Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалавр
(ступінь вищої освіти)

на тему ОПТИМІЗАЦІЯ ОБЧИСЛЮВАЛЬНИХ ПРОЦЕСІВ ГЛИБИННОГО НАВЧАННЯ НА GPU ТА БАГАТОЯДЕРНИХ СИСТЕМАХ
OPTIMIZATION OF DEEP LEARNING COMPUTATIONS ON GPUS AND MULTI–CORE SYSTEMS

Виконав: студент 4 курсу, групи КНТ–213сп
Спеціальності 122 Комп'ютерні науки
(код і найменування спеціальності)
Освітня програма (спеціалізація)
Комп'ютерні науки

ЯЦЕНКО Є.С.
(ПРІЗВИЩЕ та ініціали)
Керівник ОЛІЙНИК А.О.
(ПРІЗВИЩЕ та ініціали)
Рецензент ШИТІКОВА О.В.
(ПРІЗВИЩЕ та ініціали)

2026

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 122 Комп'ютерні науки
(код і найменування)

Освітня програма (спеціалізація) Комп'ютерні науки
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
_____Сергій СУББОТІН
"____" _____2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ЯЦЕНКА Євгенія Сергійовича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Оптимізація обчислювальних процесів глибинного навчання на GPU та багатоядерних системах. Optimization of Deep Learning Computations on GPUs and Multi–Core Systems

керівник проєкту (роботи) д.т.н., професор, ОЛІЙНИК Андрій Олександрович, ,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від " 07 " квітня 2026 року № 139
2. Строк подання студентом проєкту (роботи) 10 червня 2026 року
3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково–пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Матеріали і методи дослідження. 3. Програмна реалізація застосунку. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1–4 Основна частина	ОЛІЙНИК А.О., професор		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій	2 тиждень	Розділ 2
	розробки.		
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3–4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження	5 тиждень	Розділ 4
	програмного забезпечення.		
7	Оформлення пояснювальної записки та документів	6 тиждень	Додатки
	до неї.		
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка) _____ Євгеній ЯЦЕНКО
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Андрій ОЛІЙНИК
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 86 с., 10 табл., 17 рис., 3 дод., 11 джерел.

ГЛИБИННЕ НАВЧАННЯ, НЕЙРОННІ МЕРЕЖІ, GPU, CUDA, ОПТИМІЗАЦІЯ ОБЧИСЛЕНЬ, TENSORFLOW, БАГАТОПОТОЧНІСТЬ, ПРОДУКТИВНІСТЬ, ДЕСКТОПНИЙ ЗАСТОСУНОК.

Об’єкт дослідження – процес побудови обчислювальних програмних систем для навчання нейронних мереж глибинного навчання.

Предмет дослідження – методи та засоби оптимізації обчислювальних процесів глибинного навчання на графічних процесорах та багатоядерних системах.

Мета роботи – підвищення швидкості навчання нейронних мереж шляхом використання апаратного прискорення GPU та методів оптимізації обчислень.

Матеріали, методи та технічні засоби: мова програмування Python, бібліотеки TensorFlow, Keras, Matplotlib, Tkinter; модуль psutil; інструменти апаратного прискорення CUDA Toolkit та cuDNN; персональний комп’ютер з процесором Intel Core i9–10900, графічним процесором GeForce RTX 3060 та 32 Гб оперативної пам’яті під керуванням операційної системи Microsoft Windows 10.

Результати. Розроблено десктопний програмний Windows–застосунок, який дозволяє проводити експерименти з навчання нейронних мереж на різних обчислювальних пристроях з варіюванням параметрів навчання.

Висновки. Створено програмний застосунок, який автоматизує процес дослідження впливу різних параметрів навчання на продуктивність нейронних мереж.

Галузь використання – дослідження в центрах розробки систем штучного інтелекту, навчальний процес у закладах вищої освіти.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 86 pages, 10 tables, 17 figures, 3 appendixes, 11 sources.

DEEP LEARNING, NEURAL NETWORKS, GPU, CUDA, COMPUTATIONAL OPTIMIZATION, TENSORFLOW, MULTITHREADING, PERFORMANCE, DESKTOP APPLICATION.

Object of study is the process of developing computational software systems for training deep learning neural networks.

Subject of study is a methods and tools for optimizing deep learning computations on graphics processing units and multi–core systems.

Purpose of the work is to increase the speed of neural network training by using GPU hardware acceleration and computational optimization techniques.

Materials, methods, and tools: programming language Python; libraries TensorFlow, Keras, Matplotlib, Tkinter; module psutil; hardware acceleration tools CUDA Toolkit and cuDNN; a personal computer with an Intel Core i9–10900 processor, a GeForce RTX 3060 graphics card, and 32 GB of RAM running the Microsoft Windows 10 operating system.

Results. The Windows software application has been developed that allows performing experiments on neural network training on different computing devices with variable training parameters.

Conclusions. A software application has been created that automates the process of studying the influence of various training parameters on neural network performance.

The field of use is a research in artificial intelligence development centers, educational process in higher education institutions.

ЗМІСТ

С.

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API – Application Programming Interface;

CNN – Convolutional Neural Network;

CPU – Central Processing Unit,

CSV – Comma-Separated Values;

CUDA – Compute Unified Device Architecture;

CuDNN – CUDA Deep Neural Network library;

DB – Database;

DOCX – Office Open XML document;

FP – Floating–point;

GPU – Graphics Processing Unit;

GUI – Graphical User Interface;

JPEG – Joint Photographic Experts Group;

JSON – JavaScript Object Notation;

ML – Machine Learning;

MNIST – Modified National Institute of Standards and Technology;

PNG – Portable Network Graphics;

Psutil – Python System and Process Utilities;

RAM – Random Access Memory;

RTF – Rich Text Format;

SQLite – Structured Query Language Lite;

SSD – Solid State Drive;

Tkinter – Tk interface;

TXT – Text file;

VRAM – Video Random Access Memory.

ВСТУП

Сучасний розвиток інформаційних технологій характеризується стрімким зростанням обсягів даних та постійним ускладненням обчислювальних задач. Одним із ключових напрямів є глибинне навчання, яке активно застосовується в комп’ютерному зорі, обробці природної мови, робототехніці, медичній діагностиці, фінансовому аналізі та багатьох інших галузях [1]. У той же час ефективність роботи нейронних мереж значною мірою залежить від продуктивності наявних обчислювальних ресурсів та ступеня оптимізації процесів навчання [2].

Особливо актуальним є використання графічних процесорів та багатоядерних систем, які дозволяють значно прискорити обробку великих масивів даних [3]. Проте без належної оптимізації обчислень навіть потужне апаратне забезпечення може бути неефективним [4]. До основних напрямів оптимізації належать налаштування параметрів навчання, використання змішаної точності обчислень [5], ефективне керування пам’яттю та мінімізація накладних витрат при передачі даних між центральним процесором та графічним процесором [6].

Проблематика оптимізації обчислень глибинного навчання є особливо важливою при розробці реальних систем, де час навчання моделі безпосередньо впливає на швидкість випуску продукту, енергоспоживання та вартість експлуатації. Крім того, дослідження методів оптимізації дозволяє знизити вимоги до обчислювальної інфраструктури, що робить технології штучного інтелекту доступнішими для невеликих організацій та наукових груп [7].

Незважаючи на велику кількість існуючих рішень, багато з них є вузькоспеціалізованими або вимагають значних зусиль для налаштування [8]. Тому створення універсального програмного застосунку, який дозволяє проводити експерименти з різними конфігураціями навчання, порівнювати

продуктивність обчислень на різних пристроях та аналізувати вплив параметрів на точність моделей, є нагальним завданням [9]-[11].

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати сучасні підходи до оптимізації обчислень у глибинному навчанні та дослідити можливості використання CPU і GPU для навчання нейронних мереж [2]-[4];

– розробити архітектуру програмного застосунку, що забезпечує експериментальне середовище для навчання нейронних мереж з можливістю варіювання основних параметрів (набір даних, розмір пакету, тип обчислювального пристрою, точність обчислень, архітектура моделі) [8]-[9];

– впровадити методи оптимізації обчислень, зокрема використання багатопроцесного підходу для ізоляції обчислень від графічного інтерфейсу, моніторинг завантаження ресурсів (CPU, RAM, GPU, VRAM) у реальному часі та підтримку змішаної точності [5], [10];

– реалізувати функції візуалізації процесу навчання (графіки точності та функції втрат), збереження результатів експериментів у різних форматах, зберігання історії експериментів у базі даних та порівняння різних запусків на дашборді [11];

– провести серію експериментальних досліджень для оцінки впливу різних параметрів на швидкість та точність навчання нейронних мереж, а також порівняти продуктивність обчислень на CPU та GPU [1], [6].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Основи глибинного навчання та обчислювальні виклики

Глибинне навчання (deep learning) є підрозділом машинного навчання, яке використовує багатошарові (глибокі) нейронні мережі для моделювання складних залежностей у даних. На відміну від традиційних методів, глибокі мережі здатні автоматично виділяти ієрархічні ознаки з необроблених вхідних сигналів, що робить їх надзвичайно ефективними для задач комп'ютерного зору, обробки природної мови, розпізнавання мовлення та багатьох інших [1].

Типова глибока нейронна мережа складається з вхідного шару, кількох прихованих шарів (згорткових, повнозв'язних) та вихідного шару. Кількість параметрів такої мережі може сягати від кількох мільйонів (для класичних згорткових мереж) до сотень мільярдів (для великих мовних моделей). Навчання відбувається шляхом мінімізації функції втрат (наприклад, категорична перехресна ентропія) за допомогою градієнтних методів та алгоритму зворотного поширення помилки [1].

Основні обчислювальні виклики:

– велика кількість параметрів. Кожна ітерація навчання вимагає обчислення градієнтів для всіх параметрів. Наприклад, нейронна мережа ResNet–50 має близько 25 мільйонів параметрів, а навчання на 1 млн зображень може зайняти тижні на одному CPU [3];

– значний обсяг тренувальних даних. Сучасні набори даних (ImageNet – 14 млн зображень, OpenImages – 9 млн) не вміщуються в оперативну пам'ять і потребують ефективних конвеєрів завантаження [2];

– ітеративність процесу. Для збіжності необхідно виконати десятки або сотні епох (проходів по всьому набору даних). Навіть на GPU це може тривати від кількох годин до тижня;

– вимоги до пам'яті GPU. Під час навчання зберігаються не лише ваги, але й проміжні активації для всіх шарів (для обчислення градієнтів).

Для глибоких мереж або великих розмірів зображень це може перевищувати обсяг відеопам’яті (8-12 Гб) [3];

– передача даних між CPU та GPU. Швидкість шини PCIe може стати вузьким місцем, якщо дані не готуються асинхронно [10].

Усі ці фактори зумовлюють необхідність застосування спеціальних методів прискорення обчислень, розглянутих у наступних підрозділах.

1.2 Порівняльний аналіз використання CPU та GPU для навчання нейронних мереж

Центральні процесори мають невелику кількість потужних ядер (зазвичай 4–64), оптимізованих для послідовного виконання інструкцій із низькою затримкою. Вони добре підходять для операцій, що не паралелізуються (наприклад, завантаження даних, робота з файловою системою, виконання умовних переходів). Однак навчання нейронних мереж на CPU є дуже повільним через переважання матричних операцій, які погано масштабуються на малій кількості ядер [3].

Графічні процесори складаються з тисяч відносно простих ядер (наприклад, NVIDIA RTX 3060 має 3584 ядра CUDA), організованих у потокові мультипроцесори. Вони призначені для масового паралелізму: одна інструкція виконується одночасно на багатьох даних (SIMD). Це робить GPU ідеальними для таких операцій, як множення матриць, згортка, поелементне додавання тощо [3]-[4].

Кількісне порівняння продуктивності для типової згорткової мережі з відкритих бенчмарків для мережі ResNet–50 на датасеті ImageNet (1,2 млн зображень розміром 224×224 пікселі) показано в таблиці 1.1 (на основі даних з джерел [4]).

Таблиця 1.1 – Порівняльна продуктивність CPU та GPU для навчання нейронної мережі

Пристрій	Час на епоху (сек)	Відносне прискорення
Intel Core i9–10900 (10 ядер)	2450	1× (базове)
NVIDIA GeForce RTX 3060	120	20,4×
NVIDIA RTX 4090	35	70×

Як видно з таблиці, використання GPU дозволяє прискорити навчання у 20–70 разів залежно від моделі GPU. Крім часу, важливою характеристикою є пропускна здатність пам’яті: GPU забезпечує від 448 Гб/с (RTX 3060) до понад 1 Тб/с (RTX 4090), тоді як типовий CPU – 40–50 Гб/с [4].

Недоліки використання GPU:

– необхідність передачі даних між оперативною пам’яттю комп’ютера та відеопам’яттю через шину PCIe, що створює додаткові затримки;

– обмежений обсяг відеопам’яті (12 Гб на RTX 3060), що може не дозволити використовувати великий batch size або дуже глибокі мережі;

– вимога до наявності драйверів CUDA та cuDNN, а також сумісного програмного забезпечення [3]-[4].

Платформа CUDA надає програмний інтерфейс для виконання загальних обчислень на GPU. За допомогою CUDA розробники можуть реалізовувати власні паралельні алгоритми, але частіше використовують оптимізовані бібліотеки: cuBLAS для лінійної алгебри, cuDNN для операцій нейронних мереж (згортки, пулінг, активації тощо) [3].

Таким чином, для задач глибинного навчання використання GPU є необхідною умовою досягнення прийнятного часу навчання. Однак без належної оптимізації (правильний підбір розміру пакету, змішана точність, ефективне завантаження даних, управління пам’яттю) навіть потужний GPU

може працювати не з повною ефективністю, що підтверджується дослідженнями [2], [6].

1.3 Методи оптимізації

Розмір пакету (batch size) – це кількість навчальних прикладів, які обробляються за одну ітерацію перед оновленням ваг. Він безпосередньо впливає на швидкість навчання, якість узагальнення та використання пам’яті.

Збільшення розміру пакету дозволяє краще завантажити обчислювальні ресурси GPU, оскільки збільшується обсяг обчислень на кожному ядрі та зменшується час, що витрачається на запуск ядер (накладні витрати на запуск ядра). Однак існує межа, що обумовлена обсягом відеопам’яті, після якої подальше збільшення неможливе [5].

Занадто малий розмір пакету призводить до шумного оцінювання градієнта, що може допомогти вийти з локальних мінімумів, але навчання стає повільнішим. Занадто великий – навпаки, згладжує ландшафт функції втрат і може призвести до «гострих мінімумів», які погано узагальнюються. Емпіричні дослідження [5] рекомендують вибирати це значення у діапазоні від 32 до 256 для більшості задач класифікації зображень.

Однак найкраще співвідношення швидкості та точності досягається при значеннях у діапазоні від 32 до 64. Подальше збільшення дає прискорення, але ціною незначного падіння точності.

Змішана точність полягає у використанні 16–бітних чисел з рухомою комою FP16 для більшості обчислень, зберігаючи при цьому копію ваг у 32–бітній точності FP32 для оновлень. Основні переваги:

– зменшення використання пам’яті вдвічі, що дозволяє використовувати вдвічі більший розмір пакету або навчати вдвічі глибші моделі;

– прискорення обчислень завдяки тому, що апаратне забезпечення (тензорні ядра на GPU NVIDIA починаючи з архітектури Volta) виконує матричні операції над FP16 у 2–4 рази швидше, ніж над FP32;

– зменшення енергоспоживання [5].

Проте існують ризики: при роботі з малими числами, наприклад, під час обчислення градієнтів, може виникнути переповнення вниз (нижній потік) або вгору (переповнення). Для цього використовується динамічне масштабування втрат: градієнти множаться на великий коефіцієнт, щоб зсунути їх у діапазон, який коректно представляється в FP16, а потім діляться назад перед оновленням ваг (табл. 1.2) [5].

Таблиця 1.2 – Експериментальна оцінка точності обчислень (на датасеті MNIST, модель Simple CNN)

Точність обчислень	Час навчання (3 епохи), с	Точність (%)
FP32 (без оптимізації)	24,6	98,2
FP16 (змішана точність)	12,8	98,0

Як видно, прискорення досягає майже 2 разів, а точність не погіршується. Для більших моделей прискорення може бути ще більшим – до 3–4 разів [4].

