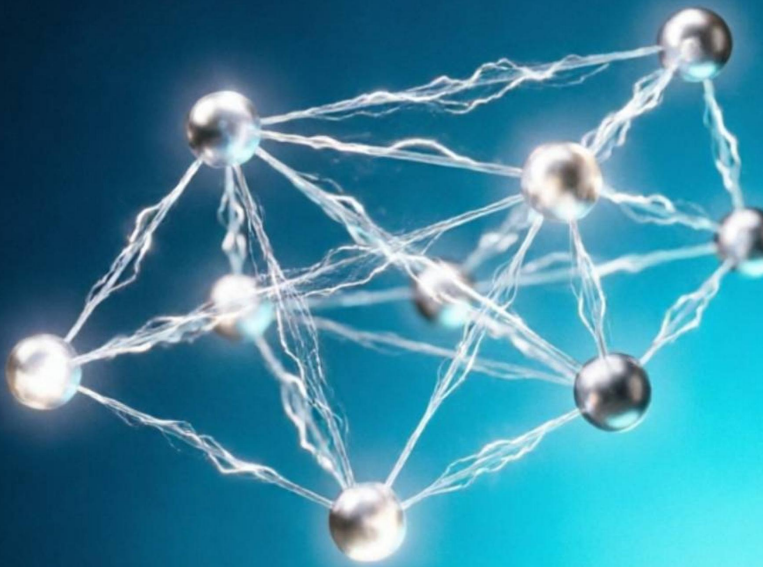
Инжиниринг искусственного интеллекта



Сергей Дегтев

Сергей Дегтев

Инжиниринг искусственного интеллекта

https://litres.ru/73943993
SelfPub; 2026

Аннотация

Представьте, что LLM - это чёрный ящик. Вы знаете, что внутри что-то есть, но каждый раз, открывая его, надеетесь на чудо. Эта книга закроет ящик навсегда.

Не учебник. Не научпоп. Инженерный путеводитель. Две книги под одной обложкой. Первая - от токена до трансформера: как работает внимание, почему память убивает процессор, сколько стоит владение моделью и что такое HBM. Вторая - от агента до лицензии на ИИ: как строить, тестировать, внедрять, считать ROI и не провалиться в цифровую ловушку самообмана.

Здесь нет магии. Есть инженерия компромиссов: скорость против памяти, качество против денег, контроль против автономии.

Автор не гуру. Автор - энтузиаст, который прошёл путь от «что такое токен» до внедрения агентов в продакшен. И честно рассказывает, где заканчивается наше понимание.

Если вы готовы заменить веру в чудо на систему координат - откройте эту книгу.

Содержание

Сергей Дегтев
Инжиниринг искусственного интеллекта

СТАРТОВАЯ КАРТА

Эта страница - только для тех, кто впервые подходит к ИИ. Если вы уже пишете промпты, разворачиваете модели, считаете TCO или внедряете RAG - закройте её. Оглавление - ваш навигатор.

Ваш маршрут

Книга 1. От токена до трансформера
Как читать: Целиком последовательно.
Зачем: Это азбука и физика LLM.

Книга 2. От инструмента к двойнику
Как читать: По диагонали, с карандашом
Зачем: Это продакшен: агенты, тестирование, ROI, безопасность. Ваша задача сейчас - запомнить *где искать*, когда столкнётесь с реальной задачей.

Как читать, чтобы не утонуть, а впитать

Первый проход

Читайте быстро. Не выписывайте формулы. Ваша цель - увидеть карту местности. Если что-то непонятно - ставьте маркер и идите дальше.

Второй проход (якоря)

Отмечайте главы, к которым вернётесь. Например: «Глава Б.3 - перечитать, когда будем настраивать RAG» «Глава В.5 - открыть, когда нужно будет защитить бюджет перед руководством» «Глава Е.4 - проверить, когда агент начнёт галлюцинировать в проде»

Третий проход (по запросу)

Возвращайтесь только когда возникает конкретная боль. Книга 2 работает как справочник, а не как учебник.

Три правила, которые могут сэкономить вам месяцы

1. Цифры устареют. Принципы - нет.

Цены на токены, бенчмарки и версии моделей изменятся быстрее, чем вы дочитаете. Книга даёт компас, а не статичную карту.

2. Повторы - не вода, а спираль.

Если встретите знакомую идею в другом контексте - не пропускайте. Так закрепляются инженерные паттерны.

3. Здесь нет кнопки «Сделать ИИ».

Есть система координат, в которой вы сами найдёте ответы, когда начнёте строить.

ВАЖНОЕ ПРЕДУПРЕЖДЕНИЕ ДЛЯ ЧИТАТЕЛЯ

Информация в этой книге носит ознакомительный и образовательный характер. Технологии искусственного интеллекта развиваются настолько быстро, а их внутренняя сложность настолько велика, что ни один специалист не может гарантировать абсолютную точность всех технических деталей, цифр и расчетов.

О процессе создания

При работе над рукописью автор использовал гибридный подход. Большая языковая модель (LLM) применялась как динамический инструмент: для генерации черновых вариантов технических рецензий, предложения формулировок, структурирования ответов и проверки гипотез. Автор полностью контролировал направление работы, определял тематику, критически оценивал, отбирал и редактировал весь предложенный контент. Итоговый текст является результатом интеллектуального труда автора.

Но если дочитаете (или перелистнете на книгу 2 в раздел Приложение, то в нем есть отдельная глава, которая посвя-

щена тому, как попробовать отличить текст, сгенерированного LLM от человеческого. В этом плане эта книга – идеальный тренажер.

Об ответственности

Автор не дает никаких гарантий, явных или подразумеваемых, относительно полноты, точности, надежности или актуальности приведенных данных. Это касается всех технических характеристик моделей, расчетов совокупной стоимости владения (TCO), архитектурных рекомендаций и любых других материалов книги.

Используйте информацию на свой страх и риск

Автор не несет ответственности за любые прямые или косвенные убытки, упущенную выгоду, потерю данных, простои оборудования или финансовые потери, возникшие в результате применения советов, формул или решений, описанных в тексте.

Технологии устаревают, цифры меняются, а каждая задача требует индивидуального тестирования. Воспринимайте эту книгу как мнение эксперта и отправную точку для вашего собственного анализа, а не как истину в последней инстанции или прямое руководство к действию в коммерческих проектах. Перед принятием серьезных инженерных или

финансовых решений - перепроверяйте данные и консультируйтесь с профильными специалистами.

В книге могут быть опечатки, и местами неровные логические переходы, но книга составлялась энтузиастом и прежде всего для себя, чтобы она не только давала ответы, а побуждала задавать вопросы в таких местах и побуждать искать ответы. Поэтому не судите строго.

И да, математики (ну или почти) здесь вы не найдете. Невозможно объять все в одной книге. А хотелось, правда хотелось.

Отдельная ремарка по визуальной части

Все схемы, диаграммы и иллюстрации в этой книге сгенерированы при помощи искусственного интеллекта (Qwen) по текстовым описаниям.

ОБРАТИТЕ ВНИМАНИЕ

В книге используется послесловие «Лицензия на ИИ», где проводится сравнение технологий искусственного интеллекта с оружием.

Автор заявляет

данная формулировка является исключительно литературной метафорой. Она не проводит прямых анало-

гий между использованием технологий ИИ и огнестрельным оружием.

Позиция автора

Оборот оружия требует строгого государственного контроля. Автор поддерживает действующее законодательство в этой сфере.

Книга не содержит пропаганды насилия или нарушения законодательства Российской Федерации.

КНИГА 1. Инженерия LLM: от токена до двойника

ПРЕДИСЛОВИЕ

Эта книга устроена необычно. Она не пытается быть «истиной в последней инстанции» и не разжёвывает каждый технический нюанс до состояния пюре. В ней два слоя.

Первый слой

для «простака» (то есть для любого нового читателя). Это толстые мазки: метафоры, объяснения «на пальцах», история развития, экономика и выбор модели. Прочитав книгу один раз, вы поймёте суть, начнёте задавать правильные вопросы и не дадите себя обмануть маркетологами.

Второй слой

для «критика» (для пытливого инженера). Это нюансы, которые сознательно оставлены «за кадром» или описаны ровно настолько, чтобы у вас зачесались руки. Вас заинтересовала балансировка экспертов в MoE? Или разница между

паттернами Sparse Attention? Отлично. Закройте книгу, откройте поисковик и углубитесь. Книга дала вам карту - клад вы найдёте сами.

И ещё одна важная деталь. Книга физически разделена на две части, каждая из которых самодостаточна.

Книга 1

это фундамент: как работает LLM, как её выбирать, сколько это стоит, как не ошибиться с архитектурой. Если вы инженер или технический лидер - начинайте здесь.

Книга 2

это продакшн и расширение: агентные системы, экономика AI-проектов, карта компетенций для продактов. В ней тоже есть инженерия - просто другая.

ЕСЛИ Вам просто нужен короткий ликбез про Искусственный интеллект в рамках этой книги – перелистывайте сразу на Книгу 2, Приложение 1.

В книге используются как русские, так и английские термины (attention, fine-tuning, inference) - они общеприняты в инженерной среде. Если вы встречаете незнакомую аббревиатуру, обратитесь к глоссарию в конце.

Идеальный читатель этой книги - не тот, кто ищет готовые ответы, а тот, кто хочет понять, куда смотреть. Эта книга - не

попытка сделать ИИ прозрачным. Это дневник инженерного пути, на котором чёрный ящик перестаёт быть чёрным ровно настолько, насколько вам нужно. Я не знал про ИИ ничего, кроме модных слов. Я задавал вопросы. Тупые. Наивные. Неправильные. Потом задавал следующие. И ещё. Я не сделал ящик стеклянным - я сделал его серым. Теперь я вижу его границы, знаю, где он врёт, где тормозит, где его можно ускорить, а куда лучше не соваться. И главное - я понял, какие вопросы можно задать, а какие - дело следующих десятилетий. Эта книга - не про ответы. Она про метод. Возьмите чёрный ящик, который вас пугает, и уменьшайте его до тех пор, пока он не перестанет быть проблемой. А когда дойдёте до границы - будем задавать новые вопросы вместе.

ВВЕДЕНИЕ

Эта книга - не для тех, кто хочет узнать, что нейросети "учатся на данных". Она для тех, кому нужно понять, как именно они устроены, чтобы принимать инженерные решения: выбирать архитектуру, оценивать стоимость инференса (время выдачи результата пользователям), читать спецификации моделей и не вестись на маркетинговые цифры.

Если вы держите эту книгу в руках, скорее всего, вы уже заметили странную пустоту на книжных полках. С одной стороны - бестселлеры про ИИ для широкой аудитории, где нейросети «учатся на данных» и «понимают контекст», но ни слова о том, как это работает на самом деле. С другой - увесистые тома с формулами и диаграммами, понятные только тем, кто уже защитил диссертацию по машинному обучению.

А где-то посередине - вы. Человек, которому нужно *понимать*, как устроены современные модели ИИ, чтобы принимать инженерные решения. Выбирать архитектуру под задачу. Оценивать стоимость инференса. Читать спецификации моделей и отделять реальные характеристики от маркетинговых цифр. Объяснять коллегам, почему одна модель лучше другой, и не выглядеть при этом ни фантазёром, ни занудой.

Эта книга - мост через эту пустоту.

За последние два года технологии ИИ прошли путь от лабораторных экспериментов до промышленного стандарта. То, что вчера было прорывом, сегодня становится рутиной. Инженеру приходится не просто успевать за изменениями, но и понимать, какие из них действительно важны, а какие - лишь шум в информационном пространстве. Эта книга даст вам систему координат, в которой вы сможете ориентироваться независимо от того, какие новые архитектуры появятся завтра.

Что вы найдёте внутри

Мы не будем учить вас программировать нейросети с нуля. Мы не будем погружаться в математические доказательства сходимости. Но мы подробно разберём то, что реально нужно знать инженеру, продакту или техническому менеджеру, работающему с ИИ.

Вы не просто получите набор фактов - вы поймёте логику, стоящую за архитектурными решениями. Увидите, почему одна оптимизация работает, а другая нет. Научитесь задавать правильные вопросы разработчикам и вендорам. И главное - сможете отделить реальные ограничения технологий от временных трудностей, которые будут решены в следующем релизе.

Вы узнаете

· Почему «токен» - это не просто слово, а единица измерения скорости и денег

· Как слова превращаются в векторы и почему «Король - Мужчина + Женщина = Королева»

· Чем отличается позиционное кодирование RoPE от ALiBi и когда что выбирать

· Как работает механизм внимания и почему без него нет современных LLM

· Зачем в трансформере целых 32 слоя и что происходит в каждом

· Почему LSTM проиграла трансформерам, хотя была прорывом

· Что такое «смесь экспертов» и как модели учатся экономить ресурсы

· Сколько *на самом деле* стоит обучение большой модели (спойлер: как крыло самолёта)

· Как из обычной языковой модели делают вежливого ассистента с помощью SFT и RLHF

· Какие оптимизации позволяют обрабатывать миллионные контексты

· Как выбирать модель под свою задачу, глядя на таблицу характеристик

· Почему будущее - не за одной гигантской моделью, а за оркестром специализированных

· Как может выглядеть ваш цифровой двойник и какие проблемы нас ждут на этом пути.

Важная оговорка

Воспринимайте конкретные цифры (количество параметров, размер контекста) как снимок эпохи 2024–2025 годов, а архитектурные решения - как фундамент, на котором вы сможете строить понимание любых новых моделей.

ГЛАВА 1. КАК РАБОТАЕТ ГЕНЕРАТИВНАЯ ЯЗЫКОВАЯ МОДЕЛЬ: ОТ ЗАПРОСА К ОТВЕТУ

«Люди - это нейронные сети. Всё, что можем делать мы, способны делать и машины» - **Джеффри Хинтон**, ученый-информатик, первопроходец в области глубокого обучения

1.1 Токен

Токен - это минимальная единица текста, на который алгоритм разбивает предложение, чтобы с ним работать. Это и единица измерения стоимости и скорости. Название подчеркивает, что это «замена» реальному тексту в цифровом мире, своего рода жетончик, который обозначает кусочек смысла.

Почему не «слово» или «символ»?

Термин «токен» выбран потому, что он шире и абстрактнее, чем «слово». Токенами становятся не только слова, но и знаки препинания, цифры, пробелы (иногда) или части слов (сабворды). Например, в предложении «Привет! Как дела?» токенами будут: [Привет] , [!] , [Как] , [дела] , [?] . Восклицательный знак - это не слово, но это важный токен.

Математическая абстракция: Для нейросети токен - это, в конечном итоге, просто индекс (число) из словаря. Назвать это «словом» было бы некорректно, так как модель оперирует числами, а не лингвистическим понятием «слова».

Итак, токен - это не просто слово, а абстрактный «жетончик смысла». Но как именно текст превращается в эти жетончики? За это отвечает следующий компонент - токенизатор.

Токенизация (Tokenization)

разделение текста на единицы обработки. Когда вы отправляете сообщение, модель не видит буквы или слова в человеческом понимании. Текст проходит через токенизатор (tokenizer) - алгоритм, который разбивает строку на минимальные единицы, называемые токенами (tokens). Существует несколько алгоритмов: BPE (используется в GPT и RoBERTa), WordPiece (BERT) и SentencePiece (LLaMA, T5, XLNet).

BPE, WordPiece и SentencePiece - в чём разница?

Токенизатор - это не «чёрный ящик». Это алгоритм, который решает одну задачу: если слова нет в словаре, на какие кусочки его разрезать? Три популярных подхода - BPE, WordPiece и SentencePiece.

BPE (Byte Pair Encoding)

Как работает

Находит самую частую пару символов в тексте, склеива-

ет её в новый токен. Повторяет, пока словарь не достигнет нужного размера.

Пример

В тексте часто встречается a b → склеиваем в ab. Потом ab c → в abc. И так далее.

+ Простой, понятный, работает для европейских языков.

– *Требует предварительной разбивки на слова (пробелы - отдельные специальные символы). Плох для языков без пробелов (китайский, японский, тайский).*

Где используется

GPT, RoBERTa.

WordPiece

Как работает

Тот же принцип склейки, но выбирает не самую частую пару, а ту, которая максимально увеличивает вероятность (правдоподобие) языковой модели.

Разница с BPE

BPE выбирает пару по частоте. WordPiece - по приросту вероятности модели (то, что лучше помогает предсказывать следующий токен).

+ Даёт чуть более «умные» токены, чем BPE.

– Тоже требует разбивки на слова. Сложнее в реализации.

Где используется
BERT.

SentencePiece

Что это
Не алгоритм, а **библиотека**, которая умеет две вещи:
· Работать с сырым текстом (без предварительной разбивки на слова по пробелам).
· Использовать внутри либо BPE, либо Unigram (третий алгоритм).

