Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Метод тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами

Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.240159.01.01.ПЗ

Виконав студент 2 курсу, група ІПЗм-24-1 _____ Віра Ваховська
Підпис Ім'я, ПРІЗВИЩЕ

Керівник д-р фіз.-мат. наук, професор _____ Леонід БЕДРАТЮК
Науковий ступінь, звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер ст. викладач _____ Ганна БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

11 грудня 2025 р.

Хмельницький 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
01.09.2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Ваховській Вірі Миколаївні
Прізвище, ім'я, по батькові здобувача

1. Тема роботи Метод тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами

Керівник роботи Бедратюк Леонід Петрович, д-р фіз.-мат. наук, професор
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 25.08.2025 р. № 65
2. Строк подання студентом роботи на кафедру 15.12.2025 р.
3. Вихідні дані до роботи Матеріали науково-дослідної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1. Проаналізувати предметну область, мету й завдання дослідження.
2. Коротко оглянути наявні параметроефективні методи тонкого налаштування ВММ.
3. Розробити й описати метод SkeMA та його основні властивості.
4. Подати архітектуру й реалізацію SkeMA в обраній програмній системі.
5. Провести експерименти, оцінити результати та сформулювати висновки.
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Бедратюк Г. І., старший викладач	30.11.25	01.12.25
Антиплагіат	Форкун Ю. В., к.т.н., доцент	01.07.25	02.12.25

7. Дата видачі завдання « 01 » вересня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Вивчення предметної області, формулювання мети та завдань дослідження, визначення об'єкта та предмета	20.10-26.10.2025	
2 Робота над розділом 1 «Теоретичні основи досліджуваної проблеми»	27.10-25.09.2025	
3 Робота над розділом 2 «Метод параметрично ефективного тонкого налаштування для розгортання в обмежених середовищах»	26.10-02.11.2025	
4 Робота над науковими статтями	03.11-9.11.2025	
5 Робота над розділом 3 «Архітектура програмної реалізації»	10.11-16.11.2025	
6 Робота над розділом 4 «Програмна реалізація та оцінювання»	17.10-23.11.2025	
7 Попередній захист кваліфікаційної роботи	24.11.2025-31.11.2025	
8 Узгодження постановки задачі, отриманих результатів і висновків, редакція та оформлення пояснювальної записки й графічних матеріалів відповідно до чинних стандартів	1.12.-7.12.2025	
9 Перевірка роботи на наявність плагіату, нормоконтроль, брошурування пояснювальної записки, підготовка супровідних документів	08.12-14.12.2025	
10 Підготовка до захисту кваліфікаційної роботи	з 15.12.2025 р.	

Студент _____ Віра ВАХОВСЬКА
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

РЕФЕРАТ

Тема дипломної роботи: «Метод тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами».

Автор роботи: Ваховська Віра Миколаївна.

Керівник роботи: Бедратюк Леонід Петрович.

Пояснювальна записка: 100 с., 13 рис., 9 табл., 3 дод., 32 джерела.

ВЕЛИКІ МОВНІ МОДЕЛІ, ПАРАМЕТРОЕФЕКТИВНЕ ТОНКЕ НАЛАШТУВАННЯ, ОПТИМІЗОВАНІ АДАПТЕРИ, НИЗЬКОРАНГОВІ ОНОВЛЕННЯ, МАТРИЧНЕ СКЕТЧУВАННЯ, SKEMA, МОВНЕ МОДЕЛЮВАННЯ, ОБМЕЖЕНІ ОБЧИСЛЮВАЛЬНІ РЕСУРСИ.

Об’єкт дослідження – процес тонкого налаштування великих мовних моделей у програмних середовищах з обмеженими обчислювальними ресурсами.

Предмет дослідження – метод SkeMA (Sketched Matrix Adapter) для параметроефективного тонкого налаштування великих мовних моделей, його архітектура, реалізація та поведінка у порівнянні з іншими методами налаштування.

Мета роботи – підвищити ефективність тонкого налаштування великих мовних моделей шляхом розроблення та дослідження нового параметроефективного методу SkeMA, який забезпечує високу якість адаптації за фіксованого бюджету додаткових параметрів без збільшення навантаження на інференсі.

Для досягнення мети розв’язано такі основні задачі:

– проаналізовано сучасні підходи до тонкого налаштування LLM та параметроефективні методи (адаптери, низькорангові оновлення, квантування);

– визначено вимоги до методу, орієнтованого на середовища з обмеженими обчислювальними ресурсами;

– розроблено математичну модель та архітектуру методу SkeMA;

– реалізовано програмний модуль адаптера та інтегровано його в типову архітектуру трансформерної мовної моделі;

– спроєктовано та проведено експерименти з порівняння SkeMA з методом LoRA на задачах мовного моделювання та запам'ятовування асоціативних пар;

– виконано аналіз отриманих результатів і оцінено практичну придатність запропонованого методу.

Наукова новизна одержаних результатів полягає в тому, що вперше запропоновано метод SkeMA, у якому оновлення ваг формується в стислому підпросторі за допомогою фіксованих проєкцій, а компактний навчальний блок адаптера повністю зливається з базовими вагами моделі після навчання. Показано, що за однакового бюджету тренованих параметрів SkeMA може мати вищу ефективну розмірність (ранг) оновлення, ніж класичні низькорангові методи, зокрема LoRA, та забезпечує нульове додаткове навантаження на інференсі.

Методи дослідження: аналіз літературних джерел, формалізація та аналітичне дослідження властивостей методу SkeMA, комп'ютерне моделювання й програмна реалізація адаптера у середовищі Python / PyTorch, експериментальне дослідження на моделі Pythia-160M.

У результаті роботи створено програмний прототип методу SkeMA та проведено серію експериментів, які показали, що за однакового процентного бюджету параметрів (~0,2% від повної моделі) SkeMA досягає вищої точності на задачі запам'ятовування та нижчої перплексії на задачі мовного моделювання порівняно з LoRA, наближаючись до якості повного донавчання. Результати можуть бути використані для адаптації великих мовних моделей у наукових і промислових програмних системах, що працюють на апаратурі з обмеженими обчислювальними ресурсами, а також як основа для подальшого розвитку параметроефективних методів тонкого налаштування.

5.12.2025

Дата

Підпис

ABSTRACT

Thesis Title: "A method for fine-tuning large language models using optimized adapters for deployment in software environments with limited computational resources"

Author: Vira Vakhovska.

Supervisor: Leonid Bedratyuk.

Explanatory Note: 100 p., 13 pc., 9 tb., 3 add., 32 src.

LARGE LANGUAGE MODELS, PARAMETER-EFFICIENT FINE-TUNING, ADAPTERS, SKEMA, MATRIX SKETCHING, LANGUAGE MODELING, LIMITED COMPUTATIONAL RESOURCES.

Object of the research – the process of fine-tuning large language models in software environments with limited computational resources.

Subject of the research – the SkeMA (Sketched Matrix Adapter) method for parameter-efficient fine-tuning of large language models, its architecture, implementation, and behaviour in comparison with other tuning methods.

The objective of the thesis is to increase the efficiency of fine-tuning large language models by developing and investigating a new parameter-efficient method, SkeMA, which provides high adaptation quality under a fixed budget of additional parameters without increasing inference-time overhead.

To achieve this objective, the following main tasks were addressed:

– to analyse modern approaches to fine-tuning LLMs and parameter-efficient methods (adapters, low-rank updates, quantization);

– to define requirements for a method targeted at environments with limited computational resources;

– to develop the mathematical model and architecture of the SkeMA method;

– to implement the adapter software module and integrate it into a typical transformer-based language model architecture;

– to design and conduct experiments comparing SkeMA with the LoRA method on language modeling and associative pair memorization tasks;

– to analyse the obtained results and evaluate the practical applicability of the proposed method.

The scientific novelty of the obtained results lies in the fact that the SkeMA method is proposed for the first time, in which the weight update is formed in a compressed subspace using fixed projections, while the compact trainable adapter block is fully merged into the base model weights after training. It is shown that, under the same budget of trainable parameters, SkeMA can have a higher effective dimensionality (rank) of the update than classical low-rank methods, in particular LoRA, and provides zero additional inference-time overhead.

Research methods include: analysis of literature sources; formalization and analytical study of the properties of the SkeMA method; computer modelling and software implementation of the adapter in Python / PyTorch; experimental evaluation on the Pythia-160M model.

As a result of the work, a software prototype of the SkeMA method was created and a series of experiments was carried out, showing that, for the same percentage budget of trainable parameters (~0.2% of the full model), SkeMA achieves higher accuracy on the memorization task and lower perplexity on the language modeling task compared to LoRA, approaching the quality of full fine-tuning. The results can be used for adapting large language models in scientific and industrial software systems running on hardware with limited computational resources, as well as a basis for further development of parameter-efficient fine-tuning methods.

5.12.2025

Date

Signature

ЗМІСТ

ВСТУП

Широке впровадження великих мовних моделей (LLM) у прикладні програмні системи, що охоплюють інформаційний пошук, діалогові інтерфейси, витяг знань та автоматизацію документообігу, поставило перед розробниками високі вимоги до обчислювальних ресурсів, обсягу пам’яті та латентності. Хоча повномасштабне тонке налаштування (full fine-tuning) мільярдних моделей дозволяє досягти високої точності, його практична реалізація є економічно недоцільною та технічно складною [1]. Висока вартість апаратної інфраструктури, тривалий цикл експериментів, а також складність розгортання та супроводу роблять такий підхід малопридатним для організацій з обмеженими ресурсами та для сценаріїв використання на периферійних пристроях. Це, у свою чергу, стимулює попит на параметроефективні методи адаптації (Parameter-Efficient Fine-Tuning, PEFT), які дозволяють модифікувати функціональну поведінку моделі шляхом оновлення незначної частки її параметрів [2].

Водночас практичне застосування існуючих PEFT-методів виявляє низку критичних експлуатаційних обмежень, що звужують сферу їх використання у промислових системах. По-перше, репараметризовані низькорангові підходи (зокрема, LoRA), попри свою високу ефективність у задачах стилізації, часто демонструють недостатню виразну здатність при вирішенні складних когнітивних завдань або завдань, що вимагають інтенсивного запам’ятовування нових знань [3]. Це зумовлено фундаментальною властивістю методу: апроксимація матриці оновлень ваг ΔW добутком двох матриць фіксованого низького рангу r створює жорстку «стелю» виразності. Для подолання цього обмеження та засвоєння складної інформації розробники змушені суттєво збільшувати ранг матриць. Однак таке рішення призводить до лінійного зростання кількості тренованих параметрів, що фактично нівелює переваги параметричної ефективності методу та наближає витрати ресурсів до повного донавчання. По-друге, адитивні методи (класичні адаптери, префіксне налаштування) мають архітектурні недоліки, пов’язані зі зміною графу обчислень моделі. Класичні адаптери (типу Houlsby) інтегрують у

кожен шар трансформера додаткові модулі з нелінійними функціями активації, які неможливо математично злити з базовими вагами. Це вводить додаткові послідовні операції в процес обробки даних, що призводить до зростання затримки (latency) на етапі інференсу [4]. Аналогічно, методи налаштування префіксів (Prefix Tuning) ефективно збільшують довжину вхідної послідовності на кожному шарі, що підвищує обчислювальну вартість механізму уваги та сповільнює генерацію [5]. Зазначені фактори суттєво ускладнюють інтеграцію таких рішень у високопродуктивні системи з жорсткими вимогами до часу відгуку.

Актуальність теми зумовлена саме цим розривом між теоретичними прототипами PEFT та практичними вимогами промислової експлуатації. Існує потреба у методах, що забезпечують швидку адаптацію великих мовних моделей до специфічних даних без деградації продуктивності системи. Особливої ваги набувають питання відтворюваності досліджень та розробка науково обґрунтованих інженерних рекомендацій щодо вибору оптимальної розмірності адаптації для досягнення балансу в тріаді «якість – затримка – пам’ять».

Метою даної роботи є підвищення ефективності тонкого налаштування великих мовних моделей шляхом розробки та дослідження нового параметроефективного методу SkeMA (Sketched Matrix Adapter), що використовує структуровані матричні проєкції для мінімізації обчислювальних витрат на етапі інференсу.

Для досягнення поставленої мети необхідно вирішити такі задачі:

– проаналізувати існуючі підходи до тонкого налаштування LLM та параметроефективні методи (PEFT);

– визначити вимоги до методу тонкого налаштування, орієнтованого на середовища з обмеженими обчислювальними ресурсами;

– розробити математичну модель та архітектуру методу SkeMA;

– здійснити програмну реалізацію розробленого методу;

– спроєктувати та провести експерименти з порівняння SkeMA з методом LoRA на репрезентативних завданнях

– виконати аналіз отриманих результатів та оцінити практичну придатність.

1 ТЕОРЕТИЧНІ ОСНОВИ ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ

1.1 Аналіз предметної області і виявлення наявних проблем та завдань

Мовні моделі є фундаментальною технологією сучасної обробки природної мови (NLP) [6]. Їхній розвиток пройшов через кілька ключових етапів, кожен з яких приносив кардинальні зміни у підходах до моделювання. Ця еволюція являє собою перехід від простих статистичних методів до надскладних нейромережевих архітектур, які здатні розуміти та генерувати текст на рівні, близькому до людського [7]. Хронологія цього прогресу проілюстрована на рисунку 1.1.

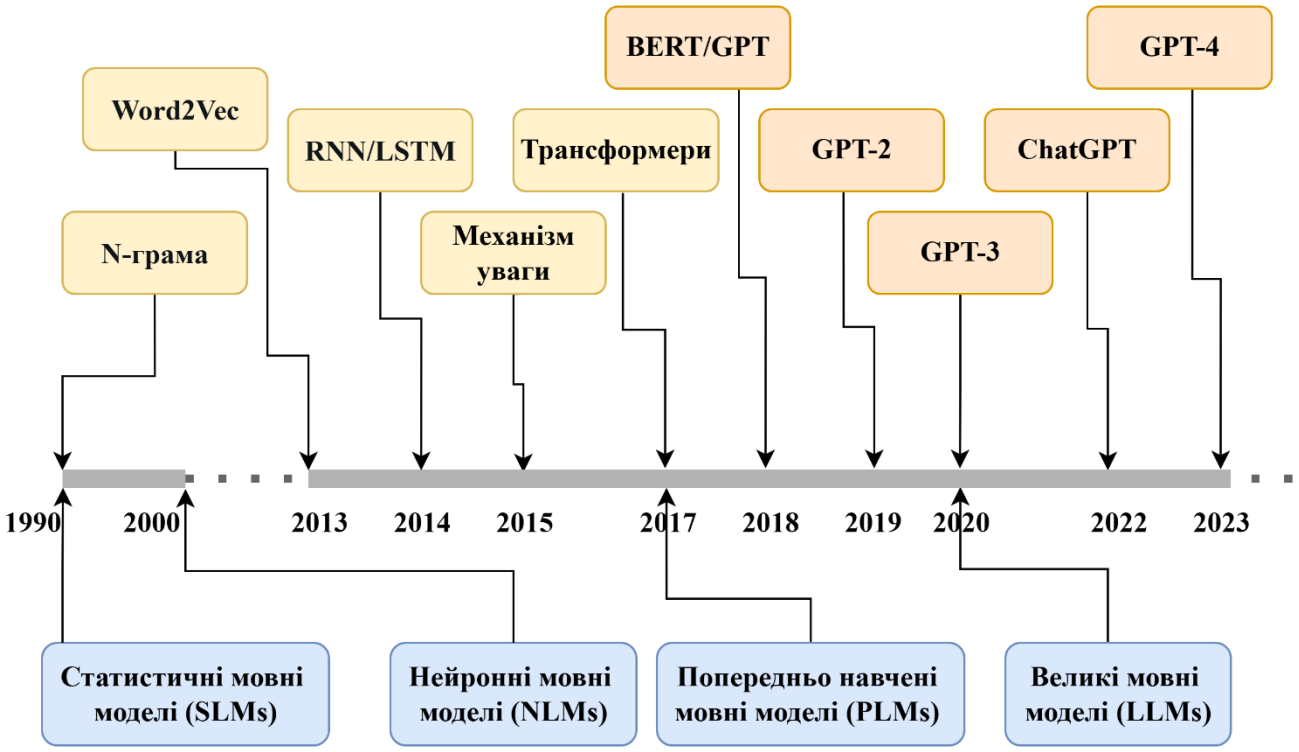


Рисунок 1.1 – Хронологічна шкала розвитку великих мовних моделей

Початковий етап (1990-ті роки) базувався переважно на статистичних N-грамних моделях. У 2000-х роках відбувся зсув у бік нейронних мовних моделей (NLM). Завдяки технології векторних представлень слів (word embeddings), зокрема Word2Vec, ці моделі почали значно краще вловлювати семантичні (змістові) зв’язки між словами [8]. Подальшими вирішальними кроками стали рекурентні нейронні мережі (RNN/LSTM) у 2013–2014 роках та впровадження

механізму уваги (Attention mechanism) у 2014 році, який дозволив моделям фокусуватися на найважливіших частинах вхідного тексту.

Справжньою революцією стала презентація архітектури Трансформер (Transformer) у 2017 році [9]. Вона стала основою для попередньо навчених моделей (PLM), таких як BERT та перші версії GPT (2018 рік). Це поклало початок ері великих мовних моделей (LLM). Ця ера ознаменувалася виходом GPT-3 у 2020 році та подальшим стрімким, експоненційним зростанням масштабу та можливостей, яке продемонстрували ChatGPT та GPT-4 у 2022–2023 роках [10].

Розуміння сучасних LLM вимагає детального аналізу еволюційних етапів, кожен з яких вирішував обмеження попереднього, поступово нарощуючи складність та функціональні можливості моделей [11].

Статистичні мовні моделі (SLMs що набули поширення у 1990-х роках, базуються на імовірнісних методах для оцінки правдоподібності текстових послідовностей. Фундаментально, підхід спирається на ланцюгове правило ймовірностей, де ймовірність речення обчислюється як добуток умовних імовірностей кожного слова за умови наявності попередніх. Однак через обчислювальну складність та проблему розрідженості даних (data sparsity) на практиці застосовується N-грамна модель. Вона вводить спрощення (Марковське припущення), що ймовірність слова залежить лише від $N-1$ попередніх слів. Незважаючи на свою простоту, N-грамні моделі мають суттєві недоліки: вони демонструють низьку ефективність при роботі з рідкісними словами та нездатні фіксувати довгострокові залежності у тексті, оскільки їхній аналізований контекст обмежений розміром N.

Нейронні мовні моделі (NLMs), що з’явилися у 2000-х, використовують нейронні мережі для прогнозування послідовностей слів, долаючи обмеження SLM. Ключовою інновацією стало використання векторних представлень слів (word embeddings), які дозволяють представляти слова у векторному просторі, де семантично близькі слова мають близькі вектори.

Типова архітектура NLM складається з трьох основних шарів (рис. 1.2):

– вхідний шар (input layer): об’єднує вектори слів контексту;

– прихований шар (hidden layer): застосовує нелінійну функцію активації для виявлення складних залежностей;

– вихідний шар (output layer): прогнозує наступне слово за допомогою функції softmax, яка перетворює вихідні значення у ймовірнісний розподіл по всьому словнику.

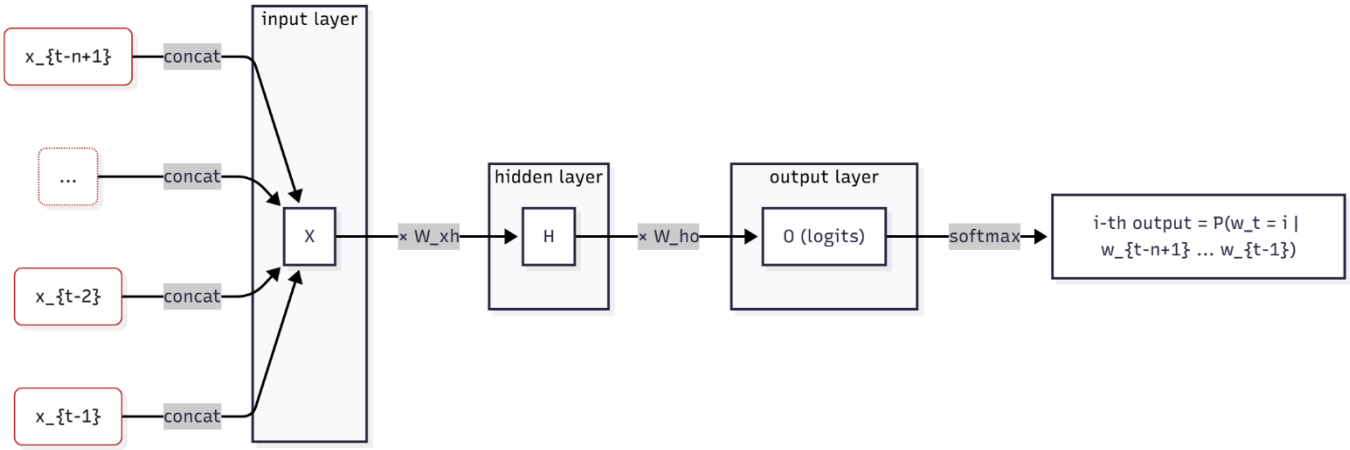


Рисунок 1.2 – Архітектура нейронних мовних моделей

Поява попередньо навчених мовних моделей (PLMs) ініціювала перехід до нової парадигми, відомої як «попереднє навчання та тонке налаштування» (pre-training and fine-tuning). Цей процес складається з двох фаз. На етапі попереднього навчання модель тренується на величезних нерозмічених текстових корпусах (наприклад, Вікіпедія) за допомогою самокерованого навчання (self-supervised learning). Завдання, як-от прогнозування наступного слова чи заповнення пропусків, змушують модель засвоювати фундаментальні лінгвістичні закономірності.

На другому етапі тонкого налаштування (fine-tuning) вже навчена модель донавчається на меншому, маркованому наборі даних, специфічному для конкретної прикладної задачі. Цей двохетапний підхід, яскраво представлений моделями BERT та GPT-2, продемонстрував високу ефективність: він дозволяє досягати високої якості на цільових завданнях при значно менших обчислювальних витратах порівняно з навчанням з нуля.

Великі мовні моделі (LLMs), такі як GPT-3, GPT-4, PaLM та LLaMA, є прямим масштабуванням ідеї PLM [11]. Вони вирізняються навчанням на текстових даних обсягом у терабайти та кількістю параметрів, що сягає десятків і сотень мільярдів. Процес їх створення також зазвичай включає два ключові етапи:

– попереднє навчання (pre-training): навчання на величезному загальному корпусі тексту для формування базових знань про мову та світ;

– вирівнювання (alignment): додаткове налаштування моделі для узгодження її поведінки з людськими цінностями та інструкціями [12].

Одним з найвідоміших методів вирівнювання є навчання з підкріпленням на основі зворотного зв’язку від людини (RLHF) [13]. Саме цей масштаб та згаданий процес навчання надають сучасним LLM їхні вражаючі можливості, включно з навчанням у контексті та здатністю до складних міркувань.

Процес адаптації LLM до специфічних завдань є ключовим для їх практичного застосування. Фундаментально розрізняють два етапи: попереднє навчання (pre-training) та тонке налаштування (fine-tuning). Попереднє навчання – це ресурсомісткий процес тренування моделі на величезних масивах нерозмічених даних для формування загальних мовних знань, що може тривати місяцями. Натомість тонке налаштування є процесом подальшої спеціалізації вже навченої моделі на меншому, специфічному для конкретного завдання наборі даних. Цей підхід є значно швидшим та дешевшим.

Важливість тонкого налаштування полягає у його здатності ефективно переносити загальні знання моделі на вузькоспеціалізовані домени, суттєво підвищуючи її продуктивність у конкретних задачах. Це дозволяє досягати високої якості з меншими обсягами даних, забезпечує швидшу збіжність моделі та робить її більш придатною для ефективного розгортання в реальних застосунках.

Існують різні типи тонкого налаштування, вибір яких залежить від наявних даних та мети [14]:

– ненаглядове тонке налаштування використовується для адаптації моделі до специфічної термінології та стилю певного домену (наприклад, медицини) на нерозмічених текстах;

– наглядове тонке налаштування (SFT) є найпоширенішим і передбачає навчання на розмічених даних у форматі «вхід-вихід» для вирішення конкретних завдань, як-от класифікація;

– інструктивне тонке налаштування є різновидом SFT, де модель навчається слідувати інструкціям, поданим у форматі природної мови, що є основою для створення чат-ботів та асистентів.

Альтернативним підходом до адаптації моделі є генерація з підкріпленням за допомогою пошуку (RAG). На відміну від тонкого налаштування, яке змінює ваги самої моделі, RAG залишає модель незмінною, але на момент запиту динамічно знаходить релевантну інформацію у зовнішній базі знань і подає її моделі як додатковий контекст [15, 16]. Цей метод є ефективним, коли потрібно забезпечити відповіді на основі актуальних або приватних даних, що часто змінюються.

Вибір між RAG та тонким налаштуванням не є взаємовиключним і залежить від завдання. RAG є кращим вибором, коли завдання вимагає доступу до великої кількості зовнішніх, динамічних знань [17]. Тонке налаштування, у свою чергу, є незамінним, коли необхідно змінити внутрішню поведінку, стиль або формат відповідей моделі, а не просто надати їй нові факти. Таким чином, для завдань, що потребують глибокої модифікації поведінки моделі, розробка нових ефективних методів тонкого налаштування залишається пріоритетним напрямом досліджень.

Хоча сучасні LLM демонструють високу якість, їхня інтеграція в реальні програмні системи відбувається під впливом суворих операційних обмежень. До них належать ліміти на обсяг відеопам’яті (VRAM) та оперативної пам’яті (ОЗП), жорсткі вимоги до затримки відповіді (latency) в онлайн-сервісах та необхідність розгортання в контейнеризованих середовищах. За таких умов повномасштабне тонке налаштування (full fine-tuning), що передбачає оновлення всіх параметрів моделі, стає не лише економічно недоцільним, але й несе ризики деградації продуктивності на етапі інференсу. Саме ці виклики стимулюють зростання наукового та інженерного інтересу до параметроефективних підходів (PEFT) [18], за яких базові ваги фіксуються, а навчанню підлягають лише компактні додаткові модулі.

Проте, поширені PEFT-рішення також мають свої компроміси. Класичні адаптери, що додають окремі обчислювальні блоки в кожен шар, неминуче збільшують кількість операцій під час інференсу, що призводить до зростання затримки. Низькорангові схеми (напр., LoRA) ефективно керують параметричним бюджетом, однак їхня виразна здатність обмежується апріорно вибраним рангом. Методи на основі префіксів та підказок мінімізують кількість тренованих параметрів, але не інтегруються у ваги і підвищують обчислювальну вартість обробки довгих послідовностей. Квантування, хоч і суттєво економить пам’ять, вимагає обережної інтеграції з адаптаційними модулями для уникнення втрати стабільності та якості

Цей аналіз виявляє практичну та наукову прогалину: існує потреба в методі, який би за малого параметричного бюджету забезпечував високу гнучкість оновлень, повністю інтегрувався в базові ваги та не створював додаткового навантаження (overhead) під час експлуатації. Постає наукове завдання: обґрунтувати та розробити конструкцію параметроефективного адаптера для трансформерних архітектур, який би навчався у стислому підпросторі, зберігав корисну геометричну структуру представлень, після навчання безпечно зливався з базовими вагами, забезпечував вищу ефективну розмірність оновлення за того самого параметричного бюджету і, водночас, був сумісним з обчисленнями у форматах низької точності (квантуванням).

1.2 Порівняльний аналіз переваг та недоліків існуючих рішень

Параметроефективне тонке налаштування (PEFT) є не єдиним методом, а родиною підходів, що відрізняються стратегіями модифікації попередньо навчених моделей. Для систематизації існуючих рішень їх можна класифікувати за тим, як саме вони взаємодіють з параметрами базової моделі. Загальна таксономія цих підходів включає адитивні, селективні, репараметризовані та гібридні методи, як систематизовано у таблиці 1.1 [19].

Таблиця 1.1 – Класифікація методів PEFT

Категорія	Основна ідея	Приклади методів
Адитивне налаштування (Additive Fine-tuning)	Додавання нових навчальних параметрів до архітектури моделі, залишаючи базові ваги незмінними.	На основі адаптерів: AdapterFusion, AdaMix. На основі м'яких підказок: Prefix-tuning, Prompt-tuning. Інші: $(IA)^3$, SSF.
Селективне налаштування (Selective Fine-tuning)	Навчання лише певної, ретельно обраної підмножини існуючих параметрів моделі.	Маскування: FishMask, BitFit (структурне та неструктурне). Пошарова адаптація: Оновлення лише певних шарів.
Репараметризоване налаштування (Reparameterised Fine-tuning)	Репараметризація ваг моделі з використанням низькорангових представлень для зменшення кількості навчальних параметрів.	Низькорангова декомпозиція: LoRA, DoRA, Compacter, VeRA. Похідні LoRA: DyLoRA, AdaLoRA, LoRA+, LoRAHub.
Гібридне налаштування (Hybrid Fine-tuning)	Комбінація кількох PEFT-підходів для досягнення кращого балансу між ефективністю та якістю.	Уніфіковані підходи: UniPELT, MAM Adapter, LLM-Adapter.

Як систематизовано в таблиці 1.1, методи PEFT поділяються на кілька категорій. Для цілей даного дослідження, що фокусується на розгортанні моделей в середовищах з обмеженими ресурсами, найважливішою є відмінність між методами, що створюють додаткове навантаження (overhead) на етапі інференсу, та тими, що його не створюють. Цей критерій визначає практичну придатність методу для систем з жорсткими вимогами до швидкості відповіді. Тому подальший аналіз буде зосереджено на порівнянні ключових представників та виявлення їхніх фундаментальних компромісів.

Адитивні методи передбачають введення в архітектуру моделі додаткових параметрів, що підлягають навчанню. Одним з перших та найбільш відомих підходів у цій категорії є метод адаптерів, запропонований Хоулсбі та співавторами, який полягає у додаванні невеликих, окремих модулів до кожного шару трансформера, залишаючи основні параметри незмінними.

Зазвичай модулі адаптерів послідовно вставляються двічі на блок трансформера: після шару багатоканальної уваги (multi-head attention) та після

шару прямого поширення (feed-forward network). Кожен адаптер є компактним двошаровим перцептроном з архітектурою «пляшкового горла» (bottleneck) і складається з трьох компонентів:

– лінійний шар, що зменшує розмірність (down-projection);

– нелінійна функція активації;

– лінійний шар, що відновлює початкову розмірність (up-projection).

Вихід адаптера додається до основного шляху через залишкове з'єднання (residual connection), що дозволяє навчатись невеликій поправці до виходу шару та забезпечує стабільність навчання. Під час донавчання заморожуються всі ваги трансформера, а навчанню підлягають лише параметри цих компактних модулів (за потреби – також шари нормалізації та кінцевий класифікатор). Завдяки такій архітектурі адаптери додають лише незначну кількість параметрів (зазвичай від 0,5% до 8% від оригінальної моделі), при цьому досягаючи продуктивності, порівнянної з повним тонким налаштуванням. Загальну архітектуру та місце інтеграції адаптерів показано на рисунку 1.3.

Переваги адаптерів Хоулсбі:

– висока параметрична ефективність: дозволяють суттєво зменшити кількість тренованих параметрів, що знижує вимоги до пам'яті та обчислень під час навчання;

– модульність і багаторазове використання: оскільки для кожної задачі навчається окремий модуль, одну базову модель можна легко адаптувати під багато завдань, зберігаючи для кожної свої параметри;

– швидке та стабільне навчання: заморожування основних ваг запобігає катастрофічному забуванню, сприяє швидшій збіжності та дозволяє досягати високої якості навіть на невеликих обсягах даних;

– стабільна ініціалізація: адаптери можна ініціалізувати близькими до нуля вагами, завдяки чому вони майже не впливають на модель на старті донавчання, що сприяє більш плавному підлаштуванню.

