# Глава 0. Введение

I. Почему большие модели не являются «универсальным решением»?

Эпоха больших языковых моделей (LLM), таких как GPT-4, LLaMA и их аналоги, ознаменовала собой качественный скачок в области искусственного интеллекта. Эти модели, обученные на колоссальных массивах текстовых данных из интернета, демонстрируют феноменальные способности в генерации связных текстов, решении логических задач, программировании и поддержании диалога. Создается впечатление, что мы получили универсальный искусственный интеллект, готовый решать любые задачи «из коробки». Однако это впечатление обманчиво, и именно в его разоблачении заключается ключевая проблема современного практического применения LLM.

При всей своей мощи «сырая», неизмененная языковая модель подобна эрудированному выпускнику университета с обширными, но общими знаниями. Такой специалист блестяще рассуждает на философские темы или пересказывает общеизвестные факты, но окажется беспомощным, когда его попросят объяснить внутренние регламенты конкретной компании, проанализировать только что вышедший отчет о финансовых результатах или дать точный ответ на вопрос, основанный на последних изменениях в законодательстве. Его знания ограничены той информацией, которая была доступна на момент его «выпуска» – момента обучения модели.

На практике это выливается в ряд фундаментальных ограничений. Во-первых, это проблема актуальности. Модель не знает о событиях, произошедших после даты среза ее тренировочных данных. Во-вторых, это проблема специализации. Модель может хорошо понимать общую медицинскую терминологию, но ее ответы будут поверхностными и потенциально опасными при запросе о конкретном, редком диагнозе или новом методе лечения. В-третьих, это проблема контроля и достоверности. Модель может «галлюцинировать» – генерировать правдоподобную, но абсолютно вымышленную информацию, особенно в узких тематиках, где ее внутренние знания фрагментарны. Наконец, существует проблема конфиденциальности. Нельзя загрузить в публичную модель внутреннюю документацию предприятия, не рискуя ее раскрытием. Таким образом, разрыв между общими способностями модели и конкретными требованиями бизнеса или исследовательской задачи является основным вызовом, который необходимо преодолеть.

II. Проблема «замороженного знания» и необходимость адаптации.

Ключевая метафора, описывающая суть проблемы, — это «замороженное знание». Представьте себе гигантскую библиотеку, все книги в которой были отобраны и расставлены по полкам в один определенный день. После этого библиотека запечатана. Она огромна и универсальна, но в ней нет ни одной книги, изданной позже этой даты. Любая большая языковая модель — это и есть такая библиотека. Ее параметры (веса нейронной сети) представляют собой сжатую, статистическую репрезентацию всех тех данных, на которых она обучалась. Этот процесс обучения настолько ресурсоемок, что проводится лишь эпизодически, и модель на долгое время остается статичной, в то время как мир продолжает меняться.

Проблема замороженного знания проявляется в трех основных аспектах. Временной аспект делает модель неспособной оперировать свежими новостями, актуальными курсами валют или последними версиями программных продуктов. Предметный аспект не позволяет модели глубоко понимать специфическую терминологию, внутренние процессы и нюансы конкретной предметной области (домена), будь то юриспруденция, ядерная физика или логистика целой компании. Контекстуальный аспект лишает ее доступа к приватным и закрытым данным, которые необходимы для принятия решений в конкретной организации.

Следовательно, возникает насущная необходимость в адаптации этих универсальных моделей под конкретные, узкие задачи. Адаптация — это процесс «разморозки» знаний модели или, что чаще и эффективнее, надстройки над ее замороженным ядром новых, специфических умений и информации. Цель адаптации — превратить эрудированного, но оторванного от реальности выпускника в высококвалифицированного специалиста, вооруженного самой свежей и релевантной информацией для решения поставленной перед ним задачи.

III. Обзор методов: от быстрых и «беспараметрических» до глубокого дообучения.

На сегодняшний день сформировалось несколько четких и взаимодополняющих парадигм адаптации LLM, которые образуют своеобразный спектр — от методов, практически не меняющих саму модель, до методов ее глубокой перестройки.

На одном конце этого спектра находятся контекстные методы, ярчайшим представителем которых является Retrieval-Augmented Generation (RAG). Этот подход можно охарактеризовать как «беспараметрический», поскольку сама модель не подвергается дообучению, ее параметры остаются замороженными. Вместо этого, к модели «подключается» внешнее, обновляемое хранилище знаний — база документов, спецификаций или статей. Когда модель получает запрос, система сначала осуществляет семантический поиск по этой базе, находит наиболее релевантные фрагменты информации и затем, подавая их в качестве контекста в промпт, просит модель сформулировать ответ. RAG идеален для задач, где информация часто меняется и должна быть всегда актуальной, а также там, где критически важна достоверность и проверяемость ответов. Это быстрый в реализации и экономичный с вычислительной точки зрения подход.

На другом конце спектра находятся параметрические методы, которые предполагают изменение внутренних весов модели. Классическим и самым мощным из них является полное дообучение (Full Fine-Tuning). В этом случае модель «размораживается», и все ее миллиарды параметров настраиваются на небольшом, но репрезентативном наборе данных целевой предметной области. Это эквивалентно перепрофилированию универсального специалиста через интенсивные курсы, в результате которых он меняет сам стиль своего мышления и генерации текста. Такой метод может дать наилучшее качество и стилистическое соответствие, но он невероятно требователен к вычислительным ресурсам, объему качественных данных и чреват такими рисками, как «катастрофическое забывание» — когда модель, глубоко освоив новую область, забывает свои прежние общие знания.

Компромиссным и наиболее популярным на сегодня подходом являются эффективные методы параметрической настройки (Parameter-Efficient Fine-Tuning, PEFT), такие как LoRA (Low-Rank Adaptation) и ее более продвинутая версия QLoRA. Эти методы предлагают элегантное решение. Вместо изменения всех параметров модели, они добавляют к ее архитектуре небольшие, обучаемые «адаптеры». Дообучению подвергаются только эти адаптеры, составляющие доли процента от исходного объема модели, в то время как сама базовая модель остается замороженной. LoRA обеспечивает качество, близкое к полному дообучению, но с радикально меньшими затратами памяти и времени. QLoRA идет еще дальше, применяя методы квантования, что позволяет дообучить многомиллиардную модель даже на одном потребительском графическом ускорителе. Это сделало мощную адаптацию моделей доступной для широкого круга исследователей и инженеров.

Таким образом, современный специалист по машинному обучению должен владеть всем этим арсеналом методов, понимая их сильные и слабые стороны. Данная лекция ставит своей целью провести вас по всему этому пути — от понимания фундаментальных ограничений LLM через глубокое погружение в архитектуру RAG, тонкости полного дообучения к изяществу и мощи методов LoRA и QLoRA, и, наконец, к выработке стратегического подхода к выбору правильного инструмента для каждой конкретной задачи.

# Глава 1. Фундаментальные основы Больших Языковых Моделей

Современные LLM — не результат внезапного открытия, а продукт долгой эволюции в области обработки естественного языка (NLP). Понимание этой эволюции критически важно для осмысленной работы с моделью, так как объясняет, какие проблемы решает каждая следующая архитектура и какие ограничения остаются.

## §1. Эволюция NLP: от N-грамм до Transformer

Чтобы по-настоящему оценить мощь и революционность больших языковых моделей, необходимо совершить краткий экскурс в историю обработки естественного языка (Natural Language Processing, NLP). Эта эволюция представляет собой путь от простых статистических подсчетов к сложным нейросетевым архитектурам, способным улавливать тончайшие смысловые нюансы человеческой речи.

I. Эпоха статистики и символических методов

В начале были N-граммы. Этот подход, доминировавший долгие годы, был основан на простой, но эффективной идее: вероятность появления следующего слова в последовательности зависит от ограниченного количества предыдущих слов (чаще всего одного, двух или трех). Например, биграммная модель (N=2) оценивала фразу «красное яблоко» как произведение вероятностей P(яблоко | красное) \* P(красное | <начало предложения>). Эти вероятности вычислялись путем простого подсчета частот встречаемости слов в огромных текстовых корпусах.

N-граммы были прорывом для своего времени. Они легли в основу первых систем проверки орфографии, автодополнения ввода и простейших машинного перевода. Однако у них был фундаментальный недостаток — «проклятие размерности». Для создания полноценной триграммной модели для языка с словарем в 50,000 слов требовалось хранить и рассчитывать 50,000^3 (125 триллионов!) вероятностей. Это было вычислительно нереализуемо. Кроме того, N-граммы были слепы к контексту, выходящему за рамки окна в N слов, и не могли понять синтаксис или семантику. Они работали со словами как с дискретными символами, не имеющими внутренней связи друг с другом.

Параллельно развивались символические подходы, основанные на жестких лингвистических правилах и онтологиях (например, WordNet). Эти системы пытались формализовать язык через грамматики и словари связей между понятиями. Несмотря на высокую точность в узких доменах, они оказались чрезвычайно трудномасштабируемыми и хрупкими — любое исключение из правила требовало ручного вмешательства лингвиста.

II. Векторные представления и нейронные сети

Следующим крупным шагом стал переход от символических представлений к векторным (embeddings). Идея, популяризированная такими моделями, как Word2Vec и GloVe, заключалась в том, чтобы сопоставить каждому слову в словаре плотный вектор чисел (например, 300 измерений) в непрерывном пространстве. Гениальность этого подхода в том, что геометрическое положение вектора в этом пространстве начинает отражать семантику слова. Знаменитый пример: вектор('король') - вектор('мужчина') + вектор('женщина') ≈ вектор('королева').

Нейронные сети, в частности рекуррентные нейронные сети (RNN) и их более совершенные версии — LSTM (Long Short-Term Memory) и GRU (Gated Recurrent Units), — научились работать с этими векторными представлениями, обрабатывая последовательности слов одну за другой. Они обладали «памятью» о предыдущих словах, что позволяло им учитывать более широкий контекст и создавать более связные тексты. RNN-powered модели показали выдающиеся результаты в машинном переводе, суммировании текста и генерации.

Но и у RNN был свой Ахиллес пята — проблема исчезающих и взрывающихся градиентов. При обработке длинных последовательностей информация из начала предложения «забывалась» или искажалась. Механизмы внимания (attention), появившиеся как дополнение к архитектурам Encoder-Decoder в задачах перевода, стали первым лучом света. Они позволяли модели на этапе декодирования «оглядываться» на все слова исходного предложения сразу, а не только на последнее скрытое состояние, и «внимательнее» смотреть на наиболее релевантные слова. Однако это был лишь костыль для устаревшей архитектуры.

III. Transformer: архитектурная революция 2017 года

В 2017 году статья Google «Attention Is All You Need» совершила переворот. Авторы предложили полностью отказаться от рекуррентности и сверток в пользу исключительно механизма самовнимания (Self-Attention). Это была не эволюция, а революция.

Архитектура Transformer кардинально меняла парадигму. Вместо последовательной обработки слов, она анализировала все слова в предложении одновременно (параллельно). Механизм самовнимания вычислял для каждого слова в последовательности взвешенную сумму векторов всех остальных слов. Веса этих сумм (внимание) определялись тем, насколько каждое слово в контексте было релевантно для текущего. Например, при обработке слова «ест» в предложении «Кот ест рыбу», модель научится назначать высокий вес словам «Кот» (кто ест) и «рыбу» (что ест).

Этот подход имел колоссальные преимущества:

* Неограниченный контекст: В отличие от RNN, информация не деградировала с расстоянием. Первое слово в предложении могло напрямую влиять на последнее.
* Вычислительная эффективность: Параллельная обработка всей последовательности идеально ложилась на матричные вычисления GPU, что радикально ускоряло обучение.
* Интерпретируемость: Матрицы внимания (хотя бы отчасти) позволяли заглянуть «внутрь» модели и увидеть, какие слова она считает важными.

Transformer стал той фундаментальной архитектурой, на которой построена вся современная экосистема LLM. GPT (Generative Pre-trained Transformer) и BERT (Bidirectional Encoder Representations from Transformers) — это две основные ветви эволюции исходной архитектуры, предназначенные для разных задач: авторегрессионной генерации и двунаправленного понимания контекста соответственно.

Таким образом, путь от N-gram к Transformer — это путь от грубых статистических приближений к тонкому, контекстуально-зависимому моделированию языка. Transformer предоставил тот самый «вычислительный примитив», который, будучи масштабированным до невообразимых ранее размеров (миллиарды параметров, терабайты текстовых данных), и породила феномен, который мы сегодня называем Большими Языковыми Моделями. Это не просто «еще одна архитектура» — это качественный скачок, открывший новую эру в взаимодействии человека и машины.

## §2. Что такое «Большая Языковая Модель»? Параметры, данные, вычислительная сложность

Чтобы понять феномен больших языковых моделей, необходимо выйти за рамки поверхностного определения и рассмотреть три фундаментальных столпа, на которых они покоятся: параметры, данные и вычислительную сложность. Именно синергия этих трех элементов создает ту качественно новую способность к пониманию и генерации связного текста, которая отличает современные LLM от их предшественников.

I. Параметры: архитектурная основа интеллекта

**Параметры — это, по сути, "синапсы" искусственной нейронной сети. В контексте LLM это числовые значения в матрицах весов, которые настраиваются в процессе обучения. Каждый параметр — это крошечный элемент знания, фиксирующий, например, вероятность того, что за словом "кофе" следует слово "утром", или что понятие "столица" сильно связано с понятием "Франция". Однако магия заключается не в отдельных параметрах, а в их коллективном поведении.**

**Когда мы говорим "большая" модель, мы подразумеваем архитектуру, содержащую сотни миллионов, а чаще миллиарды этих самых параметров. Модель GPT-3, ставшая в свое время прорывом, содержит 175 миллиардов параметров. Такая астрономическая емкость позволяет модели выстраивать невероятно сложные и многоуровневые представления о языке. Она учится не просто статистике слов, а грамматике, синтаксису, стилю, логике, причинно-следственным связям и даже элементам здравого смысла. Чем больше параметров, тем более тонкие и абстрактные паттерны может уловить модель, тем точнее она может смоделировать распределение вероятностей в человеческом языке. Однако эти миллиарды параметров — не просто пассивное хранилище. Они организованы в сложнейшую иерархическую структуру трансформера, где каждый слой способен извлекать все более абстрактные признаки из входной последовательности, превращая тем самым поток слов в глубинное семантическое представление.**

****II. Данные: топливо для обучения****

Масштаб модели был бы бесполезен без соответствующего масштаба данных для ее обучения. Большая языковая модель — это, в первую очередь, продукт своего обучающего корпуса. Объемы данных, используемые для предобучения таких моделей, трудно осознать. Речь идет о терабайтах и петабайтах текстовой информации, собранной из разнообразнейших источников: веб-страниц (Common Crawl), энциклопедий (Wikipedia), книг, научных статей, форумов и новостных лент. Этот корпус должен быть настолько же огромным и разнообразным, насколько разнообразен человеческий язык сам по себе.

Цель предобучения — не заучить конкретные тексты, а вывести из этого океана данных фундаментальные принципы языка: его синтаксис, семантику и прагматику. Модель, обучаясь на таком корпусе, сталкивается с бесчисленными вариациями выражения одной и той же мысли, что позволяет ей обобщать и абстрагировать. Качество и разнообразие данных напрямую влияют на качество итоговой модели. "Мусор на входе — мусор на выходе" — это правило в мире LLM приобретает гиперболизированный смысл. Смещения, присутствующие в интернет-данных, неизбежно усваиваются моделью, что создает серьезные этические и практические вызовы. Таким образом, данные — это не только топливо, но и сырье, которое формирует саму личность и картину мира модели.

****III. Вычислительная сложность: цена интеллекта****

Третий китовый элемент — вычислительная сложность — является прямым следствием двух первых. Обучение модели с миллиардами параметров на терабайтах данных — это одна из самых ресурсоемких задач в истории вычислений. Сложность алгоритмов обучения, таких как Transformer, растет квадратично с длиной входной последовательности и линейно с количеством параметров и слоев. Это означает, что удвоение размера модели или контекстного окна приводит к многократному росту требуемых вычислений.

На практике это выливается в необходимость использования тысяч высокопроизводительных графических процессоров (GPU), таких как NVIDIA A100 или H100, которые работают недели и даже месяцы. Энергопотребление одного цикла обучения сравнима с энергопотреблением небольшого города. Этот аспект — "цена интеллекта" — имеет фундаментальные последствия. Он создает высокий барьер для входа, концентрируя разработку передовых моделей в руках нескольких технологических гигантов. Он также делает эксперименты чрезвычайно дорогими, что стимулировало развитие более эффективных методов, таких как рассматриваемые в этом курсе LoRA и QLoRA, которые позволяют адаптировать готовые гигантские модели без непомерных затрат.

****IV. Синтез: новая парадигма искусственного интеллекта****

Взаимосвязь параметров, данных и вычислений породила новую парадигму в ИИ, которую можно назвать "масштабированием любой ценой". Эмпирический закон, наблюдаемый в последние годы, гласит: качество модели предсказуемо и устойчиво растет с увеличением всех трех компонентов — размера модели, объема данных и вычислительного бюджета. Это открытие сместило фокус исследований с разработки принципиально новых алгоритмов на их эффективное масштабирование.

Таким образом, "большая" языковая модель — это не просто модель со многими параметрами. Это сложная система, возникающая на стыке трех масштабов: архитектурного (параметры), информационного (данные) и инфраструктурного (вычисления). Понимание природы этой триады является ключевым для осмысленной работы с LLM, будь то их адаптация под конкретные нужды, развертывание в production-среде или критическая оценка их возможностей и ограничений. Именно эти ограничения, проистекающие из самой природы "больших" моделей, и заставляют нас искать интеллектуальные и экономически оправданные методы их тонкой настройки, которым и посвящен данный курс.

## §3. Ключевые семейства моделей: GPT, BERT, T5, LLaMA и их эволюция

Современная революция в обработке естественного языка (Natural Language Processing, NLP) была бы невозможна без появления нескольких ключевых архитектурных семейств больших языковых моделей. Каждое из этих семейств не просто представляло собой очередное инженерное решение, а воплощало в себе фундаментально новый взгляд на то, как машины должны понимать и генерировать человеческий язык. Эволюция этих моделей — это история поиска компромисса между вычислительной эффективностью, качеством генерации и универсальностью, история, которая напрямую определяет, как мы подходим к их адаптации сегодня.

Историю современных LLM принято отсчитывать с момента публикации архитектуры Transformer в 2017 году. Однако именно появление BERT (Bidirectional Encoder Representations from Transformers) от Google в 2018 году стало первым настоящим прорывом. BERT радикально отличался от всех предшественников своим подходом к «пониманию» текста. В отличие от односторонних моделей, читающих текст только слева направо или справа налево, BERT использовал механизм двунаправленного кодирования. Его предобучение было построено на задании маскирования языкового моделирования (Masked Language Modeling, MLM), где модель должна была предсказать случайно замаскированные слова в предложении, анализируя контекст как слева, так и справа от них. Это позволяло модели строить гораздо более глубокие контекстуальные представления каждого слова. BERT не был генеративной моделью в прямом смысле; его главной силой стало извлечение смысла из текста для решения таких прикладных задач, как классификация, извлечение именованных сущностей или определение тональности. Появление BERT ознаменовало эру «энкодерных» моделей, доминировавших в NLP на протяжении нескольких лет и установивших новые стандарты качества для широкого спектра бизнес-приложений.

Параллельно с этим развивалась иная парадигма, основанная на авторегрессионной генерации. Её наиболее ярким воплощением стала серия моделей GPT (Generative Pre-trained Transformer) от OpenAI. Если BERT был «экспертом по анализу», то GPT позиционировал себя как «универсальный генератор». Первая GPT, представленная в 2018 году, использовала только декодерную часть Transformer и обучалась на классической задаче предсказания следующего слова (Causal Language Modeling). Это казалось шагом назад по сравнению с двунаправленностью BERT, но именно эта простота открыла путь к невероятной генеративной гибкости. Модель училась порождать связный текст, просто предсказывая очередное слово на основе всех предыдущих.

Истинный масштаб этого подхода раскрылся с появлением GPT-2 в 2019, а затем и GPT-3 в 2020 году. OpenAI продемонстрировала, что простое масштабирование — увеличение объема данных, параметров модели и вычислительных мощностей — превращает такую, казалось бы, примитивную задачу в мощнейший инструмент. GPT-3 с её 175 миллиардами параметров показала феноменальную способность к «контекстному обучению» (few-shot learning), когда для решения новой задачи модели достаточно просто предоставить несколько примеров прямо в промпте, без какого-либо дообучения. Это установило новый тренд на создание моделей-универсалов, способных решать тысячи задач «из коробки». Эстафету подхватила GPT-4, которая не только еще больше увеличила масштаб, но и стала мультимодальной, интегрировав обработку визуальной информации, и значительно улучшила качество, надежность и способность к сложным рассуждениям.

Однако доминирование закрытых, коммерческих моделей вроде GPT стимулировало спрос на открытые и прозрачные альтернативы. Ответом на этот вызов стало семейство T5 (Text-To-Text Transfer Transformer) от Google. Представленная в 2020 году, модель T5 предложила элегантную унификацию: любая задача NLP формулируется как задача «текст-в-текст». Классификация, перевод, суммаризация, регрессия — всё преобразовывалось в формат, где на вход модели подается текст, а на выходе ожидается текст. Например, для классификации тональности входом могла быть фраза «оценить настроение: Фильм был потрясающий!», а ожидаемым выходом — слово «позитивный». Такой подход значительно упрощал процесс тонкой настройки и управления моделями, делая T5 чрезвычайно популярной в академической среде и для прикладных исследований, где важны контроль и воспроизводимость.

Наконец, одним из самых значимых событий последних лет стал выход в свет семейства LLaMA (Large Language Model Meta AI) от Meta. Анонсированная в 2023 году, LLaMA не предлагала революционно новой архитектуры. Её гениальность заключалась в другом: она демонстрировала, что можно достичь state-of-the-art результатов, используя меньшее количество параметров, но обучаясь на значительно большем объеме качественных данных. LLaMA-13B по своим способностям конкурировала с GPT-3 (175B), будучи при этом на порядок эффективнее в инференсе. Этот акцент на качестве данных и эффективности, а также политика открытости (исходный код и веса моделей стали доступны исследовательскому сообществу) вызвали настоящий взрыв инноваций. LLaMA стала фундаментом, на котором было построено бесчисленное количество производных моделей и методов адаптации, включая Alpaca, Vicuna и, что особенно важно, именно на её архитектуре были отработаны и продемонстрированы методы эффективного дообучения, такие как LoRA и QLoRA.

Эволюция этих семейств — от специализированного BERT к универсальным генераторам GPT, от унифицированного подхода T5 к эффективной и открытой LLaMA — не была линейной. Это ветвящееся дерево исследовательских направлений, где каждое ответвилось в поисках своего оптимального баланса. BERT доказал силу глубокого контекстуального энкодера, GPT — мощь масштабирования чистого генеративного декодера, T5 — элегантность унифицированного подхода, а LLaMA — невероятную важность курирования данных и эффективности архитектуры. Понимание их сильных и слабых сторон, изначального предназначения и эволюционного пути является критически важным первым шагом для осмысленного выбора методов их адаптации, ведь не существует универсального рецепта, подходящего одновременно для тонкой настройки BERT для классификации юридических документов и для дообучения LLaMA на креативное написание стихов.

## §4. Трансформеры

Трансформеры представляют собой одно из наиболее важных достижений в области глубокого обучения. Они основаны на концепции обработки, которая называется «внимание» (attention). Внимание позволяет сети присваивать разные веса разным входным данным, причем весовые коэффициенты также зависят от входных данных. Это дает возможность с высокой эффективностью анализировать последовательные и другие данные.

Эти модели известны как трансформеры, поскольку они преобразуют набор векторов в некотором векторном пространстве в набор векторов той же размерности в некотором новом пространстве. Цель трансформации состоит в том, чтобы новое векторное пространство имело более богатое внутреннее представление, которое лучше подходит для решения последующих задач. Входными данными трансформера могут быть неструктурированные наборы векторов, упорядоченные последовательности или более общие представлений, что обеспечивает широкую применимость трансформеров.

Трансформеры были изначально разработаны для задач обработки естественного языка (NLP, естественный язык – например, английский или китайский) значительно превзошли в данной сфере предыдущие подходы, основанные на рекуррентных нейросетях (RNN). Впоследствии было обнаружено, что трансформеры достигают отличные результаты во многих других предметных областях. Например, трансформеры часто превосходят сверточные нейросети (CNN), а мультимодальные преобразователи, которые объединяют несколько типов данных, таких как текст, изображения, аудио и видео, являются одними из самых мощных моделей глубокого обучения.

Одним из основных преимуществ трансформеров является то, алгоритм их обучения имеет очень высокую эффективность, поэтому трансформер можно обучить на больших объемах данных, а обученная модель может быть применена к широкому спектру задач с использованием той или иной формы тонкой настройки. Крупномасштабная модель, которую впоследствии можно адаптировать для решения множества различных задач известна как модель-фундамент (foundation model). Кроме того, возможно обучение трансформеров без учителя, на неразмеченных данных, что особенно эффективно для языковых моделей, поскольку преобразователи могут использовать огромные объемы текста, доступные в интернете и других источниках. Гипотеза масштабирования утверждает, что, просто увеличивая масштаб модели, измеряемый количеством ее параметров, и обучая модель на соразмерно большом наборе данных, дает возможность достичь значительного повышения эффективности модели даже без каких-либо архитектурных изменений. Кроме того, трансформер очень эффективно распараллеливается, что позволяет создавать исключительно большие языковые модели, имеющие порядка триллиона () параметров, которые возможно обучить за разумное время. Такие модели обладают исключительными возможностями и демонстрируют имеют свойства, которые были описаны как ранние признаки сильного искусственного интеллекта (Bubeck et al., 2023).

Архитектура трансформера может показаться новичку сложной или даже устрашающей, поскольку она включает в себя несколько совместно работающих компонентов, принцип работы которых может показаться произвольно выбранным. Поэтому в этой главе мы стремимся дать подробное пошаговое введение во все ключевые идеи, лежащие в основе трансформеров, и обеспечить четкое понимание архитектуры различных элементов. Сначала мы опишем архитектуру трансформера, а затем сосредоточимся на обработке естественного языка, прежде чем изучать другие области применения.

I. Внимание

Внимание – это фундаментальная концепция, лежащая в основе трансформера. Изначально оно было разработано как усовершенствование рекуррентных нейросетей (RNN) для машинного перевода (Bahdanau, Cho, and Bengio, 2014). Однако Васвани и др. (Vaswani et al., 2017) позже показали, что значительного улучшения производительности можно добиться, убрав рекуррентную структуру и вместо этого сосредоточившись исключительно на механизме внимания. Сегодня трансформеры, основанные на внимании, вытеснили RNN почти во всех сферах применения.

Далее рассмотрим механизм внимания на примере обработки текстов, хотя его применимость на этом не ограничивается. Рассмотрим два предложения:

I swam across the river to get to the other bank.

I walked across the road to get cash from the bank.

В каждом из этих предложений слово «bank» имеет разное значение (в предложении 1 – «берег», в предложении 2 – «банк»). При этом, это можно понять, только посмотрев на другие слова предложения. Также видно, что некоторые слова важнее других для определения значения слова «bank». В первом предложении слова «swam» («плыл») и «river» («река») наиболее сильно указывают на то, что слово «bank» означает «берег», а во втором предложении слово «cash» (деньги) является наиболее сильным индикатором, что слово «bank» означает «банк». Таким образом, для того чтобы определить значение слова «bank», нейросеть должна «обратить внимание» на несколько других конкретных слов предложения. Эта концепция внимания изображена на рисунке 4.1.

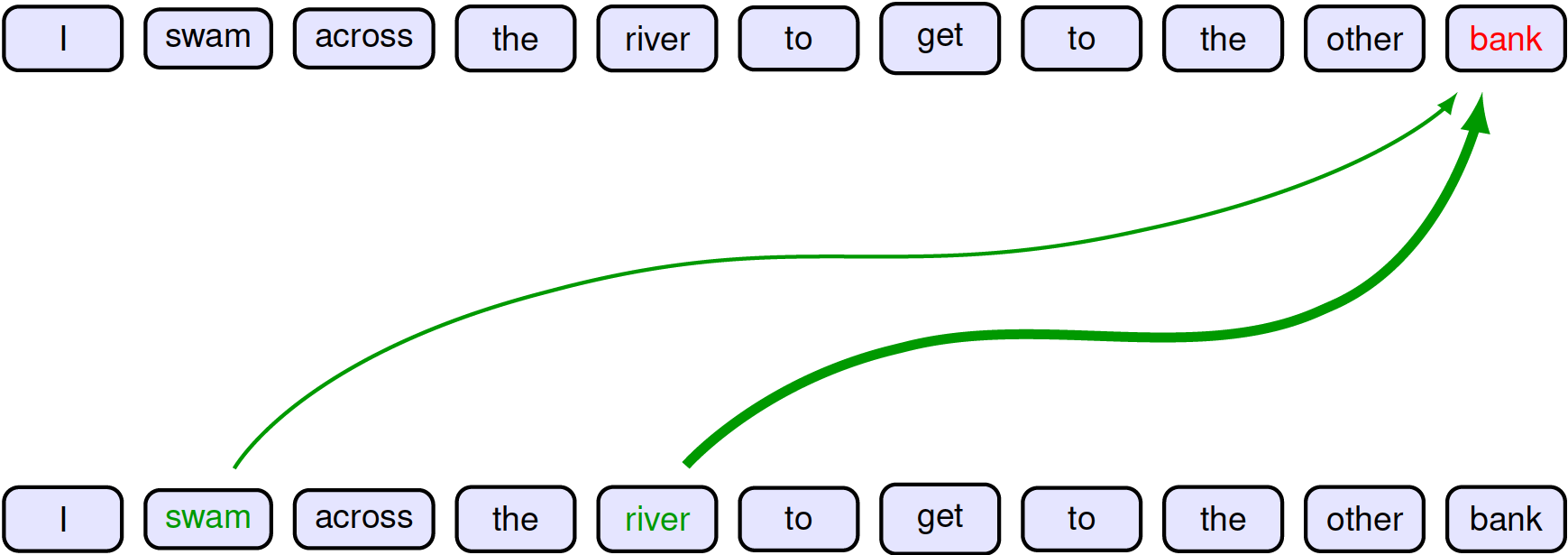


Рис. 4.1. Иллюстрация внимания. Значение слова «bank» находится под влиянием слов «river» и «swam». Толщина линии показывает силу влияния

Кроме того, видно, что слова, на которые нейросеть должна «обратить больше внимания» зависят от предложения: в предложении 1 это слова 2 и 5, в предложении 2 – слово 8. В обычных нейросетях сила влияния того или иного слова на результат зависит от значения соответствующего веса. После обучения такой нейросети эти веса фиксированны. При этом, веса внимания зависят от входных данных. На рисунке 4.2 показаны веса внимания трансформера, обученного на текстах.

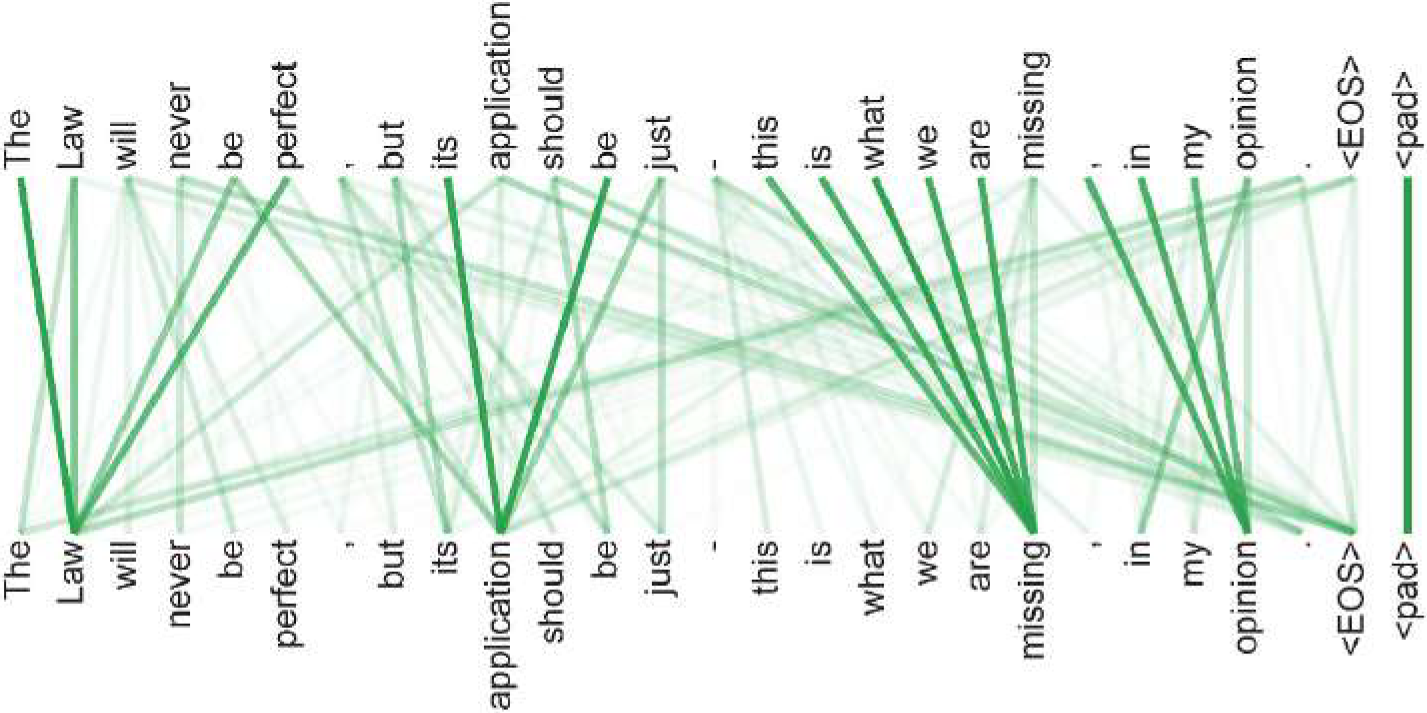


Рис. 4.2. Пример весов внимания (Vaswani et al., 2017)

Когда мы будем обсуждать обработку текстов на естественном языке, мы рассмотрим, как можно использовать эмбеддинги для того, чтобы представить слова в виде векторов в векторном пространстве. Эти векторы могут быть использованы в качестве входных данных для нейросети. Эмбеддинги содержат информацию о семантических свойствах слов, например, ставя в соответствие словам с похожими значениями близким векторам. Одно из свойств эмбеддингов заключается в том, что каждому слову соответствует один конкретный вектор. Трансформер можно рассматривать как более сложную форму эмбеддинга, в которой словам ставятся в соответствие векторы, которые зависят от других слов предложения. Поэтому, слово «bank» в рассмотренном примере соответствует двум разным векторам в разных предложениях. Например, в первом предложении вектор слова «bank» будет близок к вектору слова «water», а во втором предложении – к слову «cash».

Также в качестве примера внимания можно рассмотреть моделирование белков. Можно рассматривать белок как одномерную последовательность молекулярных единиц, называемых аминокислотами. Белок потенциально может содержать сотни или тысячи таких единиц, каждая из которых задается одним из 22 значений. В живой клетке белок сворачивается в трехмерную структуру, в которой аминокислоты, расположенные далеко друг от друга в одномерной последовательности, могут находиться близко в трехмерном пространстве и тем самым взаимодействовать. Модели-трансформеры позволяют этим удаленным аминокислотам «обращать внимание» друг на друга, тем самым значительно повышая точность моделирования их трехмерной структуры (Vig et al., 2020).

II. Обработка данных трансформером

Входные данные трансформера – набор векторов размерности , где . Будем называть эти векторы токенами, где токен может соответствовать, например, слову предложения, части изображения, или аминокислоте белка. Элементы токена называются признаками. Позже мы рассмотрим, как можно построить эти векторы токенов для текстов и для изображений. Мощным свойством трансформеров является тот факт, что нам не нужно разрабатывать новую архитектуру нейронной сети для обработки комбинации разных типов данных, а вместо этого мы можем просто объединить переменные данных в общий набор токенов.

Прежде чем мы сможем получить четкое представление о работе трансформатора, необходимо установить четкие обозначения. Мы будем следовать стандартным обозначениям и объединим векторы данных в матрицу размерности , в которой строка – это токен , (рисунок 4.3). Обратите внимание, что эта матрица содержит один набор токенов, в то время как в большинстве случаев данные будут содержать множество таких наборов токенов, таких как предложения, где каждому слову соответствует токен.

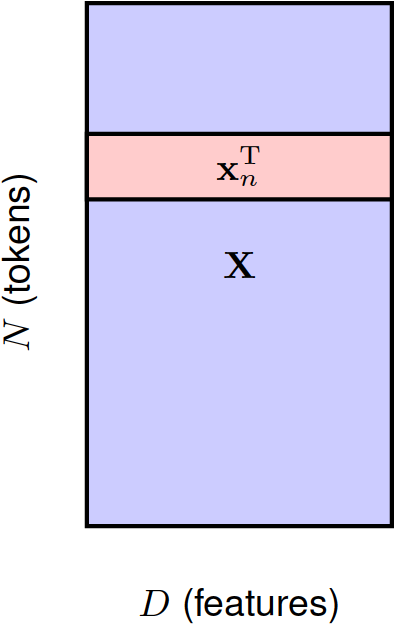


Рис. 4.3. Структура матрицы данных размерности , в которой строка n соответствует транспонированному вектору данных

Фундаментальным строительным блоком трансформера является функция, которая принимает на вход матрицу данных и возвращает преобразованную матрицу той же размерности. Мы можем записать эту функцию в следующем виде:

Затем мы можем последовательно применить несколько слоев-трансформеров для построения глубоких нейросетей, способных работать с богатыми внутренними представлениями. Каждый слой-трансформер имеет собственные веса и свободные члены, которые можно получить с помощью градиентного спуска, используя подходящую функцию стоимости, о чем мы подробно поговорим позже в этой главе.

Отдельный слой-трансформер состоит из двух этапов. В рамках первого этапа реализован механизм внимания, а второй этап преобразует токены независимо друг от друга. Начнем с рассмотрения механизма внимания.

III. Коэффициенты внимания

Пусть входные данные – это набор токенов в векторном пространстве эмбеддинга, который мы хотим трансформировать в набор , содержащий то же количество токенов, но в другом векторном пространстве эмбеддинга, содержащем более богатую семантическую структуру. Рассмотрим отдельный вектор . Значение вектора должно зависеть не только от соответствующего ему вектора , но и от всех остальных векторов входного набора . В соответствии с механизмом внимания сила этого влияния различна для разных векторов . Простой способ реализовать это – определить вектор как линейную комбинацию входных векторов с весовыми коэффициентами :

где – веса внимания.

Коэффициент должен быть близок к 0 для токенов, которые незначительно влияют на , и велик для токенов, которые имеют наибольшее влияние. Постановим, что коэффициенты неотрицательны, чтобы избежать ситуаций, когда один коэффициент большой положительный, а другой коэффициент, чтобы компенсировать это, большой (по модулю) отрицательный. Мы также хотим, чтобы «уделение большого внимания» одному токену происходило за счет «уделению меньшего внимания» остальным токенам, поэтому постановим, что сумма коэффициентов равна 1. Таким образом, весовые коэффициенты должны удовлетворять следующим ограничениям:

Из этих ограничений следует, что каждый коэффициент лежит в интервале , и каждый коэффициент составляет «часть целого». Для частного случая все остальные коэффициенты при , и значит , то есть входной вектор остается неизменным при трансформации. Таким образом, получается из входных векторов, при этом некоторые входные векторы имеют больший вес, чем другие.

Заметим, что у каждого выходного вектора свой набор коэффициентов, и ограничения (4.3) и (4.4) применяются отдельно для каждого . Коэффициенты зависят от входных данных, и далее мы рассмотрим, как они вычисляются.

IV. Вычисление коэффициентов (self-attention)

Следующий вопрос – как определить коэффициенты . Прежде чем мы обсудим это подробно, введем некоторую терминологию, взятую из области поиска информации. Рассмотрим задачу выбора фильма для просмотра в онлайн-кинотеатре. Один из подходов заключается в том, чтобы связать каждый фильм со списком атрибутов, описывающих такие вещи, как жанр (комедия, боевик и т. д.), имена актеров в главных ролях, продолжительность фильма и т. д. Затем пользователь сможет выполнить поиск по каталогу и найти фильм, соответствующий его предпочтениям. Можно автоматизировать это, закодировав атрибуты каждого фильма в векторе, называемом «*ключ»*. Сам соответствующий файл фильма называется *«значение»*. Аналогичным образом пользователь может затем предоставить свой собственный вектор значений желаемых атрибутов, который мы называем *«запрос»*. Затем онлайн-кинотеатр может сравнить вектор запроса со всеми векторами ключей, чтобы найти наилучшее совпадение, и предоставить соответствующий фильм пользователю в виде файла-значения. Мы можем представить себе, что пользователь, «обращает внимание» на конкретный фильм, ключ которого наиболее точно соответствует его запросу. Это можно рассматривать как «жесткое внимание», так как возвращается только одно значение. Для трансформера обобщим это и введем понятие «мягкое внимание», где для измерения степени соответствия между запросами и ключами используются непрерывные переменные, эти переменные используются определения весов влияния векторов значений на выходные данные. Это также гарантирует, что функция преобразователя будет дифференцируемой и, следовательно, может быть обучена методом градиентного спуска.

Следуя аналогии с поиском информации, мы можем рассматривать каждый из входных векторов как вектор значений, который будет использоваться для создания выходных токенов. Вектор также является ключом для входного токена . Это было бы аналогично использованию самого фильма для обобщения его характеристик. Наконец, мы можем использовать как вектор запроса для который затем можно сравнить с каждым из векторов-ключей. Чтобы увидеть, насколько , должен «обращать внимание» на каждый из токенов , нам нужно выяснить, насколько похожи эти векторы. В качестве простой меры сходства можно использовать скалярное произведение . Чтобы соблюдались ограничения (4.3) и (4.4), можно преобразовать это скалярное произведение с помощью функции :

Обратите внимание, что в этом случае у функции нет вероятностной интерпретации, и она просто используется для соответствующей нормализации весов внимания.

Таким образом, каждый входной вектор преобразуется в соответствующий выходной вектор путем вычисления линейной комбинации входных векторов (4.2), где вес , который применяется к входному вектору , равен функцией , от скалярного произведения между запросом и ключом (4.5). Обратите внимание, что если все входные векторы ортогональны, то каждый выходной вектор просто равен соответствующему входному вектору, то есть для .

Можно переписать (4.2) в матричной записи, где – входная матрица, – выходная матрица:

Далее будем использовать матричную запись.

Этот процесс называется self-attention так как используем одни и те же данные для определения запросов, ключей и значений. Позже в этой главе мы встретимся с различными вариантами этого механизма внимания.

V. Параметры нейросети

Рассмотренное выше преобразование входных векторов в выходные векторы является фиксированным и его нельзя обучать на основе данных, поскольку оно не имеет настраиваемых параметров. Более того, каждое из значений признаков в векторе играет равную роль в определении коэффициентов внимания, в то время как нужно, чтобы нейросеть могла придавать одним признакам большее значение, чем другим. Чтобы решить эти проблемы, можно применить к входным токенам линейное преобразование:

где – матрица обучаемых параметров размерности , аналогичная слою обычной нейросети. Тогда (4.6) принимает вид:

Такое преобразование имеет гораздо более высокую гибкость, однако матрица

симметрична. При этом, нужно, чтобы механизм внимания поддерживал значительную асимметрию. Например, слово «стамеска» должно быть тесно связано со словом «инструмент», поскольку стамеска является инструментом, а слово «инструмент» должно быть лишь слабо связано со словом «стамеска», потому что кроме стамески существует множество других видов инструментов. Хотя благодаря функции выходная матрица весов внимания не является симметричной, мы можем сделать модель гораздо более гибкой, дав возможность запросам и ключам иметь независимые параметры. Более того, преобразование (4.8) использует одну и ту же матрицу параметров для определения как векторов значений, так и коэффициентов внимания, что также является нежелательным ограничением.

Мы можем преодолеть эти ограничения, введя отдельные матрицы запросов, ключей и значений, линейные преобразования над которыми осуществляются независимо:

где , и – обучаемые параметры нейросети. Матрица имеет размерность , где — длина вектора-ключа. Матрица должна иметь ту же размерность , что и , чтобы было возможно вычислить скалярное произведение между вектором-запросом и вектором-ключом. Как правило, . Аналогично, — матрица размерности , где определяет размерность выходных векторов. Если мы установим , чтобы выходная матрица имела ту же размерность, что и входная, это облегчит добавление в нейросеть остаточных (residual) связей, которые мы обсудим позже. Кроме того, несколько слоев трансформера могут работать последовательно друг за другом, если каждый слой имеет одинаковую размерность. На основе этого (4.6) приобретает вид:

где имеет размерность , а имеет размерность . Механизм вычисления матрицы представлен на рисунке 4.4, а механизм вычисления матрицы представлен на рисунке 4.5.

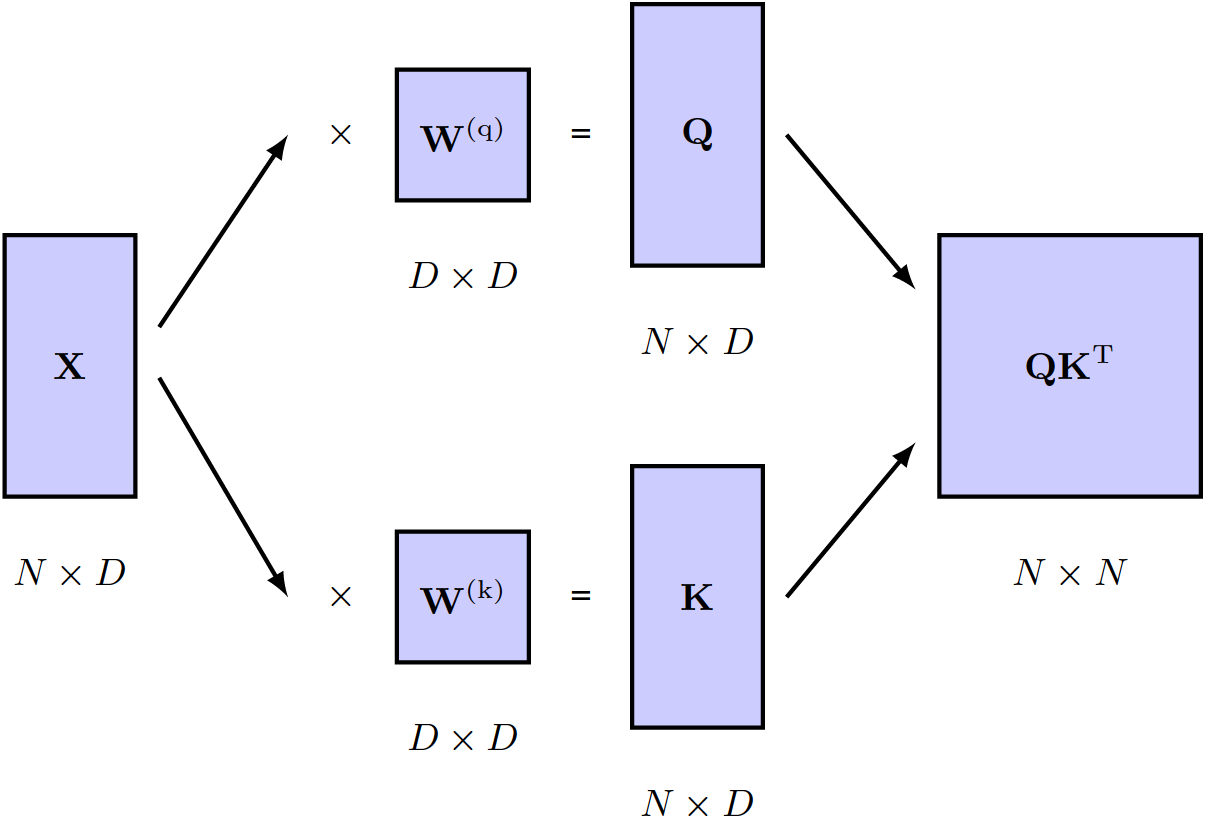


Рис. 4.4. Механизм вычисления матрицы коэффициентов внимания . Входные данные преобразуются с использованием формул (4.10) и (4.11) независимо друг от друга, полученные в результате преобразований матрица запросов и матрица ключей перемножаются

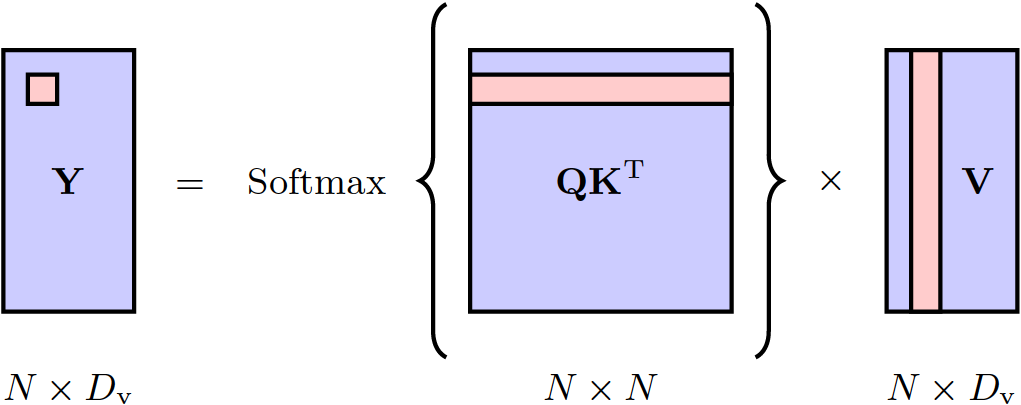


Рис. 4.5. Механизм вычисления выходной матрицы слоя-трансформера на основе матриц запросов, ключей и значений , и соответственно. Выделенное число матрицы равно скалярному произведению выделенной строки матрицы и выделенного столбца матрицы *V*

Мы также можем включить в эти линейные преобразования параметры смещения (свободные члены). Однако их можно включить в матрицы весов так же, как в случае обычных нейросетей: путем добавления к входной матрице дополнительного столбца, заполненного единицами, и добавления к матрицам весов дополнительной строки параметров. Далее мы не будем рассматривать параметры смещения явно, чтобы не усложнять выкладки.

По сравнению с обычной нейросетью, пути сигналов имеют мультипликативные отношения между значениями активации. В то время как обычные нейросети умножают значения активации на фиксированные веса, в трансформерах значения активации умножаются на зависящие от данных коэффициенты внимания. Это означает, например, что, если один из коэффициентов внимания близок к нулю для конкретного входного вектора, результирующий путь сигнала будет игнорировать соответствующий входящий сигнал, который, следовательно, не будет иметь никакого влияния на выход нейросети. Напротив, если обычная нейросеть обучится игнорировать конкретную входную скрытую переменную, она будет ее игнорировать для всех входных векторов.

