Reformer: the efficient transformer

Охрименко Дмитрий, 172

Проблемы обычного Трансформера

- Он показывает отличные результаты, но крайне неэффективен
- При работе с большим количеством данных приходится обучать трансформеры с 64 слоями, количество параметров каждого из которых больше полумиллиарда
- Это занимает огромное количество памяти, которое могут обеспечить только суперкомпьютеры

Проблемы обычного Трансформера

- 500 000 000 параметров = 2 ГБ памяти
- 64 000 значения функции активации * 1024 (embedding size) * 8 (batch size) = 2 ГБ памяти
- Кажется, что в таком случае вполне хватит памяти, если держать в ней по одному слою за раз

Проблемы обычного Трансформера

- Однако мы кое-что упускаем:
- - нужно хранить все значения всех слоев для бэкпропа (получаем N*N)
- - глубина средних слоев в случае прямой нейронной сети гораздо больше глубины слоев attention, что потребует больше памяти
- - сложность attention по памяти и по времени для последовательности длины L будет L*L
- В итоге такой трансформер не сможет быть обработан без суперкомпьютеров

Как эти проблемы решает Reformer

- - Использование Reversible layers, чтобы хранить только одну копию активаций для всей модели
- - Для нейронных сетей без рекуррентной связи разделять активацию внутри слоев и обрабатывать ее в блоках для сохранения памяти внутри этих слоев
- - Приблизительно вычислять attention (идея основана на локальном хешировании – LSH (locality-sensitivity hashing) позволяет изменить O(L*L) на O(L*log(L))
- Данные методы позволяют не потерять в качестве, при этом сильно прибавить в эффективности относительно обычного Трансформера

Dot-product attention

Самый обычный вид attention.

На вход подаются запросы (Q), ключи (K) и значения (V).

Для получения весов значений запросы перемножаются с ключами, масштабируются по размерности ключей, после чего к полученному результату применяется softmax:

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

Multi-head attention

- Усовершенствованный dot-product attention.
- H раз линейно проецируем запросы, ключи и значения на dk, dk, dv
- Применяем Attention к каждой из прогнозируемых версий запросов, ключей и значений параллельно, давая двумерные выходные значения.
- Объединяем и еще раз проецируем промежуточные ответы и получаем окончательный разультат.

Memory-efficient attention

- Попробуем подсчитать сложность по памяти для $softmax(\frac{QK^T}{\sqrt{d_k}})V$.
- Пусть все Q, K и V имеют размерности [batch size, length, dmodel]. Тогда размерность QK^T = [batch size, length, length].
- Для последовательностей длины 64 000, даже при размере батча 1, это матрица 64K × 64K, которая займет 16 ГБ памяти, что очень много.
- На самом деле, не обязательно хранить всю матрицу QK^T.
- Attention может быть вычислено для каждого запроса q_i отдельно, только вычисляя $softmax(\frac{q_iK^T}{\sqrt{d_k}})V$ один раз в памяти, а затем повторно вычислять его на обратном проходе, когда это необходимо для градиентов.
- Для данного способ сложность по памяти = O(L), хотя может слегка упасть эффективность.

Откуда берутся Q, K, V?