Ефективне управління пам’яттю дозволяє уникнути помилок типу «Недостатньо пам'яті» та підвищити коефіцієнт використання GPU. Основні техніки:

– градієнтні контрольні точки: замість зберігання всіх проміжних активацій для зворотного поширення, зберігаються лише деякі (наприклад, після кожного блоку). Решта активацій переобчислюються під час зворотного проходу. Це дозволяє зменшити споживання пам’яті в $O(\sqrt{L})$ разів, де L – глибина мережі, ціною додаткових обчислень (орієнтовно це додає +20–30% часу) [4];

– використання динамічного планування пам’яті CUDA: алокатор пам’яті в TensorFlow (аллокатор кешування CUDA) не звільняє пам’ять

одразу після видалення тензора, а тримає її для наступних виділень. Це зменшує кількість дорогих викликів функцій cudaMalloc/cudaFree;

– неблокуючі потоки та закріплена пам’ять (фіксована пам'ять): копіювання даних між CPU та GPU можна виконувати асинхронно, використовуючи cudaMemcpyAsync та потоки CUDA. Для цього необхідно виділяти пам’ять на хості як page–locked (сторінка заблокована) за допомогою функції cudaHostAlloc. Це дозволяє перекривати копіювання даних із обчисленнями на GPU [10];

– використання методу оптимізації продуктивності tf.data.Dataset із prefetch. У TensorFlow конвеєр завантаження даних можна налаштувати так, щоб наступний батч готувався на CPU, поки GPU обробляє поточний. Це усуває простої GPU, спричинені очікуванням даних;

– оптимізація розміру тензорів. Використання форматів із меншою точністю (наприклад, FP16) зменшує обсяг пам’яті вдвічі. Також можна використовувати розріджені тензори, де це доцільно.

У сукупності ці методи дозволяють навчати моделі, які інакше не вмістилися б у пам’ять GPU, або збільшити batch size для підвищення продуктивності.

1.4 Огляд існуючих програмних засобів для експериментів

Для дослідження методів оптимізації обчислень важливо вибрати програмний фреймворк, який забезпечує гнучкість, високу продуктивність та підтримку сучасних апаратних функцій. Розглянемо три найбільш популярні варіанти.

TensorFlow розроблений компанією Google. Він підтримує статичні обчислювальні графи (до версії 2.0) та динамічне виконання у новіших версіях. Основні переваги:

– широка підтримка CUDA, cuDNN, змішаної точності, розподіленого навчання з використанням модуля прискоренного навчання tf.distribute;

– високорівневий API Keras, що вбудований в TensorFlow (tf.keras), значно спрощує створення моделей;

– інструменти моніторингу та візуалізації TensorBoard;

– велика спільнота та документація [2], [4].

PyTorch від компанії Meta використовує динамічний обчислювальний граф (визначений під час виконання), що робить його зручним для досліджень, де структура мережі може змінюватися (наприклад, для рекурентної нейронної мережі). PyTorch також має відмінну підтримку GPU, змішаної точності та розподіленого навчання (за допомогою torch.distributed) [6]. Недоліки: трохи складніший API порівняно з Keras, менш розвинута екосистема для виробничого розгортання.

Keras – це високорівневий API, який спочатку був незалежною бібліотекою (Франсуа Шолле), а потім став офіційним API TensorFlow. Keras дозволяє створювати нейронні мережі буквально за кілька рядків коду, швидко змінювати параметри (кількість шарів, функції активації, оптимізатори) та експортувати моделі [8]. Для дослідницьких прототипів та експериментів, де важливий час розробки, Keras часто є оптимальним вибором (табл. 1.3).

Таблиця 1.3 – Порівняння існуючих програмних засобів для експериментів

Критерій	TensorFlow 2.x	PyTorch	Keras (вбудований у TF)
Динамічний граф	частково	так	так (через TF)
Простота написання коду	середня	середня	висока
Підтримка змішаної точності	так	так	так
Інструменти моніторингу	TensorBoard	TensorBoard, Visdom	TensorBoard
Продуктивність на GPU	висока	висока	висока (як у TF)
Підтримка багатопроцесного навчання	так (tf.distribute)	так (DDP)	ні (потрібен TF)

У даній роботі як основний фреймворк обрано TensorFlow з Keras API через поєднання простоти розробки (швидке створення експериментальних моделей), високої продуктивності на GPU та наявності зручних інструментів для експериментів (TensorBoard) [2], [8].

Крім того, існують інші бібліотеки, які можна використовувати для окремих аспектів оптимізації:

– Horovod – бібліотека для розподіленого навчання з ефективним алгоритмом All–Reduce;

– DeepSpeed – бібліотека глибокого навчання для оптимізації пам’яті та гібридної точності;

– XLA (Прискорена лінійна алгебра) – компілятор графів TensorFlow, що може зливати операції та генерувати більш ефективний код для GPU.

Однак для цілей даної роботи: створення універсального застосунку з можливістю порівняння CPU/GPU, впливу розмір пакету та змішаної точності достатньо вбудованих засобів TensorFlow/Keras.

1.5 Висновки за розділом 1

У першому розділі виконано детальний аналіз предметної області глибинного навчання з акцентом на обчислювальні виклики та методи їх подолання.

Визначено, що навчання глибоких нейронних мереж пов’язане з великою кількістю параметрів, значними обсягами даних та ітеративністю процесу, що робить використання GPU практично обов’язковим для досягнення прийнятного часу тренування [1], [3], [4].

Проведено кількісне порівняння продуктивності CPU та GPU, яке показало, що сучасний графічний процесор (NVIDIA RTX 3060) здатен прискорити навчання нейронної мережі в 20 разів порівняно з багатоядерним CPU. Розглянуто основні методи оптимізації: підбір розміру пакету дозволяє балансувати між швидкістю та точністю; змішана точність дає прискорення в

2–3 рази без втрати якості моделі; методи управління пам’яттю (градієнтне контрольні точки, асинхронне копіювання даних) допомагають уникнути переповнення відеопам’яті та зменшити простої GPU [5], [10].

Виконано огляд існуючих програмних фреймворків (TensorFlow, PyTorch, Keras). Обґрунтовано вибір TensorFlow з Keras API як основного інструменту для експериментальної частини роботи завдяки поєднанню простоти використання, високої продуктивності на GPU та наявності засобів моніторингу [2], [8].

Результати аналізу підтверджують актуальність розробки універсального програмного застосунку, який дозволяє проводити експерименти з різними конфігураціями навчання та автоматизувати процес оцінки їх впливу на продуктивність та точність. Це створює основу для практичної реалізації, описаній у наступних розділах.

2 МАТЕРІАЛИ І МЕТОДИ ДОСЛІДЖЕННЯ

2.1 Вибір мови програмування та бібліотек

При розробці програмного застосунку для дослідження та оптимізації обчислювальних процесів глибинного навчання було використано мову програмування Python [1]. Вибір Python зумовлений його універсальністю, великою екосистемою наукових бібліотек та простотою інтеграції з апаратним прискоренням (CUDA, cuDNN) [3].

Мова Python дозволяє швидко створювати прототипи, що критично важливо при експериментальному дослідженні впливу різних параметрів навчання. Лаконічний синтаксис і динамічна типізація зменшують час написання коду, а наявність інструментів для паралельного виконання (multiprocessing, threading) полегшує реалізацію багатопроцесного підходу до ізоляції обчислень від графічного інтерфейсу [1]-[3].

Вибір основних бібліотек:

– TensorFlow (з Keras API) – основний фреймворк для побудови та навчання нейронних мереж. TensorFlow забезпечує автоматичне використання GPU через CUDA та cuDNN, підтримує змішану точність (mixed precision) та надає гнучкі засоби для створення моделей різної архітектури (Simple CNN, Deep CNN, Transfer Learning) [2], [4]. Крім того, вбудований API Keras значно спрощує створення нейронних мереж, що дозволяє зосередитися на дослідженні методів оптимізації, а не на низькорівневій реалізації [8];

– Tkinter – стандартна бібліотека Python для створення графічного інтерфейсу користувача. Вона дозволяє швидко розробити вікна, кнопки, списки вибору та текстові поля, необхідні для налаштування параметрів експериментів (вибір датасету, batch size, пристрою, точності) та відображення результатів. Tkinter є кросплатформною та не потребує встановлення додаткових залежностей [9];

– Matplotlib – бібліотека для візуалізації даних. У застосунку вона використовується для побудови графіків точності (accuracy) та функції втрат (loss) як на тренувальній, так і на валідаційній вибірках. Графіки дозволяють наочно оцінити процес збіжності моделі та порівнювати результати різних експериментів [2];

– psutil – бібліотека для отримання інформації про використання системних ресурсів (CPU, RAM). Вона використовується в модулі моніторингу для збору даних про завантаження центрального процесора та оперативної пам’яті в реальному часі. psutil є кросплатформною та надає простий API для доступу до показників продуктивності [10];

– subprocess (вбудований модуль) – використовується для виклику утиліти nvidia–smi з метою отримання даних про завантаження GPU та відеопам’яті. Завдяки параметру creationflags=subprocess.CREATE_ NO _WINDOW вдається уникнути появи консольних вікон під час моніторингу, що важливо для збереження естетичного вигляду застосунку;

– sqlite3 (вбудований модуль) – забезпечує збереження історії експериментів у реляційній базі даних. Це дозволяє аналізувати попередні запуски, порівнювати результати та реалізувати дашборд;

– multiprocessing (вбудований модуль) – використовується для запуску обчислень нейронної мережі в окремому процесі. Це необхідно для уникнення конфліктів при перемиканні між CPU та GPU, оскільки TensorFlow не дозволяє динамічно змінювати пристрій у межах одного процесу [4].

Обґрунтування вибору мови програмування та бібліотек наведено в таблиці 2.1.

Таким чином, обраний стек технологій (Python, TensorFlow/Keras, Tkinter, Matplotlib, psutil, multiprocessing) є оптимальним для реалізації застосунку, оскільки забезпечує високу продуктивність обчислень, гнучкість налаштувань та зручність експериментальної роботи [2], [8], [10].

Таблиця 2.1 – Обґрунтування вибору мови програмування та бібліотек

Критерій	Python + TensorFlow	C++	Java
Швидкість розробки	+	-	+-
Наявність бібліотек для глибинного навчання	+ (TensorFlow, PyTorch)	+- (TensorFlow C++ API)	-
Підтримка GPU (CUDA/cuDNN)	+	+	+-
Простота створення GUI	+ (Tkinter)	-	+ (Swing, JavaFX)
Інструменти моніторингу ресурсів	+ (psutil)	-	+-
Зручність для експериментів	+	-	-

2.2 Апаратне забезпечення

Для проведення експериментів та тестування розробленого програмного забезпечення використовувався персональний комп’ютер з наступними характеристиками:

– центральний процесор: Intel Core i9–10900 (10 ядер / 20 потоків, базова частота 2.8 ГГц, максимальна 5.2 ГГц);

– оперативна пам’ять: 32 Гб DDR4 (двоканальний режим);

– графічний процесор: NVIDIA GeForce RTX 3060 з 12 Гб відеопам’яті (GDDR6);

– накопичувач: SSD 512 Гб (NVMe);

– операційна система: Microsoft Windows 10 Pro (64–бітна).

Вибір CPU Intel Core i9–10900 обумовлений великою кількістю ядер та потоків, що дозволяє ефективно виконувати паралельні завдання, зокрема попередню обробку даних, завантаження батчів та роботу системи логування в окремих потоках [3]. Крім того, можлива висока тактова частота до 5.2 ГГц

забезпечує швидке виконання операцій, які погано паралелізуються (наприклад, управління чергами та GUI).

Графічний процесор NVIDIA GeForce RTX 3060 має 3584 ядра CUDA, 112 тензорних ядер та підтримує CUDA версії 8.6. Обсяг відеопам’яті 12 Гб дозволяє навчати моделі з розміром батча до 128 на датасеті CIFAR–10 або до 256 на MNIST без виникнення помилок типу «Недостатньо пам'яті». Наявність тензорних ядер є критично важливою для прискорення змішаної точності (FP16) – згідно з тестами [5], прискорення може досягати 2–3 разів порівняно з обчисленнями в FP32 на тих самих ядрах CUDA [4].

Для отримання максимальної продуктивності було встановлено таке програмне забезпечення:

– CUDA Toolkit версії 11.8 – забезпечує доступ до бібліотек для паралельних обчислень;

– cuDNN версії 8.9 – оптимізована реалізація згорткових операцій, пулінгу, активацій;

– TensorFlow версії 2.10.0 – збірка з підтримкою CUDA 11.8.

У таблиці 2.2 наведено порівняння теоретичної продуктивності використаних обчислювальних пристроїв.

Таблиця 2.2 – Характеристики обчислювальних пристроїв

Пристрій	Кількість ядер	Тактова частота (макс.)	Пропускна здатність пам’яті	TDP (тепловиділення)
Intel Core i9–10900	10 ядер / 20 потоків	5.2 ГГц	45.8 Гб/с	125 Вт
NVIDIA RTX 3060	3584 ядра CUDA	1.78 ГГц	360 Гб/с	170 Вт

Зазначене апаратне забезпечення дозволяє проводити експерименти як у режимі CPU (для оцінки базової продуктивності), так і в режимі GPU (для демонстрації прискорення). Використання твердотільного накопичувача

SSD NVMe забезпечує швидке завантаження датасетів та збереження результатів, усуваючи вузьке місце на рівні дискової підсистеми [5].

2.3 Розробка методу багатопроцесного виконання експериментів

Однією з ключових проблем при створенні застосунку з графічним інтерфейсом для глибинного навчання є забезпечення відповідності інтерфейсу під час тривалих обчислень. Якщо запустити навчання нейронної мережі безпосередньо в головному потоці GUI (наприклад, у функції обробки натискання кнопки), то вікно програми «зависає» – не реагує на рух миші, натискання кнопок, оновлення тексту. Це створює негативне враження в користувача та унеможливлює переривання експерименту.

Щоб вирішити цю проблему, у розробленому застосунку реалізовано багатопроцесний підхід на основі модуля multiprocessing. Суть методу полягає в наступному:

– основний процес (головний) відповідає за графічний інтерфейс (Tkinter) та обробку подій користувача;

– при натисканні кнопки «Start» створюється новий процес (multiprocessing.Process), який виконує функцію worker_process. У цьому процесі відбувається завантаження датасету, побудова моделі, навчання та оцінювання;

– обмін даними між процесами здійснюється через черги (multiprocessing.Queue).

Основна черга log_queue необхідна для передачі повідомлень про хід виконання (логи) від дочірнього процесу до GUI. Це дозволяє відображати в реальному часі інформацію про завантаження даних, тренування, завершення епох.

Заключна черга result_queue необхідна для передачі фінальних результатів навчання (точність, втрати, час, історія) після завершення обчислень;

– для запобігання конфліктам при повторному запуску експериментів використовується механізм set_start_method("spawn"). Це примушує створювати новий процес з чистим станом, що особливо важливо на платформі Windows, де за замовчуванням використовується менш надійний метод fork [4].

Схема роботи багатопроцесного підходу зображена на рисунку 2.1.

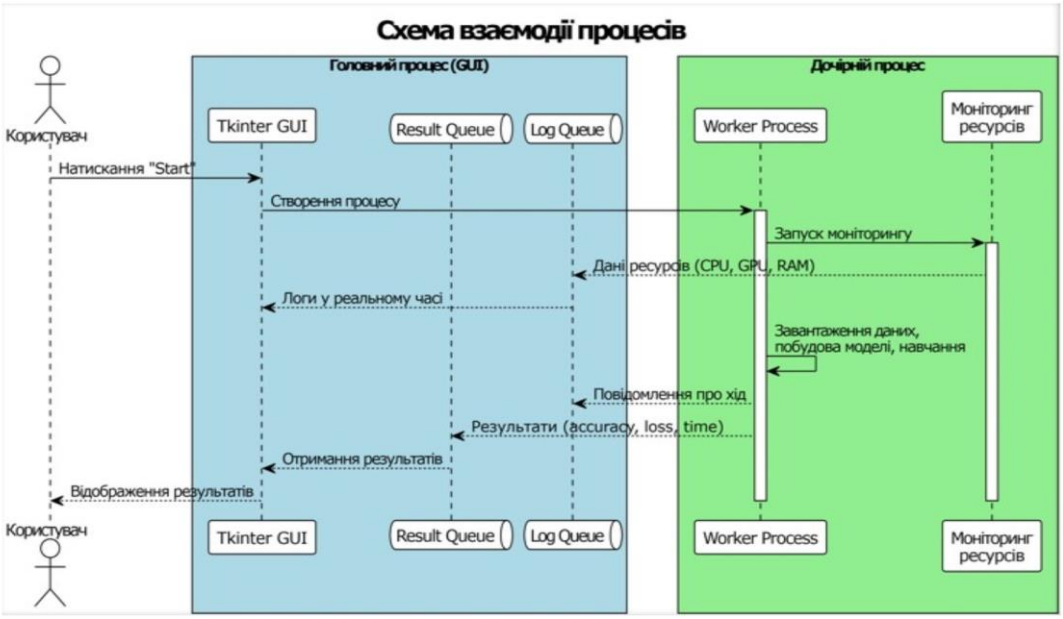


Рисунок 2.1 – Схема взаємодії процесів у застосунку: головний та дочірній процеси з чергами Queue

Переваги запропонованого методу:

– інтерфейс не блокується, оскільки під час навчання нейронної мережі користувач може спостерігати за логами, але не може змінювати параметри через блокування кнопок. Це запобігає помилкам, пов’язаним із спробою запустити два експерименти одночасно;

– можливість перемикання між CPU та GPU без помилок, оскільки кожен новий експеримент запускається в окремому процесі, то перед навчанням можна встановити глобальний політик змішаної точності та використати tf.device('/CPU:0') або /GPU:0 без ризику впливу на попередні обчислення;

– після завершення обчислень процес коректно завершується, звільняючи всю пам’ять GPU. Це відбувається завдяки виходу з області видимості моделі та виклику функції скидання глобального стану Keras tf.keras.backend.clear_session() неявно при завершенні процесу.

