Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики

Кваліфікаційна робота
освітній ступінь - бакалавр

на тему: «**ОПТИМІЗАЦІЯ НЕЙРОННОЇ МЕРЕЖІ ДЛЯ РОБОТИ В РОЗШИРЕННІ IOS-ЗАСТОСУНКУ**»

Виконав: студент 4-го року навчання,
спеціальності 121 Інженерія
програмного забезпечення
Гібський Владислав Ігорович
Керівник: Франків О.О.,
старший викладач
Рецензент _____
Кваліфікаційна робота захищена
з оцінкою _____
Секретар ЕК _____
« ___ » _____ 2026 р.

Київ - 2026

ЗМІСТ

ПЕРЕЛІК ПРИЙНЯТИХ СКОРОЧЕНЬ

- ШІ (AI - Artificial Intelligence) - штучний інтелект.
- ANE (Apple Neural Engine) - спеціалізований співпроцесор в архітектурі Apple Silicon, призначений для нейронних обчислень.
- API (Application Programming Interface) - прикладний програмний інтерфейс.
- CPU (Central Processing Unit) - центральний процесор.
- GPU (Graphics Processing Unit) - графічний процесор.
- FP (Floating Point) - формат числа з плаваючою точкою
- LLM (Large Language Model) - велика мовна модель.
- ML (Machine Learning) - машинне навчання.
- NLP (Natural Language Processing) - обробка природної мови.
- RAM (Random Access Memory) - оперативна пам'ять.
- Core ML - базовий фреймворк машинного навчання від компанії Apple для інтеграції моделей в екосистему iOS/macOS.
- MLX - фреймворк для машинного навчання від Apple Research, оптимізований для виконання на архітектурі пам'яті Apple Silicon.

ВСТУП

Стрімкий розвиток великих мовних моделей у останні роки кардинально змінив підходи до обробки тексту у програмних застосунках. Водночас домінуюча парадигма використання LLM передбачає їх розгортання на хмарних серверах, що породжує низку фундаментальних проблем, включно з залежністю від стабільного з'єднання, мережевими затримками та, найголовніше, ризиками розкриття приватної інформації користувачів.

Однак перенесення цієї парадигми на iOS-розширення, які є природним місцем інтеграції інтелектуальної обробки тексту в системі (зокрема, через Custom Keyboard Extension), наражається на принципові архітектурні обмеження. Операційна система iOS накладає на процеси розширень жорсткі ліміти оперативної пам'яті, що контролюються системним механізмом Jetsam і призводять до негайного завершення процесу при їх перевищенні. У результаті, навіть найкомпактніші сучасні генеративні моделі виявляються несумісними з цим середовищем без застосування глибокої оптимізації.

Сучасні підходи до стиснення нейронних мереж теоретично відкривають шлях до розгортання LLM у середовищах з обмеженими ресурсами. Водночас, як показує аналіз представлених на ринку рішень, жоден з наявних застосунків не реалізує розгортання власної мовної моделі безпосередньо у iOS-розширенні. Хмарні рішення не функціонують офлайн і не гарантують конфіденційності, а застосунки на основі Apple Intelligence доступні лише на найновіших флагманських пристроях. Це створює суттєвий розрив на ринку та підкреслює науково-практичну актуальність дослідження.

Об'єктом дослідження є процес розгортання компактних генеративних нейронних мереж у середовищі iOS-розширень з обмеженими ресурсами.

Предметом дослідження є методи оптимізації та стиснення трансформерних мовних моделей для їх виконання в умовах жорстких лімітів оперативної пам'яті, накладених механізмом Jetsam на процеси iOS-розширень.

Метою дослідження є розробити та експериментально перевірити підхід до оптимізації компактних генеративних нейронних мереж для їх стабільного функціонування в умовах ресурсних обмежень iOS-розширень, а також визначити фактичні межі застосовності методів стиснення на ANE для цього класу моделей.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1) Проаналізувати сучасні підходи до локального розгортання нейронних мереж та систематизувати специфічні обмеження iOS-розширень, що визначають вимоги до оптимізації.
2) Дослідити теоретичні основи трансформерної архітектури та методи її стиснення з оцінкою їх придатності для цієї роботи.
3) Провести порівняльний аналіз фреймворків Core ML та MLX з точки зору їхньої сумісності з жорсткими лімітами пам'яті.
4) Експериментально виміряти фактичні ліміти пам'яті для основних типів розширень, придатних для обробки тексту, та ключові метрики виконання моделей у різних конфігураціях.
5) Сформулювати практичні рекомендації щодо вибору інструментарію та обхідних архітектурних схем для розгортання локального AI у середовищі iOS-розширень.

Методи дослідження. Для розв'язання поставлених завдань використано комплекс методів: аналіз наукових джерел та технічної документації; порівняльний аналіз для зіставлення фреймворків Core ML та MLX, а також методів стиснення; методи конвертації нейронних мереж за допомогою інструментаріїв coremltools та mlx-lm; експерименти на фізичному пристрої з прямими вимірами використання оперативної пам'яті, статистична обробка

результатів для оцінювання продуктивності моделей у різних обчислювальних режимах.

Огляд джерел та методологічна основа. Методологічну основу роботи становлять праці у трьох напрямах: фундаментальні роботи з архітектури трансформерів та методів їх оптимізації, дослідження ANE та його програмного оточення nf офіційна документація Apple. Експериментальна частина використовує відкритий інструментарій john-rocky/CoreML-LLM для конвертації мовних моделей під ANE.

Структура роботи. Робота складається зі вступу, трьох розділів, висновків, переліку прийнятих скорочень та списку використаних джерел. Перший розділ присвячено аналізу предметної області та постановці задачі. Другий розділ містить теоретичні основи оптимізації трансформерних моделей для Apple Silicon. У третьому розділі викладено експериментальне дослідження.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Огляд сучасних підходів до розгортання нейронних мереж на мобільних пристроях

Традиційна парадигма використання великих мовних моделей (LLM) передбачає їх розгортання на потужних хмарних серверах, забезпечуючи доступ до моделей з сотнями мільярдів параметрів. Проте цей підхід має низку критичних недоліків для мобільних застосунків, зокрема залежність від стабільного інтернет-з'єднання, затримки передачі даних та ризики витоку конфіденційної інформації користувачів [1].

У відповідь на ці виклики в індустрії та науковій спільноті активно розвивається концепція On-device AI - виконання нейронних мереж безпосередньо на кінцевих пристроях, таких як ноутбуки чи смартфони. Перехід до локального розгортання зумовлений кількома ключовими перевагами [1].

По-перше, локальна обробка гарантує, що дані ніколи не залишають пристрій і не передаються на сторонні сервери. Особливо це критично для завдань, де передбачено роботу з текстом, коли користувач вводить особисті повідомлення, паролі або фінансову інформацію. Загроза витоку приватних даних під час використання хмарних рішень є однією з головних причин стрімкого зростання інтересу до локального розгортання моделей. Навіть за наявності зашифрованих каналів передачі хмарна модель обслуговування не може забезпечити тих самих гарантій конфіденційності, що й локальна обробка, оскільки сторонні сервери отримують доступ до контексту запитів користувача.

По-друге, хмарні запити залежать від пропускної здатності мережі, стану мережі та завантаженості серверів. Локальні моделі здатні генерувати текст миттєво, що є критично важливим для інтерактивних елементів інтерфейсу.

По-третє, Локальні моделі забезпечують безперебійну роботу функціоналу незалежно від якості покриття мережі. Це робить застосунок більш надійним та незалежним від стану серверної інфраструктури розробника.

Попри очевидні переваги, перенесення нейронної мережі на мобільні пристрої є складною інженерною задачею. Головна причина - невідповідність між обчислювальними вимогами сучасних LLM та апаратними можливостями мобільних платформ [1]. Навіть найкомпактніші практично корисні генеративні моделі потребують значного обсягу пам'яті. Наприклад, модель Qwen2.5-0.5B з 500 мільйонами параметрів у форматі FP16 займає близько 1 ГБ оперативної пам'яті лише для зберігання ваг, ще до початку обчислень. Більші моделі, такі як LLaMA 3.1 8B, потребують понад 16 ГБ у тому самому форматі. Також, очевидно, пропускна здатність пам'яті мобільних процесорів значно поступається серверним рішенням, що обмежує швидкість обробки. Окрім цього, інтенсивні обчислення під час роботи моделі призводять до швидкого розрядження акумулятора та перегріву пристрою.

Ці обмеження стають ще більш гострими в контексті iOS-розширень, де операційна система додатково накладає жорсткі ліміти на споживання пам'яті окремим процесом. Таким чином, для успішного розгортання генеративних моделей на мобільних пристроях необхідне застосування агресивних методів стиснення та оптимізації, що дозволяють знайти баланс між якістю генерації тексту, швидкістю роботи та споживанням ресурсів.

1.2 Особливості розширень в iOS та їх обмеження

1.2.1 Загальний огляд

Операційна система iOS відома своєю закритою файловою системою, яка ізолює застосунки один від одного задля безпеки. Однак, для забезпечення гнучкого користувацького досвіду, компанія Apple запровадила механізм App Extensions (розширення застосунків) - ізольованих модулів, що дозволяють розширювати функціональність системи поза межами основного застосунку. На відміну від повноцінного застосунку, розширення не є самостійною програмою - воно розповсюджується виключно у складі основного застосунку і не може існувати без нього. Водночас, під час виконання розширення працює як окремий процес із власним адресним простором, комунікація між ним та основним застосунком здійснюється виключно через міжпроцесну взаємодію, без прямого доступу до пам'яті одне одного [2].

Apple надає багато типів розширень, кожне з яких призначене для конкретного сценарію інтеграції з системою, наприклад, Keyboard Extension - замінює системну клавіатуру в будь-якому текстовому полі, Share Extension - дозволяє ділитися контентом між застосунками, Today Extension - відображає компактну інформацію в центрі сповіщень, Photo Editing Extension - інтегрується в застосунок «Фото» для редагування зображень, Notification Service Extension - обробляє пуш-сповіщення перед їх відображенням [2].

Ключова архітектурна особливість усіх розширень - принцип пісочниці. Розширення не має прямого доступу до пам'яті основного застосунку, не може виконувати довільні фонові задачі та взаємодіє з операційною системою лише через чітко визначені API. Комунікація між розширенням та основним застосунком можлива лише через спільний контейнер або стандартні системні механізми.

1.2.2 Апаратні обмеження розширень

Найбільш критичним обмеженням для розгортання нейронних мереж у розширеннях є жорсткі ліміти оперативної пам'яті, що встановлюються операційною системою [2]. На відміну від основного застосунку, розширення

отримує значно менший ліміт пам'яті, перевищення якого призводить до негайного завершення процесу механізмом Jetsam без можливості відновлення. Jetsam працює на рівні ядра системи iOS та функціонує незалежно від стандартних сигнальних механізмів, що унеможливлює перехоплення або обробку події завершення процесу [3].

Конкретні значення лімітів залежать від типу розширення та покоління пристрою. Наприклад, для Keyboard Extension Apple встановлює ліміт не більше 100 мегабайтів оперативної пам'яті. Для порівняння, основний застосунок на тому ж пристрої може використовувати гігабайти пам'яті.

Окрім обмежень пам'яті, розширення стикаються з рядом інших системних обмежень [2]. Перш за все, це відсутність фонового виконання. Розширення активне лише під час безпосередньої взаємодії користувача. Воно не може виконувати попереднє завантаження моделі у фоні, тобто модель має бути готова до роботи в межах часу, відведеного на запуск розширення.

Наступним обмеженням є час запуску розширення. Система очікує, що воно стане готовим до роботи протягом кількох секунд після виклику. Тривала ініціалізація, наприклад через повільне завантаження моделі, може призвести до збою.

Також важливо, що розширення мають обмежений пріоритет доступу до графічного процесору. Вони працюють із нижчим системним пріоритетом порівняно з основним застосунком, що на практиці обмежує ефективність використання GPU, особливо під час активної роботи застосунку на фоні. Натомість залишається доступ до Apple Neural Engine (ANE). Однак якщо модель містить непідтримувані операції або ANE зайнятий іншими системними задачами, обчислення можуть автоматично перемикатися на CPU та GPU.

