НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ

І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

_комп'ютерних наук_____

(назва кафедри)

_____ _Голуб Б.Л.,доц.,к.т.н._

(підпис) (ПІБ)

"____" червня 2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему

«Розробка моделі для доповнення коду на основі LSTM і Transformer»

Спеціальність 122 – «Комп'ютерні науки»

Гарант освітньої програми

_____Д.е.н., професор_____ _____ _____Руденський Р.А.

(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи

_____Д.е.н., професор_____ _____ _____Руденський Р.А.

(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ _____Костенко А.В._____

(підпис) (ПІБ)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ

І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних наук

_____/ Голуб Б.Л., доцент, к.т.н /
підпис

"____"_____2025 р.

З А В Д А Н Н Я
на виконання бакалаврської кваліфікаційної роботи студенту
Костенку Андрій Вікторовичу
(прізвище, ім'я, по батькові)

Спеціальність 122 – «Комп'ютерні науки»

1. Тема бакалаврської кваліфікаційної роботи
«Розробка моделі для доповнення коду на основі LSTM і Transformer»
Затверджена наказом ректора НУБіП України №2246,С від "16" 12 2024 р.
2. Термін подання завершеної роботи на кафедру 2025 . 06 . ____
рік, місяць, число

3. Вихідні дані до бакалаврської кваліфікаційної роботи
Опис програмного забезпечення,
4. Перелік питань що розглядається:
   1. Системний аналіз предметної області інформаційної системи
   2. Проєктування інформаційного та програмного забезпечення
   3. Розробка інформаційного та програмного забезпечення
   4. Рекомендації щодо впровадження та експлуатації системи
   5. Висновки

Керівник бакалаврської кваліфікаційної роботи_____/ Р.А. Руденський/
підпис ініціали та прізвище

Завдання прийняв до виконання_____/ А.В. Костенко/
підпис ініціали та прізвище

Дата отримання завдання 2025 . 01 . 08
Рік місяць, число

АНОТАЦІЯ

У дипломній роботі розглядається розробка моделі доповнення коду на основі архітектури Transformer. Інтеграція великих мовних моделей (LLM) в IDE відкриває нові можливості підвищення продуктивності розробників. Архітектура LSTM, хоча й застосовувалась раніше, має обмеження в задачах доповнення коду через труднощі з паралелізацією та збереженням довготривалого контексту. Натомість Transformer, завдяки механізму самозвертання краще підходить для таких завдань. У роботі розглянуто принципи дії цієї архітектури та її навчання. Також створено плагін для VS Code, що забезпечує зручне використання моделі.

Робота складається з чотирьох розділів: аналіз предметної області та постановка задач; теоретичні засади Transformer і генерації коду; розробка моделі та плагіна; оцінка результатів і пропозиції для подальших досліджень.

ABSTRACT

This thesis investigates the development of a code completion model based on the Transformer architecture. Integrating large language models (LLMs) into the IDE opens up new possibilities for code generation and increases developer productivity. Although LSTM architectures have previously been used for sequence modeling, they exhibit limitations in code completion tasks due to issues with parallelization and maintaining long-term dependencies. In contrast, Transformers, becouse of their introspection mechanism are better suited for such tasks. The thesis examines the principles of the Transformer architecture and its learning process. A VS Code plugin was also developed for convenient use of the model.

The work consists of four chapters: an analysis of the code completion domain and research objectives; theoretical foundations of the Transformer and code generation; development of the model and plugin; and evaluation of results with suggestions for future research.

ЗМІСТ

ВСТУП

Актуальність. Зараз у сфері програмування все більше уваги приділяється інструментам, які можуть автоматично виконувати рутинні завдання. Одним із таких напрямів є системи автодоповнення коду, які допомагають розробникам писати код швидше та з меншими зусиллями. Завдяки появі великих мовних моделей (LLM) з'явилися нові можливості для створення розумніших і точніших інструментів автодоповнення.

Багато сучасних середовищ розробки (IDE) вже мають функції штучного інтелекту, які допомагають програмістам. Але побудувати модель, яка справді розуміє, що відбувається в коді, і може підказувати щось дійсно корисне – досить складна задача. Для цього потрібні спеціальні архітектури, які добре працюють з послідовностями.

Однією з найуспішніших архітектур для таких завдань є Transformer. Вона лежить в основі багатьох або можна навіть сказати більшості сучасних мовних моделей. Раніше аналізу послідовностей використовували LSTM, але в неї є певні обмеження – особливо коли мова йде про довгі послідовності або потребу в швидких обчисленнях.

Метою даної дипломної роботи полягає у розробці моделі для інтелектуального доповнення коду на основі архітектури Transformer. Для практичного застосування розробленої моделі передбачається створення плагіна для популярного редактора коду VS Code, що забезпечить зручний інтерфейс для користувачів.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

1. Провести огляд існуючих інструментів, які використовують великі мовні моделі (LLM) для генерації та доповнення коду. Зокрема, розглянути такі рішення, як GitHub Copilot та Codeium (раніше відомий як Windsurf).

2. Оглянути архітектуру Transformer як основну технологію, використану в більшості мовних моделей для генерації коду, порівняти її з альтернативними підходами (RNN/LSTM) та пояснити переваги Transformer.
3. Розробити власну модель доповнення коду, використовуючи сучасні архітектури (на основі модифікованого Transformer), та навчити її на релевантному наборі даних, включаючи відкриті репозиторії з GitHub, приклади коду та спеціалізовані інструкції.
4. Реалізувати інтерфейс взаємодії з користувачем у вигляді інтерфейсу, що дозволяє задавати запити до моделі у зручній текстовій формі.

Проєктування даної роботи охоплює реалізацію наступних ключових компонентів:

1. **Модель автодоповнення коду** – створена на базі архітектури Transformer і навчена на великій кількості прикладів коду.
2. **Інтеграція з IDE** – зроблений плагін для VS Code, який дозволяє використовувати цю модель прямо під час написання коду.
3. **Оцінка результатів** – були проведені тести, щоб перевірити, наскільки добре і швидко працює як сама модель, так і плагін.

Технології, що використовуються:

1. Архітектура моделі: Transformer
2. Мови програмування: Python (для розробки моделі та серверу), JavaScript (для плагіна VS Code)
3. Бібліотека машинного навчання: PyTorch
4. Середовище розробки: VS Code та VS Code API для створення плагіна

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка завдання

Сьогодні розробка програмного забезпечення стає все складнішою: проєкти ростуть, коду стає більше, а вимоги до швидкості та якості роботи розробників постійно зростають. У такій ситуації особливо корисними стають інструменти, які можуть автоматизувати рутинні задачі й допомагати розробникам у контексті того, над чим вони працюють. Один із напрямів, який активно розвивається останніми роками – це системи автодоповнення коду. Якщо раніше вони обмежувались простими підказками, то зараз, завдяки використанню ШІ, здатні передбачати наступні кроки програміста значно точніше.

З появою великих мовних моделей, побудованих на архітектурі Transformer, відкрились нові можливості у сфері генерації коду. Завдяки механізму самозвертання ці моделі добре розуміють довгі шматки коду і можуть знаходити складні залежності між елементами. Це дуже важливо для правильного доповнення та аналізу коду. Якщо інтегрувати ці моделі безпосередньо в середовище розробки, це може суттєво підвищити продуктивність програміста і зменшити час, необхідний на реалізацію рішень.

Коли розробник має інструмент, який не тільки підказує, як завершити код, а ще й може відповісти на запитання про синтаксис, пояснити фрагменти коду, допомогти з API чи пошуком помилок – це реально піднімає розробку на новий рівень. А ще краще, коли всі ці запити можна ставити звичайною мовою прямо в IDE, не витрачаючи час на читання документації чи пошуки в інтернеті.

Система має підтримувати такі можливості:

- контекстне доповнення коду;

- генерація коду за текстовим описом;
- пояснення частин коду на вимогу;
- підтримка популярних мов програмування (наприклад Python/JavaScript).

Практична цінність

Практична цінність цієї дипломної роботи полягає в наступному:

- Створена система може стати основою для відкритого інструменту автодоповнення коду, який не буде залежати від сторонніх сервісів.
- Її можна підлаштувати під конкретні задачі або напрямки – наприклад, для фронтенду, data science чи роботи з Python.
- Завдяки чат-інтерфейсу з моделлю взаємодія стає зручною: новачкам легше писати код, а досвідчені розробники можуть працювати швидше.

1.2 Огляд інформаційних джерел та існуючих рішень

Сфера генерації коду за допомогою LLM динамічно розвивається, і на ринку вже існує чимало рішень. Найпоширенішими серед них є GitHub Copilot, Codeium, а також Claude Code, який хоча й дещо відрізняється, але виконує схожі завдання [12].

GitHub Copilot – це розширення, створене GitHub у співпраці з OpenAI. Воно працює як інтелектуальний помічник програміста, підказуючи фрагменти коду або навіть цілі функції в процесі написання. Copilot підтримує багато мов програмування та інтегрується безпосередньо в редактори коду, такі як VS Code. З практики видно, що Copilot особливо добре справляється з шаблонними завданнями: наприклад, коли потрібно написати обробник події чи стандартний запит до API. Але бувають ситуації, коли його підказки не завжди доречні, особливо якщо проєкт складніший або має нестандартну архітектуру [12].

Codeium(Windsurf) – це безкоштовна альтернатива Copilot, яка також надає функції автодоповнення. Перевагою Codeium є його доступність: для використання не потрібно платити, а функціональність при цьому залишається досить високою. У багатьох випадках Codeium пропонує не гірші (а іноді навіть кращі) підказки, ніж Copilot. Особисто при тестуванні Codeium я помітив, що він іноді точніше вгадує мої наміри, особливо якщо я пишу коментар англійською перед кодом [12]. Приклад того як вигладає інтерфейс codeium на рис. 1.1

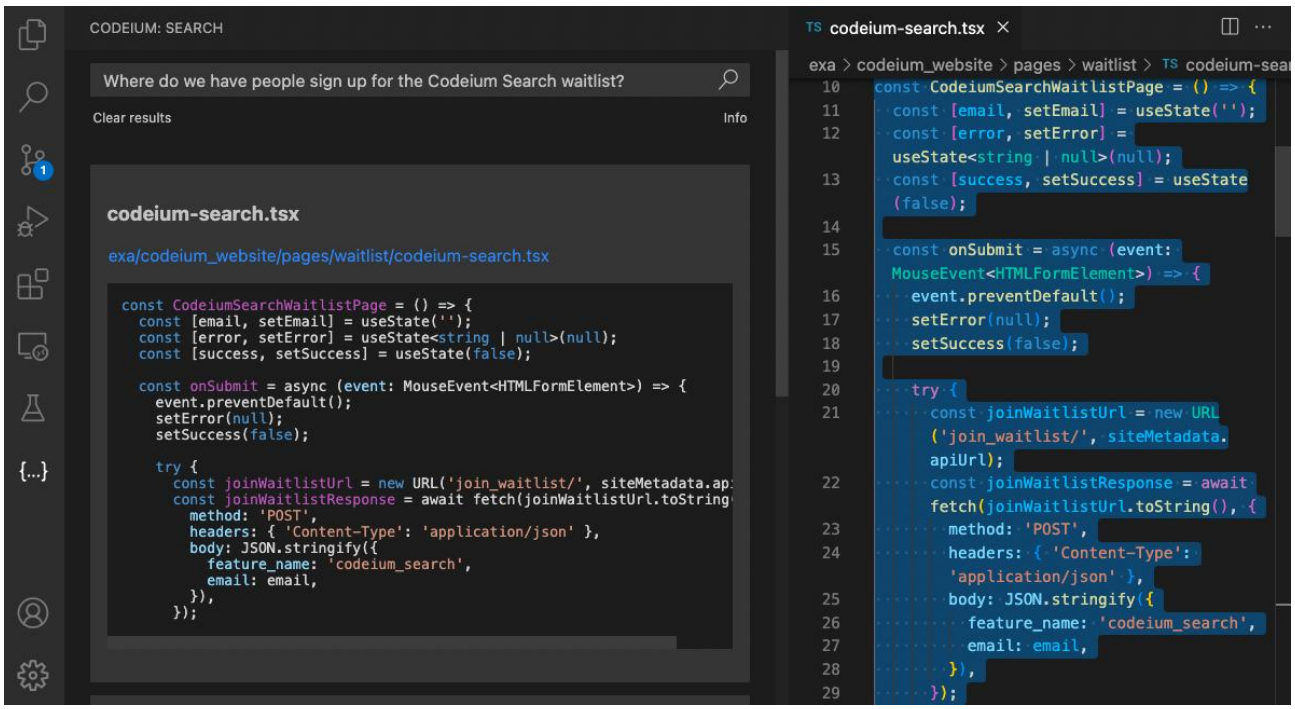

Рис. 1.1 Інтерфейс Codeium(WindSurf) (за Atlassian, 2023)[1]

Claude Code – це модель від Anthropic, яка може працювати з кодом, давати пояснення, писати фрагменти функцій і навіть допомагати у відлагодженні. Особливість Claude в тому, що він краще підходить для роботи з більшими обсягами тексту: наприклад, він може проаналізувати одразу цілий файл або дати коментар до великого блоку коду. У порівнянні з Copilot і Codeium, Claude більше схожий на співрозмовника чи асистента, з яким можна обговорити ідеї або проблеми в коді, а не просто отримувати підказки. Для цього встановлюється CLI-інструмент, після чого стає доступною команда, claude code. Користувач може викликати модель безпосередньо з командного рядка: наприклад, написати “claude code explain index.js” і Claude проаналізує вміст файлу index.js та надасть пояснення до коду. Також можна ставити запитання або просити згенерувати функцію, наприклад claude code “напиши функцію для сортування масиву” [12].

Ще одна сильна сторона Claude – він добре запам’ятовує контекст і логіку проєкту під час роботи в одній сесії. Завдяки цьому з ним можна вести справжній діалог: уточнювати, редагувати й допрацьовувати ідеї. Це робить його корисним не тільки для автодоповнення коду, а й для повного написання та проєктування програм [12].

1.3 Моделювання предметної області

У сучасній розробці програм все більше використовують інструменти, які допомагають автоматизувати рутинні задачі і загалом пришвидшити роботу. Один з таких напрямів – це системи автодоповнення коду, що працюють на базі великих мовних моделей (LLM). Вони можуть розуміти, що саме пише розробник, і підказувати відповідні шматки коду, пояснення або навіть цілі функції. Також такі системи можуть адаптуватися під стиль коду, до якого звик сам програміст [13, с.1].

Процес створення програмного забезпечення зазвичай включає кілька етапів: постановку задачі, проєктування, написання коду, тестування, налагодження і подальшу підтримку. Системи автодоповнення найчастіше використовуються саме під час написання коду та його налагодження. На цьому етапі розробник продумує логіку програми, її структуру, і часто має швидко реалізовувати типові частини – наприклад, запити до бази даних, обробку подій чи створення інтерфейсу. Тут якраз і стають у пригоді такі інструменти.

У таких ситуаціях LLM-системи відіграють роль асистентів, які допомагають швидше реалізувати потрібне, а також можуть підказати, як правильно реалізувати певний алгоритм, виправити або просто найти помилку помилку в коді. Інтеграція таких систем безпосередньо в IDE дозволяє зробити цей процес максимально безшовним [13, с.1].

Основні учасники процесу:

1. **Розробник** – основний користувач системи. Він формулює запити до моделі у вигляді текстових інструкцій природною мовою, наприклад: «Напиши функцію для сортування списку». Після цього він отримує результат – згенерований код, який може використовувати, змінювати або ігнорувати [13, с.3].

2. **Інтерфейс** – середовище, в якому відбувається взаємодія з моделлю. Його завдання – передавати текстові запити до моделі та показувати відповіді користувачеві. [13, с.3]
3. **Модель генерації коду** – ядро системи. Вона приймає текстові інструкції, аналізує їх зміст і генерує відповідний програмний код [13, с.3].

Система працює за простим і зрозумілим сценарієм:

1. Розробник формулює завдання у вигляді тексту.
2. Інтерфейс передає цей запит до моделі.
3. Модель обробляє запит, аналізує ключові слова та контекст.
4. У відповідь модель повертає згенерований текст.
5. Розробник переглядає результат і вирішує, чи підходить він для задачі, або вносить правки.

Типові сценарії використання:

1. **Генерація функцій**
   Користувач просить створити функцію. Модель генерує відповідний код.
2. **Створення класів та структур**
   Запит: «Створи клас для представлення товару з полями назва, ціна і кількість». Система генерує шаблон класу.
3. **Пояснення коду**
   Користувач вставляє фрагмент коду й запитує: «Що робить ця функція?». Модель може надати пояснення.
4. **Рефакторинг або зміна стилю**
   Користувач просить переписати код у певному стилі або зробити його більш ефективним.
5. **Навчання програмуванню**
   Система може допомогти новачкам, пояснюючи, як працюють певні конструкції або алгоритми, й одразу показуючи приклад.

Щоб краще зрозуміти, як саме функціонують такі системи на технічному рівні, варто розібратись у базових принципах, на яких вони побудовані. У центрі сучасних LLM лежить архітектура, заснована на штучних нейронних мережах. Ці мережі імітують роботу людського мозку, дозволяючи машині навчатися на прикладах і приймати рішення на основі отриманих знань.

Штучна нейронна мережа – це система, що складається з багатьох з'єднаних між собою “нейронів”. Кожен окремий “нейрон” приймає на вхід числові значення, виконує над ними необхідні обчислення і потім транслює отриманий результат далі [15].

Основна мета нейронної мережі – навчитися апроксимувати деяку функцію $F(x)=y$, тобто знайти таку відповідність між входом x і бажаним виходом y, яка дозволить робити правильні передбачення для нових, раніше невідомих даних [15].

На вхід нейронної мережі подається вектор чисел. Це може бути, наприклад, вектор токенів.

Між нейронами різних шарів існують зв'язки. Кожен зв'язок має числовий коефіцієнт – його називають вагою. Вага визначає, наскільки сильно певний вхід впливає на результат роботи нейрона. Якщо вага велика – значення з цього входу “важливіше”. Якщо вага нульова або близька до нуля – нейрон майже ігнорує цей вхід. Нейрон множить кожне вхідне значення на свою вагу і далі обчислює суму всіх цих добутків [15]. Схематичне зображення цього процесу наведено на рис. 1.2.

