НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
“___”_____2026 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “Вебплатформа аналізу продуктивності виконання завдань моделями машинного навчання на стороні клієнта”

Виконав: студент 4 курсу, групи ТР-21
Єрохов Валерій Сергійович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент Владислав ТІТОВ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент доцент каф. ТАЕ, к.т.н., Артур РАЧИНСЬКИЙ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль ст.викладач Ольга БЕСПАЛА _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ - 2026

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
“___” _____2026 р.

ЗАВДАННЯ
на дипломну роботу студенту

Єрохову Валерію Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Вебплатформа аналізу продуктивності виконання завдань моделями машинного навчання на стороні клієнта”
Науковий керівник Тітов Владислав Миколайович
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “28” травня 2026 року № 1992

2. Термін подання студентом роботи 04.06.2026

3. Вихідні дані до роботи мова програмування JavaScript, бібліотека React, фреймворки машинного навчання TensorFlow.js та ONNX Runtime Web, веб-стандарти WebAssembly, WebGL, WebGPU, набір даних MNIST, середовище розробки VS Code

4. Перелік питань, які потрібно розробити _____

1) провести аналіз проблематики виконання моделей машинного навчання у гетерогенному веб-середовищі та порівняти існуючі фреймворки

2) розробити методи та алгоритми для оптимізації і динамічного препроцесингу вхідних даних

3) спроектувати та реалізувати архітектуру клієнтського веб-застосунку для бенчмаркінгу

4) провести експериментальні дослідження продуктивності обчислювальних бекендів (CPU, GPU) та проаналізувати отримані результати

5. Орієнтований перелік ілюстративного матеріалу скріншоти графічного інтерфейсу користувача, гістограми порівняння продуктивності обчислювальних бекендів (WebGPU, WASM, WebGL).

6. Дата видачі завдання 15.09.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	15.09.2025	Виконано
2.	Аналіз методів та засобів розв’язання задачі	20.04.-23.04.2026	Виконано
3.	Розробка архітектури та загальної структури системи	24.04.-27.04.2026	Виконано
4.	Розробка окремих підсистем	28.04.-01.05.2026	Виконано
5.	Програмна реалізація системи	02.05.-07.05.2026	Виконано
6.	Оформлення пояснювальної записки	08.05.-29.05.2026	Виконано
7.	Захист програмного забезпечення	15.05.2026	Виконано
8.	Передзахист	05.06.2026	
9.	Захист	15.06.-19.06.2026	

Студент _____ Валерій ЄРОХОВ
(підпис) (прізвище та ініціали)

Керівник _____ Владислав ТІТОВ
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота виконана на 51 сторінці, містить 5 ілюстрацій, 4 додатки, 11 джерел у переліку посилань.

Мета роботи – розробка веб-системи для емпіричного аналізу та порівняння продуктивності виконання моделей машинного навчання безпосередньо у клієнтському веб-середовищі (Edge ML).

Методи та засоби: методи об'єктно-орієнтованого програмування, апаратне прискорення обчислень через графічні та центральні процесори, бібліотека побудови інтерфейсів React, фреймворки машинного навчання TensorFlow.js та ONNX Runtime Web, технології веб-стандартів WebAssembly (WASM), WebGL, WebGPU, інструментарій маніпуляції піксельними даними HTML Canvas API.

Результат – розроблено та протестовано програмний інструментарій (бенчмаркінг-систему) для оцінювання швидкодії інференсу нейронних мереж на стороні клієнта із вбудованим алгоритмом динамічного препроцесингу вхідних даних.

Ключові слова: МАШИННЕ НАВЧАННЯ, БЕНЧМАРКІНГ, EDGE ML, WEBGPU, WEBASSEMBLY, TENSORFLOW.JS, ONNX RUNTIME, REACT.

ANNOTATION

The thesis is executed on 51 pages, contains 5 illustrations, 4 appendices, 11 sources in the reference list.

The aim of the work is the development of a web system for empirical analysis and comparison of the performance of executing machine learning models directly in the client web environment (Edge ML).

Methods and tools: object-oriented programming methods, hardware acceleration of computations via graphic and central processing units, React user interface library, TensorFlow.js and ONNX Runtime Web machine learning frameworks, WebAssembly (WASM), WebGL, WebGPU web standard technologies, HTML Canvas API pixel data manipulation toolkit.

Result – a software toolkit (benchmarking system) for evaluating the inference speed of neural networks on the client side with a built-in dynamic input data preprocessing algorithm has been developed and tested.

Keywords: MACHINE LEARNING, BENCHMARKING, EDGE ML, WEBGPU, WEBASSEMBLY, TENSORFLOW.JS, ONNX RUNTIME, REACT.

ЗМІСТ

ВСТУП

Актуальність теми. Інтеграція систем штучного інтелекту (ШІ) та машинного навчання (МН) у сучасні веб-застосунки є одним із найбільш динамічних напрямків розвитку інформаційних технологій. Традиційно архітектура програмних продуктів, що використовують нейронні мережі, будувалася за централізованою (хмарною) моделлю. Клієнтський пристрій виконував лише функцію збору даних та їх передачі на віддалені сервери, де відбувалися безпосередні математичні обчислення (інференс), після чого результати поверталися користувачеві. Незважаючи на доступність високих обчислювальних потужностей, такий підхід супроводжується низкою критичних обмежень. По-перше, виникають неминучі затримки (latency) при передачі інформації через мережу Інтернет, що є неприпустимим для систем реального часу. По-друге, обробка персональних даних користувачів на сторонніх серверах створює серйозні ризики компрометації конфіденційної інформації. По-третє, масштабування серверної інфраструктури для обслуговування мільйонів запитів вимагає колосальних фінансових витрат від компаній-розробників.

Вирішенням цих проблем стала зміна парадигми на користь децентралізованих обчислень — технології Edge ML (машинне навчання на периферійних пристроях). Сучасні веб-браузери перестали бути простими інструментами для рендерингу гіпертексту і перетворилися на повноцінні платформи виконання складного програмного коду. Завдяки розвитку стандарту WebAssembly (WASM) стало можливим ефективне використання ресурсів центрального процесора (CPU) користувача зі швидкістю, що наближається до нативної. Водночас розвиток графічних інтерфейсів (API), таких як WebGL та новітній WebGPU, дозволив розробникам отримувати прямий доступ до обчислювальних потужностей дискретних та інтегрованих відеокарт (GPU) для масивної паралелізації матричних операцій.

Разом з тим, перенесення моделей машинного навчання у браузер породило нову проблему — екстремальну гетерогенність (різнорідність) клієнтських апаратних платформ. Розробники стикаються з широким спектром пристроїв: від малопотужних мобільних телефонів до високопродуктивних комп'ютерів з архітектурою ARM (наприклад, процесори Apple серії M). У зв'язку з цим, вибір оптимального фреймворку (зокрема лідерів індустрії — TensorFlow.js або ONNX Runtime Web) та обчислювального бекенду стає нетривіальною задачею. Технологія, яка демонструє найвищу швидкодію на одній системі, може працювати вкрай неефективно або навіть викликати критичні збої на іншій через високі накладні витрати на передачу даних або відсутність апаратної підтримки специфічних інструкцій.

Відсутність універсального рішення обумовлює гостру необхідність у створенні спеціалізованих програмних інструментів для бенчмаркінгу — систем, здатних емпіричним шляхом оцінювати продуктивність виконання моделей у реальних умовах на конкретному обладнанні. Розробка такої веб-системи дозволить інженерам проводити точні вимірювання швидкодії (інференсу), аналізувати поведінку різних API (WASM, WebGL, WebGPU) та приймати науково обґрунтовані рішення щодо вибору технологічного стеку для оптимізації сучасних Edge ML застосунків. Саме ці фактори визначають високу актуальність обраної теми дипломної роботи.

Метою роботи є розробка веб-системи моніторингу та аналізу продуктивності виконання моделей машинного навчання на стороні клієнта для забезпечення обґрунтованого вибору технологічного стеку у веб-розробці.

Для досягнення поставленої мети необхідно виконати наступні завдання:

1. Аналіз підходів, методів та алгоритмів розв’язання задачі оцінювання продуктивності моделей машинного навчання у гетерогенному веб-середовищі.

2. Аналіз програмних засобів для реалізації програмної системи бенчмаркінгу (можливостей фреймворків TensorFlow.js, ONNX Runtime Web та сучасних графічних/процесорних веб-стандартів).

3. Розробка програмного забезпечення для порівняльного аналізу виконання МН-моделей на стороні клієнта із вбудованими алгоритмами динамічного препроцесингу вхідних даних.

4. Тестування розробленого програмного забезпечення на сучасному апаратному забезпеченні з метою формування статистичної бази та підтвердження коректності роботи системи.

Структура й обсяг роботи. Дипломна робота складається зі вступу, чотирьох розділів основної частини, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 74 сторінок тексту, що містить 5 ілюстрацій. Список використаних джерел налічує 11 найменувань.

1 ЗАДАЧА АНАЛІЗУ ПРОДУКТИВНОСТІ EDGE ML

У даному розділі наведено розгорнуту постановку прикладної задачі створення веб-системи для аналізу продуктивності моделей машинного навчання. Здійснено аналіз проблематики інтеграції інтелектуальних систем у гетерогенне клієнтське середовище веб-браузерів. Розглянуто ключові методи виконання та оптимізації інференсу, а також проведено детальний огляд сучасних програмних засобів, фреймворків та стандартів апаратного прискорення (WASM, WebGL, WebGPU), що використовуються для реалізації Edge ML рішень.

1.1 Постановка прикладної задачі дослідження

Упровадження інтелектуальних систем безпосередньо у клієнтське середовище веб-браузерів (Edge ML) відкриває нові можливості для створення швидких, безпечних та автономних веб-застосунків. Проте, фундаментальною проблемою даного підходу є відсутність жорстко детермінованого середовища виконання. На відміну від хмарних серверів, де розробник точно знає специфікації апаратного забезпечення (кількість ядер CPU, обсяг VRAM, тип GPU), веб-застосунок має однаково стабільно працювати на надзвичайно гетерогенному (різнорідному) парку пристроїв. Користувацькі системи різняться за архітектурою процесорів (x86-64, ARM), графічними адаптерами (інтегровані або дискретні), обсягом оперативної пам'яті та рівнем підтримки новітніх веб-стандартів.

Прикладна задача, що розв’язується у даній дипломній роботі, полягає у розробці спеціалізованого програмного інструментарію (бенчмаркінгової системи) для емпіричного оцінювання та порівняння швидкодії інференсу (процесу виконання навчених моделей) у браузері. Розв’язання цієї задачі вимагає реалізації механізму, який дозволить завантажувати ідентичні вхідні дані, пропускати їх

через різні комбінації фреймворків машинного навчання та обчислювальних бекендів, а також з високою точністю фіксувати витрачений час і спожиті ресурси.

Ключовими підзадачами, які формують загальну прикладну задачу, є:

1. Забезпечення інваріантності вхідних даних: система повинна вміти приймати користувацькі зображення різних форматів та автоматично приводити їх до єдиного еталонного тензорного вигляду, необхідного для коректної роботи моделі (наприклад, алгоритмів комп'ютерного зору).

2. Реалізація архітектури взаємодії з декількома незалежними фреймворками (TensorFlow.js та ONNX Runtime) у межах одного сеансу роботи браузера без конфліктів за виділення пам'яті.

3. Розробка модуля статистичної агрегації: оскільки виконання обчислень у браузері піддається впливу фонових процесів операційної системи та механізмів збирання сміття (Garbage Collection) у JavaScript, одиничне вимірювання часу є нерепрезентативним. Необхідна реалізація багаторазових прогонів з усуненням похибок "холодного старту".

4. Оцінка навантаження на систему: окрім часових показників (латенсі), прикладна задача вимагає моніторингу споживання оперативної пам'яті (RAM) під час виконання математичних матричних операцій.

Вирішення описаної прикладної задачі дозволить розробникам веб-систем уникати методу "спроб і помилок" при інтеграції ШІ, спираючись на точні емпіричні дані щодо поведінки конкретного бекенду на цільовій апаратній платформі.

1.2 Огляд методів виконання інференсу в гетерогенному веб-середовищі

Для успішного вирішення задачі бенчмаркінгу необхідно розуміти методи, які лежать в основі життєвого циклу моделі машинного навчання в інтернеті.

Процес виконання ШІ-задач у браузері радикально відрізняється від класичного серверного підходу і базується на методах трансляції та адаптивної оптимізації.

Першим важливим методом є метод конвертації та серіалізації моделей. Класічні нейронні мережі зазвичай проектуються та тренуються за допомогою мови програмування Python у середовищах на кшталт Keras, PyTorch або TensorFlow. Збережені у такому вигляді моделі містять надмірну метаінформацію, необхідну для подальшого донавчання, та є занадто великими для швидкого завантаження через інтернет. Для їх виконання у браузері застосовуються методи графової оптимізації (Graph Optimization) та квантування (Quantization). Модель конвертується у спеціалізовані формати (наприклад, graph_model для TF.js або .onnx для ONNX Runtime). Цей метод передбачає видалення вузлів, які використовуються лише для тренування (наприклад, оптимізаторів), злиття суміжних операцій (operator fusion) та розбиття масивів ваг (weights) на дрібні фрагменти (shards) для паралельного завантаження браузером.