1.3 Аналіз інструментарію для інтеграції нейронних мереж в iOS

Для розгортання та виконання нейронних мереж на пристроях екосистеми Apple розробникам доступні кілька різних підходів. Вибір фреймворку безпосередньо визначає, який апаратний прискорювач буде задіяний, як модель споживає оперативну пам'ять і, відповідно, чи взагалі є можливість вписатись у ліміти iOS-розширень.

1.3.1 Core ML

Core ML - це власний фреймворк машинного навчання від компанії Apple, інтегрований безпосередньо в iOS, iPadOS та macOS. Його ключова перевага полягає у глибокій системній інтеграції. Фреймворк автоматично визначає оптимальний обчислювальний вузол (CPU, GPU або ANE) для кожної операції моделі та керує розподілом навантаження без участі розробника. Розробник може вплинути на вибір вузла, що набуває особливої ваги в контексті розширень [4].

Моделі для Core ML зберігаються у форматі .mlpackage - статичному графі обчислень, що компілюється під конкретний пристрій при першому запуску. Для конвертації моделей, навчених у PyTorch чи TensorFlow, використовується відкрита бібліотека coremltools, яка також надає засоби стиснення, такі як квантизація, палетизація та прунінг.

З точки зору розгортання в розширеннях, Core ML у конфігурації cpuAndNeuralEngine має фундаментальну перевагу, адже підготовка і виконання моделі на ANE відбувається поза основним процесом розширення. Внаслідок цього суттєво знижується phys_footprint процесу - метрики, за якою Jetsam визначає ліміт [6]. Саме цей механізм робить Core ML основним кандидатом для розгортання моделей у середовищах з жорсткими лімітами.

Основний недолік фреймворку - статичність графу. Розмір контексту фіксується на етапі конвертації, що вимагає окремих версій моделі для різних

сценаріїв використання. Робота з динамічною довжиною послідовностей, що є основою генерації тексту, реалізована менш гнучко, а сам процес виконання є чорною скринькою для розробника, що ускладнює точкове управління оперативною пам'яттю.

1.3.2 MLX

Наприкінці 2023 року дослідницький підрозділ Apple представив MLX - відкритий фреймворк для машинного навчання на Apple Silicon [7]. На відміну від Core ML, MLX є динамічним, його граф обчислень будується під час виконання, що спрощує роботу з генеративними моделями, де довжина послідовності не відома наперед.

Принциповою архітектурною особливістю MLX є використання моделі об'єднаної пам'яті (UMA) Apple Silicon. Масиви даних знаходяться у спільній пам'яті без копіювання між CPU та GPU, що знижує накладні витрати під час інференсу. MLX має рідну підтримку архітектур трансформерів та сучасних методів квантизації, що робить його привабливим для локальних моделей. Також фреймворк має Swift API, що дозволяє легко інтегрувати логіку роботи з моделлю безпосередньо у вихідний код застосунку [7].

Проте для задачі розгортання в iOS-розширеннях MLX має критичне обмеження, бо на iOS фреймворк виконує обчислення виключно на GPU через Metal - ANE недоступний. Це означає, що всі ваги моделі повністю входять до phys_footprint процесу розширення, що унеможливлює вписання навіть агресивно квантизованих LLM у жорсткі Jetsam-ліміти. Порівняльний аналіз підтверджує, що MLX досягає найвищої швидкості обробки даних на Apple Silicon серед альтернатив, однак ця перевага реалізується лише в основному застосунку, а не в розширеннях [8].

1.3.3 Apple Foundation Models

На WWDC 2025 Apple представила Foundation Models framework - Swift API для доступу до системної мовної моделі (~3B параметрів), що лежить в основі Apple Intelligence [9]. Модель виконується на ANE, підтримує керовану генерацію тексту зі структурованими виходами та виклик програмних інструментів, і є безкоштовною для розробників. Фреймворк дозволяє переписувати текст, змінювати його тональність та виправляти помилки безпосередньо на пристрої, не передаючи дані на сторонні сервери.

Проте для цілей даного дослідження цей фреймворк має суттєві обмеження. Головним недоліком цього рішення є його вимогливість до апаратного забезпечення - він доступний лише на пристроях з підтримкою Apple Intelligence. Через це функціонал фреймворку обмежений лише найновішими флагманськими смартфонами, починаючи з iPhone 15 Pro та лінійки iPhone 16, що виключає значну частину активних пристроїв [10]. Крім того, Foundation Models не вирішує задачу розгортання довільної оптимізованої моделі в розширенні - він надає доступ лише до фіксованої системної моделі через обмежений API. Таким чином, цей фреймворк є корисним доповненням, але не замінює потребу в дослідженні методів оптимізації власних моделей.

1.4 Огляд існуючих рішень для обробки тексту

Для обґрунтування необхідності розробки власного оптимізованого рішення було проведено аналіз існуючих продуктів, що надають функціонал інтелектуальної обробки тексту в розширеннях застосунків iOS. За архітектурним принципом виконання моделі їх можна поділити на хмарні рішення та системні рішення Apple.

Найпоширенішими представниками категорії хмарних рішень є Wispr Flow, Grammarly Keyboard та Microsoft SwiftKey. Наприклад, Wispr Flow функціонує як клавіатурне розширення з підтримкою голосового введення та автоматичного редагування тексту. Однак, згідно з власною документацією компанії,

транскрипція та обробка тексту завжди відбуваються на хмарних серверах - без підключення до мережі застосунок повністю втрачає функціональність [11]. Grammarly та SwiftKey пропонують корекцію граматики та автодоповнення, також покладаючись на серверну інфраструктуру для розширених AI-функцій. Для функціонування цих клавіатур цих застосунків операційна система вимагає від користувача увімкнення налаштування повного доступу, що дозволяє розширенню підключатися до інтернету. Обробка на віддалених серверах унеможливлює роботу клавіатури в офлайн-режимі, додає мережеві затримки і створює серйозні ризики для конфіденційності, оскільки приватний контент користувача регулярно залишає межі пристрою. Експериментальне тестування підтвердило, що без підключення до мережі зазначені клавіатури виявляються неспроможними виконувати свої завдання.

Ще одну категорію формують застосунки, що покладаються на Apple Intelligence як на базову платформу. Представником цієї категорії є Say Better та Type Buddy, які надають клавіатурне розширення, що позиціонує себе як офлайн AI-рішення. Однак аналіз показав, що вони не містять власної моделі, а лише є інтерфейсом, що лише викликає системний функціонал Apple Intelligence через стандартне API. Це принципово відрізняє його від рішень з власними вбудованими моделями, адже при відсутності підтримки Apple Intelligence на пристрої застосунок стає непрацездатним. Це було підтверджено відповідним системним повідомленням при запуску на непідтримуваному пристрої. Таким чином, ця категорія рішень успадковує ті самі апаратні обмеження Apple Intelligence, тобто функціонал доступний тільки на пристроях старших за iPhone 15 Pro, що суттєво обмежує охоплення аудиторії порівняно з іншими аналогами.

Таким чином, проведений аналіз виявляє суттєвий функціональний розрив. Жодне з розглянутих рішень не реалізує розгортання власної мовної моделі безпосередньо всередині iOS-розширення. Хмарні рішення працюють на широкому спектрі смартфонів, але жертвують абсолютною конфіденційністю та

не здатні функціонувати офлайн. Системні рішення Apple, при всіх своїх перевагах, просто недоступні для значної частини користувачів екосистеми Apple. Саме це визначає науково-практичну актуальність даного дослідження.

1.5 Висновок до розділу та постановка задачі

Проведений аналіз предметної області виявив фундаментальну суперечність у сфері мобільної розробки інтелектуальних інтерфейсів. Існує потреба у приватних, незалежних від мережі інструментах обробки тексту, що найбільш природно реалізуються у вигляді iOS-розширень, зокрема Custom Keyboard Extension. Проте архітектура iOS накладає на розширення жорсткі апаратні обмеження: ліміт оперативної пам'яті, що контролюється системним механізмом Jetsam, та обмежений пріоритет доступу до GPU. Оскільки виконання моделі на GPU повністю входить до показника phys_footprint процесу розширення, не лише запуск квантизованих LLM перевищить встановлені ліміти, але й використання значно менших архітектур буде проблематичним. Аналіз існуючих рішень не виявив застосунків, що надають подібний функціонал із власними локальними моделями безпосередньо у розширенні.

Головною метою даної дипломної роботи є дослідження, програмна реалізація та експериментальна перевірка методів оптимізації компактних генеративних нейронних мереж для їх стабільного функціонування в умовах жорстких ресурсних обмежень iOS-розширень.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Провести глибокий аналіз сучасних методів та алгоритмів стиснення нейронних мереж, зокрема квантизації, палетизації та прунінгу, та оцінити їхній вплив на розмір моделі, споживання оперативної пам'яті та якість генерації.

2. Спроєктувати та розробити прототип iOS-застосунку з власним розширенням.
3. Провести порівняльний аналіз фреймворків Core ML та MLX з метою визначення придатного для розгортання моделі в умовах жорстких лімітів розширень.
4. Адаптувати та оптимізувати компактну мовну модель для виконання на Apple Neural Engine через Core ML, застосувавши методи квантизації та палетизації.
5. Провести експериментальні виміри ключових метрик: пікового споживання оперативної пам'яті, часу завантаження моделі, часу до першого токена та швидкості генерації.
6. На основі отриманих результатів сформулювати практичні висновки та рекомендації щодо вибору інструментарію та методів оптимізації для розгортання локального AI в розширеннях застосунків iOS.

РОЗДІЛ 2. ТЕОРЕТИЧНІ ОСНОВИ ОПТИМІЗАЦІЇ ТРАНСФОРМЕРНИХ МОДЕЛЕЙ ДЛЯ APPLE SILICON

2.1 Архітектура трансформерних мовних моделей

Усі сучасні великі мовні моделі побудовані на архітектурі трансформера, представленої у роботі «Attention Is All You Need» у 2017 році [12]. Розуміння внутрішньої будови цієї архітектури є необхідною передумовою для коректного застосування методів оптимізації, оскільки саме особливості трансформера визначають, які компоненти моделі споживають найбільше пам'яті та як їх можна стискати без критичної втрати якості.

2.1.1 Структура декодерного трансформера

Оригінальна архітектура трансформера складалася з енкодера та декодера, призначених для задач машинного перекладу [12]. Однак для задачі генерації тексту виявилось достатнім використання лише декодерної частини. Саме такий декодерний варіант лежить в основі переважної більшості сучасних генеративних моделей.

Декодерний трансформер являє собою послідовність ідентичних за структурою блоків, кожен з яких містить два основні компоненти, а саме механізм багатоголової самоуваги та мережу прямого поширення. Між цими компонентами застосовуються нормалізація шару та залишкові з'єднання. Для моделі Qwen2.5-0.5B, що розглядається у практичній частині, кількість таких блоків становить 24, а розмір прихованого стану 896.

2.1.2 Механізм самоуваги

Самоувага є ключовим компонентом, що дозволяє моделі враховувати контекстуальні зв'язки між токенами. Для кожного токена обчислюються три вектори: запит, ключ та значення, що утворюються лінійними перетвореннями

вхідного представлення. Увага обчислюється як:

$$Attention(Q,\ K,\ V)\ = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

де: Q - матриця запитів,

K - матриця ключів,

V - матриця значень,

T - операція транспонування,

d_k - розмірність векторів ключів.

На практиці використовується багатоголова увага, коли вектори Q, K, V розділяються на декілька груп, що обчислюють увагу паралельно. Для задачі стиснення моделей особливе значення має варіант розподілу голів між ними. У класичній схемі Multi-Head Attention (MHA) кількість голів запитів дорівнює кількості голів ключів та значень. Більшість сучасних моделей натомість використовують Grouped-Query Attention (GQA), при якій кілька голів запитів спільно використовують одну пару ключ-значення [13]. Наприклад, у моделі Qwen2.5-0.5B налічується 14 голів запитів, але лише 2 пари ключ-значення. Це важливо для розгортання на пристроях з обмеженою пам'яттю, оскільки напряму впливає на розмір кешу, який буде розглянуто далі.

