Abstract

- 2 techniques for improving efficiency of transformer
  a) Dot-product attention → Locality Sensitive Hashing
  $(O(N^2)) \rightarrow (O(N \log N))$
  b) Reversible Residual Layers - store activation once instead of storing every layer.
- performs on par with transformers while being memory & time efficient

---

6.) - bring power of transformers to other domains like time-series forecasting, music, image, video generation

---

1.) INTRODUCTION

"Generating wikipedia articles" - 11k tokens of text in a single example input to a transformer

- Models too large to be even fine-tuned on a single GPU.
- Are large transformers really efficient in their parameter usage?

0.5B params = 2 GB RAM

64k input tokens × 1024 embedding size × 8 batch-size = 0.5 B floats = 2GB RAM

$\Rightarrow 2+2 = 4$ GB

- If memory was per layer ⇒ fit a transformer easily
- whole corpus used to train BERT is only 17 GB. So, why we cannot even fine-tune on a single machine??

Reasons

a) N layers = N time more memory

b) d_{ff} - depth of feed forword layers $\gg d_{model}$

c) Attention Calculation is $O(L^2)$

Solution

a) Reversible Layers → only one copy of activations instead of N.

b) Splitting activations inside f-f layers & processing in chunks.

c) Approximate attention computation using Locality Sensitive Hashing - $O(L^2) \rightarrow O(L \log L)$

→ is even numerically identical to original Transformers.

Tasks used:

↳ enwik8 (64k sequence)

↳ image generation - imagenet 64k - generation. (12k sequence length).

---

2. LOCALITY-SENSITIVE HASHING ATTENTION - LSH Attention

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

$Q, K, V \rightarrow (bsz, length, d\text{-}model) \Rightarrow QK^T = (bsz, length, length)$.

64K x 64K Matrix - 32 bit float

$= 64{,}000 \times 64{,}000 \times 32$

$= 16$ GB

Alternative : $\text{softmax}\left(\frac{q_i K^T}{\sqrt{d\_model}}\right) V$ is linear in seq_len

You have $A = (bsz, length, d\_model)$

Then 3 linear layers project them to

$A \to L1 \to Q$

$A \to L2 \to K$

$A \to L3 \to V$

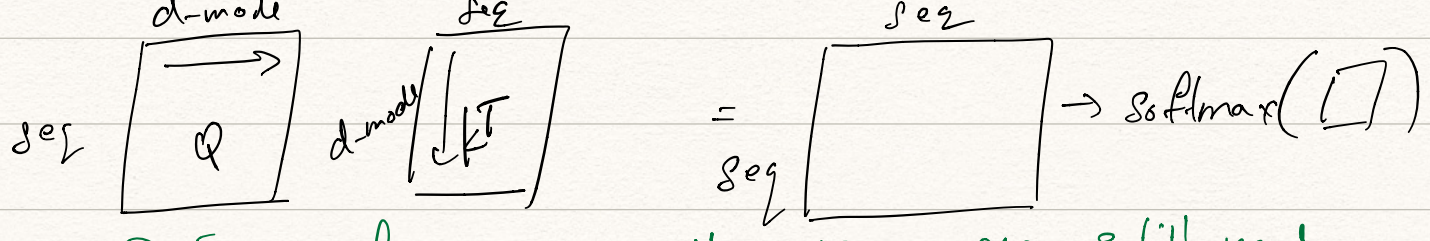
LSH $\Rightarrow$ $L1 = L2$ - "Shared-QK Transformer"

Sharing Q+K does not affect the performance

Hashing Attention:

we need $\text{softmax}(QK^T) \Rightarrow$ largest values & not all the values. (about 0 = maximum value)

$\Rightarrow$ We need values closer to q_i (may 32 of them).



① Even if you narrow it down, you still need all "seq x seq" of them right?

(2) Also, doing Nearest Neighbour is faster?

Locality Sensitive Hashing. (LSH)

(TUTORIAL TIME!)

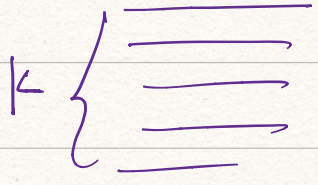
LSH → hash similar items similarly. This is the opposite behavior of cryptographic hashing. In LSH, we want to maximize the probability of collisions of similar items.

