

с июля

СТРОИМ

LLM

Себастьян Рашка



Build a Large Language Model (From Scratch)

SEBASTIAN RASCHKA



MANNING

SHELTER ISLAND

Строим LLM с нуля

СЕБАСТЬЯН РАШКА



Санкт-Петербург • Москва • Минск

2025

Себастьян Рашка

Строим LLM с нуля

Серия «Библиотека программиста»

Перевел на русский О. Окунь

Научный редактор Д. Бардин

ББК 32.813+32.973.2-018.1

УДК 004.8+004.432.45

Рашка Себастьян

P28 Строим LLM с нуля. — СПб.: Питер, 2025. — 384 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-4325-2

Говорят, что физик Ричард Фейнман однажды сказал: «Я не понимаю того, чего не могу создать». Основываясь на этом же важном принципе, автор бестселлеров Себастьян Рашка шаг за шагом ведет вас к созданию LLM в стиле GPT, которую вы сможете запустить на своем ноутбуке. Это увлекательная книга, которая охватывает каждый этап процесса — от планирования и кодирования до обучения и тонкой настройки.

«Строим LLM с нуля» — это чрезвычайно интересное путешествие в основы генеративного ИИ. Не полагаясь на существующие библиотеки LLM, вы реализуете в коде базовую модель, превратите ее в классификатор текста и в конечном счете создадите чат-бот, который сможет следовать вашим инструкциям в диалоге. И вы действительно поймете LLM, потому что создали ее сами!

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1633437166 англ.

Authorized translation of the English edition © 2025 Manning Publications.
This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

ISBN 978-5-4461-4325-2

© Перевод на русский язык ООО «Прогресс книга», 2025
© Издание на русском языке, оформление ООО «Прогресс книга», 2025
© Серия «Библиотека программиста», 2025

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес:
194044, Россия, г. Санкт-Петербург, Б. Сапсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 28.08.25. Формат 70×100/16. Бумага офсетная. Усл. п. л. 30,960. Тираж 700. Заказ 0000.

Краткое содержание

Предисловие	11
Глава 1. Знакомство с большими языковыми моделями.....	22
Глава 2. Работа с текстовыми данными	40
Глава 3. Программирование механизмов внимания	76
Глава 4. Создание GPT-подобной модели для генерации текста с нуля	122
Глава 5. Предварительное обучение на неразмеченных данных	160
Глава 6. Тонкая настройка по классификации.....	204
Глава 7. Тонкая настройка по инструкциям	242
Приложение А. Введение в PyTorch	292
Приложение Б. Ссылки и дополнительные источники	335
Приложение В. Решения упражнений	347
Приложение Г. Добавление новых возможностей в процесс обучения	359
Приложение Д. Эффективная настройка параметров с помощью LoRA	368

Оглавление

Предисловие	11
Благодарности.....	13
О книге.....	15
Кому следует прочитать эту книгу	15
Структура издания.....	16
О коде.....	17
Другие онлайн-ресурсы.....	18
Об авторе.....	19
Иллюстрация на обложке	20
От издательства	21
О научном редакторе русскоязычного издания.....	21
Глава 1. Знакомство с большими языковыми моделями.....	22
1.1. Что такое LLM	23
1.2. Применение больших языковых моделей	25
1.3. Зачем создавать собственные модели.....	27
1.4. Этапы обучения LLM	27
1.5. Введение в архитектуру трансформера.....	29
1.6. Использование больших выборок данных.....	33
1.7. Более подробный обзор архитектуры GPT.....	34
1.8. План создания LLM с нуля.....	36
Итоги главы	38

Глава 2. Работа с текстовыми данными.....	40
2.1. Векторные представления слов.....	42
2.2. Токенизация текста	45
2.3. Преобразование токенов в идентификаторы токенов.....	49
2.4. Добавление специальных контекстных токенов.....	53
2.5. Кодирование пар байтов.....	57
2.6. Выборка данных с помощью скользящего окна.....	60
2.7. Создание векторных представлений токенов	66
2.8. Кодирование позиций слов	70
Итоги главы	75
Глава 3. Программирование механизмов внимания.....	76
3.1. Проблема моделирования длинных последовательностей	78
3.2. Улавливание зависимостей данных с помощью механизмов внимания.....	81
3.3. Обращение к разным частям входных данных с помощью самовнимания	83
3.3.1. Простой механизм самовнимания без обучаемых весов.....	83
3.3.2. Вычисление весов внимания для всех входных токенов	89
3.4. Реализация самовнимания с обучаемыми весами.....	91
3.4.1. Пошаговое вычисление весовых коэффициентов внимания.....	93
3.4.2. Реализация компактного класса Python для самовнимания.....	98
3.5. Соккрытие будущих слов с помощью причинно-следственного внимания.....	102
3.5.1. Применение маски причинно-следственного внимания	103
3.5.2. Маскирование дополнительных весов внимания с помощью отсева.....	107
3.5.3. Реализация компактного класса причинно-следственного внимания.....	109
3.6. Расширение одноцелевого внимания до многоцелевого	111
3.6.1. Объединение нескольких одноцелевых слоев внимания	112
3.6.2. Реализация многоцелевого внимания с разделением весов.....	115
Итоги главы	121

Глава 4. Создание GPT-подобной модели для генерации текста с нуля.....	122
4.1. Программирование архитектуры LLM	124
4.2. Нормализация активаций с помощью нормализации слоев	130
4.3. Реализация сети с прямым распространением и активацией GELU... ..	136
4.4. Добавление коротких соединений	140
4.5. Объединение механизма внимания и линейных слоев в блоке трансформера	144
4.6. Программирование модели GPT	148
4.7. Генерация текста	154
Итоги главы	159
Глава 5. Предварительное обучение на неразмеченных данных	160
5.1. Оценка генеративных текстовых моделей	162
5.1.1. Использование GPT для генерации текста.....	163
5.1.2. Расчет потерь при генерации текста.....	165
5.1.3. Расчет потерь для обучающей и проверочной выборки	173
5.2. Обучение LLM	179
5.3. Стратегии декодирования для контроля случайности	185
5.3.1. Масштабирование температуры	186
5.3.2. Выборка по принципу top-k	190
5.3.3. Изменение функции генерации текста.....	192
5.4. Загрузка и сохранение весов модели в PyTorch.....	193
5.5. Загрузка предварительно обученных весов из OpenAI	195
Итоги главы	203
Глава 6. Тонкая настройка по классификации	204
6.1. Различные категории тонкой настройки	206
6.2. Подготовка данных.....	208
6.3. Создание загрузчиков данных.....	211
6.4. Инициализация модели с предварительно обученными весами	218
6.5. Добавление цели классификации	220
6.6. Расчет потерь и точности классификации	228
6.7. Тонкая настройка модели на данных с метками	232
6.8. Использование LLM в качестве классификатора спама	239
Итоги главы	241

Глава 7. Тонкая настройка по инструкциям.....	242
7.1. Введение в тонкую настройку по инструкциям	244
7.2. Подготовка набора данных для контролируемой тонкой настройки по инструкциям	244
7.3. Организация данных в обучающие пакеты.....	249
7.4. Создание загрузчиков данных для набора инструкций	263
7.5. Загрузка предварительно обученной LLM	266
7.6. Тонкая настройка LLM по инструкциям.....	269
7.7. Извлечение и сохранение ответов.....	274
7.8. Оценка точно настроенной LLM	279
7.9. Выводы.....	288
7.9.1. Что дальше.....	290
7.9.2. Будьте в курсе последних событий в быстро меняющейся области	290
7.9.3. Заключительные слова.....	290
Итоги главы	291
 Приложение А. Введение в PyTorch	 292
А.1. Что такое PyTorch.....	292
А.1.1. Три основных компонента PyTorch	293
А.1.2. Определение глубокого обучения	294
А.1.3. Установка PyTorch	296
А.2. Тензоры.....	300
А.2.1. Скаляры, векторы, матрицы и тензоры	301
А.2.2. Типы тензорных данных.....	301
А.2.3. Распространенные операции с тензорами в PyTorch.....	302
А.3. Представление моделей в виде графов вычислений.....	304
А.4. Автоматическое дифференцирование упростилось	305
А.5. Реализация многослойных нейронных сетей.....	308
А.6. Настройка эффективных загрузчиков данных.....	313
А.7. Типичный цикл обучения	318
А.8. Сохранение и загрузка моделей	323
А.9. Оптимизация производительности обучения с помощью графических процессоров	323
А.9.1. Вычисления PyTorch на графических процессорах	324

А.9.2. Обучение с использованием одного графического процессора	325
А.9.3. Обучение с использованием нескольких графических процессоров.....	327
Итоги приложения	334
Приложение Б. Ссылки и дополнительные источники	335
Приложение В. Решения упражнений	347
Приложение Г. Добавление новых возможностей в процесс обучения	359
Г.1. Переменная скорость обучения	360
Г.2. Затухание по косинусу.....	362
Г.3. Ограничение градиента.....	364
Г.4. Измененная функция обучения.....	365
Приложение Д. Эффективная настройка параметров с помощью LoRA...	368
Д.1. Введение в LoRA	368
Д.2. Подготовка набора данных	370
Д.3. Инициализация модели	373
Д.4. Эффективная тонкая настройка параметров с помощью LoRA.....	375

Предисловие

Я всегда был очарован языковыми моделями. Более десяти лет назад мой путь в ИИ начался с курса статистической классификации образов, который привел меня к моему первому независимому проекту: разработке модели и интернет-приложения для определения тональности песни по ее тексту.

Перенесемся в 2022 год, когда с выходом ChatGPT большие языковые модели (large language models, LLM) вызвали небывалый ажиотаж в мире и кардинально изменили подход к работе для многих из нас. Эти модели невероятно универсальны и помогают в таких задачах, как проверка грамматики, составление электронных писем, обобщение содержания текстовых документов и многое другое. Такой прогресс связан со способностью LLM анализировать и генерировать текст, похожий на человеческий, что важно в различных сферах, от обслуживания клиентов до создания контента, и даже в технических областях, таких как программирование и анализ данных.

Как следует из названия, отличительной чертой больших языковых моделей является их размер (large) — очень большой, насчитывающий миллионы и даже миллиарды параметров (для сравнения: набор данных о цветах ириса можно классифицировать с точностью более 90 %, задействуя небольшую модель всего с двумя параметрами и используя более традиционные методы машинного обучения или статистики). Однако, несмотря на большой размер по сравнению с более традиционными методами, LLM необязательно должны быть «черным ящиком».

В этой книге вы узнаете, как создавать большие языковые модели. К концу книги вы будете хорошо понимать, как LLM, подобные тем, которые используются в ChatGPT, работают на базовом уровне. Я считаю, что для достижения успеха важно хорошо разбираться в каждой части фундаментальных концепций и базового кода. Это не только помогает устранять ошибки в коде и повышать

продуктивность ежедневного труда, но и позволяет экспериментировать с новыми идеями.

Несколько лет назад, когда я начинал работать с большими языковыми моделями, мне пришлось на собственном опыте изучать способы их реализации на практике, просматривая множество научных статей и репозиторий кода, чтобы получить общее представление об этой теме. Я надеюсь сделать LLM более доступными, разработав и представив в этой книге пошаговое руководство по их реализации, в котором подробно описаны все основные компоненты и этапы разработки большой языковой модели.

Я твердо убежден, что лучший способ понять LLM — написать ее код с нуля.

И вы увидите, что этот процесс может быть очень увлекательным!

Приятного чтения и программирования!

Благодарности

Написание книги — важное дело, и я хотел бы выразить искреннюю благодарность моей жене Лайзе за ее терпение и поддержку на протяжении всего этого процесса. Ее безусловная любовь и постоянное поощрение были крайне необходимы.

Я сердечно благодарен Дэниэлу Кляйну за бесценные отзывы о содержании глав и коде. Его внимательное отношение к деталям и содержательные предложения помогли улучшить эту книгу, чтобы ее чтение стало более легким и приятным.

Кроме того, я хотел бы поблагодарить замечательных сотрудников издательства Manning Publications, в том числе Майкла Стивенса, за множество продуктивных обсуждений, которые помогли сфокусировать направление развития этой книги, а также Дастина Арчибальда, чья конструктивная критика и рекомендации по соблюдению инструкций по оформлению были крайне важны. Я также ценю его гибкость в удовлетворении уникальных требований моего нетрадиционного подхода к объяснению материала с нуля. Я выражаю особую благодарность Александру Драгосавлевичу, Кари Лакке и Майку Биди за их работу над профессиональным макетом книги, а также Сюзан Ханивэлл и ее команде за доработку графики.

Я хочу выразить искреннюю благодарность Робин Кэмпбелл и ее выдающейся команде маркетологов за неоценимую поддержку на протяжении всего процесса написания книги.

Наконец, я выражаю благодарность рецензентам: Анандаганешу Балакришнану, Анто Аравинту, Аюшу Бихани, Бассаму Исмаилу, Бенджамину Маскалла, Бруно Соннино, Кристиану Прокоппу, Дэниэлю Кляйну, Дэвиду Каррану, Дибьенду Рою Чоудхури, Гэри Пассу, Георгу Зоммеру, Джованни Альцетта,

14 Благодарности

Гильермо Алькантара, Джонатану Ривзу, Куналу Гошу, Николасу Модржику, Полу Силистеану, Раулю Чотеску, Скотту Лингу, Шрираму Махарла, Сумиту Палу, Вахиду Мирджалили, Вайджанату Рао и Уолтеру Риду за их подробные комментарии о черновиках книги. Ваши ценные замечания помогли улучшить ее качество.

Я искренне благодарен всем, кто участвовал в этом проекте. Ваша поддержка, опыт и преданность делу сыграли важную роль в создании данной книги. Спасибо вам!

О книге

Книга призвана помочь вам с нуля создавать собственные большие языковые модели, подобные GPT. В начале книги основное внимание уделяется основам работы с текстовыми данными и программированию механизмов внимания, а затем описываются способы создания модели GPT с нуля. Далее рассматривается механизм предварительного обучения, а также тонкая настройка для конкретных задач, таких как классификация текста и выполнение инструкций. Прочитав книгу, вы будете хорошо понимать, как работают LLM, и научитесь создавать собственные. Ваши модели будут меньше по размеру, чем обычные LLM, но будут использовать те же концепции и служить мощными образовательными инструментами, которые помогут понять основные механизмы и методы, используемые при создании современных больших языковых моделей.

Кому следует прочитать эту книгу

Книга предназначена для приверженцев машинного обучения, инженеров, исследователей, студентов и практиков, которые хотят получить глубокое представление о том, как работают LLM, и научиться создавать собственные модели с нуля. Как новички, так и опытные разработчики, применяя уже имеющиеся навыки и знания, смогут понять концепции и методы, используемые при создании LLM.

Отличие этой книги от других состоит в том, что в ней подробно описан весь процесс создания большой языковой модели: от работы с наборами данных до реализации архитектуры модели, предварительного обучения на неразмеченных данных и тонкой настройки для конкретных задач. На момент написания книги ни один другой ресурс не предлагает такого полного и практического подхода к созданию LLM с нуля.

Чтобы понять примеры кода в данной книге, вы должны хорошо разбираться в программировании на Python. Знакомство с машинным обучением, глубоким обучением и искусственным интеллектом (ИИ) может быть полезным, однако обширные знания в этих областях не требуются. Большие языковые модели — уникальное подмножество ИИ, поэтому, даже если вы новичок в области LLM, вы сумеете разобраться в материале этой книги.

Если у вас есть опыт работы с глубокими нейронными сетями, то некоторые концепции могут показаться вам более знакомыми, поскольку LLM основаны на этих моделях. Однако знание PyTorch не является обязательным. В приложении А представлено краткое введение в PyTorch, которое поможет вам освоить навыки, необходимые для понимания примеров кода в книге.

Понимание математики на уровне средней школы, особенно работа с векторами и матрицами, может быть полезным при изучении внутреннего устройства LLM. Тем не менее ключевые концепции и идеи, представленные в этой книге, не требуют глубоких математических знаний.

Самое важное — наличие опыта программирования на Python. Обладая им, вы будете хорошо подготовлены к изучению больших языковых моделей и сможете понять концепции и примеры кода, представленные в книге.

Структура издания

Книга содержит семь глав, в которых рассматриваются основные понятия и особенности больших языковых моделей и их реализация. Главы нужно читать последовательно, поскольку каждая из них опирается на концепции и методы, представленные в предыдущих главах.

Глава 1 является вводной. В ней на абстрактном уровне рассматриваются базовые концепции, лежащие в основе больших языковых моделей, в том числе архитектура трансформера, на которой строятся LLM на платформе ChatGPT.

Глава 2 содержит план создания большой языковой модели с нуля. Вы узнаете, как подготовить текст для обучения LLM, в том числе как разбивать его на токены слов и подслов, использовать кодировку пар байтов для расширенной токенизации, делать выборку обучающих примеров с помощью метода скользящего окна и преобразовывать токены в формат входных данных для LLM.

Глава 3 посвящена механизмам внимания в LLM. В ней описываются базовая структура самовнимания и усовершенствованный механизм самовнимания. Кроме того, в главе рассматривается реализация модуля причинно-следственного внимания, который позволяет модели генерировать по одному символу (токену) за раз, маскируя случайно выбранные весовые значения внимания отсеком, чтобы снизить риск излишнего обучения нейронной сети и объединить несколько модулей причинно-следственного внимания в модуль многозадачного внимания.

Глава 4 фокусируется на программировании LLM, подобной GPT, которую можно научить генерировать понятный человеку текст. Вы узнаете, как нормализовать значения внутренних слоев нейронной сети, чтобы стабилизировать процесс обучения; как добавлять короткие связи в глубокие нейронные сети, чтобы сделать обучение моделей более эффективным; как внедрять блоки трансформера в целях создания GPT-моделей различных размеров; как вычислять количество параметров модели и объем памяти, необходимый для ее хранения.

Глава 5 описывает процесс предварительного обучения LLM. В ней рассказывается о том, как вычислить потери на обучающей и проверочной выборках данных, чтобы оценить качество сгенерированного текста. Кроме того, вы узнаете о реализации функции обучения и предварительном обучении LLM, о сохранении и загрузке весов модели для продолжения обучения LLM, а также о загрузке предварительно обученных весов из OpenAI.

Глава 6 знакомит с различными подходами к тонкой настройке LLM. Вы узнаете, как подготовить выборку данных для классификации текста, изменять настройки предварительно обученной LLM, настраивать модель так, чтобы она была способна выявлять спам-сообщения, и оценивать точность классификатора на основе LLM с тонкой настройкой.

В главе 7 рассматривается процесс подробной настройки инструкций для LLM. Вы узнаете, как подготовить выборку данных для контролируемой настройки по инструкциям; как организовать данные инструкций в обучающих пакетах, загрузить предварительно обученную LLM и настроить ее для выполнения инструкций человека; как извлекать сгенерированные ответы на инструкции и оценивать эффективность точно настроенной LLM.

О коде

Все примеры кода в книге доступны на сайте издательства Manning по адресу <https://www.manning.com/books/build-a-large-language-model-from-scratch>, а также в формате блокнота Jupyter на GitHub по адресу <https://github.com/rasbt/LLMs-from-scratch>. Решения всех упражнений с кодом можно найти в приложении В.

В книге много примеров исходного кода приведено как в нумерованных листингах, так и внутри обычного текста. В обоих случаях исходный код оформлен шрифтом фиксированной ширины, чтобы отделить его от обычного текста.

Во многих случаях исходный код был переформатирован; были добавлены разрывы строк и изменены отступы, чтобы уместить его на странице книги. В редких случаях даже этого было недостаточно, и в листингах используются маркеры продолжения строки (➡). Кроме того, комментарии в исходном коде часто удалялись из листингов, если код был описан в тексте. Многие листинги сопровождаются аннотациями, которые выделяют важные концепции.

Одна из ключевых особенностей книги — доступность материала, поэтому примеры кода тщательно проверялись на предмет того, могут ли они эффективно работать на обычном ноутбуке. В некоторых разделах книги приведены полезные советы по масштабированию наборов данных и моделей. Если у вас есть доступ к графическому процессору, то благодаря этим советам вы сможете воспользоваться дополнительной вычислительной мощностью.

В ходе работы с книгой вам предстоит пользоваться PyTorch в качестве основной библиотеки глубокого обучения для реализации LLM с нуля. Если вы сталкиваетесь с ней впервые, то рекомендую начать с приложения А, в котором представлено подробное введение с рекомендациями по установке и настройке этой библиотеки.

Другие онлайн-ресурсы

Интересуетесь последними тенденциями в области ИИ и LLM?

- Ознакомьтесь с моим блогом на <https://magazine.sebastianraschka.com>, в котором я регулярно рассказываю о последних исследованиях в области ИИ, делая акцент на LLM.

Нужна помощь в освоении глубокого обучения и PyTorch?

- Я предлагаю несколько бесплатных курсов на своем сайте <https://sebastian-raschka.com/teaching>. Эти ресурсы помогут вам быстро освоить новейшие методы.

Ищете дополнительные материалы, связанные с книгой?

- Посетите репозиторий книги на GitHub по адресу <https://github.com/rasbt/LLMs-from-scratch>, чтобы найти дополнительные ресурсы и примеры для закрепления материала.

Об авторе



Себастьян Рашка, PhD, более десяти лет работает в области машинного обучения и искусственного интеллекта. Известен своими бестселлерами, посвященными машинному обучению на Python, и вкладом в открытый исходный код.

Себастьян — штатный инженер-исследователь в Lightning AI, специализирующийся на обучении и внедрении больших языковых моделей. До работы в этой компании был доцентом кафедры статистики в Университете Висконсин-Мэдисон, где занимался исследованиями в области глубокого обучения. Подробнее о Себастьяне можно узнать по адресу <https://sebastianraschka.com>.

Иллюстрация на обложке

Иллюстрация называется «Герцогиня» и взята из книги Анри-Леона Кюрмье *Les Français peints par eux-mêmes*. Книга опубликована в 1841 году, каждая иллюстрация в ней нарисована и раскрашена вручную.

В те времена по одежде можно было легко определить, где живут люди и чем они занимаются. Издательство Manning прославляет изобретательность и инициативность информационных технологий, создавая обложки книг, основанные на богатом разнообразии региональной культуры прошлых веков, которые возвращаются к жизни благодаря иллюстрациям из таких коллекций, как эта.

От издательства

Мы выражаем огромную благодарность клубу рецензентов ИТ-литературы ReadIT Club за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

О научном редакторе русскоязычного издания

Дмитрий Бардин — ведущий разработчик, архитектор решений, один из авторов курса «Архитектор ПО» от «Яндекс Практикума». В настоящее время занимается разработкой бэкенда «КиноПоиска» с применением языков Go и Java. В прошлом руководитель службы продуктовой разработки и ресурс-менеджер. Опыт в ИТ — более 15 лет.

Знакомство с большими языковыми моделями

В этой главе

- ✓ Концепции, лежащие в основе LLM.
- ✓ Архитектура трансформера, на базе которой созданы LLM.
- ✓ План создания LLM с нуля.

Большие языковые модели, такие как ChatGPT от OpenAI, представляют собой модели глубоких нейронных сетей, которые были разработаны за последние несколько лет. Они открыли новую эру в обработке естественного языка (natural language processing, NLP). До появления LLM традиционные методы хорошо подходили для решения задач категоризации, таких как классификация спама в электронной почте и распознавание простых закономерностей, которые можно было обнаружить с помощью разработанных вручную правил или простых моделей машинного обучения. Однако эти методы, как правило, не помогали в случае с языковыми задачами, требующими сложных навыков понимания текста, такими как анализ подробных инструкций, проведение контекстного анализа и создание логически связного текста. Например, предыдущие поколения языковых моделей не могли написать электронное письмо, взяв за основу список ключевых слов, — задача, которая для современных LLM является тривиальной.

Большие языковые модели обладают замечательными способностями понимать, генерировать и интерпретировать человеческий язык. Однако важно уточнить: под словами «языковые модели “понимают”» имеется в виду, что они могут

обрабатывать и генерировать текст так, чтобы он казался связным и контекстуально правильным с точки зрения человека, а не то, что они обладают сознанием или пониманием, подобным человеческому.

Благодаря достижениям в области глубокого обучения, которое является частью машинного обучения и искусственного интеллекта, ориентированного на нейронные сети, большие языковые модели обучаются на огромных объемах текстовых данных. Это масштабное обучение позволяет LLM улавливать более глубокий контекст и лучше (по сравнению с предыдущими подходами) разбираться в тонкостях человеческого языка. В результате модели достигли значительных результатов по широкому спектру задач NLP, таких как перевод текстов, анализ тональности текста, ответы на вопросы и многое другое.

Еще одно важное различие между современными LLM и более ранними моделями NLP заключается в том, что последние обычно разрабатывались для решения конкретных задач: категоризации текста, перевода с одного языка на другой и т. д., и преуспели в своих узких областях применения. LLM же демонстрируют более широкий спектр возможностей в различных задачах NLP.

Успех больших языковых моделей можно объяснить архитектурой трансформера, которая лежит в основе многих LLM, и огромными объемами данных, на которых обучаются модели, что позволяет им улавливать множество лингвистических нюансов, контекстов и закономерностей, которые было бы сложно запрограммировать вручную.

Этот переход к внедрению моделей, основанных на архитектуре трансформера, и использование больших выборок данных для обучения LLM коренным образом изменили обработку естественного языка, предоставив более эффективные инструменты, способные понимать человеческий язык и взаимодействовать с ним.

Основная цель этой книги — предоставить информацию, благодаря которой вы сможете пошагово создать LLM, подобную ChatGPT, используя код на основе архитектуры трансформера. В следующем разделе мы заложим основу для достижения этой цели.

1.1. Что такое LLM

Большая языковая модель — это нейронная сеть, предназначенная для понимания, генерации и обработки текста, похожего на человеческий. Такие модели представляют собой глубокие нейронные сети, обученные на огромных массивах текстовых данных, иногда содержащих значительную часть всего общедоступного текста в Интернете.

Слово «большая» в определении термина относится как к размеру модели по количеству параметров, так и к огромной выборке данных, на которой она обучается. Подобные модели часто содержат десятки или даже сотни миллиардов

параметров, которые представляют собой настраиваемые весовые коэффициенты в сети, во время обучения оптимизируемые для предсказания следующего слова в последовательности. При данном предсказывании используется присущая языку последовательность, чтобы обучать модели пониманию контекста, структуры и взаимосвязей в тексте. Это очень простая задача, и многие исследователи удивляются, как LLM может решить ее столь эффективно. В следующих главах мы подробно обсудим процедуру подобного обучения и реализуем ее на практике.

Большие языковые модели используют архитектуру, называемую *трансформером*, позволяющую уделять выборочное внимание различным частям входных данных при составлении предсказаний модели. Это делает такие модели особенно эффективными в работе с нюансами и сложностями человеческого языка.

Кроме того, LLM способны генерировать текст, поэтому их часто называют формой генеративного искусственного интеллекта, сокращенно *генеративным ИИ* или *GenAI*. Как показано на рис. 1.1, ИИ охватывает более широкую область создания машин, которые могут выполнять задачи, требующие человеческого интеллекта, такие как понимание языка, распознавание образов и принятие решений, и содержит такие области, как машинное обучение и глубокое обучение.

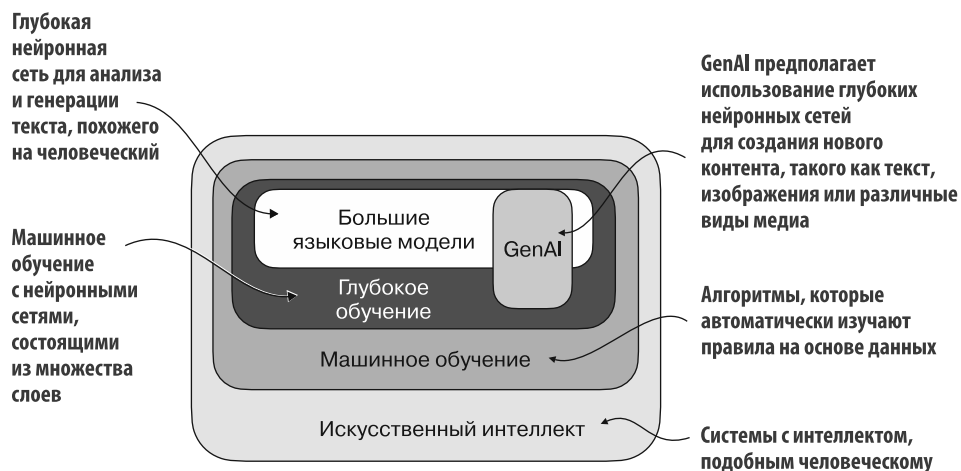


Рис. 1.1. Иерархическое описание взаимосвязи различных областей создания машин

Машинное обучение и глубокое обучение — это области, направленные на реализацию алгоритмов, которые позволяют компьютерам обучаться на основе данных и выполнять задачи, которые обычно требуют человеческого интеллекта.

Алгоритмы, используемые для реализации ИИ, являются предметом изучения в области машинного обучения. В частности, оно включает в себя разработку алгоритмов, которые могут обучаться на основе данных и генерировать предсказания или принимать решения, не нуждаясь в явном программировании.

Глубокое обучение — разновидность машинного; оно фокусируется на использовании нейронных сетей с тремя или более слоями (также называемых глубокими нейронными сетями) для моделирования сложных закономерностей и абстракций в данных.

Машинное и глубокое обучение в настоящее время доминируют в области ИИ, однако в ней применяются и другие подходы — например, использование систем, основанных на правилах, генетических алгоритмов, экспертных систем, нечеткой логики или символьных рассуждений.

В качестве примера практического применения машинного обучения представьте фильтр спама. Модель машинного обучения получает примеры электронных писем, помеченных как спам, и обычных писем. Минимизируя ошибки в своих предсказаниях на обучающей выборке данных, модель учится распознавать закономерности и характеристики, указывающие на спам, что позволяет ей классифицировать новые письма как спам или не спам.

При традиционном машинном обучении люди-эксперты могут вручную извлекать из текста электронного письма такие признаки, как частота определенных слов-триггеров (например, «приз», «выиграть», «бесплатно»), количество восклицательных знаков, использование заглавных букв или наличие подозрительных ссылок. Выборка данных, созданная на основе подобных признаков, будет использоваться для обучения модели.

В отличие от традиционного машинного, глубокое обучение не требует извлечения признаков вручную. Это означает, что экспертам не нужно определять и выбирать наиболее важные признаки для модели глубокого обучения (однако как традиционное машинное, так и глубокое обучение для классификации спама по-прежнему требуют присвоения меток, таких как «спам» или «не спам», которые должны быть присвоены либо экспертом, либо пользователями).

1.2. Применение больших языковых моделей

LLM обладают передовыми возможностями анализа и понимания неструктурированных текстовых данных, поэтому имеют широкий спектр применения в различных областях. Модели используются для машинного перевода, создания текстов (рис. 1.2), анализа тональности текста, обобщения текста и многих других задач. Кроме того, в последнее время LLM стали применяться для создания контента (например, статей), написания художественной литературы и даже

генерации компьютерного кода. Интерфейсы LLM позволяют пользователям и системам ИИ общаться на человеческом языке.

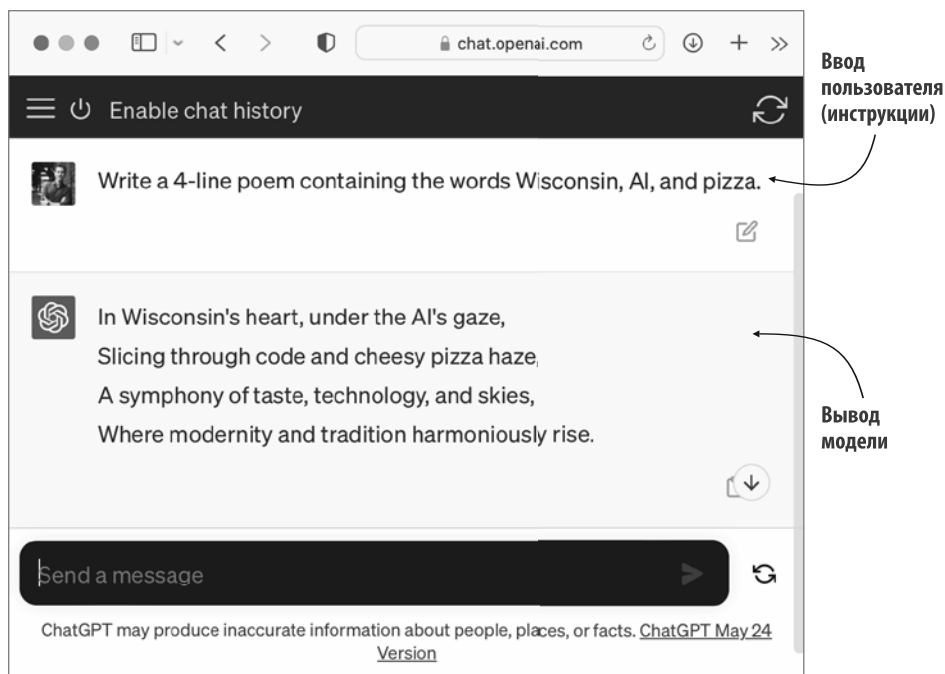


Рис. 1.2. ChatGPT пишет стихотворение по заданию пользователя

Вдобавок LLM являются ядром сложных чат-ботов и виртуальных помощников, таких как ChatGPT от OpenAI или Google Gemini (ранее называвшийся Bard), которые могут отвечать на запросы пользователей и дополнять традиционные поисковые системы, такие как Google Search или Microsoft Bing.

Более того, LLM можно использовать для эффективного извлечения данных из больших объемов текста в специализированных областях, таких как медицина или юриспруденция. Модели помогут отыскать нужные сведения в документах, обобщить длинные отрывки текста и найти ответы на технические вопросы.

Таким образом, LLM незаменимы для автоматизации практически любой задачи, связанной с анализом и генерацией текста. Их возможности практически безграничны, и, по мере того как мы продолжаем внедрять инновации на практике и изучать новые способы использования этих моделей, становится ясно, что у LLM имеется отличный потенциал изменить наше отношение к технологиям, сделав их понятными и более доступными.

1.3. Зачем создавать собственные модели

Написание модели с нуля — отличное упражнение, позволяющее понять ее механику и ограничения. Кроме того, оно дает знания, необходимые для предварительного обучения или тонкой настройки существующих архитектур LLM с открытым исходным кодом, выполняемой пользователем в целях работы с собственными наборами данных или решения задач, связанных с конкретной предметной областью.

ПРИМЕЧАНИЕ Большинство современных LLM реализованы с использованием библиотеки глубокого обучения PyTorch. Мы тоже будем ее применять. Подробное введение в PyTorch можно найти в приложении А.

Исследования показали: пользовательские LLM, созданные для решения конкретных задач или для работы в определенных областях, по производительности могут превосходить универсальные модели, такие как ChatGPT, предназначенные для широкого спектра приложений.

Использование специализированных LLM дает несколько преимуществ, особенно в отношении конфиденциальности данных. Например, компании могут предпочесть не делиться подобными данными со сторонними организациями, такими как OpenAI. Кроме того, небольшие специализированные LLM можно развертывать непосредственно на устройствах клиентов — ноутбуках и смартфонах. Данную возможность в настоящее время изучают такие компании, как Apple.

Локальная реализация может значительно снизить задержки в получении ответа от модели и финансовые расходы, связанные с ее обслуживанием на сервере. Кроме того, специализированные LLM предоставляют разработчикам полную автономность, позволяя контролировать обновления и при необходимости вносить изменения в модель.

Примерами специализированных LLM являются BloombergGPT (предназначенная для финансовой сферы) и LLM, созданные для ответов на вопросы, связанные с медициной. Более подробная информация о подобных моделях представлена в приложении Б.

1.4. Этапы обучения LLM

Процесс обучения модели состоит из двух этапов: предварительного обучения и тонкой настройки (рис. 1.3). Слово «предварительное» в «предварительном обучении» относится к начальному этапу, на котором модель, подобная LLM, обучается на большой и разнообразной выборке данных для широкого понимания языка. Эта предварительно обученная модель затем служит в качестве базового ресурса, который может быть дополнительно усовершенствован с помощью

тонкой настройки. Это процесс, при котором модель обучается на меньшей выборке данных, более специфичной для конкретных задач или областей.



Рис. 1.3. Предварительное обучение LLM включает в себя предсказание следующего слова на больших текстовых выборках данных. Предварительно обученную модель затем можно точно настроить с помощью меньшей по размеру размеченной выборки данных

Разберем эти два этапа более подробно.

Сначала LLM обучается предсказывать следующее слово в тексте, обрабатывая большой массив разнообразных данных, часто именуемых *необработанным* текстом. Это обычный текст без какой-либо информации о метках. (Может применяться фильтрация, например удаление символов форматирования или документов на неизвестных языках.)

ПРИМЕЧАНИЕ Читатели, имеющие опыт в машинном обучении, могут сказать, что информация о разметке данных обычно требуется для традиционных моделей машинного обучения и глубоких нейронных сетей, в которых используется обучение с учителем. Однако для предварительного обучения модели такая разметка не требуется. На данном этапе LLM используют самообучение, при котором модель сама генерирует собственные метки на основе входных данных.

Таким образом, на этапе *предварительного обучения* создается LLM, которую часто называют *базовой*. Типичный пример — GPT-3 (предшественница модели в ChatGPT). Подобная модель способна дополнять текст, то есть завершать наполовину написанное предложение, предоставленное пользователем. Кроме

того, она обладает ограниченными возможностями обучения всего на нескольких примерах вместо больших обучающих данных.

Когда базовая LLM создана, можно переходить ко второму этапу — *тонкой (или точной) настройке модели*. Здесь модель обучается на меньшей по размеру размеченной выборке данных, более характерной для конкретных задач или предметных областей.

Существует две наиболее популярные категории тонкой настройки LLM. При *тонкой настройке по инструкциям* размеченная выборка данных состоит из пар «инструкция — ответ», например фрагмент текста на языке оригинала и его правильный перевод на другой язык. При *тонкой настройке по классификации* размеченная выборка данных состоит из текстов и связанных с ними меток классов, например электронных писем с метками «спам» и «не спам».

Реализацию кода для предварительного обучения LLM, а также специфику тонкой настройки по инструкциям и классификации мы рассмотрим в следующих главах.

1.5. Введение в архитектуру трансформера

Большинство современных LLM основаны на архитектуре *трансформера* — архитектуре глубоких нейронных сетей, предложенной в статье 2017 года *Attention Is All You Need* (<https://arxiv.org/abs/1706.03762>), написанной группой авторов. Стоит учесть, что первый трансформер был разработан для машинного перевода с английского на немецкий и французский языки. Упрощенная версия архитектуры такого трансформера показана на рис. 1.4.

Архитектура трансформера состоит из двух подмодулей: кодировщика и декодировщика. Модуль кодировщика обрабатывает входной текст и кодирует его в виде набора числовых векторов, которые фиксируют контекстную информацию. Затем модуль декодировщика принимает эти векторы и генерирует выходной текст. Например, в задаче перевода кодировщик преобразует текст на исходном языке в векторы, а декодировщик преобразует их в текст на целевом языке.

На рис. 1.4 показан заключительный этап процесса перевода, на котором декодировщику нужно сгенерировать только последнее слово (Beispiel) на основе исходного входного текста (This is an example) и частично переведенного предложения (Das ist ein). (Если у вас возникли вопросы о том, как входные данные предварительно обрабатываются и кодируются, то не волнуйтесь: мы поговорим об этом в последующих главах, когда будем разбирать пошаговую реализацию модели.)

И кодировщик, и декодировщик состоят из множества слоев, соединенных так называемым *механизмом самовнимания* (на рис. 1.4 не показан, будет описан в главе 3). Это ключевой компонент трансформеров и LLM, который позволяет модели оценивать важность различных слов или токенов в последовательности

по отношению друг к другу. Данный механизм позволяет модели улавливать глубокие зависимости и контекстуальные связи во входных данных, улучшая ее способность генерировать связные и правильные выходные данные. Подробным обсуждением и пошаговой реализацией этого механизма мы займемся в главе 3.

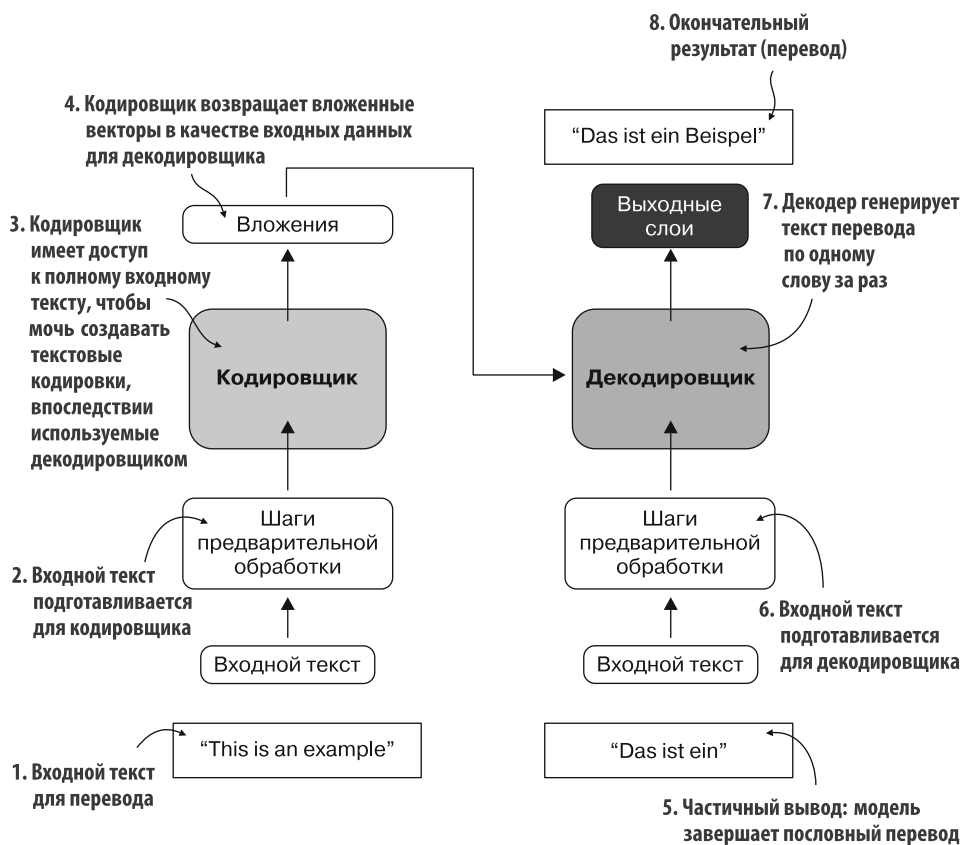


Рис. 1.4. Упрощенное изображение начальной архитектуры трансформера, который представляет собой модель глубокого обучения для языкового перевода

Более поздние варианты архитектуры трансформера, такие как BERT (сокращение от *bidirectional encoder representations from transformers*) и различные модели GPT (сокращение от *generative pretrained transformers*), основаны на концепции трансформеров и адаптированы для решения различных задач. Если вам интересно, то в приложении Б вы найдете информацию о других вариантах трансформеров.

BERT, основанный на подмодуле кодировщика первого трансформера, отличается от GPT подходом к обучению. В то время как GPT предназначен для

генеративных задач, BERT и его варианты специализируются на предсказании слов, скрытых маской (рис. 1.5). Эта уникальная стратегия обучения делает BERT эффективным в задачах классификации текста, таких как прогнозирование тональности текста и категоризация документов. На момент написания книги известно, что X (бывший Twitter) использует BERT для обнаружения вредного контента.

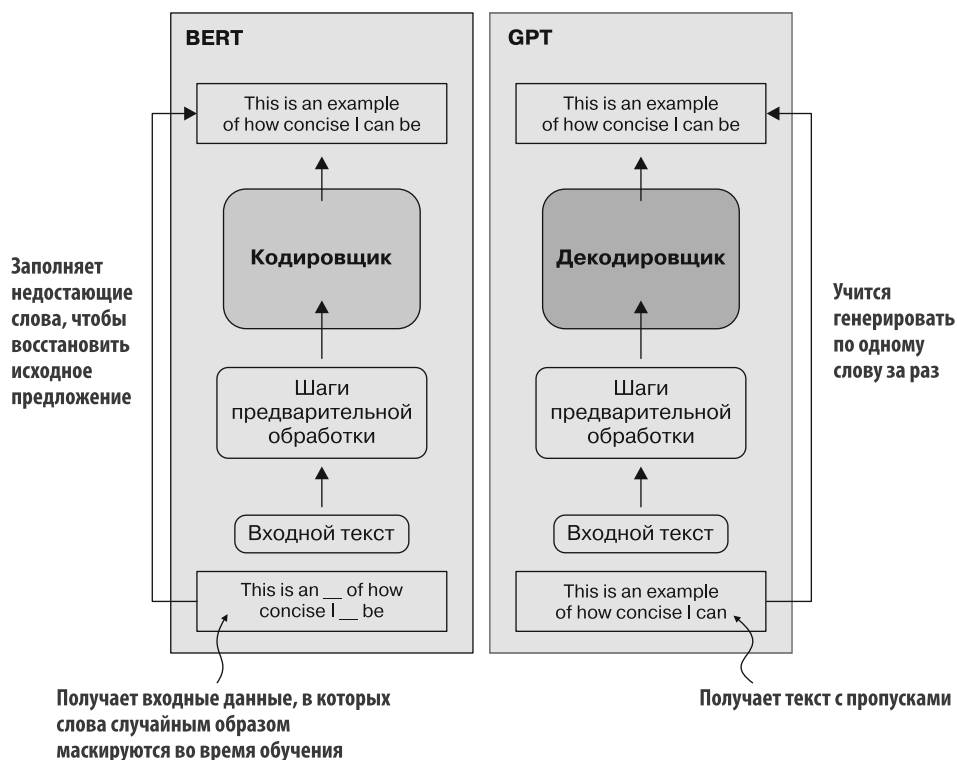


Рис. 1.5. Визуальное представление подмодулей кодировщика и декодировщика для трансформера. Фрагмент кодировщика (*слева*) представляет собой LLM-подобные модели, такие как BERT, которые фокусируются на предсказывании слов с подстановками и в основном используются для таких задач, как классификация текста. Фрагмент декодировщика (*справа*) представляет собой LLM-подобные модели, такие как GPT, предназначенные для генеративных задач и создания связанных текстовых последовательностей

В свою очередь, GPT фокусируется на декодирующей части оригинальной архитектуры трансформера и предназначен для задач, требующих создания текстов, — машинного перевода, обобщения текста, написания художественной литературы, компьютерного кода и многого другого.

Кроме того, модели GPT, разработанные и обученные в первую очередь для выполнения задач по дополнению текста, обладают универсальными возможностями. Эти модели отлично справляются как с обучением без ознакомления, так и с задачами с ограниченным количеством примеров. *Обучение без ознакомления (zero-shot)* позволяет обучить модель справляться с совершенно новыми задачами без каких-либо предварительных конкретных примеров новых данных. В свою очередь, *обучение с ограниченным количеством примеров (few-shot)* предполагает обучение на нескольких примерах, которые пользователь предоставляет в качестве входных данных (рис. 1.6).

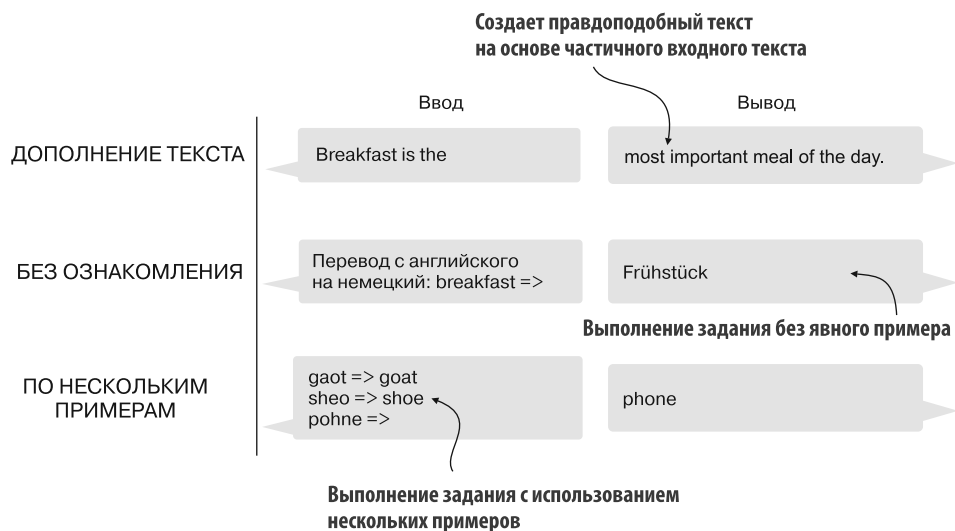


Рис. 1.6. Два варианта обучения LLM: без ознакомления и по нескольким примерам

Трансформеры в сравнении с большими языковыми моделями

Современные LLM основаны на архитектуре трансформеров. Вследствие этого трансформеры и LLM — термины, которые в соответствующей литературе часто используются как синонимы. Однако не все трансформеры являются большими языковыми моделями, поскольку могут использоваться и для компьютерного зрения. Кроме того, не все LLM являются трансформерами, так как существуют большие языковые модели, основанные на рекуррентных и сверточных архитектурах. Основная цель этих альтернативных подходов — повысить вычислительную эффективность LLM. Смогут ли эти альтернативные архитектуры конкурировать с возможностями LLM, основанных на трансформерах, и будут ли они применяться на практике, еще предстоит выяснить. Для простоты я использую термин LLM для обозначения GPT-подобных моделей, основанных на трансформерах (ссылки на литературу, описывающую эти архитектуры, можно найти в приложении Б).

1.6. Использование больших выборок данных

Большие выборки данных для обучения популярных моделей, подобных GPT и BERT, представляют собой разнообразные текстовые корпуса, содержащие миллиарды слов, которые охватывают широкий спектр тем, а также естественных и компьютерных языков. В качестве конкретного примера в табл. 1.1 представлена выборка, использованная для предварительного обучения GPT-3 — базовой модели для первой версии ChatGPT.

Таблица 1.1. Выборка для предварительного обучения популярной модели GPT-3 LLM

Название выборки	Описание данных	Количество токенов в выборке, млрд	Объем в выборке, %
CommonCrawl (отфильтрованный)	Данные веб-сканирования	410	60
WebText2	Данные веб-сканирования	19	22
Books1	Электронные книги из Интернета	12	8
Books2	Электронные книги из Интернета	55	8
Wikipedia	Тексты из Википедии	3	3

В табл. 1.1 указано количество *токенов* — единиц текста, которые считывает модель. Количество токенов в выборке примерно соответствует количеству слов и знаков препинания в тексте. Токенизация — процесс преобразования текста в токены — рассматривается в главе 2.

Масштаб и разнообразие этой обучающей выборки позволяют моделям хорошо справляться с различными задачами, касающимися синтаксиса языка, семантики и контекста, а также с задачами, для решения которых требуются общие знания.

Предварительное обучение LLM требует доступа к значительным объемам вычислительных ресурсов и обходится очень дорого. Например, стоимость предварительного обучения GPT-3 оценивается в 4,6 млн долларов в расценках для облачных вычислений (<https://mng.bz/VxEW>).

Хорошая новость — многие предварительно обученные LLM, доступные как модели с исходным кодом, могут использоваться в качестве инструментов общего назначения для написания, извлечения и редактирования текстов, которые не были частью обучающих данных. Кроме того, LLM могут быть адаптированы для решения конкретных задач с помощью относительно небольших выборок данных, что сокращает необходимые вычислительные ресурсы и повышает производительность.

Подробная информация о выборке данных для GPT-3

В табл. 1.1 представлен состав выборки данных, используемых для обучения GPT-3. Сумма значений в последнем столбце приблизительно равна 100 % с учетом ошибок округления. В третьем столбце указано 499 млрд токенов, однако модель была обучена только на 300 млрд токенов. Авторы статьи о GPT-3 не уточнили, почему модель не была обучена на всех токенах.

Для сравнения: выборка данных CommonCrawl состоит из 410 млрд токенов и требует около 570 Гбайт памяти. В последующих версиях GPT-3, таких как LLaMA от Meta, возможности обучения были расширены за счет добавления дополнительных источников данных (научные статьи на Arxiv (92 Гбайт) и вопросы и ответы о коде на StackExchange (78 Гбайт)).

Авторы статьи о GPT-3 не поделились выборкой данных для обучения, но есть сопоставимый набор данных, находящийся в открытом доступе, — Dolma, созданный Солдайни и др. в 2024 году (<https://arxiv.org/abs/2402.00159>). Обратите внимание: этот набор может содержать материалы, защищенные авторским правом, и условия использования данных могут зависеть от предполагаемого способа применения и страны.

В ходе работы с книгой вы напишете код для предварительного обучения LLM и будете использовать его в образовательных целях для предварительной подготовки модели. Все вычисления будут выполняться на обычных компьютерах. Создав этот код, вы узнаете, как повторно использовать общедоступные весовые коэффициенты модели и загружать их в архитектуру в целях последующей тонкой настройки LLM, что позволит вам пропустить дорогостоящий этап предварительного обучения.

1.7. Более подробный обзор архитектуры GPT

GPT был впервые представлен в статье *Improving Language Understanding by Generative Pre-Training* (<https://mng.bz/x2qg>) Алека Рэдфорда и др. из OpenAI. GPT-3 — улучшенная версия этой модели, которая имеет больше параметров и была обучена на более крупной выборке данных. Кроме того, модель ChatGPT была создана путем тонкой настройки GPT-3 по инструкциям с помощью InstructGPT от OpenAI (<https://arxiv.org/abs/2203.02155>). Как видно на рис. 1.6, GPT являются компетентными моделями дополнения текста. Кроме того, они могут выполнять другие задачи, такие как исправление орфографии, классификация или перевод. Это действительно замечательно, учитывая, что модели предварительно обучались на относительно простой задаче предсказания следующего слова (рис. 1.7).

Задача предсказания следующего слова является формой самообучения, то есть формой самостоятельной разметки данных. Это означает, что нам не нужно собирать метки для обучающих данных, а можно задействовать структуру самих данных: следующее слово в предложении можно использовать в качестве метки,

которую должна предсказывать модель. Эта задача позволяет создавать метки динамически, так что для обучения LLM можно использовать большие массивы текстовых данных без меток.



Рис. 1.7. В задаче предварительного обучения GPT-моделей предсказанию следующего слова система учится предсказывать следующее слово в предложении, глядя на слова, которые были до него. Такой подход помогает модели понять, как слова и фразы обычно сочетаются друг с другом в языке, формируя основу, которую можно применять для решения многих других задач

По сравнению с первоначальной архитектурой трансформера, которую мы рассмотрели в разделе 1.5, общая архитектура GPT относительно проста. По сути, это просто декодировщик без кодировщика (рис. 1.8). Модели в стиле декодировщика, такие как GPT, генерируют текст, предсказывая его по одному слову за раз, поэтому считаются разновидностью *авторегрессионных* моделей. Подобные модели используют предыдущие результаты в качестве входных данных для будущих предсказаний. Следовательно, в GPT каждое новое слово выбирается на основе предшествующей ему последовательности, что повышает связность итогового текста.

Архитектуры, подобные GPT-3, значительно больше по размеру, чем исходная модель-трансформер. Например, в исходном трансформере блоки кодировщика и декодировщика повторялись шесть раз. В свою очередь, GPT-3 содержит 96 слоев трансформера и в общей сложности 175 млрд параметров.

GPT-3 была представлена в 2020 году, что по меркам прогресса глубокого обучения и LLM считается далеким прошлым. Однако более современные архитектуры, такие как модели Llama от Meta, основаны на тех же базовых концепциях, лишь с небольшими изменениями. Таким образом, понимание GPT по-прежнему актуально. Поэтому в последующих главах мы обсудим реализацию известной архитектуры, лежащей в основе GPT, и конкретные изменения, используемые альтернативными LLM.

Оригинальная модель-трансформер была специально разработана для перевода с одного языка на другие, однако модели GPT, несмотря на их больший размер, имеют более простую архитектуру, состоящую только из декодировщика, и тоже способны выполнять задачи по переводу. Эта возможность изначально показала исследователям неожиданной, поскольку появилась в модели, которая в первую очередь обучалась на задаче предсказания следующего слова, а не на задаче перевода.

Способность выполнять задачи, для которых модель не была специально обучена, называется *неожидаемым поведением*. Эта способность не предполагается

в явном виде во время обучения, но возникает как естественное следствие того, что модель работает с огромным количеством многоязычных данных в различных контекстах. Тот факт, что модели GPT могут «изучать» закономерности перевода между языками и выполнять задачи по переводу, даже если не были специально обучены этому, демонстрирует преимущества и возможности таких крупномасштабных генеративных языковых моделей. Различные задачи можно выполнять, не используя для каждой из них отдельную модель.

Определяет следующее слово
на основе входного текста

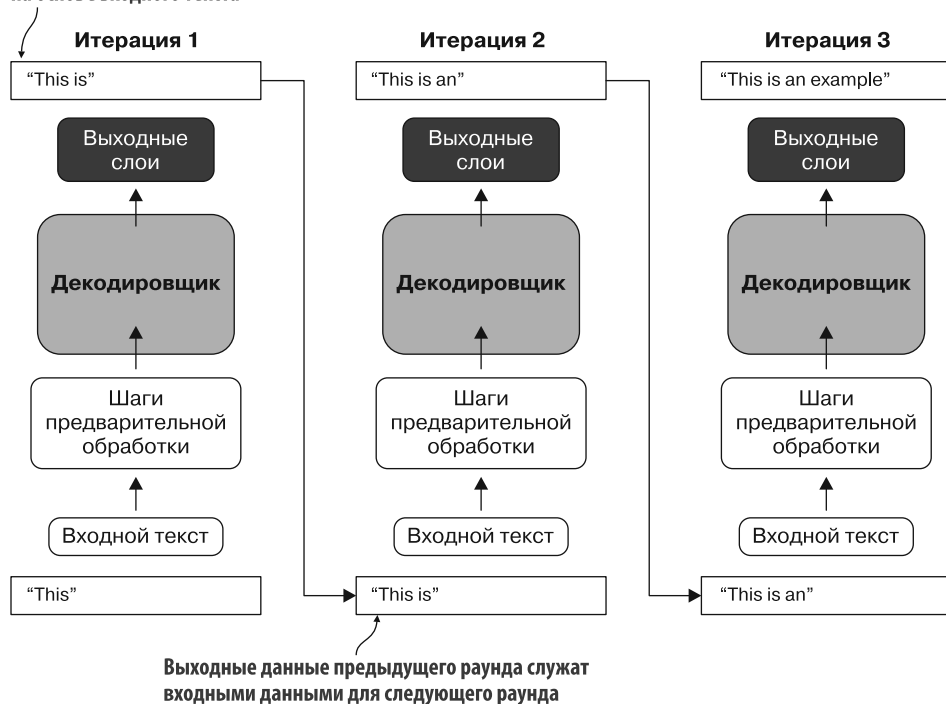


Рис. 1.8. Архитектура GPT использует только декодировщик исходного трансформера. Она предназначена для односторонней обработки последовательности слов слева направо, что делает ее идеально подходящей для генерации текста и задач предсказания следующего слова, позволяя генерировать текст итеративно, по одному слову за раз

1.8. План создания LLM с нуля

Теперь, когда вы получили базовые знания о больших языковых моделях, обсудим план создания одной из них. Мы возьмем за основу фундаментальную идею, лежащую в основе GPT, и реализуем ее в три этапа (рис. 1.9).

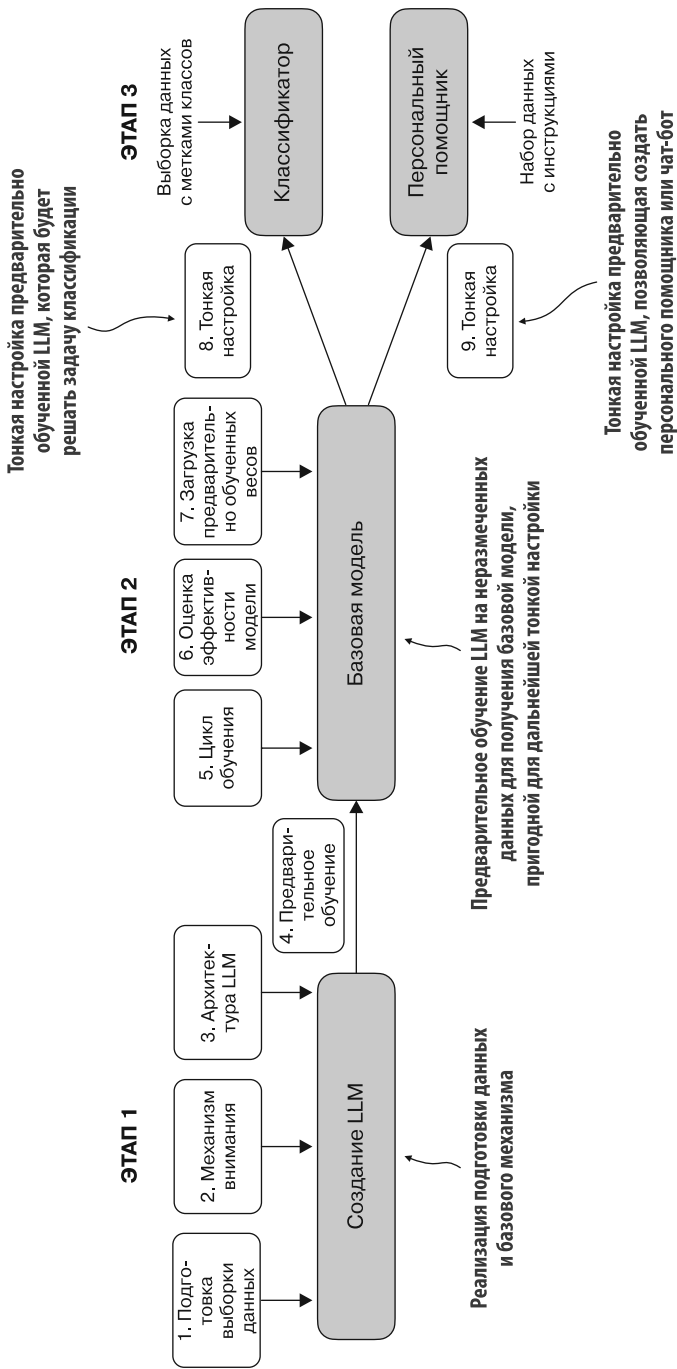


Рис. 1.9. Три этапа создания большой языковой модели

На первом этапе мы обсудим основные шаги предварительной обработки данных, напишем код для механизма внимания, лежащего в основе каждой LLM, и реализуем архитектуру GPT-подобной модели. Всем этим мы займемся в главах 2, 3 и 4.

На втором этапе вы узнаете, как писать код и предварительно обучать LLM, подобную GPT и способную генерировать новые тексты. Кроме того, мы рассмотрим основы оценки эффективности модели, что важно для разработки эффективных систем обработки естественного языка. Данному этапу посвящена глава 5.

Как я уже говорил, предварительное обучение LLM с нуля — серьезная работа, требующая от тысяч до миллионов долларов вычислительных затрат на модели, подобные GPT. Поэтому на втором этапе основное внимание уделяется обучению модели с использованием небольшой выборки данных. Кроме того, я приведу примеры кода для загрузки весов моделей, находящихся в открытом доступе.

Наконец, на третьем этапе мы доучим предварительно обученную LLM, чтобы она выполняла такие операции, как ответы на запросы или классификация текстов — наиболее распространенные задачи во многих реальных приложениях и исследованиях. Описание действий, необходимых для дообучения, представлено в главах 6 и 7.

Итоги главы

- Большие языковые модели изменили сферу обработки естественного языка, которая ранее в основном опиралась на простые системы. Создание LLM породило новые решения, основанные на глубоком обучении, которые привели к возникновению инструментов, способных понимать человеческий язык, генерировать тексты на нем и переводить их.
- Обучение современных LLM состоит из двух основных этапов.
 - Сначала они проходят предварительную подготовку на большом объеме текста без меток, используя в качестве метки предсказание следующего слова в предложении.
 - Затем они настраиваются на меньшей по размеру целевой выборке с разметкой, чтобы следовать инструкциям человека или выполнять задачи классификации.
- LLM основаны на архитектуре трансформера. Ключевой ее компонент — механизм внимания, который предоставляет модели выборочный доступ ко всей входной последовательности для генерации одного слова за раз.
- Первоначальная архитектура трансформера состоит из кодировщика для анализа текста и декодировщика для генерации текста.

- LLM для генерации текста и выполнения инструкций, такие как GPT-3 и ChatGPT, реализуют только модули декодировщика, что упрощает архитектуру.
- Для предварительного обучения LLM необходимы огромные выборки данных, состоящие из миллиардов слов.
- В то время как основная задача предварительного обучения моделей, подобных GPT, заключается в предсказании следующего слова в предложении, эти LLM обладают новыми свойствами, такими как способность классифицировать, переводить или обобщать тексты.
- После предварительного обучения полученную базовую модель можно более эффективно доучить, чтобы она смогла решать другие задачи.
- LLM, доученные на специализированных выборках, по части решения конкретных задач могут превосходить универсальные модели.

Работа с текстовыми данными

В этой главе

- ✓ Подготовка текста для обучения большой языковой модели.
- ✓ Разделение текста на слова и более точные токены.
- ✓ Кодирование пар байтов как более продвинутый способ токенизации текста.
- ✓ Выбор обучающих примеров с помощью скользящего окна.
- ✓ Преобразование токенов в векторы, которые служат входом для большой языковой модели.

В предыдущей главе мы рассмотрели общую структуру больших языковых моделей и узнали, что они предварительно обучаются на огромных объемах текста. В частности, мы сосредоточились на LLM, основанных только на декодировщике и архитектуре трансформера, на которой строятся модели, используемые в ChatGPT и других популярных LLM, подобных GPT.

Как вы уже знаете, на этапе предварительного обучения LLM обрабатывают текст по одному слову за раз. Обучение LLM с миллионами и миллиардами параметров задаче предсказания следующего слова позволяет создавать модели с впечатляющими возможностями. Затем эти модели можно дополнительно доучить, чтобы они следовали общим инструкциям или выполняли конкретные задачи. Но прежде чем создавать и обучать LLM, нужно подготовить набор данных для обучения (рис. 2.1).

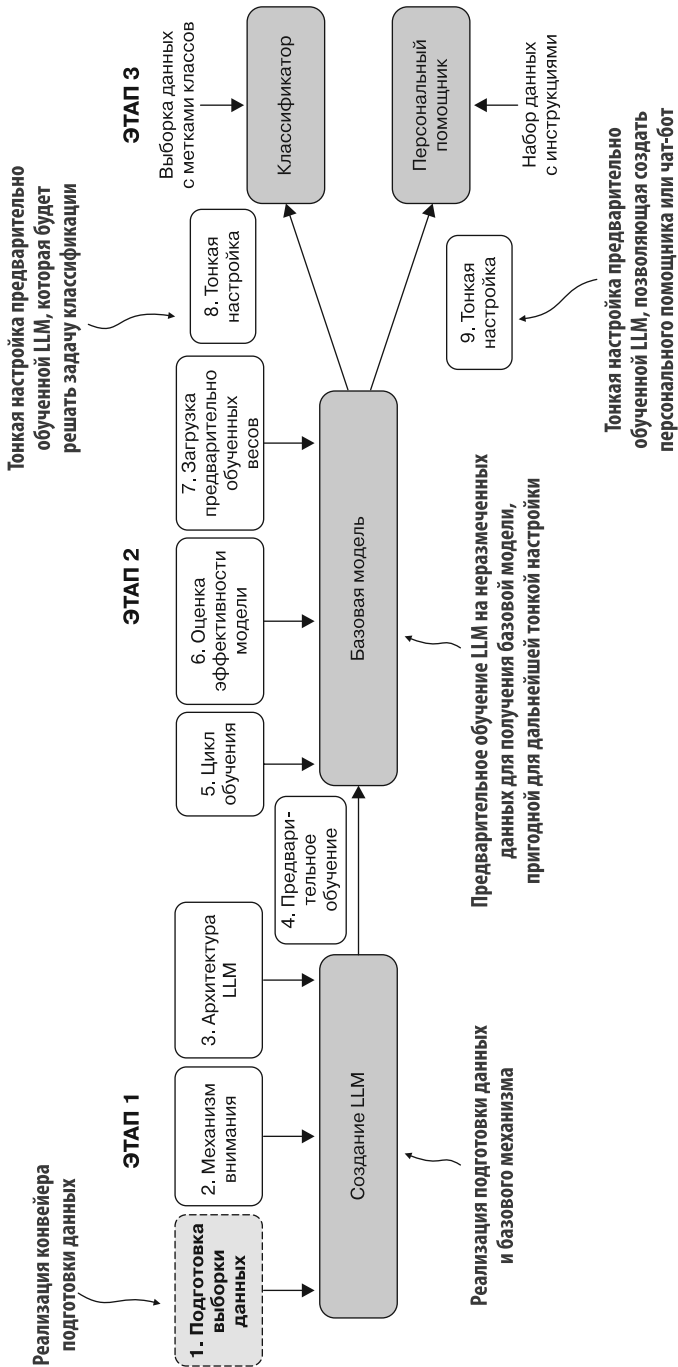


Рис. 2.1. Три этапа разработки LLM. В этой главе основное внимание уделяется первому шагу этапа 1: реализации конвейера подготовки текста

В данной главе вы узнаете, как подготовить текстовые данные, которые будут использоваться для обучения LLM. Процесс подготовки включает в себя разделение текста на слова и более мелкие структуры (так называемая токенизация), которые затем могут быть представлены в виде векторов для LLM. Кроме того, вы узнаете о более сложных схемах токенизации, таких как кодирование пар байтов, которое используется в популярных моделях наподобие GPT. Наконец, мы реализуем стратегию выборки и загрузки данных для создания пар «вход — выход», необходимых для обучения LLM.

2.1. Векторные представления слов

Модели глубоких нейронных сетей, в том числе LLM, не могут напрямую обрабатывать исходный текст, поскольку его формат несовместим с математическими операциями, используемыми для реализации и обучения нейронных сетей. Поэтому нужен способ, позволяющий представлять слова в виде векторов с числовыми значениями.

ПРИМЕЧАНИЕ Читатели, незнакомые с векторами и тензорами в контексте вычислений, могут узнать больше информации об этом в подразделе A.2.2 приложения A.

Концепция преобразования данных в векторный формат часто называется *вложением* (*эмбедингом*). Используя определенный слой нейронной сети или другую предварительно обученную модель такой сети, мы можем осуществлять вложение различных типов данных, например видео, аудио и текста (рис. 2.2). Однако важно отметить, что для разных форматов данных требуются разные модели вложения. Например, модель вложения текста не подойдет для вложения аудио- или видеоданных.

Модели глубокого обучения не могут обрабатывать такие форматы данных, как видео, аудио и текст, в их исходном виде. Поэтому модели вложения используются для того, чтобы преобразовать эти исходные данные в векторное представление, которое архитектуры глубокого обучения могут легко понять и обработать.

По сути, вложение представляет собой преобразование дискретных объектов, таких как слова, изображения или даже целые документы, в точки в непрерывном векторном пространстве. Основная цель вложения — преобразование нечисловых данных в формат, который могут обрабатывать нейронные сети.

В то время как вложение слов является наиболее распространенной формой вложения текста, существует также вложение предложений, абзацев или целых

документов. Вложение предложений или параграфов — популярное решение для создания *дополненной поиском генерации* (*retrieval-augmented generation, RAG*). Она сочетает в себе генерацию (например, создание текста) с поиском (например, во внешней базе знаний) для получения нужной информации, но этот подход к генерации текста выходит за рамки данной книги. Наша цель — обучение LLM, подобных GPT, которые учатся генерировать текст по одному слову за раз, поэтому мы сосредоточимся на вложении слов.

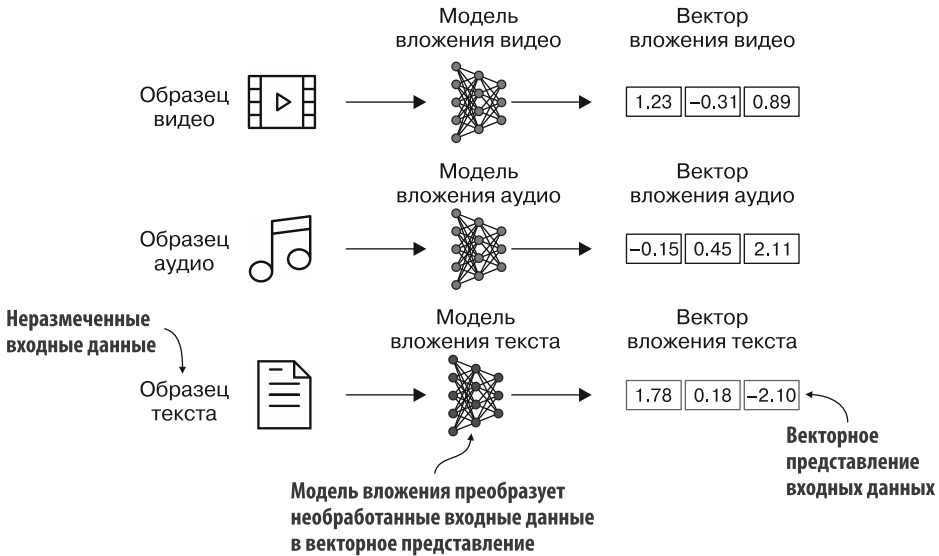


Рис. 2.2. Процесс преобразования исходных данных в трехмерный числовой вектор

Для получения вложения слов было разработано несколько алгоритмов. Один из ранних и наиболее популярных примеров — *Word2Vec*. Этот алгоритм обучает архитектуру нейронной сети генерировать векторные представления слов, предсказывая контекст слова по целевому слову или наоборот. Основная идея *Word2Vec* заключается в том, что слова, которые появляются в схожих контекстах, как правило, имеют схожие значения. Следовательно, при проецировании на двумерные векторные представления слов в целях визуализации похожие термины группируются (рис. 2.3).

Векторные представления слов могут иметь разную длину или размерность, от одной до нескольких тысяч. Более высокая размерность может вызвать более специфичные взаимосвязи, но за счет снижения вычислительной эффективности.

Можно использовать предварительно обученные модели, такие как Word2Vec, для создания векторных представлений, однако LLM обычно создают собственные представления, которые являются частью входного слоя и обновляются во время обучения. Векторные представления, оптимизированные в рамках обучения LLM, имеют преимущество по сравнению с использованием Word2Vec: они изменены под конкретную задачу и имеющиеся данные. Мы реализуем в коде такие векторные представления позже в этой главе. (Кроме того, LLM могут создавать векторные представления, привязанные к контексту, которые мы обсудим в главе 3.)

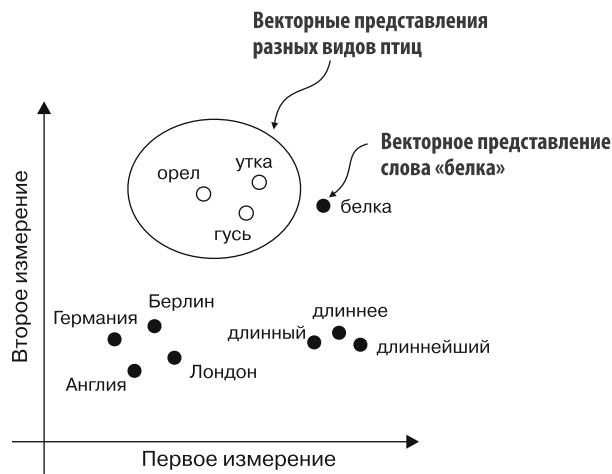


Рис. 2.3. Если векторные представления слов двумерные, то мы можем визуализировать эти представления. При использовании методов вложения слов, таких как Word2Vec, слова, соответствующие схожим понятиям, часто оказываются близко друг к другу во вложенном пространстве. Например, в таком пространстве разные виды птиц оказываются ближе друг к другу, чем страны и города

К сожалению, многомерные векторные представления затрудняют визуализацию, поскольку человеческое сенсорное восприятие и распространенные графические представления, по сути, ограничены двумя-тремя измерениями, поэтому на рис. 2.3 показаны двумерные векторные представления. Однако при работе с LLM обычно используются векторные представления, обладающие гораздо большей размерностью. Для GPT-2 и GPT-3 длина вектора вложения (часто называемая размерностью скрытых состояний модели) варьируется в зависимости от варианта и размера модели. Это компромисс между производительностью и эффективностью. В самых маленьких моделях GPT-2 (117 и 125 млн параметров) длина вектора вложения составляет 768 элементов. В самой большой модели GPT-3 (175 млрд параметров) длина такого вектора составляет 12 288 элементов.

Далее мы рассмотрим действия, необходимые для подготовки результатов вложения, используемых LLM: разделение текста на слова, преобразование слов в токены и преобразование токенов в векторы вложения.

2.2. Токенизация текста

Разделение на токены — этап предварительной обработки текста, необходимый для создания векторных представлений для LLM. Эти токены представляют собой либо отдельные слова, либо специальные символы, в том числе знаки препинания (рис. 2.4).

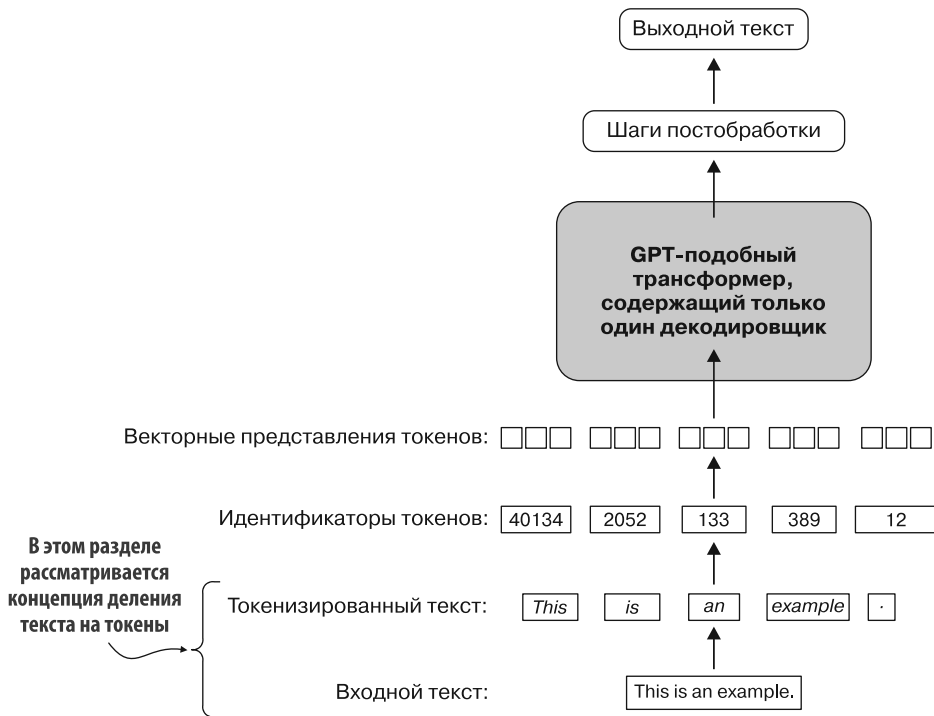


Рис. 2.4. Шаги обработки текста в контексте LLM. Вводимый текст разбивается на отдельные токены, которые представляют собой либо слова, либо специальные символы, например знаки препинания

Текст, который мы будем токенизировать в целях обучения LLM, — это рассказ *The Verdict* Эдит Уортон, который находится в открытом доступе и, таким образом, может быть использован для обучения модели. Текст можно найти на Викискладе по адресу https://en.wikisource.org/wiki/The_Verdict, и вы можете скопировать и вставить его в текстовый файл (я скопировал его в файл `the-verdict.txt`).

Кроме того, вы можете найти этот файл в репозитории GitHub этой книги по адресу <https://mng.bz/Adng>. Скачать файл можно с помощью следующего кода Python:

```
import urllib.request
url = ("https://raw.githubusercontent.com/rasbt/"
      "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
      "the-verdict.txt")
file_path = "the-verdict.txt"
urllib.request.urlretrieve(url, file_path)
```

Далее можно загрузить файл `the-verdict.txt` с помощью стандартных средств Python для чтения файлов (листинг 2.1).

Листинг 2.1. Чтение файла с коротким рассказом на Python

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
print("Total number of character:", len(raw_text))
print(raw_text[:99])
```

Оператор `print` выводит общее количество символов, а затем первые 100 символов этого файла для наглядности:

```
Total number of character: 20479
I HAD always thought Jack Gisburn rather a cheap genius--though a good fellow
  ➤ enough--so it was no
```

Наша цель — разделить этот рассказ из 20 479 символов на отдельные слова и специальные символы, которые затем можно будет преобразовать во вложенные векторы.

ПРИМЕЧАНИЕ При работе с LLM обычно обрабатываются миллионы статей и сотни тысяч книг — многие гигабайты текста. Однако в образовательных целях достаточно работать с небольшими текстовыми фрагментами, например с одной книгой, чтобы увидеть основные идеи, лежащие в основе шагов обработки текста, и чтобы можно было запустить код на обычном ноутбуке.

Как лучше всего разделить текст, чтобы получить список токенов? Для этого мы воспользуемся библиотекой регулярных выражений `re` в демонстрационных целях (вам не нужно изучать или запоминать синтаксис регулярных выражений, поскольку позже мы перейдем к встроенному токенизатору, который сделает всю работу за вас).

Используя простой пример, мы можем разделить текст по пробелам с помощью команды `re.split`:

```
import re
text = "Hello, world. This, is a test."
result = re.split(r'(\s)', text)
print(result)
```

Результатом является список отдельных слов, пробелов и знаков препинания:

```
['Hello', ' ', ' ', 'world.', ' ', ' ', 'This', ' ', ' ', 'is', ' ', ' ', 'a', ' ', ' ', 'test.']
```

Эта простая схема токенизации в основном подходит для деления текста на отдельные слова; однако некоторые слова по-прежнему связаны с символами пунктуации, которые мы хотим видеть в виде отдельных записей в списке. Кроме того, мы не станем менять регистр всех букв на строчный, поскольку заглавные буквы помогают модели различать имена собственные и нарицательные, понимать структуру предложений и учиться генерировать текст с правильными заглавными буквами.

Изменим регулярное выражение, чтобы оно разделяло текст по пробелам (\s), запятым и точкам ([, .]):

```
result = re.split(r'([,.]|\s)', text)
print(result)
```

Мы видим, что слова и знаки препинания теперь являются отдельными элементами списка, как мы и хотели:

```
['Hello', ',', ' ', ' ', ' ', 'world', '.', ' ', ' ', ' ', 'This', ',', ' ', ' ', ' ', 'is', ' ', ' ', 'a', ' ', ' ', 'test', '.', ' ']
```

Небольшая оставшаяся проблема заключается в том, что в списке по-прежнему есть пробелы. При необходимости мы можем безопасно удалить эти лишние символы следующим образом:

```
result = [item for item in result if item.strip()]
print(result)
```

Результат выглядит так:

```
['Hello', ',', 'world', '.', 'This', ',', 'is', 'a', 'test', '.']
```

ПРИМЕЧАНИЕ При разработке простого токенизатора вопрос о том, что делать с пробелами (кодировать их как отдельные символы или просто удалять), зависит от приложения и его требований. Удаление пробелов снижает требования к памяти и вычислениям. Однако сохранение пробелов может быть полезным, если вы обучаете модели, чувствительные к структуре текста (например, кода на Python, чувствительного к отступам и пробелам). Здесь мы удаляем пробелы для простоты и краткости результатов. Позже мы перейдем к схеме токенизации, которая учитывает пробелы.

Схема токенизации, которую мы разработали, хорошо работает с простым текстом. Изменим ее, чтобы она могла обрабатывать и другие знаки препинания, такие как вопросительные знаки, кавычки и двойные тире, которые мы

видели в первых 100 символах рассказа Эдит Уортон, а также дополнительные специальные символы:

```
text = "Hello, world. Is this-- a test?"
result = re.split(r'([,.;?_!"()\`]|--|\s)', text)
result = [item.strip() for item in result if item.strip()]
print(result)
```

Получаем:

```
['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?']
```

Как мы видим по результатам, представленным на рис. 2.5, наша схема токенизации теперь может успешно обрабатывать различные специальные символы в тексте.

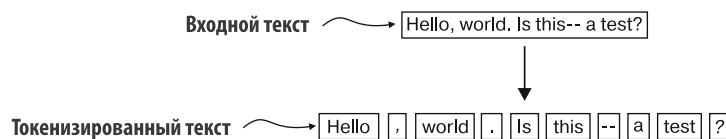


Рис. 2.5. Схема токенизации разделяет текст на отдельные слова и знаки препинания. В этом конкретном примере текст разбивается на десять отдельных токенов

Теперь, когда у нас есть базовый токенизатор, применим его ко всему рассказу Эдит Уортон:

```
preprocessed = re.split(r'([,.;?_!"()\`]|--|\s)', raw_text)
preprocessed = [item.strip() for item in preprocessed if item.strip()]
print(len(preprocessed))
```

Этот оператор `print` выводит **4690** — общее число токенов в данном тексте (без пробелов). Выведем первые 30 токенов, чтобы выполнить быструю визуальную проверку:

```
print(preprocessed[:30])
```

Полученный результат показывает, что токенизатор, по-видимому, хорошо обрабатывает текст, поскольку все слова и специальные символы аккуратно разделены:

```
['I', 'HAD', 'always', 'thought', 'Jack', 'Gisburn', 'rather', 'a',
'cheap', 'genius', '--', 'though', 'a', 'good', 'fellow', 'enough',
--, 'so', 'it', 'was', 'no', 'great', 'surprise', 'to', 'me', 'to',
'hear', 'that', ',', 'in']
```

2.3. Преобразование токенов в идентификаторы токенов

Далее мы преобразуем полученные токены из строки Python в целочисленное представление, чтобы получить идентификаторы токенов. Это преобразование — промежуточный этап перед преобразованием идентификаторов токенов в векторы вложения.

Чтобы преобразовать ранее сгенерированные токены в идентификаторы, сначала нужно создать словарь (рис. 2.6). Он определяет, как мы сопоставляем каждое уникальное слово или специальный символ с уникальным целым числом.

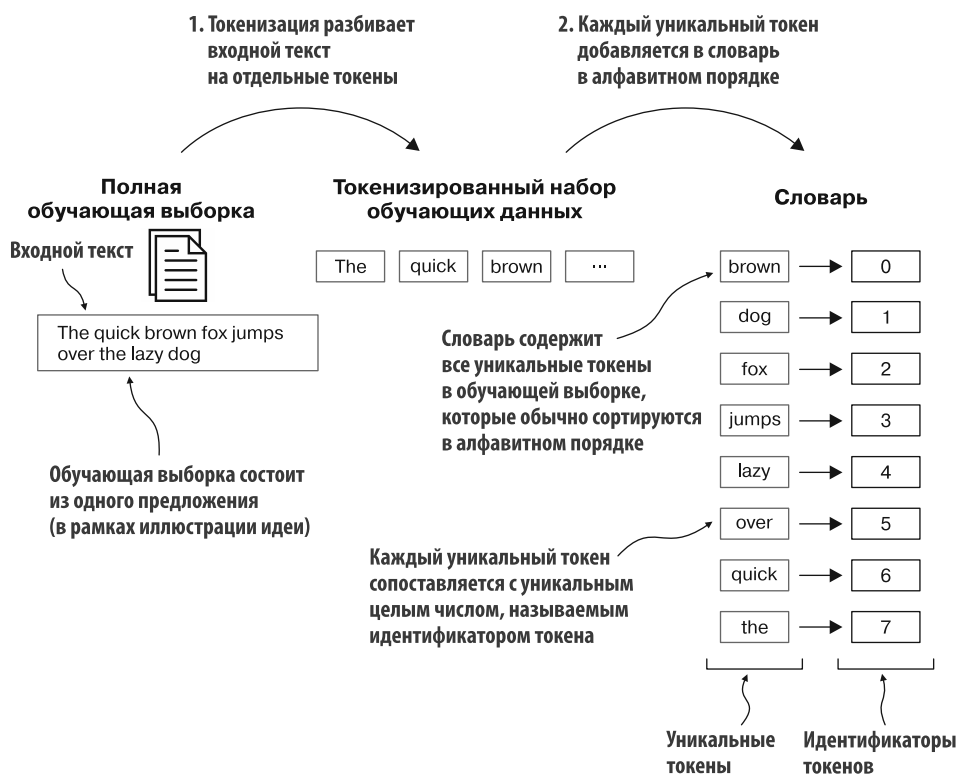


Рис. 2.6. Создание словаря

Итак, весь текст в обучающей выборке данных разбивается на отдельные токены. Затем они сортируются в алфавитном порядке, а повторяющиеся удаляются. Далее уникальные токены объединяются в словарь, который определяет

соответствие между каждым таким токеном и уникальным целым числом. Словарь на рис. 2.6 намеренно небольшой и в целях упрощения демонстрации идеи не содержит знаков препинания или специальных символов.

Теперь, когда мы разбили рассказ Эдит Уортон на токены и присвоили их переменной Python `preprocessed`, создадим список всех уникальных токенов и отсортируем их в алфавитном порядке, чтобы определить размер словаря:

```
all_words = sorted(set(preprocessed))
vocab_size = len(all_words)
print(vocab_size)
```

Определив с помощью этого кода, что размер словаря составляет 1130 слов, мы создаем словарь и выводим на экран первый 51 элемент для наглядности (листинг 2.2).

Листинг 2.2. Создание словаря

```
vocab = {token:integer for integer,token in enumerate(all_words)}
for i, item in enumerate(vocab.items()):
    print(item)
    if i >= 50:
        break
```

В результате получаем:

```
('!', 0)
('\"', 1)
('\"', 2)
...
('Her', 49)
('Hermia', 50)
```

Как мы видим, словарь содержит отдельные токены, связанные с уникальными целочисленными метками. Наша следующая цель — применить его для преобразования нового текста в идентификаторы токенов (рис. 2.7).

Итак, начиная с нового примера текста, мы разбиваем текст на токены и используем словарь для преобразования текстовых токенов в идентификаторы токенов. Словарь создается на основе всей обучающей выборки и может применяться как к ней самой, так и к любым новым примерам текста. Для простоты в словаре-образце нет знаков препинания или специальных символов.

Чтобы преобразовать выходные данные LLM из чисел обратно в текст, нужно преобразовать идентификаторы токенов в текст. Для этого можно создать обратную версию словаря, которая сопоставляет идентификаторы токенов с соответствующими текстовыми токенами.

Реализуем полноценный класс токенизатора на Python с методом `encode`, который разбивает текст на токены и преобразует строки в целые числа, чтобы

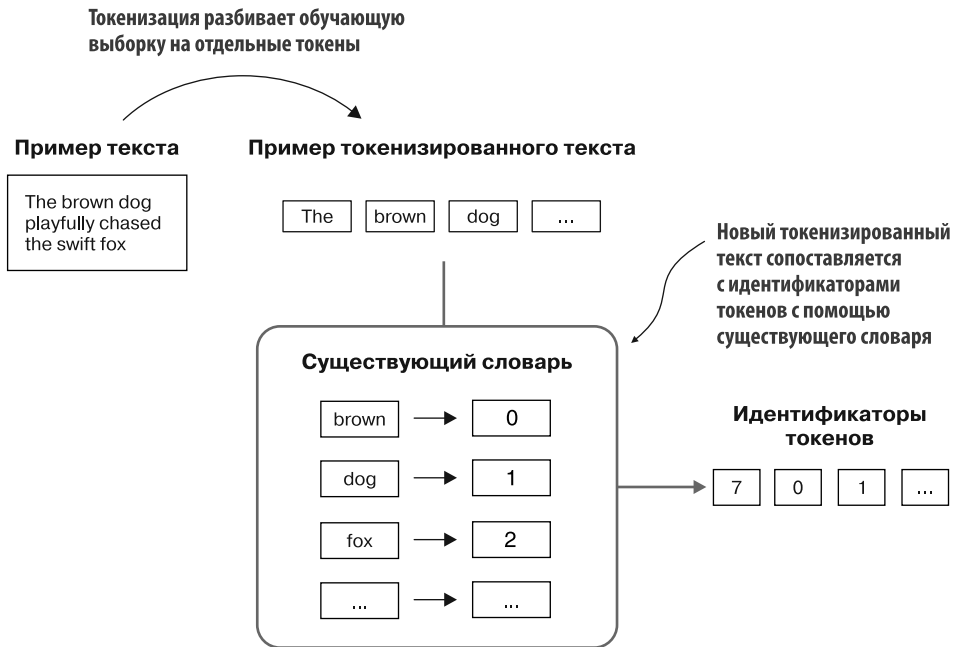


Рис. 2.7. Преобразование нового текста в идентификаторы токенов с помощью словаря

получить идентификаторы токенов из словаря. Кроме того, мы реализуем метод `decode`, который преобразует целые числа обратно в строки, чтобы преобразовать идентификаторы токенов в текст. В листинге 2.3 приведен код для токенизатора.

Листинг 2.3. Реализация простого токенизатора

```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([.?!"()\'']|--|\s)', text)
        preprocessed = [
            item.strip() for item in preprocessed if item.strip()
        ]
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])

        text = re.sub(r'\s+([.?!"()\''])', r'\1', text)
```

Хранит словарь как атрибут класса для доступа к методам `encode` и `decode`

Создает обратный словарь, проецирующий идентификаторы обратно в токены

Преобразует текст в идентификаторы токенов

Трансформирует идентификаторы обратно в текст

Удаляет пробелы перед определенным знаком пунктуации

Используя класс `SimpleTokenizerV1` в Python, мы можем создавать новые объекты токенизатора на основе существующего словаря, который затем можно использовать для кодирования и декодирования текста (рис. 2.8).

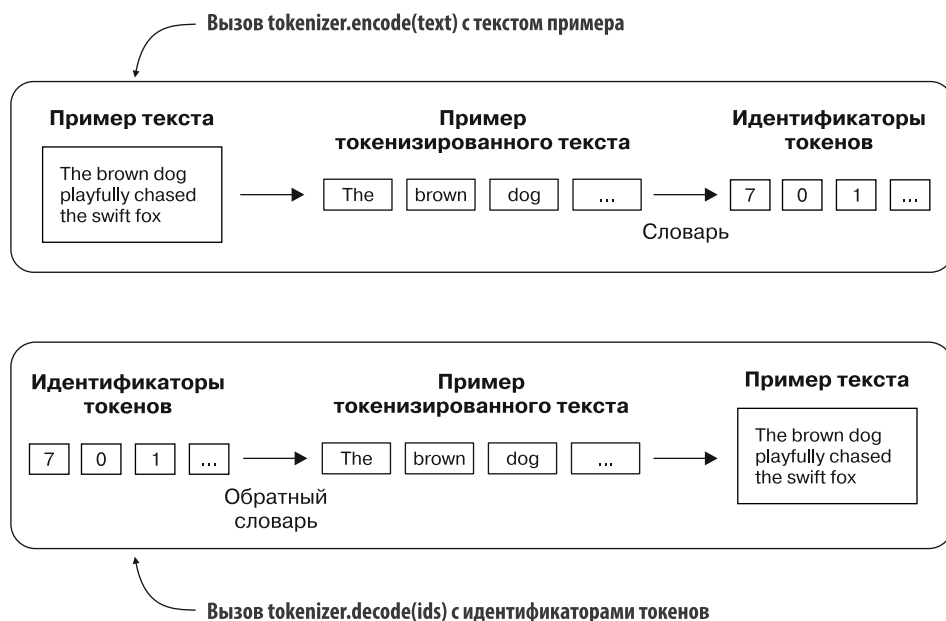


Рис. 2.8. В реализацию токенизатора добавлены методы `encode` и `decode`. Метод `encode` принимает текст, разбивает его на отдельные токены и преобразует их в идентификаторы токенов с помощью словаря. Метод `decode` принимает идентификаторы токенов, преобразует их обратно в текстовые токены и объединяет текстовые токены в связный текст

Создадим новый объект токенизатора на основе класса `SimpleTokenizerV1` и токенизируем отрывок из рассказа Эдит Уортон, чтобы опробовать его на практике:

```
tokenizer = SimpleTokenizerV1(vocab)
text = """It's the last he painted, you know,"
Mrs. Gisburn said with pardonable pride."""
ids = tokenizer.encode(text)
print(ids)
```

В результате выполнения этого кода мы получаем следующие идентификаторы:

```
[1, 56, 2, 850, 988, 602, 533, 746, 5, 1126, 596, 5, 1, 67, 7, 38, 851, 1108,
754, 793, 7]
```


Теперь посмотрим, сможем ли мы преобразовать эти идентификаторы токенов обратно в текст с помощью метода `decode`:

```
print(tokenizer.decode(ids))
```

Результат выглядит так:

```
'" It\' s the last he painted, you know," Mrs. Gisburn said with pardonable pride.'
```

Видно, что метод `decode` успешно преобразовал идентификаторы токенов обратно в исходный текст.

Пока все хорошо. Мы реализовали токенизатор, способный токенизировать и детокенизировать текст на основе фрагмента из обучающей выборки. Теперь применим этот токенизатор к новому примеру текста, которого нет в обучающей выборке:

```
text = "Hello, do you like tea?"
print(tokenizer.encode(text))
```

При выполнении этого кода появляется следующая ошибка:

```
KeyError: 'Hello'
```

Проблема в том, что слова `Hello` нет в рассказе *The Verdict*. Следовательно, оно не содержится в словаре. Именно чтобы избежать подобных проблем, для обучения LLM необходимо использовать большие и разнообразные выборки данных, чтобы расширить словарный запас модели.

Далее мы протестируем токенизатор на тексте, содержащем неизвестные слова, и обсудим специальные токены, которые можно использовать для предоставления LLM дополнительного контекста во время обучения.

2.4. Добавление специальных контекстных токенов

Итак, нам нужно изменить токенизатор, чтобы он обрабатывал неизвестные слова. Кроме того, стоит рассмотреть использование и добавление специальных контекстных токенов, которые могут улучшить понимание моделью контекста или другой важной информации в тексте. Эти специальные токены могут содержать, например, маркеры неизвестных слов и границ документов. В частности, мы изменим словарь и токенизатор `SimpleTokenizerV2`, чтобы они поддерживали два новых токена: `<|unk|>` и `<|endoftext|>` (рис. 2.9).

Токен `<|unk|>` служит для обозначения новых и неизвестных слов, которые не были частью обучающих данных и, следовательно, не входили в существующий словарь. Токен `<|endoftext|>` можно использовать для разделения двух несвязанных источников текста.

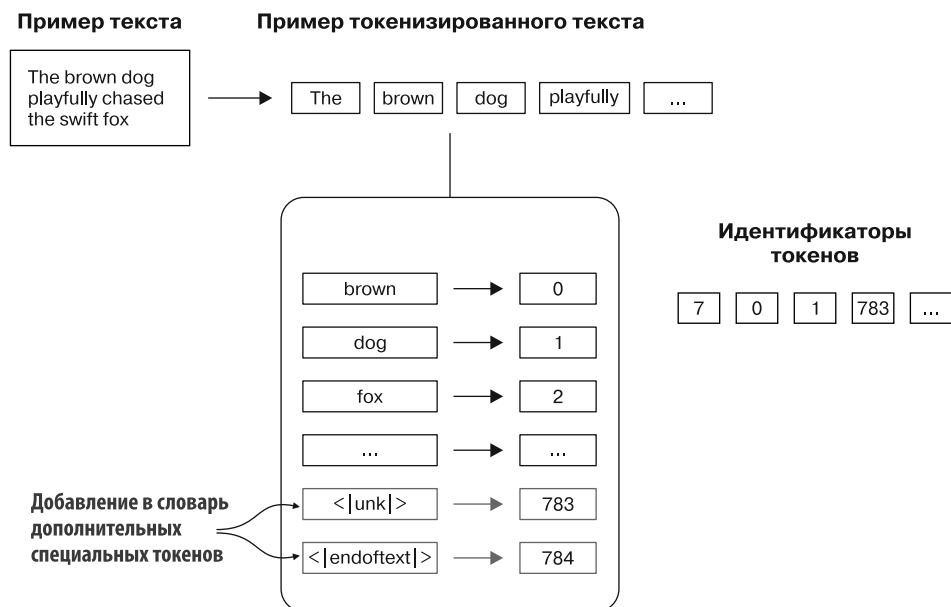


Рис. 2.9. Добавление в словарь специальных токенов для обработки особых случаев

Например, при обучении LLM, подобных GPT, на нескольких независимых документах или книгах обычно вставляется токен перед каждым документом или книгой, которые следуют за предыдущим источником текста (рис. 2.10). Это помогает модели понять, что, хоть эти текстовые источники и объединены для обучения, на самом деле они не связаны между собой.

Теперь изменим словарь, поместив в него эти два специальных токена, `<|unk|>` и `<|endoftext|>`, добавив их в наш список всех уникальных слов:

```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token:integer for integer,token in enumerate(all_tokens)}

print(len(vocab.items()))
```

Судя по выводу этого оператора `print`, размер нового словаря составляет 1132 слова (предыдущий размер — 1130 слов).

В качестве дополнительной быстрой проверки выведем последние пять записей обновленного словаря:

```
for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

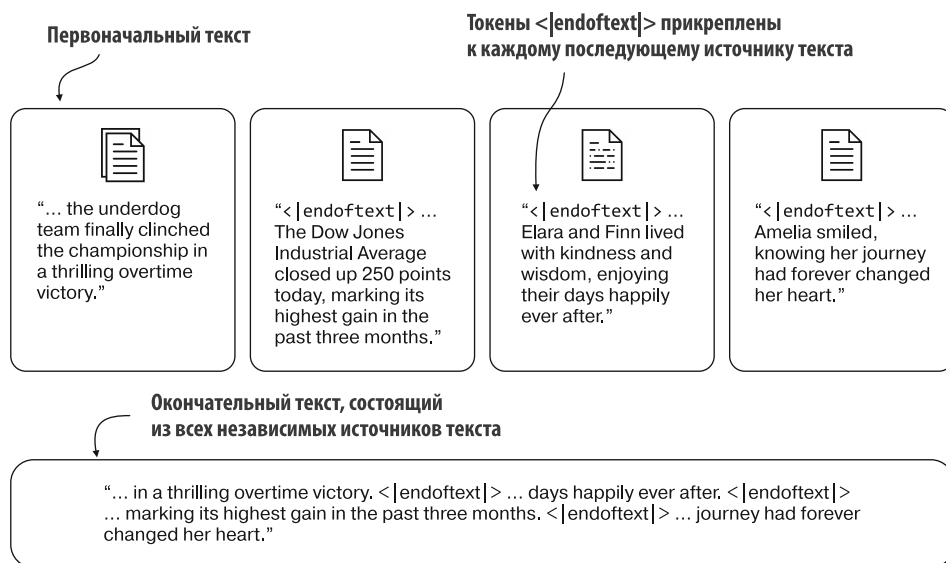


Рис. 2.10. Токены <|endoftext|>, добавленные между источниками, действуют как маркеры, обозначающие начало или конец определенного сегмента, что позволяет LLM более эффективно обрабатывать и понимать текст

Результат выглядит так:

```
('younger', 1127)
('your', 1128)
('yourself', 1129)
('<|endoftext|>', 1130)
('<|unk|>', 1131)
```

Основываясь на данном результате, мы можем подтвердить, что два новых специальных токена действительно были успешно добавлены в словарь. Далее мы соответствующим образом скорректируем токенизатор из листинга 2.3 (листинг 2.4).