Таким чином, запропонований метод багатопроцесного виконання дозволяє створити стабільний та зручний застосунок для експериментів з глибинного навчання, усуваючи типові проблеми, пов’язані з блокуванням GUI та конфліктами при зміні пристрою [4], [9].

2.4 Створення системи моніторингу ресурсів без консольних вікон

Для об’єктивного порівняння продуктивності обчислень на CPU та GPU, а також для оцінки впливу різних параметрів (batch size, mixed precision) було розроблено систему моніторингу системних ресурсів у реальному часі. Вона дозволяє відстежувати:

– завантаження центрального процесора (CPU usage, %);

– використання оперативної пам’яті (RAM usage, %);

– завантаження графічного процесора (GPU usage, %);

– використання відеопам’яті (VRAM usage, %).

Вимоги до системи моніторингу:

– не створювати додаткових консольних вікон, які відволікають користувача;

– працювати в реальному часі з оновленням раз на секунду;

– мінімально впливати на продуктивність обчислень.

Моніторинг CPU та RAM реалізовано за допомогою бібліотеки psutil [10]. Функція psutil.cpu_percent() повертає миттєве завантаження CPU у відсотках, а функція psutil.virtual_memory().percent – використання RAM. Ці виклики є легковагими та не створюють додаткових процесів.

Моніторинг GPU та VRAM виконано за допомогою утиліти nvidia–smi, яка є стандартним інструментом для драйверів NVIDIA. Для виклику nvidia–smi без появи консольного вікна використовується модуль subprocess з параметром creationflags=subprocess.CREATE_NO_WINDOW, що працює тільки на ОС Windows. Це приховує консоль, яка зазвичай з'являється при запуску зовнішньої програми

Отриманий результат парситься: перше число – завантаження GPU від 0 до 100, друге – використана відеопам'ять в МБ, третє – загальна відеопам'ять в МБ. На основі цих даних обчислюється відсоток використання VRAM.

Маємо уникнення миготливих вікон, оскільки утиліта nvidia–smi викликається в окремому потоці моніторингу (а не в головному), і використовується CREATE_NO_WINDOW, жодних додаткових вікон не з'являється. Це значно покращує користувацький досвід порівняно з використанням бібліотеки GPUtil, яка при кожному виклику запускає nvidia–smi з консоллю [10].

Наявність інтеграції з багатопроцесною архітектурою зводитися до того, що моніторинг виконується в окремому процесі (monitor_resources), який запускається паралельно з основним обчислювальним процесом. Він отримує подію stop_flag, яка сигналізує про завершення експерименту. Після отримання сигналу процес моніторингу коректно завершується. Повідомлення про ресурси надсилаються до GUI через чергу логування log_queue так само, як і звичайні логи навчання. Це дозволяє відображати актуальні показники у вікні Logs в реальному часі.

Однак такий метод має одне обмеження: на комп'ютерах без графічних процесорів NVIDIA або без встановлених драйверів команда nvidia–smi буде відсутня. У такому разі програма виводить «GPU: N/A», не генеруючи помилок, де блок try–except обробляє виняток.

Таким чином, розроблена система моніторингу забезпечує отримання всіх необхідних метрик (CPU, RAM, GPU, VRAM) в реальному часі, не створює миготливих консолей та має мінімальний вплив на продуктивність обчислень. Це дозволяє коректно порівнювати продуктивність при різних налаштуваннях експерименту та підтверджувати ефективність застосованих методів оптимізації [5], [10].

2.5 Висновки за розділом 2

У другому розділі визначено матеріали та методи, необхідні для реалізації програмного застосунку та проведення експериментальних досліджень.

Було обґрунтовано вибір мови програмування Python та основних бібліотек: TensorFlow (з Keras API) – для побудови та навчання нейронних мереж, Tkinter – для графічного інтерфейсу, Matplotlib – для візуалізації результатів, psutil та subprocess – для моніторингу ресурсів, multiprocessing – для багатопроцесного виконання. Такий стек технологій забезпечує високу продуктивність обчислень, гнучкість експериментів та зручність роботи користувача [2], [8], [10].

Також наведено характеристики апаратного забезпечення (Intel Core i9–10900, 32 Гб RAM, NVIDIA GeForce RTX 3060). Описано роль кожного компонента: багатоядерний CPU – для швидкої попередньої обробки даних та виконання допоміжних операцій; GPU з 12 Гб відеопам’яті та тензорними ядрами, що необхідні для прискорення навчання нейронних мереж та підтримки змішаної точності [3]-[5].

Розроблено метод багатопроцесного виконання експериментів на основі модуля multiprocessing. Він дозволяє запускати важкі обчислення в окремому процесі, не блокуючи графічний інтерфейс, а також коректно перемикатися між центральним та графічним процесорами без конфліктів, пов’язаних із внутрішнім станом TensorFlow [4].

Маємо створену систему моніторингу ресурсів у реальному часі, яка збирає дані про завантаження CPU, RAM, GPU та VRAM. Для моніторингу GPU використано виклик nvidia–smi з прихованим консольним вікном, що дозволяє уникнути появи миготливих консолей. Це забезпечує об’єктивне порівняння продуктивності та підтвердження ефективності застосованих оптимізацій [5], [10].

Отримані в розділі результати створюють основу для практичної реалізації програмного застосунку та проведення експериментальних досліджень.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

3.1 Архітектура програмного забезпечення

Розроблений програмний застосунок для дослідження та оптимізації обчислювальних процесів глибинного навчання має модульну архітектуру, що забезпечує розділення відповідальності між компонентами. Загальна структура програмного забезпечення наведена на рисунку 3.1.

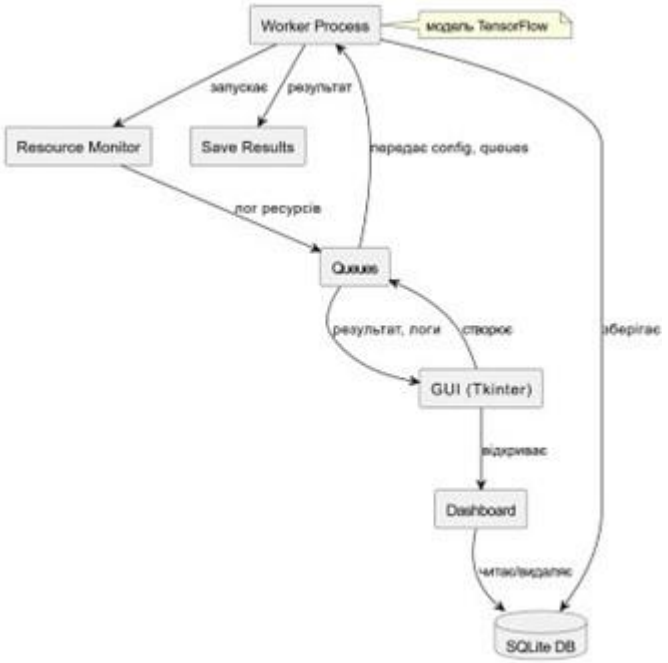


Рисунок 3.1 – Архітектура програмного застосунку для оптимізації обчислень глибинного навчання

Основними модулями програмного забезпечення є:

– модуль графічного інтерфейсу (GUI), що реалізований за допомогою бібліотеки Tkinter. Відповідає за взаємодію з користувачем: вибір параметрів експерименту (набір даних, розмір пакету, пристрій, точність, архітектура моделі), відображення результатів, побудову графіків та збереження даних. GUI працює в головному процесі та не блокується під час обчислень завдяки багатопроцесній архітектурі [9];

– модуль обчислень (worker_process), що виконується в окремому процесі, створеному за допомогою модуля multiprocessing. Цей модуль завантажує датасет, будує нейронну мережу (Simple CNN, Deep CNN або Transfer Learning), виконує навчання на вказаному пристрої (CPU або GPU) з заданою точністю (FP32 або FP16) та оцінює отриману модель. Після завершення обчислень результати (точність, втрати, час, історія навчання) передаються назад до GUI через черги [4];

– модуль черг (Queues) – використовує multiprocessing.Queue для обміну даними між процесами;

Існують дві черги: log_queue для передачі повідомлень про хід виконання (логи) від обчислювального процесу до GUI, що дозволяє відображати у вікні логів інформацію про завантаження даних, навчання, завершення епох, а також показники моніторингу ресурсів; result_queue для передачі фінальних результатів експерименту після завершення навчання;

– модуль моніторингу ресурсів, що запускається як окремий процес усередині worker_process. За допомогою бібліотеки psutil отримує дані про завантаження CPU та RAM, а через виклик nvidia–smi (з параметром CREATE_NO_WINDOW) – про завантаження GPU та відеопам’яті. Повідомлення надсилаються до log_queue кожну секунду [10];

– модуль збереження результатів (save_results), що реалізує експорт результатів експерименту у файли форматів RTF, TXT або DOCX. Використовує бібліотеку python–docx для створення документів Word та вбудовані засоби Python для роботи з текстовими файлами;

– модуль дашборду (open_dashboard) забезпечує перегляд історії експериментів, збереженої в базі даних SQLite. Дозволяє порівнювати результати різних запусків (графіки точності та часу), видаляти окремі експерименти або всю історію.

Така модульна структура забезпечує гнучкість, зручність підтримки та можливість розширення функціоналу без внесення змін до вже існуючих компонентів.

3.2 Реалізація графічного інтерфейсу та параметрів експерименту

Графічний інтерфейс застосунку реалізовано за допомогою бібліотеки Tkinter, яка є стандартним інструментом Python для створення десктопних додатків. Інтерфейс складається з головного вікна (рис. 3.2) та вікна експерименту (рис. 3.3).

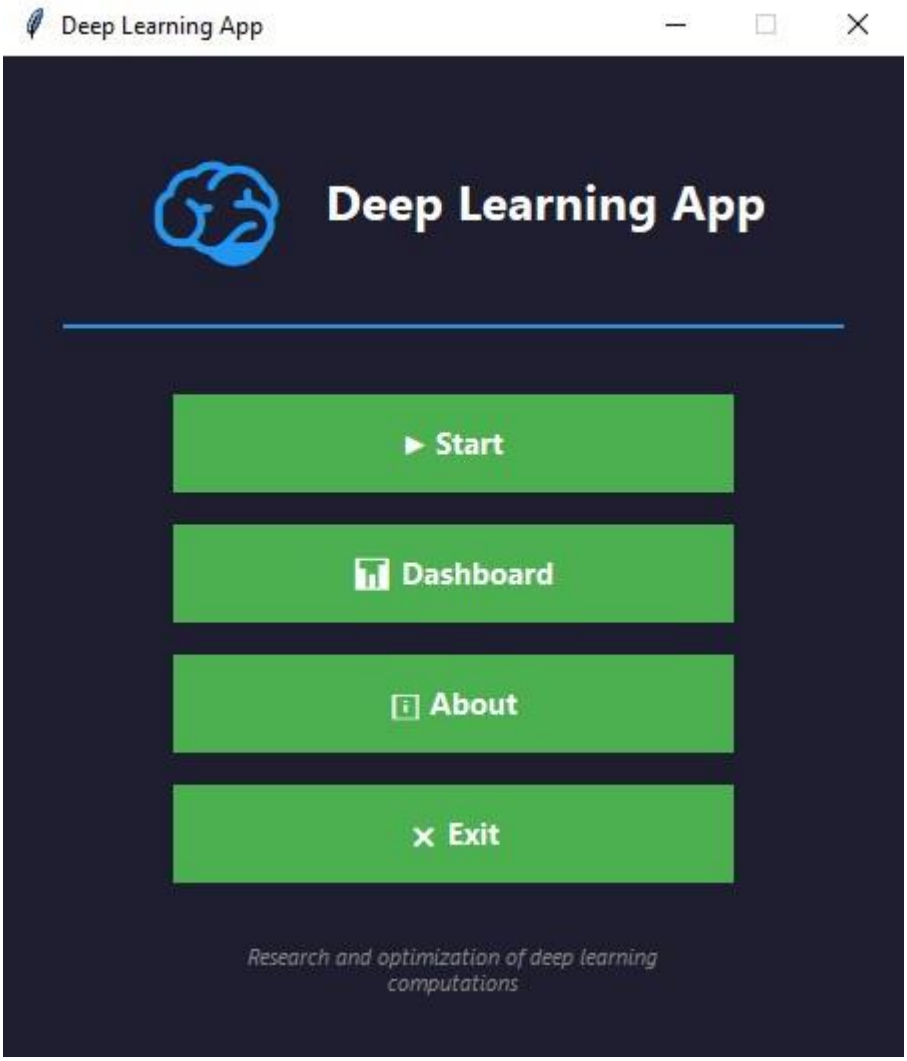


Рисунок 3.2 Головне вікно програми Deep Learning App

Головне вікно містить:

– заголовок з іконкою та назвою програми.

– кнопки Start (перехід до вікна експерименту), Dashboard (перегляд історії), About (інформація про розробника) та Exit (вихід з програми).

– текстовий підвал із описом призначення програми (рис. 3.3).

Інтерфейс виконано в темній кольоровій гамі, що знижує навантаження на зір користувача під час тривалої роботи [9].