2.1.3 Фази генерації: попереднє заповнення та декодування

Генерація тексту трансформером логічно поділяється на фази попереднього заповнення та декодування. Під час фази попереднього заповнення модель обробляє увесь вхідний промпт одним проходом, обчислюючи приховані стани та видає перший згенерований токен. Під час фази декодування модель генерує наступні токени один за одним, а кожен новий токен передається на вхід моделі разом із попереднім контекстом для отримання наступного [14].

Різниця цих фаз полягає в обчислювальних характеристиках. Попереднє заповнення є обчислювально-обмеженим, оскільки виконує паралельні обчислення над усім промптом одразу, тоді як декодування є пам'ять-обмеженим, тобто кожен крок обробляє лише один токен, але потребує доступу до всього накопиченого контексту. Це безпосередньо впливає на час до появи першого токена та швидкість генерації токенів за секунду.

2.1.4 KV-кеш та його роль у споживанні пам'яті

Пряма реалізація генерації мала б на кожному кроці повторно обчислювати вектори K та V для всього попереднього контексту, що є обчислювально нераціональним. Натомість використовується механізм KV-кешу, коли вже обчислені ключі та значення для попередніх токенів зберігаються у пам'яті та повторно використовуються на наступних кроках декодування [14].

Розмір KV-кешу безпосередньо визначає верхню межу пам'яті, необхідної для роботи моделі, і обчислюється за формулою:

$$M_{KV} = 2 \cdot L \cdot B \cdot H_{KV} \cdot N \cdot d_h \cdot b,$$

де: L - кількість шарів,

B - розмір батча,

H_{KV} - кількість голів ключ-значення,

N - довжина контексту,

d_h - розмірність кожної голови,

b - кількість байтів на елемент.

Для моделі, наприклад, Qwen2.5-0.5B у форматі FP16 з контекстом 512 токенів розрахунок дає 6,3 МБ. Для контексту 1024 токени відповідне значення зростає до 12,6 МБ. Хоча у масштабах серверного інференсу такі обсяги є

несуттєвими, в умовах лімітів iOS-розширень розмір KV-кешу стає значним параметром оптимізації.

2.2 Методи стиснення нейронних мереж

Стиснення нейронної мережі - це процес зменшення її розміру та обчислювальної складності при збереженні прийнятної якості. У контексті розгортання на пристроях з обмеженими ресурсами стиснення є не опціональним кроком, а необхідною умовою функціонування моделі. Згідно з документацією Core ML Tools, існує три основних підходи до стиснення нейронних мереж: лінійна квантизація, палетизація та прунінг [5]. Усі три методи спрямовані на зменшення обсягу пам'яті, необхідного для зберігання ваг моделі, однак досягають цього принципово різними шляхами і дають різний ефект залежно від цільового апаратного забезпечення.

2.2.1 Лінійна квантизація

Квантизація - це процес перетворення параметрів моделі високої точності (зазвичай 32-бітних чи 16-бітних чисел з плаваючою комою) у представлення нижчої розрядності, найчастіше 8-бітні або 4-бітні цілі числа [15]. Основна ідея методу полягає у тому, що повна точність FP32 значно перевищує реальні потреби моделі під час інференсу. Розподіл ваг у навченій нейронній мережі зазвичай зосереджений у вузькому діапазоні значень, що дозволяє представити його з використанням обмеженого набору дискретних значень при збереженні цілісності моделі

Найпоширенішою є лінійна квантизація, що використовує афінне перетворення для відображення діапазону FP-значень у цілочисельний діапазон [15]:

$$Q(r) = round\left(\frac{r}{S}\right) - Z,$$

де: S - масштабний коефіцієнт,

Z - точка нуля.

У симетричній квантизації Z = 0, що спрощує обчислення на апаратному рівні, але може давати менш точне представлення для асиметричних розподілів. Асиметрична квантизація використовує ненульовий zero-point і краще підходить для асиметричних активацій.

Важливим параметром є гранулярність - рівень, на якому обчислюються параметри квантизації S та Z. У потензорному режимі для всього тензора використовується лише одна їх пара; у поканальному - окрема пара для кожного каналу вихідного шару; у поблочному - пара для кожного блоку фіксованого розміру. Зменшення гранулярності покращує точність, але збільшує накладні витрати на зберігання параметрів квантизації [15].

За способом застосування розрізняють два підходи [16]. Пост-тренувальна квантизація застосовується до вже навченої моделі без подальшого навчання та потребує лише невеликого калібрувального набору даних для визначення оптимальних масштабів. Це найшвидший спосіб, що зазвичай займає кілька хвилин, однак при низькій бітності точність моделі може суттєво деградувати. Навчання з урахуванням квантування імітує квантизацію під час навчання за допомогою механізму псевдоквантування, що дозволяє моделі адаптувати свої ваги до низькоточного представлення. Цей підхід забезпечує значно кращу точність при меншій кількості бітів, але вимагає доступу до тренувального набору даних і значно більших обчислювальних ресурсів.

2.2.2 Палетизація

Палетизація (або також відома як кластеризація вагів) - це нелінійний метод стиснення, що групує ваги моделі у фіксовану кількість кластерів за допомогою алгоритму k-середніх. Замість збереження кожної ваги окремо модель зберігає спільну таблицю пошуку з 2^N значеннями (центроїдами), а кожна вага представляється як індекс до відповідного центроїда. Цей підхід був вперше запропонований у роботі Han et al. (2015) «Deep Compression», де показано стиснення моделі AlexNet у 35 разів без втрати точності [17].

Принципова відмінність палетизації від лінійної квантизації полягає у нерівномірності представлення значень. Лінійна квантизація розподіляє рівні рівномірно у діапазоні, тоді як палетизація розміщує центроїди адаптивно, де концентрується більшість ваг. Це дозволяє досягти кращої точності при однаковій кількості біт, особливо коли розподіл ваг далекий від рівномірного.

Подібно до квантизації, палетизація має кілька рівнів гранулярності. У потензорному режимі вся вагова матриця ділить одну таблицю пошуку, що дає максимальне стиснення, але може призводити до значної помилки для великих матриць. У режимі для групи каналів, група каналів використовує спільну таблицю, що дає компроміс між стисненням і точністю [5]. Загальний принцип такий: чим більше центроїдів на канал, тим менше стиснення, але менша помилка.

Серед алгоритмів реалізації палетизації виділяють три основних. Автономна кластеризація - найпростіший підхід, що використовує лише ваги моделі без жодних даних. Чутлива кластеризація - алгоритм на основі калібрувальних даних. Він визначає ступінь впливу кожної ваги на кінцеву помилку моделі завдяки аналізу матриці Гессе, і приділяє більше уваги в палітрі критично важливим вагам. Навчальна кластеризація - алгоритм під час донавчання мережі. Його використання дозволяє градієнтам проходити крізь таблицю пошуку та поступово підлаштовувати центроїди палітри так, щоб вони ідеально відповідали потребам моделі [5].

Ключова перевага палетизації у цій роботі зафіксована безпосередньо у документації Core ML Tools, де зазначено, що використання палетизації є найбільш ефективним способом стиснення нейронної мережі на ANE для оптимізації оперативної пам'яті та мінімізації затримки виконання [5]. Саме ця властивість робить палетизацію основним кандидатом для стиснення моделі для використання в iOS-розширенні.

2.2.3 Прунінг

Прунінг - це метод стиснення, що полягає у видаленні ваг, які мають найменший вплив на якість моделі. Ваги, близькі до нуля, замінюються точними нулями, після чого модель може бути представлена у розрідженому форматі, де зберігаються лише ненульові значення з їхніми позиціями [5].

Існує два види прунінгу. Неструктурний прунінг обнуляє окремі ваги незалежно одна від одної на основі їхньої абсолютної величини або іншого критерію важливості. Цей підхід може досягати високих рівнів розрідженості (60-90%) з мінімальною втратою точності, однак для отримання реального виграшу у швидкості потребує апаратної підтримки розрідженого обчислення. Структурний прунінг видаляє цілі структурні одиниці моделі, наприклад, голови уваги, шари або канали, і дає виграш на будь-якому обладнанні, але супроводжується значно більшою деградацією точності і потребує подальшого донавчання.

Але застосування класичного прунінгу до великих мовних моделей виявляється не таким простим. Дослідження Frantar та Alistarh (2023) показало, що такий підхід катастрофічно деградує якість LLM навіть при помірних рівнях розрідженості, причому більші моделі деградують швидше за менші. Для подолання цієї проблеми було запропоновано метод SparseGPT. Це алгоритм одноразового прунінгу, що розглядає стиснення як пошарове розв'язання розрідженої регресії, та дозволяє стиснути GPT-моделі до 50-60% розрідженості без донавчання з мінімальною втратою якості [18].

У Core ML прунінг реалізовано через формат розріджених тензорів. Прунінг моделей може давати виграш у швидкості [5]. Однак, реальний виграш у пам'яті процесу від неструктурного прунінгу залежить від того, чи здатне середовище зберігати розріджений тензор без матеріалізації нулів. Проте часто така підтримка обмежена - обнулені ваги фактично продовжують займати ту саму кількість пам'яті, що й до прунінгу.

З огляду на це, прунінг у даній роботі розглядається переважно у теоретичному аспекті. Він є потенційно цінним методом для подальших досліджень, особливо у поєднанні з іншими методами, але для основного експерименту перевага надана палетизації як методу з доведеною ефективністю.

2.3 Архітектурні оптимізації під Apple Neural Engine

Методи стиснення, розглянуті у попередньому підрозділі, дозволяють зменшити розмір моделі, однак для досягнення максимального ефекту від виконання на Apple Neural Engine необхідні додаткові архітектурні модифікації самої моделі. Ці модифікації пов'язані з суттєвими відмінностями ANE від CPU чи GPU з його специфічними апаратними обмеженнями щодо форми тензорів, типів операцій та послідовності обчислень.

2.3.1 Базова архітектура Apple Neural Engine

Apple Neural Engine - це спеціалізований обчислювальний прискорювач, інтегрований у чіпі Apple починаючи з A11 Bionic (iPhone 8/X, 2017 рік). Архітектурно ANE є масивом обчислювальних елементів, оптимізованим під операції згортки та матричного множення - двох класів обчислень, що домінують у нейронних мережах [6]. На відміну від GPU, доступного через Metal, ANE не має публічного API. Тобто прямий доступ до апаратних інтерфейсів закритий, а взаємодія з ним можлива виключно через високорівневі фреймворки Apple, насамперед Core ML.

ANE позиціонується як обчислювача з мінімальним впливом на ресурси застосунку. Він спроектований для виконання нейромережевих навантажень з мінімізацією використання пам'яті та споживання енергії [6].

Через закритість архітектури публічної інформації про апаратні характеристики та програмні обмеження ANE дуже мало. Найбільш повну картину дають три взаємодоповнюючі джерела: систематизована документація спільноти, упорядкована у репозиторії Голлеманса [19], академічна робота Orion (2026), що каталогізує 20 програмних обмежень ANE, виявлених через дослідження приватних API [20], а також офіційна публікація Apple Machine Learning Research 2022 року «Deploying Transformers on the Apple Neural Engine»[6].

2.3.2 Принципи оптимізації трансформерів під ANE

У публікації 2022 року дослідницька група Apple представила реалізацію трансформерної архітектури, оптимізовану під ANE. Застосування описаних у ній принципів до моделі DistilBERT забезпечило прискорення інференсу до 10 разів та зменшення пікового споживання пам'яті процесом у 14 разів порівняно з звичайною конвертацією [6]. Цей результат досягнуто за рахунок кількох взаємопов'язаних модифікацій базової реалізації трансформера.