Главная фишка
Пробел в SentencePiece - обычный символ, как и все остальные. Поэтому он не требует «знать», где границы слов. Это важно для языков без пробелов (китайский, японский, тайский).
+ Универсален, работает с любыми языками.
– *Требует отдельного шага - обучения токенизатора на корпусе (хотя это делается один раз).*

Сравнительная таблица

	BPE	WordPiece	SentencePiece
Что это	Алгоритм	Алгоритм	Библиотека + BPE/Unigram
Критерий склейки	Частота пары	Прирост вероятности модели	BPE или Unigram
Нужна ли разбивка на слова	Да (пробел — спецсимвол)	Да	Нет (работает с сырым текстом)
Где используется	GPT, RoBERTa	BERT	LLaMA, T5, XLNet

Что важно для инженера

Если вы работаете с английским и европейскими языками

BPE или WordPiece дадут почти одинаковый результат. Разница заметна только на гранях.

Если ваша модель мультиязычная или работает с китайским/японским

вам нужен SentencePiece. Без него пробелы будут мешать, а слова без пробелов - резаться в случайные кусочки.

SentencePiece сегодня - стандарт для open-source LLM (LLaMA, Qwen, DeepSeek). BPE и WordPiece сегодня

используют реже (GPT-2, BERT - уже не передний край), но понимать их полезно для чтения старых статей и спецификаций.

Токен может быть:
· целым словом («кот», «человек»);
· частью слова («нейро», «сеть»);
· отдельным символом («!», «?»).

Выбор стратегии - инженерное решение, у которого нет идеального варианта, только компромиссы.

Целые слова
интуитивно понятно, но словарь разрастается до миллионов токенов (неэффективно), а модель не сможет обработать слово, которого не видела в обучении (проблема OOV - out of vocabulary).

Части слов (сабворды)
золотая середина. Словарь фиксированного размера (32K–256K), редкие слова собираются из кусочков, модель может обрабатывать новые слова. Минус: иногда нарезка выглядит нелогичной («нейро» + «сети» вместо «нейросети»).

Отдельные символы
словарь крошечный (десятки/сотни токенов), нет пробле-

мы OOV, но длинные тексты превращаются в очень длинные последовательности, что замедляет обработку и усложняет поиск смысловых связей.

Большинство современных моделей выбирают **компромиссный вариант - сабворды** (BPE, SentencePiece, WordPiece).

Как это работает технически

Токенизатор анализирует статистику встречаемости символов и их сочетаний в обучающем корпусе. Часто встречающиеся последовательности объединяются в один токен. Это позволяет держать размер словаря (обычно 32 000–256 000 токенов) фиксированным, но при этом кодировать любой текст.

Как токен получает свой номер (и почему спецсимволы - особенные)

Может показаться, что числа, которыми обозначаются токены, назначаются случайно. Но это не так. У токенизатора есть строгая логика, и она подчиняется одной цели: **сделать словарь максимально эффективным**.

Алгоритм работает следующим образом

Токенизатор начинает с **базового алфавита**. Каждому

символу - букве, цифре, знаку препинания - присваивается начальный номер. Это «фундамент» словаря. Никто не спрашивал разрешения у запятой или точки с запятой - они получили свои номера по умолчанию, потому что без них текст рассыпается.

Затем токенизатор «смотрит» на обучающий корпус (миллиарды текстов) и ищет самые частотные пары уже существующих токенов.

Самую частотную пару он «склеивает» в новый токен и присваивает ему следующий свободный номер.

Процесс повторяется тысячи раз. С каждым шагом словарь пополняется новыми токенами, которые собраны из более мелких кусочков.

Два важных нюанса про спецсимволы

Первый: пробелы

В большинстве токенизаторов пробел - это не «пустота», а полноправный символ, который получает свой номер. Когда токенизатор «склеивает» пары, он может объединить пробел со следующим словом. Именно поэтому в некоторых моделях токен « Как» (с пробелом в начале) существует отдельно - это наследие того, что на каком-то шаге пробел и слово «Как» встретились вместе достаточно часто.

Второй: пунктуация

Знаки препинания (точка, запятая, восклицательный знак) - это отдельные токены с самого начала. Они никогда не «склеиваются» со словами, если только модель специально не обучена на текстах без пробелов после знаков (а такое бывает). Поэтому в примере выше восклицательный знак «!» остался самостоятельным токеном номер 67, хотя мог бы теоретически приклеиться к слову «Привет».

Что в итоге?

Цифра 3245 - это не случайный индекс, а история: этот токен был «склеен» на каком-то шаге из двух более мелких, которые встретились вместе достаточно часто, чтобы заслужить собственное место в словаре. А цифра 67 (восклицательный знак) - это «первокирпичик», который был заложен в фундамент ещё до начала склеивания.

Чтобы увидеть этот процесс в действии, давайте проследим, как одна и та же фраза выглядит глазами человека и глазами модели. Разница окажется разительной.

Разница между тем, что видит человек и что видит модель.

Текст: [Привет! Как дела?]
Токенизация (глазами человека):

Привет	!	Как	дела	?
[слово]	[знак]	[слово]	[слово]	[знак]

Токенизация (глазами BPE):

Привет	!	Как	дела	?
[3245]	[67]	[8902]	[14567]	[67]

Что видит модель:
Вход: [3245, 67, 8902, 14567, 67]
(просто числа, никаких “слов”!)
Словарь модели (условно):
67 → “!”
3245 → “Привет”
8902 → “ Как” (с пробелом!)

Но токенизация работает по-разному не только для человека и машины, но и для разных человеческих языков. Английский, русский и китайский «весят» в токенах совершенно по-разному.

Токенизация разных языков

Токенизаторы обычно обучаются на смеси языков, но английский в ней доминирует. Это создаёт перекос:

Английский

Частотные слова становятся одним токеном. Коэффициент токенизации (токенов на слово) - 1.0-1.5.

Русский

Из-за развитой морфологии слово часто разбивается на корень и окончание. Коэффициент - 1.5-2.0.

Японский/Китайский

Иероглифы кодируются посимвольно. Коэффициент - 2.0-3.0.

Практическое следствие

При прочих равных, работа с русским текстом обходится дороже в токенах, чем с английским. Эти коэффициенты - не просто академические цифры. Для инженера они прямо переводятся в деньги.

Важное уточнение: это не потому, что английский «лучше».

Глядя на эти цифры, легко сделать неверный вывод: мол, английский - самый «экономный» язык, а русский и тем более китайский - «расточительные». Это не так.

Разница в коэффициентах - не лингвистическая данность, а исторический артефакт. Подавляющее большинство токе-

низаторов обучались на корпусах, где доминировал английский. Они «настроены» на его частотности, на его типичные сочетания символов. Русский, китайский, арабский и другие языки просто «подсели» в этот словарь, который под них не оптимизирован.

Теоретически ничто не мешает создать токенизатор, который будет экономичнее для китайского, чем для английского. Если взять корпус, где 90% текстов - на китайском, и обучить BPE или SentencePiece на нём, иероглифы начнут «склеиваться» в осмысленные частотные пары, а английские слова, наоборот, будут дробиться на части. Коэффициенты поменяются местами.

Практически это уже происходит. Китайские модели (Qwen, DeepSeek) используют токенизаторы, оптимизированные под китайский и английский одновременно, с балансировкой словаря. Русскоязычные модели (GigaChat, YandexGPT) тоже имеют свои особенности токенизации, хотя часто дообучаются поверх «английских» словарей. В специализированных моделях разница в «весе» между языками может быть сведена к минимуму или даже обращена в пользу локального языка.

Вывод для инженера

Если вы строите систему под русский язык и используете «английскую» модель - да, вы будете платить больше за токены. Но это не проклятие языка, а вопрос выбора инструмента. Существуют модели и токенизаторы, где русский «весит» столько же, сколько английский (к примеру, GigaChat и Saiga (адаптированная LLaMA)), а в некоторых случаях - даже меньше. Просто их нужно знать и уметь выбирать.

Давайте теперь посмотрим, как знание о токенах помогает считать бюджет.

Токенизация и опечатки: как модель видит ошибки

Мы только что разобрали, как токенизатор превращает текст в числа. Но что происходит, если в тексте есть опечатка? Для человека слово «следуюет» - это очевидная ошибка, мы её автоматически исправляем и понимаем смысл. Для модели всё иначе.

Посмотрим на примере

Слово: "следуюет" (опечатка, должно быть "следует")
Токены: [сле] [дую] [ет] или [с] [ле] [ду] [ю] [ет] (зависит от токенизатора)
Индексы: [1245] [7890] [3456]

Для модели это просто три числа: 1245, 7890, 3456. Она

не знает, что это «неправильное слово». Она видит последовательность токенов, которые могут быть редкими (если опечатка встречается редко) или, наоборот, часто встречаться в корпусе (если люди часто делают эту ошибку).

Два сценария

Сценарий 1: Редкая опечатка

Если слово «следуюет» встречается в обучающих данных редко, токены, на которые оно разобьётся, будут иметь низкую вероятность. Модель может не понять контекст или дать странный ответ. Но если вокруг достаточно контекста, она может догадаться по соседним словам.

Сценарий 2: Частая опечатка

Если ошибка распространена (например, «вообщем» вместо «в общем»), токенизатор может даже выделить для неё отдельный токен. Такое слово становится для модели таким же «нормальным», как и правильный вариант. Она будет обрабатывать его как обычное слово, не подозревая, что это ошибка.

Я, как человек, увидел слово «следуюет» и автоматически сопоставил его с известным мне словом «следует», проигнорировав лишнюю букву «ю». У меня есть ментальная модель

языка и знание, что такой ошибки быть не должно.

Модель же видит последовательность токенов. Если эта последовательность достаточно часто встречалась в обучении, она воспримет её как допустимый вариант. Если редко - может «растеряться». Но у неё нет «чувства правильности» - только статистика.

Практический вывод для инженера

При построении систем, которые общаются с пользователями, важно учитывать:

· Токенизатор не исправляет опечатки, он просто кодирует их

· Если ваш продукт работает с неграмотными текстами (чаты, соцсети), стоит подумать о предобработке - нормализации опечаток до токенизации

· Некоторые модели специально дообучают на текстах с ошибками, чтобы они лучше понимали реальных пользователей

Пример предобработки

Вход: "следуует заменить"

Шаг 1: нормализация опечаток (опционально) → "следует заменить"

Шаг 2: токенизация → [сле] [дует] [за] [ме] [нить]

Без нормализации модель получит редкую последовательность токенов и может ответить хуже. С нормализацией - стандартную последовательность, с которой она училась работать.

Расход токенов и бюджет

Это критически важно для

· расчёта стоимости API (Application Programming Interface)[2];

· планирования длины контекста (один и тот же текст на русском займёт больше токенов);

· выбора модели.

Итак, мы превратили текст в последовательность чисел-токенов. Но сами по себе эти числа - просто индексы, они ничего не говорят модели о смысле слов. Чтобы наделить их значением, нужен следующий шаг - эмбеддинги.

1.2 Эмбеддинги

В предыдущем разделе мы превратили текст в последовательность чисел - токенов. Но у этой победы есть обратная сторона: сами по себе эти числа [3245, 67, 8902, ...] ничего не говорят модели о смысле слов. Для компьютера 3245 и 8902 - просто разные индексы[3], между которыми нет никакой связи. Как же объяснить машине, что «кот» и «кошка» - это почти одно и то же, а «кот» и «астероид» - нет? Здесь на сцену выходят эмбеддинги.

Эмбеддинги (Embeddings) - это способ объяснить искусственному интеллекту смысл слов на языке цифр.

1. Проблема

Компьютер не понимает слова

Для компьютера текст - это просто набор непонятных значков. Единственный язык, который он реально понимает, - это числа.

Самое простое, что пришло в голову учёным - это присвоить каждому слову номер:

Кот = 1
Собака = 2
Груша = 3
Яблоко = 4
ИИ видит: 1, 2, 3, 4.

В чём здесь проблема?

Для человека очевидно, что Кот и Собака - это животные (похожи), а Яблоко и Груша - это фрукты (тоже похожи).

Но для компьютера числа 1 и 2 - это такая же пара, как и числа 1 и 4. Просто цифры.

Для него «Кот» (1) так же далек от «Собаки» (2), как и от «Груши» (3). Смысл потерян.

Очевидно, что нужен способ не просто присвоить словам номера, а расположить их в пространстве так, чтобы близкие по смыслу слова оказались рядом.

2. Решение

Погружаем слова в «пространство смыслов»

Эмбеддинги решают эту проблему. Они превращают слово не в одно число, а в вектор - длинный список координат (как на карте).

Представьте себе огромное многомерное пространство. Условно, назовем оси этого пространства признаками:

Ось X: «Степень животности» (от -1 (предметность) до +1).

Ось Y: «Степень съедобности».

Ось Z: «Степень пушистости».

Теперь мы можем расставить слова по координатам:

Кот: (Животное: +0.9, Съедобность: -1.0, Пушистость: +0.8)

Собака: (Животное: +0.9, Съедобность: -1.0, Пушистость: +0.5)

Груша: (Животное: -1.0, Съедобность: +0.9, Пушистость: 0.0)

Яблоко: (Животное: -1.0, Съедобность: +0.9, Пушистость: 0.0)

Расстояния:

Кот <-> Собака = 0.3 (близко – похожи)

Кот <-> Груша = 2.3 (далеко – не похожи)

(На самом деле осей там сотни или тысячи, и смысл их гораздо сложнее, но суть та же).

3. Зачем это нужно ИИ?

Когда у каждого слова есть координаты, компьютер начинает видеть геометрию смысла: понимание похожести: Компьютер вычисляет расстояние между точками.

Кот и Собака находятся *рядом* в пространстве (потому что координаты похожи: оба животные).

Кот и Груша находятся *далеко*.

Вывод

Компьютер «знает», что кот и собака - это похожие понятия.

Но этим всё не ограничивается. Когда слова выстроены в пространстве, компьютер может не только определять их близость, но и находить направления смысловых переходов. Например, вектор от «плохого» к «хорошему» будет примерно одинаков для разных контекстов - будь то фильмы, погода или настроение. Это позволяет модели обобщать абстрактные понятия и переносить их из одной области в другую.

Важно

Это метафора для понимания принципа. В реальности у эмбеддингов сотни или тысячи измерений, и человек не может присвоить им осмысленные названия вроде «пушистость». Оси не интерпретируемы, важны лишь расстояния между точками.

Но эмбеддинги умеют не только группировать похожие

слова. Самое удивительное свойство этого пространства - в нём можно выполнять арифметические операции, и они будут давать осмысленные результаты.

Аналогии (Математика смысла): с векторами можно производить математические операции. Если вы возьмете вектор слова «Король», вычтете вектор «Мужчина» и прибавите вектор «Женщина», вы получите вектор, ближайший к слову «Королева».

Формула выглядит так:

Король - Мужчина + Женщина = Королева.

Аналогия (векторная арифметика):
Король -> [0.8, 0.7, 0.3]
Мужчина -> [0.7, 0.6, 0.2]
Женщина -> [0.2, 0.7, 0.8]

Королева <- [0.3, 0.8, 0.9] (результат сложения/вычитания)

ИИ учится проводить такие аналогии, просто подбирая координаты.

Подведём итог: что же мы получаем, переводя слова в векторы?

Итог

Эмбеддинги нужны, чтобы перевести текст (который ИИ не понимает) в координаты точек на карте смыслов (потому что с ними можно считать и находить закономерности).

Благодаря им, нейросеть понимает, что:

«Кот» и «кошка» - это почти одно и то же.

«Хороший» и «плохой» - противоположности (векторы смотрят в разные стороны).

Для инженера

Размерность эмбеддингов (например, 4096) напрямую влияет на способность модели различать тонкие оттенки смыслов и на объём памяти, занимаемой моделью.

Итак, эмбеддинги решили проблему «слепоты» модели к смыслу отдельных слов. Теперь компьютер знает, что «кот» и «кошка» - близкие понятия. Но в языке важен не только смысл самих слов, но и их порядок. Фразы «кот укусил собаку» и «собака укусила кота» состоят из одних и тех же слов, но означают прямо противоположное. Как сообщить модели, что порядок имеет значение? Для этого существует следующий механизм - позиционное кодирование.

1.3 Позиционное кодирование (Positional Encoding)

Эмбеддинги наделили каждое слово смыслом, расположив их в пространстве так, что «кот» и «кошка» оказались рядом. Но в языке важен не только смысл самих слов, но и их порядок. Фразы «кот укусил собаку» и «собака укусила кота» состоят из одних и тех же слов, но означают прямо противоположное. Если мы просто сложим эмбеддинги, модель не увидит разницы. Как же сообщить ей, что порядок имеет значение?

Вектор токена сам по себе не содержит информации о его позиции в последовательности. Чтобы модель могла различать порядок слов, к эмбеддингам добавляют специальную метку - позиционное кодирование. Но как именно его реализовать? Здесь инженеры придумали несколько способов, и выбор между ними влияет на то, как модель будет работать с длинными текстами.

За годы развития трансформеров два метода стали самыми популярными - каждый со своей философией и областью применения. Познакомимся с ними поближе.

RoPE

Rotary Position Embedding / вращательное позиционное кодирование (используют в LLaMA, DeepSeek);

Это способ объяснить нейросети порядок слов, не теряя при этом связь между ними. Представьте, каждое слово - это бусинка, нанизанная на длинную нить (это ось предложения). Если мы просто повесим табличку с номером, бусинка так и останется бусинкой. При использовании Rotary мы начинаем закручивать каждую бусинку по мере удаления от начала нити. Первая бусинка чуть-чуть повернута, вторая - еще немного, третья - еще сильнее... В итоге все бусинки закручены в спираль вокруг основной нити.