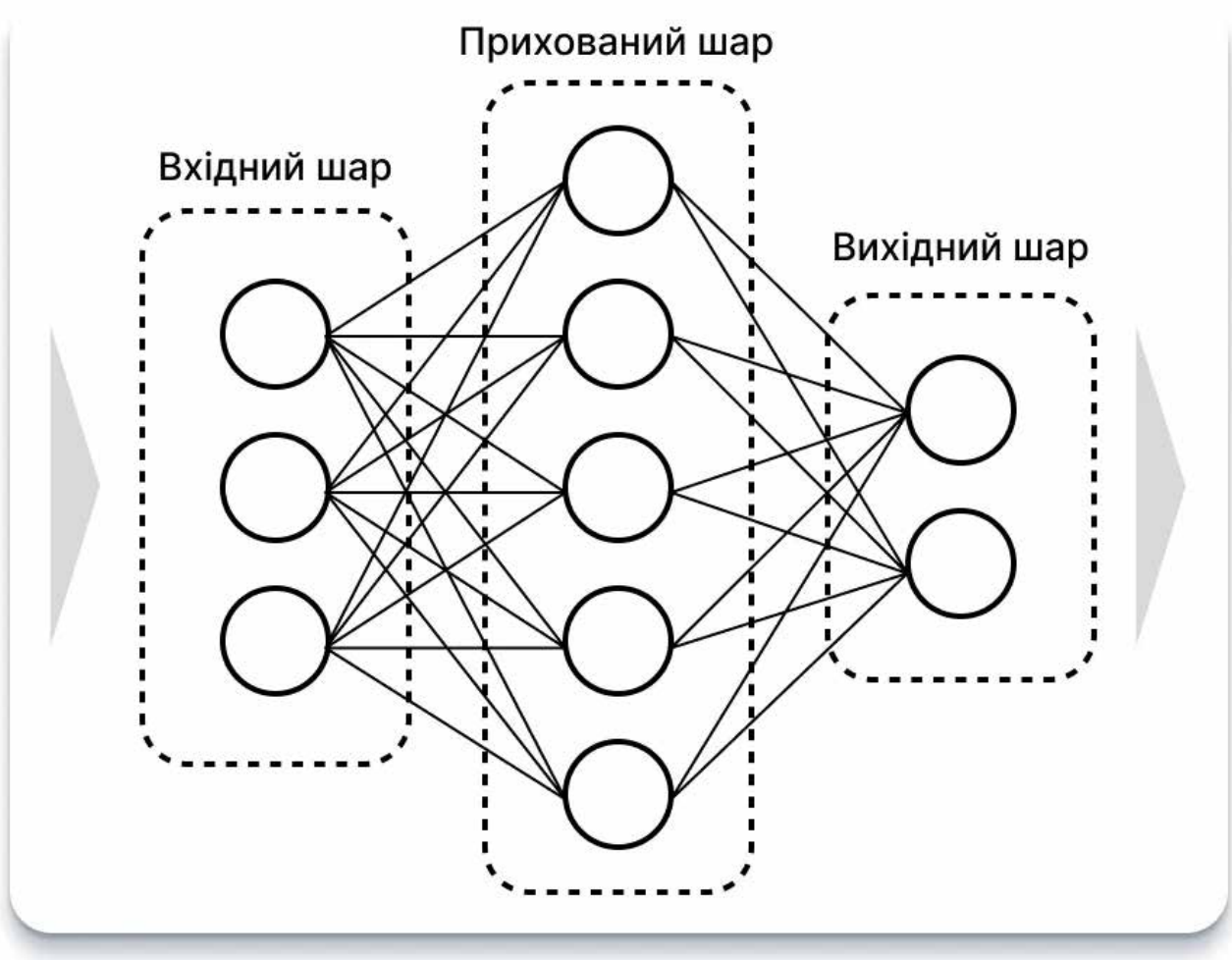

Рис. 1.2 Схема мультишарового персептрона (за The Transmitted, 2023)[2].

Крім ваг, кожен нейрон також має додаткове число – зсув (bias). Це значення додається до зваженої суми входів безпосередньо перед застосуванням функції активації. Наявність зсуву дає нейрону можливість генерувати вихідне значення навіть у випадку, коли всі його вхідні сигнали дорівнюють нулю. Це суттєво підвищує гнучкість моделі, дозволяючи їй краще адаптуватися до різноманітних сценаріїв під час навчання [15].

Після обчислення зваженої суми та додавання зсуву, результат передається через функцію активації. Це нелінійна функція, яка вирішує, що саме “пропустити далі”. Якщо використовувати її не будемо – мережа з кількома шарами буде поводитись як один великий лінійний перетворювач, тобто буде дуже обмеженою. Саме завдяки цій функції мережа здатна розпізнавати складні закономірності, як мову чи зображення [15].

Найпоширеніші функції активації – ReLU, яка обнуляє все, що менше нуля, tanh, яка нормалізує значення в межах від -1 до 1 і sigmoid, яка звужує значення до інтервалу від 0 до 1 [15].

Коли ми подаємо приклад на вхід і мережа генерує передбачення, ми можемо порівняти їх з правильною відповіддю. Різниця між передбаченим результатом і правильним називається помилкою або втратою (loss). Це може бути, наприклад, відстань між числовими значеннями або ймовірностями у випадку класифікації. Чим менша ця помилка, тим краща модель [15].

Для того щоб зменшити помилку, мережа повинна змінювати свої ваги і bias'и і для цього потрібно знати в який бік і наскільки сильно їх треба змінювати. Для цього використовується градієнт – це значення, які показують, як треба оновити параметри щоб зменшити помилку [15].

Backpropagation, або зворотне поширення помилки – це процес, під час якого ми обчислюємо градієет і як саме кожна вага і bias вплинули на загальну помилку, і як їх потрібно змінити, щоб помилка зменшилась [15].

Ось як це відбувається крок за кроком:

1. **Прямий прохід:** ми подаємо вхідні дані, всі нейрони обчислюють свої значення шар за шаром, поки не отримаємо передбачення.
2. **Обчислення помилки:** порівнюємо передбачення з реальною відповіддю і оцінюємо втрати (loss).
3. **Зворотне поширення:**
    1. Починаючи з останнього шару, ми визначаємо, які параметри найбільше вплинули на помилку.
    2. Потім крок за кроком рухаємось до попередніх шарів, розраховуючи вплив кожного параметра.
    3. Цей процес базується на використанні правила похідної та ланцюгового правила для “передачі” впливу помилки у зворотному напрямку через усю архітектуру мережі.
4. **Оновлення параметрів:** Після того, як ми розрахували градієнти, ми поступово коригуємо ваги та зміщення (bias'и). Мета цього

коригування – зменшити помилку під час наступного проходу даних. Величина цього “кроку” коригування залежить від заздалегідь визначеного параметра – швидкості навчання.

Весь цей цикл повторюється величезну кількість разів. Щоразу мережа навчається на нових прикладах, трохи оптимізує свої внутрішні параметри, і з кожним таким кроком її продуктивність покращується.

До появи архітектури Transformer для роботи з послідовними даними – такими як текст, аудіо чи часові ряди – домінували рекурентні нейронні мережі (RNN). Однією з найефективніших модифікацій RNN стали мережі Long Short-Term Memory (LSTM), які зробили прорив у здатності моделей обробляти довгі залежності в даних. Вперше запропоновані у 1997 році, LSTM частково розв’язали проблему затухання градієнтів, яка заважала класичним RNN зберігати довготривалу інформацію [9].

На відміну від звичайних RNN, LSTM мають складнішу внутрішню структуру, яка включає спеціальні елементи управління – так звані гейти [9]:

1. вхідний (input gate) – вирішує, яку нову інформацію зберігати [9];
2. забувальний (forget gate) – визначає, яку інформацію з попереднього стану потрібно "забути" [9];
3. вихідний (output gate) – контролює, яка інформація з пам’яті передається далі [9].

Ці гейти дозволяють LSTM вибірково зберігати, оновлювати та витягати інформацію на кожному кроці, завдяки чому мережа формує ефективну внутрішню пам’ять, яка може зберігати інформацію протягом сотень і навіть тисяч часових кроків. Схематичне зображення структури LSTM наведено на рис. 1.3.

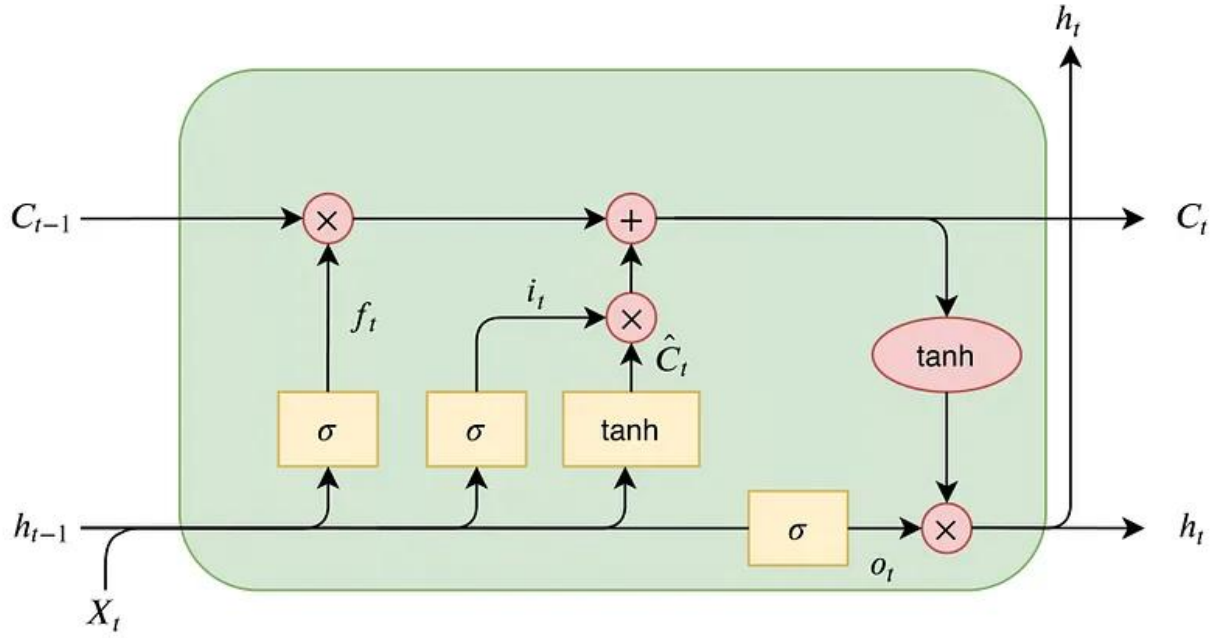

Рис. 1.3 Схема LSTM (за Saba, 2020) [3].

Але попри те, що LSTM зберігає контекст, у неї є кілька критичних обмежень, які зробили її застарілою в порівнянні з Transformer:

- **Не знає майбутнього.** LSTM читає послідовність поетапно, зліва направо (або справа наліво), і тому не може знати, яка частина інформації стане важливою у майбутньому. Наприклад, вона може не “запам’ятати” важливу змінну, бо не знає, що вона використається далі [16].
- **Обмежена пам’ять**. На практиці LSTM рідко запам’ятовує ефективно більше ніж 20-50 токенів, оскільки з кожним кроком пам’ять поступово “розмивається”. Це абсолютно недостатньо для обробки складного коду, де залежності можуть сягати сотень або тисяч токенів [16].
- **Погана масштабованість.** LSTM не паралелізується по часу, бо кожен крок залежить від попереднього. Це означає, що обробка послідовності довжиною 1000 токенів займає в 1000 разів більше часу, ніж одного токена – і це серйозне обмеження на GPU, де паралелізм критичний [16].

Були спроби вдосконалити LSTM: XLSTM, mLSTM, sLSTM та інші варіанти які додавали більші блоки пам’яті або можливість певного паралелізму. Але жодна з них не досягла ефективності Transformer.

2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Логічна модель даних

Основна ідея цієї системи – генерувати код на основі того, що користувач вводить у текстовому вигляді. Щоб це працювало, модель повинна вміти працювати з різними типами даних. Уся інформація проходить кілька етапів обробки: спочатку йде текстовий запит від користувача, а в результаті – готовий шматок коду, який створює модель.

Типи даних, які використовуються в системі:

1. **Текст (запит користувача)**
   На початку взаємодії з системою користувач вводить текстовий запит. Це звичайне речення або коротка інструкція, написана природною мовою (наприклад: “Створи функцію, яка перевіряє, чи число парне”). Цей текст є вхідними даними для моделі.
2. **Токени**
   Перед тим як модель почне обробляти текст, вона розбиває його на дрібні частинки – токени. Це можуть бути окремі слова, частини слів або навіть окремі символи. Токенізація – це процес, який переводить звичайний текст у набір чисел, з якими вже може працювати модель.
3. **Вектори представлень (embeddings)**
   Після токенізації кожен токен перетворюється на вектор – набір чисел, який містить інформацію про його значення та контекст. Завдяки цьому модель розуміє, що саме має на увазі користувач, і може краще сформувати відповідь або згенерувати відповідний код.
4. **Параметри моделі**
   Всередині модель має мільйони або навіть мільярди параметрів.

Саме вони зберігають знання, які модель отримала під час навчання. Ці параметри впливають на те, як модель формує відповіді, підбирає слова чи будує структуру коду.

5. **Згенерований код**

   На виході модель повертає згенерований текст – це програмний код, що відповідає запиту користувача. Він теж спочатку зберігається як послідовність токенів, а потім перетворюється назад у зрозумілий для людини вигляд.

Перш ніж модель зможе виконувати реальні запити користувачів, її потрібно навчити. Навчання відбувається у два основних етапи.

На першому етапі модель навчається передбачати наступний токен у послідовності символів. Це може бути як частина тексту, так і частина програмного коду. Проте у моїй моделі основний акцент в навчальній вибірці зроблено саме на коді, оскільки вона призначена для генерації програм.

Під час навчання модель отримує вхідні дані без жодних пояснень або інструкцій – лише «сирі» приклади. Це можуть бути великі фрагменти коду з відкритих репозиторіїв, документації, приклади з підручників або просто розмови людей в Reddit.

На цьому етапі модель вчиться розуміти, як взагалі побудована мова, подібно до того, як люди вивчають звичайну мову. Вона читає величезну кількість тексту і поступово починає помічати загальні закономірності: які слова (токени) йдуть одне за одним, як виглядають типові конструкції, де ставляться дужки, кому, назви змінних тощо.

Це основний етап, без якого модель просто не зможе виконувати складніші задачі. Тому важливо, щоб у навчальній вибірці було достатньо прикладів і коду і звичайного тексту, хоча надто багато даних теж може заплутати модель тому тут важливо балансувати між кількістю і якістю.

Такий процес називається попереднє навчання (pretraining). Після нього модель вже може більш-менш розуміти, як виглядає код і як його логічно

продовжити. Але поки що вона ще не вміє виконувати конкретні завдання чи відповідати на запитання – це навчається пізніше.

На другому етапі модель навчають слідувати інструкціям, так як після переднавчання модель може тільки продовжувати текст який їй видали, наприклад на привіт, вона не привітається у відповідь, а продовжить речення, наприклад «Привіт, я мовна модель...». Для цього використовують спеціалізовані набори даних, що складаються з пар «запит – відповідь», де запит формулюється як інструкція користувача, а відповідь – це очікувана реакція моделі.

На цьому етапі можуть застосовуватись різні підходи. Один з них – supervised fine-tuning (SFT), коли модель навчається безпосередньо на прикладах правильних відповідей. Інший, більш сучасний підхід – навчання з підкріпленням за допомогою зворотного зв’язку від людини (Reinforcement Learning from Human Feedback, RLHF). У цьому випадку модель спочатку генерує кілька варіантів відповіді, а потім за участю людських оцінювачів чи допоміжної моделі оцінюється якість цих відповідей, щоб удосконалити поведінку основної моделі [17, с.2].

На цьому етапі модель не просто передбачає наступний токен, а навчається розуміти завдання і створювати код відповідно до інструкції. Вона вчиться зв’язувати текстовий запит користувача з відповідним фрагментом коду, функцією чи алгоритмом, цей етап займає менше часу і обчислень так як модель вже розуміє як побудована мова після першого етаму і все що чому їй потрібно навчитися це слідувати інструкціям замість простого продовження коду [17, с.2].

Також у навчальних даних використовуються **спеціальні токени** – це символи або послідовності, які на перший погляд нічого не означають для людини, але мають велике значення для моделі. Вони допомагають розділяти частини даних, позначати початок чи кінець інструкції, відокремлювати різні типи інформації. Наприклад, спеціальний токен **<|instruction|>** може позначати початок інструкції, а **<|code|>** – початок коду. Це допомагає моделі краще

розуміти структуру вхідних даних і відділити запит користувача від тексту який модель сама згенерувала.

Завдяки такому навчанню – спочатку на основі передбачення токенів, а потім на основі виконання інструкцій – модель може починає генерувати щось корисне за новими запитами.

2.2 Вибір системи управління інформаційною базою

Для зберігання і обробки даних, які використовуються під час навчання моделі, я обрав файлову систему.

Чому я обрав файлову систему:

1. **Простота використання.** Файлова система є стандартним інструментом у будь-якій операційній системі, і з нею легко працювати без додаткових налаштувань.
2. **Гнучкість формату збереження.** Я зберігаю дані у вигляді об’єктів NumPy, які дуже зручні для роботи з числовими масивами і векторами. NumPy файли (.npy, .npz) швидко читаються і записуються, а також оптимізовані для зберігання великих обсягів числових даних.
3. **Відсутність складнощів з інсталяцією і підтримкою.** Спеціалізовані векторні бази даних для часто потребують налаштувань і ресурсів, що не завжди виправдано для моєї задачі, де мені потрібно лише один раз зберегти навчальну вибірку.

Вимоги до системи управління інформаційною базою

1. **Обсяг даних.** Для навчання моделі потрібно зберігати великі обсяги інформації. Система має підтримувати можливість зберігати багато окремих файлів, щоб можна було організувати дані у вигляді невеликих частин.

2. **Швидкість доступу.** Використання файлової системи і формату NumPy дуже зручне, тому що я можу відкривати тільки ті файли, які потрібні в конкретний момент, завантажувати їх у оперативну пам'ять і швидко отримувати доступ до потрібних даних.
3. **Покрокова обробка**. Під час навчання не потрібно випадково чи хаотично звертатися до різних частин даних – модель обробляє дані послідовно. Тому зручніше мати багато невеликих файлів, які можна відкривати по черзі, обробляти їх, потім закривати і переходити до наступного.
4. **Масштабованість**. Завдяки файловій системі можна легко додавати нові файли з даними, збільшуючи обсяг інформації без складних налаштувань.