Першочерговою зміною став перехід від традиційних тривимірних тензорів у форматі (S, B, C) або (B, S, C), де B - розмір батча, S - довжина послідовності, C - розмірність прихованого стану, до чотиривимірного формату (B, C, 1, S). Така вимога зумовлена тим, що остання вісь буфера ANE не пакується щільно, а потребує безперервності та вирівнювання до 64-байтової межі. Якщо на останній позиції розмістити вісь з одиничним розміром, буфер штучно розшириться до 64 байтів. Для FP16 це збільшує споживання пам'яті у 32 рази, а для 8-бітного у 64 рази, що призводить до витіснення даних із L2-кешу в повільну оперативну пам'ять. Розміщення довжини послідовності (S) як останньої забезпечує природне вирівнювання для реалістичних значень

довжини контексту. Щоб адаптувати архітектуру до нового формату, усі стандартні лінійні шари замінюються на математично еквівалентні згорткові шари з ядром 1 на 1. Це зберігає логіку моделі, дозволяючи компілятору ANE, який найкраще оптимізований якраз під згорткові операції, задіяти найефективніший шлях.

Іншим важливим принципом є оптимізація проміжних перестановок. Стандартне обчислення зваженого скалярного добутку в механізмі уваги передбачає операції зміни форми та транспонування тензорів між матричним множенням запитів і ключів та подальшим множенням на тензор значень. В архітектурі Neural Engine кожна така зміна потенційно спричиняє копіювання даних у пам'яті, оскільки принаймні одна вісь буферів залишається непакованою. У реалізації Apple ці перестановки замінюються на спеціально підібрану схему тензорного множення за правилом індексного підсумовування, що безпосередньо відповідає апаратному формату даних і не потребує створення проміжних копій [6].

Також важливо не забувати про розбиття великих проміжних тензорів. Для оптимізації обчислень у механізмі уваги проміжні тензори запитів, ключів та значень не обробляються монолітно, а розділяються на окремі функції для кожної голови уваги. Використання менших тензорів підвищує ймовірність їхнього розміщення у кеш-пам'яті другого рівня та дозволяє краще розпаралелювати операції між ядрами Neural Engine на етапі компіляції. Ця модифікація є особливо важливою для конфігурацій, продуктивність яких обмежена пропускною здатністю пам'яті, а не інтенсивністю самих обчислень.

Окремою практичною проблемою при адаптації сучасних мовних моделей є невідповідність нормалізаційних шарів. Більшість сучасних архітектур, включно з Qwen2.5-0.5B, використовують RMSNorm замість LayerNorm для обчислювальної ефективності. Однак ANE має апаратно-прискорений шлях саме для LayerNorm, а RMSNorm у його наборі операцій відсутній [22]. На

практиці використовується трюк еквівалентного перетворення, коли вхідний вектор конкатенується зі своєю інверсією, і застосування LayerNorm до отриманого симетричного розподілу математично співпадає з RMSNorm на оригінальному [22]. Таким чином, RMSNorm-шари моделі при адаптації під ANE замінюються на еквівалентну композицію операцій, доступних компілятору.

Описані оптимізації пов'язані з методами стиснення. Apple зазначає, що навіть після всіх архітектурних модифікацій багато трансформерних конфігурацій залишаються обмеженими пропускною здатністю пам'яті, тобто час виконання визначається переважно швидкістю завантаження ваг з оперативної пам'яті, а не самими обчисленнями над ними [6]. Зменшення розміру ваг за допомогою квантизації чи палетизації прямо покращує цей режим, скорочуючи обсяг даних, що вибираються з пам'яті на кожному кроці інференсу.

2.3.3 Виконання моделі поза процесом застосунку

Властивість ANE, що має визначальне значення для розгортання моделей у розширеннях iOS, полягає у тому, що його робочий цикл відбувається фізично поза процесом застосунку. Хоча Apple не публікує деталей цієї архітектури, дослідження останніх років [20] дозволяють описати її наступним чином. Компіляція моделі для ANE виконується системним сервісом, що запускається як окремий процес поза адресним простором застосунку. Сама ж диспетчеризація запитів на інференс делегується привілейованому системному демону aned (Apple Neural Engine daemon), що взаємодіє з апаратним інтерфейсом через ядро операційної системи. Обмін даними між процесом застосунку та aned реалізовано через спільні буфери, які відображаються в обидва адресні простори без копіювання.

Внаслідок цієї архітектури ваги моделі, завантаженої для виконання на ANE, фізично зберігаються у пам'яті, керованій системним демоном, а не у

адресному просторі застосунку. Це означає, що моделі, виконувані у режимі cpuAndNeuralEngine, не вносять у метрику процесу phys_footprint, за якою Jetsam визначає перевищення ліміту пам'яті, повний обсяг своїх ваг. Натомість у режимах cpuAndGPU чи cpuOnly параметри моделі представлені безпосередньо у просторі застосунку і повністю враховуються Jetsam. Саме цей механізм пояснює, чому моделі, які у GPU-конфігурації спричиняли б миттєве завершення процесу, у конфігурації з ANE можуть стабільно вписатися у бюджет iOS-розширення. Експериментальне підтвердження та кількісна оцінка цього ефекту є одним із ключових результатів практичної частини даної роботи.

2.3.4 Обмеження ANE та їх практичні наслідки

Попри суттєві переваги, виконання моделі на ANE супроводжується низкою обмежень, що впливають на проєктування та конвертацію моделі ще на етапі підготовки.

Одним із найважливіших чинників є обмежений набір підтримуваних операцій. Компілятор ANE приймає лише певну підмножину внутрішнього представлення MIL (Model Intermediate Language). У разі виявлення непідтримуваної операції обчислювальний граф розбивається на сегменти, що призводить до перенесення відповідних частин на CPU або GPU [20]. Така фрагментація не лише додає затримок через перемикання між обчислювачами, а й повертає виконання у адресний простір застосунку, нехтуючи виграшем від використання спеціалізованого прискорювача.

Додаткові труднощі виникають при роботі з дуже великими шарами, характерними для сучасних мовних моделей. Наприклад, фінальний проєкційний шар на словник у LLM, розмір якого може сягати десятків тисяч, часто відхиляються компілятором і виконуються на CPU [20].

Крім того, ANE підтримує тензори не вище 5-вимірних. Операції, що оперують 6 і більше вимірними структурами, потребують декомпозиції на набір

5-вимірних операцій. Це обмеження найкритичніше для архітектур типу window attention у візуальних трансформерах [21].

Процес оптимізації ускладнюється відсутністю публічного API для прямого моніторингу стану ANE під час роботи. Розробник може лише непрямо оцінювати ефективність розподілу за допомогою Xcode Performance Reports та Instruments, що показують частку операцій, фактично виконаних на ANE для кожного шару моделі.

Сукупність цих обмежень робить розгортання довільної трансформерної моделі на ANE досить складною задачею. Для досягнення стабільної роботи моделі потрібна цілеспрямована адаптація як архітектури мережі, так і конвертаційного процесу: підбір сумісних операцій, перерозподіл тензорів у правильний формат і виокремлення непідтримуваних частин у CPU ще до того, як модель буде передана у на конвертацію.

2.4 Моделі зі станом та оптимізація KV-кешу в Core ML

Як було показано у підрозділі 2.1, KV-кеш є критичним компонентом ефективного інференсу трансформерних моделей. Без нього кожен новий токен потребує повторного обчислення ключів та значень для всього попереднього контексту. Проте спосіб реалізації цього кешу у конкретному фреймворку значно впливає як на швидкість виконання, так і на споживання пам'яті. До iOS 18 Core ML не мав вбудованої підтримки внутрішнього стану моделі, що змушувало розробників передавати весь KV-кеш як вхід і вихід моделі на кожному кроці генерації з відповідними накладними витратами на копіювання значних обсягів пам'яті між викликами. У 2024 Apple представила механізм моделей зі станом разом із декількома іншими оптимізаціями, що принципово змінили підхід до реалізації KV-кешу і стали необхідною умовою ефективної генерації тексту трансформерами на Apple Silicon [5, 23].

2.4.1 Концепція моделей зі станом

Модель зі станом у Core ML - це модель, що зберігає внутрішній стан між послідовними викликами. На відміну від традиційних моделей без стану, де всі вхідні дані передаються явно при кожному виклику, модель зі станом оголошує спеціальні буфери стану, що залишаються у пам'яті, керованій фреймворком, між викликами і модифікуються самою моделлю безпосередньо. Підтримка цього механізму у Core ML була введена починаючи з iOS 18 для моделей формату mlprogram [5].

На практиці робота зі станом проходить через три рівні. У PyTorch-моделі тензори, які мають стати станом, реєструються як внутрішні буфери. При конвертації через coremltools ці структури отримують статус змінних стану, що визначає логіку їхньої подальшої обробки середовищем виконання Core ML. Уже на пристрої, при виконанні моделі, програма отримує доступ до стану, який передається до обчислювального графу разом із вхідним вектором даних на кожному кроці генерації. Цей підхід забезпечує автоматичне збереження та оновлення контексту обчислень безпосередньо у пам'яті прискорювача, що дозволяє уникнути зайвого копіювання даних [5].

2.4.2 Реалізація KV-кешу через шаблон зрізового оновлення

У документації по конвертації Llama 3.1 та Mistral 7B Apple представила реалізацію KV-кешу як моделі зі станом - шаблон зрізового оновлення [23, 24]. Замість звичайного зчеплення нових ключів-значень з попередніми використовується пряме записування у потрібний зріз попередньо виділеного буфера.

Компілятор Core ML розпізнає цей шаблон і будує внутрішнє представлення моделі, у якому оновлення кешу виконується безпосередньо у виділеній ділянці пам'яті [5]. Звичайне зчеплення тут не підійшло, бо воно

вимагало б виділення нової пам'яті на кожному кроці, і давало б змінні розміри виходів, несумісні зі статичним Core ML-графом.

Виграш цього підходу Apple продемонструвала на прикладі Llama-3.1-8B [23]. Базова модель без KV-кешу дає 0,19 токенів в секунду, реалізація кешу - 1,2, а той самий кеш зі станом уже 16,26. Тобто перехід до оновлення буфера на місці дає приріст більш ніж у 30 разів. Для Qwen2.5-0.5B числа безумовно відрізнятимуться, але сама залежність робочих швидкостей від механізму стану повинна зберігатися.

Розмір буфера фіксується ще на етапі конвертації моделі. Він виділяється одноразово при ініціалізації моделі. Вона прибирає фрагментацію пам'яті, але задає фіксовану нижню межу її споживання, тому розмір контексту варто підбирати ретельно.

2.4.3 Гнучкі розмірності входу та злита операція уваги

KV-кеш зі станом обслуговує режим попереднього заповнення та декодування. Щоб не створювати дві окремі моделі, використовується механізм гнучких розмірностей входу, який дозволяє оголосити, що довжина послідовності може варіюватися у заданому діапазоні [5, 23]. При виклику моделі Core ML підбирає відповідний шлях виконання під фактичну довжину поточної послідовності, і одна й та сама модель так обробляє і весь промпт за раз, і окремі токени по одному, працюючи зі спільним KV-кешем.

Починаючи з iOS 18 Core ML також підтримує обчислення уваги як єдину злиту операцію [23]. Раніше увага розбивалась на окремі множення матриць, функцію softmax і маскування, і кожен крок створював свій проміжний тензор. В такому випадку для довгих контекстів матриця уваги могла досягати сотень мегабайт. Злите ядро уникає створення цієї проміжної матриці, скорочуючи і використання пам'яті, і затримку.

2.4.4 Багатофункціональні моделі

У тому ж 2024 було представлено механізм багатофункціональних моделей, що надає можливість тримати в одному файлі моделі кілька функцій зі спільними вагами, але різними графами обчислень [24]. Основним його призначенням є пакування LoRA-адаптерів: одна базова модель та кілька наборів адаптерів для різних задач, де кожен набір представлено окремою функцією.

У задачі розгортання LLM на ANE цей механізм застосовується інакше, компілюючи окремо стадії попереднього заповнення та декодування. Перша функція компілюється під обробку фіксованої кількості токенів за прохід, а друга під обробку одного токена за раз. Ваги завантажуються в пам'ять один раз, а кожна функція отримує план виконання, оптимізований саме під свій режим.

