ТРАНСФОРМЕР

Сергей Николенко

СПбГУ — Санкт-Петербург

23 октября 2026 г.





Random facts:

- 23 октября 4004 г. до н.э., по расчётам архиепископа Джеймса Ашшера (1650), был сотворён мир; начало творения он отнёс к вечеру накануне этого дня
- 23 октября 1545 г. запорожские казаки, выйдя в море на 32 челнах-чайках, подошли к турецкой крепости Ачи-Кале (Очаков) и захватили её
- 23 октября 1707 г. впервые собрался парламент Великобритании, объединённого королевства Англии и Шотландии
- 23 октября 1958 г. в бельгийском журнале «Спиру» впервые появились смурфы — синие человечки ростом «в три яблока»
- 23 октября 2001 г. Apple представила iPod: «тысяча песен у вас в кармане»
- 23 октября — День моля: его отмечают с 6:02 утра до 6:02 вечера в честь числа Авогадро $6{,}02 \cdot 10^{23}$

Краткое резюме предыдущей серии

- В прошлый раз мы научили рекуррентные сети *смотреть назад*: зрительное внимание, машинный перевод с вниманием.
- А потом выкинули рекуррентность и посмотрели на *самовнимание*: запросы, ключи и значения, многоголовое внимание.
- Сегодня соберём из этого *трансформер*: энкодер и декодер, маскированное внимание, токенизация, позиционные кодировки.
- А потом посмотрим, каким трансформер стал сегодня: *современный рецепт*, к которому сошлись почти все большие модели.

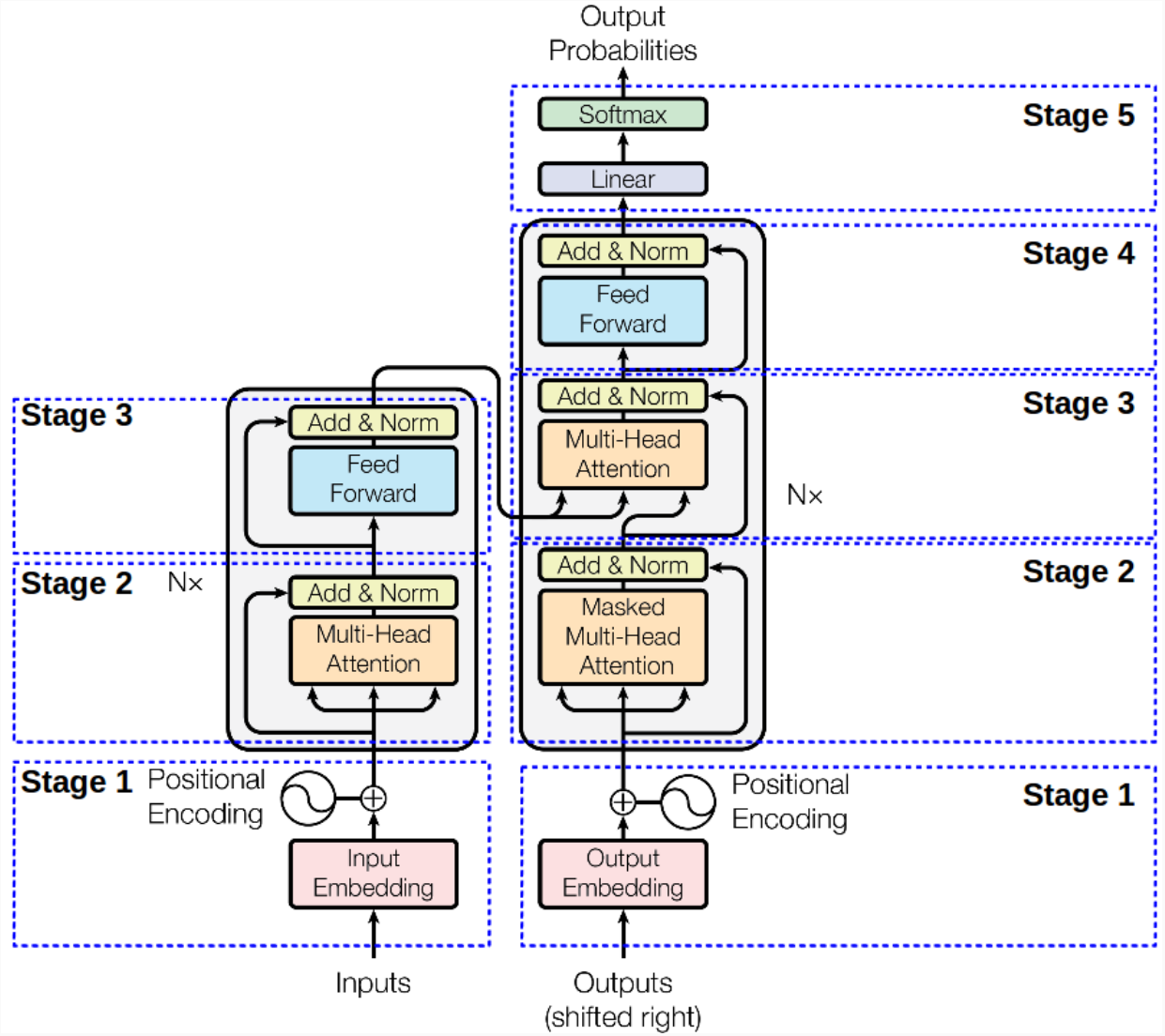
Самовнимание и Transformer

Transformer

- Мы изучали перевод на рекуррентных сетях и дошли до архитектуры Google NMT
- Но в 2017 году оказалось, что всё может быть ещё проще и интереснее
- Google: «Attention is all you need» (Vaswani et al., 2017)
- Основная идея – self-attention; оказывается очень плодотворной для всевозможных seq2seq задач
- Главная мотивация – попробовать всё-таки уйти от кодирования вектором постоянной длины

Transformer

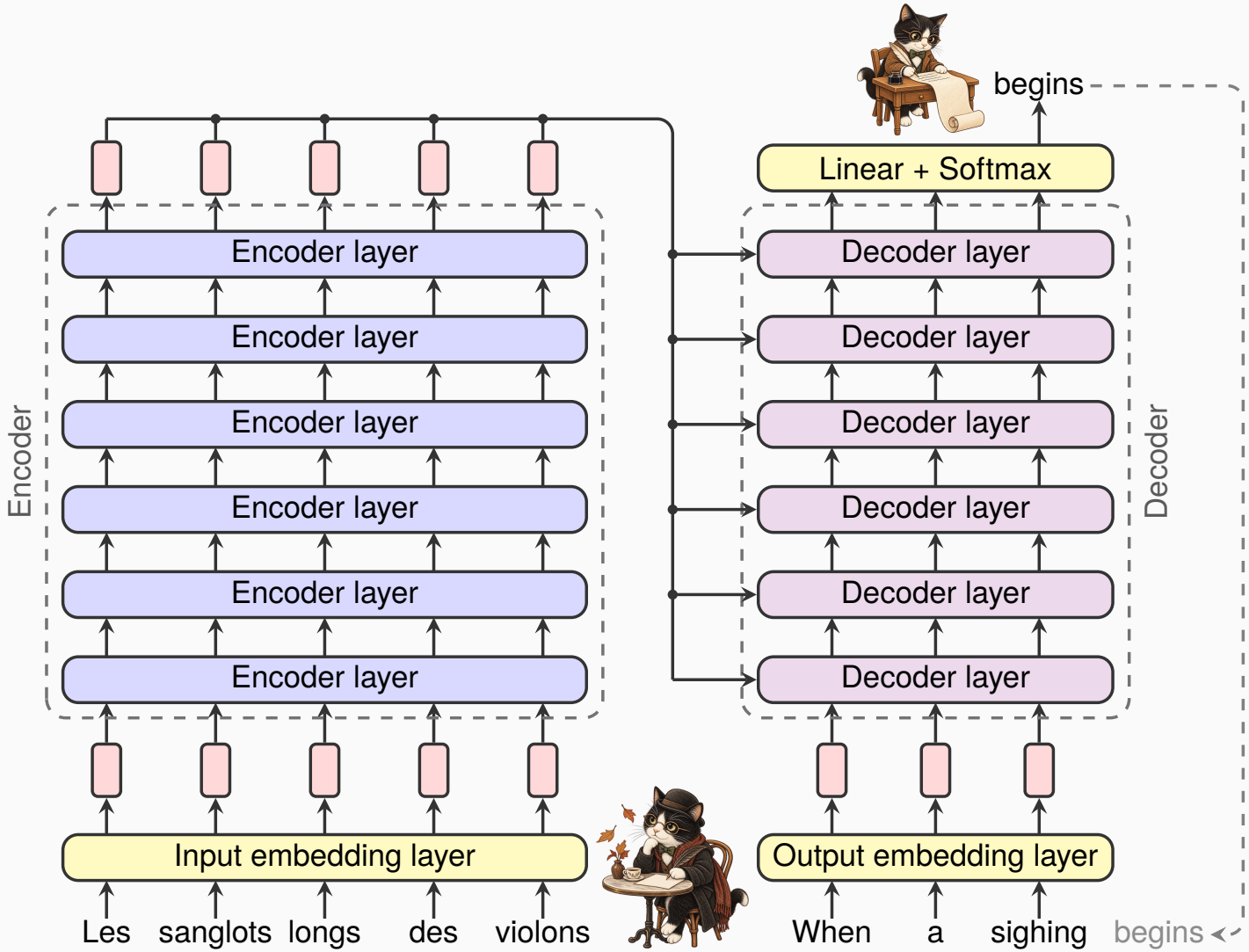
- Общая схема:



- Теперь подробнее...

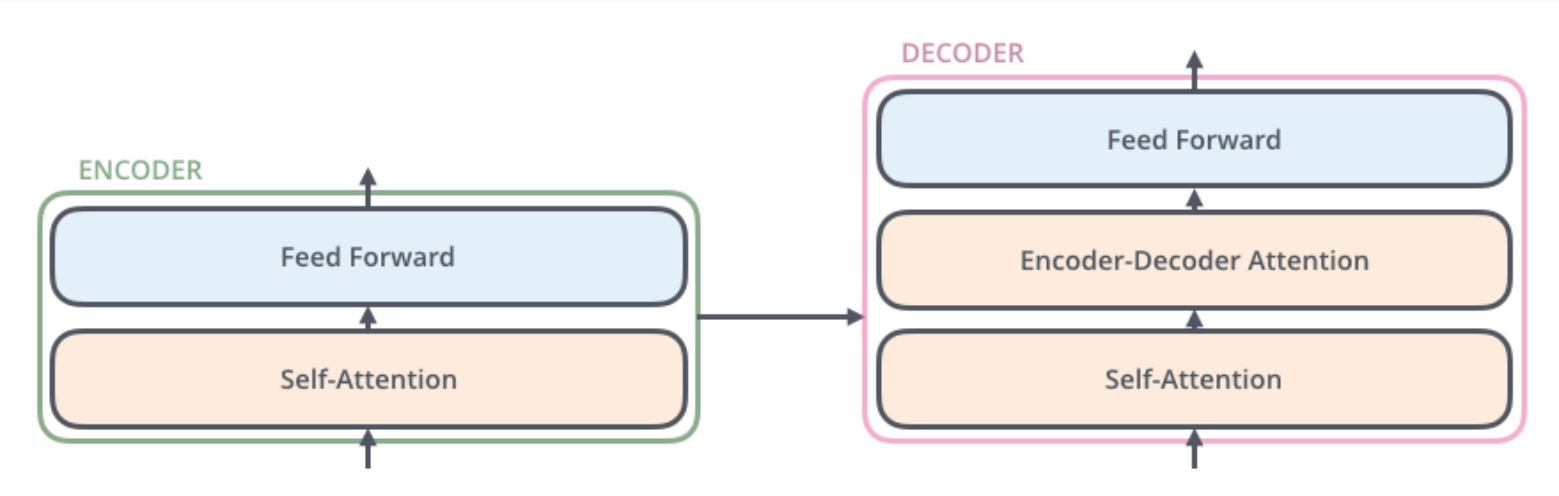
Transformer

- Суть, как и раньше, – encoder-decoder:



