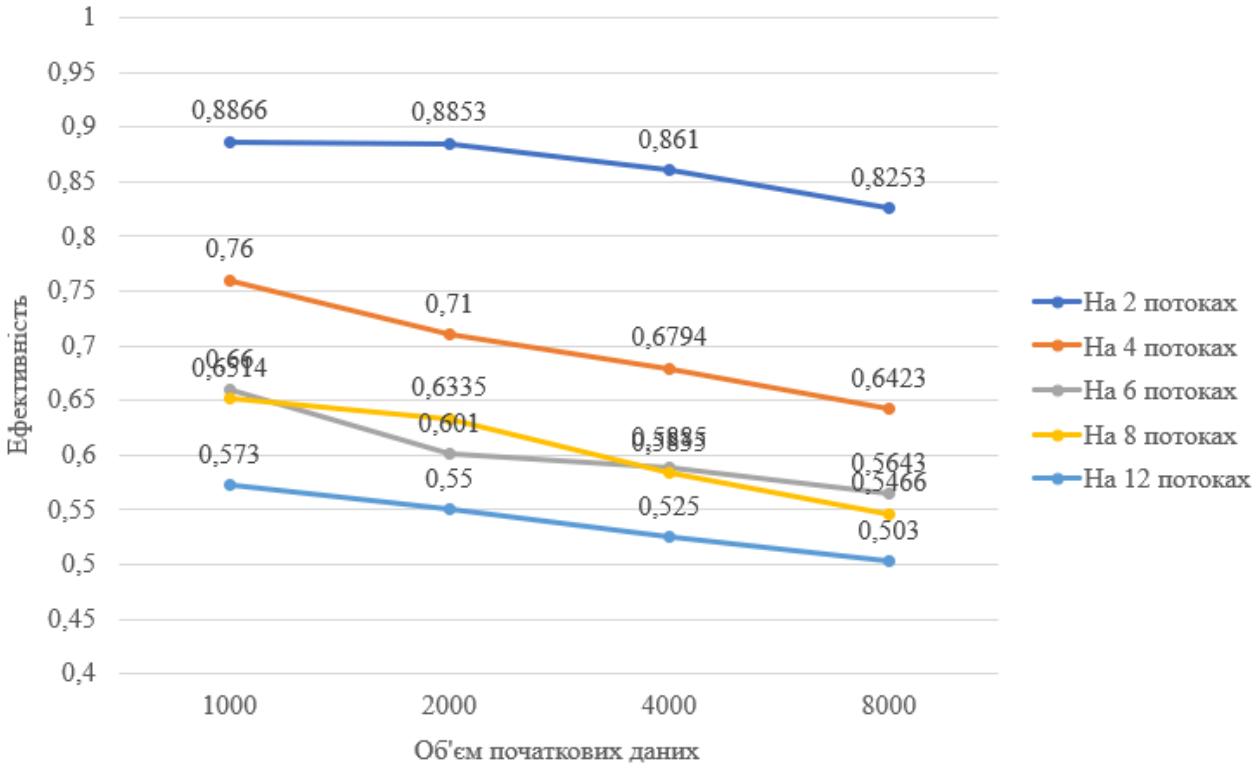


Рис. 3.5 – Графік залежності ефективності від об’єму початкових даних для різної кількості потоків

Згідно з наданою таблицею та графіком, можна зробити такі висновки:

4. Загалом, ефективність обчислень зростає зі збільшенням кількості потоків. Це пов'язано з тим, що більша кількість потоків дозволяє паралельно виконувати більше роботи.

5. Однак, ефективність обчислень знижується зі збільшенням об'єму даних. Це пов'язано з тим, що збільшення об'єму даних збільшує кількість операцій, які необхідно виконати.

6. На 2 потоках ефективність обчислень залишається високою навіть при обробці великих обсягів даних. На 4 і 6 потоках ефективність обчислень також залишається високою при обробці невеликих обсягів даних. На 8 і 12 потоках ефективність обчислень знижується навіть при обробці невеликих обсягів даних. Це пов'язано з тим, що збільшення кількості потоків призводить до збільшення накладних витрат. До них відносяться витрати на перемикання між потоками, а також витрати на доступ до спільних ресурсів. Також це пов'язано з тим, що

процесор використовує частину своєї обчислювальної потужності на роботу системи і всі її процеси, тобто деякі потоки частково зайняті, що несе за собою зниження ефективності при збільшені кількості потоків.

Отримані результати можна вважати позитивними, тому що при використанні більшої кількості потоків час зменшується. Прискорення у всіх випадках більше 1, а ефективність більше 0.5 (найбільша 0,8866, при використанні двох потоків), що вважається позитивним результатом.

Висновки до розділу 3

В розділі було розглянуто процес вибору та підготовки тестових даних. Зосереджено увагу на якості та репрезентативності даних, щоб забезпечити об'єктивність та достовірність експериментів.

Далі представлено детальний аналіз ефективності використаної нейронної мережі. На основі наведених метрик можна зробити висновок, що розроблена нейронна мережа виявилася дуже ефективною та точною в прогнозуванні та класифікації валідаційних даних.

Також було наведено порівняльний аналіз між паралельною та послідовною реалізацією моделі. Розглянуті результати порівняння паралельної реалізації з послідовною за допомогою показників прискорення та ефективності. Порівняння проводилось з різною кількістю потоків та різними об’ємами даних. Було зроблено висновки стосовно отриманих результатів.

ВИСНОВКИ

В даній кваліфікаційній роботі були досягнуті наступні результати:

- визначено постановку задачі даної кваліфікаційної роботи, включаючи мету, об’єкт та предмет дослідження;
- проведено аналіз теоретичних аспектів методу Гауса, паралельних обчислень та нейронних мереж;
- розроблено паралельну реалізацію методу Гауса та виконано опис обчислювального алгоритму;
- інтегровано нейронну мережу у паралельну реалізацію методу Гауса;
- проведено вибір та підготовка даних для навчання НМ;
- проведено експерименти, спрямовані на визначення параметрів моделі та оцінку результатів.
- виконано опис методів обробки та підготовки вхідних даних та проведено підготовку та обробку вхідних даних для покращення ефективності;
- проведено порівняння ефективності послідовного методу Гауса з паралельним та виконано огляд та аналіз отриманих результатів.

