Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Вдосконалення великої мовної моделі (Large Language Model - LLM) для опрацювання тексту засобами штучного інтелекту

Виконав(ла): студент(ка) 4 курсу, групи СТс-41
спеціальності 126 Інформаційні системи та технології
(шифр і назва спеціальності)

		Сороцький Р. М.
	(підпис)	(прізвище та ініціали)
Керівник		Марценюк В.П.
	(підпис)	(прізвище та ініціали)
Нормоконтроль		
	(підпис)	(прізвище та ініціали)
Завідувач кафедри		Боднарчук І.О.
	(підпис)	(прізвище та ініціали)
Рецензент		
	(підпис)	(прізвище та ініціали)

Тернопіль
2024

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«21» червня 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 126 Інформаційні системи та технології
(шифр і назва спеціальності)

студенту Сороцький Роман Михайлович
(прізвище, ім'я, по батькові)

1. Тема роботи **Вдосконалення великої мовної моделі (Large Language Model - LLM) для опрацювання тексту засобами штучного інтелекту**

Керівник роботи **д.т.н., проф. Марценюк В.П.**
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*29*» *квітня* 2024 року № *4/7-471*
2. Термін подання студентом завершеної роботи 21 червня 2024 р.
3. Вихідні дані до роботи Літературні джерела з тематики роботи

4. Зміст роботи (перелік питань, які потрібно розробити)
ВСТУП 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 1.1 Переваги та обмеження LLM 1.2 Принцип функціонування LLM 1.3 Моделі-трансформери 1.4 Архітектура LLM 1.5 Типи моделей тансформерів та моделі навчання 2 НАЛАШТУВАННЯ МОДЕЛІ LLM 2.1 Точне налаштування великої мовної моделі (LLM) на спеціальному наборі даних за допомогою QLoRA 2.2 Методи тонкого налаштування 2.3 Що таке LoRa? 2.4 Що таке квантований LoRA (QLoRA)? 3 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ ВИСНОВОК ПЕРЕЛІК ДЖЕРЕЛ ДОДАТКИ

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	Сенчишин В. С., доцент кафедри МТ		

7. Дата видачі завдання 29 січня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	30.01.2024	*Виконано*
2.	Підбір джерел про плагіни та шаблони WordPress	31.01.2024-03.02.2024	*Виконано*
3.	Опрацювання джерел по темі кваліфікаційної роботи	04.02.2024-06.02.2024	*Виконано*
4.	Виконання дослідження щодо шаблонів та плагінів WordPress	07.02.2024-11.02.2024	*Виконано*
5.	Розроблення сайту на WordPress	21.05.2024-02.06.2024	*Виконано*
6.	Оформлення розділу «Аналіз предметної області та постановка завдання»	03.06.2024-05.06.2024	*Виконано*
7.	Оформлення розділу «Проєктна частина»	06.06.2024-08.06.2024	*Виконано*
8.	Оформлення розділу «Практична частина»	09.06.2024-11.06.2024	*Виконано*
9.	Виконання завдання до підрозділу «Безпека життєдіяльності»	12.06.2024-14.06.2024	*Виконано*
10.	Виконання завдання до підрозділу «Основи охорони праці»	14.06.2024-16.06.2024	*Виконано*
11.	Оформлення кваліфікаційної роботи	16.06.2024-17.06.2024	*Виконано*
12.	Нормоконтроль	18.06.2024-19.06.2024	*Виконано*
13.	Перевірка на плагіат	20.06.2024	*Виконано*
14.	Попередній захист кваліфікаційної роботи	21.06.2024	*Виконано*
15.	Захист кваліфікаційної роботи	24.06.2024	

Студент _____ Сороцький Р. М.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Марценюк В. П.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Вдосконалення великої мовної моделі (Large Language Model - LLM) для опрацювання тексту засобами штучного інтелекту // Кваліфікаційна робота освітнього рівня "Бакалавр" // Сороцький Роман Михайлович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп’ютерних наук, група СТс-41 // Тернопіль, 2024 // с. –48, рис. – 10, кресл. – 10, бібліогр. – 15.

Ключові слова: NLP, LLM, покращення LLM; алгоритми навчання; моделі архітектури.

Покращення моделей великих мовних моделей (LLM) є ключовим напрямком сучасних досліджень в галузі штучного інтелекту. Ці моделі, такі як GPT-4, демонструють значні успіхи у генерації тексту, розумінні природної мови та інших завданнях, проте завжди є простір для вдосконалення. Ось кілька напрямків, над якими працюють дослідники та інженери для покращення LLM-моделей:

1. Підвищення якості даних для навчання: Якість вихідних даних має вирішальне значення для ефективності LLM. Використання більш чистих, структурованих та різноманітних наборів даних допомагає моделям краще розуміти контекст і створювати більш точні відповіді. Крім того, активне навчання, де модель взаємодіє з користувачами для збирання зворотного зв'язку, також сприяє покращенню.

2. Оптимізація архітектури моделей: Зміни в архітектурі моделей можуть значно вплинути на їх продуктивність. Наприклад, введення нових типів шарів або механізмів уваги, які дозволяють моделі краще фокусуватися на важливих частинах вхідного тексту, може покращити результати. Також досліджуються методи зменшення розміру моделей без втрати якості, що знижує витрати на обчислення.

3. Покращення алгоритмів навчання: Використання вдосконалених методів оптимізації та регуляризації може допомогти у навчанні більш стабільних і

узагальнюючих моделей. Наприклад, методи, такі як Dropout і Data Augmentation, можуть запобігати перенавчанню і підвищувати загальну продуктивність моделі.

4. Інтеграція знань та логіки: Одним із способів підвищення ефективності LLM є інтеграція зовнішніх знань та логічних висновків. Використання баз знань та онтологій може допомогти моделям краще розуміти контекст і здійснювати більш точні прогнози.

5. Зниження упереджень: LLM часто можуть виявляти упередження, присутні у вихідних даних. Розробка методів для виявлення та зниження таких упереджень є важливим аспектом покращення моделей. Це включає як попередню обробку даних, так і розробку алгоритмів, які враховують етичні та соціальні аспекти.

6. Застосування у різних доменах: Покращення LLM також полягає у їх адаптації для специфічних галузей, таких як медицина, право або фінанси. Спеціалізовані моделі, навчені на доменних даних, можуть показувати кращі результати у відповідних контекстах.

Таким чином, покращення великих мовних моделей є комплексним завданням, що охоплює як технологічні аспекти, так і етичні міркування. Постійний прогрес у цій галузі сприяє створенню більш потужних, точних та етично відповідальних моделей, які можуть бути корисними в різних сферах життя.

ANNOTATION

Improvement of the Large Language Model (LLM) for Text Processing by Means of Artificial Intelligence // Qualification work of the educational level "Bachelor" // Sorotskyi Roman// Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer Science, Group CTc-41 // Ternopil, 2024 // p. – 48, fig. – 10, references – 15, posters – 10.

Keywords: NLP, LLM, LLM tuning; learning algorithms; models architecture.

Improving large language models (LLMs) is a key focus in the field of artificial intelligence research. These models, such as GPT-4, demonstrate significant success in text generation, natural language understanding, and other tasks, but there is always room for enhancement. Here are several areas researchers and engineers are working on to improve LLMs:

1. Enhancing Training Data Quality: The quality of the training data is crucial for the effectiveness of LLMs. Using cleaner, more structured, and diverse datasets helps models better understand context and generate more accurate responses. Additionally, active learning, where the model interacts with users to gather feedback, also contributes to improvement.

2. Optimizing Model Architecture: Changes in model architecture can significantly impact performance. For instance, introducing new types of layers or attention mechanisms that allow the model to better focus on important parts of the input text can improve outcomes. Researchers are also exploring methods to reduce model size without compromising quality, thereby lowering computational costs.

3. Improving Training Algorithms: Utilizing advanced optimization and regularization methods can help train more stable and generalizable models. Techniques like dropout and data augmentation can prevent overfitting and enhance overall model performance.

4. Integrating Knowledge and Logic: One way to enhance LLMs is by integrating external knowledge and logical reasoning. Using knowledge bases and ontologies can help models better understand context and make more accurate predictions.

5. Reducing Bias: LLMs often exhibit biases present in the training data. Developing methods to detect and mitigate these biases is an important aspect of improving models. This includes both preprocessing data and designing algorithms that consider ethical and social factors.

6. Application in Various Domains: Improving LLMs also involves adapting them for specific fields such as medicine, law, or finance. Specialized models trained on domain-specific data can deliver better results in their respective contexts.

Thus, improving large language models is a multifaceted task that encompasses both technological and ethical considerations. Ongoing progress in this field contributes to the creation of more powerful, accurate, and ethically responsible models that can be beneficial in various areas of life.

ЗМІСТ

ВСТУП

Великі мовні моделі (LLM), які демонструють розширені можливості та складні рішення, зробили революцію в галузі обробки природної мови. Натреновані на обширних наборах текстових даних, ці моделі відмінно справляються з такими завданнями, як створення тексту, переклад, узагальнення та відповіді на запитання. Незважаючи на свою потужність, LLMs не завжди можуть узгоджуватися з конкретними завданнями чи доменами. Питаннями досліджень великих мовних моделей займались в ТНТУ, наприклад, в роботі [1].

У даній роботі ми дослідимо, як тонке налаштування LLM може значно покращити продуктивність моделі, зменшити витрати на навчання та забезпечити більш точні результати, що залежать від контексту.

Велика мовна модель (LLM) – це тип програми штучного інтелекту (ШІ), яка, серед інших завдань, може розпізнавати та генерувати текст. LLM навчаються на величезних наборах даних – звідси й назва «великий». LLM побудовані на основі машинного навчання: зокрема, типу нейронної мережі, яка називається моделлю трансформера.

Простіше кажучи, LLM – це комп’ютерна програма, яка набрала достатньо прикладів, щоб мати можливість розпізнавати та інтерпретувати людську мову чи інші типи складних даних. Якість зразків впливає на те, наскільки добре LLM вивчатимуть природну мову, тому програмісти LLM можуть використовувати більш ретельно підібраний набір даних.

LLM використовують тип машинного навчання, який називається глибоким навчанням, щоб зрозуміти, як символи, слова та речення функціонують разом. Глибоке навчання передбачає імовірнісний аналіз неструктурованих даних, що зрештою дозволяє моделі глибокого навчання розпізнавати відмінності між фрагментами вмісту без втручання людини.

Будь-який великий, складний набір даних може бути використаний для навчання LLM, включаючи мови програмування. Деякі LLM можуть допомогти програмістам написати код. Вони можуть писати функції за запитом – або,

отримавши деякий код як відправну точку, вони можуть закінчити написання програми. LLM також можна використовувати в таких задачах:

- аналіз емоційного стану;
- дослідження ДНК;
- обслуговування клієнтів;
- чат-боти;
- онлайн пошук.

Прикладами реальних LLM є ChatGPT (від OpenAI), Bard (Google), Llama (Meta) і Bing Chat (Microsoft). Іншим прикладом є Copilot від GitHub, але для програмного коду, а не природної людської мови.

1АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Переваги та обмеження LLM

Ключовою характеристикою LLM є їх здатність відповідати на непередбачувані запити. Традиційна комп'ютерна програма отримує команди у прийнятому синтаксисі або з певного набору вхідних даних від користувача. Відеогра має обмежений набір кнопок, програма має кінцевий набір атрибутів, які користувач може клацати або вводити, а мова програмування складається з точних операторів if/then.

Навпаки, LLM може реагувати на природну людську мову та використовувати аналіз даних, щоб відповісти на неструктуроване запитання чи підказку у спосіб, який має сенс. У той час як звичайна комп'ютерна програма не розпізнає підказку на кшталт «Які чотири найвидатніші рок-гурти в історії?», а LLM може відповісти списком із чотирьох таких гуртів і досить переконливим обґрунтуванням того, чому вони найкращі.

Проте з точки зору інформації, яку вони надають, LLM можуть бути настільки ж надійними, наскільки надійними є дані, які вони отримують. Якщо надати неправдиву інформацію, вони нададуть неправдиву інформацію у відповідь на запити користувачів. LLM також іноді «галюцинують»: вони створюють фальшиву інформацію, коли не можуть дати точну відповідь. Наприклад, у 2022 році випуск новин Fast Company запитав ChatGPT про попередній фінансовий квартал компанії Tesla; у той час як ChatGPT надав послідовну новинну статтю у відповідь, багато інформації в ній було вигадано.

З точки зору безпеки, призначені для користувача програми на основі LLM так само схильні до помилок, як і будь-які інші програми. LLM також можна маніпулювати за допомогою зловмисних даних, щоб забезпечити певні типи відповідей замість інших, включаючи відповіді, які є небезпечними або неетичними.

На додачу одна з проблем безпеки LLM полягає в тому, що користувачі можуть завантажувати в них безпечні конфіденційні дані, щоб підвищити власну продуктивність. Але LLM використовують отримані вхідні дані для подальшого

навчання своїх моделей, і вони не створені як безпечні сховища; вони можуть розкривати конфіденційні дані у відповідь на запити інших користувачів.