Другим критично важливим методом є метод нормалізації та препроцесингу неструктурованих даних. Моделі комп'ютерного зору (наприклад, для класифікації рукописних символів MNIST) є вкрай чутливими до розподілу пікселів на вхідному зображенні. Будь-які шуми, зміна масштабу або інверсія кольорів призводять до ефекту Out-of-Distribution, коли модель видає хибні результати з високою впевненістю. У даній роботі застосовується розроблений метод динамічної евристичної нормалізації фону. Суть методу полягає у витягненні масиву RGBA-пікселів завантаженого користувачем зображення за допомогою HTML Canvas API, перетворенні його у градації сірого за допомогою вирахування показника яскравості (Luminance) та математичному обчисленні середньої яскравості кутових зон зображення. На основі цього показника застосовується метод автоматичної інверсії тензора, що гарантує ідеальну відповідність тестових даних очікуванням еталонної нейромережі, зводячи похибку розпізнавання до мінімуму.

Третім методом є метод ізольованого вимірювання продуктивності (Microbenchmarking). Виконання JavaScript-коду у сучасних рушіях (наприклад,

V8 у Chrome) супроводжується процесами JIT-компіляції (Just-In-Time). Перший виклик будь-якої складної функції завжди займає значно більше часу, оскільки рушій витрачає ресурси на аналіз типів даних та трансляцію байт-коду в машинні інструкції. Для роботи з відеокартами (GPU) цей ефект ще більш виражений, адже вимагає компіляції графічних або обчислювальних шейдерів драйверами операційної системи. Метод мікробенчмаркінгу, що застосовується у даній роботі, передбачає поділ процесу тестування на фазу "прогріву" (виконання фіктивної кількості ітерацій без збереження результатів) та фазу "вимірювання" (цикл послідовних інференсів із записом часових міток). Це дозволяє отримати статистично значущі показники середнього та медіанного часу.

1.3 Огляд програмних засобів та технологій апаратного прискорення

Для реалізації методів, описаних у попередньому підрозділі, та створення ефективного бенчмарку необхідне залучення широкого спектру сучасних веб-технологій та прикладних програмних інтерфейсів. Їх можна умовно поділити на технології низькорівневого апаратного прискорення та високорівневі фреймворки машинного навчання.

Технології процесорного прискорення (CPU-bound):

Головним інструментом для виконання складних математичних розрахунків на центральному процесорі у браузері є WebAssembly (WASM). На відміну від динамічно типізованого JavaScript, WASM є низькорівневим бінарним форматом, що виконується у безпечній "пісочниці" браузера (sandbox). WASM дозволяє ефективно використовувати багатопоточність (через Web Workers) та інструкції SIMD (Single Instruction, Multiple Data), що дає змогу процесору виконувати одну математичну операцію одразу над цілим вектором чисел. Це критично важливо для швидкодії матричних множень, які є основою роботи нейромереж.

Технології графічного прискорення (GPU-bound):

Для використання потужності відеокарт у браузерах традиційно використовується WebGL (Web Graphics Library). Оскільки WebGL спочатку проектувався для рендерингу 3D-сцен, використання його для машинного навчання (GPGPU) вимагає певних архітектурних компромісів: матриці даних нейромережі перетворюються на графічні текстури, а обчислення виконуються як обробка кольору пікселів за допомогою фрагментних шейдерів. Цей процес супроводжується значними накладними витратами (overhead) на пересилання даних між оперативною пам'яттю комп'ютера та пам'яттю відеокарти.

Наступним еволюційним кроком є новітній стандарт WebGPU. Це сучасний API, розроблений з урахуванням архітектури таких графічних інтерфейсів, як Vulkan, Metal та Direct3D 12. Його головною відмінністю від WebGL є наявність вбудованої підтримки обчислювальних конвеєрів (Compute Pipelines) та шейдерів (мовою WGSL). WebGPU значно ефективніше управляє пам'яттю та дозволяє браузеру безпосередньо звертатися до потужностей GPU для математичних операцій, мінімізуючи затримки.

Високорівневі програмні фреймворки:

1. TensorFlow.js – передова екосистема від Google. Це найпопулярніша бібліотека для розгортання моделей машинного навчання у JavaScript-середовищі. Вона має глибоку інтеграцію з технологіями WebGL та WebGPU, орієнтуючись переважно на досягнення максимальної продуктивності за рахунок використання відеокарти. Проте, автоматичний вибір бекендів у TensorFlow.js не завжди є оптимальним для моделей малого розміру, де накладні витрати на ініціалізацію GPU перевищують виграш у швидкості обчислень.

2. ONNX Runtime Web – програмне рішення від Microsoft, створене для виконання моделей у відкритому універсальному форматі ONNX. Його фундаментальною перевагою є феноменальна оптимізація процесорних обчислень за допомогою WebAssembly. Завдяки використанню оптимізованих математичних ядер (XNNPACK), ONNX Runtime часто демонструє вищу швидкодію та

стабільність на мобільних пристроях і слабких комп'ютерах порівняно з GPU-орієнтованими рішеннями.

1.4 Методи оптимізації та стиснення моделей машинного навчання для веб-середовища

Перенесення процесів інференсу на сторону клієнта (Edge ML) висуває надзвичайно суворі вимоги до фізичного розміру моделей машинного навчання та їхньої обчислювальної складності. Класичні нейронні мережі, які тренуються у серверних середовищах на базі потужних графічних прискорювачів (NVIDIA CUDA), зазвичай оперують ваговими коефіцієнтами у форматі 32-бітної рухомої коми (Float32). Розмір таких моделей може сягати сотень мегабайт або навіть гігабайт, що є абсолютно неприйнятним для веб-застосунків, де час завантаження сторінки (Time to Interactive) та ліміти оперативної пам'яті браузера є критичними факторами.

Для вирішення цієї проблеми перед етапом бенчмаркінгу моделі обов'язково проходять етап комплексної математичної та структурної оптимізації. Першим базовим підходом є оптимізація обчислювального графа (Graph Optimization). Під час тренування модель містить велику кількість допоміжних вузлів (наприклад, вузли розрахунку градієнтів, dropout-шари, оптимізатори на кшталт Adam або SGD), які є абсолютно непотрібними під час фази інференсу. Спеціалізовані конвертери (як tensorflowjs_converter або tf2onnx) виконують процес "заморожування" (Freezing) графа. У результаті видаляються всі тренувальні метадані, а змінні (Variables) перетворюються на константи (Constants). Крім того, конвертери виконують злиття операцій (Operator Fusing) — наприклад, об'єднання операції згортки (Convolution) та активаційної функції (ReLU) в єдиний монолітний обчислювальний блок, що зменшує кількість звернень до пам'яті під час виконання.

Другим, найбільш радикальним методом зменшення розміру моделі є квантування (Quantization). Суть цього методу полягає у зниженні точності представлення вагових коефіцієнтів. Замість використання 32-бітних чисел (Float32), ваги можуть бути математично наближені (сконвертовані) до 16-бітних чисел (Float16) або навіть до 8-бітних цілих чисел (Int8). Процес квантування дозволяє зменшити фізичний розмір моделі на диску та в оперативній пам'яті у 2-4 рази, зберігаючи при цьому загальну топологію нейронної мережі. Хоча квантування може призвести до незначної втрати точності розпізнавання (accuracy drop на рівні 1-2%), для переважної більшості Edge ML задач цей компроміс є виправданим.

У розробленій бенчмаркінговій системі еталонна модель для розпізнавання датасету MNIST була спеціально підготовлена з урахуванням веб-специфіки. Модель була сегментована (sharded) на фрагменти, що дозволяє браузеру виконувати паралельне завантаження вагових коефіцієнтів через HTTP-протокол. Ця архітектурна оптимізація гарантує, що час початкового завантаження моделі (Model Load Time) не стає "вузьким місцем", дозволяючи системі сфокусуватися виключно на вимірюванні швидкості самого математичного інференсу на різних апаратних бекендах.

1.5 Безпекові аспекти та забезпечення конфіденційності даних у парадигмі Edge ML

Одним із найважливіших рушійних факторів, що стимулюють перехід від класичних хмарних ШІ-систем до парадигми Edge ML у веб-браузерах, є стрімке посилення вимог до інформаційної безпеки та конфіденційності персональних даних користувачів. Зі вступом у дію таких нормативно-правових актів, як Загальний регламент захисту даних (GDPR) у Європейському Союзі та Закон про

конфіденційність споживачів (CCPA) у Каліфорнії, відповідальність розробників за збір, передачу та зберігання чутливої інформації критично зросла.

У класичній архітектурі, коли користувач завантажує зображення (наприклад, фотографію документа або обличчя) для розпізнавання, ці дані серіалізуються та передаються через відкриту мережу Інтернет на бекенд-сервер. Навіть за умови використання захищених протоколів (HTTPS/TLS), залишається ризик компрометації даних на проміжних вузлах (Man-in-the-Middle attacks) або безпосередньо у хмарному сховищі компанії внаслідок цілеспрямованих кібератак або витоків баз даних.

Виконання моделей машинного навчання на стороні клієнта за допомогою веб-браузера радикально змінює ландшафт кібербезпеки застосунку. Технологія Edge ML запроваджує концепцію "нульової передачі даних" (Zero Data Transfer). У розробленій системі бенчмаркінгу цей принцип реалізовано на архітектурному рівні: всі процеси препроцесингу зображень за допомогою HTML Canvas API та їх подальша тензорна обробка фреймворками TensorFlow.js або ONNX Runtime Web відбуваються виключно в межах локальної оперативної пам'яті користувацького пристрою.

Оскільки вхідні зображення ніколи не залишають межі браузера і не пересилаються через мережу, застосунок апріорі стає стійким до мережевих перехоплень. Це відкриває широкі перспективи для розробки медичних, фінансових та корпоративних систем, де обробка даних має відбуватися з дотриманням суворої комерційної або медичної таємниці (наприклад, стандарт HIPAA).

Крім того, перенесення інференсу на пристрій клієнта знімає з компанії-розробника тягар криптографічного шифрування великих масивів даних "у стані спокою" (Data at Rest) на серверах. Вся інформація існує лише у вигляді тимчасових змінних у купі (Heap) JavaScript-рушія і гарантовано знищується механізмами збирання сміття (Garbage Collection) після закриття вкладки браузера.

1.6 Сфери практичного застосування технологій клієнтського машинного навчання

Перенесення нейронних мереж на сторону клієнта (Edge ML) не лише вирішує проблеми конфіденційності, але й відкриває нові класи веб-застосунків, створення яких раніше було неможливим через мережеві обмеження. Оцінка продуктивності технологій, що проводиться у даному дослідженні, має безпосередній вплив на кілька ключових індустрій.

По-перше, це системи обробки потокового відео та доповненої реальності (AR) у реальному часі. Сучасні платформи відеоконференцій (наприклад, веб-версії Google Meet чи Zoom) використовують моделі семантичної сегментації для динамічного розмиття фону або заміни бекграунду. Для забезпечення плавного відеопотоку з частотою 30 кадрів на секунду (FPS), час інференсу моделі на кожен кадр не повинен перевищувати 33 мілісекунди. Застосування серверних обчислень для таких задач є неможливим через неминучі мережеві затримки (ping) та втрату пакетів. Тому точний бенчмаркінг WebGL та WebGPU дозволяє розробникам обирати технології, здатні вкластися у цей жорсткий часовий ліміт на пристрої клієнта.

По-друге, це медичні та діагностичні системи. Розробка веб-інструментів для попереднього аналізу рентгенівських знімків, дерматологічних зображень або ЕКГ безпосередньо у браузері лікаря чи пацієнта вимагає абсолютної гарантії того, що дані не будуть перехоплені в мережі. Використання оптимізованих WASM-модулів забезпечує локальну класифікацію з високим рівнем безпеки.

По-третє, це системи автономної роботи (Offline-first applications). Веб-застосунки, що функціонують в умовах нестабільного з'єднання (наприклад, системи розпізнавання шкідників на листках рослин для агрономів у полі), можуть завантажити модель у кеш браузера (Service Workers) при першому підключенні і надалі виконувати інференс повністю автономно. Результати

бенчмаркінгу, отримані у даній роботі, дозволяють оптимізувати такі системи під процесори мобільних пристроїв, мінімізуючи споживання заряду акумулятора.

2 МЕТОДИ ТА АЛГОРИТМИ РОЗВ'ЯЗАННЯ ЗАДАЧІ

У даному розділі представлено теоретичне та алгоритмічне обґрунтування підходів, які були використані для розв'язання задачі розробки веб-системи аналізу продуктивності моделей машинного навчання. Детально розглядаються алгоритми динамічної попередньої обробки вхідних даних, методика проведення мікробенчмаркінгу в недетермінованому середовищі браузера, а також математичний апарат статистичної обробки отриманих результатів вимірювання.

2.1 Алгоритм динамічного препроцесингу та інверсії вхідних даних

Однією з головних проблем при розгортанні моделей комп'ютерного зору є жорстка залежність нейронної мережі від формату вхідних даних. Еталонна згорткова нейронна мережа, використана у дослідженні, була натренована на базі даних MNIST. Цей датасет містить зображення розміром 28x28 пікселів, де цільовий об'єкт (рукописна цифра) має білий колір (інтенсивність пікселя наближається до 1.0), а фон є абсолютно чорним (інтенсивність 0.0).

