Міністерство освіти і науки України

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Обробка природних мов та великі мовні моделі

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня магістра
за освітніми програмами «Інтелектуальні сервіс-орієнтовані обчислювання»,
«Комп'ютерні науки» спеціальності F3 «Комп'ютерні науки»

Укладачі: Р. В. Кислий, І. О. Письменний

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2026

УДК 004.89:004.912(075.8)
О-24

Укладачі: *Кислий Роман Володимирович*, канд. техн. наук
Письменний Ігор Олександрович, канд. техн. наук

Рецензент *Шаповалова С. І.,* канд. техн. наук, доцент, доцент кафедри цифрових технологій в енергетиці, Національний технічний університет України «Київський політехнічний інститут ім. Ігоря Сікорського»

Відповідальний редактор *Безносик О. Ю.,* канд. техн. наук

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 7 від 08.05.2026 р.)
за поданням вченої ради Навчально-наукового інституту
прикладного системного аналізу
(протокол № 4 від 27 квітня 2026 р.)

О-24 Обробка природних мов та великі мовні моделі. [Електронний ресурс] : навч. посіб. для здобувачів ступеня магістра за освіт. програмами «Інтелектуальні сервіс-орієнтовані обчислювання», «Комп'ютерні науки» спец. F3 Комп'ютерні науки / КПІ ім. Ігоря Сікорського ; уклад.: Р. В. Кислий, І. О. Письменний. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2026. – 238 с.

Посібник містить матеріали лекцій з дисципліни «Обробка природних мов та великі мовні моделі», яка викладається здобувачам ступеня магістра галузі знань F «Інформаційні технології» спеціальності F3 «Комп’ютерні науки» освітніх програм «Інтелектуальні сервіс-орієнтовані обчислювання», «Комп’ютерні науки». Тематика лекцій присвячена методам обробки текстів та текстової інформації.

УДК 004.89:004.912(075.8)

Реєстр. № НП 25/26-338. Обсяг 7 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© КПІ ім. Ігоря Сікорського, 2026

ЗМІСТ

ЗМІСТ **3**

ПЕРЕДМОВА **5**

ВСТУП **7**

Історія та еволюція NLP 7

Чому NLP — одна з найвпливовіших галузей AI 8

РОЗДІЛ 1. КЛАСИЧНИЙ NLP ТА ТЕКСТОВІ ПРЕДСТАВЛЕННЯ **11**

ЛЕКЦІЯ 1: Вступ до NLP, формалізація задач та класичні підходи 11

ЛЕКЦІЯ 2: Токенізація та підслівне моделювання 26

ЛЕКЦІЯ 3: Нейронні ембедінги та дистрибутивна семантика 41

ЛЕКЦІЯ 4: Бенчмарки та метрики оцінювання в NLP 53

ЛЕКЦІЯ 5: Енкодерні та Seq2Seq моделі: T5, BART, переклад і сумаризація 68

ЛЕКЦІЯ 6: Декодерні LLM та ефективний висновок 84

РОЗДІЛ 2. ВЕЛИКІ МОВНІ МОДЕЛІ (LLM) **100**

ЛЕКЦІЯ 7: Вступ до навчання з підкріпленням для LLM та основи RLHF 100

ЛЕКЦІЯ 8: Вирівнювання LLM: RLHF, DPO, GRPO та оптимізація переваг 113

ЛЕКЦІЯ 9: Візійно-мовні моделі та мультимодальні LLM 127

ЛЕКЦІЯ 10: Векторний пошук та методи наближеного пошуку найближчих сусідів 140

РОЗДІЛ 3. ГЕНЕРАЦІЯ З ДОПОВНЕННЯМ ПОШУКОМ (RAG) **154**

ЛЕКЦІЯ 11: Розріджений та щільний пошук. Дотренування ембедінг-моделей 154

ЛЕКЦІЯ 12: End-to-End RAG-системи та шаблони проєктування 164

ЛЕКЦІЯ 13: MLOps для LLM: деплой, квантизація та дистиляція 181

ЛЕКЦІЯ 14: Reasoning LLM та Chain-of-Thought...................................... 196

ЛЕКЦІЯ 15: Агентний штучний інтелект та системи з інструментами ... 208

ЗАГАЛЬНІ РЕКОМЕНДАЦІЇ 231

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ 235

ПЕРЕДМОВА

Обробка природної мови (Natural Language Processing, NLP) стала однією з найбільш динамічно розвиваючихся та впливових галузей штучного інтелекту (Artificial Intelligence, AI). За останні десять років ми стали свідками революційної трансформації у цій галузі — від статистичних методів, які домінували в 2010-х роках, до потужних великих мовних моделей (Large Language Models, LLMs), які демонструють видатні здібності в розумінні та генеруванні природної мови.

Сьогодні мовні моделі знаходяться у центрі уваги як наукової спільноти, так і промисловості. Вони дозволяють розв'язувати складні задачі обробки тексту, від анотування сутностей та класифікації документів до побудови діалогових систем та автоматичного синтезу тексту. NLP став критичною компетенцією для фахівців у галузі інженерії даних, машинного навчання та дослідження штучного інтелекту.

Цей навчальний посібник розроблений для студентів програми "Штучний інтелект" спеціальності "Комп'ютерні науки" Київського Політехнічного Інституту ім. Ігоря Сікорського. Курс має обсяг 4 кредити ECTS та трудомісткість 120 годин аудиторних та самостійних занять. Посібник структурований таким чином, щоб забезпечити як теоретичне розуміння фундаментальних концепцій NLP, так і практичні навички розробки реальних систем обробки тексту.

Матеріал охоплює чотири основні розділи: класичні методи NLP та обробка тексту (розділ 1), великі мовні моделі та трансформери (розділ 2), системи пошуку та генерування з аугментацією інформацією (Retrieval-Augmented Generation, RAG) (розділ 3) та просунуті методи розробки агентних систем та практичні застосування (розділ 4). Кожен розділ включає лекції, лабораторні роботи з практичною реалізацією та завдання для оцінювання.

Посібник написаний українською мовою з використанням англійської

термінології для методів і прямих назв концепцій у дужках. Такий підхід забезпечує локалізацію контенту для українських студентів при збереженні наукової точності та міжнародної сумісності з англомовною літературою та документацією.

ВСТУП

Історія та еволюція NLP

Обробка природної мови (Natural Language Processing, NLP) як дисципліна має коріння, що сягають 1950-х років. У той час дослідники намагалися розв'язати задачу машинного перекладу, вважаючи її порівняно простою проблемою підстановки слів одної мови на слова іншої. Однак вже до кінця 1960-х років стало зрозуміло, що розуміння природної мови вимагає набагато глибшого розуміння контексту, синтаксису, семантики та світового знання.

Протягом 1970-х та 1980-х років панували так звані "символічні" або "правилові" підходи. Дослідники розробляли складні граматичні правила, семантичні мережі та системи експертних знань (Expert Systems). Незважаючи на певні успіхи в спеціалізованих доменах, ці методи виявилися недостатньо масштабованими та жорсткими для обробки різноманітності природної мови.

З появою комп'ютерів більшої потужності та розвитком статистичного машинного навчання (Statistical Machine Learning) у 1990-х роках почався період "емпіричної революції" в NLP. Методи, засновані на розпізнаванні схем у великих текстових корпусах, стали домінуючим підходом. Цей період дав нам алгоритми для токенізації, морфологічного аналізу, розпізнавання іменованих сутностей (Named Entity Recognition, NER) та парсингу синтаксису.

Період 2010-2015 років характеризувався переходом до глибокого навчання (Deep Learning). Рекурентні нейронні мережі (Recurrent Neural Networks, RNN), включаючи довгострокову пам'ять (Long Short-Term Memory, LSTM) та механізми уваги (Attention Mechanisms), дозволили моделям краще захоплювати послідовні залежності в тексті. Словні вкладення (Word Embeddings) як Word2Vec та GloVe надали моделям досяжного способу представляти слова у вигляді щільних векторів (Dense Vectors).

Революцію почав у 2017 році прорив з архітектури трансформера (Transformer), представленої у роботі “Attention Is All You Need”. Трансформери замінили рекурентні механізми на чистий механізм уваги, що дозволило паралельно обробляти послідовності й масштабуватися на величезні набори даних. Це призвело до розвитку попередньо навчених мовних моделей (Pre-trained Language Models): BERT (2018), GPT (2018 та подальші версії), RoBERTa, ELECTRA та багато інших.

Сьогодні ми перебуваємо в епосі великих мовних моделей (Large Language Models, LLMs), які навчені на текстах трильйонів слів. Моделі як GPT-4, Claude 3, Llama 3 та інші демонструють видатні здібності в розумінні контексту, логічному висновку, творчих завданнях та навіть програмуванні. Цей прогрес був можливий завдяки комбінації трьох факторів: (1) архітектурним інноваціям, (2) наявності масивних наборів даних та (3) обчислювальним ресурсам для тренування.

Чому NLP — одна з найвпливовіших галузей AI

Мова є унікальним для людства феноменом. Вона є основним засобом, через який люди обмінюються знаннями, вирішують проблеми та будують культуру. Здатність машин розуміти та генерувати природну мову має мільярди потенційних застосувань.

По-перше, NLP знаходить застосування у комерції та індустрії. Автоматичні системи обробки клієнтських запитів, аналіз почуттів (Sentiment Analysis) для контролю якості, класифікація документів для управління знаннями (Knowledge Management), автоматичне анотування медичних записів — все це економить мільйони людино-годин роботи. Системи діалогу зі штучним інтелектом (Conversational AI) та чат-боти (Chatbots) гарантують цілодобову підтримку клієнтів без людського втручання.

По-друге, NLP має глибокий суспільний вплив. Автоматичний переклад порушує мовні бар’єри, дозволяючи людям з усього світу спілкуватися.

Системи питання-відповіді (Question Answering, QA) демократизують доступ до інформації. Моделі для автоматичного створення контенту можуть допомогти людям з обмеженнями створювати письмові матеріали.

По-третє, розв'язання задач NLP потребує розуміння складних явищ у штучному інтелекті — представлення знання (Knowledge Representation), логічного висновку (Reasoning), контекстного розуміння та адаптації до нових доменів (Domain Adaptation). Успіхи в NLP часто приносять нові інсайти, які можуть бути застосовані до інших галузей AI.

Структура курсу та модулі

Цей курс охоплює чотири основні модулі:

Розділ 1: Класичні методи NLP та базові техніки текстової обробки — Розглядаємо фундаментальні концепції: токенізацію, представлення тексту (Bag-of-Words, TF-IDF, N-grами), класичні алгоритми машинного навчання для класифікації та розпізнавання структури. Цей розділ будує міцний фундамент розуміння.

Розділ 2: Трансформери та великі мовні моделі — Вивчаємо архітектуру трансформера, механізм уваги, попередньо навчені моделі (Pre-training), тонке налаштування (Fine-tuning) та інтерпретацію нейронних мовних моделей. Розглядаємо практичні аспекти: BERT, GPT, та інші державні моделі.

Розділ 3: Пошук та аугментація інформацією (RAG) — Розглядаємо системи, які поєднують мовні моделі з системами пошуку та аргументовані генерування. Вивчаємо семантичний пошук (Semantic Search), векторні бази даних (Vector Databases), та практичні застосування для збільшення точності та свіжості інформації моделей.

Розділ 4: Просунуті методи та агентні системи — Вивчаємо техніки для розробки складних систем: функціональна кличка (Function Calling), планування та помилкова обробка (Error Handling), розробка багатокрокових

агентів (Multi-step Agents), та практичні найкращі практики для продукційної розробки.

Передумови

Для успішного засвоєння матеріалу курсу потрібні наступні базові знання: **лінійна алгебра** (матриці, вектори, скалярні добутки), **теорія ймовірностей та статистика** (розподіли, умовна ймовірність, залежні змінні), **машинне навчання** (класифікація, регресія, крос-валідація, метрики оцінювання), **глибоке навчання** (нейронні мережі, зворотне поширення помилки, навчання), **програмування на Python** та практичне використання бібліотек NumPy та Pandas, **знайомість з PyTorch** та/або TensorFlow, а також **базове розуміння бібліотеки HuggingFace Transformers**.

Очікувані результати навчання

По завершенню курсу студенти зможуть:

1. Розуміти теоретичні основи NLP, включаючи представлення тексту та класичні алгоритми
2. Розв’язувати практичні задачі класифікації тексту, розпізнавання сутностей та класифікації послідовностей
3. Працювати з попередньо навченими мовними моделями, включаючи завдання тонкого налаштування
4. Розробляти системи на основі RAG для покращення генерування та пошуку інформації
5. Реалізовувати практичні агентні системи з функціональною кличкою та планування
6. Критично оцінювати та вибирати відповідні моделі та архітектури для конкретних задач
7. Розуміти обмеження та етичні аспекти сучасних мовних моделей

РОЗДІЛ 1. КЛАСИЧНИЙ NLP ТА ТЕКСТОВІ ПРЕДСТАВЛЕННЯ

Посібник організований послідовно, кожна лекція будується на матеріалі попередніх. Рекомендується читати матеріали лекцій у запропонованому порядку. Кожна лекція включає основний контент з теоретичними поясненнями та практичними прикладами, контрольні запитання для самоперевірки розуміння, рекомендовану літературу для глибшого вивчення, а також посилання на практичні завдання та лабораторні роботи.

Рекомендується паралельно з вивченням лекційного матеріалу виконувати практичні завдання та лабораторні роботи, які закріплюють теоретичні знання через безпосередню реалізацію на коді.

ЛЕКЦІЯ 1: Вступ до NLP, формалізація задач та класичні підходи

1. Що таке NLP як галузь науки, зв'язок з лінгвістикою та AI

Обробка природної мови (Natural Language Processing, NLP) є міждисциплінарною областю, що лежить на перетині комп'ютерних наук, штучного інтелекту, лінгвістики та когнітивної психології. На найбільш абстрактному рівні, NLP займається розробкою методів та алгоритмів, які дозволяють комп'ютерам розуміти, інтерпретувати та генерувати природну мову таким способом, який є змістовним та корисним для людей.

Зв'язок з лінгвістикою

Хоча NLP часто розглядається як інженерна дисципліна, воно має глибокий теоретичний зв'язок з лінгвістикою. Традиційна лінгвістика вивчає структуру мови — звуки (фонетика), слова та їхні морфологічні властивості, речення та їхні синтаксичні відносини, значення (семантика) та як значення змінюється у залежності від контексту. Ці лінгвістичні поняття прямо впливають на дизайн NLP систем.

Наприклад, розуміння морфології (процес утворення слів шляхом

додавання афіксів) критично важливе для обробки мов з багатою морфологією, як українська. Синтаксичний аналіз (Syntactic Parsing) базується на теоріях синтаксичної структури, розроблених у лінгвістиці. Семантичний аналіз потребує розуміння того, як слова та фрази комбінуються для передачі сенсу.

Однак, важливо зазначити, що сучасний NLP, особливо в епосі глибокого навчання, менш залежний від явного лінгвістичного знання. Замість цього, нейронні моделі навчаються розпізнавати та використовувати структурні закономірності безпосередньо з даних. Це привело до того, що глибокі моделі часто можуть досягти або перевищити якість систем, побудованих на явних лінгвістичних правилах, на умові наявності достатньої кількості даних для навчання.

Зв’язок зі штучним інтелектом

NLP є однією з семи основних сфер AI. Успіх у розв’язанні задач NLP потребує розв’язання багатьох фундаментальних проблем AI, включаючи **представлення знання** — як ми представляємо значення слів, фраз та текстів у формі, з якою може працювати комп’ютер, **логічний висновок** — як ми можемо виводити нові факти та взаємозв’язки з наявного знання, **обробка невизначеності**, оскільки мова за своєю природою неоднозначна та один рядок може мати декілька інтерпретацій, **адаптивність** — як системи можуть адаптуватися до нових доменів, мов та контекстів, та **комбінація з іншими модальностями**, оскільки часто розуміння потребує інтеграції інформації з тексту, зображень та інших джерел.

Прогрес в NLP часто призводить до прогресу й у інших областях AI, і навпаки.

2. Таксономія ключових NLP задач

NLP задачі можна категоризувати за різними критеріями. Ось основні типи задач, з якими працює НЛП:

Класифікація тексту (Text Classification)

Це задача призначення тексту до однієї або декількох перевизначених категорій. До прикладів належать класифікація настрою (Sentiment Classification), де потрібно визначити, чи позитивний, негативний або нейтральний текст, фільтрація спаму (Spam Detection), де необхідно визначити, чи є лист спамом або легітимним, детекція мови (Language Detection) для визначення, яка мова використовується в тексті, та класифікація теми документу (Topic Classification), коли потрібно визначити тему документу (спорт, політика, технологія тощо).

Математично, класифікація — це задача навчання функції $f: X \rightarrow Y$, де X є простором текстових документів, а Y є скінченною множиною класів. Виходимо з набору тренувальних прикладів $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ і намагаємося узагальнити до невидимих прикладів.

Розпізнавання іменованих сутностей (Named Entity Recognition, NER)

Це задача ідентифікації та класифікації іменованих сутностей у тексті. Сутностями можуть бути особи, організації, місця, дати, грошові суми тощо. Приклад:

Текст: "Ілля Мачін працює у Київський Національній Технічній Університеті"

Очікуваний вихід включає визначення "Іллі Мачіна" як PERSON та "Київського Національного Технічного Університету" як ORGANIZATION.

NER є послідовною задачею позначення (Sequence Labeling Task) — кожному токену (слову) присвоюється мітка.

Позначення послідовностей (Sequence Labeling)

Більш загальна категорія, де кожен елемент у послідовності отримує мітку з фіксованого набору. Прикладами є позначення частини мови (Part-of-Speech Tagging, POS Tagging), де потрібно позначити кожне слово його

граматичною категорією (іменник, дієслово, прикметник тощо), морфологічна сегментація (Morphological Segmentation), яка включає розбиття слова на морфеми, та семантичне ролювання (Semantic Role Labeling, SRL), де здійснюється ідентифікація аргументів дієслова та їхніх семантичних ролей.

Машинний переклад (Machine Translation, MT)

Задача перекладу тексту з однієї мови (вихідна мова, Source Language) на іншу (цільова мова, Target Language), наприклад перекладу з української на англійську або навпаки.

Це є задачею послідовність-у-послідовність (Sequence-to-Sequence, Seq2Seq), де вхід та вихід мають потенційно різні довжини та залежать один від одного складним способом.

Автоматичне резюмування (Automatic Summarization)

Задача утворення стислого резюме з однієї або декількох вхідних документів, яке передає найважливішу інформацію. Вона може бути реалізована як екстрактивне резюмування (Extractive Summarization), що передбачає вибір найважливіших речень або фрагментів з вихідного тексту, або як абстрактивне резюмування (Abstractive Summarization), яке включає генерування нових речень, що синтезують інформацію.

Відповіді на запитання (Question Answering, QA)

Задача знайти відповідь на задане запитання на основі вхідного контексту або знання. До прикладів належать читання розуміння (Reading Comprehension), де потрібно знайти відповідь у наданому контексті за запитанням, та питання-відповіді у відкритому доменні (Open-domain QA), де необхідно відповісти на запитання користувача, використовуючи весь інтернет або велику базу знання.

Генерування тексту (Text Generation)

Задача утворення нового тексту на основі певної умови або контексту. Прикладами є діалог (Dialogue), коли необхідно тримати розмову з

користувачем, генерування на основі промпту (Prompt-based Generation), де потрібно завершити речення або параграф за наданим вступом, та умовне генерування (Conditional Generation), яке передбачає генерування тексту, що задовольняє певні умови.

3. Формалізація бізнес-задач як NLP задач

На практиці, коли у компанії виникає потреба “щось зробити з текстом”, часто вона не є явно сформульована як стандартна NLP задача. Роль NLP інженера полягає у трансляції бізнес-вимоги до формалізованої машинної задачі навчання (Learning Task).

Приклад 1: Система обробки скарг клієнтів

Бізнес-потреба: “Нам потрібно автоматично опрацьовувати скарги клієнтів та маршрутизувати їх до відповідних департаментів.”

Аналіз та формалізація передбачає обробку текстової скарги від клієнта, класифікацію скарги на категорії (технічна проблема, проблема з біллінгом, якість сервісу, інше) та видобування номера замовлення та імені продукту з скарги (NER). На виході маємо клас категорії, ймовірності для кожної категорії та видобуті сутності.

Технічне рішення включає збір набору даних з історичних скарг та їхніх вручну призначених категорій, навчання моделі класифікації тексту (наприклад, Logistic Regression з TF-IDF або нейронна модель), виконання контрольного тестування на виділеному тестовому наборі та розгортання моделі у продакшен з моніторингом якості.

Приклад 2: Система пошуку релевантних документів

Бізнес-потреба: “Юристам потрібно знаходити релевантні прецеденти для поточних справ серед базою даних попередніх судових рішень.”

Аналіз та формалізація включає обробку описання поточної справи (запиту) з завданням ранжування всіх попередніх рішень за релевантністю до

поточної справи. На виході отримуємо впорядкований список рішень з оцінками релевантності.

Технічне рішення передбачає представлення опису справи та попередніх рішень числовими векторами (наприклад, TF-IDF) та обчислення подібності (Similarity) між запитом та кожним документом за допомогою косинусної подібності:

$$\mathrm{sim}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{||\mathbf{a}|| \cdot ||\mathbf{b}||}.$$

Завершуючи процес, ми ранжуємо документи та повертаємо топ-K результатів.

Це є задачею інформаційного пошуку (Information Retrieval, IR), хоча часто розглядається як частина більш загального NLP.

Приклад 3: Автоматичне позначення цитат у наукових статтях

Бізнес-потреба: “Дослідницькій платформі потрібно автоматично виявляти цитати та цільові роботи у рукописах статей.”

Аналіз та формалізація передбачає обробку тексту наукової статті з завданням визначення для кожного токена, чи є він частиною цитати чи ні. На виході маємо послідовність міток (цитата/не-цитата) для кожного токена.

Технічне рішення включає розпізнання цієї як задачі послідовного позначення (Sequence Labeling), де можна навчити модель типу Conditional Random Field (CRF) або використати нейронну модель (RNN/LSTM), якої дозволяє отримати залежні від контексту переходи (Context-dependent transitions).

4. Представлення тексту: модель “Мішок слів” (Bag-of-Words)

Першим кроком у машинному навчанні для NLP є перетворення тексту у представлення, з яким можуть працювати алгоритми машинного навчання. Текст — це послідовність символів, яка є дискретною та неструктурованою.

Машинне навчання, з іншого боку, зазвичай працює з числовими векторами у неперервних просторах.

Модель "Мішок слів" (Bag-of-Words, BoW)

Найпростіша та найпопулярніша модель представлення — це модель "мішок слів". Концепція:

1. Взяти всі унікальні слова у наборі текстів (словник, Vocabulary)
2. Для кожного документа утворити вектор, у якому i-та компонента вказує на кількість разів, коли i-те слово з'являється у документі

Приклад:

Словарь: {"кіт", "собака", "бігти", "сидіти"}

Документ 1: "Кіт сидить. Собака біжить." Вектор BoW: [1, 1, 1, 1].

Документ 2: "Кіт сидить. Кіт сидить." Вектор BoW: [2, 0, 0, 2].

$$\mathbf{x} = [c(w_1), c(w_2), \ldots, c(w_V)].$$

Властивості BoW

Переваги цього методу включають простоту реалізації та розрахунку, збереження інформації про частоту слів та часту ефективність для простих задач класифікації.

Недоліки: Повна втрата порядку слів — документи "собака кусає чоловіка" та "чоловік кусає собаку" матимуть однаковий вектор BoW, Повна втрата синтаксичної та семантичної структури, Висока розмірність вектора (розмір словника може бути сотнями тисяч слів) та Розрідженість вектора — більшість компонент будуть нулями.

5. N-грами та їхня роль

Щоб частково вирішити проблему BoW з втратою порядку, використовуються N-грами.

Визначення N-грами

N-грама — це послідовність з N послідовних токенів (зазвичай слів). До прикладів належать уніграми (Unigrams, N=1) як “кіт”, “сидить”, “собака”, біграми (Bigrams, N=2) як “кіт сидить”, “собака біжить” та триграми (Trigrams, N=3) як “кіт сидить собака”, “собака біжить швидко”

Використання N-грам

Замість просто підрахувати слова, ми можемо підрахувати N-грами. Це зберігає деяку інформацію про порядок та комбінацію слів.

Приклад з біграмами:

Словник біграм: {“кіт сидить”, “собака біжить”, “біжить швидко”, “сидить собака”}

Документ: “Кіт сидить. Собака біжить швидко. Кіт сидить.” Вектор N-грам: [2, 1, 1, 0]

Властивості N-грам:

До переваг N-грам належать певна фіксація послідовності та контексту слів та часто кращі результати на практиці в порівнянні з просто уніграмами.

Недоліки N-грам включають експоненціальне зростання розміру словника зі збільшенням N, спарсеність проблеми, яка стає ще гіршою з більшістю N-грам, що мають нульову частоту, та отримання надто специфічних послідовностей при $N > 3$, які рідко повторюються у новому тексті.

На практиці часто використовують комбінацію уніграм та біграм (або одного з них), але вищі N-грами дають меншу користь.

6. TF-IDF: формула, інтуїція та властивості

TF-IDF (Term Frequency-Inverse Document Frequency) — це схема зважування термінів, яка прагне придати більше значення рідкісним та інформативним словам при одночасному зниженні ваги частих слів, які несуть мало інформації.

Формула

Вага терміна t у документі d визначається як:

TF-IDF(t, d) = TF(t, d) × IDF(t),

де:

TF(t, d) = (кількість разів, коли t з’являється в d) / (загальна кількість слів у d),

IDF(t) = log(N / (1 + число документів, що містять t)).

$$\text{TF-IDF}(t, d, D) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \times \log \frac{|D|}{|\{d \in D: t \in d\}|},$$

де N — загальна кількість документів у корпусі.

Варіанти формули:

Деякі бібліотеки використовують трохи різні формули:

TF без нормалізації: просто кількість разів t у d, IDF з різними логарифмічними базами: log10, log2, натуральний логарифм та Додавання 1 у знаменнику IDF для запобігання діленню на нуль (як показано вище).

Інтуїція

TF (Term Frequency) — чим більше разів слово з’являється у документі, тим більше воно може бути релевантним для цього документа та IDF (Inverse Document Frequency) — чим більше документів містять слово, тим менше специфічним воно є і тим менш інформативним. Слова як “the” (англ.) чи “та” (укр.) з’являються у майже всіх документах, тому їхнє IDF низьке. Рідкі слова, що з’являються лише у декількох документах, мають високе IDF.

Поєднання TF та IDF дає нам схему, яка покращує частоті слова у документі, але штрафує загальні слова.

Приклад обчислення TF-IDF

Корпус із трьох документів:

Документ 1: “кіт сидить на килимку” Документ 2: “собака біжить у сад” Документ 3: “кіт та собака живуть разом”

Розглянемо слово “кіт”:

TF(“кіт”, Документ 1) = 1/5 = 0.2 TF(“кіт”, Документ 2) = 0/4 = 0 TF(“кіт”, Документ 3) = 1/6 ≈ 0.167

IDF(“кіт”) = log(3 / (1 + 2)) = log(3/3) = log(1) = 0

Хм, це вказує на проблему з формулою. Давайте використаємо альтернативну формулу:

IDF(“кіт”) = log(3 / 2) ≈ 0.176

TF-IDF(“кіт”, Документ 1) = 0.2 × 0.176 ≈ 0.0352

Аналогічно можемо обчислити для усіх слів та всіх документів.

Представлення документа у виді вектора TF-IDF

Після обчислення TF-IDF для всіх термінів у словнику, кожен документ представляється як вектор, у якому компонента i — це вага TF-IDF терміну i.

Приклад:

Словник (в порядку): [“та”, “біжить”, “живуть”, “килимку”, “кіт”, “на”, “собака”, “сад”, “сидить”, “у”, “разом”]

Документ 1: [0, 0, 0, 0.1, 0.04, 0.08, 0, 0, 0.12, 0, 0] …

Цей вектор має розмірність, рівний розміру словника.

Властивості TF-IDF

Переваги включають простоту обчислення, ефективність на практиці для багатьох задач, інтуїтивну зрозумілість та меншу обчислювальну складність у порівнянні з більш складними методами.

Недоліки включають те, що це статистична модель, яка не враховує семантику слів, не враховує порядок слів (окрім як використовуючи N-грами), розрідженість, коли вектори залишаються розрідженими у більшості випадків,

та повільність для дуже великих словарів та корпусів.

7. Побудова класичного конвеєра: TF-IDF + Logistic Regression

Типовий машинно-навчальний конвеєр для класифікації тексту з використанням класичних методів складається з таких кроків:

Крок 1: Підготовка та попередня обробка тексту

Обробка починається з токенізації (розбиття тексту на слова/токени) та нормалізації в нижній регістр (Lowercasing), коли всі букви перетворюються у нижній регістр для стандартизації, Видалення пунктуації та спеціальних символів, Видалення стоп-слів (Stop Words Removal) — слова, які мало сприяють змісту (та, і, в, на тощо) та Стемінг або лематизація (Stemming/Lemmatization) — приведення слів до кореня або канонічної форми.

Крок 2: Побудова словника

Проходимо через всі тренувальні документи, Збираємо всі унікальні токени та Опціонально: видаляємо дуже частіші та дуже рідкісні слова (їх інформативність низька).

Крок 3: Векторизація з TF-IDF

Для кожного документа обчислюємо TF-IDF вагу для кожного токена та Отримуємо матрицю (документи × терміни), де кожна клітинка — це вага TF-IDF.

Крок 4: Навчання класифікатора

Мішок слів + TF-IDF представлення часто комбінуються з простим лінійним класифікатором, як Logistic Regression. Logistic Regression вивчає лінійний роздільник (Linear Separator) у просторі ознак (Feature Space).

Модель:

$P(y=1|x) = \sigma(w \cdot x + b)$,

де σ — сигмоїдна функція, w — вектор ваг, b — зміщення (bias), x —

вектор ознак (TF-IDF вектор):

$$P(y=1|\mathbf{x})=\sigma(\mathbf{w}^T\mathbf{x}+b)=\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}.$$

Крок 5: Оцінювання та гіперпараметри

Крос-валідація (Cross-validation) для оптимізації гіперпараметрів, Вибір порога класифікації (Decision Threshold) та Оцінювання на тестовому наборі.

Практичний приклад коду (псевдокод):

