Reformer

эффективный трансформер

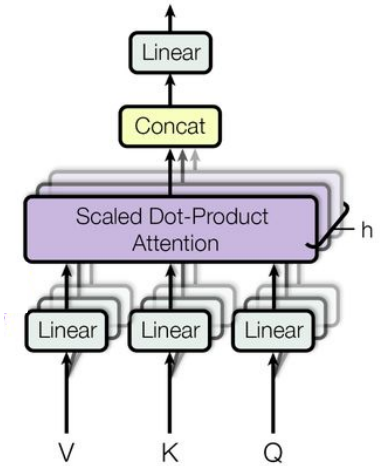
Аржанцев Алексей
ПМИ 192

Проблемы стандартного трансформера

Некоторые проблемы:

1. O(L²) операций на attention-слое.
2. Затраты памяти растут линейно с числом слоев (т.к. надо хранить значения на всех слоях).
3. Feed forward слой требует очень много памяти.

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



Решение проблем стандартного трансформера

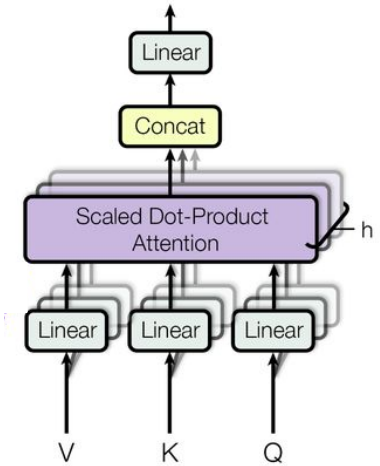
Решения проблем:

1. Используем хэширование с учетом местоположения!

2. Используем обратимые слои!

3. Разобьем слой на чанки и будем работать независимо в каждом чанке!

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



Обо всем по порядку

1. Хэширование с учетом местоположения
2. Обратимые слои
3. Разбиение на чанки

Хэширование с учетом местоположения

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

Матрица QK^T имеет размер (batch_size x L x L). Хотим избавиться от квадрата.

Рассмотрим случай когда Q = K. Как показывает практика это не сильно ухудшает результаты трансформера (!?)

Идея: хорошим приближением будет рассмотреть k ближайших. Как сделать это быстро?

Хэширование с учетом местоположения

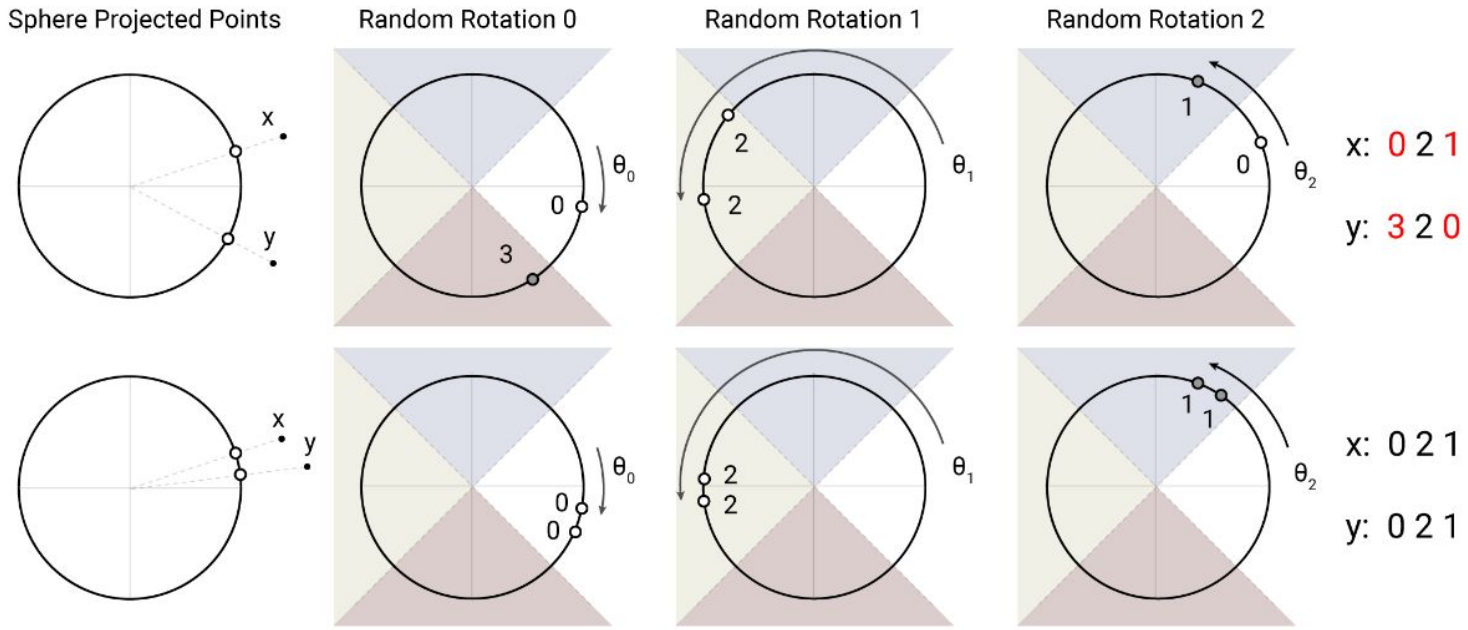
Основная идея: функция $h(x) : \mathbb{R}^n \to \{1, 2, \ldots, b\}$, такая что близкие векторы с большой вероятность получают один и тот же хэш, а далекие разный.