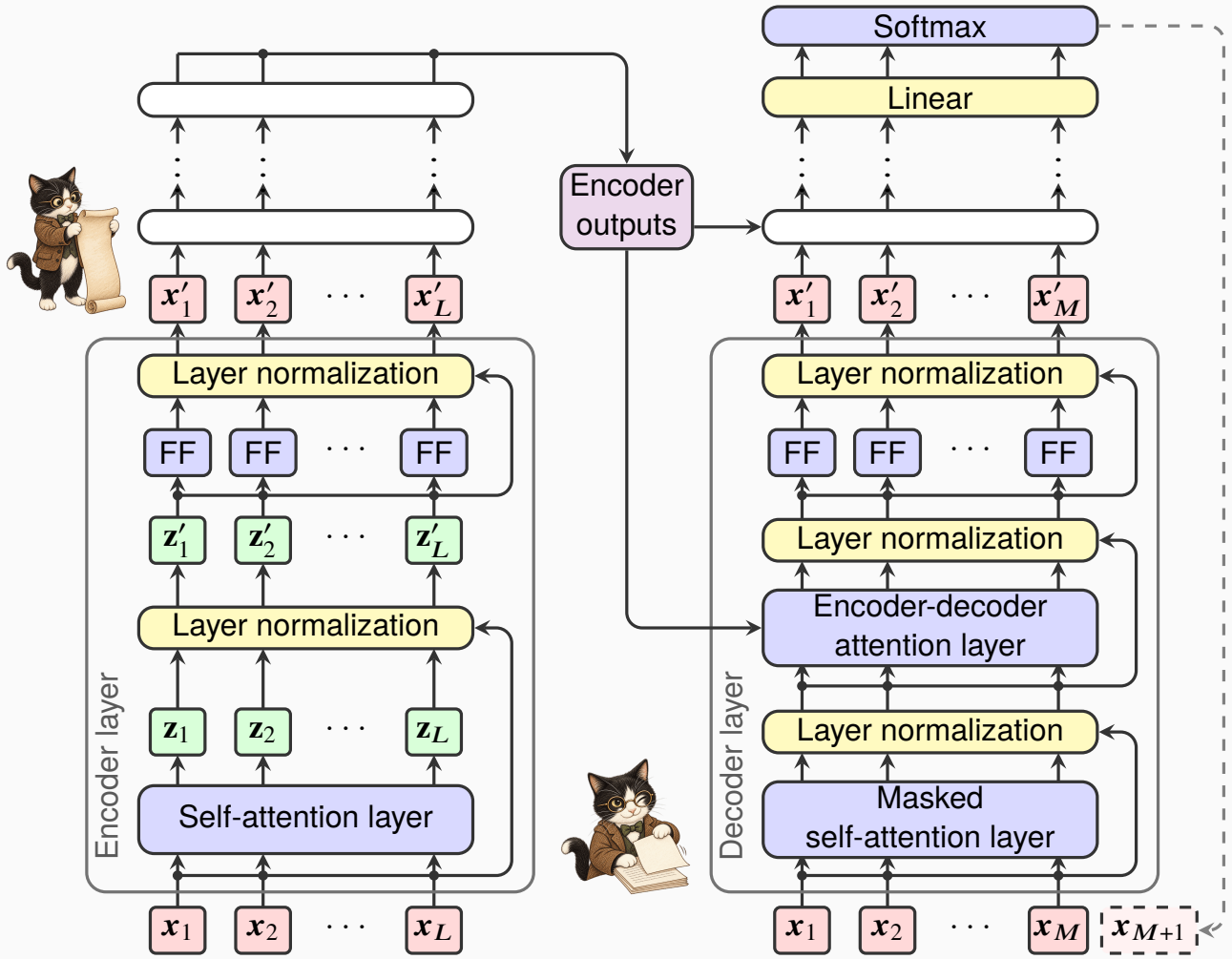
Transformer

- В каждом слое – слой self-attention, а потом feedforward layer, который независимо применяется к каждой позиции входа. У декодера ещё есть attention между ними:



Transformer

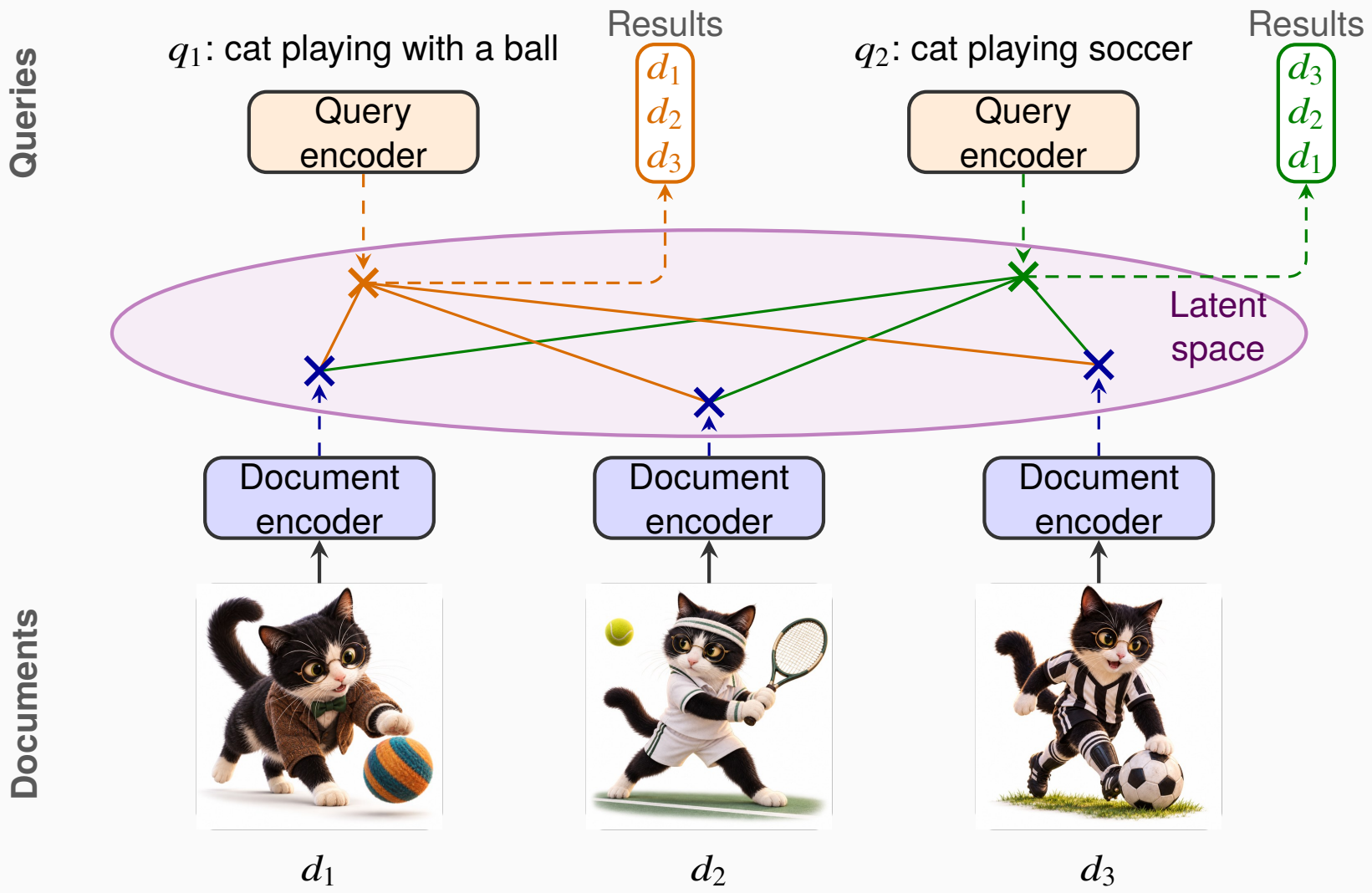
- Слова, естественно, представляются векторами, в feedforward слое всё параллельно:



- Но что же это такое – self-attention?

Self-attention

- Чтобы понять самовнимание (self-attention), начнём с информационного поиска (information retrieval)



Self-attention

- Запросы и документы отображаются в одно и то же латентное пространство (но разными кодировщиками — это могут быть вообще объекты разной природы)
- Запрос идёт через query encoder
- Документы (на иллюстрации это картинки) — через другой encoder, но в то же пространство
- Чтобы найти самые релевантные документы, мы ищем ближайших соседей в латентном пространстве среди документов; часто предполагают, что латентное пространство линейно, и расстояние там — это просто скалярное произведение $\mathrm{dist}(q, d) = \mathrm{Enc}_q(q)^\top \mathrm{Enc}_d(d)$.

Self-attention

- В самовнимании эта интуиция используется очень абстрактно. Слой получает на вход последовательность векторов $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_L$, т.е. матрицу $X \in \mathbb{R}^{d \times L}$.
- Запросы, ключи и документы берутся из самих $\mathbf{x}_i$:
  - умножая $\mathbf{x}_1$ на матрицу весов W^Q, мы получаем вектор *запроса* $\mathbf{q}_1 = W^Q \mathbf{x}_1$; обратите внимание, что размерность q векторов запросов, $\mathbf{q}_i \in \mathbb{R}^q$, может отличаться (обычно меньше) от входной размерности d, $\mathbf{x}_i \in \mathbb{R}^d$;
  - умножая $\mathbf{x}_i$ на матрицу весов W^K, мы получаем векторы *ключей* $\mathbf{k}_i = W^K \mathbf{x}_i$ для $i = 1, \dots, L$; поскольку мы хотим, чтобы запросы и ключи находились в одном латентном пространстве, ключи имеют ту же размерность, что и запросы, $\mathbf{k}_i \in \mathbb{R}^q$, таким образом $W^K \in \mathbb{R}^{q \times d}$;
  - третья матрица весов W^V даёт векторы *значений* $\mathbf{v}_i = W^V \mathbf{x}_i$ для $i = 1, \dots, L$; это те *документы*, которые мы будем «извлекать» с помощью ключей $\mathbf{k}_i$; формально мы имеем другую размерность v для значений, $\mathbf{v}_i \in \mathbb{R}^v$ и $W^V \in \mathbb{R}^{v \times d}$; на практике обычно $v = q$.

Self-attention

- Затем мы выполняем поиск, вычисляя оценки внимания как скалярные произведения между запросами и документами
- Нормализуем $\mathbf{q}_i^\top \mathbf{k}_j$, деля на $\sqrt{q}$, и получаем оценки через softmax, чтобы преобразовать их в вероятности:

$$\alpha_{ij} = \mathrm{softmax}\left(\frac{1}{\sqrt{q}}\mathbf{q}_i K^\top\right)_j,$$

где $K \in \mathbb{R}^{q \times L}$ — это все ключи, объединенные в матрицу, $K = W^K X$.

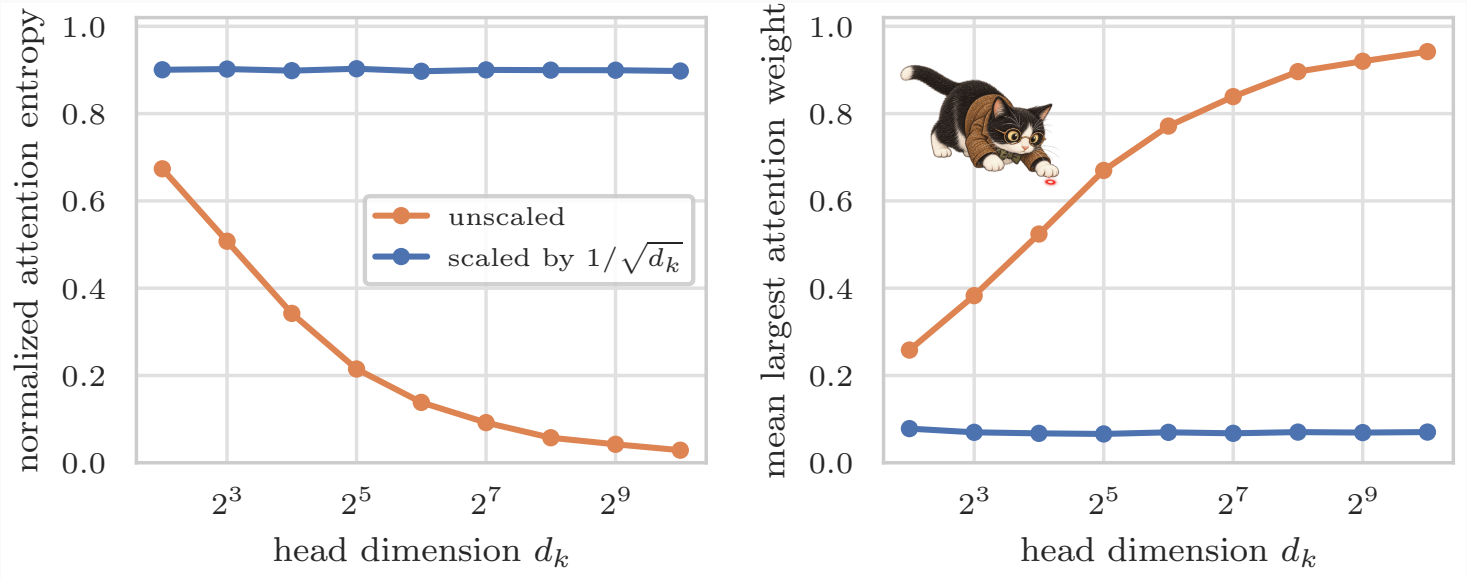
- Затем используем результат в качестве коэффициентов для выпуклой комбинации *значений* $\mathbf{v}_j$:

$$\mathbf{z}_i = \mathrm{softmax}\left(\frac{1}{\sqrt{q}}\mathbf{q}_i K^\top\right) V,$$

где $V \in \mathbb{R}^{q \times L}$ — значения (values), $V = W^V X$.