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
# Крок 1-3: Векторизація та побудова конвеєра
pipeline = Pipeline([
    ('tfidf', TfidfVectorizer(max_features=5000, ngram_range=(1,2))),
    ('clf', LogisticRegression(max_iter=1000, C=1.0))
])
# Крок 4: Навчання
pipeline.fit(X_train, y_train)
# Крок 5: Оцінювання
accuracy = pipeline.score(X_test, y_test)
```

8. Основи оцінювання класифікації: точність, точність, повнота, F1

Після навчання моделі, нам потрібно оцінити, як добре вона виконує завдання. Для класифікації використовуються кілька метрик.

Матриця плутанини (Confusion Matrix)

Для бінарної класифікації матриця плутанини — це таблиця 2×2:

	Передбачено 1	Передбачено 0
Насправді 1	TP (True Positive)	FN (False Negative)
Насправді 0	FP (False Positive)	TN (True Negative)

Точність (Accuracy)

Точність — це частка правильно класифікованих прикладів:

Accuracy = (TP + TN) / (TP + TN + FP + FN).

Це найпростіша метрика, але вона може бути оманливою, якщо класи незбалансовані (дисбаланс класів, Class Imbalance).

Точність (Precision)

Точність — це частка позитивних передбачень, які були правильними:

Precision = TP / (TP + FP).

Інтерпретація: "З усіх випадків, коли модель передбачила позитивний клас, скільки вона насправді передбачила правильно?"

Використовується, коли помилки типу FP дорогі (наприклад, у виявленні спама, FP означає, що легітимна пошта блокується).

Повнота (Recall)

Повнота — це частка позитивних прикладів, які модель правильно визначила:

Recall = TP / (TP + FN).

Інтерпретація: "З усіх справжніх позитивних випадків, скільки модель знайшла?"

Використовується, коли помилки типу FN дорогі (наприклад, у виявленні хвороб, FN означає, що хвора людина не виявлена).

Гармонійна середина (F1-Score)

F1 — це гармонійна середина Precision та Recall:

F1 = 2 × (Precision × Recall) / (Precision + Recall),

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}, \quad F_1 = \frac{2 \cdot P \cdot R}{P + R}.$$

F1 корисна, коли ми хочемо збалансувати Precision та Recall. Вона дає

більш об’єктивну оцінку, ніж Accuracy, особливо при дисбалансі класів.

Приклад обчислення

Припустимо, маємо 100 прикладів: TP = 40, FP = 10, FN = 30 та TN = 20.

Accuracy = (40 + 20) / 100 = 0.6 Precision = 40 / (40 + 10) = 40 / 50 = 0.8 Recall = 40 / (40 + 30) = 40 / 70 ≈ 0.571 F1 = 2 × (0.8 × 0.571) / (0.8 + 0.571) ≈ 0.667

Багатокласова класифікація

Для задач з більш ніж двома класами ми можемо розширити метрики:

Macro-F1 (незважена середина F1 для всіх класів), Micro-F1 (розраховується на основі глобального TP, FP, FN, TN) та Weighted F1 (зважена за кількістю прикладів у кожному класі).

9. Переваги та обмеження класичних підходів

Переваги класичних методів (BoW, TF-IDF, лінійні класифікатори)

Швидкість навчання та інференції — прості моделі навчаються дуже швидко, Інтерпретованість — ми можемо легко зрозуміти, які слова найбільше впливають на рішення, Низькі вимоги до обчислювальних ресурсів — можуть працювати на звичайних комп’ютерах, Малі об’єми даних — часто добре працюють навіть з невеликими наборами даних та Стабільність — менш схильні до перетренування (Overfitting).

Обмеження класичних методів

Втрата семантики — TF-IDF не розуміє, що слова “хороший” та “чудовий” мають подібне значення, Втрата синтаксичної структури — порядок слів ігнорується (окрім як через N-грами), Проблема розмірності — словник стає дуже великим для великих корпусів, Спарсеність — вектори містять переважно нулі, що не ефективно, Нездатність до складного логічного висновку — наприклад, заперечення: “це не добре” матиме позитивні компоненти для слова “добре” та Нездатність до контекстної залежності —

одне й те ж слово у різних контекстах матиме одну й ту ж представлення.

Перехід від класичних методів до нейронних мереж та подальше глибоке навчання відбулось саме через необхідність вирішення цих обмежень.

Контрольні запитання

1. Що таке обробка природної мови та яких розділів науки вона торкається?

2. Опишіть основні етапи еволюції NLP від символічних систем до сучасних мовних моделей.

3. Які основні NLP задачі ви знаєте? Наведіть по два приклади для кожної.

4. Як би ви формалізували наступну бізнес-потребу як машинно-навчальну задачу: "Нам потрібно виявляти залежністю від контексту неофіціальний тон у письмовому спілкуванні всередину компанії"?

5. Поясніть моделі представлення "Мішок слів" та визначте його основні обмеження.

6. У чому різниця між уніграмами, біграмами та триграмами? Чому найчастіше не використовуються N-грами з N > 3?

7. Наведіть формулу TF-IDF та поясніть, чому IDF знижує вагу частих слів.

8. Опишіть всі кроки побудови класичного конвеєра класифікації з TF-IDF та Logistic Regression.

9. Визначте Precision, Recall, F1-Score та поясніть, коли кожна метрика є особливо важливою.

10. Які основні переваги та обмеження класичних методів NLP на основі TF-IDF?

Рекомендована література

1. Bird, S., Klein, E., & Loper, E. (2009). “Natural Language Processing with Python” (O’Reilly). Класичний посібник для початківців з практичними прикладами.

2. Manning, C. D., Raghavan, P., & Schütze, H. (2008). “Introduction to Information Retrieval” (Cambridge University Press). Фундаментальна книга про методи пошуку та обробки тексту, включаючи детальне розкриття TF-IDF.

3. Goldberg, Y. (2015). “Neural Network Methods for Natural Language Processing” (Morgan & Claypool). Перехідний текст від класичних до нейронних методів.

4. Jurafsky, D., & Martin, J. H. (2020). “Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition” (3rd edition, online). Ґрунтовна лекційна книга на базі класичного курсу в Stanford.

5. Scikit-learn Документація: https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction. Практичні приклади реалізації TF-IDF в Python.

ЛЕКЦІЯ 2: Токенізація та підслівне моделювання

1. Що таке токенізація та чому вона важлива

Токенізація (Tokenization) — це перший та найважливіший крок обробки природної мови. Вона являє собою розділення тексту на менші одиниці, звані токенами (Tokens). Токени — це, зазвичай, слова, але можуть бути також підслова, окремі символи або навіть фонеми в залежності від рівня.

Чому токенізація важлива

1. **Базис для усіх подальших операцій** — Практично всі алгоритми

NLP працюють на рівні токенів, а не на рівні символів. Токенізація створює основу для представлення тексту.

2. **Встановлення границь слів** — У таких мовах як англійська, слова відділяються пробілами. Однак, у мовах без явних роздільників (наприклад, китайська, японська) це стає складною задачею.

3. **Обробка пунктуації та спеціальних символів** — Рішення про те, чи розглядати пунктуацію як окремий токен чи приєднувати її до слова, впливає на якість подальшої обробки.

4. **Управління контекстом** — Максимальна довжина послідовності у нейронних мережах часто вимірюється у токенах. Вибір методу токенізації впливає на кількість токенів у документі та, відповідно, на ефективність моделей.

5. **Обробка мовних явищ** — У деяких мовах складні морфологічні структури вимагають спеціальної токенізації.

2. Словесна (Word-level) токенізація: переваги та обмеження

Словесна токенізація

Це найпростіший підхід — розділення тексту на окремі слова. Зазвичай, це робиться шляхом розділення на пробіли та видалення пунктуації.

Приклад:

Вхід: “Привіт, світе! Як у тебе справи?” Вихід: [“Привіт”, “світе”, “Як”, “у”, “тебе”, “справи”]

Переваги

Простота реалізації, Людська інтуїтивність — слова є природними одиницями мови, Ефективна для деяких мов та задач та Менша кількість токенів у порівнянні з символ-рівневою токенізацією.

Обмеження

Проблема невідомих слів (Out-of-Vocabulary, OOV)

Найбільша проблема словесної токенізації — це явище OOV. Якщо модель навчалася на певному наборі слів (словниці) та зустрічає слово, якого немає у словниці, вона не може його обробити.

Приклад:

Словник (побудований на тренувальних даних): {“кіт”, “собака”, “бігти”, “спати”}

Тестовий текст: “Кіт та собака часто грають”

Слово “грають” не у словнику → OOV помилка

Це особливо проблемно для: Рідкісних слів, Неологізмів (новоутворених слів), Назв власних, Технічних термінів та Орфографічних помилок.

Морфологічна складність

Для мов з багатою морфологією (як українська), словесна токенізація може розділити слово на лексему та морфеми, які носять важливу інформацію. Наприклад, слова “читати”, “читав”, “читаючи” — це різні морфологічні форми одного слова.

Проблема невизначених границь слів

У деяких мовах (китайська, японська, корейська) слова не розділяються пробілами. Виявлення границь слів само по собі є складною задачею.

Проблема контрактацій та апострофу

“don’t” — це одне слово чи два? “don” та “t”? Або “do” та “not”?

3. Символ-рівневе (Character-level) представлення

Символ-рівнева токенізація

Замість розділення на слова, розділяємо на окремі символи.

Приклад:

Вхід: “Привіт” Вихід: [“П”, “р”, “и”, “в”, “і”, “т”]

Переваги

Немає проблеми OOV — всі символи будуть в словнику, Природне представлення морфології — префікси, суфікси, закінчення видно явно, Мінімальний розмір словника — для англійської мови ~100 символів, для української ~50-70 та Можливість роботи з орфографічними помилками та одруківками.

Обмеження

Набагато більше токенів — одне слово може бути 5-10 токенів, Контекстне вікно моделей стає коротшим у рамках глобального обмеження на довжину послідовності, Складнішим стає навчання залежностей на рівні слів та речень, Обчислювальна складність зростає через більшу кількість токенів та Рідше використовується на практиці як основна одиниця для сучасних моделей.

Гібридні підходи

Деякі системи використовують гібридний підхід — спочатку розділяють на слова, але слова, які не в словнику, розбиваються на символи. Це дає деякий баланс між ефективністю та обробкою OOV.

4. Мотивація для підслівної (Subword) токенізації

Проблема словесної токенізації полягає у тому, що вона дає занадто багато невідомих токенів для мов з багатою морфологією або для спеціалізованих доменів. Проблема символ-рівневої токенізації полягає у тому, що вона дає занадто багато токенів та робить навчання важким.

Підслівна токенізація знаходиться посередині — вона розбиває слова на смислові підслівні одиниці (Subword Units), які часто відповідають морфемам, але не завжди.

Ключові ідеї:

1. Розбиття рідкісних слів на частини
2. Зберігання компактної довжини послідовності
3. Обробка морфології природно
4. Дозвіл “складання” підслів у слова на рівні моделі

Приклад:

Вхід: “неймовірно” Глобальна токенізація (словесна): [“неймовірно”] (якщо у словнику) або OOV Підслівна токенізація: [“ней”, “мовір”, “но”] або [“не”, “йм”, “овір”, “но”]

Модель вивчає, як складати ці підслова разом для розуміння семантики.

5. Byte Pair Encoding (BPE): алгоритм крок за кроком з прикладом

Byte Pair Encoding (BPE) — один з найпопулярніших алгоритмів підслівної токенізації. Він був запропонований Sennrich et al. (2016) для машинного перекладу та пізніше прийнятий багатьма великими мовними моделями.

Основна ідея

BPE працює шляхом ітераційного об’єднання найбільш частої пари байтів (або символів) в один символ.

Алгоритм BPE: крок за кроком

1. **Ініціалізація**: Починаємо з набору всіх унікальних символів (байтів) у корпусі як початкові “токени”. Кожен символ повинен з’являтися як окремий токен.

2. **Підрахунок частот**: Для кожної суміжної пари токенів, підраховуємо кількість разів, коли ця пара з’являється поспіль у корпусі.

3. **Об’єднання**: Вибираємо найбільш частоту пару токенів та об’єднуємо її в один новий токен. Замінюємо всі входження цієї пари на

новий токен.

4. **Повтор**: Повторюємо кроки 2-3 для заданої кількості ітерацій (гіперпараметр, зазвичай 10,000-50,000).

Приклад

Припустимо, у нас є малий корпус слів:

Слова та їхні частоти: “hug” — 10 разів, “pug” — 5 разів, “bug” — 2 рази та “dug” — 1 раз.

Крок 1: Ініціалізація словника символів:

{‘h’: 1, ‘u’: 1, ‘g’: 1, ‘p’: 1, ‘b’: 1, ‘d’: 1}.

Представлення слів: h u g (10 разів) — появляються як “h”, “u”, “g” окремо, p u g (5 разів), b u g (2 рази) та d u g (1 раз).

Крок 2: Підрахунок пар.

Всі пари та їхні частоти: (h, u): 10 разів (у “hug”), (u, g): 10 + 5 + 2 + 1 = 18 разів, (p, u): 5 разів, (b, u): 2 рази та (d, u): 1 раз.

Найбільша пара: (u, g) з 18 появленнями.

Крок 3: Об’єднання (u, g) → UG.

Новий словник: {‘h’: 1, ‘u’: 1, ‘g’: 1, ‘p’: 1, ‘b’: 1, ‘d’: 1, ‘ug’: 1}.

Представлення слів після заміни: h ug (10 разів), p ug (5 разів), b ug (2 рази) та d ug (1 раз).

Крок 4: Наступна ітерація.

Підрахунок нових пар: (h, ug): 10 разів, (p, ug): 5 разів, (b, ug): 2 рази та (d, ug): 1 раз.

Найбільша пара: (h, ug) з 10 появленнями.

Об’єднання (h, ug) → HUG.

Представлення слів: hug (10 разів), p ug (5 разів), b ug (2 рази) та d ug (1

раз).

Продовжуючи, ми отримаємо все більше та більше підслів.

Застосування для нового тексту

Після навчання BPE маємо упорядкований список об’єднань. Для токенізації нового тексту:

1. Розділяємо на символи;
2. Послідовно застосовуємо об’єднання у порядку, у якому вони були навчені.

Приклад:

Нове слово: “huge”

Вихідно: h u g e

Застосуємо об’єднання (u, g) → ug: h ug e

Застосуємо об’єднання (h, ug) → hug: hug e

Вихід: [“hug”, “e”]

Властивості BPE

Невизначена довжина словника (визначена гіперпараметром кількості ітерацій), Частоти-орієнтована — рідкі слова залишаються розбитими на більше частин, Детерміністична — одна й та ж послідовність об’єднань завжди дає один результат та Ефективна — простий алгоритм з лінійної складністю для кожної ітерації.

6. WordPiece: як він відрізняється від BPE, використання у BERT

WordPiece — це алгоритм підслівної токенізації, розроблений у Google для японської та китайської мов обробки, а пізніше широко прийнятий у BERT та інших моделях.

Основні відмінності від BPE

1. **Критерій вибору пари**

BPE: Вибирає пару з найбільшою частотою.

WordPiece: Вибирає пару на основі ймовірності (Likelihood). Конкретно, вибирається пара, яка максимізує:

P(AB) / (P(A) × P(B))

де A та B — токени. Це вказує на те, наскільки "складається разом" пара у порівнянні з незалежністю.

2. **Мотивація**

WordPiece фокусується на комбінаціях, які є більш "симптоматичними" (більш імовірне поділення разом). BPE просто слідує частотам.

3. **Практичний результат**

WordPiece часто дає більш морфологічно та семантично значущі підслова.

Використання у BERT

BERT використовує варіант WordPiece, яка називається "##" позначення (Marker). Підслова, які являють собою частину слова (не на початку), позначаються з "##":

Приклад:

Слово: "неймовірний"

Можлива токенізація BERT WordPiece: ["ней", "##мовір", "##ний"]

Позначення "##" вказує на те, що це продовження попередньої лексеми.

Це дозволяє моделі розуміти структуру слова та виконувати задачі, як морфологічний аналіз або визначення границь морфем.

7. Unigram Language Model токенізатор: SentencePiece

SentencePiece — це більш сучасний підхід до підслівної токенізації, розроблений у Google та широко використовується у великих мовних моделях (наприклад, T5, XLNet, ALBERT).

Основна ідея Unigram LM

На відміну від BPE та WordPiece, які явно моделюють процес об'єднання або розбиття, Unigram Language Model підходить до токенізації як до задачі вибору оптимального набору підслів:

$$P(\mathbf{x}) = \prod_{i=1}^{M} P\,(x_i).$$

Модель вважає кожний можливий розбір тексту та обирає розбір з найвищою мовною моделлю вірогідністю:

$P(\text{текст}) = P(t_1) \times P(t_2|t_1) \times P(t_3|t_1, t_2) \times \ldots$

Для спрощення, часто використовується уніграма модель (звідки й назва):

$P(\text{текст}) = P(t_1) \times P(t_2) \times P(t_3) \times \ldots,$

де $P(t_i)$ — це "вартість" (cost) або "вірогідність" токена t_i.

Алгоритм тренування Unigram LM

1. **Ініціалізація**: Починаємо з великого словника (наприклад, всі символи + часті підслова).
2. **E-крок (Expectation)**: Для кожного слова у тренувальному корпусі, знаходимо K найбільш вірогідних розборів.
3. **M-крок (Maximization)**: Оновлюємо вірогідності $P(t_i)$ на основі частоти, з якою підслова з'являються у найбільш вірогідних розборах.
4. **Видалення**: Видаляємо найменш вірогідні підслова зі словника.
5. **Повтор**: Повторюємо E та M кроки, поки словник не досягне

бажаного розміру.

SentencePiece у практиці

SentencePiece має кілька особливостей:

Обробка пробілів: SentencePiece трансформує пробіли у спеціальний токен (▁ або), що дозволяє моделям краще розуміти границі слів.

Підтримка багатьох мов: SentencePiece добре працює з мовами без явних роздільників слів.

Детермінованість: На відміну від деяких інших методів, SentencePiece дає однозначне розбиття для кожного тексту.

Приклад використання:

```
import sentencepiece as spm
# Тренування токенізатора
spm.SentencePieceTrainer.train(
    input='corpus.txt',
    model_prefix='ukr_tokenizer',
    vocab_size=8000,
    model_type='unigram'
)
# Завантаження та використання
sp = spm.SentencePieceProcessor()
sp.Load('ukr_tokenizer.model')
tokens = sp.EncodeAsPieces('Привіт, світе!')
# Приклад результату: ['▁При', 'віт', ',', '▁світе', '!']
```

8. Практичні аспекти: тренування токенізатора на користувацькому корпусі

Хоча сучасні моделі часто поставляються з попередньо навченими токенізаторами, іноді необхідно тренувати власний, особливо для:

Спеціалізованих доменів (медицина, право, технологія), Менш поширених мов та Попереджень адаптації (Domain Adaptation).

Кроки тренування токенізатора

1. **Зібрати корпус**: Зберіть репрезентативний текстовий корпус для вашого домену/мови.
2. **Вибрати алгоритм**: BPE, WordPiece, Unigram LM тощо.
3. **Встановити гіперпараметри**:
   - Розмір словника (vocabulary_size): обичайно 8K-50K токенів
   - Кількість ітерацій / процент видалення
   - Мінімальна частота слова
4. **Тренування**: Запустити алгоритм тренування на корпусі.
5. **Оцінювання та ітерація**: Перевірити, як токенізатор працює на тестових даних, виконати коригування гіперпараметрів при необхідності.

Приклад з HuggingFace Tokenizers

```
from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import Whitespace
# Створення токенізатора
tokenizer = Tokenizer(BPE())
# Встановлення pre-tokenizer (розділяє на слова по пробілам)
tokenizer.pre_tokenizer = Whitespace()
# Налаштування trainer
trainer = BpeTrainer(
    vocab_size=10000,
    special_tokens=["<unk>", "<pad>", "<bos>", "<eos>"],
    min_frequency=2
```

```
)
# Тренування на файлі
tokenizer.train(["corpus.txt"], trainer=trainer)
# Використання
encoded = tokenizer.encode("Привіт, світе!")
print(encoded.tokens)  # Приклад: ['При', 'віт', ',', '▁світе', '!']
```

9. Вплив токенізації на задачі нижче по конвеєру, довжину контексту, ефективність LLMs

Вибір токенізатора має далекосяжні наслідки для всієї системи.

Вплив на якість

OOV та морфологія: Добрий токенізатор обробляє OOV та морфологічну структуру, дозволяючи моделі краще узагальнювати на нові слова та форми.

Семантика підслів: Якщо токенізатор розбиває семантично пов'язані слова однаково (префікси, суфікси), це може допомогти моделі розуміти відносини між словами.

Шум та артефакти: Поганий токенізатор може ввести шум у представлення. Наприклад, якщо технічна термін розбита недоречно.

Вплив на довжину контексту

Більшість трансформер-моделей має максимальну довжину послідовності (context length), наприклад 512, 2048, або 4096 токенів.

Якщо токенізатор дає багато токенів на слово (характер-рівневий), то модель може оброблювати менше слів. та Якщо токенізатор дає мало токенів на слово (великі підслова), то модель може обробляти більше слів, але може мати більше OOV помилок.

Приклад:

Текст: “This is a test document.” (5 слів)

Символ-рівневий токенізатор: 25 токенів BPE з 30K словником: 5 токенів WordPiece BERT: 7 токенів ([“This”, “is”, “a”, “test”, “do”, “##cument”, “.”])

Вплив на ефективність та обчислювальні витрати

Меньше токенів = швидша інференція, менше пам’яті та Більше токенів = довший контекст, можливо краще розуміння довгих документів.

Для LLMs, які коштують O(n^2) відносно довжини послідовності (через механізм уваги), число токенів критично впливає на швидкість та пам’ять.

10. Токенізація для української та інших не-англійських мов

Обробка не-англійських мов створює специфічні виклики для токенізації.

Виклики для мов з вам морфологією (як українська)

Багате префіксальне та суфіксальне утворення, Складна система часів, осіб та відмінків, що впливає на форму слова та Складні дієслова з префіксами, які можуть змінювати значення.

Приклад (українська морфологія):

Лексема “читати”: читати (інфінітив), читаю (теперішній час, 1 особа, однина), читав (минулий час, чоловічий род), читаючи (дієприкметник) та прочитати (перфективна форма, інший префікс).

Хороший токенізатор повинен розпізнавати, що всі ці форми пов’язані.

Виклики для мов без явних роздільників слів (як китайська)

Потреба у виявленні границь слів (Word Segmentation) та Вибір рівня детальності: символи, слова, фрази.

Рекомендації для української

1. **Використовувати спеціалізовані токенізатори**: Наприклад, XLM-RoBERTa або інші моделі, навчені на багатомовних даних,

включаючи українську.

2. **Тренувати власні**: Якщо в домену/задачі є особливості, можна тренувати власний токенізатор на українських текстах.

3. **Морфологічна лематизація**: У деяких випадках корисно лематизувати (привести до канонічної форми) слова перед токенізацією.

4. **Обробка закінчень та префіксів**: Модель повинна розуміти, що "-ся", "-те", префікси та суфікси змінюють слово, але зберігають основний смисл.

Контрольні запитання

1. Визначте токенізацію та поясніть, чому вона є першим кроком у обробці природної мови.

2. Що таке проблема Out-of-Vocabulary (OOV) та чому вона виникає у словесній токенізації?

3. Порівняйте переваги та обмеження символ-рівневої та словесної токенізації.

4. Поясніть мотивацію для підслівної токенізації та яку проблему вона вирішує.

5. Опишіть алгоритм Byte Pair Encoding крок за кроком. Чому вибирається найбільш часта пара?

6. У чому різниця між BPE та WordPiece у відношенні до вибору пари для об'єднання?

7. Поясніть концепцію Unigram Language Model токенізатора та як вона відрізняється від BPE.

8. Що таке SentencePiece та які його особливості для обробки багатомовних текстів?

9. Як вибір токенізатора впливає на довжину контексту та

обчислювальну ефективність LLM?

10. Які виклики виникають при токенізації української мови та як їх вирішувати?

Рекомендована література

1. Sennrich, R., Haddow, B., & Birch, A. (2016). “Neural Machine Translation of Rare Words with Subword Units” (Proceedings of the 54th Annual Meeting of the ACL). Оригінальна стаття про BPE.

2. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding” (arXiv preprint arXiv:1810.04805). Описує WordPiece токенізацію у контексті BERT.

3. Kudo, T., & Richardson, J. (2018). “SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text generation” (arXiv preprint arXiv:1808.06226). Оригінальна стаття про SentencePiece.

4. HuggingFace Tokenizers Documentation: https://huggingface.co/docs/tokenizers/. Практичні приклади та документація для тренування та використання різних токенізаторів.

5. HuggingFace Transformers Tokenizer Guide: https://huggingface.co/docs/transformers/tokenizer_summary. Порівняння токенізаторів у різних попередньо навчених моделях.

6. Лукашевич, Н. В., та ін. (2016). “Теория обработки текста на украинском языке” (Київський державний лінгвістичний університет). Спеціалізована про обробку української мови.

ЛЕКЦІЯ 3: Нейронні ембедінги та дистрибутивна семантика

1. Мотивація: чому щільні ембедінги замість розріджених векторів

Традиційні підходи в обробці природної мови опирались на лексичні представлення, засновані на схемах типу “мішок слів” (bag-of-words) та TF-IDF. Ці методи породжують розріджені, високовимірні вектори, які мають суттєві недоліки:

1. **Розрідженість**: вектори містять переважно нулі, що неефективно для обчислень;
2. **Ортогональність**: слова, які розглядаються як різні розмірності, є повністю ортогональними один до одного, хоча вони можуть мати схожі значення;
3. **Відсутність смислової близькості**: семантично схожі слова розташовуються далеко одне від одного в просторі;
4. **Високовимірність**: часто призводить до проблеми прокляття розмірності (curse of dimensionality);
5. **Неефективність для розподілених обчислень**: розріджені матриці вимагають спеціальних алгоритмів для GPU обробки.

Щільні ембедінги (dense embeddings) вирішують ці проблеми, представляючи слова як невеликі, щільно заповнені вектори в низьковимірному просторі (зазвичай 100–300 розмірів). Основна ідея полягає в тому, що семантично близькі слова повинні мати близькі вектори, що природним чином виникає при тренуванні на великих корпусах.

2. Дистрибутивна гіпотеза (Harris, Firth)

Дистрибутивна гіпотеза утворює теоретичний фундамент для навчання ембедінгів. Вона стверджує:

“Слова, що зустрічаються в схожих контекстах, мають схожі значення”.

Гіпотеза була сформульована Зелігом Харрісом в 1954 році та розвинена John Rupert Firth в його знаменитій фразі: “Слово знається за компанією, яку воно тримає” (You shall know a word by the company it keeps).

Математичний вираз цієї гіпотези: якщо два слова часто з’являються в однакових контекстах, то їхні представлення мають бути близькими. Контекст може бути визначений різними способами:

Лінійний контекст: слова, що знаходяться на відстані ±n токенів (вікно контексту), **Синтаксичний контекст**: слова з однаковими синтаксичними залежностями та **Семантичний контекст**: слова, що інтерпретуються як семантичні аргументи однієї предикати.

Ця гіпотеза виявилась надзвичайно плідною. Емпіричні дослідження показали, що векторні простори, побудовані на її основі, мають багато цікавих властивостей: геометричні відносини між векторами відображають семантичні та синтаксичні відносини між словами.

3. Матриці входження та скорочення розмірності (SVD/LSA)

Перший практичний спосіб побудови щільних ембедінгів базувався на матрицях входження (co-occurrence matrices) та методах скорочення розмірності.

Побудова матриці входження:

1. Визначимо контекстне вікно розміру 2c (c слів ліворуч, c слів праворуч);
2. Для кожної пари (слово, контекстне_слово), що з’являється в корпусі, збільшимо відповідний елемент матриці на 1;
3. Отримаємо матрицю M розмірності |V| × |V|, де |V| — розмір словника.

Ця матриця часто буває дуже розрідженою та високовимірною. Стандартна обробка включає:

Масштабування TF-IDF: надання більшої ваги рідким, але інформативним вживанням та **Вибіркове ймовірносне зважування (PMI)**:

$$PMI(w, c) = \log(P(w, c) / (P(w) \times P(c))).$$

Це підкреслює специфічні асоціації, приховуючи загальні слова.

Скорочення розмірності за допомогою SVD:

Сингулярне розкладання (Singular Value Decomposition, SVD) розкладає матрицю M:

$$M = U\Sigma V^T.$$

Зберігаючи лише k найбільших сингулярних значень, отримуємо низькорозмірне наближення:

$$M \approx U_k \Sigma_k V_k^T.$$

Рядки матриці U_k служать як щільні ембедінги розмірності k. Цей метод відомий як **Latent Semantic Analysis (LSA)**.

Однак LSA мав обмеження:

Важко оновлювати при додаванні нових слів до корпусу, Обробка матриці розмірності $|V| \times |V|$ обчислювально дорога та Не масштабується добре для дійсно великих словників (мільйони слів).

4. Word2Vec: архітектури CBOW та Skip-gram

Word2Vec (представлений Mikolov et al. у 2013 році) революціонізував область. Замість явного розкладання матриці входження, Word2Vec навчає щільні ембедінги за допомогою простої нейронної мережі.

Архітектура CBOW (Continuous Bag of Words):

CBOW передбачає слово за його контекстом:

input: $[w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}]$
→ усереднення їхніх векторів,
→ прихований шар розмірності d,

→ output: $P(w_t \mid \text{контекст})$.

Формально: дано контекстні слова в межах вікна розміру c, модель передбачає цільове слово w_t, максимізуючи:

$$L = \log P(w_t \mid w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}).$$

Архітектура Skip-gram:

Skip-gram виконує зворотну операцію: передбачає контекстні слова за цільовим словом:

input: w_t
→ прихований шар розмірності d,
→ output: $P(w_{t-c}, ..., w_{t+c} \mid w_t)$.

Цільова функція:

$$L = \log P(w_{t-c}, \ldots, w_{t+c} \mid w_t) = \Sigma_{-c \le j \le c,\, j \ne 0} \log P(w_{t+j} \mid w_t).$$

Skip-gram часто працює краще, особливо на малих корпусах та для рідких слів.

Тренувальна мета: Negative Sampling:

Обчислення повної softmax для словника розміру |V| дорого обходиться. Negative Sampling замінює це на бінарну класифікацію:

$$P(w_O|w_I) = \frac{\exp\left(\mathbf{v}'_{w_O}{}^{T}\mathbf{v}_{w_I}\right)}{\sum_{w=1}^{V} \exp\left(\mathbf{v}'_{w}{}^{T}\mathbf{v}_{w_I}\right)}.$$

Для правильної пари (w, c), максимізуємо:

$$\log \sigma(v_c \cdot v_w) + \Sigma_{i=1}^{k} E_{w_i \sim P_n} [\log \sigma(-v_{w_i} \cdot v_w)],$$

де σ — логістична функція, k — кількість негативних зразків, P_n — розподіл негативних зразків.

Це робить тренування значно швидшим ($k << |V|$), зберігаючи якість навчання.

Альтернатива: Hierarchical Softmax:

Інший підхід побудови ієрархічного дерева над словником, як у деревах Хаффмана, зменшує складність з O(|V|) до O(log |V|).

5. FastText: субсловні ембедінги та OOV слова

FastText (Bojanowski et al., 2017) розширює Word2Vec, обробляючи слова як суми n-грамних векторів символів (character n-grams).

Замість кожного слова w мати один вектор v_w, FastText представляє слово як суму векторів його n-грам:

$$\mathbf{v}_w = \sum_{g \in G_w} \mathbf{z}_g,$$

Стандартно використовуються n-грами розмірів від 3 до 6. Наприклад, для слова “привіт” (5 букв):

bigrams: <п, пр, ри, ив, ві, іт, т>, trigrams: <пр, при, при, рив, ивіт, вітт, т> та тощо.

Переваги:

1. **OOV слова**: навіть якщо слово не з’явилось під час тренування, його вектор можна отримати з векторів його n-грам;
2. **Морфологія**: слова з однаковими префіксами або суфіксами матимуть більш схожі вектори;
3. **Рідкі слова**: тренування на субсловному рівні забезпечує кращі представлення для малочастотних слів;
4. **Morfologically rich languages**: особливо корисно для української та інших мовлення з багатою морфологією.

Тренування FastText здійснюється аналогічно Word2Vec (з CBOW або Skip-gram), але з інфіксацією та aggregation на рівні n-грам.

6. Тренування ембедінгів на власних корпусах

На практиці часто потрібно навчити ембедінги на доменних даних. Процес складається з кількох етапів:

Підготовка корпусу:

1. **Токенізація**: розділити текст на токени (слова). Для української: враховуйте апострофи, дефіси, що є частиною слів;
2. **Нижній регістр**: зазвичай конвертують до нижнього реєстру, хоча для деяких завдань зберігання капіталізації корисне;
3. **Видалення розривів**: видалення дуже рідких слів (крім випадків, коли ваш словник має особливі потреби);
4. **Видалення стоп-слів**: необов'язково, але часто покращує якість.

Конфігурація параметрів:

Основні параметри включають **vector_size** (розмірність ембедінгу з типовими значеннями 100, 200, 300; більші значення для великих корпусів), **window** (розмір контекстного вікна, типово 5 або 10; менші значення дають синтаксичну подібність, більші — тематичну), **min_count** (мінімальна частота слова, видаляє дуже рідкі слова, типово 5), **workers** (кількість потоків для паралельного тренування) та **epochs** (кількість проходів по корпусу, типово 5-10).

Тренування:

```
from gensim.models import Word2Vec
sentences = [['слово1', 'слово2', ...], ...]  # токенізовані речення
model = Word2Vec(
    sentences=sentences,
    vector_size=300,
    window=5,
    min_count=5,
    workers=4,
```

```
    epochs=5,
    sg=1  # 1=Skip-gram, 0=CBOW
)
model.save('my_embeddings.model')
```

Поради:

Більший корпус дозволяє більш надійно вчити більш високовимірні ембедінги, контекстне вікно залежить від цілі (для синтаксичних завдань менше вікно краще), та тренування на доменних даних часто дає кращі результати для спеціалізованих завдань.

7. Завантаження та використання попередньо навчених ембедінгів

Зазвичай немає потреби навчати ембедінги з нуля, оскільки існують високоякісні попередньо навчені моделі.

Завантаження з gensim:

```
from gensim.models import KeyedVectors
# Завантаження попередньо навченої моделі
model = KeyedVectors.load('path/to/model.model')
# Доступ до вектора
word_vector = model['слово']  # numpy масив розмірності 300
# Найбільш схожі слова
similar_words = model.most_similar('слово', topn=10)
# Аналогії
result = model.most_similar(positive=['король', 'жінка'], negative=['чоловік'])
$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} +
\mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}$$
# Очікується результат схожий на 'королева'
```

Інші джерела ембедінгів:

1. **FastText**: попередньо навчені моделі для 157 мов від Facebook;

2. **GloVe**: глобальні векторні представлення слів (від Стенфорда);

3. **ELMo**: контекстуальні ембедінги від AllenAI (2018);

4. **Контекстуальні ембедінги з трансформерів**: BERT, RoBERTa (див. наступні лекції).

Включення ембедінгів в моделі:

```
import torch
import torch.nn as nn
class TextClassifier(nn.Module):
  def __init__(self, embeddings, hidden_dim, num_classes):
    super().__init__()
    self.embedding = nn.Embedding.from_pretrained(embeddings)
    self.lstm = nn.LSTM(embeddings.shape[1], hidden_dim, batch_first=True)
    self.classifier = nn.Linear(hidden_dim, num_classes)
  def forward(self, input_ids):
    embedded = self.embedding(input_ids)
    lstm_out, (h_n, c_n) = self.lstm(embedded)
    return self.classifier(h_n[-1])
```

8. Оцінювання: внутрішня та зовнішня оцінка

Оцінювання якості ембедінгів може проводитись двома способами:

Внутрішня оцінка (Intrinsic Evaluation):

1. **Завдання на аналогії**:

   – Синтаксичні: “король” – “королева” ≈ “принц” – ?

   – Семантичні: “Париж” – “Франція” ≈ “Київ” – ?

Вектор = embedding(‘королева’) + embedding(‘принц’) – embedding(‘король’). Знаходимо найближче слово до отриманого вектора.

Accuracy = (кількість правильних відповідей) / (всього запитань).

2. **Оцінка подібності слів**:

– Порівняння косинусної подібності між ембедінгами та людськими судженнями про подібність.

– Датасети: RareWord Similarity, SimLex-999, WordSim353.

– Метрика: Spearman ρ кореляція між машинними та людськими рейтингами.

```
from scipy.stats import spearmanr
human_scores = [...]  # людські судження про подібність
model_scores = [cos_sim(w1, w2) for w1, w2 in word_pairs]
correlation, p_value = spearmanr(human_scores, model_scores)
```

Зовнішня оцінка (Extrinsic Evaluation):

Використання ембедінгів як feature у downstream задачах:

1. **Класифікація текстів**: набір текстів з лейблами;
2. **Визначення сентименту**: позитивно/негативно розмічені рецензії;
3. **Вилучення назв сутностей**: виділення осіб, організацій, місць;
4. **Завдання на подібність текстів**: визначення парафрази.

Метрики залежать від завдання (точність, F1, AUC тощо).

9. Візуалізація: t-SNE та UMAP

Хоча ембедінги живуть у високовимірних просторах (300+ розмірів), ми можемо їх візуалізувати в 2D або 3D.

t-Distributed Stochastic Neighbor Embedding (t-SNE):

t-SNE є непараметричним методом зменшення розмірності, який зберігає локальні структури:

1. Обчислюється подібність між точками в початковому просторі

(p_{ij});

2. Градієнтний спуск мінімізує KL дивергенцію між p_{ij} та подібністю в низькорозмірному просторі (q_{ij});

3. Результат часто показує кластери семантично близьких слів.

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
# Припустимо, embeddings має форму (num_words, 300)
tsne = TSNE(n_components=2, random_state=42)
embeddings_2d = tsne.fit_transform(embeddings)
plt.scatter(embeddings_2d[:, 0], embeddings_2d[:, 1])
# Додати лейбли до точок для слів
for i, word in enumerate(words):
   plt.annotate(word, (embeddings_2d[i, 0], embeddings_2d[i, 1]))
plt.show()
```

UMAP (Uniform Manifold Approximation and Projection):

UMAP є альтернативою t-SNE з кількома перевагами:

Швидше тренується, Краще зберігає глобальну структуру, Детерміністичніший та Масштабується до більших наборів даних.

```
import umap
reducer = umap.UMAP(n_components=2, random_state=42)
embeddings_2d = reducer.fit_transform(embeddings)
```

Обидва методи корисні для дослідження та розуміння структури ембедінгів.

10. Використання ембедінгів у downstream моделях

Ембедінги служать як input до більш складних архітектур.

Логістична регресія над ембедінгами:

```
from sklearn.linear_model import LogisticRegression
# Припустимо, для кожного тексту ми маємо aggregated embedding (наприклад, середнє)
X_train = np.array([np.mean(embeddings[tokens], axis=0) for tokens in train_texts])
y_train = np.array(train_labels)
clf = LogisticRegression()
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)
```

Багатошаровий перцептрон (MLP):

```
import torch.nn as nn
class MLPClassifier(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, num_classes):
        super().__init__()
        self.fc1 = nn.Linear(embedding_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, num_classes)
    def forward(self, x):
        # x має форму (batch_size, embedding_dim)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

RNN/LSTM над послідовностями ембедінгів:

```
class RNNClassifier(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, num_classes):
        super().__init__()
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, num_classes)
```

```python
    def forward(self, x):
        # x має форму (batch_size, sequence_length, embedding_dim)
        lstm_out, (h_n, c_n) = self.lstm(x)
        # Використовуємо останній прихований стан
        return self.fc(h_n[-1])
```

Ці архітектури часто показують добрі результати, особливо коли ембедінги попередньо навчені на великих корпусах.

Контрольні запитання

1. Назвіть три основні недоліки розріджених векторів типу TF-IDF порівняно зі щільними ембедінгами.
2. Сформулюйте дистрибутивну гіпотезу та поясніть, як вона обґрунтовує навчання ембедінгів.
3. Які основні кроки при побудові матриці входження для дистрибутивної семантики?
4. Поясніть різницю між архітектурами CBOW та Skip-gram моделей Word2Vec.
5. Чому negative sampling робить тренування Word2Vec значно швидшим за стандартний softmax?
6. Як FastText вирішує проблему OOV (Out-of-Vocabulary) слів, і чому це особливо корисно для української мови?
7. Які параметри найбільш важливі при тренуванні Word2Vec на власному корпусі, і як їх вибір впливає на результат?
8. Поясніть різницю між внутрішньою та зовнішньою оцінкою якості ембедінгів. Наведіть приклади для кожної.
9. Чому t-SNE та UMAP дають різні візуалізації одних і тих же ембедінгів, і коли варто використовувати кожен з них?

10. Як можна включити попередньо навчені ембедінги (Word2Vec, FastText) в нейронну мережу для завдання класифікації текстів?

Рекомендована література

1. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. In Advances in Neural Information Processing Systems, 3111-3119.
2. Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching Word Vectors with Subword Information. Transactions of the Association for Computational Linguistics, 5, 135-146.
3. Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe: Global Vectors for Word Representation. In EMNLP, 1532-1543.
4. Harris, Z. S. (1954). Distributional Structure. Word, 10(2-3), 146-162.
5. Goldberg, Y., & Levy, O. (2014). word2vec Explained: Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method. arXiv preprint arXiv:1402.6115.

ЛЕКЦІЯ 4: Бенчмарки та метрики оцінювання в NLP

1. Чому оцінювання має значення в NLP

Оцінювання є критичною частиною розробки NLP систем. На відміну від комп'ютерного зору, де метрики часто менш суб'єктивні (точність розпізнавання об'єктів), NLP часто має досить суб'єктивний характер, особливо в завданнях на розуміння та генерацію мови.

Окремі ролі оцінювання:

1. **Розвиток**: розрізняти альтернативні підходи та комбінації параметрів;
2. **Порівняння з літературою**: переконатися, що модель досягає

конкурентних результатів;

3. **Монітор якості в production**: переконатися, що модель залишається стабільною з часом;

4. **Виявлення дрейфу даних**: змін у характеристиках вхідних даних;

5. **Бізнес валідація**: прив’язка до метрик, які мають справжню цінність для користувачів.

Без правильного оцінювання неможливо узнати, чи робить нова ідея справжній прогрес чи просто вписується в варіацію шумових даних.

2. Стандартні бенчмарки NLP: GLUE, SuperGLUE, SQuAD, WMT, MMLU

Бенчмарки (benchmarks) — це стандартизовані набори завдань, які дозволяють порівнювати моделі на рівні всієї індустрії.

GLUE (General Language Understanding Evaluation):

GLUE складається з 9 завдань на англійській мові, спрямованих на оцінювання розуміння природної мови:

1. **CoLA** (Corpus of Linguistic Acceptability): класифікація граматичної прийнятності речень (2 класи);

2. **SST-2** (Stanford Sentiment Treebank): визначення сентименту речень з фільмів (2 класи);

3. **MRPC** (Microsoft Research Paraphrase Corpus): виявлення парафрази (2 класи);

4. **QQP** (Quora Question Pairs): визначення парафрази для запитань (2 класи, ~364k прикладів);

5. **MNLI** (Multi-Genre Natural Language Inference): визначення природного наслідування (3 класи, ~393k прикладів);

6. **QNLI** (Question Natural Language Inference): інференція для

вилучення запитань (2 класи);

7. **RTE** (Recognizing Textual Entailment): визначення наслідування (2 класи, менший набір);

8. **WNLI** (Winograd Natural Language Inference): вирішення coreference (2 класи);

9. **STS-B** (Semantic Textual Similarity Benchmark): оцінювання подібності текстів (неперервна шкала 0-5).

Для кожного завдання розраховується метрика точності (або кореляція для STS-B), а потім усереднюється.

SuperGLUE:

Розширення GLUE з більш складними завданнями:

1. **BoolQ**: так/ні запитання з відповіддю відсутньої в тексті;

2. **CB** (CommitmentBank): визначення наслідування для діалогів (3 класи);

3. **COPA** (Choice Of Plausible Alternatives): вибір найплаусибільної альтернативи;

4. **MultiRC**: читання рекомендацій з багатьма правильними відповідями;

5. **ReCoRD**: читання рекомендацій з масками сутностей;

6. **RTE**: визначення наслідування;

7. **WiC** (Word-in-Context): визначення того, чи використовується слово з однаковим значенням в двох реченнях;

8. **WSC** (Winograd Schema Challenge): вирішення складних кореференцій.

SQuAD (Stanford Question Answering Dataset):

SQuAD складається з 100k+ запитань та відповідей на інформацію зі

сторінок англійської Вікіпедії:

Відповідь завжди є span (послідовність токенів) в абзаці та Метрики: точне збігання (Exact Match) та F1 над токенами.

WMT (Workshop on Machine Translation):

Щорічний конкурс машинного перекладу з розпорядками та оцінюванням. Включає:

Тестові набори для різних пар мов та Метрики: BLEU, TER (Translation Error Rate), METEOR (розглядатимуться далі).

MMLU (Massive Multitask Language Understanding):

Бенчмарк з 57 завдань, охоплюючих нижні школи, середні школи, коледж та випускні рівні. Включає дисципліни від математики до етики. Часто використовується для оцінювання великих мовних моделей (LLM).

3. Метрики класифікації: точність, прецизійність, повнота, F1

Для завдань класифікації існує багатозначна матриця помилок та похідні метрики.

Матриця помилок (Confusion Matrix):

Для бінарної класифікації:

	Передбачено позитивне	Передбачено негативне
Справді позитивне	TP (True Positive)	FN (False Negative)
Справді негативне	FP (False Positive)	TN (True Negative)

Метрики:

1. **Точність (Accuracy)**:

   Accuracy = (TP + TN) / (TP + TN + FP + FN)

   Доля правильних передбачень від усіх. Хороша для збалансованих наборів даних.

2. **Прецизійність (Precision)**:

Precision = TP / (TP + FP)

З усіх передбачень як позитивні, скільки насправді були позитивні? Важлива, коли хибне позитивне дорого.

3. **Повнота (Recall) або чутливість (Sensitivity)**:

Recall = TP / (TP + FN)

З усіх справді позитивних, скільки ми знайшли? Важлива, коли хибне негативне дорого.

4. **F1 оцінка**:

F1 = 2 × (Precision × Recall) / (Precision + Recall)

Гармонічне середнє между прецизійністю та повнотою. Добре для незбалансованих наборів.

Для багатокласної класифікації:

При більш ніж 2 класах розраховуються:

Macro F1: усереднюємо F1 для кожного класу (кожен клас має однакову вагу), **Micro F1**: визначаємо глобальні TP, FP, FN і розраховуємо F1 (еквівалентна точності) та **Weighted F1**: усереднюємо F1 за вагами класів (залежить від частоти кожного класу).