В нашей задаче важно первое требование + что каждое значение хэша должно получить примерно равное число векторов.

Используем случайные проеции! Выберем случайную матрицу $R \in \mathbb{R}^{d_k \times b/2}$, тогда

$$h(x) = \arg\max([Rx; -Rx])$$

Хэширование с учетом местоположения (наглядно)



Хэширование с учетом местоположения

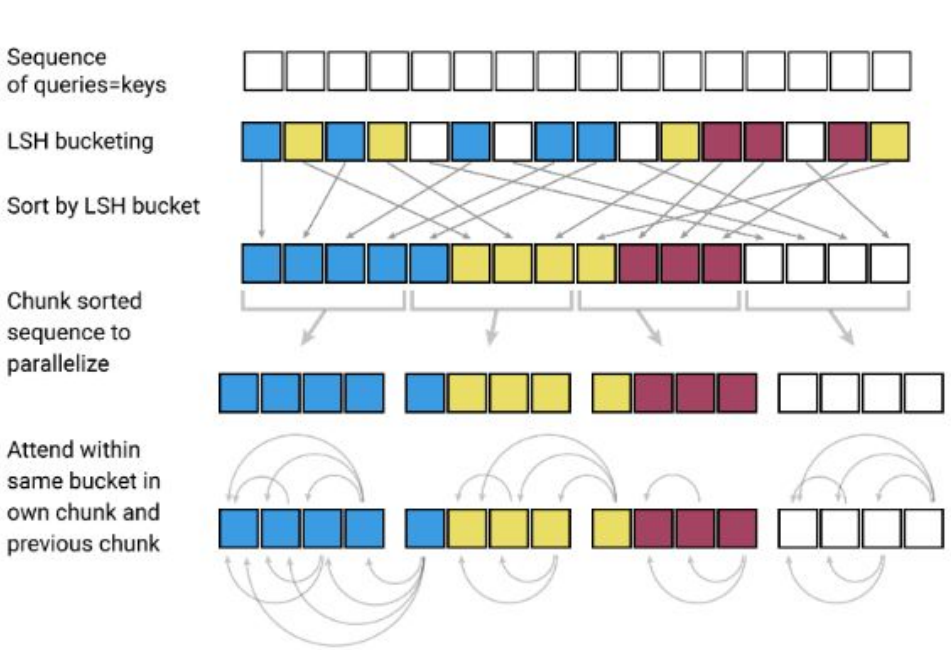
Как мы теперь используем эту функцию?

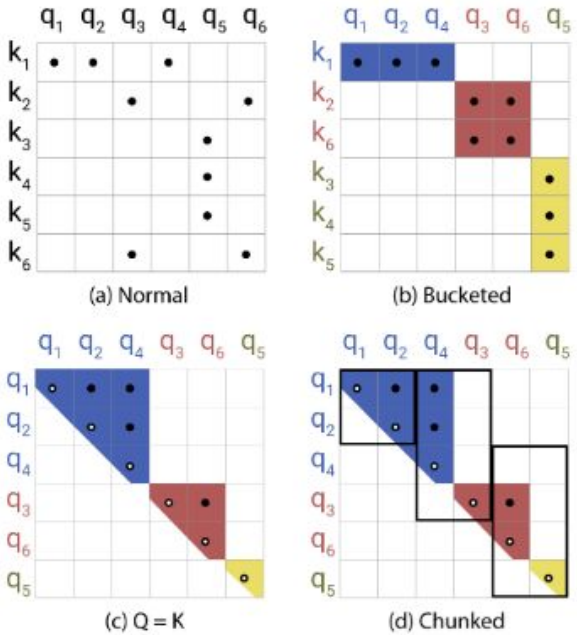
Скажем, что каждый вектор может смотреть только на векторы с таким же хэшэм.