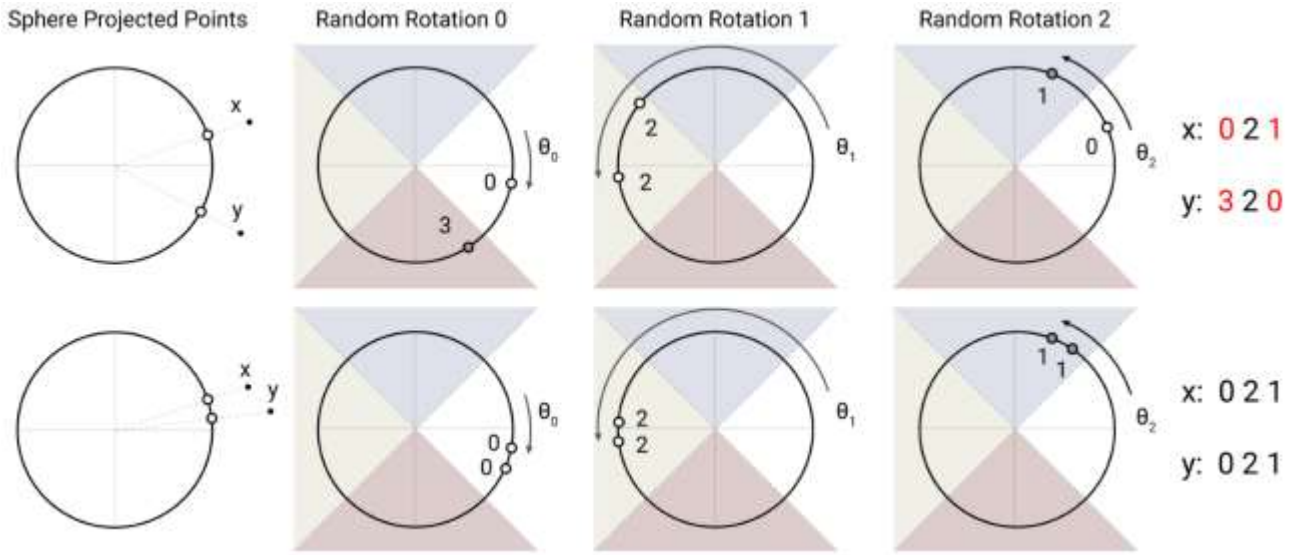
- Обычно нам дается только тензор активации A с размерностями [batch size, length, dmodel]
- Чтобы построить Q, K и V из A, используются 3 различных линейных слоя, проецирующих A в Q, K и V с различными параметрами.
- Идентичность Q и K достигается использованием одного и того же слоя для перехода из A в Q и в K соответственно.

Hashing attention

- D LSH attention на вход подается два тензора Q = K и V размерности [batch size, length, dmodel].
- Основная проблема возникает с матрицей QK^T.
- Однако на самом деле нас интересует softmax(QK^T).
- Поскольку в softmax наибольшие элементы вносят наибольший вклад, для каждого запроса q_i нам нужно сосредоточиться только на тех ключах в K, которые наиболее близки к q_i.

Locality sensitive hashing

- Соседние векторы получают одинаковый хэш с высокой вероятностью, а так же размер хэш-блоков должен быть равным.
- Достигается использованием случайных проекций, как на рисунке.



