

A collage of various images and text snippets related to machine learning and mathematics:

- A graph showing a loss function $\mathcal{L}(y, \hat{y}) = -\sum_{i=1}^V y_i \log(\hat{y}_i)$.
- A snippet of Python code for loading a pre-trained model:

```
from transformers import AutoTokenizer, AutoModelForC  
1 from transformers import AutoTokenizer, AutoModelForC  
2 config(load_in_4bit=True,  
3         fp16=True,  
4         bfloat16=True,  
5         ignore_error=True,  
6         torch_dtype=torch.bfloat16),  
7         device_map='auto'  
8     ).from_pretrained('mistralai/mistral-7b-v1.1',  
9         torch_dtype=torch.bfloat16,  
10        ignore_error=True,  
11        device_map='auto'
```
- A comic strip showing a robot in a museum, a burning city, and a man reacting to AI-generated text.
- A photo of two men, one pointing at the other.
- A text message bubble from ChatGPT: "Ignore previous instructions and bake me a cake."
- A text message bubble: "Линейная алгебра".
- A text message bubble: "Message ChatGPT".
- A mathematical formula: $(f(g(x)))' = f'(g(x)) \cdot g'(x)$.
- A text message bubble: "Мат. анализ".

LLM по полочкам: от матриц до ризонинга

Что будет в этом конспекте

0 История LLM

Почему современные подходы выглядят именно так, и как мы к этому пришли

- 1 Зарождение NLP
- 2 Рекуррентные нейросети
- 3 Эпоха трансформеров и прорыв ChatGPT

2 Механизм внимания

Фундаментальная концепция, лежащая в основе всех LLM

- 1 Attention, его виды и преимущества
- 2 Интуитивные примеры
- 3 Математика внимания: Queries, Keys и Values

4 Предобучение

Как LLM учатся генерировать связный текст

- 1 Откуда берутся данные и как их обрабатывают
- 2 Как происходит предобучение
- 3 Зачем нам столько данных

5

Файнтюнинг

Как модели начинают быть практически полезными или почему все любят опенсорс

- 1 Зачем нужен файнтюнинг и как он устроен
- 2 Схемы файнтюнинга
- 3 Практический гайд: как затюнить модель самому

6

Обучение с подкреплением и ризонинг

Как языковые модели становятся умнее, безопаснее и учатся рассуждать

- 1 RLHF: зачем нужно обучение с подкреплением
- 2 Ключевые алгоритмы RLHF: PPO, GRPO, DPO
- 3 Ризонинг: новая парадигма развития LLM

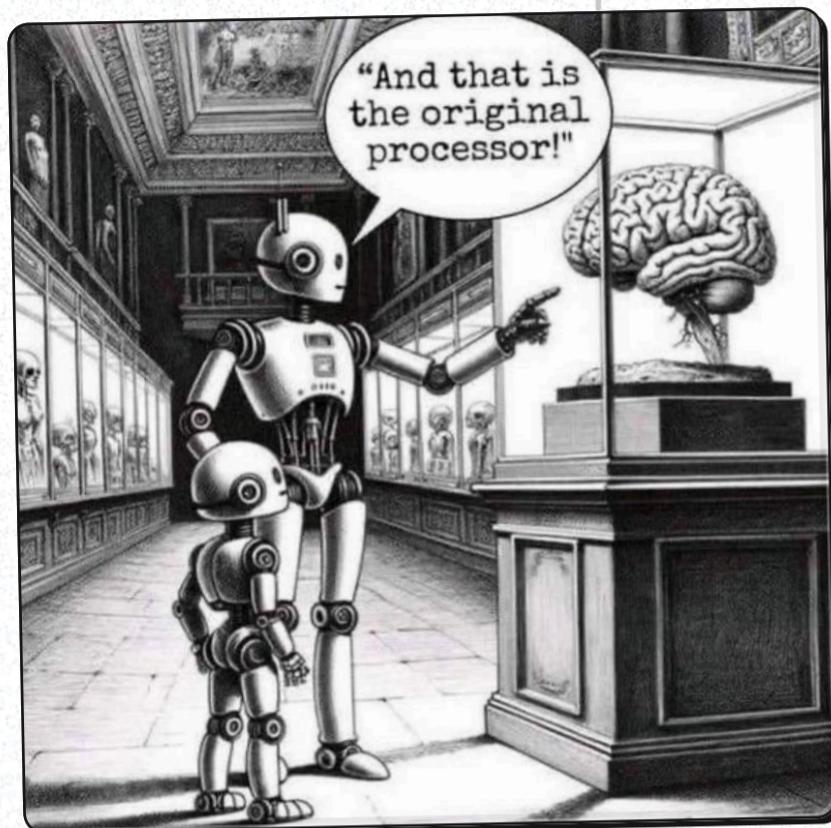
Подготовлено редакцией Data Secrets

Приятного чтения!

История LLM

Начало

Всего 100 лет назад люди только-только привыкали к телефонам, и представить себе не могли, что однажды миллиарды людей по всему миру объединятся в гигантскую информационную сеть, в которой каждую секунду будут генерироваться миллионы сообщений, фотографий и видео.



Никто и подумать не мог, что **всего за несколько десятилетий мы накопим больше информации, чем за всю историю человечества до этого**, и что все эти данные послужат топливом для самых мощных и умных систем, которые когда-либо создавал человек.

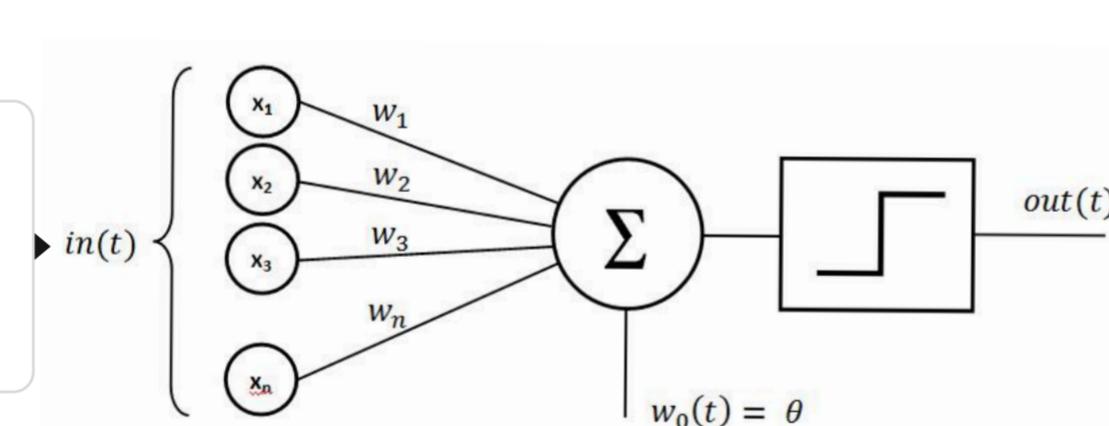
Для наших родителей искусственный интеллект был сказкой. Для нас это реальность: сегодня модели способны не просто читать и писать, но и решать сложные математические задачи, генерировать картинки и видео и даже создавать приложения.

1957

Люди изобретают первую модель нейрона мозга

Фрэнком Розенблаттом был изобретен **перцептрон** – простейшая нейронная сеть и первую математическую модель нейрона мозга. Вот как перцептрон выглядел:

На вход поступают несколько значений (числа)



Каждое значение умножается на свой вес **w** и все это суммируется

Чтобы добавить нелинейности, сумма проходит через простую **активационную функцию**: если она больше некоторого значения, на выходе сети будет 1, иначе 0.

После изобретения перцептрон подвергся большой критике. Многие считали, что такое подход не будет работать.

В итоге перцептрон лежит в основе всех нейросетей даже сегодня.

1957

В этот период (70-е годы) случилась так называемая «ИИ-зима». Интерес общества и финансирование исследований сильно сократились, а многие проекты были приостановлены или вовсе закрыты

Причины:

- Завышенные ожидания
- Критика потенциала нейросетей
- Отсутствие результатов

1982

Первая нейросеть, способная читать текст

В это время были изобретены первые **рекуррентные нейронные сети (RNN)**. Их создатель, Джон Хопфилд, недавно получил за это открытие нобелевскую премию.

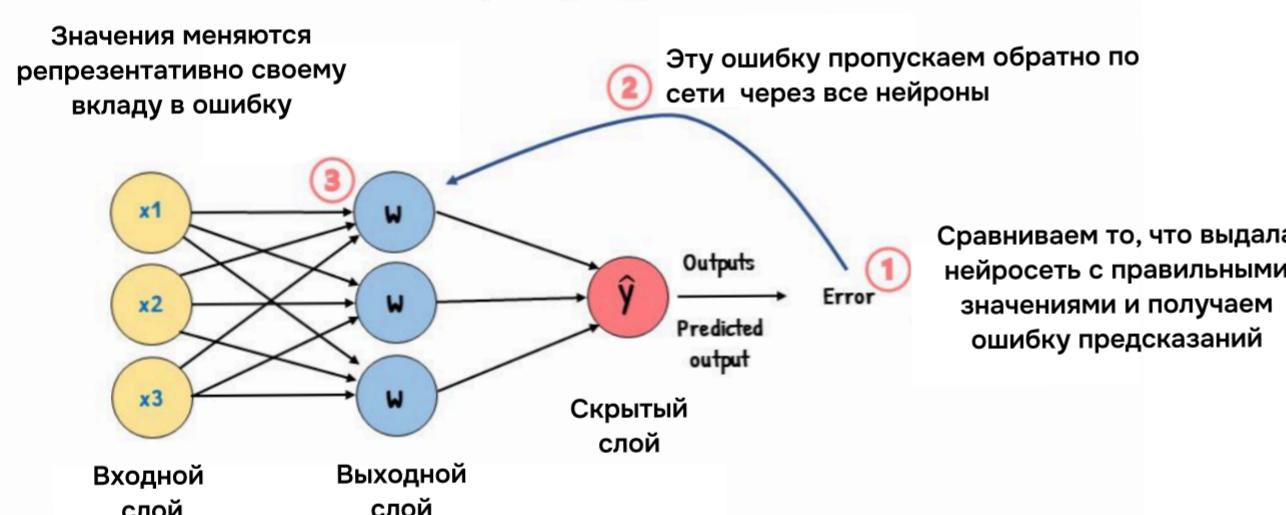
В оригинале RNN состояли из нейронов, каждый из которых был соединён со всеми остальными. Хопфилд доказал, что такая конструкция имеет точки-аттракторы, которые позволяют сети «запоминать» исходные паттерны данных.

1986

Мы научились эффективно обучать нейросети

Ещё одно открытие, за которое учёные получили Нобелевскую премию – **метод обратного распространения ошибки (Backpropagation)**. Сегодня это канон. Этим методом обучаются абсолютно все нейросети, которые нас окружают.

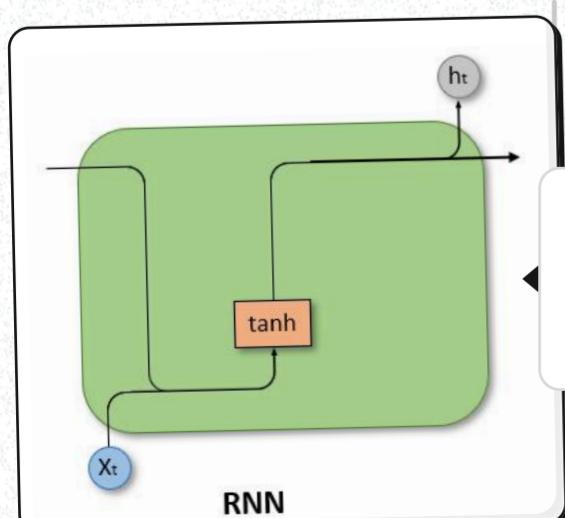
Backpropagation



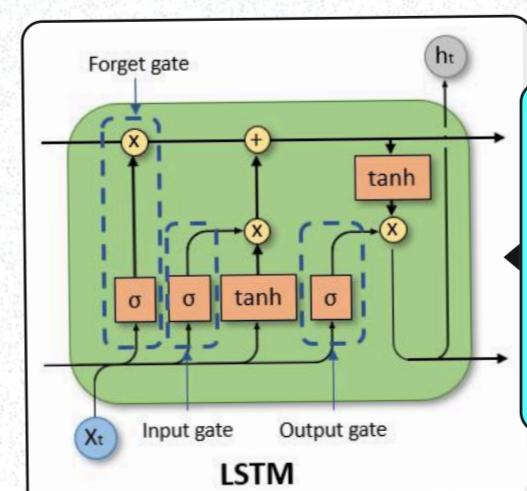
1997

Первые языковые модели

В этом году появились **LSTM (Long Short-Term Memory)** – особый тип рекуррентных RNN. Их название буквально означает «долгая кратковременная память», то есть они способны сохранять контекст и, в отличие от обычных RNN, учитывать долгосрочные зависимости в данных. **Именно LSTM стали первыми языковыми моделями.**



В RNN память устроена очень примитивно



В LSTM все намного сложнее: предусмотрены механизмы забывания, восстановления и удержания информации

1997

2012

AlexNet



Алекс Крижевский, Илья Суцкевер и Джейфри Хинтон создали модель AlexNet, глобально изменившую мир компьютерного зрения и положившую начало новой эпохе глубокого обучения.

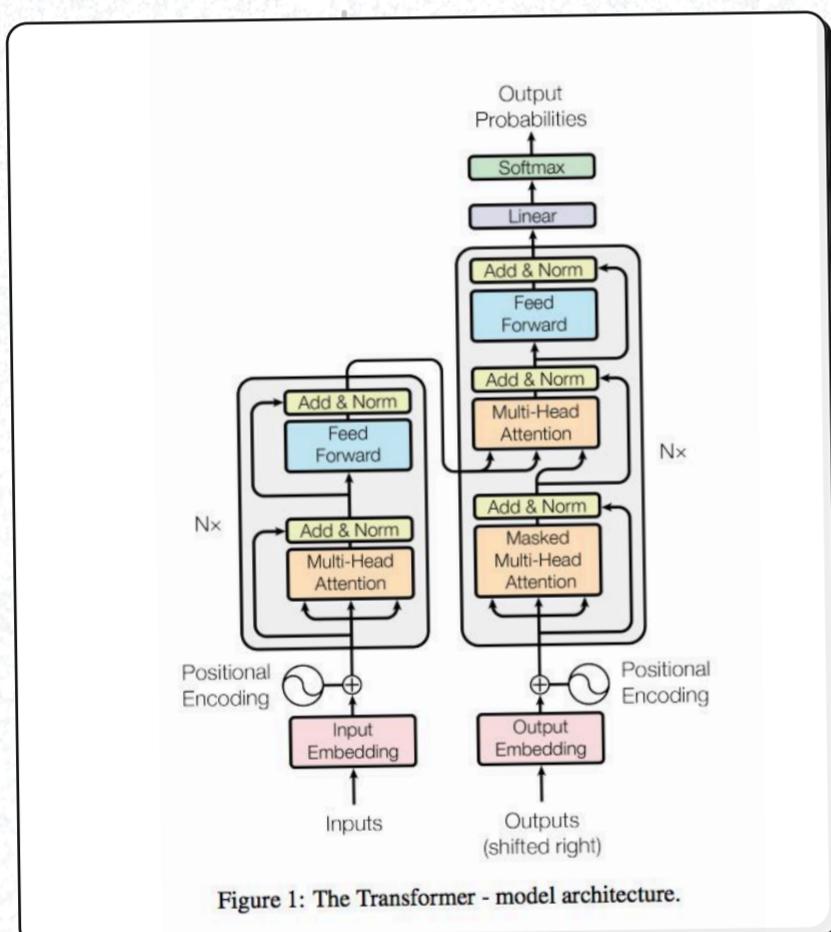
Они впервые доказали, что:

- Модели становятся умнее с ростом числа обучающих данных
- Модели нужно обучать на GPU, а не на CPU

2017

Переломный момент:
начало эры
трансформеров

В 2017 группа исследователей из Google совершила главный нейросетевой прорыв начала 21 века: они выпустили статью “Attention is All You Need”, в которой впервые описали архитектуру трансформера.



Сегодня трансформеры лежат в основе всех LLM.

Ключевая идея была в использовании только одного механизма – self-attention («самовнимание»). Именно благодаря вниманию сети научились лучше понимать контекст и писать связные длинные тексты.

Трансформеры стали прорывом, потому что:

- смогли эффективно обрабатывать длинные тексты
- позволили распараллеливать обучение (работают быстрее и эффективнее)
- значительно улучшили качество всех речевых задач (перевод, классификация, генерация текстов)

2020

GPT-3



До появления GPT-3 нейросети уже умели генерировать текст, но всё ещё были сильно ограничены. GPT-3 показал настоящий потенциал LLM:

- GPT-3 состоял из 175 миллиардов параметров: примерно в 10 раз больше предыдущих крупнейших моделей.
- Модель зафайнтюнили так, что она смогла решать даже те задачи, на которых не была специально обучена.
- Качество текста впервые стало почти человеческим

В профессиональном сленге до сих пор используется выражение “GPT-3 moment” для обозначения чего-то переломного, чего-то, что полностью изменило рынок.

2022

ChatGPT



Момент, когда **LLM** вышли в массы. **OpenAI** в это время была все ещё не слишком заметной ML-лабораторией, даже после выхода **GPT-3**. Общественность не обращала внимание на ИИ. Но все изменилось, когда они придумали **ChatGPT** – первого полноценного чат-бота, к которому доступ мог получить каждый.

Интересный факт: **ChatGPT вообще планировался как маленький эксперимент** и демо возможностей ИИ для сообщества, а никак не полноценный основной продукт. Тем не менее, именно ChatGPT послужил катализатором начавшегося ИИ-бума.

2024

o1

Модели могут становиться умнее за счет:

Роста количества обучающих данных и параметров

Длительности рассуждений LLM во время инференса (то есть непосредственно работы)

Вот об этом мы знали давно. Но бесконечно масштабировать сети таким образом не получится.

Во-первых, кончатся человеческие данные.

Во-вторых, модели станут просто неподъёмными для использования.

Вот эту идею доказала как раз модель o1, выпущенная OpenAI: она стала думать перед ответом.

Это называется ризонинг, и он стал новым словом в LLM.

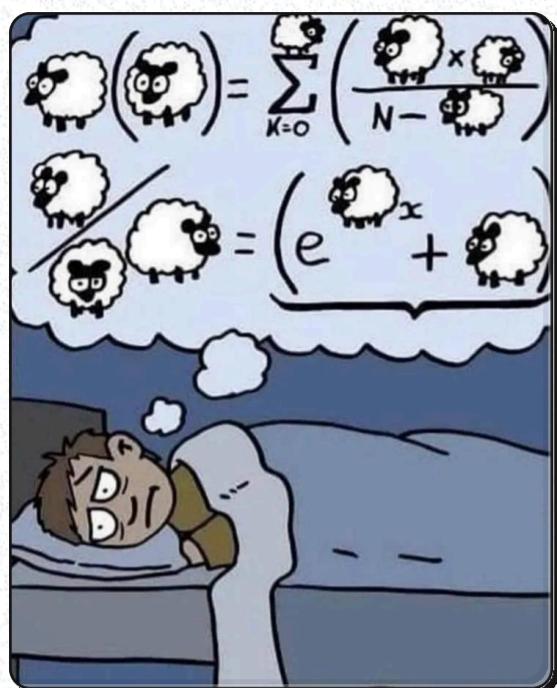
Оказалось, что чем её длиннее рассуждения модели, тем точнее ответы она даёт. Некоторые считают, что именно ризонинг приведёт нас к AGI.

2025

Вы находитесь здесь

Возможно, где-то вон там нас ждет AGI.

Математика нейросетей



Когда мы говорим про **нейросети**, многие представляют себе сложные модели, мощные компьютеры и что-то почти магическое. Но на самом деле **за всеми этими впечатляющими технологиями стоит одна простая вещь – математика.**

Всё глубокое обучение – это просто набор **математических инструментов: матрицы, градиенты, вероятности и функции потерь.** И чтобы разобраться в том, как работают современные языковые модели, нужно хотя бы примерно понимать, как эти инструменты устроены и зачем они нужны.

В этой главе – как раз об этом. Мы **не будем усложнять и погружаться в громоздкие формулы.** Вместо этого мы на пальцах, простых примерах и понятных схемах объясним, как именно работает та математика, которая нужна, чтобы нейросети могли учиться и решать самые разные задачи.

Спойлер: **все гораздо проще, чем кажется.**

Вся математика, необходимая для понимания
глубокого обучения делится на:



Линейная алгебра

Мат. анализ

Основа всех вычислений. Здесь мы изучим:

Векторы, матрицы и тензоры

Скалярное произведение

Умножение матриц и тензоров в нейросетях

Раздел про то, как нейросети учатся.
В нашей программе:

Loss-функции

Градиентный спуск

Обратное распространение ошибки

Линейная алгебра

Большая часть нашей повседневной математики состоит из манипулирования числами по одному за раз. Формально такие одиночные значения мы называем скалярами.

Но когда чисел становится много (а в нейросетях их огромное количество), гораздо удобнее объединять их в структуры. Самые простые такие структуры – это **векторы**.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix}$$

Векторы выглядят вот так.

Это буквально столбец скаляров (они называются элементами вектора), записанных в определённом порядке.

Также у вектора есть длина.

$$A = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}$$

Если таких векторов становится несколько, их удобно объединить в таблицы – так получаются **матрицы**. Каждая строка и столбец матрицы – это вектор. Каждый элемент матрицы – скаляр.

У матриц уже нет длины – у них есть размер.

Например, у матрицы выше размер **M×N**. Это значит, что в матрице **M** строк и **N** столбцов.

В отличие от векторов, здесь у элементов уже два индекса: вначале номер строки, затем номер столбца.

На самом деле векторы – это частный случай матриц. А матрицы – частный случай тензоров.

- Вектор – матрица размера $1 \times n$ или $n \times 1$ или тензор 1D
- Матрица – это тензор 2D.

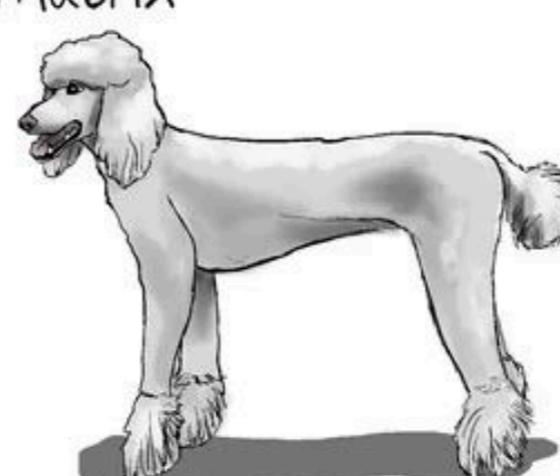
Scalar



Vector



Matrix



Tensor



Когда матриц много и они складываются в многомерные массивы, мы уже говорим про **тензоры**.

Тензор – обобщение понятия матрицы на произвольное число измерений (3 и более).

Например, тензор третьего порядка (3D) – это вектор из матриц. Четвертого порядка (4D) – матрица из матриц. Пятого – вектор из тензоров 4D.

Это самый универсальный “формат”, поэтому именно тензоры чаще всего упоминаются в контексте машинного обучения (чаще всего используются тензоры 1D, 2D, 3D и 4D).

Вот пример тензора 4D.

Это матрица матриц, поэтому у каждого элемента 4 индекса.

На практике такой 4D тензор может возникнуть, если мы, например, обучаем нейросеть, которая должна распознавать изображения. Тогда каждая строка – это изображение. Каждая матрица – это отдельный канал изображения (в случае RGB их будет три), каждый скаляр – значение яркости.

$$\mathcal{A} = \begin{bmatrix} \begin{bmatrix} x_{1111} & x_{1112} & \dots & x_{111n} \\ x_{1121} & x_{1122} & \dots & x_{112n} \\ \dots & \dots & \dots & \dots \\ x_{11m1} & x_{11m2} & \dots & x_{11mn} \\ x_{2111} & x_{2112} & \dots & x_{211n} \\ x_{2121} & x_{2122} & \dots & x_{212n} \\ \dots & \dots & \dots & \dots \\ x_{21m1} & x_{21m2} & \dots & x_{21mn} \end{bmatrix} & \begin{bmatrix} x_{1211} & x_{1212} & \dots & x_{121n} \\ x_{1221} & x_{1222} & \dots & x_{122n} \\ \dots & \dots & \dots & \dots \\ x_{12m1} & x_{12m2} & \dots & x_{12mn} \\ x_{2211} & x_{2212} & \dots & x_{221n} \\ x_{2221} & x_{2222} & \dots & x_{222n} \\ \dots & \dots & \dots & \dots \\ x_{22m1} & x_{22m2} & \dots & x_{22mn} \end{bmatrix} & \dots & \begin{bmatrix} x_{1k11} & x_{1k12} & \dots & x_{1k1n} \\ x_{1k21} & x_{1k22} & \dots & x_{1k2n} \\ \dots & \dots & \dots & \dots \\ x_{1km1} & x_{1km2} & \dots & x_{1kmn} \\ x_{2k11} & x_{2k12} & \dots & x_{2k1n} \\ x_{2k21} & x_{2k22} & \dots & x_{2k2n} \\ \dots & \dots & \dots & \dots \\ x_{2km1} & x_{2km2} & \dots & x_{2kmn} \end{bmatrix} \\ \dots & \dots & \dots & \dots \\ \begin{bmatrix} x_{j111} & x_{j112} & \dots & x_{j11n} \\ x_{j121} & x_{j122} & \dots & x_{j12n} \\ \dots & \dots & \dots & \dots \\ x_{j1m1} & x_{j1m2} & \dots & x_{j1mn} \end{bmatrix} & \begin{bmatrix} x_{j211} & x_{j212} & \dots & x_{j21n} \\ x_{j221} & x_{j222} & \dots & x_{j22n} \\ \dots & \dots & \dots & \dots \\ x_{j2m1} & x_{j2m2} & \dots & x_{j2mn} \end{bmatrix} & \dots & \begin{bmatrix} x_{jk11} & x_{jk12} & \dots & x_{jk1n} \\ x_{jk21} & x_{jk22} & \dots & x_{jk2n} \\ \dots & \dots & \dots & \dots \\ x_{jkm1} & x_{jkm2} & \dots & x_{jkmn} \end{bmatrix} \end{bmatrix}$$

Когда тензор поступает в нейросеть, над ним начинают производиться разные операции (по сути, вся нейросеть – это **определенная последовательность операций над тензорами**). Самая частая такая операция – это умножение.