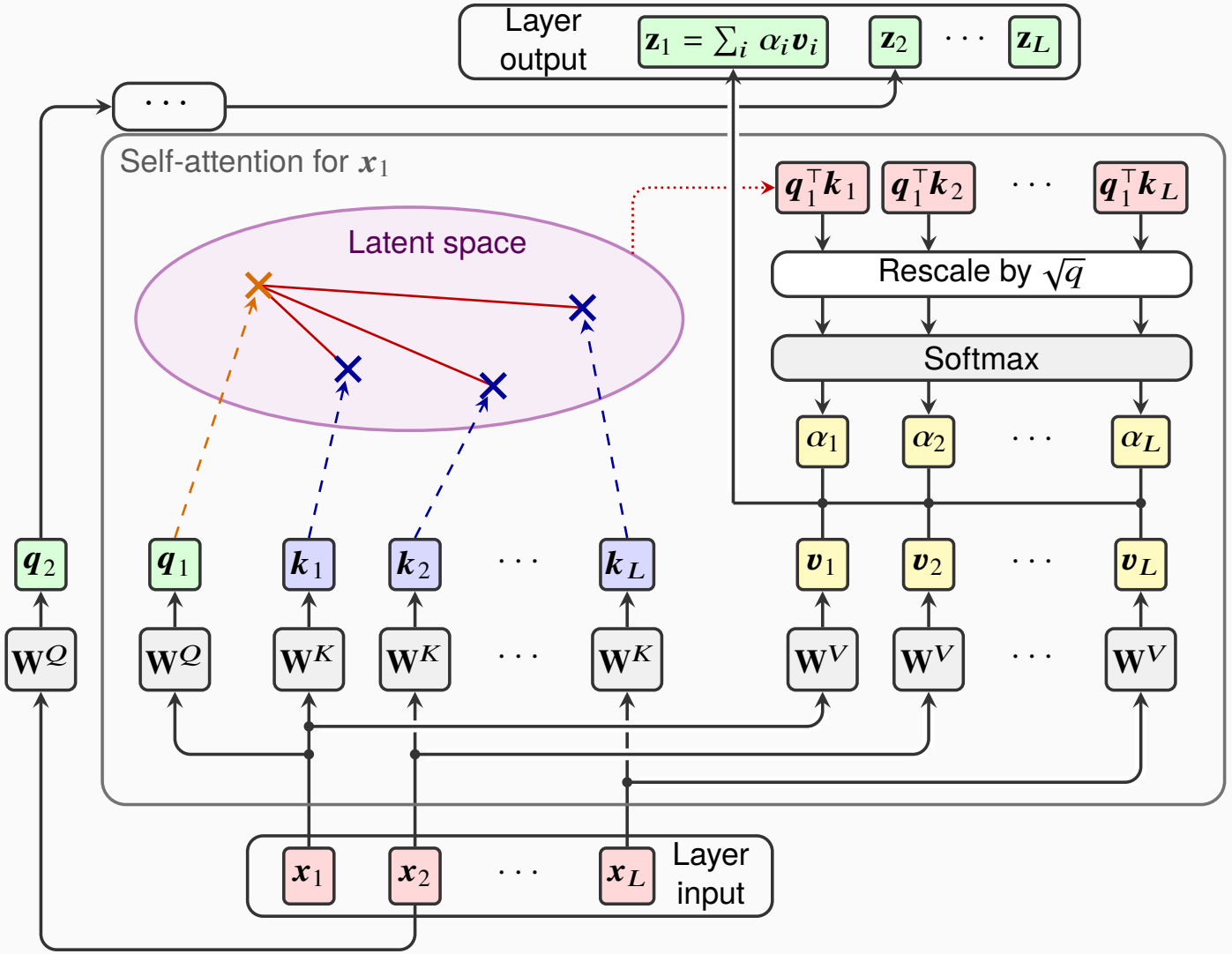
Self-attention

- Зачем делить на $\sqrt{q}$? Скалярное произведение $\mathbf{q}_i^\top \mathbf{k}_j = \sum_{r=1}^{q} q_{ir} k_{jr}$ — сумма q примерно независимых слагаемых с дисперсией около 1, так что оно порядка $\sqrt{q}$.
- Без нормировки softmax насыщается, и внимание превращается в почти детерминированный поиск; слева — нормированная энтропия весов, справа — самый большой вес:



Self-attention

- Общая структура слоя самовнимания (self-attention):



Self-attention

- И это вся интуиция! Мы можем объединить вычисления каждого $\mathbf{z}_i$ в одну формулу в матричном виде:

$$Z = \mathrm{softmax}\left(\frac{1}{\sqrt{q}} Q K^\top\right) V.$$

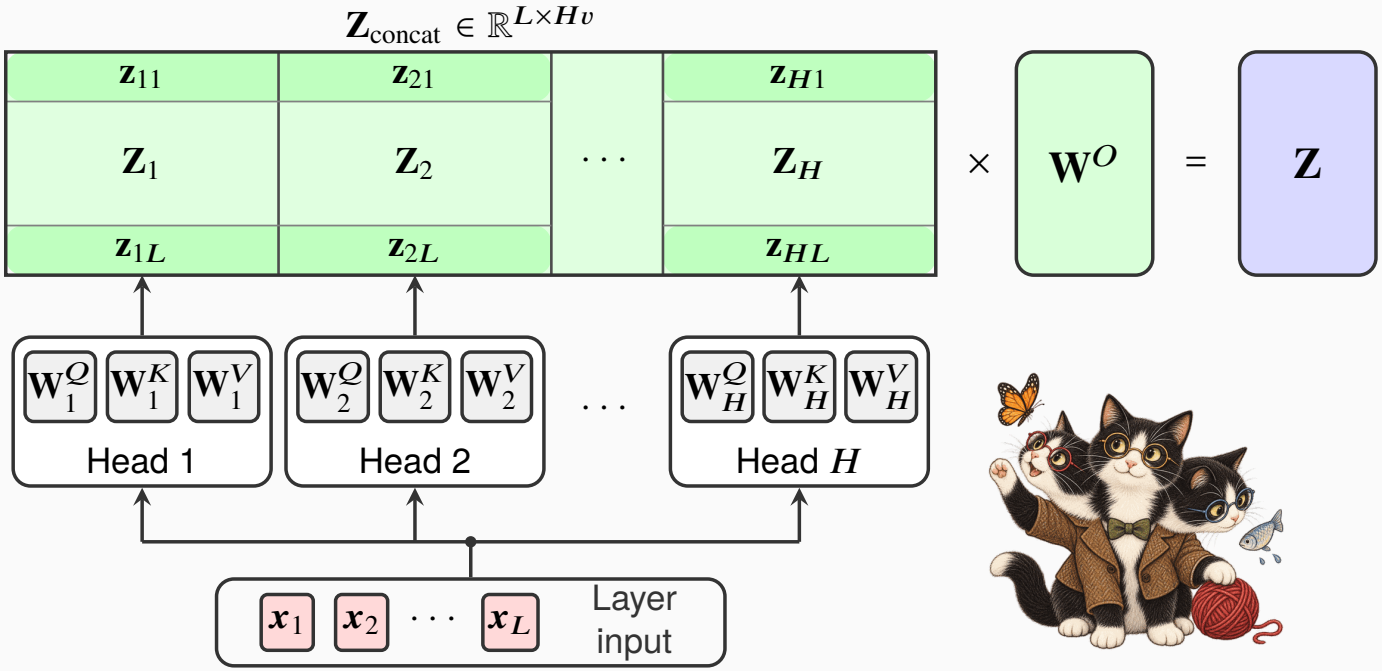
- Но это только один способ «смотреть» на входные векторы; то, что мы сейчас определили, — это не полный слой самовнимания, а только одна *голова* (self-attention head)
- Можно распараллелить эти вычисления вдоль H различных голов, используя разные матрицы весов $W_1^Q, \dots W_H^Q$, $W_1^K, \dots W_H^K$ и $W_1^V, \dots W_H^V$. Это *многоголовое внимание* (multi-head attention)

Self-attention

- После H параллельных голов мы получаем матрицы выходов $Z_1, Z_2, \ldots, Z_H$ размерности $v \times L$, объединяем их все в $Z_{\text{concat}} \in \mathbb{R}^{L \times Hv}$ и добавляем ещё одну матрицу весов $W^O \in \mathbb{R}^{Hv \times d}$, чтобы сжать результат обратно до необходимой размерности:
$$Z = \left(Z_{\text{concat}} W^O\right)^\top.$$
- С помощью W^O слой самовнимания может комбинировать представления, полученные из различных голов внимания; это также важный способ добавить гибкость и выразительность в архитектуру.

Self-attention

- Многоголовое внимание:



- Это самая важная часть, но это ещё не всё.

Два вида смешивания: токены и каналы

- Блок трансформера удобно понимать как чередование двух операций:
  - *внимание смешивает токены*: каждая позиция собирает информацию из других;
  - *FFN смешивает каналы*: каждая позиция обрабатывается отдельно, $d \to 4d \to d$.
- С нормализацией N и остаточными связями:

$$A = X + \mathrm{MHA}(N(X)), \qquad Y = A + \mathrm{FFN}(N(A)).$$

- Одного внимания мало: без остаточных связей и FFN стопка слоёв самовнимания вырождается в матрицу ранга 1, причём дважды экспоненциально быстро (Dong et al., 2021).

Два вида смешивания: токены и каналы

- Где параметры? При $d_{\text{ff}} = 4d$ у FFN около $8d^2$ весов, у W^Q, W^K, W^V, W^O — около $4d^2$; большая часть модели — в FFN.
- Где вычисления? Проекции — $4Ld^2$, $QK^\top$ и умножение на V — $2L^2d$, FFN — $8Ld^2$; квадратичная часть доминирует, только когда L в несколько раз больше d.
- А вот с памятью хуже: наивная реализация хранит HL^2 весов внимания (к этому мы ещё вернёмся).
- Удобный взгляд — *residual stream*: каждый слой читает из общего потока и пишет в него. Голова внимания — маленькая ассоциативная память: запросы и ключи решают, *откуда* читать, значения — *что* скопировать.

Декодер

- Декодер включает два дополнительных типа слоёв: *маскированное самовнимание* (masked self-attention) и *encoder-decoder attention*
- Маскированное внимание в основном означает, что, поскольку декодер работает авторегрессионно, он не должен смотреть на токены, которые ещё не породил
- Для этого проще всего ввести всю последовательность и замаскировать будущие позиции внутри слоёв самовнимания
- Формально это означает, что мы задаем аргументы softmax как $-\infty$ для будущих токенов, что означает, что их веса внимания всегда будут нулевыми

Декодер

- Encoder-decoder attention — это вариация механизма самовнимания, которая опирается на результаты кодировщика
- Звучит сложно... но на самом деле это почти тривиальное изменение! Мы используем разные векторы в качестве входных данных в том же самовнимании:
  - для создания запросов используем векторы из предыдущего слоя, то есть текущие представления уже порождённых выходных токенов;
  - но для «документов» в задаче «поиска», то есть для векторов ключей и значений, мы используем векторы из энкодера
- Неформально это значит, что мы выполняем «поиск» на выходе энкодера с запросами, состоящими из уже порождённых токенов
- Формально: запросов T (уже порождённые токены), ключей и значений S (токены источника); $QK^\top$ — матрица $T \times S$, каждый целевой токен распределяет внимание по всем S токенам источника, и выходов ровно T, по одному на

Декодер

- Обучается декодер параллельно, хотя модель авторегрессионная: подаём сдвинутую последовательность $\langle \mathrm{BOS} \rangle, y_1, \ldots, y_{T-1}$ и предсказываем $y_1, \ldots, y_T$ сразу во всех позициях (teacher forcing).
- Маскированное внимание — одна треугольная матрица:

$$\mathrm{Attn}_{\mathrm{causal}}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{q}} + M\right) V,$$

$$M_{ij} = 0 \text{ при } j \le i, \qquad M_{ij} = -\infty \text{ при } j > i.$$

- А порождение — честно по одному токену; ключи и значения префикса хранятся в *KV-кэше*, чтобы не пересчитывать их на каждом шаге.
- Тонкость: при обучении префикс написан человеком, а при порождении — самой моделью, и ранняя ошибка остаётся в контексте (exposure bias).

