REFORMER : The efficient Transformer

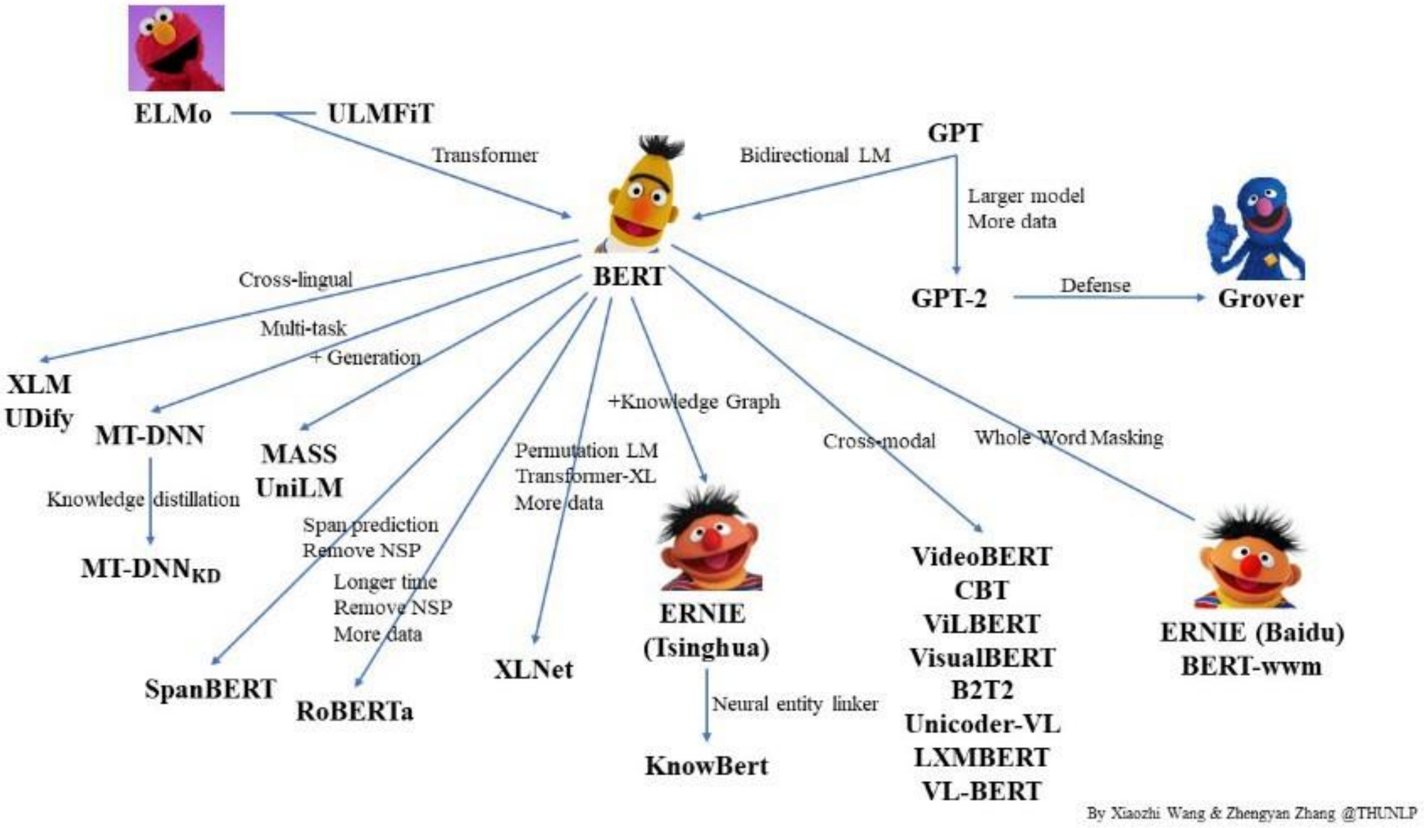
Blog: https://ai.googleblog.com/2020/01/reformer-efficient-transformer.html
Paper: https://arxiv.org/abs/2001.04451
Openreview: https://openreview.net/forum?id=rkgNKkHtvB