При використанні системи реальними користувачами виникає проблема "зсуву даних" (Data Shift): користувачі завантажують зображення довільного розміру, у різних колірних форматах (RGB, RGBA) та переважно з темним об'єктом на світлому (білому) фоні. Пряма подача такого тензора до моделі призводить до повної втрати точності розпізнавання.

Для вирішення цієї проблеми було розроблено алгоритм динамічного адаптивного препроцесингу. Даний алгоритм виконує просторову та колірну трансформацію зображення у кілька послідовних кроків.

Першим кроком є просторова нормалізація — білінійна інтерполяція (Bilinear Interpolation) для зміни розміру завантаженого зображення до фіксованого розміру 28x28 пікселів з усуненням високочастотних графічних шумів.

Другим кроком є колірна нормалізація. Оскільки вхідне зображення може містити чотири канали (Red, Green, Blue, Alpha), алгоритм виконує згортку колірного простору у єдиний канал градацій сірого (Grayscale). Для розрахунку відносної яскравості (Luminance) кожного пікселя застосовується стандартна колориметрична формула:

$$L = 0.299 * R + 0.587 * G + 0.114 * B$$

де L — вихідна яскравість пікселя у нормалізованому діапазоні [0, 1],

R, G, B — нормалізовані інтенсивності червоного, зеленого та синього каналів відповідно

Третім і ключовим кроком є алгоритм евристичного аналізу фону та інверсії. Для того щоб система автоматично розпізнавала, чи намальована цифра чорним по білому, чи білим по чорному, алгоритм сканує показники яскравості у чотирьох крайніх точках матриці зображення (кутах). Оскільки цільовий об'єкт (цифра) зазвичай розташований у центрі зображення, кутові пікселі з вірогідністю понад 95% належать фону.

Математично процес визначення середньої яскравості фону (BgAvg) виражається так:

$$BgAvg = (L(0,0) + L(0, W-1) + L(H-1, 0) + L(H-1, W-1)) / 4$$

де L(x, y) — функція отримання яскравості пікселя за координатами,

W та H — ширина та висота зображення (у нашому випадку W = H = 28)

На основі розрахованого показника BgAvg приймається рішення про інверсію. Якщо BgAvg > 0.5 (тобто фон класифікується як світлий), до всього масиву пікселів застосовується лінійна трансформація інверсії:

$$L\_new(x, y) = 1.0 - L\_old(x, y)$$

де L_new(x, y) – нове (інвертоване) значення яскравості пікселя,

L_old(x, y) – початкове значення яскравості пікселя

Цей алгоритм гарантує, що на вхід обчислювального графа моделі машинного навчання завжди надходить тензор, параметри якого відповідають розподілу навчальної вибірки, що унеможливлює вплив формату картинки на результати бенчмаркінгу.

2.2 Методика проведення мікробенчмаркінгу в гетерогенному середовищі

Веб-браузер є багатопотоковим та вкрай недетермінованим середовищем виконання. На швидкість роботи JavaScript-коду постійно впливають побічні фактори: робота планувальника завдань операційної системи, JIT-компіляція (Just-In-Time) у рушії V8 або SpiderMonkey, а також механізми автоматичного збирання сміття (Garbage Collection). Особливо складним є процес оцінювання продуктивності графічних бекендів (WebGL та WebGPU), де ініціалізація вимагає міжпроцесної взаємодії (Inter-Process Communication) між браузером та драйверами відеокарти.

Для отримання об'єктивних та статистично значущих даних про продуктивність інференсу була розроблена спеціальна методика проведення мікробенчмаркінгу, що складається з двох ізольованих фаз.

Фаза 1: Ініціалізація та прогрів (Warm-up). У цій фазі система завантажує топологію моделі та її вагові коефіцієнти в оперативну пам'ять (або відеопам'ять VRAM у випадку WebGPU). Далі алгоритм виконує фіксовану кількість ($N_{warmup} = 2$) повних циклів інференсу. Основна мета цієї фази — дозволити браузеру та графічним драйверам скомпілювати та оптимізувати обчислювальні шейдери (WGSL або GLSL) та закешувати конвеєри (pipelines). Часові метрики, отримані під час фази прогріву, свідомо відкидаються і не включаються до підсумкової статистики, оскільки "холодний старт" (Cold Start) може займати в

десятки разів більше часу, ніж номінальний режим роботи, і суттєво викривить загальну картину продуктивності.

Фаза 2: Робоче вимірювання (Benchmarking). Після завершення прогріву система переходить до фази вимірювання, виконуючи задану кількість робочих ітерацій (N_runs = 10). Для вимірювання часу (латенсі) кожної ітерації використовується інтерфейс performance.now(). На відміну від стандартних методів роботи з часом (наприклад, об'єкта Date), цей API забезпечує мікросекундну точність та не піддається впливу коригувань системного годинника операційної системи (monotonic clock).

2.3 Математичні методи агрегації та статистичної обробки результатів

Після завершення циклу вимірювань система формує масив значень латенсі для кожної з N ітерацій. Оскільки у веб-середовищі можливі раптові мікрозатримки (spikes) через активацію процесу Garbage Collection, використання лише одного статистичного показника є недостатнім для об'єктивної оцінки стабільності бекенду.

Для комплексного аналізу швидкодії використовуються такі математичні методи агрегації даних:

1. Розрахунок середнього арифметичного часу (Mean Time). Показує загальну тенденцію продуктивності бекенду. Визначається як сума всіх виміряних значень часу, поділена на кількість ітерацій. Даний показник є основним для побудови порівняльних графіків між різними фреймворками.

2. Визначення медіанного часу (Median Time). Для пошуку медіани масив часових показників сортується за зростанням. Якщо кількість ітерацій непарна, медіаною є центральне значення. Якщо парна — середнє арифметичне двох центральних значень. Медіана є критично важливою метрикою для бенчмаркінгу,

оскільки, на відміну від середнього арифметичного, вона є робастною (стійкою) до аномальних викидів (outliers), спричинених системними затримками браузера.

3. Розрахунок стандартного відхилення (Standard Deviation). Даний показник використовується для оцінки стабільності та передбачуваності обчислювального бекенду (визначення рівня "джитеру"). Чим нижчим є значення стандартного відхилення, тим стабільніше працює обрана технологія. Розрахунок відбувається шляхом визначення квадратного кореня з дисперсії — середнього квадрата відхилень кожного окремого виміру від середнього арифметичного.

Додатково в межах алгоритму передбачено фіксацію часу першого вимірюваного запуску (First Run Time). Хоча він і не враховує початкову компіляцію (яка відбулася на етапі прогріву), він слугує індикатором ефективності кешування тензорів між ітераціями.

2.4 Теоретичні основи та математична модель згорткових нейронних мереж

Оскільки розроблена система бенчмаркінгу оперує моделями комп'ютерного зору, необхідним є глибоке розуміння математичного апарату, що лежить в основі еталонної нейронної мережі. Як базову архітектуру для тестування продуктивності браузерних технологій було обрано згорткову нейронну мережу (Convolutional Neural Network, CNN). Вибір цієї архітектури зумовлений її здатністю ефективно виявляти просторові ієрархії та локальні патерни у двовимірних масивах даних (зображеннях).

На відміну від повнозв'язних багатошарових персептронів (MLP), де кожен нейрон з'єднаний з усіма нейронами попереднього шару, CNN використовує операцію математичної згортки (convolution). Це дозволяє радикально зменшити кількість параметрів моделі та підвищити її стійкість до зсувів (translation invariance) на зображенні.

Основним обчислювальним елементом такої мережі є згортковий шар (Convolutional Layer). Математично операція дискретної двовимірної згортки для вхідного тензора (зображення) та ядра згортки (фільтра) описується наступним чином:

$$S(i,\ j)\ =\ (I\ *\ K)(i,\ j)\ =\ \Sigma(m)\ \Sigma(n)\ I(i\ +\ m,\ j\ +\ n)\ K(m,\ n)$$

де S(i, j) – значення пікселя на вихідній карті ознак (feature map);

I – вхідна матриця зображення;

K – матриця ядра згортки (фільтра) розміром m x n.

Після кожної лінійної операції згортки застосовується нелінійна функція активації. В еталонній моделі використовується функція ReLU (Rectified Linear Unit), яка ефективно вирішує проблему затухаючого градієнта під час тренування та є обчислювально легкою для інференсу у веб-браузері:

$$f(x)\ =\ max(0,\ x)$$

Для зменшення просторової розмірності карт ознак та забезпечення інваріантності до дрібних деформацій застосовується шар субдискретизації (Pooling Layer). У тестуємій архітектурі використано метод Max Pooling, який обирає максимальне значення у заданому вікні (зазвичай 2x2 пікселі). Ця операція є однією з найбільш вимогливих до пропускної здатності пам'яті (Memory Bandwidth), що робить її чудовим індикатором продуктивності при бенчмаркінгу між CPU та GPU бекендами.

Фінальним етапом класифікації є розгортання двовимірних карт ознак у вектор (шар Flatten) та передача його до повнозв'язного шару (Dense Layer). Для отримання ймовірнісного розподілу класів (якій саме цифрі від 0 до 9 відповідає зображення) на вихідному шарі застосовується математична функція Softmax, що нормалізує вихідні значення у діапазоні [0, 1], де їхня сума дорівнює одиниці.

Окрім прямого поширення сигналу (Forward Pass), який використовується під час інференсу, еталонна модель потребує специфічного математичного апарату для фази тренування. Для запобігання ефекту перенавчання (Overfitting) в архітектуру було інтегровано метод регуляризації Dropout. Його математична суть полягає у тому, що під час кожної ітерації навчання випадково обрана частка

нейронів (у розробленому алгоритмі p = 0.4) "вимикається", тобто їхні вихідні значення примусово прирівнюються до нуля. Це змушує нейронну мережу формувати більш стійкі внутрішні репрезентації ознак, не покладаючись на окремі пікселі вхідного зображення. Під час інференсу у веб-браузері шар Dropout деактивується, а всі вагові коефіцієнти масштабуються на коефіцієнт (1 - p), що гарантує детермінованість результатів тестування.

Як функцію втрат (Loss Function) для задачі багатокласової класифікації MNIST застосовано категоріальну перехресну ентропію (Categorical Crossentropy). Вона кількісно оцінює розбіжність між передбаченим ймовірнісним розподілом та істинним (One-Hot Encoded) вектором класів:

$$L = -\Sigma(c = 1..M)\, y\_\{o, c\} * log(p\_\{o, c\})$$

де M – кількість класів (10);

y_{o,c} – бінарний індикатор (0 або 1), який вказує, чи є клас c правильною відповіддю для спостереження o;

p_{o,c} – передбачена ймовірність того, що спостереження належить класу c.

Для мінімізації функції втрат застосовано алгоритм адаптивної оцінки моментів – Adam (Adaptive Moment Estimation). На відміну від класичного стохастичного градієнтного спуску (SGD), Adam обчислює індивідуальні адаптивні швидкості навчання для різних параметрів на основі оцінок першого (середнє значення градієнтів) та другого (нецентрована дисперсія градієнтів) моментів. Це дозволило досягти точності розпізнавання еталонної моделі понад 98% всього за 6 епох навчання, що гарантує високу валідність математичної складової бенчмарку.

2.5 Аналіз специфікацій еталонного набору даних MNIST

Для забезпечення об'єктивності та відтворюваності бенчмаркінгових експериментів було прийнято рішення використовувати загальновизнаний

еталонний набір даних. У сфері машинного навчання стандартом "де-факто" для тестування базових архітектур комп'ютерного зору є датасет MNIST (Modified National Institute of Standards and Technology).

Оригінальна база даних MNIST була створена Яном Лекуном (Yann LeCun) та містить 70 000 чорно-білих зображень рукописних цифр (60 000 для тренування та 10 000 для тестування). Кожне зображення являє собою одноканальну матрицю розміром 28x28 пікселів. Значення яскравості кожного пікселя в оригінальному форматі лежать у діапазоні від 0 (абсолютно білий фон) до 255 (абсолютно чорний піксель об'єкта). Проте у більшості сучасних застосувань зображення інвертуються: фон стає чорним (0.0), а пікселі, що формують цифру, наближаються до значення 1.0.

Використання MNIST у розробленій бенчмаркінговій веб-системі обґрунтоване низкою факторів:

1. Легкотривалість: розмірність вхідного тензора становить лише 784 елементи (28 * 28 * 1). Це дозволяє оцінювати швидкість ініціалізації та передачі даних до відеокарти (GPU overhead), оскільки самі обчислення займають мілісекунди. Якби використовувалися зображення високої роздільної здатності (наприклад, 1024x1024), час передачі даних був би прихований за часом обчислень, що унеможливило б об'єктивну оцінку ефективності WebAssembly-бекендів.

2. Стандартизація: передбачуваність результатів класифікації дозволяє не лише тестувати час виконання, але й гарантувати коректність роботи алгоритму динамічного препроцесингу зображень у веб-браузері.