Токенизация

- Мы используем вложения (embeddings), но как мы делим текст на токены?
- *Byte-pair encoding* (BPE): старая идея из сжатия данных (Gage, 1994), которую для перевода приспособили Sennrich et al. (2016): жадно склеиваем самую частую пару символов
- Давайте начнем с построения словаря:

$$\{\text{cat} : 10, \text{pet} : 12, \text{mat} : 5, \text{rat} : 8, \text{eats} : 4\}.$$

Original words:	cat		pet		mat		rat		eats
Frequencies:	10		12		5		8		4
Characters:	c	a	t	p	e	m	r	s	
Frequencies:	10	27	39	12	16	5	8	4	
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts	
Frequencies:	10	**27**	12	12	5	8	4	4	



Токенизация

- Затем разделим его на символы и подсчитаем пары символов:

$$\{\text{cat} : 10, \text{pet} : 12, \text{mat} : 5, \text{rat} : 8, \text{eats} : 4\},$$

$$\{c : 10, a : 27, t : 39, p : 12, e : 16, m : 5, r : 8, s : 4\},$$

$$\{ca : 10, at : 27, pe : 12, et : 12, ma : 5, ra : 8, ea : 4, ts : 4\}.$$

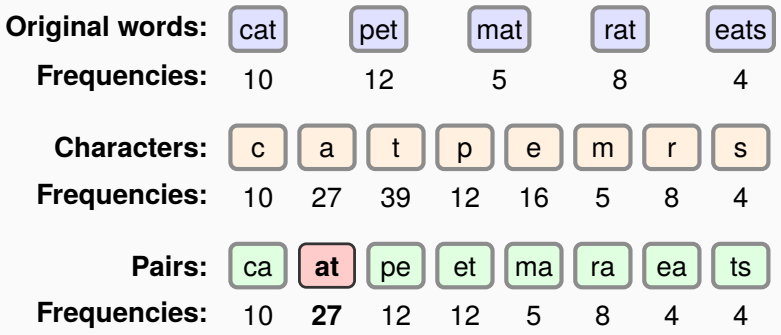
Original words:	cat		pet		mat		rat	eats
Frequencies:	10		12		5		8	4
Characters:	c	a	t	p	e	m	r	s
Frequencies:	10	27	39	12	16	5	8	4
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts
Frequencies:	10	**27**	12	12	5	8	4	4



Токенизация

- Теперь возьмем наиболее частую пару («*at*») и перекодируем её в новый символ (Z), который добавляется в словарь:

$$\{\mathrm{cZ}:10, \mathrm{pet}:12, \mathrm{mZ}:5, \mathrm{rZ}:8, \mathrm{eZs}:4\},$$
$$\{c:10, Z:27, t:12, p:12, e:16, m:5, r:8, s:4\},$$
$$\{cZ:10, pe:12, et:12, mZ:5, rZ:8, eZ:4, Zs:4\}.$$





Токенизация

- Можно теперь выбрать самую частую пару — *pe* или *et* — и заменить её другим новым символом *Y*:

$$\{\mathrm{cZ}:10,\mathrm{Yt}:12,\mathrm{mZ}:5,\mathrm{rZ}:8,\mathrm{eZs}:4\},$$
$$\{c:10,Z:27,Y:12,t:12,e:4,m:5,r:8,s:4\},$$
$$\{cZ:10,Yt:12,mZ:5,rZ:8,eZ:4,Zs:4\}.$$

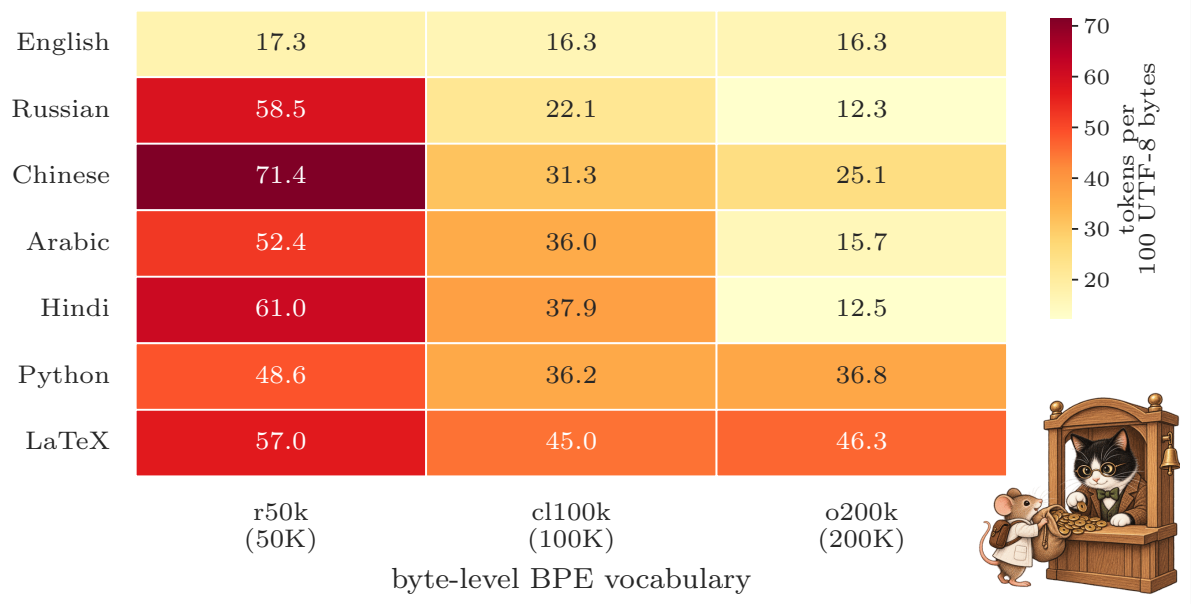
- И так далее, пока не достигнем требуемого размера словаря

Original words:	cat		pet		mat		rat	eats
Frequencies:	10		12		5		8	4
Characters:	c	a	t	p	e	m	r	s
Frequencies:	10	27	39	12	16	5	8	4
Pairs:	ca	**at**	pe	et	ma	ra	ea	ts
Frequencies:	10	**27**	12	12	5	8	4	4



Токенизация

- Варианты: byte-level BPE (GPT-2), WordPiece (BERT), Unigram, SentencePiece.
- Токенизатор — часть архитектуры: чем больше словарь, тем короче последовательности, но больше таблица вложений и softmax на выходе.
- И налог на разные языки неравномерный; токенов на 100 байт UTF-8 в трёх поколениях BPE из tiktoken:



Positional encodings

- В трансформере каждый входной токен может “посмотреть” на любой другой *напрямую*, через любое число других токенов
- Это большое преимущество по сравнению с RNN!
- Цена — стоимость и память, квадратичные по длине, так что контекстное окно (*context window*) приходится ограничивать; а главное, поскольку веса внимания покрывают все токены равномерно, мы *теряем последовательность*: слой self-attention “не знает”, что некоторые токены стоят рядом, а некоторые далеко друг от друга
- Нужно как-то дать понять трансформеру, что у последовательности есть порядок, и это делается через *positional encodings*
- Формально: $\mathrm{Attn}(PX) = P\,\mathrm{Attn}(X)$, вход для самовнимания — множество

Positional encodings

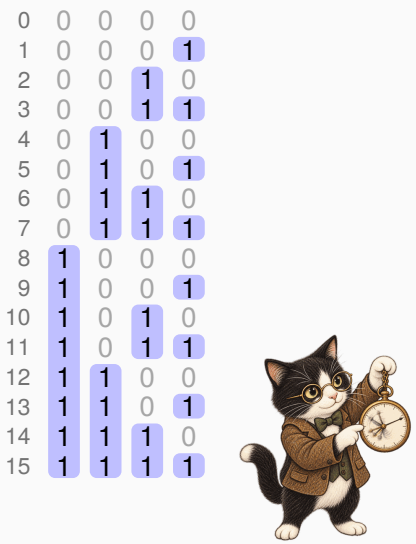
- Positional encodings — векторы, добавляемые ко вложениям, которые отражают порядковый номер токена
- Как его закодировать? Можно просто число увеличивать, но это не так просто:
  - если использовать увеличивающуюся последовательность вроде $1, 2, 3, \ldots$, она не обобщится на последовательности длиннее, чем часто бывало в датасете, да и вообще поведение будет плохо определено для больших значений;
  - если уложить в интервал вроде $[0, 1]$, то positional encoding не будет знать, сколько на самом деле слов между токенами: расстояние от 0 до $\frac{1}{2}$ может быть 1 токен или 100.

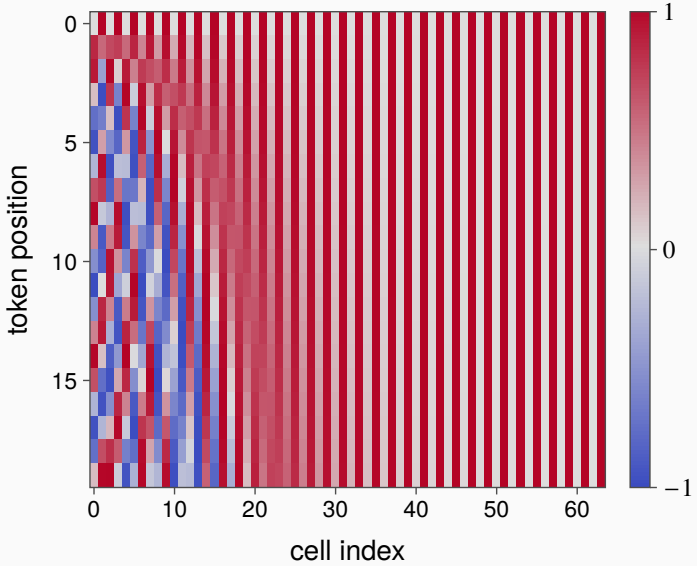
Positional encodings

- Так что мы используем нечто вроде позиционной системы счисления, только непрерывную:

$$\mathrm{PE}\,(\mathrm{pos}, 2i) = \sin\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right), \quad \mathrm{PE}\,(\mathrm{pos}, 2i+1) = \cos\left(\frac{\mathrm{pos}}{10000^{2i/d}}\right),$$

где pos — позиция токена, d — размерность вложения.