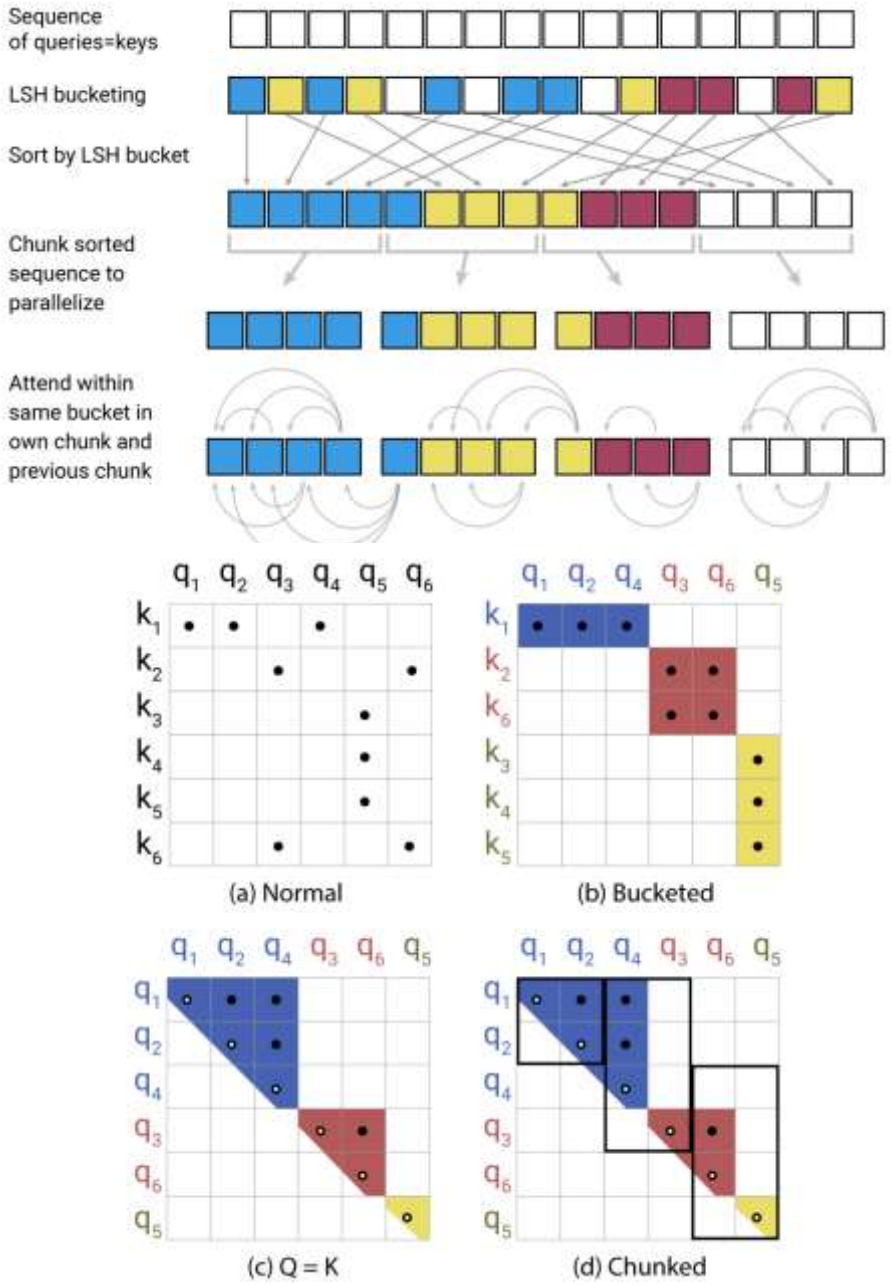
Чтобы получить b хешей, мы сначала фиксируем случайную матрицу R размера $[d_k, \frac{b}{2}]$. Затем мы определяем h(x) = argmax([xR; −xR]), где [u; v] обозначает конкатенацию двух векторов.

LSH attention

- $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ для q_i: $o_i = \sum_{\{j \in P_i\}} \exp\left(q_i k_j - z(i, P_i)\right) v_j$, где P $= \{j, i \geq j\}$
- P – множество, к которому обращается запрос на позиции i
- z - функция разбиения (нормализующий член в softmax)
- Удобно применять attention к $\widehat{P_i} = \{0, 1, \ldots, l\} \subseteq P_i$, маскируя элементы, не входящие в P:
- $o_i = \sum_{\{j \in \widehat{P_i}\}} \exp\left(q_i k_j - m(j, P_i) - z(i, P_i)\right) v_j$, $\mathrm{m(j, P_i)} = \begin{cases} \infty, j \notin P_i \\ 0, \text{иначе} \end{cases}$
- Для LSH attention: $P_i = \{j : h(q_i) = h(k_j)\}$

LSH attention

- a) Полный attention, разреженная матрица, которое не используется
- b) Q и K отсортированы по хэшам
- c) Решаем проблему неравенства размеров Q и K: h(k_j) = h(q_i) [$k_j = \frac{q_j}{\|q_i\|}$], затем сортируем запросы по номеру сегмента и, внутри каждого сегмента, по позиции последовательности
- d) После сортировки объекты взаимодействуют с объектами в своем и предыдущем сегментах
- $\widetilde{P_i} = \{j: \left\lfloor \frac{s_i}{m} \right\rfloor - 1 \leq \left\lfloor \frac{s_j}{m} \right\rfloor \leq \left\lfloor \frac{s_i}{m} \right\rfloor\}$.
- Если $\max_i |P\_i| < m$, то $P_i \subseteq \widetilde{P_i}$
- На практике берем $m = \frac{2l}{nbuckets}$



Результаты

- Задача – продублировать последовательность символов
- 0w0w = 0 19 113 72 0 19 113 72
- Обучаем языковую модель, где каждый w имеет длину 511
- Точность при оценивании с 8 хэшами практически идеальна

Table 2: Accuracies on the duplication task of a 1-layer Transformer model with full attention and with locality-sensitive hashing attention using different number of parallel hashes.

