Реферат

Общий объем основного текста, без учета приложений 35 страниц, с учетом приложений 38. Количество использованных источников 22. Количество приложений 1.

КЛЮЧЕВЫЕ СЛОВА: LLM, GAN, FINE TUNING, PROMPT TUNING, RL

Целью данной работы является разработка метода генеративного дообучения больших языковых моделей.

В первой главе проведен обзор и анализ существующих методов дообучения больших языковых моделей.

Во второй главе описывают метод дообучения и метод оценивания языковой модели с теоретической стороны.

В третьей главе приводится описание программной реализации и ее тестирование.

В четвертой главе описывают исследование работы разработанного программного модуля и его сравнение с существующими аналогами.

Содержание

Введение

В последнее время стали очень популярны большие языковые модели. Эти модели, благодаря своей масштабной архитектуре и обучению на огромных корпусах текста, демонстрируют впечатляющие результаты в самых разных областях, от естественного языка до компьютерного зрения. Однако, их потенциал полностью раскрывается лишь в контексте конкретных задач и предметных областей.

С учетом огромного спектра применений этих моделей: в медицине [1], в обучении программированию [2; 3], в экономике [4], в кибербезопасности [5], в поиске данных, в генерации контента, в переводе текстов - множество предприятий и исследовательских групп стремятся интегрировать их в свои проекты и продукты. Однако, обучение большой языковой модели с нуля является крайне затратным процессом, как в плане вычислительных ресурсов, так и временных затрат.

Именно поэтому приобрели большую популярность методы дообучения предварительно обученных моделей. Эти методы позволяют адаптировать существующие и предварительно обученные модели к конкретным задачам, минимизируя затраты на обучение и значительно повышая эффективность использования. Благодаря процессу дообучения, модели становятся более адаптированными и специализированными, что приводит к значительному повышению их точности и релевантности в контексте конкретных прикладных задач. Можно выделить следующие виды дообучения:

1. Тонкая настройка (FMT, LoRA, настройка префикса, настройка адаптера)
2. Инженерия промптов (zero-shot prompting, few-shot prompting, метод цепочки мыслей, метод самосогласованности)
3. Другие методы (Prompt tuning, RAG, RLHF, RLAIF)

В настоящей работе представлен обзор современных больших языковых моделей и подходов к их дообучению. Дополнительно предоставляется описание метода дообучения языковых моделей, разработанного в рамках данной работы, а также способ его оценки. Кроме того, в рамках работы представлен программный модуль, реализующий разработанный метод. Также работа включает в себя тестирование предложенного метода и его экспериментальное сопоставление с альтернативными подходами к дообучению.

1. Обзор методов дообучения больших языковых моделей

В данной главе приводится обзор больших языковых моделей и существующих методов дообучения.

1.1 Обзор больших языковых моделей

В настоящее время стали очень популярны большие языковые модели (Large language models - LLM). LLM - это большая нейронная сеть, обученная на огромном наборе данных, которая воспринимает текст в виде многомерных векторов - эмбедингов. Обычно эмбединги устроены так, что слова со схожими контекстными значениями или другими взаимосвязями находились близко друг к другу в векторном пространстве. Кроме того, в статьях [2; 5] выделяют следующие свойства, которыми должна обладать современная LLM:

– улучшенные возможности понимания и генерации: эти модели демонстрируют лучшее понимание контекста, грамматики и семантики, что приводит к более связному и контекстуально точному созданию текста.
– способность генерировать текст, похожий на человеческий.

К самым популярным современным LLM можно отнести следующие модели: ChatGPT и GPT-4 от OpenAI, PALM-2 и Gemini от Google, Llama-2 от Meta, Grok-1 от X.Ai, YandexGPT-2 от Yandex, GigaChat от Сбер.

Стоит отметить, что Hugging Face является самой популярной платформой для размещения пред обученных LLM. Кроме того, на этой платформе можно найти рейтинг открытых больших языковых моделей. Анализируя этот рейтинг, представленный на Рисунке 1.1, можно прийти к выводу, что самыми производительными открытыми языковыми моделями являются различные вариации следующих моделей: Mistral и LLaMa.

Model	Average	ARC	HellaSwag	MMLU	TruthfulQA	Winogrande	GSM8K
alykassem/ds_diasum_md_mixtral	68.42	66.3	85.45	69.51	55.72	80.35	53.22
ajibawa-2023/Code-Mistral-7B	68.4	63.57	83.71	63.38	51.81	81.22	66.72
TheBloke/neural-chat-7B-v3-2-GPTQ	66.93	65.96	83.24	60.29	59.79	79.48	52.84
Zangs3011/mixtral_8x7b_MonsterInstruct	66.34	65.19	85.81	70.15	48.47	80.27	48.14
jrahn/llama-3-8b-claudstruct-v2	66.32	59.9	80.01	64.8	51.87	75.61	65.73
jrahn/llama-3-8b-claudstruct-v1	65.9	59.73	79.94	64.98	51.82	74.43	64.52
ibivibiv/llama-3-8b-instruct-alpaca-gpt-4	65.72	59.13	79	65.23	53.87	75.69	61.41
jrahn/llama-3-8b-claudstruct-v3	65.62	58.96	80.05	64.55	51.76	74.19	64.22
robinsmits/Mistral-Instruct-7B-v0.2-ChatAlpacaV2-4bit	65.34	62.12	84.55	60.66	67.29	77.11	40.33

Рисунок 1.1 – Рейтинг Open LLM

1.2 Обзор методов тонкой настройки

Тонкая настройка (fine tuning) — это класс методов дообучения LLM, при котором корректируют веса уже обученной языковой модели. Этот подход очень полезен при ограниченном количестве данных. Так как LLM уже обучена на большом объеме текстов и обладает лингвистическими знаниями, то остается адаптировать ее к нюансам решаемой задачи или предметной области. Кроме того, тонкая настройка обладает меньшей вычислительной сложностью из-за меньшего объема обучающих данных.

1.2.1 Метод полной настройки

К тонкой настройке относится метод полной настройка модели (full model tuning - FMT), описанный в статье [6]. При данном методе вычисляют функцию потерь на обучающей выборке, и осуществляют градиентный спуск, изменяя все веса, входящие в LLM. Проблема данного метода в том, что хоть он вычислительно менее сложен, чем обучение LLM с нуля, он все равно остается вычислительно затратным и дорогостоящим. Еще одной проблемой является возможное катастрофическое забывание. Катастрофическое забывание — это явление, когда LLM при настройке к конкретной задаче может забыть общие знания, то есть начать хуже отвечать на вопросы, не относящиеся к предметной области, к которой проводилось дообучение.

1.2.2 Метод эффективной настойки параметров

Также к тонкой настройке относится класс методов - эффективная настройка параметров (parameter efficient fine-tuning - PEFT). PEFT предполагает настройку небольшого количества весов, входящих в LLM. Это приводит к значительному снижению требований к памяти и вычислениям, необходимым для дообучения. Более того этот метод позволяет бороться с катастрофическим забыванием.

1.2.3 Адаптация низкого ранга

Одним из подходов PEFT является адаптация низкого ранга (low-rank adaptation - LoRA), описанная в статье [7]. В слоях LLM есть матрицы весов W. При дообучении вычисляются некоторые матрицы ΔW, которые прибавляют к исходным матрицам W. Обычно матрицу ΔW вычисляют с помощью градиентного спуска. Идея подхода LoRA зафиксировать веса матрицы W и обучать только матрицу ΔW. Так как матрица W и матрица ΔW имеют одинаковые размерности выигрыша от такого подхода не будет. Поэтому матрицу ΔW представ-

ляют в виде произведения двух других матриц меньшей размерности. Схематично разбиение показано на Рисунке 1.2 из статьи [7]. Такой подход позволяет обучать на порядки меньше

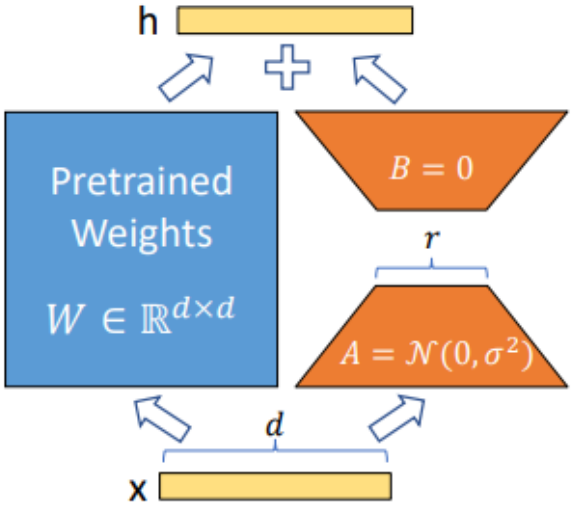

Рисунок 1.2 – Разбиение матрицы весов в методе LoRA

параметров. Достаточность матриц низкого ранга обуславливается в статье [7] гипотезой, что ”внутренняя размерность” (intrinsic rank) у LLM очень низкая. На практике, при гораздо меньших вычислительных затратах, LoRA имеет почти такую же эффективность, как и метод FMT.