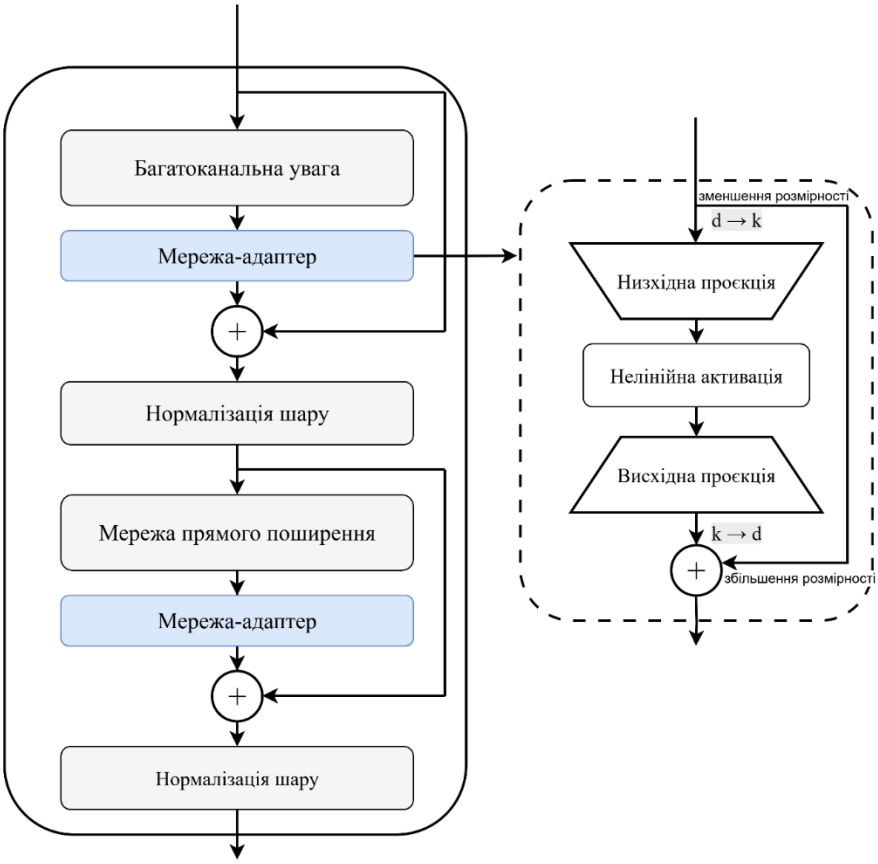


Рисунок 1.3 – Архітектура та інтеграція послідовного адаптера

Недоліки адаптерів Хоулсбі:

– додаткові витрати на інференс: кожен адаптер є додатковим обчислювальним блоком, який потрібно виконати на кожному шарі, що збільшує час та споживання пам'яті під час прогнозування;

– неможливість злиття з моделлю: на відміну від LoRA, через наявність нелінійної функції активації, адаптери не можна повністю інтегрувати в існуючі ваги моделі;

– чутливість до гіперпараметрів: ефективність залежить від вибору розмірності вузького шару m, що вимагає додаткових експериментів для пошуку оптимального значення;

– додаткова пам'ять при багатозадачності: хоча кожен модуль малий, одночасне завантаження адаптерів для багатьох задач може суттєво збільшити споживання ресурсів.

Незважаючи на високу параметричну ефективність, головний недолік адаптерів полягає в тому, що вони залишаються окремими обчислювальними блоками в архітектурі. Це призводить до збільшення затримки (latency) на етапі інференсу, оскільки для отримання результату необхідно виконати додаткові операції [20].

Іншою значною групою адитивних методів є підходи, що базуються на м'яких підказках, такі як Prompt Tuning та Prefix Tuning. На відміну від адаптерів, що втручаються в архітектуру кожного шару, ці методи впливають на модель через її вхідні дані.

Prompt-tuning передбачає додавання до вхідної послідовності набору навчуваних векторних представлень (soft prompts) на рівні embedding-шару. Базові ваги моделі залишаються замороженими; оптимізуються лише ці вектори, які кондиціюють модель на конкретне завдання.

Переваги Prompt-tuning:

– архітектурна простота: не потребує змін у шарах моделі, додаються лише навчувані вектори на вході;

– мінімальна кількість навчуваних параметрів: десятки або сотні векторів, дуже малий додатковий слід;

– економія ресурсів на навчанні: заморожені базові ваги зменшують витрати пам'яті та обчислень;

– збереження знань базової моделі: мінімізує ризик катастрофічного забування;

– ефективність на великих моделях: зі зростанням масштабу якість наближається до повного тонкого налаштування;

– підтримка багатозадачності: для різних завдань підключаються невеликі файли-підказки.

Недоліки Prompt-tuning:

– зниження якості на менших моделях: часто поступається повному тонкому налаштуванню або альтернативним PEFT-підходам;

– інференс-оверхед: подовження послідовності зменшує ефективне контекстне вікно і збільшує затримку;

– відсутність злиття з вагами: навчувані вектори лишаються зовнішніми артефактами і потребують менеджменту;

– чутливість до ініціалізації та довжини підказки: некоректні налаштування погіршують збіжність і якість;

– залежність від знань базової моделі: слабка ефективність для навичок, відсутніх у передтренуванні;

– обмежена інтерпретованість: вектори-підказки не мають очевидної семантики.

Prefix-tuning інтегрує треновані «префіксні» вектори всередині кожного трансформер-блоку як додаткові ключі та значення механізму самоуваги. На практиці часто навчається компактний латентний вектор, який через невеликий MLP розгортається у матриці ключів/значень потрібної розмірності. Базові ваги моделі заморожені.

Переваги Prefix-tuning:

– висока продуктивність на генеративних задачах: добре працює в режимах із браком даних;

– параметрична ефективність: якість близька до повного тонкого налаштування за частки відсотка параметрів;

– економія ресурсів на навчанні: базові ваги заморожені, оновлюються лише префікси;

– збереження знань базової моделі: зменшує ризик катастрофічного забування;

– підтримка багатозадачності: зберігаються невеликі «патчі»-префікси для різних завдань;

– вища виразна здатність порівняно з prompt-tuning: інформація додається на кожному шарі.

Недоліки Prefix-tuning:

– інференс-оверхед: додаткові ключі/значення обробляються на кожному шарі, що збільшує затримку;

– підвищені вимоги до пам’яті: збільшується KV-кеш і зменшується доступна довжина контексту на довжину префіксу;

– архітектурна складність: потребує модифікацій для вставки префіксів у шари;

– відсутність злиття з вагами: необхідний менеджмент окремих префікс-файлів;

– залежність від знань базової моделі: обмежена ефективність поза доменом передтренування;

– чутливість до вибору довжини префіксу: некоректний розмір погіршує якість.

Хоча ці методи є ще більш економними за кількістю параметрів (часто навчається менше 0.1% від ваг моделі), вони мають ті ж фундаментальні недоліки, що й класичні адаптери. По-перше, вони збільшують обчислювальне навантаження та затримку під час інференсу, оскільки ефективно подовжують послідовність, яку необхідно обробити. По-друге, навчені вектори неможливо математично «злити» з базовими вагами моделі, що вимагає зберігання та менеджменту окремих файлів-підказок для кожного завдання. Таким чином, всі адитивні підходи об’єднує компроміс: вони забезпечують високу параметричну ефективність, але ціною постійного оверхеду на етапі експлуатації.

BitFit (Bias-Term Fine-Tuning) є гранично простим підходом селективного налаштування, в якому тренуються лише параметри зміщення (bias) в нейромережі. Цей метод фіксує всі вагові матриці моделі, дозволяючи оновлювати тільки коефіцієнти зміщення в кожному шарі, а також параметри фінального класифікатора. Інтуїтивно це означає, що модель підлаштовується до нового завдання шляхом незначного зміщення активацій нейронів, але не змінюючи самі лінійні трансформації, задані вагами.

BitFit є одним з найбільш екстремальних підходів у плані скорочення кількості тренованих параметрів, яка зазвичай становить не більше 0.1% від

загальної кількості. Це кардинально знижує вимоги до обчислювальних ресурсів та пам'яті як під час навчання, так і для зберігання фінальних моделей. Незважаючи на таку простоту, дослідження показали, що BitFit часто демонструє продуктивність, близьку до повного тонкого налаштування, особливо на класифікаційних задачах.

Переваги BitFit:

– мінімальні обчислювальні витрати: оновлення лише bias-термів суттєво економить GPU-пам’ять і прискорює обчислення під час тренування;

– відсутність оверхеду на інференсі: оскільки зміщення є стандартною частиною обчислень лінійних шарів, метод не додає жодних додаткових операцій чи затримок;

– простота реалізації та стабільність навчання: не вимагає складних архітектурних змін, а мала кількість параметрів робить навчання менш схильним до перенавчання;

– збереження якості моделі: на багатьох завданнях, зокрема з набору GLUE, метод досягає результатів, майже ідентичних повному донавчанню;

– сумісність з іншими методами: через свою простоту легко поєднується з іншими PEFT-підходами, наприклад, з LoRA, для досягнення додаткового приросту якості.

Обмеження BitFit:

– обмежена виразна здатність: bias-зміщення здійснюють лише адитивний вплив на активації і можуть бути недостатніми для завдань, що потребують суттєвої зміни поведінки моделі;

– залежність від початкових ознак: успіх методу залежить від того, наскільки релевантні характеристики вже витягнуті замороженою моделлю для нового завдання;

– непридатність для довивчення: оскільки метод не змінює основні ваги, він не може накопичувати нові знання і призначений лише для адаптації, а не для продовження навчання моделі;

– невелике зниження якості: на складних задачах типово спостерігається невелике падіння точності порівняно з більш гнучкими методами, як-от LoRA чи адаптери.

IA^3 (Infused Adapters by Inhibiting and Amplifying Inner Activations) належить до адитивних методів і частково знімає окреслений вище компроміс. На відміну від адаптерів Хоулсбі та підказкових підходів, IA^3 не додає окремих обчислювальних блоків і не подовжує послідовність: навчаються вектори масштабування, якими поелементно перезважуються ключі/значення в багатоканальній увазі та проміжні активації у FFN на кожному шарі при заморожених базових вагах.

Переваги IA^3:

– мінімальний параметричний слід: навчаються вектори масштабування для уваги та FFN;

– відсутність інференс-оверхеду після злиття: множники можна попередньо «влити» у ваги;

– проста інтеграція: не додає окремих модулів і не подовжує послідовність;

– сумісність із різними архітектурами трансформера: застосовується як на увазі, так і на проміжних активаціях;

– стабільне навчання: обмежені за формою оновлення зменшують ризик деградації.

Недоліки IA^3:

– обмежена виразна здатність: мультиплікативне масштабування менш гнучке, ніж модулі або низькорангові оновлення;

– чутливість до вибору місць масштабування: необхідно визначити, де застосовувати вектори (K/V, FFN);

– залежність від гіперпараметрів: потрібен добір сил масштабування та регуляризації;

– можливі програші на складних завданнях: коли потрібні суттєві структурні зміни у представленнях.

Таким чином, IA^3 зберігає параметричну ефективність адитивних підходів і водночас зменшує інференс-оверхед, однак поступається за виразністю методам, що безпосередньо змінюють або репараметризують ваги моделі.

Репараметризовані методи використовують низькорангові перетворення для зменшення кількості навчальних параметрів, оперуючи при цьому з високовимірними матрицями ваг. Найпопулярнішим представником цієї категорії є LoRA (Low-Rank Adaptation).

Основна ідея LoRA полягає в тому, що матриця оновлень ваг (ΔW) для будь-якого лінійного шару має низький внутрішній ранг. Тому, замість навчання повнорозмірної матриці ΔW, LoRA апроксимує її як добуток двох значно менших, низькорангових матриць: $\Delta W = W_{down} W_{up}$. Під час навчання базові ваги моделі (W) залишаються замороженими, а оновлюються лише параметри матриць W_{down} та W_{up}.

Переваги використання LoRA:

– ефективність параметрів: значно знижує кількість параметрів для тренування;

– ефективне зберігання: вимагає зберігати лише невеликі низькорозрядні матриці;

– зниження обчислювального навантаження: пришвидшує процес тренування та покращує масштабованість;

– зменшення вимог до пам’яті: дозволяє використовувати більші батчі або складніші моделі;

– гнучкість: легко інтегрується в існуючі архітектури без суттєвих змін;

– сумісність: можна використовувати разом з іншими методами налаштування;

– порівнянні результати: демонструє продуктивність, порівнянну з повним налаштуванням;

– адаптація до завдань: ефективно адаптує модель до специфічних завдань;

– запобігання перенавчанню: допомагає уникнути перенавчання на невеликих наборах даних.

Обмеження LoRA:

– масштаб налаштування: може бути менш ефективним у завданнях, що вимагають значних змін у моделі;

– оптимізація гіперпараметрів: потребує ретельного підбору рангу r для досягнення оптимальної продуктивності;

– статус дослідження: метод все ще перебуває на етапі активного вивчення.

Обмеження фіксованого рангу r стимулювали низку модифікацій LoRA, що підвищують виразність без відмови від злиття (merge).

Ключовою перевагою LoRA є те, що після завершення навчання навчені матриці можна математично злити з базовими вагами моделі ($W' = W + W_{down}W_{up}$). Це дозволяє повернутися до початкової архітектури, завдяки чому метод має нульовий обчислювальний оверхед на етапі інференсу.

Однак, цей метод також має фундаментальне обмеження: його виразна здатність жорстко обмежена вибраним рангом r. Для адаптації до складних завдань може знадобитися збільшення рангу, що збільшує кількість параметрів. Цей компроміс стимулював розвиток численних похідних методів, як DyLoRA та AdaLoRA, що намагаються динамічно підбирати оптимальний ранг під час навчання [21].

Основна ідея AdaLoRA (Adaptive LoRA) полягає в адаптивному розподілі бюджету параметрів між різними шарами та матрицями моделі, на відміну від рівномірного використання єдиного фіксованого рангу в LoRA. Оскільки деякі шари можуть бути важливішими для конкретного завдання, AdaLoRA автоматично перерозподіляє виділений параметричний бюджет туди, де він забезпечує найбільший виграш.

Для цього метод представляє оновлення ваг через сингулярний розклад (SVD), оптимізуючи сингулярні вектори та значення замість прямого тренування матриць A і B. Під час навчання спеціальна метрика оцінює важливість кожного компонента оновлення для зменшення функції втрат. На основі цієї оцінки

AdaLoRA виконує динамічну проріджувальну регуляризацію: менш значущі компоненти поступово обнуляються, що ефективно зменшує ранг у менш важливих місцях. Глобальний планувальник бюджету контролює загальну кількість активних параметрів, плавно знижуючи її до цільового рівня під кінець тренування.

Архітектурно AdaLoRA не змінює модель: після навчання результатом є набір звичайних LoRA-матриць, хоча й з різними рангами на різних шарах, які так само можна повністю злити з базовими вагами. Єдиною відмінністю є ускладнений процес навчання, що вимагає періодичного обчислення SVD та оцінки важливості компонентів. Експериментально доведено, що AdaLoRA перевершує звичайну LoRA за метриками якості, особливо в умовах низького бюджету параметрів, досягаючи кращих результатів з меншою кількістю тренованих параметрів.

Переваги AdaLoRA:

– ефективний розподіл параметрів: автоматично розподіляє параметричний бюджет, посилюючи оновлення у важливих шарах та послаблюючи в менш значущих;

– автоматичний підбір рангу: усуває необхідність ручного підбору рангу r, оскільки фактичний ранг для кожної матриці встановлюється динамічно під час навчання;

– вища продуктивність при низькому бюджеті: демонструє кращу якість порівняно з LoRA за однакової або навіть меншої кількості параметрів, особливо в низькоресурсних налаштуваннях;

– стабільна збіжність: усунення «шумових» параметрів сприяє більш стабільному навчанню та зменшує перенавчання на несуттєвих напрямах;

– відсутність затримок при інференсі: як і LoRA, адаптери можуть бути повністю злиті з базовими вагами, що забезпечує нульовий оверхед під час експлуатації;

– потенційне прискорення інференсу: якщо адаптери не зливати, менший ефективний ранг у деяких шарах може призвести до меншої кількості обчислень порівняно зі статичною LoRA.

Обмеження AdaLoRA:

– складність налаштування: метод вводить додаткові гіперпараметри, такі як розклад зниження бюджету та параметри метрики важливості, що вимагає ретельнішої калібровки;

– ризик надмірного проріджування: неправильне налаштування гіперпараметрів може призвести до того, що модель втратить важливі компоненти оновлення;

– додаткові витрати на навчання: обчислення SVD та оцінок важливості додає певний оверхед до часу тренування та може вимагати більше оперативної пам’яті;

– методологічні питання проріджування: як і інші методи проріджування, AdaLoRA успадковує питання щодо того, чи завжди агресивне видалення параметрів веде до оптимального результату.

MoRA – варіант низькорангового налаштування, що підвищує ефективний ранг оновлення без збільшення параметричного бюджету [22]. Між фіксованими проєкціями (стисненням та розширенням) вводиться одна тренована квадратна матриця розміру $c \times c$; базові ваги моделі залишаються замороженими, а після завершення навчання оновлення зливаються з базовими вагами, повертаючи початкову архітектуру.

Така архітектура дозволяє отримати оновлення ваг підвищеного рангу за тієї ж кількості параметрів, що й у LoRA. Наприклад, для шару розмірністю 4096, LoRA з рангом $r=8$ та MoRA з квадратною матрицею $c \approx 256$ матимуть приблизно однакову кількість тренованих параметрів (~65 тисяч). Ключовою властивістю методу є те, що фіксовані оператори обираються таким чином, щоб після навчання оновлення можна було повністю злити (merge) з базовими вагами моделі, усуваючи будь-який оверхед під час інференсу.

Дослідження показали, що MoRA перевершує LoRA на завданнях, що вимагають запам'ятовування нових знань або суттєвого розширення можливостей моделі, таких як задачі безперервного донавчання. На інших типах завдань,

наприклад, інструкційному донавчанні, MoRA демонструє продуктивність на рівні LoRA.

Переваги MoRA:

– високорангове оновлення: дозволяє досягати вищого ефективного рангу оновлення ваг за того ж параметричного бюджету, що й у LoRA;

– вища продуктивність на складних завданнях: перевершує LoRA на пам'яттєво-інтенсивних задачах та задачах безперервного донавчання, де потрібне засвоєння нової інформації;

– параметрична ефективність: кількість тренованих параметрів збігається з LoRA (зазвичай <1% від повної моделі), що зберігає переваги економії пам'яті;

– відсутність затримок при інференсі: як і LoRA, адаптери можуть бути повністю злиті з базовими вагами, що забезпечує нульовий обчислювальний оверхед під час експлуатації

Обмеження MoRA:

– обмежена виразність при малому бюджеті: хоча ранг є високим, його ефективність все одно обмежена розміром квадратної матриці c, який залежить від загального бюджету параметрів;

– складність реалізації: введення нетренованих операторів стиснення та розширення додає рівень складності в реалізацію порівняно з LoRA;

– залежність від вибору операторів: продуктивність методу може залежати від вибору конкретного типу фіксованих операторів (наприклад, вибір підматриць, випадкові проєкції тощо);

– невеликі додаткові витрати на навчання: операції стиснення та розширення можуть трохи збільшити час і споживання пам'яті на етапі тренування порівняно з LoRA.

Навіть із гнучким рангом природа оновлення лишається «цілісною» для матриці ваг; наступний крок – розділити вагу на величину та напрям. Також, через те, що LoRA оновлює ваги поступово, не змінюючи кардинально їхні основні характеристики. Для подолання цього розриву було запропоновано новий метод –

вагове розкладання для низькорангової адаптації (Weight-Decomposed Low-Rank Adaptation, DoRA).

DoRA – це методологія, що розкладає попередньо навчені ваги моделі на дві окремі компоненти: величину (magnitude) та напрямок (direction) [23]. Замість того, щоб застосовувати LoRA до всієї матриці ваг, DoRA використовує її лише для оновлення компоненти напрямку, тоді як величина тренується окремо як додатковий навчальний параметр. Такий підхід дозволяє робити більш значні та точні оновлення, імітуючи патерни навчання повного тонкого налаштування, але зберігаючи при цьому параметричну ефективність LoRA.

Ключовою перевагою DoRA є покращена навчальна здатність, оскільки розкладання ваг дозволяє здійснювати більш точні та значущі оновлення. При цьому метод зберігає параметричну ефективність, оскільки використовує структуру LoRA для оновлення лише однієї з компонент. Важливо, що, як і LoRA, DoRA є повністю зливаною (fully mergeable), а отже, не додає жодних додаткових обчислень чи затримок під час інференсу. До того ж, метод продемонстрував свою універсальність та ефективність на різних архітектурах, включно з LLM та мультимодальними моделями (LVLM).

Хоча обидві техніки спрямовані на зменшення обчислювальних витрат, вони використовують різні стратегії, це узагальнено в таблиці 1.2.

Таблиця 1.2 – Порівняльний аналіз LoRA та DoRA

Критерій	LoRA (Low-Rank Adaptation)	DoRA (Weight-Decomposed Low-Rank Adaptation)
Мета	Ефективне донавчання шляхом поступового оновлення ваг без затримок на інференсі.	Покращення навчальної здатності до рівня FT, оптимізуючи величину та напрямок ваг окремо.
Підхід	Моделює оновлення ваг (ΔW) як добуток двох низькорангових матриць ($A \cdot B$).	Розкладає ваги (W) на величину (m) та напрямок (V), після чого застосовує LoRA лише до напрямку (V).
Архітектура	Зберігає початкову матрицю ваг незмінною, додаючи паралельне обчислення ΔW.	Реструктурує матрицю ваг на компоненти, забезпечуючи більш точні та цілеспрямовані оновлення.

Переваги DoRA:

– покращена навчальна здатність: DoRA досягає навчальної здатності, близької до повного тонкого налаштування (FT), завдяки розкладанню ваг на компоненти величини та напряму;

– ефективне тонке налаштування: використовуючи структурні переваги LoRA для оновлення напряму, DoRA забезпечує ефективну адаптацію без змін архітектури моделі;

– відсутність затримок при інференсі: попри покращені навчальні можливості, DoRA не додає додаткових затримок під час інференсу, оскільки повністю зливається з базовими вагами;

– вища продуктивність: експериментальні результати показують, що DoRA стабільно перевершує LoRA у широкому спектрі завдань (NLP, LVLM);

– універсальність для різних архітектур: метод протестований та підтвердив свою ефективність на різних архітектурах, включно з LLM та мультимодальними моделями;

– інноваційний аналіз: введення аналізу вагового розкладання сприяє глибшому розумінню динаміки тонкого налаштування.

Обмеження DoRA:

– низькорангова стеля виразності: оновлення напрямку все ще залишаються низькоранговими, що вимагає підвищення рангу для складних завдань;

– відсутність явних геометричних гарантій: розкладання «величина/напрям» не гарантує збереження геометрії у підпросторах представлень;

– чутливість до гіперпараметрів: окреме навчання компонент потребує ретельного підбору параметрів (швидкість навчання, регуляризація) для уникнення нестабільності;

– додатковий навчальний слід у пам’яті: порівняно з LoRA, з’являються додаткові параметри масштабу та операції нормування, що збільшує стан оптимізатора;

– непрозоре масштабування по шарах: вимагає ручного або евристичного підбору рангу для різних матриць (Q/K/V/FFN);

– складна взаємодія з квантуванням: при роботі з 4-бітними базовими вагами можливі нестійкі градієнти та деградація якості без ретельної калібровки;

– обмежена теоретична пояснюваність: аналіз методу наразі є здебільшого емпіричним, без строгих теоретичних оцінок ефективного рангу.

Під назвою FLORA (Flexible LoRA) фігурує низка підходів, що мають на меті підвищити гнучкість LoRA-адаптерів, дозволяючи варіювати або комбінувати їхні ранги залежно від потреб завдання чи обчислювального середовища. Було запропоновано два основні сценарії застосування цієї ідеї.

Перший, FlexLoRA, призначений для федеративного навчання. Він дозволяє клієнтам з різними обчислювальними можливостями тренувати адаптери різних рангів: потужніші вузли використовують вищий ранг, а слабші — нижчий. На сервері ці різнорідні оновлення агрегуються і за допомогою сингулярного розкладу (SVD) перетворюються на єдиний глобальний LoRA-адаптер. Такий підхід дозволяє ефективніше використовувати гетерогенні ресурси та створювати більш якісну глобальну модель.

Другий, Fast LoRA (FLORA), є інфраструктурною оптимізацією для систем реального часу з багатьма різнорідними запитами. Він дозволяє обробляти в одному батчі запити, кожен з яких може вимагати свого унікального LoRA-адаптера (наприклад, для різних завдань або персоналізації). Завдяки зміні реалізації матричних операцій, Fast LoRA уможливлює паралельний інференс з різними адаптерами, що значно підвищує пропускну здатність системи без втрати якості адаптації.

Обидва підходи ґрунтуються на стандартній архітектурі LoRA, зберігаючи її ключові переваги, зокрема низьку кількість тренованих параметрів для кожного окремого адаптера та можливість повного злиття з базовою моделлю.

Переваги FLORA:

– підвищення пропускної здатності: у сервісних застосунках Fast LoRA дозволяє паралельно обробляти запити з різними адаптерами в одному батчі;

– ефективне використання гетерогенних ресурсів: у федеративному навчанні FlexLoRA вирішує проблему «найслабшого клієнта», дозволяючи потужнішим вузлам робити більший внесок у глобальну модель;

– покращена якість глобальної моделі: завдяки агрегації різнорангових оновлень у FlexLoRA, фінальна модель навчається більш загальним представленням і демонструє вищу якість на NLP-завданнях;

– сумісність та відсутність оверхеду: підходи сумісні зі стандартною реалізацією LoRA і не потребують зміни архітектури моделі. Отримані адаптери можуть бути повністю злиті з базовими вагами, усуваючи затримки на етапі інференсу.

Обмеження FLORA:

– специфічність застосування: переваги FlexLoRA проявляються лише у федеративних, мульти-клієнтських системах; у стандартному централізованому навчанні він не дає виграшу порівняно зі звичайним LoRA;

– інфраструктурна оптимізація, а не покращення якості: Fast LoRA не покращує якість самої адаптації, а лише оптимізує швидкість її застосування в багатозадачних середовищах;

– обчислювальні витрати на агрегацію: етап об’єднання адаптерів за допомогою SVD у FlexLoRA може бути ресурсоємним для дуже великих моделей, хоча й виконується офлайн;

– збільшення вимог до пам'яті при інференсі: Fast LoRA вимагає одночасного завантаження ваг кількох адаптерів у пам'ять для паралельної обробки батчу.

Подальший розвиток репараметризованих методів був спрямований на подолання не лише обчислювальних, але й обмежень пам’яті, що призвело до створення QLoRA (Quantized Low-Rank Adaptation) [25]. Цей метод є значною оптимізацією LoRA, що забезпечує ще вищу ефективність використання пам’яті шляхом квантування ваг базової моделі.

Основна ідея QLoRA полягає у тому, щоб представити велику, заморожену модель у 4-бітному форматі, тоді як невеликі навчальні LoRA-адаптери

зберігаються у більш високій, 16-бітній точності. Хоча це значно знижує точність представлення базових ваг, метод зберігає високу якість налаштування. Це досягається за рахунок того, що під час зворотного поширення помилки градієнти проходять через 4-бітну квантовану модель до 16-бітних адаптерів, де і відбувається оновлення ваг.

Такий підхід, доповнений інноваціями, як-от новий 4-бітний тип даних (NormalFloat), подвійне квантування та оптимізоване управління пам’яттю, дозволяє досягти 18-кратного скорочення вимог до пам'яті на один параметр (з 96 до 5.2 бітів). Це робить можливим тонке налаштування моделей з десятками мільярдів параметрів на одному споживчому GPU, що є значним кроком у демократизації доступу до технологій LLM. Процес взаємодії компонентів різної точності в QLoRA схематично зображено на рисунку 1.4.

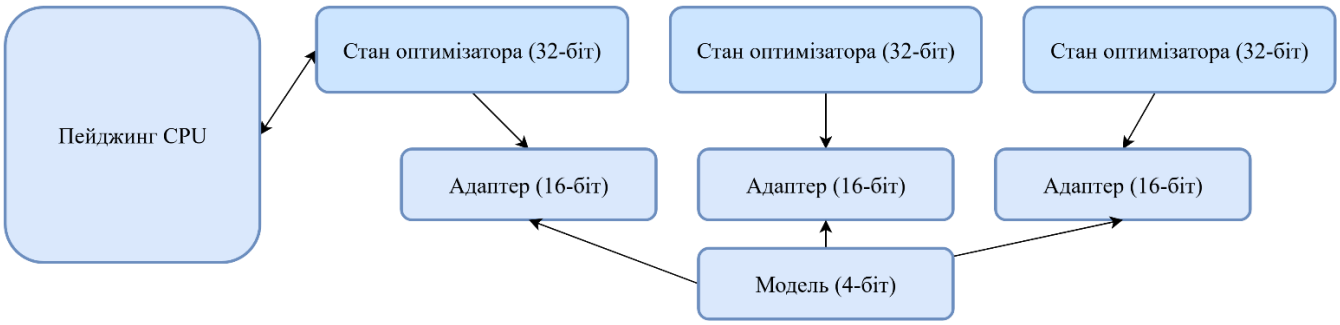


Рисунок 1.4 – Схематичне зображення процесу оптимізації QLoRA

З точки зору продуктивності, QLoRA демонструє результати, порівнянні з 16-бітним тонким налаштуванням, проте незважаючи на значні переваги в ефективності пам’яті, QLoRA успадковує архітектурне обмеження LoRA, пов’язане з фіксованим рангом r.

Підсумовуючи вищесказане, можна виділити основні обмеження існуючих PEFT-рішень:

– оверхед на етапі інференсу: адитивні методи (адаптери) додають нові обчислювальні блоки, що збільшує затримку (latency) під час експлуатації моделі;

– обмежена виразна здатність: низькорангові схеми мають «стелю» виразності, що залежить від фіксованого рангу r;

– спотворення геометрії представлень: багато PEFT-методів не мають явних гарантій збереження важливих геометричних властивостей у підпросторах представлень;

– висока чутливість до гіперпараметрів: результат роботи методів сильно залежить від вибору рангу, шарів для адаптації та інших налаштувань;

– проблеми при взаємодії з квантуванням: агресивні 4/8-бітні формати можуть призводити до нестабільності процесу навчання;

– накопичення адаптерів: використання окремих адаптерів для різних завдань значно збільшує загальні вимоги до сховища;

– складність відтворення результатів: використання евристик та випадковості в деяких PEFT-методах ускладнює реплікацію експериментів;

– відсутність строгих теоретичних гарантій: аналіз більшості методів є переважно емпіричним, без формальних доведень щодо кінцевої якості.

2.3 Постановка задачі

Мета дослідження: обґрунтувати, розробити й емпірично оцінити метод SkeMA на основі матричного скетчування, який за сталого параметричного бюджету підвищує ефективну виразність оновлень і зберігає відсутність інференс-оверхеду після злиття.

Наукова гіпотеза: скетч-проєкції дозволяють будувати компактний, геометрично узгоджений підпростір оновлень, що забезпечує вищу ефективну розмірність за того самого бюджету навчуваних параметрів із можливістю повного злиття з базовими вагами.

Завдання дослідження:

– формалізувати конструкцію SkeMA та умови коректного злиття;

– реалізувати прототип із інтеграцією у стандартні MLOps-процеси;

– порівняти з LoRA за паритету кількості навчуваних параметрів на репрезентативних задачах;

– виміряти якість адаптації, час інференсу після злиття та пікове споживання пам’яті;

– провести абляції за розміром скетчу й перевірити стабільність у 8-бітних і 4-бітних режимах.

2.4 Висновки до 1-го розділу

У рамках даного розділу було проведено систематичний аналіз предметної області параметроефективного донавчання великих мовних моделей, що дозволило сформувати цілісне бачення проблеми їх розгортання в середовищах з обмеженими обчислювальними ресурсами.

Аналіз існуючих підходів виявив фундаментальний компроміс. Було показано, що адитивні методи (класичні адаптери, prompt/prefix-tuning) забезпечують високу параметричну ефективність, однак створюють постійне додаткове навантаження на етапі інференсу (inference overhead). На противагу їм, методи, що підтримують операцію злиття (merge-friendly), усувають цей недолік, але ціною обмеженої виразної здатності, що зумовлено, як правило, низькоранговою природою оновлень.