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%

Сравнение точностей различных attention

Table 1: Memory and time complexity of attention variants. We write l for length, b for batch size, n_h for the number of heads, n_c for the number of LSH chunks, n_r for the number of hash repetitions.

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	$\max(bn_hld_k, bn_hln_r(4l/n_c)^2)$	$\max(bn_hld_k, bn_hn_rl(4l/n_c)^2)$

- Сложность может быть уменьшена с квадратной до линейной
- Активации перед каждым слоем уже имеют размер b · l · dmodel, поэтому использование памяти всей модели с nl слоями составляет не менее b · l · dmodel · nl.
- В Transformer с прямой связью это возрастает до b · l · dff · nl.
- В больших Transformer устанавливают dff = 4 000 и nl = 16, поэтому при l = 64 000 это будет использовать 16 ГБ памяти

RevNets

- Основная идея состоит в том, чтобы позволить активациям на любом данном уровне быть восстановленными из активаций на следующем слое, используя только параметры модели.
- Суть в том, чтобы при бэкпропе инвертировать слои один за другим, а не проверять промежуточные значения
- Вместо работы с единичным входом и выходом: x -> y, y = x + F(x), RevNets работает на парах: (x1,x2) -> (y1, y2) с такими формулами:

$$y_1 = x_1 + F(x_2); y_2 = x_2 + F(x_1)$$

- Для переворота слоя достаточно вычесть остатки:

$$x_1 = y_1 - F(x_2); x_2 = y_2 - F(x_1)$$

Reversible transformer

- Применяем идею RevNet к трансформеру, комбинируя attention и слои с прямой связью внутри revnet блока:

$$Y_1 = X_1 + Attention(X_2); Y_2 = X_2 + FeedForward(Y_1)$$

- Не нужно хранить активации на каждом слое, поэтому множитель n_l уходит
- Если взять x1 и x2 размера dmodel, то аналогичен результату обычного Трансформера

Chunking

- Толстые слои все равно могут использовать много памяти, поэтому стоит разбить вычисления в них на куски:

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

- Обратный проход тоже делится на куски, кроме того, в случае модели с большим словарем мы также разбиваем логарифмические вероятности на выходе и рассчитываем потери для участков последовательности за раз.

Reformer

- Итак, благодаря кускованию и RevNets память, которую мы использует для активации во всей сети, не зависит от количества слоев
- Это не касается параметров, однако это может быть исправлено путем свапанья параметров слоя в память процессора и из нее, когда этот уровень не вычисляется.
- Этот же метод неэффективен в обычном Трансформере из-за медленной передачи данных на CPU и их большего количества

Результаты

Table 3: Memory and time complexity of Transformer variants. We write d_{model} and d_{ff} for model depth and assume $d_{ff} \geq d_{model}$; b stands for batch size, l for length, n_l for the number of layers. We assume $n_c = l/32$ so $4l/n_c = 128$ and we write $c = 128^2$.

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	$\max(bld_{model}, bn_h l n_r c)$	$(bld_{ff} + bn_h n_r lc)n_l$

Эксперименты

- Экспериментировали с задачами imagenet64 и enwik8-64K (enwik8, разбитый на подпоследовательности длины 2^16)
- Использовали трехслойные модели, чтобы была возможность сравнения с обычным Трансформером со сложностью O(L^2)
- Параметры: dmodel = 1024, dff = 4096, nheads = 8, batch size = 8
- Оптимизатор для обучения – Adafactor
- Обучение для всех экспериментов было распараллелено на 8 устройствах