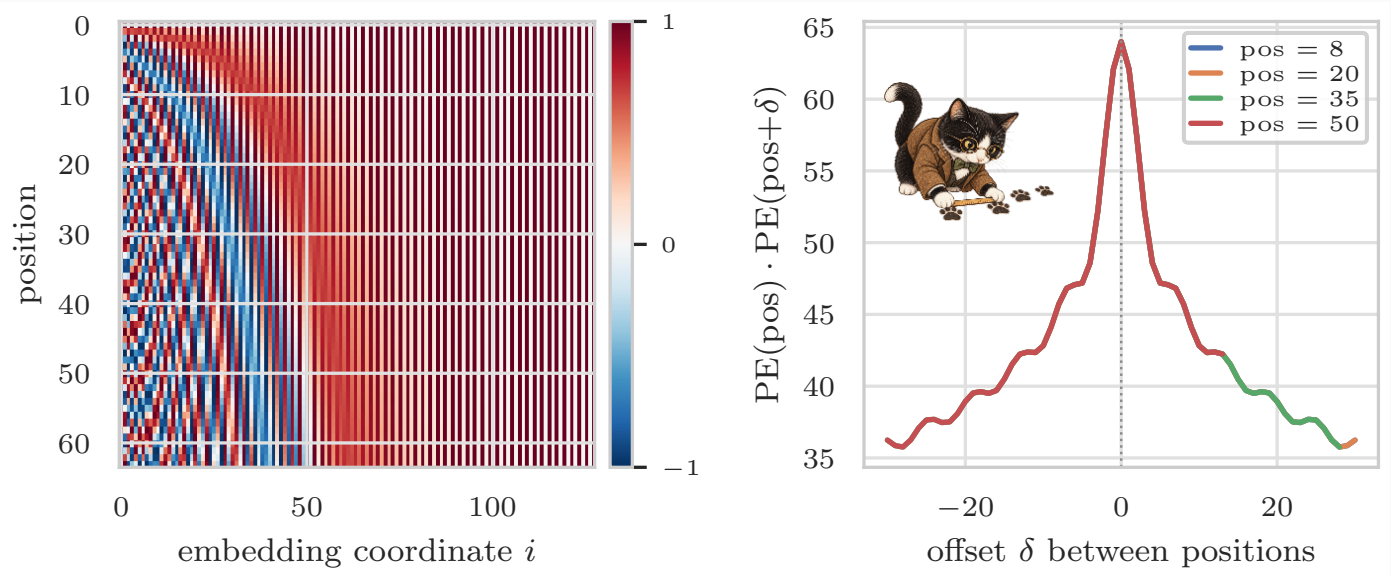




Positional encodings

- Главное свойство: $\mathrm{PE}(p)^\top \mathrm{PE}(p+\delta)$ зависит только от сдвига δ, ведь

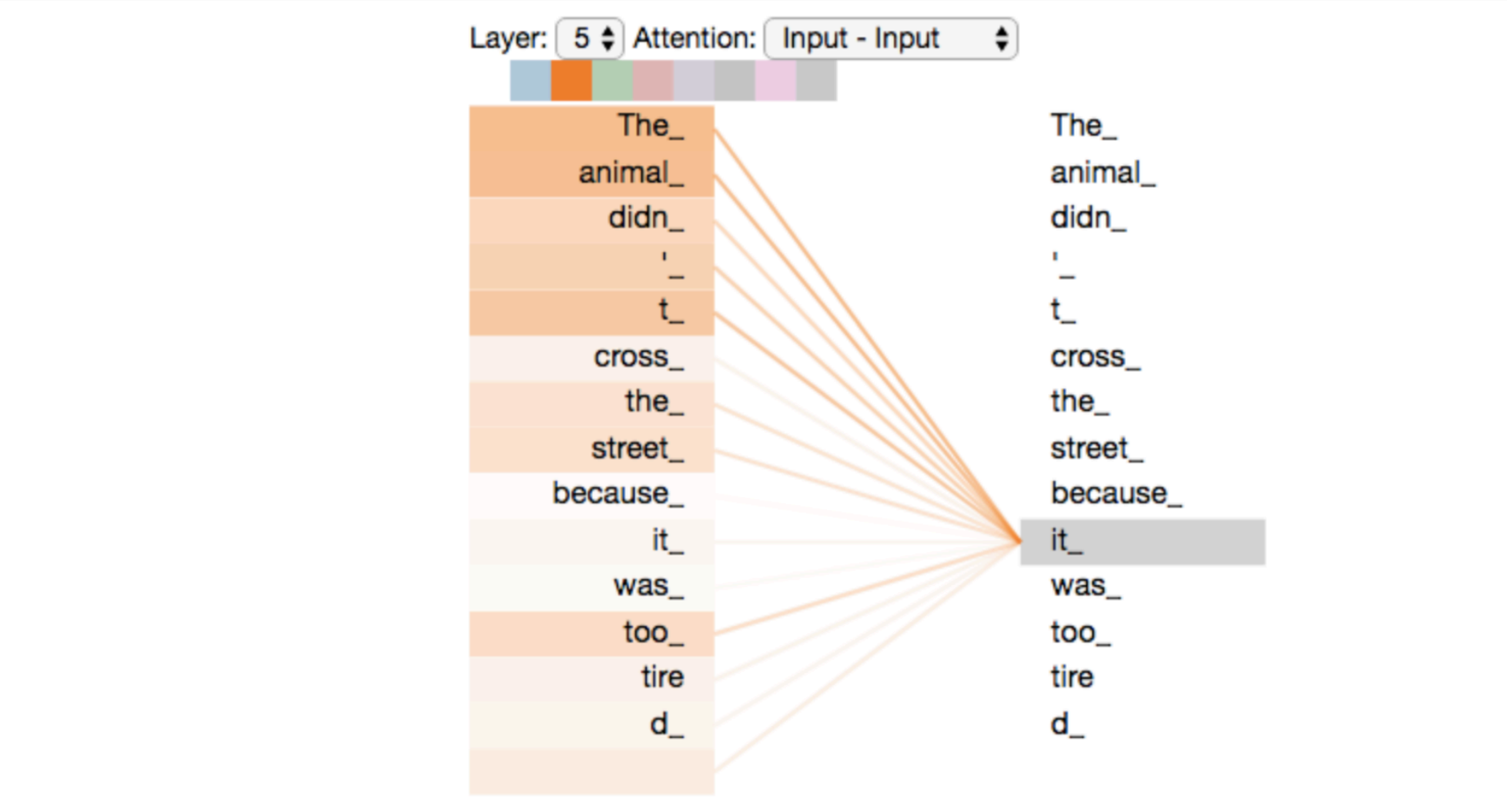
$$\sin(p\omega_i)\sin((p+\delta)\omega_i) + \cos(p\omega_i)\cos((p+\delta)\omega_i) = \cos(\delta\omega_i).$$



- Позиционные кодировки не конкатенируют, а *складывают* с вложениями токенов; выученные позиционные векторы работают почти так же (Vaswani et al., 2017).

Результаты

- В результате трансформер обучает веса того, как одни слова ”участвуют в обработке” других слов:



Результаты

- Бонус от self-attention – во-первых, вычислительный, во-вторых, сокращает пути между словами, в-третьих, потенциальная интерпретируемость:

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Результаты

- Работает лучше и обучается гораздо дешевле: примерно в 50 раз меньше FLOPs, чем у лучшего ансамбля того времени, и меньше четверти стоимости лучшей одиночной модели:

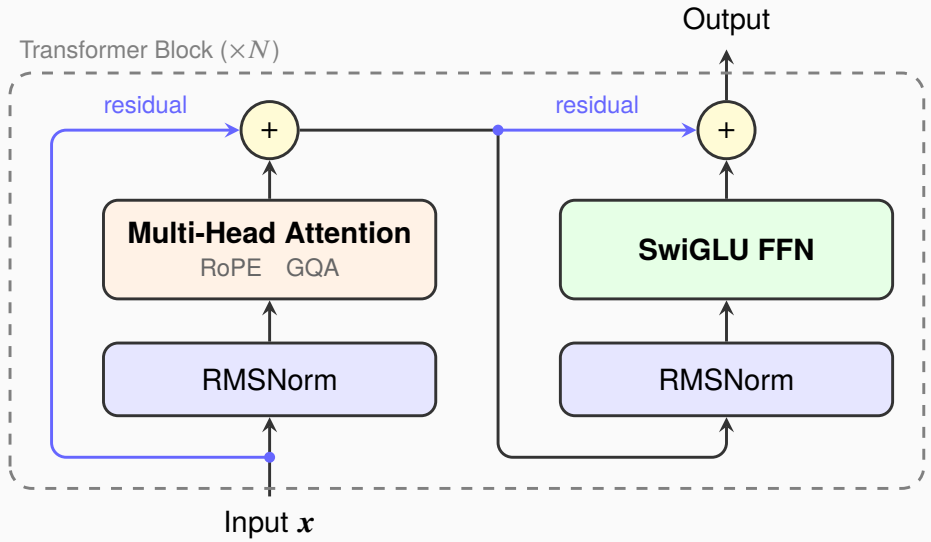
Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

Современный рецепт трансформера

Что изменилось с 2017 года

- Изменений к исходному трансформеру предлагали десятки, и большинство из них стабильно не помогает (Narang et al., 2021).
- Но LLaMA, Mistral, PaLM, GPT-NeoX, Pythia и другие пришли примерно к одним и тем же ответам:
  - как кодировать позицию — RoPE;
  - где и как нормализовать — Pre-LN с RMSNorm;
  - каким сделать FFN — SwiGLU;
  - сколько хранить голов ключей и значений — GQA;
  - как считать внимание, не переполняя память — FlashAttention.





Rotary position embeddings (RoPE)

- Синусоиды и выученные позиционные векторы кодируют *абсолютную* позицию; а вниманию на самом деле нужна *относительная*.
- RoPE (Su et al., 2021): не прибавляем позицию ко входу, а *поворачиваем* запросы и ключи:

$$\mathbf{q}'_m = R_m \mathbf{q}, \qquad \mathbf{k}'_n = R_n \mathbf{k},$$

$$R_m = \begin{pmatrix} \cos m\theta_0 & -\sin m\theta_0 & & & \\ \sin m\theta_0 & \cos m\theta_0 & & & \\ & & \cos m\theta_1 & -\sin m\theta_1 & \\ & & \sin m\theta_1 & \cos m\theta_1 & \\ & & & & \ddots \end{pmatrix},$$

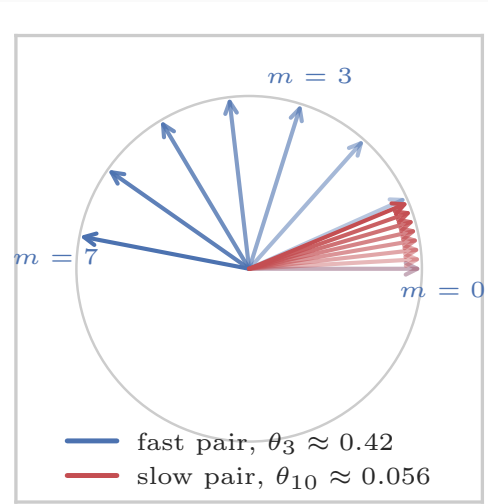
где $\theta_j = 10000^{-2j/d_h}$ — те же частоты, что у синусоид: пара координат j поворачивается на угол $m\theta_j$.

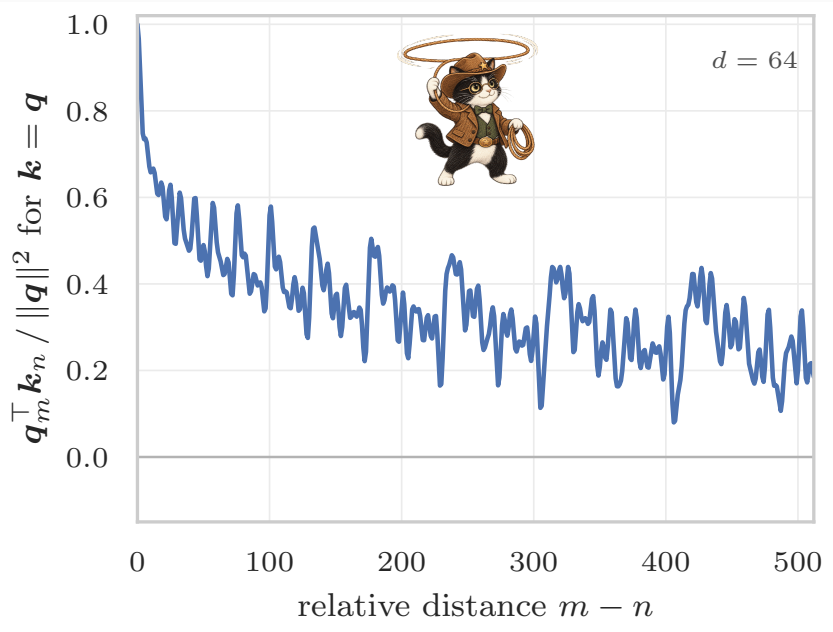
Rotary position embeddings (RoPE)

- Теперь в скалярном произведении

$$\mathbf{q}_m'^{\top}\mathbf{k}_n' = \mathbf{q}^{\top}R_m^{\top}R_n\mathbf{k} = \mathbf{q}^{\top}R_{n-m}\mathbf{k}$$

  остаётся только *относительная* позиция $n - m$; и никаких новых параметров.
- Быстрые пары координат делают много оборотов за предложение, медленные почти не двигаются:





Rotary position embeddings (RoPE)

- Проще всего увидеть это через комплексные числа: пару координат (q_{2j}, q_{2j+1}) считаем числом

$$\tilde{q}_j = q_{2j} + \mathrm{i}\, q_{2j+1} \in \mathbb{C}, \qquad j = 0, \dots, \tfrac{d_h}{2} - 1.$$

- Поворот на угол φ — это умножение на $e^{\mathrm{i}\varphi}$, так что RoPE — просто

$$\tilde{q}'_{m,j} = \tilde{q}_j\, e^{\mathrm{i}m\theta_j}, \qquad \tilde{k}'_{n,j} = \tilde{k}_j\, e^{\mathrm{i}n\theta_j}.$$

- Скалярное произведение двумерных векторов a, b — это $\mathrm{Re}(a\bar{b})$, поэтому

$$\mathbf{q}'^{\top}_m \mathbf{k}'_n = \mathrm{Re} \sum_j \tilde{q}_j e^{\mathrm{i}m\theta_j}\, \overline{\tilde{k}_j e^{\mathrm{i}n\theta_j}} = \mathrm{Re} \sum_j \tilde{q}_j \overline{\tilde{k}_j}\, e^{\mathrm{i}(m-n)\theta_j}.$$

- Абсолютные позиции сократились, осталась разность $m - n$. RoPE применяют к $\mathbf{q}$ и $\mathbf{k}$ в *каждом* слое (не к $\mathbf{v}$), а синусоиды прибавлялись один раз на входе.