2.3 Створення інформаційної бази

Перш ніж починати навчати модель, потрібно підготувати дані. У моєму випадку я використовую вже готові відкриті датасети, які містять велику кількість тексту та коду. Основні два джерела, з якими я працюю:

- **C4 (Colossal Clean Crawled Corpus) –** це великий набір текстів, зібраних з відкритих ресурсів. Його почистили від реклами, дублікатів та іншого непотрібного тексту. Є різні версію цього датасету, наприклад тільки з англійським текстом або не очищений.
- **Код із GitHub –** публічні репозиторії, які містять реальний код з проєктів. Такий код особливо корисний, якщо модель має вміти розуміти програмування.

Перед початком навчання, навіть якщо датасети вже зібрані, необхідно провести їх підготовку. Один із ключових етапів – токенізація.

Токенізація – це процес розбиття тексту на дрібні частини і перетворення них на числа. Це можуть бути окремі слова, символи або частини слів. Після цього токени перетворюються у вектори, з якими вже працює модель.

Токенізація – це не дуже швидкий процес, особливо коли даних багато. Тому, щоб не повторювати цю операцію кожного разу, я зберігаю готові токени у вигляді масивів NumPy на диск. Це значно економить час під час запуску навчання. Приклад розбиття тексту на токени наведено на рис. 2.1.

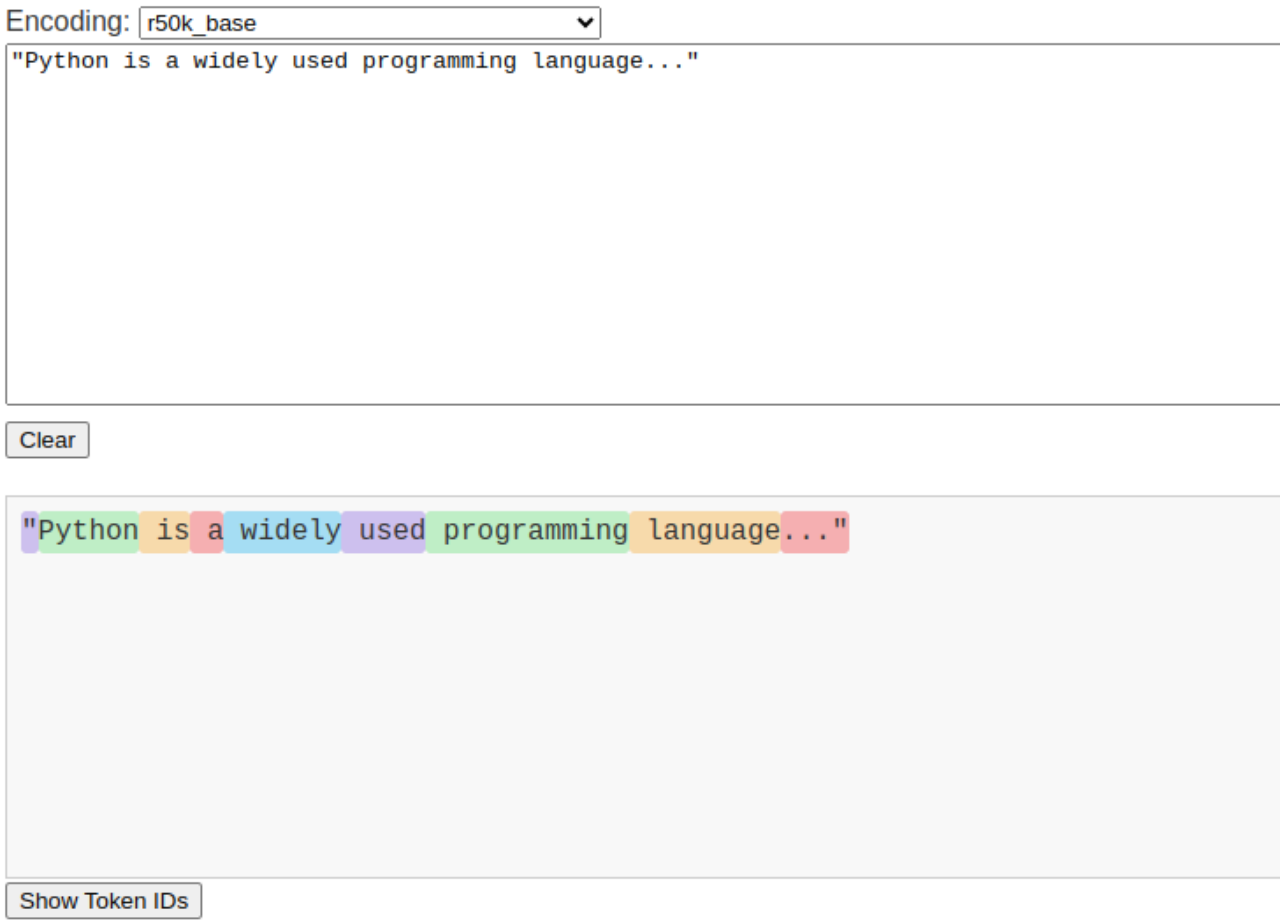

Рис. 2.1 Приклад роботи токенізатора (за GPT Tokenizer, 2023) [4].

Спочатку для токенізації я використовував tiktoken від OpenAI – ефективний та оптимізований токенізатор, що підтримує різні словники (vocabularies) і добре підходить для популярних моделей OpenAI. Проте з переходом на власну мовну модель, сумісну з архітектурою DeepSeek (1.5B параметрів), я замінив токенізатор на той, що використовує DeepSeek.

Новий токенізатор підтримує більше 150 000 токенів. Це дозволяє досягти кращого ступеня стиснення тексту, зменшуючи середню кількість токенів для одного рядка коду чи тексту.

Перехід на токенізатор DeepSeek був необхідним кроком для повної сумісності з моделлю, а також дозволив краще використовувати її потенціал при навчанні.

Після завершення токенізації отримані послідовності токенів я зберігаю на диск у вигляді NumPy-масивів (.npy). Це дозволяє не виконувати повторну токенізацію при кожному запуску навчання, а одразу завантажувати готові числові представлення тексту в оперативну пам’ять, що значно пришвидшує процес тренування моделі.

3 ПРИКЛАДЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Організаційна структура програмного забезпечення

У цьому розділі розглядається архітектура розробленого програмного забезпечення для доповнення коду. Рішення складається з кількох незалежних, але взаємопов’язаних компонентів: моделі на основі архітектури Transformer, серверної частини для генерації коду та клієнтського плагіна для VS Code. Кожна з частин виконує свою чітку функцію, і разом вони забезпечують роботу системи (див. рис. 3.1).

Система складається з трьох основних блоків:

1. **Модуль навчання моделі** – відповідає за підготовку та тренування моделі на великій кількості токенізованих даних. Коли навчання закінчується, модель зберігається у файл, який потім можна використати на сервері.
2. **Сервер автодоповнення коду –** це простий HTTP-сервер, який завантажує вже навчену модель у пам’ять і чекає на запити від користувача. Він отримує інструкцію, генерує відповідь і відправляє її назад клієнту.
3. **Плагін VS Code** – клієнтська частина, яка дозволяє користувачеві надсилати запит безпосередньо з редактора коду. Плагін звертається до сервера, отримує результат і відображає його в інтерфейсі редактора.

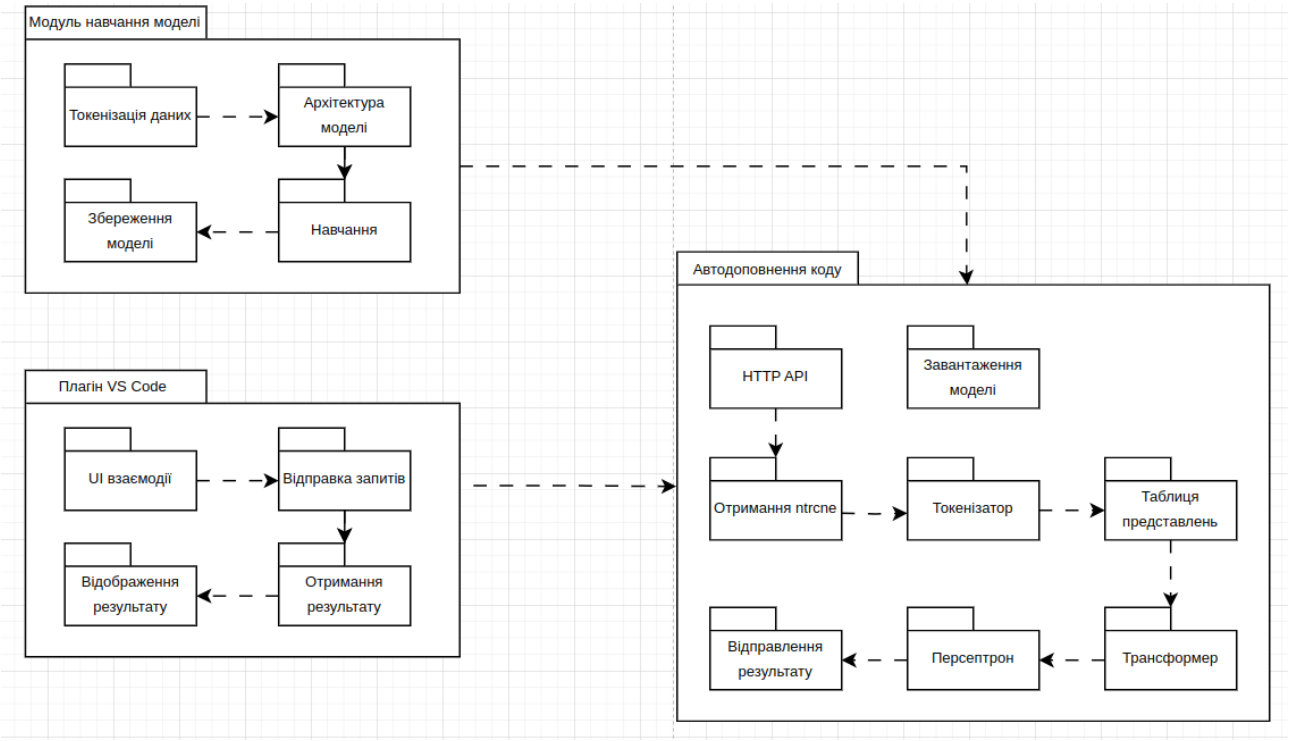

Рис. 3.1 Діаграма пакетів

Модуль навчання моделі реалізовано як окремий скрипт на Python, який використовує PyTorch для навчання моделі. **Він виконує наступні функції:**

1. завантаження токенізованих навчальних даних (у форматі .npy);
2. ініціалізація та конфігурація моделі Transformer;
3. навчання моделі з використанням алгоритму оптимізації;
4. збереження моделі у файл після завершення навчання, а також створення контрольних точок під час навчання (model_0000.pt).

Цей модуль є ізольованим – він не взаємодіє напряму з плагіном чи сервером. Модель після навчання переноситься вручну на сервер.

Серверна частина – це мінімалістичний веб-сервер, реалізований на Python. Він завантажує збережену модель у пам’ять і чекає на HTTP-запити.

Основні функції:

1. завантаження моделі (torch.load(...));
2. прийом POST-запиту з інструкцією від клієнта;
3. генерація відповіді за допомогою моделі;
4. повернення згенерованого тексту у вигляді JSON.

Оскільки модель велика, її завантаження відбувається один раз під час запуску сервера, після чого вона залишається в пам’яті для швидкої відповіді.

Плагін VS Code який реалізовано на JavaScript за допомогою API Visual Studio Code. Його завдання – забезпечити просту інтеграцію між середовищем розробки та сервером.

Основні функції плагіна:

1. відправлення запиту на сервер з текстом інструкції;
2. прийом відповіді;
3. відображення результату згенерованого коду;

Плагін не має прямого доступу до моделі – усі запити обробляються через сервер.

Взаємодія між компонентами побудована максимально просто. Модуль навчання зберігає модель у файл – це єдина точка дотику з іншими частинами системи. Далі:

1. Сервер завантажує цей файл моделі при старті.
2. Плагін надсилає запит із VS Code до сервера (через HTTP).
3. Сервер передає запит моделі та повертає відповідь.
4. Плагін вставляє результат у редактор коду.

Таким чином, між компонентами відсутні жорсткі залежності. Це робить систему гнучкою та легкою у підтримці – можна змінювати чи оновлювати окремі модулі без необхідності переписувати все з нуля.

Обрана архітектура дозволяє масштабувати рішення в майбутньому. Наприклад:

1. Можна замінити простий сервер на повноцінний кластер зі шардінгом моделей, якщо в майбутньому модель виявиться надто великою для однієї відеокарти.
2. Можна інтегрувати кешування запитів для прискорення відповідей.
3. Плагін можна оновити для підтримки інших редакторів, окрім VS Code.

3.2 Вибір інструментарію для створення ППЗ

Під час розробки системи доповнення коду було важливо обрати інструменти, які є надійними, зручними у використанні, добре документованими та не створюють зайвих технічних проблем. Так як проект складається з кількох частин – моделі машинного навчання, сервера для генерації, а також плагіна для VS Code – для кожного компонента були обрані відповідні засоби реалізації, виходячи з їх практичності та сумісності між собою.

Реалізація та навчання моделі виконана на фреймворці PyTorch. Це одна з найпопулярніших бібліотек для машинного навчання, особливо у сфері досліджень та експериментів з нейронними мережами. Основні причини такого вибору:

- **Простота та зрозумілий синтаксис.** У порівнянні з TensorFlow, код на PyTorch виглядає більш “природним”. Більшість речей реалізується інтуїтивно, без зайвих обгорток.
- **Гнучкість у налаштуванні.** PyTorch дозволяє легко будувати власні архітектури, змінювати внутрішню логіку моделей, контролювати навчання, писати свої класси для обчислення помилки, оптимізатори тощо. Це дуже зручно при експериментах із трансформерами і іншими архітектурами.
- **Широке ком’юніті та приклади.** Практично будь-яку задачу вже хтось реалізовував на PyTorch, тому завжди можна знайти приклади, туторіали або відповіді на форумах.
- **Легке встановлення.** У моєму випадку TensorFlow виявився дуже важким у встановленні. Часто виникали проблеми з CUDA, версіями драйверів, сумісністю з процесором або GPU. PyTorch виявився набагато зручнішим – достатньо однієї команди, і все працює. Також можна обрати точну конфігурацію для своєї системи прямо на офіційному сайті.

- **Збереження та завантаження моделей.** У PyTorch це відбувається дуже просто через torch.save() та torch.load(). Я зберігаю модель після навчання, перекидаю її на сервер і використовую для генерації.

У PyTorch існує можливість компілювати моделі, що в деяких випадках дозволяє суттєво підвищити їх швидкодію. Зокрема, функція torch.compile() перетворює модель в оптимізовану форму, яка виконується значно швидше. Після такої компіляції швидкість роботи моделі може зрости в кілька разів.

Крім цього, PyTorch має багато інструментів для оптимізації:

- Підтримка змішаної точності (mixed precision). Це коли деякі частини моделі обчислюються не в стандартному 32-бітному форматі, а у 16-бітному. Це дозволяє пришвидшити навчання на GPU, особливо на сучасних відеокартах, при цьому точність результатів майже не втрачається.
- Інструменти для профілювання продуктивності (torch.profiler) – з їх допомогою можна аналізувати час виконання кожної операції та знаходити “вузькі місця”.
- Інтеграцію з бібліотеками на базі CUDA, що дозволяє ефективно використовувати апаратні ресурси.
- TorchScript – ще один інструмент для перетворення моделей у більш оптимізовану форму, яку можна запускати незалежно від Python-оточення, наприклад, на мобільних пристроях.

Для написання плагіна під Visual Studio Code я використовував звичайний JavaScript.

Основні причини мого вибору

- **Офіційна підтримка.** VS Code добре підтримує плагіни на JavaScript, є офіційна документація, шаблони та багато прикладів.
- **Досвід.** Я вже мав базові знання JavaScript, тому було легше почати. TypeScript потребує додаткового розуміння типів, що зайняло б більше часу.

- **Потреби проекту невеликі.** У моєму випадку плагін виконує лише базову функцію – відправляє запит на сервер і вставляє відповідь. Для цього достатньо можливостей JavaScript.

Реалізація серверної частини виконана на Python, щоб залишити усе в одному стеку і було менше проблем з інтеграцією моделі. Сервер реалізований за допомогою бібліотеки Flask.

Переваги такого підходу:

- **Проста логіка.** Мені потрібно лише один маршрут (/generate), який приймає текст, передає його моделі та повертає результат. Flask ідеально підходить для цього.
- **Сумісність з моделлю.** Так як модель написана на PyTorch, і працює у Python, було логічно залишити й сервер у тому ж середовищі.
- **Можливість легкої відладки.** Я можу швидко змінювати код сервера, тестувати запити з Postman, запускати його навіть локально – все просто і швидко.

3.3 Алгоритмізація та програмування програмних модулів

Метою цього розділу є пояснення того, як функціонує модель на рівні коду та архітектури: як вона розуміє контекст, навчається та обробляє запити.

Потім процес навчання: як дані перетворюються на токени, яка використовується функція втрат, який оптимізатор та стратегія навчання. Далі розповім про серверну частину, яка приймає запити від плагіна й викликає модель для генерації коду, а також як працює сам плагін VS Code: як він, формує запит і відображує результат.

Загалом, цей розділ має на меті показати, як ідеї з теорії машинного навчання втілюються у практичну систему, що працює в реальному середовищі. Щоб краще зрозуміти, як функціонує така система, важливо ознайомитися з її

базовими архітектурними принципами – зокрема з моделлю, яка лежить в її основі.

Transformer – це архітектура нейронної мережі, що здійснила революцію в обробці послідовностей. Цю архітектуру вперше представили у 2017 році в роботі “Attention is All You Need”, і відтоді вона стала основою більшості сучасних моделей. Хоча існує чимало спроб створити альтернативні архітектури, які б перевершили Transformer, на практиці вони часто виявляються менш ефективними, гірше масштабуються або зрештою комбінуються з елементами самого Transformer, щоб досягти його ефективності.

На відміну від попередніх рішень, таких як рекурентні нейронні мережі (RNN) або LSTM, Transformer не обробляє дані послідовно. Його ключова перевага в тому, що він може "бачити" всю послідовність одночасно. Це значно підвищує його ефективність для паралельних обчислень і дозволяє моделі враховувати як короткостроковий, так і довготривалий контекст [5, с. 2].