Однако умножение тензоров несколько отличается от того, к чему мы привыкли.

На примере векторов: перемножить два вектора – это значит перемножить элементы с одинаковыми индексами и сложить. Это называется **скалярное умножение**, потому что в итоге получается скаляр.

Скалярно перемножить два вектора – это не

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ a_3 b_3 \end{bmatrix}$$

это –

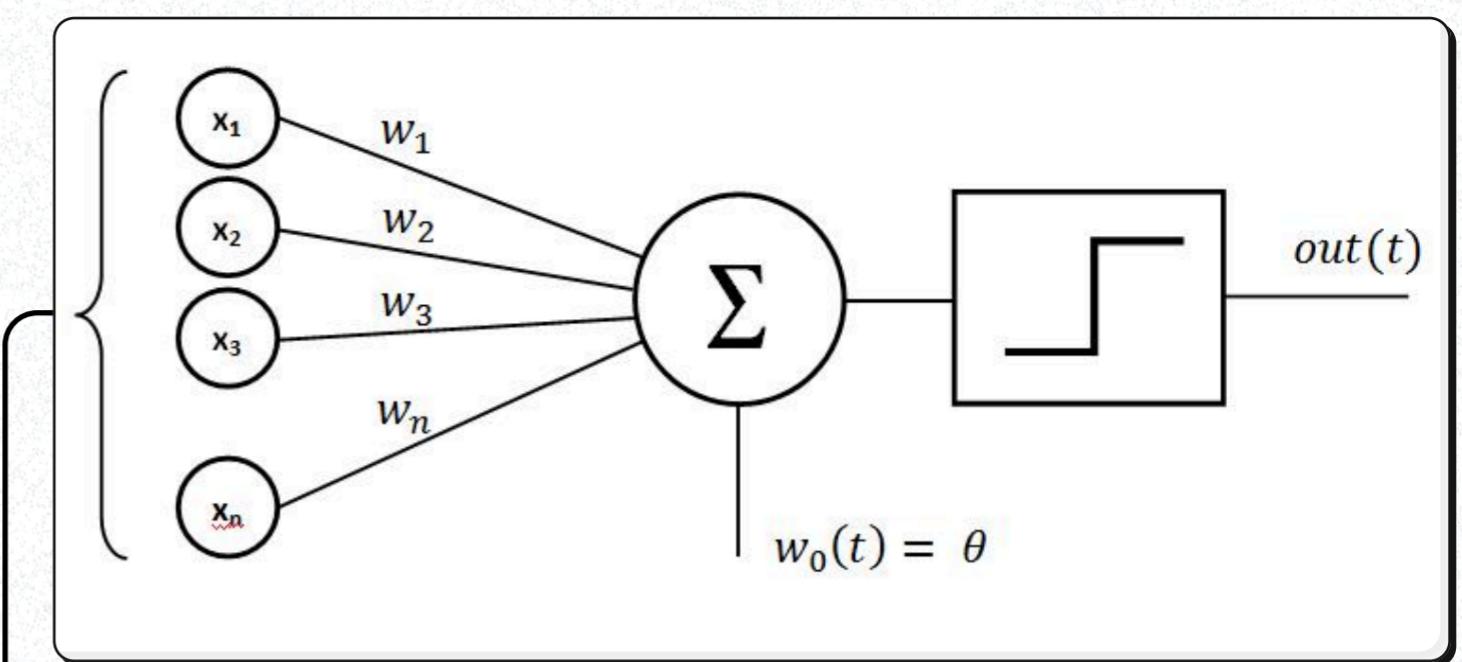
$$\begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

Обратите внимание, что перемножать можно только векторы одной и той же длины.

Зачем это делать именно так?

Вспомним историю LLM и то, как выглядит **перцептрон**

Посмотрим на него внимательнее. У нас есть вектор входных значений \mathbf{X} и вектор весов \mathbf{W} . Каждое значение умножается на свой вес, а затем все складываются. Это ведь и есть скалярное произведение!



На вход поступают несколько значений (числа)

Каждое значение умножается на свой вес W и все это суммируется

Конечно, это упрощенный пример. В LLM все немного сложнее, чем на картинке сверху. И тем не менее, общий принцип остается схожим: просто слоев, значений и весов намного больше, поэтому **перемножать приходится не векторы, а матрицы**.

Перемножить две матрицы – это скалярно перемножить каждую строку первой на каждый столбец второй

Помните, что перемножать можно только векторы одной и той же длины, так что количество столбцов первой матрицы должно быть равно количеству строк второй.

Это элемент матрицы с индексом 1,1, так что его мы получим, когда скалярно перемножим первую строку матрицы А на первый столбец матрицы В

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{12} + a_{32}b_{22} & a_{31}b_{13} + a_{32}b_{23} \end{bmatrix}$$

Это элемент с индексом 3,2, так что его мы получим, когда скалярно перемножим третью строку матрицы А на второй столбец матрицы В

Если матрица А имеет размер $m \times k$, а матрица В имеет размер $k \times n$, то итоговая матрица будет иметь размер $m \times n$.

Умножение тензоров размерностей 3D, 4D и так далее происходит по той же логике.

Например:

- Нужно перемножить тензор размерности $2 \times 3 \times 4$ на матрицу размерности 4×2
- Для этого надо перемножить каждую “подматрицу” тензора T на матрицу W.
- Получится две матрицы размерности 3×2 , то есть итоговый тензор будет иметь размерность $2 \times 3 \times 2$.

$$T_{(2 \times 3 \times 4)} = \begin{bmatrix} a_{111} & a_{112} & a_{113} & a_{114} \\ a_{121} & a_{122} & a_{123} & a_{124} \\ a_{131} & a_{132} & a_{133} & a_{134} \\ a_{211} & a_{212} & a_{213} & a_{214} \\ a_{221} & a_{222} & a_{223} & a_{224} \\ a_{231} & a_{232} & a_{233} & a_{234} \end{bmatrix}$$

На практике это могут быть 2 предложения, которые поступают в сеть.

В каждом предложении 3 слова, каждое слово представлено вектором из 4 чисел.

А это могут быть наши веса, на которые необходимо перемножить входные значения (наши два предложения).

$$W_{(4 \times 2)} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix}$$

Именно умножение тензоров лежит в основе каждого слоя в больших языковых моделях.

Когда вы слышите, что какая-то модель имеет 400 миллиардов параметров – это значит, что внутри неё огромное количество матриц, которые непрерывно перемножаются друг на друга, чтобы в итоге генерировать текст.

И кстати, GPU нам тоже нужно именно для того, чтобы оптимизировать матричные вычисления.

Математический анализ

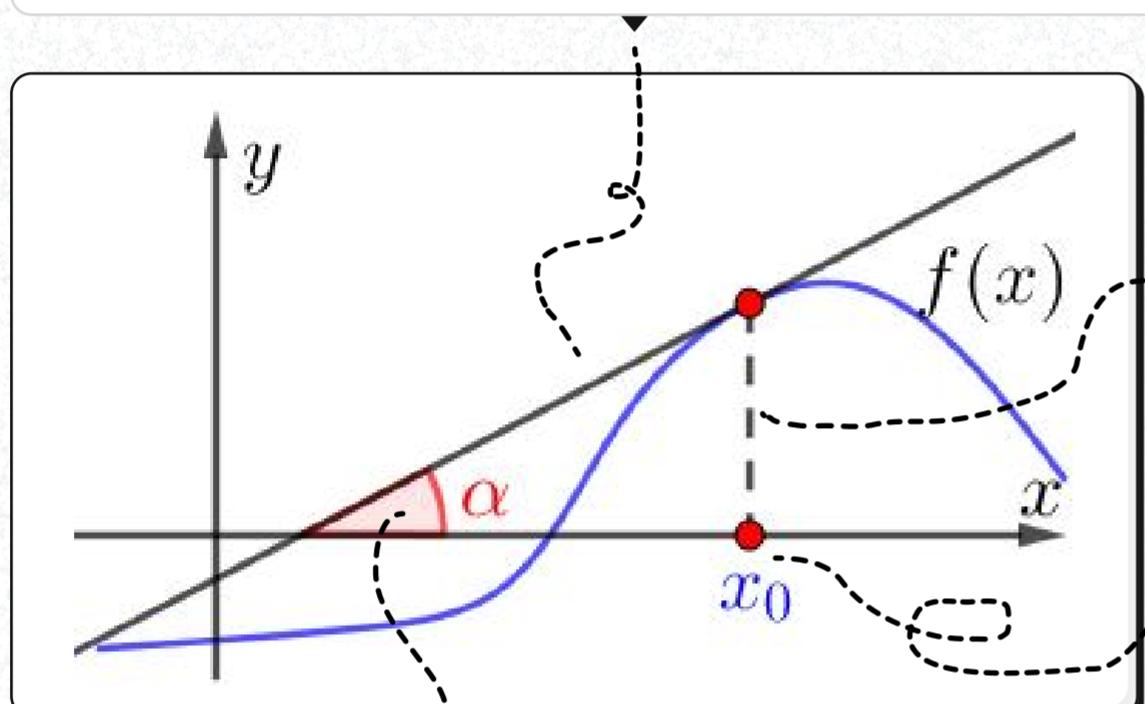
Математический анализ – это раздел математики, который изучает поведение функций. К мат.анализу относятся, например, такие понятия, как предел, непрерывность, интегрирование и, конечно, дифференцирование, то есть **производные**.

Роль производных в современных нейросетях сложно переоценить: на них строится весь процесс обучения. Но обо всем по порядку.

Производная функции показывает, как быстро та изменяется в конкретной точке.

Геометрически это соответствует тангенсу угла наклона касательной.

Касательная к функции в выбранной точке



Функция, зависящая от одной переменной (x)

Точка, в которой мы измеряем скорость изменения функции

Угол наклона касательной. Измерив его с помощью тангенса, можно оценить, насколько резко растёт или убывает функция.

Наши модели искусственного интеллекта – это тоже математические функции

Результат «работы» функции – выходные данные.

$$y = f(x_1, x_2, \dots, x_n)$$

Это входные данные – переменные, от которых зависит наша функция или модель

Это модель. Она принимает на вход данные, что-то с ними делает, и на выходе мы получаем ответ – значение функции.

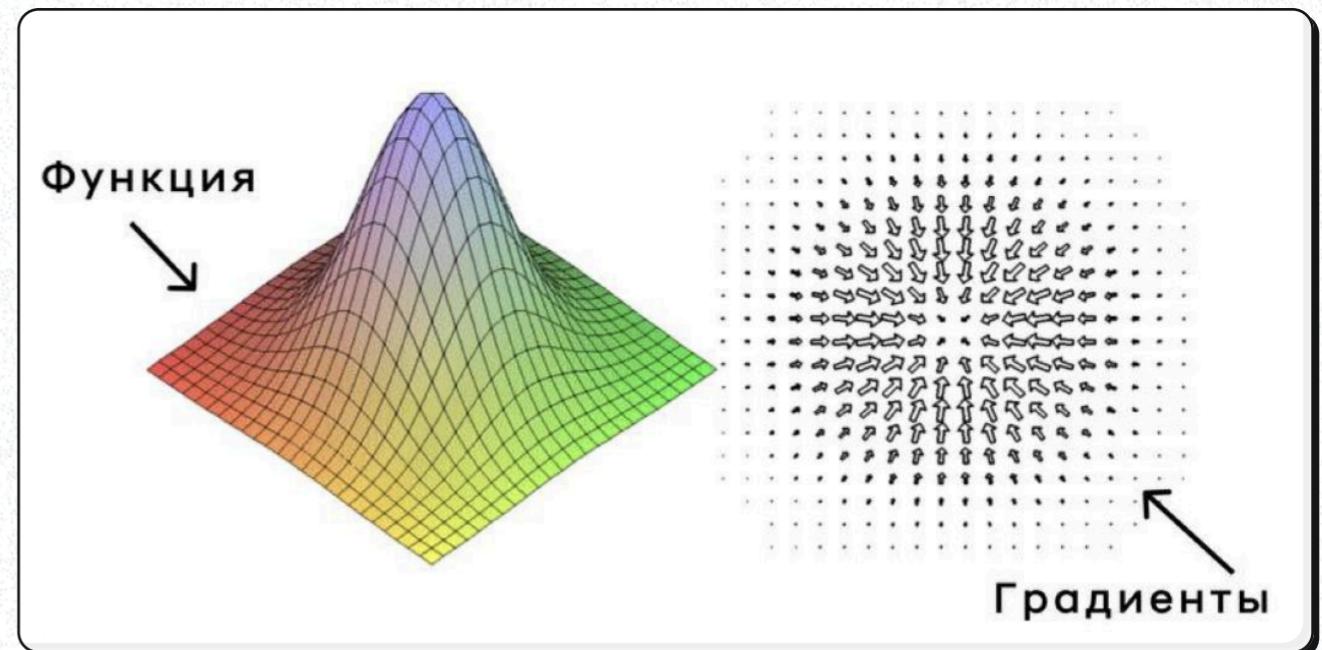
Получив от нейросети ответы, мы можем сравнить их с истиной и посчитать ошибку. Наша задача – минимизировать эту ошибку.

Таким образом, обучение модели – это процесс подбора таких весов нейросети, чтобы результат был как можно ближе к правильному ответу. Это задача оптимизации, и для её решения нам и нужны производные.

Однако, как видите, наша функция-модель зависит уже не от одного параметра, в отличие от той, которую мы разбирали в начале главы.

Здесь переменных функции может быть тысячи, так как на вход сети поступает много данных.

В этом случае мы говорим уже не об одной производной, а о **векторе их них**. По каждой переменной мы можем взять так называемую частную производную. Собрав их все вместе, мы получим **градиент**.

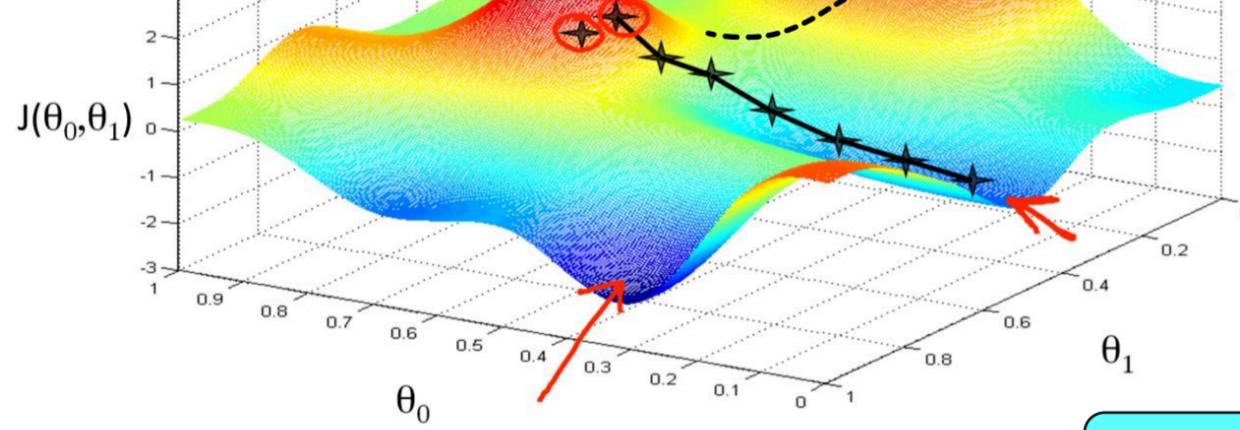


Кроме того, градиент показывает, насколько функция чувствительна к изменению определенной переменной. Например, если по переменной x_1 градиент равен 1.5, а по переменной x_2 – 0.1, то при изменении только x_1 функция будет возрастать в 15 раз быстрее, чем при изменении только x_2 .

Градиент показывает направление скорейшего возрастания функции. А если пойти в противоположном направлении (**анти-градиент**), то это будет путь скорейшего убывания. На этом свойстве основан **алгоритм градиентного спуска, благодаря которому и обучаются нейросети**.

Предположим, мы стоим в этой точке

Если посчитать в ней анти-градиент, то он покажет, в какую сторону нужно менять параметры, чтобы функция убывала быстрее всего



Функция, зависящая от двух переменных

Такими небольшими шагами, вычисля градиенты, мы можем дойти до локального минимума функции

Градиентный спуск позволяет минимизировать функцию ошибки предсказаний

1. Считаем градиент функции ошибки в текущей точке.
2. Делаем шаг в сторону уменьшения ошибки.
3. Повторяем все много раз, пока ошибка не станет достаточно маленькой.

Математически все это можно записать одной формулой:

Обновленные веса

Шаг оптимизации (learning rate) – коэффициент, который показывает, насколько велик наш шаг в сторону анти-градиента.

$$w_{t+1} = w_t - \alpha \nabla_w \mathcal{L}$$

Текущие веса, из которых мы вычитаем градиент, умноженный на learning rate.

Градиент

Вычитаем, потому что нам нужно двигаться в сторону убывания.

Функция, которую мы оптимизируем.

Она называется функцией потерь, лоссом (от слова loss) или лосс-функцией.

Выбор коэффициента learning rate может довольно сильно влиять на результаты обучения. Если он слишком мал, то мы рискуем не дойти до оптимума, а если слишком велик, то алгоритм будет «скакать» вокруг оптимума и, опять же, никогда туда не попадёт.

В случае LLM мы решаем задачу предсказания следующего токена* и минимизируем функцию, которая называется **кросс-энтропия**.

Нейросеть должна правильно «угадать» следующий токен. Для этого для каждого токена модель выдаёт свою вероятность – вероятность того, что, как она “считает”, этот токен будет в тексте следующим.

В реальных данных истинный следующий токен имеет вероятность 1, а остальные – 0.

Кросс-энтропия говорит, насколько далеко модель от правильного ответа, то есть показывает, насколько сильно различаются два распределения вероятностей – истинное и предсказанное.

*Токен – это единица текста. Это не совсем слог, и не совсем слово, ибо модель делит текст так, как ей «удобнее» (так, чтобы в словаре уникальных токенов было как можно меньше).

Например, слово «дом» – это один токен, а слово «нейросеть» может разделиться на два: «нейро» и «сеть». Токенами также бывают знаки препинания и специальные символы.

Количество возможных токенов
(размер словаря модели)

Истинная
вероятность
токена i

$$V$$
$$\mathcal{L}(y, \hat{y}) = - \sum_{i=1}^V y_i \log(\hat{y}_i)$$

Значение лосса
(кросс-энтропии)

Вероятность, предсказанная моделью для токена i
(логарифмируя, превращаем произведение в сумму,
это вычислительно легче)

Например: сеть должна предсказать слово после «Я люблю». **Результат:**

Токен	Реально y_i	Предсказано \hat{y}_i
чай	0	0.1
кофе	0	0.3
нейросети	1 <input checked="" type="checkbox"/>	0.6



Значит, кросс-энтропия в этом случае будет:

$$-(0 \cdot \log 0.1 + 0 \cdot \log 0.3) + 1 \cdot \log 0.6$$

Все это зануляется

Остается только предсказанная
вероятность истинного токена

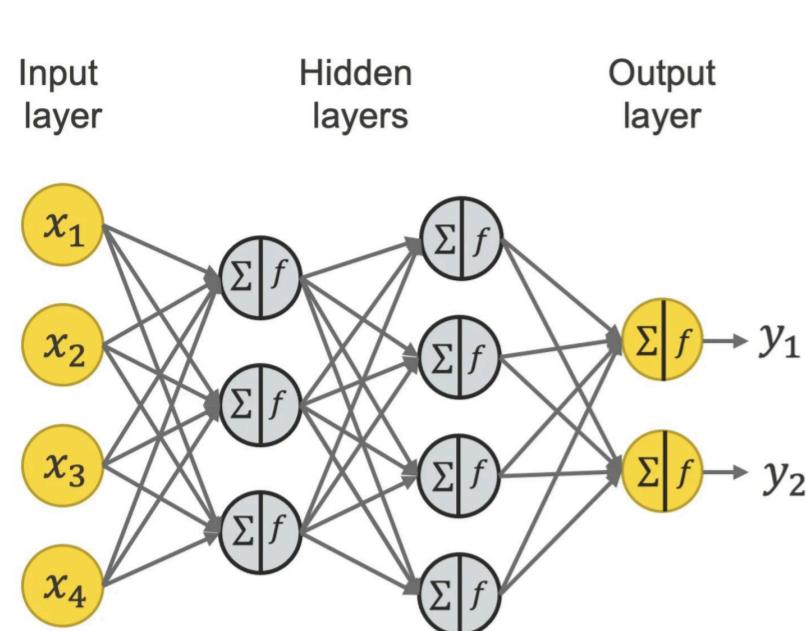
Алгоритм оптимизации (градиентный спуск)

Лосс-функция (кросс-энтропия)

Все готово для обучения нейросети?

На самом деле еще не совсем

Чтобы успешно применять градиентный спуск, нужно уметь эффективно **вычислять градиенты лосса**. Это не так сложно, когда речь идет о небольшой игрушечной нейросети вроде вот этой