Rotary position embeddings (RoPE)

- Беда та же, что у синусоид: обычный RoPE плохо экстраполирует. Обучили на длине L, а на длине $L' = sL$ модель видит углы $m\theta_j$, каких не было при обучении.
- *Position interpolation* (Chen et al., 2023): сжимаем позиции, $m \mapsto m/s$, то есть делим *все* частоты на s. Новых углов нет, но соседние токены становятся ближе, чем при обучении: страдают быстрые частоты, отвечающие за локальный порядок.
- *NTK-aware scaling* (bloc97, Reddit, 2023): меняем не позиции, а основание,

$$b' = b \cdot s^{d_h/(d_h-2)} \quad \Longrightarrow \quad \theta'_j = \theta_j \cdot s^{-2j/(d_h-2)}.$$

  Самая быстрая пара ($j = 0$) не меняется, самая медленная ($j = \frac{d_h}{2} - 1$) делится ровно на s, между ними — плавно.
- Отсюда и главный гиперпараметр — основание: 10000 досталось от синусоид, у Llama 3 оно 500000.

Rotary position embeddings (RoPE)

- *YaRN* (Peng et al., 2023): смотрим, сколько оборотов делает пара j за обученный контекст, $r_j = L\theta_j/2\pi$:
  - $r_j > 32$ (много оборотов): частоту не трогаем;
  - $r_j < 1$ (меньше оборота): интерполируем полностью, θ_j/s;
  - между ними — линейная смесь $\theta'_j = (1-\gamma_j)\frac{\theta_j}{s} + \gamma_j\theta_j$, $\gamma_j = \frac{r_j - 1}{31}$;
  - и ещё немного «остужаем» softmax: логиты умножаем на $(0.1\ln s + 1)^2$.
- Дообучение — около 0.1% исходных токенов, и Llama 2 работает на 64K–128K токенов.
- *ALiBi* (Press et al., 2022): вообще без позиционных векторов, просто штраф за расстояние в оценках внимания:

$$\alpha_{ij} = \mathrm{softmax}_j\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_h}} - m_h\,(i-j)\right), \qquad m_h = 2^{-8h/H},\ h = 1, \ldots, H.$$

У каждой головы свой наклон (для $H = 8$: $\frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{256}$); BLOOM, MPT. Но в современных LLM по умолчанию — RoPE.

Нормализация: RMSNorm, Pre-LN и Post-LN

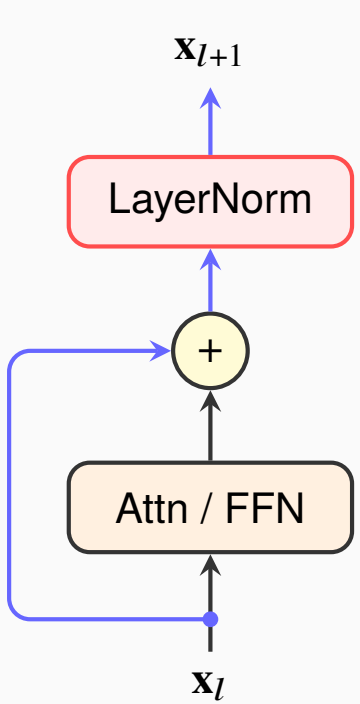
- LayerNorm вычитает среднее и делит на стандартное отклонение; RMSNorm (Zhang, Sennrich, 2019) не вычитает среднее:

$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\mathrm{RMS}(\mathbf{x})} \odot \gamma, \qquad \mathrm{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \varepsilon}.$$

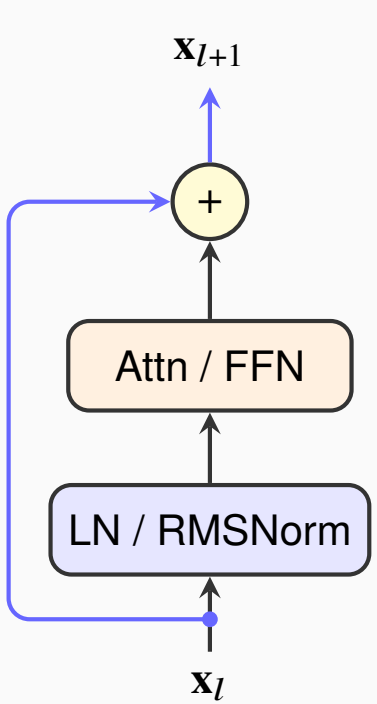
- Дешевле и работает примерно так же.

Нормализация: RMSNorm, Pre-LN и Post-LN

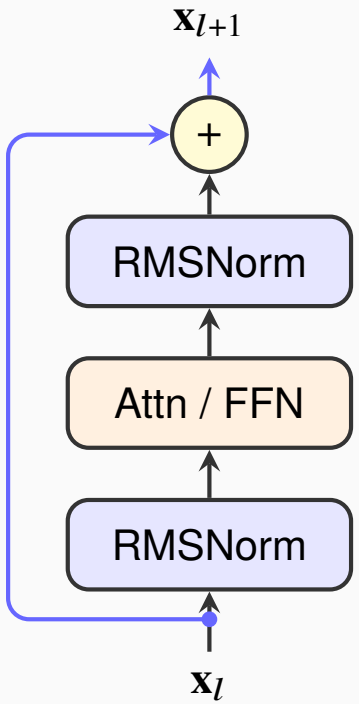
- Но важнее, *где* нормализовать:



(a) Post-LN
Transformer 2017, BERT



(b) Pre-LN
GPT-2, LLaMA, most LLMs



(c) Peri-LN (“sandwich”)
Gemma 2/3; OLMo 2: output norm only

Нормализация: RMSNorm, Pre-LN и Post-LN

- Post-LN (исходный трансформер): нормализация *после* сложения с остаточной связью,

$$\mathbf{x}_{l+1} = \mathrm{LN}(\mathbf{x}_l + F_l(\mathbf{x}_l)).$$

- Pre-LN: нормализация *перед* подслоем, остаточный путь чистый,

$$\mathbf{x}_{l+1} = \mathbf{x}_l + F_l(\mathrm{LN}(\mathbf{x}_l)) \quad \Longrightarrow \quad \mathbf{x}_L = \mathbf{x}_0 + \sum_{l=0}^{L-1} F_l(\mathrm{LN}(\mathbf{x}_l)).$$

- В Pre-LN от входа к выходу идёт тождественный путь, как в ResNet: $\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} = I + \ldots$, и градиент доходит до любого слоя.
- В Post-LN по этому пути градиент проходит через нормализацию в *каждом* слое.

Нормализация: RMSNorm, Pre-LN и Post-LN

- Xiong et al. (2020): при инициализации градиент по весам последних слоёв
  - в Post-LN — порядка $d\sqrt{\ln d}$, большой и не зависит от глубины L;
  - в Pre-LN — порядка $d\sqrt{\ln d / L}$, убывает с глубиной.
- Поэтому Post-LN с большой скоростью обучения в начале просто расходится, и исходному трансформеру нужен был прогрев (warmup): 4000 шагов линейного роста η.
- С Pre-LN warmup можно сильно сократить или убрать, и обучение устойчивее. GPT-2 (2019) уже был Pre-LN; теперь это стандарт.
- Но у Post-LN, если его удаётся обучить, качество бывает чуть лучше; отсюда множество компромиссов (DeepNorm, Peri-LN…).

Нормализация: RMSNorm, Pre-LN и Post-LN

- Обратная сторона Pre-LN — *проклятие глубины* (curse of depth; Sun et al., 2025).
- Остаточный поток только растёт: $\mathbf{x}_L = \mathbf{x}_0 + \sum_l F_l(\ldots)$, и его дисперсия увеличивается со слоем.
- Вклад очередного слоя $F_l(\mathrm{LN}(\mathbf{x}_l))$ имеет фиксированный масштаб (вход нормализован), а поток всё больше,

$$\frac{\partial \mathbf{x}_{l+1}}{\partial \mathbf{x}_l} = I + \frac{\partial F_l(\mathrm{LN}(\mathbf{x}_l))}{\partial \mathbf{x}_l} \approx I \quad \text{для глубоких } l.$$

- Глубокие слои почти тождественны и мало что делают: у Llama-2-70B можно выбросить до половины последних слоёв почти без потерь на вопросно-ответных тестах (Gromov et al., 2024).
- Лекарство Sun et al.: LayerNorm Scaling — выход нормализации в слое l умножаем на $1/\sqrt{l}$.

Нормализация: RMSNorm, Pre-LN и Post-LN

- Другое лекарство — нормализовать ещё и *выход* подслоя, перед сложением: Swin v2, Gemma 2 и 3 (RMSNorm и до, и после, Peri-LN на схеме), OLMo 2 (только после).
- А можно вообще без нормализации: Zhu et al. (2025) заменили её на поэлементный

$$\mathrm{DyT}(\mathbf{x}) = \gamma \odot \tanh(\alpha \mathbf{x}) + \beta$$

  и получили те же результаты на зрении и языке.
- Так что где нормализовать и нормализовать ли вообще — вопрос до сих пор открытый.

SwiGLU

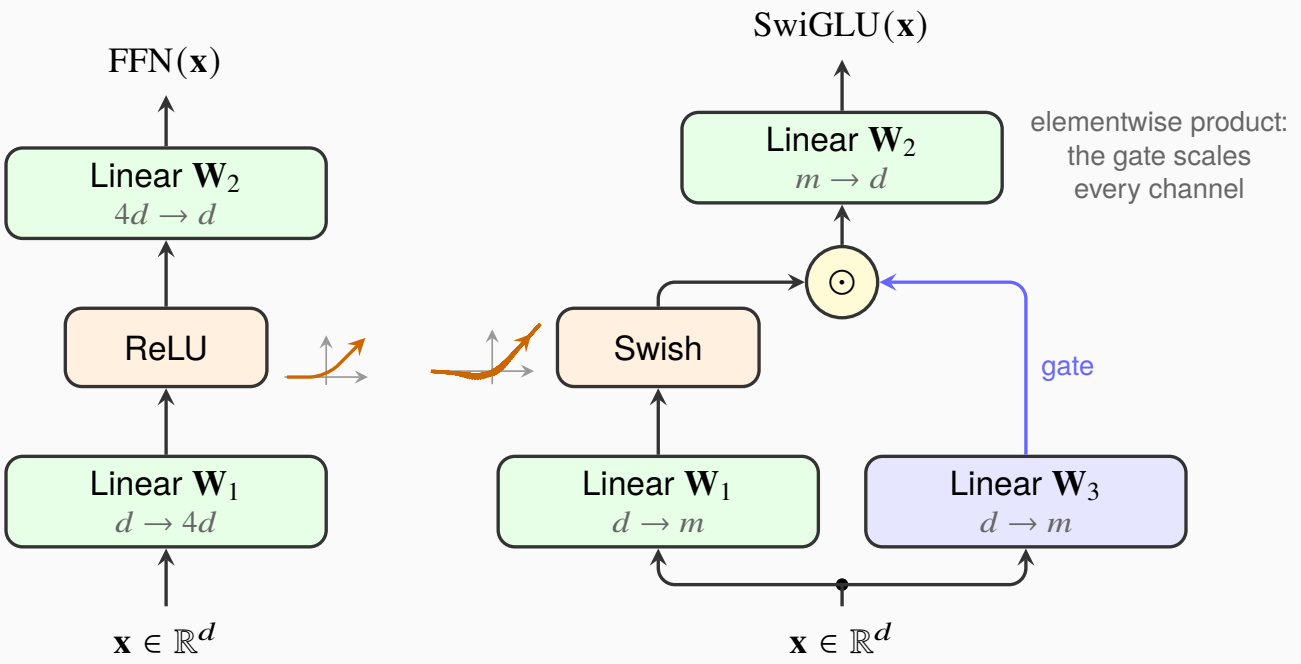
- FFN в исходном трансформере:
  $\mathrm{FFN}(\mathbf{x}) = \mathrm{ReLU}(\mathbf{x}W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$.
- SwiGLU (Shazeer, 2020) — FFN с *гейтом*:

$$\mathrm{SwiGLU}(\mathbf{x}) = \left(\mathrm{Swish}(\mathbf{x}W_1) \odot (\mathbf{x}W_3)\right) W_2, \qquad \mathrm{Swish}(x) = x \cdot \sigma(x).$$

- Ветка $\mathbf{x}W_3$ работает как гейт, модулирующий сигнал из ветки со Swish; это gated linear unit, тот же принцип, что в LSTM.
- Почему работает? Статья честно объясняет успех «as all else, to divine benevolence».
- Лишняя матрица W_3 окупается сужением: у ReLU-FFN ширины d_{ff} примерно $2dd_{\text{ff}}$ весов, у SwiGLU ширины m — $3dm$, так что берут $m \approx \frac{2}{3}d_{\text{ff}}$.
- Теперь это стандарт: PaLM, LLaMA, Mistral, почти все современные LLM (у Gemma — GEGLU).