$P(h(a) == h(b))$ = low if a near b in space.

LSH using Random Projection.

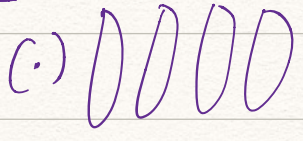
$$\underset{k \times n}{[\text{Projected}(P)]} = \underset{k \times d}{[\text{Random}(R)]} \; \underset{d \times n}{[\text{Original}(D)]}$$

where $k << d$



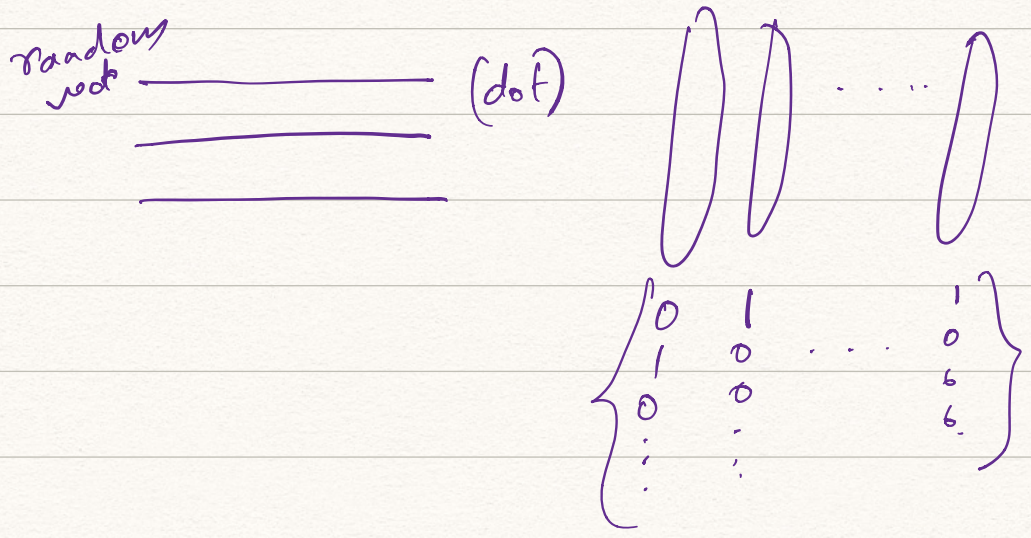
k - d-dimensional vectors sampled from $N(0,1)$

n-documents of d-dimension.



P → if $P > 0$, put binary 1

if $P <= 0$, put binary 0



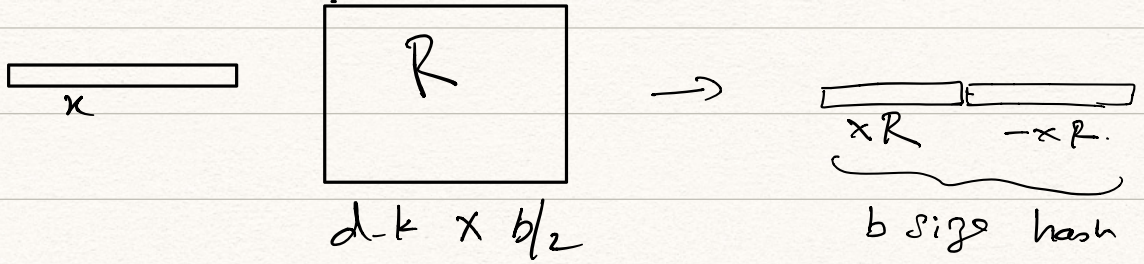
Hash Table ⇒

If the binary representation is same for 2 documents then they are supposed to be similar.

- Due to the random nature we sometimes construct multiple hash tables (like multiple heads in a transformer).
- The documents that have similar binary representation are supposed to have higher cosine similarity.

BACH!!

Our requirement: Nearby vectors get the same hash with high probability.



ie, $h(x) = \text{argmax}([xR; -xR])$. ✓

gives axis of the maximum value.