2.4.5 Практичні обмеження

Проте реалізація моделей зі станом має кілька обмежень. По-перше, розмір контексту фіксований ще на етапі конвертації. Щоб підтримати кілька сценаріїв використання, потрібно або вибрати достатньо великий розмір з відповідним споживанням пам'яті, або готувати окремі моделі для різних режимів, що збільшує розмір самого застосунку.

Також залишається відкритим питання, де саме перебувають буфери стану при виконанні моделі на ANE, і як їхній обсяг впливає на phys_footprint процесу. Спільні буфери відображаються в обидва адресні простори, але документація Apple не уточнює, яка частина буфера потрапляє у метрику процесу.

2.5 Висновки до розділу

У розділі систематизовано теоретичну базу, на якій будується подальша практична частина роботи, а саме основні джерела споживання пам'яті трансформерною моделлю, методи її стиснення, архітектурні особливості Apple Neural Engine та механізм моделей зі станом у Core ML.

Аналіз архітектури трансформера показав, що головними джерелами споживання пам'яті при інференсі є ваги моделі та KV-кеш, причому останній зростає лінійно з довжиною контексту, кількістю шарів та розмірністю головок уваги. Механізм Grouped-Query Attention, реалізований у сучасних моделях, дозволяє суттєво скоротити розмір кешу порівняно з класичним Multi-Head Attention. Асиметрія між стадіями попереднього заповнення та декодування визначає характер навантаження на пам'ять і обчислювальні ресурси, що безпосередньо впливає на вибір стратегії оптимізації.

Огляд методів стиснення виявив три основні підходи, які включають лінійну квантизацію, палетизацію та прунінг. З них палетизація у форматі 4-біт з груповою по-канальною гранулярністю є найдоцільнішим методом для розгортання на ANE. Прунінг визнано неоптимальним вибором для основного експерименту через обмежену підтримку розрідженого виконання трансформерних моделей на ANE.

Аналіз архітектурних оптимізацій під ANE показав, що максимальний виграш досягається не лише через стиснення ваг, а й через адаптацію самої структури моделі. Для цього варто використовувати 4-вимірний формат тензорів, замінити лінійні шари на згортки з ядром 1×1, перетворити RMSNorm на еквівалентну форму через LayerNorm та застосувати злиту реалізацію операцію уваги. Ключовим для даної роботи є те, що виконання моделі на ANE відбувається поза адресним простором застосунку в окремому системному процесі, внаслідок чого ваги моделі не входять до показника phys_footprint процесу, за яким Jetsam визначає перевищення ліміту пам'яті.

Розгляд механізму моделей зі станом у Core ML показав, що для ефективної генерації на Apple Silicon необхідна попереднє виділення буферу KV-кешу як окремої змінної стану із застосуванням шаблону зрізового оновлення, що дозволяє модифікувати кеш безпосередньо у виділеній пам'яті. Це усуває накладні витрати на копіювання пам'яті між кроками генерації, але вимагає фіксації розміру контексту ще на етапі конвертації моделі.

РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Методологія дослідження

Достовірність експериментальних результатів безпосередньо залежить від чіткого визначення методики, тому перш ніж переходити до самих експериментів, доцільно зафіксувати тестове середовище, набір метрик та принципи проведення вимірів.

3.1.1 Тестове середовище

Усі експериментальні виміри проводилися на фізичному пристрої iPhone 13 Pro з операційною системою iOS 26.4. Цей пристрій оснащений процесором Apple A15 Bionic, що містить 16-ядерний Apple Neural Engine продуктивністю до 15,8 трильйонів операцій за секунду, та 6 ГБ оперативної пам'яті. Вибір саме цієї моделі обумовлений тим, що вона не є флагманською на момент проведення дослідження, що дозволяє продемонструвати застосовність запропонованих рішень на широкому спектрі активних пристроїв, а не лише на найновіших.

Окремої уваги потребує обґрунтування того, чому всі виміри проводилися виняткова на фізичному пристрої, а не на симуляторі iOS. Симулятор виконує застосунок безпосередньо на апаратному забезпеченні хост-комп'ютера, через що всі обчислення відбуваються на ресурсах Mac, а не на цільовому iPhone. Це критично для обчислень, що залежать від ANE, адже хоч Mac і має власний ANE, його архітектура та продуктивність не відповідають ANE цільового пристрою. Те ж саме стосується і підтримки Metal. Крім того, як було емпірично підтверджено у попередніх запусках, симулятор не накладає тих обмежень пам'яті на розширення, що механізм Jetsam застосовує на фізичному пристрої. Внаслідок цього виміри, отримані в симуляторі, не дозволяють коректно дослідити саме той клас обмежень, що складає центральну проблему даної роботи [25].

3.1.2 Метрики дослідження

Для оцінювання різних варіантів моделі та конфігурацій виконання використовується набір метрик, що відображають як ресурсні витрати, так і користувацькі характеристики системи.

Піковий phys_footprint процесу - максимальне значення фізичної пам'яті, спожитої процесом за час експерименту. Саме за цією метрикою Jetsam визначає перевищення ліміту, тому вона є основною у даній роботі.

Час до першого токена - час від подачі промпта на вхід моделі до отримання першого згенерованого токена. Ця метрика безпосередньо впливає на сприйняття затримки користувачем.

Швидкість генерації - кількість токенів, що генеруються моделлю за секунду після завершення фази попереднього заповнення.

Час завантаження моделі - тривалість ініціалізації моделі від виклику завантаження до готовності до інференсу.

Розмір файлу моделі на диску. Хоча ця характеристика напряму не впливає на використання пам'яті, вона визначає загальний розмір застосунку.

3.1.3 Інструменти вимірювання

Для отримання значення phys_footprint у коді застосунку використовується системний виклик task_info з прапором TASK_VM_INFO, що повертає структуру task_vm_info_data_t. Поле phys_footprint цієї структури відповідає метриці Jetsam та вимірюється в байтах. Реалізована функція замірювання інкапсулює цей виклик і повертає значення у мегабайтах для зручності аналізу.

Для контрольної перевірки використовувався Xcode Instruments, зокрема профіль Allocations, що дозволяє візуалізувати динаміку споживання пам'яті у часі та виявляти підозрілі піки. Виміри, отримані через task_vm_info всередині

застосунку, систематично узгоджувалися з показаннями Instruments для виявлення можливих артефактів.

3.1.4 Особливості організації експериментів

Структура експериментів у цій роботі побудована з урахуванням специфіки досліджуваного середовища. Жорсткі ліміти пам'яті роблять проведення прямого порівняння варіантів моделі безпосередньо в розширенні неможливим. В такому випадку варіанти зі значним споживанням пам'яті будуть автоматично завершені ще на стадії завантаження, що не дозволить отримати порівняльні метрики.

З огляду на це, основне порівняльне дослідження різних варіантів моделі, методів стиснення та обчислювальних режимів проводиться в основному застосунку, де процес не підлягає жорстким лімітам пам'яті, але працює на тому ж апаратному забезпеченні, з тією ж операційною системою та тими ж версіями фреймворків. Це забезпечує об'єктивність порівняння. Експерименти в розширенні слугують меті перевірки, чи здатна обрана конфігурація моделі стабільно функціонувати в реальних умовах виконання у розширенні.

Що цікаво, у попередніх тестових запусках було встановлено, що при роботі застосунку з під'єднаним дебагером ANE стає фактично недоступним для процесу. Незалежно від вибраного обчислювального вузла, Core ML просто виконує модель на CPU або GPU. Внаслідок цього виміри метрик у такій конфігурації не відображають характеристик моделі при її роботі на ANE. Тож, усі експериментальні запуски, результати яких використовуються у подальших підрозділах, проводилися на реліз-збірках, встановлених на пристрій та запущених з домашнього екрана пристрою після від'єднання від ноутбука.

3.1.5 Протокол повторюваності та набір тестових промптів

Для забезпечення коректності порівняння кожна конфігурація тестується на фіксованому наборі промптів. Як цільовий сценарій застосунку було обрано

перефразування неформальних повідомлень користувача у формальний стиль. Як системний промпт було використано текст: «You are a text rewriting assistant. Rewrite the user's text in a formal, professional style. Output ONLY the rewritten text, no explanations, no preamble, no commentary».

Набір тестових промптів користувача складено з типових прикладів неформального тексту англійською мовою, з характерними скороченнями та розмовними зворотами:

1) «hey, can u send me that file asap? kinda need it rn»
2) «so basically the thing is we messed up the deadline and idk what to do»
3) «tbh the meeting was pretty useless, we didn't figure anything out»
4) «gonna be late today, smth came up, sorry»
5) «the new feature is like really buggy and users are super annoyed about it»

Кожну конфігурацію моделі прогналися на цих п'яти промптах послідовно, по 3 повторні запуски на кожен, із фіксацією окремих метрик для кожного. Це дає змогу як отримати усереднені значення, так і виявити можливі залежності характеристик моделі від конкретного типу вхідних даних.

Перед кожною серією експериментів пристрій приводився до контрольованого стану. Рівень заряду акумулятора не нижче 50%, тепловий стан нормальний, фонові застосунки примусово завершені. Для уникнення наслідків холодного старту, перед кожним заміром виконувався один пробний прогін, результати якого до підсумків не включалися.

3.2 Дослідження лімітів пам'яті iOS-розширень

Як було зазначено у підрозділі 1.2.2, ліміти оперативної пам'яті розширень не задокументовані офіційно та змінюються з версіями iOS, типом розширення та поколінням пристрою. Більше того, представник служби технічної підтримки Apple Developer Technical Support прямо вказує, що жодне числове значення

ліміту не має сприйматися як гарантоване, а єдиним надійним способом переконатися у можливості роботи розширення в межах системних обмежень є тестування на цільовому пристрої з конкретною версією iOS [26]. З огляду на це, фактичні значення лімітів для типів розширень, придатних для задачі обробки тексту, визначалися експериментально.

3.2.1 Методика експериментального виміру

Принцип експериментального визначення лімітів ґрунтується на властивості механізму Jetsam при завершенні процесу залишати у системному журналі детальний звіт про подію перевищення ліміту. Згідно з документацією Apple, цей звіт містить, серед іншого, точне значення ліміту оперативної пам'яті, застосованого до процесу на момент його завершення, а також фактичне значення phys_footprint, при якому відбулося спрацювання [3]. Таким чином, iOS самостійно повідомляє точну межу, без потреби у визначенні її значення всередині процесу.

Для емпіричного визначення лімітів був розроблений мінімальний тестовий зразок розширення, що після запуску циклічно виділяє блоки пам'яті, заповнюючи їх сміттям, до моменту примусового завершення процесу. Після спрацювання Jetsam з пристрою зчитується відповідний звіт із точним значенням ліміту, перевищеним процесом.

3.2.2 Типи розширень, придатних для використання LLM

iOS пропонує понад десяток різних точок розширення, кожна з яких призначена для конкретного сценарію взаємодії з користувачем або системою. Проте не всі з них становлять практичний інтерес з точки зору розгортання великих мовних моделей. Для цього завдання не підходять розширення, що працюють у фоні без візуального інтерфейсу та з жорсткими обмеженнями на час виконання, не дозволяють здійснювати тривалу обробку і не надають користувачеві природного способу її ініціювати. Розширення, що обробляють

відмінний від тексту вміст як аудіо, чи відео, архітектурно не передбачають взаємодії з мовною моделлю. Також виключаються розширення з малими ресурсними бюджетами, у яких ліміти пам'яті аж надзвичайно малі.

Таким чином, до розгляду включаються три типи розширень, у яких розгортання мовної моделі є осмисленим. Вони надають візуальний інтерфейс, активно ініціюються користувачем та мають теоретично достатній ресурсний бюджет для можливого запуску моделі.

Custom Keyboard Extension інтегрується безпосередньо у системну клавіатуру і дозволяє виконувати обробку тексту, що набирається користувачем, у будь-якому застосунку. Share Extension викликається з системного меню «Поділитися» при виділенні тексту і дозволяє виконати над ним довільну обробку. Action Extension викликається з того самого меню, проте призначений саме для перетворення вмісту з поверненням результату назад до основного застосунку.