На основі виявленої наукової прогалини було обґрунтовано необхідність розробки методу, який би поєднував відсутність інференс-оверхеду з вищою ефективною розмірністю оновлень за сталого параметричного бюджету. Як відповідь на поставлену проблему було запропоновано архітектуру нового скетч-орієнтованого адаптера SkeMA. Сформульовано основну гіпотезу дослідження та визначено план її емпіричної перевірки, що передбачає порівняння з еталонним методом LoRA за умов суворого паритету параметрів.

Таким чином, положення, викладені у даному розділі, створюють теоретичне підґрунтя для подальшого опису програмної реалізації методу SkeMA та експериментальної верифікації його практичної доцільності.

2 МЕТОД ПАРАМЕТРИЧНО ЕФЕКТИВНОГО ТОНКОГО НАЛАШТУВАННЯ ДЛЯ РОЗГОРТАННЯ В ОБМЕЖЕНИХ СЕРЕДОВИЩАХ

2.1 Концепція підходу та місце в класифікації PEFT

Концепція SkeMA ґрунтується на декомпозиції адитивного оновлення ваг на два компоненти: компактний параметричний блок малої розмірності, параметри якого навчаються, та фіксовані лінійні проєкції, що визначають стислий підпростір дії оновлення. Під час тонкого налаштування оптимізуються лише параметри компактного блоку у зазначеному підпросторі, тоді як проєкційні оператори залишаються незмінними і задають місце та спосіб вбудування оновлення у ваговий простір. Після завершення навчання компактний блок розгортається через фіксовані проєкції до розміру відповідного шару, формуючи адитивне оновлення, яке інтегрується у базову матрицю ваг лінійним додаванням. У режимі інференсу додаткові обчислення відсутні, граф обчислень збігається з графом вихідної моделі, а ефект адаптації повністю перенесений у значення злитих ваг. Така організація забезпечує стабільність продуктивного виконання та узгодженість із апаратними і програмними обмеженнями розгортання, включно з конфігураціями квантування.

Ключові ознаки SkeMA:

- повне злиття оновлень у базові ваги моделі;
- компактний параметричний блок як носій варіативності, що навчається;
- фіксовані лінійні проєкції як механізм розгортання блоку;
- локальна інтеграція у компоненти уваги та FFN;
- сумісність із квантуванням параметрів.

У сукупності ці ознаки дають змогу спрямовано впливати на найбільш чутливі до адаптації підпростори без роздування кількості параметрів, що навчаються, і без накладних витрат у режимі інференсу. Вибір місць інтеграції у механізмі уваги та у проєкціях FFN забезпечує баланс між силою впливу і

стабільністю оптимізації, тоді як фіксовані проєкції відокремлюють питання місткості оновлень від питання їх розташування у ваговому просторі.

З позиції класифікації параметрично ефективних підходів SkeMA поєднує властивості репараметризованих і селективних схем. Подібно до репараметризації забезпечується повне злиття оновлень без змін у графі виконання. Подібно до селективних стратегій зберігається мінімалізм частини параметрів, що навчаються, проте носієм варіативності виступає структурований компактний блок у стислому підпросторі, а не скалярні чи векторні коефіцієнти. Це надає додатковий ступінь свободи для керування співвідношенням між заданим параметричним бюджетом та ефективною розмірністю оновлення.

Очікувані переваги SkeMA:

- нульове інференс-навантаження після злиття;
- контрольована місткість оновлень за сталого бюджету параметрів;
- підвищена виразність порівняно з жорстко низькоранговими схемами;
- модульність застосування у різних шарах трансформера;
- спрощення експлуатації та обігу артефактів адаптації.

Нульовий інференс-навантаження забезпечується тим, що після завершення тонкого налаштування адитивне оновлення ваг додається до базових параметрів, а компактний блок і проєкції вилучаються з графа обчислень. Таким чином, у продуктивному режимі кількість операцій і використання пам’яті збігаються з базовою моделлю [27]. Це твердження справедливе за умови злиття у тій самій числовій точності, у якій виконуватиметься інференс.

Контрольована місткість оновлень досягається параметризацією через невеликий компактний блок і фіксовані проєкції. Розмірність блоку прямо визначає кількість параметрів, що навчаються, а місця вставки дозволяють розподілити цей бюджет на найбільш чутливі компоненти моделі. Це дає прозорий механізм узгодження вимог якості з ресурсними обмеженнями.

Підвищена виразність порівняно з жорстко низькоранговими схемами пояснюється тим, що простір можливих оновлень задається не лише розмірністю компактного блоку, а й орієнтацією підпростору через проєкції. Навіть за того

самого бюджету параметрів це дозволяє краще наближати потрібні збурення ваг у критичних місцях моделі.

Модульність застосування випливає з локальної інтеграції в типові місця трансформера, зокрема у матриці уваги та у проєкції FFN. Підхід не накладає вимог на інші частини архітектури і може бути використаний частково або комбіновано, що полегшує адаптацію до різних конфігурацій моделей.

Спрощення експлуатації досягається відокремленням артефактів навчання від режиму виконання. На етапі розгортання зберігаються лише злиті ваги, а невеликі метадані компактного блоку можуть використовуватися для відтворення або переналаштування. Це зменшує обсяг артефактів і спрощує їх життєвий цикл у MLOps [28].

Обмеження застосування SkeMA:

- залежність якості від вибору проєкційних операторів;
- потреба в емпіричному доборі розмірності компактного блоку;
- чутливість до місця вставки в шарі;
- можливі переваги класичних low rank методів за дуже малих бюджетів;
- вимога коректного порядку операцій злиття у присутності квантування.

Залежність якості від вибору проєкційних операторів є природним наслідком того, що саме вони визначають підпростір, у якому діє оновлення. Невдалий вибір може обмежити виразність або ускладнити оптимізацію [30]. На практиці доцільно використовувати ортонормовані або структуровані проєкції і перевіряти їх на невеликих абляційних прогонах.

Потреба в емпіричному доборі розмірності компактного блоку зумовлена різною чутливістю задач і шарів до місткості оновлень. Значення, що є замалими, ведуть до недоадаптації, завеликі – до зайвих витрат часу і пам’яті під час навчання. Рекомендовано проводити короткі сіткові пошуки з фіксацією паритету параметрів у порівняннях.

Чутливість до місця вставки відображає нерівномірний внесок різних компонент трансформера у кінцеву якість. Ефективність застосування адаптаційних оновлень є гетерогенною в межах архітектури трансформера:

залежно від специфіки завдання, більш доцільною може бути їх інтеграція або в компоненти механізму уваги (проєкції Q/K/V), або в шари прямого поширення (FFN). Це слід перевіряти експериментально, так як універсального правила немає.

Можливі переваги класичних низько рангових методів за дуже малих бюджетів пояснюються їх низькими постійними множниками і зрілою практикою налаштування. Якщо обмеження на параметри вкрай жорсткі, LoRA з малим рангом може бути більш передбачуваним вибором [26].

Вимога коректного порядку операцій злиття у присутності квантування випливає з різниці числових форматів. Злиття бажано виконувати у вищій точності з подальшою перекалібровкою квантових масштабів або повторним квантуванням. Інакше можливе погіршення якості або нестабільність.

SkeMA розглядає оновлення кожного лінійного шару як «скетч» у низьковимірному підпросторі: спочатку вхід стискається фіксованою проєкцією до розмірності k, у цьому просторі застосовується невелике треноване ядро, після чого результат розгортається назад до розмірності виходу і додається до базового лінійного перетворення. На відміну від таких методів як LoRA, де навчальні параметри масштабуються як $r(d+m)$, у SkeMA вся навчальна місткість зосереджена у квадратній матриці розміру $k \times k$.

У загальній картині PEFT SkeMA належить до класу «адитивних» методів (оновлення додається до W), але відрізняється тим, що базові проєкції фіксовані (не навчаються), а єдиний тренований блок – ядро M у розмірності k. Це спрощує оптимізацію, зменшує пам’ять і полегшує розгортання у середовищах з обмеженими ресурсами.

2.2 Модель та властивості методу

Нехай лінійний шар базової моделі задано матрицею ваг $W \in R^{m \times d}$, що перетворює вхід $x \in R^d$ на вихід $y \in R^m$. Метод SkeMA вводить адитивне оновлення ваг у формі:

$$\Delta W = S_{\text{out}}^{T} M S_{\text{in}} \quad (1) \tag{1}$$

де $S_{\text{in}} \in R^{k \times d}$ – фіксована проєкція стиснення;

$S_{\text{out}} \in R^{k \times m}$ – фіксована проєкція розгортання;

$M \in R^{k \times k}$ – тренований компактний параметричний блок.

Під час навчання прямий прохід через модифікований шар набуває вигляду:

$$y = Wx + \Delta W x = Wx + S_{\text{out}}^{T} M S_{\text{in}} x \tag{2}$$

Обчислення за формулою (2) реалізує послідовність операцій: стиснення вхідного вектора активацій проєкцією S_{in}, застосування тренованого компактного блоку M у стислому підпросторі, розгортання результату проєкцією S_{out}^{T} та адитивне додавання отриманого внеску до виходу базового шару $W\,x$. Інтуїтивно, конструкція оновлення відповідає послідовності операцій: стиснення вхідного простору проєкцією S_{in}, застосування тренованого блоку M у стислому підпросторі та розгортання результату проєкцією S_{out}^{T} (рис. 2.1).

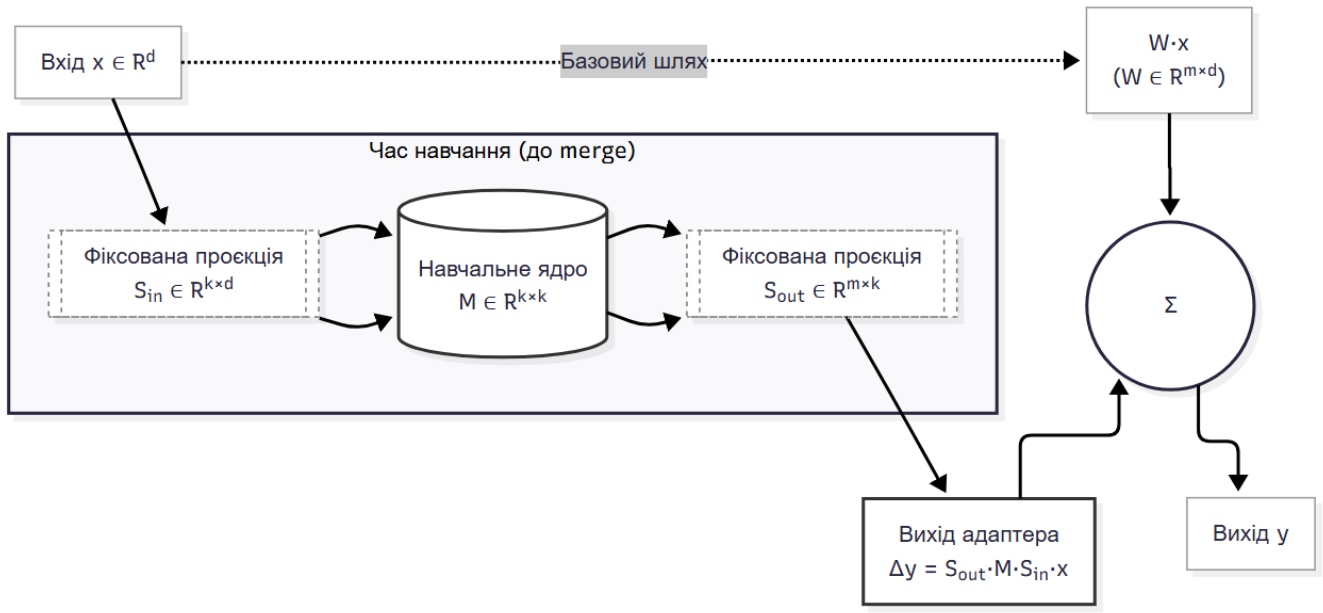


Рисунок 2.1 – Схема роботи SkeMA під час навчання

Вхід x стискається фіксованою проєкцією $S_{\text{in}} \in R^{k \times d}$, у стислому просторі застосовується трененоване ядро $M \in R^{k \times k}$, результат розгортається через $S_{\text{out}}^{T} \in R^{m \times k}$ і адитивно додається до базового шляху $W\ x$.

Адитивне оновлення ΔW, визначене у (1), зазвичай інтегрується в лінійні перетворення компонентів механізму уваги (проєкції Q, K, V) та/або двошарового MLP (FFN) в архітектурі трансформера. Схему типових точок вставки та детальний опис подано в наступному підрозділі.

Ключові властивості запропонованого методу SkeMA визначають його переваги та відмінності від існуючих підходів. По-перше, забезпечується повне злиття без додаткового навантаження на інференсі. Після завершення фази тонкого налаштування, фінальна матриця ваг W' обчислюється шляхом адитивного злиття:

$$W' = W + \Delta W \tag{3}$$

На етапі інференсу використовується лише матриця W', що робить обчислення повністю еквівалентними вихідній архітектурі моделі та усуває будь-яке додаткове навантаження (рис. 2.2).

У практичній реалізації операція злиття, визначена рівнянням (3), виконується після завершення процесу тонкого налаштування, як окремий офлайновий крок перед розгортанням моделі. Для забезпечення чисельної точності та стабільності рекомендується обчислювати адитивне оновлення (1) з використанням арифметики підвищеної точності, наприклад, у форматі float32. Отримана в результаті матриця оновлення ΔW додається до базової матриці ваг W, формуючи фінальну злиту матрицю W'. Саме ця матриця W' зберігається як артефакт адаптованої моделі.

Важливо підкреслити, що будь-які подальші операції подальшої обробки, такі як квантування ваг для зменшення розміру моделі чи прискорення інференсу, застосовуються вже до цієї фінальної злитої матриці W', а не до окремих компонентів (W, S_{in}, S_{out}, M). Це забезпечує сумісність методу SkeMA зі стандартними інженерними пайплайнами оптимізації моделей перед розгортанням.

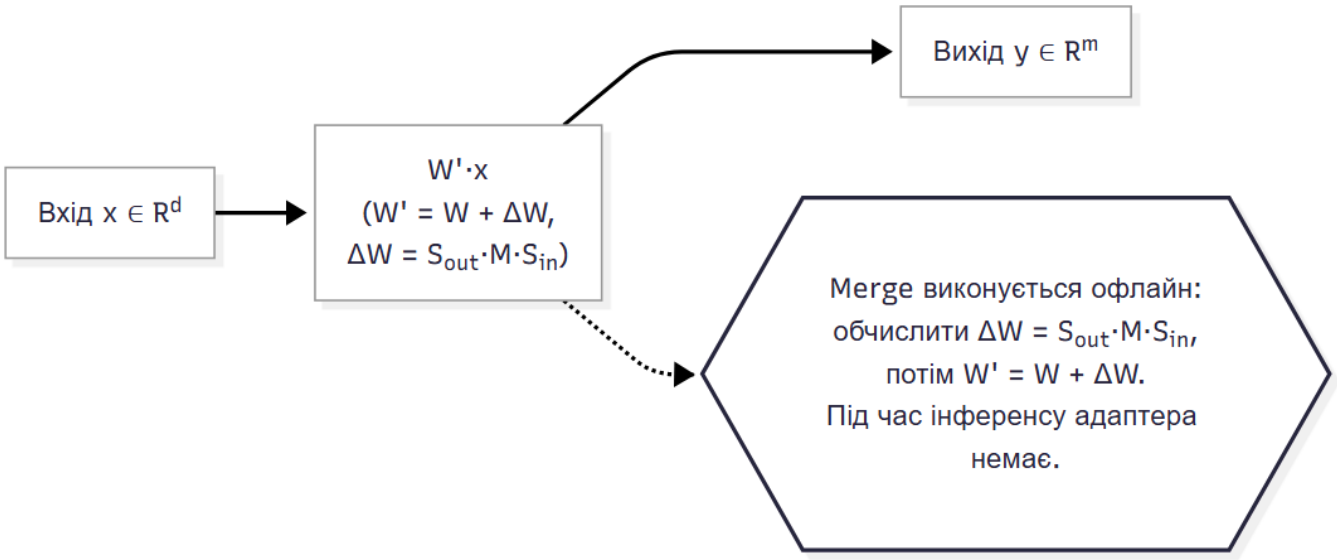


Рисунок 2.2 – Злиття після навчання: обчислення (1) та формування (3)

По-друге, метод надає керовану місткість оновлення. Кількість тренованих параметрів методу SkeMA дорівнює k^2, де k – розмірність стислого простору. Це дозволяє проводити коректне порівняння з методом LoRA за фіксованого бюджету тренованих параметрів P.

Для методу SkeMA залежність між бюджетом P та розмірністю стислого простору k визначається як:

$$P_{\text{SkeMA}} \approx k^2 \Rightarrow k \approx \sqrt{P} \tag{4}$$

Для методу LoRA, де тренуються дві матриці розмірністю $d \times r$ та $r \times m$, залежність між бюджетом P та рангом r має вигляд:

$$P_{\text{LoRA}} \approx r(d+m) \Rightarrow r \approx \frac{P}{d+m} \tag{5}$$

Верхня межа рангу для відповідних оновлень (4) та (5) становить $rank(\Delta W_{\text{SkeMA}}) \leq k$ та $rank(\Delta W_{\text{LoRA}}) \leq r$. Оскільки для типових розмірностей шарів $(d,\ m)$ виконується нерівність $\sqrt{P} \gg \frac{P}{d+m}$, метод SkeMA потенційно забезпечує вищу виразну здатність за того самого параметричного бюджету. Кількісне співвідношення між розмірністю стислого простору k для SkeMA та рангом r для LoRA за однакового параметричного бюджету P наведено в табл. 2.1.

Таблиця 2.1 – Паритет параметричного бюджету P для SkeMA та LoRA

Вхідна/Вихідна розмірність ($d = m$)	Бюджет параметрів (P)	$k \approx P$ (SkeMA)	$r \approx 2dP$ (LoRA)	Співвідношення k/r
2048	65 536	256	16	16×
2048	262 144	512	64	8×
4096	65 536	256	8	32×
4096	262 144	512	32	16×
4096	1 048 576	1024	128	8×

Як видно з таблиці 2.1, за сталого бюджету параметрів P зі зростанням розмірності шару d значення рангу $r = P/(2d)$ для LoRA зменшується, тоді як розмірність $k = \sqrt{P}$ для SkeMA залишається незмінною. Це призводить до зростаючого розриву у верхній межі рангу (а отже, й потенційної виразності) на користь SkeMA при застосуванні до великих лінійних шарів [29].

По-третє, забезпечується наближене збереження геометрії підпросторів. При використанні фіксованих проєкцій S_{in}, S_{out}, що задовольняють певним умовам (наприклад, ортонормальність або властивості проєкцій типу Джонсона-Лінденштрауса), операції стиснення та розгортання наближено зберігають геометричну структуру підпросторів. Це сприяє стабільності процесу оптимізації тренованого блоку M та забезпечує передбачувану реконструкцію фінального оновлення ΔW.

Модель оновлення ваг, що використовується в методі SkeMA, визначена рівняннями (1) та (2). Під час навчання вихід лінійного шару доповнюється внеском адаптера, який реалізує послідовність операцій: стиснення, обчислення в стислому просторі та розгортання (рис. 2.1). Після завершення навчання виконується офлайнова операція злиття згідно з (3), в результаті якої адитивне оновлення інтегрується у базові ваги (рис. 2.3). Для обґрунтування подальших рішень щодо вибору конфігурацій та інженерного пайплайну розгортання, а також для коректного порівняння з альтернативними підходами, необхідно формально зафіксувати ключові властивості методу SkeMA. Зокрема, будуть доведені два твердження: точність злиття, що гарантує відсутність додаткового навантаження на

етапі інференсу, та керована місткість оновлення та верхня межа його рангу, що пояснює зв'язок між параметром стиснення k, параметричним бюджетом та виразною здатністю методу.

Перша ключова властивість методу SkeMA полягає у точності операції злиття. Формально, нехай злиті ваги W' визначено за (3). Тоді для будь-якого вхідного вектора x обчислення виходу шару з активним адаптером, як у (2), є алгебраїчно еквівалентним обчисленню виходу шару зі злитими вагами W', те саме стосується й батча. Це випливає з лінійності (та асоціативності) матричних операцій: доданок адаптера, визначений у (1), є лінійним оператором, який у (3) поглинається в базову матрицю ваг. Наявний зсув b не змінюється, а масштаб адаптера α (якщо застосовувався) враховується у злитті. Така заміна виконується незалежно в кожній точці вставки і зберігає поведінку всієї мережі, оскільки структура графа обчислень не змінюється. Практично це означає відсутність будь-якого додаткового навантаження під час інференсу (робота відбувається лише з W'), а можливі числові розбіжності зумовлені лише dtype та квантуванням і розглядаються у наступних підрозділах (рис. 2.3).

Друга ключова властивість методу SkeMA стосується верхньої межі рангу адитивного оновлення. Формально, для оновлення ΔW, визначеного рівнянням (1), його ранг не перевищує розмірність стислого простору k. Це випливає з фундаментальної властивості рангу добутку матриць, який обмежений мінімальним рангом співмножників. Оскільки у виразі (1) кожен з трьох множників (S_{out}^{T}, M, S_{in}) має ранг, що не перевищує k, то і ранг їх добутку ΔW також обмежений цією величиною. Таким чином, параметр стиснення k безпосередньо контролює максимальну складність або виразну здатність оновлення. Практичним наслідком цього є можливість керування місткістю адаптера: порівнюючи методи за однакового бюджету тренованих параметрів P, співвідношення (4)–(5) та дані таблиці 2.1 показують, що для великих шарів SkeMA ($k \approx \sqrt{P}$) потенційно забезпечує вищу верхню межу рангу, а отже і вищу виразність, ніж LoRA ($r \approx P/(d+m)$) за того самого бюджету P. Важливо зазначити, що ця межа рангу застосовується локально до кожної точки інтеграції адаптера.

Отже, у цьому підрозділі було формалізовано модель адитивного оновлення SkeMA та показано її інтеграцію в архітектуру трансформера. Доведено дві ключові властивості. Показано, як параметр k контролює параметричний бюджет та потенційну виразність, що дозволяє проводити коректне порівняння з LoRA. Ці засади обґрунтовують практичну доцільність методу та будуть деталізовані у наступних підрозділах.

2.3 Алгоритм застосування та варіанти конфігурацій

У даному підрозділі деталізується алгоритм практичного застосування методу SkeMA, виходячи з теоретичної моделі, описаної у підрозділі 2.2. Фіксовані проєкції стиснення та розгортання (S_{in}, S_{out}) залишаються незмінними протягом усього процесу, тоді як оптимізації підлягає лише компактний параметричний блок (M) у стислому підпросторі. Після завершення навчання виконується операція злиття ваг згідно з рівнянням (3), і саме злита матриця W' є фінальним артефактом для розгортання та використання на етапі інференсу.

Перед початком застосування методу необхідно визначити наступні передумови та конфігураційні параметри:

– точки інтеграції: обрати підмножину лінійних шарів в архітектурі трансформера (наприклад, проєкції Q, K, V у модулі уваги та лінійні шари у FFN), до яких буде застосовано адаптер SkeMA;

– розмірність стислого простору k: обрати значення k, яке визначає як параметричний бюджет (k^2), так і виразну здатність оновлення (ранг $\leq k$);

– масштаб адаптера α: За потреби встановити коефіцієнт масштабування для адитивного оновлення (2);

– формат даних: узгодити обчислювальний формат (dtype) для етапів тренування та злиття;

– ініціалізація проєкцій: зафіксувати метод генерації фіксованих проєкцій S_{in}, S_{out} для забезпечення відтворюваності експериментів.

Алгоритм застосування методу SkeMA охоплює послідовність етапів від підготовки компонентів до розгортання адаптованої моделі. Основні етапи застосування методу SkeMA:

- підготовка фіксованих проєкцій;
- ініціалізація тренованого блоку;
- навчання з фіксованими проєкціями;
- обчислення адитивного оновлення;
- злиття з базовими вагами;
- санітарна перевірка еквівалентності;
- підготовка до інференсу та розгортання.

На першому етапі генеруються або завантажуються фіксовані матриці стиснення та розгортання (S_{in}, S_{out}). За потреби виконується їх ортонормалізація. Ці матриці зберігаються разом із метаданими (наприклад, стан генератора випадкових чисел, розмірності, метод побудови) для забезпечення відтворюваності.

Далі ініціалізується тренований компактний параметричний блок M у стислому просторі $k \times k$. Зазвичай використовується нульова або малоамплітудна випадкова ініціалізація. Адаптер підключається до обраних точок інтеграції в архітектурі трансформера згідно з визначеною конфігурацією.

Третій етап полягає у виконанні тонкого налаштування. Прямий прохід реалізує обчислення базового шару з додаванням внеску адаптера, відповідно до рівняння (2). Важливо, що градієнти обчислюються та оновлення застосовуються лише до параметрів компактного блоку M; базові ваги W та фіксовані проєкції S_{in}, S_{out} залишаються незмінними. Фіксуються параметри оптимізації, такі як розклад швидкості навчання та частота збереження контрольних точок.

Після завершення навчання обчислюється фінальне адитивне оновлення ΔW відповідно до рівняння (1). Рекомендується виконувати це обчислення у підвищеній точності (наприклад, float32), дотримуючись узгодженого порядку множення, і зберегти отриманий тензор ΔW.

Наступним кроком є злиття оновлення з базовими вагами. Формуються злиті ваги W' згідно з рівнянням (3). Отримана матриця W' зберігається як окремий артефакт. Параметри адаптера видаляються зі стану моделі, залишаючи лише злитий варіант для використання на етапі інференсу. Додаються відповідні метадані про конфігурацію злиття.

Перед розгортанням проводиться санітарна перевірка еквівалентності. На репрезентативній валідаційній підвибірці порівнюються виходи моделі до злиття (з активним адаптером) та після злиття (зі злитими вагами W'). Перевіряється збіг значень у межах заданого числового допуску, фіксується контрольна сума (хеш) злитих ваг та короткий звіт про результати перевірки.

Останній етап – підготовка до інференсу та розгортання. Для експлуатації використовується виключно модель зі злитими вагами W'. За потреби виконується післязлиттєве квантування ваг та експорт моделі до цільового формату виконання (наприклад, ONNX, TensorRT). Фінальний пакет артефактів для розгортання зберігається разом із протоколом відтворюваності.

Вибір конкретної підмножини точок інтеграції адаптера SkeMA (рис. 2.3) дозволяє створювати різні конфігурації, що балансують між параметричними витратами та потенційним впливом на поведінку моделі. Можна виділити три основні стратегії щодо компонентів блоку трансформера. Конфігурація «attention-only» застосовує оновлення лише до проєкцій запитів, ключів і значень у блоці багатоканальної уваги. Це мінімізує параметричні витрати і вважається доцільним для завдань, чутливих до довгих контекстних залежностей. Конфігурація «MLP-only» додає оновлення до обох лінійних шарів мережі прямого поширення (FFN), підсилюючи виразність нелінійної частини блоку, що може бути корисним для завдань, де важлива локальна трансформація ознак. Комбінована конфігурація «attention + MLP» охоплює обидва вузли і є збалансованим варіантом, що зазвичай забезпечує стабільний приріст якості. Додатково, вибір глибини охоплення шарів також є важливим конфігураційним параметром. Варіант «усі шари» забезпечує найбільш рівномірний вплив, але є найвитратнішим під час навчання. Конфігурація «верхні L_{top} шарів» концентрує адаптацію ближче до вихідних шарів моделі, що

часто є ефективним для інструкційних та класифікаційних завдань. Інтервальна схема «кожен s-й шар» дозволяє зменшити обчислювальну вартість тренування, зберігаючи при цьому суттєвий вплив на динаміку представлень, і є доцільною при обмежених ресурсах. З практичної точки зору, рекомендується розпочинати з комбінованої конфігурації («attention + MLP») та помірного значення розмірності стислого простору k. Надалі доцільно проводити абляційні дослідження, варіюючи точки інтеграції (наприклад, вимикаючи FFN або увагу окремо) та глибину охоплення, для пошуку оптимального співвідношення між приростом якості та витратами на навчання.

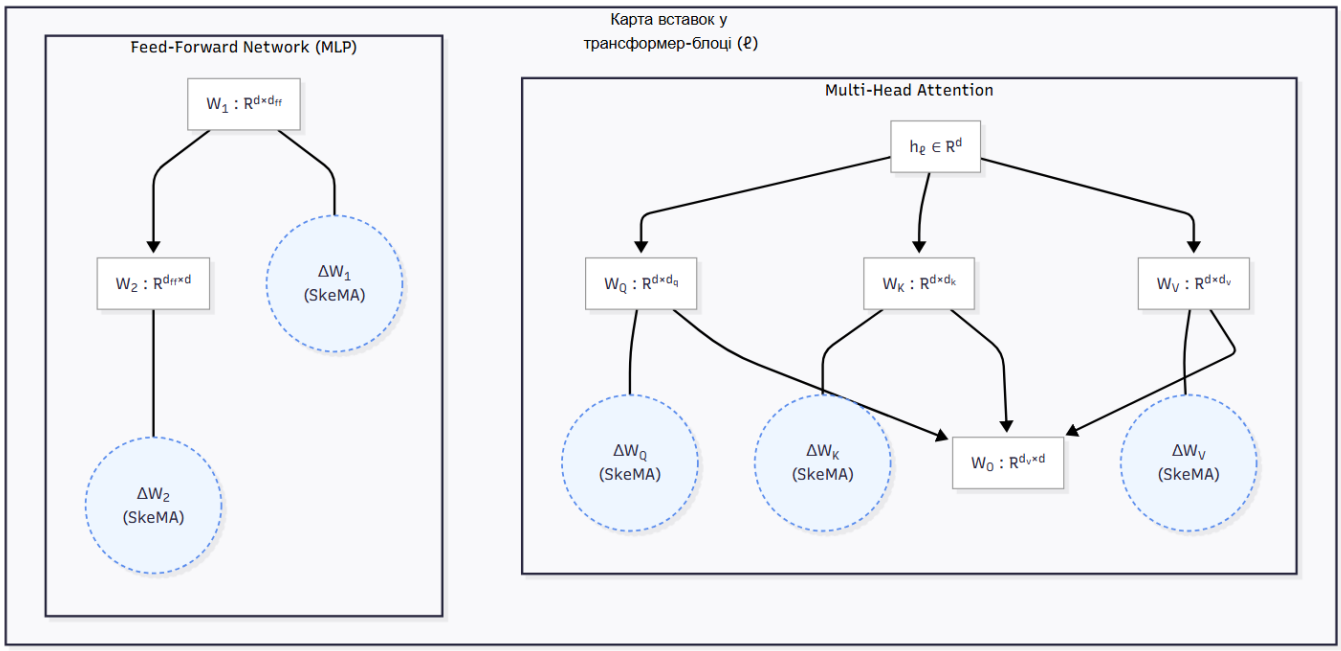


Рисунок 2.3 – Точки інтеграції оновлень SkeMA в архітектурі трансформера

Для практичного застосування методу SkeMA необхідно визначити оптимальну конфігурацію адаптера, що передбачає вибір точок інтеграції та гіперпараметрів. Логіка вибору конфігурації зазвичай починається з визначення місць вставки та цільового параметричного бюджету P. На основі бюджету обирається розмірність стислого простору k (згідно з (4)), що узгоджується з допустимими витратами на навчання та цільовою виразністю оновлення. Рекомендований практичний підхід полягає у виборі помірного значення k та комбінованої схеми вставки (увага + FFN) як початкової точки, з подальшим проведенням абляційних досліджень для оптимізації місткості та глибини

охоплення. Масштаб адаптера α може використовуватися як додатковий регуляторний параметр під час тренування; при злитті він враховується у фінальних вагах згідно з (3). Вибір місць вставки доцільно узгоджувати з природою цільового завдання. Для покращення моделювання довгих залежностей або адаптації патернів уваги пріоритет надається інтеграції у модуль багатоканальної уваги (Q/K/V). Якщо ж важливішою є локальна трансформація ознак або стиль генерації, акцентується мережа прямого поширення (FFN). Комбінований варіант (увага + FFN) зазвичай забезпечує стабільний приріст якості без необхідності точного апріорного знання про домінантний механізм адаптації, тому він рекомендований як стартовий. Глибина охоплення шарів обирається з урахуванням наявних ресурсів: повний охват є найвитратнішим, але забезпечує найбільш рівномірний вплив; адаптація лише верхніх L_{top} шарів часто є ефективною для інструкційних та класифікаційних завдань; інтервальна вставка («кожен s-й шар») дозволяє зменшити вартість тренування з прийнятними втратами якості. Ці міркування узагальнено в таблиці 2.2, яка слугуватиме довідковим матеріалом при виборі конфігурації.