Почему это круто для связи между словами?

Потому что поворот - это геометрическое свойство. Когда нейросеть смотрит на две бусинки (два слова), она видит не просто их «номера», а угол между ними. Чем дальше слова друг от друга, тем больше этот угол (потому что они сильнее закручены). Модель обучается реагировать на эти углы.

Например, внимание (Attention) модели может быть настроено так: «Мне нужно найти слово, которое повернуто относительно текущего ровно на определенный угол». Это математически эквивалентно фразе «найди слово через два слова слева».

Как это работает

Проблема

Трансформер (мозг GPT) видит все слова сразу. Если не отметить позицию, для него фраза «Мальчик ударил мяч» и «Мяч ударил мальчика» будут одинаковыми.

Обычные методы

Просто добавляют число к смыслу слова (позицию). Это работает, но нейросеть со временем «забывает» числа и путается в длинных текстах.

RoPE (Вращение)

Вместо того чтобы приписывать цифру, RoPE **поворачивает** вектор смысла слова. Чем дальше слово от начала, тем сильнее оно повернуто вокруг своей оси.

Почему это гениально?

Относительность

Нейросеть видит не только, что слово стоит на 5-м месте, но и *угол поворота* между словами. Это позволяет ей легко понимать зависимость между «далекими» словами (например, если в начале предложения было "он", а в конце - "потому что").

Затухание

Вращение устроено так, что очень далекие слова «перекручиваются» и их влияние ослабевает (как в реальной жизни - первые слова уже слабо влияют на конец огромной книги).

RoPE - это хитрый способ накрутить на вектор слова спиральку, которая показывает его место в предложении, чтобы нейросеть понимала не только ЧТО сказано, но и в каком ПОРЯДКЕ.

Формула расчета угла поворота

angle_diff = (pos2 - pos1) / (10000 ^ (2i / d)), где

Символ	Что означает
pos	Номер позиции токена (1, 2, 3, ...)
i	Номер пары координат в векторе (0, 1, 2, ...)
d	Размерность вектора эмбеддинга (например, 4096)
10000	Базовая константа (выбрана эмпирически)

ALiBi

Attention with Linear Biases / внимание с линейными смещениями.

Еще один способ объяснить нейросети порядок слов, но работает он **проще и эффективнее**, чем сложное вращение RoPE.

Суть одной фразой

ALiBi наказывает далекие слова, вычитая из их «внимания» штраф, который растет пропорционально расстоянию.

Как это работает

Проблема

Нейросеть (трансформер) пытается понять связь между словами. Связь между близкими словами (например, «я» и «пошел») обычно сильнее, чем между далекими.

Механизм

Когда модель смотрит на два слова и решает, насколько они связаны, ALiBi просто **вычитает число из оценки этой связи**.

Если слова стоят рядом - штраф маленький (почти ноль).

Если между ними 10 слов - штраф побольше.

Если между ними 1000 слов - штраф огромный, модель понимает: «Они слишком далеко, забудь».

Почему это круто?

Длинные тексты

ALiBi помогает модели работать с очень длинными текстами, потому что она «знает»: на каждый шаг расстояния связь ухудшается.

Простота

Не нужно сложных математических преобразований (вращений), как в RoPE. Просто берем и уменьшаем внимание к дальним словам.

ALiBi - это как линейка, которой модель бьет по рукам за попытку обращать слишком много внимания на слова, которые находятся далеко в тексте. Штраф растет ровно и предсказуемо (линейно), отсюда и название.

RoPE vs ALiBi

RoPE (вращательное кодирование):

Слова закручиваются по спирали, угол поворота = позиция

[Кот]$^{0\circ}$ [съел]$^{90\circ}$ [мышь]$^{180\circ}$

(чем дальше, тем больше поворот)

ALiBi (линейные смещения):
Внимание штрафуется за расстояние
Вес внимания = исходная_оценка - расстояние × штраф
Кот → съел: вес 0.9 - 1×0.1 = 0.8
Кот → мышь: вес 0.3 - 2×0.1 = 0.1

Итак, мы добавили к словам информацию об их порядке. Теперь модель знает, кто за кем стоит. Но этого всё ещё недостаточно, чтобы понимать сложные связи между словами. Предложение «кот укусил собаку» - это не просто последовательность слов, это история про то, кто совершил действие, а кто его испытал. Чтобы уловить эти отношения, нужен следующий, самый важный компонент трансформера - механизм самовнимания.

1.4 Механизм самовнимания (Self-Attention)

Мы наделили слова смыслом с помощью эмбеддингов, указали их порядок через позиционное кодирование, но модель до сих пор не понимает, как слова связаны между собой в предложении. Кто на кого влияет? Какое слово уточняет смысл другого? Без ответов на эти вопросы текст останется для модели просто набором независимых элементов. Механизм самовнимания - это первый шаг к тому, чтобы слова начали «общаться» друг с другом.

Каждый токен «смотрит» на все остальные и вычисляет, насколько они важны для его собственного представления. Но как один токен может «смотреть» на другой? Ведь у него нет глаз. В мире нейросетей это выглядит как математическая операция, для которой каждому слову выдаются три специальные роли:

Q - query (запрос) - что я ищу?
K - key (ключ) - что я могу дать?
V - value (значение) - какую информацию я несу?

Q, K, V - звучит загадочно, почти как код к сейфу. Но на

самом деле за этими буквами скрывается простая и элегантная логика, которую легко понять на примере онлайн-кинотеатра.

Представьте, что вы зашли на Netflix и ищете, что посмотреть вечером. Ваш поисковый запрос - это Q (что я хочу?). Вы вводите «комедии с Адамом Сэндлером». Система смотрит на карточки фильмов - это K (ключи): у каждой карточки есть жанр, актёры, год выпуска. Она ищет карточки, которые лучше всего соответствуют вашему запросу. А когда нужные фильмы найдены, вы получаете доступ к их содержимому - это V (значения), то есть сами фильмы, которые можно смотреть.

Теперь представьте, что на сайт зашли одновременно миллионы пользователей. Каждый пользователь (Q) ищет своё, сравнивая свой запрос со всеми карточками фильмов (K). Самые подходящие фильмы (V) система рекомендует каждому. Именно это и происходит в механизме внимания - только вместо пользователей у нас токены, вместо карточек - их ключи, а вместо фильмов - их значения. И все эти сравнения происходят одновременно, за один вычислительный проход.

Разница лишь в том, что в кинотеатре пользователи конкурируют за внимание системы, а в механизме самовнимания каждый токен одновременно является и пользователем

(ищет), и карточкой (описывает себя), и фильмом (несёт содержание).

В мире нейросетей всё работает точно так же, только вместо миллионов сайтов у нас - слова в предложении, а вместо поисковика - механизм внимания, который выполняется за доли секунды на GPU (подробно рассмотрено в главе 5.4).

Чтобы превратить нашу метафору в работающий алгоритм, нужно записать её на языке математики.

Технический уровень

Внимание (**Attention(Q, K, V)**) - это взвешенная сумма значений V, где веса определяются сходством Q и K.

Математически

Attention (Q , K , V) = softmax ((Q · K ^ T) / √ d #) · V

$Q \cdot K^T$ (Скалярное произведение)

Мы берем матрицу всех запросов (Q) и умножаем на матрицу всех ключей (K), транспонированную набок. На выходе получается матрица «совместимости» или «сырых весов». На пересечении слова *i* и слова *j* стоит число: насколько

запрос слова *i* похож на ключ слова *j*.

/ √d# (Масштабирование)

Делим результат на корень из размерности ключей. Это нужно, чтобы числа не были слишком большими и не забили градиент при обучении.

softmax (Нормализация)

Превращаем полученные числа в вероятности, которые в сумме дают 1. Это и есть итоговые веса внимания (например, слово «банка» решило, что на 70% нужно смотреть на слово «открыла», на 20% на «положила», а на 10% на всё остальное).

· V (Умножение на значения)

Умножаем эти вероятности на матрицу значений (V). По сути, мы берем содержимое слов, умножаем на их важность и суммируем.

Результат

Для каждого слова мы получаем новый вектор, который вобрал в себя смысл контекста. В генеративных моделях

(Decoder-only) внимание считается только к прошлым токенам, чтобы модель не «подглядывала» в будущее. Это критически важно для понимания разницы с энкодерами и называется **каузальным вниманием** (causal attention).

Сухие цифры и проценты - это хорошо, но давайте посмотрим на живую картину. Как реально выглядят эти веса внимания в работающей модели? Возьмём простое предложение и заглянем ей в голову.

Как слова "смотрят" друг на друга.

Предложение: “Она бросила ей мяч”

Матрица внимания (веса, с которым слова смотрят друг на друга):

	она	бросила	ей	мяч	Кто смотрит
она	0,1	0,7	0,1	0,1	"Она" смотрит в основном на "бросила"
бросила	0,3	0,1	0,3	0,3	"бросила" смотрит на всех
ей	0,6	0,2	0,1	0,1	"ей" смотрит на "Она" (кореференция!)
мяч	0,1	0,6	0,1	0,2	"мяч" смотрит на "бросила"

Цветовая легенда

Красный (>0.5) - сильная связь
Желтый (0.2-0.5) - средняя связь
Синий (<0.2) - слабая связь

Что мы видим:

"Она" и "бросила" - сильная связь (подлежащее-глагол)
"ей" и "Она" - сильная связь (оба об одном человеке)
"мяч" и "бросила" - сильная связь (глагол-объект)

Итак, мы разобрали, как слова общаются внутри одного слоя внимания. Но в настоящем трансформере таких слоёв не один, а десятки - от 32 до 128 и больше. Каждый следующий слой получает на вход результат работы предыдущего и строит всё более сложные абстракции. Первый слой видит только отдельные слова и их простые связи. Десятый слой уже понимает роли в предложении - кто действует, на кого направлено действие. Тридцатый слой способен уловить настроение текста, иронию, скрытые смыслы.

Но внимание - не единственный компонент в этом конвейере. Между слоями происходят и другие важные операции, которые помогают модели не забывать прошлое и глубже перерабатывать информацию. Как устроен этот конвейер целиком и зачем нужны все его детали - разберём в следующем разделе.

1.5 Трансформерные блоки (Transformer Blocks)

В предыдущем разделе мы увидели, как слова общаются друг с другом внутри одного механизма внимания. Но в реальном трансформере такой «разговор» происходит не один раз, а многократно - и каждый раз с разных точек зрения. Больше того, одного внимания недостаточно: услышанное нужно осмыслить, переработать и передать дальше. Именно так устроен трансформерный блок - базовый «станок» на конвейере производства смысла. Сырье заходит с одной стороны (просто числа - токены), а на выходе получается готовое «понимание» контекста.

Ниже приведен пример того как устроен один такой станок/трансформер и зачем нужна каждая его деталь.

1. Multi-Head Attention (Многоголовое внимание)

Это несколько параллельных механизмов внимания, каждый со своими матрицами Q, K, V. Они учатся выделять разные типы связей: грамматические, семантические, синтаксические, анафорические.

Что это

Позволяет каждому токену общаться со всеми остальными токенами сразу, но с разных точек зрения (разные «головы»).

Технически

Это тот самый механизм Q/K/V, который мы разбирали, только запущенный параллельно несколько раз (например, 32 раза).

Зачем и почему

Одной «голове» внимания мало. Одна голова может искать синтаксис (связь подлежащего и сказуемого), вторая - семантику (похожесть слов «котик» и «собачка»), третья - кореференцию. Если голова одна, она запутается.

Но почему недостаточно одной «головы»? Дело в том, что связи между словами бывают разной природы. Одни - грамматические, другие - смысловые, третьи - указательные. Чтобы разобраться во всех, нужны разные специалисты. Например, есть особый тип связей, который называется кореференцией. Разберём его подробнее.

Кореференция *- это явление, когда два или более выражений в тексте (слова, словосочетания) относятся к одному и тому же объекту реальности (человеку, предмету,*

явлению).

Это фундаментальное понятие для понимания связности текста. Без него текст был бы набором бессмысленных повторов.

Простой пример

"Иван купил машину. Он очень рад ей."

Иван и Он - кореферентны (оба указывают на одного и того же человека).

машину и ей - кореферентны (оба указывают на один и тот же предмет).

Именно такие сложные отношения, как кореференция, и помогает выявлять многоголовое внимание. Одна голова ищет подлежащее и сказуемое, другая - улавливает, что «он» и «Иван» - это один человек, третья - понимает, что «ей» относится к «машине». Вместе они создают объёмную картину связей в предложении.

Результат

На выходе мы получаем векторы, которые уже «посмотрели» на соседей и впитали немного контекста.

Итак, токены пообщались, обменялись информацией, каждый впитал немного контекста от соседей. Казалось бы, можно двигаться дальше. Но есть проблема: при передаче

информации через много слоёв есть риск потерять то, что было в самом начале. Чтобы этого не случилось, в блоке предусмотрен специальный механизм - остаточная связь.

2. Add & Norm (Добавление и нормализация) - Первый раз

Состоит из

Add (Остаточная связь)

Мы берем вход, который зашел в блок ДО Attention, и прибавляем его к выходу Attention.

Norm (Слоевая нормализация)

Мы «причесываем» полученную сумму, чтобы числа не выходили за разумные пределы.

Зачем и почему

Add (Зачем складывать?)

Представьте, что вы учитесь рисовать. Attention - это ваш новый штрих. Если штрих неудачный, вы можете откатиться назад и начать заново. Остаточная связь - это «ластик» и «карандаш» одновременно. Она позволяет модели на глубоких слоях не забывать, что было в начале. Если следующий слой не привнес ничего полезного, он может просто «пропу-

стить» информацию дальше (скопировать вход). Это помогает обучать очень глубокие сети (32+ слоя), иначе градиенты[4] затухнут.

Зачем это нужно

Проблема	Решение
В глубоких сетях (32+ слоя) градиенты затухают - модель перестаёт учиться	Остаточная связь даёт «обходной путь»: градиент может течь напрямую через сложение, минуя слой
Слой может испортить информацию, которая была полезна	Модель может «обнулить» вклад слоя, просто скопировав вход (если Attention(x) = 0)
Невозможно обучить очень глубокие сети	Без остаточных связей трансформеры с 100+ слоями просто не сходятся

Norm (Зачем нормализовать?)

Чтобы у всех нейронов был примерно одинаковый разброс чисел. Это ускоряет обучение и делает его стабильным, чтобы одни нейроны не кричали громко, а другие шептали.

Метафора

Вы работаете в команде из 10 человек. Один говорит шёпотом, другой кричит, третий вообще молчит. Вы не можете их услышать и понять. **Нормализация -** это когда вы просите всех говорить в одном тоне, чтобы каждый был слышен,

но никто не перекрикивал.

Зачем это нужно

Проблема	Решение
Значения в нейронах могут стать очень большими или очень маленькими	Нормализация приводит их к стандартному разбросу
Обучение становится нестабильным - loss скачет	Нормализация стабилизирует градиенты
Одни нейроны «перекрикивают» другие	Нормализация даёт всем нейронам сопоставимый масштаб

Add - это как если бы художник, сделав новый штрих, всё время держал перед глазами исходный набросок. Если новый штрих не удался, он всегда может вернуться к оригиналу. Norm - это «причесывание» красок, чтобы они не растекались и не смешивались в грязь.

После того как токены обменялись информацией, и мы подстраховались остаточной связью, наступает время для самого важного - осмысления. Ведь просто услышать соседей недостаточно, нужно понять, что это значит для тебя самого. Эту задачу решает следующий компонент - сеть прямого распространения.

3. Feed-Forward Network (FFN) - Сеть прямого рас-

пространения

Что это

Маленькая двухслойная нейросеть, которая применяется к каждому токену отдельно.

Технически

Сначала вектор расширяется в 4 раза (например, с 4096 до 16384 нейронов), потом сжимается обратно. Внутри - нелинейная функция (например, SwiGLU (Swish-Gated Linear Unit) или GeLU (Gaussian Error Linear Unit) - функции активации, улучшающие обучение) - они просто решают, какие нейроны зажечь, а какие погасить).

Зачем и почему

Attention смешивает информацию между токенами (как бы по горизонтали). FFN же работает внутри одного токена (вертикально). После того как токен «послушал» соседей, ему нужно этот новый услышанный смысл **переварить** и записать в удобном виде.

Аналогия

Attention - это совещание (обмен информацией между сотрудниками). FFN - это рабочий кабинет, куда сотрудник уходит после совещания, чтобы осмыслить услышанное и составить отчет. Без FFN модель не сможет делать сложные

логические выводы, она будет просто перекладывать слова. Эта метафора точно отражает разделение труда: Attention - это коллективное обсуждение, где все делятся тем, что знают. FFN - это личный кабинет, куда сотрудник уходит, чтобы обдумать услышанное и сформулировать свою новую позицию. Без FFN модель была бы просто «пересказчиком»: она бы собрала информацию от соседей, но не смогла бы её переработать в новые смыслы. FFN даёт токену возможность «обдумать» услышанное, сделать выводы, сформулировать новую позицию. Это тот шаг, который превращает коллективное обсуждение в личное понимание.