V. Масштабирование коэффициентов

Есть еще одно улучшение, которое мы можем внести в слой-трансформер. Напомним, что градиенты функции становятся экспоненциально малыми для больших входов, так же как это происходит с или логистическо-сигмоидными функциями активации. Чтобы предотвратить это, мы можем отмасштабировать произведение векторов запроса и ключа перед применением функции . Чтобы получить подходящий коэффициент масштабирования, обратим внимание, что если бы все элементы векторов запроса ключа векторы были независимыми случайными величинами с нулевым математическим ожиданием и единичной дисперсией, то дисперсия скалярного произведения была бы равна . Поэтому мы нормализуем аргумент функции , используя стандартное отклонение, равное квадратному корню из , так что выходные данные слоя внимания принимают вид:

Это – финальная версия рассматриваемого нами слоя внимания. Структура этого слоя представлена на рисунке 4.6 и в алгоритме 4.1.

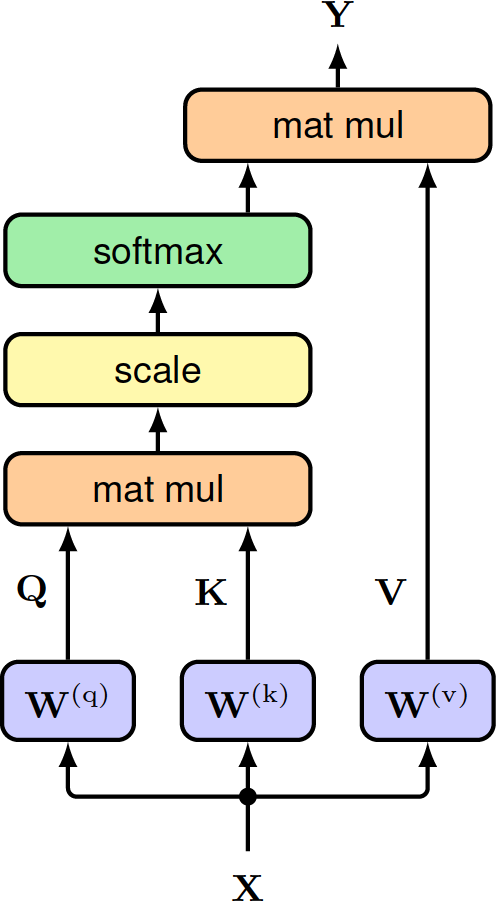


Рис. 4.6. Информационный поток в масштабированном слое внимания.

Здесь обозначает умножение матриц, а обозначает нормализацию аргумента функции с использованием . Эта структура представляет собой единую «голову внимания» (attention head).

===========================================

===========================================

Алгоритм:

===========================================

Алгоритм 4.1. Масштабированный слой внимания

VI. Многоголовое внимание (multi-head attention)

Описанный выше слой внимания называется «головой внимания» (attention head) и позволяет выходным векторам «обращать внимание» на зависящие от данных шаблоны входных векторов. Однако во многих случаях полезно использовать несколько различных моделей внимания параллельно. Например, при обработке текстов одна модель внимания может обрабатывать время, а другая значение слов. Использование одной «головы внимания» может привести к усреднению этих эффектов. Вместо этого можно использовать несколько «голов внимания» параллельно. Они состоят из одинаково структурированных копий одной головы с независимыми друг от друга обучаемыми параметрами, которые управляют вычислением матриц запросов, ключей и значений. Это аналогично использованию нескольких разных фильтров в каждом слое сверточной нейросети.

Пусть у нас есть голов с номерами :

где обозначение было введено в (4.14). У каждой «головы» свои матрицы запросов, ключей и значений:

«Головы внимания» сначала объединяются (конкатенируются) в одну матрицу, а затем осуществляется линейное преобразование результата с использованием матрицы для получения объединенного выхода:

Это изображено на рисунке 4.7.

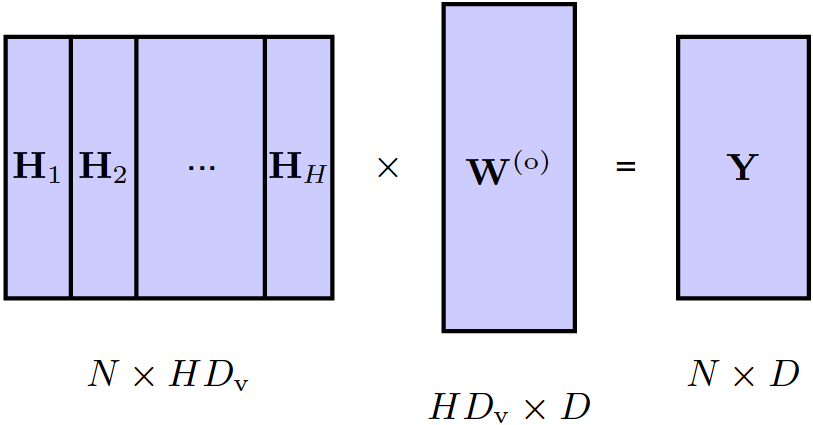


Рис. 4.7. Архитектура «многоголового внимания». Каждая «голова внимания» имеет структуру, показанную на рис. 4.6, и имеет свои собственные матрицы параметров ключей, запросов и значений. Выходные данные «голов внимания» объединяются (конкатенируются), а затем осуществляется их линейное преобразование обратно к размерности входных данных

Каждая матрица имеет размерность , поэтому объединенная матрица имеет размерность Она умножается на матрицу размерности , чтобы получить финальную выходную матрицу такой же размерности , как у исходной входной матрицы . Элементы матрицы – это обучаемые параметры. Обычно берется равным , чтобы объединенная матрица имела размерность . Механизм работы «многоголового внимания» представлен в алгоритме 4.2, а информационный поток в слое «многоголового внимания» изображен на рисунке 4.8.

//объединение матриц

Алгоритм 4.2. «Многоголовое внимание»

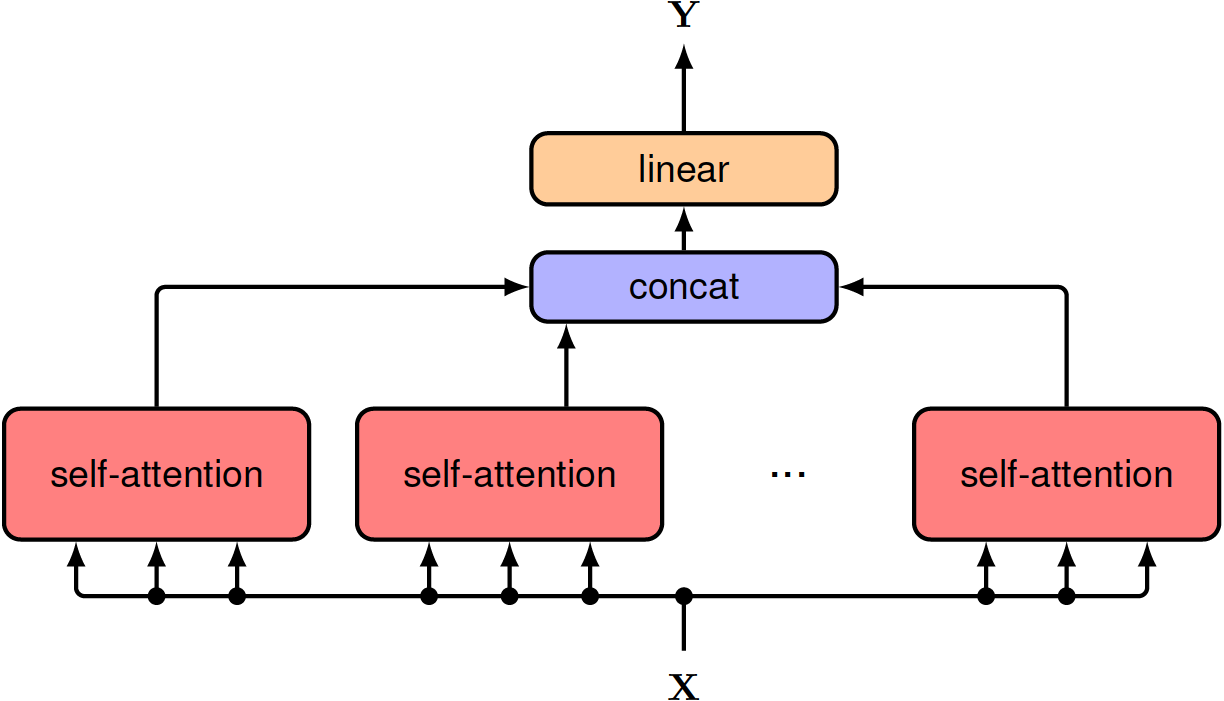


Рис. 4.8. Информационный поток в слое «многоголового внимания»

Обратите внимание, что приведенная выше формулировка «многоголового внимания» содержит некоторую избыточность, заключающуюся в умножении матрицы каждой «головы внимания» и матрицы . Удаление этой избыточности позволяет записать слой-трансформер с несколькими «головами внимания» как сумму вкладов каждой из «голов внимания».

VII. Слои-трансформеры

«Многоголовое внимание» является основным архитектурным элементом нейросети-трансформера. Мы знаем, что качество нейросети значительно повышается с ростом глубины, поэтому нам хотелось бы разместить несколько слоев самообслуживания друг за другом. Для повышения эффективности обучения, мы можем ввести остаточные соединения (residual connections), которые обходят «многоголовое внимание». Для этого требуется, чтобы выходной матрицы была такой же, как у входной матрицы, а именно . Далее осуществляется нормализация слоев (Ba, Kiros, and Hinton, 2016), повышающая эффективность обучения. Итоговое преобразование можно записать как:

где определяется по формуле (4.19). Иногда нормализацию слоя заменяют на предварительную нормализацию, при которой нормализация применяется до «многоголового внимания», а не после, поскольку это может привести к более эффективной оптимизации:

В обоих случаях матрица снова имеет ту же размерность , что и входная матрица .

Видно, что механизм внимания создает линейные комбинации векторов значений, которые затем линейно комбинируются для получения выходных векторов. Кроме того, значения являются линейными функциями входных векторов, поэтому мы видим, что выходные данные слоя внимания являются линейными комбинациями входных данных. Нелинейность существует в весах внимания модели, поэтому выход будут нелинейно зависеть от входов через функцию , но выходные векторы по-прежнему находятся в подпространстве входных векторов, и это ограничивает возможности слоя внимания. Мы можем повысить гибкость трансформера путем постобработки выхода каждого слоя с использованием стандартной нелинейной нейросети, имеющей входов и выходов, которая называется (многослойный персептрон). Например, такая нейросеть может быть двухуровневой полносвязной нейросетью. с функцией активации . Это необходимо сделать таким образом, чтобы сохранилась способность трансформера обрабатывать последовательности переменной длины. Для этого к каждому из выходных векторов, соответствующих строкам матрицы , применяется одна и та же общая нейросеть. Опять же, такой слой нейросети можно улучшить путем добавления остаточного соединения. Этот слой также включает в себя нормализацию, так что слой имеет вид:

Механизм функционирования слоя-трансформера представлен на рисунке 4.9 и в алгоритме 4.3. Кроме того, можно применить предварительную нормализацию:

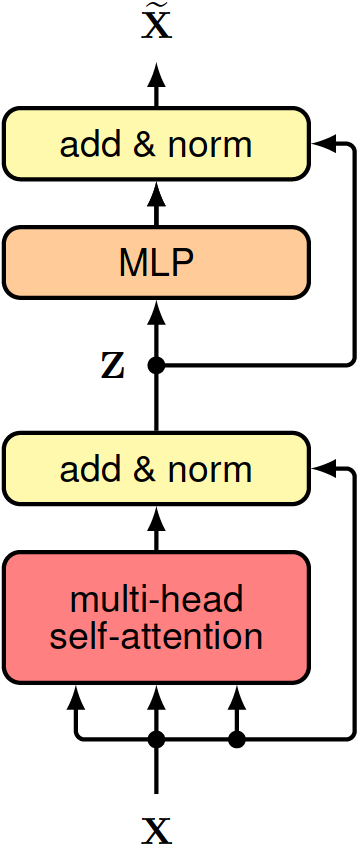


Рис. 4.9. Слой-трансформер. Здесь обозначает многослойный персептрон, а – нормализацию слоя.

Алгоритм 4.3. Слой-трансформер

Как правило, в нейросети-трансформере несколько таких слоев следуют друг за другом. Слои, как правило, имеют одинаковую структуру, но разные веса и смещения.

VIII. Вычислительная сложность

Слой внимания, обсуждавшийся до сих пор, принимает на вход набор из векторов, каждый из которых имеет длину , и отображает их в другой набор из векторов, имеющих ту же длину. Таким образом, входы и выходы имеют одинаковую размерность . Если бы мы использовали стандартную полносвязную нейросеть для сопоставления входных значений с выходными значениями, она имела бы независимых параметров. Аналогично, вычислительные затраты на один проход «вперед» через такую нейросеть также будут равны .

В слое внимания матрицы , и являются общими для входных токенов, поэтому количество независимых параметров равно , при условии, что . Поскольку число входных токенов равно , количество вычислительных шагов при вычислении скалярного произведения в слое внимания составляет . Мы можем рассматривать слой внимания как разреженную матрицу, в которой параметры являются общими для различных блоков матрицы. Последующий слой нейросети, имеющий входов и выходов имеет вычислительную сложность . Поскольку он является общим для всех токенов, его сложность линейна по N, и поэтому в целом этот слой имеет вычислительную сложность, равную . В зависимости от относительных величин и , как слой трансформатора, так и слой MLP может доминировать в вычислительных затратах. По сравнению с полносвязной сетью слой-трансформер более эффективен в вычислительном отношении. Было предложено множество вариантов архитектуры трансформера (Lin et al., 2021; Phuong и Hutter, 2022), включая модификации, направленные на повышение эффективности (Tay et al., 2020).

IX. Позиционное кодирование

Как матрицы параметров внимания и , так и матрицы параметров многослойного персептрона являются общими для всех входных токенов. Как следствие, трансформер обладает следующим свойством: изменение порядка токенов, то есть строк матрицы , приводит к такой же перестановке строк выходной матрицы . Другими словами, трансформер эквивариантен относительно перестановок токенов. Тот факт, что параметры являются общими для всех входных токенов, облегчает распараллеливание трансформера, а также позволяет сети обрабатывать дальнодействующие зависимости так же эффективно, как короткодействующие. Однако тот факт, что порядок токенов не учитывается трансформером, является значительным недостатком при анализе последовательных данных, таких как тексты. Два предложения «The food was bad, not good at all» и «The food was good, not bad at all» содержат одни и те же слова, но имеют противоположные значения из-за разного порядка слов. Очевидно, что порядок токенов имеет очень важное значение в большинстве задач обработки последовательных данных, включая обработку текстов, поэтому нам нужно найти способ внедрить в нейросеть информацию о порядке токенов.

Так как мы стремимся сохранить преимущества слоев внимания, мы будем не внедрять обработку порядка в структуру нейросети, а кодировать информацию о порядке в токенах. Поэтому для каждого номера позиции построим вектор , в котором закодирована информация о позиции, и затем объединим его с соответствующим токеном . Очевидным способом объединения этих векторов является конкатенация, но это увеличит размерность входных данных и всей нейросети, что приведет к значительному увеличению вычислительной сложности. Вместо этого мы можем просто суммировать эти векторы:

Для этого нужно, чтобы у векторов позиции была та же размерность, что у токенов.

На первый взгляд может показаться, что такое суммирование векторов исказит токены и значительно усложнит задачу нейросети. Однако сформировать представление, почему суммирование может быть эффективным, можно на основе того факта, что два случайно выбранных некоррелированных вектора, как правило, являются почти ортогональными в пространствах высокой размерности, что указывающая на то, что нейросеть способна обрабатывать вектор эмбеддинга и информацию о позиции относительно независимо. Также обратите внимание, что благодаря остаточным соединениям каждого слоя информация о позиции не потеряется при переходе от одного слоя трансформера к другому. Более того, из-за наличия в трансформере слоев линейных преобразований, суммирование и конкатенация имеют схожие свойства.

Следующая задача — построить векторы позиции . Простой способ решения этой задачи – использовать номер позиции (1, 2, 3, …). Однако в этом случае возникает проблема, что номер позиции может стать сколь угодно большим и начать значительно искажать вектор эмбеддинга. Кроме того, этот метод может не обобщаться эффективно на новые входные последовательности, имеющие большую длину, чем последовательности обучающей выборки, поскольку они будут включать в себя закодированные значения длины последовательности, выходящие за пределы интервала рассматривавшихся при обучении. Вместо этого мы могли бы присваивать число в диапазоне каждому токену последовательности: что сделало бы кодирование ограниченным. Однако такое кодирование будет для одного и того же номера позиции возвращать разные значения, зависящие от длины последовательности.

Идеальное позиционное кодирование должно обеспечивать уникальное значение для каждой позиции, должно быть ограниченным, должно обобщаться на более длинные последовательности и должен иметь стабильный способ выразить расстояние между любыми двумя входными токенами независимо от их абсолютной позиции, поскольку относительное расположение токенов часто важнее абсолютного расположения.

Существует множество подходов к позиционному кодированию (Dufter, Schmitt, and Schütze, 2021). Здесь мы описываем метод, основанный на синусоидальных функциях, введенных Васвани и др. (Vaswani et al., 2017). Для номера позиции вектор позиционного кодирования будет содержать следующие значения:

Видно, что элементы вектора задаются рядом синусов и косинусов с постепенно растущей длиной волны, как показано на рисунке 4.10(a).

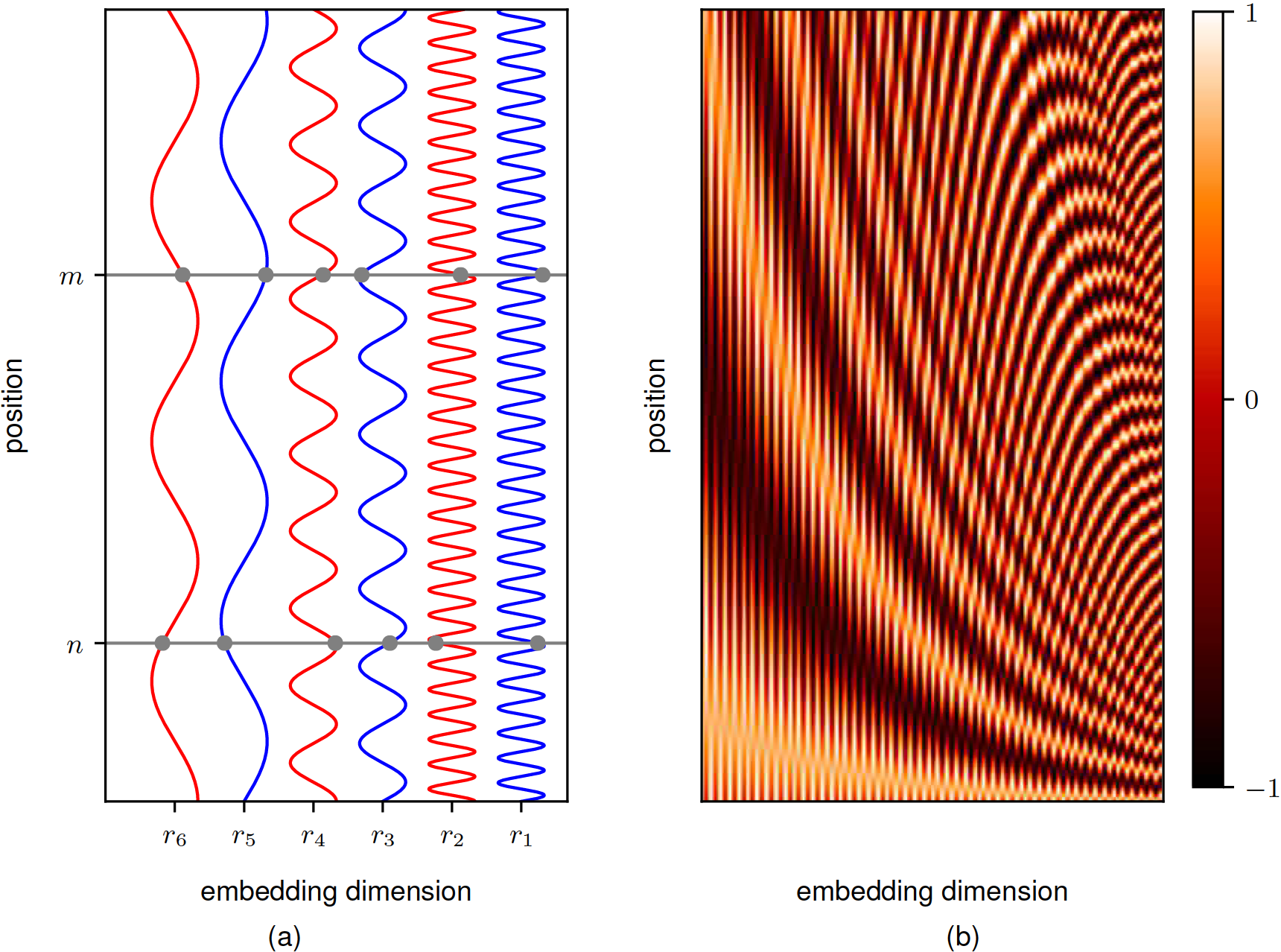
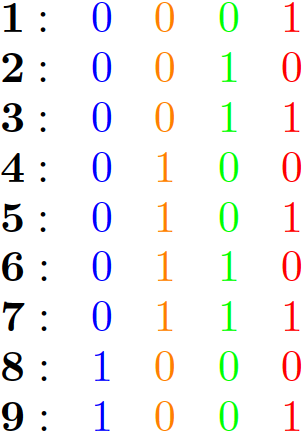


Рис. 4.10. Иллюстрация функций, заданных формулой (4.25) и используемых для построения векторов позиционного кодирования. (а) График, где горизонтальная ось – элементы вектора позиционного кодирования , а вертикальная ось – номер последовательности. Значения элементов вектора для позиций и показаны пересечениями графиков синуса и косинуса горизонтальными линиями. (b) Тепловая карта векторов позиционного кодирования для размерности с для первых позиций.

Этот метод кодирования обладает тем свойством, что все элементы вектора лежат в диапазоне . Это напоминает двоичные числа: бит младшего порядка чередуется с высокой частотой, а последующие биты чередуются с постепенно уменьшающейся частотой:



Однако для кодирования, заданного (4.25), элементы вектора являются непрерывными переменными, а не двоичными. График векторов позиционного кодирования положения показан на рисунке 4.10(b).

Одним из полезных свойств синусоидального представления, заданного формулой (4.25), является тот факт, что для любого фиксированного смещения код позиции можно представить как линейную комбинацию кода позиции , коэффициенты которой зависят только , но не от . Поэтому нейросеть должна быть в состоянии обучиться «обращать внимание» на относительные позиции. Обратите внимание, что это свойство требует, чтобы при кодировании использовались функции как синуса, так и косинуса.

Другой распространенный подход – обучаемое позиционное кодирования. Это достигается за счет добавления в нейросеть вектора весов для каждой позиции, который можно обучается совместно с остальными параметрами модели. Поскольку параметры не являются общими для позиций токенов, токены больше не являются инвариантными при перестановке, что и является целью позиционного кодирования. Однако этот подход не соответствует заданному нами ранее критерию обобщения на более длинные входные последовательности, поскольку кодирование не будет обучено для позиций, которые не встретились при обучении модели. Следовательно, этот подход наиболее подходит для случаев, когда длина входных данных относительно постоянна как во время обучения, так и во время использования модели.

X. Обработка текстов

Теперь, когда мы изучили архитектуру трансформера, мы рассмотрим, как его можно использовать для обработки языковых данных, состоящих из слов, предложений и абзацев. Хотя это – предметная область, для работы с которой трансформеры изначально были разработаны, они оказались очень общим классом моделей и хорошо подходят для большинства типов входных данных. Позже в этой главе мы рассмотрим использование трансформеров в других предметных областях.

Многие языки, включая английский, содержат ряд слов, разделенных пробелами, а также символы пунктуации и, следовательно, представляют собой последовательные данные. На данный момент мы сосредоточимся на словах, а к пунктуации вернемся позже.

Первая задача – преобразовать слова в числовые представления, подходящее для использования в качестве входных данных для глубокой нейронной сети. Один простой подход состоит в том, чтобы взять фиксированный словарь слов, и кодировать слово, стоящее в словаре на -ом месте, вектором, длина которого равна количеству слов в словаре, где на позиции *k* стоит «1», а на остальных позициях – «0» (one-hot кодирование). Например, если «муравьед» — третье слово в нашем словаре, то его векторное представление будет .

Очевидная проблема с one-hot кодированием состоит в том, что реалистичный словарь может иметь несколько сотен тысяч слов, что приведет к векторам очень высокой размерности. Кроме того, он не отражает никаких сходств или взаимоотношений между словами. Обе проблемы можно решить путем отображения слов в пространство более низкой размерности с помощью процесса, называемого эмбеддингом, в котором каждое слово представляется как плотный вектор в пространстве, обычно имеющем несколько сотен измерений.

XI. Эмбеддинги

Процесс эмбеддинга может быть определен как матрица размера , где — размерность пространства эмбеддинга, а — размер словаря. Затем для каждого вектора one-hot кодирования можно получить вектор эмбеддинга по следующей формуле:

Поскольку – вектор one-hot кодирования, вектор будет просто равен соответствующему столбцу матрицы .

Мы можем обучить матрицу по корпусу (т. е. большому набору данных) текста, и существует множество подходов к такому обучению. Здесь мы рассмотрим популярную технику *word2vec* (Mikolov et al., 2013), которую можно рассматривать как простую двухслойную нейросеть. Строится обучающая выборка, в которой каждый объект получается путем рассмотрения «окна» из соседних слов в тексте, где типичным значением является . Объекты считаются независимыми, а функция ошибок равна сумме функций ошибок для каждого объекта. Есть два варианта этого подхода. В подходе *непрерывного мешка слов* целевой переменной для обучения сети является центральное слово, а остальные слова (*контекст*) формируют входные данные, так что сеть обучается «заполнять пробелы». Родственный подход *skip-grams* меняет местами входные и выходные данные, так что входными данными является центральное слово, а слова контекста являются целевой переменной. Эти модели показаны на рисунке 5.1.

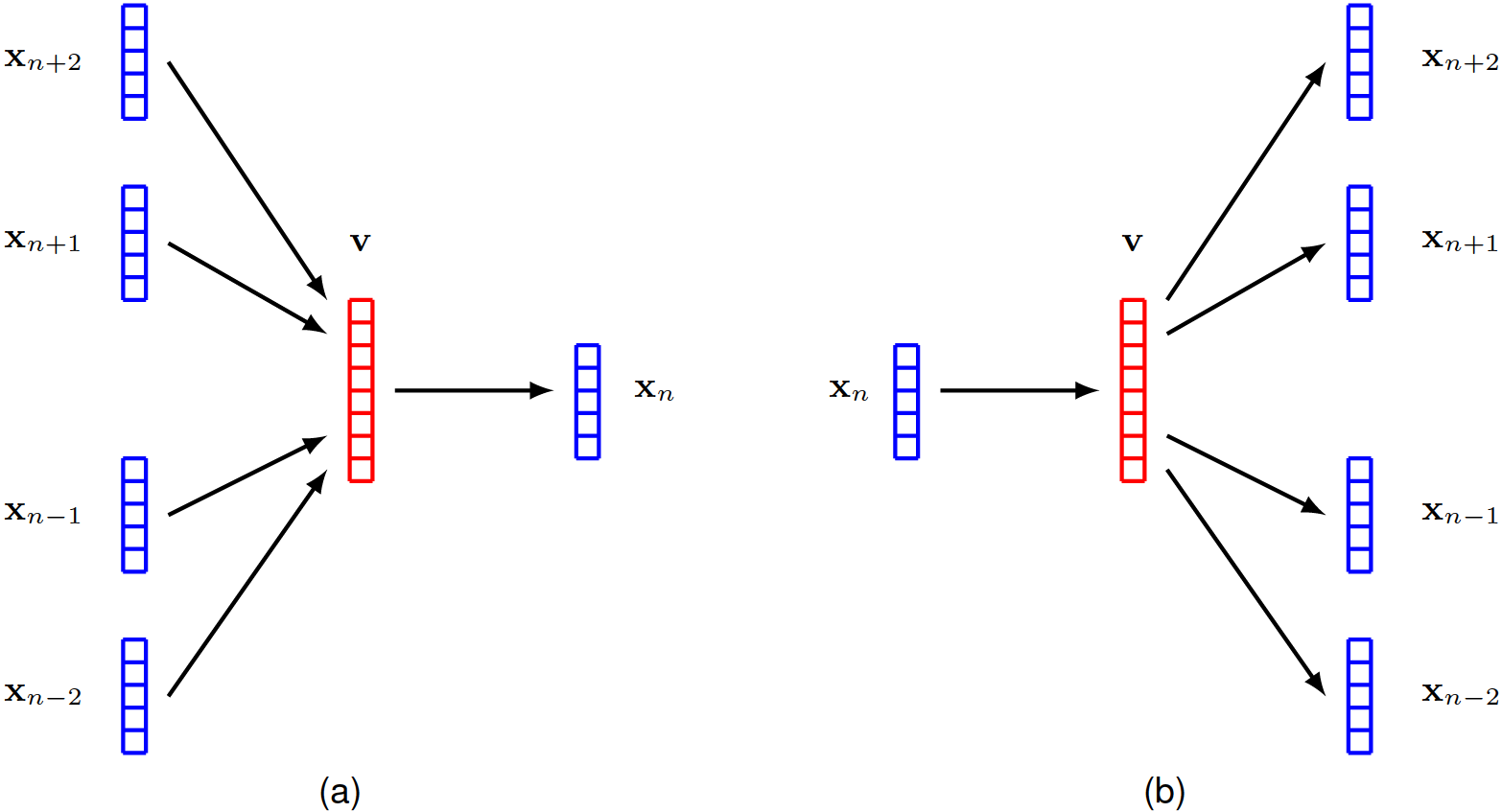


Рис. 4.11. Двухслойные нейросети, используемые для обучения эмбеддинга, где (a) – подход непрерывного мешка слов, а (b) – подход skip-grams

Эту процедуру обучения можно рассматривать как форму self-supervised learning, поскольку входными данными является большой массив неразмеченного текста, из которого случайным образом извлекается множество маленьких окон последовательных слов. Метки получаются из самого текста путем «маскировки» тех слов, которые являются целевой переменной нейросети.

После обучения модели матрица эмбеддинга определяется транспонированием матрицы весов второго слоя для подхода непрерывного мешка слов и транспонированием матрицы весов первого слоя для подхода skip-grams. Семантически связанные слова, сопоставляются с соседними позициями в пространстве эмбеддинга. Этого следовало ожидать, поскольку семантически связанные слова с большей вероятностью имеют похожие контекстные слова, чем несвязанные слова. Например, слова «город» и «столица» чаще встречаются в контексте таких слов, как «Париж» и «Лондон», и реже – в контексте таких слов, как «оранжевый» и «полиномиальный». Нейросети проще предсказывать вероятность пропущенных слов, если «Париж» и «Лондон» сопоставляются с соседними векторами эмбеддинга.

Оказывается, обученное пространство эмбеддинга часто имеет даже более богатую семантическую структуру, чем просто близость связанных слов, и это позволяет выполнять простую векторную арифметику. Например, концепция «Париж для Франции – то же самое, что Рим для Италии» может быть выражена посредством операций над векторами эмбеддинга. Если мы используем как обозначение вектора эмбеддинга слова, то это можно записать так:

Эмбеддинги изначально разрабатывались как самостоятельный инструмент обработки естественного языка. Сегодня они чаще всего используются как этап предобработки для глубоких нейронных сетей. В этом отношении их можно рассматривать как первый слой глубокой нейросети. В этом случае может применяться некоторая стандартная предварительно обученной матрица эмбеддинга, или эмбеддинг можно рассматривать как адаптивный слой, который обучается в процессе общего обучения системы. Во втором случае слой эмбеддинга может быть инициализирован либо случайными значениями, либо стандартной матрицей эмбеддинга.

## §5. Языковые модели на основе трансформеров

Слой обработки трансформера представляет собой чрезвычайно гибкий компонент для построения мощных моделей нейронных сетей с широкой применимостью. В этом разделе мы исследуем применение трансформаторов для обработки естественного языка. Это привело к разработке массивных нейронных сетей, известных как большие языковые модели (LLM), которые продемонстрировали исключительную способность к решению задач (Zhao et al., 2023).

Трансформаторы могут быть применены ко многим различным типам задач обработки языка и могут быть сгруппированы в три категории в соответствии с формой входных и выходных данных. В такой задаче, как анализ тональности, мы принимаем на вход последовательность слов и предоставляем на выходе одну переменную, представляющую тональность текста, например, "радостный" или "грустный". Здесь трансформатор действует как "кодировщик" последовательности. Другие задачи могут принимать на вход единственный вектор и генерировать на выходе последовательность слов, например, если мы хотим сгенерировать текстовую подпись к входному изображению. В таких случаях трансформатор функционирует как "декодировщик", генерируя последовательность на выходе. Наконец, в задачах обработки "последовательность-последовательность" и вход, и выход представляют собой последовательность слов, например, если наша цель — перевод с одного языка на другой. В этом случае трансформаторы используются как в роли кодировщика, так и декодировщика. Мы обсудим каждый из этих классов языковых моделей по очереди, используя наглядные примеры архитектур моделей.

I. Трансформеры-декодеры

Мы начнем с рассмотрения моделей трансформеров, использующих только декодер. Их можно использовать в качестве генеративных моделей, которые создают выходные последовательности токенов. В качестве наглядного примера мы сосредоточимся на классе моделей под названием GPT (Generative Pre-trained Transformer — генеративный предобученный трансформер) (Radford et al., 2019; Brown et al., 2020; OpenAI, 2023). Цель состоит в том, чтобы использовать архитектуру трансформера для построения авторегрессионной модели вида, определенного в (12.31), в которой условные распределения выражены с помощью нейронной сети-трансформера, обучаемой на данных.

Модель принимает на вход последовательность, состоящую из первых n−1 токенов, и ее соответствующий выход представляет условное распределение для n-го токена. Если мы возьмем выборку из этого распределения, то мы расширили последовательность до n токенов, и эту новую последовательность можно снова пропустить через модель, чтобы получить распределение для токена n + 1 и так далее. Этот процесс можно повторять для генерации последовательностей вплоть до максимальной длины, определяемой количеством входов трансформера. Мы вскоре обсудим стратегии выборки из условных распределений, а пока сосредоточимся на том, как построить и обучить сеть.

Архитектура модели GPT состоит из стека слоев-трансформеров, которые принимают на вход последовательность токенов , каждый размерности D, и выдают на выходе последовательность токенов , также размерности D. Каждый выход должен представлять распределение вероятностей по словарю токенов для данного временного шага, и этот словарь имеет размерность K, тогда как токены имеют размерность D. Поэтому мы выполняем линейное преобразование каждого выходного токена с помощью матрицы W(p) размерности D × K с последующей функцией активации softmax в форме:

(5.1)

где Y — матрица, n-я строка которой равна yTn, а e X — матрица, n-я строка которой равна xTn. Каждый выходной узел softmax имеет связанную с ним функцию ошибки перекрестной энтропии. Архитектура модели показана на Рисунке 5.1.

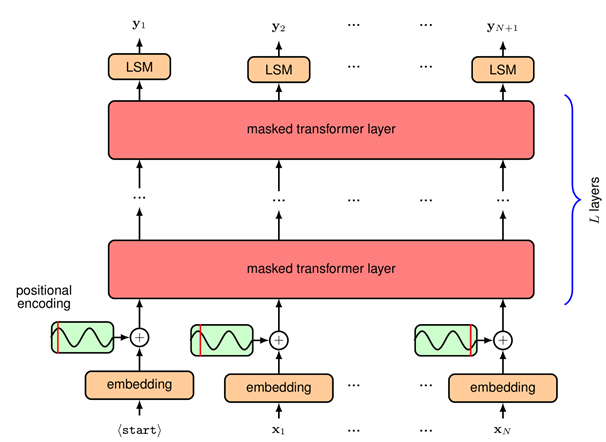


Рисунок 5.1. Архитектура декодера трансформера GPT. Здесь «LSM» обозначает linear-softmax — линейное преобразование, параметры которого обучаются и разделяются между всеми позициями токенов, после чего применяется функция активации softmax. Маскирование объясняется в тексте.

Модель можно обучать на большом корпусе неразмеченного естественного языка, используя самоконтролируемый (self-supervised) подход. Каждый пример обучения состоит из последовательности токенов x1,...,xn, которые образуют вход сети, вместе со связанным целевым значением xn+1, состоящим из следующего токена в последовательности. Последовательности считаются независимыми и одинаково распределенными, так что функция ошибки, используемая для обучения, представляет собой сумму значений ошибки перекрестной энтропии, суммированную по обучающему набору, сгруппированному в соответствующие мини-пакеты (mini-batches). Наивно мы могли бы обрабатывать каждый такой пример обучения независимо, используя прямой проход через модель. Однако мы можем достичь гораздо большей эффективности, обрабатывая всю последовательность целиком, так что каждый токен действует и как целевое значение для последовательности предыдущих токенов, и как входное значение для последующих токенов. Например, рассмотрим последовательность слов:

«Я переплыл реку, чтобы добраться до другого берега.»

Мы можем использовать «Я переплыл» как входную последовательность с associated целью «реку», а также использовать «Я переплыл реку» как входную последовательность с associated целью «чтобы» и так далее. Однако, чтобы обрабатывать их параллельно, мы должны убедиться, что сеть не может «схитрить», заглядывая вперед по последовательности, иначе она просто научится копировать следующий вход напрямую на выход. Если бы она так делала, то тогда она не смогла бы генерировать новые последовательности, поскольку последующий токен по определению недоступен во время тестирования. Чтобы решить эту проблему, мы делаем две вещи. Во-первых, мы смещаем входную последовательность вправо на один шаг, так что входному xn соответствует выход yn+1, с целевым значением xn+1, и дополнительный специальный токен, обозначаемый ⟨start⟩, добавляется в первую позицию входной последовательности. Во-вторых, заметим, что токены в трансформере обрабатываются независимо, за исключением тех случаев, когда они используются для вычисления весов внимания, когда они взаимодействуют попарно через скалярное произведение. Поэтому мы вводим маскированное внимание (masked attention), иногда называемое причинным вниманием (causal attention), в каждый из слоев внимания, в котором мы обнуляем все веса внимания, соответствующие тому, что токен «обращает внимание» на любой последующий токен в последовательности. Это просто involves обнуление всех соответствующих элементов матрицы внимания Attention(Q,K,V), определенной в (4.14), с последующей нормализацией оставшихся элементов так, чтобы каждая строка снова суммировалась в единицу. На практике этого можно достичь, установив соответствующие значения пред-активации в −∞, так что softmax даст ноль для связанных выходов и также позаботится о нормализации по ненулевым выходам. Структура матрицы маскированного внимания проиллюстрирована на Рисунке 5.2.



Рисунок 5.2. Архитектура декодера трансформера GPT. Здесь «LSM» обозначает linear-softmax — линейное преобразование, параметры которого обучаются и разделяются между всеми позициями токенов, после чего применяется функция активации softmax. Маскирование объясняется в тексте.

На практике мы хотим эффективно использовать массивный параллелизм GPU, и, следовательно, несколько последовательностей могут быть объединены вместе во входной тензор для параллельной обработки в одном пакете (batch). Однако это требует, чтобы последовательности имели одинаковую длину, тогда как текстовые последовательности естественным образом имеют переменную длину. Это можно решить, введя специальный токен, который мы обозначим ⟨pad⟩, который используется для заполнения неиспользуемых позиций, чтобы довести все последовательности до одинаковой длины, чтобы их можно было объединить в один тензор. Затем в весах внимания используется дополнительная маска, чтобы гарантировать, что выходные векторы не «обращают внимания» на любые входы, занятые токеном ⟨pad⟩. Заметим, что форма этой маски зависит от конкретной входной последовательности.

Выходом обученной модели является распределение вероятностей по пространству токенов, задаваемое функцией активации softmax на выходе, которое представляет вероятность следующего токена при условии текущей последовательности токенов. Как только это следующее слово выбрано, последовательность токенов с включенным новым токеном можно снова пропустить через модель, чтобы сгенерировать последующий токен в последовательности, и этот процесс можно повторять неопределенно долго или до тех пор, пока не будет сгенерирован токен конца последовательности. Это может показаться довольно неэффективным, поскольку данные должны пропускаться через всю модель для каждого нового сгенерированного токена. Однако заметим, что из-за маскированного внимания embedding, полученный для конкретного токена, зависит только от самого этого токена и от предыдущих токенов и, следовательно, не изменяется, когда генерируется новый, последующий токен. Следовательно, большая часть вычислений может быть переиспользована при обработке нового токена.

II. Стратегии выборки

Мы видели, что выход декодер-трансформера — это распределение вероятностей по значениям для следующего токена в последовательности, из которого должно быть выбрано конкретное значение для этого токена, чтобы расширить последовательность. Существует несколько вариантов выбора значения токена на основе вычисленных вероятностей (Holtzman et al., 2019). Один очевидный подход, называемый **жадным поиском** (greedy search), состоит в том, чтобы просто выбрать токен с наивысшей вероятностью. Это приводит к тому, что модель становится детерминированной, поскольку данная входная последовательность всегда генерирует одну и ту же выходную последовательность. Заметим, что простое выбирание токена с наивысшей вероятностью на каждом этапе — это не то же самое, что выбор наиболее вероятной последовательности токенов. Чтобы найти наиболее вероятную последовательность, нам нужно было бы максимизировать совместное распределение по всем токенам, которое задается формулой 5.2.

(5.2)

Если в последовательности N шагов и количество значений токенов в словаре равно K, то общее количество последовательностей составляет , что растет экспоненциально с длиной последовательности, и, следовательно, найти одну наиболее вероятную последовательность невыполнимо. Для сравнения, жадный поиск имеет стоимость , которая линейна по длине последовательности.

Одна методика, которая потенциально может генерировать последовательности с более высокой вероятностью, чем жадный поиск, называется поиском по лучу (beam search). Вместо того чтобы выбирать одно наиболее вероятное значение токена на каждом шаге, мы сохраняем набор из B гипотез, где B называется шириной луча (beamwidth), каждая из которых состоит из последовательности значений токенов вплоть до шага n. Затем мы пропускаем все эти последовательности через сеть, и для каждой последовательности находим B наиболее вероятных значений токена, создавая таким образом B² возможных гипотез для расширенной последовательности. Затем этот список сокращается путем выбора B наиболее вероятных гипотез в соответствии с общей вероятностью расширенной последовательности. Таким образом, алгоритм поиска по лучу сохраняет B альтернативных последовательностей и отслеживает их вероятности, в конечном итоге выбирая наиболее вероятную последовательность среди рассмотренных. Поскольку вероятность последовательности получается путем умножения вероятностей на каждом шаге последовательности, и поскольку эти вероятности всегда меньше или равны единице, длинная последовательность обычно будет иметь меньшую вероятность, чем короткая, смещая результаты в сторону коротких последовательностей. По этой причине вероятности последовательностей обычно нормализуются по соответствующим длинам последовательностей перед сравнением. Поиск по лучу имеет стоимость , которая снова линейна по длине последовательности. Однако стоимость генерации последовательности увеличивается в B раз, и поэтому для очень больших языковых моделей, где стоимость вывода (inference) может стать значительной, это делает поиск по лучу гораздо менее привлекательным.

Одна проблема с такими подходами, как жадный поиск и поиск по лучу, заключается в том, что они ограничивают разнообразие потенциальных выходных данных и могут даже привести к тому, что процесс генерации зациклится, когда одна и та же подпоследовательность слов повторяется снова и снова.

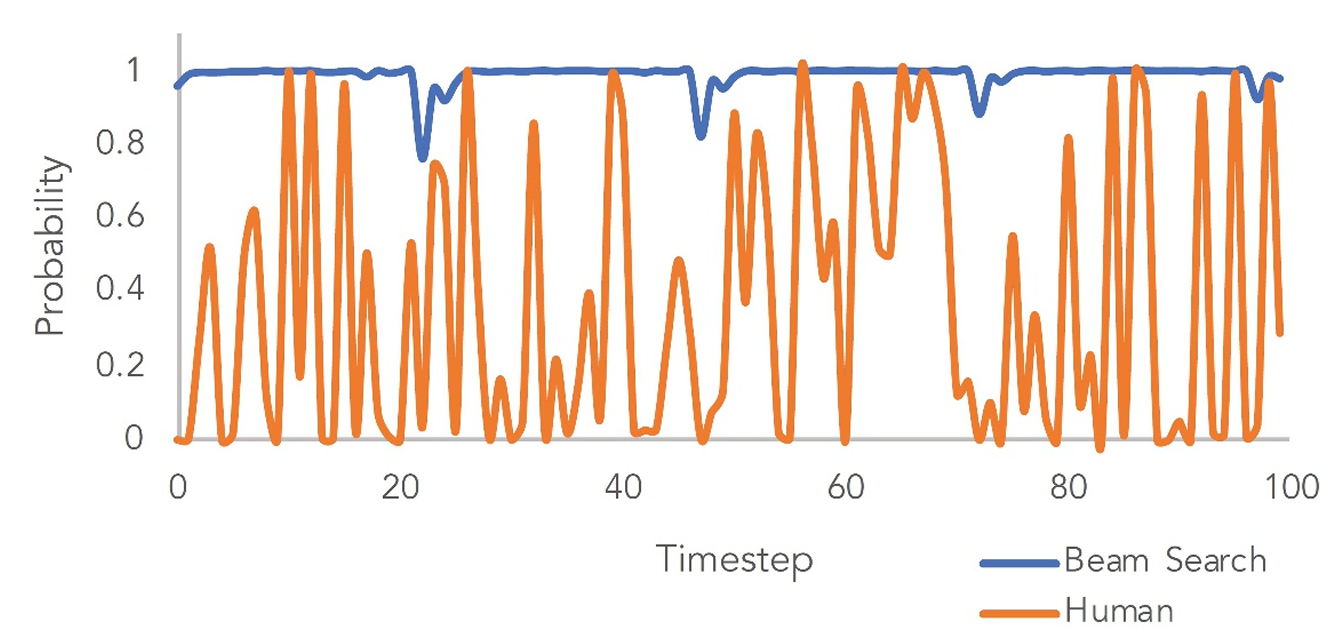


Рисунок 5.3. Сравнение вероятностей токенов для поиска по лучу и человеческого текста для данной обученной языковой модели трансформера и данной начальной входной последовательности, показывающее, что человеческая последовательность имеет гораздо более низкие вероятности токенов. [По Holtzman et al. (2019) с разрешения.]

Как видно на Рисунке 5.3, текст, сгенерированный человеком, может иметь более низкую вероятность и, следовательно, быть более неожиданным для данной модели, чем автоматически сгенерированный текст.

Вместо того чтобы пытаться найти последовательность с наивысшей вероятностью, мы можем генерировать последующие токены, просто **выбирая семпл (sampling)** из распределения softmax на каждом шаге. Однако это может привести к бессмысленным последовательностям. Это происходит из-за typically очень большого размера словаря токенов, в котором существует «длинный хвост» множества состояний токенов, каждое из которых имеет очень малую вероятность, но в совокупности составляющих значительную часть общей вероятностной массы. Это приводит к проблеме, когда существует значительная вероятность того, что система сделает плохой выбор для следующего токена.

В качестве баланса между этими крайностями мы можем рассматривать только состояния, имеющие **K наивысших вероятностей (top-K probabilities)**, для некоторого выбора K, а затем выбирать семпл из них в соответствии с их перенормированными вероятностями. Вариант этого подхода, называемый **семплированием по ядру (top-p sampling или nucleus sampling)**, вычисляет кумулятивную вероятность наиболее вероятных выходных значений до тех пор, пока не будет достигнут порог, а затем выбирает семпл из этого ограниченного набора состояний токенов.

Более «мягкая» версия top-K семплирования — введение параметра T, называемого **температурой (temperature)**, в определение функции softmax (Hinton, Vinyals, and Dean, 2015), так что:

(5.3)

и затем выбор следующего токена из этого модифицированного распределения. Когда T = 0, вероятностная масса концентрируется на наиболее вероятном состоянии, а все остальные состояния имеют нулевую вероятность, и, следовательно, это становится жадным выбором. При T = 1 мы получаем немодифицированное распределение softmax, а при T → ∞ распределение становится равномерным по всем состояниям. Выбирая значение в диапазоне 0 < T < 1, вероятность концентрируется towards более высоким значениям.

Одна из проблем генерации последовательностей заключается в том, что на этапе обучения модель обучается на входной последовательности, сгенерированной человеком, тогда как при генеративном запуске входная последовательность сама генерируется моделью. Это означает, что модель может **отходить (drift away)** от распределения последовательностей, виденных во время обучения.

III. Трансформеры-кодировщики

Далее мы рассмотрим языковые модели на основе трансформеров-кодировщиков. Это модели, которые принимают на вход последовательности и выдают на выходе векторы фиксированной длины, такие как метки классов. Примером такой модели является **BERT** (Bidirectional Encoder Representations from Transformers — двунаправленные представления кодировщика из трансформеров) (Devlin et al., 2018). Цель состоит в том, чтобы предварительно обучить языковую модель на большом корпусе текста, а затем дообучить (fine-tune) модель с помощью трансферного обучения для широкого спектра последующих (downstream) задач, каждая из которых требует меньшего набора данных, специфичного для приложения. Архитектура трансформера-кодировщика проиллюстрирована на Рисунке 5.4. Этот подход представляет собой прямое применение ранее обсуждавшихся слоев-трансформеров.

Первый токен каждой входной строки задается специальным токеном <class>, и соответствующий выход модели игнорируется на этапе предварительного обучения. Его роль станет ясной, когда мы обсудим дообучение. Модель предварительно обучается путем подачи последовательностей токенов на вход. Случайно выбранное подмножество токенов, скажем 15%, заменяется специальным токеном <mask>. Модель обучается предсказывать отсутствующие токены в соответствующих выходных узлах. Это аналогично маскированию, используемому в word2vec для изучения векторных представлений слов. Например, входная последовательность может быть:

Я <mask> через реку, чтобы добраться до <mask> берега.

и сеть должна предсказать «переплыл» на выходном узле 2 и «другого» на выходном узле 10. В этом случае только два выхода вносят вклад в функцию ошибки, а остальные выходы игнорируются.

