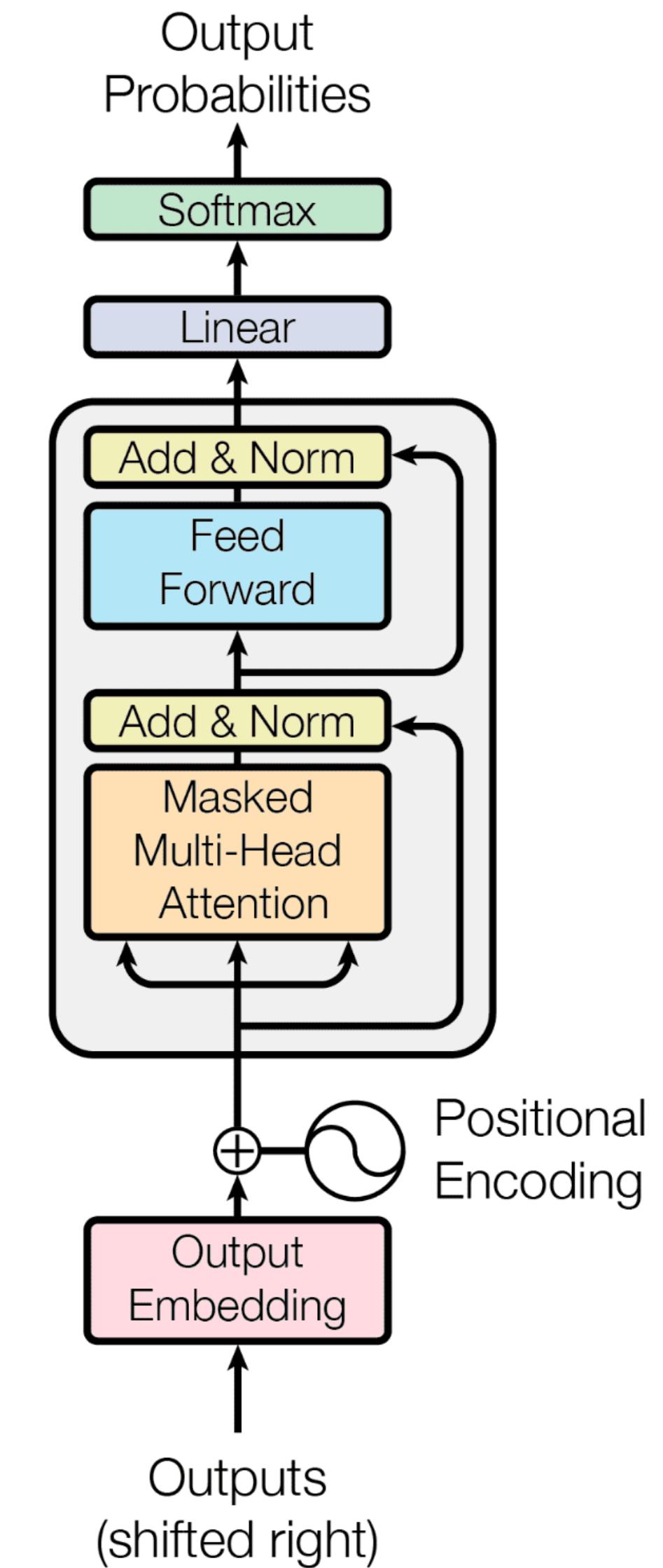


Устройство современных LLM

План

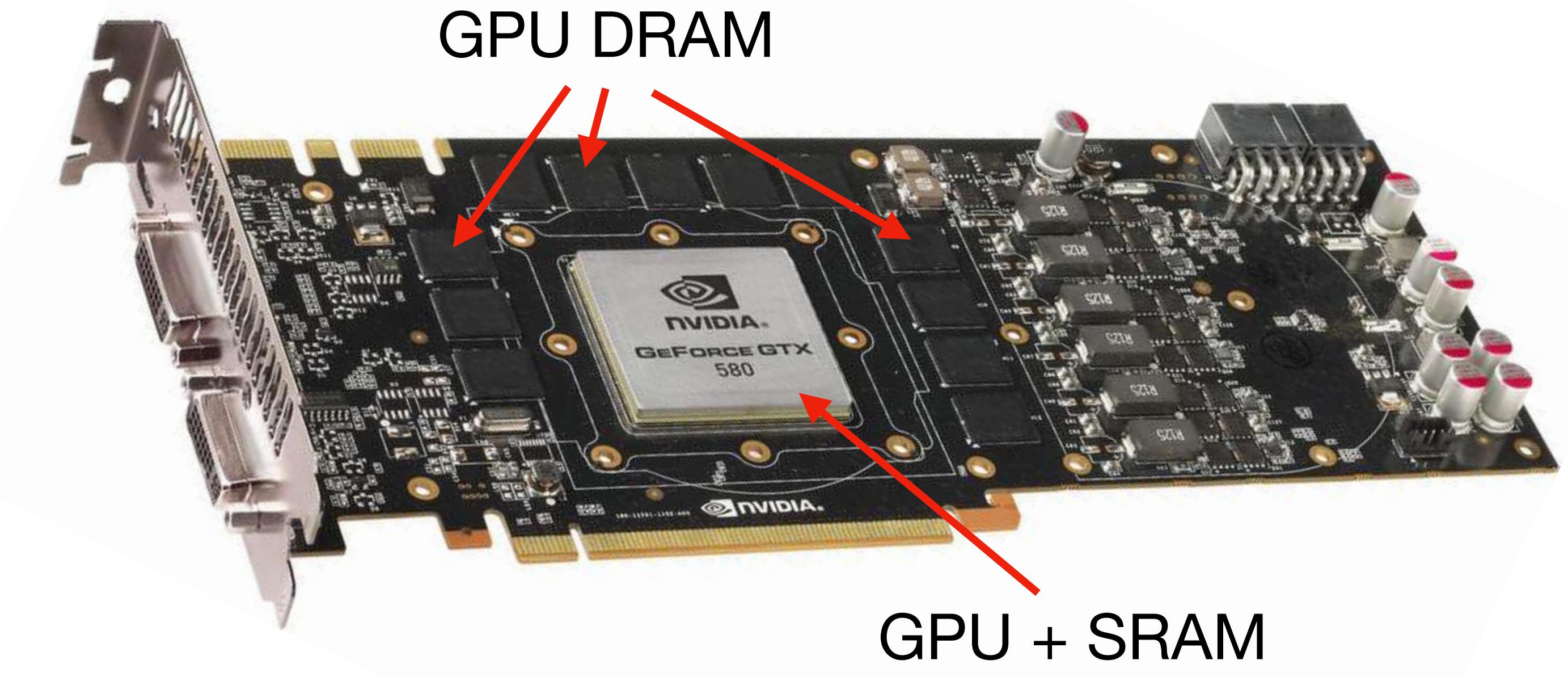
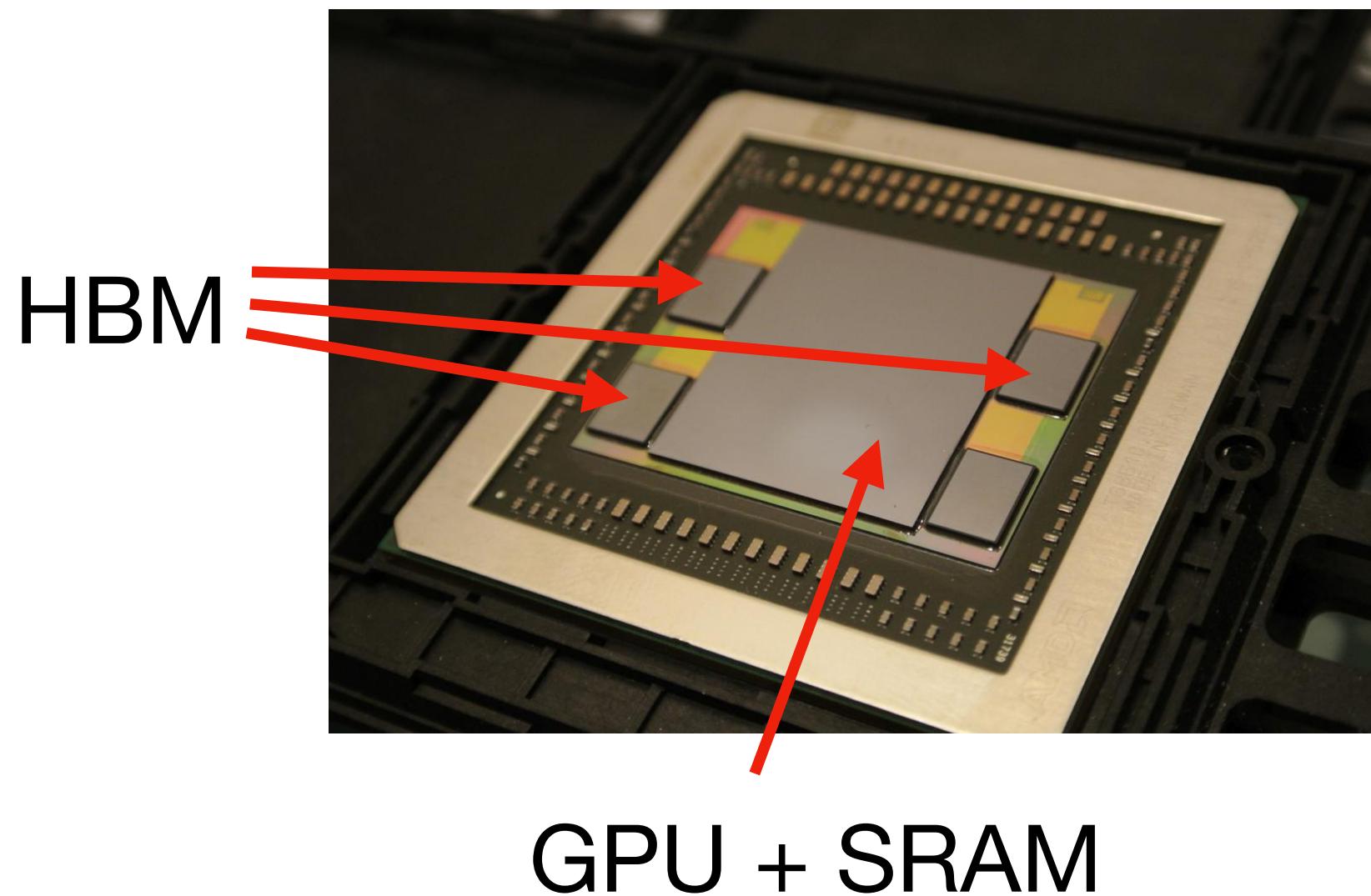
- Attention
- Positional Encoding
- Feed Forward Network
- Normalization



Attention

Проблема со скоростью памяти

- Видеокарты слишком быстрые, скорость ограничивается работой с памятью
- Существуют три вида памяти:
 - GPU SRAM (Static Random-Access Memory) – самая быстрая, но небольшая память, вшита в процессор (19 TB/s, 20 MB)
 - GPU HBM (High Bandwidth Memory) – основная память GPU (1.5 TB/s, 40 GB)
 - CPU DRAM (Dynamic Random-Access Memory) – оперативная память срн, самая медленная (12.8 GB/s, >1 TB)



Обычный attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

- 1: Load \mathbf{Q}, \mathbf{K} by blocks from HBM, compute $\mathbf{S} = \mathbf{QK}^\top$, write \mathbf{S} to HBM.
 - 2: Read \mathbf{S} from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write \mathbf{P} to HBM.
 - 3: Load \mathbf{P} and \mathbf{V} by blocks from HBM, compute $\mathbf{O} = \mathbf{PV}$, write \mathbf{O} to HBM.
 - 4: Return \mathbf{O} .
-

Flash attention

- Flash attention уменьшает число взаимодействий с НВМ, производя почти все вычисления на SRAM
- Attention считается по блокам, которые влезают в SRAM

$$m(x) := \max_i x_i, \quad f(x) := [e^{x_1 - m(x)} \quad \dots \quad e^{x_B - m(x)}], \quad \ell(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{\ell(x)}.$$

For vectors $x^{(1)}, x^{(2)} \in \mathbb{R}^B$, we can decompose the softmax of the concatenated $x = [x^{(1)} \ x^{(2)}] \in \mathbb{R}^{2B}$ as:

$$m(x) = m([x^{(1)} \ x^{(2)}]) = \max(m(x^{(1)}), m(x^{(2)})), \quad f(x) = [e^{m(x^{(1)}) - m(x)} f(x^{(1)}) \quad e^{m(x^{(2)}) - m(x)} f(x^{(2)})],$$

$$\ell(x) = \ell([x^{(1)} \ x^{(2)}]) = e^{m(x^{(1)}) - m(x)} \ell(x^{(1)}) + e^{m(x^{(2)}) - m(x)} \ell(x^{(2)}), \quad \text{softmax}(x) = \frac{f(x)}{\ell(x)}.$$