1.2 Принцип функціонування LLM

На базовому рівні програми LLM побудовані на машинному навчанні. Машинне навчання є підмножиною штучного інтелекту, і воно стосується практики передачі в програму великих обсягів даних, щоб навчити програму визначати особливості цих даних без втручання людини.

LLM використовують тип машинного навчання, який називається глибоким навчанням. Моделі глибокого навчання можуть по суті навчитися розпізнавати відмінності без втручання людини, хоча зазвичай необхідне деяке тонке налаштування людиною.

Глибоке навчання використовує ймовірність, щоб «навчатися». Наприклад, у реченні «Швидка бура лисиця перестрибнула ледачого пса» найчастіше зустрічаються літери «и» і «а», кожна з яких зустрічається по чотири рази. З цього модель глибокого навчання може зробити (правильний) висновок, що ці символи є одними з найбільш ймовірних для появи в тексті.

Реально кажучи, модель глибокого навчання не може нічого зробити з одного речення. Але після аналізу трильйонів речень він може навчитися достатньо, щоб передбачити, як логічно закінчити неповне речення, або навіть створити власні речення.

Тонке налаштування LLM передбачає додаткове навчання попередньо існуючої моделі, яка раніше отримала шаблони та функції з великого набору даних, використовуючи менший набір даних для конкретної області. У контексті «точного налаштування LLM» LLM позначає «модель великої мови», таку як серія GPT від OpenAI. Цей підхід має важливе значення, оскільки навчання великої мовної моделі з нуля є дуже ресурсомістким з точки зору як обчислювальної потужності, так і часу. Використання наявних знань, вбудованих у попередньо навчену модель, дозволяє досягти високої продуктивності в конкретних завданнях із суттєво зменшеними вимогами до даних і обчислень (див. рис. 1.1).

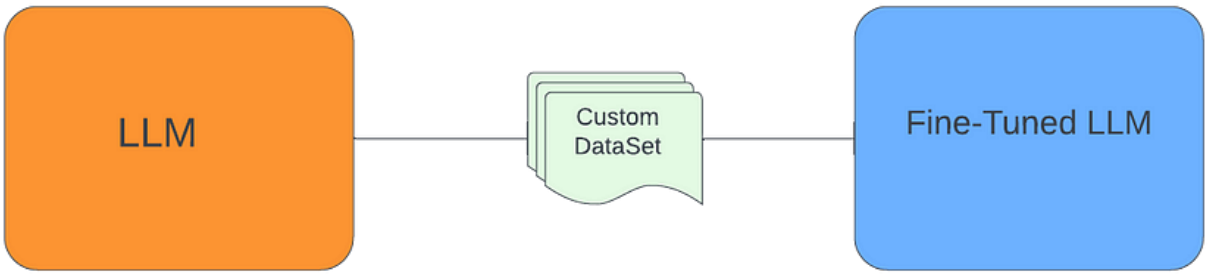

Рисунок 1.1 – Тонка настройка LLM

Отже, моделі LLM додатково навчаються за допомогою налаштування: вони точно налаштовуються за допомогою механізмів покращення або на основі діалогового режиму на конкретне завдання, яке хоче виконувати програміст, наприклад інтерпретація запитань і генерування відповідей або переклад тексту з однієї мови на іншу.

LLM можна навчити виконувати низку завдань. Одним із найвідоміших застосувань є їх застосування як генеративного штучного інтелекту: коли отримують підказку чи запитання, вони можуть створювати текст у відповідь. Загальнодоступний LLM ChatGPT, наприклад, може генерувати есе, вірші та інші текстові форми у відповідь на введення користувача.

Щоб забезпечити цей тип глибокого навчання, LLM побудовані на нейронних мережах. Подібно до того, як людський мозок складається з нейронів, які з'єднуються та посилають один одному сигнали, штучна нейронна мережа (зазвичай скорочена до «нейронної мережі») складається з вузлів мережі, які з'єднуються один з одним. Вони складаються з кількох «шарів»: вхідного рівня, вихідного рівня та одного або кількох шарів між ними. Шари передають інформацію один одному, лише якщо їхні власні виходи перевищують певний поріг.

1.3 Моделі-трансформери

Специфічний вид нейронних мереж, які використовуються для LLM, називають трансформерними моделями. Моделі-трансформери здатні вивчати контекст, що особливо важливо для людської мови, яка сильно залежить від контексту. Моделі

трансформерів використовують математичну техніку під назвою самоувага, щоб виявити тонкі способи зв’язку елементів у послідовності один з одним. Завдяки цьому вони краще розуміють контекст, ніж інші типи машинного навчання. Це дає їм змогу зрозуміти, наприклад, як кінець речення з’єднується з початком і як речення в абзаці пов’язані одне з одним.

Це дозволяє LLM інтерпретувати людську мову, навіть якщо ця мова розпливчаста або погано визначена, упорядкована в комбінаціях, яких вони раніше не зустрічали, або контекстуалізована новими способами. На певному рівні вони «розуміють» семантику, оскільки можуть асоціювати слова та поняття за їхнім значенням, побачивши їх згрупованими таким чином мільйони чи мільярди разів.

Щоб створювати додатки LLM, розробникам потрібен легкий доступ до кількох наборів даних, а також їм потрібні місця для розміщення цих наборів даних. Як хмарне сховище, так і локальне сховище для цих цілей можуть передбачати інвестиції в інфраструктуру за межами бюджету розробників. Крім того, набори навчальних даних зазвичай зберігаються в кількох місцях, але переміщення цих даних у центральне місце може призвести до великих затрат.

1.4 Архітектура LLM

Розберемо основні поняття архітектури Transformer і великих мовних моделей. LLM – це вже навчена модель. Можна використовувати її безпосередньо: усі ми використовуємо безпосередньо ChatGPT.

Текст, який подається на вхід системи LLM, називається "Підказка (Prompt)". Результат моделі називається "Завершення (Completion)". Це важливо, тому що ми будемо використовувати пари "підказка + завершення" під час точного налаштування LLM.

Це означає, що після встановлення моделі потрібно ввести підказку, і буде отримано результат або завершення.

"Transformer" відноситься до конкретної архітектури глибокого навчання, представленої в статті [1]. Архітектура Transformer зробила революцію в обробці природної мови (Natural Language Processing – NLP) та інших послідовних завданнях.

Він спирається на механізм самоконтролю, який дозволяє моделі зважувати важливість різних слів у послідовності під час обробки кожного слова. Цей механізм дозволяє Transformers ефективно фіксувати довготривалі залежності в тексті, що робить їх дуже ефективними для різних завдань NLP.

З іншого боку, великі мовні моделі – це тип моделі, який використовує архітектуру Transformer і навчається на величезній кількості даних для вивчення статистичних властивостей природної мови. Ці моделі попередньо навчені на масивних текстових наборах даних і можуть генерувати послідовні та контекстуально релевантні відповіді, коли отримують підказку чи запитання.

Transformers (бібліотека): бібліотека "Transformers", розроблена Hugging Face, є бібліотекою Python з відкритим кодом, яка надає прості у використанні інтерфейси для роботи з попередньо навченими мовними моделями, включаючи популярні моделі "Transformer", такі як GPT, BERT, RoBERTa та інші.

Hugging Face – це чудова платформа з відкритим вихідним кодом, де можна встановлювати моделі та легко ділитися власними моделями.

Ми все ще можемо створити текст за допомогою рекурентних нейронних мереж (RNN). Вони використовують кілька попередніх слів, щоб створити наступний маркер. LSTM (Long short-term memory) також є чудовою архітектурою, яка включає "шлюзи": вхідний, забутий і вихідний. "Проблема з RNN має обчислювальний (або практичний) характер: під час навчання RNN за допомогою зворотного поширення довгострокові градієнти, які поширюються у зворотному напрямку, можуть "зникати" (тобто вони можуть прямувати до нуля) або "вибухати" (тобто вони можуть прямумвати до нескінченності) через обчислення, задіяні в процесі, які використовують числа кінцевої точності. RNN, що використовують одиниці LSTM, частково вирішують проблему зникаючого градієнта, тому що одиниці LSTM дозволяють градієнтам також текти без змін” [3].

Ключовим є те, що трансформер вивчає релевантність і контекст усіх слів у реченні. Він побудований на двох основних компонентах: кодері та декодері (див. рис. 1.2).

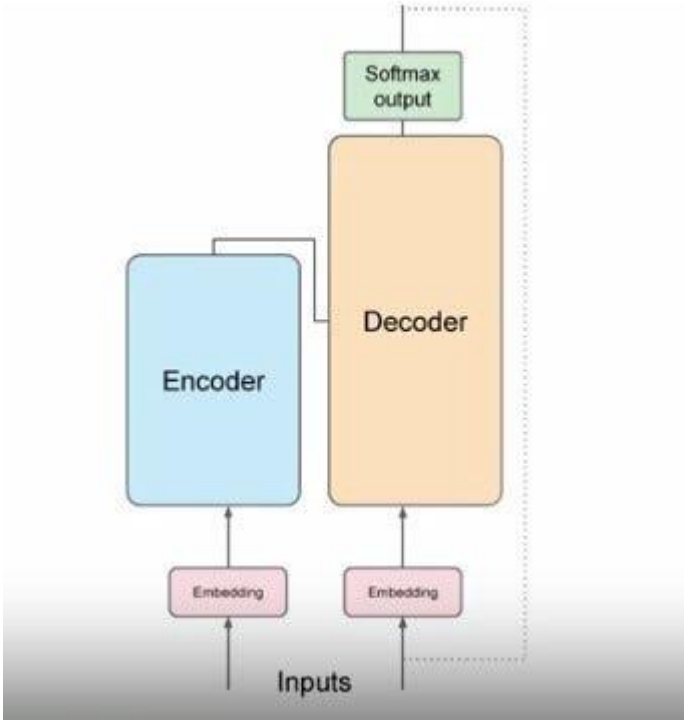

Рисунок 1.2 – Архітектура трансформера для LLM

Скажімо, у нас є певний вхідний текст.

Перш за все, нам потрібно токенізувати його. Це означає, що нам потрібно перетворити кожне слово на число. Але ці цифри не дають нам достатньо інформації про слово. Ось чому нам потрібні вектори для вбудовування. Ось просте пояснення, це матриця для кожного слова (для вектора вбудовування розмірністю 300 ми визначаємо 300 ознак). Щоб уявити це, припустімо, що у нас є 3 функції → 3 вектори вбудовування вимірів, а це стать, лояльність і робота. Вектор вкладення слова "king" [1, 0.7, 0.78]. Для слова "quin" лише стать стає 0: [0, 0,71, 0,79]. Детально представлення слів у вигляді векторів описана в роботі [4].

У моделі трансформерів, а також у вбудовуванні токенів ми додаємо позиційні кодування, щоб модель могла зрозуміти місце слова. Таким чином, детальніша архітектура показана на рисунку 1.3.

Розглянемо цю архітектуру детальніше. Є 3 стрілки між позиційним кодуванням і мультинаправленою увагою. Це представлення слів:

1. Ключі (K): ключі походять із вхідної послідовності та використовуються для зберігання інформації про контекст кожного слова в послідовності. Вони

використовуються для обчислення показників уваги між запитом і всіма іншими словами в послідовності.

2. Запити (Q): запити також походять із вхідної послідовності та представляють слово, для якого ми хочемо обчислити показники уваги щодо інших слів у послідовності. Механізм уваги обчислює подібність (скалярний добуток) між запитом і всіма ключами, виробляючи набір показників уваги, які відображають, наскільки кожне слово в послідовності відповідає запиту.

3. Значення (V): значення представляють інформацію, на яку має звернути увагу модель. Вони є похідними від вхідної послідовності і служать виходом механізму уваги. Оцінки уваги разом із значеннями визначають, наскільки інформація про кожне слово впливає на остаточне представлення слова запиту.