SwiGLU

- Обычный FFN против SwiGLU: две ветки, одна проходит через Swish, другая служит гейтом, и они перемножаются поэлементно:



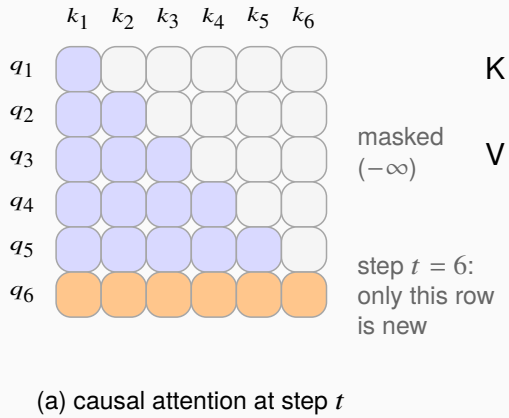
(a) classical feedforward

(b) SwiGLU

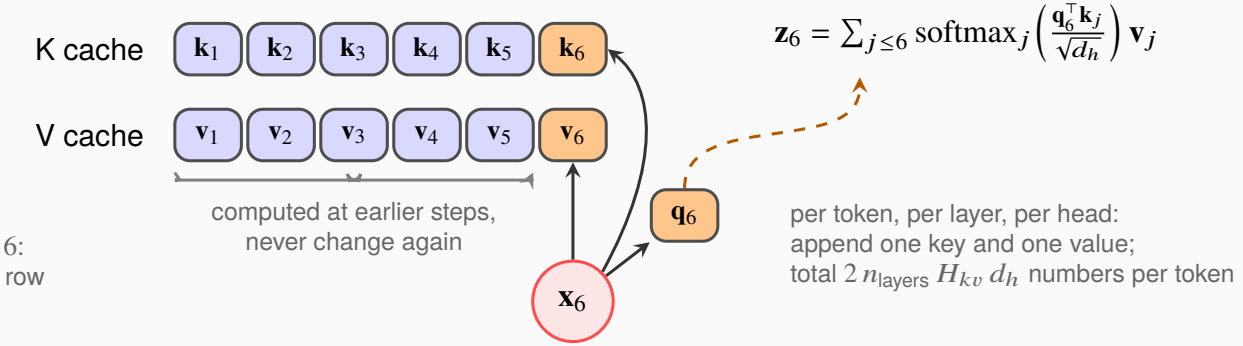
- Например, у LLaMA-7B $d = 4096$: обычный FFN был бы шириной $4d = 16384$, а SwiGLU — $\frac{2}{3} \cdot 4d \approx 10923$, округлённое до кратного 256: $m = 11008$.

KV-кэш и grouped-query attention

- Порождение идёт по одному токену: на шаге t нужен выход только *последней* позиции.
- Маска стоит в *каждом* слое, так что по индукции по слоям представление позиции j на любом слое зависит только от $\mathbf{x}_1, \dots, \mathbf{x}_j$; значит, ключи и значения прошлых токенов не меняются, когда приходят новые (это точное равенство, а не приближение).
- Значит, их можно сохранить и на шаге t посчитать только $\mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t$ и одну строку внимания:



(a) causal attention at step t



(b) one layer, one head: the cache grows by one column

KV-кэш и grouped-query attention

- Шаг порождения с кэшем, в каждом слое и каждой голове:

$$\mathbf{q}_t, \mathbf{k}_t, \mathbf{v}_t = W^Q \mathbf{x}_t,\ W^K \mathbf{x}_t,\ W^V \mathbf{x}_t, \qquad K \leftarrow [K;\ \mathbf{k}_t], \quad V \leftarrow [V;\ \mathbf{v}_t],$$

$$\mathbf{z}_t = \mathrm{softmax}\left(\frac{\mathbf{q}_t K^\top}{\sqrt{d_h}}\right) V.$$

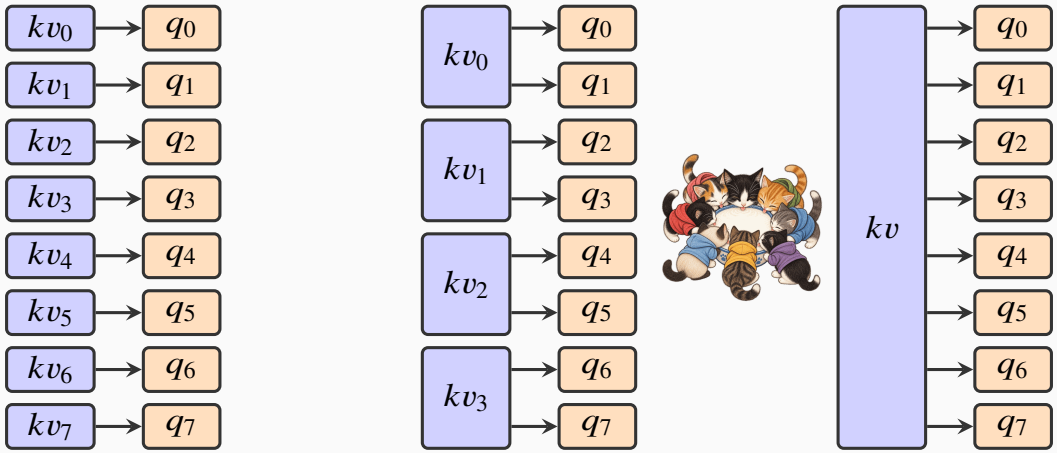
- Без кэша на шаге t пришлось бы заново прогнать все t токенов через все слои: $O(t\, d^2)$ на шаг, $O(T^2 d^2)$ на ответ длины T.
- С кэшем: проекции $O(d^2)$ и одна строка внимания $O(t\, d)$ на шаг, $O(T d^2 + T^2 d)$ на весь ответ.
- Платим памятью: кэш — это $2\, n_{\text{layers}}\, L\, H_{kv}\, d_h$ чисел на последовательность длины L, и он растёт с каждым токеном.

KV-кэш и grouped-query attention

- При развёртывании главное узкое место — не арифметика, а *пропускная способность памяти*.
- Два режима вывода:
  - *prefill*: весь промпт за один проход, упираемся в вычисления, $O(L^2 d)$ один раз;
  - *decode*: по одному токену, и на каждом шаге надо прочитать из памяти все веса и весь KV-кэш.
- Llama 2 7B: 32 слоя, 32 головы, $d_h = 128$, т.е. $2 \cdot 32 \cdot 32 \cdot 128 \approx 262$ тысячи чисел, полмегабайта на *каждый токен*.
- Одна последовательность из 4096 токенов — 2 ГБ кэша, батч из 16 — 32 ГБ, больше чем вдвое больше самих весов (13.5 ГБ).

KV-кэш и grouped-query attention

- Multi-query attention (MQA, Shazeer, 2019): одна голова ключей и значений на все головы запросов; кэш меньше в H раз, а качество теряется не так уж сильно.
- Grouped-query attention (GQA, Ainslie et al., 2023) — середина: $1 < H_{kv} < H_q$, одна голова ключей и значений на H_q / H_{kv} голов запросов.



- MHA можно «доучить» в GQA за $\approx 5\%$ исходного обучения; GQA в Llama 2 70B, во всей Llama 3 (восемь KV-голов), Mistral, Mixtral...

KV-кэш и grouped-query attention

- Multi-head latent attention (MLA, DeepSeek-V2): ключи и значения сжимаются в латентное пространство малой размерности *до* кэширования.
- Кэш меньше на 90% с лишним, а качество почти не страдает; матрицы «разжатия» можно алгебраически вмонтировать в соседние проекции.
- И трюки на уровне систем: paged attention (кэш хранится несмежными блоками), квантизация кэша до 8 или 4 бит.

FlashAttention

- Самовнимание строит матрицу $L \times L$: при $L = 8192$ и 32 головах в FP16 это $8192^2 \cdot 32 \cdot 2$ байта $= 4$ ГиБ на *один* тензор.
- FlashAttention (Dao et al., 2022) считает *точное* внимание, никогда не храня эту матрицу целиком.
- Идея — *IO-awareness*: у GPU маленькая быстрая SRAM на кристалле и большая медленная HBM снаружи.
- Считаем внимание *блоками*: загружаем куски Q, K, V в SRAM, считаем частичные выходы и аккумулируем, используя онлайн-softmax.
- Дополнительная память $O(L^2) \to O(L)$, обращений к медленной памяти гораздо меньше; арифметика та же, а по времени обычно в 2–4 раза быстрее.

FlashAttention

- Как считать softmax по блокам? Сначала — *безопасный* softmax: вычитаем максимум, иначе e^{s_j} переполняется (в FP16 уже $e^{12} > 65504$):

$$\mathrm{softmax}(\mathbf{s})_j = \frac{e^{s_j - m}}{\sum_k e^{s_k - m}}, \qquad m = \max_k s_k.$$

- Проблема: для этого нужен максимум по *всей* строке, а строка оценок $s_j = \mathbf{q}^\top \mathbf{k}_j / \sqrt{d_h}$ приходит блоками ключей.
- Online softmax (Milakov, Gimelshein, 2018): для каждой строки храним три вещи —
  - m — максимум оценок, которые уже видели;
  - ℓ — сумму $\sum e^{s_j - m}$ по ним;
  - $\mathbf{o}$ — ненормированный выход $\sum e^{s_j - m} \mathbf{v}_j$.