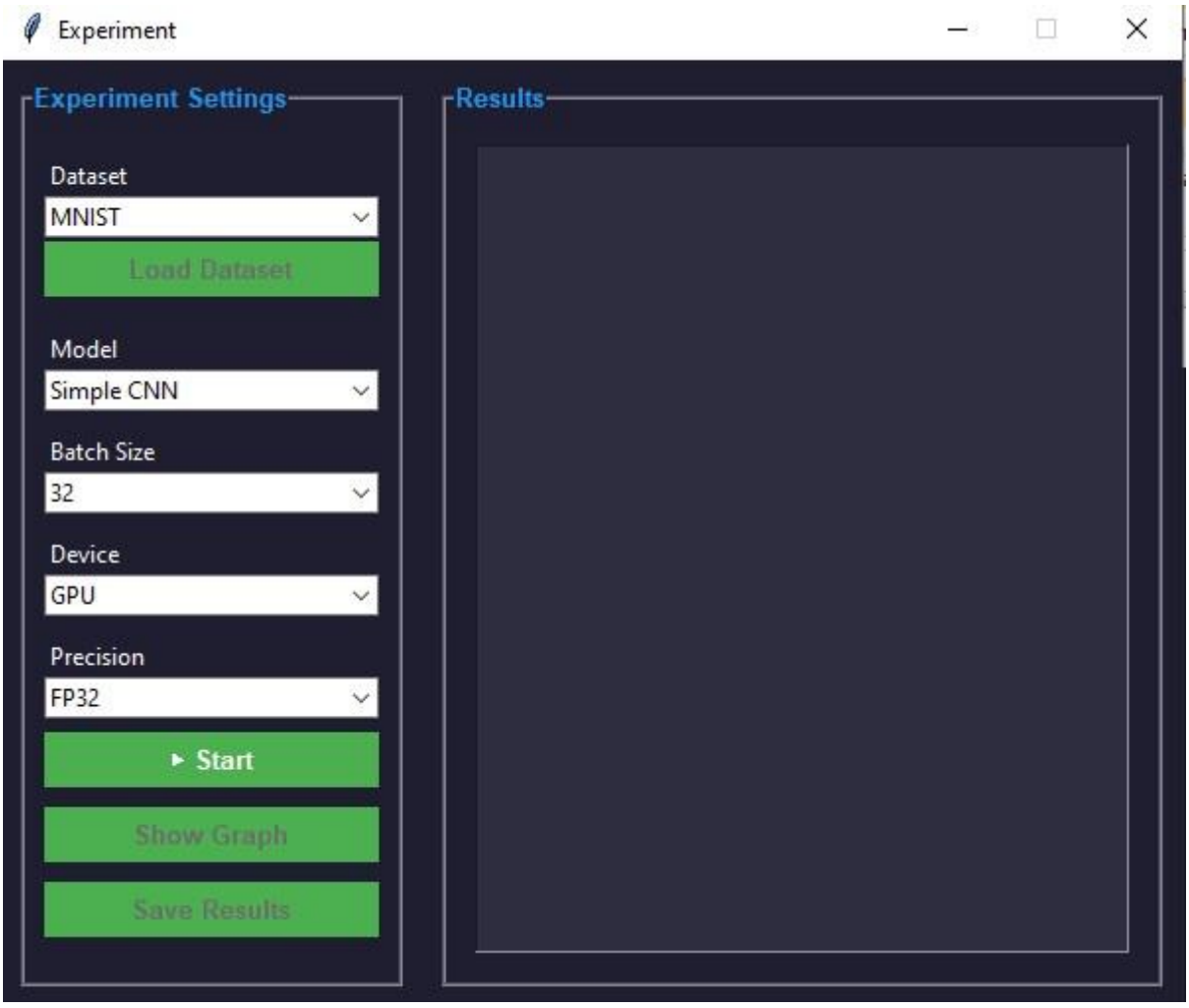


Рисунок 3.3 – Вікно експерименту

Вікно експерименту містить дві основні колонки: ліву та праву.

Ліва колонка «Experiment Settings» об’єднана в рамку LabelFrame і містить наступні елементи керування:

– Dataset – випадаючий список для вибору набору даних: MNIST, Fashion–MNIST, CIFAR–10, CIFAR–100 або CUSTOM;

– Load Dataset – кнопка для вибору папки з власним датасетом, що активується лише при виборі CUSTOM. Структура кастомного датасету

повинна відповідати стандарту TensorFlow: кожна папка всередині кореневої директорії є назвою класу, а всередині неї містяться зображення;

– Model – вибір архітектури нейронної мережі: Simple CNN, Deep CNN або Transfer Learning;

– Batch Size – вибір розміру пакету: 32, 64 або 128;

– Device – вибір обчислювального пристрою: CPU або GPU;

– Precision – вибір точності обчислень: FP32 або FP16;

– кнопки Start (ініціює новий експеримент), Show Graph (містить графічні показники навчання по епохах) та Save Results (забезпечує збереження всієї інформації про проведений експеримент).

Права колонка «Results» містить текстове поле з прокруткою для відображення фінальних результатів експерименту. Параметри експерименту зберігаються у змінних StringVar та передаються до обчислювального процесу через словник config.

Параметри експерименту зберігаються у змінних StringVar та передаються до обчислювального процесу через словник config. Завдяки використанню trace на змінній dataset, кнопка «Load Dataset» стає активною лише при виборі значення «CUSTOM», що запобігає помилкам введення.

3.3 Модуль навчання нейронних мереж

Модуль навчання реалізовано у функції worker_process, яка виконується в окремому процесі. Залежно від вибраних параметрів, будуються різні архітектури нейронних мереж.

Simple CNN – це найпростіша згорткова нейронна мережа, яка добре підходить для базових експериментів. Вона складається з наступних частин:

– два згорткові шари. Кожен з них використовує 32 маленькі фільтри розміром 3×3 пікселі. Ці фільтри «ковзають» по зображенню і виявляють прості ознаки, такі як краї, лінії або плями кольору;

– два шари підвибірки (MaxPooling). Вони зменшують розмір зображення вдвічі, залишаючи найважливішу інформацію. Це прискорює обчислення та допомагає уникнути запам'ятовування зайвих деталей;

– шар вирівнювання (Flatten). Він перетворює двовимірну карту ознак у один довгий вектор, щоб передати його у звичайну нейронну мережу;

– два повнозв'язні шари. Перший має 128 нейронів, які аналізують усі виявлені ознаки разом. Другий (вихідний) шар має кількість нейронів, що дорівнює кількості класів одягу (наприклад, 10 для Fashion MNIST). Він видає ймовірність того, до якого типу одягу належить зображення.

Ця архітектура є базовою для порівняння та підходить для простих датасетів, таких як MNIST та Fashion–MNIST.

Deep CNN – це глибша згорткова нейронна мережа, яка має більше шарів, ніж проста CNN. Вона складається з шести згорткових шарів. У перших двох шарах використовується 64 фільтри, у наступних двох – 128 фільтрів. Більша кількість фільтрів дозволяє мережі виявляти більше різноманітних ознак на зображенні. Між групами згорткових шарів додано шари підвибірки (MaxPooling). Вони зменшують розмір зображень, що прискорює обчислення та допомагає уникнути перенавчання. Після всіх згорткових шарів отримані дані вирівнюються в один рядок (Flatten) і передаються у повнозв'язний шар із 256 нейронами.

Для боротьби з перенавчанням використовується метод dropout з параметром 0,5. Це означає, що під час навчання випадковим чином «вимикається» половина нейронів, що змушує мережу не покладатися на окремі ознаки та стає стійкішою. Останній шар – вихідний, кількість нейронів у ньому дорівнює кількості класів, які потрібно розпізнати (наприклад, 10 для CIFAR–10 або 100 для CIFAR–100). Така глибока архітектура забезпечує вищу точність на складних наборах даних (CIFAR–10, CIFAR–100), але потребує більше часу на навчання та більше пам'яті (як відеопам'яті GPU, так і оперативної пам'яті) [1].

Transfer Learning – це метод, за якого використовується вже готова, заздалегідь навчена нейронна мережа. У нашому випадку це модель MobileNetV2, яку попередньо навчили на величезному наборі зображень (ImageNet) розпізнавати тисячі різних об'єктів (тварини, предмети, рослини тощо). Вона вже «знає», як виділяти важливі візуальні ознаки (краї, форми, текстури). У програмі ми беремо цю готову модель, але замінюємо її останні шари, що відповідають за класифікацію на 1000 загальних категорій, на нові, які будуть навчатися розпізнавати саме наші види одягу. Для цього додаються такі шари:

– шар, що усереднює отримані ознаки;

– один проміжний шар із 128 нейронами;

– вихідний шар із кількістю нейронів, що дорівнює числу класів одягу (наприклад, 10 для Fashion MNIST).

Основна частина моделі MobileNetV2 заморожується – її ваги не змінюються під час навчання. Це дозволяє:

– не навчати модель з нуля, що потребує величезної кількості даних і часу;

– використовувати вже набуті знання про загальні ознаки зображень;

– швидко адаптувати модель навіть до невеликого набору даних (наприклад, кілька тисяч зображень одягу).

Оскільки оригінальна модель очікує кольорові зображення розміром 96×96 пікселів, у програмі передбачено автоматичне змінення розміру будь–якого вхідного зображення та перетворення чорно–білих картинок на кольорові (за необхідності). Це реалізовано за допомогою спеціального шару Lambda [9].

Після того як нейронна мережа побудована, її потрібно підготувати до навчання. Цей етап називається компіляцією. Під час компіляції вказуються три важливі речі:

– оптимізатор – це алгоритм, який налаштовує ваги мережі так, щоб вона вчилася краще. У нашій програмі використовується оптимізатор Adam. Він добре підходить для більшості задач глибинного навчання;

– функція втрат – це показник, який показує, наскільки сильно помиляється мережа. Чим менше значення, тим краще. Для задач класифікації (коли потрібно вибрати один правильний клас з кількох) використовується categorical_crossentropy (категоріальна крос–ентропія);

– метрика – це критерій, який ми хочемо відстежувати під час навчання. У нашому випадку це accuracy (точність) – частка правильно вгаданих прикладів.

Під час навчання програма використовує спеціальний механізм EpochLogger. Після кожної епохи (тобто після того, як мережа перегляне всі навчальні картинки один раз) він надсилає у вікно логів повідомлення про точність на тренувальних і тестових даних. Це дозволяє спостерігати за прогресом навчання в реальному часі.

Крім того, перед навчанням програма вказує, на якому пристрої виконувати обчислення – на центральному або на графічному процесорі. Це робиться за допомогою конструкції tf.device('/CPU:0') або tf.device('/GPU:0'). Вона означає: «Всі операції, що будуть виконані всередині цього блоку, повинні працювати на обраному пристрої». Завдяки цьому можна легко порівняти швидкодію CPU та GPU.

3.4 Розширення: дашборд історії експериментів, порівняння, видалення

Для зручного аналізу результатів експериментів розроблено дашборд (вікно Dashboard), який відображає всю історію запусків, збережену в базі даних SQLite з назвою за замовчуванням experiments.db. Структура бази даних наведена в таблиці 3.1.

Таблиця 3.1 – Схема бази даних experiments.db

Колонка	Тип	Опис
id	INTEGER	Унікальний ідентифікатор (автоінкремент)
dataset	TEXT	Назва використаного датасету
model	TEXT	Тип моделі (Simple CNN, Deep CNN, Transfer Learning)
accuracy	REAL	Точність моделі на тестовій вибірці
loss	REAL	Значення функції втрат
time	REAL	Час навчання (секунди)
logs	TEXT	Текстовий журнал виконання експерименту
history	TEXT	Історія навчання (точність та втрати по епохах) у форматі JSON
created_at	TEXT	Дата та час проведення експерименту

Після кожного завершення експерименту результати автоматично зберігаються в базі даних (рядок INSERT INTO experiments ...). Це дозволяє зберігати історію навіть після перезапуску програми.

Інтерфейс дашборду містить:

– таблицю з колонками: ID, Dataset, Model, Accuracy, Loss, Time (s), Date. Вміст кожної колонки вирівняно по центру, числові значення відформатовано до трьох знаків після коми;

– кнопки Compare Selected, Refresh, Delete Selected, Delete All (розташовані горизонтально на всю ширину вікна).

Функція порівняння (Compare Selected) дозволяє вибрати два або більше експерименти (використовуючи selectmode="extended") та побудувати три порівняльні графіки:

– стовпчаста діаграма точності (Accuracy Comparison);

– стовпчаста діаграма часу виконання (Time Comparison);

– накладені графіки точності по епохах для кожного з вибраних експериментів (якщо історія збережена).

Дані для графіків зчитуються з бази даних, а історія навчання десеріалізується з JSON [2].

Видалення експериментів реалізовано з підтвердженням:

– Delete Selected – видаляє вибрані рядки з таблиці та з бази даних (за ID);

– Delete All – видаляє всі записи з таблиці experiments. Перед виконанням видалення виводиться додаткове попередження.

Після будь–якої операції видалення таблиця оновлюється автоматично.

3.5 Моніторинг у реальному часі та логування

Система моніторингу ресурсів працює в реальному часі та інтегрована як в обчислювальний процес, так і в графічний інтерфейс.

Моніторинг виконується в окремому процесі monitor_resources, який запускається всередині worker_process. Кожну секунду він збирає дані:

– завантаження CPU: psutil.cpu_percent();

– використання RAM: psutil.virtual_memory().percent;

– завантаження GPU та VRAM: через виклик утиліти nvidia–smi з параметром creationflags=subprocess.CREATE_NO_WINDOW , щоб уникнути появи консольних вікон;

– Отримані дані парсяться, формують рядок виду "CPU: 35% | RAM: 42% | GPU: 28% | VRAM: 5.4%" та надсилаються до черги log_queue. Цей рядок потрапляє у вікно логів Logs, яке відкривається під час експерименту.

Логування етапів виконання реалізовано через функцію log(msg), яка надсилає повідомлення до log_queue. Основні етапи, що логуються:

– завантаження датасету;

– побудова моделі;

– початок навчання;

– завершення кожної епохи (за допомогою callback EpochLogger);

– оцінка моделі;

– завершення процесу.

У головному вікні під час експерименту відкривається окреме вікно Logs, в якому текстовий поле відображає всі повідомлення в реальному часі завдяки механізму after (періодична перевірка черги).

Після завершення експерименту результати з’являються в правій колонці вікна експерименту.

3.6 Збереження результатів

Розроблений застосунок надає користувачеві можливість зберігати результати експериментів у трьох форматах: RTF, TXT та DOCX. Вибір формату здійснюється через стандартний діалог filedialog.asksaveasfilename.

Функція save_results формує вміст файлу, який включає:

– рядок–роздільник "===== RESULTS =====";

– точність (accuracy);

– значення функції втрат (loss);

– час виконання (time) у секундах;

– рядок–роздільник "===== LOGS =====";

– Повний журнал логів експерименту (кожен рядок логу, розділений переводами рядків).

Залежно від розширення вибраного файлу виконуються різні дії:

– TXT – звичайний текстовий файл з кодуванням UTF–8. Записується вміст без додаткового форматування;

– DOCX – документ Microsoft Word, створюється за допомогою бібліотеки python–docx. Заголовок "Experiment Results" додається як heading level 0, а основний вміст – як звичайний абзац (параграф) [11];

– RTF – Rich Text Format, який підтримує базове форматування та чудово відображає кирилицю. Формується шляхом додавання RTF–заголовка та переходів рядків за допомогою тегу \par.

Збереження графіків реалізовано у функції show_graph. Після побудови та відображення графіків точності та втрат (два підграфіки поруч)

з’являється кастомне вікно з питанням: "Would you like to save the graph as an image?" ("Ви бажаєте зберегти графік як зображення?") з кнопками Yes/No. При виборі Yes відкривається діалог вибору місця збереження у форматах PNG або JPEG. Графік зберігається з роздільною здатністю 150 DPI та параметром bbox_inches="tight", що усуває зайві поля [2].

3.7 Висновки за розділом 3

У третьому розділі описано програмну реалізацію застосунку для оптимізації обчислень глибинного навчання.

Запропоновано модульну архітектуру, яка включає графічний інтерфейс (Tkinter), обчислювальний процес (worker_process), черги для обміну даними (multiprocessing.Queue), модуль моніторингу ресурсів, модуль збереження результатів та модуль дашборду. Така архітектура забезпечує неблокувальну роботу інтерфейсу під час виконання важких обчислень та коректне перемикання між CPU та GPU [4], [9].

Реалізовано графічний інтерфейс, який дозволяє користувачеві вибирати всі ключові параметри експерименту: набір даних (MNIST, Fashion–MNIST, CIFAR–10, CIFAR–100, CUSTOM), архітектуру моделі (Simple CNN, Deep CNN, Transfer Learning), розмір пакету (32, 64, 128), тип обчислювального пристрою (CPU, GPU) та точність обчислень (FP32, FP16). Інтерфейс виконано в темній кольоровій гамі для зручності роботи.

Розроблено модуль навчання нейронних мереж, який підтримує три архітектури, а також автоматичне використання змішаної точності та прив’язку до конкретного пристрою через tf.device [1], [5], [9].

Створено дашборд історії експериментів на основі SQLite, який дозволяє переглядати всі попередні запуски, порівнювати їх між собою (графіки точності, часу, історії навчання), видаляти окремі експерименти або очищати всю історію. Дані зберігаються постійно.

Реалізовано систему моніторингу ресурсів у реальному часі (CPU, RAM, GPU, VRAM) без появи консольних вікон завдяки використанню subprocess.CREATE_NO_WINDOW. Моніторинг працює в окремому процесі та надсилає дані до вікна логів кожну секунду [10].

Забезпечено збереження результатів експериментів у трьох форматах (TXT, DOCX, RTF), а також збереження графіків у растрових форматах (PNG, JPEG) з можливістю вибору місця збереження через діалогове вікно.

Усі компоненти інтегровані в єдиний програмний застосунок, який дозволяє проводити експерименти з різними конфігураціями навчання, зберігати та аналізувати результати, що створює основу для подальшого розширення функціоналу.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Методика проведення експериментів

Для оцінки ефективності розробленого програмного застосунку та дослідження впливу різних параметрів навчання на продуктивність і точність нейронних мереж було проведено серію експериментів. Усі експерименти виконувалися на одному й тому самому апаратному забезпеченні (Intel Core i9–10900, 32 Гб RAM, NVIDIA GeForce RTX 3060) з використанням програмного застосунку, описаного в розділі 3. Для забезпечення відтворюваності результатів фіксувалися всі параметри навчання та метрики.