Почему FFN называют чёрным ящиком (даже инженеры)

Мы спроектировали FFN: два линейных слоя, нелинейность между ними, расширение в 4 раза и сжатие обратно. Архитектура полностью прозрачна. Но числа внутри - веса - выучились автоматически из данных. И вот тут начинается проблема.

Мы не можем открыть обученную модель, ткнуть пальцем в конкретный нейрон и сказать: «вот этот отвечает за котов, а этот - за глагол „есть“». Знание не локализовано. Оно размазано по миллионам параметров.

Более того, мы не можем объяснить на человеческом языке, *почему* конкретный входной вектор превратился в кон-

кретный выходной. Ответ всегда один: «потому что так сошлись миллиарды умножений».

Это не «чёрный ящик» в смысле «не знаем, как он устроен». Архитектуру мы знаем идеально. Это «чёрный ящик» в смысле «не можем перевести его внутреннюю работу на человеческие правила и понятия».

И это фундаментальное ограничение. Не баг. Не фича. Просто реальность глубоких нейросетей.

4. Add & Norm (Добавление и нормализация) - Второй раз

И снова, после того как токен «подумал» в FFN, мы подстраховываемся остаточной связью. Вдруг размышления завели не туда? Всегда можно вернуться к тому, что было до них. После этого блок может считаться завершённым, и обработанные векторы отправляются на следующий этаж конвейера.

Что это

То же самое, что и в пункте 2, но теперь мы складываем вход в FFN с выходом из FFN.

Зачем

Опять страхуемся, чтобы FFN не испортила то, что насобирал Attention. Если FFN сработала плохо, у нас есть «путь

обхода» через остаточную связь.

Мы разобрали, как устроен один блок. Но в настоящем трансформере таких блоков не один, а десятки - от 32 до 128 и больше. Зачем нужно так много? Неужели одного недостаточно?

Дело в том, что понимание текста строится поэтапно, от простого к сложному. Первые слои разбираются с базовой механикой языка, следующие видят более крупные структуры, а глубокие слои способны улавливать тонкие смысловые оттенки. Давайте проследим этот путь.

*Каждый следующий блок **строит абстракцию**[5].*

***Слой 1:** Видит буквы и слоги. Понимает, что «кот» - это одно слово.*

***Слой 2-5:** Видит словосочетания («рыжий кот»).*

***Слой 10-20:** Видит роли в предложении (кто действует, на кого действует).*

***Слой 30+:** Видит смысл всего абзаца, тон, настроение, сложные логические связи.*

Эти данные подтверждены визуализациями активаций нейросетей - например, в работах OpenAI (2019) по анализу GPT-2, где исследователи наглядно показали, как разные слои модели отвечают за разные уровни абстракции: от отдельных токенов до целых предложений и их смысла.

Если слоев мало - модель «мелковата» и может понимать только простые фразы. Чем больше слоев, тем сложнее концепции она способна выучить, но тем сложнее ее обучить (тут-то и помогают Add & Norm связи).

Как работает трансформер на уровне логики по шагам и на примере можно ознакомиться в Приложении 1.

Итак, после прохождения через все блоки у нас есть векторы, которые вобрали в себя и смысл каждого слова, и его связи с другими словами, и контекст всего предложения. Это - глубокое понимание текста, выраженное на языке чисел. Но наша цель - не просто понять, а сгенерировать ответ. Как из этого богатого внутреннего представления рождается новое слово? Об этом - в следующем разделе.

1.6 Выходной слой и генерация

Мы прошли долгий путь. Текст превратился в токены, токены - в эмбеддинги, эмбеддинги обогатились информацией о позиции, затем многократно прошли через блоки внимания и переработки. На выходе последнего блока у нас есть векторы, которые содержат в себе глубокое понимание всего контекста. Но как из этого понимания рождается новое слово? Как модель делает следующий шаг? Ответ - в выходном слое.

После последнего блока вектор каждого токена преобразуется в вероятность следующего токена через LM Head (language modeling head / головная сеть языкового моделирования) - это линейная проекция на размер словаря с последующей функцией softmax.

Звучит сложно, но на деле последний шаг устроен довольно просто. Вектор последнего токена (или специального токена-подсказки) проходит через финальный линейный слой, который превращает его в набор сырых оценок - по одной на каждое слово в словаре. Эти оценки называются логитами. Именно с ними и начинается игра.

1. Логиты и Softmax (Как получить вероятности)

Логиты - это «сырые» оценки, которые мозг ИИ выдает каждому слову. Например, кот = 100, собака = 90, астероид = 1. Это просто баллы, они могут быть любыми (хоть 1000, хоть -5). Сами по себе они неудобны для выбора.

Но логиты - это просто числа. Они могут быть 100, 90, 1, -5 - сами по себе они не говорят, насколько вероятно то или иное слово. Чтобы получить удобоваримые вероятности, нужен специальный преобразователь.

Softmax - это функция, которая превращает эти сырые баллы в проценты (вероятности), чтобы их сумма равнялась 100%. Результат: кот = 60%, собака = 39%, астероид = 1%.

Итак, у нас есть честные вероятности: кот 60%, собака 39%, астероид 1%. Казалось бы, всегда выбирай самое вероятное - и дело с концом. Но такой подход сделает текст скучным и предсказуемым. Иногда нам нужна креативность, иногда - строгая точность. Для этого инженеры придумали несколько регулировок.

На этапе генерации модель не всегда выбирает токен с максимальной вероятностью. Используются параметры:

2. Температура (Temperature)

Первая и самая известная регулировка - температура. Это ручка, которая крутит «креативность» модели. T -> 0 (Холодно): Модель становится жадной и почти всегда выбирает

самое вероятное слово (Кот). Текст получается точный, но скучный и предсказуемый.

T = 1 (Норма): Оставляем проценты как есть (Кот 60%, Собака 39%).

T > 1 (Горячо): Модель выравнивает шансы. Кот может получить 35%, Собака 34%, и даже Астероид 31%. Модель начинает «креативно» выбирать странные варианты, но может начать бредить.

Иногда же нам нужно сказать: «Редкие слова не рассматриваем, оставляем только основных претендентов». Для этого есть два фильтра - P и K.

3. Top-P (Ядро, nucleus sampling)

Это фильтр «пока не накопится».

Как работает

Модель сортирует слова от самых вероятных к самым редким и складывает их проценты, пока сумма не достигнет порога P (например, 0.9 или 90%). Остальные слова выкидываются.

Пример

Вы включили этот фильтр. Модель берет Кота (60%) + Со-

баку (39%) = 99% (порог P=0.9 пройден). Астероид (1%) вылетает, потому что он лишний. Выбор идет только между Котом и Собакой.

4. Top-K (K-лучших)

Top-P работает по принципу «накопи достаточно». А есть другой подход - более жёсткий: оставить ровно K лучших, и точка. Это фильтр «оставить только K претендентов». Модель оставляет только K самых вероятных слов, все остальные отсекает.

Пример: K=2. Оставляем Кота (60%) и Собаку (39%). Астероид (1%) вылетает. Выбор только из двух.

Итог

> Логиты - оценки.

> Softmax - превращает оценки в понятные доли.

> Температура - делает выбор более плоским или острым.

> Top-P - оставляет слова, пока не наберется нужная сумма вероятностей.

> Top-K - оставляет строго K самых вероятных слов.

Процесс повторяется авторегрессивно - только что сгенерированное слово добавляется к входной последовательности, и модель предсказывает следующее. Так продолжается, пока не будет сгенерирован токен конца последовательности

или не достигнут лимит длины.

Теперь вы знаете главные рычаги управления генерацией. Комбинируя температуру, Top-P и Top-K, можно получать и строгие технические тексты, и творческие рассказы, и всё что между ними. Инженерный выбор зависит от задачи.

Мы замкнули цикл. От одинокого токена на входе мы дошли до генерации нового слова на выходе. Теперь вы понимаете, как устроен «мозг» современных языковых моделей - от первой до последней операции.

Но мир ИИ не ограничивается генеративными трансформерами. Существуют и другие архитектуры, каждая со своими сильными сторонами: свёрточные сети для картинок, рекуррентные для последовательностей, диффузионные для генерации изображений. И даже внутри семейства трансформеров есть важное разделение на энкодеры, декодеры и их комбинации.

В следующей главе мы совершим экскурс по основным типам архитектур, чтобы понимать, какой инструмент для какой задачи выбирать.

1.7 Галлюцинации: когда модель уверенно врёт

Мы разобрали, как модель генерирует текст: шаг за шагом, выбирая следующее слово на основе вероятностей. Но из этого механизма вытекает фундаментальная особенность, о которой нельзя молчать.

Что такое галлюцинация?

Галлюцинация
это когда модель генерирует утверждение, которое звучит правдоподобно, но не соответствует фактам, не следует из контекста или даже противоречит здравому смыслу.

Модель может
· Придумать несуществующую цитату из книги, которой вы её спросили.
· Назвать неверную дату исторического события с полной уверенностью.
· Составить список литературы, где авторы и названия перемешаны случайным образом.
· Описать функциональность API, которого никогда не су-

ществовало.

Почему это происходит?

Не потому, что модель «врет» в человеческом смысле. У неё нет намерения обманывать. У неё вообще нет намерений.

Причина - в самой природе генеративного трансформера:

Вероятностная природа

Модель не хранит «базу фактов». Она хранит статистику - какие последовательности токенов с какой вероятностью встречались в обучающих данных. Когда она предсказывает следующее слово, она выбирает вариант, который статистически правдоподобен. Но «правдоподобно» не равно «истинно».

Отсутствие референции

Модель не имеет доступа к «источнику истины». Она не может «посмотреть в энциклопедию» во время генерации (если только вы не добавили RAG - Retrieval-Augmented Generation – Поиск и генерация). Она работает только с тем, что у неё есть в весах и в контекстном окне. Если нужной информации нет, она не скажет «я не знаю», а сгенерирует *наиболее вероятное продолжение* - которое может оказаться вымыслом.

Компрессия знаний

Обучение модели

это сжатие триллионов токенов в миллиарды параметров. Как в любом сжатии, часть информации теряется. Модель «запоминает» общие закономерности, но может забыть (или смешать) редкие детали. Когда её спрашивают о таком редком факте, она «восстанавливает» его по аналогии - и часто ошибается.

Соблазн угодить

В процессе RLHF модели учат быть полезными. Иногда это приводит к тому, что модель предпочитает дать *какой-то* ответ, даже если не уверена, вместо того чтобы признаться в незнании.

Как снижать галлюцинации по степени надёжности

Метод	Эффективность	Цена	Когда применять
RAG	Высокая (снижает на 50–70%)	Средняя (поиск + LLM)	Всегда, где есть база знаний
Снижение температуры	Средняя (снижает на 20–30%)	Низкая (просто параметр)	Для фактологических задач
Самоопрос (self-ask)	Средняя (30–40%)	Высокая (доп. токены)	Для сложных рассуждений
Ансамбль моделей	Высокая (40–60%)	Очень высокая	Только для критичных сценариев
Дообучение на отказах	Высокая (30–50%)	Высокая (fine-tuning)	Если модель слишком «уверенная»

Инженерный взгляд: это не баг, это особенность архитектуры

Галлюцинации - не ошибка, которую можно «починить» в следующем обновлении. Это фундаментальное свойство генеративных моделей, основанных на предсказании следующего токена.

Их можно *снижать*, но нельзя *устранить полностью*:

Увеличением контекста

Чем больше релевантной информации вы дадите модели в промте, тем меньше ей придется «додумывать».

RAG (Retrieval-Augmented Generation)

Подставляйте в контекст факты из надёжных источников (баз знаний, документов) перед генерацией.

Дообучением

Можно уменьшить склонность к галлюцинациям, дообучая модель на данных, где «я не знаю» - допустимый и поощряемый ответ.

Температурой и Top-P

Снижение «креативности» (температура → 0) уменьшает вероятность выбора неожиданных, а значит, потенциально ложных вариантов.

Ансамблями

Спросить одну и ту же модель несколько раз с разными настройками или несколько разных моделей - и сравнить ответы.

Что это значит для вас

Если вы строите систему, где фактические ошибки недопустимы (медицина, юриспруденция, финансы), вы *не можете* полагаться на одну LLM в чистом виде. Вам нужны дополнительные слои: проверка фактов, RAG, контрольные модели-критики, человеческая валидация.

Галлюцинации - это плата за универсальность и креа-

тивность. Понимание их природы - первый шаг к тому, чтобы строить надежные системы на ненадежных в своей основе инструментах.

1.8 RAG: как модель не выдумывает факты

В разделе 1.7 мы выяснили, что галлюцинации - фундаментальное свойство генеративных моделей. Но их можно снижать. Самый эффективный способ - RAG (Retrieval-Augmented Generation). Однако RAG не устраняет галлюцинации полностью - он лишь заменяет одни их источники на другие: вместо выдумывания из памяти модель начинает ошибаться в извлечении, ранжировании или игнорировании найденных документов.

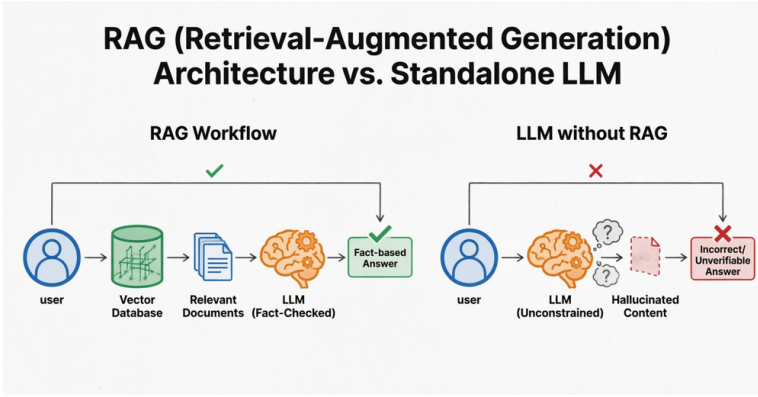
Проблема

Модель знает только то, что выучила во время обучения (данные 2024–2025). Она не знает ваш договор, переписку или свежие новости.

Идея RAG

Перед тем как ответить, модель идёт в базу знаний (векторную БД), находит там релевантные документы и подставляет их в контекст. Ответ строится не на памяти модели, а на фактах из ваших документов. Однако важно понимать: RAG не переучивает модель - он лишь временно докладывает до-

кументы в окно контекста, поэтому после ответа модель «забывает» эти данные.



Как именно модель ищет документы? Каждый документ (или его фрагмент) превращается в вектор - эмбеддинг. Эмбеддинги устроены так, что семантически похожие тексты оказываются рядом в пространстве. Запрос пользователя тоже превращается в вектор, и база данных найдет документы с самыми близкими векторами. Для этого используют специальные эмбеддинг-модели (например, intfloat/multilingual-e5-large), которые обучаются максимально точно отражать смысл текста в векторе.

Семантический поиск

это как раз то, что здесь описано. Поиск по смыслу, а не по словам («автомобиль» найдёт «машина», «транспортное средство» и «железный конь»). В отличие от лексического поиска (как Ctrl+F), он работает через эмбеддинги. Именно благодаря ему RAG понимает, какие документы реально релевантны запросу, даже если слова не совпадают буквально.

Критический нюанс

размер фрагмента (chunk size) при индексации сильно влияет на качество - слишком короткие чанки теряют контекст, слишком длинные размывают релевантность.

Как это работает (метафора)

Представьте, что вы сдаёте экзамен по открытой книге. Модель без RAG - это студент, который учил билеты год назад и всё забыл. Модель с RAG - это студент, который открывает книгу, находит нужную страницу и читает ответ с неё. Но даже с открытой книгой студент может прочитать не тот параграф - поэтому критически важна гигиена данных: мусор в базе знаний ведёт к мусору в ответе. Качество индексации и чанкинга важнее самой модели.

Почему RAG не заменяет память модели

· RAG требует хорошего поиска. Если поиск нашёл не тот документ - ответ будет неверным.

· RAG не добавляет «понимания». Модель может прочитать правильный документ, но интерпретировать его неправильно.

· RAG медленнее (поиск в БД + генерация). На практике латентность RAG-пайплайна часто складывается из трёх этапов: эмбеддинг запроса (десятки мс), поиск по индексу (единицы-сотни мс) и генерация LLM (секунды).

Когда RAG обязателен

· Работа с вашими документами (двойник, корпоративный чат-бот)
· Актуальные данные (новости, курс валют)
· Ситуации, где галлюцинации недопустимы (юриспруденция, медицина)

Есть и обратная сторона

RAG бесполезен для задач, требующих обобщения знаний со всего корпуса документов (например, «какие основные тренды были во всех наших договорах за 5 лет?») - здесь поиск по отдельным чанкам принципиально не работает, нужны другие подходы.