Чтобы избежать ситуации, когда какого-то значения получилось слишком много, дополнительно разобьем на m чанков и разрешим смотреть только на свой и на предыдущий чанк.

Возьмем $m = \frac{2L}{n_{buckets}}$ чтобы вероятность, что какого-то значения будет больше чем размер чанка была крайне мала.

Хэширование с учетом местоположения (наглядно)





Хэширование с учетом местоположения
дополнительные идеи

Идея 1:

Чтобы уменьшить вероятность, что близкие векторы получат разные хэши, можно сделать несколько раундов.

$$\mathcal{P}_i = \bigcup_{r=1}^{n_{rounds}} \mathcal{P}_i^{(r)} \qquad \text{where } \mathcal{P}_i^{(r)} = \left\{ j : h^{(r)}(q_i) = h^{(r)}(q_j) \right\}$$

Идея 2:

Все еще можно делать masked-attention. Достаточно переставить строчки и столбики в маске так же как мы переставили наши слова.

Результаты

Память и сложность вычислений

Attention Type	Memory Complexity	Time Complexity
Scaled Dot-Product	$\max(bn_hld_k, bn_hl^2)$	$\max(bn_hld_k, bn_hl^2)$
Memory-Efficient	$\max(bn_hld_k, bn_hl^2)$	$\max(bn_hld_k, bn_hl^2)$
LSH Attention	$\max(bn_hld_k, bn_hln_r(4l/n_c)^2)$	$\max(bn_hld_k, bn_hn_rl(4l/n_c)^2)$

Точность

Train \ Eval	Full Attention	LSH-8	LSH-4	LSH-2	LSH-1
Full Attention	100%	94.8%	92.5%	76.9%	52.5%
LSH-4	0.8%	100%	99.9%	99.4%	91.9%
LSH-2	0.8%	100%	99.9%	98.1%	86.8%
LSH-1	0.8%	99.9%	99.6%	94.8%	77.9%

Обо всем по порядку

1. Хэширование с учетом местоположения
2. Обратимые слои
3. Разбиение на чанки

Обратимые слои

Хотим сделать так, чтобы значения каждого слоя можно было восстановить по значениям со следующего слоя.

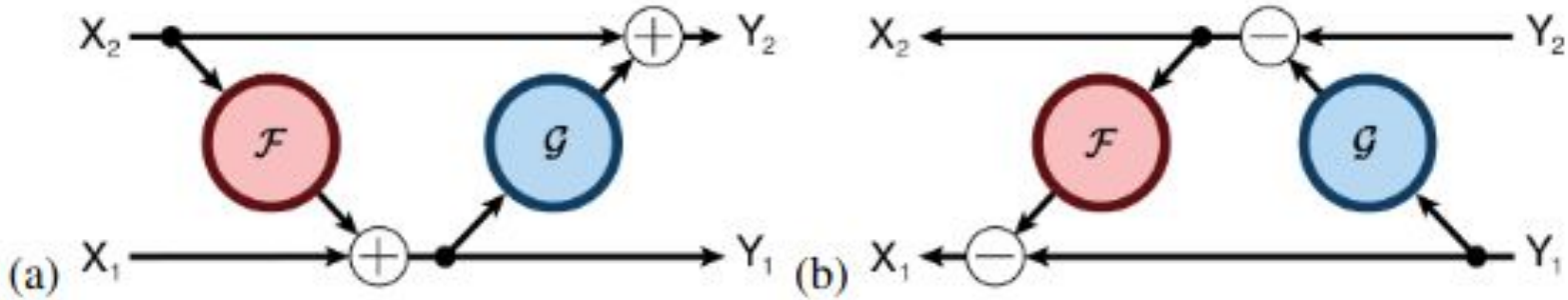
Общая идея такая: разбиваем ввод и вывод на 2 части. При проходе вперед делаем следующее преобразование.

$$y_1 = x_1 + F(x_2) \qquad y_2 = x_2 + G(y_1)$$

Тогда при проходе назад мы сможем восстановить результат!

$$x_2 = y_2 - G(y_1) \qquad x_1 = y_1 - F(x_2)$$

Обратимые слои
(наглядно)



Обратимые слои

В нашем случае

$$Y_1 = X_1 + \text{Attention}(X_2) \qquad Y_2 = X_2 + \text{FeedForward}(Y_1)$$

Эксперименты показывают, что такой трансформер показывает результат не хуже классического, при равном количестве параметров.

В качестве размеров длин x1 и x2 можно взять размерность одного эмбеддинга.