Використані датасети:

– MNIST – 70 000 чорно–білих зображень рукописних цифр (28×28 пікселів). Класифікація на 10 класів (цифри від 0 до 9). Цей датасет є простим, дозволяє швидко оцінити базову продуктивність [1];

– CIFAR–10 – 60 000 кольорових зображень (32×32 пікселів), 10 класів (літаки, автомобілі, птахи, коти, олені, собаки, жаби, коні, кораблі, вантажівки). Вважається складнішим за MNIST через кольоровість та різноманітність об’єктів [1];

– власний датасет (CUSTOM) – для демонстрації роботи з кастомними даними було створено невелику вибірку з двох класів: «коти» та «собаки» (по 4000 зображень кожного класу, отриманих з відкритих джерел). Зображення було приведено до розміру 32×32 пікселів (кольорові) та розміщено у відповідні підпапки згідно з вимогами TensorFlow.

Параметри, що варіювалися:

– розмір пакету (batch size): 32, 64, 128;

– тип обчислювального пристрою: CPU та GPU;

– точність обчислень: FP32 (стандартна) та FP16 (змішана точність);

– архітектура моделі: Simple CNN (для всіх експериментів, щоб уникнути впливу складності моделі).

Метрики, що фіксувалися:

– час навчання для трьох епох;

– точність на тестовій вибірці після трьох епох;

– завантаження CPU, RAM, GPU, VRAM, а саме середні значення за час навчання.

Кожен експеримент повторювався три рази, для подальшого аналізу використовувалося середнє арифметичне значення часу та точності.

4.2 Вплив розміру пакету на швидкість та точність

Розмір пакету (batch size) є одним із ключових гіперпараметрів, що впливає як на швидкість навчання, так і на якість кінцевої моделі. Для дослідження цього впливу було проведено експерименти на датасеті CIFAR–10 з використанням GPU, точності FP32 та моделі Simple CNN. Результати наведено в таблиці 4.1.

Таблиця 4.1 – Вплив розміру пакету на час навчання та точність (CIFAR–10, GPU, FP32)

Batch size	Час трьох епох, с	Точність, %	Відносне прискорення	Зміна точності
32	22,80	65,02	1,00×	0 %
64	13,54	63,29	1,68×	–1,73 %
128	9,71	62,67	2,35×	–2,35 %

Аналіз показав такі результати:

– час навчання: збільшення розміру пакету з 32 до 128 дозволяє скоротити час навчання більш ніж у 2,3 рази (з 22,80 с до 9,71 с). Це пояснюється тим, що більше значення краще завантажує обчислювальні ресурси GPU: зменшується кількість ітерацій за епоху, а отже, зменшуються накладні витрати на запуск ядер CUDA та передачу даних між CPU та GPU [4];

– точність: із збільшенням розміру пакету точність дещо знижується – з 65,02 % до 62,67 % (падіння на 2,35 %). Це узгоджується з відомими дослідженнями [5]: надто велике значення згладжує ландшафт функції втрат, що може призводити до «гострих мінімумів» і погіршення узагальнювальної здатності;

– оптимальний вибір: для подальших експериментів було обрано значення розмір пакету саме 64, як компроміс між швидкістю (прискорення 1,68×) та мінімальною втратою точності (1,73 %). Крім того, це значення є стандартним для багатьох досліджень, що полегшує порівняння результатів.

Спостереження з логів моніторингу:

− при розмірі пакету = 32 завантаження GPU коливалося в межах 35–40 %, досягаючи піків 50 %;

− при розмірі пакету = 64 завантаження GPU зросло до 41–45 %, що свідчить про краще використання ресурсів;

− при розмірі пакету = 128 завантаження GPU сягало 51–52 %, що є максимальним для даної моделі.

VRAM при цьому була зайнята приблизно на 89–90 % у всіх випадках, що пояснюється резервуванням пам’яті драйвером. Навіть якщо модель займає менше, то TensorFlow виділяє майже всю доступну пам’ять.

4.3 Порівняння продуктивності CPU та GPU

Для кількісної оцінки прискорення, яке забезпечує використання графічного процесора, було проведено експерименти на датасетах MNIST та CIFAR–10 з розміром пакету 64 та точністю FP32. Результати наведено в таблиці 4.2.

Аналіз показав такі результати:

– прискорення: використання GPU дозволяє прискорити навчання у 3,2–3,4 рази залежно від датасету. Для CIFAR–10 прискорення дещо більше (3,42×), оскільки більший обсяг даних краще використовує паралельні

обчислювальні можливості GPU. Для MNIST прискорення менше (3,24×) через те, що зображення невеликі (28×28) і ступінь паралелізації обмежений;

Таблиця 4.2 – Порівняння часу навчання на CPU та GPU (розміром пакету = 64, FP32)

Датасет	Пристрій	Час трьох епох, с	Точність, %	Прискорення (GPU/CPU)	Середнє завантаження пристрою
MNIST	CPU	43,94	98,95	1×	CPU: 65–85 %
MNIST	GPU	13,55	98,92	3,24×	GPU: 35–45 %
CIFAR–10	CPU	46,28	63,27	1×	CPU: 60–75 %
CIFAR–10	GPU	13,54	63,29	3,42×	GPU: 40–45 %

– точність: точність моделей при навчанні на CPU та GPU практично однакова (різниця менше 0,03 %), що підтверджує коректність реалізації обчислень та відсутність додаткових похибок через використання GPU;

– завантаження ресурсів: при навчанні на CPU завантаження процесора досягало 85–100 % (особливо під час обчислювально–інтенсивних операцій), що свідчить про максимальне використання доступних ресурсів. При навчанні на GPU завантаження CPU було значно нижчим (16–24 %), оскільки основне навантаження перенесено на графічний процесор.

Однак у логах спостерігається незначне завантаження GPU (10–20 %) навіть при виборі пристрою CPU. Це пояснюється тим, що TensorFlow все одно виявляє наявність GPU та може використовувати його для деяких операцій (наприклад, для прискорення окремих математичних функцій). Для повного вимкнення GPU необхідно встановити змінну середовища CUDA_VISIBLE_DEVICES="–1", що не було зроблено в цих експериментах, оскільки метою було порівняння «як є», а не максимально чистого CPU–режиму.

4.4 Ефективність змішаної точності

Змішана точність (mixed precision) використовує 16–бітні числа з рухомою комою (FP16) для більшості обчислень, зберігаючи при цьому 32–бітні копії ваг для оновлень.

Очікувані переваги: зменшення використання пам’яті вдвічі, прискорення обчислень (особливо на GPU з тензорними ядрами).

Для оцінки ефективності цього методу в нашій конфігурації було проведено експерименти на датасетах MNIST та CIFAR–10 з розміром пакету = 64, використовуючи GPU.

Таблиця 4.3 – Порівняння FP32 та FP16 (GPU, розмір пакету = 64)

Датасет	Точність	Час трьох епох, с	Точність моделі, %	Зміна часу	Зміна точності
MNIST	FP32	13,49	99,03	0 %	0 %
MNIST	FP16	21,43	98,98	+58,9 %	–0,05 %
CIFAR–10	FP32	13,26	62,06	0 %	0 %
CIFAR–10	FP16	19,91	64,06	+50,1 %	+2,00 %

Аналіз показав такі результати:

− неочікуване сповільнення: замість прискорення спостерігається збільшення часу навчання на 50–59 % при використанні FP16. Це суперечить теоретичним очікуванням [5]. Можливо причина сповільнення криється в використані малій моделі, наприклад Simple CNN, що має відносно невелику кількість параметрів і не створює достатнього навантаження, щоб виграш від тензорних ядер переважив накладні витрати. Тензорні ядра найефективніші для великих матричних множень (наприклад, у глибоких мережах або з великими розмірами зображень);

− точність: для MNIST точність практично не змінилася (падіння на 0,05 %). Для CIFAR–10 спостерігається навіть невелике підвищення точності (з 62,06 % до 64,06 %). Це може бути пов’язано з випадковою варіацією або з тим, що FP16 діє як форма регуляризації [5];

− рекомендації: для даної архітектури (Simple CNN) використання FP16 не є доцільним, оскільки воно сповільнює навчання без суттєвого покращення точності. Однак для більш глибоких моделей (Deep CNN або Transfer Learning) ефект може бути позитивним, і цей метод слід застосовувати.

4.5 Аналіз завантаження ресурсів

Під час експериментів система моніторингу, вбудована в застосунок, збирала дані про завантаження ресурсів кожну секунду. У таблиці 4.4 наведено середні значення для трьох типів експериментів (MNIST, CIFAR–10, CUSTOM) на GPU з FP32 та розмір пакету = 64.

Таблиця 4.4 – Середнє завантаження ресурсів під час навчання (GPU, FP32, розмір пакету = 64)

Датасет	CPU, %	RAM, %	GPU, %	VRAM, %	Спостереження
MNIST	18,2	49,3	36,8	89,3	Низьке завантаження CPU, VRAM майже повністю зайнята
CIFAR–10	18,0	54,0	41,0	89,2	GPU завантажений краще, ніж для MNIST
CUSTOM	48,7	47,7	20,5	89,1	Високе завантаження CPU через попередню обробку зображень

Аналіз ресурсів для кожного датасету:

– MNIST (28×28, чорно–білий): завантаження GPU становить 36,8 %, що є помірним. VRAM зайнята на 89,3 %, але це здебільшого резервування, а не реальне використання. CPU завантажений лише на 18,2 %, оскільки дані невеликі та швидко передаються;

– CIFAR–10 (32×32, кольоровий): завантаження GPU зростає до 41 % через більший обсяг даних (3 канали замість 1). CPU залишається на низькому рівні (18 %). VRAM – 89,2 %;

– власний датасет CUSTOM (32×32, кольоровий, 8000 зображень сумарно): завантаження CPU значно вище (48,7 %), оскільки зчитування зображень з диску, їх декодування, зміна розміру та нормалізація виконуються на центральному процесорі. GPU завантажений лише на 20,5 %, що свідчить про те, що вузьким місцем є швидкість подачі даних, а не обчислення. Це типова проблема для малих датасетів.

Однак у всіх експериментах використання VRAM становило близько 89 %. Це не означає, що модель споживає майже 11 Гб з 12 доступних. Насправді TensorFlow за замовчуванням резервує майже всю відеопам’ять для уникнення фрагментації та прискорення виділення. Фактичне використання моделі Simple CNN становить лише кілька сотень мегабайт, але драйвер показує резервовану пам’ять. Це нормальна поведінка для TensorFlow [2].

4.6 Висновки за розділом 4

У четвертому розділі проведено експериментальне дослідження впливу різних параметрів навчання на продуктивність та точність нейронних мереж. На основі отриманих даних можна зробити такі висновки.

Збільшення batch size з 32 до 128 скорочує час навчання у 2,35 рази при незначній втраті точності (близько 2,35 %). Оптимальним для подальших експериментів є batch size = 64, який забезпечує баланс між швидкістю та точністю.

Використання графічного процесора прискорює навчання у 3,2–3,4 рази порівняно з центральним процесором при збереженні точності (різниця менше 0,1 %). Це підтверджує доцільність застосування GPU для навчання нейронних мереж, особливо на датасетах середнього розміру [3], [4].

Для моделі Simple CNN застосування FP16 призвело до сповільнення на 50–59 % без суттєвої зміни точності. Це пов’язано з малим розміром моделі, через що переваги тензорних ядер не реалізуються, а накладні витрати на динамічне масштабування стають домінуючими. Для глибших моделей рекомендується повторити експерименти [5].

Система моніторингу дозволила виявити, що VRAM резервується майже повністю (близько 89 %) через політику TensorFlow; для малих датасетів (власний) вузьким місцем є завантаження даних (CPU до 49 %), а не обчислення (GPU лише 20 %); для великих датасетів таких, як CIFAR–10, GPU завантажений на 41 %, що є прийнятним показником.

На основі експериментів сформульовано такі рекомендації для користувачів застосунку: для швидких експериментів обирати GPU та розмір пакету 64–128; для досягнення максимальної точності використовувати FP32 та розмір пакету 32; для кастомних датасетів з невеликою кількістю зображень слід забезпечити швидке завантаження даних, наприклад, використовувати SSD, попередньо завантажувати дані в пам’ять; FP16 варто застосовувати лише для глибоких моделей таких, як Deep CNN та Transfer Learning, на великих датасетах.

Розроблений програмний застосунок довів свою працездатність: він дозволяє проводити експерименти з різними конфігураціями, автоматично фіксувати результати та візуалізувати процес навчання. Отримані дані можуть бути використані для подальших досліджень у сфері оптимізації обчислень глибинного навчання.

ВИСНОВКИ

У результаті виконання дипломної роботи досягнуто поставленої мети – підвищення швидкості навчання нейронних мереж шляхом використання апаратного прискорення GPU та методів оптимізації обчислень. Розроблено універсальний програмний застосунок, який дозволяє проводити експериментальні дослідження впливу різних параметрів навчання (набір даних, розмір пакету, тип обчислювального пристрою, точність обчислень, архітектура моделі) на продуктивність та точність нейронних мереж.

У ході роботи виконано всі поставлені завдання. Проведено аналіз сучасних підходів до оптимізації обчислень у глибинному навчанні, досліджено можливості використання CPU та GPU для навчання нейронних мереж. Розроблено архітектуру програмного застосунку на основі багатопроцесного підходу, що забезпечує неблокувальну роботу графічного інтерфейсу під час виконання ресурсомістких обчислень. Впроваджено методи оптимізації: вибір розміру пакету, змішану точність, моніторинг завантаження ресурсів у реальному часі без появи консольних вікон. Реалізовано функції візуалізації процесу навчання, збереження результатів експериментів у форматах RTF, TXT, DOCX та збереження історії експериментів у базі даних SQLite з можливістю порівняння та видалення через дашборд. Проведено серію експериментів на трьох типах датасетів.

На основі експериментальних даних встановлено, що збільшення розміру пакету з 32 до 128 скорочує час навчання у 2,35 рази при незначній втраті точності (близько 2,35 %); оптимальним є розмір пакету = 64. Використання GPU забезпечує прискорення навчання у 3,2–3,4 рази порівняно з CPU при збереженні точності (різниця менше 0,1 %). Застосування змішаної точності FP16 для моделі Simple CNN призвело до сповільнення на 50–59 % без суттєвої зміни точності через малий розмір моделі, що свідчить про необхідність використання цього методу лише для глибоких архітектур на великих датасетах.

Практичне значення роботи полягає у створенні готового до використання програмного застосунку, який може бути застосований у навчальному процесі закладів вищої освіти для підготовки фахівців з комп’ютерних наук та штучного інтелекту, а також у дослідницьких центрах розробки систем штучного інтелекту для проведення експериментів з оптимізації навчання нейронних мереж.

Наукова новизна роботи полягає в розробці методу багатопроцесного виконання експериментів з ізоляцією обчислень від графічного інтерфейсу, системи моніторингу ресурсів без консольних вікон на основі виклику nvidia–smi з прихованим вікном, а також дашборду для порівняння історії експериментів із візуалізацією.

На основі проведених досліджень сформульовано практичні рекомендації для користувачів застосунку: для швидких експериментів обирати GPU та розмір пакету 64–128; для досягнення максимальної точності використовувати FP32 та розмір пакету 32; для кастомних датасетів з невеликою кількістю зображень забезпечити швидке завантаження даних (наявність SSD, попереднє завантаження в пам’ять); точність FP16 варто застосовувати лише для глибоких моделей, таких як Deep CNN та Transfer Learning, на великих датасетах.

Таким чином, усі завдання дипломної роботи виконано в повному обсязі, мету роботи досягнуто. Розроблений програмний застосунок є працездатним, ефективним та може бути рекомендований до впровадження в навчальний процес та дослідницьку діяльність. Перспективними напрямками подальшого розвитку є додавання підтримки багатографічного навчання, автоматичного підбору гіперпараметрів та інтеграція з хмарними сервісами для розподілених обчислень.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Goodfellow I., Bengio Y., Courville A. Deep learning. Cambridge : MIT Press, 2016. 781 p. URL: https://www.deeplearningbook.org/ (date of access: 10.05.2026).

2. Abadi M., Agarwal A., Barham P., Brevdo E., Chen Z., Citro C., et al. TensorFlow: large–scale machine learning on heterogeneous distributed systems. arXiv preprint. 2016. URL: https://arxiv.org/abs/1603.04467 (date of access: 10.05.2026).