Листинг 2.4. Простой токенизатор, который знает, как обрабатывать неизвестные слова

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = { i:s for s,i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([.,:;?_!"()\\']|--|\\s)', text)
        preprocessed = [
```

56 Глава 2. Работа с текстовыми данными

```
item.strip() for item in preprocessed if item.strip()
]
preprocessed = [item if item in self.str_to_int
                 else "<|unk|>" for item in preprocessed]

ids = [self.str_to_int[s] for s in preprocessed]
return ids

def decode(self, ids):
    text = " ".join([self.int_to_str[i] for i in ids])

    text = re.sub(r'\s+([.,:;!\"()\\'\']|)', r'\1', text)
    return text
```

Заменяет неизвестные слова токенами <|unk|>

Заменяет пробелы в тексте перед знаками пунктуации

По сравнению с `SimpleTokenizerV1`, который мы реализовали в листинге 2.3, новый `SimpleTokenizerV2` заменяет неизвестные слова на токены <|unk|>.

Теперь опробуем этот новый токенизатор на практике. Мы будем использовать простой пример текста, который объединим из двух независимых и не связанных между собой предложений:

```
text1 = "Hello, do you like tea?"
text2 = "In the sunlit terraces of the palace."
text = " ".join((text1, text2))
print(text)
```

Результат выглядит так:

```
Hello, do you like tea? In the sunlit terraces of the palace.
```

Теперь мы разберем на токены текст примера, используя `SimpleTokenizerV2` и словарь, который создали ранее в листинге 2.2:

```
tokenizer = SimpleTokenizerV2(vocab)
print(tokenizer.encode(text))
```

Код выводит такую последовательность:

```
[1131, 5, 355, 1126, 628, 975, 10, 1130, 55, 988, 956, 984, 722, 988, 1131, 7]
```

Мы видим, что список идентификаторов токенов содержит 1130 для токена <|endoftext|>, а также два токена 1131, которые используются для неизвестных слов.

Применим детокенизацию сразу же после токенизации для быстрой проверки:

```
print(tokenizer.decode(tokenizer.encode(text)))
```

Результат выглядит так:

```
<|unk|>, do you like tea? <|endoftext|> In the sunlit terraces of the <|unk|>.
```

Сравнивая этот результат с исходным текстом, мы видим, что в обучающей выборке нет слов `Hello` и `palace`.

В зависимости от LLM некоторые исследователи также используют дополнительные специальные токены:

- `[BOS]` (beginning of sequence — «начало последовательности») — этот токен обозначает начало текста. Он указывает LLM, где начинается фрагмент нового текста;
- `[EOS]` (end of sequence — «конец последовательности») — токен находится в конце текста и особенно полезен при объединении нескольких несвязанных текстов, аналогично `<|endoftext|>`. Например, при объединении двух разных статей из Википедии токен `[EOS]` указывает, где заканчивается одна статья и начинается другая;
- `[PAD]` (padding — «заполнение») — при обучении LLM с использованием нескольких пакетов каждый из этих пакетов может содержать тексты разной длины. Чтобы гарантировать, что все тексты имеют одинаковую длину, более короткие тексты расширяют или «дополняют» с помощью маркера `[PAD]` до длины самого длинного текста в пакете.

Токенизатору, используемому для моделей GPT, не нужен ни один из этих токенов; для простоты он использует только `<|endoftext|>`. Данный токен аналогичен токenu `[EOS]`. Вдобавок `<|endoftext|>` используется для заполнения. Однако, как мы увидим в следующих главах, при обучении на данных, организованных в пакеты, мы обычно используем маску, то есть не обращаем внимания на метки с заполнениями. Таким образом, токен, выбранный для заполнения, при обучении модели ни на что не влияет.

Более того, токенизатор, используемый для моделей GPT, не использует токен `<|unk|>` для слов, которые не входят в словарь. Вместо этого в моделях GPT задействуется токенизатор кодирования пар байтов, который разбивает слова на более мелкие составляющие. О нем мы и поговорим далее.

2.5. Кодирование пар байтов

В этом разделе мы рассмотрим более сложную схему токенизации, основанную на концепции *кодирования пар байтов* (byte pair encoding, BPE). Токенизатор BPE использовался для обучения больших языковых моделей, таких как GPT-2, GPT-3, и оригинальной модели в ChatGPT.

Реализация BPE может быть относительно сложной, поэтому мы воспользуемся существующей библиотекой Python с открытым исходным кодом под названием *tiktoken* (<https://github.com/openai/tiktoken>), которая эффективно реализует

алгоритм BPE на основе исходного кода на Rust. Как и другие библиотеки Python, установить библиотеку `tiktoken` можно с помощью установщика `pip` из терминала:

```
pip install tiktoken
```

Код, который мы будем использовать, основан на `tiktoken 0.7.0`. Вы можете использовать следующий код, чтобы проверить, какая версия установлена у вас:

```
from importlib.metadata import version
import tiktoken
print("tiktoken version:", version("tiktoken"))
```

После установки мы можем создать экземпляр токенизатора BPE из `tiktoken` следующим образом:

```
tokenizer = tiktoken.get_encoding("gpt2")
```

Использование данного токенизатора аналогично `SimpleTokenizerV2`:

```
text = (
    "Hello, do you like tea? <|endoftext|> In the sunlit terraces"
    "of someunknownPlace."
)
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})
print(integers)
```

Мы получаем следующие идентификаторы токенов:

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252, 18250, 8812,
2114, 286, 617, 34680, 27271, 13]
```

Затем идентификаторы токенов можно преобразовать обратно в текст, используя метод декодирования, аналогичный нашему `SimpleTokenizerV2`:

```
strings = tokenizer.decode(integers)
print(strings)
```

Результат выглядит так:

```
Hello, do you like tea? <|endoftext|> In the sunlit terraces of
someunknownPlace.
```

Основываясь на идентификаторах токенов и декодированном тексте, мы можем сделать два примечательных наблюдения.

Во-первых, токену `<|endoftext|>` присваивается относительно большой идентификатор, а именно `50256`. На самом деле общий размер словаря токенизатора BPE, который использовался для обучения таких моделей, как GPT-2, GPT-3, и оригинальной модели, применяемой в ChatGPT, составляет `50 257` слов, при этом `<|endoftext|>` получает самое большое значение.

Во-вторых, токенизатор BPE правильно кодирует и декодирует неизвестные слова, такие как `someunknownPlace`. Токенизатор BPE может работать с любым неизвестным словом. Как ему это удастся без использования токенов `<|unk|>`?

Алгоритм, лежащий в основе BPE, разбивает слова, которых нет в его предопределенном словаре, на более мелкие части или даже отдельные символы, что позволяет ему работать со словами, отсутствующими в словаре. Таким образом, благодаря алгоритму BPE токенизатор, встречая незнакомое слово, может представить его в виде последовательности токенов или символов (рис. 2.11).

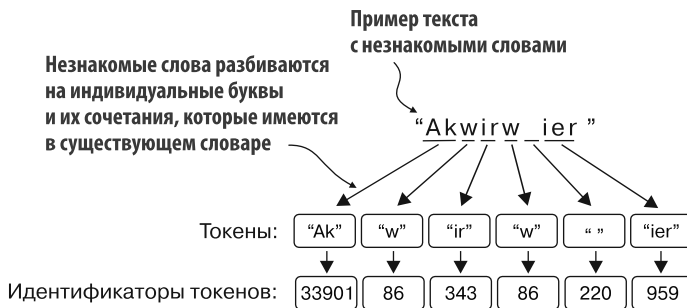


Рис. 2.11. Токенизаторы BPE разбивают неизвестные слова на части и отдельные символы. Таким образом, токенизатор BPE может анализировать любое слово и не нуждается в замене неизвестных слов специальными токенами, такими как `<|unk|>`

Благодаря способности токенизатора разбивать неизвестные слова на отдельные символы сам он и, следовательно, обученная с его помощью LLM смогут обрабатывать любой текст, даже если он содержит слова, которых не было в обучающих данных.

Упражнение 2.1. Кодирование неизвестных слов с помощью пар байтов

Попробуйте токенизатор BPE из библиотеки `tiktoken` на неизвестных словах `Akwirw ier` и напечатайте идентификаторы отдельных токенов. Затем вызовите функцию `decode` для каждого из полученных целых чисел в этом списке, чтобы воспроизвести результат, показанный на рис. 2.11. Наконец, вызовите метод `decode` для идентификаторов токенов, чтобы проверить, может ли он восстановить исходный ввод, `Akwirw ier`.

Подробное обсуждение и реализация BPE выходят за рамки этой книги, но я все же дам краткое пояснение. BPE формирует свой словарь путем итеративного объединения часто встречающихся символов в части слов, а часто встречающихся частей — в целые слова. Например, он начинает с добавления

в словарь всех отдельных символов (a , b и т. д.). На следующем этапе он объединяет часто встречающиеся комбинации символов в части слов. Например, d и e могут быть объединены в сочетание de , которое часто встречается во многих английских словах, таких как *define*, *depend*, *made* и *hidden*. Слияния определяются по частотному порогу.

2.6. Выборка данных с помощью скользящего окна

Следующий шаг при создании векторных представлений для LLM — генерация пар «входные данные — цель», необходимых для обучения модели. Как выглядят эти пары? Вы уже знаете, что LLM предварительно обучаются, предсказывая следующее слово в тексте (рис. 2.12).

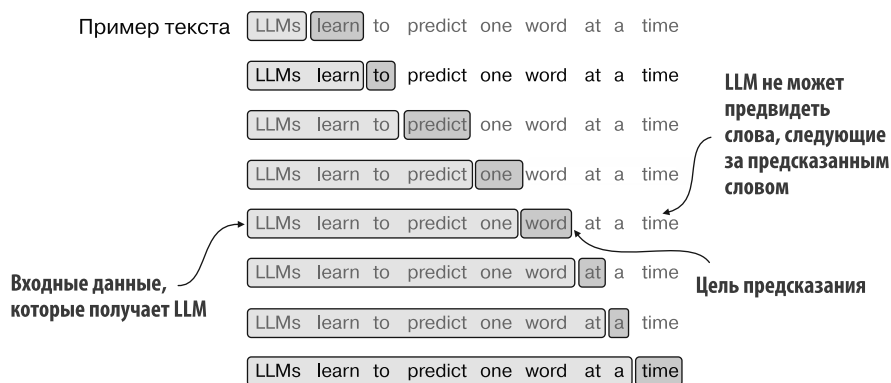


Рис. 2.12. В этом примере входные данные для LLM формируются из блоков текста, которые служат входными данными для модели, и задача LLM во время обучения состоит в том, чтобы предсказать целевое слово, которое следует за входным блоком. Во время обучения мы маскируем все слова, которые находятся после целевого

Обратите внимание: текст, показанный на этом рисунке, должен пройти токенизацию, прежде чем LLM сможет его обработать; однако в данном случае для наглядности шаг токенизации опущен.

Теперь реализуем загрузчик данных, который извлекает пары «входные данные — цель» из обучающей выборки данных с помощью скользящего окна. Для начала токенизируем весь рассказ *The Verdict*, используя токенизатор BPE:

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
```

```
enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```


Выполнение этого кода вернет 5145, общее количество токенов в обучающем наборе, после применения токенизатора ВРЕ.

Далее мы в демонстрационных целях удалим первые 50 токенов из набора данных, так как на следующих шагах это приведет к получению более интересного фрагмента текста:

```
enc_sample = enc_text[50:]
```

Один из самых простых и интуитивно понятных способов создания пар «входные данные — цель» для задачи предсказания следующего слова — создать две переменные, *x* и *y*, где *x* содержит входные токены, а *y* — цели, которые являются входными токенами, сдвинутыми на 1:

```
context_size = 4
x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y:      {y}")
```

← context_size определяет, сколько токенов
добавляется во входные данные

Получаем следующий результат:

```
x: [290, 4920, 2241, 287]
y: [4920, 2241, 287, 257]
```

Обработывая входные данные вместе с целевыми, которые представляют собой входные данные, сдвинутые на одну позицию, мы можем создать задачи по предсказанию следующего слова (см. рис. 2.12):

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(context, "---->", desired)
```

Вывод выглядит так:

```
[290] ----> 4920
[290, 4920] ----> 2241
[290, 4920, 2241] ----> 287
[290, 4920, 2241, 287] ----> 257
```

Все, что находится слева от стрелки (---->), относится к входным данным, которые получит LLM, а идентификатор токена справа от стрелки представляет собой целевой идентификатор токена, который модель должна предсказать. Теперь используем предыдущий код, но преобразуем идентификаторы токенов в текст:

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(tokenizer.decode(context), "---->", tokenizer.decode([desired]))
```

Вот как выглядят входные и выходные данные, когда идентификаторы замещаются словами:

```
and ----> established
and established ----> himself
and established himself ----> in
and established himself in ----> a
```

Теперь мы создали пары «входные данные — цель», которые можно использовать для обучения LLM.

Прежде чем мы сможем преобразовать токены во вложенные векторы, осталось выполнить только одну задачу — реализовать эффективный загрузчик данных, который перебирает входной набор данных и возвращает входные данные и целевые значения в виде тензоров PyTorch, которые можно рассматривать как многомерные массивы. В частности, нас интересует возврат двух тензоров: входного, содержащего текст, который видит LLM, и целевого, содержащего целевые значения, которые модель должна предсказать (рис. 2.13). Для наглядности токены на рисунке показаны в виде строк, но реализация кода будет работать непосредственно с идентификаторами токенов, поскольку метод `encode` токенизатора BPE выполняет как токенизацию, так и преобразование в идентификаторы токенов за один шаг.

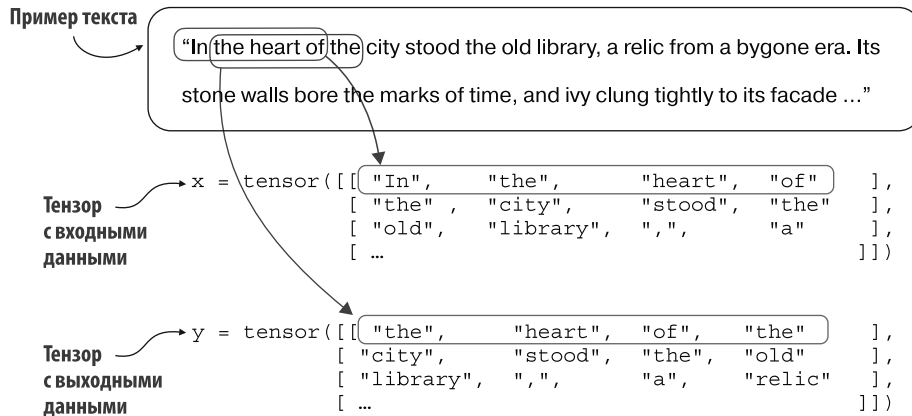


Рис. 2.13. Для реализации эффективных загрузчиков данных мы собираем входные данные в тензор `x`, где каждая строка представляет собой один вход. Второй тензор, `y`, содержит соответствующие целевые значения (следующие слова), которые создаются путем смещения входных данных на одну позицию

ПРИМЕЧАНИЕ Для эффективной реализации загрузчика данных мы будем использовать встроенные в PyTorch классы `Dataset` и `DataLoader`. Дополнительная информация и рекомендации по установке PyTorch представлены в подразделе А.1.3 приложения А.

Код класса формирования данных показан в листинге 2.5.

Листинг 2.5. Формирование данных, состоящих из пакетов

```
import torch
from torch.utils.data import Dataset, DataLoader

class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride)::
        self.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(txt)

        for i in range(0, len(token_ids) - max_length, stride):
            input_chunk = token_ids[i:i + max_length]
            target_chunk = token_ids[i + 1: i + max_length + 1]
            self.input_ids.append(torch.tensor(input_chunk))
            self.target_ids.append(torch.tensor(target_chunk))

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return self.input_ids[idx], self.target_ids[idx]
```

Применяет метод скользящего окна для деления книги на пересекающиеся последовательности текста длины max_length

Токенизирует весь текст

Возвращает общее количество строк в наборе данных

Возвращает одну строку из набора данных

Класс `GPTDatasetV1` основан на классе `Dataset` PyTorch и определяет, как извлекаются отдельные строки из набора данных, где каждая строка состоит из нескольких идентификаторов токенов (в зависимости от `max_length`), присвоенных тензору `input_chunk`. Данный тензор содержит соответствующие целевые значения. Я рекомендую продолжать чтение, чтобы узнать, как выглядят данные из этой выборки данных, когда мы объединяем ее с `DataLoader`, — это даст вам дополнительную информацию и прояснит ситуацию.

ПРИМЕЧАНИЕ Если вы не знакомы со структурой классов `Dataset`, таких как показанные в листинге 2.5, обратитесь к разделу A.6 приложения A, где объясняется общая структура и использование классов `Dataset` и `DataLoader`.

В следующем коде используется `GPTDatasetV1` для загрузки входных данных пакетами с помощью `DataLoader` PyTorch (листинг 2.6).

Листинг 2.6. Загрузчик данных для создания пакетов

```
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                        stride=128, shuffle=True, drop_last=True,
                        num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2")
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
    dataloader = DataLoader(
        dataset,
```

Инициализирует токенизатор

Создает выборку данных

```

    batch_size=batch_size,
    shuffle=shuffle,
    drop_last=drop_last,
    num_workers=num_workers
)
return dataloader

```

drop_last=True опускает последний пакет данных, если его размер короче, чем batch_size, чтобы предотвратить неожиданный всплеск функции потерь при обучении модели

Количество процессов CPU, используемых при предобработке

Теперь протестируем загрузчик данных с размером пакета 1 для LLM с размером контекста 4, чтобы выяснить, как класс `GPTDatasetV1` из листинга 2.5 и функция `create_dataloader_v1` из листинга 2.6 работают вместе:

```

with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

dataloader = create_dataloader_v1(
    raw_text, batch_size=1, max_length=4, stride=1, shuffle=False)
data_iter = iter(dataloader)
first_batch = next(data_iter)
print(first_batch)

```

Преобразует dataloader в Python-итератор для подачи следующей порции данных с помощью встроенной функции next()

При запуске данного кода мы получаем следующий результат:

```
[tensor([[ 40, 367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1807]])]
```

Переменная `first_batch` содержит два тензора: первый хранит идентификаторы входных токенов, а второй — идентификаторы целевых токенов. Поскольку `max_length` установлено на 4, то каждый из тензоров содержит по четыре идентификатора токенов. Обратите внимание: размер входных данных 4 довольно мал и выбран только для простоты. Обычно LLM обучают на входных данных размером не менее 256.

Чтобы понять значение `stride=1`, получим еще один пакет данных из этой выборки:

```

second_batch = next(data_iter)
print(second_batch)

```

Второй пакет данных выглядит так:

```
[tensor([[ 367, 2885, 1464, 1807]]), tensor([[2885, 1464, 1807, 3619]])]
```

Если мы сравним первый и второй пакеты, то увидим, что идентификаторы токенов из второго пакета сдвинуты на одну позицию вправо (например, второй идентификатор во входных данных первого пакета — 367, и он появляется первым во входных данных второго). Значение `stride` определяет количество позиций, на которое сдвигаются входные данные в разных пакетах, имитируя подход со скользящим окном (рис. 2.14).

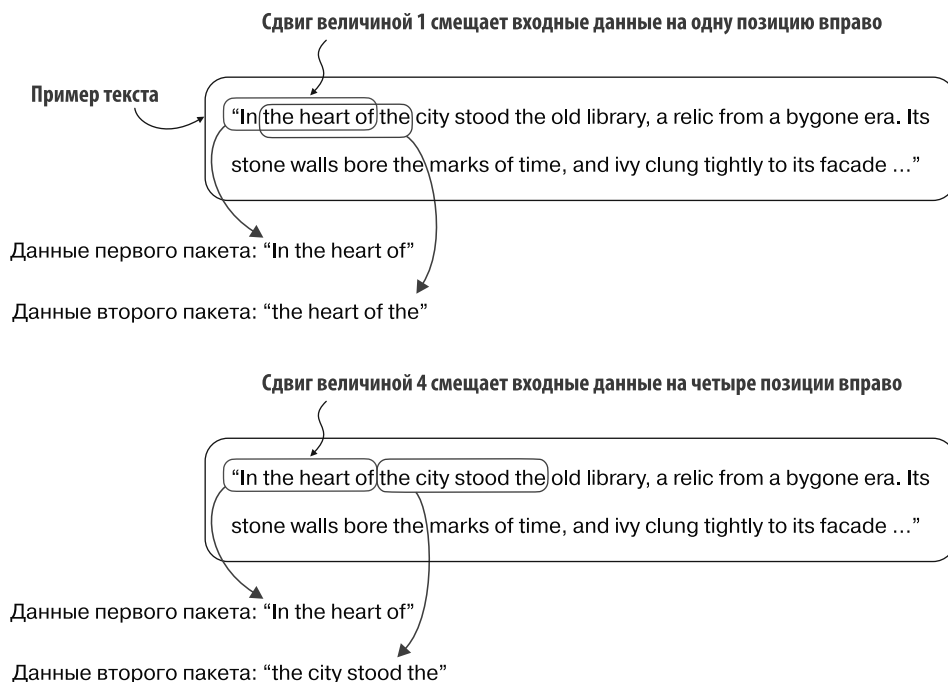


Рис. 2.14. При создании нескольких пакетов из входного набора данных мы перемещаем окно по тексту. Если шаг равен 1, то мы сдвигаем окно на одну позицию вправо, чтобы создать следующий пакет. Установив шаг смещения равным размеру окна, мы сможем избежать пересечений между пакетами

Упражнение 2.2. Загрузчики данных с разными значениями шага смещения и размера контекста

Чтобы лучше понять, как работает загрузчик данных, попробуйте запустить его с разными настройками, например, `max_length=2` и `stride=2`, а также `max_length=8` и `stride=2`.

Размер пакета, равный 1, который мы использовали в загрузчике данных, полезен для наглядности. Если вы уже занимались глубоким обучением, то, возможно, знаете, что небольшие размеры пакетов требуют меньше памяти во время обучения, но приводят к обновлениям модели, содержащим больше шумов. Как и в обычном глубоком обучении, размер пакета является компромиссом и гиперпараметром, с которым можно экспериментировать при обучении LLM.

Коротко рассмотрим, как можно использовать загрузчик данных для выборки с пакетом, размер которого больше 1:

```
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=4, stride=4,
    shuffle=False
)

data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("\nTargets:\n", targets)
```

Данный код выводит следующий результат:

```
Inputs:
tensor([[ 40, 367, 2885, 1464],
        [ 1807, 3619, 402, 271],
        [10899, 2138, 257, 7026],
        [15632, 438, 2016, 257],
        [ 922, 5891, 1576, 438],
        [ 568, 340, 373, 645],
        [ 1049, 5975, 284, 502],
        [ 284, 3285, 326, 11]])

Targets:
tensor([[ 367, 2885, 1464, 1807],
        [ 3619, 402, 271, 10899],
        [ 2138, 257, 7026, 15632],
        [ 438, 2016, 257, 922],
        [ 5891, 1576, 438, 568],
        [ 340, 373, 645, 1049],
        [ 5975, 284, 502, 284],
        [ 3285, 326, 11, 287]])
```

Обратите внимание: мы увеличиваем шаг до 4, чтобы полностью использовать набор данных (мы не пропускаем ни одного слова). Это позволяет избежать наложения пакетов друг на друга, так как большее наложение может привести к излишнему обучению модели.

2.7. Создание векторных представлений токенов

Последний шаг в подготовке входного текста для обучения LLM — преобразование идентификаторов токенов в векторные представления (рис. 2.15). В качестве предварительного действия мы должны инициализировать весовые коэффициенты вложения случайными значениями. Такая инициализация служит отправной точкой для процесса обучения LLM. В главе 5 мы оптимизируем весовые коэффициенты вложения в рамках обучения LLM.

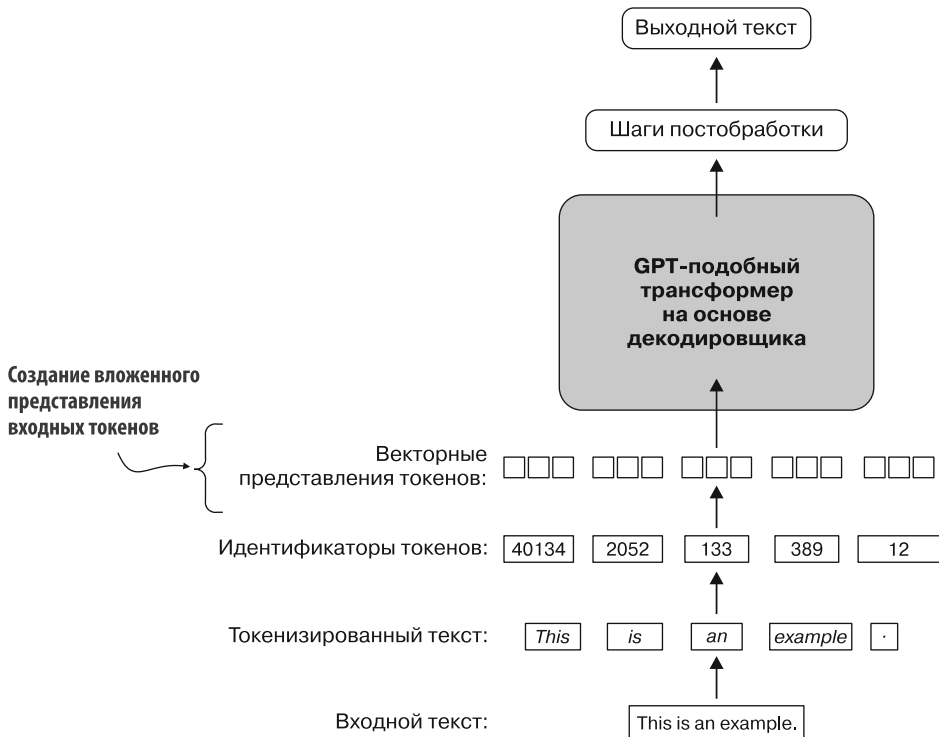


Рис. 2.15. В процесс подготовки текста входит токенизация текста, преобразование текстовых токенов в идентификаторы токенов и преобразование этих идентификаторов в векторы вложения. В данном случае мы используем ранее созданные идентификаторы токенов для получения вложенных векторов токенов

Непрерывное векторное представление, или вложение, необходимо, поскольку LLM, подобные GPT, представляют собой глубокие нейронные сети, обученные с помощью алгоритма обратного распространения ошибки.

ПРИМЕЧАНИЕ Если вы не знакомы с тем, как нейронные сети обучаются с помощью данного алгоритма, то, пожалуйста, прочтите раздел А.4 приложения А.

Рассмотрим преобразование идентификатора токена во вложенный вектор на практическом примере. Допустим, у нас есть следующие четыре входных токена с идентификаторами 2, 3, 5 и 1:

```
input_ids = torch.tensor([2, 3, 5, 1])
```

Для простоты предположим, что у нас есть небольшой словарь, состоящий всего из шести слов (вместо 50 257 слов в словаре токенизатора BPE), и мы хотим создать векторные представления размером 3 (в GPT-3 размер векторных представлений составляет 12 288 измерений):

```
vocab_size = 6
output_dim = 3
```

Используя `vocab_size` и `output_dim`, мы можем создать экземпляр слоя вложения в PyTorch, задав для генератора случайных чисел начальное значение 123 в целях воспроизводимости результатов:

```
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```

Оператор `print` выводит следующую матрицу весов слоя вложения:

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
        [ 0.9178, 1.5810, 1.3010],
        [ 1.2753, -0.2010, -0.1606],
        [-0.4015, 0.9666, -1.1481],
        [-1.1589, 0.3255, -0.6315],
        [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

Данная матрица содержит небольшие случайные значения. Они оптимизируются во время обучения LLM как часть оптимизации самой модели. Кроме того, мы видим, что весовая матрица состоит из шести строк и трех столбцов. В ней есть по одной строке для каждого из шести возможных токенов в словаре и по одному столбцу для каждого из трех размерностей вложения.

Теперь применим ее к идентификатору токена, чтобы получить вектор вложения:

```
print(embedding_layer(torch.tensor([3])))
```

Возвращаемый вектор выглядит так:

```
tensor([-0.4015, 0.9666, -1.1481]), grad_fn=<EmbeddingBackward0>)
```

Если мы сравним вектор вложения для идентификатора токена 3 с предыдущей матрицей вложения, то увидим, что он идентичен четвертой строке (Python начинает «счет» с нулевого индекса, поэтому это строка, соответствующая индексу 3). Другими словами, слой вложения — это, по сути, операция поиска, которая извлекает строки из весовой матрицы слоя вложения по идентификатору токена.

ПРИМЕЧАНИЕ Для тех, кто знаком с прямым (one-hot) кодированием, описанный здесь подход с использованием слоя вложения выглядит как, по сути, более эффективный способ реализации данного кодирования с последующим перемножением матриц в полностью связанном слое, что показано в дополнительном коде, размещенном на GitHub по адресу <https://mng.bz/ZEB5>. Поскольку слой вложения — просто более эффективная реализация, эквивалентная прямому кодированию и перемножению матриц, то его можно рассматривать как слой нейронной сети, который можно оптимизировать с помощью алгоритма обратного распространения ошибки.

Вы увидели, как преобразовать один идентификатор токена в трехмерный вектор вложения. Теперь применим это ко всем четырем входным идентификаторам (`torch.tensor([2, 3, 5, 1])`):

```
print(embedding_layer(input_ids))
```

Результат — матрица 4×3 :

```
tensor([[ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-2.8400, -0.7849, -1.4096],
        [ 0.9178,  1.5810,  1.3010]], grad_fn=<EmbeddingBackward0>)
```

Каждая строка в этой выходной матрице образуется с помощью операции поиска в матрице весовых коэффициентов (рис. 2.16).

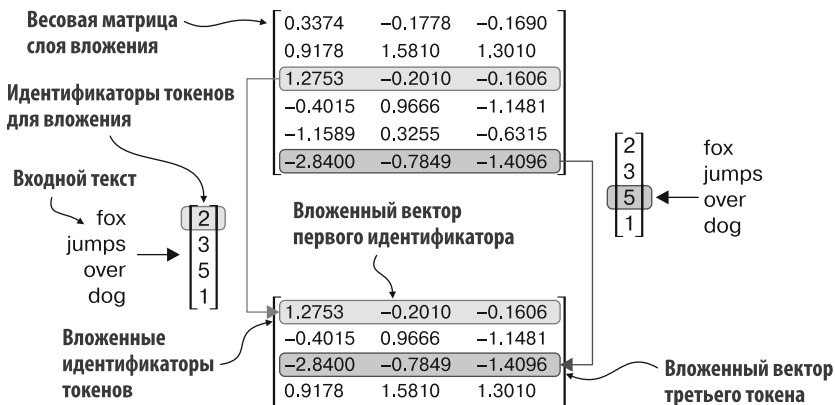


Рис. 2.16. Слои вложения выполняют операцию поиска, извлекая вектор вложения, соответствующий идентификатору токена, из весовой матрицы слоя вложения. Например, вектор вложения для идентификатора токена 5 — это шестая строка весовой матрицы слоя вложения (шестая, а не пятая, поскольку Python начинает отсчет с 0). Мы предполагаем, что идентификаторы токенов были получены с помощью небольшого словаря из раздела 2.3

Теперь, когда мы создали векторы вложения на основе идентификаторов токенов, слегка изменим эти векторы, чтобы закодировать информацию о позиции токена в тексте.

2.8. Кодирование позиций слов

В принципе, векторы вложения токенов — подходящий входной сигнал для моделей. Однако незначительным недостатком LLM является то, что их механизм самовнимания (см. главу 3) не учитывает положение или порядок токенов в последовательности. Ранее представленный слой вложения работает так, что один и тот же идентификатор токена всегда сопоставляется с одним и тем же векторным представлением, независимо от того, в каком месте последовательности находится идентификатор токена (рис. 2.17).

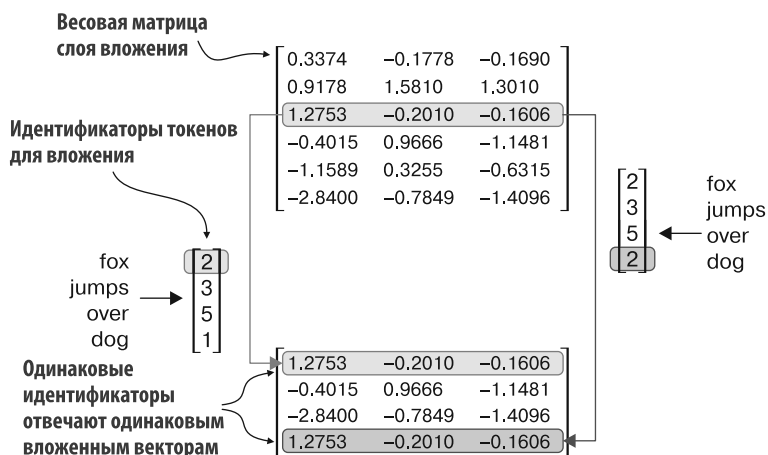


Рис. 2.17. Слой вложения преобразует идентификатор токена в одно и то же векторное представление независимо от его позиции во входной последовательности. Например, идентификатор токена 5, независимо от того, в какой позиции во входном векторе идентификаторов токенов он находится: в первой или четвертой, приведет к одному и тому же вектору вложения

В общем-то, детерминированное, не зависящее от местоположения идентификатора токена, векторное представление полезно для воспроизводимости. Но поскольку механизм самовнимания LLM тоже не зависит от местоположения, то полезно ввести дополнительную информацию о местоположении в модели.

Достичь этой цели можно с помощью двух широких категорий вложений с учетом местоположения: относительные позиционные вложения и абсолютные

позиционные вложения. Абсолютные напрямую связаны с определенными позициями в последовательности. Для каждой позиции во входной последовательности к вложению токена добавляется уникальное вложение, чтобы передать его точное местоположение. Например, первое вложение будет иметь одну позицию, второе — другую и т. д. (рис. 2.18).

Вместо того чтобы фокусироваться на абсолютном положении токена, в относительных позиционных вложениях акцент делается на относительной позиции или расстоянии между токенами. Это означает, что модель изучает взаимосвязи с точки зрения «на каком расстоянии друг от друга», а не «на какой именно позиции». Преимущество здесь в том, что модель может лучше адаптироваться к последовательностям разной длины, даже если не встречала их во время обучения.

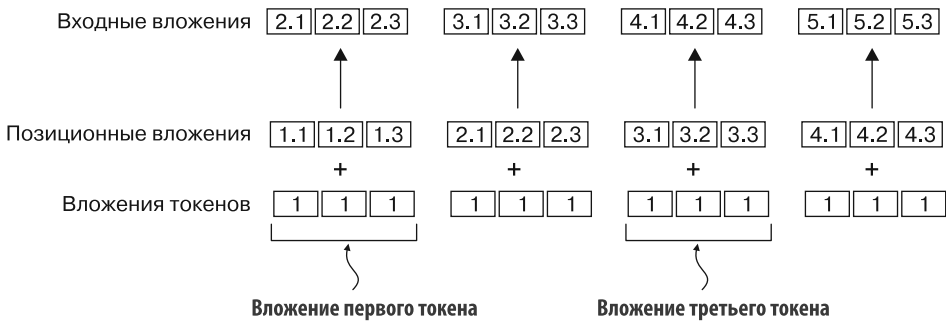


Рис. 2.18. Позиционные векторы добавляются к вектору вложения токенов для создания входных векторов для LLM. Позиционные векторы имеют ту же размерность, что и исходные векторы вложения токенов. Для простоты векторы вложения токенов здесь показаны со значением 1

Оба типа позиционных вложений направлены на расширение возможностей LLM по части понимания порядка и взаимосвязей между токенами, обеспечивая более точные прогнозы с учетом контекста. Выбор между ними часто зависит от конкретного приложения и природы обрабатываемых данных.

В GPT-моделях OpenAI используются абсолютные позиционные вложения, которые оптимизируются в процессе обучения, а не являются фиксированными или предопределенными, как позиционные кодировки в оригинальной модели трансформера. Этот процесс оптимизации является частью обучения самой модели.

Создадим начальные позиционные вложения, чтобы сформировать входные данные для LLM.

Ранее мы для простоты использовали очень маленькие размеры векторных представлений. Теперь рассмотрим более реалистичные и полезные размеры и закодируем входные токены в 256-мерное векторное представление, которое меньше, чем в оригинальной модели GPT-3 (размер представлений в которой составляет 12 288), но все равно подходит для экспериментов. Кроме того, мы предполагаем, что идентификаторы токенов были созданы токенизатором BPE, который мы использовали ранее и в словаре которого насчитывается 50 257 слов:

```
vocab_size = 50257
output_dim = 256
token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

Используя предыдущий слой `token_embedding_layer`, когда мы извлекаем данные из загрузчика данных, мы преобразуем каждый токен в каждом пакете в 256-мерный вектор. Если размер пакета равен 8, а в каждом пакете по четыре токена, результатом будет тензор $8 \times 4 \times 256$.

Сначала создадим экземпляр загрузчика данных (см. раздел 2.6):

```
max_length = 4
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=max_length,
    stride=max_length, shuffle=False
)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Token IDs:\n", inputs)
print("\nInputs shape:\n", inputs.shape)
```

Результат выглядит так:

```
Token IDs:
tensor([[ 40,  367, 2885, 1464],
        [1807, 3619,  402,  271],
        [10899, 2138,  257, 7026],
        [15632,  438, 2016,  257],
        [ 922, 5891, 1576,  438],
        [ 568,  340,  373,  645],
        [1049, 5975,  284,  502],
        [ 284, 3285,  326,  111] ])
```

```
Inputs shape:
torch.Size([8, 4])
```

Как мы видим, тензор идентификаторов токенов имеет размерность 8×4 ; это значит, пакет данных состоит из восьми текстовых примеров с четырьмя токенами в каждом.

Теперь используем слой вложения, чтобы преобразовать эти идентификаторы токенов в 256-мерные векторы:

```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)
```

Функция `print` возвращает следующий результат:

```
torch.Size([8, 4, 256])
```

Выходной $8 \times 4 \times 256$ -мерный тензор показывает, что каждый идентификатор токена теперь представлен в виде 256-мерного вектора.

Для абсолютного подхода к вложению в модели GPT просто нужно создать еще один слой вложения с той же размерностью, что и у слоя `token_embedding_layer`:

```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)
pos_embeddings = pos_embedding_layer(torch.arange(context_length))
print(pos_embeddings.shape)
```

Входными данными для `pos_embeddings` обычно является вектор-заполнитель `torch.arange(context_length)`, который содержит последовательность чисел 0, 1... до максимальной длины входных данных — 4. Элемент `context_length` — это переменная, которая представляет размер входных данных LLM. Здесь мы выбираем ее аналогично максимальной длине входного текста. На практике входной текст может быть длиннее, чем `context_length`, и в этом случае приходится обрезать текст.

Результат выполнения оператора `print` выглядит так:

```
torch.Size([4, 256])
```

Как мы видим, тензор позиционного вложения состоит из четырех 256-мерных векторов. Теперь мы можем добавить их непосредственно к вложению токенов, где PyTorch добавит тензор `pos_embeddings` размером 4×256 к каждому тензору вложения токенов размером 4×256 в каждой из восьми партий:

```
input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)
```

Результат таков:

```
torch.Size([8, 4, 256])
```

Созданные нами `input_embeddings` — это вложения входных данных, которые теперь могут быть обработаны основными модулями LLM, которые мы начнем реализовывать в следующей главе.

Полная схема обработки входных данных показана на рис. 2.19.

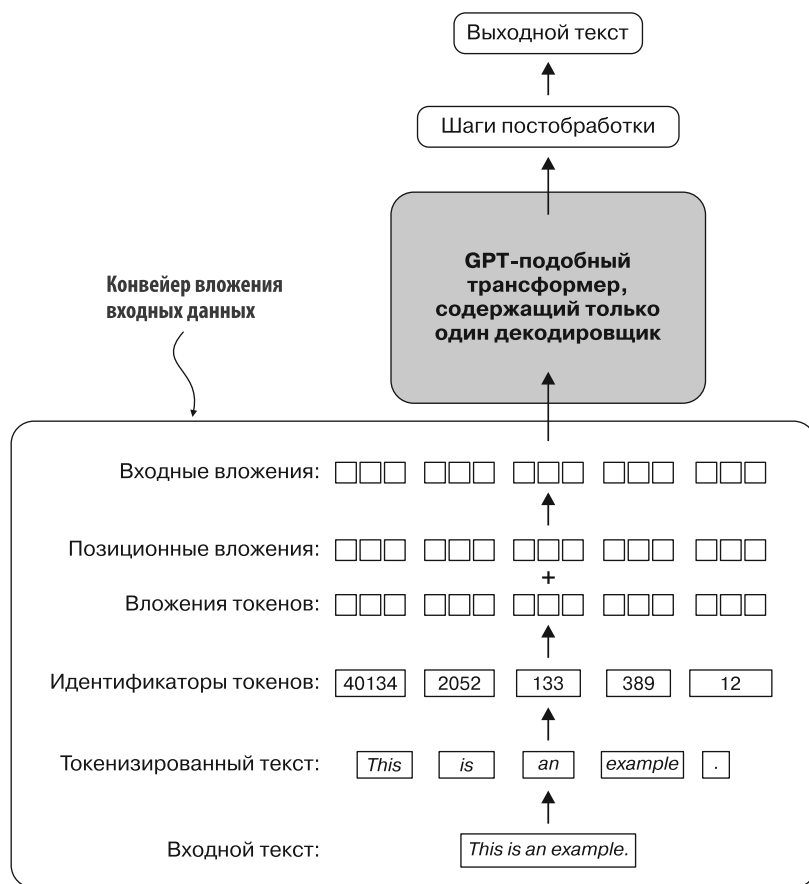


Рис. 2.19. Полная схема обработки входных данных

Таким образом, в рамках конвейера обработки входных данных текст сначала разбивается на отдельные токены. Затем они преобразуются в идентификаторы токенов с помощью словаря. Идентификаторы преобразуются в векторы вложения, к которым добавляются позиционные векторы вложения аналогичного размера. В результате получают входные векторы вложения, которые используются в качестве входных данных для основных слоев LLM.

Итоги главы

- Большим языковым моделям требуются текстовые данные, преобразованные в числовые векторы, известные как вложения (эмбединги), поскольку LLM не могут обрабатывать текст как есть. Вложения преобразуют дискретные данные (например, слова или изображения) в числовые векторы, делая их совместимыми с операциями нейронных сетей.
- Сначала необработанный текст разбивается на токены, которые могут быть словами или символами. Затем токены преобразуются в целочисленные представления, называемые идентификаторами токенов.
- Специальные токены, такие как `<|unk|>` и `<|endoftext|>`, могут быть добавлены для улучшения способности модели понимать текст и обрабатывать различные контексты, такие как неизвестные слова или границы между несвязанными текстами.
- Токенизатор пар байтов (BPE), используемый для больших языковых моделей, таких как GPT-2 и GPT-3, может эффективно обрабатывать неизвестные слова, разбивая их на части или отдельные символы.
- Скользящее окно можно использовать для токенизированных данных, чтобы генерировать пары «входные данные — цель» для обучения LLM.
- Слои вложения в PyTorch функционируют как операция поиска, возвращая векторы, соответствующие идентификаторам токенов. Полученные векторы вложения обеспечивают непрерывное представление токенов, что крайне важно для обучения моделей глубокого обучения, таких как LLM.
- Векторные представления токенов используются для приемлемого представления каждого токена, но не учитывают его положение в последовательности. Исправить этот недостаток помогают два основных типа позиционных векторов: абсолютные и относительные. Модели GPT от OpenAI используют абсолютные позиционные векторы, которые добавляются к векторам токенов и оптимизируются во время обучения модели.

3

Программирование механизмов внимания

В этой главе

- ✓ Причины использования механизмов внимания в нейронных сетях.
- ✓ Базовая структура самовнимания, переходящая в усовершенствованный механизм самовнимания.
- ✓ Модуль причинно-следственного внимания, который позволяет LLM генерировать по одному токenu за раз.
- ✓ Маскировка случайно выбранных весовых коэффициентов внимания с помощью отсева.
- ✓ Объединение нескольких модулей причинно-следственного внимания в модуль многоцелевого внимания.

К этому моменту вы знаете, как подготовить входной текст для обучения модели, разделив его на отдельные слова и более мелкие токены, которые можно закодировать в виде векторных представлений для LLM.

Теперь мы рассмотрим неотъемлемую часть архитектуры LLM — механизмы внимания (рис. 3.1). Мы рассмотрим их в изоляции от всего остального. Затем мы реализуем в коде оставшиеся части модели, чтобы увидеть механизм самовнимания в действии и создать модель для генерации текста.

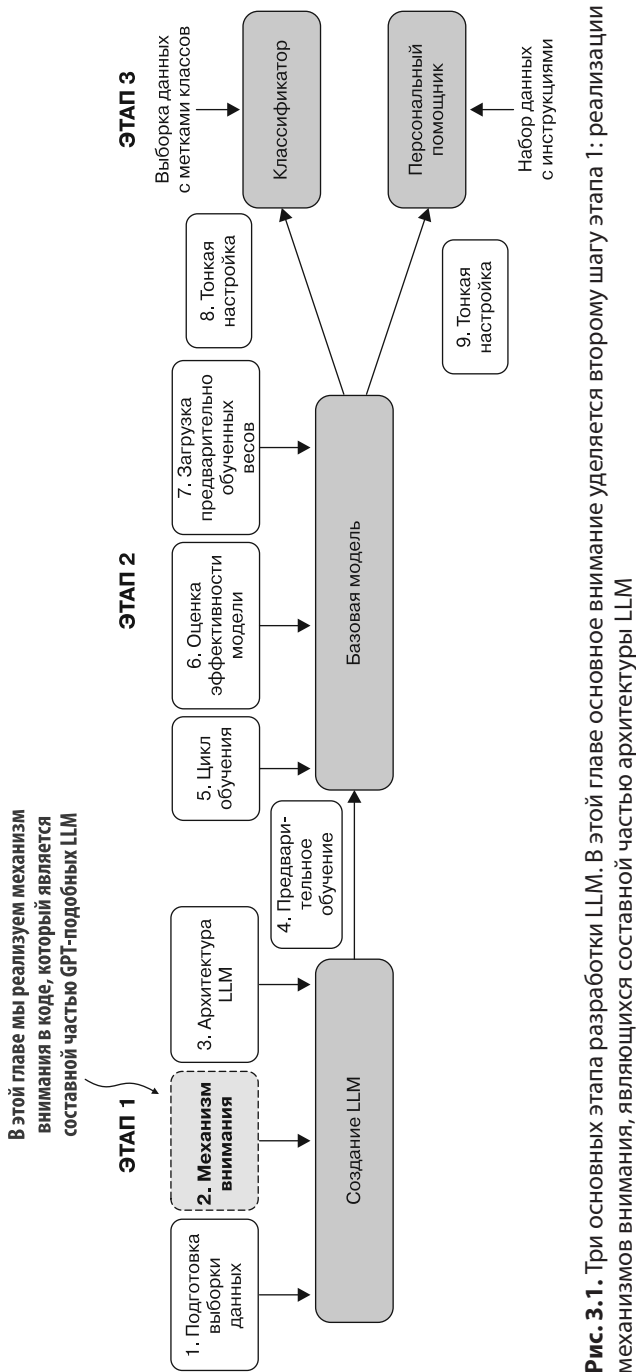


Рис. 3.1. Три основных этапа разработки LLM. В этой главе основное внимание уделяется второму шагу этапа 1: реализации механизмов внимания, являющихся составной частью архитектуры LLM

Мы реализуем четыре различных варианта механизмов внимания (рис. 3.2). Они дополняют друг друга, и цель состоит в том, что мы получим компактную и эффективную реализацию многоцелевого внимания, которую затем можно будет встроить в архитектуру LLM, реализованную в следующей главе.

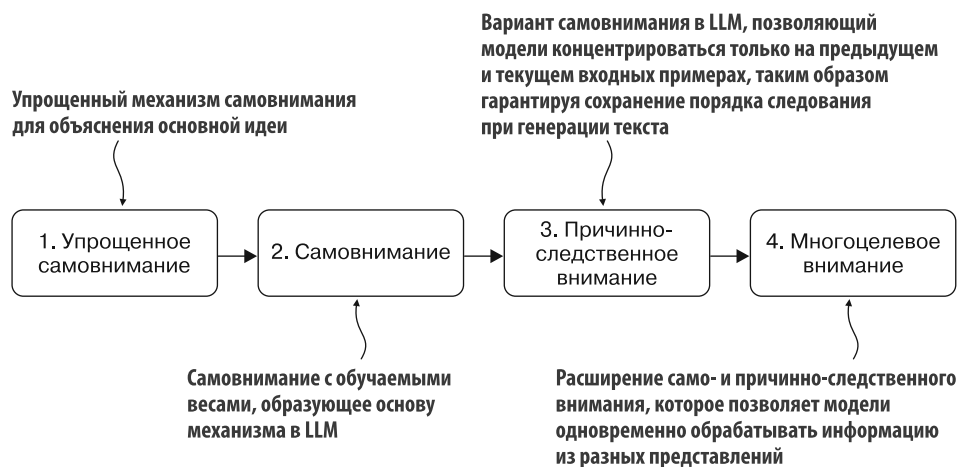


Рис. 3.2. Различные механизмы внимания

Мы начнем с упрощенной версии самовнимания, прежде чем добавлять обучаемые веса. Механизм причинно-следственного внимания добавляет к самовниманию маску, которая позволяет LLM генерировать по одному слову за раз. Наконец, многоцелевое внимание адаптирует механизм внимания, чтобы модель могла отслеживать несколько путей генерации текста одновременно, параллельно фиксируя различные аспекты входных данных.

3.1. Проблема моделирования длинных последовательностей

Прежде чем углубляться в механизм самовнимания, лежащий в основе больших языковых моделей, рассмотрим проблему, связанную с архитектурами, которые предшествуют LLM и не имеют в своей структуре механизмов внимания.

Предположим, мы хотим разработать модель, которая переводит текст с одного языка на другой. Как показано на рис. 3.3, мы не можем просто переводить текст слово за словом из-за грамматических структур в исходном и целевом языках.

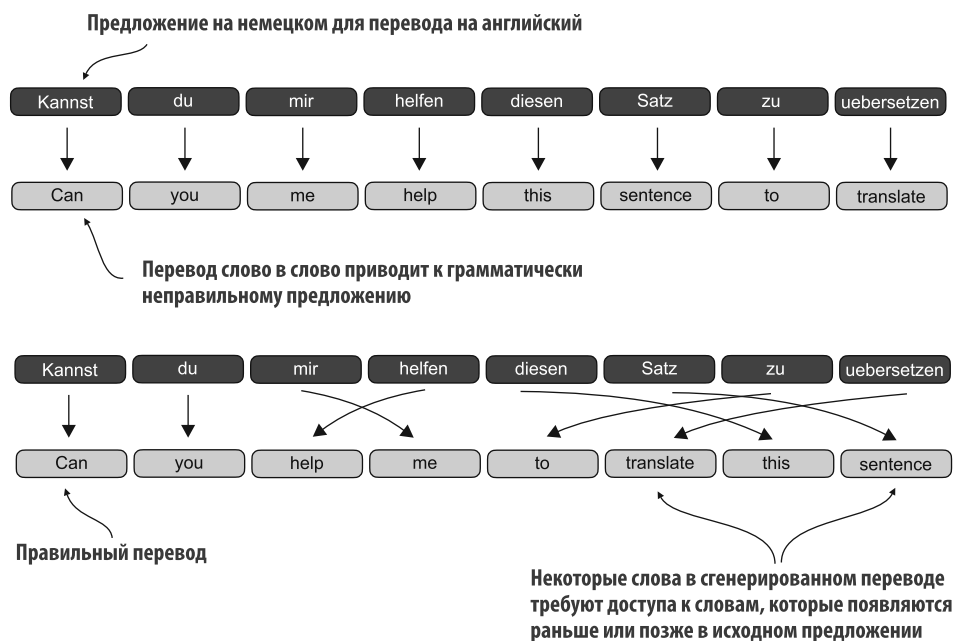


Рис. 3.3. При переводе текста с одного языка на другой, например с немецкого на английский, невозможно просто переводить слово за словом. Процесс требует понимания контекста и грамматического согласования смысла перевода

Для решения этой проблемы обычно используется глубокая нейронная сеть с двумя подмодулями: *кодировщиком* и *декодировщиком*. Задача кодировщика — сначала считать и обработать весь текст, а затем декодировщик выдает переведенный текст.

До появления трансформеров *рекуррентные нейронные сети* (recurrent neural networks, RNN) были самой популярной архитектурой «кодировщик — декодировщик» для языкового перевода. RNN — тип нейронной сети, в которой выходные данные, полученные на предыдущих шагах, подаются в качестве входных данных на текущем шаге, что делает их хорошо подходящими для обработки последовательностей, таких как текст. Если вы не знакомы с RNN, то не волнуйтесь — вам не нужно знать, как работают эти сети, чтобы понять содержимое данного раздела. Мы сосредоточимся на общей концепции кодировщика и декодировщика.

В рекуррентной нейронной сети с архитектурой «кодировщик — декодировщик» входной текст поступает в кодировщик, который обрабатывает его последовательно. Кодировщик обновляет свое скрытое состояние (внутренние

значения в скрытых слоях) на каждом шаге, пытаюсь определить смысл всей входной фразы, заключенный в финальном скрытом состоянии (рис. 3.4). Затем декодировщик использует это финальное скрытое состояние, чтобы начать генерировать переведенную фразу, по одному слову за раз. Кроме того, на каждом шаге он обновляет собственное скрытое состояние, которое должно содержать контекст, необходимый для генерации следующего слова.

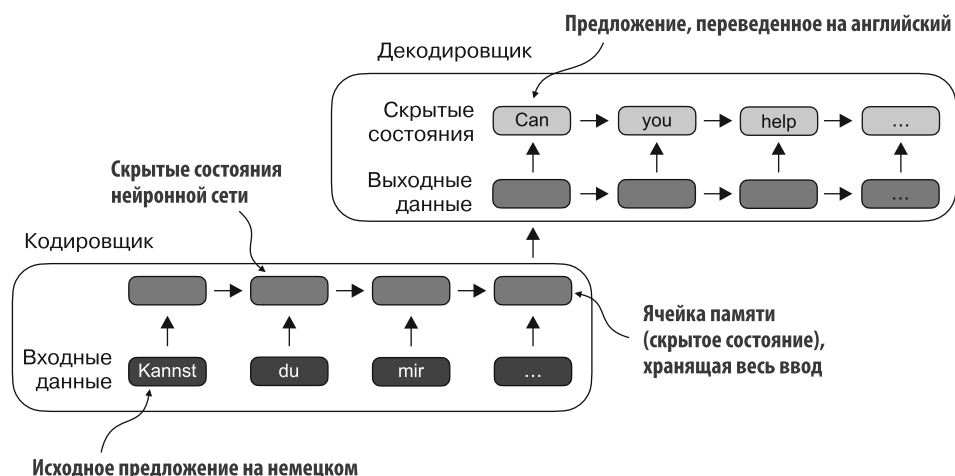


Рис. 3.4. Схема работы рекуррентной нейронной сети с архитектурой «кодировщик — декодировщик»

Ключевая идея состоит в том, что часть кодировщика преобразует весь входной текст в скрытое состояние (ячейку памяти). Затем декодировщик использует это состояние для создания выходного сигнала. Вы можете рассматривать это скрытое состояние как вектор вложения — концепцию, которую мы обсуждали в главе 2.

Большим ограничением RNN с архитектурой «кодировщик — декодировщик» является то, что сеть не может напрямую получить доступ к ранее скрытым состояниям из кодировщика на этапе декодирования. Следовательно, RNN использует исключительно текущее скрытое состояние, которое содержит всю необходимую информацию. Это может привести к потере контекста, особенно в сложных предложениях, где зависимости могут распространяться на большие расстояния.

К счастью, для создания LLM необязательно разбираться в рекуррентных нейронных сетях. Просто помните, что у RNN с архитектурой «кодировщик — декодировщик» был недостаток, который побудил разработчиков создать механизмы внимания.

3.2. Улавливание зависимостей данных с помощью механизмов внимания

Хотя рекуррентные нейронные сети отлично подходят для перевода коротких предложений, они плохо справляются с длинными текстами, поскольку не имеют прямого доступа к предыдущим словам во входных данных. Одним из основных недостатков этого подхода является то, что RNN должна запоминать весь закодированный ввод в одном скрытом состоянии, прежде чем передать его декодировщику (см. рис. 3.4).

Это привело к тому, что в 2014 году исследователи разработали *механизм внимания по Богданову* (Bahdanau¹ attention) для рекуррентных нейронных сетей (названный в честь первого автора соответствующей статьи; дополнительную информацию см. в приложении Б). Данный механизм изменяет RNN архитектурой «кодировщик — декодировщик» таким образом, что декодировщик может выборочно обращаться к различным частям входной последовательности на каждом шаге декодирования (рис. 3.5).

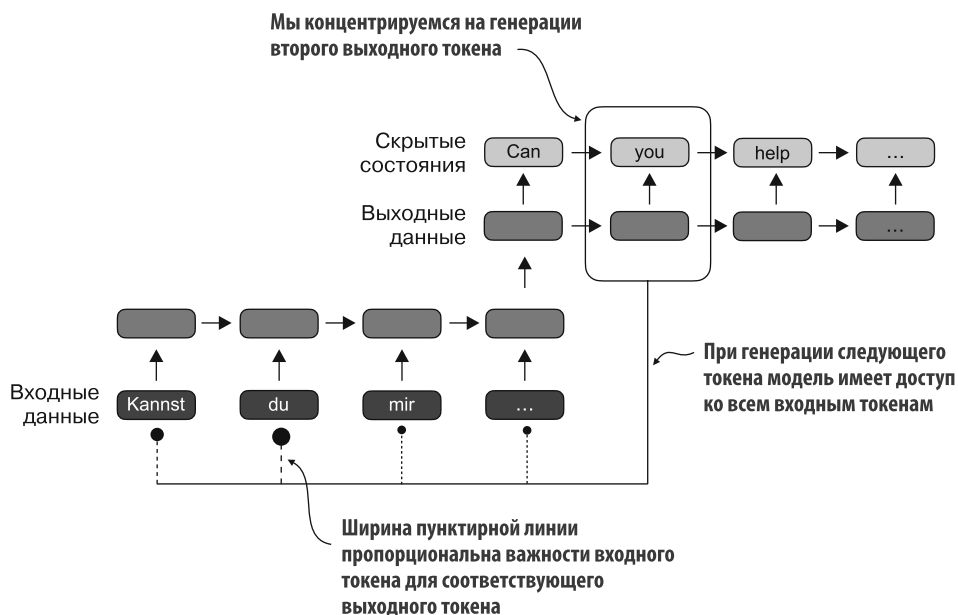


Рис. 3.5. Общая идея механизма внимания по Богданову (точная реализация выходит за рамки этой книги)

¹ Багданаў — это белорусский вариант написания фамилии Богданов. Транслитерация с белорусского на английский привела к Bahdanau. — *Примеч. пер.*

Как видите, с помощью механизма внимания декодирующая часть сети, генерирующая текст, может выборочно обращаться ко всем входным токенам. Это означает, что некоторые входные токены более важны для генерации конкретного выходного токена, чем другие. Важность определяется весами внимания, которые мы вычислим позже.

В 2017-м исследователи обнаружили, что для создания глубоких нейронных сетей для обработки естественного языка архитектуры RNN не требуются, и предложили архитектуру *трансформера* (которую мы обсуждали в главе 1), содержащую механизм самовнимания, по аналогии с механизмом внимания по Богданову.

Самовнимание — это механизм, который используется для вычисления более эффективных входных представлений. Он позволяет каждой позиции во входной последовательности взаимодействовать со всеми остальными позициями в той же последовательности и оценивать их вклад (важность) при вычислении представления последовательности. Самовнимание — ключевой компонент современных больших языковых моделей, основанных на архитектуре трансформера, таких как GPT.

В этой главе основное внимание мы уделим написанию кода механизма самовнимания, используемого в GPT-подобных моделях (рис. 3.6). Код для остальных частей LLM мы будем писать в следующей главе.

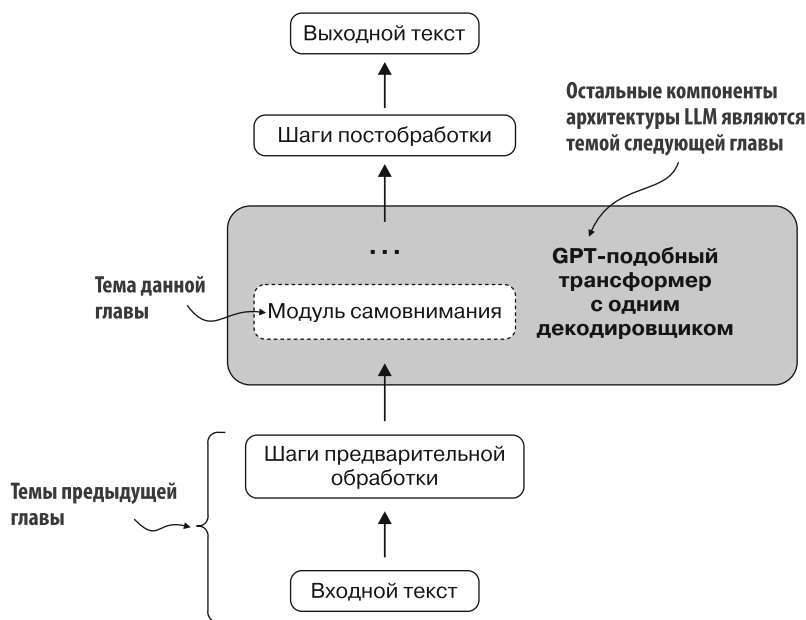


Рис. 3.6. Схема механизма самовнимания

3.3. Обращение к разным частям входных данных с помощью самовнимания

Далее мы рассмотрим внутреннюю работу механизма самовнимания и обсудим, как запрограммировать его с нуля. Как я уже сказал, данный механизм лежит в основе каждой большой языковой модели, основанной на архитектуре трансформера. Эта тема может потребовать много внимания, но, освоив ее азы, вы сможете понять один из самых сложных аспектов данной книги и реализации LLM в целом.

«Само» в самовнимании

Часть «само» в названии механизма относится к его способности вычислять весовые коэффициенты внимания, связывая различные позиции в пределах одной входной последовательности. Он оценивает и изучает взаимосвязи и зависимости между различными частями входных данных, такими как слова в предложении или пиксели на изображении.

Это отличается от традиционных механизмов внимания, которые фокусируются на взаимосвязях между элементами двух разных последовательностей, например, в моделях типа «последовательность — последовательность», где внимание может быть сосредоточено на взаимосвязи входной и выходной последовательностей, как показано на рис. 3.5.

Механизм самовнимания может показаться сложным, особенно если вы сталкиваетесь с ним впервые, поэтому мы начнем с его упрощенной версии. Затем реализуем механизм самовнимания с обучаемыми весами, используемый в LLM.

3.3.1. Простой механизм самовнимания без обучаемых весов

Начнем с реализации упрощенного варианта самовнимания без каких-либо обучаемых весов (рис. 3.7). Моя цель — продемонстрировать несколько ключевых концепций самовнимания, прежде чем переходить к варианту с обучаемыми весами.

Цель механизма самовнимания — вычислить контекстный вектор для каждого входного элемента, который объединяет информацию из всех остальных входных элементов. В данном примере мы вычисляем контекстный вектор $z^{(2)}$. Важность или вклад каждого входного элемента в вычислении $z^{(2)}$ определяется коэффициентами внимания от α_{21} до α_{2T} . При вычислении $z^{(2)}$ коэффициенты внимания рассчитываются относительно входного элемента $x^{(2)}$ и всех остальных входных данных.

На рис. 3.7 показана входная последовательность, обозначенная как x , состоящая из T элементов, представленных как $x^{(1)} - x^{(T)}$. Она обычно представляет

собой текст, например предложение, который уже был преобразован во вложенные токены.

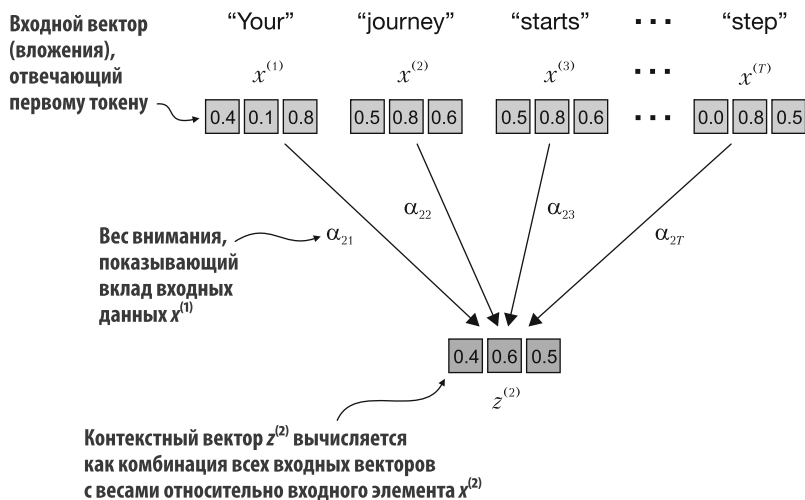


Рис. 3.7. Упрощенный механизм самовнимания

Входной текст в нашем примере — *Your journey starts with one step*. В этом случае каждый элемент последовательности, например $x^{(1)}$, соответствует d -мерному вектору вложения, представляющему конкретный токен, например *Your*. На рис. 3.7 эти входные векторы представлены в виде трехмерных векторов вложения.

При самовнимании наша цель — вычислить контекстные векторы $z^{(i)}$ для каждого элемента $x^{(i)}$ во входной последовательности. *Контекстный вектор* можно интерпретировать как расширенный вектор вложения.

Чтобы вы могли представить эту концепцию, сосредоточимся на векторе вложения второго входного элемента, $x^{(2)}$ (который соответствует токеноу *journey*), и соответствующем ему расширенном контекстном векторе, $z^{(2)}$, показанном в нижней части рис. 3.7. Этот вектор $z^{(2)}$ представляет собой вложение, содержащее информацию об $x^{(2)}$ и всех остальных входных элементах, от $x^{(1)}$ до $x^{(T)}$.

Контекстные векторы играют важнейшую роль в механизме самовнимания. Их цель — создавать расширенные представления каждого элемента во входной последовательности (например, предложения) путем объединения информации из всех остальных элементов последовательности (см. рис. 3.7). Это важно для больших языковых моделей, которым необходимо понимать взаимосвязь и значимость слов в предложении по отношению друг к другу. Позже мы добавим обучаемые весовые коэффициенты, которые помогут LLM научиться создавать

контекстные векторы, чтобы они были актуальны для генерации следующего токена. Но сначала реализуем упрощенный механизм самообучения, чтобы шаг за шагом вычислять эти веса и результирующий контекстный вектор.

Рассмотрим следующее исходное предложение, которое уже преобразовано в трехмерные векторы вложения (см. главу 2). Я выбрал небольшую размерность вложения, чтобы пример поместился на странице без разрывов строк:

```
import torch
inputs = torch.tensor(
    [[0.43, 0.15, 0.89], # Your      (x^1)
     [0.55, 0.87, 0.66], # journey  (x^2)
     [0.57, 0.85, 0.64], # starts   (x^3)
     [0.22, 0.58, 0.33], # with     (x^4)
     [0.77, 0.25, 0.10], # one      (x^5)
     [0.05, 0.80, 0.55]] # step      (x^6)
)
```

Первый шаг при реализации самовнимания — вычисление промежуточных значений ω , называемых *показателями внимания* (рис. 3.8). Из-за нехватки места на рисунке показаны значения предыдущего тензора `inputs` в усеченной версии; например, 0.87 усечено до 0.8. В этой версии вложения слов *journey* и *starts* могут показаться похожими по случайному совпадению.

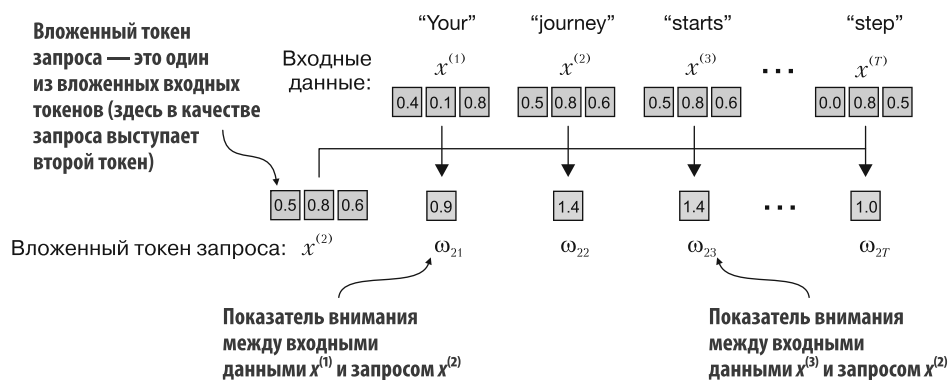


Рис. 3.8. Общая цель состоит в том, чтобы проиллюстрировать вычисление контекстного вектора $z^{(2)}$ с использованием второго входного элемента $x^{(2)}$ в качестве запроса. На этом рисунке показан первый промежуточный шаг — вычисление показателей внимания ω между запросом $x^{(2)}$ и всеми остальными входными элементами в виде скалярного произведения. (Обратите внимание, что числа округлены до одной цифры после запятой, чтобы улучшить визуальное восприятие.)

На рис. 3.8 показано, как мы вычисляем промежуточные показатели внимания между токеном запроса и каждым входным токеном. Мы определяем эти

показатели через скалярное произведение запроса $x^{(2)}$ и каждого другого входного токена:

```
query = inputs[1]
attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
    attn_scores_2[i] = torch.dot(x_i, query)
print(attn_scores_2)
```

← Второй входной токен
служит в качестве запроса

Вычисленные показатели внимания приведены ниже:

```
tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

Скалярное произведение

Скалярное произведение — это, по сути, краткий способ поэлементного перемножения двух векторов и последующего суммирования произведений, что можно продемонстрировать следующим образом:

```
res = 0.
for idx, element in enumerate(inputs[0]):
    res += inputs[0][idx] * query[idx]
print(res)
print(torch.dot(inputs[0], query))
```

Результат подтверждает, что сумма поэлементного перемножения дает те же результаты, что и скалярное произведение:

```
tensor(0.9544)
tensor(0.9544)
```

Скалярное произведение можно рассматривать как математический инструмент, объединяющий два вектора для получения скалярного значения. Кроме того, оно является мерой сходства, поскольку определяет, насколько близки друг к другу два вектора: чем больше скалярное произведение, тем выше степень сходства векторов. В контексте механизмов самоконтроля скалярное произведение определяет, в какой степени каждый элемент последовательности фокусируется на другом элементе или «обращает на него внимание»: чем больше скалярное произведение, тем выше степень сходства двух элементов и их внимания друг к другу.

На следующем шаге (рис. 3.9) мы нормализуем каждый из показателей внимания, которые вычислили ранее. Основная цель нормализации — получить веса внимания, сумма которых равна 1. Нормализация — это операция, полезная для интерпретации и поддержания стабильности обучения в LLM. Вот простой способ выполнить нормализацию:

```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()
print("Attention weights:", attn_weights_2_tmp)
print("Sum:", attn_weights_2_tmp.sum())
```

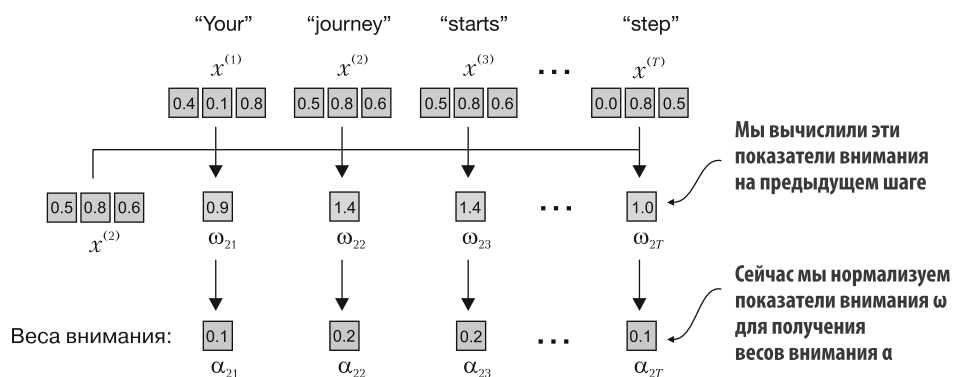


Рис. 3.9. После вычисления показателей внимания $\omega_{21} \dots \omega_{2T}$ по отношению к входному запросу $x^{(2)}$ следующим шагом является получение весов внимания $\alpha_{21} \dots \alpha_{2T}$ путем нормализации показателей внимания

Как показывает следующий результат, сумма весов внимания равна 1:

```
Attention weights: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656])
Sum: tensor(1.0000)
```

На практике более распространенным и целесообразным вариантом нормализации показателей внимания является использование функции `softmax`. Она лучше справляется с экстремальными значениями и обеспечивает более благоприятные свойства градиента во время обучения. Базовая реализация функции `softmax` приведена ниже:

```
def softmax_naive(x):
    return torch.exp(x) / torch.exp(x).sum(dim=0)

attn_weights_2_naive = softmax_naive(attn_scores_2)
print("Attention weights:", attn_weights_2_naive)
print("Sum:", attn_weights_2_naive.sum())
```

Как видно из результатов, функция `softmax` нормализует веса внимания таким образом, чтобы их сумма равнялась 1:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

Кроме того, функция `softmax` гарантирует, что весовые коэффициенты внимания всегда будут положительными. Это позволяет интерпретировать выходные значения как вероятности или относительную важность, где более высокие весовые коэффициенты указывают на большую важность.

Обратите внимание: при работе с большими или маленькими входными значениями в этой наивной реализации `softmax` (`softmax_naive`) могут возникать

проблемы с числовой нестабильностью, такие как переполнение и потеря значимости. Поэтому на практике рекомендуется использовать реализацию `softmax` в PyTorch, которая была всесторонне оптимизирована для повышения ее производительности:

```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
print("Attention weights:", attn_weights_2)
print("Sum:", attn_weights_2.sum())
```

В этом случае результат такой же, как и для нашей функции `softmax_naive`:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

Теперь, когда мы вычислили нормализованные веса внимания, мы готовы к последнему шагу — вычислению контекстного вектора $z^{(2)}$ путем умножения вложенных входных токенов $x^{(i)}$ на соответствующие веса внимания, а затем суммирования полученных векторов (рис. 3.10).

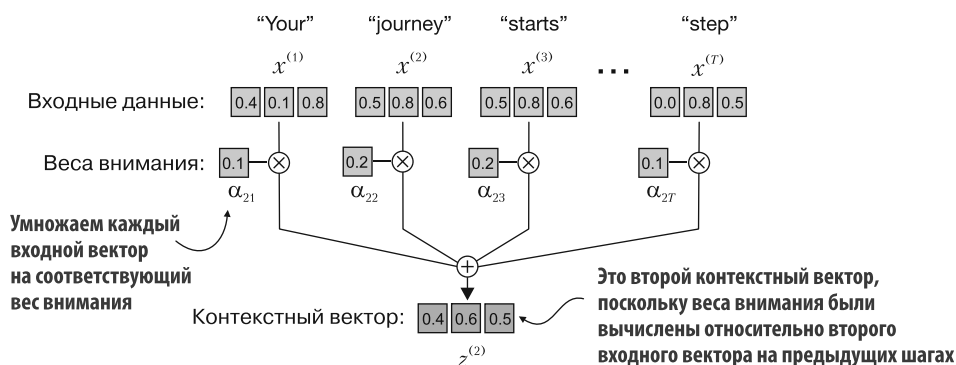


Рис. 3.10. Вычисление контекстного вектора $z^{(2)}$

Таким образом, контекстный вектор $z^{(2)}$ — это взвешенная сумма всех входных векторов от $x^{(1)}$ до $x^{(T)}$, полученная путем умножения каждого входного вектора на соответствующий вес внимания:

```
query = inputs[1]
context_vec_2 = torch.zeros(query.shape)
for i, x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

Второй входной токен является запросом

Результат выглядит так:

```
tensor([0.4419, 0.6515, 0.5683])
```

Далее мы обобщим эту процедуру вычисления контекстных векторов, чтобы рассчитать все контекстные векторы одновременно.

3.3.2. Вычисление весов внимания для всех входных токенов

До сих пор мы вычисляли весовые коэффициенты внимания и контекстный вектор для одного входного токена 2 (рис. 3.11).

	Your	journey	starts	with	one	step
Your	0.20	0.20	0.19	0.12	0.12	0.14
journey	0.13	0.23	0.23	0.12	0.10	0.15
starts	0.13	0.23	0.23	0.12	0.11	0.15
with	0.14	0.20	0.20	0.14	0.12	0.17
one	0.15	0.19	0.19	0.13	0.18	0.12
step	0.13	0.21	0.21	0.14	0.09	0.18

← Эта строка содержит веса внимания (нормализованные показатели внимания), вычисленные ранее

Рис. 3.11. Веса внимания для второго входного элемента указаны в выделенной строке. Обратите внимание: числа на этом рисунке округлены до двух знаков после запятой, чтобы упростить визуальное восприятие. Значения в каждой строке должны в сумме составлять 1 или 100 %

Теперь расширим эти вычисления, чтобы вычислить веса внимания и контекстные векторы для всех входных токенов. Мы выполняем те же три шага, что и раньше, за исключением того, что вносим несколько изменений в код для вычисления всех контекстных векторов, а не только второго, $z^{(2)}$ (рис. 3.12):

```
attn_scores = torch.empty(6, 6)
for i, x_i in enumerate(inputs):
    for j, x_j in enumerate(inputs):
        attn_scores[i, j] = torch.dot(x_i, x_j)
print(attn_scores)
```



Рис. 3.12. На шаге 1 мы добавляем дополнительный цикл for для вычисления скалярных произведений для всех пар входных данных

Окончательные показатели внимания выглядят так:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

Каждый элемент в тензоре представляет собой показатели внимания между каждой парой входных данных, как мы видели на рис. 3.11. Обратите внимание: значения на этом рисунке нормализованы, поэтому отличаются от ненормированных показателей внимания предыдущего тензора. О нормализации мы позаботимся позже.

При вычислении предыдущего тензора показателей внимания мы использовали циклы `for` в Python. Однако эти циклы обычно работают медленно, и мы можем добиться тех же результатов, умножая матрицы:

```
attn_scores = inputs @ inputs.T
print(attn_scores)
```

Мы видим, что результаты совпадают с теми, которые мы получили ранее:

```
tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
        [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
        [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
        [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
        [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
        [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])
```

На шаге 2 (см. рис. 3.12) мы нормализуем каждую строку так, чтобы сумма значений в строке равнялась 1:

```
attn_weights = torch.softmax(attn_scores, dim=-1)
print(attn_weights)
```

Получаем следующий тензор весов внимания со значениями, идентичными показанным на рис. 3.10:

```
tensor([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452],
        [0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581],
        [0.1390, 0.2369, 0.2326, 0.1242, 0.1108, 0.1565],
        [0.1435, 0.2074, 0.2046, 0.1462, 0.1263, 0.1720],
        [0.1526, 0.1958, 0.1975, 0.1367, 0.1879, 0.1295],
        [0.1385, 0.2184, 0.2128, 0.1420, 0.0988, 0.1896]])
```

В контексте использования PyTorch параметр `dim` в функциях, таких как `torch.softmax`, указывает размерность входного тензора, по которой будет вычисляться функция. Установив `dim=-1`, мы указываем функции `softmax` применять нормализацию по последней размерности тензора `attn_scores`. Если `attn_scores` — это двумерный тензор (например, в форме [строки, столбцы]), то он

будет нормализован по столбцам таким образом, чтобы значения в каждой строке (при суммировании по столбцам) в сумме составляли 1.

Мы можем убедиться, что все строки действительно в сумме дают 1:

```
row_2_sum = sum([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
print("Row 2 sum:", row_2_sum)
print("All row sums:", attn_weights.sum(dim=-1))
```

Результат выглядит так:

```
Row 2 sum: 1.0
All row sums: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

На шаге 3, финальном (см. рис. 3.12 выше), мы используем эти веса внимания для вычисления всех контекстных векторов через перемножение матриц:

```
all_context_vecs = attn_weights @ inputs
print(all_context_vecs)
```

В результирующем тензоре каждая строка содержит трехмерный контекстный вектор:

```
tensor([[0.4421, 0.5931, 0.5790],
        [0.4419, 0.6515, 0.5683],
        [0.4431, 0.6496, 0.5671],
        [0.4304, 0.6298, 0.5510],
        [0.4671, 0.5910, 0.5266],
        [0.4177, 0.6503, 0.5645]])
```

Мы можем перепроверить правильность кода, сравнив вторую строку с контекстным вектором $z^{(2)}$, который вычислили в подразделе 3.3.1:

```
print("Previous 2nd context vector:", context_vec_2)
```

Исходя из результата, мы видим, что ранее рассчитанный `context_vec_2` точно соответствует второй строке в предыдущем тензоре:

```
Previous 2nd context vector: tensor([0.4419, 0.6515, 0.5683])
```

На этом мы завершаем обзор кода простого механизма самовнимания. Далее мы добавим обучаемые весовые коэффициенты, что позволит LLM обучаться на данных и повышать свою эффективность при выполнении конкретных задач.

3.4. Реализация самовнимания с обучаемыми весами

Нашим следующим шагом будет реализация механизма самовнимания, используемого в первоначальной архитектуре трансформера, моделях GPT и большинстве других популярных больших языковых моделей. Этот механизм также называется *вниманием с масштабированным скалярным произведением (scaled dot-product attention)*. На рис. 3.13 показано, как он вписывается в более широкий контекст реализации LLM.

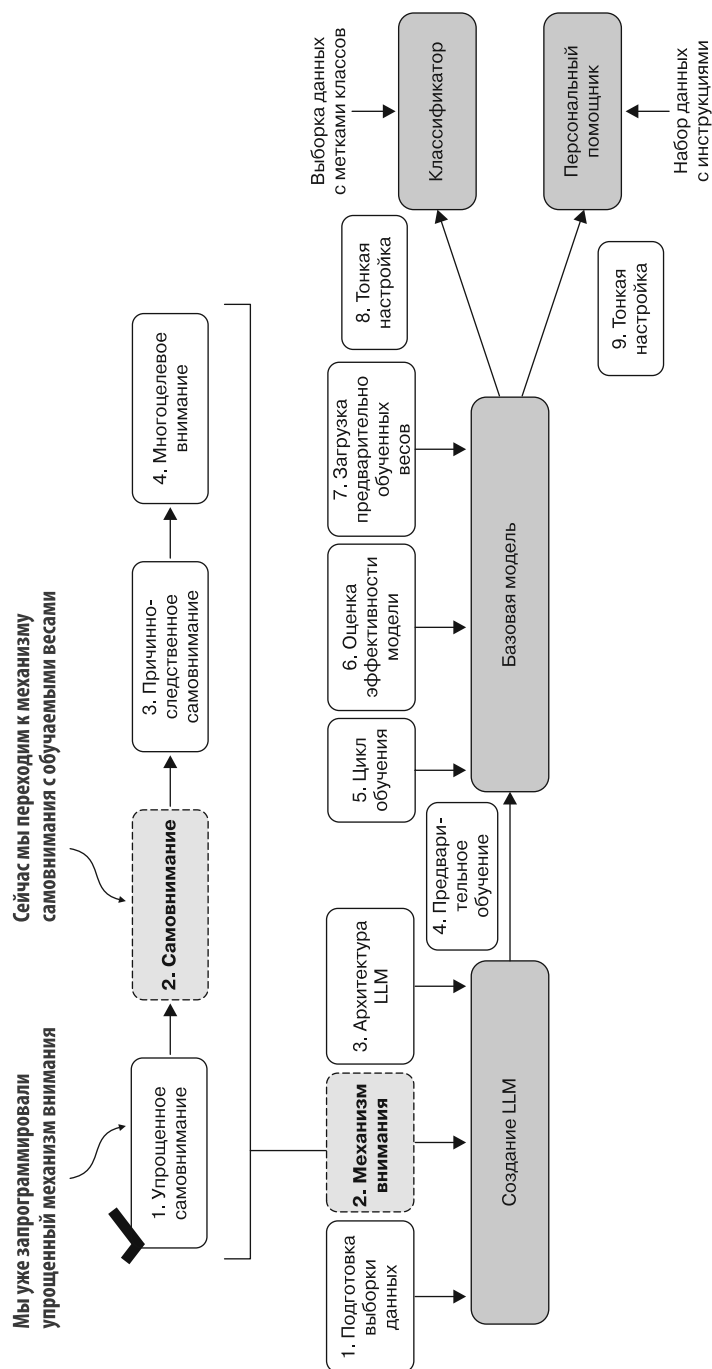


Рис. 3.13. Ранее мы создали упрощенный механизм внимания для понимания принципа его работы. Теперь мы добавим к нему обучаемые веса. Позже мы расширим этот механизм самовнимания, добавив причинно-следственную маску и механизм нескольких путей внимания

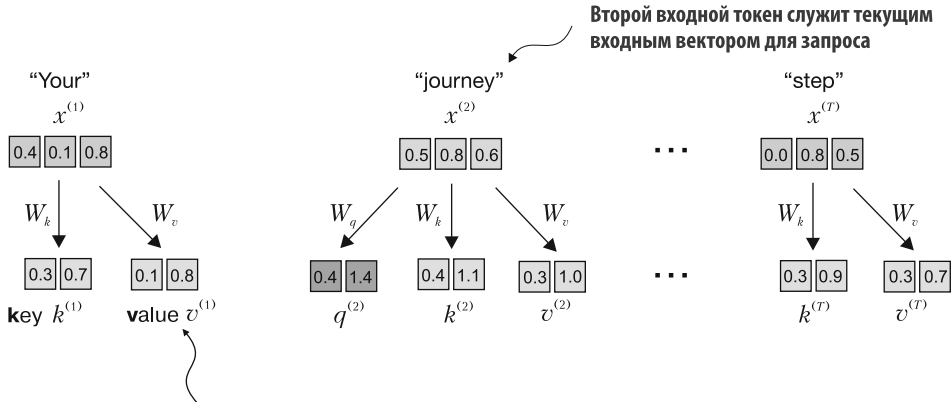
Как показано на рис. 3.13, механизм самовнимания с обучаемыми весами основан на предыдущих концепциях: мы хотим вычислять контекстные векторы как взвешенные суммы входных векторов, относящихся к определенному входному элементу. Как вы увидите, есть лишь небольшие различия по сравнению с базовым механизмом самовнимания, который мы запрограммировали ранее.

Наиболее заметное различие заключается во введении весовых матриц, которые обновляются во время обучения модели. Они имеют решающее значение для того, чтобы модель (в частности, модуль внимания внутри нее) могла научиться создавать «хорошие» контекстные векторы. (Обучать LLM мы будем в главе 5.)

Мы рассмотрим этот механизм самовнимания в двух следующих подразделах. Сначала мы шаг за шагом напишем код, как и раньше. Затем организуем его в компактный класс Python, который можно импортировать в архитектуру LLM.

3.4.1. Пошаговое вычисление весовых коэффициентов внимания

Мы реализуем механизм самовнимания пошагово, вводя три обучаемые весовые матрицы W_q , W_k и W_v . Они используются для проецирования вложенных входных токенов $x^{(i)}$ на векторы запроса, векторы ключей и векторы значений соответственно (рис. 3.14).



Это вектор значений, соответствующий первому входному токенту, полученный путем перемножения весовой матрицы W_v и входного токена $x^{(1)}$

Рис. 3.14. Три обучаемые весовые матрицы W_q , W_k и W_v , используемые для проецирования на три вектора

На первом шаге механизма самовнимания с обучаемыми весовыми матрицами мы вычисляем векторы запроса (q), векторы ключей (k) и векторы значений (v) для входных элементов x . Как и в предыдущих разделах, мы обозначаем второй входной элемент $x^{(2)}$ как запрос. Вектор запроса $q^{(2)}$ получается путем умножения

входного элемента $x^{(2)}$ на весовую матрицу W_q . Аналогичным образом мы получаем векторы ключей и значений, умножая входной элемент $x^{(2)}$ на весовые матрицы W_k и W_v .

Ранее мы определили второй входной элемент $x^{(2)}$ в качестве запроса, когда вычисляли упрощенные веса внимания и контекстного вектора $z^{(2)}$. Затем мы распространили данный подход на вычисление всех контекстных векторов $z^{(1)} \dots z^{(T)}$ для входного предложения из шести слов *Your journey starts with one step*.

Аналогичным образом мы начнем с вычисления только одного вектора, $z^{(2)}$, в целях демонстрации. Затем изменим этот код, чтобы вычислить все векторы.

Начнем с определения нескольких переменных:

```
x_2 = inputs[1]  ← Второй входной элемент
d_in = inputs.shape[1]  ← Размерность входного вложения, d=3
d_out = 2  ← Размерность выходного вложения
```

Обратите внимание, что в моделях, подобных GPT, входные и выходные размерности обычно совпадают, но для того, чтобы вы лучше поняли вычисления, мы будем использовать здесь разные входные ($d_{in}=3$) и выходные ($d_{out}=2$) значения.

Далее мы инициализируем три весовые матрицы W_q , W_k и W_v , показанные на рис. 3.14:

```
torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_key = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
```

Мы устанавливаем значение `requires_grad=False`, чтобы уменьшить количество выходных данных, но если бы мы использовали весовые матрицы для обучения модели, то установили бы значение `requires_grad=True`, чтобы обновлять эти матрицы во время обучения LLM.

Далее вычисляем векторы запроса, векторы ключей и векторы значений:

```
query_2 = x_2 @ W_query
key_2 = x_2 @ W_key
value_2 = x_2 @ W_value
print(query_2)
```

Результатом запроса является двумерный вектор, поскольку мы установили количество столбцов, d_{out} , соответствующей весовой матрицы равным 2:

```
tensor([0.4306, 1.4551])
```

Наша промежуточная цель — вычислить только один контекстный вектор $z^{(2)}$, однако нам все равно нужны векторы ключей и значений для всех входных элементов, поскольку они участвуют в вычислении весов внимания по отношению к запросу $q^{(2)}$ (см. рис. 3.14).

Весовые параметры в сравнении с весами внимания

В весовых матрицах W термин «вес» является сокращением от «весовых параметров» — значений нейронной сети, которые оптимизируются во время обучения. Его не следует путать с весами внимания. Как вы уже видели, веса внимания определяют степень, в которой контекстный вектор зависит от различных частей входных данных (то есть в какой степени сеть фокусируется на различных частях входных данных).

Таким образом, весовые параметры — это фундаментальные, полученные в результате обучения коэффициенты, которые определяют связи в сети, а веса внимания — это динамические значения, зависящие от контекста.

Мы можем получить все ключи и значения с помощью перемножения матриц:

```
keys = inputs @ W_key
values = inputs @ W_value
print("keys.shape:", keys.shape)
print("values.shape:", values.shape)
```

Как видно из результатов печати, мы успешно спроецировали шесть входных токенов из трехмерного пространства в двумерное:

```
keys.shape: torch.Size([6, 2])
values.shape: torch.Size([6, 2])
```

Второй шаг — вычислить показатели внимания (рис. 3.15).

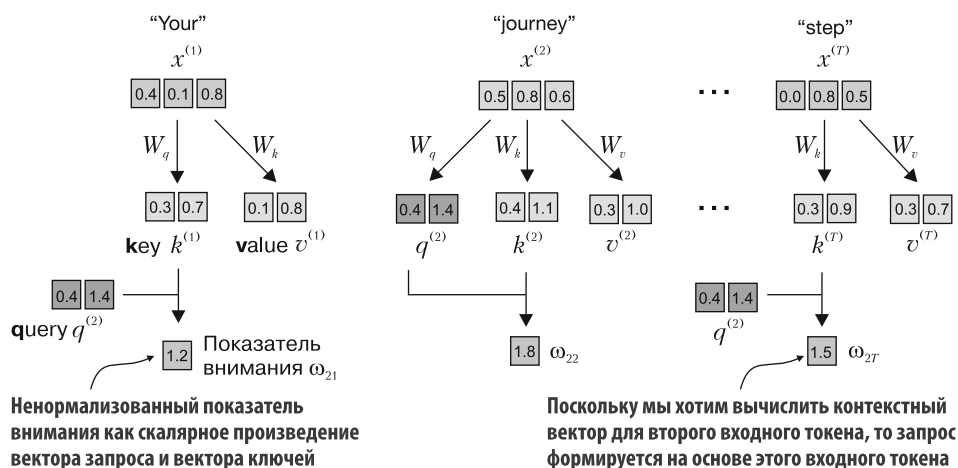


Рис. 3.15. Вычисление показателя внимания — это вычисление скалярного произведения, аналогичное тому, что мы использовали в упрощенном механизме самовнимания в разделе 3.3. Новым аспектом здесь является то, что мы вычисляем скалярное произведение между входными элементами не напрямую, а с помощью запроса и ключа, полученных путем преобразования входных данных с помощью соответствующих весовых матриц

Сначала вычислим показатель внимания ω_{22} :

```
keys_2 = keys[1]
attn_score_22 = query_2.dot(keys_2)
print(attn_score_22)
```

← Вспомним, что в Python индексация начинается с нуля

Результат для ненормализованного показателя внимания таков:

```
tensor(1.8524)
```

Опять же, мы можем обобщить это вычисление для всех показателей внимания, используя перемножение матриц:

```
attn_scores_2 = query_2 @ keys.T
print(attn_scores_2)
```

← Все показатели внимания для данного запроса

Как мы видим по результату, второй элемент совпадает с `attn_score_22`, который мы вычислили ранее:

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

Теперь мы хотим перейти от показателей внимания к весам внимания (рис. 3.16). Мы вычисляем эти веса, масштабируя показатели внимания и используя функцию `softmax`. Однако теперь мы масштабируем показатели, деля их на квадратный корень из размерности вложения ключей (извлечение квадратного корня математически эквивалентно возведению в степень 0,5):

```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

Результирующие значения весов таковы:

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

Обоснование масштабированного скалярного произведения в механизме внимания

Нормализация с помощью размерности вектора вложения необходима для повышения эффективности обучения за счет устранения малых градиентов. Например, при таком масштабировании, где данная размерность обычно превышает 1000 для LLM-подобных моделей, большие скалярные произведения могут приводить к очень малым градиентам при обратном распространении ошибки из-за применяемой к ним функции `softmax`. По мере увеличения скалярных произведений данная функция становится более похожей на ступенчатую, в результате чего градиенты приближаются к нулю. Эти небольшие по величине градиенты могут значительно замедлить обучение или привести к его остановке.