Механізм уваги (Attention) є центральним компонентом архітектури Transformer. На відміну від традиційних рекурентних мереж, де дані обробляються послідовно, механізм уваги дозволяє кожному окремому токену взаємодіяти з усіма іншими токенами в послідовності одночасно. Це досягається завдяки ефективним матричним операціям, що, в свою чергу, забезпечує можливість паралелізації обчислень [5, с. 2].

Серед різних типів механізмів уваги, найфундаментальнішим є Self-Attention (увага до самого себе). Його принцип полягає в тому, що кожен елемент послідовності обчислює свою залежність або зв'язок з усіма іншими елементами тієї ж послідовності [5, с. 2].

Для реалізації self-attention кожен токен з послідовності проєктується трьома окремими лінійними шарами наступним чином [5, с. 4]:

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

Де:

- Q (Query) – запит
- K (Key) – ключ
- V (Value) – значення

Ці проєкції обчислюються за допомогою окремих лінійних шарів. Далі увага розраховується через добуток запитів на транспоновані ключі, після чого цей результат масштабується, ділячись на корінь квадратний з розмірності простору ключів. Отримані значення проходять через softmax для формування ваг, які використовуються для зважування значень V. Це дозволяє кожному токену отримати контекстуалізовану інформацію з усієї послідовності [5, с. 4]. Функція softmaxпрацює наступним чином:

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Де:

- x^i — це і-й елемент вхідного вектору логітів,
- e^{xi} — експонента кожного значення,
- знаменник — сума експонент усіх елементів вектора.

Це означає, що кожне значення буде нормалізовано в межах [0, 1], і вся сума буде дорівнювати 1, формуючи ймовірнісний розподіл. Чим вище значення, тим більша його "вага" у розподілі. У контексті механізму уваги це означає, що токени з вищою релевантністю до запиту отримують більшу вагу [5, 4 с.].

Цей тип уваги у статті названо **Scaled Dot-Product Attention**. Автори відзначають, що при великих розмірностях dkd_kdk звичайна dot-product attention (без масштабування) призводить до дуже великих значень у softmax, що знижує градієнти. Тому введення масштабування критично покращує стабільність моделі при навчанні великих моделей [5, 4 с.].

На цьому етапі до scaled scores може додаватися маска – наприклад, застосовується маска, яка забороняє токену бачити “майбутні” токени. Це потрібно для навчання моделі, щоб вона не могла бачити майбутнє [5, с. 5].

Отже, ми маємо scaled attention scores – це по суті матриця, яка показує, наскільки “схожі” наші запити (Q) на ключі (K). Але ці сирі значення ще не готові до використання, адже вони не мають нормованого масштабу. Щоб перетворити їх на те, що ми називаємо “attention weights” (фактично, це показує, наскільки модель повинна “зосередитися” на певних елементах), ми застосовуємо функцію Softmax. Вона діє на кожен рядок матриці, приводячи значення до зрозумілого, нормованого вигляду [5, с. 4]. Графічний приклад того як відбувається процесс обрахування уваги наведено на рис. 3.2.

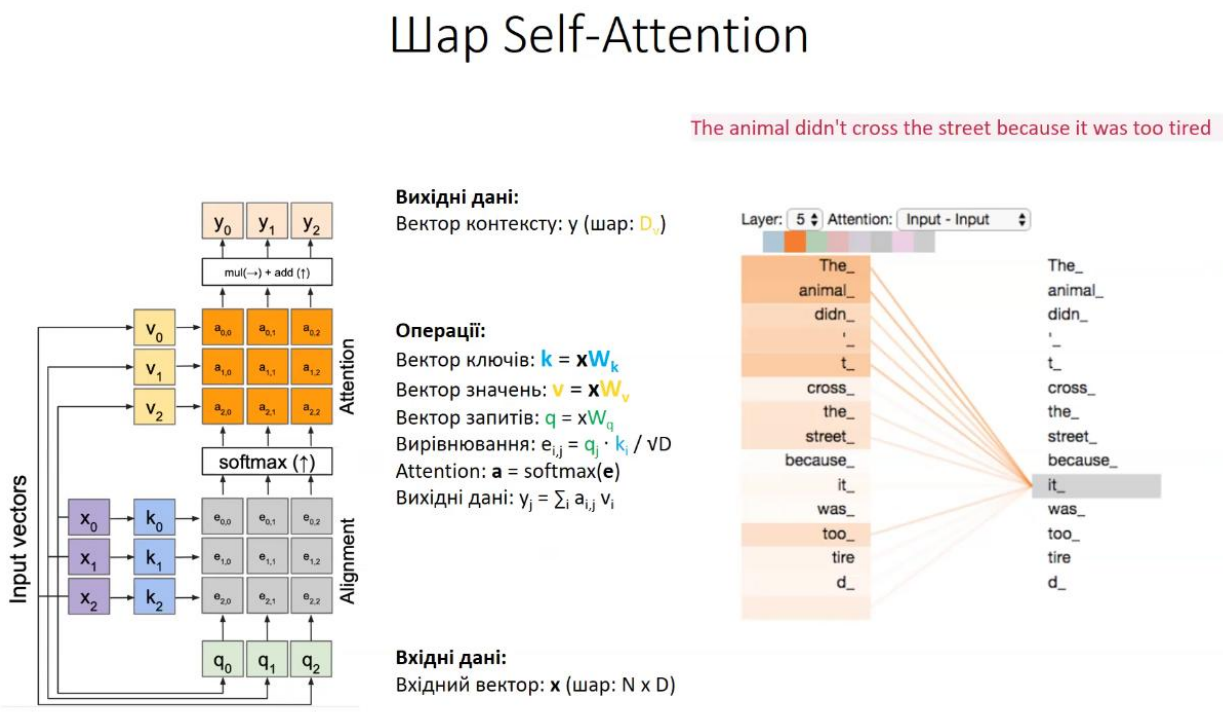

Рис. 3.2 Приклад обчислення уваги. (Школа штучного інтелекту, 2024) [6].

Softmax – це функція активації, яка перетворює довільні дійсні числа на розподіл ймовірностей. Ця функція робить так, що кожен елемент стає невід’ємним числом від 0 до 1, а вся сума – дорівнює 1. Тобто, після softmax ми отримуємо який відсоток уваги (вага) слід приділити кожному токену в послідовності. Ці ваги уваги (attention weights) використовуються для зваженого сумування векторів значень (V): Output = Attention weights · V

Але азовий механізм уваги обробляє залежності між токенами, використовуючи лише один “канал”. Проте, оскільки різні аспекти (як-от граматика, синтаксис чи семантика) можуть бути важливими для розуміння тексту, було розроблено концепцію мультиголової уваги(multihead attention) [5, с. 5].

Суть цього підходу полягає в тому, що модель створює кілька окремих “головок”, кожна з яких незалежно проєктує вхідні дані у власні простори запитів (Q), ключів (K) та значень (V). Кожна така “голова” виконує свій розрахунок уваги (self-attention) з унікальними параметрами, дозволяючи їй зосередитися на конкретних нюансах вхідної послідовності [5, с. 5].

Після того, як кожна голова згенерує свій контекстний вихід, усі ці результати об'єднуються шляхом конкатенації. Утворена велика матриця далі проходить через лінійний шар, який ефективно комбінує інформацію, отриману від усіх голів, в єдину цілісну репрезентацію [5, с. 5].

Такий підхід дозволяє моделі не просто зважувати зв’язки між токенами, а аналізувати їх з кількох перспектив одночасно. Наприклад, одна голова може зосередитись на локальній структурі речення, інша – на довготривалих залежностях, а ще інша – на синтаксичних ролях [5, с. 7] (див. рис. 3.3).

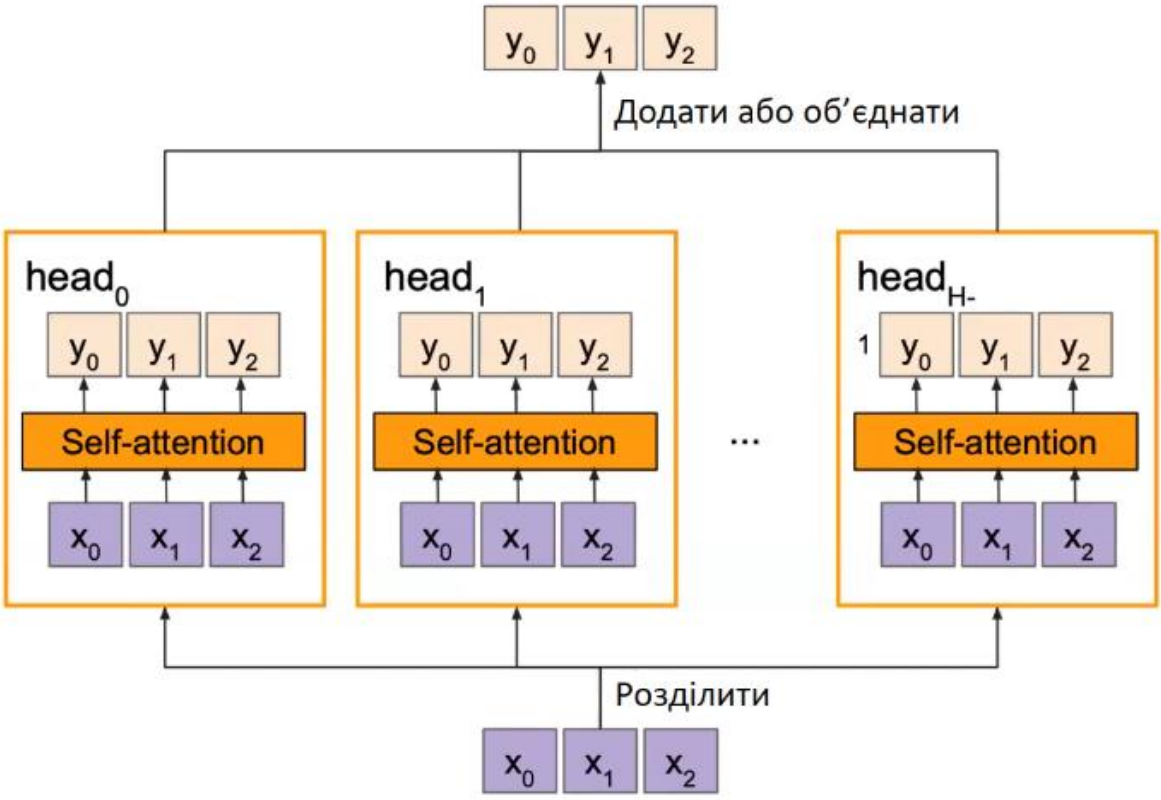

Рис. 3.3 Мультиголова увага [6].

Також варто зазначити, що Transformer обробляє всю послідовність паралельно, він не знає, в якому порядку йдуть токени. Тому, щоб надати моделі інформацію про порядок, використовують позиційне кодування – тобто додають до векторів токенів додаткову інформацію про їхню позицію [5, с. 6].

Класичне (синусоїдальне) позиційне кодування полягає в тому, що кожній позиції в послідовності відповідає унікальний набір значень, які додаються до вхідних векторів токенів. Ідея цього підходу полягає в тому, щоб кодувати позиції подібно до бінарних чисел, але у вигляді синусоїд і косинусів з різними частотами, що дозволяє моделі розпізнавати як абсолютні позиції, так і відносні відстані між токенами. Приклад того яка ідея була за синусоїдальним кодуванням на рис. 3.4.

Це кодування є:

- є детермінованим (не навчається) [5, с. 6];
- дозволяє моделі дізнатись не лише позицію токена, а й відстань між токенами (через різницю фаз) [5, с. 6].

Це підхід з оригінального Transformer. Однак у цього підходу є суттєвий недолік: модель не має прямого доступу до інформації про позицію токена. Замість цього вона змушена запам’ятовувати, як виглядає кожна позиція, що ускладнює генерацію на довгих послідовностях і вимагає перенавчання або тонкого налаштування при зміні довжини вхідного контексту. Приклад того як вигладає матриця позицій яка додається до векторів знаходиться на рис. 3.5.

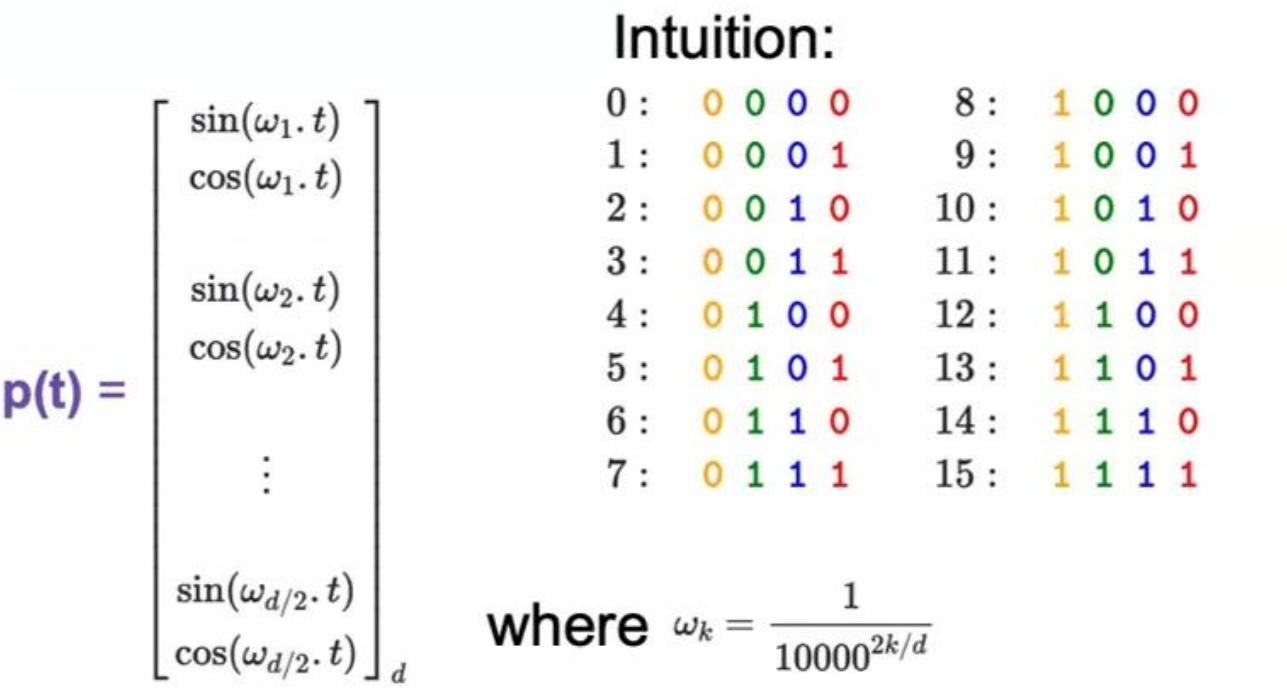

Рис. 3.4 Ідея за синусоїдальним позиційним кодуванням. [6].

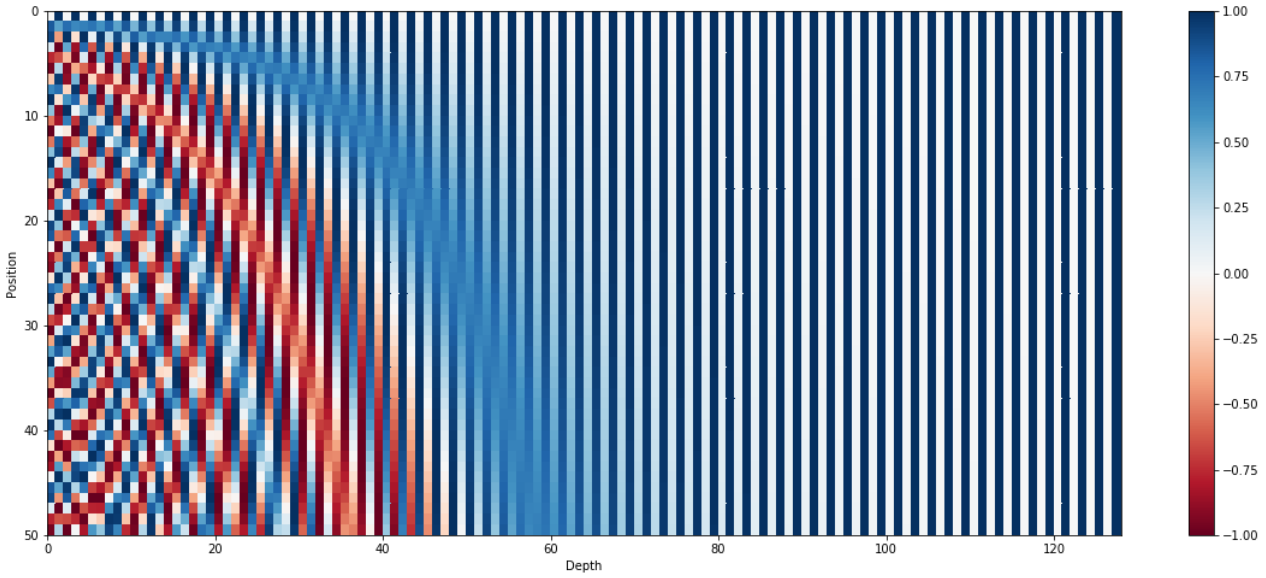

Рис. 3.4 Результат обчислення позиційної матриці. [22].

Більш сучасним та ефективним підходом до позиційного кодування є RoPE (Rotary Positional Embedding). На відміну від традиційних методів, RoPE не додає позиційні вектори до початкових, а модифікує самі вектори слів, “обертаючи” їхні компоненти у багатовимірному просторі. Це обертання залежить від позиції токена в послідовності [18, с.2].

Найголовніша особливість RoPE – у тому, що він не прив’язується до абсолютної позиції токена. Замість цього він повертає вектори, “вшиваючи” інформацію про відносне розташування: модель розуміє, наскільки один елемент ближче або далі від іншого. Такий підхід дозволяє їй краще узагальнювати – навіть на послідовностях, довших за ті, на яких вона тренувалась. Завдяки своїй обертовій природі, RoPE зберігає просторову структуру, що критично важливо для довгих текстів, де зв’язки між токенами можуть тягнутись через сотні, а то й тисячі позицій [18, с.2].

Цей метод полягає в тому, щоб перетворити позицію токена в обертання у векторному просторі. Уявімо, що кожен вектор слова – це комплексне число, а позиція – це кут, на який ми обертаємо це число. Таким чином, якщо ми одночасно обертаємо два токени на однаковий кут (тобто зсуваємо їх однаково в послідовності), кут між ними – а отже і скалярний добуток – не змінюється. Це означає, що відносна позиція зберігається [11].

