Fast Attention

NLP: Fall 2024

Anoop Sarkar

FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

Tri Dao[†], Daniel Y. Fu[†], Stefano Ermon[†], Atri Rudra[‡], and Christopher Ré[†]

[†]Department of Computer Science, Stanford University
[‡]Department of Computer Science and Engineering, University at Buffalo, SUNY

https://arxiv.org/abs/2205.14135

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

https://arxiv.org/abs/2205.14135

Given the inputs $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, we aim to compute the attention output $\mathbf{O} \in \mathbb{R}^{N \times d}$ and write it to HBM. Our goal is to reduce the amount of HBM accesses (to sub-quadratic in N).

split the inputs $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ into blocks, load them from slow HBM to fast SRAM.

Tiling. We compute attention by blocks. Softmax couples columns of $\mathbf{K}$, so we decompose the large softmax with scaling [51, 60, 66]. For numerical stability, the softmax of vector $x \in \mathbb{R}^B$ is computed as:

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

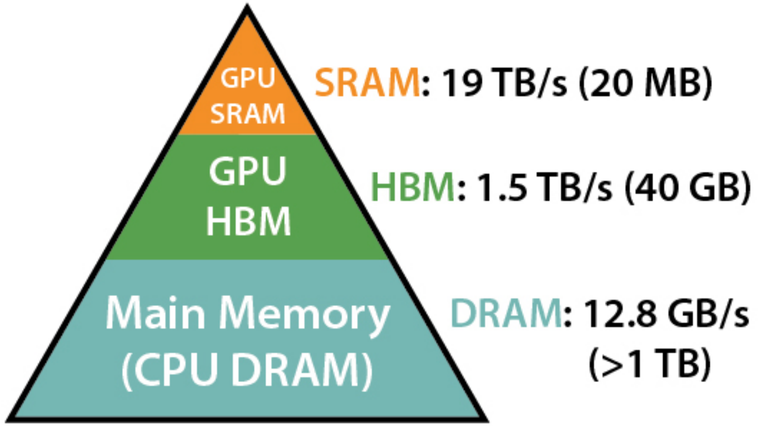
https://arxiv.org/abs/2205.14135

Algorithm 1 FLASHATTENTION

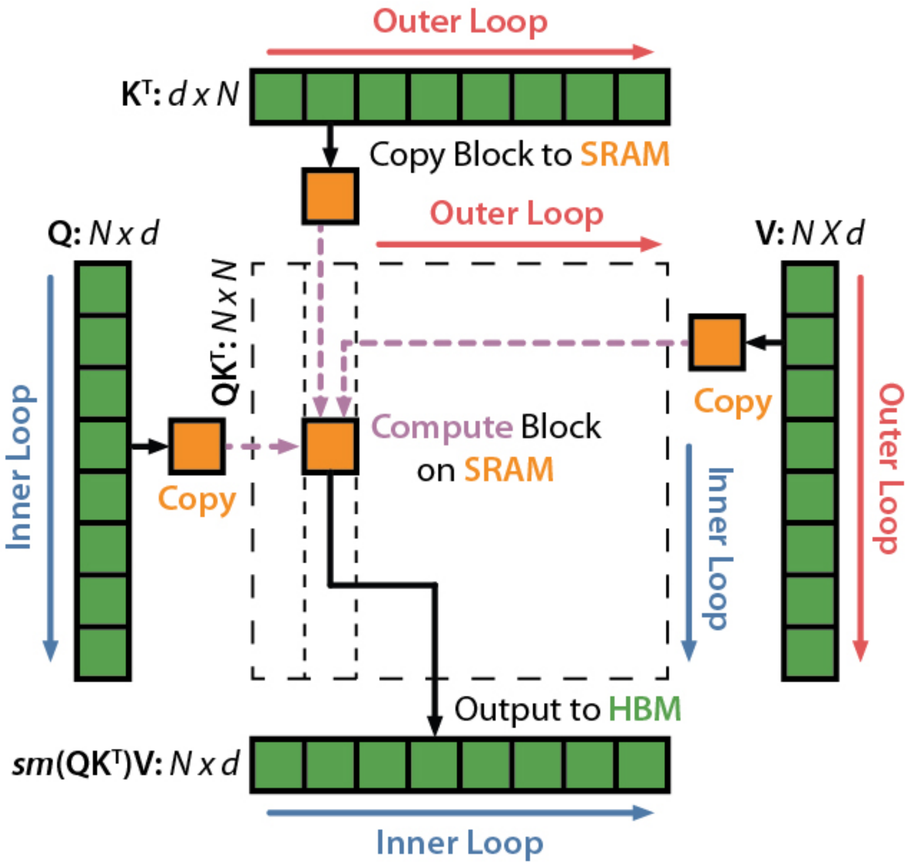
Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

1: Set block sizes $B_c = \lceil \frac{M}{4d} \rceil$, $B_r = \min\left(\lceil \frac{M}{4d} \rceil, d\right)$.
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 $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ 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 $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.
5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **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 $\mathbf{O}$.