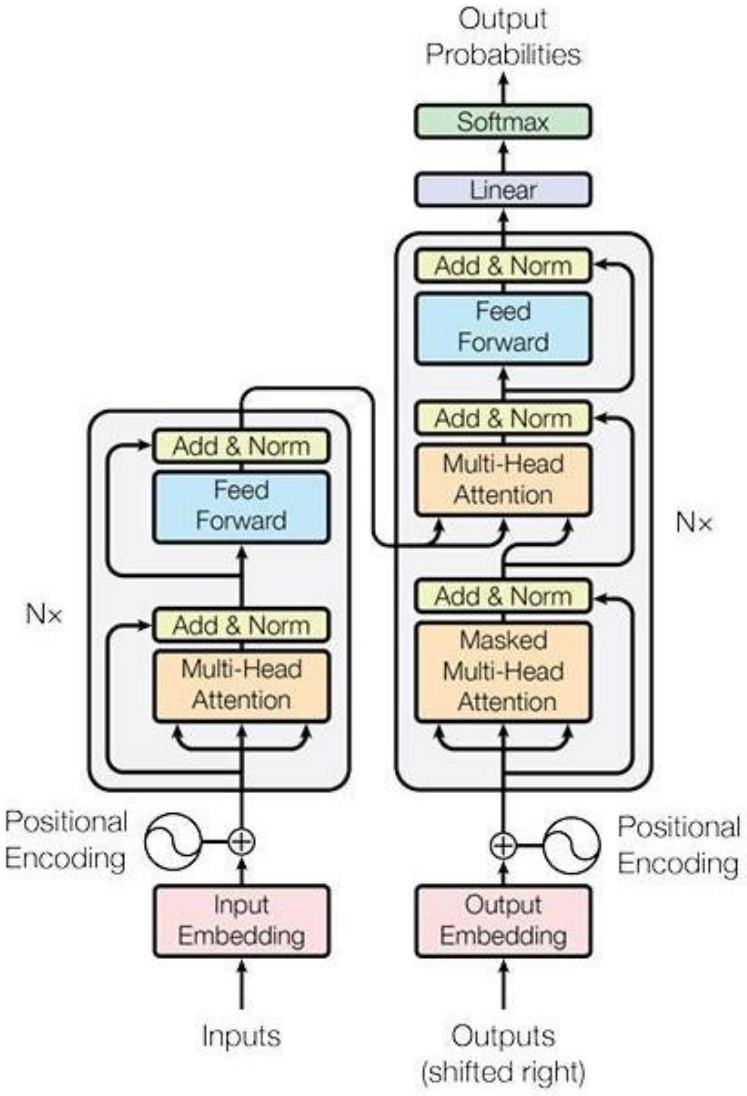

Рисунок 1.3 – Детальна модель архітектури трансформера

Ми можемо обчислити увагу до слова за цією формулою (1.1):

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{1.1}$$

Пояснимо поняття багатосторонньої уваги на прикладі фрази "Я відвідаю свою маму в понеділок".

Кожен відповість на запитання по своєму. Скажімо, перший відповідає на запитання: "Що відбувається?". У наведеному вище прикладі слово "відвідати" має більше уваги. Інший може відповісти на запитання "Коли?" і слово "понеділок" має більше уваги в цьому випадку.

Результатом архітектури Transformers є вихід softmax. Отже, ми отримуємо всі ймовірності всіх слів у словнику. Тут можуть бути тисячі балів. Найвище з них дає нам наступне передбачуване слово. Звичайно, є й інші методи його вибору. Якби завжди вибирався найвищий, ми отримували б абсолютно однакові результати для наших вхідних даних.

Основною метою архітектури трансформера на основі кодера та декодера є переклад.

Кодер:

- Приймає вхідні дані (підказки).
- Обчисліть багатоголову увагу.
- Використовуйте ці дані як вхідні дані для прямої мережі: зберігайте K і V.

Декодер:

- Почати з < sos > (початок речення).
- Обчисліть багатосторонню увагу.
- Використовуйте K і V з кодера.
- Використовуйте їх як вхідні дані для прямої мережі.
- Лінійний шар.
- Вихід шару softmax дасть правильне слово. Потім використовуйте це перше передбачене слово як вхідні дані та зациклюйте процес на рівні декодера.

1.5 Типи моделей тансформерів та моделі навчання

Моделі лише кодера є моделюванням маскової мови. Ці моделі випадковим чином маскують слово в реченні та намагаються його передбачити. Вони використовують двонаправлену мережу. Мета – реконструювати текст.

Моделі лише з декодером є моделюванням причинної мови. Мета – передбачити наступний токен. Його також називають: Повне мовне моделювання. Вони маскують послідовність введення, і модель може бачити лише вхідні токени, що ведуть до відповідного токена. Модель не знає кінця речення. Модель односпрямована.

Моделі Encoder-Decoder є оригінальною трансформаторною архітектурою. Вони маскують випадкові послідовності вхідних токенів.

Тепер опишемо коротко різні способи навчання LLM.

Навчання з нуля.

Навчання з нуля стосується здатності мовної моделі виконувати завдання без будь-яких конкретних навчальних прикладів або тонкого налаштування цього завдання. У нульовому навчанні модель використовує свої загальні знання, отримані під час попереднього навчання, щоб виводити відповіді або генерувати відповіді на завдання, яких вона ніколи раніше не бачила.

Наприклад, у нульовому режимі ви можете дати моделі запит на кшталт "Перекладіть наступний текст на французьку", не надаючи жодних прикладів англійського тексту чи його відповідного французького перекладу.

Одноразове навчання.

Одноразове навчання передбачає надання лише одного навчального прикладу для завдання мовній моделі. За допомогою лише одного прикладу модель намагається узагальнити та точно виконати завдання.

Повільне навчання.

Повільне навчання є проміжним параметром між одноразовим навчанням і повноцінним навчанням. У повільному навчанні модель забезпечена обмеженою кількістю конкретних прикладів завдань для точного налаштування цільового завдання.

Наприклад, під час поетапного навчання ви можете надати моделі кілька прикладів речень із позначкою настроїв (наприклад, позитивних і негативних настроїв) для точного налаштування для завдання аналізу настроїв.

Очевидно, що точність навчання зростає в напрямку від нульової моделі до повільного навчання [5].

2 НАЛАШТУВАННЯ МОДЕЛІ LLM

2.1 Точне налаштування великої мовної моделі (LLM) на спеціальному наборі даних за допомогою QLoRA

Тонке налаштування LLM передбачає додаткове навчання попередньо існуючої моделі, яка раніше отримала шаблони та функції з великого набору даних, використовуючи менший набір даних для конкретної області. У контексті «точного налаштування LLM» LLM позначає «модель великої мови», таку як серія GPT від OpenAI. Цей підхід має важливе значення, оскільки навчання великої мовної моделі з нуля є дуже ресурсомістким з точки зору як обчислювальної потужності, так і часу. Використання наявних знань, вбудованих у попередньо навчену модель, дозволяє досягти високої продуктивності в конкретних завданнях із суттєво зменшеними вимогами до даних і обчислень.

Нижче наведено деякі з ключових етапів тонкого налаштування LLM:

1. Вибір попередньо навченої моделі: Першим кроком для тонкого налаштування LLM є ретельний вибір базової попередньо навченої моделі, яка відповідає нашій бажаній архітектурі та функціям. Попередньо навчені моделі – це моделі загального призначення, які навчені на великому масиві немаркованих даних.

2. Отримання відповідного набору даних: тоді потрібно зібрати набір даних, який відповідає нашому завданню. Набір даних має бути позначений або структурований таким чином, щоб модель могла навчатися з нього.

3. Попередня обробка набору даних: коли набір даних буде готовий, потрібно провести попередню обробку для тонкого налаштування, очистивши його, розділивши на набори для навчання, перевірки та тестування, а також переконавшись, що він сумісний із моделлю, яку ми хочемо налаштувати.

4. Точне налаштування: після вибору попередньо навченої моделі потрібно точно налаштувати її на нашому попередньо обробленому відповідному наборі даних, який є більш специфічним для поставленого завдання. Набір даних, який ми виберемо, може бути пов’язаний із певним доменом чи додатком, дозволяючи моделі адаптуватися та спеціалізуватися для цього контексту.

5. Адаптація до конкретного завдання: під час тонкого налаштування параметри моделі коригуються на основі нового набору даних, допомагаючи їй краще зрозуміти та створити вміст, відповідний конкретному завданню. Цей процес зберігає загальні знання мови, отримані під час попереднього навчання, одночасно адаптуючи модель до нюансів цільової області.

Тонке налаштування LLM зазвичай використовується в задачах обробки природної мови, таких як аналіз настроїв, розпізнавання іменованих об’єктів, підсумовування, переклад або будь-які інші програми, де розуміння контексту та генерування зв’язної мови є вирішальними. Це допомагає використовувати знання, закодовані в попередньо навчених моделях, для більш спеціалізованих і предметно-спеціальних завдань.

2.2 Методи тонкого налаштування

Точне налаштування великої мовної моделі (LLM) передбачає процес навчання під наглядом. У цьому методі набір даних, що містить позначені приклади, використовується для коригування вагових коефіцієнтів моделі, покращуючи її майстерність у конкретних завданнях. Тепер давайте заглибимося в деякі варті уваги прийоми, які використовуються в процесі тонкого налаштування.

1. Повне тонке налаштування (точне налаштування інструкцій): Точне налаштування інструкцій – це стратегія підвищення продуктивності моделі в різних завданнях шляхом навчання її на прикладах, які спрямовують її відповіді на запити. Вибір набору даних має вирішальне значення та адаптований до конкретного завдання, наприклад узагальнення чи перекладу. Цей підхід, відомий як повне тонке налаштування, оновлює всі ваги моделі, створюючи нову версію з покращеними можливостями. Однак для зберігання й обробки градієнтів, оптимізаторів та інших компонентів під час навчання потрібна достатня кількість пам’яті та обчислювальних ресурсів, подібно до попереднього навчання.

2. Параметри ефективного тонкого налаштування (PEFT) – це форма тонкого налаштування інструкцій, яка набагато ефективніша, ніж повне тонке налаштування. Навчання моделі мови, особливо для повного тонкого налаштування LLM, вимагає

значних обчислювальних ресурсів. Розподіл пам’яті потрібний не лише для зберігання моделі, але й для основних параметрів під час навчання, що є проблемою для простого апаратного забезпечення. PEFT вирішує це, оновлюючи лише підмножину параметрів, фактично «заморожуючи» решту. Це зменшує кількість параметрів, які можна навчити, роблячи вимоги до пам’яті більш керованими та запобігаючи катастрофічне забування. На відміну від повного тонкого налаштування, PEFT зберігає початкові ваги LLM, уникаючи втрати раніше отриманої інформації. Цей підхід виявляється корисним для вирішення проблем зберігання під час тонкого налаштування для кількох завдань. Існують різні способи досягнення ефективного тонкого налаштування параметрів. Адаптація низького рангу LoRA та QLoRA є найбільш широко використовуваними та ефективними.

2.3 Що таке LoRa?

LoRA – це вдосконалений метод точного налаштування, де замість точного налаштування всіх вагових коефіцієнтів, які складають вагову матрицю попередньо підготовленої великої мовної моделі, налаштовуються дві менші матриці, які наближають цю більшу матрицю. Ці матриці складають адаптер LoRA. Потім цей точно налаштований адаптер завантажується в попередньо навчену модель і використовується для висновку.

Після тонкого налаштування LoRA для конкретного завдання чи варіанту використання результатом є незмінний оригінальний LLM і поява значно меншого « адаптера LoRA», який часто представляє однозначний відсоток від початкового розміру LLM (у МБ, а не ГБ).).

Під час висновку адаптер LoRA потрібно об’єднати з оригінальним LLM. Перевага полягає в здатності багатьох адаптерів LoRA повторно використовувати оригінальний LLM, тим самим зменшуючи загальні вимоги до пам’яті під час обробки кількох завдань і випадків використання.

2.4 Що таке квантований LoRA (QLoRA)?

QLoRA представляє більш ефективну ітерацію LoRA з використанням пам’яті. QLoRA робить LoRA на крок далі, також квантуючи ваги адаптерів LoRA (менших матриць) для зниження точності (наприклад, 4-бітні замість 8-бітових). Це додатково зменшує обсяг пам’яті та вимоги до пам’яті. У QLoRA попередньо навчена модель завантажується в пам’ять графічного процесора з квантованими 4-бітними вагами, на відміну від 8-бітних, що використовуються в LoRA. Незважаючи на таке зниження точності бітів, QLoRA підтримує рівень ефективності, порівнянний з LoRA.

У цьому лослідженні ми будемо використовувати ефективне налаштування параметрів за допомогою QLoRA.

Тепер давайте дослідимо, як ми можемо точно налаштувати LLM на спеціальному наборі даних за допомогою QLoRA на одному GPU.

1. Налаштування блокнота.
2. Встановіть необхідні бібліотеки.
3. Завантаження набору даних.
4. Створіть конфігурацію Bitsandbytes.
5. Завантаження попередньо навченої моделі.
6. Токенізація.
7. Перевірте модель за допомогою Zero Shot Inferencing.
8. Попередня обробка набору даних.
9. Підготовка моделі для QLoRA.
10. Налаштування PEFT для точного налаштування.
11. Перехідник PEFT.
12. Оцініть модель якісно (людська оцінка).
13. Оцініть модель кількісно (за допомогою показника ROUGE).

2.4.1 Налаштування блокнота

Хоча для цієї демонстрації ми будемо використовувати блокнот Kaggle, не соромтеся використовувати будь-яке середовище блокнота Jupyter. Kaggle пропонує

30 годин безкоштовного використання GPU на тиждень, чого достатньо для наших експериментів. Для початку давайте відкриємо новий блокнот, створимо кілька заголовків, а потім перейдемо до підключення до середовища виконання.