Масштабирование на квадратный корень из размерности вложения — причина, по которой этот механизм самовнимания также называют вниманием с масштабированным скалярным произведением.

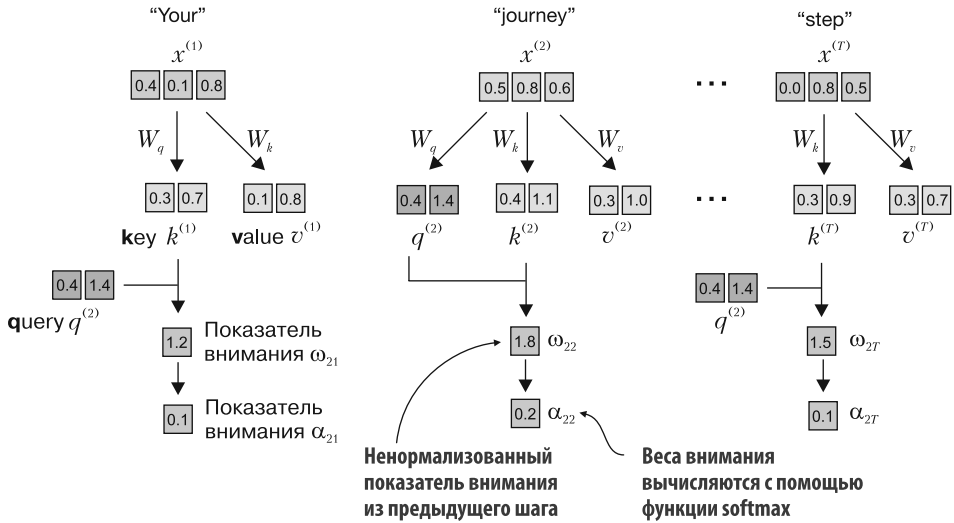


Рис. 3.16. После вычисления показателей внимания ω следующим шагом является их нормализация с помощью функции softmax для получения весов внимания α

Последний шаг — вычислить контекстные векторы (рис. 3.17).

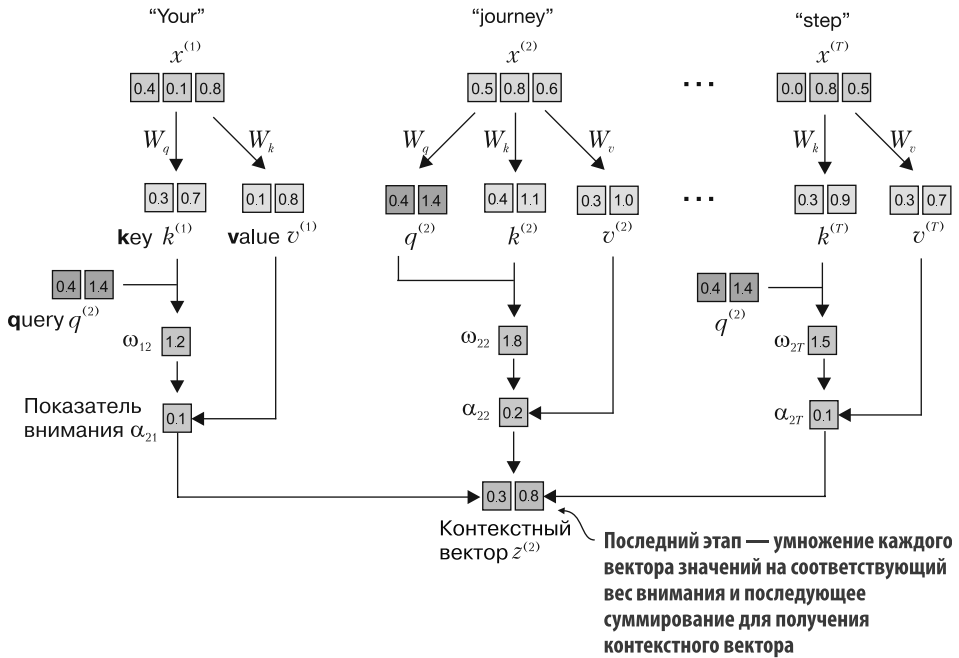


Рис. 3.17. Вычисление контекстного вектора

Подобно тому как мы вычисляли контекстный вектор как взвешенную сумму входных векторов (см. раздел 3.3), теперь мы вычисляем контекстный вектор как взвешенную сумму векторов значений. Здесь веса внимания служат в качестве весового коэффициента, который определяет важность каждого вектора значений. Как и прежде, мы можем использовать перемножение матриц, чтобы получить результат всего за один шаг:

```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

Содержимое результирующего вектора выглядит так:

```
tensor([0.3061, 0.8210])
```

Пока мы вычислили только один контекстный вектор, $z^{(2)}$. Далее мы обобщим этот код для вычисления всех контекстных векторов во входной последовательности, от $z^{(1)}$ до $z^{(T)}$.

Почему запрос, ключ и значение

Термины «ключ», «запрос» и «значение» в контексте механизмов внимания заимствованы из области поиска информации и баз данных, где аналогичные понятия используются для хранения, поиска и извлечения информации.

Запрос аналогичен поисковому запросу в базе данных. Он представляет собой текущий элемент (например, слово или токен в предложении), на котором модель фокусируется или который пытается понять.

Запрос используется для изучения других частей входной последовательности, чтобы определить, какая степень внимания им требуется.

Ключ похож на ключ базы данных, используемый для индексирования и поиска. В механизме внимания каждый элемент входной последовательности (например, каждое слово в предложении) имеет связанный с ним ключ. Эти ключи используются для сопоставления с запросом.

Значение в этом контексте похоже на значение в паре «ключ — значение» в базе данных. Оно представляет фактическое содержимое или представление входных элементов. Определив, какие ключи (и, следовательно, какие части входных данных) наиболее подходят для запроса (текущего элемента в фокусе), модель извлекает соответствующие значения.

3.4.2. Реализация компактного класса Python для самовнимания

К настоящему моменту мы проделали множество шагов, чтобы вычислить результаты самовнимания. Мы сделали это в основном для наглядности, чтобы можно было переходить от одного шага к другому. На практике, учитывая

реализацию LLM в следующей главе, полезно организовать этот код в класс Python (листинг 3.1).

Листинг 3.1. Компактный класс самовнимания

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))

    def forward(self, x):
        keys = x @ self.W_key
        queries = x @ self.W_query
        values = x @ self.W_value
        attn_scores = queries @ keys.T # omega
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        context_vec = attn_weights @ values
        return context_vec
```

В этом коде PyTorch `SelfAttention_v1` — это класс, производный от `nn.Module`, который является основным строительным блоком моделей PyTorch и предоставляет функции, необходимые для создания и управления слоями модели.

Метод `__init__` инициализирует обучаемые весовые матрицы (`W_query`, `W_key` и `W_value`) для запросов, ключей и значений, каждая из которых преобразует входное измерение `d_in` в выходное измерение `d_out`.

Во время прямого прохода по выборке данных, используя метод `forward`, мы вычисляем показатели внимания (`attn_scores`), умножая запросы и ключи и нормализуя показатели с помощью `softmax`. Наконец, мы создаем контекстный вектор, взвешивая значения с помощью этих нормализованных показателей.

Мы можем использовать этот класс следующим образом:

```
torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
```

Поскольку `inputs` содержат шесть векторов вложений, в результате получается матрица, в которой хранятся шесть контекстных векторов:

```
tensor([[0.2996, 0.8053],
        [0.3061, 0.8210],
        [0.3058, 0.8203],
        [0.2948, 0.7939],
        [0.2927, 0.7891],
        [0.2990, 0.8040]], grad_fn=<MmBackward0>)
```

Обратите внимание: вторая строка $[0, 3061, 0, 8210]$ соответствует содержанию `context_vec_2` из предыдущего раздела. На рис. 3.18 представлен механизм самовнимания, который мы только что реализовали.

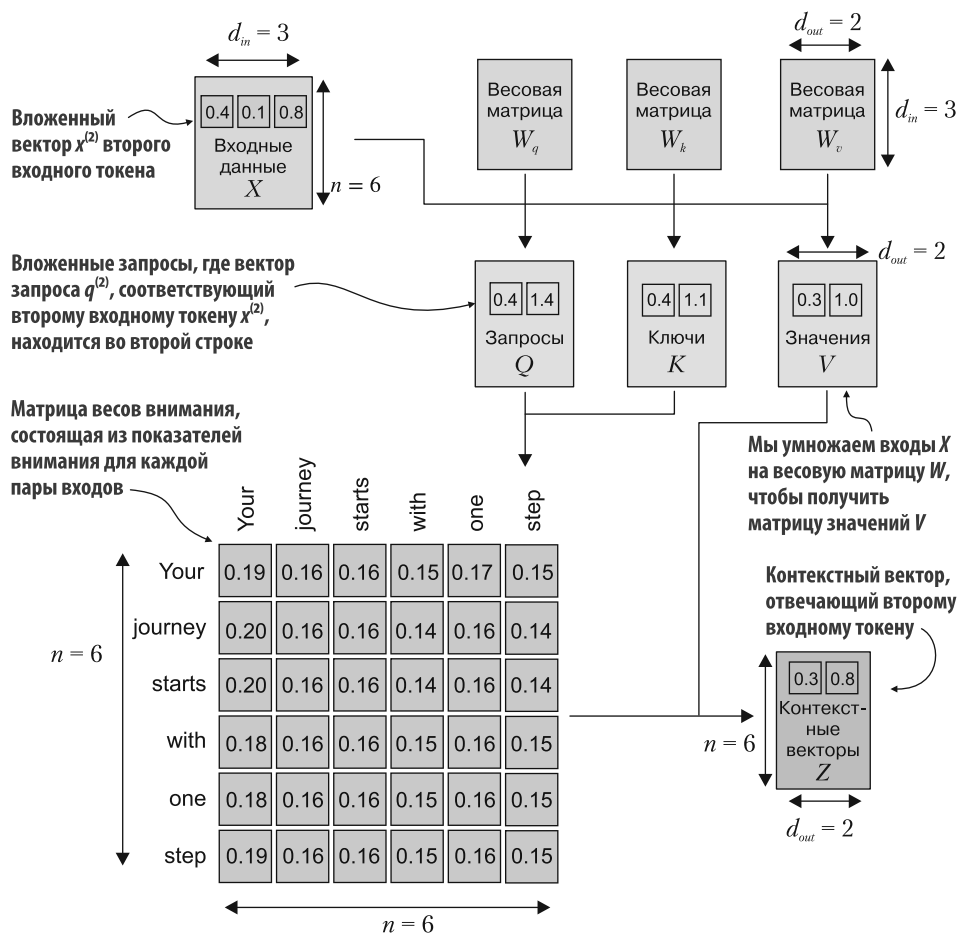


Рис. 3.18. Механизм самовнимания

Как видите, в самовнимании мы преобразуем входные векторы во входной матрице X с помощью трех весовых матриц: W_q , W_k и W_v . Новая матрица весов внимания вычисляется на основе полученных запросов (Q) и ключей (K). Затем с помощью весов внимания и значений (V) вычисляются контекстные векторы (Z). Для наглядности мы рассматриваем один входной текст с n токенами, а не пакет из нескольких входных данных. Следовательно, в этом контексте трехмерный входной тензор упрощается до двумерной матрицы. Такой подход позволяет более наглядно визуализировать и лучше понять задействованные

процессы. Для согласованности с последующими рисунками значения в матрице внимания не отображают реальные веса внимания. (Числа на рис. 3.18 округлены до двух знаков после запятой, чтобы упростить визуальное восприятие. Значения в каждой строке должны в сумме составлять 1 или 100 %.)

Итак, самовнимание включает в себя обучаемые весовые матрицы W_q , W_k и W_v . Они преобразуют входные данные в запросы, ключи и значения соответственно, которые являются важнейшими компонентами механизма внимания. По мере того как модель получает больше данных во время обучения, она корректирует эти обучаемые веса, как мы увидим в следующих главах.

Мы можем улучшить реализацию `SelfAttention_v1`, используя линейные слои PyTorch `nn.Linear` (листинг 3.2), которые эффективно выполняют перемножение матриц, когда смещение (`bias`) отключено. Кроме того, важное преимущество использования `nn.Linear` вместо ручной реализации `nn.Parameter(torch.rand(...))` состоит в том, что `nn.Linear` имеет оптимизированную схему инициализации весов, что способствует более стабильному и эффективному обучению модели.

Листинг 3.2. Класс самовнимания, использующий слой `Linear` из PyTorch

```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        context_vec = attn_weights @ values
        return context_vec
```

Вы можете использовать `SelfAttention_v2` по аналогии с `SelfAttention_v1`:

```
torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

Результат выглядит так:

```
tensor([[ -0.0739,  0.0713],
        [ -0.0748,  0.0703],
        [ -0.0749,  0.0702],
        [ -0.0760,  0.0685],
        [ -0.0763,  0.0679],
        [ -0.0754,  0.0693]], grad_fn=<MmBackward0>)
```

Обратите внимание: `SelfAttention_v1` и `SelfAttention_v2` дают разные результаты, поскольку используют разные начальные веса для весовых матриц, так как `nn.Linear` задействует более сложную схему инициализации весов.

Упражнение 3.1. Сравнение `SelfAttention_v1` и `SelfAttention_v2`

Обратите внимание, что `nn.Linear` в `SelfAttention_v2` использует другую схему инициализации весов, чем `nn.Parameter(torch.rand(d_in, d_out))` в `SelfAttention_v1`. Это приводит к тому, что оба механизма дают разные результаты. Проверить, что обе реализации, `SelfAttention_v1` и `SelfAttention_v2`, в остальном похожи, можно так: перенести весовые матрицы из объекта `SelfAttention_v2` в объект `SelfAttention_v1`, чтобы оба объекта давали одинаковые результаты.

Ваша задача — правильно присвоить веса экземпляру `SelfAttention_v1` на основании экземпляра `SelfAttention_v2`. Для этого вам нужно понять взаимосвязь между весами в обеих версиях. (Подсказка: `nn.Linear` хранит весовую матрицу в транспонированном виде.) После присвоения вы должны заметить, что оба экземпляра выдают одинаковые результаты.

Далее мы усовершенствуем механизм самовнимания, сосредоточившись на добавлении причинно-следственных и многоцелевых элементов.

Причинно-следственный аспект предполагает изменение механизма внимания, чтобы модель не могла получить доступ к будущей информации в последовательности, что крайне важно для таких задач, как языковое моделирование, где предсказывание каждого слова должно зависеть только от предыдущих слов.

Многоцелевой компонент предполагает разделение механизма внимания на несколько «целей». Каждая цель изучает различные аспекты данных, что позволяет LLM одновременно учитывать информацию из разных подпространств представления в разных позициях. Это повышает эффективность модели при выполнении сложных задач.

3.5. Соккрытие будущих слов с помощью причинно-следственного внимания

Для многих задач LLM будет нужно, чтобы механизм самовнимания учитывал только токены, которые появляются до текущей позиции при предсказывании следующего токена в последовательности. Причинно-следственное внимание, также известное как *скрытое внимание*, — специализированная форма самовнимания. Оно ограничивает модель, побуждая ее рассматривать только предыдущие и текущие входные данные в последовательности при обработке любого заданного токена, выполняемой для вычисления показателей внимания.

Такое поведение отличается от стандартного механизма самовнимания, который позволяет получить доступ ко всей входной последовательности одновременно.

Теперь мы изменим стандартный механизм самовнимания, чтобы создать механизм *причинно-следственного внимания*, который необходим для разработки LLM в последующих главах. Чтобы добиться этого в GPT-подобных LLM, для каждого обрабатываемого токена мы маскируем будущие токены, которые следуют за текущим токеном во входном тексте (рис. 3.19). Мы маскируем веса внимания выше диагонали и нормализуем незамаскированные веса внимания так, чтобы сумма весов внимания в каждой строке равнялась 1. Позже мы реализуем эту процедуру маскирования и нормализации в коде.

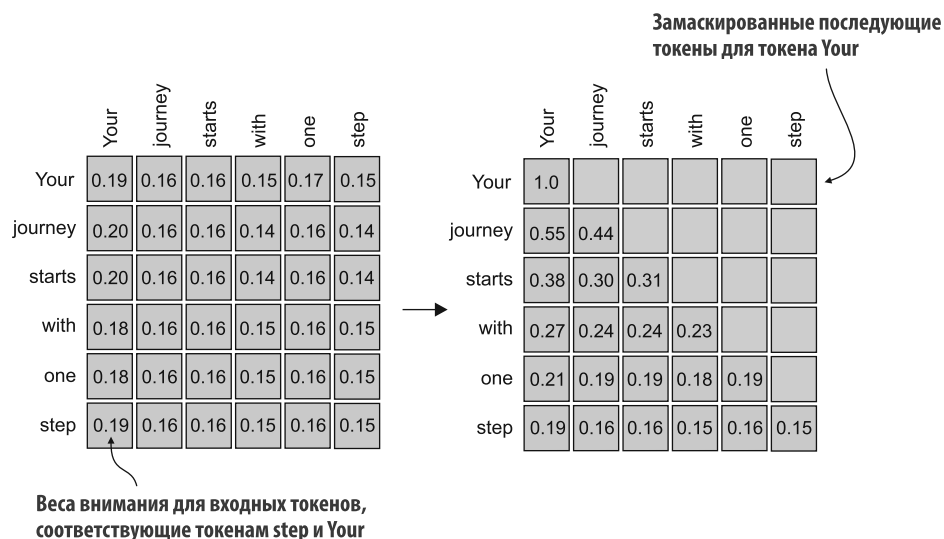


Рис. 3.19. В причинно-следственном внимании мы маскируем веса внимания выше диагонали так, чтобы при заданных входных данных LLM не могла получить доступ к будущим токенам при вычислении контекстных векторов с помощью весов внимания. Например, для слова *journey* во втором ряду мы сохраняем только веса внимания для слов, стоящих перед ним (*Your*) и в текущей позиции (*journey*)

3.5.1. Применение маски причинно-следственного внимания

Следующий шаг — реализовать маску причинно-следственного внимания в коде. На рис. 3.20 показаны действия по применению маски причинно-следственного внимания. Чтобы выполнить их, воспользуемся показателями и весами внимания из предыдущего раздела.



Рис. 3.20. Один из способов получения маскированной матрицы весов внимания в причинно-следственном внимании — применить функцию `softmax` к показателям внимания, обнулить элементы над диагональю и нормализовать полученную матрицу

На первом шаге мы вычисляем веса внимания с помощью функции `softmax`, как делали ранее:

```
queries = sa_v2.W_query(inputs)
keys = sa_v2.W_key(inputs)
attn_scores = queries @ keys.T
attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
print(attn_weights)
```

← Для удобства повторно используются матрицы запросов и весов объекта `SelfAttention_v2` из предыдущего раздела

Это приводит к следующему результату:

```
tensor([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510],
        [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
        [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.1585],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
        grad_fn=<SoftmaxBackward0>)
```

Мы можем реализовать второй шаг, используя функцию `tril` из PyTorch для создания маски, где значения выше диагонали — нули:

```
context_length = attn_scores.shape[0]
mask_simple = torch.tril(torch.ones(context_length, context_length))
print(mask_simple)
```

Маска выглядит так:

```
tensor([[1., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0.],
        [1., 1., 1., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1., 1.]])
```

Теперь мы можем умножить данную маску на веса внимания, чтобы обнулить значения выше диагонали весовой матрицы:

```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

Как мы видим, элементы выше диагонали действительно стали нулями:

```
tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<MulBackward0>)
```

На третьем шаге веса внимания нормализуются снова, чтобы сумма значений в каждой строке равнялась 1. Мы достигаем этого, деля каждое значение в каждой строке на сумму значений в строке:

```
row_sums = masked_simple.sum(dim=-1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
```

Получается матрица весов внимания, в которой значения выше диагонали обнулены и сумма по строкам равна 1:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
       grad_fn=<DivBackward0>)
```

Утечка информации

Когда мы применяем маску, а затем повторно нормализуем веса внимания, то может показаться, что информация из будущих токенов (которые мы собираемся замаскировать) все равно может повлиять на текущий токен, поскольку их значения являются частью вычислений softmax. Однако ключевая идея заключается в том, что когда мы повторно нормализуем веса внимания после маскирования, то, по сути, мы пересчитываем softmax для меньшего подмножества значений (поскольку замаскированные позиции не влияют на значение softmax).

Математическая элегантность softmax состоит в том, что хоть изначально все позиции и учитываются в знаменателе, после маскирования и повторной нормализации эффект от замаскированных позиций сводится к нулю — они не влияют на значение softmax каким-либо значимым образом.

Проще говоря, после маскирования и повторной нормализации распределение весов внимания выглядит так, как если бы изначально рассчитывалось только для незамаскированных позиций. Это гарантирует отсутствие утечки информации из будущих (замаскированных) токенов, как мы и планировали.

На данном этапе мы могли бы завершить реализацию причинно-следственного внимания, но все еще можем улучшить его. Воспользуемся математическим свойством функции `softmax` и реализуем вычисление весов замаскированного внимания более эффективно, в несколько шагов (рис. 3.21).

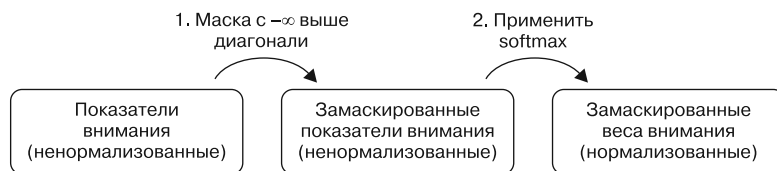


Рис. 3.21. Более эффективный способ получить весовую матрицу внимания с маскировкой в механизме причинно-следственного внимания — маскировать оценки внимания отрицательными значениями бесконечности перед применением функции `softmax`

Функция `softmax` преобразует входные данные в распределение вероятностей. Если в строке присутствуют значения отрицательной бесконечности ($-\infty$), то функция `softmax` рассматривает их как нулевую вероятность. (С математической точки зрения это происходит потому, что $e^{-\infty}$ приближается к 0.)

Мы можем реализовать этот более эффективный «трюк» с маскировкой, создав маску с единицами над диагональю, а затем заменив их значениями отрицательной бесконечности ($-\text{inf}$):

```
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
```

В результате получаем следующую маску:

```
tensor([[0.2899,  -inf,  -inf,  -inf,  -inf,  -inf],
        [0.4656,  0.1723,  -inf,  -inf,  -inf,  -inf],
        [0.4594,  0.1703,  0.1731,  -inf,  -inf,  -inf],
        [0.2642,  0.1024,  0.1036,  0.0186,  -inf,  -inf],
        [0.2183,  0.0874,  0.0882,  0.0177,  0.0786,  -inf],
        [0.3408,  0.1270,  0.1290,  0.0198,  0.1290,  0.0078]],
      grad_fn=<MaskedFillBackward0>)
```

Теперь все, что нужно сделать, — применить `softmax` к этим замаскированным значениям, и мы у цели:

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
print(attn_weights)
```

Глядя на результат выше, можно увидеть, что сумма значений в каждой строке равна 1 и дополнительной нормализации не требуется:

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
        [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
        [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
        grad_fn=<SoftmaxBackward0>)
```

Теперь мы можем использовать измененные веса внимания для вычисления контекстных векторов с помощью `context_vec = attn_weights @ values`, как в разделе 3.4. Однако сначала мы рассмотрим еще одну незначительную модификацию механизма причинно-следственного внимания, которая может помочь уменьшить переобучение больших языковых моделей.

3.5.2. Маскирование дополнительных весов внимания с помощью отсева

Отсев (dropout) в глубоком обучении — это метод, при котором случайно выбранные элементы скрытого слоя игнорируются во время обучения, фактически «будучи отсеянными». Данный метод помогает предотвратить переобучение, гарантируя, что модель не будет чрезмерно полагаться на какой-либо конкретный набор элементов скрытого слоя. Важно подчеркнуть, что отсев используется только во время обучения и отключается после него.

В архитектуре трансформера, в том числе таких моделей, как GPT, отсев в механизме внимания обычно применяется в двух случаях: после вычисления весов внимания или после применения весов внимания к векторам значений. Здесь мы применим маску отсева после вычисления весов внимания (рис. 3.22), поскольку на практике это более распространенный вариант.

В следующем примере кода мы используем коэффициент отсева 50 %, что означает маскировку половины весовых коэффициентов внимания. (При обучении модели GPT в следующих главах мы будем использовать более низкую частоту отсева, например 0,1 или 0,2.) Для простоты мы сначала применим функцию отсева в PyTorch к тензору 6×6 , состоящему из единиц:

```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5)
example = torch.ones(6, 6)
print(dropout(example))
```

← Выбираем коэффициент отсева, равный 50 %

← Здесь мы создаем матрицу, состоящую из единиц

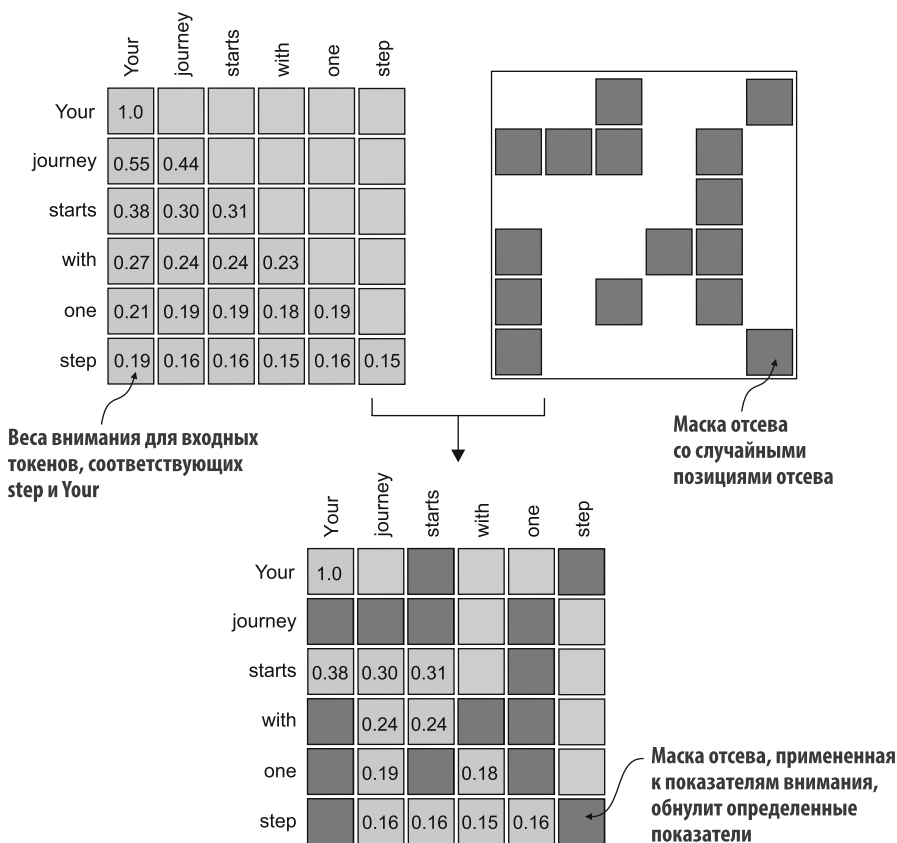


Рис. 3.22. Используя маску причинно-следственного внимания (вверху слева), мы применяем дополнительную маску отсева (вверху справа), чтобы обнулить дополнительные веса внимания и уменьшить переобучение

Как мы видим, примерно половина значений обнуляется:

```
tensor([[2., 2., 0., 2., 2., 0.],
        [0., 0., 0., 2., 0., 2.],
        [2., 2., 2., 2., 0., 2.],
        [0., 2., 2., 0., 0., 2.],
        [0., 2., 0., 2., 0., 2.],
        [0., 2., 2., 2., 2., 0.]])
```

При применении отсева к матрице весов внимания с частотой 50 % половина элементов матрицы случайным образом устанавливается в нулевое значение. Чтобы компенсировать уменьшение количества активных элементов, значения оставшихся элементов матрицы увеличиваются в $1 / 0,5 = 2$ раза. Это масштабирование крайне важно для поддержания общего баланса весов внимания,

чтобы среднее влияние механизма внимания оставалось постоянным как на этапе обучения, так и на этапе предсказания.

Теперь применим отсев к самой матрице весов внимания:

```
torch.manual_seed(123)
print(dropout(attn_weights))
```

Итоговая матрица весов внимания сейчас имеет элементы, установленные в 0, и заново масштабированные элементы.

```
tensor([[2.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.4921, 0.4925, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000],
        [0.0000, 0.3327, 0.3331, 0.3084, 0.3331, 0.0000]],
        grad_fn=<MulBackward0>)
```

Обратите внимание: результирующие выходные данные могут выглядеть по-разному в зависимости от вашей операционной системы. Подробнее об этом несоответствии можно прочитать в системе отслеживания проблем PyTorch по адресу <https://github.com/pytorch/pytorch/issues/121595>.

Теперь, разобравшись с причинно-следственным вниманием и отсевом, мы можем разработать соответствующий класс на Python.

3.5.3. Реализация компактного класса причинно-следственного внимания

Настало время добавить причинно-следственное внимание и изменение отсева в класс `SelfAttention`, который мы разработали в разделе 3.4. Этот класс послужит шаблоном для разработки *многоцелевого внимания* (multi-head attention), которое является последним классом внимания, который мы реализуем (в разделе 3.6).

Но прежде чем начать, убедимся, что код может обрабатывать пакеты, состоящие из нескольких элементов, так, чтобы класс `CausalAttention` поддерживал пакетные выходные данные, создаваемые загрузчиком данных, который мы реализовали в главе 2.

Для простоты, чтобы смоделировать такой пакетный ввод, мы дублируем пример входного текста:

```
batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape)
```

Два входа по шесть токенов каждый
(размерность вложения — 3)

В результате получается трехмерный тензор, состоящий из двух входных текстов по шесть токенов в каждом.

Каждый токен представляет собой трехмерный вектор-вложение:

```
torch.Size([2, 6, 3])
```

Класс `CausalAttention` похож на класс `SelfAttention`, который мы реализовали ранее, за исключением того, что мы добавили компоненты отсева и причинно-следственной маски (листинг 3.3).

Листинг 3.3. Компактный класс причинно-следственного внимания

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            'mask',
            torch.triu(torch.ones(context_length, context_length),
                       diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2)
        attn_scores.masked_fill_(
            self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1
        )
        attn_weights = self.dropout(attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

По сравнению с классом `SelfAttention_v1` мы добавили слой отсева

Вызов `register_buffer` — также новое добавление (подробности в последующем тексте)

Мы транспонируем размерности 1 и 2, помещая размер пакета в первую позицию (индекс 0)

В PyTorch операции с конечным символом подчеркивания выполняются на месте (то есть в самом объекте, к которому применяется операция), что позволяет избежать ненужных копий объектов в памяти

На данном этапе все добавленные строки кода должны быть вам знакомы, но мы добавили вызов `self.register_buffer()` в метод `__init__`. Использование `register_buffer` в PyTorch необязательно во всех случаях, однако здесь оно дает несколько преимуществ. Например, когда мы используем класс `CausalAttention` в нашей LLM, буферы автоматически перемещаются на соответствующее устройство (ЦП (CPU) или ГП (GPU)) вместе с нашей моделью, что будет полезно при обучении модели. Это означает, что не нужно вручную проверять, находятся ли эти тензоры на том же устройстве, что и параметры модели. Следовательно, мы устраняем причины ошибок, связанные с несоответствием устройств.

Мы можем использовать класс `CausalAttention` следующим образом, аналогично `SelfAttention`, рассмотренному ранее:

```
torch.manual_seed(123)
context_length = batch.shape[1]
ca = CausalAttention(d_in, d_out, context_length, 0.0)
context_vecs = ca(batch)
print("context_vecs.shape:", context_vecs.shape)
```

Полученный контекстный вектор представляет собой трехмерный тензор, в котором каждый токен теперь представлен двумерным вектором вложения:

```
context_vecs.shape: torch.Size([2, 6, 2])
```

На рис. 3.23 показано, чего мы достигли на данный момент. Мы начали с упрощенного механизма внимания, добавили обучаемые весовые коэффициенты, а затем добавили маску причинно-следственного внимания. Далее мы расширим механизм причинно-следственного внимания и добавим многоцелевое внимание, которое будем использовать в нашей LLM.

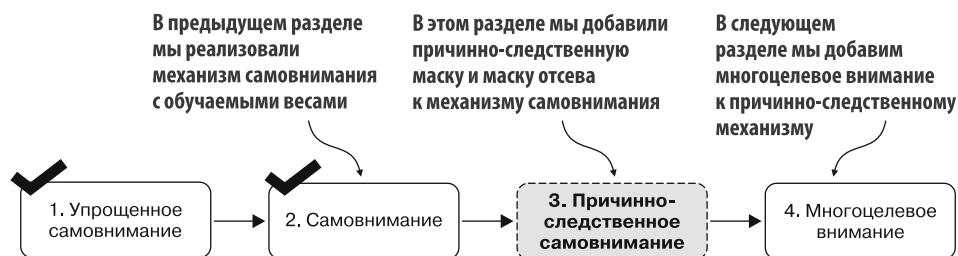


Рис. 3.23. Механизм самовнимания, реализованный на данный момент

3.6. Расширение одноцелевого внимания до многоцелевого

Последним шагом по созданию механизма самовнимания будет расширение ранее реализованного класса причинно-следственного внимания до нескольких направлений. Это также называется *многоцелевым вниманием*.

Термин «многоцелевое» относится к разделению механизма внимания на несколько «целей», каждая из которых работает независимо. В этом контексте один модуль причинно-следственного внимания можно считать одноцелевым вниманием, где есть только один набор весов внимания, обрабатывающих входные данные последовательно.

Мы рассмотрим расширение от причинно-следственного внимания к многоцелевому. Сначала мы интуитивно создадим многоцелевой модуль внимания,

объединив несколько модулей причинно-следственного внимания. Затем реализуем тот же многоцелевой модуль внимания более сложным, но при этом и более эффективным с точки зрения вычислений способом.

3.6.1. Объединение нескольких одноцелевых слоев внимания

С практической точки зрения реализация многоцелевого внимания предполагает создание нескольких экземпляров механизма самовнимания (см. рис. 3.18), каждый из которых имеет свои собственные весовые коэффициенты, а затем объединение их результатов. Использование нескольких экземпляров механизма самовнимания может быть вычислительно затратным, но это необходимо для распознавания сложных образов моделями, подобными LLM.

На рис. 3.24 показана структура многоцелевого модуля внимания, состоящего из нескольких одноцелевых модулей, уже знакомых вам по рис. 3.18, расположенных друг над другом.

Таким образом, вместо использования одной матрицы W_v для вычисления матриц значений в многоцелевом модуле внимания с двумя целями у нас теперь есть две матрицы весовых значений: W_{v_1} и W_{v_2} . То же самое относится к другим матрицам весовых значений: W_q и W_k . Мы получаем два набора контекстных векторов Z_1 и Z_2 , которые можно объединить в одну матрицу векторов Z .

Как упоминалось ранее, основная идея многоцелевого внимания заключается в том, чтобы запускать механизм внимания несколько раз (параллельно) с разными изученными линейными проекциями — результатами умножения входных данных (например, векторов запроса, ключа и значения в механизмах внимания) на весовую матрицу. В коде мы можем добиться этого, реализовав простой класс `MultiHeadAttentionWrapper`, который объединяет несколько экземпляров нашего ранее реализованного модуля `CausalAttention` (листинг 3.4).

Листинг 3.4. Класс-обертка для реализации многоцелевого внимания

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                  dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(
                d_in, d_out, context_length, dropout, qkv_bias
            )
             for _ in range(num_heads)]
        )

    def forward(self, x):
        return torch.cat([head(x) for head in self.heads], dim=-1)
```

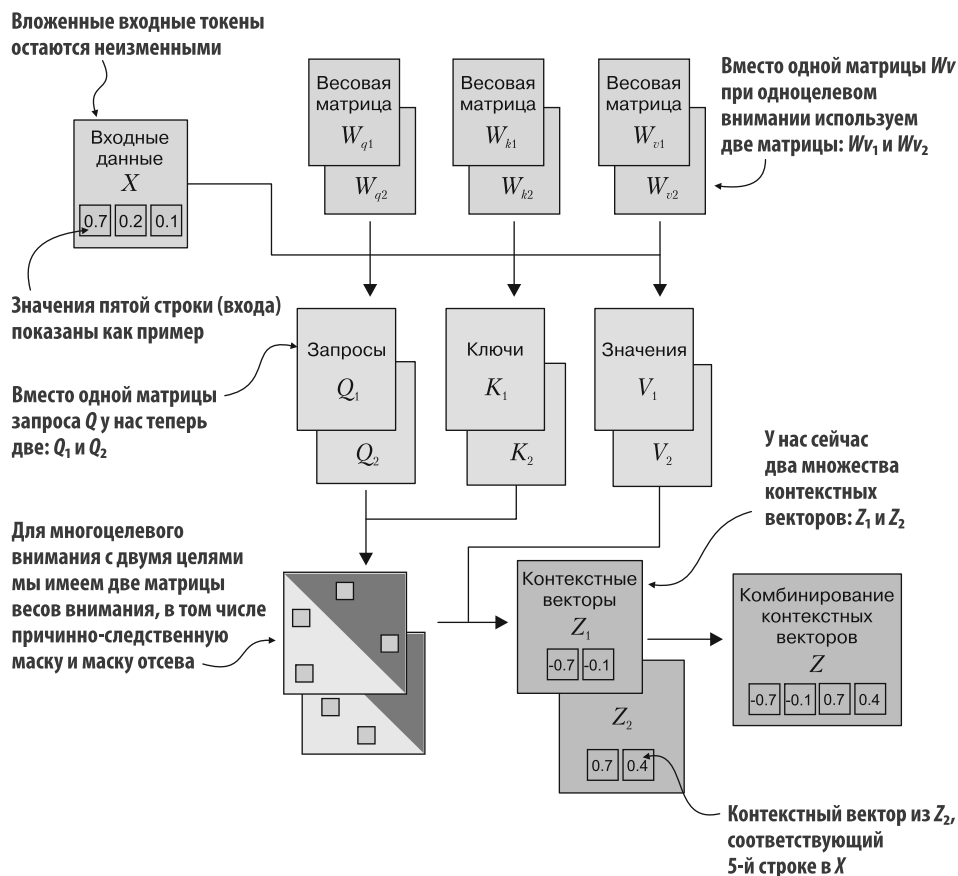


Рис. 3.24. Многоцелевой модуль внимания

Например, если мы используем класс `MultiHeadAttentionWrapper` с двумя целями внимания (`num_heads=2`) и выходным размером `CausalAttention d_out=2`, то получим четырехмерный контекстный вектор (`d_out*num_heads=4`) (рис. 3.25).

Используя `MultiHeadAttentionWrapper`, мы указали количество целей внимания (`num_heads`). Если мы установим `num_heads=2`, как в этом примере, то получим тензор с двумя наборами матриц контекстных векторов. В каждой такой матрице строки представляют векторы, соответствующие токенам, а столбцы соответствуют размерности вложения, заданной с помощью `d_out=4`. Мы объединяем эти матрицы контекстных векторов по столбцам. Поскольку у нас есть две цели внимания и размерность вложения 2, то итоговая размерность вложения составляет $2 \times 2 = 4$.

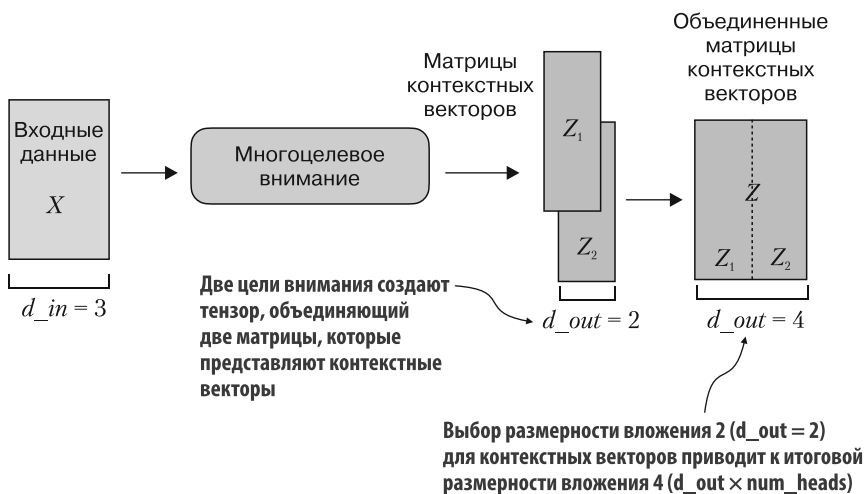


Рис. 3.25. Четырехмерный контекстный вектор

Чтобы вы могли увидеть идею на конкретном примере, мы можем использовать класс `MultiHeadAttentionWrapper`, аналогичный классу `CausalAttention`:

```
torch.manual_seed(123)
context_length = batch.shape[1] # Это номер токена
d_in, d_out = 3, 2
mha = MultiHeadAttentionWrapper(
    d_in, d_out, context_length, 0.0, num_heads=2
)
context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

Это приведет к следующему тензору с контекстными векторами:

```
tensor([[[[-0.4519, 0.2216, 0.4772, 0.1063],
          [-0.5874, 0.0058, 0.5891, 0.3257],
          [-0.6300, -0.0632, 0.6202, 0.3860],
          [-0.5675, -0.0843, 0.5478, 0.3589],
          [-0.5526, -0.0981, 0.5321, 0.3428],
          [-0.5299, -0.1081, 0.5077, 0.3493]],

         [[-0.4519, 0.2216, 0.4772, 0.1063],
          [-0.5874, 0.0058, 0.5891, 0.3257],
          [-0.6300, -0.0632, 0.6202, 0.3860],
          [-0.5675, -0.0843, 0.5478, 0.3589],
          [-0.5526, -0.0981, 0.5321, 0.3428],
          [-0.5299, -0.1081, 0.5077, 0.3493]]], grad_fn=<CatBackward0>)]
context_vecs.shape: torch.Size([2, 6, 4])
```

Первая размерность результирующего тензора `context_vecs` равна 2, так как у нас есть два входных текста (они продублированы, поэтому контекстные векторы для них абсолютно одинаковы). Вторая размерность относится к шести токенам в каждом входном тексте. Третья относится к четырехмерному вложению каждого токена.

Упражнение 3.2. Возврат двумерных векторов вложения

Измените входные аргументы в вызове `MultiHeadAttentionWrapper(..., num_heads=2)` так, чтобы выходные контекстные векторы были двумерными, а не четырехмерными, сохранив при этом параметр `num_heads=2`. Подсказка: вам не нужно изменять реализацию класса, достаточно изменить один из входных аргументов.

До этого момента мы реализовали `MultiHeadAttentionWrapper`, который объединял несколько однонаправленных модулей внимания. Однако они обрабатываются последовательно с помощью `[head(x) for head in self.heads]` в прямом методе. Мы можем улучшить эту реализацию, обрабатывая цели параллельно. Один из способов добиться желаемого — вычислять выходные данные для всех целей внимания одновременно с помощью перемножения матриц.

3.6.2. Реализация многоцелевого внимания с разделением весов

До сих пор мы создавали `MultiHeadAttentionWrapper` для реализации многоцелевого внимания, объединяя несколько одноцелевых модулей. Это делалось путем создания и объединения нескольких объектов `CausalAttention`.

Вместо того чтобы поддерживать два отдельных класса, `MultiHeadAttentionWrapper` и `CausalAttention`, мы можем объединить эти концепции в один класс `MultiHeadAttention`. Кроме того, помимо объединения `MultiHeadAttentionWrapper` с `CausalAttention`, мы внесем дополнительные изменения, чтобы сделать реализацию многоцелевого внимания более эффективной.

В `MultiHeadAttentionWrapper` несколько целей реализованы путем создания списка объектов `CausalAttention` (`self.heads`), каждый из которых представляет отдельную цель внимания. Класс `CausalAttention` независимо выполняет механизм внимания, а результаты от каждой цели объединяются. В отличие от этого, класс `MultiHeadAttention` объединяет функциональность нескольких целей в рамках одного класса. Он разбивает входные данные на несколько частей, изменяя форму тензоров запроса, ключа и значения, а затем объединяет результаты из этих частей после вычисления внимания.

Прежде чем продолжить обсуждение, рассмотрим класс `MultiHeadAttention` (листинг 3.5).

Листинг 3.5. Эффективная реализация класса многоцелевого внимания

```

class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out,
                  context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert (d_out % num_heads == 0), \
            "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            "mask",
            torch.triu(torch.ones(context_length, context_length), diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)
        queries = queries.view(b, num_tokens, self.num_heads, self.head_dim)
        values = values.view(b, num_tokens, self.num_heads, self.head_dim)

        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        attn_scores = queries @ keys.transpose(2, 3)
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]

        attn_scores.masked_fill_(mask_bool, -torch.inf)

        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = (attn_weights @ values).transpose(1, 2)

        context_vec = context_vec.contiguous().view(
            b, num_tokens, self.d_out)

        context_vec = self.out_proj(context_vec)
        return context_vec

```

Сокращает размерность проекции, чтобы она соответствовала выходной размерности

Использует линейный слой для объединения выходов целей

Мы неявно разделяем матрицу, добавляя размерность num_heads. Затем мы «раскатываем» последнюю размерность: (b, num_tokens, d_out) -> (b, num_tokens, num_heads, head_dim)

Форма тензора: (b, num_tokens, d_out)

Изменяет форму (b, num_tokens, num_heads, head_dim) в (b, num_heads, num_tokens, head_dim)

Вычисляет скалярное произведение для каждой цели

Маски, приведенные по размеру к количеству токенов

Использует маску для «заливки» показателей внимания

Форма тензора: (b, num_tokens, n_heads, head_dim)

Объединяет цели, где self.d_out = self.num_heads * self.head_dim

Добавляет необязательную линейную проекцию

Несмотря на то что изменение формы (`.view`) и транспонирование (`.transpose`) тензоров внутри класса `MultiHeadAttention` выглядят очень сложными с математической точки зрения, класс `MultiHeadAttention` реализует ту же концепцию, что и `MultiHeadAttentionWrapper`, описанный ранее.

На общем уровне мы объединяли в `MultiHeadAttentionWrapper` несколько одноцелевых слоев внимания в один многоцелевой. Класс `MultiHeadAttention` использует интегрированный подход. Он начинает с многоцелевого слоя, а затем разделяет этот слой на отдельные цели внимания (рис. 3.26).

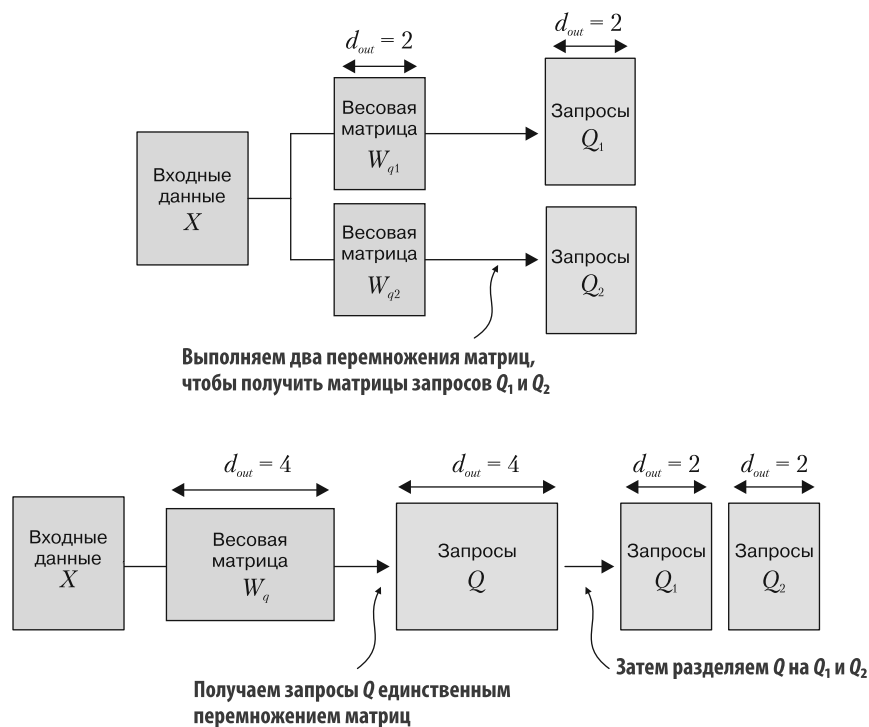


Рис. 3.26. В классе `MultiHeadAttentionWrapper` с двумя целями внимания мы инициализировали две весовые матрицы, W_{q1} и W_{q2} , и вычислили две матрицы запросов, Q_1 и Q_2 (вверху). В классе `MultiheadAttention` мы инициализируем одну большую весовую матрицу W_q , выполняем только одно перемножение матриц с входными данными, чтобы получить матрицу запросов Q , а затем разделяем ее на Q_1 и Q_2 (внизу). То же самое мы делаем для ключей и значений, которые здесь не отображены в целях улучшения визуального восприятия

Разделение тензоров запроса, ключа и значения достигается с помощью операций изменения формы и транспонирования тензоров, выполняемых с использованием

методов `.view` и `.transpose` в PyTorch. Входные данные сначала преобразуются (с помощью линейных слоев для запросов, ключей и значений), а затем изменяются для представления нескольких целей.

Ключевая операция заключается в разделении измерения `d_out` на `num_heads` и `head_dim`, где `head_dim = d_out` делится на `num_heads`. Затем это разделение достигается с помощью метода `.view`: тензор с размерами `(b, num_tokens, d_out)` преобразуется в тензор с размерами `(b, num_tokens, num_heads, head_dim)`.

Затем тензоры транспонируются, чтобы размерность `num_heads` оказалась перед размерностью `num_tokens`, в результате чего получается форма `(b, num_heads, num_tokens, head_dim)`. Это транспонирование играет решающую роль в правильном выравнивании запросов, ключей и значений, осуществляемом в разных целях, и эффективном выполнении пакетного перемножения матриц.

Чтобы вы могли увидеть такое пакетное перемножение матриц, предположим, что у нас есть следующий тензор:

```
a = torch.tensor([[[[0.2745, 0.6584, 0.2775, 0.8573],
                    [0.8993, 0.0390, 0.9268, 0.7388],
                    [0.7179, 0.7058, 0.9156, 0.4340]],
                  [[0.0772, 0.3565, 0.1479, 0.5331],
                    [0.4066, 0.2318, 0.4545, 0.9737],
                    [0.4606, 0.5159, 0.4220, 0.5786]]]])
```

← Форма этого тензора —
(b, num_heads, num_tokens,
head_dim) = (1, 2, 3, 4)

Теперь мы выполняем пакетное перемножение матриц между самим тензором и представлением тензора, в котором транспонировали последние две размерности, `num_tokens` и `head_dim`:

```
print(a @ a.transpose(2, 3))
```

Результат выглядит так:

```
tensor([[[[1.3208, 1.1631, 1.2879],
          [1.1631, 2.2150, 1.8424],
          [1.2879, 1.8424, 2.0402]],
        [[0.4391, 0.7003, 0.5903],
          [0.7003, 1.3737, 1.0620],
          [0.5903, 1.0620, 0.9912]]]])
```

В этом случае реализация перемножения матриц в PyTorch обрабатывает четырехмерный входной тензор так, что такое перемножение выполняется между двумя последними размерностями (`num_tokens`, `head_dim`), а затем повторяется для каждой из целей.

Например, предыдущий код становится более компактным способом вычисления перемножения матриц для каждой цели по отдельности:

```
first_head = a[0, 0, :, :]
first_res = first_head @ first_head.T
print("First head:\n", first_res)

second_head = a[0, 1, :, :]
second_res = second_head @ second_head.T
print("\nSecond head:\n", second_res)
```

Результаты точно такие же, как и при использовании пакетного перемножения матриц `print(a @ a.transpose(2, 3))`:

```
First head:
tensor([[1.3208, 1.1631, 1.2879],
        [1.1631, 2.2150, 1.8424],
        [1.2879, 1.8424, 2.0402]])

Second head:
tensor([[0.4391, 0.7003, 0.5903],
        [0.7003, 1.3737, 1.0620],
        [0.5903, 1.0620, 0.9912]])
```

Возвращаясь к `MultiHeadAttention`, после вычисления весов внимания и контекстных векторов последние из всех целей транспонируются обратно в форму `(b, num_tokens, num_heads, head_dim)`. Затем эти векторы преобразуются (сглаживаются) в форму `(b, num_tokens, d_out)`, эффективно объединяя выходные данные из всех целей.

Кроме того, мы добавили выходной проекционный слой (`self.out_proj`) в `MultiHeadAttention` после объединения целей, которого нет в классе `CausalAttention`. Этот выходной проекционный слой не является строго обязательным (подробнее см. в приложении Б), но он широко используется во многих архитектурах LLM, поэтому я добавил его сюда для полноты картины.

Несмотря на то что класс `MultiHeadAttention` выглядит более сложным, чем `MultiHeadAttentionWrapper`, из-за дополнительного изменения формы и транспонирования тензоров он более эффективен. Причина в том, что нам нужно только одно перемножение матриц для вычисления ключей, например `keys = self.w_key(x)` (то же самое верно для запросов и значений). В `MultiHeadAttentionWrapper` нам нужно было повторить это перемножение, которое с точки зрения вычислений является одним из самых дорогостоящих операций, для каждой цели.

Класс `MultiHeadAttention` можно использовать аналогично классам `SelfAttention` и `CausalAttention`, которые мы реализовали ранее:

```
torch.manual_seed(123)
batch_size, context_length, d_in = batch.shape
d_out = 2
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_heads=2)
context_vecs = mha(batch)
print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

Результаты показывают, что размер выходного файла напрямую зависит от аргумента `d_out`:

```
tensor([[[[0.3190, 0.4858],
          [0.2943, 0.3897],
          [0.2856, 0.3593],
          [0.2693, 0.3873],
          [0.2639, 0.3928],
          [0.2575, 0.4028]],

         [[0.3190, 0.4858],
          [0.2943, 0.3897],
          [0.2856, 0.3593],
          [0.2693, 0.3873],
          [0.2639, 0.3928],
          [0.2575, 0.4028]]], grad_fn=<ViewBackward0>])
context_vecs.shape: torch.Size([2, 6, 2])
```

Мы сейчас реализовали класс `MultiHeadAttention`, который будем использовать при реализации и обучении LLM. Обратите внимание: хотя код полностью функционален, я использовал относительно небольшие размеры векторов и количество целей внимания, чтобы результаты были читаемыми.

Для сравнения, самая маленькая модель GPT-2 (117 млн параметров) имеет 12 целей внимания и размер контекстного вектора, составляющий 768. Самая крупная модель GPT-2 (1,5 млрд параметров) имеет 25 целей внимания и размер контекстного вектора, равный 1600. Размеры входных токенов и контекстных вложений одинаковы в моделях GPT (`d_in = d_out`).

Упражнение 3.3. Инициализация модулей внимания размером с GPT-2

С помощью класса `MultiHeadAttention` инициализируйте многоцелевой модуль внимания, имеющий такое же количество целей внимания, как у самой маленькой модели GPT-2 (12 целей). Кроме того, убедитесь, что вы используете соответствующие размеры входных и выходных вложений, как у GPT-2 (768). Обратите внимание, что самая маленькая модель GPT-2 поддерживает длину контекста 1024 токена.

Итоги главы

- Механизмы внимания преобразуют входные элементы в расширенные контекстные векторные представления, которые содержат информацию обо всех входных данных.
- Механизм самовнимания вычисляет контекстное векторное представление как взвешенную сумму входных данных.
- В упрощенном механизме внимания весовые коэффициенты вычисляются с помощью скалярного произведения.
- Скалярное произведение — это способ поэлементного перемножения двух векторов с последующим суммированием произведений.
- Перемножение матриц, хотя и не является строго обязательным, помогает выполнять вычисления более эффективно и компактно, заменяя вложенные циклы `for`.
- В механизмах самовнимания, используемых в LLM, также называемых вниманием с масштабированным скалярным произведением, мы добавляем обучаемые весовые матрицы, чтобы вычислить промежуточные преобразования входных данных: запросы, значения и ключи.
- При работе с моделями, которые считывают и генерируют текст слева направо, мы добавляем маску причинно-следственного внимания, чтобы LLM не могла получить доступ к будущим токенам.
- Помимо масок причинно-следственного внимания для обнуления весовых коэффициентов внимания, мы можем добавить маску отсева, чтобы уменьшить эффект переобучения LLM.
- Модули внимания в LLM на основе трансформера содержат несколько экземпляров причинно-следственного внимания, которое называется многоцелевым.
- Чтобы создать многоцелевой модуль внимания, нужно объединить несколько экземпляров причинно-следственных модулей.
- Более эффективный способ создания многоцелевых модулей внимания предполагает пакетное перемножение матриц.

Создание GPT-подобной модели для генерации текста с нуля

В этой главе

- ✓ Программирование GPT-подобной LLM, которую можно обучить генерировать текст, похожий на созданный человеком.
- ✓ Нормализация слоев активации как способ стабилизировать обучение нейронной сети.
- ✓ Добавление коротких соединений в глубокие нейронные сети.
- ✓ Реализация блоков трансформера в целях создания GPT-моделей различных размеров.
- ✓ Вычисление количества параметров и требований к памяти для хранения информации, имеющей отношение к GPT-моделям.

Вы уже изучили и запрограммировали механизм многоцелевого внимания, один из основных компонентов LLM. Теперь мы сделаем следующий шаг (рис. 4.1): закодируем другие компоненты LLM и соберем их в модель, похожую на GPT, которую в следующей главе обучим генерировать текст, похожий на человеческий.

Архитектура LLM состоит из нескольких компонентов. Мы начнем с общего обзора, а затем более подробно рассмотрим отдельные компоненты модели.

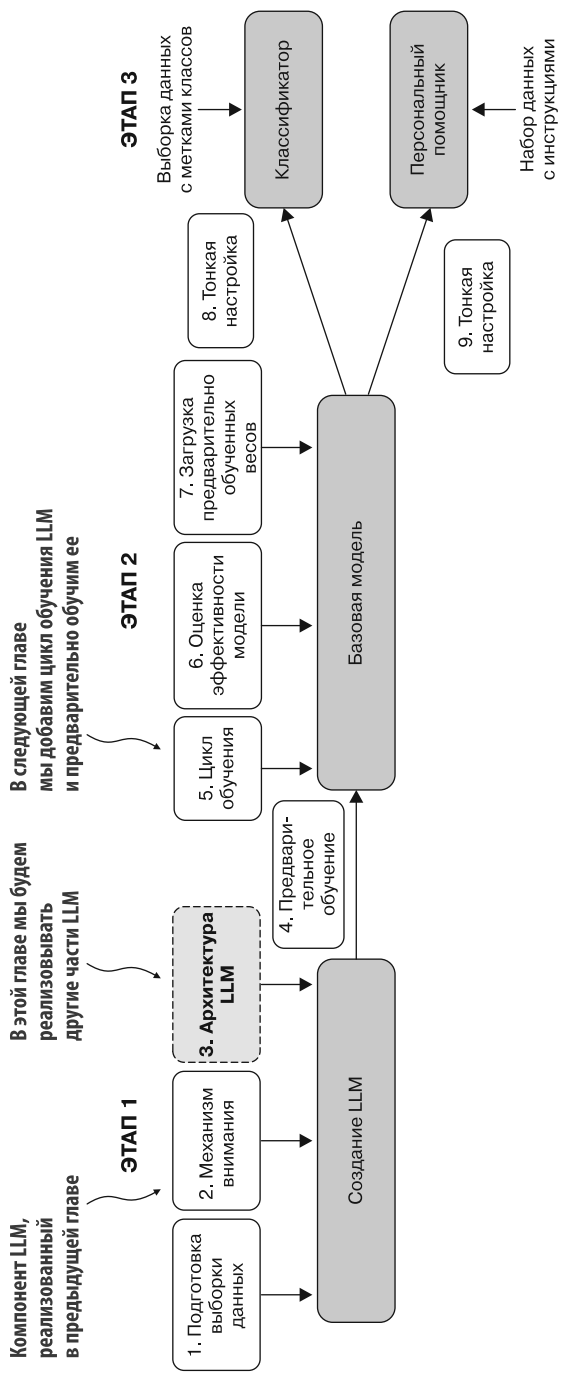


Рис. 4.1. Три основных этапа разработки LLM. В этой главе основное внимание уделяется третьему шагу этапа 1: реализации архитектуры LLM

4.1. Программирование архитектуры LLM

LLM, такие как GPT, представляют собой большие глубокие нейронные сети, предназначенные для генерации нового текста по одному слову (или токену) за раз. Однако, несмотря на их размер, архитектура модели менее сложна, чем вы могли бы подумать, поскольку многие ее компоненты повторяются, как вы увидите далее. На рис. 4.2 представлен вид сверху на GPT-подобную LLM; основные компоненты выделены.

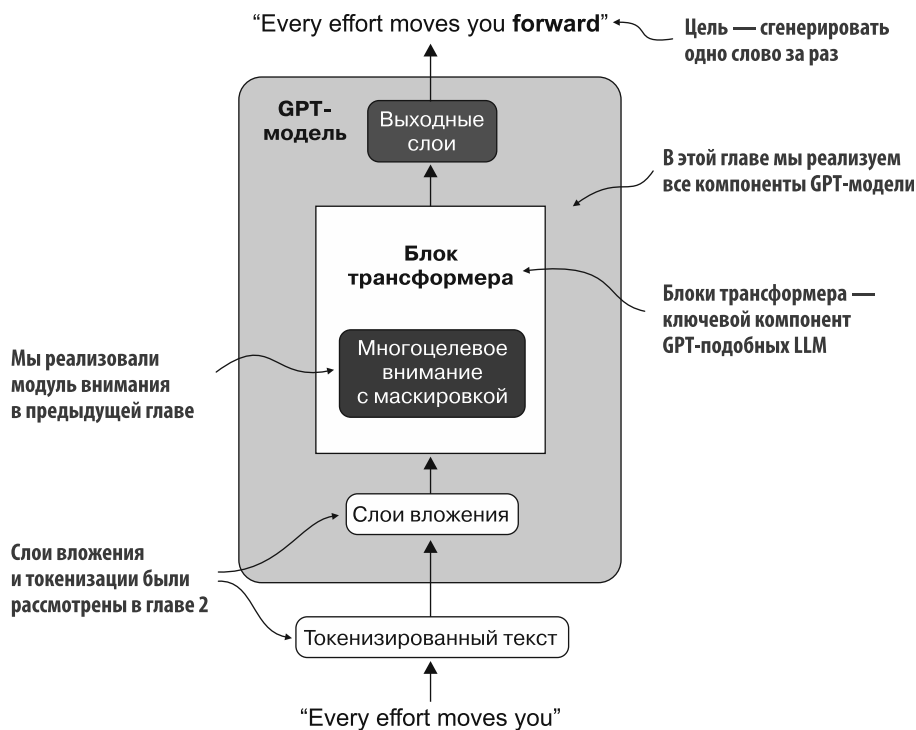


Рис. 4.2. Модель GPT. Помимо слоев вложения, она состоит из одного или нескольких блоков трансформера, содержащих модуль многоцелевого внимания с маской, который мы реализовали ранее

В предыдущих главах мы уже рассмотрели несколько аспектов архитектуры LLM, таких как токенизация и вложение входных данных, а также модуль многоцелевого внимания с маской. Теперь мы реализуем основную структуру модели GPT, в том числе ее блоки-трансформеры, которые позже обучим генерировать текст, похожий на человеческий.

Ранее я использовал меньшие размерности вложенных данных, чтобы концепции и примеры могли поместиться на одной странице. Теперь масштаб будет

увеличен до размера небольшой модели GPT-2, а именно до самой маленькой версии с 124 млн параметров, как описано в статье Алека Рэдфорда и др. *Language Models Are Unsupervised Multitask Learners* (<https://mng.bz/yoBq>). Обратите внимание, что в первоначальном отчете упоминалось 117 млн параметров, но позже это количество было скорректировано. В главе 6 мы сосредоточимся на загрузке предварительно обученных весов в нашу реализацию и ее адаптации для более крупных моделей GPT-2 с 345, 762 и 1542 млн параметров.

В контексте глубокого обучения и больших языковых моделей, таких как GPT, термин «параметры» относится к обучаемым весам модели. Эти веса, по сути, являются внутренними переменными модели, которые настраиваются и оптимизируются в процессе ее обучения при минимизации конкретной функции потерь. Такая оптимизация позволяет модели извлекать полезную информацию из обучающих данных.

Например, в слое нейронной сети, представленном матрицей (или тензором) весов размером 2048×2048 , каждый элемент этой матрицы является параметром. Поскольку в ней 2048 строк и 2048 столбцов, то общее количество параметров в этом слое равно 2048×2048 , то есть 4 194 304 параметра.

GPT-2 в сравнении с GPT-3

Обратите внимание, что мы сосредоточимся на GPT-2, поскольку OpenAI сделала общедоступными веса предварительно обученной модели, которые мы загрузим в нашу реализацию в главе 6. GPT-3 в целом имеет ту же архитектуру, за исключением того, что количество параметров увеличено с 1,5 млрд в GPT-2 до 175 млрд в GPT-3 и она обучается на большем количестве данных. На момент написания этой главы веса для GPT-3 не были опубликованы. Кроме того, GPT-2 лучше подходит для изучения реализации больших языковых моделей, поскольку ее можно запустить на ноутбуке, в то время как GPT-3 для обучения и предсказания требуется кластер графических процессоров. По данным Lambda Labs (<https://lambdalabs.com/>), для обучения GPT-3 на одном графическом процессоре V100 потребуется 355 лет, а на потребительском графическом процессоре RTX 8000 — 665 лет.

Мы задаем конфигурацию небольшой модели GPT-2 с помощью следующего словаря Python, который будем использовать в примерах кода далее:

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,      # Размер словаря
    "context_length": 1024,   # Длина контекста
    "emb_dim": 768,           # Размерность вложения
    "n_heads": 12,            # Количество голов внимания
    "n_layers": 12,           # Количество слоев
    "drop_rate": 0.1,         # Процент отсева
    "qkv_bias": False         # Смещение запроса-ключа-значения
}
```

В словаре `GPT_CONFIG_124M` мы для ясности и во избежание длинных строк кода используем краткие имена переменных:

- `vocab_size` относится к словарю из 50 257 слов, используемому токенизатором BPE (см. главу 2);
- `context_length` обозначает максимальное количество входных токенов, которые модель может обрабатывать с помощью позиционных вложений (см. главу 2);
- `emb_dim` — размерность вложения, преобразующего каждый токен в вектор размерностью 768 элементов;
- `n_heads` — количество целей в механизме многоцелевого внимания (см. главу 3);
- `n_layers` — количество блоков трансформера в модели, которые мы рассмотрим в следующей главе;
- `drop_rate` указывает на интенсивность механизма отсева (0.1 означает 10%-ный случайный отсев скрытых (внутренних) слоев модели) для предотвращения переобучения (см. главу 3);
- `qkv_bias` определяет, следует ли добавлять вектор смещения в слои `Linear` многоцелевого внимания для вычисления запроса, ключа и значения. Сначала мы отключим эту функцию, следуя принципам современных LLM, но вернемся к ней в главе 6, когда будем загружать в нашу модель предварительно обученные веса GPT-2 от OpenAI (см. главу 6).

Используя эту конфигурацию, мы реализуем пустой шаблон для архитектуры GPT (`DummyGPTModel`) (рис. 4.3). Это даст общее представление о том, как все взаимосвязано и какие другие компоненты нужно запрограммировать, чтобы собрать воедино полную архитектуру модели GPT.

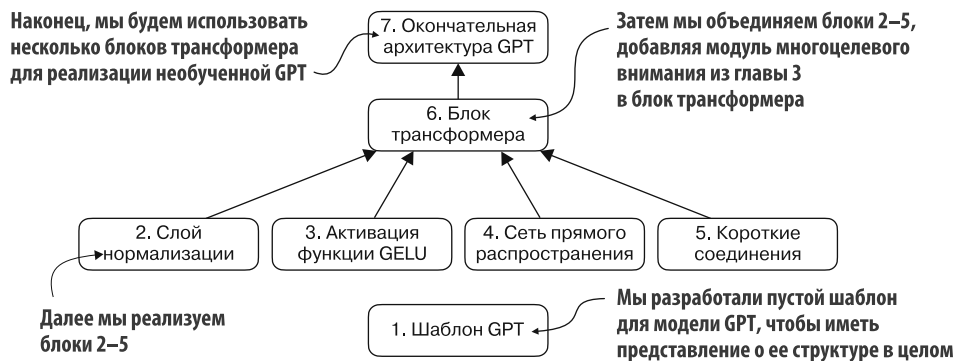


Рис. 4.3. Порядок программирования архитектуры GPT. Мы начинаем с базового шаблона, прежде чем перейти к отдельным ключевым компонентам и в конечном итоге собрать их в блок-трансформер в окончательной архитектуре GPT

Пронумерованные элементы на рис. 4.3 показывают порядок, в котором мы рассмотрим отдельные концепции, необходимые для программирования финальной архитектуры GPT. Мы начнем с шага 1 — базовой модели GPT, которую назовем `DummyGPTModel` (листинг 4.1).

Листинг 4.1. Класс шаблона архитектуры GPT-модели

```
import torch
import torch.nn as nn

class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg)
              for _ in range(cfg["n_layers"])]
        )
        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeddings = self.tok_emb(in_idx)
        pos_embeddings = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeddings + pos_embeddings
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits

class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()

    def forward(self, x):
        return x

class DummyLayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()

    def forward(self, x):
        return x
```

Использует шаблон для TransformerBlock

Использует шаблон для LayerNorm

Простой класс шаблона, который будет далее замещен настоящим TransformerBlock

Этот блок ничего не делает и возвращает свой ввод

Простой класс шаблона, который будет далее замещен настоящим LayerNorm

Параметры, имитирующие интерфейс LayerNorm

Класс `DummyGPTModel` в этом коде определяет упрощенную версию GPT-подобной модели, используя модуль нейронной сети PyTorch (`nn.Module`). Архитектура

модели в классе `DummyGPTModel` состоит из вложений токенов и позиций, отсева, серии блоков трансформера (`DummyTransformerBlock`), нормализации последнего перед выходным слоем (`DummyLayerNorm`) и линейного выходного слоя (`out_head`). Конфигурация передается через словарь Python, например, через словарь `GPT_CONFIG_124M`, который мы создали ранее.

Метод `forward` описывает поток данных, проходящих через модель: вычисляет вложения токенов и позиций для входных индексов, применяет отсев, пропускает данные через блоки трансформера, применяет нормализацию и, наконец, трансформирует данные с помощью линейного выходного слоя.

Код в листинге 4.1 уже функционален. Однако пока обратите внимание, что мы используем пустые шаблоны (`DummyLayerNorm` и `DummyTransformerBlock`) для блока-трансформера и нормализации слоев, которые разработаем позже.

Далее мы подготовим входные данные и инициализируем новую модель GPT, чтобы вы могли увидеть ее использование. Основываясь на нашем коде токенизатора (см. главу 2), рассмотрим в общих чертах, как данные поступают в модель GPT и выходят из нее (рис. 4.4).

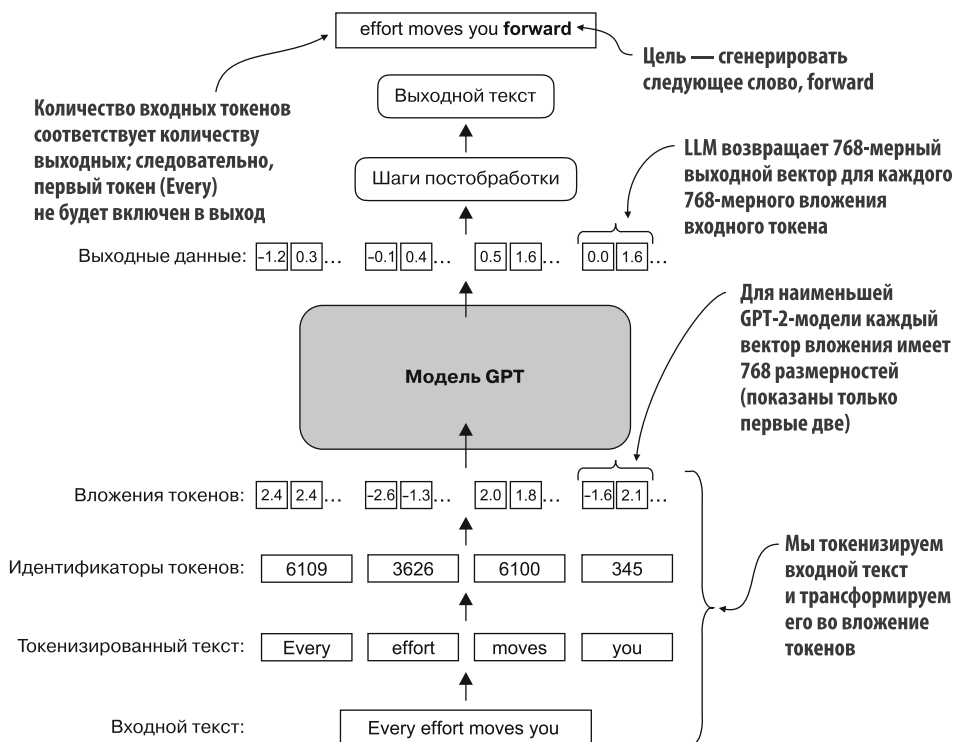


Рис. 4.4. Движение данных внутри модели GPT

Итак, на рис. 4.4 видно, что входные данные токенизируются, трансформируются и передаются в модель GPT. Обратите внимание, что в нашем ранее запрограммированном классе `DummyGPT` вложение токенов выполняется внутри модели GPT. В LLM размерность вложенных входных токенов обычно совпадает с размерностью выходных данных. Выходные векторы здесь представляют собой контекстные векторы (см. главу 3).

Чтобы реализовать эти операции, мы токенизируем пакет, состоящий из двух текстовых строк, с помощью токенизатора `tiktoken` из главы 2:

```
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
batch = []
txt1 = "Every effort moves you"
txt2 = "Every day holds a"

batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)
```

Результирующие идентификаторы токенов для этих двух текстов имеют следующий вид:

```
tensor([[6109, 3626, 6100, 345],
        [6109, 1110, 6622, 257]])
```

Первая строка соответствует первому
тексту, а вторая — второму

Далее мы инициализируем новый экземпляр класса `DummyGPTModel` с 124 млн параметров и подаем на вход токенизированный пакет `batch`:

```
torch.manual_seed(123)
model = DummyGPTModel(GPT_CONFIG_124M)
logits = model(batch)
print("Output shape:", logits.shape)
print(logits)
```

Выходные данные модели, которые обычно называют *логитами* (logits), выглядят так:

```
Output shape: torch.Size([2, 4, 50257])
tensor([[[[-1.2034,  0.3201, -0.7130, ..., -1.5548, -0.2390, -0.4667],
          [-0.1192,  0.4539, -0.4432, ...,  0.2392,  1.3469,  1.2430],
          [ 0.5307,  1.6720, -0.4695, ...,  1.1966,  0.0111,  0.5835],
          [ 0.0139,  1.6755, -0.3388, ...,  1.1586, -0.0435, -1.0400]],

         [[[-1.0908,  0.1798, -0.9484, ..., -1.6047,  0.2439, -0.4530],
          [-0.7860,  0.5581, -0.0610, ...,  0.4835, -0.0077,  1.6621],
          [ 0.3567,  1.2698, -0.6398, ..., -0.0162, -0.1296,  0.3717],
          [-0.2407, -0.7349, -0.5102, ...,  2.0057, -0.3694,  0.1814]]],

         grad_fn=<UnsafeViewBackward0>])
```

Выходной тензор содержит две строки, соответствующие двум текстовым образам. Каждый образ состоит из четырех токенов; каждый токен представляет собой вектор из 50 257 измерений, что соответствует размеру словаря токенизатора.

Вложение имеет 50 257 размерностей, поскольку каждая из них соответствует уникальному токену в словаре. Реализуя код постобработки, мы преобразуем эти 50 257-мерные векторы обратно в идентификаторы токенов, которые затем можно декодировать в слова.

Теперь, рассмотрев архитектуру GPT и ее входные и выходные данные, мы будем программировать индивидуальные шаблоны, начиная с класса нормализации, который заменит `DummyLayerNorm`, использованный в предыдущем коде.

4.2. Нормализация активаций с помощью нормализации слоев

Обучение глубоких нейронных сетей с большим количеством слоев иногда может быть затруднено из-за таких проблем, как исчезающие или растущие градиенты. Эти проблемы приводят к нестабильной динамике обучения и затрудняют эффективную настройку весов сети. В свою очередь, это означает, что процесс обучения усложняется из-за поиска набора параметров (весов) для нейронной сети, который минимизирует функцию потерь. Другими словами, нейросети трудно выявить закономерности в данных в той степени, которая позволила бы ей делать точные предсказания или принимать решения.

ПРИМЕЧАНИЕ Если вы новичок в обучении нейронных сетей и не знакомы с концепцией градиентов, то краткое введение в эти понятия можно найти в разделе A.4 приложения A. Однако, чтобы понимать содержание книги, вам не нужно обладать глубокими математическими познаниями в области градиентов.

Теперь реализуем *нормализацию слоев* (layer normalization), чтобы повысить стабильность и эффективность обучения нейронных сетей. Основная идея данной нормализации заключается в том, чтобы привести активацию (выходы) слоя нейронной сети к среднему значению 0 и дисперсии 1, также известной как единичная дисперсия. Эта корректировка ускоряет сходимость процесса обучения к эффективным весам и обеспечивает стабильное и надежное обучение. В GPT-2 и современных архитектурах трансформеров нормализация слоев обычно применяется до и после модуля многоцелевого внимания, а также, как мы видели на примере шаблона `DummyLayerNorm`, перед конечным выходным слоем. На рис. 4.5 показано, как функционирует нормализация слоев.

Мы можем воссоздать пример, показанный на рис. 4.5, с помощью следующего кода, в котором реализуем слой нейронной сети с пятью входами и шестью выходами, применяя его к двум входным примерам:

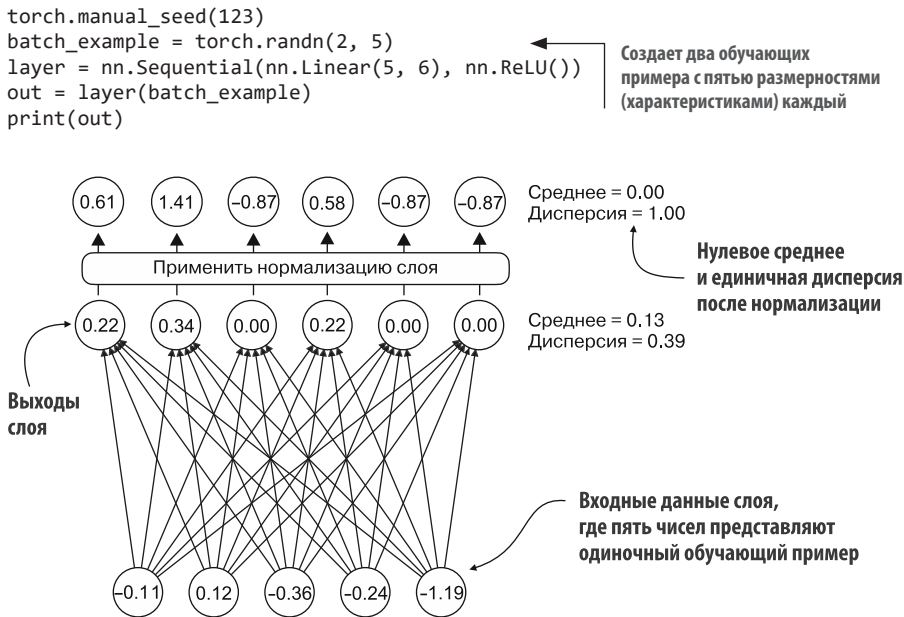


Рис. 4.5. Нормализация слоя, при которой шесть выходов слоя, также называемых активациями, нормализуются так, чтобы их среднее значение равнялось 0, а дисперсия — 1

В результате на экран выводится следующий тензор, где в первой строке находятся выходные данные слоя для первого входа, а во второй строке — выходные данные слоя для второго входа:

```

tensor([[0.2260, 0.3470, 0.0000, 0.2216, 0.0000, 0.0000],
        [0.2133, 0.2394, 0.0000, 0.5198, 0.3297, 0.0000]], grad_fn=<ReluBackward0>)

```

Слой нейронной сети, который мы запрограммировали, состоит из линейного слоя, за которым следует нелинейная функция активации ReLU (сокращение от rectified linear unit — «усеченное линейное преобразование»), которая является стандартной функцией активации в нейронных сетях. Если вы не знакомы с ReLU, то имейте в виду: она просто преобразует отрицательные входные данные в 0, гарантируя, что слой выводит только положительные значения. Позже мы будем использовать другую, более сложную функцию активации в GPT.

Прежде чем применить нормализацию слоя к этим результатам, посмотрим на среднее значение и дисперсию:

```

mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)

```

Получаем на выходе следующее:

```
Mean:
  tensor([[0.1324],
          [0.2170]], grad_fn=<MeanBackward0>)
Variance:
  tensor([[0.0231],
          [0.0398]], grad_fn=<VarBackward0>)
```

Первая строка в тензоре среднего значения содержит среднее значение для первой входной строки, а вторая строка в выходном — среднее значение для второй входной.

Использование `keepdim=True` в таких операциях, как вычисление среднего значения или дисперсии, гарантирует, что выходной тензор сохранит то же количество размерностей, что и входной, даже если операция сокращает размерность, указанную с помощью `dim`. Например, без `keepdim=True` возвращаемый тензор среднего значения был бы двумерным вектором `[0, 1324, 0, 2170]` вместо двумерной матрицы 2×1 `[[0, 1324], [0, 2170]]`.

Параметр `dim` указывает размерность, по которой в тензоре должна быть вычислена статистика (в данном случае среднее значение или дисперсия). Как показано на рис. 4.6, для двумерного тензора (например, матрицы) использование `dim=-1` в таких операциях, как вычисление среднего значения или дисперсии, аналогично использованию `dim=1`. Это связано с тем, что `-1` относится к последней размерности тензора, которая соответствует столбцам в двумерном тензоре. Позже, при добавлении нормализации слоя к модели GPT, которая создает трехмерные тензоры с формой `[batch_size, num_tokens, embedding_size]`, по-прежнему можно использовать `dim=-1` для нормализации по последнему измерению, избегая перехода от `dim=1` к `dim=2`.

Далее применим нормализацию слоя к полученным ранее выходным данным. Операция заключается в вычитании среднего значения и делении на квадратный корень из дисперсии (также известной как стандартное отклонение):

```
out_norm = (out - mean) / torch.sqrt(var)
mean = out_norm.mean(dim=-1, keepdim=True)
var = out_norm.var(dim=-1, keepdim=True)
print("Normalized layer outputs:\n", out_norm)
print("Mean:\n", mean)
print("Variance:\n", var)
```

Как мы видим по результатам, выходные данные нормализованного слоя, которые теперь содержат и отрицательные значения, имеют среднее значение 0 и дисперсию 1:

```
Normalized layer outputs:
  tensor([[ 0.6159,  1.4126, -0.8719,  0.5872, -0.8719, -0.8719],
          [-0.0189,  0.1121, -1.0876,  1.5173,  0.5647, -1.0876]],
          grad_fn=<DivBackward0>)
```



```
Mean:
  tensor([[ -5.9605e-08],
          [ 1.9868e-08]], grad_fn=<MeanBackward1>)
Variance:
  tensor([[ 1.],
          [ 1.]], grad_fn=<VarBackward0>)
```

dim=1 или dim=-1 вычисляет среднее значение в каждом столбце, что приводит к одному среднему на строку

							Среднее
Вход 1	0.22	0.34	0.00	0.22	0.00	0.00	0.13
Вход 2	0.21	0.23	0.00	0.51	0.32	0.00	0.21

dim=0 вычисляет среднее значение в каждой строке, что приводит к одному среднему на столбец

Вход 1	0.22	0.34	0.00	0.22	0.00	0.00	
Вход 2	0.21	0.23	0.00	0.51	0.32	0.00	
Среднее	0.21	0.29	0.00	0.37	0.16	0.00	

Рис. 4.6. Действие параметра `dim` при вычислении среднего значения тензора. Например, если у нас есть двумерный тензор (матрица) с размерами [строки, столбцы], то при использовании `dim=0` операция будет выполняться по строкам (по вертикали) (*внизу*), в результате чего будут получены агрегированные данные по каждому столбцу. При использовании `dim=1` или `dim=-1` операция будет выполняться по столбцам (по горизонтали) (*вверху*), в результате чего будут получены агрегированные данные для каждой строки

Обратите внимание, что значение $-5,9605 \times 10^{-8}$ в выходном тензоре — это экспоненциальное представление числа $-5,9605 \times 10^{-8}$, которое в десятичной форме равно $-0,000000059605$. Это значение очень близко к нулю, но не равно ему из-за небольших числовых ошибок, которые могут накапливаться ввиду конечной точности, с которой компьютеры представляют числа.

Чтобы улучшить читаемость, мы также можем отключить экспоненциальное представление при выводе значений тензора, установив для `sci_mode` значение `False`:

```
torch.set_printoptions(sci_mode=False)
print("Mean:\n", mean)
print("Variance:\n", var)
```

Теперь значения выглядят так:

```
Mean:
  tensor([[ 0.0000],
          [ 0.0000]], grad_fn=<MeanBackward1>)
Variance:
  tensor([[1.],
          [1.]], grad_fn=<VarBackward0>)
```

До сих пор мы пошагово программировали и применяли нормализацию слоев. Теперь заключим этот процесс в модуль PyTorch (листинг 4.2), который сможем использовать в модели GPT позже.

Листинг 4.2. Класс нормализации слоя

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```

Эта конкретная реализация нормализации слоя работает с последней размерностью входного тензора `x`, которая представляет собой размерность вложения (`emb_dim`). Переменная `eps` — это небольшая константа (эпсилон), добавляемая к дисперсии, чтобы предотвратить деление на ноль во время нормализации. `scale` и `shift` — два обучаемых параметра (той же размерности, что и входные данные), которые LLM автоматически настраивает во время обучения, если определено, что это улучшит ее производительность при выполнении ею обучающей задачи. Это позволяет модели научиться правильно масштабировать и смещать значения, которые лучше всего подходят для обрабатываемых данных.

Теперь попробуем модуль `LayerNorm` на практике и применим его к пакету входных данных:

```
ln = LayerNorm(emb_dim=5)
out_ln = ln(batch_example)
mean = out_ln.mean(dim=-1, keepdim=True)
var = out_ln.var(dim=-1, unbiased=False, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

Смещение дисперсии

В нашем методе вычисления дисперсии мы устанавливаем `unbiased=False`. Это значит, что при вычислении дисперсии мы делим на количество входных данных n в формуле дисперсии. При таком подходе не применяется поправка Бесселя, которая обычно использует $n - 1$ вместо n в знаменателе для корректировки смещения при оценке дисперсии выборки. Это решение приводит к так называемой смещенной оценке дисперсии. Для LLM, где размерность вложения n очень большая, разница между использованием n и $n - 1$ незначительна. Я выбрал этот подход, чтобы обеспечить совместимость со слоями нормализации модели GPT-2, а также потому, что он отражает поведение TensorFlow по умолчанию, которое использовалось для реализации оригинальной модели GPT-2. Использование аналогичной настройки гарантирует, что наш метод совместим с предварительно обученными весами, которые мы загрузим в главе 6.

Результаты показывают, что код нормализации слоя работает должным образом и нормализует значения каждого из двух входов так, чтобы их среднее значение равнялось 0, а дисперсия — 1:

```
Mean:
tensor([[ -0.0000],
        [ 0.0000]], grad_fn=<MeanBackward1>)
Variance:
tensor([[1.0000],
        [1.0000]], grad_fn=<VarBackward0>)
```

К этому моменту мы рассмотрели два блока, которые понадобятся для реализации архитектуры GPT (рис. 4.7). Далее мы рассмотрим функцию активации GELU, которая является одной из функций активации, используемых в LLM, вместо традиционной функции ReLU, которую использовали ранее.

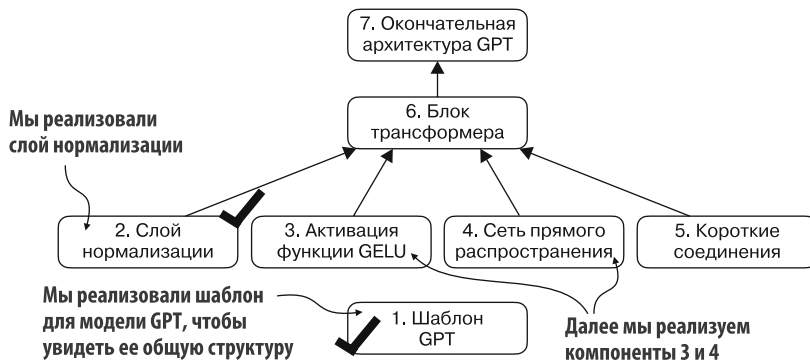


Рис. 4.7. Блоки, необходимые для создания архитектуры GPT. На данный момент мы завершили создание основы GPT и нормализацию слоев. Далее сосредоточимся на активации GELU и нейронной сети прямого распространения

Нормализация слоя в сравнении с пакетной нормализацией

Если вы знакомы с пакетной нормализацией, распространенным и традиционным методом нормализации для нейронных сетей, то вам может быть интересно, как она соотносится с нормализацией слоя. В отличие от пакетной, которая нормализует по размерности пакета, нормализация слоя нормализует по размерности признаков. LLM часто требуют значительных вычислительных ресурсов, и размер пакета во время обучения или предсказания может определяться техническими характеристиками имеющегося компьютера или сервера или конкретным вариантом использования. Поскольку нормализация слоя нормализует каждый входной экземпляр независимо от размера пакета, то это обеспечивает большую гибкость и стабильность в таких сценариях. Это особенно полезно при распределенном обучении или развертывании моделей в средах с ограниченными ресурсами.

4.3. Реализация сети с прямым распространением и активацией GELU

Далее мы реализуем небольшой подмодуль нейронной сети, используемый в качестве блока трансформера в LLM. Начнем с реализации функции активации GELU, которая играет ключевую роль в этом подмодуле.

ПРИМЕЧАНИЕ Дополнительную информацию о реализации нейронных сетей в PyTorch см. в разделе A.5 приложения A.

Исторически функция активации ReLU широко использовалась в глубоком обучении из-за ее простоты и эффективности в различных архитектурах нейронных сетей. Однако в LLM, помимо традиционной ReLU, используются несколько других функций активации. Двумя примечательными примерами являются GELU (Gaussian error linear unit) и SwiGLU (Swish-gated linear unit).

GELU и SwiGLU — более сложные и плавные функции активации, включающие линейные и нелинейные преобразования. Они обеспечивают более высокую эффективность моделей глубокого обучения, в отличие от более простой ReLU.

Функция активации GELU может быть реализована несколькими способами; распространенная версия определяется как $GELU(x) = x \cdot \Phi(x)$, где $\Phi(x)$ — кумулятивная функция распределения стандартного нормального распределения. Однако на практике обычно используется более дешевое в вычислительном плане приближение (исходная модель GPT-2 также обучалась с использованием этого приближения, которое было найдено путем подгонки кривой (curve fitting)):

$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + \tanh \left[\sqrt{\frac{2}{\pi}} \cdot (x + 0.044715 \cdot x^3) \right] \right).$$

В коде мы можем реализовать эту функцию как модуль PyTorch (листинг 4.3).

Листинг 4.3. Реализация функции активации GELU

```
class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3))
        ))
```

Далее, чтобы вы могли понять, как выглядит функция GELU и чем она отличается от функции ReLU, построим графики этих функций рядом:

```
import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()

x = torch.linspace(-3, 3, 100)
y_gelu, y_relu = gelu(x), relu(x)
plt.figure(figsize=(8, 3))
for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "ReLU"]), 1):
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)
plt.tight_layout()
plt.show()
```

Генерирует 100 точек в диапазоне значений от -3 до 3

Таким образом, ReLU (рис. 4.8, *справа*) — это кусочно-линейная функция, которая выдает на выходе входные данные, если они положительны; в противном случае она выдает ноль. GELU (см. рис. 4.8, *слева*) — гладкая нелинейная функция, которая аппроксимирует ReLU, но с ненулевым градиентом почти для всех отрицательных значений (за исключением примерно $x = -0,75$).

Плавность GELU может привести к улучшению свойств оптимизации во время обучения, так как позволяет более точно настраивать параметры модели. В отличие от GELU, у ReLU есть резкий скачок в точке нуля (см. рис. 4.8, *справа*), что иногда усложняет оптимизацию, особенно в очень глубоких сетях или сетях со сложной архитектурой. Более того, в отличие от ReLU, которая выдает ноль при любом отрицательном входном значении, GELU позволяет получать небольшое ненулевое значение при отрицательных входных данных. Эта особенность означает, что в процессе обучения нейроны, получающие отрицательный сигнал, все равно могут вносить свой вклад в процесс обучения, хотя и в меньшей степени, чем нейроны, получающие положительный сигнал.

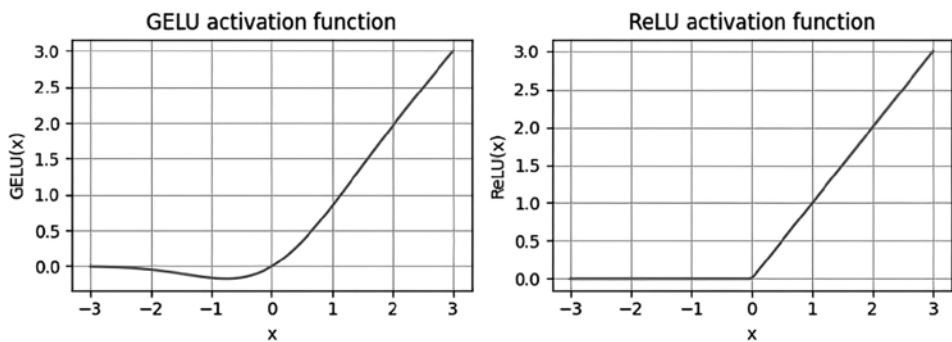


Рис. 4.8. Графики GELU и ReLU, произведенные с помощью matplotlib. По оси X показаны входные данные функции, а по оси Y — выходные

Далее с помощью функции GELU мы реализуем небольшой модуль нейронной сети, FeedForward (листинг 4.4), который будем использовать в блоке трансформера LLM.