3. NVIDIA Corporation. CUDA C++ programming guide. Santa Clara : NVIDIA, 2023. 350 p. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (date of access: 10.05.2026).

4. TensorFlow. TensorFlow core guide. Mountain View : Google, 2024. URL: https://www.tensorflow.org/guide (date of access: 10.05.2026).

5. Micikevicius P., Narang S., Alben J., Diamos G., Elsen E., Garcia D., et al. Mixed precision training. International Conference on Learning Representations (ICLR). 2018. URL: https://arxiv.org/abs/1710.03740 (date of access: 10.05.2026).

6. Чумаченко С. В., Ковальчук А. М. Оптимізація обчислень нейронних мереж на графічних процесорах. Вісник Національного університету «Львівська політехніка». Серія: Комп’ютерні науки та інформаційні технології. 2022. № 12. С. 45–52.

7. Paszke A., Gross S., Massa F., Lerer A., Bradbury J., Chanan G., et al. PyTorch: an imperative style, high–performance deep learning library. Advances in Neural Information Processing Systems. 2019. Vol. 32. P. 8024–8035.

8. Keras team. Keras API documentation. 2024. URL: https://keras.io/api/ (date of access: 10.05.2026).

9. Howard A. G., Zhu M., Chen B., Kalenichenko D., Wang W., Weyand T., et al. MobileNets: efficient convolutional neural networks for mobile vision applications. arXiv preprint. 2017. URL: https://arxiv.org/abs/1704.04861 (date of access: 10.05.2026).

10. Psutil development team. psutil – cross platform system monitoring library. 2024. URL: https://psutil.readthedocs.io/ (date of access: 10.05.2026).

11. McKinney W. Python for data analysis : data wrangling with Pandas,NumPy, and IPython. 3rd ed. Sebastopol : O’Reilly Media, 2022. 576 p.

ДОДАТОК А

Технічне завдання

Вступ

Дане технічне завдання поширюється на розробку програмного застосунку для дослідження та оптимізації обчислювальних процесів глибинного навчання на GPU та багатоядерних системах. Програмний застосунок призначений для проведення експериментів з навчання нейронних мереж з варіюванням параметрів (набір даних, розмір пакету, тип обчислювального пристрою, точність обчислень, архітектура моделі), а також для візуалізації результатів та збереження історії експериментів.

А.1 Підстави для розробки

Підставою для розробки є завдання на дипломну кваліфікаційну роботу бакалавра на тему «Оптимізація обчислювальних процесів глибинного навчання на GPU та багатоядерних системах», затверджене кафедрою програмних засобів Національного університету «Запорізька політехніка», згідно з Наказом № 139 від 07.04.2026.

А.2 Призначення розробки

Програмний застосунок призначений для дослідження та оптимізації обчислювальних процесів глибинного навчання. Він забезпечує можливість проведення експериментів з навчання нейронних мереж на різних обчислювальних пристроях (CPU та GPU) з варіюванням параметрів навчання.

Застосунок може використовуватися у таких сферах:

– навчальний процес у закладах вищої освіти для підготовки фахівців з комп’ютерних наук та штучного інтелекту;

– дослідження в центрах розробки систем штучного інтелекту для оптимізації навчання нейронних мереж на великих наборах даних;

– експериментальні дослідження впливу різних параметрів на продуктивність та точність нейронних мереж.

А.3.1 Вимоги до функціональних характеристик

Програмний застосунок повинен забезпечувати виконання наступних функцій:

– вибір параметрів експерименту: вибір набору даних (MNIST, Fashion–MNIST, CIFAR–10, CIFAR–100, CUSTOM); вибір архітектури нейронної мережі (Simple CNN, Deep CNN, Transfer Learning); вибір розміру пакету (batch size: 32, 64, 128); вибір обчислювального пристрою (CPU або GPU); вибір точності обчислень (FP32 або FP16); можливість завантаження власного датасету (CUSTOM).

– запуск експерименту:запуск навчання нейронної мережі в окремому процесі (без блокування графічного інтерфейсу); відображення процесу навчання у реальному часі (логи, моніторинг ресурсів).

– моніторинг ресурсів: відображення завантаження CPU, RAM, GPU, VRAM у реальному часі (оновлення кожну секунду); відсутність додаткових консольних вікон під час моніторингу.

– візуалізація результатів: побудова графіків точності (accuracy) та функції втрат (loss) для тренувальної та валідаційної вибірок; можливість збереження графіків у форматах PNG або JPEG.

– збереження результатів: збереження результатів експерименту у форматах RTF, TXT, DOCX; автоматичне збереження історії експериментів у базі даних SQLite.

– дашборд історії експериментів: перегляд списку всіх проведених експериментів (ID, датасет, модель, точність, втрати, час, дата); порівняння вибраних експериментів (графіки точності та часу); видалення окремих експериментів або всієї історії.

– довідкова інформація: відображення інформації про розробника та призначення програми.

А.3.2 Вимоги до надійності

Програмний застосунок повинен:

– коректно обробляти помилки введення даних користувачем (наприклад, відсутність шляху до кастомного датасету);

– коректно завершувати обчислювальний процес після завершення експерименту, звільняючи пам’ять GPU;

– забезпечувати стабільну роботу при повторних запусках експериментів;

– не допускати запуску кількох експериментів одночасно (блокування кнопок під час виконання);

– забезпечувати коректне перемикання між CPU та GPU без конфліктів.

А.3.3 Умови експлуатації

Програмний застосунок призначений для експлуатації на персональних комп’ютерах під керуванням операційної системи Microsoft Windows 10 або новіших версій.

Необхідне програмне забезпечення:

– Python 3.10;

– встановлені бібліотеки: TensorFlow 2.10.0, psutil, matplotlib, python–docx, tkinter (входить до стандартної бібліотеки);

– для роботи з GPU: драйвери NVIDIA, CUDA Toolkit 11.8, cuDNN 8.9.

А.3.4 Вимоги до складу та параметрів технічних засобів

Мінімальні системні вимоги для роботи програми:

– процесор: не менше 4 ядер, частота не менше 2 ГГц;

– оперативна пам’ять: не менше 8 Гб (рекомендовано 16 Гб);

– вільне місце на диску: не менше 5 Гб (для збереження датасетів та результатів);

– графічний процесор: NVIDIA з підтримкою CUDA (об’єм відеопам’яті не менше 4 Гб, рекомендовано 8–12 Гб);

– операційна система: Windows 10 64–bit або новіше.

А.3.5 Вимоги до маркування та упакування

Програмний застосунок може постачатися у вигляді:

– вихідного коду (файли з розширенням .py);

– виконуваного файлу (.exe), створеного за допомогою PyInstaller.

На упаковці (або у файлі README) повинна бути зазначена назва програми: «Deep Learning App – інструмент для оптимізації обчислень глибинного навчання».

А.3.6 Вимоги до транспортування та зберігання

Програмний застосунок не має спеціальних вимог до транспортування та зберігання. Допускається передача на електронних носіях (флеш–накопичувачах, оптичних дисках) або через мережу Інтернет (електронна пошта, хмарні сховища).

А.4 Стадії та етапи розробки

Етапи розробки:

– постановка завдання роботи (1 тиждень): формулювання мети, завдань, об’єкта, предмета дослідження. Складання технічного завдання та календарного плану;

– аналіз предметної області (1 тиждень): дослідження методів оптимізації обчислень глибинного навчання, порівняльний аналіз CPU та GPU, огляд існуючих програмних фреймворків;

– вибір мови програмування та технологій розробки (2 тиждень): обґрунтування вибору Python, бібліотек TensorFlow, Keras, Tkinter, Matplotlib, psutil, а також інструментів багатопроцесного програмування;

– розробка архітектури програми (2 тиждень): проєктування модульної структури застосунку (GUI, worker_process, черги, моніторинг, дашборд);

– розробка програми (3–4 тижні): кодування основних модулів: графічного інтерфейсу, обчислювального процесу, системи моніторингу ресурсів, дашборду історії експериментів, функцій збереження результатів;

– тестування та експериментальне дослідження (5 тиждень): проведення серії експериментів з різними параметрами, аналіз отриманих даних, оцінка продуктивності та точності;

– оформлення пояснювальної записки та додатків (6 тиждень): написання вступу, чотирьох розділів, висновків, списку літератури, підготовка додатків (текст програми, технічне завдання, слайди презентації);

– нормоконтроль та рецензування (7 тиждень): перевірка роботи на відповідність вимогам ДСТУ 3008:2015, отримання рецензії та відгуку керівника;

– захист роботи (8 тиждень): підготовка презентації та доповіді, публічний захист дипломної роботи перед екзаменаційною комісією.

А.5 Порядок контролю та приймання

Приймання програмного застосунку здійснюється на основі:

– аналізу відповідності реалізованого функціоналу вимогам даного технічного завдання;

– результатів тестування програмного застосунку (перевірка основних функцій: запуск експерименту, вибір параметрів, візуалізація графіків, збереження результатів);

– перевірки коректності роботи на різних обчислювальних пристроях (CPU та GPU);

– перевірки стабільності роботи графічного інтерфейсу під час тривалих обчислень.

Контроль розробки здійснюється керівником дипломної роботи відповідно до календарного плану. Остаточне приймання роботи виконується комісією на захисті дипломної кваліфікаційної роботи.