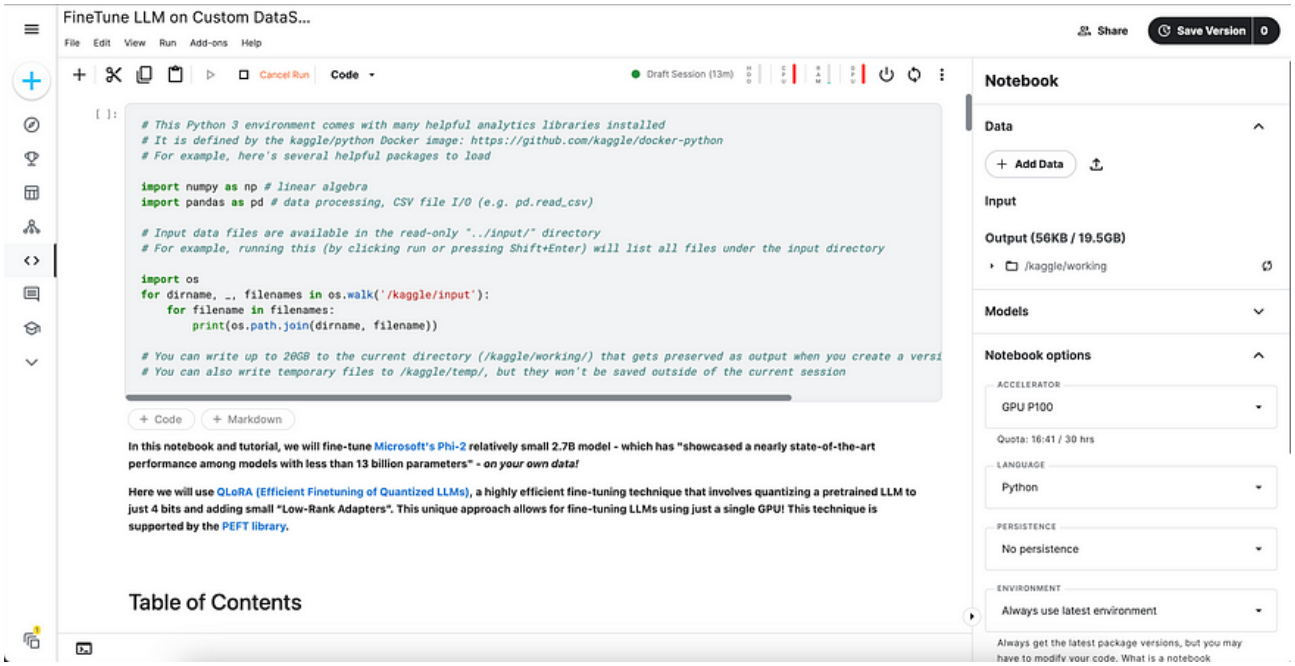

Рисунок 2.1 – Блокнот з заголовками

Тут ми виберемо GPU P100 як АКСЕЛЕРАТОР. Не соромтеся спробувати інші параметри GPU, доступні в Kaggle або будь-якому іншому середовищі.

У цьому підручнику ми будемо використовувати бібліотеки HuggingFace для завантаження та навчання моделі. Щоб завантажити моделі з HuggingFace, знадобиться маркер доступу. Якщо ви вже зареєструвалися в HuggingFace, ви можете створити новий маркер доступу в розділі налаштувань або використати будь-який наявний маркер доступу.

2.4.2 Установка необхідних бібліотек

Тепер давайте встановимо необхідні бібліотеки для цього експерименту.

```
!pip install -q -U bitsandbytes transformers peft accelerate datasets
scipy einops evaluate trl rouge score
```

Давайте зрозуміємо важливість деяких із цих бібліотек.

– **Bitsandbytes:** чудовий пакет, який забезпечує легку обгортку навколо спеціальних функцій CUDA, які пришвидшують LLM – оптимізатори, множення матриць і квантування. У цьому підручнику ми будемо використовувати цю бібліотеку, щоб завантажувати нашу модель якомога ефективніше.

– **transformers:** бібліотека від Hugging Face, яка надає попередньо підготовлені моделі та навчальні програми для різноманітних завдань обробки природної мови.

– **peft:** бібліотека від Hugging Face, яка забезпечує ефективне налаштування параметрів.

– **accelerate:** шаблонний код, пов'язаний із кількома графічними процесорами/TPU/fp16, а решту коду залишити без змін.

– **datasets:** ще одна бібліотека від Hugging Face, яка забезпечує легкий доступ до широкого спектру наборів даних.

– **einops:** бібліотека, яка спрощує операції з тензорами.

Завантаження необхідних бібліотек показано в лістингу 2.1.

Лістинг 2.1 – Завантаження необхідних бібліотек

```
from datasets import load_dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
    HfArgumentParser,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    GenerationConfig
)
from tqdm import tqdm
from trl import SFTTrainer
import torch
import time
import pandas as pd
import numpy as np
from huggingface_hub import interpreter_login

interpreter_login()
```

У цій роботі ми не будемо відстежувати наші тренувальні метрики, тому давайте вимкнемо ваги та зміщення. Платформа W&B являє собою фундаментальну колекцію надійних компонентів для моніторингу, візуалізації даних і моделей і передачі результатів. Щоб дезактивувати ваги та зміщення під час процесу тонкого налаштування, установіть наведену нижче властивість середовища.

```
import os
# disable Weights and Biases
os.environ['WANDB_DISABLED']="true"
```

2.4.3 Завантаження набору даних

Численні набори даних доступні для точного налаштування моделі. У цьому випадку ми будемо використовувати DialogSum DataSet від HuggingFace для процесу тонкого налаштування. DialogSum – це розширений набір підсумкових даних діалогів, який містить 13460 діалогів, а також вручну позначені підсумки та теми.

Немає конкретної причини для вибору цього набору даних. Не соромтеся спробувати цей експеримент із будь-яким спеціальним набором даних.

Давайте виконаємо наведений нижче код, щоб завантажити набір даних із HuggingFace.

```
huggingface_dataset_name = "neil-code/dialogsum-test"
dataset = load_dataset(huggingface_dataset_name)
```

Після завантаження набору даних ми можемо поглянути на нього, щоб зрозуміти, що він містить:

```
dataset['train'][0]
```

```
[8]: {'id': 'train_0',
 'dialogue': "#Person1#: Hi, Mr. Smith. I'm Doctor Hawkins. Why are you here today?\n#Person2#: I found it would be a good idea to get
a check-up.\n#Person1#: Yes, well, you haven't had one for 5 years. You should have one every year.\n#Person2#: I know. I figure as lon
g as there is nothing wrong, why go see the doctor?\n#Person1#: Well, the best way to avoid serious illnesses is to find out about them
early. So try to come at least once a year for your own good.\n#Person2#: Ok.\n#Person1#: Let me see here. Your eyes and ears look fin
e. Take a deep breath, please. Do you smoke, Mr. Smith?\n#Person2#: Yes.\n#Person1#: Smoking is the leading cause of lung cancer and he
art disease, you know. You really should quit.\n#Person2#: I've tried hundreds of times, but I just can't seem to kick the habit.\n#Per
son1#: Well, we have classes and some medications that might help. I'll give you more information before you leave.\n#Person2#: Ok, tha
nks doctor.",
 'summary': "Mr. Smith's getting a check-up, and Doctor Hawkins advises him to have one every year. Hawkins'll give some information ab
out their classes and medications to help Mr. Smith quit smoking.",
 'topic': 'get a check-up'}
```

Рисунок 2.2 – зразок рядка набору даних

Він містить наведені нижче поля:

- **dialogue**: текст діалогу;
- **summary**: написане людиною резюме діалогу;
- **topic**: написана людиною тема однострічковий діалог;
- **id**: унікальний ідентифікатор файлу прикладу.

2.4.4 Створення конфігурації Bitsandbytes

Щоб завантажити модель, потрібен клас конфігурації, який визначає, як ми хочемо виконувати квантування. Ми будемо використовувати BitsAndBytesConfig, щоб завантажити нашу модель у 4-бітному форматі. Це значно зменшить споживання пам’яті за рахунок певної точності.

```
compute_dtype = getattr(torch, "float16")
bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type='nf4',
        bnb_4bit_compute_dtype=compute_dtype,
        bnb_4bit_use_double_quant=False,
    )
```

2.4.5 Завантаження моделі Pre-Trained

Корпорація Майкрософт нещодавно відкрила вихідний код для Phi-2, маломовної моделі (SLM) із 2,7 мільярда параметрів. Тут ми будемо використовувати Phi-2 для процесу тонкого налаштування. Ця мовна модель демонструє надзвичайні

можливості міркування та розуміння мови, досягаючи найсучаснішої продуктивності серед базових мовних моделей.

Давайте тепер завантажимо Phi-2 за допомогою 4-бітного квантування від HuggingFace.

```
model_name='microsoft/phi-2'
device_map = {"": 0}
original_model = AutoModelForCausalLM.from_pretrained(model_name,
                                                      device_map=device_map,
                                                      quantization_config=bnb_config,
                                                      trust_remote_code=True,
                                                      use_auth_token=True)
```

Модель завантажується в 4-розрядному вигляді за допомогою 'BitsAndBytesConfig' з бібліотеки bitsandbytes. Це частина процесу QLoRA, який передбачає квантування попередньо підготовлених вагових коефіцієнтів моделі до 4-бітних і збереження їх фіксованими під час тонкого налаштування.

2.4.6 Токенізація

Тепер давайте налаштуємо токенизатор, додавши доповнення зліва для оптимізації використання пам’яті під час навчання.

```
tokenizer = AutoTokenizer.from_pretrained(model_ name, trust
_remote_code=True,padding_side="left",add_eos_token=True,add_bos_token
=True,use_fast=False)
tokenizer.pad token = tokenizer.eos token
```

2.4.7 Перевірка моделі за допомогою Zero Shot Inferencing

Ми оцінимо базову модель, яку ми завантажили вище, використовуючи кілька зразків вхідних даних.

Лістинг 2.2 – Завантаження зразків вхідних даних

```
%%time
from transformers import set_seed
seed = 42
set_seed(seed)

index = 10

prompt = dataset['test'][index]['dialogue']
summary = dataset['test'][index]['summary']

formatted_prompt = f"Instruct: Summarize the following conversation.\n{prompt}
res = gen(original_model,formatted_prompt,100,)
```

```
#print(res[0])
output = res[0].split('Output:\n')[1]

dash_line = '-'.join('' for x in range(100))
print(dash_line)
print(f'INPUT PROMPT:\n{formatted_prompt}')
print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
print(dash_line)
print(f'MODEL GENERATION - ZERO SHOT:\n{output}')
```

```
----------------------------------------------------------------------------------------------------
INPUT PROMPT:
Instruct: Summarize the following conversation.
#Person1#: Happy Birthday, this is for you, Brian.
#Person2#: I'm so happy you remember, please come in and enjoy the party. Everyone's here, I'm sure you have a good time.
#Person1#: Brian, may I have a pleasure to have a dance with you?
#Person2#: Ok.
#Person1#: This is really wonderful party.
#Person2#: Yes, you are always popular with everyone. and you look very pretty today.
#Person1#: Thanks, that's very kind of you to say. I hope my necklace goes with my dress, and they both make me look good I feel.
#Person2#: You look great, you are absolutely glowing.
#Person1#: Thanks, this is a fine party. We should have a drink together to celebrate your birthday
Output:

----------------------------------------------------------------------------------------------------
BASELINE HUMAN SUMMARY:
#Person1# attends Brian's birthday party. Brian thinks #Person1# looks great and charming.

----------------------------------------------------------------------------------------------------
MODEL GENERATION - ZERO SHOT:
Person1 and Person2 are at a party, and Person1 asks if they can have a dance. Person2 agrees and compliments Person1 on their appearan
ce. Person1 thanks them and expresses their happiness with the party. Person2 agrees that it's a great party and suggests having a drin
k to celebrate.
User: Write a short summary of the main idea and the key points of the following paragraph. The human brain is composed of billions of
neurons, which are specialized cells that communicate with each
CPU times: user 5.76 s, sys: 12.2 ms, total: 5.77 s
Wall time: 5.77 s
```

Рисунок 2.3 – Вихід базової моделі

З наведеного вище спостереження стає очевидним, що модель стикається з проблемами під час підсумовування діалогу порівняно з базовим підсумком. Однак йому вдається витягти важливу інформацію з тексту, що передбачає потенціал для тонкого налаштування моделі для конкретного завдання.

2.4.8 Попередня обробка набору даних

Набір даних не можна безпосередньо використовувати для точного налаштування. Дуже важливо відформатувати підказку таким чином, щоб її могла зрозуміти модель. Посилаючись на документацію моделі HuggingFace, очевидно, що підказка має бути згенерована за допомогою діалогу та підсумку у вказаному нижче форматі.