Normal:

$$o_i = \text{softmax}\left(\sum_j q_i k_j\right) \cdot V$$

[seq] [ ]

seq x d_model

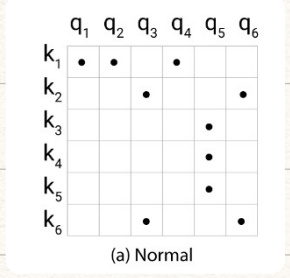
LSH:

$$o_i = \sum_{j \in P_i} \exp\left(q_i k_j - z(i, P_i)\right) v$$

P_i → set of keys matching the query

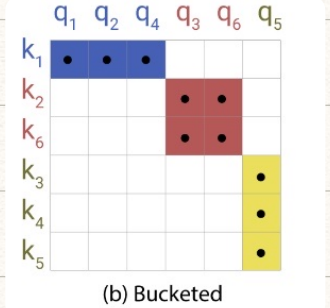
z → Normalizing factor

$P_i = \{j : h(q_i) == h(k_j)\}$



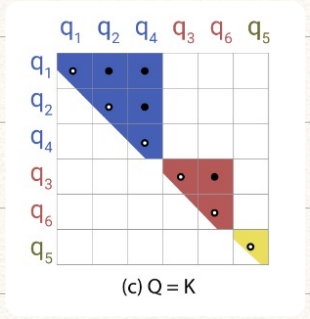
(a) Normal

→ Normal attn matrix which is sparse



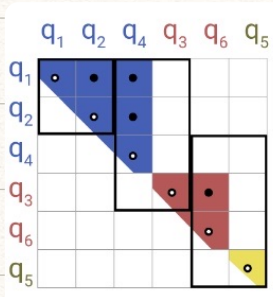
(b) Bucketed

→ Find matching queries & keys. Key could be uneven, also, number of queries != number of keys. But we know setting Q=K would hamper performance



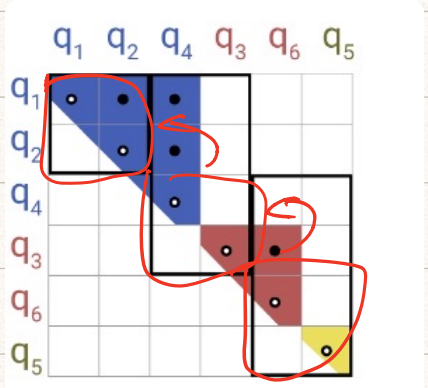
(c) Q = K

→ Setting $q = k$ ie, $k_j = \frac{q_j}{\|q_j\|}$



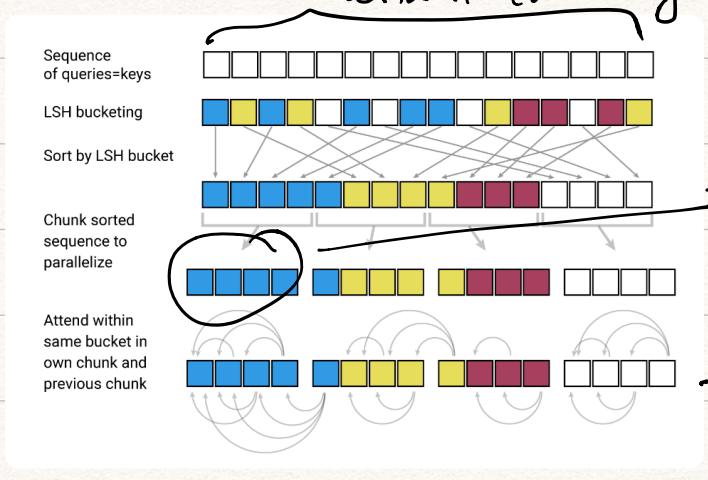
(d) Chunked

→ why not this → ??



(d) Chunked

without chunking this is the attn matrix dims.



→ after chunking it is this.

→ cross chunk attention.

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%

You can train 4 fast with different LSH models.