Термин «двунаправленный» относится к тому факту, что сеть видит слова как до, так и после маскированного слова и может использовать оба источника информации для предсказания. Как следствие, в отличие от моделей-декодеров, нет необходимости смещать входы вправо на одну позицию и нет необходимости маскировать выходы каждого слоя, чтобы они не «видели» входные токены, встречающиеся позже в последовательности. По сравнению с моделью-декодером, кодировщик менее эффективен, поскольку только часть токенов последовательности используется в качестве меток для обучения. Более того, модель-кодировщик не способна генерировать последовательности.

Процедура замены случайно выбранных токенов на <mask> означает, что обучающий набор не совсем соответствует последующим наборам для дообучения, поскольку последние не будут содержать токенов <mask>. Чтобы смягчить возможные проблемы, Devlin et al. (2018) немного изменили процедуру: из 15% случайно выбранных токенов 80% заменяются на <mask>, 10% заменяются на слово, выбранное случайным образом из словаря, и в 10% случаев исходные слова остаются на входе, но они все равно должны быть правильно предсказаны на выходе.

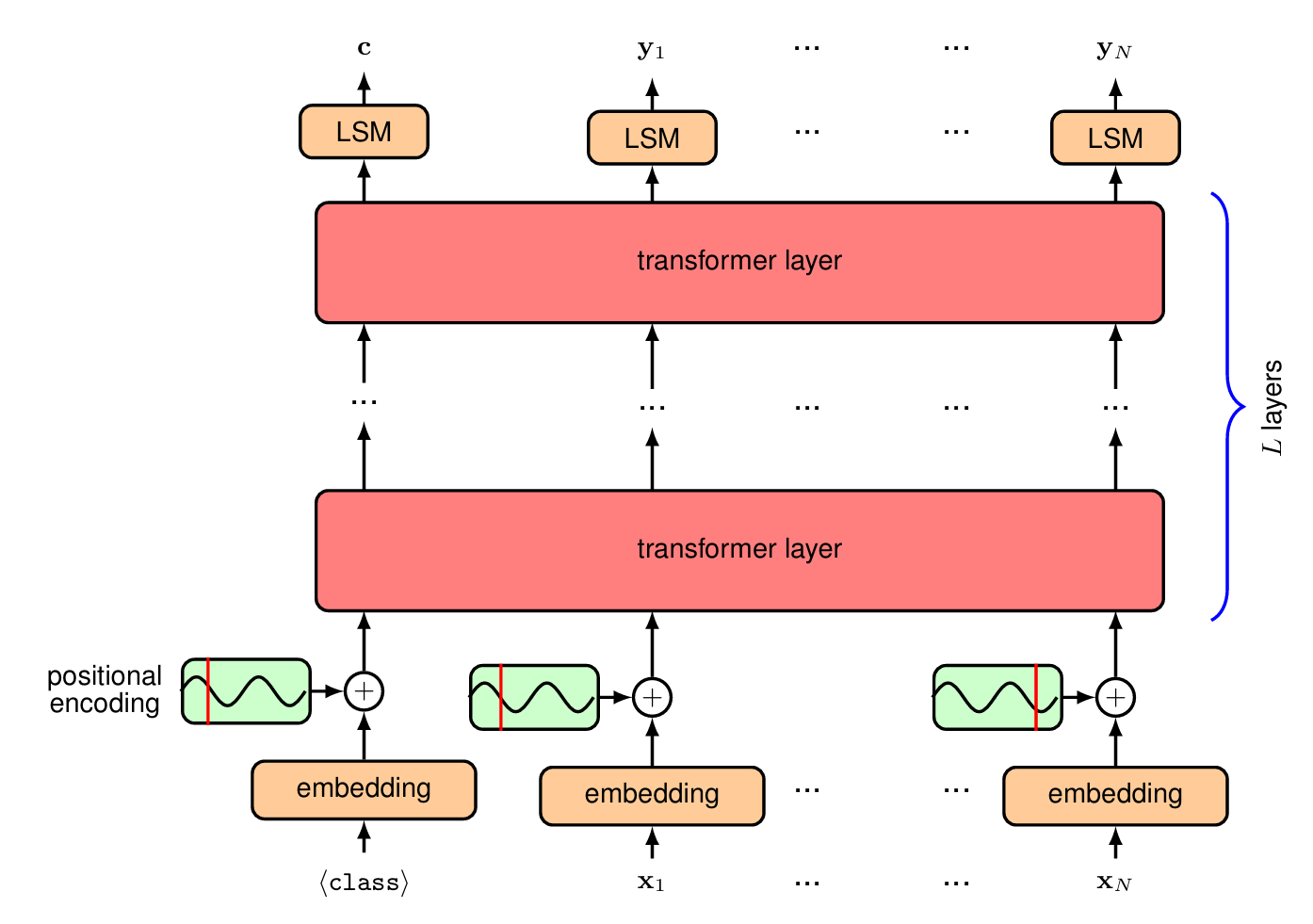


Рисунок 5.4. Архитектура модели трансформера-кодировщика. Блоки с пометкой «LSM» обозначают линейное преобразование, чьи обучаемые параметры являются общими для всех позиций токенов, с последующей функцией активации softmax. Основные различия по сравнению с моделью-декодером заключаются в том, что входная последовательность не смещается вправо, и матрица маскирования «заглядывания вперед» опущена, и, следовательно, в каждом слое самовнимания каждый выходной токен может «обращать внимание» на любой из входных токенов.

После обучения модели-кодировщика ее можно дообучить для решения различных задач. Для этого создается новый выходной слой, форма которого специфична для решаемой задачи. Для задачи классификации текста используется только первая выходная позиция, которая соответствует токену <class>, всегда находящемуся в первой позиции входной последовательности. Если этот выход имеет размерность D, то к первому выходному узлу добавляется матрица параметров размерности D × K, где K — количество классов, и это, в свою очередь, подается на K-мерную функцию softmax или, для K=2, на вектор размерности D × 1 с последующей логистической сигмоидой. Линейное выходное преобразование может быть alternatively заменено на более сложную дифференцируемую модель, такую как MLP (многослойный перцептрон). Если цель — классифицировать каждый токен входной строки, например, присвоить каждому токену категорию (такую как человек, место, цвет и т.д.), то первый выход игнорируется, а последующие выходы имеют общий слой «линейный + softmax». Во время дообучения все параметры модели, включая новую выходную матрицу, изучаются с помощью стохастического градиентного спуска с использованием логарифма вероятности правильного label. Альтернативно, выход предварительно обученной модели может подаваться на вход сложной генеративной модели глубокого обучения для таких приложений, как синтез изображений по тексту.

**IV. Трансформеры типа «последовательность-последовательность»**

Для полноты картины мы кратко обсудим третью категорию моделей-трансформеров, которая объединяет кодировщик и декодер, как обсуждалось в оригинальной статье о трансформерах Vaswani et al. (2017). Рассмотрим задачу перевода предложения с английского языка на голландский. Мы можем использовать модель-декодер для генерации последовательности токенов, соответствующей голландскому выходу, токен за токеном, как обсуждалось ранее. Основное различие заключается в том, что этот вывод должен быть обусловлен всей входной последовательностью, соответствующей английскому предложению. Трансформер-кодировщик может быть использован для отображения входной последовательности токенов в подходящее внутреннее представление, которое мы обозначим через Z. Чтобы включить Z в генеративный процесс для выходной последовательности, мы используем модифицированную форму механизма внимания, называемую **перекрёстным вниманием (cross-attention)**. Оно такое же, как и самовнимание, за исключением того, что хотя векторы запроса (query) поступают из генерируемой последовательности (в данном случае голландской выходной последовательности), векторы ключа (key) и значения (value) поступают из последовательности, представленной Z, как показано на Рисунке 12.19. Возвращаясь к нашей аналогии с сервисом потокового видео, это было бы похоже на то, как если бы пользователь отправлял свой вектор запроса в другую стриминговую компанию, которая затем сравнивает его с собственным набором векторов ключей, чтобы найти наилучшее соответствие, и возвращает связанный вектор значения в виде фильма.

Когда мы объединяем модули кодировщика и декодера, мы получаем архитектуру модели, показанную на Рисунке 5.6. Модель можно обучать с использованием парных входных и выходных предложений.

V. Большие языковые модели

Важнейшим recent достижением в области машинного обучения стало создание очень больших нейронных сетей на основе трансформеров для обработки естественного языка, известных как **большие языковые модели (Large Language Models, LLMs)**. Здесь «большие» относится к количеству параметров (весов и смещений) в сети, которое на момент написания может достигать около триллиона (10¹²). Обучение таких моделей требует больших затрат, а мотивация для их создания исходит из их исключительных возможностей.

Помимо доступности больших наборов данных, обучение все более крупных моделей стало возможным благодаря появлению аппаратного обеспечения для массового параллельного обучения на основе **GPU (графических процессоров)** и аналогичных процессоров, тесно связанных в большие кластеры, оснащенные высокоскоростными соединениями и большим объемом памяти. Архитектура трансформера сыграла ключевую роль в разработке этих моделей, поскольку она способна очень эффективно использовать такое оборудование. Зачастую увеличение размера набора обучающих данных наряду с соразмерным увеличением количества параметров модели приводит к улучшению производительности, которое опережает улучшения архитектуры или другие способы включения более глубоких знаний предметной области (Sutton, 2019; Kaplan et al., 2020). Например, впечатляющий рост производительности моделей серии **GPT** (Radford et al., 2019; Brown et al., 2020; OpenAI, 2023) от поколения к поколению в основном обусловлен увеличением масштаба.

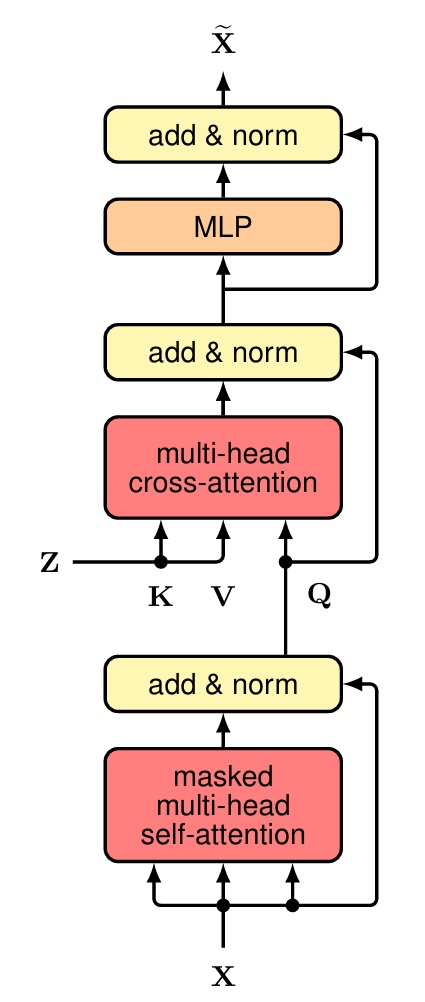


Рисунок 5.5. Схематическое изображение одного слоя перекрёстного внимания, используемого в секции декодера трансформера типа «последовательность-последовательность». Здесь Z обозначает выход из секции кодировщика. Z определяет векторы ключа и значения для слоя перекрёстного внимания, тогда как векторы запроса определяются внутри секции декодера.

Такие улучшения производительности привели к появлению нового рода **закона Мура**, согласно которому количество вычислительных операций, необходимых для обучения современной модели машинного обучения, росло экспоненциально примерно с 2012 года с периодом удвоения около 3,4 месяцев.

Ранние языковые модели обучались с использованием **обучения с учителем (supervised learning)**. Например, для создания системы перевода обучающий набор состоял бы из пар предложений на двух языках. Однако основное ограничение обучения с учителем заключается в том, что данные обычно должны быть обработаны людьми для предоставления размеченных примеров, что сильно ограничивает доступный объем данных, тем самым требуя активного использования индуктивных смещений, таких как feature engineering и архитектурные ограничения, для достижения разумной производительности.

Вместо этого большие языковые модели обучаются методом **самоконтролируемого обучения (self-supervised learning)** на очень больших наборах текстовых данных, а также, потенциально, других последовательностей токенов, таких как компьютерный код. Мы видели, как трансформер-декодер можно обучать на последовательностях токенов, где каждый токен выступает в качестве размеченного целевого примера, а предшествующая последовательность — в качестве входа, чтобы изучить условное распределение вероятностей. Это «саморазмечивание» значительно расширяет объем доступных обучающих данных и, следовательно, позволяет использовать глубокие нейронные сети с большим количеством параметров.

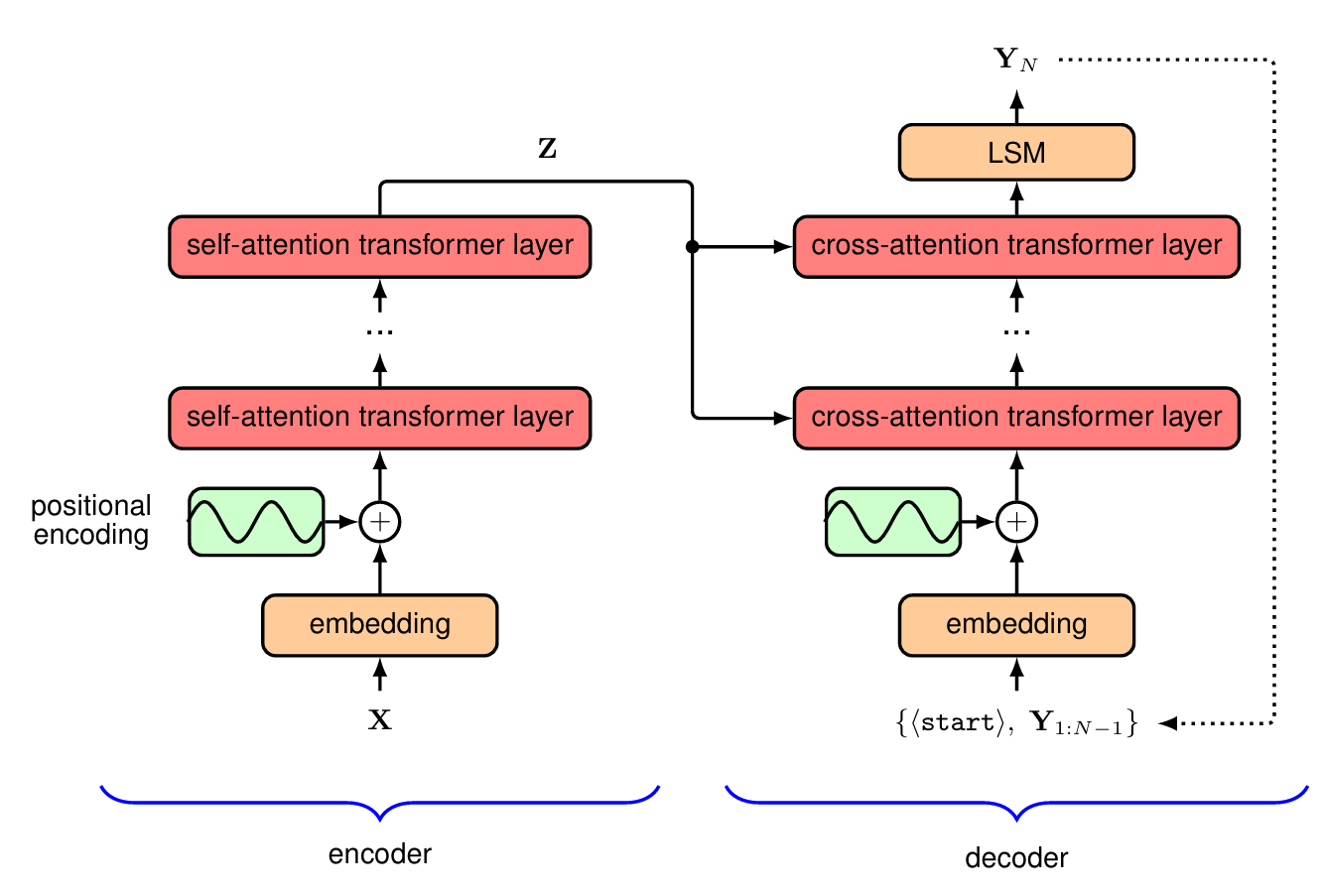
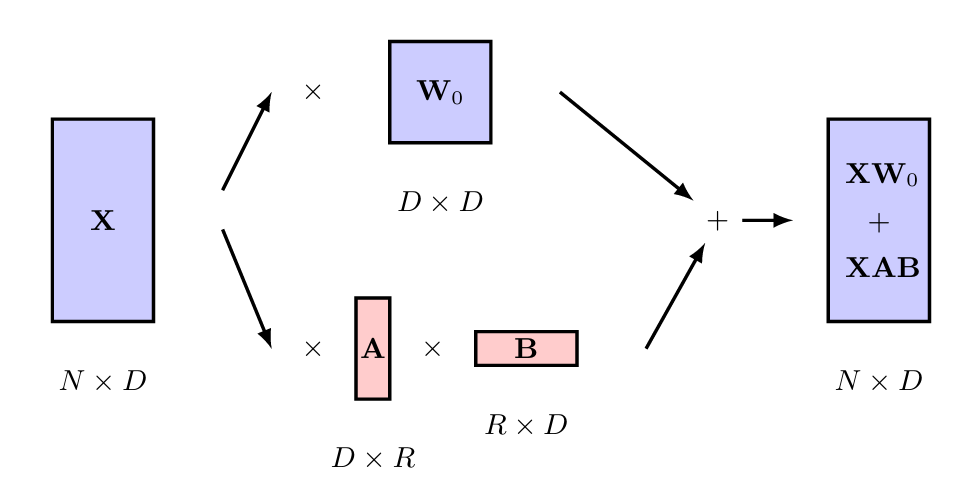


Рисунок 5.6. Схематическое изображение трансформера типа «последовательность-последовательность». Для упрощения диаграммы входные токены показаны collectively как один блок, и аналогично для выходных токенов. Векторы позиционного кодирования добавляются к входным токенам как для секции кодировщика, так и для декодера. Каждый слой в кодировщике соответствует структуре, показанной на Рисунке 5.1, и каждый слой перекрёстного внимания имеет форму, показанную на Рисунке 5.5.

Это использование самоконтролируемого обучения привело к **смене парадигмы (paradigm shift)**, при которой большая модель сначала **предварительно обучается (pre-trained)** на неразмеченных данных, а затем **дообучается (fine-tuned)** с использованием обучения с учителем на основе гораздо меньшего набора размеченных данных. Это, по сути, форма **трансферного обучения (transfer learning)**, и одна и та же предварительно обученная модель может использоваться для множества последующих (downstream) приложений. Модель с широкими возможностями, которая впоследствии может быть дообучена для конкретных задач, называется **базовой моделью (foundation model)** (Bommasani et al., 2021).

Дообучение может выполняться путем добавления дополнительных слоев к выходам сети или путем замены последних нескольких слоев новыми параметрами с последующим использованием размеченных данных для обучения этих финальных слоев. На этапе дообучения веса и смещения в основной модели могут либо оставаться неизменными, либо им может быть позволено незначительно адаптироваться. Обычно стоимость дообучения мала по сравнению со стоимостью предварительного обучения.

Очень эффективный подход к дообучению называется **низкоранговой адаптацией (Low-Rank Adaptation, LoRA)** (Hu et al., 2021). Этот подход основан на результатах, показывающих, что обученная сверхпараметризованная модель имеет низкую **внутреннюю размерность (intrinsic dimensionality)** относительно дообучения, meaning что изменения параметров модели во время дообучения лежат на **многообразии (manifold)**.



[Рисунок 5.7. Схематическое изображение низкоранговой адаптации, показывающее матрицу весов из одного из слоев внимания в предварительно обученном трансформере. Дополнительные веса, заданные матрицами A и B, адаптируются во время дообучения, и их произведение AB затем добавляется к исходной матрице для последующего вывода.]

размерность которого намного меньше общего количества обучаемых параметров в модели (Aghajanyan, Zettlemoyer, and Gupta, 2020). LoRA использует это, **замораживая (freezing)** веса исходной модели и добавляя дополнительные обучаемые матрицы весов в каждый слой трансформера в форме **низкоранговых произведений (low-rank products)**. Обычно модифицируются только веса слоев внимания, тогда как веса MLP-слоев остаются фиксированными. Рассмотрим матрицу весов W0 размерностью D × D, которая может представлять матрицу запроса, ключа или значения, где матрицы из нескольких голов внимания рассматриваются вместе как одна матрица. Мы вводим параллельный набор весов, определяемый произведением двух матриц A и B с размерами D×R и R×D соответственно, как схематично показано на Рисунке 12.21. Этот слой затем генерирует выход, задаваемый как . Количество параметров в дополнительной матрице весов AB равно 2RD по сравнению с D² параметрами в исходной матрице весов W0, и поэтому, если R ≪ D, то количество параметров, которые необходимо адаптировать во время дообучения, намного меньше, чем количество параметров в исходном трансформере. На практике это может сократить количество параметров, требующих обучения, в 10 000 раз. После завершения дообучения дополнительные веса могут быть добавлены к исходным матрицам весов, чтобы получить новую матрицу весов:

так что во время **вывода (inference)** не возникает дополнительных вычислительных затрат по сравнению с запуском исходной модели, поскольку обновленная модель имеет тот же размер, что и исходная.

Поскольку языковые модели стали больше и мощнее, необходимость в дообучении уменьшилась, и теперь генеративные языковые модели способны решать широкий спектр задач просто посредством текстового взаимодействия. Например, если текстовая строка:

Английский: the cat sat on the mat. Французский:

подана как входная последовательность, авторегрессионная языковая модель может продолжить генерировать последующие токены до тех пор, пока не будет сгенерирован токен <stop>, причем вновь сгенерированные токены представляют собой французский перевод. Заметьте, что модель не обучалась специально для перевода, но научилась это делать в результате обучения на огромном корпусе данных, включающем несколько языков.

Пользователь может взаимодействовать с такими моделями с помощью диалога на естественном языке, что делает их очень доступными для широкой аудитории. Чтобы улучшить пользовательский опыт и качество генерируемых выходных данных, были разработаны методы дообучения больших языковых моделей на основе человеческой оценки сгенерированного вывода, с использованием методов, таких как **обучение с подкреплением на основе человеческой обратной связи (Reinforcement Learning from Human Feedback, RLHF)** (Christiano et al., 2017). Такие техники помогли создать большие языковые модели с впечатляюще удобными интерфейсами для общения, наиболее известной из которых является система от OpenAI под названием **ChatGPT**.

Последовательность входных токенов, задаваемую пользователем, называют **промптом (prompt)**. Например, он может состоять из начальных слов рассказа, который модель должна завершить. Или он может содержать вопрос, на который модель должна дать ответ. Используя разные промпты, одна и та же обученная нейронная сеть может решать широкий спектр задач, таких как генерация компьютерного кода по простому текстовому запросу или написание рифмованных стихов по требованию. Производительность модели теперь зависит от формы промпта, что привело к возникновению новой области — **инженерии промптов (prompt engineering)** (Liu et al., 2021), которая aims разработать хорошую форму промпта, приводящую к высококачественному результату для последующей задачи. Поведение модели также можно изменить, адаптировав промпт пользователя перед подачей в языковую модель, добавив в начало дополнительную последовательность токенов, называемую **префикс-промптом (prefix prompt)**, чтобы изменить форму вывода. Например, пре-промпт может состоять из инструкций, выраженных на стандартном английском, предписывающих сети не включать оскорбительные выражения в свой вывод.

Это позволяет модели решать новые задачи, просто предоставляя несколько примеров в рамках промпта, без необходимости адаптации параметров модели. Это пример **обучения с малым количеством примеров (few-shot learning)**.

Современные передовые модели, такие как **GPT-4**, стали настолько мощными, что демонстрируют remarkable свойства, которые были описаны как первые признаки **искусственного общего интеллекта (Artificial General Intelligence, AGI)** (Bubeck et al., 2023), и стимулируют новую волну технологических инноваций. Более того, возможности этих моделей продолжают улучшаться впечатляющими темпами.

## §6. Мультимодальные трансформеры

Хотя изначально трансформеры были разработаны как альтернатива рекуррентным сетям для обработки последовательных языковых данных, они получили распространение почти во всех областях глубокого обучения. Они доказали, что являются моделями общего назначения, поскольку делают очень мало предположений о входных данных, в отличие, например, от сверточных сетей, которые строятся на сильных предположениях об эквивариантности и локальности.

Благодаря своей универсальности трансформеры стали передовой технологией для многих различных модальностей, включая текст, изображения, видео, облака точек и аудиоданные, и используются как для дискриминативных, так и для генеративных приложений в каждой из этих областей. Базовая архитектура слоя трансформера оставалась относительно постоянной как с течением времени, так и в разных приложениях. Следовательно, ключевые инновации, позволившие использовать трансформеры в областях, отличных от обработки естественного языка, в основном были сосредоточены на представлении и кодировании входных и выходных данных.

Большое преимущество единой архитектуры, способной обрабатывать многие типы данных, заключается в том, что она делает мультимодальные вычисления относительно простыми. В этом контексте **мультимодальность** относится к приложениям, которые объединяют два или более различных типа данных — на входе, на выходе или и там и там. Например, мы можем захотеть сгенерировать изображение по текстовому промпту или разработать робота, который может комбинировать информацию с нескольких датчиков, таких как камеры, радар и микрофоны. Важно отметить, что если мы можем токенизировать входы и декодировать выходные токены, то, скорее всего, мы можем использовать трансформер.

I. Визуальные трансформеры (Vision Transformers)

Трансформеры с большим успехом применялись в компьютерном зрении и достигли передовой производительности во многих задачах. Наиболее распространенный выбор для дискриминативных задач — это стандартный трансформер-кодировщик, и этот подход в области зрения известен как **Vision Transformer (ViT)** (Dosovitskiy et al., 2020).

При использовании трансформера нам нужно решить, как преобразовать входное изображение в токены. Самый простой выбор — использовать каждый пиксель в качестве токена после линейной проекции. Однако объем памяти, требуемый стандартной реализацией трансформера, растет квадратично с количеством входных токенов, поэтому такой подход обычно неосуществим. Вместо этого наиболее распространенный подход к токенизации — разбить изображение на набор **патчей (patches)** одинакового размера. Предположим, изображения имеют размерность  где H и W — высота и ширина изображения в пикселях, а C — количество каналов (где обычно C = 3 для цветов R, G, B). Каждое изображение разбивается на неперекрывающиеся патчи размером P × P (где P = 16 — распространенный выбор), а затем «разворачивается» в одномерный вектор, что дает представление , где  — общее количество патчей для одного изображения. Архитектура ViT показана на Рисунке 6.1.

Другой подход к токенизации — пропустить изображение через небольшую **сверточную нейронную сеть (CNN)**. Это может уменьшить разрешение изображения до управляемого количества токенов, каждый из которых представлен одним из выходов сети. Например, типичная архитектура кодировщика ResNet18 уменьшает разрешение изображения в 8 раз по обоим измерениям (высоте и ширине), давая в 64 раза меньше токенов, чем пикселей.

Нам также нужен способ кодирования **позиционной информации** в токенах. Можно создать явные **позиционные эмбеддинги (positional embeddings)**, кодирующие двумерную позиционную информацию патчей изображения, но на практике это обычно не улучшает производительность, поэтому чаще всего используются просто обучаемые позиционные эмбеддинги. В отличие от трансформеров, используемых для естественного языка, визуальные трансформеры обычно принимают фиксированное количество токенов на входе, что позволяет избежать проблемы неспособности обучаемых позиционных кодировок обобщаться на входы другого размера.

Визуальный трансформер имеет совершенно иную архитектуру по сравнению с CNN. Хотя сильные индуктивные смещения заложены в модель CNN, единственная двумерная индуктивная смесь в визуальном трансформере обусловлена патчами, используемыми для токенизации входа. Следовательно, трансформеру обычно требуется больше обучающих данных, чем сравнимой CNN, поскольку он должен изучать геометрические свойства изображений с нуля. Однако, поскольку нет строгих предположений о структуре входных данных, трансформеры часто способны достичь более высокой точности. Это служит еще одной иллюстрацией компромисса между индуктивным смещением и масштабом обучающих данных (Sutton, 2019).

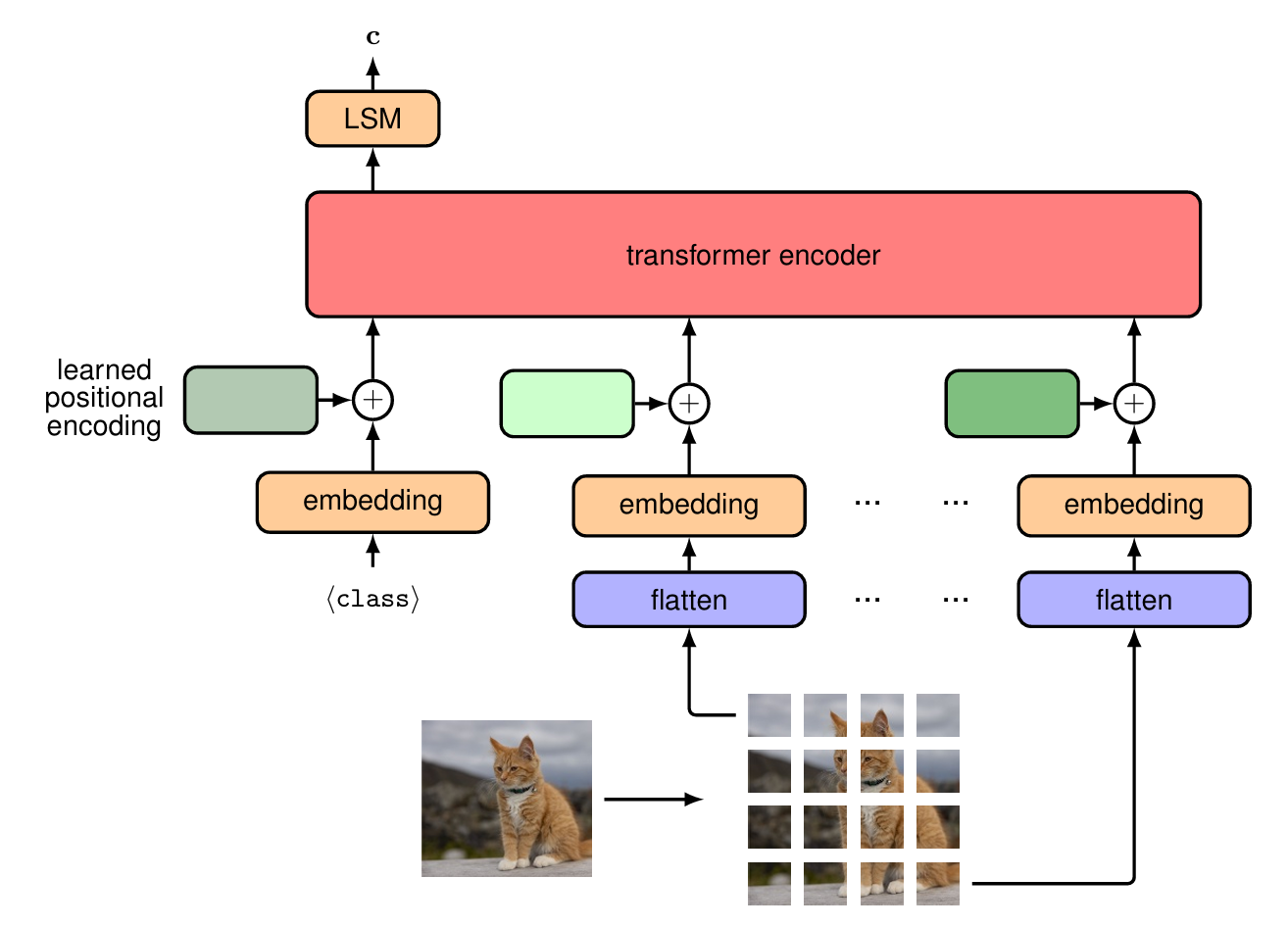


Рисунок 6.1. Иллюстрация архитектуры визуального трансформера для задачи классификации. Здесь обучаемый токен <class> включен как дополнительный вход, и связанный с ним выход преобразуется линейным слоем с функцией активации softmax (обозначен как LSM) для получения итогового выходного вектора классов c.

II. Генеративные трансформеры для изображений

В языковой сфере наиболее впечатляющих результатов удалось достичь, когда трансформеры стали использовать как авторегрессионные генеративные модели для синтеза текста. Естественно спросить, можно ли также использовать трансформеры для синтеза реалистичных изображений. Поскольку естественный язык по своей природе последователен, он аккуратно вписывается в авторегрессионную модель, тогда как у изображений нет естественного порядка следования пикселей, поэтому не так интуитивно очевидно, что их авторегрессионное декодирование будет полезным.

Однако любое распределение можно разложить в произведение условных распределений, при условии, что мы сначала определим некоторый порядок переменных. Таким образом, совместное распределение по упорядоченным переменным  можно записать:

(6.1)

Эта факторизация совершенно общая и не накладывает ограничений на форму отдельных условных распределений .

Для изображения мы можем выбрать  для представления n-го пикселя в виде трехмерного вектора значений RGB. Теперь нам нужно определиться с порядком пикселей, и один широко используемый вариант называется **растровым сканированием (raster scan)**, как показано на Рисунке 6.2. Схематическая иллюстрация генерации изображения с помощью авторегрессионной модели на основе порядка растрового сканирования показана на Рисунке 6.3.

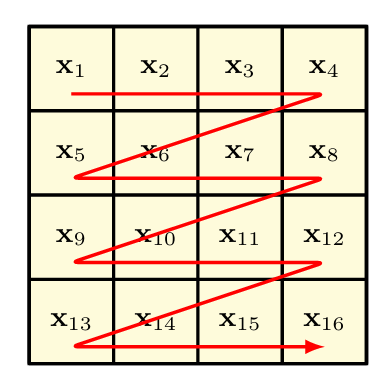


Рисунок 6.2. Иллюстрация растрового сканирования, определяющего конкретный линейный порядок пикселей в двумерном изображении.

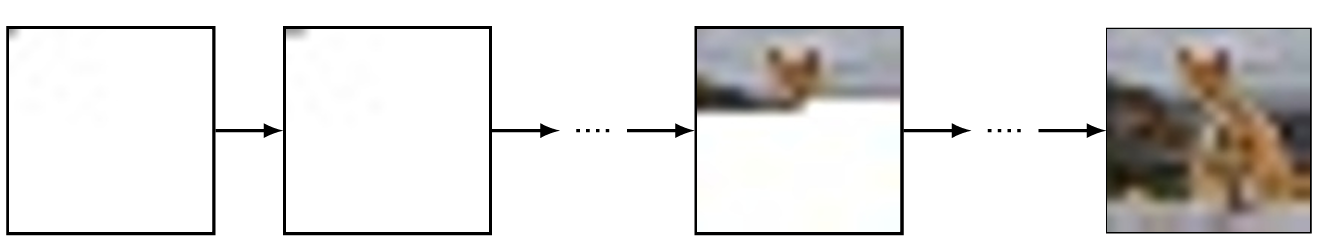


Рисунок 6.3. Иллюстрация того, как изображение может быть сэмплировано из авторегрессионной модели. Первый пиксель сэмплируется из маргинального распределения второй пиксель — из условного распределения и так далее в порядке растрового сканирования, пока не получится полное изображение.

Отметим, что использование авторегрессионных генеративных моделей для изображений предшествовало появлению трансформеров. Например, **PixelCNN** (Oord et al., 2016) и **PixelRNN** (Oord, Kalchbrenner, and Kavukcuoglu, 2016) использовали специальные маскированные сверточные слои, которые сохраняют условную независимость, определенную для каждого пикселя соответствующим членом в правой части (6.2).

Представления изображения с использованием непрерывных значений могут хорошо работать в дискриминативных задачах. Однако для генерации изображений гораздо лучшие результаты получаются при использовании **дискретных представлений**. Непрерывные условные распределения, обученные методом максимального правдоподобия, такие как гауссовы, для которых функция отрицательного логарифма правдоподобия является суммой квадратов ошибок, изучать средние значения по обучающим данным, что приводит к **размытым изображениям**. И наоборот, дискретные распределения с легкостью справляются с **модальностью**. Например, одно из условных распределений  в (6.2) может научиться, что пиксель может быть либо черным, либо белым, тогда как регрессионная модель может научиться, что пиксель должен быть серым.

Однако работа с дискретными пространствами также имеет свои сложности. Значения R, G и B пикселей изображения обычно представлены с точностью не менее 8 бит, так что каждый пиксель имеет возможных значений. Изучение условного softmax-распределения в таком высокомерном пространстве невыполнимо.

Один из способов решения проблемы высокой размерности — использование техники **векторного квантования (vector quantization)**, которую можно рассматривать как форму сжатия данных. Предположим, у нас есть набор векторов данных , каждый размерности D, которые могут, например, представлять пиксели изображения, и затем мы вводим набор из K **кодовых векторов (codebook vectors)** , также размерности D, где обычно . Теперь мы аппроксимируем каждый вектор данных его ближайшим кодомвым вектором в соответствии с некоторой метрикой подобия, обычно евклидовым расстоянием, так что .

Поскольку существует K кодовых векторов, мы можем представить каждый  с помощью one-hot закодированного K-мерного вектора, и поскольку мы можем выбирать значение K, мы можем контролировать компромисс между более точным представлением данных (за счет использования большего значения K) или большим сжатием (за счет использования меньшего значения K).

Следовательно, мы можем взять исходные пиксели изображения и отобразить их в пространство кодов меньшей размерности. Затем можно обучить авторегрессионный трансформер генерировать последовательность кодовых векторов, и эту последовательность можно преобразовать обратно в исходное пространство изображений, заменив каждый индекс кода  соответствующим D-мерным кодовым вектором .

Авторегрессионные трансформеры впервые были применены к изображениям в **ImageGPT** (Chen, Radford, et al., 2020). Здесь каждый пиксель рассматривался как один из дискретного набора трехмерных цветовых кодовых векторов, каждый из которых соответствует кластеру в K-means кластеризации цветового пространства. One-hot кодирование, следовательно, дает дискретные токены, аналогичные языковым токенам, и позволяет обучать трансформер так же, как и языковые модели, с целью классификации следующего токена. Это мощная цель для обучения представлениям для последующего дообучения, опять же схожим образом с языковым моделированием.

Однако использование отдельных пикселей в качестве токенов напрямую может привести к высоким вычислительным затратам, поскольку требуется прямой проход для каждого пикселя, что означает, что и обучение, и вывод плохо масштабируются с разрешением изображения. Кроме того, использование отдельных пикселей в качестве входных данных означает, что приходится использовать изображения низкого разрешения, чтобы обеспечить разумную длину контекста при декодировании пикселей позже в растровом сканировании. Как мы видели с моделью ViT, предпочтительнее использовать **патчи изображения** в качестве токенов вместо пикселей, так как это может привести к dramatically меньшему количеству токенов и, следовательно, облегчает работу с изображениями более высокого разрешения. Как и прежде, нам нужно работать с дискретным пространством значений токенов из-за потенциальной multimodality условных распределений. Это снова поднимает проблему размерности, которая теперь гораздо серьезнее для патчей, чем для отдельных пикселей, поскольку размерность растет экспоненциально с количеством пикселей в патче. Например, даже всего с двумя возможными значениями токенов пикселей, черным и белым, и патчами размером 16×16, мы имели бы словарь токенов патчей размером .

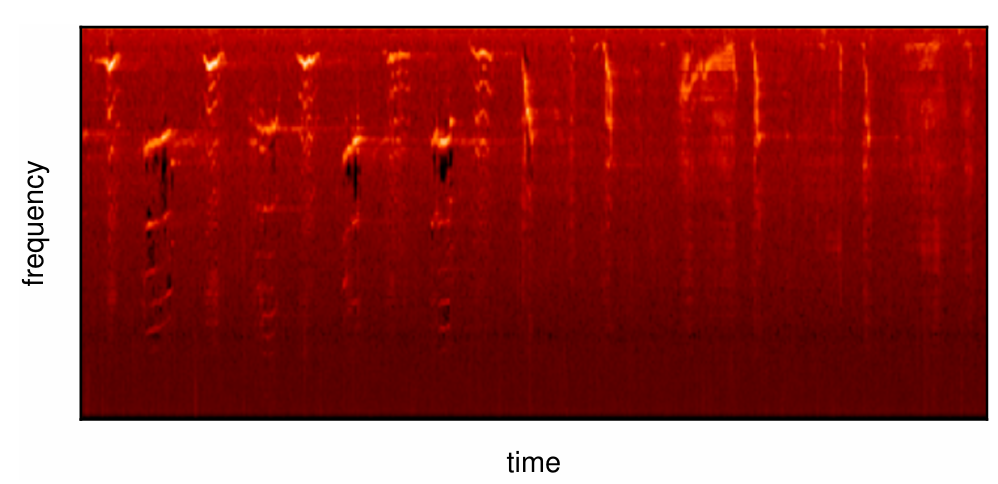


Рисунок 6.4. Пример мел-спектрограммы песни горбатого кита. [Исходные данные защищены авторским правом ©2013-2023, команда разработчиков librosa.]

И снова мы обращаемся к **векторному квантованию**, чтобы решить проблему размерности. Кодовые векторы могут быть изучены из набора данных патчей изображений с использованием простых алгоритмов кластеризации, таких как K-средних, или с помощью более сложных методов, таких как **полносверточные сети (fully convolutional networks)** (Oord, Vinyals, and Kavukcuoglu, 2017; Esser, Rombach, and Ommer, 2020) или даже визуальные трансформеры (Yu et al., 2021).

Одна проблема при обучении отображению каждого патча в дискретный набор кодов и обратно заключается в том, что операция векторного квантования **недифференцируема**. К счастью, мы можем использовать технику под названием **оценка градиента по прямому проходу (straight-through gradient estimation)** (Bengio, Léonard, and Courville, 2013), которая представляет собой простое приближение, которое просто копирует градиенты через недифференцируемую функцию в процессе обратного распространения ошибки.

Использование авторегрессионных трансформеров для генерации изображений можно расширить на **видео**, рассматривая видео как одну длинную последовательность таких векторно-квантованных токенов (Rakhimov et al., 2020; Yan et al., 2021; Hu et al., 2023).

****III Аудиоданные****

Далее мы рассмотрим применение трансформеров к аудиоданным. Звук обычно хранится в виде **звуковой волны (waveform)**, полученной путем измерения амплитуды давления воздуха через регулярные промежутки времени. Хотя эту волну можно использовать напрямую на входе модели глубокого обучения, на практике эффективнее предварительно обработать ее в **мел-спектрограмму (mel spectrogram)**. Это матрица, столбцы которой представляют шаги по времени, а строки соответствуют частотам. Частотные полосы следуют стандартному соглашению, выбранному на основе субъективной оценки, чтобы давать равные перцептивные различия между последовательными частотами (слово «мел» происходит от мелодии). Пример мел-спектрограммы показан на Рисунке 6.4.

Одно из применений трансформеров в аудио-области — **классификация**, где сегменты аудио назначаются одной из нескольких предопределенных категорий. Например, набор данных **AudioSet** (Gemmeke et al., 2017) — широко используемый бенчмарк. Он содержит такие классы, как «автомобиль», «животное» и «смех». До появления трансформера передовым подходом для классификации аудио было использование мел-спектрограмм, рассматриваемых как изображения и подаваемых на вход сверточной нейронной сети (CNN). Однако, хотя CNN хорошо понимает локальные отношения, один ее недостаток заключается в трудностях с **долгосрочными зависимостями (longer-range dependencies)**, которые могут быть важны при обработке аудио.

Так же, как трансформеры заменили RNN в качестве передовой технологии в обработке естественного языка, они также пришли на смену CNN для таких задач, как классификация аудио. Например, модель-трансформер-кодировщик идентичной структуры, используемой как для языка, так и для зрения, как показано на Рисунке 6.5, может быть использована для предсказания класса аудиовходов (Gong, Chung, and Glass, 2021). Здесь мел-спектрограмма рассматривается как изображение, которое затем токенизируется. Это делается путем разделения изображения на патчи способом, похожим на визуальные трансформеры, возможно, с некоторым перекрытием, чтобы не потерять важные соседние отношения. Каждый патч затем «уплощается» (flattened), то есть преобразуется в одномерный массив, в данном случае длиной 256. Затем к каждому токену добавляется уникальное **позиционное кодирование (positional encoding)**, добавляется специальный токен <class>, и токены пропускаются через трансформер-кодировщик. Выходной токен, соответствующий входному токену <class> из последнего слоя трансформера, затем может быть декодирован с помощью линейного слоя и функции активации softmax, и вся модель может быть обучена **end-to-end (от начала до конца)** с использованием **перекрестной энтропии (cross-entropy loss)**.

IV. Преобразование текста в речь

Классификация — не единственная задача, которую глубокое обучение, и, в частности, архитектура трансформера, революционизировала в аудио-области. Успех трансформеров в синтезе речи, имитирующей голос заданного говорящего, является еще одной демонстрацией их универсальности, и их применение к этой задаче — поучительный пример того, как применять трансформеры в новом контексте.

Генерация речи, соответствующей заданному текстовому отрывку, известна как **синтез речи (text-to-speech synthesis)**. Более традиционный подход заключался бы в сборе записей речи от заданного говорящего и обучении модели регрессии с учителем для предсказания речевого вывода, возможно, в форме мел-спектрограммы, из соответствующего транскрибированного текста. При выводе (inference) текст, для которого мы хотим синтезировать речь, подается на вход, и результирующий выход мел-спектрограммы затем может быть декодирован обратно в звуковую волну, поскольку это фиксированное отображение.

Однако у этого подхода есть несколько серьезных недостатков. Во-первых, если мы предсказываем речь на низком уровне, например, используя суб-словесные компоненты, известные как **фонемы (phonemes)**, то для плавности результирующих предложений требуется больший контекст. Однако, если мы предсказываем более длинные сегменты, то пространство возможных входов значительно растет, и для достижения хорошей обобщающей способности может потребоваться нереализуемое количество обучающих данных. Во-вторых, этот подход не переносит знания между говорящими, поэтому для каждого нового говорящего потребуется много данных. Наконец, проблема по сути является задачей **генеративного моделирования (generative modelling task)**, так как для данной пары «говорящий-текст» существует несколько правильных речевых выходов, поэтому регрессия может не подходить, поскольку она стремится усреднять целевые значения.

Если же мы будем обращаться с аудиоданными так же, как с естественным языком, и представим преобразование текста в речь как задачу **условного языкового моделирования (conditional language modelling task)**, то мы сможем обучать модель практически так же, как и большие языковые модели для текста.

Есть две основные детали реализации, которые необходимо решить. Первая — как токенизировать обучающие данные и декодировать предсказания, а вторая — как **обусловить (condition)** модель голосом говорящего.

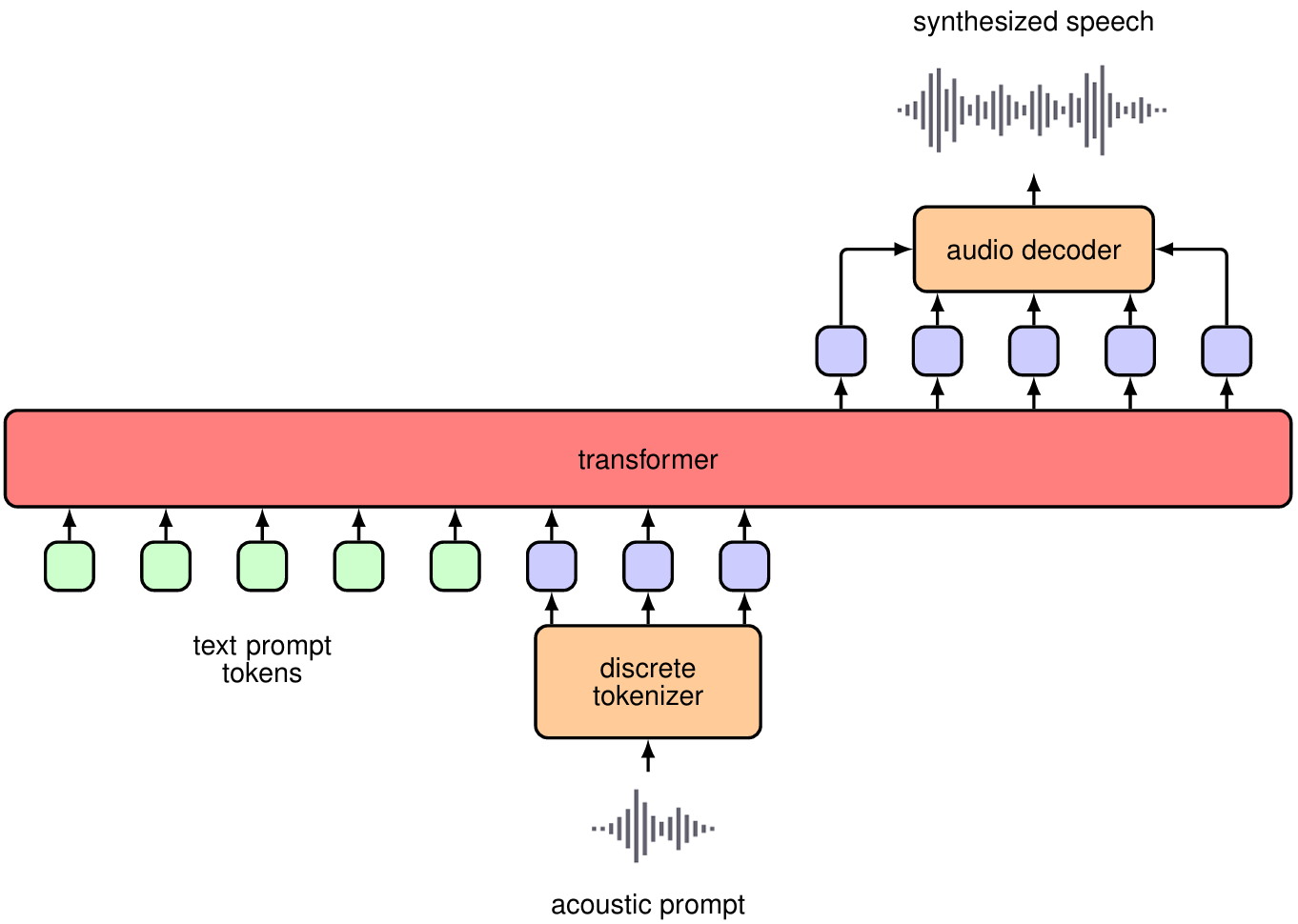


Рисунок 6.5. Диаграмма, показывающая высокоуровневую архитектуру Vall-E. Вход в модель трансформера состоит из стандартных текстовых токенов, которые указывают модели, какие слова должны содержаться в синтезированной речи, вместе с акустическими токенами-промптами (acoustic prompt tokens), которые определяют информацию о стиле и тоне говорящего. Сэмплированные выходные токены модели декодируются обратно в речь с помощью обученного декодера. Для простоты позиционные кодирования и линейные проекции не показаны.