Інакше кажучи, RoPE забезпечує, щоб операція скалярного добутку між токенами – яка лежить в основі self-attention – була чутливою саме до відносних, а не абсолютних позицій. Це досягається завдяки обертанню пари координат у векторі ембеддінгу кожного токена (наприклад, перші дві координати обертаються на певний кут, потім наступні дві – на інший, і так далі) [11]. Приклад обчислення позиції за допомогою RoPE наведено на рис.

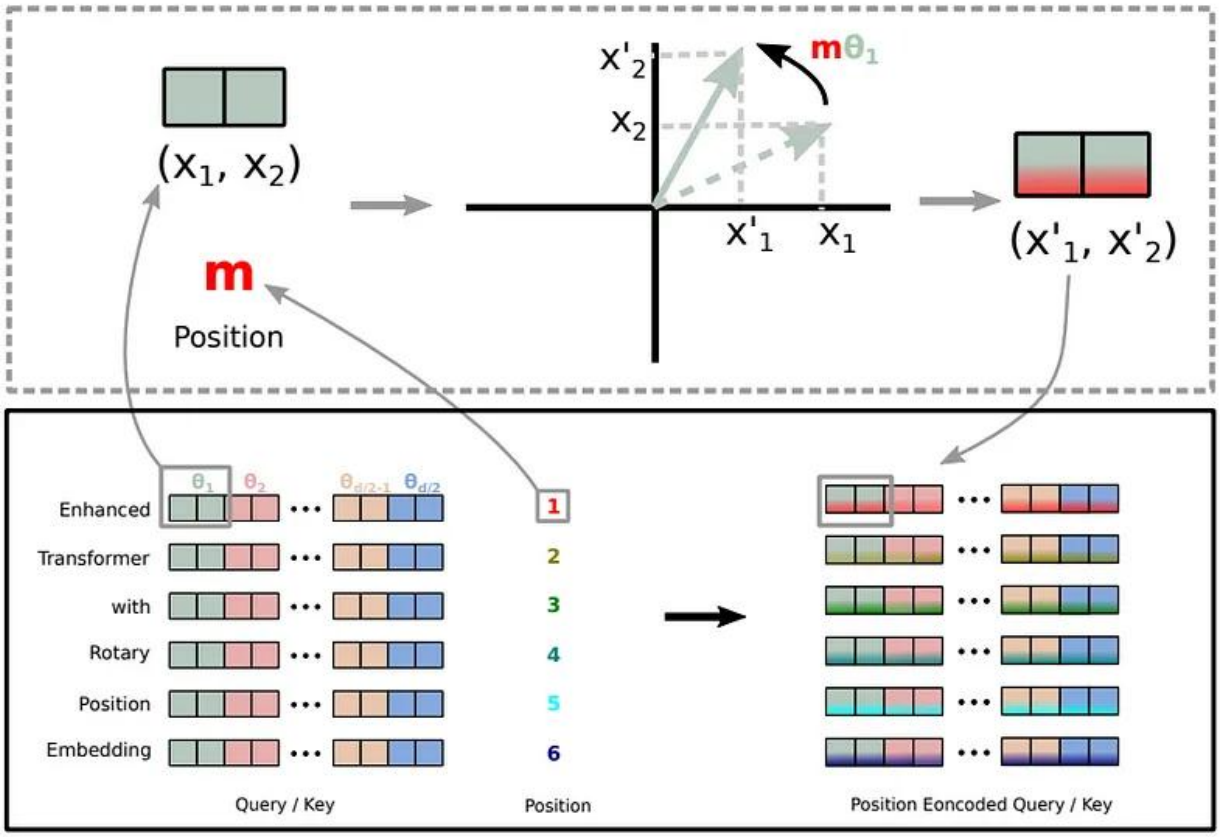

Figure 1: Implementation of Rotary Position Embedding(RoPE).

Рис. 3.5 Обчислення позиції у RoPE. [22].

Формально, RoPE реалізується як множення вектора токена на блокову діагональну матрицю повороту, побудовану з кутів, які відповідають його позиції. Таким чином, позиційне кодування стає мультиплікативним, а не адитивним, як у синусоїдальних ембеддінгах [11].

У підсумку, RoPE поєднує переваги абсолютного і відносного позиціонування, залишаючись при цьому простим і ефективним для реалізації. У багатьох експериментах, описаних у статті, він або не поступався, або перевершував класичні позиційні кодування – включно з синусоїдальними кодуваннями.

В основі будь-якої сучасної великої мовної моделі (LLM) знаходиться трансформер-блок, яку багато разів повторюють у мережі.

Першим і головним елементом є механізм самоуваги (self-attention). Після фази уваги дані передаються у багатошарову перцептронну мережу (MLP або feedforward-мережу). Це компонент, який застосовується до кожного токена

окремо. Як правило, він складається з двох лінійних (fully-connected) шарів із нелінійною функцією активації між ними, найчастіше GELU. MLP дозволяє здійснювати глибшу обробку інформації – наприклад, виявляти складні шаблони в даних, узагальнювати або трансформувати інформацію, яку зібрала self-attention [5, с.3].

Обидві частини – і увага, і MLP – обгорнуті у допоміжні шари: нормалізацію (LayerNorm) та dropout. Нормалізація стабілізує обчислення та сприяє швидшому й стабільнішому навчанню, вирівнюючи розподіли всередині шару. Dropout же випадково вимикає частину нейронів під час тренування, що зменшує ймовірність перенавчання і змушує модель не покладатися лише на окремі особливості вхідних даних [5, с.3].

Крім того, трансформер-блок реалізує залишкові з’єднання (residual connections) – тобто, до результату кожної фази (уваги або MLP) додається вхідний сигнал. Це допомагає зберегти початкову інформацію та забезпечує кращу передачу градієнтів при зворотному поширенні помилки [5, с.3]. Приклад архітектури наведено на рис. 3.6

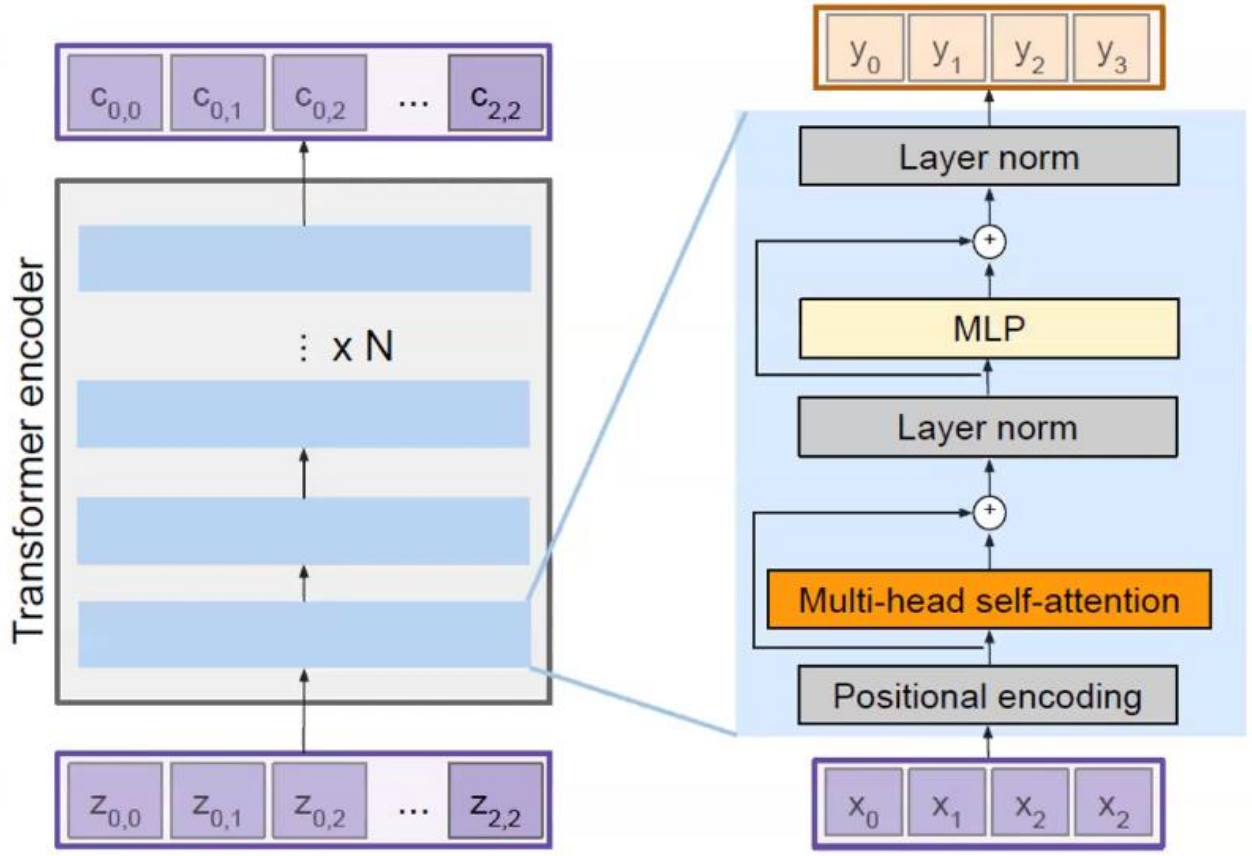

Рис. 3.6 Архітектура трансформера. (Школа штучного інтелекту, 2024) [6].

Однак із ростом розмірів моделей постає проблема обчислювальної ефективності: використання всіх параметрів одночасно стає надто затратним. Щоб подолати це, дослідники запропонували нові підходи до організації шарів моделі, зокрема архітектуру Mixture of Experts (MoE), яка дозволяє зменшити обчислювальні витрати без втрати якості [19].

Mixture of Experts – це підхід, у якому модель складається з великої кількості експертів(невеликих підмоделей, зазвичай MLP), але на кожному кроці обчислень активується лише невелика їх частина [19]. Реалізація MoE наведена на рис. 3.7

У звичайному шарі трансформера всі параметри використовуються завжди. У MoE навпаки – для кожного токена або прикладу модель обирає лише декілька експертів, які будуть використані. Цей процес називається роутинг (routing).

Роутинг реалізується за допомогою роутерної функції, в моєму випадку це невеликий лінійний шар, яка для кожного входу обирає, які експерти будуть

активовані. Найчастіше активуються 2 найкращих експерти, що відповідають за конкретний токен або контекст, щоб не активувати надто багато експертів і втратити всі переваги експертів [19]. Реалізація роутингу наведена на рис. 3.7. Код модифікований автором, були використані звичайні лінійні шари без додаткових ускладнень.

```
class Gate(nn.Module):
  def __init__(self, config: ModelConfig):
    super().__init__()
    self.dim = config.dim
    self.topk = config.n_activated_experts
    self.n_groups = config.n_expert_groups
    self.topk_groups = config.n_limited_groups
    self.score_func = config.score_func
    self.linear = nn.Linear(self.dim, config.n_routed_experts, bias=config.moe_bias)

  def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    scores = self.linear(x)

    if self.score_func == "softmax":
      scores = scores.softmax(dim=-1, dtype=torch.float32)
    else:
      scores = scores.sigmoid()
    original_scores = scores

    if self.n_groups > 1:
      scores = scores.view(x.size(0), self.n_groups, -1)
      group_scores = scores.topk(2, dim=-1)[1]

      indices = group_scores = scores.topk(2, dim=-1)[1]
      mask = scores.new_ones(x.size(0), self.n_groups, dtype=bool).scatter_(1, indices, False)
      scores = scores.masked_fill_(mask.unsqueeze(-1), float("-inf")).flatten(1)
    indices = torch.topk(scores, self.topk, dim=-1)[1]
    weights = original_scores.gather(1, indices)
    if self.score_func == "sigmoid":
      weights /= weights.sum(dim=-1, keepdim=True)

    return weights.type_as(x), indices
```

Рис. 3.7 Код для реалізації роутингу (модифіковано автором на основі DeepSeek, 2024) [7].

У моделі DeepSeek-R1 архітектура активно використовує MoE-підхід. Зокрема, перші 3 шари є щільними (dense) – вони містять стандартну Feed-Forward Network (FFN), а починаючи з 4-го і до 61-го шару, використовується MoE-шар. Таким чином, більшість параметрів моделі є неактивними на кожному окремому кроці, що значно знижує обчислювальні витрати [10].

У DeepSeek-R1 при 671 мільярді загальних параметрів, активується лише близько 37 мільярдів параметрів для кожного токена, що забезпечує високу ефективність без втрати точності моделі [10]. Код для реалізації експертів

наведено на рис. 3.8. У моїй реалізації я використовую звичайні лінійні шари як експертів без складного розділення на окремі відеокарти (GPU). Усі експерти виконуються в межах однієї одиниці, що спрощує реалізацію.

Переваги MoE:

- **Економія обчислень** – модель має багато параметрів, але активується лише частина з них. Наприклад, якщо у нас є 64 експерти і активуються лише 2 – ми задіюємо лише дуже малу частину параметрів у кожному проході [19].
- **Масштабованість** – можна створити модель з десятками або навіть сотнями мільярдів параметрів, яка все ще запускається з обмеженими ресурсами [19].
- **Гнучкість** – різні експерти можуть спеціалізуватися на різних типах запитів: одні на коді, інші на природній мові, ще інші на обчисленнях [19].

```python
class Expert(nn.Module):
    def __init__(self, dim: int, inter_dim: int):
        super().__init__()
        self.w1 = nn.Linear(dim, inter_dim)
        self.w2 = nn.Linear(inter_dim, dim)
        self.w3 = nn.Linear(dim, inter_dim)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
      return self.w2(F.silu(self.w1(x)) * self.w3(x))

class MoE(nn.Module):
  def __init__(self, config: ModelConfig):
    super().__init__()
    self.dim = config.dim
    self.n_routed_experts = config.n_routed_experts
    self.n_activated_experts = config.n_activated_experts
    self.gate = Gate(config)
    self.experts = nn.ModuleList([Expert(config.dim, config.moe_inter_dim)
                                  for i in range(self.n_routed_experts)])
    self.shared_experts = MLP(config.dim, config.n_shared_experts * config.moe_inter_dim)

  def forward(self, x: torch.Tensor) -> torch.Tensor:
    shape = x.size()
    x = x.view(-1, self.dim)
    weights, indices = self.gate(x)

    y = torch.zeros_like(x)
    counts = torch.bincount(indices.flatten(), minlength=self.n_routed_experts).tolist()

    for i in range(self.n_routed_experts):
      if counts[i] == 0:
        continue

      expert = self.experts[i]
      idx, top = torch.where(indices == i)
      y[idx] += expert(x[idx]) * weights[idx, top, None]

    z = self.shared_experts(x)
    if world_size > 1:
      dist.all_reduce(y)

    return (y + z).view(shape)
```

Рис. 3.8 Код для реалізації експертів (модифіковано автором на основі DeepSeek, 2024) [7].

Для того щоб забезпечити швидкість генерації, особливо під час обробки довгих контекстів, у трансформерах використовується механізм кешування. Його основна мета – уникнути повторного обчислення результатів для вже оброблених токенів. У класичному трансформері при кожному кроці генерації новий токен потребує обчислення уваги до всіх попередніх токенів. Без кешу це означає, що з кожним новим токеном обсяг обчислень збільшується, що призводить до значного навантаження на процесор чи графічну карту і до зростання затримки відповіді [19].

Щоб цього уникнути, трансформери зберігають ключі (keys) і значення (values) для кожного попереднього токена окремо в пам’яті – це і є кеш. Коли генерується наступний токен, модель обчислює лише запит (query) для нового токена, і порівнює його з усіма збереженими ключами в кеші. Таким чином, модель не обробляє всю послідовність повторно, а лише оновлює кеш і виконує мінімально необхідні обчислення. Єдина проблема в цьому кеші що він займає дуже багато пам’яті, що намагається вирішити нова архітектура трансформера від компанії DeepSeek яка називається латентна увага(latent attention) [19].

Один із запропонованих підходів полягає в тому, щоб працювати не з усіма ключами і запитами кожного окремого токена, а стискати їх в кілька узагальнених представлень. Ці представлення називаються “латентними” і зберігають суть усього попереднього контексту в компактнішій формі. Таким чином, замість зберігання великого кешу для кожного токена, модель оперує значно меншою кількістю структурованих об’єктів, що дозволяє зменшити вимоги до пам’яті, зберігаючи при цьому якість генерації [19]. Фрагмент адаптованої реалізації цього механізму, на основі коду з офіційного репозиторію DeepSeek, наведено в додатку А.

Зазвичай, коли модель генерує нове слово, вона зберігає в кеші ключі та запити для кожного попереднього слова – і чим довший текст, тим більшим стає цей кеш. Але в цьому випадку все працює інакше. Замість того, щоб накопичувати повний набір даних, модель оперує компактним, латентним простором. У кеші зберігаються лише агреговані латентні ключі й запити. Цей підхід значно знижує вимоги до пам’яті [19].

Хоча під час навчання така схема не дає значного виграшу в швидкості, оскільки обчислення все одно виконуються над повною послідовністю в паралельному режимі, її переваги стають особливо помітними при інференсі, тобто у режимі покрокової генерації тексту.

Навчання моделі починається з підготовки токенів. Я використовую два основні джерела даних:

- **C4 (Colossal Clean Crawled Corpus)** – англомовний датасет, широко використовуваний у тренуванні великих мовних моделей.
- **Код з GitHub** – датасет, що містить приклади коду з відкритих репозиторіїв.

Для ефективної обробки великих обсягів даних я використовую streaming-режим з datasets, що дозволяє не завантажувати весь датасет в оперативну пам’ять одразу, а обробляти його по частинах. Алгоритм токенізації модифіковано автором до власних потреб, зокрема було змінено датасети та змінні, на основі ідеї Andrej Karpathy [8] наведено на рис. 3.9.

```
def prepare_dataset():
    c4_dataset = load_dataset("c4", "en", split="train", streaming=True)
    github_dataset = load_dataset("bigcode/github_codespell_80k", split="train", streaming=True)

    combined_iterator = itertools.chain.from_iterable(itertools.zip_longest(c4_dataset, github_dataset))

    nprocs = max(1, os.cpu_count()//2)
    with mp.Pool(nprocs) as pool:
        shard_index = 0
        all_tokens_np = np.empty((shard_size,), dtype=np.uint16)
        token_count = 0
        progress_bar = None
        for tokens_batch in pool.imap(tokenize, combined_iterator, chunksize=16):

            if token_count + len(tokens) < shard_size:
                all_tokens_np[token_count:token_count+len(tokens)] = tokens
                token_count += len(tokens)
            else:
                split = "val" if shard_index == 0 else "train"
                filename = os.path.join(DATA_CACHE_DIR, f"edu_{split}_{shard_index:06d}")
                remainder = shard_size - token_count
                progress_bar.update(remainder)
                all_tokens_np[token_count:token_count+remainder] = tokens[:remainder]
                write_datafile(filename, all_tokens_np)
                shard_index += 1
                progress_bar = None
                all_tokens_np[0:len(tokens)-remainder] = tokens[remainder:]
                token_count = len(tokens)-remainder

        if token_count != 0:
            split = "val" if shard_index == 0 else "train"
            filename = os.path.join(DATA_CACHE_DIR, f"edu_{split}_{shard_index:06d}")
            write_datafile(filename, all_tokens_np[:token_count])
```

Рис. 3.9 Модифікований алгоритм токенізації.