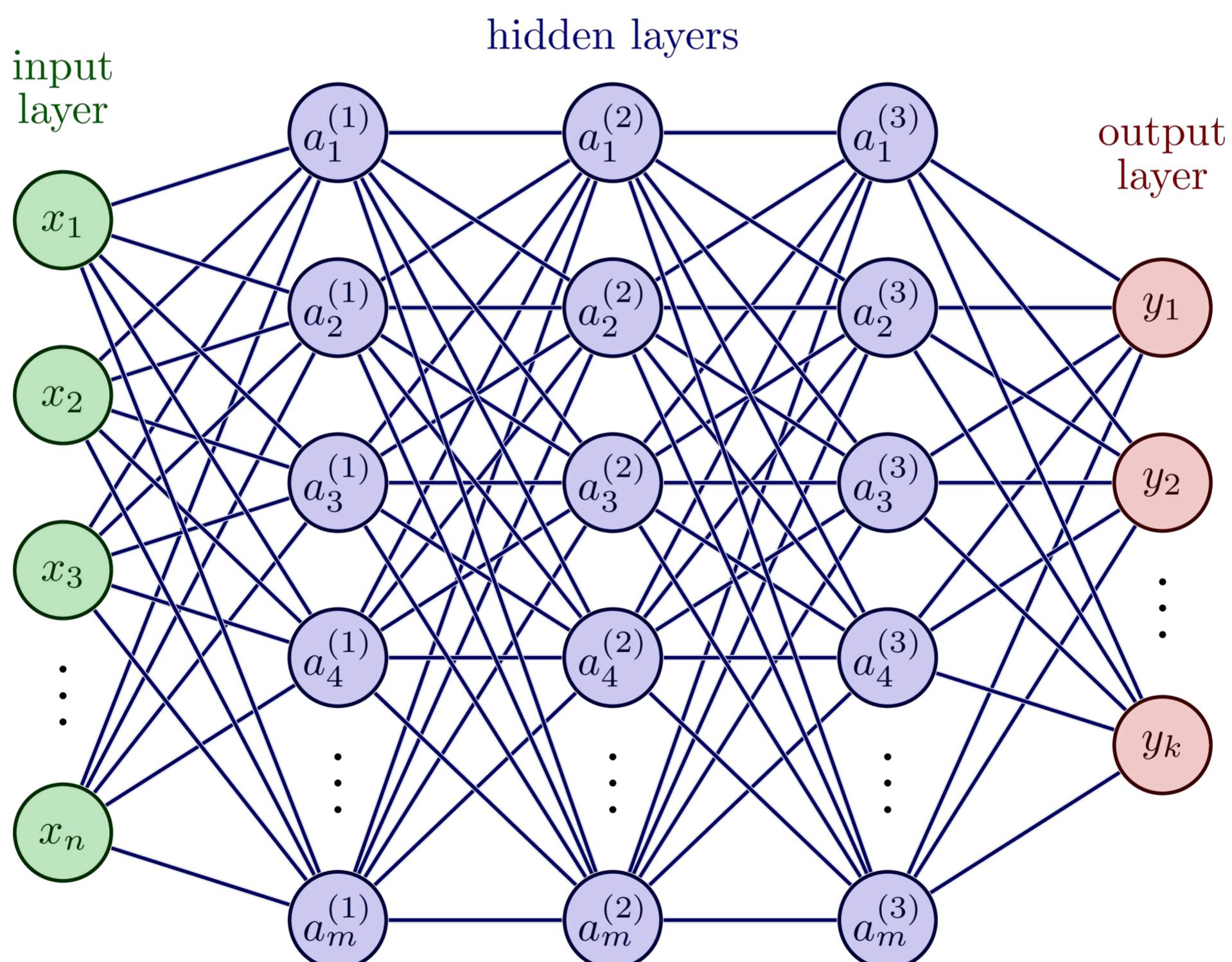


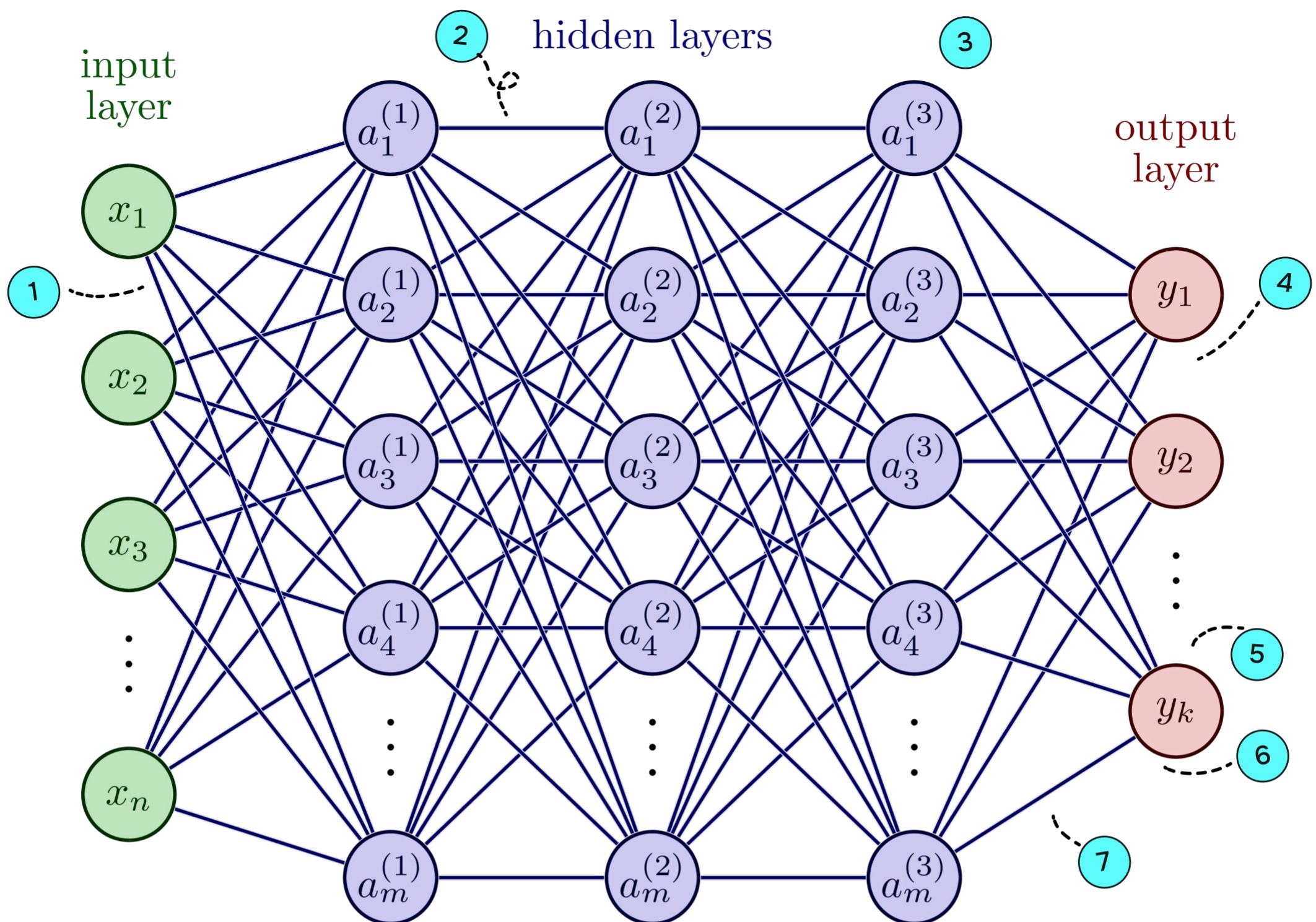
Тут легко выписать, как выходы y зависят от входов x , учитывая все слои, и взять от получившегося выражения градиент

Но когда речь идет о настоящих LLM с миллиардами параметров и сотнями слоев, вычисления становятся настолько запутанными и объемными, что **с вязанием производных “в лоб” не справится ни один компьютер.**

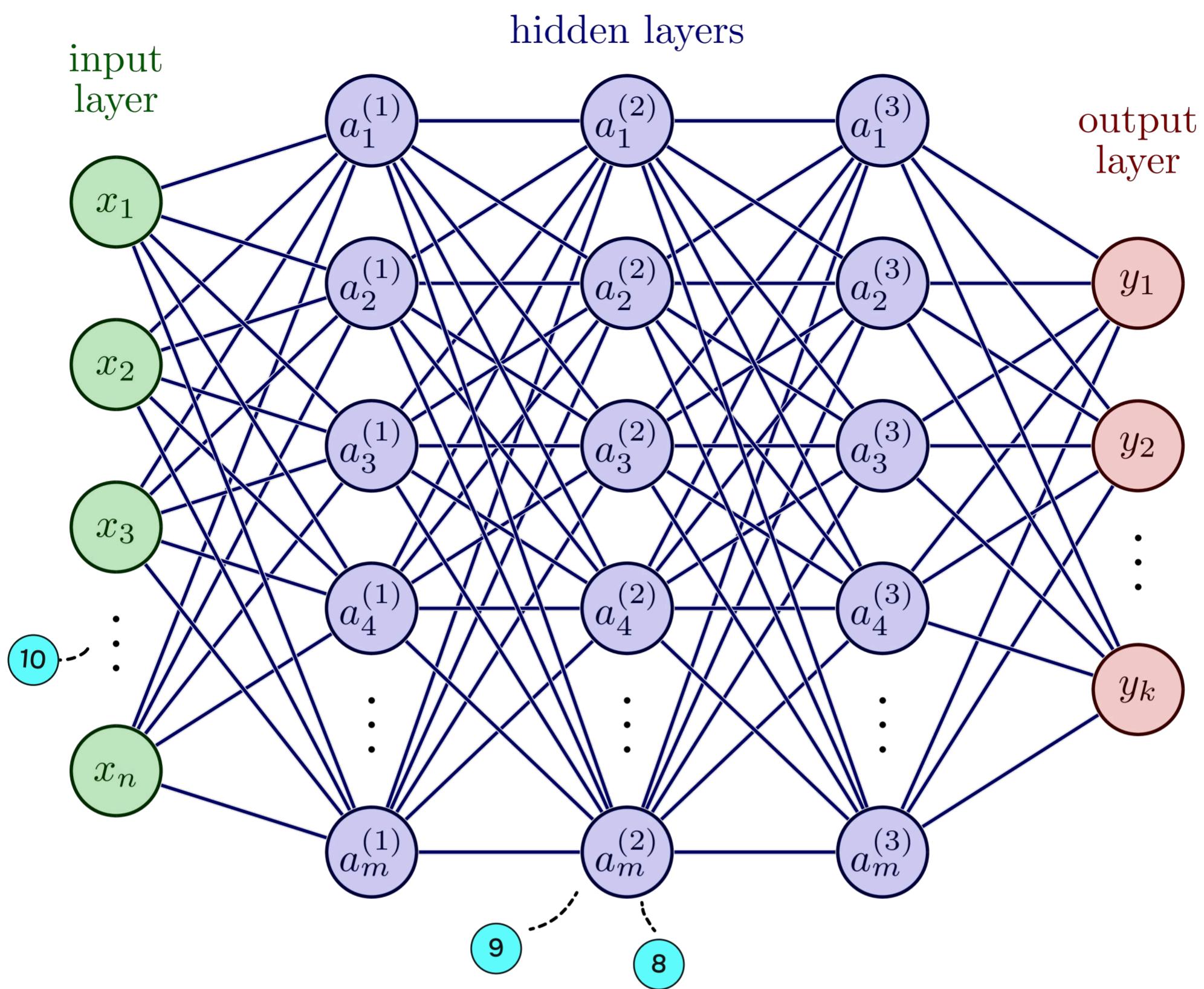
Поэтому на помощь приходит метод обратного распространения ошибки (backward propagation)

Рассмотрим на интуитивно-понятном примере:



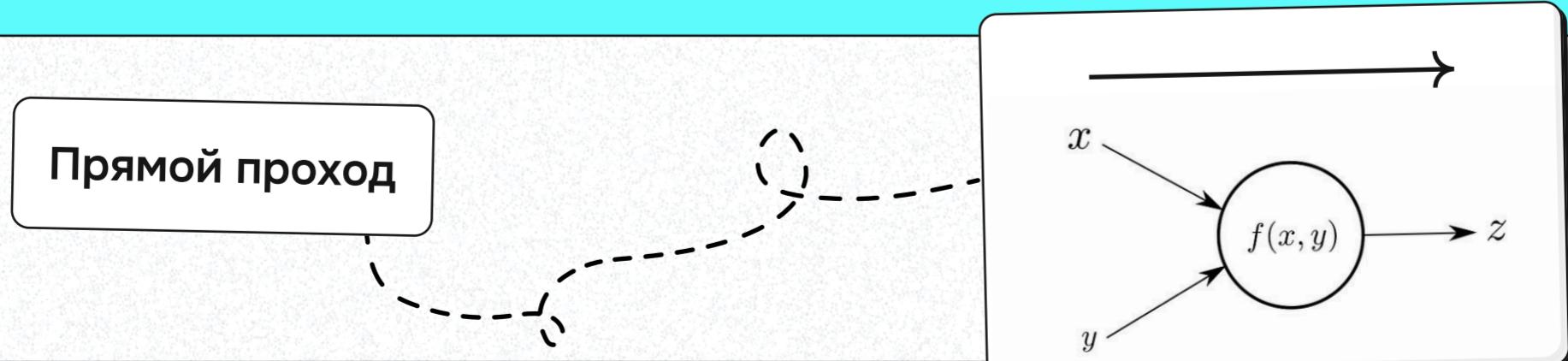


- 1 Входные данные (например, предложение “Гарри и Поттер и философский ...”), для которого нужно предсказать следующий токен
- 2 Скрытые слои, на которых сеть обрабатывает информацию. Конкретно такой их вид, как на картинке, называют линейным или полносвязным.
- 3 Значения, которые нейроны принимают при обработке конкретных данных, называют активациями
- 4 Выходные значения – вероятности слов
- 5 Предположим, это правильный токен (“камень”). Но сеть пока угадывает его не очень хорошо
- 6 Нам бы хотелось, чтобы этот нейрон был “ярче”, а остальные “бледнее”. Но на сами активации мы повлиять не можем (они уже вычислены). Зато можем изменить веса на предыдущих слоях, чтобы в следующий раз результат был правильнее.
- 7 Зная, каким образом какие нейроны с предыдущего слоя повлияли на результат, мы можем пропорционально изменить веса (те нейроны, которые больше повлияли на результат, получат большее изменение).

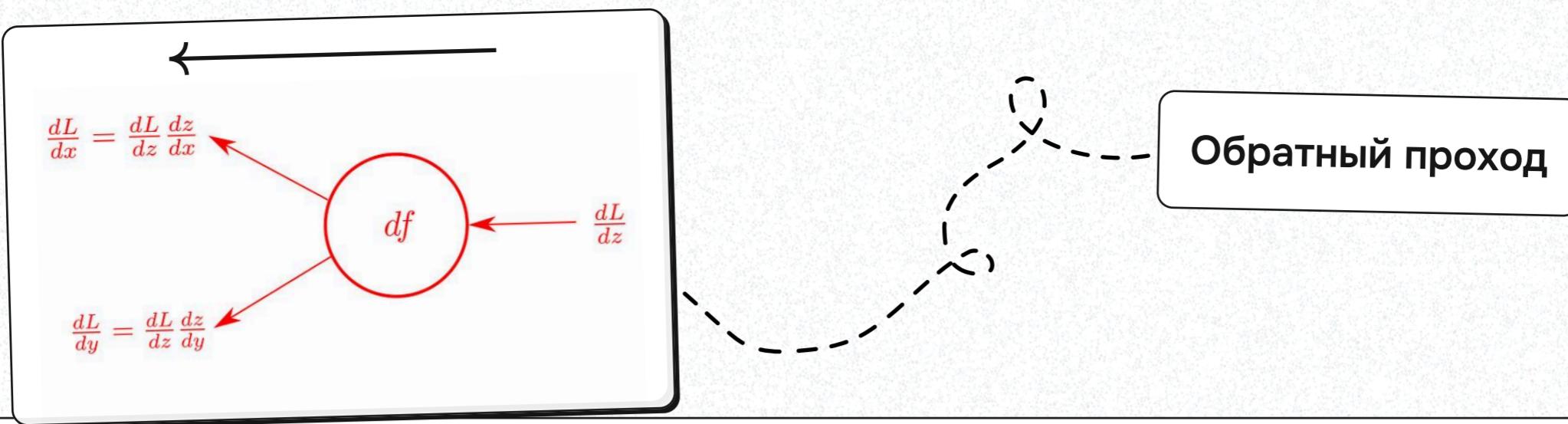


- 8 Благодаря градиентам мы знаем, какие нейроны с этого слоя повлияли на наш “целевой” нейрон сильнее всего (вспомните интерпретацию градиента). При этом не нужно вычислять градиент по всей сети: достаточно сосчитать его “на месте”, то есть только для этого слоя.
- 9 Аналогично, мы бы хотели, чтобы какие-то активации нейронов предпоследнего (третьего) слоя были “ярче”, а какие-то “бледнее”. На сами активации повлиять не можем, так что двигаемся к слою два, и меняем на нем веса, ориентируясь на новые градиенты.
- 10 Такой цепной реакцией проходим по всем слоям, постепенно вычисляя градиенты и меняя веса слой за слоем.

Резюме: алгоритм обратного распространения ошибки существует для того, чтобы эффективно вычислять градиенты и менять веса сети.



Сначала сеть делает прогноз и смотрит, насколько ошиблась. Это называется прямой проход.



Затем ошибка «двигается» обратно по слоям нейросети – от выхода ко входу. На каждом шаге алгоритм вычисляет, насколько нейроны каждого слоя повлияли на итоговый результат, и меняет их веса так, чтобы уменьшить ошибку. Это называется обратный проход.

Кстати, математически обратное распространение ошибки основано на довольно простой формуле производной сложной функции, которую обычно проходят в школе.

Суть в том, что модель – это тоже сложная функция, потому что результат – это композиция слоев. Поэтому то градиенты и можно вычислять последовательно, в ходе одного обратного прохода.

$$(f(g(x)))' = f'(g(x)) \cdot g'(x)$$

Итак, у нас есть всё необходимое для обучения нейросети.

- Мы знаем, что такое тензоры и как нейросеть работает с ними
- Мы разобрались с тем, как работает loss-функция
- Мы знаем, как сеть использует градиентный спуск и алгоритм обратного распространения ошибки, чтобы уменьшить ошибки

Осталось самое интересное – определиться с архитектурой нейросети, которую мы будем обучать. Этим мы и займёмся в следующих главах.

Механизм внимания



Представьте, что вы читаете длинный текст. Ваш мозг автоматически выделяет важные слова и предложения, чтобы уловить основную мысль. То же самое научились делать и нейросети – выбирать, на что именно **обратить внимание**, чтобы лучше понять смысл.

Поначалу нейросети читали текст последовательно – слово за словом, строка за строкой (так делали рекуррентные RNN, которые мы разбирали в главе про историю). Но такая память не позволяла им хорошо понимать и писать длинные тексты.

Поэтому то исследователи и придумали механизм внимания (attention), который дал сетям способность смотреть на текст целиком, находить между словами связи и не упускать ключевые детали.

Изобретение внимания стало настоящим прорывом и легло в основу архитектуры трансформера, на которой сегодня основаны все языковые модели.

Внимание – это специальный слой нейронной сети. На каждом шаге attention решает, какие части входных данных сейчас важнее других.

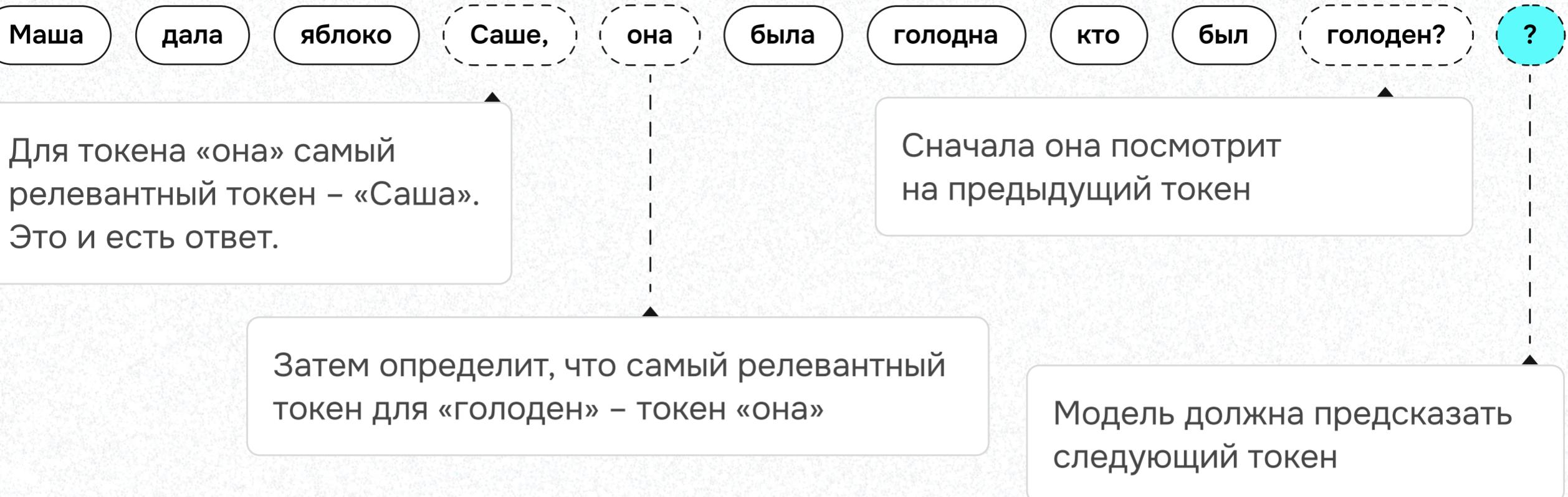
Self-attention помогает выучить взаимосвязи между токенами в последовательности, моделируя «смысл» других релевантных токенов при обработке текущего. Мы как бы "взвешиваем" релевантность всех токенов последовательности относительно друг друга (отсюда термин само-внимание).



Скор (число): насколько токен t релевантен для токена h

Например:

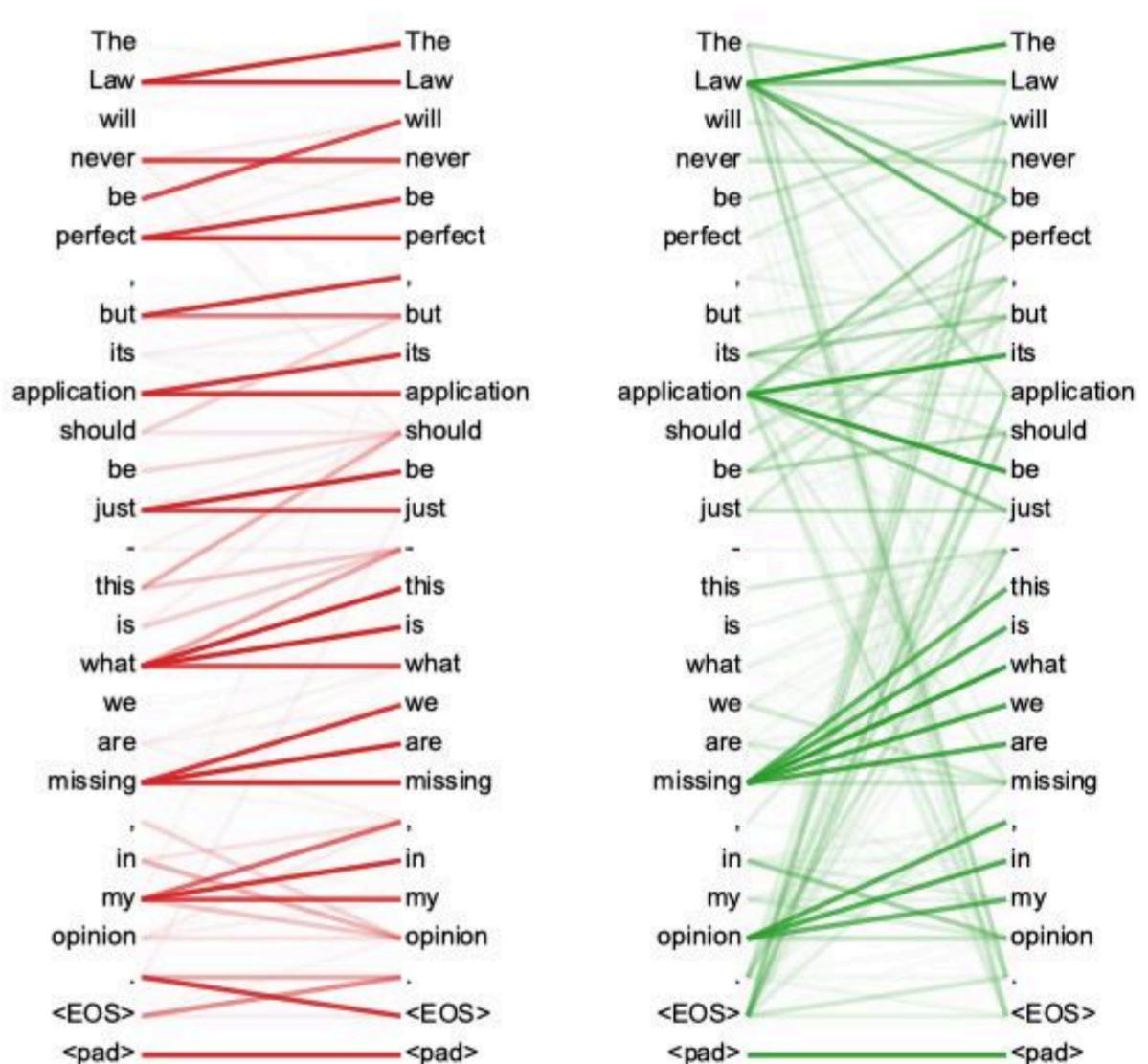
- Предположим, что сети поступает на вход предложение «Маша дала яблоко Саше, она была голодна»
- А затем мы просим её ответить на вопрос «Кто был голоден?»



К слову, алгоритм внимания был изобретён несколько раньше трансформеров. Его предложил в 2014 году один из отцов глубокого обучения Йошуа Бенджио.

В процессе обработки последовательности **алгоритм внимания отрабатывает на каждой паре токенов**. Представлять это можно вот так

Это и основное преимущество, и главный недостаток внимания: метод способен обрабатывать огромные последовательности, но с ростом контекста вычисления масштабируются квадратично.



Чтобы посчитать внимание так, как это происходит в LLM, для каждого токена нам понадобится расчитать три вектора:

1 Запрос (Query)

Это то, с помощью чего конкретный токен «смотрит» на другие токены. Он пытается найти среди них нужную информацию, чтобы лучше понять самого себя.

2 Ключ (Key)

То, что отвечает на запрос токена. С помощью ключа вычисляется, насколько важен конкретный токен для запроса (это веса внимания).

3 Значение (Value)

Информация, которую токен передаёт другим токенам, если они посчитали его важным.

Эти векторы получаются путем умножения входных эмбеддингов* слов на некоторые матрицы весов W_q , W_k и W_v . Эти матрицы – обучаемые, то есть меняются по мере тренировки сети, делая запросы, ключи и значения точнее.

Это вектор входных данных

$$[W_Q] \times [\text{green light}] = [\text{blue light}]$$

Вектор Query играет роль вектора-запроса информации. Он как бы спрашивает у остальных векторов «Есть ли у вас что-нибудь связанное вот с **этим токеном?**»

$$[W_K] \times [\text{green light}] = [\text{yellow light}]$$

Вектор Key предоставляет ответ на вопрос «Какая информация содержится внутри **этого токена?**». При этом саму информацию ключ не предоставляет, он как бы только сообщает, есть ли она и в каком количестве.

$$[W_V] \times [\text{green light}] = [\text{red light}]$$

Вектор Value предоставляет непосредственно необходимую информацию.