```
from sklearn.metrics import precision_recall_fscore_support, f1_score
y_true = [0, 1, 1, 2, 1]
y_pred = [0, 1, 0, 2, 1]
# Macro F1
f1_macro = f1_score(y_true, y_pred, average='macro')
# Micro F1
f1_micro = f1_score(y_true, y_pred, average='micro')
# Weighted F1
f1_weighted = f1_score(y_true, y_pred, average='weighted')
```

```
precision, recall, f1, support = precision_recall_fscore_support(
    y_true, y_pred, average='weighted'
)
```

4. ROC криві, PR криві, AUC

Коли критерій класифікатора виводить ймовірність (замість жорсткого 0 або 1), можна варіювати поріг для отримання різних компромісів між точністю та повнотою.

ROC крива (Receiver Operating Characteristic):

ROC крива будується шляхом:

1. Варіювання порога класифікатора від 0 до 1;
2. Для кожного порога розраховується True Positive Rate (TPR) та False Positive Rate (FPR):
   - TPR = TP / (TP + FN) = Recall,
   - FPR = FP / (FP + TN) = 1 - Specificity;
3. Побудова кривої: FPR по x-осі, TPR по y-осі.

Ідеальний класифікатор проходить через точку (0, 1). Випадковий класифікатор наслідує діагональну лінію.

AUC (Area Under the Curve):

AUC — площа під ROC кривою. Це ймовірність того, що класифікатор виставить вищий рейтинг випадковому позитивному прикладу, ніж випадковому негативному.

AUC = 1: ідеальний класифікатор, AUC = 0.5: випадковий класифікатор та AUC < 0.5: гірше за випадковий.

```
from sklearn.metrics import roc_curve, auc
y_true = [0, 0, 1, 1]
y_scores = [0.1, 0.4, 0.35, 0.8]
```

```
fpr, tpr, thresholds = roc_curve(y_true, y_scores)
roc_auc = auc(fpr, tpr)
import matplotlib.pyplot as plt
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], 'k--', label='Random')
plt.show()
```

PR крива (Precision-Recall Curve):

Альтернатива ROC, особливо корисна для незбалансованих наборів:

1. Варіювання порога;
2. Для кожного порога розраховується Precision та Recall;
3. Побудова кривої: Recall по x-осі, Precision по y-осі.

Ідеальний класифікатор знаходиться в верхньому правому куті (високі точність та повнота).

```
from sklearn.metrics import precision_recall_curve, average_precision_score
precision, recall, thresholds = precision_recall_curve(y_true, y_scores)
ap = average_precision_score(y_true, y_scores)
plt.plot(recall, precision, label=f'PR curve (AP = {ap:.2f})')
plt.show()
```

5. Генеративні метрики: BLEU, ROUGE, METEOR

Для завдань, де модель генерує текст (переклад, сумаризація), метрики складніші, оскільки є багато можливих правильних відповідей.

BLEU (Bilingual Evaluation Understudy):

BLEU порівнює n-грами між виробленим текстом та еталонними переводами.

Визначення:

1. Розраховується точність n-грам для n від 1 до 4:

p_n = (# n-грам у виробленому тексті, що присутні в еталоні) / (# n-грам у виробленому тексті);

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right), \quad BP = \min(1, e^{1-r/c}),$$

де BP — штраф за коротку відповідь:

BP = exp(1 - r/c) якщо c < r, інакше 1

(r — довжина еталону, c — довжина виробленого тексту).

Типовій BLEU для людського перекладу: 30-40. Для машинного перекладу: 20-30.

Обмеження BLEU:

Штрафує семантично еквівалентні перекладі за невключення точних n-грам, Може бути заангажованої щодо довгих текстів та Не враховує синоніми.

```
from nltk.translate.bleu_score import sentence_bleu, corpus_bleu
# Для одного речення
reference = [['це', 'чудовий', 'день']]
hypothesis = ['це', 'чудовий', 'день']
score = sentence_bleu(reference, hypothesis)
# Для корпусу
references = [[['це', 'чудовий'], ['це', 'гарний']]]
hypotheses = [['це', 'чудовий']]
score = corpus_bleu(references, hypotheses)
```

ROUGE (Recall-Oriented Understudy for Gisting Evaluation):

ROUGE коллекція метрик, особливо корисна для сумаризації:

1. **ROUGE-N**: n-грамова точність повнообчислена (як BLEU, але з акцентом на повноту):

ROUGE-N = (# n-грам у еталоні, що присутні у виробленому) / (# n-грам у еталоні),

$$\text{ROUGE-N} = \frac{\sum_{s \in \text{ref}} \sum_{\text{gram}_n \in s} \text{Count}_{\text{match}}(\text{gram}_n)}{\sum_{s \in \text{ref}} \sum_{\text{gram}_n \in s} \text{Count}(\text{gram}_n)};$$

2. **ROUGE-L**: найдовша загальна послідовність (LCS). Враховує послідовність слів, а не просто наявність;
3. **ROUGE-W**: зважена найдовша загальна послідовність.

ROUGE-1 часто корелює з людським оцінюванням для сумаризації краще, ніж BLEU.

```
from rouge_score import rouge_scorer
scorer = rouge_scorer.RougeScorer(['rouge1', 'rougeL'], use_stemmer=True)
scores = scorer.score(
    target='Це чудовий день',
    prediction='День чудовий'
)
```

METEOR (Metric for Evaluation of Translation with Explicit ORdering):

METEOR базується на точній відповідності, стемінгу, синонімії та перефразування.

1. Вирівнювання слів між еталоном та гіпотезою з використанням синонімів та варіантів;
2. Розрахування точності та повнообчислення на основі вирівнювання;
3. Штраф за фрагментацію (розриви у послідовності).

METEOR часто корелює з людським оцінюванням краще, ніж BLEU, але обчислювально дорожча.

6. Перплексія для мовних моделей

Перплексія (perplexity) — стандартна метрика для оцінювання мовних

моделей. Вона вимірює, наскільки добре модель передбачає наступне слово.

Визначення:

Перплексія набору тестів W = {w_1, w_2, …, w_N}:

PP(W) = P(w_1, w_2, …, w_N)^{-1/N} = (∏*{i=1}^{N} P(w_i | w_1, …, w*{i-1}))^{-1/N}.

На логарифмічній шкалі:

Інтуїція: перплексія — це середня кількість біт, потрібна для кодування кожного слова. Нижча перплексія = краща модель:

$$\mathrm{PPL}(W) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N} \log P(w_i|w_{<i})\right).$$

Приклад:

Якщо модель для невідомого слова призначає ймовірність 0.1, то це вносить log_2(0.1) ≈ -3.32 біта в перплексію.

```
import math
# Припустимо, model.log_prob(word | context) повертає лог-ймовірність
total_log_prob = 0.0
num_words = 0
for context, target_word in test_data:
    log_prob = model.log_prob(target_word, context)
    total_log_prob += log_prob
    num_words += 1
cross_entropy = -total_log_prob / num_words  # в натуральних логарифмах
perplexity = math.exp(cross_entropy)
print(f"Перплексія: {perplexity:.2f}")
```

Порівняння перплексій

Проста мовна модель на корпусі англійської Вікіпедії досягає перплексії ~200-300, державна мовна модель (п’ять років тому) мала ~30-50, а сучасні

великі мовні моделі демонструють 10-20 на різних benchmark.

7. Обмеження автоматичних метрик

Автоматичні метрики мають серйозні обмеження:

1. **Наявність жорсткого матчингу**: BLEU, ROUGE вимагають точного посиль до еталонних n-грам. Семантично еквівалентні варіанти розглядаються як невірні;

2. **Однорідність еталонів**: монорідність еталонів (однолітка як правильна відповідь). У дійсності багато запитань можуть мати кілька однаково правильних відповідей;

3. **Ненаглядні помилки**: метрика не розрізняє маленькі помилки від великих. Помилка в імені особи розглядається як помилка в артиклі;

4. **Залежність від домену**: метрики, які добре працюють на одному домені, можуть погано працювати на іншому;

5. **Відсутність контексту**: автоматичні метрики не розуміють контекст та призначення даних.

Навіть незважаючи на ці обмеження, автоматичні метрики цінні для: Швидкої перевірки прогресу під час розвитку, Монітування порівняних змін та Великомасштабного порівняння моделей.

8. Підходи до людського оцінювання

Людське оцінювання залишається золотим стандартом, особливо для завдань з неоднозначністю.

Схеми оцінювання:

1. **Рівні рейтингу** (Likert scale): оцінювачі оцінюють на шкалі від 1 до 5:

   – 5: Чудово, не вимагає редагування;

– 4: Добре, незначні редагування;

– 3: Задовільно, деякі помилки;

– 2: Погано, значні помилки;

– 1: Неприйнятно;

2. **Парне порівняння**: оцінювачам показують дві системи та просять вибрати кращу (або визнати рівність);

3. **Вимірювання помилок**: лист помилок із категоріями (граматичні, семантичні і т.д.) та затратою для кожної.

Міри вмісту:

Флі (Fleiss' kappa): міра узгодження між кількома оцінювачами та **Спірмен ρ**: кореляція між рейтингами пар оцінювачів.

Хорошою практикою є: 3+ оцінювачів на кожний приклад, Явні інструкції з оцінювання, Тренування оцінювачів на прикладах та Регулярна перевірка узгодження.

9. LLM-as-judge оцінювання

Недавно виник новий підхід: використання великих мовних моделей як оцінювачів.

Мотивація:

Людське оцінювання дороге та повільне та Велика мовна модель може зрозуміти контекст та семантику краще, ніж автоматичні метрики.

Приклад з використанням GPT-4:

```
from openai import OpenAI
client = OpenAI()
def evaluate_response(question, reference, candidate):
    prompt = f"""Порівняйте наступні відповіді на запитання.
Запитання: {question}
```

```
Еталонна відповідь: {reference}
Кандидат видповіді: {candidate}
Оцініть кандидат видповіді на шкалі від 1 до 5:
1 - Неправильна або беззмістовна
2 - Частково правильна, але з значними помилками
3 - В основному правильна, але з недоліками
4 - Правильна та корисна
5 - Відмінна, готова до використання
Надайте оцінку та коротко обґрунтуйте."""
    response = client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "user", "content": prompt}
        ],
        temperature=0.0  # Детерміністичні результати
    )
    return response.choices[0].message.content
```

Обмеження LLM-as-judge:

1. **Упередженість до однакової довжини**: LLM часто віддають перевагу довшим текстам;
2. **Цінність самовідповідей LLM**: якщо обидві відповіді від LLM, може виявити упередженість;
3. **Вартість**: запити до OpenAI API коштують грошей;
4. **Нерепродуктивність**: LLM на основі OpenAI API можуть мати недетерміністичні результати.

Незважаючи на обмеження, LLM-as-judge показує перспективу для масштабування оцінювання та часто корелює дуже добре з людським оцінюванням.

10. Вибір метрик для бізнес-цілей

Технічні метрики не завжди узгоджуються з бізнес-цілями.

Приклади виправлення:

1. **Рекомендаційна система**:

   – Технічна метрика: RMSE прогнозів рейтингів;

   – Бізнес-ціль: користувачі залишаються довше на сайті;

   – Кращий вибір: Click-through rate (CTR), дозвіл, утримання користувачів.

2. **Система розпізнавання токсичності**:

   – Технічна метрика: Точність;

   – Бізнес-ціль: мінімізувати шкідливий контент без впливу на інші повідомлення;

   – Кращий вибір: Precision для токсичності (мінімізувати false positives), але збалансувати з Recall (не пропустити токсичність).

3. **Перекладач з англійської на українську**:

   – Технічна метрика: BLEU;

   – Бізнес-ціль: користувачам легко читати переклад, можливо, без редагування;

   – Кращий вибір: людське оцінювання на 3-5 рівнях, за можливістю LLM-as-judge.

Процес вибору метрик:

1. Визначити бізнес-цель та KPI;
2. Визначити, як змінна NLP впливає на KPI;
3. Вибрати метрики, які корелюють з цією змінною;
4. Встановити мінімальні пороги на основі поточного стану та цілей;

5. Регулярно переглядати та коригувати.

Контрольні запитання

1. Поясніть різницю між внутрішньою та зовнішньою оцінкою в контексті бенчмарків NLP.

2. Що таке матриця помилок та як вона використовується для розрахування прецизійності, повнообчислення та F1?

3. Поясніть, чому Macro F1 та Weighted F1 дають різні результати для незбалансованого набору даних.

4. Як побудувати ROC криву та яка інтерпретація AUC?

5. Охарактеризуйте основні компоненти BLEU та його обмеження для оцінювання перекладу.

6. Чим ROUGE відрізняється від BLEU, і чому ROUGE краще для сумаризації?

7. Що таке перплексія для мовної моделі та як вона розраховується?

8. Назвіть три основні обмеження автоматичних метрик типу BLEU та ROUGE.

9. Які фактори варто враховувати при проведенні людського оцінювання для NLP систем?

10. Як вибрати правильні метрики для вирівнювання технічних результатів з бізнес-цілями?

Рекомендована література

1. Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2018). GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. arXiv preprint arXiv:1804.07461.

2. Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002). BLEU: a

Method for Automatic Evaluation of Machine Translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, 311-318.

3. Lin, C. Y. (2004). ROUGE: A Package for Automatic Evaluation of Summaries. In Proceedings of the Workshop on Text Summarization Branches Out.

4. Perplexity in Language Modeling. In Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT press.

5. Kenton, J. D. M. W. C., & Toutanova, L. K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of NAACL-HLT, 4171-4186.

6. Zhong, V., Xiong, C., & Socher, R. (2017). Seq2SQL: Generating Structured Queries from Natural Language. arXiv preprint arXiv:1709.00103.

7. Banerjee, S., & Lavie, A. (2005). METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, 65-72.

ЛЕКЦІЯ 5: Енкодерні та Seq2Seq моделі: T5, BART, переклад і сумаризація

1. Повтор: архітектура Transformer (енкодер, декодер, cross-attention)

Перш ніж розглядати специфічні моделі, коротко повернемось до архітектури Transformer, представленої у статті “Attention is All You Need” (Vaswani et al., 2017).

Основні компоненти:

1. **Multi-Head Self-Attention**:

Для кожної позиції в послідовності модель обчислює увагу до всіх

інших позицій:

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

де Q (Query), K (Key), V (Value) — лінійні проекції вхідних представлень.

Multi-head запускає кілька паралельних операцій attention з різними проекціями:

MultiHead(Q, K, V) = Concat(head_1, …, head_h) W^O.

2. **Cross-Attention**:

У seq2seq моделях Query береться з декодера, а K та V — з енкодера:

CrossAttention(Q_decoder, K_encoder, V_encoder).

Це дозволяє декодеру звертатися до релевантних частин вхідної послідовності при генерації кожного токену.

3. **Feed-Forward Networks**:

Після attention, кожна позиція обробляється двошаровою мережею:

FFN(x) = ReLU(xW_1 + b_1) W_2 + b_2.

4. **Позиційне кодування**:

Оскільки attention є перестановочно-інваріантним, додається позиційна інформація:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right),$$

Енкодер:

Стопка (зазвичай 12, 24 або 48) однакових шарів. Кожен шар містить: та Multi-head self-attention. Feed-forward мережа. та Residual connections та Layer

Normalization.

Декодер:

Подібна до енкодера, але з трьома підшарами на шар: Self-attention (з маскуванням для майбутніх позицій) та Cross-attention до енкодера. Feed-forward мережа.

Маскування в декодері:

При тренуванні та інференсі для позиції i закривають увагу до позицій j > i, щоб модель не могла “шахраювати”, дивлячись на майбутні токени.

2. Енкодерні моделі: BERT, RoBERTa — претренування та fine-tuning

BERT (Bidirectional Encoder Representations from Transformers):

BERT, представлений Google (Devlin et al., 2018), революціонізував NLP, ввівши ефективне претренування для глибоких енкодерів.

Архітектура:

Архітектура включає тільки енкодер без декодера, складається з 12 або 24 шарів (для BERT-base та BERT-large відповідно) та передбачає маскування позиції 0 для спеціального токену [CLS], який представляє весь приклад.

Претренування:

BERT тренується на двох objetivo:

1. **Masked Language Model (MLM)**:

   – Випадково маскуються 15% токенів входу (заміна на [MASK]);

   – Модель передбачає оригінальний токен;

   – Це мотивує модель розуміти контекст.

Технічно: Вектор маски з вихідної репрезентації токена [MASK] подається на vocab prediction head:

P(token) = softmax(hidden_state_[MASK] @ W),

де W — матриця вагів розміру d_model × |vocab|.

2. **Next Sentence Prediction (NSP)**:

– Дається пара речень;

– Модель передбачає, чи є друга наступною в документі або випадковою;

– Це сприяє розумінню зв'язків на рівні речення.

Fine-tuning:

Для downstream завдань (класифікація, вилучення сутностей):

1. Добавляється малий шар на вершину BERT;
2. Весь модель (або лише верхні шари) підлаштовуються на annotated даних;
3. Типовий fine-tuning: 2-5 епох, learning rate 2e-5 до 5e-5.

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from torch.optim import AdamW
from transformers import get_linear_schedule_with_warmup
model_name = "bert-base-multilingual-cased"  # Для української
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(
    model_name, num_labels=2
)
# Fine-tuning цикл
optimizer = AdamW(model.parameters(), lr=2e-5)
total_steps = len(train_loader) * num_epochs
scheduler = get_linear_schedule_with_warmup(
    optimizer, num_warmup_steps=0, num_training_steps=total_steps
)
```

```python
for epoch in range(num_epochs):
    for batch in train_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
        optimizer.zero_grad()
        outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
        scheduler.step()
```

RoBERTa (Robustly Optimized BERT Pretraining Approach):

RoBERTa (Facebook, 2019) покращила BERT через кращі практики претренування:

1. **Більше даних і більше тренування**: 160GB тексту (vs 16GB у BERT), 500K кроків;
2. **Видалення NSP**: NSP об'єктив виявився менш важливим;
3. **Більш довгі послідовності**: тренування на послідовностях до 512 токенів з більш високими batch sizes;
4. **Динамічне маскування**: маски обираються різними для кожної епохи (а не один раз на початку);
5. **Byte-Pair Encoding (BPE)**: замість WordPiece tokenization.

Результат: RoBERTa перевершує BERT на більшості бенчмарків.

3. Архітектура енкодер-декодер (seq2seq)

Seq2Seq моделі складаються з енкодера та декодера:

Enncoder: input_sequence → encoder_hidden_states
Decoder: encoder_hidden_states + input_start_token → output_sequence

Процес інферент (генерація):

1. Енкодер обробляє вхідну послідовність, виробляючи контекстні представлення (hidden states);
2. Декодер починається зі спеціального токену [BOS] (Beginning of Sequence);
3. На кожному кроці декодер має:
   - Вбудовування поточного токена;
   - Self-attention над попередніми генерованими токенами;
   - Cross-attention до енкодера;
   - Прогноз наступного токена;
4. Процес повторюється, поки не буде згенерований токен [EOS] або максимальна довжина.

Training: Teacher Forcing:

При тренуванні замість використання попередньо передбачених токенів (які можуть бути помилковими), модель видить правильні попередні токени. Це називається “teacher forcing”.

Без teacher forcing:

Step 1: Model predicts "Привіт" (correct)
Step 2: Model uses "Привіт" as input... and predicts "world" (error)
Step 3: Model uses "world" as input... compounding error

З teacher forcing:

Step 1: Model sees correct token "Привіт"
Step 2: Model sees correct token "світе"
Step 3: Model sees correct token "!"

Це прискорює тренування, але може привести до “exposure bias”: модель на тесті використовує свої помилкові передбачення, а при тренуванні —

правильні.

4. T5: “Text-to-Text Transfer Transformer” — уніфікована формулювання

T5 (Text-to-Text Transfer Transformer), представлена Google (Raffel et al., 2019), надає уніфіковану рамку для всіх NLP завдань.

Ключова ідея:

Всі завдання форматуються як text-to-text:

Input: "translate English to German: The house is wonderful"
Output: "Das Haus ist wunderbar"
Input: "summarize: The quick brown fox..."
Output: "A fast fox..."
Input: "sst2 sentence: This movie was great"
Output: "positive"

Архітектура:

Стандартний Transformer encoder-decoder з: Модель має різні розміри: T5-base містить 12 шарів енкодера та 12 шарів декодера, T5-large має 24 шарів енкодера та 24 шарів декодера, а T5-3B та T5-11B містять 48 шарів енкодера та 48 шарів декодера.

Претренування: Span Corruption

T5 використовує спеціалізований objective під назвою “span corruption”:

1. Вибираються випадкові spans (послідовності токенів) у вхідному тексті;
2. Вони замінюються спеціальним токеном [X], [Y], [Z] тощо;
3. Модель передбачає оригінальні spans.

Приклад:

Input: "The [X] fox jumps over [Y] lazy dog"

Target: "quick [X] the [Y] brown"

Це дозволяє моделі розуміти як на рівні слів, так і на рівні понять.

Коваріююча на 750GB тексту (C4 dataset):

T5 була натренована на Англійській Вікіпедії, веб-сторінках та книгах з загальною кількістю близько 180B токенів та розміром словника 32k.

Fine-tuning T5:

```
from transformers import T5Tokenizer, T5ForConditionalGeneration
model_name = "google-t5/t5-base"
tokenizer = T5Tokenizer.from_pretrained(model_name)
model = T5ForConditionalGeneration.from_pretrained(model_name)
# Для сумаризації
input_text = "summarize: The quick brown fox jumps over the lazy dog"
input_ids = tokenizer.encode(input_text, return_tensors="pt")
# Генерація
output_ids = model.generate(input_ids, max_length=50)
output_text = tokenizer.decode(output_ids[0], skip_special_tokens=True)
```

Переваги T5:

1. Уніфікована архітектура для багатьох завдань;
2. Відмінне претренування на C4;
3. Чудові результати на GLUE, SuperGLUE, SQuAD з мінімальним fine-tuning;
4. Легко адаптується до нових завдань завдяки текстовому формату.

5. BART: Denoising Autoencoder Претренування

BART (Denoising Autoencoder for Sequence-to-Sequence Learning), представлена Facebook (Lewis et al., 2019), поєднує ідеї BERT та seq2seq.

Архітектура:

ELECTRA доступна у варіантах з 6 або 12 шарами енкодера та 6 або 12

шарами декодера, займаючи середню позицію за розміром між BERT та T5.

Претренування: Denoising Task:

На відміну від BERT (MLM) та T5 (span corruption), BART є noising autoencoder:

1. Вхідний текст обурюється різними шумовими функціями;
2. Модель вивчає відновлення оригіналу.

Різні типи шумів: Методи попередньої обробки включають **token masking** (як у BERT, замінити 15% на [MASK]), **token deletion** (видалити 10% токенів), **text infilling** (як у T5, замінити spans одним [MASK]), **sentence permutation** (випадково переставити речення) та **document rotation** (переставити весь текст ротацією на випадкову позицію).

Комбінація цих шумів робить BART дуже гнучким.

Fine-tuning для сумаризації:

```
from transformers import BartForConditionalGeneration, BartTokenizer
model_name = "facebook/bart-large-cnn"
tokenizer = BartTokenizer.from_pretrained(model_name)
model = BartForConditionalGeneration.from_pretrained(model_name)
article = """Будівництво нового мосту почалось у січні. Мрість буде
з'єднувати..."
inputs = tokenizer(article, return_tensors="pt", max_length=1024,
truncation=True)
summary_ids = model.generate(inputs['input_ids'], num_beams=4,
max_length=150)
summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
```

BART, як T5, показує чудові результати на сумаризації та генеративних завданнях.

Різниця BART vs T5:

T5 пропонує універсальну формулювання, де всі завдання трактуються як text-to-text, у той час як BART дозволяє різнорідні вхідні та вихідні послідовності, роблячи його гнучкішим для деяких завдань. На практиці обидві моделі можна успішно застосовувати, а вибір залежить від конкретного завдання.

6. Fine-tuning для сумаризації з HuggingFace Trainer

HuggingFace Trainer *надає зручний API для fine-tuning моделей.*

```
from transformers import (
    T5Tokenizer, T5ForConditionalGeneration,
    Seq2SeqTrainingArguments, Seq2SeqTrainer,
    DataCollatorForSeq2Seq
)
from datasets import load_dataset
model_name = "google-t5/t5-base"
tokenizer = T5Tokenizer.from_pretrained(model_name)
model = T5ForConditionalGeneration.from_pretrained(model_name)
# Завантаження датасету
dataset = load_dataset("cnn_dailymail", "3.0.0")
# Препроцесінг
def preprocess(batch):
    inputs = ["summarize: " + doc for doc in batch['article']]
    model_inputs = tokenizer(inputs, max_length=1024, truncation=True)
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(batch['highlights'], max_length=128, truncation=True)
    model_inputs['labels'] = labels['input_ids']
    return model_inputs
dataset = dataset.map(preprocess, batched=True)
# Тренування
args = Seq2SeqTrainingArguments(
```

```
    output_dir="./t5-summarization",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    weight_decay=0.01,
    num_train_epochs=3,
    save_total_limit=3,
    predict_with_generate=True,
)
trainer = Seq2SeqTrainer(
    model=model,
    args=args,
    train_dataset=dataset['train'],
    eval_dataset=dataset['validation'],
    tokenizer=tokenizer,
    data_collator=DataCollatorForSeq2Seq(tokenizer, model=model),
)
trainer.train()
```

7. Fine-tuning для перекладу

Переклад відрізняється від сумаризації тим, що потребує більш точного відтворення деталей.

```
# Схожа до сумаризації, але без префікса
def preprocess_for_translation(batch):
    inputs = batch['en']  # англійські речення
    model_inputs = tokenizer(inputs, max_length=512, truncation=True)
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(batch['uk'], max_length=512, truncation=True)  #
українські
```

```
    model_inputs['labels'] = labels['input_ids']
    return model_inputs
# Гіпер-параметри можуть бути іншими:
args = Seq2SeqTrainingArguments(
    output_dir="./t5-translation",
    learning_rate=3e-4,  # Трохи вище для перекладу
    per_device_train_batch_size=8,
    num_train_epochs=5,  # Більше епох для більш складного завдання
    save_total_limit=3,
)
```

Практичні поради:

1. Переклад часто потребує більше даних, ніж сумаризація;
2. Виділяйте мовну пару явно у префіксі;
3. Використовуйте більш високий learning rate для перекладу;
4. Evaluation: для перекладу використовуйте BLEU або METEOR (про які розповідали раніше).

8. Beam search

При інференті, декодер потребує стратегії для вибору наступного токена на кожному кроці.

Greedy Decoding:

```
output_ids = model.generate(input_ids, max_length=50)
```

За замовчуванням, вибирається токен з найвищою ймовірністю. Швидко, але часто не оптимально.

Beam Search:

```
output_ids = model.generate(input_ids, max_length=50, num_beams=4)
```

Beam search утримує k найбільш ймовірних гіпотез на кожному кроці,

розкривши їх до довжини послідовності.

Алгоритм: 1. Почати з вхідної послідовності (1 гіпотеза). 2. Для кожного кроку генерації: - Розширити кожну гіпотезу на всі словник (k × |vocab| гіпотез). - Обрати k гіпотез з найвищим накопленим cScore (log ймовірності). 3. Повторити до [EOS] або max_length.

k=4 часто дає хороший компроміс між якістю та швидкістю.

Sampling (Temperature Sampling):

```
output_ids = model.generate(
    input_ids,
    max_length=50,
    do_sample=True,
    temperature=0.7,
    top_k=50,
    top_p=0.95
)
```

Замість вибору найбільш ймовірного токена, семплюємо з розподілу ймовірностей:

P(token) = softmax(logits / temperature):

$$P(x_t = w | x_{<t}) = \frac{\exp\left(\frac{z_w}{\tau}\right)}{\sum_{w'} \exp\left(\frac{z_{w'}}{\tau}\right)}.$$

Температурний параметр істотно впливає на форму розподілу ймовірностей. Коли температура менша за одиницю, розподіл стає більш зосередженим, що призводить до вибірки ймовірніших токенів. При температурі, рівній одиниці, розподіл залишається без змін. Коли температура більша за одиницю, розподіл стає більш рівномірним, що дозволяє отримувати більш різнородні відповіді.

Top-k та Top-p (nucleus sampling):

Дані методи базуються на двох ключових стратегіях. **Top-k** передбачає розгляд лише k найбільш ймовірних токенів, а **Top-p** орієнтується на найбільш ймовірні токени (до суми ймовірностей p >= 0.95). Ці методи можуть запобігти генеруванню низькоймовірних "хвіст"-токенів, які часто виявляються неадекватними.

Вибір стратегії:

Серед можливих стратегій слід виокремити beam search з параметром k=4, який найбільш підходить для наукових завдань та перекладу, де потрібна висока точність, та sampling, який кращий для творчих завдань та діалогу, де різнорідність відповідей є цінною. Можна також комбінувати обидва підходи для отримання оптимальних результатів.

9. Практичні вправи: пам'ять, розмір batch, learning rate scheduling

Пам'ять (Memory):

Тренування великих seq2seq моделей може потребувати багато пам'яті GPU:

Memory = (parameter_count × 4 bytes)
+ (batch_size × sequence_length × hidden_size × 4 bytes) × 2
+ (optimizer_state)

Для T5-large (770M параметрів) с batch_size=16 та seq_length=512 потребується ~20GB.

Способи зменшити пам'ять:

1. **Gradient Accumulation**: виконувати forward/backward для малих batches, усереднювати градієнти перед update:

```
args = Seq2SeqTrainingArguments(
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,  # Еквівалент batch_size=16
)
```

2. **Mixed Precision Training**: використовувати float16 замість float32:

```
args = Seq2SeqTrainingArguments(
    fp16=True,  # або bf16=True на TPU
)
```

3. **Gradient Checkpointing**: у вибраних шарах не утримувати activeations, переобчислювати під час backward:

```
model.gradient_checkpointing_enable()
```

Learning Rate Scheduling:

```
from transformers import get_linear_schedule_with_warmup
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4)
num_steps = len(train_loader) * num_epochs
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=0.1 * num_steps,  # 10% warmup
    num_training_steps=num_steps
)
for step, batch in enumerate(train_loader):
    outputs = model(**batch)
    loss = outputs.loss
    loss.backward()
    optimizer.step()
    scheduler.step()
    optimizer.zero_grad()
```

Типові темпи навчання залежать від типу моделі. Для BERT fine-tuning рекомендуються значення від 2e-5 до 5e-5, для T5 fine-tuning — від 1e-4 до 3e-4, а для декодерів LLM — від 5e-5 до 2e-4.

Batch Size та Accumulation:

Більший розмір батчу дозволяє отримати кращу оцінку градієнтів, проте потребує більше пам'яті. Типові розміри батчу для fine-tuning складають 8, 16 або 32. Накопичення градієнтів дозволяє досягти еквівалентного ефективного розміру батчу: маючи batch_size=4 на кожному з 4 GPU з 4 кроками накопичення, отримуємо ефективний розмір 16.

10. Контрольні запитання

1. Поясніть роль cross-attention в seq2seq моделях та як вона різниться від self-attention.
2. Що таке masked language modeling (MLM) у BERT та чому це корисно для претренування?
3. Яка була ключова ідея T5 щодо представлення завдань, і яка її перевага?
4. Поясніть різні функції шуму, що використовуються під час претренування BART.
5. Що таке teacher forcing при тренуванні seq2seq моделей, і яка його потенціальна проблема?
6. Як fine-tunuvati T5 для завдання сумаризації на користувацьких даних за допомогою HuggingFace Trainer?
7. Поясніть різницю між greedy decoding та beam search генерацією.
8. Як temperature, top-k та top-p впливають на результати sampling-based генерації?
9. Як зменшити пам'ять при тренуванні великих seq2seq моделей? Назвіть три методи.
10. Які гіпер-параметри найбільш важливі при fine-tuning T5/BART для конкретного доменного завдання?

Рекомендована література

1. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … & Polosukhin, I. (2017). Attention is All You Need. In Advances in Neural Information Processing Systems, 5998-6008.

2. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., … & Liu, P. Q. (2019). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. Journal of Machine Learning Research, 21(140), 1-67.

3. Lewis, M., Liu, Y., Goyal, N., Grangier, D., Zellers, A., Dauphin, Y., & Schwenk, H. (2019). BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. arXiv preprint arXiv:1910.13461.

4. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of NAACL-HLT.

5. Holtzman, A., Buys, J., Du, L., Forbes, M., & Choi, Y. (2019). The Curious Case of Neural Text Degeneration. arXiv preprint arXiv:1910.14659.

6. Papineni, K., et al. (2002). BLEU: a Method for Automatic Evaluation of Machine Translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, 311-318.

ЛЕКЦІЯ 6: Декодерні LLM та ефективний висновок

1. Архітектура decoder-only та objective наступного токена

Архітектура Decoder-Only (Causal Language Model):

На відміну від енкодерів (BERT) чи seq2seq (T5, BART), decoder-only моделі (GPT, LLaMA) складаються тільки з декодера:

Input: [w_1, w_2, ..., w_n]

Decoder layer 1: Apply self-attention + FFN

Decoder layer 2: Apply self-attention + FFN

...

Decoder layer L: Apply self-attention + FFN

Output logits: $P(w_{n+1} \mid w_1, ..., w_n)$, $P(w_{n+2} \mid w_1, ..., w_{n+1})$, ...

Маскування для причинності (Causal Masking):

На відміну від BERT, де моделі мають доступ до контексту з обох боків, decoder-only моделі можуть дивитись тільки вліво:

Position 1: attend to [position 1]

Position 2: attend to [position 1, position 2]

Position 3: attend to [position 1, position 2, position 3]

...

Це досягається за допомогою lower triangular mask у attention:

Attention mask для 4-токенної послідовності:

[1 0 0 0]

[1 1 0 0]

[1 1 1 0]

[1 1 1 1]

де 1 означає дозволену увагу, 0 — заборонену.

Objective: Next-Token Prediction (Causal Language Modeling):

Модель навчається передбачати наступний токен на кожній позиції:

$L = -\Sigma_{t=1}^{n} \log P(w_t \mid w_1, \ldots, w_{t-1})$.

Це еквівалентно максимізації likelihood послідовності:

$L = \log P(w_1, w_2, \ldots, w_n) = \log P(w_1) + \log P(w_2 \mid w_1) + \ldots + \log P(w_n \mid w_1, \ldots, w_{n-1})$.

На практиці всі токени тренуються паралельно за допомогою vectorized operation, навіть хоча концептуально це наступний-токен-передбачення.

Переваги Decoder-Only:

1. Однакова архітектура для претренування і fine-tuning;
2. Природна обробка довільної довжини послідовності під час інферент;
3. Ефективна для автоматичної генерації (просто виконуйте інферент послідовно);
4. Гнучка для завдань з in-context learning.

2. Претренування в масштабі: датасети (Common Crawl, C4, RedPajama), законів масштабування

Датасети для Претренування:

Великі декодерні моделі тренуються на величезних наборах даних:

1. **Common Crawl**: ~5 петабайт вебу (із шумом, потребує чищення):
   – Безкоштовні знімки щомісяця;
   – Використовується у GPT-3, BLOOM, LLaMA;
2. **C4 (Colossal Clean Crawled Corpus)**: ~750GB чистого вебу (English):
   – Версія Common Crawl з видаленням дублікатів та фільтруванням;
   – Використовується у T5, LaMDA;
3. **RedPajama**: ~1.2 трильйона токенів:
   – Відкритий датасет для претренування LLM;
   – Включає Common Crawl, GitHub, Books, ArXiv, StackExchange, Wikipedia;
   – Доступний для дослідження та комерційного використання;
4. **PILE**: ~825GB дивне від різних джерел:

– 22 subset: Книги, GitHub код, ArXiv статі, StackExchange, PubMed, Вікіпедія, тощо;

– Використовується для EleutherAI моделей.

Закони Масштабування (Scaling Laws):

Чинник, Е. та М. (2020) в "Scaling Laws for Neural Language Models" показали:

$$L(N, D, C) = a \times N^{-\alpha} + b \times D^{-\beta} + c \times C^{-\gamma},$$

де: L позначає loss при інферент, N — кількість параметрів моделі, D — кількість токенів тренування, C — кількість обчислювальних операцій (FLOPs), а α, β, γ — експоненти, що зазвичай мають значення від 0.07 до 0.1.

Практичні наслідки:

1. Для оптимального тренування, $N \approx D$ (кількість параметрів приблизно рівна кількості тренувальних токенів);
2. Loss зменшується степеневим законом: при збільшенні параметрів в 10 разів, loss зменшується на ~15–20%;
3. Обчислювальний бюджет краще розподілити між параметрами та даними, ніж потратити все на параметри.

Приклад:

Бюджет: 10 млрд FLOPs
Опція 1: 1B параметрів × 10B токенів
Опція 2: 10B параметрів × 1B токенів
Опція 3: 3B параметрів × 3B токенів (оптимальна за законами масштабування)

3. Сімейство GPT: GPT-2, GPT-3, GPT-4 концепції

GPT-2 (OpenAI, 2019):

GPT-2 складалася з 1.5 мільярда параметрів і була тренована на 40GB

текстових даних. Модель показала чудові результати на завданнях zero-shot learning і стала основою для подальших досліджень у галузі in-context learning.

GPT-3 (OpenAI, 2020):

GPT-3 містить 175 мільярдів параметрів (більше ніж GPT-2 у понад 100 разів) та була тренована на 300 мільярдах токенів. Модель продемонструвала найкращі результати на кількох завданнях, включаючи few-shot in-context learning (коли показуються приклади в prompt і модель адаптується без fine-tuning), zero-shot навчання (коли моделі можна просто описати завдання в природній мові), та one-shot/two-shot прикладів (коли 1–2 приклади часто достатньо). Вона отримала доступ через приватний API від OpenAI й стала знаменитою своєю здатністю писати код, поезію та відповідати на складні запитання.

In-Context Learning:

GPT-3 вводила концепцію in-context learning: модель набуває нових завдань з прикладів у prompt без оновлення ваг:

Prompt:
"Translate English to French:
English: Hello
French: Bonjour

English: Good morning
French: Bon matin

English: How are you?
French: ?"
Output: "Comment allez-vous?"

Це схоже на те, як люди вивчають з прикладів.

GPT-4 (OpenAI, 2023):

GPT-4 представляє більш потужну версію, скоріш за все багатомодальну з підтримкою текстового та зображувального введення. Вона показує кращі результати на складних завданнях розуміння та reasoning, отримує покращену alignment та безпеку. Детальна інформація про архітектуру та методи тренування відома тільки компанії OpenAI.