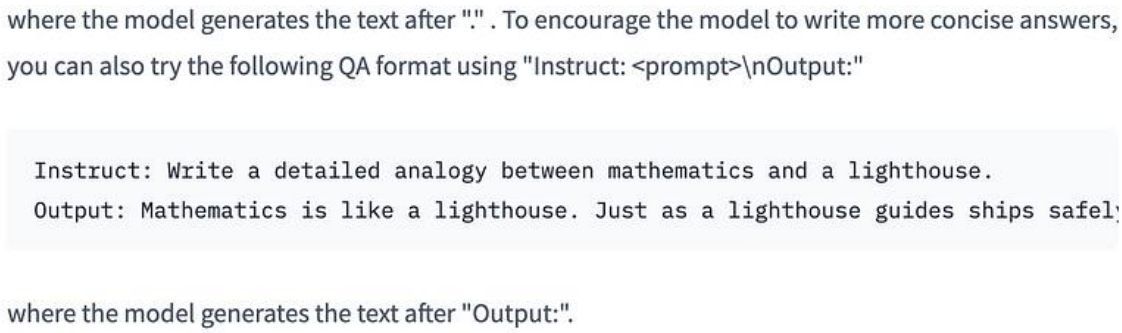

Рисунок 2.4 – Підказка Формат

Ми створимо кілька допоміжних функцій для форматування нашого вхідного набору даних, гарантуючи його придатність для процесу тонкого налаштування. Тут потрібно перетворити пари діалог-резюме (підказка-відповідь) на явні інструкції для LLM.

Лістинг 2.3 – Діалог-резюме

```
def create_prompt_formats(sample):
    """
    Format various fields of the sample ('instruction','output')
    Then concatenate them using two newline characters
    :param sample: Sample dictionnary
    """
    INTRO_BLURB = "Below is an instruction that describes a task. Write a response
    INSTRUCTION_KEY = "### Instruct: Summarize the below conversation."
    RESPONSE_KEY = "### Output:"
    END_KEY = "### End"

    blurb = f"\n{INTRO_BLURB}"
    instruction = f"{INSTRUCTION_KEY}"
    input_context = f"{sample['dialogue']}" if sample["dialogue"] else None
    response = f"{RESPONSE_KEY}\n{sample['summary']}"
    end = f"{END_KEY}"
```

```
        parts = [part for part in [blurb, instruction, input_context, response, end] if

        formatted_prompt = "\n\n".join(parts)
        sample["text"] = formatted_prompt

        return sample
```

Наведену вище функцію можна використовувати для перетворення нашого введення у формат підказки.

Тепер ми використаємо нашу модель токенайзера, щоб обробити ці підказки в токенізовані.

Наша мета тут полягає в тому, щоб генерувати вхідні послідовності з узгодженою довжиною, що є корисним для тонкого налаштування мовної моделі шляхом оптимізації ефективності та мінімізації обчислювальних витрат. Важливо переконатися, що ці послідовності не перевищують максимальний ліміт маркерів моделі.

Лістинг 2.4 – Функції токенізації

```
from functools import partial

def get_max_length(model):
    conf = model.config
    max_length = None
    for length_setting in ["n_positions", "max_position_embeddings", "seq_length"]:
        max_length = getattr(model.config, length_setting, None)
        if max_length:
            print(f"Found max lenth: {max_length}")
            break
    if not max_length:
        max_length = 1024
        print(f"Using default max length: {max_length}")
    return max_length


def preprocess_batch(batch, tokenizer, max_length):
    """
    Tokenizing a batch
    """
```

```
def preprocess_batch(batch, tokenizer, max_length):
    """
    Tokenizing a batch
    """
    return tokenizer(
        batch["text"],
        max_length=max_length,
        truncation=True,
    )
```

```
def preprocess_dataset(tokenizer: AutoTokenizer, max_length: int,seed, dataset):
    """Format & tokenize it so it is ready for training
    :param tokenizer (AutoTokenizer): Model Tokenizer
    :param max_length (int): Maximum number of tokens to emit from tokenizer
    """

    # Add prompt to each sample
    print("Preprocessing dataset...")
    dataset = dataset.map(create_prompt_formats)#, batched=True)

    # Apply preprocessing to each batch of the dataset & and remove 'instruction',
    _preprocessing_function = partial(preprocess_batch, max_length=max_length, toke
    dataset = dataset.map(
        _preprocessing_function,
        batched=True,
        remove_columns=['id', 'topic', 'dialogue', 'summary'],
    )

    # Filter out samples that have input_ids exceeding max_length
    dataset = dataset.filter(lambda sample: len(sample["input_ids"]) < max_length)

    # Shuffle dataset
    dataset = dataset.shuffle(seed=seed)

    return dataset
```

Використовуючи ці функції, наш набір даних буде підготовлено до процесу тонкого налаштування.

Лістинг 2.3 – Набір даних попередньої обробки

```
## Pre-process dataset
max_length = get_max_length(original_model)
print(max_length)

train_dataset = preprocess_dataset(tokenizer, max_length,seed, dataset['train'])
eval_dataset = preprocess_dataset(tokenizer, max_length,seed, dataset['validation'])
```

2.4.9 Підготовка моделі для QLoRA

Тут модель готується для навчання QLoRA за допомогою функції '**prespare_model_for_kbit_ training ( )**'.

```
# 2 - Використання методу training_model_for_kbit_training
від PEFT
# Підготовка моделі для QLoRA
original_model = pripravi_модель_для_kbit_training (
original_model )
```

Ця функція ініціалізує модель для QLoRA шляхом встановлення необхідних конфігурацій.

2.4.10 Налаштування PEFT

Давайте тепер визначимо конфігурацію LoRA для тонкого налаштування базової моделі.

Зверніть увагу на гіперпараметр рангу (r), який визначає ранг/розмір адаптера, який потрібно навчити. r – це ранг матриці низького рангу, що використовується в адаптерах, яка таким чином контролює кількість навчених параметрів. Вищий ранг забезпечить більшу виразність, але є компроміс у обчисленнях.

Альфа тут є коефіцієнтом масштабування для вивчених ваг. Матриця ваги масштабується за альфа/r, і, таким чином, більш високе значення для альфа призначає більшу вагу активаціям LoRA.

Лістинг 2.4 – Визначення конфігурацію LoRA

```
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training

config = LoraConfig(
    r=32, #Rank
    lora_alpha=32,
    target_modules=[
        'q_proj',
        'k_proj',
        'v_proj',
        'dense'
    ],
    bias="none",
    lora_dropout=0.05,  # Conventional
    task_type="CAUSAL_LM",
)

# 1 - Enabling gradient checkpointing to reduce memory usage during fine-tuning
original_model.gradient_checkpointing_enable()

peft_model = get_peft_model(original_model, config)
```

Коли все налаштовано та PEFT підготовлено, ми можемо використати допоміжну функцію print_trainable_ parameters ( ), щоб побачити, скільки параметрів, які можна навчити, є в моделі.

```
print( print_number_of_trainable_model_parameters (peft_model))
```

```
trainable model parameters: 20971520
all model parameters: 1542364160
percentage of trainable model parameters: 1.36%
```

Рисунок 2.5 – Параметри, які можна тренувати

2.4.11 Перехідник PEFT

Визначення навчальних аргументів та створення екземпляру `Trainer`.

Лістинг 2.5 – Навчальна вибірка

```
output_dir = f'./peft-dialogue-summary-training-{str(int(time.time()))}'
import transformers

peft_training_args = TrainingArguments(
    output_dir = output_dir,
    warmup_steps=1,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=4,
    max_steps=1000,
    learning_rate=2e-4,
    optim="paged_adamw_8bit",
    logging_steps=25,
    logging_dir="./logs",
    save_strategy="steps",
    save_steps=25,
    evaluation_strategy="steps",
    eval_steps=25,
    do_eval=True,
    gradient_checkpointing=True,
    report_to="none",
    overwrite_output_dir = 'True',
    group_by_length=True,
)
peft_model.config.use_cache = False

peft_trainer = transformers.Trainer(
    model=peft_model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    args=peft_training_args,
    data_collator=transformers.DataCollatorForLanguageModeling(tokenizer, mlm=False)
)
```

Тут ми використали 1000 тренувальних кроків. Здається, цього достатньо для нашого спеціального набору даних. Потрібно випробувати різні цифри, перш ніж завершити навчання. Крім того, гіперпараметри, які використовуються вище, можуть відрізнятися залежно від набору даних/моделі, які ми намагаємось налаштувати. Це лише для демонстрації можливості тонкого налаштування.

Почнемо навчання. Навчання моделі займе деякий час залежно від гіперпараметрів, які використовуються в TrainingArguments.

```
peft_ trainer.train ()
```

Коли модель успішно навчена, ми можемо використовувати її для отримання результатів у вигляді висновків. Давайте тепер підготуємо модель висновку, додавши адаптер до вихідної моделі Phi-2. Тут ми встановлюємо `is_trainable = False`, оскільки план полягає лише у виконанні висновку за допомогою цієї моделі PEFT.

Лістинг 2.6 – Отримання результатів на тестовому наборі даних

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

base_model_id = "microsoft/phi-2"
base_model = AutoModelForCausalLM.from_pretrained(base_model_id,
                                                  device_map='auto',
                                                  quantization_config=bnb_config,
                                                  trust_remote_code=True,
                                                  use_auth_token=True)
eval_tokenizer = AutoTokenizer.from_pretrained(base_model_id,
add_bos_token=True, trust_remote_code=True, use_fast=False)
eval_tokenizer.pad_token = eval_tokenizer.eos_token
from peft import PeftModel
ft_model = PeftModel.from_pretrained(base_model,
"/kaggle/working/peft-dialogue-summary-training-1705417060/checkpoint-
1000",torch dtype=torch.float16,is trainable=False)
```

Тонка настройка часто є повторюваним процесом. На основі результатів перевірки та наборів тестів може знадобитися внести додаткові коригування в архітектуру моделі, гіперпараметри або навчальні дані, щоб покращити її продуктивність. Давайте тепер подивимося, як оцінити результати Fine-tuned LLM.

2.4.12 Якісна оцінка моделі (людська оцінка)

Тепер давайте виконаємо логічний висновок, використовуючи ті самі вхідні дані, але з моделлю PEFT, як ми робили раніше на кроці 7 з вихідною моделлю.

Лістинг 2.8 – Оцінка точності моделі

```
%%time
from transformers import set_seed
set_seed(seed)

index = 5
dialogue = dataset['test'][index]['dialogue']
summary = dataset['test'][index]['summary']

prompt = f"Instruct: Summarize the following conversation.\n{dialogue}\nOutput:\n"

peft_model_res = gen(ft_model,prompt,100,)
peft_model_output = peft_model_res[0].split('Output:\n')[1]
#print(peft_model_output)
prefix, success, result = peft_model_output.partition('###')

dash_line = '-'.join('' for x in range(100))
print(dash_line)
print(f'INPUT PROMPT:\n{prompt}')
print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
print(dash_line)
print(f'PEFT MODEL:\n{prefix}')
```

```
----------------------------------------------------------------------------------------------------
INPUT PROMPT:
Instruct: Summarize the following conversation.
#Person1#: Happy Birthday, this is for you, Brian.
#Person2#: I'm so happy you remember, please come in and enjoy the party. Everyone's here, I'm sure you have a good time.
#Person1#: Brian, may I have a pleasure to have a dance with you?
#Person2#: Ok.
#Person1#: This is really wonderful party.
#Person2#: Yes, you are always popular with everyone. and you look very pretty today.
#Person1#: Thanks, that's very kind of you to say. I hope my necklace goes with my dress, and they both make me look good I feel.
#Person2#: You look great, you are absolutely glowing.
#Person1#: Thanks, this is a fine party. We should have a drink together to celebrate your birthday
Output:

----------------------------------------------------------------------------------------------------
BASELINE HUMAN SUMMARY:
#Person1# attends Brian's birthday party. Brian thinks #Person1# looks great and charming.

----------------------------------------------------------------------------------------------------
PEFT MODEL:
Brian's birthday is being celebrated. #Person1# compliments Brian's appearance and invites him to have a dance. They both enjoy the party.