В статье [8] описывают модификацию подхода LoRA - QLoRA. Хоть LoRA настраивает не все веса, она все равно вычислительно сложная. Поэтому в методе QLoRA используют квантование весов LLM. Обычно веса хранятся в 32-битном формате. При квантовании веса конвертируются в 4-битный формат. Идея метода состоит в том, что большинство весов языковой модели близко к нулю. При такой конвертации веса близкие к нулю будут иметь большую значимость. Исследования показывают, что такой метод позволяет сохранить примерную эффективность классического подхода LoRA, требуя меньших вычислительных затрат.

1.2.4 Настройка префикса

Другим подходом PEFT является настройка префикса (prefix tuning). Идея этого метода связана с тем, что добавляя дополнительный контекст к запросу, который потом передается LLM, можно получить более точный ответ. Поэтому к некоторым слоям языковой модели добавляют дополнительные параметры, отвечающие за контекст. Именно эти параметры и настраивают, не трогая изначальные веса языковой модели. Схематично это представлено

на Рисунке 1.3 из статьи [9]. В статье [9] делают вывод, что настройка префикса показывает

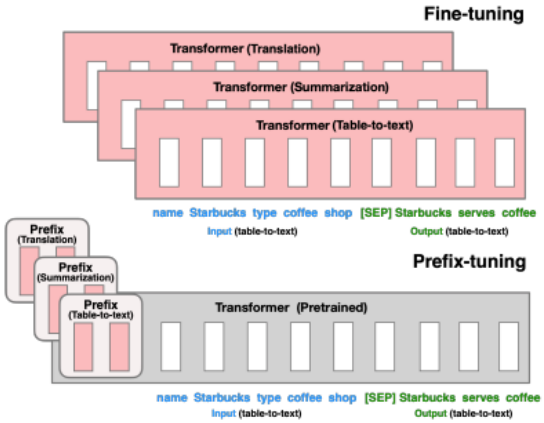

Рисунок 1.3 – Архитектура метода настройки префикса

производительность на уровне производительности при FMT.

1.2.5 Настройка адаптера

Другим метод, похожий на настройку префикса и относящийся к PEFT, является настройка адаптера. При этом подходе в LLM добавляются дополнительные слои, параметры которых настраивают, не трогая изначальные веса моделей. Добавляемые слои состоят из четырех компонентов. Первый компонент — это линейное преобразование, уменьшающее размерность входных данных. Уменьшая размерность, получается вектор, концентрирующийся на более значимых параметрах для предметной области. Следующий компонент применяет нелинейную функции активации к данным. Он необходим для установления сложных взаимосвязей, связанных с предметной областью. Третий компонент восстанавливает изначальную размерность данных. Последний компонент осуществляет суммирование полученного результата и исходных данных, которые подавались на вход адаптера. Архитектура адаптера представлена на Рисунке 1.4 из статьи [10].

У всех описанных методов, относящихся к PEFT, можно выделить следующие свойства:

- Меньшую вычислительную сложность по сравнению с FMT. Однако это все равно ресурсоемкие методы.
- Преодоление катастрофического забывания, свойственного FMT.
- Большая производительность при небольшом объеме обучающих данных по сравнению с FMT.
- Сравнимая с FMT эффективность.

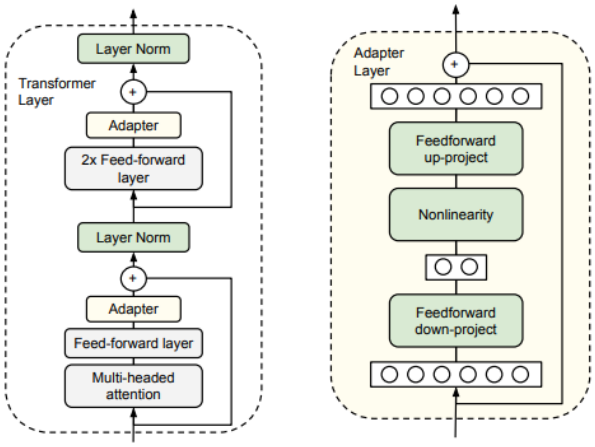

Рисунок 1.4 – Архитектура метода настройки адаптера

1.3 Обзор методов инженерии промптов

В последнее время очень популярным стал способ дообучения связанный с промптами - инженерия промптов (prompt engineering). Промпт это запрос, который подают на вход языковой модели. В инженерии промптов никак не меняют веса языковой модели.

1.3.1 Техника нулевой разметки

Одной их техник инженерии промптов является техника нулевой разметки (zero-shot prompting). При этом методе языковой модели не передают никаких инструкций заранее. В этом методе подсказки для LLM подаются в виде контекста с самим промптом. Например: решай задачу шаг за шагом, классифицируй текст, как нейтральный, негативный или позитивный.

1.3.2 Техника нескольких подсказок

Другой техникой является техника нескольких подсказок (few-shot prompting). В этой технике в промпт добавляется несколько примеров в виде инструкций. Инструкция представляет из себя: запрос, который могли бы передать языковой модели, и ответ, который должна была бы выдать языковая модель. Используя эту технику, можно задать стиль и формат ответа LLM.

1.3.3 Техника цепочки мыслей и метод самосогласованности

Цепочка мыслей (Chain-of-Thought Prompting) - техника, которую применяют, когда требуется решить математическую задачу или задачу, требующую аналитических рассуждений.

Этот метод использует технику нескольких подсказок, но в инструкции дополнительно указывают способ рассуждения, которым был получен ответ.

Модификацией цепочки мыслей является метод самосогласованности (Self-Consistency), предложенный в статье [11]. В отличии от цепочки мыслей в методе самосогласованности содержится больше инструкций с различными путями рассуждений. В этом методе мы несколько раз запускаем LLM, подавая каждый раз один и тот же промпт. Затем нужно выбрать наиболее встречающийся вариант из полученных ответов.

В итоге, у методов из инженерии промптов можно выделить следующие свойства: высокую эффективность, отсутствие вычислительных затрат, сильную ограниченность применения. Еще одним важным недостаткам является отсутствие обоснования.

1.4 Обзор других методов дообучения

1.4.1 Prompt tuning

Prompt tuning - метод дообучения, занимающий промежуточное положение между тонкой настройкой и инженерией промптов, принцип работы которого описан в статье [12]. В инженерии промптов использовались “жесткие” промпты: добавление примеров или подсказок в запрос, передаваемый на вход LLM. Prompt tuning работает с “мягкими” промптами: добавление в запрос последовательности символов, которая скорее всего будет бессмысленной с лингвистической точки зрения. Эта последовательность добавляется небольшой моделью, которую необходимо обучить (Prompt tuning имеет несколько подходов реализации. Все подходы связаны с преобразованием входной последовательности. В данной главе описан один из самых популярных). Этот метод похож на метод настройки префикса, но в настройке префикса добавляются дополнительные параметры к нескольким слоям LLM, а в prompt tuning добавляются параметры только перед входным слоем. Данный метод обладает большой эффективностью, но также имеет проблему с обоснованностью: неизвестно почему добавление именно таких последовательностей улучшает точность модели.

1.4.2 Генерация ответа с использованием результатов поиска

Другим методом с подходом, похожим на подход инженерии промптов, является генерация ответа с использованием результатов поиска (Retrieval Augmented Generation - RAG). Данный метод предполагает создание системы, работающей по следующему алгоритму:

1. Пользователь вводит вопрос
2. Система ищет подходящие документы, которые могут содержать ответ на вопрос, вво-

димый пользователем. Эти документы часто включают в себя собственные данные компании, которая разработала систему, а хранятся они обычно в некотором каталоге документов.

