Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Штучного інтелекту
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
## Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Бібліотека глибокого навчання мовою Scala з використанням CUDA
(тема)

Виконав:
здобувач четвертого року навчання,
групи ІТШІ-21-2
Мельнічук Андрій
(власне ім'я, прізвище)
Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Штучний інтелект
(повна назва освітньої програми)
Керівник ст.викл. В'ячеслав Гребенюк
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ШІ ________________ Олег ЗОЛОТУХІН
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Штучного інтелекту

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Штучний інтелект
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«_____» __________ 20 ___ р.

# ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Мельнічуку Андрію Євгеновичу
(прізвище, ім'я, по батькові)

1. Тема роботи Бібліотека глибокого навчання мовою Scala з використанням CUDA

затверджена наказом університету від 19 травня 20 25 р. № 378Ст
2. Термін подання студентом роботи до екзаменаційної комісії 17 червня 20 25 р.
3. Вихідні дані до роботи Науково-технічні публікації, дані Інтернет джерел, PyTorch документація

4. Перелік питань, що потрібно опрацювати в роботі
1) Аналіз предметної галузі
2) Теоретичне дослідження
3) Розробка бібліотеки
4) Експериментальне дослідження бібліотеки

**КАЛЕНДАРНИЙ ПЛАН**

| № | Назва етапів роботи | Строк / терміни виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Отримання завдання на кваліфікаційну роботу | 19.05.2025 | виконано |
| 2 | Аналіз предметної галузі | 20.05.2025 | виконано |
| 3 | Теоретичне дослідження | 21.05.2025 | виконано |
| 4 | Розробка бібліотеки | 25.05.2025 | виконано |
| 5 | Експериментальне дослідження бібліотеки | 26.05.2025 | виконано |
| 6 | Написання пояснювальної записки | 30.05.2025 | виконано |
| 7 | Перевірка на академічний плагіат | 31.05.2025 | виконано |
| 8 | Нормконтроль | 02.06.2025 | виконано |
| 9 | Підготовка презентації та доповіді | 03.06.2025 | виконано |
| 10 | Попередній захист | 04.06.2025 | виконано |
| 11 | Рецензування | 05.06.2025 | виконано |
| 12 | Захист перед ЕК | 17.06.2025 | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Дата видачі завдання 19 травня 20 25 р.

Здобувач ______________________________
(підпис)

Керівник роботи ______________________________ ст.викл. В'ячеслав Гребенюк
(підпис) (посада, власне ім'я, прізвище)

# РЕФЕРАТ

Пояснювальна записка: 62 с., 13 рис., 5 табл., 1 дод., 20 джерел.

БЛОК ПОТОКІВ, ГЛИБОКЕ НАВЧАННЯ, ПОТОК, СІТКА, ТЕНЗОР, ЯДРО, CUDA, JCUDA, SCALA.

Об’єкт дослідження – процес та засоби навчання нейронних мереж на різних видах обчислювальних пристроїв.

Предмет дослідження – бібліотека для навчання нейронних мереж з використанням графічних процесорів, що підтримують CUDA.

Мета роботи – створення бібліотеки глибокого навчання мовою програмування Scala, з використанням технології CUDA, для пришвидшення обчислювань, шляхом використанням паралельних обчислень.

Методи дослідження – теоретичний (збір та структуризація теоретичного матеріалу), експериментальний (програмна реалізація прототипу бібліотеки). Методи розробки базуються на технологіях Scala та CUDA.

У результаті роботи було проведено теоретичне дослідження матричних та тензорних операцій, шару прямого поширення, популярних функцій активації та оптимізаторів. Реалізовано прототип бібліотеки глибокого навчання з використанням технології CUDA для прискорення обчислень. Досліджено швидкість виконання операцій на центральному та графічному процесорі. Визначено різницю у швидкості роботи, а також її причини, між розробленою бібліотекою та іншими реалізаціями.

# ABSTRACT

Bachelor's thesis contains: 62 pp., 13 fig., 5 tabl., 1 ann., 20 references.

CUDA, DEEP LEARNING, GRID, JCUDA, KERNEL, SCALA, TENSOR, THREAD BLOCKS, THREADS.

Object of study: the process and means of training neural networks on various types of computational devices.

Subject of study: a library for training neural networks using CUDA-enabled graphics processing units.

Aim of the work: to create a deep learning library in the Scala programming language, utilizing CUDA technology to accelerate computations through the application of parallel computing.

Research methods: theoretical (collection and structuring of theoretical material), experimental (software implementation of the library prototype). The development methods are based on Scala and CUDA technologies.

As a result of this work, a theoretical study of matrix and tensor operations, the feedforward layer, popular activation functions, and optimizers was conducted. A prototype of a deep learning library utilizing CUDA technology for computational acceleration was implemented. The execution speed of operations on central and graphics processing units was investigated. The difference in operational speed and its causes between the developed library and other implementations were determined.

# ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів ........ 7
Вступ ........ 8
1 Аналіз предметної галузі ........ 9
1.1 Опис проблеми ........ 9
1.2 Актуальність дослідження ........ 11
1.3 Огляд існуючих рішень ........ 13
1.4 Постановка задачі ........ 18
2 Теоретичні дослідження ........ 20
2.1 Автоматичний пошук градієнту ........ 20
2.2 Основні тензорні операції ........ 22
2.3 Шари нейронних мереж ........ 25
2.4 Функції активації ........ 27
2.5 Функції втрат ........ 29
2.6 Оптимізатори ........ 31
3 Розробка бібліотеки ........ 33
3.1 Основні використані бібліотеки ........ 33
3.2 Огляд створених сутностей ........ 35
3.3 Алгоритм зворотного поширення помилки ........ 41
4 Експеріментальні дослідження бібліотеки ........ 43
4.1 Навчання моделі на підготовлених даних ........ 43
4.2 Аналіз швидкості обчислення на різних процесорах ........ 50
4.3 Порівняння реалізації та pyTorch ........ 54
Висновки ........ 59
Перелік джерел посилання ........ 60
Додаток А Відомість кваліфікаційної роботи ........ 62

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

BLAS – Basic Linear Algebra Subprograms – базові підпрограми лінійної алгебри;

CPU – Central Processing Unit – центральний процесор;

CUDA – Compute Unified Device Architecture – єдина архітектура обчислювальних пристроїв;

GPU – Graphics Processing Unit – графічний процесор;

JVM – Java Virtual Machine – віртуальна машина Java.

# ВСТУП

Нейронні мережі стали невід’ємною частиною сучасного світу. Їх використовують у медицині для діагностики, у фінансовій сфері для прогнозування цін на товари та інші економічні показники, у наукових дослідженнях, а також у безлічі інших галузей, де необхідне розпізнавання образів, обробка природної мови або прийняття рішень на основі великих обсягів даних.

Незважаючи на архітектурну оптимізацію, процес навчання нейронних мереж залишається вкрай ресурсоємним. Саме тому велике значення має оптимізація обчислювального процесу. Більшість операцій які виконуються під час навчання мережі є матричними або тензорними операціями, також, зазвичай, результат елементів тензорів не залежить від інших елементів результату, через що розпаралелювання обчислень є очевидним та правильним кроком. Процесори у середньому мають тридцять два потоки, коли графічні процесори можуть мати сотні, або тисячі потоків. Через це використання графічних процесорів стало ключовим фактором успішного поширення нейронних мереж. Графічні карти від NVIDIA, які є найпопулярнішими у наш час, надають розробникам платформу CUDA для використання обчислювача у вирішенні задач, які потребують великої паралельності, через що використання цієї платформи є доцільним і актуальним.

У рамках цієї роботи досліджуються основні блоки нейронних мереж, такі як шари та функції активації, досліджуються функції витрат та оптимізатори, визначаються основні проблеми та створюється прототип бібліотеки глибокого навчання мовою Scala з використанням технології CUDA.

Отже, розроблена в рамках цієї роботи бібліотека може бути використана для ефективного та швидкого навчання та використання нейронних мереж.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

## 1.1 Опис проблеми

У наш час нейроні мережі набули широкого поширення та стали важливим інструментом у вирішенні багатьох різних задач. У зв'язку з цим потреба в їх створенні та навчанні виникає все частіше. Однак реалізація мережі з нуля є складним та довгим процесом, оскільки вимагає програмування прямого та зворотного проходу, пошуку градієнту, та створення алгоритмів оптимізації. Для спрощення процесу створення мереж були розроблені бібліотеки глибокого навчання, які приховують складну логіку обчислень, та надають простий інтерфейс для проектування, навчання та разгортання мереж. Бібліотеки надають людям які не мають знань з того, як математично працюють нейронні мережі можливість їх проектувати, що знижує час та гроші необхідні для навчання такого працівника, через що поширюється кількість людей, що можуть проектувати нейроні мережі.

Бібліотека глибокого навчання має об'єднувати зручний інтерфейс та оптимізовані обчислювальні алгоритми. З кожним роком нейронні мережі стають глибшими – збільшується кількість шарів, складність архітектур і водночас – обчислювальні вимоги. Основна мета таких змін – досягти більшої точності, кращої узагальненості та ефективнішого навчання на великих наборах даних. Проте з ростом глибини зростає і обчислювальна складність моделі, що обумовлює зростання часу навчання та вимог до обчислювальних ресурсів. Для чисельної оцінки кількості операції використовується міра FLOP (floating point operations), яку кількість операції із плаваючою точкою необхідно зробити для отримання результату.

Яскравим прикладом еволюції архітектур глибокого навчання є порівняння моделей VGG16, ResNet110, ViT-B/16 [1], [2]. Як видно з таблиці 1.1, попри те, що ResNet110 має майже в 6,8 разів більше шарів,

кількість FLOP у нього майже вдвічі менша, що стало можливим завдяки застосуванню пропускних зв'язків та оптимізації структури. Хоча ResNet100 є прикладом еволюції архітектури, більш точною у задачах класифікації є архітектура ViT. Ця архітектура збільшила кількість обчислень, та зменшила кількість параметрів у порівнянні з VGG16.

Таблиця 1.1 – Порівняння архітектури нейронних мереж

| Архітектура | Рік розробки | Кількість параметрів | Кількість FLOP |
|---|---|---|---|
| VGG16 | 2014 | $122 * 10^6$ | $15.3 * 10^9$ |
| ResNet110 | 2015 | $1.7 * 10^6$ | $7.6 * 10^9$ |

Великим кроком у розвитку архітектури нейронних мереж було створення трансформерів. Ця архітектура була створена у 2017 році для задач обробки природньої мови, та спирається на механізм уваги, завдяки якому модель може розуміти контекст.

Популярні нейронні мережі сімейства GPT використовують архітектуру трансформерів, та є прикладом росту мереж у останній час, що можна побачити у таблиці 1.2. Збільшення кількості параметрів визначає зростання кількості обчислень, через що задача пошуку засобів пришвидшення обчислень є дуже актуальною.

Таблиця 1.2 – Порівняння кількості параметрів GPT архітектури

| Архітектура | Рік розробки | Кількість параметрів |
|---|---|---|
| GPT-1 | 2018 | 117M |
| GPT-2 | 2019 | 1.5B |
| GPT-3 | 2020 | 175B |

Одним з способів вирішення цієї задачі є використання графічних процесорів. Причиною цього є особливість розвитку відеокарт як

обчислювачів, що мають на кілька порядків більшу кількість потоків, у порівнянні з процесорами. Найпопулярнішими відеокартами для навчання штучного інтелекту є відеокарти від компанії NVIDIA. Це пов’язано з постійним розвитком відеокарт для цієї задачі. Прикладом цього розвитку є створення тензорних ядр, та підтримка змішаної точності.

## 1.2 Актуальність дослідження

Scala – мультипарадигмова мова програмування, що поєднує властивості об'єктно-орієнтованого та функційного програмування. Вона була створена швейцарським дослідником Мартіном Одерським у 2004 році як відповідь на обмеження традиційних мов, таких як Java, при збереженні сумісності з екосистемою Java Virtual Machine (JVM).

JVM – це віртуальна машина, яка виконує байт-код, згенерований компілятором мови Java або інших мов, що працюють на платформі JVM (наприклад, Scala, Kotlin). Вона забезпечує середовище виконання програм, незалежне від конкретної операційної системи чи апаратного забезпечення. Це надає гарантії того, що програма яка написана на scala запуститься на будь якому пристрої, що має встановлену JVM.

Scala широко використовується в галузі обробки та аналізу великих обсягів даних завдяки такій популярній платформі, як Apache Spark, яка спочатку була реалізована саме на цій мові. Це робить Scala природним вибором для задач, пов’язаних із Data Science, Big Data, машинним навчанням та розподіленими обчисленнями.

Для обчислювальних задач у сфері штучного інтелекту застосовуються різні типи апаратного забезпечення. Найпоширенішими серед них є:

– центральний процесор (CPU) – універсальний пристрій, здатний виконувати широкий спектр завдань. Хоча процесори мають обмежену

кількість ядер (зазвичай від 4 до 32), вони дуже гнучкі та використовуються у фазах підготовки даних і запуску програм;

– графічний процесор (GPU) – пристрій, оптимізований для масивного паралельного виконання обчислень. Сучасні GPU містять сотні або тисячі обчислювальних ядер, які здатні одночасно обробляти великі обсяги даних. Це робить їх ідеальними для задач глибокого навчання, особливо у сферах комп’ютерного зору, обробки природної мови та прогнозної аналітики. Ілюстрацію архітектури GPU можна побачити на рисунку 1.1 [7];

– TPU (Tensor Processing Unit), FPGA (Field-Programmable Gate Array), ASIC (Application-Specific Integrated Circuit) та NPU (Neural Processing Unit) – це спеціалізовані обчислювальні пристрої, розроблені для прискорення окремих типів задач ШІ. Вони забезпечують ще вищу продуктивність у певних випадках, але мають обмежену доступність і часто вимагають особливого середовища або глибоких технічних знань для налаштування.