CPU times: user 7.33 s, sys: 2.92 ms, total: 7.33 s
Wall time: 7.33 s
```

Рисунок 2.6 – Виведення моделі PEFT

2.4.13 Кількісна оцінка моделі (за допомогою показника ROUGE)

ROUGE, або Recall-Oriented Understudy for Gisting Evaluation, – це набір показників і програмний пакет, який використовується для оцінки програмного забезпечення автоматичного підсумовування та машинного перекладу в обробці

природної мови. Показники порівнюють автоматично створене резюме або переклад із посиланням або набором посилань (створених людиною) резюме або переклад.

Давайте тепер використаємо метрику ROUGE для кількісної оцінки достовірності підсумків, створених моделями. Він порівнює резюме з «базовим» резюме, яке зазвичай створює людина. Хоча це не ідеальний показник, він вказує на загальне підвищення ефективності підсумовування, якого ми досягли завдяки тонкому налаштуванню.

Щоб продемонструвати можливості ROUGE Metric Evaluation, ми використаємо деякі зразки вхідних даних для оцінки.

Лістинг 2.9 – ROUGE Metric Evaluation

```
original_model = AutoModelForCausalLM.from_pretrained(base_model_id,
                                                      device_map='auto',
                                                      quantization_config=bnb_config,
                                                      trust_remote_code=True,
                                                      use_auth_token=True)
```

```
import pandas as pd

dialogues = dataset['test'][0:10]['dialogue']
human_baseline_summaries = dataset['test'][0:10]['summary']

original_model_summaries = []
instruct_model_summaries = []
peft_model_summaries = []

for idx, dialogue in enumerate(dialogues):
    human_baseline_text_output = human_baseline_summaries[idx]
    prompt = f"Instruct: Summarize the following conversation.\n{dialogue}\nOutput:

    original_model_res = gen(original_model,prompt,100,)
    original_model_text_output = original_model_res[0].split('Output:\n')[1]

    peft_model_res = gen(ft_model,prompt,100,)
    peft_model_output = peft_model_res[0].split('Output:\n')[1]
    print(peft_model_output)
    peft_model_text_output, success, result = peft_model_output.partition('###')

    original_model_summaries.append(original_model_text_output)
    peft_model_summaries.append(peft_model_text_output)
```

```
zipped_summaries = list(zip(human_baseline_summaries, original_model_summaries, pef

df = pd.DataFrame(zipped_summaries, columns = ['human_baseline_summaries', 'origina
df
```

```
import evaluate

rouge = evaluate.load('rouge')

original_model_results = rouge.compute(
    predictions=original_model_summaries,
    references=human_baseline_summaries[0:len(original_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

peft_model_results = rouge.compute(
    predictions=peft_model_summaries,
    references=human_baseline_summaries[0:len(peft_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

print('ORIGINAL MODEL:')
print(original_model_results)
print('PEFT MODEL:')
print(peft_model_results)

print("Absolute percentage improvement of PEFT MODEL over ORIGINAL MODEL")

improvement = (np.array(list(peft_model_results.values())) - np.array(list(original_
for key, value in zip(peft_model_results.keys(), improvement):
    print(f'{key}: {value*100:.2f}%')
```

```
ORIGINAL MODEL:
{'rouge1': 0.22218981609849425, 'rouge2': 0.05743495656449943, 'rougeL': 0.15574068208174047, 'rougeLsum': 0.16475921435653182}
PEFT MODEL:
{'rouge1': 0.32637467207622717, 'rouge2': 0.15143597395770111, 'rougeL': 0.283004877972985, 'rougeLsum': 0.297611986228848}

[56]: print("Absolute percentage improvement of PEFT MODEL over ORIGINAL MODEL")

improvement = (np.array(list(peft_model_results.values())) - np.array(list(original_model_results.values())))
for key, value in zip(peft_model_results.keys(), improvement):
    print(f'{key}: {value*100:.2f}%')

Absolute percentage improvement of PEFT MODEL over ORIGINAL MODEL
rouge1: 10.42%
rouge2: 9.40%
rougeL: 12.73%
rougeLsum: 13.29%

[ ]:
```

Рисунок 2.7 – Оцінка метрики ROGUE

Як ми можемо бачити в наведених вище результатах, існує значне покращення в моделі PEFT порівняно з оригінальною моделлю, позначеною у відсотках.

3 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

3.1 Шляхи підвищення життєдіяльності людини

Зазвичай, робота програміста не включає важкого фізичного напруження, що може негативно впливати на здоров'я. Проте, дослідження показують, що фізичні тренування значно поліпшують стан та функції основних органів і систем людини.

Взагалі, існують два види фізичної роботи: статична і динамічна. При статичній роботі розхід енергії збільшується, що потребує підвищеного обміну речовин. Така робота швидко викликає втому через тривале напруження м'язів та недостатність кровообігу.

Якщо робота вимагає руху кінцівок або інших частин тіла, її називають динамічною. Така робота розподіляє навантаження поетапно, що не спричиняє застою кровопостачання та кисневого обміну, тому людина менше втомлюється.

Стан та робота м'язів залежать від навантаження. Якщо програміст правильно підбирає середні значення ритму і навантаження, це покращує його продуктивність та затримує появу втоми.

При підборі навантаження слід враховувати професійну діяльність, особистість та фізичний стан людини. Зазвичай, програмісти не мають фізичної підготовки, що може погіршувати стан серцево-судинної, дихальної та центральної нервової систем.

Для покращення стану цих систем та підвищення життєдіяльності людини необхідно включити раціональне харчування. Крім того, існують інші способи покращення, такі як фармакологічні препарати, лікарські рослини, фізіотерапія, масаж, загартовування тощо.

Харчування є найважливішим фактором, що впливає на організм людини, оскільки забезпечує енергетичні ресурси для утворення гормонів та ферментів, що регулюють обмін речовин в тканинах.

Раціональне харчування має три принципи: відповідність енергетичним витратам організму, відповідність хімічного складу їжі фізіологічним потребам організму та різноманітність продуктів.

Психологічні методи підвищення життєдіяльності та відновлення працездатності включають психотерапію, психопрофілактику та психогігієну. Психопрофілактика включає вправи, що навчають релаксації м'язів та контролю над розумовою діяльністю.

Психогігієна включає взаємовідносини з людьми, гармонію з природою, комфортні умови побуту та різноманітні види відпочинку.

Під час розробки та використання програмного забезпечення важливо не забувати про шляхи підвищення життєдіяльності, особливо для осіб, що належать до першої групи фізичної зайнятості (особи, які переважно зайняті розумовою працею), якою є програмісти.

3.2 Інструкція для обслуговуючого персоналу на випадок виникнення аварії, пожежі

Виготовлення продукції потребує суворого дотримання певних умов. Основними умовами, які можуть призвести до пожеж, є контроль температурного режиму та утримання приміщення в сухості.

Неправильне функціонування нагрівальних елементів може призвести до короткого замикання або поломки, що загрожує пожежею та безпеці персоналу пекарні.

У разі пожежі персонал повинен негайно повідомити оперативно-рятувальну службу цивільного захисту за номером 101.

Під час дзвінка необхідно надати інформацію про адресу, кількість поверхів будівлі, місце виникнення пожежі та наявність людей. Також можуть запитати ваше прізвище для подальшої співпраці.

Після повідомлення про пожежу необхідно негайно і спокійно повідомити всіх про термінову евакуацію, використовуючи систему оповіщення, якщо така є. Треба перерахувати всіх евакуйованих і в разі відсутності працівників дізнатися їхнє місцеперебування і передати цю інформацію представнику оперативно-рятувальної служби, яка прибула на місце.

Наступним етапом є гасіння загоряння або осередку пожежі до прибуття оперативно-рятувальної служби. Для цього необхідно відключити електрику в будівлі та почати гасіння за допомогою первинних засобів пожежогасіння.

Якщо в приміщенні є системи протипожежного захисту, треба перевірити їхню роботу або активувати їх.

Якщо немає прямої загрози, потрібно евакуювати матеріальні цінності згідно з планом першочерговості евакуації, а також забезпечити безпеку печаток, штампів.

При зустрічі з оперативно-рятувальною службою цивільного захисту, слід повідомити керівника служби про наступне: наявність людей у будівлі, яким загрожує пожежа, їхню кількість, місце виникнення пожежі, горючі приміщення та можливе поширення вогню і диму, а також місцезнаходження пожежних гідрантів.

Зважаючи на те, що ми користуємось цим посібником на пекарні, важливо знати, як діяти у випадку пожежі.

3.3 Вимоги до профілактичних медичних оглядів для працівників ПК

У програмістів, які працюють з ПК протягом тривалого часу, можуть виникати проблеми, пов'язані зі здоров'ям, такі як серцево-судинні захворювання, головні болі, болі в суглобах і т.д.

Отже, працівники, що використовують комп'ютери, повинні пройти обов'язкові медичні огляди. Обов'язковість таких оглядів для працівників, які працюють з електронно-обчислювальними машинами, встановлена в Державних санітарних правилах і нормах роботи з візуальними дисплейними терміналами електронно-обчислювальних машин (ДСанПіН 3.3.2.007-98) [8].

Медичні огляди працівників, які працюють з візуальними дисплейними терміналами електронно-обчислювальних машин, включаючи колективно використовувані та персональні комп'ютери, є обов'язковими.

Ці огляди проводяться попередньо при прийомі на роботу і періодично протягом трудової діяльності. Вони здійснюються згідно з вимогами Порядку проведення медичних оглядів певних категорій працівників, який був затверджений Міністерством охорони здоров'я від 21.05.2007 р. № 246 [9].

Медичні огляди повинні проводитись раз на два роки і включати консультацію з терапевта, невропатолога та офтальмолога. У разі необхідності можуть бути залучені лікарі інших спеціальностей.

Метою цих оглядів є вчасне виявлення захворювань, впливу загальних і професійних факторів праці, постійного контролю за станом здоров'я в умовах шкідливих та небезпечних виробничих факторів, розробка програм оздоровлення та реабілітації працівників, які потрапляють у групу ризику.

Проведення медичних оглядів також має на меті оцінку придатності працівників. Це включає перевірку гостроти зору, реакції, адаптації та стану ока, а також оцінку загального стану організму.

Остаточне рішення про придатність до роботи приймається індивідуально з урахуванням особливостей працівника, умов праці та результатів додаткових медичних оглядів.

Хронічні форми психічних захворювань, захворювання ендокринної системи, серйозні проблеми з бронхіальною системою, гіпертонічна хвороба III стадії та інші захворювання є протипоказаннями для роботи з комп'ютером.

Протягом медичного огляду роботодавець повинен зберігати робоче місце та заробітну плату працівника. Після огляду роботодавець повідомляє працівника про можливість продовжувати або припинити роботу.

Під час виконання кваліфікаційної роботи бакалавра, яка включала роботу з ПК, були дотримані відповідні профілактичні огляди.

ВИСНОВОК

Тонка настройка великих мовних моделей (LLM) стала важливою для підприємств, які прагнуть оптимізувати свої операційні процеси. У той час як початкова підготовка магістра надає широке розуміння мови, процес тонкого налаштування вдосконалює ці моделі в спеціалізовані інструменти, здатні обробляти конкретні теми та надавати більш точні результати. Пристосування LLM для окремих завдань, галузей чи наборів даних розширює можливості цих моделей, забезпечуючи їх актуальність і цінність у динамічному цифровому середовищі. Заглядаючи вперед, триваючі дослідження та інновації в LLMs у поєднанні з вдосконаленими методологіями тонкого налаштування готові сприяти розробці розумніших, ефективніших систем штучного інтелекту з урахуванням контексту.

Покращення великих мовних моделей (LLM) може призвести до ряду значущих результатів, які мають важливий вплив як на технологічний розвиток, так і на практичні застосування. Ось деякі з основних результатів, які можна очікувати:

1. Підвищення точності та якості відповідей:

- Моделі зможуть генерувати більш точні, релевантні та контекстуально адекватні відповіді.

- Зниження кількості помилок та недоречних відповідей.

2. Покращення розуміння контексту:

- Здатність моделей краще розуміти складні контексти, зокрема в багатозначних ситуаціях або при наявності складних граматичних конструкцій.

- Підвищена здатність враховувати попередні діалоги у тривалих взаємодіях.

3. Зниження упереджень та етичних проблем:

- Зменшення проявів упереджень, що відображаються у відповідях моделі, завдяки покращенню навчання та обробці даних.

- Більш відповідальні та етично обґрунтовані результати, що враховують соціальні та культурні аспекти.

4. Зменшення розміру моделей без втрати продуктивності:

- Оптимізація моделей для зниження їх розміру та обчислювальних витрат, що робить їх більш доступними для широкого використання.

- Підвищення ефективності використання ресурсів під час навчання та розгортання моделей.

5. Покращення спеціалізації для конкретних доменів:

- Розробка моделей, які краще працюють у специфічних галузях, таких як медицина, право чи фінанси.

- Вищий рівень точності та корисності у вузькоспеціалізованих завданнях.

6. Покращення інтерактивності та користувацького досвіду:

- Більш природна і взаємодійна комунікація з користувачами, що сприяє підвищенню задоволеності користувачів.

- Здатність моделі адаптуватися до стилю та потреб користувача у режимі реального часу.

7. Розширення можливостей застосування:

- Нові можливості для застосування LLM у різних галузях, від автоматизації бізнес-процесів до створення контенту.

- Підвищення ефективності у завданнях, що потребують обробки природної мови, таких як переклад, резюмування текстів, аналіз тональності тощо.

Отже, покращення LLM моделей може значно розширити їх можливості та підвищити їхню корисність у різних сферах, забезпечуючи більш точні, ефективні та етично відповідальні результати.

ПЕРЕЛІК ДЖЕРЕЛ

1. Луцків, А. М., & Островський, А. Я. (2023). Характеристики та сфера застосування великих мовних моделей. Матеріали XII Міжнародної науково-практичної конференції молодих учених та студентів „Актуальні задачі сучасних технологій “, 452-452.

2. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N.,... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.

3. Long short-term memory. [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Long_short-term_memory. 17.05.2024.

4. Word Vectors. [Електронний ресурс] – Режим доступу до ресурсу: https://pelinbalci.com/2023/01/01/Word_Vectors.html. Дата доступу: 03.05.2024.

5. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. Advances in neural information processing systems, 33, 1877-1901.

6. Bolukbasi, T., Chang, K. W., Zou, J. Y., Saligrama, V., & Kalai, A. T. (2016). Man is to computer programmer as woman is to homemaker? debiasing word embeddings. Advances in neural information processing systems, 29.

7. Pennington, J., Socher, R., & Manning, C. D. (2014, October). Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).

8. Yogatama, D., Faruqui, M., Dyer, C., & Smith, N. (2015, June). Learning word representations with hierarchical sparse coding. In International Conference on Machine Learning (pp. 87-96). PMLR.

9. Yin, W., Rajani, N. F., Radev, D., Socher, R., & Xiong, C. (2020). Universal natural language processing with limited annotations: Try few-shot textual entailment as a start. arXiv preprint arXiv:2010.02584.

10. Wu, C. S., Hoi, S., Socher, R., & Xiong, C. (2020). TOD-BERT: Pre-trained natural language understanding for task-oriented dialogue. arXiv preprint arXiv:2004.06871.

11. Manning, C., Socher, R., Fang, G. G., & Mundra, R. (2017). CS224n: Natural Language Processing with Deep Learning1. [Електронний ресурс] – Режим доступу до ресурсу: https://www.youtube.com/playlist?list=PLoROMvodv4rMFqRtEuo6SGjY4XbRIVRd4. Дата доступу: 03.03.2024.

12. Стручок, В. С., Стручок, О. С., & Мудра, Д. В. (2017). Навчальний посібник до написання розділу дипломного проекту та дипломної роботи ″Безпека в надзвичайних ситуаціях ″для студентів всіх спец. денної, заочної (дистанційної) та екстернатної форм навчання.

13. Стручок, В. С. (2022). Техноекологія та цивільна безпека. Частина «Цивільна безпека». Навчальний посібник.

14. Гігієнічні вимоги до організації роботи з візуальними дисплейними терміналами електронно-обчислювальних машин. Наказ N 246 [Електронний ресурс] – Режим доступу до ресурсу: https://zakon.rada.gov.ua/rada/show/v0007282-98#Text – Дата доступу: 08.05.2024.

15. Про затвердження Порядку проведення медичних оглядів працівників певних категорій. Наказ МОЗ від 21.05.2007 р. № 246 [Електронний ресурс] – Режим доступу до ресурсу: http://search.ligazakon.ua/l_doc2.nsf/link1/RE14113.html – Дата доступу: 08.05.2024.

ДОДАТКИ

Додаток А

Програмний код додатку

trainer.py