3. Составляется промпт, в котором найденные документы являются контекстом для вопроса пользователя.
4. Промпт передают в LLM

Архитектура RAG, реализующая данный алгоритм, описана в статье [13]. Данная архитектура состоит из двух компонентов: поискового компонента и генератора. В поисковом компоненте кодировщик запроса преобразует запрос от пользователя в вектор из чисел. Затем кодировщик документов преобразует каждый документ в вектор чисел и сохраняет их в поисковом индексе. На следующем шаге поисковой компонент находит в поисковом индексе документы, которые относятся к запросу пользователя. На последнем шаге поисковой компонент преобразует найденные документы обратно в текст и передает их вместе с текстом запроса генератору. Генератор объединяет текст документов с текстом запроса в один промпт и передает его LLM. Ответ языковой модели и будет ответом всей системы. В статье [13] отмечают, что при обучении модели с такой архитектурой стоит проводить совместное обучение только генератора запроса и LLM. При этом не стоит обучать кодировщик документов по двум причинам. Во-первых, из-за затратности этого процесса. Во-вторых, без дообучения кодировщика документов можно добиться хороших результатов работы системы. Причины, по которым используют RAG:

– Языковые модели могут генерировать общую или устаревшую информацию
– Предметная область, с которой планируется работать, с течением времени быстро пополняется новой информацией
– Имеется большое количество документов, среди которых часто нужно найти ответ

Недостатками данного метода являются затраты, необходимые для поддержания хранения документов, используемых в методе, а также поддержания актуальной информации, то есть обновления документов.

1.4.3 Обучение с подкреплением на основе обратной связи

Еще одним методом дообучения является обучение с подкреплением на основе обратной связи от человека (Reinforcement Learning with Human Feedback – RLHF). В данном методе необходим сбор отзывов от людей о правильности и качестве ответов LLM. Полученные ответы используются в качестве сигнала для вознаграждения или наказания в обучении с под-

креплением языковой модели. Метод довольно эффективный, но требует больших затрат на сбор и хранение отзывов от людей. Модификацией данного метода является обучение с подкреплением на основе обратной связи от искусственного интеллекта (Reinforcement Learning from AI Feedback - RLAIF). Отличие состоит в том, что в RLAIF используют отзывы, созданные некоторой моделью искусственного интеллекта. Например, в статье [14] реализуют RLAIF с использованием генеративной состязательной сети.

1.4.4 Генеративная состязательная сеть

Генеративная состязательная сеть (Generative Adversarial Networks - GAN) состоит из двух нейронных сетей: генератора и дискриминатора. Изначально генератор создает данные из распределения, которое называют шумом. К примеру, в качестве шума может выступать нормальное распределение. Дискриминатор представляет собой двоичный классификатор. Его обучают на реальных данных, размеченных значением “1”, и данных, созданных генератором, размеченных значением “0”. Используя размеченные данные и значения на выходе дискриминатора, вычисляют значение функции потерь, которое используют для обучения генератора. Таким образом генератор учится лучше обманывать дискриминатор, создавая данные похожие на реальные, а дискриминатор учится отличать сгенерированные и реальные данные. Архитектура GAN представлена на Рисунке 1.5 из статьи [15].

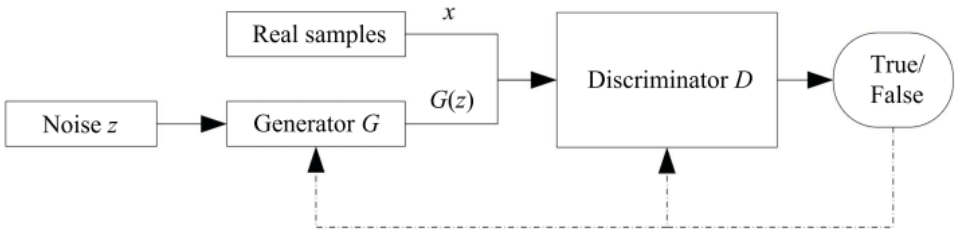

Рисунок 1.5 – Архитектура GAN

Обоснование того, что генератор начинает создавать ”правдоподобные” данные, связано с тем, что генератор в ходе обучения меняет свое начальное распределение на распределение реальных данных. GAN обладает хорошей эффективностью, но вычислительно сложен. Также к недостаткам можно отнести следующую трудность: сложность настройки обучения. Обучение нужно настроить так, чтобы дискриминатор и генератор обучались на “равных”. Если генератор будет обучен гораздо лучше дискриминатора, то дискриминатор будет принимать все сгенерированные данные за настоящие, и соответственно дальнейшее обучение генератора не будет происходить. Если дискриминатор будет обучен гораздо лучше генера-

тора, то это будет означать, что генератор создает нереалистичные данные.

В статье [14] используют LLM в качестве генератора и дискриминатор в качестве обратной связи для RLAIF. В статье [14] отмечают, что необходимы дополнительные исследования, но уже можно утверждать, что данный подход имеет высокую эффективность, но обладает сложностью в правильном обучении.

1.5 Выводы

В ходе обзора существующих методов дообучения были получены следующие выводы:

1. Методы тонкой настройки обладают высокой эффективностью, но требуют больших вычислительных затрат. Кроме этого, для них необходимо наличие высококачественных обучающих данных.
2. Методы инженерии промптов обладают неплохой эффективностью и почти не требуют дополнительных данных. К недостаткам можно отнести: отсутствие обоснования методов, ограниченность применения.
3. Prompt tuning не очень вычислительно затратен, но при этом имеет большую эффективность. Но, как и методы инженерии промптов, имеет некоторые проблемы с обоснованностью.
4. К недостаткам методов RAG и RLHF относятся затраты на хранение дополнительных данных и их обновление.
5. Метод RLAIF с использованием GAN обладает высокой эффективностью, но имеет сложности при обучении.

1.6 Цели и задачи УИР

Поскольку prompt tuning является новым популярным методом с большой эффективностью, то подход этого метода был выбран в качестве основы для разрабатываемого метода дообучения. Prompt tuning предполагает обучение небольшой модели, которая будет преобразовывать входной запрос, который затем будет передан в LLM. В качестве такой модели был выбран генератор из генеративной состязательной сети, поскольку генератор и дискриминатор из GAN обладают впечатляющей эффективностью после обучения. Таким образом основной целью работы является разработка метода дообучения, основанного на подходе prompt tuning, с использованием генеративной состязательной сети.

К основным задачам данной работы относятся:

1. Формальная постановка задачи дообучения LLM
2. Разработка метода дообучения

3. Разработка метода оценивания дообученной модели
4. Выбор языка и среды программирования для реализации метода дообучения
5. Программная реализация модуля дообучения
6. Тестирование разработанного модуля
7. Экспериментальные исследования разработанного метода дообучения больших языковых моделей

2. Разработка моделей и алгоритмов дообучения языковых моделей

В данной главе приводится формальная постановка метода дообучения, сам метод дообучения, а также метод оценки обученной модели.

2.1 Формальная постановка задачи дообучения больших языковых моделей

Формальная постановка задачи дообучения LLM включает в себя следующие этапы:

1. **Настройка модели.** Пусть $M_{pretrain}$ и $T_{pretrain}$ - предварительно обученная LLM и ее токенизатор соответственно. Необходимо определить архитектуру генератора G, состоящего из $M_{pretrain}$ и дополнительной модели $M_{tunable}$, архитектуру которого тоже необходимо определить. Также необходимо определить архитектуру дискриминатора D.
2. **Сбор и предобработка данных.** Пусть $B = \{(q_i, r_i)\}_{i=1}^{n}$ - набор данных, где q_i - вопрос в текстовом формате, r_i - ответ на вопрос q_i в текстовом формате, n - количество обьектов в выборке. Предобработка данных заключается в токенизации B и приведении к одной длине токенизерованных последовательностей. Набор данных после предобработки: $B^* = \{Y_{0:T}^{i(real)}\}_{i=1}^{n} = \{(y_0^i, y_1^i, \ldots, y_T^i)\}_{i=1}^{n}$, где $y_0^i = (y_{01}^i, \ldots, y_{0N}^i)$ это токенизированный с помощью $T_{pretrain}$ вопрос q_i, а последовательность $(y_1^i, \ldots, y_T^i)$ - токенизированный с помощью $T_{pretrain}$ ответ a_i. Пусть $B_{train}^* = \{Y_{0:T}^{i(real)}\}_{i=1}^{m}$ - обучающая выборка данных, $B_{test}^* = \{Y_{0:T}^{i(real)}\}_{i=m+1}^{n}$ - тестовая выборка данных.
3. **Дообучение модели.** Необходимо разработать метод дообучения для генератора G с использованием B_{train}^*, при котором изменяются только веса модели $M_{tunable}$. Также необходимо обучить дискриминатор D, используя B_{train}^* и сгенерированные с помощью генератора G данные, для минимизации функции потерь - бинарной кросс-энтропии.
4. **Оценка эффективности.** Необходимо выбрать методы оценки для обученной модели G на выборке B_{test}^*.