Обо всем по порядку

1. Хэширование с учетом местоположения
2. Обратимые слои
3. Разбиение на чанки

Разбиение на чанки

Так как FeedForward выполняется независимо, можно разбить вычисления на чанки и сэкономить по потреблению памяти.

$$Y_2 = \left[Y_2^{(1)}; \ldots; Y_2^{(c)}\right] = \left[X_2^{(1)} + \text{FeedForward}(Y_1^{(1)}); \ldots; X_2^{(c)} + \text{FeedForward}(Y_1^{(c)})\right]$$

Итоги

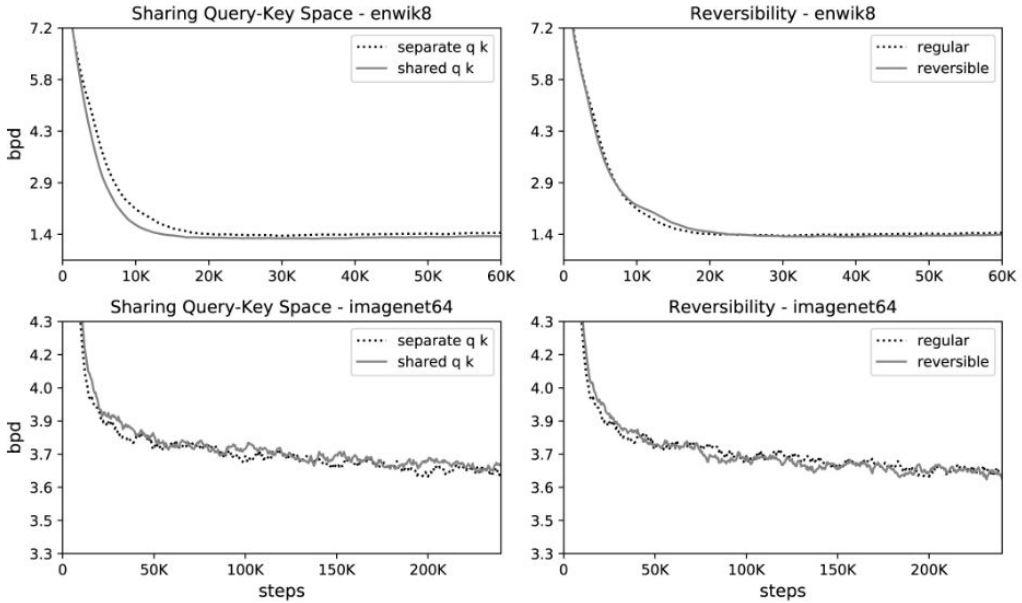
После всех наших стараний мы получили модель, потребление памяти в которой не зависит ни от глубины сети, ни от размерности представлений в ff-слое.

Также во времени выполнения L^2 поменялось на Lnr.

Model Type	Memory Complexity	Time Complexity
Transformer	$\max(bld_{ff}, bn_h l^2)n_l$	$(bld_{ff} + bn_h l^2)n_l$
Reversible Transformer	$\max(bld_{ff}, bn_h l^2)$	$(bn_h l d_{ff} + bn_h l^2)n_l$
Chunked Reversible Transformer	$\max(bld_{model}, bn_h l^2)$	$(bn_h l d_{ff} + bn_h l^2)n_l$
LSH Transformer	$\max(bld_{ff}, bn_h l n_r c)n_l$	$(bld_{ff} + bn_h n_r lc)n_l$
Reformer	$\max(bld_{model}, bn_h l n_r c)$	$(bld_{ff} + bn_h n_r lc)n_l$

Итоги

Осталось убедиться, что нашими экономиями мы не ухудшили качество.