3. Низьке споживання пам'яті: еталонна модель для MNIST у форматі TensorFlow.js займає менше 2 мегабайт, що дозволяє миттєво завантажувати її у пам'ять браузера навіть при низькошвидкісному інтернет-з'єднанні, не створюючи додаткових похибок під час тестування "холодного старту".

2.6 Алгоритм автоматичного вибору обчислювального бекенду

Одним із завдань дослідження було створення механізму, здатного імітувати логіку вибору оптимального середовища виконання "за замовчуванням". Оскільки продуктивність технологій залежить від апаратних можливостей пристрою, алгоритм базується на принципі детектування наявних API (Feature Detection).

Логіка алгоритму базується на перевірці глобального об'єкта navigator у браузері. Першочергово виконується асинхронний запит до інтерфейсу navigator.gpu. Якщо цей об'єкт існує та операційна система дозволяє запит адаптера (requestAdapter), алгоритм класифікує пристрій як такий, що має підтримку новітнього стандарту WebGPU. У цьому випадку для фреймворку TensorFlow.js автоматично встановлюється пріоритет бекенду tfjs-webgpu.

У випадку, якщо браузер не підтримує WebGPU (наприклад, у застарілих версіях оглядачів), алгоритм не намагається використати застарілий та менш стабільний WebGL, а вдається до стратегії відкочування (Fallback Strategy), призначаючи пріоритетним бекенд WebAssembly (WASM). Цей підхід зумовлений результатами попередніх досліджень, які доводять, що для невеликих моделей машинного навчання CPU-обчислення через WASM забезпечують більш стабільний та передбачуваний час виконання порівняно зі спробами трансляції тензорних операцій через графічні конвеєри WebGL.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-СИСТЕМИ АНАЛІЗУ ПРОДУКТИВНОСТІ

У даному розділі наведено детальний опис практичної реалізації спроектованої системи. Розглянуто загальну архітектуру односторінкового веб-застосунку (SPA), структуру та взаємодію його візуальних компонентів. Особливу увагу приділено програмній реалізації обчислювального ядра, логіці маршрутизації тензорів між різними фреймворками машинного навчання, а також інкапсуляції алгоритмів препроцесингу вхідних даних та збору системних метрик у середовищі виконання JavaScript.

3.1 Архітектура клієнтського застосунку на базі React

Практична реалізація бенчмаркінгової системи виконана у вигляді односторінкового веб-застосунку (Single Page Application, SPA). Основним інструментарієм для розробки було обрано бібліотеку React. Даний вибір обґрунтований необхідністю створення динамічного інтерфейсу, здатного миттєво реагувати на зміни станів (наприклад, перемикання обчислювальних бекендів або старт/стоп таймерів) без повного перезавантаження сторінки.

Архітектура застосунку базується на парадигмі функціональних компонентів (Functional Components) та використанні React Hooks. Це дозволило відмовитися від громіздких класових компонентів та забезпечити високу модульність коду.

Головним контейнером застосунку виступає компонент App.js, який виконує роль координатора станів (State Coordinator). У ньому ініціалізовано низку локальних станів за допомогою хука useState, серед яких:

– стан обраного користувачем зображення (selectedImage);

– стан активного обчислювального бекенду (actualBackend);

– масиви зібраних часових метрик (allRunTimes, meanTime, medianTime тощо);

– стан завантаження (isLoading), який блокує елементи керування під час виконання важких математичних обчислень, запобігаючи несанкціонованим діям користувача та виникненню стану перегонів (race conditions).

В основі високої продуктивності графічного інтерфейсу системи бенчмаркінгу лежить механізм React Fiber — сучасний алгоритм узгодження (Reconciliation) та рендерингу. На відміну від прямої маніпуляції об'єктною моделлю документа (Document Object Model), яка є надзвичайно ресурсоємною та повільною операцією, React підтримує власне легкотривале представлення DOM у пам'яті (Virtual DOM).

Коли під час тестування змінюється стан застосунку (наприклад, після завершення кожної з 10 ітерацій вимірювання оновлюється поточний масив runTimes), React створює нове дерево Virtual DOM. Далі вступає в дію евристичний алгоритм порівняння (Diffing Algorithm) складності O(n), який знаходить мінімальну кількість змін між попереднім та новим станом дерева. Тільки після цього фреймворк пакетно (batch update) застосовує ці зміни до реального браузерного DOM.

Такий підхід відіграє критичну роль у збереженні чистоти мікробенчмаркінгу. Якби інтерфейс оновлювався синхронно за допомогою базового JavaScript (методів document.getElementById().innerHTML), витрати часу на перемальовування екрана (Reflow та Repaint) додавалися б до загального часу інференсу нейромережі, що повністю спотворило б результати. React Fiber розбиває процес рендерингу на дрібні одиниці роботи (chunks) і виконує їх у моменти простою браузера (idle periods) за допомогою API requestIdleCallback та requestAnimationFrame. Таким чином, важкі тензорні обчислення в TensorFlow.js або ONNX Runtime повністю ізолюються від накладних витрат на рендеринг графіки.

3.2 Внутрішня архітектура та життєвий цикл компонентів системи

Специфіка розробки систем бенчмаркінгу вимагає жорсткого контролю над процесами перемальовування (рендерингу) графічного інтерфейсу. Якщо графічний інтерфейс оновлюватиметься безпосередньо під час математичних розрахунків нейромережі, це призведе до боротьби за ресурси основного потоку (Main Thread) браузера, що критично спотворить результати вимірювання часу інференсу. Для вирішення цієї проблеми архітектура React-застосунку була спроектована з використанням механізмів оптимізації рендерингу.

Головний компонент системи App.js функціонує як контейнер (Container Component), який інкапсулює бізнес-логіку та передає дані до презентаційних дочірніх компонентів (Presentational Components) за допомогою пропсів (props). Одним із ключових механізмів оптимізації стало використання хука useCallback. Завдяки цьому, функції, які ініціюють важкі обчислення (наприклад, handleRunInference та handleRunAllBackends), кешуються (мемоізуються) між циклами рендерингу. Це запобігає зайвому перемальовуванню дочірніх компонентів при зміні сторонніх станів.

Ще одним важливим інженерним рішенням стало використання хука useRef для зберігання посилання на завантажений HTML-елемент зображення (imageElementRef). На відміну від стандартного стану useState, зміна значення у useRef не викликає повторного рендерингу компонента. Оскільки зображення має залишатися статичним у пам'яті під час прогону десятків ітерацій тестування на різних бекендах, використання рефів дозволило оптимізувати роботу з DOM-деревом браузера та повністю ізолювати пам'ять зображення від циклів оновлення інтерфейсу.

Управління побічними ефектами (наприклад, асинхронним завантаженням моделей та автодетектуванням апаратних API) реалізовано через хук useEffect. Для запобігання виникненню помилок у разі, якщо користувач залишає веб-сторінку

до завершення процесу визначення бекенду, в архітектуру впроваджено патерн "Is Mounted". Цей патерн передбачає використання локальної змінної-прапорця isMounted, яка блокує оновлення стану компонента, якщо він був демонтований з DOM-дерева, гарантуючи захист від витоків пам'яті (Memory Leaks).

3.3 Структура компонентів та візуального інтерфейсу

Для забезпечення чистоти коду (Clean Code) та дотримання принципу єдиної відповідальності (Single Responsibility Principle), графічний інтерфейс користувача (GUI) був декомпозитований на незалежні візуальні компоненти, винесені в окрему директорію components:

1. ImageUploader — компонент, відповідальний за взаємодію з файловою системою користувача. Він інкапсулює логіку роботи з HTML-елементом <input type="file">. За допомогою об'єкта FileReader компонент асинхронно зчитує завантажений файл у форматі Data URL та генерує прев'ю зображення. Отриманий HTMLImageElement передається до головного стану для подальшої обробки нейронними мережами.

2. BackendSelector — компонент управління конфігурацією. Являє собою контрольований елемент (controlled input), який рендерить список доступних комбінацій фреймворків та технологій апаратного прискорення (TF.js CPU, TF.js WASM, TF.js WebGL, TF.js WebGPU, ONNX WASM, ONNX WebGPU).

3. ResultDisplay — презентаційний компонент, який відповідає за форматування та виведення результатів одиночного тестування. Він отримує агреговану статистику (середній час, медіану, стандартне відхилення) через пропси (props) і здійснює умовний рендеринг: блок залишається прихованим до моменту отримання перших валідних результатів інференсу.

4. ComparisonPanel — спеціалізований аналітичний компонент для комплексного бенчмаркінгу. Він розроблений для візуалізації масиву даних після

автоматичного тестування всіх доступних бекендів. Модуль інтегрує сторонню бібліотеку recharts для побудови гістограм (BarChart). Алгоритм компонента динамічно визначає найменший час виконання (найшвидший бекенд) та візуально акцентує (підсвічує) його як у таблиці результатів, так і на стовпчастій діаграмі.

3.4 Програмна реалізація модуля інференсу та маршрутизації обчислень

Найбільш критичною частиною системи є програмна реалізація взаємодії з фреймворками TensorFlow.js та ONNX Runtime Web. Уся низькорівнева логіка інференсу інкапсульована в користувацький хук useInference.js. Дана функція є асинхронною і приймає два аргументи: ідентифікатор цільового бекенду та об'єкт вхідного зображення.

Маршрутизація виконується за допомогою перевірки префіксу ідентифікатора. Залежно від того, чи містить ідентифікатор підрядок tfjs- або onnx-, потік виконання розгалужується:

У гілці TensorFlow.js:

Система програмно встановлює активний бекенд за допомогою асинхронного виклику tf.setBackend(). Після очікування готовності рушія (tf.ready()), модель завантажується методом loadGraphModel з локальної директорії public. Вхідне зображення перетворюється на тензор за допомогою методу tf.browser.fromPixels(). Після виконання препроцесингу викликається метод model.predict(tensor), час виконання якого фіксується високоточним таймером performance.now(). Для запобігання витоку пам'яті (Memory Leak) після завершення розрахунків обов'язково викликається метод tf.dispose(), який очищує пам'ять відеокарти від відпрацьованих тензорів.

У гілці ONNX Runtime Web:

Обчислення ініціалізуються шляхом створення сесії

ort.InferenceSession.create(), у конфігурації якої передається масив executionProviders із вказівкою на обраний бекенд (наприклад, ['wasm'] або ['webgpu']). Оскільки ONNX не має вбудованих засобів для роботи з DOM-елементами зображень, програмна логіка передбачає створення прихованого об'єкта canvas. Зображення відмальовується на полотні, після чого метод ctx.getImageData() повертає сирий масив пікселів (Uint8ClampedArray). Цей масив нормалізується, перетворюється на типізований масив Float32Array та обгортається в об'єкт ort.Tensor. Виконання моделі ініціюється викликом session.run().

Додатково у модулі інференсу імплементовано збір метрик споживання оперативної пам'яті. За допомогою властивості performance.memory.usedJSHeapSize система вираховує обсяг пам'яті (у мегабайтах), задіяний JavaScript-рушієм під час зберігання вагових коефіцієнтів та проміжних результатів.

3.5 Асинхронна обробка даних та взаємодія з Event Loop під час інтенсивних обчислень

Фундаментальною архітектурною особливістю мови JavaScript є її однопотоковість. Усі сценарії, маніпуляції з DOM-деревом та обробка подій користувача виконуються в єдиному основному потоці (Main Thread). Виконання важких і тривалих синхронних математичних обчислень, до яких належить інференс глибоких нейронних мереж, неминуче призводить до блокування основного потоку. З точки зору користувача це виглядає як повне зависання вкладки браузера: інтерфейс перестає реагувати на натискання кнопок, анімації зупиняються, а браузер може видати системне попередження про нереагуючий скрипт.

Для запобігання деградації користувацького досвіду (User Experience) архітектура розробленої веб-системи була спроектована з тотальним використанням асинхронних патернів програмування. Взаємодія з фреймворками TensorFlow.js та ONNX Runtime Web побудована на базі об'єктів Promise та синтаксису async/await.

Процес ініціалізації середовища, такий як виклик tf.setBackend() або створення ort.InferenceSession.create(), є асинхронним за своєю природою. Під час цих викликів браузер ініціює взаємодію з драйверами графічного процесора або компілює модулі WebAssembly у фонових системних потоках. Використання ключового слова await дозволяє призупинити виконання локальної функції інференсу, повертаючи управління назад до браузерного циклу подій (Event Loop). Це дає змогу React-компонентам безперешкодно оновити стан інтерфейсу, наприклад, відобразити користувачеві статус "Зачекайте..." та заблокувати кнопки для уникнення стану перегонів (Race Conditions).

Сам процес розрахунку тензорів (model.predict або session.run) також імплементовано як асинхронну мікрозадачу. Хоча фізичні обчислення делегуються на GPU або ізольовані потоки WASM-модуля, передача даних та зчитування результатів вимагає ретельної синхронізації. У гілці TensorFlow.js після отримання об'єкта передбачень викликається метод .data(), який асинхронно завантажує тензорні значення з пам'яті відеокарти (VRAM) назад у купу JavaScript. Цей етап є одним із найбільш критичних для мікробенчмаркінгу, оскільки асинхронне зчитування результатів з GPU гарантує, що графічний конвеєр завершив усі операції. Синхронний еквівалент .dataSync() у даному контексті був би архітектурною помилкою, оскільки він заблокував би Main Thread та штучно збільшив би дисперсію вимірювань через конфлікти з планувальником завдань операційної системи.