2.2 Разработка метода дообучения

2.2.1 Метод дообучения генератора

Идея prompt-tuning заключается в разработке модели, которая обучается добывать дополнительную информацию из входной последовательности. $M_{tunable}$ - является нейронной сетью, которая реализует модель из prompt-tuning, сочетая при этом в себе идеи генератора из классической архитектуры GAN.

Большие языковые модели имеют слой или слои эмбеддингов (эмбеддинг - векторное представление входной последовательности токенов LLM). Результат слоя/слоев эбеддингов $Z = (z_1, \ldots, z_M)$ от модели $M_{pretrain}$ вместе вектором случайных чисел $X = (x_1, \ldots, x_100)$, где x_i $N(0, 1)$ для $i = 1, \ldots, 100$, подается на вход $M_{tunable}$.

$M_{tunable}$ имеет 4 слоя: первые два слоя снижают размерность входной последоватльности, следующие два слоя повышают размерность последоватльности до размерности входного эмбеддинга. Функцией активации для трех первых слоев является $LeakyReLU(x) = max(0, x)+0.01min(0, x)$. У последнего слоя линейная функция активации. Выходом $M_{tunable}$ является сумма 4 слоя и входного эмбеддинга. Таким образом, архитектура $M_{tunable}$ напоминает архитектуру слоя адаптера из метода настройки адаптера. Архитектура $M_{tunable}$ представлена на Рисунке 2.1.

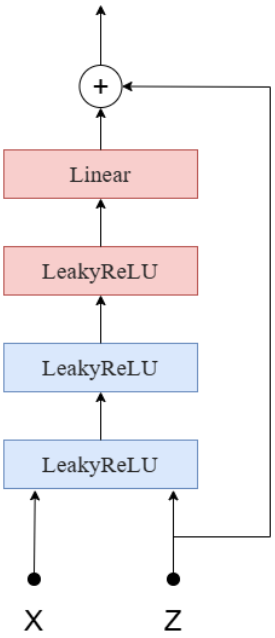

Рисунок 2.1 – Архитектура модели $M_{tunable}$

Генератор G представляет из себя $M_{pretrain}$, у которой между слоем эмбеддинга и бло-

ком трансформеров расположена модель $M_{tunable}$. Архитектура генератора G представлена на Рисунке 2.2.

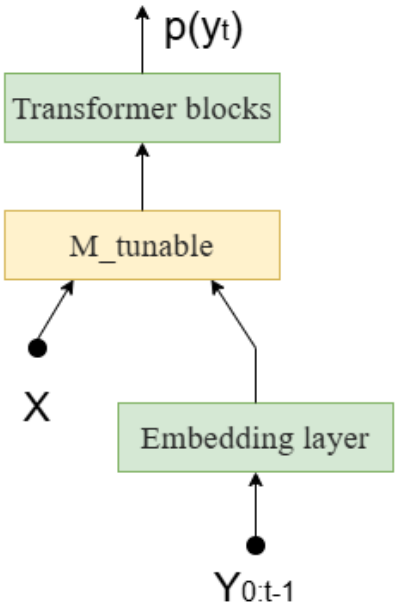

Рисунок 2.2 – Архитектура модели G

Большие языковые модели, разработанные на основе архитектур трансформеров, принимают на вход последовательность токенов, а имеют на выходе вероятности для всех токенов в словаре, означающие вероятность того, что именно этот токен является продолжением последовательности. Поскольку генератор G имеет на выходе только один токен, а ответ из обучающей выборки B^*_{train} представляет собой последоватльность токенов, то было принято решение использовать метод SeqGan, предложенный в статье [16]. Подход SeqGAN использует подход градиента стратегии (policy gradient) из обучения с подкреплением (reinforcement learning - RL). Исходя из этого подхода генератор G моделирует политику напрямую:

$$G = \pi(a_t = y_t | s_t = Y_{0:t-1}) \tag{2.1}$$

В этом подходе дискриминатор D рассматривается как нейронная сеть моделирующая функцию награды. С учетом того, что дискриминатор вычисляет награду только по итоговой последовательности токенов $Y_{0:T}$, сгенерированной генератором G, моделируемая функция наград принимает следующий вид:

$$D = \sum_{t=1}^{T} r(a_t = y_t | s_t = Y_{0:t-1}) = r(a_t = y_T | s_t = Y_{0:T-1}) = r(Y_{0:T}) \tag{2.2}$$

Тогда задача обучения генератора состоит в максимизации следующего функционала:

$$J(\theta) = \mathbb{E}_{Y_{0:T}\ \pi_\theta} = \sum_{t \geq 0} r(Y_{0:T}) \rightarrow \max_\theta \tag{2.3}$$

где θ - параметры модели $M_{tunable}$.

Для максимизации функционала $J(\theta)$, выбран Монте-Карло градиент стратегии, представленный в статье [17]:

$$\nabla_\theta J = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=1}^{T} \nabla_\theta log \pi_\theta(a_t = y_t | s_t = Y_{0:t-1}) r(Y_{0:T}) \tag{2.4}$$

2.2.2 Метод дообучения дискриминатора

Дискриминатор D состоит из 4 слоев, в которых постепенно снижается размерность входной последовательности. Первые три слоя имеют функцию активации $LeakyReLU$, а последний слой - сигмоиду. Архитектура дискриминатора D представлена на Рисунке 2.3.

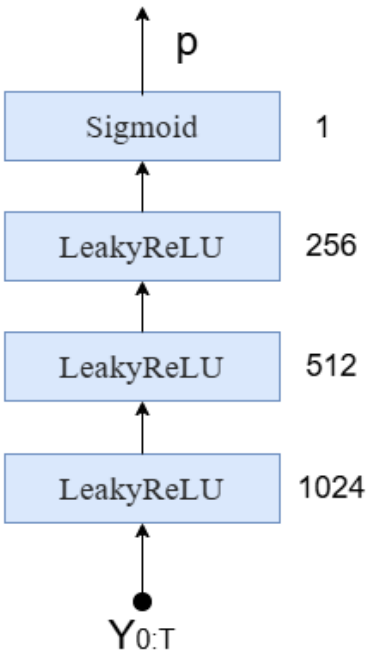

Рисунок 2.3 – Архитектура модели D. Справа от слоев подписаны количества нейронов на ”выходе” из слоя

Обучение дискриматора D происходит по классической для GAN схеме. Для обучения используются реальные данные $Y_{0:T}^{(real)}$ и сгенерированные с помощью генератора $Y_{0:T}$. Задача дискриминатора классифицировать реальные и сгенерированные данные, поэтому функ-

ция потерь для дискриминатора имеет следующий вид:

$$L = -\frac{1}{n}\sum_{i=1}^{n} log(D(Y_{0:T}^{i(real)})) + log(1 - D(Y_{0:T}^{i})) \tag{2.5}$$

Таким образом для обучения итоговой модели необходимо последовательно обучать дискриминатор D и генератор G.

2.3 Разработка метода оценивания дообученной модели

2.3.1 Метод оценивания ROUGE

Для оценки эффективности дообученной модели G необходимо определить метрики, которые будут использоваться для оценки ключевых показателей. Метрики позволяют количественно оценить качество работы модели и дают возможность сравнивать модели между собой. Поскольку набор данных B содержит вопросы и эталонные ответы, для оценки близости ответов модели и эталонных ответов можно использовать метрику ROUGE (Recall-Oriented Understudy for Gisting Evaluation) из статьи [18], которая равна:

$$ROUGE - N = \frac{\sum\limits_{S \in \{(r_i, r_i^{(generated)})\}_{i=1}^{k}} \sum\limits_{gram_n \in S}^{n} Count_{match}(gram_n)}{\sum\limits_{S \in \{(r_i, r_i^{(generated)})\}_{i=1}^{k}} \sum\limits_{gram_n \in S}^{n} Count(gram_n)} \tag{2.6}$$

где:

- $\{(r_i, r_i^{(generated)})\}_{i=1}^{k}$ – пара из i-го ответа из B_{test}^{*} и i-го ответа от генератора G, преобразованного в текст с помощью $T_{pretrain}$.
- $Count_{match}(n)$ – совпадающее количество n-грамм среди эталонных и сгенерированных ответов.
- $Count(n)$ – общее количество n-грамм среди эталонных и сгенерированных ответов.