Flash attention

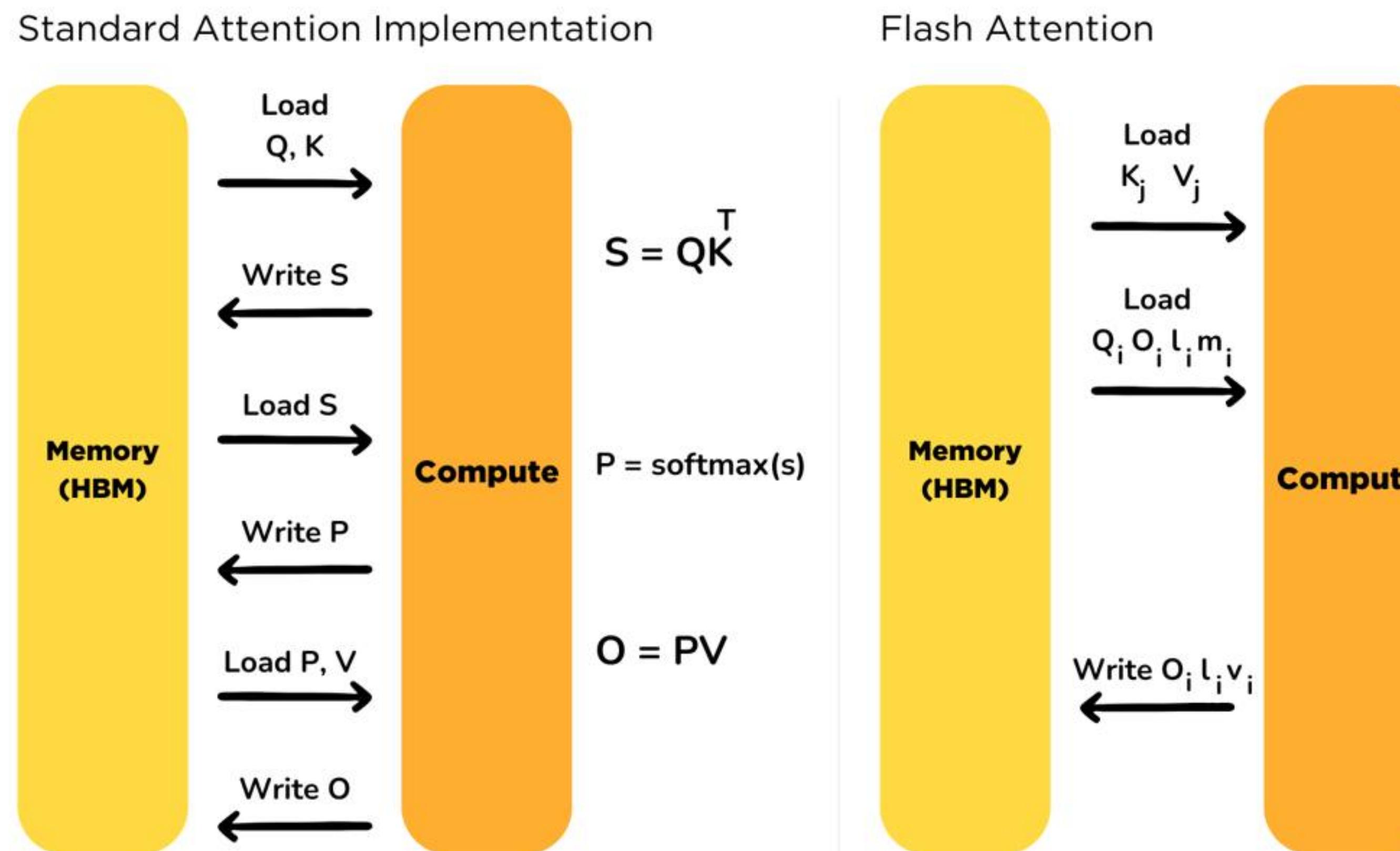
Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M .

- 1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil, B_r = \min(\lceil \frac{M}{4d} \rceil, d)$.
 - 2: Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.
 - 3: Divide \mathbf{Q} into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide \mathbf{K}, \mathbf{V} in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
 - 4: Divide \mathbf{O} into T_r blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \dots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks m_1, \dots, m_{T_r} of size B_r each.
 - 5: **for** $1 \leq j \leq T_c$ **do**
 - 6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
 - 7: **for** $1 \leq i \leq T_r$ **do**
 - 8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
 - 9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.
 - 10: On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}, \tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
 - 11: On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}, \ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
 - 12: Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1} (\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
 - 13: Write $\ell_i \leftarrow \ell_i^{\text{new}}, m_i \leftarrow m_i^{\text{new}}$ to HBM.
 - 14: **end for**
 - 15: **end for**
 - 16: Return \mathbf{O} .
-

Flash attention

- Flash attention уменьшает число взаимодействий с HBM, производя почти все вычисления на SRAM
- Attention считается по блокам, которые влезают в SRAM
- O – матрица выходов, I – нормировочная константа softmax, m – максимальное значение скора внимания



Flash attention

Flash attention увеличивает число операций, но значительно (в 6-9 раз) уменьшает время вычисления благодаря сокращению обращений к HBM

Attention	Standard	FLASHATTENTION
GFLOPs	66.6	75.2
HBM R/W (GB)	40.3	4.4
Runtime (ms)	41.7	7.3

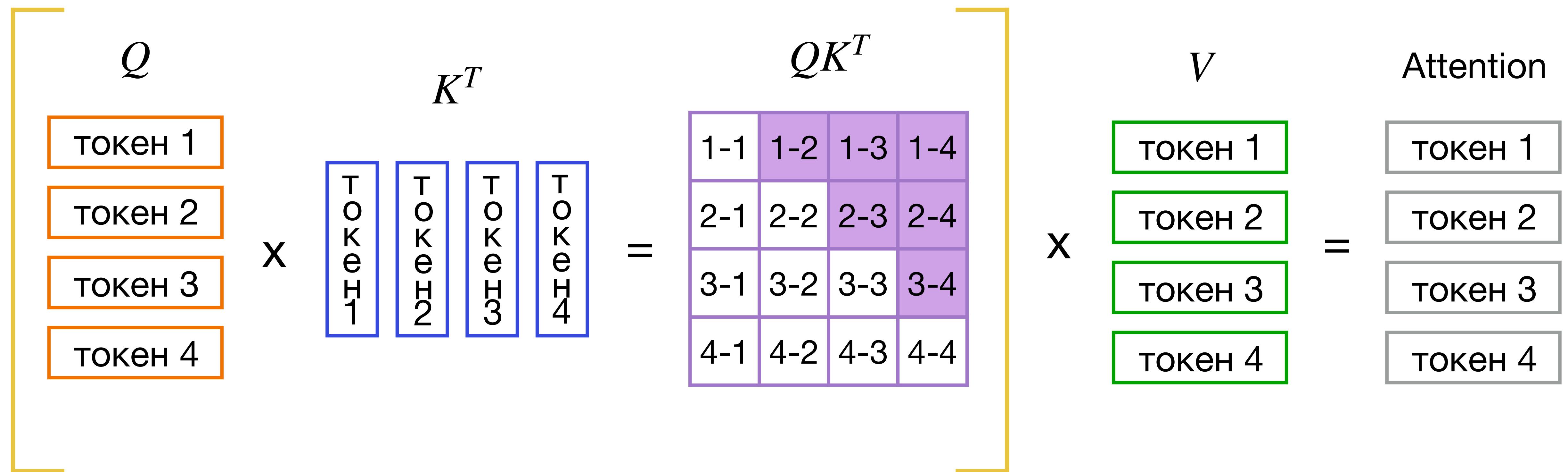
Генерация текста

- Все авторегрессионные модели генерируют текст по одному токену
- На каждой итерации необходимо пропускать всю последовательность через модель
- Один подсчет внимания занимает $O(l^2hd + lh^2d^2)$ операций

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

Генерация текста

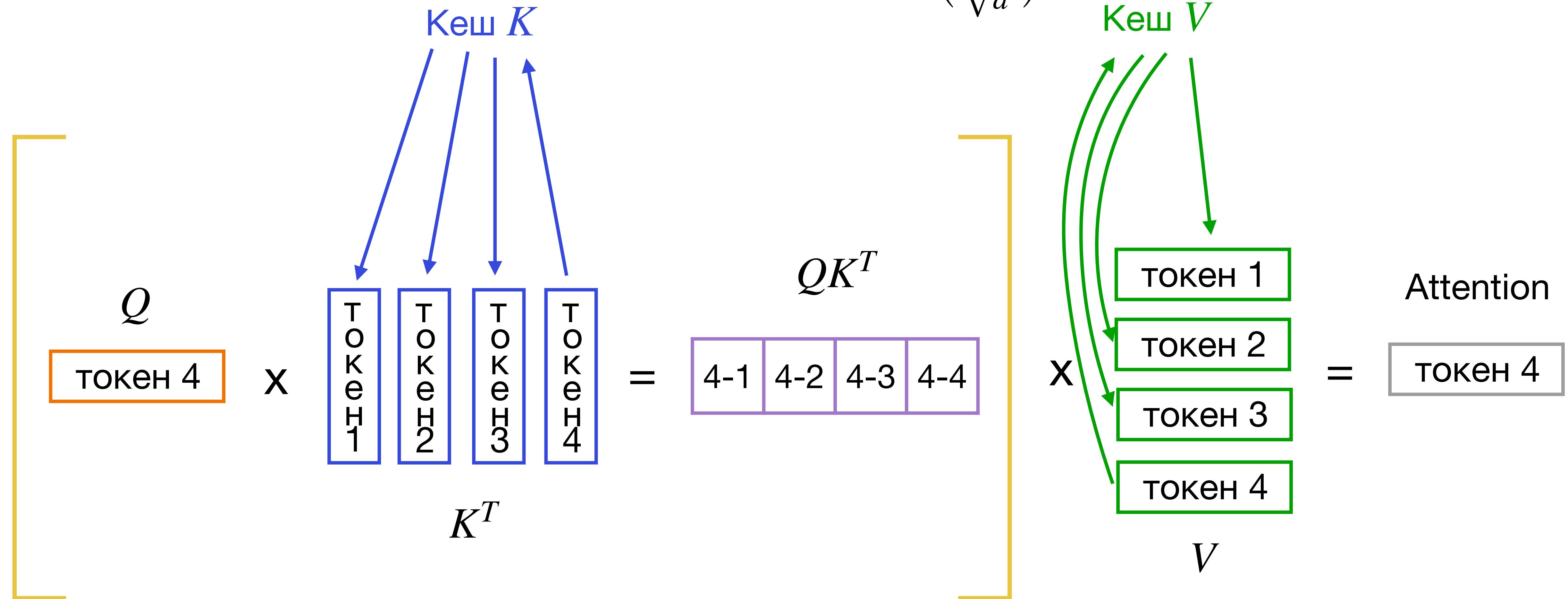
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



Мы пересчитываем заново значения Q , K и V , хотя они не изменяются!
Каждый токен зависит только от тех, что перед ним

KV кеширование

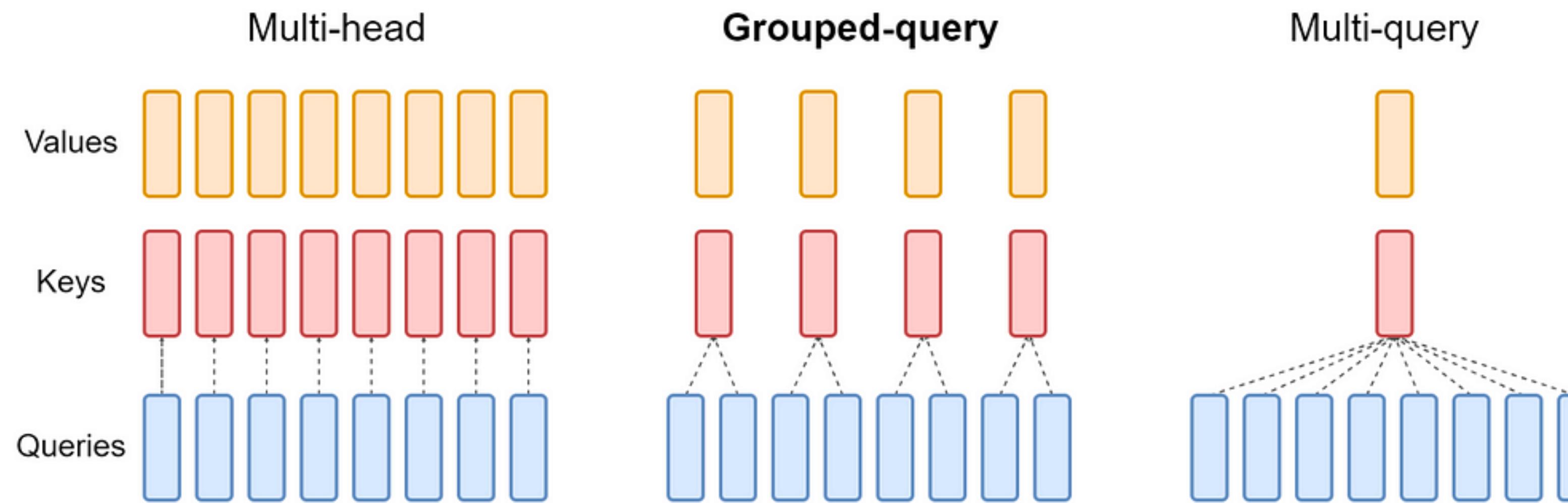
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



На каждую итерацию тратим $O(lhd + hd^2)$ времени вместо $O(l^2hd + lh^2d^2)$!