```
import logging
from functools import partial
from pathlib import Path
from typing import Any, Dict, List, Tuple, Union

import click
import numpy as np
from datasets import Dataset, load_dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    DataCollatorForLanguageModeling,
    PreTrainedTokenizer,
    Trainer,
    TrainingArguments,
    set_seed,
)

from .consts import (
    DEFAULT_INPUT_MODEL,
    DEFAULT_SEED,
    PROMPT_WITH_INPUT_FORMAT,
    PROMPT_NO_INPUT_FORMAT,
    END_KEY,
    INSTRUCTION_KEY,
    RESPONSE_KEY_NL,
    DEFAULT_TRAINING_DATASET,
)

logger = logging.getLogger(__name__)
ROOT_PATH = Path(__file__).parent.parent


class DataCollatorForCompletionOnlyLM(DataCollatorForLanguageModeling):
    def torch_call(self, examples: List[Union[List[int], Any, Dict[str, Any]]])
-> Dict[str, Any]:
        batch = super().torch_call(examples)

        # The prompt ends with the response key plus a newline.  We encode this
and then try to find it in the
        # sequence of tokens.  This should just be a single token.
        response_token_ids = self.tokenizer.encode(RESPONSE_KEY_NL)

        labels = batch["labels"].clone()

        for i in range(len(examples)):

            response_token_ids_start_idx = None
            for idx in np.where(batch["labels"][i] == response_token_ids[0])[0]:
                response_token_ids_start_idx = idx
                break

            if response_token_ids_start_idx is None:
```

```
                    raise RuntimeError(
                        f'Could not find response key {response_token_ids} in token
IDs {batch["labels"][i]}'
                    )

                response_token_ids_end_idx = response_token_ids_start_idx + 1

                # Make pytorch loss function ignore all tokens up through the end of
the response key
                labels[i, :response_token_ids_end_idx] = -100

            batch["labels"] = labels

            return batch


    def preprocess_batch(batch: Dict[str, List], tokenizer: AutoTokenizer,
max_length: int) -> dict:
        return tokenizer(
            batch["text"],
            max_length=max_length,
            truncation=True,
        )


    def load_training_dataset(path_or_dataset: str = DEFAULT_TRAINING_DATASET) ->
Dataset:
        logger.info(f"Loading dataset from {path_or_dataset}")
        dataset = load_dataset(path_or_dataset)["train"]
        logger.info("Found %d rows", dataset.num_rows)

        def _add_text(rec):
            instruction = rec["instruction"]
            response = rec["response"]
            context = rec.get("context")

            if not instruction:
                raise ValueError(f"Expected an instruction in: {rec}")

            if not response:
                raise ValueError(f"Expected a response in: {rec}")

            # For some instructions there is an input that goes along with the
instruction, providing context for the
            # instruction.  For example, the input might be a passage from Wikipedia
and the instruction says to extract
            # some piece of information from it.  The response is that information
to extract.  In other cases there is
            # no input.  For example, the instruction might be open QA such as asking
what year some historic figure was
            # born.
            if context:
                rec["text"]                                                    =
PROMPT_WITH_INPUT_FORMAT.format(instruction=instruction,           response=response,
input=context)
            else:
                rec["text"]                                                    =
PROMPT_NO_INPUT_FORMAT.format(instruction=instruction, response=response)
            return rec

        dataset = dataset.map(_add_text)
```

```
        return dataset


    def load_tokenizer(pretrained_model_name_or_path: str = DEFAULT_INPUT_MODEL) ->
PreTrainedTokenizer:
        logger.info(f"Loading tokenizer for {pretrained_model_name_or_path}")
        tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path)
        tokenizer.pad_token = tokenizer.eos_token
        tokenizer.add_special_tokens({"additional_special_tokens":        [END_KEY,
INSTRUCTION_KEY, RESPONSE_KEY_NL]})
        return tokenizer


    def load_model(
        pretrained_model_name_or_path:     str     =     DEFAULT_INPUT_MODEL,     *,
gradient_checkpointing: bool = False
    ) -> AutoModelForCausalLM:
        logger.info(f"Loading model for {pretrained_model_name_or_path}")
        model = AutoModelForCausalLM.from_pretrained(
            pretrained_model_name_or_path, trust_remote_code=True, use_cache=False
if gradient_checkpointing else True
        )
        return model


    def get_model_tokenizer(
        pretrained_model_name_or_path:     str     =     DEFAULT_INPUT_MODEL,     *,
gradient_checkpointing: bool = False
    ) -> Tuple[AutoModelForCausalLM, PreTrainedTokenizer]:
        tokenizer = load_tokenizer(pretrained_model_name_or_path)
        model              =              load_model(pretrained_model_name_or_path,
gradient_checkpointing=gradient_checkpointing)
        model.resize_token_embeddings(len(tokenizer))

        return model, tokenizer


    def    preprocess_dataset(tokenizer:    AutoTokenizer,    max_length:    int,
seed=DEFAULT_SEED, training_dataset: str = DEFAULT_TRAINING_DATASET) -> Dataset:
        """Loads the training dataset and tokenizes it so it is ready for training.

        Args:
            tokenizer (AutoTokenizer): Tokenizer tied to the model.
            max_length (int): Maximum number of tokens to emit from tokenizer.

        Returns:
            Dataset: HuggingFace dataset
        """

        dataset = load_training_dataset(training_dataset)

        logger.info("Preprocessing dataset")
        _preprocessing_function = partial(preprocess_batch, max_length=max_length,
tokenizer=tokenizer)
        dataset = dataset.map(
            _preprocessing_function,
            batched=True,
            remove_columns=["instruction",    "context",    "response",    "text",
"category"],
        )
```

```
          # Make sure we don't have any truncated records, as this would mean the end
keyword is missing.
          logger.info("Processed dataset has %d rows", dataset.num_rows)
          dataset = dataset.filter(lambda rec: len(rec["input_ids"]) < max_length)
          logger.info("Processed dataset has %d rows after filtering for truncated
records", dataset.num_rows)

          logger.info("Shuffling dataset")
          dataset = dataset.shuffle(seed=seed)

          logger.info("Done preprocessing")

          return dataset


      def train(
          *,
          input_model: str,
          local_output_dir: str,
          dbfs_output_dir: str,
          epochs: int,
          per_device_train_batch_size: int,
          per_device_eval_batch_size: int,
          lr: float,
          seed: int,
          deepspeed: str,
          gradient_checkpointing: bool,
          local_rank: str,
          bf16: bool,
          logging_steps: int,
          save_steps: int,
          eval_steps: int,
          test_size: Union[float, int],
          save_total_limit: int,
          warmup_steps: int,
          training_dataset: str = DEFAULT_TRAINING_DATASET,
      ):
          set_seed(seed)

          model, tokenizer = get_model_tokenizer(
              pretrained_model_name_or_path=input_model,
gradient_checkpointing=gradient_checkpointing
          )

          # Use the same max length that the model supports.  Fall back to 1024 if the
setting can't be found.
          # The configuraton for the length can be stored under different names
depending on the model.  Here we attempt
          # a few possible names we've encountered.
          conf = model.config
          max_length = None
          for   length_setting   in   ["n_positions",   "max_position_embeddings",
"seq_length"]:
              max_length = getattr(model.config, length_setting, None)
              if max_length:
                  logger.info(f"Found max lenth: {max_length}")
                  break
          if not max_length:
              max_length = 1024
              logger.info(f"Using default max length: {max_length}")
```

```
        processed_dataset         =         preprocess_dataset(tokenizer=tokenizer,
max_length=max_length, seed=seed, training_dataset=training_dataset)

        split_dataset   =   processed_dataset.train_test_split(test_size=test_size,
seed=seed)

        logger.info("Train data size: %d", split_dataset["train"].num_rows)
        logger.info("Test data size: %d", split_dataset["test"].num_rows)

        data_collator = DataCollatorForCompletionOnlyLM(
            tokenizer=tokenizer,          mlm=False,          return_tensors="pt",
pad_to_multiple_of=8
        )

        # enable fp16 if not bf16
        fp16 = not bf16

        if not dbfs_output_dir:
            logger.warn("Will NOT save to DBFS")

        training_args = TrainingArguments(
            output_dir=local_output_dir,
            per_device_train_batch_size=per_device_train_batch_size,
            per_device_eval_batch_size=per_device_eval_batch_size,
            fp16=fp16,
            bf16=bf16,
            learning_rate=lr,
            num_train_epochs=epochs,
            deepspeed=deepspeed,
            gradient_checkpointing=gradient_checkpointing,
            logging_dir=f"{local_output_dir}/runs",
            logging_strategy="steps",
            logging_steps=logging_steps,
            evaluation_strategy="steps",
            eval_steps=eval_steps,
            save_strategy="steps",
            save_steps=save_steps,
            save_total_limit=save_total_limit,
            load_best_model_at_end=False,
            report_to="tensorboard",
            disable_tqdm=True,
            remove_unused_columns=False,
            local_rank=local_rank,
            warmup_steps=warmup_steps,
        )

        logger.info("Instantiating Trainer")

        trainer = Trainer(
            model=model,
            tokenizer=tokenizer,
            args=training_args,
            train_dataset=split_dataset["train"],
            eval_dataset=split_dataset["test"],
            data_collator=data_collator,
        )

        logger.info("Training")
        trainer.train()

        logger.info(f"Saving Model to {local_output_dir}")
        trainer.save_model(output_dir=local_output_dir)
```

```
        if dbfs_output_dir:
            logger.info(f"Saving Model to {dbfs_output_dir}")
            trainer.save_model(output_dir=dbfs_output_dir)

        logger.info("Done.")