Головним результатом даної кваліфікаційної роботи було розроблення комп'ютерної моделі для паралельної реалізації методу Гауса з використанням елементів штучного інтелекту. Ця модель дозволяє адаптувати її до різних конфігурацій обчислювальних систем.

У даній роботі було проведено комплексне дослідження з об'єднання потужності паралельної обробки для ефективного розв'язання лінійних систем рівнянь з використанням нейронної мережі. Задачею було підвищення ефективності методу Гауса за допомогою паралельного програмування та інтеграції НМ у процес розв'язання.

Дослідження також показало, що розроблений метод перевершує традиційний метод Гауса. Практичне значення отриманих результатів полягає в можливості застосування розробленої моделі у високоефективних алгоритмах для сучасних обчислювальних систем використовувати потужність

багатоядерних процесорів та забезпечує значне підвищення ефективності розв'язання лінійних систем рівнянь великих розмірностей. Наприклад, для системи з 1000 рівнянь і 1000 невідомих, розроблена модель забезпечила підвищення швидкості розв'язання майже в 2 рази при двох потоках та в майже в 7 разів при дванадцяти потоках порівняно з послідовною реалізацією методу Гауса. Точність навчання досягла 94,84% і її можна вважати ефективною.

З метою подальшого вдосконалення методу, в майбутньому планується використання глибокого навчання для зменшення розмірності вхідних даних та тестування методу на різних вибірках для адаптації до різних вхідних даних. Такі покращення дозволять зробити розроблений метод більш універсальним та ефективним у різних умовах.

Отже, результати дослідження вказують на успішне досягнення поставлених завдань, а саме підвищення ефективності розв'язання систем рівнянь великих розмірностей методом Гауса об'єднуючи потужність паралельної обробки та нейронної мережі. Розроблена модель має високий потенціал для подальшого практичного використання у високоефективних обчислювальних системах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Дудкін М. Є., Дюженкова О. Ю., Степахно І. В. Вища математика: підручник. Київ: КПІ ім. Ігоря Сікорського, 2022. 449 с. URL: https://ela.kpi.ua/bitstream/123456789/51064/1/Dudkin_V_matymatyka_22.pdf (Дата звернення: 08.08.2023).

2. Вища математика у прикладах і задачах для економістів: навчальний посібник. / Алілуйко А. М. та ін. Тернопіль: ТНЕУ, 2017. 148 с. URL: http://dspace.wunu.edu.ua/bitstream/316497/20458/1/%D0%90%D0%BB%D1%96%D0%BB%D1%83%D0%B9%D0%BA%D0%BE.PDF (Дата звернення: 08.08.2023).

3. Хмельницький М. О. Алгебра та геометрія: навчальний посібник. Київ: КПІ ім. Ігоря Сікорського, 2022. 171 с. URL: https://ela.kpi.ua/bitstream/123456789/48441/1/AGlect.pdf (Дата звернення: 08.08.2023).

4. Задачин В. М., Конюшенко І. Г. Чисельні методи: навчальний посібник. Харків: ХНЕУ ім. С. Кузнеця, 2014. 180 с. URL: http://kist.ntu.edu.ua/textPhD/CHM_Zadachin.pdf (Дата звернення: 08.08.2023).

5. Матриці та системи лінійних рівнянь: навчальний посібник / Савастру О. В., Яковлєва О. М., Драганюк С. В., Болдарєва О. М. Одеса: ОДУ, 2019. 124 с. URL: http://dspace.onu.edu.ua:8080/bitstream/123456789/27831/1/savastru.pdf (Дата звернення: 08.08.2023).

6. Шахно С.М., Дудикевич А.Т., Левицька С.М. Практична реалізація чисельних методів лінійної алгебри: навчальний посібник. Львів: ЛНУ ім. Івана Франка, 2009. 137 с.

7. Попов В. В. Методи обчислень: конспект лекцій для студентів механіко-математичного факультету. Київ: Видавничо-поліграфічний центр "Київський університет", 2012. 303 с. URL: https://mechmat.knu.ua/wp-content/uploads/2018/03/mo_popov.pdf (Дата звернення: 08.08.2023).

8. Кузьма К. Т., Мельник О. В. Паралельні та розподілені обчислення: навчальний посібник для вищих закладів освіти. Миколаїв: МНУ імені В.О. Сухомлинського, 2020. 172 с. URL: http://dspace.mdu.edu.ua/jspui/bitstream/123456789/860/1/%D0%9A%D1%83%D0%B7%D1%8C%D0%BC%D0%B0%2C%20%D0%9C%D0%B5%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA_%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%96%20%D1%82%D0%B0%20%D1%80%D0%BE%D0%B7%D0%BF%D0%BE%D0%B4%D1%96%D0%BB%D0%B5%D0%BD%D1%96%20%D0%BE%D0%B1%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F.pdf (Дата звернення: 17.09.2023).

9. Семеренко В. П. Технології паралельних обчислень: навчальний посібник. Вінниця: ВНТУ, 2018. 104 с. URL: https://pdf.lib.vntu.edu.ua/books/IRVC/2021/Semerenko_2018_104.pdf (Дата звернення: 17.09.2023).

10. Foster I. Designing and Building Parallel Programs. London: Pearson Plc, 2019. 408 p. URL: https://edoras.sdsu.edu/~mthomas/docs/foster/Foster_Designing_and_Building_Parallel_Programs.pdf (Дата звернення: 17.09.2023).