ДОДАТОК Б

Текст програми

Б.1 Текст файлу DeepLearningApp.py

```
import tkinter as tk
from tkinter import ttk, messagebox, filedialog
import time
import multiprocessing
import os
import sqlite3
from datetime import datetime
import json
import psutil
import subprocess

import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist, cifar10, fashion_mnist, cifar100
from docx import Document

# ============================
# COLORS
# ============================
BG_COLOR = "#1e1e2f"
BTN_COLOR = "#4CAF50"
BTN_HOVER = "#45a049"
TEXT_COLOR = "#ffffff"
ACCENT = "#2196F3"

# ============================
# DATABASE INIT
# ============================
def init_db():
    conn = sqlite3.connect("experiments.db")
    cursor = conn.cursor()

    cursor.execute("""
    CREATE TABLE IF NOT EXISTS experiments (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        dataset TEXT,
        model TEXT,
        accuracy REAL,
        loss REAL,
        time REAL,
        logs TEXT,
```

```
        history TEXT,
        created_at TEXT
    )
    """)

    cursor.execute("PRAGMA table_info(experiments)")
    columns = [col[1] for col in cursor.fetchall()]
    if "history" not in columns:
        cursor.execute("ALTER TABLE experiments ADD COLUMN history TEXT")
        print("✓ Added history column to experiments table")

    cursor.execute("CREATE INDEX IF NOT EXISTS idx_created_at ON experiments(created_at)")
    conn.commit()
    conn.close()

# ==========================
# BUTTON STYLE
# ==========================
def styled_button(parent, text, command):
    btn = tk.Button(parent,
                    text=text,
                    command=command,
                    bg=BTN_COLOR,
                    fg=TEXT_COLOR,
                    activebackground=BTN_HOVER,
                    width=20,
                    relief="flat",
                    font=("Arial", 10, "bold"))

    def on_enter(e):
        btn['bg'] = BTN_HOVER
    def on_leave(e):
        btn['bg'] = BTN_COLOR

    btn.bind("<Enter>", on_enter)
    btn.bind("<Leave>", on_leave)
    return btn

# ==========================
# RESOURCE MONITOR
# ==========================
def monitor_resources(log_queue, stop_flag):
    while not stop_flag.is_set():
```

```
        cpu = psutil.cpu_percent()
        ram = psutil.virtual_memory().percent
        # GPU через nvidia–smi без вікна
        try:
            cmd = ['nvidia–smi', '—query–gpu=utilization.gpu,memory.used,memory.total', '—format=csv,noheader,nounits']
            result = subprocess.run(cmd, capture_output=True, text=True, creationflags=subprocess.CREATE_NO_WINDOW)
            if result.returncode == 0:
                gpu_line = result.stdout.strip().split(',')
                gpu_load = int(gpu_line[0].strip())
                mem_used = int(gpu_line[1].strip())
                mem_total = int(gpu_line[2].strip())
                gpu_mem = (mem_used / mem_total) * 100
                msg = f"📊 CPU: {cpu}% | RAM: {ram}% | GPU: {gpu_load:.1f}% | VRAM: {gpu_mem:.1f}%"
            else:
                msg = f"📊 CPU: {cpu}% | RAM: {ram}% | GPU: N/A"
        except:
            msg = f"📊 CPU: {cpu}% | RAM: {ram}% | GPU: N/A"
        log_queue.put(msg)
        time.sleep(1)

# ============================
# WORKER PROCESS
# ============================
def worker_process(config, result_queue, log_queue):
    def log(msg):
        log_queue.put(msg)

    stop_flag = multiprocessing.Event()
    monitor = multiprocessing.Process(
        target=monitor_resources,
        args=(log_queue, stop_flag)
    )
    monitor.start()

    start_time = time.time()

    log("🚀 Start")
    log(f"Dataset: {config['dataset']}")
    log(f"Device: {config['device']}")
    log(f"Precision: {config['precision']}")
    log(f"Model: {config['model_type']}")
```

```
from tensorflow.keras import mixed_precision # type: ignore

if config["precision"] == "FP16":
    mixed_precision.set_global_policy('mixed_float16')
    log("FP16 enabled")
else:
    mixed_precision.set_global_policy('float32')
    log("FP32 enabled")

try:
    log("Loading dataset...")

    if config["dataset"] == "MNIST":
        (x_train, y_train), (x_test, y_test) = mnist.load_data()
        x_train = x_train.reshape(–1,28,28,1)/255.0
        x_test = x_test.reshape(–1,28,28,1)/255.0
        num_classes = 10

    elif config["dataset"] == "Fashion–MNIST":
        (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
        x_train = x_train.reshape(–1,28,28,1)/255.0
        x_test = x_test.reshape(–1,28,28,1)/255.0
        num_classes = 10

    elif config["dataset"] == "CIFAR–10":
        (x_train, y_train), (x_test, y_test) = cifar10.load_data()
        x_train = x_train/255.0
        x_test = x_test/255.0
        num_classes = 10

    elif config["dataset"] == "CIFAR–100":
        (x_train, y_train), (x_test, y_test) = cifar100.load_data()
        x_train = x_train/255.0
        x_test = x_test/255.0
        num_classes = 100

    elif config["dataset"] == "CUSTOM":
        data_dir = config["data_path"]
        if not os.path.exists(data_dir):
            log("❌ Dataset path not found!")
            return

        train_ds = tf.keras.utils.image_dataset_from_directory(
            data_dir,
```

```
            validation_split=0.2,
            subset="training",
            seed=123,
            image_size=(32, 32),
            batch_size=config["batch"]
        )
        val_ds = tf.keras.utils.image_dataset_from_directory(
            data_dir,
            validation_split=0.2,
            subset="validation",
            seed=123,
            image_size=(32, 32),
            batch_size=config["batch"]
        )
        class_names = train_ds.class_names
        num_classes = len(class_names)
        log(f"Classes: {class_names}")
        train_ds = train_ds.map(lambda x, y: (x/255.0, tf.one_hot(y, num_classes)))
        val_ds = val_ds.map(lambda x, y: (x/255.0, tf.one_hot(y, num_classes)))

    log("Building model...")
    if config["dataset"] in ["CIFAR–10", "CIFAR–100", "CUSTOM"]:
        input_shape = (32,32,3)
    else:
        input_shape = (28,28,1)

    if config["model_type"] == "Simple CNN":
        model = models.Sequential([
            layers.Conv2D(32, (3,3), activation='relu', input_shape=input_shape),
            layers.MaxPooling2D(),
            layers.Conv2D(32, (3,3), activation='relu'),
            layers.MaxPooling2D(),
            layers.Flatten(),
            layers.Dense(128, activation='relu'),
            layers.Dense(num_classes, activation='softmax')
        ])
    elif config["model_type"] == "Deep CNN":
        model = models.Sequential([
            layers.Conv2D(64, (3,3), activation='relu', input_shape=input_shape),
            layers.Conv2D(64, (3,3), activation='relu'),
            layers.MaxPooling2D(),
            layers.Conv2D(128, (3,3), activation='relu'),
            layers.Conv2D(128, (3,3), activation='relu'),
            layers.MaxPooling2D(),
            layers.Flatten(),
```

```
            layers.Dense(256, activation='relu'),
            layers.Dropout(0.5),
            layers.Dense(num_classes, activation='softmax')
        ])
    elif config["model_type"] == "Transfer Learning":
        base_model = tf.keras.applications.MobileNetV2(
            input_shape=(96,96,3),
            include_top=False,
            weights='imagenet'
        )
        base_model.trainable = False
        model = models.Sequential([
            layers.Resizing(96, 96),
            layers.Lambda(lambda x: tf.image.grayscale_to_rgb(x) if x.shape[–1] ==
1 else x),
            base_model,
            layers.GlobalAveragePooling2D(),
            layers.Dense(128, activation='relu'),
            layers.Dense(num_classes, activation='softmax')
        ])

    model.compile(optimizer='adam',                loss='categorical_crossentropy',
metrics=['accuracy'])
    log("Training...")

    class EpochLogger(tf.keras.callbacks.Callback):
        def __init__(self, log_func, total_epochs=3):
            super().__init__()
            self.log_func = log_func
            self.total_epochs = total_epochs
        def on_epoch_end(self, epoch, logs=None):
            acc = logs.get('accuracy', 0.0)
            val_acc = logs.get('val_accuracy', 0.0)
            self.log_func(f"📊 Epoch {epoch+1}/{self.total_epochs} – accuracy:
{acc:.4f}, val_accuracy: {val_acc:.4f}")

    epoch_logger = EpochLogger(log, total_epochs=3)

    with tf.device('/CPU:0' if config["device"] == "CPU" else '/GPU:0'):
        if config["dataset"] == "CUSTOM":
            history = model.fit(
                train_ds, epochs=3, validation_data=val_ds, verbose=0,
                callbacks=[epoch_logger]
            )
            loss, acc = model.evaluate(val_ds, verbose=0)
```

```
            else:
                y_train = tf.keras.utils.to_categorical(y_train, num_classes)
                y_test = tf.keras.utils.to_categorical(y_test, num_classes)
                history = model.fit(
                    x_train, y_train, epochs=3, batch_size=config["batch"],
                    validation_data=(x_test, y_test), verbose=0,
                    callbacks=[epoch_logger]
                )
                loss, acc = model.evaluate(x_test, y_test, verbose=0)

            stop_flag.set()
            monitor.terminate()
            log("Done ✓")

            result_queue.put({
                "accuracy": float(acc),
                "loss": float(loss),
                "time": time.time() – start_time,
                "history": history.history
            })
            log("PROCESS FINISHED")

        except Exception as e:
            log(f"❌ ERROR: {str(e)}")
            import traceback
            log(traceback.format_exc())
            stop_flag.set()
            monitor.terminate()

    # ==========================
    # GRAPH
    # ==========================
    def show_graph(history):
        if not history:
            messagebox.showwarning("Warning", "No history data to display!")
            return

        has_val_acc = 'val_accuracy' in history
        has_val_loss = 'val_loss' in history

        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))
        fig.suptitle('Training Progress', fontsize=16)

        epochs = range(1, len(history.get('accuracy', [])) + 1)
```

```
if 'accuracy' in history:
    ax1.plot(epochs, history['accuracy'], 'b–', label='Train Accuracy')
if has_val_acc:
    ax1.plot(epochs, history['val_accuracy'], 'r–', label='Validation Accuracy')
ax1.set_title('Accuracy')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Accuracy')
ax1.legend()
ax1.grid(True)

if 'loss' in history:
    ax2.plot(epochs, history['loss'], 'b–', label='Train Loss')
if has_val_loss:
    ax2.plot(epochs, history['val_loss'], 'r–', label='Validation Loss')
ax2.set_title('Loss')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Loss')
ax2.legend()
ax2.grid(True)

plt.tight_layout()
plt.show()

answer = False
dialog = tk.Toplevel()
dialog.title("Save Graph")
dialog.geometry("300x120")
dialog.configure(bg=BG_COLOR)
dialog.transient()
dialog.grab_set()
dialog.resizable(False, False)

label = tk.Label(dialog, text="Would you like to save the graph as an image?",
                 bg=BG_COLOR, fg=TEXT_COLOR, wraplength=250)
label.pack(pady=15)

def on_yes():
    nonlocal answer
    answer = True
    dialog.destroy()

def on_no():
    nonlocal answer
    answer = False
```

```
        dialog.destroy()

    btn_yes = tk.Button(dialog, text="Yes", command=on_yes,
                        bg=BTN_COLOR, fg=TEXT_COLOR, width=10)
    btn_yes.pack(side="left", padx=20, pady=10)

    btn_no = tk.Button(dialog, text="No", command=on_no,
                       bg=BTN_COLOR, fg=TEXT_COLOR, width=10)
    btn_no.pack(side="right", padx=20, pady=10)

    # Center the dialog
    dialog.update_idletasks()
    x = (dialog.winfo_screenwidth() // 2) – (dialog.winfo_width() // 2)
    y = (dialog.winfo_screenheight() // 2) – (dialog.winfo_height() // 2)
    dialog.geometry(f"+{x}+{y}")

    dialog.wait_window()

    if answer:
        file_path = filedialog.asksaveasfilename(
            defaultextension=".png",
            filetypes=[("PNG image", "*.png"), ("JPEG image", "*.jpg")]
        )
        if file_path:
            fig.savefig(file_path, dpi=150, bbox_inches="tight")
            messagebox.showinfo("Saved", f"Graph saved to {file_path}")

# ============================
# SAVE RESULTS
# ============================
def save_results(result, logs, config):
    file = filedialog.asksaveasfilename(
        defaultextension=".rtf",
        filetypes=[
            ("RTF file", "*.rtf"),
            ("Text file", "*.txt"),
            ("Word Document", "*.docx")
        ]
    )
    if not file:
        return

    content = f"""===== RESULTS =====
Accuracy: {result['accuracy']}
Loss: {result['loss']}
```

```
Time: {result['time']} seconds

===== LOGS =====
{chr(10).join(logs)}
"""

    if file.endswith(".txt"):
        with open(file, "w", encoding="utf–8") as f:
            f.write(content)

    elif file.endswith(".docx"):
        doc = Document()
        doc.add_heading("Experiment Results", 0)
        doc.add_paragraph(content)
        doc.save(file)

    elif file.endswith(".rtf"):
        rtf_header = r"{\rtf1\ansi\deff0\uc1\deftab720{\fonttbl{\f0\fswiss
Arial;}}{\colortbl;\red0\green0\blue0;}\viewkind4\uc1\pard\lang1033\fs20"
        rtf_footer = r"}"
        rtf_body = content.replace("\n", "\\par ")
        rtf_content = rtf_header + " " + rtf_body + " " + rtf_footer
        with open(file, "w", encoding="utf–8") as f:
            f.write(rtf_content)

    messagebox.showinfo("Saved", "Saved successfully!")

# ============================
# DASHBOARD WINDOW
# ============================
def open_dashboard():
    win = tk.Toplevel(root)
    win.title("Dashboard – Experiment History")
    win.geometry("900x650")
    win.configure(bg=BG_COLOR)

    frame_tree = tk.Frame(win, bg=BG_COLOR)
    frame_tree.pack(fill="both", expand=True, padx=10, pady=10)

    columns = ("id", "dataset", "model", "accuracy", "loss", "time", "created_at")
    tree = ttk.Treeview(frame_tree, columns=columns, show="headings",
selectmode="extended")

    tree.heading("id", text="ID", anchor="center")
    tree.heading("dataset", text="Dataset", anchor="center")
```

```
    tree.heading("model", text="Model", anchor="center")
    tree.heading("accuracy", text="Accuracy", anchor="center")
    tree.heading("loss", text="Loss", anchor="center")
    tree.heading("time", text="Time (s)", anchor="center")
    tree.heading("created_at", text="Date", anchor="center")

    tree.column("id", width=40, anchor="center")
    tree.column("dataset", width=100, anchor="center")
    tree.column("model", width=120, anchor="center")
    tree.column("accuracy", width=80, anchor="center")
    tree.column("loss", width=80, anchor="center")
    tree.column("time", width=80, anchor="center")
    tree.column("created_at", width=140, anchor="center")

    scrollbar = ttk.Scrollbar(frame_tree, orient="vertical", command=tree.yview)
    tree.configure(yscrollcommand=scrollbar.set)
    tree.pack(side="left", fill="both", expand=True)
    scrollbar.pack(side="right", fill="y")

    def refresh_table():
        for item in tree.get_children():
            tree.delete(item)
        conn = sqlite3.connect("experiments.db")
        cursor = conn.cursor()
        cursor.execute("SELECT id, dataset, model, accuracy, loss, time, created_at
FROM experiments ORDER BY created_at DESC")
        rows = cursor.fetchall()
        conn.close()
        for row in rows:
            formatted_row = (
                row[0],
                row[1],
                row[2],
                f"{row[3]:.3f}" if isinstance(row[3], (int, float)) else row[3],
                f"{row[4]:.3f}" if isinstance(row[4], (int, float)) else row[4],
                f"{row[5]:.3f}" if isinstance(row[5], (int, float)) else row[5],
                row[6]
            )
            tree.insert("", "end", values=formatted_row)

    refresh_table()

    def delete_selected():
        selected = tree.selection()
        if not selected:
```

```
            messagebox.showwarning("Warning", "No experiments selected!")
            return
        if not messagebox.askyesno("Confirm Delete", f"Delete {len(selected)} selected experiment(s)?"):
            return
        ids_to_delete = [int(tree.item(item, "values")[0]) for item in selected]
        conn = sqlite3.connect("experiments.db")
        cursor = conn.cursor()
        for eid in ids_to_delete:
            cursor.execute("DELETE FROM experiments WHERE id = ?", (eid,))
        conn.commit()
        conn.close()
        refresh_table()
        messagebox.showinfo("Deleted", f"{len(ids_to_delete)} experiment(s) deleted.")

    def delete_all():
        if not messagebox.askyesno("Confirm Delete All", "⚠ This will delete ALL experiments! Are you sure?"):
            return
        conn = sqlite3.connect("experiments.db")
        cursor = conn.cursor()
        cursor.execute("DELETE FROM experiments")
        conn.commit()
        conn.close()
        refresh_table()
        messagebox.showinfo("Deleted", "All experiments have been deleted.")

    def compare_selected():
        selected = tree.selection()
        if len(selected) < 2:
            messagebox.showwarning("Warning", "Select at least 2 experiments to compare!")
            return
        exp_ids = [int(tree.item(item, "values")[0]) for item in selected]
        conn = sqlite3.connect("experiments.db")
        cursor = conn.cursor()
        experiments_data = []
        for eid in exp_ids:
            cursor.execute("SELECT id, dataset, model, accuracy, loss, time, history FROM experiments WHERE id=?", (eid,))
            row = cursor.fetchone()
            if row:
                hist = {}
                if row[6] and row[6] != "null":
```

```
                try:
                    hist = json.loads(row[6])
                except:
                    hist = {}
            exp = {
                "id": row[0],
                "dataset": row[1],
                "model": row[2],
                "accuracy": row[3],
                "loss": row[4],
                "time": row[5],
                "history": hist
            }
            experiments_data.append(exp)
        conn.close()
        if len(experiments_data) < 2:
            messagebox.showwarning("Warning", "Not enough valid experiments to compare.")
            return

        fig, axes = plt.subplots(1, 3, figsize=(15, 4))
        names = [f"Exp {e['id']} ({e['model']})" for e in experiments_data]
        accuracies = [e["accuracy"] for e in experiments_data]
        times = [e["time"] for e in experiments_data]
        axes[0].bar(names, accuracies, color='green')
        axes[0].set_title("Accuracy Comparison")
        axes[0].set_ylabel("Accuracy")
        axes[0].tick_params(axis='x', rotation=45)
        axes[1].bar(names, times, color='orange')
        axes[1].set_title("Time Comparison")
        axes[1].set_ylabel("Time (s)")
        axes[1].tick_params(axis='x', rotation=45)
        axes[2].set_title("Training Accuracy per Epoch")
        for e in experiments_data:
            hist = e.get("history", {})
            if "accuracy" in hist:
                axes[2].plot(hist["accuracy"], label=f"Exp {e['id']}")
            elif "val_accuracy" in hist:
                axes[2].plot(hist["val_accuracy"], label=f"Exp {e['id']}")
        axes[2].set_xlabel("Epoch")
        axes[2].set_ylabel("Accuracy")
        axes[2].legend()
        plt.tight_layout()
        plt.show()
```

```
    frame_buttons = tk.Frame(win, bg=BG_COLOR)
    frame_buttons.pack(fill="x", padx=10, pady=10)

    btn_compare = tk.Button(frame_buttons, text="Compare Selected",
command=compare_selected,
                            bg=BTN_COLOR, fg=TEXT_COLOR,
activebackground=BTN_HOVER,
                            relief="flat", font=("Arial", 10, "bold"))
    btn_refresh = tk.Button(frame_buttons, text="Refresh",
command=refresh_table,
                            bg=BTN_COLOR, fg=TEXT_COLOR,
activebackground=BTN_HOVER,
                            relief="flat", font=("Arial", 10, "bold"))
    btn_delete_selected = tk.Button(frame_buttons, text="Delete Selected",
command=delete_selected,
                                    bg=BTN_COLOR, fg=TEXT_COLOR,
activebackground=BTN_HOVER,
                                    relief="flat", font=("Arial", 10, "bold"))
    btn_delete_all = tk.Button(frame_buttons, text="Delete All",
command=delete_all,
                               bg=BTN_COLOR, fg=TEXT_COLOR,
activebackground=BTN_HOVER,
                               relief="flat", font=("Arial", 10, "bold"))

    for i, btn in enumerate([btn_compare, btn_refresh, btn_delete_selected,
btn_delete_all]):
        btn.grid(row=0, column=i, sticky="ew", padx=3, pady=3)
        frame_buttons.grid_columnconfigure(i, weight=1)

    def on_enter(e, b):
        b['bg'] = BTN_HOVER
    def on_leave(e, b):
        b['bg'] = BTN_COLOR
    for btn in [btn_compare, btn_refresh, btn_delete_selected, btn_delete_all]:
        btn.bind("<Enter>", lambda e, b=btn: on_enter(e, b))
        btn.bind("<Leave>", lambda e, b=btn: on_leave(e, b))

    status_label = tk.Label(win, text="", bg=BG_COLOR, fg=TEXT_COLOR)
    status_label.pack(side="bottom", fill="x", padx=10, pady=5)

# ==========================
# EXPERIMENT WINDOW
# ==========================
def open_experiment():
    win = tk.Toplevel(root)
```

```
win.title("Experiment")
win.geometry("600x480")
win.configure(bg=BG_COLOR)
win.resizable(False, False)

dataset = tk.StringVar(value="MNIST")
batch = tk.StringVar(value="32")
device = tk.StringVar(value="GPU")
precision = tk.StringVar(value="FP32")
data_path = tk.StringVar(value="")
model_type = tk.StringVar(value="Simple CNN")

result_data = {}
logs = []
config = {}

result_queue = multiprocessing.Queue()
log_queue = multiprocessing.Queue()
result_var = tk.StringVar()

def load_dataset():
    path = filedialog.askdirectory()
    if path:
        data_path.set(path)

def update_result_text():
    acc = result_data.get("accuracy", 0)
    loss = result_data.get("loss", 0)
    t = result_data.get("time", 0)
    result_text.delete(1.0, tk.END)
    result_text.insert(tk.END, f"Accuracy: {acc:.4f}\n")
    result_text.insert(tk.END, f"Loss: {loss:.4f}\n")
    result_text.insert(tk.END, f"Time: {t:.2f} seconds\n")

def start():
    if dataset.get() == "CUSTOM" and not data_path.get():
        messagebox.showerror("Error", "Please select dataset folder!")
        return

    result_var.set("")
    logs.clear()
    result_text.delete(1.0, tk.END)
    btn_start.config(state="disabled")
    btn_save.config(state="disabled")
    btn_graph.config(state="disabled")
```

```
config.clear()
config.update({
    "dataset": dataset.get(),
    "batch": int(batch.get()),
    "device": device.get(),
    "precision": precision.get(),
    "data_path": data_path.get(),
    "model_type": model_type.get()
})

log_win = tk.Toplevel(win)
log_win.title("Logs")
log_win.geometry("500x300")
log_text = tk.Text(log_win)
log_text.pack(fill="both", expand=True)

def add_log(msg):
    logs.append(msg)
    log_text.insert("end", msg + "\n")
    log_text.see("end")

def check_log():
    while not log_queue.empty():
        add_log(log_queue.get())
    log_win.after(100, check_log)
```