Листинг 4.4. Модуль нейронной сети прямого распространения

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

    def forward(self, x):
        return self.layers(x)
```

Как мы видим, модуль FeedForward представляет собой небольшую нейронную сеть, состоящую из двух линейных слоев и функции активации GELU. В модели GPT с 124 млн параметров он получает входные пакеты с токенами размерностью вложения 768 каждый из словаря GPT_CONFIG_124M, где GPT_CONFIG_124M["emb_dim"] = 768. На рис. 4.9 показано, как изменяется размерность вложения в этой небольшой нейронной сети, когда мы передаем ей некоторые входные данные.

Следуя примеру на рис. 4.9, инициализируем новый модуль FeedForward с размерностью вложения токенов 768 и подадим ему на вход пакет с двумя примерами и тремя токенами в каждом:

```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768)  ← Создает вход с размером
out = ffn(x)                пакета, равным 2
print(out.shape)
```

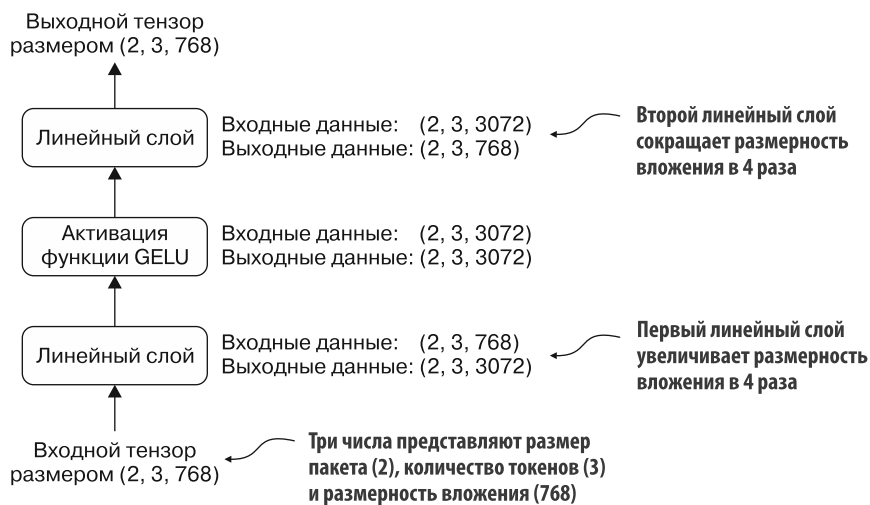


Рис. 4.9. Соединения между слоями нейронной сети прямого распространения.

Эта нейронная сеть может работать с переменными размерами пакетов и количеством токенов во входных данных. Однако размер вложения для каждого токена определяется и фиксируется при инициализации весов

Как мы видим, форма выходного тензора совпадает с формой входного:

```
torch.Size([2, 3, 768])
```

Модуль `FeedForward` играет важнейшую роль в улучшении способности модели обучаться на данных и обобщать их. Несмотря на то что входные и выходные размерности одинаковы, он расширяет размерность вложения до более высокоразмерного пространства, используя первый линейный слой (рис. 4.10). За этим расширением следует нелинейная активация функции GELU, а затем сжатие до исходного размера с помощью второго линейного преобразования. Такая конструкция позволяет исследовать более богатое пространство представлений.

Кроме того, единообразие входных и выходных размерностей упрощает архитектуру, позволяя объединять несколько слоев, как мы сделаем позже, не вынуждая корректировать размеры между ними, что делает модель более масштабируемой.

На данный момент мы реализовали большинство строительных блоков LLM (рис. 4.11). Далее мы рассмотрим концепцию коротких соединений, которые вставляются между различными слоями нейронной сети. Такие соединения важны для повышения эффективности обучения в глубоких нейронных сетях.

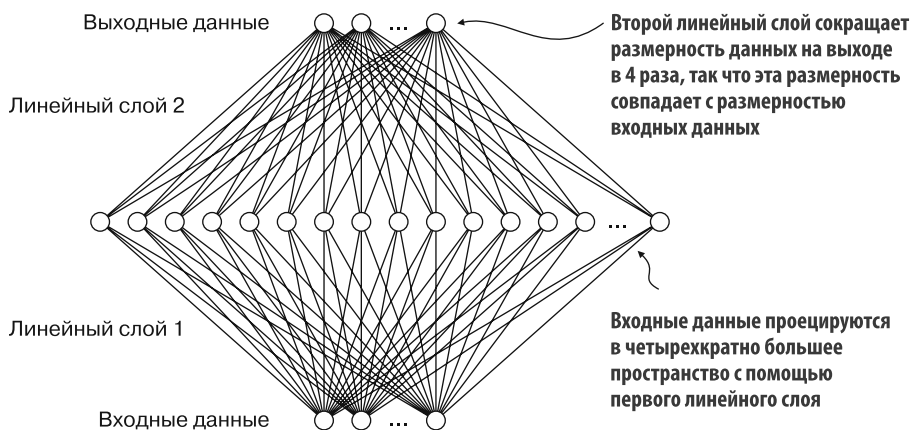


Рис. 4.10. Расширение и сжатие выходных данных слоя в нейронной сети прямого распространения. Сначала входные данные увеличиваются четырехкратно с 768 до 3072 значений. Затем второй слой сжимает 3072 значения до 768-мерного представления

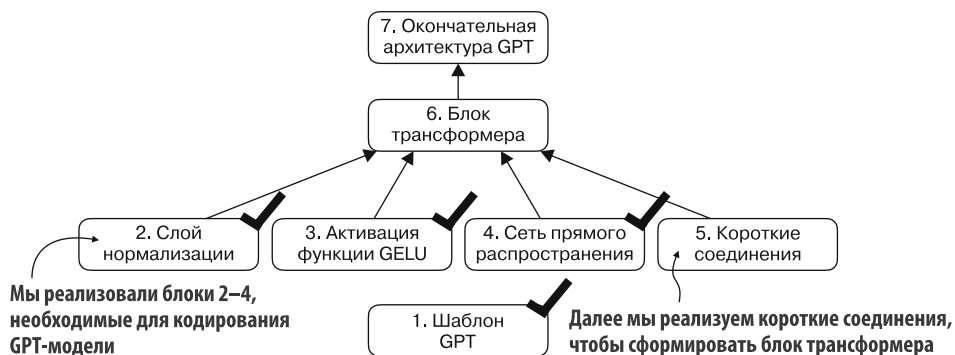


Рис. 4.11. Блоки, необходимые для создания архитектуры GPT. Черные «галочки» отмечают действия, которые мы уже выполнили

4.4. Добавление коротких соединений

В этом разделе мы обсудим концепцию *коротких соединений* (shortcut connections), также известных как пропускные или остаточные. Изначально они были предложены для глубоких сетей в компьютерном зрении (в частности, в остаточных сетях) как средство решения проблемы исчезающих градиентов. Проблема исчезающих градиентов заключается в том, что градиенты (которые управляют обновлением весов во время обучения) становятся все меньше по мере того, как распространяются в обратном направлении через слои, что затрудняет эффективное обучение предыдущих слоев.

На рис. 4.12 показано, что короткое соединение создает альтернативный, более короткий путь для распространения градиента по сети, пропуская один или несколько слоев, что достигается путем добавления выходных данных одного слоя к выходным данным более позднего слоя.

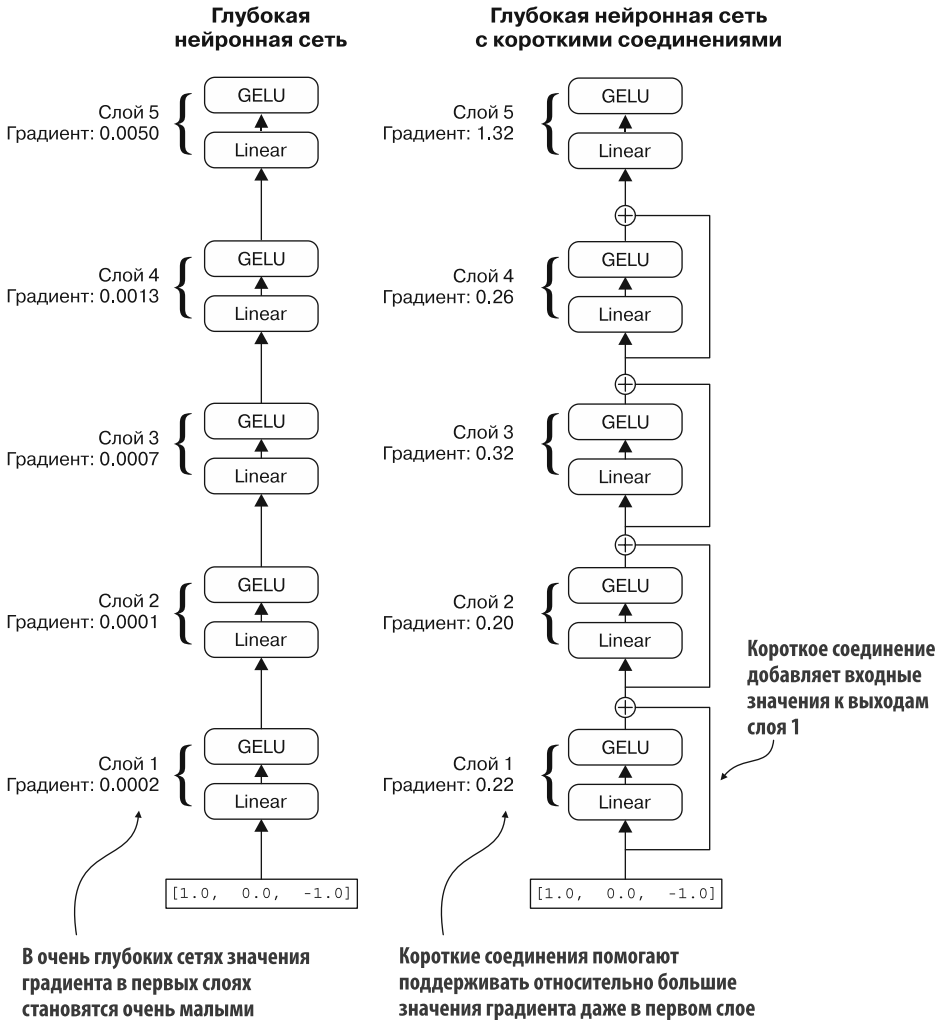


Рис. 4.12. Сравнение глубокой нейронной сети, состоящей из пяти слоев, без коротких соединений (слева) и с ними (справа). Короткие соединения подразумевают добавление входных данных слоя к его выходным данным, фактически создавая альтернативный путь, который обходит некоторые слои. Градиенты обозначают средний абсолютный градиент на каждом слое, который мы вычисляем в листинге 4.5

Вот почему эти соединения также известны как пропускные. Они играют решающую роль в сохранении градиентов во время прохождения сигнала назад при обучении нейронной сети.

В листинге 4.5 мы реализуем нейронную сеть, изображенную на рис. 4.12, чтобы показать, как можно добавить короткие соединения в методе `forward`.

Листинг 4.5. Нейронная сеть с короткими соединениями

```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([
            nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]), GELU())
        ])

    def forward(self, x):
        for layer in self.layers:
            layer_output = layer(x)
            if self.use_shortcut and x.shape == layer_output.shape:
                x = x + layer_output
            else:
                x = layer_output
        return x
```

Реализует пять слоев

Вычисляем выход текущего слоя

Проверяем возможность укорачивания соединений

В данном коде мы реализуем глубокую нейронную сеть с пятью слоями, каждый из которых состоит из слоя `Linear` и функции активации `GELU`. При прямом проходе мы итеративно передаем входные данные через слои и при необходимости добавляем короткие соединения, если для атрибута `self.use_shortcut` установлено значение `True`.

Теперь используем этот код для инициализации нейронной сети без коротких соединений. Каждый слой будет инициализирован так, чтобы он принимал данные с тремя входными значениями и возвращал три выходных. Последний слой возвращает единственное выходное значение:

```
layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])
torch.manual_seed(123)
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)
```

Задает начальное случайное значение для воспроизводимой инициализации весов

Далее мы реализуем функцию, которая вычисляет градиенты при обратном проходе модели:

```
def print_gradients(model, x):
    output = model(x)  # ← Прямой проход
    target = torch.tensor([[0.]])

    loss = nn.MSELoss()
    loss = loss(output, target)  # ← Вычисляет потери по близости
    #                               выходов к целям

    loss.backward()  # ← Обратный проход
    #                   для вычисления градиентов

    for name, param in model.named_parameters():
        if 'weight' in name:
            print(f"{name} has gradient mean of {param.grad.abs().mean().item()}")
```

В этом коде определяется функция потерь, которая вычисляет, насколько близки выходные данные модели и заданная пользователем цель (здесь для простоты используется значение 0). Затем при вызове `loss.backward()` PyTorch вычисляет градиент потерь для каждого слоя в модели. Мы можем посмотреть на весовые параметры с помощью `model.named_parameters()`. Предположим, что у нас есть матрица весовых параметров 3×3 для заданного слоя. В таком случае у этого слоя будет 3×3 значений градиента, и мы выводим среднее абсолютное значение градиента этих 3×3 значений, чтобы получить одно значение градиента для каждого слоя и упростить сравнение градиентов между слоями.

Метод `.backward()` — удобный метод в PyTorch, который вычисляет градиенты потерь, необходимые во время обучения модели, избавляя нас от необходимости самостоятельно выполнять математические вычисления для расчета градиента, что делает работу с глубокими нейронными сетями гораздо более доступной.

ПРИМЕЧАНИЕ Если вы не знакомы с концепцией градиентов и обучением нейронных сетей, то я рекомендую прочитать разделы А.4 и А.7 приложения А.

Теперь воспользуемся функцией `print_gradients` и применим ее к модели без коротких соединений:

```
print_gradients(model_without_shortcut, sample_input)
```

Результат выглядит так:

```
layers.0.0.weight has gradient mean of 0.00020173587836325169
layers.1.0.weight has gradient mean of 0.0001201116101583466
layers.2.0.weight has gradient mean of 0.0007152041653171182
layers.3.0.weight has gradient mean of 0.001398873864673078
layers.4.0.weight has gradient mean of 0.005049646366387606
```

Вывод в функции `print_gradients` показывает, что градиенты становятся меньше по мере перехода от последнего слоя (`layers.4`) к первому (`layers.0`). Это явление называется *проблемой исчезающего градиента* (vanishing gradient problem).

Теперь создадим модель с короткими соединениями и сравним ее с предыдущей:

```
torch.manual_seed(123)
model_with_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=True
)
print_gradients(model_with_shortcut, sample_input)
```

Результат таков:

```
layers.0.0.weight has gradient mean of 0.22169792652130127
layers.1.0.weight has gradient mean of 0.20694105327129364
layers.2.0.weight has gradient mean of 0.32896995544433594
layers.3.0.weight has gradient mean of 0.2665732502937317
layers.4.0.weight has gradient mean of 1.3258541822433472
```

Последний слой (`layers.4`) по-прежнему имеет больший градиент, чем другие слои. Однако значение градиента стабилизируется по мере продвижения к первому слою (`layers.0`) и не уменьшается до ничтожно малого значения.

В заключение отмечу, что короткие соединения важны для преодоления ограничений, связанных с проблемой исчезающего градиента в глубоких нейронных сетях. Короткие соединения являются основным строительным блоком очень больших моделей, таких как LLM, и помогут нам повысить эффективность обучения, обеспечивая равномерное распределение градиентов по слоям при обучении модели GPT в следующей главе.

Далее мы объединим все рассмотренные ранее концепции (нормализацию слоев, активацию функции GELU, модуль прямого распространения и короткие соединения) в блоке трансформера, который является последним блоком, необходимым для программирования архитектуры GPT.

4.5. Объединение механизма внимания и линейных слоев в блоке трансформера

Теперь реализуем *блок трансформера*, фундаментальный элемент GPT и других архитектур LLM. Этот блок, повторяющийся десятки раз в архитектуре GPT-2 с 124 млн параметров, объединяет несколько концепций, которые мы рассматривали ранее: многоцелевое внимание, нормализацию слоев, отсев, слои прямого распространения и активацию функции GELU. Позже мы подключим этот блок к остальным частям архитектуры GPT.

На рис. 4.13 показан блок трансформера, объединяющий несколько компонентов, в том числе модуль многоцелевого внимания с маскировкой (см. главу 3) и модуль `FeedForward`, который мы реализовали ранее (см. раздел 4.3). Когда блок обрабатывает входную последовательность, каждый ее элемент (например, слово или часть слова) представляется вектором фиксированного размера

(в данном случае 768 элементов). Операции в блоке, в том числе многоцелевое внимание и слои прямого распространения, предназначены для преобразования этих векторов так, чтобы их размерность была сохранена.

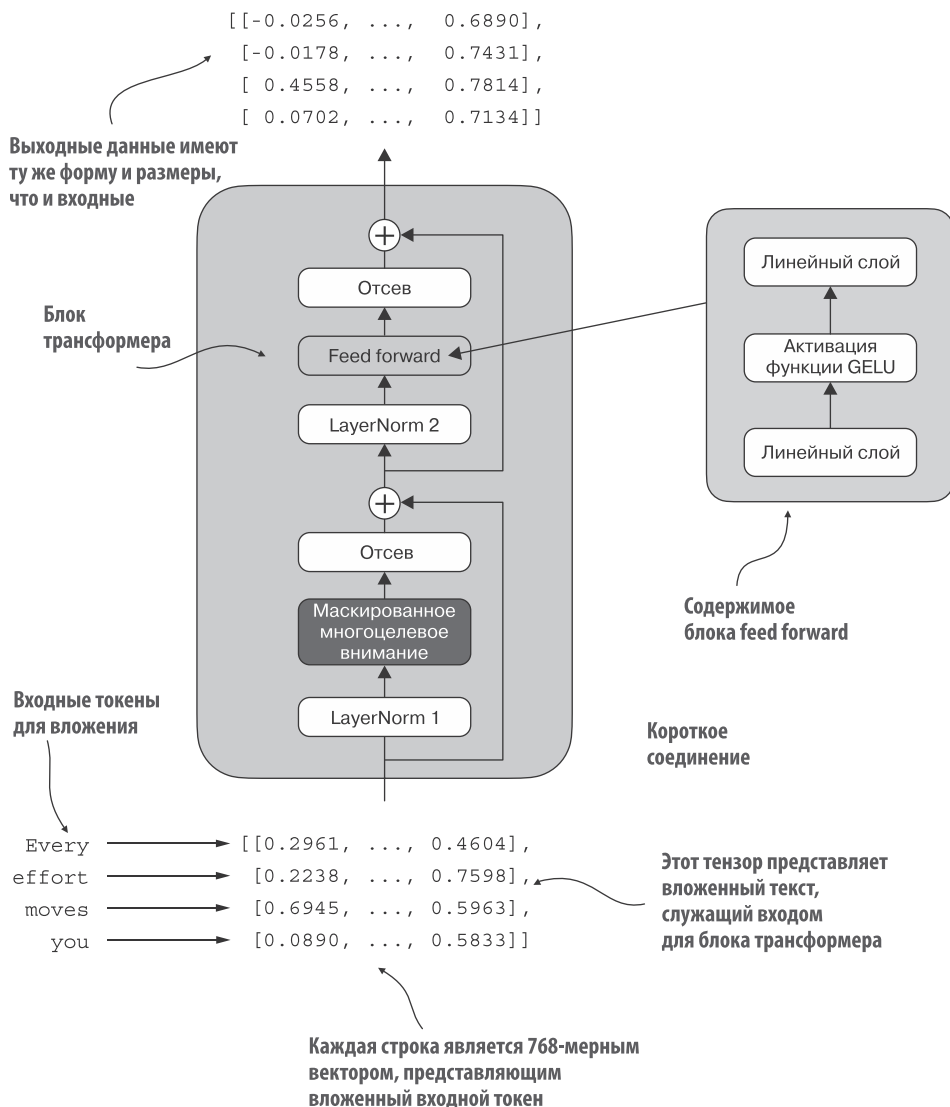


Рис. 4.13. Блок трансформера. Входные токены были преобразованы в 768-мерные векторы. Каждая строка соответствует векторному представлению одного токена. На выходе блока получают векторы того же размера, что и на входе, которые затем можно передать в последующие слои LLM

Идея заключается в том, что механизм самовнимания в блоке многоцелевого внимания определяет и анализирует взаимосвязи между элементами во входной последовательности. В отличие от этого, сеть прямого распространения изменяет данные индивидуально в каждой позиции. Такое сочетание не только обеспечивает более глубокое понимание и обработку входных данных, но и повышает общую способность модели обрабатывать сложные структуры данных.

Мы можем создать класс `TransformerBlock` в коде (листинг 4.6).

Листинг 4.6. Блок трансформера — компонент GPT

```
from chapter03 import MultiHeadAttention

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut

        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        return x
```

Короткое соединение для блока внимания

Возвращаем исходный вход

Короткое соединение для блока прямого распространения

Возвращаем исходный вход

В данном коде определяется класс `TransformerBlock` в PyTorch, который содержит механизм многоцелевого внимания (`MultiHeadAttention`) и сеть прямого распространения (`FeedForward`), оба из которых настраиваются на основе предоставленного словаря конфигурации (`cfg`), например `GPT_CONFIG_124M`.

Нормализация слоя (*LayerNorm*) применяется перед каждым из этих двух компонентов, а отсев используется после них для нормализации модели и предотвращения переобучения. Такая операция также известна как *Pre-LayerNorm*. В более старых архитектурах, наподобие первоначальной модели трансформера, применялась (после самовнимания и сетей прямого распространения) нормализация слоев, известная как *Post-LayerNorm*, что часто приводило к ухудшению динамики обучения.

Кроме того, класс *TransformerBlock* реализует прямой проход, при котором за каждым компонентом следует короткое соединение, добавляющее входные данные блока к его выходным данным. Эта важная функция помогает градиентам проходить через сеть во время обучения и улучшает обучение глубоких моделей (см. раздел 4.4). Используя словарь *GPT_CONFIG_124M*, который мы определили ранее, создадим экземпляр блока трансформера и передадим ему некоторые примеры данных:

```
torch.manual_seed(123)
x = torch.rand(2, 4, 768)
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)
```

← Создает вход размером [batch_size, num_tokens, emb_dim]

```
print("Input shape:", x.shape)
print("Output shape:", output.shape)
```

Получаем следующий результат:

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])
```

Как видите, блок трансформера сохраняет входные размеры в своих выходных данных; это значит, что архитектура трансформера обрабатывает последовательности данных, не изменяя их форму по всей сети.

Сохранение формы во всей архитектуре блока трансформера — не случайный, а важный аспект его конструкции. Такая конструкция обеспечивает его эффективное применение для широкого спектра задач типа «последовательность — последовательность», где каждый выходной вектор непосредственно соответствует входному, поддерживая взаимно однозначную связь. Однако на выходе получается контекстный вектор, который содержит информацию обо всей входной последовательности (см. главу 3). Это означает следующее: физические размеры последовательности (длина и количество признаков) остаются неизменными при прохождении через блок трансформера, однако содержимое каждого выходного вектора перекодируется в целях интеграции контекстной информации из всей входной последовательности.

Теперь, когда блок трансформера реализован, у нас есть все блоки, необходимые для реализации архитектуры GPT. Он объединяет нормализацию слоев, сеть прямого распространения, активацию функции GELU и короткие соединения (рис. 4.14). Как вы увидите далее, этот блок станет основным компонентом архитектуры GPT.

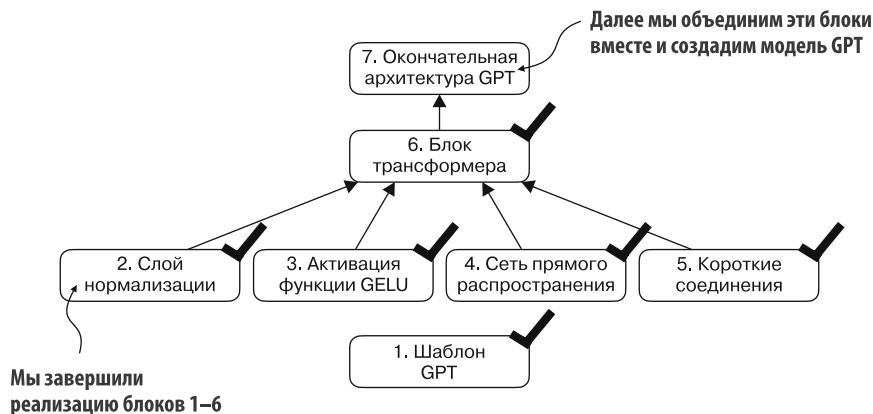


Рис. 4.14. Строительные блоки, необходимые для создания архитектуры GPT. Черными «галочками» отмечены блоки, которые мы уже создали

4.6. Программирование модели GPT

Мы начали эту главу с общего обзора архитектуры GPT, которую мы назвали `DummyGPTModel`. В этой реализации `DummyGPTModel` мы показали входные и выходные данные модели GPT, но ее основные блоки оставались «черным ящиком», использующим классы `DummyTransformerBlock` и `DummyLayerNorm` в качестве пустых шаблонов.

Теперь мы заменим `DummyTransformerBlock` и `DummyLayerNorm` на классы `TransformerBlock` и `LayerNorm`, которые написали ранее. Мы сделаем это, чтобы собрать полностью работающую версию первоначальной модели GPT-2 с 124 млн параметров. В главе 5 мы предварительно обучим модель GPT-2, а в главе 6 загрузим предварительно обученные веса из OpenAI.

Прежде чем мы соберем модель GPT-2 в коде, посмотрим на ее общую структуру (рис. 4.15), содержащую все концепции, которые были описаны до сих пор. Как мы видим, блок трансформера многократно повторяется в архитектуре модели GPT. В случае модели GPT-2 с 124 млн параметров он повторяется 12 раз, что мы указываем с помощью `n_layers` в словаре `GPT_CONFIG_124M`. Этот блок преобразования повторяется 48 раз в самой большой модели GPT-2 с 1542 млн параметров.

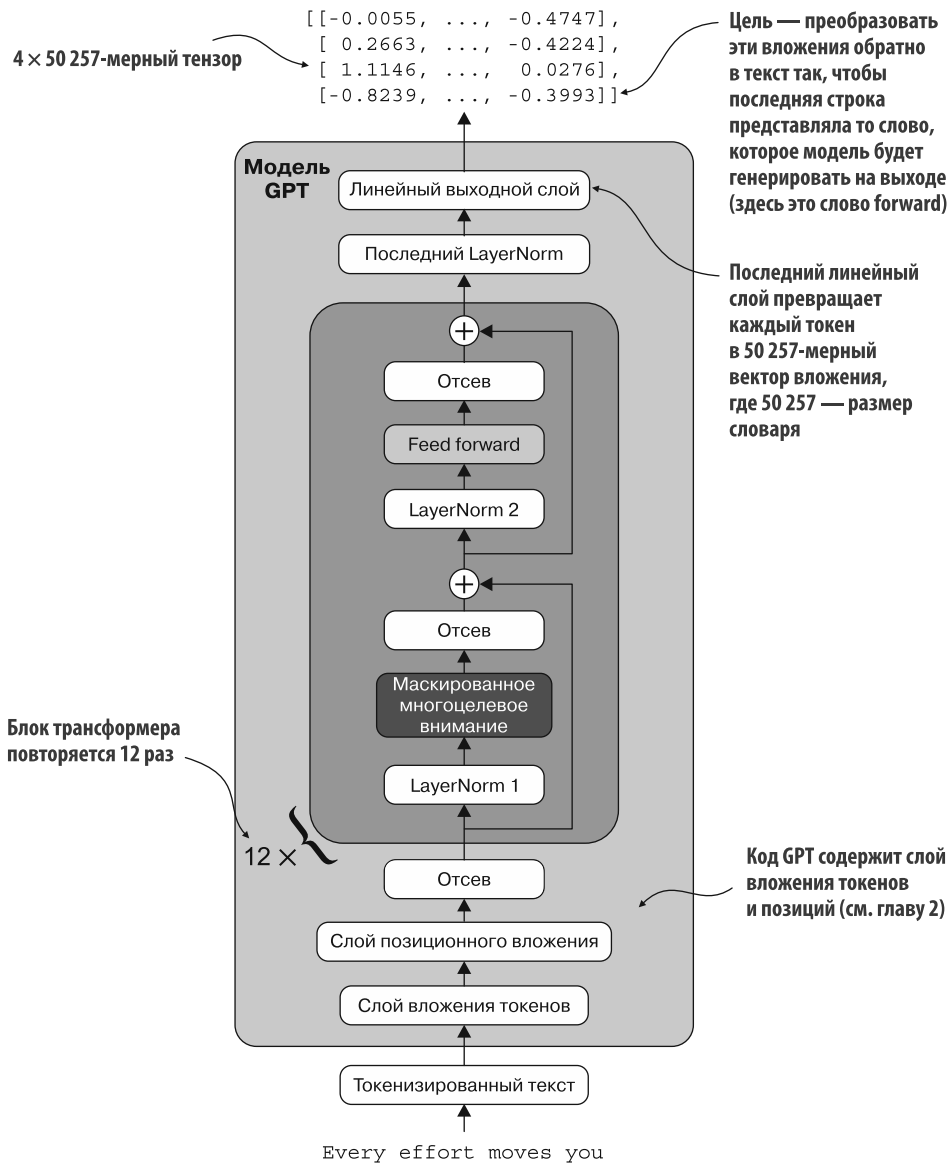


Рис. 4.15. Обзор архитектуры модели GPT, демонстрирующий поток данных в модели. Токенизированный текст (внизу) сначала преобразуется во вложения токенов, которые затем дополняются позиционными вложениями. Эта объединенная информация формирует тензор, проходящий через ряд блоков трансформера (в центре) (каждый из которых содержит слои многоцелевого внимания и прямого распространения с отсеком и нормализацией слоев), которые соединяются между собой и повторяются 12 раз

Затем выходные данные последнего блока трансформера нормализуются, прежде чем попасть в линейный выходной слой. Он проецирует выходные данные трансформера в многомерное пространство (в данном случае 50 257 измерений, соответствующих размеру словаря модели), чтобы можно было предсказать следующий токен в последовательности.

Теперь напомним код архитектуры, показанной на рис. 4.15 (листинг 4.7).

Листинг 4.7. Реализация архитектуры модели GPT

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])]

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeddings = self.tok_emb(in_idx)
        pos_embeddings = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeddings + pos_embeddings
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

← Настройка переменной device позволяет обучать модель на CPU или GPU в зависимости от того, где расположены входные данные

Благодаря классу `TransformerBlock` класс `GPTModel` получился относительно небольшим и компактным.

Конструктор `__init__` класса `GPTModel` инициализирует слои токенизации и позиционного вложения с помощью конфигураций, передаваемых через словарь Python `cfg`. Эти слои вложения отвечают за преобразование индексов входных токенов в векторы и добавление позиционной информации (см. главу 2).

Затем метод `__init__` создает стек модулей `TransformerBlock` размером, равным количеству слоев, указанных в `cfg`. После блоков трансформера применяется слой `LayerNorm`, который стандартизирует выходные данные блоков трансформера для стабилизации процесса обучения. Наконец, определяется линейная

выходная цель без смещения, которая проецирует выходные данные трансформера в пространство словаря токенизатора, чтобы можно было создать логиты для каждого токена в словаре.

Метод `forward` принимает на вход пакет индексов токенов, вычисляет их вложения, применяет позиционные вложения, пропускает последовательность через блоки трансформера, нормализует конечный результат, а затем вычисляет логиты, представляющие ненормированные вероятности следующего токена. В следующем разделе мы преобразуем эти логиты в токены и текстовые результаты.

Теперь инициализируем модель GPT с 124 млн параметров, используя словарь `GPT_CONFIG_124M`, который мы передаем в параметр `cfg`, и подадим на ее вход пакет текста, созданный нами ранее:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)
```

В результате выполнения этого кода мы получаем содержимое входного пакета и выходного тензора:

```
Input batch:
tensor([[6109, 3626, 6100, 345],
        [6109, 1110, 6622, 257]])
```

Идентификаторы токенов
первого примера текста

Идентификаторы токенов
второго примера текста

```
Output shape: torch.Size([2, 4, 50257])
tensor([[[ 0.3613,  0.4222, -0.0711, ...,  0.3483,  0.4661, -0.2838],
         [-0.1792, -0.5660, -0.9485, ...,  0.0477,  0.5181, -0.3168],
         [ 0.7120,  0.0332,  0.1085, ...,  0.1018, -0.4327, -0.2553],
         [-1.0076,  0.3418, -0.1190, ...,  0.7195,  0.4023,  0.0532]],

        [[-0.2564,  0.0900,  0.0335, ...,  0.2659,  0.4454, -0.6806],
         [ 0.1230,  0.3653, -0.2074, ...,  0.7705,  0.2710,  0.2246],
         [ 1.0558,  1.0318, -0.2800, ...,  0.6936,  0.3205, -0.3178],
         [-0.1565,  0.3926,  0.3288, ...,  1.2630, -0.1858,  0.0388]]],
        grad_fn=<UnsafeViewBackward0>)
```

Как мы видим, выходной тензор имеет форму `[2, 4, 50257]`, поскольку мы передали два входных текста по четыре токена в каждом. Последнее измерение, `50257`, соответствует размеру словаря токенизатора. Позже мы увидим, как преобразовать каждый из этих `50 257`-мерных выходных векторов обратно в токены.

Прежде чем переходить к написанию функции, которая преобразует выходные данные модели в текст, уделим немного внимания самой архитектуре модели и проанализируем ее размер. С помощью метода `numel()` (сокращение от `number`

of elements — «количество элементов») мы можем подсчитать общее количество параметров в тензорах параметров модели:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Total number of parameters: {total_params:,}")
```

Результат выглядит так:

```
Total number of parameters: 163,009,536
```

Теперь любопытный читатель может заметить несоответствие. Ранее мы говорили об инициализации модели GPT с 124 млн параметров, так почему же фактическое количество параметров составляет 163 млн?

Причина в концепции, называемой *привязкой весов* (weight tying), которая использовалась в первоначальной архитектуре GPT-2. Это означает, что в данной архитектуре GPT-2 веса слоя вложения токенов повторно используются в выходном слое. Чтобы вы могли лучше понять эту концепцию, взглянем на формы слоя вложения токенов и линейного выходного слоя, которые мы ранее инициализировали в модели с помощью GPTModel:

```
print("Token embedding layer shape:", model.tok_emb.weight.shape)
print("Output layer shape:", model.out_head.weight.shape)
```

Как мы видим, тензоры весов для обоих слоев имеют одинаковую форму:

```
Token embedding layer shape: torch.Size([50257, 768])
Output layer shape: torch.Size([50257, 768])
```

Входные и выходные слои очень большие из-за количества строк в словаре токенизатора, состоящем из 50 257 слов. Исключим количество параметров выходного слоя из общего количества параметров модели GPT-2 в соответствии с привязкой весов:

```
total_params_gpt2 = (
    total_params - sum(p.numel()
        for p in model.out_head.parameters())
)
print(f"Number of trainable parameters "
      f"considering weight tying: {total_params_gpt2:,}")
)
```

Результат получается следующим:

```
Number of trainable parameters considering weight tying: 124,412,160
```

Как мы видим, размер модели теперь составляет всего 124 млн параметров, что соответствует исходному размеру модели GPT-2.

Связывание весов уменьшает общий объем памяти и вычислительную сложность модели. Однако, по моему опыту, использование отдельных слоев для

вложения токенов и выходных данных модели приводит к более эффективному ее обучению. Поэтому в нашей реализации `GPTModel` мы используем отдельные слои. То же самое относится к современным LLM. Мы еще вернемся к концепции привязки весов и реализуем ее позже, в главе 6, когда будем загружать предварительно обученные веса из OpenAI.

Упражнение 4.1. Количество параметров в модуле прямого распространения и в модуле внимания

Подсчитайте и сравните количество параметров, которые содержатся в модуле прямого распространения и в модуле многоцелевого внимания.

Наконец, рассчитаем требования к памяти для 163 млн параметров в нашем объекте `GPTModel`:

```
total_size_bytes = total_params * 4
total_size_mb = total_size_bytes / (1024 * 1024)
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

Вычисляет общий размер в байтах
(предполагая float32, 4 байта на параметр)

Преобразует
в мегабайты

Получаем следующий результат:

Total size of the model: 621.83 MB

В заключение, рассчитав требования к памяти для 163 млн параметров в нашем объекте `GPTModel` и предположив, что каждый параметр представляет собой 32-битное число с плавающей запятой, занимающее 4 байта, мы обнаружили, что общий размер модели составляет 621,83 Мбайт, что свидетельствует об относительно большой емкости хранилища, необходимой для размещения даже сравнительно малых LLM.

Теперь, когда мы реализовали архитектуру `GPTModel` и убедились, что она производит на выходе числовые тензоры размером `[batch_size, num_tokens, vocab_size]`, напомним код для преобразования этих выходных тензоров в текст.

Упражнение 4.2. Инициализация более крупных моделей GPT

Мы инициализировали модель GPT с 124 млн параметров, которая известна как GPT-2 small. Не внося никаких изменений в код, кроме обновления файла конфигурации, используйте класс `GPTModel` для реализации GPT-2 medium (задействуя 1024-мерные вложения, 24 блока трансформера, 16 многоцелевых блоков внимания), GPT-2 large (задействуя 1280-мерные вложения, 36 блоков трансформера, 20 многоцелевых блоков внимания) и GPT-2 XL (задействуя 1600-мерные вложения, 48 блоков трансформера, 25 многоцелевых блоков внимания). В качестве бонуса рассчитайте общее количество параметров в каждой модели GPT.

4.7. Генерация текста

Теперь мы реализуем код, который преобразует выходные тензоры модели GPT обратно в текст. Прежде чем приступить к работе, вкратце рассмотрим, как генеративная модель, такая как LLM, генерирует текст по одному слову (или токену) за раз.

На рис. 4.16 показан пошаговый процесс, с помощью которого модель GPT генерирует текст по имеющейся подсказке (контексту) типа *Hello, I am*. На каждой итерации входной контекст расширяется, позволяя модели генерировать связный и контекстуально правильный текст. Она предсказывает следующий токен, добавляя его к входному контексту для следующего раунда предсказания. На первой итерации добавляется *a*, на второй — *model*, на третьей — *ready*. К шестой итерации модель построила полное предложение: *Hello, I am a model ready to help*.

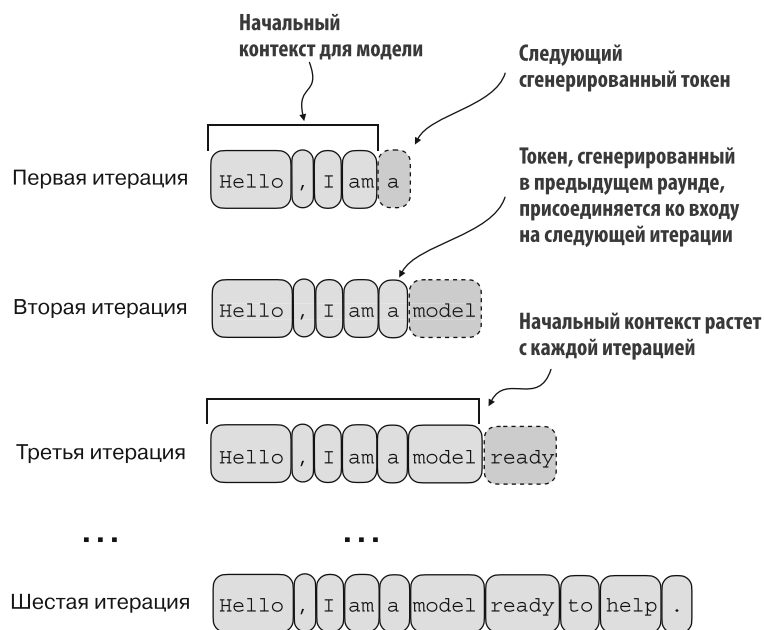


Рис. 4.16. Пошаговая генерация текста моделью

Мы увидели, что наша текущая реализация `GPTModel` выводит тензоры размерами `[batch_size, num_token, vocab_size]`. Теперь возникает вопрос: как модель GPT переходит от этих выходных тензоров к сгенерированному тексту?

Этот процесс состоит из нескольких шагов (рис. 4.17). Среди них — декодирование выходных тензоров, выбор токенов на основе распределения вероятностей и преобразование этих токенов в удобочитаемый текст.

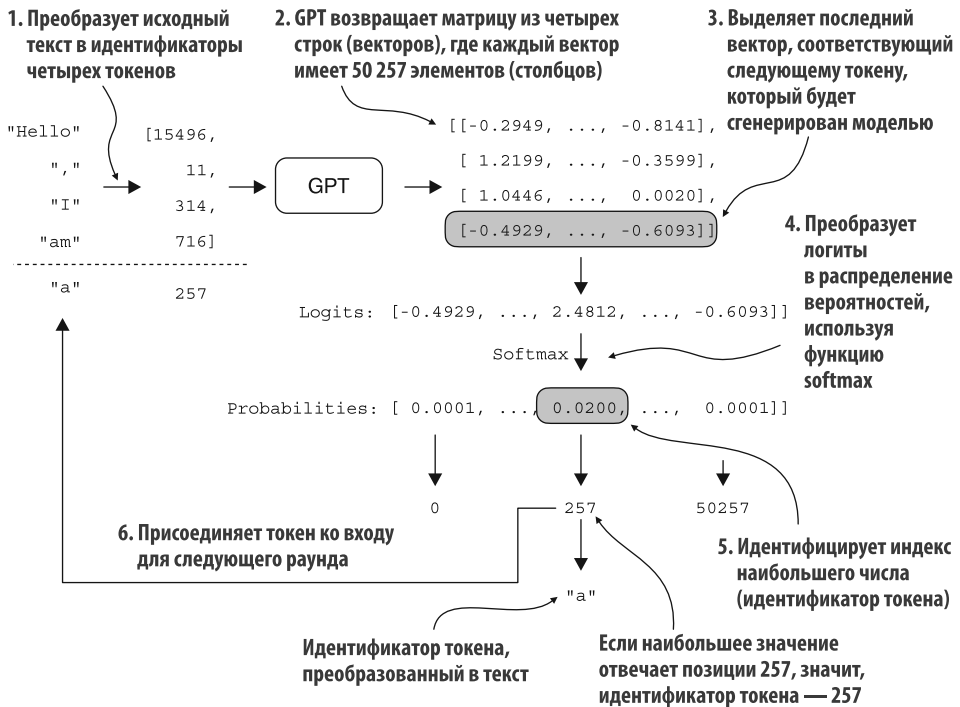


Рис. 4.17. Механика генерации текста в модели GPT на примере одной итерации. Процесс начинается с кодирования входного текста в идентификаторы токенов, которые после этого передаются в модель GPT. Выходные данные модели затем преобразуются обратно в текст и добавляются к исходному входному тексту

Процесс генерации следующего токена, подробно показанный на рис. 4.17, иллюстрирует один шаг, на котором модель GPT генерирует следующий токен на основе входных данных. На каждом шаге модель выводит матрицу с векторами, представляющими потенциальные следующие токены. Вектор, соответствующий следующему токеноу, извлекается и преобразуется в распределение вероятностей с помощью функции `softmax`. В векторе, содержащем результирующие оценки вероятности, находится индекс наибольшего значения, который соответствует идентификатору токена. Затем этот идентификатор токена декодируется обратно в текст, образуя следующий токен в последовательности. Наконец, этот токен добавляется к предыдущим входным данным, формируя новую входную последовательность для последующей итерации. Этот пошаговый процесс позволяет модели последовательно генерировать текст, создавая связные фразы и предложения на основе исходного входного контекста.

На практике мы повторяем этот процесс на протяжении многих итераций (см. рис. 4.16), пока не достигнем заданного пользователем количества

сгенерированных токенов. Мы можем реализовать процесс генерации токенов (листинг 4.8).

Листинг 4.8. Функция модели GPT для генерации текста

```
def generate_text_simple(model, idx,
                        max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
```

Укорачивает текущий контекст, если его длина превышает пороговое значение, например, если LLM поддерживает только пять токенов, а длина контекста — 10, то только пять последних токенов используются как контекст

idx — массив индексов размером (batch, n_tokens) текущего контекста

Фокусируется только на последнем по времени шаге, так что (batch, n_token, vocab_size) становится (batch, vocab_size)

Форма probas — (batch, vocab_size)

Форма idx_next — (batch, 1)

Присоединяет индекс к текущей последовательности, где форма idx — (batch, n_tokens+1)

В этом коде показана простая реализация генеративного цикла для языковой модели с использованием PyTorch. Здесь выполняется итерация для генерации заданного количества новых токенов, обрезается текущий контекст до максимального размера контекста модели, вычисляются предсказания, а затем выбирается следующий токен на основе предсказания с наибольшей вероятностью.

Чтобы запрограммировать функцию `generate_text_simple`, мы используем функцию `softmax` для преобразования логитов в распределение вероятностей, из которого определяем позицию с наибольшим значением с помощью `torch.argmax`. Функция `softmax` монотонна, то есть сохраняет порядок входных данных при преобразовании в выходные. Таким образом, на практике шаг использования `softmax` является избыточным, поскольку позиция с наибольшим значением в выходном тензоре `softmax` совпадает с позицией в тензоре логитов. Другими словами, мы могли бы применить функцию `torch.argmax` непосредственно к тензору логитов и получить идентичные результаты. Однако я привожу код для преобразования, чтобы показать весь процесс преобразования логитов в вероятности. Это может дать дополнительную информацию о том, как модель генерирует наиболее вероятный следующий токен (процесс, известный как *жадное декодирование* (greedy decoding)).

Когда мы в следующей главе напишем код для обучения GPT, то будем использовать дополнительные методы выбора, чтобы изменить результаты `softmax` так, чтобы модель не всегда выбирала наиболее вероятный токен. Это позволяет сделать сгенерированный текст более разнообразным и креативным.

Процесс создания одного идентификатора токена за раз и добавления его в контекст с помощью функции `generate_text_simple` дополнительно показан на

рис. 4.18. (Процесс генерации идентификаторов токенов для каждой итерации был представлен на рис. 4.17.) Идентификаторы токенов генерируются итеративно. Например, на первой итерации модель получает токены, соответствующие *Hello, I am*, предсказывает следующий токен (с идентификатором 257, который соответствует букве *a*) и добавляет его к входным данным. Этот процесс повторяется до тех пор, пока модель не выдаст полное предложение *Hello, I am a model ready to help* после шести итераций.

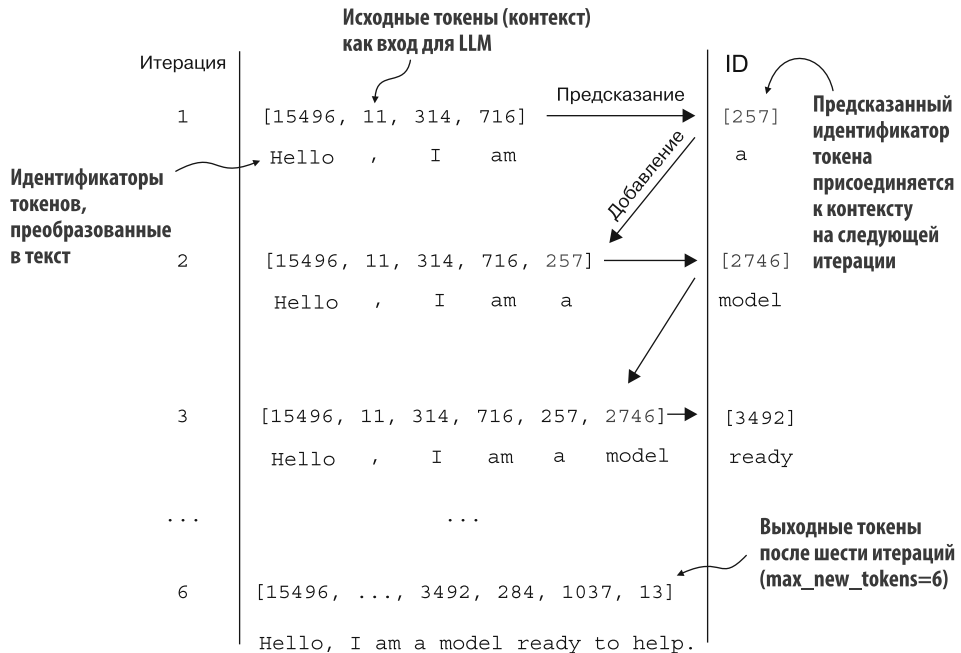


Рис. 4.18. Шесть итераций цикла предсказания токенов

Таким образом, в ходе итераций модель принимает на вход последовательность идентификаторов начальных токенов, предсказывает следующий токен и добавляет его во входную последовательность для следующей итерации. (Кроме того, идентификаторы токенов переводятся в соответствующий текст, чтобы модель могла лучше понимать весь процесс.)

Теперь попробуем функцию `generate_text_simple` с контекстом *Hello, I am* в качестве входных данных модели. Сначала преобразуем входной контекст в идентификаторы токенов:

```
start_context = "Hello, I am"
encoded = tokenizer.encode(start_context)
print("encoded:", encoded)
encoded_tensor = torch.tensor(encoded).unsqueeze(0)
print("encoded_tensor.shape:", encoded_tensor.shape)
```

← Добавляет размерность пакета

Закодированные идентификаторы выглядят так:

```
encoded: [15496, 11, 314, 716]
encoded_tensor.shape: torch.Size([1, 4])
```

Далее мы переводим модель в режим `.eval()`. Это отключает случайные компоненты, такие как отсев, которые используются только во время обучения, и применяем функцию `generate_text_simple` к закодированному входному тензору:

```
model.eval()
out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output:", out)
print("Output length:", len(out[0]))
```

← Отключает отсев в режиме предсказания

Получаемые идентификаторы выходных токенов выглядят так:

```
Output: tensor([[15496, 11, 314, 716, 27018, 24086, 47843, 30961, 42348, 7267]])
Output length: 10
```

Используя метод `.decode` токенизатора, мы можем преобразовать идентификаторы обратно в текст:

```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
```

Выходные данные модели в текстовом формате имеют следующий вид:

```
Hello, I am Featureiman Byeswickattribute argue
```

Как мы видим, модель генерировала тарабарщину, которая совсем не похожа на связный текст *Hello, I am a model ready to help*. Что произошло? Причина, по которой модель не может создавать связный текст, заключается в том, что мы еще не обучили ее. До этого момента мы только реализовали архитектуру GPT и инициализировали экземпляр модели с начальными случайными весами. Обучение модели — большая тема, и мы рассмотрим ее в следующей главе.

Упражнение 4.3. Использование различных параметров отсева

В начале этой главы мы определили глобальный параметр `drop_rate` в словаре `GPT_CONFIG_124M`, чтобы задать коэффициент отсева в различных местах архитектуры `GPTModel`. Измените код, чтобы задать свое значение коэффициента отсева для каждого из слоев отсева в архитектуре модели. (Подсказка: мы использовали слои отсева в трех разных местах — в слое вложения, в слое короткого соединения и в модуле многоцелевого внимания.)

Итоги главы

- Нормализация слоя стабилизирует обучение, гарантируя, что выходные данные каждого слоя имеют постоянное среднее значение и дисперсию.
- Короткими называются соединения, которые пропускают один или несколько слоев, передавая выходные данные одного слоя непосредственно на более глубокий слой, что помогает решить проблему исчезающего градиента при обучении глубоких нейронных сетей, таких как LLM.
- Блоки трансформера — основной структурный компонент моделей GPT, объединяющий модули многоцелевого внимания, содержащие маски, и полносвязные сети прямого распространения, использующие функцию активации GELU.
- Модели GPT представляют собой большие языковые модели с множеством повторяющихся блоков трансформера, насчитывающих от миллионов до миллиардов параметров.
- Модели GPT бывают разных размеров и могут содержать, например, 124, 345, 762 и 1542 млн параметров, которые можно реализовать с помощью одного и того же класса `GPTModel` на Python.
- Способность GPT-подобной LLM генерировать текст заключается в декодировании выходных данных в удобочитаемый текст путем последовательного предсказания одного токена за раз на основе заданного входного контекста.
- Без обучения модель GPT генерирует бессвязный текст. Поэтому важно обучать ее, чтобы она могла выдавать связный текст.

5

Предварительное обучение на неразмеченных данных

В этой главе

- ✓ Вычисление потерь на обучающей и проверочной выборках данных.
- ✓ Реализация функции обучения и предварительное обучение LLM.
- ✓ Сохранение и загрузка весов модели для продолжения обучения LLM.
- ✓ Загрузка предварительно обученных весов из OpenAI.

На данный момент мы реализовали выборку данных и механизм внимания, а также запрограммировали архитектуру LLM. Теперь пришло время реализовать функцию обучения и предварительно обучить LLM. Мы рассмотрим основные методы оценки моделей, позволяющие измерить качество сгенерированного текста, что необходимо для оптимизации LLM в процессе обучения (рис. 5.1). Кроме того, мы обсудим, как загрузить предварительно обученные веса, чтобы дать LLM надежную отправную точку для более тонкой настройки.

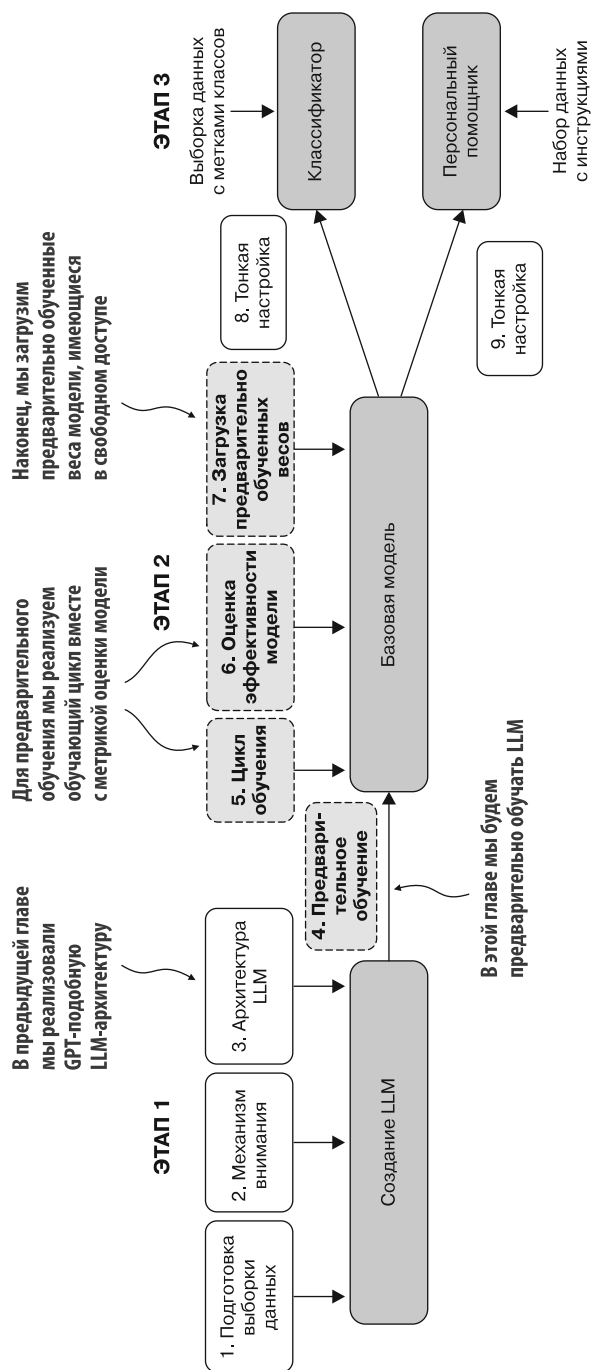


Рис. 5.1. Три основных этапа разработки LLM. В этой главе основное внимание уделяется этапу 2: предварительному обучению LLM (шаг 4), в которое входит реализация кода обучения (шаг 5), оценка качества обучения (шаг 6), сохранение и загрузка весов модели (шаг 7)

Параметры весовых коэффициентов

В контексте LLM и других моделей глубокого обучения весовые коэффициенты относятся к обучаемым параметрам, которые настраиваются (изменяются) в процессе обучения. Эти коэффициенты также известны как *весовые параметры* (weight parameters) или просто *параметры* (parameters). В таких фреймворках, как PyTorch, данные коэффициенты хранятся в линейных слоях; мы использовали их для реализации многоцелевого модуля внимания в главе 3 и GPTModel в главе 4. После инициализации слоя (`new_layer = torch.nn.Linear(...)`) мы можем получить доступ к его весам с помощью атрибута `.weight`, `new_layer.weight`. Кроме того, чтобы сделать работу с моделью более удобной, PyTorch позволяет напрямую получить доступ ко всем ее обучаемым параметрам, в том числе весам и смещениям, с помощью метода `model.parameters()`, который мы будем использовать позже при обучении модели.

5.1. Оценка генеративных текстовых моделей

В главе 4 был представлен краткий обзор генерации текста. В текущей же главе мы настроим нашу LLM, а затем обсудим основные способы оценки качества сгенерированного текста. Затем мы рассчитаем потери при обучении и проверке. На рис. 5.2 показано, что мы рассмотрим в этой главе.

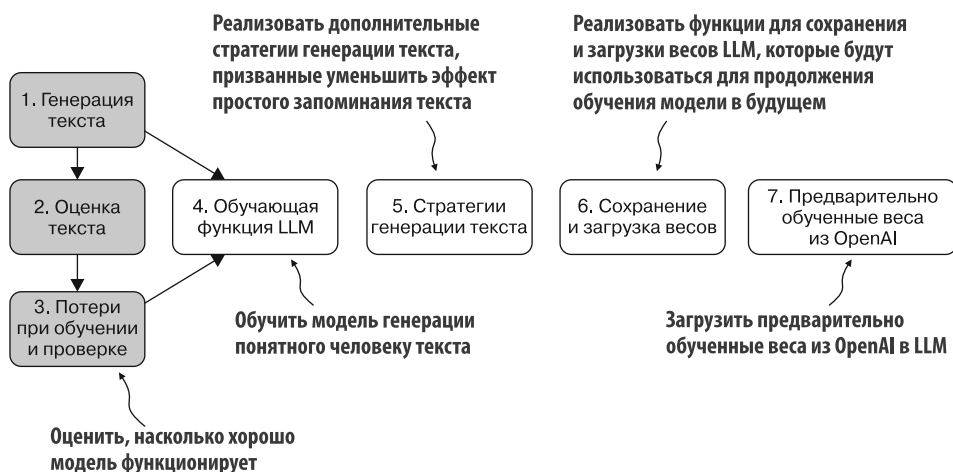


Рис. 5.2. Обзор тем этой главы. Мы начнем с краткого описания генерации текста (шаг 1), а затем перейдем к обсуждению основных методов оценки моделей (шаг 2) и потерь при обучении и проверке (шаг 3)

5.1.1. Использование GPT для генерации текста

Перейдем к настройке LLM и вкратце вспомним процесс генерации текста, который мы реализовали в главе 4. Начнем с инициализации модели GPT, которую позже оценим и обучим с помощью класса `GPTModel` и словаря `GPT_CONFIG_124M` (см. главу 4):

```
import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 256,  # Мы укорачиваем длину
                             # контекста с 1024 до 256 токенов
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,  # Общепринято задавать
                       # нулевое значение отсева
    "qkv_bias": False
}
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval()
```

Если посмотреть на словарь `GPT_CONFIG_124M`, то единственное изменение, которое мы внесли по сравнению с предыдущей главой, заключается в том, что мы уменьшили длину контекста (`context_length`) до 256 токенов. Это изменение снижает вычислительные затраты на обучение модели, позволяя проводить обучение на обычном компьютере.

Первоначально модель GPT-2 с 124 млн параметров была сконфигурирована для обработки 1024 токенов. После завершения процесса обучения мы обновим настройку размера контекста и загрузим предварительно обученные веса для работы с моделью, настроенной на длину контекста 1024 токена.

Используя экземпляр `GPTModel`, мы применяем функцию `generate_text_simple` из главы 4 и вводим две удобные функции: `text_to_token_ids` и `token_ids_to_text`. Они упрощают преобразование текста в токены и токенов в текст, которые мы будем использовать в этой главе.

На рис. 5.3 показан трехшаговый процесс генерации текста с помощью модели GPT. Сначала токенизатор преобразует входной текст в последовательность идентификаторов токенов (см. главу 2). Затем эти идентификаторы токенов поступают на вход модели, генерирующей соответствующие логиты, которые представляют собой векторы, описывающие распределение вероятностей для каждого токена в словаре (см. главу 4). Далее эти логиты преобразуются обратно в идентификаторы токенов, которые токенизатор декодирует в понятный человеку текст, завершая цикл от текстового ввода до текстового вывода.

1. Токенизатор используется для кодирования исходного текста в идентификаторы токенов

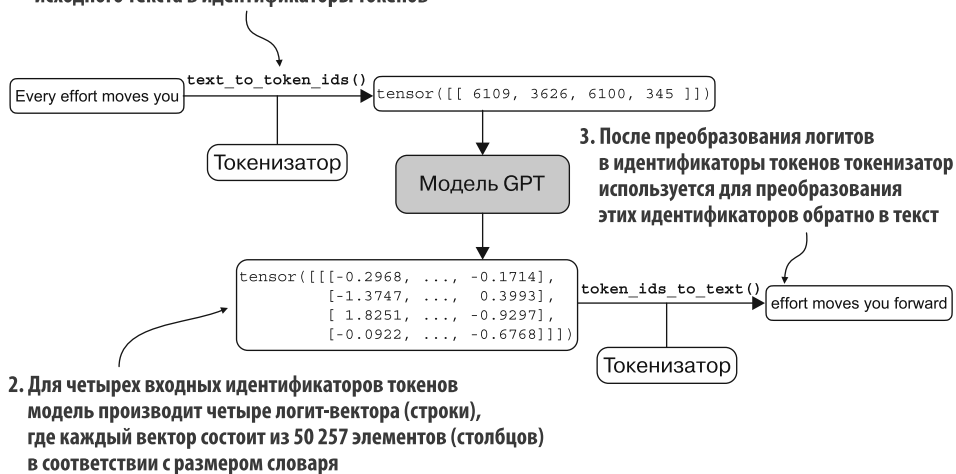


Рис. 5.3. Генерация текста включает в себя кодирование текста в виде идентификаторов токенов, которые LLM преобразует в логит-векторы. Затем эти векторы преобразуются обратно в идентификаторы токенов и детокенизируются в текстовое представление

Процесс генерации текста реализован в листинге 5.1.

Листинг 5.1. Служебные функции для преобразования текста в идентификатор токена

```

import tiktoken
from chapter04 import generate_text_simple

def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
    encoded_tensor = torch.tensor(encoded).unsqueeze(0)
    return encoded_tensor

def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0)
    return tokenizer.decode(flat.tolist())

start_context = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(start_context, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))

```

.unsqueeze(0) добавляет размерность пакета

Удаляет размерность пакета

Используя данный код, модель производит следующий текст:

Output text:

```
Every effort moves you rentingetic wasn? refres RexMcHicular stren
```

Очевидно, что модель еще не может создавать связный текст, поскольку не прошла обучение. Чтобы определить, что делает текст «связным» или «высококачественным», мы должны реализовать метод, позволяющий оценивать сгенерированный контент. Такой подход позволит отслеживать и улучшать качество модели на протяжении всего процесса обучения.

Далее мы рассчитаем *метрику потерь* (loss metric) для сгенерированных результатов. Она служит показателем прогресса и успешности обучения. Кроме того, в последующих главах, когда мы будем дорабатывать LLM, мы рассмотрим дополнительные методы оценки качества модели.

5.1.2. Расчет потерь при генерации текста

Далее мы рассмотрим методы численной оценки качества текста, сгенерированного во время обучения, путем *расчета потерь* (text generation loss). Мы рассмотрим практический пример и начнем с краткого обзора того, как загружаются данные и как генерируется текст с помощью функции `generate_text_simple`.

На рис. 5.4 представлен общий процесс преобразования входного текста в текст, сгенерированный LLM. Он состоит из пяти шагов и показывает, что делает функция `generate_text_simple`. Нам нужно выполнить те же начальные шаги, прежде чем мы сможем вычислить потери, которые измеряют качество сгенерированного текста.

Итак, для каждого из трех входных токенов, показанных слева, вычисляется вектор, содержащий оценки вероятности, соответствующие каждому токenu в словаре. Позиция с наивысшей оценкой вероятности в каждом векторе представляет собой наиболее вероятный идентификатор следующего токена. Эти идентификаторы токенов, связанные с наивысшими оценками вероятности, выбираются и преобразуются обратно в текст, представляющий собой текст, сгенерированный моделью.

Чтобы уместить все изображение на одной странице, на рис. 5.4 показан процесс генерации текста с маленьким словарным запасом из семи токенов. Однако наша модель GPT работает с гораздо большим словарем, состоящим из 50 257 слов; следовательно, идентификаторы токенов в следующем коде будут находиться в диапазоне от 0 до 50 256, а не от 0 до 6.

Кроме того, на рис. 5.4 для простоты показан только один текстовый пример ("every effort moves"). В следующем практическом примере кода, реализующем шаги, показанные на данном рисунке, мы будем работать с двумя входными примерами для модели GPT ("every effort moves" и "I really like").

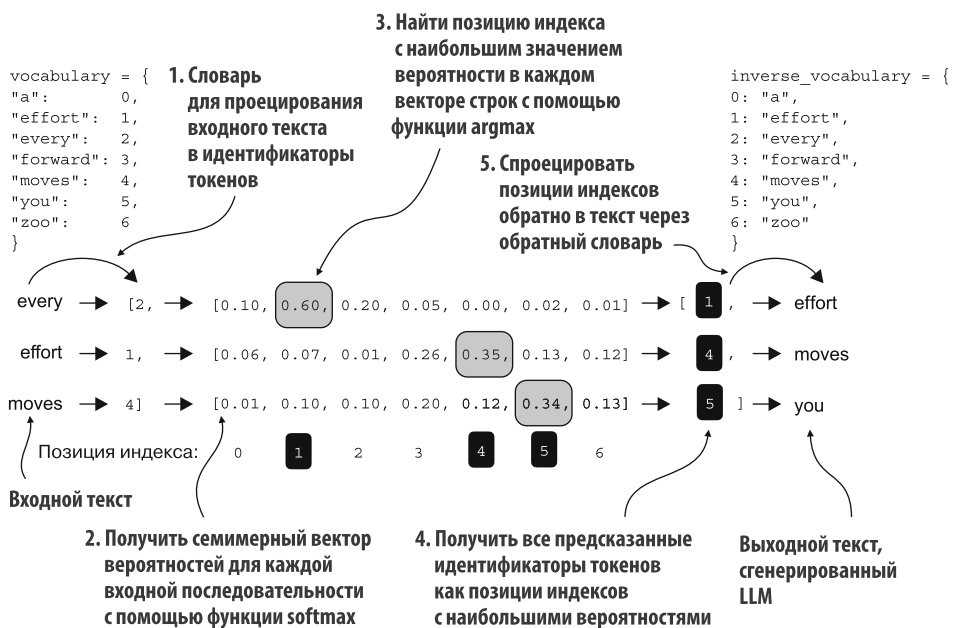


Рис. 5.4. Общий процесс преобразования входного текста в текст, сгенерированный LLM

Рассмотрим эти два входных примера, которые уже сопоставлены с идентификаторами токенов (рис. 5.4, шаг 1):

```

inputs = torch.tensor([[16833, 3626, 6100], # ["every effort moves",
                      [40, 1107, 588]]) # "I really like"]

```

В соответствии с этими входными данными целевые значения содержат идентификаторы токенов, которые мы хотим получить от модели:

```

targets = torch.tensor([[3626, 6100, 345 ], # [" effort moves you",
                        [1107, 588, 11311]]) # " really like chocolate"]

```

Обратите внимание, что целевые значения — это входные данные, но сдвинутые на одну позицию вперед. Эту концепцию мы рассмотрели в главе 2 при реализации загрузчика данных. Стратегия смещения имеет решающее значение для обучения модели предсказанию следующего токена в последовательности.

Теперь мы передаем входные данные в модель, чтобы вычислить векторы логитов для двух входных примеров, каждый из которых состоит из трех токенов. Затем мы применяем функцию `softmax` для преобразования этих логитов в оценки вероятности (`probas`; рис. 5.4, шаг 2):

```
with torch.no_grad():
    logits = model(inputs)
    probas = torch.softmax(logits, dim=-1)
    print(probas.shape)
```

← Отключает отслеживание градиента,
так как мы еще не тренируем модель

← Вероятность каждого
токена в словаре

Результирующая размерность тензора оценки вероятности (`probas`) равна:
`torch.Size([2, 3, 50257])`

Первое число, 2, соответствует двум примерам (строкам) во входных данных, также известным как размер пакета. Второе число, 3, соответствует количеству токенов в каждом входном примере (строке). Наконец, последнее число, 50257, соответствует размерности вложения, которая определяется размером словаря. После преобразования логитов в вероятности с помощью функции `softmax` функция `generate_text_simple` преобразует полученные оценки вероятности обратно в текст (рис. 5.4, шаги 3–5).

Мы можем выполнить шаги 3 и 4, применив функцию `argmax` к показателям вероятности, чтобы получить соответствующие идентификаторы токенов:

```
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Token IDs:\n", token_ids)
```

Учитывая, что у нас есть две входные выборки, каждая из которых содержит по три токена, применение функции `argmax` к оценкам вероятности (рис. 5.4, шаг 3) позволяет получить два набора результатов, каждый из которых содержит по три прогнозируемых идентификатора токенов:

```
Token IDs:
tensor([[[16657],
         [ 339],
         [42826]],
        [[49906],
         [29669],
         [41751]]])
```

← Первый пакет

← Второй пакет

Наконец, на шаге 5 идентификаторы токенов преобразуются обратно в текст:

```
print(f"Targets batch 1: {token_ids_to_text(targets[0], tokenizer)}")
print(f"Outputs batch 1:"
      f" {token_ids_to_text(token_ids[0].flatten(), tokenizer)}")
```

Декодируя эти токены, мы обнаруживаем, что выходные токены сильно отличаются от целевых токенов, которые, как мы рассчитываем, должна сгенерировать модель:

```
Targets batch 1: effort moves you
Outputs batch 1: Armed heNetflix
```

Модель генерирует случайный текст, который отличается от целевого, поскольку еще не обучена. Теперь мы хотим оценить качество ее работы с помощью функции потерь (рис. 5.5). Это не только полезно для измерения качества сгенерированного текста, но и является основой для реализации функции обучения, которую мы будем использовать для обновления весов модели, чтобы улучшить сгенерированный текст.

Реализовать функцию потерь, чтобы оценить, насколько хорошо функционирует модель

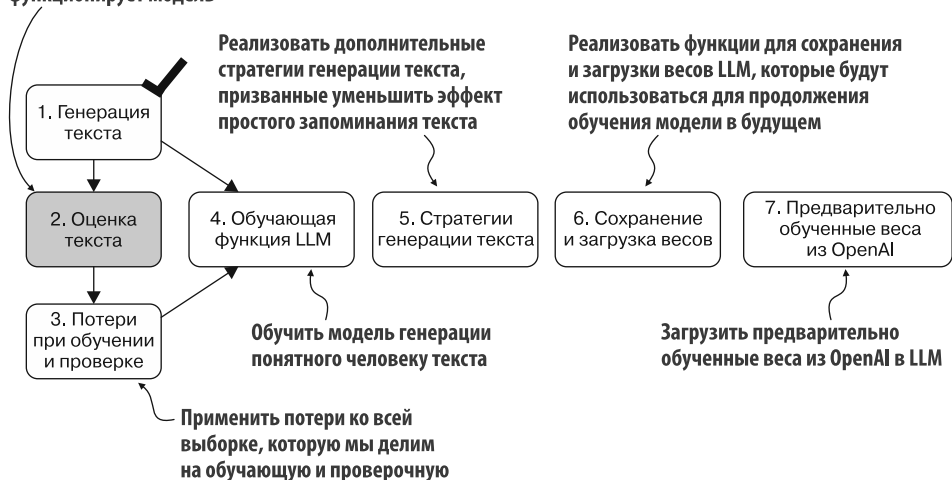


Рис. 5.5. Темы, рассматриваемые в этой главе. Мы завершили шаг 1. Теперь мы готовы реализовать функцию оценки текста (шаг 2)

Часть процесса оценки текста, которую мы реализуем по схеме, показанной на рис. 5.5, заключается в измерении «степени отклонения» сгенерированных токенов от правильных предсказаний (целей). Функция обучения, которую мы реализуем позже, будет использовать эту информацию для корректировки весов модели, чтобы генерировать текст, который больше похож на целевой (или, в идеале, соответствует ему).

Обучение модели направлено на увеличение вероятности `softmax` в позициях индекса, соответствующих правильным идентификаторам целевых токенов (рис. 5.6). Эта вероятность `softmax` также используется в метрике оценки, которую мы реализуем далее, чтобы численно оценить сгенерированные моделью результаты: чем выше вероятность в правильных позициях, тем лучше.

Помните: чтобы уместить изображение на одной странице, на рис. 5.6 показаны вероятности `softmax` для маленького словаря из семи токенов. Это означает, что начальные случайные значения будут колебаться в диапазоне $1 / 7$, что

примерно равно 0,14. Однако в словаре, который мы используем для нашей модели GPT-2, 50 257 токенов, поэтому большая часть начальных вероятностей будет около 0,00002 ($1 / 50\,257$).

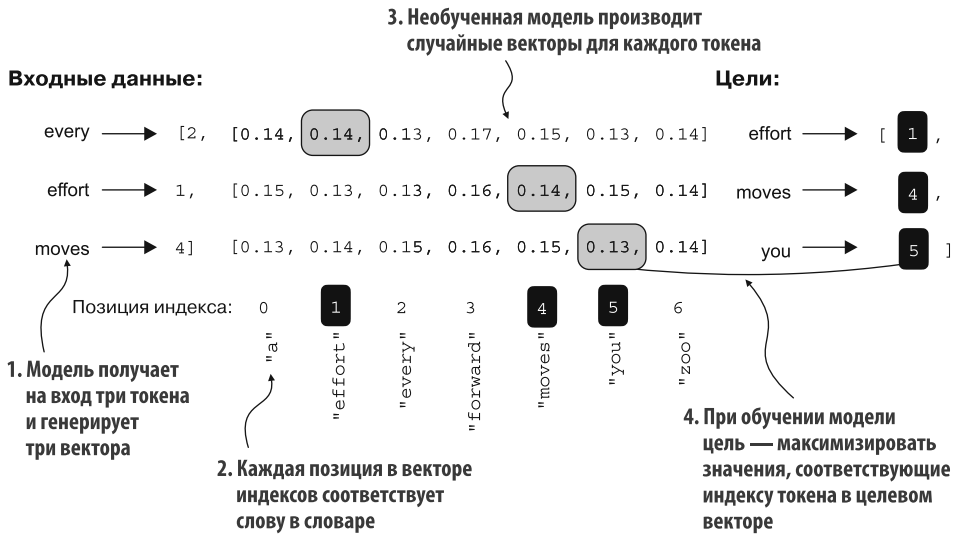


Рис. 5.6. Перед обучением модель генерирует случайные векторы вероятностей для следующего токена. Цель обучения модели — добиться того, чтобы значения вероятностей, соответствующие выделенным целевым идентификаторам токенов, были максимальными

Для каждого из двух входных текстов мы можем вывести начальные значения вероятности `softmax`, соответствующие целевым токенам, используя следующий код:

```
text_idx = 0
target_probab_1 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 1:", target_probab_1)

text_idx = 1
target_probab_2 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 2:", target_probab_2)
```

Тремя вероятностями для каждого пакета являются:

```
Text 1: tensor([7.4541e-05, 3.1061e-05, 1.1563e-05])
Text 2: tensor([1.0337e-05, 5.6776e-05, 4.7559e-06])
```

Цель обучения LLM — максимизировать вероятность правильного токена, что подразумевает увеличение его вероятности по сравнению с другими токенами. Таким образом, мы гарантируем, что модель последовательно выбирает целевой

токен — по сути, следующее слово в предложении — в качестве следующего генерируемого токена.

Обратное распространение ошибки

Как максимизировать значения вероятности softmax, соответствующие целевым токенам? Если в общих чертах, то мы обновляем веса модели так, чтобы модель выдавала более высокие значения для соответствующих токенов, которые мы хотим сгенерировать. Обновление весов выполняется с помощью процесса, называемого *обратным распространением ошибки (backpropagation)*, — стандартной методики обучения глубоких нейронных сетей (более подробную информацию см. в разделах A.3–A.7 приложения A).

Для обратного распространения ошибки требуется функция потерь, которая вычисляет разницу между предсказанным результатом модели (в данном случае — вероятностями, соответствующими идентификаторам целевых токенов) и фактическим желаемым результатом. Эта функция измеряет, насколько предсказания модели отличаются от целевых значений.

Далее мы рассчитаем потери для показателей вероятности двух примеров: `target_probab_1` и `target_probab_2`. Основные шаги показаны на рис. 5.7. Мы уже выполнили шаги 1–3, чтобы получить `target_probab_1` и `target_probab_2`, поэтому переходим к шагу 4, применяя логарифм к показателям вероятности:

```
log_probab = torch.log(torch.cat((target_probab_1, target_probab_2)))
print(log_probab)
```

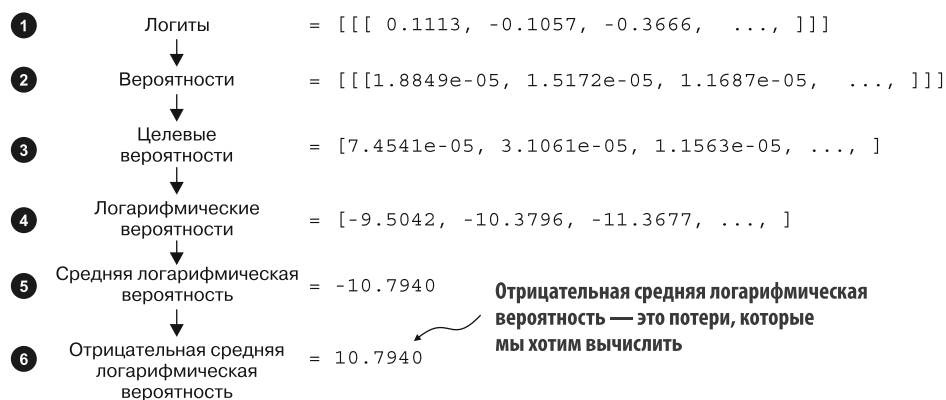


Рис. 5.7. Вычисление потерь состоит из нескольких шагов. На шагах 1–3, которые мы уже выполнили, вычисляются вероятности токенов, соответствующие целевым тензорам. Затем эти вероятности преобразуются с помощью логарифма и усредняются на шагах 4–6

В результате мы получаем следующие значения:

```
tensor([ -9.5042, -10.3796, -11.3677, -11.4798, -9.7764, -12.2561])
```

В математической оптимизации удобнее работать с логарифмами вероятностей, чем с самими вероятностями. Эта тема выходит за рамки данной книги, но я подробно рассмотрел ее в приложении Б.

Далее мы объединяем эти логарифмы вероятностей в один показатель, вычисляя среднее значение (шаг 5 на рис. 5.7):

```
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)
```

Итоговая оценка средней логарифмической вероятности такова:

```
tensor(-10.7940)
```

Цель состоит в том, чтобы приблизить среднюю логарифмическую вероятность к 0, обновляя веса модели в процессе обучения. Однако в глубоком обучении обычно стремятся к тому, чтобы нулю была равна не средняя логарифмическая вероятность, а отрицательная средняя логарифмическая вероятность. Это просто средняя логарифмическая вероятность, умноженная на -1 , что соответствует шагу 6 на рис. 5.7:

```
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas)
```

В результате выводится тензор (10.7940). В глубоком обучении термин, обозначающий преобразование этого отрицательного значения -10.7940 в 10.7940 , известен как потери *перекрестной энтропии* (cross-entropy). Здесь библиотека PyTorch оказывается полезна, так как в ней уже есть встроенная функция `cross_entropy`, которая выполняет все эти шесть шагов на рис. 5.7 за нас.

Потери перекрестной энтропии

По сути, потери перекрестной энтропии — популярная мера в машинном и глубоком обучении, которая определяет разницу между двумя распределениями вероятностей — как правило, истинным распределением меток (в данном случае токенов в наборе данных) и прогнозируемым распределением, полученным с помощью модели (например, вероятностей токенов, сгенерированных LLM).

В контексте машинного обучения и, в частности, в таких фреймворках, как PyTorch, функция `cross_entropy` вычисляет эту меру для дискретных результатов, которая аналогична отрицательной средней логарифмической вероятности целевых токенов с учетом вероятностей токенов, сгенерированных моделью. Таким образом, термины «перекрестная энтропия» и «отрицательная средняя логарифмическая вероятность» взаимосвязаны и на практике часто используются как взаимозаменяемые.

Прежде чем применять функцию `cross_entropy`, коротко вспомним размеры тензоров логитов и целевых тензоров:

```
print("Logits shape:", logits.shape)
print("Targets shape:", targets.shape)
```

Они равны:

```
Logits shape: torch.Size([2, 3, 50257])
Targets shape: torch.Size([2, 3])
```

Как мы видим, тензор логитов имеет три размерности: размер пакета, количество токенов и размер словаря. Тензор целей имеет две размерности: размер пакета и количество токенов.

Для функции потерь `cross_entropy` в PyTorch мы хотим «расплющить» эти тензоры, объединив их по размеру пакета:

```
logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()
print("Flattened logits:", logits_flat.shape)
print("Flattened targets:", targets_flat.shape)
```

Получающиеся размерности тензоров таковы:

```
Flattened logits: torch.Size([6, 50257])
Flattened targets: torch.Size([6])
```

Помните, что целевые значения — это идентификаторы токенов, которые мы хотим сгенерировать с помощью LLM, а логиты содержат немасштабированные выходные данные модели до того, как они попадут в функцию `softmax` для получения оценок вероятности.

Ранее мы применяли функцию `softmax`, выбирали оценки вероятности, соответствующие целевым идентификаторам, и вычисляли отрицательные средние логарифмы вероятностей. Функция `cross_entropy` в PyTorch выполнит все эти действия за нас:

```
loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)
print(loss)
```

В результате мы получим те же потери, что и при выполнении отдельных шагов, показанных на рис. 5.7:

```
tensor(10.7940)
```

Мы только что рассчитали потери для двух небольших текстовых входных данных и сделали это в демонстрационных целях. Далее мы применим расчет потерь ко всем данным из обучающей и проверочной выборок.

Затруднение

Затруднение — показатель, часто используемый наряду с потерями перекрестной энтропии для оценки эффективности моделей в таких задачах, как языковое моделирование. Он может обеспечить более интуитивное понимание неопределенности модели при прогнозировании следующего токена в последовательности.

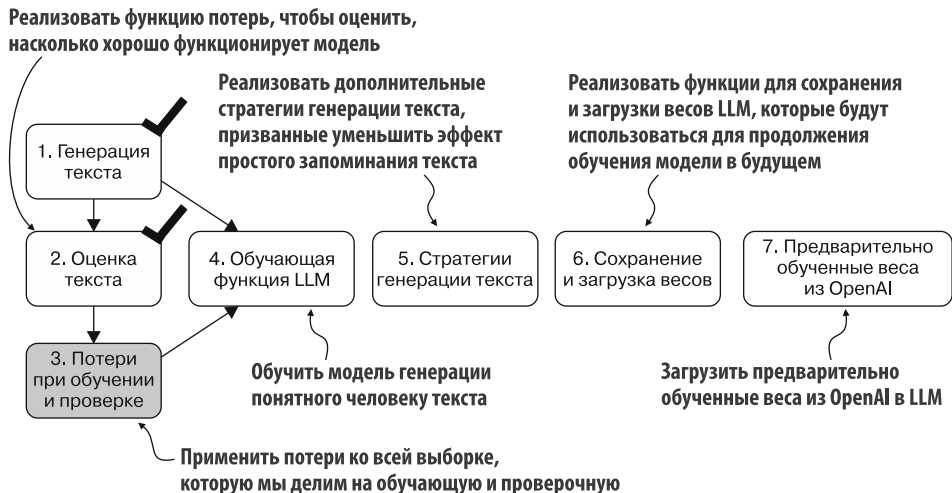
Затруднение измеряет, насколько хорошо распределение вероятностей, предсказанное моделью, соответствует фактическому распределению слов в наборе данных. Как и в случае с потерей, меньшее затруднение указывает на то, что предсказания модели ближе к фактическому распределению.

Затруднение можно вычислить по формуле `perplexity = torch.exp(loss)`, которая возвращает `tensor(48725.8203)` при применении к ранее вычисленному значению потерь.

Затруднение часто считается более интерпретируемым, чем исходное значение потерь, поскольку показывает эффективный размер словаря, в котором модель не уверена на каждом шаге. В приведенном примере это означает, что модель не уверена в том, какой из 48 725 токенов в словаре следует сгенерировать в качестве следующего токена.

5.1.3. Расчет потерь для обучающей и проверочной выборок

Сначала мы должны подготовить обучающие и проверочные выборки, которые будем использовать для обучения LLM. Затем рассчитаем перекрестную энтропию для этих выборок (рис. 5.8), что необходимо для процесса обучения модели.



Чтобы вычислить потери в обучающем и проверочном наборах данных, мы используем очень маленький текстовый набор данных — рассказ *The Verdict* Эдит Уортон, с которым уже работали в главе 2. Выбрав общедоступный текст, мы избежим любых проблем, связанных с правами собственности. Кроме того, использование такого небольшого набора данных позволяет выполнять примеры кода на стандартном ноутбуке за считанные минуты, даже без мощного графического процессора, что особенно ценно для образовательных целей.

ПРИМЕЧАНИЕ Заинтересованные читатели также могут использовать дополнительный код для этой книги, чтобы подготовить более масштабный набор данных, состоящий из более чем 60 000 книг из общедоступного проекта «Гутенберг», и обучить на них LLM (подробнее см. в приложении Г).

Стоимость предварительного обучения LLM

Чтобы представить масштаб нашего проекта в перспективе, рассмотрим обучение модели Llama 2 с 7 млрд параметров, относительно популярной общедоступной LLM. Для этой модели потребовалось 184 320 часов работы на дорогостоящих графических процессорах A100 для обработки 2 трлн токенов. На момент написания книги запуск облачного сервера 8 × A100 на AWS стоил около 30 долларов в час. По приблизительным оценкам, общая стоимость обучения такой LLM составляет около 691 200 долларов (184 320 часов делим на 8, а затем умножаем на 30).

Следующий код загружает короткий рассказ *The Verdict*:

```
file_path = "the-verdict.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text_data = file.read()
```

После загрузки набора данных мы можем проверить количество символов и токенов в наборе:

```
total_characters = len(text_data)
total_tokens = len(tokenizer.encode(text_data))
print("Characters:", total_characters)
print("Tokens:", total_tokens)
```

На выходе имеем:

```
Characters: 20479
Tokens: 5145
```

При наличии всего 5145 токенов текст может показаться слишком маленьким для обучения LLM, но, как упоминалось ранее, он предназначен для образовательных целей, чтобы мы могли завершить вычисления за несколько минут,

а не за несколько недель. Кроме того, позже мы загрузим предварительно обученные веса из OpenAI в наш код `GPTModel`.

Далее мы разделим набор данных на обучающую и проверочную выборки и используем загрузчики данных из главы 2, чтобы подготовить пакеты для обучения LLM. Этот процесс представлен на рис. 5.9. Из-за нехватки места на странице мы используем `max_length=6`. Однако для реальных загрузчиков данных мы устанавливаем `max_length` равным 256 токенам, чтобы LLM видела более длинные тексты во время обучения.

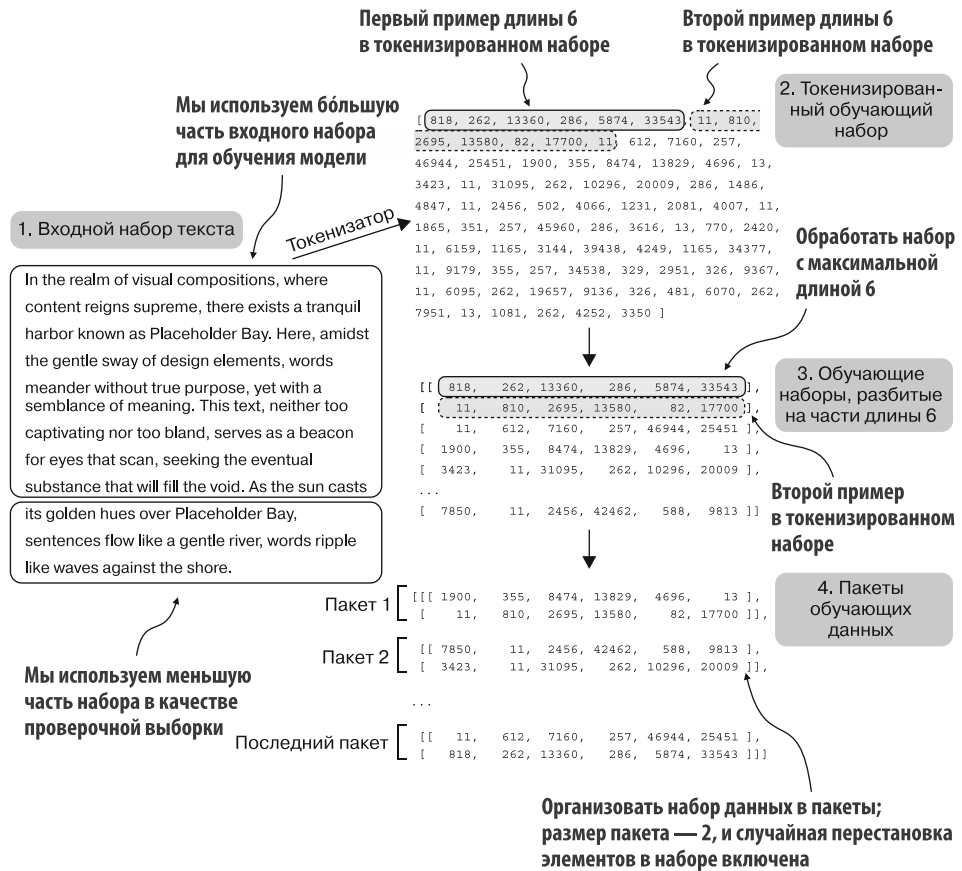


Рис. 5.9. Подготовка загрузчиков данных

Итак, при подготовке загрузчиков мы разделяем входной текст на части для обучения и проверки. Затем токенизируем текст (для простоты показано только

для части, предназначенной для обучения) и делим токенизированный текст на фрагменты заданной пользователем длины (здесь — 6). Наконец, мы перемешиваем строки и организуем фрагментированный текст в пакеты (здесь размер пакета — 2), которые мы можем использовать для обучения модели.

ПРИМЕЧАНИЕ Для простоты и эффективности мы обучаем модель на данных, представленных в виде фрагментов одинакового размера. Однако на практике может быть полезно обучать LLM на входных данных переменной длины, чтобы модель лучше адаптировалась к различным типам входных данных.

Чтобы разделить и загрузить данные, мы сначала определяем `train_ratio`, чтобы использовать 90 % данных как основной материал для обучения, а оставшиеся 10 % — в качестве проверочных данных, с помощью которых можно будет оценить модель во время обучения:

```
train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

Используя подмножества `train_data` и `val_data`, мы можем создать соответствующий загрузчик данных, используя код `create_dataloader_v1` из главы 2:

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123)

train_loader = create_dataloader_v1(
    train_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    val_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

Мы использовали относительно небольшой размер пакета, чтобы снизить требования к вычислительным ресурсам, поскольку работали с очень небольшим

набором данных. На практике обучение LLM с размером пакета 1024 или больше не является чем-то необычным.

В качестве дополнительной проверки мы можем перебрать загрузчики данных, чтобы убедиться, что они были созданы правильно:

```
print("Train loader:")
for x, y in train_loader:
    print(x.shape, y.shape)

print("\nValidation loader:")
for x, y in val_loader:
    print(x.shape, y.shape)
```

Мы увидим следующие результаты:

```
Train loader:
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
torch.Size([2, 256]) torch.Size([2, 256])
Validation loader:
torch.Size([2, 256]) torch.Size([2, 256])
```

Исходя из предыдущего вывода кода, у нас есть девять пакетов обучающих данных с двумя примерами и 256 токенами в каждом. Мы выделили только 10 % данных для проверки, поэтому есть только один пакет для проверки, состоящий из двух входных примеров. Как и ожидалось, входные данные (x) и целевые данные (y) имеют одинаковую форму (размер пакета, умноженный на количество токенов в каждом пакете), поскольку целевые данные — это входные данные, сдвинутые на одну позицию, как описано в главе 2.

Далее мы реализуем вспомогательную функцию, которая позволит вычислить потери перекрестной энтропии для заданного пакета, возвращаемого загрузчиком для обучения и проверки:

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)
    logits = model(input_batch)
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1), target_batch.flatten()
    )
    return loss
```

Перенос на данное устройство позволяет переслать данные графическому процессору

Теперь мы можем с помощью этой служебной функции `calc_loss_batch`, которая вычисляет потери для одной выборки, реализовать следующую функцию `calc_loss_loader`, вычисляющую потери для всех выборок, загруженных этим загрузчиком данных (листинг 5.2).

Листинг 5.2. Функция для вычисления потерь при обучении модели и ее проверке

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches
```

Если `num_batches` не определена, то проходить пакет за пакетом

Уменьшает количество пакетов до общего количества пакетов в загрузчике данных, если `num_batches` превышает количество пакетов в загрузчике данных

Суммы потерь для каждого пакета

Усредняет потери по всем пакетам

По умолчанию функция `calc_loss_loader` выполняет итерацию по всем пакетам в этом загрузчике данных, накапливает потери в переменной `total_loss`, а затем вычисляет и усредняет потери по всем пакетам. В качестве альтернативы мы можем указать меньшее количество пакетов с помощью `num_batches`, чтобы ускорить вычисление во время обучения модели.

Теперь посмотрим на функцию `calc_loss_loader` в действии, применив ее к загрузчикам обучающих и проверочных выборок:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
with torch.no_grad():
    train_loss = calc_loss_loader(train_loader, model, device)
    val_loss = calc_loss_loader(val_loader, model, device)
print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```

Если у вас есть компьютер с графическим процессором, поддерживающим CUDA, то LLM будет обучаться на графическом процессоре, при этом в код не будут вноситься никакие изменения

Отключает отслеживание градиента для повышения эффективности, поскольку мы еще не тренируем модель

С помощью настройки переменной `device` мы гарантируем, что данные будут загружены на то же устройство, что и LLM

Значения потерь таковы:

Training loss: 10.98758347829183
Validation loss: 10.98110580444336

Они относительно высоки, поскольку модель еще не обучена. Для сравнения, потери приближаются к 0, если она учится генерировать следующие токены по мере их появления в обучающей и проверочной выборках.

Теперь, когда у нас есть способ измерить качество сгенерированного текста, мы обучим LLM, чтобы уменьшить потери и улучшить генерацию текста (рис. 5.10).

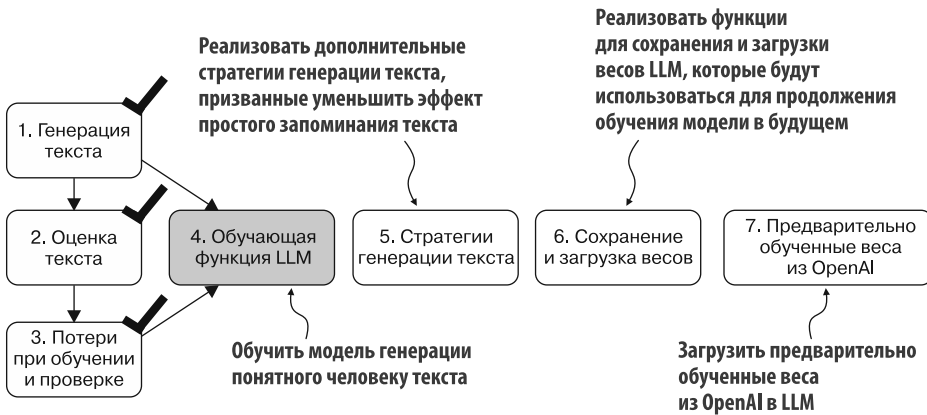


Рис. 5.10. Мы описали процесс генерации текста (шаг 1) и внедрили базовые методы оценки модели (шаг 2), позволяющие вычислить потери на обучающей и проверочной выборках (шаг 3). Далее мы перейдем к функциям обучения и предварительному обучению LLM (шаг 4)

Далее мы сосредоточимся на предварительном обучении LLM. Затем мы реализуем альтернативные стратегии генерации текста, а также сохраним и загрузим веса предварительно обученной модели.

5.2. Обучение LLM

Наконец-то пришло время реализовать код для предварительного обучения LLM, нашей `GPTModel1`. Для этого мы сосредоточимся на простом цикле обучения, чтобы код был лаконичным и понятным.

ПРИМЕЧАНИЕ Заинтересованные читатели могут узнать о более сложных методах, таких как прогрев скорости обучения (learning rate warmup), косинусный отжиг (cosine annealing) и обрезка градиента (gradient clipping), в приложении Г.

На рис. 5.11 показан типичный восьмишаговый процесс обучения нейронной сети в PyTorch, который мы применяем для обучения LLM.

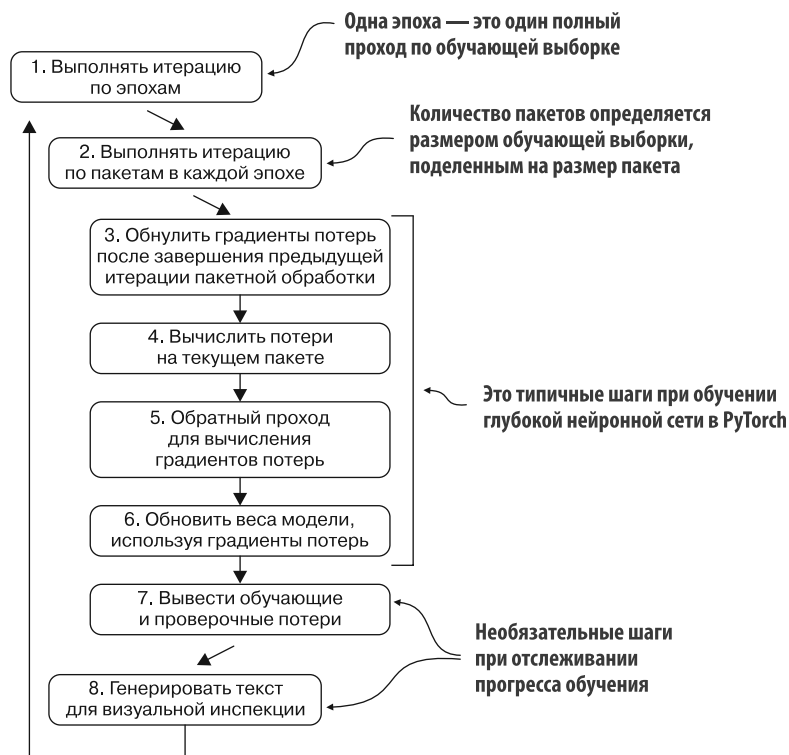


Рис. 5.11. Типичный цикл обучения для тренировки глубоких нейронных сетей в PyTorch

Таким образом, цикл состоит из множества шагов, в течение которых мы перебираем пакеты из обучающей выборки один за другим на протяжении нескольких эпох. На каждой итерации мы вычисляем потери для каждого пакета из обучающей выборки, чтобы вычислить градиенты потерь, которые мы используем для обновления весов модели так, чтобы потери на обучающей выборке были минимальными.

ПРИМЕЧАНИЕ Если вы относительно недавно начали изучать глубокие нейронные сети с помощью PyTorch и какие-то из этих шагов вам не знакомы, то я советую обратиться к разделам A.5–A.8 приложения A.

Мы можем реализовать этот процесс обучения с помощью функции `train_model_simple` (листинг 5.3).