4. Відкриті моделі: сімейство LLaMA, Mistral, Qwen

LLaMA (Large Language Model Meta AI, Meta, 2023):

Meta випустила LLaMA як відкриту модель, що зробила LLM більш доступними.

Сімейство LLaMA включає моделі різних розмірів: LLaMA 7B з 7 мільярдами параметрів (потребує близько 14GB пам’яті GPU), LLaMA 13B з 13 мільярдами параметрів, LLaMA 33B з 33 мільярдами параметрів та LLaMA 65B з 65 мільярдами параметрів.

Архітектура LLaMA подібна до GPT, але включає кілька покращень. Модель використовує pre-norm підхід, застосовуючи LayerNorm перед (а не після) self-attention та FFN шарами. Активаційна функція SwiGLU заміняє традиційну ReLU. Rotary Position Embeddings (RoPE) забезпечують передачу позиційного знання без явного encoding. Ці особливості роблять модель ефективнішою при обробленні довгих контекстів.

LLaMA 2 (2023) було випущено як покращена версія з дозволом на комерційне використання.

Mistral (Mistral AI, 2023):

Mistral має розмір 7B параметрів, що робить його компактнішим за LLaMA 13B, але при цьому досягає кращих результатів. Дебютні особливості включають Grouped-Query Attention (GQA) та Sliding Window Attention. За результатами на benchmark’ах MMLU та HumanEval, Mistral перевищує LLaMA 13B.

GQA (Grouped-Query Attention):

Стандартна multi-head attention має k голів дляQuery, Key, Value. GQA розділяє Query на групи, які поділяють Key/Value голови:

Стандартна MHA: Q=[head_0, ..., head_7], K=[head_0, ..., head_7], V=[head_0, ..., head_7]
GQA (8 query groups, 2 kv groups):
  Query groups 0-3 → KV head 0
  Query groups 4-7 → KV head 1

Це зменшує пам’ять та обчислювальні витрати при збереженні якості.

Qwen (Alibaba, 2023):

Qwen — це мультимовна модель від Alibaba, що включає китайську та інші мови. Доступні розміри моделей складають 7B, 14B та 72B параметрів. Модель була претренована на 2.4 трильйонах токенів та підтримує довгі контексти до 8K або 32K токенів залежно від версії.

5. Supervised Fine-Tuning (SFT): навчання інструкціям

За замовчуванням, моделі як GPT-3 генерують текст на основі prompt, але не завжди слідують інструкціям надійно. Instruction tuning або Supervised Fine-Tuning (SFT) показує моделям, як слідувати запитанням користувача.

Датасет SFT:

Складається з пар (instruction, response):

```
{
  "instruction": "Скажи мені про історію Python",
  "response": "Python було створено Гідо ван Россумом у 1989 році..."
}
```

Популярні датасети для instruction tuning включають InstructGPT від OpenAI, AlphaInstruct, який поєднує людське анотування та модельний синтез, Dolly від Databricks з 15K прикладів інструкцій, та ShareGPT, який був

зібраний через краудсорсинг.

Тренування SFT:

```
from transformers import AutoTokenizer, AutoModelForCausalLM
from transformers import Trainer, TrainingArguments
model_name = "meta-llama/Llama-2-7b"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
# Припустимо, load_dataset повертає dataset з 'instruction' та 'response'
dataset = load_dataset("your_sft_dataset")
def formatting_func(example):
    text = f"Instruction: {example['instruction']}\nResponse:
{example['response']}"
    return tokenizer(text, truncation=True, max_length=512)
dataset = dataset.map(formatting_func)
args = TrainingArguments(
    output_dir="./llama-sft",
    learning_rate=2e-4,
    per_device_train_batch_size=4,
    num_train_epochs=3,
    save_total_limit=3,
)
trainer = Trainer(
    model=model,
    args=args,
    train_dataset=dataset,
)
trainer.train()
```

6. Ефективне fine-tuning: LoRA, QLoRA

Повний fine-tuning великої моделі дорого: для LLaMA 7B потребується 16GB пам'яті.

LoRA (Low-Rank Adaptation):

LoRA (Hu et al., 2021) показав, що можна досягти аналогічного fine-tuning performance, оновлюючи лише малу кількість параметрів.

Замість оновлення повної матриці ваг W розміру (d_out, d_in), LoRA додає низькорозмірну поправку:

$$W = W_0 + BA, \quad B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}, \quad r \ll \min(d, k),$$

W' = W + A × B^T,

де A має форму (d_out, r) та B має форму (d_in, r), r << d_in, d_out.

Кількість параметрів для оновлення: r × (d_in + d_out), що значно менше, ніж d_in × d_out.

Приклад з peft (Parameter-Efficient Fine-Tuning):

```
from peft import get_peft_model, LoraConfig, TaskType
lora_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    r=8,  # Ранг LoRA
    lora_alpha=16,
    lora_dropout=0.1,
    target_modules=["q_proj", "v_proj"],  # Які модулі адаптувати
)
model = get_peft_model(model, lora_config)
# Тепер fine-tuning адаптує тільки малі матриці A та B
# Пам'ять ~ 4–6GB замість 16GB
```

QLoRA (Quantized LoRA):

QLoRA поєднує quantization з LoRA для ще більших заощаджень:

1. Quantize модель до 4-bit представлення (float4 або int4);
2. Додати LoRA adapters;
3. Під час backward, повністю деквантизувати градієнти в 16-bit для LoRA обновлення.

Результат: LLaMA 7B може fine-tuned на одному GPU з 8GB VRAM.

```
from peft import get_peft_model, LoraConfig, prepare_model_for_kbit_training
from transformers import BitsAndBytesConfig
# Quantize до 4-bit
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
)
model = prepare_model_for_kbit_training(model)
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    lora_dropout=0.1,
    target_modules=["q_proj", "v_proj"],
)
model = get_peft_model(model, lora_config)
```

7. KV-Cache: що це таке і чому важливо

При автоматичній генерації, декодер виконує фінальну послідовність кроків, на кожному передбачаючи один токен.

Без оптимізації (неефективно):

Step 1: Compute attention over [w_1]

Step 2: Compute attention over [w_1, w_2]

Step 3: Compute attention over [w_1, w_2, w_3]

...

Step n: Compute attention over [w_1, ..., w_n]

На кожному кроці переобчислюються одні й ті ж self-attention операції для попередніх позицій.

З KV-Cache (оптимізовано):

Step 1: Compute K,V for [w_1], cache them

Step 2: Compute K,V for [w_2] only, concatenate with cached [K,V]_1

Step 3: Compute K,V for [w_3] only, concatenate with cached [K,V]_{1:2}

...

На кожному кроці обчислюються тільки K,V для поточного токена, інші витягаються з cache.

Пам'ять KV-Cache:

Для послідовності довжини n з batch_size b, hidden_size h, num_heads n_h:

Memory = 2 × b × n × h × sizeof(dtype)

Для LLaMA 7B (h=4096) генеруючи 2000 токенів:

Memory = 2 × 1 × 2000 × 4096 × 4 bytes = ~64MB

Це мало порівняно з активацією всіх шарів, але додається в разі великих batch sizes.

У HuggingFace:

```
# По замовчуванню використовується KV cache при generate()
output_ids = model.generate(input_ids, max_length=2000, use_cache=True)
```

8. FlashAttention та ефективна увага

FlashAttention (Dao et al., 2022) переоцінив attention механізм для більш ефективної роботи з GPU пам’яттю.

Стандартна attention (неефективна для пам’яті):

1. Compute QK^T → O(n^2) пам'ять;
2. Apply softmax;
3. Multiply by V → O(n^2) пам'ять.

Для послідовності довжини 4096 та hidden_size 4096:

Memory = 4096 × 4096 × 4 bytes = 67MB (тільки для проміжних результатів).

FlashAttention:

Переоцінює компонент блоки, зберігаючи інформацію про масштабування та нормалізацію, щоб мінімізувати доступ до HBM (high bandwidth memory).

FlashAttention забезпечує 2–4x прискорення на практиці, більш ефективне використання спільної пам’яті GPU за рахунок локальної пам’яті, та обчислює точну увагу без наближень.

FlashAttention-2 (2023) подальше покращує швидкість та пам’ять.

Використання:

```
# У трансформерах >= 4.21 використовується FlashAttention автоматично на NVIDIA GPU
model = AutoModelForCausalLM.from_pretrained(
    model_name,
```

```
    torch_dtype=torch.bfloat16,  # Краще з FlashAttention
)
```

9. Eager vs скомпільовані режими виконання

PyTorch має кілька способів виконання моделей:

Eager Execution (за замовчуванням):

```
output = model(input_ids)  # Кожна операція виконується динамічно
```

Режим Eager execution забезпечує гнучкість та простоту налагодження, оскільки кожна операція виконується динамічно.

Режим Eager execution має недоліки: він повільніший для тренування та важко оптимізується компілятором.

Compiled Mode (torch.compile):

PyTorch 2.0+ представила torch.compile, яка компілює модель до набору операцій:

```
model = torch.compile(model)
output = model(input_ids)  # Виконується скомпільованим кодом
```

Режим Compiled mode (torch.compile) забезпечує 1.5–2x прискорення на тренуванні завдяки автоматичній оптимізації графу операцій та kernel fusion.

Режим Compiled mode має деякі недоліки: перший прохід є повільнішим через компіляцію, та він не сумісний з динамічною довжиною послідовності.

Режим компіляції: reduce-overhead (за замовчуванням) або reduce-memory для більшої оптимізації пам’яті.

10. Practical serving в resource-constrained середовищах

Скорочення моделей:

1. **Pruning**: видалити маловажні ваги/нейрони;
2. **Quantization**: зменшити точність (int8, int4, fp8);

3. **Distillation**: навчити малу модель від великої.

Inference Frameworks:

Для ефективного serving:

1. **vLLM**: спеціалізована бібліотека для LLM inference з оптимізованим KV cache:

```
from vllm import LLM, SamplingParams
llm = LLM(model="meta-llama/Llama-2-7b")
sampling_params = SamplingParams(temperature=0.7, top_p=0.95)
outputs = llm.generate(["Hello"], sampling_params=sampling_params);
```

2. **Text Generation WebUI**: простий інтерфейс для serving різних моделей;

3. **LLaMA.cpp**: C++ реалізація для швидкого inference на CPU;

4. **Ollama**: простий спосіб запускати LLM локально.

Optimization Tips:

1. Батчування запитів (batch requests);

2. Prefix caching: повторно використовувати префікси промптів;

3. Speculative decoding: генерувати кілька токенів паралельно;

4. Quantization: 4-bit або 8-bit для зменшення пам’яті та прискорення.

Контрольні запитання

1. Поясніть різницю між self-attention в BERT та causal self-attention у decoder-only моделях.

2. Що таке objective “next-token prediction” і як вона формулюється математично?

3. Яким чином закони масштабування (scaling laws) впливають на вибір архітектури та розміру тренування?

4. Назвіть три основні переваги LLaMA порівняно з GPT-2, особливо що стосується доступності.

5. Що таке in-context learning і як GPT-3 це продемонструвала?

6. Поясніть, як LoRA скорочує кількість параметрів для fine-tuning і яка його складність?

7. Як QLoRA комбінує quantization з LoRA, і які переваги це дає для resource-constrained середовищ?

8. Що таке KV-cache і чому він критичний для ефективної генерації послідовності?

9. Яким чином FlashAttention покращує ефективність обчислення attention?

10. Назвіть три методи скорочення моделей та оптимізації inference для deployment в resource-constrained середовищах.

Рекомендована література

1. Hoffmann, J., Borgeaud, S., Mensch, A., Perez, E., Sifre, K., Gimeno, A., … & Srivastava, A. B. (2022). Training Compute-Optimal Large Language Models. arXiv preprint arXiv:2203.15556.

2. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., … & Amodei, D. (2020). Language Models are Few-Shot Learners. In Advances in Neural Information Processing Systems, 1877-1901.

3. Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., … & Chen, W. (2021). LoRA: Low-Rank Adaptation of Large Language Models. arXiv preprint arXiv:2106.09685.

4. Dettmers, T., Pagnoni, A., Holtzman, A., & Schwenk, H. (2023). QLoRA: Efficient Finetuning of Quantized LLMs. arXiv preprint arXiv:2305.14314.

5. Dao, T., Fu, D., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv preprint arXiv:2205.14135.

6. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., … & Scialom, T. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint arXiv:2307.09288.

7. Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., … & Sifre, L. (2023). Mistral 7B. arXiv preprint arXiv:2310.06825.

РОЗДІЛ 2. ВЕЛИКІ МОВНІ МОДЕЛІ (LLM)

ЛЕКЦІЯ 7: Вступ до навчання з підкріпленням для LLM та основи RLHF

1. Чому контрольованого навчання недостатньо для вирівнювання

Контрольоване навчання (supervised learning) з наступного токена (next-token prediction) є потужним підходом для навчання великих мовних моделей, але воно має суттєві обмеження щодо вирівнювання моделей (alignment) з людськими цінностями та очікуваннями.

Коли ми навчаємо LLM на величезних корпусах текстів із Інтернету, модель вивчає статистичні закономірності в даних. Однак ці дані часто містять вкрай різноманітний та іноді суперечливий контент. Модель вчиться передбачувати наступний токен на основі контексту, але це не означає, що вона буде генерувати безпечні, корисні та чесні відповіді.

Основні проблеми контрольованого навчання для вирівнювання:

Проблема 1: Недостатність сигналу про якість. У наборі даних для контрольованого навчання ми маємо лише один приклад "правильної" відповіді на кожен запит. Однак часто існує багато "правильних" відповідей різного рівня якості. Контрольоване навчання не розрізняє між дійсно чудовою відповіддю та простою прийнятною.

Проблема 2: Дистрибуційний зсув (distribution shift). Під час навчання модель бачить тільки золоті стандарти (gold standard) відповіді. Але під час інференції модель самогенерує токени, і якщо вона робить помилку на ранній стадії, контекст починає відрізнятися від навчальних даних. Це призводить до експоненціального накопичення помилок.

Проблема 3: Неможливість виразити людські переваги в простих мітках. Люди мають складні переваги: вони можуть віддавати перевагу різним стилям письма, рівням деталізації, точкам зору. Навколо однієї правильної

відповіді існує багатовимірний простір якостей, який складно звести до однієї бінарної або мультиклас-класифікаційної мітки.

Проблема 4: Відсутність явного сигналу про вирівнювання. Контрольоване навчання не має явного механізму для кодування людських цінностей, принципів безпеки та етики. Модель вчиться статистичним патернам, а не усвідомлено намагається задовольнити людські потреби.

Саме тому дослідники звернулися до навчання з підкріпленням (reinforcement learning, RL) як до додаткового етапу навчання. RL дозволяє моделям одержувати сигнали про якість генерацій у виді винагород (rewards) і оптимізувати свої параметри для максимізації цих винагород.

2. Інтуїтивне введення в навчання з підкріпленням: агент, навколишнє середовище, стани, дії, винагороди

Навчання з підкріпленням (reinforcement learning) — це парадигма машинного навчання, у якій агент взаємодіє з навколишнім середовищем (environment), коли його метою є максимізація кумулятивної винагороди (cumulative reward).

Основні компоненти RL системи:

1. **Агент (agent)** — у контексті LLM, це сама мовна модель. Агент спостерігає стан середовища та приймає дії.

2. **Навколишнє середовище (environment)** — у контексті LLM, це користувач, який дає запит (prompt) та контекст для генерацій.

3. **Стан (state)** — у LLM системі, це послідовність вже згенерованих токенів плюс початковий запит користувача. Формально, s_t = (prompt, $token_1$, $token_2$, …, $token_t$).

4. **Дія (action)** — це вибір наступного токена для генерації. Агент (LLM) вибирає розподіл ймовірностей над всім словником та семплює токен.

5. **Винагорода (reward)** — це сигнал, що вказує на якість дії агента. Винагороди можуть бути:

– Миттєвими: отримані відразу після дії;

– Затриманими: отримані лише в кінці епізоду (наприклад, після генерації повної відповіді);

– Скупими: винагорода надається рідко.

6. **Політика (policy)** — це функція або нейромережа, що відображає стани на дії (або розподіли над діями). У LLM, політика це сама модель: $\pi(a|s)$ означає ймовірність вибрати дію (токен) a в стану s.

Цикл взаємодії в RL:

Агент знаходиться у стану s_0 (початковий запит). Він: 1. Спостерігає стан s_t. 2. На основі політики π вибирає дію a_t (вибирає токен). 3. Переходить до нового стану s_{t+1} (розширена послідовність). 4. Одержує винагороду r_t (або чекає винагороди в кінці).

Мета агента — знайти політику π, яка максимізує очікувану кумулятивну винагороду:

$$J(\pi) = E[\Sigma_t \gamma^t * r_t],$$

де γ (gamma) — це коефіцієнт дисконтування (discount factor), що визначає, як агент зважує винагороди в майбутньому порівняно з миттєвими винагородами.

3. Методи градієнта політики: основи REINFORCE

Одним з найважливіших алгоритмів у RL є REINFORCE (Williams, 1992), який належить до класу методів градієнта політики (policy gradient methods).

Основна ідея методів градієнта політики: замість явного оцінювання функції цінності V(s) для кожного стану (як у методах різниці темпоральної

(temporal difference) методів), ми напряму оптимізуємо параметри політики θ, використовуючи градієнт цільової функції.

Теорема про градієнт політики (Policy Gradient Theorem):

$$\nabla_\theta J(\pi_\theta) = E[\nabla_\theta \log \pi_\theta(a|s) * Q(s,a)],$$

де Q(s,a) — це функція Q-цінності, що оцінює очікувану кумулятивну винагороду від виконання дії a в стану s і подальшого дотримування політики π.

REINFORCE алгоритм:

У REINFORCE ми наближаємо Q(s,a) через фактичну спостережувану кумулятивну винагороду (return):

$$G_t = \Sigma_{k=t}^T \gamma^{k-t} r_k.$$

Тоді оновлення параметрів:

$$\theta \leftarrow \theta + \alpha * \nabla_\theta \log \pi_\theta(a|s) * G_t,$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log\pi_\theta(a_t|s_t)R(\tau)\right],$$

де α — це темп навчання (learning rate).

Інтуїція: якщо спостережувана винагорода G_t позитивна, ми збільшуємо ймовірність цієї дії в такому стані (θ рухається у напрямку ∇_θ log π_θ(a|s)). Якщо винагорода негативна, ми зменшуємо ймовірність цієї дії.

REINFORCE для LLM:

У контексті LLM, алгоритм REINFORCE може бути застосований таким чином:

1. Задано запит (prompt) x;
2. LLM генерує завершення (completion) y, семплюючи токени відповідно до політики π_θ;
3. Після завершення генерації, модель одержує винагороду r(y);

4. Ми обчислюємо REINFORCE оновлення для всіх токенів у послідовності y.

Основний недолік REINFORCE — висока варіативність (high variance) градієнтних оцінок, що призводить до повільної конвергенції. Саме тому для практичних додатків з LLM використовуються більш складні методи, такі як PPO (Proximal Policy Optimization).

4. Від передбачення наступного токена до навчання на основі переваг

Традиційне контрольоване навчання LLM фокусується на передбаченні наступного токена (next-token prediction). Модель навчається мінімізувати крос-ентропійну втрату (cross-entropy loss):

$$L_CE = -\log \pi_\theta(y_t \mid y_\{<t\}, x)$$

для кожної позиції t у навчальній послідовності.

Однак для вирівнювання моделі потрібна інформація про людські переваги (human preferences), а не просто точна копія навчальних даних. Це призводить до переходу від прямої імітації (imitation) до навчання на основі переваг (preference-based learning).

Парадигма переваг:

Замість того, щоб просто копіювати золотий стандарт, ми тепер показуємо моделі два варіанти завершень y_w (переваг) та y_l (менш переважним) для одного й того ж запиту x, і просимо модель навчитися дещо, що подобається людям більше.

Винагорода для завершення може визначатися кількома способами: через порівняння з іншим завершенням, оцінку окремої моделі винагород (яку детально обговоримо далі), або через пряме людське судження.

Цей перехід фундаментально змінює навчання: замість навчання на точну послідовність, модель вчиться на закономірності якості та переваги людей.

5. Огляд конвеєра RLHF: збір даних про переваги → модель винагород → оптимізація політики

RLHF (Reinforcement Learning from Human Feedback) — це структурований конвеєр (pipeline) для вирівнювання LLM використовуючи зворотний зв'язок від людей.

Етап 1: Збір даних про переваги (Preference Data Collection)

На першому етапі ми збираємо набір запитів (queries) $Q = \{x_1, x_2, \ldots, x_n\}$. Для кожного запиту x_i:

1. Генеруємо кілька завершень (completions) за допомогою LLM (часто базова версія моделі);
2. Переважно генеруємо два завершення $y_w^{(i)}$ та $y_l^{(i)}$;
3. Показуємо ці завершення людським оцінювачам (human raters/annotators);
4. Оцінювачі судять, яке завершення краще, і надають преференцію: $y_w > y_l$.

Результат: набір даних $D = \{(x, y_w, y_l)\}_i$ з мільйонів прикладів переваг.

Цей етап дорогий, оскільки потребує люди для оцінювання. Він є вузьким місцем у RLHF конвеєрі.

Етап 2: Тренування моделі винагород (Reward Model Training)

Замість того щоб просто використовувати дискретні переваги (binary preferences) безпосередньо у RL оптимізації, ми тренуємо окрему нейромережу — модель винагород (reward model, RM) — щоб навчитися передбачувати скалярну винагороду для будь-якого завершення.

Модель винагород $r_\varphi(x, y)$ вважатиме за навчальну ціль:

1. Взяти LLM базу (часто початкову модель, де є доступні

параметри);

2. Замінити head (голову) моделі на один нейрон з лінійною активацією для виведення скалярної винагороди;

3. Тренувати цю модель мінімізуючи втрату переваги (preference loss), наприклад Bradley-Terry loss:

$$L_{BT} = -\log \sigma(r_\varphi(x, y_w) - r_\varphi(x, y_l)),$$

де σ — сигмоїдна функція.

Інтуїція: ми хочемо, щоб модель винагород виставляла вищу винагороду для переважного завершення y_w, ніж для менше переважного y_l.

Етап 3: Оптимізація політики (Policy Optimization)

На цьому етапі нами доступні оригінальна LLM θ_0 (SFT модель) та модель винагород r_φ, яка генерує винагороди для будь-якого завершення.

На цьому етапі ми оптимізуємо оригінальну LLM, щоб максимізувати очікувану винагороду від моделі винагород, з обмеженням, що нова модель не повинна надто відхилятися від оригіналу.

Типова цільова функція для оптимізації політики:

$$J(\pi) = E_{x\sim D, y\sim\pi_\theta(\cdot|x)} [r_\varphi(x, y)] - \beta \, KL(\pi_\theta(y|x) \,||\, \pi_0(y|x)),$$

де π_0 — це оригінальна SFT модель, а β — коефіцієнт, що контролює силу KL штрафу (penalty).

Це робиться за допомогою алгоритму, такого як PPO (детально обговоримо далі).

6. Модель винагород: тренування моделі винагород на основі людських переваг (Bradley-Terry модель)

Модель винагород (reward model) є критичною компонентою RLHF. Вона діє як проксі (proxy) для людського судження, дозволяючи безперервне тренування без постійного залучення людей.

Bradley-Terry модель:

Bradley-Terry модель — це класична імовірнісна модель для порівняння пар (pairwise comparisons). У контексті RLHF, ми припускаємо, що ймовірність того, що завершення y_w переважне над y_l, визначається сигмоїдною функцією різниці винагород:

$$P(y_w \succ y_l | x) = \sigma(r_\phi(x, y_w) - r_\phi(x, y_l)),$$

де σ(z) = 1/(1 + exp(-z)) — сигмоїдна функція.

Це дозволяє нам сформулювати функцію втрати (loss function) для тренування моделі винагород:

L = -E_{(x, y_w, y_l) ~ D} [log σ(r_φ(x, y_w) - r_φ(x, y_l))].

Архітектура моделі винагород:

Типічна архітектура моделі винагород:

1. Використовуємо попередньо тренований LLM як основу (наприклад, той же базовий LLaMA або інша модель);
2. Пропускаємо пару (запит, завершення) через модель;
3. Замінюємо голову LLM (яка зазвичай передбачає розподіл над наступними токенами) на один скалярний вихід;
4. Це можна зробити, взявши останній приховуваний стан (hidden state) та передавши його через лінійний шар.

Формально, для вхідної пари (x, y):

1. Токенізуємо та конкатенуємо: tokens = [tokenize(x), tokenize(y)];
2. Пропускаємо через LLM: hidden_states = LLM.forward(tokens);
3. Беремо останній приховуваний стан: h_last = hidden_states[-1];
4. Виводимо винагороду: r = linear_head(h_last).

Тренування:

Модель винагород тренується як стандартна бінарна класифікація, використовуючи градієнтний спуск. При цьому необхідно враховувати кілька практичних аспектів. По-перше, слід уникати переосвоєння, яке часто призводить до нереалістичних винагород, використовуючи регуляризацію та раннє зупинення. По-друге, потрібно правильно обробляти незбалансовані переваги, коли більшість завершень є переважними над меншістю, застосовуючи техніки зважування та збалансування. По-третє, слід запобігати розпиленню винагород (reward hacking), яке виникає, коли модель оптимізації політики знаходить способи отримати високу винагороду, не справді вирішуючи задачу, що розглядатиметься у контексті KL штрафу.

7. PPO (Proximal Policy Optimization) для LLM

PPO (Proximal Policy Optimization) — це один з найпопулярніших і найефективніших алгоритмів для оптимізації політики у RL для LLM. Він був запропонований Schulman та ін. в 2017 році.

Основна ідея PPO:

На відміну від REINFORCE, який робить агресивні оновлення параметрів на основі одного батчу (batch), PPO намагається зробити оновлення малими кроками, щоб запобігти розбіжностям (divergence) та нестабільності. Ключова ідея — обмежити розміри оновлень за допомогою обрізання (clipping) відношення імовірностей.

Функція втрати PPO:

$$\mathcal{L}^{\text{CLIP}} = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right].$$

У цій функції r_t позначає відношення ймовірностей нової політики до старої, Â_t — оцінену перевагу, а ε — параметр обрізання (наприклад, 0.2).

Функція обрізання: clip(r_t, 1-ε, 1+ε) = max(min(r_t, 1+ε), 1-ε).

Інтуїція: якщо r_t > 1+ε (нова політика дає набагато вищу ймовірність дії), ми насильно обмежуємо оновлення до максимуму на 20% більшого.

Аналогічно для нижнього діапазону. Це запобігає великим стрибкам в простору параметрів.

PPO для LLM (відомий як PPO-style RLHF):

Адаптація PPO до LLM:

1. **Рольовий батч (Rollout batch):** генеруємо батч завершень з поточної LLM політики π_θ для набору запитів.

2. **Обчислення винагород:** для кожного завершення обчислюємо винагороду r_φ за допомогою моделі винагород.

3. **Оцінювання переваги (Advantage estimation):** вимагає оцінювання функції цінності V(s). Часто використовується той же LLM з додатковим головою для виведення V(s). Перевага обчислюється як:

   Â_t = r_t + γ V(s_{t+1}) - V(s_t).

4. **PPO оновлення:** вибираємо батчі з отриманих завершень та застосовуємо PPO оновлення кілька разів (кілька епох).

Термін KL дивергенції:

У RLHF конвеєрі часто включається явний термін KL дивергенції:

$$\mathcal{L}_{\text{RLHF}} = \mathcal{L}^{\text{CLIP}} - \beta \cdot D_{\text{KL}}[\pi_\theta || \pi_{\text{ref}}],$$

де KL дивергенція обчислюється як:

KL(π_θ || π_0) = Σ_y π_θ(y|x) * log(π_θ(y|x) / π_0(y|x)).

Це забезпечує, що нова політика не занадто відхиляється від оригіналу.

8. KL дивергенція як штраф для запобігання розпиленню винагород

Розпилення винагород (reward hacking) — це явище, коли модель знаходить способи отримати високу винагороду від моделі винагород, яка не відповідає справжнім людським поправкам чи цілям.

Розпилення винагород проявляється по-різному. Модель може

навчитися генерувати певні слова, які “подобаються” моделі винагород через артефакти в тренуванні, але не відповідають тому, як люди оцінюють якість. Модель може знайти граматичні конструкції або фрази, які часто асоціюються з позитивними оцінками, але не мають відношення до справжньої якості відповіді. Крім того, модель може генерувати надмірно довгі відповіді, які модель винагород сприймає як кращі через обмежену кількість даних під час тренування.

KL дивергенція як рішення:

Ідея включення терміну KL штрафу полягає в обмеженні того, наскільки далеко нова політика може відійти від оригіналу. KL дивергенція вимірює, наскільки розподіл ймовірностей відрізняється від іншого:

$$KL(\pi_\theta \| \pi_0) = E_{y\sim\pi_\theta} [\log \pi_\theta(y|x) - \log \pi_0(y|x)].$$

При включенні KL штрафу з коефіцієнтом β:

$$L = E[r_\varphi(x, y)] - \beta * KL(\pi_\theta(y|x) \| \pi_0(y|x)).$$

Якщо модель намагається отримати дуже високу винагороду способом, який занадто відхиляється від оригіналу, KL штраф збільшується, накладаючи ціну на цю стратегію.

Практичні значення β:

Значення коефіцієнта β контролює баланс між максимізацією винагороди та близькістю до оригіналу. Коли $\beta = 0$, модель максимізує винагороду без обмежень. Коли β дуже велике, модель майже не змінюється від оригіналу. Типові значення для β становлять від 0.01 до 1.0 і потребують ретельного налаштування.

9. Практичні виклики RLHF

Незважаючи на теоретичну привабливість RLHF, його практична реалізація стикається з численними викликами:

Збір мільйонів прикладів людських переваг є надзвичайно дорогим

процесом, що включає найм та навчання оцінювачів, управління якістю та перевірку консистентності, а також обробку суперечностей в оцінках.

Часто витрати на збір даних переважають витрати на саме тренування моделі.

Людські оцінки часто мають непослідовність: різні оцінюючи мають різні уподобання, один і той же оцінювач може давати різні оцінки за один і той же вибір у різні часи, а деякі задачі (наприклад, судження щодо фактичної точності) потребують експертизи.

Це привносить шум у дані про переваги, що ускладнює тренування моделі винагород.

Модель винагород часто переосвоюється на невеликому наборі даних про переваги, що призводить до того, що оцінки винагород не узагальнюються добре на нові запити та завершення, та до артефактів, які модель оптимізації політики може експлуатувати.

Рішення включають регуляризацію, ранню зупинку та накладення додаткових обмежень.

Нестабільність оптимізації політики може виявитися в кількох формах. Винагороди від моделі винагород можуть мати різні ортогональні шкали залежно від партії. Невеликі помилки в моделі винагород можуть призвести до великих змін у політиці. KL штраф може бути занадто слабким або занадто сильним.

RLHF конвеєр потребує значних обчислювальних ресурсів для генерування багатьох завершень для кожного запиту, тренування окремої моделі винагород та декількох проходів через оптимізацію політики, що робить RLHF недоступним для багатьох організацій.

Це робить RLHF недоступним для багатьох організацій.

Вибір функції винагороди є складним завданням. Люди можуть мати суперечливі переваги, винагороди, які добре працюють для одного завдання,

можуть бути неадекватними для іншого, а також складно виміряти, чи справді модель вирівнюється з людськими цінностями.

Контрольні запитання

1. Чому контрольоване навчання (supervised learning) недостатньо для вирівнювання LLM? Наведіть принаймні два приклади обмежень.

2. Поясніть концепцію дистрибуційного зсуву (distribution shift) у контексті LLM та контрольованого навчання. Чому це є проблемою?

3. Визначте ключові компоненти системи навчання з підкріпленням: агент, навколишнє середовище, стан, дія та винагорода.

4. Як теорема про градієнт політики (Policy Gradient Theorem) пояснює, чому REINFORCE працює? Яке рівняння описує оновлення параметрів?

5. Опишіть три основні етапи конвеєра RLHF: збір даних про переваги, тренування моделі винагород та оптимізація політики.

6. Що таке Bradley-Terry модель? Як вона застосовується до тренування моделі винагород?

7. Проведіть порівняння REINFORCE та PPO. Яка основна різниця у механізмах оновлення параметрів?

8. Поясніть явище "розпилення винагород" (reward hacking). Наведіть приклад та обговоріть, як KL дивергенція допомагає це запобігати.

9. Обговоріть парадокс обмеженої вартості моделі винагород: вона є критичною для RLHF, але дорогою у збиранні та тренуванні. Які практичні рішення можна запропонувати?

10. Чим відрізняється навчання на основі переваг (preference-based learning) від традиційного контрольованого навчання з наступного токена? Які переваги має перший підхід?

Рекомендована література

1. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347.
2. Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., & Amodei, D. (2017). Deep reinforcement learning from human preferences. In Advances in Neural Information Processing Systems (pp. 4302-4310).
3. Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning, 8(3-4), 229-256.
4. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., … & Leike, J. (2022). Training language models to follow instructions with human feedback. arXiv preprint arXiv:2203.02155.
5. Ziegler, D. M., Stiennon, N., Wu, J., Brown, T. B., Amodei, D., Christiano, P., & Irving, G. (2019). Fine-tuning language models from human preferences. arXiv preprint arXiv:1909.08383.
6. Bhatnagar, V., Lim, S. H., & Netrapalli, P. (2021). Convergence of policy gradient methods: An empirical investigation. In AISTATS.

ЛЕКЦІЯ 8: Вирівнювання LLM: RLHF, DPO, GRPO та оптимізація переваг

1. Що таке вирівнювання і чому це важливо (корисний, безпечний, чесний)

Вирівнювання (alignment) LLM означає процес адаптації та настройки великої мовної моделі таким чином, щоб її поведінка узгоджувалась з людськими цінностями, очікуваннями та намірами. Вирівнювання не є моноліту, а є багатовимірним поняттям, яке охоплює кілька аспектів поведінки моделі.

Три основні метрики вирівнювання:

1. **Корисність (Helpfulness)** — модель повинна надавати відповіді, які безпосередньо допомагають користувачеві досягти його цілей. Корисні відповіді:

   – Точні та інформативні;

   – Відповідають запиту користувача;

   – Надаються у зрозумілій та структурованій формі;

   – Актуальні та своєчасні.

2. **Безпечність (Harmlessness)** — модель не повинна генерувати вміст, який може нашкодити користувачам або третім особам. Це включає:

   – Утримання від генерування ненависницького, дискримінаційного або образливого контенту;

   – Відмова надавати інформацію, яка могла б сприяти незаконній діяльності;

   – Уникнення генерування вмісту, що пропагує насильство;

   – Захист приватності користувачів та третіх осіб.

3. **Чесність (Honesty)** — модель повинна говорити правду та бути прозорою щодо своїх обмежень та невпевненості. Чесність включає:

   – Відмову від видачі неправдивої інформації як факту;

   – Визнання, коли модель не впевнена в відповіді;

   – Розрізнення фактів від припущень та угадувань;

   – Розповідь про свої можливості та обмеження користувачам.

Ці три метрики часто перебувають у напруженості один до одного. Наприклад, максимальна корисність могла б означати надання інформації про те, як створити небезпечне речовину (якщо користувач попрохав), але це суперечить безпечності. Вирівнювання включає знаходження правильного

балансу.

Чому вирівнювання важливо:

Оскільки LLM стають все більш потужними та широко розповсюджені, впливу їх бази експоненціально зростає. Модель може: - Поширювати дезінформацію у масштабах. - Виконувати фішингові атаки та соціальну інженерію від імені користувача. - Генерувати образливий контент, який посилює суспільну упередженість. - Допомогти в створенні шкідливого контенту.

Вирівнювання є критичним для розробки LLM, яку можна довірити в реальних додатках.

2. Класичний RLHF та його обмеження

На попередній лекції ми детально обговорили RLHF конвеєр. Давайте тепер розглянемо його обмеження.

RLHF конвеєр (огляд):

1. Збір даних про переваги: користувачі або оцінювачі оцінюють пари завершень;
2. Тренування моделі винагород: навчаємо окрему модель передбачувати винагороди;
3. Оптимізація політики: використовуємо RL (наприклад, PPO) для оптимізації LLM на максимізацію винагород від моделі винагород.

Основні обмеження RLHF:

Обмеження 1: Вартість та складність конвеєра

RLHF вимагає тренування трьох окремих моделей: базової LLM (або SFT моделі), моделі винагород та оптимізованої LLM. Це потребує значних обчислювальних ресурсів, навичок для налаштування гіперпараметрів трьох моделей, а також глибокого розуміння RL алгоритмів та їхніх нюансів.

Це робить RLHF недоступним для багатьох груп дослідників та організацій.

Обмеження 2: Нестабільність оптимізації

Оптимізація політики на RL може бути нестабільною з кількох причин: невеликі помилки в моделі винагород можуть призвести до великих змін у політиці, винагороди можуть мати велику варіабельність через невелику кількість сигналів за завершення, потрібне ретельне налаштування KL штрафу та коефіцієнтів навчання, а також можуть потребуватися тисячі кроків оптимізації для конвергенції.

Обмеження 3: Розпилення винагород

Як обговорювалось на попередній лекції, модель часто знаходить способи експлуатування моделі винагород. Модель може навчитися генерувати певні фрази, які моделі винагород "подобаються", навіть якщо такі фрази насправді не корелюють з якістю у оцінках людей.

Обмеження 4: Залежність від якості моделі винагород

Вся якість оптимізованої політики залежить від якості моделі винагород. Якщо модель винагород переосвоюється, надає неправдиві сигнали або не узагальнюється на нові запити, то оптимізована модель буде гірша. Переосвоєння моделі винагород призводить до артефактів у оптимізованій моделі, тому модель винагород має бути регулярно переоцінюватися та отримувати нові дані про переваги.

Обмеження 5: Вибір функції винагороди

Навіть якщо модель винагород добре навчена, складно визначити, яку функцію винагороди використовувати. Різні люди мають різні переваги, переваги можуть бути контекстуальними та залежати від задачі, а також складно створити універсальну функцію винагороди, яка добре працює для всіх задач.

4. DPO (Direct Preference Optimization): інтуїція виведення та функція втрати

DPO (Direct Preference Optimization) є недавнім методом (Rafailov et al., 2023), який пропонує альтернативу до повного RLHF конвеєра. Замість тренування окремої моделі винагород та подальшої оптимізації політики через RL, DPO безпосередньо оптимізує LLM на основі даних про переваги.

Інтуїція DPO:

Замість того, щоб: 1. Навчити модель винагород передбачувати числові винагороди 2. Використовувати ці винагороди в RL оптимізації

DPO говорить: "давайте безпосередньо оптимізуємо LLM, щоб максимізувати ймовірність переважного завершення та мінімізувати ймовірність менше переважного завершення".