Oral paper at ICLR 2020, spotlight

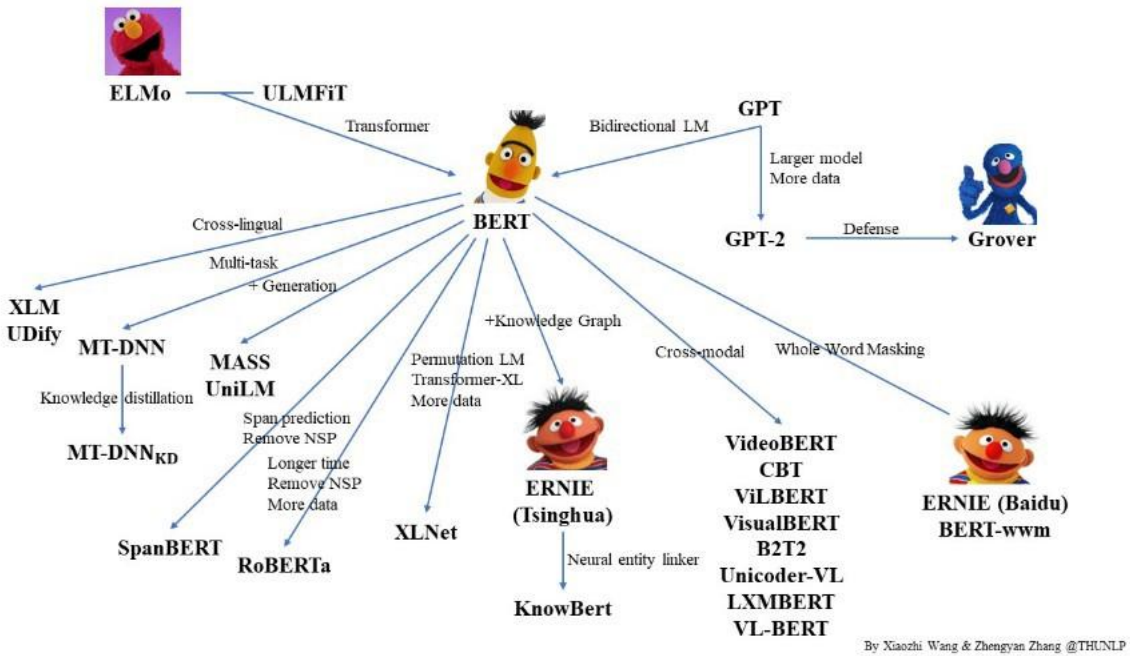
Slide index

1) Some background on state of the art NLP and the issues : quadratic memory growth
2) Reduce memory usage using hash and divide-and-conquer
3) Reduce more memory by implementing reversible transformer
4) Ablation study results
5) Summary
6) Reference

The **Cool kids** in Natural Language Processing



The **Cool kids** in Natural Language Processing



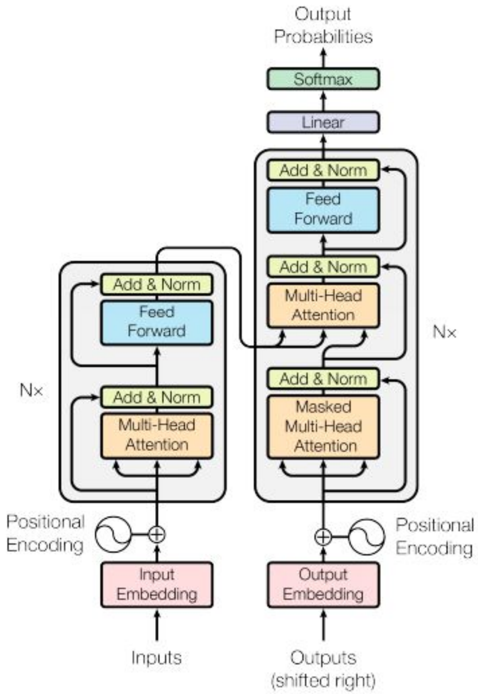


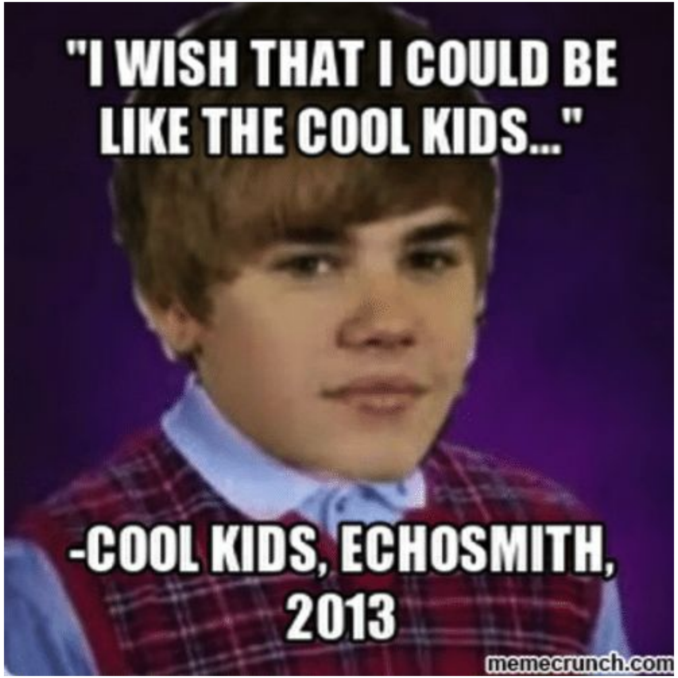
Figure 1: The Transformer - model architecture.

Me : I want to be the cool kids

Try to run XLM model in GTX 1080, get out of memory errors instead

```
tokenizer = XLMTokenizer.from_pretrained('xlm-mlm-100-1280')
model = XLMWithLMHeadModel.from_pretrained('xlm-mlm-100-1280')
args.length = adjust_length_to_model(args.length, max_sequence_length=
model.to('cuda')
preprocessed_prompt_text = prepare_xlm_input(args, model, tokenizer, 
encoded_prompt = tokenizer.encode(preprocessed_prompt_text, add_specia
encoded_prompt = encoded_prompt.to('cuda')
```

```
  in linear
    output = input.matmul(weight.t())
RuntimeError: CUDA out of memory. Tried to allocate 1.6
8 GiB (GPU 0; 7.80 GiB total capacity; 2.17 GiB already
 allocated; 1.08 GiB free; 2.21 GiB reserved in total b
y PyTorch)
(env) theblackcat102@lab:~/Documents/HW2$ 
```

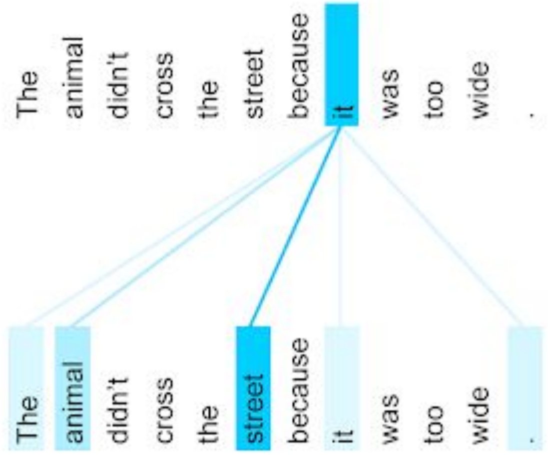


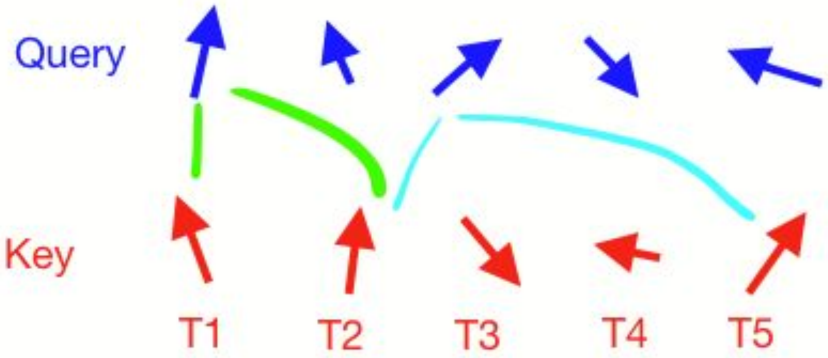
Why?