Multi-Query Attention

- KV кэширование требует хранения выходов всех слоев
- Скорость генерации падает из-за работы с памятью
- Можно оставить одну голову для K и V
- Обычно (Llama2, Falcon, Mistral, PaLM) выбирают средний вариант и оставляют около 8 голов

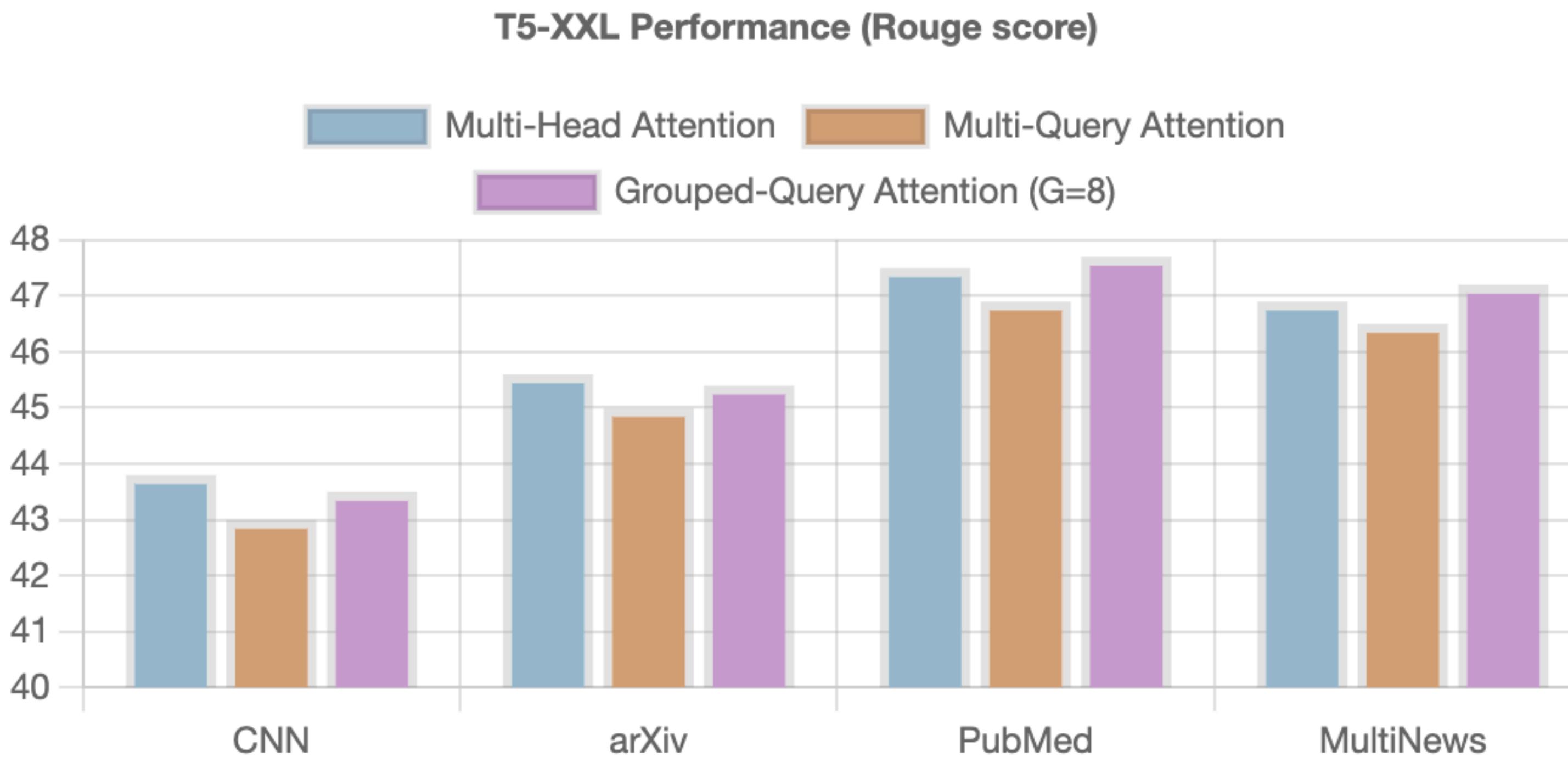


Multi-Query Attention

Attention Type	h	d_k, d_v	d_{ff}	ln(PPL) (dev)	BLEU (dev)	BLEU (test) beam 1 / 4
multi-head	8	128	4096	1.424	26.7	27.7 / 28.4
multi-query	8	128	5440	1.439	26.5	27.5 / 28.5
multi-head local	8	128	4096	1.427	26.6	27.5 / 28.3
multi-query local	8	128	5440	1.437	26.5	27.6 / 28.2

Attention Type	Training	Inference enc. + dec.	Beam-4 Search enc. + dec.
multi-head	13.2	1.7 + 46	2.0 + 203
multi-query	13.0	1.5 + 3.8	1.6 + 32
multi-head local	13.2	1.7 + 23	1.9 + 47
multi-query local	13.0	1.5 + 3.3	1.6 + 16

Multi-Query Attention



Позиционные энкодинги

Абсолютные позиционные энкодинги

- Не учитывается относительное расстояние
 - Модель нельзя применять к более длинным текстам
 - Важно не потерять информацию о позиции при работе с эмбеддингами

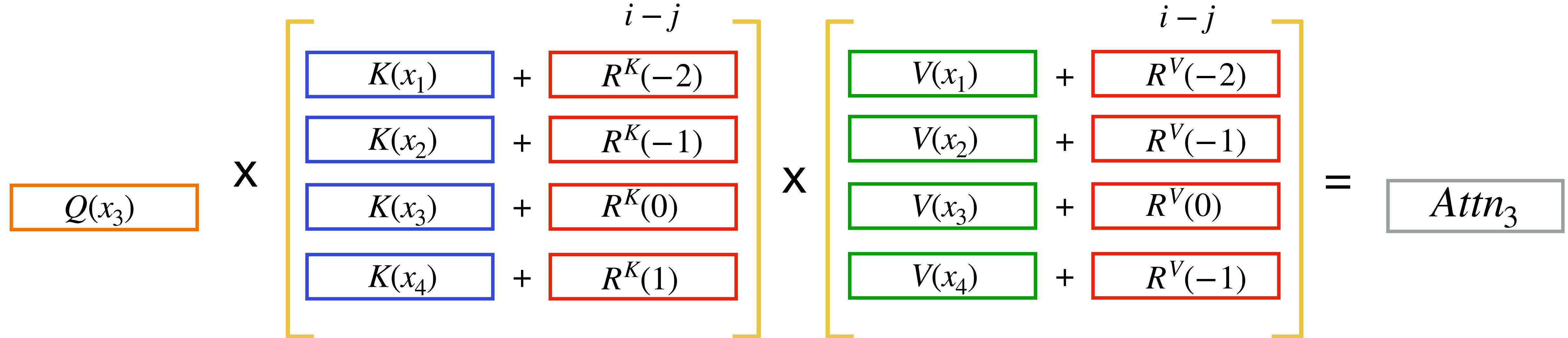
$$x_i = Emb(w_i) + Emb_{pos}(i)$$

Relative Position Encodings (RPE)

- Не используем абсолютные позиционные энкодинги
- Явно передаем в механизм внимания информацию об относительном расположении токенов
- RPE позволяет использовать модель с длинными текстами
- Используется в T5, ALBERT, DeBERTa, Longformer и т.д.

Relative Position Encodings (RPE)

$$Attn_i = \text{softmax} \left(\frac{Q_i^T (K^T + R_i^K)}{\sqrt{d}} \right) (V + R_i^V)$$



Relative Position Encodings (RPE)

- При реализации генерируется матрица относительных позиций и из нее находятся эмбеддинги
- Для возможности адаптации модели к длинным текстам можно ограничить максимальное расстояние между токенами

$$Attn_i = \text{softmax} \left(\frac{Q_i^T (K^T + R_i^K)}{\sqrt{d}} \right) (V + R_i^V)$$

$$P = \begin{pmatrix} 0 & 1 & 2 & 2 & 2 \\ -1 & 0 & 1 & 2 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -2 & -1 & 0 & 1 \\ -2 & -2 & -2 & -1 & 0 \end{pmatrix}$$

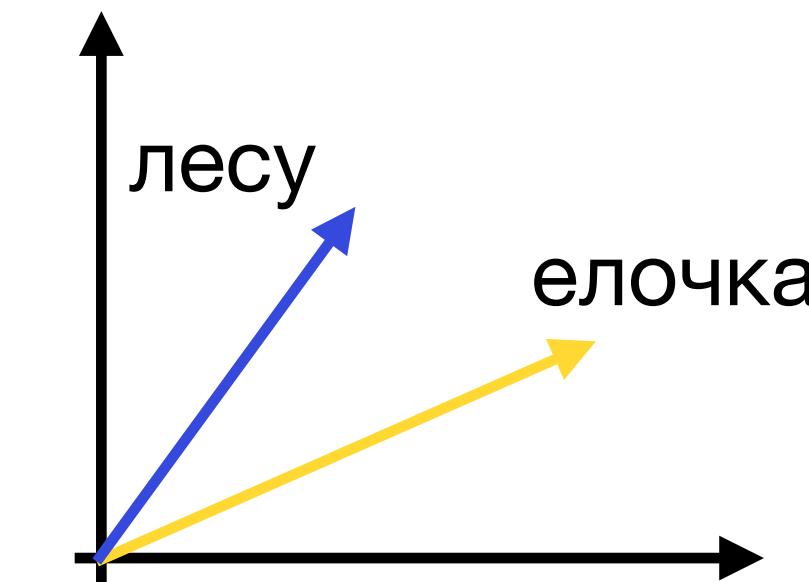
$$R^K = Emb_K(P) \in \mathbb{R}^{[n \times n \times d]}$$

$$R^V = Emb_V(P) \in \mathbb{R}^{[n \times n \times d]}$$

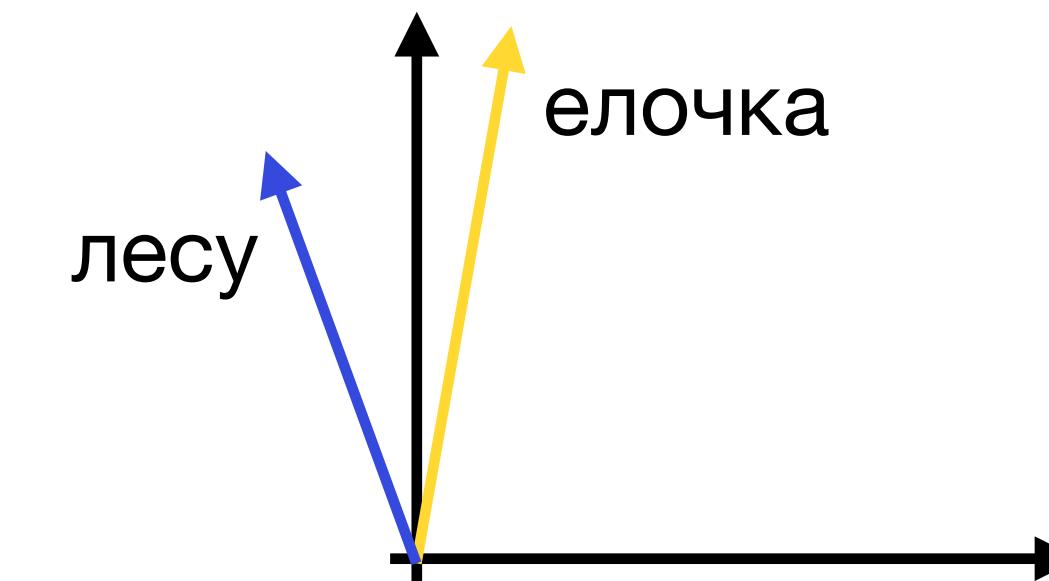
Rotary Position Embeddings (RoPE)

- Векторы q и k поворачиваются на угол $i\theta$, где i – позиция в тексте
- Таким образом, относительное расстояние не меняется при изменении позиции
- Векторы для похожих позиций будут поворачиваться на похожий угол, далекие векторы – на разные углы

в лесу родилась елочка



в нашем зимнем лесу родилась елочка



Математическая запись RoPE

Операция поворота вектора производится с помощью домножения на матрицу поворота

$$Q_i^r = \begin{pmatrix} \cos i\theta & -\sin i\theta \\ \sin i\theta & \cos i\theta \end{pmatrix} Q_i \quad K_j^r = \begin{pmatrix} \cos j\theta & -\sin j\theta \\ \sin j\theta & \cos j\theta \end{pmatrix} K_j$$

Внимание считается так же, как и раньше, но с повернутыми матрицами Q и K

$$\text{Attn} = \text{softmax}\left(\frac{Q^r K^{rT}}{\sqrt{d}}\right) V$$

Многомерный случай RoPE

- Матрица поворота заменяется на блочно-диагональную матрицу
- Каждый блок – матрица поворота с определенным углом

$$R_j^d = \begin{pmatrix} \cos j\theta_1 & -\sin j\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin j\theta_1 & \cos j\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos j\theta_2 & -\sin j\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin j\theta_2 & \cos j\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos j\theta_{d/2} & -\sin j\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin j\theta_{d/2} & \cos j\theta_{d/2} \end{pmatrix},$$

$$Q_i^{rT} K_j^r = (R_i^d Q_i)^T (R_j^d K_j) = Q_i^T R_i^{dT} R_j^d K_j = Q_i^T R_{j-i}^d K_j$$