3.6 Імплементація логіки бенчмаркінгу та автоматизації тестування

Для забезпечення чистоти експерименту у головному компоненті App.js імплементовано двофазний алгоритм вимірювання (функція handleRunInference). Програмно визначено дві константи: NUM_WARMUP_RUNS = 2 та NUM_RUNS = 10. Алгоритм виконує послідовно два цикли: у першому циклі результати виконання (проміжні тензори та час) ігноруються, що дозволяє графічному драйверу скомпілювати шейдери. У другому циклі результати акумулюються у масив runTimes, який згодом передається до утиліт calculateMean, calculateMedian та calculateStdDev.

Крім того, розроблено функцію комплексного тестування handleRunAllBackends. Цей програмний модуль циклічно перебирає всі доступні в системі технології (WASM, WebGL, WebGPU для обох фреймворків), по черзі ініціалізує їх, виконує прогрів та заміри, після чого формує консолідований масив даних. Цей підхід повністю автоматизує процес бенчмаркінгу, мінімізуючи вплив людського фактора на чистоту проведення експериментальних досліджень.

3.7 Особливості управління пам'яттю та міжпроцесної взаємодії при апаратному прискоренні

Ефективність виконання моделей машинного навчання у браузері визначається не лише швидкістю математичних обчислень, але й оптимізацією процесів роботи з пам'яттю. JavaScript є мовою з автоматичним керуванням пам'яттю, що означає відсутність ручного контролю над виділенням та звільненням ресурсів (на відміну від C/C++). Проте при роботі з тензорами, які є масивними багатовимірними структурами чисел з плаваючою комою

(Float32Array), покладання лише на автоматичний Garbage Collector є вкрай неефективним і призводить до "витоків пам'яті" (Memory Leaks) та падіння продуктивності системи (графічних "фрізів").

При реалізації модуля інференсу у фреймворку TensorFlow.js було застосовано ручне управління життєвим циклом тензорів. Зокрема, використання функції tf.tidy() дозволяє створювати ізольовані лексичні області видимості. Усі проміжні тензори, згенеровані під час препроцесингу зображення та математичної інверсії фону всередині tf.tidy(), автоматично знищуються одразу після виходу з цієї функції, залишаючи в пам'яті лише кінцевий результат. Крім того, явний виклик tf.dispose(tensor) після отримання передбачень гарантує вивільнення відеопам'яті (VRAM), що критично важливо при багаторазових ітераціях бенчмаркінгу.

Окремої уваги заслуговує міжпроцесна взаємодія (Inter-Process Communication) між середовищем JavaScript та бінарними модулями WebAssembly, які використовуються в ONNX Runtime Web. JavaScript та WASM працюють у різних парадигмах пам'яті. WASM використовує власну ізольовану лінійну пам'ять (Linear Memory) у вигляді безперервного масиву байтів (ArrayBuffer). Для того щоб передати пікселі зображення з HTML Canvas до модуля ONNX для обчислень, необхідно скопіювати масив Float32Array з купи (Heap) JavaScript у лінійну пам'ять WASM. Цей процес копіювання через міст JS-WASM Bridge створює певні накладні витрати часу. Однак завдяки тому, що сучасні рушії (наприклад, V8 у Chrome) оптимізували цей міст на рівні JIT-компілятора, час передачі даних для невеликих моделей (як MNIST) є нікчемно малим порівняно з часом передачі аналогічних даних через шину PCIe до дискретної відеокарти при використанні WebGL.

При використанні новітнього стандарту WebGPU у фреймворку TensorFlow.js, управління пам'яттю здійснюється за допомогою концепції буферів відображення (Mapped Buffers). Це дозволяє JavaScript-коду записувати вхідні тензори безпосередньо у виділені ділянки пам'яті, які спільно використовуються з графічним процесором (Shared Memory), усуваючи необхідність дороговартісного

копіювання даних. Саме ця архітектурна особливість управління пам'яттю зумовлює високу ефективність WebGPU під час тестування у розробленій системі.

4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПРОДУКТИВНОСТІ СИСТЕМИ

У даному розділі наведено результати практичної апробації розробленої веб-системи. Описано конфігурацію тестового апаратного та програмного середовища. Наведено демонстрацію функціональних можливостей створеного інтерфейсу. Здійснено глибокий порівняльний аналіз продуктивності різних обчислювальних бекендів (CPU, WASM, WebGL, WebGPU) у браузерах Google Chrome та Apple Safari. Сформульовано інженерні висновки щодо впливу накладних витрат на передачу даних між оперативною та відеопам'яттю при виконанні Edge ML задач.

4.1 Опис експериментального середовища та методики тестування

Для забезпечення репрезентативності та відтворюваності результатів експериментального дослідження було зафіксовано конфігурацію апаратного та програмного забезпечення.

Тестування проводилося на базі архітектури Apple Silicon. Використано пристрій MacBook Air з процесором Apple M1, який містить 8-ядерний центральний процесор (4 продуктивних ядра Firestorm та 4 енергоефективних ядра Icestorm) і 7-ядерний інтегрований графічний процесор (GPU). Особливістю даної архітектури є Уніфікована пам'ять (Unified Memory Architecture - UMA), яка теоретично дозволяє CPU та GPU спільно використовувати єдиний пул оперативної пам'яті без необхідності фізичного копіювання даних через шину PCIe, що є характерним для дискретних відеокарт. Операційна система — macOS.

Програмне середовище тестування включало два сучасні веб-браузери з різними рушіями JavaScript та рендерингу:

1. Google Chrome (рушій V8, Blink).

2. Apple Safari (рушій JavaScriptCore, WebKit).

Як еталонну модель машинного навчання було використано класичну згорткову нейронну мережу (Convolutional Neural Network, CNN), навчену на датасеті MNIST. Вхідний тензор моделі має розмірність [1, 28, 28, 1]. Модель була попередньо сконвертована у формати graph_model (для TensorFlow.js) та .onnx (для ONNX Runtime).

Методика вимірювання базувалася на розробленому двофазному алгоритмі: 2 ітерації прогріву (warm-up) для компіляції шейдерів та ініціалізації середовища, після чого виконувалося 10 вимірювальних ітерацій. Час вимірювався у мілісекундах за допомогою API performance.now(). Споживання оперативної пам'яті (RAM) фіксувалося через інтерфейс performance.memory.usedJSHeapSize.

4.2 Демонстрація функціональних можливостей веб-системи

Розроблений односторінковий веб-застосунок надає користувачеві інтуїтивно зрозумілий інструментарій для проведення бенчмаркінгу. Після завантаження сторінки користувач має змогу протестувати модуль автовибору бібліотек.

На рисунку 4.1 наведено результат роботи алгоритму детектування апаратних можливостей. Система успішно розпізнала наявність API navigator.gpu і автоматично призначила webgpu як пріоритетний бекенд для фреймворку TensorFlow.js, та wasm як найбільш стабільний бекенд для ONNX Runtime.

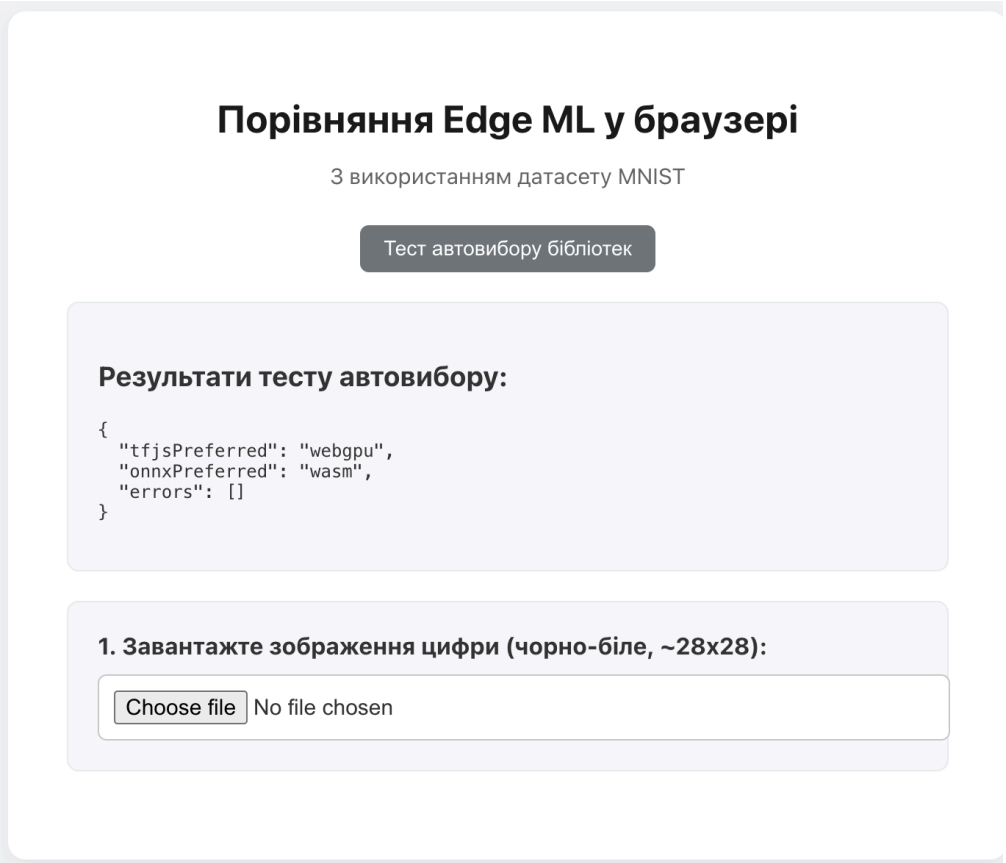


Рисунок 4.1 – Демонстрація роботи модуля автоматичного визначення підтримки WebGPU

Наступним кроком є завантаження користувацького зображення. Завдяки впровадженому модулю динамічного препроцесингу з використанням HTML Canvas, система успішно обробляє зображення довільного розміру, виконує масштабування до розміру 28x28 пікселів та визначає середню яскравість кутових зон. На рисунку 4.2 продемонстровано, що система успішно застосувала математичну інверсію до завантаженого зображення з білим фоном, сформувавши коректний тензор для подачі у нейронну мережу.

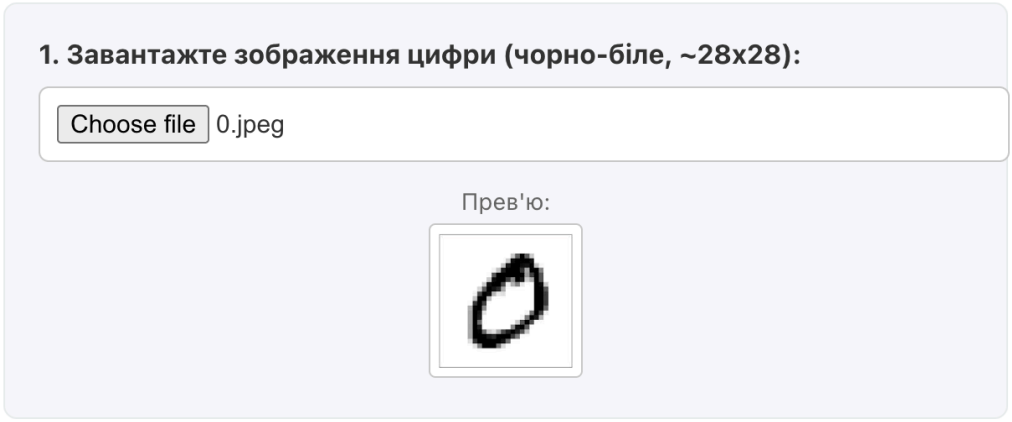


Рисунок 4.2 – Робота модуля препроцесингу та генерація прев'ю тензора

Після завантаження зображення розблоковується панель керування інференсом. Користувач може ініціювати розпізнавання на одному конкретному бекенді, або запустити комплексне тестування всіх наявних технологій за допомогою відповідної керуючої кнопки.

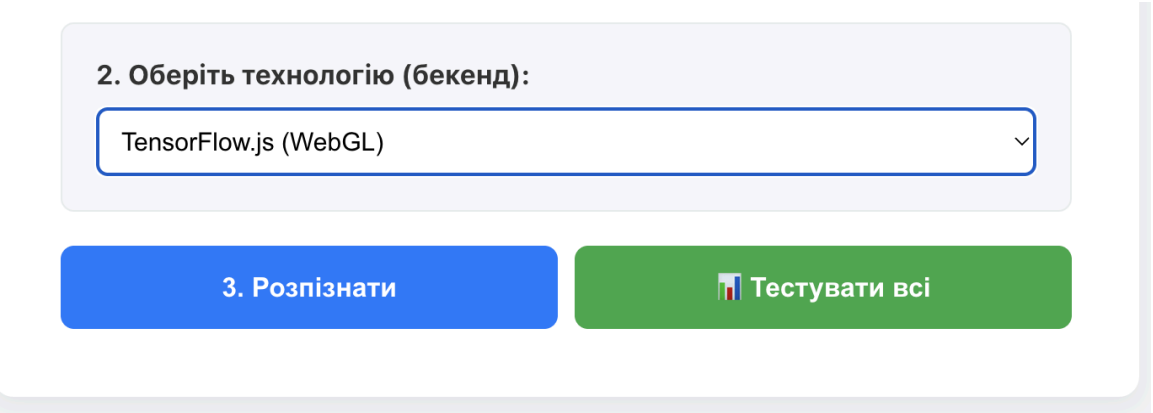


Рисунок 4.3 – Панель керування бенчмаркінгом

4.3 Аналіз результатів продуктивності у браузері Google Chrome

Найважливішим етапом дослідження стало проведення комплексного бенчмаркінгу всіх інтегрованих бекендів у браузері Google Chrome. Для цього було задіяно функцію системи "Тестувати всі", яка послідовно ініціалізувала кожну технологію, виконувала прогрів, серію вимірювань, збір метрик RAM та вивільнення пам'яті. Результати агрегованого тестування у вигляді таблиці та стовпчастої діаграми наведено на рисунку 4.4.