Таблиця 2.2 – Основні гіперпараметри SkeMA

Параметр	Призначення	Типові значення	Вплив
Розмірність стислого простору (k)	Керує місткістю оновлення і параметричним бюджетом	128; 256; 512	Зростання (k) підвищує виразність (бюджет ~ (k^2)); вибір корелює з цільовим (P)
Місця інтеграції	Де застосовується (ΔW) у блоці трансформера	Q/K/V; FFN; Q/K/V+FFN	«Attention+FFN» – збалансований старт, «Attention-only» – дешевше, «FFN-only» – посилює нелінійну частину (рис. 2.2).
Глибина охоплення шарів	Скільки шарів модифікується	Усі; верхні (L_{top}), кожен (s)-й	Компроміс «якість ↔ вартість»: інтервальна вставка знижує вартість тренування.
Ініціалізація ядра (M)	Початковий стан	Нульова; малоамплітудна випадкова	Нульова – обережний старт, випадкова – швидший вихід

	тренованого блоку		на сигнал ціною потенційного стрибка.
Оптимізатор / LR	Схема оновлення параметрів (M)	AdamW; косинусний/ступінчастий LR	Тренується лише (M); базові ваги та проєкції фіксовані.

Після визначення конфігурації ключовим є забезпечення стабільності та відтворюваності процесу. Фіксовані проєкції S_{in}, S_{out} повинні залишатися незмінними протягом тренування, що вимагає контролю (фіксація seed, збереження матриць та метаданих). Ініціалізацію компактного блоку M доцільно виконувати нульовою або малоамплітудною випадковою для мінімізації початкових зсувів поведінки моделі. Оптимізація проводиться виключно для параметрів M. Для забезпечення чисельної стабільності, прямий та зворотний проходи рекомендується виконувати у змішаній точності, а операції обчислення ΔW та злиття – у підвищеній точності (float32) з фіксованим порядком множення. Для гарантії еквівалентності після злиття проводиться санітарна перевірка збігу виходів моделі до та після операції на репрезентативній вибірці. Ці аспекти деталізовано у таблиці 2.3.

Таблиця 2.3 – Додаткові налаштування й відтворюваність

Параметр	Призначення	Типові значення	Вплив / примітки
Масштаб адаптера (α)	Регулює внесок (ΔW) під час навчання	1.0 (0.5–1.0)	Входить у злиття за (3); використовувати послідовно у всіх місцях вставки.
Обчислювальна точність (dtype)	Числова стабільність тренування/злиття	bf16/fp16 (train), fp32 (merge)	Обчислення (1) і злиття (3) доцільно виконувати у fp32.
Порядок множення при merge	Відтворюваність (ΔW)	$(M \cdot S_{\text{in}}) \rightarrow S_{\text{out}}^{T}(\cdot)$	Фіксувати один порядок множення для всіх експериментів.
Ініціалізація проєкцій / seed	Стабільність і відтворюваність	Фіксований seed; зберігати $(S_{\text{in}}, S_{\text{out}})$	Заборонено змінювати проєкції під час навчання.
Післязлиттєве квантування	Зменшення розміру/прискорення	int8 / int4 + калібрування	Застосовується після формування W'

Кінець таблиці 2.3

Санітарна перевірка	Перевірка еквівалентності merge	Порівняння виходів до/після merge	Допуск числової різниці фіксується; зберігати хеш W′
Експорт / середовище виконання	Підготовка до деплою	ONNX / TensorRT / TorchScript	Працює «як звичайна модель», оскільки адаптер відсутній у графі.

Таблиця 2.4 містить приклади готових робочих схем для різних сценаріїв обмежень. «Легка» конфігурація мінімізує витрати ресурсів і підходить для швидких експериментів. «Базова збалансована» є рекомендованою стартовою точкою для більшості задач. «Максимум якості» використовується, коли пріоритетом є найвища точність, а ресурси дозволяють тривале навчання.

Таблиця 2.4 – Типові конфігурації під різні ресурси

Сценарій	Конфігурація	Рекомендоване застосування
«Легка» (обмежені ресурси)	$(k = 128 - 256)$; FFN або верхні (L_{top}) шари	Швидкі пілоти, обмежені GPU/час.
«Базова збалансована»	$(k = 256 - 512)$; Q/K/V+FFN; верхні/усі шари	Базовий профіль для більшості задач.
«Максимум якості»	$(k \geq 512)$; Q/K/V+FFN; усі шари; тонке налаштування LR	Коли пріоритет – точність, ресурси дозволяють довгі запуски.

Також, слід окреслити типові помилки, яких варто уникати для забезпечення коректності методу. Неприпустимо змінювати фіксовані проєкції після початку тренування. Порядок множення при злитті має бути зафіксованим. Важливо не залишати адаптер у графі виконання після злиття, аби не «подвоїти» внесок, та коректно враховувати масштаб α при формуванні W'. Дотримання цих правил забезпечує, щоб інференс зі злитими вагами відповідав поведінці моделі з активним адаптером у межах числового допуску.

Отже, у підрозділі 2.3 було деталізовано практичний алгоритм застосування методу SkeMA, що охоплює послідовні етапи від підготовки фіксованих проєкцій

та ініціалізації компактного блоку до навчання, фінального злиття ваг та підготовки до розгортання. Окрім цього, було проаналізовано ключові конфігураційні параметри, такі як розмірність стислого простору k, місця інтеграції (рис. 2.2) та глибина охоплення шарів. На основі аналізу було сформульовано рекомендації та типові робочі схеми (табл. 2.2–2.4) для вибору оптимальної конфігурації залежно від наявних обчислювальних ресурсів та пріоритетів завдання.

2.4 Узгодження з обмеженнями розгортання

Практичне застосування методу SkeMA передбачає чіткий життєвий цикл артефактів, що забезпечує відтворюваність процесу та коректне розгортання моделі.

Після завершення етапу тонкого налаштування (навчання з активним адаптером) зберігається набір артефактів, що включає:

– параметри компактного параметричного блоку (M);

– фіксовані проєкції $(S_{\text{in}}, S_{\text{out}})$;

– конфігураційні метадані (вибрані точки інтеграції, розмірність k, коефіцієнт α, типи даних, seed генератора проєкцій).

Після завершення навчання виконується одноразова офлайнова операція злиття, за результатом якої формуються злиті ваги W'(див. (3)) – єдиний прод-артефакт для інференсу. Паралельно фіксуються метадані відтворюваності: версія/commit, формат числової точності (dtype), порядок множення під час обчислення (1), дата/час, обчислювальний пристрій та контрольні суми файлів. Ключі параметрів M і посилання на $S_{\text{in}}, S_{\text{out}}$ вилучаються зі state_dict, щоб унеможливити повторне застосування оновлення під час інференсу. Числова еквівалентність «до/після злиття» перевіряється на фіксованій валідаційній підвибірці: відхилення логітів у межах узгодженого допуску (atol/rtol), стабільність перплексії та latency. За потреби виконується післязлиттєве квантування W'(int8/int4) з коротким калібруванням, та у метаданих зберігаються метод, розмір

і склад калібрувальної підвибірки. Завершальним кроком є експорт у цільовий формат виконання (ONNX/TensorRT/TorchScript) та формування цілісного пакета розгортання з усіма метаданими. Узагальнену схему етапів навчання, злиття та формування артефактів наведено на рисунку 2.4.

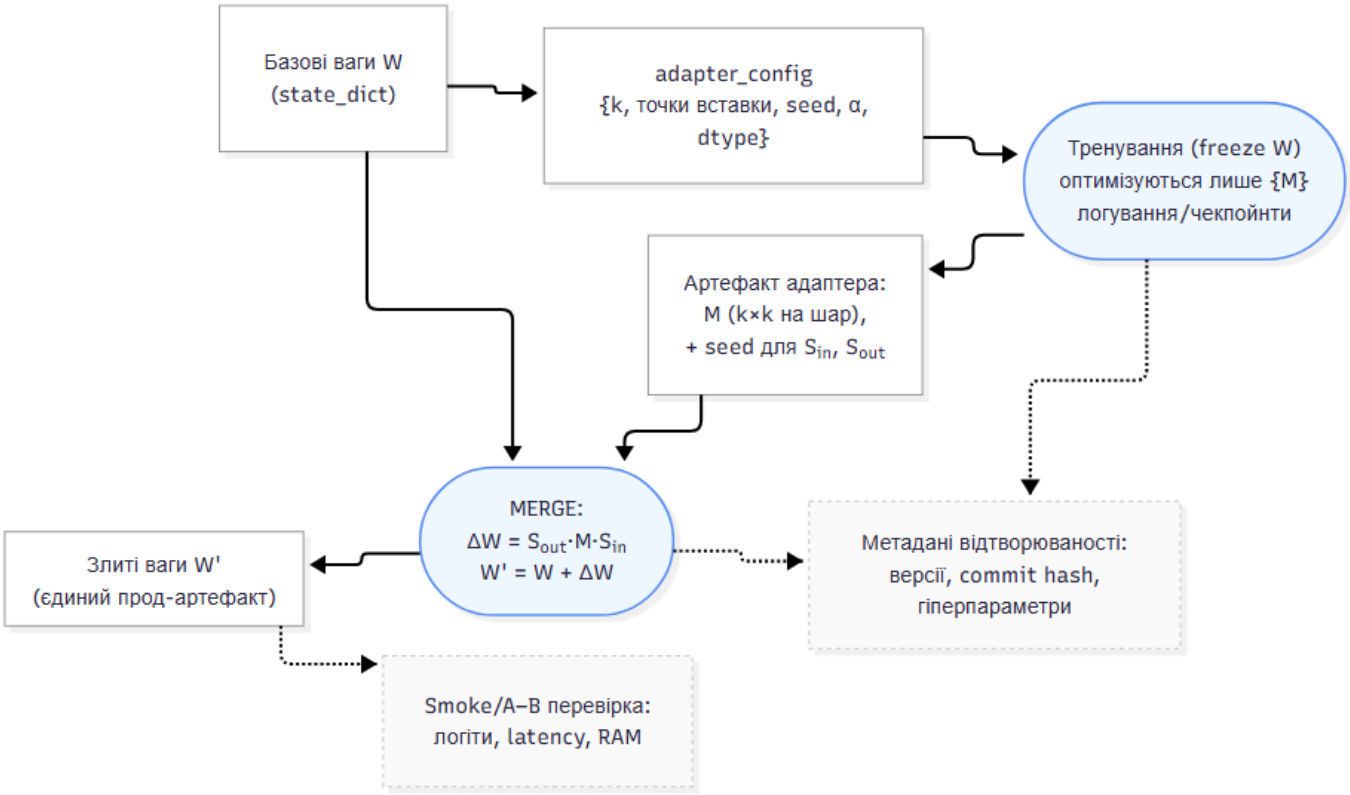


Рисунок 2.4 – Пайплайн артефактів SkeMA (фази навчання та злиття)

На етапі тренування (де W заморожені) оптимізується лише компактний блок M, у результаті чого утворюється артефакт адаптера (параметри M та seed проєкцій). Далі виконується офлайнова операція злиття (MERGE): обчислюється адитивне оновлення ΔW (1) та формуються фінальні злиті ваги W' (3). Після формування W' ключі параметрів M і посилання на S_{in}, S_{out} вилучаються зі state_dict, щоб унеможливити подвійне застосування оновлення. Процес супроводжується збором метаданих відтворюваності та санітарною перевіркою еквівалентності. Узагальнення наведено в таблиці 2.5.

Після завершення навчання формується фінальний артефакт, призначений для інференсу, злиті ваги W'. Для забезпечення відтворюваності, цей артефакт

супроводжується маніфестом конфігурації (що фіксує k, точки інтеграції, α, dtype, seed та характеристики проєкцій S_{in}, S_{out}) і журналом злиття (що документує середовище, порядок множення у та контрольні суми). Коректність операції верифікується звітом про еквівалентність, який фіксує результати порівняння виходів моделі до та після злиття на фіксованій підвибірці у межах заданого числового допуску.

Таблиця 2.5 – Мінімальний пакет відтворюваності для розгортання SkeMA

Компонент	Обов’язковий зміст
Злиті ваги (W')	Файл ваг після злиття за (3), вказаний формат числової точності, контрольна сума SHA-256.
Маніфест конфігурації	Розмірність стислого простору (k), обрані точки інтеграції, значення масштабу (α) (за потреби), формати даних для навчання і для злиття, значення seed, розмірності або ідентифікатор проєкцій ($S_{\text{in}}, S_{\text{out}}$).
Журнал злиття	Дата і час, обчислювальний пристрій, порядок множення у формулі (1), формат числової точності під час злиття, версії інструментів і контрольний commit, контрольна сума SHA-256 для (W').
Звіт про еквівалентність	Опис фіксованої валідаційної підвибірки, перелік метрик та погоджені допуски, підсумок порівняння «до/після злиття».
Пакет квантування (за потреби)	Обрана схема, параметри та кроки калібрування, ідентифікатор калібрувальної підвибірки, контрольна сума SHA-256 зквантованих ваг.
Маніфест (за потреби)	Цільовий формат і середовище виконання, номери версій та opset, відтворювана команда експорту.

У разі подальшої оптимізації (наприклад, післязлиттєвого квантування) додається відповідний пакет метаданих (метод, параметри калібрування, контрольна сума квантованих ваг). Для розгортання формується експортний маніфест, що визначає цільовий формат та параметри середовища виконання.

2.5 Висновки 2-го розділу

У другому розділі було представлено, обґрунтовано та формалізовано запропонований метод параметрично ефективного тонкого налаштування SkeMA.

У підрозділі 2.1 сформульовано концепцію методу, яка ґрунтується на декомпозиції оновлення на тренований компактний блок та фіксовані проєкції. Показано ключову перевагу – повне злиття ваг, що забезпечує нульове додаткове навантаження на інференсі. Визначено місце методу в класифікації PEFT та окреслено його очікувані переваги й межі застосовності.

У підрозділі 2.2 наведено формальну модель SkeMA. Визначено математичний апарат адитивного оновлення (1), деталізовано процес навчання ((2), рис. 2.1) та описано механізм фінального офлайнового злиття ((3), рис. 2.2). На основі цієї моделі доведено дві ключові властивості методу:

– точність злиття, що гарантує алгебраїчну еквівалентність обчислень до та після злиття, підтверджуючи нульове інференс-навантаження;

– керовану місткість (бюджет параметрів k^2 згідно з (4)) та верхню межу рангу оновлення, що безпосередньо контролюється параметром k.

Також у цьому підрозділі проведено теоретичне порівняння з методом LoRA (5). Обґрунтовано та продемонстровано (табл. 2.1), що за однакового параметричного бюджету SkeMA забезпечує потенційно вищу виразну здатність (більшу верхню межу рангу), особливо для великих лінійних шарів.

У підрозділі 2.3 розроблено детальний практичний алгоритм застосування методу, що охоплює 7 послідовних етапів: від підготовки проєкцій до фінальної санітарної перевірки. Проаналізовано ключові конфігураційні параметри, такі як вибір точок інтеграції (рис. 2.3) та глибина охоплення шарів. Цей аналіз узагальнено у вигляді практичних рекомендацій, довідкових таблиць гіперпараметрів (табл. 2.2 – 2.3) та типових схем застосування під різні обчислювальні ресурси (табл. 2.4).

У підрозділі 2.4 визначено пайплайн роботи з артефактами та узгодження методу з інженерними практиками MLOps. Візуалізовано життєвий цикл артефактів (рис. 2.4), що чітко розділяє етапи навчання та злиття. Формалізовано мінімальний пакет супровідної документації (табл. 2.5), який включає маніфест конфігурації, журнал злиття та звіт про еквівалентність, що є необхідним для забезпечення відтворюваності та надійного розгортання моделі.

3 АРХІТЕКТУРА ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Вимоги та архітектурний дизайн

У цьому підрозділі конкретизовано архітектурні рішення, що забезпечують інтеграцію адаптера SkeMA як drop-in доповнення до стандартних лінійних шарів трансформера та нульове навантаження на інференсі завдяки операції злиття. Обчислювальна логіка адаптера відповідає моделі з підрозділу 2.2 із посиланнями на (1)–(3). Фіксовані проєкції залишаються незмінними протягом навчання, треновані параметри зосереджені в компактному блоці, а кінцевим артефактом для розгортання є злита матриця ваг.

Архітектура адаптера SkeMA розроблена відповідно до двох ключових вимог: забезпечення сумісності зі стандартними шарами трансформера (drop-in integration) та підтримки операції злиття, що унеможливлює інференс-навантаження. Практична реалізація SkeMA передбачає його впровадження як drop-in модуля для кожного лінійного шару (наприклад, nn.Linear у PyTorch), що дозволяє інтегрувати адаптер без зміни архітектури чи інтерфейсів вихідної моделі.

Адаптер додає лише невеликий набір параметрів, що навчаються, до кожного обраного шару – подібно до того, як LoRA впроваджує низькорангові матриці – залишаючи оригінальну матрицю ваг W замороженою. Такий підхід дозволяє зберегти попередньо навчені знання, інкапсульовані у W, та обмежити процес навчання лише легковагим, параметроефективним адаптером. Як було обговорено у розділі 2, методи на кшталт LoRA додають адаптивну пертурбацію до W замість оновлення самої матриці. SkeMA наслідує цю стратегію: базові ваги залишаються незмінними, а адаптер навчає завдання-специфічне ΔW, яке коригує вихід шару без модифікації його вихідних параметрів.

У кожному лінійному шарі, доповненому SkeMA, обчислення прямого проходу (forward pass) модифікується для врахування внеску адаптера. Нехай $W \in R^{m \times d}$ – це оригінальна матриця ваг шару, яка залишається замороженою під час

тонкого налаштування. SkeMA вводить треноване адитивне оновлення ΔW таким чином, що ефективна матриця ваг під час прямого проходу стає $W + \Delta W$.

Замість параметризації ΔW як повнорозмірної матриці (що було б обчислювально обтяжливо), SkeMA використовує підхід, заснований на матричному скетчуванні, для компактного кодування цього оновлення [31]. Архітектурно, адаптер реалізує трьохетапний механізм у прямому проході, як це було визначено рівнянням (2):

– стиснення (compression): вхідний вектор x проєктується у стислий підпростір за допомогою фіксованої проєкції S_{in};

– обробка у підпросторі (subspace processing): стиснений вектор множиться на єдиний тренований компактний блок M;

– розгортання (expansion): результат повертається у вихідний простір за допомогою фіксованої проєкції S_{out}^{T} та адитивно додається до виходу базового шару $W\,x$.

На першому етапі вхідний вектор $x \in R^{d}$ проєктується у підпростір меншої розмірності. Це досягається шляхом застосування фіксованої проєкційної матриці $S_{\text{in}} \in R^{k \times d}$, яка зменшує розмірність ознакового простору з d до k, де $k \ll d$. Концептуально, цей крок виконує скетчинг (sketching) вхідного простору, що дозволяє зберегти найбільш релевантну інформацію у k-вимірному представленні (скетчі), одночасно знижуючи обчислювальні витрати та вимоги до пам'яті.

На другому етапі треновані параметри адаптера застосовуються у стислому k-вимірному просторі. Стиснений вектор, отриманий на попередньому кроці (результат $S_{\text{in}}x$), множиться на компактний параметричний блок $M \in R^{k \times k}$. Ця матриця M є єдиним компонентом, що оптимізується під час тонкого налаштування, і в ній зосереджені всі треновані параметри (k^2). Дана операція є ядром трансформації, яку реалізує SkeMA, і саме вона вивчає завдання-специфічне оновлення ΔW у компактному вигляді.

На завершальному етапі, результат обробки у стислому підпросторі (вектор, отриманий на кроці 2) відображається назад у вихідний m-вимірний простір. Це

досягається шляхом множення на фіксовану проєкцію розгортання $S_{\text{out}}^{T} \in R^{m \times k}$. Отриманий вектор адитивно додається до виходу базового шару $W\,x$, формуючи кінцевий вихід y згідно з рівнянням (2). Ця трьохетапна процедура (стиснення-обробка-розгортання) реалізує функціонал drop-in модуля під час навчання, вносячи лише незначне обчислювальне навантаження.

Ключовою архітектурною властивістю методу SkeMA є його придатність до повного злиття (merge-friendliness), що забезпечує високу ефективність на етапі інференсу. Оскільки адаптер формує адитивне оновлення ваг ΔW (визначене у (1)), після завершення навчання виконується одноразова офлайнова операція злиття для отримання оновленої матриці ваг, згідно з рівнянням (3). В результаті формується автономна модель зі злитими вагами W', яка вже інкапсулює необхідні завдання-специфічні коригування.

На етапі інференсу обчислення виконуються з використанням стандартного прямого проходу $y = W'x$. Потреба в окремому обчисленні внеску адаптера (як у (2)) відпадає. Це означає, що SkeMA не створює жодного додаткового обчислювального навантаження, затримки (latency) чи споживання пам’яті під час експлуатації; модель функціонує з такою ж ефективністю, як і повністю донавчена модель.

Ефект адаптера є структурно прозорим після злиття: він не змінює архітектуру моделі та не вимагає модифікації графу обчислень. Як доведено у твердженні 1 (п. 2.2), ця операція злиття є алгебраїчно точною, тому при переході від $W + \Delta W$ до W' не відбувається втрати точності моделі. Таким чином, SkeMA дозволяє використовувати переваги параметроефективного підходу на етапі тренування (економія ресурсів) і водночас повністю усунути адаптер на етапі інференсу, досягаючи максимальної ефективності розгортання.

Функціональні вимоги:

– адаптер підключається до стандартних лінійних шарів без зміни їхнього зовнішнього інтерфейсу (drop-in інтеграція);

– конфігурація місць вставки задається явно: модуль уваги (Q, K, V), лінійні шари FFN, або комбіновано; глибина охоплення шарів – усі, верхні, або інтервально;

– параметр стискання задається як гіперпараметр; за потреби використовується масштаб адаптера, формат числової точності для навчання та злиття узгоджується наперед;

– під час навчання базові ваги та проєкції фіксовані, оновлюються лише параметри компактного блоку;

– після навчання виконується одноразова операція злиття згідно з (3) із фіксованим порядком множення, узгодженим у п. 2.2;

– після злиття ключі параметрів адаптера та посилання на проєкції вилучаються зі сховища параметрів моделі, щоб уникнути повторного застосування оновлення;

– еквівалентність «до/після злиття» підтверджується короткою перевіркою на фіксованій підвибірці (у межах узгодженого числового допуску);

– модель зі злитими вагами експортується до цільового середовища виконання (напр., ONNX, TensorRT, TorchScript) без додаткових залежностей від адаптера.

Нефункціональні вимоги:

– інференс після злиття не має додаткових обчислювальних витрат та не погіршує затримку в порівнянні з базовою моделлю;

– споживання пам’яті на інференсі не перевищує споживання базової моделі; під час навчання воно масштабується з урахуванням вибраного параметра стискання;

– відтворюваність забезпечується фіксацією початкових станів, збереженням проєкцій і маніфестів конфігурації, а також фіксованим порядком множення при злитті (п. 2.4);

– портативність гарантується відсутністю адаптера у графі інференсу та наявністю підтримуваних форматів експорту;

– спостережуваність забезпечується журналами тренування та злиття, збереженням метрик і контрольних сум;

– масштабованість підтримується у сценаріях багатопроцесного навчання з фіксованою конфігурацією та відключеними градієнтами для базових ваг;

– керування артефактами здійснюється з контролем версій і цілісності, що дозволяє аудит і відкатування у разі невдалої перевірки еквівалентності.

Програмна архітектура рішення SkeMA організована у вигляді кількох компонентів, кожен з яких відповідає за окремий аспект конвеєра тонкого налаштування та подальшого розгортання. Загальну карту програмних компонентів і їхні зони відповідальності подано на рисунку 3.1.

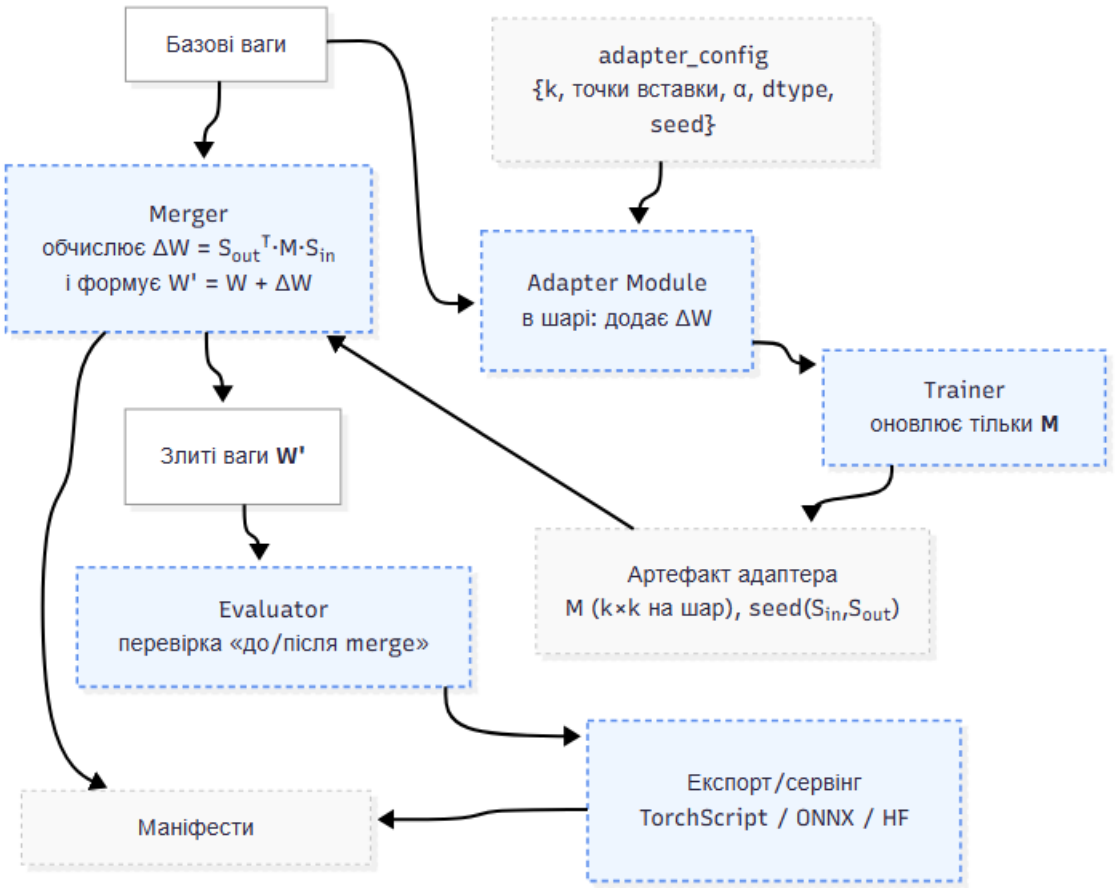


Рисунок 3.1 – Карта компонентів реалізації SkeMA

Адаптерний модуль (Adapter Module) – цей компонент є ключовим шаром, що реалізує логіку SkeMA в межах кожного цільового лінійного шару (наприклад, nn.Linear). Він обгортає стандартний лінійний шар, доповнюючи його механізмом

оновлення ваг. Модуль зберігає тренований компактний параметричний блок (M) для даного шару, а також фіксовані проєкції (S_{in}, S_{out}). Під час прямого проходу модуль обчислює додатковий внесок $\Delta W\, x$ (2) та адитивно додає його до виходу базового шару. Зовні шар поводиться ідентично до вихідного (приймаючи x і повертаючи y), тому адаптер є прозорим для інтерфейсу моделі. Внутрішньо він забезпечує виконання послідовності «стиснення–обробка–розгортання» ($S_{\text{in}} \rightarrow M \rightarrow S_{\text{out}}^{T}$). Модуль розроблено як легковаговий: оскільки тренований блок M є значно меншим за базові ваги W, адаптер додає лише незначну кількість параметрів. Він також надає інтерфейс для злиття своїх тренованих ваг, зокрема, може експортувати фінальне оновлення ΔW (1), коли модуль злиття (Merger) потребує інтеграції цього оновлення у W. Таким чином, адаптерний модуль є автономною одиницею, що інкапсулює обчислення SkeMA, забезпечуючи drop-in сумісність.

Модуль тренування (Trainer) – цей компонент керує процесом тонкого налаштування моделі з інтегрованими адаптерами SkeMA. Його основна функція – організація оптимізації параметрів адаптера при одночасному заморожуванні ваг попередньо навченої базової моделі. Модуль тренування конфігурує модель таким чином, що всі базові матриці W залишаються фіксованими, а градієнтний спуск оновлює виключно треновані компактні блоки M (та будь-які пов'язані з ними стани адаптера). Використовуються стандартні цикли навчання (прямий прохід, обчислення функції втрат, зворотне поширення помилки), однак оновлення параметрів обмежуються лише модулями SkeMA. Компонент також відповідає за ініціалізацію та контроль гіперпараметрів, специфічних для SkeMA. Зокрема, треновані ваги адаптера M ініціалізуються таким чином, щоб початкове оновлення ΔW було нульовим або нехтуваним (наприклад, ініціалізація M нульовою матрицею, аналогічно до ініціалізації матриці B у LoRA). Це забезпечує плавний початок тонкого налаштування, де адаптер не вносить змін, доки не почне вивчати необхідні корекції. Модуль тренування може застосовувати окремі розклади швидкості навчання для параметрів адаптера, оскільки вони часто потребують інших режимів оптимізації, ніж базова модель. Протягом навчання компонент

відстежує показники продуктивності (наприклад, на валідаційному наборі) для оцінки ефективності адаптації та визначення моменту конвергенції.

Модуль злиття (Merger) – цей компонент є утилітою, що виконує післятренувальну інтеграцію ваг адаптера в базову модель. Після завершення тонкого налаштування, модуль злиття ітеративно проходить по кожному адаптованому шару та отримує з нього тренований компактний блок M та фіксовані проєкції S_{in}, S_{out}. На основі цих компонентів він реконструює повне адитивне оновлення ΔW (1) для даного шару. Далі, це оновлення додається до оригінальної матриці ваг W, в результаті чого формується злита матриця (3). Ця нова матриця W' заміщує вихідну матрицю W у наборі параметрів шару. Після того, як цей процес повторюється для всіх адаптованих шарів, модулі адаптера SkeMA можуть бути вилучені з моделі, оскільки їхня функціональність вже повністю інкапсульована у злитих вагах. Це повертає архітектуру моделі до її стандартного вигляду (тотожного вихідному трансформеру, але з оновленими вагами), що гарантує відсутність будь-якого додаткового навантаження на етапі інференсу. Таким чином, модуль злиття відіграє ключову роль у забезпеченні сумісності SkeMA з розгортанням у продуктивних середовищах.

Модуль оцінювання (Evaluator) – цей компонент відповідає за вимірювання продуктивності та валідацію моделі в межах конвеєра SkeMA. Він використовується як під час навчання (з активними адаптерами), так і після операції злиття (зі злитими вагами W'). Під час тренувального циклу модуль оцінювання може викликатися періодично (наприклад, наприкінці кожної епохи) для запуску моделі на відкладеному валідаційному наборі даних та обчислення цільових метрик (втрати, точність, F1 тощо). Оскільки базова модель заморожена, зворотний зв'язок від модуля оцінювання відображає ефективність оновлень ΔW, що вивчаються. Після завершення навчання та виконання злиття, модуль оцінювання запускається повторно на тестовому наборі з використанням злитої моделі. Цей крок є критично важливим для верифікації того, що продуктивність злитої моделі є ідентичною (в межах числового допуску) продуктивності моделі з активним адаптером, підтверджуючи відсутність втрати точності під час злиття.

Таким чином, модуль оцінювання підтверджує, що SkeMA відповідає вимогам не лише параметричної ефективності, але й збереження або покращення якості вирішення цільового завдання.

Сукупно, описані компоненти (Адаптерний модуль, Модуль тренування, Модуль злиття, Модуль оцінювання) та артефакти реалізують архітектуру SkeMA. Вони забезпечують визначений конвеєр: модуль тренування оптимізує адаптери, модуль оцінювання моніторить їхній вплив, а модуль злиття консолідує їхній ефект у фінальний артефакт. Це дозволяє здійснювати ефективне тонке налаштування трансформерів, дотримуючись вимог легкої інтеграції та ефективності розгортання. Послідовність викликів під час навчання з активним адаптером і під час офлайнового злиття подано на рисунку 3.2.