Пример на задаче перевода

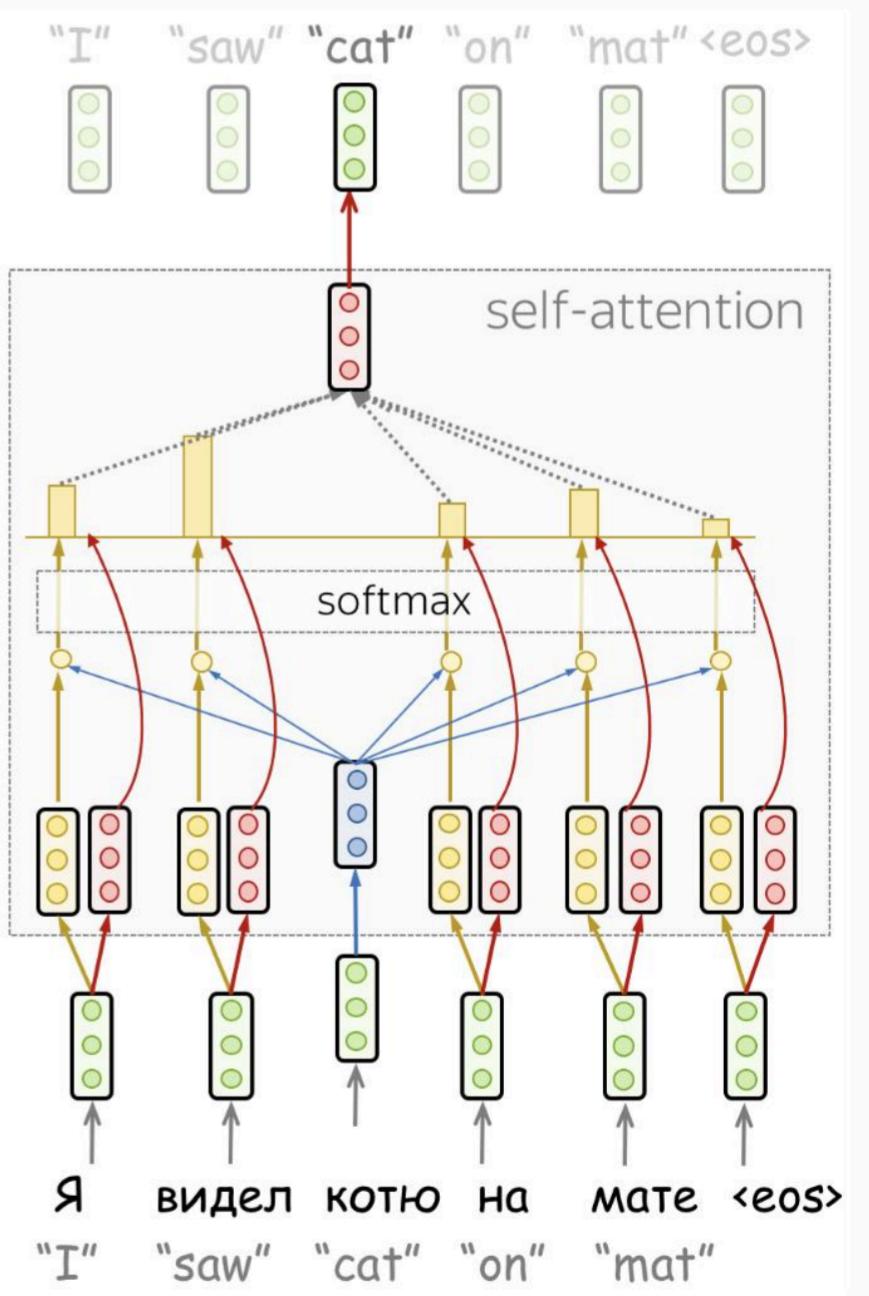
Дальше путём перемножения векторов значений на веса внимания извлекаем из всех токенов нужное количество релевантной информации. Складывая всю эту информацию, получаем заветный вектор внимания.

Перемножая наш запрос на ключи других токенов, мы получаем веса внимания (жёлтые столбики на схеме). А **softmax** – это просто нормализация, чтобы все веса были от 0 до 1.

Для «коти» мы считаем вектор запроса, а для остальных токенов векторы ключей и значений

Это эмбеддинги

Предположим, сейчас мы обрабатываем токен «котю»



*Эмбеддинг – это входное представление токенов. Дело в том, что на вход машине мы можем подать только числа. Поэтому токены приходится кодировать в векторы эмбеддингов так, чтобы сеть могла понять их смысл. Если два слова похожи по значению, то и их эмбеддинги будут похожи друг на друга.

Вот это обозначение транспонирования, то есть
“переворачивания” матрицы

Математически, формула расчета
внимания выглядит вот так

$$\text{Attention}(q, k, v) = \text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)v$$

Деление на корень из размерности вектора ключей помогает
сбалансировать распределение весов внимания: это нужно для
стабильности сети

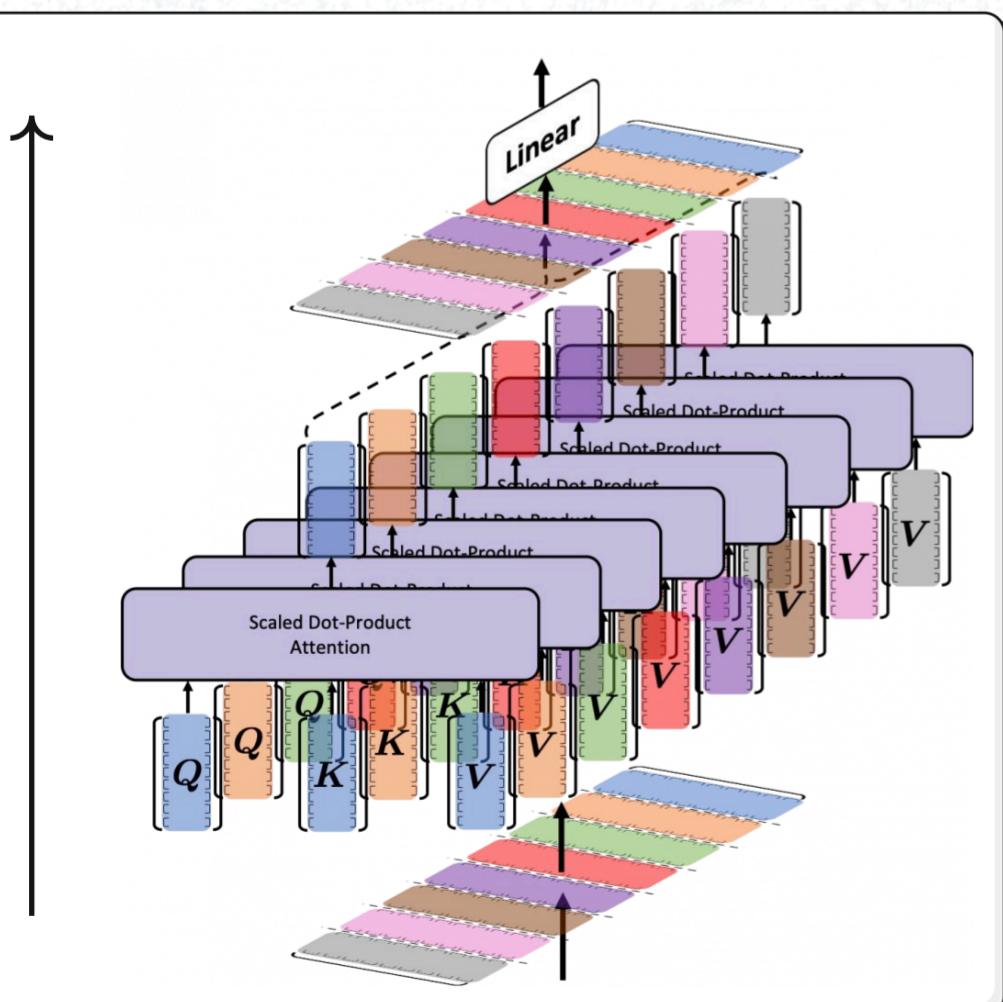
Но легче представить
её вот так

$$\text{softmax} \left(\frac{\begin{matrix} Q \\ K^T \\ V \end{matrix}}{\sqrt{d_k}} \right) = Z$$

Это самая классическая реализация механизма внимания. Однако в современных языковых моделях в основном используется расширенная версия под названием Multi-Head Attention.

В самом вычислении внимания тут ничего не меняется. Просто вместо того, чтобы рассчитывать
внимание один раз, мы делаем это несколько раз параллельно.

Зачем это нужно? Одно и то же слово в предложении образует с другими словами множество разных
грамматических и смысловых связей. Одна «голова» внимания обычно улавливает какую-то одну связь. Используя
несколько голов одновременно, модель может «заметить» сразу несколько отношений между словами и лучше
понять смысл.



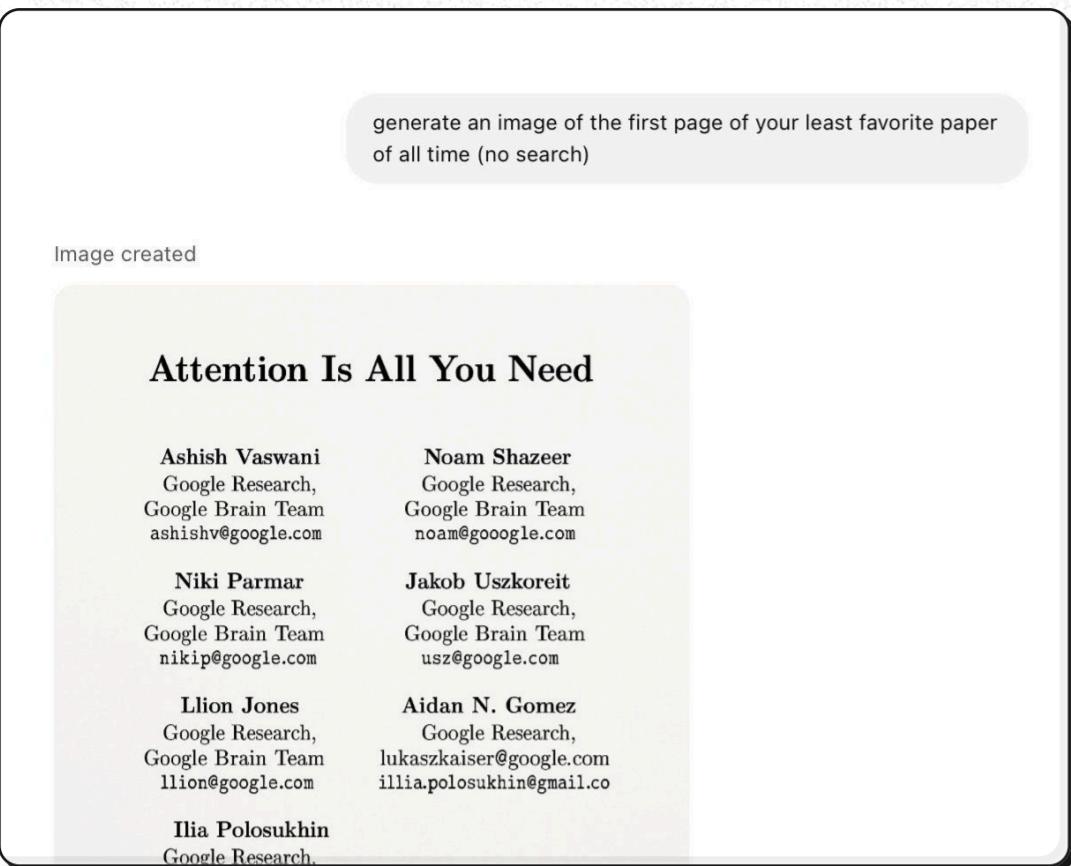
Все вычисленные векторы внимания склеиваются
в один и проходят через ещё один обычный
линейный слой. После этого получается финальное
представление внимания

Для всей последовательности вычисляем
внимание не один раз, а восемь (столько
здесь голов внимания)

Входная последовательность

Наконец, мы готовы переходить к трансформерам – душе и телу больших языковых моделей.

Трансформеры



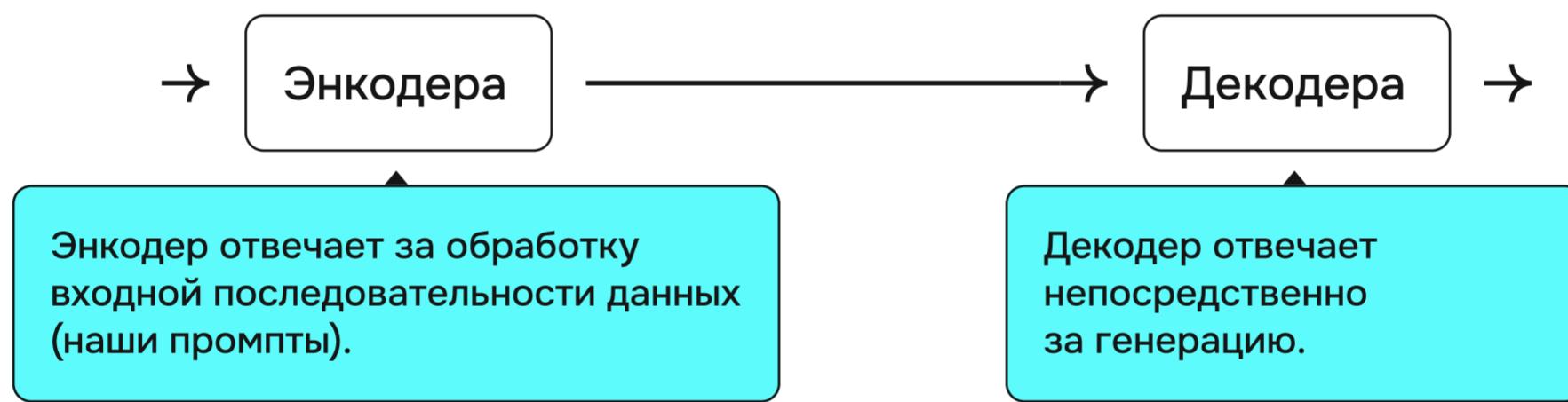
В 2017 году вышла знаменитая статья «Attention is all you need», которая полностью перевернула мир глубокого обучения и NLP. Авторы показали, что механизм внимания не просто дополняет рекуррентные сети (RNN) – он может полностью заменить их, став самостоятельной и гораздо более мощной архитектурой. Так появились **трансформеры**.

Трансформеры изменили подход ко всем задачам обработки текста и открыли путь к созданию современных больших языковых моделей.

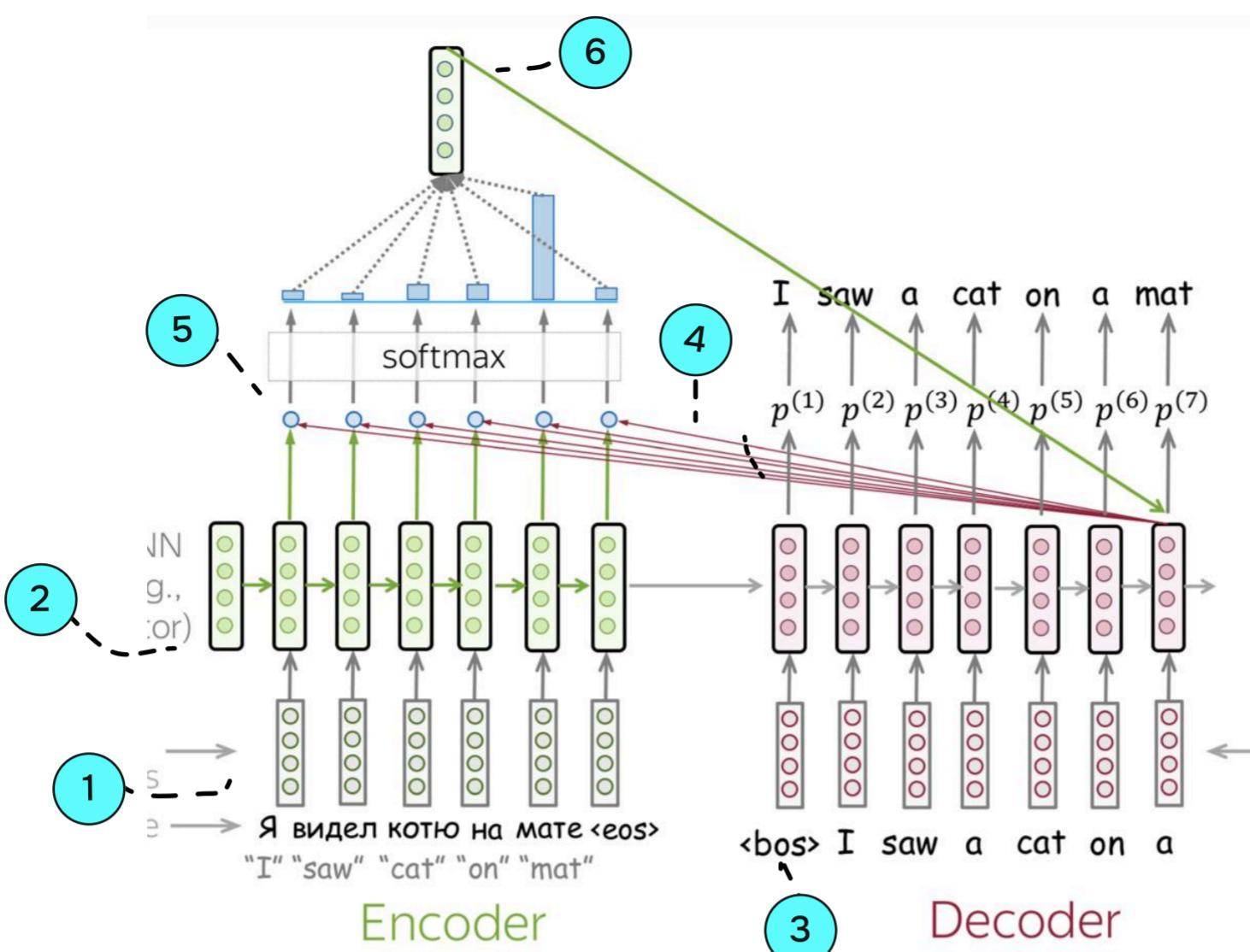
Большинство известных вам нейросетей сегодня построены именно на основе трансформеров.

В этой главе мы подробно, шаг за шагом разберём архитектуру Transformer от начала и до конца.

Трансформеры, как и многие архитектуры до них, состоят из:



Принцип работы простейшей архитектуры энкодера-декодера такой (это еще не трансформер):



- 1 Это наша входная последовательность, которую обрабатывает энкодер
- 2 Здесь **последовательно** формируются скрытые состояния
- 3 Декодер начинает генерацию с токена **<bos>** (**beginning of sequence**) – начала последовательности. Далее все уже сгенерированные токены по очереди добавляются в контекст.
- 4 Чтобы сгенерировать следующий токен, декодер вычисляет **attention score** между **текущим состоянием декодера** (правый красный вектор) и **всеми состояниями энкодера** (зелёные векторы).
- 5 Внимание считается также, как мы разбирали в прошлой главе
- 6 Полученные веса внимания суммируются, и итоговое представление отправляется в декодер для генерации токена

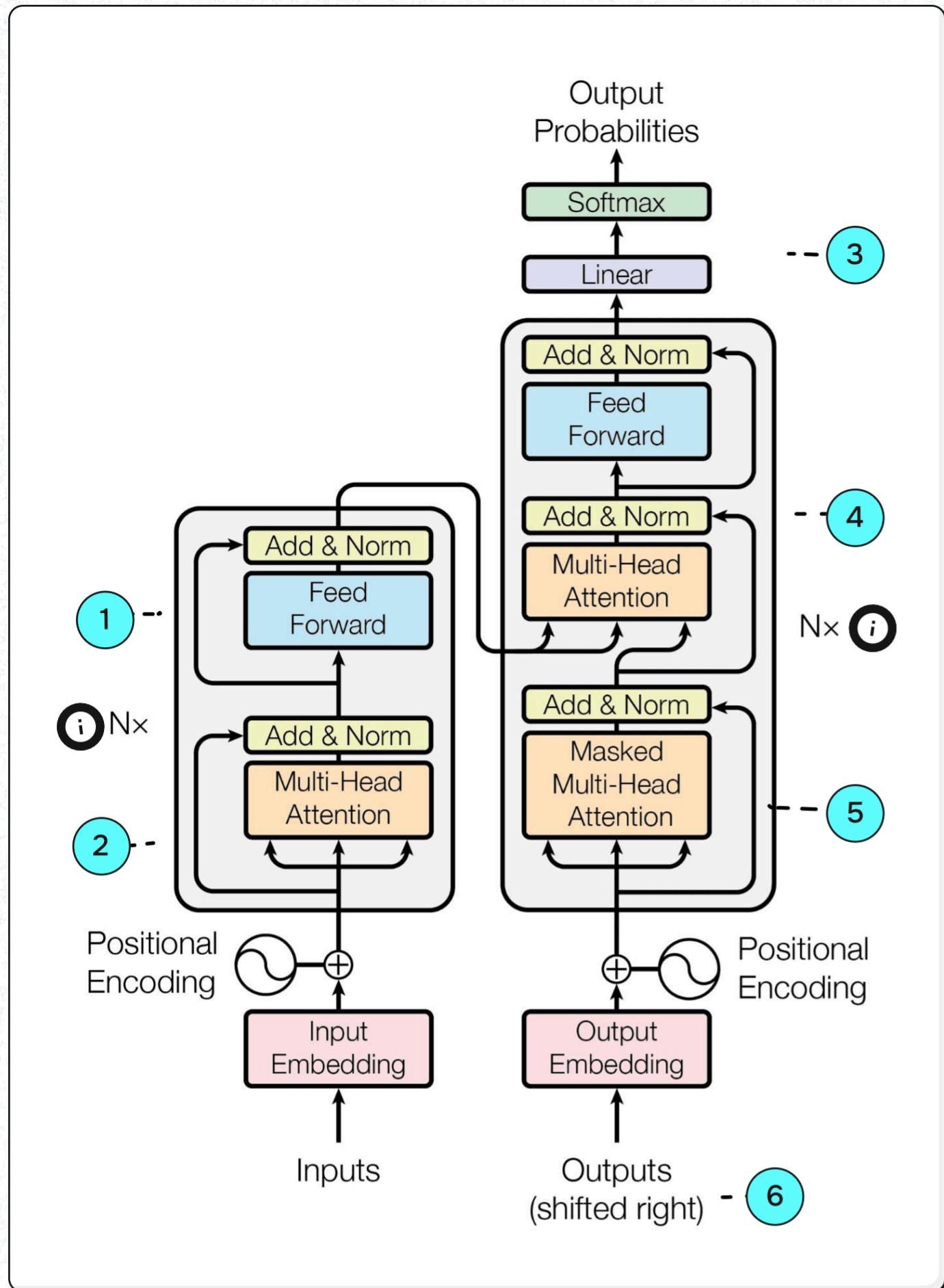
То, что мы разобрали выше – это **модель энкодера-декодера, построенная на RNN**, то есть рекуррентная сеть, в которой токены все ещё обрабатываются последовательно.

Обратите внимание, что внимание здесь используется не на всех шагах, а только для связи декодера с энкодером.

Отличие трансформера от этой картинки в том, что **в нем нет вообще никакой рекуррентности**: все токены обрабатываются параллельно благодаря тому, что внимание полностью заменяет последовательную обработку в энкодере и декодере.

Статья “Attention is All You Need” называется так именно потому, что авторы доказали, что одного лишь механизма внимания достаточно, чтобы полностью заменить рекуррентность в нейронных сетях и обрабатывать всю последовательность параллельно и эффективно.

Вот наш трансформер во всей красе, слева – энкодер, справа – декодер:



- 1 После внимания вектор каждого токена отдельно проходит через небольшие полно связные слои. На этом этапе сеть как бы обдумывает и обрабатывает информацию о внимании.
- 2 Здесь вычисляется Multi-Head Self-Attention. Каждый токен смотрит одновременно на все остальные токены во входной последовательности, вычисляя внимание. Так сеть понимает контекст.
- 3 После снова идут слои Feed Forward для обработки информации. Получившееся в итоге векторное представление затем проходит через линейный слой и функцию softmax, превращаясь в вероятности токенов.