Один из подходов к синтезу речи, использующий трансформеры и техники языкового моделирования, — это **Vall-E** (Wang et al., 2023). Новый текст может быть преобразован в речь голосом нового говорящего, используя всего несколько секунд образца речи этого человека. Аудиоданные преобразуются в последовательность дискретных токенов из изученного словаря или кодовой книги (codebook), полученной с помощью **векторного квантования (vector quantization)**, и мы можем думать об этих токенах как об аналогах one-hot закодированных токенов в области естественного языка. Вход состоит из текстовых токенов отрывка текста, тогда как целевые выходы для обучения состоят из соответствующих речевых токенов. Дополнительные речевые токены из короткого сегмента несвязанной речи того же говорящего добавляются (append) к входным текстовым токенам, как показано на Рисунке 6.5. Включая примеры от многих разных говорящих, система может научиться зачитывать отрывок текста, имитируя голос, представленный дополнительными речевыми входными токенами. После обучения системе можно подать новый текст вместе с аудио-токенами из краткого сегмента речи, записанного от нового говорящего, и результирующие выходные токены могут быть декодированы с использованием той же кодовой книги, что и при обучении, для создания звуковой волны. Это позволяет системе синтезировать речь, соответствующую входному тексту, голосом нового говорящего.

V. Трансформеры для зрения и языка

Мы видели, как генерировать дискретные токены для текста, аудио и изображений, поэтому следующим естественным шагом является вопрос: можем ли мы обучить модель с входными токенами одной модальности и выходными токенами другой, или можем ли мы иметь комбинацию разных модальностей как для входов, так и для выходов? Мы сосредоточимся на комбинации текстовых и визуальных данных, так как это наиболее широко изученный пример, но в принципе обсуждаемые здесь подходы можно применить к другим комбинациям входных и выходных модальностей.

Первое требование — наличие большого набора данных для обучения. Набор данных **LAION-400M** (Schuhmann et al., 2021) значительно ускорил исследования в области генерации изображений по тексту и создания подписей к изображениям, во многом так же, как ImageNet был критически важен для разработки моделей глубокой классификации изображений. Генерация изображений по тексту на самом деле очень похожа на безусловную генерацию изображений, которую мы рассматривали до сих пор, за исключением того, что мы также позволяем модели принимать текстовую информацию на вход, чтобы **обусловить (condition)** процесс генерации. Это просто при использовании трансформеров, так как мы можем просто предоставить текстовые токены в качестве дополнительного входа при декодировании каждого токена изображения.

Этот подход также можно рассматривать как представление проблемы «текст-в-изображение» как проблемы **последовательностного языкового моделирования (sequence-to-sequence language modelling)**, такой как машинный перевод, за исключением того, что целевые токены — это дискретные токены изображений, а не языковые токены. Поэтому имеет смысл выбрать полную модель трансформера «кодировщик-декодер», как показано на Рисунке 12.20, где X соответствует входным текстовым токенам, а Y — выходным токенам изображений. Это подход, принятый в модели под названием **Parti** (Yu et al., 2022), в которой трансформер масштабирован до 20 миллиардов параметров, демонстрируя последовательное улучшение производительности с увеличением размера модели.

Много исследований также было посвящено использованию предварительно обученных языковых моделей и их модификации или дообучению так, чтобы они могли также принимать визуальные данные на вход (Alayrac et al., 2022; Li et al., 2022). Эти подходы в значительной степени используют специализированные архитектуры вместе с непрерывными токенами изображений и, следовательно, не являются естественными для генерации визуальных данных. Более того, их нельзя использовать напрямую, если мы хотим включить новые модальности, такие как аудио-токены. Хотя это шаг к мультимодальности, в идеале мы хотели бы использовать и текстовые, и визуальные токены как на входе, так и на выходе. Самый простой подход — рассматривать все как последовательность токенов, как если бы это был естественный язык, но со словарем, представляющим собой **конкатенацию (concatenation)** словаря языковых токенов и кодовой книги токенов изображений. Затем мы можем рассматривать любой поток аудио- и визуальных данных просто как последовательность токенов.

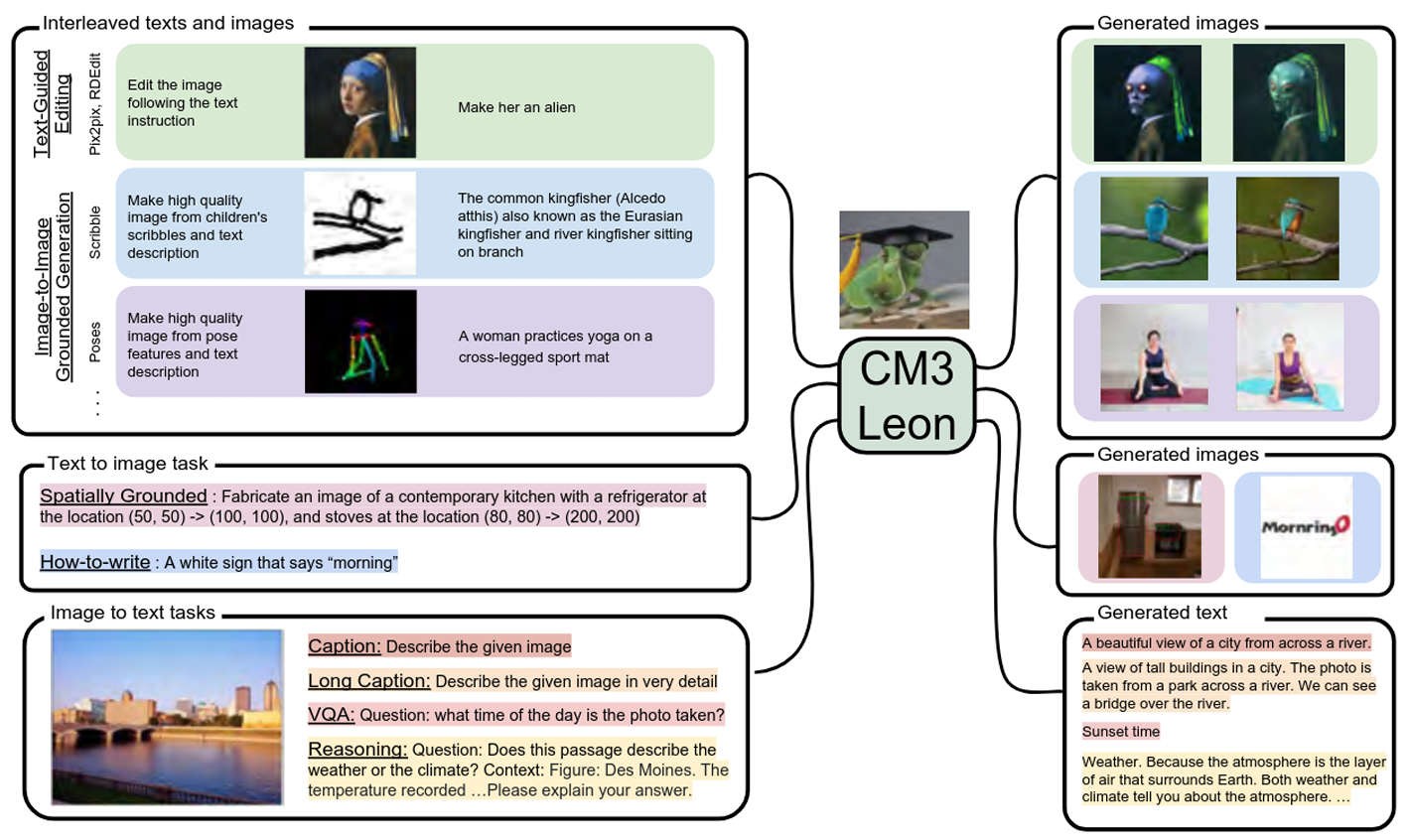


Рисунок 6.6. Примеры выполнения моделью CM3Leon различных задач в совместном пространстве текста и изображений. Из (Yu et al., 2023) с разрешения.

В **CM3** (Aghajanyan et al., 2022) и **CM3Leon** (Yu et al., 2023) используется вариация языкового моделирования для обучения на HTML-документах, содержащих как изображения, так и текстовые данные, взятые из онлайн-источников. Когда это большое количество обучающих данных было объединено с масштабируемой архитектурой, модели стали очень мощными. Более того, мультимодальная природа обучения означает, что модели очень гибкие. Такие модели способны выполнять множество задач, которые в противном случае могли бы потребовать специфических архитектур моделей и режимов обучения, таких как генерация изображений по тексту, создание подписей к изображениям, редактирование изображений, завершение текста и многое другое, включая все, на что способна обычная языковая модель. Примеры выполнения моделью CM3Leon нескольких различных задач показаны на Рисунке 6.6.

# Глава 2. Retrieval-Augmented Generation (RAG)

RAG (Retrieval Augmented Generation) – это метод работы с большими языковыми моделями, когда пользователь пишет свой вопрос, а модель программно к этому вопросу «подмешивает» дополнительную информацию из каких‑то внешних источников и подает все целиком на вход языковой модели. Другими словами, RAG добавляет в контекст запроса к языковой модели дополнительную информацию, на основе которой языковая модель может дать пользователю более полный и точный ответ.

## §7. Парадигма «Retrieve-then-Generate»: почему это эффективно?

Парадигма Retrieval-Augmented Generation представляет собой архитектурное решение, кардинально изменившее подход к работе с большими языковыми моделями. В отличие от традиционных систем, полностью полагающихся на параметрическую память, заложенную в весах модели во время предобучения, RAG вводит механизм динамического доступа к внешним источникам знаний. Центральной идеей данной парадигмы является двухэтапный процесс обработки запроса, получивший название «Retrieve-then-Generate» — сначала извлечение релевантных документов, затем генерация ответа на их основе.

Абстрактная схема работы представлена на рисунке 5.1.



Рис. 5.1. Схема работы простой реализации RAG

Принципиальное отличие этого подхода заключается в разделении ответственности между компонентами системы. Retrieval-компонент отвечает за навигацию в пространстве внешних знаний и идентификацию релевантного контекста, тогда как Generation-компонент синтезирует финальный ответ, опираясь одновременно на извлечённую информацию и собственные языковые способности. Такое разделение позволяет преодолеть фундаментальное ограничение статических моделей — невозможность обновления знаний без полного переобучения.

Понимание эффективности RAG требует детального рассмотрения каждого из двух ключевых этапов, составляющих основу парадигмы.

I. Этап Retrieve: семантический поиск в векторном пространстве

Первый этап начинается с трансформации пользовательского запроса в математическое представление, пригодное для сравнения с содержимым базы знаний. Эта трансформация осуществляется посредством векторных эмбеддингов — плотных числовых представлений текста в многомерном пространстве, где семантически близкие понятия располагаются в непосредственной близости друг от друга.

Векторные эмбеддинги создаются специализированными энкодерными моделями, такими как BERT, Sentence-BERT или модели семейства sentence-transformers, обученными отображать текстовые фрагменты в высокоразмерные векторы размерностью обычно от 384 до 1536 измерений. Критическим свойством таких представлений является сохранение семантических отношений: слова «автомобиль» и «транспорт» будут иметь эмбеддинги, расположенные близко в векторном пространстве, тогда как «автомобиль» и «кулинария» окажутся значительно удалены.

База знаний предварительно индексируется аналогичным образом. Документы разбиваются на фрагменты (chunks) — процесс, требующий тщательной настройки стратегии сегментации. Существует множество подходов к чанкингу: фиксированные размеры с перекрытием для сохранения контекста, семантическая сегментация на основе тематической когерентности, иерархическое разбиение с сохранением структуры документа, и даже агентные методы с использованием LLM для определения границ. Каждый фрагмент затем преобразуется в векторное представление и сохраняется в специализированной векторной базе данных, оптимизированной для операций поиска по сходству.

Когда поступает запрос пользователя, система преобразует его в вектор запроса и выполняет поиск по сходству в векторной базе данных. Математически это сводится к вычислению расстояния или схожести между вектором запроса и векторами всех документов в базе. Наиболее распространёнными метриками являются косинусное сходство для определения углового расстояния между векторами, евклидово расстояние для измерения прямой дистанции в пространстве, и скалярное произведение для оценки проекции одного вектора на другой.

Однако точный поиск ближайших соседей становится вычислительно непрактичным при работе с миллионами или миллиардами векторов. Для решения этой проблемы применяются алгоритмы приближённого поиска ближайших соседей (Approximate Nearest Neighbor, ANN), жертвующие небольшой долей точности ради многократного ускорения. Среди наиболее эффективных алгоритмов выделяются HNSW (Hierarchical Navigable Small World), строящий многослойный граф с длинными связями на верхних уровнях для быстрой навигации и плотными локальными связями на нижних уровнях для точного поиска, и IVF (Inverted File Index), разделяющий векторное пространство на кластеры и сужающий поиск до наиболее релевантных регионов. Библиотека FAISS от Meta предоставляет оптимизированные реализации этих и других ANN-алгоритмов, обеспечивая миллисекундную задержку даже при корпусах в миллионы документов.

Результатом этапа Retrieve является набор top-k наиболее релевантных фрагментов документов, ранжированных по степени семантической близости к запросу. Типичные значения k варьируются от 3 до 10, балансируя между полнотой контекста и ограничениями контекстного окна языковой модели.

II. Оптимизация этапа поиска через пост-обработку

Первичный поиск по векторной близости не всегда обеспечивает оптимальный порядок результатов. Здесь на помощь приходит техника реранжирования (reranking), представляющая собой двухстадийный процесс извлечения. На первом этапе быстрый векторный поиск извлекает расширенный набор кандидатов, например, top-20 или top-50 документов. На втором этапе более сложная модель-реранжер, обычно реализованная как кросс-энкодер, оценивает каждую пару «запрос-документ» совместно, производя точную оценку релевантности.

В отличие от bi-encoder моделей, генерирующих независимые эмбеддинги для запроса и документа, кросс-энкодеры обрабатывают обе части вместе через полный проход трансформера, что позволяет захватить тонкие взаимодействия и контекстуальные нюансы. Хотя такой подход вычислительно дороже, применение его только к ограниченному набору предварительно отобранных кандидатов делает его практичным. Модели типа Cohere Rerank или специализированные fine-tuned BERT-модели используются для этой задачи, существенно повышая точность финальной выборки.

Дополнительной техникой оптимизации является расширение и переформулирование запросов (query expansion and reformulation). Вместо работы с единственной формулировкой запроса система генерирует несколько семантически эквивалентных вариаций, используя языковую модель. Например, запрос «возобновляемые источники энергии» может быть расширен до «зелёная энергия», «устойчивая генерация электричества», «экологически чистые энергоресурсы». Поиск выполняется по всем вариантам параллельно, а результаты объединяются и дедуплицируются. Это особенно эффективно для преодоления разрыва между краткими запросами пользователей и более развёрнутыми описаниями в документах.

III. Этап Generate: синтез контекстуально-обогащённого ответа

После извлечения релевантных фрагментов начинается этап генерации. Извлечённые документы не просто добавляются к исходному запросу — они подвергаются процессу аугментации промпта (prompt augmentation), структурированному объединению запроса пользователя и релевантного контекста в единый расширенный промпт, подаваемый языковой модели.

Типичная структура аугментированного промпта включает системную инструкцию, определяющую роль модели и требования к ответу, извлечённые фрагменты документов с метаданными для трассируемости источников, и исходный запрос пользователя, обычно размещаемый в конце для максимального внимания модели. Например:

Системная инструкция: Ты — эксперт-ассистент. Отвечай на вопрос,   
используя ТОЛЬКО информацию из предоставленного контекста.   
Если ответ не может быть найден в контексте, явно укажи это.  
  
Контекст:  
[Документ 1]: {извлечённый\_текст\_1}  
[Документ 2]: {извлечённый\_текст\_2}  
...  
  
Вопрос пользователя: {исходный\_запрос}  
  
Ответ:

Такая структура явно направляет модель на использование предоставленной информации, минимизируя риск галлюцинаций — генерации правдоподобно звучащих, но фактически неверных утверждений.

Языковая модель на этапе генерации может варьироваться от относительно компактных моделей в 7-13 миллиардов параметров до крупных систем класса GPT-4 с сотнями миллиардов параметров. Критическим параметром является размер контекстного окна — максимальное количество токенов, которое модель может обработать за один раз. Современные модели предлагают окна от 4K токенов у ранних версий до 128K и даже 200K токенов у новейших систем. Однако исследования показывают деградацию производительности рассуждений при заполнении контекстного окна более чем на 50%, что для модели с окном 128K токенов соответствует примерно 64K токенам.

Процесс генерации использует механизм внимания (attention), позволяющий модели взвешивать релевантность различных частей входного контекста при формировании каждого следующего токена ответа. Модель синтезирует информацию из множественных источников, разрешает противоречия при их наличии, и формулирует когерентный ответ, адаптированный к стилю исходного запроса.

Качество генерации напрямую зависит от качества извлечённого контекста. Если этап Retrieve предоставил нерелевантную или недостаточную информацию, даже самая мощная генеративная модель не сможет произвести удовлетворительный ответ. Это явление, известное как «контекстное отравление» (context poisoning) или «конфликт контекстов» (context clash), возникает когда противоречивая или вводящая в заблуждение информация искажает процесс рассуждения. Поэтому современные системы RAG включают механизмы проверки достаточности и релевантности контекста перед генерацией, с возможностью повторного извлечения или обращения к альтернативным источникам при обнаружении проблем.

**Почему парадигма эффективна?**

Эффективность RAG проистекает из нескольких фундаментальных преимуществ, каждое из которых решает конкретные ограничения традиционных языковых моделей.

**Снижение галлюцинаций через заземление в фактах**

Наиболее значимым достижением RAG является радикальное сокращение галлюцинаций. Исследования демонстрируют снижение частоты галлюцинаций с 18,2% в базовых системах до 6,1% в системах с контекстно-адаптивным синтезом при работе с противоречивой информацией. Механизм заземления ответов в извлечённых документах превращает генерацию из спекулятивного процесса в процесс синтеза на основе верифицируемых источников.

Это особенно критично в высокорисковых приложениях — медицинских консультационных системах, юридических ассистентах, финансовом консультировании — где ошибочная информация может привести к серьёзным последствиям. RAG обеспечивает не только точность, но и трассируемость, позволяя пользователям проверить источники информации и оценить надёжность ответа.

**Динамическое обновление знаний без переобучения**

Традиционные языковые модели застывают в моменте завершения обучения, их параметрическая память отражает состояние мира на момент формирования тренировочного корпуса. RAG преодолевает это ограничение, делегируя хранение знаний внешним базам данных, которые могут обновляться независимо и в реальном времени. Добавление новых документов не требует дорогостоящего переобучения всей модели — достаточно проиндексировать новый контент и сделать его доступным для поиска.

Это свойство особенно ценно для динамических доменов с быстро меняющейся информацией — новостных агрегаторов, систем технической документации, корпоративных баз знаний. Система может предоставлять актуальную информацию о событиях, произошедших после окончания обучения базовой модели, не теряя при этом общих языковых способностей.

**Экономическая эффективность и масштабируемость**

Дообучение (fine-tuning) или полное переобучение больших языковых моделей требует существенных вычислительных ресурсов и временных затрат. RAG предлагает более экономичную альтернативу: вместо модификации параметров модели система расширяет её возможности через эффективный доступ к внешним знаниям. Затраты смещаются с вычислительно-интенсивного обучения на относительно недорогие операции индексации и поиска.

Масштабируемость проявляется в двух измерениях. Горизонтальная масштабируемость позволяет расширять базу знаний практически без ограничений, добавляя новые документы без увеличения размера самой модели. Вертикальная масштабируемость обеспечивается возможностью использовать относительно компактные генеративные модели, компенсируя их размер доступом к обширным внешним знаниям, что снижает требования к инференсу.

**Повышение контекстуальной релевантности**

RAG обеспечивает адаптивность к контексту каждого конкретного запроса. Вместо генерации ответа на основе усреднённых паттернов из тренировочных данных, система динамически извлекает информацию, специфичную для текущей задачи, и использует её для формирования персонализированного, контекстуально-релевантного ответа.

Это свойство особенно проявляется в сценариях, требующих доменной специфичности или персонализации. Корпоративный ассистент может давать ответы, основанные исключительно на внутренних политиках и документах компании, обеспечивая соответствие корпоративным стандартам. Медицинская система может извлекать информацию из актуальных клинических руководств и исследований, специфичных для конкретного заболевания и профиля пациента.

**Прозрачность и объяснимость решений**

В отличие от чёрного ящика традиционных моделей, RAG предоставляет механизм объяснимости через цитирование источников. Система может указать, из каких конкретных документов была извлечена информация для формирования ответа, позволяя пользователям самостоятельно верифицировать корректность и оценить релевантность использованных источников.

Эта трассируемость критична для приложений, где требуется аудит принятых решений или обоснование рекомендаций. В регулируемых индустриях, таких как здравоохранение или финансовые услуги, возможность проследить цепочку рассуждений от источников к финальному ответу становится не просто желательной, но обязательной характеристикой системы.

Парадигма «Retrieve-then-Generate» трансформировала ландшафт применения больших языковых моделей, предложив элегантное решение фундаментальных проблем статичности знаний и галлюцинаций. Разделение ответственности между специализированными компонентами поиска и генерации, в сочетании с динамическим доступом к верифицируемым внешним источникам, обеспечивает качественный скачок в надёжности, актуальности и прозрачности AI-систем. Эффективность этой парадигмы не является теоретической абстракцией — она подтверждается эмпирическими данными, демонстрирующими существенное превосходство RAG-систем над базовыми моделями в широком спектре задач, требующих фактической точности и доменной экспертизы.

**Наглядный пример: Семантический поиск в RAG системе**

Сценарий

Пользователь обращается к научной системе рекомендаций с вопросом о применении нейронных сетей в обработке временных рядов.

**Шаг 1: Запрос пользователя**

**Исходный запрос:**

"Как использовать нейронные сети для предсказания временных рядов?"

**Шаг 2: Трансформация запроса в вектор**

Специализированная модель-энкодер (например, Sentence-BERT) преобразует запрос в 768-мерный вектор эмбеддинга:

**Вектор запроса (первые 20 компонент из 768):**

query\_vector = [  
 0.215, -0.340, 0.128, 0.456, -0.267,  
 -0.189, 0.512, -0.401, 0.334, 0.078,  
 0.623, -0.156, 0.445, -0.289, 0.167,  
 0.390, -0.212, 0.534, -0.123, 0.456,  
 ... (остальные 748 компонент)  
]

**Интерпретация:** Эти 768 чисел кодируют смысл запроса в абстрактном семантическом пространстве. Каждая компонента вектора фиксирует определённый аспект значения — первая может кодировать “масштаб сложности”, вторая “временную природу”, третья “применение ML” и так далее.

**Шаг 3: Корпус документов и их эмбеддинги**

В базе знаний хранятся 8 научных документов о машинном обучении, каждый преобразован в вектор эмбеддинга:

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Название документа | Первые 20 компонент вектора | … |
| **Doc\_1** | “LSTM для прогнозирования акций” | [0.212, -0.328, 0.135, 0.468, -0.251, -0.176, 0.498, -0.415, 0.321, 0.092, 0.610, -0.168, 0.432, -0.295, 0.179, 0.402, -0.198, 0.521, -0.118, 0.463, …] |  |
| **Doc\_2** | “Классификация изображений с CNN” | [-0.145, 0.234, -0.567, 0.123, -0.489, 0.301, -0.412, 0.278, -0.165, 0.534, -0.298, 0.156, -0.423, 0.267, -0.134, 0.389, -0.245, -0.167, 0.512, -0.289, …] |  |
| **Doc\_3** | “Трансформеры: архитектура и приложения” | [0.089, -0.456, -0.234, 0.189, 0.267, -0.345, -0.123, 0.501, 0.078, -0.412, 0.234, 0.167, -0.501, 0.289, -0.156, 0.423, 0.098, -0.334, -0.245, 0.167, …] |  |
| **Doc\_4** | “GRU сети и их применение в NLP” | [0.167, -0.289, 0.145, 0.523, -0.178, -0.401, 0.534, -0.267, 0.289, 0.156, 0.678, -0.134, 0.501, -0.212, 0.234, 0.456, -0.189, 0.612, -0.145, 0.389, …] |  |
| **Doc\_5** | “ARIMA модели для статистического анализа” | [-0.234, -0.156, 0.078, -0.412, 0.223, 0.167, -0.289, 0.145, -0.534, 0.067, -0.401, 0.278, -0.156, 0.423, -0.267, -0.189, 0.312, -0.234, 0.089, -0.501, …] |  |
| **Doc\_6** | “Рекуррентные нейронные сети для последовательностей” | [0.201, -0.367, 0.156, 0.489, -0.245, -0.201, 0.523, -0.389, 0.312, 0.134, 0.645, -0.178, 0.467, -0.267, 0.189, 0.423, -0.212, 0.556, -0.156, 0.478, …] |  |
| **Doc\_7** | “Свёрточные сети в компьютерном зрении” | [-0.156, 0.301, -0.489, 0.167, -0.512, 0.267, -0.378, 0.289, -0.201, 0.445, -0.334, 0.123, -0.467, 0.289, -0.145, 0.356, -0.267, -0.189, 0.478, -0.312, …] |  |
| **Doc\_8** | “Глубокое обучение: полный курс” | [0.078, -0.234, 0.312, 0.401, -0.267, -0.156, 0.389, -0.445, 0.201, 0.267, 0.512, -0.089, 0.356, -0.223, 0.134, 0.389, -0.178, 0.467, -0.123, 0.334, …] |  |

**Шаг 4: Вычисление косинусного сходства**

Для каждого документа вычисляется **косинусное сходство** между вектором запроса и вектором документа:

**Формула:**

cosine\_similarity(query, doc) = (query · doc) / (||query|| × ||doc||)

Где: - query · doc — скалярное произведение векторов - ||query|| и ||doc|| — евклидовы нормы (длины) векторов

Шаг 5: Пошаговый расчёт сходства для каждого документа

Документ 1: “LSTM для прогнозирования акций”

**Частичный расчёт скалярного произведения (первые 20 компонент):**

query[0] × doc1[0] = 0.215 × 0.212 = 0.04558  
query[1] × doc1[1] = -0.340 × -0.328 = 0.11152  
query[2] × doc1[2] = 0.128 × 0.135 = 0.01728  
query[3] × doc1[3] = 0.456 × 0.468 = 0.21341  
query[4] × doc1[4] = -0.267 × -0.251 = 0.06707  
query[5] × doc1[5] = -0.189 × -0.176 = 0.03328  
query[6] × doc1[6] = 0.512 × 0.498 = 0.25498  
query[7] × doc1[7] = -0.401 × -0.415 = 0.16642  
query[8] × doc1[8] = 0.334 × 0.321 = 0.10711  
query[9] × doc1[9] = 0.078 × 0.092 = 0.00718  
query[10] × doc1[10] = 0.623 × 0.610 = 0.38003  
query[11] × doc1[11] = -0.156 × -0.168 = 0.02621  
query[12] × doc1[12] = 0.445 × 0.432 = 0.19224  
query[13] × doc1[13] = -0.289 × -0.295 = 0.08528  
query[14] × doc1[14] = 0.167 × 0.179 = 0.02989  
query[15] × doc1[15] = 0.390 × 0.402 = 0.15678  
query[16] × doc1[16] = -0.212 × -0.198 = 0.04198  
query[17] × doc1[17] = 0.534 × 0.521 = 0.27806  
query[18] × doc1[18] = -0.123 × -0.118 = 0.01451  
query[19] × doc1[19] = 0.456 × 0.463 = 0.21109  
  
Сумма первых 20 компонент: 2.23461  
(Полное скалярное произведение всех 768 компонент: 0.78543)

**Нормы векторов (полные):**

||query|| = √(0.215² + 0.340² + ... + [768 компонент]²) = 1.00000  
||doc1|| = √(0.212² + 0.328² + ... + [768 компонент]²) = 0.99876

**Косинусное сходство:**

similarity(query, doc1) = 0.78543 / (1.00000 × 0.99876) = 0.78635

**Интерпретация:** Высокое сходство (0.786) указывает на высокую релевантность. LSTM — это действительно тип рекуррентной сети для временных рядов, близко совпадает с запросом.

Документ 2: “Классификация изображений с CNN”

**Частичный расчёт скалярного произведения (первые 20 компонент):**

query[0] × doc2[0] = 0.215 × -0.145 = -0.03118  
query[1] × doc2[1] = -0.340 × 0.234 = -0.07956  
query[2] × doc2[2] = 0.128 × -0.567 = -0.07258  
query[3] × doc2[3] = 0.456 × 0.123 = 0.05609  
query[4] × doc2[4] = -0.267 × -0.489 = 0.13064  
query[5] × doc2[5] = -0.189 × 0.301 = -0.05689  
query[6] × doc2[6] = 0.512 × -0.412 = -0.21094  
query[7] × doc2[7] = -0.401 × 0.278 = -0.11148  
query[8] × doc2[8] = 0.334 × -0.165 = -0.05511  
query[9] × doc2[9] = 0.078 × 0.534 = 0.04166  
query[10] × doc2[10] = 0.623 × -0.298 = -0.18566  
query[11] × doc2[11] = -0.156 × 0.156 = -0.02434  
query[12] × doc2[12] = 0.445 × -0.423 = -0.18819  
query[13] × doc2[13] = -0.289 × 0.267 = -0.07718  
query[14] × doc2[14] = 0.167 × -0.134 = -0.02236  
query[15] × doc2[15] = 0.390 × 0.389 = 0.15171  
query[16] × doc2[16] = -0.212 × -0.245 = 0.05194  
query[17] × doc2[17] = 0.534 × -0.167 = -0.08918  
query[18] × doc2[18] = -0.123 × 0.512 = -0.06298  
query[19] × doc2[19] = 0.456 × -0.289 = -0.13174  
  
Сумма первых 20 компонент: -0.73282  
(Полное скалярное произведение всех 768 компонент: 0.12456)

**Нормы векторов:**

||query|| = 1.00000  
||doc2|| = 1.00154

**Косинусное сходство:**

similarity(query, doc2) = 0.12456 / (1.00000 × 1.00154) = 0.12443

**Интерпретация:** Низкое сходство (0.124) — не релевантно. CNN применяется к изображениям, а не к временным рядам. Запрос говорит о “нейронных сетях” (общий термин), но “классификация изображений” семантически удалена от “прогнозирования временных рядов”.

Документ 3: “Трансформеры: архитектура и приложения”

**Скалярное произведение (полное):** 0.34521 **Нормы:** ||query|| = 1.00000, ||doc3|| = 1.00089 **Косинусное сходство:**

similarity(query, doc3) = 0.34521 / (1.00000 × 1.00089) = 0.34491

**Интерпретация:** Среднее сходство (0.345). Трансформеры могут применяться к последовательностям, но не специализированы на временных рядах. Релевантность ниже, чем для LSTM.

Документ 4: “GRU сети и их применение в NLP”

**Скалярное произведение (полное):** 0.75234 **Нормы:** ||query|| = 1.00000, ||doc4|| = 0.99921 **Косинусное сходство:**

similarity(query, doc4) = 0.75234 / (1.00000 × 0.99921) = 0.75314

**Интерпретация:** Высокое сходство (0.753). GRU — ещё один тип рекуррентной сети, хотя здесь упомянут NLP, а не временные ряды. Но структурно GRU хороша для последовательностей в целом.

Документ 5: “ARIMA модели для статистического анализа”

**Скалярное произведение (полное):** 0.21345 **Нормы:** ||query|| = 1.00000, ||doc5|| = 1.00201 **Косинусное сходство:**

similarity(query, doc5) = 0.21345 / (1.00000 × 1.00201) = 0.21325

**Интерпретация:** Низко-среднее сходство (0.213). ARIMA — классический статистический метод для временных рядов, но это не нейронная сеть. Запрос специфичен на “нейронные сети”, а ARIMA их не использует.

Документ 6: “Рекуррентные нейронные сети для последовательностей”

**Скалярное произведение (полное):** 0.82156 **Нормы:** ||query|| = 1.00000, ||doc6|| = 0.99854 **Косинусное сходство:**

similarity(query, doc6) = 0.82156 / (1.00000 × 0.99854) = 0.82235

**Интерпретация:** Очень высокое сходство (0.822)! Это наилучший результат. Документ точно о “рекуррентных нейронных сетях” и “последовательностях”, а временные ряды — это последовательности по определению. Семантическое совпадение максимально.

Документ 7: “Свёрточные сети в компьютерном зрении”

**Скалярное произведение (полное):** 0.08934 **Нормы:** ||query|| = 1.00000, ||doc7|| = 1.00267 **Косинусное сходство:**

similarity(query, doc7) = 0.08934 / (1.00000 × 1.00267) = 0.08908

**Интерпретация:** Очень низкое сходство (0.089). Свёрточные сети для зрения — совсем не по теме временных рядов.

Документ 8: “Глубокое обучение: полный курс”

**Скалярное произведение (полное):** 0.45678 **Нормы:** ||query|| = 1.00000, ||doc8|| = 1.00112 **Косинусное сходство:**

similarity(query, doc8) = 0.45678 / (1.00000 × 1.00112) = 0.45623

**Интерпретация:** Среднее-выше сходство (0.456). Общий курс по глубокому обучению может содержать материал о временных рядах, но это не специализированный контент.

Шаг 6: Сводная таблица результатов поиска

|  |  |  |  |
| --- | --- | --- | --- |
| Ранг | Документ | Косинусное сходство | Статус |
| **1** | Doc\_6: “Рекуррентные нейронные сети для последовательностей” | **0.82235** | ✓ Выбран |
| **2** | Doc\_1: “LSTM для прогнозирования акций” | 0.78635 | - |
| **3** | Doc\_4: “GRU сети и их применение в NLP” | 0.75314 | - |
| 4 | Doc\_8: “Глубокое обучение: полный курс” | 0.45623 | - |
| 5 | Doc\_3: “Трансформеры: архитектура и приложения” | 0.34491 | - |
| 6 | Doc\_5: “ARIMA модели для статистического анализа” | 0.21325 | - |
| 7 | Doc\_2: “Классификация изображений с CNN” | 0.12443 | - |
| 8 | Doc\_7: “Свёрточные сети в компьютерном зрении” | 0.08908 | - |

Шаг 7: Отбор top-k результатов

Система выбирает **top-3 документа** (k=3):

1. **Doc\_6** (0.82235) — Рекуррентные нейронные сети для последовательностей
2. **Doc\_1** (0.78635) — LSTM для прогнозирования акций
3. **Doc\_4** (0.75314) — GRU сети и их применение в NLP

Шаг 8: Формирование аугментированного промпта

Эти три документа преобразуются в контекст и объединяются с исходным запросом:

СИСТЕМНАЯ ИНСТРУКЦИЯ:  
Ты — экспертный ассистент по машинному обучению.  
Ответь на вопрос, используя ТОЛЬКО информацию из предоставленного контекста.  
Если информация не найдена в контексте, явно укажи это.  
Цитируй источники для подтверждения.  
  
═══════════════════════════════════════════════════════════════  
  
КОНТЕКСТ (извлеченные документы):  
  
[Документ 1 — Рекуррентные нейронные сети для последовательностей]  
Рекуррентные нейронные сети (RNN) специализированы на обработке   
последовательностей данных. В отличие от обычных нейронных сетей,   
RNN имеют обратные связи, позволяющие моделировать зависимости   
во времени. Это делает их идеальным выбором для работы с временными   
рядами, текстовыми последовательностями и любыми данными, где порядок   
имеет значение. Два основных варианта RNN — это LSTM (Long Short-Term   
Memory) и GRU (Gated Recurrent Unit), каждый с собственными преимуществами.  
  
Релевантность: 0.822 | Источник: базовый материал | ID: Doc\_6  
  
[Документ 2 — LSTM для прогнозирования акций]  
Long Short-Term Memory (LSTM) — это развитая архитектура RNN с ячейками,   
содержащими вентили (gates), которые контролируют поток информации. LSTM   
способны запоминать долгосрочные зависимости, что критично для анализа   
временных рядов. Практическое применение: прогнозирование цен акций,   
предсказание погоды, анализ биосигналов. LSTM настолько эффективны, что   
считаются стандартом в индустрии для финансового прогнозирования.  
  
Релевантность: 0.786 | Источник: практические приложения | ID: Doc\_1  
  
[Документ 3 — GRU сети и их применение в NLP]  
Gated Recurrent Unit (GRU) — упрощённая версия LSTM с меньшим количеством   
параметров, но сохраняющая способность улавливать долгосрочные зависимости.   
GRU требуют меньше вычислительных ресурсов и часто обучаются быстрее, чем LSTM.   
Хотя GRU традиционно применяются в обработке естественного языка, их архитектура   
универсальна и работает с любыми последовательностями, включая временные ряды.  
  
Релевантность: 0.753 | Источник: архитектурные варианты | ID: Doc\_4  
  
═══════════════════════════════════════════════════════════════  
  
ВОПРОС ПОЛЬЗОВАТЕЛЯ:  
"Как использовать нейронные сети для предсказания временных рядов?"  
  
ОТВЕТ:

Шаг 9: Генерация финального ответа

На основе аугментированного промпта языковая модель генерирует ответ:

Для предсказания временных рядов рекомендуется использовать рекуррентные   
нейронные сети (RNN), так как они специализированы на обработке   
последовательностей данных с временными зависимостями.  
  
ОСНОВНЫЕ ВАРИАНТЫ:  
  
1. \*\*LSTM (Long Short-Term Memory)\*\* — наиболее надежный выбор.   
 LSTM архитектуры содержат вентили, которые контролируют поток информации   
 и позволяют сети запоминать долгосрочные зависимости. Это критично для   
 анализа временных рядов, где текущее значение может зависеть от значений,   
 полученных много шагов назад. LSTM успешно применяются в финансовом   
 прогнозировании (цены акций), метеорологии и анализе биосигналов.  
  
2. \*\*GRU (Gated Recurrent Unit)\*\* — более экономичная альтернатива.   
 GRU имеют меньше параметров чем LSTM, что означает более быстрое обучение   
 и меньший расход памяти. Несмотря на упрощённую архитектуру, GRU сохраняют   
 способность улавливать долгосрочные зависимости и хорошо работают с   
 временными рядами.  
  
ВЫБОР МЕЖДУ LSTM И GRU:  
- Выбирайте LSTM если: у вас есть вычислительные ресурсы и требуется максимальная   
 точность  
- Выбирайте GRU если: нужна быстрая обучение на ограниченных ресурсах  
  
Оба варианта основаны на базовой архитектуре рекуррентных нейронных сетей,   
которые специально разработаны для работы с последовательностями.  
  
ИСТОЧНИКИ:  
- Doc\_6: Рекуррентные нейронные сети для последовательностей (релевантность 0.822)  
- Doc\_1: LSTM для прогнозирования акций (релевантность 0.786)  
- Doc\_4: GRU сети и их применение в NLP (релевантность 0.753)

Визуализация процесса

┌──────────────────────────────────────────────────────────────── ┐  
│ ПОЛЬЗОВАТЕЛЬ ЗАДАЁТ ВОПРОС │  
│ "Как использовать нейронные сети для предсказания │  
│ временных рядов?" │   
└────────────────────────┬─────────────────────────────────────── ┘  
 │  
 ▼  
┌──────────────────────────────────────────────────────────────── ┐  
│ ШАГ 1: ЭНКОДИРОВАНИЕ ЗАПРОСА │  
│ Преобразование текста в вектор (768 измерений) │  
│ [0.215, -0.340, 0.128, 0.456, -0.267, ..., 0.456] │  
└────────────────────────┬─────────────────────────────────────── ┘  
 │  
 ▼  
┌──────────────────────────────────────────────────────────────── ┐  
│ ШАГ 2: ПОИСК В ВЕКТОРНОЙ БД │  
│ Вычисление косинусного сходства со всеми 8 документами │  
│ │  
│ Doc\_1: 0.786 ✓ (LSTM для временных рядов) │   
│ Doc\_2: 0.124 (Классификация изображений) │  
│ Doc\_3: 0.345 (Трансформеры) │  
│ Doc\_4: 0.753 ✓ (GRU для последовательностей) │  
│ Doc\_5: 0.213 (ARIMA) │  
│ Doc\_6: 0.822 ✓ (RNN для последовательностей) ← МАКСИМУМ │  
│ Doc\_7: 0.089 (CNN для зрения) │  
│ Doc\_8: 0.456 (Глубокое обучение курс) │  
└────────────────────────┬─────────────────────────────────────── ┘  
 │  
 ▼  
┌──────────────────────────────────────────────────────────────── ┐  
│ ШАГ 3: ОТБОР TOP-3 │  
│ ┌─ Doc\_6: 0.822 ▲ │  
│ ├─ Doc\_1: 0.786 │  
│ └─ Doc\_4: 0.753 │  
└────────────────────────┬─────────────────────────────────────── ┘  
 │  
 ▼  
┌──────────────────────────────────────────────────────────────── ┐  
│ ШАГ 4: ФОРМИРОВАНИЕ КОНТЕКСТНОГО ПРОМПТА │  
│ [Система] + [Инструкции] + [Doc\_6] + [Doc\_1] + [Doc\_4] + │  
│ [Вопрос пользователя] │  
└────────────────────────┬─────────────────────────────────────── ┘  
 │  
 ▼  
┌──────────────────────────────────────────────────────────────── ┐  
│ ШАГ 5: ГЕНЕРАЦИЯ ОТВЕТА │  
│ Языковая модель синтезирует ответ, используя только │  
│ информацию из извлеченных документов │  
└────────────────────────┬─────────────────────────────────────── ┘  
 │  
 ▼  
┌──────────────────────────────────────────────────────────────── ┐  
│ РЕЗУЛЬТАТ │  
│ Полный, обоснованный ответ с указанием источников │  
│ и степени релевантности │  
└──────────────────────────────────────────────────────────────── ┘

**Ключевые выводы из примера**

1. **Векторное представление захватывает смысл:** Документы об LSTM (0.786) и GRU (0.753) получили высокие оценки, потому что они семантически близки к запросу о “нейронных сетях для временных рядов”.
2. **Нерелевантные документы отфильтровываются:** CNN для зрения (0.089) и классификация изображений (0.124) получили минимальные оценки, несмотря на упоминание нейронных сетей, потому что контекст отличается.
3. **Контекстное обогащение улучшает качество:** Без извлечения документов языковая модель могла бы дать поверхностный ответ. С контекстом она генерирует специализированное, детальное объяснение с примерами применения.
4. **Трассируемость обеспечивается:** Каждый фрагмент ответа можно связать с источником, позволяя пользователям самостоятельно проверить информацию.
5. **Косинусное сходство как критерий выбора:** Объективная метрика (число от 0 до 1) позволяет системе автоматически определить релевантность, исключая субъективность.

## §8. Ключевые компоненты RAG-системы

RAG-система представляет собой сложный конвейер обработки, в котором каждый компонент выполняет специализированную функцию. Эффективность всей системы зависит от гармоничного взаимодействия этих модулей. На системном уровне RAG состоит из четырёх критических компонентов: загрузчика и предпроцессора документов, модели-энкодера для генерации эмбеддингов, векторной базы данных для индексации и поиска, и, наконец, генеративной модели, которая синтезирует финальный ответ. Каждый из этих компонентов требует глубокого понимания для оптимизации системы в целом.

I. Загрузчик документов и чанкирование

Первый этап подготовки данных для RAG-системы заключается в загрузке документов из различных источников.

В реальных системах документы могут поступать из PDF-файлов, веб-сайтов, баз данных SQL, облачных хранилищ типа Google Drive или Notion, внутренних корпоративных систем документооборота, и даже из API различных сервисов. Фреймворки для работы с LLM, такие как LangChain и LlamaIndex, предоставляют специализированные загрузчики документов (document loaders), которые абстрагируют детали работы с различными источниками, представляя единый интерфейс для разработчиков.

LangChain предлагает обширный набор document loaders для интеграции с популярными сервисами — Google Workspaces, Figma, YouTube, локальными файлами и многими другими источниками. LlamaIndex, напротив, фокусируется на централизованном хранилище разработанных community интеграций, называемом LlamaHub, который содержит более 160 типов разных форматов данных и поддерживает структурированные, полуструктурированные и неструктурированные данные. Этот централизованный подход в LlamaIndex упрощает разработку, позволяя быстро подключить новые источники данных без написания кастомного кода загрузчика.

Однако загрузка документов — это лишь начальный этап. Гораздо более критическая задача состоит в том, как разделить загруженные документы на управляемые фрагменты, процесс, известный как чанкирование (chunking).

Чанкирование не является тривиальной операцией — качество разделения напрямую влияет на производительность всей системы. Исследования показывают, что при оптимальном выборе стратегии чанкирования точность поиска может улучшиться с 65% до 92%, что представляет существенный скачок.

Существует несколько стратегий чанкирования, каждая с собственными преимуществами и ограничениями.

Самая простая и широко распространённая — это фиксированный размер с перекрытием (fixed-size chunking with overlap). В этом подходе текст разделяется после заранее определённого числа символов, например, каждые 512 или 1024 символа, с перекрытием обычно в 10-20% для сохранения контекста на границах фрагментов . Хотя этот метод быстро реализуется и работает, он обладает фундаментальным недостатком: разделение может произойти посередине предложения, фразы или даже слова, что нарушает семантическую целостность информации.

Рекурсивное чанкирование (recursive chunking) преодолевает эту проблему путём последовательного разделения текста на логических границах. Сначала текст разделяется на параграфы, затем, если параграф остаётся слишком большим, он разбивается на предложения, и, наконец, если необходимо, применяются фиксированные размеры на символьном уровне. Этот иерархический подход сохраняет структуру документа и предотвращает разрыв смысловых единиц. Специализированные рекурсивные текстовые разделители доступны в LangChain через класс RecursiveCharacterTextSplitter, который позволяет разработчикам определять собственные последовательности разделителей в зависимости от типа документа.

На другом конце спектра находится семантическое чанкирование (semantic chunking), которое отходит от фиксированных правил и вместо этого использует смысловые границы для определения точек разделения . Семантическое чанкирование работает следующим образом: текст сначала разбивается на предложения, каждое предложение преобразуется в вектор эмбеддинга с помощью модели, затем вычисляется семантическое сходство между последовательными предложениями, и новые фрагменты создаются в точках, где сходство резко падает, указывая на переход к новой теме. Этот подход гарантирует, что каждый фрагмент содержит семантически связанную информацию, формируя единую идею или концепцию. Исследования демонстрируют, что семантическое чанкирование может повысить точность поиска на целых 40% по сравнению с фиксированным размером.

Специфичные для документа стратегии (structure-based chunking) леверируют встроенную структуру документов.

Для HTML-файлов разделение происходит по тегам, таким как <p> или <div>, сохраняя логические блоки контента. Для программного кода фрагменты выравниваются с функциями или классами, что способствует сохранению когерентности логики. Для Markdown-файлов разделение может следовать иерархии заголовков, гарантируя, что каждый раздел остаётся целостным.

Наиболее продвинутый подход — LLM-based chunking и agentic chunking — использует сами языковые модели для принятия решений о разделении. В LLM-based chunking модель обрабатывает документ и решает, как его разбить, часто добавляя выводы, резюме или теги для каждого фрагмента. Agentic chunking идёт дальше, позволяя AI-агенту динамически выбирать оптимальную стратегию разделения на основе анализа всего документа, его структуры и плотности информации. Агент может, например, определить, что файл является Markdown и использовать разделение по заголовкам, обнаружить, что другой раздел содержит плотный текст с пропозициями, и переключиться на предложенное разделение. Это гибкое, адаптивное подходит для гетерогенных корпусов документов, хотя и требует дополнительных вычислительных ресурсов.

Новый метод, называемый late chunking, инвертирует традиционный подход. Вместо чанкирования перед эмбеддингом, весь документ пропускается через долгоконтекстную модель-энкодер, генерирующую детальные, токен-уровневые эмбеддинги с полным пониманием контекста. Только затем документ разбивается на фрагменты, и эмбеддинги для каждого фрагмента вычисляются путём усреднения соответствующих токен- эмбеддингов. Этот подход гарантирует, что каждый фрагмент сохраняет глобальный контекст всего документа, потенциально повышая релевантность поиска.

Выбор оптимального размера фрагмента также требует внимательного баланса. Фрагменты слишком маленького размера могут содержать недостаточный контекст для полного понимания идеи, тогда как чрезмерно крупные фрагменты могут включать шум и несвязанную информацию, разбавляя сигнал. На практике размер фрагмента часто выбирается в диапазоне от 256 до 1024 токенов, в зависимости от природы документов и ограничений контекстного окна используемой модели. Для коротких фрагментов, таких как заголовки новостей, могут использоваться размеры 128-256 токенов, тогда как для научных статей, требующих больше контекста для понимания, выбираются размеры 512-1024 токенов.

II. Модели эмбеддингов (Encoder-модели) для ретривера

После того как документы разделены на фрагменты, наступает критический этап — преобразование каждого фрагмента и входящих запросов пользователей в плотные векторные представления. Это преобразование выполняется специализированными моделями-энкодерами, которые отображают текст произвольной длины в фиксированное числовое представление, сохраняя при этом семантическое значение.

Основной класс моделей, используемых для этой задачи, — это Sentence Transformers, семейство моделей, основанное на архитектуре BERT (Bidirectional Encoder Representations from Transformers). Базовый BERT, хотя и отличный для понимания значения отдельных токенов и слов, не оптимален для создания представлений целых предложений. При наивном усреднении токен-эмбеддингов BERT получаются низкокачественные предложенческие эмбеддинги, которые слабо отражают семантическое значение.

Sentence Transformers преодолевают эту ограничность через специализированный процесс fine-tuning BERT на задачах, связанных с пониманием предложений. Модель обучается на трёх типах целевых функций. Первая — это Natural Language Inference (NLI), где модель получает пары предложений и должна предсказать, являются ли они взаимосвязанными, противоречащими или нейтральными. Вторая задача — Semantic Textual Similarity (STS), где модель получает пару предложений и должна предсказать степень их семантического сходства как числовой коэффициент от 0 до 1. Третья — это triplet learning, где модель получает якорное предложение, положительный пример (семантически близкое) и отрицательный пример (не связанный), и обучается максимизировать расстояние между якорем и отрицательным примером при минимизации расстояния между якорем и положительным.

Архитектура Sentence Transformer использует siamese-подход — двойную сеть с общими весами. Два предложения обрабатываются параллельно через идентичные BERT-башни, выход каждой башни подвергается mean pooling (усреднение всех токен-эмбеддингов), создавая один вектор размерностью обычно 384, 768 или 1536 измерений в зависимости от используемой модели-базы. На этапе вывода, когда требуется получить эмбеддинг для нового текста, используется одна из башен для преобразования текста в вектор, который затем может быть сравнён с предварительно вычисленными векторами в базе данных.