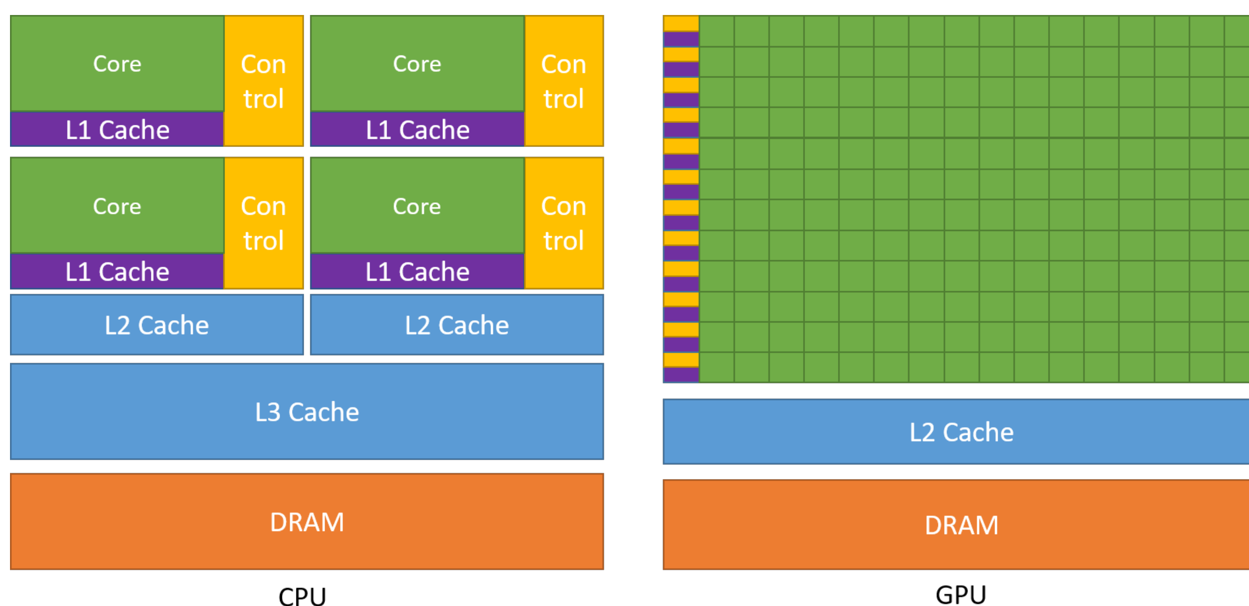


Рисунок 1.1 – Порівняння архітектури CPU та GPU

CUDA (Compute Unified Device Architecture) – архітектура паралельних обчислень і програмна платформа, розроблена компанією

NVIDIA. Вона дає змогу використовувати GPU не лише для візуалізації графіки, а й для виконання загальнообчислювальних задач: від моделювання до глибокого навчання. Завдяки домінуванню NVIDIA на ринку графічних процесорів, підтримка та використання CUDA сьогодні є надзвичайно актуальними для розробників та дослідників у сфері штучного інтелекту.

Одним із ключових понять у CUDA є ядро – функція, яка викликається одночасно великою кількістю паралельних потоків. Кожен потік виконує окремий екземпляр ядра незалежно, що забезпечує ефективну обробку великих обсягів даних. Потоки об'єднуються в блоки, а блоки – у сітки, утворюючи ієрархічну модель обчислень. Під час виконання ядра кожен потік має доступ до інформації про свій ідентифікатор у межах блоку та ідентифікатор блоку у межах сітки, що дає змогу визначати, яку частину даних повинен обробляти саме цей потік.

Окрім правильної організації потоків, важливе значення має розуміння архітектури пам'яті GPU, зокрема: глобальної, локальної, та спільної пам'яті. Усі потоки одного блоку мають доступ до спільної пам'яті, що дозволяє ефективно обмінюватися даними між потоками та оптимізувати використання ресурсів.

Отже актуальність даної роботи зумовлена можливістю розробленої бібліотеки використовувати графічний процесор для пришвидшення обчислень. Крім того, важливою перевагою є можливість використовувати розробку у поєднанні з іншими інструментами екосистеми Java, що робить її придатною для широкого кола прикладних задач

## 1.3 Огляд існуючих рішень

На сьогоднішній день найпопулярнішими бібліотеками глибокого навчання є PyTorch та TensorFlow [4]. Вони активно використовуються як у наукових дослідженнях, так і в промислових застосуваннях. Серед

бібліотек, що мають переважно історичне значення, варто згадати Caffe та Caffe2.

PyTorch – це бібліотека глибокого навчання з відкритим кодом, яка була створена дослідницьким підрозділом компанії Meta AI у 2016 році. Вона є наступницею бібліотеки Torch, яка була написана на мові програмування Lua і активно використовувалась у дослідницьких цілях до появи PyTorch. На відміну від Torch, PyTorch реалізований на мові Python, що зробило його більш доступним для широкої спільноти розробників та дослідників у галузі машинного навчання.

Центральною структурною одиницею PyTorch є тензор – багатовимірний масив (або матриця), що зберігає значення одного типу, зазвичай числові, такі як float32, float64, int32, bool [5]. Тензори PyTorch дуже схожі на масиви з бібліотеки NumPy, але мають важливу перевагу – підтримку обчислень на графічних процесорах (GPU) за допомогою бібліотеки CUDA, що значно прискорює процеси тренування моделей.

Однією з ключових особливостей PyTorch є наявність прапорця requires_grad у тензора. Якщо цей прапорець встановлений у True, PyTorch автоматично відслідковує всі операції, які виконуються над цим тензором, і створює динамічний обчислювальний граф. На відміну від статичних графів, як у TensorFlow 1.x, динамічний граф будується на льоту під час виконання програми, що дозволяє більш гнучко моделювати нейронні мережі, особливо зі складною або змінною структурою.

Завдяки динамічному графу PyTorch підтримує автоматичне диференціювання – можливість автоматично обчислювати градієнти функцій втрат по відношенню до параметрів моделі. Це реалізовано за допомогою модуля Autograd.

Для спрощення побудови моделей, PyTorch включає модуль torch.nn, що містить попередньо визначені нейронні шари прикладом яких є, лінійні шари, згорткові, та функції активації, які автоматично створюють необхідні параметри у вигляді тензорів. Модуль torch.optim містить оптимізатори

прикладами яких є SGD, Adam, RMSprop, які автоматично оновлюють параметри моделей під час навчання відповідно до обчислених градієнтів. Також у бібліотеці передбачено широкий набір функцій втрат, які використовуються для вимірювання похибки між передбаченими та справжніми значеннями під час навчання моделі.

TensorFlow – це одна з найпопулярніших бібліотек для машинного та глибокого навчання, розроблена дослідницькою групою Google Brain і вперше представлена публіці в 2015 році. TensorFlow розроблявся як платформа для розробки моделей штучного інтелекту, яка буде масштабованою, продуктивною і зручною як для досліджень, так і для розгортання у виробництві.

Основою TensorFlow є обчислювальний граф, у якому вузли представляють математичні операції, а ребра – тензори, що передають дані між операціями. У ранніх версіях (до TensorFlow 2.0) бібліотека використовувала статичний граф – граф необхідно було побудувати повністю перед його виконанням. Це ускладнювало налагодження та створення моделей зі змінною структурою. Проте з виходом TensorFlow 2.0 у 2019 році було запроваджено динамічне виконання за замовчуванням, що зробило API набагато гнучкішим і схожим на PyTorch.

TensorFlow, як і PyTorch, оперує тензорами, які можуть виконуватися як на CPU, так і на GPU/TPU. TensorFlow також підтримує автоматичне диференціювання за допомогою системи tf.GradientTape, яка, подібно до Autograd у PyTorch, записує всі операції для подальшого обчислення градієнтів.

Для спрощення створення нейронних мереж, TensorFlow містить високорівневий API – Keras, який є стандартним інтерфейсом моделювання з версії 2.0. За допомогою Keras можна швидко конструювати, тренувати та тестувати моделі. Він включає попередньо реалізовані шари (Dense, Conv2D, LSTM тощо), функції втрат, оптимізатори та метрики.

Apache SINGA – розподілена бібліотека глибокого навчання, розроблена організацією Apache Software Foundation та уперше опублікована у 2015 році. Проект було ініційовано у 2014 році у Національному Університеті Сингапуру.

Головною особливістю цієї бібліотеки є підтримка навчання моделей з використанням паралельних даних на декількох графічних процесорах. Це є дуже важливим через тенденцію підвищення кількості параметрів у моделях, бо під час тренування мережі на відеокарті, усі параметри повинні зберігатися у її пам’яті. Через підвищення кількості вузлів з графічними процесорами зростає обчислювальна потужність та кількість графічної пам’яті, через що виникає можливість навчати модель з більшою кількістю параметрів, та на більшій кількості даних. SIGNA надає можливість використовувати паралельні дані на відеокартах тільки від компанії NVIDIA, та підтримуючих технологію NVIDIA Collective Communications Library.

NVIDIA Collective Communications Library (NCCL) – бібліотека спеціалізована для обміну даними між кількома графічними процесорами, як у одному комп’ютері так і між різними пристроями. Цю бібліотеку використовують й інші бібліотеки глибокого навчання, такі як Chainer, MxNet, PyTorch та TensorFlow.

На сьогоднішній день головною мовою програмування нейронних мереж є Python, через простоту вивчення та швидкість написання коду, та C++, через швидкість роботи. Виключенням з цього правилу є Java з бібліотекою Deeplearning4j. Ця бібліотека може працювати з Apache Spark, та спеціалізується на розгортанні моделі для використання, на відміну від PyTorch бібліотеку складно та не зручно застосовувати для експериментування.

Серед бібліотек, що втратили актуальність, особливо виділяється Caffe. Вона розроблена у Каліфорнійському університеті в Берклі під час створення докторської дисертації. Caffe спеціалізована на вирішенні задач

комп’ютерного зору, особливо задачах класифікації зображень та сегментації зображень.

Головною особливістю та відмінністю Caffe від інших бібліотек глибокого навчання – спосіб створення та визначення архітектури нейронних мереж. Модель, дані для навчання та гіперпараметри вказувалися у файлі з форматом ‘.prototxt’. Такий спосіб визначення ускладнює модифікацію та дослідження архітектури.

Caffe втратила актуальність через наступні причини:

– відсутність підтримки механізмів уваги, трансформерів, та засобів вирішення сучасних задач;

– відсутність підтримки сучасних версій CUDA, через що бібліотека програє у швидкості PyTorch;

– відсутність офіціальних оновлень.

Згодом з’явилась Caffe2 – спроба модернізувати оригінальний Caffe. Однак вона не здобула широкого поширення і з часом була об'єднана з PyTorch у єдину екосистему [6].

Бібліотеки можуть дуже відрізнятися за реалізацією різних функцій, та може статися ситуація, що розробник навчить модель, але йому треба буде перенести її до іншої бібліотеки глибокого навчання. Прикладом цієї ситуації є навчання моделі з використанням PyTorch, та потреба перенести модель до Deeplearning4j, для використання і побудови сервісу. Для того, щоб модель не було потрібно навчати кілька разів компаніями Microsoft та Meta, було розроблено відкритий стандарт ONNX. Перелічені бібліотеки можуть зберігати моделі у цьому форматі, через що розробникам стає легко обмінюватися навченими моделями в незалежності від бібліотеки яку вони використовують. Файли які зберігають модель зберігають граф обчислень, ваги та метадані.

Отже, після розгляду основних бібліотек глибокого навчання можна зробити висновок, що актуальна бібліотека має мати реалізацію тензора та основних операцій на ньому, реалізацію шарів, функцій активацій,

оптимізаторів та засобів автоматичного пошуку градієнту. Усі реалізації мають бути оптимізованими для максимально швидкого виконання.

## 1.4 Постановка задачі

Метою цієї роботи є створення бібліотеки глибокого навчання мовою програмування Scala з використанням архітектури CUDA для ефективного виконання паралельних обчислень на графічних процесорах (GPU). Особливістю цієї роботи є відмова від використання сторонніх бібліотек, що реалізують базову логіку обчислень з матрицями та тензорами. Замість цього передбачається створення власних алгоритмів обчислень. Для реалізації цієї мети необхідно вирішити наступні задачі:

– розробка моделі зберігання даних (тензорів), що дозволить ефективно працювати з багатовимірними масивами даних, які є основною структурною одиницею в нейронних мережах. Це включає визначення типів даних, підтримку різних форматів (CPU/GPU), реалізацію базових математичних операцій над тензорами;

– інтеграція з CUDA через бібліотеку JCUDA для забезпечення доступу до низькорівневих обчислювальних ресурсів GPU. Це дозволить реалізувати обчислення, які виконуються на графічних процесорах із високим рівнем паралелізму;

– розробка механізму автоматичного диференціювання, який дозволить обчислювати градієнти параметрів моделі шляхом побудови обчислювального графа та виконання зворотного проходу. Це критично необхідно для навчання моделей за допомогою градієнтного спуску, або інших алгоритмів;

– створення базових шарів нейронної мережі (наприклад, лінійних, активаційних функцій), які будуть використовувати тензори;

– реалізація функцій втрат, таких як середньоквадратична помилка (MSE), та перехресна ентропія;

– реалізація оптимізаторів, таких як стохастичний градієнтний спуск (SGD), та momentum;

– використання розробленої біблотеки для вирішення прикладних задач, таких як класифікація зображень на підготовлених наборах даних.

– тестування та аналіз продуктивності: порівняння ефективності реалізованої бібліотеки з іншими фреймворками, зокрема у контексті обчислень на GPU.

Таким чином, завданням цієї роботи є створення прототипу ефективної бібліотеки глибокого навчання, яка зможе бути основою для подальшого розвитку використання Scala як мови для програмування глибоких нейронних мереж.

# 2 ТЕОРЕТИЧНІ ДОСЛІДЖЕННЯ

Бібліотека, що розробляється, має можливість виконувати математичні операції між тензорами, має реалізовані шари нейронних мереж, функції втрат та засоби пошуку градієнтів для оптимізації параметрів моделей.

Теоретичне підґрунтя цієї кваліфікаційної роботи є особливо важливим, оскільки всі зазначені компоненти реалізовано без використання сторонніх бібліотек або готових рішень з попередньо реалізованим функціоналом.

## 2.1 Автоматичний пошук градієнту

У 1986 році відбувся прорив у сфері машинного навчання: Девід Румельхарт, Джеффрі Хінтон та Рональд Вільямс опублікували роботу «Learning representations by back-propagating errors», в якій докладно описали алгоритм зворотного поширення помилки (backpropagation) для тренування багатошарових нейронних мереж [8]. Цей алгоритм дав змогу ефективно обчислювати похідні функцій втрат за параметрами моделі, що стало основою для розвитку сучасних методів оптимізації та глибокого навчання.