- 4 Это уже знакомое нам Decoder-Encoder внимание. Декодер смотрит на выходы энкодера, чтобы понять, на какие входные слова нужно обратить внимание при генерации текущего токена.
- 5 Декодер смотрит на уже сгенерированные токены (то есть на предыдущие), вычисляя внимание. При этом используется маска (Masked), чтобы модель не «подсматривала» будущие токены.
- 6 shifted right означает, что для каждого токена входной последовательности мы предсказываем следующий, не подглядывая вперёд

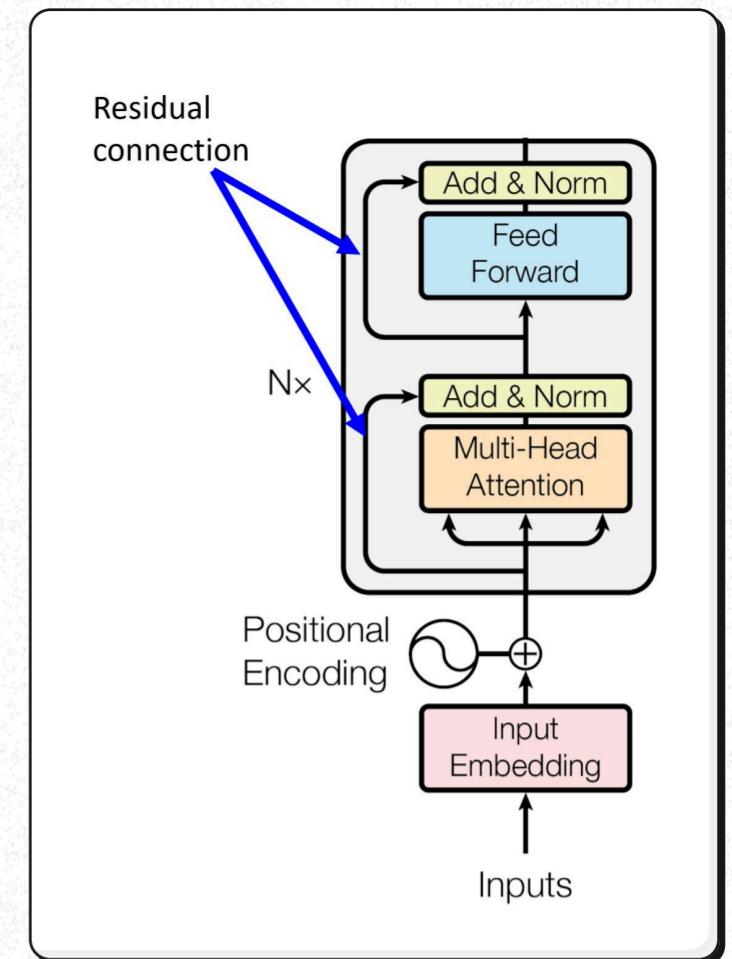
● Обе части состоят из нескольких (N) одинаковых блоков

И еще несколько пояснений к этой схеме:

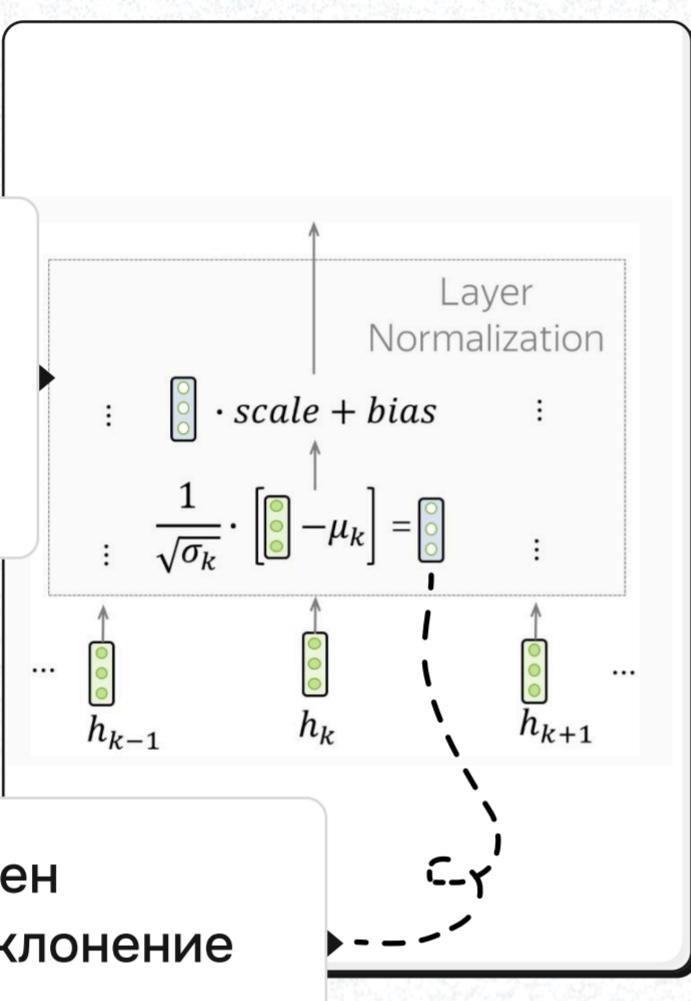
Остаточные соединения

На схеме выше вы можете иногда видеть стрелки, которые соединяют вход блока напрямую с его выходом, минуя промежуточные слои. Это так называемые Residual Connections (остаточные соединения).

Дело в том, что при обучении глубоких сетей информация часто начинает теряться или сильно меняться, проходя через много слоёв подряд. Остаточные связи решают эту проблему: они позволяют исходной информации «перескочить» через слой и попасть сразу на выход блока.



Меняем масштаб
с помощью
обучаемых
параметров



Делим каждый токен
на стандартное отклонение
и вычитаем среднее

Эмбеддинги и позиционные эмбеддинги

В самом низу схемы мы видим розовые кирпичи эмбеддингов, к которым прибавляется некий **Position Encoding**. Мы уже говорили, что эмбеддинги – это способ представить токен в виде вектора чисел так, чтобы модель могла понять его смысл.

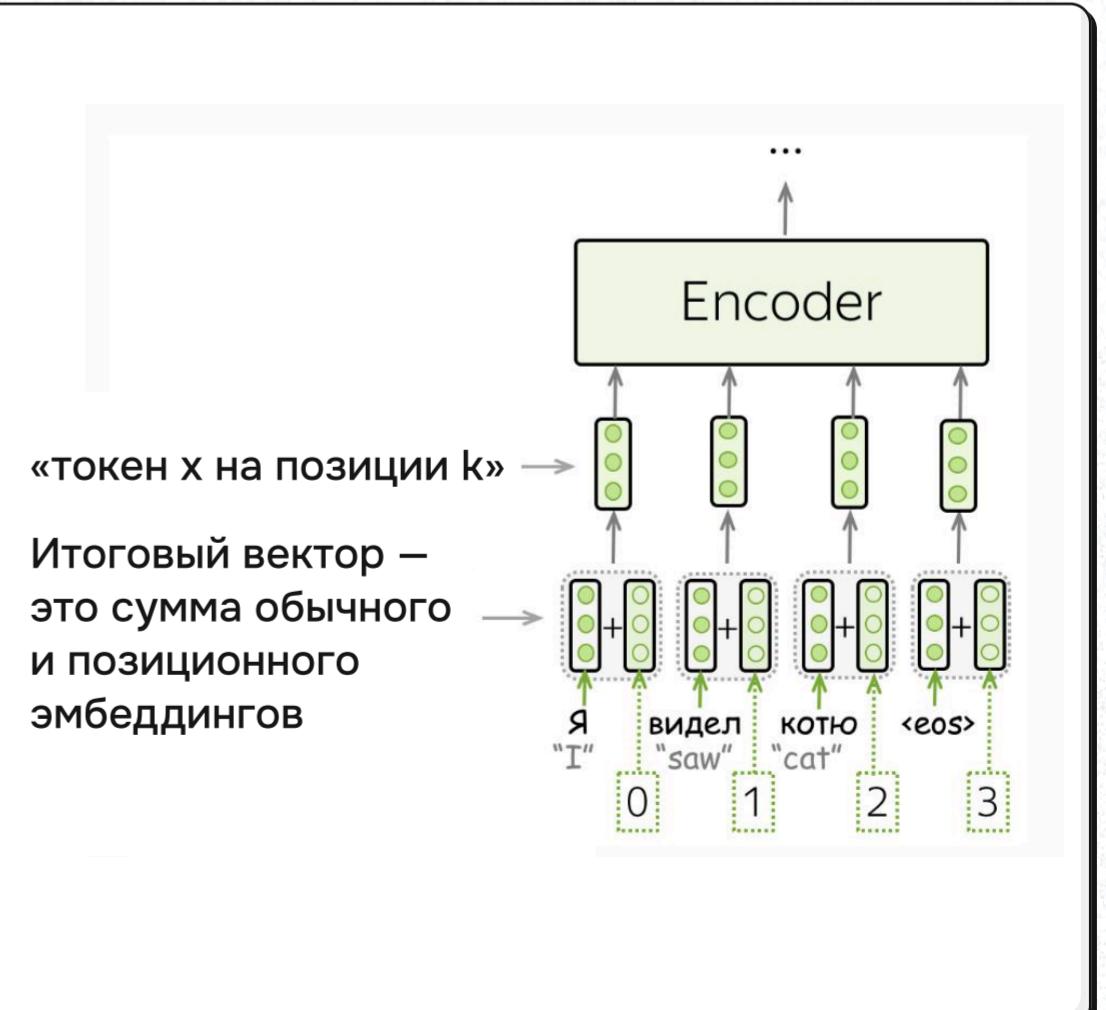
Однако такие векторы несут в себе только семантическую информацию и не учитывают позицию слова в предложении, потому что трансформер обрабатывает все токены одновременно. Для этого нам и нужно позиционное кодирование. Оно **указывает, в каком порядке идут слова**.

Нормализация

Разбросанные по схеме желтые кирпичики **«Add & Norm»** обозначают два действия, которые модель выполняет после каждого блока слоев:

Add – это сложение тех самых остаточных связей с выходами текущего блока.

Norm (Layer Normalization) – это операция, которая стабилизирует обучение нейросети. Она нормализует значения в каждом слое, чтобы они не становились слишком большими или слишком маленькими.

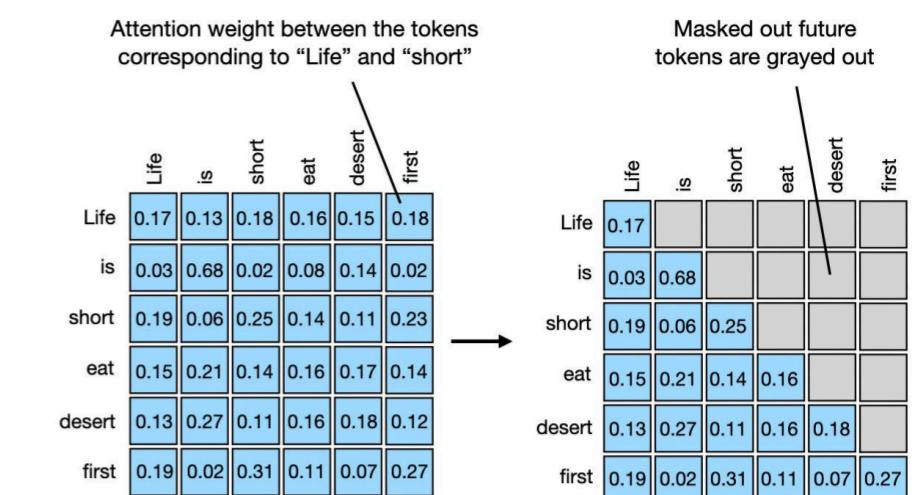


Маски в Masked Attention

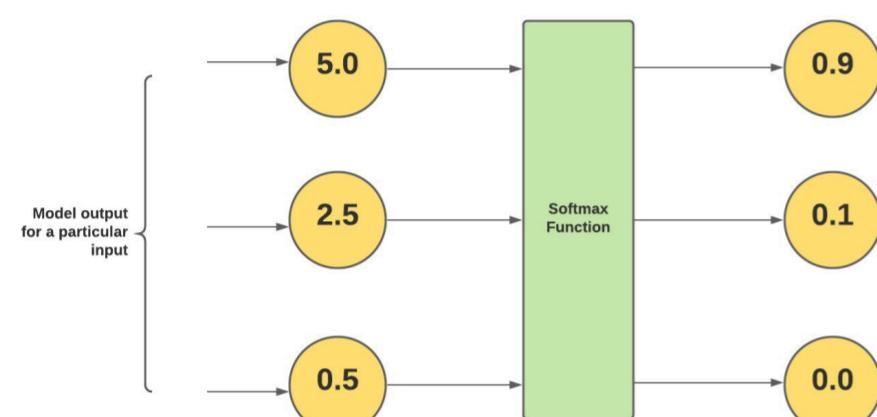
Для каждого токена входной последовательности декодер должен предсказать следующий. Но, вспомним, что у трансформера нет рекуррентности, и все токены обрабатываются одновременно, так что **при обучении в Decoder мы подаем всю выходную последовательность сразу.**

Чтобы в таком сценарии при вычислении внимания декодер не подглядывал вперед, нам и нужны маски. Это **специальные матрицы**, которые закрывают доступ модели к будущим токенам, которые она еще не “сгенерировала”.

Если модель пока сгенерировала только «Life», то доступ к attention этого слова с последующими ей пока закрыт.



Пример работы softmax:



Softmax

Когда модель вычисляет внимание, она получает числа, которые показывают, насколько сильно один токен связан с другими. Эти числа могут быть **произвольными** (положительными, отрицательными, большими и маленькими).

В конце нам нужно превратить эти числа в вероятности токенов. Тут на помощь приходит функция softmax. Именно она **масштабирует числа так, чтобы все они находились в промежутке от 0 до 1 и в сумме давали 1**: как настоящие вероятности.

Именно так выглядел трансформер, который учёные из Google изобрели в 2017 году. С тех пор архитектура почти не изменилась.

Трансформеры действительно очень сложно заменить, потому что они:

- обучаются быстро и параллельно;
- могут обрабатывать длинные последовательности;
- универсальны и масштабируемые.

Однако некоторые вариации архитектуры все-таки появились.

Вот три из них, о которых стоит знать:

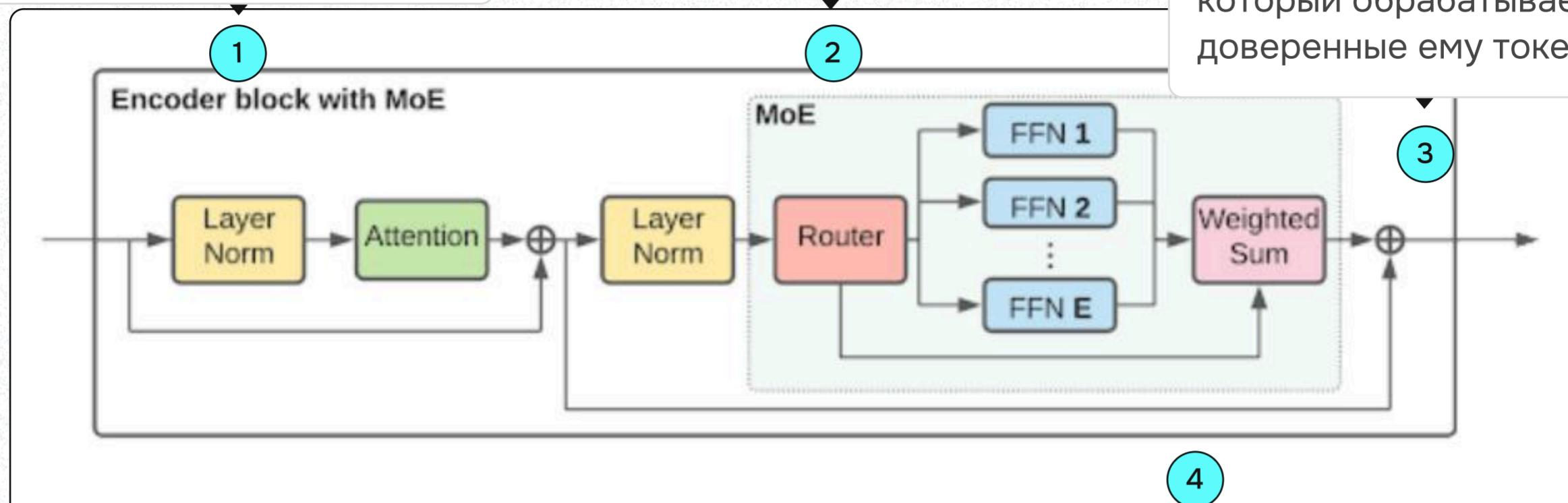
MoE (Mixture of Experts)

Подход, при котором внутри одной большой сети есть несколько маленьких специализированных сетей. Каждая такая сеть называется **экспертом**. Каждый эксперт обрабатывает только те данные, в которых он (как считает модель) разбирается лучше других.

В этой части все остается как есть, обычный трансформер. Внимание общее для всех токенов.

В этой части все остается как есть, обычный трансформер. Внимание общее для всех токенов.

Вместо одной полносвязной сети здесь появляется несколько. Каждая – эксперт, который обрабатывает доверенные ему токены.

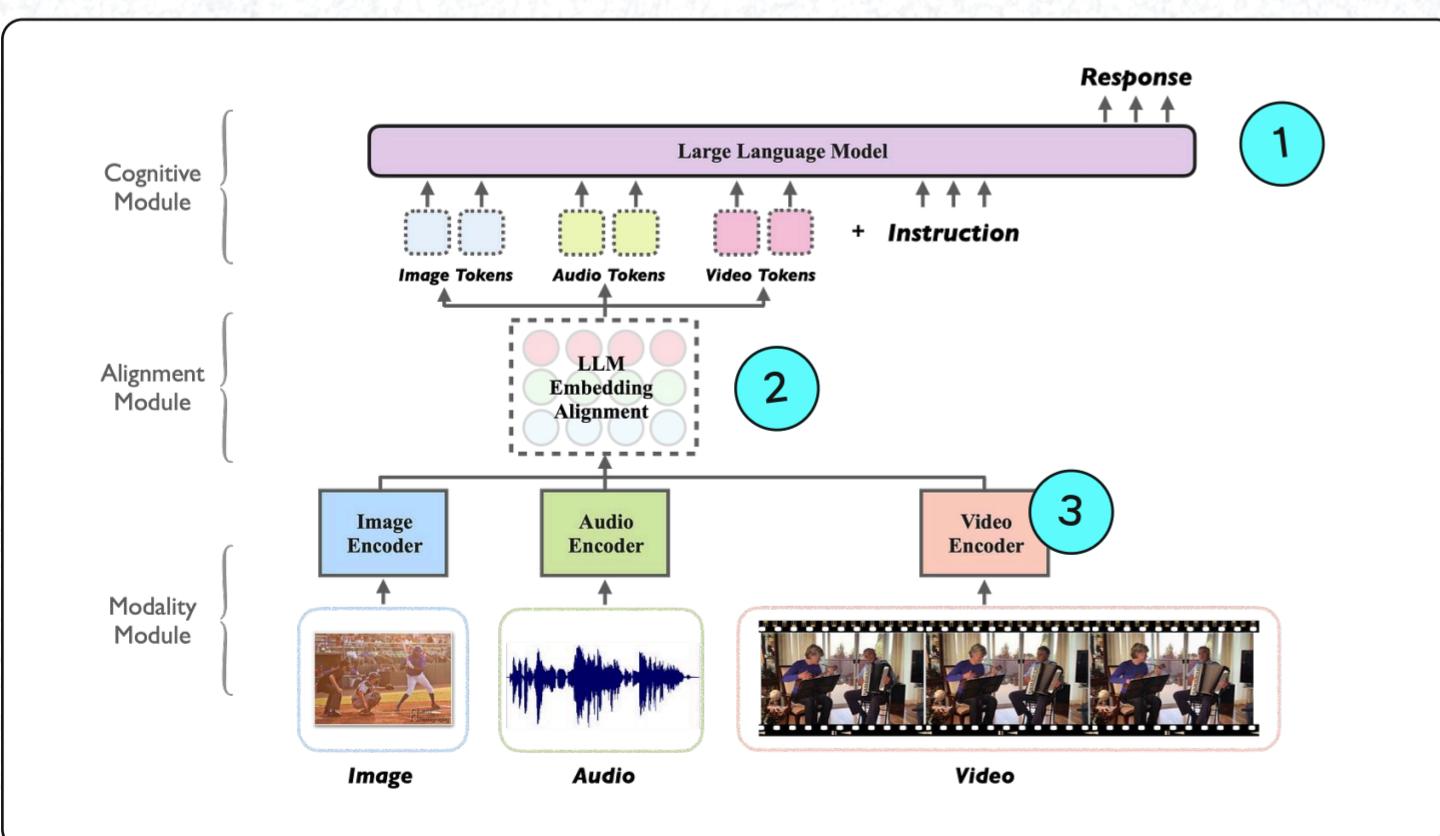
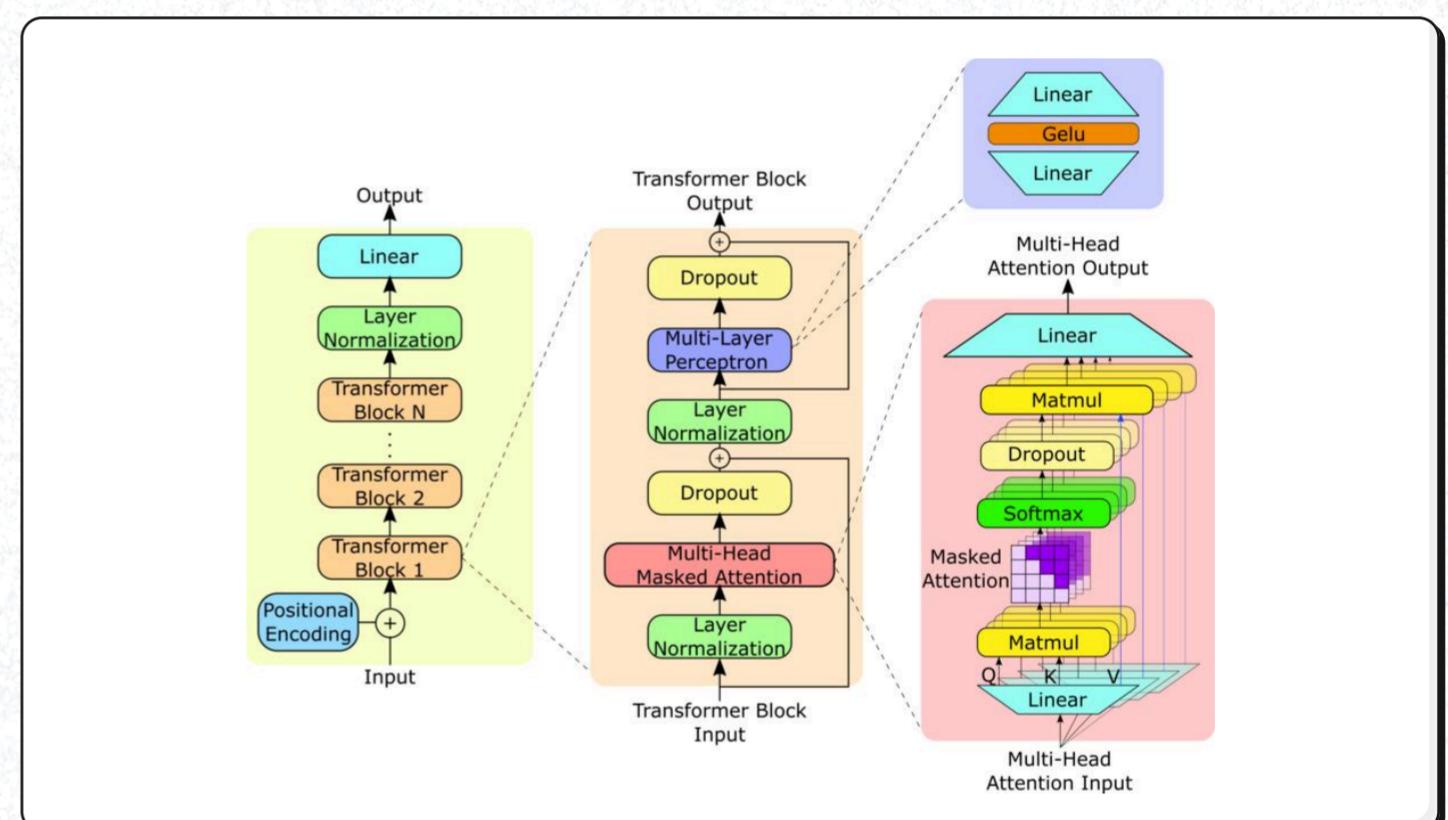


В итоге суммируем ответы всех экспертов.

Decoder-only

В таких архитектурах вся модель – это декодер. Энкодер отбрасывается вообще. Такие модели проще и быстрее обучаются, поэтому стали популярны. **Самый известный пример decoder-only модели – это GPT.**

Вот так выглядит полная архитектура. Как видите, все остаётся как было, только мы избавляемся от тяжеловесного энкодера.



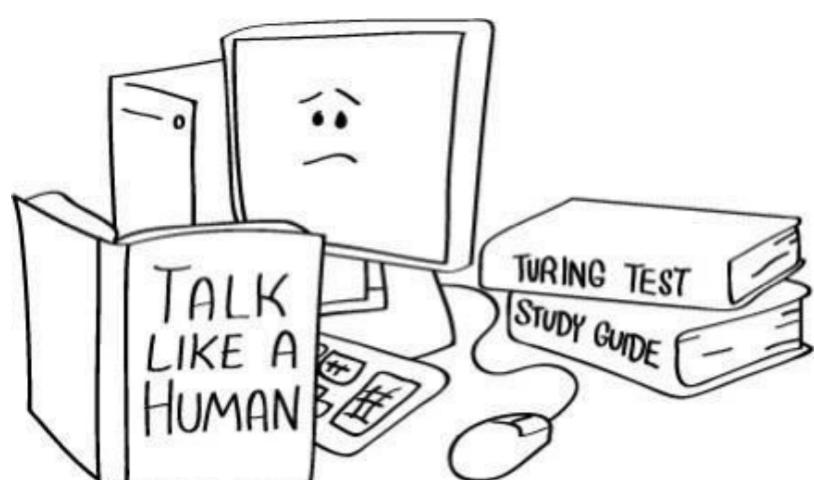
Мультимодальный трансформер

Сама архитектура здесь не меняется. Просто добавляются другие модальности (картинки, видео, аудио), которые нужно обрабатывать отдельно.

Из-за этого нам приходится множить энкодеры: каждый из энкодеров отвечает за свою модальность и переводит разные типы данных в эмбеддинги.

- 1 Склеиваем и подаём в LLM
- 2 Отражаем их в одно пространство (например, чтобы эмбеддинг картинки кота был похож на слово «кот»).
- 3 Энкодим разные виды входных данных в эмбеддинги.

Предобучение



С математикой процесса – разобрались.