11. Barlas G. Multicore and GPU Programming: An Integrated Approach. Burlington, MA: Morgan Kaufmann Publishers, 2014. 698 p. URL: http://b3.stmik-banjarbaru.ac.id/data.bc/18.%20Programming/18.%20Programming/2015%20Multicore%20and%20GPU%20Programming%20An%20Integrated%20Approach.pdf (Дата звернення: 11.09.2023).

12. Robey R., Zamora Y. Parallel and High Performance Computing. New York, NY: Manning Publications, 2021. 704 p. URL: https://livebook.manning.com/book/parallel-and-high-performance-computing/copyright-2021-manning-publications/v-13/ (Дата звернення: 11.09.2023).

13. Pacheco P. An Introduction to Parallel Programming. Burlington, MA: Morgan Kaufmann Publishers, 2011. 392 p.

14. Pacheco P., Malensek M. An Introduction to Parallel Programming. Burlington, MA: Morgan Kaufmann Publishers, 2020. 496 p. URL: https://books.google.com.ua/books?id=SEmfraJjvfwC&printsec=frontcover&hl=ru#v=onepage&q&f=false (Дата звернення: 11.09.2023).

15. Grama A., Gupta A., Karypis G., Vipin K. Introduction to Parallel Computing. Boston, MA: Addison-Wesley, 2003. 664 p.

16. Parallel Computing. Architectures, Algorithms and Applications / Bischof C., Bücker M., Gibbon P., Joubert G.R., Lippert T., Mohr B., Peters F. OS Press, 2008. 825 p.

17. Кононюк А. Ю. Нейроні мережі і генетичні алгоритми. Київ: «Корнійчук», 2008. – 470 с. URL: https://ep3.nuwm.edu.ua/2252/1/Kononiuk%20NMIGA%20zah.pdf (Дата звернення: 05.10.2023).

18. Руденко О. Г., Бодянський Є. В. Штучні нейронні мережі. Харків: Компанія СМІТ, 2006. 404 с.

19. Коцовський В. М. Нейронні системи: конспект лекцій. Ужгород, 2013. URL: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/16450/1/%D0%9D%D0%B5%D0%B9%D1%80%D0%BE%D0%BD%D0%BD%D1%96%20%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B8.%20%D0%9B%D0%B5%D0%BA%D1%86%D1%96%D1%97.pdf

20. Субботін С. О., Олійник А. О. Нейронні мережі: навчальний посібник. Запоріжжя: ЗНТУ, 2014. 132 с. URL: http://eir.zp.edu.ua/bitstream/123456789/2080/4/Subbotin_Neural_Networks_Tutorial_20141.pdf (Дата звернення: 08.10.2023).

21. Федорін І. В. Методи та технології обчислювального інтелекту: навчальний посібник. Київ: КПІ ім. Ігоря Сікорського, 2022.

22. Субботін С. О. Нейронні мережі: теорія та практика. Житомир: Видавець О. О. Євенок, 2020. 184 с. URL:

http://eir.zntu.edu.ua/bitstream/123456789/6800/1/Subbotin_Neural.pdf (Дата звернення: 08.10.2023).

23. Терейковський І. А., Бушуєв Д. А., Терейковська Л. О. Штучні нейронні мережі: навчальний посібник. Київ: КПІ ім. Ігоря Сікорського, 2022. 123 с. URL: https://ela.kpi.ua/bitstream/123456789/50135/1/ANN.pdf (Дата звернення: 08.10.2023).

24. Hagan Martin T. Neural Network Design / T. Martin Hagan, B. Howard Demuth, Mark Beale. – USA: Colorado University Bookstore, 2002. 734 p.

25. Добровська Л. М., Добровська І. А. Теорія та практика нейронних мереж: навчальний посібник. Київ: НТУУ «КПІ» Видавництво «Політехніка», 2015. 396 с. URL: https://ela.kpi.ua/bitstream/123456789/49841/1/Teoriia_praktyka_neironnykh_merezh.pdf (Дата звернення: 08.10.2023).

26. Madan M. Gupta Statistic and Dynamic Neural Networks: From Fundamentals to Advanced Theory / M. Gupta Madan, Jin Liang, Homma Noriyasu. USA : IEEE Press, 2003. 722 p.

27. Сучасні інформаційні технології: матеріали засідань школи семінару. Випуск 7. Харків: ХНУ імені В.Н. Каразіна, 2023. 84 с.

28. Task Parallel Library (TPL). URL: Режим доступу: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl (Дата звернення: 10.09.2023).

29. C# documentation. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (Дата звернення: 21.11.2023).

30. TensorFlow.NET. URL: Режим доступу: https://scisharp.github.io/tensorflow-net-docs/#/essentials/introduction?id=why-tensorflow-in-c-and-f- (Дата звернення: 04.10.2023).

ДОДАТКИ

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Факультет комп'ютерних наук
Кафедра теоретичної та прикладної системотехніки
Рівень вищої освіти (освітньо-кваліфікаційний рівень) **Магістр**
Галузь знань: **15 – Автоматизація та приладобудування**
Спеціальність: **151 – «Автоматизація та комп'ютерно-інтегровані технології»**

ЗАТВЕРДЖУЮ
Завідувач кафедри теоретичної та прикладної системотехніки
_____д.т.н., проф. Шматков С. І.