Коли ми навчаємо модель, потрібно розуміти наскільки добре вона справляється зі своїм завданням. Для цього використовують функцію втрат. Вона порівнює те, що модель згенерувала, з тим, якою мала бути правильна відповідь і повертає помилку моделі, яка потім використовується в backpropagation.

Для задачі передбачення наступного токена (слова чи символу) найбільш поширеною є функція втрат під назвою крос-ентропія. Вона вимірює різницю між тим, що модель передбачає у вигляді ймовірностей для кожного можливого токена, і тим, який токен насправді мав бути наступним.

Крос-ентропія обчислюється наступним чином [25]:

$$L = -\sum_{k=1}^{K} y_k \log(p_k)$$

Де:

- N — кількість класів (у випадку мовної моделі — розмір словника токенів),
- y_k — справжній розподіл (one-hot: 1 для правильного токена, 0 для всіх інших),
- p_k — ймовірність, яку передбачила модель для токена i,
- $\log(p_k)$ — логарифм передбаченої ймовірності.

Таким чином, крос-ентропія штрафує модель, якщо вона призначає низьку ймовірність правильному токену, і навпаки — "винагороджує", якщо ця ймовірність висока. Це мотивує модель навчатися точніше передбачати наступні токени у послідовності [25].

Після того як ми оцінили, наскільки сильно модель помиляється, наступне завдання – зменшити цю помилку. І тут у гру вступають так звані оптимізатори – спеціальні алгоритми, які поступово коригують внутрішні параметри моделі, тобто ваги нейронної мережі. Мета проста: зробити так, щоб з кожним кроком навчання модель помилялася все менше й менше.

Серед усього розмаїття оптимізаторів один з найвідоміших – Adam (Adaptive Moment Estimation). Він став своєрідним стандартом у машинному навчанні. Його популярність не випадкова, Adam вдало поєднує два підходи:

1. **Врахування середнього напрямку градієнтів**
   При оновленні ваг модель не просто дивиться на поточний градієнт (напрямок, у якому треба рухатися, щоб зменшити помилку), а також бере до уваги середнє значення градієнтів за попередні кроки [20, с.3].
2. **Врахування середньої швидкості зміни градієнтів**
   Adam не просто рухається по градієнтах, він ще й “пам'ятає” середню швидкість їхньої зміни. Це дозволяє йому автоматично налаштовувати розмір кроку (learning rate) для кожного окремого параметра моделі. Adam намагається зменшить крок, щоб не перескочити оптимальне значення або збільшити. А якщо градієнти стабільні й невеликі, він зробить крок більшим, щоб швидше дійти до мети. Завдяки цьому модель навчається ефективніше і швидше знаходить найкращі налаштування [20, с.2].

Варіант Adam під назвою AdamW додатково вводить регуляризацію, яка допомагає моделі уникати перенавчання. Це означає, що модель не буде “запам’ятовувати” лише конкретні приклади, а зможе краще узагальнювати нові дані [21].

Регуляризація – це техніка, яка допомагає моделі уникати перенавчання (overfitting). Перенавчання відбувається тоді, коли модель надто добре “запам’ятовує” навчальні дані і втрачає здатність узагальнювати нову інформацію, тобто поводиться погано на невідомих прикладах [21].

Коли ми говоримо про покращення моделі, також використовують “weight decay”. Ідея полягає в тому, щоб злегка рухати ваги моделі до нуля при кожному оновленні. Це ніби додає певне “стримування”, яке не дозволяє вагам ставати занадто великими і складними [21].

Крім оптимізатора, дуже важливо керувати темпом навчання, який називається learning rate. Занадто великий темп може призвести до нестабільності, коли модель “стрибає” навколо мінімуму функції втрат, а занадто малий – уповільнює навчання.

Щоб модель добре вчилася, потрібно гнучко підходити до того, як швидко вона “запам'ятовує” інформацію – це називається темп навчання (learning rate). Для цього ми використовуємо scheduler – такий механізм, який протягом всього тренування підлаштовує цей темп. Наприклад, на початку навчання може бути “warmup” – модель повільно розганяється від дуже низького до нормального темпу навчання, так як на початку модель дуже нестабільна. Після цього learning rate поступово зменшується – це допомагає моделі краще підлаштуватись і працювати стабільніше на пізніших етапах навчання.

Вдалий вибір оптимізатора, scheduler’а та інших гіперпараметрів справді має велике значення. Від цього залежить не лише швидкість навчання, а й те, наскільки добре модель зрештою навчиться розуміти й генерувати текст.

Переднавчання – це початковий етап навчання нейронної мережі, під час якого модель вчиться розпізнавати загальні закономірності у великих обсягах текстових або кодових даних без конкретного цільового завдання. Метою переднавчання є сформувати “загальне розуміння” мови, граматики, логіки та контексту.

У випадку моєї моделі, яка працює з кодом, під час переднавчання вона навчається передбачати наступний токен за послідовністю попередніх токенів. Це називається задачою автопрогнозування (auto-regressive modeling). Модель отримує вхідний рядок токенів і намагається максимально точно вгадати, що буде далі. Так, вона опановує синтаксис, структуру коду, типові шаблони та логіку.

Цей етап має ключове значення. Саме тут вона здобуває свої “базові знання”, на яких потім можна будувати щось значно складніше: наприклад, навчати її доповнювати код або навіть створювати цілі функції.

У процесі переднавчання ми використовуємо функцію втрат – крос-ентропію. Простими словами, вона допомагає моделі зрозуміти, наскільки її передбачення відрізняються від правильних відповідей. Завдяки цьому модель поступово вчиться робити менш випадкові й точніші припущення.

Крос-ентропія обчислюється як міра "відстані" між імовірнісним розподілом, що видає модель, і фактичним (однозначним) розподілом правильних відповідей – чим більша ця відстань, тим більшим буде штраф, тобто втрата.

Після завершення переднавчання модель повинна мати достатній рівень “загальних знань” і стає основою для подальшого навчання з підкріпленням (RL), щоб адаптуватися під конкретні вимоги і покращити результати.

Після того модель вже освоїла основи мови або коду на етапі переднавчання, наступний крок – зробити її корисною, так як після переднавчання все що може модель це писати продовження для тексту, не розуміючи запити користувача. Тут на допомогу приходить навчання з підкріпленням (RL). Саме так працюють найсучасніші системи, як DeepSeek, і я планую використовувати схожий підхід для своєї моделі.

Навчання з підкріпленням – це коли модель або в цьому випадку її називають “агентом”, вчиться приймати рішення крок за кроком у певному “середовищі”, щоб отримати якомога більше “нагороди”. У випадку генерації тексту чи коду, “агент" вирішує, який наступний фрагмент згенерувати, а потім отримує оцінку, наскільки цей вибір був вдалим. Ми хочемо, щоб модель прагнула до великої нагороди – це означає, що вона добре справляється з поставленим завданням і генерує якісний результат.

Але навіть попри те, що я маю код для переднавчання моделі і розумію його принципи, реалізація повноцінного переднавчання є дуже дорогою з точки зору ресурсів і часу. Тренування великих моделей вимагає потужних графічних процесорів, значного обсягу оперативної пам’яті і тривалого часу обчислень, що суттєво обмежує можливості для виконання такого етапу самостійно. Через це я прийняв рішення використати вже готову базову модель від DeepSeek. Ця

модель стала основою для моєї власної, і я планую проводити подальше покращення на її базі за допомогою навчання з підкріпленням (reinforcement learning).

Варто зазначити, що навіть етап навчання з підкріпленням є досить витратним і складним – він вимагає значної кількості ітерацій, а також ретельного налаштування функції винагороди і середовища, в якому працює модель. Проте порівняно з повним переднавчанням з нуля, цей підхід дозволяє ефективніше використовувати наявні ресурси, фокусуючись на вдосконаленні моделі з урахуванням конкретних цілей і задач.

Я також намагався навчати меншу модель з нуля, сподіваючись на більш доступні витрати обчислювальних ресурсів. Однак навіть цей підхід виявився досить дорогим, а результати – незадовільними. Менша модель не досягала потрібного рівня якості, оскільки через обмежену кількість параметрів вона не могла ефективно моделювати складні закономірності коду і тексту. Тому, враховуючи обмеження у ресурсах і потребу в якості, вибір на користь використання вже попередньо навчених моделей від провідних розробників видається найбільш розумним і практичним. Використовувати я буду найменшу преднавчену модель з 1.5 мільярдами параметрів від компанії DeepSeek, DeepSeek-V3, яка є готовою до подальшого донавчання.

Донавчання буде відбуватися за допомогою навчання з підкріпленням (Reinforcement Learning, RL). Ідея цього типу навчання дуже проста:

- Модель генерує послідовність токенів.
- Кожне її передбачення оцінюється спеціальною функцією винагороди (reward function).
- Якщо модель передбачила правильний токен, вона отримує нагороду.
- Якщо передбачення було помилковим або неякісним, то нагорода буде низькою або відсутньою.

Мета навчання – навчитися генерувати такі токени, які максимізують суму отриманих нагород.

Щоб це реалізувати, модель вважається агентом у середовищі:

- Агент приймає рішення – який токен згенерувати наступним.
- За кожен вибір він отримує сигнал винагороди.
- Модель постійно адаптує свої параметри під, щоб отримувати більше нагороди.

Таким чином, навчання з підкріпленням допомагає їй ще краще генерувати код.

Реалізація серверної частини

Сервер поєднує плагін VS Code, з яким працює користувач, з моделлю машинного навчання. Сервер написаний на Python, використовуючи фреймворк Flask. Flask дозволив дуже швидко зробити REST API, який приймає від користувача запита та відправляє їх моделі, а потім віддає результат назад клієнту у форматі JSON.

Сервер має одну основну точку входу – маршрут /chat (див. рис. 3.10), що приймає POST-запити. Ці запити містять у собі текстове повідомлення користувача у форматі JSON. Сервер перевіряє коректність отриманих даних, зокрема, чи є в запиті поле “message”. Якщо дані не відповідають вимогам, сервер повертає помилку з відповідним кодом 400. У разі успішного прийому повідомлення, воно передається у функцію обробки запиту process_request.

Функція process_request відповідає за основну логіку взаємодії з моделлю. Далі відбувається ітеративний цикл генерації тексту. Цикл завершується, якщо модель видає спеціальний токен кінця послідовності або якщо досягається максимальна довжина. Після генерації кожної послідовності токени очищаються від спеціальних символів і декодуються назад у текст за допомогою токенізатора.

В кінці функція повертає сформовану відповідь, при цьому всі символи нового рядка замінюються на HTML-розмітку для коректного відображення у плагіні.

На рівні Flask-маршруту /chat після отримання відповіді від process_request також обчислюється час обробки запиту. Це дозволяє

моніторити продуктивність і давати користувачу зворотний зв’язок про швидкість генерації.

У випадку виникнення будь-яких помилок під час обробки запиту сервер повертає відповідь з помилкою і кодом 500, що допомагає відлагоджувати проблеми.

Таким чином, ця проста, але ефективна серверна архітектура забезпечує зв’язок між моделлю машинного навчання та користувачем. Вона зручно розширюється, підтримує паралельну обробку запитів (threaded=True) і використовує всі можливості PyTorch для швидкої та економної генерації тексту.

```python
@app.route('/chat', methods=['POST'])
def chat():
    try:
        data = request.get_json()
        if not data or 'message' not in data:
            return jsonify({'error': 'Missing "message" in request'}), 400

        user_input = data['message']

        start_time = time.time()
        response = process_request(user_input)
        duration = time.time() - start_time

        return jsonify({'response': response, 'duration': duration})

    except Exception as e:
        return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, threaded=True)
```

Рис. 3.10 Шлях на сервері для генерації тексту

Останній елемент системи – плагін для Visual Studio Code є інтерфейсом взаємодії користувача з запущеною на сервері моделлю штучного інтелекту (див. рис. 3.10). Основна мета плагіна – забезпечити зручну комунікацію між розробником і мовною моделлю безпосередньо в межах середовища розробки.

Плагін реалізований за допомогою API розширень VS Code на JavaScript. Він складається з двох основних частин:

- **Back-end логіка** – взаємодія з сервером моделі, збереження історії повідомлень, обробка команд користувача.

- **Front-end (Webview)** – візуальна частина інтерфейсу (чат), що рендериться у вигляді WebView всередині VS Code.

Основна функціональність:

- **Відображення WebView:** при активації плагіна створюється панель з чатом. HTML-сторінка, що завантажується у WebView, реалізує інтерфейс з полем введення повідомлень, кнопкою надсилання та панеллю відображення історії.
- **Взаємодія з користувачем:** користувач може ввести текст у чаті, натиснути кнопку або клавішу Enter. Повідомлення надсилається до бекенду плагіна.
- **Передача повідомлень до моделі:** бекенд плагіна пересилає повідомлення через HTTP-запит (POST) на сервер. У відповіді повертається згенерований текст від моделі.
- **Збереження історії:** повідомлення зберігаються у глобальному стані (context.globalState), що дозволяє зберігати історію між сесіями VS Code.
- **Очищення чату:** Користувач може легко очистити історію чату за допомогою спеціальної кнопки.
- **Інтеграція з файловою системою:** Щоб модель краще могла розуміти контекст, плагін знаходить і читає вміст сусідніх файлів, які знаходяться поруч з тим, що зараз відкритий. Всі ці дані потім додаються до запиту, щоб модель мала більше контексту про проєкт.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Під час тестування системи, яка складається з мовної моделі, серверного модуля та плагіна для VS Code, я провів тестування, щоб перевірити, чи все працює правильно, стабільно, безпечно та зручно для користувача.

Основним типом тестування було функціональне тестування, бо головне – перевірити, як плагін, сервер і модель взаємодіють між собою. Я імітував звичайні дії користувача: запускав розширення в VS Code, підключався до сервера, відправляв текстовий запит і дивився, чи правильно модель повертає відповідь.

Наприклад, під час тесту згенерованого коду було перевірено, чи правильно плагін:

- надсилає запит до сервера;
- надсилає текст користувача як запит до моделі;
- отримує від сервера відповідь;
- відображає результат у вікні розширення.

У разі коректного виконання всіх кроків користувач отримує відповідь майже миттєво після натискання кнопки. У разі помилок (наприклад, сервер недоступний), система показує відповідне повідомлення про помилку.

Надсилання запитів: перевірялося, чи коректно формуються повідомлення перед відправленням. Також перевірялася реакція системи при спробі відправити порожній запит (див. рис. 4.1).

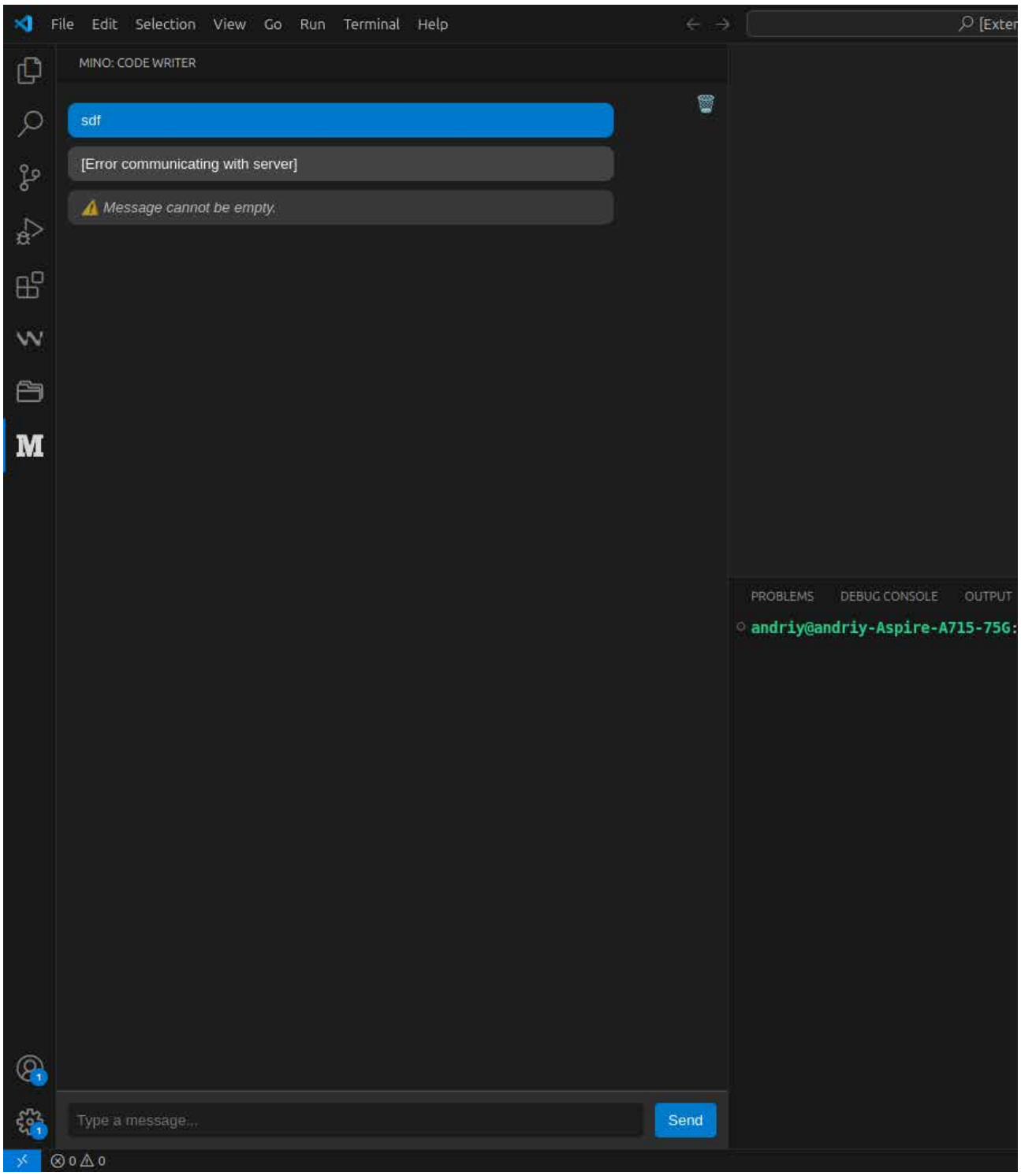

Рис. 4.1 Відправлення пустого запиту

Перевірка чи відбулася помилка під час відправлення на сервер запиту (див. рис. 4.2).