Також було розглянуто Photo Editing Extension. І хоча він орієнтований на редагування зображень і не пов'язаний з обробкою тексту, проте серед усіх типів розширень має найбільший, з дуже значним відривом, ліміт пам'яті. Завдяки цьому запасу пам'яті, на ньому можна буде тестувати, як моделі поводяться саме у розширеннях після успішних тестів у звичайному застосунку.

3.2.3 Виміряні значення лімітів пам'яті

Експериментальні виміри проводилися для кожного з чотирьох типів розширень за методикою, описаною у підрозділі 3.2.1. Отримані результати наведено у Таблиці 3.1.

Таблиця 3.1 - Ліміти пам'яті iOS-розширень на iPhone 13 Pro з iOS 26.4

Тип розширення	Виміряний ліміт, МБ
Custom Keyboard Extension	77

Share Extension	120
Action Extension	300
Photo Editing Extension	760

3.2.4 Вибір цільового типу розширення для прототипу

Серед чотирьох розглянутих типів розширень Custom Keyboard Extension становить найбільший інтерес з точки зору користувацького досвіду. Клавіатура присутня у будь-якому полі введення тексту в будь-якому застосунку, що дозволяє використовувати функціонал роботи з моделлю у момент написання повідомлення без потреби копіювати текст, перемикатися між застосунками або викликати додаткові меню. На відміну від цього, використання Share або Action Extension передбачає виділення тексту і явне звернення до системного меню, що робить їх менш зручними.

Проте орієнтація на клавіатурне розширення має іншу перевагу. З цих чотирьох типів розширень саме клавіатура має найменший ліміт пам'яті. Розробка рішення, що вписується в її обмеження, гарантує можливість розгортання у інших типах розширень.

З огляду на ці міркування, основним цільовим типом розширення для прототипу обрано Custom Keyboard Extension. У разі, якщо за результатами інтеграції виявиться, що отриманий варіант моделі не вписується у ліміт 77 мегабайтів, цільовий тип розширення буде змінено на Share з лімітом 120 МБ або Action Extension з 300 мегабайтів.

3.3 Попередній експеримент: класифікація тональності на основі RoBERTa

Перш ніж переходити до основних експериментів з оптимізації генеративної моделі, було проведено попередній експеримент з конвертації та стиснення меншої моделі іншого класу - RoBERTa-large для задачі класифікації емоційного забарвлення тексту. Метою цього експерименту була перевірка, які методи стиснення моделі дозволяють зменшити використання пам'яті, чи можливе взагалі використання нейронних мереж у розширеннях, та перевірка гіпотези про те, що виконання моделі на ANE колосально змінює споживання пам'яті процесом порівняно з виконанням на CPU або GPU. Менший масштаб і простіша архітектура цієї моделі дозволили швидко отримати кількісне підтвердження ефекту, на якому далі будувалися всі основні експерименти.

3.3.1 Постановка експерименту та обрана модель

Як цільову модель використано j-hartmann/emotion-english-roberta-large, яка донавчена на задачі семикласової класифікації емоційного забарвлення англомовного тексту. Модель має близько 355 мільйонів параметрів та архітектуру стандартного трансформера типу BERT з 24 шарами та прихованою розмірністю 1024.

Як основні метрики для оцінювання результатів обрано розмір файлу скомпільованої моделі на диску, піковий phys_footprint процесу під час виконання інференсу, час завантаження моделі, час одного інференсу та точність класифікації на тестовому наборі прикладів. Тестовий набір складається з п'яти різних фраз для кожного класу емоцій.

3.3.2 Конвертація та методи стиснення

Конвертація моделі з PyTorch у Core ML здійснювалася за стандартним сценарієм. Над оригінальною моделлю будувалася тонка обгортка, що повертає нормалізовані ймовірності класів замість сирих значень. Далі модель аналізувалася на тестовому прикладі, що дозволило зафіксувати її обчислювальний граф у проміжному поданні, придатному для подальшої

трансформації. Цей граф конвертувався у формат Core ML mlprogram з фіксованою формою входу довжиною 128 для ідентифікаторів токенів та маски уваги.

Над отриманою базовою моделлю у форматі FP16 застосовувалися такі методи стиснення:

1) Лінійна симетрична квантизація до INT8
2) Палетизація на 6 біт методом k-середніх
3) Палетизація на 4 біт методом k-середніх
4) Прунінг 50%

3.3.3 Результати експерименту

Базова модель FP16 тестувалася у всіх чотирьох доступних обчислювальних режимах, стиснені варіанти тільки у двох ключових: all та cpuAndNeuralEngine. Результати наведено у Таблицях 3.2, 3.3 та 3.4.

Таблиця 3.2 - Базова модель FP16 у різних обчислювальних режимах

MLComputeUnits	Пікова RAM, МБ	Завантаження, мс	Інференс, мс	Точність, %
cpuOnly	37	52	56	97
cpuAndGPU	887	178	76	97
cpuAndNeuralEngine	37	65	21	97
all	136	80	23	97

Таблиця 3.3 - Стиснені варіанти моделі у режимі all

Конфігурація	Розмір файлу, МБ	Пікова RAM, МБ	Завантаження, мс	Інференс, мс	Точність, %

FP16 (базова)	711	136	80	23	97
INT8	356	2038	90	114	97
Palette-6	267	140	88	13	97
Palette-4	178	141	78	12	97
Prune-50%	747	140	620	23	97

Таблиця 3.4 - Стиснені варіанти моделі у режимі cpuAndNeuralEngine

Конфігурація	Розмір файлу, МБ	Пікова RAM, МБ	Завантаження, мс	Інференс, мс	Точність, %
FP16 (базова)	711	37	65	21	97
INT8	356	39	76	13	97
Palette-6	267	36	72	12	97
Palette-4	178	36	64	12	97
Prune-50%	747	36	377	21	97

Окремо слід зазначити, що при поєднання режиму cpuAndGPU з INT8-квантизованою під час запуску Core ML примусово завершує процес з помилкою у Metal. Це свідчить, що Core ML не може виконати операції з лінійно квантизованими 8-бітними вагами для цієї архітектури моделі, що, напевно, також і стало причиною аномального скачка RAM до 2000+ МБ у конфігурації all.

3.3.4 Інтеграція моделі у розширення клавіатури

Для практичної верифікації отриманих характеристик у цільовому середовищі обрана конфігурація з палетизацією 4 біти у режимі cpuAndNeuralEngine. Модель була інтегрована безпосередньо у Custom

Keyboard Extension розробленого застосунку. Файл моделі вбудовано у бандл розширення без використання App Groups чи комунікації з основним застосунком. Завантаження моделі та виконання інференсу винесені в окремий фоновий потік, що дозволяє не блокувати обробку введення тексту користувачем.

Виміри phys_footprint, проведені безпосередньо з контексту розширення, відтворюють близько 36 МБ після завантаження моделі та незмінне значення під час виконання інференсу. Час інференсу та час завантаження також не відрізняються від виміряних в основному застосунку, що підтверджує ідентичність виконавчого тракту ANE для обох типів процесів у випадку моделі-класифікатора. Сумарне споживання пам'яті розширенням з урахуванням власних службових структур клавіатури утримується у межах ліміту 77 МБ, що забезпечує стабільну роботу розширення впродовж тривалого сеансу.

3.3.5 Аналіз результатів та висновки експерименту

Отримані результати дають кількісне підтвердження центральної тези теоретичної частини про вплив обчислювального вузла на phys_footprint процесу. У режимі cpuAndGPU базова модель FP16 спричиняє споживання 887 МБ оперативної пам'яті, тоді як та сама модель у режимі cpuAndNeuralEngine лише 37 МБ. Різниця у понад 20 разів узгоджується з описаним у підрозділі 2.3.3 механізмом виконання моделей на ANE поза адресним простором застосунку. Точність моделі залишається незмінною у всіх режимах, що підтверджує відсутність впливу вибору обчислювального вузла на якість виходу.

Аномальне значення phys_footprint для INT8-моделі у режимі all (2038 МБ) є прямим наслідком виявленої несумісності квантизованих ваг із GPU. Оскільки у режимі all частина обчислень розподіляється на GPU, Core ML змушений деквантизувати ваги назад у формат FP та матеріалізувати відповідні буфери у пам'яті процесу. Звідси походить значення, що на порядок перевищує усі інші виміряні конфігурації. У режимі cpuAndNeuralEngine, де модель повністю

виконується на ANE, ця проблема не проявляється і споживання пам'яті залишається на рівні базової моделі.

У режимі cpuAndNeuralEngine усі методи стиснення демонструють стабільну поведінку. Палетизація на 4 біти забезпечує найбільший виграш у розмірі файлу з 711 до 178 МБ, тобто майже у 4 рази без втрати точності. Цей результат збігається з рекомендацією документації Core ML Tools, що визначає палетизацію найефективнішим методом стиснення для виконання на ANE [5], та обґрунтовує вибір саме цього методу для подальшої роботи з генеративною моделлю.

Прунінг 50% не дав очікуваного виграшу ні у розмірі файлу, ні у споживанні пам'яті процесу. Розмір файлу склав 747 МБ, що навіть перевищує базову модель. Цей результат прямо підтверджує, що обнулені ваги продовжують зберігатися у щільному форматі і фактично займають ту саму кількість пам'яті, що й до прунінгу. Також помітно суттєво більший час завантаження прунінгованої моделі у 6-8 разів. На основі цих спостережень прунінг далі не розглядається як основний метод стиснення.

Таким чином, експеримент підтвердив, що виконання моделі на ANE забезпечує значно менше споживання пам'яті процесом порівняно з CPU та GPU. Серед розглянутих методів стиснення палетизація на 4 біти визначена як найбільш перспективна, тоді як прунінг визнаний недоцільним. Також експеримент довів, що нейронні мережі можуть працювати у середовищі розширень застосунків, при цьому виконуючись на ANE.

3.4 Вибір та обґрунтування цільової моделі

3.4.1 Критерії відбору

Виходячи з обмежень, встановлених у попередніх розділах роботи, цільова модель повинна задовольняти кілька вимог.

По-перше, кількість параметрів моделі має бути такою, щоб після стиснення палетизацією на 4 біти результуючі ваги вписувалися у прийнятний розмір застосунку, а KV-кеш помірного контексту. Це фактично обмежує розгляд моделями з кількістю параметрів менше одного мільярда.

По-друге, модель має бути побудована на стандартному декодерному трансформері без екзотичних операцій, що не підтримуються компілятором ANE. Бажано, щоб модель використовувала Grouped-Query Attention для скорочення розміру KV-кешу, а нестандартні нормалізації допускали еквівалентну заміну на доступні ANE примітиви.

По-третє, ліцензія моделі повинна дозволяти її вбудовування у комерційний застосунок без додаткових обмежень щодо кількості користувачів, обов'язкової атрибуції чи заборон на похідні роботи.

По-четверте, модель має бути попередньо налаштованою на виконання інструкцій, оскільки цільова задача перефразування неформального тексту у формальний потребує надійного слідування системному промпту.

3.4.2 Огляд кандидатів

Серед компактних мовних моделей з відкритими вагами, доступних на момент проведення дослідження, основними кандидатами є наступні.

Llama 3.2 1B Instruct - 1 мільярд параметрів, інструкційно налаштована, з якісним слідуванням системному промпту. Поширюється під ліцензією Llama 3.2 Community License [27], що містить вимогу обов'язкової атрибуції моделі у відповідному файлі повідомлення, обмеження щодо досягнення 700 мільйонів активних користувачів на місяць та заборону на використання її виходів для навчання моделей, що не є похідними від Llama.

SmolLM2-360M-Instruct - 360 мільйонів параметрів під ліцензією Apache 2.0. Менший розмір не дає принципової переваги в межах виявленого бюджету пам'яті, при цьому моделі цього розмірного класу типово демонструють нижчу якість слідування інструкціям, що дуже критично.