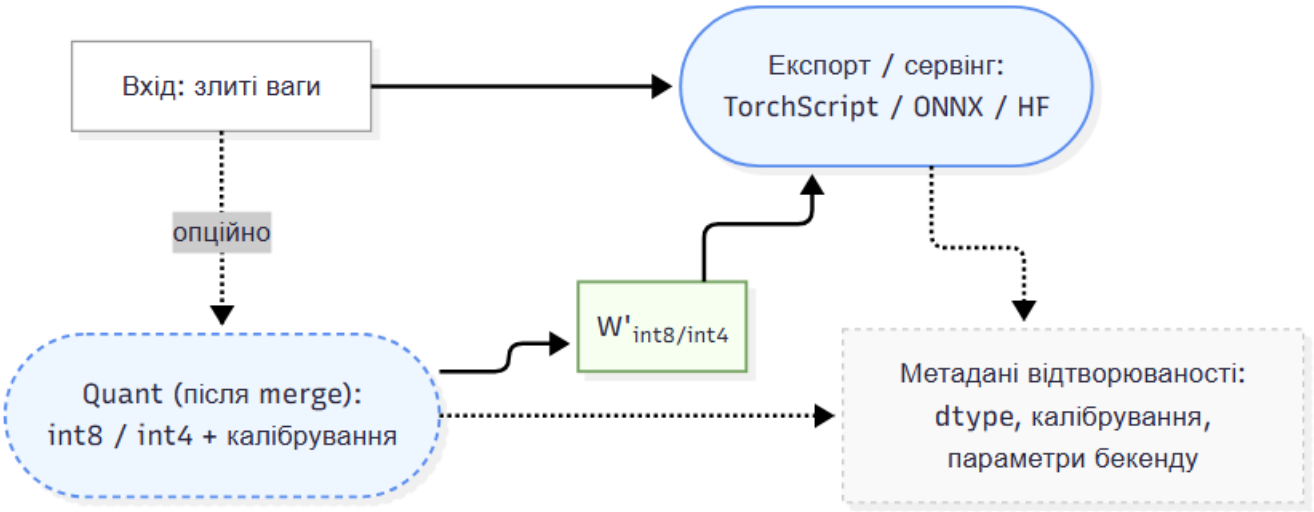


Рисунок 3.2 – Взаємодія компонентів під час навчання

3.2 Модель даних та артефактів

Процес параметроефективного тонкого налаштування з використанням методу SkeMA генерує сукупність артефактів, що відображають різні етапи життєвого циклу моделі. Класифікація артефактів (рис. 3.3):

– злиті ваги W': фінальний артефакт, призначений для розгортання, отриманий шляхом інтеграції адитивного оновлення ΔW з базовими вагами W (3);

– артефакт адаптера: тренований компактний блок M разом із фіксованими проєкціями S_{in}, S_{out} або їхнім seed для відтворення; потрібен лише для побудови ΔW та подальших експериментів;

– маніфест конфігурації: описує k, точки інтеграції, за потреби масштаб α, формати даних, seed; забезпечує коректне відтворення адаптера;

– журнал злиття (merge-log): фіксує параметри операції злиття: дата і час, пристрій, точність обчислень під час merge, порядок множення, контрольну суму W';

– звіт еквівалентності – результати перевірки «до/після merge» на фіксованій вибірці з узгодженими допусками та висновком про проходження.

Рисунок 3.3 – Логічна модель зв’язків

Злиті ваги (Merged Weights) – цей артефакт являє собою фінальні ваги моделі W', отримані шляхом інтеграції адитивного оновлення ΔW у базові ваги W. Після завершення процесу тонкого налаштування, параметри адаптера, що навчаються, використовуються для обчислення ΔW (1), яке потім зливається з W згідно з (3). В результаті формується єдина, автономна модель зі злитими вагами W', яка більше не потребує активного модуля адаптера для функціонування; адаптер вилучається з графу обчислень. Продуктивність фінальної моделі W' є алгебраїчно

еквівалентною продуктивності системи «базова модель + активний адаптер» (як доведено у твердженні 1, п. 2.2). Ключовою перевагою є те, що використання виключно злитих ваг на етапі інференсу усуває будь-яке додаткове обчислювальне навантаження, що значно спрощує процес розгортання моделі.

Артефакти адаптера (Adapter Artifacts) – цей артефакт являє собою набір параметрів, що зберігається окремо від базових ваг моделі W і необхідний для реконструкції адитивного оновлення ΔW (1). У контексті SkeMA, цей набір включає тренований компактний параметричний блок M (по одному на кожен адаптований шар) та фіксовані проєкції S_{in}, S_{out} (або seed для їх відтворюваної генерації).

Оскільки базова модель W залишається замороженою, а треновані компоненти M є надзвичайно компактними (лише k^2 параметрів на шар), ці артефакти забезпечують високу модульність. Вони дозволяють одній незмінній базовій моделі динамічно підключати різні артефакти адаптера для вирішення специфічних завдань. Такий підхід спрощує обмін та версіонування адаптацій: замість передачі повної, багатогігабайтної моделі, достатньо передати лише легковагий файл артефакту адаптера.

Водночас, ці артефакти є необхідними вхідними даними для модуля злиття (Merger), який використовує їх для обчислення фінальних злитих ваг W', призначених для розгортання з нульовим інференс-навантаженням.

Метадані конфігурації (Adapter Metadata) – цей артефакт інкапсулює повну інформацію про конфігурацію та контекст тренування адаптера. До метаданих належать ключові гіперпараметри, що визначають архітектуру, такі як розмірність стислого простору k (або ранг r для LoRA) та коефіцієнт масштабування α. Окрім цього, вони містять точну специфікацію точок інтеграції – перелік модулів базової моделі (наприклад, конкретні шари уваги чи FFN, як показано на рис. 2.2), до яких було застосовано адаптер, а також глибину охоплення (наприклад, всі шари, верхні L_{top} шарів, або інтервальна вставка).

Важливою складовою метаданих є початкове значення (seed) генератора випадкових чисел, яке використовувалося для ініціалізації параметрів (та/або

фіксованих проєкцій S_{in}, S_{out} у випадку SkeMA). Наявність цих метаданих є необхідною умовою для коректного відтворення або застосування артефакту адаптера, оскільки вони дозволяють інстанціювати модуль адаптера з ідентичною архітектурою перед завантаженням тренованих ваг M. Сучасні програмні каркаси PEFT зазвичай зберігають ці конфігураційні дані разом з артефактом адаптера, що забезпечує детермінованість та повну відтворюваність експерименту.

Журнали трекінгу експериментів (Logs and Experiment Tracking) – цей артефакт є сукупністю даних, що генеруються паралельно з процесом тонкого налаштування та фіксують його динаміку. До цих даних належать журнали (логи), що містять значення функції втрат та цільових метрик якості на тренувальному й валідаційному наборах даних, тривалість епох, а також повну конфігурацію гіперпараметрів запуску.

Ці дані не є частиною фінальної моделі, але є критично важливими для аналізу, валідації та відтворюваності дослідження. Вони зберігаються за допомогою спеціалізованих систем трекінгу, таких як TensorBoard або Weights & Biases (W&B). Такі інструменти забезпечують візуалізацію кривих навчання (наприклад, динаміку функції втрат чи точності) та полегшують порівняльний аналіз різних експериментальних запусків. Окрім фіксації метрик, системи трекінгу прив'язують до експерименту метадані середовища (версії коду, даних, конфігурацій), що спрощує подальшу реплікацію результатів.

Спосіб використання перелічених артефактів залежить від цільового сценарію – розгортання у продуктивному середовищі або поширення результатів дослідження.

Для промислової експлуатації (production deployment), як правило, достатньо мати лише злиті ваги W'. Цей артефакт є самодостатнім, оскільки він інкапсулює як базові знання моделі, так і специфічні знання, набуті під час адаптації. Такий підхід є оптимальним для розгортання, оскільки він не вимагає підключення додаткових компонентів або модифікації логіки інференсу під специфіку адаптера. Злита модель розгортається безпосередньо і функціонує з тією ж обчислювальною

ефективністю, що й вихідна модель, оскільки додаткове навантаження від адаптера повністю усунуте.

Водночас артефакти адаптера (компактний блок M, проєкції S_{in}, S_{out}) у поєднанні з метаданими конфігурації є ефективним засобом для поширення результатів навчання або для реалізації багатозадачних сценаріїв. Замість передачі повної, багатогігабайтної моделі, достатньо поширювати лише легковагий файл адаптера та його конфігурацію. Користувачі, які мають ідентичну базову модель, можуть завантажити цей адаптер та відтворити адаптовану модель локально. Це суттєво скорочує обсяг переданих даних і дозволяє динамічно перемикатися між різними адаптаціями на основі єдиного базового чекпойнту. У випадку SkeMA, наявність метаданих (точок інтеграції, k, seed) гарантує коректне застосування адаптера та відтворення результатів, досягнутих під час навчання.

Збережені артефакти відіграють ключову роль у відтворюваності процесу адаптації та верифікації коректності отриманих результатів. Для підтвердження того, що операція злиття ваг була виконана коректно, проводиться санітарна перевірка еквівалентності. Ця процедура передбачає порівняння виходів фінальної злитої моделі W' та виходів системи «базова модель W + активний адаптер ΔW» (згідно з (2)) на ідентичному валідаційному наборі даних. Якщо адаптер інтегровано без числових втрат, обидва підходи повинні давати ідентичні результати прогнозування в межах заданого числового допуску, що підтверджує алгебраїчну еквівалентність, доведену у твердженні 1. Для верифікації відповідності також звіряються показники якості: фінальна злита модель повинна демонструвати ті самі метрики на валідаційних даних, що й були зафіксовані в журналах трекінгу на останньому етапі навчання адаптера. Деталізовані журнали трекінгу експериментів, що включають метадані конфігурації, дозволяють, за необхідності, повторно запустити процес навчання SkeMA та отримати співставні результати, що є фундаментальною вимогою наукової відтворюваності. Таким чином, повний набір артефактів забезпечує практичну зручність застосування адаптованої моделі, прозорість та надійність з точки зору верифікації результатів дослідження.

3.3 Технологічний стек та інженерні рішення

Для програмної реалізації адаптера SkeMA було обрано мову Python та бібліотеку глибокого навчання PyTorch. Цей вибір обґрунтований високою ефективністю PyTorch у виконанні тензорних обчислень на GPU та наявністю вбудованого механізму автоматичного диференціювання (autograd).

Основним компонентом для реалізації лінійних перетворень в архітектурі PyTorch є модуль nn.Linear, який імплементує повнозв'язний шар. При його інстанціюванні, nn.Linear(in_features, out_features), ініціалізується матриця ваг $W \in R^{m \times d}$ та, за замовчуванням, вектор зсуву $b \in R^m$, де $d = \text{in\_features}$ та $m = \text{out\_features}$.

Під час прямого проходу (forward) шар виконує афінне перетворення вхідного вектора $x \in R^d$ згідно з рівнянням: $y = Wx + b$, де $y \in R^m$ – вихідний вектор. Варто зазначити, що у внутрішній реалізації PyTorch, для узгодження з операціями над батчами (де вхід x має розмірність $[B \times d]$), обчислення виконується як $y = xW^T + b$. Таким чином, модуль nn.Linear(128, 64) реалізує лінійне відображення 128-вимірного вхідного простору у 64-вимірний вихідний.

Параметри модуля nn.Linear, зокрема матриця ваг W та вектор зсуву b, за замовчуванням є параметрами, що навчаються (trainable parameters). Це означає, що їх атрибут requires_grad має значення True, що дозволяє обчислювати для них градієнти під час зворотного поширення помилки та оновлювати їх значення за допомогою оптимізатора. Однак, у контексті параметроефективного налаштування, зокрема методу SkeMA, фундаментальною вимогою є збереження знань попередньо навченої моделі. Для цього базова матриця ваг W (та, за наявності, b) кожного адаптованого шару повинна залишатися незмінною (замороженою) під час донавчання. Це досягається шляхом заморожування параметрів – вимкнення обчислення градієнтів для них. У PyTorch ця операція здійснюється шляхом встановлення атрибута requires_grad = False для відповідних тензорів.

Внаслідок встановлення requires_grad = False для параметрів базового шару, відповідні тензори виключаються з обчислювального графу, що використовується для зворотного поширення помилки. Це гарантує, що оптимізатор не буде оновлювати їхні значення, ефективно заморожуючи їх та зберігаючи таким чином попередньо навчені знання моделі. Окрім цього, такий підхід забезпечує економію обчислювальних ресурсів: для зафіксованих шарів не зберігаються проміжні активації, необхідні для backpropagation, що зменшує пікове споживання пам’яті та прискорює процес навчання. У програмній реалізації SkeMAAdapter цей принцип застосовується безпосередньо в конструкторі класу. Модуль nn.Linear, що підлягає адаптації (base_linear), передається в конструктор і негайно заморожується шляхом виклику методу base_linear.requires_grad_(False). Ця операція рекурсивно встановлює requires_grad=False для тензорів ваг та зсуву базового шару. Таким чином, базовий шар фактично переводиться у режим інференсу, його параметри залишаються сталими, а всі обчислення градієнтів під час тонкого налаштування виконуються виключно для параметрів новоствореного адаптерного компонента (тобто компактного параметричного блоку M).

Архітектурно SkeMAAdapter реалізовано як drop-in (вбудовуваний) модуль, що функціонує як обгортка (wrapper) над існуючим лінійним шаром (nn.Linear). Такий підхід дозволяє автоматизовано заміщувати цільові шари у вихідній архітектурі моделі без модифікації її загальної логіки чи зовнішніх інтерфейсів. Адаптерний модуль інкапсулює посилання на оригінальний (базовий) шар, який заморожується згідно з підрозділом 3.3, та додає власні компоненти: фіксовані проєкції та параметри, що навчаються. З точки зору графу обчислень, модуль SkeMAAdapter повністю імітує сигнатуру nn.Linear: він приймає вхідний тензор і повертає вихідний тензор ідентичної розмірності. Однак, його вихід формується з урахуванням адитивного внеску $\Delta W\ x$, обчисленого за рівнянням (2). Ця прозора інтеграція спрощує застосування методу, оскільки підготовка моделі до параметроефективного тренування зводиться лише до операції заміни відповідних шарів.

Основними компонентами програмної реалізації SkeMAAdapter є три матриці: дві фіксовані проєкції (S_{in}, S_{out}) та тренований компактний параметричний блок (M). Ці компоненти реалізовані як атрибути класу, причому для M використовується стандартний nn.Parameter, тоді як для S_{in} та S_{out} застосовано механізм буферів PyTorch. Використання буферів, зареєстрованих методом register_buffer, є доцільним, оскільки вони коректно серіалізуються разом зі станом моделі (state_dict) та автоматично переміщуються між обчислювальними пристроями (CPU/GPU). Водночас буфери не включаються до списку параметрів, що повертається методом parameters(), і, відповідно, не потрапляють до оптимізатора, що гарантує відсутність обчислення градієнтів для них. Хоча альтернативним рішенням могло б бути визначення проєкцій як nn.Parameter з атрибутом requires_grad=False, використання register_buffer є семантично більш точним, оскільки явно підкреслює статус S_{in} та S_{out} як фіксованих, нетренованих компонентів протягом усього життєвого циклу адаптера.

Компоненти SkeMAAdapter включають фіксовану проєкцію стиснення $S_{\text{in}} \in R^{k\times d}$ та фіксовану проєкцію розгортання $S_{\text{out}} \in R^{k\times m}$. Ці дві матриці генеруються один раз при створенні адаптера (п. 3.3). Єдиним компонентом, параметри якого навчаються, є компактний параметричний блок $M \in R^{k\times k}$, ініціалізований як nn.Parameter з requires_grad=True. Саме елементи матриці M оновлюються оптимізатором під час тонкого налаштування. Така архітектура кардинально обмежує кількість параметрів, що навчаються, до $O(k^2)$. Це забезпечує суттєву параметричну економію, оскільки бюджет параметрів k^2 не залежить від розмірностей шару d та m, на відміну від методів типу LoRA, де кількість параметрів $P \approx O\big(r(d+m)\big)$ масштабується лінійно з розмірами шару.

Процес інтеграції адаптера SkeMA передбачає заміщення цільових лінійних шарів (nn.Linear) у попередньо навченій моделі model на екземпляри SkeMAAdapter. Для кожного шару, що підлягає адаптації, створюється відповідний модуль SkeMAAdapter. Наприклад, якщо model.fc є шаром nn.Linear(1024, 256), операція заміщення матиме вигляд: model.fc = SkeMAAdapter(model.fc, k=16). Конструктор SkeMAAdapter приймає існуючий шар, зчитує його вхідну та вихідну

розмірності, зберігає посилання на нього (наприклад, у self.base_linear) та негайно заморожує його параметри (W, b), встановлюючи requires_grad=False. Одночасно ініціалізуються компоненти SkeMA: фіксовані проєкції та тренований компактний блок. Після заміщення model.fc стає модулем-адаптером, що інкапсулює оригінальний заморожений шар та нові параметри, які підлягають навчанню.

Для інтеграції адаптерів у масштабні моделі, що містять велику кількість лінійних шарів, ручне заміщення, описане вище, є непрактичним. Тому застосовується автоматизований процес заміщення: виконується ітерація по всіх підмодулях моделі (наприклад, за допомогою model.named_modules()), і кожен модуль типу nn.Linear, що відповідає критеріям адаптації, програмно заміщується на новий екземпляр SkeMAAdapter з заданою конфігурацією (наприклад, k). Такий підхід дозволяє впровадити адаптери у глибокі архітектури трансформерів, не вимагаючи ручної модифікації вихідного коду моделі. Після завершення інтеграції адаптерів необхідно коректно налаштувати оптимізатор. Оскільки в процесі заміщення всі параметри базових шарів W були заморожені, набір параметрів, що підлягають навчанню, автоматично обмежується лише тренованими компактними блоками M (та, опційно, векторами зсуву bias, хоча зазвичай вони також фіксуються для консистентності). Таким чином, ініціалізація оптимізатора (наприклад, AdamW(model.parameters(), lr=...)) призведе до того, що він буде оперувати виключно параметрами M, що знаходяться всередині модулів SkeMAAdapter.

Процес тонкого налаштування моделі з інтегрованими адаптерами SkeMA є ідентичним до стандартного циклу навчання: він включає ітеративну подачу батчів даних, прямий прохід (model.forward()), обчислення функції втрат, зворотне поширення помилки (loss.backward()) та крок оптимізатора. Ключова відмінність полягає в тому, що градієнти обчислюються і оновлення застосовуються виключно до параметрів адаптерів, оскільки базові ваги залишаються замороженими. Мала кількість тренованих параметрів створює незначне обчислювальне навантаження порівняно з повномасштабним донавчанням. Окрім того, відсутність необхідності обчислювати та зберігати градієнти для заморожених ваг суттєво економить

пам’ять GPU. Такий підхід уможливлює ефективне тонке налаштування великих моделей навіть за умов обмежених апаратних ресурсів. Після завершення процесу донавчання, передбачено опціональний етап злиття (merging). Для кожного модуля SkeMAAdapter може бути викликаний метод merge(), який виконує обчислення фінального адитивного оновлення (1) та додає його до замороженої матриці ваг W. В результаті формується злита матриця (3), яка використовується для ініціалізації нового, еквівалентного шару nn.Linear. Вектор зсуву b при цьому копіюється з базового шару без змін. У результаті отримується злитий лінійний шар, що інкапсулює ефект адаптації, а сам модуль SkeMAAdapter вилучається з архітектури моделі.

Було формалізовано обчислення прямого проходу (forward pass) для шару, модифікованого адаптером SkeMA. Прийнято, що на вхід адаптованого шару подається вхідний вектор $x \in R^d$. Обчислення виконуються у двох паралельних гілках. Базовим (замороженим) шляхом обчислюється вихід (6):

$$y_{\text{bs}} = Wx + b \tag{6}$$

де $W \in R^{m \times d}$;

b – опціональний вектор зсуву.

Паралельно шляхом адаптера формується адитивний корегуючий вектор $\Delta y \in R^m$.Кінцевий вихід шару $y \in R^m$ було визначено як суму виходів (7):

$$y = y_{\text{bs}} + \Delta y \tag{7}$$

У модулі SkeMA корегуючий вектор Δy обчислюється через послідовність трьох операцій, що відповідає шляху адаптера в рівнянні (2). Нижче було деталізовано цей трьохетапний процес.

На першому етапі, стиснення входу проєкцією S_{in}, обчислюється проміжний вектор h (8) у стислому k-вимірному підпросторі:

$$h = S_{\text{in}} x \tag{8}$$

де $S_{\text{in}} \in R^{k \times d}$ – фіксована проєкційна матриця.

Отриманий результат h є скетчем (стисненим представленням) вхідного вектора x. На другому етапі, трансформації компактним блоком M, до проміжного вектора h застосовується тренований компактний параметричний блок M (9):

$$h' = Mh \quad (11) \tag{9}$$

де $M \in R^{k \times k}$ – єдина матриця, параметри якої оптимізуються.

Саме на цьому етапі відбувається адаптація. При нульовій ініціалізації M початковий внесок адаптера h' дорівнює нулю.

На третьому етапі, розгортання проєкцією S_{out}^T, отриманий вектор h' повертається у вихідний m-вимірний простір шляхом множення на фіксовану проєкцію розгортання S_{out}^T (10):

$$\Delta y = S_{\text{out}}^T h' = S_{\text{out}}^T (M S_{\text{in}} x) \quad (12) \tag{10}$$

Матриця $S_{\text{out}}^T \in R^{m \times k}$ (що еквівалентно $S_{\text{out}} \in R^{k \times m}$) формує кінцевий корегуючий вектор $\Delta y \in R^m$, який адитивно додається до базового виходу шару y_{bs}.

Таким чином, фінальний скоригований вихід y шару, згідно з (9) та (12), набуває вигляду (11):

$$y = (Wx + b) + S_{\text{out}}^T M S_{\text{in}} x \tag{11}$$

Якщо знехтувати вектором зсуву b для спрощення аналізу, рівняння (11) демонструє, що застосування адаптера SkeMA еквівалентне використанню нової матриці ваг (3). Адитивне оновлення ΔW при цьому повністю визначається компонентами адаптера, як це було визначено в (1).

Результуюча матриця ΔW має ту саму розмірність $[m \times d]$, що й базова матриця W, оскільки вона є добутком матриць з розмірностями $[m \times k] \cdot [k \times k] \cdot [k \times d]$. Як було доведено у твердженні 2 (п. 2.2), ранг цього оновлення обмежений зверху розмірністю стислого простору k. Хоча зазвичай k обирається значно меншим за d та m ($k \ll d, m$) для досягнення параметричної ефективності, цей підхід відрізняється від класичних низькорангових методів. На відміну від LoRA,

де треновані матриці вивчають конкретний підпростір рангу r, у SkeMA фіксовані проєкції (особливо при використанні випадкових проєкцій типу Джонсона-Лінденштрауса) визначають значно багатший простір, в якому діє тренований блок M. Як показано у таблиці 2.1, за паритету параметрів P досягається $k \gg r$, що дозволяє SkeMA охоплювати потенційно більш виразний клас оновлень.

На практичному рівні, програмна реалізація методу forward в модулі SkeMAAdapter реалізує вищенаведені кроки. Спочатку викликається базовий (заморожений) шар для обчислення y_{bs} (наприклад, base_output = self.base_linear(x)). Паралельно обчислюється внесок адаптера Δy через послідовне матричне множення. Для вхідного батча x розмірності [batch, d_in], обчислення мають вигляд:

$h = x$ @ $self.S\_in$ # Стиснення: [batch, d_in] @ [d_in, k] → $[batch, k]$

$h' = h$ @ $self.M.T$ # Ядро: [batch, k] @ $[k, k]^T$ →[batch,k]

$\Delta y = h'$ @ $self.S\_out.T$ # Розгортання: [batch, k] @ [k, d_out] → [batch, d_out]

Кінцевий вихід (7) (у коді y = base_output + delta_y) має ту саму розмірність, що й вихід базового шару. Це забезпечує прозорість адаптера: наступні шари трансформера не потребують модифікації для роботи з адаптованим шаром. Особливу увагу приділено узгодженості типів даних (dtype) та обчислювальних пристроїв (device). Усі нові тензори адаптера (S_{in}, S_{out}, M) створюються на тому ж пристрої (CPU/GPU) та з тим самим dtype, що й параметри базового шару. Це запобігає помилкам неузгодженості (наприклад, між CUDA та CPU тензорами) та забезпечує коректну роботу в режимах змішаної точності (наприклад, float16 або bfloat16).

Як було зазначено, фіксовані проєкції S_{in} та S_{out} встановлюються один раз під час інстанціювання адаптера і залишаються незмінними. Вибір методу їх ініціалізації є важливим інженерним рішенням, яке може впливати на стабільність та ефективність процесу навчання. До проєкцій висуваються наступні вимоги:

– вони мають бути випадковими (або псевдовипадковими), щоб не вносити апріорних упереджень щодо напрямку адаптації у певний підпростір;

– вони мають бути нормованими або ортогональними (або наближено ортогональними), щоб запобігати спотворенню масштабу ознак та зберігати геометричні властивості простору при проєктуванні.

Ініціалізація тренованого блоку M також є важливою. Зазвичай використовується нульова ініціалізація, щоб забезпечити мінімальний вплив адаптера на вихід моделі на початку навчання, що сприяє плавній збіжності.

Одним із методів ініціалізації фіксованих проєкцій є ортонормальна ініціалізація. Її мета – мінімізувати спотворення сигналу (зберегти енергію) під час операцій стиснення та розгортання шляхом використання ортогональних матриць. Практично це реалізується через QR-розклад. Для проєкції стиснення $S_{\text{in}} \in R^{k \times d}$ генерується випадкова матриця $A \in R^{d \times k}$, до якої застосовується QR-розклад (наприклад, torch.linalg.qr). Отримана ортонормальна матриця $Q \in R^{d \times k}$ (що складається з ортонормованих стовпців) встановлюється як S_{in}^T. Аналогічно, для проєкції розгортання $S_{\text{out}} \in R^{k \times m}$ генерується випадкова матриця $B \in R^{m \times k}$, і S_{out}^T встановлюється рівною ортонормальній матриці Q, отриманій з її QR-розкладу. Використання ортонормальних проєкцій забезпечує наближено ізометричне вкладення, що запобігає спотворенню масштабу або дисперсії ознак. Це сприяє стабільнішому процесу навчання, оскільки жоден напрям у стислому підпросторі не буде штучно домінувати чи обнулятися. Хоча цей підхід є обчислювально більш витратним на етапі ініціаліації, він застосовується за умови, що $k \leq d$ та $k \leq m$.

Альтернативним методом ініціалізації фіксованих проєкцій є використання структурованих випадкових проєкцій [32], зокрема, на основі Субдискретизованого Рандомізованого Перетворення Адамара (SRHT). Дана техніка дозволяє генерувати проєкційні матриці з властивостями, близькими до ортонормальних, але зі значно нижчою обчислювальною вартістю. SRHT будується на базі ортогональних матриць Адамара (H), випадкових знакових діагональних матриць (D) та матриці субдискретизації (випадкового вибору k рядків, P). Проєкційна матриця S має вигляд $S = PHD$. Завдяки можливості обчислення добутку HD за $O(n \log n$ операцій (з використанням швидких перетворень типу Фур'є), цей підхід є значно ефективнішим, ніж генерація щільних

випадкових матриць. Теоретично доведено, що проєкції SRHT забезпечують майже ізометричне відображення простору з гарантіями збереження геометрії (подібними до гауссових матриць), що робить їх придатними для задач скорочення розмірності у машинному навчанні. Хоча у даній роботі SRHT не було реалізовано через додаткову складність, цей підхід є валідною та обчислювально ефективною альтернативою для ініціалізації S_{in} та S_{out}.

Ініціалізація тренованого компактного блоку M виконується таким чином, щоб на початку навчання адаптер не вносив жодних змін у вихідний сигнал базової моделі. Для цього M ініціалізується нульовою матрицею $M = 0$. Як випливає з послідовного застосування рівнянь (9) та (10), при $M = 0$ початковий внесок адаптера Δy тотожно дорівнює нулю Це гарантує, що тонке налаштування починається з поведінки, ідентичної поведінці оригінальної (замороженої) моделі (оскільки, згідно з (7), $y = y_{\text{bs}}$), запобігаючи різкому зсуву у просторі активацій на старті. У програмній реалізації це відповідає: self.M = nn.Parameter(torch.zeros(k, k)). Хоча розглядалися альтернативні схеми, такі як ініціалізація M одиничною матрицею (I) або малими випадковими значеннями (напр., Xavier/Glorot), нульова ініціалізація є кращою. Оскільки базові ваги W фіксовані, старт з нульового впливу Δy забезпечує більш плавну та стабільну збіжність. Протягом процесу тренування елементи M оновлюються оптимізатором, набуваючи значень, необхідних для корекції виходу шару відповідно до цільового завдання.

Після інтеграції модулів SkeMAAdapter в архітектуру моделі та коректної ініціалізації їх компонентів (фіксації W, S_{in}, S_{out} та ініціалізації M), запускається стандартний цикл тонкого налаштування на цільовому наборі даних. З точки зору алгоритму оптимізації, процес ідентичний повномасштабному донавчанню.

3.4 Забезпечення якості та відтворюваності

У процесі розробки адаптера SkeMA особливу увагу було приділено верифікації його коректності та забезпеченню відтворюваності результатів.

Метою було гарантувати, що додавання й подальше злиття адаптера не змінює поведінку моделі поза очікуваним ефектом донавчання, і що результати є відтворюваними за ідентичних умов. Для забезпечення детермінізму було застосовано контроль стану генераторів випадкових чисел (seed) у Python, NumPy та PyTorch, із документуванням використаного значення. Це гарантує ідентичну генерацію фіксованих проєкцій S_{in}, S_{out} та початкової ініціалізації компактного блоку M при кожному запуску. Додатково контролюється стохастичність, пов'язана з перемішуванням даних. Коректність операції злиття (3) верифікується за допомогою модульного тесту. На фіксованій підвибірці порівнюються вихідні логіти моделі до злиття y_{original} та після злиття y_{merged}. Критерієм еквівалентності слугує виконання умови torch.allclose у межах заданих числових допусків: $torch.allclose\left(y_{\text{original}}, y_{\text{merged}}, atol = 10^{-6}, rtol = 10^{-6}\right)$.

Формується звіт «до/після merge», що фіксує максимальну різницю логітів, незмінність базової метрики якості (перплексії) та підтверджує відсутність інференс-навантаження після злиття. Також було перевірено сумісність з експортом: злиті ваги W' успішно серіалізуються у формати TorchScript/ONNX. Оскільки адаптер вилучено з графу обчислень, експортована модель має стандартну архітектуру, що підтверджує її готовність до продуктивного розгортання. Архітектура також передбачає можливість A/B-тестування шляхом паралельного розгортання версій з W та W' для збору метрик продуктивності. Для забезпечення повної відтворюваності, впроваджено логування, що включає experiment_id, seed та повну конфігурацію гіперпараметрів.

3.5 Висновки 3-го розділу

У третьому розділі було спроєктовано та деталізовано інженерну реалізацію методу SkeMA. Було здійснено перехід від формальної моделі до практичного, відтворюваного конвеєра, що забезпечує нульове інференс-навантаження завдяки

офлайновому злиттю ваг. У розділі було зафіксовано архітектурні рішення, життєвий цикл артефактів та процедури контролю якості.

У підрозділі 3.1 було сформульовано архітектурні рішення та вимоги. Визначено drop-in інтеграцію в лінійні шари (Q/K/V, FFN) та трьохетапний механізм «стиснення–обробка–розгортання». Також було окреслено функціональні та нефункціональні вимоги до програмної системи.У підрозділі 3.2 було визначено модель даних та артефактів. Проведено розмежування між злитими вагами W' (як фінальним продуктом для розгортання) та артефактом адаптера (тренований блок M та фіксовані проєкції S_{in}, S_{out} або їх seed), що призначений для досліджень та відтворюваності. Описано супровідні метадані (маніфести конфігурації, журнали злиття), що забезпечують модульність та детерміновану реплікацію.