Листинг 5.3. Основная функция для предварительного обучения LLM

```
def train_model_simple(model, train_loader, val_loader,
                       optimizer, device, num_epochs,
                       eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()
            optimizer.step()
            tokens_seen += input_batch.numel()
            global_step += 1

            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                    f"Train loss {train_loss:.3f}, "
                    f"Val loss {val_loss:.3f}")

        generate_and_print_sample(
            model, tokenizer, device, start_context
        )
    return train_losses, val_losses, track_tokens_seen
```

Инициализирует списки для отслеживания потерь и встреченных токенов

Начинает основной цикл обучения

Обнуляет градиенты потерь после завершения предыдущей итерации пакетной обработки

Вычисляет градиенты потерь

Обновляет веса модели, используя градиенты потерь

Необязательный шаг оценки

Выводит на печать текст после каждой эпохи

Обратите внимание: созданная только что функция `train_model_simple` использует две функции, которые мы еще не определили: `evaluate_model` и `generate_and_print_sample`.

Функция `evaluate_model` соответствует шагу 7 на рис. 5.11. Она выводит информацию о потерях на обучающей и проверочной выборках после каждого обновления модели, чтобы мы могли оценить, улучшает ли обучение модель. Если говорить более конкретно, эта функция вычисляет потери на обучающей и проверочной выборках, гарантируя, что модель находится в режиме оценки,

и при этом отслеживание градиента и отсев при вычислении потерь на этих выборках отключены:

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_loss = calc_loss_loader(
            val_loader, model, device, num_batches=eval_iter
        )
    model.train()
    return train_loss, val_loss
```

Отсев отключен при оценке, чтобы полученные результаты были стабильными и воспроизводимыми

Отключает отслеживание градиента, которое не требуется при оценке, тем самым сокращая вычислительные затраты

Как и `evaluate_model`, функция `generate_and_print_sample` полезна для отслеживания улучшения модели во время обучения. В частности, она принимает на вход фрагмент текста (`start_context`), преобразует его в идентификаторы токенов и передает в LLM, чтобы та могла создать текстовый образец с помощью функции `generate_text_simple`, которую мы использовали ранее:

```
def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
    decoded_text = token_ids_to_text(token_ids, tokenizer)
    print(decoded_text.replace("\n", " "))
    model.train()
```

Компактный формат вывода результатов

В то время как функция `evaluate_model` выдает оценку прогресса в обучении модели, функция `generate_and_print_sample_text` предоставляет конкретный текстовый пример, сгенерированный моделью, на основе которого можно оценить ее возможности во время обучения.

AdamW

При обучении глубоких нейронных сетей часто используются оптимизаторы *Adam*. Однако в нашем цикле обучения мы используем оптимизатор *AdamW*. Это вариант алгоритма *Adam*, улучшающий подход к постепенному уменьшению весов сети, направленный на минимизацию сложности модели и предотвращение переобучения за счет ограничения неконтролируемого роста весовых значений. Такая корректировка позволяет алгоритму *AdamW* обеспечивать более эффективную регуляризацию и лучшее обобщение обученной сети, поэтому алгоритм *AdamW* часто используется при обучении больших языковых моделей.

Посмотрим, как это работает, обучив экземпляр `GPTModel` за десять эпох с помощью оптимизатора `AdamW` и функции `train_model_simple`, которую мы определили ранее:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenizer
)
```

Метод `.parameters()` возвращает
все обучаемые параметры модели

Выполнение функции `train_model_simple` запускает процесс обучения, который занимает около пяти минут на MacBook Air или аналогичном ноутбуке. Во время выполнения этой функции выводится следующая информация:

```
Ep 1 (Step 000000): Train loss 9.781, Val loss 9.933
Ep 1 (Step 000005): Train loss 8.111, Val loss 8.339
Every effort moves you,,,,,,,,,,,,,
Ep 2 (Step 000010): Train loss 6.661, Val loss 7.048
Ep 2 (Step 000015): Train loss 5.961, Val loss 6.616
Every effort moves you, and, and, and, and, and, and, and, and, and, and,
and, and, and, and, and, and, and, and, and, and, and, and, and, and,
[...]
```

Промежуточные
результаты удалены
для экономии места

```
Ep 9 (Step 000080): Train loss 0.541, Val loss 6.393
Every effort moves you?" "Yes--quite insensible to the irony. She wanted
him vindicated--and by me!" He laughed again, and threw back the
window-curtains, I had the donkey. "There were days when I
Ep 10 (Step 000085): Train loss 0.391, Val loss 6.452
Every effort moves you know," was one of the axioms he laid down
across the Sevres and silver of an exquisitely appointed
luncheon-table, when, on a later day, I had again run over
from Monte Carlo; and Mrs. Gis
```

Как мы видим, потери при обучении значительно уменьшаются, начиная со значения 9.781 и сходя к 0.391. Языковые навыки модели значительно улучшились. Вначале модель могла только добавлять запятые к начальному контексту (`Every effort moves you,,,,,,,,,,,,,`) или повторять слово `and`. В конце обучения она могла произвести грамматически правильный текст.

Как и в случае с потерями на обучающей выборке, мы видим, что потери на проверочной выборке начинаются с высоких значений (9.933) и постепенно уменьшаются в процессе обучения. Однако они никогда не становятся такими же низкими, как потери на обучающей выборке, и остаются на уровне 6.452 после десятой эпохи.

Прежде чем подробнее рассмотреть потери на проверочной выборке, нарисуем простой график, на котором потери на обучающей и проверочной выборках показаны рядом:

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_losses, label="Training loss")
    ax1.plot(epochs_seen, val_losses, linestyle="-. ", label="Validation loss")
    )
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")
    ax1.legend(loc="upper right")
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
    ax2 = ax1.twinx()
    ax2.plot(tokens_seen, train_losses, alpha=0)
    ax2.set_xlabel("Tokens seen")
    fig.tight_layout()
    plt.show()
```

Добавляет другую ось X, которой отвечает та же самая ось Y

Невидимый график для выравнивания отметок по осям

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

Полученный график потерь при обучении и проверке показан на рис. 5.12. Как мы видим, потери при обучении и проверке начинают снижаться в течение первой эпохи. Это свидетельство того, что модель обучается. Однако после второй эпохи потери начинают расходиться: потери на обучающей выборке продолжают снижаться, а потери на проверочной остаются неизменными. Это расхождение и тот факт, что потери при проверке намного больше, чем потери при обучении, указывают на то, что модель слишком сконцентрирована на запоминании самих данных, а не на обучении. Мы можем подтвердить, что модель дословно запоминает обучающие данные, просматривая сгенерированные текстовые фрагменты, например *quite insensible to the irony* в текстовом файле *The Verdict*.

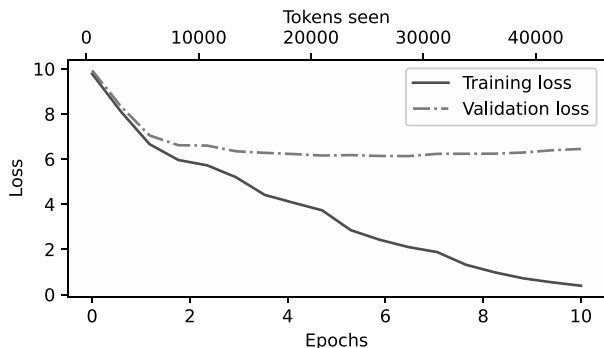


Рис. 5.12. График потерь

Такое запоминание ожидаемо, поскольку мы работаем с очень маленькой обучающей выборкой и обучаем модель в течение нескольких эпох. Обычно модель обучают на гораздо большей выборке в течение всего одной эпохи.

ПРИМЕЧАНИЕ Как упоминалось ранее, заинтересованные читатели могут попробовать обучить модель на 60 000 общедоступных книг из проекта Гутенберг, где переобучения не происходит; подробности см. в приложении Б.

На данный момент мы выполнили четыре задачи из поставленных в этой главе (рис. 5.13). После реализации функции обучения наша модель может генерировать связный текст. Однако она часто дословно запоминает отрывки из обучающей выборки. Далее мы рассмотрим стратегии генерации текста для LLM, чтобы уменьшить запоминание обучающих данных и повысить оригинальность текста, сгенерированного моделью. А позже перейдем к загрузке и сохранению предварительно обученных весов модели GPT от OpenAI.

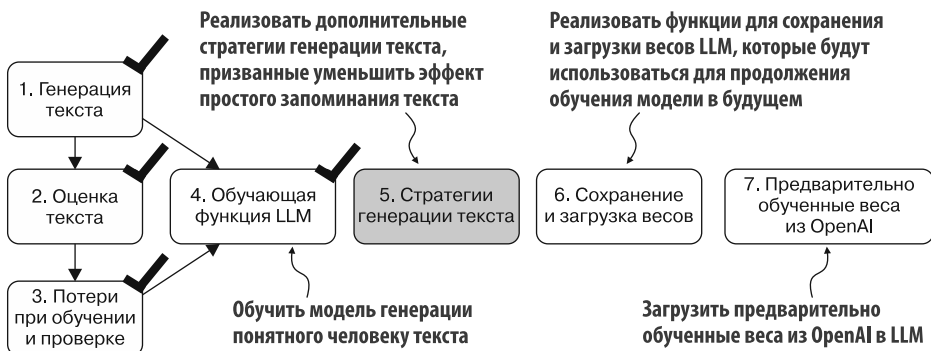


Рис. 5.13. Темы, рассматриваемые в этой главе. Мы завершили шаги 1–4. Далее мы обсудим стратегии для создания более разнообразных выходных текстов (шаг 5)

5.3. Стратегии декодирования для контроля случайности

В этом разделе мы рассмотрим стратегии генерации текста (также называемые стратегиями декодирования), позволяющие создавать более оригинальный текст. Сначала мы вкратце рассмотрим функцию `generate_text_simple`, которую использовали в `generate_and_print_sample` ранее. Затем рассмотрим два метода: *масштабирование температуры* (temperature scaling) и *выборку top-k* (top-k sampling), позволяющих улучшить эту функцию.

Начнем с переноса модели с графического процессора (GPU) на центральный (CPU), поскольку для работы относительно небольшой модели в режиме генерации заключений (inference) не требуется графический процессор. Кроме

того, после обучения мы переводим модель в режим оценки, чтобы отключить случайные компоненты, такие как отсев:

```
model.to("cpu")
model.eval()
```

Затем мы подключаем экземпляр `GPTModel (model)` к функции `generate_text_simple`, которая использует LLM для генерации одного токена за раз:

```
tokenizer = tiktoken.get_encoding("gpt2")
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

Генерируемый текст выглядит так:

```
Output text:
Every effort moves you know," was one of the axioms he laid down across the
Sevres and silver of an exquisitely appointed lun
```

Как объяснялось ранее, на каждом шаге генерации выбирается токен с наибольшей вероятностью среди всех токенов в словаре. Это означает, что LLM всегда будет генерировать одни и те же результаты, даже если мы несколько раз запустим предыдущую функцию `generate_text_simple` с одним и тем же начальным контекстом (`Every effort moves you`).

5.3.1. Масштабирование температуры

Теперь рассмотрим масштабирование температуры — метод, который добавляет процесс вероятностного выбора в задачу генерации следующего токена. Ранее в функции `generate_text_simple` мы всегда выбирали токен, который с наибольшей вероятностью станет следующим, используя `torch.argmax`, также известное как *жадное декодирование* (greedy decoding). Чтобы генерировать более разнообразный текст, мы можем заменить `argmax` функцией, которая выбирает значения из распределения вероятностей (в данном случае это оценки вероятностей, которые LLM генерирует для каждого элемента в словаре на каждом шаге генерации токена).

Чтобы вы могли увидеть вероятностную выборку на конкретном примере, вкратце обсудим процесс генерации следующего токена, используя очень маленький словарный запас в качестве иллюстрации:

```
vocab = {
    "closer": 0,
    "every": 1,
    "effort": 2,
    "forward": 3,
```

```

    "inches": 4,
    "moves": 5,
    "pizza": 6,
    "toward": 7,
    "you": 8,
}
inverse_vocab = {v: k for k, v in vocab.items()}

```

Далее предположим, что LLM получает начальный контекст "every effort moves you" и генерирует такие логиты для следующего токена:

```

next_token_logits = torch.tensor(
    [4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79]
)

```

Как обсуждалось в главе 4, в функции `generate_text_simple` мы преобразуем логиты в вероятности с помощью функции `softmax` и получаем идентификатор токена, соответствующего сгенерированному токenu, используя функцию `argmax`, а затем преобразуем его обратно в текст с помощью обратного словаря:

```

probas = torch.softmax(next_token_logits, dim=0)
next_token_id = torch.argmax(probas).item()
print(inverse_vocab[next_token_id])

```

Наибольшее значение логита и, соответственно, наибольшая оценка вероятности по методу `softmax` находятся в четвертой позиции (индексная позиция 3, так как в Python используется 0-индексация), поэтому сгенерированное слово — "forward".

Чтобы реализовать процесс вероятностной выборки, мы можем заменить `argmax` функцией `multinomial` в PyTorch:

```

torch.manual_seed(123)
next_token_id = torch.multinomial(probas, num_samples=1).item()
print(inverse_vocab[next_token_id])

```

Выведенный результат — "forward", как и раньше. Что произошло? Функция `multinomial` выбирает следующий токен пропорционально его вероятности. Другими словами, "forward" по-прежнему является наиболее вероятным токеном и будет выбран функцией `multinomial` в большинстве случаев, но не всегда. Чтобы вы могли увидеть это, реализуем функцию, которая повторяет выборку 1000 раз:

```

def print_sampled_tokens(probas):
    torch.manual_seed(123)
    sample = [torch.multinomial(probas, num_samples=1).item()
               for i in range(1_000)]
    sampled_ids = torch.bincount(torch.tensor(sample))
    for i, freq in enumerate(sampled_ids):
        print(f"{freq} x {inverse_vocab[i]}")

print_sampled_tokens(probas)

```

На выходе получаем:

```
73 x closer
0 x every
0 x effort
582 x forward
2 x inches
0 x moves
0 x pizza
343 x toward
```

Как мы видим, слово `forward` выбирается чаще всего (582 раза из 1000), но другие токены, такие как `closer`, `inches` и `toward`, тоже будут выбираться в некоторых случаях. Это означает, что если бы мы заменили функцию `argmax` на функцию `multinomial` внутри функции `generate_and_print_sample`, то LLM иногда генерировала бы такие тексты, как `every effort moves you toward`, `every effort moves you inches` и `every effort moves you closer`, а не `every effort moves you forward`.

Можно дополнительно контролировать процесс распределения и выбора, используя концепцию, называемую *температурным масштабированием*. Это просто красивое название для деления логитов на число больше 0:

```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

При температуре выше 1 вероятность появления токена распределяется более равномерно, а при температуре ниже 1 распределение становится более сосредоточенным на узком интервале значений. Чтобы вы могли это увидеть, построим график исходных вероятностей и вероятностей, смасштабированных с учетом различных значений температуры:

```
temperatures = [1, 0.1, 5]
scaled_probas = [softmax_with_temperature(next_token_logits, T)
                  for T in temperatures]
x = torch.arange(len(vocab))
bar_width = 0.15
fig, ax = plt.subplots(figsize=(5, 3))
for i, T in enumerate(temperatures):
    rects = ax.bar(x + i * bar_width, scaled_probas[i],
                  bar_width, label=f'Temperature = {T}')
ax.set_ylabel('Probability')
ax.set_xticks(x)
ax.set_xticklabels(vocab.keys(), rotation=90)
ax.legend()
plt.tight_layout()
plt.show()
```

← Исходные данные, более низкие и более высокие значения доверительной вероятности

Полученный график показан на рис. 5.14. Температура, равная 1, делит логиты на 1 перед передачей их в функцию `softmax` для вычисления значений вероятности. Другими словами, использование температуры, равной 1, эквивалентно отсутствию температурного масштабирования. В этом случае токены выбираются с вероятностью, равной исходным значениям вероятности `softmax`, с помощью функции `multinomial` в PyTorch. Например, при настройке температуры 1 token, соответствующий `forward`, будет выбираться примерно в 60 % случаев.

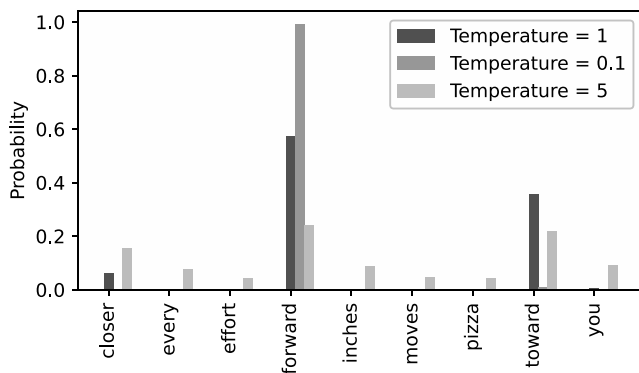


Рис. 5.14. График вероятностей

Кроме того, как мы видим на рис. 5.14, применение очень низких температур, например 0,1, приводит к более «острым» распределениям, поэтому поведение функции `multinomial` выбирает наиболее вероятный token (в данном случае `"forward"`) почти в 100 % случаев, приближаясь к поведению функции `argmax`. Аналогичным образом температура 5 приводит к более равномерному распределению, при котором другие токены выбираются чаще. Это может сделать сгенерированные тексты более разнообразными, но также часто приводит к бессмысленным текстам. Например, использование температуры 5 приводит к тому, что в таких текстах, как `every effort moves you`, слово `pizza` встречается примерно в 4 % случаев.

Упражнение 5.1. Использование функции `print_sampled_tokens`

Используйте функцию `print_sampled_tokens`, чтобы вывести частоты выборки вероятностей `softmax`, масштабированных с помощью температур, показанных на рис. 5.14. Как часто в каждом случае встречается слово `pizza`? Можете ли вы придумать более быстрый и точный способ определения частоты встречаемости этого слова?

5.3.2. Выборка по принципу top-k

Мы реализовали подход с вероятностной выборкой в сочетании с температурным масштабированием, позволяющим повысить разнообразие результатов. Мы заметили, что более высокие значения температуры приводят к более равномерному распределению вероятностей следующего токена. Это позволяет получить более разнообразные результаты, поскольку снижает вероятность того, что модель будет постоянно выбирать наиболее вероятный токен. Данный метод позволяет исследовать менее вероятные, но потенциально более интересные и креативные пути в процессе генерации. Но одним из недостатков метода является то, что иногда он приводит к грамматически неправильным или совершенно бессмысленным результатам, например *every effort moves you pizza*.

Улучшить результаты генерации текста можно, используя вероятностную выборку и температурное масштабирование в сочетании с выборкой top-k. Это позволяет ограничить данную выборку наиболее вероятными токенами и исключить все остальные токены из процесса выбора, замаскировав их оценки вероятности (рис. 5.15).

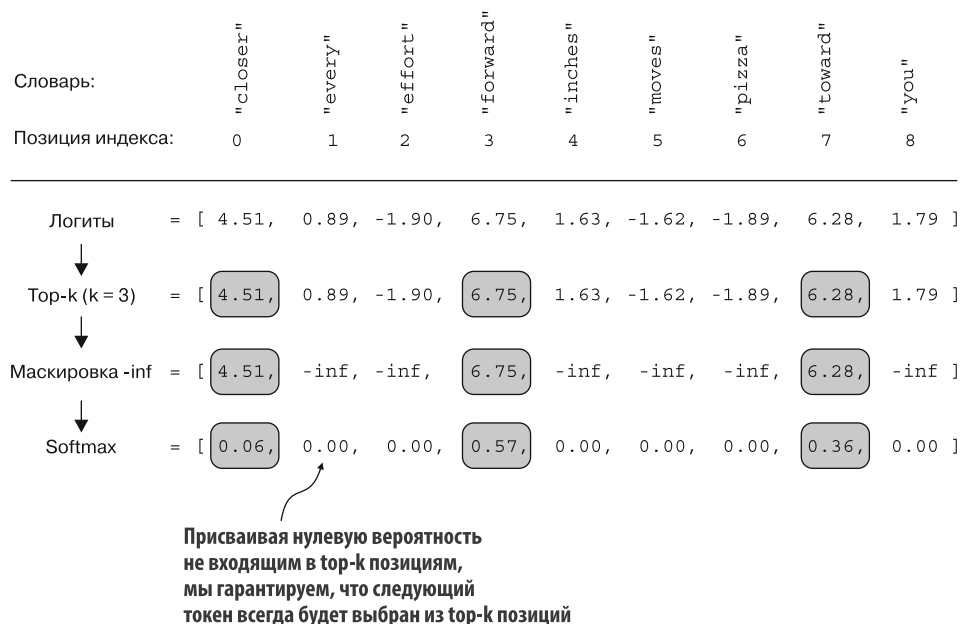


Рис. 5.15. Использование выборки top-k

Используя выборку top-k с $k = 3$, мы фокусируемся на трех токенах, связанных с наивысшими логитами, и маскируем все остальные токены отрицательной бесконечностью ($-\text{inf}$) перед применением функции softmax. Это приводит

к распределению вероятностей, в котором всем токенам, не входящим в top-k, присваивается значение вероятности 0. (Числа на рис. 5.15 округлены до двух знаков после запятой. Значения в строке Softmax должны в сумме составлять 1,0.)

Подход с использованием top-k заменяет все невыбранные логиты значением отрицательной бесконечности ($-\infty$), так что при вычислении значений softmax вероятность токенов, не входящих в top-k, равна 0, а сумма остальных вероятностей равна 1. (Внимательные читатели могут вспомнить этот прием с маскировкой из модуля причинно-следственного внимания, который мы реализовали в разделе 3.5.1.)

Мы можем реализовать процедуру top-k, изображенную на рис. 5.15, следующим образом, начиная с выбора токенов с наибольшими значениями логитов:

```
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)
print("Top logits:", top_logits)
print("Top positions:", top_pos)
```

Значения логитов и идентификаторы токенов трех выбранных токенов в порядке убывания:

```
Top logits: tensor([6.7500, 6.2800, 4.5100])
Top positions: tensor([3, 7, 0])
```

Затем мы применяем функцию `where` в PyTorch, чтобы установить значения логитов токенов, которые находятся ниже минимального значения логита в нашей тройке лидеров, на отрицательную бесконечность ($-\infty$):

```
new_logits = torch.where(
    condition=next_token_logits < top_logits[-1],  ← Идентифицирует логиты,
    input=torch.tensor(float('-inf')),             ← не входящие в топ-3
    other=next_token_logits                        ← Присваивает им -inf
)
print(new_logits)                                ← Ничего не делает с остальными логитами
```

Полученные логиты для следующего токена в словаре из девяти токенов:

```
tensor([4.5100, -inf, -inf, 6.7500, -inf, -inf, -inf, 6.2800, -inf])
```

Наконец, применим функцию `softmax`, чтобы преобразовать их в вероятности следующего токена:

```
topk_probab = torch.softmax(new_logits, dim=0)
print(topk_probab)
```

Как мы видим, результатом такого топ-3 подхода являются три ненулевые оценки вероятности:

```
tensor([0.0615, 0.0000, 0.0000, 0.5775, 0.0000, 0.0000, 0.0000, 0.3610, 0.0000])
```

Теперь мы можем применить температурное масштабирование и функцию `multinomial` для вероятностной выборки, чтобы выбрать следующий токен на основании этих трех ненулевых вероятностных оценок. Для этого мы изменим функцию генерации текста.

5.3.3. Изменение функции генерации текста

Теперь объединим температурную выборку и выборку `top-k`, чтобы изменить функцию `generate_text_simple`, которую мы использовали для генерации текста с помощью LLM ранее, и создать новую функцию генерации (листинг 5.4).

Листинг 5.4. Измененная функция генерации более разнообразного текста

```
def generate(model, idx, max_new_tokens, context_size,
            temperature=0.0, top_k=None, eos_id=None):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]
        if top_k is not None:
            top_logits, _ = torch.topk(logits, top_k)
            min_val = top_logits[:, -1]
            logits = torch.where(
                logits < min_val,
                torch.tensor(float('-inf')).to(logits.device),
                logits
            )
        if temperature > 0.0:
            logits = logits / temperature
            probs = torch.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
        else:
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)
        if idx_next == eos_id:
            break
        idx = torch.cat((idx, idx_next), dim=1)
    return idx
```

Цикл `for` такой же, как и раньше: он получает логиты и фокусируется только на последнем временном шаге

Фильтрует логиты с помощью выборки `top-k`

Применяет температурное масштабирование

Выполняет жадный выбор следующего токена, как и при отключенном температурном масштабировании

Завершает генерацию раньше, если обнаружен токен «конец последовательности»

Посмотрим на эту новую функцию `generate` в действии:

```
torch.manual_seed(123)
token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=25,
    temperature=1.4
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

Сгенерированный текст выглядит так:

Output text:

```
Every effort moves you stand to work on surprise, a one of us had gone
with random-
```

Как мы видим, сгенерированный текст сильно отличается от того, который мы создали ранее с помощью функции `generate_simple` в разделе 5.3 ("Every effort moves you know, " was one of the axioms he laid...!"), и это был заученный отрывок из обучающей выборки.

Упражнение 5.2. Разные значения температуры и параметры top-k

Позэкспериментируйте с различными значениями температуры и параметрами top-k. Основываясь на своих наблюдениях, можете ли вы придумать примеры, в которых желательны более низкие значения температуры и параметров top-k? Аналогичным образом можете ли вы придумать приложения, в которых предпочтительны более высокие значения температуры и top-k? (Рекомендуется также вернуться к этому упражнению в конце главы, после загрузки предварительно обученных весов из OpenAI.)

Упражнение 5.3. Настройки для функции generate

Какие комбинации настроек для функции `generate` позволяют обеспечить детерминированное поведение, то есть отключают случайную выборку так, чтобы она всегда выдавала одни и те же результаты, как функция `generate_simple`?

5.4. Загрузка и сохранение весов модели в PyTorch

До сих пор мы обсуждали, как оценивать прогресс в обучении и предварительно обучать LLM с нуля. Несмотря на то что и LLM, и набор данных были относительно небольшими, это упражнение показало, что предварительное обучение LLM требует больших вычислительных мощностей. Таким образом, важно иметь возможность сохранять модель, чтобы не приходилось заново запускать обучение каждый раз, когда мы хотим использовать ее в новом сеансе.

В этом разделе мы обсудим, как сохранять и загружать предварительно обученную модель (рис. 5.16). Это позволит использовать ее или продолжить обучение позже. В разделе 5.5 мы загрузим более мощную предварительно обученную модель GPT от OpenAI в наш экземпляр `GPTModel`.

К счастью, сохранить модель PyTorch относительно просто. Рекомендуемый способ — сохранить `state_dict` модели (словарь с параметрами для каждого слоя), используя функцию `torch.save`:

```
torch.save(model.state_dict(), "model.pth")
```

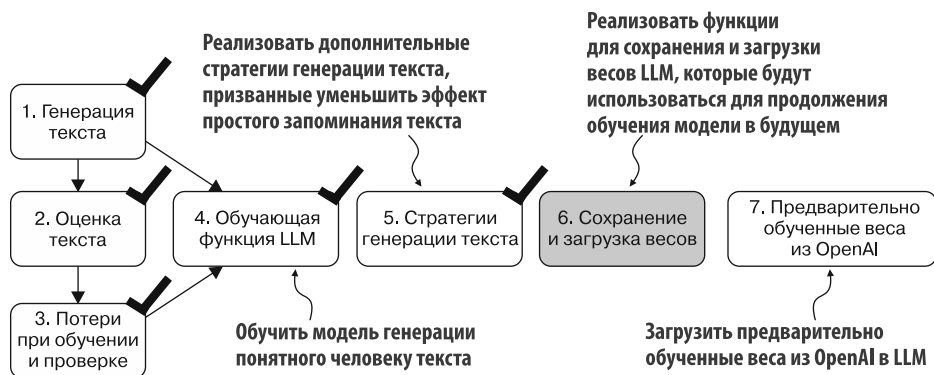


Рис. 5.16. На данный момент мы выполнили шаги 1–5. Далее мы рассмотрим сохранение обученной и проверенной модели (шаг 6)

`model.pth` — это имя файла, в котором сохраняется `state_dict`. Расширение `.pth` используется для файлов PyTorch, хотя технически мы могли бы использовать любое другое расширение.

Затем, сохранив веса модели с помощью `state_dict`, мы можем загрузить веса модели в новый экземпляр `GPTModel`:

```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval()
```

Как обсуждалось в главе 4, отсев помогает предотвратить переобучение модели на обучающих данных за счет случайного «отсеивания» нейронов слоя во время обучения. Однако на этапе предсказания уже обученной сетью мы не хотим случайным образом отсеивать какую-либо информацию, которую усвоила эта сеть. Функция `model.eval()` переключает модель в режим предсказания, отключая слои отсева в модели. Если мы планируем продолжить предварительное обучение модели позже (например, используя функцию `train_model_simple`, которую определили ранее в этой главе), то рекомендуется также сохранять состояние оптимизатора.

Адаптивные оптимизаторы, такие как AdamW, хранят дополнительные параметры для каждого веса модели. AdamW использует исторические данные, чтобы динамически корректировать скорость обучения для каждого параметра модели. Без этого значение оптимизатора устанавливается как нулевое, и модель может обучаться неоптимально или даже не сходиться должным образом — а это означает, что она потеряет способность генерировать связный текст. С помощью `torch.save` мы можем сохранить как модель, так и содержимое `state_dict` оптимизатора:

```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
},
    "model_and_optimizer.pth"
)
```

Мы можем восстановить состояния модели и оптимизатора, загружая сохраненные данные через `torch.load`, а затем используя метод `load_state_dict`:

```
checkpoint = torch.load("model_and_optimizer.pth", map_location=device)
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```

Упражнение 5.4. Еще одна эпоха обучения

После сохранения весов загрузите модель и оптимизатор в новый сеанс Python или файл блокнота Jupyter и продолжите предварительное обучение еще на одну эпоху, используя функцию `train_model_simple`.

5.5. Загрузка предварительно обученных весов из OpenAI

Ранее мы обучали небольшую модель GPT-2, используя ограниченный набор данных, состоящий из рассказа *The Verdict*. Такой подход позволил сосредоточиться на основах, не тратя много времени и вычислительных ресурсов.

К счастью, OpenAI открыто поделилась весами своих моделей GPT-2, что избавило нас от необходимости вкладывать десятки и сотни тысяч долларов в переобучение модели на большом массиве данных. Итак, загрузим эти веса в наш класс `GPTModel` и будем использовать модель для генерации текста. Здесь под весами подразумеваются весовые параметры, хранящиеся в атрибутах `.weight`, например, в слоях `Linear` и `Embedding` в PyTorch. Ранее мы обращались к ним с помощью `model.parameters()` при обучении модели. В главе 6 мы повторно используем эти предварительно обученные веса, чтобы выполнить более тонкую настройку, подходящую под решение задачи классификации текста, и будем следовать инструкциям, аналогичным ChatGPT.

Обратите внимание: OpenAI изначально сохраняла веса GPT-2 с помощью TensorFlow, который нам нужно установить, чтобы загрузить веса в Python. В коде, приведенном ниже, для отслеживания процесса загрузки используется индикатор выполнения `tqdm`, который нам тоже нужно установить.

Вы можете установить эти библиотеки, выполнив в своем терминале следующую команду:

```
pip install tensorflow>=2.15.0 tqdm>=4.66
```

Код скачивания относительно длинный, в основном шаблонный и не очень интересный. Поэтому вместо того чтобы тратить драгоценное место в книге на код Python для скачивания файлов из Интернета, мы скачаем модуль `gpt_download.py` непосредственно из онлайн-репозитория этой главы:

```
import urllib.request
url = (
    "https://raw.githubusercontent.com/rasbt/"
    "LLMs-from-scratch/main/ch05/"
    "01_main-chapter-code/gpt_download.py"
)
filename = url.split('/')[-1]
urllib.request.urlretrieve(url, filename)
```

Далее, когда этот файл будет скачан в локальную папку, вам следует кратко изучить его содержимое, чтобы убедиться, что он был сохранен правильно и содержит корректный код Python.

Теперь мы можем импортировать функцию `download_and_load_gpt2` из файла `gpt_download.py`, что позволит загрузить настройки архитектуры GPT-2 (`settings`) и параметры весов (`params`) в наш сеанс Python:

```
from gpt_download import download_and_load_gpt2
settings, params = download_and_load_gpt2(
    model_size="124M", models_dir="gpt2"
)
```

При выполнении этого кода скачиваются следующие семь файлов, связанных с моделью GPT-2 с 124 млн параметров:

```
checkpoint: 100%|████████████████████████████████████████| 77.0/77.0 [00:00<00:00,
63.9kiB/s]
encoder.json: 100%|██████████████████████████████████████| 1.04M/1.04M [00:00<00:00,
2.20MiB/s]
hparams.json: 100%|██████████████████████████████████████| 90.0/90.0 [00:00<00:00,
78.3kiB/s]
model.ckpt.data-00000-of-00001: 100%|██████████████████| 498M/498M [01:09<00:00,
7.16MiB/s]
model.ckpt.index: 100%|██████████████████████████████████| 5.21k/5.21k [00:00<00:00,
3.24MiB/s]
model.ckpt.meta: 100%|██████████████████████████████████| 471k/471k [00:00<00:00,
2.46MiB/s]
vocab.bpe: 100%|██████████████████████████████████████| 456k/456k [00:00<00:00,
1.70MiB/s]
```


ПРИМЕЧАНИЕ Если код скачивания не работает, это может быть связано с нестабильным подключением к Интернету, проблемами с сервером или изменениями в том, как OpenAI предоставляет доступ к весам модели GPT-2 с открытым исходным кодом. В этом случае, пожалуйста, посетите онлайн-репозиторий кода данной главы по адресу <https://github.com/rasbt/LLMs-from-scratch>, чтобы получить альтернативные и обновленные инструкции, а также обратитесь на форум издательства Manning для получения дополнительной информации.

Предположим, что выполнение предыдущего кода завершено; проверим содержимое `settings` и `params`:

```
print("Settings:", settings)
print("Parameter dictionary keys:", params.keys())
```

Содержимое выглядит так:

```
Settings: {'n_vocab': 50257, 'n_ctx': 1024, 'n_embd': 768, 'n_head': 12,
          'n_layer': 12}
Parameter dictionary keys: dict_keys(['blocks', 'b', 'g', 'wpe', 'wte'])
```

Как `settings`, так и `params` — словари Python. В `settings` хранятся параметры архитектуры LLM аналогично нашим вручную заданным параметрам GPT_CONFIG_124M. Словарь `params` содержит фактические тензоры весов. Обратите внимание, что мы вывели только ключи словаря, так как вывод содержимого тензоров весов занял бы слишком много места на экране. Однако мы можем просмотреть эти тензоры весов, выведя весь словарь с помощью `print(params)` или выбрав отдельные тензоры по соответствующим ключам словаря, например веса слоя вложения:

```
print(params["wte"])
print("Token embedding weight tensor dimensions:", params["wte"].shape)
```

Веса слоя вложения токенов выглядят так:

```
[[-0.11010301 ... -0.1363697  0.01506208  0.04531523]
 [ 0.04034033 ...  0.08605453  0.00253983  0.04318958]
 [-0.12746179 ...  0.08991534 -0.12972379 -0.08785918]
 ...
 [-0.04453601 ...  0.10435229  0.09783269 -0.06952604]
 [ 0.1860082 ... -0.09625227  0.07847701 -0.02245961]
 [ 0.05135201 ...  0.00704835  0.15519823  0.12067825]]
Token embedding weight tensor dimensions: (50257, 768)
```

Мы скачали и сохранили веса самой маленькой модели GPT-2, используя `download_and_load_gpt2(model_size="124M", ...)`. Кроме того, OpenAI предоставляет веса более крупных моделей: 355M, 774M и 1558M. Их общая архитектура одинакова (рис. 5.17), за исключением того, что разные архитектурные элементы

(такие как блоки внимания и трансформеры) повторяются разное количество раз, а размер вложения различается. Код, который приводится далее в главе, также совместим с этими более крупными моделями.

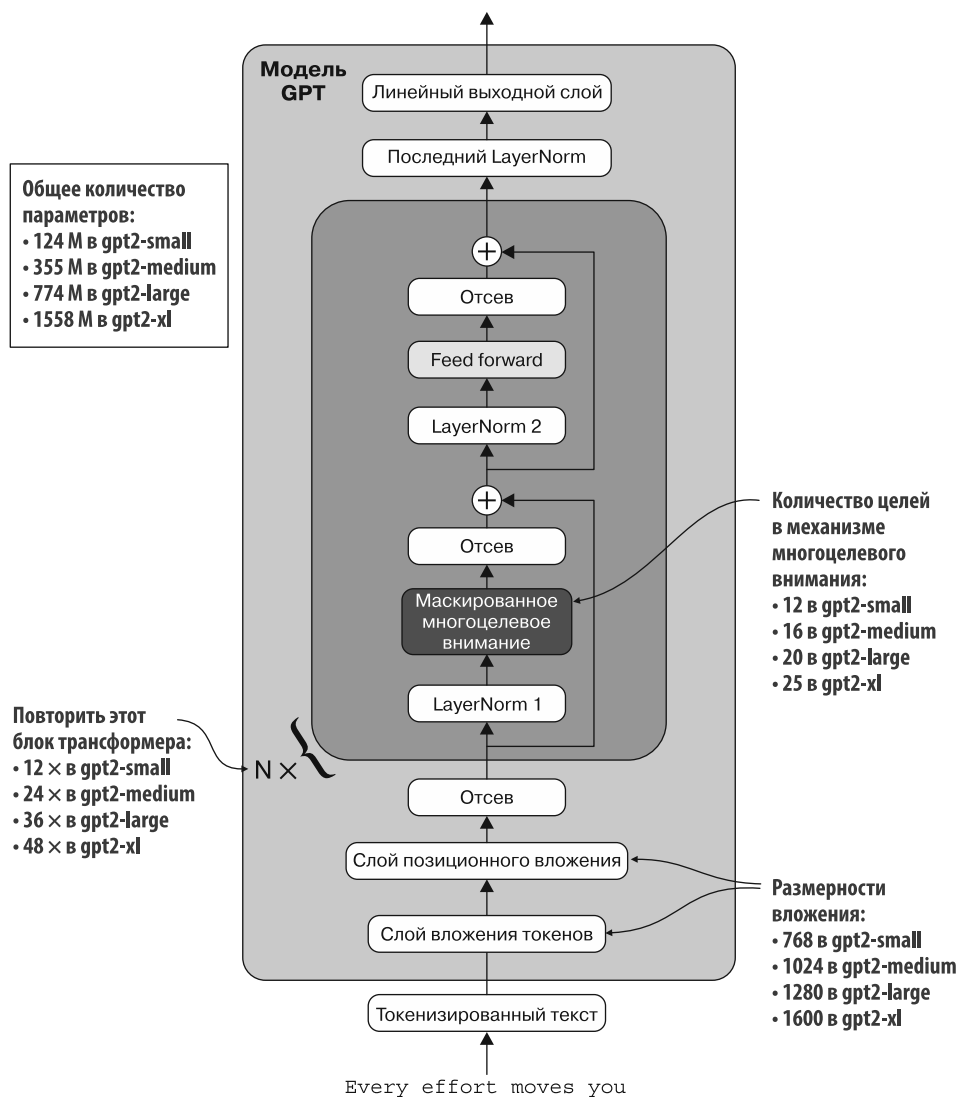


Рис. 5.17. Модели GPT-2-подобных LLM с разным количеством параметров: от 124 до 1558 млн

После загрузки весов модели GPT-2 в Python нужно перенести их из словарей `settings` и `params` в наш экземпляр `GPTModel`. Сначала мы создаем словарь, в котором перечислены различия между моделями GPT разных размеров, показанных на рис. 5.17:

```
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
```

Предположим, нам нужно загрузить самую маленькую модель "gpt2-small (124M)". Мы можем использовать соответствующие настройки из таблицы `model_configs`, чтобы обновить `GPT_CONFIG_124M`, который определили и использовали ранее:

```
model_name = "gpt2-small (124M)"
NEW_CONFIG = GPT_CONFIG_124M.copy()
NEW_CONFIG.update(model_configs[model_name])
```

Внимательные читатели, возможно, помнят, что ранее мы использовали длину 256 токенов, но оригинальные модели GPT-2 от OpenAI были обучены на контексте длиной 1024 токена, поэтому нам нужно соответствующим образом обновить `NEW_CONFIG`:

```
NEW_CONFIG.update({"context_length": 1024})
```

Кроме того, OpenAI использовала векторы смещения в линейных слоях многоцелевого модуля внимания, чтобы реализовать вычисление матриц запросов, ключей и значений. Векторы смещения больше не применяются в LLM, так как не улучшают производительность моделирования и, следовательно, не нужны. Однако мы работаем с предварительно обученными весами, поэтому нам в целях единообразия нужно согласовать настройки и включить эти векторы смещения:

```
NEW_CONFIG.update({"qkv_bias": True})
```

Теперь мы можем использовать обновленный словарь `NEW_CONFIG` для инициализации нового экземпляра `GPTModel`:

```
gpt = GPTModel(NEW_CONFIG)
gpt.eval()
```

По умолчанию экземпляр `GPTModel` инициализируется случайными весами для предварительного обучения. Последний шаг для использования весов модели

OpenAI — заменить эти случайные веса на веса, которые мы загрузили в словарь `params`. Для этого сначала определим небольшую вспомогательную функцию `assign`, которая проверяет, имеют ли два тензора или массива (`left` и `right`) одинаковые размеры или форму, и возвращает правый тензор в качестве обучаемых параметров PyTorch:

```
def assign(left, right):
    if left.shape != right.shape:
        raise ValueError(f"Shape mismatch. Left: {left.shape}, "
                        "Right: {right.shape}")
    )
    return torch.nn.Parameter(torch.tensor(right))
```

Далее мы определяем функцию `load_weights_into_gpt`, которая загружает веса из словаря `params` в экземпляр `gpt` (листинг 5.5).

Листинг 5.5. Загрузка весов OpenAI в нашу модель GPT

```
import numpy as np

def load_weights_into_gpt(gpt, params):
    gpt.pos_emb.weight = assign(gpt.pos_emb.weight, params['wpe'])
    gpt.tok_emb.weight = assign(gpt.tok_emb.weight, params['wte'])

    for b in range(len(params["blocks"])):
        q_w, k_w, v_w = np.split(
            (params["blocks"][b]["attn"]["c_attn"])[ "w"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.weight = assign(
            gpt.trf_blocks[b].att.W_query.weight, q_w.T)
        gpt.trf_blocks[b].att.W_key.weight = assign(
            gpt.trf_blocks[b].att.W_key.weight, k_w.T)
        gpt.trf_blocks[b].att.W_value.weight = assign(
            gpt.trf_blocks[b].att.W_value.weight, v_w.T)

        q_b, k_b, v_b = np.split(
            (params["blocks"][b]["attn"]["c_attn"])[ "b"], 3, axis=-1)
        gpt.trf_blocks[b].att.W_query.bias = assign(
            gpt.trf_blocks[b].att.W_query.bias, q_b)
        gpt.trf_blocks[b].att.W_key.bias = assign(
            gpt.trf_blocks[b].att.W_key.bias, k_b)
        gpt.trf_blocks[b].att.W_value.bias = assign(
            gpt.trf_blocks[b].att.W_value.bias, v_b)

        gpt.trf_blocks[b].att.out_proj.weight = assign(
            gpt.trf_blocks[b].att.out_proj.weight,
            params["blocks"][b]["attn"]["c_proj"])[ "w"].T)
```

Устанавливает веса позиционного вложения и вложения токенов в соответствии со значениями в `params`

Функция `np.split` используется для разделения весов внимания и смещения на три равные части для компонентов запроса, ключа и значения

Выполняет итерации по каждому блоку трансформера в модели

```

gpt.trf_blocks[b].att.out_proj.bias = assign(
    gpt.trf_blocks[b].att.out_proj.bias,
    params["blocks"][b]["attn"]["c_proj"]["b"])

gpt.trf_blocks[b].ff.layers[0].weight = assign(
    gpt.trf_blocks[b].ff.layers[0].weight,
    params["blocks"][b]["mlp"]["c_fc"]["w"].T)
gpt.trf_blocks[b].ff.layers[0].bias = assign(
    gpt.trf_blocks[b].ff.layers[0].bias,
    params["blocks"][b]["mlp"]["c_fc"]["b"])
gpt.trf_blocks[b].ff.layers[2].weight = assign(
    gpt.trf_blocks[b].ff.layers[2].weight,
    params["blocks"][b]["mlp"]["c_proj"]["w"].T)
gpt.trf_blocks[b].ff.layers[2].bias = assign(
    gpt.trf_blocks[b].ff.layers[2].bias,
    params["blocks"][b]["mlp"]["c_proj"]["b"])

gpt.trf_blocks[b].norm1.scale = assign(
    gpt.trf_blocks[b].norm1.scale,
    params["blocks"][b]["ln_1"]["g"])
gpt.trf_blocks[b].norm1.shift = assign(
    gpt.trf_blocks[b].norm1.shift,
    params["blocks"][b]["ln_1"]["b"])
gpt.trf_blocks[b].norm2.scale = assign(
    gpt.trf_blocks[b].norm2.scale,
    params["blocks"][b]["ln_2"]["g"])
gpt.trf_blocks[b].norm2.shift = assign(
    gpt.trf_blocks[b].norm2.shift,
    params["blocks"][b]["ln_2"]["b"])

gpt.final_norm.scale = assign(gpt.final_norm.scale, params["g"])
gpt.final_norm.shift = assign(gpt.final_norm.shift, params["b"])
gpt.out_head.weight = assign(gpt.out_head.weight, params["wte"])

```

В оригинальной модели GPT-2 от OpenAI веса вложений токенов используются повторно в выходном слое, чтобы уменьшить общее количество параметров. Эта концепция известна как привязка весов

В функции `load_weights_into_gpt` мы тщательно сопоставляем веса из реализации OpenAI с нашей реализацией `GPTModel`. Например, OpenAI сохранила тензор весов для выходного проекционного слоя первого блока трансформера как `params["blocks"][0]["attn"]["c_proj"]["w"]`. В нашей реализации этот тензор соответствует `gpt.trf_blocks[b].att.out_proj.weight`, где `gpt` — экземпляр `GPTModel`.

Разработка функции `load_weights_into_gpt` потребовала много догадок, поскольку OpenAI использовала немного другое соглашение об именовании, чем мы. Однако функция `assign` предупредила бы нас, если бы мы попытались сопоставить два тензора с разными размерами. Кроме того, если бы мы допустили ошибку в этой функции, то заметили бы это, так как полученная модель GPT не смогла бы генерировать связный текст.

202 Глава 5. Предварительное обучение на неразмеченных данных

Теперь попробуем использовать `load_weights_into_gpt` на практике и загрузим веса модели OpenAI в наш экземпляр `gpt`:

```
load_weights_into_gpt(gpt, params)
gpt.to(device)
```

Если модель загружена правильно, то мы можем использовать ее для создания нового текста с помощью нашей предыдущей функции генерации:

```
torch.manual_seed(123)
token_ids = generate(
    model=gpt,
    idx=text_to_token_ids("Every effort moves you", tokenizer).to(device),
    max_new_tokens=25,
    context_size=NEW_CONFIG["context_length"],
    top_k=50,
    temperature=1.5
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

Результирующий текст выглядит следующим образом:

Output text:

Every effort moves you toward finding an ideal new way to practice something!
What makes us want to be on top of that?

Мы можем быть уверены, что правильно загрузили веса модели, поскольку она может генерировать связный текст. Малейшая ошибка в этом процессе приведет к сбою модели. В следующих главах мы продолжим работу с этой предварительно обученной моделью и настроим ее для классификации текста и выполнения инструкций.

Упражнение 5.5. Расчет потерь

Рассчитайте потери на обучающей и проверочной выборках модели GPT с предварительно обученными весами от OpenAI на наборе данных *The Verdict*.

Упражнение 5.6. Эксперименты с моделями и сравнение текстов

Поэкспериментируйте с моделями GPT-2 разного размера (например, с самой большой моделью с 1558 млн параметров) и сравните полученный текст с текстом, сгенерированным моделью, обладающей 124 млн параметров.

Итоги главы

- Генерируя текст, большие языковые модели выводят по одному токену за раз.
- По умолчанию следующий токен генерируется путем преобразования выходных данных модели в оценки вероятности и выбора токена из словаря, соответствующего наивысшей оценке вероятности, что называется жадным декодированием.
- Используя вероятностную выборку и температурное масштабирование, можно влиять на разнообразие и связность генерируемого текста.
- Потери на обучающей и проверочной выборках можно использовать для оценки качества текста, генерируемого LLM во время обучения.
- Предварительное обучение модели включает в себя изменение ее весовых коэффициентов, позволяющее минимизировать потери при обучении.
- Цикл обучения LLM как таковой — стандартная процедура в глубоком обучении, в которой используются обычная функция потерь перекрестной энтропии и оптимизатор AdamW.
- Предварительное обучение LLM на большом текстовом массиве требует много времени и ресурсов, поэтому можно загрузить общедоступные весовые коэффициенты, вместо того чтобы самостоятельно выполнять предварительное обучение модели на большом наборе данных.

Тонкая настройка по классификации

В этой главе

- ✓ Знакомство с различными подходами к тонкой настройке LLM.
- ✓ Подготовка набора данных для классификации текста.
- ✓ Изменение предварительно обученной модели в целях тонкой настройки.
- ✓ Тонкая настройка LLM для выявления спам-сообщений.
- ✓ Оценка точности настроенной модели классификатора.
- ✓ Использование LLM, настроенной для классификации новых данных.

На данный момент мы запрограммировали архитектуру LLM, предварительно обучили ее и научились импортировать предварительно обученные веса из внешнего источника, например OpenAI, в нашу модель. Теперь мы пожнем плоды нашего труда, настроив LLM на конкретную задачу, например классификацию текста. Конкретный пример, который мы рассмотрим, — это классификация текстовых сообщений как «спам» или «не спам». На рис. 6.1 показаны два основных способа настройки LLM: тонкая настройка для классификации (шаг 8) и тонкая настройка для выполнения инструкций (шаг 9).

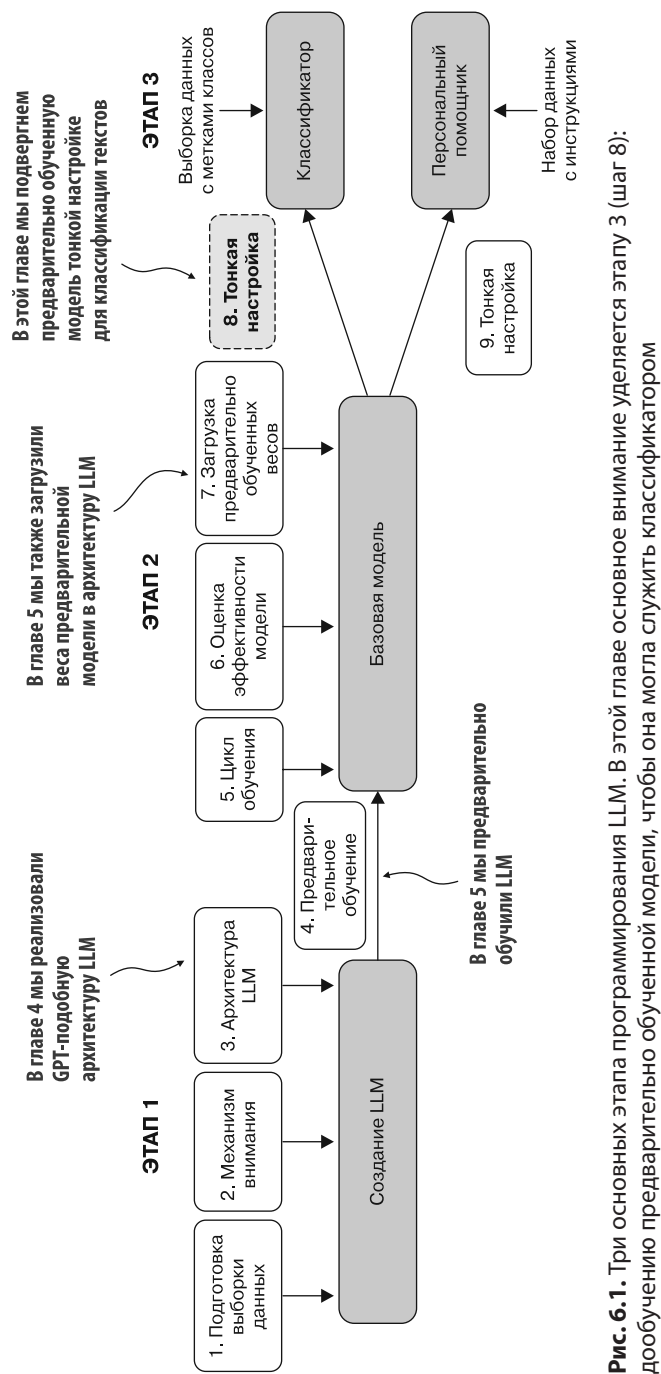


Рис. 6.1. Три основных этапа программирования LLM. В этой главе основное внимание уделяется этапу 3 (шаг 8): дообучению предварительно обученной модели, чтобы она могла служить классификатором

6.1. Различные категории тонкой настройки

Наиболее распространенные способы настройки языковых моделей — *тонкая настройка по инструкциям* (instruction fine-tuning) и *тонкая настройка по классификации* (classification fine-tuning). Тонкая (или точная) настройка по инструкциям предполагает обучение модели на наборе задач с помощью конкретных инструкций, предназначенное для улучшения ее способности понимать и выполнять задачи, описанные в подсказках (или промтах, prompts), введенных на естественном языке. На рис. 6.2 показаны два сценария такой настройки.

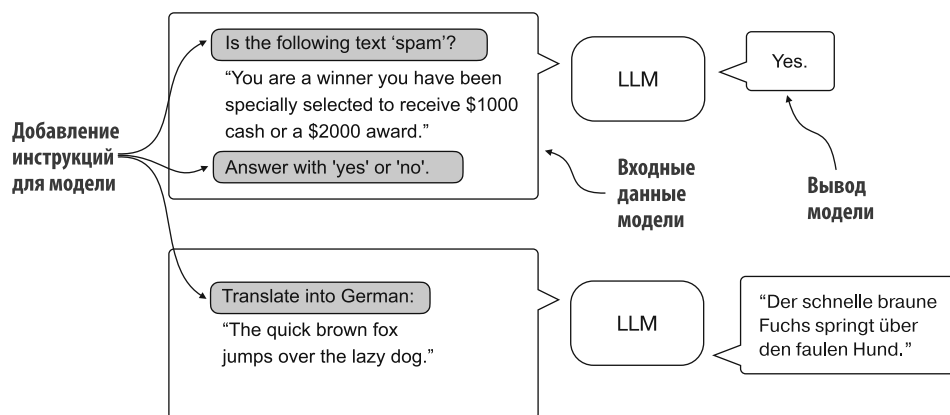


Рис. 6.2. Два разных сценария тонкой настройки по инструкциям: (*вверху*) модели ставится задача определить, является ли заданный текст спамом; (*внизу*) модели дается инструкция по переводу английского предложения на немецкий

При тонкой настройке по классификации, с которой вы, возможно, уже знакомы, если сталкивались с машинным обучением, модель обучается распознавать определенный набор меток классов, например «спам» и «не спам». Примеры задач классификации выходят за рамки больших языковых моделей и фильтрации электронной почты: они включают в себя определение различных видов растений по изображениям, категоризацию новостных статей по таким темам, как спорт, политика и технологии, а также различение доброкачественных и злокачественных опухолей на медицинских изображениях.

Ключевой момент состоит в том, что модель классификации, доработанная с помощью тонкой настройки, может предсказывать только те классы, с которыми сталкивалась во время обучения. Например, она может определить, является ли что-то «спамом» или «не спамом» (рис. 6.3), но не может сказать ничего другого о входном тексте.

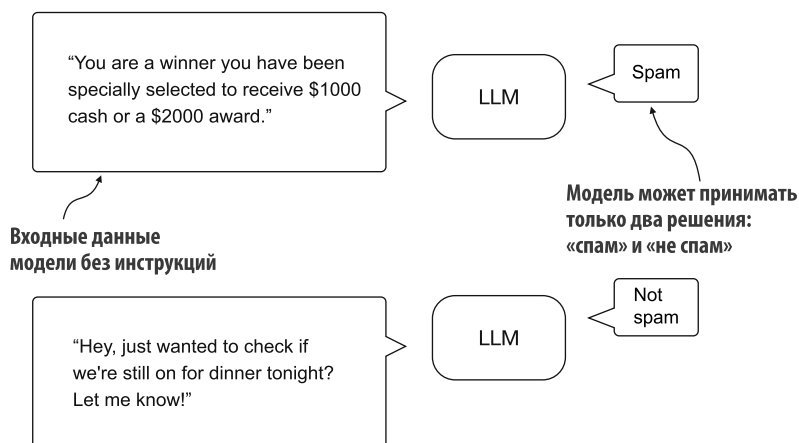


Рис. 6.3. Сценарий классификации текста с использованием LLM. Модель, обученная для классификации спама, не требует дополнительных инструкций наряду с вводом данных. В отличие от модели, обученной на инструкциях, она может отвечать только «спам» или «не спам»

В отличие от модели с тонкой настройкой по классификации, изображенной на рис. 6.3, модель с тонкой настройкой по инструкциям, как правило, может выполнять более широкий спектр задач. Мы можем рассматривать модель с тонкой настройкой по классификации как узкоспециализированную, и, как правило, такую модель разработать проще, чем универсальную.

Выбор правильного подхода

Тонкая настройка по инструкциям улучшает способность модели понимать и генерировать ответы на основе конкретных инструкций пользователя. Такую настройку лучше всего использовать для моделей, которым необходимо выполнять различные задачи на основе сложных пользовательских инструкций, поскольку она позволяет повысить гибкость и качество взаимодействия. Настройка же по классификации идеально подходит для проектов, требующих точной категоризации данных на заранее заданные классы, например для анализа тональности или обнаружения спама.

Тонкая настройка по инструкциям более универсальна, но требует больших наборов данных и вычислительных ресурсов для разработки моделей, способных выполнять различные задачи. В отличие от этого, тонкая настройка по классификации требует меньше данных и вычислительных мощностей, но ее использование ограничено конкретными классами, на которых была обучена модель.

6.2. Подготовка данных

В данной главе мы изменим и дообучим модель GPT, которую ранее внедрили и предварительно обучили. Начнем со скачивания и подготовки набора данных (рис. 6.4). Чтобы вы могли увидеть наглядный и полезный пример дообучения по классификации, мы будем работать с набором текстовых сообщений, состоящим из спама и не спама.

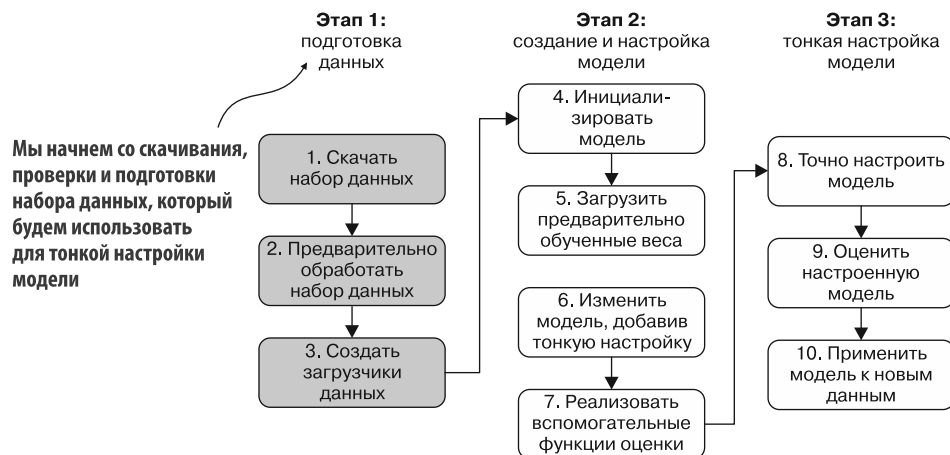


Рис. 6.4. Трехэтапный процесс тонкой настройки LLM по классификации

ПРИМЕЧАНИЕ Текстовые сообщения обычно отправляются по телефону, а не по электронной почте. Однако те же шаги применимы и к классификации сообщений электронной почты, и ссылки на наборы данных, используемые при решении этой задачи, заинтересованные читатели могут найти в приложении Б.

Первый шаг — скачать набор данных (листинг 6.1).

Листинг 6.1. Скачивание и распаковка набора данных

```
import urllib.request
import zipfile
import os
from pathlib import Path

url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"

def download_and_unzip_spam_data(
    url, zip_path, extracted_path, data_file_path):
```

```

if data_file_path.exists():
    print(f"{data_file_path} already exists. Skipping download "
          "and extraction.")
)
return

with urllib.request.urlopen(url) as response:
    with open(zip_path, "wb") as out_file:
        out_file.write(response.read())

with zipfile.ZipFile(zip_path, "r") as zip_ref:
    zip_ref.extractall(extracted_path)

original_file_path = Path(extracted_path) / "SMSSpamCollection"
os.rename(original_file_path, data_file_path)
print(f"File downloaded and saved as {data_file_path}")
download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

```

Скачивает файл

Распаковывает файл

Добавляет расширения файла .tsv

Когда этот код будет выполнен, набор данных сохраняется в виде текстового файла `SMSSpamCollection.tsv` (с табуляцией как разделителем столбцов) в папке `sms_spam_collection`. Мы можем загрузить его в pandas `DataFrame` следующим образом:

```

import pandas as pd
df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
df

```

Отображает фрейм данных в блокноте Jupyter. В качестве альтернативы можно использовать `print(df)`

На рис. 6.5 показан результирующий фрейм набора данных со спамом.

	Label	Text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...
...
5571	ham	Rofl. Its true to its name

5572 rows x 2 columns

Рис. 6.5. Фрагмент набора данных `SMSSpamCollection` в виде pandas `DataFrame`, показывающий метки классов («спам» или «не спам») и соответствующие текстовые сообщения. Набор состоит из 5572 строк (текстовых сообщений и меток)

Посмотрим на распределение меток классов.

```
print(df["Label"].value_counts())
```

Выполнив предыдущий код, мы обнаруживаем, что в данных слово *ham* (то есть не спам) встречается гораздо чаще, чем слово *spam*:

```
Label
ham      4825
spam      747
Name: count, dtype: int64
```

Для простоты и потому, что мы предпочитаем небольшие наборы данных (которые способствуют более быстрой настройке LLM), мы решили уменьшить размер набора данных до 747 экземпляров из каждого класса.

ПРИМЕЧАНИЕ Существует несколько других методов устранения дисбаланса классов, но их описание выходит за рамки этой книги. Найти дополнительную информацию читатели, заинтересованные в изучении методов работы с несбалансированными данными, могут в приложении Б.

Мы можем использовать код из листинга 6.2, чтобы уменьшить размер набора данных и создать сбалансированный набор.

Листинг 6.2. Создание сбалансированного набора данных

```
def create_balanced_dataset(df):
    num_spam = df[df["Label"] == "spam"].shape[0]
    ham_subset = df[df["Label"] == "ham"].sample(
        num_spam, random_state=123
    )
    balanced_df = pd.concat([
        ham_subset, df[df["Label"] == "spam"]
    ])
    return balanced_df
```

Подсчитывает количество примеров с меткой spam

Случайно выбирает примеры с меткой ham, чтобы их количество соответствовало количеству примеров с меткой spam

Объединяет оба подмножества примеров в одно

```
balanced_df = create_balanced_dataset(df)
print(balanced_df["Label"].value_counts())
```

После выполнения предыдущего кода для балансировки набора данных мы видим, что теперь у нас одинаковое количество сообщений «спам» и «не спам»:

```
Label
ham      747
spam      747
Name: count, dtype: int64
```

Далее мы преобразуем строковые метки "ham" и "spam" в целые числа 0 и 1 соответственно:

```
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})
```

Этот процесс похож на преобразование текста в идентификаторы токенов. Однако вместо использования словаря GPT, состоящего из более чем 50 000 слов, мы работаем только с двумя идентификаторами токенов: 0 и 1.

Далее мы создадим функцию `random_split` для разделения набора данных на три части: 70 % для обучения, 10 % для проверки и 20 % для тестирования (листинг 6.3). (Такие соотношения часто используются в машинном обучении для обучения, настройки и оценки моделей.)

Листинг 6.3. Разделение набора данных

```
def random_split(df, train_frac, validation_frac):

    df = df.sample(
        frac=1, random_state=123
    ).reset_index(drop=True)
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(len(df) * validation_frac)

    train_df = df[:train_end]
    validation_df = df[train_end:validation_end]
    test_df = df[validation_end:]

    return train_df, validation_df, test_df

train_df, validation_df, test_df = random_split(
    balanced_df, 0.7, 0.1)
```

Перетасовывает
весь DataFrame

Вычисляет индексы
разбивки

Разбивает DataFrame

Размер тестовой выборки
составляет 20 % от общего
размера данных

Сохраним набор данных в виде файлов CSV, чтобы использовать его позже:

```
train_df.to_csv("train.csv", index=None)
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

На данный момент мы скачали набор данных, сбалансировали его и разделили на обучающие и проверочные подмножества. Далее мы настроим загрузчики данных PyTorch, которые будут использоваться для обучения модели.

6.3. Создание загрузчиков данных

В этом разделе мы разработаем загрузчики данных PyTorch, концептуально аналогичные тем, которые мы использовали при работе с текстовыми данными. Ранее мы применяли метод скользящего окна для создания текстовых фрагментов одинакового размера, которые затем группировали в пакеты, чтобы сделать обучение модели более эффективным. Каждый фрагмент функционировал как отдельный обучающий пример.

Однако теперь мы работаем с набором данных о спаме, который содержит текстовые сообщения разной длины. Объединить их в пакет, как мы сделали с текстовыми фрагментами, можно двумя способами:

- укоротить все сообщения до длины самого короткого в наборе данных или пакете;
- расширить все сообщения до длины самого длинного в наборе данных или пакете.

Первый способ требует меньше вычислительных ресурсов, но может привести к значительной потере информации, если короткие сообщения намного меньше средних или самых длинных, что потенциально снижает производительность модели. Поэтому мы выбираем второй способ, который сохраняет содержание всех сообщений.

Чтобы реализовать пакетную обработку, при которой все сообщения дополняются до длины самого длинного в наборе данных, мы добавляем заполняющие токены ко всем более коротким сообщениям. Для этого используем "`<|endoftext|>`" в качестве заполняющего токена.

Однако вместо того чтобы добавлять строку "`<|endoftext|>`" к каждому текстовому сообщению напрямую, мы можем добавить идентификатор токена, соответствующий "`<|endoftext|>`", к закодированным текстовым сообщениям.

Весь процесс подготовки входного текста показан на рис. 6.6. Сначала каждое текстовое сообщение преобразуется в последовательность идентификаторов токенов. Затем, чтобы обеспечить одинаковую длину последовательностей, более короткие из них дополняются токеном-заполнителем (в данном случае идентификатор токена — `50256`), чтобы соответствовать длине самой длинной последовательности.

Здесь `50256` — идентификатор токена заполнения "`<|endoftext|>`". Мы можем перепроверить правильность идентификатора, закодировав данный токен с помощью *токенизатора GPT-2* из пакета `tiktoken`, который использовали ранее:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>}"))
```

Действительно, при выполнении этого кода возвращается `[50256]`.

Сначала нам нужно реализовать класс `PyTorch Dataset`, который определяет, как загружаются и обрабатываются данные, прежде чем мы сможем создать загрузчики данных (листинг 6.4). Для этого мы определяем класс `SpamDataset`, который реализует концепции, показанные на рис. 6.6. Он выполняет несколько

ключевых задач: определяет самую длинную последовательность в обучающей выборке данных, кодирует текстовые сообщения и обеспечивает добавление *токена-заполнителя* ко всем остальным последовательностям, чтобы их длина соответствовала размеру самой длинной из них.

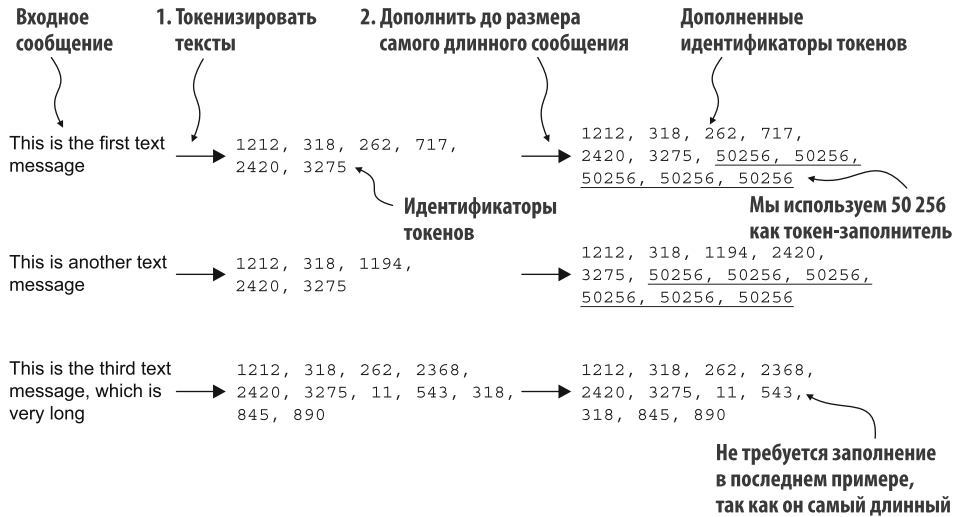


Рис. 6.6. Процесс подготовки текста

Листинг 6.4. Настройка класса PyTorch Dataset

```
import torch
from torch.utils.data import Dataset

class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None,
                  pad_token_id=50256):
        self.data = pd.read_csv(csv_file)

        self.encoded_texts = [
            tokenizer.encode(text) for text in self.data["Text"]
        ]

        if max_length is None:
            self.max_length = self._longest_encoded_length()
        else:
            self.max_length = max_length

            self.encoded_texts = [
                encoded_text[:self.max_length]
```

← Предварительно токенизирует тексты

← Усекает последовательности, если они длиннее, чем max_length

```

        for encoded_text in self.encoded_texts
    ]
    self.encoded_texts = [
        encoded_text + [pad_token_id] *
        (self.max_length - len(encoded_text))
        for encoded_text in self.encoded_texts
    ]

def __getitem__(self, index):
    encoded = self.encoded_texts[index]
    label = self.data.iloc[index]["Label"]
    return (
        torch.tensor(encoded, dtype=torch.long),
        torch.tensor(label, dtype=torch.long)
    )

def __len__(self):
    return len(self.data)

def _longest_encoded_length(self):
    max_length = 0
    for encoded_text in self.encoded_texts:
        encoded_length = len(encoded_text)
        if encoded_length > max_length:
            max_length = encoded_length
    return max_length

```

← Дополняет последовательности
до требуемой максимальной длины

Класс `SpamDataset` загружает данные из CSV-файлов, которые мы создали ранее, токенизирует текст с помощью токенизатора GPT-2 из `tiktoken` и позволяет *дополнять* или *усекать* последовательности до одинаковой длины, определяемой либо самой длинной последовательностью, либо заранее заданной максимальной длиной. Это гарантирует, что каждый входной тензор будет иметь одинаковый размер, а это, в свою очередь, необходимо для создания пакетов в загрузчике обучающих данных, который мы реализуем:

```

train_dataset = SpamDataset(
    csv_file="train.csv",
    max_length=None,
    tokenizer=tokenizer
)

```

Максимальная длина последовательности хранится в атрибуте `max_length` набора данных. Если вам интересно узнать количество токенов в самой длинной последовательности, то вы можете использовать следующий код:

```
print(train_dataset.max_length)
```

Код выводит 120, показывая, что самая длинная последовательность содержит не более 120 токенов, что является обычной длиной для текстовых сообщений. Модель может обрабатывать последовательности длиной до 1024 токенов, учитывая ограничение на длину контекста. Если ваш набор данных содержит более длинные тексты, то вы можете передать `max_length=1024` при создании обучающей выборки в предыдущем коде, чтобы убедиться, что данные не превышают поддерживаемую моделью длину входных данных (контекста).

Далее мы дополняем проверочную и тестовую выборки, чтобы они соответствовали длине самой длинной обучающей последовательности. Важно отметить, что любые экземпляры из этих выборок, превышающие длину самого длинного обучающего примера, усекаются с помощью `encoded_text[:self.max_length]` в коде `SpamDataset`, который мы определили ранее. Это усеечение необязательно; вы можете установить `max_length=None` для проверочной и тестовой выборок при условии, что в них нет последовательностей, превышающих 1024 токена:

```
val_dataset = SpamDataset(
    csv_file="validation.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
test_dataset = SpamDataset(
    csv_file="test.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
```

Упражнение 6.1. Увеличение длины контекста

Расширьте входные данные до максимального количества токенов, поддерживаемых моделью, и посмотрите, как это повлияет на точность предсказания.

Используя наборы данных в качестве входных, мы можем создать загрузчики так же, как при работе с текстовыми данными. Однако в этом случае целевые значения представляют собой метки классов, а не следующие токены в тексте. Например, если мы выберем размер пакета, равный 8, то каждый пакет будет состоять из восьми обучающих примеров длиной 120 токенов и соответствующей метки класса для каждого примера: 0 («не спам») или 1 («спам») (рис. 6.7).

Код в листинге 6.5 создает загрузчики данных, предназначенных для обучения, проверки и тестирования модели, которые загружают текстовые сообщения и метки пакетами размером 8 токенов.

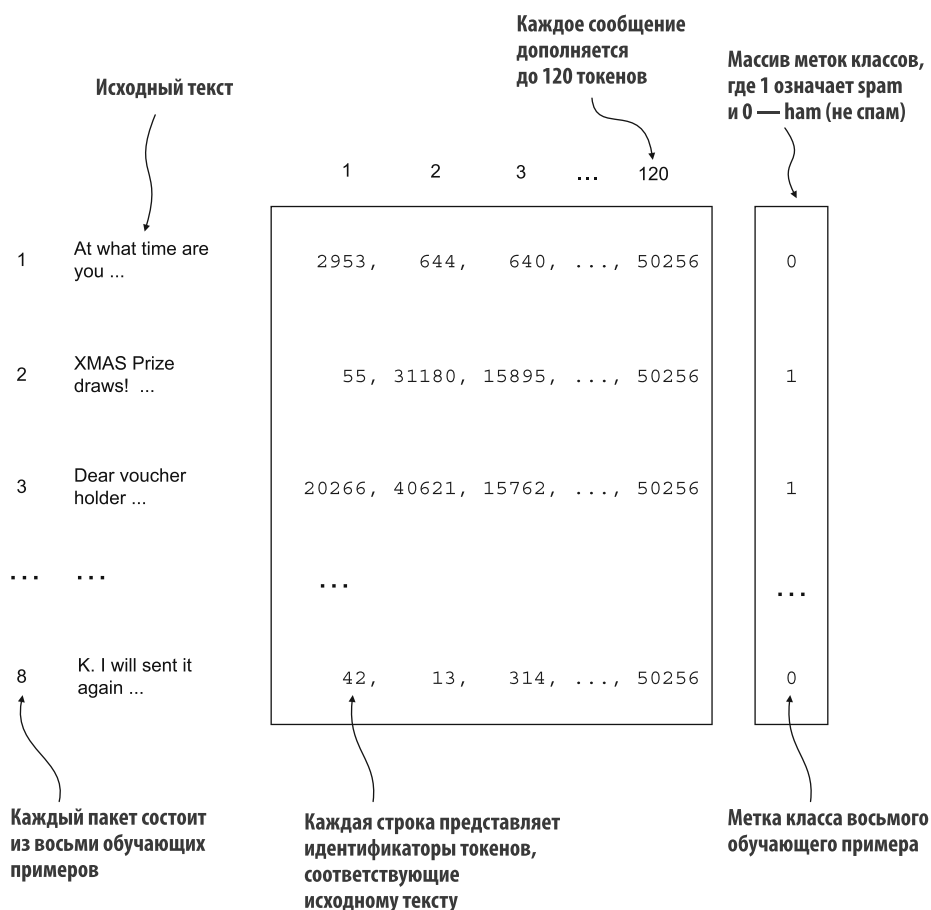


Рис. 6.7. Пакет обучающей выборки, состоящий из восьми текстовых сообщений, представленных в виде идентификаторов токенов

Листинг 6.5. Создание загрузчиков данных PyTorch

```
from torch.utils.data import DataLoader
```

```
num_workers = 0
```

```
batch_size = 8
```

```
torch.manual_seed(123)
```

Эта установка обеспечивает совместимость
с большинством компьютеров

```
train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
```

```

        shuffle=True,
        num_workers=num_workers,
        drop_last=True,
    )
    val_loader = DataLoader(
        dataset=val_dataset,
        batch_size=batch_size,
        num_workers=num_workers,
        drop_last=False,
    )
    test_loader = DataLoader(
        dataset=test_dataset,
        batch_size=batch_size,
        num_workers=num_workers,
        drop_last=False,
    )

```

Чтобы убедиться, что загрузчики данных работают и действительно возвращают пакеты ожидаемого размера, мы проходим цикл по загрузчику обучающих данных и затем выводим размеры тензора последнего пакета:

```

for input_batch, target_batch in train_loader:
    pass
print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)

```

Результат выглядит так:

```

Input batch dimensions: torch.Size([8, 120])
Label batch dimensions torch.Size([8])

```

Как мы видим, входные пакеты состоят из восьми обучающих примеров по 120 токенов в каждом, как и ожидалось. Тензор меток хранит метки классов, соответствующие восьми обучающим примерам.

Наконец, чтобы получить представление о размере набора данных, выведем общее количество пакетов в каждой выборке:

```

print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")

```

Количество пакетов в каждом наборе данных:

```

130 training batches
19 validation batches
38 test batches

```

Теперь, когда мы подготовили данные, нужно подготовить модель к тонкой настройке.

6.4. Инициализация модели с предварительно обученными весами

Мы должны подготовить модель к тонкой настройке по классификации, чтобы распознавать спам-сообщения. Начнем с инициализации нашей предварительно обученной модели (рис. 6.8).

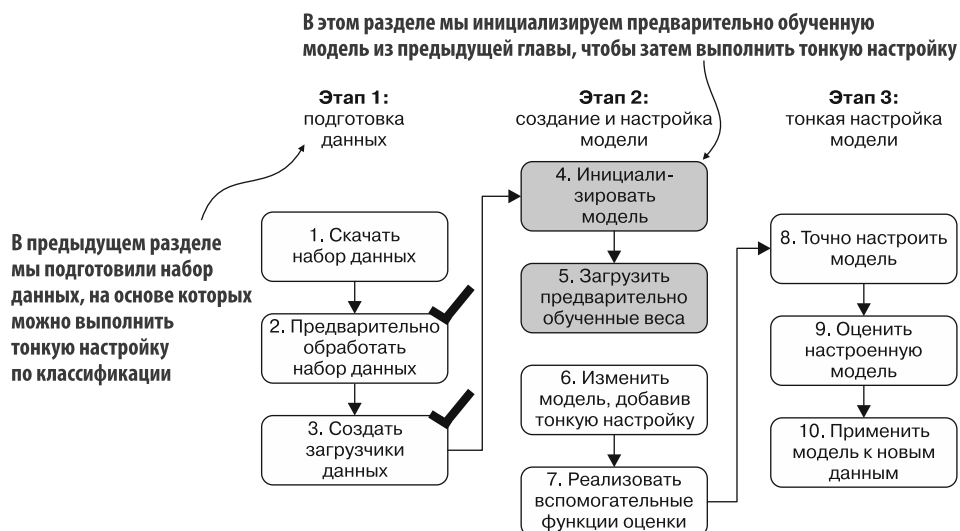


Рис. 6.8. Трехэтапный процесс с тонкой настройкой LLM по классификации. Завершив первый этап, подготовку набора данных, мы теперь должны инициализировать LLM, которую далее будем настраивать для классификации спам-сообщений

Чтобы начать процесс подготовки модели, мы используем те же конфигурации, что и для предварительного обучения на неразмеченных данных:

```

CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"
BASE_CONFIG = {
    "vocab_size": 50257,  ← Размер словаря
    "context_length": 1024,  ← Длина контекста
    "drop_rate": 0.0,  ← Процент отсева
    "qkv_bias": True  ← Смещение запроса-ключа-значения
}
model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])
  
```

Далее мы импортируем функцию `download_and_load_gpt2` из файла `gpt_download.py` и повторно используем класс `GPTModel` и функцию `load_weights_into_gpt`, применяемую для предварительного обучения (см. главу 5), чтобы загрузить скачанные веса в модель GPT (листинг 6.6).

Листинг 6.6. Загрузка предварительно обученной модели GPT

```
from gpt_download import download_and_load_gpt2
from chapter05 import GPTModel, load_weights_into_gpt

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(
    model_size=model_size, models_dir="gpt2"
)

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()
```

После загрузки весов модели в `GPTModel` мы повторно используем функцию генерации текста, которую применяли в главах 4 и 5, чтобы модель генерировала связный текст:

```
from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))
```

Следующий результат показывает, что модель генерирует связный текст и веса модели были правильно загружены:

```
Every effort moves you forward.
The first step is to understand the importance of your work
```

Прежде чем мы начнем настраивать модель для распознавания спама, посмотрим, распознает ли она спам уже сейчас, дав ей инструкции:

```
text_2 = (
    "Is the following text 'spam'? Answer with 'yes' or 'no':"
    " 'You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award.'"
)
token_ids = generate_text_simple(
    model=model,
```

```

idx=text_to_token_ids(text_2, tokenizer),
max_new_tokens=23,
context_size=BASE_CONFIG["context_length"]
)
print(token_ids_to_text(token_ids, tokenizer))

```

Модель выдает следующий результат:

```

Is the following text 'spam'? Answer with 'yes' or 'no': 'You are a winner
you have been specially selected to receive $1000 cash
or a $2000 award.'
The following text 'spam'? Answer with 'yes' or 'no': 'You are a winner

```

Судя по результатам, очевидно, что модель с трудом следует инструкциям. Такой результат ожидаем, поскольку она прошла только предварительное обучение и нуждается в более тонкой настройке по классификации. Далее мы подготовим модель к такой настройке.

6.5. Добавление цели классификации

Мы должны изменить предварительно обученную LLM, чтобы подготовить ее к тонкой настройке по классификации. Для этого мы заменяем исходный выходной слой, который проецирует скрытое представление в словарь из 50 257 слов, на выходной слой меньшего размера, который проецирует в два класса: 0 («не спам») и 1 («спам») (рис. 6.9). Мы используем ту же модель, что и раньше, за исключением того, что заменяем выходной слой.

Узлы выходного слоя

Мы могли бы использовать один выходной узел, поскольку имеем дело с задачей бинарной классификации. Однако для этого потребовалось бы изменить функцию потерь, как я описываю в статье *Losses Learned — Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch* (<https://mng.bz/NRZ2>). Поэтому мы выбираем более общий подход, при котором количество выходов соответствует количеству классов. Например, для задачи с тремя классами, такой как классификация новостных статей по темам «Технологии», «Спорт» или «Политика», мы будем использовать три выходных узла и т. д.

Прежде чем мы попытаемся внести изменения, показанные на рис. 6.9, выведем архитектуру модели с помощью `print(model)`:

```

GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    ...

```



```

(11): TransformerBlock(
  (att): MultiHeadAttention(
    (W_query): Linear(in_features=768, out_features=768, bias=True)
    (W_key): Linear(in_features=768, out_features=768, bias=True)
    (W_value): Linear(in_features=768, out_features=768, bias=True)
    (out_proj): Linear(in_features=768, out_features=768, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
  )
  (ff): FeedForward(
    (layers): Sequential(
      (0): Linear(in_features=768, out_features=3072, bias=True)
      (1): GELU()
      (2): Linear(in_features=3072, out_features=768, bias=True)
    )
  )
  (norm1): LayerNorm()
  (norm2): LayerNorm()
  (drop_resid): Dropout(p=0.0, inplace=False)
)
)
(final_norm): LayerNorm()
(out_head): Linear(in_features=768, out_features=50257, bias=False)
)

```

Этот вывод точно представляет архитектуру, которую мы описали в главе 4. Как мы уже говорили, `GPTModel` состоит из слоев вложения, за которыми следуют 12 идентичных *блоков трансформера* (для краткости показан только последний), за которыми, в свою очередь, следуют последний `LayerNorm` и выходной слой `out_head`.

Далее мы заменяем `out_head` новым выходным слоем (см. рис. 6.9), который будем дообучать.

Дообучение выбранных слоев по сравнению с дообучением всех слоев

Мы начинаем с предварительно обученной модели, поэтому нет необходимости выполнять тонкую настройку всех слоев модели. В языковых моделях на основе нейронных сетей нижние слои обычно отражают базовые языковые структуры и семантику, применимые к широкому спектру задач и наборов данных. Таким образом, для адаптации модели к новым задачам часто достаточно тонкой настройки только последних слоев (то есть слоев, расположенных ближе к выходу), которые более характерны для определенных лингвистических закономерностей и особенностей конкретных задач. Приятным побочным эффектом является то, что тонкая настройка требует меньше вычислительных ресурсов. Больше информации о результатах экспериментов на тему того, какие слои следует настраивать, заинтересованные читатели могут найти в приложении Б.

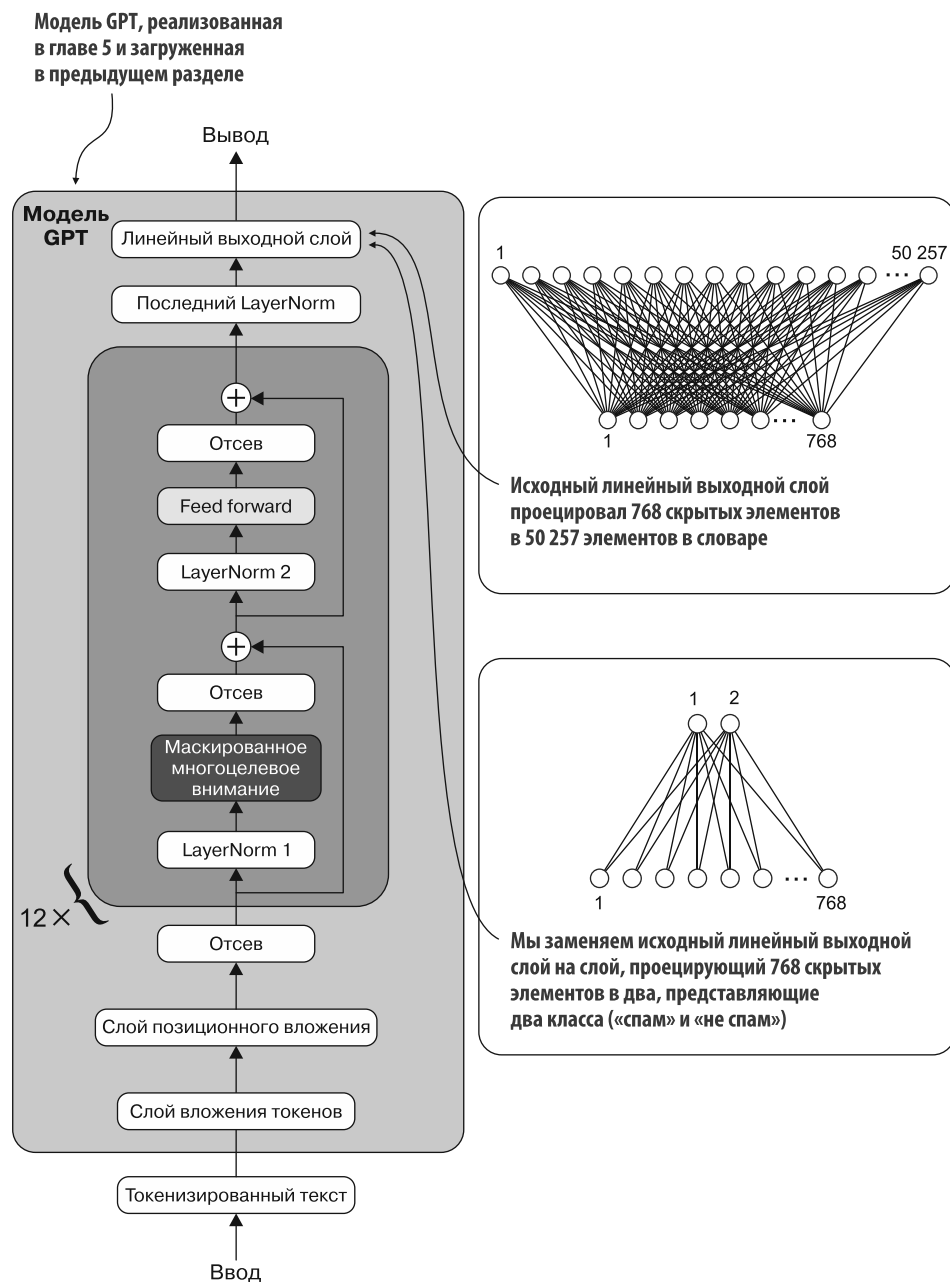


Рис. 6.9. Изменение архитектуры модели GPT: добавление тонкой настройки для классификации спама

Чтобы подготовить модель к тонкой настройке по классификации, мы сначала «замораживаем» ее, то есть делаем все слои необучаемыми:

```
for param in model.parameters():
    param.requires_grad = False
```

Затем мы заменяем выходной слой (`model.out_head`) (листинг 6.7), который изначально преобразует входные данные слоя в 50 257 измерений — размер словаря (см. рис. 6.9).

Листинг 6.7. Добавление слоя классификации

```
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(
    in_features=BASE_CONFIG["emb_dim"],
    out_features=num_classes
)
```

Чтобы сделать код более универсальным, мы используем `BASE_CONFIG["emb_dim"]`, который равен 768 в модели "gpt2-small (124M)". Таким образом, мы можем задействовать тот же код для работы с более крупными вариантами модели GPT-2.

Для выходного слоя `model.out_head` по умолчанию установлено значение `True` для атрибута `requires_grad`. Это значит, что это единственный слой в модели, который будет обновляться во время обучения. Технически обучения выходного слоя, который мы только что добавили, достаточно. Однако, как я выяснил в ходе экспериментов, тонкая настройка дополнительных слоев может заметно улучшить прогнозирующую способность модели. (Подробнее см. в приложении Б.) Мы также настраиваем последний блок трансформера и последний модуль `LayerNorm`, который соединяет этот блок с выходным слоем, чтобы они были обучаемыми (рис. 6.10).

Чтобы сделать последний блок `LayerNorm` и последний блок трансформера обучаемыми, мы устанавливаем для них соответствующее значение `requires_grad` равным `True`:

```
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True
for param in model.final_norm.parameters():
    param.requires_grad = True
```

Упражнение 6.2. Тонкая настройка всей модели

Вместо тонкой настройки только последнего блока трансформера выполните тонкую настройку всей модели и оцените влияние на точность предсказания.

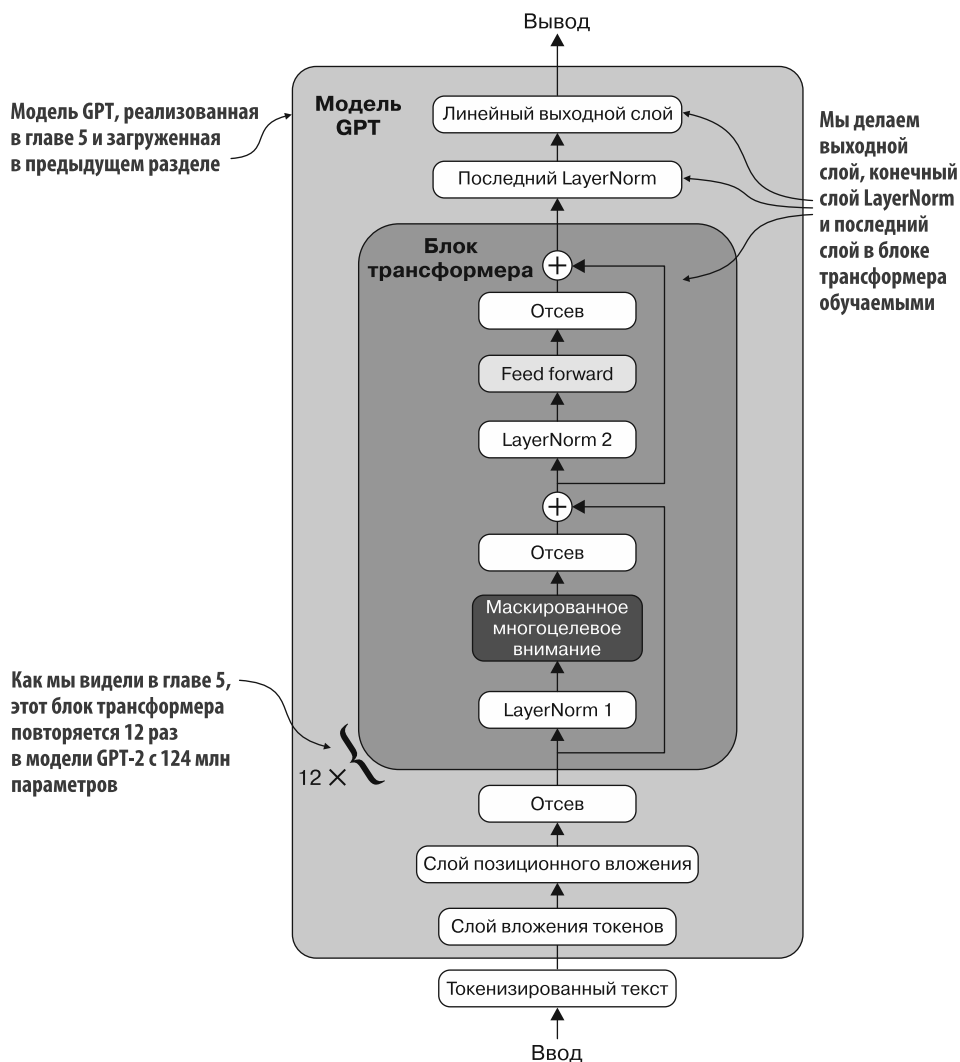


Рис. 6.10. Модель GPT содержит 12 повторяющихся блоков трансформера. Наряду с выходным слоем мы сделали обучаемыми конечный слой LayerNorm и последний блок трансформера. Остальные 11 блоков и вложенные слои не обучаются

Мы добавили новый выходной слой и пометили некоторые слои как обучаемые или необучаемые, но по-прежнему можем использовать эту модель так же, как и раньше. Например, мы можем подать ей пример текста, идентичный ранее использованному:

```
inputs = tokenizer.encode("Do you have time")
inputs = torch.tensor(inputs).unsqueeze(0)
print("Inputs:", inputs)
print("Inputs dimensions:", inputs.shape) ← shape: (batch_size, num_tokens)
```

Вывод показывает, что предыдущий код преобразует входные данные в тензор, состоящий из четырех входных токенов:

```
Inputs: tensor([[5211, 345, 423, 640]])
Inputs dimensions: torch.Size([1, 4])
```

Затем мы можем, как обычно, передать закодированные идентификаторы токенов в модель:

```
with torch.no_grad():
    outputs = model(inputs)
print("Outputs:\n", outputs)
print("Outputs dimensions:", outputs.shape)
```

Итоговый тензор выглядит следующим образом:

```
Outputs:
tensor([[[[-1.5854, 0.9904],
          [-3.7235, 7.4548],
          [-2.2661, 6.6049],
          [-3.5983, 3.9902]]]])
Outputs dimensions: torch.Size([1, 4, 2])
```

Аналогичные входные данные ранее приводили к выходному тензору `[1, 4, 50257]`, где `50257` — размер словаря. Количество выходных строк соответствует количеству входных токенов (в данном случае четыре). Однако размерность каждого выходного вектора (количество столбцов) теперь равна 2, а не 50 257, поскольку мы заменили выходной слой модели.

Помните, что мы заинтересованы в тонкой настройке этой модели, чтобы она возвращала метку класса, указывающую, является ли входной сигнал модели «спамом» или «не спамом». Нам не нужно настраивать все четыре выходные строки; вместо этого мы можем сосредоточиться на одном выходном токене. В частности, мы сосредоточимся на последней строке, соответствующей последнему выходному токenu (рис. 6.11).