«08 » грудня 2022 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Востришев Ілля Сергійович
(прізвище, ім'я, по батькові студента)

1. Тема роботи «Комп'ютерна модель паралельної реалізації методу Гауса з елементами штучного інтелекту»

керівник роботи Толстолузька Олена Геннадіївна, доктор технічних наук, с.н.с.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «10» листопада 2023 року № 4101-5/3197

2. Строк подання студентом роботи 28 .11.2023

3. Перелік питань, які потрібно розробити
1) Постановка задачі паралельної реалізації методу Гауса для розв'язання систем лінійних рівнянь.
2) Обґрунтування та вибір середовища розробки моделі.
3) Огляд та аналіз методів Гауса та штучного інтелекту.
4) Розробка моделі паралельної реалізації методу Гауса з елементами штучного інтелекту.
6) Розробка програмної реалізації моделі та інтерфейсу користувача.
7) Дослідження розробленої моделі, перевірка на працездатність та оцінка результатів моделювання.
8) Підготовка та оформлення пояснювальної записки.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1	Визначення постановки задачі та актуальності	08.12.2022 – 25.12.2022
2	Огляд існуючих методів Гауса та штучного інтелекту	26.12.2022 – 17.01.2023
3	Аналіз та підбір літератури та створення літературної бази для розробки моделі	18.01.2023 – 13.02.2023
4	Розробка комп'ютерної моделі паралельної реалізації методу Гауса з елементами штучного інтелекту	14.02.2023 – 22.04.2023
5	Реалізація комп'ютерної моделі	23.04.2023 – 16.07.2023
6	Валідація та тестування моделі	17.07.2023 – 19.08.2023
7	Опис та аналіз отриманих результатів	20.08.2023 – 21.09.2023
8	Висновки та розробка рекомендацій щодо застосування комп'ютерної моделі	22.09.2023 – 20.10.2023
9	Оформлення пояснювальної записки	21.10.2023 – 14.11.2023
10	Представлення кваліфікаційної роботи науковому керівнику і рецензенту	15.11.2023 – 28.11.2023

5. Дата видачі завдання 08.12.2022 р.

Студент І. С. Вострішев
ініціали, прізвище
підпис

Керівник роботи О. Г. Толстолузька
ініціали, прізвище
підпис

Додаток Б

Технічне завдання

на розробку програмного виробу

«Комп’ютерна модель паралельної реалізації методу Гауса

з елементами штучного інтелекту»

Назва розділу	Назва і зміст підрозділу
1. Введення	1.1. Назва виробу: комп’ютерна модель паралельної реалізації методу Гауса з елементами штучного інтелекту. 1.2. Галузь застосування: обчислювальна математика.
2. Підстава для розробки	2.1. Навчальний план за спеціальністю 151 – «Автоматизація та комп'ютерно-інтегровані технології». 2.2. Завдання на кваліфікаційну роботу магістра затверджено наказом по університету № 4101-5/3197 від 10.11.2023.
3. Призначення розробки	3.1. Мета розробки програмного виробу: підвищення ефективності процесу обчислень методу Гауса шляхом використання паралельного програмування та елементів штучного інтелекту. 3.2. Призначення виробу: використання виробу для підвищення ефективності процесу обчислень методу Гауса (розв'язання лінійних систем рівнянь великих розмірностей).
4. Вихідні дані	У якості прототипу взяти класичний метод Гауса для розв’язання систем лінійних рівнянь.
5. Технічні вимоги до виробу	5.1. Вимоги до функціональних характеристик: паралельна реалізація методу Гауса з підбором оптимальних параметрів використовуючи нейромережу. 5.2. Вимоги до надійності: забезпечення стійкості та точності результатів при розв'язанні великих систем лінійних рівнянь. 5.3. Вимоги до умов експлуатації: встановлення мови програмування C# та необхідних бібліотек для роботи програмного виробу. 5.4. Вимоги до складу і параметрів технічних засобів:

<table>
<tr><td></td><td colspan="2">комп’ютер або ноутбук із 8 ГБ оперативної пам’яті та процесором не нижче Intel Core i3 9-го покоління або AMD Ryzen 3 1-го покоління.
5.5. Вимоги до інформаційної та програмної сумісності: підтримка ОС Linux або Windows 10/11, підтримка Visual Studio або Rider, C# 11, .NET SDK 6-8.
5.6. Вимоги до маркування та упаковки: відсутні.
5.7. Вимоги до транспортування і зберігання: відсутні.
5.8. Спеціальні вимоги: відсутні.</td></tr>
<tr><td>6. Вимоги до документації.</td><td colspan="2">Документацією до виробу «Комп’ютерна модель паралельної реалізації методу Гауса з елементами штучного інтелекту» вважати:
1) Технічне завдання на розробку виробу (представити як Додаток Б до пояснювальної записки до кваліфікаційної роботи).
2) Програму і методику випробувань розробленого виробу (представити як Додаток В до пояснювальної записки до кваліфікаційної роботи).
3) Опис виробу.
4) Текст програми (представити як Додаток Г до пояснювальної записки до кваліфікаційної роботи).</td></tr>
<tr><td>7. Техніко-економічні показники</td><td colspan="2">Орієнтовна оцінка ефективності повинна бути вище 0.85.
Орієнтовна точність навчання: більше 90%.</td></tr>
<tr><td rowspan="9">8. Стадії і етапи розробки</td><td>Дата</td><td>Назва етапу</td></tr>
<tr><td>08.12.22 – 25.12.22</td><td>1. Визначення постановки задачі та актуальності.</td></tr>
<tr><td>26.12.22 – 17.01.23</td><td>2. Огляд існуючих методів Гауса та штучного інтелекту.</td></tr>
<tr><td>18.01.23 – 13.02.23</td><td>3. Аналіз та підбір літератури та створення літературної бази для розробки моделі.</td></tr>
<tr><td>14.02.23 – 22.04.23</td><td>4. Розробка комп’ютерної моделі паралельної реалізації методу Гауса з елементами штучного інтелекту.</td></tr>
<tr><td>23.04.23 – 16.07.23</td><td>5. Реалізація комп’ютерної моделі.</td></tr>
<tr><td>17.07.23 – 19.08.23</td><td>6. Валідація та тестування моделі.</td></tr>
<tr><td>20.08.23 – 21.09.23</td><td>7. Опис та аналіз отриманих результатів.</td></tr>
<tr><td>22.09.23 – 20.10.23</td><td>8. Висновки та розробка</td></tr>
</table>

		рекомендацій щодо застосування комп’ютерної моделі.
	21.10.23 – 14.11.23	9. Оформлення пояснювальної записки.
	15.11.23 – 28.11.23	10. Представлення кваліфікаційної роботи науковому керівнику і рецензенту.
9. Порядок контролю і приймання	Загальні вимоги до приймання розроблюваного виробу: 1) Перевірка ходу розробки програмного виробу керівнику робіт виконувати 1-2 рази кожні 3 тижні. 2) Випробування виробу відповідно до Програми і методики випробувань провести на базі комп'ютерного класу або на базі приватного приміщення. 3) Захист розробленого виробу провести на засіданні атестаційної комісії. Пояснювальну записку представити на паперових носіях в одному примірнику та в електронному вигляді.	