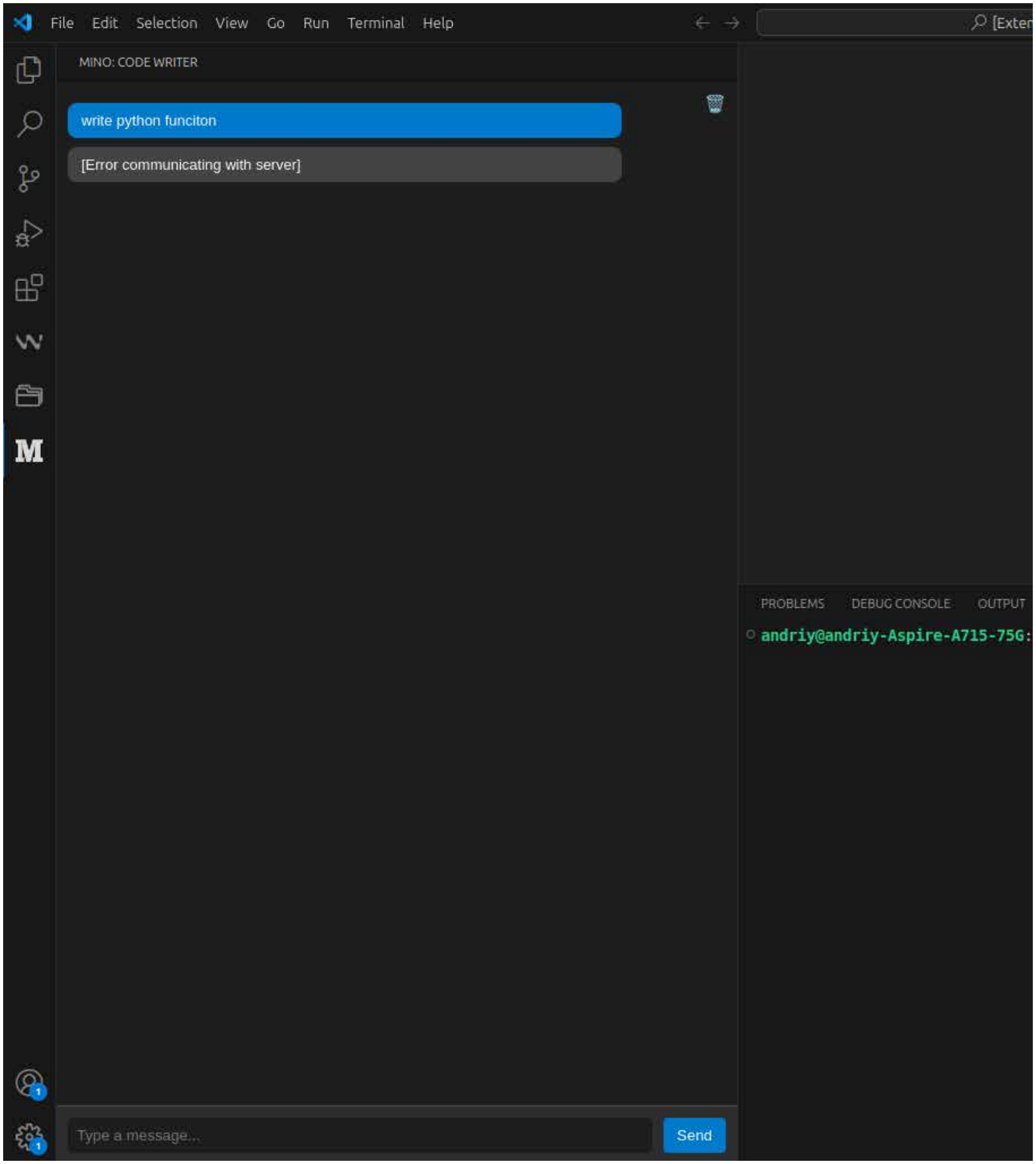

Рис. 4.2 Помилка підключення до серверу

Важливим аспектом тестування є перевірка реакції сервера на невалідні або неповні дані. Наприклад, у випадку, до сервера надходять запити без обов’язкового поля message, сервер повинен повернути відповідь із кодом з помилкою. Це реалізовано через перевірку наявності даних та ключа message. (див. рис. 4.3)

```python
@app.route('/chat', methods=['POST'])
def chat():
    try:
        data = request.get_json()
        if not data or 'message' not in data:
            return jsonify({'error': 'Missing "message" in request'}), 400

        user_input = data['message']

        start_time = time.time()
        response = process_request(user_input)
        duration = time.time() - start_time

        return jsonify({'response': response, 'duration': duration})

    except Exception as e:
        return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, threaded=True)
```

Рис. 4.3 Перевірка запиту

Тестування ініціалізації та навчання моделі

Після того як модель була реалізована, я переходжу до її ініціалізації та навчання. Спочатку створюю екземпляр моделі, передаючи необхідну конфігурацію, а потім переношу її на відповідний пристрій – CPU або GPU, залежно від умов. Щоб пришвидшити виконання, компілюю модель перед початком тренування.

Далі налаштовую оптимізатор і планувальник (scheduler) для контролю швидкості навчання. Спершу learning rate поступово зростає (етап warmup), а потім повільно знижується впродовж усього процесу тренування. Я поетапно завантажую батчі з тренувального датасету, подаю їх на вхід моделі, розраховую функцію втрат, виконую зворотне поширення помилки і оновлюю ваги.

Паралельно з цим періодично перевіряю якість моделі на валідаційному наборі даних. А ще зберігаю чекпоінти, щоб у будь-який момент можна було відновити тренування або порівняти результати на різних етапах (див. рис. 4.4).

```
num decayed parameter tensors: 50, with 124,354,560 parameters
num non-decayed parameter tensors: 98, with 121,344 parameters
using fused AdamW: True
validation loss: 10.9514
step     0 | loss: 10.955009 | lr 5.0562e-06 | norm: 15.3464 | dt: 50681.09ms | tok/sec: 10344.84
Model checkpoint saved at log/model_00000.pt
step   100 | loss: 6.515263 | lr 5.1067e-04 | norm: 1.0224 | dt: 2706.43ms | tok/sec: 193719.17
step   200 | loss: 5.832395 | lr 8.9995e-04 | norm: 0.4256 | dt: 2707.62ms | tok/sec: 193633.93
validation loss: 5.5848
step   300 | loss: 5.401945 | lr 8.9859e-04 | norm: 0.5298 | dt: 2709.02ms | tok/sec: 193534.32
step   400 | loss: 4.857390 | lr 8.9533e-04 | norm: 0.5464 | dt: 2707.61ms | tok/sec: 193634.78
validation loss: 4.5891
step   500 | loss: 4.550300 | lr 8.9020e-04 | norm: 0.5809 | dt: 4108.06ms | tok/sec: 127624.25
step   600 | loss: 4.154492 | lr 8.8322e-04 | norm: 0.4032 | dt: 2710.48ms | tok/sec: 193429.96
step   700 | loss: 4.169420 | lr 8.7442e-04 | norm: 0.4605 | dt: 2710.26ms | tok/sec: 193445.45
validation loss: 4.0704
step   800 | loss: 4.073336 | lr 8.6385e-04 | norm: 0.3204 | dt: 2710.07ms | tok/sec: 193459.17
step   900 | loss: 3.984676 | lr 8.5155e-04 | norm: 0.4528 | dt: 2710.11ms | tok/sec: 193456.63
validation loss: 3.8416
step  1000 | loss: 3.789350 | lr 8.3758e-04 | norm: 0.3107 | dt: 4106.60ms | tok/sec: 127669.61
Model checkpoint saved at log/model_01000.pt
step  1100 | loss: 3.674728 | lr 8.2200e-04 | norm: 0.3163 | dt: 2705.61ms | tok/sec: 193778.14
step  1200 | loss: 3.591383 | lr 8.0489e-04 | norm: 0.3003 | dt: 2706.07ms | tok/sec: 193745.26
validation loss: 3.7192
step  1300 | loss: 3.667581 | lr 7.8633e-04 | norm: 0.2770 | dt: 2709.26ms | tok/sec: 193517.19
step  1400 | loss: 3.752133 | lr 7.6641e-04 | norm: 0.2963 | dt: 2709.40ms | tok/sec: 193507.00
validation loss: 3.6254
step  1500 | loss: 3.687134 | lr 7.4522e-04 | norm: 0.3299 | dt: 4094.22ms | tok/sec: 128055.67
step  1600 | loss: 3.720565 | lr 7.2285e-04 | norm: 0.3370 | dt: 2708.66ms | tok/sec: 193559.72
step  1700 | loss: 3.549440 | lr 6.9942e-04 | norm: 0.2433 | dt: 2706.60ms | tok/sec: 193707.41
validation loss: 3.5637
step  1800 | loss: 3.469413 | lr 6.7503e-04 | norm: 0.3020 | dt: 2705.64ms | tok/sec: 193776.01
step  1900 | loss: 3.434795 | lr 6.4979e-04 | norm: 0.2808 | dt: 2701.34ms | tok/sec: 194084.42
validation loss: 3.5098
step  2000 | loss: 3.549697 | lr 6.2384e-04 | norm: 0.3052 | dt: 4097.88ms | tok/sec: 127941.21
Model checkpoint saved at log/model_02000.pt
step  2100 | loss: 3.553772 | lr 5.9728e-04 | norm: 0.3371 | dt: 2706.26ms | tok/sec: 193731.32
step  2200 | loss: 3.464810 | lr 5.7024e-04 | norm: 0.2669 | dt: 2708.88ms | tok/sec: 193544.28
validation loss: 3.4595
step  2300 | loss: 3.494261 | lr 5.4284e-04 | norm: 0.2625 | dt: 2706.27ms | tok/sec: 193730.69
```

Рис. 4.4 Результат тренування

Наступний тест перевіряє завантаження моделі (див. рис. 4.5) та запуск Flask-сервера для обробки простих текстових запитів через шлях /chat (див. рис. 4.6).

```python
def load_model():
    checkpoint = torch.load(checkpoint_path, map_location=device, weights_only=False)

    config = ModelConfig(**checkpoint['config'].__dict__)
    model = Model(config)

    new_state_dict = {}
    for k, v in checkpoint['model'].items():
        if k.startswith('_orig_mod.'):
            new_state_dict[k[len('_orig_mod.'):]] = v
        else:
            new_state_dict[k] = v

    model.load_state_dict(new_state_dict)
    model.to(device)

    if use_compile:
        model = torch.compile(model)

    model.eval()

    return model
```

Рис. 4.5 Завантаження моделі на сервері

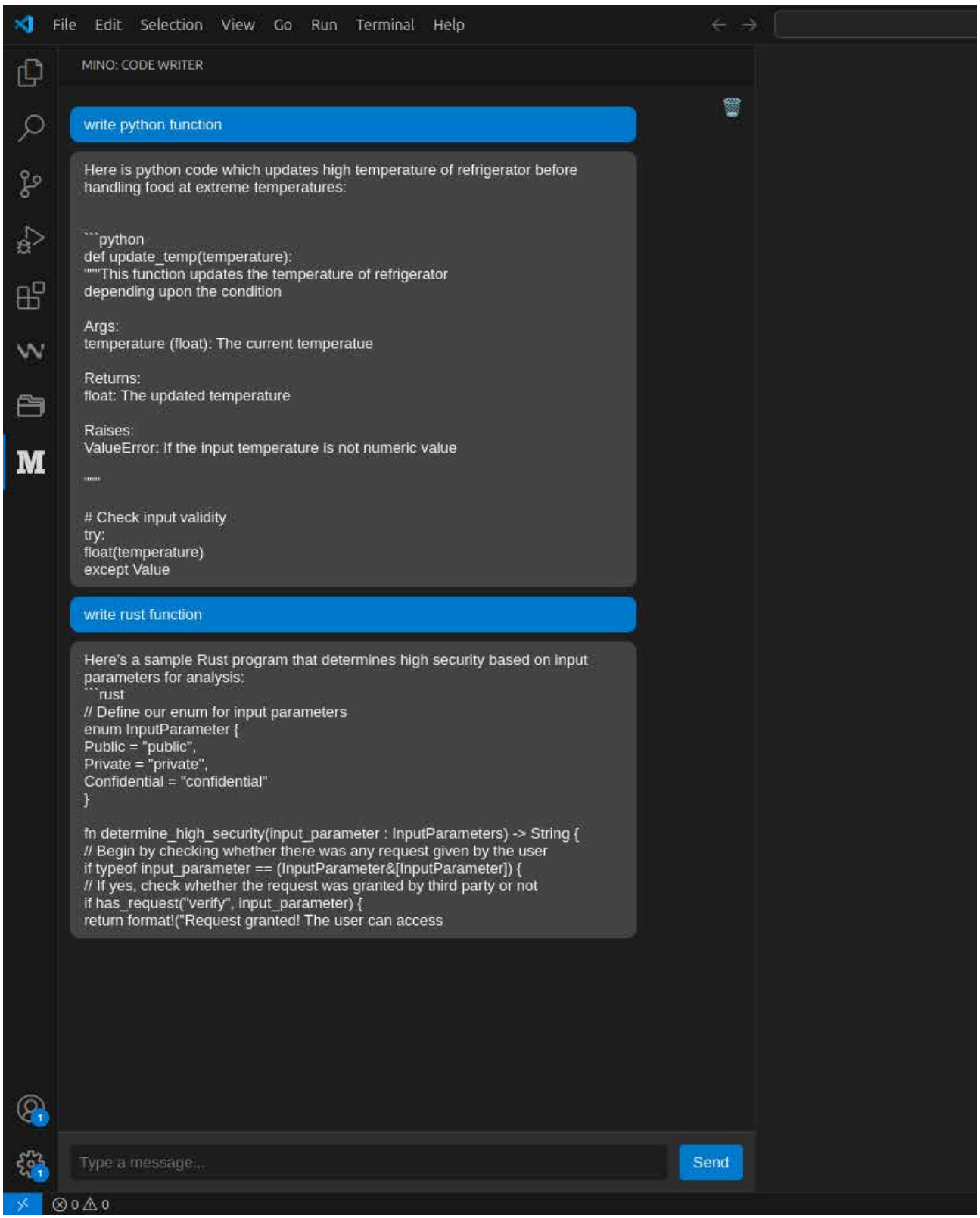

Рис. 4.6 Тестування простих запитів

4.2 Вимоги до апаратного та програмного забезпечення

Вимоги до системи (для власника системи)

На Рис. 4.6 відображено фізичну архітектуру розміщення компонентів системи, яка включає користувача, клієнтський пристрій і сервер, що поєднує модель ШІ, токенізатор та API.

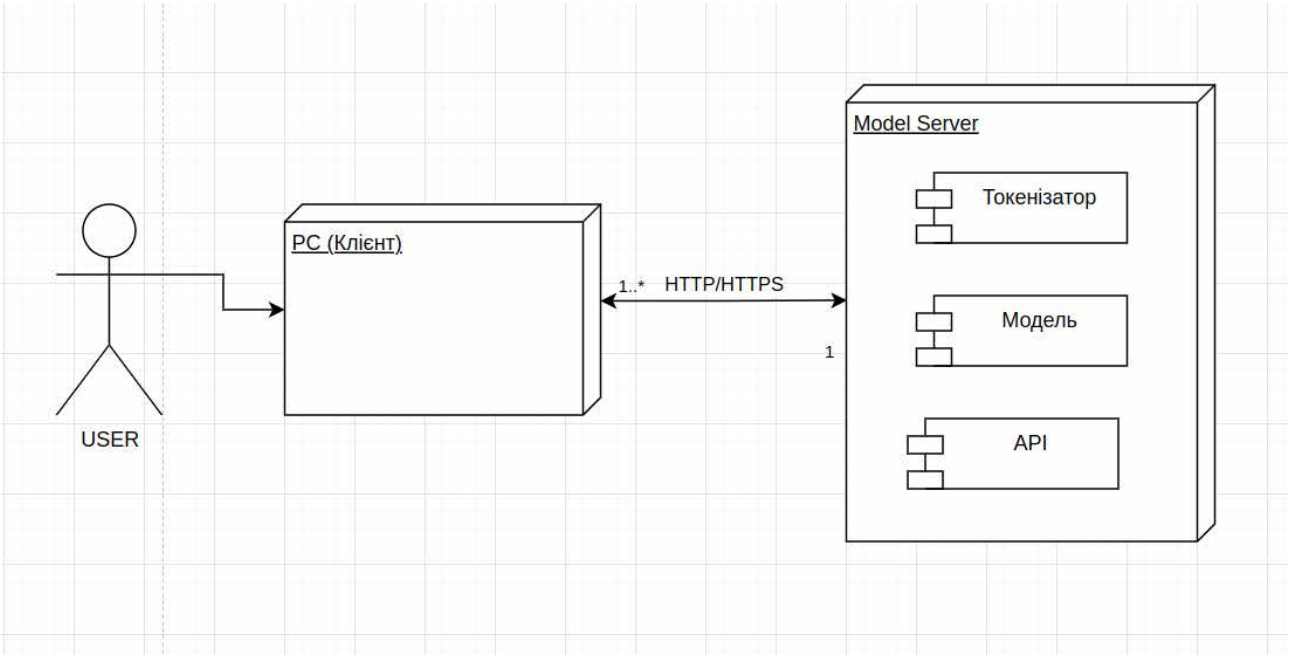

Рис. 4.6. Діаграма розгортання

Користувач (USER)

- **Опис:** Людина, яка взаємодіє з клієнтським застосунком.
- **Основні дії:** Надсилання запитів, отримання оброблених відповідей від моделі.

Клієнтський пристрій (PC користувача)

- **Роль:** Фізичний пристрій користувача, на якому виконується клієнтська частина системи.
- **Взаємодія з користувачем:** Клавіатура, миша, екран.