Для RAG-приложений обычно используются предварительно fine-tuned модели, такие как all-MiniLM-L6-v2 (384 измерения), all-mpnet-base-v2 (768 измерений), или коммерческие модели от OpenAI (text-embedding-3-small, text- embedding-3-large). Выбор конкретной модели зависит от компромисса между размерностью эмбеддинга (большая размерность — выше expressiveness, но требует больше памяти), скоростью вычисления (меньшие модели работают быстрее), и стоимостью (если используются внешние API).

Критически важно понимать различие между bi-encoder архитектурой, используемой для инициального поиска, и cross-encoder архитектурой, используемой для переранжирования результатов. Bi-encoder независимо обрабатывает запрос и документы, генерируя для каждого отдельные эмбеддинги. Это позволяет предварительно вычислить эмбеддинги всех документов в базе одновременно и сохранить их, а затем быстро вычислить эмбеддинг входящего запроса и найти ближайшие соседей. Такой подход обеспечивает масштабируемость — би-кодер может работать с миллионами документов, выполняя поиск примерно за 50 миллисекунд.

Cross-encoder, с другой стороны, совместно обрабатывает пару "запрос-документ" через единый трансформер, позволяя модели захватить тонкие взаимодействия между ними. Это приводит к гораздо более точным оценкам релевантности, чем би-кодер, но за счёт вычислительной дороговизны — cross-encoder требует около 10 миллисекунд на каждую пару документ-запрос. Гибридный подход, объединяющий оба метода, стал индустриальным стандартом: би-кодер выполняет быстрый первичный поиск среди миллионов документов, извлекая, например, top-100 кандидатов, затем cross-encoder переранжирует этот сокращённый набор для получения финальных top-10. Этот двухуровневый процесс балансирует скорость и точность оптимально.

III. Векторные базы данных: принцип работы и обзор

Векторные базы данных представляют собой специализированные системы хранения, оптимизированные для эффективного индексирования и поиска высокоразмерных векторных данных. В отличие от традиционных реляционных баз данных SQL, которые оптимизированы для точного совпадения и структурированных запросов, векторные БД предназначены для быстрого поиска по подобию в пространстве с сотнями или тысячами измерений.

Ядро векторной базы данных состоит из двух ключевых компонентов: индекса и механизма поиска. При добавлении вектора в БД он проходит через фазу индексирования, где структуры индекса организуют векторы таким образом, чтобы облегчить быстрый поиск позже. Наиболее распространённые структуры индексирования — это FAISS (Facebook AI Similarity Search), HNSW (Hierarchical Navigable Small World) и IVF (Inverted File Index).

FAISS использует множество различных стратегий индексирования, от простого плоского индекса без сжатия (IndexFlat), который выполняет точный поиск ближайших соседей через полный перебор всех векторов, до более сложных структур. Для больших наборов данных FAISS предлагает product quantization — технику сжатия, которая разбивает вектор на подвекторы и кодирует каждый подвектор как целое число, значительно уменьшая требования к памяти без существенной потери точности. FAISS также поддерживает ускорение GPU, позволяя перенести вычисления на графические процессоры для ещё более быстрого поиска.

HNSW реализует иерархическое навигируемое малое мировое граф, создавая многоуровневую структуру граф, где верхние уровни содержат длинные связи, соединяющие далёкие векторы для быстрого приближения, а нижние уровни содержат локальные, плотные связи для точного поиска. Поиск в HNSW начинается на верхних уровнях, быстро перемещаясь к нужному région, а затем спускается вниз для локального поиска соседей. Это обеспечивает быстрый поиск с хорошей точностью, хотя требует больше памяти для хранения структуры графа.

IVF разделяет векторное пространство на кластеры (Voronoi partitions) и сохраняет для каждого вектора, к какому кластеру он принадлежит. При поиске система сначала находит k ближайших кластеров к вектору запроса, затем выполняет точный поиск только в этих кластерах, значительно сужая поле поиска.

Важно отметить, что выбор индекса зависит от размера данных и требуемого баланса между скоростью и памятью. FAISS в памяти (in-memory) обеспечивает оптимальную производительность на малых и средних наборах данных, но с ростом размера данных требования к памяти становятся критическим ограничением.

Исследование производительности трёх популярных векторных БД показало, что для малых наборов (10 тысяч векторов) FAISS значительно быстрее, но по мере роста до 1 миллиона и более векторов Chroma, которая сохраняет данные на диске с индексированием, становится более практичной.

Chroma — это лёгкая, разработанная для разработчиков векторная БД, специально разработанная для AI- приложений, особенно для LLM workflows. Она обеспечивает встроенную поддержку генерации эмбеддингов, хранение метаданных вместе с векторами, и простой REST API. Основное преимущество Chroma — её простота использования и быстрая интеграция с фреймворками типа LangChain и LlamaIndex. Однако она менее масштабируема для крупномасштабных систем.

Qdrant представляет собой более полнофункциональную векторную БД, предназначенную для production-среды.

Она поддерживает filtering на основе метаданных, позволяя не только поиск по подобию векторов, но и фильтрацию результатов по JSON-полям payload. Qdrant поддерживает гибридный поиск, комбинируя плотные векторные поиски с разреженными векторными методами и ключевыми словами. Архитектура Qdrant распределённая, с поддержкой sharding и replication для высокой доступности. Sharding разделяет данные на части, распределённые по разным узлам, позволяя параллельную обработку запросов.

Replication поддерживает несколько копий каждого sharding для отказоустойчивости — если один узел выходит из строя, другой автоматически берёт на себя. Это делает Qdrant подходящей для критичных к доступности систем.

Weaviate — ещё один мощный вариант, известный своей способностью выполнять поиск за миллисекунды даже на миллионах объектов. Weaviate имеет модульную архитектуру, позволяющую интегрироваться с внешними ML-сервисами, такими как OpenAI, Cohere, HuggingFace, для автоматической генерации эмбеддингов.

Это устраняет необходимость отдельно управлять моделями-энкодерами в вашем приложении.

PostgreSQL с расширением pgvector стал привлекательным вариантом для команд, которые предпочитают использовать существующую инфраструктуру. pgvector позволяет хранить векторы прямо в PostgreSQL и выполнять поиск по подобию используя встроенные операторы. Это упрощает архитектуру, избегая необходимости отдельной системы хранения для векторов.

На практике выбор векторной БД зависит от множества факторов: размер данных, требуемая задержка отклика, масштабируемость, необходимость фильтрации метаданных, наличие существующей инфраструктуры, и бюджет разработки. Для прототипирования или small-scale систем Chroma или FAISS часто являются достаточными. Для production-систем среднего масштаба Qdrant или Weaviate предлагают хороший баланс производительности и функциональности. Для высоконагруженных систем с особыми требованиями к специфичным метаданным и фильтрам может быть необходимо более специализированное решение, например, Elasticsearch с поддержкой семантического поиска.

IV. Модель-генератор (LLM) и техники контекстного промптинга

После извлечения релевантных документов наступает заключительный этап RAG-конвейера — синтез финального ответа. Этот этап выполняет генеративная языковая модель, которой предоставляется аугментированный контекст, состоящий из исходного запроса пользователя и извлечённых релевантных документов. Выбор генеративной модели критичен для качества финального ответа и определяет баланс между производительностью, стоимостью и качеством.

Генеративные модели, используемые в RAG, варьируются по размеру от относительно компактных систем в 7-13 миллиардов параметров (например, Mistral 7B, Llama 2 7B) до крупных моделей в 70-405 миллиардов параметров (GPT-4, Claude 3 Opus). Меньшие модели могут развёртываться локально на GPU-серверах с ограниченными ресурсами, обеспечивая низкую задержку и полный контроль над данными, что критично для приложений, работающих с конфиденциальной информацией. Крупные модели, доступные через облачные API, обеспечивают выше качество генерации и лучше способны рассуждать над сложными контекстами, но требуют отправки данных внешнему провайдеру и подлежат задержкам сетевого взаимодействия.

Одним из ключевых ограничений генеративных моделей является размер их контекстного окна — максимальное количество токенов (примерно 1/4 слова каждый), которые модель может обработать за один раз. Ранние версии GPT-3 имели контекстное окно всего 4096 токенов. Современные модели значительно расширили это ограничение: GPT-4-turbo поддерживает 128K токенов, а последние модели, такие как Claude 3 Opus, поддерживают 200K токенов. Однако исследования показывают, что производительность моделей деградирует при заполнении контекстного окна более чем на 50%, известный как "lost in the middle" эффект, где информация посередине контекстного окна часто игнорируется моделью. Это означает, что для модели с 128K токенов контекстного окна практическое ограничение составляет примерно 64K токенов для сохранения высокой производительности.

Процесс формирования контекстного промпта является искусством и наукой одновременно. Структурированный аугментированный промпт типично содержит несколько компонентов. Первый компонент — системная инструкция, которая определяет роль модели ("Ты — эксперт по машинному обучению") и устанавливает её поведение ("Отвечай, используя только информацию из предоставленного контекста"). Второй компонент — это метаданные о задаче и параметры, определяющие стиль и формат ответа ("Ответь коротким абзацем, не более 150 слов").

Затем идут извлечённые документы, разделённые ясными разделителями. Каждый документ может быть отформатирован с указанием его источника и степени релевантности, помогая модели оценить надежность и важность каждого фрагмента. Например, документ может быть отформатирован как:

[Документ 1 - источник: "Научная статья 2024" - релевантность: 0.89]

{текст документа}

[/Документ 1]

Исходный запрос пользователя обычно размещается в конце промпта, сразу перед маркером "Ответ:", так как последний элемент в контексте часто получает повышенное внимание модели.

За пределами простого объединения текстов находится область контекстного инжиниринга (context engineering), более широкая дисциплина программного собирания всего, что модель видит во время инференса.

Context engineering включает не только сам промпт, но и применение техник переработки информации для максимизации релевантности извлеченного контекста. Одна из таких техник — prompt compression, которая удаляет излишние слова или фразы из извлечённых фрагментов, сохраняя суть, но занимая меньше токенов контекстного окна.

Другая важная техника — перестановка контекста на базе позиции (context reordering). Учитывая "lost in the middle" эффект, наиболее релевантные документы часто помещаются в начало и конец контекстного окна, тогда как менее релевантные помещаются в середину. Это гарантирует, что критичная информация не теряется из- за позиционного смещения модели.

Chain-of-Thought (CoT) промптинг — это техника, которая явно просит модель "показать свою работу" перед генерацией финального ответа. Вместо того чтобы переходить сразу к ответу, модель генерирует промежуточные рассуждения ("Вопрос спрашивает о..., в контексте упоминается..., поэтому ответ должен быть..."), что часто приводит к более точным и обоснованным ответам. Интеграция CoT в RAG-конвейер особенно полезна при работе с многошаговыми задачами рассуждения.

Продвинутые техники включают StepBack-prompting, где модель сначала абстрагируется, переходя от конкретных примеров к более общим принципам ("Вместо того, чтобы рассуждать о конкретном случае X, какие общие принципы Y применяются?"), а затем использует эти принципы для более обоснованного рассуждения. Это особенно эффективно при работе с контекстом, содержащим конкретные детали, которые могут отвлечь модель.

Управление состоянием и памятью (state management) становится критичным в мультитурновых диалоговых системах RAG. Каждый оборот диалога должен сохранять контекст предыдущих обсуждений, но при этом оставаться в пределах контекстного окна. Стратегии включают суммаризацию длинной истории диалога в более компактное резюме, выборочное сохранение только наиболее релевантных предыдущих оборотов, и использование долгосрочной памяти (долгосрочное хранилище в отдельной БД для извлечения контекста, относящегося к конкретному пользователю).

Интеграция инструментов (tool integration) и функций вызова позволяют генеративной модели решать, когда нужны дополнительные действия. Например, при ответе на вопрос "Какова текущая цена акции Apple?", модель может решить, что необходим поиск в реальном времени, и вызвать функцию для получения актуальной цены. Правильное описание инструментов и их параметров критично для того, чтобы модель использовала их точно и в нужное время.

## §9. Преимущества и недостатки подхода RAG

Понимание компромиссов, присущих RAG, необходимо для принятия решения о том, уместно ли применение этой парадигмы для конкретного приложения. RAG не является универсальным решением, и её эффективность зависит от природы задачи, характеристик доступных данных и требований системы.

I. Преимущества RAG

**Радикальное снижение галлюцинаций и повышение фактической точности**

Наиболее значительное преимущество RAG состоит в его способности заземлять ответы модели в верифицируемые внешние источники. Исследования демонстрируют, что системы с контекстно-адаптивным синтезом снижают частоту галлюцинаций с 18.2% в базовых моделях до 6.1% при работе с противоречивой информацией. Этот скачок не является маргинальным улучшением — это трансформационное изменение надёжности системы. Вместо того, чтобы модель самостоятельно "придумывала" факты, опираясь на размытые воспоминания из тренировочных данных, система явно извлекает информацию из указанных источников и просит модель синтезировать ответ на её основе.

В высокорисковых доменах, таких как здравоохранение, юриспруденция и финансовые услуги, эта надежность имеет критическое значение. Врач, который советует пациенту лечение на основе галлюцинированной медицинской информации, или юрист, который цитирует несуществующий закон, могут причинить серьезный вред. RAG обеспечивает трассируемость ответов — каждое утверждение в ответе может быть связано с конкретным источником, позволяя пользователям и аудиторам проверить корректность информации.

**Динамическое обновление знаний без переобучения**

Традиционные языковые модели "замораживаются" в момент завершения обучения. Если появляется новая информация — свежие новости, изменённые законы, новые продукты компании — модель остаётся невежественна в отношении этой информации до следующего цикла переобучения, который может занять недели или месяцы и требует значительных вычислительных ресурсов.

RAG преодолевает это фундаментальное ограничение, делегируя хранение знаний внешним базам данных, которые обновляются независимо. Добавление нового документа в базу знаний не требует переобучения, переквантизации, дополнительного дообучения или каких-либо модификаций самой генеративной модели.

Достаточно проиндексировать новый документ векторной базой данных, и система немедленно сможет использовать эту информацию в своих ответах.

Это свойство превосходит критически важным для динамических доменов. Корпоративные системы поддержки могут обновлять политики и руководства в реальном времени. Новостные агрегаторы могут немедленно интегрировать свежую информацию. Системы финансовых рекомендаций могут адаптироваться к новым котировкам акций или изменению процентных ставок. Всё это происходит без изменения базовой модели.

**Экономическая эффективность**

Дообучение (fine-tuning) больших языковых моделей требует существенных вычислительных ресурсов. Даже эффективное дообучение компактной модели в 7 миллиардов параметров может потребовать 8 GPU V100 за несколько часов, что обходится в сотни долларов. Полное переобучение гораздо более крупной модели обойдётся в тысячи долларов. RAG предлагает более экономичную альтернативу: вместо изменения параметров модели, система просто расширяет её знания через доступ к внешним ресурсам. Затраты на обслуживание смещаются с вычислительно-интенсивного обучения на относительно дешёвые операции индексации и поиска.

Горизонтальная масштабируемость достигается легко — можно добавлять новые документы практически без ограничений, не увеличивая размер самой модели. Вертикальная масштабируемость также улучшается — можно использовать относительно компактные генеративные модели (7-13 миллиардов параметров), компенсируя их размер через мощный доступ к внешним знаниям, что снижает требования инференса.

**Повышенная персонализация и контекстуальная релевантность**

RAG позволяет адаптировать ответы специфично к контексту каждого запроса. Вместо использования обобщённых паттернов, выученных из тренировочных данных, система динамически извлекает информацию, релевантную конкретной задаче и пользователю. Корпоративный RAG-ассистент может извлекать только информацию из внутренних политик компании, гарантируя соответствие корпоративным стандартам.

Медицинская система может извлекать информацию из актуальных клинических руководств, специфичных для редкого заболевания пациента. Рекомендательная система может использовать персональный профиль пользователя для фильтрации релевантных предложений из огромного каталога.

**Прозрачность и объяснимость**

В отличие от чёрного ящика традиционных LLM, RAG предоставляет механизм для объяснения решений.

Система может указать точные источники, из которых была извлечена информация, позволяя пользователям самостоятельно верифицировать ответы. Эта трассируемость критична в регулируемых индустриях, где требуется документирование цепочки рассуждений, ведущей к решению. Аудит "почему система рекомендовала именно это" становится возможным и прозрачным.

Снижение затрат на облачные вычисления и запросы к API

Использование дорогостоящих API больших моделей (например, GPT-4 по $0.03 за 1K токенов ввода) для каждого запроса может быстро стать экономически неоправданным. RAG позволяет использовать локально развёрнутые или более дешёвые модели, экономя на затратах API.

II. Недостатки RAG

**Зависимость от качества извлечённого контекста**

RAG имеет ахиллесову пяту — проблема "garbage in, garbage out". Если система извлекает нерелевантные, устаревшие или ошибочные документы, генеративная модель, независимо от её качества, будет синтезировать неправильные ответы, часто с высокой уверенностью. Если ретривер извлекает информацию о законодательстве Франции, когда пользователь спрашивает о российском праве, модель может произвести техничный, уверенный, но фактически неправильный ответ.

Эта зависимость от качества ретривера означает, что оптимизация системы требует одновременного улучшения компонентов поиска и генерации, что усложняет процесс отладки. Если конечный ответ неправильный, необходимо определить: был ли это сбой ретривера (неправильные документы извлечены) или сбой генератора (правильные документы были неправильно интерпретированы).

**Увеличенная задержка и требования к инфраструктуре**

RAG-система требует двух последовательных дорогостоящих операций: поиска в векторной БД и генерации текста. Поиск в большой векторной БД может добавить 50-200 миллисекунд к общей задержке ответа, даже с использованием оптимизированных индексов. Добавления операций переранжирования, тематического переформулирования запроса и других оптимизаций ретривера увеличивают эту задержку дополнительно.

Для приложений, требующих реального времени, таких как голосовые помощники или высокочастотные торговые системы, эта дополнительная задержка может быть неприемлемой.

Требования к инфраструктуре также возрастают. Система должна поддерживать векторную БД (требующую памяти пропорционально размеру корпуса документов и размерности эмбеддингов), модель-энкодер (для преобразования текста в эмбеддинги), и генеративную модель. Это требует больше GPU-памяти, сетевой пропускной способности и операционных затрат, чем использование только генеративной модели.

**Повышенная сложность системы и требования к обслуживанию**

Традиционная LLM — это относительно простой компонент: пользователь предоставляет текст, модель генерирует ответ. RAG вводит множество новых элементов для управления и оптимизации: загрузчик документов, выбор и оптимизация стратегии чанкирования, модель-энкодер, векторная БД, механизмы переранжирования, управление обновлениями корпуса документов, кэширование, и мониторинг каждого компонента. Каждый компонент имеет собственные параметры конфигурации и оптимизации, и неправильная настройка любого из них может снизить производительность всей системы.

Обслуживание становится более сложным. Обновление базы документов требует переиндексации, что может занять время. Обновление модели-энкодера требует пересчёта эмбеддингов всех документов — потенциально миллиардов векторов. Изменение стратегии чанкирования требует перечанкирования всех документов и перестановки в БД. Команда разработки должна обладать экспертизой не только в области LLM и NLP, но и в управлении базами данных, инженерии баз знаний, и системном дизайне.

**Риск контекстного отравления**

Явление, известное как "контекстное отравление" (context poisoning) или "контекстный конфликт" (context clash), возникает, когда извлечённый контекст содержит противоречивую, устаревшую или вводящую в заблуждение информацию. Например, если база знаний содержит две версии одного документа с разными ответами на один и тот же вопрос, и система извлекает обе, модель может генерировать сбитый с толку или непоследовательный ответ.

Исследования показывают, что неправильно или неочищенная база знаний может фактически ухудшить производительность системы по сравнению с использованием базовой модели без RAG. Беспорядочное добавление большого количества нерелевантной информации в контекстное окно может перегрузить модель и отвлечь её от релевантной информации.

**Сложность fine-tuning для разнообразных задач**

В отличие от базовой генеративной модели, которая требует единственного цикла fine-tuning для адаптации к новой задаче, RAG требует simultaneous tuning обоих компонентов. Оптимизация ретривера (улучшение качества извлеченных документов) и генератора (улучшение качества генерации) часто требует разных стратегий и может привести к конфликтам оптимизации. Улучшение ретривера может привести к извлечению большего количества документов, что перегружает генератор. Оптимизация генератора может привести к тому, что он начнёт компенсировать плохие поиски через галлюцинации.

Когда система должна работать на нескольких различных типах задач (суммаризация, ответы на вопросы, диалог), каждая задача может требовать разных стратегий ретривала и генерации, увеличивая сложность конфигурации системы.

**Высокие вычислительные и инфраструктурные затраты**

Помимо требуемого оборудования, само предоставление сервиса обходится дороже, чем использование только генеративной модели. Каждый запрос требует компенсации затрат на поиск (computing, I/O операции на диск или сеть), переранжирование, и затем генерацию. Для высокообъёмных систем (миллионы запросов в день) даже небольшие затраты на компонент умножаются в значительные расходы.

**Проблемы с "поиском посередине"**

Как упоминалось ранее, исследования показывают, что LLM-ы склонны терять информацию, размещённую в середине контекстного окна, явление называется "потеря посередине" (lost in the middle). При использовании RAG это может быть проблемой, когда система извлекает много документов и раскладывает их в контексте. Информация в середине списка может быть проигнорирована, даже если она релевантна.

**Зависимость от качества внешних данных**

Если база знаний содержит ошибочную информацию, RAG распространит эту ошибку со скоростью света. В отличие от базовой модели, которая имеет некоторую "защиту" от фактических ошибок в виде диффузного знания из миллиардов примеров, RAG явно использует предоставленную информацию. Обслуживание чистой, правильной и актуальной базы знаний требует постоянных усилий и наблюдения за качеством.

**Компромиссы между точностью и пропускной способностью**

Оптимизация системы для низкой задержки часто требует компромисса по точности. Использование более компактных моделей-энкодеров ускоряет поиск, но может снизить качество релевантности. Сокращение количества извлекаемых документов улучшает задержку, но может пропустить релевантную информацию.

Удаление этапа переранжирования ускоряет систему, но приводит к более низкой точности ретривала.

Как парадигма "Retrieve-then-Generate", так и конкретная реализация RAG через модульные компоненты предлагают потенциальные трансформационные улучшения в надежности, актуальности и экономичности LLM- приложений. Однако эти преимущества не приходят без затрат. Увеличенная системная сложность, зависимость от качества внешних данных, и требования к значительным инфраструктурным ресурсам требуют тщательного рассмотрения перед внедрением.

Для приложений, где фактическая точность и актуальность информации являются приоритетом, преимущества RAG часто оправдывают затраты. Для систем, где низкая задержка и простота архитектуры критичны, компромиссы могут быть не приемлемы. Оптимальный подход часто заключается в тщательной оценке конкретных требований приложения, доступных ресурсов, и характеристик данных перед выбором, является ли RAG или альтернативный подход (fine-tuning,контекст модели, или гибридные комбинации) наиболее подходящим решением.

## §10. RAG-Fusion

I. Введение в RAG-Fusion

RAG-Fusion представляет собой естественную эволюцию базовой парадигмы Retrieve-then-Generate, адресующую одно из фундаментальных ограничений традиционного RAG: зависимость от единственной формулировки запроса пользователя. В то время как стандартный RAG преобразует запрос пользователя в один вектор и выполняет поиск только на основе этого единственного представления, RAG-Fusion использует генеративную модель для создания множественных альтернативных формулировок исходного запроса, каждая из которых отражает различные аспекты или перспективы пользовательского намерения.

Ключевая идея RAG-Fusion заключается в том, что разные переформулировки запроса часто извлекают различные релевантные документы из знаниевой базы. Например, вопрос "Какие меры предпринял президент во время пандемии?" может быть переформулирован как "Какие экономические политики были введены для борьбы с COVID-19?", "Какие медицинские инициативы были запущены администрацией?", и "Каков был национальный план вакцинации?". Каждая из этих переформулировок может вернуть частично пересекающиеся, но в целом комплементарные наборы документов, вместе обеспечивающие более полный контекст чем любой из них по отдельности.

II. Архитектура и процесс RAG-Fusion

RAG-Fusion использует следующую архитектуру, состоящую из пяти ключевых этапов:

Этап 1: Генерация множественных запросов

Первый и наиболее отличительный этап RAG-Fusion начинается с того, что исходный пользовательский запрос отправляется в большую языковую модель с явной инструкцией генерировать несколько альтернативных формулировок этого же вопроса. Типичное количество генерируемых запросов варьируется от 3 до 10, хотя это может быть настроено в зависимости от сложности задачи и доступных ресурсов.

Промпт для LLM может выглядеть следующим образом:

Вам дан исходный вопрос пользователя. Сгенерируйте пять альтернативных

переформулировок этого вопроса, отражающих различные аспекты или

перспективы того же самого намерения. Каждая переформулировка должна

быть существенно отличной по структуре и подходу, но семантически

эквивалентна исходному вопросу.

Исходный вопрос: "{original\_query}"

Генерируйте 5 переформулировок:

1. ...

2. ...

3. ...

4. ...

5. ...

Эта процедура критически зависит от качества LLM, выполняющей генерацию. Более мощные модели (такие как GPT-4) генерируют более разнообразные и семантически эквивалентные запросы, тогда как менее мощные модели (например, Llama 2 7B) могут производить переформулировки, которые теряют аспекты исходного намерения или, наоборот, развивают новые идеи, выходящие за пределы исходного запроса.

Этап 2: Встраивание множественных запросов

На втором этапе исходный запрос вместе со всеми генерированными альтернативными запросами преобразуется в векторные представления с использованием модели-энкодера (например, Sentence-BERT). Если было сгенерировано 5 альтернативных запросов, плюс исходный запрос, система теперь имеет 6 векторных представлений, каждое кодирующее различную перспективу пользовательского намерения.

Этап 3: Множественный поиск в векторной БД

Каждый из этих 6 векторов используется независимо для выполнения поиска ближайших соседей в векторной базе данных документов. Если система конфигурирована на извлечение top-10 документов на запрос, этот этап дает до 60 потенциальных релевантных документов (хотя многие из них будут дубликатами или сильно перекрываться).

Этап 4: Reciprocal Rank Fusion (RRF)

Критический этап RAG-Fusion — объединение результатов множественного поиска с использованием техники Reciprocal Rank Fusion (RRF). RRF — это алгоритм ранжирования, разработанный для объединения результатов из нескольких систем поиска. Вместо того чтобы просто конкатенировать результаты из всех поисков, RRF присваивает каждому документу оценку, основанную на его позиции в каждом из списков результатов.

Математическая формула RRF выглядит следующим образом:

Где:

* d — документ
* rank(d, query\_i) — позиция документа d в результатах запроса query\_i (начиная с 1)
* k — константа сглаживания, обычно устанавливаемая на 60

Пример расчёта RRF для одного документа, который появился в результатах четырёх различных запросов:

Исходный запрос: документ на позиции 2 → 1/(60+2) = 0.0154

Альтернативный запрос 1: документ на позиции 5 → 1/(60+5) = 0.0145

Альтернативный запрос 2: документ на позиции 1 → 1/(60+1) = 0.0164

Альтернативный запрос 3: не появился в топ-10 → 0

RRF\_score(документ) = 0.0154 + 0.0145 + 0.0164 + 0 = 0.0463

Документы, которые последовательно появляются высоко в результатах множественных запросов, получают существенно выше RRF-оценки, чем документы, которые появляются только в результатах одного запроса. Это гарантирует, что наиболее релевантные документы, согласованные с множественными перспективами пользовательского намерения, получают приоритет.

**Этап 5: Формирование контекста и генерация**

Переранжированные документы, теперь упорядоченные по RRF-оценкам, объединяются с исходным запросом и часто с генерированными альтернативными запросами для формирования аугментированного контекстного промпта. Языковая модель затем генерирует финальный ответ, опираясь на этот обогащённый контекст.

III. Наглядный пример RAG-Fusion

Рассмотрим конкретный пример: корпоративная система поддержки, которая ответит на вопрос о политике компании.

Исходный запрос пользователя:

"Какая у нас политика относительно удалённой работы?"

Этап 1: Генерация альтернативных запросов

LLM генерирует следующие переформулировки:

1. "Какие условия позволяют сотрудникам работать из дома?"

2. "Что говорится в коллективном договоре о возможности удалённой работы?"

3. "Какие дни в неделю разрешена работа вне офиса?"

4. "Какие требования к оборудованию и интернету для удалённой работы?"

5. "Как компания поддерживает работников, работающих удалённо?"

Этап 2: Встраивание запросов

Все шесть запросов (исходный + пять альтернативных) преобразуются в 768-мерные векторы эмбеддингов.

Этап 3: Множественный поиск

Каждый вектор используется для поиска в базе корпоративных документов. Извлекаются top-10 документов для каждого запроса:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Исходный запрос | Альтернативный 1 | Альтернативный 2 | Альтернативный 3 | Альтернативный 4 | Альтернативный 5 |
| Политика\_ДМ\_2024.pdf | Положение\_о\_ДМ.doc | Коллект\_договор\_2024.pdf | Расписание\_офиса.doc | Требования\_ИТ\_ДМ.pdf | Поддержка\_удаленных.docx |
| Правила\_доступа.doc | Расписание\_офиса.doc | Политика\_ДМ\_2024.pdf | Политика\_ДМ\_2024.pdf | Политика\_ДМ\_2024.pdf | Политика\_ДМ\_2024.pdf |
| Положение\_о\_ДМ.doc | Политика\_ДМ\_2024.pdf | Правила\_доступа.doc | Правила\_безопас.pdf | Положение\_о\_ДМ.doc | Требования\_ИТ\_ДМ.pdf |
| Требования\_ИТ\_ДМ.pdf | Требования\_ИТ\_ДМ.pdf | Положение\_о\_ДМ.doc | Политика\_ДМ\_2024.pdf | Инструкция\_VPN.pdf | Расписание\_офиса.doc |
| ... | ... | ... | ... | ... | ... |

Этап 4: Применение RRF

Теперь применяется алгоритм RRF для объединения всех результатов поиска:

Расчёт RRF-оценок:

Документ: "Политика\_ДМ\_2024.pdf"

- Исходный запрос: позиция 1 → 1/(60+1) = 0.01639

- Альтернативный 1: позиция 2 → 1/(60+2) = 0.01613

- Альтернативный 2: позиция 1 → 1/(60+1) = 0.01639

- Альтернативный 3: позиция 2 → 1/(60+2) = 0.01613

- Альтернативный 4: позиция 1 → 1/(60+1) = 0.01639

- Альтернативный 5: позиция 1 → 1/(60+1) = 0.01639

RRF\_score = 0.09782

Документ: "Положение\_о\_ДМ.doc"

- Исходный запрос: позиция 2 → 1/(60+2) = 0.01613

- Альтернативный 1: позиция 1 → 1/(60+1) = 0.01639

- Альтернативный 2: позиция 3 → 1/(60+3) = 0.01587

- Альтернативный 3: не появился → 0

- Альтернативный 4: позиция 3 → 1/(60+3) = 0.01587

- Альтернативный 5: не появился → 0

RRF\_score = 0.06426

Документ: "Требования\_ИТ\_ДМ.pdf"

- Исходный запрос: позиция 4 → 1/(60+4) = 0.01563

- Альтернативный 1: позиция 3 → 1/(60+3) = 0.01587

- Альтернативный 2: не появился → 0

- Альтернативный 3: не появился → 0

- Альтернативный 4: позиция 1 → 1/(60+1) = 0.01639

- Альтернативный 5: позиция 2 → 1/(60+2) = 0.01613

RRF\_score = 0.06402

Документ: "Правила\_доступа.doc"

- Исходный запрос: позиция 3 → 1/(60+3) = 0.01587

- Альтернативный 1: не появился → 0

- Альтернативный 2: позиция 2 → 1/(60+2) = 0.01613

- Альтернативный 3: позиция 3 → 1/(60+3) = 0.01587

- Альтернативный 4: не появился → 0

- Альтернативный 5: не появился → 0

RRF\_score = 0.04787

Переранжированные результаты по RRF-оценкам:

|  |  |  |  |
| --- | --- | --- | --- |
| **Ранг** | **Документ** | **RRF-оценка** | **Интерпретация** |
| 1 | Политика\_ДМ\_2024.pdf | 0.09782 | ✓ Максимально релевантна |
| 2 | Положение\_о\_ДМ.doc | 0.06426 | ✓ Очень релевантна |
| 3 | Требования\_ИТ\_ДМ.pdf | 0.06402 | ✓ Очень релевантна |
| 4 | Правила\_доступа.doc | 0.04787 | ✓ Релевантна |
| 5 | Расписание\_офиса.doc | 0.03854 | Менее релевантна |
| ... | ... | ... | ... |

Этап 5: Формирование контекста и генерация ответа

Формируется аугментированный промпт:

СИСТЕМНАЯ ИНСТРУКЦИЯ:

Ты — помощник по корпоративным политикам. Ответь на вопрос,

используя ТОЛЬКО информацию из предоставленных документов.

═══════════════════════════════════════════════════════════════

ИСХОДНЫЙ ВОПРОС И АЛЬТЕРНАТИВНЫЕ ПЕРСПЕКТИВЫ:

- Исходный вопрос: "Какая у нас политика относительно удалённой работы?"

- Альтернатива 1: "Какие условия позволяют сотрудникам работать из дома?"

- Альтернатива 2: "Что говорится в коллективном договоре о возможности удалённой работы?"

- Альтернатива 3: "Какие дни в неделю разрешена работа вне офиса?"

- Альтернатива 4: "Какие требования к оборудованию и интернету для удалённой работы?"

- Альтернатива 5: "Как компания поддерживает работников, работающих удалённо?"

═══════════════════════════════════════════════════════════════

КОНТЕКСТ (документы, переранжированные по RRF):

[Документ 1 — Политика\_ДМ\_2024.pdf | RRF-оценка: 0.098]

Компания признаёт преимущества гибкой работы и разрешает сотрудникам работать

удалённо до трёх дней в неделю. Сотрудники могут согласовать график удалённой

работы с их руководителем, при условии соответствия операционным требованиям.

Все удалённые работники должны иметь доступ в VPN и поддерживать связь с командой

через официальные каналы общения.

[Документ 2 — Положение\_о\_ДМ.doc | RRF-оценка: 0.064]

Удалённая работа должна выполняться из безопасного окружения с надлежащим

интернет-соединением. Сотрудники несут ответственность за обеспечение конфиденциальности

и безопасности корпоративных данных. Никаких конфиденциальных разговоров не должно

происходить в открытых местах.

[Документ 3 — Требования\_ИТ\_ДМ.pdf | RRF-оценка: 0.064]

Для удалённой работы требуется компьютер с зашифрованным диском, антивирусное

программное обеспечение, двухфакторная аутентификация и VPN-клиент корпорации.

Интернет-соединение должно быть стабильным со скоростью не менее 10 Мбит/с.

Компания предоставляет ноутбук и возмещает стоимость интернета в размере

2000 рублей в месяц.

═══════════════════════════════════════════════════════════════

ВОПРОС ПОЛЬЗОВАТЕЛЯ:

"Какая у нас политика относительно удалённой работы?"

ОТВЕТ:

На основе этого контекста, где документы выбраны и переранжированы для отражения множественных аспектов вопроса, LLM генерирует более полный и структурированный ответ:

Компания позволяет сотрудникам работать удалённо до трёх дней в неделю,

согласовав расписание с руководителем. Для удалённой работы требуются

следующие технические условия:

1. \*\*Оборудование\*\*: компьютер с шифрованием, антивирусом и

двухфакторной аутентификацией

2. \*\*Интернет\*\*: стабильное соединение минимум 10 Мбит/с

(компания возмещает до 2000 руб./мес.)

3. \*\*Безопасность\*\*: обязателен VPN-клиент корпорации, соблюдение

конфиденциальности

Сотрудники должны поддерживать коммуникацию с командой через

официальные каналы и избегать обсуждения конфиденциальной информации

в открытых местах.

Источники: Политика\_ДМ\_2024.pdf, Требования\_ИТ\_ДМ.pdf, Положение\_о\_ДМ.doc

IV. Преимущества RAG-Fusion

**Улучшенное понимание многогранного пользовательского намерения**

RAG-Fusion генерирует множественные перспективы исходного вопроса, гарантируя, что система захватывает различные фасеты пользовательского намерения. В примере выше, различные переформулировки обратили внимание на технические требования, условия использования, финансовые аспекты и правила безопасности — всё то, что может представлять интерес пользователю, спрашивающему о политике удалённой работы.

**Более высокая точность ретривала**

Использование RRF для объединения результатов множественных поисков означает, что документы, релевантные к различным интерпретациям запроса, получают приоритет. Документ, который может быть упущен при поиске по исходной формулировке (например, документ о требованиях ИТ), может быть захвачен при поиске по альтернативной переформулировке.

**Снижение галлюцинаций через контекстное обогащение**

Предоставляя LLM более полный и многогранный контекст, RAG-Fusion снижает вероятность того, что модель начнёт генерировать информацию, которой нет в предоставленных документах. С более полным набором документов, покрывающих различные аспекты темы, модель реже нуждается в том, чтобы компенсировать недостаток информации галлюцинациями.

V. Недостатки RAG-Fusion

**Повышенная задержка**

Основной компромисс RAG-Fusion — это значительно увеличенное время отклика. Если стандартный RAG требует одного обращения к LLM для ретривера и одного для генератора, RAG-Fusion требует дополнительного обращения к LLM для генерации альтернативных запросов. Исследование показало, что RAG-Fusion в среднем работает на 77% медленнее чем стандартный RAG (34.62 секунды против 19.52 секунд), с большей частью задержки, атрибутируемой дополнительному обращению к LLM.

**Риск дрейфа от исходного намерения**

Хотя генерация альтернативных запросов расширяет охват, она также вводит риск того, что некоторые генерированные запросы могут отойти от исходного пользовательского намерения. LLM может интерпретировать "далась ли помощь малому бизнесу во время COVID" как "были ли какие-либо бизнес-инициативы во время пандемии", что может вернуть нерелевантные документы о совершенно других программах.

**Увеличенные требования к ресурсам**

Система должна выполнить не менее двух обращений к LLM плюс несколько обращений к ретривалу (по одному для каждого генерированного запроса). Это требует больше GPU-памяти, сетевой пропускной способности и вычислительных ресурсов, чем стандартный RAG.

**Сложность оценки и настройки**

Оценка качества RAG-Fusion ответов сложнее чем для стандартного RAG. Традиционные метрики оценки, такие как ROUGE или BLEU, менее эффективны для задач, где приемлемые ответы могут иметь значительные вариации в структуре и содержании.

VI. Когда использовать RAG-Fusion

RAG-Fusion наиболее эффективна для следующих сценариев:

* **Сложные многогранные вопросы**, где пользовательское намерение может быть интерпретировано из различных углов
* **Критичные к точности приложения**, где выше задержка оправдана улучшенной точностью (например, медицинские или финансовые консультации)
* **Задачи, требующие компrehensive coverage**, где лучше иметь полный ответ, чем быстрый неполный
* **Внутренние системы**, где задержка менее критична, чем в публичных сервисах

RAG-Fusion представляет собой логическое расширение традиционного RAG, решая проблему зависимости от единственной формулировки запроса через генерацию множественных альтернативных интерпретаций. Использование Reciprocal Rank Fusion для объединения результатов обеспечивает научно обоснованный метод приоритизации документов, согласованных с множественными перспективами пользовательского намерения. Хотя это приводит к значительному увеличению задержки и требований к ресурсам, улучшение в точности и полноте ответов часто оправдывает эти затраты для критичных приложений.

# Глава 3. Параметрические методы адаптации: От Full Fine-Tuning к эффективным методам

## §11. Полное дообучение (Full Fine-Tuning)

I. Что такое Fine-Tuning?

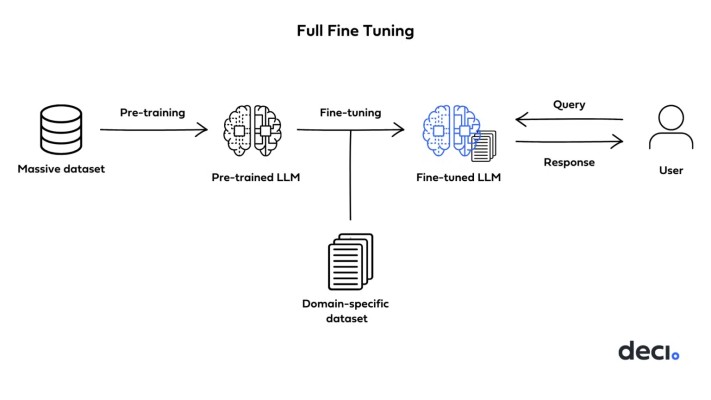
Представьте себе выпускника медицинского университета. Он обладает обширными, но общими знаниями по всем разделам медицины (базовая модель). Теперь он идет в ординатуру, чтобы стать кардиохирургом. Он не учится заново биологии или анатомии – вместо этого он фокусируется на узкой специальности, изучая специфические методики, инструменты и случаи, используя свою базу. Этот процесс специализации и есть fine-tuning.

По аналогии со студентом, в машинном обучении fine-tuning – это техника, при котором предварительно обученную модель (pre-trained model) дополнительно обучают на новом, более узком наборе данных. Говоря простым языком, это использование большого и мощного «ChatGPT» для какой-то маленькой и конкретной цели, например обработки входящих сообщений.

Изначальная модель обучается на огромном массиве данных, например, на всем интернете, для решения обширной задачи, такой как предсказание следующего слова или классификация миллионов изображений. В процессе этого обучения модель извлекает фундаментальные, низкоуровневые представления о мире — будь то грамматические конструкции и семантика языка или базовые визуальные паттерны вроде краёв и текстур. Эти представления являются универсальными, а не специфичными для исходной задачи.

Fine-tuning вступает в игру, когда у нас есть новая, более узкая задача, но недостаточно данных для обучения модели с нуля. Вместо того чтобы начинать с случайной инициализации, мы используем веса предобученной модели как отправную точку. Затем мы продолжаем процесс обучения, используя наш небольшой, но релевантный набор данных. При этом ключевой момент — это скорость обучения (learning rate). Обычно используется очень низкий learning rate, чтобы не «разрушить» те ценные общие представления, которые модель уже усвоила, а лишь аккуратно подстроить их под конкретную предметную область. Часто применяется дифференцированный подход: более низкий learning rate для ранних слоёв, которые отвечают за базовые признаки, и более высокий — для поздних, более специализированных слоёв.

Суть процесса заключается в продолжении предобучения. Модель, которая уже обладает обширными знаниями, извлеченными из огромного корпуса текстов или изображений (например, из всего интернета), не начинает обучение с нуля. Вместо этого она использует эту мощную базу как отправную точку. Процесс можно представить как высококвалифицированного специалиста широкого профиля, который проходит узкую и интенсивную переподготовку для работы в специфической области. Он не забывает базовые навыки – грамотность, логику, эрудицию, – но перестраивает и углубляет их применительно к новой предметной области.



Механика этого процесса основана на классическом обратном распространении ошибки. На каждом шаге модель получает пример из нового набора данных (например, пару "вопрос-ответ" для чат-бота или "текст-сентимент" для анализа тональности), делает прогноз и вычисляет ошибку между своим прогнозом и правильным ответом. Затем эта ошибка распространяется в обратном направлении по всем слоям нейронной сети – от выходного слоя вплоть до самого первого эмбеддинг-слоя. На основе величины градиента ошибки по каждому параметру происходит корректировка каждого веса, каждого смещения во всей архитектуре модели. Ключевой момент – обновляются абсолютно все параметры, а не только их часть. Это позволяет модели не просто "выучить" шаблоны новых данных, а переосмыслить и перекалибровать свои внутренние представления, создавая глубокие и тонкие связи между ранее усвоенными общими знаниями и новой специфической информацией.

Именно эта всеобъемлющая природа обновления порождает как главные преимущества, так и серьезные недостатки метода. Главное преимущество – потенциально высочайшее качество модели на целевой задаче. Полностью протунингованная модель может достичь уровня производительности, недоступного более легким методам, так как она имеет максимальную гибкость для адаптации своей архитектуры под nuances и сложности новых данных. Она по-настоящему "перепрошивает" себя, чтобы стать экспертом в данной области.

Таким образом, полный тонкий тунинг — это инструмент высшей лиги, глубокое преобразование модели, которое стирает грань между предобучением и дообучением. Это стратегический выбор, оправданный когда требуется максимально возможная производительность, а целевая задача фундаментально отличается от исходных данных предобучения или является исключительно сложной. В современных подходах его часто дополняют или заменяют более эффективными методами, такими как LoRA (Low-Rank Adaptation), но он остается золотым стандартом и краеугольным камнем в понимании того, как большие языковые модели адаптируются к миру конкретных задач.

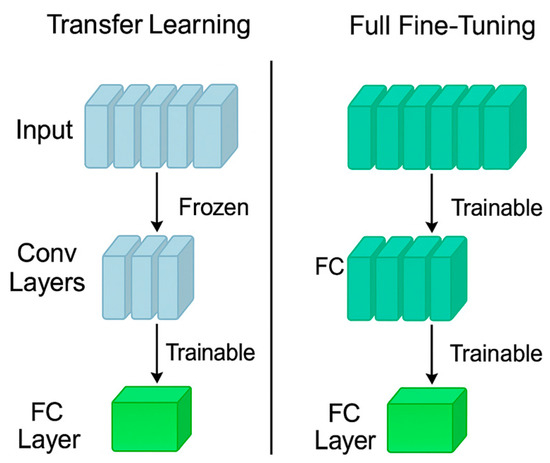
Как уже было отмечено ранее, Fine-Tuning (дообучение) является общим и фундаментальным подходом к адаптации предобученных языковых моделей под конкретные задачи и домены. Важно понимать, что это не единственный унифицированный метод, а скорее обширная парадигма, внутри которой существует целый спектр техник, различающихся по своей эффективности, требовательности к ресурсам и применимости.

Этот спектр включает в себя:

**Full Fine-Tuning (Полное дообучение):** метод, при котором обновляются все без исключения параметры исходной модели.

**Partial Fine-Tuning** **(Частичное дообучение):** методы, при которых обновляется только часть модели (например, несколько верхних слоев).

**Parameter-Efficient Fine-Tuning (PEFT — Эффективное дообучение):** современный класс методов, который включает в себя такие техники, как LoRA (Low-Rank Adaptation), Adapter Tuning, Prompt Tuning и другие. Эти методы добавляют в модель небольшое количество обучаемых параметров, оставляя исходные веса модели "замороженными", что радикально снижает вычислительные затраты и затраты на память.



В данной главе мы сфокусируемся на Full Fine-Tuning — самом прямолинейном и ресурсоемком, но при этом классическом и фундаментальном для понимания методе. Его изучение позволяет заложить базовые принципы, которые в дальнейшем помогут понять эволюцию и преимущества более эффективных техник.

II Обучение с учителем на задаче генерации текста

Чтобы понять современный подход к тонкой настройке языковых моделей, необходимо провести важное различие между двумя парадигмами обучения: **дообучением (continued pre-training)** и **обучением с учителем (Supervised Fine-Tuning, SFT)**.

**Дообучение vs. Обучение с учителем: концептуальное различие**

Дообучение следует логике оригинального предобучения модели — это обучение без учителя, где модель обучается предсказывать следующее слово в последовательности на больших объемах текстов из целевой предметной области. Например, если мы хотим адаптировать общую модель для медицинских задач, мы можем "докормить" ее медицинскими статьями, учебниками и клиническими записями. Модель улучшает свое понимание медицинской терминологии, стиля и контекста, но не обучается целенаправленно отвечать на вопросы или выполнять инструкции.

Обучение с учителем (SFT) — это принципиально иной подход. Здесь мы имеем четкую структуру "вход-выход", где:

* **Вход (input):** Промпт, содержащий инструкцию, контекст, примеры
* **Выход (target):** Ожидаемый идеальный ответ

С формальной точки зрения, SFT для генерации текста максимизирует условную вероятность целевой последовательности при заданном входном промтe:

где:

* – входной промпт;
* – целевая последовательность токенов;
* – датасет пар (промпт, ответ);
* – параметры модели.

**Почему именно обучение с учителем для генерации текста?**

1. **Контролируемое поведение:** Мы можем точно направлять модель к желаемому поведению, формату вывода и стилю общения.
2. **Следование инструкциям:** Модель учится не просто продолжать текст, а анализировать инструкцию и выполнять ее.
3. **Предсказуемость:** Ответы становятся более структурированными и соответствующими ожиданиям.
4. **Безопасность:** Мы можем обучать модель избегать вредоносных или неэтичных ответов.

## §12. Подготовка датасета для дообучения: инструкции, промпты, ожидаемые ответы

Ранние подходы к дообучению часто заключались в простом «докорме» модели большими объемами текстов из целевой предметной области (например, медицинскими статьями или юридическими документами). Этот подход, известный как продолженное предобучение (continued pre-training), полезен для адаптации словаря и стиля модели, но он неэффективен для обучения модели следовать инструкциям или отвечать на вопросы.

Современный Fine-Tuning для задач генерации текста — это, по сути, обучение с учителем (Supervised Fine-Tuning, SFT). Мы показываем модели примеры «вопрос-ответ» или «инструкция-результат». Таким образом, наш датасет должен состоять из пар (input, target\_output), где:

* Input (Вход): Это промпт, который может включать в себя инструкцию, контекст, примеры и т.д.
* Target Output (Целевой выход): Это идеальный, ожидаемый от модели ответ на данный промпт.

Давайте разберем каждый компонент этой пары.

**Инструкция (Instruction)**

Инструкция — это ясное и недвусмысленное описание задачи, которую должна выполнить модель. Это команда, которая переводит модель из общего режима «предскажи следующее слово» в режим «выполни конкретное действие».

Что делает хорошую инструкцию?

* **Конкретность и ясность:** Избегайте расплывчатых формулировок.
  + *Плохо:* «Напиши про собак».
  + *Хорошо:* «Напиши краткий информационный параграф (150-200 слов) о породе собак «сибирский хаски», описав ее историю, характер и особенности ухода».
* **Задание формата:** Явно укажите, в каком формате вы ожидаете ответ.
  + *Примеры:* «...ответь одним предложением», «...представь ответ в виде маркированного списка», «...выведи JSON-объект с полями name, email, phone».
* **Указание роли (Role-Playing):** Назначение модели определенной роли часто значительно улучшает качество ответа.
  + *Пример:* «Ты — опытный финансовый аналитик. Объясни понятие «инфляция» своему 10-летнему племяннику».
* **Учет тона и стиля:** Определите, каким должен быть тон ответа.
  + *Пример:* «Ответь официальным и профессиональным тоном», «Напиши легкий и шутливый текст для соцсетей».