Виконавець:
студент групи КУ-61
Вострішев І.С.

Замовник:
д. т. н., с.н.с.
Толстолузька О. Г.

Додаток В

Програма і методика випробувань програмного виробу

«Комп'ютерна модель паралельної реалізації методу Гауса з елементами штучного інтелекту»

1. Об'єкт випробувань

1.1 Найменування випробуваного програмного виробу: комп'ютерна модель паралельної реалізації методу Гауса з елементами штучного інтелекту.

1.2 Галузь його застосування: обчислювальна математика.

2. Мета випробувань

Перевірка працездатності виробу та правильності роботи. Підтвердження характеристик розробленого виробу вимогам, які сформульовані в ТЗ.

3. Загальні положення

3.1 Підстави для проведення випробувань

Підставою для проведення випробувань є наказ про призначення атестаційної комісії та перевірку даного виробу.

3.2 Місце і тривалість випробувань

Приймальні (приймально-здавальні) випробування проводяться на базі комп'ютерного класу або будь-якого приміщення в період роботи атестаційної комісії.

3.3 Обсяг випробувань

Приймальні випробування програмного виробу проводяться в обсязі відповідному цієї Програми і методики випробувань.

3.4 Організації, які беруть участь у випробуваннях

Приймальні випробування проводяться атестаційною комісією напередодні засідання (або в процесі засідання) за участю Замовника, Виконавця та інших осіб, присутніх на засіданні.

4. Вимоги до виробу

4.1. Вимоги до функціональних характеристик: паралельна реалізація методу Гауса з підбором оптимальних параметрів використовуючи нейромережу.

4.2. Вимоги до надійності: забезпечення стійкості та точності результатів при розв'язанні великих систем лінійних рівнянь.

4.3. Вимоги до умов експлуатації: встановлення мови програмування C# та необхідних бібліотек для роботи програмного виробу.

4.4. Вимоги до складу і параметрів технічних засобів: комп’ютер або ноутбук із 8 ГБ оперативної пам’яті та процесором не нижче Intel Core i3 8-го покоління або AMD Ryzen 3 1-го покоління.

4.5. Вимоги до інформаційної та програмної сумісності: підтримка ОС Linux або Windows 10/11, підтримка Visual Studio або VS Code, C# 11, .NET SDK 6.

4.6. Вимоги до маркування та упаковки: відсутні.

4.7. Вимоги до транспортування і зберігання: відсутні.

4.8. Спеціальні вимоги: відсутні.

5. Вимоги до документації

Програмною документацією до програмного продукту «Комп'ютерна модель паралельної реалізації методу Гауса з елементами штучного інтелекту» вважати:

1) Технічне завдання на розробку програмного продукту (представити як Додаток Б до пояснювальної записки до кваліфікаційної роботи).

2) Програму і методику випробувань розробленої програми (представити як Додаток В до пояснювальної записки до кваліфікаційної роботи);

3) Опис виробу.

4) Текст програми (представити як Додаток Г до пояснювальної записки до кваліфікаційної роботи).

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Засоби випробувань представлено на ПК на яких встановлено наступні програмні засоби: Visual Studio або Rider, C# 11, .NET SDK 6-8. В Visual Studio створено проект, де встановлені усі необхідні пакети для запуску, такі як TensorFlow.NET, TensorFlow.Keras, SciSharp.TensorFlow.Redist, pythonnet, Keras.NET.

6.2 Порядок проведення випробувань

Як правило, випробування проводяться в два етапи:

- ознайомчий (1-й етап);
- випробування програмного виробу (2-й етап).

Перелік перевірок, що проводяться на 1 етапі випробувань, включає в себе:

1) Перевірку комплектності складу програмної документації здійснюється за критерієм наявності зазначеної в ТЗ документації.

2) Виконання тесту:

2.1) Запустити Visual Studio або Rider.

2.2) Переглянути написаний код та перевірити на наявність помилок.

2.3) Перевірити наявність встановлених пакетів шляхом запуску програми на даних, які згенеровані випадковим чином.

3) Якість програмної документації перевіряється на відповідність вимогам стандартів ЄСПД.

Перелік перевірок, що проводяться на 2 етапі випробувань, включає в себе:

1) Перевірку відповідності технічних характеристик програми вимогам технічного завдання.

2) Перевірку ступеня виконання функціональних вимог до програми.

3) Методику проведення перевірок:

3.1) Запустити Visual Studio або Rider.

3.2) Перевірити наявність встановлених пакетів шляхом запуску програми на даних, які згенеровані випадковим чином. При відсутності бібліотек, втановити їх за допомогою Управління пакетами NuGet.

3.3) Порядок проведення випробувань:

- Запустити програму за допомогою Visual Studio або Rider.

- Перевірити працездатність розробленого програмного продукту.

- Перевірити правильність розв’язку систем лінійних рівнянь методом Гауса.

- Перевірити правильність результатів.

- Перевірити працездатність паралелізму провівши ще декілька розрахунків використовуючи різну кількість потоків.

4) Якщо перевірки на першому та другому етапах виконано успішно, то виріб вважається таким, що пройшов випробування.

Тест 1

1. Перевірка виконання програми;

2. введення розмірності матриці та виконання на одному потоці;

3. отримання результатів і перевірка на правильність.