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

$O(n^2)$

1. Self attention matrix grows quadratic with input length
2. Existing backpropagation strategy stores inputs and outputs of before each activation (memory x 2)

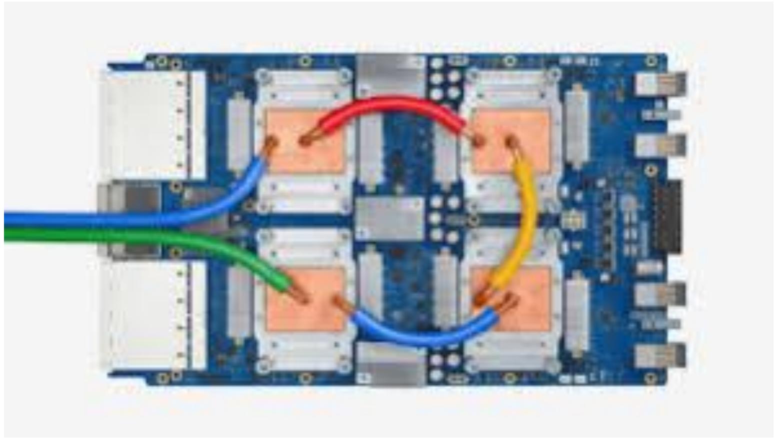


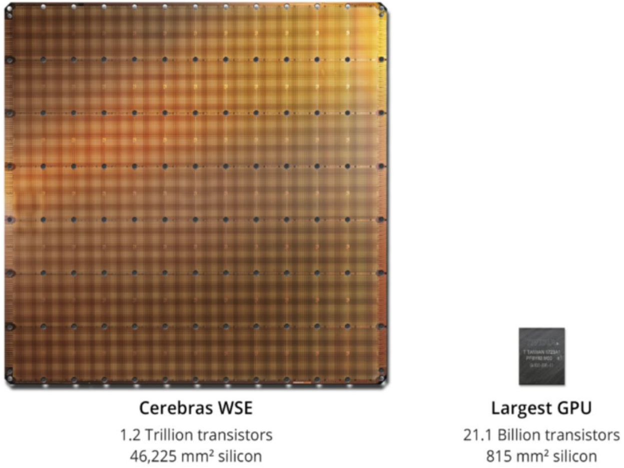


Solution 1 : Sparse operation

Self attention between query and key vector is normally a sparse matrix. We can use sparse operation to save memory footprint.

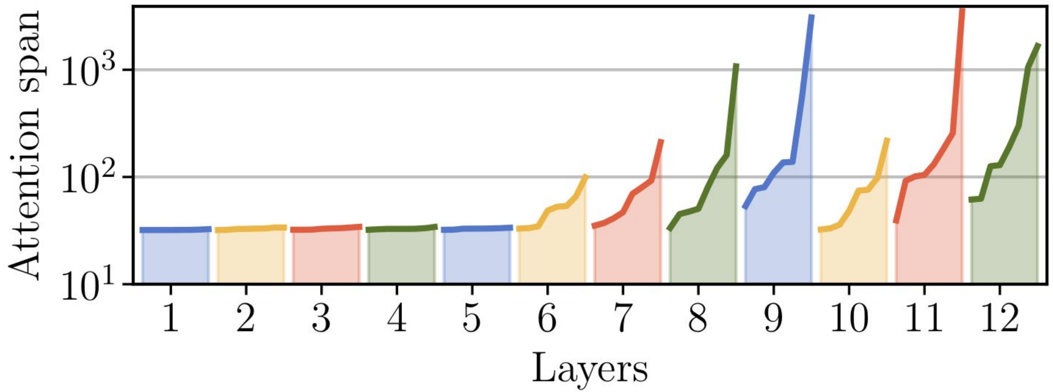
But, sparse operation require ~~cool kids toys~~ fancy hardware support (ie: TPU) for fast performance; software implementation is slow as he**





List of sparse transformer

- [Sparse Sinkhorn Attention]() - google
- [Generating Long Sequences with Sparse Transformers]() - openai
- [Adaptive Attention Span in Transformers]() - facebook