Чтобы извлечь последний выходной токен из выходного тензора, мы используем следующий код:

```
print("Last output token:", outputs[:, -1, :])

и получаем следующий результат:

Last output token: tensor([[-3.5983, 3.9902]])
```

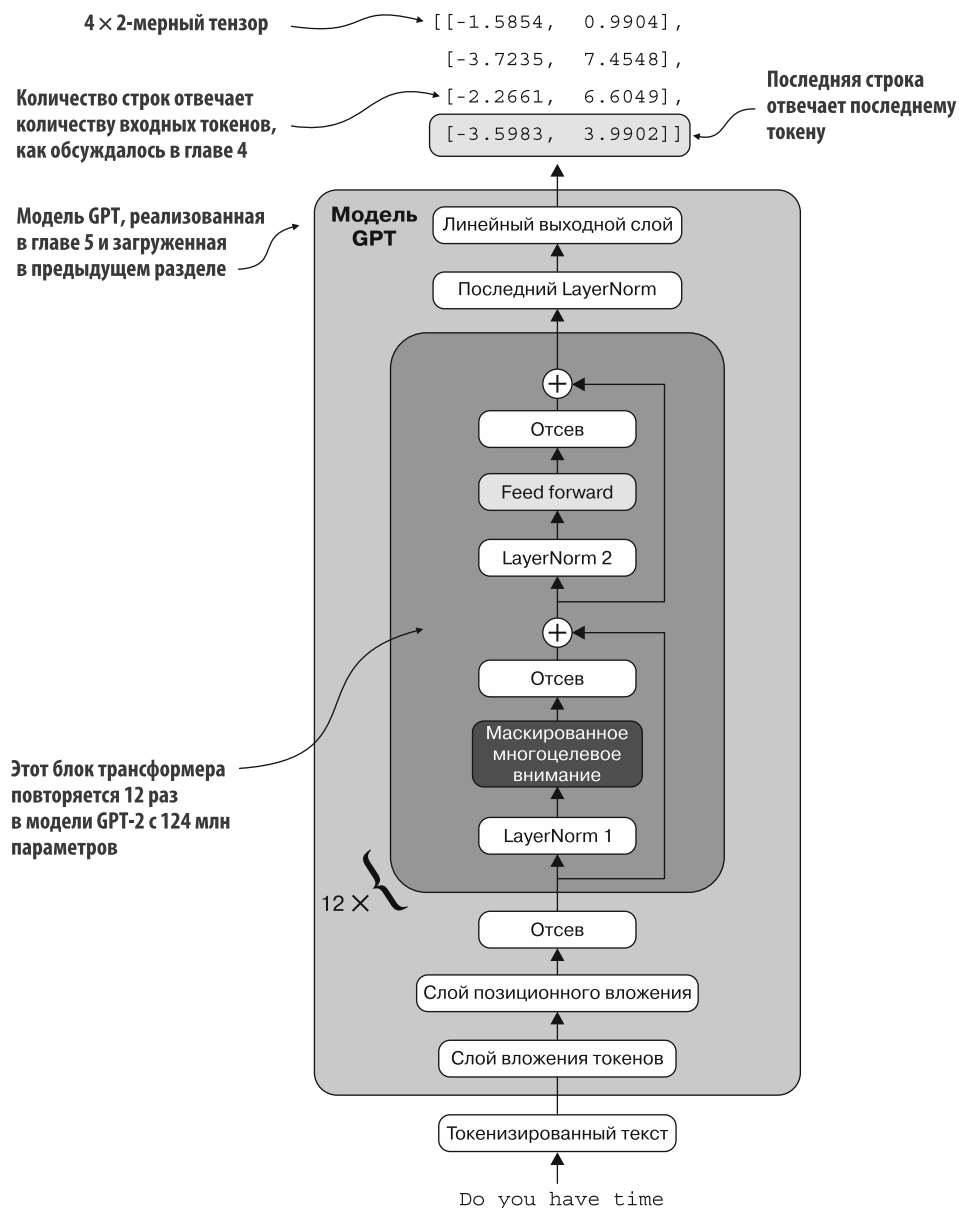
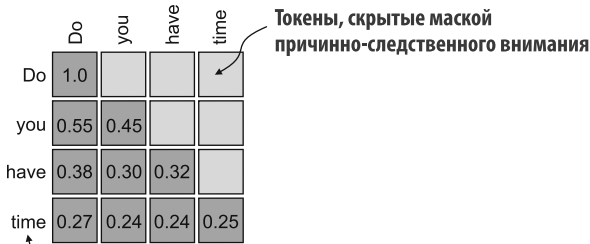


Рис. 6.11. Модель GPT с примером из четырех токенов. Выходной тензор состоит из двух столбцов из-за измененного выходного слоя. При тонкой настройке модели для классификации спама нас интересует только последняя строка, соответствующая последнему токеноу

Нам по-прежнему нужно преобразовать значения в предсказания классов. Но сначала разберемся, почему нас особенно интересует только последний выходной токен.

Мы уже изучили механизм внимания, который устанавливает связь между каждым входным токеном и любым другим входным токеном, а также концепцию *маски причинно-следственного внимания*, которая обычно используется в моделях, подобных GPT (см. главу 3). Эта маска ограничивает фокус внимания токена его текущим положением и предыдущими токенами, гарантируя, что на каждый токен могут влиять только он сам и предыдущие токены (рис. 6.12).



Последний токен — единственный с показателем внимания относительно всех остальных токенов

Рис. 6.12. Механизм причинно-следственного внимания, в котором показатели внимания между входными токенами отображаются в матричном формате

Пустые ячейки в данной матрице указывают на скрытые позиции из-за маски причинно-следственного внимания, которая не позволяет токенам обращать внимание на будущие токены. Значения в ячейках представляют собой показатели внимания; последний токен, *time*, является единственным, который вычисляет показатели внимания для всех предшествующих токенов.

Если учитывать настройку маски причинно-следственной связи, последний токен в последовательности накапливает больше всего информации, поскольку это единственный токен, у которого есть доступ к данным всех предыдущих токенов. Поэтому в нашей задаче по классификации спама мы в процессе тонкой настройки фокусируемся на этом последнем токене.

Теперь мы готовы преобразовать последний токен в предсказания классов и вычислить начальную точность предсказания модели. Далее мы выполним тонкую настройку, благодаря которой модель сможет выполнять задачу классификации спама.

Упражнение 6.3. Тонкая настройка первого и последнего токенов

Попробуйте выполнить тонкую настройку первого выходного токена. Обратите внимание на изменения в производительности предсказания по сравнению с тонкой настройкой последнего выходного токена.

6.6. Расчет потерь и точности классификации

Прежде чем приступить к тонкой настройке модели, нам остается выполнить лишь одну небольшую задачу: реализовать функции оценки модели, используемые во время тонкой настройки (рис. 6.13).

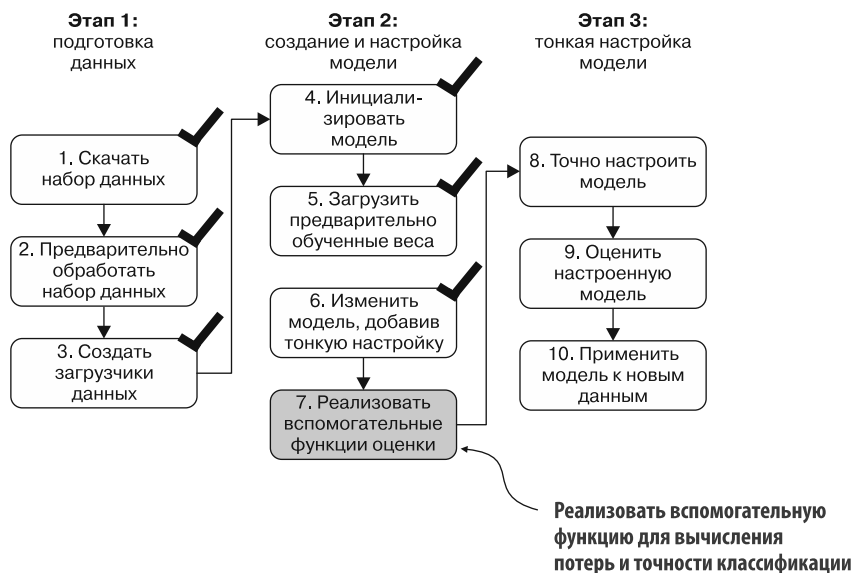


Рис. 6.13. Трехэтапный процесс тонкой настройки LLM для классификации. Мы выполнили первые шесть шагов. Теперь мы готовы к последнему шагу этапа 2: реализации функций, которые позволят оценить эффективность модели при классификации спам-сообщений до, во время и после тонкой настройки

Прежде чем реализовать функции оценки, вкратце обсудим, как мы преобразуем выходные данные модели в предсказания классов. Ранее мы вычисляли идентификатор следующего токена, сгенерированного LLM, преобразуя 50 257 результатов в вероятности с помощью функции `softmax`, а затем возвращая позицию с наибольшей вероятностью путем применения функции `argmax`. Здесь мы используем тот же подход, чтобы вычислить, выдает ли модель прогноз «спам»

или «не спам» для заданного входа (рис. 6.14). Единственная разница в том, что мы работаем с двумерными, а не с 50 257-мерными результатами.

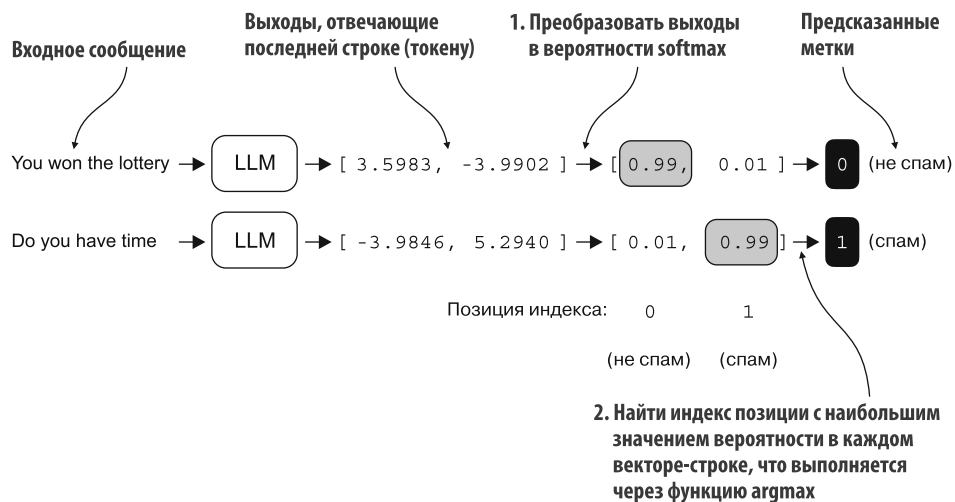


Рис. 6.14. Выходные данные модели, соответствующие последнему токену, преобразуются в оценки вероятности для каждого входного текста. Метки классов получаются путем поиска позиции индекса с наивысшей оценкой вероятности. Модель неправильно предсказывает метки спама, поскольку еще не обучена

Рассмотрим выходные данные последнего токена на конкретном примере:

```
print("Last output token:", outputs[:, -1, :])
```

Значения тензора, соответствующего последнему токену, таковы:

```
Last output token: tensor([[ -3.5983,  3.9902]])
```

Мы можем получить метку класса:

```
probas = torch.softmax(outputs[:, -1, :], dim=-1)
label = torch.argmax(probas)
print("Class label:", label.item())
```

В этом случае код возвращает 1. Это значит, модель предсказывает, что входной текст является «спамом». Использование функции `softmax` здесь необязательно, поскольку наибольшие значения напрямую соответствуют наивысшим оценкам вероятности. Следовательно, мы можем упростить код, не используя `softmax`:

```
logits = outputs[:, -1, :]
label = torch.argmax(logits)
print("Class label:", label.item())
```

С помощью этой концепции можно вычислять точность классификации, которая измеряет процент правильных предсказаний в наборе данных.

Чтобы определить точность классификации, мы применяем код предсказания на основе `argmax` ко всем примерам в наборе данных и вычисляем долю правильных предсказаний, определив функцию `calc_accuracy_loader` (листинг 6.8).

Листинг 6.8. Вычисление точности классификации

```
def calc_accuracy_loader(data_loader, model, device, num_batches=None):
    model.eval()
    correct_predictions, num_examples = 0, 0

    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            input_batch = input_batch.to(device)
            target_batch = target_batch.to(device)

            with torch.no_grad():
                logits = model(input_batch)[: , -1, :]
                predicted_labels = torch.argmax(logits, dim=-1)
                num_examples += predicted_labels.shape[0]
                correct_predictions += (
                    (predicted_labels == target_batch).sum().item()
                )

        else:
            break
    return correct_predictions / num_examples
```

Логиты последнего
выходного токена

Воспользуемся этой функцией, чтобы определить точность классификации для различных наборов данных. Для повышения эффективности мы используем десять пакетов:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
```

```

val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")

```

В настройках устройства модель автоматически запускается на графическом процессоре, если доступен GPU с поддержкой Nvidia CUDA; в противном случае она запускается на центральном. Результат выглядит так:

```

Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%

```

Как мы видим, точность предсказания близка к случайному, которое в данном случае составило бы 50 %. Чтобы повысить точность предсказания, нужно дообучить модель.

Однако прежде чем приступить к дообучению, мы должны определить функцию потерь, которую будем оптимизировать во время обучения. Наша цель — максимизировать точность классификации спама моделью. Это значит, предыдущий код должен выводить правильные метки классов: 0 для не спама и 1 для спама.

Точность классификации не является дифференцируемой функцией, поэтому мы используем функцию потерь перекрестной энтропии в качестве прокси для максимизации точности. Соответственно, функция `calc_loss_batch` остается прежней, но с одной поправкой: мы фокусируемся на оптимизации только последнего токена, `model(input_batch)[: , -1, :]`, а не всех токенов, `model(input_batch)`:

```

def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)
    logits = model(input_batch)[: , -1, :]  ← Логиты последнего
    loss = torch.nn.functional.cross_entropy(logits, target_batch)  выходного токена
    return loss

```

Мы используем функцию `calc_loss_batch`, чтобы вычислить потери для одного пакета, полученного из ранее определенных загрузчиков данных. Потери для всех пакетов в загрузчике данных мы можем вычислить, определив функцию `calc_loss_loader`, как и раньше (листинг 6.9).

Листинг 6.9. Вычисление потерь при классификации

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches
```

Гарантирует, что количество пакетов не превышает их действительное количество

Подобно вычислению точности обучения, мы теперь вычисляем начальные потери для каждого набора данных:

```
with torch.no_grad():
    train_loss = calc_loss_loader(
        train_loader, model, device, num_batches=5
    )
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)
    test_loss = calc_loss_loader(test_loader, model, device, num_batches=5)
print(f"Training loss: {train_loss:.3f}")
print(f"Validation loss: {val_loss:.3f}")
print(f"Test loss: {test_loss:.3f}")
```

Выключает отслеживание градиента, поскольку мы еще не обучаем модель

Начальные потери таковы:

```
Training loss: 2.453
Validation loss: 2.583
Test loss: 2.322
```

Далее мы реализуем функцию обучения для тонкой настройки, что означает корректировку модели при минимизации потерь на обучающей выборке. Такая минимизация поможет повысить точность классификации, что является нашей главной целью.

6.7. Тонкая настройка модели на данных с метками

Итак, мы должны определить и использовать функцию обучения, которая позволит выполнить тонкую настройку предварительно обученной LLM и повысить точность классификации спама. Цикл обучения, показанный на рис. 6.15, представляет собой тот же цикл обучения, который мы использовали для

предварительного обучения; единственное различие заключается в том, что мы вычисляем точность классификации вместо того, чтобы генерировать текст для оценки модели.

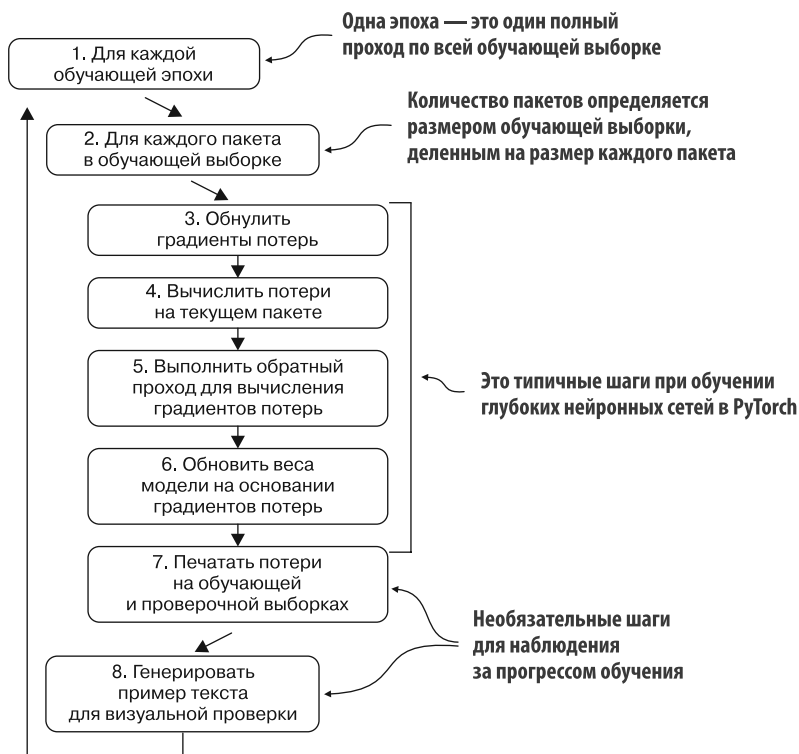


Рис. 6.15. Типичный цикл обучения для глубоких нейронных сетей в PyTorch

Данный цикл состоит из нескольких шагов, на которых мы циклически перебираем пакеты в обучающей выборке в течение нескольких эпох. В каждом цикле мы вычисляем потери для каждого пакета в обучающей выборке, чтобы определить градиенты потерь, которые мы используем для обновления весов модели в целях минимизации потерь на всей обучающей выборке.

Кроме того, функция обучения, реализующая концепции, показанные на рис. 6.15, во многом повторяет функцию `train_model_simple` (листинг 6.10), используемую для предварительного обучения модели. Единственное различие заключается в том, что теперь мы отслеживаем количество просмотренных обучающих примеров (`examples_seen`) вместо количества токенов и вычисляем точность после каждой эпохи, вместо того чтобы выводить пример текста.

Листинг 6.10. Тонкая настройка модели для классификации спама

```
def train_classifier_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs, eval_freq, eval_iter):
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    examples_seen, global_step = 0, -1

    for epoch in range(num_epochs):
        model.train()

        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()
            optimizer.step()
            examples_seen += input_batch.shape[0]
            global_step += 1

            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                    f"Train loss {train_loss:.3f}, "
                    f"Val loss {val_loss:.3f}")

        train_accuracy = calc_accuracy_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_accuracy = calc_accuracy_loader(
            val_loader, model, device, num_batches=eval_iter
        )

        print(f"Training accuracy: {train_accuracy*100:.2f}% | ", end="")
        print(f"Validation accuracy: {val_accuracy*100:.2f}%")
        train_accs.append(train_accuracy)
        val_accs.append(val_accuracy)

    return train_losses, val_losses, train_accs, val_accs, examples_seen
```

Инициализировать списки для отслеживания потерь и рассмотренных примеров

Основной обучающий цикл

Устанавливает модель в режим обучения

Обнуляет градиенты потерь

Вычисляет градиенты потерь

Обновляет веса модели, используя градиенты потерь

Отслеживает примеры вместо токенов

Необязательный шаг оценки

Вычисляет точность после каждой эпохи

Функция `evaluate_model` идентична функции, используемой для предварительного обучения:

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(
```

```

        train_loader, model, device, num_batches=eval_iter
    )
    val_loss = calc_loss_loader(
        val_loader, model, device, num_batches=eval_iter
    )
    model.train()
    return train_loss, val_loss

```

Далее мы инициализируем оптимизатор, задаем количество эпох обучения и запускаем обучение с помощью функции `train_classifier_simple`. Оно занимает около шести минут на ноутбуке MacBook Air с процессором M3 и менее половины минуты на графическом процессоре V100 или A100:

```

import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)
num_epochs = 5

train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(
        model, train_loader, val_loader, optimizer, device,
        num_epochs=num_epochs, eval_freq=50,
        eval_iter=5
    )

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")

```

Результат, который мы видим во время обучения, выглядит так:

```

Ep 1 (Step 000000): Train loss 2.153, Val loss 2.392
Ep 1 (Step 000050): Train loss 0.617, Val loss 0.637
Ep 1 (Step 000100): Train loss 0.523, Val loss 0.557
Training accuracy: 70.00% | Validation accuracy: 72.50%
Ep 2 (Step 000150): Train loss 0.561, Val loss 0.489
Ep 2 (Step 000200): Train loss 0.419, Val loss 0.397
Ep 2 (Step 000250): Train loss 0.409, Val loss 0.353
Training accuracy: 82.50% | Validation accuracy: 85.00%
Ep 3 (Step 000300): Train loss 0.333, Val loss 0.320
Ep 3 (Step 000350): Train loss 0.340, Val loss 0.306
Training accuracy: 90.00% | Validation accuracy: 90.00%
Ep 4 (Step 000400): Train loss 0.136, Val loss 0.200
Ep 4 (Step 000450): Train loss 0.153, Val loss 0.132
Ep 4 (Step 000500): Train loss 0.222, Val loss 0.137
Training accuracy: 100.00% | Validation accuracy: 97.50%
Ep 5 (Step 000550): Train loss 0.207, Val loss 0.143
Ep 5 (Step 000600): Train loss 0.083, Val loss 0.074
Training accuracy: 100.00% | Validation accuracy: 97.50%
Training completed in 5.65 minutes.

```

Затем мы используем Matplotlib, чтобы создать график функции потерь для обучающей и проверочной выборок (листинг 6.11).

Листинг 6.11. Создание графика потерь при классификации

```
import matplotlib.pyplot as plt

def plot_values(
    epochs_seen, examples_seen, train_values, val_values,
    label="loss"):
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_values, label=f"Training {label}")
    ax1.plot(
        epochs_seen, val_values, linestyle="-. ",
        label=f"Validation {label}")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel(label.capitalize())
    ax1.legend()

    ax2 = ax1.twinx()
    ax2.plot(examples_seen, train_values, alpha=0)
    ax2.set_xlabel("Examples seen")

    fig.tight_layout()
    plt.savefig(f"{label}-plot.pdf")
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(epochs_tensor, examples_seen_tensor, train_losses, val_losses)
```

Отображает обучающие и проверочные
потери на каждой эпохе

Создает вторую горизонтальную ось
для обработанных примеров

Невидимое
отображение
для выравнивания
меток графика

Настраивает общий вид

На рис. 6.16 показаны результирующие кривые потерь.

Потери при обучении (обозначены сплошной линией) и потери при проверке (обозначены пунктиром) резко снижаются в первую эпоху и постепенно стабилизируются к пятой. Такая динамика указывает на хороший прогресс в обучении и предполагает, что модель научилась на обучающих данных и хорошо обобщает их на новых данных из проверочной выборки.

Как видно из резкого снижения графика, показанного на рис. 6.16, модель хорошо обучается на обучающих данных, и признаков переобучения практически нет, то есть нет заметной разницы между потерями на обучающей и проверочной выборках.

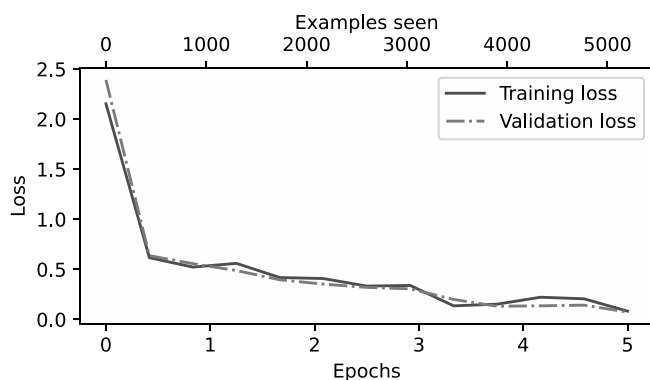


Рис. 6.16. Потери при обучении и проверке модели в течение пяти эпох обучения

Используя ту же функцию `plot_values`, теперь создадим график точности классификации:

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_accs))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_accs))

plot_values(
    epochs_tensor, examples_seen_tensor, train_accs, val_accs,
    label="точность"
)
```

Выбор количества эпох

Ранее, начиная обучение, мы установили количество эпох равным пяти. Количество зависит от набора данных и сложности задачи, и универсального решения или рекомендации не существует, хотя пять эпох обычно являются хорошей отправной точкой. Если модель слишком переобучается после первых нескольких эпох, как показано на графике потерь (см. рис. 6.16), то может понадобиться уменьшить количество эпох. И наоборот, если линия тренда показывает, что потери при проверке могут уменьшиться при дальнейшем обучении, количество эпох следует увеличить. В данном конкретном случае пять эпох — разумное количество, поскольку нет признаков раннего переобучения, а потери при проверке близки к 0.

На рис. 6.17 показана результирующая точность. Модель достигает относительно высокой точности при обучении и проверке после четвертой и пятой эпох. Важно отметить, что при использовании функции `train_classifier_simple` мы ранее установили `eval_iter=5`. Это значит, что наши оценки эффективности обучения и проверки основаны только на пяти пакетах данных.

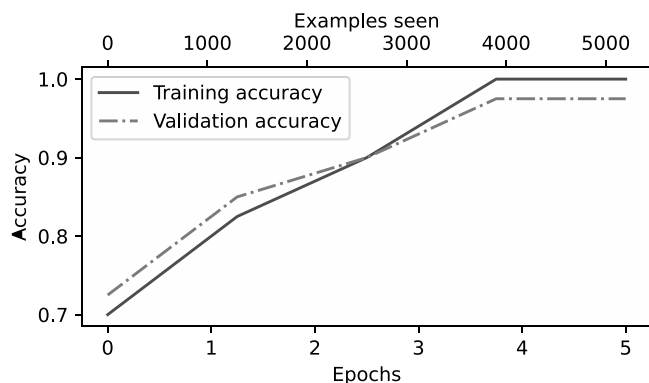


Рис. 6.17. Точность обучения (сплошная линия) и точность проверки (пунктирная линия) существенно возрастают в начале эпох, а затем стабилизируются, достигая почти идеальной точности 1,0. Близкое расположение двух линий на протяжении всех эпох говорит о том, что модель не слишком сильно переобучается

Теперь мы должны рассчитать показатели производительности для обучающей, проверочной и тестовой выборок, выполнив следующий код, на этот раз не определяя значение `eval_iter`:

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

Полученные значения точности таковы:

```
Training accuracy: 97.21%
Validation accuracy: 97.32%
Test accuracy: 95.67%
```

Показатели для обучающей и проверочной выборок практически идентичны. Небольшое расхождение между точностью для обеих выборок указывает на минимальное переобучение модели. Как правило, точность на проверочной выборке несколько выше, чем точность на тестовой, поскольку при разработке модели часто настраиваются гиперпараметры, позволяющие достичь высоких показателей на проверочной выборке, которые могут не так эффективно переноситься на тестовую. Такая ситуация встречается часто, но разрыв можно свести к минимуму, изменив настройки модели, например увеличив коэффициент отсева (`drop_rate`) или параметр `weight_decay` в конфигурации оптимизатора.

6.8. Использование LLM в качестве классификатора спама

После тонкой настройки и оценки модели мы готовы классифицировать сообщения со спамом (рис. 6.18).

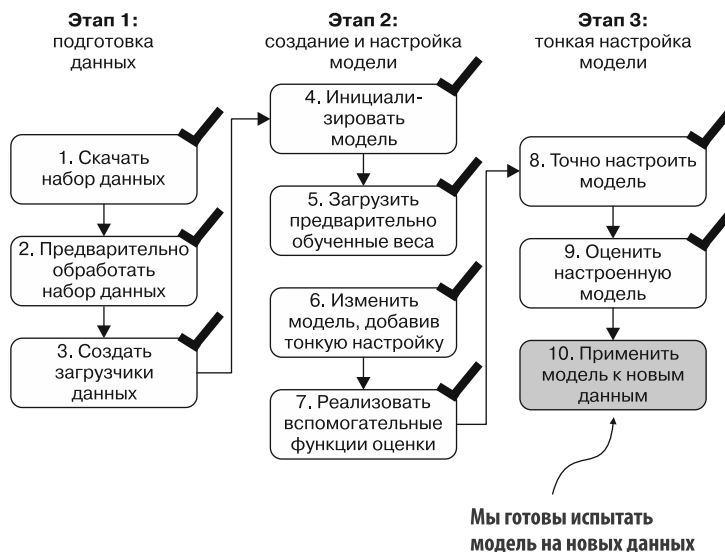


Рис. 6.18. Трехэтапный процесс классификации с тонкой настройкой LLM. Шаг 10 — заключительный шаг этапа 3: использование точно настроенной модели для классификации новых спам-сообщений

Воспользуемся нашей моделью классификации спама на основе GPT. Функция `classify_review` (листинг 6.12) выполняет этапы предварительной обработки данных, аналогичные тем, которые мы использовали в `SpamDataset`, реализованном ранее. Затем, после преобразования текста в идентификаторы токенов, функция использует модель для предсказания метки класса, аналогично тому, что мы реализовали в разделе 6.6, а затем возвращает соответствующее название класса.

Листинг 6.12. Использование модели для классификации новых текстов

```
def classify_review(
    text, model, tokenizer, device,
    max_length=None, pad_token_id=50256):
    model.eval()

    input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[1]
```

Подготавливает входные данные

```

input_ids = input_ids[:min(
    max_length, supported_context_length
)]
input_ids += [pad_token_id] * (max_length - len(input_ids))
input_tensor = torch.tensor(
    input_ids, device=device
).unsqueeze(0)
with torch.no_grad():
    logits = model(input_tensor)[: , -1, :]
    predicted_label = torch.argmax(logits, dim=-1).item()
return "spam" if predicted_label == 1 else "not spam"

```

Усекает слишком длинные последовательности

Добавляет размер пакета

Дополняет последовательности до требуемой длины

Моделирует предсказание без отслеживания градиента

Логиты последнего выходного токена

Возвращает результат классификации

Попробуем применить эту функцию `classify_review` к следующему примеру:

```

text_1 = (
    "You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award."
)

print(classify_review(
    text_1, model, tokenizer, device, max_length=train_dataset.max_length
))

```

Полученная модель правильно предсказывает «спам». Попробуем другой пример:

```

text_2 = (
    "Hey, just wanted to check if we're still on"
    " for dinner tonight? Let me know!"
)

print(classify_review(
    text_2, model, tokenizer, device, max_length=train_dataset.max_length
))

```

Модель снова делает правильный прогноз и возвращает метку «не спам».

Наконец, сохраним модель на случай, если захотим использовать ее позже, не обучая ее повторно. Мы можем использовать метод `torch.save`:

```
torch.save(model.state_dict(), "review_classifier.pth")
```

После сохранения модель может быть загружена:

```

model_state_dict = torch.load("review_classifier.pth", map_location=device")
model.load_state_dict(model_state_dict)

```

Итоги главы

- Существуют различные стратегии тонкой настройки LLM, в том числе настройка по классификации и настройка по инструкциям.
- Настройка по классификации предполагает замену выходного слоя модели небольшим слоем классификации.
- В случае классификации текстовых сообщений как «спам» или «не спам» новый слой классификации состоит всего из двух выходов. Ранее мы использовали количество выходов, равное количеству уникальных токенов в словаре (то есть 50 256).
- Вместо того чтобы предсказывать следующий токен в тексте, как при предварительном обучении, при тонкой настройке по классификации модель обучается выдавать правильную метку класса, например «спам» или «не спам».
- В качестве входных данных для тонкой настройки используется текст, преобразованный в идентификаторы токенов, как при предварительном обучении.
- Перед тонкой настройкой LLM мы загружаем предварительно обученную модель в качестве базовой.
- В оценку модели классификации входит расчет точности классификации (доли или процента правильных прогнозов).
- При тонкой настройке модели классификации используется та же функция потерь перекрестной энтропии, что и при предварительном обучении LLM.

7

Тонкая настройка по инструкциям

В этой главе

- ✓ Тонкая настройка по инструкциям для LLM.
- ✓ Подготовка набора данных для тонкой настройки по инструкциям.
- ✓ Организация данных инструкций в обучающих пакетах.
- ✓ Загрузка предварительно обученной LLM и ее тонкая настройка с помощью инструкций человека.
- ✓ Извлечение ответов по сгенерированным LLM инструкциям для оценки модели.
- ✓ Оценка LLM, настроенной по инструкциям.

Ранее мы реализовали архитектуру LLM, провели предварительное обучение и импортировали предварительно обученные веса из внешних источников в нашу модель. Затем мы сосредоточились на тонкой настройке LLM для конкретной задачи классификации: различение текстовых сообщений со спамом и без него. Теперь мы реализуем процесс тонкой настройки модели по инструкциям человека (рис. 7.1). Это один из основных методов разработки LLM для чат-ботов, персональных помощников и других диалоговых задач. Мы будем выполнять тонкую настройку модели, используя *набор инструкций*.

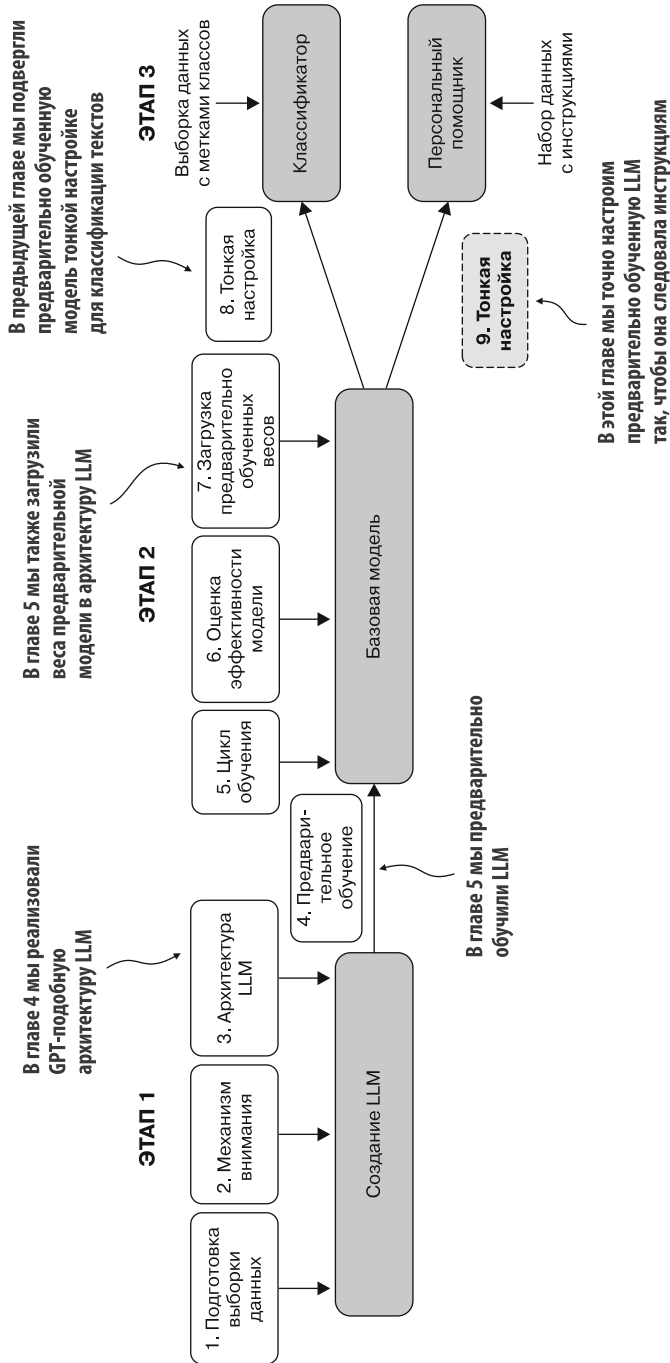


Рис. 7.1. Три основных этапа программирования LLM. В этой главе основное внимание уделяется шагу 9 этапа 3: тонкой настройке предварительно обученной LLM по инструкциям человека

7.1. Введение в тонкую настройку по инструкциям

Вы уже знаете, что в предварительное обучение LLM входит процедура, в ходе которой модель учится генерировать по одному слову за раз. В результате такая модель способна *дополнять текст*, то есть может заканчивать предложения или писать абзацы, если в качестве входных данных ей предоставить фрагмент. Однако предварительно обученные LLM часто испытывают трудности с выполнением конкретных инструкций, таких как «Исправь грамматику в этом тексте» или «Преобразуй этот текст, изменив залог на страдательный». Позже мы рассмотрим конкретный пример, в котором загрузим предварительно обученную LLM в качестве основы для тонкой настройки по инструкциям, также известной как *контролируемая тонкая настройка по инструкциям*.

Сначала мы сосредоточимся на улучшении способности LLM следовать таким инструкциям и генерировать желаемый ответ (рис. 7.2). Подготовка набора данных — ключевой аспект тонкой настройки по инструкциям. Затем мы выполним все шаги трех этапов процесса такой настройки (рис. 7.3).

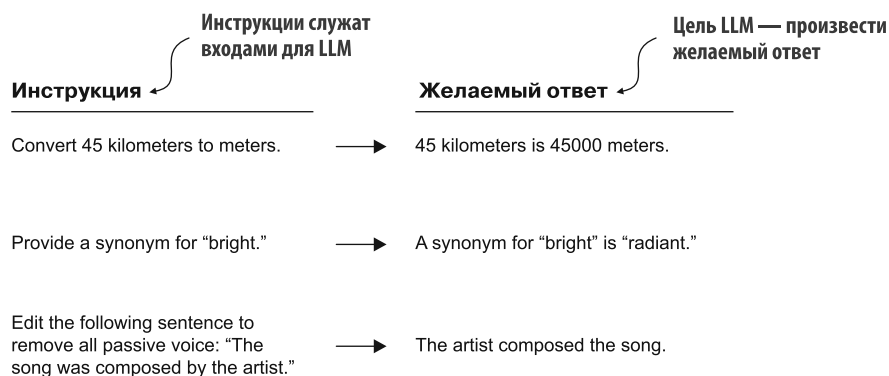
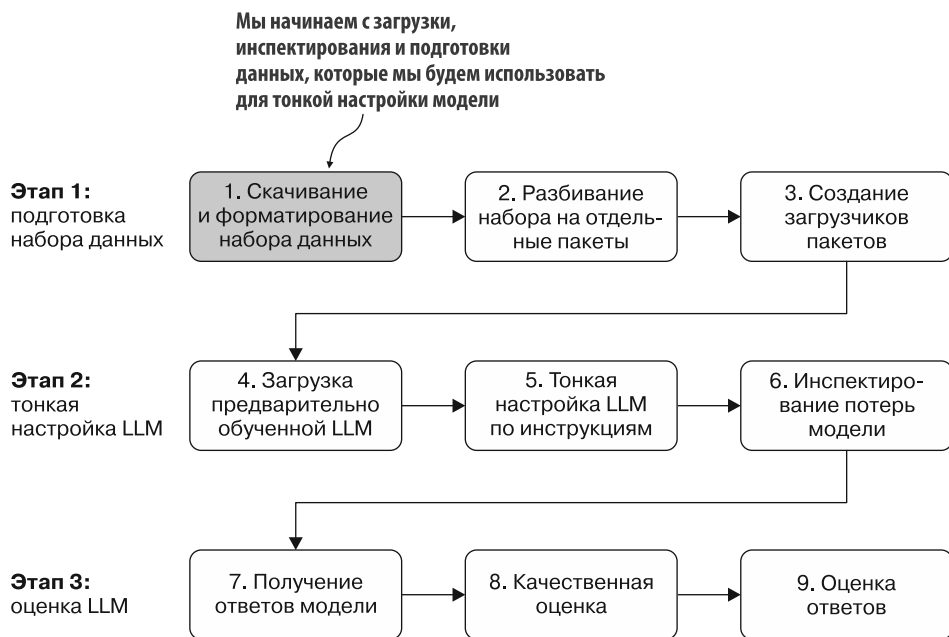


Рис. 7.2. Примеры инструкций, которые обрабатываются LLM, чтобы она могла генерировать желаемые ответы

Начнем с первого этапа: загрузки и форматирования набора данных.

7.2. Подготовка набора данных для контролируемой тонкой настройки по инструкциям

Набор данных, который мы будем загружать и форматировать, состоит из 1100 пар «инструкция — ответ», аналогичных тем, которые показаны на рис. 7.2. Этот набор был создан специально для данной книги, но заинтересованные читатели могут найти альтернативные общедоступные наборы по ссылкам, указанным в приложении Б.

**Рис. 7.3.** Трехэтапный процесс тонкой настройки LLM

Следующий код (листинг 7.1) реализует и выполняет функцию загрузки этого набора данных, который представляет собой относительно небольшой файл (всего 204 Кбайт) в формате JSON (JavaScript Object Notation). Этот формат повторяет структуру словарей Python, предоставляя простую структуру для обмена данными, понятную как человеку, так и машине.

Листинг 7.1. Загрузка набора данных

```

import json
import os
import urllib

def download_and_load_file(file_path, url):
    if not os.path.exists(file_path):
        with urllib.request.urlopen(url) as response:
            text_data = response.read().decode("utf-8")
        with open(file_path, "w", encoding="utf-8") as file:
            file.write(text_data)
    else:
        with open(file_path, "r", encoding="utf-8") as file:
            text_data = file.read()
        with open(file_path, "r") as file:
            data = json.load(file)
        return data

```

← Пропускает загрузку, если файл уже был загружен

246 Глава 7. Тонкая настройка по инструкциям

```
file_path = "instruction-data.json"
url = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch"
    "/main/ch07/01_main-chapter-code/instruction-data.json"
)

data = download_and_load_file(file_path, url)
print("Number of entries:", len(data))
```

Результат выполнения предыдущего кода выглядит так:

```
Number of entries: 1100
```

Список данных, который мы загрузили из файла JSON, содержит 1100 записей инструкций. Выведем одну из записей, чтобы посмотреть, как она структурирована:

```
print("Example entry:\n", data[50])
```

Содержание примера:

```
Example entry:
{'instruction': 'Identify the correct spelling of the following word.',
 'input': 'Ocassion', 'output': "The correct spelling is 'Occasion.'"}

```

Как мы видим, элементы примера — это объекты словаря Python, содержащие 'instruction', 'input' и 'output'. Посмотрим на другой пример:

```
print("Another example entry:\n", data[999])
```

Здесь поле 'input' может иногда быть пустым:

```
Another example entry:
{'instruction': "What is an antonym of 'complicated'?",
 'input': '',
 'output': "An antonym of 'complicated' is 'simple.'"}

```

В тонкую настройку по инструкциям входит обучение модели на наборе данных, в котором явно указаны пары ввода-вывода, подобные тем, которые мы извлекли из файла JSON. Существуют различные методы форматирования этих записей для LLM. На рис. 7.4 показаны два различных формата примеров, часто называемых *стилями подсказок*, используемых при обучении известных LLM, таких как Алраса и Phi-3.

Алраса была одной из первых LLM с публично описанным процессом тонкой настройки по инструкциям. Phi-3, разработанная компанией Microsoft, включена в список, чтобы продемонстрировать разнообразие стилей подсказок.

В оставшейся части этой главы используется стиль подсказок Alpaca, поскольку он один из самых популярных, в основном потому, что помог определить первоначальный подход к тонкой настройке.

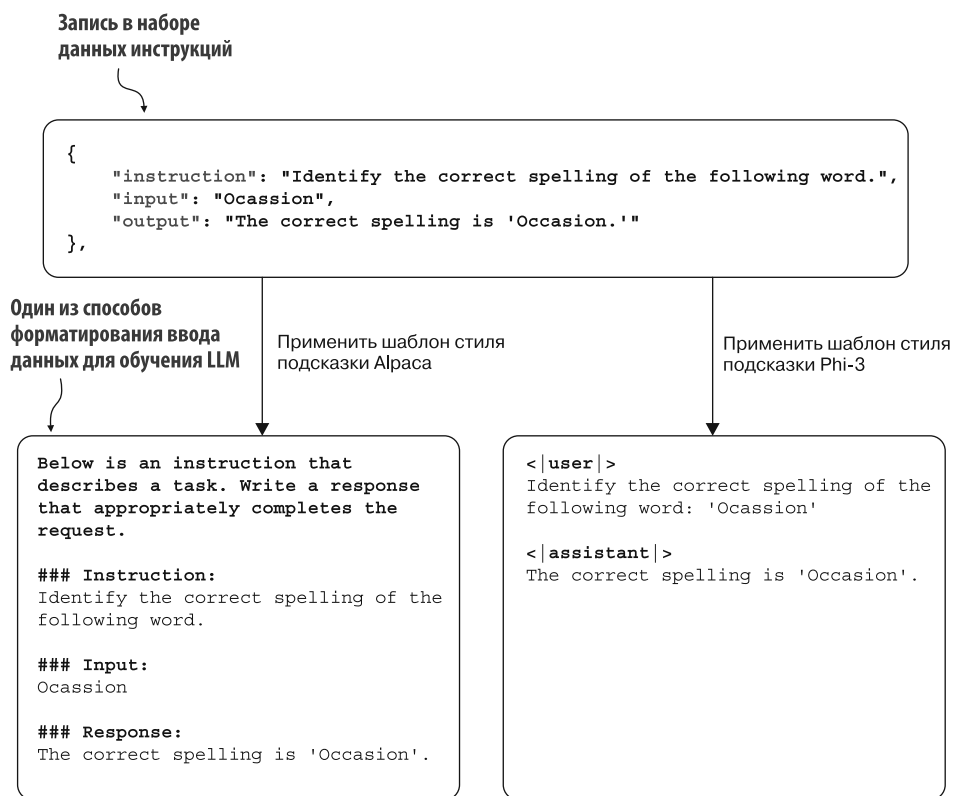


Рис. 7.4. Сравнение стилей подсказок для тонкой настройки по инструкциям. Стиль Alpaca (*слева*) использует структурированный формат с определенными разделами для инструкций, ввода и ответа. Стиль Phi-3 (*справа*) применяет более простой формат с назначенными токенами `<|user|>` и `<|assistant|>`

Упражнение 7.1. Изменение стилей подсказок

После тонкой настройки модели с помощью стиля подсказок Alpaca попробуйте стиль подсказок Phi-3 (см. рис. 7.4) и посмотрите, влияет ли он на качество ответов модели.

Теперь определим функцию `format_input`, которую можем использовать для преобразования записей в список `data` в стиле Алраса (листинг 7.2).

Листинг 7.2. Реализация функции форматирования подсказок

```
def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = (
        f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

Эта функция `format_input` принимает на вход `entry` из словаря и создает отформатированную строку. Протестируем ее на записи из набора данных `data[50]`, которую мы рассматривали ранее:

```
model_input = format_input(data[50])
desired_response = f"\n\n### Response:\n{data[50]['output']}"
print(model_input + desired_response)
```

Отформатированные входные данные выглядят так:

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:
Identify the correct spelling of the following word.

Input:
Ocassion

Response: The correct spelling is 'Occasion.'

Обратите внимание: функция `format_input` пропускает необязательный раздел `### Input:`, если поле `'input'` пусто. Мы можем проверить это, применив функцию `format_input` к записи `data[999]`, которую проверили ранее:

```
model_input = format_input(data[999])
desired_response = f"\n\n### Response:\n{data[999]['output']}"
print(model_input + desired_response)
```

Вывод показывает, что записи с пустым полем `'input'` не содержат раздел `### Input:` в отформатированном входе:

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

What is an antonym of 'complicated'?

Response:

An antonym of 'complicated' is 'simple'.

Прежде чем переходить к настройке загрузчиков данных PyTorch, разделим набор данных на обучающую, проверочную и тестовую выборки (листинг 7.3) аналогично тому, как мы это сделали с набором данных для классификации спама в предыдущей главе.

Листинг 7.3. Разделение набора данных

```
train_portion = int(len(data) * 0.85)  ← Использовать 85 % данных для обучения модели
test_portion = int(len(data) * 0.1)   ← Использовать 10 % для тестирования
val_portion = len(data) - train_portion - test_portion ← Использовать оставшиеся 5 % для проверки

train_data = data[:train_portion]
test_data = data[train_portion:train_portion + test_portion]
val_data = data[train_portion + test_portion:]

print("Training set length:", len(train_data))
print("Validation set length:", len(val_data))
print("Test set length:", len(test_data))
```

Такое разделение приводит к следующим размерам выборок:

```
Training set length: 935
Validation set length: 55
Test set length: 110
```

Теперь, успешно загрузив набор данных, разбив его на выборки и получив четкое представление о форматировании соответствующих подсказок, мы готовы к основной реализации процесса тонкой настройки по инструкциям. Далее мы сосредоточимся на разработке метода формирования обучающих пакетов, предназначенных для тонкой настройки LLM.

7.3. Организация данных в обучающие пакеты

Следующий наш шаг (рис. 7.5) направлен на эффективное формирование обучающих пакетов. Нам нужно определить метод, благодаря которому модель в процессе настройки получит отформатированные обучающие данные.

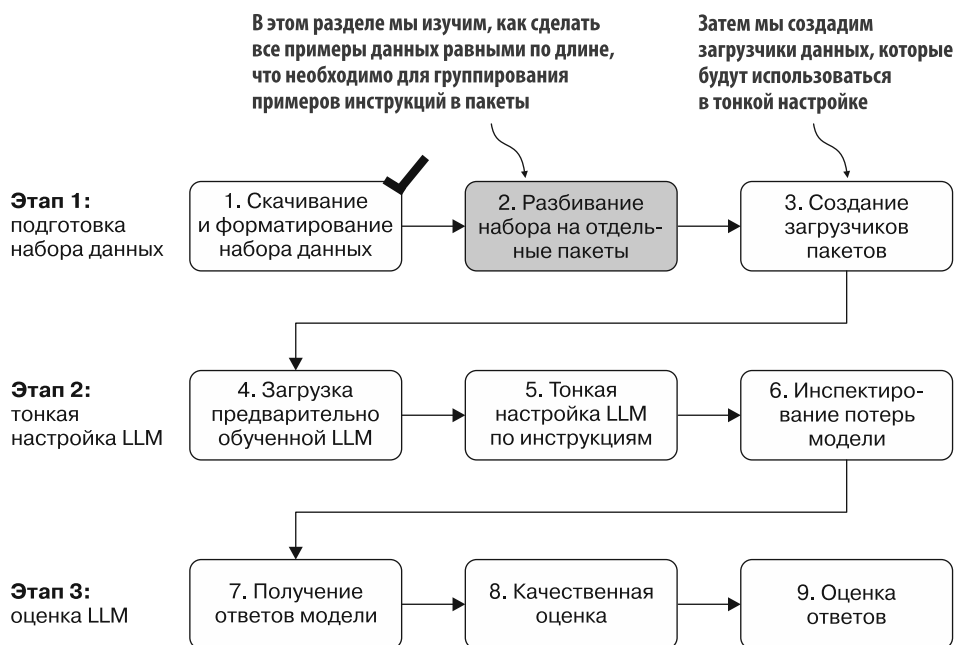


Рис. 7.5. Трехэтапный процесс тонкой настройки LLM. Далее мы рассмотрим второй шаг этапа 1: формирование обучающих пакетов

В предыдущей главе обучающие пакеты создавались автоматически с помощью класса `PyTorch DataLoader`, который использует функцию `collate` по умолчанию для объединения списков примеров в пакеты. Она отвечает за получение списка отдельных примеров данных и объединение их в один пакет, который модель может эффективно обработать во время обучения.

Однако процесс пакетной обработки для тонкой настройки по инструкциям является более сложным и требует создания собственной функции `collate`, которую мы позже подключим к `DataLoader`. Мы реализуем эту пользовательскую функцию для обработки конкретных требований и форматирования нашего набора данных.

Рассмотрим процесс *пакетной обработки*, в который входит и написание пользовательской функции `collate` (рис. 7.6). Сначала, чтобы реализовать мини-шаги 2.1 и 2.2, мы напишем класс `InstructionDataset` (листинг 7.4), который применяет функцию `format_input` и предварительно токенизирует все входные данные в наборе, аналогично `SpamDataset` в главе 6. Этот двухшаговый процесс, подробно описанный на рис. 7.7, реализован в методе `__init__` конструктора класса `InstructionDataset`.

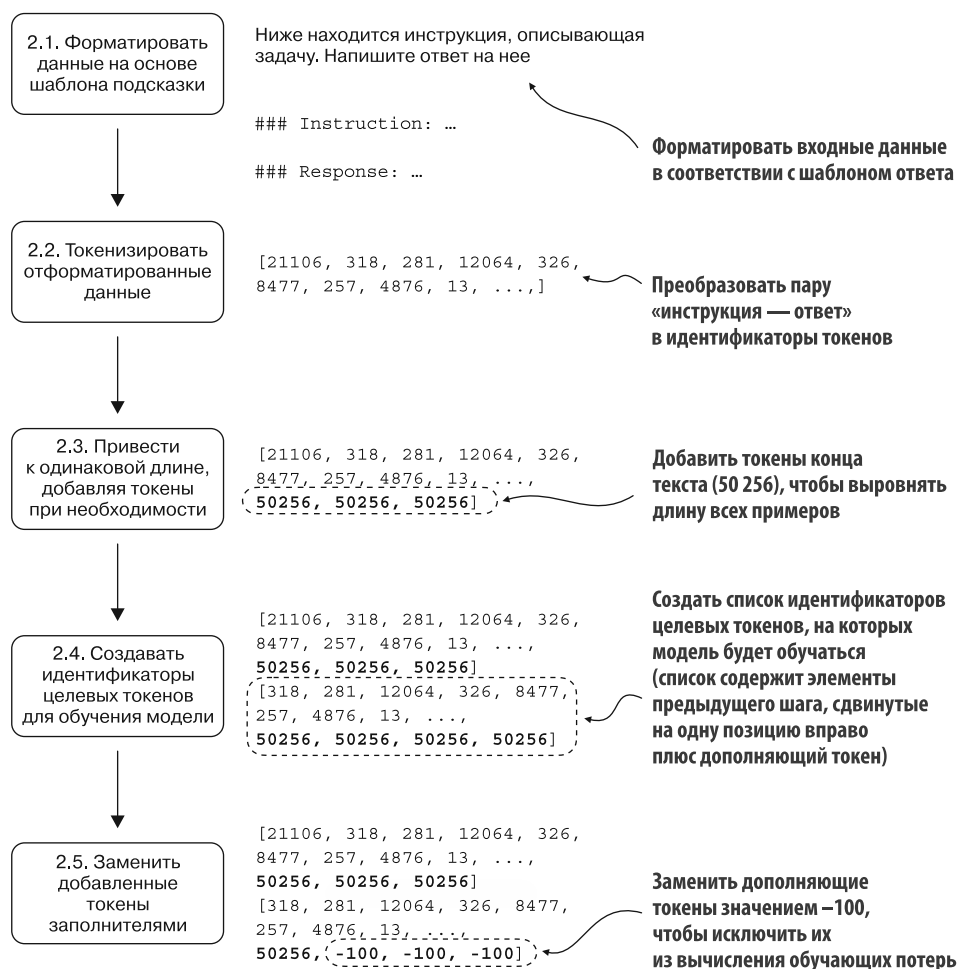


Рис. 7.6. Пять мини-шагов, необходимых для реализации пакетной обработки: 2.1) применение шаблона подсказки; 2.2) использование токенизации из предыдущих глав; 2.3) добавление дополнительных токенов; 2.4) создание целевых идентификаторов токенов; 2.5) замена дополнительных токенов на -100, чтобы замаскировать их в функции потерь

Подобно подходу, используемому для тонкой настройки по классификации, мы хотим ускорить обучение, собирая несколько обучающих примеров в пакет. Для этого нужно дополнить все входные последовательности так, чтобы их длина была одинаковой. Как и при тонкой настройке по классификации, мы используем `<|endoftext|>` в качестве заполняющего токена.

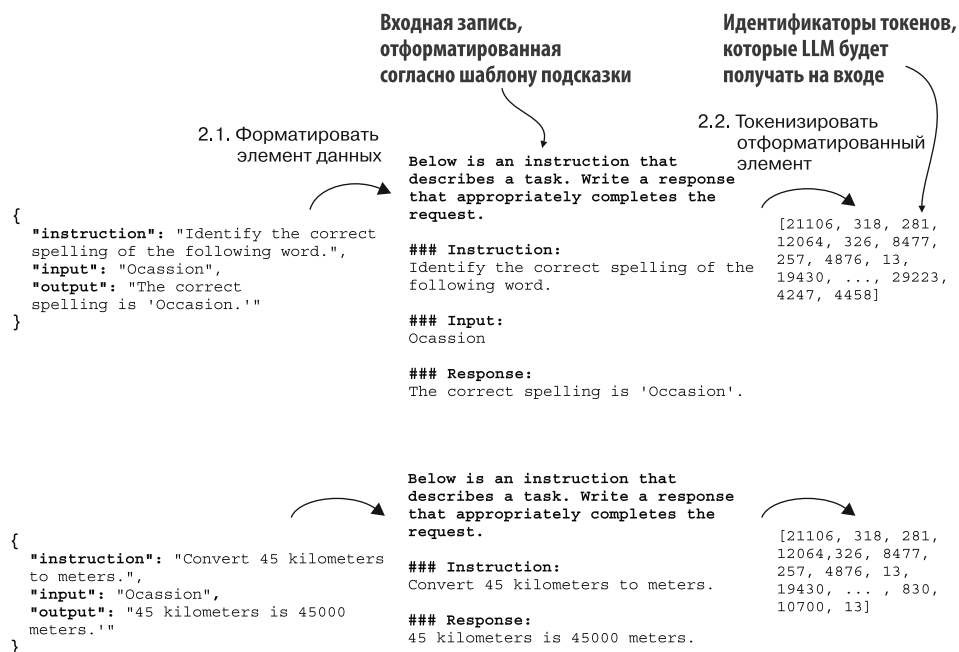


Рис. 7.7. Первые два мини-шага, необходимые для реализации пакетного процесса. Записи сначала форматируются с помощью специального шаблона подсказки (мини-шаг 2.1), а затем токенизируются (мини-шаг 2.2), в результате чего образуется последовательность идентификаторов токенов, которую может обработать модель

Листинг 7.4. Реализация класса набора инструкций

```
import torch
from torch.utils.data import Dataset

class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.encoded_texts = []
        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text
            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )

    def __getitem__(self, index):
        return self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```

Предварительно токенизирует тексты

Вместо того чтобы добавлять токены `<|endoftext|>` к текстовым входным данным, мы можем напрямую добавить идентификатор, соответствующий этому токenu, к предварительно токенизированному входным данным. Мы можем использовать метод токенизатора `.encode` для токена `<|endoftext|>`, чтобы запомнить, какой идентификатор следует использовать:

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"}))
```

Полученный идентификатор токена — 50256.

Переходя к мини-шагу 2.3 (см. рис. 7.6), мы применяем более сложный подход, разрабатывая пользовательскую функцию `collate`, которую мы можем передать загрузчику данных. Она приводит обучающие примеры в каждом пакете к одинаковой длине, позволяя при этом разным пакетам иметь разную длину (рис. 7.8). Такой подход сводит к минимуму ненужное дополнение, расширяя последовательности только до самой длинной в каждом пакете, а не во всем наборе данных.

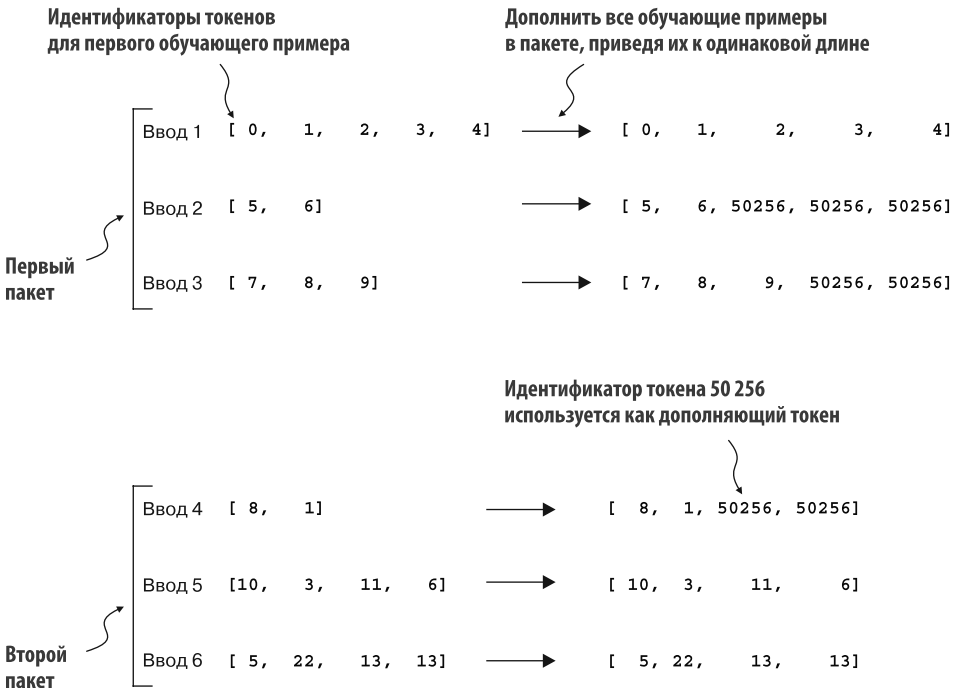


Рис. 7.8. Дополнение обучающих примеров в пакетах с использованием идентификатора токена 50 256, позволяющее обеспечить одинаковую длину каждого пакета

Мы можем реализовать процесс дополнения, используя следующую функцию:

```
def custom_collate_draft_1(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst = []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]

        padded = (
            new_item + [pad_token_id] *
            (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])
        inputs_lst.append(inputs)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    return inputs_tensor
```

Находит самую длинную последовательность в пакете

Дополняет и подготавливает входные данные

Удаляет токен, добавленный ранее

Преобразует список входных данных в тензор и посылает его на целевое устройство

Реализованная нами `custom_collate_draft_1` предназначена для интеграции в PyTorch `DataLoader`, но может использоваться и как самостоятельный инструмент. Здесь мы используем эту функцию независимо для тестирования и проверки того, что она работает должным образом. Попробуем ее на трех разных входных данных, которые мы хотим объединить в пакет, где каждый пример дополняется до одинаковой длины:

```
inputs_1 = [0, 1, 2, 3, 4]
inputs_2 = [5, 6]
inputs_3 = [7, 8, 9]
batch = (
    inputs_1,
    inputs_2,
    inputs_3
)
print(custom_collate_draft_1(batch))
```

Пакет, который мы получим, выглядит следующим образом:

```
tensor([[ 0,  1,  2,  3,  4],
        [ 5,  6, 50256, 50256, 50256],
        [ 7,  8,  9, 50256, 50256]])
```

Мы видим, что все входные данные были дополнены до длины самого длинного списка входных данных, `inputs_1`, содержащего пять идентификаторов токенов (рис. 7.9).

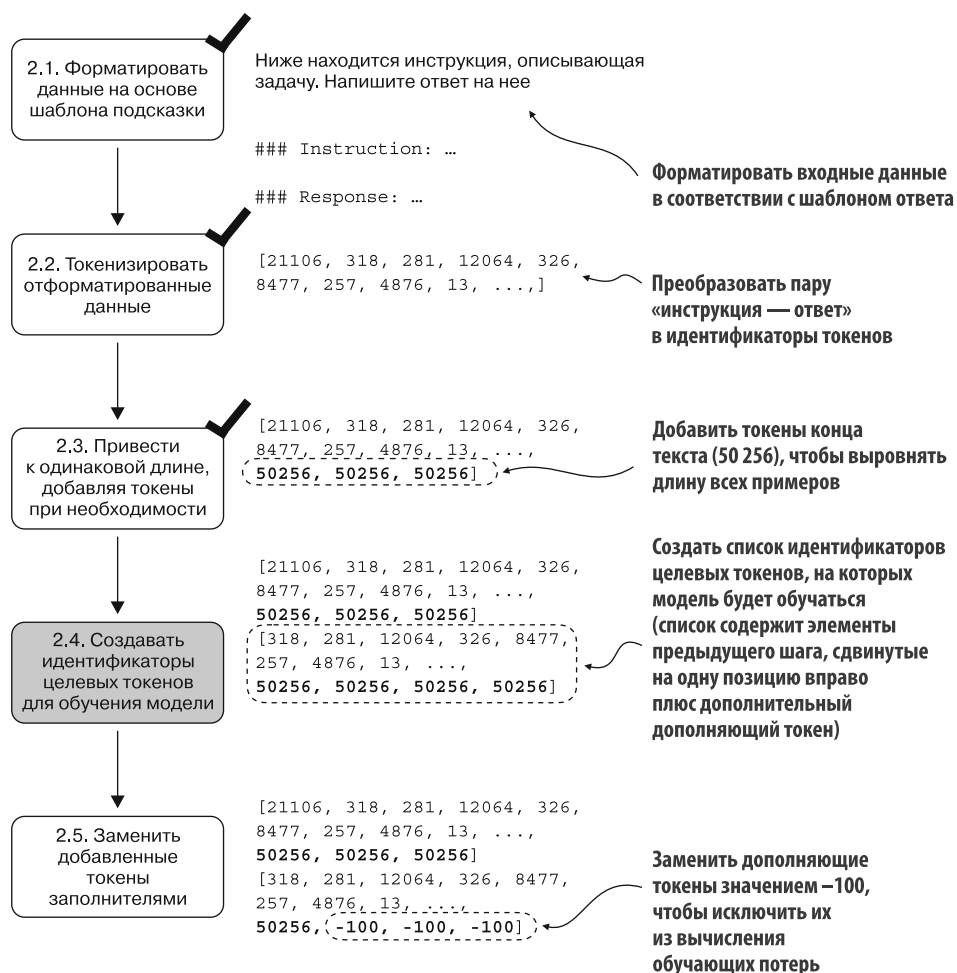


Рис. 7.9. Пять мини-шагов, необходимых для реализации процесса пакетной обработки. Сейчас мы сосредоточимся на мини-шаге 2.4, создании целевых идентификаторов токенов. Он важен, так как позволяет модели обучаться и прогнозировать токены, которые ей необходимо генерировать

Мы только что внедрили нашу первую пользовательскую функцию `collate` для создания пакетов из списков входных данных. Однако, как мы узнали ранее, нам также необходимо создавать пакеты с идентификаторами целевых токенов, соответствующими пакету входных идентификаторов. Эти идентификаторы имеют решающее значение, поскольку представляют то, что мы хотим от модели и что нам нужно во время тренировки для расчета потерь при обновлении ее весов.

То есть мы изменяем нашу пользовательскую функцию `collate`, чтобы она возвращала целевые идентификаторы токенов вдобавок к входным.

Как и в процессе предварительного обучения LLM, целевые идентификаторы токенов совпадают с входными, но смещены на одну позицию вправо. Эта настройка (рис. 7.10) позволяет LLM научиться предсказывать очередной символ в последовательности.

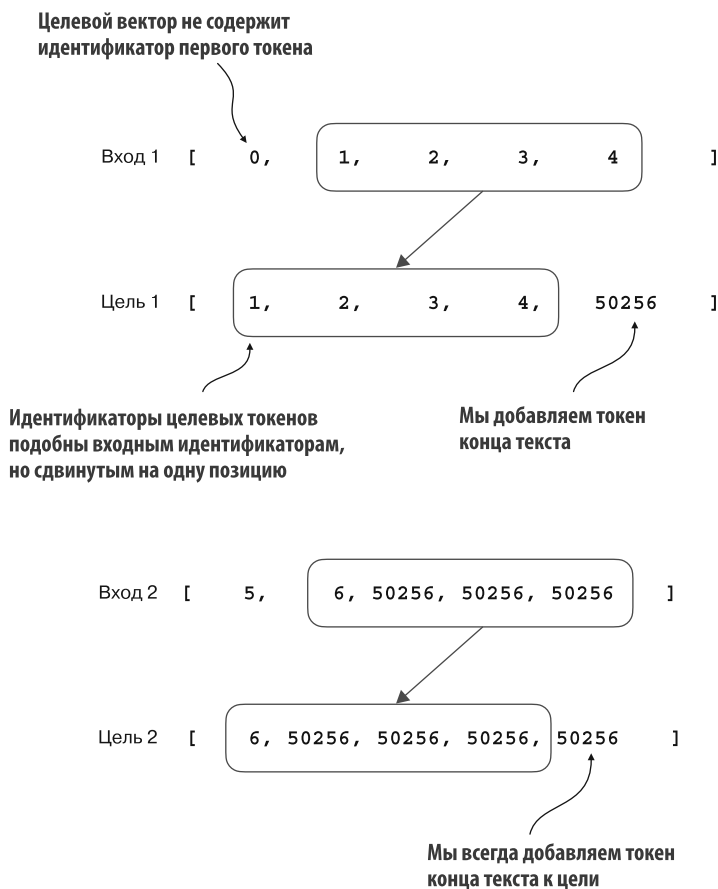


Рис. 7.10. Выравнивание входных и целевых токенов, используемое в процессе тонкой настройки LLM. Для каждой входной последовательности соответствующая целевая последовательность создается путем смещения идентификаторов токенов на одну позицию вправо, пропуска первого токена во входной последовательности и добавления токена конца текста

Следующая обновленная функция `collate` генерирует целевые идентификаторы токенов из входных идентификаторов токенов:

```
def custom_collate_draft_2(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]
        padded = (new_item + [pad_token_id] *
                  (batch_max_length - len(new_item))
                 )
        inputs = torch.tensor(padded[:-1])
        targets = torch.tensor(padded[1:])
        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor

inputs, targets = custom_collate_draft_2(batch)
print(inputs)
print(targets)
```

Отсекает последний
токен для входов

Смещение +1 вправо
для целей

Применительно к `batch`, состоящему из трех входных списков, которые мы определили ранее, новая функция `custom_collate_draft_2` теперь возвращает входной и целевой пакеты:

```
tensor([[ 0, 1, 2, 3, 4],
        [ 5, 6, 50256, 50256, 50256],
        [ 7, 8, 9, 50256, 50256]])
tensor([[ 1, 2, 3, 4, 50256],
        [ 6, 50256, 50256, 50256, 50256],
        [ 8, 9, 50256, 50256, 50256]])
```

Первый тензор
представляет входы

Второй тензор
представляет цели

На следующем мини-шаге мы присваиваем всем дополняющим токенам специальное значение `-100` (рис. 7.11). Оно позволяет исключить такие токены из расчета потерь при обучении, гарантируя, что на обучение модели влияют только значимые данные. Мы обсудим данный процесс более подробно после того, как реализуем это изменение. (При тонкой настройке по классификации нам не нужно было беспокоиться об этом, поскольку мы обучали модель только на основе последнего выходного токена.)



Рис. 7.11. Пять мини-шагов, необходимых для реализации процесса пакетной обработки. Создав целевую последовательность путем смещения идентификаторов токенов на одну позицию вправо и добавив токены конца текста на мини-шаге 2.5, мы заменяем токены конца текста значением-заполнителем (-100)

Однако обратите внимание, что мы сохраняем один токен конца текста с идентификатором 50256 в целевом списке (рис. 7.12). Его сохранение позволяет LLM научиться генерировать токен конца текста в ответ на инструкции, которые мы используем в качестве индикатора того, что сгенерированный ответ завершен.

В листинге 7.5 мы изменяем нашу пользовательскую функцию collate, чтобы заменить токены с идентификатором 50256 значением -100 в целевых списках. Кроме того, мы вводим параметр allowed_max_length, чтобы при необходимости ограничить длину текстов. Эта настройка будет полезна, если вы планируете работать с собственными наборами данных, размер контекста которых превышает 1024 токена, поддерживаемых моделью GPT-2.

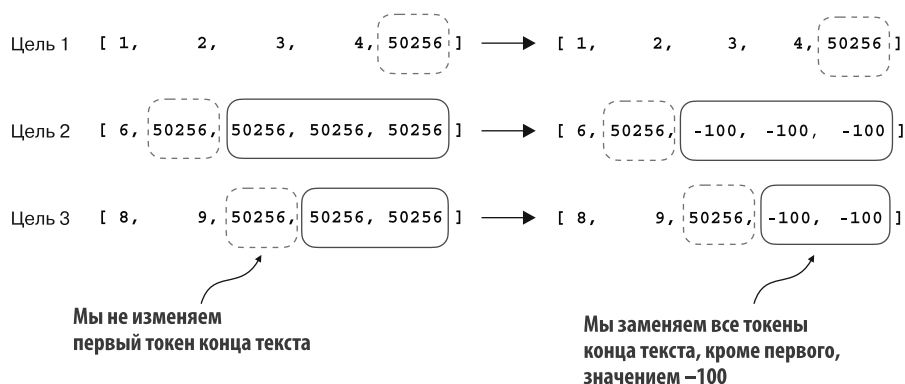


Рис. 7.12. Мини-шаг 2.4 в процессе замены токенов в целевом пакете для подготовки обучающих данных. Мы заменяем все токены конца текста, кроме первого, который используем в качестве дополнения, значением заполнителя -100, сохраняя при этом начальный token конца текста в каждой целевой последовательности

Листинг 7.5. Реализация пользовательской функции `collate` для пакетов

```
def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]

        padded = (
            new_item + [pad_token_id] *
            (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])
        targets = torch.tensor(padded[1:])

        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index

    if allowed_max_length is not None:
        inputs = inputs[:allowed_max_length]
        targets = targets[:allowed_max_length]
```

Дополняет последовательности до `max_length`

Отсекает последний token для входов

Смещение на одну позицию вправо для целей

Заменяет все дополняющие токены в целях, кроме первого, на `ignore_index`

Укорачивает последовательность до максимальной ожидаемой длины (необязательная операция)

```

inputs_lst.append(inputs)
targets_lst.append(targets)

inputs_tensor = torch.stack(inputs_lst).to(device)
targets_tensor = torch.stack(targets_lst).to(device)
return inputs_tensor, targets_tensor

```

Снова попробуем функцию `collate` на примере пакета, который создали ранее, чтобы убедиться, что она работает должным образом:

```

inputs, targets = custom_collate_fn(batch)
print(inputs)
print(targets)

```

Результаты: первый тензор представляет входные данные, а второй тензор — цели:

```

tensor([[ 0, 1, 2, 3, 4],
        [ 5, 6, 50256, 50256, 50256],
        [ 7, 8, 9, 50256, 50256]])
tensor([[ 1, 2, 3, 4, 50256],
        [ 6, 50256, -100, -100, -100],
        [ 8, 9, 50256, -100, -100]])

```

Измененная функция работает должным образом, меняя целевой список путем добавления идентификатора токена `-100`. Какова логика этой корректировки? Обсудим основную цель данного изменения.

В качестве демонстрации рассмотрим следующий простой и самодостаточный пример, в котором каждый выходной логит соответствует потенциальному токenu из словаря модели. Ниже показан способ, с помощью которого можно вычислить потери перекрестной энтропии (введенной в главе 5) во время обучения, когда модель предсказывает последовательность токенов. Это похоже на то, что мы делали при предварительном обучении модели и ее тонкой настройке для классификации:

```

logits_1 = torch.tensor(
    [[-1.0, 1.0], ← Предсказания для первого токена
     [-0.5, 1.5]] ← Предсказания для второго токена
)
targets_1 = torch.tensor([0, 1]) # Правильные индексы токенов для генерации
loss_1 = torch.nn.functional.cross_entropy(logits_1, targets_1)
print(loss_1)

```

Потери, вычисленные предыдущим кодом, равны **1.1269**:

```
tensor(1.1269)
```

Как и ожидалось, добавление дополнительного идентификатора токена повлияло на вычисление потерь:


```
logits_2 = torch.tensor(
    [[-1.0, 1.0],
     [-0.5, 1.5],
     [-0.5, 1.5]]
)
targets_2 = torch.tensor([0, 1, 1])
loss_2 = torch.nn.functional.cross_entropy(logits_2, targets_2)
print(loss_2)
```

Предсказание идентификатора
нового третьего токена

После добавления третьего токена значение потерь составляет 0.7936.

До сих пор мы выполняли более или менее очевидные вычисления с помощью функции перекрестной энтропии (в PyTorch) для потерь, той же самой, которую использовали ранее в функциях для предварительного обучения и тонкой настройки для классификации. Теперь посмотрим, что произойдет, если мы заменим третий целевой токен значением `-100`:

```
targets_3 = torch.tensor([0, 1, -100])
loss_3 = torch.nn.functional.cross_entropy(logits_2, targets_3)
print(loss_3)
print("loss_1 == loss_3:", loss_1 == loss_3)
```

Получаем такой результат:

```
tensor(1.1269)
loss_1 == loss_3: tensor(True)
```

Результирующие потери на этих трех обучающих примерах идентичны потерям, которые мы рассчитали ранее на двух обучающих примерах. Другими словами, функция потерь перекрестной энтропии проигнорировала третью запись в векторе `targets_3`, идентификатор токена, соответствующий `-100`. (Заинтересованные читатели могут попробовать заменить значение `-100` другим идентификатором токена, отличным от 0 или 1; это приведет к ошибке.)

Итак, что же такого особенного в значении `-100`, что оно игнорируется функцией потерь перекрестной энтропии? По умолчанию эта функция в PyTorch имеет параметр `cross_entropy(..., ignore_index=-100)`. Это значит, что она игнорирует цели, помеченные `-100`. Мы используем параметр `ignore_index`, чтобы игнорировать дополнительные токены в конце текста, с помощью которых мы дополняли обучающие примеры до одинаковой длины в каждом пакете. Однако мы хотим сохранить один идентификатор токена 50256 (конец текста) в целевых данных, поскольку он помогает LLM научиться генерировать токены конца текста, которые мы можем использовать в качестве индикатора того, что ответ завершен.

Помимо токенов дополнения, часто маскируют идентификаторы целевых токенов, которые соответствуют инструкции (рис. 7.13). При маскировке таких

токенов функция потерь перекрестной энтропии вычисляется только для сгенерированных целевых идентификаторов ответов. Таким образом, модель обучается генерировать точные ответы, а не запоминать инструкции, что может помочь уменьшить переобучение.

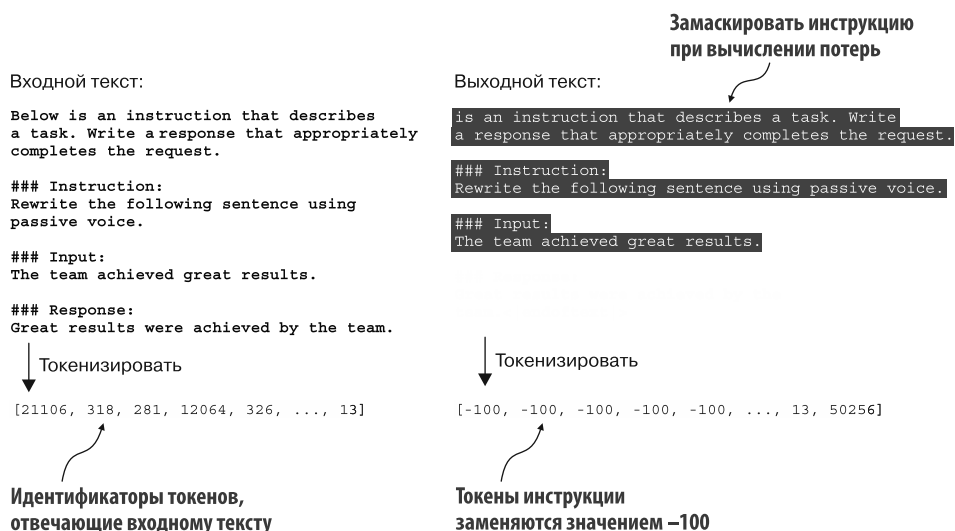


Рис. 7.13. Слева: отформатированный входной текст, который мы разбиваем на токены, а затем передаем в LLM во время обучения. Справа: целевой текст, который мы подготавливаем для LLM, где при необходимости можно замаскировать раздел инструкций, заменив соответствующие идентификаторы токенов значением -100 (ignore_index)

На момент написания этой главы мнения исследователей о том, является ли маскировка инструкций универсальной полезной функцией при тонкой настройке по инструкциям, разделились. Например, в статье 2024 года *Instruction Tuning With Loss Over Instructions* Ши и др. (<https://arxiv.org/abs/2405.14394>) было показано, что отсутствие маскировки инструкций повышает производительность LLM (подробнее см. в приложении Б). Мы не будем применять маскировку и оставим это в качестве дополнительного упражнения для заинтересованных читателей.

Упражнение 7.2. Маскировка инструкций и входных данных

После завершения работы над главой и тонкой настройки модели с помощью InstructionDataset замените токены инструкций и входных данных маской -100, чтобы использовать метод маскировки инструкций, показанный на рис. 7.13. Затем оцените, оказывает ли это положительное влияние на производительность модели.

7.4. Создание загрузчиков данных для набора инструкций

Мы завершили несколько этапов реализации класса `InstructionDataset` и функции `custom_collate_fn` для набора инструкций. Теперь мы готовы пожинать плоды своего труда, просто подключив объекты `InstructionDataset` и функцию `custom_collate_fn` к загрузчикам данных PyTorch (рис. 7.14). Они будут автоматически перемешивать и упорядочивать пакеты для процесса тонкой настройки по инструкциям.



Рис. 7.14. Трехэтапный процесс тонкой настройки LLM по инструкциям. К этому моменту мы подготовили набор данных и реализовали пользовательскую функцию `collate` для пакетной обработки инструкций. Теперь мы можем создать и применить загрузчики данных для обучающей, проверочной и тестовой выборки, необходимых для тонкой настройки модели по инструкциям и ее оценки

Прежде чем мы реализуем этап создания загрузчика данных, нужно коротко обсудить настройку переменной `device` в функции `custom_collate_fn`. Эта функция содержит код для перемещения входных и целевых тензоров (например, `torch.stack(inputs_lst).to(device)`) на указанное устройство, которым может быть "cpu" или "cuda" (для графических процессоров NVIDIA) или, при необходимости, "mps" для компьютеров Mac с чипами Apple Silicon.

ПРИМЕЧАНИЕ Использование устройства «mps» может привести к количественным различиям по сравнению с содержанием этой главы, поскольку поддержка Apple Silicon в PyTorch все еще является экспериментальной.

Ранее мы переносили данные на целевое устройство (например, в память GPU, когда `device="cuda"`) в основном цикле обучения. Такое действие в составе функции `collate` предотвращает блокировку графического процессора во время обучения модели, так как перенос выполняется в фоновом режиме.

Следующий код инициализирует переменную `device`:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# if torch.backends.mps.is_available():
#     device = torch.device("mps")
print("Device:", device)
```

Раскомментируйте эти две строки, чтобы использовать графический процессор на чипе Apple Silicon

В зависимости от вашего компьютера будет выведено либо `"Device: cpu"`, либо `"Device: cuda"`.

Далее выбранное устройство в функции `custom_collate_fn` можно использовать повторно. Для этого, подключая его к классу PyTorch `DataLoader`, мы используем частичную функцию из стандартной библиотеки Python `functools`, чтобы создать новую версию функции с предварительно заполненным аргументом устройства. Кроме того, мы задаем допустимую максимальную длину — 1024, что ограничивает данные до максимальной длины контекста, поддерживаемой моделью GPT-2, которую мы будем настраивать позже:

```
from functools import partial

customized_collate_fn = partial(
    custom_collate_fn,
    device=device,
    allowed_max_length=1024
)
```

Далее мы можем настроить загрузчики данных, как и раньше, но на этот раз будем использовать нашу пользовательскую функцию `collate` для пакетной обработки (листинг 7.6).

Листинг 7.6. Инициализация загрузчиков данных

```
from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8

torch.manual_seed(123)

train_dataset = InstructionDataset(train_data, tokenizer)
train_loader = DataLoader(
```

Вы можете попробовать увеличить это число, если параллельные процессы Python поддерживаются вашей операционной системой

```

        train_dataset,
        batch_size=batch_size,
        collate_fn=customized_collate_fn,
        shuffle=True,
        drop_last=True,
        num_workers=num_workers
    )

    val_dataset = InstructionDataset(val_data, tokenizer)
    val_loader = DataLoader(
        val_dataset,
        batch_size=batch_size,
        collate_fn=customized_collate_fn,
        shuffle=False,
        drop_last=False,
        num_workers=num_workers
    )

    test_dataset = InstructionDataset(test_data, tokenizer)
    test_loader = DataLoader(
        test_dataset,
        batch_size=batch_size,
        collate_fn=customized_collate_fn,
        shuffle=False,
        drop_last=False,
        num_workers=num_workers
    )

```

Рассмотрим размеры входных и целевых пакетов, сгенерированных загрузчиком:

```

print("Train loader:")
for inputs, targets in train_loader:
    print(inputs.shape, targets.shape)

```

Результат (сокращен в целях экономии места на странице) выглядит так:

```

Train loader:
torch.Size([8, 61]) torch.Size([8, 61])
torch.Size([8, 76]) torch.Size([8, 76])
torch.Size([8, 73]) torch.Size([8, 73])
...
torch.Size([8, 74]) torch.Size([8, 74])
torch.Size([8, 69]) torch.Size([8, 69])

```

Этот вывод показывает, что первый входной и целевой пакеты имеют размеры 8×61 , где 8 — размер пакета, а 61 — количество токенов в каждом обучающем примере в данном пакете. Во втором входном и целевом пакетах другое количество токенов — например, 76. Благодаря нашей пользовательской функции загрузчик данных может создавать пакеты разной длины. В следующем разделе мы загрузим предварительно обученную модель, которую затем дообучим с помощью нашего загрузчика данных.

7.5. Загрузка предварительно обученной LLM

Мы потратили много времени на подготовку набора данных для тонкой настройки по инструкциям — ключевого элемента контролируемого процесса тонкой настройки. Многие другие аспекты — такие же, как и при предварительном обучении, что позволяет нам повторно использовать большую часть кода из предыдущих глав.

Прежде чем приступить к тонкой настройке по инструкциям, мы должны сначала загрузить предварительно обученную модель GPT, которую хотим настроить (рис. 7.15). Этот процесс мы уже выполняли ранее. Однако вместо того, чтобы использовать самую маленькую модель с 124 млн параметров, мы загружаем модель среднего размера с 355 млн параметров. Причина такого выбора в том, что модель с 124 млн параметров слишком ограничена по возможностям для достижения удовлетворительных результатов с помощью тонкой настройки по инструкциям. В частности, более мелким моделям не хватает необходимых возможностей для изучения и запоминания сложных закономерностей и нюансов, требуемых для выполнения высококачественных задач по следованию инструкциям.



Рис. 7.15. Трехэтапный процесс тонкой настройки LLM по инструкциям. После подготовки набора данных начинается процесс тонкой настройки модели, чтобы она могла выполнять инструкции, — загружается предварительно обученная LLM, которая служит основой для последующего обучения

Для загрузки наших предварительно обученных моделей требуется тот же код, что и при предварительном обучении данных (см. раздел 5.5) и тонкой настройке по классификации (см. раздел 6.4), за исключением того, что теперь мы указываем `gpt2-medium` (355M) вместо `gpt2-small` (124M) (листинг 7.7).

ПРИМЕЧАНИЕ В результате выполнения этого кода запустится загрузка модели GPT среднего размера с объемом занимаемой памяти около 1,42 Гбайт. Это примерно в три раза больше, чем объем памяти, необходимый для малой модели.

Листинг 7.7. Загрузка предварительно обученной модели

```
from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt

BASE_CONFIG = {
    "vocab_size": 50257,      # Размер словаря
    "context_length": 1024,  # Длина контекста
    "drop_rate": 0.0,        # Процент отсева
    "qkv_bias": True         # Смещение запроса-ключа-значения
}

model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}

CHOOSE_MODEL = "gpt2-medium (355M)"
BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")

settings, params = download_and_load_gpt2(
    model_size=model_size,
    models_dir="gpt2"
)

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval();
```

После выполнения кода будет загружено несколько файлов:

```
checkpoint: 100%|██████████| 77.0/77.0 [00:00<00:00, 156kiB/s]
encoder.json: 100%|██████████| 1.04M/1.04M [00:02<00:00, 467kiB/s]
hparams.json: 100%|██████████| 91.0/91.0 [00:00<00:00, 198kiB/s]
model.ckpt.data-00000-of-00001: 100%|██████████| 1.42G/1.42G
[05:50<00:00, 4.05MiB/s]
model.ckpt.index: 100%|██████████| 10.4k/10.4k [00:00<00:00, 18.1MiB/s]
model.ckpt.meta: 100%|██████████| 927k/927k [00:02<00:00, 454kiB/s]
vocab.bpe: 100%|██████████| 456k/456k [00:01<00:00, 283kiB/s]
```

Теперь оценим производительность предварительно обученной LLM на одной из задач проверки, сравнив результат с ожидаемым ответом. Это даст нам базовое представление о том, насколько хорошо модель справляется с задачей следования инструкциям сразу после обучения, до тонкой настройки, и поможет оценить эффект от тонкой настройки в дальнейшем. Чтобы выполнить оценку, используем первый пример из проверочной выборки:

```
torch.manual_seed(123)
input_text = format_input(val_data[0])
print(input_text)
```

Инструкция будет следующей:

Below is an instruction that describes a task. Write a response that appropriately completes the request.

```
### Instruction:
Convert the active sentence to passive: 'The chef cooks the meal every day.'
```

Далее мы генерируем ответ модели, используя ту же функцию `generate`, которую применяли для предварительного обучения в главе 5:

```
from chapter05 import generate, text_to_token_ids, token_ids_to_text

token_ids = generate(
    model=model,
    idx=text_to_token_ids(input_text, tokenizer),
    max_new_tokens=35,
    context_size=BASE_CONFIG["context_length"],
    eos_id=50256,
)
generated_text = token_ids_to_text(token_ids, tokenizer)
```

Функция `generate` возвращает объединенный входной и выходной текст. Такое поведение было ранее удобным, поскольку предварительно обученные LLM в первую очередь предназначены для дополнения текста, где входной и выходной текст объединяются для создания связного и разборчивого текста. Однако при оценке работы модели над конкретной задачей мы часто хотим сосредоточиться исключительно на сгенерированном моделью ответе.

Чтобы выделить текст ответа модели, нужно вычесть длину входной инструкции из начала `generated_text`:

```
response_text = generated_text[len(input_text):].strip()
print(response_text)
```

Этот код удаляет входной текст из начала `generated_text`, оставляя только сгенерированный моделью ответ. Затем применяется функция `strip()` для удаления начальных и конечных пробелов. Результат выглядит так:

Response:

The chef cooks the meal every day.

Instruction:

Convert the active sentence to passive: 'The chef cooks the

Этот вывод показывает, что предварительно обученная модель пока не способна правильно следовать данной инструкции. Она создает раздел Response, но просто повторяет исходное предложение и часть инструкции, не преобразуя активный залог в страдательный, как было запрошено. Далее мы реализуем процесс тонкой настройки, чтобы улучшить способность модели понимать такие запросы и правильно отвечать на них.

7.6. Тонкая настройка LLM по инструкциям

Пришло время дообучить LLM, чтобы она могла работать с инструкциями (рис. 7.16). Мы загрузим предварительно обученную модель, описанную в предыдущем разделе, и продолжим ее обучение, используя ранее подготовленный набор инструкций, описанный выше в этой главе.



Рис. 7.16. Трехэтапный процесс тонкой настройки LLM по инструкциям. На шаге 5 мы обучаем предварительно обученную модель, которую загрузили ранее, используя подготовленный ранее набор инструкций

Мы уже проделали всю тяжелую работу, когда реализовали обработку набора данных с инструкциями в начале текущей главы. Для самого процесса тонкой настройки мы можем повторно использовать функции расчета потерь и обучения, реализованные в главе 5:

```
from chapter05 import (
    calc_loss_loader,
    train_model_simple
)
```

Работа с аппаратными ограничениями

Использование и обучение более крупной модели, такой как GPT-2 medium (355 млн параметров), требует больше вычислительных ресурсов, чем меньшая модель GPT-2 (124 млн параметров). Если вы столкнулись с проблемами из-за аппаратных ограничений, то можете переключиться на меньшую модель, изменив `CHOOSE_MODEL = "gpt2-medium (355M)"` на `CHOOSE_MODEL = "gpt2-small (124M)"` (см. раздел 7.5). В качестве альтернативы, чтобы ускорить обучение модели, рассмотрите возможность использования GPU. В дополнительном разделе репозитория кода этой книги перечислены несколько вариантов использования облачных графических процессоров: <https://mng.bz/EOEq>.

В таблице ниже указано примерное время обучения каждой модели на различных устройствах, в том числе центральных и графических процессорах, для GPT-2. Запуск этого кода на совместимом GPU не требует изменений в коде и может значительно ускорить обучение. Для получения результатов, показанных в текущей главе, я использовал среднюю модель GPT-2 и обучал ее на графическом процессоре A100.

Название модели	Устройство	Время выполнения двух эпох, мин
gpt2-medium (355M)	CPU (M3 MacBook Air)	15,78
gpt2-medium (355M)	GPU (NVIDIA L4)	1,83
gpt2-medium (355M)	GPU (NVIDIA A100)	0,86
gpt2-small (124M)	CPU (M3 MacBook Air)	5,74
gpt2-small (124M)	GPU (NVIDIA L4)	0,69
gpt2-small (124M)	GPU (NVIDIA A100)	0,39

Прежде чем приступить к обучению, рассчитаем начальные потери для обучающей и проверочной выборок:

```
model.to(device)
torch.manual_seed(123)

with torch.no_grad():
    train_loss = calc_loss_loader(
```

```

        train_loader, model, device, num_batches=5
    )
    val_loss = calc_loss_loader(
        val_loader, model, device, num_batches=5
    )

print("Training loss:", train_loss)
print("Validation loss:", val_loss)

```

Начальные значения потерь таковы (как и прежде, наша цель — минимизировать потери):

```

Training loss: 3.825908660888672
Validation loss: 3.7619335651397705

```

Подготовив модель и загрузчики данных, мы можем приступить к обучению модели. Код, приведенный в листинге 7.8, настраивает процесс обучения, в который входит инициализация оптимизатора, установка количества эпох, а также определение частоты и начального контекста, позволяющее оценить сгенерированные ответы LLM во время обучения на основе первой инструкции проверочной выборки (`val_data[0]`), которую мы рассмотрели в разделе 7.5.

Листинг 7.8. Инструкция по тонкой настройке предварительно обученной LLM

```

import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(
    model.parameters(), lr=0.00005, weight_decay=0.1
)
num_epochs = 2

train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context=format_input(val_data[0]), tokenizer=tokenizer
)

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")

```

Вывод выглядит так:

```

Ep 1 (Step 000000): Train loss 2.637, Val loss 2.626
Ep 1 (Step 000005): Train loss 1.174, Val loss 1.103
Ep 1 (Step 000010): Train loss 0.872, Val loss 0.944
Ep 1 (Step 000015): Train loss 0.857, Val loss 0.906
...

```

```

Ep 1 (Step 000115): Train loss 0.520, Val loss 0.665
Below is an instruction that describes a task. Write a response that
appropriately completes the request. ### Instruction: Convert the active
sentence to passive: 'The chef cooks the meal every day.'
### Response: The meal is prepared every day by the chef. <|endoftext|>
The following is an instruction that describes a task.
Write a response that appropriately completes the request.
### Instruction: Convert the active sentence to passive:
Ep 2 (Step 000120): Train loss 0.438, Val loss 0.670
Ep 2 (Step 000125): Train loss 0.453, Val loss 0.685
Ep 2 (Step 000130): Train loss 0.448, Val loss 0.681
Ep 2 (Step 000135): Train loss 0.408, Val loss 0.677
...
Ep 2 (Step 000230): Train loss 0.300, Val loss 0.657
Below is an instruction that describes a task. Write a response that
appropriately completes the request. ### Instruction:
Convert the active sentence to passive: 'The chef cooks the meal
every day.' ### Response: The meal is cooked every day by the
chef. <|endoftext|>The following is an instruction that describes
a task. Write a response that appropriately completes the request.
### Instruction: What is the capital of the United Kingdom
Training completed in 0.87 minutes.

```

Результаты показывают, что модель эффективно обучается, о чем можно судить по последовательно снижающимся значениям потерь при обучении и проверке в течение двух эпох. Это говорит о том, что модель постепенно улучшает свою способность понимать и выполнять предоставленные инструкции. (Поскольку модель продемонстрировала эффективное обучение в течение этих двух эпох, продление обучения до третьей или более эпох не является необходимым и может даже быть контрпродуктивным, поскольку может привести к усилению переобучения.)

Более того, сгенерированные ответы в конце каждой эпохи позволяют проверить прогресс модели по части правильного выполнения данной задачи на примере набора проверок. В этом случае модель успешно преобразует предложение в активном залоге "The chef cooks the meal every day." в его аналог, имеющий форму страдательного залога: "The meal is cooked every day by the chef.".

Мы вернемся к рассмотрению и оценим качество ответов модели более подробно позже. Теперь рассмотрим кривые потерь при обучении и проверке, чтобы получить дополнительные сведения о процессе обучения модели. Для этого мы используем ту же функцию `plot_losses`, что и при предварительном обучении:

```

from chapter05 import plot_losses
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)

```

Из графика потерь (рис. 7.17) мы видим, что производительность модели как на обучающей, так и на проверочной выборке существенно улучшается в процессе

обучения. Быстрое снижение потерь на начальном этапе указывает на то, что модель быстро распознает значимые закономерности и представления в данных. Затем, по мере перехода ко второй эпохе обучения, потери продолжают снижаться, но медленнее. Это говорит о том, что модель дорабатывает полученные представления и приближается к стабильному решению.

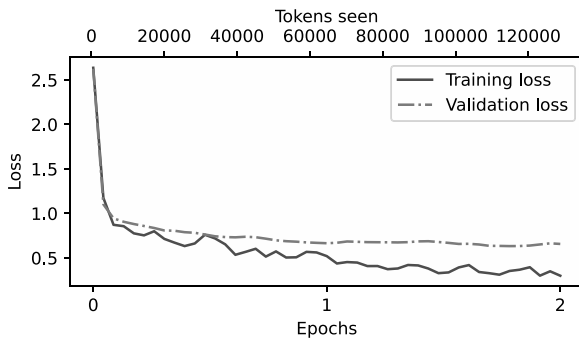


Рис. 7.17. Тренды потерь при обучении и проверке в течение двух эпох. Сплошной линией обозначены потери при обучении, которые резко снижаются, а затем стабилизируются, а пунктирной — потери при проверке, имеющие аналогичную тенденцию

Этот график потерь указывает на то, что модель обучается эффективно, однако наиболее важным аспектом является ее производительность с точки зрения качества и правильности ответов.

Далее мы извлечем ответы и сохраним их в формате, который позволит оценить и количественно измерить качество ответов.

Упражнение 7.3. Тонкая настройка на исходном наборе данных Alpaca

Набор данных Alpaca, созданный исследователями из Стэнфорда, — один из самых ранних и популярных наборов инструкций, находящихся в открытом доступе. В качестве альтернативы файлу `instruction-data.json`, который мы используем в данной книге, рассмотрите возможность тонкой настройки LLM на этом наборе. Он доступен по адресу <https://mng.bz/NBnE>.

Набор содержит 52 002 записи, что примерно в 50 раз превышает содержимое наборов, которые мы использовали, и большинство записей длиннее. Поэтому я настоятельно рекомендую использовать для обучения GPU, что ускорит процесс тонкой настройки. Если вы столкнетесь с ошибкой нехватки памяти, то попробуйте уменьшить размер пакета с 8 до 4, 2 или даже 1. Уменьшение допустимой максимальной длины с 1024 до 512 или 256 также может помочь решить проблемы с памятью.

7.7. Извлечение и сохранение ответов

Выполнив тонкую настройку LLM на обучающей выборке набора инструкций, мы готовы оценить производительность модели, используя тестовую выборку. Сначала мы извлекаем ответы, сгенерированные моделью, для каждого входного значения в тестовой выборке и собираем их для анализа, а затем оцениваем LLM, чтобы количественно оценить качество ответов (рис. 7.18).