2.3.2 Метод оценивания BLEURT

Оценка ROUGE относится к статистическим метрикам оценки. Такие метрики при работе с большими языковыми моделями обладают хорошей надежностью, но малой точностью. Противоположными свойствами обладают метрики, построенные на основе нейросетей, например метрики, использующие LLM. Поэтому в качестве второй метрики был выбран BLEURT (Bilingual Evaluation Understudy with Representations from Transformers), описанный в статье [19]. Данная метрика использует пред обученную языковую модель BERT,

представленную в статье [20], для сравнения семантической схожести эталонных и сгенерированных ответов – $\{(r_i, r_i^{(generated)})\}_{i=1}^{k}$.

2.4 Выводы

В ходе разработки метода дообучения больших языковых моделей были получены следующие выводы:

1. Разработаны архитектуры генератора G и дискриминатора D.
2. Разработан метод дообучения на основе SeqGAN с использованием градиента стратегии Монте-Карло.
3. Выбраны метрики для оценки обученной модели G: ROUGE и BLEURT.

3. Программная реализация метода дообучения

В данной главе приводится формулировка требований к программной реализации метода дообучения и сама программная реализация. В конце главы представлены результаты тестирование разработанного программного модуля

3.1 Выбор языка и модулей для реализации метода дообучения

Для реализации метода дообучения больших языковых моделей, нужно выбрать соответствующий язык программирования и комплект модулей. На выбор языка и инструментов влияют несколько важных аспектов: производительность, наличие библиотек для работы с нейронными сетями и языковыми моделями, а также удобство разработки.

Таким образом язык должен удовлетворять следующим условиям:

1. иметь специализированные библиотеки, работающие с нейронные сетями и LLM
2. иметь высокую производительность при работе с LLM.
3. поддерживать работу с графическим процессором для ускорения работы с нейронными сетями.

Под эти требования подходит язык Python, имеющий большое количество подходящих под требования библиотек. Кроме того, этот язык очень удобен в использовании.

Также для реализации метода дообучения были выбраны следующие модули:

1. PyTorch — фреймворк машинного обучения для языка Python с открытым исходным кодом, созданный на базе Torch. PyTorch позволяет легко и быстро создавать сложные модели благодаря модульной структуре, что особенно полезно для исследований и прототипирования. Также PyTorch обеспечивает отличную поддержку GPU с помощью CUDA, что позволяет значительно ускорить обучение и прямое распространение моделей.
2. Платформа Hugging Face это коллекция готовых современных предварительно обученных моделей глубокого обучения в том числе LLM. Поэтому была выбрана библиотека Transformers, которая предоставляет инструменты и интерфейсы для их простой загрузки и использования.

3.2 Разработка требований к программной системе

Программный модуль, реализующий метод обучения больших языковых моделей, должен соответствовать ряду важных требований для обеспечения его корректной и эффектив-

ной работы. Предъявляются следующие функциональные требования к программной реализации:

1. Модуль должен уметь генерировать текстовые ответы на вопросы после обучения.
2. Модуль должен уметь пред обрабатывать входные текстовые наборы данных.
3. Модуль должен уметь осуществлять дообучение различных LLM.
4. Модуль должен уметь рассчитывать метрики из пункта 2.3 для больших языковых моделей.

Нефункциональные требования к программной реализации:

1. При обучении модуль должен изменять только параметры нейронных сетей реализующих функционал $M_{tunable}$ и D из пунктов 2.1–2.2.
2. Модуль должен иметь возможность замены отдельных компонентов реализующих $M_{tunable}$, G и D из пунктов 2.1–2.2 соответственно.

3.3 Проектирование программного модуля

Реализация программного модуля, осуществляющего метод дообучения на основе генеративно-состязательных сетей, должна обладать четким планом для успешной и удобной разработки. Основные компоненты разрабатываемого модуля включают следующие элементы:

1. **Компонент, реализующий модель** $M_{tunable}$. Данный компонент должен осуществлять функциональные возможности для $M_{tunable}$ и его взаимодействие со слоем эмбеддинга модели $M_{pretrain}$ из пункта 2.1. На вход данный компонент должен принимать выход слоя эмбеддинга и случайный вектор. На выходе данный компонент должен содержать вектор с размерностью выхода слоя эмбеддинга.
2. **Компонент, реализующий дискриминатор** D. Данный компонент должен классифицировать входные данные на реальные и сгенерированные. На вход компонент принимает токенизированную последовательность. На выходе компонент должен иметь вероятность реальности входной последовательности.
3. **Компонент, реализующий архитектуру GAN**. Данный компонент должен осуществлять функциональные возможности для генератора G, а также хранить внутри себя 1 и 2 компонент. Кроме того, данный компонент должен осуществлять взаимодействие G и D, включая в том числе метод обучения на основе подхода SeqGAN. При генерации компонент принимает на вход токенизированную последовательность и имеет на выходе ответ в виде другой токеннизированной последовательности. При обучении компонент должен принимать обучающую выборку данных в виде вопросов и эталонных ответов в текстовом формате.

3.4 Программная реализация модуля дообучения больших языковых моделей

При программной реализации метода дообучения использовались объектно-ориентированные возможности языка Python в виде использования классов для разработки компонентов из пункта 3.3. Описание реализованных классов:

Класс EmbeddingGenerator. Данный класс реализует компонент 1 из пункта 3.3. Также этот класс является наследником класса torch.nn.Module для использования всех преимуществ работы с нейронными сетями библиотеки torch.

Атрибуты класса EmbeddingGenerator:

- $device$ (torch.device): Информация о процессоре, используемым при прямом распространении.
- $embedding\_layer$ (torch.nn.Module): слой эмбеддинга от $M_{pretrain}$.
- $embedding\_layer\_type$ (torch.dtype): информация о типе данных весов для $embedding\_layer$.
- $layer1$ (torch.nn.Sequential): первый слой из архитектуры $M_{tunable}$. Функцией активации является $LeakyReLU$.
- $layer2$ (torch.nn.Sequential): второй слой из архитектуры $M_{tunable}$. Функцией активации является $LeakyReLU$.
- $layer3$ (torch.nn.Sequential): третий слой из архитектуры $M_{tunable}$. Функцией активации является $LeakyReLU$.
- $layer4$ (torch.nn.Linear): четвертый слой из архитектуры $M_{tunable}$. Функцией активации является $Linear$.

Методы класса EmbeddingGenerator:

- $forward$ – вычисляет результат работы $M_{tunable}$. Выполняет следующие действия:
  1. Распространяет входную последовательность через $embedding\_layer$.
  2. Вычисляет случайный вектор размерностью 100 из нормального распространения.
  3. Последовательно распространяет результат 1-го действия вместе с результатом 2-го действия через $layer1, \ldots, layer4$.
  4. Возвращает сумму результата первого и третьего действия.
- $generator\_parameters$ – возвращает параметры $layer1, \ldots, layer4$.

Класс Discriminator. Данный класс реализует компонент 2 из пункта 3.3. Данный класс также является наследником класса torch.nn.Module.

Атрибуты класса Discriminator:

- $layer1$ (torch.nn.Sequential): первый слой из архитектуры дискриминатора D. Функцией активации является $LeakyReLU$.
- $layer2$ (torch.nn.Sequential): второй слой из архитектуры дискриминатора D. Функцией активации является $LeakyReLU$.
- $layer3$ (torch.nn.Sequential): третий слой из архитектуры дискриминатора D. Функцией активации является $LeakyReLU$.
- $layer4$ (torch.nn. Sequential): четвертый слой из архитектуры дискриминатора D. Функцией активации является $Sigmoid$.

Методы класса Discriminator:

- $forward$ – вычисляет вероятность реальности входных данных. Для этого последовательно распространяет входные данные через $layer1, \ldots, layer4$.