Настоящий боевой RAG в промышленности всегда вклю-

чает обратную связь: система логирует, на какие документы ссылалась модель, и если пользователь поставил дизлайк - этот кейс идёт в датасет для переранжировщика или дообучения эмбеддингов.

С точки зрения бизнеса, RAG - это единственный способ сделать LLM аудитируемой: вы всегда можете посмотреть, на какой документ ссылалась модель, и проверить факт, тогда как при дообучении (fine-tuning) факты «вшиваются» в веса и становятся непроверяемыми.

Для регуляторики (GDPR, HIPAA, персональные данные) RAG безопаснее дообучения, потому что документы не покидают вашу базу знаний - модель только читает их в рантайме, а не запоминает навсегда.

Самый недооценённый факт

RAG резко снижает стоимость эксплуатации - обновить документы в векторной БД стоит копейки и не требует переобучения модели за десятки тысяч долларов, как в случае с fine-tuning.

Инженерный вывод

Чистая LLM без RAG - это экзамен без книг. Хорошо для творчества, плохо для фактов. RAG - это открытая кни-

га. Медленнее, но честнее. На практике гибридные системы (RAG + дообучение на фактах) дают лучший результат, но стоят дороже.

Чек-лист: когда RAG работает (а когда - нет)

RAG - не панацея. Есть задачи, где он обязателен, есть - где бесполезен, а есть - где даже вреден.

Сценарий	RAG помогает?	Почему
«Найди в договоре пункт про неустойку»	✅ Да	Точный поиск по документу
«Сравни две версии политики конфиденциальности»	✅ Да	Нужны актуальные тексты
«Напиши стихотворение про весну»	❌ Нет	Модель и так умеет, RAG только замедлит
«Почему клиент ушёл к конкуренту?» (данных нет)	❌ Нет	Нечего искать - нечего давать
«Что сказано про скидки в последнем письме поставщика?»	⚠ Осторожно	Если письмо не проиндексировано - RAG не найдёт

Главное ограничение

RAG не добавляет «понимания». Если поиск нашёл не те документы, модель сгенерирует ответ на их основе - и это

может быть хуже, чем галлюцинация. Потому что у модели будет ложное чувство опоры.

Правило

RAG - это не «улучшатель модели». Это архитектура. Если у вас нет качественного поиска - не будет и качественного RAG.

Что делать, если RAG не помог (три следующих шага)

Вы настроили RAG, но модель всё равно галлюцинирует или отвечает невпопад. Что проверять?

Шаг 1. Диагностика поиска (самое частое узкое место)

Вывод RAG

«Я нашёл 5 фрагментов». Откройте их вручную. Релевантны ли они? Если нет - проблема не в модели, а в эмбеддерах или чанкинге.

Чанки (фрагменты текста) слишком большие → модель не удерживает внимание, теряет факты в длинном тексте.

Чанки слишком маленькие → потерян контекст, фрагмент вырван из общего смысла.

Эмбеддер не понимает язык/домен → берите multilingual-e5-large или дообучайте свой.

Шаг 2. Reranking (двухпроходный поиск)

Сначала находим 20–50 фрагментов быстрым векторным поиском. Потом пропускаем их через модель-кросс-энкодер (медленную, но точную), которая переранжирует и оставляет 3–5 самых релевантных. Это стандарт для серьёзных RAG-систем (например, Cohere Rerank, BGE-reranker).

Шаг 3. Порог уверенности и fallback

Если максимальное сходство найденного фрагмента ниже порога (например, 0.7) - модель должна сказать «не знаю», а не пытаться ответить. Добавьте в промпт:

«Если в найденных документах нет прямого ответа на вопрос - скажи: "Информация не найдена. Уточните запрос". Не додумывай.»

Это снизит галлюцинации ценой увеличения отказов. Но в юридических, медицинских, финансовых сценариях отказ лучше, чем ложь.

Эволюция RAG: графы и гибридный поиск

Классический RAG (векторный поиск + LLM) хорошо работает, когда запрос можно закрыть одним-двумя фрагментами документа. Но есть задачи, где важны не просто похожие куски текста, а цепочки связей.

Примеры таких запросов

«Какие компоненты нашей системы зависят от сервиса платежей?»

«Кто тестировал модуль, который упал вчера на проде?»

«Какие контракты связывают поставщика X с нашей компанией через других юрлиц?»

Векторный поиск ответит на них плохо. Потому что нужные факты разбросаны по разным документам, а связь между ними не выражена явно в тексте.

Паровозик знаний (GraphRAG)

Идея

построить из документов **граф знаний** - сущности (люди, компании, документы, сервисы) и связи между ними («владеет», «зависит от», «отвечает за»). Запрос превращается не в поиск похожих фрагментов, а в обход графа по цепочкам.

Пример:
запрос «Кому писать, если упал платёжный шлюз?»

Стартуем от сущности «платёжный шлюз».
Идём по связи «используется в» → «модуль оплаты».
От модуля по связи «отвечает за» → «тимлид Иванов».
От Иванова по связи «подписан на» → канал оповещений.
LLM получает не разрозненные цитаты, а связный подграф - и может дать точный, обоснованный ответ.

GraphRAG даёт выигрыш в точности на сложных, многозвеньевых запросах (multi-hop) до 50–70% по сравнению с классическим RAG. Но требует значительно больших усилий на внедрение (строить граф, поддерживать его актуальность, настраивать обход).

Гибридный поиск и реранкинг (без графов, но проще)

Если строить граф пока не нужно, но точности векторного поиска не хватает - можно усилить классический RAG без перехода к графу.

Гибридный поиск

комбинация BM25 (точный поиск по словам, как в старых поисковиках) и векторного (похожий смысл). BM25 ищет «инвойс АС-457», векторный - «документ про задерж-

ку оплаты». Выбор лучшего из двух или их взвешенная сумма дают более стабильный результат.

Ретривер

это компонент, который занимается поиском (неважно, векторным, лексическим или гибридным). Многие недооценивают его настройку, а зря: хороший ретривер сбивает входящий шум до того, как запрос попадёт в LLM.

Реранкинг (переранжирование)

сначала ретривер быстро находит 20–50 кандидатов (дешёво, неточно), а маленькая модель-кросс-энкодер (медленно, дорого) пересортировывает и оставляет топ-3–5 самых релевантных.

РЕЗЮМЕ ГЛАВЫ 1

Токен - минимальная единица текста для модели (может быть словом, частью слова или символом), а также единица измерения стоимости и скорости инференса.

Токенизация разбивает текст на токены с помощью BPE или SentencePiece; размер словаря обычно 32K–256K токенов.

Разные языки «весят» по-разному: английский - 1.0-1.5 токена на слово, русский - 1.5-2.0, китайский - 2.0–3.0, что напрямую влияет на бюджет.

Эмбеддинги превращают токены (числа) в векторы - координаты в многомерном пространстве смыслов, где близкие по смыслу слова находятся рядом.

Номер токена - не случайность, а результат статистической иерархии. Базовые символы (буквы, цифры, знаки препинания, пробелы) получают номера первыми. Частотные пары «склеиваются» в новые токены, и чем больше номер, тем глубже токен «упакован» из более мелких частей. Пробелы и знаки препинания - полноправные участники этого

процесса.

Разница в «весе» языков (английский 1.0-1.5, русский 1.5-2.0, китайский 2.0–3.0) - не лингвистическое превосходство английского, а исторический артефакт. Токенизаторы обучались на англоцентричных корпусах. При обучении на корпусе с доминированием другого языка коэффициенты поменяются местами. Китайские модели (Qwen, DeepSeek) и русскоязычные (GigaChat, YandexGPT) уже демонстрируют, что локальный язык можно сделать столь же «экономным», как английский, а иногда и экономичнее.

Векторная арифметика работает: Король - Мужчина + Женщина = Королева; это прямое следствие геометрии смыслов.

Позиционное кодирование добавляет информацию о порядке слов; два основных метода: RoPE (вращение векторов) и ALiBi (штраф за расстояние).

Механизм самовнимания (Self-Attention) позволяет словам обмениваться информацией через триплеты Q (запрос), K (ключ), V (значение).

Формула внимания: $\text{Attention} = \text{softmax}(Q \cdot K/\sqrt{d}) \cdot V$, где каждый шаг имеет инженерный смысл (совместимость, масштабирование, нормализация, взвешенная сумма).

Кореференция - явление, когда разные слова в тексте указывают на один объект («Иван купил машину. Он рад ей»); без неё текст был бы набором повторов.

Multi-Head Attention - несколько параллельных механизмов внимания, каждый ищет свои типы связей (синтаксис, семантику, кореференцию).

FFN (сеть прямого распространения) внутри каждого блока переваривает информацию после «совещания» внимания; без FFN модель просто перекладывала бы слова.

Add & Norm (остаточная связь + нормализация) позволяет информации течь через глубокие сети без затухания - как «ластик» для неудачных штрихов.

Иерархия слоёв: первые слои видят буквы и слова, средние - словосочетания и роли, глубокие - смысл абзаца, тон, настроение.

Трансформерные блоки состоят из Multi-Head Attention, Add&Norm, FFN и второго Add&Norm; 32–128 таких блоков создают глубокое понимание текста.

Галлюцинации - фундаментальное свойство генератив-

ных моделей. Они возникают из-за вероятностной природы генерации, отсутствия доступа к источнику истины и компрессии знаний при обучении. Это не баг, а особенность архитектуры, которую можно снижать, но нельзя устранить полностью. Инженерный подход: RAG, снижение температуры, дообучение, ансамбли, и всегда - критическая проверка фактов в ответственных сценариях.

RAG (Retrieval-Augmented Generation) - метод борьбы с галлюцинациями: перед генерацией модель ищет релевантные документы в векторной БД и подставляет их в контекст. Это как «экзамен по открытой книге» - факты берутся из документов, а не из памяти модели. Обязателен для задач, где ложь недопустима (юриспруденция, медицина, работа с документами). Минусы: зависит от качества поиска, медленнее чистой генерации.

Вопросы для самопроверки

1. Что такое токен и почему он является не только лингвистической, но и экономической единицей?

2. Объясните своими словами, чем эмбеддинги лучше простого присвоения числовых индексов словам.

3. В чём принципиальная разница между позиционным кодированием RoPE и ALiBi? Для каких задач стоит выбирать каждый из методов?

4. Опишите, что происходит на каждом из четырёх шагов формулы $\text{Attention}(Q, K, V) = \text{softmax}(Q \cdot K/\sqrt{d}) \cdot V$.

5. Почему в трансформере используется не одна, а множество «голов внимания» (Multi-Head Attention)?

6. Зачем нужна остаточная связь (Add) и нормализация (Norm) внутри трансформерного блока?

7. Что произойдёт с качеством модели, если убрать FFN-слои и оставить только Attention?

8. Какие параметры влияют на «креативность» генерации текста и как именно?

10. Что такое RAG? Почему он помогает бороться с галлюцинациями? В чём его ограничения и когда он обязателен?

ГЛАВА 2. ТИПЫ АРХИТЕКТУР НЕЙРОСЕТЕЙ

«Иерархическое распознавание шаблонов - вот ключ к пониманию того, как мозг, а вслед за ним и нейросети, строят картину мира» - из работ по глубокому обучению

В первой главе мы подробно разобрали, как устроены генеративные языковые модели - те самые, что лежат в основе современных чат-ботов. Но мир искусственных нейросетей гораздо шире. Прежде чем трансформеры стали главным мейнстримом, инженеры придумали множество других архитектур, каждая из которых была прорывом для своего времени и до сих пор используется в разных задачах. Более того, даже внутри семейства трансформеров есть важное разделение на энкодеры, декодеры и их комбинации. В этой главе мы совершим экскурс по основным типам архитектур, чтобы понимать, какой инструмент для какой задачи выбирать, и почему в итоге для работы с текстом победил именно трансформер.

Начнём с архитектуры, которая совершила революцию в компьютерном зрении и до сих пор остаётся основой для работы с изображениями, видео и объёмными данными. Её главная идея - не рассматривать картинку целиком, а скани-

ровать её маленьким окошком, выделяя простые признаки и собирая из них сложные.

2.1 CNN Convolutional Neural Network / Свёрточная нейросеть

Использует операцию свёртки (convolution) - скользящее окно с обучаемыми фильтрами. Эффективна для данных с пространственной структурой: изображения, видео, объёмные данные. В тексте используется редко.

Представьте, что вы рассматриваете фотографию не целиком, а через **маленькое окошко**, которое ползает по картинке. Это и есть **свертка (convolution)**.

Давайте разберем все части этого определения на простом примере - поиске границ на фотографии.

1. Скользящее окно (Как это движется?)

Что это

Представьте, что изображение - это поле 10x10 клеточек. Скользящее окно - это рамка размером 3x3 клеточки (как в игре «Сапёр», когда ты открываешь клетку и видишь соседей).

Как работает

Мы ставим эту рамку в левый верхний угол, смотрим на

9 пикселей внутри. Потом сдвигаем рамку на 1 шаг вправо, опять смотрим. Прошли строчку - переходим на следующую. Так окно «прочесывает» всю картинку.

2. Обучаемые фильтры (Что ищет окно?)

Внутри этого окошка есть специальная табличка с числами - **фильтр** (или ядро свертки). Фильтр - это детектор. В начале обучения эти числа случайные, но потом они «обучаются» искать конкретные вещи.

Фильтр на поиск горизонтальных границ

Если внутри окошка картинка резко меняется сверху (темно) вниз (светло), фильтр выдает большое число. Если картинка однородная, фильтр выдает ноль.

Фильтр на поиск углов

Реагирует на диагональные линии.

Фильтр на поиск цвета

Реагирует на красные пятна.

Практическая польза

Мы не программируем правила «ищи вертикальную линию» ручками. Нейросеть сама подбирает числа в фильтрах во время обучения, чтобы находить то, что нужно для задачи

(например, для распознавания котиков).

Фильтры умеют находить линии, границы, углы. Но почему это работает именно с картинками, а не с текстом? Потому что у изображений есть важное свойство - пространственная структура.

3. Пространственная структура (Почему это важно?)

Что это значит

в данных важен порядок элементов. У пикселя есть соседи сверху, снизу, слева и справа. Если перемешать пиксели как попало, картинка рассыплется, и смысл пропадет.

Зачем это свёртке

Свёртка использует эту структуру. Она смотрит на локальные группы пикселей (близких соседей), чтобы понять простые вещи (края, текстуры), а потом из простых вещей собирает сложные (глаз, ухо, нос).

Главная практическая польза (3 вещи)

1. Экономия ресурсов (разреженность)

Обычная полносвязная нейросеть на картинку 1000x1000 пикселей с тремя цветовыми каналами потребовала бы около 3 миллионов входных нейронов и, если следующий слой

тоже будет размером 1000x1000, около 3×10^{12} связей - это триллионы параметров.

2. Инвариантность к положению (неважно, где объект)

Фильтр, который научился искать нос, найдет его в любом месте фотографии - слева, справа, вверху. Ему все равно, потому что он скользит везде.

3. Иерархия признаков

· Первые слои видят просто линии и точки.

· Средние слои видят комбинации линий (круги, прямоугольники) - например, колесо.

· Глубокие слои видят сложные объекты (машина, дом).

Почему в тексте используется редко?

У текста нет такой сильной пространственной структуры. Если в тексте переставить два соседних слова, смысл может измениться кардинально или остаться тем же. А если на картинке переставить два пикселя местами, это будет уже «шум». Для текста лучше подходит **трансформер (внимание)**, который смотрит на связь слов попарно, независимо от расстояния, а не только на соседей в окошке.

Итак, свёрточные сети отлично подходят для данных с пространственной структурой, где важны локальные связи и

иерархия признаков. Но текст устроен иначе: смысл зависит не столько от соседних пикселей, сколько от порядка слов и их взаимосвязей на расстоянии. Для обработки последовательностей инженеры придумали другой класс архитектур - рекуррентные нейросети. Именно они правили бал в задачах с текстом до появления трансформеров.

2.2 RNN (Recurrent Neural Network / Рекуррентная нейросеть)

Мы только что разобрались, как свёрточные сети видят мир через маленькое окошко и собирают картинку из простых деталей. Но у текста другая природа - это не застывшее изображение, а поток, последовательность, где важен каждый предыдущий шаг. Чтобы работать с такими данными, нужна архитектура с памятью. Знакомьтесь: рекуррентные нейросети - главные звезды обработки текстов до эры трансформеров.

Представьте, что вы читаете книгу, но у вас очень плохая память - вы помните только последние пару предложений, а начало главы уже стерлось. Примерно так работают рекуррентные сети. Давайте разберемся, почему они устроены именно так и в чем их главные проблемы.

RNN обрабатывает последовательности шаг за шагом, передавая скрытое состояние. Она плохо параллелизуется и страдает от затухания градиента. Практически вытеснена трансформерами.

Вся суть RNN крутится вокруг одной простой идеи: у сети есть внутренняя память - скрытое состояние, которое об-

новляется с каждым новым словом. Это как блокнотик, куда модель записывает свои мысли по ходу чтения.

1. Скрытое состояние (Память в голове)

Что это

У RNN есть внутренняя переменная - вектор скрытого состояния (hidden state). Это как блокнотик, в котором нейросеть записывает свои мысли по ходу чтения.