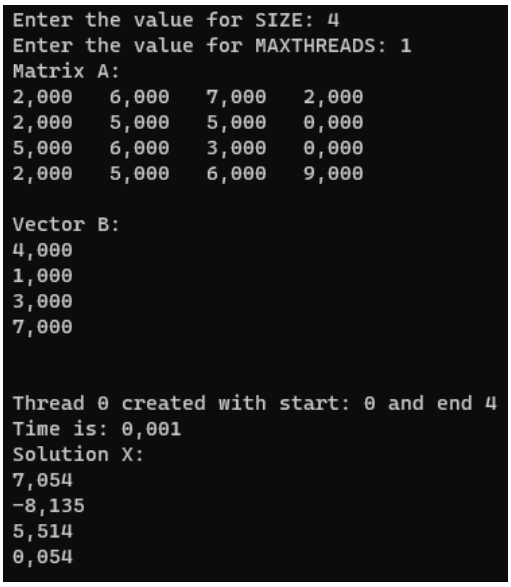


Рис. В.1 – Тест 1

Тест 2

1. Перевірка виконання програми;
2. введення розмірності матриці та виконання на двох та чотирьох потоках;
3. отримання результатів і порівняння часу виконання програми.

```
Enter the value for SIZE: 2000
Enter the value for MAXTHREADS: 2
Equation saved to equation.csv
Thread 0 created with start: 0 and end 1000
Thread 1 created with start: 1000 and end 2000
Time is: 34,362
```

Рис. В.2 - Тест 2.1

```
Enter the value for SIZE: 2000
Enter the value for MAXTHREADS: 4

Loaded Equation:
Thread 0 created with start: 0 and end 500
Thread 1 created with start: 500 and end 1000
Thread 2 created with start: 1000 and end 1500
Thread 3 created with start: 1500 and end 2000
Time is: 22,003
```

Рис. В.3 – Тест 2.2

Виконавець:

студент групи КУ-61 Вострішев І.С. 