Solution 2: Recursive operation

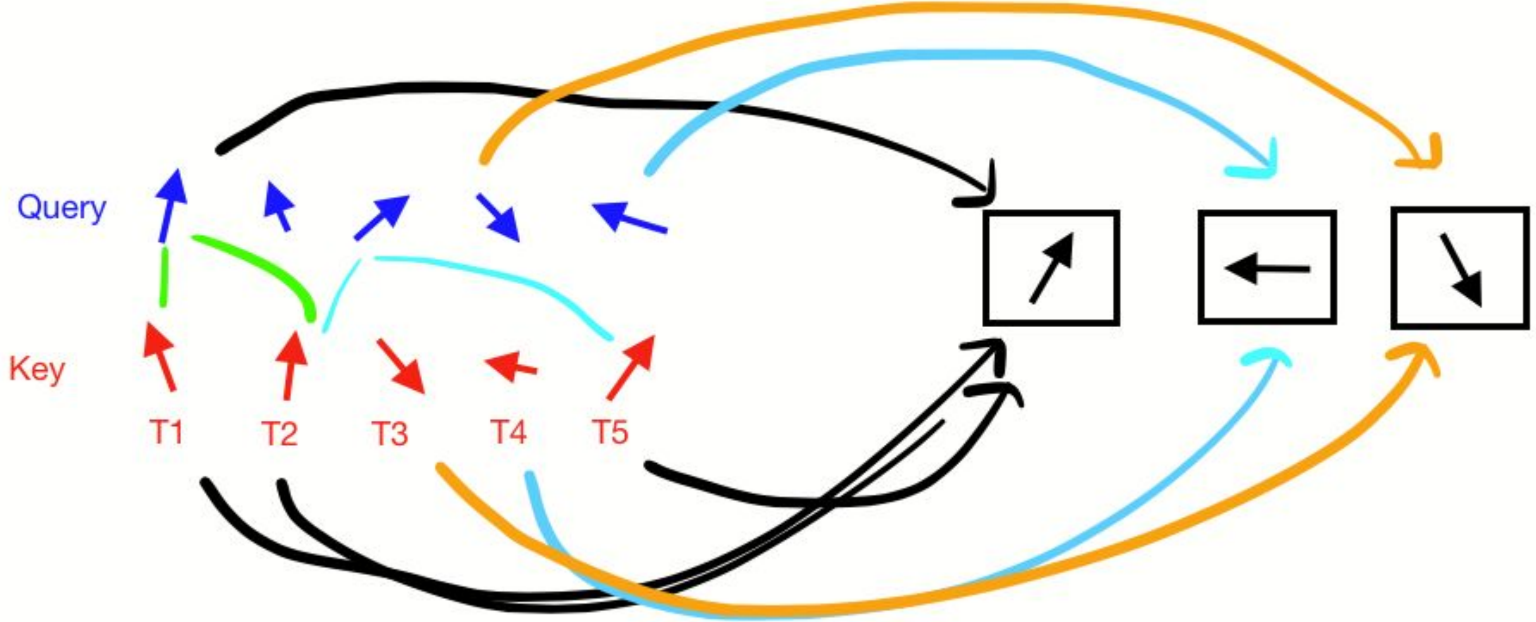
A lite BERT (ALBERT) : recursive transformer layers and matrix factorized embedding to save memory.

But backpropagation still need to store inputs, outputs for each recursive iteration.

And the “real” culprit : self attention matrix size = length x length

Trade computation for memory : LSH Attention

1. Share matrix for query and key (follow up later)
2. Split self attention matrix into buckets , only calculate attention similar vectors