Эксперименты с QK и Reversible Transformer

- Эффект от приравнивания Q = K. Shared QK attention устанавливает $k_j = \frac{q_j}{\|q_j\|}$ - левые графики. Результат не хуже, чем у обычного attention
- Эффект Reversible transformer. С одинаковыми параметрами кривые обучения Reversible и обычного трансформеров примерно равны – правые графики

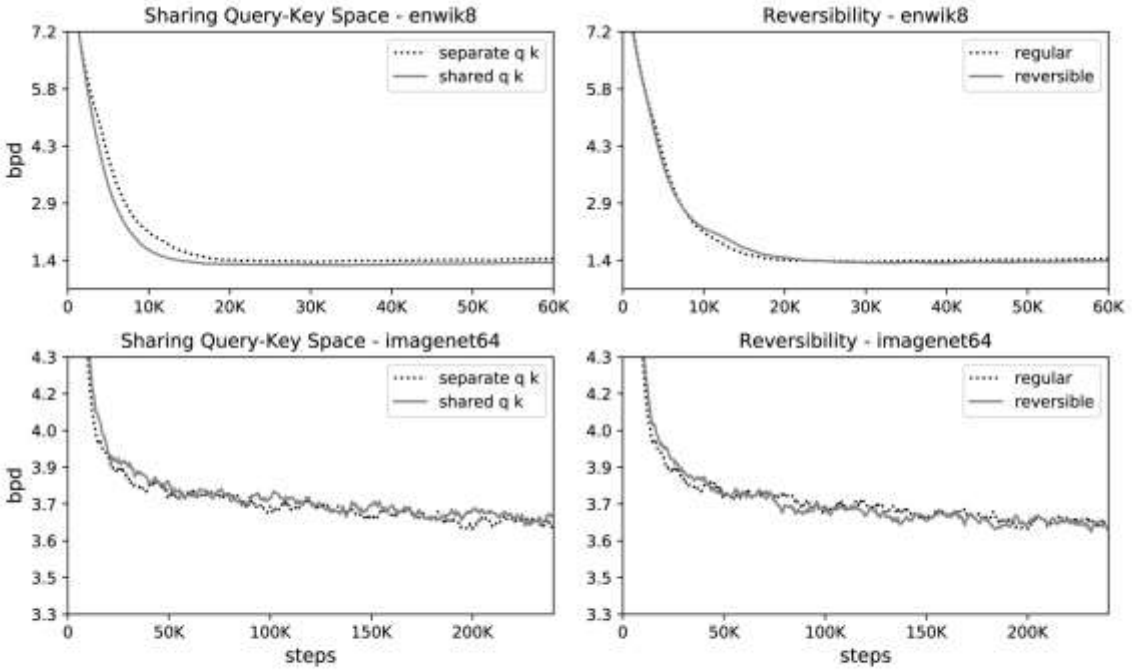


Figure 3: Effect of shared query-key space (left) and reversibility (right) on performance on enwik8 and imagenet64 training. The curves show bits per dim on held-out data.

Эксперименты с RevNets в машинном переводе

- Перевод с английского на немецкий языки
- При обучении на 100К шагов результат сравним с моделью VasWani 2017 года
- Наша модель очень эффективна по памяти

Table 4: BLEU scores on newstest2014 for WMT English-German (EnDe). We additionally report detokenized BLEU scores as computed by sacreBLEU (Post, 2018).

		sacreBLEU	
Model	BLEU	*Uncased*[3]	*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*

Эксперименты с LSH attention

- С увеличением числа хэшей LSH attention приближается к полному.
- В то время как обычный attention становится медленнее при большей длине последовательности, скорость внимания LSH остается неизменной.