ДОДАТОК В

Слайди презентації

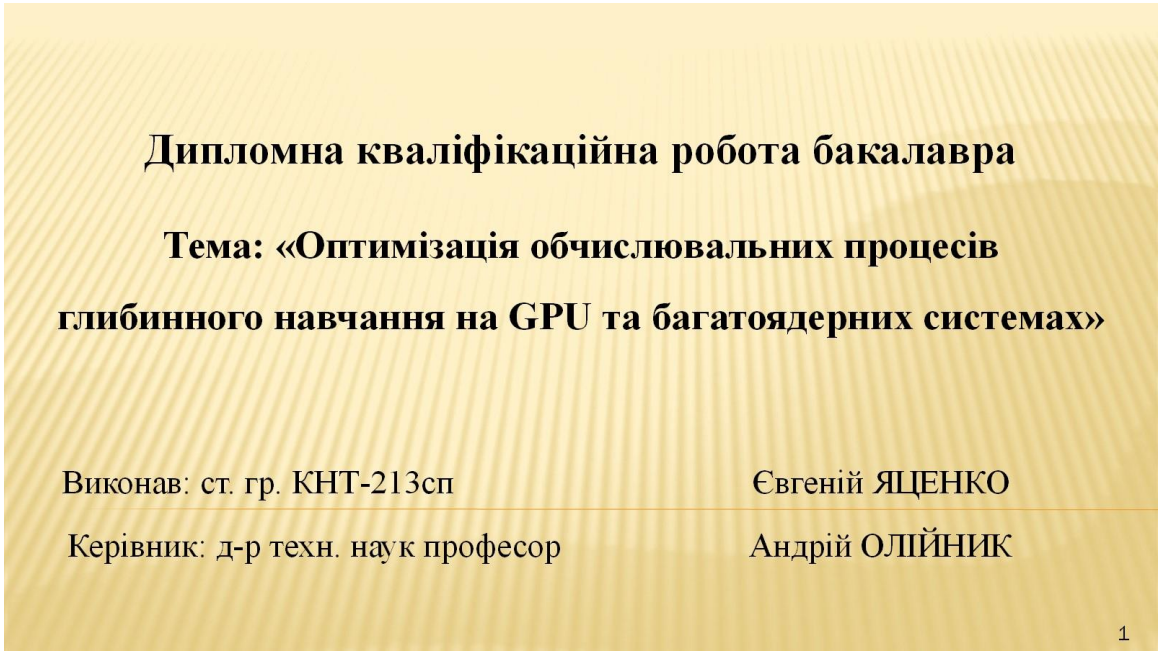


Рисунок В.1 – Слайд 1

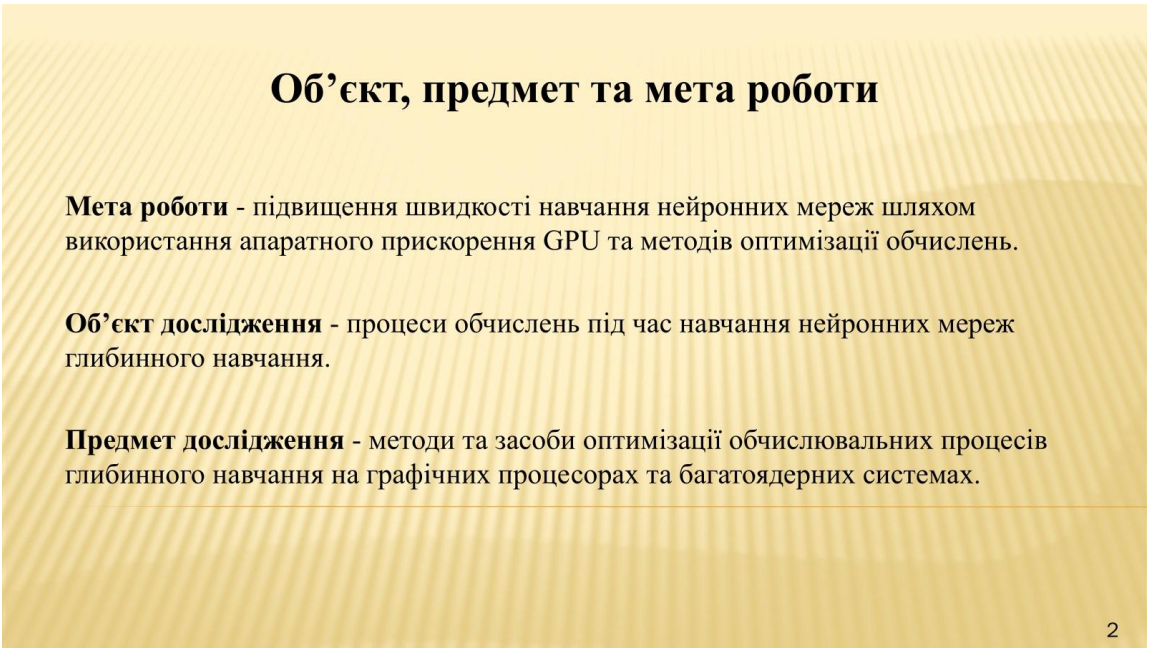


Рисунок В.2 – Слайд 2

Порівняльний аналіз існуючих програмних засобів для експериментів

Засіб	Переваги	Недоліки для задачі
TensorBoard (TensorFlow)	Потужна візуалізація	Складна інтеграція, немає порівняння CPU/GPU
MLflow	Керування експериментами	Важке налаштування, немає GUI для моніторингу
W&B	Зручний веб-інтерфейс	Хмарний сервіс (потребує інтернету), платний
Скрипти PyTorch	Простота, повний контроль	Немає GUI, історії, моніторингу ресурсів
Розроблений застосунок	GUI + моніторинг + порівняння + локально	Обмеження (3 епохи, проста модель, без багатопроцесорної системи)

Висновок: жоден з існуючих засобів не забезпечує одночасно легкість використання, графічний інтерфейс, багатопроцесний моніторинг ресурсів без консольних вікон та можливість прямого порівняння продуктивності CPU/GPU з вбудованими експериментами. Тому розробка власного застосунку є обґрунтованою.

3

Рисунок В.3 – Слайд 3

Обґрунтування вибору мови програмування та бібліотек

Компонент	Обраний інструмент	Ключова причина
Мова	Python	Екосистема ML, простота
Фреймворк	PyTorch	Динамічні графи, контроль GPU, підтримка змішаної точності
GUI	Tkinter	Вбудований у Python, легкий
Моніторинг ресурсів	psutil + GPUtil	Дані CPU/GPU без консольних вікон
Паралелізм	multiprocessing	Не блокує GUI, черги експериментів
Візуалізація	Matplotlib	Інтеграція з Tkinter

Висновок: обраний стек (Python + PyTorch + Tkinter + psutil/GPUtil + multiprocessing) забезпечує швидкість розробки, повний контроль над обчисленнями та можливість запуску без консольних вікон, що ідеально відповідає завданню дослідження продуктивності CPU/GPU для навчання нейронних мереж.

4

Рисунок В.4 – Слайд 4

Аналіз CPU та GPU

Порівняльна продуктивність на основі результатів, взятих з відкритих бенчмарків для мережі ResNet-50 на датасеті ImageNet (1,2 млн зображень розміром 224×224 пікселі)

Пристрій	Час на епоху (с)	Прискорення
Intel Core i9-10900 (10 ядер)	2450	1×
NVIDIA RTX 3060	120	20,4×
NVIDIA RTX 4090	35	70×

GPU забезпечує прискорення в 20–70 разів завдяки масовому паралелізму та високій пропускній здатності пам’яті.

5

Рисунок В.5 – Слайд 5

Архітектура програмного застосунку

Модулі:

— Графічний інтерфейс (Tkinter);

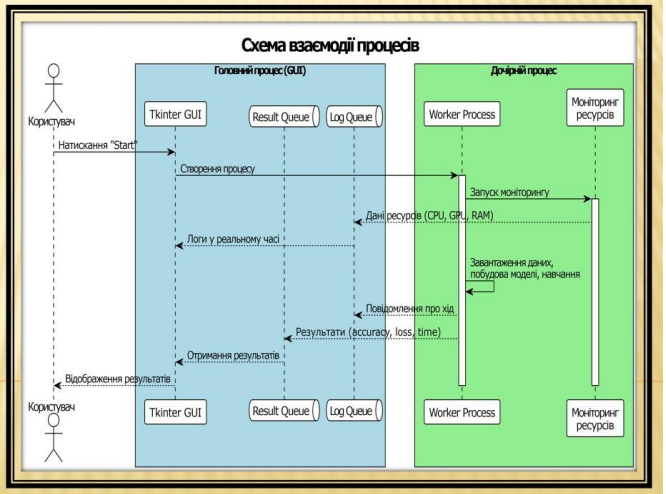
— Обчислювальний процес (worker_process);

— Черги для обміну даними (multiprocessing.Queue);

— Моніторинг ресурсів (psutil + nvidia-smi);

— Збереження результатів (RTF, TXT, DOCX);

— Дашборд (SQLite + графіки).



6

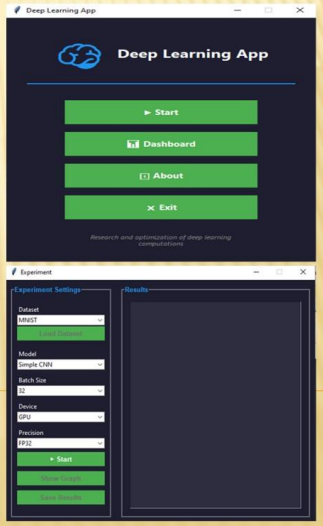
Рисунок В.6 – Слайд 6

Реалізація GUI

Головне вікно – кнопки Start, Dashboard, About, Exit.

Вікно експерименту – дві колонки:
— Ліва: налаштування (датасет, модель, розмір пакету, пристрій, точність);
— Права: поле результатів.

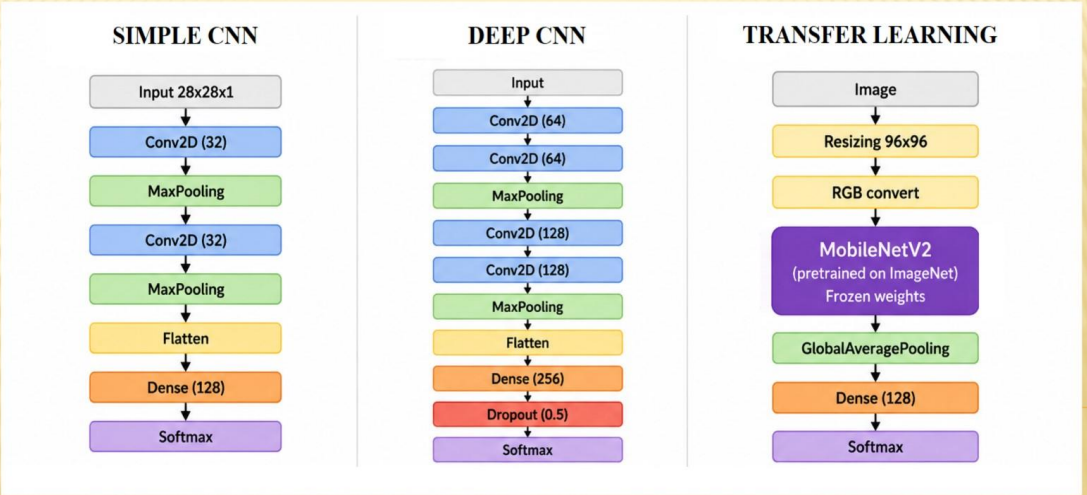
Кнопки:
— Start – запуск окремого процесу навчання;
— Load Dataset – кнопка для вибору папки з власним датасетом, що активується лише при виборі CUSTOM;
— Show Graph – графіки точності та втрат (з можливістю збереження PNG/JPEG);
— Save Results – експорт у RTF, TXT або DOCX.



7

Рисунок В.7 – Слайд 7

Модуль навчання нейронних мереж



8

Рисунок В.8 – Слайд 8

Експериментальні результати (розмір пакету)

Датасет: CIFAR-10, GPU, FP32, Simple CNN

Розмір пакету (Batch size)	Час (с)	Точність (%)
32	22,80	65,02
64	13,54	63,29
128	9,71	62,67

Висновок: збільшення розміру пакету у 2,35 рази скорочує час, точність падає на 2,35 %.

9

Рисунок В.9 – Слайд 9

Порівняння CPU та GPU (розмір пакету = 64, FP32)

Датасет	Пристрій	Час (с)	Точність (%)	Прискорення
MNIST	CPU	43,94	98,95	1×
MNIST	GPU	13,55	98,92	3,24×
CIFAR-10	CPU	46,28	63,27	1×
CIFAR-10	GPU	13,54	63,29	3,42×

Висновок: GPU прискорює навчання у 3,2–3,4 рази без втрати точності.

10

Рисунок В.10 – Слайд 10

Змішана точність FP16 (GPU, розмір пакету = 64)

Датасет	Точність	Час (с)	Точність (%)	Зміна часу
MNIST	FP32	13,49	99,03	0 %
MNIST	FP16	21,43	98,98	+58,9 %
CIFAR-10	FP32	13,26	62,06	0 %
CIFAR-10	FP16	19,91	64,06	+50,1 %

Висновок: для малої моделі (Simple CNN) змішана точність сповільнює навчання; рекомендується для глибоких моделей.

11

Рисунок В.11 – Слайд 11

Аналіз завантаження ресурсів

Середні значення (GPU, FP32, розмір пакету = 64)

Датасет	CPU (%)	RAM (%)	GPU (%)	VRAM (%)
MNIST	18,2	49,3	36,8	89,3
CIFAR-10	18,0	54,0	41,0	89,2
CUSTOM	48,7	47,7	20,5	89,1

Висновки:

— VRAM резервується майже повністю (політика TensorFlow);

— Для малих датасетів вузьке місце – завантаження даних (CPU);

— Для великих датасетів GPU використовується ефективно.

12

Рисунок В.12 – Слайд 12

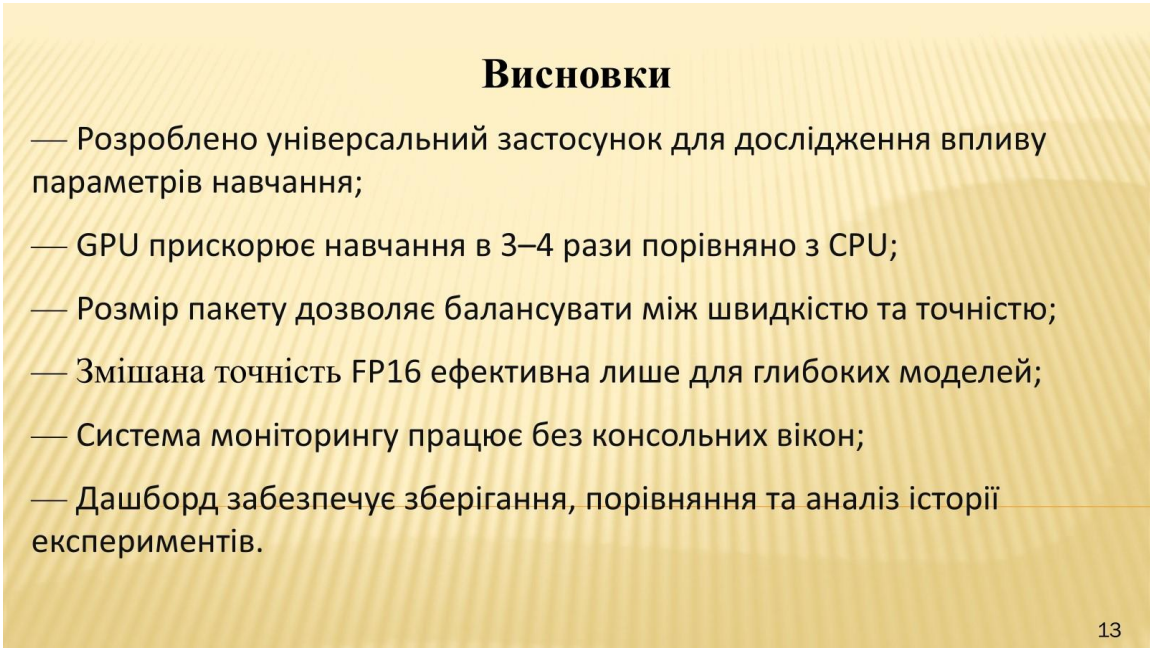


Рисунок В.13 – Слайд 13