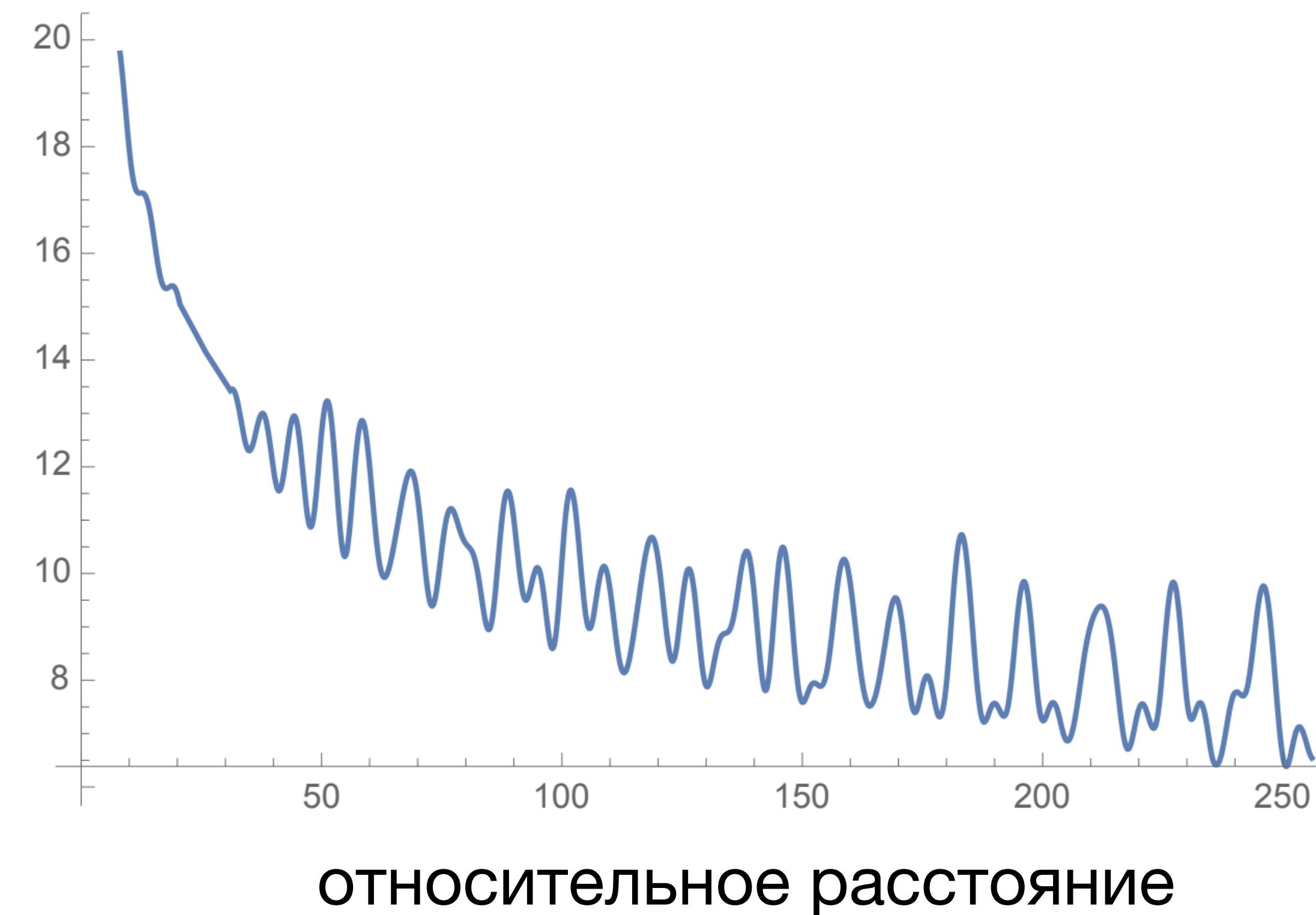
где $\theta_i = 10000^{-2(i-1)/d}$. Значение 10000 выбрано эмпирически.

Каждый угол задает свою частоту волны, поэтому часть координат хранит локальную информацию о позиции, а часть – глобальную

Обоснование RoPE

Благодаря выбранному углу, чем больше расстояние между токенами, тем меньше значение внимания между ними

Верхняя оценка на значение внимания



Эффективность подсчета RoPE

Пользуясь разреженностью матрицы R_j^d , можно значительно упростить умножение на нее

$$\begin{pmatrix} \cos j\theta & -\sin j\theta \\ \sin j\theta & \cos j\theta \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1 \cos j\theta - x_2 \sin j\theta \\ x_2 \cos j\theta + x_1 \sin j\theta \end{pmatrix}$$

$$R_j^d x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos j\theta_1 \\ \cos j\theta_1 \\ \cos j\theta_2 \\ \cos j\theta_2 \\ \vdots \\ \cos j\theta_{d/2} \\ \cos j\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin j\theta_1 \\ \sin j\theta_1 \\ \sin j\theta_2 \\ \sin j\theta_2 \\ \vdots \\ \sin j\theta_{d/2} \\ \sin j\theta_{d/2} \end{pmatrix}$$

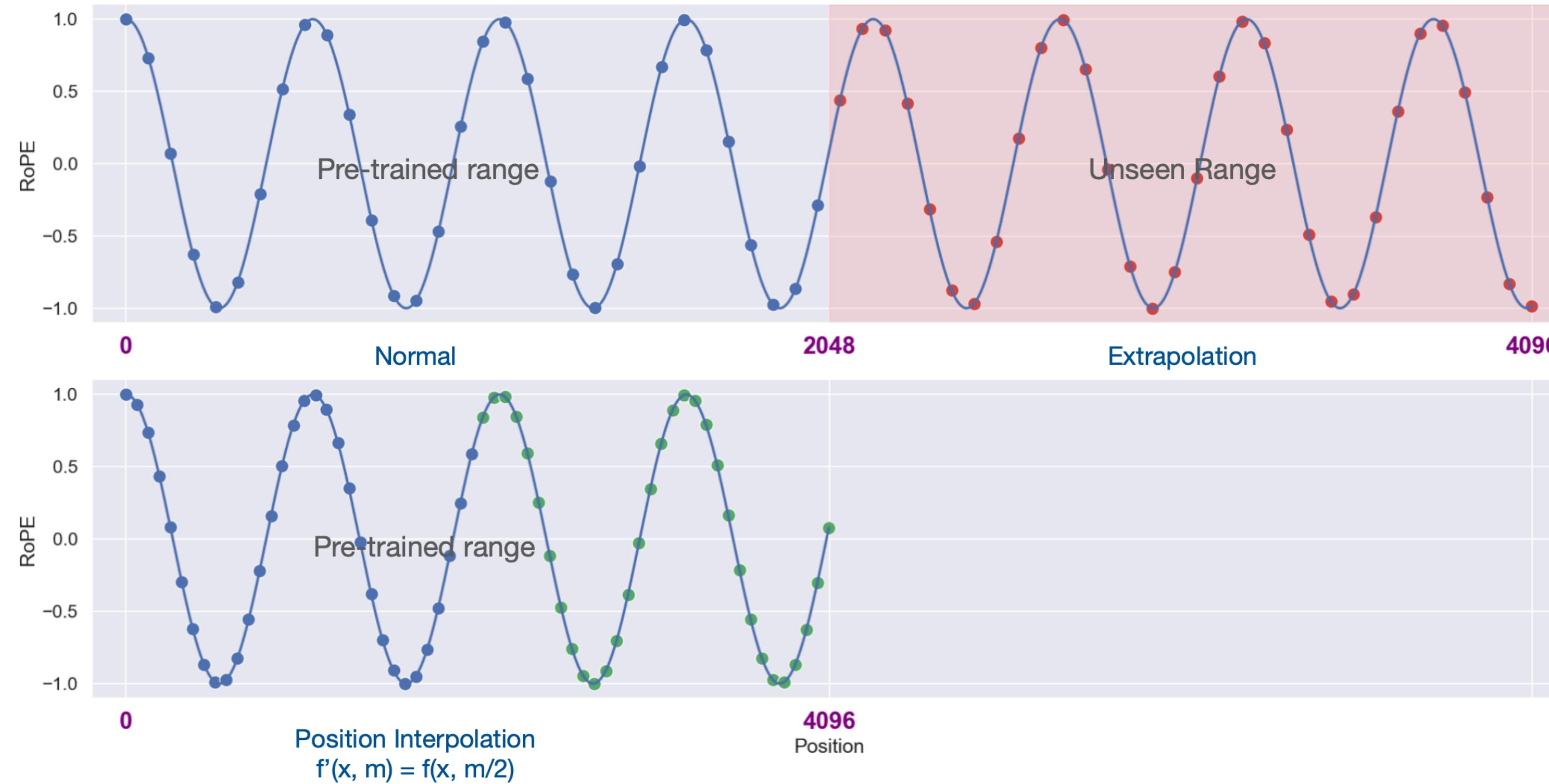
Таким образом, умножение занимает $O(d)$ операций вместо $O(d^2)$.

RoPE: Выводы

- RoPE позволяет учитывать относительное расположение токенов
- RoPE не добавляет обучаемых параметров
- RoPE очень эффективно считается
- RoPE очень популярен, используется в Llama, Mistral, PaLM, GPT-4, Falcon, Qwen2 и т.д.
- RoPE плохо работает при увеличении длины текста

Интерполяция RoPE

Экстраполяция не работает для RoPE, зато интерполяция – отлично



Интерполяция RoPE

FT – дообучение с экстраполяцией

PI – дообучение с интерполяцией

Size	Context Window	Method	Evaluation Context Window Size				
			2048	4096	8192	16384	32768
7B	2048	None	7.20	> 10 ³	> 10 ³	> 10 ³	> 10 ³
	8192	FT	7.21	7.34	7.69	-	-
7B	8192	PI	7.13	6.96	6.95	-	-
	16384	PI	7.11	6.93	6.82	6.83	-
7B	32768	PI	7.23	7.04	6.91	6.80	6.77
13B	2048	None	6.59	-	-	-	-
	8192	FT	6.56	6.57	6.69	-	-
13B	8192	PI	6.55	6.42	6.42	-	-
	16384	PI	6.56	6.42	6.31	6.32	-
13B	32768	PI	6.54	6.40	6.28	6.18	6.09
33B	2048	None	5.82	-	-	-	-
	8192	FT	5.88	5.99	6.21	-	-
33B	8192	PI	5.82	5.69	5.71	-	-
	16384	PI	5.87	5.74	5.67	5.68	-
65B	2048	None	5.49	-	-	-	-
	8192	PI	5.42	5.32	5.37	-	-

Attention with Linear Biases (ALiBi)

- Добавляем отрицательный сдвиг к каждому значению внимания
- Чем больше разница между позициями, тем меньше внимания остается

$$Attn = \text{softmax}\left(\frac{QK^T + M}{\sqrt{d}}\right)V$$

Обычный механизм внимания

$$M_{ij} = \begin{cases} 0, & i \leq j \\ -\infty, & i > j \end{cases}$$

ALiBi

$$M_{ij}^h = \begin{cases} m_h(i - j), & i \leq j \\ -\infty, & i > j \end{cases}$$

Для каждой головы свой $m_h = 2^{-\frac{h}{2}}$

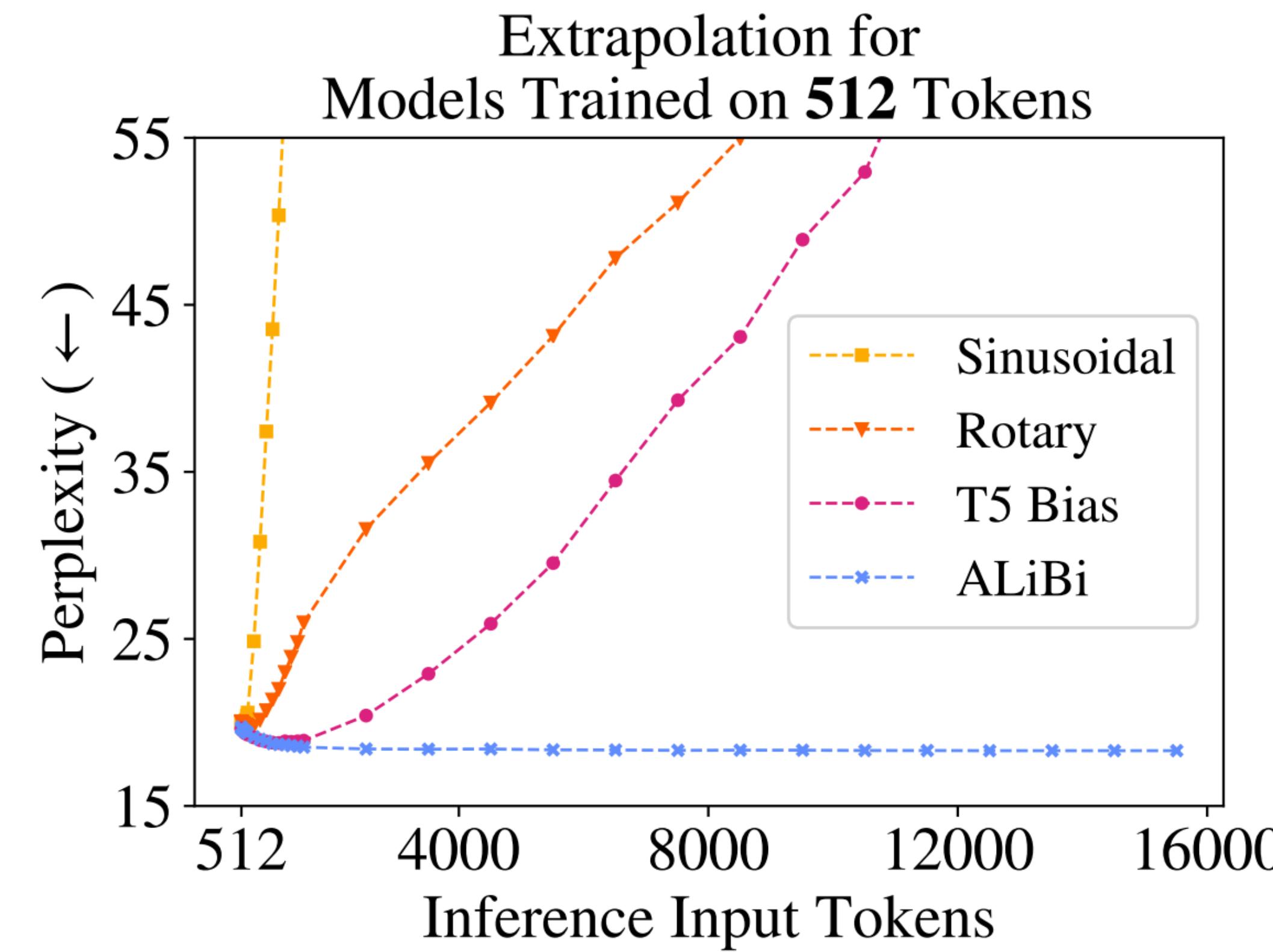
Attention with Linear Biases (ALiBi)