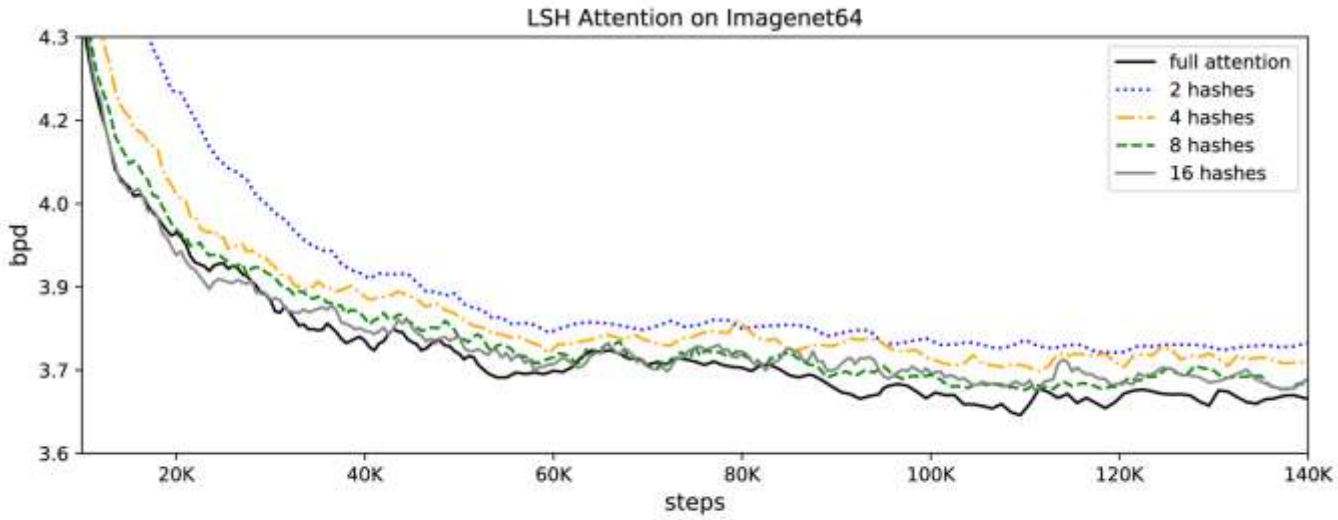


Figure 4: LSH attention performance as a function of hashing rounds on imagenet64.

Эксперименты с большими моделями Reformer

- До 2—слойные модели Reformer на enwik8 и imagenet64 вписываются в память и обучаются на одном ядре.

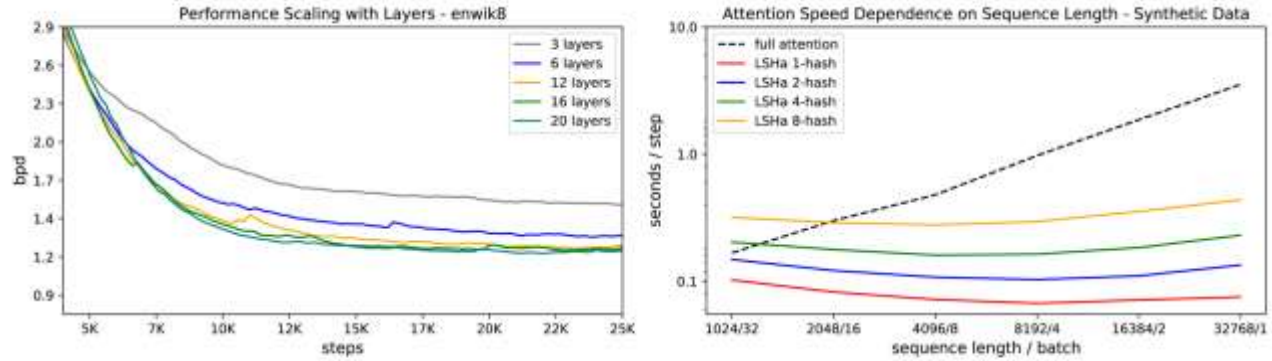


Figure 5: Left: LSH attention performance as a function of number of layers on enwik8. Right: Speed of attention evaluation as a function of input length for full- and LSH- attention.

Выводы

- Reformer объединяет возможности Transformer с архитектурой, способной эффективно выполняться на длинных последовательностях с небольшим использованием памяти даже для моделей с большим количеством слоев
- Способность обрабатывать длинные последовательности дает возможность Reformer быть использованным в задачах прогнозирования временных рядов, генерации музыки, изображений или видео

Ссылки

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