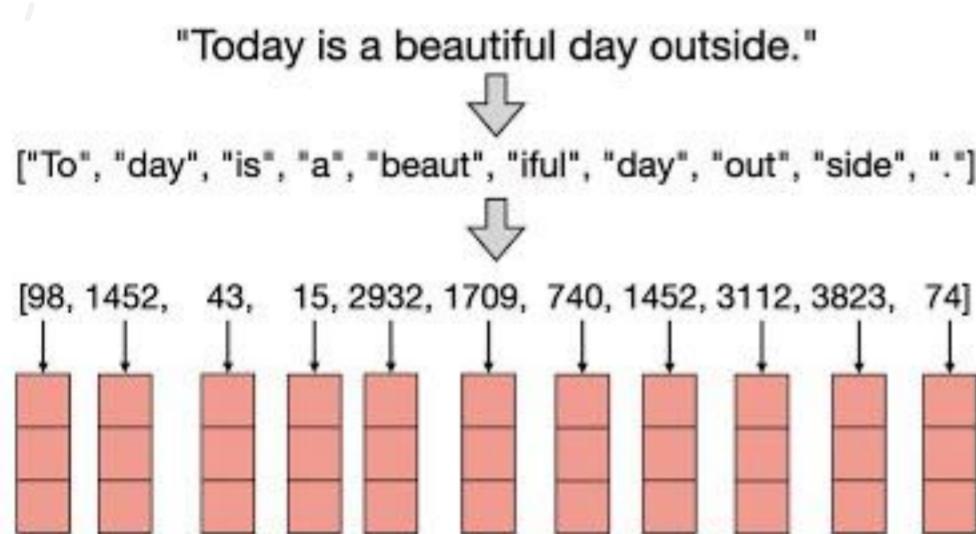
С архитектурой – тоже разобрались.

Осталось глубже погрузиться в сам процесс обучения. Ведь архитектура и математика – это только фундамент, чтобы получить умную модель, ей нужно «прочитать» огромное количество текстов и запомнить из них важную информацию. Этот этап называется **предобучением (pre-training)**.

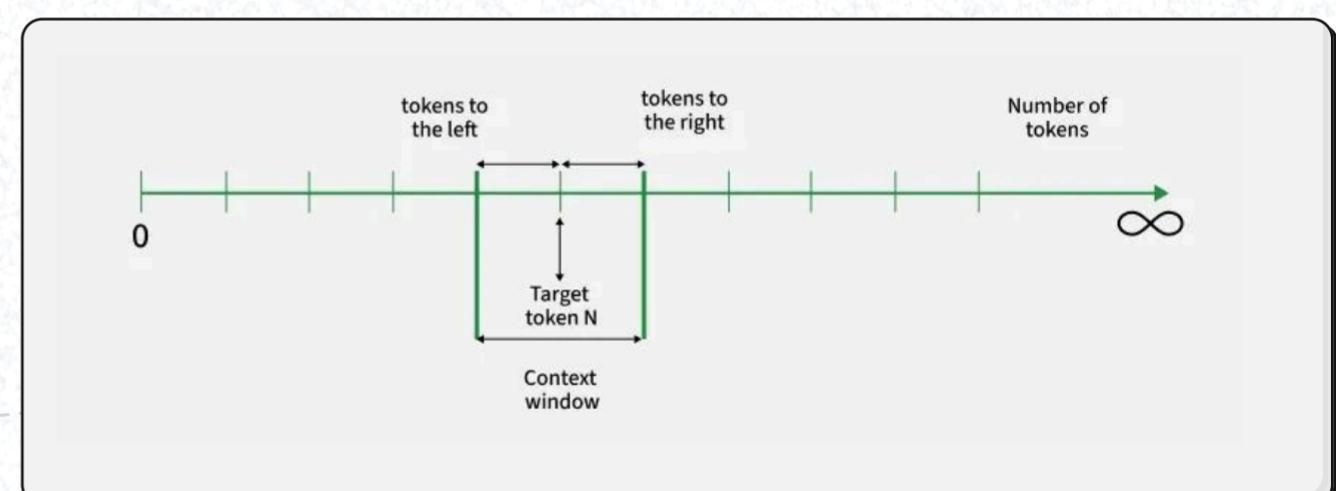
На этом этапе модель видит миллиарды символов из книг, интернета, научных статей и других источников и учится понимать язык на глубоком уровне. В этой главе мы подробно рассмотрим, как именно проходит процесс предобучения, какие данные для него используются, и что такое **self-supervised learning**.

Для начала еще раз про процесс обучения, на примере. Предположим, на вход нам пришел огромный текст (какая-нибудь книжка о Гарри Поттере). Как мы будем на нем учиться?

- 1 Во-первых, мы разобьём этот текст на **токены**, а токенам сопоставим их id.



- 2 Во-вторых, мы разделим токенизованный текст на кусочки (**чанки**). Например, по 512 токенов в каждом. Это наше **контекстное окно**. Чем больше окно, тем длиннее контекст, и тем больше информации модель учитывает при предсказании. У современных моделей контекст достигает сотен тысяч токенов.



- 3 Затем созданные нами чанки начинают поступать в модель для обучения. Модель получает задание: **предсказать следующий токен в каждом месте последовательности**.

Например, если на вход приходит последовательность токенов

[Гарри, Поттер, взял, волшебную, палочку, и,...]



...то последовательность предсказаний будет такой:

Гарри → Поттер
Гарри Поттер → взял
Гарри Поттер взял → волшебную

Во время инференса реальный следующий токен – не обязательно самый вероятный. Это можно контролировать с помощью специального параметра «температура» – чем она ниже, тем чаще модель выбирает просто самый вероятный следующий токен.

Это подходит для строгих задач, требущих краткого точного ответа. Ну а чем температура выше – тем ответы модели разнообразнее и креативнее.

- 4 Только тут следует помнить о том, что в трансформере нет последовательной обработки, поэтому модель видит весь текст сразу. При этом, чтобы избежать подглядывания при предсказании, применяются **маски** (их мы разбирали в предыдущем разделе).

Например, для токена «Поттер»:

Виден токен «Гарри»
Не видны «взял, волшебную, палочку,...»

- 5 Предсказания модель делает в виде **вероятностей** для каждого возможного следующего токена.

Например, предсказываем токен после «Гарри Поттер взял».

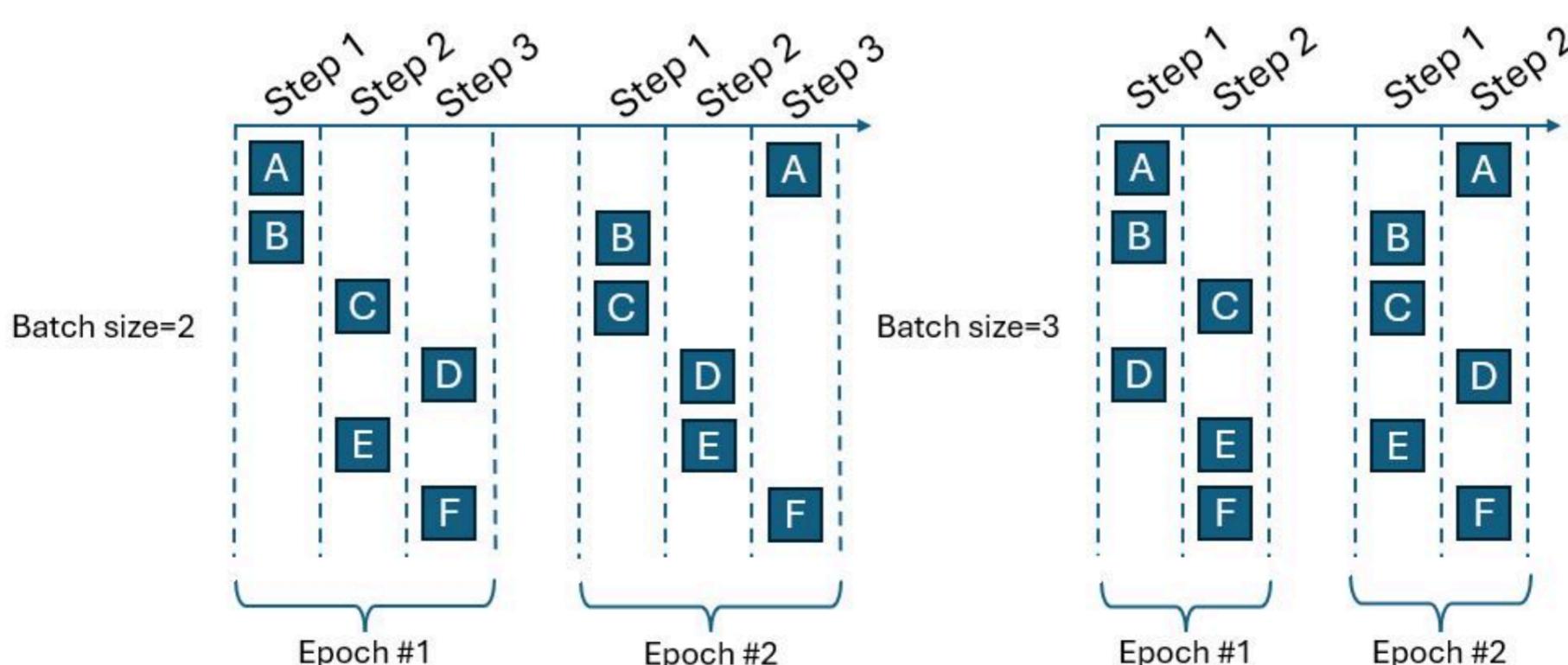
Модель выдаёт:
волшебную: 0.78
метлу: 0.15
шапку: 0.04
котёл: 0.03

- 6 **Вычисляем ошибку** предсказания таким способом, как мы разбирали в главе про математику и применяем **алгоритм обратного распространения ошибки**.

На практике такие чанки обрабатываются партиями (**батчами**). На вход модели поступает батч (например, это 32 окна по 512 токенов), который проходит через модель. Сначала прямой проход, затем обратный. Это один шаг обучения, называемый **итерацией**.

Спустя несколько итераций модель увидит весь датасет. Это называется **эпоха**. Обычно таких эпох по время обучения много, то есть мы несколько раз гоняем датасет по модели для того, чтобы она хорошо его усвоила.

Хороший пример, иллюстрирующий логику итераций и эпох на разных размерах батча:



Сколько будет эпох и какого размеры брать батчи – решает разработчик. Тут важно поймать баланс: слишком большой батч может не вместится в память, слишком маленький – это неэффективное использование памяти. С эпохами тоже все не очень просто.

Если их слишком мало – модель может недообучиться. Если много – переобучится, и снова будет плохо. Приходится экспериментально искать золотую середину.

Обратите внимание, что в алгоритме выше мы используем **данные без меток**, то есть обычные тексты, никем специально не размеченные. Модель просто учится предсказывать следующий токен, для этого ей не требуются человеческие метки. Такой подход называется **self-supervised learning** (самообучение).



Главное преимущество self-supervised learning – это огромное количество доступных данных.

Нет необходимости в ручной разметке, что позволяет использовать миллиарды текстов для обучения мощных моделей вроде GPT.



Но есть и нюанс: модель учится исключительно на закономерностях в данных. Если данные некачественные, ограниченные или содержат предвзятости (*bias*), то модель обязательно перенимает эти недостатки.

Модель может легко усвоить нежелательные стереотипы, ошибки или предрассудки, которые присутствуют в огромном количестве реальных текстов, поскольку ей никто специально не показывает, что правильно, а что нет.

Именно поэтому сбор данных – это, пожалуй, самое важная составляющая процесса предобучения.

Датасеты должны быть чистые, разнообразные, сбалансированные и тщательно отобранные.

Откуда берутся данные?

1. Интернет (архив страниц интернета Common Crawl, новости, блоги, социальные сети, форумы)
2. Книги и научные статьи
3. Википедия
4. Другое: код с GitHub, транскрипции видео и фильмов, тексты песен и т.д.

Как их обрабатывают?

1. Удаление дубликатов (данные обычно содержат множество повторяющихся фрагментов, а это чревато переобучением)
2. Фильтрация нерелевантного и мусорного контента (реклама, ошибки кодировки, случайные наборы символов)
3. Фильтрация по языку или теме. Для этого, а также для оценки грамотности, часто обучаются специальные классификаторы.
4. Фильтрация токсичных данных / оскорблений / неэтичного контента.

Почему чем больше данных – тем лучше?

Сегодня для нас совершенно ясно: чем больше данных видела языковая модель и чем в ней больше параметров, тем она умнее.

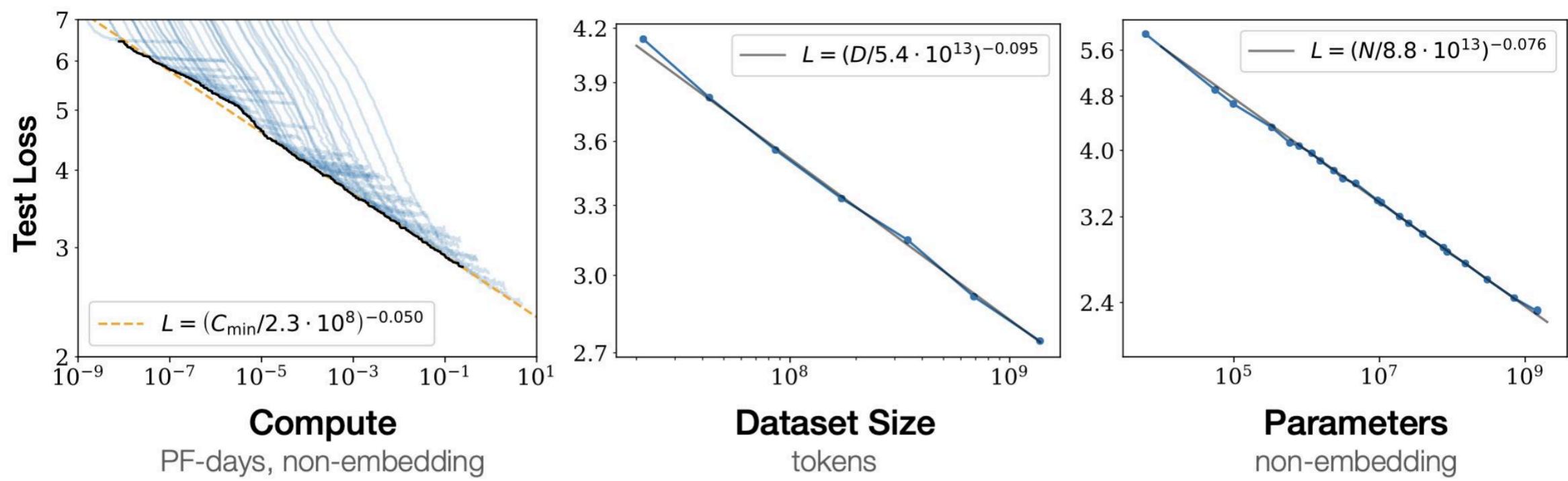
Но до 2020 года это было совсем неочевидно, а гипотеза и масштабирования была лишь гипотезой.

И вот почему

Считалось, что:

- Большие модели быстро начнут переобучаться.
- После какого-то размера перестанут эффективно обучаться и вообще не смогут улучшаться.
- Данные начнут повторяться, и модель просто запомнит их и перестанет «понимать» язык.

Но выяснилось, что LLM ведут себя совсем иначе:



Это картинка из статьи «**Scaling Laws for Neural Language Models**» 2020 от учёных из OpenAI. Именно в ней впервые было доказано, что качество языковых моделей напрямую зависит от трёх факторов:

- Количество параметров (мощность модели)
- Количество данных (объём обучающих текстов)
- Вычислительных ресурсов (GPU/TPU часы)

То есть оказалось, что, увеличивая данные и параметры модели, можно практически бесконечно улучшать качество, причём эти улучшения хорошо описываются простыми математическими закономерностями. Это называется **Scaling Laws**.

Получается, что даже гигантские модели (миллиарды и триллионы параметров) продолжают обучаться и улучшаться без серьёзного переобучения, и чем больше данных – тем дольше модель сохраняет способность учиться и улучшаться. Но есть и минусы:

Данные ограничены

Количество текста в мире кажется огромным, но на самом деле открытые качественные данные имеют предел. Можно использовать синтетику, но она менее эффективна.

Для обучения нужны огромные вычислительные ресурсы

Современные гигантские модели, такие как GPT-4.5, требуют для обучения целых кластеров из тысяч GPU. Это стоит миллиарды долларов (а это приводит к монополии корпораций) и, кроме того, генерирует огромное количество выбросов вредных веществ в окружающую среду.

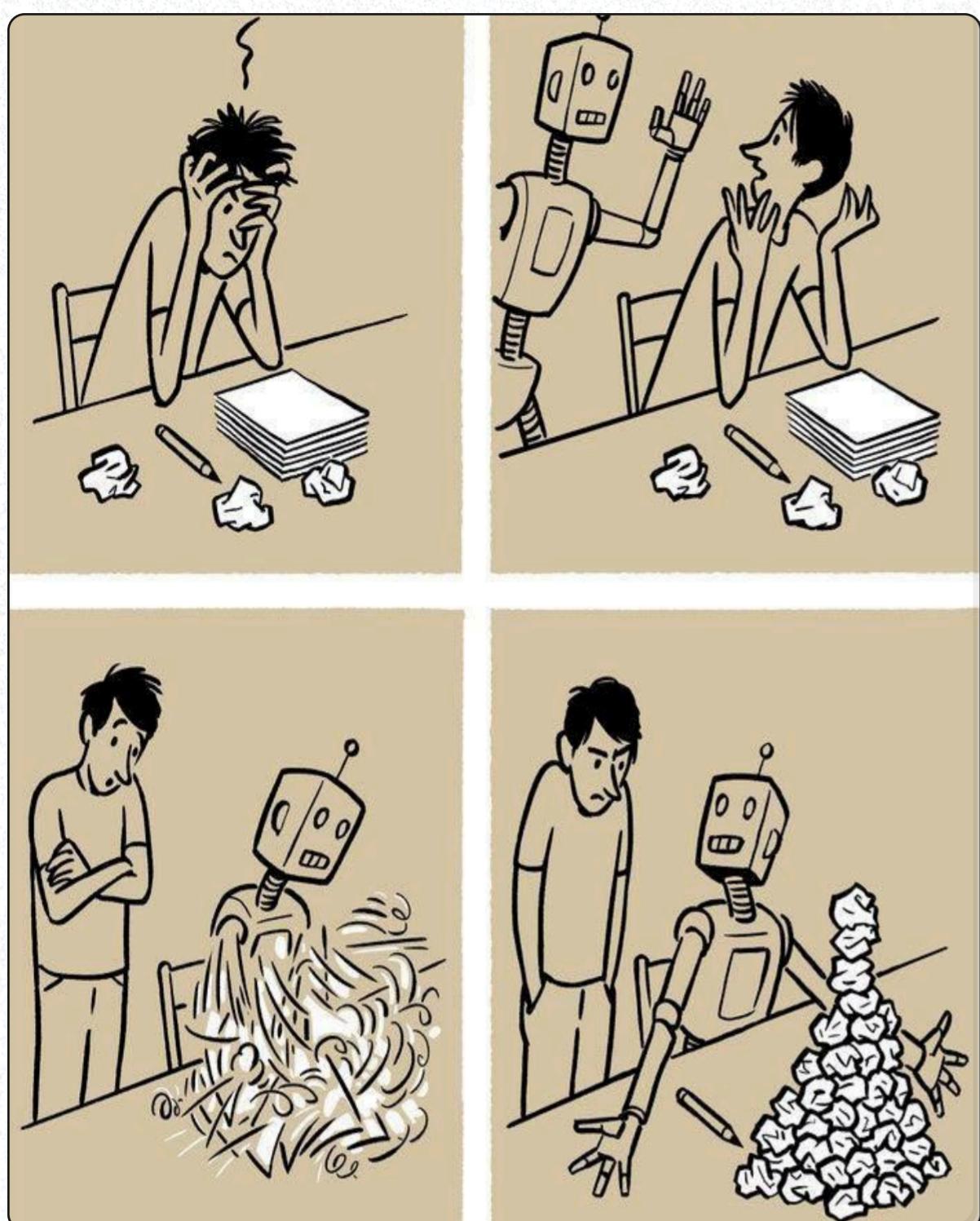
Чем больше модель – тем сложнее ей пользоваться

Гигантские модели требуют огромного количества памяти и мощностей даже на этапе инференса. Это усложняет их практическое применение и часто делает их недоступными для простого потребителя.

На данный момент мы так и не научились решать эти проблемы. Поэтому – все впереди.

Кроме того, предобучение – это ещё не все. В обучении моделей присутствуют и другие этапы. О них поговорим в следующих главах.

Файн-тюнинг



На этапе предобучения модель уже узнала много полезного: она изучила язык, разобралась в закономерностях текстов и может довольно слаженно разговаривать.

Однако надо понимать, что пока что ей просто скормили много текста. Её не обучали следовать инструкциям, решать задачи или вести себя определённым образом.

Чтобы модель стала действительно полезной для пользователя, её нужно дополнительно «донастроить». Именно этот этап и называется файнтюнингом (fine-tuning).

Файнтюнинг – это процесс, когда мы берём уже предобученную модель и дополнительно обучаем её на гораздо меньшем, но специализированном наборе данных с конкретными примерами решений нужных задач. В этой главе разбираемся, как именно это работает.

Технически, файнтюнинг – это когда мы берем готовую модель с ее обученными параметрами и немного подкручиваем некоторые из них.

Файнтюнинг используется

С самими производителями моделей

Сторонними разработчиками и исследователями

Для того, чтобы улучшить качество и полезность своих моделей в конкретных сценариях.

Например, с помощью файнтюнинга модели учат лучше решать задачи по программированию или математике.

Также файнтюнинг используется для элаймента, то есть для того, чтобы сделать модель более безопасной и снизить количество нежелательных ответов.

Для того, чтобы адаптировать уже готовую модель под свои конкретные бизнес-задачи или доменные области.

Так можно создавать собственные специализированные продукты, не обучая модель с нуля.

Правда, возможна такая экономия только с открытыми моделями, то есть с теми, веса которых выложены в открытый доступ. Ну или вендоры должны предоставить специальное API.

Файнтюнинг бывает двух видов:

1 Unsupervised fine-tuning

Это дообучение модели **без использования размеченных данных**, на текстах, которые модель должна понять или с которыми должна адаптироваться работать.

Например, если в претрейне у модели было мало текстов по математике, то на этапе файнтюнинга можно просто дообучить ее на математических учебниках, и понимание предмета уже немного улучшится.

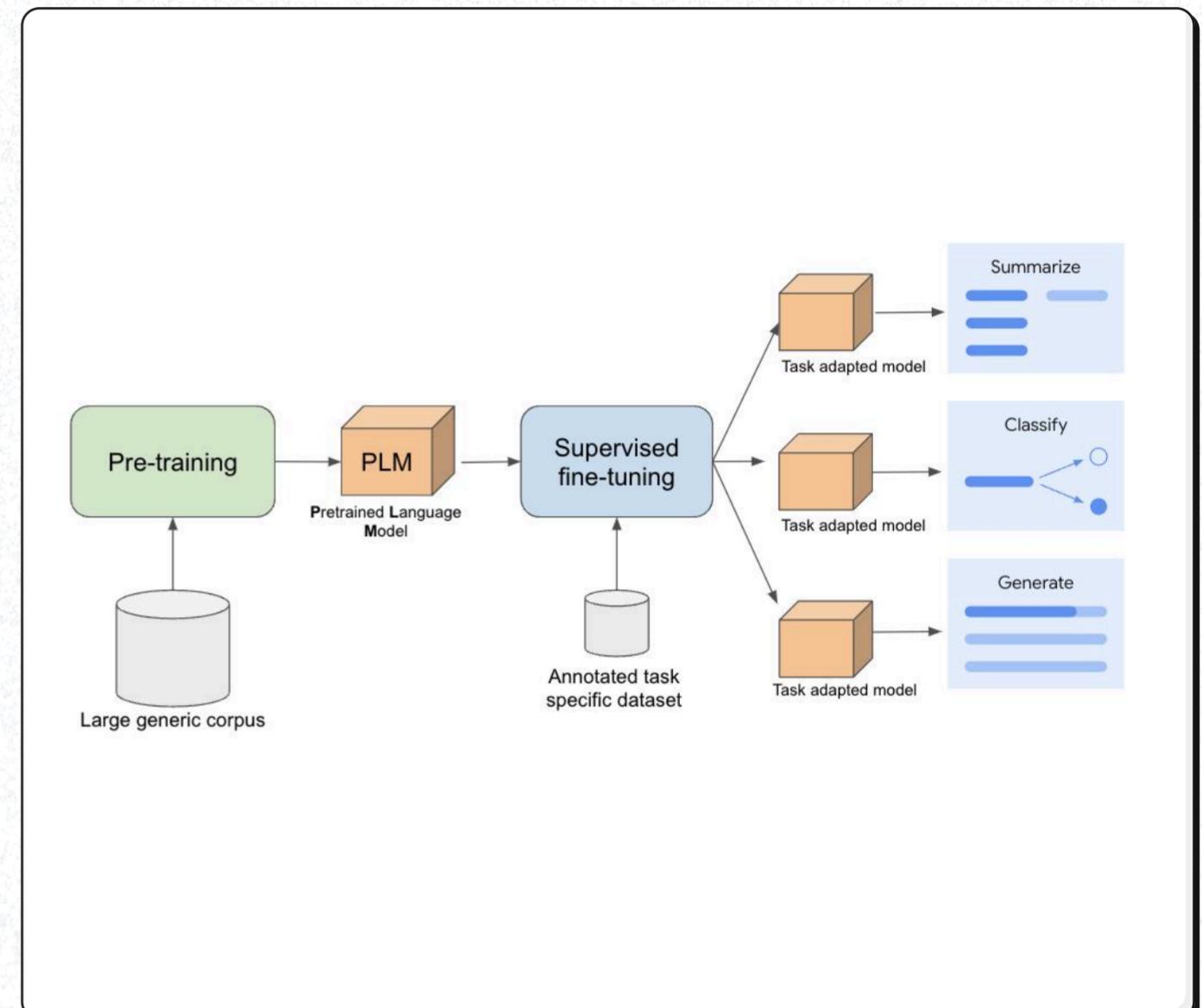
Такой вид файнтюнинга используется в условиях нехватки размеченных данных, ну или когда ее просто нужно адаптировать под новый домен знаний. Например, так можно дообучить на внутренней документации компании.