Однією з ключових концепцій, яка забезпечує працездатність алгоритму зворотного поширення помилки, є використання обчислювальних графів. Обчислювальний граф – це орієнтований ациклічний граф, вузли якого представляють математичні операції або змінні (дані), тоді як ребра задають порядок обчислень і передачі інформації між вузлами. Кожен вузол виконує певну операцію над вхідними значеннями, генеруючи новий вихід, або ж виступає джерелом початкових даних. Процес роботи з обчислювальним графом передбачає два основних етапи: прямий прохід, та зворотній прохід. Прямий прохід (forward

pass) – послідовне обчислення вихідних значень усіх вузлів графа на основі вхідних даних і операцій. Зворотний прохід (backward pass) – поширення градієнтів похідних назад через граф з метою обчислення похідних цільової функції за кожним параметром моделі [8].

Щоб забезпечити коректність обчислення градієнтів під час зворотного проходу, дотримуються таких правил:

– градієнт $g_A$, що надходить у будь-який вузол A, інтерпретується як похідна цільової функції (значення кореня дерева) за значенням у цьому вузлі;

– тензорна розмірність градієнта, який входить у вузол, збігається з розмірністю тензора, що генерується на виході цього вузла;

– у вузлах, які представляють змінні, градієнт передається без змін; у вузли, що відповідають константам, градієнт не передається взагалі;

– початковий градієнт, що надходить і виходить із кореня дерева, дорівнює одиниці або тензору одиниць;

– під час проходження через вузол градієнт розщеплюється між усіма вхідними ребрами і змінюється в залежності від функції, яка створила цей вузол;

– якщо до вузла надходить декілька градієнтів через різні шляхи, їх необхідно підсумувати, що відповідає правилу диференціювання складеної функції

Таким чином, алгоритм зворотного поширення дозволяє обчислювати градієнти всіх параметрів моделі в рамках єдиного проходження через обчислювальний граф, що суттєво підвищує ефективність процесу навчання нейронних мереж. На рисунку 2.1 наведено приклад побудови обчислювального графа і обчислення відповідних градієнтів для функції:

$$z = x^2 + \sin(2y). \quad (2.1)$$

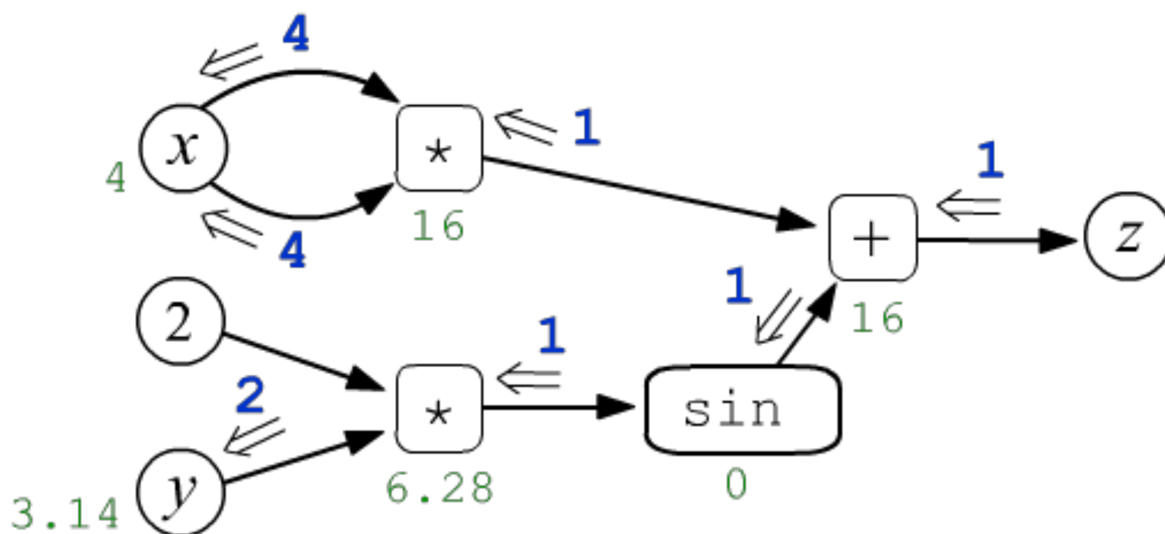


Рисунок 2.1 – Обчислювальний граф

Математичний фундамент алгоритму спирається на правило ланцюгової похідної, яке в скалярному випадку записується у вигляді:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} * \frac{\partial y}{\partial x}. \tag{2.2}$$

У випадку роботи з тензорними величинами застосовується узагальнена версія цього правила:

$$g^{(X)}_{\alpha\beta\ldots} = \frac{\partial L}{\partial X_{\alpha\beta\ldots}} = \sum_{ij,\ldots} \frac{\partial L}{\partial Z_{ij\ldots}} \frac{\partial Z_{ij\ldots}}{\partial X_{\alpha\beta\ldots}} = \sum_{i,j,\ldots} g^{(Z)}_{ij\ldots} \frac{\partial Z_{ij\ldots}}{\partial X_{\alpha\beta\ldots}}. \tag{2.3}$$

## 2.2 Основні тензорні операції

У цьому підрозділі розглянуто базові елементарні тензорні операції, які є фундаментальними компонентами обчислювальних графів та визначають можливість автоматичного обчислення градієнтів. Усі ці операції мають чітко визначені правила диференціювання, що дає змогу коректно поширювати градієнти під час зворотного проходження.

Під час пошуку градієнту, що проходить через операцію, що є поелементною, тобто такою операцією, що виконується незалежно над кожною парою відповідних елементів двох тензорів однакової форми, або над кожним елементом одного тензора, без урахування його положення в структурі, коректним є використання формули ланцюгової похідної. Основними поелементними операціями є сума, різниця, ділення, множення, та зведення у ступінь.

Розглянемо тензори A та B, які мають однакову форму (розмірність по всіх осях), та тензор C, що є результатом операції над A і B. У таблиці 2.1 наведені формули пошуку часткових градієнтів для поелементних бінарних операцій, та градієнтів які передаються до вузлів, що позначенні як $g$.

Таблиця 2.1 – Формулу пошуку похідних

| Опера-ція | $C_{i_1 i_2 \dots i_n}$ | $\frac{\partial C_{i_1 i_2 \dots i_n}}{\partial A_{i_1 i_2 \dots i_n}}$ | $\frac{\partial C_{i_1 i_2 \dots i_n}}{\partial B_{i_1 i_2 \dots i_n}}$ | $g^{(A)}$ | $g^{(B)}$ |
|---|---|---|---|---|---|
| A+B | $A_{i_1 i_2 \dots i_n} + B_{i_1 i_2 \dots i_n}$ | 1 | 1 | $g^{(C)}$ | $g^{(C)}$ |
| A-B | $A_{i_1 i_2 \dots i_n} - B_{i_1 i_2 \dots i_n}$ | 1 | -1 | $g^{(C)}$ | $-g^{(C)}$ |
| A*B | $A_{i_1 i_2 \dots i_n} * B_{i_1 i_2 \dots i_n}$ | $B_{i_1 i_2 \dots i_n}$ | $A_{i_1 i_2 \dots i_n}$ | $B * g^{(C)}$ | $A * g^{(C)}$ |
| A/B | $\frac{A_{i_1 i_2 \dots i_n}}{B_{i_1 i_2 \dots i_n}}$ | $\frac{1}{B_{i_1 i_2 \dots i_n}}$ | $-\frac{A_{i_1 i_2 \dots i_n}}{{B_{i_1 i_2 \dots i_n}}^2}$ | $\frac{g^{(C)}}{B}$ | $-\frac{g^{(C)} A}{B^2}$ |

Операції транспонування, та матричного множення відрізняються тим, що не є поелементними, та визначенні тільки на двомірних тензорах, або тензорах. Транспонування можна визначити формулою:

$$A_{ij}^T = A_{ji}. \quad (2.4)$$

Градієнт, що буде переданий до аргументу операції є транспонованим градієнтом що надійшов до вузлу.

Операція матричного множення відрізняється від усіх приведених операцій тим, що може бути виконана тільки якщо тензори є матрицями, та кількість стовпців першої матриці дорівнює кількості рядків другої матриці. Нехай A – матриця розміру (m × k), а B – матриця розміру (k × n), тоді результат їхнього множення C буде матрицею розміру (m × n), елементи якої обчислюються за формулою:

$$C_{ij} = \sum_{l=1}^{k} A_{il} * B_{lj}. \quad (2.5)$$

Диференціюючи елементи матричного добутку C за елементами матриці A або B, отримаємо формули:

$$\frac{\partial C_{ij}}{\partial A_{\alpha\beta}} = \begin{cases} B_{\beta j}, & i = \alpha \\ 0, & i \neq \alpha \end{cases}, \quad (2.6)$$

$$\frac{\partial C_{ij}}{\partial B_{\alpha\beta}} = \begin{cases} A_{i\alpha}, & j = \beta \\ 0, & j \neq \beta \end{cases}. \quad (2.7)$$

Використовуючи формули 2.6, 2.7 та правило ланцюгової похідної для тензорів отримаємо формули для отримання градієнтів для тензорів A і B:

$$g_{\alpha\beta}^{(A)} = \sum_j g_{\alpha j}^{(C)} \frac{\partial C_{\alpha j}}{\partial A_{\alpha\beta}} = \sum_j g_{\alpha j}^{(C)} B_{\beta j} \Rightarrow g^{(A)} = g^{(C)} ** B^T, \quad (2.8)$$

$$g_{\alpha\beta}^{(B)} = \sum_j g_{i\beta}^{(C)} \frac{\partial C_{i\beta}}{\partial B_{\alpha\beta}} = \sum_j g_{i\beta}^{(C)} A_{i\alpha} \Rightarrow g^{(B)} = A^T g^{(C)}. \quad (2.9)$$

Таким чином, основні тензорні операції, розглянуті вище, є фундаментом для побудови більш складних обчислювальних графів, що використовуються у задачах машинного навчання.

## 2.3 Шари нейронних мереж

У 1943 році в праці «A Logical Calculus of the Ideas Immanent in Nervous Activity» Уоррен МакКаллок та Волтер Піттс запропонували першу спрощену математичну модель нейрона головного мозку – так званий нейрон Маккалоха-Піттса [9]. Ця модель стала основою для побудови сучасних нейронних мереж.

Математично нейрон Маккалоха-Піттса описується наступним чином:

$$u = \sum_{i=1}^{n} w_i x_i + w_0 x_0, \tag{2.10}$$

$$y = f(u), \tag{2.11}$$

де $u$ – лінійна комбінація виходів;

$w_i$ – ваги виходів;

$x_i$, – вхідні сигнали;

y – вихід нейрону;

$f(u)$ – функція активації.

На основі моделі нейрона Маккалоха-Піттса будуються повнозв'язні шари штучних нейронних мереж. Такий шар складається з множини нейронів, кожен з яких з'єднаний із усіма виходами попереднього шару. На рисунку 2.2 представлено схему повнозв'язного шару: зелені точки позначають вхідні дані, кількість яких відповідає числу ознак, сині – скритий шар нейронів, а жовті – вихідний шар нейронів.

Якщо функція активації є тотожним відображенням, тоді глибока нейронна мережа, яка складається з кількох повнозв'язних шарів, еквівалентна одному повнозв'язному шару. Це пояснюється тим, що послідовність лінійних перетворень без нелінійності зводиться до одного

лінійного перетворення. Таким чином, функція активації є причиною здатності мережі моделювати складні, нелінійні залежності у даних.

Під час побудови повнозв'язного шару з m нейронами та n входами формується матриця ваг W, де кожен рядок відповідає ваговим коефіцієнтам окремого нейрона. Додатково створюється вектор зсувів b, що містить зміщення для кожного нейрона в шарі. Математично це представлено наступним чином:

$$u = Wx + b, \quad (2.12)$$

де $x \in R^n$ – вхід;

$W \in R^{m \times n}$– матриця ваг;

$b \in R^m$– матриця ваг.

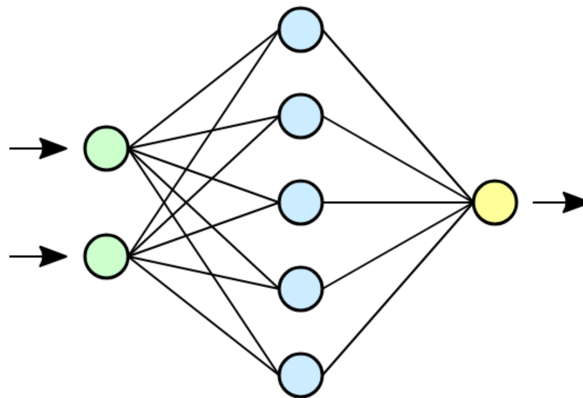

Рисунок 2.2 – Повнозв'язний шар

Однак наведене рівняння застосовне лише до одного вхідного прикладу. Для ефективної обробки відразу декількох прикладів, що є обчислювально ефективною практикою, доцільно використовувати таку формулу:

$$y = xW^T + b^T, \quad (2.13)$$

де $y \in R^{k \times m}$– матриця ваг;

k – кількість вхідних векторів;

$x \in R^{k \times n}$ – вхід;

$W \in R^{m \times n}$– матриця ваг;

$+$ – операція додавання до кожного рядка лівої матриці правий вектор;

$b \in R^{m}$– матриця ваг.

Це представлення дозволяє ефективно реалізовувати обчислення векторизовано, що особливо корисно для застосування методів оптимізації, таких як градієнтний спуск із використанням батчів.

## 2.4 Функції активації

Найпопулярнішою функцією активації у наш час є випрямлений лінійний вузол (ReLU) , що математично представляється наступним чином:

$$ReLU(x) = max(x, 0), \tag{2.14}$$

$$\frac{d}{dx} ReLU(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}. \tag{2.15}$$

Популярність ReLU зумовлена простотою обчислення та здатністю уникати проблеми зникаючого градієнта [10], [11]. Проте, якщо нейрон постійно отримує від’ємні значення на вході, його градієнт дорівнюватиме нулю, що унеможливлює подальше навчання цього нейрона.

До появи ReLU широкого поширення набула сигмоїдна функція активації, яка є монотонною та обмежена на проміжку (0, 1) [12], [13]. Функція визначена формулою:

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{2.16}$$

$$\sigma(x)' = \sigma(x)\big(1 - \sigma(x)\big). \qquad (2.17)$$

Цю функцію доцільно використовувати тоді, коли мережа має повертати значення, що інтерпретуються як ймовірність (впевненість моделі у передбаченні). Недоліками сигмоїди є складність обчислення та схильність до зникання градієнта при великих вхідних значеннях за модулем.