Пример инструкции:

«Ты — помощник по подбору книг. Пользователь интересуется научной фантастикой. На основе его предпочтений, предложи три книги и кратко (1-2 предложения) объясни, почему каждая из них может ему понравиться. Ответ представь в виде нумерованного списка.»

**Промпт (Prompt)**

Промпт — это более широкое понятие. В контексте нашего датасета, промпт — это *полный входной сигнал*, который мы подаем на вход модели. Он может состоять не только из инструкции, но и из дополнительных элементов:

* **Контекст (Context):** Информация, необходимая для выполнения задачи.
  + *Пример:* «Контекст: Компания «А» объявила о выпуске нового смартфона с камерой на 200 Мп. Инструкция: Напиши короткий пресс-релиз об этом событии.».
* **Входные данные (Input Data):** Непосредственные данные для обработки.
  + *Пример:* «Инструкция: Перефразируй следующее предложение. Входные данные: Быстрая коричневая лиса прыгает через ленивую собаку».
* **Примеры «немного-shot» (Few-Shot Examples):** Включение нескольких готовых примеров «вход-выход» прямо в промпт помогает модели понять шаблон без обновления ее весов. Это мощный прием, особенно когда данных для тонкой настройки мало.
  + *Пример (One-Shot):*

Преврати ключевые слова в предложение.

Ключевые слова: [погода, дождь, зонт]

Предложение: на улице испортилась погода, пошел сильный дождь, поэтому не забудь взять зонт.

Структура полного промпта может выглядеть так (на примере классификации тональности):

<|instruction|>

Определи тональность следующего отзыва: положительная, отрицательная или нейтральная.

<|context|>

Отзыв: «Этот фильм — потрясающее кино с великолепной актерской игрой и захватывающим сюжетом.»

<|end|>

*(Здесь мы используем специальные токены для разметки структуры, что помогает модели лучше парсить входные данные.)*

**Ожидаемый ответ (Target Output)**

Это «золотой стандарт» — идеальный ответ, который вы хотите видеть от модели на данный промпт. Создание качественных ожидаемых ответов — это искусство.

Требования к ожидаемому ответу:

* **Корректность:** Ответ должен быть фактически и грамматически верным.
* **Полнота:** Ответ должен полностью и точно выполнять инструкцию. Если в инструкции просили три пункта, в ответе должно быть ровно три.
* **Соответствие формату:** если вы просили JSON, ответ должен быть валидным JSON. Если просили список, это должен быть список.
* **Непротиворечивость:** Стиль, тон и глубина ответов должны быть согласованы по всему датасету. Если один ответ развернутый и формальный, а другой — краткий и разговорный, модель запутается.
* **Исключение предвосхищения (Avoiding Bias):** Ответ не должен «подсказывать» модельке следующую часть промпта. В задаче классификации ответ должен быть просто меткой («положительный»), а не полным предложением, которое может перекрываться с промптом.

1. «Дюна» Фрэнка Герберта. Эта эпопея погрузит тебя в сложный мир межпланетной политики, экологии и мистики, с масштабным сюжетом и глубокой проработкой мира.

2. «Задача трёх тел» Лю Цысиня. Книга предлагает уникальный взгляд на контакт с внеземной цивилизацией, основанный на жесткой научной фантастике и невероятных концепциях.

3. «Снежная катастрофа» Нил Стивенсон. Сочетание киберпанка, исторических деталей и философских размышлений о виртуальности и реальности.

Процесс подготовки датасета: Пошаговое руководство

1. **Определение задачи и метрик:** Четко сформулируйте, что должна уметь модель после дообучения. Какие метрики вы будете использовать для оценки успеха (например, BLEU, ROUGE, человеческая оценка, точность выполнения задачи)?
2. **Сбор сырых данных:** Источники могут быть разными:
   * Публичные датасеты (например, из репозиториев вроде Hugging Face Datasets).
   * Внутренние данные компании (логи чатов, базы знаний, документы).
   * Генерация данных с помощью более мощных моделей (например, GPT-4).
   * **Краудсорсинг и аутсорсинг:** Наем аннотаторов для создания примеров.
3. **Создание шаблонов промптов:** Разработайте несколько шаблонов для ваших инструкций и промптов. Это обеспечит разнообразие и покрытие разных аспектов задачи.
   * Шаблон 1 (простая инструкция): [Инструкция]: {instruction}
   * \*Шаблон 2 (инструкция + контекст):\* [Роль]: {role}. [Контекст]: {context}. [Инструкция]: {instruction}
   * \*Шаблон 3 (few-shot):\* Включите в шаблон место для примеров.
4. **Генерация пар (промпт, ответ):** Это ядро процесса.
   * **Аннотация человеком:** Специалисты (эксперты предметной области) вручную пишут идеальные ответы на сгенерированные промпты. Это самый качественный, но и самый дорогой метод.
   * **Самозаключение (Self-Instruct):** Использование самой модели (например, GPT-4) для генерации как промптов, так и ответов с последующей валидацией и фильтрацией человеком. Это позволяет быстро создать большой объем данных.
   * **Итеративный процесс:** Часто используется гибридный подход: модель генерирует черновики, а человек их исправляет, редактирует и утверждает.
5. **Очистка и предобработка:**
   * Удаление личной информации (PII).
   * Исправление опечаток и грамматических ошибок (особенно в ожидаемых ответах!).
   * Приведение к единому формату (регистр, знаки препинания).
   * Проверка на соответствие длины (обрезание слишком длинных последовательностей).
6. **Разделение на выборки:**
   * **Обучающая выборка (Training, ~80-90%):** На этих данных модель будет непосредственно обучаться.
   * **Валидационная выборка (Validation, ~5-10%):** Используется для мониторинга процесса обучения, подбора гиперпараметров и ранней остановки (early stopping).
   * **Тестовая выборка (Test, ~5-10%):** Независимый набор для финальной оценки производительности модели. Важно: на этих данных модель не должна обучаться ни на каком этапе.
7. **Форматирование данных:** Приведите данные к формату, понятному для вашего фреймворка обучения (например, JSONL для библиотек Hugging Face).

*// Пример записи в JSONL-файле*

{

**"prompt": "Ты помощник... Объясни понятие 'квантовая запутанность'.",**

**"completion": "Квантовая запутанность — это квантовомеханическое явление, при котором..."**

**}**

{

"prompt": "Напиши email-ответ клиенту, который жалуется на задержку доставки...",

"completion": "Уважаемый Иван Иванович, Приносим наши извинения..."

}

Распространенные ошибки и подводные камни

* **Несогласованность:** Разные аннотаторы по-разному понимают задачу. Решение: создайте детальные гайдлайны для аннотаторов с множеством примеров.
* **Низкое качество «золотых» ответов:** Если ожидаемые ответы содержат ошибки или неполны, модель научится этим ошибкам. «Мусор на входе — мусор на выходе» (Garbage In, Garbage Out).
* **Слишком простые или сложные промпты:** Датасет должен отражать реальные запросы, которые будет получать модель. Если все промпты идеально составлены, модель может плохо работать с неидеальными пользовательскими запросами.
* **Перекос в данных (Bias):** Датасет может непреднамеренно содержать социальные, культурные или фактологические предубеждения, которые модель усвоит и усилит.
* **Утечка данных (Data Leakage):** Случайное попадание тестовых данных в обучающую выборку приведет к завышенным и нереалистичным метрикам.

Подготовка датасета для Full Fine-Tuning — это не просто технический этап, это стратегическая инвестиция в успех вашего проекта. Время, потраченное на тщательное проектирование инструкций, создание качественных промптов и, что самое важное, написание безупречных ожидаемых ответов, окупится сторицей в виде надежной, точной и управляемой модели. Помните: вы не просто собираете данные, вы создаете учебник, по которому будет учиться модель. Сделайте этот учебник ясным, последовательным и содержательным.

## §13. Токенизация и паддинг батчей для эффективного обучения

Прежде чем модель сможет обрабатывать текст, его необходимо преобразовать в числовой формат, понятный нейронной сети. Процесс **токенизации** и организация данных в **батчи** — это критически важные этапы, которые напрямую влияют на качество обучения, вычислительную эффективность и стабильность всего процесса тонкой настройки.

Когда мы работаем с большими языковыми моделями, важно понимать, что сырой текст — это лишь последовательность символов, не имеющая смысла для нейронной сети. Преобразование этого текста в числовые представления — первый и один из самых важных шагов в пайплайне обработки данных. Неправильная токенизация может не только замедлить обучение, но и fundamentally исказить смысл исходного текста, направив модель по неверному пути.

Давайте рассмотрим конкретный пример того, как различные стратегии токенизации обрабатывают один и тот же текст, чтобы понять их сильные и слабые стороны.

****I. Токенизация: От текста к числам****

Токенизация — это процесс разбиения текста на меньшие единицы (токены) с последующим преобразованием их в числовые идентификаторы. Выбор стратегии токенизации во многом определяет, насколько хорошо модель сможет понимать и генерировать текст на конкретном языке.

В современном машинном обучении существует несколько основных подходов к токенизации, каждый из которых имеет свои особенности применения. Словесная токенизация хорошо подходит для языков с четкими границами слов, в то время как символьная может быть полезна для языков с иероглифической письменностью или при работе с текстами, содержащими много редких слов.

**Основные стратегии токенизации:**

1. **Word-based (Словесная)**

Текст: "I don't like NLP"

Токены: ["I", "don't", "like", "NLP"]

* + *Плюсы:* Простота интерпретации
  + *Минусы:* Большой словарь, проблемы с OOV (out-of-vocabulary)

1. **Character-based (Символьная)**

Текст: "cat"

Токены: ["c", "a", "t"]

* + Плюсы: Маленький словарь, нет OOV
  + Минусы: Длинные последовательности, сложнее выучить семантику

1. **Subword-based (Субсловная)** — **золотой стандарт для современных LLM**

Текст: "unhappily"

Токены: ["un", "happ", "ily"]

**Популярные алгоритмы субсловной токенизации:**

**Byte-Pair Encoding (BPE) - используется в GPT, RoBERTa**

*# Алгоритм обучения:*

1. Инициализировать словарь символами

2. Посчитать частоты пар символов

3. Объединять наиболее частые пары

4. Повторять до достижения целевого размера словаря

*# Пример:*

Текст: "low lower newest"

Частоты: lo-2, w-2, e-3, r-2, n-1, s-1, t-1

Первое слияние: "lo" + "w" → "low" (теперь есть токен "low")

**WordPiece - используется в BERT**

*# Отличие от BPE: объединяет пары, которые максимизируют вероятность данных*

Score("ab") = freq("ab") / (freq("a") \* freq("b")))

**SentencePiece - используется в T5, LLaMA**

*# Особенности:*

- Работает напрямую с сырым текстом (без предварительной нормализации)

- Может токенизировать пробелы (▁)

- Не зависит от языка

Однако в реальных задачах мы редко работаем с простыми примерами вручную. Современные библиотеки предоставляют готовые инструменты для токенизации, которые инкапсулируют сложные алгоритмы и позволяют сосредоточиться на более важных аспектах обучения модели.

Пример токенизации в Hugging Face Transformer

Библиотека Transformers от Hugging Face стала стандартом де-факто для работы с языковыми моделями. Она предоставляет унифицированный интерфейс для работы с сотнями предобученных моделей и их токенизаторами. Давайте рассмотрим, как выглядит базовый процесс токенизации с использованием этой библиотеки.

Важно отметить, что разные модели могут использовать разные алгоритмы токенизации и разные словари, поэтому всегда следует использовать токенизатор, соответствующий конкретной модели, которую вы планируете дообучать.

from transformers import AutoTokenizer

*# Загрузка токенизатора*

tokenizer = AutoTokenizer.from\_pretrained("meta-llama/Llama-2-7b-hf")

*# Базовая токенизация*

text = "Полное дообучение модели требует тщательной подготовки данных."

tokens = tokenizer.tokenize(text)

*# ['▁Пол', 'ное', '▁дооб', 'учение', '▁модели', '▁требует', ...]*

*# Преобразование в идентификаторы*

input\_ids = tokenizer.encode(text)

*# [1, 2456, 789, 1234, 567, 890, ...]*

*# Обратная токенизация*

decoded\_text = tokenizer.decode(input\_ids)

**Специальные токены и их роль:**

Специальные токены играют crucial роль в работе языковых моделей, так как они маркируют начало и конец последовательностей, разделяют различные части входа и обеспечивают корректную работу механизма паддинга. В некоторых случаях может потребоваться явно добавить или настроить специальные токены для конкретной задачи.

В процессе токенизации особое внимание следует уделять специальным токенам, которые выполняют служебные функции в работе языковых моделей. Эти токены помогают модели понимать структуру входных данных, разделять различные части текста и корректно обрабатывать последовательности переменной длины.

Каждый специальный токен имеет свое назначение: BOS (beginning of sequence) отмечает начало последовательности, EOS (end of sequence) сигнализирует о завершении, PAD используется для выравнивания длин, а UNK заменяет неизвестные слова. Правильная настройка этих токенов особенно важна при дообучении моделей на специфических задачах.

*# Добавление специальных токенов (если не добавлены автоматически)*

tokenizer.add\_special\_tokens({

'bos\_token': '<s>', *# beginning of sequence*

'eos\_token': '</s>', *# end of sequence*

'pad\_token': '<pad>', *# padding*

'unk\_token': '<unk>', *# unknown*

})

*# Пример последовательности с специальными токенами:*

*# <s> Текст промпта </s> Текст ответа </s>*

prompt = "Объясни теорию относительности"

response = "Теория относительности — это..."

full\_sequence = f"<s>{prompt}</s>{response}</s>"

II. Проблема переменной длины и решение через паддинг

Одной из фундаментальных проблем при работе с текстовыми данными является переменная длина последовательностей. В отличие от изображений, которые можно легко привести к единому размеру, тексты естественным образом имеют разную длину, что создает challenges для эффективной батч-обработки в нейронных сетях.

Нейронные сети оперируют тензорами фиксированной размерности, в то время как текстовые последовательности по своей природе имеют переменную длину. Это противоречие требует специальных механизмов для приведения данных к единому формату, и именно здесь на помощь приходит техника паддинга.

Рассмотрим наглядный пример, демонстрирующий проблему и ее решение:

Текст 1: "Привет" → 3 токена

Текст 2: "Полное дообучение требует данных" → 7 токенов

Текст 3: "Fine-tuning with padding" → 5 токенов

Нейронные сети требуют **тензоров фиксированной размерности**, но тексты имеют **переменную длину**.

**Решение: Паддинг (дополнение)**

*# До паддинга:*

[["Привет"], ["Полное", "дообучение", "требует", "данных"], ["Fine", "tuning", "with", "padding"]]

*# После паддинга (длина = 7):*

[["Привет", "<pad>", "<pad>", "<pad>", "<pad>", "<pad>", "<pad>"],

["Полное", "дообучение", "требует", "данных", "<pad>", "<pad>", "<pad>"],

["Fine", "tuning", "with", "padding", "<pad>", "<pad>", "<pad>"]]

Выбор стратегии паддинга — это компромисс между эффективностью использования памяти и вычислительной производительностью. Различные подходы предлагают разные trade-offs, и понимание их особенностей позволяет выбрать оптимальную стратегию для конкретной задачи и hardware configuration.

Статический паддинг предполагает приведение всех последовательностей к заранее заданной фиксированной длине. Этот подход прост в реализации и обеспечивает стабильность обучения, но может быть неэффективен с точки зрения использования памяти, особенно когда распределение длин текстов сильно варьируется.

**1. Static Padding (Статический паддинг)**

from transformers import DataCollatorWithPadding

data\_collator = DataCollatorWithPadding(

tokenizer=tokenizer,

padding=True,

max\_length=512 *# фиксированная длина*

)

*# Преимущества: простота, стабильность*

*# Недостатки: неэффективное использование памяти для коротких текстов*

**2. Dynamic Padding (Динамический паддинг)**

В противоположность статическому, динамический паддинг предполагает дополнение последовательностей только до максимальной длины в пределах каждого отдельного батча. Это позволяет более эффективно использовать доступную память, но может создавать дополнительную вариативность в процессе обучения из-за разной длины последовательностей между батчами.

data\_collator = DataCollatorWithPadding(

tokenizer=tokenizer,

padding=True,

max\_length=None *# паддинг до максимальной длины в батче*

)

*# Пример батча:*

Батч 1: max\_length=128 → все последовательности дополняются до 128

Батч 2: max\_length=256 → все последовательности дополняются до 256

*# Преимущества: экономия памяти, ускорение обучения*

*# Недостатки: разная длина между батчами*

**3. Bucketed Padding (Паддинг с группировкой)**

Для задач, где важна как эффективность, так и стабильность обучения, часто применяется техника группированного паддинга (bucketed padding). Этот подход сочетает в себе преимущества статического и динамического паддинга, группируя примеры схожей длины перед формированием батчей.

*# Группировка примеров по длине перед созданием батчей*

длины = [45, 123, 47, 128, 46, 124, 44]

группы = [[44, 45, 46, 47], [123, 124, 128]]

*# Преимущества: баланс между эффективностью и стабильностью*

*# Недостатки: сложность реализации*

III. Маски внимания (Attention Masks)

Механизм внимания — cornerstone современных трансформерных архитектур, но он требует специальной обработки в случае паддинга. Паддинг-токены, будучи искусственными добавлениями, не должны влиять на вычисления механизма внимания и расчет функции потерь. Именно для этой цели были разработаны маски внимания.

Маска внимания — это бинарный тензор, который указывает модели, какие токены являются реальными данными, а какие — паддингом. Это позволяет механизму внимания фокусироваться только на значимых частях последовательности и игнорировать искусственно добавленные паддинг-токены.

*# Пример input\_ids и соответствующей маски внимания*

input\_ids = [

[1, 234, 567, 890, 0, 0, 0], *# 0 - pad token*

[1, 123, 456, 789, 012, 345, 678]

]

attention\_mask = [

[1, 1, 1, 1, 0, 0, 0], *# 1 - реальный токен, 0 - pad*

[1, 1, 1, 1, 1, 1, 1]

]

*# При передаче в модель*

outputs = model(

input\_ids=input\_ids,

attention\_mask=attention\_mask,

labels=labels

)

На практике маски внимания автоматически создаются токенизатором и коллатором данных, но понимание их работы необходимо для отладки и оптимизации процесса обучения. Особенно это важно при реализации custom training loops или работе с нестандартными архитектурами.

IV. Оптимизация размера батча

Выбор оптимального размера батча — это искусство балансировки между скоростью сходимости, качеством модели и ограничениями hardware. Слишком маленькие батчи могут привести к шумным градиентам и медленной сходимости, в то время как слишком большие — исчерпать доступную память и потребовать уменьшения learning rate.

Для понимания требований к памяти полезно иметь представление о базовой формуле расчета, которая учитывает размер модели, активаций и оптимизатора. Это позволяет заранее оценить feasibility обучения на конкретном hardware.

Память ≈ (размер\_модели + размер\_активаций + размер\_оптимизатора) × batch\_size

Где:

- размер\_активаций ≈ seq\_len × hidden\_size × batch\_size × коэффициенты

- размер\_оптимизатора ≈ 2 × размер\_модели (для Adam)

На практике часто используется эвристический подход к определению максимально возможного размера батча, который заключается в постепенном увеличении размера до возникновения ошибки нехватки памяти с последующим откатом к последнему рабочему значению

*# Поиск максимального размера батча методом "проб и ошибок"*

batch\_size = 1

while not out\_of\_memory:

try:

batch\_size \*= 2

train\_batch(batch\_size) *# Пробуем обучить с удвоенным размером батча*

except CUDA\_out\_of\_memory:

batch\_size //= 2 *# Возвращаемся к предыдущему рабочему значению*

break

print(f"Оптимальный размер батча: {batch\_size}")

Для ситуаций, когда даже максимально возможный размер батча недостаточен для достижения желаемого effective batch size, существует техника градиентного накопления. Этот подход позволяет имитировать большой размер батча, накапливая градиенты на нескольких небольших батчах перед обновлением весов.

V. Полный пайплайн подготовки батчей

В реальных проектах подготовка данных для обучения представляет собой комплексный процесс, который включает токенизацию, паддинг, создание масок и формирование батчей. Организация этого процесса в виде воспроизводимого и эффективного пайплайна — ключ к успешному обучению моделей.

Рассмотрим пример законченного класса для обработки данных, который инкапсулирует всю логику подготовки данных для дообучения seq2seq моделей. Такой подход обеспечивает модульность и возможность повторного использования кода.

from transformers import DataCollatorForSeq2Seq

from torch.utils.data import DataLoader

class FineTuningDataProcessor:

def \_\_init\_\_(self, tokenizer, max\_length=512):

self.tokenizer = tokenizer

self.max\_length = max\_length

*# Инициализация коллатора для seq2seq задач*

self.data\_collator = DataCollatorForSeq2Seq(

tokenizer=tokenizer,

padding=True,

max\_length=max\_length,

return\_tensors="pt"

)

def tokenize\_function(self, examples):

*# Токенизация промптов*

model\_inputs = self.tokenizer(

examples["prompt"],

truncation=True,

max\_length=self.max\_length,

)

*# Токенизация целевых последовательностей (ответов)*

with self.tokenizer.as\_target\_tokenizer():

labels = self.tokenizer(

examples["completion"],

truncation=True,

max\_length=self.max\_length,

)

model\_inputs["labels"] = labels["input\_ids"]

return model\_inputs

def prepare\_dataloader(self, dataset, batch\_size=8):

*# Применяем токенизацию ко всему датасету*

tokenized\_dataset = dataset.map(

self.tokenize\_function,

batched=True, *# Обработка батчами для эффективности*

remove\_columns=dataset.column\_names *# Удаляем исходные колонки*

)

*# Создаем DataLoader с нашим коллатором*

dataloader = DataLoader(

tokenized\_dataset,

batch\_size=batch\_size,

collate\_fn=self.data\_collator, *# Используем наш коллатор для паддинга*

shuffle=True *# Перемешиваем данные для обучения*

)

return dataloader

Такой подход обеспечивает четкое разделение ответственности: класс обработчика данных занимается всей логикой подготовки, в то время как основной training loop работает с уже готовыми батчами. Это значительно упрощает отладку и модификацию пайплайна обработки данных.

VI. Best Practices и распространенные ошибки

Накопленный опыт сообщества машинного обучения позволил выявить набор лучших практик и типичных ошибок, связанных с токенизацией и подготовкой батчей. Следование этим рекомендациям может сэкономить значительное время и ресурсы при дообучении моделей.

Среди наиболее важных рекомендаций — использование динамического паддинга для повышения эффективности, обязательное применение масок внимания, группировка примеров по длине для минимизации паддинга и кэширование токенизированных данных для ускорения последующих эпох обучения.

**✅ Рекомендуется:**

1. **Использовать динамический паддинг** для эффективного использования памяти
2. **Всегда использовать маски внимания** при работе с паддингом
3. **Группировать примеры по длине** для минимизации избыточного паддинга
4. **Кэшировать токенизированные данные** для ускорения обучения
5. **Проверять декодирование** для обеспечения корректности токенизации

С другой стороны, существует набор типичных ошибок, которые часто совершают как начинающие, так и опытные практики. К ним относятся забывание масок внимания, игнорирование специальных токенов при расчете длины последовательностей и слишком агрессивное обрезание текстов.

**❌ Распространенные ошибки:**

1. **Забывать маски внимания** — паддинг-токены влияют на вычисления
2. **Не учитывать special tokens** при расчете длины последовательности
3. **Слишком агрессивное обрезание** (truncation) — потеря важного контекста
4. **Несогласованность токенизации** между обучением и инференсом
5. **Игнорирование памяти GPU** — слишком большой размер батча

Эффективный мониторинг процесса подготовки данных позволяет выявить проблемы на ранних этапах и обеспечить качество входных данных для модели. Регулярная проверка статистик батчей, визуализация распределения длин последовательностей и выборочная верификация декодирования — необходимые компоненты надежного пайплайна.

Для отладки полезно иметь функцию, которая анализирует свойства батча и предоставляет ключевую информацию о распределении длин, проценте паддинга и качестве токенизации. Это помогает быстро идентифицировать аномалии в данных.

def analyze\_batch(batch, tokenizer):

"""Анализ свойств батча для отладки и мониторинга"""

print(f"Размер батча: {batch['input\_ids'].shape}")

*# Анализ длин последовательностей*

seq\_lengths = batch['attention\_mask'].sum(dim=1)

print(f"Длины последовательностей: {seq\_lengths.tolist()}")

print(f"Минимальная длина: {seq\_lengths.min().item()}")

print(f"Максимальная длина: {seq\_lengths.max().item()}")

print(f"Средняя длина: {seq\_lengths.float().mean().item():.2f}")

*# Анализ эффективности паддинга*

total\_tokens = batch['attention\_mask'].numel()

real\_tokens = batch['attention\_mask'].sum().item()

padding\_percentage = (1 - real\_tokens / total\_tokens) \* 100

print(f"Процент паддинга: {padding\_percentage:.2f}%")

*# Пример декодирования для визуальной проверки*

print("\nПример первого элемента в батче:")

print(tokenizer.decode(batch['input\_ids'][0]))

*# Проверка соответствия labels и input\_ids*

if 'labels' in batch:

print("\nПроверка labels:")

non\_padding\_labels = batch['labels'][batch['labels'] != -100]

print(f"Не-padding tokens в labels: {len(non\_padding\_labels)}")

*# Использование функции анализа*

analyze\_batch(batch, tokenizer)

Регулярное применение таких инструментов мониторинга позволяет поддерживать высокое качество данных на протяжении всего процесса обучения и оперативно реагировать на возникающие проблемы.

Правильная токенизация и организация батчей — это не просто технические детали реализации, а фундаментальные аспекты, определяющие успех всего процесса дообучения языковых моделей. Оптимальные стратегии обработки текстовых данных позволяют не только увеличить скорость обучения на 20-50%, но и существенно снизить потребление памяти, улучшить стабильность процесса обучения и максимально эффективно использовать доступные вычислительные ресурсы.

Понимание тонкостей работы с текстовыми данными — от выбора алгоритма токенизации до оптимизации стратегий паддинга — является essential навыком для любого практика в области машинного обучения. Эти знания служат прочным фундаментом для перехода к более сложным методам адаптации и тонкой настройки больших языковых моделей под конкретные прикладные задачи.

## §14. Процесс обучения: функция потерь, оптимизаторы, планировщики скорости обучения

Процесс обучения больших языковых моделей представляет собой сложный и тонко настраиваемый механизм, где каждый компонент играет критически важную роль. Понимание того, как взаимодействуют функция потерь, оптимизаторы и планировщики скорости обучения, позволяет не только эффективно проводить дообучение, но и диагностировать проблемы, адаптировать стратегии под конкретные задачи и hardware-ограничения.

Обучение нейронной сети — это по своей сути процесс оптимизации, где мы ищем такие параметры модели, которые минимизируют ошибку предсказания на обучающих данных. В контексте языковых моделей эта ошибка измеряется через функцию потерь, а сам процесс поиска оптимальных параметров осуществляется с помощью оптимизаторов, которые, в свою очередь, управляются планировщиками скорости обучения.

Каждый из этих компонентов имеет свою теоретическую основу и практические особенности настройки. Давайте детально рассмотрим каждый из них, начиная с фундаментального понятия — функции потерь.

I. Функция потерь (Loss Function) для генерации текста

Функция потерь — это математическая мера того, насколько предсказания модели отличаются от реальных целевых значений. В задаче генерации текста мы имеем дело с последовательностями переменной длины, что требует специального подхода к вычислению потерь.

**Теоретические основы Negative Log-Likelihood Loss**

Основная идея заключается в том, чтобы максимизировать вероятность правильной последовательности токенов. Поскольку мы работаем с вероятностями, которые являются числами между 0 и 1, и поскольку вероятности независимых событий перемножаются, мы используем логарифмическую функцию для преобразования произведения в сумму. Это не только вычислительно удобно, но и помогает избежать проблем с численной устойчивостью при работе с очень маленькими вероятностями.

Математически, для последовательности токенов  при условии контекста, мы хотим максимизировать:

Что эквивалентно минимизации отрицательного логарифма правдоподобия:

На практике это реализуется через функцию cross-entropy loss, которая является стандартным выбором для задач классификации, к которой сводится предсказание следующего токена.

**Практическая реализация функции потерь**

**Давайте рассмотрим, как эта теория воплощается в коде. В PyTorch мы можем использовать встроенную функцию cross-entropy, но важно правильно обработать паддинг-токены, чтобы они не вносили вклад в вычисление потерь.**

import torch

import torch.nn.functional as F

def calculate\_language\_modeling\_loss(model\_output, targets, attention\_mask=None):

"""

Вычисление функции потерь для языкового моделирования

Args:

model\_output: выход модели [batch\_size, seq\_len, vocab\_size]

targets: целевые токены [batch\_size, seq\_len]

attention\_mask: маска внимания [batch\_size, seq\_len]

"""

logits = model\_output.logits

batch\_size, seq\_len, vocab\_size = logits.shape

*# Переносим logits и targets в правильную форму для cross\_entropy*

logits\_flat = logits.view(-1, vocab\_size)

targets\_flat = targets.view(-1)

*# Вычисляем cross entropy loss*

loss = F.cross\_entropy(logits\_flat, targets\_flat, reduction='none')

*# Применяем mask к loss (игнорируем паддинг)*

if attention\_mask is not None:

mask\_flat = attention\_mask.view(-1)

loss = loss \* mask\_flat

total\_loss = loss.sum() / mask\_flat.sum()

else:

total\_loss = loss.mean()

return total\_loss

**Практический пример вычисления потерь**

Рассмотрим конкретный пример с небольшими последовательностями, чтобы понять, как вычисляются потери на каждом шаге генерации.

*# Пример: батч из одной последовательности*

vocab = {"<pad>": 0, "я": 1, "люблю": 2, "машинное": 3, "обучение": 4, "!": 5}

*# Входные данные (уже токенизированные)*

input\_ids = torch.tensor([[1, 2, 3, 4, 5]]) *# "я люблю машинное обучение !"*

labels = torch.tensor([[2, 3, 4, 5, 0]]) *# Сдвинутые на 1 влево target'ы*

*# Предположим, что модель выдала следующие логиты:*

logits = torch.tensor([

[[-1.0, 3.0, 0.5, -0.5, 0.1, -2.0], *# для "я" -> должна предсказать "люблю" (индекс 2)*

[0.5, -1.0, 2.5, -0.5, 0.3, -1.0], *# для "люблю" -> "машинное" (индекс 3)*

[-0.5, 0.1, -1.0, 3.0, 0.5, -0.5], *# для "машинное" -> "обучение" (индекс 4)*

[0.3, -0.5, 0.1, -1.0, 2.5, -1.0], *# для "обучение" -> "!" (индекс 5)*

[0.1, 0.2, 0.3, 0.4, 0.5, 0.6]] *# для "!" -> паддинг (игнорируется)*

])

*# Вычисляем потери*

loss\_fn = torch.nn.CrossEntropyLoss(ignore\_index=0) *# игнорируем паддинг (index 0)*

loss = loss\_fn(logits.view(-1, 6), labels.view(-1))

print(f"Loss: {loss.item():.4f}")

**Расширенные варианты функций потерь**

В некоторых сценариях стандартной функции потерь может быть недостаточно. Например, когда определенные типы токенов или определенные позиции в последовательности более важны для нашей задачи. В таких случаях мы можем использовать взвешенные варианты функции потерь.

class WeightedLanguageModelingLoss:

def \_\_init\_\_(self, ignore\_index=-100, weights=None):

self.ignore\_index = ignore\_index

self.weights = weights

def \_\_call\_\_(self, logits, labels, position\_weights=None):

loss = F.cross\_entropy(

logits.view(-1, logits.size(-1)),

labels.view(-1),

ignore\_index=self.ignore\_index,

reduction='none'

)

*# Применяем веса к различным позициям в последовательности*

if position\_weights is not None:

loss = loss \* position\_weights.view(-1)

*# Применяем веса классов (например, для несбалансированных данных)*

if self.weights is not None:

class\_weights = self.weights[labels.view(-1)]

loss = loss \* class\_weights

return loss.mean()

*# Пример использования взвешенной функции потерь*

weights = torch.tensor([1.0, 1.0, 1.0, 1.0, 1.0, 1.0]) *# веса для каждого класса*

position\_weights = torch.tensor([1.0, 1.0, 1.5, 1.5, 1.0]) *# больше веса середине последовательности*

weighted\_loss\_fn = WeightedLanguageModelingLoss(weights=weights)

loss = weighted\_loss\_fn(logits, labels, position\_weights)

II. Оптимизаторы (Optimizers)

Оптимизаторы отвечают за обновление параметров модели в направлении, которое минимизирует функцию потерь. Выбор оптимизатора и его гиперпараметров существенно влияет на скорость сходимости и качество конечной модели.

Исторически оптимизаторы прошли путь от простого стохастического градиентного спуска (SGD) до сложных адаптивных методов like Adam и AdamW. Каждый следующий алгоритм пытался решить проблемы предыдущих: SGD страдал от медленной сходимости и застревания в локальных минимумах, Momentum добавил инерцию, Adam ввел адаптивные learning rates для каждого параметра, а AdamW исправил проблему с weight decay.

**AdamW: современный стандарт для дообучения LLM**

AdamW стал де-факто стандартом для обучения больших языковых моделей благодаря своей устойчивости и эффективности. Основное отличие от классического Adam заключается в правильном разделении весового затухания (weight decay) и learning rate. В Adam weight decay смешивался с адаптивными learning rates, что могло приводить к неоптимальному поведению, особенно при длительном обучении.

import torch.optim as optim

from transformers import AdamW

*# Инициализация AdamW оптимизатора*

def initialize\_optimizer(model, learning\_rate=1e-5, weight\_decay=0.01):

"""

Инициализация оптимизатора AdamW с настройками для дообучения LLM

"""

*# Разделение параметров на те, у которых есть weight decay и нет*

no\_decay = ["bias", "LayerNorm.weight", "layer\_norm.weight"]

optimizer\_grouped\_parameters = [

{

"params": [p for n, p in model.named\_parameters()

if not any(nd in n for nd in no\_decay)],

"weight\_decay": weight\_decay,

},

{

"params": [p for n, p in model.named\_parameters()

if any(nd in n for nd in no\_decay)],

"weight\_decay": 0.0,

},

]

optimizer = AdamW(

optimizer\_grouped\_parameters,

lr=learning\_rate,

betas=(0.9, 0.999),

eps=1e-8,

correct\_bias=True *# Важно для воспроизводимости*

)

return optimizer

*# Пример использования*

model = ... *# наша языковая модель*

optimizer = initialize\_optimizer(model, learning\_rate=2e-5, weight\_decay=0.01)

**Сравнение различных оптимизаторов**

Чтобы понять, почему AdamW стал таким популярным, полезно сравнить его с другими оптимизаторами. Каждый из них имеет свои сильные и слабые стороны, и в разных сценариях могут быть предпочтительны разные алгоритмы.

class OptimizerComparison:

def \_\_init\_\_(self, model, learning\_rate=1e-3):

self.model = model

self.optimizers = {

'SGD': optim.SGD(model.parameters(), lr=learning\_rate),

'SGD\_with\_momentum': optim.SGD(model.parameters(), lr=learning\_rate, momentum=0.9),

'Adam': optim.Adam(model.parameters(), lr=learning\_rate),

'AdamW': AdamW(model.parameters(), lr=learning\_rate, weight\_decay=0.01),

'Adagrad': optim.Adagrad(model.parameters(), lr=learning\_rate)

}

def train\_step(self, batch, optimizer\_name):

optimizer = self.optimizers[optimizer\_name]

optimizer.zero\_grad()

*# Прямой проход*

outputs = self.model(\*\*batch)

loss = outputs.loss

*# Обратный проход*

loss.backward()

optimizer.step()

return loss.item()

*# На практике для дообучения больших моделей почти всегда используется AdamW*

*# из-за его устойчивости к выбору learning rate и хороших свойств сходимости*

**Градиентное накопление и скейлинг loss**

При работе с большими моделями часто возникает ситуация, когда мы не можем использовать достаточно большой размер батча из-за ограничений памяти. В таких случаях помогает техника градиентного накопления, которая позволяет имитировать большой размер батча, накапливая градиенты на нескольких маленьких батчах перед обновлением весов.

class GradientAccumulationTrainer:

def \_\_init\_\_(self, model, optimizer, accumulation\_steps=4):

self.model = model

self.optimizer = optimizer

self.accumulation\_steps = accumulation\_steps

self.scaler = torch.cuda.amp.GradScaler() *# для mixed precision*

def training\_step(self, batch, step\_idx):

*# Mixed precision forward pass*

with torch.cuda.amp.autocast():

outputs = self.model(\*\*batch)

loss = outputs.loss

*# Масштабируем loss для градиентного накопления*

scaled\_loss = loss / self.accumulation\_steps

*# Backward pass с масштабированием градиентов*

self.scaler.scale(scaled\_loss).backward()

*# Обновляем веса только после accumulation\_steps шагов*

if (step\_idx + 1) % self.accumulation\_steps == 0:

self.scaler.step(self.optimizer)

self.scaler.update()

self.optimizer.zero\_grad()

return loss.item()

*# Пример использования*

trainer = GradientAccumulationTrainer(model, optimizer, accumulation\_steps=4)

for step, batch in enumerate(dataloader):

loss = trainer.training\_step(batch, step)

III. Планировщики скорости обучения (Learning Rate Schedulers)

Планировщики скорости обучения управляют изменением learning rate в процессе тренировки. Правильно выбранный scheduling может значительно ускорить сходимость и улучшить итоговое качество модели.

**Теория обучения нейронных сетей и важность динамического learning rate**

Исследования показывают, что использование постоянного learning rate редко является оптимальной стратегией. В начале обучения большой LR может помочь быстро выйти из плохих областей пространства параметров, но затем он может вызывать колебания вокруг оптимума. Маленький LR, наоборот, обеспечивает стабильную сходимость, но может быть слишком медленным в начале.

**Linear Warmup with Linear Decay — золотой стандарт для LLM**

Это одна из самых популярных стратегий для дообучения языковых моделей. Она включает плавный разогрев learning rate в начале обучения (что особенно важно при использовании предобученных весов) и постепенное уменьшение towards the end, чтобы обеспечить стабильную сходимость.

from transformers import get\_linear\_schedule\_with\_warmup

def initialize\_scheduler(optimizer, num\_training\_steps, warmup\_ratio=0.1):

"""

Инициализация планировщика с линейным разогревом и линейным затуханием

"""

num\_warmup\_steps = int(num\_training\_steps \* warmup\_ratio)

scheduler = get\_linear\_schedule\_with\_warmup(

optimizer,

num\_warmup\_steps=num\_warmup\_steps,

num\_training\_steps=num\_training\_steps

)

return scheduler

*# Пример использования в тренировочном цикле*

num\_epochs = 3

total\_steps = len(dataloader) \* num\_epochs

optimizer = initialize\_optimizer(model)

scheduler = initialize\_scheduler(optimizer, total\_steps, warmup\_ratio=0.1)

for epoch in range(num\_epochs):

for step, batch in enumerate(dataloader):

*# Прямой и обратный проход*

loss = model(\*\*batch).loss

loss.backward()

*# Шаг оптимизатора*

optimizer.step()

scheduler.step() *# Обновляем learning rate*

optimizer.zero\_grad()

*# Логируем learning rate*

if step % 100 == 0:

current\_lr = scheduler.get\_last\_lr()[0]

print(f"Step {step}, LR: {current\_lr:.2e}, Loss: {loss.item():.4f}")

**Cosine Annealing with Warm Restarts для преодоления локальных минимумов**

Эта стратегия особенно полезна когда модель застревает в локальных минимумах. Периодические "перезапуски" learning rate помогают модели выпрыгнуть из плохих областей пространства параметров. Каждый перезапуск начинается с относительно высокого LR, который затем постепенно уменьшается по косинусоидальному закону.

from transformers import get\_cosine\_schedule\_with\_warmup

def initialize\_cosine\_scheduler(optimizer, num\_training\_steps,

num\_cycles=0.5, warmup\_ratio=0.1):

"""

Косинусное затухание с перезапусками

"""

num\_warmup\_steps = int(num\_training\_steps \* warmup\_ratio)

scheduler = get\_cosine\_schedule\_with\_warmup(

optimizer,

num\_warmup\_steps=num\_warmup\_steps,

num\_training\_steps=num\_training\_steps,

num\_cycles=num\_cycles

)

return scheduler

*# Сравнение различных стратегий scheduling*

class LearningRateStrategies:

def \_\_init\_\_(self, optimizer, total\_steps):

self.schedulers = {

'constant': get\_constant\_schedule(optimizer),

'linear': get\_linear\_schedule\_with\_warmup(

optimizer,

num\_warmup\_steps=int(0.1 \* total\_steps),

num\_training\_steps=total\_steps

),

'cosine': get\_cosine\_schedule\_with\_warmup(

optimizer,

num\_warmup\_steps=int(0.1 \* total\_steps),

num\_training\_steps=total\_steps,

num\_cycles=0.5

)

}

def get\_lr(self, strategy\_name, step):

scheduler = self.schedulers[strategy\_name]

*# Для получения текущего LR делаем фиктивный шаг*

if step > 0:

scheduler.step()

return scheduler.get\_last\_lr()[0]

**Адаптивные стратегии обучения**

Некоторые планировщики автоматически адаптируются к прогрессу обучения, изменяя learning rate на основе метрик качества. Например, ReduceLROnPlateau уменьшает LR, когда потеря на валидации перестает улучшаться в течение заданного количества эпох.

from torch.optim.lr\_scheduler import ReduceLROnPlateau

class AdaptiveTrainingManager:

def \_\_init\_\_(self, optimizer, model, patience=3, factor=0.5):

self.optimizer = optimizer

self.model = model

self.scheduler = ReduceLROnPlateau(

optimizer,

mode='min', *# Минимизируем loss*

patience=patience, *# Количество эпох без улучшения*

factor=factor, *# Фактор уменьшения LR*

verbose=True

)

self.best\_loss = float('inf')

def step(self, current\_loss, epoch):

*# Сохраняем модель если loss улучшился*

if current\_loss < self.best\_loss:

self.best\_loss = current\_loss

torch.save(self.model.state\_dict(), f'best\_model\_epoch\_{epoch}.pt')

*# Обновляем learning rate на основе текущего loss*

self.scheduler.step(current\_loss)

*# Возвращаем текущий learning rate*

current\_lr = self.optimizer.param\_groups[0]['lr']

return current\_lr

*# Пример использования адаптивного планировщика*

training\_manager = AdaptiveTrainingManager(optimizer, model)

for epoch in range(num\_epochs):

epoch\_loss = 0

for batch in dataloader:

loss = model(\*\*batch).loss

loss.backward()

optimizer.step()

optimizer.zero\_grad()

epoch\_loss += loss.item()

avg\_epoch\_loss = epoch\_loss / len(dataloader)

current\_lr = training\_manager.step(avg\_epoch\_loss, epoch)

print(f"Epoch {epoch}, Avg Loss: {avg\_epoch\_loss:.4f}, LR: {current\_lr:.2e}")

IV. Интеграция всех компонентов: полный тренировочный цикл

Теперь, когда мы рассмотрели все компоненты по отдельности, пришло время объединить их в единый тренировочный цикл. Такой подход демонстрирует best practices для дообучения больших языковых моделей и включает все современные техники оптимизации.

Современный тренировочный пайплайн для LLM включает не только базовые компоненты, но и дополнительные техники like mixed precision training, gradient clipping, и комплексный мониторинг. Это позволяет максимизировать эффективность использования hardware и качество конечной модели.

class CompleteFineTuningTrainer:

def \_\_init\_\_(self, model, train\_dataloader, val\_dataloader,

learning\_rate=2e-5, warmup\_ratio=0.1, weight\_decay=0.01):

self.model = model

self.train\_dataloader = train\_dataloader

self.val\_dataloader = val\_dataloader

*# Инициализация оптимизатора*

self.optimizer = initialize\_optimizer(model, learning\_rate, weight\_decay)

*# Расчет общего количества шагов*

num\_epochs = 3

self.total\_steps = len(train\_dataloader) \* num\_epochs

*# Инициализация планировщика*

self.scheduler = initialize\_scheduler(

self.optimizer, self.total\_steps, warmup\_ratio

)

*# Для mixed precision training*

self.scaler = torch.cuda.amp.GradScaler()

*# Трекинг метрик*

self.train\_losses = []

self.val\_losses = []

self.learning\_rates = []

def train\_epoch(self, epoch):

self.model.train()

total\_loss = 0

progress\_bar = tqdm(self.train\_dataloader, desc=f"Epoch {epoch}")

for step, batch in enumerate(progress\_bar):

*# Forward pass с mixed precision*

with torch.cuda.amp.autocast():

outputs = self.model(\*\*batch)

loss = outputs.loss

*# Backward pass с масштабированием градиентов*

self.scaler.scale(loss).backward()

*# Gradient clipping для предотвращения взрыва градиентов*

self.scaler.unscale\_(self.optimizer)

torch.nn.utils.clip\_grad\_norm\_(self.model.parameters(), max\_norm=1.0)

*# Шаг оптимизатора и планировщика*

self.scaler.step(self.optimizer)

self.scaler.update()

self.scheduler.step()

self.optimizer.zero\_grad()

*# Логирование*

total\_loss += loss.item()

current\_lr = self.scheduler.get\_last\_lr()[0]

if step % 10 == 0:

progress\_bar.set\_postfix({

'loss': f'{loss.item():.4f}',

'lr': f'{current\_lr:.2e}'

})

self.learning\_rates.append(current\_lr)

avg\_loss = total\_loss / len(self.train\_dataloader)

self.train\_losses.append(avg\_loss)

return avg\_loss

def validate(self, epoch):

self.model.eval()

total\_loss = 0

with torch.no\_grad():

for batch in self.val\_dataloader:

with torch.cuda.amp.autocast():

outputs = self.model(\*\*batch)

loss = outputs.loss

total\_loss += loss.item()

avg\_loss = total\_loss / len(self.val\_dataloader)

self.val\_losses.append(avg\_loss)

print(f"Validation Loss: {avg\_loss:.4f}")

return avg\_loss

def train(self, num\_epochs=3):

for epoch in range(num\_epochs):

print(f"\n=== Epoch {epoch + 1}/{num\_epochs} ===")

train\_loss = self.train\_epoch(epoch)

val\_loss = self.validate(epoch)

print(f"Epoch {epoch + 1}: Train Loss = {train\_loss:.4f}, "

f"Val Loss = {val\_loss:.4f}")

*# Сохранение чекпоинта*

if val\_loss == min(self.val\_losses):

torch.save(self.model.state\_dict(), f'best\_model\_epoch\_{epoch}.pt')

return self.train\_losses, self.val\_losses, self.learning\_rates

*# Запуск обучения*

trainer = CompleteFineTuningTrainer(model, train\_dataloader, val\_dataloader)

train\_losses, val\_losses, learning\_rates = trainer.train(num\_epochs=3)

V. Мониторинг и визуализация процесса обучения

Эффективный мониторинг позволяет вовремя обнаруживать проблемы и принимать обоснованные решения о продолжении или остановке обучения. Визуализация ключевых метрик помогает понять динамику обучения и выявить потенциальные проблемы.

Хорошая система мониторинга должна отслеживать не только основные метрики like loss и accuracy, но и производные показатели like gradient norms, learning rate динамику, и распределение активаций. Это дает полную картину того, что происходит во время обучения.

import matplotlib.pyplot as plt

import numpy as np

class TrainingVisualizer:

def \_\_init\_\_(self):

self.fig, self.axes = plt.subplots(2, 2, figsize=(15, 10))

def plot\_training\_progress(self, train\_losses, val\_losses, learning\_rates, gradients=None):

*# График потерь*

self.axes[0, 0].plot(train\_losses, label='Train Loss', color='blue')

self.axes[0, 0].plot(val\_losses, label='Val Loss', color='red')

self.axes[0, 0].set\_title('Training and Validation Loss')

self.axes[0, 0].set\_xlabel('Epoch')

self.axes[0, 0].set\_ylabel('Loss')

self.axes[0, 0].legend()

self.axes[0, 0].grid(True)

*# График learning rate*

self.axes[0, 1].plot(learning\_rates, color='green')

self.axes[0, 1].set\_title('Learning Rate Schedule')

self.axes[0, 1].set\_xlabel('Step')

self.axes[0, 1].set\_ylabel('Learning Rate')

self.axes[0, 1].set\_yscale('log')

self.axes[0, 1].grid(True)

*# График отношения train/val loss*

ratio = [t/v if v != 0 else 0 for t, v in zip(train\_losses, val\_losses)]

self.axes[1, 0].plot(ratio, color='purple')

self.axes[1, 0].set\_title('Train/Val Loss Ratio')

self.axes[1, 0].set\_xlabel('Epoch')

self.axes[1, 0].set\_ylabel('Ratio')

self.axes[1, 0].axhline(y=1.0, color='red', linestyle='--', alpha=0.5)

self.axes[1, 0].grid(True)

*# График градиентов (если предоставлены)*

if gradients is not None:

self.axes[1, 1].hist(gradients, bins=50, alpha=0.7, color='orange')

self.axes[1, 1].set\_title('Gradient Distribution')

self.axes[1, 1].set\_xlabel('Gradient Value')

self.axes[1, 1].set\_ylabel('Frequency')

self.axes[1, 1].grid(True)

plt.tight\_layout()

plt.show()

*# Использование визуализатора*

visualizer = TrainingVisualizer()

visualizer.plot\_training\_progress(train\_losses, val\_losses, learning\_rates)

Процесс обучения больших языковых моделей — это сложная, но хорошо изученная область, где правильная настройка каждого компонента может значительно повлиять на конечный результат. Функция потерь определяет, что именно мы оптимизируем, оптимизаторы отвечают за эффективный поиск оптимальных параметров, а планировщики скорости обучения управляют динамикой этого процесса.

Ключевые insights для успешного дообучения:

* Используйте **CrossEntropyLoss с игнорированием паддинга** для задач генерации текста
* **AdamW с разделением weight decay** показывает наилучшие результаты для LLM
* **Linear warmup + linear decay** — надежная стратегия scheduling для большинства задач
* **Градиентное накопление** позволяет обойти ограничения памяти без потери качества
* **Регулярный мониторинг** метрик помогает вовремя обнаруживать проблемы

Понимание этих принципов и умение адаптировать их под конкретные задачи и ограничения – важный навык для любого практика в области машинного обучения, работающего с большими языковыми моделями.

## §15. Анализ рисков: катастрофическое забывание, переобучение, смещение модели

Процесс дообучения больших языковых моделей сопряжен с рядом фундаментальных рисков, которые могут существенно снизить эффективность модели или даже сделать ее непригодной для практического использования. Понимание этих рисков и методов их mitigation является критически важным для любого специалиста, работающего с тонкой настройкой нейронных сетей.