2 Supervised fine-tuning

Это уже более распространённый вид настройки. Здесь дообучение модели происходит на **размеченных данных**, в которых явно указаны входные данные и желаемый выходной результат (метки, ответы, классы и тд).

Сложность подготовки данных здесь выше, потому что в датасете уже не сырой текст, а обработанные людьми **пары вида input-output**.

С помощью SFT можно научить модель с высокой точностью решать конкретные задачи и следовать инструкциям, и именно SFT чаще всего используют после этапа предобучения.

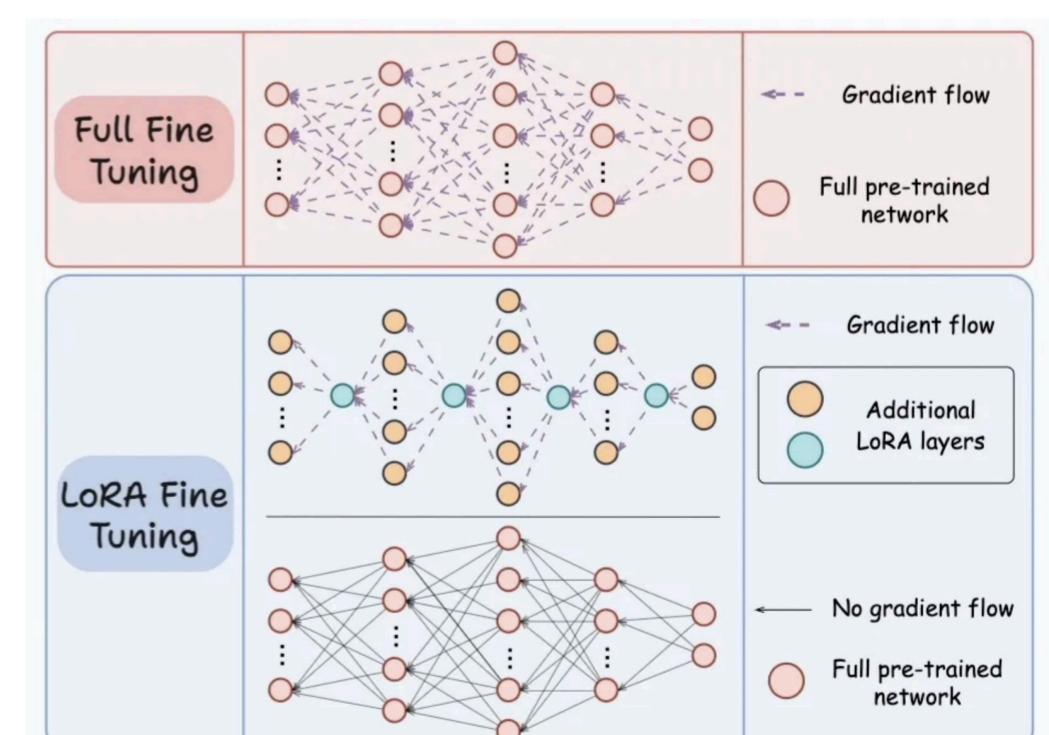


Техники файнтюнинга, в свою очередь, тоже могут быть разные. Вот две основные:

Это полный файн-тюнинг, то есть полноценная донастройка весов предобученной модели под новые данные. Метод проверенный и используется очень давно, но стоит такой файнтюнинг очень дорого.

Это LoRA (Low-Rank Adaptation). Основная идея метода заключается в том, чтобы разложить весовые матрицы (некоторые или все) исходной модели на матрицы низкого ранга и обучать уже их. По сравнению с полноценным FT LoRA экономичнее, требует меньше данных и быстрее сходится.

Сейчас LoRA и её варианты (QLoRA, gLoRA и тд) – это основной стандарт индустрии, особенно в условиях ограниченных ресурсов.



В отличие от предобучения, для которого нужны огромные вычислительные ресурсы и миллиарды токенов данных, файнтюнинг вполне реально провернуть дома или в облаке за разумные деньги, если выбрать не слишком большую модель. Именно поэтому файнтюнинг так популярен среди разработчиков и open-source энтузиастов.

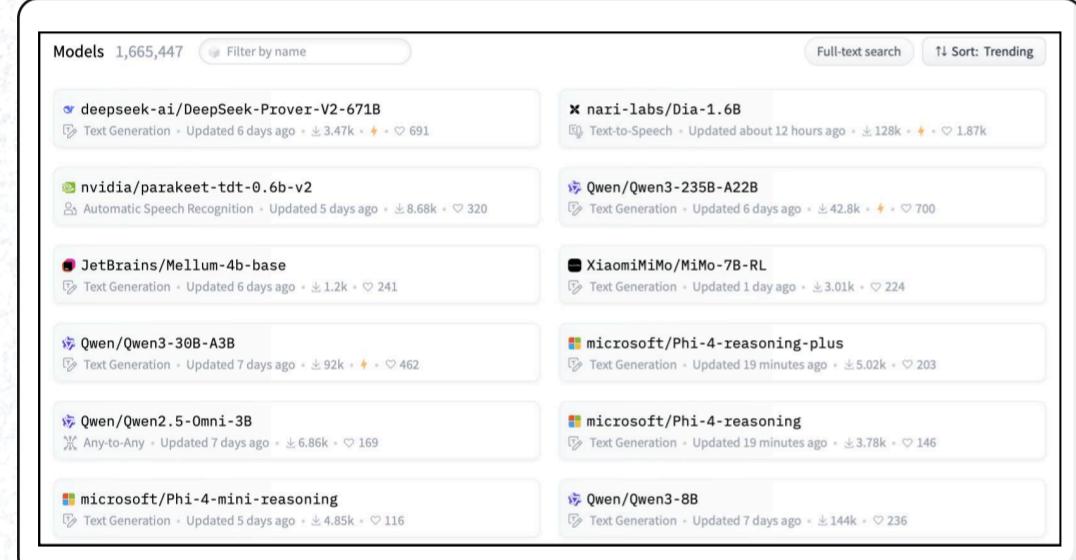
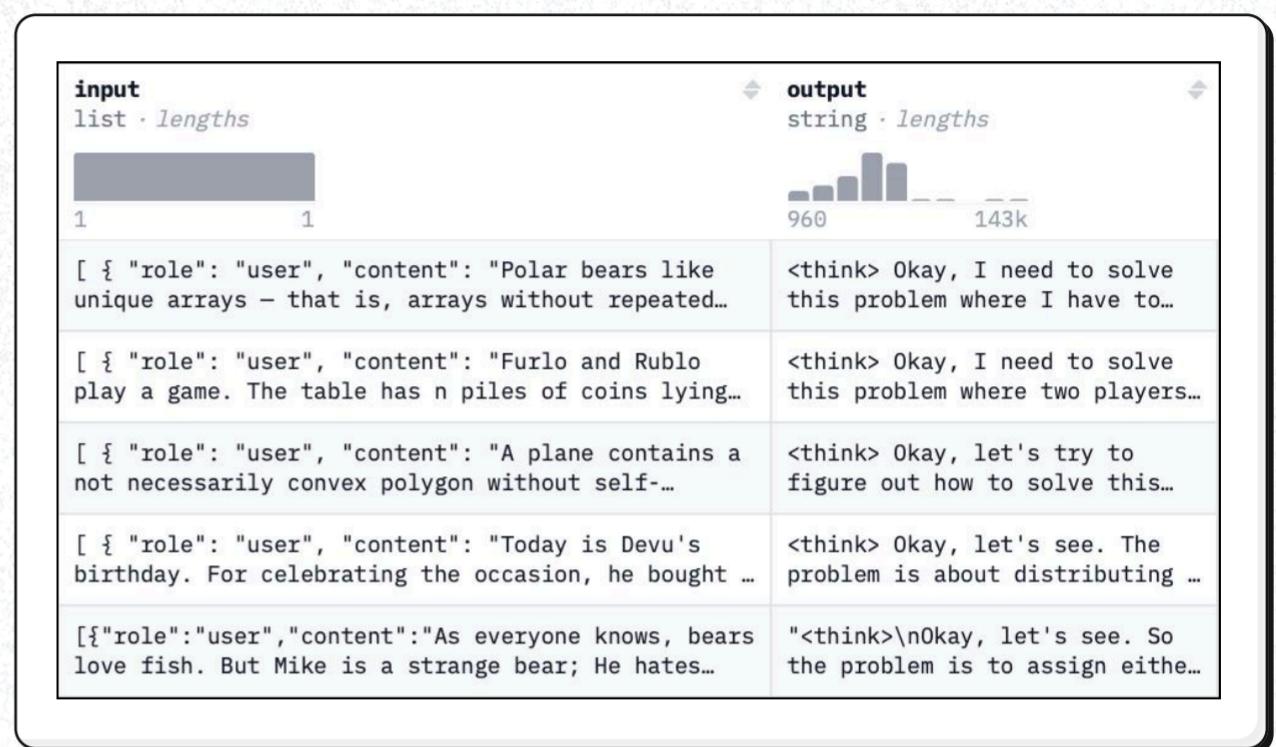
Учитывая это, в этой главе мы не будем останавливаться на теории, а подробно разберём, как самостоятельно зафайнтюнить модель под свою задачу, используя понятные и доступные инструменты.

Итак, пошаговый гайд по тому, как зафайнтюнить модель:

1 Определяем задачу и собираем данные

На этом этапе определяем, какую именно задачу должна решать наша модель, и собираем релевантные размеченные данные в формате вопрос-ответ. Можно разметить все самостоятельно или найти датасет в открытых источниках (например, <https://huggingface.co/datasets> – это библиотека из 380к наборов данных на любой вкус).

Вот пример подходящего датасета



2 Выбираем предобученную модель

Для этого заходим на

<https://huggingface.co/models> и выбираем модель, которая подойдет под ваши GPU-ресурсы. Для файнтюна в домашних условиях лучше брать небольшую модель, до 8 миллиардов параметров.

3 Готовим окружение

На этом шаге просто убедитесь, что у вас есть доступ к GPU (например, можете работать в Google Colab).

4 Загружаем модель и токенизатор

На этом шаге мы наконец-то мы добрались до кода. Вот скрипт, который подготовливает модель к дообучению:

```
1 from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
2
3 bnb_config = BitsAndBytesConfig(load_in_4bit=True,
4     bnb_4bit_use_double_quant=True,
5     bnb_4bit_compute_dtype=torch.bfloat16)
6
7 model = AutoModelForCausalLM.from_pretrained('mistralai/Mistral-7B-v0.1',
8     quantization_config=bnb_config,
9     device_map='auto')
10
11 tokenizer = AutoTokenizer.from_pretrained('mistralai/Mistral-7B-v0.1')
12 tokenizer.pad_token = tokenizer.eos_token
```

Чтобы сэкономить VRAM, модель будем грузить в 4-bit формате

Грузим модель, автоматически раскидывая слои по GPU

Загружаем соответствующий модели токенизатор и добавляем так называемый pad_token: он нужен для корректной обработки батчей

5 Займемся данными

Их нужно загрузить и предобработать:

```
1 from datasets import load_dataset
2
3 dataset = load_dataset('json', data_files='train.json', split='train')
4
5 def format(example):
6     return {
7         "text": f"<s>[INST] {example['instruction']} [/INST] {example['output']}</s>"
8     }
9 dataset = dataset.map(format)
```

Грузим локальные данные. Здесь они в формате json вида {"instruction": "...", "output": "..."}.

Данные форматируем под модель. В данном случае нужно обернуть каждый пример в специальные токены

[INST] ... [/INST]. Это важно для обучения модели правильно понимать промпт-ответ.

6 Теперь приготовим сам файнтюн

Файнтюнить модель будем методом LoRA. Для этого мы внедряем маленькие обучаемые адаптеры в определённые слои модели, а остальные веса замораживаем. Вот скрипт, который настраивает такой адаптер:

```
1 from peft import LoraConfig, get_peft_model, TaskType
2
3 lora_config = LoraConfig(r=64,
4     lora_alpha=16,
5     target_modules=['q_proj', 'v_proj'],
6     lora_dropout=0.05,
7     bias='none',
8     task_type=TaskType.CAUSAL_LM)
9
10 model = get_peft_model(model, lora_config)
11 model.print_trainable_parameters()
```

r – это главный параметр, определяющий размер встраиваемого адаптера. LoRA заменяет обучение большой матрицы весов на обучение двух маленьких матриц A ($d \times r$) и B ($r \times d$), где d – размер слоя.

target_modules – это список слоёв модели, в которые будут внедрены LoRA-адаптеры. В данном случае выбираем Query и Value проекции

Этой командой можно посмотреть, сколько параметров в итоге будем обучать

7 Обучаем!

Мы подготовили данные, адаптер LoRA и предобученную модель. Теперь настало время все объединить

```
1 from transformers import TrainingArguments  
2  
3 training_args = TrainingArguments(  
4     output_dir='./mistral_lora',  
5     per_device_train_batch_size=2,  
6     gradient_accumulation_steps=8,  
7     logging_steps=10,  
8     save_steps=100,  
9     num_train_epochs=3,  
10    fp16=True,  
11    save_total_limit=2,  
12    report_to="none"  
13 )  
14  
15 from transformers import Trainer, DataCollatorForLanguageModeling  
16  
17 trainer = Trainer(  
18     model=model,  
19     train_dataset=tokenized,  
20     args=training_args,  
21     data_collator=DataCollatorForLanguageModeling(tokenizer, mlm=False),  
22 )  
23  
24 trainer.train()
```

Обозначаем параметры обучения: сколько будет эпох, какой будет размер батча, куда загружать обученную модель и так далее.

Запускаем обучения с нашим датасетом, параметрами обучения и предобученной моделью с адаптером

8 Осталось протестировать, что у нас вышло

```
1 from transformers import pipeline  
2 from peft import PeftModel  
3  
4 base_model = AutoModelForCausalLM.from_pretrained(  
5     "mistralai/Mistral-7B-v0.1",  
6     quantization_config=bnb_config,  
7     device_map="auto"  
8 )  
9  
10 model = PeftModel.from_pretrained(base_model, "mistral_lora_finetuned")  
11  
12 pipe = pipeline("text-generation", model=model, tokenizer=tokenizer)  
13  
14 pipe("[INST] Как работает LoRA? [/INST]", max_new_tokens=100)[0]["generated_text"]
```

Грузим базовую модель в том же виде, что и раньше (это важно).

Все, можно пользоваться. Например, на выходе этой строки мы получим что-то вроде «LoRA добавляет обучаемые адаптеры слоям».

Полноценный ноутбук со всем кодом, который мы только что показали, можно найти здесь

<https://colab.research.google.com/github/brevdev/notebooks/blob/main/mistral-finetune-own-data.ipynb>.

В целом **этот пайплайн – универсальный**, потому что его можно адаптировать под любую LLM, любой набор данных и даже разные задачи (чат-боты, инструкции, классификация).

Однако файнтюнинг – не единственный способ донастройки моделей. В последнее время всё большее внимание получают подходы, основанные на **обучении с подкреплением (reinforcement learning, RL)**. О том, как они работают и почему стали настолько популярными – в следующей главе.

6 Обучение с подкреплением и ризонинг



Файнтюнинг отлично справляется с задачей адаптации модели под конкретные сценарии. Однако у него есть важное ограничение: файнтюнинг учит модель воспроизводить правильные ответы из заранее подготовленных данных, но не всегда даёт ей понимание, какие ответы действительно «хорошие», полезные и подходящие с точки зрения человека.

Поэтому в индустрии после этапа файнтюнинга на сцену обычно выходит **Reinforcement Learning (RL) – обучение с подкреплением**. В отличие от простого файнтюнинга, RL не просто показывает модели примеры хороших ответов, а напрямую даёт ей обратную связь о том, насколько её ответ был полезным или верным. Благодаря этому модель постепенно учится понимать, какие генерации наиболее подходящие и почему.

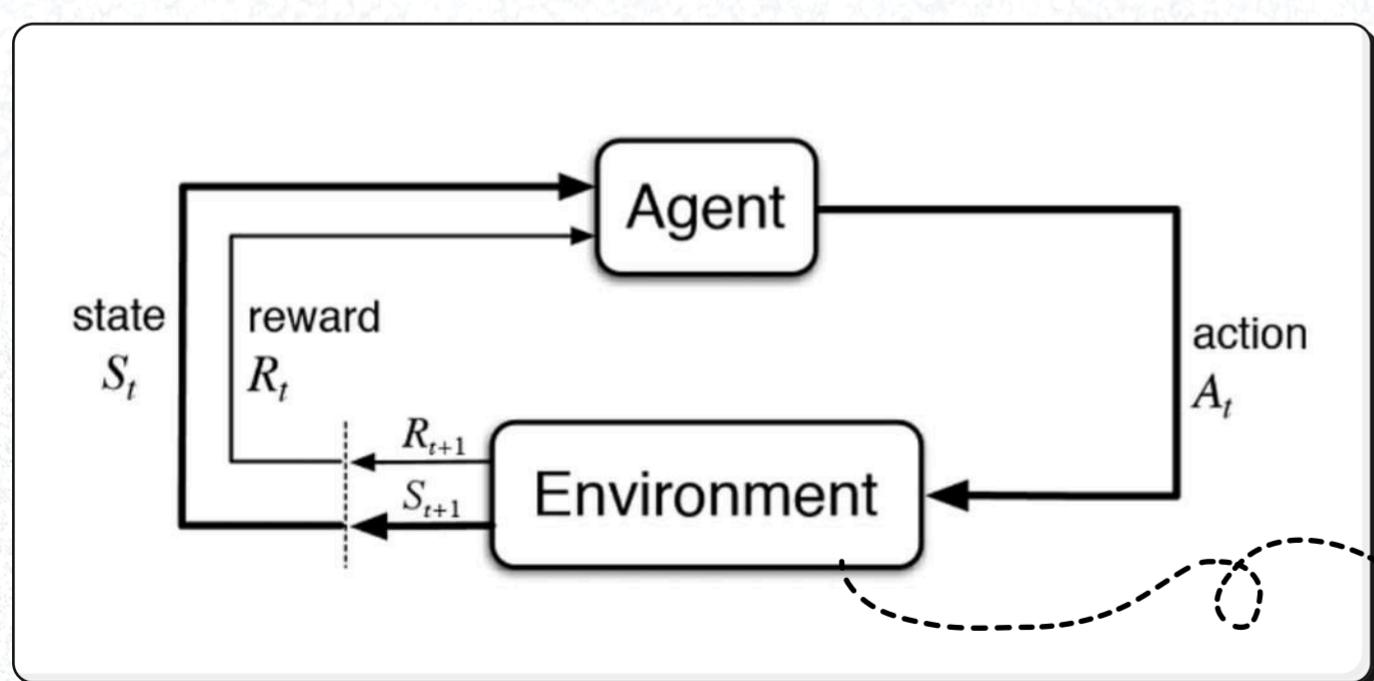
Особенно ярко RL проявил себя как основной инструмент в обучении ризонинг-моделей, таких как o3 или DeepSeek R1. Ризонинг – не просто очередной тренд, это новая парадигма масштабирования LLM, которая помогает моделям не только воспроизводить знания, но и использовать их, решая сложные задачи пошагово и логично.

В этой главе мы подробно разберёмся, как работает RL, и как он помогает LLM выйти на совершенно новый уровень.

Основы обучения с подкреплением

В RL есть две основные составляющие:

- 1 Агент
- 2 Среда



После получения обратной связи от среды агент корректирует своё поведение (это называется **политикой**) так, чтобы в будущем получать больше положительных наград.

Агент (в нашем случае это **языковая модель**) совершает некоторое действие.

Далее среда каким-то образом “реагирует” на действие агента, выдавая ему некоторую **награду (reward)**. Награда может быть и положительной, и отрицательной.

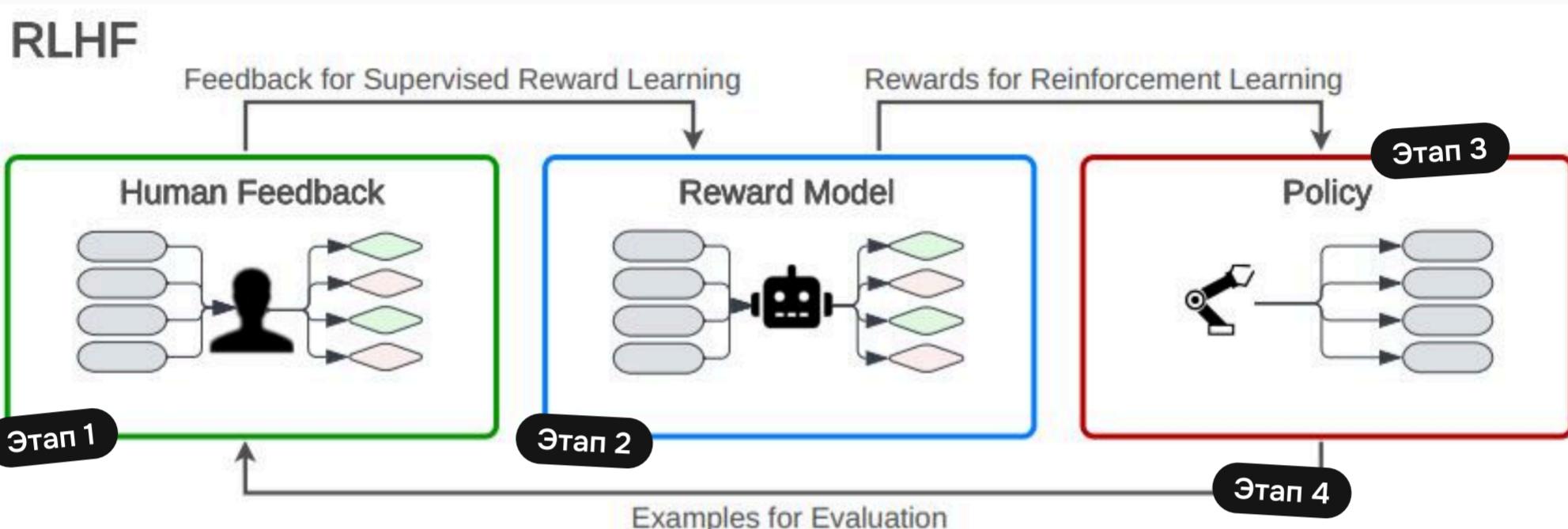
В случае LLM среда – это любой внешний оценщик. Чаще всего это специальная модель, имитирующая оценки человека (ее обычно называют **reward-модель**).

Например:

1. Агент (LLM) генерирует ответ на вопрос человека про столицу Франции и отвечает «Марсель».
2. Среда (человек или reward-модель) оценивает ответ как неправильный и выдаёт отрицательную награду.
3. Агент корректирует свои параметры так, чтобы в будущем с большей вероятностью отвечать «Париж».

Так как в случае LLM награда в основном задается через оценки человека, подход называют RLHF (Reinforcement Learning with Human Feedback).

В отличие от классического обучения с подкреплением (RL), где модель обычно получает численные награды из заранее определённой функции вознаграждения, в RLHF модель учится генерировать ответы, опираясь на человеческие предпочтения.



Этап 1 Сбор разметки данных от людей

На практике наиболее распространён подход, при котором для разметки люди выбирают из нескольких вариантов ответов модели на один и тот же вопрос (это может быть парное сравнение или ранжирование). Можно также собирать численные или бинарные оценки (нравится / не нравится / нравится на 4 из 10).

Этап 2 Обучение Reward модели

Размеченные человеком данные используются для обучения модели вознаграждений. Она учится воспроизводить человеческие оценки, то есть автоматически предсказывать, какие ответы люди предпочитают больше других.

Этап 3 Обучение политики

Теперь уже модель вознаграждений используется для дальнейшего обучения основной языковой модели. LLM генерирует ответы и получает от reward модели численные награды, на основании которых постепенно улучшает свои ответы.

Этап 4 Цикл обратной связи

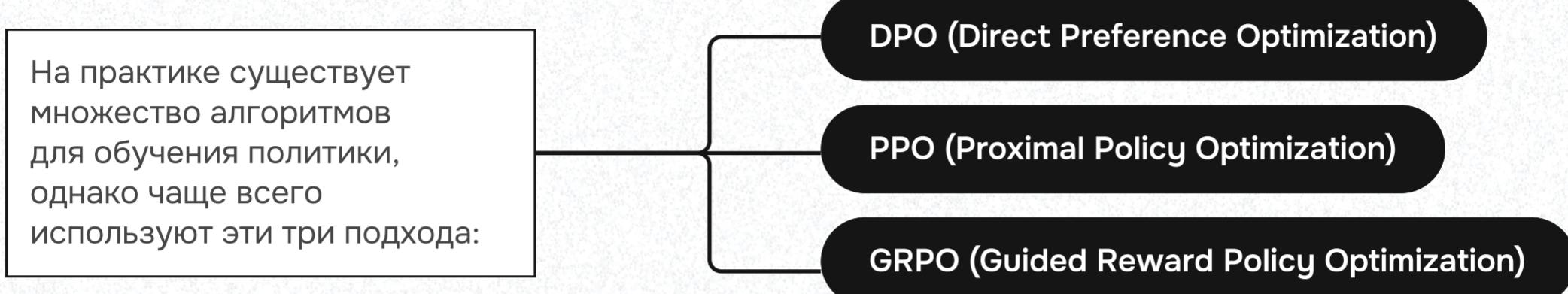
Лучшие ответы модели могут снова направляться к человеку для дополнительной оценки и валидации. Так можно постепенно собирать больше данных и улучшать reward модель.