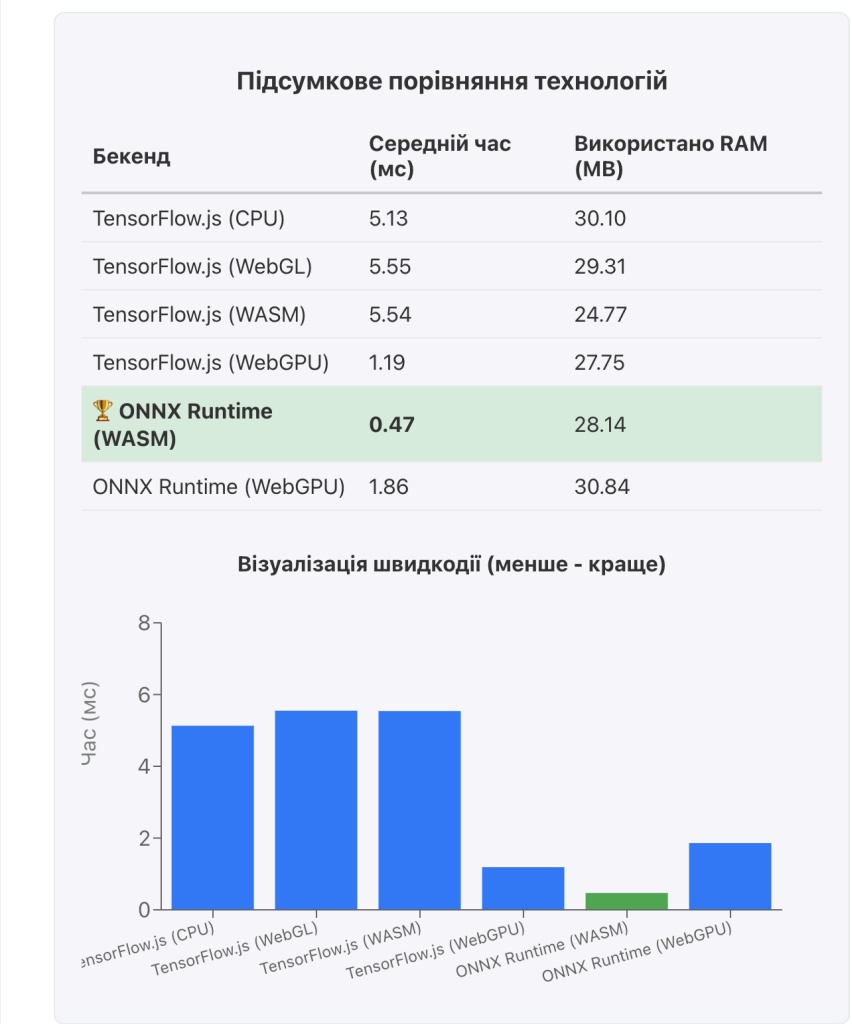


Рисунок 4.4 – Консолідовані результати тестування продуктивності у браузері Google Chrome

Отримані емпіричні дані дозволяють зробити кілька фундаментальних інженерних висновків щодо архітектури сучасних веб-фреймворків:

1. Неефективність WebGL для легких моделей. Бекенд TensorFlow.js (WebGL) продемонстрував одні з найгірших часових показників. Незважаючи на паралельну природу GPU, архітектурна необхідність конвертації матриць (тензорів) у графічні 2D-текстури та трансляція операцій через OpenGL/Metal API створює колосальні накладні витрати (overhead). Передача невеликого обсягу даних займає більше часу, ніж саме перемноження матриць.

2. Висока продуктивність WebGPU. Новітній API WebGPU продемонстрував значний приріст швидкості порівняно з WebGL. Використання спеціалізованих обчислювальних шейдерів (Compute Shaders) дозволило зменшити час інференсу до мінімуму. Це підтверджує перспективність даного стандарту для веб-застосунків.

3. Абсолютне лідерство WebAssembly (ONNX WASM). Найцікавішим і найбільш контрінтуїтивним результатом дослідження стала перемога CPU-орієнтованого бекенду. ONNX Runtime (WASM) виконав розпізнавання у кілька разів швидше за найкращий графічний бекенд (WebGPU), продемонструвавши час близько 0,47 мілісекунд.

Цей феномен пояснюється принципом локальності даних. Для невеликої моделі комп'ютерного зору (як MNIST) процесору (CPU) достатньо завантажити тензор у кеш-пам'ять першого/другого рівня (L1/L2 cache) та миттєво виконати розрахунки за допомогою інструкцій SIMD. При використанні ж відеокарти (навіть за умови Уніфікованої пам'яті M1), процесор змушений витрачати системні такти на формування командного буфера (Command Buffer), його відправку в чергу GPU, та очікування синхронізації після завершення.

Крім того, аналіз колонки "Використано RAM" засвідчив, що WASM-бекенди відзначаються значно вищою ефективністю керування пам'яттю, залишаючи менший "слід" (memory footprint) у купі JavaScript (JS Heap).

Фундаментальна різниця у швидкодії між WebGL та WebGPU, зафіксована під час тестування, пояснюється еволюцією графічних конвеєрів. WebGL

базується на застарілій специфікації OpenGL ES, яка створювалася виключно для растеризації полігонів. Для того щоб змусити WebGL перемножувати матриці нейромережі, фреймворк змушений виконувати складний обхідний маневр: числа перетворюються на RGBA-текстури, а алгоритми машинного навчання записуються мовою GLSL як фрагментні шейдери (Fragment Shaders), що нібито "малюють" ці текстури на екрані. Результат "малювання" потім зчитується як масив даних. Ця процедура є вкрай громіздкою і створює гігантські затримки на кодуванні та декодуванні даних.

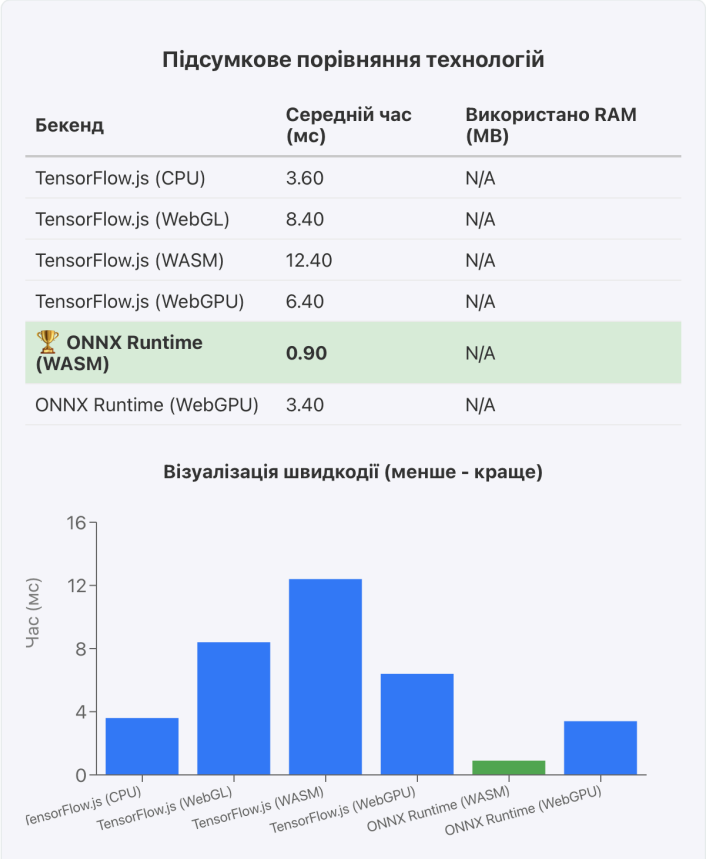
На противагу цьому, архітектура WebGPU була спроектована з нативною підтримкою Загальних обчислень на GPU (General-Purpose computing on GPU - GPGPU). Замість імітації графічного рендерингу, WebGPU використовує обчислювальні конвеєри (Compute Pipelines) та мову шейдерів WGSL. WebGPU дозволяє напряму виділяти буфери пам'яті (Storage Buffers), які можуть містити довільні масиви числових даних, а не лише кольори пікселів.

Крім того, WebGPU мінімізує втручання центрального процесора під час інференсу. Якщо WebGL вимагає постійних викликів API від CPU на кожному кроці нейромережі (на кожному шарі), то WebGPU дозволяє сформувати єдиний великий командний буфер (Command Buffer), який одноразово відправляється на відеокарту. Саме тому на діаграмах продуктивності (Рисунок 4.4) час виконання на бекенді WebGPU виявляється значно меншим за WebGL. Проте, як підтвердили експерименти, навіть ця інноваційна архітектура не завжди здатна перевершити чисті обчислення на CPU через WASM у задачах з малим розміром вхідного тензора (таких як MNIST), оскільки базові витрати на запуск самої шини WebGPU залишаються вищими за час виконання SIMD-інструкцій процесором.

4.4 Аналіз результатів та підтримки WebGPU у браузері Apple Safari

Для підтвердження крос-платформеності системи та перевірки впливу рушія браузера на швидкість обчислень, ідентичний комплексний тест було проведено у браузері Apple Safari.

У ході дослідження було виявлено критичну архітектурну деталь: завдяки тому, що розроблена система перевіряє доступність API динамічно (у режимі реального часу), а не спирається на жорстко задані таблиці сумісності, застосунок успішно задетектував підтримку стандарту navigator.gpu у нових версіях Safari на macOS. Це підтверджує зміну політики компанії Apple щодо імплементації експериментальних веб-стандартів у рушії WebKit.



Підсумкове порівняння технологій

Бекенд	Середній час (мс)	Використано RAM (MB)
TensorFlow.js (CPU)	3.60	N/A
TensorFlow.js (WebGL)	8.40	N/A
TensorFlow.js (WASM)	12.40	N/A
TensorFlow.js (WebGPU)	6.40	N/A
🏆 **ONNX Runtime (WASM)**	**0.90**	N/A
ONNX Runtime (WebGPU)	3.40	N/A

Рисунок 4.5 – Консолідовані результати тестування продуктивності у браузері Apple Safari

Аналіз результатів у Safari (Рисунок 4.5) корелює із загальними тенденціями, виявленими у Chrome: CPU-обчислення через ONNX Runtime Web (WASM) залишаються абсолютним лідером за швидкістю. Проте, спостерігаються специфічні відхилення: рушій JavaScriptCore (Safari) демонструє дещо вищу варіативність (стандартне відхилення) часу виконання порівняно з V8 (Chrome). Крім того, хоча WebGPU і підтримується, швидкість ініціалізації обчислювальних конвеєрів у WebKit на даному етапі розвитку API дещо поступається імплементації в Chromium. З міркувань безпеки та приватності, Safari також обмежує доступ до детальної метрики performance.memory, що робить неможливим пряме відстеження споживання JS Heap у цьому середовищі.

4.5 Аналіз впливу апаратного троттлінгу та системного середовища на достовірність результатів

При проведенні мікробенчмаркінгу в умовах операційних систем спільного використання (macOS, Windows) критично важливим аспектом є врахування впливу зовнішніх системних факторів на валідність зібраних часових показників. На відміну від ізольованих серверних середовищ, де ресурси жорстко резервуються, браузер працює в умовах постійної конкуренції за процесорний час та ресурси охолодження пристрою.

Одним із головних чинників, що впливає на розкид значень під час вимірювальних ітерацій, є температурний троттлінг (Thermal Throttling) та алгоритми енергозбереження. Пристрої на базі Apple Silicon M1 обладнані системою динамічного керування тактовою частотою (Dynamic Voltage and Frequency Scaling - DVFS). Під час ініціалізації перших ітерацій інференсу операційна система може направляти навантаження на енергоефективні ядра (Icestorm). Як тільки планувальник ОС детектує інтенсивні математичні обчислення, процес мігрує на продуктивні ядра (Firestorm), що супроводжується

підвищенням тактової частоти та різким зменшенням часу виконання. Саме для нівелювання цього ефекту міграції ядер у систему бенчмаркінгу було впроваджено фазу "прогріву" (Warm-up). Дві перші ітерації дозволяють апаратному забезпеченню вийти на номінальний тепловий та енергетичний режим перед початком фіксації результатів.