Ще однією популярною функцією активації є гіперболічний тангенс, який є масштабованим варіантом сигмоїди з областю значень у межах $(-1, 1)$ [14]. Залежність між гіперболічним тангенсом та сигмоїдою представляється наступним чином:

$$tanh(x) = 2sigmoid(2x) - 1 = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \qquad (2.18)$$

$$\frac{d}{dx} tanh(x) = 1 - tanh^{2}(x). \qquad (2.19)$$

Хоча гіперболічний тангенс має центр симетрії в нулі, на відміну від сигмоїди, він успадковує її недоліки, зокрема проблему зникання градієнта.

Функція softmax є спеціальним видом функції активації, яка використовується переважно на вихідному шарі нейронної мережі для задач багатокласової класифікації. Вона перетворює вектор дійсних чисел на вектор ймовірностей, які сумуються до одиниці, що дозволяє інтерпретувати вихід мережі як розподіл ймовірностей по класах [15]. Функція softmax має модифікацію під назвою стабільний softmax. Результат та градієнт обох функцій є однаковим, але при надто великих значеннях у вхідному вектору, при обчисленні ступеню від e, число може стати надто великим, що приведе до обчислювальної помилки. Функція softmax, та її стабільна версія визначені наступним чином:

$$y_i = softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \qquad (2.20)$$

$$stableSoftmax(z_i) = \frac{e^{z_i - max(z)}}{\sum_{j=1}^{K} e^{z_j - max(z)}}, \qquad (2.21)$$

$$\frac{dy_i}{dz_j} = \begin{cases} y_i(1 - y_i), & i = j \\ -y_i y_j, & i \neq j \end{cases}, \qquad (2.22)$$

де z – вхідний вектор довжиною K.

## 2.5 Функції втрат

Функція втрат є важливим компонентом процесу навчання нейронної мережі, оскільки визначає міру помилки між передбаченням і реальністю. Під час навчання нейронної мережі функція втрат оптимізується, знаходиться мінімум, за допомогою алгоритмів оптимізації. При створенні функції втрат бажано, щоб функція була диференційованою, бо у наш час для навчання використовують алгоритми які оптимізують спираючись на градієнт, та досягання глобального мінімуму, якщо передбачення збігаються з мітками. Функції втрат обирають в залежності від задачі, для регресії, найчастіше, обирають середньоквадратичне відхилення, для класифікації обирають крос-ентропію.

Функція середньоквадратичне відхилення (MSE) має наступну формулу [16]:

$$MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \qquad (2.23)$$

$$\frac{d}{d\hat{y}_i} MSE = \frac{2}{n}(\hat{y}_i - y_i), \qquad (2.24)$$

де n – кількість прикладів;

$y_i$ – істинне значення для i-го елементу;

$\hat{y}_i$ – передбачення моделі.

При повному співпадінні істинних значень та передбачень функція буде дорівнювати нулю. Існує сімейство функції маючих різницю з MSE у тому, що різниця між істиною та передбаченням буде зведена до парного ступеню. Чим більше ступень, тим сильніше функція буде штрафувати за незбіжність істини та передбачення. Недоліком MSE є її чутливість до викидів, оскільки великі помилки мають надмірний вплив на результат.

Ще однією поширеною функцією втрат, яка активно застосовується у задачах регресії, є середня абсолютна помилка (MAE) [17]. Вона оцінює якість передбачень моделі, вимірюючи середню величину абсолютного відхилення передбачених значень від істинних. MAE широко використовується у задачах, де важливо мати інтерпретовану оцінку помилки в тих же одиницях, що й вихідні дані.

Формально функція MAE визначається як:

$$MAE(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|, \tag{2.25}$$

$$\frac{d}{d\hat{y}_i}MAE = \begin{cases} \frac{-1}{n}, \ \hat{y}_i > y_i \\ \frac{1}{n}, \ \hat{y}_i < y_i \\ 0, \ \hat{y}_i = y_i \end{cases}, \tag{2.26}$$

де n – кількість прикладів;

$y_i$ – істинне значення для i-го елементу;

$\hat{y}_i$ – передбачення моделі.

На відміну від MSE, градієнт MAE не є безперервним, оскільки абсолютна функція не має похідної в точці нуля. Недоліком MAE є знижена

швидкість навчання, оскільки градієнт для великої помилки є рівним до градієнту малої помилки.

Функція крос-ентропія використовується у випадку задачі класифікації, як багатоскладової, так і бінарної, та має наступну формулу:

$$Loss = -\sum_{i=1}^{C} y_i log(\hat{y}_i), \tag{2.27}$$

$$\frac{d}{d\hat{y}_i} Loss = -\frac{y_i}{\hat{y}_i}, \tag{2.28}$$

де C – кількість класів;

$y_i$ – істинне значення для і-го елементу, визначене одиницею, якщо приклад і-го класу, та нулем інакше;

$\hat{y}_i$ – передбачення ймовірності приналежності до класу.

Хоча ця функція є найпопулярнішою у задачах класифікації, вона має недолік, вона чутлива до викидів, та є складно інтерпретованою в порівнянні з MSE [18].

## 2.6 Оптимізатори

Для підбору параметрів, що мінімізують функцію витрат використовуються оптимізатори – алгоритми знаходження мінімуму функції. Одними з найпопулярніших та найпростіших оптимізаторів є стохастичний градієнтний спуск та його модифікація з імпульсом.

Градієнтний спуск – чисельний алгоритм оптимізації функції, що полягає у зміні параметрів додаванням протилежного градієнту втрат. Градієнт вказує напрямок найшвидшого зростання функції, через це віднімання градієнту зменшує функцію. Формально це має наступний вигляд:

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t), \quad (2.29)$$

де $\theta_{t+1}$ – оновлені значення параметрів;

$\theta_t$ – попередні значення параметрів;

$\eta$ – додатній коефіцієнт навчання;

$\nabla L(\theta_t)$ – градієнт втрат функції L за параметром $\theta_t$.

Стохастичний градієнтний спуск використовує градієнти від похибок одразу кількох прикладів, що стабілізує градієнт та прискорює оптимізацію, через відсутність необхідності шукати градієнт для кожного випадку окремо.

Оскільки стохастичний спуск може бути повільним, або нестабільним через долини на функції витрат, використовується покращений імпульсом алгоритм. Цей метод накопичує попередні градієнти, що дозволяє проходити долини та локальні мінімуми. Формально це має наступний вигляд:

$$v_{t+1} = \gamma v_t + \eta \nabla L(\theta_t), \quad (2.30)$$

$$\theta_{t+1} = \theta_t - v_{t+1}, \quad (2.31)$$

де $\theta_{t+1}$ – оновлені значення параметрів;

$\theta_t$ – попередні значення параметрів;

$\gamma$ – коефіцієнт імпульсу, що визначає ступінь інертності;

$v_t$ – накопичений градієнт;

$\eta$ – додатній коефіцієнт навчання;

$\nabla L(\theta_t)$ – градієнт втрат функції L за параметром $\theta_t$.

Отже, у ході теоретичного дослідження було досліджено усі функції активації, шари, оптимізатори, для прототипу бібліотеки глибокого навчання.

# 3 РОЗРОБКА БІБЛІОТЕКИ

У межах цього розділу було розроблено прототип бібліотеки глибокого навчання. Для цього було створено систему класів, обрано бібліотеки для надання можливості використання CUDA, реалізовано алгоритм зворотного поширення помилки.

## 3.1 Основні використані бібліотеки

Для реалізації бібліотеки була обрана мова програмування Scala. Для того щоб Scala мала можливість запускати CUDA ядра використано бібліотеку JCuda. Ця бібліотека надає функції з бібліотеки CUDA, прикладом яких є звільнення пам'яті з девайсу, копіювання даних з одного девайсу на інший, та виконання ядр з визначеною кількістю потоків та блоків. Також бібліотека надає можливість використовувати наступні популярні CUDA бібліотеки:

– cuBLAS, бібліотека базових операцій лінійної алгебри;

– cuFFT, бібліотека для виконання швидкого перетворення Фур'є;

– cuRAND, бібліотека для генерації випадкових значень;

– cuSPARSE, бібліотека операцій для розріджених матриць;

– cuSOLVER, бібліотека для розв'язання систем лінійних рівнянь.

Для прискорення обчислень на центральному процесорі використовувалася бібліотека Parallel Collections, яка дозволяє створювати паралельні колекції, головною особливістю яких є можливість виконувати операції, прикладом яких є map, reduce, filter, незалежно на кількох елементах колекції одразу. Проблемою для розробника може стати операція reduce, функція вищого порядку, аргументом якої є функція, що приймає два аргументи типу колекції, та повертає один. На послідовних колекціях ця функція послідовно з перших двох елементів зводить колекцію до одного значення. На паралельних колекціях порядок виконання функції не

визначено, через що результат операції може бути різним при багаторазовому використанню цієї функції. Для забезпечення однакового результату операція має буту асоціативною, тобто результат не має залежати від черговості виконання операції. Сумування Float чисел не є асоціативною операцією, через що точність обчислення спадає, що можна вважати недоліком для використання при реалізації бібліотеки глибокого навчання.

Для тестування коректності обчислень використано бібліотеку ScalaTest. Особливістю цієї бібліотеки є підтримка різних стилів написання тестів. Приклад реалізації тесту наведено у лістингу 3.1.

Лістинг 3.1 – Тестування суми тензорів

```
class TensorMainOperationTest extends AnyFunSuite:
     def tensorEqual(a: Tensor, b: Tensor): Boolean =
          val st1 = a.storage.toCpu.storage
          val st2 = b.storage.toCpu.storage
          val d = 0.0001
          (0 until st1.length)
          .map(i => math.abs(st1(i) - st2(i)) < d).reduce(_
&& _) &&  a.storage.shape == b.storage.shape
     test("+ shoud return the sum of two tensors"){
          val st1 = 0 until 20 map (_.toFloat)
          val st2 = st1 map (_ + 100)
          val shape = Seq(1, 20)

          val a = Tensor(st1, shape)
          val b = Tensor(st2, shape)
          val res = a + b

          val tres = Tensor((0 until 20).map(i => st1(i) +
st2(i)), shape)
          assert(tensorEqual(res, tres))
     }
```

## 3.2 Огляд створених сутностей

У ході реалізації було створенно наступні сутності: Storage, ArrayStorage, CudaStorage, Tensor, GeneralFunction, ReplicatibleFunction, ForwardLayer, ReLU, Sigmoid, Tanh, Sequential, StableSoftmax.

Storage – абстракція що представляє сховище даних, та оголошує основні математичні операції, найменування яких можна побачити у лістингу 3.2. Реалізація має зберігати посилання на дані, та інформацію о структурі у полі shape. Storage є моделлю тензора та операцій на ньому, тож shape визначає розмірність цього тензору з наступними обмеженнями:

– кількість осей має бути більше нуля;

– розмірність осі не може бути нулем, або меншим значенням;

– кількість об'єктів що зберігається має збігатися з кількістю елементів що вказується у розмірності.

Лістинг 3.2 – інтерфейс трейту Storage

```
trait Storage:
     val shape: Seq[Int]
     def +(other: Storage): Storage
     def -(other: Storage): Storage
     def *(other: Storage): Storage
     def /(other: Storage): Storage
     def **(other: Storage): Storage

     def +(alpha: Float): Storage
     def -(alpha: Float): Storage
     def *(alpha: Float): Storage
     def /(alpha: Float): Storage
     def pow(n: Float): Storage
     def unary_- : Storage

     // device change
     def toCpu: ArrayStorage
```

Продовження лістингу 3.2

```
def toCuda: CudaStorage

// reduce
def sum: Storage
def item: Float
def split(dim: Int = 0, size: Int = 1): Seq[Storage]
def sum(axis: Int = 0): Storage

def T: Storage
def reshape(seq: Int*): Storage
def reshape(seq: Iterable[Int]): Storage
def unsqueeze(ax: Int = 0): Storage
def apply(args: Seq[Int | Iterable[Int]]): Storage
def flatten(from: Int = 0, to: Int = shape.length): Storage
def cat(st: Storage, dim: Int = 0): Storage
```

Операції по елементної суми, різниці, добутку, та ділення визначені операціями +, -, *, / відповідно. При визначенні, операції має перевірятися чи однакову кількість даних зберігають сховища, якщо ні, дані, якщо це можливо розширюються до потрібних розмірів, та тільки після цього виконується операція. Алгоритм розширення наступний:

– обирається сховище для розширення – сховище що має меншу кількість елементів;

– якщо обране сховище має кількість осей меншу, то додаються нові з розміром у одиницю;

– починаючи з останньої осі виконуються наступні кроки: якщо значення осі одиниця, то дублюються значення, якщо значення осі відрізняється від значення до якого треба розширити, то розширення неможливе.

Операції +, -, *, / аргументами яких є значення типу Float є операціями додавання, відніманням числа, множеннями та діленням на число. Pow є операцією зведення у ступінь елементів, а не зведення квадратної матриці до ступеню. Unary_- є еквівалентом множення матриці на -1.

Методи toCpu та toCuda призначені для копіювання даних між центральним процесором та графічним без видалення, та обгортання у реалізації сховищ. Якщо цільовий пристрій збігається з поточним, копіювання не здійснюється, і повертається оригінальний об’єкт.

До зменшувальних операції належать sum, sum(axis), item, split(dim, size), apply(args). Метод sum знаходить суму усіх елементів сховища, що корисно при пошуку втрат моделі. Операція item повертає елемент у випадку коли кількість елементів що зберігається дорівнює одиниці, інакше виникає помилка. Метод split(dim, size) розділяє тензор за віссю dim та об’єднує по size частин. З використанням цього методу визначено sum(axis). Ця операція розділяє за віссю сховище та сумує між собою. Метод apply(args) використовується для отримання частин тензору за індиксами. Для обрання розроблено наступну логіку:

– кожен аргумент у списку args відповідає певній осі сховища;

– якщо для певної осі потрібно обрати всі елементи, передається значення -1;

– якщо передано ціле число і, обирається тільки і-ий елемент уздовж відповідної осі;

– якщо передано перелік індексів, то обираються відповідні елменти уздовж осі.

Метод ** та T відповідають операціям матричного множення та транспонуванням відповідно. Ці операції виконується тільки для сховищ з двома осями, тобто матриць. Для виконання множення кількість стовбців першого сховища має дорівнювати кількості рядків другої матриці. При недотриманні цих правил має виникати помилка.