FlashAttention

- Пришёл новый блок ключей и значений B:

$$m' = \max\left(m,\ \max_{j \in B} s_j\right),$$

$$\ell' = e^{m - m'}\,\ell + \sum_{j \in B} e^{s_j - m'},$$

$$\mathbf{o}' = e^{m - m'}\,\mathbf{o} + \sum_{j \in B} e^{s_j - m'}\,\mathbf{v}_j.$$

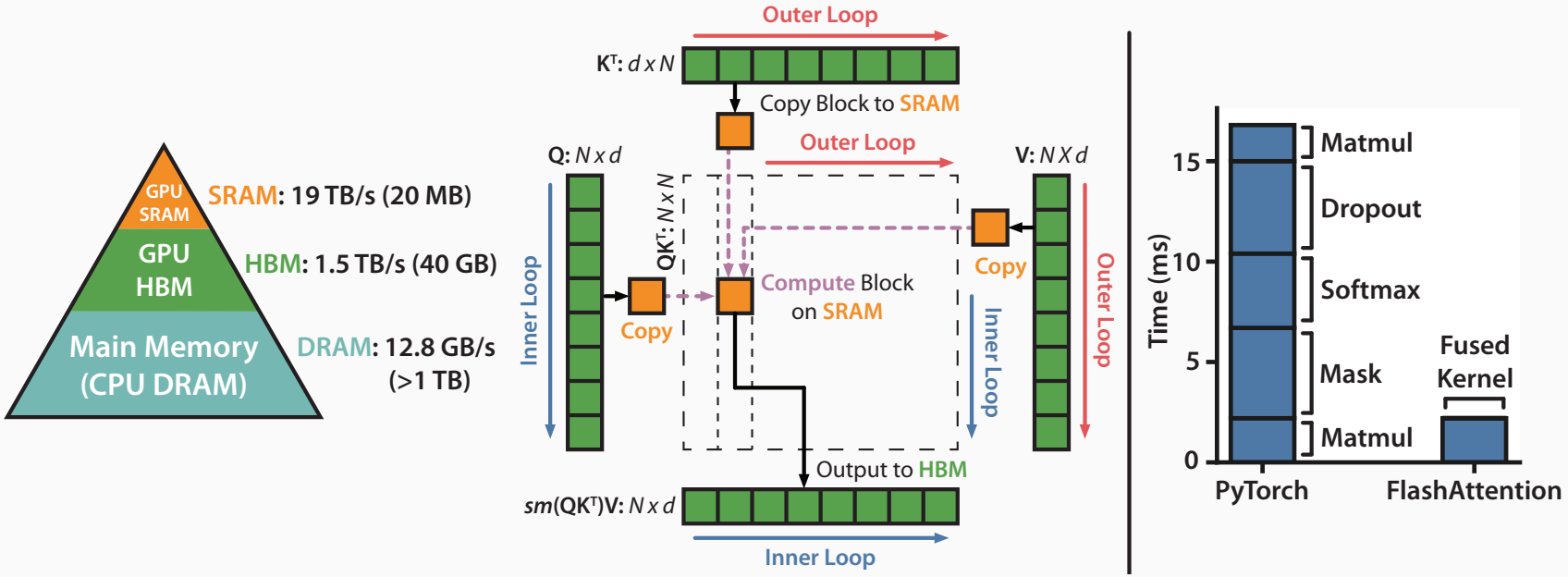
- Множитель $e^{m - m'}$ «перевешивает» всё накопленное к новому максимуму; если максимум не изменился, он равен 1.
- После последнего блока $\mathbf{z} = \mathbf{o}/\ell$ — ровно $\mathrm{softmax}(\mathbf{s})\,V$, без всяких приближений.
- Пример: строка $\mathbf{s} = (1, 3 \mid 2, 5)$ двумя блоками. После первого $m = 3$, $\ell = e^{-2} + 1 \approx 1.135$; после второго $m' = 5$, $\ell' = e^{-2} \cdot 1.135 + e^{-3} + 1 \approx 1.203$ — столько же, сколько $e^{-4} + e^{-2} + e^{-3} + 1$ сразу.

FlashAttention

- Каждый блок Q, K, V один раз загружается в SRAM, там считаются оценки, обновляются m, ℓ, $\mathbf{o}$, и в HBM пишется только итоговый $\mathbf{z}$.
- Матрица $L \times L$ не появляется в медленной памяти никогда; в обратном проходе её блоки *пересчитывают* заново: лишние вычисления дешевле лишних чтений памяти.

FlashAttention

- Иерархия памяти, разбиение на блоки и одно слитое ядро вместо пяти проходов по медленной памяти:



- Это не новая архитектура, а та же функция; но теперь FlashAttention фактически часть спецификации трансформера.
- FlashAttention-2 — лучше параллелизм (50–73% пиковой производительности GPU); FlashAttention-3 — под Hopper и FP8, ещё в 1.5–2 раза.

Другие улучшения

- *Параллельные внимание и FFN* (GPT-J, PaLM): вместо последовательного блока

$$\mathbf{z} = \mathbf{x} + \mathrm{Attn}(N(\mathbf{x})) + \mathrm{FFN}(N(\mathbf{x}));$$

  оба подслоя читают один и тот же нормализованный вход, цепочка зависимостей короче.
- *QK-нормализация*: на больших масштабах логиты $\mathbf{q}^\top \mathbf{k}/\sqrt{q}$ растут, softmax становится почти one-hot, и обучение расходится. Нормализуем запросы и ключи перед скалярным произведением; к 2025 году это стандарт (Gemma 3, OLMo 2, Qwen3).
- Альтернатива — soft-capping логитов $c\tanh(x/c)$ (Gemma 2).
- *Без смещений*: вслед за PaLM большинство моделей убирают bias из линейных слоёв и нормализаций; стабильнее и ничего не стоит.

Другие улучшения

- *Локальное и глобальное внимание*: самая дешёвая запись в KV-кэше — та, которую не храним.
- Слой со скользящим окном смотрит только на W предыдущих токенов: кэш не больше W, а рецептивное поле всё равно растёт на W с каждым слоем, как у свёрток.
- Mistral 7B: окно 4096 во всех слоях; Gemma 2: локальные и глобальные слои через один; Gemma 3: пять локальных с окном 1024 на один глобальный.
- И позиционная информация делится между ними: в Command A и Llama 4 глобальные слои вообще без позиционных кодировок (NoPE), порядок слов дают локальные.

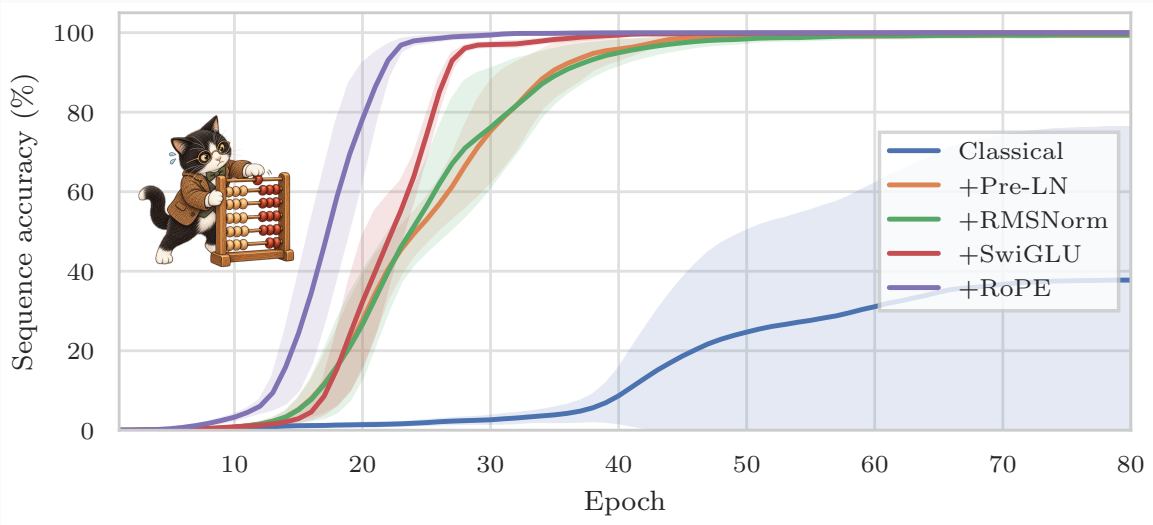
Рецепт в деле: эксперимент

- Маленький трансформер ($d = 64$, 4 головы, $2 + 2$ слоя), пять вариантов, каждый добавляет один ингредиент:
  - Classical (Post-LN, LayerNorm, ReLU, синусоиды) $\rightarrow$ +Pre-LN $\rightarrow$ +RMSNorm $\rightarrow$ +SwiGLU $\rightarrow$ +RoPE.
- Две задачи: развернуть строку из 10 символов и сложить два трёхзначных числа.

		Разворот		Сложение	
Вариант	Параметров	Точн., %	Эп.$\rightarrow$90%	Точн., %	Эп.$\rightarrow$90%
Classical	237 533	100.0	17	37.7 ± 38.3	—
+Pre-LN	237 789	100.0	13	99.4 ± 0.2	34
+RMSNorm	237 021	100.0	13	99.5 ± 0.4	35
+SwiGLU	233 693	100.0	14	100.0 ± 0.0	26
+RoPE	233 693	98.5	33	100.0 ± 0.0	21

Рецепт в деле: эксперимент

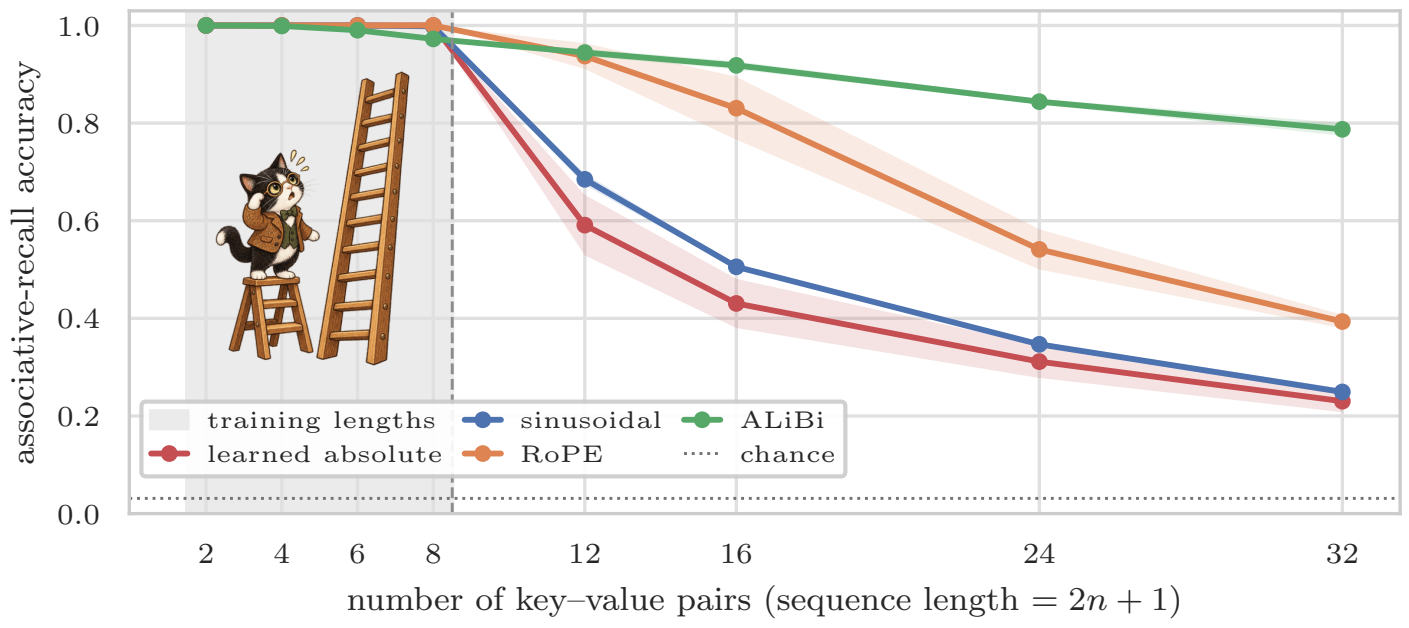
- Разворот: Pre-LN — самый большой выигрыш в скорости; а RoPE *замедляет*: задаче нужны абсолютные позиции («символ 0 идёт на место 9»).
- Сложение: Classical нестабилен (одно зерно выучило перенос разрядов, два — нет), Pre-LN это полностью лечит, а RoPE теперь быстрее всех:



- Универсально лучшего набора нет: Pre-LN — всегда, SwiGLU — немного лучше, а RoPE помогает или мешает в зависимости от задачи.

Экстраполяция на длину

- Ассоциативное припоминание: $[k_1, v_1, \ldots, k_n, v_n, k_q] \mapsto v_q$; учим на $2 \le n \le 8$ парах, проверяем до $n = 32$.
- Четыре модели отличаются только позиционным механизмом: выученные позиции, синусоиды, RoPE, ALiBi.



- Идеальная точность на длинах из обучения ничего не гарантирует за их пределами; лучше всех обобщает ALiBi.

Экстраполяция на длину

- А что внутри? Естественная двухслойная схема (*induction head*): на первом слое значение v_i читает стоящий перед ним ключ k_i, на втором запрос k_q находит значение с нужным ключом.
- В обученной модели с RoPE такая схема действительно появляется; на длине из обучения точность почти 100%, на вдвое большей — около 75%:



- Но веса внимания — не объяснение: на втором слое все головы смотрят куда надо, а выключение любой одной почти ничего не меняет; нужны и визуализация, и абляции.

Спасибо за внимание!