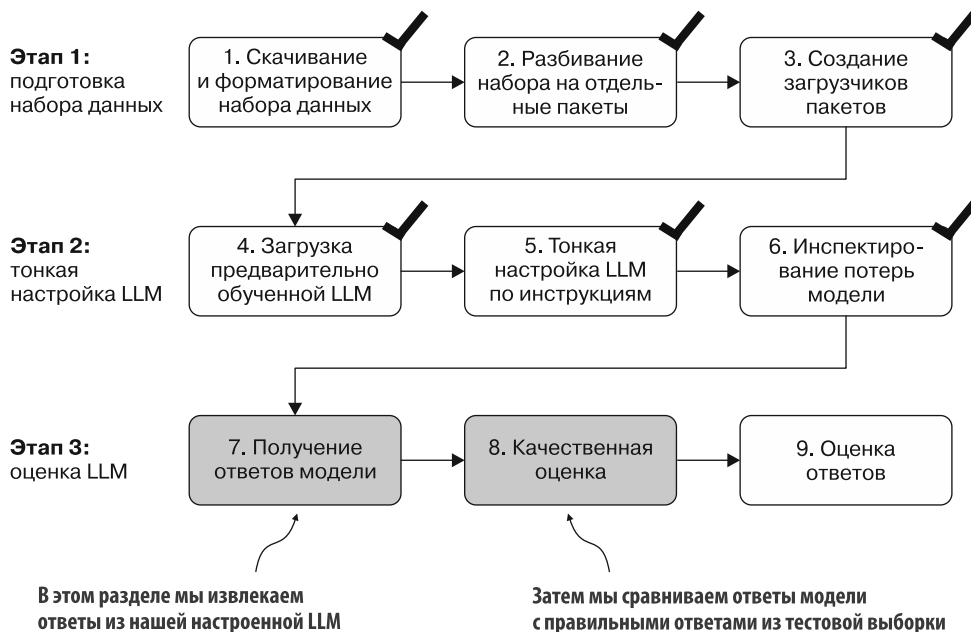


Рис. 7.18. Трехэтапный процесс тонкой настройки LLM с помощью инструкций. На первых двух шагах этапа 3 мы извлекаем ответы модели на тестовой выборке и собираем их, чтобы провести их анализ, а затем оцениваем модель, чтобы количественно оценить эффективность модели, точно настроенной по инструкциям

Чтобы завершить этап создания инструкции по ответу, мы используем функцию `generate`. Затем мы выводим ответы модели вместе с ожидаемыми ответами на первые три задания из тестового набора, представляя их вместе для сравнения:

```

torch.manual_seed(123)

for entry in test_data[:3]:
    input_text = format_input(entry)
    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
    )

```

Проходит в цикле по первым трем тестовым примерам

Использует функцию `generate` из раздела 7.5

```

max_new_tokens=256,
context_size=BASE_CONFIG["context_length"],
eos_id=50256
)
generated_text = token_ids_to_text(token_ids, tokenizer)

response_text = (
    generated_text[len(input_text):]
    .replace("### Response:", "")
    .strip()
)

print(input_text)
print(f"\nCorrect response:\n>> {entry['output']}")
print(f"\nModel response:\n>> {response_text.strip()}")
print("-----")

```

Как упоминалось ранее, функция `generate` возвращает объединенный входной и выходной текст, поэтому мы используем нарезку и метод `.replace()` для содержимого `generated_text`, чтобы извлечь ответ модели. Далее приведены инструкции, за которыми следует ответ из тестовой выборки и ответ модели.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

Rewrite the sentence using a simile.

Input:

The car is very fast.

Correct response:

>> The car is as fast as lightning.

Model response:

>> The car is as fast as a bullet.

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

What type of cloud is typically associated with thunderstorms?

Correct response:

>> The type of cloud typically associated with thunderstorms is cumulonimbus.

Model response:

```
>> The type of cloud associated with thunderstorms is a cumulus cloud.
```

Below is an instruction that describes a task. Write a response that appropriately completes the request.

Instruction:

Name the author of 'Pride and Prejudice.'

Correct response:

```
>> Jane Austen.
```

Model response:

```
>> The author of 'Pride and Prejudice' is Jane Austen.
```

Как мы видим, судя по инструкциям тестовой выборки, заданным ответам и ответам модели, они работают относительно хорошо. Ответы на первую и последнюю инструкции явно правильные, а ответ на вторую близок к правильному, но не совсем точен. Модель отвечает *cumulus cloud* вместо *cumulonimbus*, хотя стоит отметить, что кучевые облака могут превращаться в кучево-дождевые, которые способны вызывать грозы.

Самое главное, что выполнить оценку модели не так просто, как при тонкой настройке по классификации, когда мы лишь вычисляем процент правильных меток для спама и не спама, чтобы получить точность классификации. На практике для оценки LLM с тонкой настройкой, таких как чат-боты, используются несколько способов:

- тесты с краткими ответами и несколькими вариантами выбора, такие как Measuring Massive Multitask Language Understanding (MMLU; <https://arxiv.org/abs/2009.03300>), которые проверяют общие знания модели;
- сравнение предпочтений человека с ответами других LLM, таких как чат-бот LMSYS (<https://arena.lmsys.org>);
- автоматизированные диалоговые тесты, в которых для оценки ответов, например AlpacaEval (https://tatsu-lab.github.io/alpaca_eval/), используется другая LLM, например GPT-4.

На практике может быть полезно рассмотреть все три типа методов оценки: ответы на вопросы с несколькими вариантами ответов, оценку человеком и автоматизированные показатели, измеряющие эффективность диалога. Но мы в первую очередь заинтересованы в оценке эффективности ведения беседы моделью, а не просто в ее умении отвечать на вопросы с несколькими вариантами ответов, поэтому более актуальными могут быть оценка человеком и автоматизированные показатели.

Эффективность ведения беседы

Коммуникабельность LLM означает их способность вступать в общение, подобное человеческому, понимая контекст, нюансы и намерения. В нее входят такие навыки, как предоставление уместных и последовательных ответов, поддержание последовательности и адаптация к различным темам и стилям взаимодействия.

Оценка, проводимая человеком, хотя и дает ценную информацию, может быть относительно трудоемкой и отнимать много времени, особенно при работе с большим количеством ответов. Например, чтение всех 1100 ответов и выставление им оценок потребовало бы значительных усилий.

Итак, учитывая масштаб поставленной задачи, мы применим подход, аналогичный автоматизированному тестированию общения, который предполагает автоматическую оценку ответов с помощью другого LLM. Этот метод позволит эффективно оценивать качество сгенерированных ответов, не привлекая активно персонал, тем самым экономя время и ресурсы, и при этом получать значимые показатели эффективности.

Далее мы применим подход, вдохновленный AlpacaEval, и используем другую LLM, чтобы оценить отклики нашей отлаженной модели. Однако вместо того чтобы взять общедоступный набор контрольных данных, мы применим наш собственный набор пользовательских тестов. Так мы сможем выполнять более целенаправленную и актуальную оценку производительности модели в контексте предполагаемых вариантов использования, представленных в нашем наборе инструкций.

Чтобы подготовить ответы для этой оценки, мы добавляем сгенерированные ответы модели в словарь `test_set` и сохраняем обновленные данные в виде файла `instruction-data-with-response.json`. Кроме того, сохранив его, мы можем позднее легко загрузить и проанализировать ответы, если это потребуется.

В следующем фрагменте кода (листинг 7.9) метод `generate` используется так же, как и раньше; однако теперь мы перебираем весь набор `test_set`. Кроме того, вместо вывода ответов модели мы добавляем их в словарь `test_set`.

Листинг 7.9. Генерация ответов на тестовые вопросы

```
from tqdm import tqdm

for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)
```

```

token_ids = generate(
    model=model,
    idx=text_to_token_ids(input_text, tokenizer).to(device),
    max_new_tokens=256,
    context_size=BASE_CONFIG["context_length"],
    eos_id=50256
)
generated_text = token_ids_to_text(token_ids, tokenizer)

response_text = (
    generated_text[len(input_text):]
    .replace("### Response:", "")
    .strip()
)
test_data[i]["model_response"] = response_text

with open("instruction-data-with-response.json", "w") as file:
    json.dump(test_data, file, indent=4)

```

Отступ для красивого вида вывода

Обработка набора данных занимает около минуты на графическом процессоре A100 и шесть минут на M3 MacBook Air:

```
100%|██████████| 110/110 [01:05<00:00, 1.68it/s]
```

Убедимся, что ответы были правильно добавлены в словарь `test_set`, проверив одну из записей:

```
print(test_data[0])
```

Вывод показывает, что `model_response` были добавлены правильно:

```
{'instruction': 'Rewrite the sentence using a simile.',
 'input': 'The car is very fast.',
 'output': 'The car is as fast as lightning.',
 'model_response': 'The car is as fast as a bullet.'}
```

Наконец, мы сохраняем модель в файле `gpt2-medium355M-sft.pth`, чтобы использовать ее в будущих проектах:

```
import re

file_name = f"{re.sub(r' [()]", '', CHOOSE_MODEL)}-sft.pth"
torch.save(model.state_dict(), file_name)
print(f"Model saved as {file_name}")
```

Удаляет пропуски и скобки в имени файла

Затем сохраненную модель можно загрузить с помощью `model.load_state_dict(torch.load("gpt2-medium355M-sft.pth"))`.

7.8. Оценка точно настроенной LLM

Ранее мы оценивали производительность модели с тонкой настройкой по инструкциям, анализируя ее ответы на три примера из тестовой выборки. Хотя это дает общее представление о том, насколько хорошо работает модель, данный метод плохо масштабируется для больших объемов ответов. Поэтому мы реализуем метод автоматизации оценки ответов модели с помощью другой, более крупной LLM (рис. 7.19).



Рис. 7.19. Трехэтапный процесс тонкой настройки LLM. На этом последнем шаге мы реализуем метод количественной оценки производительности модели с тонкой настройкой, оценивая сгенерированные ею ответы на тест

Для автоматической оценки ответов на тестовые задания мы используем существующую модель Llama 3 с 8 млрд параметров, разработанную Meta AI. Ее можно запустить локально с помощью приложения Ollama с открытым исходным кодом (<https://ollama.com>).

ПРИМЕЧАНИЕ Ollama — эффективное приложение для запуска LLM на ноутбуке. Оно служит оболочкой для библиотеки llama.cpp с открытым исходным кодом (<https://github.com/ggerganov/llama.cpp>), которая реализует LLM на чистом C/C++ в целях максимальной эффективности. Однако Ollama является всего лишь инструментом для генерации текста с помощью LLM (inference) и не поддерживает обучение или тонкую настройку LLM.

Использование более крупных LLM через веб-API

Модель Llama 3 с 8 млрд параметров — очень мощная LLM, которая работает локально. Однако она не так эффективна, как крупные закрытые LLM, такие как GPT-4, предлагаемые OpenAI. Читатели, интересующиеся тем, как использовать GPT-4 через API OpenAI для оценки сгенерированных ответов модели, в дополнительных материалах к этой книге (<https://mng.bz/BgEv>) могут найти код.

Чтобы выполнить код, приведенный ниже, установите Ollama, перейдя на сайт <https://ollama.com>, и следуйте инструкциям для вашей операционной системы:

- для пользователей *macOS* и *Windows* — откройте загруженное приложение Ollama. Если появится запрос на установку с использованием командной строки, то выберите **Да**;
- для пользователей *Linux* — используйте команду установки, доступную на сайте Ollama.

Прежде чем реализовывать код для оценки модели, сначала загрузим модель Llama 3 и убедимся, что приложение Ollama работает правильно, запустив его из командной строки. Чтобы использовать Ollama из командной строки, необходимо либо запустить приложение, либо ввести команду `ollama serve` в отдельном терминальном окне (рис. 7.20).

Запустив приложение Ollama одним из этих способов, выполните следующую команду в командной строке (не в сеансе Python), чтобы опробовать модель Llama 3 с 8 млрд параметров:

```
ollama run llama3
```

При первом выполнении этой команды модель, занимающая 4,7 Гбайт памяти, будет скачана автоматически. Вывод выглядит так:

```
pulling manifest
pulling 6a0746a1ec1a... 100% |████████████████████| 4.7 GB
pulling 4fa551d4f938... 100% |████████████████████| 12 KB
pulling 8ab4849b038c... 100% |████████████████████| 254 B
pulling 577073ffcc6c... 100% |████████████████████| 110 B
pulling 3f8eb4da87fa... 100% |████████████████████| 485 B
verifying sha256 digest
writing manifest
removing any unused layers
success
```


Учтите, что ответ, который вы получите, может быть другим, так как на момент написания книги ответы Ollama не являются детерминированными.

Вы можете завершить сеанс `ollama run llama3`, введя `/bye`. Однако не забудьте оставить команду `ollama serve` активной или приложение Ollama запущенным до конца этой главы.

Запуск кода в новом сеансе Python

Если вы уже закрыли сеанс Python или предпочитаете выполнить оставшийся код в другом сеансе, то используйте код ниже, который загружает файл инструкций и ответов, созданный нами ранее, и переопределяет функцию `format_input`, которую мы использовали ранее (`tqdm` для отображения хода выполнения программы используется позже):

```
import json
from tqdm import tqdm

file_path = "instruction-data-with-response.json"
with open(file_path, "r") as file:
    test_data = json.load(file)

def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = (
        f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""
    )
    return instruction_text + input_text
```

С помощью следующего кода можно проверить, что сеанс Ollama работает правильно, прежде чем использовать это приложение для оценки ответов из тестовой выборки:

```
import psutil

def check_if_running(process_name):
    running = False
    for proc in psutil.process_iter(["name"]):
        if process_name in proc.info["name"]:
            running = True
            break
    return running
```

```
ollama_running = check_if_running("ollama")

if not ollama_running:
    raise RuntimeError(
        "Ollama not running. Launch ollama before proceeding."
    )
print("Ollama running:", check_if_running("ollama"))
```

Убедитесь, что в результате выполнения предыдущего кода отображается `Ollama running: True`. Если вы видите `False`, то убедитесь, что команда `ollama serve` или приложение Ollama активно работают.

Альтернативой команде `ollama run` для взаимодействия с моделью является использование REST API с помощью Python. Функция `query_model` демонстрирует применение API (листинг 7.10).

Листинг 7.10. Запрос к локальной модели Ollama

```
import urllib.request

def query_model(
    prompt,
    model="llama3",
    url="http://localhost:11434/api/chat"
):
    data = {
        "model": model,
        "messages": [
            {"role": "user", "content": prompt}
        ],
        "options": {
            "seed": 123,
            "temperature": 0,
            "num_ctx": 2048
        }
    }

    payload = json.dumps(data).encode("utf-8")
    request = urllib.request.Request(
        url,
        data=payload,
        method="POST"
    )

    request.add_header("Content-Type", "application/json")

    response_data = ""
    with urllib.request.urlopen(request) as response:
```

Создает данные для загрузки в виде словаря

Установки для детерминированных ответов

Преобразует словарь в строку в формате JSON и кодирует ее в байты

Создает объект запроса, устанавливая метод в POST и добавляя необходимые заголовки

Отправляет запрос и фиксирует ответ

```

while True:
    line = response.readline().decode("utf-8")
    if not line:
        break
    response_json = json.loads(line)
    response_data += response_json["message"]["content"]

return response_data

```

Прежде чем запускать последующие строки кода, убедитесь, что Оллама по-прежнему работает. Предыдущие строки должны выводить сообщение "Ollama running: True" — это подтверждает, что модель активна и готова принимать запросы.

Ниже приведен пример использования функции `query_model`, которую мы только что реализовали:

```

model = "llama3"
result = query_model("What do Llamas eat?", model)
print(result)

```

Получаем следующий ответ:

Llamas are ruminant animals, which means they have a four-chambered stomach that allows them to digest plant-based foods. Their diet typically consists of:

1. Grasses: Llamas love to graze on grasses, including tall grasses, short grasses, and even weeds.
- ...

Используя функцию `query_model`, определенную ранее, мы можем оценить ответы, сгенерированные нашей настроенной моделью, которая побуждает модель Ллама 3 оценивать ответы нашей доработанной модели по шкале от 0 до 100 баллов, взяв в качестве эталона заданный тестовый набор ответов.

Сначала мы применим этот подход к первым трем примерам из тестового набора, которые рассматривали ранее:

```

for entry in test_data[:3]:
    prompt = (
        f"Given the input `{format_input(entry)}` "
        f"and correct output `{entry['output']}`, "
        f"score the model response `{entry['model_response']}` "
        f"on a scale from 0 to 100, where 100 is the best score. "
    )
    print("\nDataset response:")
    print(">>", entry['output'])

```



```
print("\nModel response:")
print(">>", entry["model_response"])
print("\nScore:")
print(">>", query_model(prompt))
print("\n-----")
```

Этот код выводит результаты, похожие на следующие (на момент написания данной книги ответы Ollama не являются полностью детерминированными, поэтому в вашем случае сгенерированные тексты могут быть другими):

Dataset response:

```
>> The car is as fast as lightning.
```

Model response:

```
>> The car is as fast as a bullet.
```

Score:

```
>> I'd rate the model response "The car is as fast as a bullet."
an 85 out of 100.
```

Here's why:

The response uses a simile correctly, comparing the speed of the car to something else (in this case, a bullet).

The comparison is relevant and makes sense, as bullets are known for their high velocity.

The phrase "as fast as" is used correctly to introduce the simile.

The only reason I wouldn't give it a perfect score is that some people might find the comparison slightly less vivid or evocative than others. For example, comparing something to lightning (as in the original response) can be more dramatic and attention grabbing. However, "as fast as a bullet" is still a strong and effective simile that effectively conveys the idea of the car's speed.

Overall, I think the model did a great job!

Dataset response:

```
>> The type of cloud typically associated with thunderstorms is
cumulonimbus.
```

286 Глава 7. Тонкая настройка по инструкциям

Model response:

>> The type of cloud associated with thunderstorms is a cumulus cloud.

Score:

>> I'd score this model response as 40 out of 100.

Here's why:

The model correctly identifies that thunderstorms are related to clouds (correctly identifying the type of phenomenon).

However, it incorrectly specifies the type of cloud associated with thunderstorms. Cumulus clouds are not typically associated with thunderstorms; cumulonimbus clouds are.

The response lacks precision and accuracy in its description.

Overall, while the model attempts to address the instruction, it provides an incorrect answer, which is a significant error.

Dataset response:

>> Jane Austen.

Model response:

>> The author of 'Pride and Prejudice' is Jane Austen.

Score:

>> I'd rate my own response as 95 out of 100.

Here's why:

The response accurately answers the question by naming the author of *Pride and Prejudice* as Jane Austen.

The response is concise and clear, making it easy to understand.

There are no grammatical errors or ambiguities that could lead to confusion. The only reason I wouldn't give myself a perfect score is that the response is slightly redundant—it's not necessary to rephrase the question in the answer. A more concise response would be simply "Jane Austen."

Сгенерированные ответы показывают, что модель Ллама 3 дает обоснованные оценки и способна выставять неполные баллы, если ответ модели не совсем

правильный. Например, если мы рассмотрим оценку ответа *cumulus cloud*, модель признает частичную правильность ответа.

В результате предыдущей подсказки возвращаются очень подробные оценки в дополнение к баллам. Мы можем изменить подсказку так, чтобы она генерировала только целочисленные оценки в диапазоне от 0 до 100 баллов, где 100 — максимально возможная оценка. Это изменение позволяет вычислить среднюю оценку для модели, которая служит более краткой и количественной оценкой ее работы. Функция `generate_model_scores` использует измененную подсказку, в которой модели говорится: "Respond with the integer number only." (листинг 7.11).

Листинг 7.11. Оценка LLM с тонкой настройкой по инструкциям

```
def generate_model_scores(json_data, json_key, model="llama3"):
    scores = []
    for entry in tqdm(json_data, desc="Scoring entries"):
        prompt = (
            f"Given the input `{format_input(entry)}` "
            f"and correct output `{entry['output']}`, "
            f"score the model response `{entry[json_key]}` "
            f"on a scale from 0 to 100, where 100 is the best score. "
            f"Respond with the integer number only."
        )
        score = query_model(prompt, model)
        try:
            scores.append(int(score))
        except ValueError:
            print(f"Could not convert score: {score}")
            continue

    return scores
```

Изменение кода, чтобы он возвращал только целочисленный результат

Теперь применим функцию `generate_model_scores` ко всему набору `test_data`, что займет около минуты на MacBook Air с процессором M3:

```
scores = generate_model_scores(test_data, "model_response")
print(f"Number of scores: {len(scores)} of {len(test_data)}")
print(f"Average score: {sum(scores)/len(scores):.2f}\n")
```

В результате получаем:

```
Scoring entries: 100%|████████████████████████████████████████| 110/110
[01:10<00:00, 1.56it/s]
Number of scores: 110 of 110
Average score: 50.32
```

Результаты оценки показывают, что наша доработанная модель набирает в среднем более 50 баллов, что является полезным ориентиром для сравнения

с другими моделями или для экспериментов с различными конфигурациями обучения, позволяющих улучшить производительность модели.

Стоит отметить, что на момент написания этих строк Оллма не была полностью детерминирована в разных операционных системах. Это значит, полученные вами оценки могут незначительно отличаться от тех, которые приведены выше. Чтобы получить более достоверные результаты, вы можете повторить оценку несколько раз и усреднить полученные баллы.

Чтобы еще больше повысить эффективность нашей модели, мы можем использовать различные стратегии:

- настроить гиперпараметры во время тонкой настройки, например скорость обучения, размер пакета или количество эпох;
- увеличить размер обучающей выборки или разнообразия примеров, чтобы охватить более широкий спектр тем и стилей;
- экспериментировать с различными подсказками или форматами инструкций, чтобы более эффективно управлять ответами модели;
- использовать более крупную предварительно обученную модель, которая может лучше улавливать сложные закономерности и генерировать более точные ответы.

ПРИМЕЧАНИЕ Для справки: при использовании описанной здесь методологии базовая модель Llama 3 8B без какой-либо тонкой настройки набирает в среднем 58,51 балла на тестовой выборке. Модель Llama 3 8B, точно настроенная для выполнения общих инструкций, набирает в среднем 82,6 балла.

Упражнение 7.4. Эффективная тонкая настройка параметров с помощью LoRA

Для более эффективного обучения и настройки LLM измените код в этой главе, чтобы использовать метод низкоранговой адаптации (low-rank adaptation, LoRA) из приложения Д. Сравните время обучения и производительность модели до и после изменения.

7.9. Выводы

Наше путешествие по циклу разработки LLM завершается. Мы рассмотрели все основные этапы: внедрение архитектуры модели, предварительное обучение LLM и тонкую настройку, благодаря которой модель сможет выполнять конкретные задачи (рис. 7.21).

Теперь обсудим, на что следует обратить внимание в дальнейшем.

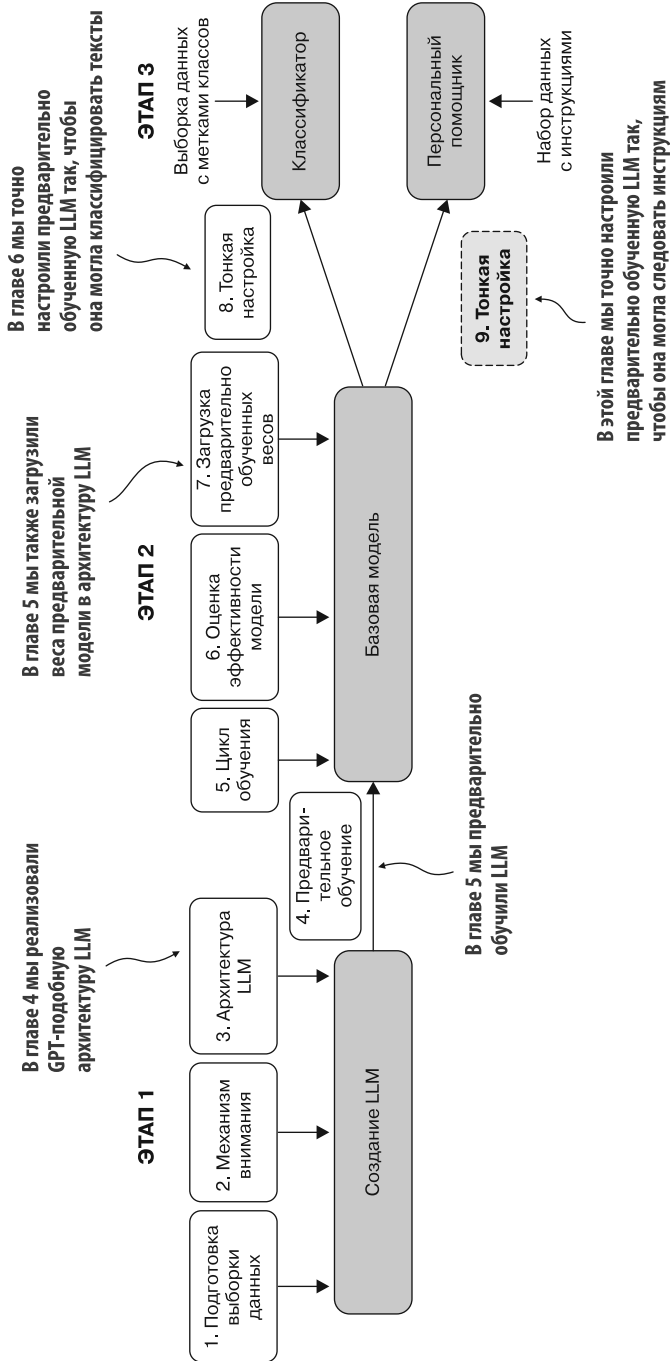


Рис. 7.21. Три основных этапа написания кода для LLM

7.9.1. Что дальше

Мы рассмотрели самые важные шаги, но есть еще один (необязательный), который можно выполнить после тонкой настройки по инструкциям: тонкая настройка под предпочтения. Она позволяет адаптировать модель к конкретным предпочтениям пользователя. Если вы хотите узнать больше, то ознакомьтесь с папкой 04_preference-tuning-with-dpo в GitHub-репозитории этой книги по адресу <https://mng.bz/dZwD>.

Вдобавок к основному контенту, описанному в данной книге, репозиторий GitHub содержит большой выбор дополнительных материалов, которые могут оказаться для вас полезными. Чтобы узнать о них больше информации, посетите раздел Bonus Material на странице README репозитория: <https://mng.bz/r12g>.

7.9.2. Будьте в курсе последних событий в быстро меняющейся области

Исследования в области искусственного интеллекта и больших языковых моделей развиваются стремительными темпами. Один из способов быть в курсе последних достижений — изучать свежие научные статьи на сайте arXiv по адресу <https://arxiv.org/list/cs.LG/recent>. Кроме того, многие исследователи и специалисты-практики активно обсуждают последние разработки на платформах социальных сетей, таких как X (ранее Twitter) и Reddit. В частности, subreddit r/LocalLLaMA — хороший ресурс для общения с сообществом и получения информации о новейших инструментах и тенденциях. Кроме того, и я регулярно делюсь своими мыслями и пишу о последних исследованиях в области LLM в своем блоге, доступном на <https://magazine.sebastianraschka.com> и <https://sebastianraschka.com/blog/>.

7.9.3. Заключительные слова

Я надеюсь, что вам понравилось создавать большие языковые модели и программировать функции предварительного обучения и тонкой настройки. На мой взгляд, создание LLM с нуля — самый эффективный способ получения глубокого понимания того, как работают модели. Я надеюсь, что этот практический подход дал вам ценные знания и прочную основу для разработки LLM.

Основная цель данной книги — образовательная, но вам может быть интересно использовать другие, более мощные LLM для реальных задач. Для этого я рекомендую изучить такие популярные инструменты, как Axolotl (<https://github.com/OpenAccess-AI-Collective/axolotl>) или LitGPT (<https://github.com/Lightning-AI/litgpt>), в разработке которых я активно участвую.

Спасибо, что присоединились ко мне в этом учебном путешествии, и я желаю вам всего наилучшего в ваших будущих начинаниях в захватывающей области больших языковых моделей и искусственного интеллекта!

Итоги главы

- Процесс тонкой настройки по инструкциям адаптирует предварительно обученную большую языковую модель так, чтобы она могла следовать инструкциям человека и генерировать желаемые ответы.
- В подготовку набора данных входит загрузка набора данных «инструкция — ответ», форматирование записей и разделение их на обучающую, проверочную и тестовую выборки.
- Обучающие пакеты создаются с помощью пользовательской функции *collate*, которая дополняет последовательности, создает целевые идентификаторы токенов и маскирует заполняющие токены.
- Предварительно обученная средняя модель GPT-2 с 355 млн параметров загружается в качестве отправной точки в процессе тонкой настройки по инструкциям.
- Предварительно обученная модель проходит тонкую настройку на наборе инструкций с помощью цикла обучения, аналогичного предварительному обучению.
- Оценка подразумевает извлечение ответов модели на примеры из тестовой выборки и их оценку (например, с помощью другой LLM).
- Приложение Ollama с моделью Llama на 8 млрд параметров можно использовать для автоматической оценки ответов точно настроенной модели с помощью примеров из тестовой выборки, получая среднюю величину оценки производительности.

Приложение А

Введение в PyTorch

Это приложение поможет вам получить навыки и знания, необходимые для применения глубокого обучения на практике и реализации больших языковых моделей с нуля. PyTorch, популярная библиотека глубокого обучения на основе Python, — наш основной инструмент в этой книге. Я расскажу о создании рабочего пространства для глубокого обучения, оснащенного PyTorch и с поддержкой графического процессора.

Затем вы узнаете об основной концепции тензоров и их использовании в PyTorch. Мы также рассмотрим механизм автоматического дифференцирования в PyTorch — функцию, которая позволяет удобно и эффективно использовать обратное распространение ошибки, что является важнейшим аспектом обучения нейронных сетей.

Это приложение предназначено для тех, кто только начинает изучать глубокое обучение в PyTorch. Здесь дано обширное описание этой библиотеки, но оно не претендует на исчерпывающее. Вместо этого мы сосредоточимся на основах PyTorch, которые будем использовать для реализации LLM. Если вы уже знакомы с глубоким обучением, то можете пропустить это приложение и сразу перейти к главе 2.

А.1. Что такое PyTorch

PyTorch (<https://pytorch.org/>) — библиотека для глубокого обучения с открытым исходным кодом на основе Python. Согласно платформе *Papers With Code* (<https://paperswithcode.com/trends>), которая отслеживает и анализирует исследовательские работы, с 2019 года PyTorch является наиболее популярной библиотекой глубокого обучения, используемой для научных исследований. Согласно опросу *Kaggle Data Science and Machine Learning Survey 2022* (<https://www.kaggle.com/c/>

kaggle-survey-2022), количество респондентов, использующих PyTorch, составляет примерно 40 %, и с каждым годом этот показатель растет.

Одна из причин популярности PyTorch — удобный интерфейс и эффективность. Несмотря на доступность, библиотека не теряет в гибкости, позволяя опытным пользователям настраивать низкоуровневые аспекты их моделей для адаптации к решаемым задачам и оптимизации моделей. Короче говоря, для многих практиков и исследователей PyTorch предлагает идеальный баланс между удобством использования и функциональностью.

A.1.1. Три основных компонента PyTorch

PyTorch — обширная библиотека, и один из способов разобраться в ней — сосредоточиться на трех ее основных компонентах (рис. A.1).

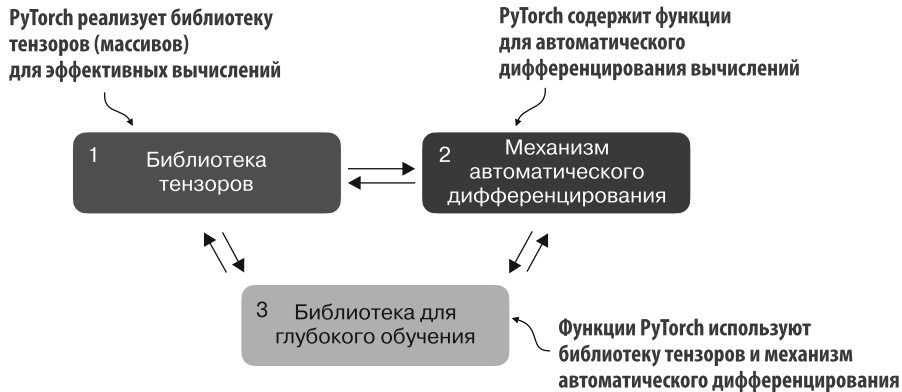


Рис. A.1. Три основных компонента библиотеки PyTorch

Рассмотрим эти компоненты более подробно.

Библиотека тензоров — фундаментальный строительный блок для вычислений. Она расширяет концепцию библиотеки программирования NumPy, ориентированной на массивы, добавляя функцию, ускоряющую вычисления на GPU, что обеспечивает плавный переход между обычными и графическими процессорами.

Механизм автоматического дифференцирования, также известный как автоградиент (autograd), позволяет автоматически вычислять градиенты для операций с тензорами, упрощая вычисление обратного распространения ошибки и оптимизацию модели.

Библиотека для глубокого обучения предлагает модульные, гибкие и эффективные строительные блоки, в том числе предварительно обученные модели, готовые

функции потерь и оптимизаторы. Все это позволяет создавать и обучать широкий спектр моделей, ориентированных как на исследователей, так и на разработчиков.

В совокупности эти компоненты упрощают реализацию и обучение моделей глубоких нейронных сетей.

А.1.2. Определение глубокого обучения

В новостях большие языковые модели часто называют моделями искусственного интеллекта (ИИ). Однако LLM при этом — еще и разновидность глубоких нейронных сетей, а PyTorch — библиотека для глубокого обучения. Звучит запутанно? Давайте коротко рассмотрим взаимосвязь этих терминов, прежде чем двигаться дальше.

По сути, *ИИ* занимается созданием компьютерных систем, способных выполнять задачи, которые обычно требуют человеческого интеллекта. Эти задачи включают в себя понимание естественного языка, распознавание визуальных образов и принятие решений. (Несмотря на большой прогресс, ИИ все еще далек от достижения такого уровня общего интеллекта.)

Машинное обучение (МО) представляет собой область ИИ (рис. А.2), которая занимается разработкой и совершенствованием алгоритмов обучения. Ключевая идея заключается в том, чтобы позволить компьютерам обучаться на основе данных и делать прогнозы или принимать решения, не будучи явно запрограммированными на выполнение такой задачи. Это предполагает разработку алгоритмов, которые могут выявлять закономерности, обучаться на основе исторических данных и со временем улучшать свою производительность за счет большего количества данных и обратной связи.

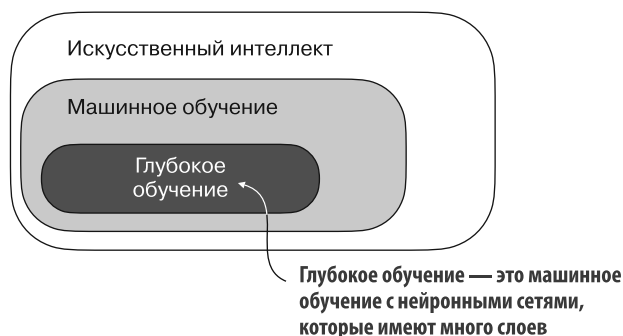


Рис. А.2. Глубокое обучение — подкатегория машинного, ориентированная на реализацию глубоких нейронных сетей. Машинное обучение — подкатегория ИИ, которая занимается алгоритмами, обучающимися на данных. ИИ — более широкое понятие, обозначающее способность машин выполнять задачи, которые обычно требуют человеческого интеллекта

Машинное обучение было неотъемлемой частью эволюции искусственного интеллекта, способствующей возникновению многих современных достижений, в том числе больших языковых моделей. Кроме того, МО лежит в основе таких технологий, как рекомендательные системы, используемые онлайн-магазинами и потоковыми сервисами, фильтрация спама, рассылаемого по электронной почте, распознавание голоса в виртуальных помощниках и даже самоуправляемые автомобили. Внедрение и развитие машинного обучения значительно улучшило возможности искусственного интеллекта, позволяющие ему выходить за рамки систем, основанных на строгих правилах, и адаптироваться к новым входным данным или изменяющейся среде.

Глубокое обучение — подкатегория машинного обучения, которая фокусируется на обучении и применении глубоких нейронных сетей. Первоначально идея создания таких сетей была основана на работе человеческого мозга, в частности на взаимосвязях между множеством нейронов. Слово «глубокий» в данном контексте относится к нескольким скрытым слоям искусственных нейронов или узлов, которые позволяют моделировать сложные нелинейные взаимосвязи в данных. В отличие от традиционных методов машинного обучения, которые хорошо справляются с простым распознаванием образов, глубокое обучение особенно полезно для работы с неструктурированными данными, такими как изображения, звук или текст, поэтому его с большой эффективностью можно использовать для работы с большими языковыми моделями.

Типичный процесс прогностического моделирования (также называемый *обучением с учителем*) в машинном и глубоком обучении представлен на рис. А.3.

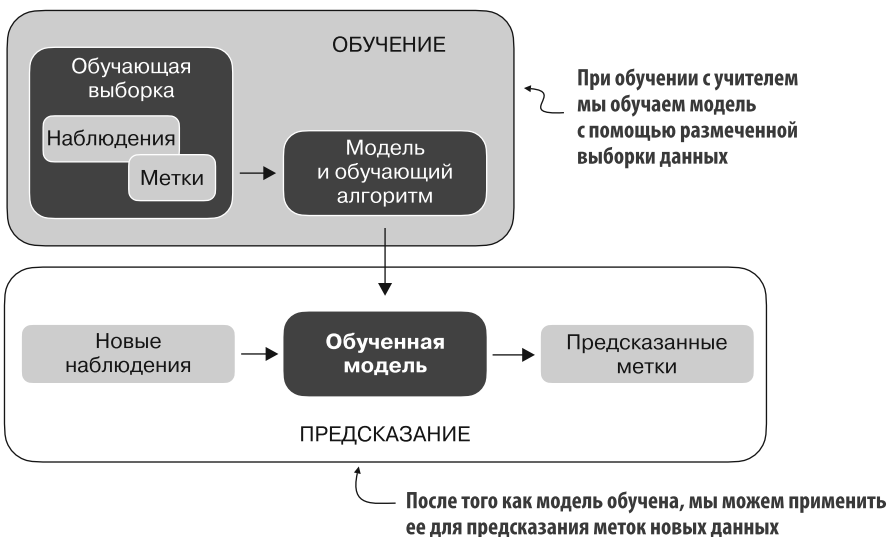


Рис. А.3. Процесс обучения с учителем, используемый в прогностном моделировании

С помощью алгоритма обучения модель обучается на наборе обучающих данных, состоящем из примеров и соответствующих им меток. Например, в случае с классификатором спама из электронной почты обучающий набор данных состоит из электронных писем и их меток «спам» и «не спам», которые присвоил человек. Затем обученную модель можно использовать для новых наблюдений (то есть новых электронных писем), чтобы предсказать их метку («спам» или «не спам»). Конечно, между этапами обучения и предсказания мы хотим добавить этап оценки модели, чтобы убедиться, что модель соответствует нашим критериям эффективности, прежде чем использовать ее в реальном приложении.

Если мы обучаем LLM классификации текстов, то процесс обучения и использования модели аналогичен тому, что показан на рис. А.3. Если мы хотим научить LLM генерировать тексты, что является нашей основной задачей, то тоже можем опираться на этот процесс. В таком случае метки во время предварительного обучения могут быть получены из самого текста (задача предсказания следующего слова, описанная в главе 1). LLM будет генерировать совершенно новый текст (а не предсказывать метки), когда мы будем вводить подсказку на этапе предсказания.

А.1.3. Установка PyTorch

PyTorch можно установить так же, как и любую другую библиотеку или пакет Python. Но это комплексная библиотека, содержащая код, совместимый с CPU и GPU, поэтому установка может потребовать дополнительных пояснений.

Версия Python

Многие библиотеки для научных вычислений не поддерживают последнюю версию Python. Поэтому при установке PyTorch рекомендуется использовать версию Python на один или два релиза младше. Например, если последняя версия Python — 3.13, то рекомендуется использовать Python 3.11 или 3.12.

Например, есть две версии PyTorch: облегченная (поддерживает только вычисления на обыкновенном процессоре) и полная (поддерживает вычисления как на обыкновенном, так и на графическом процессоре). Если на вашем компьютере есть GPU, совместимый с CUDA, который можно использовать для глубокого обучения (в идеале — NVIDIA T4, RTX 2080 Ti или новее), то я рекомендую установить версию с графическим процессором. Команда по умолчанию для установки PyTorch в терминале выглядит так:

```
pip install torch
```

Предположим, ваш компьютер поддерживает графический процессор, совместимый с CUDA. В этом случае будет автоматически установлена версия PyTorch, поддерживающая ускорение GPU через CUDA, при условии, что в среде Python, с которой вы работаете, установлены необходимые зависимости (например, `pip`).

ПРИМЕЧАНИЕ На момент написания этой книги в PyTorch также добавлена экспериментальная поддержка графических процессоров AMD через ROCm. Дополнительные инструкции можно найти на <https://pytorch.org>.

Чтобы явно установить версию PyTorch, совместимую с CUDA, часто лучше указать, с какой версией CUDA она должна быть совместима. На официальном сайте PyTorch (<https://pytorch.org>) приведены команды для установки PyTorch с поддержкой CUDA для различных операционных систем. На рис. A.4 показана команда для установки PyTorch, а также библиотек `torchvision` и `torchaudio`, необязательных для работы с материалами этой книги.

Выберите новейшую стабильную версию

PyTorch Build	Stable (2.0.1)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 11.7	CUDA 11.8	ROCm 5.4.2	CPU
Run this Command:	<code>pip3 install torch torchvision torchaudio</code>			

Выберите версию CUDA, совместимую с вашей графической картой

Если у вас нет графической карты от Nvidia, поддерживающей CUDA, то выберите версию CPU

Рис. A.4. Меню установки PyTorch

Для примеров я использую PyTorch 2.4.0, поэтому рекомендую вам выполнять следующую команду для установки точной версии, чтобы гарантировать совместимость с примерами, приведенными в этой книге:

```
pip install torch==2.4.0
```

Однако, как упоминалось ранее, в зависимости от вашей операционной системы команда установки может немного отличаться от приведенной здесь. Поэтому

я рекомендую вам посетить сайт <https://pytorch.org> и воспользоваться меню установки (см. рис. А.4), чтобы выбрать команду установки для вашей операционной системы. Не забудьте заменить `torch` на `torch==2.4.0` в команде.

Чтобы проверить версию PyTorch, выполните следующий код:

```
import torch
torch.__version__
```

Вывод будет выглядеть так:

```
'2.4.0'
```

PyTorch и Torch

Библиотека называется PyTorch в первую очередь потому, что является продолжением библиотеки Torch, но адаптирована для Python (отсюда и PyTorch). Torch в названии библиотеки отсылает к Torch — фреймворку для научных вычислений, обладающему широкой поддержкой алгоритмов машинного обучения, который изначально был создан с использованием языка программирования Lua.

Если вы ищете дополнительные рекомендации и инструкции по настройке среды Python или установке других библиотек, используемых в этой книге, то посетите репозиторий GitHub по адресу <https://github.com/rasbt/LLMs-from-scratch>.

После установки PyTorch вы можете проверить, распознает ли ваша установка встроенный графический процессор NVIDIA, запустив следующий код на Python:

```
import torch
torch.cuda.is_available()
```

Если команда возвращает `True`, значит, все готово. Если `False`, то, возможно, на вашем компьютере нет совместимого графического процессора или PyTorch его не распознает. Для работы с материалом первых глав этой книги, посвященных внедрению LLM в образовательных целях, GPU не требуются, но они могут значительно ускорить вычисления, связанные с глубоким обучением.

Если у вас нет доступа к графическому процессору, то можете прибегнуть к услугам провайдеров облачных вычислений, у которых пользователи могут запускать вычисления на GPU за почасовую плату. Популярной средой, похожей на блокнот Jupyter, является Google Colab (<https://colab.research.google.com>), которая на момент написания книги предоставляет ограниченный по времени доступ к графическим процессорам. Используя меню Runtime (Среда выполнения), можно выбрать GPU (рис. А.5).

Чтобы открыть это меню, выберите команду **Change runtime type** (Сменить среду выполнения) на вкладке **Runtime** (Среда выполнения)

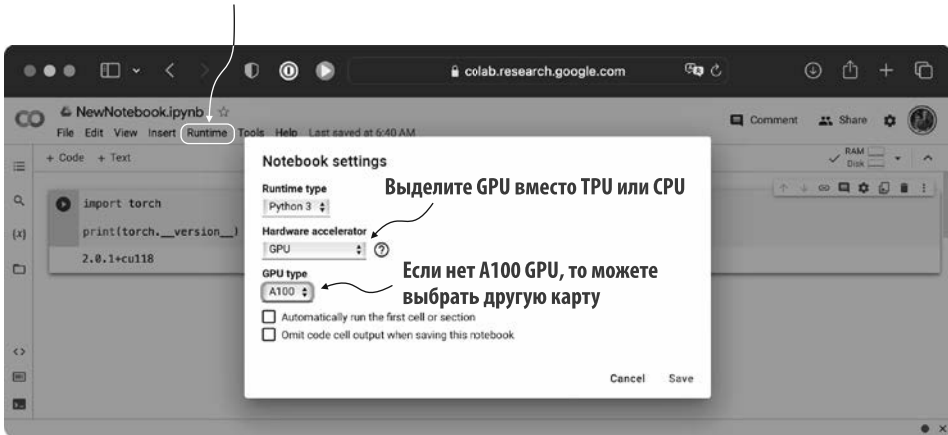


Рис. А.5. Меню выбора графического процессора

PyTorch на Apple Silicon

Если у вас есть Apple Mac с чипом Apple Silicon (например, M1, M2, M3 или более новые модели), то вы можете использовать его возможности для ускорения выполнения кода PyTorch. Чтобы использовать данный чип, сначала необходимо установить PyTorch обычным способом. Затем, чтобы проверить, поддерживает ли ваш Mac ускорение PyTorch с помощью этого чипа, вы можете запустить простой фрагмент кода на Python:

```
print(torch.backends.mps.is_available())
```

Если возвращается True, это означает, что чип Apple Silicon установлен на вашем Mac.

Упражнение А.1. Установка и настройка PyTorch

Установите и настройте PyTorch на своем компьютере.

Упражнение А.2. Запуск кода

Запустите код с <https://mng.bz/o05v>, который проверяет правильность настройки вашей среды.

А.2. Тензоры

Тензоры — математическая концепция, которая обобщает векторы и матрицы до потенциально более высоких размерностей. Другими словами, тензоры — это математические объекты, которые можно охарактеризовать их порядком (или рангом), определяющим количество размерностей. Например, скаляр (просто число) — тензор ранга 0, вектор — тензор ранга 1, а матрица — тензор ранга 2 (рис. А.6).

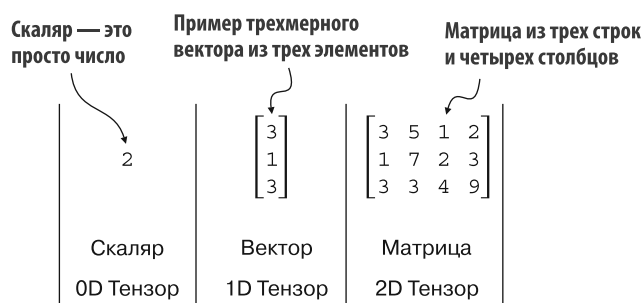


Рис. А.6. Тензоры разных рангов. 0D соответствует рангу 0, 1D — рангу 1, а 2D — рангу 2. Трехмерный вектор, состоящий из трех элементов, по-прежнему является тензором ранга 1

С точки зрения вычислений тензоры служат контейнерами для данных. Например, содержат многомерные данные, где каждая размерность представляет собой отдельный признак. Библиотеки тензоров, такие как PyTorch, могут эффективно создавать эти массивы, управлять ими и использовать их для выполнения вычислений. В этом контексте библиотека тензоров функционирует как библиотека массивов.

Тензоры PyTorch похожи на массивы NumPy, но имеют несколько дополнительных функций, важных для глубокого обучения. Например, PyTorch добавляет механизм автоматического дифференцирования, упрощающий *вычисление градиентов* (см. раздел А.4). Кроме того, тензоры PyTorch поддерживают вычисления на графическом процессоре, что позволяет ускорить обучение глубоких нейронных сетей (см. раздел А.8).

PyTorch с API, похожим на NumPy

PyTorch использует большую часть API и синтаксиса массивов NumPy для своих тензорных операций. Если вы новичок в NumPy, то можете найти краткий обзор наиболее важных концепций в моей статье *Scientific Computing in Python: Introduction to NumPy and Matplotlib* по адресу <https://sebastianraschka.com/blog/2020/numpy-intro.html>.

A.2.1. Скаляры, векторы, матрицы и тензоры

Как упоминалось ранее, тензоры PyTorch — это контейнеры данных для массивоподобных структур. Скаляр — тензор нулевой размерности (например, просто число), вектор — одномерный тензор, а матрица — двумерный тензор. Для тензоров более высокой размерности нет специального термина, поэтому обычно трехмерный тензор называется просто 3D-тензором и т. д. Создавать объекты класса Tensor в PyTorch можно с помощью функции `torch.tensor` (листинг A.1).

Листинг A.1. Создание тензоров PyTorch

```
import torch

tensor0d = torch.tensor(1)  ← Создает 0-мерный тензор (скаляр)
                             из целого числа Python

tensor1d = torch.tensor([1, 2, 3])  ← Создает одномерный тензор
                                     (вектор) из списка Python

tensor2d = torch.tensor([[1, 2],
                        [3, 4]])  ← Создает двумерный тензор
                                   из вложенного списка Python

tensor3d = torch.tensor([[[1, 2], [3, 4]],
                        [[5, 6], [7, 8]]])  ← Создает трехмерный тензор
                                              из вложенного списка Python
```

A.2.2. Типы тензорных данных

PyTorch использует 64-битный целочисленный тип данных по умолчанию из Python. Мы можем получить доступ к типу данных тензора, используя атрибут `.dtype` тензора:

```
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```

Вывод выглядит так:

```
torch.int64
```

Если мы создаем тензоры из чисел с плавающей запятой в Python, то PyTorch по умолчанию создает тензоры с 32-битной точностью:

```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

Вывод следующий:

```
torch.float32
```

Этот выбор обусловлен в первую очередь балансом между точностью и эффективностью вычислений. 32-битное число с плавающей запятой обеспечивает достаточную точность для большинства задач глубокого обучения, потребляя

при этом меньше памяти и вычислительных ресурсов, чем 64-битное число с плавающей запятой. Более того, архитектуры графических процессоров оптимизированы для 32-битных вычислений, и использование этого типа данных может значительно ускорить обучение модели и вывод.

Кроме того, можно изменить точность с помощью метода `.to` тензора. В коде ниже это показано на примере преобразования 64-битного целочисленного тензора в 32-битный тензор с плавающей запятой:

```
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```

Код возвращает следующий вывод:

```
torch.float32
```

Дополнительную информацию о различных типах тензорных данных, доступных в PyTorch, можно найти в официальной документации по адресу <https://pytorch.org/docs/stable/tensors.html>.

А.2.3. Распространенные операции с тензорами в PyTorch

Подробное описание всех операций с тензорами в PyTorch и команд выходит за рамки этой книги. Однако я кратко опишу некоторые операции по мере их представления в книге.

Вы уже знакомы с функцией `torch.tensor()`, позволяющей создавать новые тензоры:

```
tensor2d = torch.tensor([[1, 2, 3],
                          [4, 5, 6]])
print(tensor2d)
```

Вывод следующий:

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

Кроме того, атрибут `.shape` позволяет получить доступ к форме тензора:

```
print(tensor2d.shape)
```

Результат выглядит так:

```
torch.Size([2, 3])
```

Как видите, `.shape` возвращает `[2, 3]`, то есть у тензора две строки и три столбца. Чтобы преобразовать тензор в тензор 3×2 , можно использовать метод `.reshape`:

```
print(tensor2d.reshape(3, 2))
```

Вывод следующий:

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

Однако обратите внимание, что более распространенной командой для изменения формы тензоров в PyTorch является `.view()`:

```
print(tensor2d.view(3, 2))
```

Результат выглядит так:

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

Как и в случае с `.reshape` и `.view`, иногда PyTorch предлагает несколько вариантов синтаксиса для выполнения одних и тех же вычислений. Изначально PyTorch следовала оригинальному синтаксису Lua Torch, но затем, по многочисленным просьбам, синтаксис изменился, чтобы сделать библиотеку похожей на NumPy. (Тонкое различие между `.view()` и `.reshape()` в PyTorch заключается в их обращении со структурой памяти: `.view()` требует, чтобы исходные данные были непрерывными, и завершится ошибкой, если это не так; в свою очередь, `.reshape()` будет работать независимо от такого предположения, копируя данные, если это необходимо, чтобы обеспечить желаемую форму.)

Далее, можно использовать `.T` для транспонирования тензора:

```
print(tensor2d.T)
```

Обратите внимание, что это похоже на изменение формы тензора, как вы можете видеть на примере следующего результата:

```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

Наконец, распространенным средством перемножения двух матриц в PyTorch является метод `.matmul`:

```
print(tensor2d.matmul(tensor2d.T))
```

с результатом:

```
tensor([[14, 32],
        [32, 77]])
```

Однако мы также можем использовать оператор `@`, который выполняет ту же задачу более компактно:

```
print(tensor2d @ tensor2d.T)
```

Результат выглядит так:

```
tensor([[14, 32],
        [32, 77]])
```

Как упоминалось ранее, при необходимости я ввожу дополнительные операции. Читателям, которые хотели бы ознакомиться со всеми операциями с тензорами, доступными в PyTorch (большинство из них не понадобятся для работы с содержанием этой книги), я рекомендую ознакомиться с официальной документацией по адресу <https://pytorch.org/docs/stable/tensors.html>.

А.3. Представление моделей в виде графов вычислений

Теперь рассмотрим механизм автоматического дифференцирования PyTorch, также известный как автоградиент. Система автоградиента в PyTorch предоставляет функции для автоматического вычисления градиентов в динамических вычислительных графах.

Граф вычислений — это ориентированный граф, который позволяет выражать и визуализировать математические выражения. В контексте глубокого обучения он представляет собой последовательность вычислений, необходимых для получения выходных данных нейронной сети, — нам эта последовательность понадобится для вычисления необходимых градиентов для обратного распространения ошибки, основного алгоритма обучения нейронных сетей.

Рассмотрим конкретный пример, чтобы вы могли познакомиться с концепцией графа вычислений. Код в листинге А.2 реализует прямой проход (этап предсказания) простого классификатора логистической регрессии, которую можно рассматривать как однослойную нейронную сеть. Он возвращает значение от 0 до 1, которое сравнивается с истинным значением класса (0 или 1) при вычислении потерь.

Листинг А.2. Прямой проход логистической регрессии

```
import torch.nn.functional as F  ← Такой импорт модуля — стандартное соглашение в PyTorch,
                                  препятствующее возникновению длинных строк кода

y = torch.tensor([1.0])  ← Истинная метка
x1 = torch.tensor([1.1])  ← Входной признак
w1 = torch.tensor([2.2])  ← Весовой параметр
b = torch.tensor([0.0])  ← Единица смещения
z = x1 * w1 + b  ← Вход сети
a = torch.sigmoid(z)  ← Активация и выход
loss = F.binary_cross_entropy(a, y)
```

Если не все компоненты в предыдущем коде вам понятны, то не волнуйтесь. Цель этого примера не в том, чтобы реализовать классификатор логистической регрессии, а в том, чтобы показать, как можно представить последовательность вычислений в виде графа вычислений (рис. А.7).

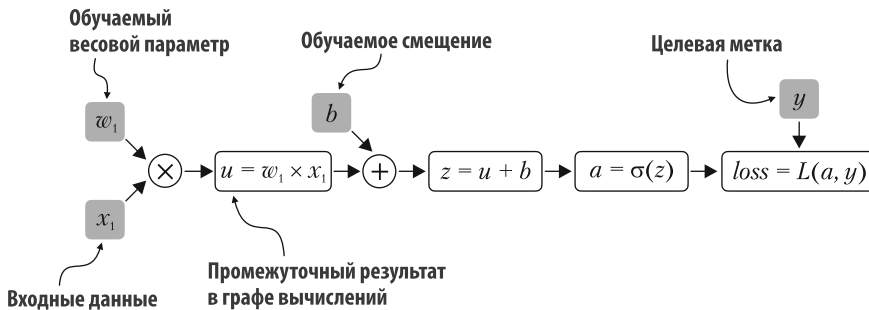


Рис. А.7. Прямой проход логистической регрессии в виде графа вычислений

Как видите, входной признак x_1 умножается на вес модели w_1 и проходит через функцию активации σ после добавления смещения. Потери вычисляются путем сравнения выходных данных модели a с заданной меткой y .

Фактически PyTorch создает такой граф в фоновом режиме, и мы можем использовать его для вычисления градиентов функции потерь по параметрам модели (здесь w_1 и b) для обучения модели.

А.4. Автоматическое дифференцирование упростилось

Если мы выполняем вычисления в PyTorch, то библиотека по умолчанию создает граф вычислений, если для одного из его конечных узлов установлен атрибут `requires_grad` со значением `True`. Это полезно, если мы хотим вычислить градиенты. Они необходимы при обучении нейронных сетей с помощью популярного алгоритма обратного распространения ошибки, который можно рассматривать как реализацию *цепного правила* из математического анализа, используемую для нейронных сетей (рис. А.8).

Наиболее распространенный способ вычисления градиентов потерь в графе вычислений заключается в применении цепного правила справа налево, также называемого автоматическим дифференцированием обратной модели или обратным распространением ошибки. Мы начинаем с выходного слоя (или самой функции потерь) и продвигаемся назад по сети к входному слою. Мы делаем

это, чтобы вычислить градиент функции потерь по каждому параметру (весам и смещениям) в сети, что определяет порядок обновления этих параметров во время обучения.



Рис. А.8. Цепное правило

Частные производные и градиенты. На рис. А.8, помимо цепного правила, показаны частные производные, которые измеряют скорость изменения функции по отношению к одной из ее переменных. *Градиент* — это вектор, содержащий все частные производные многомерной функции, то есть функции, принимающей несколько переменных в качестве входных данных.

Если вы не знакомы с частными производными, градиентами или цепным правилом из математического анализа, то не волнуйтесь. Чтобы работать с содержимым этой книги, вам нужно знать лишь то, что это правило представляет собой способ вычисления градиентов функции потерь с учетом параметров модели в графе вычислений. Это дает информацию, необходимую для обновления каждого параметра в целях минимизации функции потерь, которая служит косвенным показателем, позволяющим определить эффективность модели с помощью такого метода, как градиентный спуск. Мы вернемся к вычислительной реализации этого цикла обучения в PyTorch в разделе А.7.

Как все это связано с механизмом автоматического дифференцирования, вторым компонентом PyTorch, упомянутым ранее? Механизм автоградиента в фоновом режиме создает граф вычислений, отслеживая каждую операцию, выполняемую

с тензорами. Затем, вызвав функцию `grad`, можно вычислить градиент функции потерь по параметру модели w_1 (листинг A.3).

Листинг A.3. Вычисление градиентов с помощью `autograd`

```
import torch.nn.functional as F
from torch.autograd import grad

y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)

z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)

grad_L_w1 = grad(loss, w1, retain_graph=True)
grad_L_b = grad(loss, b, retain_graph=True)
```

По умолчанию PyTorch, вычислив градиенты, разрушает граф вычислений, чтобы освободить память. Но мы вскоре будем использовать его повторно, поэтому устанавливаем `retain_graph=True`, чтобы он оставался в памяти

Полученные значения потерь с учетом параметров модели таковы:

```
print(grad_L_w1)
print(grad_L_b)
```

Результат выглядит так:

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

Здесь мы использовали функцию `grad` вручную, что может быть полезно для экспериментов, отладки и демонстрации концепций. Однако на практике PyTorch предоставляет еще больше высокоуровневых инструментов, позволяющих автоматизировать этот процесс. Например, мы можем вызвать `.backward` для функции потерь, и PyTorch вычислит градиенты всех конечных узлов в графе, которые будут сохранены в атрибутах `.grad` тензоров:

```
loss.backward()
print(w1.grad)
print(b.grad)
```

Результаты таковы:

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

Я предоставил много информации, и математические понятия могут повергнуть вас в замешательство, но не волнуйтесь. Этот математический жаргон служит для объяснения компонента автоградиента в PyTorch, но вам нужно усвоить только то, что библиотека позаботится о вычислениях за нас, используя метод `.backward`, — нам не нужно будет вычислять производные или градиенты вручную.

А.5. Реализация многослойных нейронных сетей

Далее мы сосредоточимся на PyTorch как на библиотеке для реализации глубоких нейронных сетей. В качестве конкретного примера рассмотрим многослойный перцептрон, полносвязную нейронную сеть (рис. А.9).

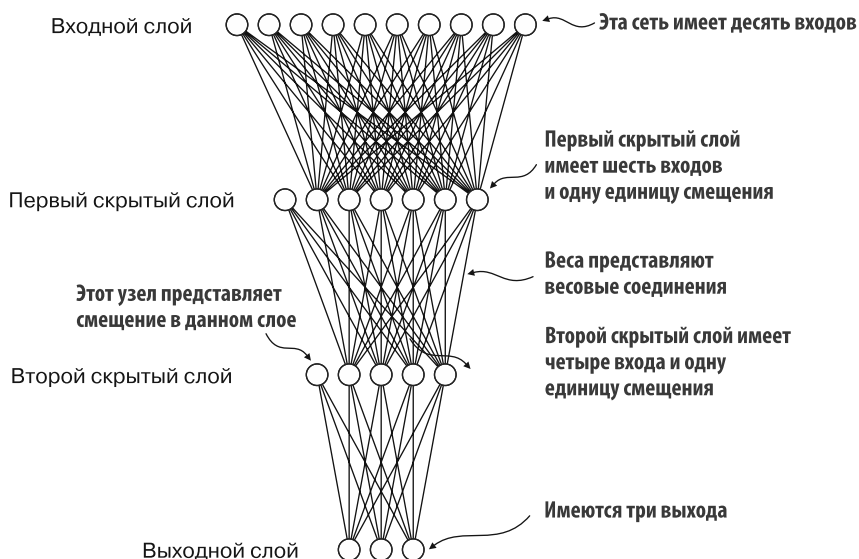


Рис. А.9. Многослойный перцептрон с двумя скрытыми слоями. Каждый узел представляет собой нейрон в соответствующем слое. Для наглядности каждый слой имеет очень небольшое количество узлов

При реализации нейронной сети в PyTorch мы можем создать подкласс `torch.nn.Module` для определения нашей пользовательской сетевой архитектуры. Базовый класс этого `Module` предоставляет множество функциональных возможностей, упрощая создание и обучение моделей. Например, это позволяет изолировать слои и операции каждого слоя нашей модели и отслеживать параметры модели.

В этом подклассе мы определяем слои сети в конструкторе `__init__` и указываем, как слои взаимодействуют в методе `forward`. Этот метод описывает, как входные данные проходят через сеть и объединяются в граф вычислений. В отличие от этого, метод обратного распространения ошибки, который нам обычно не нужно реализовывать самостоятельно, используется во время обучения для вычисления градиентов функции потерь с учетом параметров модели (см. раздел А.7). В листинге А.4 реализуется классический многослойный перцептрон с двумя скрытыми слоями, что позволяет показать типичное использование класса `Module`.

Листинг А.4. Многослойный перцептрон с двумя скрытыми слоями

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()

        self.layers = torch.nn.Sequential(

            # Первый скрытый слой
            torch.nn.Linear(num_inputs, 30),
            torch.nn.ReLU(),

            # Второй скрытый слой
            torch.nn.Linear(30, 20),
            torch.nn.ReLU(),

            # Слой выходных данных
            torch.nn.Linear(20, num_outputs),
        )

    def forward(self, x):
        logits = self.layers(x)
        return logits
```

Кодирование количества входов и выходов в виде переменных позволяет повторно использовать один и тот же код для наборов данных с разным количеством признаков и классов

Слой Linear принимает количество входов и выходов в качестве аргументов

Функции нелинейной активации помещены между скрытыми слоями

Количество выходов одного скрытого слоя должно соответствовать количеству входов следующего слоя

Выходы последнего слоя называются логитами

Создать новый объект нейронной сети можно так:

```
model = NeuralNetwork(50, 3)
```

Прежде чем использовать этот новый объект модели, можно вызвать функцию `print`, чтобы просмотреть краткое описание его структуры:

```
print(model)
```

Результат выглядит так:

```
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

Обратите внимание, что при реализации класса `NeuralNetwork` мы используем класс `Sequential`. Он не является обязательным, но может облегчить работу, если у нас есть несколько слоев, которые мы хотим выполнить в определенном порядке, как в данном случае. Таким образом, после создания экземпляра `self.layers = Sequential(...)` в конструкторе `__init__` нужно будет просто вызвать `self.layers`, вместо того чтобы вызывать каждый слой по отдельности в методе `forward` класса `NeuralNetwork`.

Теперь проверим общее количество обучаемых параметров этой модели:

```
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

В итоге имеем следующий результат:

```
Total number of trainable model parameters: 2213
```

Каждый параметр, для которого `requires_grad=True`, считается обучаемым и будет обновляться во время обучения (см. раздел А.7).

В случае нашей модели нейронной сети с двумя предшествующими скрытыми слоями эти обучаемые параметры содержатся в слоях `torch.nn.Linear`. Слой `Linear` умножает входные данные на матрицу весов и добавляет вектор смещения. Этот слой иногда называют *слоем с прямой связью* или *полносвязным*.

Основываясь на вызове `print(model)`, который мы выполнили здесь, мы можем видеть, что первый слой `Linear` находится в позиции индекса 0 в атрибуте `layers`. Мы можем получить доступ к соответствующей матрице весовых параметров следующим образом:

```
print(model.layers[0].weight)
```

Результат выглядит так:

```
Parameter containing:
tensor (
  [[ 0.1174, -0.1350, -0.1227, ...,  0.0275, -0.0520, -0.0192],
    [-0.0169,  0.1265,  0.0255, ..., -0.1247,  0.1191, -0.0698],
    [-0.0973, -0.0974, -0.0739, ..., -0.0068, -0.0892,  0.1070],
    ...,
    [-0.0681,  0.1058, -0.0315, ..., -0.1081, -0.0290, -0.1374],
    [-0.0159,  0.0587, -0.0916, ..., -0.1153,  0.0700,  0.0770],
    [-0.1019,  0.1345, -0.0176, ...,  0.0114, -0.0559, -0.0088]],
  requires_grad=True)
```

Поскольку эта большая матрица не отображается целиком, то используем атрибут `.shape`, чтобы показать ее размеры:

```
print(model.layers[0].weight.shape)
```

Результат выглядит так:

```
torch.Size([30, 50])
```

(Аналогичным образом вы можете получить доступ к вектору смещения через `model.layers[0].bias`.)

Матрица весов здесь представляет собой матрицу 30×50 , и мы видим, что для параметра `requires_grad` установлено значение `True`. Следовательно, ее

элементы обучаемы — это настройка по умолчанию для весов и смещений в `torch.nn.Linear`.

Если вы запустите предыдущий код на своем компьютере, то числа в весовой матрице, скорее всего, будут отличаться от показанных здесь. Веса модели инициализируются небольшими случайными числами, которые меняются при каждом создании экземпляра сети. В глубоком обучении инициализация этих весов такими числами необходима для нарушения симметрии во время обучения. В противном случае узлы будут выполнять одни и те же операции и обновления во время обратного распространения ошибки, что не позволит сети изучать сложные отображения входных данных на выходные.

Но если мы хотим продолжать использовать небольшие случайные числа в качестве начальных значений для весов нашего слоя, то можем сделать инициализацию случайных чисел воспроизводимой, задав начальное значение генератору случайных чисел PyTorch с помощью `manual_seed`:

```
torch.manual_seed(123)
model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

Результат выглядит так:

```
Parameter containing:
tensor([[ -0.0577,  0.0047, -0.0702, ...,  0.0222,  0.1260,  0.0865],
        [ 0.0502,  0.0307,  0.0333, ...,  0.0951,  0.1134, -0.0297],
        [ 0.1077, -0.1108,  0.0122, ...,  0.0108, -0.1049, -0.1063],
        ...,
        [-0.0787,  0.1259,  0.0803, ...,  0.1218,  0.1303, -0.1351],
        [ 0.1359,  0.0175, -0.0673, ...,  0.0674,  0.0676,  0.1058],
        [ 0.0790,  0.1343, -0.0293, ...,  0.0344, -0.0971, -0.0509]],
        requires_grad=True)
```

Теперь, когда мы уделили некоторое время изучению экземпляра `NeuralNetwork`, посмотрим, как он используется при прямом проходе:

```
torch.manual_seed(123)
X = torch.rand((1, 50))
out = model(X)
print(out)
```

Результат выглядит так:

```
tensor([[ -0.1262,  0.1080, -0.1792]], grad_fn=<AddmmBackward0>)
```

В предыдущем коде мы сгенерировали один случайный обучающий пример `x` (обратите внимание, что наша сеть ожидает 50-мерные векторы признаков) и передали его в модель, которая вернула три оценки. Когда мы вызываем `model(x)`, она автоматически выполняет прямой проход модели.

Прямой проход — это вычисление выходных тензоров на основе входных тензоров. Этот процесс подразумевает прохождение входных данных через все слои нейронной сети, начиная с входного, далее через скрытые слои и заканчивая выходным слоем.

Три числа, возвращаемые при этом, соответствуют оценке, присвоенной каждому из трех выходных узлов. Обратите внимание, что выходной тензор также содержит значение `grad_fn`.

Здесь `grad_fn=<AddmmBackward0>` представляет собой последнюю использованную функцию для вычисления переменной в графе вычислений. В частности, `grad_fn=<AddmmBackward0>` означает, что проверяемый нами тензор был создан с помощью операции перемножения матриц и сложения. PyTorch будет использовать эту информацию при вычислении градиентов во время обратного распространения. Часть `<AddmmBackward0>` в `grad_fn=<AddmmBackward0>` указывает на выполненную операцию. В данном случае это операция `Addmm`. Она означает перемножение матриц (`mm`), за которым следует сложение (`Add`).

Если мы просто хотим использовать сеть без обучения или обратного распространения ошибки (например, если используем ее для предсказания после обучения), то создание этого вычислительного графа для обратного распространения ошибки может быть нецелесообразным, поскольку оно приводит к ненужным вычислениям и требует дополнительной памяти. Итак, когда мы используем модель для предсказания, а не для обучения, лучше всего использовать контекстный менеджер `torch.no_grad()`. Благодаря этому библиотека будет знать, что ей не нужно отслеживать градиенты, а это, в свою очередь, может привести к значительной экономии памяти и вычислительных ресурсов:

```
with torch.no_grad():
    out = model(X)
print(out)
```

Результат следующий:

```
tensor ([-0,1262, 0,1080, -0,1792]))
```

В PyTorch принято программировать модели так, чтобы они возвращали выходные данные последнего слоя (логиты), не передавая их в нелинейную функцию активации. Это связано с тем, что функции потерь, обычно используемые в библиотеке, объединяют операцию `softmax` (или `sigmoid` для бинарной классификации) с отрицательной логарифмической функцией потерь в одном классе. Причина этого — численная эффективность и стабильность. Итак, если мы хотим вычислить вероятности принадлежности к классу для наших прогнозов, то нужно явно вызвать функцию `softmax`:

```
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```

Результат следующий:

```
tensor([[0.3113, 0.3934, 0.2952]]))
```

Теперь эти значения можно интерпретировать как вероятности принадлежности к классу, которые в сумме дают 1. Для этого случайного набора данных значения примерно равны, что ожидаемо для случайно инициализированной модели без обучения.

A.6. Настройка эффективных загрузчиков данных

Прежде чем мы сможем обучить нашу модель, мы должны коротко обсудить создание эффективных загрузчиков данных в PyTorch, с которыми будем работать во время обучения. Общая идея загрузки данных в эту библиотеку показана на рис. A.10.

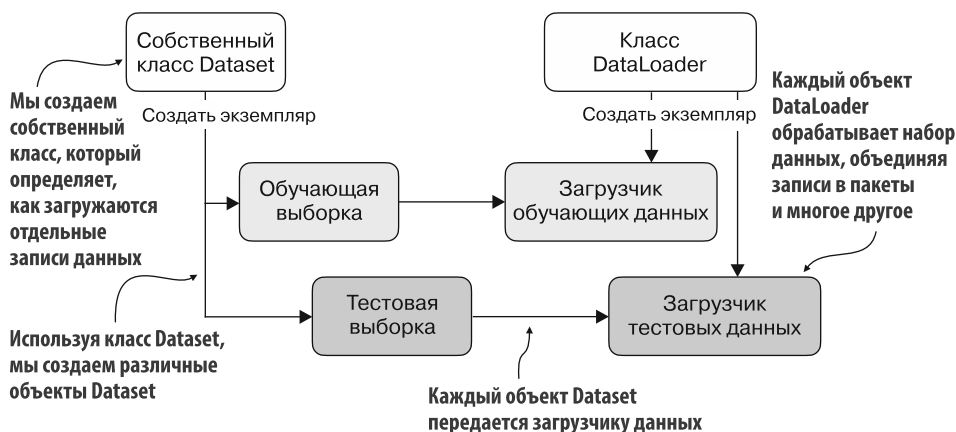


Рис. A.10. PyTorch реализует классы Dataset и DataLoader. Класс Dataset используется для создания объектов, определяющих способ загрузки каждой записи данных. Класс DataLoader определяет способ перетасовки данных и их объединения в пакеты

Следуя рис. A.10, мы реализуем собственный класс Dataset, который будем использовать для создания обучающей и тестовой выборок, с помощью которых позже будем создавать загрузчики данных. Начнем с создания простого небольшого набора данных из пяти обучающих примеров с двумя признаками в каждом. Вместе с обучающими примерами мы создаем и тензор, содержащий соответствующие метки классов: три примера относятся к классу 0, а два — к классу 1. Кроме того, мы создаем тестовый набор, состоящий из двух записей. Код для создания этого набора данных показан в листинге A.5.

Листинг А.5. Создание небольшого игрушечного набора данных

```

X_train = torch.tensor([
    [-1, 2, 3, 1],
    [-0, 9, 2, 9],
    [-0, 5, 2, 6],
    [2, 3, -1, 1],
    [2, 7, -1, 5]
])
y_train = torch.tensor([0, 0, 0, 1, 1])

X_test = torch.tensor([
    [-0, 8, 2, 8],
    [2, 6, -1, 6],
])
y_test = torch.tensor([0, 1])

```

ПРИМЕЧАНИЕ PyTorch требует, чтобы метки классов начинались с 0, а наибольшее значение метки не должно превышать количество выходных узлов минус 1 (поскольку в Python счет начинается с нуля). Таким образом, если у нас есть метки классов 0, 1, 2, 3 и 4, то выходной слой нейронной сети должен состоять из пяти узлов.

Далее мы создаем собственный класс набора данных `ToyDataset`, наследуя его от родительского класса PyTorch `Dataset` (листинг А.6).

Листинг А.6. Определение пользовательского класса Dataset

```

from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0]

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)

```

Инструкции по извлечению
ровно одной записи данных
и соответствующей метки

← Инструкция по возвращению
общей длины набора данных

Цель данного пользовательского класса `ToyDataset` — создание экземпляра PyTorch `DataLoader`. Но прежде чем мы перейдем к этому шагу, кратко рассмотрим общую структуру кода `ToyDataset`.

В PyTorch тремя основными компонентами пользовательского класса `Dataset` являются конструктор `__init__`, а также методы `__getitem__` и `__len__` (см. листинг А.6). В конструкторе `__init__` мы задаем атрибуты, к которым можем получить доступ позже в методах `__getitem__` и `__len__`. Это могут быть пути к файлам, файловые объекты, подключения к базам данных и т. д. Мы создали тензорный набор, который хранится в памяти, поэтому просто присваиваем `X` и `y` этим атрибутам, которые являются заполнителями для наших тензорных объектов.

В методе `__getitem__` мы определяем инструкции для возврата ровно одного элемента из набора данных по индексу. Это относится к функциям и метке класса, соответствующим одному обучающему примеру или тестовому экземпляру. (Загрузчик данных предоставит этот индекс, о котором мы вскоре поговорим.)

Наконец, метод `__len__` содержит инструкции по получению длины набора данных. Здесь мы используем атрибут `.shape` тензора, чтобы получить количество строк в массиве. В случае с обучающей выборкой у нас есть пять строк, и этот факт мы можем перепроверить:

```
print(len(train_ds))
```

Результат следующий:

```
5
```

Теперь, определив класс PyTorch `Dataset`, который можно использовать для нашего набора данных, мы можем использовать класс `DataLoader` для выборки из него (листинг А.7).

Листинг А.7. Создание экземпляров загрузчиков данных

```
from torch.utils.data import DataLoader

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,  # ← Экземпляр ToyDataset, созданный ранее,
                       #      служит входом для загрузчика данных
    batch_size=2,
    shuffle=True,      # ← Перемешивать или нет данные
    num_workers=0      # ← Количество фоновых процессов
)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,     # ← Не нужно перетасовывать
                       #      тестовую выборку
    num_workers=0
)
```

После создания экземпляра загрузчика обучающих данных мы можем выполнить итерацию по данным. Итерация по `test_loader` работает аналогично, но здесь для краткости опущена:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

Результат выглядит так:

```
Batch 1: tensor([[ -1,2000,  3,1000],
                 [-0,5000,  2,6000]]) tensor([0, 0])
Batch 2: tensor([[ 2,3000, -1,1000],
                 [-0,9000,  2,9000]]) tensor([1, 0])
Batch 3: tensor([[ 2,7000, -1,5000]]) tensor([1])
```

Как мы видим по этому результату, `train_loader` перебирает набор обучающих данных, посещая каждый обучающий пример ровно один раз. Этот процесс известен как *эпоха обучения*. Мы задали начальное значение для генератора случайных чисел с помощью `torch.manual_seed(123)`, поэтому вы должны получить точно такой же порядок перестановки обучающих примеров. Но если вы пройдетесь по набору данных во второй раз, то увидите, что порядок перестановки изменится. Это необходимо для того, чтобы глубокие нейронные сети не попадали в повторяющиеся циклы обновления во время обучения.

Здесь мы указали размер пакета, равный 2, но третий пакет содержит только один пример. Это потому, что у нас пять обучающих примеров, а 5 не делится на 2 без остатка. На практике наличие значительно меньшего пакета в качестве последнего на обучающей эпохе может нарушить сходимость во время обучения. Чтобы предотвратить это, установите `drop_last=True`, что приведет к удалению последнего пакета на каждой эпохе (листинг А.8).

Листинг А.8. Обучающий загрузчик, удаляющий последнюю партию

```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
```

Теперь, перебирая данные с помощью обучающего загрузчика, мы видим, что последний пакет пропущен:

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```


Результат выглядит так:

```
Batch 1: tensor([[ -0.9000,  2.9000],
                 [ 2.3000, -1.1000]]) tensor([0, 1])
Batch 2: tensor([[ 2.7000, -1.5000],
                 [-0.5000,  2.6000]]) tensor([1, 0])
```

Наконец, обсудим параметр `num_workers=0` в `DataLoader`. Этот параметр в функции `DataLoader` библиотеки PyTorch имеет решающее значение для распараллеливания загрузки при обработке данных. Если `num_workers` равно 0, то загрузка данных будет выполняться в основном процессе, а не в отдельных рабочих процессах. Это может не выглядеть как проблема, но способно привести к значительному замедлению обучения модели, когда мы обучаем большие сети на GPU. Вместо того чтобы сосредоточиться исключительно на обработке модели глубокого обучения, CPU должен тратить время на загрузку и предварительную обработку данных. В результате GPU может простаивать в ожидании завершения этих задач CPU. Напротив, когда для параметра `num_workers` задано значение больше 0, запускается несколько рабочих процессов для параллельной загрузки данных, что позволяет основному процессу сосредоточиться на обучении вашей модели и более эффективном использовании ресурсов вашей системы (рис. A.11).

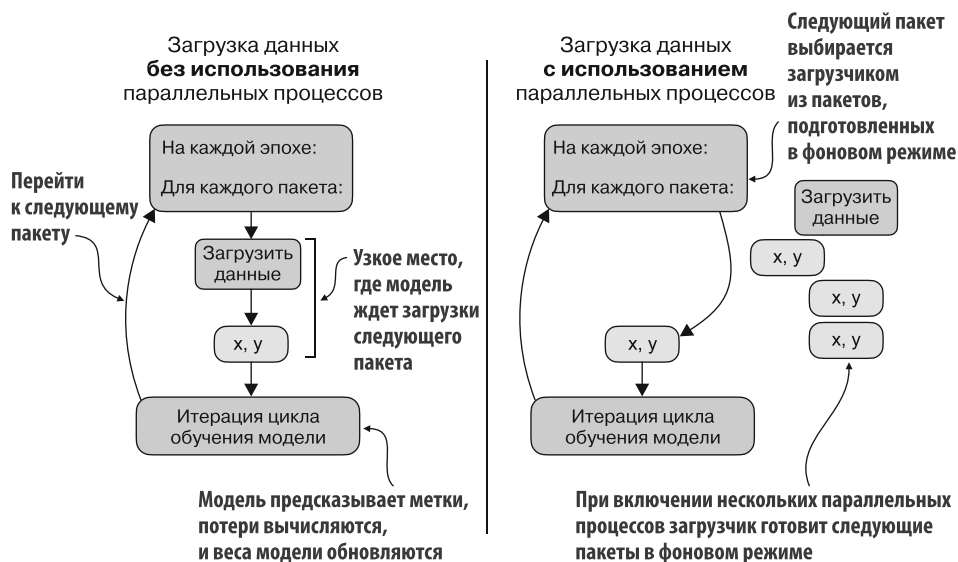


Рис. A.11. Загрузка данных без использования нескольких рабочих процессов (установка `num_workers=0`) приведет к возникновению «узкого места», когда модель простаивает до загрузки следующего пакета (слева). Если использовать несколько рабочих процессов, то загрузчик может поставить в очередь следующий пакет в фоновом режиме (справа)

Но если мы работаем с очень маленькими наборами данных, то установка значения `num_workers` равным 1 или большему числу может не понадобиться, поскольку общее время обучения в любом случае занимает доли секунды. Итак, если вы работаете с маленькими наборами данных или интерактивными средами, такими как блокноты Jupyter, то увеличение `num_workers` может не привести к заметному ускорению. На самом деле это может вызвать некоторые проблемы. Одна из потенциальных трудностей — затраты на запуск нескольких рабочих процессов, которые могут занять больше времени, чем фактическая загрузка данных, если ваш набор данных небольшой.

Кроме того, в случае с блокнотами Jupyter установка `num_workers` больше 0 иногда может привести к проблемам, связанным с совместным использованием ресурсов между различными процессами, что приводит к ошибкам или сбоям блокнота. Поэтому важно найти компромисс и принять взвешенное решение о настройке параметра `num_workers`. При правильном использовании он может быть полезным инструментом, но для достижения оптимальных результатов его следует адаптировать к размеру вашего конкретного набора данных и вычислительной среде.

По моему опыту, установка `num_workers = 4` обычно обеспечивает оптимальную производительность на многих реальных наборах данных, но оптимальные настройки зависят от вашего оборудования и кода, используемого для загрузки обучающего примера, определенного в классе `Dataset`.

А.7. Типичный цикл обучения

Теперь мы обучим нейронную сеть на нашем наборе данных. В листинге А.9 показан код обучения.

Листинг А.9. Обучение нейронной сети в PyTorch

```
import torch.nn.functional as F

torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)
optimizer = torch.optim.SGD(
    model.parameters(), lr=0.5
)
num_epochs = 3
for epoch in range(num_epochs):

    model.train()

    for batch_idx, (features, labels) in enumerate(train_loader):
        logits = model(features)
```

Оптимизатор должен знать, какие параметры улучшать

Набор данных содержит два признака и два класса

```

loss = F.cross_entropy(logits, labels)

optimizer.zero_grad()
loss.backward()
optimizer.step()

### LOGGING
print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
      f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
      f" | Train Loss: {loss:.2f}")

model.eval()
# Insert optional model evaluation code

```

Устанавливает градиенты из предыдущего раунда в 0, чтобы предотвратить непреднамеренное накопление градиентов

Вычисляет градиенты потерь с учетом параметров модели

Оптимизатор использует градиенты, чтобы обновлять параметры модели

Выполнение этого кода приводит к следующим результатам:

```

Epoch: 001/003 | Batch 000/002 | Train Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train Loss: 0.00

```

Как мы видим, после трех эпох потери достигают 0. Это говорит о том, что модель сходится на обучающей выборке. Здесь мы инициализируем модель с двумя входами и двумя выходами, поскольку в нашем наборе данных есть два входных признака и две метки классов. Мы использовали оптимизатор стохастического градиентного спуска (SGD) с частотой обучения (1r) 0,5. Частота обучения — это гиперпараметр, то есть настраиваемая величина, с которой мы должны экспериментировать, наблюдая за потерями. В идеале мы хотим выбрать такую скорость обучения, чтобы потери сходились через определенное количество периодов — оно является еще одним гиперпараметром, который можно выбрать.

Упражнение А.3. Определение параметров нейронной сети

Сколько параметров имеет нейронная сеть, представленная в листинге А.9?

На практике мы часто используем третий набор данных, так называемую проверочную выборку, чтобы найти оптимальные настройки гиперпараметра. Проверочная выборка похожа на тестовую. Но в то время как мы хотим использовать тестовую выборку только один раз, чтобы избежать необъективной оценки модели, проверочную мы обычно задействуем несколько раз, чтобы настроить параметры модели.

Мы также ввели новые настройки, называемые `model.train()` и `model.eval()`. Как следует из их названий, они служат для перевода модели в режим обучения и режим оценки соответственно. Это необходимо для компонентов, которые ведут себя по-разному во время обучения и предсказания, таких как слои с *отсевом* или *пакетной нормализацией*. У нас нет отсева или других компонентов в нашем классе `NeuralNetwork`, на которые влияют эти настройки, поэтому использование `model.train()` и `model.eval()` в предыдущем коде избыточно. Однако лучше все равно их использовать, чтобы избежать неожиданного поведения при изменении архитектуры модели или повторном использовании кода в целях обучения другой модели.

Как обсуждалось ранее, мы передаем логиты непосредственно в функцию потерь `cross_entropy`, которая внутри применяет функцию `softmax` для повышения эффективности и стабильности вычислений. Затем при вызове `loss.backward()` будут вычислены градиенты в графе вычислений, который PyTorch создала в фоновом режиме. Метод `optimizer.step()` будет использовать градиенты для обновления параметров модели, чтобы минимизировать потери. В случае оптимизатора `SGD` это означает умножение градиентов на скорость обучения и добавление масштабированного отрицательного градиента к параметрам.

ПРИМЕЧАНИЕ Чтобы предотвратить нежелательное накопление градиентов, важно добавлять в каждый раунд обновления вызов `optimizer.zero_grad()` для сброса градиентов до 0. В противном случае градиенты будут накапливаться, что может быть нежелательно.

После обучения модели мы можем использовать ее для предсказания:

```
model.eval()
with torch.no_grad():
    outputs = model(X_train)
print(outputs)
```

Результаты выглядят так:

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
        [ 2.0944, -3.1820],
        [-1.4814,  1.4816],
        [-1.7176,  1.7342]])
```

Для получения вероятностей принадлежности к классам мы можем применять в PyTorch функцию `softmax`:

```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)
```

Выполнение этого кода приводит к следующим результатам:

```
tensor([[ 0.9991,  0.0009],
        [ 0.9982,  0.0018],
        [ 0.9949,  0.0051],
        [ 0.0491,  0.9509],
        [ 0.0307,  0.9693]])
```

Рассмотрим первую строку этого результата. Первое значение говорит о том, что обучающий пример с вероятностью 99,91 % относится к классу 0 и с вероятностью 0,09 % — к классу 1. (Вызов `set_printoptions` используется здесь для более наглядного отображения результатов.)