Операції reshape, unsqueeze, flatten єдині методи що не змінюють дані, але змінюють кількість осей або їх розмір. Метод reshape змінює форму сховища, яка обмежена тільки кількістю елементів. Операція unsqueeze збільшує кількість осей додаючи нову вісь розміром 1. Метод flatten зменшуючи кількість осей об'єднуючих їх до однієї.

Метод cat об'єднує сховища за віссю. Умовою для об'єднаня є співпадіння кількості осей та співпадіння розмірів усіх осей окрім осі об'єднання.

Для зручності створення сховищ було створено об'єкт компаньйон, у якому визначені функції fill, ones, zeros, rand, arrange, кожна з яких має неявний строковий аргумент який визначає пристрій зберігання даних. Змінюючи цей аргумент розробник зможе контролювати яким саме процесором буде виконуватися обчислення. Функції ones та zeros мають як аргумент розмір майбутнього сховища та повертають його заповнене одиницями або нулями відповідно. Функція arrange повертає одномірне сховище заповнене значеннями від нуля до n. Для створення сховища заповнене іншими значеннями за якимось правилами, використовується метод fill. Ця функція має аргумент value який передається за ім'ям, через що можливо передати функцію яка кожного разу буде надавати нове значення. Цю особливість використано для створенні методу rand, який повертає сховище визначеної форми, яке заповнене випадковими значеннями.

При реалізації нащадків Storage, при визначенні бінарних операцій, рекомендується перевіряти тип другого операнду. У випадку, якщо він не є об'єктом того ж класу доцільно викликати помилку. Це дозволяє уникнути неявного копіювання даних на різні пристрої, що облегшує розуміння програми та зменшує можливість появи помилок. Реалізаціями Storage є ArrayStorage та CudaStorage.

ArrayStorage зберігає дані у оперативній пам'яті у вигляді масиву чисел з плаваючою точкою. Для підвищення швидкості обчислень, під час

виконання операцій масив перетворюється до паралельної колекції, що дозволяє використовувати многопоточність.

CudaStorage реалізує зберігає даних у відеопам'яті, та призначен для виконання обчислень на графічному процесорі. Особливістю цієї реалізації є те, що збирач смітті не має контролю над пам'яттю графічного процесору, через що її звільнення повинно виконуватися вручну, інакше можливе накопичення неочищених ресурсів, що призводить до фатальної помилки JVM.

Для опису функцій, що мають прямий прохід та зворотній визначено абстракцію GeneralFunction. Вона зберігає аргументи, об'єкти класу Tensor, які були використані при обчисленні конкретної функції, та результат прямого проходу. Також GeneralFunction реалізує функцію що визначає зворотне поширення, аргументами якої є тензор для якого шукається похідна та градієнт від вищої за рівнем функції. Якщо заданий тензор не приймав участі у обчисленні прямого проходу результатом зворотного поширення є сховище тієї ж розмірності заповнене нулями. Таким чином GeneralFunction є ключовим компонентом побудови алгоритму зворотного поширення.

Клас Tensor є високорівневим інтерфейсом для управління даними. Він зберігає дані у об'єкті типу Storage, та надає можливість формування обчислювального графа. Конструктор класу Tensor приймає два параметри: функція що породжує, типу GeneralFunction, та булевий прапорець hasVar, який вказує чи є містить відповідний обчислювальний граф змінні. У даному контексті змінними вважаються тензори які не є константами, та для яких існує можливість обчислення градієнту. При створенні змінної функція що породжує має не мати вхідних аргументів. Важливою властивістю є те, що операції між тензорами, які не мають змінних, не можуть породжувати тензори, що мають змінні. Прикладом побудови операції на тензорах є операція матричного добутку, яка описана у лістингу 3.3.

Лістинг 3.3 – Матричний добуток тензорів

```
def **(this: Tensor, other: Tensor) =
     val a = this
     val b = other
     new Tensor(new GeneralFunction {
          val args: Seq[Tensor] = Seq(a, b)
          val forward = a.storage ** b.storage
          def backward(arg: Tensor, chainGrad: Storage) =
               if forward.shape != chainGrad.shape then
                    throw new Exception()
               if a == arg then chainGrad ** b.storage.T
               else  if  b  ==  arg  then  a.storage.T  **
chainGrad
               else Storage.zeros(arg.storage)
     }, a.hasVar || b.hasVar)
```

Бібліотеки глибокого навчання цього часу мають заздалегідь створені шари нейронних мереж. У PyTorch класом, що представляє шари та функції активацій є Module. Його аналогом у реалізації є абстракція ReplicatibleFunction, яка має наступний інтерфейс:

```
trait ReplicatibleFunction:
     def apply(x: Tensor): Tensor
     def   replicate(grad:   Map[Tensor,   Storage],   opt:
Optimizer): ReplicatibleFunction
```

Функція apply використовується для прямого поширення через шар та отримання нового обчислювального графу. При реалізації нащадків цього трейту є важливим пам’ятати, що граф має бути розширеним, а не переписаним. Функція replicate використовуються для оновлення ваг за правилом переданого оптимізатору та створенням нового об’єкта класу.

Для моделювання оптимізаторів створюється функція, аргументами якої є ваги шару та градієнт, а результатом сховище градієнт. Найпростішим прикладом оптимізатору є стохастичний градієнтний спуск визначений наступним чином:

```
def SGD(tau: Float): (Storage, Storage) => Storage=
    (w, grad) => w - grad * tau
```

Нащадками ReplicatibleFunction є ForwardLayer, ReLU, Sigmoid, Tanh, Sequential, StableSoftmax. ForwardLayer – повнозв'язний шар прямого поширення, ReLU, Sigmoid, Tanh – однойменні функції активації. Для облегшення проектування моделей створено клас Sequential. При створенні цей клас як аргумент приймає послідовність шарів, та при використанні методу apply передає на вхід наступному шару результат попереднєго.

### 3.3 Алгоритм зворотного поширення помилки

Алгоритм зворотного поширення помилки є однією з найважливіших частин для будь якої бібліотеки глибокого навчання. Його реалізація може залежати від бібліотеки до бібліотеки. У PyTorch тензори мають метод backward, після виклику якого кожна змінна у полі grad отримає значення градієнту. Реалізація у даній роботі дотримується принципів незмінності, через що схожий алгоритм не побудовано.

Storage, Tensor, GeneralFunction – сутності яких достатньо для побудови алгоритму зворотного поширення. Результатом цього алгоритму є відображення, яке зберігає усі тензори змінні та відповідні сховища градієнти. Градієнти зберігаються саме у вигляду сховищ, через відсутність необхідності зберігати обчислювальний граф градієнту. Пошук градієнту є рекурсивним та працює наступним чином:

– якщо функція що породжує тензор не має аргументів, та булевий прапорець hasVar вказує на те, що тензор є змінною, повертається пара тензор та накоплений градієнт;

– якщо значення прапорця не є істиною, то нічого не повертається;

– якщо пункти 1 та 2 не виконані, то з аргументу функції що породжується обираються усі тензори які у графі мають змінні та алгоритм повторюється.

Початковим градієнтом є сховище одиниць розмірність якого співпадає з розмірністю тензору на якому викликано пошук градієнту. Для прискорення алгоритму фільтрація та рекурсивний запуск є паралельними. Реалізація пошуку градієнту на графі представлено у лістингу 3.4.

Лістинг 3.4 – Алгоритм зворотного поширення

```
def gradientSearch(t: Tensor): Map[Tensor, Storage] =
  def helper(t: Tensor, accGrad: Storage): List[(Tensor, Storage)] =
    if t.origin.args.isEmpty && t.hasVar then List((t, accGrad))
    else if !t.hasVar then List()
    else t.origin.args.par.filter(_.hasVar).flatMap(v => helper(v, t.origin.backward(v, accGrad))).toList
  helper(t, Storage.ones(t.storage)) .groupMapReduce(_._1)(_._2)(_ + _)
```

Через відсутність розповсюдження градієнту у підграфах, які не мають змінних, та використання паралельної рекурсії алгоритм є швидким. Проблемою використання цього алгоритму є його паралельність. Якщо розмірність градієнтів буде надто велика, або операції будуть використовувати занадто багато пам’яті, може статися помилка.

Цей спосіб пошуку градієнту не є єдиним можливим алгоритмом. Другим алгоритмом є попередній пошук усіх можливих шляхів до змінних, групування шляхів за змінною, паралельний прохід та накопичення градієнтів, та сума градієнтів у групах. Проблемою цього рішення багаторазове обчислення однакових шляхів, через що алгоритм не є оптимальним.

# 4 ЕКСПЕРІМЕНТАЛЬНІ ДОСЛІДЖЕННЯ БІБЛІОТЕКИ

Для оцінки ефективності розробленого рішення бібліотеку використано для навчання мереж на підготовлених даних, та порівняно обчислювальну ефективність з іншими реалізаціями.

## 4.1 Навчання моделі на підготовлених даних

Метою цієї роботи є створення бібліотеки для навчання нейронних мереж, отже перевірка її можливостей є важливим та необхідним кроком. Для навчання моделі дані мають бути підготовленні та попередньо оброблені, що не входить до задач цієї роботи. Для навчання використано наступні набори даних: MNIST, FashionMNIST, та іриси Фішера.

Іриси Фішера – класичний набір даних для задачі класифікації, що було використано у 1936 році Рональдом Фішером, англійським статистиком, для демонстрації розробленого дискримінантного аналізу. Дані було зібрано американським ботаніком Едгаром Андерсоном. Набір даних має 150 прикладів ірисів, трьох видів, з наступними характеристиками:

– довжина зовнішньої частки (sepal length);

– ширина зовнішньої частки оцвітини (sepal width);

– довжина внутрішньої частки оцвітини (petal length);

– ширина внутрішньої частки оцвітини (petal width).

Набір даних є повністю збалансованим, кожний клас має рівну кількість прикладів. Класифікація ірисів Фішера є однією з найпростіших задач машинного навчання, через що використання нейронних мереж є занадто складним. Після навчання мережі на цьому наборі даних мережа не має переваг у порівнянні з іншими методами машинного навчання, прикладами яких є логістична регресія, та дерево рішень.

Набір даних MNIST (Modified National Institute of Standards and Technology) є еталонною колекцією зображень рукописних цифр, розміром 28x28 пікселів, яка була сформована у 1988 році. Його створення ініціював Ян Лекун у співпраці з Корінною Кортес та Крістофером Бьорджесом.

Цей набір даних містить 60 000 зображень для навчання та 10 000 зображень для тестування, що забезпечує достатній обсяг даних для навчання та оцінювання алгоритмів класифікації. Усі зображення передані у одному каналі, тобто є сірими. Через велику якість даних та загальнодоступність MNIST став одним з найпопулярніших наборів даних у комп’ютерному зорі.

Для першого експерименту було обрано розмір пакета в 100 зображень, алгоритм оптимізації – стохастичний градієнтний спуск зі швидкістю навчання 0.001. функція втрат – крос-ентропія, а також встановлено кількість епох 5.

Для порівняння були обрані наступні архітектури нейронних мереж:

– вхід (784), Linear(100), ReLU, Linear(10), softmax;

– вхід (784), Linear(100), ReLU, Linear(100), ReLU, Linear(10), softmax;

– вхід (784), Linear(256), ReLU, Linear(10), softmax;

– вхід (784), Linear(256), ReLU, Linear(256), ReLU, Linear(10), softmax;

Усі моделі мають вхідний шар розміром 784, що відповідає кількості пікселів в одному зображенні, та вихідний шар з 10 нейронами, що відповідає кількості класів. На відміну від бібліотеки pyTorch останнім шаром має бути явно застосовано softmax функцію.

Результат цього експерименту наведено у таблиці 4.1, З аналізу таблиці можна зробити висновок, що моделі з меншою кількістю шарів, при однаковій кількості епох демонструють вищу точність або швидше навчаються ніж у порівнянні з більш глибшими архітектурами.

На рисунку 4.1 зображено динаміку зміни значення функції втрат у залежності від номера епохи для визначених моделей. З графіка можна зробити висновок, що моделі з проміжною кількість нейронів 100, мають

більшу помилку ніж моделі з 256 нейронами. Отримані результати можна використовувати для побудови покращених моделей, збільшуючи кількість нейронів у прихованих шарах. Оскільки точність класифікації тісно пов'язана зі значенням функцій втрат, у даному аналізі залежність точності від кількості епох не розглядається.

Таблиця 4.1 – Результат навчання на MNIST

| Номер моделі | Втрати | Точність |
|---|---|---|
| 1 | 1.6424 | 0.7508 |
| 2 | 2.1969 | 0.4973 |
| 3 | 1.5557 | 0.7398 |
| 4 | 2.1538 | 0.6018 |

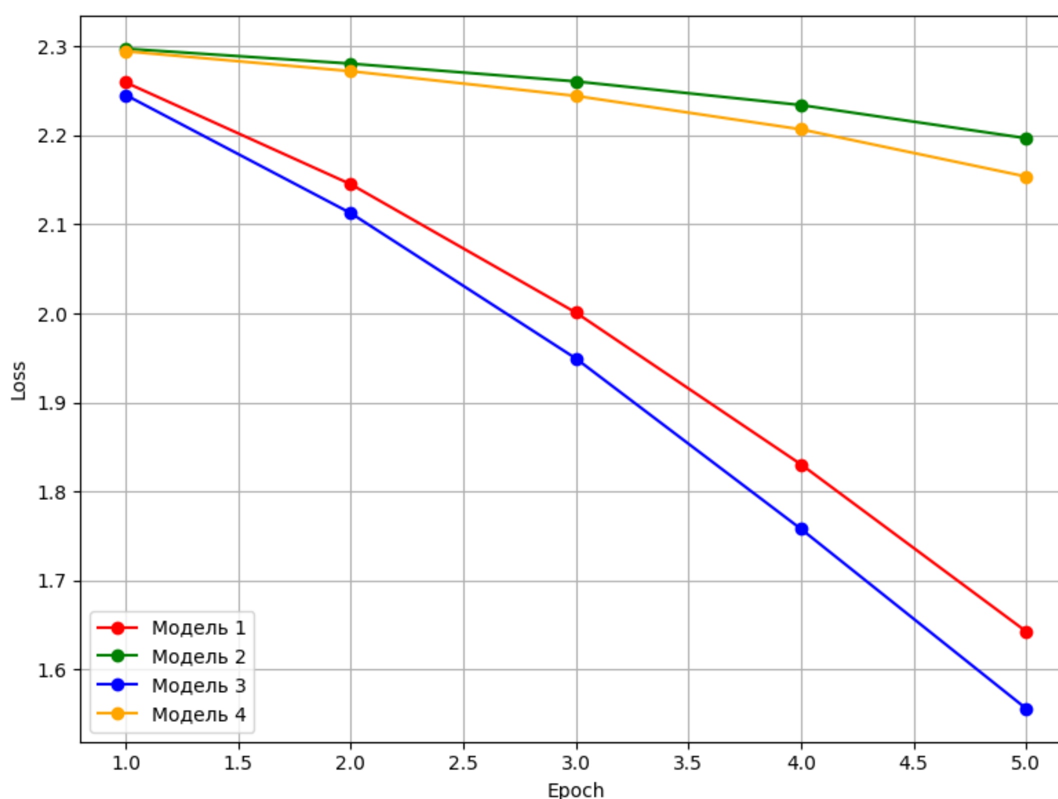


Рисунок 4.1 – Навчання моделей на наборі MNIST

Точність нейронних мереж також значною мірою залежить від кількості епох навчання. У рамках попереднього експерименту було обрано

лише п’ять епох, що є недостатнім для повноцінного навчання моделі. Зокрема, моделі зі складнішою архітектурою потребують біль тривалого часу на зменшення функції втрат.