Класс LanguageGAN. Данный класс реализует компонент 3 из пункта 3.3. Данный класс также является наследником класса torch.nn.Module.

Атрибуты класса LanguageGAN:

- $device$ (torch.device): Информация о процессоре, используемым при прямом распространении и обучении.
- max_length_input (int): длина входной токенизированной последовательности.
- max_new_tokens (int): длина выходной токенизированной последовательности.
- $embedding\_generator\_layer$ (torch.nn.Module): компонент EmbeddingGenerator.
- $discriminator$ (torch.nn.Module): компонент Discriminator.
- $model$ (torch.nn.Module): компонент реализующий генератор G.
- $g\_optimizer$ (torch.optim.Adam): модель, осуществляющая градиентный спуск по методу Adam для $embedding\_generator\_layer$.
- $d\_optimizer$ (torch.optim.Adam): модель, осуществляющая градиентный спуск по методу Adam для $discriminator$.

Методы класса LanguageGAN:

- $forward$ – вычисляет вероятности для токенов из словаря $M_{pretrain}$.
- $generate$ – генерируют ответ в виде токенизированной последовательности для вопроса из входных данных.
- $generator\_step$ – вычисляет градиент стратегии по методу Монте-Карло и делает подстройку для параметров $M_{tunable}$ - $embedding\_generator\_layer$.
- $discriminator\_step$ – вычисляет функции потерь и делает подстройку для параметров

D – $discriminator$

– $train$ – осуществляет обучение по эпохам по разработанному в пункте 2.2 методу. Выполняет следующие действия:

1. Пред обрабатывает входные данные, производя их токенизацию и приведение к одному размеру.
2. Делить пред обработанные данные на батчи.
3. Осуществляет обучение по эпохам используя последовательно $discriminator\_step$ и $generator\_step$

Разработанный модуль также содержит следующие функции:

– $tokenize\_func\_map$ и $tokenize\_func$ - осуществляют пред обработку данных.

– $calculate\_metric$ – вычисляет метрики ROUGE и BLEURT из пункта 2.3.

– $visualize\_weights$ – строит графики для тестирования поведения весов G.

– $test\_question\_answering$ – проверяет возможность модели G генерировать ответы.

– $get\_memory\_history$ – создает файл формата picle для сохранения истории использования GPU.

3.5 Тестирование разработанных программных модулей

Важным этапом разработки является тестирование программного кода. Для проверки корректности работы разработанного программного модуля было проведено тестирование выполнения требований из пункта 3.2.

Для проверки трех первых функциональных требований была проверена возможность отвечать уже обученной модели G – $LanguageGAN$ на вопрос, имеющий текстовый тип данных. Ответ модели представлен на Рисунке 3.1.

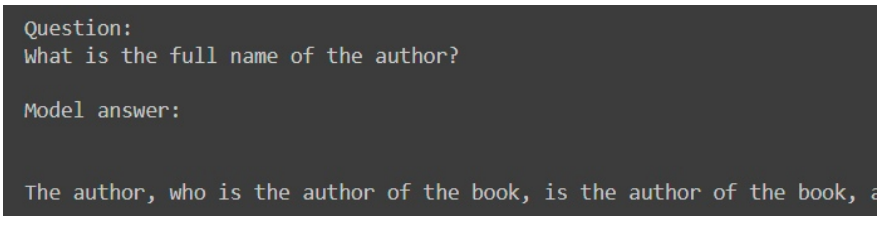

Рисунок 3.1 – Результат первого теста

Для проверки четвертого функционального требования были вычислены метрики из пункта 2.3 для пред обученной языковой модели GPT-2. Результат тестирование представлен на Рисунке 3.2.

Для проверки нефункциональных требований был построен график изменения весов для $EmbeddingGenerator$ и отдельный график для остальных весов $LanguageGAN$, соответ-

Рисунок 3.2 – Результат тестирования способности вычисления метрик

ствующих генератору G. Первый график представлен на Рисунке 3.3, второй график представлен на Рисунке 3.4.

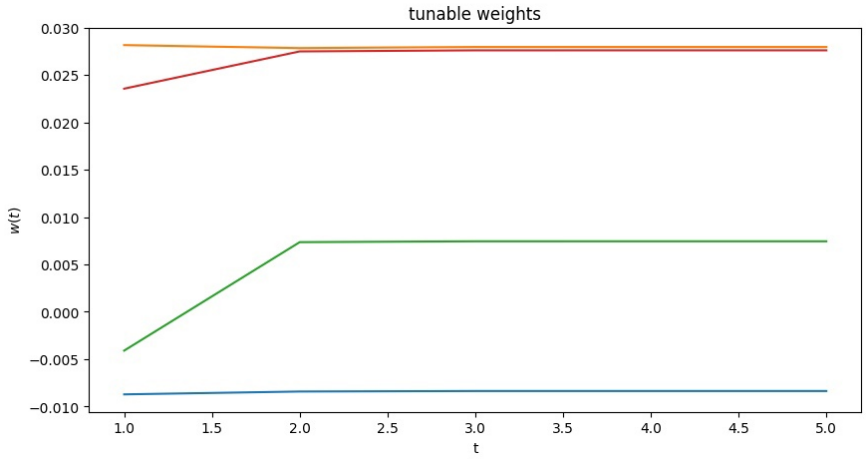

Рисунок 3.3 – График зависимости весов модели $EmbeddingGenerator$ от эпохи обучения

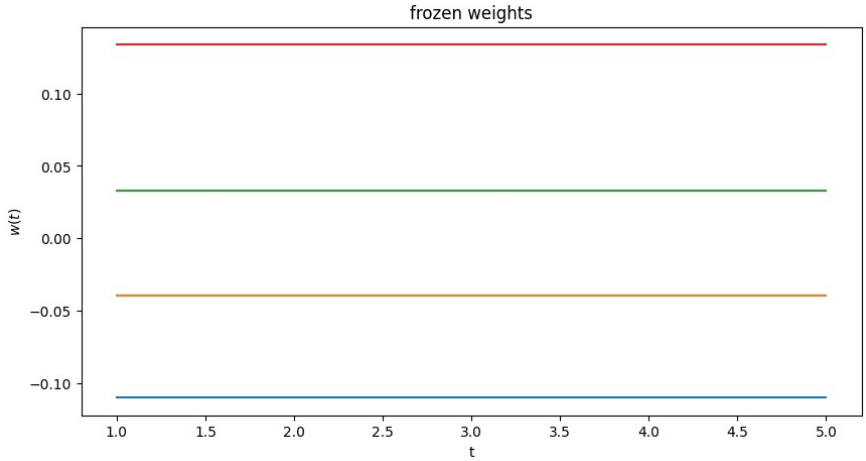

Рисунок 3.4 – График зависимости остальных весов $LanguageGAN$ от эпохи обучения

Также было проведено нагрузочное тестирование для разработанного метода обучения LLM. Для проведения нагрузочного тестирование использовались встроенные инструменты PyTorch. Результаты тестирования представлены в виде диаграммы на Рисунке 3.5. Полученная диаграмма соответствует зависимости использованного объема оперативной памяти графического процессора от выполнения метода обучения при следующих параметрах обучения:

– Эпох обучения: 1
– Длина входной токенезированной последовательности: 100
– Количество генерируемых генератором токенов: 50
– Объем обучающей выборки: 50
– Размер батча: 1

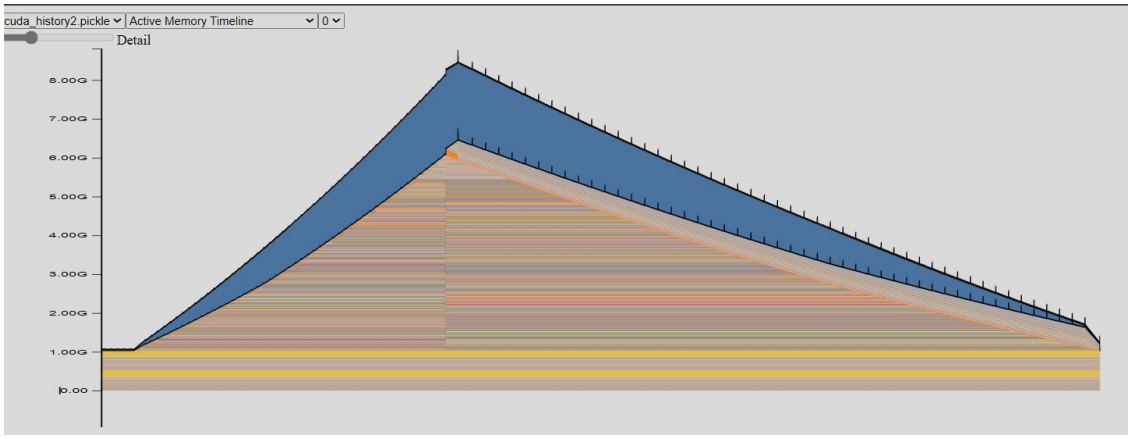

Рисунок 3.5 – Диаграмма изменения использованного объема оперативной памяти графического процессора при одной эпохе обучения

Таким образом, после проведения тестирования установлено, что разработанный программный модуль выполняет все предъявляемые требования.