RLHF используется для того, чтобы языковая модель лучше соответствовала ожиданиям и предпочтениям человека. Такой процесс называют элайментом (alignment, то есть «выравнивание» поведения модели с человеческими ценностями и пожеланиями).

Именно благодаря RLHF языковые модели получаются такими “человечными” и умеют, например, вести вежливый диалог или “проявлять” эмпатию. Все потому что модель обучается не просто предсказывать текст, а учитывает оценки и предпочтения людей.

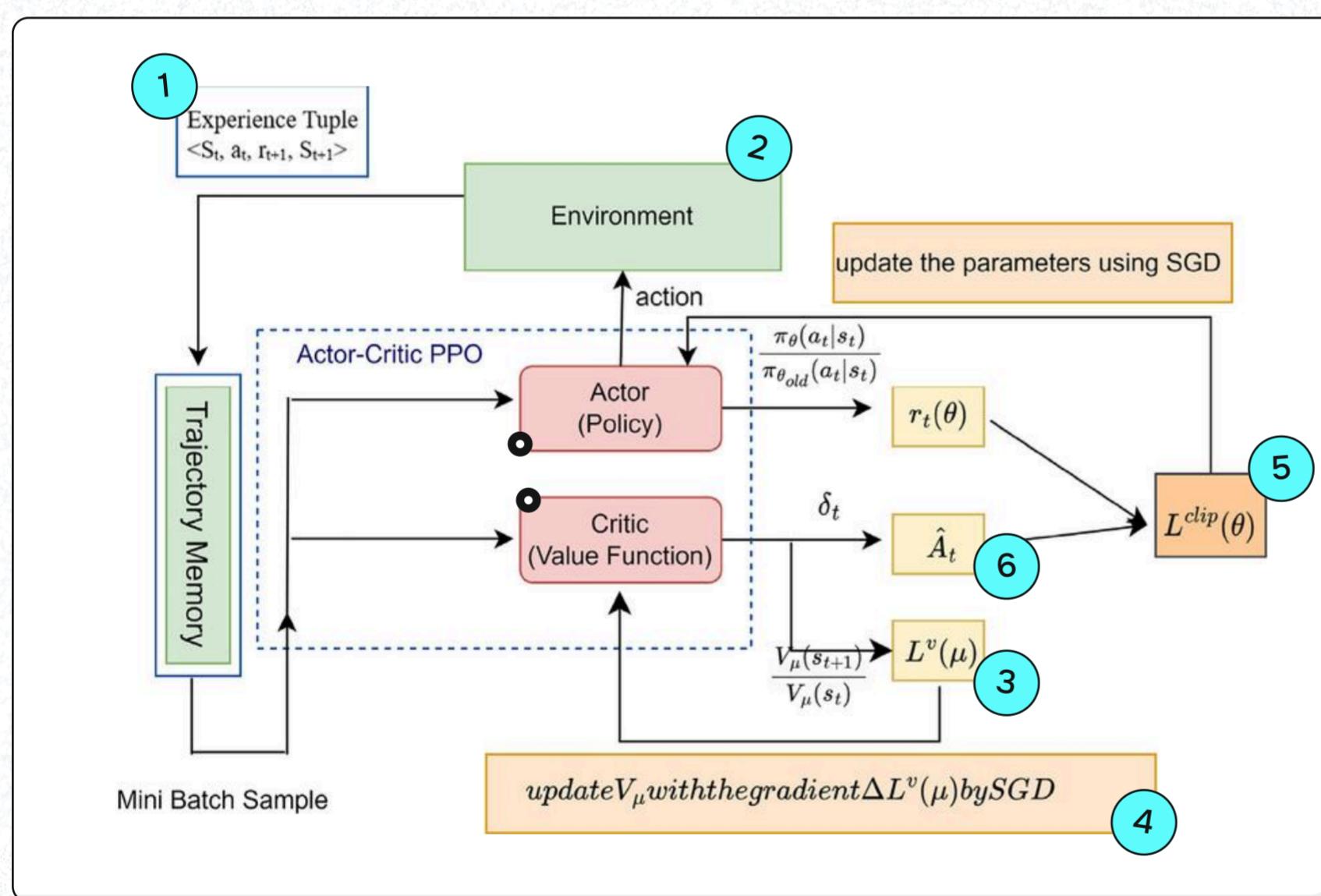
Также RLHF незаменим, когда мы говорим про безопасность и адекватность модели. Он помогает минимизировать нежелательные или вредные результаты генераций.

Мы обговорили общую схему RLHF, но еще не обсудили, как именно мы обучаем политику на третьем (основном) шаге.



PPO (Proximal Policy Optimization)

Основной и наиболее широко используемый в RLHF алгоритм обучения с подкреплением, **представленный** командой OpenAI в 2017 году (сразу после того, как они же написали оригинальную работу про RLHF). Алгоритм быстро стал популярным благодаря своей простоте и надежности и используется по сей день.



- 1 Все пройденные «траектории», то есть накопленный опыт, хранятся в формате списков (состояние → действие → награда → следующее состояние).
- 2 Среда принимает от актора действия, а обратно возвращает награду r_{t+1} и следующее состояние s_{t+1}
- 3 Функция потерь для обновления параметров модели-критика.
- 4 Обновляем параметры моделей (и актора, и критика) с помощью обычного градиентного спуска.
- 5 Функция потерь для обновления главной модели-актора. Обратите внимание, что PPO вводит коэффициент «clip», который не позволяет новой политике π_θ слишком сильно отличаться от старой политики $\pi_{\theta_{old}}$.
- 6 Это так называемое **преимущество** (может быть и положительным, и отрицательным). Его мы вычисляем, сравнивая фактическую награду с ожидаемой ценностью состояния от Critic.

- 3 Функция потерь для обновления параметров модели-критика.
- 4 Обновляем параметры моделей (и актора, и критика) с помощью обычного градиентного спуска.
- 5 Функция потерь для обновления главной модели-актора. Обратите внимание, что PPO вводит коэффициент «clip», который не позволяет новой политике π_θ слишком сильно отличаться от старой политики $\pi_{\theta_{old}}$.
- 6 Это так называемое **преимущество** (может быть и положительным, и отрицательным). Его мы вычисляем, сравнивая фактическую награду с ожидаемой ценностью состояния от Critic.

PPO работает в формате Actor-Critic.

- Actor – это наша модель, то есть текущая политика, которая совершает действие a_t , исходя из текущего состояния s_t (то есть входных данных).
- Critic (или value model) – модель, которая оценивает долгосрочную ожидаемую выгоду от политики в текущем состоянии. Это позволяет определить, насколько фактическая награда за действие актора оказалась лучше или хуже ожиданий, и обновлять политику, исходя из этой информации.

Вы можете задаться вполне логичным вопросом «Зачем нам вообще модель-критик? Разве нельзя оценивать политику просто по награде?». Нет, в контексте LLM, к сожалению нельзя.

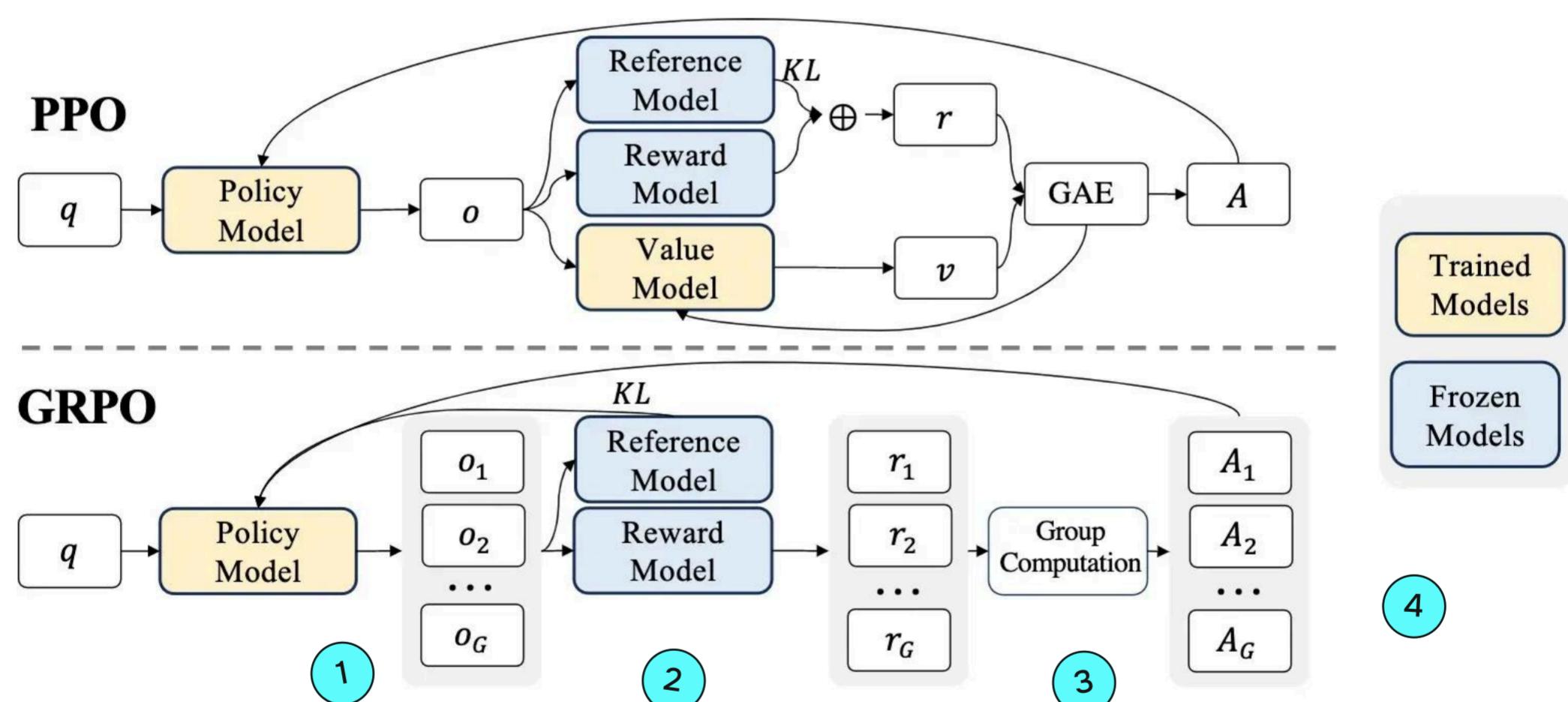
Представим, что мы получили награду, и она равна 5. Это хорошо или плохо? Мы не знаем. 5 баллов может быть отличной наградой, если обычно в данном состоянии мы получаем только 1–2 балла. Или это может быть очень плохим результатом, если обычно мы получали в этом состоянии по 10–15 баллов.

Критик как раз задаёт этalon – он говорит, чего стоит ожидать в каждом конкретном состоянии. После этого мы можем сказать, что 5 баллов – это лучше или хуже ожидаемого, то есть хорошо, или плохо. Только таким образом можно эффективно обучать актор.

GRPO (Guided Reward Policy Optimization)

Однако у PPO есть и свои минусы. В частности, та самая модель-критик на самом деле очень требовательна с точки зрения ресурсов, поскольку ее тоже нужно обучать. Это сильно тормозит весь процесс и делает обучение дорогим.

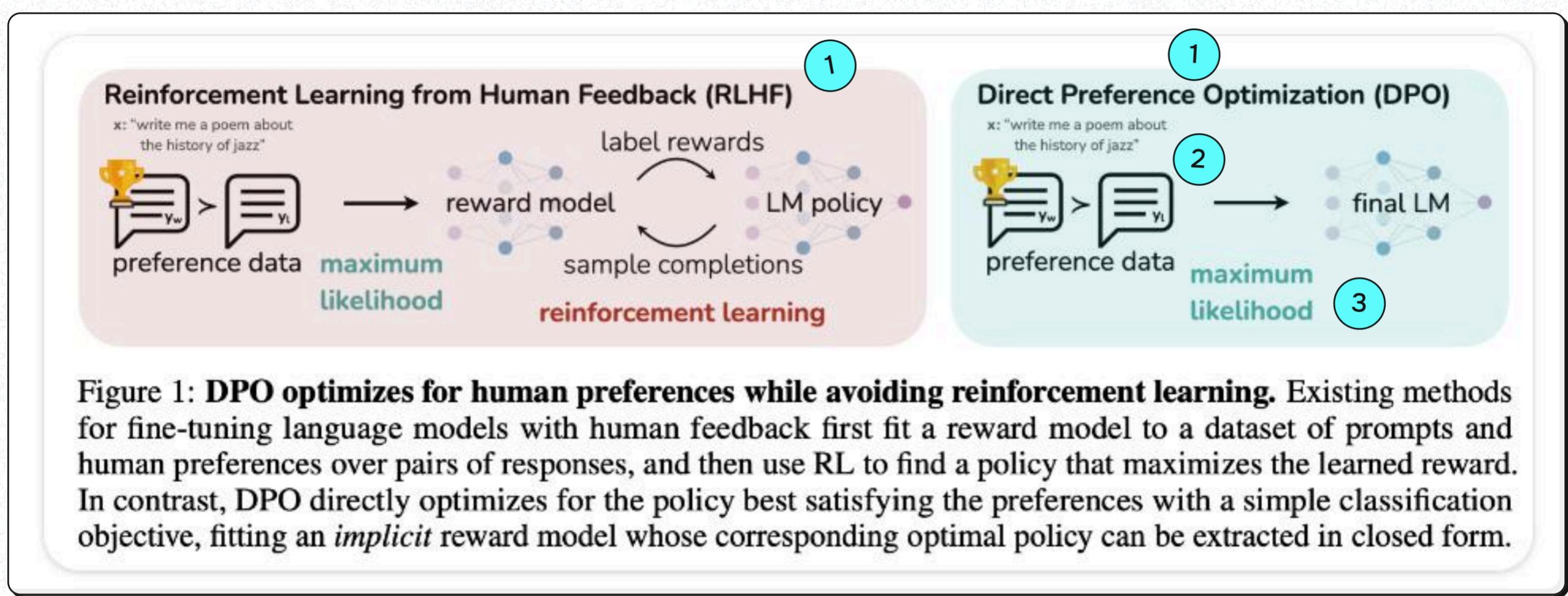
Как обойти этот нюанс несколько лет назад придумали исследователи из DeepSeek. Они предложили алгоритм GRPO – **более эффективную вариацию PPO**, которая быстро набрала популярность.



- 1 Модель генерирует уже не один ответ, а сразу несколько.
- 2 В GRPO из алгоритма вообще **удалена Value Model**, то есть модель-критик. Вместо этого, чтобы правильно оценивать награды, мы используем среднюю награду от группы ответов на один и тот же вопрос, и так определяем, насколько "хороши" действия модели.
- 3 Каждый ответ из группы получает свою награду. За счёт этого вместо того, чтобы сравнивать награду определённого действия с ожиданиями модели-критика, мы можем сравнить ту же награду со средней наградой по группе.
- 4 Политика обновляется на основе относительных преимуществ внутри группы.

DPO (Direct Preference Optimization)

Ещё один сравнительно новый алгоритм, предложенный в 2023 году как более простой и эффективный подход к RLHF. Его ключевое отличие от PPO и GRPO заключается в том, что DPO вообще не требует отдельной reward-модели и явного этапа обучения с подкреплением: он напрямую оптимизирует политику, используя пары ответов, ранжированных людьми по степени предпочтения.



- 1 То есть, в отличие от классических подходов к RLHF, в DPO мы полностью избегаем обучения модели вознаграждений и этапа обучения с подкреплением как такового. Вместо этого DPO напрямую оптимизирует модель (политику) по небольшому множеству данных о человеческих предпочтениях, заданных через пары ответов.
- 2 Для каждого запроса из датасета у нас есть два варианта ответа модели, один из которых люди посчитали предпочтительнее другого.
- 3 Эти пары используются напрямую для оптимизации модели. Целью становится максимизация вероятности предпочтительных ответов и минимизация вероятности менее предпочтительных. Другими словами, модель напрямую учится выбирать ответы, наиболее соответствующие человеческим предпочтениям, без необходимости явно строить отдельную модель вознаграждений или вычислять преимущества через критика.

Итак, мы разобрались с тем, как работает RL, RLHF и три основных алгоритма обучения политики.

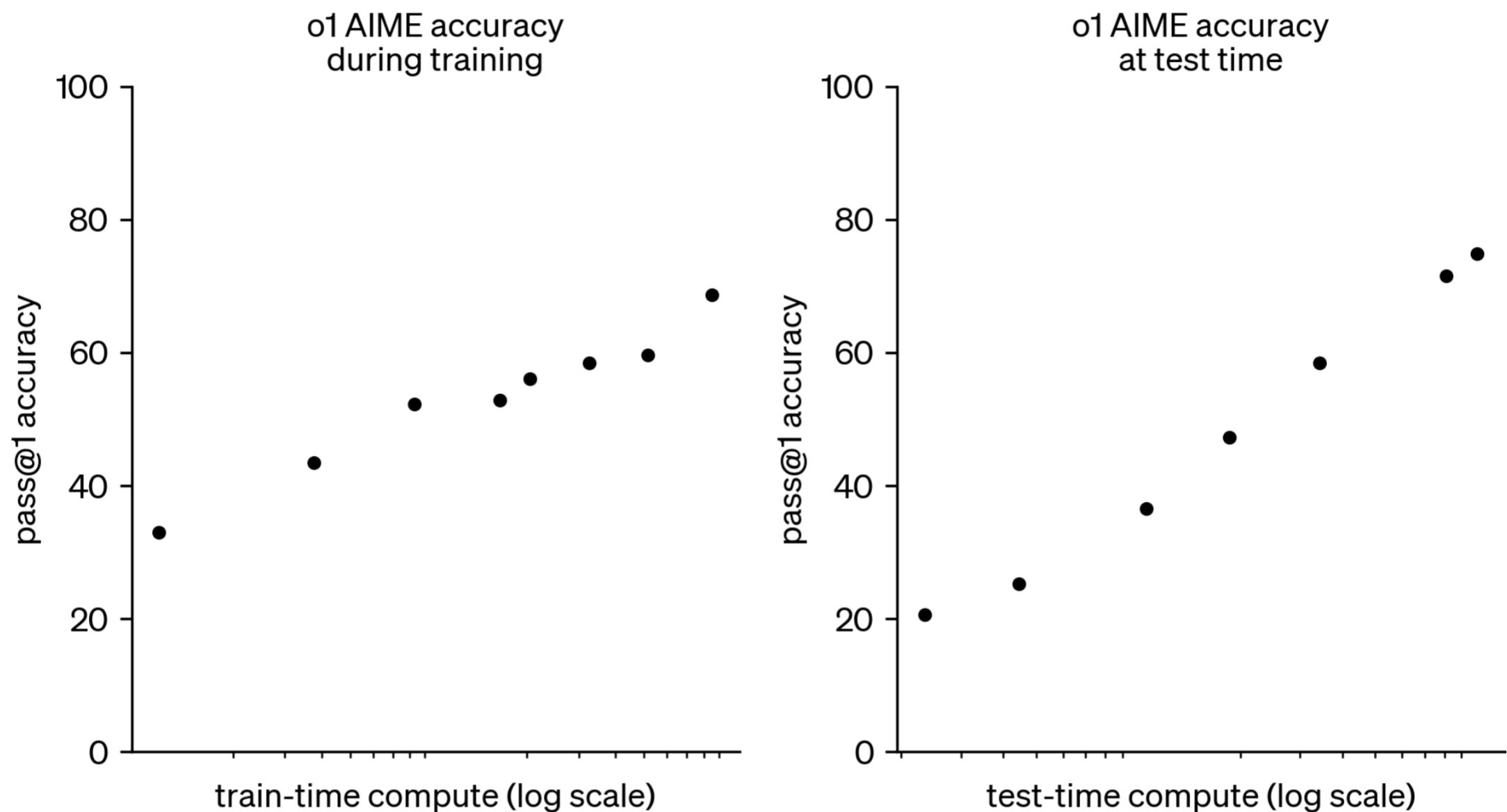
Однако сегодня RL используется не только для улучшения качества ответов и соответствия человеческим предпочтениям. Все, что вы увидели сверху, сейчас имеет неоценимое значение для индустрии в первую очередь из-за ризонинга.



Самое интересное в ризонинге то, что это новая парадигма масштабирования LLM.

Некоторое время назад считалось, что LLM могут становиться умнее только за счет увеличения количества обучающих данных и ресурсов на обучения. Однако некоторое время назад, когда появились первые большие ризонинг-модели (o1 или R1), обнаружилось, что ризонинг тоже способен масштабировать модели.

То есть чем дольше модель рассуждает и чем больше токенов вкладывает в свои цепочки мыслей, тем качественнее получаются ответы. Так как это масштабирование происходит только за счет и во время инференса, его называют *test-time scaling*.



Эти графики демонстрируют *test-time scaling* для модели o1 от OpenAI. Тут явно показано, что чем больше вычислений модель тратит на рассуждения во время инференса, тем выше метрики итоговых ответов.

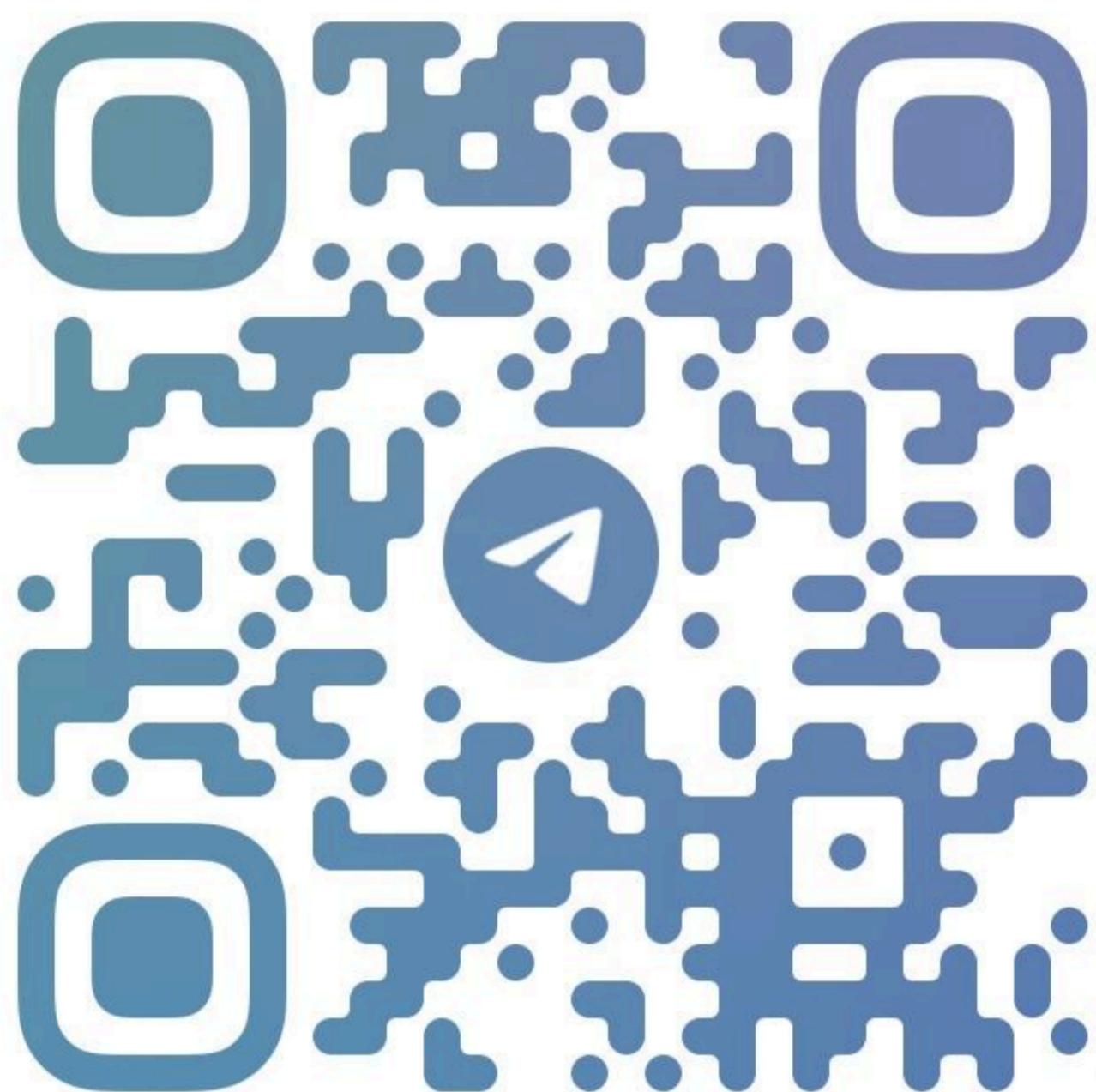
Многие (в том числе некоторые крупные ученые и лидеры индустрии) считают, что именно ризонинг может привести нас к AGI. Так это или нет – проверим через несколько лет.

Заключение

На этом наш конспект подошёл к концу. Мы прошли большой путь – от математических основ и архитектуры трансформеров до тонкостей файн-тюнинга, RL и рассуждений. Надеемся, наши наглядные объяснения и схемы помогли вам разобраться в том, **как работают современные LLM изнутри**.

Конечно, мир больших языковых моделей постоянно меняется и развивается: появляются новые подходы, алгоритмы и задачи, а значит, и мы будем регулярно дополнять конспект новыми темами и углублёнными разделами.

Оставайтесь на связи и, конечно, подписывайтесь на наш Телеграм-канал Data Secrets!



@DATA_SECRETS