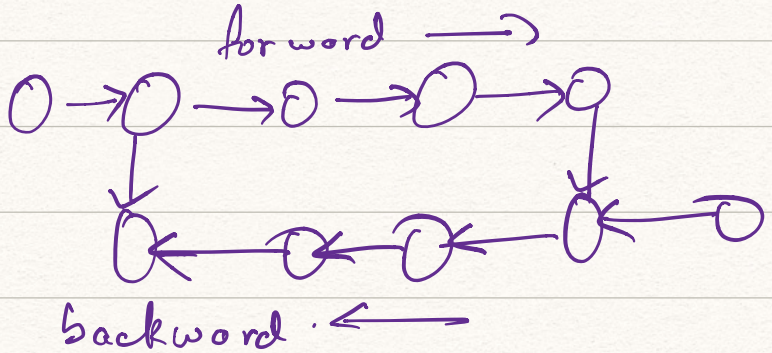
3) Reversible Transformer

Rev Nets:

Interesting paper: "The Reversible Residual Network: Back propagation without storing activations"

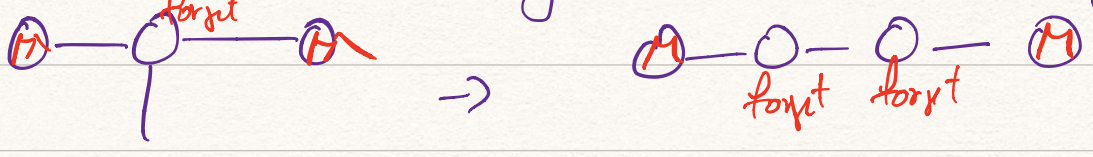
- instead of checkpointing for use in backward pass, you can reverse layer-by-layer during backprop — wow!

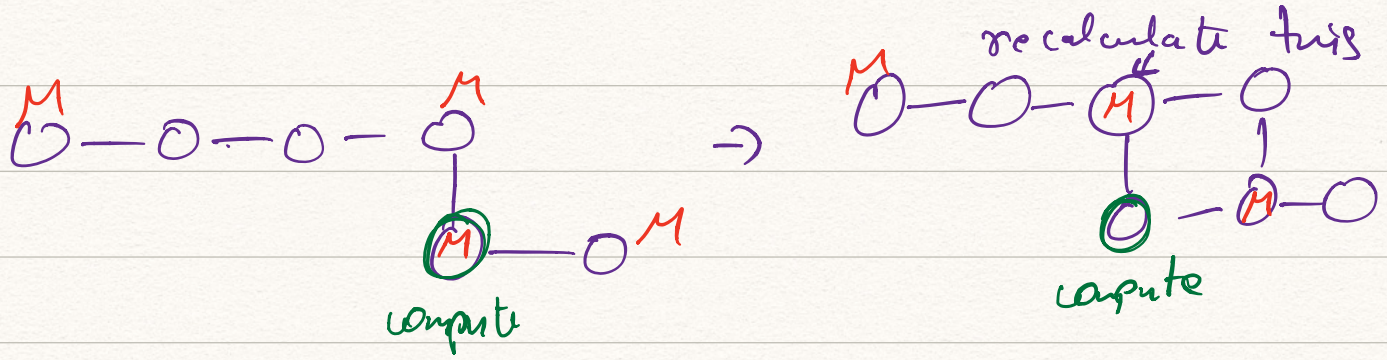
Checkpointing



You need to keep all the circles in memory to compute backward pass one layer at a time.

Instead there is a memory efficient - computationally inefficient way

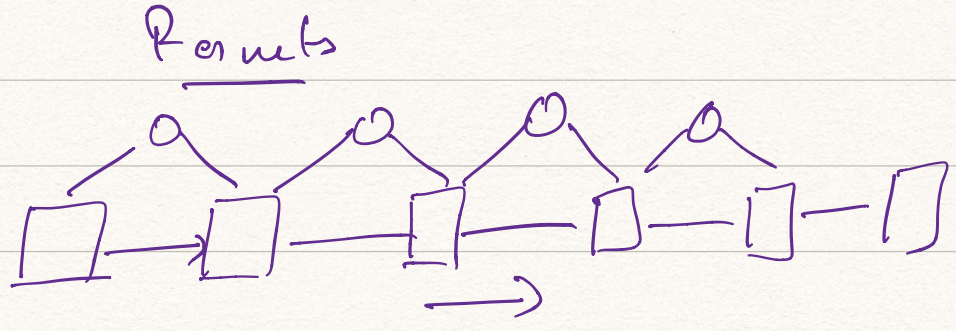




Yep, You recalculate the forward pass till you meet the required node & then compute backward pass on it.