3.6 Выводы

В ходе программной реализации были получены следующие выводы:

1. Выбран язык и модули для программной реализации: Python, Transformers, PyTorch
2. Разработаны требования к программной реализации
3. Выполнена программная реализация на языке Python
4. Проведено тестирование разработанного программного модуля. В результате тестирования установлено, что разработанный программный модуль соответствует всем предъявленным требованиям

4. Экспериментальная проверка

В данной главе описаны экспериментальные исследования разработанного метода дообучения, а также эксперементальное сравнение с другими методами дообучения.

4.1 Выбор набора данных для исследований

Для проведения экспериментальной проверки разработанного метода необходимо выбрать набор данных для обучения и расчета значений метрик, а также выбрать пред обученную языковую модель, на которой будет производиться проверка метода дообучения.

В качестве набора данных был выбран TOFU, описанный в статье [21]. Данный набор данных состоит из пар вопросов и ответов, основанных на автобиографиях 200 различных авторов, которые не существуют и полностью вымышлены с помощью большой языковой модели GPT-4. Этот набор данных был выбран за подходящий, удобный формат хранения данных, а также за то, что существующие пред обученные языковые модели не имели его в своих обучающих выборках.

В качестве обучаемой языковой модели была выбрана популярная модель GPT-2, представленная в статье [22]. Данная модель представляет из себя нейронную сеть основанную на компоненте декодера из архитектуры трансформера. Несмотря на относительно небольшой размер, GPT-2 обладает значительной эффективностью.

4.2 Экспериментальные исследования разработанного метода

Для экспериментальной проверки разработанного модуля набор данных TOFU был сокращен до 100 пар, из которых одна половина используется в обучении, а другая для вычисления метрик из пункта 3.2.

При обучении были установлены следующие параметры обучения:

- объем обучающей выборки: 50
- длина входной токенизированной последовательности: 100
- длина сгенерированной токенизированной последовательности: 50
- количество эпох обучения: 30
- размер батча: 1

В Таблице 4.1 приведены значения метрик для базовой модели GPT-2 и обученной модели GPT-2. Проведенная экспериментальная проверка показала, что обученная модель лучше отвечает на вопросы, то есть разработанный метод обучения работает успешно.

Таблица 4.1 – Значения метрик, полученных в ходе эксперимента

Model	ROUGE-1	ROUGE-2	ROUGE-L	ROUGE-Lsum	BLEURT
base model	0.066	0.02	0.062	0.061	-0.936
tuned model	0.109	0.036	0.099	0.101	-0.611

4.3 Экспериментальное сравнение предлагаемого метода с существующими методами

Для экспериментального сравнения разработанного метода был выбран метод адаптации низкого ранга (LoRA). Этот метод демонстрирует высокую эффективность обучения при низкой вычислительной сложности.

При обучении модели GPT-2 по методу LoRA использовалась реализация данного метода из библиотеки transformers. Параметры обучения по возможности совпадают с параметрами обучения из пункта 4.2, обучающие наборы данных полностью совпадают.

В Таблице 4.2 приведены значения метрик для модели GPT-2 обученной разработанным методом и методом LoRA. Значения метрик для модели, обученной методом LoRA, немного выше значений метрик для модели, обученной разработанным методом. Кроме того, из Таблицы 4.3 можно сдлеать вывод, что модель, обученная методом LoRA, требует значительно меньше времени на обучение и меньший объём используемой оперативной памяти. Данные о времени обучения и используемом объёме оперативной памяти получены с помощью функционала программной среды Google Colab.

Таблица 4.2 – Значения метрик, полученных в ходе экспериментального сравнения

Model	ROUGE-1	ROUGE-2	ROUGE-L	ROUGE-Lsum	BLEURT
GAN tuning	0.109	0.036	0.099	0.101	-0.611
LoRA tuning	0.127	0.027	0.111	0.112	-0.536

Таблица 4.3 – Вычислительные затраты обучения моделей

Model	time, min	memory usage, GB
GAN tuning	91	12.3
LoRA tuning	2	1.1

Таким образом разработанный метод обучения в текущей версии имеет меньшую эффективность по сравнению с методом адаптации низкого ранга.

Поскольку проводилось сравнение только с одним методом дообучения на небольшом наборе данных при небольшом количестве эпох, то для полной сравнительной оценки разработанного метода необходимо произвести дополнительные экспериментальные сравнения.

4.4 Выводы

В ходе проведения экспериментов с разработанным методом дообучения были получены следующие выводы:

1. Разработанный программный модуль работоспособен
2. Разработанный метод обучения сильно уступает в вычислительных затратах методу адаптации низкого ранга
3. Для полной сравнительной оценки разработанного метода необходимы дополнительные экспериментальные сравнения

Заключение

Большие языковые модели приобретают всю большую популярность, вследствие чего многие компании пытаются внедрить языковые модели самостоятельно. В этих условиях задача разработки эффективного метода дообучения языковых моделей становится крайне важной. В данной работе был разработан метод дообучения больших языковых моделей с его реализацией в виде программного модуля.

Основной целью данной работы была разработка метода дообучения основанного на подходе prompt tuning с использованием генеративных состязательных сетей. Основная цель вместе со всеми под задачами была выполнена. Кроме того, проведено тестирование выполнения всех функциональных и нефункциональных требований предъявляемых к программному модулю.

В ходе экспериментальной проверки метода дообучения были получены хорошие значения метрик. При экспериментальном сравнении разработанный метод показал примерно равные значения метрик с методом низкой адаптации ранга, но значительно уступал в показателях, связанных с оценкой вычислительных затрат. Кроме того, был получен вывод, что необходимо произвести дополнительные экспериментальные сравнения с другими методами дообучения.

В ходе проведенного исследования было установлено, что возможно использование архитектуры GAN в качестве метода дообучения больших языковых моделей. Эти результаты подчеркивают важность дальнейших исследований в этой области, так как они могут привести к созданию более эффективных и менее затратных методов дообучения. В будущем планируется продолжить работу над улучшением предложенного метода, а также провести более детальное сравнение с альтернативными подходами, чтобы обеспечить наилучшие результаты в применении больших языковых моделей в различных областях.

Список литературы

1. Large language models in medicine / A. J. Thirunavukarasu [и др.] // Nature medicine. — 2023. — Т. 29, № 8. — С. 1930—1940.
2. *Брагин А.В. Б. А.* Большие языковые модели четвёртого поколения как новый инструмент в научной работе // Искусственные общества. — 2023. — Т. 18, № 1. — DOI: 10.18254/S207751800025046-9.
3. Programming Is Hard – Or at Least It Used to Be: Educational Opportunities And Challenges of AI Code Generation [Электронный ресурс] / B. A. Becker [и др.]. — 2022. — arXiv: 2212.01020 [cs.HC]. — URL: https://arxiv.org/abs/2212.01020.
4. *Korinek A.* Language Models and Cognitive Automation for Economic Research : Working Paper / National Bureau of Economic Research. — 02.2023. — № 30957. — DOI: 10.3386/w30957. — URL: http://www.nber.org/papers/w30957.
5. A survey on Large Language Model (LLM) security and privacy: The Good, The Bad, and The Ugly / Y. Yao [и др.] // High-Confidence Computing. — 2024. — Март. — С. 100211. — ISSN 2667-2952. — DOI: 10.1016/j.hcc.2024.100211. — URL: http://dx.doi.org/10.1016/j.hcc.2024.100211.
6. When Scaling Meets LLM Finetuning: The Effect of Data, Model and Finetuning Method [Электронный ресурс] / B. Zhang [и др.]. — 2024. — arXiv: 2402.17193 [cs.CL]. — URL: https://arxiv.org/abs/2402.17193.
7. LoRA: Low-Rank Adaptation of Large Language Models [Электронный ресурс] / E. J. Hu [и др.]. — 2021. — arXiv: 2106.09685 [cs.CL]. — URL: https://arxiv.org/abs/2106.09685.
8. QLoRA: Efficient Finetuning of Quantized LLMs / T. Dettmers [и др.] // Advances in Neural Information Processing Systems. Т. 36 / под ред. A. Oh [и др.]. — Curran Associates, Inc., 2023. — С. 10088—10115.
9. *Li X. L.*, *Liang P.* Prefix-Tuning: Optimizing Continuous Prompts for Generation // Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers) /

под ред. C. Zong [и др.]. — Online : Association for Computational Linguistics, 08.2021. — С. 4582—4597. — DOI: 10.18653/v1/2021.acl-long.353.