Для перевірки цього припущення було проведенно додатковий експеримент на прикладі третьої моделі, у якому кількість епох збільшено до 50. Результатом є підвищення точності до 90%, та зменшення помилки до 0.35. Динаміка навчання наведена у рисунку 4.2.

Отже з цих двох експериментів можна зробити висновок, що на наборі MNIST демонструють кращі результати, моделі з більшою кількістю нейронів у шарах показують нижчі втрати, та при збільшенні кількості епох значно підвищується точність моделі. Використання повнозв’язних шарів для вирішення цієї задачі достатньо, щоб отримати гарний результат точності.

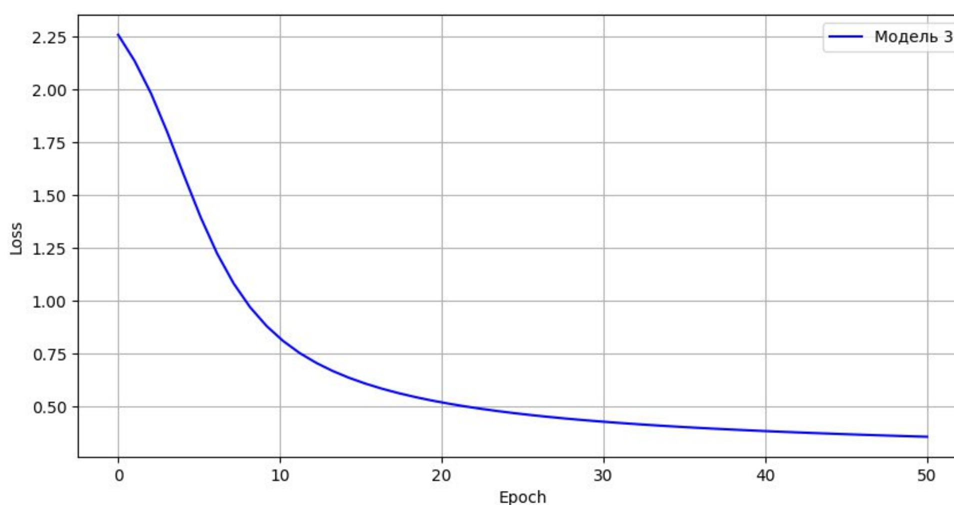


Рисунок 4.2 – Навчання моделі на великій кількості епох

Fashion MNIST – це набір зображень, що широко використовується як альтернатива MNIST для задач класифікації в задачах комп’ютерного зору. Він був створений з метою ускладнення задачі для алгоритмів глибокого навчання. Набір містить 60 000 зображень для навчання та 10 000 зображень для тесту та розподілений на 10 класів. Усі зображення мають розмір 28x28 в один канал, тобто є сірими.

Не зважаючи на те, що Fashion MNIST має ту ж структуру даних, що і MNIST, він є складнішим для отримання високого результату. Це пов’язано з тим, що зображення одягу у одному класі відрізаються сильніше ніж цифри у один від одного у одному класі.

Оскільки Fashion MNIST має ту ж структуру даних, що і MNIST, для експериментів були використані ті самі архітектури що і раніше. Це дозволить побачити різницю між складністю цих наборів даних, та порівняти поведінку бібліотеки під час навчання моделей на різних даних. Оптимізатор, гіперпараметри, та розмір пакету не було змінено.

Після проведення експерименту на 5 епохі отримано, що результат майже не відрізняється, окрім зниженої точності та близьких втрат, що наведено у таблиці 4.2.

Таблиця 4.2 – Результат навчання на Fashion MNIST

| Номер моделі | Втрати | Точність |
|---|---|---|
| 1 | 1.2256 | 0.6553 |
| 2 | 1.8915 | 0.5551 |
| 3 | 1.1988 | 0.6568 |
| 4 | 1.7204 | 0.5327 |

На рисунку 4.3 подано графік зміни значення функції втрат у залежності від номера епохи для чотирьох визначених моделей. Аналіз графіка показує, що зміна тенденції майже відсутня, але можна побачити, що є вірогідність пересікання графіків моделей 1 та 3 з моделями 2 та 4, при збільшенні кількості епох. Це є причиною для проведення наступного експерименту: навчанню усіх моделей протягом 50 епох.

У результаті навчання всіх чотирьох моделей протягом 50 епох було побудовано графік зміни функції втрат, наведений на рисунку 4.4. З аналізу графіку виходить, що лінії втрат для всіх моделей не перетинаються протягом усього періоду навчання, що свідчить про сталість ранжування

моделей за точністю в межах цього експерименту. Починаючи приблизно з 40-ї епохи, криві втрат моделей наближаються одна до іншої, зі значенням похибки приблизно 0.6. Також виходячи з графіку навчання після 40-ї епохи є не доречним через відсутність значного покращення точності.

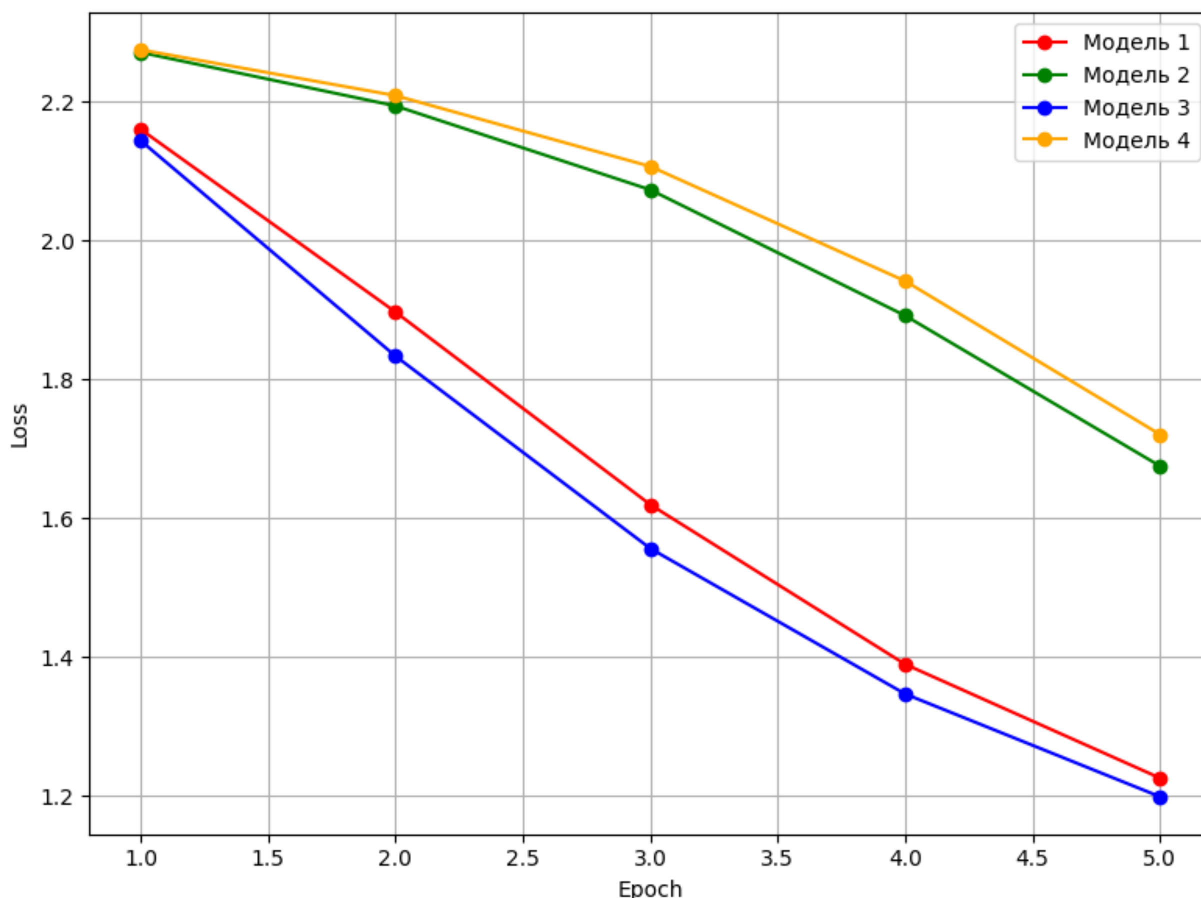


Рисунок 4.3 – Навчання моделей на наборі Fashion MNIST

Отже у ході усіх експериментів з навчанням нейронних мереж, що мають різну архітектуру на різних даних, доведено можливість використання цієї бібліотеки для отримання працюючої моделі для задач глибокого навчання.

Під час моделювання експерименту, через нестачу ресурсів експеримент переривався, щоб запобігти втраті даних, після кожної епохи навчання ваги мережі зберігалися, та після виснаження графічної пам’яті програма перезапускалася та зчитувала збережені ваги. Параметри моделі

зберігалися у спеціальній директорії, кожний набір параметрів зберігався у своєму файлі, назва якого залежала від типу параметру, та номеру шару у моделі. Код для зберігання наведено у лістингу 4.1.

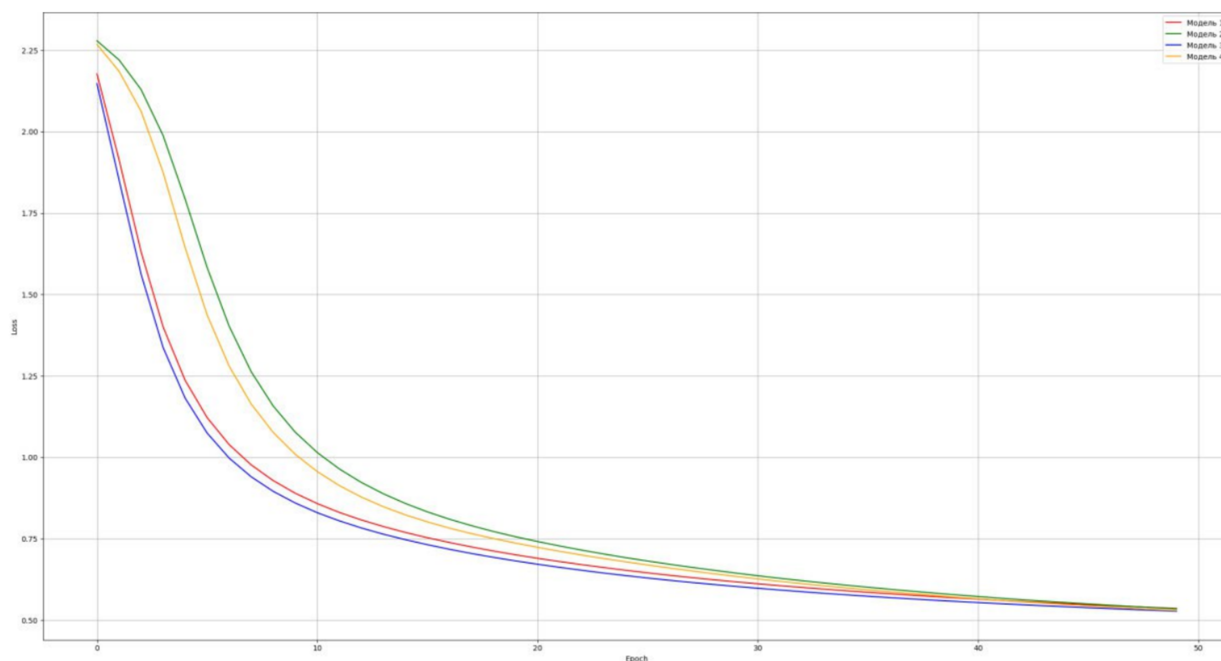


Рисунок 4.4 – Навчання моделей на наборі Fashion MNIST протягом 50 епох

Лістинг 4.1 – Функція зберігання параметрів моделі

```
def dataSave(model: Sequential)  =
  for
    i <- 0 until model.layers.length
  do
    import scala.util.Using
    import java.io._
    model.layers(i) match
      case l: ForwardLayer =>
        Using(new PrintWriter(s"weight_save/w$i")){
          writer => writer.print(
            l.w.storage.toCpu.storage.par
            .map(_.toString()).toList.mkString(",")
          )
        }
```

Продовження лістингу 4.1

```
        Using(new PrintWriter(s"weight_save/b$i")){
          writer => writer.print(
            l.b.storage.toCpu.storage.par
            .map(_.toString()).toList.mkString(",")
          )
        }
      case _ => ()
```

Для підвищення точності розглянутих моделей можливо використовувати більш складні алгоритми оптимізації, прикладом яких є алгоритм Adam або стохастичний градієнтний спуск з імпульсом. При підборі правильних гіперпарметрів оптимізаторів похибка може зменшитись.

Іншим способом покращення є зміна кількості прикладів у пакеті. Велика кількість прикладів у пакеті зменшує вклад кожного, через що послабляється вплив градієнту викиду або аномалії, та навчання моделі стає стабільнішим. Кількість прикладів у пакеті обмежена пам’яттю пристрою, та через особливості архітектури графічного процесору у найкращому випадку має бути студінню двійки.

Третім способом покращення точності є покращення попередньої обробки даних. У ході експериментів дані не були нормалізовані та не були стандартизовані, що може призвести до складності навчання нейронної мережі.