Мы можем преобразовать эти значения в предсказания классов с помощью функции `argmax` в PyTorch, которая возвращает индекс максимального значения в каждой строке, если мы зададим `dim=1` (при `dim=0` будет возвращено максимальное значение в каждом столбце):

```
predictions = torch.argmax(probas, dim=1)
print(predictions)
```

Получаем следующий результат:

```
tensor([0, 0, 0, 1, 1])
```

Обратите внимание, что для получения меток классов необязательно вычислять вероятности `softmax`. Кроме того, можно применить функцию `argmax` непосредственно к логитам (выходам):

```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

Результат выглядит так:

```
tensor([0, 0, 0, 1, 1])
```

Здесь мы получили предсказанные метки для обучающей выборки. Она невелика, поэтому мы можем сравнить результат с истинными обучающими метками и убедиться, что модель на 100 % верна. Мы можем перепроверить это с помощью оператора сравнения `==`:

```
predictions == y_train
```

Итог:

```
tensor([True, True, True, True, True])
```

Используя `torch.sum`, мы можем подсчитать количество правильных предсказаний:

```
torch.sum(predictions == y_train)
```

Результат следующий:

5

Выборка состоит из пяти обучающих примеров, поэтому у нас есть пять правильных предсказаний из пяти, что составляет $5 / 5 \times 100 \% = 100 \%$ точности.

Чтобы обобщить вычисление точности предсказаний, реализуем функцию `compute_accuracy` (листинг А.10).

Листинг А.10. Функция для вычисления точности предсказаний

```
def compute_accuracy(model, dataloader):

    model = model.eval()
    correct = 0.0
    total_examples = 0

    for idx, (features, labels) in enumerate(dataloader):

        with torch.no_grad():
            logits = model(features)

            predictions = torch.argmax(logits, dim=1)
            compare = labels == predictions
            correct += torch.sum(compare)
            total_examples += len(compare)

    return (correct / total_examples).item()
```

Возвращает тензор, состоящий из значений True/ False

Оператор суммирования подсчитывает количество значений True

Доля правильных предсказаний, значение от 0 до 1. `.item()` возвращает значение тензора в виде числа с плавающей запятой

Код вычисляет количество и долю правильных предсказаний. Работая с большими наборами данных, мы обычно можем вызывать модель только для небольшой части набора данных из-за ограничений памяти. Функция `compute_accuracy` здесь — это общий метод, который масштабируется для наборов данных произвольного размера, поскольку на каждой итерации модель получает фрагмент набора того же размера, что и размер пакета, который она видела во время обучения. Внутреннее устройство функции `compute_accuracy` похоже на то, что мы использовали ранее, когда преобразовывали логиты в метки классов.

Затем мы можем применить эту функцию к обучающим данным:

```
print(compute_accuracy(model, train_loader))
```

Получаем:

1.0

Аналогичным образом мы можем применить эту функцию к тестовым данным:

```
print(compute_accuracy(model, test_loader))
```

Результат:

1.0

А.8. Сохранение и загрузка моделей

Теперь, когда мы обучили нашу модель, посмотрим, как ее сохранить, чтобы использовать повторно. Рекомендуемый способ сохранения и загрузки моделей в PyTorch выглядит так:

```
torch.save(model.state_dict(), "model.pth")
```

`state_dict` модели — это словарь Python, который проецирует каждый слой модели на его обучаемые параметры (веса и смещения). `model.pth` — это произвольное имя файла модели, сохраненного на диске. Мы можем дать ему любое имя и присвоить любое расширение, которые нам нравятся; однако `.pth` и `.pt` являются наиболее распространенными.

После сохранения модели мы можем восстановить ее с диска:

```
model = NeuralNetwork(2, 2)
model.load_state_dict(torch.load("model.pth"))
```

Функция `torch.load("model.pth")` считывает файл `model.pth` и воссоздает словарь Python, содержащий параметры модели, а `model.load_state_dict()` применяет эти параметры к модели, фактически восстанавливая ее обученное состояние на момент сохранения.

Строка кода `model = NeuralNetwork(2, 2)` не является строго обязательной, если вы выполняете этот код в том же сеансе, в котором сохранили модель. Однако я добавил ее сюда, чтобы показать, что нам нужен экземпляр модели в памяти, чтобы сохраненные параметры можно было применять. Здесь архитектура `NeuralNetwork(2, 2)` должна точно соответствовать исходной сохраненной модели.

А.9. Оптимизация производительности обучения с помощью графических процессоров

Далее мы рассмотрим, как использовать GPU, которые ускоряют обучение глубоких нейронных сетей в сравнении с обычными процессорами. Сначала мы рассмотрим основные концепции, лежащие в основе вычислений на GPU в PyTorch. Затем обучим модель на одном таком процессоре. Наконец, мы рассмотрим распределенное обучение, в котором используются несколько графических процессоров.

А.9.1. Вычисления PyTorch на графических процессорах

Изменить обучающий цикл так, чтобы код запускался на GPU, относительно просто — нужно исправить всего три строки кода (см. раздел А.7). Прежде чем мы сделаем это, важно понять основную концепцию вычислений на графическом процессоре в рамках библиотеки PyTorch. В ней устройство — это место, где выполняются вычисления и хранятся данные. Центральный и графический процессоры — примеры устройств. Тензор PyTorch находится в устройстве, и его операции выполняются на том же устройстве.

Посмотрим, как это работает. Предположим, что мы установили версию PyTorch, совместимую с GPU (см. подраздел А.1.3). Мы можем перепроверить, действительно ли наша среда выполнения поддерживает вычисления на GPU, используя такой код:

```
print(torch.cuda.is_available())
```

Результат:

```
True
```

Теперь предположим, что у нас есть два тензора, которые мы можем сложить; по умолчанию это вычисление будет выполняться на центральном процессоре:

```
tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.])
print(tensor_1 + tensor_2)
```

Вывод следующий:

```
tensor([5., 7., 9.])
```

Теперь мы можем использовать метод `.to()`, чтобы перенести эти тензоры на графический процессор и выполнить там сложение. Этот метод аналогичен тому, который мы используем для изменения типа данных тензора (см. подраздел А.2.2):

```
tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda")
print(tensor_1 + tensor_2)
```

Результат выглядит так:

```
tensor([5., 7., 9.], device='cuda:0')
```

Полученный тензор теперь содержит информацию об устройстве, `device='cuda:0'`; это значит, тензоры находятся на первом графическом процессоре. Если на вашем компьютере несколько таких процессоров, то вы можете указать, на какой из них хотите перенести тензоры. Для этого укажите идентификатор устройства в команде

переноса. Например, вы можете использовать `.to("cuda:0")`, `.to("cuda:1")` и т. д. Однако все тензоры должны находиться на одном устройстве. Если один тензор находится на обычном процессоре, а другой — на графическом, то вычисление завершится ошибкой:

```
tensor_1 = tensor_1.to("cpu")
print(tensor_1 + tensor_2)
```

Получаем следующий результат:

```
RuntimeError      Traceback (most recent call last)
<ipython-input-7-4ff3c4d20fc3> в <строке: 2>()
      1 tensor_1 = tensor_1.to("cpu")
----> 2 print(tensor_1 + tensor_2)
RuntimeError: Expected all tensors to be on the same device, but found at
least two devices, cuda:0 and cpu!
```

В общем, нужно только перенести тензоры на одно и то же устройство GPU, а PyTorch сделает все остальное.

А.9.2. Обучение с использованием одного графического процессора

Теперь, когда вы знакомы с передачей тензоров на GPU, можно изменить цикл обучения для работы на таком процессоре. Для этого нужно изменить всего три строки кода (листинг А.11).

Листинг А.11. Обучающий цикл на GPU

```
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)

device = torch.device("cuda")  ← Определяет тип устройства
                                (по умолчанию GPU)
model = model.to(device)      ← Переносит модель на GPU

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):

    model.train()
    for batch_idx, (features, labels) in enumerate(train_loader):
        features, labels = features.to(device), labels.to(device) ← Переносит
                                                                       данные
                                                                       на GPU
        logits = model(features)
        loss = F.cross_entropy(logits, labels) # Функция потерь

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```

### LOGGING
print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
      f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
      f" | Train/Val Loss: {loss:.2f}")

model.eval()
# Вставьте дополнительный код оценки модели

```

При запуске предыдущего кода будет выведена следующая информация, аналогичная результатам, полученным на центральном процессоре (см. раздел А.7):

```

Epoch: 001/003 | Batch 000/002 | Train/Val Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train/Val Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train/Val Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train/Val Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train/Val Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train/Val Loss: 0.00

```

Мы можем использовать `.to("cuda")` вместо `device = torch.device("cuda")`. Перенос тензора на "cuda" вместо `torch.device("cuda")` работает так же и является более коротким (см. подраздел А.9.1). Кроме того, можно изменить оператор, чтобы сделать тот же код исполняемым на центральном процессоре, если графический недоступен. Это считается лучшей практикой при распространении кода PyTorch:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

В случае с измененным циклом обучения мы, вероятно, не увидим ускорения из-за затрат, которых требует передача данных из CPU в GPU. Однако мы можем ожидать значительного ускорения при обучении глубоких нейронных сетей, особенно больших языковых моделей.

PyTorch на macOS

На Apple Mac с чипом Apple Silicon (например, M1, M2, M3 или более новые модели) вместо компьютера с графическим процессором Nvidia вы можете поменять

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

на

```
device = torch.device(
    "mps" if torch.backends.mps.is_available() else "cpu"
)
```

чтобы воспользоваться преимуществами этого чипа.

Упражнение А.4. Сравнение времени выполнения

Сравните время выполнения перемножения матриц на центральном и графическом процессорах. При каком размере матрицы перемножение на GPU становится более быстрым, чем на CPU? Подсказка: используйте команду `%timeit` в Jupyter, чтобы сравнить время выполнения. Например, для матриц `a` и `b` выполните команду `%timeit a@b` в новой строке.

А.9.3. Обучение с использованием нескольких графических процессоров

Распределенное обучение — концепция разделения процесса обучения модели между несколькими GPU и машинами. Зачем это нужно? Даже если можно обучить модель на одном GPU или машине, процесс может занять очень много времени. Его можно значительно сократить, распределив между несколькими машинами, каждая из которых потенциально может иметь несколько графических процессоров. Это особенно важно на экспериментальных этапах разработки модели, когда для тонкой настройки параметров и архитектуры модели может потребоваться множество итераций обучения.

ПРИМЕЧАНИЕ Для выполнения примеров из этой книги не требуется доступ к нескольким GPU или их использование. Данный раздел предназначен для тех, кто интересуется работой вычислений с несколькими графическими процессорами в PyTorch.

Начнем с самого простого случая распределенного обучения: стратегии PyTorch `DistributedDataParallel` (DDP). Она обеспечивает параллелизм за счет разделения входных данных между доступными устройствами и одновременной обработки этих подмножеств данных.

Как работает эта стратегия? PyTorch запускает отдельный процесс на каждом GPU, и каждый процесс получает и хранит копию модели; эти копии будут синхронизироваться во время обучения. Для наглядности предположим, что у нас есть два GPU, которые мы хотим использовать для обучения нейронной сети (рис. А.12).

Этот процесс передачи состоит из двух ключевых этапов. Сначала мы создаем копию модели на каждом из GPU. Затем на каждой итерации обучения каждая модель получит мини-пакет (или просто «пакет») от загрузчика данных. Мы можем использовать `DistributedSampler`, чтобы каждый GPU получал разные, непересекающиеся пакеты данных.

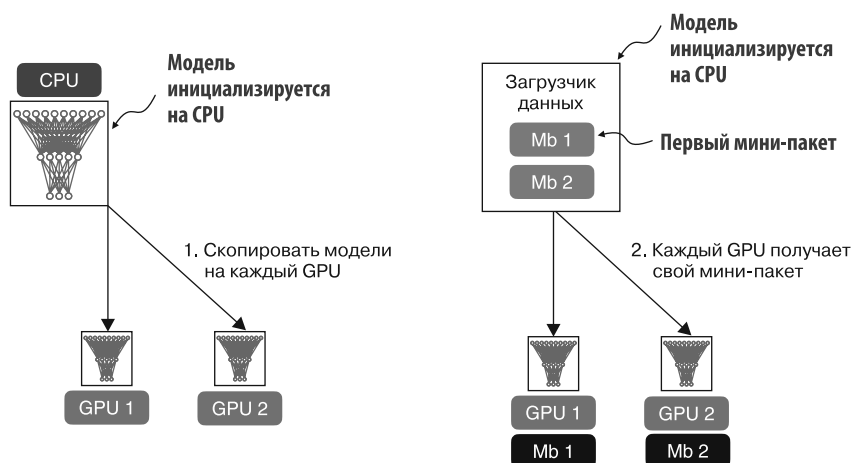


Рис. А.12. Передача модели и данных в DDP

Каждая копия модели будет видеть разные выборки обучающих данных, поэтому копии модели будут возвращать разные логиты в качестве выходных данных и вычислять разные градиенты во время обратного прохода. Затем эти градиенты усредняются и синхронизируются во время обучения для обновления моделей. Это позволяет гарантировать, что модели не будут различаться (рис. А.13).

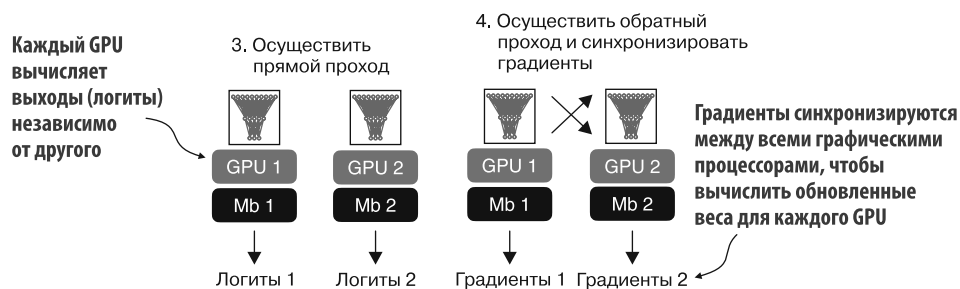


Рис. А.13. Прямой и обратный проходы в DDP выполняются независимо на каждом GPU с соответствующим подмножеством данных. Как только они завершаются, градиенты из каждой копии модели (на каждом GPU) синхронизируются на всех GPU. Это гарантирует, что каждая копия модели будет иметь одинаковые обновленные веса

Преимущество использования DDP заключается в повышении скорости обработки набора данных по сравнению с одним графическим процессором. Если

не учитывать незначительные накладные расходы на связь между устройствами, возникающие при использовании DDP, то теоретически использование двух GPU позволяет обработать одну эпоху обучения вдвое быстрее, чем с помощью одного. Эффективность по времени увеличивается по мере роста количества GPU, что позволяет обрабатывать эпоху в восемь раз быстрее, если процессоров восемь, и т. д.

ПРИМЕЧАНИЕ DDP не работает должным образом в интерактивных средах Python, таких как блокноты Jupyter, которые не поддерживают многопроцессорную обработку так же, как автономный скрипт Python. Вследствие этого код, приведенный ниже, должен выполняться как скрипт, а не как блокнот Jupyter. DDP требует запускать несколько процессов, и у каждого из них должен быть свой экземпляр интерпретатора Python.

Теперь посмотрим, как это работает на практике. Для краткости я сосредоточусь на основных частях кода, которые необходимо изменить в целях обучения DDP. Однако читателям, которые хотят запустить код на собственном компьютере с несколькими графическими процессорами или в облаке по своему выбору, следует использовать автономный скрипт, представленный в репозитории GitHub данной книги по адресу <https://github.com/rasbt/LLMs-from-scratch>.

Сначала мы импортируем несколько дополнительных подмодулей, классов и функций для распределенного обучения PyTorch (листинг А.12).

Листинг А.12. Утилиты PyTorch для распределенного обучения

```
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group
```

Прежде чем углубляться в изучение изменений, необходимых для того, чтобы сделать обучение совместимым с DDP, коротко рассмотрим обоснование и использование этих недавно импортированных утилит, которые нам понадобятся вместе с классом `DistributedDataParallel`.

Подмодуль `multiprocessing` содержит такие функции, как `multiprocessing.spawn`, которые мы будем использовать для запуска процессов обучения (по одному на каждый GPU) и параллельного применения функции к нескольким входным данным. Если мы запускаем несколько процессов, то нам понадобится способ разделить между ними набор данных. Для этого мы будем использовать `DistributedSampler`.

`init_process_group` и `destroy_process_group` используются для инициализации и завершения работы модулей распределенного обучения. Функция

`init_process_group` должна вызываться в начале сценария обучения как способ инициализации группы процессов, а функция `destroy_process_group` должна вызываться в конце обучения как средство уничтожения заданной группы процессов и освобождения ее ресурсов. Код в листинге А.13 показывает, как эти новые компоненты используются для реализации DDP обучения модели `NeuralNetwork`, которую мы реализовали ранее.

Листинг А.13. Обучение модели с использованием стратегии `DistributedDataParallel`

```
def ddp_setup(rank, world_size):
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "12345"
    init_process_group(
        backend="nccl",
        rank=rank,
        world_size=world_size
    )
    torch.cuda.set_device(rank)

def prepare_dataset():
    # Вставить код подготовки набора данных
    train_loader = DataLoader(
        dataset=train_ds,
        batch_size=2,
        shuffle=False,
        pin_memory=True,
        drop_last=True,
        sampler=DistributedSampler(train_ds)
    )
    return train_loader, test_loader

def main(rank, world_size, num_epochs):
    train_loader, test_loader = prepare_dataset()
    model = NeuralNetwork(num_inputs=2, num_outputs=2)
    model.to(rank)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
    model = DDP(model, device_ids=[rank])
    for epoch in range(num_epochs):
        for features, labels in train_loader:
            features, labels = features.to(rank), labels.to(rank)
            # Вставить модель предсказания и код обратного распространения
            print(f"[GPU{rank}] Epoch: {epoch+1:03d}/{num_epochs:03d}"
                  f" | Batchsize {labels.shape[0]:03d}"
                  f" | Train/Val Loss: {loss:.2f}")
        model.eval()
    train_acc = compute_accuracy(model, train_loader, device=rank)
    print(f"[GPU{rank}] Training accuracy", train_acc)
```

Адрес главного узла

Любой свободный порт на компьютере

nccl расшифровывается как NVIDIA Collective Communication Library

rank обозначает индекс GPU, который мы хотим использовать

world_size обозначает требуемое количество GPU

Делает текущим GPU, на котором будут храниться тензоры и выполняться операции

DistributedSampler делает перетасовку

Обеспечивает более быструю передачу данных при обучении на GPU

Разделяет набор данных на непересекающиеся подмножества для каждого процесса (GPU)

Функция main, запускающая обучение модели

rank — это GPU ID или индекс

```

test_acc = compute_accuracy(model, test_loader, device=rank)
print(f"[GPU{rank}] Test accuracy", test_acc)
destroy_process_group()
if __name__ == "__main__":
    print("Number of GPUs available:", torch.cuda.device_count())
    torch.manual_seed(123)
    num_epochs = 3
    world_size = torch.cuda.device_count()
    mp.spawn(main, args=(world_size, num_epochs), nprocs=world_size)

```

Освобождает выделенные ресурсы

Запускает функцию main, используя несколько процессов, где nprocs=world_size означает один процесс на каждый GPU

Прежде чем запускать этот код, коротко рассмотрим, как он работает. Внизу у нас есть условие `__name__ == "__main__"`, содержащее код, который выполняется, когда мы запускаем код как скрипт Python, а не импортируем его как модуль.

Этот код сначала выводит количество доступных GPU с помощью `torch.cuda.device_count()`, устанавливает значение генератора случайных чисел для воспроизводимости результатов, а затем запускает новые процессы с помощью функции PyTorch `multiprocessing.spawn`. Здесь функция `spawn` запускает по одному процессу на каждый GPU с параметром `nprocs=world_size`, где `world_size` — количество доступных GPU. Функция `spawn` запускает код в функции `main`, которую мы определяем в том же скрипте, с некоторыми дополнительными аргументами, переданными через `args`. Обратите внимание, что у функции `main` есть аргумент `rank`, который мы не добавляем в вызов `mp.spawn()`. Это связано с тем, что `rank`, который относится к идентификатору процесса, используемому в качестве идентификатора GPU, уже передается автоматически.

Функция `main` настраивает распределенную среду с помощью `ddp_setup` (еще одной функцией, которую мы определили), загружает обучающую и тестовую выборки, настраивает модель и выполняет обучение. По сравнению с обучением на одном графическом процессоре (см. подраздел A.9.2) теперь мы передаем модель и данные на целевое устройство с помощью `.to(rank)`, где `rank` — это идентификатор GPU. Кроме того, мы оборачиваем модель с помощью DDP, что позволяет синхронизировать градиенты между разными графическими процессорами во время обучения. После завершения обучения и оценки моделей мы используем `destroy_process_group()`, чтобы корректно завершить распределенное обучение и освободить выделенные ресурсы.

Ранее я упоминал, что каждый графический процессор будет получать разные подмножества обучающих данных. Чтобы гарантировать это, мы установили `sampler=DistributedSampler(train_ds)` в загрузчике обучающих данных.

Последняя функция, которую следует обсудить, — это `ddp_setup`. Она устанавливает адрес и порт главного узла, чтобы обеспечить связь между различными

процессами, инициализирует группу процессов с помощью серверной части NCCL (предназначенной для взаимодействия GPU) и устанавливает `rank` (идентификатор процесса) и `world_size` (общее количество процессов). Наконец, она указывает на устройство GPU, соответствующее текущему рангу процесса обучения модели.

Выбор доступных графических процессоров на компьютере с несколькими GPU. Если вы хотите ограничить количество графических процессоров, используемых для обучения на компьютере с несколькими GPU, то проще всего использовать переменную среды `CUDA_VISIBLE_DEVICES`. Для наглядности предположим, что на вашем компьютере несколько GPU и вы хотите использовать только один — например, с индексом 0. Вместо `python some_script.py` вы можете запустить следующий код из терминала:

```
CUDA_VISIBLE_DEVICES=0 python some_script.py
```

Или если на вашем компьютере четыре GPU и вы хотите использовать только первый и третий, то можете использовать такой код:

```
CUDA_VISIBLE_DEVICES=0,2 python some_script.py
```

Настройка `CUDA_VISIBLE_DEVICES` таким образом — простой и эффективный способ управления распределением графических процессоров, не требующий изменения скриптов PyTorch.

Теперь запустим этот код как скрипт из терминала и посмотрим, как он работает на практике:

```
python ch02-DDP-script.py
```

Обратите внимание, что он должен работать на компьютерах как с одним графическим процессором, так и с несколькими. Если мы запустим этот код на компьютере с одним GPU, то увидим следующий результат:

```
PyTorch version: 2.2.1+cu117
CUDA available: True
Number of GPUs available: 1
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.62
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.32
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.11
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.07
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.02
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.03
[GPU0] Training accuracy 1.0
[GPU0] Test accuracy 1.0
```


Этот вывод выглядит так же, как при использовании одного графического процессора (см. подраздел A.9.2), что является хорошей проверкой работоспособности.

Теперь, если мы запустим ту же команду и код на компьютере с двумя GPU, то увидим следующее:

```
PyTorch version: 2.2.1+cu117
CUDA available: True
Number of GPUs available: 2
[GPU1] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.60
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.59
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.16
[GPU1] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.17
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Training accuracy 1.0
[GPU0] Training accuracy 1.0
[GPU1] Test accuracy 1.0
[GPU0] Test accuracy 1.0
```

Как и ожидалось, мы видим, что некоторые пакеты обрабатываются на первом графическом процессоре (GPU0), а другие — на втором (GPU1). Однако при выводе точности обучения и тестирования мы видим повторяющиеся строки. Каждый процесс (другими словами, каждый GPU) выводит точность тестирования независимо. DDP копирует модель на каждый графический процессор, и каждый процесс работает независимо, поэтому если в вашем цикле тестирования есть оператор `print`, то каждый процесс будет выполнять его, что приведет к дублированию строк вывода. Если вас это беспокоит, то можете исправить ситуацию, используя ранг каждого процесса для управления операторами `print`:

```
if rank == 0:
    print("Test accuracy: ", accuracy)  ← Только выводит в первом процессе
```

Вот, в общих чертах, как работает распределенное обучение с помощью DDP. Если вас интересуют дополнительные подробности, то рекомендую ознакомиться с официальной документацией по API по адресу <https://mng.bz/9dPr>.

Альтернативные API PyTorch для обучения с использованием нескольких графических процессоров

Если вы предпочитаете более простой способ использования нескольких GPU в PyTorch, то можете рассмотреть дополнительные API, такие как библиотека Fabric с открытым исходным кодом. Я писал об этом в статье *Accelerating PyTorch Model Training: Using Mixed-Precision and Fully Sharded Data Parallelism* (<https://mng.bz/jXle>).

Итоги приложения

- PyTorch — библиотека с открытым исходным кодом, состоящая из трех основных компонентов: библиотеки тензоров, механизма автоматического дифференцирования и служебных функций для глубокого обучения.
- Библиотека тензоров PyTorch похожа на библиотеки массивов, такие как NumPy.
- В контексте PyTorch тензоры — это структуры данных, похожие на массивы, представляющие скаляры, векторы, матрицы и многомерные массивы.
- Тензорные вычисления в PyTorch могут выполняться на обычном процессоре, но одним из основных преимуществ тензорного формата PyTorch является поддержка графического процессора, позволяющего ускорить вычисления.
- Возможности автоматического дифференцирования (автоградиента) в PyTorch позволяют обучать нейронные сети с помощью обратного распространения ошибки, не вычисляя градиенты вручную.
- Утилиты для глубокого обучения в PyTorch предоставляют строительные блоки, позволяющие создавать специализированные глубокие нейронные сети.
- PyTorch содержит классы `Dataset` и `DataLoader`, с помощью которых можно настраивать эффективную работу конвейеров загрузки данных.
- Проще всего обучать модели на центральном процессоре или одном графическом.
- Использование `DistributedDataParallel` — самый простой способ ускорить обучение в PyTorch, если доступно несколько графических процессоров.

Приложение Б

Ссылки и дополнительные источники

Глава 1

Специализированные LLM способны превзойти универсальные, как показала команда Bloomberg на примере версии GPT, предварительно обученной на финансовых данных с нуля. Специализированная модель превзошла ChatGPT в решении финансовых задач, сохранив при этом хорошую производительность в целом.

- *Wu et al.* BloombergGPT: A Large Language Model for Finance (2023) <https://arxiv.org/abs/2303.17564>.

Кроме того, существующие LLM можно адаптировать и точно настроить так, чтобы они превосходили обычные модели, как показали команды Google Research и Google DeepMind в медицинском контексте:

- *Singhal et al.* Towards Expert-Level Medical Question Answering with Large Language Models (2023) <https://arxiv.org/abs/2305.09617>.

В следующей статье представлена первоначальная архитектура трансформера:

- *Vaswani et al.* Attention Is All You Need (2017) <https://arxiv.org/abs/1706.03762>.

Подробнее о первоначальном трансформере, называемом BERT, см. здесь:

- *Devlin et al.* BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2018) <https://arxiv.org/abs/1810.04805>.

Статья, описывающая модель GPT-3 с одним декодировщиком, которая вдохновила на создание современных больших языковых моделей и используется в качестве шаблона для реализации LLM с нуля в этой книге:

- *Brown et al.* Language Models are Few-Shot Learners (2020) <https://arxiv.org/abs/2005.14165>.

По ссылке ниже представлен первый трансформер для классификации изображений, который показывает, что трансформеры не ограничены работой только с текстовыми данными:

- *Dosovitskiy et al.* An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (2020) <https://arxiv.org/abs/2010.11929>.

Экспериментальные (но менее популярные) архитектуры LLM, представленные по ссылкам ниже, доказывают, что не все модели должны быть основаны на архитектуре трансформеров:

- *Peng et al.* RWKV: Reinventing RNNs for the Transformer Era (2023) <https://arxiv.org/abs/2305.13048>;
- *Poli et al.* Hyena Hierarchy: Towards Larger Convolutional Language Models (2023) <https://arxiv.org/abs/2302.10866>;
- *Gu, Dao.* Mamba: Linear-Time Sequence Modeling with Selective State Spaces (2023) <https://arxiv.org/abs/2312.00752>.

Модель Meta AI — популярная реализация, похожая на GPT и находящаяся в открытом доступе, в отличие от GPT-3 и ChatGPT.

- *Touvron et al.* Llama 2: Open Foundation and Fine-Tuned Chat Models (2023) <https://arxiv.org/abs/2307.09288>.

Для читателей, заинтересованных в дополнительных сведениях о ссылках на наборы данных в разделе 1.5, в следующей статье описывается общедоступный набор данных The Pile, созданный Eleuther AI:

- *Gao et al.* The Pile: An 800GB Dataset of Diverse Text for Language Modeling (2020) <https://arxiv.org/abs/2101.00027>.

В следующей статье представлена ссылка на InstructGPT для тонкой настройки GPT-3, которая была упомянута в разделе 1.6 и более подробно рассмотрена в главе 7:

- *Ouyang et al.* Training Language Models to Follow Instructions with Human Feedback (2022) <https://arxiv.org/abs/2203.02155>.

Глава 2

Читатели, которым интересно обсуждение и сравнение пространств вложения с латентными пространствами и общим понятием векторных представлений, могут найти больше информации в первой главе моей книги:

- *Raschka S.* Machine Learning Q and AI (2023) <https://leanpub.com/machine-learning-q-and-ai>.

В следующей статье более подробно рассматривается использование кодирования пар байтов в качестве метода токенизации:

- *Sennrich et al.* Neural Machine Translation of Rare Words with Subword Units (2015) <https://arxiv.org/abs/1508.07909>.

Код токенизатора для кодирования пар байтов, используемый для обучения GPT-2, был опубликован компанией OpenAI:

- <https://github.com/openai/gpt-2/blob/master/src/encoder.py>

OpenAI предоставляет интерактивный веб-интерфейс для демонстрации работы токенизатора пар байтов в моделях GPT:

- <https://platform.openai.com/tokenizer>.

Для читателей, заинтересованных в написании и обучении токенизатора BPE с нуля, в репозитории `minbpe` на GitHub от Андрея Карпатого (Andrej Karpathy) представлена минимальная и читаемая реализация:

- A Minimal Implementation of a BPE Tokenizer, <https://github.com/karpathy/minbpe>.

Читатели, заинтересованные в изучении альтернативных схем токенизации, которые используются некоторыми другими популярными LLM, могут найти дополнительную информацию в статьях SentencePiece и WordPiece:

- *Kudo, Richardson.* SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing (2018) <https://aclanthology.org/D18-2012/>;
- *Song et al.* Fast WordPiece Tokenization (2020) <https://arxiv.org/abs/2012.15524>.

Глава 3

Читатели, заинтересованные в том, чтобы узнать больше информации о внимании по Богданову для рекуррентных нейронных сетей и языкового перевода, могут найти подробную информацию в следующей статье:

- *Bahdanau, Cho, Bengio.* Neural Machine Translation by Jointly Learning to Align and Translate (2014) <https://arxiv.org/abs/1409.0473>.

Концепция самовнимания как масштабированного точечного произведения была представлена в следующей статье о трансформерах:

- *Vaswani et al.* Attention Is All You Need (2017) <https://arxiv.org/abs/1706.03762>.

FlashAttention — высокоэффективная реализация механизма самовнимания, которая ускоряет процесс вычислений за счет оптимизации шаблонов доступа к памяти. Математически FlashAttention идентичен стандартному механизму самовнимания, но оптимизирует процесс вычислений, повышая его эффективность:

- *Dao et al.* FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness (2022) <https://arxiv.org/abs/2205.14135>;
- *Dao.* FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning (2023) <https://arxiv.org/abs/2307.08691>.

PyTorch реализует функцию для само- и причинно-следственного внимания, которая поддерживает FlashAttention для повышения эффективности. Эта функция находится на стадии бета-тестирования и может быть позднее изменена:

- документация по `scaled_dot_product_attention`: <https://mng.bz/NRJd>.

PyTorch также реализует эффективный класс `MultiHeadAttention` на основе функции `scaled_dot_product`:

- документация по `MultiHeadAttention`: <https://mng.bz/DdJV>.

Отсев (Dropout) — это метод регуляризации, используемый в нейронных сетях для предотвращения переобучения путем случайного удаления элементов (вместе с их связями) из нейронной сети во время обучения:

- *Srivastava et al.* Dropout: A Simple Way to Prevent Neural Networks from Overfitting (2014) <https://jmlr.org/papers/v15/srivastava14a.html>.

В то время как использование многоцелевого внимания на основе масштабированного точечного произведения остается наиболее распространенным вариантом самовнимания на практике, авторы следующей статьи обнаружили, что можно добиться хорошей производительности и без матрицы весовых коэффициентов и проекционного слоя:

- *He, Hofmann.* Simplifying Transformer Blocks (2023) <https://arxiv.org/abs/2311.01906>.

Глава 4

В следующей статье представлена методика, которая стабилизирует динамику скрытого состояния нейронных сетей, нормализуя суммарные входные данные

для нейронов в скрытом слое, что значительно сокращает время обучения по сравнению с ранее опубликованными методами:

- *Ba, Kiros, Hinton*. Layer Normalization (2016) <https://arxiv.org/abs/1607.06450>.

Post-LayerNorm, используемая в первоначальной модели трансформера, применяет нормализацию слоя после самовнимания и сетей прямого распространения. В отличие от нее, Pre-LayerNorm, используемая в таких моделях, как GPT-2, и более новых LLM, применяет нормализацию слоя перед этими компонентами, что может привести к более стабильной динамике обучения и в некоторых случаях, как показано в следующих статьях, повысить производительность:

- *Xiong et al.* On Layer Normalization in the Transformer Architecture (2020) <https://arxiv.org/abs/2002.04745>;
- *Tie et al.* ResiDual: Transformer with Dual Residual Connections (2023) <https://arxiv.org/abs/2304.14802>.

Популярный вариант LayerNorm, используемый в современных LLM по причине его повышенной вычислительной эффективности, — RMSNorm. Он упрощает процесс нормализации, нормализуя входные данные с помощью только среднего квадратичного отклонения входных данных, без вычитания среднего значения перед возведением в квадрат. Это означает, что данные не центрируются перед вычислением масштабирующего коэффициента. RMSNorm более подробно описан в следующей статье:

- *Zhang, Sennrich*. Root Mean Square Layer Normalization (2019) <https://arxiv.org/abs/1910.07467>.

Функция активации GELU (Gaussian Error Linear Unit) сочетает в себе свойства классической функции активации ReLU и нормальной кумулятивной функции распределения, предназначенных для моделирования выходных данных слоя, обеспечивая стохастическую регуляризацию и нелинейности в моделях глубокого обучения:

- *Hendricks, Gimpel*. Gaussian Error Linear Units (GELUs) (2016) <https://arxiv.org/abs/1606.08415>.

В статье о GPT-2 представлена серия больших языковых моделей на основе трансформеров с разным количеством параметров — 124, 355, 774 млн и 1,5 млрд:

- *Radford et al.* Language Models Are Unsupervised Multitask Learners (2019) https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

GPT-3 от OpenAI использует принципиально ту же архитектуру, что и GPT-2, за исключением того, что самая большая версия (175 млрд) в 100 раз больше, чем

самая большая модель GPT-2, и была обучена на гораздо большем объеме данных. Заинтересованные читатели могут обратиться к официальной статье о GPT-3 от OpenAI и техническому обзору от Lambda Labs, в котором подсчитано, что обучение GPT-3 на одном графическом процессоре RTX 8000 займет 665 лет:

- *Brown et al.* Language Models are Few-Shot Learners (2023) <https://arxiv.org/abs/2005.14165>;
- OpenAI's GPT-3 Language Model: A Technical Overview, <https://lambdalabs.com/blog/demystifying-gpt-3>.

NanoGPT — это репозиторий кода с минималистичной, но эффективной реализацией модели GPT-2, аналогичной модели, представленной в этой книге. Хотя код в этой книге отличается от nanoGPT, данный репозиторий вдохновил на реорганизацию большого родительского класса GPT Python в виде более мелких подмодулей:

- NanoGPT, a Repository for Training Medium-Sized GPTs, <https://github.com/karpathy/nanoGPT>.

Информативный блог, показывающий, что большая часть вычислений в LLM выполняется на слоях прямой передачи, а не на слоях внимания, когда размер контекста меньше 32 000 токенов, можно найти здесь:

- *Harm de Vries*. In the Long (Context) Run, <https://www.harmdevries.com/post/context-length/>.

Глава 5

Подробную информацию о функции потерь и применении логарифмического преобразования для упрощения математической оптимизации см. в моем видео с лекцией:

- L8.2 Logistic Regression Loss Function, <https://www.youtube.com/watch?v=GxJe0DZvydM>.

В следующих лекции и примере кода объясняется, как работает функция перекрестной энтропии в PyTorch:

- L8.7.1 OneHot Encoding and Multi-category Cross Entropy, <https://www.youtube.com/watch?v=4n71-tZ94yk>;
- Understanding Onehot Encoding and Cross Entropy in PyTorch, <https://mng.bz/o05v>.

В следующих двух статьях подробно описаны набор данных, гиперпараметры и архитектура, используемые для предварительной подготовки LLM:

- *Biderman et al.* Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling (2023) <https://arxiv.org/abs/2304.01373>;

- *Groeneveld et al.* OLMo: Accelerating the Science of Language Models (2024) <https://arxiv.org/abs/2402.00838>.

Код, представленный по ссылке ниже, содержит инструкции по подготовке 60 000 общедоступных книг от проекта Gutenberg для обучения LLM:

- Pretraining GPT on the Project Gutenberg Dataset, <https://mng.bz/Bdw2>.

В главе 5 обсуждается предварительная подготовка LLM, а в приложении Г рассматриваются более сложные функции обучения, такие как линейный разогрев и косинусный отжиг. В следующей статье показано, что аналогичные методы можно успешно применять для продолжения предварительной подготовки уже обученных LLM, а также даны дополнительные советы и рекомендации:

- *Ibrahim et al.* Simple and Scalable Strategies to Continually Pre-train Large Language Models (2024) <https://arxiv.org/abs/2403.08763>.

BloombergGPT — пример специализированной языковой модели, созданной на основе как общих, так и частных текстовых корпусов, например в области финансов:

- *Wu et al.* BloombergGPT: A Large Language Model for Finance (2023) <https://arxiv.org/abs/2303.17564>.

GaLore — недавний исследовательский проект, целью которого является повышение эффективности предварительного обучения LLM. Необходимое изменение кода сводится к простой замене оптимизатора PyTorch AdamW в функции обучения на оптимизатор GaLoreAdamW, предоставляемый пакетом Python `galore-torch`:

- *Zhao et al.* GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection (2024) <https://arxiv.org/abs/2403.03507>;
- репозиторий кода GaLore, <https://github.com/jiaweizzhao/GaLore>.

В статьях и на ресурсах, доступных по ссылкам в перечне ниже, представлены общедоступные крупномасштабные наборы данных для предварительной подготовки LLM, которые состоят из сотен гигабайтов и терабайтов текстовых данных:

- *Soldaini et al.* Dolma: An Open Corpus of Three Trillion Tokens for LLM Pre-training Research (2024) <https://arxiv.org/abs/2402.00159>;
- *Gao et al.* The Pile: An 800GB Dataset of Diverse Text for Language Modeling (2020) <https://arxiv.org/abs/2101.00027>;
- *Penedo et al.* The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data and Web Data Only (2023) <https://arxiv.org/abs/2306.01116>;
- RedPajama, <https://mng.bz/d6nw>;

- датасет FineWeb Dataset, который включает в себя более 15 триллионов токенов очищенных и дедуплицированных англоязычных веб-данных, полученных из CommonCrawl, <https://mng.bz/rVzy>.

Статья, в которой впервые была представлена выборка top-k:

- *Fan et al.* Hierarchical Neural Story Generation (2018) <https://arxiv.org/abs/1805.04833>.

Альтернатива выборке top-k — выборка top-p (не рассматривается в главе 5), при которой выбирается наименьший набор токенов, совокупная вероятность которых превышает пороговое значение p , в то время как при выборке top-k выбираются первые k токенов по вероятности:

- Top-p sampling, https://en.wikipedia.org/wiki/Top-p_sampling.

Лучевой поиск (Beam search) (не рассматривается в главе 5) — альтернативный алгоритм декодирования, который генерирует выходные последовательности, сохраняя на каждом этапе только наиболее эффективные частичные последовательности, чтобы сбалансировать эффективность и качество:

- *Vijayakumar et al.* Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models (2016) <https://arxiv.org/abs/1610.02424>.

Глава 6

Дополнительные ресурсы, в которых рассматриваются различные типы тонкой настройки:

- Using and Finetuning Pretrained Transformers, <https://mng.bz/VxJG>;
- Finetuning Large Language Models, <https://mng.bz/x28X>.

Описание дополнительных экспериментов, в том числе сравнение тонкой настройки первого и последнего выходных токенов, можно найти в дополнительных материалах на GitHub:

- Additional spam classification experiments, <https://mng.bz/AdJx>.

Решить задачу бинарной классификации, такой как классификация спама, технически можно с помощью только одного выходного узла вместо двух, как я описываю в следующей статье:

- Losses Learned — Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch, <https://mng.bz/ZEJA>.

Описание дополнительных экспериментов по тонкой настройке различных слоев LLM можно найти в следующей статье, которая показывает, что тонкая

настройка последнего блока трансформера в дополнение к выходному слою существенно повышает точность прогнозирования:

- Finetuning Large Language Models, <https://mng.bz/RZJv>.

Дополнительные ресурсы и информацию по работе с несбалансированными наборами данных для классификации можно найти в документации `imbalanced-learn`:

- Imbalanced-Learn User Guide, <https://mng.bz/2KNa>.

По ссылке ниже читатели, заинтересованные в классификации спама в письмах электронной почты, а не в текстовых сообщениях, могут найти большой набор данных для классификации в удобном формате CSV, аналогичном формату набора данных, использованному в главе 6:

- Email Spam Classification Dataset, <https://mng.bz/1GEq>.

GPT-2 — это модель, основанная на модуле декодирования архитектуры трансформеров, и ее основное назначение — генерировать новый текст. В качестве альтернативы модели на основе кодировщика, такие как BERT и RoBERTa, могут эффективно решать задачи классификации:

- *Devlin et al.* BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2018) <https://arxiv.org/abs/1810.04805>;
- *Liu et al.* RoBERTa: A Robustly Optimized BERT Pretraining Approach (2019) <https://arxiv.org/abs/1907.11692>;
- Additional Experiments Classifying the Sentiment of 50k IMDB Movie Reviews, <https://mng.bz/PZJR>.

Последние исследования показывают, что эффективность классификации можно дополнительно повысить, удалив причинно-следственную маску во время тонкой настройки по классификации наряду с другими изменениями:

- *Li et al.* Label Supervised LLaMA Finetuning (2023) <https://arxiv.org/abs/2310.01208>;
- *Behnam Ghader et al.* LLM2Vec: Large Language Models Are Secretly Powerful Text Encoders (2024) <https://arxiv.org/abs/2404.05961>.

Глава 7

Набор данных Alpaca для тонкой настройки по инструкциям содержит 52 000 пар «инструкция — ответ» и является одним из первых и наиболее популярных общедоступных наборов данных для тонкой настройки по инструкциям:

- Stanford Alpaca: An Instruction-Following Llama Model, https://github.com/tatsu-lab/stanford_alpaca.

Дополнительные общедоступные наборы данных, подходящие для тонкой настройки по инструкциям, можно найти по ссылкам ниже:

- LIMA, <https://huggingface.co/datasets/GAIR/lima>;
 - см. также: *Zhou et al.* LIMA: Less Is More for Alignment, <https://arxiv.org/abs/2305.11206>;
- UltraChat, <https://huggingface.co/datasets/openchat/ultrachat-sharegpt>;
 - большой набор данных, состоящий из 805 000 пар «инструкция — ответ»; см. также: *Ding et al.* Enhancing Chat Language Models by Scaling High quality Instructional Conversations, <https://arxiv.org/abs/2305.14233>;
- Alpaca GPT4, <https://mng.bz/Aa0p>;
 - похожий на Alpaca набор данных с 52 000 парами «инструкция-ответ», сгенерированный GPT-4 вместо GPT-3.5.

Phi-3 — модель с 3,8 млрд параметров и вариантом с тонкой настройкой по инструкциям, которая, как сообщается в следующей статье, сопоставима с гораздо более крупными закрытыми моделями, такими как GPT-3.5:

- *Abdin et al.* Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone (2024) <https://arxiv.org/abs/2404.14219>.

Исследователи предлагают метод синтеза инструкций, который генерирует 300 000 высококачественных пар «инструкция — ответ» на основе модели Llama 3 с тонкой настройкой по инструкциям. Базовая модель Llama 3 с предварительной настройкой на эти примеры инструкций работает так же хорошо, как и исходная модель Llama-3 с тонкой настройкой по инструкциям.

- *Xu et al.* Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing (2024) <https://arxiv.org/abs/2406.08464>.

Исследования показали, что отсутствие маскировки инструкций и входных данных при тонкой настройке эффективно повышает производительность в различных задачах, связанных с естественным языком, особенно при обучении на наборах данных с длинными инструкциями и краткими ответами или при использовании небольшого количества обучающих примеров:

- *Shi.* Instruction Tuning with Loss Over Instructions (2024) <https://arxiv.org/abs/2405.14394>.

Prometheus и PHUDGE — общедоступные большие языковые модели, которые, как и GPT-4, оценивают длинные ответы по настраиваемым критериям (в этой книге они не используются, поскольку на момент ее написания они не поддерживаются Олламой и, следовательно, не могут эффективно работать на ноутбуке):

- *Kim et al.* Prometheus: Inducing Finegrained Evaluation Capability in Language Models (2023) <https://arxiv.org/abs/2310.08491>;
- *Deshwal, Chawla.* PHUDGE: Phi-3 as Scalable Judge (2024) <https://arxiv.org/abs/2405.08029>;
- *Kim et al.* Prometheus 2: An Open Source Language Model Specialized in Evaluating Other Language Models (2024) <https://arxiv.org/abs/2405.01535>.

Результаты, представленные в следующем отчете, подтверждают мнение о том, что большие языковые модели в основном приобретают фактические знания во время предварительного обучения, а тонкая настройка, как правило, повышает эффективность использования ими этих знаний. Кроме того, в данном исследовании рассматривается, как тонкая настройка больших языковых моделей с помощью новой фактической информации влияет на их способность применять уже имеющиеся знания. Выяснилось, что LLM медленнее усваивают новые факты, а их введение во время настройки повышает склонность модели генерировать неверную информацию.

- *Gekhman.* Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations? (2024) <https://arxiv.org/abs/2405.05904>.

Тонкая настройка по предпочтениям — необязательный этап после тонкой настройки по инструкциям, который позволяет более точно согласовать LLM с предпочтениями человека. В следующих моих статьях содержится дополнительная информация об этом процессе:

- LLM Training: RLHF and Its Alternatives, <https://mng.bz/ZVPm>;
- Tips for LLM Pretraining and Evaluating Reward Models, <https://mng.bz/RNXj>.

Приложение А

Приложение А должно быть достаточно, чтобы познакомить вас с глубоким обучением. Но если вы ищете более подробное введение в эту тему, то я рекомендую следующие книги:

- *Raschka S., Hayden L., Mirjalili V.* Machine Learning with PyTorch and Scikit-Learn¹ (2022);
- *Stevens E., Antiga L., Viehmann T.* Deep Learning with PyTorch² (2021).

¹ Хэйден Л. Ю., Мирджалили В., Рашка С. Машинное обучение с PyTorch и Scikit-Learn. — СПб.: Питер, 2024.

² Стивенс Э., Антига Л., Виман Т. PyTorch. Освещающая глубокое обучение. — СПб.: Питер, 2022.

Для более подробного ознакомления с концепцией тензоров читатели могут посмотреть мое 15-минутное видео:

- Lecture 4.1: Tensors in Deep Learning, <https://www.youtube.com/watch?v=JXfDIgrfOBY>.

Если вы хотите узнать больше об оценке моделей в машинном обучении, то я рекомендую эту мою статью:

- *Raschka S.* Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning (2018) <https://arxiv.org/abs/1811.12808>.

Читатели, которым интересно освежить в памяти или вкратце ознакомиться с математическим анализом, могут найти мой материал по этой теме, который находится в свободном доступе на моем сайте:

- *Raschka S.* Introduction to Calculus, <https://mng.bz/WEyW>.

Почему PyTorch не вызывает `optimizer.zero_grad()` автоматически в фоновом режиме? В некоторых случаях может быть желательно накапливать градиенты, и PyTorch предоставляет нам такую возможность. Если вы хотите узнать больше о накоплении градиентов, то ознакомьтесь со следующей статьей:

- *Raschka S.* Finetuning Large Language Models on a Single GPU Using Gradient Accumulation, <https://mng.bz/8wPD>.

DDP — популярный подход к обучению моделей глубокого обучения с использованием нескольких графических процессоров. Для более сложных случаев, когда одна модель не помещается на GPU, можно задействовать метод PyTorch Fully Sharded Data Parallel (FSDP), который обеспечивает распределенный параллелизм данных и распределяет большие слои по разным графическим процессорам. Дополнительную информацию можно найти в этом обзоре, содержащем ссылки на документацию по API:

- Introducing PyTorch Fully Sharded Data Parallel (FSDP) API, <https://mng.bz/EZJR>.

Приложение В

Решения упражнений

Полные примеры кода для ответов на упражнения можно найти в репозитории GitHub по адресу <https://github.com/rasbt/LLMs-from-scratch>.

Глава 2

Упражнение 2.1

Вы можете получить отдельные идентификаторы токенов, передавая кодировщику по одной строке за раз:

```
print(tokenizer.encode("Ak"))
print(tokenizer.encode("w"))
# ...
```

Вывод выглядит так:

```
[33901]
[86]
# ...
```

Затем вы можете использовать следующий код для сборки исходной строки:

```
print(tokenizer.decode([33901, 86, 343, 86, 220, 959]))
```

Он возвращает следующий результат:

```
'Akwirw ier'
```

Упражнение 2.2

Код загрузчика данных с `max_length=2` и `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=2, stride=2
)
```

348 Приложение В. Решения упражнений

Он генерирует пакеты в следующем формате:

```
tensor([[ 40, 367],
        [2885, 1464],
        [1807, 3619],
        [ 402, 271]])
```

Код второго загрузчика данных с `max_length=8` и `stride=2`:

```
dataloader = create_dataloader(
    raw_text, batch_size=4, max_length=8, stride=2
)
```

Пример пакета выглядит так:

```
tensor([[ 40, 367, 2885, 1464, 1807, 3619, 402, 271],
        [ 2885, 1464, 1807, 3619, 402, 271, 10899, 2138],
        [ 1807, 3619, 402, 271, 10899, 2138, 257, 7026],
        [ 402, 271, 10899, 2138, 257, 7026, 15632, 438]])
```

Глава 3

Упражнение 3.1

Правильное присваивание значений весам:

```
sa_v1.W_query = torch.nn.Parameter(sa_v2.W_query.weight.T)
sa_v1.W_key = torch.nn.Parameter(sa_v2.W_key.weight.T)
sa_v1.W_value = torch.nn.Parameter(sa_v2.W_value.weight.T)
```

Упражнение 3.2

Чтобы получить выходную размерность 2, как в случае с одноцелевым вниманием, нам нужно изменить проекционную размерность `d_out` на 1:

```
d_out = 1
mha = MultiHeadAttentionWrapper(d_in, d_out, block_size, 0.0, num_heads=2)
```

Упражнение 3.3

Инициализация для самой маленькой модели GPT-2:

```
block_size = 1024
d_in, d_out = 768, 768
num_heads = 12
mha = MultiHeadAttention(d_in, d_out, block_size, 0.0, num_heads)
```

Глава 4

Упражнение 4.1

Мы можем определить количество параметров в модуле прямого распространения и в модуле внимания следующим образом:


```

block = TransformerBlock(GPT_CONFIG_124M)

total_params = sum(p.numel() for p in block.ff.parameters())
print(f"Total number of parameters in feed forward module: {total_params:,}")

total_params = sum(p.numel() for p in block.att.parameters())
print(f"Total number of parameters in attention module: {total_params:,}")

```

Как мы видим, модуль прямого распространения содержит примерно в два раза больше параметров, чем модуль внимания:

```

Total number of parameters in feed forward module: 4 722 432
Total number of parameters in attention module: 2 360 064

```

Упражнение 4.2

Чтобы создать экземпляры других моделей GPT, можно изменить словарь конфигурации следующим образом (здесь показано для GPT-2 XL):

```

GPT_CONFIG = GPT_CONFIG_124M.copy()
GPT_CONFIG["emb_dim"] = 1600
GPT_CONFIG["n_layers"] = 48
GPT_CONFIG["n_heads"] = 25
model = GPTModel(GPT_CONFIG)

```

Затем, используя код из раздела 4.6 для расчета количества параметров и требований к оперативной памяти, мы получаем:

```

gpt2-xl:
Total number of parameters: 1 637 792 000
Number of trainable parameters considering weight tying: 1 557 380 800
Total size of the model: 6247.68 МБ

```

Упражнение 4.3

В главе 4 есть три разных места, где мы использовали слои с отсевом: слой вложения, слой коротких соединений и модуль многоцелевого внимания. Мы можем контролировать коэффициенты отсеивания для каждого из слоев, кодируя их отдельно в файле конфигурации, а затем соответствующим образом изменяя реализацию кода.

Измененная конфигурация выглядит так:

```

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 1024,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate_attn": 0.1,
    "drop_rate_shortcut": 0.1,
    "drop_rate_emb": 0.1,
    "qkv_bias": False
}

```

Отсев для многоцелевого
внимания

← Отсев для коротких соединений

← Отсев для слоя
вложения

Измененные TransformerBlock и GPTModel выглядят так:

```
class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate_attn"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(
            cfg["drop_rate_shortcut"])

    def forward(self, x):
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut

        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut
        return x

class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate_emb"])
        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])]

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False)

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
```

Отсев для многоцелевого внимания

Отсев для коротких соединений

Отсев для слоя вложения

```

tok_embeddings = self.tok_emb(in_idx)
pos_embeddings = self.pos_emb(
    torch.arange(seq_len, device=in_idx.device)
)
x = tok_embeddings + pos_embeddings
x = self.drop_emb(x)
x = self.trf_blocks(x)
x = self.final_norm(x)
logits = self.out_head(x)
return logits

```

Глава 5

Упражнение 5.1

Мы можем вывести, сколько раз выбирается токен (или слово) *pizza*, используя функцию `print_sampled_tokens`, которую определили в разделе 5.1. Начнем с кода, который мы определили в подразделе 5.3.1.

Токен *pizza* выбирается 0 раз, если температура равна 0 или 0.1, и 32 раза, если она увеличена до 5. Предполагаемая вероятность составляет $32 / 1000 \times 100 \% = 3,2 \%$.

Фактическая вероятность составляет 4,3 % и содержится в тензоре смасштабированной `softmax` вероятности (`scaled_probas[2][6]`).

Упражнение 5.2

Выборка `top-k` и масштабирование температуры — настройки, которые необходимо корректировать в зависимости от LLM и желаемой степени разнообразия и случайности выходных данных модели.

При использовании относительно небольших значений `top-k` (например, меньше 10) и при температуре ниже 1 выходные данные модели становятся менее случайными и более детерминированными. Эта настройка полезна, когда нужно, чтобы сгенерированный текст был более предсказуемым, связным и приближенным к наиболее вероятным результатам на основе обучающих данных.

Такие низкие значения `k` и температуры используются для создания официальных документов или отчетов, где важнее всего ясность и точность. Другие примеры применения содержат задачи технического анализа или генерации кода, где точность имеет решающее значение. Кроме того, ответы на вопросы и образовательный контент требуют точных ответов, что достигается при температуре ниже 1.

В свою очередь, большие значения `top-k` (например, значения в диапазоне от 20 до 40) и температуры выше 1 полезны при использовании LLM для мозговых штурмов или создания креативного контента, например художественной литературы.

Упражнение 5.3

Существует несколько способов добиться детерминированного поведения с помощью функции `generate`.

1. Установка `top_k=None` и без масштабирования температуры.
2. Установка `top_k=1`.

Упражнение 5.4

По сути, вам нужно загрузить модель и оптимизатор, которые мы сохранили в основной главе:

```
checkpoint = torch.load("model_and_optimizer.pth")
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
```

Затем вызовите функцию `train_simple_function` с `num_epochs=1`, чтобы обучить модель еще на одной эпохе.

Упражнение 5.5

Мы можем использовать следующий код для расчета потерь на обучающей и проверочной выборках для модели GPT:

```
train_loss = calc_loss_loader(train_loader, gpt, device)
val_loss = calc_loss_loader(val_loader, gpt, device)
```

В результате потери для модели с 124 млн параметров составляют:

```
Training loss: 3.754748503367106
Validation loss: 3.559617757797241
```

Главное наблюдение заключается в том, что показатели при обучении и проверке находятся примерно на одном уровне. Это может быть объяснено несколькими факторами.

1. Рассказ *The Verdict* не входил в набор данных для предварительного обучения, когда OpenAI обучала модель GPT-2. Таким образом, модель не демонстрирует явного переобучения на обучающей выборке и одинаково хорошо работает на обучающей и проверочной выборках из *The Verdict*. (Потери на проверочной выборке немного ниже, чем на обучающей, что необычно для глубокого обучения. Однако это, вероятно, связано со случайным шумом, поскольку набор данных относительно мал. На практике, если переобучения нет, то потери на обучающей и проверочной выборках должны быть примерно одинаковыми.)

2. Рассказ *The Verdict* был частью обучающей выборки для GPT-2. В этом случае мы не можем сказать, переобучается ли модель на обучающих данных, поскольку проверочная выборка тоже использовалась для обучения. Чтобы оценить степень переобучения, понадобится новый набор данных, созданный после того, как OpenAI завершила обучение GPT-2, чтобы убедиться, что он не мог быть частью предварительного обучения.

Упражнение 5.6

В основной главе мы экспериментировали с самой маленькой моделью GPT-2, которая имеет всего 124 млн параметров. Это было сделано для того, чтобы требований к ресурсам было как можно меньше. Однако вы можете легко экспериментировать с более крупными моделями, внося в код минимальные изменения. Например, чтобы загрузить 1558 млн вместо 124 млн весов модели в главе 5, нужно изменить только две строки кода:

```
hparams, params = download_and_load_gpt2(model_size="124M", models_dir="gpt2")
model_name = "gpt2-small (124M)"
```

Обновленный код выглядит так:

```
hparams, params = download_and_load_gpt2(model_size="1558M", models_dir="gpt2")
model_name = "gpt2-xl (1558M)"
```

Глава 6

Упражнение 6.1

Мы можем дополнить входные данные до максимального количества токенов, поддерживаемых моделью, установив максимальную длину `max_length = 1024` при инициализации наборов данных:

```
train_dataset = SpamDataset(..., max_length=1024, ...)
val_dataset = SpamDataset(..., max_length=1024, ...)
test_dataset = SpamDataset(..., max_length=1024, ...)
```

Однако дополнительное заполнение приводит к значительному снижению точности тестирования до 78,33 % (по сравнению с 95,67 %, указанных в главе 6).

Упражнение 6.2

Вместо тонкой настройки только последнего блока трансформера мы можем выполнить тонкую настройку всей модели, удалив из кода следующие строки:

```
for param in model.parameters():
    param.requires_grad = False
```

Это изменение приводит к повышению точности тестирования на 1 % до 96,67 % (по сравнению с 95,67 %, указанных в главе 6).

Упражнение 6.3

Вместо тонкой настройки последнего выходного токена мы можем выполнить тонкую настройку первого, заменив `model(input_batch)[: , -1, :]` на `model(input_batch)[: , 0, :]` везде в коде.

Как и ожидалось, ввиду того что первый токен содержит меньше информации, чем последний, это изменение приводит к значительному снижению тестовой точности до 75,00 % (по сравнению с 95,67 %, указанных в главе 6).

Глава 7

Упражнение 7.1

Формат запроса Phi-3, показанный на рис. 7.4, выглядит следующим образом для заданного примера ввода:

```
<user>
Identify the correct spelling of the following word: 'Occasion'

<assistant>
The correct spelling is 'Occasion'.
```

Чтобы использовать такой шаблон, мы можем изменить функцию `format_input` следующим образом:

```
def format_input(entry):
    instruction_text = (
        f"<|user|>\n{entry['instruction']}"
    )
    input_text = f"\n{entry['input']}" if entry["input"] else ""
    return instruction_text + input_text
```

Наконец, нам также нужно изменить способ извлечения сгенерированного ответа при сборе тестовых ответов:

```
for i, entry in tqdm(enumerate(test_data), total=len(test_data)):
    input_text = format_input(entry)
    tokenizer=tokenizer
    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer)
    response_text = (
        generated_text[len(input_text):]
        .replace("<|assistant|>:", "")
        .strip()
    )
    test_data[i]["model_response"] = response_text
```

← Новое: Настройте ###Response на <|assistant|>

Тонкая настройка модели с использованием шаблона Phi-3 выполняется примерно на 17 % быстрее, поскольку это приводит к сокращению ее входных данных. Результат близок к 50 баллам, что примерно соответствует результату, который мы получили ранее с подсказками в стиле Alpaca.

Упражнение 7.2

Чтобы замаскировать инструкции, как показано на рис. 7.13, нужно внести небольшие изменения в класс `InstructionDataset` и функцию `custom_collate_fn`. Мы можем изменить этот класс, чтобы собирать данные о длине инструкций, которые будем использовать в функции `collate` для определения позиций инструкций в целях:

```
class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.instruction_lengths = []
        self.encoded_texts = []

        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text

            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )
            instruction_length = (
                len(tokenizer.encode(instruction_plus_input))
            )
            self.instruction_lengths.append(instruction_length)

    def __getitem__(self, index):
        return self.instruction_lengths[index], self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```

Выделяет список для длины инструкций

Добавляет длину инструкций в список

Возвращает как длину инструкций, так и тексты

Далее мы обновляем функцию `custom_collate_fn`, где каждый пакет теперь представляет собой кортеж, содержащий `(instruction_length, item)`, а не просто `item`, из-за изменений в наборе данных `InstructionDataset`. Кроме того, теперь мы маскируем соответствующие токены инструкций в целевом списке идентификаторов:

```
def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
```

```

batch_max_length = max(len(item)+1 for instruction_length, item in batch)
inputs_lst, targets_lst = [], []
for instruction_length, item in batch:
    new_item = item.copy()
    new_item += [pad_token_id]
    padded = (
        new_item + [pad_token_id] * (batch_max_length - len(new_item))
    )
    inputs = torch.tensor(padded[:-1])
    targets = torch.tensor(padded[1:])
    mask = targets == pad_token_id
    indices = torch.nonzero(mask).squeeze()
    if indices.numel() > 1:
        targets[indices[1:]] = ignore_index

    targets[:instruction_length-1] = -100

    if allowed_max_length is not None:
        inputs = inputs[:allowed_max_length]
        targets = targets[:allowed_max_length]

    inputs_lst.append(inputs)
    targets_lst.append(targets)

inputs_tensor = torch.stack(inputs_lst).to(device)
targets_tensor = torch.stack(targets_lst).to(device)

return inputs_tensor, targets_tensor

```

Пакет сейчас кортеж

Маскирует все входные токены и токены инструкций в целях

При оценке модели, дообученной с помощью этого метода маскирования инструкций, она работает немного хуже (примерно на 4 балла по методу Ollama Ллама 3, описанному в главе 7). Это согласуется с наблюдениями, приведенными в статье Чж. Ши и др. *Instruction Tuning With Loss Over Instructions* (<https://arxiv.org/abs/2405.14394>).

Упражнение 7.3

Чтобы дообучить модель на исходном наборе данных Stanford Alpaca (https://github.com/tatsu-lab/stanford_alpaca), просто нужно изменить URL файла с

```
url = https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/main/ch07/01_main-chapter-code/instruction-data.json
```

на

```
url = https://raw.githubusercontent.com/tatsu-lab/stanford_alpaca/main/alpaca_data.json
```

Обратите внимание, что набор содержит 52 000 записей (в 50 раз больше, чем в главе 7), и записи в нем длиннее, чем те, с которыми мы работали в главе 7.

Поэтому настоятельно рекомендуется проводить обучение на графическом процессоре.

Если вы столкнулись с ошибками нехватки памяти, то попробуйте уменьшить размер пакета с 8 до 4, 2 или 1. Помимо уменьшения размера пакета, вы также можете рассмотреть возможность уменьшения `allowed_max_length` с 1024 до 512 или 256.

Упражнение 7.4

Чтобы выполнить тонкую настройку модели с помощью LoRA, используйте соответствующие классы и функции из приложения Д:

```
from appendix_E import LoRALayer, LinearWithLoRA, replace_linear_with_lora
```

Затем добавьте следующие строки кода под кодом загрузки модели, приведенным в разделе 7.5:

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")
```

```
for param in model.parameters():
    param.requires_grad = False
```

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
replace_linear_with_lora(model, rank=16, alpha=16)
```

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
model.to(device)
```

Обратите внимание, что на графическом процессоре Nvidia L4 тонкая настройка с помощью LoRA (low-rank adaptation, низкоранговая адаптация) занимает 1,30 минуты. На том же процессоре исходный код выполняется за 1,80 минуты. Таким образом, в этом случае LoRA примерно на 28 % быстрее. Оценка, рассчитанная с помощью метода Ollama Llama 3, описанного в главе 7, составляет около 50 баллов, что примерно соответствует исходной модели.

Приложение А

Упражнение А.3

Сеть имеет два входа и два выхода. Кроме того, в ней есть два скрытых слоя с 30 и 20 узлами соответственно. Мы можем вычислить количество параметров следующим образом:

```
model = NeuralNetwork(2, 2)
num_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print("Total number of trainable model parameters:", num_params)
```

Код возвращает следующий результат:

752

Мы также можем вычислить этот показатель вручную:

- *первый скрытый слой* — 2 входа, умноженные на 30 скрытых нейронов, плюс 30 элементов смещения;
- *второй скрытый слой* — 30 входов, умноженные на 20 нейронов, плюс 20 элементов смещения;
- *выходной слой* — 20 входов, умноженные на 2 выхода, плюс 2 элемента смещения.

Затем, сложив все параметры в каждом слое, получаем $2 \times 30 + 30 + 30 \times 20 + 20 + 20 \times 2 + 2 = 752$.

Упражнение А.4

Точные результаты времени выполнения будут зависеть от оборудования, используемого для этого эксперимента. В моих экспериментах я наблюдал значительное ускорение даже для небольших перемножений матриц, таких как следующее, при использовании экземпляра Google Colab, подключенного к графическому процессору V100:

```
a = torch.rand(100, 200)
b = torch.rand(200, 300)
%timeit a@b
```

На центральном процессоре это заняло

63.8 мкс ± 8.7 мкс на цикл

При выполнении на графическом процессоре

```
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

результат составил

13.8 мкс ± 425 нс на цикл

В этом случае на V100 вычисления выполнялись примерно в четыре раза быстрее.

Приложение Г

Добавление новых возможностей в процесс обучения

В этом приложении мы улучшим функцию обучения для процессов предварительной и тонкой настройки, описанных в главах 5–7. В частности, мы рассмотрим изменение *скорости обучения*, *затухание по косинусу* и *ограничение градиента*. Затем мы добавим эти методы в функцию обучения и предварительно обучим LLM.

Чтобы сделать код самодостаточным, мы повторно инициализируем модель, которую обучали в главе 5:

```
import torch from chapter04
import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 256,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,
    "qkv_bias": False
}

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
model.eval()
```

Размер словаря

Длина укороченного контекста (первоначально: 1024)

Размерность вложения

Количество целей внимания

Количество слоев

Коэффициент отсева

Смещение запроса-ключа-значения

После инициализации модели нужно инициализировать загрузчики данных. Сначала мы загружаем рассказ *The Verdict*:

```
import os
import urllib.request

file_path = "the-verdict.txt"
```

```

url = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/"
    "main/ch02/01_main-chapter-code/the-verdict.txt"
)

if not os.path.exists(file_path):
    with urllib.request.urlopen(url) as response:
        text_data = response.read().decode('utf-8')
    with open(file_path, "w", encoding="utf-8") as file:
        file.write(text_data)
else:
    with open(file_path, "r", encoding="utf-8") as file:
        text_data = file.read()

```

Далее мы загружаем `text_data` в загрузчики данных:

```

from previous_chapters import create_dataloader_v1

train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
torch.manual_seed(123)
train_loader = create_dataloader_v1(
    text_data[:split_idx],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    text_data[split_idx:],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)

```

Г.1. Переменная скорость обучения

Переменная скорость обучения может стабилизировать процесс обучения сложных моделей, таких как LLM. В него входит постепенное увеличение скорости обучения от очень низкого начального значения (`initial_lr`) до максимального, заданного пользователем (`peak_lr`). Начало обучения с небольшими обновлениями весов снижает риск того, что модель столкнется с большими дестабилизирующими обновлениями на этапе обучения.

Предположим, мы планируем обучать LLM в течение 15 эпох, взяв первоначальную скорость обучения 0,0001 и увеличивая ее до максимальной — 0,01:

```
n_epochs = 15
initial_lr = 0.0001
peak_lr = 0.01
warmup_steps = 20
```

Количество шагов прогрева (warmup) обычно составляет от 0,1 до 20 % от общего количества шагов, которое можно рассчитать следующим образом:

```
total_steps = len(train_loader) * n_epochs
warmup_steps = int(0.2 * total_steps)  ← 20 % прогрева
print(warmup_steps)
```

Выводится число 27, означающее, что у нас есть 20 шагов для увеличения начальной скорости обучения с 0,0001 до 0,01 в течение первых 27 шагов обучения.

Далее мы реализуем простой шаблон цикла обучения, чтобы вы могли увидеть этот процесс:

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
lr_increment = (peak_lr - initial_lr) / warmup_steps  ← Это увеличение определяется тем, насколько мы повышаем initial_lr на каждом из 20 шагов прогрева

global_step = -1
track_lrs = []

for epoch in range(n_epochs):  ← Выполняет типичный цикл обучения, перебирая пакеты данных в каждой эпохе
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:  ← Обновляет скорость обучения, если мы еще находимся на стадии прогрева
            lr = initial_lr + global_step * lr_increment
        else:
            lr = peak_lr

        for param_group in optimizer.param_groups:  ← Применяет вычисленную скорость обучения к оптимизатору
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])  ← В процессе полного цикла обучения вычислялись бы потери и обновления модели, которые здесь для простоты не рассматриваются
```

После запуска предыдущего кода мы отображаем графически, как изменялась скорость обучения в цикле, чтобы убедиться, что ее прогрев работает должным образом:

```
import matplotlib.pyplot as plt

plt.ylabel("Learning rate")
plt.xlabel("Step")
total_training_steps = len(train_loader) * n_epochs
plt.plot(range(total_training_steps), track_lrs)
plt.show()
```

Полученный график показывает, что скорость обучения начинается с низкого значения и увеличивается в течение 20 шагов, пока не достигнет максимального значения через 20 шагов (рис. Г.1).

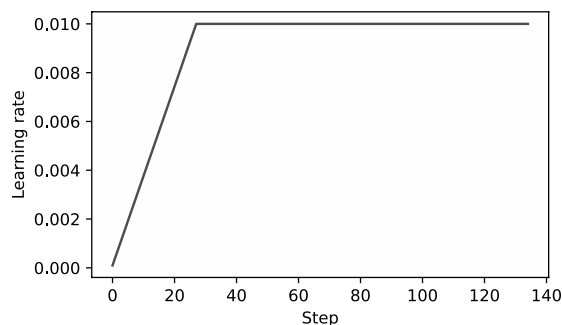


Рис. Г.1. Прогрев скорости обучения увеличивает скорость обучения на первых 20 шагах обучения. После этого скорость достигает пика 0,01 и остается постоянной до конца обучения

Далее мы изменим скорость обучения таким образом, чтобы она уменьшалась после достижения максимального показателя, что еще больше поможет улучшить обучение модели.

Г.2. Затухание по косинусу

Другой широко используемый метод обучения сложных глубоких нейронных сетей и больших языковых моделей — *затухание по косинусу*. Он модулирует скорость обучения на протяжении всех эпох обучения, заставляя ее следовать по косинусоидальной кривой после этапа разогрева.

В своем популярном варианте затухание по косинусу снижает (или уменьшает) скорость обучения почти до нуля, имитируя траекторию полукосинусоидального цикла. Постепенное снижение скорости с помощью коэффициента затухания направлено на замедление темпа обновления весов модели. Это особенно важно, поскольку помогает минимизировать риск пропуска минимума потерь в процессе обучения, а это, в свою очередь, позволяет обеспечивать стабильность обучения на более поздних этапах.

Мы можем изменить шаблон цикла обучения, добавив коэффициент затухания:

```
import math

min_lr = 0.1 * initial_lr
track_lrs = []
lr_increment = (peak_lr - initial_lr) / warmup_steps
global_step = -1
```

```

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        else:
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
            lr = min_lr + (peak_lr - min_lr) * 0.5 * (
                1 + math.cos(math.pi * progress)
            )

        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
            track_lrs.append(optimizer.param_groups[0]["lr"])

```

Применяет линейный прогрев

Использует косинусный отжиг после прогрева

И снова, чтобы убедиться, что скорость обучения изменилась должным образом, мы построим график скорости обучения:

```

plt.ylabel("Learning rate")
plt.xlabel("Step")
plt.plot(range(total_training_steps), track_lrs)
plt.show()

```

Полученный график показывает, что скорость обучения начинается с линейной фазы прогрева, которая увеличивается в течение 20 шагов, пока не достигнет максимального значения через 20 шагов. После 20 шагов линейной фазы прогрева начинается затухание по косинусоиде, постепенно снижающее скорость обучения, пока она не достигнет минимума (рис. Г.2).

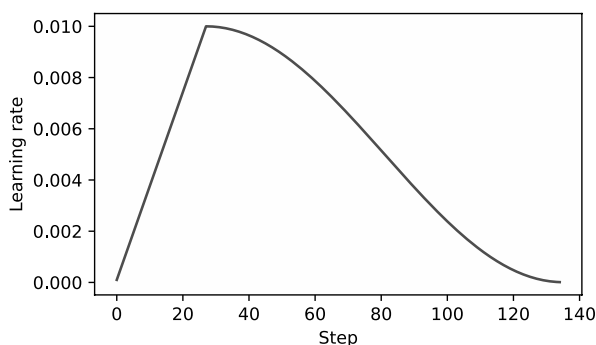


Рис. Г.2. Первые 20 шагов линейного прогрева скорости обучения сопровождаются косинусоидальным затуханием, которое уменьшает скорость в течение половины косинусоидального цикла, пока она не достигнет минимума в конце обучения

Г.3. Ограничение градиента

Ограничение градиента — еще один важный метод повышения стабильности при обучении LLM. Он предполагает установку порога, выше которого градиенты уменьшаются до заданной максимальной величины. Этот процесс гарантирует, что обновления параметров модели во время обратного распространения ошибки остаются в допустимом диапазоне.

Так, применение установки `max_norm=1.0` в функции PyTorch `clip_grad_norm` гарантирует, что норма градиентов не превысит 1,0. Здесь термин «норма» означает меру длины или величины вектора градиента в пространстве параметров модели, в частности норму L2, также известную как евклидова норма.

В математическом смысле для вектора v , состоящего из компонентов $v = [v_1, v_2, \dots, v_n]$, норма L2 определяется следующим образом:

$$|v|_2 = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}.$$

Этот метод вычисления применяется и к матрицам. Например, рассмотрим матрицу градиента, заданную следующим образом:

$$G = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

Если мы хотим ограничить эти градиенты максимальной нормой, равной 1, то сначала вычисляем норму L2 этих градиентов, которая равна:

$$|G|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5.$$

Учитывая, что $|G|_2 = 5$ превышает нашу `max_norm`, равную 1, мы уменьшаем градиенты, чтобы их норма была равна 1. Это достигается с помощью коэффициента масштабирования, рассчитанного как $\text{max_norm}/|G|_2 = 1/5$. Следовательно, скорректированная матрица градиентов G' становится:

$$G' = \frac{1}{5} \times G = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{3}{5} & \frac{4}{5} \end{bmatrix}.$$

Чтобы вы могли увидеть процесс ограничения градиента, мы начнем с инициализации новой модели и вычисления потерь для обучающей выборки, аналогично процедуре в стандартном цикле обучения:

```
from chapter05 import calc_loss_batch

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()
```


При вызове метода `.backward()` PyTorch вычисляет градиенты потерь и сохраняет их в атрибуте `.grad` для каждого тензора весов (параметров) модели.

Чтобы прояснить ситуацию, мы можем определить функцию `find_highest_gradient`, которая будет искать наибольшее значение градиента, сканируя все атрибуты `.grad` тензоров весов модели после вызова `.backward()`:

```
def find_highest_gradient(model):
    max_grad = None
    for param in model.parameters():
        if param.grad is not None:
            grad_values = param.grad.data.flatten()
            max_grad_param = grad_values.max()
            if max_grad is None or max_grad_param > max_grad:
                max_grad = max_grad_param
    return max_grad
print(find_highest_gradient(model))
```

Наибольшее значение градиента, найденное предыдущим кодом, равно `tensor(0.0411)`

Теперь применим ограничение градиента и посмотрим, как это повлияет на наибольшее значение градиента:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
print(find_highest_gradient(model))
```

Наибольшее значение градиента после применения ограничения с максимальной нормой 1 стало значительно меньшим, чем раньше:

```
tensor(0.0185)
```

Г.4. Измененная функция обучения

Наконец, мы улучшим функцию обучения `train_model_simple`, описанную в главе 5, добавив три концепции, представленные в этой главе: линейный прогрев, затухание по косинусу и ограничение градиента. В совокупности эти методы помогают стабилизировать обучение LLM.

Код с изменениями по сравнению с `train_model_simple` выглядит так:

```
from chapter05 import evaluate_model, generate_and_print_sample

def train_model(model, train_loader, val_loader, optimizer, device,
                n_epochs, eval_freq, eval_iter, start_context, tokenizer,
                warmup_steps, initial_lr=3e-05, min_lr=1e-6):

    train_losses, val_losses, track_tokens_seen, track_lrs = [], [], [], []
    tokens_seen, global_step = 0, -1
```

```

peak_lr = optimizer.param_groups[0]["lr"]
total_training_steps = len(train_loader) * n_epochs
lr_increment = (peak_lr - initial_lr) / warmup_steps

for epoch in range(n_epochs):
    model.train()
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        else:
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
            lr = min_lr + (peak_lr - min_lr) * 0.5 * (
                1 + math.cos(math.pi * progress))
        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(lr)
        loss = calc_loss_batch(input_batch, target_batch, model, device)
        loss.backward()

        if global_step > warmup_steps:
            torch.nn.utils.clip_grad_norm_(
                model.parameters(), max_norm=1.0
            )

        optimizer.step()
        tokens_seen += input_batch.numel()

    if global_step % eval_freq == 0:
        train_loss, val_loss = evaluate_model(
            model, train_loader, val_loader,
            device, eval_iter
        )
        train_losses.append(train_loss)
        val_losses.append(val_loss)
        track_tokens_seen.append(tokens_seen)
        print(f"Ep {epoch+1} (Iter {global_step:06d}): "
              f"Train loss {train_loss:.3f}, "
              f"Val loss {val_loss:.3f}")

    generate_and_print_sample(
        model, tokenizer, device, start_context
    )

return train_losses, val_losses, track_tokens_seen, track_lrs

```

Получает начальную скорость обучения из оптимизатора, предполагая, что мы используем ее в качестве максимальной скорости обучения

Вычисляет общее количество итераций в обучающем процессе

Вычисляет увеличение скорости обучения на стадии прогрева

Настраивает скорость обучения, основываясь на текущей стадии (прогрев или косинусный отжиг)

Применяет вычисленную скорость обучения к оптимизатору

Применяет ограничение градиента после прогрева, чтобы предотвратить резкое увеличение градиентов

Все, что находится ниже, остается неизменным по сравнению с функцией `train_model_simple`, использованной в главе 5

Определив функцию `train_model`, мы можем использовать ее для обучения модели аналогично методу `train_model_simple`, который применяли для предварительного обучения:

```
import tiktoken

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
peak_lr = 5e-4
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
tokenizer = tiktoken.get_encoding("gpt2")

n_epochs = 15
train_losses, val_losses, tokens_seen, lrs = train_model(
    model, train_loader, val_loader, optimizer, device, n_epochs=n_epochs,
    eval_freq=5, eval_iter=1, start_context="Every effort moves you",
    tokenizer=tokenizer, warmup_steps=warmup_steps,
    initial_lr=1e-5, min_lr=1e-5
)
```

Обучение займет около 5 минут на MacBook Air или аналогичном ноутбуке. По окончании будут получены следующие результаты:

```
Ep 1 (Iter 000000): Train loss 10.934, Val loss 10.939
Ep 1 (Iter 000005): Train loss 9.151, Val loss 9.461
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
Ep 2 (Iter 000010): Train loss 7.949, Val loss 8.184
Ep 2 (Iter 000015): Train loss 6.362, Val loss 6.876
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,, the,,,,,,,, the,,,,,,,,
the,,,,,,,,
...
Ep 15 (Iter 000130): Train loss 0.041, Val loss 6.915
Every effort moves you?" "Yes--quite insensible to the irony. She wanted him
vindicated--and by me!" He laughed again, and threw back his head to look up
at the sketch of the donkey. "There were days when I
```

Как и при предварительном обучении, модель начинает переобучаться после нескольких эпох, поскольку это очень маленький набор данных и мы проходим по нему несколько раз. Тем не менее мы видим, что функция работает, так как минимизирует потери на обучающем наборе.

Читателям рекомендуется обучить модель на более крупном наборе текстовых данных и сравнить результаты, полученные с помощью этой более сложной функции обучения, с результатами, которые можно получить благодаря использованию функции `train_model_simple`.

Приложение Д

Эффективная настройка параметров с помощью LoRA

Низкоранговая адаптация (low-rank adaptation, LoRA) — один из наиболее широко используемых методов *эффективной тонкой настройки параметров*. Ниже приводится обсуждение на примере тонкой настройки классификации спама, приведенном в главе 6. Однако тонкая настройка LoRA применима и к настройке с учителем по инструкциям, рассмотренной в главе 7.

Д.1. Введение в LoRA

LoRA — это метод, который адаптирует предварительно обученную модель к работе с конкретным, часто небольшим набором данных, настраивая только небольшое подмножество весовых параметров модели. «Низкий ранг» относится к математической концепции ограничения настроек модели подпространством меньшей размерности, чем все пространство весовых параметров. Это позволяет эффективно выявить наиболее значимые направления изменения весовых параметров во время обучения. Метод LoRA полезен и популярен, поскольку помогает эффективно дообучать большие модели на данных, характерных для конкретной задачи, значительно сокращая вычислительные затраты и ресурсы, обычно необходимые для дообучения.

Предположим, что большая весовая матрица W связана с конкретным слоем. LoRA можно применить ко всем линейным слоям в LLM. Однако для наглядности мы сосредоточимся на одном слое.

При обучении глубоких нейронных сетей в процессе обратного распространения ошибки мы изучаем матрицу ΔW , которая содержит информацию о том, насколько мы хотим обновить исходные весовые параметры, чтобы минимизировать функцию потерь во время обучения. Далее я использую термин «вес» для обозначения весовых параметров модели.

При обычном обучении и тонкой настройке обновление весовых параметров определяется следующим образом:

$$W_{updated} = W + \Delta W.$$

Метод LoRA, предложенный Э. Ху и др. (<https://arxiv.org/abs/2106.09685>), предлагает более эффективную альтернативу вычислению обновлений весов ΔW путем их аппроксимации:

$$\Delta W \approx AB,$$

где A и B — две матрицы, размер которых намного меньше, чем у W , а AB представляет собой произведение матриц A и B .

Используя LoRA, мы можем переформулировать обновление весов, которое определили ранее:

$$W_{updated} = W + AB.$$

На рис. Д.1 показаны формулы обновления весовых коэффициентов для полной тонкой настройки и LoRA.

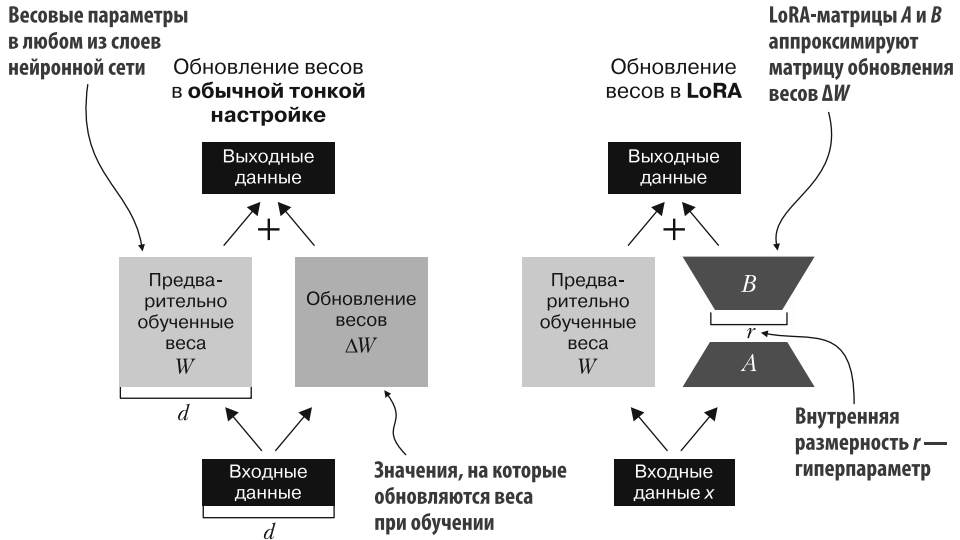


Рис. Д.1. Сравнение методов обновления весовых коэффициентов: обычная тонкая настройка и LoRA. Обычная (слева) предполагает обновление предварительно обученной весовой матрицы W непосредственно с помощью ΔW . LoRA (справа) использует две меньшие матрицы, A и B , для аппроксимации ΔW , где к W добавляется произведение AB , а r обозначает внутреннюю размерность, которая является настраиваемым гиперпараметром

Если вы внимательно присмотритесь, то заметите, что визуальные представления полной тонкой настройки и LoRA на рис. Д.1 немного отличаются от представленных ранее формул. Это различие объясняется распределительным законом перемножения матриц, который позволяет разделять исходные и обновленные веса, а не объединять их. Например, в случае обычной тонкой настройки с входными данными x мы можем выразить вычисление следующим образом:

$$x(W + \Delta W) = xW + x\Delta W.$$

Аналогичным образом мы можем написать следующее для LoRA:

$$x(W + AB) = xW + xAB.$$

Помимо сокращения количества весовых коэффициентов, которые необходимо обновлять во время обучения, возможность хранить весовые матрицы LoRA отдельно от весов исходной модели делает LoRA еще более полезным методом. На практике это позволяет сохранять веса предварительно обученной модели без изменений, а весовые матрицы LoRA применять динамически после обучения при использовании модели.

Хранить веса LoRA отдельно очень полезно, поскольку модель можно настраивать, не сохраняя несколько полных версий LLM. Это снижает требования к хранению данных и повышает масштабируемость, поскольку при настройке модели для каждого конкретного клиента или приложения необходимо корректировать и сохранять только меньшие по размеру матрицы LoRA.

Далее мы рассмотрим, как LoRA можно использовать для тонкой настройки LLM при классификации спама, аналогично примеру тонкой настройки, показанному в главе 6.

Д.2. Подготовка набора данных

Прежде чем применять LoRA к примеру классификации спама, мы должны загрузить набор данных и предварительно обученную модель, с которыми будем работать. Приведенный здесь код повторяет подготовку данных из главы 6. (Вместо того чтобы повторять код, мы могли бы запустить блокнот из главы 6 и вставить туда код LoRA из раздела Д.4.)

Сначала мы скачиваем набор данных и сохраняем его в виде файлов CSV (листинг Д.1).

Листинг Д.1. Загрузка и подготовка набора данных

```
from pathlib import Path
import pandas as pd
from ch06 import (
    download_and_unzip_spam_data,
```

```

        create_balanced_dataset,
        random_split
    )

url = \
https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
balanced_df = create_balanced_dataset(df)
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})

train_df, validation_df, test_df = random_split(balanced_df, 0.7, 0.1)
train_df.to_csv("train.csv", index=None)
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)

```

Далее мы создаем экземпляры SpamDataset (листинг Д.2).

Листинг Д.2. Создание экземпляров наборов данных PyTorch

```

import torch
from torch.utils.data import Dataset
import tiktoken
from chapter06 import SpamDataset

tokenizer = tiktoken.get_encoding("gpt2")
train_dataset = SpamDataset("train.csv", max_length=None,
                             tokenizer=tokenizer)
)
val_dataset = SpamDataset("validation.csv",
                           max_length=train_dataset.max_length, tokenizer=tokenizer)
)
test_dataset = SpamDataset(
    "test.csv", max_length=train_dataset.max_length, tokenizer=tokenizer
)

```

Создав объекты наборов данных, мы переходим к созданию загрузчиков данных (листинг Д.3).

Листинг Д.3. Создание загрузчиков данных

```

from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8

```

```

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)

val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)

test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)

```

В качестве проверки мы просматриваем загрузчики данных и убеждаемся, что каждый пакет содержит восемь обучающих примеров, состоящих каждый из 120 токенов:

```

print("Train loader:")
for input_batch, target_batch in train_loader:
    pass

print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)

```

Получаем следующий результат:

```

Train loader:
Input batch dimensions: torch.Size([8, 120])
Label batch dimensions torch.Size([8])

```

Наконец, выводим общее количество пакетов для каждой выборки:

```

print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")

```

В данном случае у нас имеются:

```

130 training batches
19 validation batches
38 test batches

```


Д.3. Инициализация модели

Мы повторяем код из главы 6, чтобы загрузить и подготовить предварительно обученную модель GPT. Мы начинаем со скачивания весов модели и их сохранения в классе GPTModel (листинг Д.4).

Листинг Д.4. Загрузка предварительно обученной модели GPT

```
from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt

CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"

BASE_CONFIG = {
    "vocab_size": 50257,  ← Размер словаря
    "context_length": 1024, ← Длина контекста
    "drop_rate": 0.0,
    "qkv_bias": True      ← Коэффициент отсева
}
                               ← Смещение запроса-ключа-значения

model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}

BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(
    model_size=model_size, models_dir="gpt2"
)

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()
```

Чтобы убедиться, что модель загружена правильно, еще раз проверим, генерирует ли она связный текст:

```
from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"

token_ids = generate_text_simple(
    model=model,
```

374 Приложение Д. Эффективная настройка параметров с помощью LoRA

```
idx=text_to_token_ids(text_1, tokenizer),
max_new_tokens=15,
context_size=BASE_CONFIG["context_length"]
)
```

```
print(token_ids_to_text(token_ids, tokenizer))
```

Следующий результат показывает, что модель генерирует связный текст, а это показатель правильной загрузки весов модели:

```
Every effort moves you forward.
The first step is to understand the importance of your work
```

Далее мы подготавливаем модель к тонкой настройке по классификации, как в главе 6, где заменяем выходной слой:

```
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(in_features=768, out_features=num_classes)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

Наконец, мы рассчитываем начальную точность классификации модели без тонкой настройки (мы ожидаем показатель около 50 %; это означает, что модель пока не способна надежно различать спамные и обычные сообщения):

```
from chapter06 import calc_accuracy_loader

torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

Начальные точности предсказания таковы:

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

Д.4. Эффективная тонкая настройка параметров с помощью LoRA

Далее мы изменяем и настраиваем LLM с помощью LoRA. Мы начинаем с инициализации слоя LoRA, который создает матрицы A и B , а также коэффициент масштабирования α и параметр rank (r). Этот слой может принимать входные данные и вычислять соответствующие выходные (рис. Д.2).

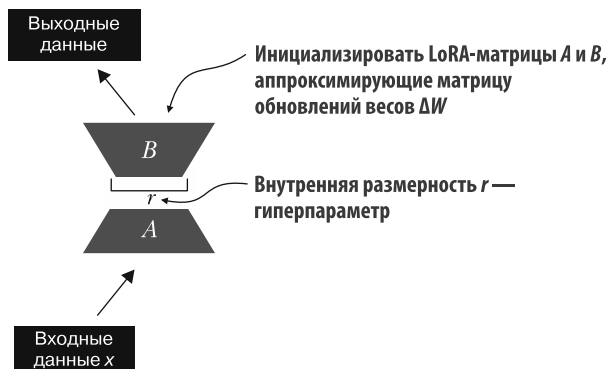


Рис. Д.2. Матрицы LoRA A и B применяются к входным данным слоя и участвуют в вычислении выходных данных модели. Внутренняя размерность r этих матриц служит настройкой, которая регулирует количество обучаемых параметров, изменяя размеры A и B

В коде этот слой LoRA может быть реализован следующим образом (листинг Д.5).

Листинг Д.5. Реализация слоя LoRA

```
import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        self.A = torch.nn.Parameter(torch.empty(in_dim, rank))
        torch.nn.init.kaiming_uniform_(self.A, a=math.sqrt(5))
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

Такая же инициализация, как и для линейных слоев в PyTorch

Параметр `rank` определяет внутреннюю размерность матриц A и B . По сути, он определяет количество дополнительных параметров, введенных в LoRA, что создает баланс между адаптивностью модели и ее эффективностью за счет количества используемых параметров.

Другой важный параметр, `alpha`, служит коэффициентом масштабирования для результатов низкоранговой адаптации. Он в первую очередь определяет степень, в которой результаты адаптированного слоя могут влиять на результаты исходного. Это можно рассматривать как способ регулирования влияния адаптации к низкому рангу на выходные данные слоя. Класс `LoRALayer`, который мы реализовали, позволяет преобразовывать входные данные слоя.

В LoRA типичной целью является замена существующих слоев `Linear`, что позволяет применять обновления весов непосредственно к уже существующим предварительно обученным весам (рис. Д.3).

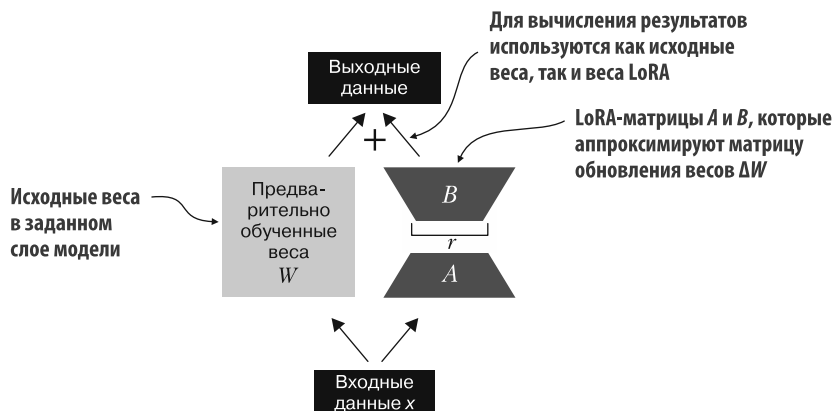


Рис. Д.3. Интеграция LoRA в слой модели. Исходные предварительно обученные веса (W) слоя объединяются с выходами матриц LoRA (A и B), которые аппроксимируют матрицу обновления весов (ΔW). Конечный результат вычисляется путем добавления выходных данных адаптированного слоя (с помощью весов LoRA) к исходному результату

Чтобы интегрировать исходные веса слоя `Linear`, мы создаем слой `LinearWithLoRA` (листинг Д.6). Он использует ранее реализованный слой `LoRALayer` и предназначен для замены существующих слоев `Linear`, таких как модули самовнимания или прямого распространения в `GPTModel`.

Листинг Д.6. Замена слоя `LinearWithLora` на слой `Linear`

```
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
```

```

self.linear = linear
self.lora = LoRALayer(
    linear.in_features, linear.out_features, rank, alpha
)

def forward(self, x):
    return self.linear(x) + self.lora(x)

```

Этот код объединяет стандартный слой `Linear` с `LoRALayer`. Метод `forward` вычисляет результат, складывая результаты исходного линейного слоя и слоя LoRA.

Поскольку весовая матрица B (`self.B` в `LoRALayer`) инициализируется нулевыми значениями, то произведение матриц A и B дает нулевую матрицу. Это гарантирует, что перемножение не изменит исходные веса, так как добавление нуля их не меняет.

Чтобы применить LoRA к ранее определенной GPT-модели, мы вводим функцию `replace_linear_with_lora`. Она заменяет все существующие слои `Linear` в модели на недавно созданные слои `LinearWithLoRA`:

```

def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear):
            setattr(model, name, LinearWithLoRA(module, rank, alpha))
        else:
            replace_linear_with_lora(module, rank, alpha)

```

← Заменяет слой `Linear` на `LinearWithLoRA`

← Рекурсивно применяет ту же функцию к дочерним модулям

Теперь мы реализовали весь необходимый код для замены слоев `Linear` в `GPTModel` только что разработанными слоями `LinearWithLoRA` в целях эффективной тонкой настройки параметров. Далее мы применим обновление `LinearWithLoRA` ко всем слоям `Linear` в модулях многоцелевого внимания и прямого распространения, а также в выходном слое `GPTModel` (рис. Д.4).

Прежде чем применять обновления для слоев `LinearWithLoRA`, мы сначала замораживаем (фиксируем) исходные параметры модели:

```

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")

```

Сейчас мы можем удостовериться, что ни один из 124 млн параметров модели не является обучаемым:

```

Total trainable parameters before: 124,441,346
Total trainable parameters after: 0

```

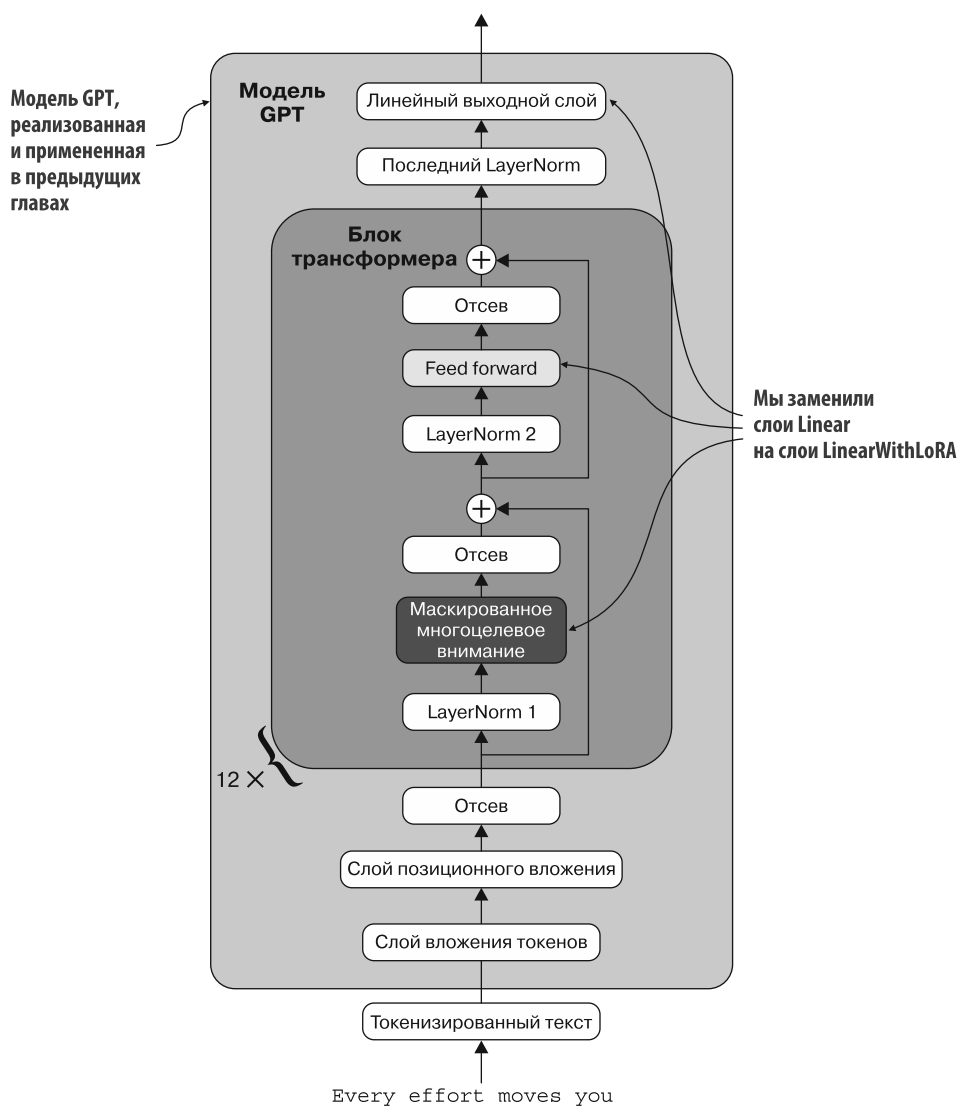


Рис. Д.4. Архитектура модели GPT. Выделены части модели, в которых слой Linear заменены слоями LinearWithLoRA в целях эффективной тонкой настройки параметров

Далее мы используем `replace_linear_with_lora` для замены слоев Linear:

```
replace_linear_with_lora(model, rank=16, alpha=16)
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
```

После добавления слоев LoRA количество обучаемых параметров таково:

Total trainable LoRA parameters: 2,666,528

Как мы видим, при использовании LoRA количество обучаемых параметров сократилось почти в 50 раз. Значения `rank=16` и `alpha=16` — хорошие варианты по умолчанию, но часто рекомендуется увеличить и параметр `rank`, что, в свою очередь, увеличивает количество обучаемых параметров. Значение `alpha` обычно выбирается равным половине `rank`, его удвоенному значению или равным ему.

Убедимся, что слои были изменены должным образом, выведя архитектуру модели:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(model)
```

Получаем следующий результат:

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    ...
    (11): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_key): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_value): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (out_proj): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (dropout): Dropout(p=0.0, inplace=False)
      )
      (ff): FeedForward(
        (layers): Sequential(
          (0): LinearWithLoRA(
            (linear): Linear(in_features=768, out_features=3072, bias=True)
            (lora): LoRALayer()
          )
          (1): GELU()
          (2): LinearWithLoRA(
```

```

        (linear): Linear(in_features=3072, out_features=768, bias=True)
        (lora): LoRALayer()
    )
)
)
(norm1): LayerNorm()
(norm2): LayerNorm()
(drop_resid): Dropout(p=0.0, inplace=False)
)
)
(final_norm): LayerNorm()
(out_head): LinearWithLoRA(
  (linear): Linear(in_features=768, out_features=2, bias=True)
  (lora): LoRALayer()
)
)

```

Теперь модель содержит новые слои `LinearWithLoRA`, которые сами состоят из исходных слоев `Linear`, веса которых замораживаются, и новых слоев `LoRA`, которые мы будем точно настраивать.

Прежде чем приступить к тонкой настройке модели, рассчитаем начальную точность классификации:

```

torch.manual_seed(123)

train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")

```

Получаемые значения таковы:

```

Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%

```

Эти значения точности идентичны значениям, указанным в главе 6. Такой результат получается потому, что мы инициализировали матрицу $LoRA\ B$ нулями. Следовательно, произведение матриц AB дает нулевую матрицу. Это гарантирует, что перемножение не изменит исходные веса, поскольку добавление нуля их не меняет.

Теперь перейдем к самому интересному — тонкой настройке модели с помощью функции обучения из главы 6 (листинг Д.7). Обучение занимает около 15 минут на ноутбуке M3 MacBook Air и менее половины минуты на графическом процессоре V100 или A100.

Листинг Д.7. Тонкая настройка модели со слоями LoRA

```
import time
from chapter06 import train_classifier_simple

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)

num_epochs = 5
train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(
        model, train_loader, val_loader, optimizer, device,
        num_epochs=num_epochs, eval_freq=50, eval_iter=5,
        tokenizer=tokenizer
    )

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

Вывод, который мы видим во время обучения, таков:

```
Ep 1 (Step 000000): Train loss 3.820, Val loss 3.462
Ep 1 (Step 000050): Train loss 0.396, Val loss 0.364
Ep 1 (Step 000100): Train loss 0.111, Val loss 0.229
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 2 (Step 000150): Train loss 0.135, Val loss 0.073
Ep 2 (Step 000200): Train loss 0.008, Val loss 0.052
Ep 2 (Step 000250): Train loss 0.021, Val loss 0.179
Training accuracy: 97.50% | Validation accuracy: 97.50%
Ep 3 (Step 000300): Train loss 0.096, Val loss 0.080
Ep 3 (Step 000350): Train loss 0.010, Val loss 0.116
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 4 (Step 000400): Train loss 0.003, Val loss 0.151
Ep 4 (Step 000450): Train loss 0.008, Val loss 0.077
Ep 4 (Step 000500): Train loss 0.001, Val loss 0.147
Training accuracy: 100.00% | Validation accuracy: 97.50%
Ep 5 (Step 000550): Train loss 0.007, Val loss 0.094
Ep 5 (Step 000600): Train loss 0.000, Val loss 0.056
Training accuracy: 100.00% | Validation accuracy: 97.50%

Training completed in 12.10 minutes.
```

Обучение модели с помощью LoRA заняло больше времени, чем обучение без LoRA (см. главу 6), поскольку слои LoRA требуют дополнительных вычислений

во время прямого прохода. Однако более крупные модели, где обратное распространение становится более затратным, обычно обучаются быстрее с помощью LoRA, чем без него.

Как мы видим, модель прошла идеальное обучение и показала очень высокую точность на проверочной выборке. Теперь визуализируем кривые потерь, чтобы лучше понять, сходится ли обучение:

```
from chapter06 import plot_values

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(
    epochs_tensor, examples_seen_tensor,
    train_losses, val_losses, label="loss"
)
```

На рис. Д.5 представлены результаты.

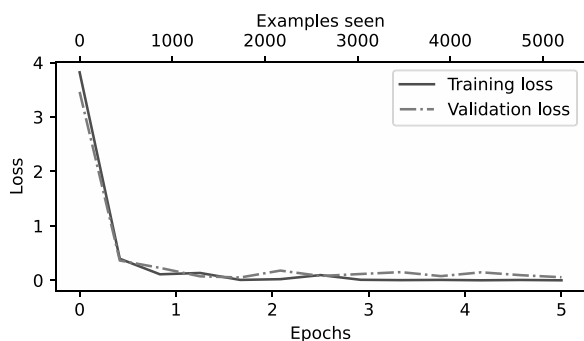


Рис. Д.5. Кривые потерь при обучении и проверке в течение шести эпох для модели машинного обучения. Сначала потери при обучении и проверке резко снижаются, а затем выравниваются, что указывает на сходимость модели, то есть на то, что при дальнейшем обучении заметного улучшения не ожидается

Помимо оценки модели на основе кривых потерь, рассчитаем точность на полной обучающей, проверочной и тестовой выборках (во время обучения мы аппроксимировали точность обучающей и проверочной выборок из пяти пакетов с помощью параметра `eval_iter=5`):

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

Значения точности таковы:

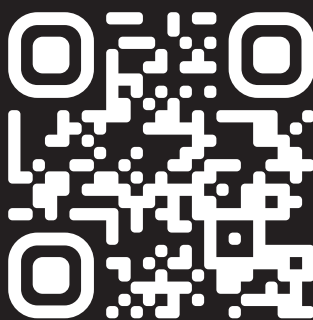
Training accuracy: 100.00%
Validation accuracy: 96.64%
Test accuracy: 98.00%

Эти результаты показывают, что модель хорошо работает на обучающей, проверочной и тестовой выборках. При точности обучения 100 % она идеально усвоила обучающие данные. Однако немного более низкая точность на проверочных и тестовых данных (96,64 и 97,33 % соответственно) указывает на небольшую степень переобучения, поскольку модель не так хорошо обобщает данные, которые не были использованы при обучении. В целом результаты очень впечатляют, учитывая, что мы оптимизировали лишь относительно небольшое количество параметров модели (2,7 млн параметров LoRA вместо исходных 124 млн параметров модели).

Read IT Club

**Комьюнити рецензентов
и переводчиков ИТ-литературы**

Миссия участников клуба – обеспечить высокое качество профессиональной переводной литературы в России. «Книжные дебагеры» проверяют корректность терминологии и подписей на схемах и иллюстрациях, чтобы сделать книги более понятными русскоязычному читателю. Стать участником Read IT Club может любой ИТ-специалист, готовый поделиться опытом с сообществом.



присоединиться к нам