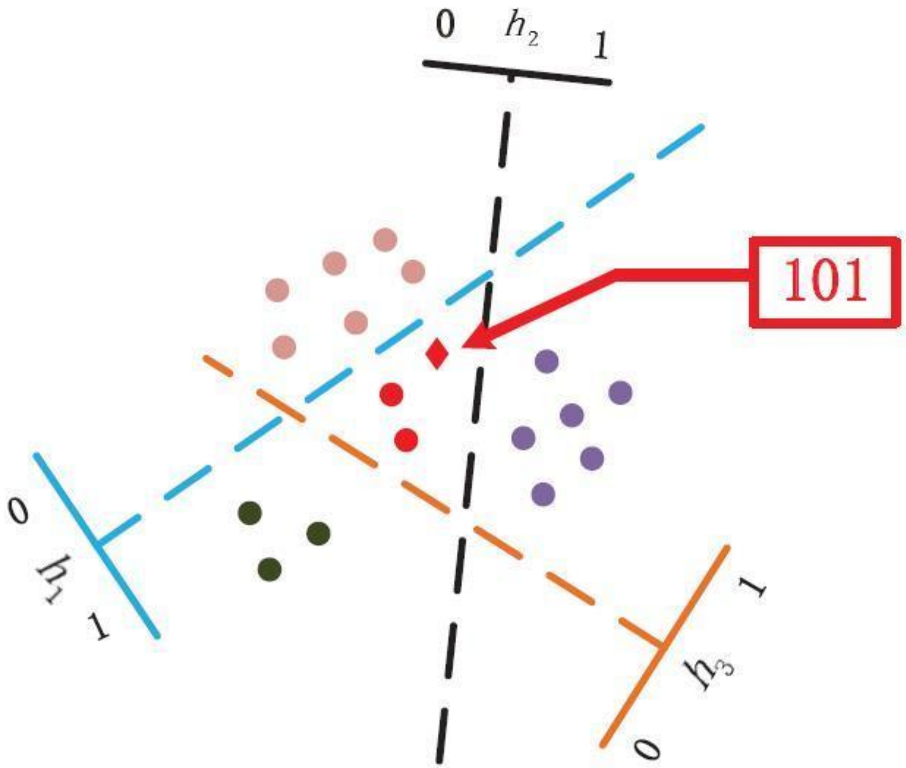


But similarity calculation can be expensive

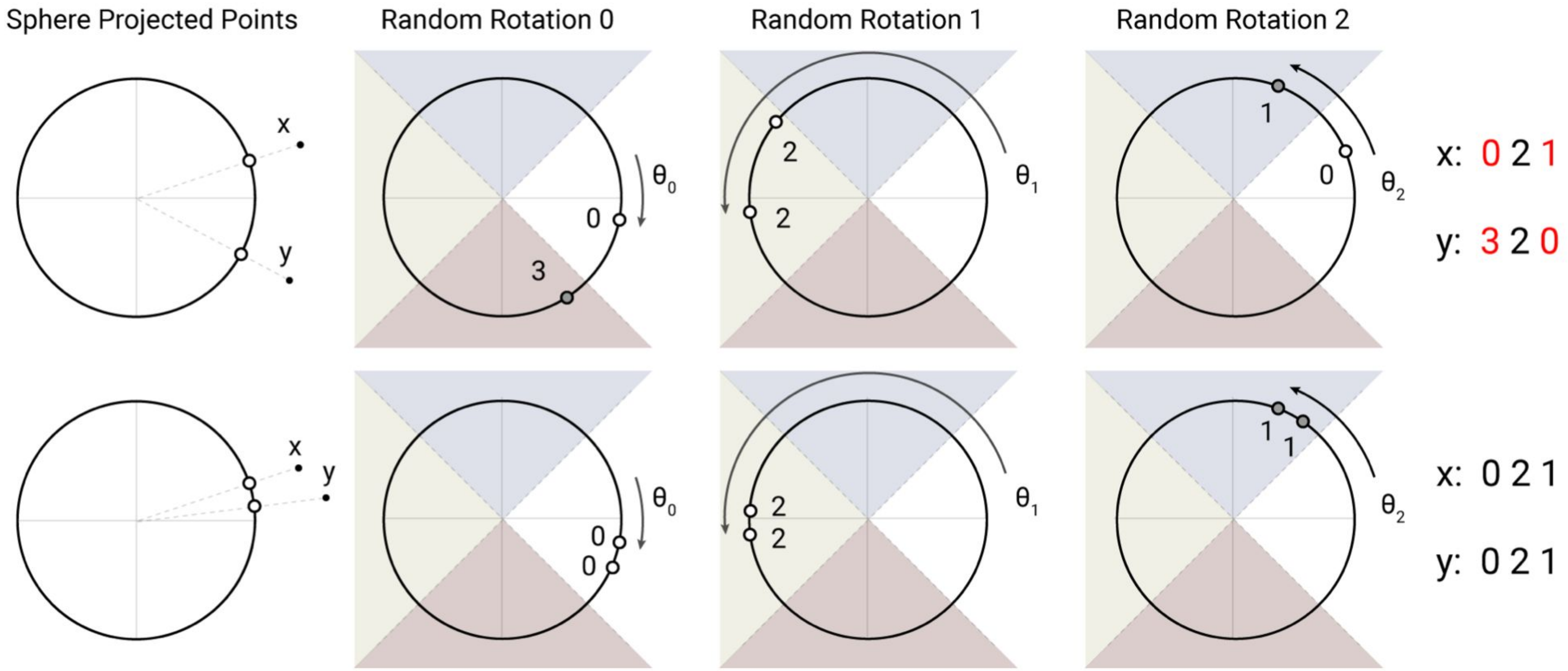
One way to do this is hashing :

Locality Sensitive Hashing!

As with the problem of hashing,
there maybe collision



Multiple random rotation hashing



Multiple random rotation hashing

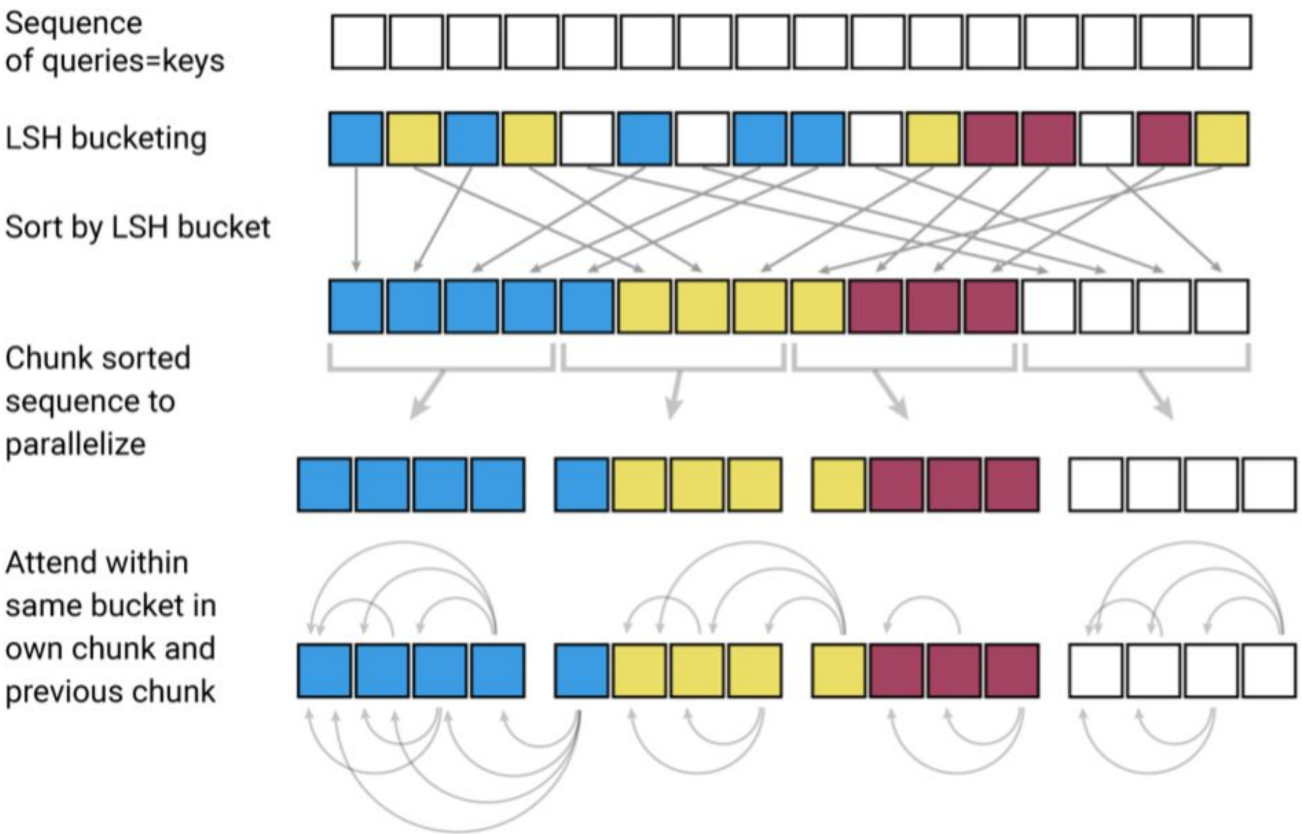
suppose we want to split vector into b buckets, given a vector x we can find the hash code as follows