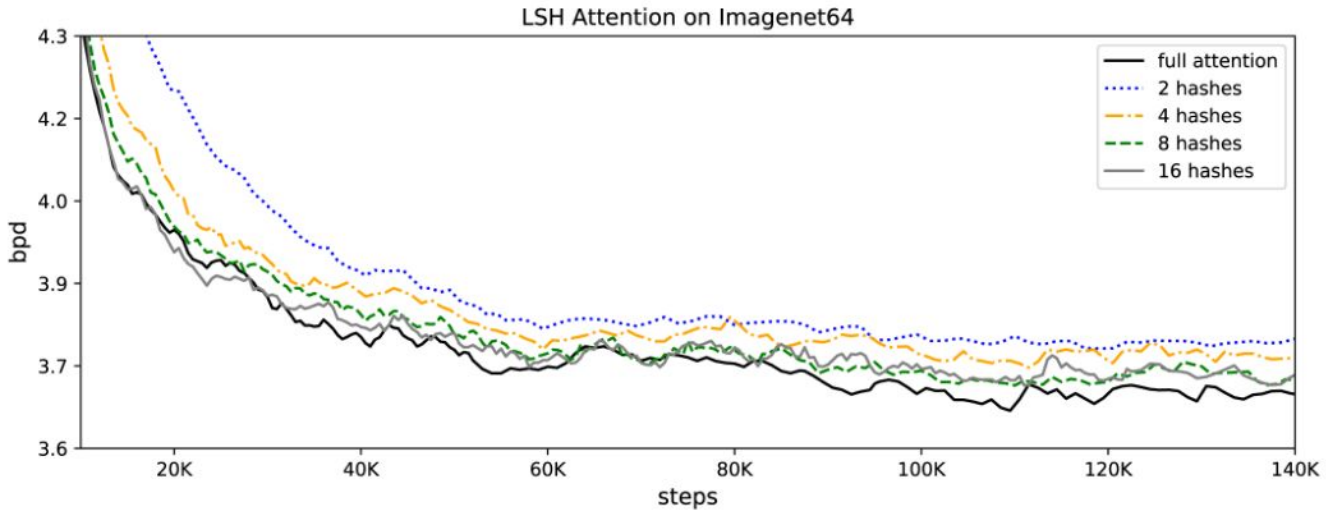
Итоги

Осталось убедиться, что нашими экономиями мы не ухудшили качество.

Model	BLEU	*sacreBLEU* *Uncased*[3]	*sacreBLEU* *Cased*[4]
Vaswani et al. (2017), base model	27.3		
Vaswani et al. (2017), big	28.4		
Ott et al. (2018), big	29.3		
Reversible Transformer (base, 100K steps)	27.6	*27.4*	*26.9*
Reversible Transformer (base, 500K steps, no weight sharing)	28.0	*27.9*	*27.4*
Reversible Transformer (big, 300K steps, no weight sharing)	29.1	*28.9*	*28.4*

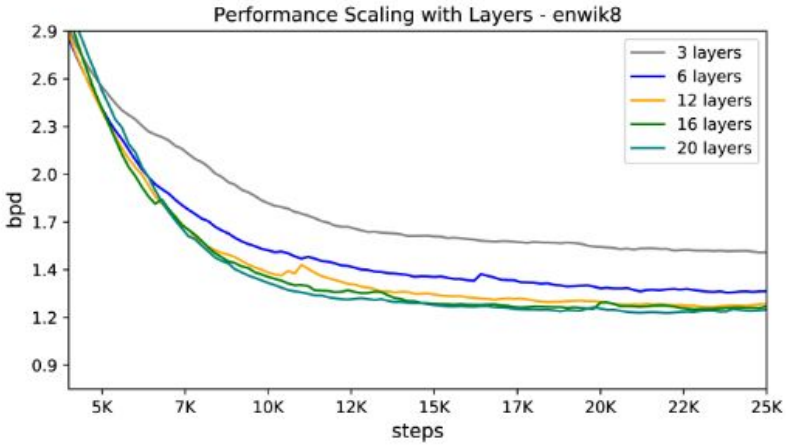
Итоги

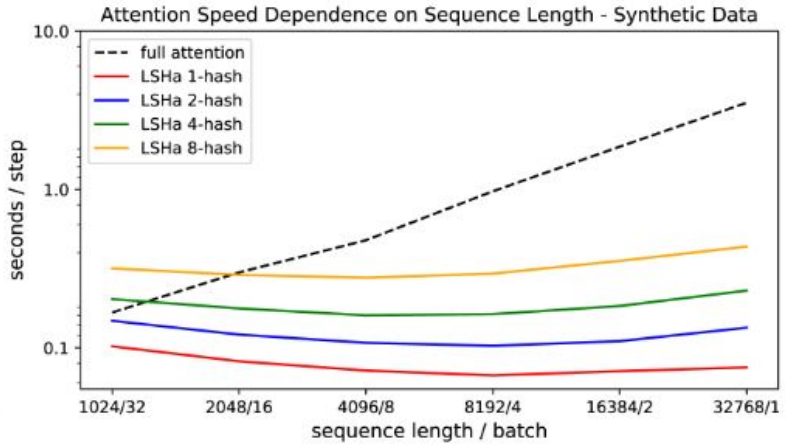
Осталось убедиться, что нашими экономиями мы не ухудшили качество.



Итоги

Осталось убедиться, что нашими экономиями мы не ухудшили качество.





Оригинальная статья

- https://arxiv.org/pdf/2001.04451.pdf