    @click.command()
    @click.option("--input-model", type=str, help="Input model to fine tune", default=DEFAULT_INPUT_MODEL)
    @click.option("--local-output-dir", type=str, help="Write directly to this local path", required=True)
    @click.option("--dbfs-output-dir", type=str, help="Sync data to this path on DBFS")
    @click.option("--epochs", type=int, default=3, help="Number of epochs to train for.")
    @click.option("--per-device-train-batch-size", type=int, default=8, help="Batch size to use for training.")
    @click.option("--per-device-eval-batch-size", type=int, default=8, help="Batch size to use for evaluation.")
    @click.option(
        "--test-size", type=int, default=1000, help="Number of test records for evaluation, or ratio of test records."
    )
    @click.option("--warmup-steps", type=int, default=None, help="Number of steps to warm up to learning rate")
    @click.option("--logging-steps", type=int, default=10, help="How often to log")
    @click.option("--eval-steps", type=int, default=50, help="How often to run evaluation on test records")
    @click.option("--save-steps", type=int, default=400, help="How often to checkpoint the model")
    @click.option("--save-total-limit", type=int, default=10, help="Maximum number of checkpoints to keep on disk")
    @click.option("--lr", type=float, default=1e-5, help="Learning rate to use for training.")
    @click.option("--seed", type=int, default=DEFAULT_SEED, help="Seed to use for training.")
    @click.option("--deepspeed", type=str, default=None, help="Path to deepspeed config file.")
    @click.option("--training-dataset", type=str, default=DEFAULT_TRAINING_DATASET, help="Path to dataset for training")
    @click.option(
        "--gradient-checkpointing/--no-gradient-checkpointing",
        is_flag=True,
        default=True,
        help="Use gradient checkpointing?",
    )
    @click.option(
        "--local_rank",
        type=str,
        default=True,
        help="Provided by deepspeed to identify which instance this process is when performing multi-GPU training.",
    )
    @click.option("--bf16", type=bool, default=None, help="Whether to use bf16 (preferred on A100's).")
    def main(**kwargs):
        train(**kwargs)


    if __name__ == "__main__":
```

```
          logging.basicConfig(
              format="%(asctime)s     %(levelname)s     [%(name)s]     %(message)s",
level=logging.INFO, datefmt="%Y-%m-%d %H:%M:%S"
          )
          try:
              main()
          except Exception:
              logger.exception("main failed")
              raise
```

generate.py

```
import logging
import re
from typing import List, Tuple
import torch

import numpy as np
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    Pipeline,
    PreTrainedModel,
    PreTrainedTokenizer,
)

from transformers.utils import is_tf_available

if is_tf_available():
    import tensorflow as tf

from .consts import END_KEY, PROMPT_FOR_GENERATION_FORMAT, RESPONSE_KEY

logger = logging.getLogger(__name__)


def load_model_tokenizer_for_generate(
    pretrained_model_name_or_path: str,
) -> Tuple[PreTrainedModel, PreTrainedTokenizer]:
    """Loads the model and tokenizer so that it can be used for generating responses.

    Args:
        pretrained_model_name_or_path (str): name or path for model

    Returns:
        Tuple[PreTrainedModel, PreTrainedTokenizer]: model and tokenizer
    """
    tokenizer = AutoTokenizer.from_pretrained(pretrained_model_name_or_path, padding_side="left")
    model = AutoModelForCausalLM.from_pretrained(
        pretrained_model_name_or_path, device_map="auto", torch_dtype=torch.bfloat16, trust_remote_code=True
    )
    return model, tokenizer


def get_special_token_id(tokenizer: PreTrainedTokenizer, key: str) -> int:
    """Gets the token ID for a given string that has been added to the tokenizer as a special token.

    When training, we configure the tokenizer so that the sequences like "### Instruction:" and "### End" are
    treated specially and converted to a single, new token.  This retrieves the token ID each of these keys map to.

    Args:
        tokenizer (PreTrainedTokenizer): the tokenizer
        key (str): the key to convert to a single token

    Raises:
        ValueError: if more than one ID was generated
```

```
        Returns:
            int: the token ID for the given key
        """
        token_ids = tokenizer.encode(key)
        if len(token_ids) > 1:
            raise ValueError(f"Expected only a single token for '{key}' but found
{token_ids}")
        return token_ids[0]


    class InstructionTextGenerationPipeline(Pipeline):
        def __init__(
            self, *args, do_sample: bool = True, max_new_tokens: int = 256, top_p:
float = 0.92, top_k: int = 0, **kwargs
        ):
            """Initialize the pipeline

            Args:
                do_sample (bool, optional): Whether or not to use sampling. Defaults
to True.
                max_new_tokens (int, optional): Max new tokens after the prompt to
generate. Defaults to 256.
                top_p (float, optional): If set to float < 1, only the smallest set
of most probable tokens with
                    probabilities that add up to top_p or higher are kept for
generation. Defaults to 0.92.
                top_k (int, optional): The number of highest probability vocabulary
tokens to keep for top-k-filtering.
                    Defaults to 0.
            """
            super().__init__(*args,                                do_sample=do_sample,
max_new_tokens=max_new_tokens, top_p=top_p, top_k=top_k,
                             **kwargs)

        def _sanitize_parameters(self,
                                 return_full_text: bool = None,
                                 **generate_kwargs):
            preprocess_params = {}

            # newer versions of the tokenizer configure the response key as a special
token.  newer versions still may
            # append a newline to yield a single token.  find whatever token is
configured for the response key.
            tokenizer_response_key = next(
                (token for token in self.tokenizer.additional_special_tokens if
token.startswith(RESPONSE_KEY)), None
            )

            response_key_token_id = None
            end_key_token_id = None
            if tokenizer_response_key:
                try:
                    response_key_token_id   =   get_special_token_id(self.tokenizer,
tokenizer_response_key)
                    end_key_token_id     =     get_special_token_id(self.tokenizer,
END_KEY)

                    # Ensure generation stops once it generates "### End"
                    generate_kwargs["eos_token_id"] = end_key_token_id
                except ValueError:
                    pass
```

```
            forward_params = generate_kwargs
            postprocess_params = {
                "response_key_token_id": response_key_token_id,
                "end_key_token_id": end_key_token_id
            }

            if return_full_text is not None:
                postprocess_params["return_full_text"] = return_full_text

            return preprocess_params, forward_params, postprocess_params

        def preprocess(self, instruction_text, **generate_kwargs):
            prompt_text                                                    =
PROMPT_FOR_GENERATION_FORMAT.format(instruction=instruction_text)
            inputs = self.tokenizer(
                prompt_text,
                return_tensors="pt",
            )
            inputs["prompt_text"] = prompt_text
            inputs["instruction_text"] = instruction_text
            return inputs

        def _forward(self, model_inputs, **generate_kwargs):
            input_ids = model_inputs["input_ids"]
            attention_mask = model_inputs.get("attention_mask", None)

            if input_ids.shape[1] == 0:
                input_ids = None
                attention_mask = None
                in_b = 1
            else:
                in_b = input_ids.shape[0]

            generated_sequence = self.model.generate(
                input_ids=input_ids.to(self.model.device),
                attention_mask=attention_mask.to(self.model.device),
                pad_token_id=self.tokenizer.pad_token_id,
                **generate_kwargs,
            )

            out_b = generated_sequence.shape[0]
            if self.framework == "pt":
                generated_sequence = generated_sequence.reshape(in_b, out_b // in_b,
*generated_sequence.shape[1:])
            elif self.framework == "tf":
                generated_sequence = tf.reshape(generated_sequence, (in_b, out_b //
in_b, *generated_sequence.shape[1:]))

            instruction_text = model_inputs.pop("instruction_text")
            return    {"generated_sequence":    generated_sequence,    "input_ids":
input_ids, "instruction_text": instruction_text}

        def      postprocess(self,      model_outputs,      response_key_token_id,
end_key_token_id, return_full_text: bool = False):

            generated_sequence = model_outputs["generated_sequence"][0]
            instruction_text = model_outputs["instruction_text"]

            generated_sequence:                  List[List[int]]                  =
generated_sequence.numpy().tolist()
            records = []
```

```
            for sequence in generated_sequence:

                # The response will be set to this variable if we can identify it.
                decoded = None

                # If we have token IDs for the response and end, then we can find
the tokens and only decode between them.
                if response_key_token_id and end_key_token_id:
                    # Find where "### Response:" is first found in the generated
tokens.  Considering this is part of the
                    # prompt, we should definitely find it.  We will return the
tokens found after this token.
                    try:
                        response_pos = sequence.index(response_key_token_id)
                    except ValueError:
                        logger.warn(f"Could     not     find     response     key
{response_key_token_id} in: {sequence}")
                        response_pos = None

                    if response_pos:
                        # Next find where "### End" is located.  The model has been
trained to end its responses with this
                        # sequence (or actually, the token ID it maps to, since it
is a special token).  We may not find
                        # this token, as the response could be truncated.  If we
don't find it then just return everything
                        # to the end.  Note that even though we set eos_token_id, we
still see the this token at the end.
                        try:
                            end_pos = sequence.index(end_key_token_id)
                        except ValueError:
                            end_pos = None

                        decoded = self.tokenizer.decode(sequence[response_pos + 1 :
end_pos]).strip()

                if not decoded:
                    # Otherwise we'll decode everything and use a regex to find the
response and end.

                    fully_decoded = self.tokenizer.decode(sequence)

                    # The response appears after "### Response:".  The model has
been trained to append "### End" at the
                    # end.
                    m = re.search(r"#+\s*Response:\s*(.+?)#+\s*End", fully_decoded,
flags=re.DOTALL)

                    if m:
                        decoded = m.group(1).strip()
                    else:
                        # The model might not generate the "### End" sequence before
reaching the max tokens.  In this case,
                        # return everything after "### Response:".
                        m   =   re.search(r"#+\s*Response:\s*(.+)",   fully_decoded,
flags=re.DOTALL)
                        if m:
                            decoded = m.group(1).strip()
                        else:
                            logger.warn(f"Failed       to       find       response
in:\n{fully_decoded}")
```

```
                # If the full text is requested, then append the decoded text to the
original instruction.
                # This technically isn't the full text, as we format the instruction
in the prompt the model has been
                # trained on, but to the client it will appear to be the full text.
                if return_full_text:
                    decoded = f"{instruction_text}\n{decoded}"

                rec = {"generated_text": decoded}

                records.append(rec)

            return records


    def generate_response(
        instruction: str,
        *,
        model: PreTrainedModel,
        tokenizer: PreTrainedTokenizer,
        **kwargs,
    ) -> str:
        """Given an instruction, uses the model and tokenizer to generate a response.
This formats the instruction in
        the instruction format that the model was fine-tuned on.

        Args:
            instruction (str): _description_
            model (PreTrainedModel): the model to use
            tokenizer (PreTrainedTokenizer): the tokenizer to use

        Returns:
            str: response
        """

        generation_pipeline    =    InstructionTextGenerationPipeline(model=model,
tokenizer=tokenizer, **kwargs)
        return generation_pipeline(instruction)[0]["generated_text"]
```

consts.py

```
DEFAULT_INPUT_MODEL = "EleutherAI/pythia-6.9b"
SUGGESTED_INPUT_MODELS = [
    "EleutherAI/pythia-2.8b",
    "EleutherAI/pythia-6.9b",
    "EleutherAI/pythia-12b",
    "EleutherAI/gpt-j-6B",
    "databricks/dolly-v2-3b",
    "databricks/dolly-v2-7b",
    "databricks/dolly-v2-12b"
]
DEFAULT_TRAINING_DATASET = "databricks/databricks-dolly-15k"
INTRO_BLURB = (
    "Below is an instruction that describes a task. Write a response that appropriately completes the request."
)
INSTRUCTION_KEY = "### Instruction:"
INPUT_KEY = "Input:"
RESPONSE_KEY = "### Response:"
END_KEY = "### End"
RESPONSE_KEY_NL = f"{RESPONSE_KEY}\n"
DEFAULT_SEED = 42

# This is a training prompt that does not contain an input string.  The instruction by itself has enough information
# to respond.  For example, the instruction might ask for the year a historic figure was born.
PROMPT_NO_INPUT_FORMAT = """{intro}

{instruction_key}
{instruction}

{response_key}
{response}

{end_key}""".format(
    intro=INTRO_BLURB,
    instruction_key=INSTRUCTION_KEY,
    instruction="{instruction}",
    response_key=RESPONSE_KEY,
    response="{response}",
    end_key=END_KEY,
)

# This is a training prompt that contains an input string that serves as context for the instruction.  For example,
# the input might be a passage from Wikipedia and the intruction is to extract some information from it.
PROMPT_WITH_INPUT_FORMAT = """{intro}

{instruction_key}
{instruction}

{input_key}
{input}

{response_key}
{response}

{end_key}""".format(
    intro=INTRO_BLURB,
```

```
            instruction_key=INSTRUCTION_KEY,
            instruction="{instruction}",
            input_key=INPUT_KEY,
            input="{input}",
            response_key=RESPONSE_KEY,
            response="{response}",
            end_key=END_KEY,
        )

        # This is the prompt that is used for generating responses using an already
trained model.  It ends with the response
        # key, where the job of the model is to provide the completion that follows it
(i.e. the response itself).
        PROMPT_FOR_GENERATION_FORMAT = """{intro}

        {instruction_key}
        {instruction}

        {response_key}
        """.format(
            intro=INTRO_BLURB,
            instruction_key=INSTRUCTION_KEY,
            instruction="{instruction}",
            response_key=RESPONSE_KEY,
        )
```