- Коэффициент m нужен для того, чтобы одни головы смотрели на локальную информацию, а другие – на глобальную
- ALiBi не усложняет модель, так как не добавляет обучающих параметров и дополнительных операций

$$\begin{array}{c} \begin{matrix} q_1 \cdot k_1 \\ q_2 \cdot k_1 & q_2 \cdot k_2 \\ q_3 \cdot k_1 & q_3 \cdot k_2 & q_3 \cdot k_3 \\ q_4 \cdot k_1 & q_4 \cdot k_2 & q_4 \cdot k_3 & q_4 \cdot k_4 \\ q_5 \cdot k_1 & q_5 \cdot k_2 & q_5 \cdot k_3 & q_5 \cdot k_4 & q_5 \cdot k_5 \end{matrix} + \begin{matrix} 0 \\ -1 & 0 \\ -2 & -1 & 0 \\ -3 & -2 & -1 & 0 \\ -4 & -3 & -2 & -1 & 0 \end{matrix} \cdot m \end{array}$$

ALiBi: Выводы

- ALiBi не усложняет модель, так как не добавляет обучающих параметров и дополнительных операций
- ALiBi очень хорошо приспосабливается к увеличению длины текста
- ALiBi используется в BLOOM, Claude

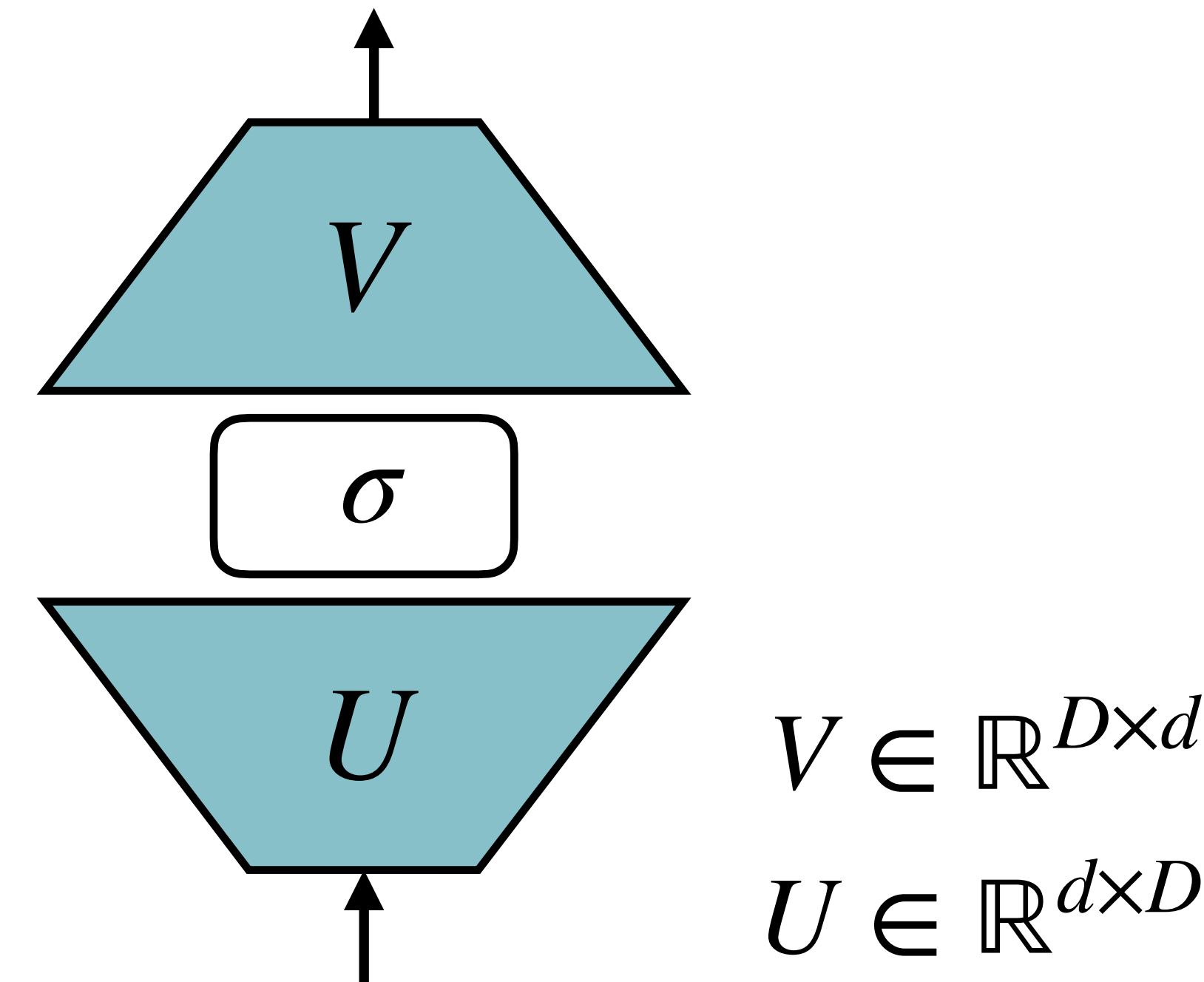


Feed forward network

Feed forward network

- FFN состоит из двух линейных слоев с нелинейностью между ними
- Первый слой повышает размерность, а второй понижает обратно

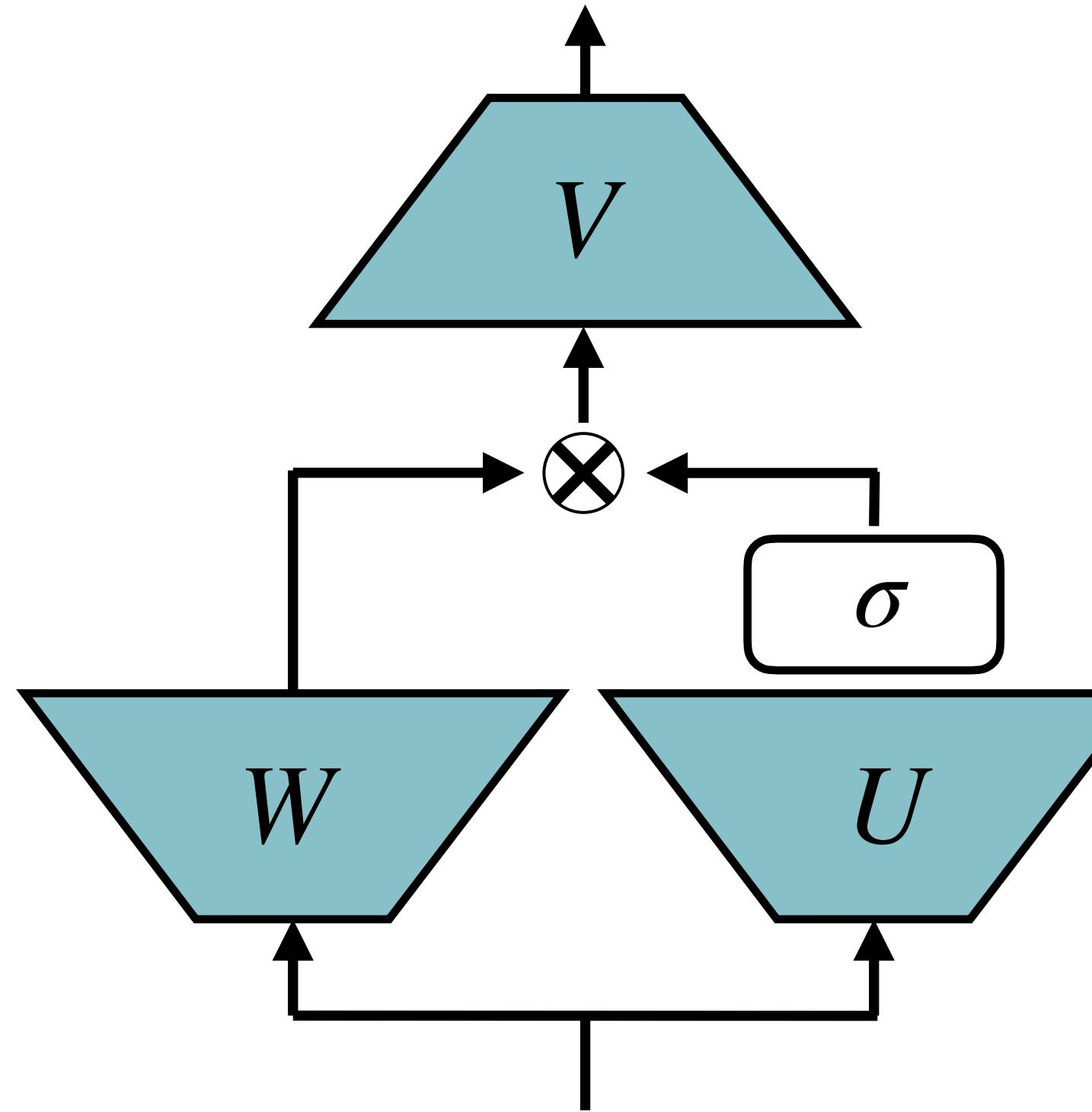
$$FFN(x) = \sigma(xU + b_u)V + b_v$$



Gated Linear Unit (GLU)

- GLU добавляет дополнительный линейный слой W , выходы которого умножаются на выходы U после активации
- W играет роль фильтра, отсеивающего ненужные компоненты

$$FFN_{GLU}(x) = (\sigma(xU + b_u) \otimes (xW + b_w))V + b_v$$



$$V \in \mathbb{R}^{D \times d}$$

$$U \in \mathbb{R}^{d \times D}$$

$$W \in \mathbb{R}^{d \times D}$$

Gated Linear Unit (GLU)

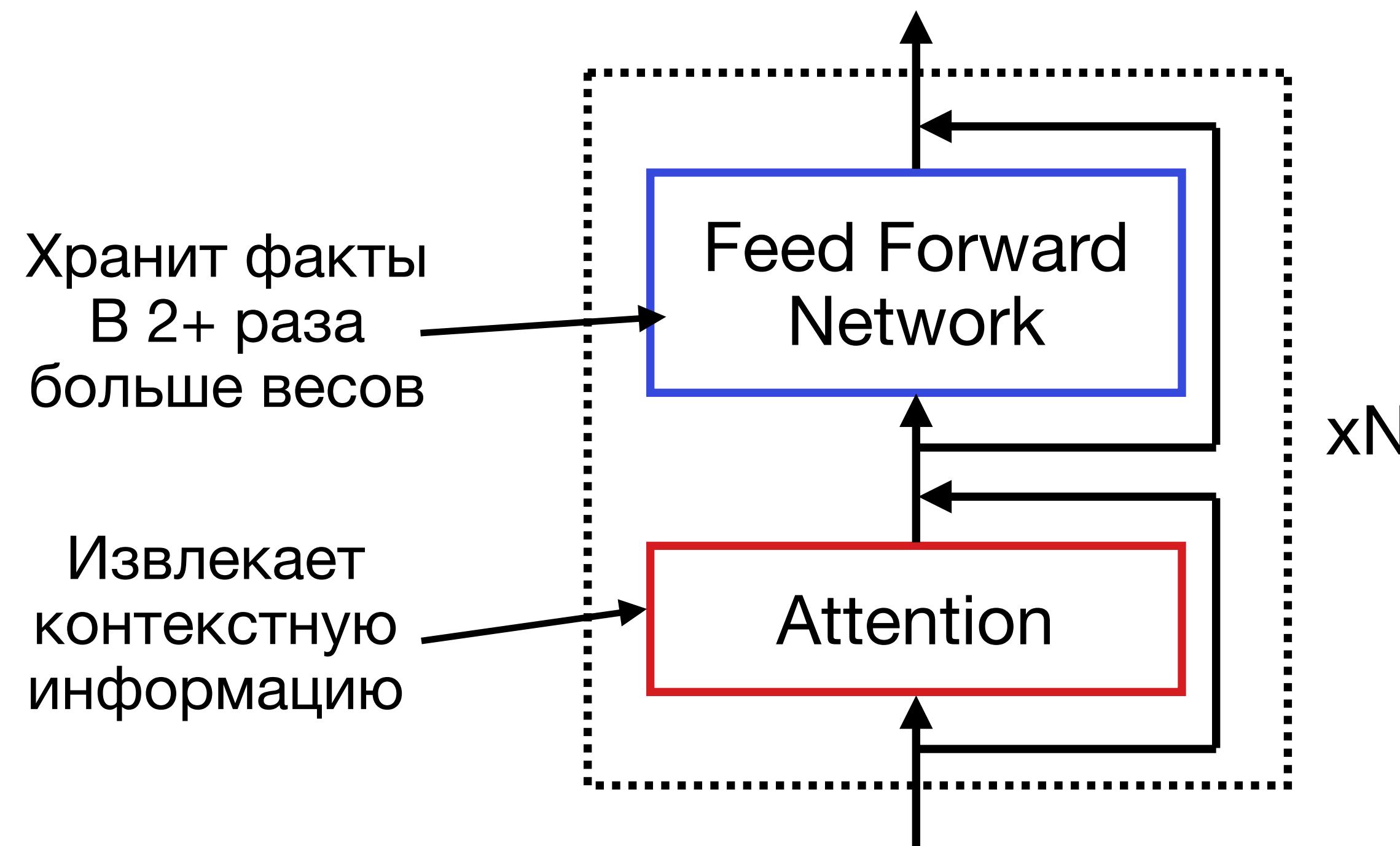
- GLU можно использовать с разными активациями
- Чаще всего используется GLU с активацией Swish (очень похожа на ReLU)
- GLU применяется вместо FFN в Llama, Mistral, Gemma, Qwen2