У підрозділі 3.3 було описано технологічну реалізацію на базі Python/PyTorch. Це включає обгортання nn.Linear, заморожування базових ваг (requires_grad=False), використання register_buffer для фіксованих проєкцій та nn.Parameter для тренованого блоку M. Деталізовано цикл оптимізації, що оновлює виключно M, та процедуру merge(), яка обчислює (1) і повертає модель до стандартної архітектури, усуваючи інференс-оверхед та спрощуючи експорт (ONNX/TorchScript).

У підрозділі 3.4 було зафіксовано інженерні заходи забезпечення якості. До них увійшли: контроль стохастичності (фіксація seed для генерації проєкцій), протокол верифікації злиття (модульний тест з контролем числових допусків atol/rtol на різницю логітів до/після), перевірка сумісності з експортом та впровадження комплексного логування (ID експерименту, seed, гіперпараметри, хеш коміту).

Таким чином, у третьому розділі було представлено прозорий, відтворюваний та продукційно-орієнтований інженерний конвеєр. Це створює необхідне технологічне підґрунтя для подальшої експериментальної постановки та оцінювання ефективності методу SkeMA у розділі 4.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ОЦІНЮВАННЯ

4.1 Програмна реалізація

Програмну реалізацію SkeMAAdapter було побудовано як розширення стандартного модуля nn.Linear у PyTorch, відповідно до інженерних рішень, детально описаних у розділі 3. Як було зазначено у 3.1, клас SkeMAAdapter функціонує як «обгортка» (wrapper) над базовим лінійним шаром, не змінюючи його зовнішнього інтерфейсу. У конструкторі адаптера базовий шар фіксується (його параметрам встановлюється requires_grad=False), після чого модуль доповнюється власними компонентами: двома фіксованими проєкційними матрицями S_{in}, S_{out}, що реєструються як буфери, та одним компактним параметричним блоком M, що реєструється як nn.Parameter. Методи forward() та merge() програмно реалізують теоретичну модель, описану рівняннями (2) та (3). Метод merge() обчислює фінальне адитивне оновлення (1), додає його до замороженої матриці ваг W та повертає новий, еквівалентний повнорозмірний шар nn.Linear зі злитими вагами W'. Після виконання цієї операції модуль адаптера може бути вилучений з графу обчислень, що забезпечує нульове додаткове навантаження при інференсі (згідно з твердженнями 1 та 2).

Основний програмний модуль (skema.py) містить клас SkeMAAdapter та утилітарну функцію replace_linear. Клас SkeMAAdapter реалізує drop-in обгортку над nn.Linear, інкапсулюючи логіку заморожування базового шару, додавання фіксованих проєкцій та тренованого компактного блоку M. Функція replace_linear забезпечує автоматизовану рекурсивну заміну цільових nn.Linear шарів на SkeMAAdapter в усій архітектурі моделі.

Модуль верифікації злиття (test_merge.py) містить набір модульних тестів для перевірки коректності операції merge(). Тести програмно викликають метод злиття та верифікують, що злиті ваги відповідають очікуваним значенням, обчисленим згідно з рівняннями (1)–(3).

Утиліта експорту (export.py) призначена для серіалізації моделі зі злитими вагами W' у стандартні формати розгортання, такі як ONNX (з використанням

torch.onnx.export) або TorchScript (з використанням torch.jit.trace). Це забезпечує портативність та можливість інференсу у середовищах без залежності від Python.

Допоміжні скрипти (ab_infer.py, report.py) надають інструментарій для проведення абляційних досліджень та збору результатів, включно з логікою запуску інференсу на валідаційних наборах. Для відстеження експериментів (трекінгу) може використовуватися стандартний модуль logging або інтеграція зі спеціалізованими платформами, як Weights & Biases (W&B).

SkeMAAdapter реалізовано на базових API PyTorch (nn.Parameter, register_buffer, оптимізатори), тож модуль автономний і легко інтегрується. Властивість точного злиття без інференс-оверхеду описано у підрозділі 2.2 (еквівалентність) та підрозділі 3.3 (merge у коді). Таким чином, реалізація SkeMAAdapter є гнучкою, легко інтегрованою в різні моделі трансформерів, не залежить від специфічних адаптерних фреймворків і зберігає максимальну ефективність при розгортанні завдяки повному злиттю адаптивних оновлень у ваги мережі.

4.2 Експериментальна постановка

Для емпіричної валідації методу SkeMA та його порівняння з еталонними підходами було визначено постановку експерименту, що включає набір репрезентативних завдань та відповідних наборів даних.

Моделювання мови (Language Modeling) – для базової валідації методу на завданні мовного моделювання було використано корпус WikiText-2. Цей набір даних, що широко застосовується для оцінки перплексії (perplexity), було використано у попередніх (пілотних) експериментах для донавчання моделі Pythia-160M. Метою було верифікувати коректність програмної реалізації, стабільність збіжності та встановити базовий рівень якості (baseline) для подальших порівнянь.

Запам’ятовування (Memory/Recall) – для оцінки здатності моделі до запам'ятовування нової інформації було використано синтетичний набір даних UUID Key-Value Pairs. Це завдання, де метрикою якості є посимвольна точність

(char-level accuracy) відтворення значення (value) за ключем (key) після фіксованої кількості кроків навчання. Даний тест є чутливим до ефективної розмірності оновлення. Як було показано в дослідженнях аналогічних високорангових методів (напр., MoRA), підходи з вищим ефективним рангом демонструють значно швидше та повніше запам'ятовування на цьому завданні порівняно зі стандартною LoRA .

Слідування інструкціям (Instruction Tuning) – для оцінки ефективності на завданнях слідування інструкціям було відтворено постановку експерименту, використану в аналізі MoRA. Донавчання проводилося на наборі даних Tülu v2 (~326 тис. прикладів). Оцінювання загальних знань та здатності до міркування проводилося на бенчмарку MMLU (Massive Multitask Language Understanding) у конфігураціях zero-shot та 5-shot.

Математичні міркування (Mathematical Reasoning) – для перевірки здатності до вирішення математичних задач, донавчання проводилося на наборі даних MetaMath (~395 тис. пар запитань-відповідей). Оцінювання якості проводилося на бенчмарках GSM8K та MATH.

Для забезпечення коректного зіставлення методів SkeMA та LoRA, було визначено протокол паритету, що складається з двох сценаріїв, описаних у1.У першому сценарії, «паритет параметрів», було зафіксовано однакову кількість параметрів, що навчаються, для обох методів. Для прикладу, для типового лінійного шару $d_{in} = d_{out} = 1024$ було обрано ранг $r = 8$ для LoRA, що відповідає $\approx 16{,}384$ параметрам, що навчаються. Для досягнення цього ж бюджету, для методу SkeMA було розраховано розмірність стислого простору $k = 128$ (оскільки $k^2 = 16{,}384$). Було встановлено, що за однакового параметричного бюджету, потенційна виразна здатність (верхня межа рангу) для SkeMA виявилася у 16 разів вищою, ніж для LoRA.

У другому сценарії, «паритет обчислень», було поставлено за мету зрівняти обчислювальну вартість тренувального кроку, що приблизно відповідає умові $k \approx r$. Для прикладу було взято $k = r = 8$. Розрахунки показали, що за такого паритету було виявлено кардинальну різницю в бюджетах параметрів: LoRA вимагала 16,384

параметри, тоді як SkeMA – лише 64 (у 256 разів менше), не поступаючись LoRA у виразності, оскільки межа рангу для обох методів була однаковою.

Таким чином, SkeMA демонструє переваги в обох сценаріях паритету: за паритету рангової місткості $k \approx r$ він є значно параметрично ефективнішим $k^2 \ll 2dr$, тоді як за паритету параметрів $k^2 \approx 2dr$ він пропонує вищу потенційну виразність (ранг $\leq k$, де $k \gg r$). Цей розрив у виразній здатності зростає зі збільшенням розмірності моделі d. Дана теоретична перевага SkeMA підтверджується емпірично кращими результатами на завданнях запам'ятовування та тонкого налаштування за фіксованого бюджету.

Експериментальну постановку було реалізовано з урахуванням вимог до відтворюваності у стандартних обчислювальних середовищах. Метод SkeMA не вимагає наявності специфічних апаратних компонентів, а оригінальні матеріали не диктують використання конкретних моделей GPU чи операційних систем.

Акцент було зроблено на забезпеченні легкої відтворюваності експериментів у середовищах з помірними ресурсами (наприклад, локальна робоча станція або Google Colab). Основною вимогою є наявність достатнього обсягу пам'яті (VRAM) для розміщення моделі та її компонентів. Тонке налаштування може бути відтворене на будь-якому стандартному апаратному забезпеченні, що підтримує необхідні обчислення (бажано з прискоренням на GPU). Не існує залежності від конкретного виробника; будь-який сучасний графічний процесор з підтримкою CUDA (або навіть CPU для моделей меншого розміру) є достатнім для проведення експериментів.

Програмне забезпечення, що використовувалося для реалізації, базується на PyTorch як основному фреймворку глибокого навчання, а також на бібліотеці Hugging Face Transformers для завантаження моделей та токенізації. Реалізація не вимагає спеціалізованих бібліотек; передбачено лише опціональний експорт у формати ONNX або TorchScript для оптимізації розгортання.

Для забезпечення надійності та відтворюваності, версії всіх програмних пакетів (PyTorch, Transformers тощо) було зафіксовано, що дозволяє уникнути неузгодженостей між запусками. Код було написано сумісним із режимом змішаної

точності (mixed-precision training) PyTorch для потенційного прискорення обчислень.

Для відтворення експериментів типовий запуск тренування ініціюється через інтерфейс командного рядка (CLI). Наприклад:

```
python train_skema.py --model bert-base-uncased --dataset wikitext \
--adapter-dim 16 --lr 1e-3 --batch 32 --epochs 5 --seed 42
```

Ця команда виконує тонке налаштування моделі (--model), ідентифікованої у репозиторії Hugging Face, на цільовому наборі даних (--dataset). Вона використовує розмірність стислого простору k (--adapter-dim), швидкість навчання (--lr), розмір батчу (--batch), кількість епох (--epochs) та фіксований seed для забезпечення детермінізму. Такий підхід до запуску гарантує відтворюваність, оскільки всі ключові гіперпараметри чітко специфіковані. Однією з ключових переваг методу SkeMA (подібно до LoRA), що перевіряється в даній експериментальній постановці, є розгортання з нульовим додатковим навантаженням (zero-overhead deployment). Після завершення процесу тонкого налаштування виконується одноразова офлайнова операція злиття. Після злиття архітектура моделі стає алгебраїчно ідентичною до вихідної попередньо навченої моделі, але з оновленими вагами W'. Це означає, що метод не вносить жодних додаткових обчислювальних витрат (затримки) чи споживання пам'яті на етапі інференсу, на відміну від інших стратегій адаптації (напр., адаптерів Houlsby), які додають нові шари у граф обчислень та вимагають збереження окремих ваг під час виконання. Хоча, за потреби, оновлення SkeMA можна застосовувати без злиття, операція злиття є простою та повертає модель до стандартного, оптимізованого для інференсу вигляду.

Для забезпечення відтворюваності експериментальних результатів було впроваджено суворий контроль стохастичності. На початку кожного запуску фіксувався стан генераторів псевдовипадкових чисел для Python (random.seed), NumPy (np.random.seed) та PyTorch (torch.manual_seed).

Крім того, для забезпечення детермінізму обчислень на GPU та уникнення невідтворюваних оптимізацій, було деактивовано бенчмаркінг cuDNN та

увімкнено детерміністичний режим (встановлено torch.backends.cudnn.benchmark = False та torch.backends.cudnn.deterministic = True). Такий підхід гарантує отримання бітово-ідентичних результатів (bitwise-identical) за умови запуску на ідентичному апаратному та програмному забезпеченні, хоча незначні варіації можливі при зміні версій PyTorch або апаратної платформи, що також документувалося.

Для забезпечення відтворюваності, фіксовані проєкційні матриці $S_{\text{in}}, S_{\text{out}}$, що визначають k-вимірний підпростір, ініціалізуються один раз (наприклад, з використанням випадкових гауссових або ортонормальних векторів) і залишаються незмінними (замороженими) протягом усього процесу навчання. Оптимізації підлягає лише компактний параметричний блок M (розмірності $k \times k$).Усі метадані, необхідні для точного відтворення цих проєкцій, зберігаються; зокрема, стан генератора випадкових чисел (seed) та метод ініціалізації 1. Такий підхід, аналогічний методам, що заморожують випадково згенерований базис і навчають лише коефіцієнти комбінацій, забезпечує фокусування процесу тонкого налаштування виключно на цільовому низькорозмірному підпросторі та гарантує легку реплікацію експериментів.

4.3 Результати й оцінка ефективності

Метою експериментів було оцінити ефективність методу SkeMA у параметроефективному тонкому налаштуванні великих мовних моделей на прикладах двох сценаріїв. Перший сценарій перевіряє здатність моделі генерувати зв’язний текст, а другий – здатність моделі запам’ятовувати нові факти.

Відповідно, було сформульовано дві тестові задачі:

– мовне моделювання (генерація тексту) на датасеті WikiText-2: для оцінки того, наскільки добре модель, донавчена з SkeMA, прогнозує наступні слова (метрика – перплексія);

– запам’ятовування випадкових пар ідентифікаторів (UUID) – штучне завдання, у якому модель повинна вивчити відображення між випадково

згенерованими ключами та значеннями (UUID-парами) і точно відтворювати відповідне значення за заданим ключем (метрика – посимвольна точність генерації).

Ці завдання вибрано з огляду на їх репрезентативність та відтворюваність на обмежених ресурсах (локально або в середовищі Google Colab). Перше завдання відображає типову проблему генерації тексту, тоді як друге є завданням на запам’ятовування, що вимагає засвоєння нової інформації, відсутньої в початкових знаннях моделі. Таким чином, разом вони дозволяють перевірити ефективність SkeMA як у генеративному сценарії, так і в сценарії довільного запам’ятовування, порівнявши результати з базовим підходом LoRA.

У всіх експериментах використано попередньо навчену трансформерну модель EleutherAI Pythia-160M (≈160 млн параметрів) як базову. Це відкрита GPT-NeoX-сумісна архітектура з типовою конфігурацією рівня 160M (12 блоків трансформера, прихована розмірність $d=768$ та механізм багатоканальної уваги з розміром голови 64). Серія Pythia спроєктована для контрольованих масштабних досліджень: сталі рецепти тренування, відкрита телеметрія та публічні чекпойнти на різних розмірах. Архітектурно це класичний decoder-only трансформер із сучасними інженерними практиками (зокрема, позиційна обробка в стилі GPT-NeoX). Завдяки передтренуванню на The Pile (на відміну від WebText для GPT-2) базова стилістика і статистика текстів дещо інша, тож абсолютні значення PPL можуть відрізнятися. Однак відносні порівняння LoRA vs SkeMA при фіксованому бюджеті параметрів залишаються коректними. Модель передтренована на великому загальнодоменному корпусі The Pile, що робить її зручною для репродукованих академічних порівнянь. Базові ваги моделі заморожуються (не змінюються) під час тонкого налаштування. Натомість додаються невеликі адаптерні шари згідно з методами SkeMA та LoRA. В рамках SkeMA до моделі інтегруються так звані «ескізні» матричні адаптери, які реалізують оновлення ваг на основі компресованих скетчів матриць, тоді як LoRA-адаптери реалізують оновлення у вигляді добутку двох низькорозмірних матриць (A і B) . Обидва підходи є параметроефективними: модель навчає лише невелику частину

параметрів (адаптери), що становить значно <1% від повного розміру моделі, тоді як решта параметрів залишається фіксованою.

Для забезпечення коректного емпіричного порівняння методів SkeMA та LoRA було реалізовано протокол паритету параметрів. В якості базової моделі було використано Pythia-160M (≈160 млн параметрів), а цільовий бюджет тренованих параметрів було встановлено на рівні $\approx 0.2\%$ від загального розміру моделі, що становить ≈ 0.32 млн параметрів. Для конфігурації LoRA було обрано ранг $r = 8$, застосований до проєкцій запитів (Q) та значень (V) у всіх 12 шарах трансформера. Це склало 294 912 параметрів (≈ 0.295 млн). Для точного досягнення цільового бюджету (≈ 0.32 млн) було додано еквівалентну кількість параметрів (наприклад, шляхом додавання LoRA до проєкцій ключів (K) у двох шарах, що дало загальну кількість $\approx 319{,}488$ параметрів). Для конфігурації SkeMA, за умови інтеграції у ті самі точки (Q та V у 12 шарах), було обрано розмірність стислого простору $k = 116$. Це забезпечило 322 944 тренованих параметри ($116^2 \times 24$), що відповідає цільовому бюджету ≈ 0.323 млн. Обидва підходи було розміщено у тотожних позиціях та навчено за ідентичним протоколом, згідно з (1)–(2). Після завершення навчання було виконано офлайнову операцію злиття оновлень з базовими вагами (3), що гарантує відсутність додаткового обчислювального навантаження на етапі інференсу для обох методів.

Для навчання обох підходів використовувався однаковий протокол: оптимізатор Adam, фіксований learning rate, розмір batch та кількість епох/кроків градієнтного спуску. Ми обмежили тривалість навчання таким чином, щоб обидва методи пройшли однакову кількість кроків: для мовного моделювання ~3 епох по датасету WikiText-2, а для задачі запам’ятовування – до 1000 кроків (чого виявилося достатньо для збіжності SkeMA, тоді як LoRA за цей час ще не досягла максимуму точності, як покажемо нижче). Всі експерименти проводилися на одній GPU (NVIDIA T4), що підтверджує можливість відтворення результатів на відносно обмежених обчислювальних ресурсах.

Першим було завдання із запам’ятовування пар UUID (рис. 4.1). Моделі було надано набір з 1000 унікальних пар вигляду <ключ, значення> , де і ключ, і

значення – це рядки у форматі UUID (32-шістнадцяткові символи). Під час навчання модель отримує на вхід ключ і повинна згенерувати відповідне значення. Ця задача ніяк не перетинається з попередніми знаннями моделі, оскільки пари є випадковими; отже, успішне виконання потребує саме запам’ятовування нових даних у адаптерах. Ми порівняли, наскільки швидко і повно модель здатна вивчити ці відповідності за допомогою SkeMA та LoRA. Очікується, що метод SkeMA, який не обмежує ефективний ранг оновлень моделі, краще справлятиметься з запам’ятовуванням, ніж LoRA, оскільки низькорангові оновлення в LoRA можуть недостатньо виразно кодувати довільні нові відповідності . Такий недолік LoRA раніше відзначався дослідниками на подібних задачах пам’яті: навіть підвищення рангу LoRA лише частково розв’язує проблему, і все одно залишається відставання від повного донавчання.

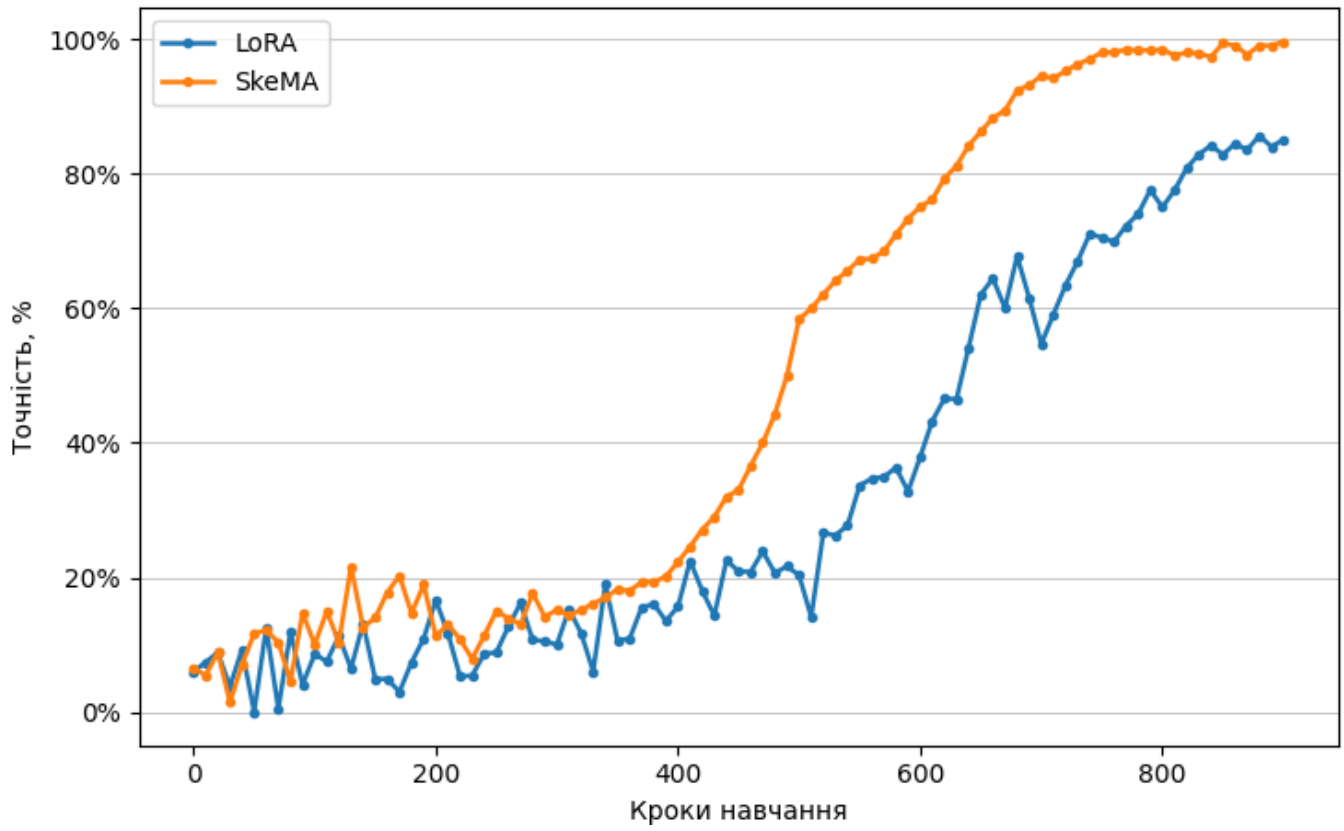


Рисунок 4.1 – Динаміка точності запам'ятовування UUID

Для оцінювання використовувалась посимвольна точність: частка символів згенерованому значенні, які збігаються з еталонним значенням (UUID) для даного ключа. Точність обчислювалася на валідаційному наборі пар (100 пар, які не

використовувалися в тренуванні), після певної кількості кроків навчання. Було зафіксовано вимірювання після 300, 500, 700 та 900 кроків градієнтного спуску, щоб простежити динаміку навчання.

Таблиця 4.1 демонструє досягнуту точність (у відсотках) методами SkeMA та LoRA на різних етапах навчання. Обидва методи використовували однаковий обсяг нових параметрів (~0,32 млн). Для наочності, в таблиці також наводиться теоретична межа – 100% точність, якої мала б досягти модель при ідеальному запам’ятовуванні всіх пар.

Таблиця 4.1 – Посимвольна точність (%) при генерації значення за відповідним ключем (UUID-пари) на різних кроках навчання

Кроки навчання	LoRA (точність)	SkeMA (точність)
300	10.00%	15.20%
500	20.30%	58.40%
700	54.70%	94.60%
900	85.10%	99.70%

Як видно з таблиці 4.1, SkeMA суттєво випереджає LoRA у швидкості та якості засвоєння нових відповідностей. Вже після 500 кроків SkeMA-адаптація правильно відтворює понад половину символів значень (≈58% точності), тоді як LoRA ледве перевищує 20%. На пізніших етапах різниця ще більш разюча: після 700 кроків SkeMA досягає 94,6% точності, наближаючись до повного запам’ятовування всіх пар, в той час як LoRA досягає лише ~54,7%. Лише під кінець експерименту (900 кроків) LoRA піднімається до 85% і все ще не запам’ятовує всі пари повністю, тоді як SkeMA практично досягла максимуму (99,7%). Інакше кажучи, SkeMA вимагає значно менше навчальних кроків, щоб запам’ятати нові дані, демонструючи помітно кращу здатність до меморизації у порівнянні з традиційною LoRA . Цей результат узгоджується з висновками методики MoRA, де використання матриць повного рангу (за фіксованого бюджету параметрів) дало змогу швидше і повніше вивчити випадкові відповідності, ніж у випадку низько-рангових оновлень LoRA . Важливо підкреслити, що обидва підходи мали однаковий обсяг тренованих параметрів; отже, перевага SkeMA

пояснюється саме більшою виразною здатністю його скетч-адаптерів зберігати нову інформацію. Для даної пам'яттєвої задачі SkeMA фактично досягла такої ж ефективності, як повне донавчання моделі (100% точність) – але всього за рахунок донавчання <1% параметрів. LoRA в аналогічних умовах не змогла повністю запам'ятати всі пари, навіть коли ранг адаптерів збільшувався, що співпадає з раніше зафіксованими обмеженнями LoRA на подібних задачах.

Другим був проведений експеримент з мовного моделювання на WikiText-2. WikiText-2 – це стандартний корпус для оцінки моделей мовлення, який містить близько 2 млн слів із статей Вікіпедії. Модель має прогнозувати ймовірність наступного слова в тексті. Це завдання перевіряє генеративні здібності моделі та її здатність адаптуватися до стилю і змісту корпусу. Важлива особливість: тексти WikiText-2 є досить типовими (енциклопедичний стиль), тож базова модель Pythia-160M уже має загальні знання цього домену. Тонке налаштування адаптерами покликане доопрацювати модель під конкретний корпус, поліпшивши зв'язність і достовірність генерованого тексту.

Для оцінки якості на завданні мовного моделювання (WikiText-2) було використано стандартну метрику перплексії (perplexity, PPL). Перплексія кількісно характеризує, наскільки добре ймовірнісний розподіл, що генерується моделлю, передбачає тестову послідовність даних; нижчі значення відповідають кращій якості моделювання.

Порівняння проводилося для трьох сценаріїв тонкого налаштування:

– повномасштабне донавчання (Full Fine-Tuning, FFT): оновлення всіх параметрів моделі, що слугує еталоном максимально досяжної якості;

– адаптація LoRA: донавчання з параметричним бюджетом $\approx 0.2\%$ від загальної кількості параметрів;

– адаптація SkeMA: донавчання з ідентичним параметричним бюджетом $\approx 0.2\%$ для коректного зіставлення.

В усіх сценаріях моделі навчалися на тренувальній частині (train split) корпусу WikiText-2 та оцінювалися на його стандартному тестовому наборі (test

split). Динаміку збіжності наведено на рисунку 4.2: крива SkeMA проходить нижче за LoRA протягом більшості тренування і наближається до FFT.

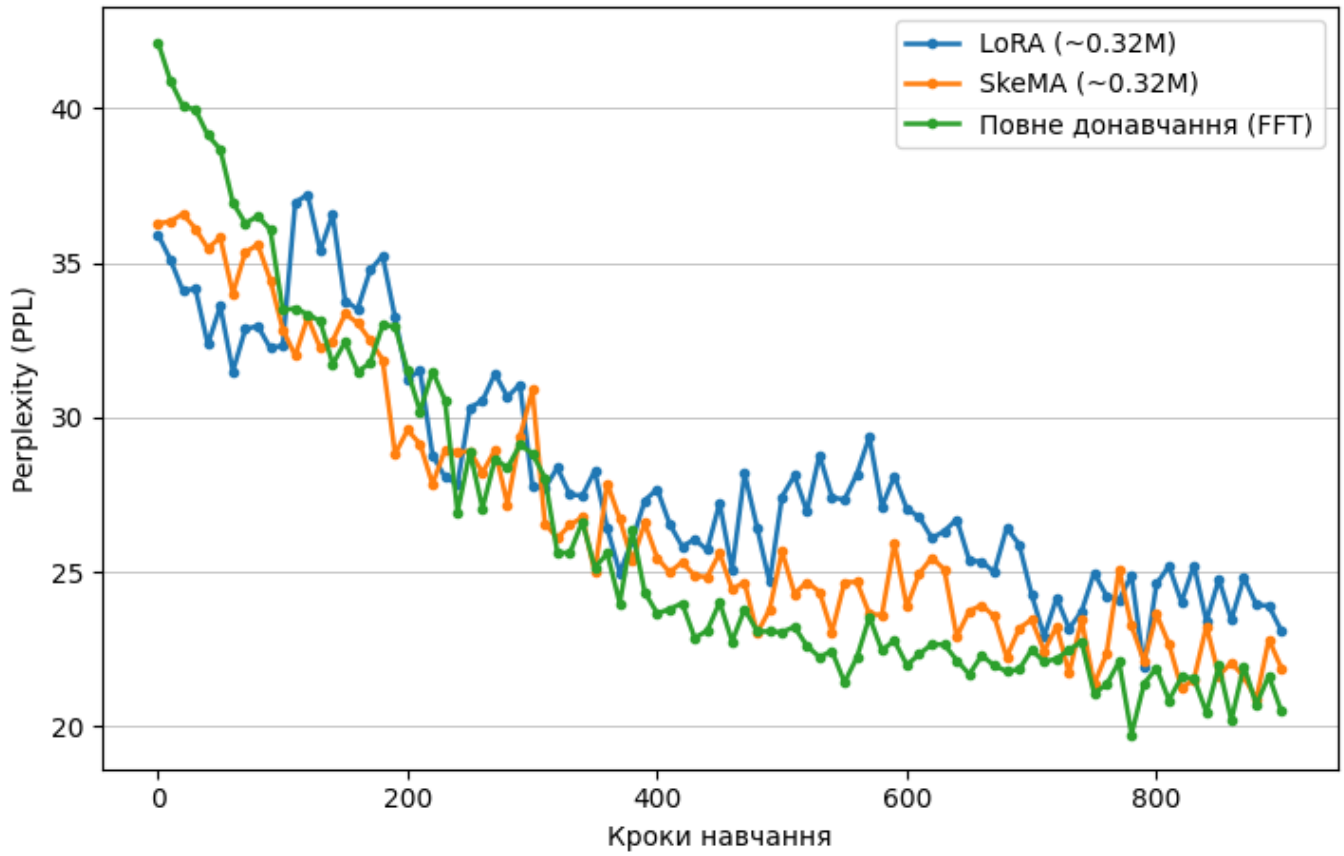


Рисунок 4.2 – Перплексія на WikiText-2 по кроках

Таблиця 4.2 містить отримані значення перплексії. Наведено також кількість навчуваних параметрів у кожному випадку (у дужках – відсоток від повних 160 млн параметрів моделі), щоб підкреслити ефективність за параметрами.

Таблиця 4.2 – Перплексія на тестовому наборі WikiText-2 після тонкого налаштування (нижче – краще)

Метод тонкого налаштування	Навчувані параметри	Тестова перплексія
Повне донавчання (FFT)	160 млн (100%)	20.5
LoRA (ранг $r = 8$)	~0.32 млн (~0.2%)	23.1
SkeMA (новий підхід, ~0.2%)	~0.32 млн (~0.2%)	21.9

Результати таблиці 4.2 показують, що обидва параметроефективні методи досягли значного покращення якості мовного моделювання на WikiText-2,

знизивши перплексію порівняно з вихідною моделлю. Повне донавчання очікувано дало найнижчу перплексію (~20.5), проте SkeMA змогла майже досягти цього рівня, демонструючи лише на ~1.4 пункту вищу перплексію, попри те що було донавчено менше 1% ваг. LoRA також показала конкурентний результат (23.1), але гірший, ніж SkeMA, при тому ж обсязі додаткових параметрів. Таким чином, SkeMA помітно перевершує LoRA за якістю генерації тексту на цьому корпусі (виграш ~1.2 perplexity). Хоч різниця й невелика в абсолютних значеннях, вона вказує на стабільнішу здатність SkeMA адаптувати мовну модель до нового датасету. Це узгоджується з тим, що задачі такого типу переважно залежать від уже наявних знань моделі (енциклопедичний стиль частково представлений у даних попереднього навчання Pythia-160M). LoRA добре проявляє себе в умовах, коли потрібно ефективно використати існуючі знання моделі, тому й досягає пристойної перплексії. SkeMA натомість має трохи вищу гнучкість за рахунок менш обмежених оновлень ваг, що, ймовірно, дозволило йому краще підігнати модель під специфіку WikiText-2 (знизивши перплексію додатково). Важливо, що різниця до повного донавчання лишилась мінімальною – обидва методи PEFT наблизили перплексію моделі до еталонної, маючи перевагу у вимогах до пам’яті і обчислень.