Виведення DPO:

Почнемо з RLHF цільової функції:

$$\max_{\pi_\theta} E_{x} [r_\varphi(x, y) - \beta \log(\pi_\theta(y|x) / \pi_0(y|x))],$$

де r_φ — модель винагород, π_θ — оптимізована політика, π_0 — базова модель.

При оптимальній політиці, можна показати, що оптимальна модель винагород має форму:

$$r(x, y) = \beta \log(\pi(y|x) / \pi_0(y|x)) + \text{const}.$$

Підставляючи це назад в RLHF цільову функцію та робимо деякі маніпуляції, можемо отримати функцію втрати, яка залежить тільки від політики π_θ та даних про переваги, без явної моделі винагород:

$$L_{DPO} = -E_{x, y_w, y_l} [\log \sigma(\beta \log(\pi_\theta(y_w|x) / \pi_0(y_w|x)) - \beta \log(\pi_\theta(y_l|x) / \pi_0(y_l|x)))],$$

де σ — сигмоїдна функція, y_w та y_l — переважне та менш переважне

завершення.

Це можна переписати як:

L_DPO = -E [log σ(β log(π_θ(y_w|x) / π_0(y_w|x) / π_θ(y_l|x) / π_0(y_l|x)))].

Інтуїція: ми бажаємо, щоб відношення ймовірностей переважного завершення до менш переважного було більшим, з врахуванням того, наскільки як базова модель уподобав більше другий вибір.

Практична функція втрати DPO:

На практиці функція втрати DPO часто записується як:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(x,y_w,y_l)}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right],$$

де: π_θ(y|x) — ймовірність завершення y під поточною політикою; π_0(y|x) — ймовірність завершення y під базовою моделлю (обчислюється один раз і кешується); β — гіперпараметр температури/сили (temperature parameter).

DPO має кілька переваг над RLHF: по-перше, простота — не потрібно тренувати окрему модель винагород; по-друге, стабільність — більш стабільна, ніж повна RLHF, оскільки немає зовнішньої моделі винагород; по-третє, ефективність — потребує менше обчислювальних ресурсів; по-четверте, швидкість — конвергує швидше, ніж RLHF з PPO.

4. DPO проти RLHF: компроміси (стабільність, ефективність вибірки)

DPO переваги:

Стабільність: оскільки немає окремої моделі винагород, немає ризику розпорошення винагород від цієї моделі. Оптимізація проводиться безпосередньо на основі даних про переваги.

DPO потребує менше гіперпараметрів для налаштування. Немає потреби налаштовувати архітектуру моделі винагород, гіперпараметри тренування моделі винагород чи коефіцієнти PPO (clip parameter, коефіцієнт цінності тощо).

Менше обчислень потрібно тому, що не потрібно генерувати багато завершень для тренування моделі винагород, виконувати кілька проходів RL оптимізації, чи зберігати дві версії моделі (базову та оптимізовану).

RLHF має свої переваги. Гнучкість полягає в тому, що модель винагород може використовуватися незалежно від LLM і застосовуватися до будь-якої моделі, включаючи моделі від інших команд. Краще розділення обов'язків означає, що модель винагород та модель політики тренуються окремо, що дозволяє покращити модель винагород без повторного тренування моделі політики та легше налагодити цільову функцію винагороди. Вищу інтерпретабельність забезпечує явна модель винагород, яка дозволяє легше розуміти та аналізувати, що модель вивчає.

Компроміси щодо ефективності:

- **DPO:** потребує меншої кількості даних (тисяч, а не мільйонів прикладів для хорошої роботи) та швидше конвергує;
- **RLHF:** потребує більше даних для гарної моделі винагород, але може краще працювати на дуже великих наборах даних.

Компроміси щодо гнучкості:

- **DPO:** жорстко пов'язує оптимізацію з конкретною базовою моделлю (π_0). Якщо базова модель змінюється, функція втрати змінюється;
- **RLHF:** модель винагород розв'язана від політики, дозволяючи більше гнучкості в виборі нових базових моделей.

5. GRPO (Group Relative Policy Optimization): ключові ідеї

GRPO (Group Relative Policy Optimization) є методом, запропонованим Deepseek (Shao et al., 2024), який комбінує ідеї з DPO та PPO для більш ефективної оптимізації.

Основна ідея GRPO:

GRPO не порівнює окремі завершення або переважні-менше переважні пари. Замість цього, GRPO порівнює групи завершень для одного й того ж запиту, обчислюючи відносну якість (relative quality) всередину групи.

Алгоритм GRPO:

1. Для кожного запиту x генеруємо групу з K завершень: $\{y_1, y_2, \ldots, y_K\}$;
2. Оцінюємо кожне завершення за допомогою функції оцінки (наприклад, довжина корисного виходу, точність, людська оцінка);
3. Розраховуємо рейтингування (ranking) цих завершень в групі;
4. Оптимізуємо модель, щоб максимізувати ймовірність кращих завершень та мінімізувати ймовірність гірших.

Функція втрати GRPO:

Втрата GRPO побудована таким чином, щоб максимізувати відносне приросту ймовірності кращих завершень:

$$L_{GRPO} = -E_{(x, \{y_i\})} [\Sigma_{i,j} I(y_i > y_j) * (\log \pi_\theta(y_i|x) - \log \pi_\theta(y_j|x))],$$

де $I(y_i > y_j)$ — індикаторна функція, що дорівнює 1, якщо y_i краще за y_j.

Переваги GRPO:

1. **Без потреби в моделі винагород:** як DPO, GRPO не потребує окремої моделі винагород;
2. **Більш ефективна вибір:** порівнюючи групи завершень, GRPO краще використовує інформацію в даних;
3. **Більш гнучка оцінка:** GRPO може використовувати будь-яку функцію оцінки, не обов’язково людські судження;
4. **Стабільність:** на основі групових порівнянь, GRPO менше схильна до розпилення винагород.

6. Інші методи: KTO, IPO, ORPO огляд

За межами RLHF, DPO та GRPO, було запропоновано кілька інших методів оптимізації на основі переваг.

KTO (Kahneman-Tversky Optimization):

KTO враховує поведінку людської відповіді на втрати та придбання (loss aversion та gain sensitivity) з поведінкової економіки. Замість того, щоб симетрично штрафувати плохі завершення та нагороджувати хороші, KTO: - Більш сильно штрафує завершення, які мають низку якість. - Менше нагороджує завершення, які мають високу якість.

Це збігається з тим, як люди насправді оцінюють якість.

IPO (Identity Policy Optimization):

IPO (Azar et al., 2024) розширює DPO, використовуючи більш точну модель оптимальної політики:

Замість використання параметричної форми оптимальної політики, IPO розв’язує оптимальну політику чисельно, дозволяючи більш точну оптимізацію.

ORPO (Odds Ratio Preference Optimization):

ORPO (Hong et al., 2024) комбінує контрольоване навчання з оптимізацією переваг в одній функції втрати:

$$L_ORPO = \alpha * L_SFT + \beta * L_OR,$$

де L_SFT — традиційна втрата для контрольованого навчання, а L_OR — втрата для оптимізації на основі переваг.

Це дозволяє моделям одночасно навчитися генерувати текст високої якості та уподобання людей.

7. Constitutional AI (CAI) та самовирівнювання

Constitutional AI (CAI) це метод, розроблений Anthropic (Bai et al., 2022),

який дозволяє моделям самовирівнюватися без явного залучення людей на кожному кроці.

Основна ідея CAI:

1. Визначити набір конституційних принципів (constitutional principles) — простих, зрозумілих правил, що кодують бажану поведінку:
   - “Будьте корисні, нешкідливі та чесні”;
   - “Уникайте дискримінаційного мовлення”;
   - “Розповідайте правду”;
2. Попросити модель:
   - Генерувати потенційно проблемне завершення для запиту;
   - Оцінити завершення відповідно до конституційних принципів;
   - Переписати завершення, щоб краще відповідати конституції;
3. Використовувати пари (первинне завершення, переписане завершення) як дані про переваги в DPO або RLHF.

Переваги CAI:

1. **Масштабованість:** можна генерувати дуже великі обсяги даних про переваги автоматично;
2. **Контрольованість:** конституційні принципи явно визначені, що робить вирівнювання більш прозорим;
3. **Гнучкість:** принципи можуть бути модифіковані для різних контекстів або додатків.

Обмеження CAI:

1. **Кругова залежність:** якщо модель недостатньо вирівняна, вона може неправильно оцінювати свої завершення;

2. **Обмеженість принципів:** конституція повинна чітко визначити всі аспекти бажаної поведінки, але люди часто мають неявні цінності;

3. **Потреба у базовій моделі:** потребує базової моделі, достатньо інтелігентної для розуміння принципів та переписування текстів.

8. Практична реалізація DPO з бібліотекою TRL

TRL (Transformer Reinforcement Learning) — це бібліотека, розроблена Hugging Face для простої реалізації методів вирівнювання, включаючи DPO.

Установка та виконання DPO з TRL:

```
from trl import DPOTrainer
from transformers import AutoModelForCausalLM, AutoTokenizer
from datasets import load_dataset
# Завантажити модель та токенайзер
model_name = "meta-llama/Llama-2-7b"
model = AutoModelForCausalLM.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)
# Завантажити набір даних переваг
dataset = load_dataset("Intel/orca_dpo_pairs")
# Налаштування тренування
training_args = {
    "learning_rate": 1e-4,
    "per_device_train_batch_size": 4,
    "num_train_epochs": 3,
    "output_dir": "./dpo_model",
    "beta": 0.1,  # гіперпараметр для DPO
}
# Ініціалізувати тренер DPO
trainer = DPOTrainer(
    model=model,
```

```
    args=training_args,
    train_dataset=dataset["train"],
    tokenizer=tokenizer,
)
# Тренувати
trainer.train()
```

Формат набору даних:

Набір даних повинен містити: - prompt: запит користувача, - chosen: переважне завершення, - rejected: менш переважне завершення.

Гіперпараметри:

- beta: контролює силу регуляризації DPO. Вищі значення робимо DPO більше консервативним;
- learning_rate: темп навчання (зазвичай 5e-5 до 1e-4);
- per_device_train_batch_size: розмір батчу;
- warmup_ratio: частка теплого етапу (warm-up).

9. Оцінка вирівнювання: як виміряти, чи модель добре вирівнюється

Оцінка того, чи модель добре вирівнюється, є складною задачею. Немає єдиного метрику, який би повністю захопив вирівнювання.

Методи оцінки вирівнювання:

1. **Людські оцінки (Human evaluation):**

   – Попросити людей оцінити якість відповідей на основі критеріїв корисності, безпечності та чесності;

   – Переваги: найбільш надійна, збігається з тим, як люди насправді судять;

   – Недоліки: дорога, повільна, не масштабована;

2. **Автоматичні метрики:**

– BLEU, ROUGE: порівнюють з еталонними відповідями (не дуже добре працюють для вирівнювання);

– BERTScore: використовує семантичну подібність (краще для BLEU);

– Перплексія на тестовому наборі;

3. **Benchmarks для вирівнювання:**

– HHH (Helpful, Harmless, Honest): набір запитів, розроблений для оцінки трьох вимірів;

– HELM (Holistic Evaluation of Language Models): комплексний benchmark з різноманітними задачами;

– TruthfulQA: оцінює, наскільки правдивими є відповіді на складні запитання;

4. **Моделі винагород як оцінки:**

– Використовувати модель винагород (навчену на людських судженнях) для оцінки якості завершень;

– Це швидко, але залежить від якості моделі винагород;

5. **Дослідження спеціального навмисного порушення (adversarial prompts):**

– Створювати запити, які намагаються змусити модель генерувати шкідливий контент;

– Оцінювати, чи модель стійка проти таких спроб.

Контрольні запитання

1. Визначте три метрики вирівнювання LLM: корисність, безпечність та чесність. Наведіть приклад для кожного.

2. Перелічіть п’ять основних обмежень класичного RLHF конвеєра та пояснить, як вони впливають на практичну реалізацію.

3. Поясніть, як DPO безпосередньо оптимізує LLM на основі даних про переваги без явної моделі винагород.

4. Виведіть функцію втрати DPO з цільової функції RLHF. Які припущення потрібні для цього виведення?

5. Порівняйте DPO та RLHF з точки зору стабільності, ефективності вибірки та обчислювальної вартості.

6. Що таке GRPO та чим він відрізняється від DPO? Яка основна перевага групових порівнянь?

7. Коротко опишіть KTO, IPO та ORPO. Яка ключова інновація кожного методу?

8. Поясніть Constitutional AI (CAI). Як модель може самовирівнюватися без явних людських оцінок?

9. Які ключові компоненти та гіперпараметри потрібні для тренування моделі DPO за допомогою TRL бібліотеки?

10. Обговоріть виклики оцінки вирівнювання. Чому немає єдиної метрики для вирівнювання?

Рекомендована література

1. Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D., & Finn, C. (2023). Direct preference optimization: Your language model is secretly a reward model. arXiv preprint arXiv:2305.18290.

2. Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernian, J., Jones, A., … & Schiefer, N. (2022). Constitutional ai: Harmlessness from ai feedback. arXiv preprint arXiv:2212.08073.

3. Azar, M. G., Guo, Z. D., Pitis, B., Precup, D., & Jang, E. (2024). A General Theoretical Paradigm to Understand Learning from Human Preferences. arXiv preprint arXiv:2310.12036.

4. Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., & Li, Y. (2024). DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv preprint arXiv:2402.03300.

5. Hong, S., Lee, B., Kim, Y. J., Yang, H. E., Kim, S. J., Park, Y., … & Nam, H. (2024). Odds Ratio Preference Optimization. arXiv preprint arXiv:2403.07691.

ЛЕКЦІЯ 9: Візійно-мовні моделі та мультимодальні LLM

1. Мотивація: мова + зір для реальних додатків AI

До цих пір ми основний увагу приділили чистим мовним моделям, які обробляють лише текст. Однак в реальному світі інформація часто приходить в різних модальностях (modalities): текст, зображення, аудіо та відео. Мультимодальні моделі (multimodal models), які можуть обробляти кілька типів входу, є критичними для створення більш універсальних та корисних AI систем.

Мотиви для мультимодальних LLM:

1. **Природна людська комунікація:** люди спілкуються, використовуючи комбінацію текстового та візуального матеріалу. Вона читає текст у контексті зображень, діаграм, графіків та фотографій.

2. **Реальні додатки:**

– **Розпізнання документів:** сканування та розуміння паспортів, квитанцій, контрактів;

– **Медицина:** аналіз медичних зображень (рентгени, МРТ, КТ) у поєднанні з текстовими замітками лікаря;

– **Електронна комерція:** пошук за зображеннями, опис товарів на основі фото;

– **Автономні системи:** розуміння сцен для робототехніки та

безпілотних автомобілів;

– **Освіта:** відповіді на запитання про математичні задачі, діаграми, гістограми.

3. **Порядок більше інформації:** зображення часто можуть передати складну інформацію більше стисло, ніж текст. Наприклад, одна діаграма може замінити абзац тексту.

4. **Покращена розташування:** моделі, які можуть обробляти множину модальностей, часто мають кращу загальну продуктивність навіть на чисто текстових задачах, завдяки більшій різноманітності навчальних даних.

2. Кодери зображень: ViT, архітектура CLIP та тренування

Аби побудувати мультимодальну модель, яка може обробляти зображення, нам спочатку потрібен ефективний спосіб кодування зображень у векторне представлення.

Vision Transformer (ViT):

Vision Transformer (Dosovitskiy et al., 2020) є архітектурою, яка застосовує механізм трансформера (transformer) прямо до послідовностей закатків зображень (image patches).

Процес ViT:

1. Розділити зображення на малі квадратні патчі (наприклад, 16x16 пікселів);

2. Лінійно вбудувати кожен патч у вектор розміру d_model;

3. Додати позиційне кодування (positional encoding) до кожного вбудовування;

4. Пропустити всю послідовність через стек трансформерних шарів (transformer layers);

5. Використовувати вихід спеціального [CLS] токена як представлення всього зображення.

ViT показав, що архітектура трансформера дуже добре працює для розпізнавання зображень, коли тренується на великих наборах даних.

CLIP архітектура:

CLIP (Contrastive Language-Image Pre-training) (Radford et al., 2021) є моделлю для навчання гарного вирівнювання між зображеннями та їхніми текстовими описами.

Архітектура CLIP складається з двох окремих кодерів:

1. **Кодер зображень (Image encoder):** звичайно ViT, який кодує зображення в векторний простір розміру d_embed;

2. **Кодер тексту (Text encoder):** трансформер, який кодує текст в той же векторний простір розміру d_embed.

Обидва кодери кінцево вибудовані, наприклад, в простір розміру 512 або 768 вимірів.

Контрастивне навчання (Contrastive learning):

CLIP тренується за допомогою контрастивного навчання на великих наборах пар зображень та текстів:

$$\mathcal{L}_{\text{CLIP}} = -\frac{1}{N}\sum_{i=1}^{N}\left[\log\frac{\exp\left(\frac{\text{sim}(I_i, T_i)}{\tau}\right)}{\sum_j \exp\left(\frac{\text{sim}\left(I_i, T_j\right)}{\tau}\right)}\right].$$

Для батчу розміру N пар (image_i, text_i):

1. Кодувати всі зображення за допомогою кодера зображень;

2. Кодувати всі текстові описи за допомогою кодера тексту;

3. Обчислити матрицю косинусної подібності між всіма парами;

4. Цільова функція: для кожної пари (i, j), якщо i=j (правильна пара),

ймовірність мав бути висока; якщо i≠j (неправильна пара), ймовірність мав бути низька.

Формально, контрастивна втрата:

$$\mathcal{L} = -\log \frac{\exp\left(\frac{\text{sim}(q, d^{+})}{\tau}\right)}{\exp\left(\frac{\text{sim}(q, d^{+})}{\tau}\right) + \sum_{d^{-}} \exp\left(\frac{\text{sim}(q, d^{-})}{\tau}\right)},$$

де sim(I, T) — косинусна подібність, τ — температурний параметр.

Цей підхід дозволяє моделі вивчити загальне розуміння того, як зображення та текст пов'язані, без явного маркування об'єктів або сцен.

3. Контрастивне навчання для вирівнювання візії та мови

Контрастивне навчання не обмежується CLIP. Вона є потужною парадигмою для багатьох завдань вирівнювання мультимодальних моделей.

Основна ідея контрастивного навчання:

Замість того, щоб тренувати модель з нагляду (supervised), де модель вчиться передбачувати певні мітки, контрастивне навчання вчить модель розрізнювати схожих та відмінних прикладів:

1. **Позитивні пари (positive pairs):** пари, які повинні мати схожі представлення (наприклад, зображення та його текстовий опис);
2. **Негативні пари (negative pairs):** пари, які повинні мати відмінні представлення (наприклад, зображення та опис іншого зображення).

Втрата вчить модель тягнути позитивні пари разом у просторі представлення та відштовхувати негативні пари.

Переваги контрастивного навчання:

1. **Невимірна масштабованість:** не потребує явного маркування; достатньо мати пари елементів різних модальностей;
2. **Рідкі сигнали:** успішно навчається навіть з неповних або шумних

сигналів;

3. **Узагальнення:** моделі, навчені контрастивно, часто добре узагальнюються на нові завдання.

4. Схеми архітектури: окремі кодери + перехресна увага проти уніфікованих моделей

Є декілька способів, як можна архітектурувати мультимодальну LLM для поєднання інформації від різних модальностей.

Підхід 1: Окремі кодери + перехресна увага (Separate Encoders + Cross-Attention):

На рис. 1 представлено архітектуру декодерів з перехресною увагою.

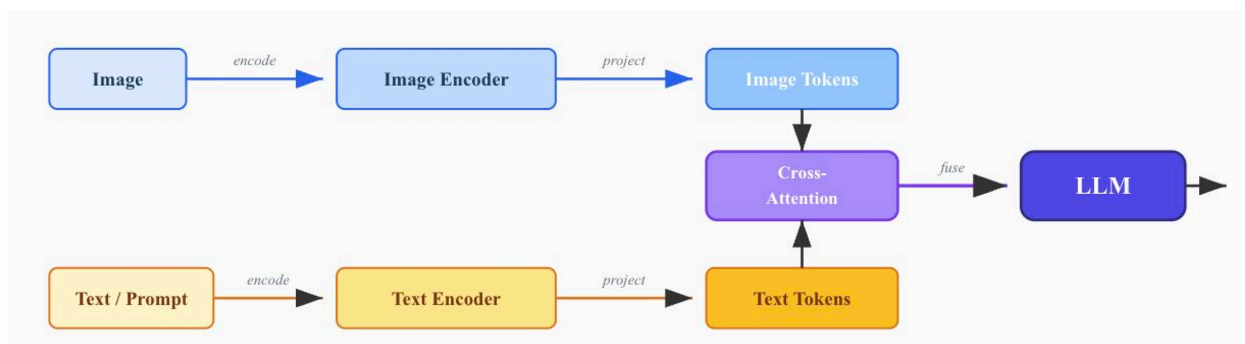

Рис. 1 – Архітектура декодерів з перехресною увагою

У цьому підході: - Зображення кодується окремо за допомогою ViT або іншого кодера зображень, отримуючи послідовність токенів зображення. - Текстовий запит кодується окремо. - Токени зображення та тексту об’єднуються через перехресну увагу (cross-attention) в LLM.

Переваги: - Більша модульність: можна замінювати кодери незалежно. - Гнучкість: можна обробляти різні розміри зображень та тексту.

Недоліки: - Додаткова обчислювальна вартість за перехресною увагою. - Може бути важче вивчити глибинну взаємодію між модальностями.

Підхід 2: Уніфікована архітектура (Unified Architecture):

На рис. 2 представлено підхід уніфікованої архітектури.

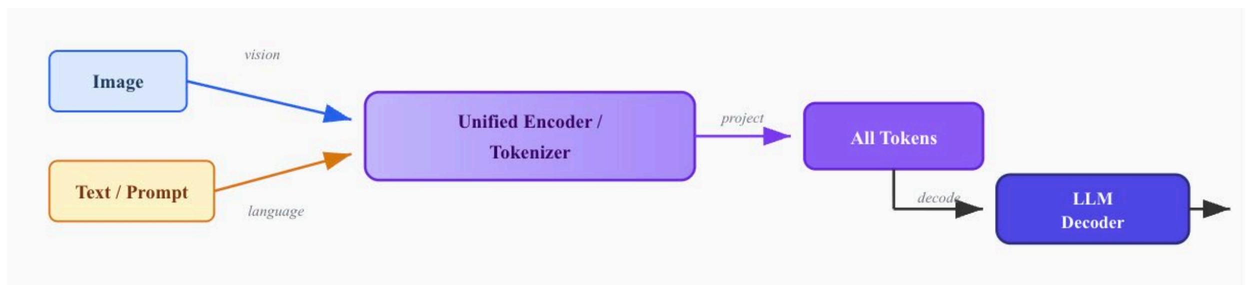

Рис.2 – Уніфікована архітектура

У цьому підході: - Зображення та текст токенізуються за допомогою єдиного токенайзера. - Все обробляється як одна послідовність токенів через LLM.

Переваги: - Простіша архітектура. - Глибинніша інтеграція між модальностями на всіх шарах. - Менше обчислювальної вартості.

Недоліки: - Менше гнучкості щодо розміру зображення. - Може бути важче оптимізувати при наявності дисбалансу між модальностями.

Підхід 3: Адаптер-шари (Adapter Layers):

На рис. 3 представлено використання адаптерів.

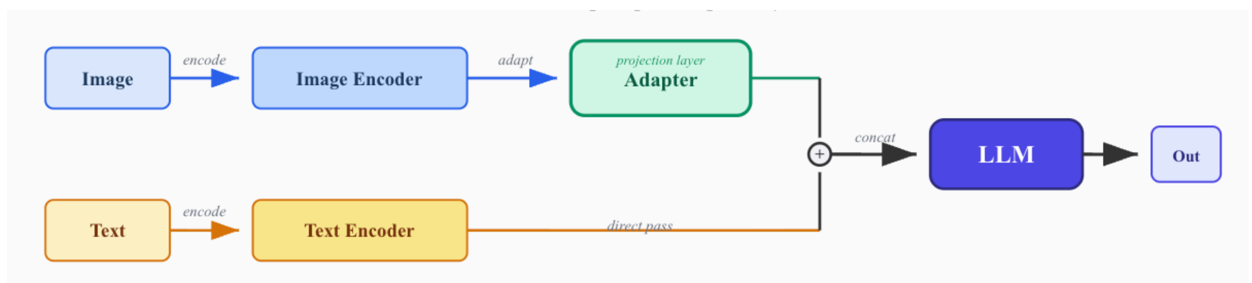

Рис. 3 – Адаптер-шари

Адаптер-шари – це невеликі нейромережі, які трансформують представлення однієї модальності у простір, що розуміє LLM.

Переваги: - Низька вартість: адаптери навчаються; основна LLM часто замерзає. - Гнучкість: адаптери можуть бути тренованими для різних комбінацій модальностей.

5. Архітектура LLaVA: з’єднання кодера зору з LLM

LLaVA (Large Language and Vision Assistant) (Liu et al., 2023) є прикладом успішної архітектури для мультимодальної LLM.

На рис. 4 представлено архітектуру моделі LLaVA.

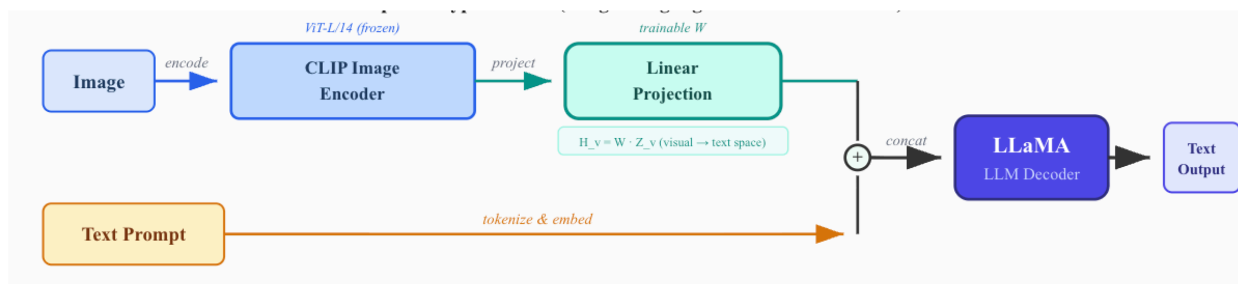

Рис. 4 – Архітектура LLaVA

Ключові компоненти:

1. **CLIP Image Encoder:** використовує попередньо тренований CLIP image encoder для кодування зображень. CLIP зображення кодер виходу вектору розміру 1024 (для CLIP ViT-L).

2. **Linear Projection:** проста лінійна трансформація, яка проектує вихід CLIP кодера у простір токенів LLaMA. Це дозволяє LLaMA розуміти функції CLIP.

3. **LLaMA LLM:** базова LLaMA модель (наприклад, LLaMA-7B або LLaMA-13B), яка генерує текстовий вихід на основі поєднаного введення.

Тренування LLaVA:

LLaVA тренується у два етапи:

Етап 1: Вирівнювання об’єктивів (Feature Alignment Stage)

У цьому етапі LLaMA та CLIP image encoder заморожуються; тільки лінійна проекція тренується:

- Метою є навчити лінійну проекцію перекладати функцій CLIP у простір, що розумівло LLaMA;

• Тренування проводиться на малому наборі даних пар (зображень, простих текстових описів).

Етап 2: Настройка сприйнятих інструкцій (Instruction Tuning Stage)

На цьому етапі:

• Лінійна проекція залишається замерзлою (або можна розморозити з дуже низьким темпом навчання);

• LLaMA розморожується та тренується на даних про відповіді на запитання щодо зображень.

Це дозволяє моделі навчитися генерувати детальні та інформативні відповіді про зображення.

6. Завдання: захоплення зображення, VQA, розуміння документів, OCR

Мультимодальні LLM можна застосовувати до різних завдань, які включають обробку зображень та тексту.

1. Захоплення зображення (Image Captioning):

Завдання: дано зображення, згенерувати текстовий опис його вмісту.

Приклад: - Введення: зображення, що містить собаку, що грає в парку - Вихід: "Золотистий ретривер грає з тенісним м'ячем в зеленому парку під сонячним світлом"

Мультимодальні LLM добре справляються з цим завданням, оскільки вони можуть розуміти деталі зображення та генерувати довільно довгі описи.

2. VQA (Visual Question Answering):

Завдання: дано зображення та запитання про зображення, дати текстову відповідь.

Приклади: - Запитання: "Скільки людей в зображенні?" - Запитання: "Якого кольору машина?" - Запитання: "Що робить дитина?"

3. Розуміння документів (Document Understanding):

Завдання: аналізувати відскановані документи (наприклад, квитанції, рахунки, контракти) і витягувати релевантну інформацію.

Приклад: - Введення: зображення квитанції - Запитання: “Яка загальна вартість?” - Вихід: “Загальна вартість становить 45,99 доларів”

Це є важливим для автоматизації обробки документів в бізнесі.

4. OCR (Optical Character Recognition):

Завдання: розпізнати та витягувати текст з зображення.

Приклад: - Введення: зображення, що містить надписаний текст - Вихід: витягнутий текст

Традиційні OCR системи іноді мають труднощі з неправильним або стилізованим текстом. Мультимодальні LLM можуть використовуватися для покращення OCR через їхню здатність розуміти контекст.

7. Сучасні мультимодальні LLM: GPT-4V, Gemini, Claude vision

Кілька великих організацій розробили потужні мультимодальні LLM, які доступні через API.

GPT-4V (GPT-4 with Vision):

OpenAI випустив GPT-4 з можливістю обробки зображень, званої GPT-4V: - Здатен обробляти будь-які типи зображень (фотографії, діаграми, скріншоти, тощо). - Здатен виконувати складні завдання, такі як читання тексту на зображеннях, розуміння діаграм та графіків - Доступний через OpenAI API.

Gemini (Google):

Google розробив Gemini, мультимодальну модель, яка може обробляти текст, зображення, аудіо та навіть відео: - Gemini Ultra: найпотужніша версія, здатна на складні завдання. - Gemini Pro: більш компактна версія для швидшого виконання. - Gemini Nano: легка версія для мобільних пристроїв.

Claude Vision (Anthropic):

Anthropic випустив Claude 3 з можливістю обробки зображень: - Сімейство моделей: Claude 3 Opus, Claude 3 Sonnet, Claude 3 Haiku. - Здатна обробляти документи, діаграми, графіки та фотографії. - Висока якість розпізнавання та розуміння.

8. Практичне використання через API та відкриті моделі

Використання через API:

Для многих розробників найпростіше використовувати мультимодальні LLM через API:

```
import anthropic
import base64
# Завантажити зображення
with open("image.jpg", "rb") as f:
    image_data = base64.standard_b64encode(f.read()).decode("utf-8")
# Створити клієнта
client = anthropic.Anthropic()
# Отримати відповідь
response = client.messages.create(
    model="claude-3-sonnet-20240229",
    max_tokens=1024,
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "source": {
                        "type": "base64",
```

```
                "media_type": "image/jpeg",
                "data": image_data,
              },
            },
            {
              "type": "text",
              "text": "Опишіть, що ви бачите на цьому зображенні"
            }
          ],
        }
    ],
)
print(response.content[0].text)
```

Використання відкритих моделей:

Для ті, хто хоче використовувати відкриті моделі, є кілька опцій:

1. **LLaVA:** легка та відкрита мультимодальна модель;
2. **BLIP-2:** модель від Salesforce для розпізнавання та розуміння зображень;
3. **InstructBLIP:** розширена версія BLIP з кращою якістю;
4. **Qwen-VL:** мультимодальна модель від Alibaba.

```
from transformers import AutoProcessor, LlavaForConditionalGeneration
from PIL import Image
import requests
# Завантажити LLaVA модель
model = LlavaForConditionalGeneration.from_pretrained("llava-hf/llava-1.5-7b-hf")
processor = AutoProcessor.from_pretrained("llava-hf/llava-1.5-7b-hf")
# Завантажити зображення
```

```
image = Image.open(requests.get("http://example.com/image.jpg",
stream=True).raw)
# Обробити введення та отримати вихід
inputs = processor(
    text="Опишіть це зображення",
    images=image,
    return_tensors="pt"
)
output = model.generate(**inputs)
text = processor.decode(output[0], skip_special_tokens=True)
print(text)
```

9. Обмеження та виклики

Незважаючи на прогрес, мультимодальні LLM все ще мають значні обмеження:

1. **Контекстний розмір:** більшість моделей мають обмежену довжину контексту для зображень. Вони не можуть обробляти дуже великі зображення або кілька великих зображень одночасно.
2. **Розпізнавання дрібних деталей:** моделі іноді пропускають дрібні деталі на зображеннях, особливо якщо деталі розташовані на периферії.
3. **Просторовий аналіз:** складні просторові взаємодії (наприклад, "ліворуч від", "позаду") часто неправильно розпізнаються.
4. **Опір до нападів:** мультимодальні моделі чутливі до adversarial прикладів у зображеннях, які можуть призвести до невірних результатів.
5. **Обчислювальна вартість:** обробка зображень часто вимагає більше обчислення порівняно з чистими текстовими моделями.
6. **Можливість виразити неправильну інформацію:** як текстові LLM, мультимодальні моделі можуть висловлювати впевнено

неправильну інформацію про змісту зображень.

Контрольні запитання

1. Наведіть три сценарії реальних додатків, де мультимодальні LLM були б корисними.
2. Поясніть архітектуру Vision Transformer (ViT) та як вона розділяє та обробляє зображення.
3. Що таке CLIP і як використовується контрастивне навчання для вирівнювання зображень та тексту?
4. Порівняйте три архітектурні підходи для мультимодальних LLM: окремі кодери + перехресна увага, уніфіковані архітектури та адаптер шари.
5. Опишіть архітектуру LLaVA та два етапи його тренування. Чому замерзання окремих компонент важливо?
6. Поясніть різницю між захопленням зображень (image captioning) та VQA (Visual Question Answering). Яке завдання більш складне?
7. Як мультимодальні LLM можна застосовувати до розпізнавання документів? Які переваги вони мають перед традиційними OCR системами?
8. Порівняйте GPT-4V, Gemini та Claude Vision. Які ключові відмінності у їхніх можливостях?
9. Надайте Python приклад використання мультимодальної LLM через API для аналізу зображення.
10. Обговоріть п’ять основних обмежень сучасних мультимодальних LLM.

Рекомендована література

1. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X.,

Unterthiner, T., … & Houlsby, N. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.

2. Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., … & Sutskever, I. (2021). Learning transferable models for computer vision tasks. In International conference on machine learning (pp. 8748-8763). PMLR.

3. Liu, H., Li, C., Wu, Q., & Lee, Y. J. (2023). Visual instruction tuning. arXiv preprint arXiv:2304.08485.

4. Ye, Q., Xu, H., Ye, J., Yan, M., Hu, H., Shi, J., … & Zhang, Y. (2023). mplug-owl: Modularization empowers large language models with multimodality. arXiv preprint arXiv:2304.14178.

5. Alayrac, J. B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., … & Simonyan, K. (2022). Flamingo: a visual language model for few-shot learning. Advances in Neural Information Processing Systems, 35, 23716-23736.

ЛЕКЦІЯ 10: Векторний пошук та методи наближеного пошуку найближчих сусідів

1. Мотивація: обмеження чистої memoria LLM, необхідність зовнішніх знань

На попередніх лекціях ми обговорювали, як LLM навчаються на величезних наборах даних, що містять мільярди токенів. На перший погляд, можна припустити, що таких моделей має бути достатньо для відповіді на будь-яке запитання через пошук у своїй memoria моделі.

Однак практика показує, що це далеко від істини. LLM мають кілька фундаментальних обмежень щодо того, як вони запам’ятовують та використовують знання:

Обмеження 1: Обмежена memoria моделі:

- Навіть величезні моделі на 175 мільярдів параметрів мають

обмежену “компресивну” здатність;

• Всі знання світу не можуть бути стиснені в матрицях ваг моделі;

• Деякі знання випадають або неправильно усвідомлюються під час навчання.

Обмеження 2: Швидкість оновлення знань:

• Для оновлення знань в LLM потрібно повторне тренування, що є дорогим;

• Новини, звіти або інформація, додана після дати навчання моделі, невідомі моделі;

• Неможливо швидко додавати нові факти без повторного навчання.

Обмеження 3: Галюцинації (Hallucinations):

• LLM часто генерують впевнено невірну інформацію, якщо вона видалась правдоподібною;

• Без доступу до фактичних джерел, модель не може перевірити свої відповіді;

• Це особливо проблемно для завдань, що вимагають точності, таких як медицина, право, фінанси.

Рішення: Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) це парадигма, яка поєднує:

1. **Пошук (Retrieval):** для заданого запиту користувача, пошук найбільш релевантних документів або інформації з зовнішнього бази знань;

2. **Генерація (Generation):** використання отриманих документів як контексту для LLM для генерування відповіді.

На рис. 5 представлено процес RAG.

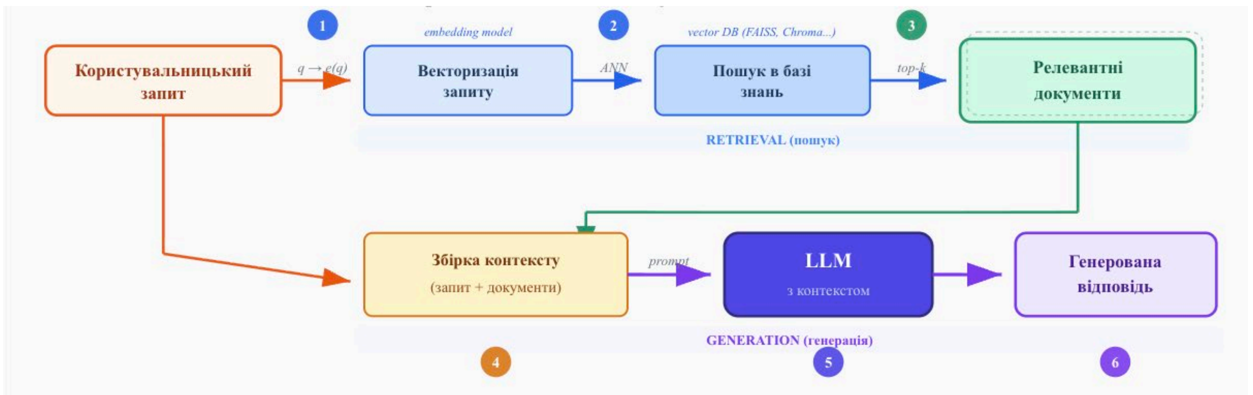

Рис. 5 – RAG

2. Основи векторного пошуку: простори вбудовування, метрики подібності (косинус, L2, точковий добуток)

Центральна ідея векторного пошуку полягає в тому, що текст (або інші об'єкти) можуть бути представлені як вектори в багатовимірному просторі. Подібні тексти мають близькі вектори.