Qwen2-0.5B-Instruct - 500 мільйонів параметрів. Найменша з сімейства мовних моделей Qwen2.5 [28].

TinyLlama 1.1B Chat - 1,1 мільярда параметрів під ліцензією Apache 2.0, проте побудована без Grouped-Query Attention, що пропорційно збільшує розмір KV-кешу та зменшує запас пам'яті для роботи в розширенні.

3.4.3 Обрана модель - Qwen2.5-0.5B-Instruct

З огляду на встановлені критерії, для основного експерименту обрано Qwen2.5-0.5B-Instruct, інструкційно налаштовану модель з сімейства Qwen2.5, опубліковану компанією Alibaba у вересні 2024 року під ліцензією Apache 2.0 [28, 29].

Модель містить пів мільярда параметрів. Вона є декодерним трансформером з 24 шарами та прихованою розмірністю 896, що використовує позиційне кодування RoPE, активацію SwiGLU, нормалізацію RMSNorm та спільні вбудовування [29]. Останнє є важливим у контексті оптимізації пам'яті, оскільки матриці вхідних та вихідних вбудовувань спільно використовують однакові ваги, скорочуючи загальний обсяг параметрів моделі.

Принципово важливим для цілей дослідження є використання Grouped-Query Attention зі співвідношенням 14 голів запитів на 2 пари ключ-значення. Це дозволяє підтримувати помірно довгий контекст у межах жорстких лімітів пам'яті розширення.

Модель є інструкційно налаштованою, надійно слідує системним промптам та покриває тренувальний словник з 29 мов. Ліцензія Apache 2.0 не накладає

обмежень на комерційне використання, кількість користувачів чи розповсюдження похідних робіт.

3.5 Експерименти з MLX

Перш ніж переходити до основного експерименту з Core ML, було проведено окреме дослідження поведінки обраної моделі у фреймворку MLX. Цей експеримент має на меті емпірично перевірити теоретичне твердження, сформульоване у підрозділі 1.3.2, про те, що виконання моделі через MLX на iOS принципово несумісне з обмеженнями iOS-розширень через відсутність доступу до ANE та повне утримання ваг моделі у адресному просторі процесу. Окрім того, отримані дані забезпечують точку відліку для подальшого порівняння з результатами Core ML.

3.5.1 Конвертація та методи стиснення

Модель було конвертовано з оригінального формату HuggingFace у внутрішній формат MLX за допомогою інструментарію mlx-lm. MLX підтримує переважно лінійну квантизацію з груповими параметрами, при якій кожній групі ваг фіксованого розміру призначається власна пара значень масштабу та зміщення [7]. Розмір групи у проведених експериментах дорівнює 64, що є значенням за замовчуванням для MLX.

Над базовою моделлю у форматі FP16 застосовано лінійну квантизацію до INT8, до INT4 та до INT2. Усі варіанти, включно з базовою FP16-моделлю, виконуються на GPU через Metal, оскільки це єдиний обчислювальний вузол, доступний MLX на iOS [7, 8].

3.5.2 Результати експерименту

Результати вимірів наведено у Таблиці 3.5.

Таблиця 3.5 - Результати виконання Qwen2.5-0.5B-Instruct у фреймворку MLX

Конфігурація	Розмір файлу, МБ	Пікова RAM, МБ	Завантаження, мс	Перший токен, мс	Інференс, мс	Токени/с
FP16 (базова)	999	1236	1998	90	910	313
INT8	536	719	1952	119	494	402
INT4	290	484	1797	118	487	463
INT2	166	333	1720	123	1208	1516

3.5.3 Аналіз результатів та висновки експерименту

Отримані виміри однозначно підтверджують центральне теоретичне твердження підрозділу 1.3.2 щодо MLX. Пікове споживання пам'яті процесом лінійно слідує за розміром файлу моделі на диску, різниця між цими двома величинами стабільно тримається у межах 167-237 МБ і відповідає обсягу, необхідному для KV-кешу, проміжних тензорів та службових структур фреймворку. Жодних ознак виконання моделі поза адресним простором застосунку не спостерігається.

З погляду придатності для розгортання у розширеннях, жоден із чотирьох варіантів не вписується у виміряні ліміти. Навіть найбільш агресивна квантизація INT2, якість якої абсолютно ніяка, і яка була протестована лише з дослідницькою метою, з піковим споживанням 333 МБ значно перевищує ліміти розширень. Формально лише Photo Editing Extension був би сумісний.

Сукупно проведений експеримент дав два ключові висновки. По-перше, емпірично підтверджено, що MLX на iOS виконує всі обчислення на GPU без використання ANE, що приводить до повного входження ваг моделі у

phys_footprint процесу. По-друге, навіть найбільш агресивна квантизація, доступна у MLX, не дозволяє вписати модель у ліміти основних типів iOS-розширень, придатних для обробки тексту.

3.6 Експеримент з Core ML

Отримані у підрозділі 3.5 результати показали, що фреймворк MLX, попри високу швидкість генерації, несумісний з обмеженнями iOS-розширень через виконання усіх обчислень на GPU з повним утриманням ваг у адресному просторі процесу. Наступним кроком стало проведення аналогічного дослідження тієї ж моделі у Core ML, фреймворку, який, як було підтверджено, здатен виконувати моделі на ANE з винесенням ваг поза адресний простір процесу. У цьому підрозділі описано конвертацію моделі, її виміри у головному застосунку та подальшу спробу інтеграції у розширення.

3.6.1 Інструмент конвертації моделі

Як уже було вияснено, онвертація сучасної мовної моделі у формат Core ML з повноцінним використанням ANE є складним інженерним завданням, що передбачає застосування набору взаємопов'язаних оптимізацій (підрозділ 2.3.2). Перелічені оптимізації на момент проведення дослідження вже мають готову відкриту реалізацію у вигляді інструментарію john-rocky/CoreML-LLM [30], що поширюється за ліцензією MIT та містить кожну з них. Самостійна імплементація тих самих технічних рішень окрім того повторювала б наявні результати без додання нової наукової цінності, тоді як основний мета даного розділу провести кількісний аналіз поведінки оптимізованої моделі у різних типах процесів iOS. З огляду на це, у якості вихідної точки для основного експерименту використано згаданий інструментарій. Репозиторій надає попередньо конвертовану версію Qwen2.5-0.5B-Instruct у форматі INT4-палетизації, яку і обрано як цільову для подальших вимірів.

3.6.2 Виміри у головному застосунку

Експеримент проведено відповідно до загальної методики, описаної у підрозділі 3.1. Модель завантажено з обчислювальним режимом cpuAndNeuralEngine та інтегровано у тестовий головний застосунок без використання ізольованих процесів. Виміри phys_footprint проведено кожні 10 мілісекунд протягом інференсу у пошуках піку використання пам'яті.

Результати показують стабільне утримання phys_footprint у межах близько 74 МБ. Це дуже відрізняється від поведінки тієї ж моделі у MLX-варіанті INT4, де пікове споживання пам'яті процесом склало 484 МБ. Спостережувана різниця у понад 10 разів відповідає попередньому теоретичному прогнозу та результату експерименту над класифікатором RoBERTa. Також це додатково підтверджує здатність ANE утримувати ваги моделі поза адресним простором застосунку і для генеративних декодерних моделей. Метрики швидкості, отримані у головному застосунку, у порівнянні з відповідними значеннями MLX-варіанта INT4 наведено у Таблиці 3.6.

Таблиця 3.6 - Порівняння виконання Qwen2.5-0.5B INT4 у Core ML (ANE) та MLX (GPU) у головному застосунку

Метрика	Core ML (ANE)	MLX (GPU)
Розмір файлу, МБ	316	290
Пікова RAM, МБ	74	484
Завантаження, мс	109	1797
Перший токен, мс	546	118
Інференс, мс	690	487
Токени/с	23	463

3.6.3 Спроба інтеграції у розширення

З огляду на успішне виконання моделі у головному застосунку зі споживанням пам'яті, що теоретично дозволяє інтеграцію навіть у Custom Keyboard Extension з його лімітом 77 МБ, проведено серію спроб запуску тієї ж моделі у процесах-розширеннях. Послідовно протестовано Custom Keyboard Extension, Action Extension та Photo Editing Extension з лімітами 77, 300 та 760 МБ відповідно. У жодному з випадків розширення не завершило завантаження моделі, а процес примусово закінчувався системою впродовж перших секунд після ініціалізації Core ML.

Логування phys_footprint дозволило відтворити динаміку споживання пам'яті у момент термінації. На відміну від поведінки у головному застосунку, де phys_footprint залишається стабільним близько 74 МБ, у розширенні спостерігається швидкий ріст споживання пам'яті, швидко досягаючи 300 МБ і перевищуючи ліміт Action Extension. Спроби запуску у Photo Editing Extension з лімітом 760 МБ, що значно перевищує зафіксоване споживання пам'яті у Action Extension, також не дали позитивного результату, процес завершується системою на тому ж етапі ініціалізації Core ML.

3.6.4 Аналіз результатів та висновки експерименту

Зіставлення результатів вимірів у головному застосунку та у розширенні дозволяє сформулювати ключовий висновок експерименту. Та сама модель з тією самою конфігурацією виконання в одному типі процесу демонструє поведінку, повністю відповідну теоретичному передбаченню, стабільно утримаючи phys_footprint на низькому рівні з виконанням обчислень поза адресним простором застосунку. У процесі-розширенні ж спостерігається протилежна поведінка, ваги моделі завантажуються в адресний простір процесу-розширення, phys_footprint швидко досягає порядку розміру файлу моделі, і процес завершується системою.

Найбільш обґрунтоване пояснення спостережуваної розбіжності полягає у відмінностях налаштувань пісочниці для процесів головного застосунку та

процесів-розширень iOS. Доступ до ANE на рівні операційної системи реалізується через XPC-комунікацію з системним демоном, права на яку для процесів-розширень обмежені більш суворо, ніж для головних застосунків. У ситуації, коли запит на виконання моделі на ANE з обчислювальним режимом cpuAndNeuralEngine не може бути задоволений, Core ML здійснює перехід на виконання моделі на CPU, що передбачає завантаження ваг у пам'ять процесу-споживача. Така поведінка не задокументована Apple для платформи iOS однак узгоджується з усією сукупністю отриманих емпіричних даних.

Слід окремо зазначити, що у підрозділі 3.3 для класифікатора RoBERTa у Custom Keyboard Extension аналогічної проблеми не спостерігалося, а phys_footprint у розширенні відповідав значенням з головного застосунку. Поведінка ANE у процесах-розширеннях, отже, не є однорідно недоступною, а залежить від класу моделі, імовірно, від типу архітектури (енкодер чи декодер), наявності стану KV-кешу або абсолютного розміру моделі. Точне з'ясування меж цього обмеження становить напрямок подальших досліджень.

З практичної точки зору, отримані результати означають, що поточна архітектура iOS не дозволяє розгортання генеративних мовних моделей у процесах-розширеннях з повноцінним використанням оптимізації пам'яті через ANE. Альтернативними напрямками подолання цього обмеження можуть слугувати винесення інференсу мовної моделі у головний застосунок з обміном даними з розширенням через XPC або спільний контейнер App Group, розробка спеціалізованих моделей, побудованих переважно з енкодерних шарів і призначених для класифікаційних або вибірково-шаблонних задач замість повноцінної генерації, або очікування подальших змін у програмному стеку iOS, що нададуть розширенням повноцінний доступ до ANE.

3.7 Висновки до розділу

У третьому розділі проведено комплексне експериментальне дослідження поведінки моделей машинного навчання в обмеженнях iOS-розширень. Емпірично виміряно ліміти phys_footprint для чотирьох типів розширень, придатних для обробки текстового вмісту, з виявленим діапазоном доступних ресурсів від 77 МБ для Custom Keyboard Extension до 760 МБ для Photo Editing Extension. На моделі-класифікаторі RoBERTa у попередньому експерименті підтверджено центральну гіпотезу теоретичної частини, що виконання моделі на ANE забезпечує утримання її ваг поза адресним простором процесу-споживача. Серед розглянутих методів стиснення палетизація на 4 біти виявилася оптимальним вибором, поєднуючи майже чотирикратне скорочення розміру файлу без сильної втрати точності та найменшим піковим споживанням пам'яті, що уможливило успішну інтеграцію класифікатора у Custom Keyboard Extension.