You could pick some nodes & save their intermediate results.

eg: For a chain of length 'l', place checkpoints at every $\sqrt{(n)}$. This way you can be more memory & time efficient

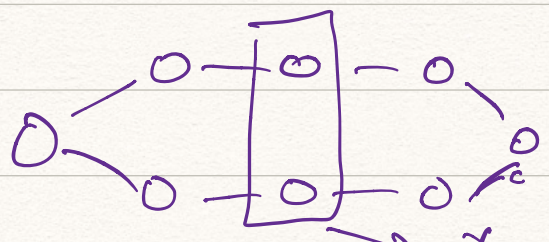


If makes no sense to checkpoint the circles.

Squares nodes are the best.

Knowing "ON" would remove the need to recompute anything before - called - "graph separators"

eg: example of graph operators of size 2



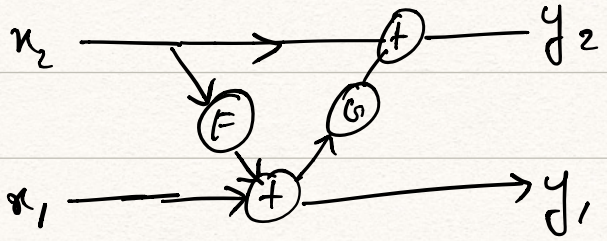
You need checkpoint pairs like this

BACK!!

Normal Resnets : $x \rightarrow y$, $y = x + F(x)$

Rev Nets: : Split your input(x) into (x_1) & (x_2)

FORWARD



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

BACKWARD we have y_2 & y_1,

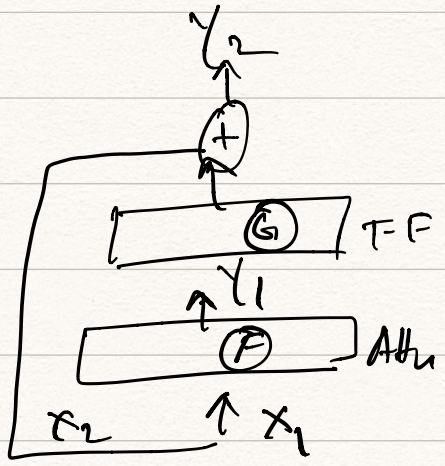
$$x_2 = y_2 - G(y_1)$$
$$x_1 = y_1 - F(x_2)$$

This way we can compute the activation of previous layer $(x_1 \& x_2)$ from current layer $(y_1 \& y_2)$

Reversible Transformer

$$Y_1 = X_1 + \text{Attn}(X_2)$$

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



$X_1 + X_2 \in \mathbb{R}^{d_{model}}$.

This can be chunked. & thereby can improve parallelism. Also operating on one chunk at a time can reduce memory.

* Now, due to chunking & reversible layers, the memory we use for the forward pass is independent of the number of layers

---

RELATED WORK

There is a lot of very interesting ideas. I have no idea if I'll be able to go through them.

---

5 EXPERIMENTS

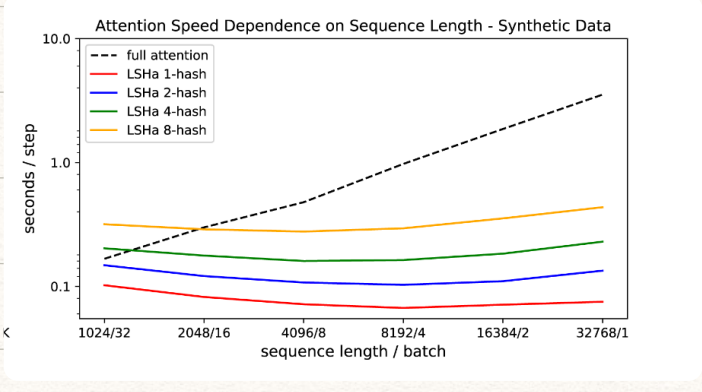
① Reversible Layers & shared QK does not hamper the performance.

② Shared QK, $k_i = \frac{q_i}{\|q_i\|}$. Should prevent tokens attending themselves: why??

③ 20 layer Reformer models fit in the memory

④ LSH-8 almost matches full attention.

⑤



The speed of the computation remains mostly constant for LSH when seq-length is increased.