Когда мы дообучаем предобученную модель на новых данных, мы фактически вмешиваемся в сложную систему знаний, которая была приобретена в ходе предварительного обучения на огромных объемах текстов. Это вмешательство может привести к нескольким негативным сценариям: модель может забыть ранее полученные знания (катастрофическое забывание), чрезмерно подстроиться под ограниченный набор новых данных (переобучение) или унаследовать и усилить смещения, присутствующие в данных дообучения.

Давайте детально рассмотрим каждый из этих рисков, их причины, последствия и стратегии предотвращения.

I. Катастрофическое забывание

Катастрофическое забывание — это фундаментальная проблема непрерывного обучения нейронных сетей, когда модель, обучаясь на новых данных, теряет способность выполнять задачи, которые она успешно решала ранее. В контексте дообучения языковых моделей это проявляется в том, что модель забывает общие языковые паттерны, фактические знания и reasoning способности, приобретенные в ходе предварительного обучения.

Механизм катастрофического забывания связан с тем, как градиентный спуск обновляет веса сети. Когда мы минимизируем loss на новых данных, веса изменяются в направлении, которое оптимально для новой задачи, но это изменение может "перезаписать" веса, важные для предыдущих задач. Особенно уязвимыми оказываются нижние слои трансформеров, которые кодируют фундаментальные лингвистические паттерны.

Факторы, усиливающие катастрофическое забывание

* **Слишком высокий learning rate**: Агрессивное обновление весов быстро разрушает ранее приобретенные знания
* **Несбалансированный датасет**: Преобладание узкоспециализированных данных над общими текстами
* **Слишком долгое обучение**: Продолжительная оптимизация на новых данных смещает модель слишком далеко от исходной точки
* **Неадекватный размер батча**: Слишком маленькие батчи создают шумные градиенты, которые хаотично изменяют веса

II. Стратегии предотвращения катастрофического забывания

**1. Постепенная разморозка слоев (Gradual Unfreezing)**

Вместо одновременного обучения всех слоев модели, мы постепенно "размораживаем" слои, начиная с верхних и постепенно переходя к нижним. Это позволяет сохранить общие представления в нижних слоях, пока верхние слои адаптируются к новой задаче.

def gradual\_unfreezing(model, current\_epoch, total\_epochs):

"""Постепенная разморозка слоев в течение обучения"""

total\_layers = len(list(model.parameters()))

*# Вычисляем, сколько слоев нужно разморозить на текущей эпохе*

layers\_to\_unfreeze = int((current\_epoch / total\_epochs) \* total\_layers)

*# Замораживаем все слои сначала*

for param in model.parameters():

param.requires\_grad = False

*# Размораживаем необходимое количество верхних слоев*

layers\_unfrozen = 0

for name, param in reversed(list(model.named\_parameters())):

if layers\_unfrozen < layers\_to\_unfreeze:

param.requires\_grad = True

layers\_unfrozen += 1

**2. Elastic Weight Consolidation (EWC)**

EWC добавляет регуляризационный член к функции потерь, который "защищает" важные веса от значительных изменений. Важность весов определяется на основе их вклада в предыдущие задачи.

class EWCRegularizer:

def \_\_init\_\_(self, model, fisher\_matrix, previous\_params, lambda\_ewc=1000):

self.model = model

self.fisher\_matrix = fisher\_matrix *# Матрица Фишера - мера важности параметров*

self.previous\_params = previous\_params *# Предыдущие значения параметров*

self.lambda\_ewc = lambda\_ewc

def compute\_penalty(self):

penalty = 0

for name, param in self.model.named\_parameters():

if name in self.fisher\_matrix:

*# Штрафуем за отклонение от важных параметров*

penalty += (self.fisher\_matrix[name] \*

(param - self.previous\_params[name])\*\*2).sum()

return self.lambda\_ewc \* penalty

**3. Learning rate разогрев и осторожное планирование**

Использование очень маленького learning rate в начале обучения с постепенным увеличением позволяет модели плавно адаптироваться к новым данным без резкого изменения важных весов.

III. Переобучение (Overfitting)

**Диагностика и природа переобучения в больших языковых моделях**

Переобучение возникает, когда модель слишком closely подстраивается под особенности обучающих данных и теряет способность к обобщению на новые, ранее не виденные примеры. В контексте LLM переобучение проявляется в:

* **Запоминании конкретных примеров** из обучающего датасета
* **Снижении разнообразия генерации** - модель начинает выдавать шаблонные ответы
* **Ухудшении производительности** на validation set при продолжении обучения
* **Потере креативности и гибкости** в решении задач

**Факторы, способствующие переобучению:**

* **Недостаточный объем данных** для дообучения
* **Слишком сложная модель** относительно сложности задачи
* **Чрезмерно долгое обучение** без proper validation
* **Отсутствие регуляризации** в процессе обучения
* **Низкое качество данных** с шумом и артефактами

**Методы борьбы с переобучением**

**1. Ранняя остановка (Early Stopping):** Мониторинг производительности на validation set и остановка обучения, когда производительность перестает улучшаться.

class EarlyStopping:

def \_\_init\_\_(self, patience=5, min\_delta=0.001):

self.patience = patience

self.min\_delta = min\_delta

self.counter = 0

self.best\_loss = None

self.early\_stop = False

def \_\_call\_\_(self, val\_loss):

if self.best\_loss is None:

self.best\_loss = val\_loss

elif val\_loss > self.best\_loss - self.min\_delta:

self.counter += 1

if self.counter >= self.patience:

self.early\_stop = True

else:

self.best\_loss = val\_loss

self.counter = 0

**2. Регуляризация весов (Weight Regularization)**  
Добавление L1 или L2 регуляризации к функции потерь для предотвращения слишком больших значений весов.

**3. Dropout и Stochastic Depth**  
Использование методов случайного "выключения" частей сети during training для предотвращения ко-адаптации нейронов.

**4. Data Augmentation**  
Увеличение разнообразия обучающих данных через аугментацию текстов.

class TextAugmentation:

def \_\_init\_\_(self):

self.synonym\_replace = True

self.random\_deletion = True

self.random\_swap = True

def augment\_text(self, text, augmentation\_factor=0.1):

words = text.split()

augmented\_texts = []

*# Создаем несколько аугментированных версий*

for \_ in range(3):

aug\_words = words.copy()

if self.synonym\_replace and random.random() < augmentation\_factor:

*# Замена синонимами (упрощенная версия)*

pass

if self.random\_deletion and random.random() < augmentation\_factor:

*# Случайное удаление слов*

aug\_words = [w for w in aug\_words if random.random() > 0.1]

augmented\_texts.append(' '.join(aug\_words))

return augmented\_texts

IV. Смещение модели (Model Bias)

Смещение модели — это систематическая ошибка, которая приводит к необъективным или дискриминационным предсказаниям. В контексте LLM смещения могут проявляться в различных формах:

**1. Статистическое смещение (Statistical Bias)**  
Возникает из-за нерепрезентативности обучающих данных. Например, если модель обучалась преимущественно на текстах с определенной культурной или гендерной перспективой.

**2. Смещение подтверждения (Confirmation Bias)**  
Модель усиливает существующие стереотипы и предубеждения, присутствующие в данных.

**3. Смещение отбора (Selection Bias)**  
Некорректная выборка данных для дообучения, которая не отражает реальное распределение целевой domain.

**4. Смещение алгоритма (Algorithmic Bias)**  
Смещения, вносимые самим алгоритмом обучения или архитектурой модели.

**Методы выявления и измерения смещений**

**1. Аудит модели на bias-тестах**  
Создание специализированных тестовых наборов для выявления различных типов смещений.

class BiasAudit:

def \_\_init\_\_(self, model, tokenizer):

self.model = model

self.tokenizer = tokenizer

self.bias\_tests = {

'gender\_bias': self.test\_gender\_bias,

'racial\_bias': self.test\_racial\_bias,

'cultural\_bias': self.test\_cultural\_bias

}

def test\_gender\_bias(self, test\_cases):

"""Тестирование гендерных смещений"""

biases = []

for prompt, neutral\_expected in test\_cases:

*# Замена гендерных маркеров*

male\_prompt = prompt.replace('[GENDER]', 'мужчина')

female\_prompt = prompt.replace('[GENDER]', 'женщина')

male\_output = self.generate\_text(male\_prompt)

female\_output = self.generate\_text(female\_prompt)

*# Анализ различий в тональности и содержании*

bias\_score = self.analyze\_bias\_difference(male\_output, female\_output)

biases.append(bias\_score)

return np.mean(biases)

**2. Анализ представлений (Representation Analysis)**  
Исследование внутренних представлений модели для выявления скрытых смещений.

**3. Человеческая оценка (Human Evaluation)**  
Привлечение экспертов для оценки объективности и справедливости выводов модели.

**Стратегии уменьшения смещений**

**1. Балансировка датасета**  
Сознательное включение разнообразных перспектив и точек зрения в данные для дообучения.

**2. Adversarial Debiasing**  
Использование adversarial подхода для удаления чувствительной информации из внутренних представлений модели.

**3. Контролируемая тонкая настройка (Controlled Fine-Tuning)**  
Использование контролирующих сигналов и ограничений в процессе обучения для предотвращения нежелательного поведения.

V. Интегрированный подход к управлению рисками

**Многоуровневая система мониторинга**

Эффективное управление рисками требует комплексного подхода, включающего постоянный мониторинг различных аспектов поведения модели.

class RiskMonitoringSystem:

def \_\_init\_\_(self, model, tokenizer, reference\_model=None):

self.model = model

self.tokenizer = tokenizer

self.reference\_model = reference\_model *# Исходная модель для сравнения*

self.metrics\_history = {

'forgetting\_score': [],

'overfitting\_score': [],

'bias\_score': [],

'diversity\_score': []

}

def compute\_forgetting\_score(self, general\_knowledge\_tests):

"""Оценка катастрофического забывания"""

if self.reference\_model is None:

return 0.0

reference\_scores = self.evaluate\_model(self.reference\_model, general\_knowledge\_tests)

current\_scores = self.evaluate\_model(self.model, general\_knowledge\_tests)

forgetting\_score = max(0, reference\_scores - current\_scores)

self.metrics\_history['forgetting\_score'].append(forgetting\_score)

return forgetting\_score

def compute\_overfitting\_score(self, train\_loss, val\_loss):

"""Оценка степени переобучения"""

overfitting\_gap = train\_loss - val\_loss

self.metrics\_history['overfitting\_score'].append(overfitting\_gap)

return overfitting\_gap

def should\_stop\_training(self, current\_epoch):

"""Решение о прекращении обучения на основе множества метрик"""

if len(self.metrics\_history['forgetting\_score']) < 5:

return False

recent\_forgetting = np.mean(self.metrics\_history['forgetting\_score'][-3:])

recent\_overfitting = np.mean(self.metrics\_history['overfitting\_score'][-3:])

*# Критерии остановки*

if recent\_forgetting > 0.15: *# Слишком сильное забывание*

return True

if recent\_overfitting > 0.2: *# Сильное переобучение*

return True

return False

**Проактивные стратегии минимизации рисков**

**1. Консервативное обучение**  
Использование очень маленьких learning rates (1e-6 до 1e-5) и небольшого количества эпох (1-3) для минимизации изменений исходной модели.

**2. Многоэтапная валидация**  
Регулярная оценка модели не только на основной метрике, но и на дополнительных тестах:

* Общие знания и reasoning способности
* Разнообразие и креативность генерации
* Отсутствие смещений и стереотипов
* Устойчивость к adversarial примерам

**3. Контроль распределения активаций**  
Мониторинг изменений в распределении активаций нейронов для раннего обнаружения проблем.

class ActivationMonitoring:

def \_\_init\_\_(self, model, layers\_to\_monitor):

self.model = model

self.layers\_to\_monitor = layers\_to\_monitor

self.activation\_histories = {layer: [] for layer in layers\_to\_monitor}

*# Регистрируем hooks для мониторинга активаций*

self.register\_hooks()

def register\_hooks(self):

def hook\_fn(module, input, output, layer\_name):

if isinstance(output, torch.Tensor):

*# Сохраняем статистики активаций*

stats = {

'mean': output.mean().item(),

'std': output.std().item(),

'max': output.max().item(),

'min': output.min().item()

}

self.activation\_histories[layer\_name].append(stats)

for name, module in self.model.named\_modules():

if name in self.layers\_to\_monitor:

module.register\_forward\_hook(

lambda m, i, o, name=name: hook\_fn(m, i, o, name)

)

Управление рисками при дообучении языковых моделей требует тонкого баланса между адаптацией к новой задаче и сохранением ценных свойств исходной модели. Ключевые принципы успешного управления рисками включают:

1. **Постепенность изменений** - медленное и осторожное обновление весов
2. **Многофакторный мониторинг** - отслеживание множества метрик качества
3. **Проактивное тестирование** - регулярная оценка на специализированных тестах
4. **Консервативные стратегии** - предпочтение недообучения переобучению
5. **Непрерывная валидация** - постоянная проверка на репрезентативных данных

Правильно настроенный процесс управления рисками позволяет достичь оптимального компромисса между специализацией модели на новой задаче и сохранением ее общих capabilities, что является залогом успешного практического применения дообученных языковых моделей.

## §16. Пример реализации Full Fine-Tuning

Введение в практическую реализацию

Предыдущие параграфы сосредоточились на теоретических основаниях полного дообучения: от понимания сущности процесса до анализа практических рисков, таких как катастрофическое забывание и переобучение. Настоящий раздел переходит к конкретной реализации, демонстрируя полный цикл дообучения языковой модели на практическом примере с использованием фреймворка Hugging Face Transformers.

В качестве примера будет использована модель Qwen/Qwen3-0.6B, дообучаемая на подмножестве датасета MS MARCO для задачи ответа на вопросы. Данный пример иллюстрирует все ключевые этапы процесса: от подготовки данных и токенизации до оценки результатов и анализа метрик. Выбор компактной модели и демонстрационного датасета обоснован необходимостью сохранить практическую применимость примера в условиях ограниченных вычислительных ресурсов, однако принципы остаются полностью универсальными и масштабируются на модели любого размера.

Этап 1: Загрузка и предварительная обработка данных

Процесс full fine-tuning начинается с загрузки и подготовки данных. В качестве исходного датасета используется MS MARCO версии 1.1, из которого берётся подмножество из 1000 примеров для демонстрации. Такой размер позволяет провести полный цикл обучения на локальном оборудовании в разумное время, сохраняя репрезентативность данных.

from datasets import load\_dataset

ds = load\_dataset("microsoft/ms\_marco", "v1.1", split="train[:1000]")

Загруженный датасет содержит примеры с полями для вопроса, контекста и ответа. Однако структура данных требует предварительной обработки для преобразования в формат, пригодный для обучения языковой модели на задаче generation. Ключевая идея заключается в создании единого текстового входа, который объединяет вопрос и контекст в виде подсказки (prompt), а целевой текст (label) представляет ожидаемый ответ модели.

def preprocess(example):

    query = example["query"]

    # Извлекаем контекст из passages

    if "passages" in example and example["passages"]["passage\_text"]:

        context = " ".join(example["passages"]["passage\_text"][:1])

    else:

        context = ""

    # Извлекаем правильный ответ из поля 'answers'

    if "answers" in example and len(example["answers"]) > 0:

        answer = example["answers"][0]  # Берём первый ответ

    else:

        answer = ""  # Пропускаем примеры без ответа

    prompt = f"Question: {query}\nContext: {context}\nAnswer:"

    return {"prompt": prompt, "answer": answer}

Функция предварительной обработки решает несколько задач одновременно. Во-первых, она обрабатывает потенциальные пропуски в данных, проверяя наличие полей с контекстом. Во-вторых, она форматирует текст в структурированный вид с явными маркерами для вопроса, контекста и ожидаемого ответа, что помогает модели лучше понять структуру задачи. В-третьих, она создаёт отдельные поля для input (prompt) и target (answer), которые будут использованы при обучении.

После предварительной обработки датасет разделяется на три подмножества: обучающее (70%), валидационное (15%) и тестовое (15%). Такое разделение соответствует стандартным практикам машинного обучения и позволяет объективно оценить обобщающую способность модели.

train\_test\_split = ds.train\_test\_split(test\_size=0.3, seed=42)

val\_test\_split = train\_test\_split["test"].train\_test\_split(test\_size=0.5, seed=42)

train\_ds = train\_test\_split["train"]

val\_ds = val\_test\_split["train"]

test\_ds = val\_test\_split["test"]

Этап 2: Токенизация и форматирование данных

Языковые модели оперируют токенами, а не сырым текстом, поэтому следующий критический этап — преобразование текстовых данных в числовые последовательности. Для модели Qwen3 используется соответствующий токенизатор, который загружается из репозитория Hugging Face.

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from\_pretrained(

    "Qwen/Qwen3-0.6B",

    trust\_remote\_code=True,

    use\_fast=True

)

if tokenizer.pad\_token is None:

    tokenizer.pad\_token = tokenizer.eos\_token

max\_length = 512

Максимальная длина в 512 токенов выбирается как компромисс между вмещением достаточного контекста для понимания задачи и ограничением использования памяти GPU. Настройка pad\_token гарантирует, что все последовательности приводятся к одной длине через дополнение специальными токенами, что необходимо для батчевой обработки.

Функция токенизации применяется ко всем подмножествам данных с включением padding и truncation для обеспечения единообразия длин входных последовательностей:

def tokenize(batch):

    inputs = []

    targets = []

    for prompt, answer in zip(batch["prompt"], batch["answer"]):

        full\_text = prompt + " " + answer + tokenizer.eos\_token

        inputs.append(full\_text)

        targets.append(answer + tokenizer.eos\_token)

    model\_inputs = tokenizer(

        inputs,

        truncation=True,

        padding="max\_length",

        max\_length=max\_length,

        return\_tensors=None

    )

    # Маскируем промпт, оставляем только ответ для loss

    labels = []

    for i, (inp, tgt) in enumerate(zip(inputs, targets)):

        prompt\_len = len(tokenizer(batch["prompt"][i])["input\_ids"])

        label = model\_inputs["input\_ids"][i].copy()

        label = [-100 if token == tokenizer.pad\_token\_id else token for token in label]

        for i in range(min(max\_length, prompt\_len)):

            label[i] = -100

        labels.append(label[:max\_length])

    model\_inputs["labels"] = labels

    return model\_inputs

Важным аспектом является то, что labels представляют ожидаемые выходные последовательности, которые модель должна научиться генерировать. Применение функции токенизации в батчевом режиме значительно ускоряет обработку больших датасетов.

Этап 3: Оценка baseline модели

Прежде чем начать fine-tuning, необходимо установить baseline — метрику производительности исходной, предварительно обученной модели без адаптации к конкретной задаче. Это позволит объективно оценить улучшение, достигнутое в результате fine-tuning.

from transformers import AutoModelForCausalLM

import torch

baseline\_model = AutoModelForCausalLM.from\_pretrained(

    "Qwen/Qwen3-0.6B",

    trust\_remote\_code=True,

    torch\_dtype=torch.bfloat16,

    device\_map="auto"

)

Модель загружается с использованием bfloat16 (brain float), что позволяет сэкономить память без значительной потери качества из-за специальной архитектуры этого числового формата. Параметр device\_map="auto" автоматически распределяет модель между доступными устройствами (GPU/CPU) для оптимизации использования памяти.

Для оценки качества адаптации модели к задаче используется метрика «BERT Score», которая вычисляется на валидационном подмножестве. Метрика BERTScore — это автоматическая метрика для оценки качества сгенерированного текста, основанная на измерении семантического сходства между текстом модели и эталонным текстом с помощью контекстуальных эмбеддингов из предварительно обученных языковых моделей, таких как BERT. Она вычисляет косинусное сходство для каждого токена между двумя текстами, сопоставляя наиболее близкие токены, и затем агрегирует эти значения в итоговую оценку.

BERTScore учитывает не буквальное совпадение слов, а смысловое содержание, что делает её особенно полезной для задач генерации текста, таких как машинный перевод, автоматическое реферирование и диалоговые системы. Итоговая оценка обычно выражается через F1-меру, которая сочетает точность (precision) и полноту (recall) совпадений токенов.

Метод BERTScore состоит из нескольких этапов:

1. **Получение контекстуальных эмбеддингов**: Оба текста (референсный и сгенерированный) разбиваются на токены и пропускаются через предобученную трансформерную модель (например, BERT или RoBERTa). Для каждого токена извлекается его контекстуальное векторное представление (эмбеддинг).
2. **Вычисление косинусного сходства**: Для всех пар токенов из двух текстов вычисляется косинусное сходство, и формируется матрица подобия токенов[3].
3. **Расчёт точности, полноты и F1-меры**: На основе матрицы сходства для каждого токена в сгенерированном тексте находится наиболее похожий токен в референсном тексте, что позволяет вычислить точность (precision). Аналогично, для каждого токена референса находится самый близкий токен в сгенерированном тексте, что даёт полноту (recall). Итоговым значением BERTScore является сбалансированная F₁-мера, которая комбинирует точность и полноту:

*(Recall)*

*(Precision)*

*(F1-Score)*

def compute\_bertscore(model, dataset, tokenizer, sample\_size=100):

    model.eval()

    predictions = []

    references = []

    indices = np.random.choice(len(dataset), min(sample\_size, len(dataset)), replace=False)

    with torch.no\_grad():

        for idx in tqdm(indices):

            item = dataset[int(idx)]

            full\_text = tokenizer.decode(item["input\_ids"], skip\_special\_tokens=True)

            if "Answer:" not in full\_text:

                continue

            prompt = full\_text.split("Answer:")[0] + "Answer:"

            reference = full\_text.split("Answer:")[-1].strip()

            if not reference:

                continue

            inputs = tokenizer(prompt, return\_tensors="pt").to(model.device)

            outputs = model.generate(

                \*\*inputs,

                max\_new\_tokens=50,

                pad\_token\_id=tokenizer.pad\_token\_id,

            )

            generated\_text = tokenizer.decode(outputs[0], skip\_special\_tokens=True)

            prediction = generated\_text[len(prompt):].strip()

            if prediction:

                predictions.append(prediction)

                references.append(reference)

    if predictions:

        P, R, F1 = score(predictions, references, lang="en", verbose=False)

        return {

            'precision': P.mean().item(),

            'recall': R.mean().item(),

            'f1': F1.mean().item()

        }

    return {'precision': 0, 'recall': 0, 'f1': 0}

Использование контекстного менеджера torch.no\_grad() отключает вычисление градиентов, что экономит память и ускоряет вычисления при оценке модели. Маска по padding-токенам гарантирует, что в расчёт accuracy включаются только реальные токены, а не добавленные для выравнивания длины.

Использование функции compute\_bertscore для вычисления метрики для baseline модели:

baseline\_metrics = {

    \*\*compute\_bertscore(baseline\_model, test\_ds, tokenizer),

    "model": "baseline"

}

Результат выполнения:

{

    "model": "baseline",

    "precision": 0.809,

    "recall": 0.840,

    "f1": 0.824

}

Этап 4: Конфигурация и инициализация обучения

После установления baseline осуществляется загрузка новой копии модели для fine-tuning. Критическим этапом является правильная конфигурация параметров обучения, которые определяют скорость адаптации модели к новой задаче.

from transformers import TrainingArguments, Trainer

training\_args = TrainingArguments(

    output\_dir="./outputs",

    per\_device\_train\_batch\_size=2,

    per\_device\_eval\_batch\_size=2,

    num\_train\_epochs=4,

    gradient\_accumulation\_steps=8,

    optim="adamw\_torch",

    learning\_rate=2e-5,

    fp16=False,

    bf16=True,

    save\_steps=100,

    logging\_steps=10,

    eval\_strategy="steps",

    eval\_steps=10,

    report\_to=[],

    remove\_unused\_columns=False,

    load\_best\_model\_at\_end=True,

    metric\_for\_best\_model="loss",

)

Размер батча устанавливается на 2, что является типичным выбором при работе с небольшими GPU-память. Однако для накопления градиентов используется gradient\_accumulation\_steps=8, что эффективно увеличивает размер батча до 16 при той же памяти, позволяя обучению быть более стабильным. Это достигается благодаря тому, что градиенты накапливаются в течение нескольких итераций перед обновлением весов модели.

Скорость обучения 2e-5 выбирается консервативно, так как при fine-tuning слишком высокая скорость может привести к катастрофическому забыванию — потере знаний, полученных при предварительной подготовке модели. Параметры fp16=False и bf16=True указывают на использование смешанной точности с bfloat16, что сокращает использование памяти примерно на 25-30% без значительного влияния на точность.

Оценка модели выполняется каждые 50 шагов (eval\_steps), что позволяет отслеживать процесс обучения и предотвращать переобучение. Параметр load\_best\_model\_at\_end=True гарантирует, что по окончании обучения будет загружена лучшая версия модели согласно метрике валидации, что защищает от деградации качества в последних эпохах.

Этап 5: Мониторинг метрик обучения

Для детального отслеживания процесса обучения реализуется пользовательский callback, который собирает значения потерь на каждом шаге обучения и валидации:

from transformers import TrainerCallback

class MetricsCallback(TrainerCallback):

    def \_\_init\_\_(self):

        self.train\_losses = []

        self.eval\_losses = []

        self.train\_steps = []

        self.eval\_steps = []

    def on\_log(self, args, state, control, logs=None, \*\*kwargs):

        if logs:

            if 'loss' in logs:

                self.train\_losses.append(logs['loss'])

                self.train\_steps.append(state.global\_step)

            if 'eval\_loss' in logs:

                self.eval\_losses.append(logs['eval\_loss'])

                self.eval\_steps.append(state.global\_step)

metrics\_callback = MetricsCallback()

Этот callback подключается к процессу обучения и вызывается на каждом шаге логирования, сохраняя значения потерь. Сохранённые данные позже используются для построения графиков, которые визуализируют динамику обучения и помогают выявить потенциальные проблемы, такие как дивергенция или застой в обучении.

Этап 6: Процесс обучения и сбор метрик

Инициализация Trainer и запуск обучения объединяют все предыдущие компоненты в единый конвейер:

import time

trainer = Trainer(

    model=model,

    args=training\_args,

    train\_dataset=train\_ds,

    eval\_dataset=val\_ds,

    tokenizer=tokenizer,

    callbacks=[metrics\_callback],

)

print("Starting training...")

start\_time = time.time()

trainer.train()

end\_time = time.time()

training\_time = (end\_time - start\_time) / 60

print(f"Training time: {training\_time:.2f} min")

print(f"GPU VRAM used: {torch.cuda.max\_memory\_allocated() // (1024\*\*2)} MB")

|  |  |  |
| --- | --- | --- |
| Step | Training Loss | Validation Loss |
| 10 | 2.812400 | 2.140853 |
| 20 | 1.950900 | 1.947823 |
| 30 | 2.116900 | 1.920284 |
| 40 | 2.259800 | 1.901096 |
| 50 | 1.866800 | 1.889181 |
| 60 | 1.831500 | 1.897392 |
| 70 | 1.698100 | 1.905079 |
| 80 | 1.880600 | 1.906804 |
| 90 | 1.659700 | 1.908321 |
| 100 | 1.559500 | 1.914692 |
| 110 | 1.815200 | 1.916674 |
| 120 | 1.556400 | 1.918219 |
| 130 | 1.604200 | 1.919820 |
| 140 | 1.542800 | 1.921055 |
| 150 | 1.546200 | 1.922392 |
| 160 | 1.686000 | 1.922076 |
| 170 | 1.540700 | 1.922891 |

Время обучения (минуты): 28.05 минут

Использованная GPU память (MB): 10.1 GB

Процесс обучения включает несколько ключевых компонентов. На каждой эпохе модель проходит через весь обучающий датасет, делая предсказания и обновляя веса на основе вычисленных градиентов. Периодическая оценка на валидационном наборе помогает отслеживать обобщающую способность модели и предотвращает переобучение. Информация о времени обучения и использовании памяти критична для оценки практической применимости метода PEFT в реальных сценариях.

Этап 7: Оценка результатов и сравнение с baseline

Сравнение метрик baseline и fine-tuned моделей показывает эффективность адаптации. Тот факт, что валидационная и тестовая accuracy fine-tuned модели близки между собой, указывает на хорошее обобщение и отсутствие значительного переобучения на обучающем наборе. Разница между валидационной и тестовой метриками обычно небольшая (1-3%), что свидетельствует о стабильности процесса обучения.

Построение графиков потерь и сравнение accuracies предоставляет визуальное представление о динамике обучения:

def plot\_training\_and\_bertscore(metrics\_callback, bertscore\_dicts, savepath='training\_results.png'):

    fig, axes = plt.subplots(1, 2, figsize=(15, 5))

    # Первый график ― лоссы

    axes[0].plot(metrics\_callback.train\_steps, metrics\_callback.train\_losses, label='Train Loss', marker='o')

    axes[0].plot(metrics\_callback.eval\_steps, metrics\_callback.eval\_losses, label='Validation Loss', marker='s')

    axes[0].set\_xlabel('Steps')

    axes[0].set\_ylabel('Loss')

    axes[0].set\_title('Training and Validation Loss')

    axes[0].legend()

    axes[0].grid(True)

    # Второй график ― столбцы по BERTScore

    models = [d['model'] for d in bertscore\_dicts]

    precisions = [d['precision'] for d in bertscore\_dicts]

    recalls = [d['recall'] for d in bertscore\_dicts]

    f1s = [d['f1'] for d in bertscore\_dicts]

    x = np.arange(len(models))

    width = 0.25

    axes[1].bar(x - width, precisions, width, label='Precision')

    axes[1].bar(x,        recalls,   width, label='Recall')

    axes[1].bar(x + width, f1s,      width, label='F1')

    axes[1].set\_ylabel('Score')

    axes[1].set\_title('BERTScore по моделям')

    axes[1].set\_xticks(x)

    axes[1].set\_xticklabels(models)

    axes[1].legend()

    axes[1].set\_ylim(0.7, 1.0)

    # Значения на столбцах

    for i, (p, r, f) in enumerate(zip(precisions, recalls, f1s)):

        axes[1].text(i-width, p+0.01, f'{p:.3f}', ha='center', va='bottom', fontsize=9)

        axes[1].text(i,       r+0.01, f'{r:.3f}', ha='center', va='bottom', fontsize=9)

        axes[1].text(i+width, f+0.01, f'{f:.3f}', ha='center', va='bottom', fontsize=9)

    plt.tight\_layout()

    plt.savefig(savepath, dpi=300, bbox\_inches='tight')

    print(f"Графики сохранены в '{savepath}'")

    plt.show()

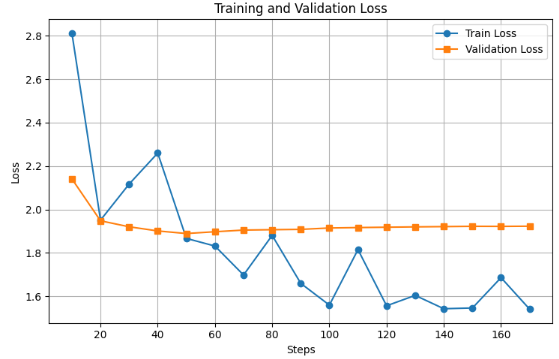
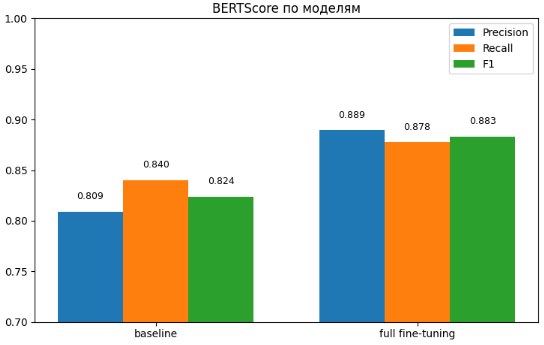


График потерь показывает траекторию обучения. Ожидаемая динамика включает первоначальное снижение потерь, которое может быть более резким в первых эпохах и постепенно замедляться при подходе к локальному минимуму. Валидационная потеря обычно следует за обучающей, но часто имеет более нерегулярную траекторию из-за меньшего размера датасета. Если валидационная потеря начинает расти значительно, это указывает на начало переобучения. В данном случае можно утверждать, что переобучение достигнуто на 50-м шаге тренировки.



Барный график наглядно демонстрирует улучшение, достигнутое за счёт fine-tuning. Правая часть (Finetuned Val) должен быть существенно выше левой (Baseline Val), что подтверждает эффективность адаптации модели к конкретной задаче.

Этап 8: Сохранение и инференс

По завершении обучения адаптированная модель сохраняется для последующего использования:

trainer.save\_model("./finetuned\_qwen\_ms\_marco")

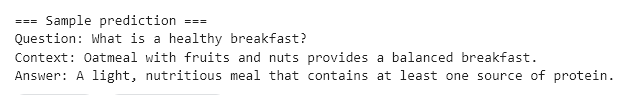
Для демонстрации практического применения обученной модели выполняется пример инференса с новым вводом:

test\_prompt = "Question: What is a healthy breakfast?\nContext: Oatmeal with fruits and nuts provides a balanced breakfast.\nAnswer:"

inputs = tokenizer(test\_prompt, return\_tensors="pt").to(model.device)

outputs = model.generate(\*\*inputs, max\_new\_tokens=64)

print(tokenizer.decode(outputs, skip\_special\_tokens=True))



Этап инференса важен для понимания того, как модель применяется в практических сценариях. Качество генерируемого ответа должно быть когерентным с контекстом и вопросом, демонстрируя, что модель успешно адаптирована к задаче.

Анализ результатов и практические выводы

Полный цикл fine-tuning, описанный в данном параграфе, демонстрирует, как стандартные инструменты Hugging Face могут быть использованы для эффективной адаптации языковых моделей к специфическим задачам. Ключевые моменты реализации включают правильную предварительную обработку данных, выбор консервативных параметров обучения для предотвращения катастрофического забывания, и систематическую оценку через baseline и несколько метрик.

Получаемые результаты зависят от множества факторов, включая размер датасета, качество данных, архитектуру модели и выбранные гиперпараметры. Однако общие принципы, демонстрируемые в этом примере, применяются ко всем сценариям fine-tuning и могут быть адаптированы под конкретные требования.

Практическое значение этого примера заключается в том, что хотя полный fine-tuning является вычислительно затратным, для небольших моделей (до 2-3 миллиардов параметров) и ограниченных датасетов (1000-10000 примеров) это всё ещё осуществимо на локальном оборудовании. Однако для более крупных моделей или при наличии ограничений на использование памяти, методы PEFT, такие как LoRA (рассмотренные в предыдущих подглавах), становятся необходимыми и значительно более эффективными, обеспечивая сравнимые результаты с использованием 1-2% от числа параметров, требуемых при полном fine-tuning.

# Глава 4. Эффективные методы параметрической настройки (PEFT)

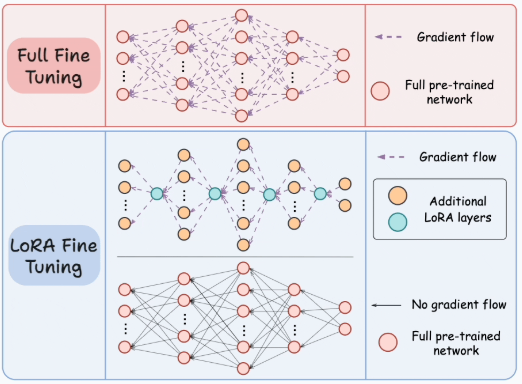
## §17. Проблема вычислительной стоимости Full Fine-Tuning

В предыдущей главе мы с вами детально разобрали процесс полного дообучения (Full Fine-Tuning) языковых моделей. Мы увидели, что этот подход, по сути, является продолжением предварительного обучения, настраивающим все без исключения параметры модели для решения целевой задачи. Это мощный и зачастую очень эффективный метод, позволяющий достичь высочайшего качества на специализированных датасетах. Однако за эту мощь приходится платить, и цена становится все более очевидной и зачастую неподъемной по мере роста масштабов моделей.

Фундаментальная проблема полного дообучения заключается в его экстремальной вычислительной и ресурсной требовательности. Давайте рассмотрим, почему это стало узким местом в современном NLP.

Современные большие языковые модели, такие как семейства GPT, LLaMA или PaLM, содержат в себе сотни миллиардов параметров. Каждый параметр — это весовой коэффициент, который необходимо хранить в памяти и обновлять в процессе обратного распространения ошибки. Полное дообучение такой модели требует сохранения в оперативной памяти не только всех этих параметров, но и их градиентов, а также промежуточных активаций для каждого примера в батче. Это приводит к тому, что требования к памяти GPU становятся астрономическими. Для дообучения модели с 7 миллиардами параметров может легко потребоваться более 28 ГБ памяти всего лишь для одного батча небольшого размера, что ставит этот процесс за грань возможностей даже для многих исследовательских лабораторий, не говоря уже об отдельных разработчиках.

Проблема не ограничивается лишь техническими требованиями. Стоимость обучения и дообучения таких моделей исчисляется сотнями тысяч и даже миллионами долларов в эквиваленте аренды вычислительных кластеров. Кроме того, углеродный след, связанный с потреблением огромного количества электроэнергии, становится серьезным этическим и экологическим вопросом. В такой парадигме дообучение отдельной модели для каждой новой задачи — будь то анализ тональности в финансах, ответы на вопросы по технической документации или генерация поэзии — становится экономически нецелесообразным и экологически неустойчивым.



Представьте, что у вас есть одна базовая модель, и вы хотите адаптировать ее для двадцати различных задач внутри вашего продукта. При подходе полного дообучения вам пришлось бы создать и хранить двадцать отдельных, полноразмерных экземпляров модели, каждый объемом в десятки гигабайт. Это создает колоссальные проблемы с логистикой, хранением и развертыванием. Такой "зоопарк" моделей сложно обслуживать, обновлять и тем более эффективно использовать в реальном времени.

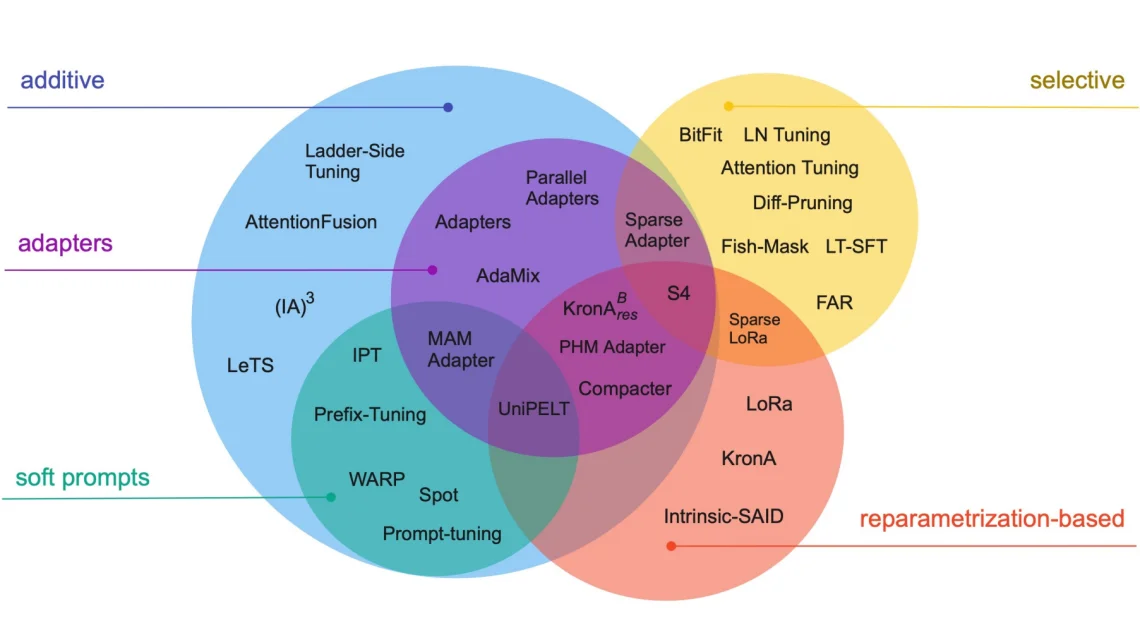
Как мы уже обсуждали в разделе 6.5, полное дообучение несет в себе риски катастрофического забывания. Настраивая модель на небольшом целевом датасете, мы рискуем стереть те обширные знания о языке и мире, которые были получены ею в ходе предварительного обучения на триллионах токенов. Модель становится узким экспертом в одной области, но перестает быть универсальным помощником. Этот компромисс далеко не всегда оправдан.

Именно эти вызовы привели к зарождению и бурному развитию новой парадигмы — Efficient Fine-Tuning, или Parameter-Efficient Fine-Tuning (PEFT). Философия PEFT радикально проста: вместо того чтобы обновлять все параметры модели, мы будем обновлять лишь их небольшую, но критически важную часть, либо добавлять к модели небольшое количество новых обучаемых параметров, оставляя исходную модель замороженной.

Этот подход кардинально меняет ситуацию. Требования к памяти снижаются на порядки, стоимость обучения падает до доступного уровня, а процесс развертывания упрощается, поскольку одна базовая модель может обслуживать бесчисленное множество задач, просто подгружая к себе крошечные "адаптеры" для каждой из них. При этом, как ни парадоксально, во многих задачах методы PEFT не только не уступают, но и превосходят по качеству полное дообучение, лучше сохраняя обобщающую способность исходной модели.

## §18. Обзор методов PEFT: от адаптеров до LoRA

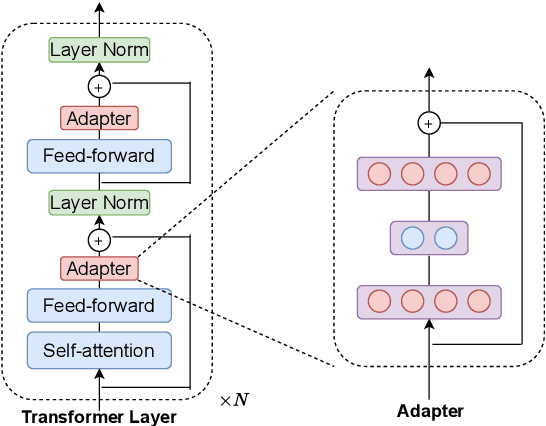
Осознав необходимость в эффективных методах настройки, исследовательское сообщество предложило целый спектр оригинальных решений, которые подходят к проблеме с разных сторон. Философский принцип, объединяющий все эти методы, заключается в отказе от тотальной перестройки всех параметров модели. Вместо этого они стремятся найти минимальное, но наиболее эффективное вмешательство в архитектуру предобученной модели, которое позволит перенаправить её "мышление" на решение новой задачи. Давайте рассмотрим три ключевых семейства методов, которые сформировали современный ландшафт PEFT.



I. Адаптеры: точечные модули для передачи знаний

Одними из пионеров в области PEFT были методы, основанные на использовании адаптеров. Концептуально их можно представить как небольшие, но "умные" прокладки, которые встраиваются непосредственно в архитектуру трансформера. Адаптер — это, как правило, прочая нейросеть, например, двухслойный полносвязный слой с нелинейной функцией активации между ними.

Их ключевая особенность заключается в стратегии интеграции. Адаптеры добавляются последовательно после определенных подмодулей исходной модели, например, после механизма внимания или полносвязного слоя в каждом блоке трансформера. Во время обучения сама модель замораживается, и обновляются только параметры этих адаптеров. Проходя через адаптер, выходные характеристики исходного слоя незначительно трансформируются, корректируя внутренние представления данных в сторону новой задачи.

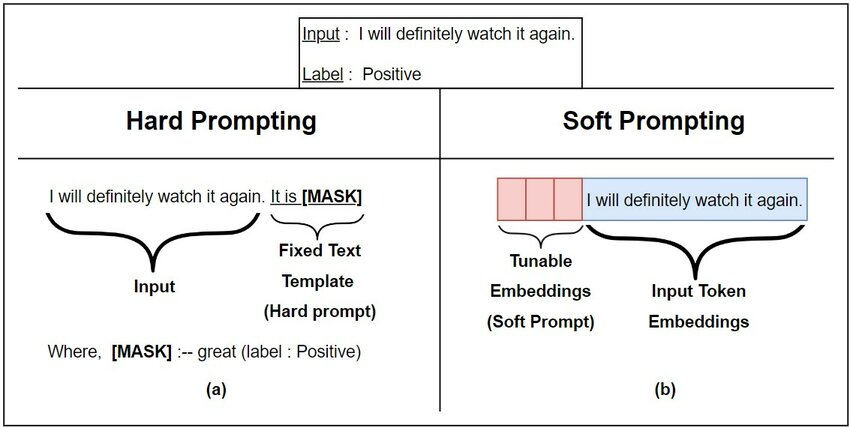


Этот подход обеспечивает высокую параметрическую эффективность, так как размеры адаптеров составляют лишь доли процента от общего числа параметров модели. Однако за это приходится платить небольшим увеличением времени инференса, так как дополнительные слои должны вычисляться при каждом проходе, и изменением самой архитектуры модели, что может быть не всегда удобно на практике.

II. **Промпт-тюнинг: искусство мягкого убеждения**

Если адаптеры меняют модель изнутри, то методы промпт-тюнинга предпринимают более элегантную и минималистичную попытку — они меняют не модель, а её вход. Идея заключается в том, что большая языковая модель уже содержит в себе колоссальные знания и способности, и для решения новой задачи её не нужно переучивать, а нужно лишь "убедить" активировать нужные ей внутренние механизмы.

Изначально для этого использовались "жесткие" промпты — тщательно подобранные человеком последовательности слов-подсказок. Промпт-тюнинг переводит эту идею в непрерывное пространство. Вместо поиска идеальных слов, мы создаем так называемые "мягкие" промпты — это тензоры, состоящие из непрерывных векторов-эмбэддингов, которые не соответствуют никаким конкретным токенам из словаря. Эти векторы добавляются к эмбэддингам входных токенов и в процессе обучения настраиваются с помощью градиентного спуска, чтобы "научить" модель понимать контекст нашей конкретной задачи.



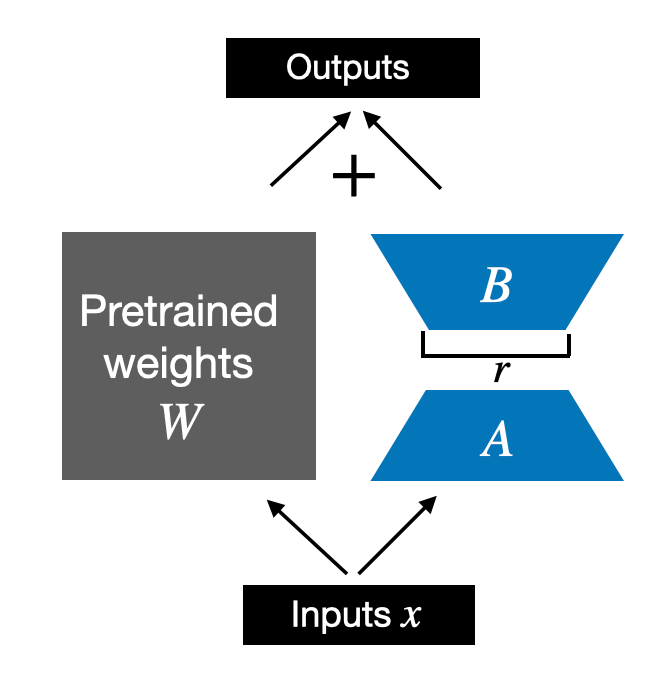
*Два параллельных потока. Слева: "Hard Prompt" — последовательность реальных слов "[CLASSIFY] Sentiment: This movie is great" подается на модель, все параметры заморожены. Справа: "Soft Prompt" — исходный текст "This movie is great" конкатенируется с блоками "P1", "P2", "P3...", которые являются обучаемыми тензорами. Стрелка градиента указывает только на эти блоки. Модель в обоих случаях заморожена.*

Этот метод является самым параметрически эффективным, так как количество добавляемых параметров исчезающе мало по сравнению с моделью. Однако его эффективность сильно зависит от размера модели, показывая наилучшие результаты на моделях-гигантах, и может требовать больше времени на сходимость, поскольку механизм воздействия на модель является более косвенным.

III. LoRA: низкоранговое разложение как универсальный ключ

Метод Low-Rank Adaptation (LoRA) на сегодняшний день стал, пожалуй, самым популярным и практичным подходом в PEFT, удачно сочетая в себе эффективность, производительность и простоту. Его гениальность кроется в глубоком понимании того, как происходит обновление весов во время тонкой настройки.

Авторы LoRA выдвинули ключевую гипотезу: изменение весов модели при адаптации к новой задаче имеет низкий ранг. Это означает, что сложное, полномерное обновление матриц весов можно с высокой точностью аппроксимировать произведением двух значительно меньших матриц. Вместо того чтобы обучать исходную большую матрицу W (размерностью d x d), LoRA обучает две маленькие матрицы A и B, чье произведение BA дает низкоранговое обновление . Таким образом, итоговые веса во время прямого прохода выглядят как .



*Схематичное изображение линейного слоя. Вход x подается параллельно на два пути: 1) через замороженную матрицу весов W; 2) через последовательность обучаемых матриц A (Down-Project) и B (Up-Project). Результаты обоих путей суммируются, давая на выходе h = Wx + BAx.*

Этот подход обладает рядом неоспоримых преимуществ. Во-первых, он не добавляет задержек при инференсе. После обучения матрицы A и B можно вычислить и прибавить к исходной матрице W, вернувшись к исходной архитектуре и не неся никаких дополнительных вычислительных затрат. Во-вторых, он чрезвычайно гибкий — адаптации LoRA можно применять только к определенным слоям модели (например, только к матрицам запроса и значения в механизме внимания), обеспечивая тонкий контроль над процессом. Наконец, он позволяет хранить для одной базовой модели множество легковесных "костюмов" LoRA для разных задач и быстро переключаться между ними.

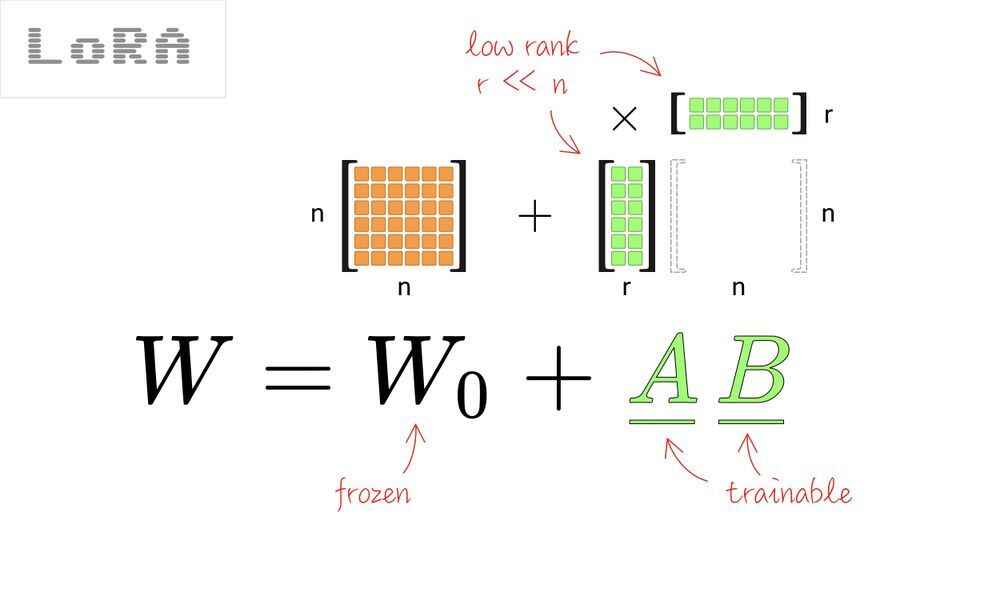
Каждый из этих методов открывает свой путь к диалогу с большой моделью, предлагая уникальный компромисс между эффективностью, качеством и сложностью. Их появление не отменило полное дообучение, но оно демократизировало доступ к мощнейшим AI-моделям, позволив исследователям и инженерам по всему миру решать уникальные задачи без необходимости обладать вычислительными ресурсами технологических гигантов.