$$R: \text{random matrix with size of } [d_k, b/2]$$

$$h(x) = argmax([xR; -xR])$$

we can use the max value of i-th index as bucket id

Full illustration of LSH attention



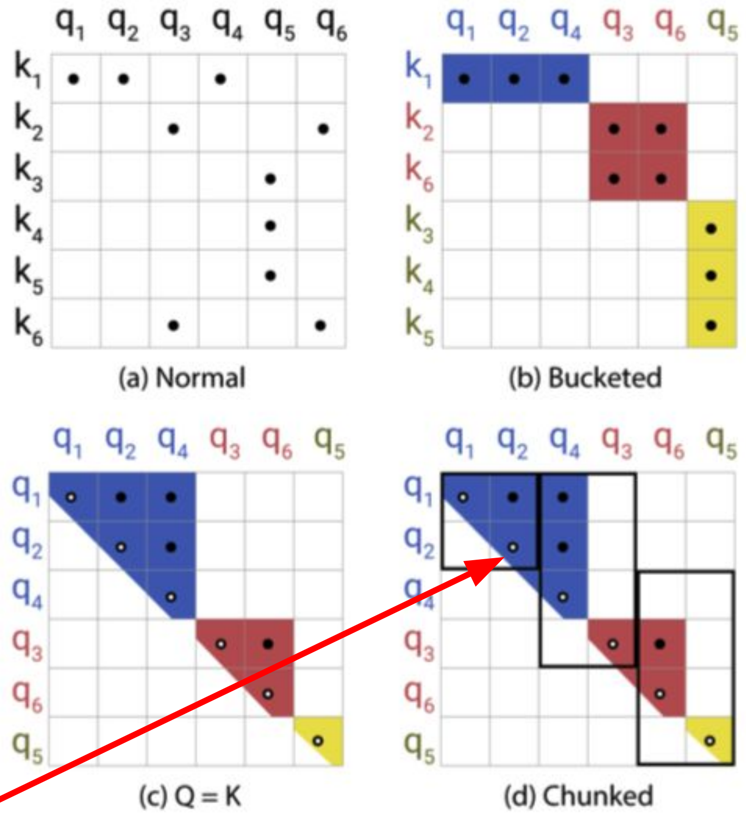
Query matrix = Key matrix

Imbalance issues if use different matrix for query, key

Number of queries and numbers of keys within a bucket maybe unequal.

Let Q matrix = K matrix and set key vector as the normalized version of query vector

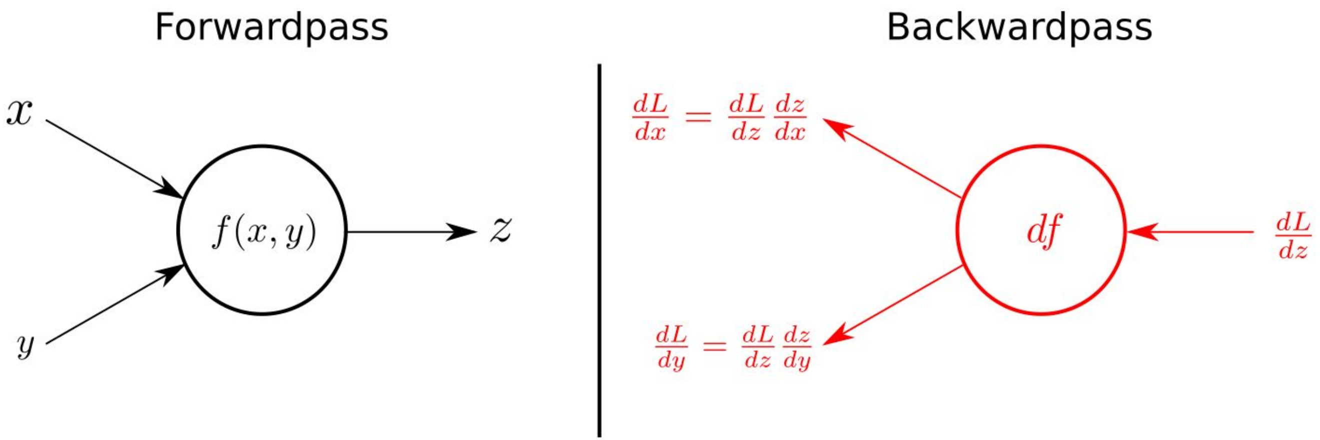
$$h(k_j) = h(q_j) \text{ by setting } k_j = \frac{q_j}{\|q_j\|}$$