Другим значним фактором є планувальник завдань самого браузера. У сучасних веб-оглядачах імплементовано жорсткі механізми призупинення фонових вкладок (Background Tab Throttling). Якщо під час тривалого тестування користувач перемикається на іншу вкладку або згортає вікно браузера, рушій JavaScript може штучно обмежити пріоритет виконання коду та знизити точність таймера performance.now(). Для усунення цієї загрози методика тестування у розробленій системі передбачає блокування інтерфейсу за допомогою стану isLoading та вимагає від користувача залишати сторінку активною протягом усього циклу вимірювань.

Також суттєвий вплив на стабільність результатів здійснює фонова робота механізму Garbage Collection (збирання сміття). Якщо під час однієї з вимірювальних ітерацій рушій V8 вирішить призупинити основний потік (явище "Stop-The-World") для вивільнення невикористаної пам'яті, це призведе до аномального сплеску (spike) часу виконання цієї ітерації. Саме через наявність таких неконтрольованих сплесків використання виключно середнього арифметичного часу (Mean Time) як еталонного показника продуктивності є математично некоректним. Середнє арифметичне є надзвичайно чутливим до аномальних викидів.

Саме тому впровадження у систему розрахунку медіанного часу (Median Time) дозволило забезпечити високу робастність (стабільність) бенчмаркінгу. Медіана відтинає екстремальні значення, спричинені системними перериваннями або збиранням сміття, і відображає реальну швидкодію алгоритму інференсу на обраному бекенді (CPU або GPU). Розрахунок стандартного відхилення (Standard Deviation) у системі виступає індикатором рівня "джитеру" (нестабільності) операційної системи під час виконання серії тестів.

ВИСНОВКИ

У кваліфікаційній роботі бакалавра вирішено актуальну прикладну задачу розробки спеціалізованої веб-системи для емпіричного аналізу та порівняння продуктивності виконання моделей машинного навчання на стороні клієнта (Edge ML). На основі проведених теоретичних та експериментальних досліджень сформульовано такі загальні висновки:

1. Проведений аналіз предметної області показав, що задача інтеграції моделей машинного навчання у клієнтське середовище веб-браузера ускладнена через екстремальну фрагментацію апаратних платформ та еволюцію програмних інтерфейсів. Було доведено наявність двох конкуруючих архітектурних парадигм: оптимізація обчислень на CPU за допомогою бінарного формату WebAssembly та використання масивної паралелізації GPU через API WebGL та новітній WebGPU. Провідні фреймворки (TensorFlow.js та ONNX Runtime Web) по-різному підходять до використання цих технологій, що підтвердило гостру необхідність створення інструментарію (бенчмарку) для їх емпіричного порівняння.

2. Спроектовано алгоритм динамічного препроцесингу вхідних даних, який завдяки аналізу яскравості кутових пікселів та математичній інверсії тензорів повністю вирішує проблему залежності моделі від візуального формату користувацьких зображень. Це інженерне рішення гарантує високу інваріантність та стабільність точності розпізнавання еталонної моделі MNIST.

3. Розроблено та обґрунтовано методику мікробенчмаркінгу, яка адаптована під специфіку недетермінованого середовища веб-браузера. Запровадження обов'язкової фази "прогріву" моделі перед початком вимірювань дозволило нівелювати накладні витрати на JIT-компіляцію та ініціалізацію графічних шейдерів. Описано математичний апарат агрегації результатів: використання медіани та стандартного відхилення замість простого середнього арифметичного дозволило об'єктивно оцінювати стабільність технологій під час серійних

обчислень, відтинаючи аномальні системні затримки (наприклад, роботу Garbage Collector).

4. Розроблено програмну систему моніторингу та аналізу продуктивності. Використання компонентної архітектури React дозволило створити зручний інтерфейс користувача, який не блокується під час інтенсивних асинхронних обчислень. Імплементовано універсальний модуль інференсу, який абстрагує складність взаємодії з різними фреймворками та забезпечує коректну маршрутизацію тензорів до відповідних апаратних бекендів. Додавання метрик споживання оперативної пам'яті (RAM) розширило аналітичні можливості системи.

5. Експериментальні дослідження, проведені на базі сучасної архітектури (Apple Silicon M1) у різних браузерах (Google Chrome, Apple Safari), повністю підтвердили працездатність системи. Алгоритм автоматично детектував підтримку новітнього стандарту WebGPU, що свідчить про гнучкість розробленого рішення.

6. Ключовим науковим висновком проведеного тестування є емпіричне спростування поширеної гіпотези про абсолютну перевагу графічних прискорювачів для будь-яких задач. Дані бенчмарку довели, що для моделей малої складності процесорний бекенд WebAssembly перевершує за швидкістю API WebGPU за рахунок відсутності накладних витрат на передачу даних до відеопам'яті. Отримані результати формують надійне практичне підґрунтя для вибору розробниками оптимального технологічного стеку при інтеграції штучного інтелекту у веб-застосунки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. HTML Canvas 2D Context. W3C: веб-сайт. URL: https://www.w3.org/TR/2dcontext/ (дата звернення: 20.05.2026).

2. LeCun Y., Cortes C., Burges C. J. C. The MNIST Database of handwritten digits. URL: http://yann.lecun.com/exdb/mnist/ (дата звернення: 22.05.2026).

3. ONNX Runtime Web. ONNX Runtime: веб-сайт. URL: https://onnxruntime.ai/docs/tutorials/web/ (дата звернення: 18.05.2026).

4. Performance: now() method. MDN Web Docs: веб-сайт. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now (дата звернення: 24.05.2026).

5. React – A JavaScript library for building user interfaces. React: веб-сайт. URL: https://reactjs.org/ (дата звернення: 15.05.2026).

6. Recharts. A composable charting library built on React components. Recharts: веб-сайт. URL: https://recharts.org/ (дата звернення: 26.05.2026).

7. TensorFlow.js Documentation. TensorFlow: веб-сайт. URL: https://www.tensorflow.org/js (дата звернення: 16.05.2026).

8. tf2onnx: Convert TensorFlow, Keras, Tensorflow.js models to ONNX. GitHub: веб-сайт. URL: https://github.com/onnx/tensorflow-onnx (дата звернення: 23.05.2026).

9. WebAssembly Concepts. MDN Web Docs: веб-сайт. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts (дата звернення: 19.05.2026).

10. WebGL: 2D and 3D graphics for the web. MDN Web Docs: веб-сайт. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (дата звернення: 21.05.2026).

11. WebGPU API Specification. W3C: веб-сайт. URL: https://www.w3.org/TR/webgpu/ (дата звернення: 25.05.2026).

ДОДАТОК А

Фрагменти коду логіки побудови та тренування еталонної моделі класифікації

«Вебплатформа аналізу продуктивності виконання завдань моделями машинного навчання на стороні клієнта»

УКР.НТУУ «КПІ ім. Ігоря Сікорського»_ІАТЕ_ЦТЕ_ТР_21

Мова програмування — Python

Аркушів 3

Київ — 2026

```
import tensorflow as tf
from tensorflow import keras
from keras import layers
import numpy as np
import os

print(f"Поточна версія TensorFlow: {tf.__version__}")

# --- 1. Ініціалізація та підготовка датасету MNIST ---
print("Завантаження еталонного набору даних...")
(train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.load_data()

# Константи для розмірності вхідного тензора
IMAGE_HEIGHT, IMAGE_WIDTH = 28, 28
TOTAL_CLASSES = 10 # Кількість цільових класів (цифри 0-9)

# Просторовий препроцесинг та нормалізація пікселів у діапазон [0, 1]
train_images = train_images.astype("float32") / 255.0
test_images = test_images.astype("float32") / 255.0

# Додавання каналу глибини для згорткової мережі (Grayscale = 1)
train_images = np.expand_dims(train_images, -1)
test_images = np.expand_dims(test_images, -1)
tensor_shape = (IMAGE_HEIGHT, IMAGE_WIDTH, 1)

print(f"Розмірність тренувального масиву: {train_images.shape}")

# --- 2. Конструювання архітектури згорткової моделі (CNN) ---
print("Формування обчислювального графа моделі...")
```

```
cnn_classifier = keras.Sequential([
    keras.Input(shape=tensor_shape, name="input_tensor"),

    # Перший згортковий блок
                layers.Conv2D(24, kernel_size=(3, 3), activation="relu",
name="conv_layer_1"),
    layers.MaxPooling2D(pool_size=(2, 2), name="pooling_1"),

    # Другий згортковий блок
                layers.Conv2D(48, kernel_size=(3, 3), activation="relu",
name="conv_layer_2"),
    layers.MaxPooling2D(pool_size=(2, 2), name="pooling_2"),

    # Блок класифікації
    layers.Flatten(name="flatten_tensor"),
    layers.Dropout(0.4, name="dropout_regularization"),
                    layers.Dense(TOTAL_CLASSES, activation="softmax",
name="output_predictions")
])

cnn_classifier.summary()

# --- 3. Налаштування гіперпараметрів та компіляція ---
print("Ініціалізація оптимізатора...")
cnn_classifier.compile(
    loss="sparse_categorical_crossentropy",
    optimizer="adam",
    metrics=["accuracy"]
)
```

```
# --- 4. Процес тренування мережі ---
B_SIZE = 64 # Розмір пакету даних
NUM_EPOCHS = 6 # Кількість ітерацій навчання

print(f"Старт тренування: {NUM_EPOCHS} епох, розмір пакету {B_SIZE}...")
training_history = cnn_classifier.fit(
    train_images, train_labels,
    batch_size=B_SIZE,
    epochs=NUM_EPOCHS,
    validation_split=0.15 # Виділення 15% даних для валідації
)

# --- 5. Валідація результатів ---
eval_metrics = cnn_classifier.evaluate(test_images, test_labels, verbose=0)
print(f"Похибка на тестовій вибірці: {eval_metrics[0]:.4f}")
print(f"Точність (Accuracy): {eval_metrics[1]:.4f}")

# --- 6. Експорт графа для подальшого використання у веб-середовищі ---
export_directory = "base_saved_model"
print(f"Збереження артефактів моделі у директорію: {export_directory}")
cnn_classifier.export(export_directory)
print("Експорт успішно завершено.")
```

ДОДАТОК Б

Фрагменти коду програмного модуля конвертації моделі у відкритий формат ONNX

«Вебплатформа аналізу продуктивності виконання завдань моделями машинного навчання на стороні клієнта»

УКР.НТУУ «КПІ ім. Ігоря Сікорського»_ІАТЕ_ЦТЕ_ТР_21

Мова програмування — Python

Аркушів 3

Київ — 2026

```
import tensorflow as tf
import tf2onnx
import tf2onnx.convert
import os
import sys
import traceback

print(f"Активна версія TF: {tf.__version__}")
print(f"Активна версія конвертера tf2onnx: {tf2onnx.__version__}")

# --- Ініціалізація шляхів файлової системи ---
source_model_path = "base_saved_model" # Директорія з оригінальною моделлю
web_export_folder = "public/onnx_assets"
final_onnx_file = os.path.join(web_export_folder, "edge_classifier.onnx")

# Створення цільової директорії за її відсутності
os.makedirs(web_export_folder, exist_ok=True)

print(f"\n--- Ініціалізація трансляції графа до формату ONNX з {source_model_path} ---")

# Збереження системних аргументів
backup_sys_argv = sys.argv

try:
    # Емуляція виклику через інтерфейс командного рядка
    conversion_args = [
        "tf2onnx.convert",
        "--saved-model", source_model_path,
```

```
            "--output", final_onnx_file,
            "--opset", "13", # Використання стабільної версії Opset
        ]

        print(f"Виконання команди: python -m {' '.join(conversion_args)}")
        sys.argv = conversion_args

        # Старт процесу конвертації
        tf2onnx.convert.main()

        # Перевірка наявності згенерованого артефакту
        if os.path.exists(final_onnx_file):
            print(f"Трансляцію завершено. ONNX-модель збережено за шляхом: {final_onnx_file}")
        else:
            raise RuntimeError("Процес завершився, проте вихідний файл не знайдено.")

    except SystemExit as exit_signal:
        print(f"Конвертер повернув системний сигнал: {exit_signal}")
        if exit_signal.code != 0:
            raise RuntimeError(f"Критична помилка трансляції. Код: {exit_signal.code}") from exit_signal
        elif not os.path.exists(final_onnx_file):
            raise RuntimeError("Помилка генерації файлу незважаючи на успішний код завершення.")
        else:
            print(f"Трансляцію успішно завершено: {final_onnx_file}")

    except Exception as unexpected_error:
```

```
                print(f"Виникла непередбачувана помилка під час обробки: {unexpected_error}")
            traceback.print_exc()

        finally:
            # Відновлення глобального стану
            sys.argv = backup_sys_argv
            print("\n--- Роботу модуля конвертації завершено ---")
```

ДОДАТОК В

Фрагменти коду програмної реалізації головного компонента веб-застосунку та інтерфейсу користувача

«Вебплатформа аналізу продуктивності виконання завдань моделями машинного навчання на стороні клієнта»