Технически

Это просто набор чисел, которые меняются после каждого прочитанного слова.

2. Обработка шаг за шагом (Последовательность)

RNN не смотрит на весь текст сразу. Она читает его строго по порядку, слово за словом.

Как это выглядит на примере фразы

«Кот съел мышь»:

Шаг 1 (Кот):

Смотрит на слово «Кот» + чистый блокнот (пустая память). Записывает в блокнот: *«Вижу кота»*.

Шаг 2 (съел):

Смотрит на слово «съел» + читает блокнот (там про кота). Понимает: *«Кот что-то делает»*. Обновляет блокнот: *«Кот сейчас ест»*.

Шаг 3 (мышь):

Смотрит на слово «мышь» + читает блокнот (там про кота, который ест). Понимает: *«Кот съел мышь. Логично»*.

Формула одной фразой

Новое состояние = Функция (Текущее слово, Старое состояние)

3. Проблема 1: Плохая параллелизация (Медленность)

Пока вы не прочитаете слово «съел», вы не можете понять слово «мышь», потому что память еще не обновилась.

Для процессора это ад: Нельзя обработать все слова одновременно (распараллелить), как в трансформере. Приходится ждать окончания первого шага, чтобы начать второй. На длинных текстах это очень медленно.

Более того, даже если у вас есть суперкомпьютер с тысячами ядер, RNN заставит их простаивать - каждое следующее слово требует результатов предыдущего, и распараллелить этот процесс принципиально невозможно. Это фунда-

ментальное ограничение последовательной архитектуры.

4. Проблема 2: Затухание градиента (Забывчивость)

Это главная техническая боль RNN. Представьте предложение: *«В далеком 1991 году, когда шел сильный дождь, а на часах было уже поздно, я ... вышел из дома».*

Чтобы понять, что «вышел» относится к «я», нейросети нужно помнить информацию из самого начала предложения.

В RNN информация при каждом шаге умножается на весовые матрицы и проходит через нелинейные функции активации. При обратном распространении ошибки градиенты, соответствующие дальним шагам, многократно умножаются на эти матрицы и экспоненциально затухают (или взрываются). К тому моменту, как мы дошли до конца, вклад слова «я» в градиент становится практически нулевым - модель не может обучиться учитывать далёкие зависимости.

Итог

RNN хорошо помнит последние 5-10 слов, но забывает, что было в начале абзаца. Она страдает от кратковременной памяти.

Инженеры пытались бороться с этим, тщательно подбирая начальные значения весов и используя специальные техники вроде отсечения градиентов (gradient clipping). Но это были лишь костыли - проблема оставалась в архитектуре, и

для действительно длинных последовательностей RNN оставались бессильны.

Почему их вытеснили Трансформеры?

Признак	RNN (Старый подход)	Трансформер (Новый подход)
Как читает	Последовательно, слово за словом.	Одновременно смотрит на все слова.
Память	Скрытое состояние (вектор), которое перезаписывается. Теряется информация.	Прямой доступ к любому слову через механизм внимания (Attention).
Скорость	Медленно (нельзя распараллелить).	Быстро (можно считать всё сразу на GPU).
Длина текста	Короткая (до 100-200 слов, потом забывает).	Огромная (до 1 млн+ токенов в современных моделях).

RNN

это последовательный процессор, который читает текст по порядку и ведет конспект в блокноте. Но блокнот маленький, и к концу главы он забывает, что было на первой странице. Трансформеры решили эту проблему, отказавшись от последовательного чтения в пользу полного обзора всего текста сразу.

Но инженеры не сдавались так легко. Они придумали, как улучшить память RNN, добавив специальные вентили, ко-

торые решают, что забывать, а что помнить. Так появились LSTM - долгая краткосрочная память. О них - в следующем разделе.

2.3 LSTM (Long Short-Term Memory / Долгая краткосрочная память)

Мы только что увидели главную проблему простых рекуррентных сетей - их короткую память. RNN помнит только последние несколько слов, а всё, что было в начале предложения, безвозвратно теряется. Но инженеры нашли способ научить сеть не забывать важное. Они добавили специальные вентили, которые решают, что выбросить, а что сохранить на долгую память. Так родилась архитектура LSTM.

LSTM - это вариант RNN с вентилями: входным, забывания и выходным. Она решает проблему долгосрочной памяти, но всё так же не параллелится.

LSTM расшифровывается как Long Short-Term Memory - долгая краткосрочная память. В названии уже скрыта суть: это всё ещё рекуррентная сеть с краткосрочной памятью (как RNN), но теперь она может хранить информацию гораздо дольше. Давайте разберёмся, как ей это удаётся.

Чтобы понять, чем LSTM отличается от обычной RNN, представьте, что простая RNN - это человек, который при

чтении книги каждый раз переписывает свой блокнот заново, стирая всё старое. LSTM же ведёт себя как опытный читатель: он носит с собой стопку стикеров-напоминалок и периодически выбрасывает те, что уже не нужны, оставляя важное.

Как же устроен этот «умный блокнот»? В LSTM появляется дополнительная «лента конвейера», которая тянется через всю сеть и несёт в себе важную информацию, почти не меняясь. А три специальных вентиля решают, что с этой лентой делать.

1. Проблема, которую решает LSTM

Обычная RNN страдает от **кратковременной памяти**. Она помнит последние слова, но забывает начало предложения. LSTM ввели специальный механизм - **ленту конвейера (cell state)** и **вентили (gates)**, которые решают, какую информацию пронести через всю последовательность нетронутой.

2. Как это работает (Три вентиля)

У LSTM есть три хитрых фильтра, которые регулируют поток информации. Снова пример с предложением: **«Кот (1) ... (много слов) ... (2) наелся и (3) спит»**.

Это главное нововведение LSTM. Информация по ней те-

чёт, минуя нелинейные преобразования, проходя только через линейные операции (сложение и умножение на вентили). Благодаря этому градиенты могут распространяться на сотни шагов без затухания.

Вентиль 1: Забывание (Forget Gate) - «Что выкинуть?»

Вопрос:

«Старая информация о коте всё ещё нужна или уже пора её забыть?»

Когда мы дошли до слова «наелся», модель понимает: «Кот всё ещё актуален, ничего не забываем».

А когда дошли до слова «спит», можно чуть ослабить информацию о том, что он «ел», оставив только факт наличия кота.

Вентиль 2: Входной (Input Gate) - «Что записать нового?»

Вопрос:

«Какая информация из нового слова («наелся») действительно важна и её стоит добавить в ленту конвейера?»

Модель решает: «Факт того, что кот сыт, важен. Добавляю его в ленту памяти».

Вентиль 3: Выходной (Output Gate) - «Что показать ми-

ру?»

Вопрос:

«Основываясь на всей ленте памяти (кот + сыт), что я должен выдать как результат прямо сейчас?»

На выходе мы получаем новое скрытое состояние (которое пойдет в следующий шаг), но сама лента памяти остаётся нетронутой для будущих шагов.

3. Почему LSTM круче RNN?

Благодаря этим вентилям, LSTM может протащить важную информацию через сотни шагов.

Пример: в начале текста было слово «**Она**». LSTM держит это в ленте памяти, не перезаписывая. Через 50 слов встречается глагол «**пошла**». LSTM сверяется с лентой, видит там «Она» и понимает, что нужно использовать женский род. RNN к этому моменту уже забыла бы, о ком речь.

4. Недостатки (Почему и её вытеснили)

LSTM решила проблему забывания, но унаследовала главную проблему RNN - **последовательность**:

Нет параллелизации

Чтобы обработать 1000-е слово, нужно дождаться резуль-

татов обработки 999 предыдущих. На современных видеокартах (GPU), которые любят считать всё одновременно, это очень медленно.

Сложность

Три вентиля делают архитектуру громоздкой и требовательной к ресурсам. Каждый вентиль - это отдельная нейросеть со своими весами, поэтому LSTM требует значительно больше памяти и вычислений, чем простая RNN.

LSTM

это RNN с «умным блокнотом». Вместо того чтобы просто переписывать страницу, она решает, какие старые записи вычеркнуть (forget gate), какие новые внести (input gate) и что сейчас прочитать вслух (output gate). Длинную память это лечит, но скорость работы остаётся последовательной, что в итоге привело к победе Трансформеров.

LSTM решила проблему забывания, но породила новую - сложность и медлительность. Инженерам это не понравилось, и они начали искать способы упростить архитектуру, сохранив её преимущества. Так появилась следующая модель - GRU, которая уволила лишнего менеджера.

2.4 GRU (Gated Recurrent Unit / Рекуррентный блок с вентилями)

GRU

это упрощённая версия LSTM, объединяющая входной вентиль и вентиль забывания в один «вентиль обновления» (update gate). У GRU всего два вентиля вместо трёх, и нет отдельного состояния ячейки - только скрытое состояние. Это даёт примерно на 25–30% меньше параметров и чуть более быстрые вычисления, но на очень длинных последовательностях (сотни шагов) LSTM может показывать slightly better результаты.

это LSTM, который уволил лишнего менеджера. За счёт такого сокращения GRU работает быстрее и требует меньше памяти. Но, как водится, за скорость приходится платить - чуть-чуть качеством на очень длинных последовательностях.

Представьте, что LSTM - это отдел с тремя сотрудниками: один решает, что забыть (forget gate), второй - что записать нового (input gate), третий - что показать на выходе (output gate). GRU оставляет только двоих:

Reset gate - «Секретарь»: решает, какую часть старых записей вообще не брать в расчёт (игнорировать прошлое).

Update gate - «Руководитель»: сразу решает две задачи - сколько нового взять из текущего слова и сколько старого сохранить (совмещает функции input и forget gate).

Зачем? Меньше сотрудников - меньше зарплата (параметры) и быстрее работа. GRU чуть проще, чем LSTM, и на многих задачах показывает почти то же качество. Но если текст очень длинный и связи сложные, LSTM всё же точнее.

Практическое применение

· Машинный перевод (Google Translate, Яндекс.Переводчик).
· Распознавание речи (Siri, Алиса).
· Генерация текста (автодополнение в поиске, чат-боты).
· Анализ временных рядов (прогноз цен акций, погоды).
· Обработка видео (анализ действий по кадрам).

Главное отличие от LSTM

LSTM чуть точнее на очень длинных последовательностях, но GRU проще, быстрее и часто даёт почти такое же качество, поэтому его часто выбирают, когда важна скорость.

RNN, LSTM, GRU - все они пытались работать с последовательностями шаг за шагом, и все упирались в одну и ту же стену: их нельзя было распараллелить. Нужно было принципиально новое решение, которое смотрело бы на текст сразу целиком. И в 2017 году оно появилось. Революция случилась - на сцену вышел Трансформер.

2.5 Трансформер (Transformer)

Все архитектуры, которые мы рассмотрели до сих пор - CNN, RNN, LSTM, GRU - были великими для своего времени, но у каждой находились фатальные недостатки. CNN не понимали порядок слов, RNN забывали начало предложения, LSTM и GRU были слишком медленными из-за последовательной обработки. Нужен был прорыв.

И в 2017 году команда Google опубликовала статью с громким названием **«Attention Is All You Need»**. Так родился Трансформер - архитектура, которая стёрла все предыдущие с доски лидеров.

Это был не просто очередной шаг в эволюции нейросетей, а настоящая смена парадигмы. Трансформер отказался от последовательной обработки, которая казалась естественной для текста, и предложил смотреть на все слова сразу. Механизм самовнимания позволил каждому токену взаимодействовать с любым другим напрямую, независимо от расстояния, а параллелизация вычислений дала возможность обучать модели с сотнями миллиардов параметров. За считанные годы трансформеры захватили не только обработку текста, но и компьютерное зрение, генерацию изображений и даже анализ молекул.

Что такое трансформер?

Технически

Трансформер - это архитектура нейросетей, основанная исключительно на механизме самовнимания (self-attention). Он позволяет модели вычислять попарные веса важности между всеми токенами последовательности **за один проход**. Благодаря этому каждый токен может напрямую взаимодействовать с любым другим, независимо от расстояния, а все вычисления легко распараллеливаются на GPU.

Метафора

Представьте, что вы читаете книгу не последовательно, а видите сразу всю страницу целиком. Вы можете мгновенно установить связь между словом на первой строке и словом на последней, потому что они оба перед глазами. Именно так работает трансформер.

Три кита, на которых стоит трансформер

Прежде чем разбирать семейства моделей, запомните три главные идеи, которые сделали трансформер революцией:

1. САМОВНИМАНИЕ (Self-Attention)

Каждый токен “смотрит” на все остальные и решает, кто для него важен. Это как совещание, где каждый слушает

каждого.

2. ПАРАЛЛЕЛИЗМ (Parallelism)

Все токены обрабатываются одновременно, а не по очереди. Это как читать все книги в библиотеке сразу, а не по одной.

3. МНОГОСЛОЙНОСТЬ (Multi-layer)

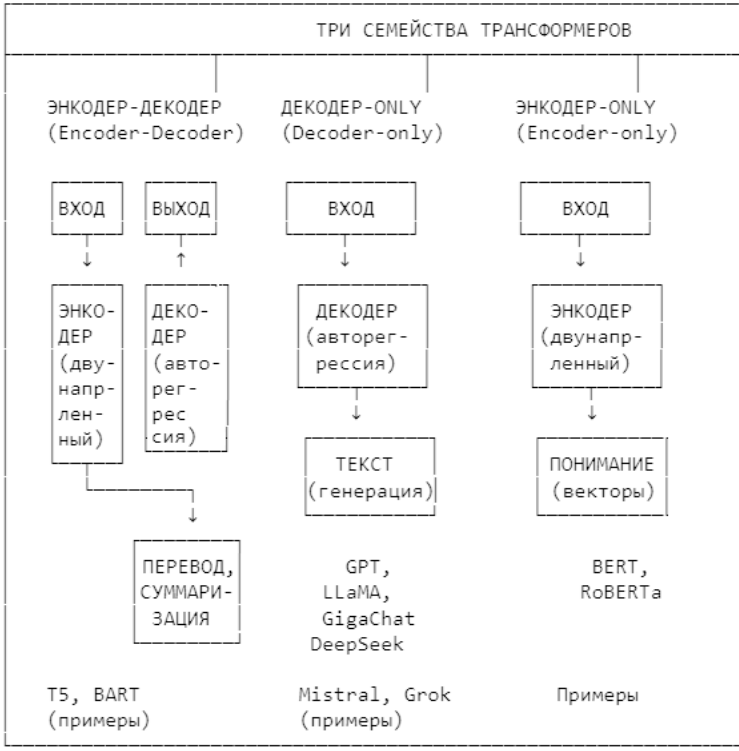
Десятки слоев трансформера строят иерархию понимая:

Слой 1: буквы -> Слой 5: слова -> Слой 15: роли -> Слой 30: смысл

Три основных семейства Transformer

В зависимости от того, какую часть трансформера мы используем - только энкодер, только декодер или оба сразу - получаются модели с совершенно разными способностями.

Визуализация трёх семейств



1. Decoder-only (только декодер)

Как работает

Использует каузальное внимание (causal attention) - каждое слово видит только предыдущие слова. Это как читать

книгу, закрывая рукой ещё не прочитанные страницы.

Зачем

Генерировать текст, предсказывая следующее слово.

Кто использует

Все современные чат-боты - GPT, GigaChat, DeepSeek, LLaMA, Mistral, Grok.

Метафора

Это писатель, который пишет роман последовательно, главу за главой, не заглядывая в будущее.

2. Encoder-only (только энкодер)

Как работает

Использует двунаправленное внимание (bidirectional attention) - каждое слово видит и левый, и правый контекст. Как читатель, который видит всю страницу целиком и может установить любые связи.

Зачем

«Понимать» текст целиком, но не генерировать новые фразы. Анализ, классификация, извлечение смысла.

Кто использует

BERT, RoBERTa.

Метафора

Это литературный критик, который анализирует готовый роман, видит все связи и может ответить на вопросы, но сам писать не умеет.

3. Encoder-Decoder (кодер-декодер)

Как работает

Состоит из двух частей: энкодер читает входной текст (двунаправленно), а декодер генерирует выходной (авторегрессионно).

Зачем

Задачи, где нужно преобразовать одну последовательность в другую: перевод, суммаризация, перефразирование.

Кто использует

T5 (Text-to-Text Transfer Transformer), BART (Bidirectional and Auto-Regressive Transformer).

Метафора

Это переводчик: сначала полностью понимает исходный текст (энкодер), потом пишет перевод (декодер).

Популярные модели: глубокое погружение

Из трёх семейств выросли все современные модели-звёзды. Одни научились глубоко понимать текст, другие - генерировать, третьи - переводить с языка на язык. Познакомимся с самыми известными представителями.

BERT: король понимания

BERT (Bidirectional Encoder Representations from Transformers) - двунаправленный энкодер от Google, который вышел в 2018 году и буквально перевернул мир NLP.

Суть

Предобучается на огромных объёмах текста с двумя специальными задачами: MLM и NSP.

Зачем

Глубокое понимание текста.

Применение

Определение тональности, ответы на вопросы (без генерации), распознавание именованных сущностей, классификация документов.