Додаток Г

Лістинг Г.1 – Код програмної моделі паралельної реалізації методу Гауса з елементами штучного інтелекту

```
using System.Diagnostics;
using Tensorflow;
using static Tensorflow.Binding;

namespace GaussianElimination;

public class LinearEquation {
    public double[,] A { get; set; }
    public double[] B { get; set; }
    public double[] X { get; set; }

    public LinearEquation(int size) {
        A = new double[size, size];
        B = new double[size];
        X = new double[size];
        Random rand = new Random();
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                A[i, j] = rand.Next(10);
            }
            B[i] = rand.Next(10);
        }
    }

    public void PrintEquation() {
        Console.WriteLine("Matrix A:");
        PrintMatrix(A);
        Console.WriteLine("\nVector B:");
        PrintVector(B);
        Console.WriteLine();
    }
```

```
    public void PrintSolution() {
        Console.WriteLine("Solution X:");
        PrintVector(X);
        Console.WriteLine();
    }

    private void PrintMatrix(double[,] matrix) {
        int rows = matrix.GetLength(0);
        int cols = matrix.GetLength(1);

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                Console.Write($"{matrix[i, j]:F3}\t");
            }
            Console.WriteLine();
        }
    }

    private void PrintVector(double[] vector) {
        int length = vector.Length;

        for (int i = 0; i < length; i++) {
            Console.WriteLine($"{vector[i]:F3}\t");
        }
        Console.WriteLine();
    }
}

public class GaussianEliminationSolver {
    private LinearEquation equation;
    private readonly object lockObject = new object();
    private readonly Graph;
    private readonly Session;
    private LinearEquation equation
    public GaussianEliminationSolver(LinearEquation equation) {
```

```
        this.equation = equation;
        graph = new Graph();

graph.Import(File.ReadAllBytes("./linear_equation_solver_model"));
        session = new Session(graph);
    }

    public void Solve() {
        double start_t = Stopwatch.GetTimestamp() /
(double)Stopwatch.Frequency;
        int size = equation.A.GetLength(0);
        int chunk = size / Program.MAXTHREADS;
        var tasks = new Task[Program.MAXTHREADS];

        for (int tid = 0; tid < Program.MAXTHREADS; tid++) {
            int start = tid * chunk;
            int end = start + chunk;
            Console.WriteLine($"Thread {tid} created with start:
{start} and end {end}");
            tasks[tid] = Task.Run(() => {
                for (int k = 0; k < size; k++) {
                    if (k >= start && k < end) {
                        lock (lockObject) {
                            // Phase 1: Вибір головного елемента
                            double max = Math.Abs(equation.A[k,
k]);
                            int maxIndex = k;
                            for (int i = k + 1; i < size; i++) {
                                if (Math.Abs(equation.A[i, k]) >
max) {
                                    max = Math.Abs(equation.A[i,
k]);
                                    maxIndex = i;
                                }
                            }
```

```
                            // Обмін рядків для вибору головного
елемента
                            if (maxIndex != k) {
                                for (int j = k; j < size; j++) {
                                    double temp = equation.A[k,
j];
                                    equation.A[k, j] =
equation.A[maxIndex, j];
                                    equation.A[maxIndex, j] =
temp;
                                }
                                double tempB = equation.B[k];
                                equation.B[k] =
equation.B[maxIndex];
                                equation.B[maxIndex] = tempB;
                            }

                            // Phase 2: Редукція рядка
                            for (int i = k + 1; i < size; i++) {
                                double factor = equation.A[i, k] /
equation.A[k, k];
                                for (int j = k; j < size; j++) {
                                    equation.A[i, j] -= factor *
equation.A[k, j];
                                }
                                equation.B[i] -= factor *
equation.B[k];
                            }
                        }
                    }
                }
            });
        }
```

```
        Task.WhenAll(tasks).Wait();

        // Phase 3: Зворотній хід методу Гаусса
        for (int k = size - 1; k >= 0; k--) {
            equation.X[k] = equation.B[k];
            for (int j = k + 1; j < size; j++) {
                equation.X[k] -= equation.A[k, j] * equation.X[j];
            }
            equation.X[k] /= equation.A[k, k];
        }

        double end_t = Stopwatch.GetTimestamp() /
(double)Stopwatch.Frequency;
        double diff_time = end_t - start_t;
        Console.WriteLine($"Time is: {diff_time:F3}");
    }


    public void SolveWithNeuralNetwork() {
        double start_t = Stopwatch.GetTimestamp() /
(double)Stopwatch.Frequency;
        equation.X = NeuralNetwork();

        int size = equation.A.GetLength(0);
        int chunk = size / Program.MAXTHREADS;
        var tasks = new Task[Program.MAXTHREADS];

        for (int tid = 0; tid < Program.MAXTHREADS; tid++) {
            int start = tid * chunk;
            int end = start + chunk;
            Console.WriteLine($"Thread {tid} created with start:
{start} and end {end}");
            tasks[tid] = Task.Run(() => {
                for (int k = 0; k < size; k++) {
                    if (k >= start && k < end) {
```

```
                        // Use lock to ensure thread safety
                        lock (lockObject) {
                            // Phase 1: Вибір головного елемента
                            double max = Math.Abs(equation.A[k,
k]);
                            int maxIndex = k;
                            for (int i = k + 1; i < size; i++) {
                                if (Math.Abs(equation.A[i, k]) >
max) {
                                    max = Math.Abs(equation.A[i,
k]);
                                    maxIndex = i;
                                }
                            }

                            // Обмін рядків для вибору головного
елемента
                            if (maxIndex != k) {
                                for (int j = k; j < size; j++) {
                                    double temp = equation.A[k,
j];
                                    equation.A[k, j] =
equation.A[maxIndex, j];
                                    equation.A[maxIndex, j] =
temp;
                                }
                                double tempB = equation.B[k];
                                equation.B[k] =
equation.B[maxIndex];
                                equation.B[maxIndex] = tempB;
                            }

                            // Phase 2: Редукція рядка
                            for (int i = k + 1; i < size; i++) {
```

```
                                double factor = equation.A[i, k] /
equation.A[k, k];
                                for (int j = k; j < size; j++) {
                                    equation.A[i, j] -= factor *
equation.A[k, j];
                                }
                                equation.B[i] -= factor *
equation.B[k];
                            }
                        }
                    }
                }
            });
        }

        Task.WhenAll(tasks).Wait();

        // Phase 3: Зворотній хід методу Гаусса
        for (int k = size - 1; k >= 0; k--) {
            equation.X[k] = equation.B[k];
            for (int j = k + 1; j < size; j++) {
                equation.X[k] -= equation.A[k, j] * equation.X[j];
            }
            equation.X[k] /= equation.A[k, k];
        }
        double end_t = Stopwatch.GetTimestamp() /
(double)Stopwatch.Frequency;
        double diff_time = end_t - start_t;
        Console.WriteLine($"Time is: {diff_time:F3}");
    }
}

public class NeuralNetwork {
    public static void neuralnetwork() {
        var layers = tf.keras.layers;
```

```
        var random = new Random();

        // Параметри для генерації даних
        int numSamples = 10000;
        int numVariables = 100; // Кількість змінних
        int numEquations = 100; // Кількість рівнянь

        // Генерація випадкових даних
        var x_train = new float[numSamples, numVariables];
        var y_train = new float[numSamples, numEquations];

        for (int i = 0; i < numSamples; i++) {
            for (int j = 0; j < numVariables; j++) {
                x_train[i, j] = (float)random.NextDouble();
            }

            for (int j = 0; j < numEquations; j++) {
                y_train[i, j] = (float)random.NextDouble();