$$FFN_{GLU}(x) = (\sigma(xU + b_u) \otimes (xW + b_w))V + b_v$$

	Score	CoLA	SST-2	MRPC	MRPC	STSB	STSB	QQP	QQP	MNLI _m	MNLI _{mm}	QNLI	RTE
	Average	MCC	Acc	F1	Acc	PCC	SCC	F1	Acc	Acc	Acc	Acc	Acc
FFN _{ReLU}	83.80	51.32	94.04	93.08	90.20	89.64	89.42	89.01	91.75	85.83	86.42	92.81	80.14
FFN _{GELU}	83.86	53.48	94.04	92.81	90.20	89.69	89.49	88.63	91.62	85.89	86.13	92.39	80.51
FFN _{Swish}	83.60	49.79	93.69	92.31	89.46	89.20	88.98	88.84	91.67	85.22	85.02	92.33	81.23
FFN _{GLU}	84.20	49.16	94.27	92.39	89.46	89.46	89.35	88.79	91.62	86.36	86.18	92.92	84.12
FFN _{GEGLU}	84.12	53.65	93.92	92.68	89.71	90.26	90.13	89.11	91.85	86.15	86.17	92.81	79.42
FFN _{Bilinear}	83.79	51.02	94.38	92.28	89.46	90.06	89.84	88.95	91.69	86.90	87.08	92.92	81.95
FFN _{SwiGLU}	84.36	51.59	93.92	92.23	88.97	90.32	90.13	89.14	91.87	86.45	86.47	92.93	83.39
FFN _{ReGLU}	84.67	56.16	94.38	92.06	89.22	89.97	89.85	88.86	91.72	86.20	86.40	92.68	81.59

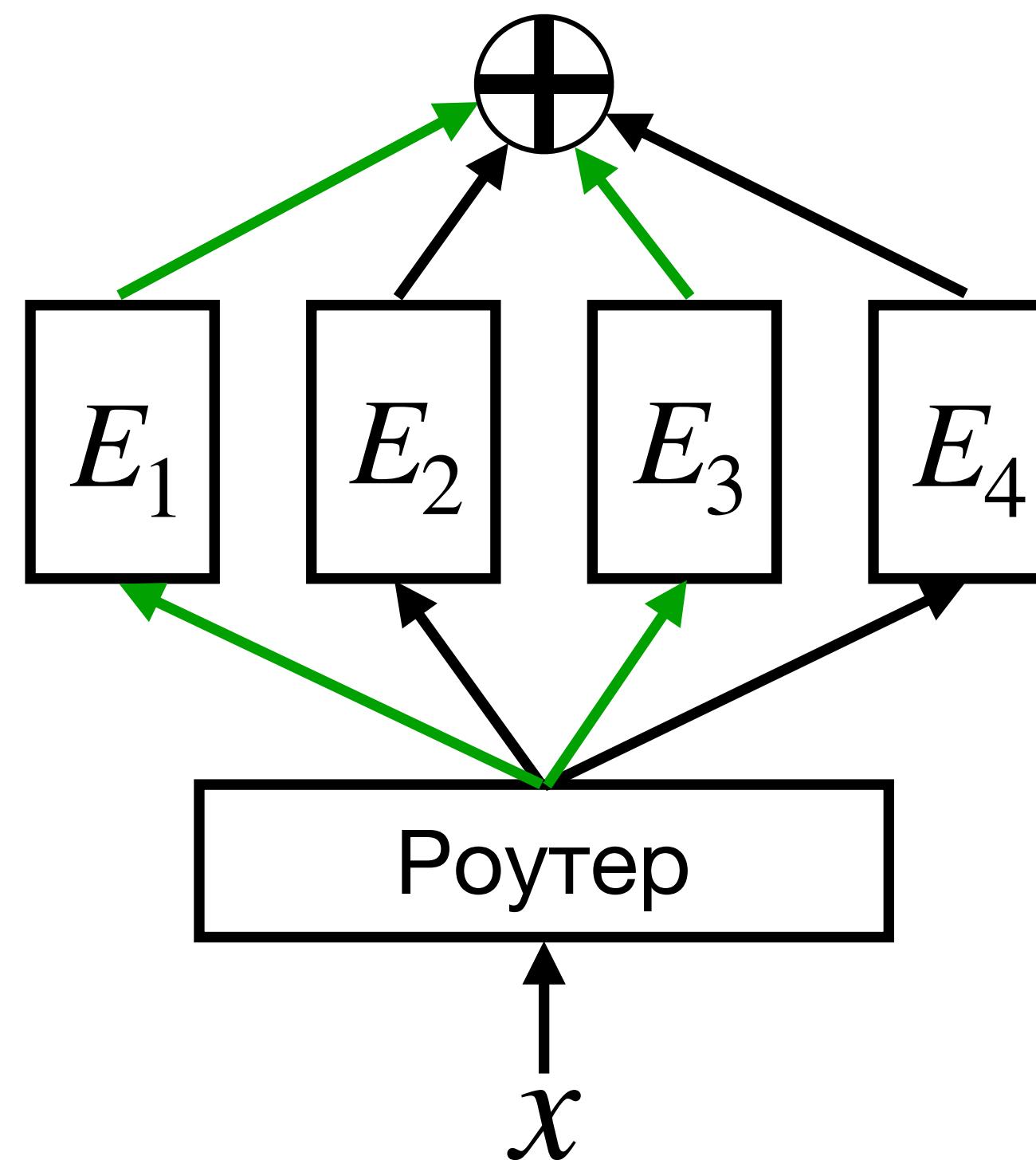
Mixture of Experts (MoE)

- Вместо только, чтобы использовать одну большую FFN, можно обучить набор небольших специализированных FFN
- Таким образом, можно ускорить применение модели и уменьшить затраты по памяти



Mixture of Experts (MoE)

- MoE заменяет слой FFN и состоит из роутера и экспертов
- Роутер определяет, какому эксперту отдать какой **токен**
- Эксперты имеют вид FFN и обрабатывают токены независимо
- Обычно, каждый токен обрабатывают несколько экспертов одновременно



Роутер

- Роутер решает стандартную задачу классификации и возвращает вероятность для каждого эксперта
- Роутер очень простой и часто состоит из одного линейного слоя

$$p(E | x) = \text{softmax}(R(x)) = \text{softmax}(xW)$$

Роутер

- Роутер решает стандартную задачу классификации и возвращает вероятность для каждого эксперта
- Роутер очень простой и часто состоит из одного линейного слоя

$$p(E | x) = \text{softmax}(R(x)) = \text{softmax}(xW)$$

- В **Sparse MoE** вероятности для большинства экспертов зануляются
- Это позволяет применять модель быстрее

$$p(E | x) = \text{softmax}(\text{Top}K(xW)) \quad K = 2$$

Агрегация выходов экспертов

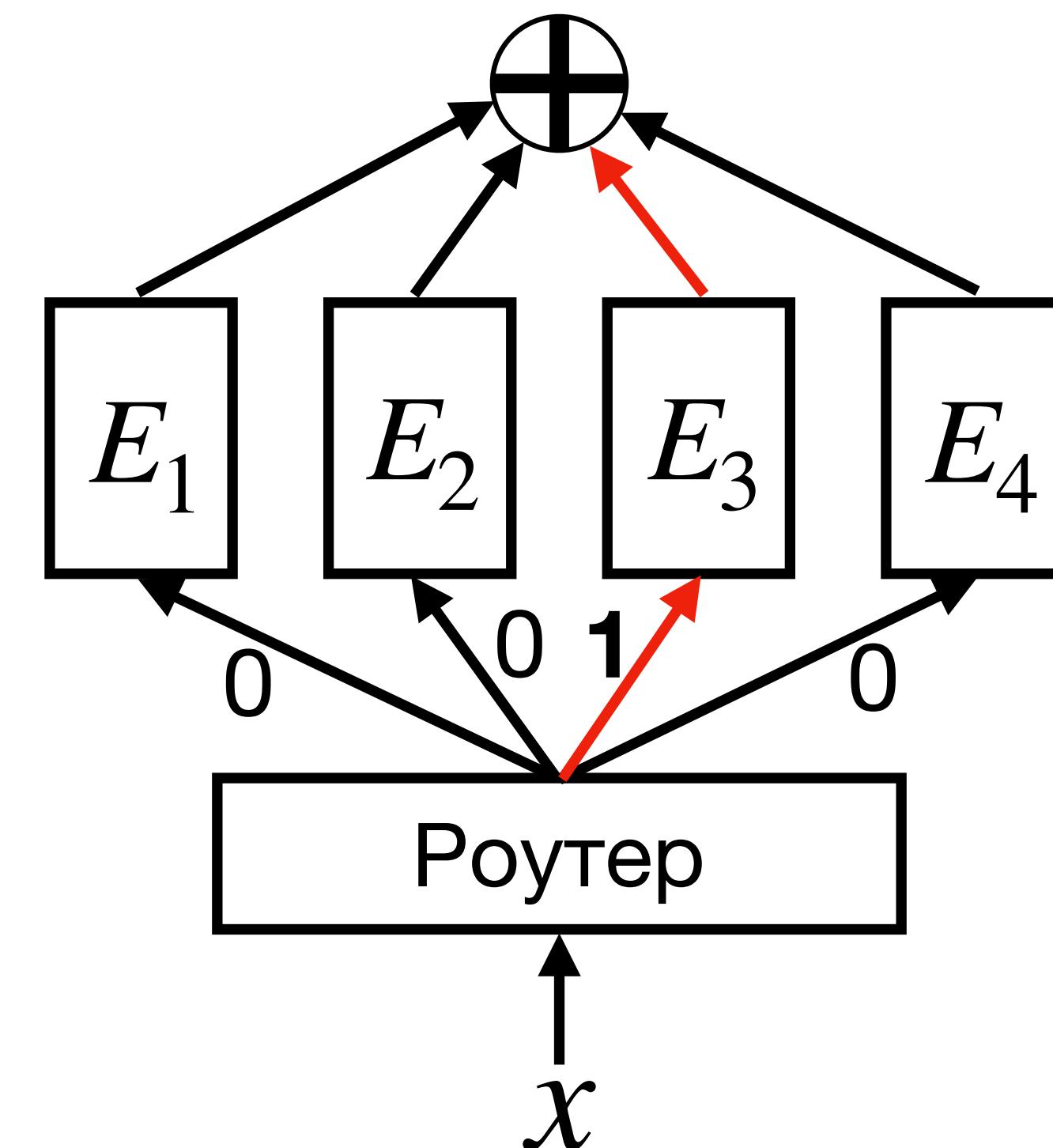
Выходы экспертов суммируются с вероятностями роутера

$$x' = \sum_{i=1}^n p(E_i | x) E_i(x),$$

где n – число экспертов.

Обучение MoE

- Во время обучения роутер учится вместе с экспертами
- Часто роутер выучивается отдавать всю вероятность одному эксперту
- Для избежания этого можно
 - Зашумлять вероятности
 - Добавлять регуляризацию



Зашумление вероятностей

- Добавляя шум в вероятности, мы уменьшаем вероятность выбора одного эксперта
- Остальные эксперты будут получать градиенты и начнут учиться

$$p(E | x) = (1 - \alpha) \cdot \text{softmax}(R(x)) + \alpha \cdot \varepsilon,$$

где $\varepsilon \in [0,1]^n$ и $\sum_{i=0}^{n-1} \varepsilon_i = 1$.