До появления BERT модели хорошо справлялись с конкретными задачами, но не умели по-настоящему понимать

контекст. BERT изменил это, научившись учитывать как левый, так и правый контекст каждого слова. Это позволило ему достичь человеческого уровня в ряде тестов на понимание языка и заложило фундамент для всех последующих энкодерных архитектур.

Визуализация обучения BERT

КАК УЧИТСЯ BERT

ЗАДАЧА 1: MLM (Masked Language Model)

```
Предложение:  [Кот] [ел] [сметану] [и] [спал]
                    ↓        ↓
На входе:     [Кот] [MASK] [сметану] [MASK] [спал]
                    ↓        ↓
BERT должен:  ──► "ел" ──► "и"
```

> СУТЬ: Как упражнение "вставь пропущенное слово" в учебнике.
> Чтобы угадать, нужно понять контекст слева И справа.
> Это делает BERT двунаправленным.

ЗАДАЧА 2: NSP (Next Sentence Prediction)

```
Предложение А: "Вчера шёл сильный дождь."
Предложение Б: "Поэтому я остался дома."
     Вопрос: Б является продолжением А? → ДА (IsNext)

Предложение А: "Вчера шёл сильный дождь."
Предложение Б: "Солнце — это звезда."
     Вопрос: Б является продолжением А? → НЕТ (NotNext)
```

> СУТЬ: Модель учится понимать связность текста на уровне
> предложений. Критически важно для ответов на вопросы и диалогов.

СРАВНЕНИЕ С GPT:

BERT	GPT
Видит всё сразу Восстанавливает пропущенное Понимает текст	Видит только прошлое Предсказывает следующее Генерирует текст

BERT : детали для инженера

Эволюция: от BERT к RoBERTa

BERT стал настоящей сенсацией, но исследователи быстро поняли, что его можно улучшить. Убрали лишнее, добавили данных, дольше обучали - так родилась его улучшенная версия.

RoBERTa (Robustly optimized BERT approach) - улучшенная версия BERT от Facebook.

Что изменили

· Убрали задачу NSP (оказалась не нужна)
· Больше данных (160GB вместо 16GB)
· Дольше обучали (больше шагов)
· Динамическое маскирование (маски меняются каждую эпоху)

Результат

RoBERTa стабильно превосходит BERT во всех тестах.

T5: всё в текст

T5 (Text-to-Text Transfer Transformer) - модель от Google, которая пошла ещё дальше.

Гениальная простота

Превращает **все** NLP-задачи в единый формат «текст на вход - текст на выход».

Визуализация идеи T5

```
┌──────────────────────────────────────────────────────────────────────────┐
│                          T5: ВСЁ ЕСТЬ ТЕКСТ                              │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│ ЗАДАЧА              ВХОД (текст)                    ВЫХОД (текст)        │
│ ──────────────────────────────────────────────────────────────────────── │
│ Перевод:            "перевести на английский:        "The cat is sleeping"│
│                     "Кот спит"                                           │
│                                                                          │
│ Суммаризация:       "кратко: 300 слов текста..."     "Краткое содержание"│
│                                                                          │
│ Классификация:      "тональность: Отличный           "positive"          │
│                     фильм!"                                              │
│                                                                          │
│ Вопрос-ответ:       "вопрос: Кто написал             "Пушкин"            │
│                     'Евгения Онегина'?"                                  │
│                                                                          │
│  ┌──────────────────────────────────────────────────────────────────┐    │
│  │ СУТЬ: Одна модель, один формат, любой вход — текст,              │    │
│  │ любой выход — текст. Не нужно думать о head-ах под каждую задачу.│    │
│  └──────────────────────────────────────────────────────────────────┘    │
│                                                                          │
│ Архитектура: Классический encoder-decoder трансформер                    │
│ Размеры: от 60M (small) до 11B (large) параметров                        │
└──────────────────────────────────────────────────────────────────────────┘
```

BART: лучший из двух миров

BART (Bidirectional and Auto-Regressive Transformer) - модель от Facebook, которая объединила лучшее от BERT и

GPT.

Как устроен

Энкодер как у BERT (двунаправленный) - понимает контекст

Декодер как у GPT (авторегрессионный) - генерирует текст

Гениальность BART в том, что шум можно добавлять практически любой - модель вынуждена учиться понимать структуру языка на всех уровнях: от отдельных слов до общего смысла предложения. Это делает BART особенно устойчивым к искажениям и эффективным в задачах, где входные данные могут быть «грязными» - например, при обработке пользовательского контента или распознавании речи с ошибками.

Как обучается

Берут текст, портят его разными способами (маскируют, переставляют слова, удаляют), а модель учится восстанавливать оригинал.

Сводная таблица: модели на основе трансформера

СРАВНЕНИЕ ПОПУЛЯРНЫХ МОДЕЛЕЙ				
Модель	Семейство	Архитектура	Сильные стороны	Типичные задачи
GPT (OpenAI)	Decoder-only	Авторегрессивный	Генерация, креативность, диалог	Чат-боты, генерация текста, ассистенты
BERT (Google)	Encoder-only	Двунаправленный	Понимание, анализ, извлечение	Классификация, NER, анализ тональности
RoBERTa (Meta)	Encoder-only	Улучшенный BERT	Понимание (ещё лучше)	То же, что BERT, но точнее
T5 (Google)	Encoder-Decoder	Текст-в-текст	Универсальность, один формат для всех задач	Перевод, суммаризация, вопрос-ответ
BART (Meta)	Encoder-Decoder	Денайзинг (шум → текст)	Восстановление из искажений	Суммаризация, исправление, перевод

Важное замечание про современность.

Обратная сторона медали: за что критикуют трансформеры

Было бы нечестно представить трансформер как абсолютное благо и замолчать его недостатки. У этой архитектуры есть три фундаментальные проблемы, над решением которых бьются лучшие инженерные команды мира.

Проблема 1: Квадратичная сложность внимания

Механизм самовнимания требует вычисления попарных

связей между всеми токенами. Это даёт сложность $O(n^2)$ по памяти и времени - каждый новый токен удорожает обработку квадратично. Для контекста в 10 тысяч токенов это ещё терпимо, но при 100 тысячах вычисления становятся тяжёлыми, а при миллионе - часто просто невозможными без специальных оптимизаций (о которых мы говорили в главе 4). Трансформеры плохо масштабируются на сверхдлинные контексты принципиально, на уровне архитектуры.

Проблема 2: «Вымывание» первых токенов

Даже если технически модель может принять миллион токенов на вход, на практике информация из начала последовательности теряется в шумах внимания. Первые токены «вымываются» - их влияние на финальный ответ становится пренебрежимо малым. Это не баг, а следствие того, что внимание распределяется по всей длине, и длинные хвосты просто «перекрикивают» начало.

Проблема 3: Гигантские требования к памяти

Даже с KV Cache трансформеры требуют огромного количества памяти. Веса модели (сотни гигабайт), кэш для длинного контекста (ещё гигабайты), промежуточные вычисления - всё это делает развёртывание больших моделей дорогим и сложным. Для задач, где важна работа на периферии

(на телефоне, в браузере), трансформеры часто оказываются слишком тяжёлыми.

Трансформеры - стандарт для LLM, но для задач на периферии, в реальном времени и на слабом железе RNN/LSTM/GRU и их современные аналоги (Mamba, RWKV) остаются лучшим выбором. Не выкидывайте их из инженерного арсенала

Альтернативы, которые бросают вызов трансформеру

Инженеры не сидят сложа руки. Уже есть архитектуры, которые пытаются решить эти проблемы и в некоторых задачах показывают результаты, сопоставимые с трансформерами.

State Space Models (SSM) - линейная сложность

Модели на основе пространства состояний (например, Mamba) предлагают принципиально иной подход: они обрабатывают последовательности как динамическую систему, обновляя скрытое состояние по мере чтения. Сложность - $O(n)$ вместо $O(n^2)$. Это позволяет работать с контекстами в миллионы токенов без специальных оптимизаций. Mamba показывает результаты, близкие к трансформерам, на задачах вроде моделирования языка, и при этом требует значи-

тельно меньше памяти. Плата за скорость - чуть хуже качество на задачах, где важны сложные дальние зависимости.

RWKV (Receptance Weighted Key Value) - смесь трансформера и RNN

RWKV - это гибрид, который берёт лучшее от двух миров. Как RNN, он обрабатывает последовательность линейно и не требует квадратичной памяти. Как трансформер, он использует механизмы, похожие на внимание, и может быть эффективно обучен на GPU. RWKV показывает достойные результаты при существенно меньших требованиях к ресурсам, что делает его привлекательным для развёртывания на слабом железе.

RetNet (Retentive Network) - ещё один гибрид

RetNet от Microsoft сочетает параллелизм трансформера (для обучения) с линейной сложностью рекуррентных сетей (для инференса). На больших контекстах он может быть в десятки раз быстрее трансформера при сопоставимом качестве.

Важно понимать

эти архитектуры пока уступают трансформерам не только в зрелости экосистемы, но и в стабильности обучения на

больших масштабах.

Трансформеры за более чем семь лет эволюции обросли огромным количеством инженерных хитростей - специальные инициализации, хитрые оптимизаторы, проверенные схемы нормализации, техники борьбы с нестабильностью градиентов. У Mamba, RWKV и RetNet такого багажа пока нет. При масштабировании до сотен миллиардов параметров они могут вести себя непредсказуемо: loss может «срываться» в бесконечность, градиенты - взрываться, а сходимость - требовать десятков экспериментов с гиперпараметрами. Для исследовательских проектов это приемлемо, для продакшена - риск, который нужно закладывать в бюджет.

Однако у этих альтернатив есть важное ограничение: **экосистема**.

Вокруг трансформеров выстроены тысячи инструментов, библиотек, фреймворков и предобученных моделей. Для Mamba или RWKV пока нет такого же богатства готовых решений, оптимизированных ядер, инструментов для дообучения и развёртывания. Если вы строите экспериментальный проект или работаете на периферии - это не проблема. Если вам нужен стабильный продакшен с поддержкой сообщества - трансформеры всё ещё вне конкуренции.

Так умрёт ли трансформер?

Скорее всего, нет. Трансформеры стали стандартом де-факто, вокруг них выстроена гигантская экосистема инструментов, библиотек и обученных моделей. Но в задачах, где критичны сверхдлинные контексты или работа на периферии, альтернативы уже начинают вытеснять трансформеры. Скорее всего, будущее - за гибридными подходами, которые возьмут лучшее от обеих архитектур.

Для инженера это означает, что выбор архитектуры зависит от задачи:

· Если нужен универсальный солдат и есть бюджет - берите трансформер.
· Если нужно обрабатывать миллионы токенов или работать на слабом железе - присмотритесь к Mamba или RWKV.
· Если вы строите систему с нуля и готовы экспериментировать - следите за новыми SSM-архитектурами, они развиваются стремительно.

Все эти модели - **плотные** трансформеры, где каждый нейрон участвует в каждом вычислении. Они стали фундаментом, на котором строится современный ИИ. Но инженеры заметили, что разные задачи требуют разных знаний. Что если не заставлять одну модель учиться всему сразу, а со-

брать команду специалистов?

Так появилась архитектура, которая позволила перешагнуть порог в триллион параметров - **смесь экспертов (Mixture of Experts, MoE)**. О ней мы подробно поговорим в следующем разделе.

2.6 MoE (Mixture of Experts / Смесь экспертов)

По мере того как модели росли, росла и цена их обучения и использования. GPT-3 со своими 175 миллиардами параметров уже стоил десятки миллионов долларов. Но мир хотел большего. И тут инженеры вспомнили старую идею: а что если не активировать все нейроны сразу, а включать только те, которые нужны для конкретной задачи? Так родилась архитектура MoE - Смесь экспертов.

Трансформер, в котором слои FFN заменены на множество параллельных экспертов (experts). Входной токен направляется не во все эксперты, а только в несколько, выбранных маршрутизатором (router). Это позволяет наращивать общее число параметров (до триллионов), сохраняя вычислительную стоимость активных параметров на уровне небольшой модели.

MoE - это трансформер, где вместо одного «толстого» специалиста работают десятки узких экспертов, а умный диспетчер решает, к кому обратиться за конкретным токеном.

Идея (метафора)

Представьте клинику, где вместо одного терапевта наняли 100 узких специалистов: окулист, кардиолог, невролог и т.д. Когда приходит пациент (токен), администратор (роутер) быстро определяет, к кому его направить - например, только к окулисту и кардиологу. Остальные 98 специалистов в это время отдыхают.

В штате клиники числится 100 врачей (общие параметры), но на каждого пациента работают только двое (активные параметры). Клиника может позволить себе лучших экспертов, но платит зарплату только за фактически отработанные часы.

В этой метафоре скрыт главный секрет MoE: общий штат врачей (параметров) может быть огромным, но на каждого пациента работают только двое. Это даёт качество большой модели при скорости маленькой.

Как это устроено технически

В обычном трансформере каждый токен проходит через один и тот же слой FFN. В MoE вместо одного FFN ставится множество параллельных FFN - это и есть эксперты. Добавляется маршрутизатор - обычно это небольшой линейный слой с softmax, который для каждого токена вычисляет распределение вероятностей по экспертам и оставляет только K лучших (обычно K=1 или 2). Выход токена - взвешенная сумма выходов выбранных экспертов.

Звучит красиво, но на практике инженеры столкнулись с неожиданной проблемой: маршрутизатор иногда «влюблялся» в двух-трёх экспертов и отправлял всех пациентов только к ним. Остальные простаивали, память была занята, а пользы - ноль.

Ключевой трюк: маршрутизатор активирует **не всех** экспертов, а только **K самых релевантных** (обычно K=1 или 2). Остальные эксперты для этого токена простаивают, экономя память и вычисления.

Зачем это нужно

Главная цель MoE

Масштабирование без взрывного роста затрат. Можно сделать модель с триллионом параметров (огромная «копилка знаний»), но на каждом шаге обрабатывать токен лишь небольшой частью сети. Это даёт качество большой модели при скорости маленькой.

Пример

У Mixtral 8x7B общее число параметров - 47 млрд, но при обработке токена работает только 13 млрд (2 эксперта из 8). Формально это огромная модель, фактически - средняя по затратам.

Где используется

LLM-гиганты

GPT-4 (предположительно 16 экспертов), Mixtral (8 экспертов), DeepSeek-V3 (256 экспертов, но активны только 6), Grok.

Мультимодальность

можно сделать экспертов под текст, под изображения, под звук - роутер сам направит данные куда надо.

Главная проблема и её решение

Проблема

Роутер может «полюбить» двух-трёх экспертов и отправлять токены только к ним. Остальные не обучаются, превращаются в «мёртвый груз» - память занята, пользы нет.

Решение

инженеры добавляют **штраф за неравномерность** в функцию потерь или динамически балансируют нагрузку, подкручивая предпочтения роутера. В современных моделях (DeepSeek) это делается без штрафов, через хитрую динамику. Балансировка нагрузки достигается комбинацией вспомогательной функции потерь (auxiliary load-balancing loss) и динамического роутинга с температурой, которая

адаптируется в процессе обучения для предотвращения коллапса на 1–2 экспертах. Полный отказ от регуляризации встречается редко, так как ведёт к коллапсу экспертов.

Инженерная деталь

эта балансировка чаще всего реализуется через дополнительный штраф (auxiliary loss) в функции ошибки. Если какой-то эксперт получает слишком мало токенов, модель штрафуется. Это заставляет роутер использовать всех экспертов равномерно, иначе часть параметров модели превратится в «мертвый груз» - будет занимать память, но не обучаться.

MoE - это способ сделать модель «широкой, но не глубокой» в вычислительном смысле. Много экспертов = много знаний, мало активных = быстрая работа. Архитектура, которая позволила индустрии перешагнуть порог в триллион параметров, не сломав бюджет на инференс.

Мы рассмотрели архитектуры для текста и чисел. Но есть ещё одна важная область, где ИИ добился потрясающих результатов - генерация изображений. Там правят бал не трансформеры (хотя они тоже начинают проникать), а диффузионные модели. О них - в следующем разделе.

2.7 Diffusion Models (Диффузионные модели)

Мы рассмотрели архитектуры для текста, чисел и последовательностей - от простых RNN до гигантских MoE-трансформеров. Но есть ещё одна важная область, где ИИ добился потрясающих результатов, - генерация изображений. Долгое время здесь правили бал GAN-ы[6] (Generative Adversarial Networks), но несколько лет назад произошла тихая революция. На сцену вышли диффузионные модели - те самые, что лежат в основе Midjourney, Stable Diffusion и Kandinsky.

Диффузионные модели обучаются постепенно добавлять шум к данным, а затем восстанавливать исходное распределение, убирая шум. Используются для генерации изображений.

Название «диффузия» пришло из физики, но не пугайтесь - идея на удивление проста и красива. Представьте, что вы учите модель не рисовать с нуля, а постепенно убирать шум с зашумлённой картинки, шаг за шагом проявляя изображение.

1. Логика идеи (Откуда взялось название)

Термин пришёл из физики (термодинамика). Если капнуть чернила в стакан с водой, они постепенно «растворятся» и равномерно распределятся - это процесс **прямой диффузии** (порядок → хаос).