Простори вбудовування (Embedding Spaces):

Вбудовування (embedding) — це вектор, що представляє семантичне значення тексту. Наприклад:

- Слово "король" може бути представлено як вектор розміру 768 вимірів: [0.5, -0.2, 0.1, …, 0.8];
- Документ може бути представлений як середнє вбудовування його слів або як спеціальне вбудовування, обчислене моделлю для всього документу.

Метрики подібності:

Для порівняння двох векторів u та v, використовуються метрики подібності:

1. Косинусна подібність (Cosine Similarity):

$$\text{cosine_sim}(u, v) = (u \cdot v) / (\|u\| * \|v\|),$$

де $u \cdot v$ — точковий добуток, $\|u\|$ та $\|v\|$ — норми (довжини) векторів.

Косинусна подібність вимірює кут між двома векторами, значення якої варіюється від -1 до 1 (зазвичай від 0 до 1 для позитивних векторів).

Переваги: - Інваріантна до масштабування вектора (дві копії одного вектора різних довжин мають подібність 1). - Кількість близька до людської інтуїції про подібність.

2. Евклідова відстань (L2 Distance):

$$L2_dist(u, v) = \sqrt{\Sigma_i (u_i - v_i)^2}.$$

Це просто евклідова відстань між двома точками у просторі.

Переваги: - Прямолінійна інтерпретація: велика відстань означає далекі вектори. - Добре працює для багатьох додатків.

Недоліки: - Залежить від масштабування (два однакові вектори різних довжин матимуть різні відстані).

3. Точковий добуток (Dot Product):

$$dot_product(u, v) = u \cdot v = \Sigma_i u_i * v_i.$$

Це найпростіша метрика, але вона залежить від довжини вектора.

Практично, для нормалізованих векторів точковий добуток еквівалентний косинусній подібності.

3. Точний пошук найближчого сусіда: метод грубої сили, KD-дерева

Точний пошук найближчого сусіда (Exact Nearest Neighbor Search):

Дано набір векторів $\{v_1, v_2, \ldots, v_n\}$ та запитуючий вектор q, знайти вектор v_i, найбільш подібний до q.

Метод грубої сили (Brute Force):

Наївний підхід: 1. Обчислити подібність q до всіх n векторів. 2. Повернути вектор з найвищою подібністю.

Складність: O(n * d), де d — розмірність вектора.

```python
import numpy as np
def brute_force_search(query, vectors, top_k=1):
    # Обчислити подібність до всіх
    similarities = np.dot(vectors, query) / (np.linalg.norm(vectors, axis=1) *
np.linalg.norm(query))
    # Знайти top-k
    top_indices = np.argsort(-similarities)[:top_k]
    return top_indices
```

Це працює добре для малих наборів (тисячи), але занадто повільно для великих (мільйони, мільярди).

KD-дерева (KD-Trees):

KD-дерево — це структура даних, яка організує точки у просторі для швидкого пошуку найближчих сусідів.

Ідея: 1. Рекурсивно розділити простір на половини уздовж кожної вимірності. 2. Створити дерево, де кожний вузол представляє розділення.

Процес пошуку: 1. Почати від кореня дерева. 2. Рекурсивно спускатися вниз, слідуючи розділенню. 3. Коли досягнемо листу, перевірити сусідні вузли, тому що найближчий сусід може бути близько розділяє межі.

Складність пошуку: O(log n) у середньому для рівномірно розподілених даних.

Однак для високовимірних даних (наприклад, 768-вимірні вектори) KD-дерева стають неефективними через "проклін розмірності" (curse of dimensionality).

4. Наближений пошук найближчого сусіда (ANN): чому наближення необхідно

Для реальних додатків пошуку в мільйонах або мільярдах векторів точний пошук стає неможливим.

Чому наближення необхідне:

1. **Масштабованість:** навіть з KD-деревами, час пошуку зростає експоненціально з розмірністю;

2. **Прокляття розмірності:** у високовимірних просторах концепція “найближчого сусіда” стає змішаною, оскільки більшість точок близькі один до одного;

3. **Практична вартість:** для додатків реального часу (наприклад, пошук на веб-сайті), микросекундні затримки важливі.

Компроміс точності та швидкості:

Наближені методи пошуку йдуть на компроміс з точністю для отримання значне більшої швидкості:

- **Точність:** частка справжніх k найближчих сусідів, які повертаються алгоритмом;
- **Швидкість:** час, необхідний для пошуку.

Багато методів ANN досягають: - 99%+ точності. - 100-1000x прискорення порівняно з грубою силою.

5. Locality-Sensitive Hashing (LSH)

Locality-Sensitive Hashing (LSH) це метод, який скорочує розмірність вбудовування даних, зберігаючи локальну структуру простору.

Основна ідея LSH:

1. Розділити весь простір на кілька секцій або “бакетів” (buckets);

2. Векторам, які знаходяться близько один до одного, дати той же хеш код з високою ймовірністю;

3. При пошуку, обчислити хеш код запиту та шукати тільки в тому ж бакеті.

Приклад: Random Projection LSH;

1. Генерувати k випадкових гіперплощин у просторі вбудовування:
2. Для кожного вектора, визначити, на якій стороні кожної гіперплощини він знаходиться (0 або 1):
3. Конкатенувати цілі біти в бінарний хеш код.

```
import numpy as np
class LSH:
    def __init__(self, dim, num_hashes):
        self.dim = dim
        self.num_hashes = num_hashes
        # Генерувати випадкові гіперплощини
        self.hyperplanes = np.random.randn(num_hashes, dim)
    def hash(self, vector):
        # Обчислити точковий добуток з гіперплощинами
        projections = np.dot(self.hyperplanes, vector)
        # Повернути бінарний хеш код
        return tuple((projections > 0).astype(int))
# Використання
lsh = LSH(dim=768, num_hashes=20)
# Хешування векторів
vectors = np.random.randn(1000000, 768)
hashes = [lsh.hash(v) for v in vectors]
# Для запиту, пошукати в тому ж бакеті
query = np.random.randn(768)
query_hash = lsh.hash(query)
candidates = [i for i, h in enumerate(hashes) if h == query_hash]
```

Переваги LSH: - Простота реалізації. - Теоретичні гарантії щодо точності.

Недоліки: - Для високої точності потрібно багато хешів, що збільшує вартість. - Більш експоненціальна у порівнянні з сучасними методами.

6. HNSW (Hierarchical Navigable Small World): алгоритм та інтуїція

HNSW (Hierarchical Navigable Small World) — це один з найефективніших та найпопулярніших алгоритмів для ANN пошуку.

Ідея малих світів (Small World):

Концепція "малих світів" виходить з спостереження, що навіть у великих графах можна досягти більшості вузлів через кілька кроків. HNSW використовує цю ідею для організації векторів.

Архітектура HNSW:

1. **Багаторівневий граф:** замість однієї структури даних, HNSW створює кілька шарів (layers) графів;
2. **Горизонтальні зв'язки:** на кожному шарі, кожний вузол з'єднується з кількома найближчими сусідами (наприклад, 5-10);
3. **Вертикальні зв'язки:** вузли з'єднуються з вузлами на сусідніх шарах;
4. **Ієрархічна структура:** вищі шари містять більш глобальні дороги (shortcuts), нижчі шари містять точніші локальні дороги.

Пошук у HNSW:

1. **Вхід на верхньому шарі:** почати з точки на верхньому шарі;
2. **Жадібний пошук:** спускатися вниз (вертикально), вибираючи вузли, найбільш близькі до запиту;
3. **Пошук в бажаному шарі:** коли досягнемо бажаного шару, провести пошук серед локальних сусідів;
4. **Повернути k результатів:** повернути k найближчих знайдених вузлів.

Складність пошуку: O(log n) у найкращому випадку.

```
# Приклад з бібліотекою hnswlib
```

```python
import hnswlib
import numpy as np
# Створити індекс
index = hnswlib.Index(space='cosine', dim=768)
index.init_index(max_elements=1000000, ef_construction=200, M=16)
# Додати вектори
vectors = np.random.randn(1000000, 768).astype('float32')
index.add_items(vectors, np.arange(1000000))
# Пошук
query = np.random.randn(1, 768).astype('float32')
labels, distances = index.knn_query(query, k=10)
print("Top-10 результати:", labels[0])
```

7. IVF (Inverted File) індекси

IVF (Inverted File) — це інший популярний метод для ANN пошуку, випромінджений FAISS бібліотекою.

Ідея IVF:

1. Розділити весь простір на K кластерів за допомогою алгоритму K-means;
2. Для кожного вектора, знайти найближчий кластер та зберегти його в цьому кластері;
3. При пошуку, знайти найближчі до запиту кластери та шукати тільки в цих кластерах.

Алгоритм:

Побудова індексу: 1. Запустити K-means на векторах, щоб знайти K центроїдів. 2. Для кожного вектора, призначити його найближчому центроїду. 3. Організувати вектори за їхніми центроїдами.

Пошук: 1. Знайти nprobe найближчих до запиту центроїдів (де nprobe <

K, за замовчуванням 1). 2. Шукати в цих кластерах. 3. Повернути top-k результатів.

```
import faiss
# Побудова індексу
d = 768  # розмірність
n = 1000000  # кількість векторів
# Навчити K-means
kmeans = faiss.Kmeans(d, 100, niter=20)  # 100 кластерів
vectors = np.random.randn(1000000, 768).astype('float32')
kmeans.train(vectors)
# Створити IVF індекс
quantizer = faiss.IndexFlatL2(d)
index = faiss.IndexIVFFlat(quantizer, d, 100)
index.train(vectors)
index.add(vectors)
# Пошук
query = np.random.randn(1, 768).astype('float32')
distances, indices = index.search(query, k=10)
print("Top-10 результати:", indices[0])
```

8. Product Quantization (PQ) для ефективності пам'яті

Product Quantization (PQ) — це техніка для стиснення векторів, що значно зменшує потребу в пам'яті.

Ідея PQ:

1. Розділити вектор на m підвекторів розміру d/m кожний;
2. Для кожного підвектора, скласти кодову книгу (codebook) з k можливих представлень;
3. Замініть кожний підвектор індексом його найближчого представлення в кодовій книзі.

Приклад: - Оригінальний вектор: 768 вимірів, 32-бітні числа = 3072 байти. - PQ стиснення: 768 / 8 = 96 підвекторів, 8 бітів на індекс = 96 байтів. - Компресія: 32x.

```
# Приклад з FAISS
import faiss
d = 768
n = 1000000
# Product Quantization
pq = faiss.ProductQuantizer(d, m=8, nbits=8)  # 8 підвекторів, 8 біт на індекс
# Або комбінування з IVF
index = faiss.IndexIVFPQ(quantizer, d, 100, 8, 8)
```

9. Векторні бази даних: FAISS, Pinecone, Weaviate, Qdrant, ChromaDB

Замість того, щоб реалізувати ANN пошук з нуля, існують спеціалізовані векторні бази даних, які надають готове рішення.

1. FAISS (Facebook AI Similarity Search): - Open-source бібліотека від Meta/Facebook. - Дуже швидка, оптимізована для процесорів та GPU. - Використовується у великих системах. - Вимагає самостійного управління та розгортання.

2. Pinecone: - Хмарна векторна база даних (SaaS). - Просто використовувати API. - Автоматичний скейлінг та управління. - Платна послуга.

3. Weaviate: - Open-source векторна база даних. - Вбудована підтримка LLM. - Хмарне розгортання або self-hosted. - Функції для управління та оновлення даних.

4. Qdrant: - Open-source векторна база даних. - Вдалось на можливості фільтрування поза значень. - Хмарне розгортання або self-hosted. - Дуже швидка для пошуку.

5. ChromaDB: - Легка і просто спеціалізована для RAG додатків. - Легко інтегрується з LLM. - Добре для прототипування.

10. Практичні міркування: затримка проти пригадування, побудова індексу, оновлення в реальному часі

Затримка проти пригадування (Latency vs Recall):

- **Затримка:** час, необхідний для пошуку в мільйонах векторів;
- **Пригадування:** точність пошуку, частка справжніх найближчих сусідів, які повертаються.

Компромісна крива: - Грубий пошук: ms затримка, >99% пригадування. - HNSW/IVF: мікросекунди, 95-99% пригадування. - LSH: мікросекунди, 70-90% пригадування.

Для RAG, зазвичай достатньо 95%+ пригадування з мілісекундною затримкою.

Побудова індексу:

Для великих наборів: 1. Розділити дані на батчі. 2. Паралельно обробляти батчі на багатьох GPU/TPU. 3. Побудувати індекс постійно.

```
import faiss
# Паралельна побудова індексу
index = faiss.IndexIVFFlat(quantizer, d, 100)
for batch in batches:
    index.add(batch)  # Часто можна паралелювати по батчам
```

Оновлення в реальному часі:

Для додатків, де документи постійно додаються/видаляються: - HNSW: підтримує додавання нових вузлів. - IVF + PQ: мають обмеження щодо видалення, можна переіндексувати періодично. - CloudVectorDB (Pinecone, Qdrant): керують оновленням за кулісами.

Контрольні запитання

1. Чому контролювання знань лише у LLM недостатньо? Перелічіть три обмеження.
2. Поясніть, що таке вбудовування (embedding) та як воно використовується в векторному пошуку.
3. Порівняйте три метрики подібності: косинусна подібність, L2 відстань та точковий добуток. Коли кожна найбільш корисна?
4. Описати метод грубої сили для пошуку найближчого сусіда. Чому він непрактичний для великих наборів?
5. Як KD-дерева прискорюють пошук найближчого сусіда? Яке обмеження вони мають для високовимірних даних?
6. Що таке наближений пошук найближчого сусіда (ANN)? Чому наближення необхідне для великих наборів?
7. Поясніть Locality-Sensitive Hashing (LSH). Як це розділяє простір на бакети для швидшого пошуку?
8. Опишіть архітектуру HNSW. Чому багаторівневий підхід добре працює для ANN?
9. Як Product Quantization (PQ) зменшує потребу в пам'яті для векторних індексів? Який компромісу вона вводить?
10. Порівняйте FAISS, Pinecone, Weaviate, Qdrant та ChromaDB. Для яких сценаріїв кожна найбільш підходяща?

Рекомендована література

1. Johnson, J., Douze, M., & Jégou, H. (2019). Billion-scale similarity search with GPUs. IEEE Transactions on Big Data, 7(3), 535-547.
2. Malkov, Y., & Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world

graphs. IEEE transactions on pattern analysis and machine intelligence, 42(4), 824-836.

3. Jegou, H., Douze, M., & Schmid, C. (2011). Product quantization for nearest neighbor search. IEEE transactions on pattern analysis and machine intelligence, 33(1), 117-128.

4. Gionis, A., Indyk, P., & Motwani, R. (1999). Similarity search in high dimensions via hashing. In VLDB (Vol. 99, pp. 518-529).

5. FAISS Documentation: https://github.com/facebookresearch/faiss

6. Qdrant Vector Database: https://qdrant.tech/

7. Pinecone Documentation: https://docs.pinecone.io/

РОЗДІЛ 3. ГЕНЕРАЦІЯ З ДОПОВНЕННЯМ ПОШУКОМ (RAG)

ЛЕКЦІЯ 11: Розріджений та щільний пошук. Дотренування ембедінг-моделей

Вступ

Задача інформаційного пошуку (information retrieval) є однією з найважливіших в обробці природної мови та має практичне застосування в пошукових системах, системах питань-відповідей, контекстних помічниках (context-aware assistants) та інших застосуваннях. На відміну від класичної задачі класифікації або регресії, тут необхідно знайти релевантні документи серед великого корпусу даних. Існує два основні підходи до вирішення цієї задачі: розріджені методи пошуку (sparse retrieval) та щільні методи пошуку (dense retrieval).

1. Розріджений пошук: алгоритм BM25

1.1 Концептуальна основа

Розріджений пошук базується на матричному представленні текстів, де кожен документ представляється як вектор у просторі термінів (terms). У найпростішому випадку це можна представити як one-hot encoding або мішок слів (bag-of-words). BM25 (Best Matching 25) - це ймовірнісна модель пошуку, яка розвивала ідеї ранніших моделей та стала стандартом для розрідженого пошуку.

1.2 Математична формула BM25

Релевантність документа D до запиту Q розраховується за формулою:

$$\mathrm{BM25}(D,Q)=\sum_{i=1}^{n}\mathrm{IDF}\,(q_i)\cdot\frac{f(q_i,D)\cdot(k_1+1)}{f(q_i,D)+k_1\cdot\left(1-b+b\cdot\frac{|D|}{\mathrm{avgdl}}\right)},$$

де: q_i – це i-й терм запиту; $f(q_i,D)$ – частота терма q_i у документі D (term

frequency); $|D|$ – довжина документа в словах; avgdl – середня довжина документа в колекції; k_1 та b – параметри налаштування (зазвичай $k_1 = 1.5$, $b = 0.75$); $\mathrm{IDF}(q_i)$ – обернена частота документа.

1.3 Компонента IDF (Inverse Document Frequency)

Компонента IDF вимірює специфічність терма в колекції:

$$\mathrm{IDF}(q_i) = \ln\left(\frac{N - df(q_i) + 0.5}{df(q_i) + 0.5}\right),$$

де N – загальна кількість документів, $df(q_i)$ – кількість документів, що містять терм q_i. Додавання 0.5 (згладжування) запобігає дуже малим значенням IDF для частих термів.

1.4 Інтуїція за формулою

BM25 враховує кілька важливих факторів:

1. **Значимість терма**: Рідкісні терми отримують вищу вагу (IDF компонента);
2. **Насичення частоти**: Функція TF має логарифмічну форму - якщо терм з'являється двічі, це набагато важливіше ніж один раз, але третій раз вже менш значущий. Це досягається через формулу з k_1 параметром;
3. **Нормалізація довжини**: Довші документи мають більше шансів містити терм, тому ми нормалізуємо за довжиною документа (параметр b контролює силу нормалізації).

1.5 Параметри BM25

Два основних параметри можна налаштовувати:

- k_1 (зазвичай 1.5): контролює наскільки сильно впливає частота терма. Більші значення означають менше логарифмічного насичення;
- b (зазвичай 0.75): контролює нормалізацію довжини. При $b = 0$ нема нормалізації, при $b = 1$ повна нормалізація.

На практиці BM25 залишається конкурентоспроможним методом навіть проти більш складних підходів.

2. TF-IDF як метод пошуку

TF-IDF (Term Frequency-Inverse Document Frequency) є спрощеною версією BM25 без насичення частоти та без нормалізації довжини:

$$\text{TF-IDF}(D, Q) = \sum_{i=1}^{n} \text{TF}\,(q_i, D) \cdot \text{IDF}(q_i),$$

де $\text{TF}(q_i, D)$ – просто частота терма в документі (або його логарифм $\log(1 + f(q_i, D))$ для пом'якшення).

Косинусна подібність (cosine similarity) між вектором запиту та вектором документа в TF-IDF просторі:

$$\text{sim}(Q, D) = \frac{\vec{Q} \cdot \vec{D}}{|\vec{Q}| \cdot |\vec{D}|}.$$

TF-IDF є простим, інтерпретованим і все ще корисним для багатьох практичних застосувань.

3. Щільний пошук: використання навчених ембедінгів

На відміну від розріджених методів, щільний пошук представляє документи та запити у вигляді векторів у неперервному просторі низької розмірності. Це дозволяє захопити семантичну подібність, а не тільки точне збігання термів.

3.1 Переваги щільного пошуку

1. **Семантична подібність**: Запит "млекопитаюче" може знайти документи про "ссавців" навіть без точного збігання слова;
2. **Обробка синонімії**: Синоніми матимуть близькі вектори;
3. **Обробка помилок**: Невеликі одруківки мають менший вплив на

семантичні вектори.

4. Архітектура bi-encoder для щільного пошуку

Bi-encoder архітектура складається з двох окремих енкодерів (encoders), які незалежно кодують запит та документ:

$$\text{embedding}_q = \text{encoder}_Q(q),$$

$$\text{embedding}_d = \text{encoder}_D(d).$$

Релевантність потім розраховується як косинусна подібність:

$$\text{score}(q, d) = \text{cos_sim}(\text{embedding}_q, \text{embedding}_d).$$

Архітектура bi-encoder має кілька суттєвих переваг. По-перше, можна заздалегідь обчислити ембедінги всіх документів у офлайн режимі. По-друге, пошук під час інференції є дуже швидким, запропоноване O(1) складністю для пошуку за індексом. По-третє, цей підхід масштабується до мільйонів документів. Однак, існують і недоліки: bi-encoder менш точна ніж cross-encoder через незалежність кодування запиту та документа, а також потребує великої кількості негативних прикладів для ефективного тренування.

5. Cross-encoder для переранжування

Cross-encoder обробляє пару запит-документ разом:

$$\text{score}(q, d) = \text{model}([q, d]).$$

На виході отримуємо скалярне значення релевантності.

Архітектура cross-encoder має суттєву перевагу у вищій точності, оскільки модель бачить весь контекст пари запит-документ одночасно, що робить її найоптимальнішою для використання на другому етапі переранжування, коли кількість кандидатів все ще невелика. Проте, вона має важливі недоліки: cross-encoder має O(n) обчислювальну складність, що вимагає обробки кожного кандидата окремо, а також її можна

використовувати виключно для переранжування, а не для початкового пошуку.

6. Тренування щільних пошуковиків: контрастивні втрати

6.1 Triplet Loss

Triplet loss орієнтована на тренування моделей на основі відстаней між прикладами:

$$L_{\text{triplet}} = \max(0, d(a,p) - d(a,n) + \epsilon),$$

де: a (anchor) – запит або документ; p (positive) – релевантний документ до a; n (negative) – нерелевантний документ до a; ϵ – маржа (margin); d – функція відстані.

Мета: зробити позитивні пари ближче, ніж негативні.

6.2 In-Batch Negatives

Простий, але ефективний підхід – використовувати інші приклади з мініпакета (batch) як негативи:

$$L = -\log \frac{\exp\left(\frac{\text{sim}(q_i, d_i^+)}{\tau}\right)}{\sum_{j=1}^{B} \exp\left(\frac{\text{sim}(q_i, d_j)}{\tau}\right)},$$

де τ – температурний коефіцієнт (temperature).

Це симетрична контрастивна втрата – можемо розглядати також пари (d, q) де d це якір.

6.3 Contrastive Learning Loss

Більш загальна форма контрастивної втрати:

$$L = -\log \frac{\exp\left(\frac{\text{sim}(q, d^+)}{\tau}\right)}{\exp\left(\frac{\text{sim}(q, d^+)}{\tau}\right) + \sum_{n=1}^{N} \exp\left(\frac{\text{sim}(q, d_n^-)}{\tau}\right)}.$$

Це спрощена версія NT-Xent (Normalized Temperature-scaled Cross Entropy) loss.

7. Hard Negative Mining

Не всі негативи однаково корисні для тренування. Hard negatives – це негативні приклади, які модель вже розглядає як подібні позитивним.

7.1 Статичне Hard Negative Mining

На початку епохи виберіть найбільш вражаючі негативи для кожного запиту за допомогою попередньо навченої моделі.

7.2 Динамічне Hard Negative Mining

Во час тренування, поміняйте легкі негативи на складніші: якщо модель вже коректно розділяє деякий позитив від негатива, зосередьтеся на випадках, коли негатив має високий score.

8. Дотренування ембедінг-моделей для домену

8.1 Sentence Transformers

Sentence Transformers (SBERT) – це бібліотека для отримання семантичних ембедінгів речень. Побудована на основі трансформерів, вона включає pooling шар для отримання представлення всього речення з представлень токенів.

8.2 Domain-Specific Fine-Tuning

Для адаптації до конкретного домену:

1. **Підготуйте дані**: Зберіть пари запит-документ зі своєю розмітку релевантності;
2. **Виберіть базову модель**: Використовуйте попередньо навчену SBERT модель;
3. **Визначте функцію втрати**: Triplet loss, contrastive loss, або

дохідність (margin-based) функція;

4. **Тренуйте**: Напротязі кількох епох на ваших даних;

5. **Оцініть**: На вальній множині перевірте поліпшення.

Приклад псевдокоду:

```
from sentence_transformers import SentenceTransformer, losses, models
import torch
# Завантажьте базову модель
model = SentenceTransformer('msmarco-distilbert-base-v3')
# Визначте функцію втрати
train_loss = losses.TripletLoss(model=model)
# Тренуйте
model.fit(
    train_objectives=[(train_dataloader, train_loss)],
    epochs=1,
    warmup_steps=100
)
```

9. Гібридний пошук: поєднання розріджених та щільних сигналів

Комбінування розріджених та щільних методів часто дає найкращі результати. Можливі варіанти:

9.1 Просте об’єднання рейтингів

Нормалізуйте оцінки з обох методів до одного діапазону, потім об’єднайте з вагами:

$$\text{score}_{\text{hybrid}} = \alpha \cdot \text{norm}(\text{score}_{\text{sparse}}) + (1 - \alpha) \cdot \text{norm}(\text{score}_{\text{dense}}),$$

де α – гіперпараметр (зазвичай 0.3-0.5 для BM25, решта для dense).

9.2 Виконання на різних етапах

1. **Перший етап**: Використовуйте BM25 для швидкого пошуку

кандидатів (retrieval);

2. **Другий етап**: Використовуйте dense retrieval для переранжування;

3. **Третій етап**: Використовуйте cross-encoder для фінального переранжування.

10. Reciprocal Rank Fusion (RRF)

RRF - це метод поєднання рейтингів без необхідності знати абсолютні оцінки:

$$\mathrm{RRF}(d) = \sum_{r \in R} \frac{1}{k + \mathrm{rank}_r(d)},$$

де k – постійна (зазвичай 60), $\mathrm{rank}_r(d)$ – ранг документа d в r-й системі рейтингу.

Метод RRF має кілька суттєвих переваг. По-перше, він не потребує нормалізації оцінок, оскільки працює безпосередньо з рангами. По-друге, він добре поєднує системи з різними шкалами оцінок, що робить його універсальним для комбінування різних підходів. По-третє, RRF є відмовостійким до викидів, оскільки ранжування пом'якшує екстремальні значення оцінок.

11. Оцінювання пошуку

11.1 Mean Reciprocal Rank (MRR)

$$\mathrm{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\mathrm{rank}_i},$$

де rank_i – позиція першого релевантного документа для i-го запиту.

Корисно коли цікавить знайти один релевантний документ.

11.2 NDCG (Normalized Discounted Cumulative Gain)

$$\mathrm{NDCG@}k = \frac{\mathrm{DCG@}k}{\mathrm{IDCG@}k},$$

де:

$$\mathrm{DCG@}k = \sum_{i=1}^{k} \frac{2^{\mathrm{rel}_i} - 1}{\log_2(i+1)}.$$

NDCG враховує релевантність (не тільки наявність), дисконтує позиції далі в списку, та нормалізується ідеальним рейтингом.

11.3 Recall@K

$$\mathrm{Recall@}k = \frac{\text{\# релевантних документів в top-k}}{\text{\# всіх релевантних документів}}.$$

Показує, яка частка всіх релевантних документів знайдена в топ-k.

11.4 Mean Average Precision (MAP)

$$\mathrm{MAP} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \mathrm{AP}\,(q_i),$$

де:

$$\mathrm{AP}(q) = \frac{1}{m} \sum_{k=1}^{n} P\,(k) \cdot \mathrm{rel}(k),$$

- m – кількість релевантних документів;
- $P(k)$ – точність на позиції k;
- $\mathrm{rel}(k)$ – індикатор релевантності на позиції k.

Контрольні запитання

1. Яка основна мета IDF компоненти в BM25? Як це впливає на рейтинг документів?

2. Пояснить різницю між розрідженим та щільним пошуком. Які недоліки кожного методу?

3. Як працює bi-encoder архітектура? Чому вона дозволяє швидкий пошук?

4. Що таке triplet loss і як він тренує моделі пошуку? Як вибирати негативи?

5. Поясніть концепцію hard negative mining. Чому це важливо для тренування?

6. Як дотренувати Sentence Transformers модель для специфічного домену?

7. Що таке cross-encoder і як воно відрізняється від bi-encoder?

8. Поясніть, як гібридний пошук поєднує розріджені та щільні методи.

9. Яка різниця між MRR, NDCG та Recall@K? Коли використовувати кожну метрику?

10. Як Reciprocal Rank Fusion коліхує рейтинги без нормалізації оцінок?

Рекомендована література

1. Robertson, S., & Zaragoza, H. (2009). The probabilistic relevance framework: BM25 and beyond. Foundations and Trends in Information Retrieval, 3(4), 333-389.

2. Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing.

3. Khattab, O., & Zaharia, M. (2020). ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In Proceedings

of the 43rd International ACM SIGIR Conference.

4. Cormack, G. V., Smucker, M. D., & Soboroff, I. (2011). Overview of the TREC 2011 Crowdsourcing Track. In TREC.

5. Formal, R., Lassance, C., Ranaldi, G., & Piwowarski, B. (2022). LyS: The Lyon System for the TREC 2021 CAST Track. In TREC.

ЛЕКЦІЯ 12: End-to-End RAG-системи та шаблони проєктування

Retrieval-Augmented Generation (RAG) - це парадигма, яка поєднує здатність мовних моделей до генерації тексту з можливістю пошуку релевантної інформації із зовнішніх джерел. На відміну від моделей, які покладаються тільки на параметри, що навчились під час тренування, RAG системи можуть отримувати доступ до актуальної, специфічної для домену інформації, що дозволяє їм генерувати більш точні та актуальні відповіді.

1. Що таке RAG та чому це важливо

1.1 Проблеми тільки-генеративних моделей

Великі мовні моделі (LLMs), навчені на фіксованому наборі даних, мають кілька суттєвих обмежень. По-перше, вони мають стійкі знання про факти в їх тренувальних даних, але невизначено поводяться щодо фактів поза межами цього набору через так звану knowledge cutoff. По-друге, вони часто генерують галюцинації, вимислюючи факти, які звучать правдоподібно, але насправді невірні. По-третє, вони не можуть легко оновлювати знання без повного переучування моделі. Нарешті, вони обмежені в контексті, коли необхідно розглядати великий обсяг документів одночасно.

1.2 Переваги RAG

1. **Актуальність**: Можна використовувати найсвіжіші документи;

2. **Точність**: Модель отримує контекст для гуртування (grounding) своїх відповідей;

3. **Інтерпретованість**: Можемо показати, з яких документів було взято інформацію;

4. **Контроль**: Можемо вибрати, які документи включити в систему;

5. **Зменшення галюцинацій**: Коли модель має правильний контекст, менше вдається до вигадування.

2. Етапи RAG конвеєру

На рис. 6 представлено типовий RAG конвеєр, що складається з кількох етапів.

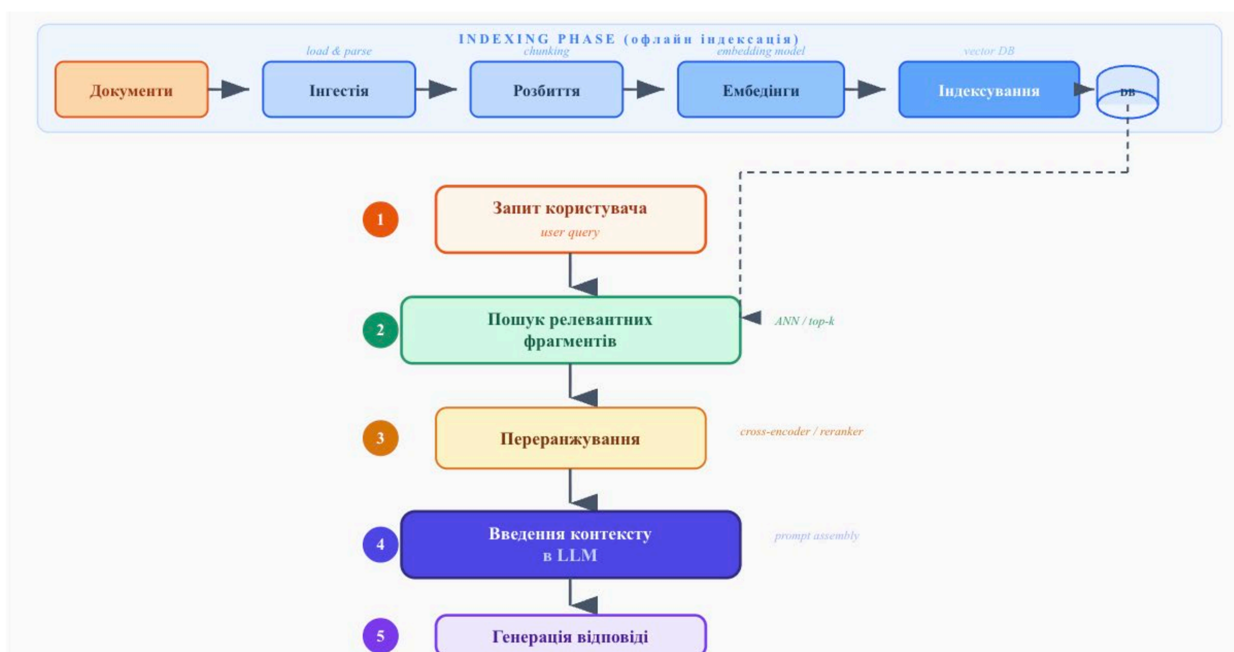

Рис. 6 – RAG процес

2.1 Інгестія документів (Document Ingestion)

Процес завантаження та підготовки документів до системи охоплює кілька ключових операцій: завантаження з різних форматів файлів, таких як PDF, Word та простий текст, веб-скрейпінг для отримання вмісту з інтернету, доступ до баз даних, а також очищення та нормалізацію даних для забезпечення консистентності.

2.2 Індексування та Інфраструктура

Документи зберігаються у спеціалізованих векторних базах даних, таких як Weaviate, Pinecone, Milvus, FAISS та Elasticsearch, які надають ефективні способи індексування та пошуку векторних представлень.

3. Стратегії розбиття документів

Як розділити великий документ на керовані фрагменти для пошуку.

3.1 Фіксований розмір (Fixed-Size Chunking)

Розділяйте документ на фіксовані блоки (наприклад, 512 токенів):

```
def fixed_size_chunks(text, chunk_size=512, overlap=0):
    tokens = tokenize(text)
    chunks = []
    for i in range(0, len(tokens), chunk_size - overlap):
        chunk = tokens[i:i + chunk_size]
        chunks.append(chunk)
    return chunks
```

Плюси: Просто, передбачуваний розмір. **Мінуси**: Може розбити речення, нездатна до семантичних границь.

3.2 Розбиття за реченнями (Sentence-Based Chunking)

Розділяйте по реченнях, потім групуйте речення в блоки:

```
def sentence_chunks(text, sentences_per_chunk=3, overlap=1):
    sentences = split_sentences(text)
    chunks = []
    for i in range(0, len(sentences), sentences_per_chunk - overlap):
        chunk = ' '.join(sentences[i:i + sentences_per_chunk])
        chunks.append(chunk)
    return chunks
```

Плюси: Природні семантичні границі. **Мінуси**: Різні розміри блоків.

3.3 Семантичне розбиття (Semantic Chunking)

Розділяйте там, де семантична подібність падає нижче порогу:

```
def semantic_chunks(text, threshold=0.5):
    sentences = split_sentences(text)
    embeddings = [embed(s) for s in sentences]
    chunks = [[sentences[0]]]
    for i in range(1, len(sentences)):
        similarity = cosine_similarity(
            embeddings[i],
            embeddings[i-1]
        )
        if similarity > threshold:
            chunks[-1].append(sentences[i])
        else:
            chunks.append([sentences[i]])
    return [' '.join(c) for c in chunks]
```

Плюси: Захоплює семантичні границі. **Мінуси**: Складніше обчислювально.

3.4 Рекурсивне розбиття (Recursive Chunking)

Спробуйте розділити на природні границі (абзаци), потім рекурсивно на більш дрібні блоки, якщо потрібно:

```
def recursive_chunks(text, delimiters=["\n\n", "\n", " "],
                     chunk_size=512, overlap=0):
    chunks = [text]
    for delim in delimiters:
        new_chunks = []
        for chunk in chunks:
```

```
            if len(chunk) < chunk_size:
                new_chunks.append(chunk)
            else:
                new_chunks.extend(chunk.split(delim))
        chunks = new_chunks
    return chunks
```

Плюси: Балансує природні границі та розмір. **Мінуси**: Більш складна логіка.

3.5 Перекриття та метадані

Додайте невелике перекриття між блоками для контексту: перекриття у 50 токенів може допомогти моделі краще зрозуміти контекст на границях блоків.

Збережіть метадані з кожним блоком:

```
chunk = {
  'text': 'фрагмент тексту...',
  'document_id': 'doc_123',
  'source': 'file.pdf',
  'chunk_index': 0,
  'metadata': {'title': 'Chapter 1', ...}
}
```

4. Стратегії пошуку

4.1 Одноетапний пошук (Single-Step Retrieval)

Прямо отримайте топ-k документів з релевантністю до запиту:

```
results = retrieve(query, top_k=5)
context = '\n'.join([r['text'] for r in results])
response = llm.generate(query, context=context)
```

Коли використовувати: Прості запити, малі колекції.

4.2 Багатостанцевий пошук (Multi-Hop Retrieval)

Для складних запитів, які можуть потребувати інформацію з кількох документів, рекурсивно шукайте:

```
def multi_hop_retrieve(initial_query, num_hops=2):
    all_results = set()
    current_query = initial_query
    for hop in range(num_hops):
        results = retrieve(current_query, top_k=3)
        all_results.update(results)
        # Синтезуйте нову запит на основі поточних результатів
        current_query = llm.generate_follow_up_query(
            initial_query, results
        )
    return list(all_results)
```

4.3 Декомпозиція запиту (Query Decomposition)

Розділіть складну запит на кілька простіших:

```
def decompose_and_retrieve(complex_query):
    sub_queries = llm.decompose(complex_query)
    all_results = []
    for sub_query in sub_queries:
        results = retrieve(sub_query, top_k=3)
        all_results.extend(results)
    return all_results
```

5. Переранжування

На першому етапі отримуємо велику кількість кандидатів, на другому етапі переранжуємо.