## 4.2 Аналіз швидкості обчислення на різних процесорах

Складність дослідження швидкості виконання операцій полягає у залежності виконання програми від реалізації JVM, операційної системи та апаратних ресурсів.

JVM здатна динамічно оптимізувати виконання коду, що виконується багаторазово. Це дозволяє підвищити швидкість виконання, але цей механізм не підкорюється розробнику. Наслідком виконання цього механізму є невизначеність часу виконання коду.

Під час виконання програми операційна система сприяє на виконання коду наступним чином: вона може зупиняти виконання програми для виконання більш пріоритетних задач. Цей вплив зростає при обмеженні ресурсів, та звеличенню навантажень.

Продуктивність виконання програми може суттєво залежати від тактової частоти процесору, кількості ядр, та архітектури процесору. Наслідком цього є прискорення виконання при підвищенні частоти та кількості ядр.

Для порівняння швидкості операцій на центральному процесорі, та на графічному процесорі обрано наступні операції: додавання, матричне множення, та транспонування. Вибір цих операцій спирається на популярність використання, та важливість для навчання бібліотеки. Через відсутність потреби у пошуку градієнту, усі операції будуть виконані на сховищах.

Операція суми має схожу реалізацію, що й операція різниці, множення, та ділення, через що досліджується лише ця операція. Для аналізу швидкості було створено сховища різного розміру та тисячу раз зібрано інформацію. На рисунку 4.5 зображено залежність швидкості виконання програми від пристрою та розміру сховища. Аналізуючи цей графік визначено, що при сховищах розміру порядку десяти тисяч швидкість суми сховищ близька для обох обчислювачів. Починаючи з четвертого порядку розбіжність швидкості швидко зростає, та для розміру 8 го порядку, час обчислення суми на центральному процесорі може займати хвилини, коли на графічних процесорах мілісекунди. Отже, для обчислення суми двох сховищ розміром до четвертого порядку має сенс

використовувати центральний процесор, інакше використовувати графічний процесор.

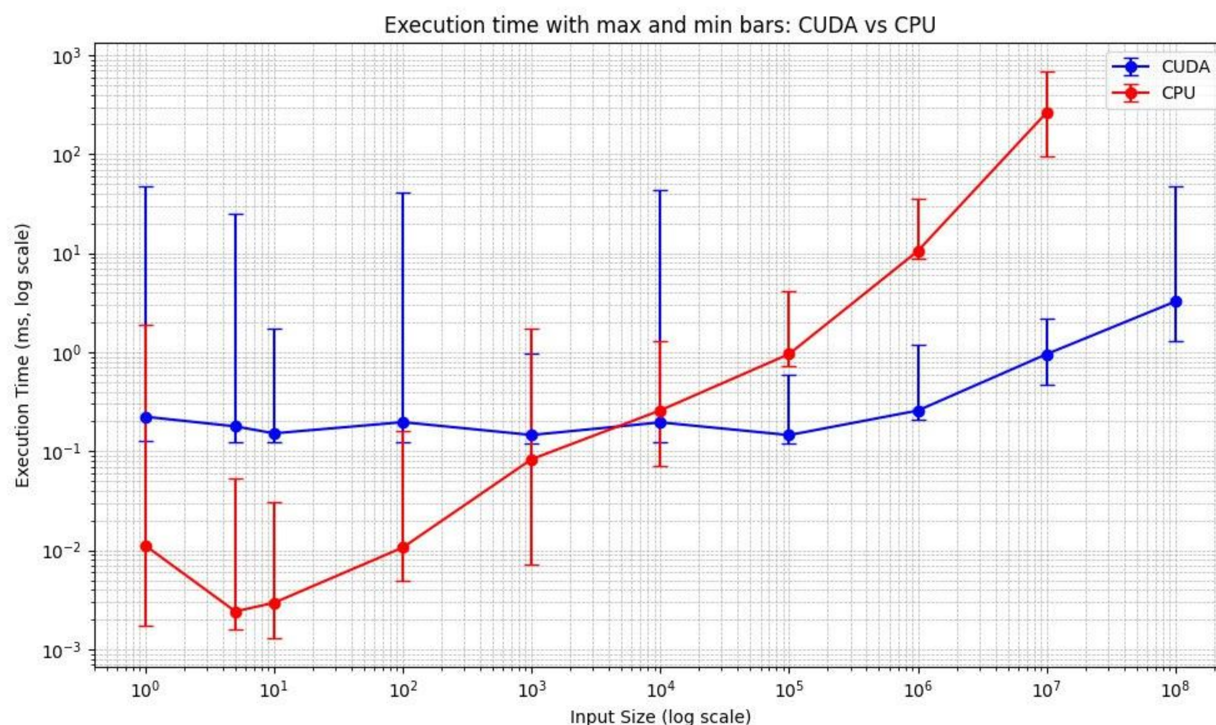


Рисунок 4.5 – Швидкість обчислення суми

Операція матричного множення є однією з найважливіших операцій у обчисленні нейронних мереж. Ця операція використовується для реалізації повно зв’язних шарів, для згорток, та трансформерів, тобто для головних та найпопулярніших шарів нашого часу. Через це швидкість реалізації матричного множення сприяє швидкості навчання та обчислення більшості з нейронних мереж. Проблемою для порівняння швидкості алгоритму є підбір даних. Швидкість обчислення матричного множення залежить від форми матриць, через що було вирішено перевіряти швидкість алгоритму на квадратних матрицях. Результат порівняння обчислення на графічному та центральному процесорі зображено на рисунку 4.6. Аналіз графіку вказує на швидкий ріст розбіжності між часом обчислення результату. При довжині сторони близької до 241 обчислення проходить у середньому 160 мілісекунд, що є дуже великим часом, при використанні у навчанні мереж.

Отже для обчислення матричного множення має бути використано графічний процесор для будь якого розміру матриці.

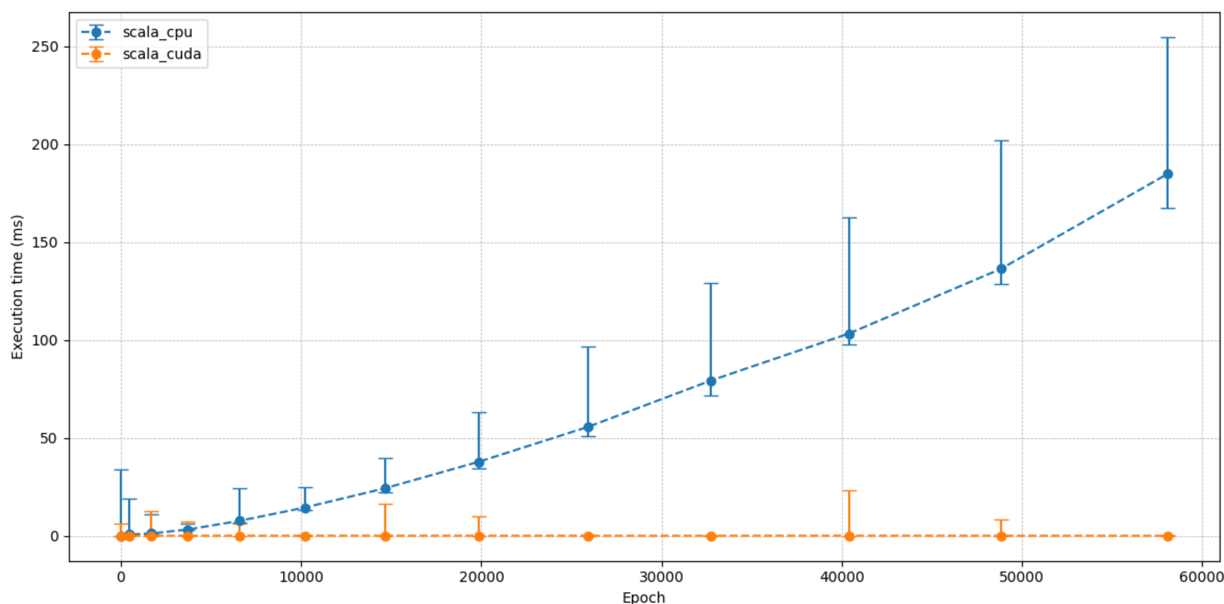


Рисунок 4.6 – Швидкість обчислення матричного множення

Операція транспонування матриці використовується для реалізація шару прямого поширення. Хоча форма матриці не сприяє на швидкість обчислення, для зручності було обрано квадратні матриці. Реалізація цієї операції відрізняється малою кількістю обчислень у порівнянні з множенням. Результат порівняння швидкості транспонування наведено на рисунку 4.7. Як і при множенні матриць у середньому графічний процесор виконує операцію швидше. Нажаль кількість розмірів матриць обмежено апаратними можливостями, та при створенні більшої матриці виникає переповнення пам’яті.

Отже у ході порівняння різних операцій на процесорі та на графічному процесорі було доведено, що швидкість виконання матричних операцій, які виділяються своєю здатністю бути легко паралельно реалізованими, вища у графічного процесору.

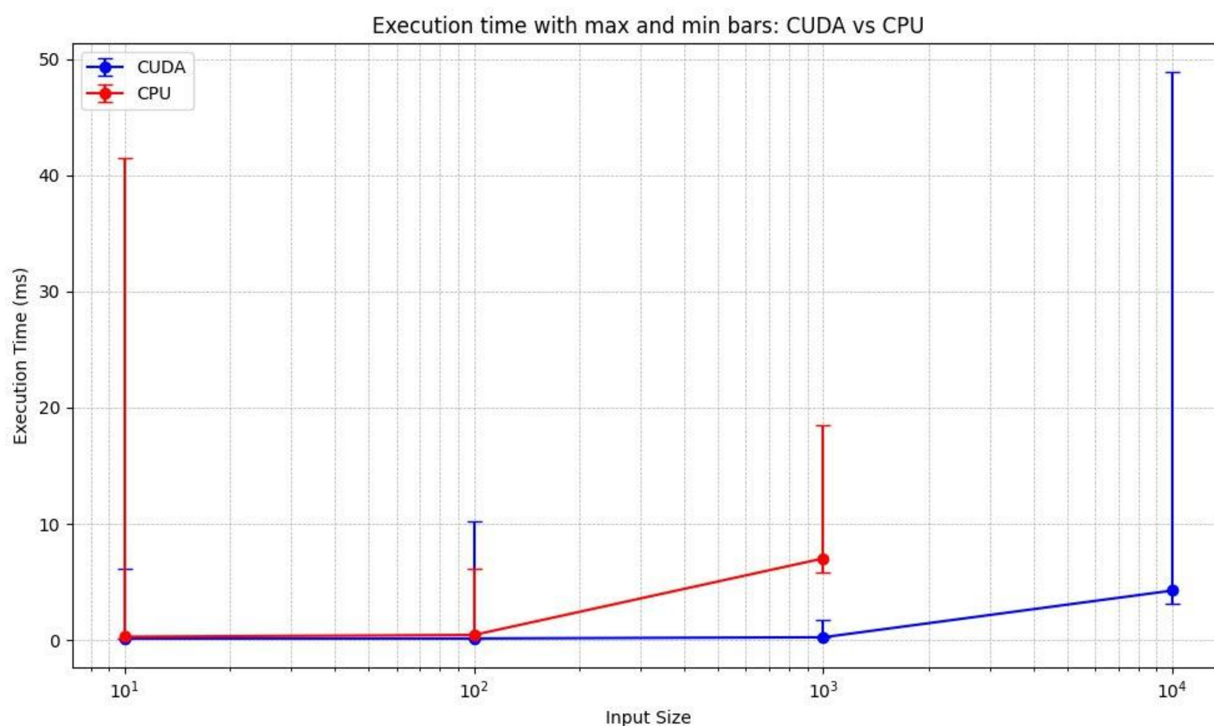


Рисунок 4.7 – Швидкість транспонування матриці

4.3 Порівняння реалізації та pyTorch

Порівняння реалізації бібліотеки з pyTorch є логічним кроком, для оцінки її ефективності. Незважаючи на те, що pyTorch існує понад десять років, активно розвивається та оптимізується великою спільнотою, спроба написання алгоритму, який може працювати швидше в певних умовах, не є поганим рішенням. Порівняння бібліотек дозволило виявити сильні та слабкі сторони власної реалізації, та потенційні напрямки для вдосконалення. Крім того порівняння надає об’єктивну оцінку ефективності реалізації та оцінку подальшого використання для вирішення задач.

Для порівняння швидкості операцій було обрано наступні операції: додавання, матричне множення, та транспонування. Вибір цих операцій спирається на популярність використання, та важливість для навчання бібліотеки. Алгоритм пошуку градієнту не було порівняно, через сильну залежність від швидкості обчислень, та відсутність можливості об’єктивно оцінити різницю у швидкості.

Операція додавання є однією з найпростіших операцій з точки зору реалізації як для графічного процесору, так і для центрального процесору. На рисунку 4.8 представлено результати порівняння швидкості виконання цієї операції на різних процесорах та у різних бібліотеках. Аналіз графіку показує, що при розмірі тензорів близькому до десяти тисяч елементів, час виконання операцій додавання практично не відрізняється між графічним та центральним процесором, незалежно від бібліотеки. Крім того форма кривих часу обчислення для PyTorch та розробленої реалізації є майже ідентичною.