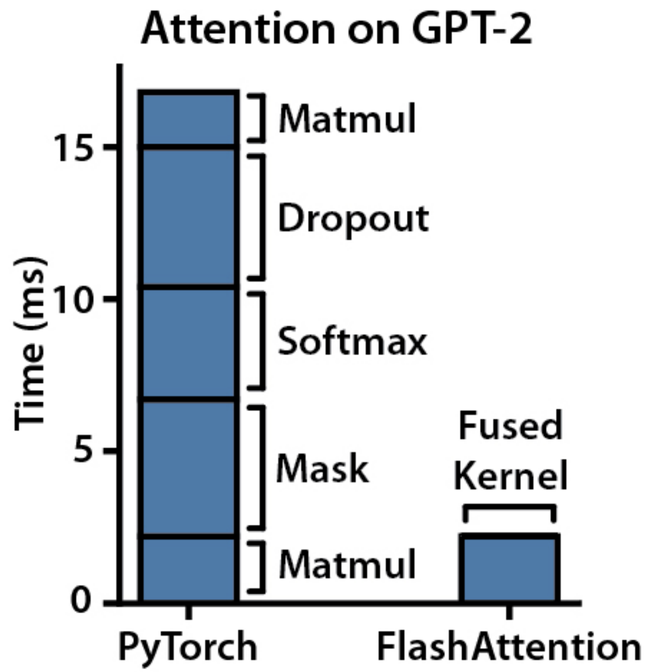
Therefore if we keep track of some extra statistics $(m(x), \ell(x))$, we can compute softmax one block at a time.[2] We thus split the inputs $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ into blocks (Algorithm 1 line 3), compute the softmax values along with extra statistics (Algorithm 1 line 10), and combine the results (Algorithm 1 line 12).



Memory Hierarchy with Bandwidth & Memory Size



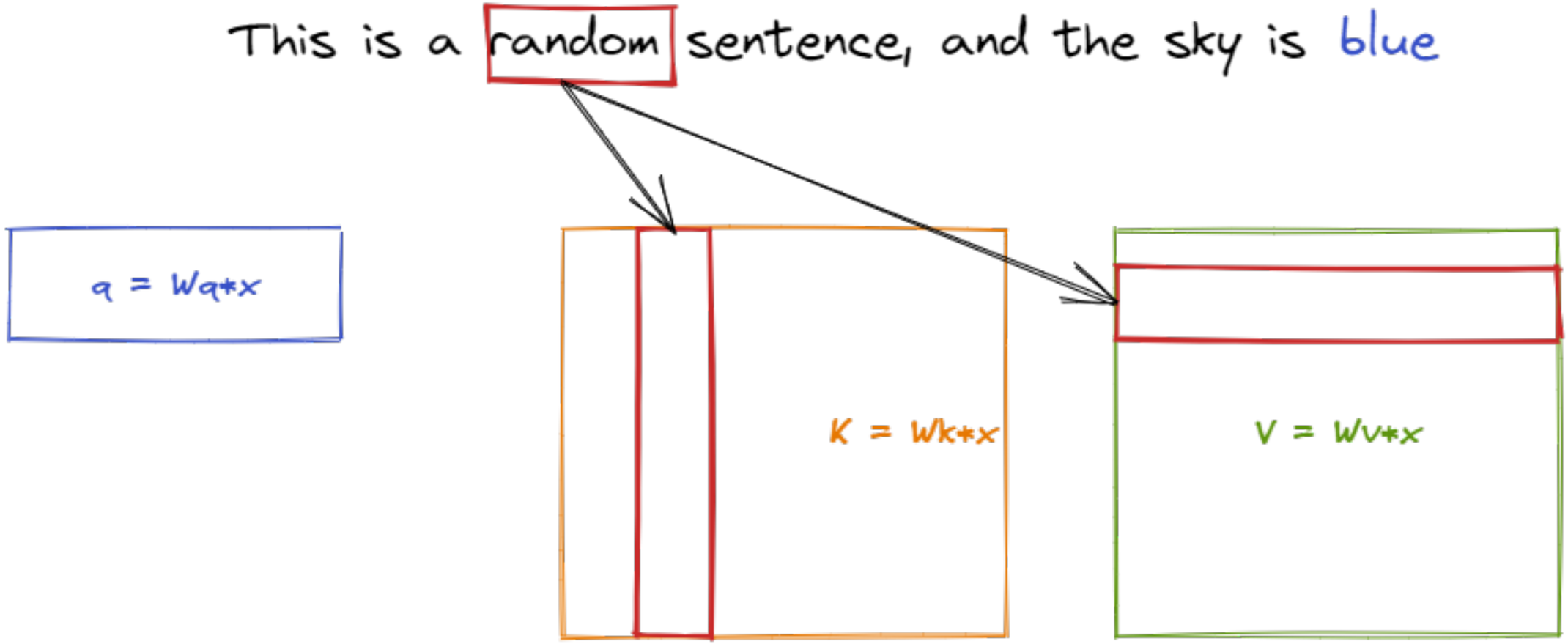
FlashAttention



https://arxiv.org/abs/2205.14135

KV cache (and paged attention)