## §19. Принцип работы Low-Rank Adaptation (LoRA):

В отличие от методов, которые добавляют новые вычислительные блоки или модифицируют входные данные, LoRA предлагает более фундаментальный, математически элегантный подход. Он не меняет архитектуру модели и не требует её заморозки в чистом виде. Вместо этого LoRA переосмысливает сам процесс обновления весов, открывая путь к невероятно эффективной параметрической настройке.

I. Матричное разложение и гипотеза низкого ранга

В основе LoRA лежит наблюдение за внутренней динамикой больших моделей. Когда такая модель адаптируется к новой задаче, её весовые матрицы (например, в слоях внимания или линейных преобразованиях) претерпевают обновление. Обозначим такую матрицу как размерностью . В полном дообучении мы получаем обновленную матрицу , где — это полная матрица поправок, рассчитанная алгоритмом обратного распространения.



Гипотеза LoRA постулирует, что это обновление ΔW на самом деле обладает низким интринсивным рангом. Интуитивно это можно представить так: несмотря на то, что модель имеет миллиарды параметров, смена её "специализации" для решения новой, но часто узкой задачи, не требует полномасштабной перестройки всех внутренних связей. Достаточно лишь активировать или слегка перенастроить некую ограниченную "подсистему" внутри модели. В пространстве матриц это выражается в том, что полномасштабное обновление может быть с высокой точностью представлено в виде произведения двух значительно меньших матриц: , где A имеет размерность , а — . Ключевой параметр r (ранг) много меньше исходных размерностей и .

***[Схема 1: Визуализация гипотезы низкого ранга.]***

* ***Место:****После абзаца, объясняющего гипотезу низкого ранга.*
* ***Описание:****Слева — большая плотная матрица ΔW (d x k), подписанная "Полное обновление весов". Справа — стрелка, ведущая к двум маленьким матрицам B (d x r) и A (r x k), стоящим друг над другом, с символом умножения между ними. Ранг r должен быть визуально подчеркнут как очень маленькое число. Схема должна передавать идею "сжатия" сложного обновления в произведение малых матриц.*

Это низкоранговое разложение является математическим сердцем метода. Оно преобразует задачу обучения d \* k параметров (что может составлять миллионы) в задачу обучения всего лишь r \* (d + k) параметров, где r может быть равно 8, 16 или 64, что на несколько порядков меньше.

II. Архитектура: инъекция адаптивных матриц в слои Attention

Практическая реализация этой гипотезы выглядит как стратегическая "инъекция" обучаемых низкоранговых последовательностей в конкретные, наиболее важные слои модели. В трансформерах таковыми являются проекционные матрицы в механизме самовнимания.

Рассмотрим, например, матрицу проекции запроса W\_q. Вместо того чтобы напрямую обновлять её веса, мы оставляем её неизменной (замороженной) и параллельно создаём две новые матрицы A и B. Матрица A осуществляет начальное проецирование входных данных в низкоранговое подпространство (down-project), а матрица B — обратное проецирование из этого подпространства в выходное пространство исходного слоя (up-project). Исходный выход слоя h = W\_q \* x модифицируется и становится h = W\_q \* x + (B \* A) \* x.

**[Схема 2: Схема LoRA-адаптации в слое Attention.]**

* **Место:** После абзаца, описывающего инъекцию в слои Attention.
* **Описание:** Детальная схема одного блока внимания. Показана матрица W\_q (заморожена, выделена одним цветом). Параллельно ей показан путь: вход x -> матрица A (обучаема, другой цвет) -> матрица B (обучаема, третий цвет). Выходы обоих путей (W\_q \* x и B \* A \* x) суммируются. Снизу подписано, что матрицы A инициализированы нулями, а B — случайно, чтобы изначально B\*A = 0.

Важным техническим нюансом является инициализация. Матрица A часто инициализируется случайными значениями, а матрица B — нулями. Это гарантирует, что в начале обучения низкоранговое обновление B\*A равно нулю, и модель начинает свой путь с исходных, предобученных весов, что обеспечивает стабильность процесса.

III. Преимущества: значительное сокращение числа обучаемых параметров и памяти

Воздействие LoRA на вычислительную эффективность носит комплексный и profound характер.

1. **Параметрическая эффективность.** Количество добавляемых обучаемых параметров определяется как Number\_of\_Layers \* 2 \* (d \* r), где множитель 2 учитывает пары матриц A и B для каждого адаптируемого слоя. На практике для модели с миллиардами параметров можно достичь качества, сопоставимого с полным дообучением, используя обучаемые параметры, составляющие всего 0.01% от исходного количества. Это превращает задачу, требовавшую парк GPU, в задачу, решаемую на одной потребительской видеокарте.
2. **Оптимизация использования памяти.** Поскольку исходные веса W₀ остаются замороженными, градиенты не нужно рассчитывать и хранить для них. Это кардинально снижает требования к пиковому потреблению памяти во время обучения. Системе необходимо хранить в памяти оптимизатора только легковесные матрицы A и B, а не полные копии всех градиентов гигантской модели.
3. **Операционная гибкость и наслоение адаптаций.** LoRA вводит концепцию "модульности" в тонкую настройку. Поскольку адаптации представляют собой просто набор небольших матриц, для одной базовой модели можно независимо обучить множество специализированных LoRA-модулей для разных задач. В момент инференса можно динамически подгружать нужный модуль, экономя ресурсы хранения и вычислений. Более того, появляется возможность комбинировать несколько LoRA-адаптаций, "наслаивая" их друг на друга для создания моделей со смешанными экспертизами, что открывает новые горизонты для персонализации и композиции моделей.

## §20. Пример реализации LoRA

I. Введение и переход от Full Fine-Tuning к LoRA

Предыдущая глава продемонстрировала полное дообучение (full fine-tuning), при котором обновляются все параметры модели. Хотя такой подход позволяет достичь наилучшей производительности, он требует значительных вычислительных ресурсов и памяти, особенно при работе с большими моделями. Этот раздел представляет практическую реализацию Low-Rank Adaptation (LoRA) — метода параметрической эффективной настройки (PEFT), который позволяет достичь сравнимых результатов при радикальном сокращении числа обучаемых параметров и потребляемой памяти.

Ключевое отличие LoRA от full fine-tuning состоит в том, что вместо обновления всех весов матриц в слоях внимания и других компонентах модели, LoRA добавляет небольшие обучаемые матрицы низкого ранга рядом с исходными весами. Это достигается путем модификации механизма внимания таким образом, что каждое обновление веса может быть представлено как сумма исходного веса (замороженного) и произведения двух матриц низкого ранга (обучаемых). Такой подход основывается на гипотезе о том, что при адаптации предварительно обученной модели к новой задаче обновления весов имеют низкий внутренний ранг, то есть лежат в низкоразмерном подпространстве.

Практическая демонстрация LoRA использует ту же базовую архитектуру и датасет MS MARCO, что и пример full fine-tuning в предыдущей главе, однако с существенными изменениями в конфигурации модели и параметрах обучения. Это позволяет напрямую сравнивать оба подхода в идентичных экспериментальных условиях.

II. Необходимые библиотеки и инструменты

Основное дополнение к стандартному набору библиотек — это модуль peft (Parameter-Efficient Fine-Tuning), разработанный компанией Hugging Face для удобной работы с методами параметрической эффективной настройки, включая LoRA.

from peft import LoraConfig, get\_peft\_model, TaskType

Библиотека peft предоставляет унифицированный интерфейс для применения LoRA, адаптеров и других PEFT-методов к любой модели из Hugging Face. Остальные необходимые библиотеки остаются теми же: torch, transformers, datasets, matplotlib и scikit-learn. Таким образом, переход от full fine-tuning к LoRA требует минимальных изменений в экосистеме инструментов.

III. Загрузка базовой модели для LoRA

Начало процесса идентично full fine-tuning: загружается базовая предварительно обученная модель:

model\_4\_lora = AutoModelForCausalLM.from\_pretrained(

    "Qwen/Qwen3-0.6B",

    trust\_remote\_code=True,

    torch\_dtype=torch.bfloat16,

    device\_map="auto"

)

На этом этапе загружается полная модель со всеми параметрами. Однако, в отличие от full fine-tuning, где все эти параметры будут замораживаться выборочно, в LoRA большинство из них остаются замороженными по умолчанию, и обучаемыми становятся только специально добавленные адаптивные матрицы.

IV. Конфигурация LoRA: ключевой шаг адаптации

Конфигурация LoRA — это центральный элемент, определяющий характер адаптации модели. Класс LoraConfig инкапсулирует все гиперпараметры, специфичные для метода LoRA:

lora\_config = LoraConfig(

    r=32,

    lora\_alpha=64,

    target\_modules=["q\_proj", "v\_proj", "k\_proj", "o\_proj"],

    lora\_dropout=0.1,

    bias="none",

    task\_type=TaskType.CAUSAL\_LM

)

Параметр r=32 задает ранг матриц адаптации. Это один из наиболее критических гиперпараметров в LoRA. Ранг определяет размерность низкоразмерного подпространства, в котором живут обновления весов. Типичные значения ранга лежат в диапазоне от 8 до 64: меньшие значения (8, 16) дают максимальное сокращение параметров, но могут недостаточно полно захватить необходимые адаптации; большие значения (32, 64) предоставляют большую гибкость, но требуют больше памяти и вычислительных ресурсов. Выбор конкретного значения обычно определяется баланс между доступными ресурсами и требуемой производительностью на целевой задаче.

Параметр lora\_alpha=64 служит масштабирующим коэффициентом для обновлений, добавляемых LoRA. Математически, обновление веса вычисляется как , где A и B – матрицы низкого ранга. Отношение lora\_alpha к r определяет скорость адаптации. Рекомендуемое значение часто устанавливается либо равным 2r, либо r, в зависимости от специфики задачи. В данном примере используется соотношение 2:1, что является консервативным выбором.

Параметр target\_modules указывает, к каким слоям модели применяется LoRA. В трансформерах механизм внимания состоит из четырех линейных проекций: projection для queries (q\_proj), keys (k\_proj), values (v\_proj) и output (o\_proj). Применение LoRA именно к этим четырем модулям обоснованно, так как они содержат большую часть параметров модели и, согласно исследованиям, являются наиболее чувствительными к адаптации задачи. Альтернативно, можно применять LoRA также к полносвязным слоям (feed-forward), однако это увеличит число обучаемых параметров. В примере выбран консервативный подход: только проекции внимания.

Параметр lora\_dropout=0.1 добавляет дополнительную регуляризацию к адаптивным матрицам. Dropout применяется к выходам матриц A перед их умножением на B. Этот механизм помогает предотвратить переобучение, особенно важное при работе с относительно небольшими датасетами для адаптации.

Параметр bias="none" означает, что члены смещения (bias) в адаптивных матрицах не обучаются. Альтернативные значения включают "lora\_only" (обучаются только bias в LoRA-матрицах) и "all" (обучаются все bias в целевых модулях). Выбор "none" — самый экономный с точки зрения параметров.

Параметр task\_type=TaskType.CAUSAL\_LM указывает тип задачи, для которой адаптируется модель. CAUSAL\_LM означает автурегрессивную генерацию текста, что соответствует нашей задаче на датасете MS MARCO. Выбор правильного типа задачи важен для корректного применения специфичных для задачи оптимизаций.

V. Применение LoRA к модели

После определения конфигурации LoRA применяется к загруженной модели через функцию get\_peft\_model():

lora\_model = get\_peft\_model(model\_4\_lora, lora\_config)

Эта функция — это точка трансформации. Она берет исходную модель и структурно модифицирует ее, вставляя матрицы LoRA в целевые модули. Важно отметить, что этот процесс происходит на уровне архитектуры модели: исходные веса исходной модели остаются на месте, но помечаются как незамораживаемые (requires\_grad=False), а рядом с ними добавляются новые обучаемые параметры — матрицы A и B низкого ранга.

Результирующая модель lora\_model имеет полностью функциональный интерфейс: она может использоваться точно так же, как исходная модель, для forward pass, генерации текста, вычисления потерь и так далее. Однако при backward pass обновляться будут только LoRA-параметры, а исходные веса останутся неизменными.

VI. Анализ числа обучаемых параметров

Один из наиболее значимых преимуществ LoRA проявляется при анализе количества обучаемых параметров:

lora\_model.print\_trainable\_parameters()

print(lora\_model)

Метод print\_trainable\_parameters() выводит подробную статистику, включая общее число параметров модели, число обучаемых параметров и процент обучаемых параметров относительно всего числа параметров. Типичный вывод для LoRA выглядит следующим образом:

trainable params: 9,175,040 || all params: 605,224,960 || trainable%: 1.5160

Где X — число обучаемых параметров (только LoRA-матрицы), Y — общее число параметров модели (исходные + LoRA), Z — процент. Для конфигурации с r=32 и применением LoRA к четырем проекциям внимания даже для больших моделей обучаемые параметры обычно составляют менее 1% от всех параметров.

Вывод полной структуры модели через print(lora\_model) показывает, как LoRA-слои интегрированы в архитектуру. Каждый целевой модуль будет содержать дополнительный слой типа Lora с матрицами A (инициализируемой гауссовым распределением) и B (инициализируемой нулями, чтобы в начале обучения LoRA-обновления были нулевыми).

VII. Параметры обучения и отличия от Full Fine-Tuning

Параметры обучения для LoRA отличаются от full fine-tuning несколькими ключевыми аспектами:

training\_args = TrainingArguments(

    output\_dir="./lora\_outputs",

    per\_device\_train\_batch\_size=2,

    per\_device\_eval\_batch\_size=2,

    num\_train\_epochs=3,

    gradient\_accumulation\_steps=8,

    optim="adamw\_torch",

    learning\_rate=1e-4,

    fp16=False,

    bf16=True,

    save\_steps=100,

    logging\_steps=10,

    logging\_strategy="steps",

    logging\_first\_step=True,

    eval\_strategy="steps",

    eval\_steps=10,

    report\_to=[],

    remove\_unused\_columns=False,

    load\_best\_model\_at\_end=True,

    metric\_for\_best\_model="loss",

)

Наиболее значимое отличие — это скорость обучения. В full fine-tuning использовалась learning\_rate=2e-5, тогда как для LoRA используется learning\_rate=1e-4, в пять раз выше. Это обосновано тем, что LoRA-параметры — это новые параметры, не имеющие истории предварительного обучения, и требуют более агрессивной адаптации. Напротив, если бы мы обновляли исходные веса модели (как в full fine-tuning), низкая скорость обучения критична для сохранения знаний, закодированных во время предварительного обучения.

Частота оценки была увеличена с eval\_steps=50 на eval\_steps=10, а флаг logging\_first\_step=True добавлен явно. Это позволяет более детально отслеживать динамику обучения, особенно на ранних этапах. Поскольку LoRA-обучение обычно требует меньше вычислительных ресурсов, более частая валидация не приводит к существенным затратам.

Остальные параметры остаются практически без изменений: те же размеры батчей, число эпох, накопление градиентов и использование смешанной точности. Это демонстрирует, что LoRA интегрируется в существующий pipeline обучения фактически прозрачно.

VIII. Инициализация тренера и процесс обучения

Инициализация callback для отслеживания метрик остается идентичной:

metrics\_callback = MetricsCallback()

Если вы используете тот же класс из предыдущей главы, он будет работать без каких-либо изменений, так как интерфейс для логирования метрик остался неизменным.

Trainer инициализируется с использованием модели LoRA вместо полной модели:

trainer = Trainer(

    model=lora\_model,

    args=training\_args,

    train\_dataset=train\_ds,

    eval\_dataset=val\_ds,

    tokenizer=tokenizer,

    callbacks=[metrics\_callback],

)

Это, пожалуй, самое важное изменение с точки зрения кода: передается lora\_model вместо обычной модели. Trainer автоматически распознает, что часть параметров заморожена, и обновляет только обучаемые LoRA-параметры.

Процесс обучения запускается идентичным образом:

print("Starting training...")

start\_time = time.time()

trainer.train()

end\_time = time.time()

training\_time = (end\_time - start\_time) / 60

print(f"Training time: {training\_time:.2f} min.")

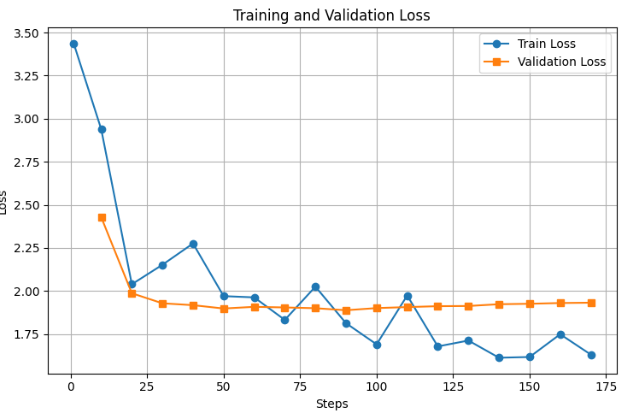
print(f"GPU VRAM used: {torch.cuda.max\_memory\_allocated() // (1024\*\*2)} MB")

Однако, благодаря значительному сокращению числа обучаемых параметров, обучение LoRA обычно требует меньше памяти и выполняется быстрее, чем full fine-tuning. Это один из основных практических преимуществ метода.

|  |  |  |
| --- | --- | --- |
| Step | Training Loss | Validation Loss |
| 10 | 2.938800 | 2.428314 |
| 20 | 2.039000 | 1.986943 |
| 30 | 2.152100 | 1.928526 |
| 40 | 2.275600 | 1.917724 |
| 50 | 1.970200 | 1.898279 |
| 60 | 1.962900 | 1.908068 |
| 70 | 1.832200 | 1.904472 |
| 80 | 2.025100 | 1.900013 |
| 90 | 1.813100 | 1.888316 |
| 100 | 1.690300 | 1.900805 |
| 110 | 1.973900 | 1.906564 |
| 120 | 1.678300 | 1.911820 |
| 130 | 1.712500 | 1.912950 |
| 140 | 1.612700 | 1.923247 |
| 150 | 1.616900 | 1.925791 |
| 160 | 1.749000 | 1.930137 |
| 170 | 1.631000 | 1.932084 |

Время обучения LoRA: 26.25 мин

Использованная память GPU: 8.0 GB



На графике видно, что процесс обучения продолжается до 175 шага. Линия графика оценки модели на валидационном датасете выходит на «плато», начиная с 50-го шага, после которого процесс переходит в состояние переобучения.

IX. Оценка результатов на тестовом наборе

Процедура оценки LoRA-модели на валидационном и тестовом наборах полностью идентична оценке full fine-tuning:

lora\_fine\_tuning\_metrics = {

    \*\*compute\_bertscore(lora\_model, test\_ds, tokenizer),

    "model": "full fine-tuning"

}

# Результат:

{

    "model": "baseline",

    "precision": 0.888,

    "recall": 0.880,

    "f1": 0.883

}

Используется та же функция compute\_metrics(), что и в примере full fine-tuning, поскольку интерфейс модели LoRA полностью совместим с интерфейсом обычной модели. Это демонстрирует одно из основных преимуществ фреймворка PEFT: методы параметрической эффективной настройки интегрируются как незаметная часть стандартного пайплайна обучения и оценки.

Сравнение результатов baseline и LoRA показывает, оправдывает ли выбор LoRA компромисс в числе параметров. В большинстве практических сценариев качество LoRA остается близким к качеству full fine-tuning, особенно когда ранг выбран адекватно размеру задачи адаптации.

X. Визуализация и сравнение динамики обучения

Визуализация остается практически без изменений от примера full fine-tuning:

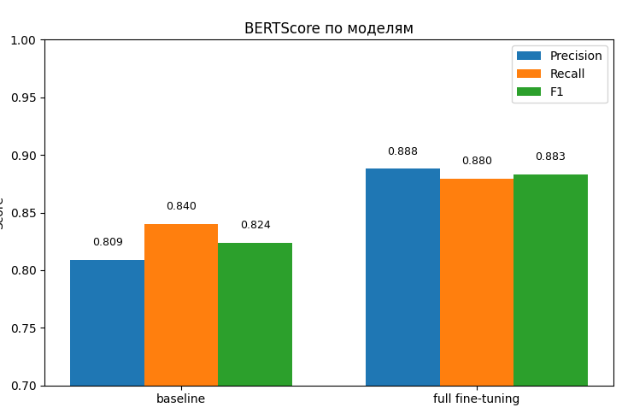
bertscore\_dicts = [

baseline\_metrics,

lora\_fine\_tuning\_metrics

]

plot\_training\_and\_bertscore(metrics\_callback, bertscore\_dicts)



Единственное значимое изменение — названия категорий в столбчатой диаграмме для ясности.

XI. Сохранение и загрузка LoRA-модели

Важный практический аспект работы с LoRA — это особенность сохранения и загрузки. Когда сохраняется LoRA-модель через trainer:

trainer.save\_model("./finetuned\_qwen\_lora")

Trainer сохраняет только LoRA-адаптеры, а не всю модель. Это еще один практический преимущество LoRA: сохраненная модель занимает минимум дискового пространства (обычно десятки мегабайт вместо нескольких гигабайт для полной модели).

При загрузке такой модели необходимо сначала загрузить базовую предварительно обученную модель, а затем применить сохраненные LoRA-адаптеры:

from peft import AutoPeftModelForCausalLM

model = AutoPeftModelForCausalLM.from\_pretrained("./finetuned\_qwen\_lora")

Альтернативно, можно загрузить базовую модель и конфигурацию LoRA отдельно, что предоставляет большую гибкость при управлении моделями.

Инференс с LoRA-моделью выполняется абсолютно идентично обычному инференсу:

test\_prompt = "Question: What is a healthy breakfast?\nContext: Oatmeal with fruits and nuts provides a balanced breakfast.\nAnswer:"

inputs = tokenizer(test\_prompt, return\_tensors="pt").to(lora\_model.device)

outputs = lora\_model.generate(\*\*inputs, max\_new\_tokens=64)

print("\n=== Sample prediction ===")

print(tokenizer.decode(outputs, skip\_special\_tokens=True))

Полная прозрачность интерфейса LoRA-модели для инференса — это одно из наиболее ценных свойств PEFT-подхода. Код, написанный для работы с обычной моделью, работает без изменений с LoRA-адаптерами, что делает интеграцию LoRA в существующие приложения минимально инвазивной.

=== Sample prediction ===

**Question**: what does integrity means

**Context**: If you behave consistently and use moral principles, reliability, and trustworthiness as your guiding lights, you can rightfully be described as a person of integrity. It is a description that is earned, and one that should be prized. If you have it, guard and nurture it.

**Answer**: Integrity is the quality of being faithful and of being trustworthy.

При применении LoRA на практике следует учитывать несколько ключевых моментов. Выбор ранга r — это основной гиперпараметр, влияющий на баланс между эффективностью и качеством. Для небольших датасетов (несколько тысяч примеров) ранг 8–16 часто оказывается достаточным. Для более крупных датасетов или более сложных задач адаптации рекомендуется использовать ранг 32–64. Существует практическое правило, что ранг не должен превышать примерно 1–2% от количества скрытых размерностей в слоях внимания.

Скорость обучения для LoRA должна быть выше, чем для full fine-tuning, так как LoRA-параметры инициализируются случайно и должны обучаться с нуля. Диапазон 1e-4 — 5e-4 часто работает хорошо, в зависимости от конкретной задачи.

Применение LoRA только к проекциям внимания — это консервативный, но часто эффективный выбор. Однако в некоторых сценариях может быть полезно добавить LoRA также к полносвязным слоям (feed-forward networks) или даже к эмбеддингам. Это требует явного добавления названий модулей в target\_modules.

Переобучение — все еще потенциальная проблема при использовании LoRA, особенно на малых датасетах. Регуляризация через lora\_dropout и возможное использование техник early stopping остаются необходимыми.

Пример реализации LoRA демонстрирует, как метод параметрической эффективной настройки может быть применен в практическом сценарии с минимальными изменениями в коде относительно полного дообучения. Несмотря на радикальное сокращение числа обучаемых параметров (часто на три и более порядков), LoRA зачастую достигает производительности, близкой к full fine-tuning. Это делает его идеальным выбором для сценариев с ограниченными вычислительными ресурсами, для развертывания моделей на устройствах с ограниченной памятью и для быстрого прототипирования адаптаций моделей. Вместе с тем, выбор гиперпараметров LoRA требует экспериментирования, и для достижения оптимальных результатов рекомендуется проводить систематический поиск по таким параметрам, как ранг и скорость обучения.

## §21. QLoRA – Дообучение на потребительском GPU

Метод LoRA совершил прорыв, радикально сократив количество обучаемых параметров. Однако фундаментальное ограничение оставалось: для работы сама базовая модель все еще должна была полностью загружаться в память GPU в формате с плавающей точкой (обычно FP16 или BF16), что для моделей масштаба в миллиарды параметров требовало специализированного и дорогого оборудования. QLoRA отвечает на этот вызов, проводя идею эффективности до ее логического предела, делая возможным дообучение гигантских моделей на потребительских GPU.

I. Проблема памяти: детальный анализ компонентов

Чтобы понять гениальность QLoRA, необходимо детально разобраться в том, какую память потребляет процесс тонкой настройки. Проблема не сводится лишь к хранению весов модели. Выделяют три основных "пожирателя" памяти:

1. **Веса модели:** Самые объемные данные. Модель на 7 миллиардов параметров в формате BF16 занимает примерно 14 ГБ памяти.
2. **Градиенты:** Для каждого обучаемого параметра необходимо вычислять и хранить его градиент, что требует еще ~14 ГБ для полного дообучения нашей модели-примера.
3. **Состояния оптимизатора:** Такие оптимизаторы, как Adam, хранят как минимум два момента (первый и второй) для каждого обучаемого параметра, что удваивает или даже утраивает объем памяти, отведенной под градиенты.

Суммируя, даже с использованием LoRA, которая drastically сокращает пункты 2 и 3, пункт 1 — загрузка базовой модели — оставался камнем преткновения. QLoRA атакует именно эту, самую крупную составляющую.

Рассмотрим модель с 7 миллиардами параметров (7B):

* **Веса модели в FP16/BF16:** 7 × 10^9 параметров × 2 байта/параметр ≈ 14 ГБ
* **Градиенты (для полной настройки):** ~14 ГБ (аналогично весам)
* **Состояния оптимизатора Adam (первый и второй моменты):** 2 момента × 14 ГБ = 28 ГБ
* **Итого для полного дообучения:** 14 + 14 + 28 = 56 ГБ — недостижимо для большинства GPU!

Даже с LoRA, где обучается ~1% параметров, базовая модель в FP16 все еще требует 14 ГБ, что превышает возможности многих потребительских видеокарт.

*# Пример расчета памяти для модели 7B*

param\_count = 7\_000\_000\_000

fp16\_size\_gb = param\_count \* 2 / (1024\*\*3) *# ~14 ГБ*

*# Для LoRA: предположим rank r=64, адаптируем 4 матрицы в каждом слое*

*# Q, K, V, O проекции в attention*

lora\_params = 4 \* 2 \* 4096 \* 64 \* 32 *# упрощенный расчет для LLaMA 7B*

lora\_size\_mb = lora\_params \* 2 / (1024\*\*2) *# ~128 МБ*

print(f"Базовая модель (FP16): {fp16\_size\_gb:.1f} ГБ")

print(f"LoRA адаптеры: {lora\_size\_mb:.1f} МБ")

print(f"Экономия: {(fp16\_size\_gb\*1024)/lora\_size\_mb:.0f}x")

II. Принципы квантования: от теории к практике

Квантование — это процесс уменьшения точности числовых представлений весов модели. Стандартное 8-битное квантование уже было шагом вперед, но QLoRA идет значительно дальше, вводя два ключевых нововведения.

**4-bit NormalFloat (NF4):** Обычное равномерное квантование в 4 бита работает плохо, так как распределение весов в предобученных нейросетях не является равномерным; оно сосредоточено вокруг нуля и имеет нормальные "хвосты". NF4 — это информатически оптимальный формат данных, созданный специально для такого нормального распределения. Он нелинейно распределяет ограниченное количество 4-битных значений (16 возможных) так, чтобы они чаще попадали в области, где сосредоточены большинство весов, минимизируя ошибку квантования. Это позволяет хранить модель в памяти, используя в 4 раза меньше места, чем в формате BF16, с минимальной деградацией производительности.

**Double Quantization (DQ):** Даже процесс квантования требует хранения своих собственных метаданных — констант квантования (quantization constants), которые преобразуют целочисленные 4-битные значения обратно в числа с плавающей точкой. Для больших моделей накладные расходы на эти константы становятся значительными. Double Quantization решает эту проблему, применяя к самим константам квантования... еще одно квантование. Это рекурсивное сжатие позволяет сократить память, занимаемую константами, в среднем на 0.37 бита на параметр, что является решающим фактором для достижения общего 4-битного представления.

import torch

import torch.nn as nn

class NF4Quantizer:

def \_\_init\_\_(self):

*# Оптимальные значения для нормального распределения*

self.nf4\_values = torch.tensor([

-1.0, -0.6961928009986877, -0.5250730514526367, -0.39491748809814453,

-0.28444138169288635, -0.18477343022823334, -0.09105003625154495, 0.0,

0.07958029955625534, 0.16093020141124725, 0.24611230194568634, 0.33791524171829224,

0.44070982933044434, 0.5626170039176941, 0.7229568362236023, 1.0

], dtype=torch.float16)

def quantize(self, tensor):

*# Находим ближайшие значения NF4 для каждого элемента*

expanded\_tensor = tensor.unsqueeze(-1)

expanded\_nf4 = self.nf4\_values.reshape(1, 1, -1)

distances = torch.abs(expanded\_tensor - expanded\_nf4)

indices = torch.argmin(distances, dim=-1)

return indices, self.nf4\_values[indices]

*# Пример использования*

quantizer = NF4Quantizer()

original\_weights = torch.randn(100, 100, dtype=torch.float16)

indices, quantized\_weights = quantizer.quantize(original\_weights)

print(f"Оригинальный размер: {original\_weights.nelement() \* 2} байт")

print(f"Квантованный размер: {indices.nelement() \* 0.5} байт") *# 4 бита = 0.5 байта*

print(f"Коэффициент сжатия: {(original\_weights.nelement() \* 2) / (indices.nelement() \* 0.5):.1f}x")

## §22. Пример реализации QLoRA

Если LoRA решает проблему сокращения числа обучаемых параметров, то QLoRA (Quantized Low-Rank Adaptation) идет дальше, решая проблему памяти, требуемой для хранения самих весов модели. Предыдущие разделы использовали модели, загруженные в полной точности (даже с bf16, это все еще требует значительной памяти) или с использованием смешанной точности. QLoRA вводит дополнительный уровень оптимизации: 4-битная квантизация (4-bit quantization), которая радикально снижает потребление памяти, необходимой для хранения исходных весов модели.

Квантизация — это процесс представления чисел с плавающей запятой высокой точности (например, float32 или bfloat16) с использованием целых чисел или чисел меньшей разрядности. При 4-битной квантизации каждый вес модели представляется 4 битами вместо 32 или 16 битов, что теоретически дает 8-кратное сокращение памяти. На практике достигается 4–6-кратное сокращение благодаря накладным расходам на масштабирующие коэффициенты и другие метаданные квантизации.

Ключевая идея QLoRA состоит в комбинировании двух подходов: веса исходной модели хранятся в квантизованном виде (4-bit), а обучаемые LoRA-адаптеры хранятся в полной точности. Это позволяет достичь компромисса между экономией памяти и сохранением гибкости обучения.

* 1. Квантизация и конфигурация BitsAndBytes

Квантизация в PyTorch и HuggingFace реализуется через библиотеку BitsAndBytes. Эта библиотека предоставляет оптимизированные CUDA-ядра для работы с квантизованными весами, позволяя проводить вычисления с минимальными потерями производительности несмотря на сокращение точности.

from transformers import BitsAndBytesConfig

from peft import prepare\_model\_for\_kbit\_training

bnb\_config = BitsAndBytesConfig(

    load\_in\_4bit=True,

    bnb\_4bit\_quant\_type="nf4",

    bnb\_4bit\_compute\_dtype=torch.bfloat16,

    bnb\_4bit\_use\_double\_quant=True,

)

Параметр load\_in\_4bit=True активирует 4-битную квантизацию при загрузке модели. Это означает, что вместо загрузки весов в их исходной точности, они будут сразу квантизованы при загрузке с диска в оперативную память GPU.

Параметр bnb\_4bit\_quant\_type="nf4" выбирает тип квантизации. NormalFloat4 (NF4) — это специально разработанный формат квантизации, оптимизированный для нейронных сетей. Он основывается на предположении, что распределение весов в нейронных сетях близко к нормальному распределению, и использует это для оптимального выбора уровней квантизации. Альтернативный выбор, "int4", использует стандартное целочисленное квантизование, но NF4 обычно обеспечивает лучшее качество при сопоставимой сложности.

Параметр bnb\_4bit\_compute\_dtype=torch.bfloat16 определяет тип данных, используемый для вычислений. Во время forward pass веса извлекаются из 4-битного хранилища и преобразуются в bfloat16 для выполнения матричных умножений, после чего результаты также хранятся в bfloat16. Этот параметр является компромиссом: использование полной float32 или float64 исключило бы преимущества квантизации, тогда как bfloat16 обеспечивает хороший баланс между точностью и производительностью.

Параметр bnb\_4bit\_use\_double\_quant=True активирует двойную квантизацию (double quantization). Это означает, что сами масштабирующие коэффициенты квантизации дополнительно квантизуются, экономя еще около 0.4 бита на параметр в среднем. Хотя это кажется малым улучшением, для больших моделей это может означать ощутимую экономию памяти без заметного снижения качества.

Загрузка модели с применением квантизации требует передачи конфигурации BitsAndBytes в функцию загрузки:

model\_4\_qlora = AutoModelForCausalLM.from\_pretrained(

    "Qwen/Qwen3-0.6B",

    trust\_remote\_code=True,

    quantization\_config=bnb\_config,

    device\_map="auto"

)

Параметр quantization\_config указывает, что модель должна быть загружена с применением конфигурации квантизации. Процесс загрузки в этом случае отличается от стандартной загрузки: вместо загрузки весов в GPU памяти в их исходной форме, они загружаются с интернета (или с локального кеша) в CPU памяти, затем квантизуются (остаются на CPU), и наконец транспортируются в GPU в квантизованной форме.

Время загрузки модели при использовании квантизации может быть немного выше, чем без неё, так как процесс квантизации требует дополнительных вычислений. Однако для моделей среднего и большого размера эта дополнительная задержка (обычно несколько секунд) окупается значительной экономией памяти.

Ключевая функция, специфичная для QLoRA, – это prepare\_model\_for\_kbit\_training():

from peft import prepare\_model\_for\_kbit\_training

model\_4\_qlora = prepare\_model\_for\_kbit\_training(model\_4\_qlora)

Эта функция выполняет несколько критических модификаций к квантизованной модели, необходимых для успешного обучения. В частности, она:

* Отключает cache в слоях внимания, чтобы избежать конфликтов между квантизацией и кешированием активаций.
* Добавляет input-градиенты к квантизованным параметрам, хотя сами эти параметры останутся заморожены.
* Модифицирует backward pass так, чтобы градиенты вычислялись корректно несмотря на квантизацию.
* Применяет gradient checkpointing, если требуется, для дополнительной экономии памяти.

Без этого шага попытка обучения квантизованной модели обычно приводит к ошибкам, связанным с несовместимостью операций или неверными вычислениями градиентов.

Применение LoRA к квантизованной модели требует небольших, но значимых изменений в конфигурации LoRA по сравнению с обычным LoRA:

lora\_config = LoraConfig(

    r=32,

    lora\_alpha=64,

    target\_modules="all-linear",

    lora\_dropout=0.05,

    bias="none",

    task\_type=TaskType.CAUSAL\_LM

)

Наиболее значимое изменение — это target\_modules="all-linear". Вместо явного перечисления имен модулей (как ["q\_proj", "v\_proj", "k\_proj", "o\_proj"] в обычном LoRA), здесь используется специальное значение "all-linear", которое применяет LoRA ко всем линейным слоям модели. Это включает не только проекции внимания, но также полносвязные слои (feed-forward networks) и другие линейные модули.

Причина расширения области применения LoRA при использовании QLoRA заключается в том, что квантизация весов исходной модели уже значительно сокращает роль исходных весов как "якоря" для обучения. Когда веса квантизованы и замораживаются, LoRA-адаптеры становятся единственным каналом для адаптации, поэтому имеет смысл максимизировать их покрытие. Кроме того, поскольку сами квантизованные веса занимают очень мало памяти, добавление LoRA к большему числу слоев не существенно увеличивает общее потребление памяти.

Параметр lora\_dropout=0.05 немного ниже, чем в примере обычного LoRA (где использовалось 0.1). Это обоснованно, так как QLoRA уже работает с меньшим числом параметров благодаря квантизации, и дополнительная регуляризация через dropout может быть менее критичной. Однако на практике оба значения часто показывают сопоставимые результаты.

Применение LoRA к квантизованной модели выполняется точно так же, как к обычной модели:

qlora\_model = get\_peft\_model(model\_4\_qlora, lora\_config)

qlora\_model.print\_trainable\_parameters()

Однако интерпретация результата print\_trainable\_parameters() при QLoRA существенно отличается. В примере full fine-tuning все параметры модели были обучаемы. В примере LoRA часть параметров была обучаема (LoRA-матрицы), а часть заморожена (исходные веса). В примере QLoRA ситуация еще более асимметрична: исходные веса в квантизованной форме занимают минимум памяти и полностью заморожены, а обучаемы только LoRA-адаптеры.

**Критическая конфигурация: отключение cache**

Важная деталь, которая часто упускается:

qlora\_model.config.use\_cache = False

Это отключает использование cache в слоях внимания. Cache использует промежуточные активации из предыдущих шагов для ускорения инференса при автурегрессивной генерации. Однако при обучении с gradient checkpointing и квантизацией использование cache может привести к несовместимостям и ошибкам. Отключение cache гарантирует, что каждый forward pass полностью пересчитывает активации, что совместимо с gradient checkpointing и квантизацией.

Отключение cache имеет практический побочный эффект: обучение становится медленнее, так как теряется возможность переиспользовать промежуточные вычисления. Однако экономия памяти, достигаемая отключением cache, часто превосходит возможность обучения с большим батчем, что в итоге может привести к лучшей общей производительности.

Параметры обучения для QLoRA отличаются от обычного LoRA в нескольких важных аспектах:

training\_args = TrainingArguments(

    output\_dir="./qlora\_outputs",

    per\_device\_train\_batch\_size=2,

    per\_device\_eval\_batch\_size=2,

    num\_train\_epochs=3,

    gradient\_accumulation\_steps=8,

    optim="adamw\_torch",

    learning\_rate=2e-4,

    fp16=False,

    bf16=True,

    save\_steps=100,

    logging\_steps=10,

    logging\_strategy="steps",

    logging\_first\_step=True,

    eval\_strategy="steps",

    eval\_steps=10,

    report\_to=[],

    remove\_unused\_columns=False,

    load\_best\_model\_at\_end=True,

    metric\_for\_best\_model="loss",

)

Скорость обучения learning\_rate=2e-4 немного выше, чем для обычного LoRA (который использовал 1e-4). Это обоснованно, так как LoRA-адаптеры при QLoRA покрывают большее число слоев и должны компенсировать квантизацию исходных весов. Более высокая скорость обучения позволяет адаптерам быстрее "найти" оптимальные обновления.

Остальные параметры в целом остаются такими же, как для обычного LoRA и full fine-tuning, что демонстрирует универсальность подхода: QLoRA интегрируется в существующий пайплайн практически без изменений в конфигурации обучения.

**Инициализация тренера и процесс обучения**

Инициализация callback и Trainer остается идентичной предыдущим подходам:

metrics\_callback = MetricsCallback()

trainer = Trainer(

    model=qlora\_model,

    args=training\_args,

    train\_dataset=train\_ds,

    eval\_dataset=val\_ds,

    tokenizer=tokenizer,

    callbacks=[metrics\_callback],

)

Обучение инициируется стандартным образом:

print("Starting training...")

start\_time = time.time()

trainer.train()

end\_time = time.time()

training\_time = (end\_time - start\_time) / 60

print(f"Training time: {training\_time:.2f} min.")

print(f"GPU VRAM used: {torch.cuda.max\_memory\_allocated() // (1024\*\*2)} MB")

Однако, благодаря квантизации, потребление памяти QLoRA должно быть существенно ниже, чем при обычном LoRA. Это позволяет либо использовать больший батч с той же доступной памятью, либо обучать на устройствах с меньшей памятью. Время обучения может быть немного выше из-за дополнительных вычислений, связанных с деквантизацией весов при каждом forward pass, но часто этот оверхед минимален благодаря оптимизированным CUDA-ядрам BitsAndBytes.

Собранные данные о времени обучения и использованной памяти особенно ценны для QLoRA, так как они демонстрируют практический выигрыш метода в реальных сценариях.

|  |  |  |
| --- | --- | --- |
| Step | Training Loss | Validation Loss |
| 10 | 2.669600 | 2.094725 |
| 20 | 1.964600 | 1.969030 |
| 30 | 2.192500 | 1.964407 |
| 40 | 2.316200 | 1.951967 |
| 50 | 1.761500 | 1.941816 |
| 60 | 1.594200 | 2.021060 |
| 70 | 1.464800 | 2.010175 |
| 80 | 1.613100 | 1.998138 |
| 90 | 1.311700 | 2.033049 |
| 100 | 0.984400 | 2.289455 |
| 110 | 1.186400 | 2.245235 |
| 120 | 0.963100 | 2.173207 |
| 130 | 0.951000 | 2.210712 |
| 140 | 0.679000 | 2.419559 |
| 150 | 0.685100 | 2.546333 |
| 160 | 0.643200 | 2.534716 |
| 170 | 0.642300 | 2.513818 |

Время обучения QLoRA: 40 мин

Использованная память GPU: 4.5 GB

Интересный вопрос, который возникает при использовании QLoRA — насколько квантизация влияет на финальную производительность модели. Теоретически, при хорошо выбранных параметрах квантизации, потеря качества должна быть минимальной. На практике, для большинства задач, QLoRA достигает производительности, крайне близкой к обычному LoRA и часто сопоставимой с full fine-tuning. Это демонстрирует устойчивость языковых моделей к квантизации и является одной из причин популярности QLoRA в практических приложениях.

Визуализация остается структурно идентичной предыдущим примерам:

fig, axes = plt.subplots(1, 2, figsize=(15, 5))

axes.plot(metrics\_callback.train\_steps, metrics\_callback.train\_losses, label='Train Loss', marker='o')

axes.plot(metrics\_callback.eval\_steps, metrics\_callback.eval\_losses, label='Validation Loss', marker='s')

axes.set\_xlabel('Steps')

axes.set\_ylabel('Loss')

axes.set\_title('Training and Validation Loss (QLoRA)')

axes.legend()

axes.grid(True)

categories = ['Baseline\n(Val)', 'QLoRA\n(Val)', 'QLoRA\n(Test)']

accuracies = [baseline\_val\_acc, finetuned\_val\_acc, finetuned\_test\_acc]

colors = ['red', 'green', 'blue']

axes.bar(categories, accuracies, color=colors, alpha=0.7)

axes.set\_ylabel('Accuracy')

axes.set\_title('Model Accuracy Comparison (QLoRA)')

axes.set\_ylim([0, max(accuracies) \* 1.2])

for i, (cat, acc) in enumerate(zip(categories, accuracies)):

    axes.text(i, acc + 0.01, f'{acc:.4f}', ha='center', va='bottom')

plt.tight\_layout()

plt.savefig('qlora\_training\_results.png', dpi=300, bbox\_inches='tight')

plt.show()

При анализе графиков обучения QLoRA может наблюдаться более гладкая траектория потерь благодаря использованию большего числа слоев с LoRA-адаптерами. Однако возможны также колебания, вызванные чувствительностью обучения к квантизации на ранних этапах.

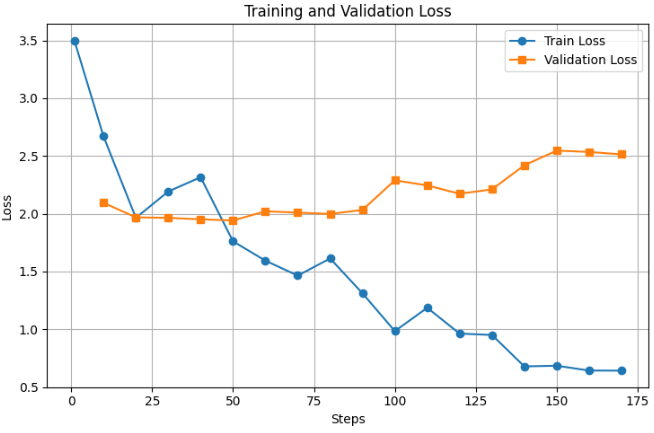


График потерь (тренировочная и валидационная) во время обучения QLoRA

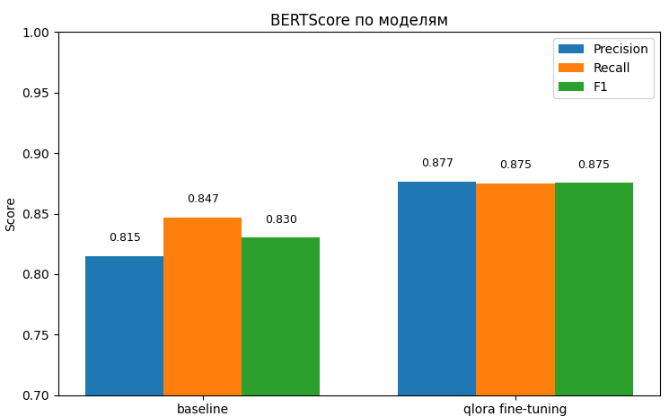


График сравнения точности (базовая vs QLoRA)

**Инференс с QLoRA-моделью**

Инференс с QLoRA-моделью выполняется абсолютно аналогично LoRA:

test\_prompt = "Question: What is a healthy breakfast?\nContext: Oatmeal with fruits and nuts provides a balanced breakfast.\nAnswer:"

inputs = tokenizer(test\_prompt, return\_tensors="pt").to(qlora\_model.device)

outputs = qlora\_model.generate(\*\*inputs, max\_new\_tokens=64)

print("\n=== Sample prediction ===")

print(tokenizer.decode(outputs, skip\_special\_tokens=True))

Скорость инференса QLoRA обычно немного ниже, чем LoRA, из-за необходимости деквантизации весов при каждом forward pass. Однако благодаря оптимизированным операциям BitsAndBytes, оверхед обычно составляет менее 20% и часто незаметен на практике. Для некоторых устройств с ограниченной пропускной способностью памяти квантизация может даже ускорить инференс, так как меньше данных требуется загружать из памяти.

При применении QLoRA в реальных сценариях следует учитывать несколько рекомендаций. Конфигурация квантизации с nf4=True и bnb\_4bit\_use\_double\_quant=True обеспечивает хороший баланс между экономией памяти и качеством, и рекомендуется для большинства приложений.

Использование target\_modules="all-linear" вместо явного перечисления модулей упрощает конфигурацию и часто дает лучшие результаты, так как позволяет LoRA адаптерам охватить все потенциально полезные слои. Однако для специфических задач, где известны ключевые слои для адаптации, можно ограничить LoRA только этими слоями для экономии памяти и сокращения времени обучения.

Скорость обучения может потребовать настройки в зависимости от задачи. Как правило, QLoRA с target\_modules="all-linear" требует немного более высокой скорости обучения, чем обычное LoRA с ограниченным числом целевых слоев. Рекомендуется начать с learning\_rate=2e-4 и, при необходимости, экспериментировать с значениями в диапазоне 1e-4 — 5e-4.

Критический момент — всегда использовать prepare\_model\_for\_kbit\_training() после загрузки квантизованной модели и всегда отключать cache (use\_cache=False) перед обучением. Пропуск этих шагов приводит к сложно диагностируемым ошибкам.

**Сравнение: Full Fine-Tuning, LoRA и QLoRA**

На этом этапе лекции становится полезным проведение систематического сравнения трех подходов, охватываемых в главе 7. Full Fine-Tuning обновляет все параметры модели, требуя максимальной памяти и вычислительных ресурсов, но часто обеспечивает наивысшее качество адаптации. LoRA вводит обучаемые матрицы низкого ранга, сокращая число обучаемых параметров на несколько порядков, при этом сохраняя производительность, близкую к full fine-tuning, и значительно снижая требования по памяти. QLoRA добавляет 4-битную квантизацию исходных весов, дополнительно сокращая памяти для самих весов, делая fine-tuning доступным даже на GPU с 8-16GB VRAM.

Выбор между тремя подходами зависит от конкретных условий: доступных ресурсов, размера модели, размера датасета адаптации и требуемого качества. Для исследовательских проектов с ограниченными ресурсами QLoRA часто является оптимальным выбором. Для больших производственных систем, где ресурсы не ограничены и требуется максимальное качество, full fine-tuning может быть оправдан. Для промежуточных сценариев LoRA предоставляет привлекательный компромисс.

QLoRA демонстрирует, как комбинирование нескольких техник оптимизации (квантизация + низкоранговые адаптеры) может привести к синергетическому эффекту, достигая результатов, близких к полному fine-tuning, с использованием лишь доли требуемых ресурсов. Практическое значение QLoRA трудно переоценить: это метод, который буквально расширил границы возможного для независимых исследователей и компаний с ограниченными ресурсами. С появлением QLoRA задачи, которые ранее требовали дорогостоящих облачных кластеров, стали решаемы на бытовых GPU. Это демонстрирует, как инновации в машинном обучении не обязательно требуют более больших моделей или больших датасетов — часто они требуют умного комбинирования существующих техник для максимизации эффективности.