Leave out self attention, otherwise it will be largest value

But LSH lose some “information” in each layers

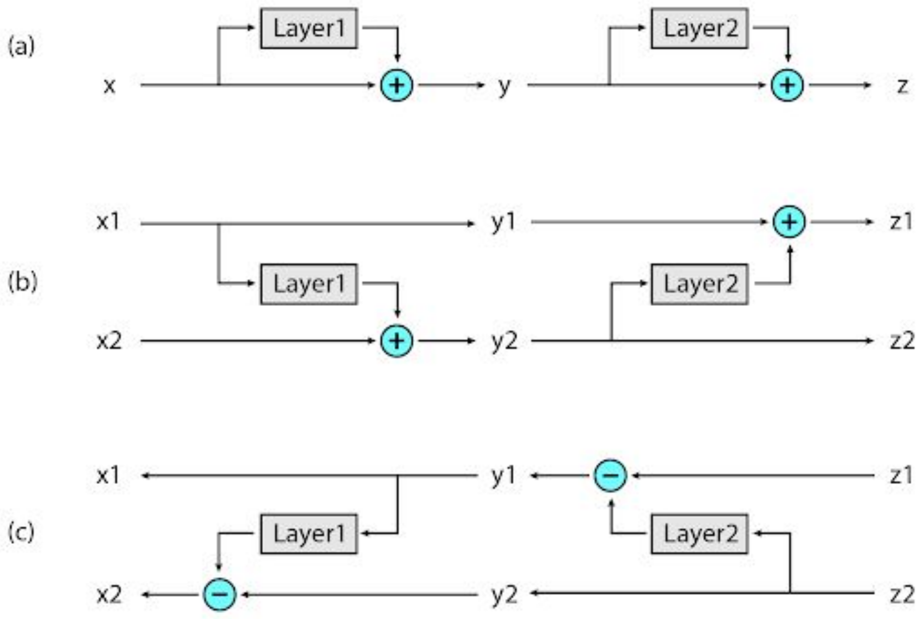
Backpropagation updates for a given weight need its own input and output to compute.



Reversible Layers

RevNet tries to recover inputs from the next layers (recover y from z)

We can make transformer reversible as well



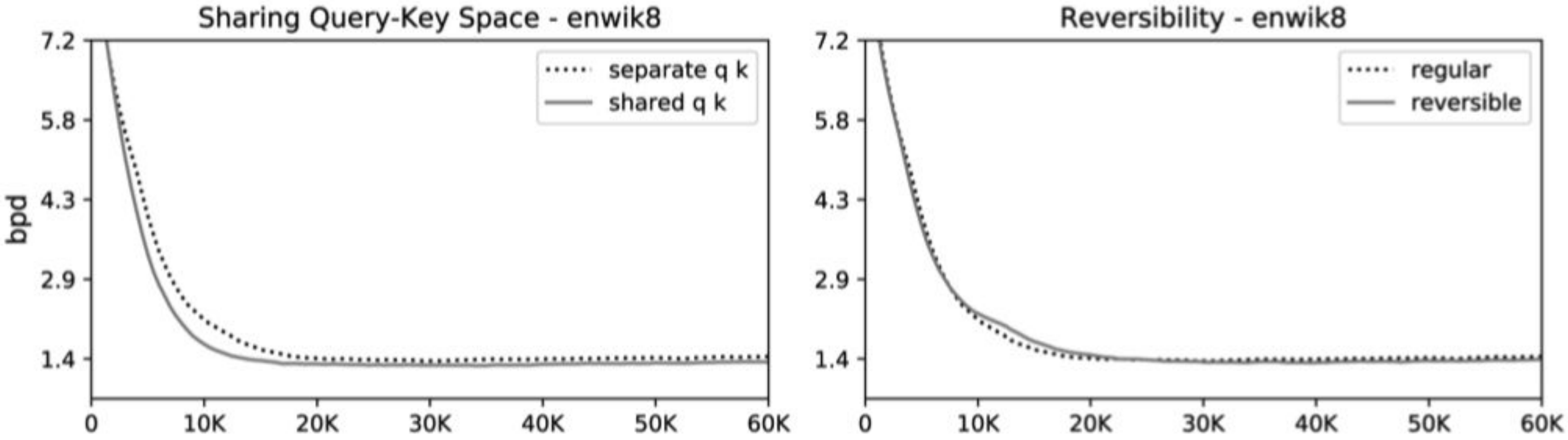
$$Y_1 = X_1 + \text{Attention}(X_2) \qquad Y_2 = X_2 + \text{FeedForward}(Y_1) \qquad (9)$$

$$Y_2 = \left[Y_2^{(1)}; \ldots ; Y_2^{(c)}\right] = \left[X_2^{(1)} + \text{FeedForward}(Y_1^{(1)}); \ldots ; X_2^{(c)} + \text{FeedForward}(Y_1^{(c)})\right] \quad (10)$$

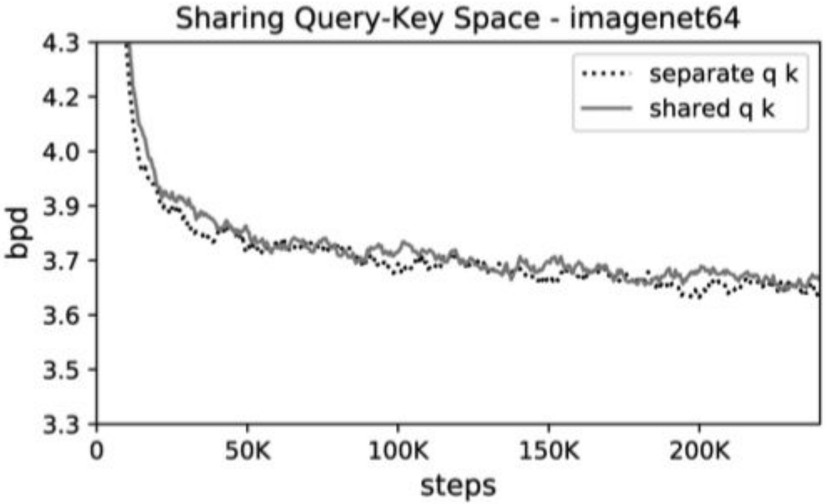
RevNet : The reversible residual network: Backpropagation without storing activations
Implementation here: https://github.com/lucidrains/reformer-pytorch/blob/master/reformer_pytorch/reversible.py