5.1 Cross-Encoder Reranker

```
from transformers import CrossEncoderModel
reranker = CrossEncoderModel('cross-encoder/ms-marco-MiniLM-L-6-v2')
# Отримайте початкові результати
candidates = retrieve(query, top_k=50)
# Переранжуйте з cross-encoder
scores = reranker.predict([
    [query, doc['text']] for doc in candidates
])
ranked = sorted(
    zip(candidates, scores),
    key=lambda x: x[1],
    reverse=True
)
```

5.2 Rerank API (Cohere, etc)

```
import cohere
client = cohere.Client(api_key='...')
results = client.rerank(
    model='rerank-english-v2.0',
    query=query,
    documents=[doc['text'] for doc in candidates],
    top_n=10
)
```

6. Введення контексту в LLM промпти

Важливо правильно форматувати контекст для LLM:

6.1 Базовий формат

Ви – корисний помічник. На основі наступного контексту, відповіді на

запитання.
Контекст:
{context}
Запитання: {query}
Відповідь:

6.2 Явна розмітка джерел

Ви - корисний помічник. На основі наступного контексту, відповіді на запитання та зазначте, з яких документів
Ви брали інформацію.
Документи:
{for each doc}
[Doc {index}] {doc_source}
{doc_text}
{/for}
Запитання: {query}
Відповідь:

6.3 Ланцюжки думок (Chain-of-Thought)

Ви - експерт в аналізі документів. Давайте розповсюджуємося
крок за кроком.
Контекст:
{context}
Запитання: {query}
Давайте розповсюджуємося крок за кроком:
1. Спочатку розглянемо...

7. Загальні шаблони RAG

7.1 Naive RAG

Найпростіша форма: пошук → генерація.

```
def naive_rag(query):
    # Пошук
    docs = retrieve(query, top_k=5)
    context = format_context(docs)
    # Генерація
    response = llm.generate(
        f"Context: {context}\n\nQuestion: {query}"
    )
    return response
```

Проблеми: Не розглядає якість пошуку, не переранжує, не оптимізує контекст.

7.2 Advanced RAG

Додайте переранжування та покращену схему (schema):

```
def advanced_rag(query):
    # Пошук + переранжування
    candidates = retrieve(query, top_k=50)
    top_docs = rerank(query, candidates, top_k=5)
    # Дополнительно перевірте якість контексту
    context = format_context(top_docs)
    # Перевірте, чи контекст достатній
    if not has_sufficient_context(query, context):
        # Спробуйте альтернативний пошук
        alt_docs = retrieve_alternative(query)
        context = combine_contexts(context, alt_docs)
    # Генерація з явною інструкцією щодо джерел
    response = llm.generate(
        f"Use the following context to answer the question. "
        f"Always cite your sources.\n\n"
        f"Context: {context}\n\n"
```

```
        f"Question: {query}"
    )
    return response
```

7.3 Modular RAG

Окремо оптимізуйте кожен компонент: пошук, переранжування, генерацію.

```
class ModularRAG:
    def __init__(self, retriever, reranker, llm):
        self.retriever = retriever  # Оптимізована bi-encoder
        self.reranker = reranker    # Оптимізований cross-encoder
        self.llm = llm              # Обраний LLM
    def retrieve_and_rank(self, query):
        # Пошук з налаштованою стратегією
        docs = self.retriever.search(
            query,
            top_k=100,
            filter=apply_filters()
        )
        # Переранжування
        ranked = self.reranker.rerank(query, docs, top_k=10)
        return ranked
    def generate(self, query, context):
        # Генерація з налаштованим промптом
        response = self.llm.generate(
            template='expert_qa',
            query=query,
            context=context,
            style='concise'
        )
```

```
        return response
    def answer(self, query):
        docs = self.retrieve_and_rank(query)
        context = format_context(docs)
        response = self.generate(query, context)
        return {
            'answer': response,
            'sources': [d['source'] for d in docs]
        }
```

8. Примітки користування

8.1 QA асистент на основі документів

```
class DocumentQAAssistant:
    def __init__(self, docs_path):
        self.docs = load_documents(docs_path)
        self.chunks = chunk_documents(self.docs)
        self.embeddings = embed_chunks(self.chunks)
        self.index = build_index(self.embeddings)
    def answer_question(self, question):
        # Пошук релевантних фрагментів
        query_emb = embed(question)
        top_chunks = self.index.search(query_emb, top_k=5)
        # Форматування контексту
        context = '\n\n'.join([c['text'] for c in top_chunks])
        # Генерація відповіді
        prompt = f"""Based on the following information from documents:
{context}
Answer this question: {question}"""
        answer = llm.generate(prompt)
        return {
```

```
            'answer': answer,
            'sources': [c['document_id'] for c in top_chunks]
        }
```

8.2 Код-асистент

```
class CodeAssistant:
    def __init__(self, codebase_path):
        # Індексуйте весь кодбейс
        self.files = load_code_files(codebase_path)
        self.chunks = split_code(self.files, language='python')
        self.index = build_code_index(self.chunks)
    def explain_code(self, query):
        # Знайдіть релевантні фрагменти коду
        relevant = self.index.search(query)
        context = format_code_context(relevant)
        prompt = f"""You are an expert code reviewer.
Relevant code:
{context}
Question: {query}
Provide a detailed explanation."""
        return llm.generate(prompt)
    def suggest_refactoring(self, function_name):
        # Знайдіть функцію та пов'язаний код
        code = self.find_function(function_name)
        related = self.find_related_code(function_name)
        prompt = f"""Analyze this code and suggest refactorings:
{code}
Related code patterns in codebase:
{related}"""
        return llm.generate(prompt)
```

8.3 Enterprise Chatbot

```
class EnterpriseChatbot:
    def __init__(self, knowledge_base):
        self.kb = knowledge_base
        self.conversation_history = []
        self.user_context = {}
    def process_message(self, user_id, message):
        # Отримайте контекст користувача
        user_context = self.get_user_context(user_id)
        # Декомпозуйте запит залежно від контексту
        interpreted_query = self.interpret_query(
            message,
            conversation_history=self.conversation_history,
            user_context=user_context
        )
        # Пошук + переранжування
        docs = self.kb.search(interpreted_query, top_k=10)
        # Форматування для генерації
        context = self.format_enterprise_context(docs)
        # Генерація з персоналізацією
        response = llm.generate(
            f"""You are a helpful enterprise assistant.
User context: {user_context}
Conversation history: {self.conversation_history[-3:]}
Knowledge base:
{context}
User message: {message}
Response:"""
        )
        # Збереження в історію
```

```
self.conversation_history.append({
    'user': message,
    'assistant': response,
    'sources': [d['id'] for d in docs]
})
return response
```

9. Оцінювання RAG систем

Оцінювання RAG вимагає оцінювання як пошуку, так і генерації.

9.1 Метрики пошуку

Для оцінювання пошуку в RAG системах використовуються кілька ключових метрик. **Recall@K** показує, яка частка релевантних документів знайдена у топ-k результатів. **NDCG** (Normalized Discounted Cumulative Gain) оцінює як добре ранжуються релевантні документи з урахуванням їх позицій. **MRR** (Mean Reciprocal Rank) визначає позицію першого релевантного документа для кожного запиту.

9.2 Метрики генерації

Для оцінювання якості генерованого тексту використовуються такі метрики. **BLEU** вимірює збіг з еталонною відповіддю через вирівнювання n-грам. **ROUGE** є recall-oriented метрикою, яка фокусується на повноті вибору важливих фраз. **METEOR** враховує синоніми та варіації слів, надаючи більш гнучку оцінку якості тексту.

9.3 RAG-специфічні метрики: RAGAS

RAGAS (Retrieval-Augmented Generation Assessment) оцінює чотири аспекти:

1. **Faithfulness**: Чи відповідь вірна наданому контексту?
    - Виділіть твердження в відповіді;

– Перевірте кожне твердження проти контексту.

2. **Answer Relevance**: Чи відповідь релевантна запиту?

– Генеруйте можливі запити для даної відповіді;

– Перевірте подібність до оригінальної запиту.

3. **Context Relevance**: Чи контекст релевантний запиту?

– Перевірте подібність кожного документа до запиту.

4. **Context Recall**: Чи контекст містить всю необхідну інформацію?

– Перевірте, чи всі твердження в еталонній відповіді покриті контекстом.

```
from ragas import evaluate
from ragas.metrics import faithfulness, answer_relevance, context_recall
results = evaluate(
    dataset,
    metrics=[faithfulness, answer_relevance, context_recall]
)
```

10. Поширені режими помилок та налагодження RAG конвеєрів

10.1 Поширені режими помилок

1. **Lost in the Middle**: Модель ігнорує релевантну інформацію в середині контексту.

– Рішення: Переупорядкуйте контекст за релевантністю.

2. **Неправильні погодження**: Контекст містить неправильну інформацію.

– Рішення: Поліпшіть якість пошуку, переранжування, або очищення документів.

3. **Недостатня інформація**: Пошук не знаходить релевантні документи.

– Рішення: Спробуйте гібридний пошук, переформулювання запиту, розширення контексту.

4. **Галюцинації**: Модель вигадує факти, не присутні в контексті.

– Рішення: Явно інструктуйте модель цитувати джерела, використовуйте chain of thoughts.

10.2 Техніки налагодження

1. **Логування та моніторинг**:

```
def log_rag_pipeline(query, retrieval_results, response):
    logger.info({
        'query': query,
        'num_retrieved': len(retrieval_results),
        'top_scores': [r['score'] for r in retrieval_results[:3]],
        'response_length': len(response),
        'response': response
    })
```

2. **Оцінювання кожного компонента окремо**:

– Чи пошук видав релевантні документи?

– Чи переранжування покращило порядок?

– Чи генерація використала контекст?

3. **Тестові набори**:

```
test_cases = [
    {
        'query': '...',
        'expected_sources': ['doc1', 'doc2'],
        'expected_answer_contains': ['key phrase']
    },
    ...
```

]

Контрольні запитання

1. Поясніть переваги RAG порівняно з тільки-генеративними LLM.
2. Які основні етапи конвеєру RAG? Як вони взаємодіють?
3. Порівняйте різні стратегії розбиття документів. Коли використовувати кожну?
4. Поясніть різницю між одноетапним та пошуком за допомогою променя (beam search). Наведіть приклади.
5. Як переранжування поліпшує якість RAG? Наведіть приклад cross-encoder переранжування.
6. Поясніть три шаблони RAG: Naive, Advanced та Modular. Які їх переваги та недоліки?
7. Як оцінювати RAG системи? Назвіть метрики пошуку та генерації.
8. Що таке RAGAS та які чотири метрики вона вимірює?
9. Назвіть три поширені режими помилок RAG конвеєрів та способи їх виправлення.
10. Як поліпшити контекстну релевантність у RAG систем, якщо пошук не знаходить потрібних документів?

Рекомендована література

1. Lewis, P., Perez, E., Piktus, A., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In Advances in Neural Information Processing Systems (NeurIPS).
2. Karpukhin, V., Oguz, B., Min, S., et al. (2020). Dense Passage Retrieval for Open-Domain Question Answering. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP).

3. Xu, T. L., Zhao, Z. J., & Zhu, Y. (2023). A Survey on Retrieval-Augmented Text Generation for Large Language Models. arXiv preprint arXiv:2312.10997.

4. OpenAI Cookbook: https://cookbook.openai.com/examples/how_to_build_a_rag_system

5. LangChain Documentation: https://python.langchain.com/docs/use_cases/question_answering/

ЛЕКЦІЯ 13: MLOps для LLM: деплой, квантизація та дистиляція

Вступ

Розгортання (deployment) великих мовних моделей має унікальні виклики, які відрізняють його від стандартних машинного навчання. LLM потребують величезного обсягу пам'яті, генерують токени послідовно (що робить inference динамічним), та часто розгортуються як сервіси з високим навантаженням. Цей розділ розглядає техніки та інструменти для ефективного розгортання та оптимізації LLM у виробництві.

1. Чому MLOps важлива для LLM

1.1 Унікальні виклики LLM

1. **Велика необхідність в пам'яті**: Навіть малі LLM потребують гігабайт VRAM. Модель Llama-7B потребує мінімум 14 GB у FP32 форматі.

2. **Повільна генерація**: Генерація токенів є авторегресивною та послідовною, що створює затримки.

3. **Варіативна довжина**: Довжина входу та виходу варіюється, що ускладнює планування ресурсів.

4. **Вартість**: Інференс на GPU дорогий, що робить оптимізацію критичною для економічної доцільності.

5. **Масштабування**: Від кількох користувачів до мільйонів.

1.2 Цілі MLOps для LLM

MLOps для LLM має на меті досягнення кількох ключових цілей. По-перше, покращення throughput, тобто кількості запитів, оброблених на одиницю часу. По-друге, зменшення latency, включаючи час від запиту до першого токена та загальний час генерації. По-третє, зменшення вартості, оскільки невикористана пам'ять та обчислювальні ресурси представляють мертві витрати. По-четверте, забезпечення надійності системи через graceful degradation під навантаженням. Нарешті, моніторинг системи для відслідковування якості, затримок та помилок.

2. Сервісні стеки: архітектура vLLM

vLLM - це високопродуктивна бібліотека для генерації LLM, розроблена в UC Berkeley. Вона вирішує сервісні проблеми через інноваційні техніки.

2.1 Проблема з наївною інференцією

Запит 1: "Розповіді про..." → GPU займає пам'ять для всієї послідовності
Запит 2: приходить поки 1 все ще генерує... → Очікує у черзі!

2.2 Continuous Batching

На відміну від статичного батчування, vLLM використовує динамічне батчування:

```
Час 1:
  Запит 1: [токен 1, 2, 3]
  Запит 2: [токен 1, 2]
Час 2:
  Запит 1: [токен 1, 2, 3, 4]  (завершене)
  Запит 3: [токен 1, 2, 3, 4, 5]  (нова, заповнює місце)
  Запит 2: [токен 1, 2, 3]
```

Запит можна видалити з батчу як тільки він завершений, і одразу додати новий. Це значно покращує GPU утилізацію.

2.3 PagedAttention

Однією з головних проблем в autoregressive декодуванні є розподіл КВ-кешу (KV cache). Для кожного токена, що генерується, всі ранішні токени зберігаються в пам'яті для обчислення attention.

PagedAttention розділяє КВ-кеш на блоки (pages), дозволяючи гнучкіше управління пам'яттю:

Традиційний KV-кеш: PagedAttention: [████████████████] [██][██][██][██][empty] (неправильна фрагментація) (можна розділяти та розширяти)

Результат: 24x менше пам'яті для KV-кешу в деяких випадках.

2.4 Архітектура vLLM

```
from vllm import LLM, SamplingParams
# Завантажьте модель
llm = LLM(
    model='meta-llama/Llama-2-7b-hf',
    tensor_parallel_size=2,  # На 2 GPU
    gpu_memory_utilization=0.9
)
# Встановіть параметри генерації
sampling_params = SamplingParams(
    temperature=0.7,
    top_p=0.95,
    max_tokens=128
)
# Обробіть батч запитів одразу
prompts = [
```

```
    "Привіт, як справи?",
    "Розповіді про...",
    "Напишіть опис..."
]
outputs = llm.generate(prompts, sampling_params)
for output in outputs:
    print(f"Prompt: {output.prompt}")
    print(f"Generated: {output.outputs[0].text}")
```

3. llama.cpp: запуск моделей на CPU

llama.cpp – це ефективна реалізація LLM інференсу на CPU, розроблена Ggerganov.

3.1 Формат GGUF

GGUF (GPT-Generated Unified Format) – це формат для зберігання квантизованих моделей:

На рис. 7 показано структуру GGUF файлу.

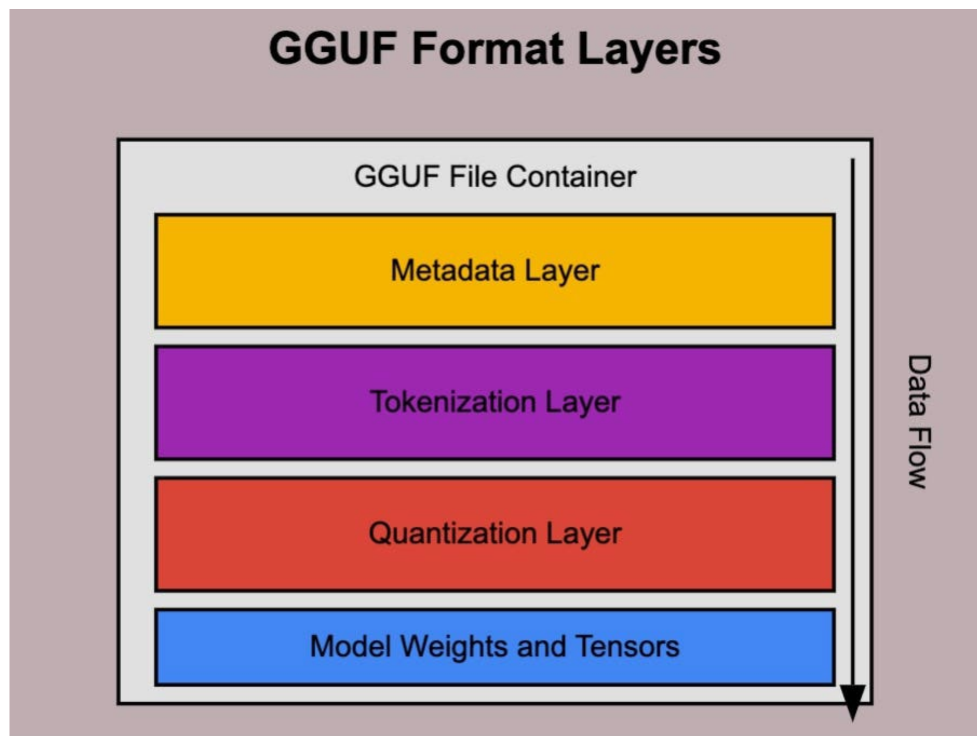

Рис. 7 – GGUF файл

3.2 Запуск моделей з llama.cpp

```
# Завантажте GGUF модель
wget https://huggingface.co/TheBloke/Llama-2-7B-GGUF/resolve/main/llama-2-7b.Q4_K_M.gguf
# Запустіть сервер
./server -m llama-2-7b.Q4_K_M.gguf -c 2048 --port 8000
# Зробіть запит
curl http://localhost:8000/completion \
  -d '{"prompt": "Привіт,", "n_predict": 32}'
```

3.3 Переваги llama.cpp

1. **Мала кількість пам'яті**: GGUF з Q4 квантизацією на ~7B моделі = ~4GB.
2. **Швидкість на CPU**: Оптимізований код для інтельського CPU.
3. **Легкість розгортування**: Один бінарний файл, можна запустити на ноутбуці.
4. **Apple Silicon**: Добра оптимізація для Metal (Apple GPU).

4. Квантизація

Квантизація зменшує точність чисел для економії пам'яті та ризику. Замість 32-бітних чисел, використовуємо 8-бітні або 4-бітні.

4.1 INT8 квантизація

Найпростіша форма: карта FP32 значення до INT8 діапазону:

```
# Оригінальний вес: 0.0342 (float32)
# Квантизований: 9 (int8) - в діапазоні [-128, 127]
def quantize_int8(weights):
    scale = 127.0 / np.max(np.abs(weights))
    quantized = np.round(weights * scale).astype(np.int8)
```

```
    return quantized, scale
def dequantize_int8(quantized, scale):
    return quantized.astype(np.float32) / scale
```

Пам'ять: FP32 → INT8 = 4x зменшення.

4.2 INT4 квантизація

Ще більше зменшення, але більша втрата якості:

```
def quantize_int4(weights):
    # Упакуйте два 4-бітних значення в один байт
    scale = 7.0 / np.max(np.abs(weights))
    quantized_8bit = np.round(weights * scale)
    # Упакуйте два int4 в int8
    packed = quantized_8bit[0::2] | (quantized_8bit[1::2] << 4)
    return packed, scale
```

Пам'ять: FP32 → INT4 = 8x зменшення.

4.3 Mixed Precision (INT8/INT4 з інтервалами)

Замість квантизації всіх ваг однаково, деякі критичні ваги залишаються у FP16:

- **Attention**: INT8 або INT4;
- **Вихідний шар**: FP16.

4.4 Практичні бібліотеки

bitsandbytes для INT8:

```
from bitsandbytes.nn import Linear8bitLt
# Замініть звичайний Linear на 8bit
model.linear = Linear8bitLt(
    input_features,
    output_features,
```

```
    has_fp16_weights=False
)
```

GPTQ для INT4:

```
from auto_gptq import AutoGPTQForCausalLM
model = AutoGPTQForCausalLM.from_quantized(
    'model-name',
    use_safetensors=True,
    device_map='auto'
)
```

AWQ – альтернатива GPTQ:

```
from awq import AutoAWQForCausalLM
model = AutoAWQForCausalLM.from_quantized('model-name')
```

4.5 Вплив квантизації на якість та швидкість

	FP32	INT8	INT4
Пам'ять (7B):	14 GB	4 GB	2 GB
Throughput:	1x	2.2x	3.5x
BLEU (англійський):	32.5	32.4	31.8
BLEU втрата:	-	0.3%	2.1%

Висновок: INT8 дає мало втрат, INT4 може мати помітну втрату якості для деяких завдань.

5. Knowledge Distillation

Knowledge distillation – це техніка переносу знань від більш великої моделі (teacher) до більшої моделі (student).

5.1 Основна концепція

```
Teacher LLM (Llama-70B)   -----(знання)-----> Student LLM (Llama-7B)
(складна, повільна)                            (мала, швидка)
```

5.2 Logit Distillation

Teacher модель генерує soft labels (розподіл ймовірностей над всіма токенами), а student навчається копіювати цей розподіл:

```
def kl_divergence_loss(logits_teacher, logits_student, temperature=3.0):
    """KL дивергенція з температурним масштабуванням"""
    # Масштабуйте логіти температурою
    probs_teacher = F.softmax(logits_teacher / temperature, dim=-1)
    log_probs_student = F.log_softmax(logits_student / temperature, dim=-1)
    # KL дивергенція
    kl_loss = F.kl_div(log_probs_student, probs_teacher, reduction='batchmean')
    # Масштабуйте на температуру в квадраті
    return temperature ** 2 * kl_loss
def distillation_loss(logits_student, logits_teacher, labels, alpha=0.7,
temperature=3.0):
    """Комбініована втрата: дистиляція + оригінальна задача"""
    # Дистиляційна втрата
    kl_loss = kl_divergence_loss(logits_teacher, logits_student, temperature)
    # Оригінальна cross-entropy втрата
    ce_loss = F.cross_entropy(logits_student, labels)
    # Комбінована втрата
    total_loss = alpha * kl_loss + (1 - alpha) * ce_loss
    return total_loss
```

Температура τ контролює м'якість цілей: - $\tau = 1$: hard labels (100% на правильному класі). - $\tau = 3$: м'які labels (розподіл ймовірностей).

5.3 Feature Distillation

Розпорядитись більш глибокими шарами моделей:

```
def feature_distillation_loss(hidden_teacher, hidden_student):
    """Втрата на основі проміжних представлень"""
```

```
    # Проектуйте до одного простору (якщо розміри різні)
    hidden_student_projected = projection_layer(hidden_student)
    # MSE між прихованими станами
    loss = F.mse_loss(hidden_student_projected, hidden_teacher)
    return loss
```

5.4 Практичний приклад дистиляції

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch.nn.functional as F
# Завантажьте teacher та student
teacher = AutoModelForCausalLM.from_pretrained('meta-llama/Llama-2-70b')
student = AutoModelForCausalLM.from_pretrained('meta-llama/Llama-2-7b')
# Оптимізатор
optimizer = torch.optim.AdamW(student.parameters(), lr=1e-5)
# Тренування
for batch in dataloader:
    input_ids = batch['input_ids']
    # Отримайте логіти від обох моделей
    with torch.no_grad():
        teacher_outputs = teacher(input_ids, output_hidden_states=True)
        teacher_logits = teacher_outputs.logits
    student_outputs = student(input_ids, output_hidden_states=True)
    student_logits = student_outputs.logits
    # Розрахуйте втрату дистиляції
    loss = distillation_loss(
        student_logits,
        teacher_logits,
        batch['labels'],
        alpha=0.7,
        temperature=3.0
```

```python
    )
    # Зворотна пропаганда
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

6. Docker та Docker Compose для розгортування LLM

6.1 Базовий Dockerfile

```dockerfile
FROM nvidia/cuda:12.2.0-runtime-ubuntu22.04
WORKDIR /app
# Встановіть залежності
RUN apt-get update && apt-get install -y \
    python3.10 python3-pip \
    && rm -rf /var/lib/apt/lists/*
# Скопіюйте вимоги
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# Скопіюйте код
COPY app.py .
COPY models/ models/
# Виставте порт
EXPOSE 8000
# Запустіть сервіс
CMD ["python3", "app.py"]
```

6.2 vLLM у Docker

```dockerfile
FROM nvcr.io/nvidia/pytorch:23.10-py3
WORKDIR /app
RUN pip install vllm
COPY app.py .
```

```
COPY models/ models/
EXPOSE 8000
CMD ["python3", "-m", "vllm.entrypoints.api_server", \
    "--model", "meta-llama/Llama-2-7b-hf", \
    "--port", "8000", \
    "--gpu-memory-utilization", "0.9"]
```

6.3 Docker Compose для масштабування

```
version: '3.8'
services:
  vllm-worker-1:
    image: vllm:latest
    environment:
      - CUDA_VISIBLE_DEVICES=0
    ports:
      - "8001:8000"
    volumes:
      - ./models:/models
  vllm-worker-2:
    image: vllm:latest
    environment:
      - CUDA_VISIBLE_DEVICES=1
    ports:
      - "8002:8000"
    volumes:
      - ./models:/models
  load-balancer:
    image: nginx:latest
    ports:
      - "8000:80"
```

```
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - vllm-worker-1
      - vllm-worker-2
```

nginx.conf:

```
upstream vllm_backend {
    server vllm-worker-1:8000;
    server vllm-worker-2:8000;
}
server {
    listen 80;
    location / {
        proxy_pass http://vllm_backend;
    }
}
```

7. Оптимізація інференсу: спекулятивне декодування та батчування

7.1 Speculative Decoding

Генеруйте кілька токенів з дрібною моделлю, потім перевірте з великою:

Дрібна модель: [токен 1] → [токен 2] → [токен 3] → [токен 4] (швидко, може бути помилково)

↓

Велика модель: [перевірка] → якщо вірно, прийміть; якщо ні, коригуйте.

Результат: 2-3x прискорення для довгих послідовностей без втрати якості.

7.2 Батчування стратегії

Існують три основні стратегії батчування для оптимізації інференсу.

Static Batching використовує фіксований розмір батчу, однак це призводить до поганої утилізації для варіативних послідовностей різної довжини. **Dynamic Batching**, реалізована в vLLM, використовує динамічні розміри батчу для досягнення кращої утилізації ресурсів. **Continuous Batching**, також реалізована в vLLM, дозволяє додавати нові запити коли попередні завершуються, що робить цей підхід найкращим для API сервісів з невизначеним потоком запитів.

8. Моніторинг та спостереження для LLM сервісів

8.1 Ключові метрики

1. **Latency**: TTFT (Time To First Token), TPS (Tokens Per Second);
2. **Throughput**: Запитів на секунду, токенів на секунду;
3. **Resource Utilization**: GPU пам'ять, CPU, пропускна спроможність;
4. **Quality**: метрики BLEU, ROUGE на вхідних прикладах;
5. **Availability**: Uptime, error rate.

8.2 Практичний моніторинг

```
import time
from prometheus_client import Counter, Histogram, Gauge

# Визначте метрики
requests_total = Counter(
    'llm_requests_total',
    'Total LLM requests',
    ['model', 'status']
)

request_duration = Histogram(
```

```python
    'llm_request_duration_seconds',
    'LLM request duration',
    ['model']
)

gpu_memory = Gauge(
    'gpu_memory_bytes',
    'GPU memory usage'
)

@app.post("/generate")
async def generate(prompt: str):
    start = time.time()

    try:
        output = llm.generate(prompt)
        requests_total.labels(model='llama-7b', status='success').inc()
        request_duration.labels(model='llama-7b').observe(time.time() - start)
        return output
    except Exception as e:
        requests_total.labels(model='llama-7b', status='error').inc()
        raise
```

Контрольні запитання

1. Які унікальні виклики стоять перед розгортанням LLM порівняно зі звичайним ML?

2. Поясніть continuous batching та PagedAttention в vLLM. Як вони поліпшують утилізацію GPU?

3. Як llama.cpp дозволяє запускати LLM на CPU? Що таке GGUF

формат?

4. Поясніть INT8 та INT4 квантизацію. Як вони впливають на пам'ять та якість?

5. Як GPTQ та AWQ відрізняються від простої квантизації?

6. Що таке knowledge distillation? Поясніть logit distillation та feature distillation.

7. Як температурний коефіцієнт впливає на цілі дистиляції?

8. Напишіть Dockerfile для розгортання vLLM моделі.

9. Як налаштувати Docker Compose для масштабування LLM з балансуванням навантаження?

10. Яких ключових метрик моніторити для LLM сервісу? Як виміряти TTFT та TPS?

Рекомендована література

1. Kwon, H., Li, Z., Zhuang, S., et al. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. In Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 2023).

2. Gerganov, G. (2024). llama.cpp: Inference of LLaMA model in pure C/C++. GitHub: https://github.com/ggerganov/llama.cpp

3. Frantar, E., Ashkboos, S., Hoover, B., et al. (2022). GPTQ: Accurate Post-Training Quantization of Generative Pre-trained Transformers. arXiv preprint arXiv:2210.17323.

4. Lin, J., Tang, J., Tang, H., et al. (2023). AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. arXiv preprint arXiv:2306.00978.

5. Hinton, G., Vanhoucke, V., & Dean, J. (2015). Distilling the

Knowledge in a Neural Network. arXiv preprint arXiv:1503.02531.

ЛЕКЦІЯ 14: Reasoning LLM та Chain-of-Thought

Вступ

Одною з найцікавіших розвитків у LLM за останні роки є визнання того, що моделі здатні до більш складного рас судження (reasoning) при дачі їм часу та структури для роботи. На відміну від наївного вибору перших токенів, техніки, такі як Chain-of-Thought (CoT), дозволяють моделям явно розповсюджуватись через проміжні кроки, що часто приводить до правильних відповідей. Цей розділ розглядає різні техніки prompting для поліпшення reasoning, спеціалізовані моделі для reasoning та бенчмарки для їх оцінювання.

1. Що таке reasoning в контексті LLM

Reasoning – це процес розповсюджування від відомих фактів через логічні кроки для досягнення висновку. У LLM контексті це означає:

1. **Явна артикуляція**: Модель явно записує проміжні кроки замість просто генерування наступного токена;
2. **Покрокове розв'язання**: Розбиття складної задачі на простіші підзадачі;
3. **Самоконтроль**: Перевірка логіки та коректування помилок.

2. Промптинг стратегії

2.1 Zero-Shot Prompting

Просто дайте завдання без прикладів:

Запит: Якщо у мене було 5 яблук, я з'їв 2, потім купив 3 більше,
скільки у мене яблук?
Відповідь: Давайте розповсюджуватися крок за кроком:
1. Почав з 5 яблук

2. З'їв 2: 5 - 2 = 3

3. Купив ще 3: 3 + 3 = 6

Відповідь: 6 яблук

2.2 Few-Shot Prompting

Надайте кілька прикладів:

Приклад 1:

Запит: У мене є 3 червоні м'ячі та 2 блакитні м'ячі. Скільки у мене м'ячів?

Розв'язання: 3 + 2 = 5 м'ячів

Приклад 2:

Запит: Моя сестра має 4 пари шкарпеток. У мене є 2 пари більше, ніж у неї. Скільки пар шкарпеток у мене?

Розв'язання: Сестра: 4 пари. У мене: 4 + 2 = 6 пар

Запит: Якщо дерево має 10 гілок, а кожна гілка має 5 листків,

скільки листків на дереві?

2.3 Chain-of-Thought (CoT)

Явно інструктуйте модель думати крок за кроком:

Запит: Якщо у групи 30 учнів, і 40% з них дівчата, скільки хлопців у групі?

Промпт:

Давайте розповсюджуватися крок за кроком.

Крок 1: Знайти кількість дівчат.

Дівчат = 30 * 0.40 = 12

Крок 2: Знайти кількість хлопців.

Хлопців = 30 - 12 = 18

Відповідь: 18 хлопців

3. Zero-Shot CoT: “Let’s think step by step”

Простий, але ефективний промпт:

```
def zero_shot_cot(question, model):
```

```
    prompt = f"{question}\n\nLet's think step by step:"
    response = model.generate(prompt)
    return response
```

Результати показують значне поліпшення на математичних задачах:

Задача	Точність без CoT	Точність з CoT	Поліпшення
GSM8K (8-клас. математика)	27.8%	56.9%	+29.1%
SVAMP (різні слово-задачі)	56.0%	78.2%	+22.2%
MAWPS (мат. задачі)	55.0%	85.1%	+30.1%

4. Self-Consistency: генерування кількох шляхів і голосування

На відміну від детермінованого вибору одного шляху reasoning, self-consistency дозволяє моделі генерувати кілька можливих шляхів і голосувати:

```
def self_consistency(question, model, num_samples=5):
    """Генеруйте кілька відповідей та виберіть найбільш часту"""
    answers = []
    for _ in range(num_samples):
        # Генеруйте з температурою > 0 для різноманітності
        response = model.generate(
            f"{question}\n\nLet's think step by step:",
            temperature=0.7,
            max_tokens=500
        )
        # Витягніть фінальну відповідь
        answer = extract_final_answer(response)
        answers.append(answer)
    # Голосуйте (модальна відповідь)
```

```python
    from collections import Counter
    most_common = Counter(answers).most_common(1)[0][0]
    return most_common, answers
```

Результати self-consistency демонструють суттєві поліпшення. На бенчмарку GSM8K точність досягає 70.5% (порівняно з 56.9% для одного шляху), на MATH — 73.1% (порівняно з 46.4%), а на MAWPS — 92.9% (порівняно з 85.1%).

5. Tree of Thoughts (ToT)

Замість лінійної послідовності (CoT), ToT дозволяє дослідження дерева можливостей:

```
                 Оригінальна задача
                        |
              __________|__________
             /          |          \
          Крок 1a    Крок 1b    Крок 1c
         /  |  \   |          |
        2a  2b  2c  2d         2e
        |   |   |   |          |
       3a  3b  ✓   3d         3e
```

де ✓ означає, що цей шлях привів до успіху.

```python
def tree_of_thoughts(question, model, max_depth=3):
    """Дослідження дерева можливих шляхів reasoning"""
    def explore(current_state, depth, path=[]):
        if depth == max_depth:
            # Оцініть фінальну відповідь
            score = evaluate_answer(current_state)
            return score, path + [current_state]
        # Генеруйте кілька можливих наступних кроків
```

```
        next_steps = model.generate_multiple(
            f"Next steps for: {current_state}",
            num_candidates=3
        )
        best_score = -float('inf')
        best_path = []
        for next_step in next_steps:
            # Перевірте чи цей крок обіцяючий
            local_score = evaluate_step(next_step)
            if local_score > threshold:  # Пруніруйте henerspektive гілки
                score, final_path = explore(
                    next_step,
                    depth + 1,
                    path + [current_state]
                )
                if score > best_score:
                    best_score = score
                    best_path = final_path
        return best_score, best_path
    score, path = explore(question, 0)
    return path, score
```

6. Graph of Thoughts (GoT)

Розширення ToT, де вузли можуть мати декілька входів (не тільки від батька). Це дозволяє більш складні залежності.

7. ReAct: Reasoning + Acting

ReAct (Reasoning + Acting) поєднує явне reasoning з діями (наприклад, пошук в Google, доступ до інструментів):

```
def react_agent(question, tools):
```

```
    """Agent поєднує reasoning та action"""
    history = []
    state = question
    for step in range(max_steps):
        # Reasoning: що робити далі?
        thought = llm.generate(
            f"Question: {question}\n"
            f"History: {history}\n"
            f"What's the next step?"
        )
        # Action: виконайте дію
        if "Search" in thought:
            query = extract_search_query(thought)
            result = search(query)
        elif "Calculate" in thought:
            expression = extract_expression(thought)
            result = calculate(expression)
        elif "Finish" in thought:
            result = extract_answer(thought)
            return result
        else:
            result = llm.generate(f"Reason about: {thought}")
        # Додайте до історії
        history.append({
            'thought': thought,
            'action': action,
            'observation': result
        })
    return state
```

Приклад:

Питання: Коли народилась Мадонна?
Думка 1: Мне потрібна інформація про народження Мадонни.
Дія 1: Пошук "Мадонна дата народження"
Спостереження: Мадонна народилась 16 серпня 1958 року
Думка 2: У мене є вся потрібна інформація.
Дія 2: Завершити
Відповідь: Мадонна народилась 16 серпня 1958 року

8. Reasoning моделі: OpenAI o1/o3, DeepSeek-R1

Недавно, компанії почали випускати спеціалізовані моделі для reasoning.

8.1 OpenAI o1

OpenAI o1 – це модель, натренована з посиленням навчання (reinforcement learning) на задачах, де потрібне глибоке reasoning.

OpenAI o1 має такі особливості: вона витрачає більше часу на reasoning перед тим, як дати відповідь, що дозволяє їй краще справлятися зі складними математичними та фізичними задачами. Однак вона є більш дорогою за стандартні моделі, що слід враховувати при виборі.

Приклад:

```
from openai import OpenAI
client = OpenAI()
response = client.chat.completions.create(
    model="o1",
    messages=[
        {"role": "user", "content": "Розв'яжіть: 3x + 5 = 20"}
    ]
)
```

8.2 DeepSeek-R1

DeepSeek-R1 – це модель, яка явно моделює процес reasoning з

«думками» (thoughts), що відкриті для користувача.

DeepSeek-R1 має унікальну архітектуру, в якій «думки» генеруються як явні проміжні кроки перед фінальною відповіддю, що дозволяє користувачу бачити весь процес reasoning та зрозуміти логіку моделі.

```
from deepseek import DeepSeekR1
model = DeepSeekR1()
response = model.reason(
    "Яка сума цифр числа 12345?"
)
# Вихід:
# Думка: Мне потрібно додати 1+2+3+4+5
# Розрахунок: 1+2=3, 3+3=6, 6+4=10, 10+5=15
# Відповідь: 15
```

9. Бенчмарки для reasoning

9.1 GSM8K (Grade School Math 8K)

Набір даних з 8500 задач з математики для 1-6 класів.