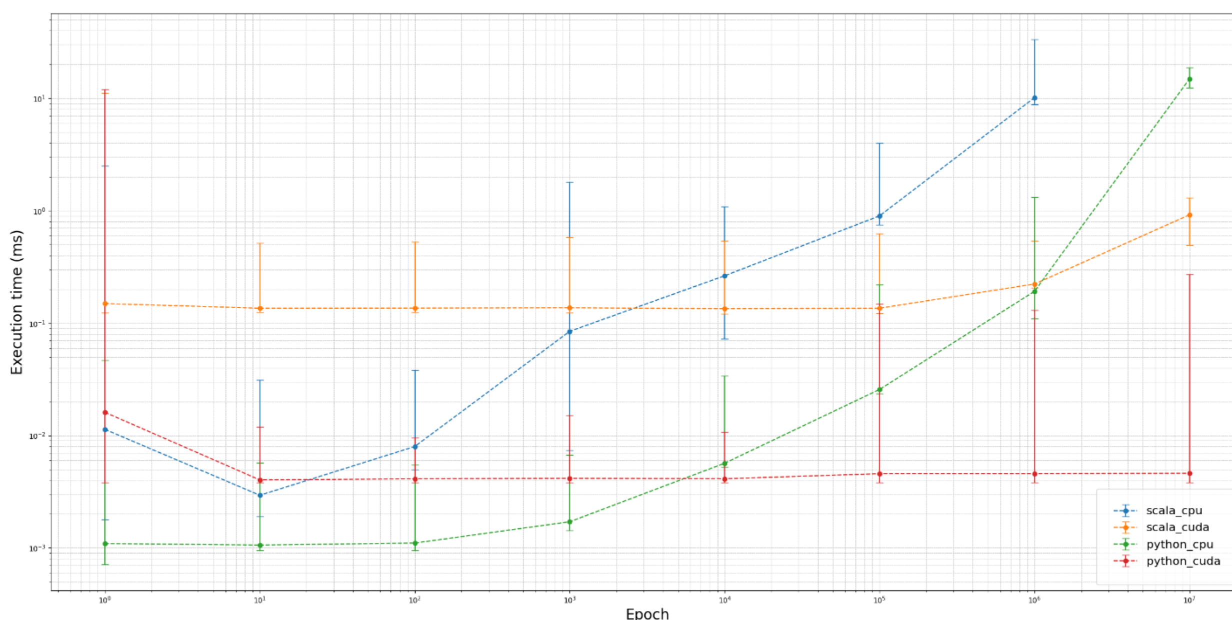


Рисунок 4.8 – Порівняння швидкості суми у різних бібліотеках

Операція матричного множення має багато різних способів реалізації. У ході роботи було реалізовано найпростіший алгоритм, що спирається на визначення матричного множення. Цей алгоритм не має великої швидкості, але простий для написання. Іншим алгоритмом є алгоритм Штрассена, який прискорює множення матриць зменшуючи кількість множень чисел. Одним з останніх проривів у цій задачі було використання штучного інтелекту AlphaTensor для пошуку більш оптимальних алгоритмів, які зменшили

кількість множень. Ці алгоритми є складними та не розглядалися як ті, що будуть реалізовані у цій роботі.

Результат порівняння швидкості обчислення матричного множення для різних бібліотек та на різних процесорах наведено на рисунку 4.9. З аналізу цього рисунку виходить що швидкість обчислень на процесорі бібліотекою, що була реалізована відрізняються повільністю, у порівнянні з іншими, що мають близьку швидкість.

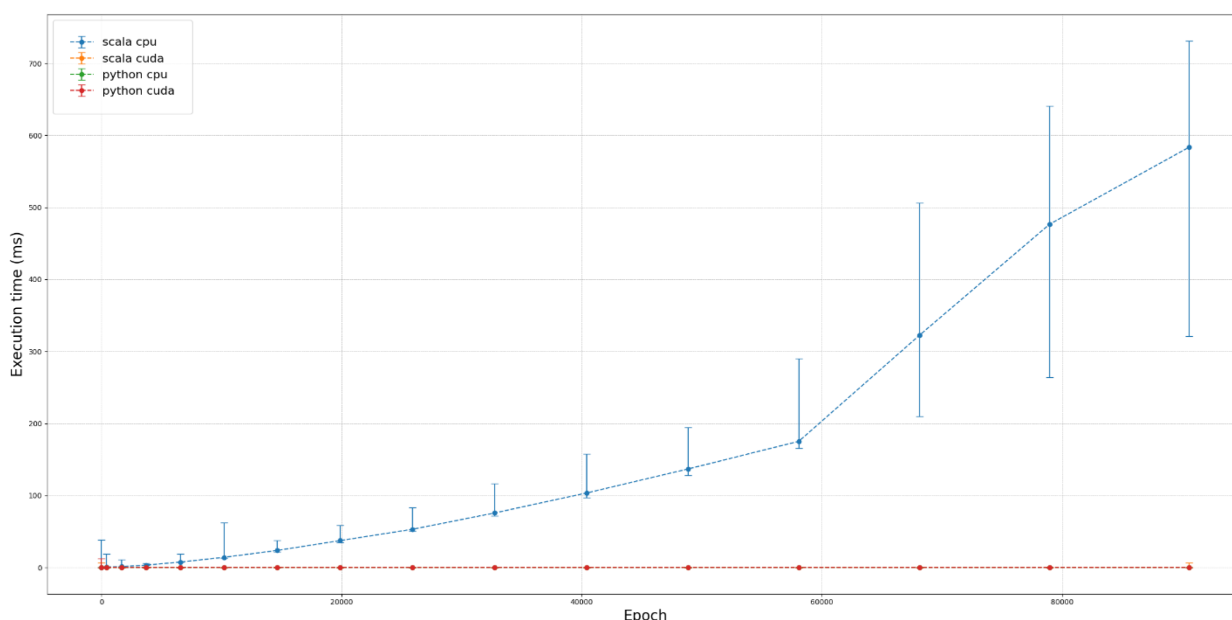


Рисунок 4.9 – Порівняння швидкості множення у різних бібліотеках

Транспонування у порівнянні з матричним множенням є простої операцією та не має великої кількості різних алгоритмів для прискорення обчислень. Графік швидкості обчислення транспонування матриці у залежності від пристрою та бібліотеки наведено на рисунку 4.10. Для транспонування створено квадратні матриці, довжини сторін яких вказані на осі x Аналізуючи цей графік отримано висновок, що у бібліотеці pyTorch швидкість транспонування не залежить від пристрою. Як і у інших експериментах pyTorch виконує операції на порядок швидше.

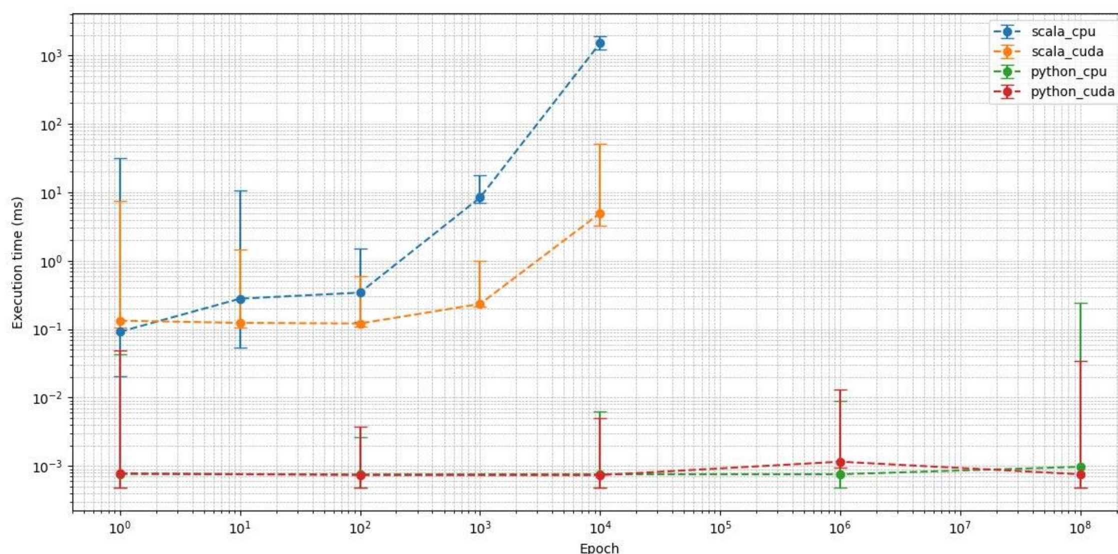


Рисунок 4.10 – Швидкість обчислення транспонування

Отже в усіх експериментах операції pyTorch виконано швидше ніж у реалізації, що була створена у ході цієї роботи.

Причиною цього є використання оптимізованих бібліотек для обчислення операцій лінійної алгебри. Такі бібліотеки називаються реалізацією BLAS (Basic Linear Algebra Subprograms). Усі операції у BLAS розподіленні на три рівні в залежності від складності операції (лінійної, квадратичної та кубічної). Яскравими прикладами реалізації є OpenBLAS та cuBLAS.

OpenBLAS – відкрите програмне забезпечення що реалізує BLAS, та було створено у 2011 році. Функції що реалізовані у цій технології оптимізовані для швидкості та використовують для цього усі можливості процесору. cuBLAS є реалізацію BLAS для обчислень на графічному процесорі, що оптимізована для відеокарт від компанії NVIDIA. Через високу оптимізацію та використання графічного процесору, швидкість обчислень є дуже високою.

Для реалізації операцій на графічному процесорі, що є дуже популярними у нейронних мережах, таких як згортка та зворотній прохід згортки, множення матриць та механізм уваги у pyTorch використано бібліотеку cuDNN. Ця бібліотека розроблена компанією NVIDIA та реалізує

алгоритми для популярних операцій у навчанні та використанні нейронних мереж. Прикладами реалізованих алгоритмів є механізми уваги, функції нормалізації, та матричне множення. Також ця бібліотека підтримує злиття ядр – об'єднання кількох простих CUDA ядр у єдине більш складне. Злиття знижує втрату ресурсів на запуск ядр, та запобігає багаторазовому звертанню до глобальної графічної пам'яті, що є дуже повільною.

Через те, що під час реалізації усі розроблені алгоритми не використовували BLAS реалізації та не були оптимізовані під різні типи процесорів, та не використовували усі його можливості реалізація втратила швидкість.

Для пришвидшення роботи бібліотеки, що було розроблено потрібно використати або розробити оптимізовані під різні архітектури процесорів алгоритми. Для розробки цих алгоритмів краще усього підходить мова C++ або C через можливість керування апаратними ресурсами.

Прикладом обчислювальної бібліотеки є Breeze – бібліотека наукових обчислень, що є аналогом Numpy бібліотеки у Scala. Ця бібліотека використовує через JNI обчислювальні ядра, прикладом якого є OpenBLAS. Використання цієї бібліотеки у реалізації бібліотеки глибокого навчання спрощує задачу. Однак Breeze не має можливості використовувати GPU.

Іншим прикладом бібліотеки для математичних обчислень є ND4J. Цю бібліотеку використовує Deeplearning4j, для моделювання тензорів, та швидких обчислень. Особливістю ND4J є підтримка GPU, що робить цю бібліотек гарним обчислювальним ядром для реалізації бібліотеки глибокого навчання.

# ВИСНОВКИ

У ході виконання роботи було розроблено бібліотеку для глибокого навчання мовою програмування Scala із використанням технології паралельних обчислень CUDA, що дозволяє ефективно використовувати ресурси графічного процесора (GPU) для прискорення тренування нейронних мереж.

У результаті детального аналізу предметної галузі було встановлено, що доцільним є використання абстракції тензора як базової одиниці для зберігання і обробки даних у бібліотеці. Використовуючи цю абстракцію було реалізовано пряме та зворотне поширення в нейронній мережі.

У рамках реалізації було створено повнозв’язні шари прямого поширення, які є ключовими компонентами більшості архітектур глибокого навчання. Крім того, були реалізовані поширені функції активації, такі як сигмоїдна функція, гіперболічний тангенс та випрямлений лінійний елемент (ReLU), що дозволяють моделювати нелінійності у мережі та сприяють ефективному навчанню моделі.

Також до складу бібліотеки увійшли функції втрат, необхідні для оцінювання якості моделі під час навчання. Було реалізовано такі функції втрат, як середньоквадратична помилка, середня абсолютна похибка та крос-ентропія, які дають змогу адаптувати навчання під різні типи задач – як регресійні, так і класифікаційні.

Таким чином, створена бібліотека є основою для побудови, тренування та тестування моделей глибокого навчання в середовищі Scala, та може бути розширена за допомогою створення нових шарів, нових оптимізаторів, та нових функцій похибок.

## ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Simonyan K., Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv*. URL: https://doi.org/10.48550/arXiv.1409.1556 (date of access: 10.04.2025).

2. Deep Residual Learning for Image Recognition / K. He et al. *ArXiv*. URL: https://doi.org/10.48550/arXiv.1512.03385 (date of access: 10.04.2025).

3. ImageNet. ImageNet. URL: https://www.image-net.org/download.php (date of access: 10.04.2025).

4. Rombaut B., Khomh F. An Empirical Study of Library Usage and Dependency in Deep Learning Frameworks. *ArXiv*. URL: https://doi.org/10.48550/arXiv.2211.15733 (date of access: 10.04.2025).

5. torch.Tensor – PyTorch 2.7 documentation. PyTorch. URL: https://pytorch.org/docs/stable/tensors.html (date of access: 02.05.2025).

6. Team C. Caffe2 and PyTorch join forces to create a Research + Production platform PyTorch 1.0. Caffe2. URL: https://caffe2.ai/blog/2018/05/02/Caffe2_PyTorch_1_0.html (date of access: 02.05.2025).

7. Introduction to CUDA C++ Programming Guide. NVIDIA Documentation Hub - NVIDIA Docs. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (date of access: 02.05.2025).

8. D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 1986.

9. McCulloch W., Pitts W. A logical calculus of the ideas immanent in nervous activity. *Nature*. 1943.

10. A theory of steady-state activity in nerve-fiber networks: I. Definitions and preliminary lemmas. *The bulletin of mathematical biophysics*. 1941. Vol. 3. P. 63–69.

11. Brownlee J. A Gentle Introduction to the Rectified Linear Unit (ReLU). URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

12. Sigmoid – PyTorch 2.7 documentation. PyTorch. URL: https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html (date of access: 02.05.2025).

13. Carmer J. The Origins of Logistic Regression. 2003.

14. George F. Becker and C. E. Van Orstrand. Hyperbolic Functions. 1909.

15. Boltzmann L. Studies on the balance of living force between moving material points. *Wiener Berichte*. 58: 517–560.

16. Mean Squared Error (MSE). Probability, Statistics & Random Processes | Free Textbook | Course. URL: https://www.probabilitycourse.com/chapter9/9_1_5_mean_squared_error_MSE.php (date of access: 02.05.2025).

17. Willmott, Cort J.; Matsuura, Kenji (December 19, 2005). "Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance". URL: https://www.int-res.com/articles/cr2005/30/c030p079.pdf (date of access: 02.05.2025).

18. CrossEntropyLoss – PyTorch 2.7 documentation. *PyTorch*. URL: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html (date of access: 02.05.2025).

19. Про університет | ХНУРЕ – Харківський національний університет радіоелектроніки. NURE. URL: https://nure.ua/universytet/pro-universitet (дата звернення: 02.05.2025).

20. Історія кафедри – Кафедра штучного інтелекту ХНУРЕ. Кафедра штучного інтелекту ХНУРЕ. URL: https://ai.nure.ua/istoriya-kafedri (дата звернення: 02.05.2025).