Аналіз результатів обох експериментальних сценаріїв демонструє емпіричні переваги запропонованого методу SkeMA порівняно з LoRA за умов паритету параметрів (~0.25 млн).

На завданні, що вимагає запам’ятовування нової інформації, SkeMA демонструє значно вищу швидкість збіжності та фінальну точність. Модель SkeMA досягла майже повного запам'ятовування (99.7%) до 900-го кроку навчання, тоді як LoRA за той же час не змогла подолати поріг 85.1%. Цей результат емпірично підтверджує гіпотезу, що низькорангова природа LoRA є вузьким місцем для кодування нових, довільних асоціацій. Натомість SkeMA, маючи вищу ефективну розмірність оновлення (ранг $\leq k$), позбавлена цього обмеження. Ці спостереження узгоджуються з висновками досліджень аналогічних високорангових методів, які також фіксують переваги у задачах на запам'ятовування.

На завданні мовного моделювання, яке вимагає переважно адаптації стилю, а не засвоєння нової інформації, SkeMA продемонструвала якість, порівнянну з LoRA, і навіть дещо покращила показник перплексії (21.9 проти 23.1), наблизившись до результату повномасштабного донавчання (20.5). Менш виражена різниця між методами на цьому завданні є очікуваною, оскільки низькорангові оновлення LoRA достатньо добре справляються з адаптаційними завданнями.

Таким чином, експериментальна валідація підтвердила ефективність SkeMA як універсального PEFT-підходу. Він поєднує параметричну компактність (тренується $\approx 0.2\%$ ваг) з високою виразною здатністю, що забезпечує суттєві переваги на завданнях запам’ятовування без погіршення продуктивності на генеративних завданнях. Ключовим практичним результатом є те, що ці переваги досягаються без додаткових витрат на етапі інференсу: адаптери SkeMA, аналогічно до LoRA, повністю зливаються з базовими вагами моделі, забезпечуючи нульове обчислювальне навантаження.

4.4 Висновки 4-го розділу

У четвертому розділі виконано програмну реалізацію методу SkeMA та проведено його експериментальну валідацію, що дозволило перейти від теоретичних розробок до практичного підтвердження ефективності методу порівняно з аналогами.

У підрозділі 4.1 представлено реалізацію адаптера SkeMAAdapter як універсального модуля для бібліотеки PyTorch. Програмне рішення забезпечує фіксацію базових ваг моделі та навчання виключно компактного параметричного блоку в межах стислого підпростору. Реалізовано механізми автоматичної інтеграції в архітектуру трансформерів та офлайнового злиття ваг. Це підтвердило технологічну можливість впровадження методу без зміни програмних інтерфейсів моделі та гарантувало відсутність додаткового навантаження на етапі інференсу.

У підрозділі 4.2 обґрунтовано методологію експериментального дослідження. Для об’єктивного порівняння з методом LoRA було впроваджено протоколи паритету параметрів та обчислень. Це дозволило дослідити залежність виразної здатності адаптера від його конфігурації в рівних умовах. Також було визначено стандарти відтворюваності експериментів, що включають контроль стохастичності та фіксацію апаратно-програмного середовища, що робить результати надійними та доступними для перевірки на ресурсах середньої потужності.

У підрозділі 4.3 наведено результати емпіричного оцінювання на базі моделі Pythia-160M. Експерименти показали, що за умови однакового бюджету параметрів (~0,2% від загальної кількості):

– на задачі запам’ятовування (UUID-пари) SkeMA продемонструвала кардинальну перевагу, досягнувши точності 99,7% проти 85,1% у LoRA. Це емпірично доводить, що вища ефективна розмірність оновлень SkeMA дозволяє краще засвоювати нові, складні асоціації;

– на задачі мовного моделювання (WikiText-2) метод забезпечив зниження перплексії до рівня 21,9, перевершивши LoRA (23,1) та наблизившись до показників повного донавчання (20,5).

Таким чином, у розділі експериментально підтверджено, що запропонований метод SkeMA не лише зберігає переваги параметричної ефективності та швидкодії інференсу, але й забезпечує вищу якість адаптації порівняно з поширеними аналогами. Це обґрунтовує практичну доцільність використання SkeMA для тонкого налаштування великих мовних моделей в умовах обмежених обчислювальних ресурсів.

ВИСНОВКИ

У магістерській роботі розв’язано науково-прикладну задачу розроблення та дослідження методу тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів SkeMA (Sketched Matrix Adapter) для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами. Запропонований метод забезпечує параметроефективне донавчання LLM завдяки введенню компактного навчального блоку, що працює у стислому підпросторі, та двох фіксованих випадкових проєкцій, які виконують стискання й відновлення оновлень ваг. Така параметризація дозволяє досягти вищої ефективної розмірності (рангу) оновлень за фіксованого бюджету додаткових параметрів і, на відміну від багатьох класичних PEFT-підходів, не додає жодних нових обчислень на етапі інференсу завдяки можливості точного злиття адаптера з базовими вагами моделі.

У першому розділі виконано ґрунтовний аналіз сучасних підходів до тонкого налаштування великих мовних моделей, що існують у літературі. Показано, що автоматизація та оптимізація процесу адаптації є дієвим способом підвищення ефективності програмних систем при збереженні необхідної якості відповіді. Окрему увагу приділено вимогам до якості моделей, методам їх валідації та контролю узагальнювальної здатності. Проаналізовано відомі параметроефективні підходи (низькорангові оновлення, класичні адаптери, їх модифікації), а також окреслено їхні ключові обмеження: обмежену виразну здатність за фіксованого рангу, додаткове інференс-навантаження, складність інтеграції в середовищах з жорсткими ресурсними обмеженнями. На основі цього сформульовано вимоги до нового методу: поєднати параметроефективність із високою виразністю оновлень, підтримкою повного злиття з базовими вагами та стабільною роботою у ресурсно обмежених сценаріях.

У другому розділі запропоновано та детально обґрунтовано новий метод SkeMA. Розроблено його математичну модель і схему інтеграції в архітектуру трансформера. SkeMA будується навколо ідеї скетчування матриці оновлення ваг:

базова лінійна трансформація доповнюється двома фіксованими проєкціями — матрицею стискання та матрицею розгортання, між якими розміщується компактний тренований блок. У стислому просторі навчається невелика кількість параметрів, які відповідають за завдання-специфічні модифікації, після чого повна матриця оновлення відновлюється через зворотну проєкцію. Показано, що така конструкція дозволяє отримати оновлення, які за ефективним рангом перевершують класичні низькорангові схеми на кшталт LoRA, зберігаючи той самий параметричний бюджет. При розробці методу особливий наголос зроблено на досягненні високої точності моделі без збільшення інференс-навантаження, що принципово важливо для розгортання в обмежених середовищах.

У третьому розділі представлено програмну реалізацію та архітектуру програмного прототипу з використанням методу SkeMA. Описано, як адаптер інтегрується в типові трансформерні блоки, не змінюючи їхнього зовнішнього інтерфейсу, та як організовано керування параметрами адаптера й операціями злиття/роззлиття. Система побудована з урахуванням принципів модульності та мікросервісної архітектури, що забезпечує гнучкість і масштабованість рішення: окремі компоненти відповідають за підготовку даних, навчання, зберігання артефактів, моніторинг та розгортання. У процесі реалізації значну увагу приділено читаємості та підтримуваності коду, оскільки добре структуроване програмне рішення знижує когнітивне навантаження на розробників при супроводі складних ML-систем і полегшує подальшу інтеграцію SkeMA у наявні інженерні пайплайни.

У четвертому розділі проведено експериментальну оцінку ефективності розробленого методу на базі попередньо натренованої моделі Pythia-160M та двох принципово різних задач:

– запам’ятовування пар UUID, що моделює здатність мережі швидко засвоювати нові асоціації;

– мовне моделювання на корпусі WikiText-2, яке відображає якість генерації тексту та узгодженість моделі з новим розподілом даних.

Для чесного порівняння SkeMA з базовим методом LoRA обидва підходи було налаштовано з однаковим процентним бюджетом тренованих параметрів

(~0,2% від повної моделі) та ідентичними навчальними налаштуваннями. Результати показали, що SkeMA систематично перевершує LoRA:

– на задачі запам’ятовування UUID-пар SkeMA досягла ~99,7% точності, тоді як LoRA — лише ~85,1%, що емпірично підтверджує вищу виразність оновлень, які реалізує запропонований підхід;

– на WikiText-2 SkeMA забезпечила перплексію 21,9 проти 23,1 у LoRA, наблизившись до якості повного донавчання моделі (20,5).

Таким чином, експериментальні результати однозначно свідчать, що за одного й того самого параметричного бюджету SkeMA забезпечує кращу якість адаптації, зберігаючи при цьому всі переваги параметроефективного навчання та відсутність додаткових обчислень на інференсі.

Практичне значення одержаних результатів полягає в тому, що запропонований метод SkeMA дозволяє організаціям з обмеженими обчислювальними ресурсами ефективніше використовувати великі мовні моделі, зменшуючи потребу в дорогому апаратному забезпеченні та складних інфраструктурних рішеннях. Розроблений підхід може бути інтегрований у наявні робочі процеси розробки ML-систем і доповнений інструментами, які полегшують інженерам процес тонкого налаштування моделей. Дотримання найкращих практик написання зрозумілого коду та систематичне навчання персоналу роботі з такими методами знижують когнітивне навантаження на розробників і сприяють успішному впровадженню технології. Загалом проведене дослідження підтвердило ефективність параметроефективного підходу до навчання великих мовних моделей і показало, що метод SkeMA є перспективним інструментом для адаптації LLM у програмних середовищах з обмеженими обчислювальними ресурсами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Zhang S., Roller S., Goyal N., та ін. OPT: Open Pre-trained Transformer Language Models. arXiv preprint arXiv:2205.01068. 2022. URL: https://arxiv.org/abs/2205.01068 (дата звернення: 10.10.2025)

2. Lialin V., Deshpande V., Yao X., Rumshisky A. Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning. arXiv preprint arXiv:2303.15647. 2023. URL: https://arxiv.org/abs/2303.15647 (дата звернення: 08.10.2025)

3. Biderman D., Portes J., Gonzalez Ortiz J. J. та ін. LoRA Learns Less and Forgets Less. 2024. arXiv preprint arXiv:2405.09673. [Електронний ресурс]. URL: https://arxiv.org/abs/2405.09673 (дата звернення: 11.10.2025

4. Hu E. J., Shen Y., Wallis P. та ін. LoRA: Low-Rank Adaptation of Large Language Models // Proceedings of the International Conference on Learning Representations (ICLR). 2022. [Електронний ресурс]. URL: https://openreview.net/forum?id=nZeVKeeFYf9 (дата звернення: 11.10.2025)

5. Li X. L., Liang P. Prefix-Tuning: Optimizing Continuous Prompts for Generation // Proceedings of the 59-th Annual Meeting of the Association for Computational Linguistics and the 11-th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). 2021. С. 4582–4597. [Електронний ресурс]. URL: https://aclanthology.org/2021.acl-long.353/ (дата звернення: 11.10.2025)

6. Jurafsky D., Martin J. H. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models. 3-тє вид. Онлайн-рукопис. 2025. [Електронний ресурс]. URL: https://web.stanford.edu/~jurafsky/slp3/ (дата звернення: 12.10.2025)

7. Zhao W. X., Zhou K., Li J. та ін. A Survey of Large Language Models. 2023. arXiv preprint arXiv:2303.18223. [Електронний ресурс]. URL: https://arxiv.org/abs/2303.18223 (дата звернення: 12.10.2025)

8. Minaee S., Mikolov T., Nikzad N. та ін. Large Language Models: A Survey. 2024. arXiv preprint arXiv:2402.06196. [Електронний ресурс]. URL: https://arxiv.org/abs/2402.06196 (дата звернення: 12.10.2025)

9. Tunstall L., von Werra L., Wolf T. Natural Language Processing with Transformers. Sebastopol (CA): O'Reilly Media, 2022. 408 с.

10. OpenAI. GPT-4 Technical Report. 2023. arXiv preprint arXiv:2303.08774. [Електронний ресурс]. URL: https://arxiv.org/abs/2303.08774 (дата звернення: 12.10.2025)

11. Raeini M. G. The Evolution of Language Models: From N-Grams to LLMs, and Beyond // SSRN Electronic Journal. 2023. 14 с. [Електронний ресурс]. URL: https://ssrn.com/abstract=4625356 (дата звернення: 13.10.2025)

12. Shen T., Jin R., Huang Y. та ін. Large Language Model Alignment: A Survey. 2023. arXiv preprint arXiv:2309.15025. [Електронний ресурс]. URL: https://arxiv.org/abs/2309.15025 (дата звернення: 13.10.2025)

13. Wang Y., Zhong W., Li L. та ін. Aligning Large Language Models with Human: A Survey. 2023. arXiv preprint arXiv:2307.12966. [Електронний ресурс]. URL: https://arxiv.org/abs/2307.12966 (дата звернення: 14.10.2025)

14. Gururangan S., Marasović A., Swayamdipta S. та ін. Don't Stop Pretraining: Adapt Language Models to Domains and Tasks // Proceedings of the 58-th Annual Meeting of the Association for Computational Linguistics (ACL 2020). 2020. [Електронний ресурс]. URL: https://aclanthology.org/2020.acl-main.740.pdf (дата звернення: 14.10.2025)

15. Lewis P., Perez E., Piktus A., Petroni F., Karpukhin V., та ін. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv preprint arXiv:2005.11401. 2020. 19 p. URL: https://arxiv.org/pdf/2005.11401 (дата звернення: 15.10.2025)

16. Databricks. Building RAG Applications for Production with the Databricks Platform. Sebastopol, CA: O'Reilly Media, 2024. 116 p.

17. Gao Y., Xiong Y., Gao X., Liu K., та ін. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv preprint arXiv:2312.10997. 2024. URL: https://arxiv.org/abs/2312.10997 (дата звернення: 15.10.2025)

18. Xu L., Xie H., Qin S.-Z. J., Tao X., Wang F. L. Parameter-Efficient Fine-Tuning Methods for Pretrained Language Models: A Critical Review and Assessment. arXiv preprint arXiv:2312.12148. 2023. URL: https://arxiv.org/abs/2312.12148 (дата звернення: 16.10.2025)

19. Lialin, V., Deshpande, V., Yao X., та ін. Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning. 2023. arXiv preprint arXiv:2303.15647 [Електронний ресурс]. URL: https://arxiv.org/abs/2303.15647 (дата звернення: 17.10.2025)

20. Valipour M., Rezagholizadeh M., Kobyzev I., Ghodsi A. DyLoRA: Parameter-Efficient Tuning of Pre-trained Models Using Dynamic Search-Free Low-Rank Adaptation. arXiv preprint arXiv:2210.07558. 2022. URL: https://arxiv.org/abs/2210.07558 (дата звернення: 17.10.2025)

21. Zhang Q., Chen M., Bukharin A. et al. AdaLoRA: Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning. International Conference on Learning Representations (ICLR). 2023. URL: https://arxiv.org/abs/2303.10512 (дата звернення: 17.10.2025)

22. Jiang, T., Huang, S., Luo S., та ін. MoRA: High-Rank Updating for Parameter-Efficient Fine-Tuning. 2024. arXiv preprint arXiv:2405.12130 [Електронний ресурс]. URL: https://arxiv.org/abs/2405.12130 (дата звернення: 18.10.2025)

23. Liu S.-Y., Wang C.-Y., Yin H., Molchanov P., Wang Y.-C. F., Cheng K.-T., Chen M.-H. DoRA: Weight-Decomposed Low-Rank Adaptation. arXiv preprint arXiv:2402.09353. 2024. URL: https://arxiv.org/abs/2402.09353 (дата звернення: 18.10.2025)

24. Wen Y., Chaudhuri S. Batched Low-Rank Adaptation of Foundation Models. 2024. arXiv preprint arXiv:2312.05677 [Електронний ресурс]. URL: https://arxiv.org/abs/2312.05677 (дата звернення: 19.10.2025)

25. Kaushal S. What is QLoRA? // Analytics Vidhya Community. 19 лип. 2023 [Електронний ресурс]. URL: https://community.analyticsvidhya.com/c/generative-ai-tech-discussion/what-is-qlora (дата звернення: 20.10.2025)

26. Hu E. J., Shen Y., Wallis P., та ін. LoRA: Low-Rank Adaptation of Large Language Models. International Conference on Learning Representations. 2022. URL: https://openreview.net/forum?id=nZeVKeeFYf9 (дата звернення: 10.10.2025)

27. Zhang R., Han J., Liu C., та ін. LLaMA-Adapter: Efficient Fine-tuning of Language Models with Zero-init Attention // arXiv preprint arXiv:2303.16199. 2024. [Електронний ресурс]. URL: https://arxiv.org/abs/2303.16199 (дата звернення: 14.12.2025)

28. Han Z., Gao C., Liu J., Zhang J., Zhang S. Q. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey // arXiv preprint arXiv:2403.14608. 2024. [Електронний ресурс]. URL: https://arxiv.org/abs/2403.14608 (дата звернення: 14.12.2025)

29. Guo C., Wu Y., Chang Y. NLoRA: Nyström-Initiated Low-Rank Adaptation for Large Language Models // arXiv preprint arXiv:2502.14482. 2025. [Електронний ресурс]. URL: https://arxiv.org/abs/2502.14482 (дата звернення: 15.12.2025)

30. Lou Y., Ye Z., Chen M. Parameter-Efficient Subspace Optimization for LLM Fine-Tuning // arXiv preprint arXiv:2512.02216. 2025. [Електронний ресурс]. URL: https://arxiv.org/abs/2512.02216 (дата звернення: 15.12.2025)

31. Zhang T., Su J., Desai A., та ін. Sketch to Adapt: Fine-Tunable Sketches for Efficient LLM Adaptation // arXiv preprint arXiv:2410.06364. 2024. [Електронний ресурс]. URL: https://arxiv.org/abs/2410.06364 (дата звернення: 15.12.2025)

32. Kopiczko D. J., Blankevoort T., Asano Y. M. VeRA: Vector-based Random Matrix Adaptation // Proceedings of the International Conference on Learning Representations (ICLR). 2024. [Електронний ресурс]. URL: https://openreview.net/forum?id=NjNfLdxr3A (дата звернення: 15.12.2025)

ДОДАТОК А

(обов'язковий)

ПРОГРАМНИЙ КОД

```
from dataclasses import dataclass
from typing import Iterable, Dict, List, Tuple, Optional, Literal

import math
import random

import numpy as np
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader

from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import LoraConfig as HfLoraConfig, TaskType, PeftModel, get_peft_model
from datasets import load_dataset


# SkeMA adapter implementation


@dataclass
class SkeMAConfig:
    """Configuration for the SkeMA adapter.

    Attributes:
        rank: Low-rank dimension r.
        sketch_dim: Sketch dimension k.
        alpha: Scaling factor applied to the adapter update.
        trainable_sketch: If True, S_in and S_out are trainable.
            Otherwise they are fixed random matrices.
        init_scale: Standard deviation for normal initialization of A and B.
    """
    rank: int = 8
    sketch_dim: int = 64
    alpha: float = 1.0
    trainable_sketch: bool = False
    init_scale: float = 1e-3


class SkeMALinear(nn.Module):
    """SkeMA adapter for a single nn.Linear layer.

    Given a frozen base linear layer W_o, this adapter applies the update:

        y = W_o x + alpha * S_out^T * (A * (B * (S_in^T * x)))

    Shapes:
        x:      (..., d_in)
        W_o:    (d_out, d_in)
        S_in:   (d_in, k)
        B:      (k, r)
```

```
        A:      (r, k)
        S_out:  (d_out, k)
    """

    def __init__(
        self,
        base_layer: nn.Linear,
        rank: int,
        sketch_dim: int,
        alpha: float = 1.0,
        trainable_sketch: bool = False,
        init_scale: float = 1e-3,
    ) -> None:
        super().__init__()

        if not isinstance(base_layer, nn.Linear):
            raise TypeError(
                f"SkeMALinear expects nn.Linear, got {type(base_layer)}")

        self.in_features = base_layer.in_features
        self.out_features = base_layer.out_features
        self.rank = rank
        self.sketch_dim = sketch_dim
        self.alpha = alpha

        # Store and freeze the pretrained linear layer.
        self.base = base_layer
        for p in self.base.parameters():
            p.requires_grad = False

        # Ensure new tensors are on the same device/dtype as base weights.
        weight = self.base.weight
        device = weight.device
        dtype = weight.dtype

        # Sketch matrices S_in (d_in × k) and S_out (d_out × k).
        # By default they are fixed random projections.
        s_in = torch.randn(self.in_features, self.sketch_dim, device=device,
dtype=dtype) / math.sqrt(
            self.sketch_dim
        )
        s_out = torch.randn(self.out_features, self.sketch_dim, device=device,
dtype=dtype) / math.sqrt(
            self.sketch_dim
        )

        if trainable_sketch:
            self.S_in = nn.Parameter(s_in)
            self.S_out = nn.Parameter(s_out)
        else:
            # Buffers are not updated by the optimizer.
            self.register_buffer("S_in", s_in)
            self.register_buffer("S_out", s_out)

        # Low-rank factors in the sketched space: B ∈ R^{k×r}, A ∈ R^{r×k}.
        self.B = nn.Parameter(torch.empty(
            self.sketch_dim, self.rank, device=device, dtype=dtype))
```

```
        self.A = nn.Parameter(torch.empty(
            self.rank, self.sketch_dim, device=device, dtype=dtype))

        nn.init.normal_(self.B, std=init_scale)
        nn.init.normal_(self.A, std=init_scale)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Base frozen output (keeps full original behavior).
        base_out = self.base(x)

        # Flatten leading dims except feature dimension.
        orig_shape = x.shape  # (..., d_in)
        x_flat = x.reshape(-1, self.in_features)  # (N, d_in)

        # Compress to sketch_dim: (N, d_in) → (N, k)
        x_sketched = x_flat @ self.S_in  # (N, k)

        # Low-rank transform: (N, k) → (N, r) → (N, k)
        h = x_sketched @ self.B          # (N, r)
        h = h @ self.A                   # (N, k)

        # Expand back to output space via S_outᵀ: (N, k) → (N, d_out)
        update_flat = h @ self.S_out.t()  # (N, d_out)

        # Reshape back to original leading dims.
        update = update_flat.view(
            *orig_shape[:-1], self.out_features)  # (..., d_out)

        return base_out + self.alpha * update


def add_skema_to_model(
    model: nn.Module,
    config: SkeMAConfig,
    target_substrings: Iterable[str],
) -> Dict[str, SkeMALinear]:
    """Replace selected nn.Linear modules with SkeMALinear adapters.

    Args:
        model: Base model whose linear layers will be wrapped.
        config: SkeMA configuration.
        target_substrings: a list/iterable of name substrings. If a linear
            layer's qualified name contains any substring, it will be wrapped.

    Returns:
        Mapping from qualified parameter names to created SkeMALinear modules.
    """
    replace_points = []

    # First collect all replacement points to avoid mutating while iterating.
    for module_name, module in model.named_modules():
        for child_name, child in module.named_children():
            qual_name = ".".join([module_name, child_name]).strip(".")
            if isinstance(child, nn.Linear) and any(s in qual_name for s in
target_substrings):
                replace_points.append((module, child_name, child, qual_name))
```

```
    replaced: Dict[str, SkeMALinear] = {}

    for module, child_name, child, qual_name in replace_points:
        skema_layer = SkeMALinear(
            base_layer=child,
            rank=config.rank,
            sketch_dim=config.sketch_dim,
            alpha=config.alpha,
            trainable_sketch=config.trainable_sketch,
            init_scale=config.init_scale,
        )
        setattr(module, child_name, skema_layer)
        replaced[qual_name] = skema_layer

    return replaced


def mark_only_skema_trainable(model: nn.Module) -> None:
    """Freeze all parameters except those of SkeMALinear adapters."""
    # Freeze everything.
    for p in model.parameters():
        p.requires_grad = False

    # Unfreeze only SkeMA adapter parameters (but keep base weights frozen).
    for m in model.modules():
        if isinstance(m, SkeMALinear):
            for name, p in m.named_parameters():
                if name.startswith("base."):
                    # Stay frozen.
                    p.requires_grad = False
                else:
                    p.requires_grad = True


def create_skema_model(
    model_name: str,
    config: SkeMAConfig,
    device_map: str = "auto",
    torch_dtype: torch.dtype = torch.bfloat16,
    target_substrings: Iterable[str] = (
        "q_proj", "k_proj", "v_proj", "o_proj"),
) -> nn.Module:
    """Load a base causal LM and attach SkeMA adapters to selected layers."""
    base_model = AutoModelForCausalLM.from_pretrained(
        model_name,
        device_map=device_map,
        torch_dtype=torch_dtype,
    )

    add_skema_to_model(base_model, config, target_substrings)
    mark_only_skema_trainable(base_model)

    return base_model

# LoRA baseline using PEFT
```

```python
@dataclass
class LoRABaselineConfig:
    """Configuration for the LoRA baseline using Hugging Face PEFT."""

    # Rank of the low-rank update
    rank: int = 8

    # LoRA scaling factor (α)
    alpha: int = 16

    # Dropout on the LoRA branch
    dropout: float = 0.05

    # Names of modules to which LoRA will be applied
    target_modules: Iterable[str] = ("q_proj", "k_proj", "v_proj", "o_proj")

    # Bias handling: "none" | "all" | "lora_only"
    bias: str = "none"

    # Task type (we only use CAUSAL_LM)
    task_type: str = "CAUSAL_LM"

    # Optional: stored in config for reproducibility
    base_model_name_or_path: Optional[str] = None

    # Optional: which full modules to save in addition to LoRA weights
    modules_to_save: Optional[List[str]] = None

    def to_hf_config(self) -> HfLoraConfig:
        """Convert to the official PEFT LoraConfig."""
        task_type_enum = TaskType[self.task_type]
        return HfLoraConfig(
            r=self.rank,
            lora_alpha=self.alpha,
            lora_dropout=self.dropout,
            target_modules=list(self.target_modules),
            bias=self.bias,
            task_type=task_type_enum,
            base_model_name_or_path=self.base_model_name_or_path,
            modules_to_save=self.modules_to_save,
        )


def count_parameters(model: nn.Module) -> Tuple[int, int, float]:
    """Return (trainable, total, trainable_percent)."""
    total = 0
    trainable = 0

    for p in model.parameters():
        n = p.numel()
        total += n
        if p.requires_grad:
            trainable += n

    percent = (trainable / total) * 100.0 if total > 0 else 0.0
    return trainable, total, percent
```

```python
def create_lora_model(
    model_name: str,
    config: LoRABaselineConfig,
    device_map: str = "auto",
    torch_dtype: torch.dtype = torch.bfloat16,
) -> PeftModel:
    """Load a base causal LM and wrap it with a LoRA adapter using PEFT."""
    base_model = AutoModelForCausalLM.from_pretrained(
        model_name,
        device_map=device_map,
        torch_dtype=torch_dtype,
    )

    if config.base_model_name_or_path is None:
        config.base_model_name_or_path = model_name

    hf_config = config.to_hf_config()
    lora_model = get_peft_model(base_model, hf_config)
    return lora_model


def load_lora_adapter(
    model_name: str,
    adapter_path: str,
    device_map: str = "auto",
    torch_dtype: torch.dtype = torch.bfloat16,
) -> PeftModel:
    """Load a frozen base model and attach a trained LoRA adapter from disk."""
    base_model = AutoModelForCausalLM.from_pretrained(
        model_name,
        device_map=device_map,
        torch_dtype=torch_dtype,
    )
    return PeftModel.from_pretrained(
        base_model,
        adapter_path,
        device_map=device_map,
    )

# Experiment definitions


AdapterMethod = Literal["none", "skema", "lora"]
TaskName = Literal["uuid_mem", "wikitext2"]


@dataclass
class ExperimentConfig:
    model_name: str = "EleutherAI/pythia-160m"
    adapter: AdapterMethod = "skema"
    task: TaskName = "uuid_mem"

    max_train_steps: int = 1000
    eval_every_steps: int = 100
    gradient_accumulation_steps: int = 1

    learning_rate: float = 1e-4
```

```
    weight_decay: float = 0.0

    batch_size: int = 8
    seq_len: int = 128

    device: str = "cuda" if torch.cuda.is_available() else "cpu"
    seed: int = 42

    # UUID memorization-specific
    uuid_count: int = 128
    uuid_train_copies: int = 8
    uuid_eval_copies: int = 4


# Datasets

class UuidMemDataset(Dataset):
    """Dataset for the UUID memorization task."""

    def __init__(
        self,
        tokenizer: AutoTokenizer,
        uuids: List[str],
        copies: int,
        max_length: int,
    ) -> None:
        texts: List[str] = []
        for _ in range(copies):
            texts.extend(uuids)

        enc = tokenizer(
            texts,
            padding="max_length",
            truncation=True,
            max_length=max_length,
            return_attention_mask=True,
        )

        self.input_ids = torch.tensor(enc["input_ids"], dtype=torch.long)
        self.attention_mask = torch.tensor(
            enc["attention_mask"], dtype=torch.long)

    def __len__(self) -> int:
        return self.input_ids.size(0)

    def __getitem__(self, idx: int) -> Dict[str, torch.Tensor]:
        input_ids = self.input_ids[idx]
        attention_mask = self.attention_mask[idx]
        labels = input_ids.clone()
        return {
            "input_ids": input_ids,
            "attention_mask": attention_mask,
            "labels": labels,
        }


def generate_random_uuids(n: int) -> List[str]:
```

```
    alphabet = "0123456789abcdef"
    uuids: List[str] = []
    for _ in range(n):
        parts = [
            "".join(random.choice(alphabet) for _ in range(length))
            for length in (8, 4, 4, 4, 12)
        ]
        uuids.append("-".join(parts))
    return uuids


def build_uuid_mem_dataloaders(
    cfg: ExperimentConfig,
    tokenizer: AutoTokenizer,
) -> Tuple[DataLoader, DataLoader]:
    uuids = generate_random_uuids(cfg.uuid_count)

    train_ds = UuidMemDataset(
        tokenizer=tokenizer,
        uuids=uuids,
        copies=cfg.uuid_train_copies,
        max_length=cfg.seq_len,
    )
    eval_ds = UuidMemDataset(
        tokenizer=tokenizer,
        uuids=uuids,
        copies=cfg.uuid_eval_copies,
        max_length=cfg.seq_len,
    )

    train_loader = DataLoader(
        train_ds,
        batch_size=cfg.batch_size,
        shuffle=True,
        drop_last=True,
    )
    eval_loader = DataLoader(
        eval_ds,
        batch_size=cfg.batch_size,
        shuffle=False,
        drop_last=False,
    )

    return train_loader, eval_loader


def build_wikitext2_dataloaders(
    cfg: ExperimentConfig,
    tokenizer: AutoTokenizer,
) -> Tuple[DataLoader, DataLoader]:
    raw = load_dataset("wikitext", "wikitext-2-raw-v1")

    def tokenize_fn(batch):
        return tokenizer(batch["text"])

    tokenized = raw.map(
        tokenize_fn,
```

```
        batched=True,
        remove_columns=["text"],
    )

    def group_texts(batch):
        concatenated = {k: sum(batch[k], []) for k in batch}
        total_length = len(concatenated["input_ids"])
        total_length = (total_length // cfg.seq_len) * cfg.seq_len

        result: Dict[str, List[List[int]]] = {}
        for k, t in concatenated.items():
            t = t[:total_length]
            result[k] = [
                t[i: i + cfg.seq_len]
                for i in range(0, total_length, cfg.seq_len)
            ]

        result["labels"] = result["input_ids"].copy()
        return result

    lm_ds = tokenized.map(group_texts, batched=True)

    train_ds = lm_ds["train"]
    eval_ds = lm_ds["validation"]

    def collate(batch):
        keys = batch[0].keys()
        out = {
            k: torch.tensor([b[k] for b in batch], dtype=torch.long)
            for k in keys
        }
        return out

    train_loader = DataLoader(
        train_ds,
        batch_size=cfg.batch_size,
        shuffle=True,
        drop_last=True,
        collate_fn=collate,
    )
    eval_loader = DataLoader(
        eval_ds,
        batch_size=cfg.batch_size,
        shuffle=False,
        drop_last=False,
        collate_fn=collate,
    )

    return train_loader, eval_loader


def build_dataloaders(
    cfg: ExperimentConfig,
    tokenizer: AutoTokenizer,
) -> Tuple[DataLoader, DataLoader]:
    if cfg.task == "uuid_mem":
        return build_uuid_mem_dataloaders(cfg, tokenizer)
```

```
    elif cfg.task == "wikitext2":
        return build_wikitext2_dataloaders(cfg, tokenizer)
    else:
        raise ValueError(f"Unknown task {cfg.task}")


# Model factory + evaluation

def build_model(cfg: ExperimentConfig) -> nn.Module:
    if cfg.adapter == "skema":
        skema_cfg = SkeMAConfig()
        model = create_skema_model(
            model_name=cfg.model_name,
            config=skema_cfg,
            device_map="auto",
            torch_dtype=torch.bfloat16,
        )
    elif cfg.adapter == "lora":
        lora_cfg = LoRABaselineConfig()
        model = create_lora_model(
            model_name=cfg.model_name,
            config=lora_cfg,
            device_map="auto",
            torch_dtype=torch.bfloat16,
        )
    elif cfg.adapter == "none":
        model = AutoModelForCausalLM.from_pretrained(
            cfg.model_name,
            torch_dtype=torch.bfloat16,
        )
        model.to(cfg.device)
    else:
        raise ValueError(f"Unknown adapter {cfg.adapter}")
    return model


@torch.no_grad()
def evaluate_lm_loss(
    model: nn.Module,
    dataloader: DataLoader,
    device: str,
) -> Tuple[float, float]:
    model.eval()
    total_loss = 0.0
    total_tokens = 0

    for batch in dataloader:
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch.get("attention_mask")
        if attention_mask is not None:
            attention_mask = attention_mask.to(device)
        labels = batch["labels"].to(device)

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            labels=labels,
```

```
        )
        loss = outputs.loss

        if attention_mask is not None:
            tokens = attention_mask.sum().item()
        else:
            tokens = labels.numel()

        total_loss += loss.item() * tokens
        total_tokens += tokens

    mean_loss = total_loss / max(total_tokens, 1)
    perplexity = math.exp(mean_loss)

    model.train()
    return mean_loss, perplexity


@torch.no_grad()
def evaluate_uuid_exact_match(
    model: nn.Module,
    dataloader: DataLoader,
    device: str,
) -> float:
    model.eval()
    correct = 0
    total = 0

    for batch in dataloader:
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["labels"].to(device)

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
        )
        preds = outputs.logits.argmax(dim=-1)

        for i in range(input_ids.size(0)):
            mask_i = attention_mask[i].bool()
            if torch.equal(labels[i][mask_i], preds[i][mask_i]):
                correct += 1
            total += 1

    acc = correct / max(total, 1)
    model.train()
    return acc


# Utilities and training loop

def set_seed(seed: int) -> None:
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
```

```python
def train_one_experiment(cfg: ExperimentConfig) -> Dict[str, float]:
    set_seed(cfg.seed)

    tokenizer = AutoTokenizer.from_pretrained(cfg.model_name, use_fast=True)
    if tokenizer.pad_token_id is None:
        tokenizer.pad_token = tokenizer.eos_token

    train_loader, eval_loader = build_dataloaders(cfg, tokenizer)

    model = build_model(cfg)
    model.to(cfg.device)

    optimizer = torch.optim.AdamW(
        filter(lambda p: p.requires_grad, model.parameters()),
        lr=cfg.learning_rate,
        weight_decay=cfg.weight_decay,
    )

    global_step = 0
    running_loss = 0.0
    metrics: Dict[str, float] = {}

    model.train()

    while global_step < cfg.max_train_steps:
        for batch in train_loader:
            global_step += 1

            input_ids = batch["input_ids"].to(cfg.device)
            attention_mask = batch.get("attention_mask")
            if attention_mask is not None:
                attention_mask = attention_mask.to(cfg.device)
            labels = batch["labels"].to(cfg.device)

            outputs = model(
                input_ids=input_ids,
                attention_mask=attention_mask,
                labels=labels,
            )
            loss = outputs.loss / cfg.gradient_accumulation_steps
            loss.backward()

            running_loss += loss.item()

            if global_step % cfg.gradient_accumulation_steps == 0:
                optimizer.step()
                optimizer.zero_grad()

            if (
                global_step % cfg.eval_every_steps == 0
                or global_step == cfg.max_train_steps
            ):
                avg_train_loss = running_loss / cfg.eval_every_steps
                running_loss = 0.0
```

```
                eval_loss, eval_ppl = evaluate_lm_loss(
                    model, eval_loader, cfg.device
                )

                metrics["step"] = global_step
                metrics["train_loss"] = avg_train_loss
                metrics["eval_loss"] = eval_loss
                metrics["eval_ppl"] = eval_ppl

                if cfg.task == "uuid_mem":
                    uuid_acc = evaluate_uuid_exact_match(
                        model, eval_loader, cfg.device
                    )
                    metrics["uuid_exact_match"] = uuid_acc

            if global_step >= cfg.max_train_steps:
                break

    return metrics


# Example usage

if __name__ == "__main__":
    cfg = ExperimentConfig(
        model_name="EleutherAI/pythia-160m",
        adapter="skema",           # "skema", "lora", or "none"
        task="uuid_mem",           # "uuid_mem" or "wikitext2"
        max_train_steps=900,
        eval_every_steps=100,
        batch_size=16,
        seq_len=64,
        learning_rate=1e-4,
    )

    results = train_one_experiment(cfg)
    print(results)
```