10. Parameter-Efficient Transfer Learning for NLP [Электронный ресурс] / N. Houlsby [и др.]. — 2019. — arXiv: 1902.00751 [cs.LG]. — URL: https://arxiv.org/abs/1902.00751.

11. Self-Consistency Improves Chain of Thought Reasoning in Language Models [Электронный ресурс] / X. Wang [и др.]. — 2023. — arXiv: 2203.11171 [cs.CL]. — URL: https://arxiv.org/abs/2203.11171.

12. P-Tuning: Prompt Tuning Can Be Comparable to Fine-tuning Across Scales and Tasks / X. Liu [и др.] // Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers) / под ред. S. Muresan, P. Nakov, A. Villavicencio. — Dublin, Ireland : Association for Computational Linguistics, 05.2022. — С. 61—68. — DOI: 10.18653/v1/2022.acl-short.8. — URL: https://aclanthology.org/2022.acl-short.8.

13. Retrieval-augmented generation for knowledge-intensive NLP tasks / P. Lewis [и др.] // Proceedings of the 34th International Conference on Neural Information Processing Systems. — Vancouver, BC, Canada : Curran Associates Inc., 2020. — (NIPS'20). — ISBN 9781713829546.

14. Fine-tuning Language Models with Generative Adversarial Reward Modelling [Электронный ресурс] / Z. Z. Yu [и др.]. — 2024. — arXiv: 2305.06176 [cs.CL]. — URL: https://arxiv.org/abs/2305.06176.

15. Generative adversarial networks: introduction and outlook / K. Wang [и др.] // IEEE/CAA Journal of Automatica Sinica. — 2017. — Т. 4, № 4. — С. 588—598. — DOI: 10.1109/JAS.2017.7510583.

16. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient [Электронный ресурс] / L. Yu [и др.]. — 2017. — arXiv: 1609.05473 [cs.LG]. — URL: https://arxiv.org/abs/1609.05473.

17. *Sutton R. S.*, *Barto A. G.* Reinforcement Learning: An Introduction. — Second. — The MIT Press, 2018. — URL: http://incompleteideas.net/book/the-book-2nd.html.

18. *Lin C.-Y.* Rouge: A package for automatic evaluation of summaries // Text summarization branches out. — 2004. — С. 74—81.

19. *Sellam T.*, *Das D.*, *Parikh A. P.* BLEURT: Learning Robust Metrics for Text Generation [Электронный ресурс]. — 2020. — arXiv: 2004.04696 [cs.CL]. — URL: https://arxiv.org/abs/2004.04696.

20. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding [Электронный ресурс] / J. Devlin [и др.]. — 2019. — arXiv: 1810.04805 [cs.CL]. — URL: https://arxiv.org/abs/1810.04805.

21. TOFU: A Task of Fictitious Unlearning for LLMs [Электронный ресурс] / P. Maini [и др.]. — 2024. — arXiv: 2401.06121 [cs.LG]. — URL: https://arxiv.org/abs/2401.06121.

22. Language Models are Unsupervised Multitask Learners / A. Radford [и др.] //. — 2019. — URL: https://api.semanticscholar.org/CorpusID:160025533.

Приложение

Приложение содержит код основных компонентов разработанного программного модуля. С полной версией кода можно ознакомиться по ссылке: [Программная реализация]
Код реализующей инициализацию и сохранение атрибутов класса $LanguageGAN$

Листинг 4.1 – LanguageGAN

```
class LanguageGAN(nn.Module):
    def __init__(self, model, embedding_layer,
       func_set_embedding_layer, max_length_input=100,
       max_new_tokens=50, device=torch.device('cpu')):

   super(LanguageGAN, self).__init__()

        self.embedding_generator_layer = EmbeddingGenerator(
    embedding_layer,
    device).to(device)
        func_set_embedding_layer(model, self.embedding_generator_layer)

        self.discriminator = Discriminator(
    max_length_input + max_new_tokens).to(device)
        self.model = model

        self.max_length_input = max_length_input
        self.max_new_tokens = max_new_tokens
        self.device = device

        self.g_optimizer = None
        self.d_optimizer = None
```

Метод $generator\_step$ класса $LanguageGAN$. Данный метод осуществляет шаг настройки весов $self.embedding\_generator\_layer$, вычисляя градиент стратегии методом Монте-Карло:

Листинг 4.2 – метод $generator\_step$

```
def generator_step(self, input_ids, attention_mask, generation_config):
  self.embedding_generator_layer.eval()
  self.discriminator.eval()

  # вычисление выходной последовательности и награды для градиента стратегии
  with torch.no_grad():
    trajectory = self.generate(input_ids = input_ids,
              attention_mask = attention_mask,
              generation_config = generation_config)
    rewards = self.discriminator(trajectory.type(torch.float32)).view(1, -1)
```

```
  self.embedding_generator_layer.train()
  log_probs = []

  # сохранение логарифма вероятности для каждого
  # токена из выходной последовательности
  state_attention_mask = attention_mask.clone()
  batch_size = trajectory.shape[0]
  one_tensor = torch.IntTensor([1] * batch_size).view(-1, 1).to(self.device)
  for i in range(self.max_new_tokens):
    state = trajectory[:, :self.max_length_input + i]
    output = self(input_ids=state, attention_mask=state_attention_mask)[0]

    probs = nn.functional.softmax(output.sum(dim=1), dim=1)
    m = torch.distributions.categorical.Categorical(probs)
    action = m.sample()
    log_prob = m.log_prob(action)
    log_probs.append(log_prob.view(-1, 1))

    state_attention_mask = torch.concat(
      (state_attention_mask, one_tensor), dim=1)
  log_probs = torch.concat(log_probs, dim=1).sum(dim=1)

  # вычисление градиента стратегии методом МонтеКарло-
  policy_loss = []
  for log_prob, reward in zip(log_probs, rewards):
    policy_loss.append(-log_prob * reward)
  policy_loss = torch.concat(policy_loss).sum() / batch_size

  # осуществление градиентного спуска для весов M_tunable
  self.g_optimizer.zero_grad()
  policy_loss.backward()
  self.g_optimizer.step()

  return policy_loss
```

Метод *discriminator_step* класса *LanguageGAN*. Данный метод осуществляет шаг настройки весов $self.discriminator$, следуя классической схеме обучения дискриминатора генеративной состязательной сети:

Листинг 4.3 – метод *discriminator_step*

```
def discriminator_step(self, input_ids, attention_mask,
          answer_input_ids, generation_config, batch_size):
  self.embedding_generator_layer.eval()
  self.discriminator.train()

  # генерация ненастоящих данных
  generator_output = self.generate(input_ids = input_ids,
              attention_mask = attention_mask,
              generation_config = generation_config)

  # вычисление вероятностей реальности для настойщих и сгенерированных данных
  fake_output = self.discriminator(generator_output.type(torch.float32))
  real_output = self.discriminator(answer_input_ids.type(torch.float32))
```

```
# вычисление функции потерь для реальных и сгенерированных данных
loss = nn.BCELoss()
loss_fake = loss(fake_output,
       torch.zeros((batch_size, 1), device = self.device))
loss_real = loss(real_output,
       torch.ones((batch_size, 1), device = self.device))
d_loss = loss_real + loss_fake

# осуществление градиентного спуска для весов дискриминатора
self.d_optimizer.zero_grad()
d_loss.backward()
self.d_optimizer.step()

return d_loss
```