- Если α слишком маленькое, то проблема не исчезнет
- Если α слишком большое, то эксперты будут учить одно и то же

Регуляризация

Для балансировки нагрузки по экспертам добавим к лоссу такую функцию

$$L = \alpha \cdot n \cdot \sum_{i=1}^n f_i \cdot P_i,$$

где f_i – доля токенов, отданная i -му эксперту

$$f_i = \frac{1}{l} \sum_{k=1}^l [\operatorname{argmax}\{p(E|x_k)\} = i]$$

и P_i – средняя вероятность для i -го эксперта

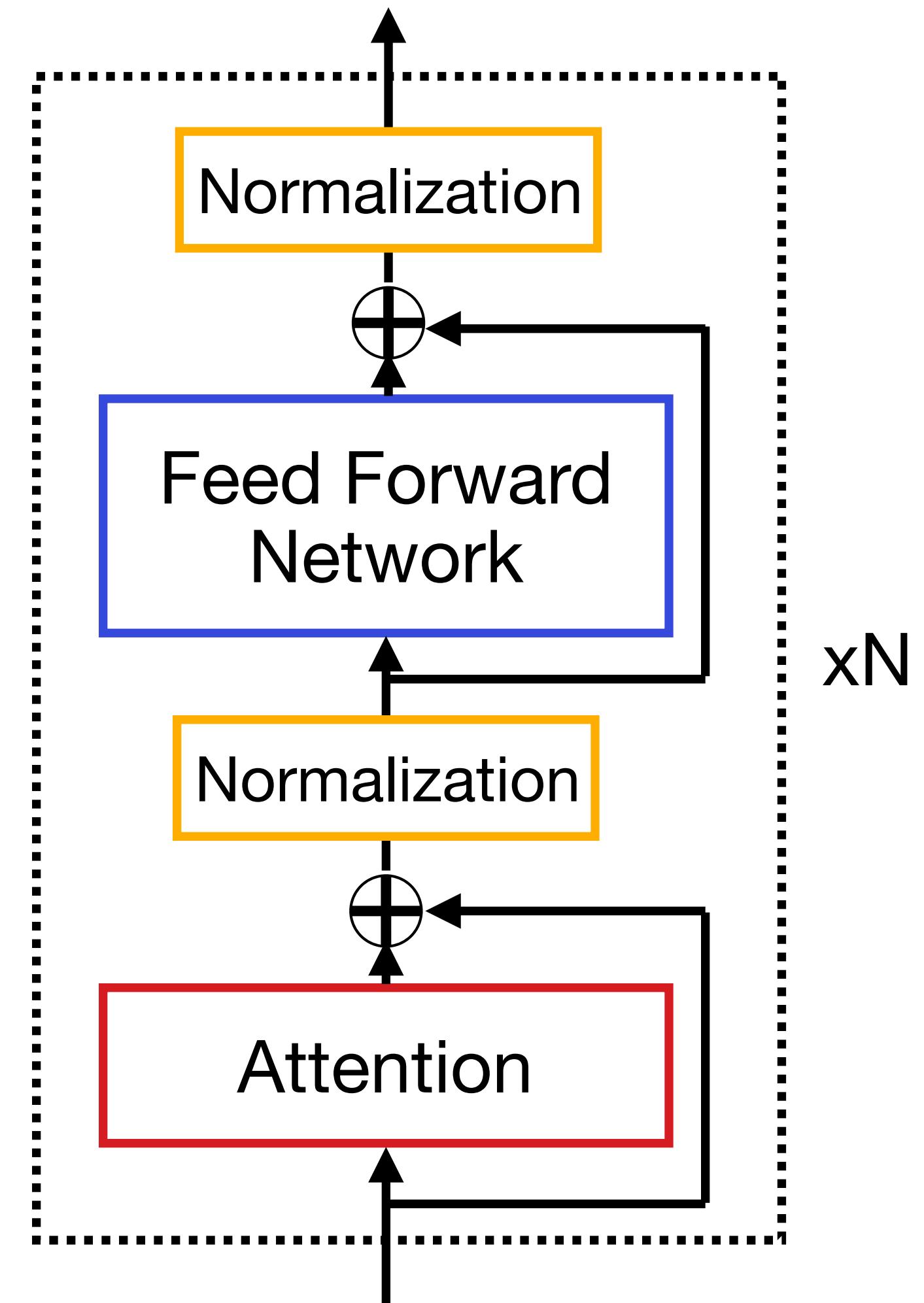
$$P_i = \frac{1}{l} \sum_{k=1}^l p(E_i|x_k)$$

Минимум функции достигается, когда $f_i = P_i = \frac{1}{n}$. Это нам и надо.

Нормализация

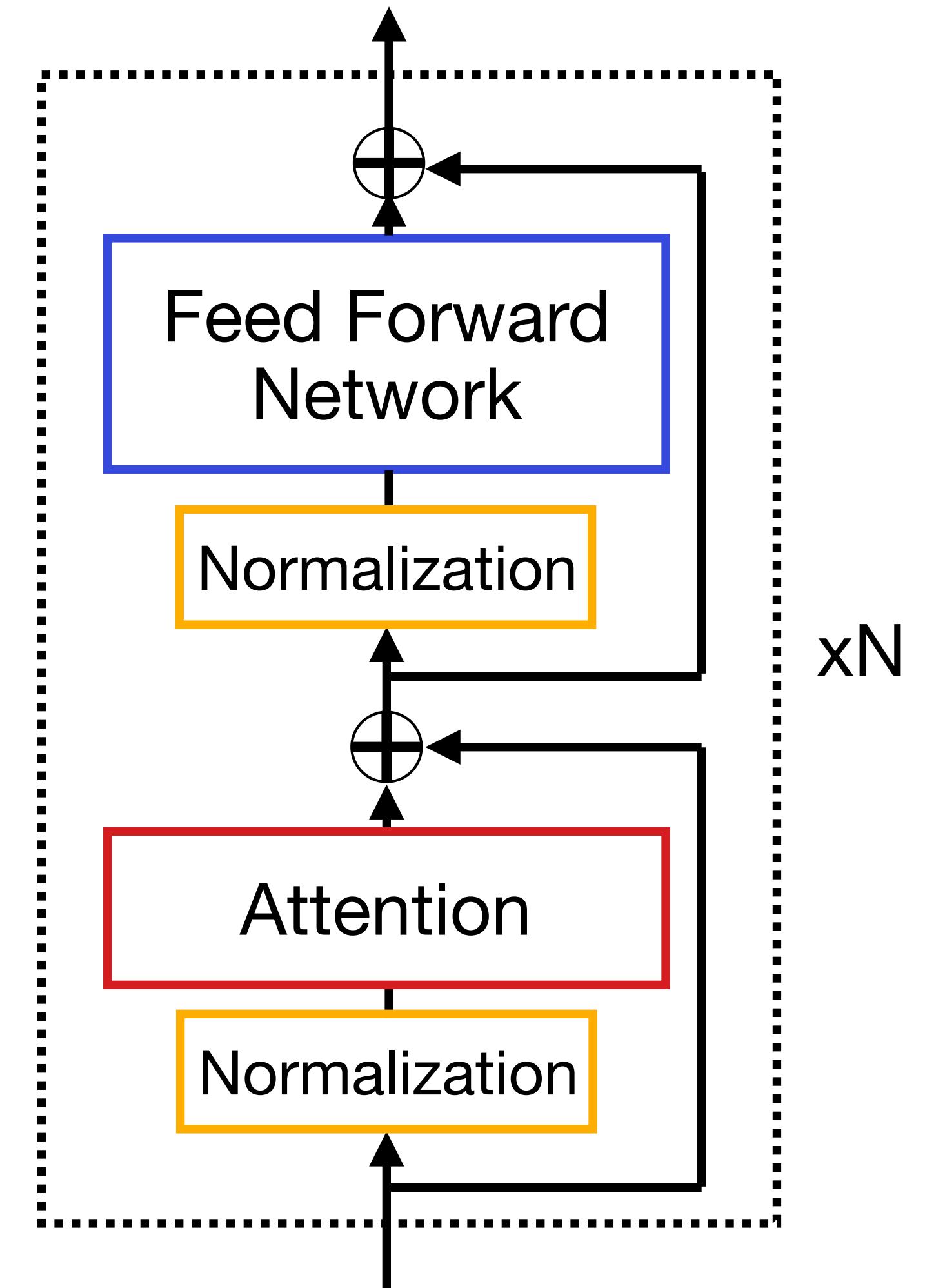
Нормализация

- Нормализация стабилизирует обучение, не позволяя выходам слоев расходиться
- Однако из-за нормализации градиенты начинают затухать сразу после начала обучения, и модель вообще не учится
- Именно поэтому используется warm-up



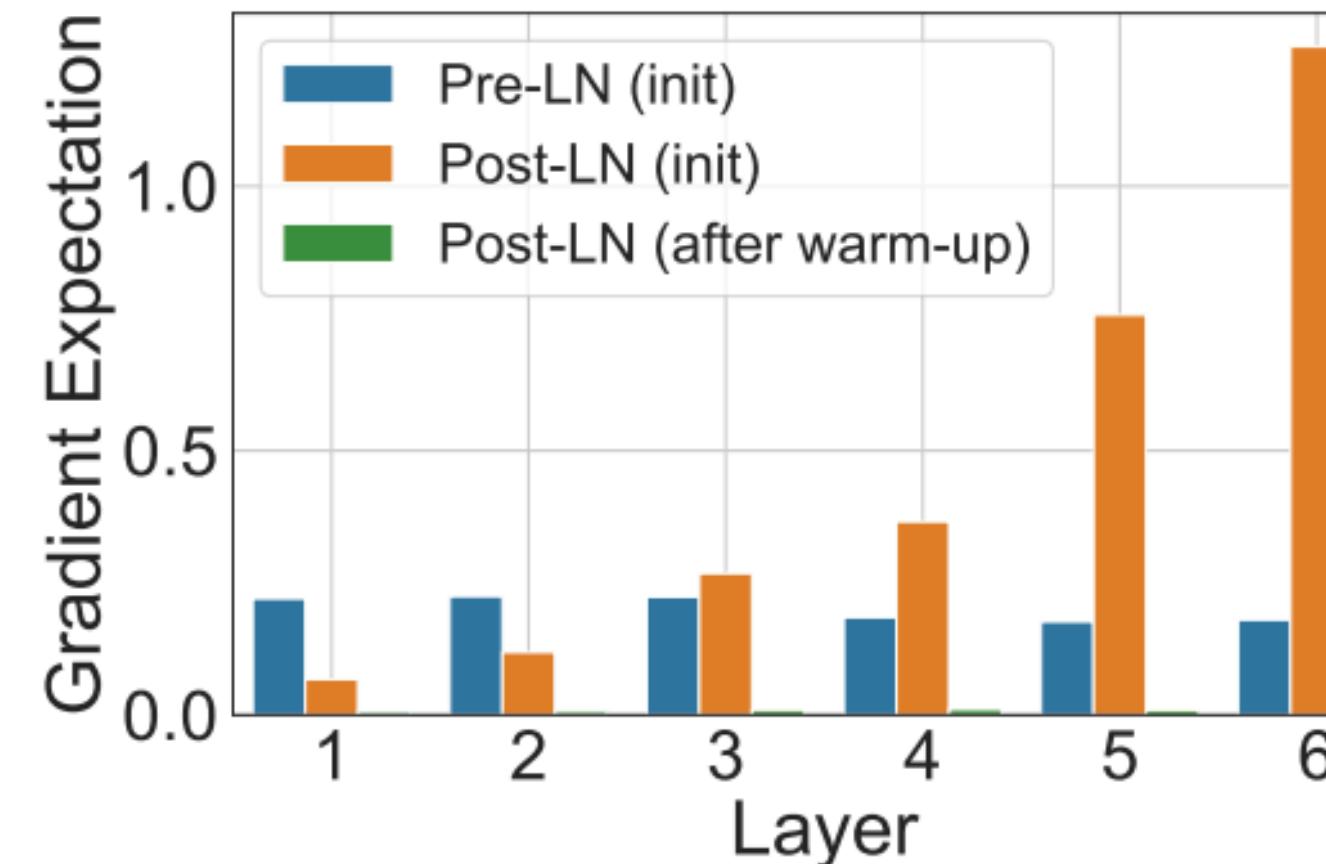
Pre-normalization

- Можно нормировать не все тензоры, а только входы для слоев внимания и FFN
- Такой подход называется **Pre-LN Transformer**
- Стандартный подход – **Post-LN**
- Градиенты теперь ведут себя стабильнее и Трансформер можно учить без warm-up