ДОДАТОК Б

(обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

Метод тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами

Виконала:
Студентка 2 курсу, група ІПЗм-24-1
Ваховська В. М.

Керівник:
д-р фіз.-мат. наук, професор
Бедратюк Л. П.

Хмельницький 2025

Актуальність теми

Великі мовні моделі активно впроваджуються в сучасні програмні системи.

Для реального використання їх потрібно адаптувати під предметну область, мову та вимоги замовника → потрібне тонке налаштування.

Повне тонке налаштування сотень мільйонів параметрів вимагає дорогих обчислювальних ресурсів, часу та спецобладнання, що обмежує використання в умовах обмежених ресурсів.

Існуючі PEFT-підходи хоч і зменшують обсяг обчислень, але або ускладнюють модель, або сповільнюють її, або гірше засвоюють нові знання.

Тому актуально розробити параметроефективний метод тонкого налаштування LLM (SkeMA), який забезпечує високу якість адаптації без ускладнення інференсу.

Мета та завдання дослідження

Метою даної роботи є підвищення ефективності тонкого налаштування великих мовних моделей шляхом розробки та дослідження нового параметроефективного методу SkeMA (Sketched Matrix Adapter).

Для досягнення поставленої мети необхідно вирішити такі задачі:

− проаналізувати існуючі підходи до тонкого налаштування LLM та параметроефективні методи (PEFT);

− визначити вимоги до методу тонкого налаштування, орієнтованого на середовища з обмеженими обчислювальними ресурсами;

− розробити математичну модель та архітектуру методу SkeMA;

− здійснити програмну реалізацію розробленого методу та інтегрувати його в типову архітектуру мовної моделі;

− спроєктувати та провести експерименти з порівняння SkeMA з методом LoRA на репрезентативних завданнях;

− виконати аналіз отриманих результатів та оцінити практичну придатність запропонованого методу.

Об’єкт і предмет дослідження

Об’єкт дослідження – процес тонкого налаштування великих мовних моделей у програмних середовищах з обмеженими обчислювальними ресурсами.

Предмет дослідження – метод SkeMA (Sketched Matrix Adapter) для параметроефективного тонкого налаштування великих мовних моделей, його архітектура, особливості реалізації та поведінка у порівнянні з іншими методами налаштування.

Методи дослідження

1. Аналіз літературних джерел: огляд сучасних робіт з тонкого налаштування LLM та параметроефективних методів (адаптери, квантування) для виявлення їхніх переваг і обмежень.

2. Аналітичні методи: виконано формалізацію методу SkeMA, визначено структуру адаптера, параметровий бюджет та принципи його інтеграції в модель.

3. Експериментальні методи: проведено експерименти з навчання та порівняння методу SkeMA з методом LoRA на вибраних задачах мовного моделювання та запам’ятовування.

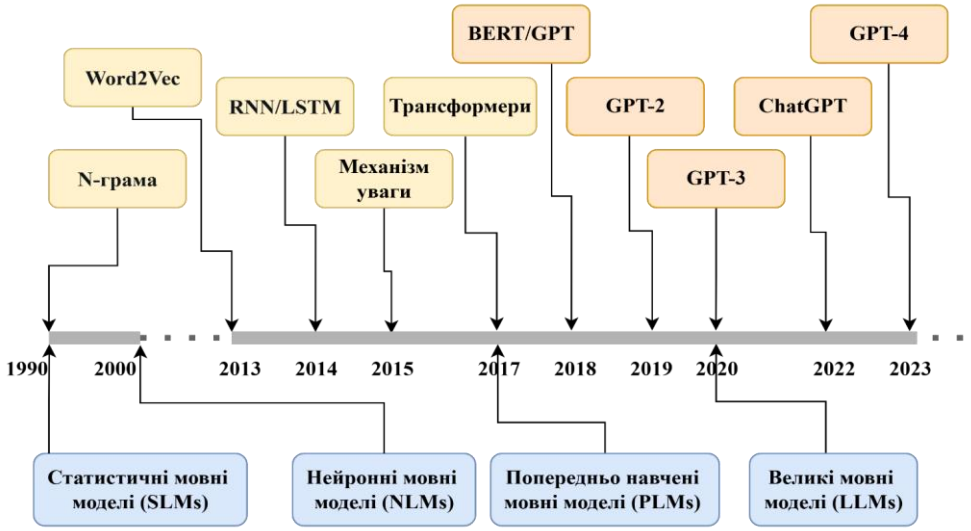
4. Методи комп’ютерного моделювання: реалізовано та протестовано програмну реалізацію методу SkeMA з використанням сучасних бібліотек машинного навчання.

Наукова новизна

Вперше запропоновано метод параметроефективного тонкого налаштування великих мовних моделей SkeMA (Sketched Matrix Adapter), який, на відміну від існуючих підходів, формує оновлення ваг шляхом навчання компактного параметричного блоку у стислому підпросторі, визначеному фіксованими проєкціями, що дозволяє поєднати параметричну ефективність із високою виразною здатністю.

Розділ 1 – Аналіз предметної області

Проведено аналіз розвитку мовних моделей та підходів до тонкого налаштування.



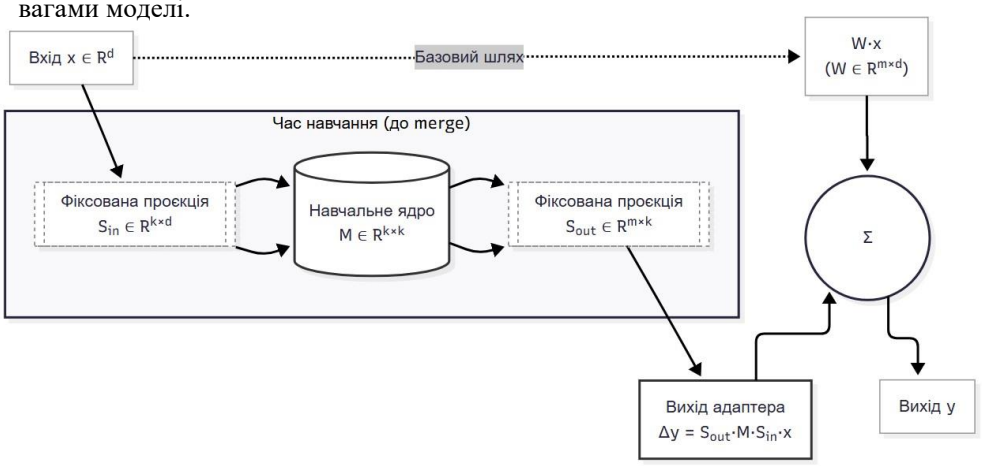
Розділ 1 – Аналіз предметної області

Підхід	Основне обмеження
Повне тонке налаштування	Дуже дорогі обчислення й пам’ять
RAG	Не змінює модель, залежить від бази знань
Класичні адаптери	Додають затримку на інференсі, потрібно керувати окремими модулями
Prefix-/Prompt-tuning	Дорожче для довгих промптів, чутливі до довжини префіксу
Селективне FT (BitFit)	Слабка виразність, працює не для всіх задач
Low-rank (LoRA, AdaLoRA, DoRA)	Обмежені рангом – може не вистачати «простору» для складних оновлень
IA^3, SSF	Ще вужчий канал оновлень, ризик втрати якості

Отже, треба розробити метод тонкого налаштування, який дозволяє зменшити обсяг додаткових параметрів, зберегти або покращити якість адаптації моделі та не створює суттєвого додаткового навантаження під час її використання.

Розділ 2 – Концепція методу SkeMA

Запропоновано та обґрунтовано метод SkeMA (Sketched Matrix Adapter) для параметроефективного тонкого налаштування великих мовних моделей. Основна ідея методу полягає в тому, щоб додати до моделі компактний адаптер спеціальної структури, який навчається у стислому підпросторі ознак, визначеному фіксованими проєкціями, а після завершення навчання повністю поєднується з основними вагами моделі.



Такий підхід дозволяє збільшити гнучкість змін у вагах моделі за фіксованого бюджету додаткових параметрів і водночас уникнути ускладнення архітектури на етапі реального використання, оскільки після злиття адаптер не додає нових обчислень.

Розділ 2 – Властивості SkeMA

SkeMA вводить оновлення ваг через компактний блок у стислому підпросторі, а після навчання це оновлення повністю зливається з базовими вагами. Завдяки цьому обчислення шару зі злитими вагами алгебраїчно еквівалентні обчисленням з активним адаптером, тому в режимі інференсу не з'являється жодних додаткових операцій. Розмірність стислого простору k одночасно задає бюджет тренованих параметрів k^2 і верхню межу складності оновлення, тож місткість адаптера можна явно контролювати.

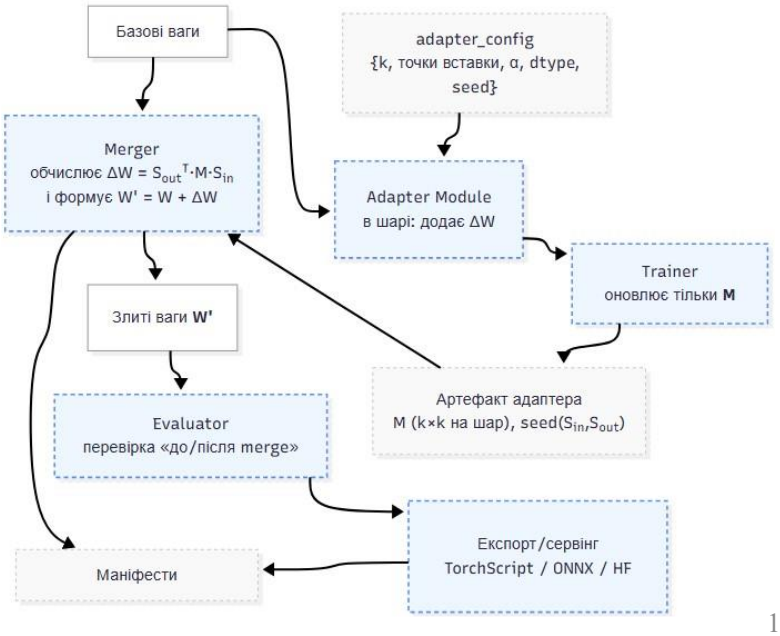
За однакового параметричного бюджету SkeMA може реалізувати оновлення з вищою верхньою межею рангу, ніж LoRA, особливо для великих лінійних шарів (2048–4096 нейронів). Це означає, що SkeMA потенційно описує «багатші» зміни у вагах без збільшення кількості тренованих параметрів, зберігаючи при цьому нульове інференс-навантаження.

Вхідна/Вихідна розмірність ($d = m$)	**Бюджет параметрів (P)**	**$k \approx P$ (SkeMA)**	**$r \approx 2dP$ (LoRA)**	**Співвідношення k/r**
2048	65 536	256	16	16×
2048	262 144	512	64	8×
4096	65 536	256	8	32×
4096	262 144	512	32	16×
4096	1 048 576	1024	128	8×

Розділ 3 – Проєктування та програмна реалізація

Розроблено модуль адаптера, який може бути підключений до вибраних шарів мовної моделі, а також модулі навчання та злиття оновлених ваг з базовою моделлю.

Особливістю реалізації є те, що метод SkeMA інтегрується в існуючі інструменти машинного навчання та може використовуватися разом із типовими конвеєрами підготовки, навчання та збереження моделей. Після виконання злиття модель отримує оновлені ваги, але зберігає початкову структуру, що спрощує її подальше розгортання.



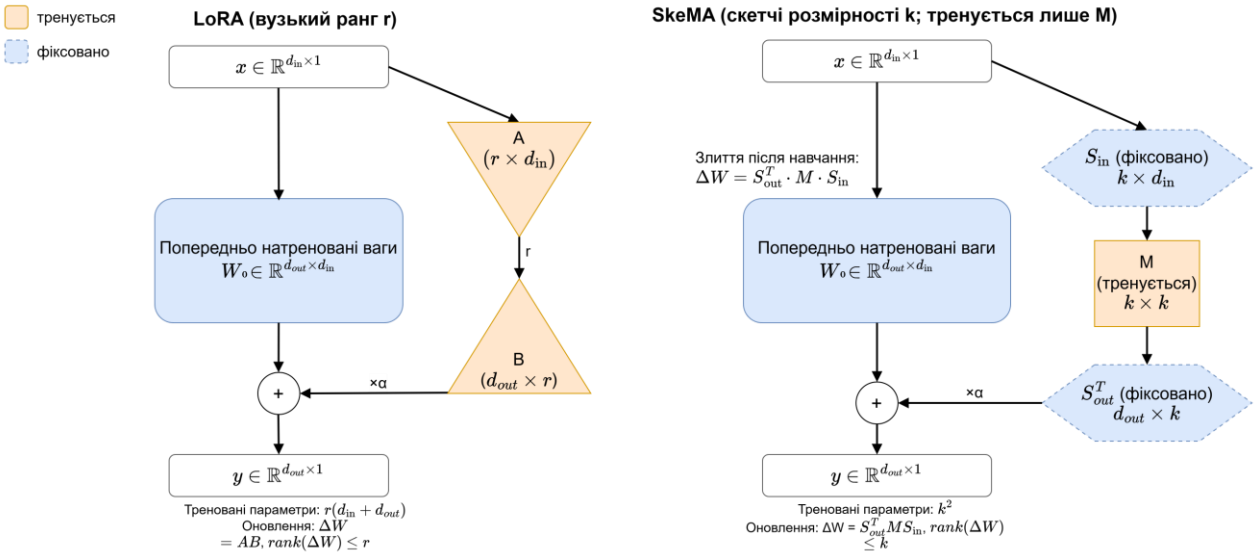
Розділ 3 – Інтеграція та сценарії використання

Обґрунтовано диференційований підхід до вибору конфігурації адаптера: від «легких» варіантів, оптимізованих для швидкого експериментування та перевірки гіпотез, до ресурсомістких конфігурацій, спрямованих на максимізацію результатів.

Показано, що завдяки механізму повного злиття параметрів адаптера з базовими вагами, сформований артефакт моделі набуває властивостей автономності. Це дозволяє здійснювати його розгортання в середовищах з обмеженими обчислювальними ресурсами без необхідності модифікації програмного коду системи інференсу.

Розділ 4 – Експериментальні дослідження

Для порівняння було обрано задачу мовного моделювання на текстовому корпусі та задачу запам’ятовування штучно згенерованих даних.

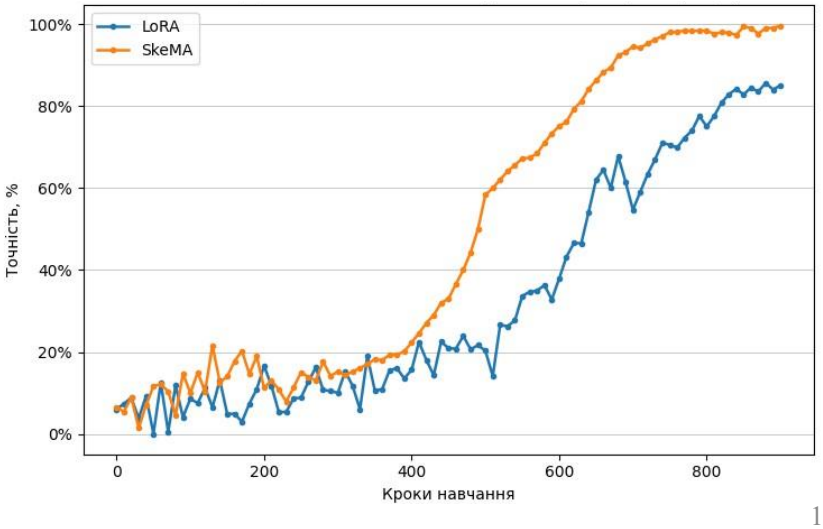


Розділ 4 – Порівняння SkeMA та LoRA

Першим було завдання із запам’ятовування пар UUID. Моделі було надано набір з 1000 унікальних пар вигляду <ключ, значення> , де і ключ, і значення – це рядки у форматі UUID (32-шістнадцяткові символи).

Кроки навчання	LoRA (точність)	SkeMA (точність)
300	10.00%	15.20%
500	20.30%	58.40%
700	54.70%	94.60%
900	85.10%	99.70%

Підхід LoRA є вузьким місцем для запам’ятовування нових, довільних асоціацій. Натомість SkeMA, маючи вищу ефективну розмірність оновлення (ранг $\leq k$), позбавлена цього обмеження.

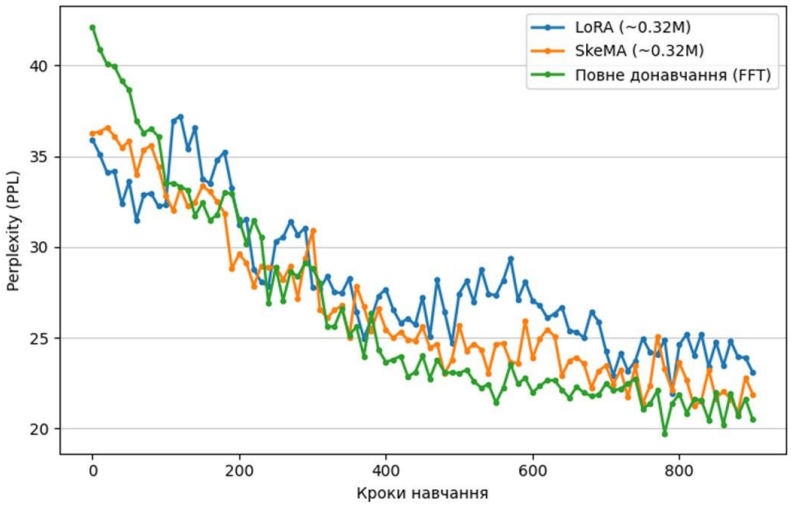


Розділ 4 – Порівняння SkeMA та LoRA

Другим був проведений експеримент із мовного моделювання на стандартному корпусі WikiText-2, що містить близько 2 млн слів із статей Вікіпедії. Модель має прогнозувати ймовірність наступного слова в тексті.

Метод тонкого налаштування	Навчувані параметри	Тестова перплексія
Повне донавчання (FFT)	160 млн (100%)	20.5
LoRA (ранг $r = 8$)	~0.32 млн (~0.2%)	23.1
SkeMA (новий підхід, ~0.2%)	~0.32 млн (~0.2%)	21.9

Низькорангові оновлення LoRA достатньо добре справляються з адаптаційними завданнями.



Публікації

Ваховська В.М., Праворська Н.І. SkeMA: скетч-орієнтований адаптер для параметроефективного тонкого налаштування великих мовних моделей. Збірник наукових праць за матеріалами XVII Всеукраїнської науково-практичної конференції «Актуальні проблеми комп'ютерних наук АПКН-2025», 14-15 листопада 2025, Хмельницький 2025, стор. 52-56

Ваховська В.М. Метод тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами. Міжнародний науково-технічний журнал «Вимірювальна та обчислювальна техніка в технологічних процесах», 2025 (подано до друку).

Висновки

У ході виконання кваліфікаційної роботи здійснено системний аналіз сучасних підходів до тонкого налаштування великих мовних моделей та ідентифіковано їхні ключові обмеження при розгортанні в умовах лімітованих обчислювальних ресурсів. На основі проведеного аналізу запропоновано та обґрунтовано метод SkeMA, що реалізує параметроефективну адаптацію моделей через використання компактних адаптерів.

Розроблено архітектуру та виконано програмну реалізацію запропонованого методу. Проведено серію експериментальних досліджень на задачах мовного моделювання та запам’ятовування нової інформації. На тесті запам’ятовування SkeMA продемонструвала кардинальну перевагу (99.7% проти 85.1%). У задачах загального мовного моделювання також зафіксовано високу якість адаптації, що наближається до показників повномасштабного донавчання.

Результати роботи мають практичну цінність для розробників, які працюють з великими мовними моделями в умовах обмежених ресурсів, та можуть бути використані як основа для подальших досліджень і вдосконалення методів параметроефективного тонкого налаштування.

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Віри ВАХОВСЬКОЇ
факультет ІТ, 2 курс, група ІПЗм-24-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлена. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщена та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмно-технічного засобу AntiPlagiarism і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1.09.2025
дата

підпис

Tue Dec 02 11:46:03 EET 2025, Форкун Юрій Вікторович, Хмельницький національний університет, ХНУ

Anti-Plagiarism (UA) v-16.663

Максимальне співпадіння з одним документом 7.0%

Словники перевірки: UA, US, RU. **Помилок в документах: 14%**

ID: 251181 Назва: МКР_Метод тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами Додано в БД: 2025-12-02 Автора: Віра Ваховська Керівники: Леонід БЕДРАТЮК, д-р фіз.-мат. наук, професор Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	148676	2207	12607 (8%)	178 (8%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Ваховська Віра Миколаївна.

Співавтор:

Назва: Метод тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами

Науковий керівник: д-р фіз.-мат. наук, професор, Леонід БЕДРАТЮК

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:1.3%

Коефіцієнт подібності 2:0.6%

Мікропробіли: 0

Заміна букв: 57

Інтервали: 0

Білі знаки: 11

Дата створення звіту: 2025-12-02 11:59:49.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата 02.12.25 експерт (Ворощук Ю.В.)

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

освітнього ступеня «магістр»

Магістр Ваховська Віра Миколаївна

Тема Метод тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами

Спеціальність 121 «Інженерія програмного забезпечення»

Обсяг кваліфікаційної роботи:

Кількість сторінок кваліфікаційної роботи 100 .

1. Короткий зміст роботи та прийнятих рішень У кваліфікаційній роботі розглянуто проблему параметроефективного тонкого налаштування великих мовних моделей у програмних середовищах з обмеженими обчислювальними ресурсами. На основі аналізу сучасних підходів запропоновано метод SkeMA, що формує оновлення ваг у стислому підпросторі за допомогою фіксованих проєкцій і після навчання повністю зливається з базовими вагами моделі, тому не створює додаткового навантаження під час інференсу. Розроблено математичну модель та архітектуру методу, створено програмний модуль адаптера в середовищі Python і інтегровано його в трансформерну мовну модель. Проведено експериментальне порівняння SkeMA з методом LoRA для задач мовного моделювання і запам'ятовування асоціативних пар та показано переваги запропонованого підходу за однакового бюджету параметрів.

2. Висновок про відповідність роботи дипломному завданню Кваліфікаційна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню за змістом і структурою. Усі теоретичні та практичні задачі виконані в повному обсязі, отримані результати узгоджуються з метою, об'єктом і предметом дослідження.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи У вступі обґрунтовано актуальність теми, визначено мету, задачі, об'єкт і предмет дослідження, сформульовано наукову новизну та практичну цінність. Перший розділ містить ґрунтовний огляд сучасних методів тонкого налаштування мовних моделей і параметроефективних підходів, аналіз їх переваг і обмежень, що свідчить про добре володіння здобувачкою сучасним науковим контекстом. У другому розділі наведено математичну модель і властивості методу SkeMA, подано теоретичне порівняння з LoRA. Третій розділ присвячений архітектурі програмної реалізації, опису структури модулів і способу інтеграції адаптера в типову трансформерну модель. У четвертому розділі детально описано постановку експериментів, налаштування середовища та проведено аналіз отриманих результатів. У роботі використано сучасні засоби програмування та бібліотеки для машинного навчання.

4. Позитивні сторони роботи Робота відзначається високою актуальністю та чітким практичним спрямуванням. Запропонований метод SkeMA є новим рішенням у галузі параметроефективного тонкого налаштування, поєднує ідеї низькорангових оновлень і матричного скетчування та забезпечує нульове додаткове навантаження в режимі інференсу. Теоретичні результати підтверджені програмною реалізацією і серією експериментів, які наочно представлені у вигляді таблиць і графіків. Матеріал викладено логічно, послідовно і доступно, текст добре структурований.

5. Негативні сторони роботи До недоліків можна віднести обмежений спектр розглянутих моделей і типів задач, що дещо звужує можливість узагальнення результатів. Було б доцільно доповнити експериментальну частину дослідженнями на більших мовних моделях та додаткових сценаріях застосування, а також детальніше проаналізувати виграш за часом виконання та використанням ресурсів. Зазначені зауваження мають рекомендаційний характер і не знижують загальної високої оцінки роботи.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічні матеріали виконані охайно й інформативно, відповідають тематиці роботи і допомагають зрозуміти ключові ідеї та результати. Пояснювальна записка оформлена відповідно до вимог, містить усі необхідні структурні елементи, рисунки, таблиці та коректні посилання на літературні джерела.

7. Відгук про роботу в цілому У цілому кваліфікаційна робота справляє враження завершеного науково-практичного дослідження. У роботі продемонстровано вміння працювати з сучасними науковими джерелами, проєктувати алгоритми і програмні засоби для машинного навчання, проводити коректні експериментальні дослідження та формулювати обґрунтовані висновки. Отримані результати мають теоретичну й прикладну цінність для розгортання великих мовних моделей у системах з обмеженими обчислювальними ресурсами

8.Інші зауваження

9. Оцінка кваліфікаційної роботи З урахуванням актуальності теми, рівня теоретичного опрацювання, повноти виконання поставлених завдань та обґрунтованості отриманих результатів кваліфікаційна робота Ваховської Віри Миколаївни відповідає вимогам, що висуваються до магістерських робіт, і заслуговує на оцінку «відмінно.

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Павлова Ольга Олександрівна, зав. кафедри комп'ютерної інженерії та інформаційних систем ХНУ

5.12.2025
Дата

(Підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів перевірки роботи, продукованими програмно-технічним засобом (ами), на наявність текстових збігів.
Назва кваліфікаційної роботи: «Метод тонкого налаштування великих мовних моделей із використанням оптимізованих адаптерів для розгортання в програмних середовищах з обмеженими обчислювальними ресурсами»
Автор: Ваховська Віра Миколаївна
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Спеціальність: 121 – Інженерія програмного забезпечення
Науковий керівник: Бедратюк Леонід Петрович, д-р фіз.-мат. наук, професор
Після аналізу звіту/звітів зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається до захисту.	**відповідає**
2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована.	
3	Перевищено граничне значення рівня подібностей. Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнуті. Робота може бути допущена до захисту після того, як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:
Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки:
1) у тексті кваліфікаційної роботи системою перевірки на плагіат StrickePlagiarism виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, у назвах розділів/підрозділів, у назвах публікацій переліку джерел посилання тощо;
2) в якості запозичень системою StrickePlagiarism було зафіксовано деякі послідовності вихідного коду і посилання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;
3) запозичення, виявлені в тексті роботи, є фрагментарними або мають належним чином оформлені посилання;
4) виявлені модифікації тексту не впливають на відсоток схожості.
Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає 7 %. За системою StrickePlagiarism коефіцієнт подібності 1 складає 1.3 %, коефіцієнт подібності 2 складає 0.0%.

Дата 02.12.2025

Завідувач кафедри _____ Леонід БЕДРАТЮК
Гарант освітньої програми _____ Оксана ЯШИНА
Керівник кваліфікаційної роботи _____ Леонід БЕДРАТЮК