            }
        }
        var x_tensor = tf.constant(x_train);
        var y_tensor = tf.constant(y_train);
        var inputs = tf.keras.Input(shape: numVariables, name:
"input");
        var x = layers.Dense(1024, activation:
"relu").Apply(inputs);
        x = layers.Dense(512, activation: "relu").Apply(x);
        x = layers.Dense(256, activation: "relu").Apply(x);
        var outputs = layers.Dense(numEquations).Apply(x);
        var model = tf.keras.Model(inputs, outputs, name:
"linear_equation_solver");
        model.summary();
        model.compile(optimizer: tf.keras.optimizers.Adam(),
            loss: tf.keras.losses.MeanSquaredError(),
            metrics: new[] { "mae", "accuracy" });
```

```
        model.fit(x_train, y_train,
            batch_size: 64,
            epochs: 100,
            validation_split: 0.2f);
        model.save("./linear_equation_solver_model");
    }
}

public class CsvFileOperations {
    public static void SaveEquationToCSV(string filePath,
double[,] A, double[] B) {
        try {
            using (StreamWriter writer = new
StreamWriter(filePath)) {
                int rows = A.GetLength(0);
                int cols = A.GetLength(1);
                for (int i = 0; i < rows; i++) {
                    for (int j = 0; j < cols; j++) {
                        writer.Write($"{A[i, j]:F3}");
                        if (j < cols - 1)
                            writer.Write(";");
                    }

                    writer.Write($";{B[i]:F3}");
                    writer.WriteLine();
                }
            }
        }
        catch (Exception ex) {
            Console.WriteLine($"Error saving equation to CSV:
{ex.Message}");
        }
    }
```

```
    public static void LoadEquationFromCSV(string filePath, out
double[,] A, out double[] B) {
        A = null!;
        B = null!;

        try {
            using (StreamReader reader = new
StreamReader(filePath)) {
                int size = int.Parse(reader.ReadLine()!);
                A = new double[size, size];
                B = new double[size];

                int bIndex = 0;

                while (!reader.EndOfStream) {
                    string line = reader.ReadLine()!;
                    if (line != null) {
                        string[] values = line.Split(';');

                        for (int j = 0; j < size; j++) {
                            if (double.TryParse(values[j].Trim(),
out double parsedValue)) {
                                A[bIndex, j] = parsedValue;
                            }
                            else {
                                Console.WriteLine($"Error parsing
value at index {j}");
                            }
                        }

                        if (values.Length > size) {
                            if
(double.TryParse(values[size].Trim(), out double parsedB)) {
                                B[bIndex] = parsedB;
                                bIndex++;
```

```
                            }
                            else {
                                Console.WriteLine($"Error parsing
value at index {size}");
                            }
                        }
                    }
                }
            }
        }
        catch (Exception ex) {
            Console.WriteLine($"Error loading equation from CSV:
{ex.Message}");
        }
    }

    private float[,] LoadEquationFromCSVNeural(string filePath) {
        try {
            var lines = File.ReadAllLines(filePath);
            int numRows = lines.Length;
            int numCols = lines[0].Split(';').Length;

            float[,] data = new float[numRows, numCols];

            for (int i = 0; i < numRows; i++) {
                string[] values = lines[i].Split(';');

                for (int j = 0; j < numCols; j++) {
                    if (float.TryParse(values[j], out float
parsedValue)) {
                        data[i, j] = parsedValue;
                    }
                    else {
                        Console.WriteLine($"Error parsing value at
index {j}");
```

```
                    }
                }
            }

            return data;
        }
        catch (Exception ex) {
            Console.WriteLine($"Error loading data from CSV:
{ex.Message}");
            return null!;
        }
    }
}

class Program {
    public static int SIZE;
    public static int MAXTHREADS;
    static void Main() {
        Console.WriteLine("Choose a case:");
        Console.WriteLine("1. Calculate random values and print
solution");
        Console.WriteLine("2. Generate random values, save to CSV,
and print solution");
        Console.WriteLine("3. Load values from CSV, calculate
solution, and print");
        Console.WriteLine("4. Generate values from CSV,
NeuralNetwork");
        Console.WriteLine("5. Load values from CSV,
NeuralNetwork");

        int choice = int.Parse(Console.ReadLine()!);

        switch (choice) {
            case 1:
                CalculateRandomValuesAndPrintSolution();
```

```
                break;

            case 2:
                GenerateRandomValuesSaveToCSVAndPrintSolution();
                break;

            case 3:
                LoadValuesFromCSVCalculateSolutionAndPrint();
                break;

            case 4:
                GenerateValuesFromCSVNeuralNetwork();
                break;

            case 5:
                LoadValuesFromCSVNeuralNetwork();
                break;

            default:
                Console.WriteLine("Invalid choice. Exiting
program.");
                break;
        }
    }

    static void CalculateRandomValuesAndPrintSolution() {
        Console.Write("Enter the value for SIZE: ");
        SIZE = int.Parse(Console.ReadLine()!);

        Console.Write("Enter the value for MAXTHREADS: ");
        MAXTHREADS = int.Parse(Console.ReadLine()!);

        LinearEquation equation = new LinearEquation(SIZE);
        equation.PrintEquation();
```

```
        GaussianEliminationSolver solver = new
GaussianEliminationSolver(equation);
        solver.Solve();
        equation.PrintSolution();
    }

    static void GenerateRandomValuesSaveToCSVAndPrintSolution() {
        Console.Write("Enter the value for SIZE: ");
        SIZE = int.Parse(Console.ReadLine()!);

        Console.Write("Enter the value for MAXTHREADS: ");
        MAXTHREADS = int.Parse(Console.ReadLine()!);

        LinearEquation equation = new LinearEquation(SIZE);
        equation.PrintEquation();

        string csvFilePath = "equation.csv";
        equation.SaveEquationToCSV(csvFilePath);
        Console.WriteLine($"Equation saved to {csvFilePath}");

        GaussianEliminationSolver solver = new
GaussianEliminationSolver(equation);
        solver.Solve();
        equation.PrintSolution();
    }

    static void LoadValuesFromCSVCalculateSolutionAndPrint() {
        Console.Write("Enter the value for SIZE: ");
        SIZE = int.Parse(Console.ReadLine()!);

        Console.Write("Enter the value for MAXTHREADS: ");
        MAXTHREADS = int.Parse(Console.ReadLine()!);

        LinearEquation equation = new LinearEquation(SIZE);
        string csvFilePath = "equation.csv";
```

```
        equation.LoadEquationFromCSV(csvFilePath);
        Console.WriteLine("\nLoaded Equation:");
        equation.PrintEquation();

        GaussianEliminationSolver solver = new
GaussianEliminationSolver(equation);
        solver.Solve();
        equation.PrintSolution();
    }

    static void GenerateValuesFromCSVNeuralNetwork() {
        Console.Write("Enter the value for SIZE: ");
        SIZE = int.Parse(Console.ReadLine()!);

        Console.Write("Enter the value for MAXTHREADS: ");
        MAXTHREADS = int.Parse(Console.ReadLine()!);

        LinearEquation equation = new LinearEquation(SIZE);
        string csvFilePath = "equation.csv";
        equation.SaveDataFromCSV(csvFilePath);
        Console.WriteLine("\nLoaded Equation:");
        equation.PrintEquation();

        GaussianEliminationSolver solver = new
GaussianEliminationSolver(equation);
        solver.SolveWithNeuralNetwork();
        equation.PrintSolution();
    }

    static void LoadValuesFromCSVNeuralNetwork() {
        Console.Write("Enter the value for SIZE: ");
        SIZE = int.Parse(Console.ReadLine()!);

        Console.Write("Enter the value for MAXTHREADS: ");
        MAXTHREADS = int.Parse(Console.ReadLine()!);
```

```
        LinearEquation equation = new LinearEquation(SIZE);
        string csvFilePath = "equation.csv";
        equation.LoadDataFromCSV(csvFilePath);
        Console.WriteLine("\nLoaded Equation:");
        equation.PrintEquation();

        GaussianEliminationSolver solver = new
GaussianEliminationSolver(equation);
        solver.SolveWithNeuralNetwork();
        equation.PrintSolution();
    }
}
```