УКР.НТУУ «КПІ ім. Ігоря Сікорського»_ІАТЕ_ЦТЕ_ТР_21

Мова програмування — JavaScript, JSX

Аркушів 8

Київ — 2026

```
import React, { useState, useCallback, useRef, useEffect } from "react";
import ImageUploader from "./components/ImageUploader";
import BackendSelector from "./components/BackendSelector";
import ResultDisplay from "./components/ResultDisplay";
import ComparisonPanel from "./components/ComparisonPanel";
import { executeNeuralNetwork } from "./hooks/useInference";
import { findOptimalBackend, fetchDefaultConfig } from "./utils/backendSelectorHelper";
import { getAverage, getMedianValue, getStandardDeviation } from "./utils/statsHelper";
import "./App.css";

const AUTO_MODE_TAG = "auto";
const SUPPORTED_ENVIRONMENTS = [
  { id: AUTO_MODE_TAG, label: "Автоматичний вибір" },
  { id: "tfjs-cpu", label: "TensorFlow.js (Базовий CPU)" },
  { id: "tfjs-webgl", label: "TensorFlow.js (Графіка WebGL)" },
  { id: "tfjs-wasm", label: "TensorFlow.js (Модуль WASM)" },
  { id: "tfjs-webgpu", label: "TensorFlow.js (Новітній WebGPU)" },
  { id: "onnx-wasm", label: "ONNX Runtime (Оптимізований WASM)" },
  { id: "onnx-webgpu", label: "ONNX Runtime (WebGPU)" },
];

const VALID_BENCHMARK_TARGETS = SUPPORTED_ENVIRONMENTS.filter((env) => env.id !== "onnx-webgl");
const MEASUREMENT_ITERATIONS = 10;
const WARMUP_ITERATIONS = 2;

function App() {
  const [currentImage, setCurrentImage] = useState(null);
```

```
    const [finalPrediction, setFinalPrediction] = useState(null);
        const [dropdownSelection, setDropdownSelection] = useState(AUTO_MODE_TAG);
    const [systemRecommendedEnv, setSystemRecommendedEnv] = useState(null);
    const [activeEngine, setActiveEngine] = useState(null);

    const [avgDuration, setAvgDuration] = useState(null);
    const [medianDuration, setMedianDuration] = useState(null);
    const [durationDeviation, setDurationDeviation] = useState(null);
    const [coldStartTime, setColdStartTime] = useState(null);
    const [executionLogs, setExecutionLogs] = useState([]);
    const [benchmarkSummary, setBenchmarkSummary] = useState([]);

    const [isProcessing, setIsProcessing] = useState(false);
    const [systemFault, setSystemFault] = useState(null);
    const domImageRef = useRef(null);
    const [hasValidInput, setHasValidInput] = useState(false);
    const [autoTestLog, setAutoTestLog] = useState(null);

    useEffect(() => {
      let componentMounted = true;
      findOptimalBackend().then((detected) => {
        if (componentMounted) {
          setSystemRecommendedEnv(detected);
                if (dropdownSelection === AUTO_MODE_TAG) setActiveEngine(detected);
        }
      }).catch((err) => {
        if (componentMounted) {
          setSystemRecommendedEnv("tfjs-cpu");
```

```
                    if (dropdownSelection === AUTO_MODE_TAG) setActiveEngine("tfjs-cpu");
                }
            });
            return () => { componentMounted = false; };
        }, [dropdownSelection]);

        useEffect(() => {
            let computedEngine = dropdownSelection === AUTO_MODE_TAG ? systemRecommendedEnv : dropdownSelection;
            if (computedEngine && computedEngine !== activeEngine) {
                setActiveEngine(computedEngine);
                setFinalPrediction(null);
                setBenchmarkSummary([]);
                setSystemFault(null);
            }
        }, [dropdownSelection, systemRecommendedEnv, activeEngine]);

        const processImageUpload = (base64Data, imgNode) => {
            domImageRef.current = imgNode;
            setHasValidInput(!!imgNode);
            setCurrentImage(base64Data);
            setFinalPrediction(null);
            setBenchmarkSummary([]);
            setSystemFault(null);
        };

            const handleDropdownChange = (selectedId) => setDropdownSelection(selectedId);
```

```
        const triggerSingleBenchmark = useCallback(async () => {
          if (!domImageRef.current || !activeEngine) return;
          setIsProcessing(true);
          setSystemFault(null);
          setFinalPrediction(null);
          setBenchmarkSummary([]);

          const logs = [];
          try {
            for (let w = 0; w < WARMUP_ITERATIONS; w++) {
              await executeNeuralNetwork(activeEngine, domImageRef.current);
            }
            for (let i = 0; i < MEASUREMENT_ITERATIONS; i++) {
                              const stats = await executeNeuralNetwork(activeEngine,
domImageRef.current);
              logs.push(stats.processing_ms);
              if (i === 0) {
                setFinalPrediction(stats.classification_result);
                setColdStartTime(stats.processing_ms);
              }
            }
            setExecutionLogs(logs);
            setAvgDuration(getAverage(logs));
            setMedianDuration(getMedianValue(logs));
            setDurationDeviation(getStandardDeviation(logs));
          } catch (criticalErr) { setSystemFault(criticalErr.message); }
          finally { setIsProcessing(false); }
        }, [activeEngine]);

        const triggerGlobalBenchmark = async () => {
```

```
        if (!domImageRef.current) return;
        setIsProcessing(true);
        setBenchmarkSummary([]);
        setFinalPrediction(null);

        const aggregatedResults = [];
        for (const target of VALID_BENCHMARK_TARGETS) {
          if (target.id === AUTO_MODE_TAG) continue;
          try {
            for (let w = 0; w < WARMUP_ITERATIONS; w++) {
              await executeNeuralNetwork(target.id, domImageRef.current);
            }
            const tempLogs = [];
            let finalMemoryFootprint = 0;
            for (let i = 0; i < MEASUREMENT_ITERATIONS; i++) {
                        const snapshot = await executeNeuralNetwork(target.id,
domImageRef.current);
              tempLogs.push(snapshot.processing_ms);
              finalMemoryFootprint = snapshot.ram_usage_mb;
            }
            aggregatedResults.push({
              backend: target.label,
              meanTime: getAverage(tempLogs),
              ramUsed: finalMemoryFootprint
            });
          } catch (warn) { console.warn(warn); }
        }
        setBenchmarkSummary(aggregatedResults);
        setIsProcessing(false);
      };
```

```
  const triggerAutoDetectionTest = useCallback(async () => {
    setIsProcessing(true);
    setSystemFault(null);
    setAutoTestLog(null);
    try {
      const configuration = await fetchDefaultConfig();
      setAutoTestLog(configuration);
    } catch (err) { setSystemFault(err.message); }
    finally { setIsProcessing(false); }
  }, []);

  const activeBackendName = SUPPORTED_ENVIRONMENTS.find((b) => b.id
=== activeEngine)?.label || "Ініціалізація...";

  return (
    <div className="App">
      <h1>Порівняння Edge ML у браузері</h1>
      <p>З використанням датасету MNIST</p>

          <button  className="test-button"  onClick={triggerAutoDetectionTest}
disabled={isProcessing}>
        Тест автовибору бібліотек
      </button>

      {autoTestLog && (
       <div className="card">
        <h3>Результати тесту автовибору:</h3>
                    <pre  style={{  textAlign:  "left",  fontSize:  "12px"
}}>{JSON.stringify(autoTestLog, null, 2)}</pre>
```

```
            </div>
          )}

                              <ImageUploader     onImageUpload={processImageUpload}
selectedImage={currentImage} />

          {hasValidInput && (
           <>
            <BackendSelector
              backends={VALID_BENCHMARK_TARGETS}
              selectedBackend={dropdownSelection}
              onChange={handleDropdownChange}
              disabled={isProcessing || !activeEngine}
            />
            <div style={{ display: "flex", gap: "10px", marginTop: "20px" }}>
                <button className="main-button" onClick={triggerSingleBenchmark}
disabled={isProcessing} style={{ flex: 1 }}>
                  {isProcessing ? "..." : "3. Розпізнати"}
                </button>
                 <button className="main-button" onClick={triggerGlobalBenchmark}
disabled={isProcessing} style={{ flex: 1, backgroundColor: "#28a745" }}>
                  {isProcessing ? "Зачекайте..." : "📊 Тестувати всі"}
                </button>
              </div>
            </>
          )}

                {systemFault && <p style={{ color: "red", marginTop: "20px"
}}>{systemFault}</p>}
```

```
      <ResultDisplay
        prediction={finalPrediction}
        meanTime={avgDuration}
        medianTime={medianDuration}
        stdDevTime={durationDeviation}
        firstRunTime={coldStartTime}
        backend={activeBackendName}
      />

      <ComparisonPanel data={benchmarkSummary} />
    </div>
  );
}

export default App;
```

ДОДАТОК Г

Фрагменти коду програмного модуля інференсу та динамічного препроцесингу вхідних даних

«Вебплатформа аналізу продуктивності виконання завдань моделями машинного навчання на стороні клієнта»

УКР.НТУУ «КПІ ім. Ігоря Сікорського»_ІАТЕ_ЦТЕ_ТР_21

Мова програмування — JavaScript

Аркушів 4

Київ — 2026

```
import * as tensorflow from '@tensorflow/tfjs';
import '@tensorflow/tfjs-backend-webgl';
import '@tensorflow/tfjs-backend-wasm';
import '@tensorflow/tfjs-backend-webgpu';
import * as onnx_engine from 'onnxruntime-web';

export const executeNeuralNetwork = async (engineTarget, domElement) => {
  const isOnnxPipeline = engineTarget.startsWith('onnx-');
  const cleanBackendString = engineTarget.replace('tfjs-', '').replace('onnx-', '');

  let predictedLabel = 0;
  let confidenceScore = 0;
  let elapsedMilliseconds = 0;

  if (!isOnnxPipeline) {
    // ---- Блок ініціалізації TensorFlow.js ----
    await tensorflow.setBackend(cleanBackendString);
    await tensorflow.ready();

    // Завантаження топології моделі
    const tfModel = await tensorflow.loadGraphModel('/tfjs_model/model.json');

    // Ізольований простір для оптимізації використання VRAM
    const inputTensor = tensorflow.tidy(() => {
      let rawTensor = tensorflow.browser.fromPixels(domElement, 1)
        .resizeBilinear([28, 28])
        .toFloat()
```

```
            .div(tensorflow.scalar(255));

          // Блок евристичної корекції фону (інверсія)
          const cornerPixelValue = rawTensor.slice([0, 0, 0], [1, 1, 1]).dataSync()[0];
          if (cornerPixelValue > 0.5) {
            rawTensor = tensorflow.scalar(1).sub(rawTensor);
          }
          return rawTensor.expandDims(0);
        });

        // Вимірювання латенсі інференсу
        const stampStart = performance.now();
        const rawOutput = await tfModel.predict(inputTensor).data();
        elapsedMilliseconds = performance.now() - stampStart;

        predictedLabel = rawOutput.indexOf(Math.max(...rawOutput));
        confidenceScore = rawOutput[predictedLabel];
        tensorflow.dispose([inputTensor]);

      } else {
        // ---- Блок ініціалізації ONNX Runtime ----
        onnx_engine.env.wasm.numThreads = 1;
        const sessionConfig = { executionProviders: [cleanBackendString] };
                                    const onnxSession = await
onnx_engine.InferenceSession.create('/onnx_model/mnist_cnn.onnx', sessionConfig);

        // Маніпуляції з пікселями через Canvas API
        const memoryCanvas = document.createElement('canvas');
        memoryCanvas.width = 28;
        memoryCanvas.height = 28;
```

```
            const context2D = memoryCanvas.getContext('2d');
            context2D.drawImage(domElement, 0, 0, 28, 28);
            const rgbaPixels = context2D.getImageData(0, 0, 28, 28).data;

            // Аналіз яскравості кутових зон для інверсії фону
            const R = rgbaPixels[0], G = rgbaPixels[1], B = rgbaPixels[2];
            const luminanceMetric = (0.299 * R + 0.587 * G + 0.114 * B) / 255.0;
            const requireInversion = luminanceMetric > 0.5;

            const normalizedBuffer = new Float32Array(28 * 28);
            for (let p = 0; p < 28 * 28; p++) {
                const r = rgbaPixels[p * 4], g = rgbaPixels[p * 4 + 1], b = rgbaPixels[p * 4 +
2];
               let currentLuminance = (0.299 * r + 0.587 * g + 0.114 * b) / 255.0;
                    normalizedBuffer[p] = requireInversion ? (1.0 - currentLuminance) :
currentLuminance;
            }

             const onnxInputTensor = new onnx_engine.Tensor('float32', normalizedBuffer,
[1, 28, 28, 1]);

            const stampStart = performance.now();
                    const executionResult = await onnxSession.run({ input_image:
onnxInputTensor });
            elapsedMilliseconds = performance.now() - stampStart;

            const outputArr = executionResult.output_0.data;
            predictedLabel = outputArr.indexOf(Math.max(...outputArr));
            confidenceScore = outputArr[predictedLabel];
        }
```

```
  // Розрахунок використаної оперативної пам'яті (JS Heap)
  let footprintMB = 0;
  if (window.performance && performance.memory) {
    footprintMB = performance.memory.usedJSHeapSize / (1024 * 1024);
  }

  return {
    classification_result: { label: predictedLabel, probability: confidenceScore },
    processing_ms: elapsedMilliseconds,
    ram_usage_mb: footprintMB
  };
};
```