Сервер застосунку з моделлю ШІ та токенізатором

- **Роль:** Серверна частина, що приймає та обробляє запити від клієнта. Виконує інференс моделі ШІ, токенізацію вхідних даних, а також реалізує API для взаємодії з клієнтами
- **Взаємодія:** Отримує запити з клієнта, обробляє їх через токенізатор і модель, зберігає необхідні дані в базі, повертає результати клієнту.

Взаємодія між компонентами

- Протокол: HTTPS
- Принцип: Клієнтський пристрій надсилає запити до API-сервера. Сервер обробляє вхідні дані, виконує токенізацію, передає запит на модель для генерації відповіді та повертає оброблену відповідь клієнту.

Для стабільної та швидкої обробки запитів, власнику системи та розробнику необхідно мати:

Сервер з потужними GPU

- Для ефективного виконання інференсу моделі та обробки одночасних запитів
- Рекомендовані GPU: NVIDIA RTX 3080 або вище, або серверні GPU типу NVIDIA A100, Tesla V100
- Достатній обсяг оперативної пам’яті (32 ГБ і більше)
- Стабільне та швидке мережеве з’єднання для обробки великої кількості запитів

Програмні інструменти для розгортання та підтримки серверної частини наведено в наступній таблиці

Назва інструменту	**Версія (рекомендована або мінімальна)**	**Додаткові зауваження**
Node.js / Python	Остання LTS-версія	Для backend API
VS Code	Остання версія	Основне середовище розробки
PyTorch	В залежності від моделі	Для розробки та запуску моделі машинного навчання

Вимоги до системи з точки зору користувача (клієнта)

Апаратні вимоги

- Будь-який комп’ютер або ноутбук, що підтримує встановлення IDE
- Мінімальні вимоги залежать від IDE, зазвичай:
  - Процесор: Intel Core i3 або еквівалент

- Оперативна пам’ять: від 4 ГБ
- Вільне місце на диску: від 1 ГБ для IDE та плагінів

Програмні вимоги

- Встановлена інтегрована середа розробки Visual Studio Code
- Встановлений плагін для підключення до серверу
- Стабільне інтернет-з’єднання для комунікації з сервером (всі обчислення та обробка даних виконуються на сервері)
- Підтримка сучасних протоколів зв’язку (HTTP/HTTPS, WebSocket, або інший використовуваний протокол)

4.3 Склад інсталяційного пакету

У моєму випадку інсталяційний пакет має просту структуру і складається з одного основного компонента – плагіна для Visual Studio Code, який дозволяє користувачеві взаємодіяти з моделлю, що розгорнута на сервері (див. рис. 4.7).

Основні компоненти інсталяційного пакету:

VS Code Extension:

- Містить клієнтський код для взаємодії з серверною частиною.
- Включає інтерфейс користувача для надсилання запитів до моделі.
- Може бути встановлений через VS Code.

Файл package.json:

- Містить опис розширення, залежності, активатори команд і точки входу.
- Указує на основний виконуваний скрипт та API-залежності.

Залежності

- Всі необхідні залежності для роботи плагіна зазначені у файлі package.json. Вони автоматично встановлюються при інсталяції розширення.

- Інших локальних бібліотек чи середовищ не потребується, оскільки вся обробка даних відбувається на сервері.

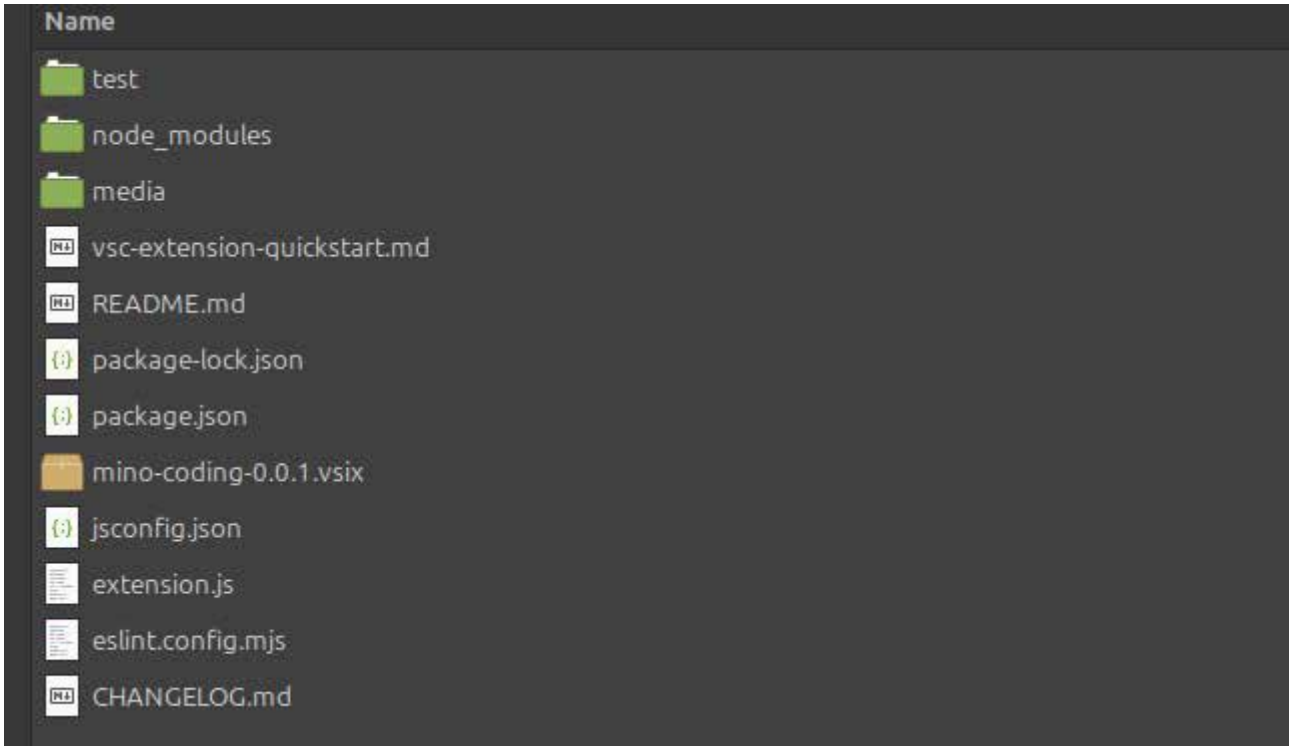

Рис. 4.7. Вміст каталогу плагіна

ВИСНОВКИ

У процесі виконання цієї дипломної роботи мені вдалося реалізувати повноцінну систему інтелектуального автодоповнення коду, яка базується на сучасній трансформерній архітектурі. Всі поставлені на початку завдання були успішно виконані.

По-перше, був проведений детальний аналіз існуючих рішень на основі великих мовних моделей, таких як GitHub Copilot та Codeium. Це дозволило краще зрозуміти підходи, які вже використовуються у промисловості, а також визначити напрями для вдосконалення власної моделі.

По-друге, я поглиблено вивчив архітектуру Transformer, її переваги над попередніми підходами (зокрема LSTM) і сучасні модифікації, які покращують ефективність генерації коду. Саме на основі цієї архітектури була створена модель.

Для основи своєї системи я використав відкриту модель DeepSeek Coder v3, яку було адаптовано та донавчено на тематичних датасетах із відкритих репозиторіїв GitHub. Завдяки цьому модель краще справляється із задачами доповнення реального коду, адаптується до різних мов програмування та розуміє контекст запиту.

Було реалізовано серверну частину (на Flask), яка відповідає за обробку запитів до моделі, та клієнтську – у вигляді плагіна для VS Code, написаного на JavaScript. Це забезпечує зручну взаємодію користувача з системою прямо під час написання коду.

Особливу увагу приділено оптимізації: застосовано змішану точність обчислень, навчальний план із поступовим зменшенням швидкості навчання, та механізми, як-от MoE (Mixture of Experts) і Latent Attention, які підвищують продуктивність та швидкість генерації.

Таким чином, мені вдалося реалізувати повноцінну систему, яка поєднує сучасні досягнення в машинному навчанні з практичними інструментами для розробників.

Загалом, ця дипломна робота стала для мене не лише технічним викликом, а й джерелом цінного досвіду: від глибокого розуміння архітектур трансформерів до створення зручного інтерфейсу для кінцевого користувача. Вважаю, що здобуті знання та навички стануть хорошою основою для подальшої професійної діяльності в IT-сфері.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Atlassian. (2023, March 10). How to use Codeium's AI coding assistant with Bitbucket – [Електронний ресурс] – Режим доступу: https://community.atlassian.com/forums/Bitbucket-articles/How-to-use-Codeium-s-AI-coding-assistant-with-Bitbucket/ba-p/2552360
2. The Transmitted. (n.d.). Що таке MLP у машинному навчанні – [Електронний ресурс] – Режим доступу: https://thetransmitted.com/adlucem/shho-take-mlp-u-mashynnomu-navchanni/
3. Saba, R. (2020, May 12). Long Short-Term Memory (LSTM). Medium – [Електронний ресурс] – Режим доступу: https://medium.com/@saba99/long-short-term-memory-lstm-fffc5eaebfdc
4. GPT Tokenizer. (2023). GPT Tokenizer: Tokenization for GPT Models – [Електронний ресурс] – Режим доступу: https://gpt-tokenizer.dev/
5. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *arXiv*. – [Електронний ресурс] – Режим доступу: https://arxiv.org/pdf/1706.03762
6. Школа штучного інтелекту. Л8. Механізм уваги і трансформери – [Відео] – Режим доступу: https://www.youtube.com/watch?v=yIYFv0zJByI
7. DeepSeek. (2024). *DeepSeek-V3*. GitHub. – [Електронний ресурс] – Режим доступу: https://github.com/deepseek-ai/DeepSeek-V3
8. Karpathy, A. (2023). *Let's reproduce GPT-2 (124M)* [Відео]. YouTube. – [Відео] – Режим доступу: https://www.youtube.com/watch?v=l8pRSuU81PU
9. Long Short-Term Memory – [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Long_short-term_memory
10. DeepSeek R1 Model Architecture – [Електронний ресурс] – Режим доступу: https://pub.towardsai.net/deepseek-r1-model-architecture-853fefac7050
11. Rotary Positional Embedding (RoPE) – [Електронний ресурс] – Режим доступу: https://blog.eleuther.ai/rotary-embeddings/

12. AI Coding Agents in 2025: Cursor vs. Windsurf vs. Copilot vs. Claude vs. VS Code AI – [Електронний ресурс] – Режим доступу: https://kingy.ai/blog/ai-coding-agents-in-2025-cursor-vs-windsurf-vs-copilot-vs-claude-vs-vs-code-ai/?utm_source=chatgpt.com
13. Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context – [Електронний ресурс] – Режим доступу: https://arxiv.org/pdf/2402.03630
14. IntelliCode Compose: Code Generation using Transformer – [Електронний ресурс] – Режим доступу: https://arxiv.org/pdf/2005.08025
15. Neural network (machine learning). Wikipedia. – [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Neural_network_%28machine_learning%29?utm_source=chatgpt.com
16. Why Transformers Are the Future: Limitations of LSTMs and How They’re Solved – [Електронний ресурс] – Режим доступу: https://www.dataairevolution.com/2024/10/why-transformers-are-the-future-limitations-of-lstms-and-how-theyre-solved/?utm_source=chatgpt.com
17. Training language models to follow instructions with human feedback – [Електронний ресурс] – Режим доступу: https://arxiv.org/pdf/1909.08593
18. ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING – [Електронний ресурс] – Режим доступу: https://arxiv.org/pdf/2104.09864
19. DeepSeek-V3 Technical Report – [Електронний ресурс] – Режим доступу: https://arxiv.org/html/2412.19437v1
20. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION – [Електронний ресурс] – Режим доступу: https://arxiv.org/pdf/1412.6980
21. Why is AdamW Often Superior to Adam with L2-Regularization in Practice? – [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/why-is-adamw-often-superior-to-adam-with-l2-regularization-in-practice/?utm_source=chatgpt.com

22. Transformer Architecture: The Positional Encoding – [Електронний ресурс] – Режим доступу: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/
23. Demystifying Transformers: Positional Encoding – [Електронний ресурс] – Режим доступу: https://medium.com/@weidagang/demystifying-transformers-positional-encoding-955dd018c76c
24. A Brief Overview of Cross Entropy Loss – [Електронний ресурс] – Режим доступу: https://medium.com/@chris.p.hughes10/a-brief-overview-of-cross-entropy-loss-523aa56b75d5

ДОДАТОК А

Фрагмент реалізації механізму Latent Attention
(адаптовано автором на основі коду з офіційного репозиторію DeepSeek, GitHub, 2024)
Джерело: https://github.com/deepseek-ai
Код було модифіковано автором з урахуванням потреб проєкту: збережено загальну архітектуру, але частково змінено імена змінних, форматування та структуру функцій для інтеграції у власну систему.

```
class MLA(nn.Module):
  def __init__(self, config: ModelConfig):
    super().__init__()
    self.dim = config.dim
    self.n_heads = config.n_heads
    self.q_lora_rank = config.q_lora_rank
    self.kv_lora_rank = config.kv_lora_rank
    self.qk_nope_head_dim = config.qk_nope_head_dim
    self.qk_rope_head_dim = config.qk_rope_head_dim

    self.qk_head_dim = config.qk_nope_head_dim + config.qk_rope_head_dim
    self.v_head_dim = config.v_head_dim

    if self.q_lora_rank == 0:
      self.wq = nn.Linear(self.dim, self.n_heads * self.qk_head_dim)
    else:
      self.wq_a = nn.Linear(self.dim, self.q_lora_rank)
      self.q_norm = RMSNorm(self.q_lora_rank)
      self.wq_b = nn.Linear(self.q_lora_rank, self.n_heads * self.qk_head_dim)

    # project input to a combined low-rank space for KV and RoPE key
```

```
    self.wkv_a = nn.Linear(self.dim, self.kv_lora_rank + self.qk_rope_head_dim)

    self.kv_norm = RMSNorm(self.kv_lora_rank)

    # projects kv lora to final key and value
    self.wkv_b = nn.Linear(self.kv_lora_rank, self.n_heads * (self.qk_nope_head_dim
+ self.v_head_dim))

    # final layer for output projection
    self.wo = nn.Linear(self.n_heads * self.v_head_dim, self.dim)

    self.softmax_scale = self.qk_head_dim ** -0.5

    if config.max_seq_len > config.original_seq_len:
      mscale = 0.1 * config.mscale * math.log(config.rope_factor) + 1.0
      self.softmax_scale = self.softmax_scale * mscale * mscale

    if attn_impl == "naive":
      self.register_buffer("k_cache", torch.zeros(config.max_batch_size,
config.max_seq_len, self.n_heads, self.qk_head_dim), persistent=False)
      self.register_buffer("v_cache", torch.zeros(config.max_batch_size,
config.max_seq_len, self.n_heads, self.v_head_dim), persistent=False)
    else:
      self.register_buffer("kv_cache", torch.zeros(config.max_batch_size,
config.max_seq_len, self.kv_lora_rank), persistent=False)
      self.register_buffer("p_cache", torch.zeros(config.max_batch_size,
config.max_seq_len, self.qk_rope_head_dim), persistent=False)

  def forward(self, x: torch.Tensor, start_pos: int, freqs: torch.Tensor, mask:
Optional[torch.Tensor]):
```

```
    batch_size, seq_len, _ = x.size()
    end_pos = start_pos + seq_len
    # project the input to get the query (q).
    if self.q_lora_rank == 0:
      q = self.wq(x)
    else:
      q = self.wq_b(self.q_norm(self.wq_a(x)))
    # reshape query tensor to get heades
    q = q.view(batch_size, seq_len, self.n_heads, self.qk_head_dim)
    # split query into non positional and positional
    q_nope, q_p = torch.split(q, [self.qk_nope_head_dim, self.qk_rope_head_dim],
dim=-1)
    # apply rope
    q_p = apply_rotary_emb(q_p, freqs)
    # get combined key and value
    kv = self.wkv_a(x)
    kv, k_p = torch.split(kv, [self.kv_lora_rank, self.qk_rope_head_dim], dim=-1)
    k_p = apply_rotary_emb(k_p.unsqueeze(2), freqs)
    if attn_impl == "naive":
      # concatenate non-pos and pos queries
      q = torch.cat([q_nope, q_p], dim=-1)
      # project KV to the final key and value dimensions
      kv = self.wkv_b(self.kv_norm(kv))
      kv = kv.view(batch_size, seq_len, self.n_heads, self.qk_nope_head_dim +
self.v_head_dim)
      k_nope, v = torch.split(kv, [self.qk_nope_head_dim, self.v_head_dim], dim=-1)
      # cat non-pos and pos keys
      k = torch.cat([k_nope, k_p.expand(-1, -1, self.n_heads, -1)], dim=-1)
      # store computed keys and values
      self.k_cache[:batch_size, start_pos:end_pos] = k
```

```
      self.v_cache[:batch_size, start_pos:end_pos] = v
      scores = torch.einsum("bshd,bthd->bsht", q, self.k_cache[:batch_size, :end_pos])
* self.softmax_scale
    else:
      wkv_b_weight = self.wkv_b.weight
      wkv_b = wkv_b_weight.view(self.n_heads, -1, self.kv_lora_rank)

      # project non pos query
      q_nope = torch.einsum("bshd,hdc->bshc", q_nope, wkv_b[:,
:self.qk_nope_head_dim])
      # store lora KV representation and positional embedding
      self.kv_cache[:batch_size, start_pos:end_pos] = self.kv_norm(kv)
      self.p_cache[:batch_size, start_pos:end_pos] = k_p.squeeze(2)

      scores = (torch.einsum("bshc,btc->bsht", q_nope, self.kv_cache[:batch_size,
:end_pos]) +
          torch.einsum("bshr,btr->bsht", q_p, self.p_cache[:batch_size, :end_pos])) *
self.softmax_scale
    if mask is not None:
      scores += mask.unsqueeze(1)
    # get attn probs
    scores = scores.softmax(dim=-1, dtype=torch.float32).type_as(x)
    # compute weighet sum
    if attn_impl == "naive":
      x = torch.einsum("bsht,bthd->bshd", scores, self.v_cache[:batch_size,
:end_pos])
    else:
      # 1. multiply attn with the cached lora KV.
      x = torch.einsum("bsht,btc->bshc", scores, self.kv_cache[:batch_size, :end_pos])
      # 2. project the result using the wkv_b weights.
```

```
    x = torch.einsum("bshc,hdc->bshd", x, wkv_b[:, -self.v_head_dim:])

x = self.wo(x.flatten(2))
return x
```