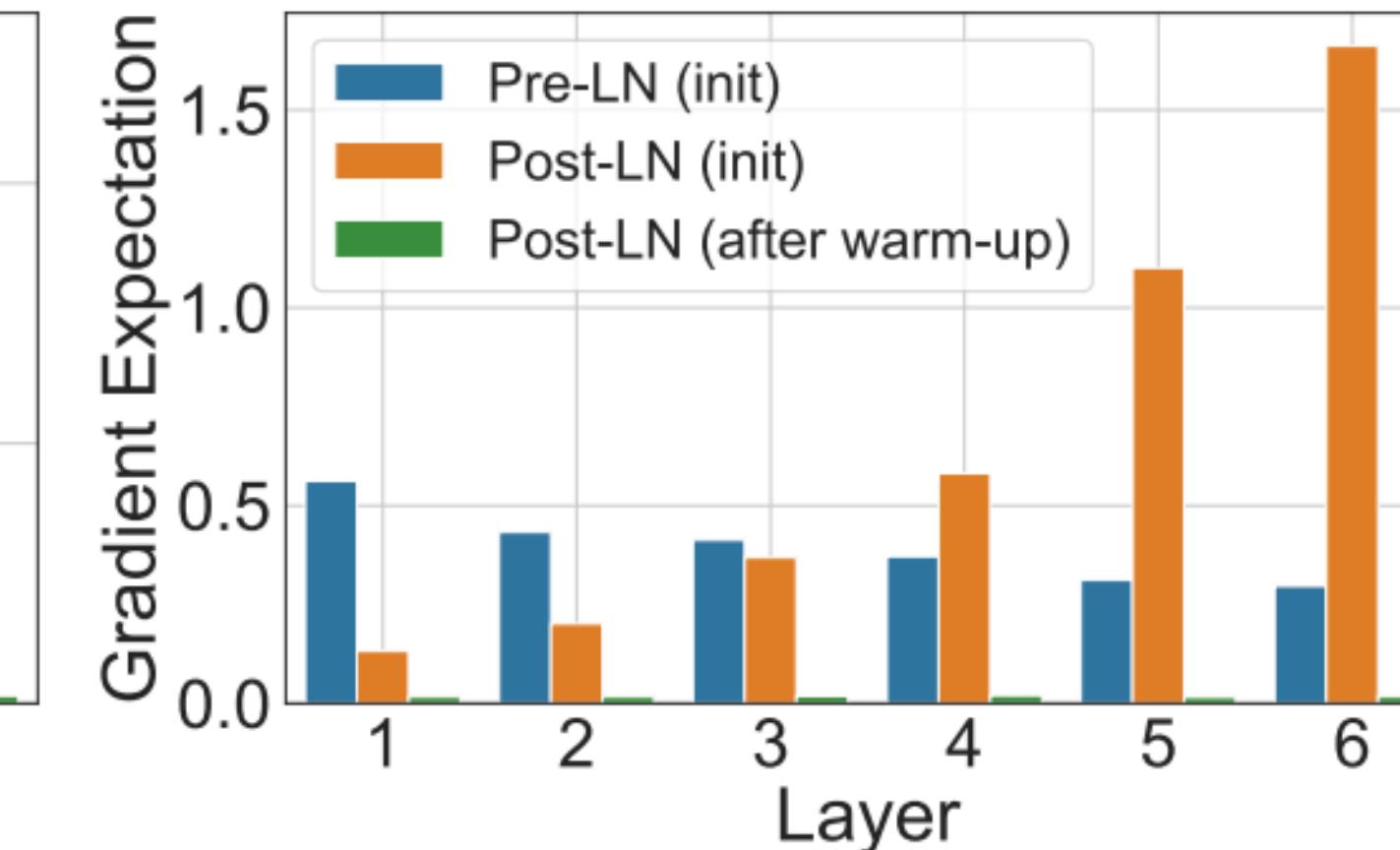


Поведение градиентов

- Градиенты у **Post-LN** подхода без warm-up увеличиваются с увеличением номера слоя
- У **Pre-LN** такого не происходит
- Pre-LN** часто используется для обучения глубоких Трансформеров: Mistral, BLOOM, Falcon, Qwen2 и т.д.
- Для небольших Трансформеров качество **Pre-LN** обычно хуже

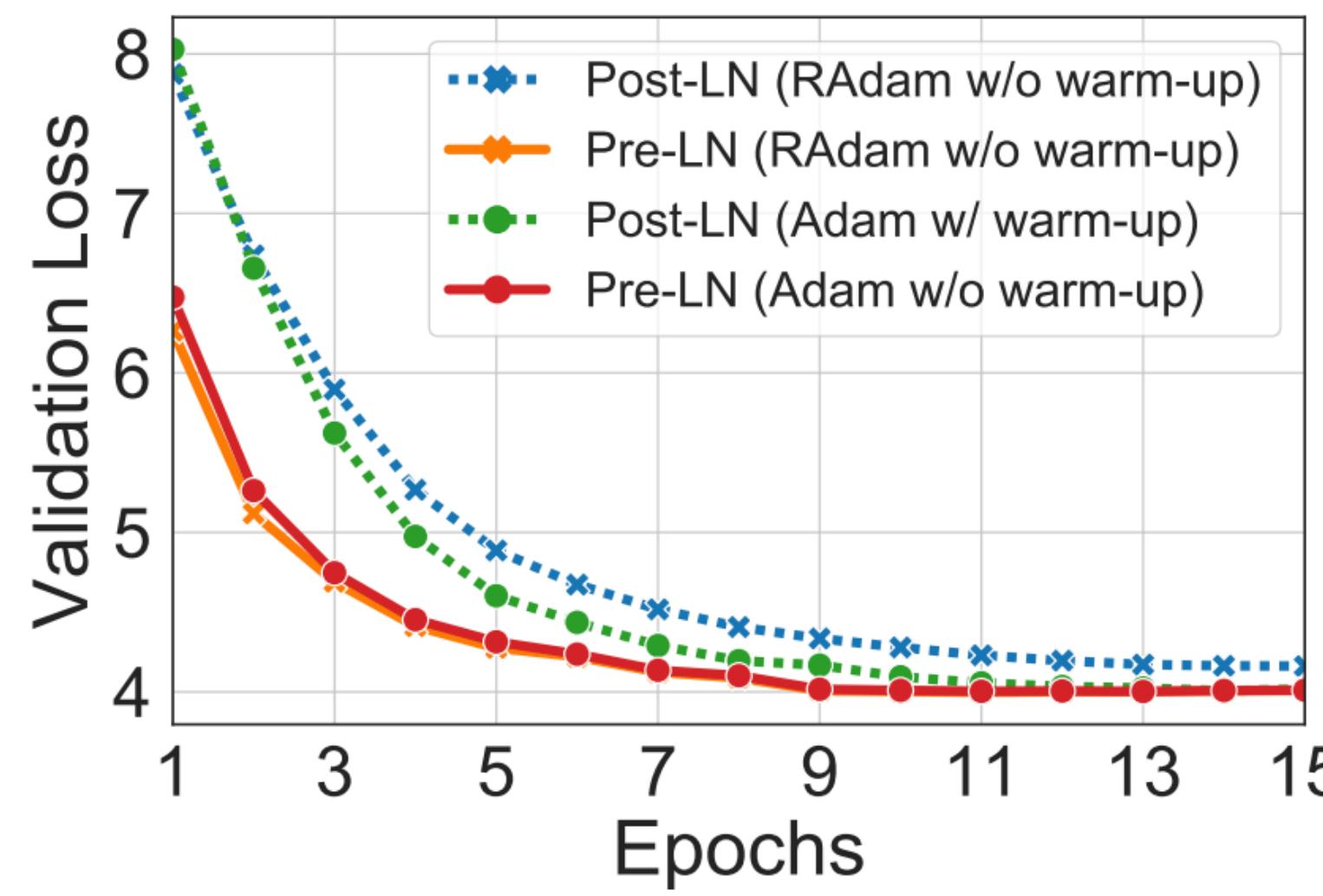


(a) W^1 in the FFN sub-layers

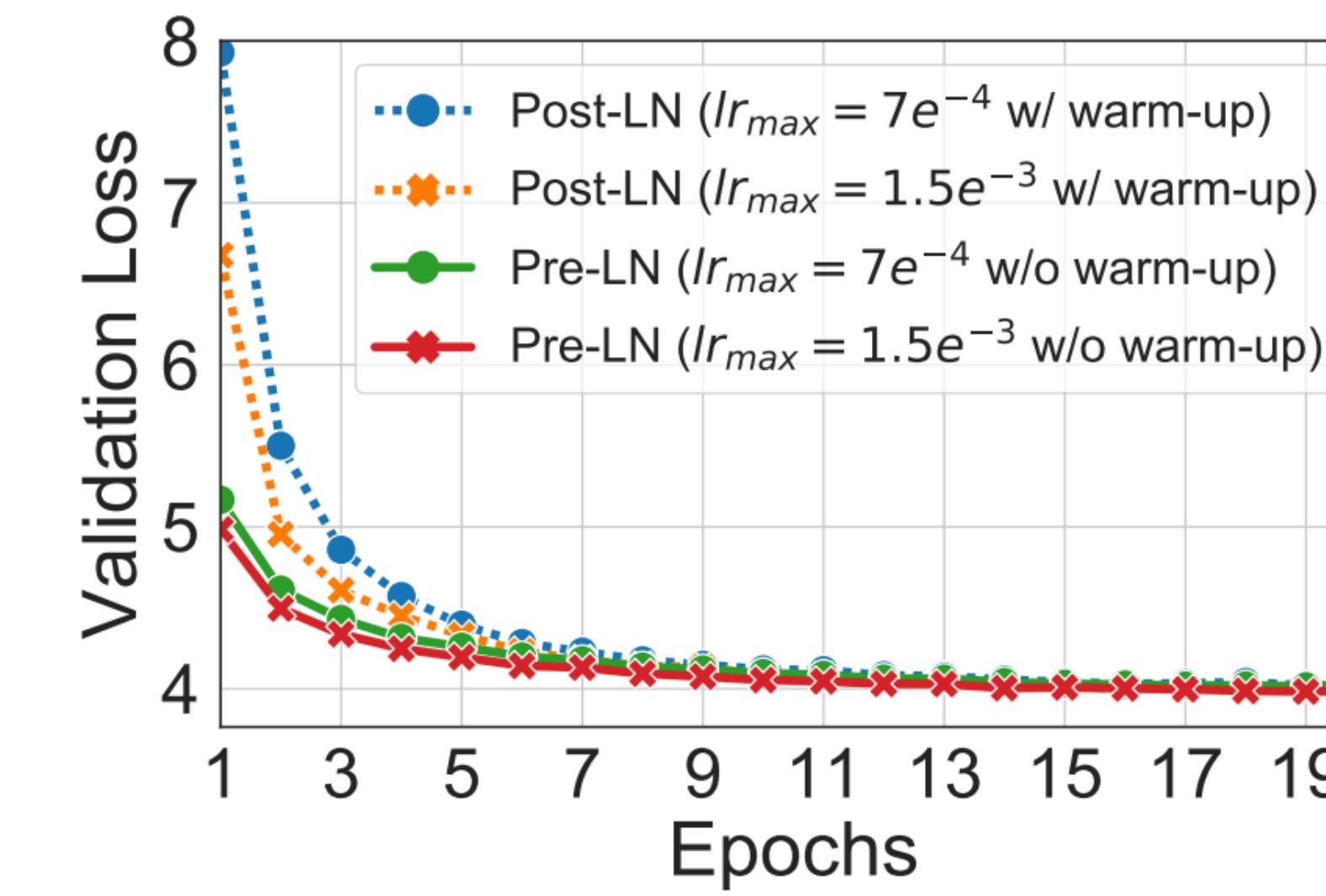


(b) W^2 in the FFN sub-layers

Pre-normalization на практике



(a) Validation Loss (IWSLT)



(c) Validation Loss (WMT)

Layer Normalization

По умолчанию в Трансформерах используется Layer Normalization

$$y = \frac{x - \mathbb{E}(x)}{\sqrt{Var(x) + \epsilon}} \cdot \gamma + \beta$$

То есть считается выборочное среднее и дисперсия

$$\mathbb{E}(x) = \frac{1}{n} \sum_{i=1}^n x \quad Var(x) = \frac{1}{n-1} \sum_{i=1}^n (x_i - \mathbb{E}(x))^2$$

Root Mean Square (RMS) Norm

По умолчанию в Трансформерах используется Layer Normalization

$$y = \frac{x - \mathbb{E}(x)}{\sqrt{Var(x) + \epsilon}} \odot \gamma + \beta$$

То есть считается выборочное среднее и дисперсия

$$\mathbb{E}(x) = \frac{1}{n} \sum_{i=1}^n x \quad Var(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \mathbb{E}(x))^2$$

Оказывается, можно считать среднее нулевым и не использовать его.

$$y = \frac{x}{\sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2 + \epsilon}} \odot \gamma \quad \gamma \in \mathbb{R}^d$$

Root Mean Square (RMS) Norm

- RMSNorm часто работает даже лучше, чем Layer Norm
- RMSNorm считается быстрее на ~25%, так как не надо оценивать среднее по тензору
- Более того, для оценки дисперсии можно брать **не все** элементы тензора
- RMSNorm используется в современных моделях: Mistral, Llama, Qwen2 и т.д.

Model	Test14	Test17	Time
Baseline	21.7	23.4	$399 \pm 3.40\text{s}$
LayerNorm	22.6	23.6	$665 \pm 32.5\text{s}$
L2-Norm	20.7	22.0	$482 \pm 19.7\text{s}$
RMSNorm	22.4	23.7	$501 \pm 11.8\text{s}$ (24.7%)
<i>p</i> RMSNorm	22.6	23.1	$493 \pm 10.7\text{s}$ (25.9%)

Table 2: SacreBLEU score on newstest2014 (Test14) and newstest2017 (Test17) for RNNSearch using Tensorflow-version Nematus. “*Time*”: the time in second per 1k training steps. We set p to 6.25%. We highlight the best results in bold, and show the speedup of RMSNorm against Layer-Norm in bracket.

Baseline – без нормализации