Диффузионная модель делает обратное: **берёт хаос (шум) и организует его в картинку**.

По шагам это выглядит так

· Берём реальную фотографию кота.

· Медленно, шаг за шагом, добавляем к ней случайный шум (как помехи на старом телевизоре).

· Через много шагов картинка превращается в чистое «снежное поле» - абстрактный шум.

Модель обучается брать этот шум и проходить процесс в обратную сторону: убирать шум шаг за шагом, восстанавливая кота.

2. Как это устроено технически (коротко)

У модели два процесса

Процесс	Что происходит?	Зачем?
Forward diffusion (Прямой процесс)	К картинке последовательно добавляется небольшой гауссов шум согласно фиксированному расписанию (noise schedule). Через T шагов (обычно 1000) изображение превращается в чистый гауссов шум. Модель ничего не делает - это просто математическая операция.	Подготовить данные: показать модели, как выглядит «порча» изображения на каждом этапе.
Reverse diffusion (Обратный процесс)	Нейросеть (обычно U-Net7) обучается предсказывать, какой именно шум был добавлен на этом шаге, и удалять его, постепенно восстанавливая исходное изображение из случайного шума. Весь процесс обратной диффузии (сэмплинг) занимает десятки или сотни таких итеративных шагов очистки.	Это и есть обучение. Модель учится догадываться, какая картинка была до того, как её испортили.

Аналогия с реставрацией

Представьте, что вам дали сильно повреждённую фреску (почти стёртую в пыль). Вы смотрите на неё и говорите: «Здесь, судя по фактуре, должна быть рука, а здесь - кусочек неба». Диффузионная модель делает то же самое, только с шумом.

3. Практическая сторона (Как это используют)

Генерация по тексту (Text-to-Image)

Когда вы пишете «красный дракон в стиле киберпанк», происходит следующее:

Текст → вектор

Ваш запрос превращается в эмбеддинг (как в языковых моделях).

Управление шумом

Модель начинает со случайного шума. На каждом шаге уборки шума она сверяется с текстовым вектором и спрашивает: «Соответствует ли то, что я сейчас восстанавливаю, слову "дракон"?». Если нет - корректирует.

Итерации

Проходит, например, 50 шагов очистки, и на выходе получается картинка.

Где применяется

· **Stable Diffusion, Kandinsky, Midjourney:** рисование по тексту.

· **Inpainting (дорисовка):** закрасили часть картинки - модель дорисует фон так, что не отличить.

· **Super-resolution:** повышение разрешения старых фотографий (модель «угадывает» детали, которых не было).

· **Видео и 3D:** генерация плавных переходов между кадрами или создание трёхмерных объектов.

4. Почему именно диффузия победила GAN-ы?

Раньше картинки генерировали через GAN, где две сети соревнуются: одна подделывает, другая ловит. Это работало, но было нестабильно (режим коллапса, когда сеть рисовала одно и то же).

Диффузия дала

Стабильность обучения
просто предсказываем шум - простая функция потерь (MSE, Mean Squared Error - средняя квадратичная ошибка), никаких соревнований.

Разнообразие
может сгенерировать бесконечное количество вариантов, потому что стартует с разного случайного шума.

Контроль
легко «подмешивать» условия (текст, контур, другую картинку) на каждом шаге очистки.

Важный нюанс
GAN-ы не умерли полностью. В задачах, где требуется фотореалистичная генерация лиц или конкретных объектов

с минимальными искажениями, GAN-ы до сих пор держат позиции (например, StyleGAN3). Диффузия победила в массовых текстово-картиночных сервисах (Midjourney, Stable Diffusion) благодаря своей стабильности и контролируемости, но в узких нишах GAN-ы продолжают использоваться.

Диффузионная модель

это генератор, который учится **превращать шум в осмысленное изображение**, проходя множество маленьких шагов очистки. На каждом шаге она сверяется с текстовым запросом, постепенно проявляя детали. Это медленнее, чем одна проходка нейросети (нужно 20–50 последовательных шагов), зато результат получается детальным и разнообразным.

2.8 За рамками текста: как трансформеры научились видеть

Мы разобрали, как трансформеры работают с текстом. Но та же архитектура - с минимальными изменениями - применяется к изображениям, звуку и даже видео. И ключевая идея здесь: **превратить «не-текст» в последовательность, похожую на токены**.

Как это работает для картинок (Vision Transformer, ViT)

Представьте, что вы берете фотографию и разрезаете её на маленькие квадратики одинакового размера - например, 16×16 пикселей. Каждый квадратик (патч) - это аналог токена. В отличие от текста, у патчей нет заранее заготовленного словаря.

Но есть важный нюанс: сырой патч - это просто 16×16×3 = 768 чисел (яркости красного, зеленого и синего для каждого пикселя). Сами по себе эти числа говорят только о цвете. Чтобы модель могла работать с патчем, этого мало - ей нужно больше «места» для записи сложных признаков: есть ли здесь граница? Похоже ли это на глаз? Какая текстура?

Поэтому каждый патч пропускается через **линейный слой**, который превращает 768 чисел в вектор гораздо большей размерности (например, 4096). Это не сжатие, а **расширение**. Модель сама учится заполнять эти 4096 чисел так, чтобы в них помещались все важные признаки патча: форма, структура, отношение к соседям. Это как если бы вы попросили эксперта описать квадратик не тремя словами («красный, круглый, яркий»), а 4096 словами - в таком описании поместится всё, что нужно.

Дальше - всё как в текстовом трансформере

К каждому вектору добавляется информация о позиции (чтобы модель знала, где был этот квадратик - в левом верхнем углу или в правом нижнем).

Механизм внимания ищет связи между патчами: «этот квадратик похож на глаз, а этот - на нос, и они должны быть рядом».

Многослойные блоки строят иерархию признаков: от простых (границы, текстуры) до сложных (ухо, хвост, морда).

Почему это важно понимать

Трансформеры не «заточены» под текст. Они универсальны. Любые данные, которые можно разбить на последовательность кусочков (патчей), можно скормить трансформе-

ру. Разница лишь в том, как мы получаем эти кусочки и как превращаем их в начальные векторы.

Для звука - это короткие фрагменты аудиодорожки. Для видео - последовательность кадров или пространственно-временные патчи. Для генома - участки ДНК.

И везде работает один и тот же принцип: сырые данные превращаются в векторы, размерность которых выбирается инженером. Больше размерность - больше «места» для сложных признаков, но медленнее работа. Меньше - экономия памяти, но риск потерять детали.

Метафора

Трансформер - это универсальный читатель. Ему всё равно, на каком языке написана книга: словами, пикселями или нотами. Но перед чтением ему нужно, чтобы каждую «букву» описали не одним словом, а целым абзацем - так, чтобы в этом описании поместились все нюансы. Линейный слой - это и есть тот самый «переводчик», который превращает сырой пиксель в такой подробный абзац-вектор.

Итак, мы совершили обзор основных архитектур - от свёрточных сетей до диффузионных моделей, а также заглянули за рамки текста, чтобы понять, как те же принципы работают с изображениями. Теперь вы понимаете, чем они отличаются и для каких задач предназначены. Но архитектура - это только «железо» модели. Чтобы она заработала, её

нужно обучить. А обучение современных моделей - это отдельная инженерная вселенная со своими этапами, ценами и компромиссами. Об этом - в следующей главе.

РЕЗЮМЕ ГЛАВЫ 2

CNN (свёрточные сети) используют скользящее окно с обучаемыми фильтрами; эффективны для изображений, но не для текста из-за отсутствия пространственной структуры.

RNN (рекуррентные сети) обрабатывают текст последовательно, ведя блокнот-память, но страдают от невозможности параллелизации и затухания градиентов.

LSTM добавляет три вентиля (забывания, входной, выходной) и «ленту конвейера» (cell state), решая проблему долгой памяти, но остаётся последовательной.

GRU - упрощённая LSTM с двумя вентилями (reset и update), на 25–30% меньше параметров, чуть быстрее, но чуть хуже на сверхдлинных последовательностях.

Трансформер обрабатывает все токены параллельно через механизм самовнимания. Однако у трансформера есть фундаментальные недостатки: квадратичная сложность внимания ($O(n^2)$), «вымывание» первых токенов на длинных контекстах и гигантские требования к памяти.

Decoder-only (GPT, LLaMA) - генерирует текст, глядя только на прошлые токены; основа всех современных чатботов.

Encoder-only (BERT, RoBERTa) - понимает текст целиком (двунаправленное внимание), но не генерирует; используется для анализа и классификации.

Encoder-Decoder (T5, BART) - преобразует одну последовательность в другую; идеален для перевода и суммаризации.

MoE (смесь экспертов) заменяет один FFN множеством экспертов, активируя только 1–2 из них; позволяет иметь триллионы параметров при умеренных затратах на инференс.

Диффузионные модели учатся постепенно убирать шум с картинки; победили GAN-ы в text-to-image благодаря стабильности и контролю (Midjourney, Stable Diffusion).

Альтернативы трансформеру - State Space Models (Mamba) и гибриды (RWKV, RetNet) предлагают линейную сложность O(n) и работу с контекстами в миллионы токенов. Плата - чуть хуже качество на задачах со сложными дальними связями.

Экосистема имеет значение. У трансформеров - тысячи готовых инструментов, библиотек и фреймворков. Альтернативы пока уступают в зрелости: для них меньше предобученных моделей, хуже документация и почти нет корпоративной поддержки. Для экспериментальных проектов это неважно, для продакшена - критично.

Гибридные подходы набирают силу. Некоторые новые архитектуры (Jamba, S4) пытаются комбинировать лучшие черты трансформеров и SSM: брать внимание для сложных зависимостей и линейную сложность для длинных контекстов. Скорее всего, будущее именно за гибридами.

Инженерный вывод

Выбор архитектуры зависит от задачи. Для универсальных решений с бюджетом - трансформер. Для сверхдлинных контекстов или работы на периферии - Mamba или RWKV. Для продакшена с жёсткими требованиями к памяти - гибриды. Трансформер остаётся стандартом, но не единственным игроком на поле.

Трансформеры универсальны: они работают не только с текстом. Изображение можно разрезать на квадратики (патчи), каждый патч превратить в вектор через линейный слой, добавить информацию о позиции - и подать в тот же транс-

формер. В отличие от текста, у патчей нет заранее заготовленного словаря, а размерность вектора выбирается инженером: больше - больше «места» для сложных признаков, но медленнее; меньше - быстрее, но можно потерять детали. Этот принцип лежит в основе Vision Transformer (ViT) и генеративных моделей вроде Stable Diffusion.

Вопросы для самопроверки

1. Почему CNN, отлично работающие с изображениями, плохо подходят для обработки текста?

2. В чём проявляется «забывчивость» простых RNN и как LSTM пытается эту проблему решить?

3. Какие недостатки LSTM устраняет GRU и какой ценой?

4. Три семейства трансформеров: decoder-only, encoder-only, encoder-decoder. Приведите примеры задач для каждого.

5. Какие фундаментальные недостатки есть у трансформеров и как альтернативные архитектуры (SSM, RWKV) пытаются их обойти?

6. Объясните метафору «клиники с узкими специалистами» применительно к архитектуре MoE.

7. Чем диффузионные модели принципиально отличаются от GAN? Почему диффузия победила в text-to-image?

8. Как трансформеры обрабатывают изображения? Что такое патч и почему его нельзя заменить токеном из словаря? Почему вектор патча обычно больше, чем количество пикселей в нем?

ГЛАВА 3. ОБУЧЕНИЕ МОДЕЛЕЙ

В предыдущей главе мы рассмотрели «железо» - различные архитектуры нейросетей, от свёрточных до диффузионных. Но архитектура - это только скелет. Чтобы модель заработала, её нужно обучить, вложить в неё знания.

И вот здесь начинается самое интересное: обучение современных языковых моделей - это не один этап, а целый конвейер.

Три ключевых этапа

1. Pre-training (предварительное обучение) - модель читает интернет и учится предсказывать следующее слово. Это фундамент, но он стоит десятков миллионов долларов.

2. SFT (Supervised Fine-Tuning, дообучение с учителем) - модель тренируется на парах «вопрос → идеальный ответ». Превращает «энциклопедию» в ассистента.

3. RLHF (Reinforcement Learning from Human Feedback, обучение с подкреплением на основе обратной связи от человека) - люди показывают модели, какие ответы хорошие. Делает модель вежливой и безопасной.

Этапы обучения: что происходит, во сколько обходится, к чему приводит

Этап	Что делает	Стоимость (оценка)	Результат
Pre-training	Учит язык и факты на триллионах токенов	$15–150 млн	Базовая модель
SFT	Учит отвечать в формате диалога на тысячах примеров	$5–50 тыс (аренда GPU) + разметка	Ассистент
RLHF	Учит быть вежливым и безопасным на предпочтениях людей	$50–500 тыс + разметка	Вежливый ассистент

В этой главе мы разберём каждый этап. Узнаем, сколько это стоит, почему данные важнее архитектуры, и почему RLHF не делает модель «объективной», а вшивает в неё ценности разработчика.

3.1 Pre-training (Предварительное обучение)

Представьте, что вы хотите вырастить специалиста широкого профиля. Сначала вы отправляете его в библиотеку читать всё подряд - тысячи книг, энциклопедий, статей. Он ещё не знает, какие именно вопросы ему будут задавать, но впитывает информацию, учится структуре языка, фактам, логике. Это и есть предварительное обучение.

Звучит просто: бери огромный корпус текстов и учи модель предсказывать следующее слово. Но за этой простотой скрывается инженерный подвиг, доступный лишь единицам. Давайте прикинем, что значит «огромный» в цифрах.

Модель обучается на огромных неразмеченных корпусах (триллионы токенов). Задача - предсказание следующего токена (next token prediction). Это обучение без учителя (self-supervised learning) - правильные ответы уже есть в данных. Цель: выучить статистику языка, факты, логику повествования.

Pre-training - это не просто "много данных". Это инженерный подвиг, который могут позволить себе единицы.

Грязный секрет больших моделей: данные важнее

архитектуры

Когда говорят о триллионах токенов, часто забывают о качестве этих токенов. А зря. **Мусор на входе - мусор на выходе** работает для нейросетей даже жёстче, чем для классического программирования. Можно собрать 100 терабайтов текста, но если там 30% - рекламный спам, 20% - машинный перевод с ошибками, а 10% - откровенно токсичный контент, модель выучит этот мусор так же хорошо, как и правильный язык.

Проблемы с данными, с которыми сталкиваются инженеры

Дубликаты

Один и тот же текст может встречаться в корпусе тысячи раз (например, типовые пользовательские соглашения). Модель переобучается на этих дубликатах, считая их важнее, чем уникальный контент.

Низкокачественный контент

Форумы с бессмысленными сообщениями, автоматически сгенерированные страницы, рекламный спам - всё это засоряет корпус и учит модель галлюцинировать.

Языковой дисбаланс

Английского в интернете в разы больше, чем всех остальных языков вместе взятых. Модель, обученная на таком корпусе, будет говорить по-английски отлично, а на других языках - со странным акцентом и ошибками.

Явный и неявный токсичный контент

Если не фильтровать корпус, модель выучит все предрассудки, ненависть и стереотипы, которые есть в интернете. RLHF потом это исправляет, но зачем создавать себе лишнюю работу?

Что делают инженеры с данными перед обучением

Дедупликация

удаляют повторяющиеся тексты (на уровне документов, абзацев и даже предложений).

Фильтрация по качеству

используют классификаторы, чтобы отсеять спам, машинный перевод, бессмысленный контент.

Балансировка языков

если английского слишком много, его искусственно ограничивают, чтобы другие языки получили достаточно внимания.

Токенизация с учётом языков

подбирают размер словаря и алгоритм так, чтобы редкие языки не разбивались на слишком мелкие токены.

Safety -фильтры

удаляют откровенно токсичный или нелегальный контент (насколько это вообще возможно для интернет-масштабов).

Практический вывод

Архитектура определяет потолок возможностей модели, но данные определяют, достигнет ли модель этого потолка. Две одинаковые архитектуры, обученные на разных корпусах, могут отличаться по качеству в разы. Поэтому, когда вы слышите «мы обучили модель на 10 триллионах токенов», всегда стоит спросить: «А что это были за токены? Сколько из них - мусор? Как вы чистили данные?» Часто ответы на эти вопросы объясняют разницу в качестве лучше, чем сравнение архитектур.

Речь идёт не только о сборе данных, но и о создании инфраструктуры, способной переварить эти объёмы. Кластеры из тысяч GPU работают месяцами, потребляя энергию малого города. Сбои оборудования, потери данных, нестабильность обучения - всё это часть ежедневной рутины инженеров, которые тренируют большие модели. Каждая такая тренировка - это лотерея: даже при идеальных настройках никто

не гарантирует, что модель "сойдётся" в нужную сторону.

Но что стоит за сухими цифрами «триллионы токенов»? Давайте переведём их в понятные инженеру величины - часы работы GPU, счета за электричество и седые волосы дата-центрщиков.

Конец ознакомительного фрагмента.