Для основної моделі Qwen2.5-0.5B-Instruct проведено порівняння двох фреймворків виконання: MLX та Core ML. Експерименти з MLX показали, що цей фреймворк, попри високу швидкість генерації, повністю утримує ваги моделі у пам'яті процесу і несумісний з лімітами навіть найбільш ресурсних розширень. Виконання тієї ж моделі через Core ML у головному застосунку з обчислювальним режимом cpuAndNeuralEngine підтвердило теоретичне передбачення - phys_footprint утримується на рівні близько 74 МБ при розмірі моделі на диску у 316 МБ. Однак спроби розгортання цієї ж моделі у процесах-розширеннях усіх протестованих типів виявили, що ваги генеративної моделі завантажуються в адресний простір розширення. Це свідчить про фактичну недоступність ANE для цього класу моделей у контексті процесів-розширень iOS.

Отриманий результат становить основний емпіричний внесок даної роботи. Було показано, що поточна архітектура iOS не дозволяє повноцінне розгортання сучасних генеративних мовних моделей у процесах-розширеннях з використанням оптимізації пам'яті через ANE, попри те, що теоретичні

передумови такого розгортання виконуються та підтверджуються для моделей-класифікаторів. Реалізація локального інференсу LLM у розширеннях iOS на поточний момент залишається неможливою без архітектурних змін з боку платформи або переходу до альтернативних схем взаємодії, таких як винесення інференсу у головний застосунок з обміном даними через App Group або XPC.

ВИСНОВКИ

У результаті виконаної дипломної роботи досліджено та експериментально перевірено методи оптимізації компактних генеративних нейронних мереж для їх розгортання в умовах жорстких ресурсних обмежень iOS-розширень. Поставлена мета досягнута, а сформульовані у вступі завдання виконані.

Проаналізовано сучасні підходи до локального розгортання нейронних мереж та систематизовано особливості iOS-розширень. Встановлено, що ключовим обмежувальним фактором є механізм Jetsam, який примусово завершує процеси при перевищенні лімітів phys_footprint, причому конкретні значення лімітів не задокументовані офіційно і залежать від типу розширення та покоління пристрою. Огляд існуючих продуктів виявив, що жодне з представлених рішень не реалізує локальне розгортання власної мовної моделі безпосередньо у розширенні застосунку.

Систематизовано теоретичну базу оптимізації трансформерних моделей. Показано, що основними джерелами споживання пам'яті при інференсі є ваги моделі та KV-кеш, причому останній лінійно зростає з довжиною контексту та кількістю шарів. Серед методів стиснення лінійна квантизація, палетизація та прунінг дають принципово різний ефект залежно від цільового апаратного забезпечення. Палетизація на 4 біти з груповою поканальною гранулярністю визначена як найбільш доцільний метод для виконання на ANE. Прунінг визнано неоптимальним вибором через обмежену апаратну підтримку розрідженого виконання у трансформерних моделях. Окремо обґрунтовано необхідність архітектурних модифікацій трансформера під ANE: переходу до 4-вимірного формату тензорів, заміни лінійних шарів на згортки 1×1, перетворення RMSNorm у еквівалентну форму через LayerNorm та використання шаблону зрізового оновлення KV-кешу як моделі зі станом.

Проведено порівняльний аналіз фреймворків Core ML та MLX. Експериментально підтверджено, що MLX на iOS виконує всі обчислення виключно на GPU через Metal, що призводить до повного входження ваг моделі до phys_footprint процесу. Для моделі Qwen2.5-0.5B-Instruct у форматі INT4 пікове споживання пам'яті в MLX становить 484 МБ, що несумісно з лімітами всіх основних типів розширень. Натомість Core ML у конфігурації cpuAndNeuralEngine утримує те ж саме навантаження на рівні близько 74 МБ за рахунок виконання обчислень поза адресним простором застосунку.

Експериментально виміряно фактичні ліміти пам'яті для чотирьох типів розширень на iPhone 13 Pro з iOS 26.4: 77 МБ для Custom Keyboard Extension, 120 МБ для Share Extension, 300 МБ для Action Extension та 760 МБ для Photo Editing Extension. На моделі-класифікаторі RoBERTa-large підтверджено центральну гіпотезу про вплив обчислювального вузла на phys_footprint процесу: різниця між режимами cpuAndGPU (887 МБ) та cpuAndNeuralEngine (37 МБ) становить понад 20 разів, що уможливило успішну інтеграцію класифікатора у клавіатурне розширення з лімітом 77 МБ.

Сформульовано ключовий емпіричний результат роботи. Виявлено, що для генеративної моделі Qwen2.5-0.5B-Instruct у форматі INT4-палетизації виконання у головному застосунку відповідає теоретичному передбаченню, проте у процесах-розширеннях усіх протестованих типів модель не завершує завантаження. Ваги завантажуються в адресний простір процесу-розширення, phys_footprint швидко досягає порядку розміру файлу моделі, і процес примусово завершується системою. Проте через закритість архітектури ANE, важко однозначно сказати яка тому причина, це питання вимагає подальших досліджень.

Науковий внесок роботи полягає в емпіричному виявленні та кількісній характеризації обмеження, яке не задокументоване Apple і раніше не описане в академічній літературі, а саме фактичної недоступності повноцінної

ANE-оптимізації пам'яті для сучасних генеративних мовних моделей у розширеннях застосунків iOS, попри її доступність для класифікаційних моделей у тому ж середовищі. Окремим невеликим внеском є експериментально виміряні значення лімітів phys_footprint для чотирьох типів iOS-розширень на конкретному пристрої.

Практичне значення роботи реалізується у двох площинах. По-перше, для класифікаційних задач у розширеннях запропоновано конкретну робочу конфігурацію - палетизація на 4 біти з виконанням у режимі cpuAndNeuralEngine, що підтверджена інтеграцією у Custom Keyboard Extension. По-друге, для задач генерації сформульовано обхідні архітектурні схеми: винесення інференсу мовної моделі у головний застосунок з обміном даними з розширенням через XPC або спільний контейнер App Group, або застосування переважно енкодерних моделей для класифікаційно-шаблонних задач замість повноцінної генерації.

Перспективи подальших досліджень включають детальну характеризацію меж описаного обмеження за допомогою серії моделей різних архітектур та розмірів, перевірку запропонованої архітектурної схеми, де розширення може спілкуватися з основним застосунком, а також моніторинг змін у програмному стеку iOS, що можуть надати розширенням повноцінний доступ до ANE у майбутніх версіях системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Xu J., Li Z., Chen W., Wang Q., Gao X., Cai Q., Ling Z. On-Device Language Models: A Comprehensive Review. arXiv.org. 2409.00088. 26.08.2024. URL: https://arxiv.org/abs/2409.00088.
2. Understand How an App Extension Works. developer.apple.com. URL: https://developer.apple.com/library/archive/documentation/General/Conceptual/ExtensibilityPG/ExtensionOverview.html.
3. Identifying high-memory use with jetsam event reports. developer.apple.com. URL: https://developer.apple.com/documentation/xcode/identifying-high-memory-use-with-jetsam-event-reports.
4. Core ML - Documentation. developer.apple.com. URL: https://developer.apple.com/documentation/coreml.
5. Core ML Tools - Documentation. apple.github.io. URL: https://apple.github.io/coremltools/docs-guides/.
6. Deploying Transformers on the Apple Neural Engine. machinelearning.apple.com. URL: https://machinelearning.apple.com/research/neural-engine-transformers.
7. MLX - Documentation. ml-explore.github.io. URL: https://ml-explore.github.io/mlx/build/html/index.html.
8. Rajesh V., Jodhpurkar O., Anbuselvan P., Singh M., Jallepali A., Godbole S., Sharma P. K., Shrivastava H. Production-Grade Local LLM Inference on Apple Silicon: A Comparative Study of MLX, MLC-LLM, Ollama, llama.cpp, and PyTorch MPS. arXiv.org. 2511.05502. 09.10.2025. URL: https://arxiv.org/abs/2511.05502.
9. Foundation Models. developer.apple.com. URL: https://developer.apple.com/documentation/FoundationModels.
10. How to get Apple Intelligence. support.apple.com. URL: https://support.apple.com/en-us/121115.

11. Data Controls. wisprflow.ai. URL: https://wisprflow.ai/data-controls.
12. Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser L., Polosukhin I. Attention Is All You Need. arXiv.org. 1706.03762. 12.06.2017. URL: https://arxiv.org/abs/1706.03762.
13. Ainslie J., Lee-Thorp J., de Jong M., Zemlyanskiy Y., Lebron F., Sanghai S. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv.org. 2305.13245. 22.05.2023. URL: https://arxiv.org/abs/2305.13245.
14. Pope R., Douglas S., Chowdhery A., Devlin J., Bradbury J., Levskaya A., Heek J., Xiao K., Agrawal S., Dean J. Efficiently Scaling Transformer Inference. arXiv.org. 2211.05102. 09.11.2022. URL: https://arxiv.org/abs/2211.05102.
15. Gholami A., Kim S., Dong Z., Yao Z., Mahoney M. W., Keutzer K. A Survey of Quantization Methods for Efficient Neural Network Inference. arXiv.org. 2103.13630. 25.03.2021. URL: https://arxiv.org/abs/2103.13630.
16. Wu H., Judd P., Zhang X., Isaev M., Micikevicius P. Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. arXiv.org. 2004.09602. 20.04.2020. URL: https://arxiv.org/abs/2004.09602.
17. Han, S., Mao, H., Dally, W. J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv.org. 1510.00149. 01.10.2015. URL: https://arxiv.org/abs/1510.00149.
18. Frantar E., Alistarh D. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. arXiv.org. 2301.00774. 02.01.2023. URL: https://arxiv.org/abs/2301.00774.
19. Hollemans M. Everything we actually know about the Apple Neural Engine. github.com. URL: https://github.com/hollance/neural-engine.
20. Kumaresan R. Orion: Characterizing and Programming Apple's Neural Engine for LLM Training and Inference. arXiv.org. 2603.06728. 06.03.2026. URL: https://arxiv.org/abs/2603.06728.

21. Deploying Attention-Based Vision Transformers to Apple Neural Engine. machinelearning.apple.com. URL: https://machinelearning.apple.com/research/vision-transformers.
22. «Anemll-style» Root-Mean-Square (RMS) Normalization on the Apple Neural Engine: A Simple Hack. huggingface.co. URL: https://huggingface.co/blog/anemll/anemll-style-rms-ane.
23. On Device Llama 3.1 with Core ML. machinelearning.apple.com. URL: https://machinelearning.apple.com/research/core-ml-on-device-llama.
24. Cuenca P., Fleetwood C., Srivastav V., Sanseviero O. WWDC 24: Running Mistral 7B with Core ML. huggingface.co. URL: https://huggingface.co/blog/mistral-coreml.
25. Testing in Simulator versus testing on hardware devices. developer.apple.com. URL: https://developer.apple.com/documentation/xcode/testing-in-simulator-versus-testing-on-hardware-devices.
26. What is the memory limit for a network extension? Apple Developer Forums. URL: https://developer.apple.com/forums/thread/73148.
27. Llama 3.2 Community License Agreement. llama.com. URL: https://www.llama.com/llama3_2/license/.
28. Qwen2.5: A Party of Foundation Models. qwenlm.github.io. URL: https://qwenlm.github.io/blog/qwen2.5-llm/.
29. Qwen2.5-0.5B-Instruct. huggingface.co. URL: https://huggingface.co/Qwen/Qwen2.5-0.5B-Instruct.
30. john-rocky. CoreML-LLM: Run LLMs on Apple devices with CoreML, optimized for Apple Neural Engine. github.com. URL: https://github.com/john-rocky/CoreML-LLM.