Приклад:

Запит: На фермі була певна кількість овець, потім народилось 20 новеньких ягнят.
Один за одним, сніговик з'їв 15 ягнят. Кількість ягнят на фермі склала 45.
Скільки ягнят було на фермі спочатку?
Розв'язання:
Ягнят, що народились: 20
Ягнят, які були з'їдені: 15
Поточна кількість: 45
Відповідь: 45 - 20 + 15 = 40

9.2 MATH (Mathematics Aptitude Test of Heuristics)

Більш важкі математичні задачі з олімпіад та стандартних тестів. 12500 задач.

Приклад (важкий):

Запит: Знайдіть y-перетин прямої, що проходить через (0, 3) та (-2, 5)

Розв'язання:

Нахил m = (5-3)/(-2-0) = 2/(-2) = -1

y = mx + b: 3 = -1(0) + b, таким чином b = 3

y-перетин: 3

9.3 ARC (AI2 Reasoning Challenge)

Задачі на множинний вибір з природничих наук, що вимагають складного reasoning.

9.4 HumanEval

Задачі програмування, де модель повинна написати коректний код.

Приклад:

Запит: Напишіть функцію has_close_elements(numbers, threshold),

яка повертає True, якщо будь-які два числа в списку на відстані менше ніж threshold.

Оцінювання: Функція повинна пройти всі тестові випадки

10. Обмеження та режими помилок CoT

10.1 “Spurious Correlations”

Модель може привести до правильної відповіді з хибним reasoning.

Приклад:

Запит: Якщо у Тома 3 яблука, а у Джейн їх в 2 рази більше, скільки яблук у них разом?

Дефектна CoT відповідь:

Том має 3 яблука. Джейн має двічі більше, тобто 9.

Разом: 9 яблук. ✓ (правильна відповідь)

Але reasoning крок містить помилку – модель помножила на 3 замість на 2 і отримала правильну відповідь галюцинуючи.

10.2 Hallucinated reasoning

Модель може придумати факти, які не вірні:

Запит: Яке число Пі?

CoT:

1. Пі – це відношення...
2. Вперше обчислено в древньому Єгипті...
3. Точне значення 3.14159... ✓

Але припущення про древній Єгипет є галюцинацією.

10.3 Ineffective reasoning

Для деяких задач reasoning не допомагає або вартує дорожче:

Запит: Яка столиця Франції?

Експенсивна CoT:

"Давайте розповсюджуватись... Франція – держава в Європі...

Європа має багато міст..."

Простої відповіді: "Париж" достатньо, і дешевше.

11. Структурований вихід та інструментальна допомога як reasoning

11.1 Структурований вихід JSON

Замість вільного тексту, попросіть модель подавати результати у структурованому форматі:

```
def structured_reasoning(question):
    prompt = f"""
    {question}
```

```
    Відповідай у форматі JSON:
    {{
        "steps": [
            {{"step": 1, "reasoning": "...", "result": "..."}},
            {{"step": 2, "reasoning": "...", "result": "..."}}
        ],
        "final_answer": "..."
    }}
    """
    response = llm.generate(prompt)
    result = json.loads(response)
    return result
```

11.2 Tool-Use як Reasoning

Дайте моделі доступ до інструментів (пошук, калькулятор):

```
def tool_augmented_reasoning(question):
    """LLM може використовувати інструменти як частину reasoning"""
    tools = {
        'search': search_engine,
        'calculate': calculator,
        'get_time': get_current_time
    }
    # Модель вирішує, який інструмент використовувати
    decision = llm.generate(f"What tool to use for: {question}")
    if 'search' in decision:
        result = tools['search'](extract_query(decision))
    elif 'calculate' in decision:
        result = tools['calculate'](extract_expression(decision))
    else:
        result = llm.generate(question)
```

```
return result
```

Контрольні запитання

1. Поясніть різницю між zero-shot, few-shot та Chain-of-Thought промптингом. Коли використовувати кожний?
2. Які результати дає Zero-Shot CoT? Чому простий промпт "Let's think step by step" так ефективний?
3. Як самоконсистентність (self-consistency) поліпшує якість reasoning? Наведіть приклади з бенчмарків.
4. Поясніть Tree of Thoughts. Як вона розрізняється від лінійної CoT?
5. Що таке ReAct? Як поєднує reasoning та acting?
6. Поясніть різницю між o1 та R1 моделями reasoning.
7. Назвіть чотири основних бенчмарки для оцінювання reasoning: GSM8K, MATH, ARC, HumanEval.
8. Які три основних режими помилок CoT? Наведіть приклади.
9. Як структурований вихід может поліпшити reasoning? Наведіть приклад JSON формату.
10. Як tool-use може служити як допоміжний механізм для reasoning?

Рекомендована література

1. Wei, J., Wang, X., Schuurmans, D., et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In Advances in Neural Information Processing Systems (NeurIPS).
2. Wang, X., Wei, J., Schuurmans, D., et al. (2023). Self-Consistency Improves Chain of Thought Reasoning in Language Models. In International Conference on Learning Representations (ICLR).

3. Yao, S., Yu, D., Zhao, J., et al. (2023). Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In Advances in Neural Information Processing Systems (NeurIPS).

4. Yao, S., Zhao, J., Yu, D., et al. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. In International Conference on Learning Representations (ICLR).

5. Cobbe, K., Kosaraju, V., Bavarian, M., et al. (2021). Training Verifiers to Solve Math Word Problems. arXiv preprint arXiv:2110.14168.

ЛЕКЦІЯ 15: Агентний штучний інтелект та системи з інструментами

Вступ

Один з найбільш захоплюючих напрямків у сучасній XAI – це розвиток агентних систем (agentic systems), де LLM не просто генерують текст, а активно вирішують задачі, використовуючи інструменти, пам'ять та планування. На відміну від простих чат-ботів, агенти можуть розкладати складні завдання на підзавдання, навчатися з помилок та взаємодіяти з зовнішніми системами. Цей розділ розглядає архітектуру агентів, популярні фреймворки та практичні приклади.

1. Від окремих запитів до мультиетапних агентів

1.1 Еволюція від простого до складного

Рівень 1: Простий запит-відповідь

Користувач: "Яке населення Франції?"
LLM: "Приблизно 68 мільйонів"

Рівень 2: Контекстні відповіді (RAG)

Користувач: "Яке населення Франції у 2024?"
Система: Пошук → Генерація з актуальним контекстом

Рівень 3: Агентний reasoning та планування

Користувач: "Порівняй населення Франції та Німеччини та розкажи, яке з них більше та на скільки"

Агент:

1. Пошук населення Франції.
2. Пошук населення Німеччини.
3. Розрахунок різниці.
4. Генерація відповіді.

Рівень 4: Складне мультиагентне рішення

Користувач: "Сплануйте вибір маршруту для поїздки з Парижа до Берліна"

Агент 1 (планування): Визначає ключові етапи.

Агент 2 (дослідження): Знаходить інформацію про маршрути.

Агент 3 (оптимізація): Вибирає найкращий варіант.

2. Основні компоненти агентних систем

2.1 Архітектура агента

На рис. 8 показано агентську архітектуру.

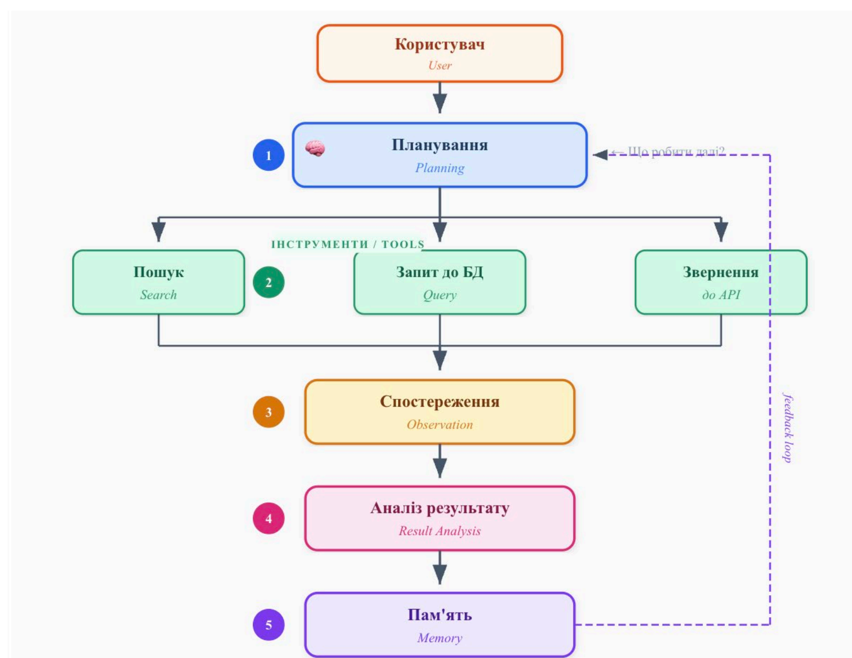

Рис.8 – Архітектура агента

2.2 Пам'ять в агентах

Коротка пам'ять (Short-term/Conversation Memory)

```
class ConversationMemory:
    def __init__(self, max_messages=10):
        self.messages = []
        self.max_messages = max_messages
    def add(self, role, content):
        self.messages.append({'role': role, 'content': content})
        if len(self.messages) > self.max_messages:
            self.messages.pop(0)  # FIFO
    def get_context(self):
        return '\n'.join([f"{m['role']}: {m['content']}"
                    for m in self.messages])
```

Довга пам'ять (Long-term/Episodic Memory)

```
class EpisodicMemory:
    def __init__(self, embedding_model):
        self.embedding_model = embedding_model
        self.memories = []  # Vector store
    def store(self, event, metadata=None):
        """Зберігає подію з ембедінгом"""
        embedding = self.embedding_model.embed(event)
        self.memories.append({
            'event': event,
            'embedding': embedding,
            'metadata': metadata,
            'timestamp': time.time()
        })
    def retrieve(self, query, top_k=3):
        """Пошук найбільш релевантних спогадів"""
```

```
        query_embedding = self.embedding_model.embed(query)
        scores = [cosine_similarity(query_embedding, m['embedding'])
                  for m in self.memories]
        top_indices = np.argsort(scores)[-top_k:]
        return [self.memories[i] for i in top_indices]
```

2.3 Інструменти (Tools)

Інструменти - це функції, які агент може викликати:

```
from typing import Callable, Dict, Any
class Tool:
    def __init__(self, name: str, description: str, func: Callable):
        self.name = name
        self.description = description
        self.func = func
    def execute(self, *args, **kwargs):
        return self.func(*args, **kwargs)
# Приклади інструментів
search_tool = Tool(
    name='search',
    description='Пошук в Інтернеті для інформації',
    func=search_engine.search
)
calculator_tool = Tool(
    name='calculate',
    description='Математичні обчислення',
    func=eval_math_expression
)
tools = {
    'search': search_tool,
    'calculate': calculator_tool
```

```
}
```

2.4 Петля виконання

```
class Agent:
    def __init__(self, llm, tools, memory):
        self.llm = llm
        self.tools = tools
        self.memory = memory
    def execute(self, task, max_steps=10):
        """Основна петля агента"""
        for step in range(max_steps):
            # 1. Планування: що робити далі?
            thought = self.llm.generate(
                f"Task: {task}\n"
                f"Memory: {self.memory.get_context()}\n"
                f"What's the next action?"
            )
            # 2. Вилучення дії
            action, params = parse_action(thought)
            # 3. Виконання дії
            if action in self.tools:
                observation = self.tools[action].execute(**params)
            else:
                observation = f"Unknown action: {action}"
            # 4. Зберігання в пам'ять
            self.memory.add('assistant', thought)
            self.memory.add('system', str(observation))
            # 5. Перевірка завершення
            if is_task_complete(observation):
                return observation
```

```
    return "Max steps exceeded"
```

3. Tool-Calling: функціональні виклики в LLM

Сучасні LLM підтримують “function calling” - структурований спосіб для моделі викликати функції.

3.1 OpenAI Function Calling

```
from openai import OpenAI
client = OpenAI()
# Визначте функції, які модель може викликати
tools = [
  {
    "type": "function",
    "function": {
      "name": "search",
      "description": "Пошук в Інтернеті",
      "parameters": {
        "type": "object",
        "properties": {
          "query": {
            "type": "string",
            "description": "Пошукова запит"
          }
        },
        "required": ["query"]
      }
    }
  },
  {
    "type": "function",
```

```
      "function": {
        "name": "calculator",
        "description": "Математичні обчислення",
        "parameters": {
          "type": "object",
          "properties": {
            "expression": {
              "type": "string",
              "description": "Математичний вираз (наприклад, '2+2*3')"
            }
          },
          "required": ["expression"]
        }
      }
  }
]
# Модель вирішує, які функції викликати
response = client.chat.completions.create(
  model="gpt-4",
  messages=[
    {"role": "user", "content": "Скільки буде 15 + 7?"}
  ],
  tools=tools,
  tool_choice="auto"
)
# Перевірте, чи модель закликала функцію
if response.choices[0].message.tool_calls:
  for tool_call in response.choices[0].message.tool_calls:
    function_name = tool_call.function.name
    function_args = json.loads(tool_call.function.arguments)
```

```python
    # Виконайте функцію
    if function_name == "calculator":
        result = eval(function_args['expression'])
        print(f"Результат: {result}")
```

3.2 Anthropic Tool Use

```python
from anthropic import Anthropic
client = Anthropic()
tools = [
    {
        "name": "search",
        "description": "Пошук інформації",
        "input_schema": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "Пошукова запит"
                }
            },
            "required": ["query"]
        }
    }
]
messages = [
    {"role": "user", "content": "Знайди інформацію про Клод"}
]
response = client.messages.create(
    model="claude-3-5-sonnet-20241022",
    max_tokens=1024,
```

```
    tools=tools,
    messages=messages
)
# Обробіть відповідь з tool use
if response.stop_reason == "tool_use":
    for block in response.content:
        if block.type == "tool_use":
            tool_name = block.name
            tool_input = block.input
            # Виконайте інструмент
            if tool_name == "search":
                result = search(tool_input['query'])
```

4. Фреймворки агентів

4.1 LangChain Agents

```
from langchain.agents import AgentExecutor, create_tool_calling_agent
from langchain_openai import ChatOpenAI
from langchain_community.tools import DuckDuckGoSearchRun
from langchain.tools import Tool
# Визначте LLM
llm = ChatOpenAI(model="gpt-4", temperature=0)
# Визначте інструменти
search_tool = DuckDuckGoSearchRun()
tools = [
    Tool(
        name="search",
        func=search_tool.run,
        description="Корисно для пошуку інформації в Інтернеті"
    )
]
```

```
# Створіть агента
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
prompt = ChatPromptTemplate.from_messages([
    ("system", "Ви - корисний помічник."),
    ("human", "{input}"),
    MessagesPlaceholder(variable_name="agent_scratchpad"),
])
agent = create_tool_calling_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
# Виконайте агента
result = agent_executor.invoke({
    "input": "Хто чемпіон світу з шахів у 2024 році?"
})
print(result['output'])
```

4.2 LlamaIndex Agents

```
from llama_index.agent import OpenAIAgent
from llama_index.tools import QueryEngineTool, ToolMetadata
# Визначте інструменти на основі пошукових індексів
document_search_tool = QueryEngineTool(
    query_engine=document_index.as_query_engine(),
    metadata=ToolMetadata(
        name="document_search",
        description="Пошук в документах компанії"
    )
)
# Створіть агента
agent = OpenAIAgent.from_tools(
    tools=[document_search_tool],
    llm=llm,
```

```
    verbose=True
)
# Виконайте агента
response = agent.chat("Яка політика компанії щодо отпусток?")
```

4.3 CrewAI (мультиагентні системи)

```
from crewai import Agent, Task, Crew
# Визначте агентів
researcher = Agent(
    role="Дослідник",
    goal="Розум глибоко про топіки",
    backstory="Досвідчений дослідник...",
    tools=[search_tool, web_scraper],
    verbose=True
)
writer = Agent(
    role="Письменник",
    goal="Писати компелінг контент",
    backstory="Талановитий письменник...",
    tools=[document_tool],
    verbose=True
)
# Визначте задачі
research_task = Task(
    description="Дослідник про ШІ та LLM",
    agent=researcher,
    expected_output="Детальний звіт"
)
writing_task = Task(
    description="Напишіть статтю на основі дослідження",
```

```
    agent=writer,
    expected_output="Готова стаття"
)
# Створіть екіпаж (crew)
crew = Crew(
    agents=[researcher, writer],
    tasks=[research_task, writing_task],
    verbose=True
)
# Виконайте
result = crew.kickoff()
```

4.4 AutoGen (мультиагентна оркестрація)

```
import autogen
# Визначте конфігурацію LLM
config_list = [{"model": "gpt-4", "api_key": "..."}]
# Визначте агентів
user_proxy = autogen.UserProxyAgent(
    name="User",
    code_execution_config={"work_dir": "coding"}
)
code_writer = autogen.AssistantAgent(
    name="CodeWriter",
    system_message="Ви - експерт з Python.",
    llm_config={"config_list": config_list}
)
code_reviewer = autogen.AssistantAgent(
    name="CodeReviewer",
    system_message="Ви - сувора код-ревьюер.",
    llm_config={"config_list": config_list}
```

```
)
# Визначте групові чати для колаборації
groupchat = autogen.GroupChat(
    agents=[user_proxy, code_writer, code_reviewer],
    messages=[],
    max_round=10
)
manager = autogen.GroupChatManager(groupchat=groupchat)
# Розпочніть конверсацію
user_proxy.initiate_chat(
    manager,
    message="Напишіть функцію, яка розраховує факторіал числа"
)
```

5. Model Context Protocol (MCP)

MCP – це відкритий протокол, розроблений Anthropic, для стандартизації доступу до інструментів:

5.1 Архітектура MCP

Приклад MCP архітектури зображено на рис. 9.

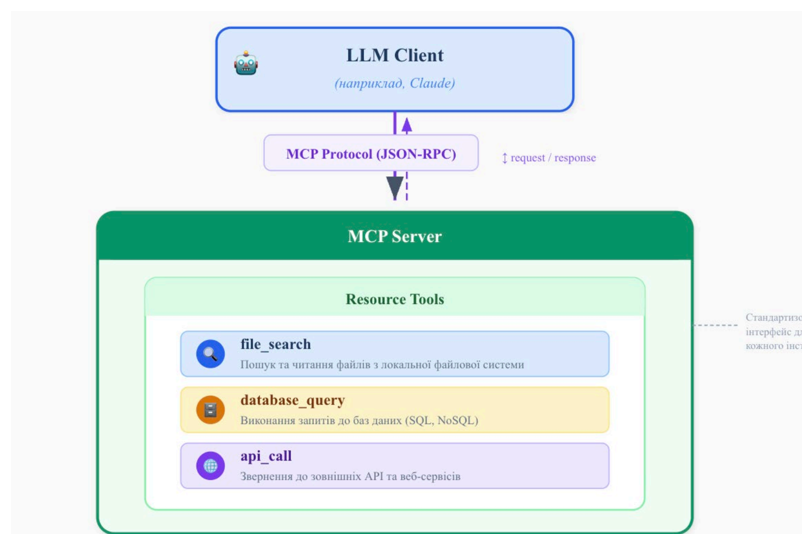

Рис. 9 – Архітектура MCP

5.2 MCP Server Implementation

```
import mcp
from mcp.server import Server, stdio_server
from mcp.types import Tool, TextContent
# Створіть MCP сервер
server = Server("my-tools")
# Визначте інструменти
@server.tool()
def search_documents(query: str) -> str:
    """Пошук в документах"""
    results = document_db.search(query)
    return f"Found {len(results)} results"
@server.tool()
def calculate(expression: str) -> str:
    """Математичні обчислення"""
    try:
        result = eval(expression)
        return str(result)
    except Exception as e:
        return f"Error: {e}"
# Запустіть сервер
if __name__ == "__main__":
    stdio_server(server)
```

5.3 MCP Client Usage

```
from mcp import ClientSession, StdioClientTransport
# Підключіться до MCP сервера
transport = StdioClientTransport(
    program="python",
    args=["-m", "my_mcp_server"]
```

```
)
async with ClientSession(transport) as session:
    # Отримайте доступні інструменти
    tools = await session.list_tools()
    # Викличте інструмент
    result = await session.call_tool(
        "search_documents",
        {"query": "ШІ та LLM"}
    )
    print(result)
```

6. Шаблони дизайну агентів

6.1 ReAct (Reasoning + Acting)

Агент обирає між reasoning та acting кроками:

Думка 1: Мне потрібна інформація про населення.

Дія 1: Пошук "населення країн 2024".

Спостереження: [результати пошуку].

Думка 2: Тепер порівняю дані.

Дія 2: Обробка результатів.

Спостереження: Франція 68M, Німеччина 84M.

6.2 Plan-and-Execute

Спочатку планує весь план, потім виконує:

Планування:

1. Знайти інформацію про Францію.
2. Знайти інформацію про Німеччину.
3. Порівняти населення.

Виконання:

[Крок 1] → [Крок 2] → [Крок 3].

6.3 Reflexion

Агент перевіряє свою роботу та виправляє помилки:

Спроба 1: Дайте неправильну відповідь.

Рефлексія: "Я допустив помилку, оскільки...".

Спроба 2: Виправте відповідь.

7. Мультиагентні системи

7.1 Співробітництво агентів

```
class CollaborativeAgents:
    def __init__(self):
        self.agents = {
            'researcher': Researcher(),
            'analyzer': Analyzer(),
            'writer': Writer()
        }
        self.shared_memory = SharedMemory()
    def execute(self, task):
        # Фаза 1: Дослідження
        research_result = self.agents['researcher'].research(task)
        self.shared_memory.store('research', research_result)
        # Фаза 2: Аналіз
        analysis = self.agents['analyzer'].analyze(research_result)
        self.shared_memory.store('analysis', analysis)
        # Фаза 3: Написання
        final_output = self.agents['writer'].write(
            research_result,
            analysis
        )
        return final_output
```

7.2 Дебат між агентами

```
class DebatingAgents:
    def __init__(self):
        self.agent_a = Agent("Pro")
        self.agent_b = Agent("Con")
        self.judge = Agent("Judge")
    def debate(self, topic, rounds=3):
        arguments = []
        for round in range(rounds):
            # Pro аргумент
            pro_arg = self.agent_a.argue(topic, arguments)
            arguments.append(('pro', pro_arg))
            # Con аргумент
            con_arg = self.agent_b.argue(topic, arguments)
            arguments.append(('con', con_arg))
        # Судження
        winner = self.judge.judge(arguments)
        return winner, arguments
```

8. Безпека, надійність та гарантії для агентів

8.1 Guardrails

```
class SafeAgent:
    def __init__(self, agent, guardrails=None):
        self.agent = agent
        self.guardrails = guardrails or []
    def execute(self, task):
        # Перевірьте завдання перед виконанням
        for guardrail in self.guardrails:
            if guardrail.blocks(task):
```

```
            return f"Task blocked: {guardrail.reason}"
        # Виконайте агента
        result = self.agent.execute(task)
        # Перевірьте результат
        for guardrail in self.guardrails:
            result = guardrail.filter(result)
        return result
# Приклади guardrails
class NoHarmfulContent:
    def blocks(self, task):
        harmful_keywords = ['вибух', 'отрута', 'вперед']
        return any(kw in task.lower() for kw in harmful_keywords)
class NoPersonalDataExfiltration:
    def filter(self, result):
        # Видаліть особисті дані з результату
        import re
        result = re.sub(r'\b\d{3}-\d{2}-\d{4}\b', '[SSN]', result)
        return result
```

8.2 Trace та Logging

```
import logging
from datetime import datetime
class TracedAgent:
    def __init__(self, agent):
        self.agent = agent
        self.logger = logging.getLogger('agent')
    def execute(self, task):
        trace = {
            'timestamp': datetime.now(),
            'task': task,
```

```
        'steps': []
    }
    def log_step(action, params, result):
        trace['steps'].append({
            'action': action,
            'params': params,
            'result': result,
            'timestamp': datetime.now()
        })
    # Виконайте з логуванням
    self.agent.execute_with_callback(task, log_step)
    # Збережіть трейс
    self.logger.info(f"Trace: {trace}")
    return trace
```

9. Моніторинг та оцінювання поведінки агентів

9.1 Метрики для агентів

1. **Task Completion Rate**: Яка частка завдань завершена успішно?
2. **Steps to Completion**: Скільки кроків потрібно для завершення?
3. **Tool Usage**: Які інструменти найбільше використовуються?
4. **Error Rate**: Як часто агент припускається помилок?
5. **Cost**: Скільки API запитів було зроблено?

9.2 Оцінювання

```
def evaluate_agent(agent, test_cases):
    results = {
        'completed': 0,
        'failed': 0,
        'total_steps': 0,
```

```
        'total_cost': 0,
        'errors': []
    }
    for task, expected_output in test_cases:
        try:
            result = agent.execute(task)
            if result == expected_output:
                results['completed'] += 1
            else:
                results['failed'] += 1
            results['total_steps'] += agent.steps_taken
            results['total_cost'] += agent.cost
        except Exception as e:
            results['errors'].append(str(e))
    # Розраховуйте метрики
    success_rate = results['completed'] / len(test_cases) * 100
    avg_steps = results['total_steps'] / len(test_cases)
    return {
        'success_rate': success_rate,
        'avg_steps': avg_steps,
        'total_cost': results['total_cost'],
        'errors': results['errors']
    }
```

10. Розгортування агентів у виробництво

10.1 API обгортка агента

```
from fastapi import FastAPI, BackgroundTasks
from pydantic import BaseModel
app = FastAPI()
class TaskRequest(BaseModel):
```

```
    task: str
    priority: str = "normal"
@app.post("/execute")
async def execute_task(request: TaskRequest, background_tasks:
BackgroundTasks):
    """Запустіть задачу асинхронно"""
    task_id = str(uuid.uuid4())
    background_tasks.add_task(
        execute_agent_task,
        task_id,
        request.task
    )
    return {"task_id": task_id, "status": "processing"}
@app.get("/status/{task_id}")
async def get_status(task_id: str):
    """Отримайте статус задачи"""
    return task_registry.get(task_id)
def execute_agent_task(task_id: str, task: str):
    """Виконайте агента в фоні"""
    try:
        result = agent.execute(task)
        task_registry.set(task_id, {
            'status': 'completed',
            'result': result
        })
    except Exception as e:
        task_registry.set(task_id, {
            'status': 'failed',
            'error': str(e)
        })
```

10.2 Масштабування агентів

```yaml
# docker-compose.yml для масштабування агентів
version: '3.8'
services:
 agent-worker:
  image: my-agent:latest
  environment:
   - REDIS_URL=redis://redis:6379
  deploy:
   replicas: 5
  volumes:
   - ./tools:/app/tools
  depends_on:
   - redis
 redis:
  image: redis:7
  ports:
   - "6379:6379"
 task-queue:
  image: celery-broker:latest
  environment:
   - BROKER_URL=redis://redis:6379
```

Контрольні запитання

1. Поясніть еволюцію від простих запит-відповідей до мультиагентних систем. Наведіть рівні складності.

2. Які основні компоненти архітектури агента? Поясніть кожен компонент.

3. Як розрізняються коротка та довга пам’ять в агентах? Наведіть

приклади.

4. Що таке tool-calling та як воно реалізується у OpenAI та Anthropic?

5. Порівняйте LangChain, LlamaIndex, CrewAI та AutoGen. Які їх основні відмінності?

6. Поясніть Model Context Protocol (MCP). Як стандартизує доступ до інструментів?

7. Назвіть три шаблони дизайну агентів: ReAct, Plan-and-Execute та Reflexion.

8. Як мультиагентні системи можуть співпрацювати та дебатувати?

9. Які guardrails необхідні для безпеки агентів? Як логувати та трасувати поведінку?

10. Як розгорнути агента в виробництво з асинхронним виконанням та масштабуванням?

Рекомендована література

1. Yao, S., Zhao, J., Yu, D., et al. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. In International Conference on Learning Representations (ICLR).

2. LangChain Documentation: https://python.langchain.com/docs/

3. LlamaIndex Documentation: https://docs.llamaindex.ai/

4. AutoGen Documentation: https://microsoft.github.io/autogen/

5. Anthropic Model Context Protocol: https://modelcontextprotocol.io/

6. OpenAI Function Calling: https://platform.openai.com/docs/guides/function-calling

7. Woof, S., Kohn, J., & Paranjape, B. (2024). Agent Design Patterns in Large Language Models. arXiv preprint.

ЗАГАЛЬНІ РЕКОМЕНДАЦІЇ

Матеріали для вивчення за розділами

Розділ 1: Класичний NLP та текстові представлення - Необхідні: - Jurafsky & Martin, "Speech and Language Processing" (розділи 2-3) - Goldberg, "Neural Network Methods for Natural Language Processing" (розділи 1-2) - HuggingFace документація: Tokenizers - Attention Is All You Need (Vaswani et al., 2017) - BERT: Pre-training of Deep Bidirectional Transformers (Devlin et al., 2018) - HuggingFace: Transformers документація - Рекомендовані: - YouTube: "NLP with Transformers" from 3Blue1Brown - arXiv: Word2Vec (Mikolov et al., 2013) - Статті про BPE та WordPiece - The Illustrated Transformer (Jay Alammar) - Illustrated BERT, ELMo, and co. - YouTube: Attention Mechanisms by StatQuest

Розділ 2: Великі мовні моделі (LLM) - Необхідні: - Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (T5, Raffel et al., 2019) - BART: Denoising Sequence-to-Sequence Pre-training (Lewis et al., 2019) - HuggingFace: Seq2Seq modeling guide - How to Finetune BERT for Text Classification? (Sun et al., 2019) - Direct Preference Optimization (DPO, Rafailov et al., 2023) - Training language models to follow instructions with human feedback (InstructGPT, Ouyang et al., 2022) - TRL (Transformers Reinforcement Learning) документація - Рекомендовані: - BLEU: a Method for Automatic Evaluation of Machine Translation (Papineni et al., 2002) - ROUGE: A Package for Automatic Evaluation of Summarization (Lin, 2004) - Papers from ACL, EMNLP на теми summarization и translation - QLoRA: Efficient Finetuning of Quantized LLMs (Dettmers et al., 2023) - Parameter-Efficient Transfer Learning for NLP (Adapter-Modules, Houlsby et al., 2019) - GRPO paper та реалізація

Розділ 3: Генерація з доповненням пошуком (RAG) - Необхідні: - Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks (RAG, Lewis et al., 2020) - Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks (Reimers & Gurevych, 2019) - LangChain та LlamaIndex документація

- ReAct: Synergizing Reasoning and Acting in Language Models (Yao et al., 2022) - LangChain: документація та агент розділи - MCP (Model Context Protocol) specification та examples - Рекомендовані: - Dense Passage Retrieval for Open-Domain Question Answering (DPR, Karpukhin et al., 2020) - Contrastive Learning of General-Purpose Sentence Embeddings (Thakur et al., 2020) - Toolformer: Language Models Can Teach Themselves to Use Tools (Schick et al., 2023) - Agent Foundations та Multi-Agent Systems

Онлайн курси та ресурси

Безкоштовні курси:

Для навчання доступні безплатні онлайн-курси, зокрема курс "Practical Deep Learning" від Fast.ai, який включає модуль з NLP, а також курс CS224N: NLP with Deep Learning від Стенфорда з лекціями доступними online. Hugging Face пропонує власний безкоштовний курс на своєму веб-сайті (huggingface.co/course), а DeepLearning.AI надає короткі курси з тем LLMs, Vector Databases та RAG.

Документація та посібники:

Для отримання деталей та рекомендацій доступна офіційна документація HuggingFace для бібліотек transformers, datasets, tokenizers та TRL, а також PyTorch з його tutorials та документацією. На сайті Paper with Code можна знайти NLP розділ з практичними реалізаціями, а GitHub містить популярні репозиторії з реалізаціями різних моделей.

Практика:

Для практичної роботи та тестування можна використовувати Kaggle, де доступні NLP задачі та датасети, Paper with Code з дослідницькими статтями та бенчмарками, Google Colab для безплатного доступу до GPU при експериментуванні, а також Hugging Face Spaces для взаємодії з готовими демонстраційними проектами.

Поради для ефективного вивчення

1. **Читайте дослідницькі статті, не просто блоги:**

– Дослідницькі статті дають більш глибоке розуміння чим блоги;

– Почніть з abstract та introduction для швидкого огляду;

– Потім читайте методологію та результати;

– Math деталі можна пропустити на перший раз.

2. **Реалізуйте алгоритми самостійно:**

– Наприклад, реалізуйте Transformer блок з нуля;

– Це значно краще допоможе вам розуміти, ніж просто читання;

– Потім можете використовувати готові реалізації з HuggingFace.

3. **Експериментуйте з гіперпараметрами:**

– Вивчайте, як змінюються результати при зміні learning rate, batch size, model size;

– Це розвиває інтуїцію щодо моделей;

– Користуйтесь Weights & Biases або TensorBoard для трекування.

4. **Прочитайте чужий код:**

– Читання хорошого коду також є навичкою;

– Дивіться реалізації з HuggingFace, OpenAI, Meta;

– Спробуйте розуміти design decisions.

5. **Займайтесь прикладними проектами:**

– Побудуйте власний NLP проект поза лабораторними

роботами;

– Наприклад: sentiment analysis для соціальних мереж, класифікація статей, генерація бота для вашої улюбленої області;

– Це краще за просто вивчення, бо ви вирішуєте реальні задачі.

6. **Спілкуйтесь з однолітками:**

– Обговорюйте складні концепції з іншими студентами;

– Навчання через пояснення іншим дуже ефективне;

– Організуйте study groups.

7. **Стежте за останніми розробками:**

– NLP розвивається дуже швидко;

– Слідкуйте за новими моделями та методами на arXiv, Twitter/X, Reddit;

– Але не потрібно дотримуватися всіх новинок, зосередьтесь на фундаменталі.

8. **Займайтесь з GPU:**

– Для дійсного тренування моделей потрібна GPU;

– Користуйтесь Google Colab (безплатно), AWS, або придбайте доступ до хмарних сервісів;

– Це дозволить вам тренувати реальні моделі, а не просто експериментувати на малих розмірах.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

Основні підручники

1. Jurafsky, D., & Martin, J. H. (2021). **Speech and Language Processing** (3rd ed.). https://web.stanford.edu/~jurafsky/slp3/. — Фундаментальний підручник з NLP, охоплює історію та основні методи

2. Goldberg, Y. (2017). **Neural Network Methods for Natural Language Processing**. Morgan & Claypool Publishers. — Детальне пояснення нейромережевих методів для NLP

3. Goodfellow, I., Bengio, Y., & Courville, A. (2016). **Deep Learning**. MIT Press. — Основний підручник з глибокого навчання

4. Karpukhin, V., & Lewis, P. (2021). **Dense Passage Retrieval for Open-Domain Question Answering**. arXiv:2004.04906. — Класичний підхід до RAG

Ключові дослідницькі статті

5. Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). **Attention is All You Need**. arXiv:1706.03762. — Революційна стаття про архітектуру Transformer

6. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**. arXiv:1810.04805. — Основна претренована модель двосторонніх трансформерів

7. Raffel, C., Shazeer, N., Roberts, A., et al. (2019). **Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer**. arXiv:1910.10683. — Модель T5 для seq2seq задач

8. Lewis, P., Perez, E., Piktus, A., et al. (2020). **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks**. arXiv:2005.11401. —

Класична стаття про RAG

9. Brown, A., Mann, B., Ryder, N., et al. (2020). **Language Models are Few-Shot Learners**. arXiv:2005.14165. — GPT-3, демонструє few-shot learning

10. Ouyang, L., Wu, J., Jiang, X., et al. (2022). **Training language models to follow instructions with human feedback**. arXiv:2203.02155. — InstructGPT, основа для alignment методів

11. Rafailov, R., Sharma, A., Mitchell, E., et al. (2023). **Direct Preference Optimization: Your Language Model is Secretly a Reward Model**. arXiv:2305.18290. — DPO для alignment

12. Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2023). **QLoRA: Efficient Finetuning of Quantized LLMs**. arXiv:2305.14314. — Параметрично-ефективне fine-tuning

13. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). **Efficient Estimation of Word Representations in Vector Space**. arXiv:1301.3781. — Word2Vec, базові ембедінги слів

14. Prabhumoye, S., Tsvetkov, Y., Salakhutdinov, R., & Black, A. W. (2020). **Style Transfer as Unsupervised Machine Translation**. arXiv:1808.09381. — Приклад seq2seq застосування

15. Yao, S., Yu, D., Zhao, J., et al. (2022). **ReAct: Synergizing Reasoning and Acting in Language Models**. arXiv:2210.03629. — Reasoning + Acting для агентів

16. Lewis, M., Liu, Y., Goyal, N., et al. (2019). **BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension**. arXiv:1910.13461. — BART модель

17. Reimers, N., & Gurevych, I. (2019). **Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks**. arXiv:1908.10084. — Sentence embeddings для RAG

18. Schick, T., Dwivedi-Yu, J., Dessì, R., et al. (2023). **Toolformer: Language Models Can Teach Themselves to Use Tools**. arXiv:2302.04761. — LLM з інструментами

19. Houlsby, N., Giurgiu, A., Jastrzebski, S., et al. (2019). **Parameter-Efficient Transfer Learning for NLP**. arXiv:1902.00751. — Adapter modules для efficient fine-tuning

20. Mosbach, M., Andriushchenko, M., & Klakow, D. (2020). **On the Stability of Fine-tuning BERT: Misconceptions, Explanations, and Strong Baselines**. arXiv:2006.04884. — Аналіз stability при fine-tuning

Практичні ресурси та документація

21. Hugging Face Team. **Transformers Documentation**. https://huggingface.co/docs/transformers/. — Офіційна документація бібліотеки Transformers

22. Hugging Face Team. **NLP Course**. https://huggingface.co/course/. — Безплатний онлайн курс по NLP з Hugging Face

23. LangChain Documentation. https://python.langchain.com/. — Документація для побудови LLM додатків

24. LlamaIndex Documentation. https://docs.llamaindex.ai/. — Документація для RAG та індексування

25. PyTorch Documentation. https://pytorch.org/docs/. — Офіційна документація глибокого фреймворку PyTorch

26. Weights & Biases. **Experiment Tracking Guide**. https://docs.wandb.ai/. — Інструмент для логування та аналізу експериментів

27. Papers with Code. https://paperswithcode.com/. — Сайт з дослідницькими статтями та їх реалізаціями

28. arXiv. https://arxiv.org/list/cs.CL/recent. — Репозиторій дослідницьких статей з NLP

Додаткові матеріали

29. Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002). **BLEU: A Method for Automatic Evaluation of Machine Translation**. Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, 311-318. — Основна метрика для оцінки перекладу

30. Lin, C. Y. (2004). **ROUGE: A Package for Automatic Evaluation of Summarization**. In Text Summarization Branches Out: Proceedings of the ACL-04 Workshop, 74-81. — Основна метрика для оцінки реферування