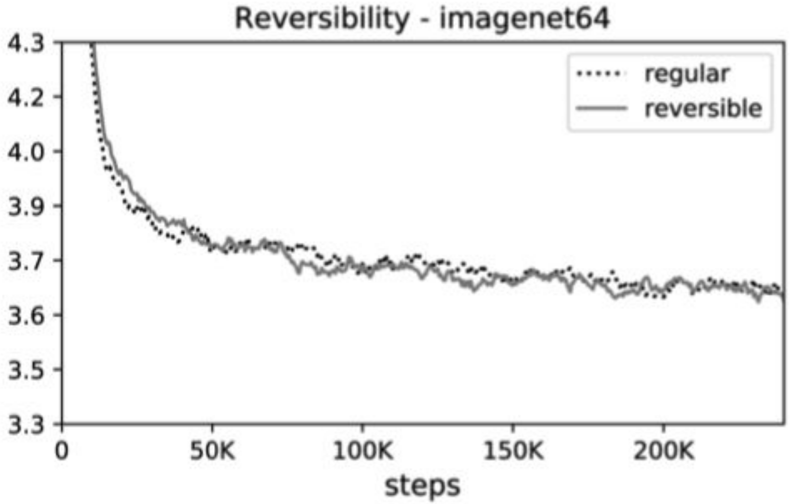
Ablation study for Q=K, Reversibility

Bpd: bits per dim, basically negative log likelyhood of outputs



Ablation study for Q=K, Reversibility





Memory, Time complexity

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 $c = 128^2$.

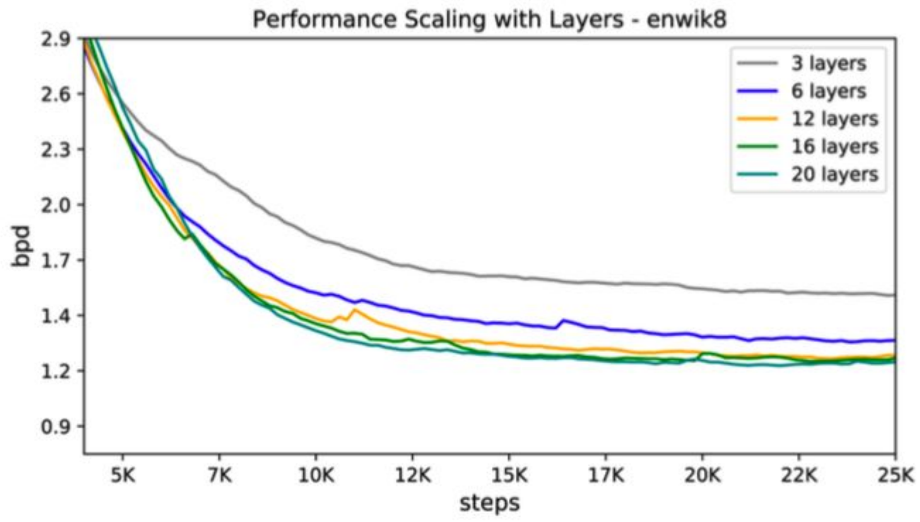
Model Type	Memory Complexity	Time Complexity
Transformer	$\max(bld_{ff}, bn_h l^2) n_l$	$(bld_{ff} + bn_h l^2) n_l$
Reversible Transformer	$\max(bld_{ff}, bn_h l^2)$	$(bn_h ld_{ff} + bn_h l^2) n_l$
Chunked Reversible Transformer	$\max(bld_{model}, bn_h l^2)$	$(bn_h ld_{ff} + bn_h l^2) n_l$
LSH Transformer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

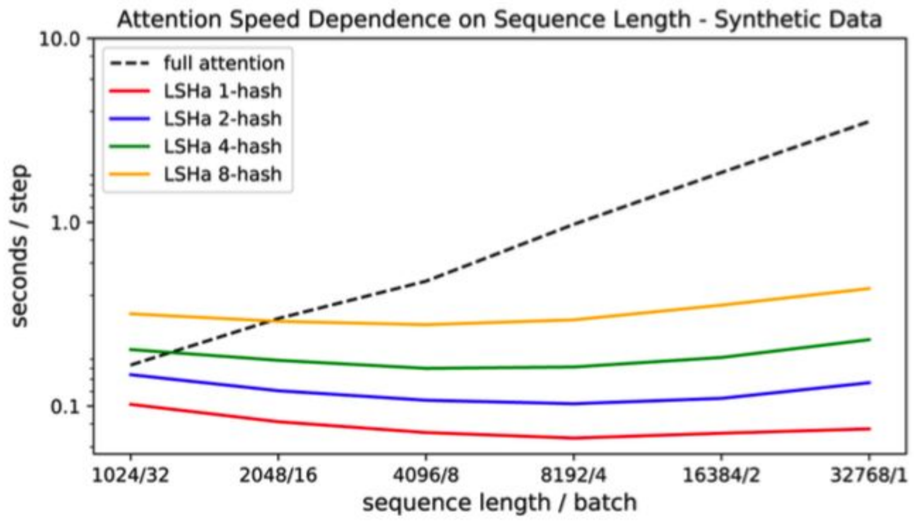
Ablation study for number of hashing rounds

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%

Performance and speed scale vs input length, layers





Take away

1. This paper use **hashing** to approximate attention matrix and introduce **reversible layers** to **save memory** at the expensive of additional computation
2. Pointed out in openreview, bucket size affects the model performance
3. Resource efficient (less training time, run in CPU) is a rising area in NLP as supervised natural language understanding performance close to near human level
    a. SuperGLUE : promote the development of effective, energy-efficient models for difficult NLU tasks.
4. This paper only shows result from auto regressive training, other existing training method such as masking, masking in sequence-to-sequence learning need to evaluate as well to obtain the full view

Questions

1. Why is shared query, key projection matrix required?
2. How do this paper reduce the self attention matrix complexity?

References

- [2001.04451] Reformer: The Efficient Transformer
- Reformer: The Efficient Transformer
- Reformer the efficient transformer - Youtube

- Code implementation : reformer-pytorch/reformer_pytorch at master · lucidrains/reformer-pytorch