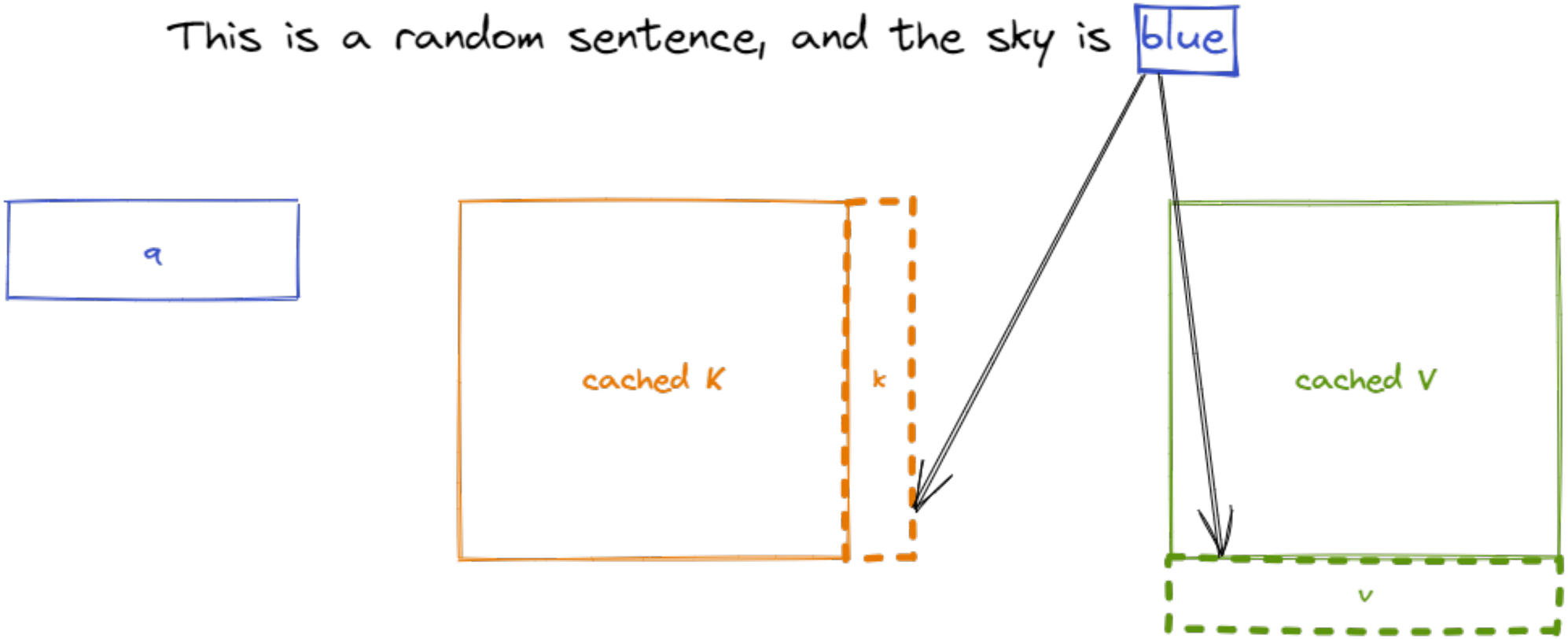
https://arxiv.org/pdf/2309.06180



Naively, for each new token (e.g. "blue" above) the key and value vectors are recomputed every time, for each new token.

K and V contain information about the entire sequence, and query is the new token being added. So we are doing *softmax*(qK)V to compute attention.

https://mett29.github.io/posts/kv-cache/



During the sequence generation one token at a time, the two matrices and do not change very much

Once the embedding for the new token is computed, it’s not going to change, no matter how many more tokens we generate

That is why the key and value vectors of existing tokens are often cached for generating future tokens. This approach leads to what is called the **KV cache**.

https://mett29.github.io/posts/kv-cache/

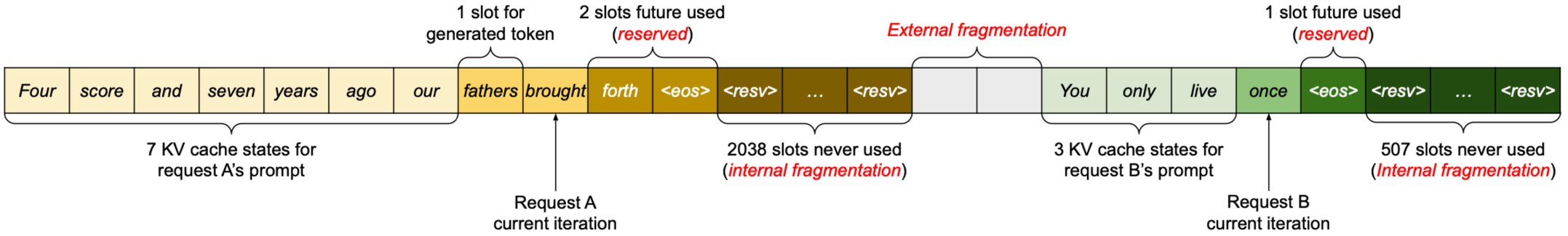


Figure 3. KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions.

https://arxiv.org/pdf/2309.06180

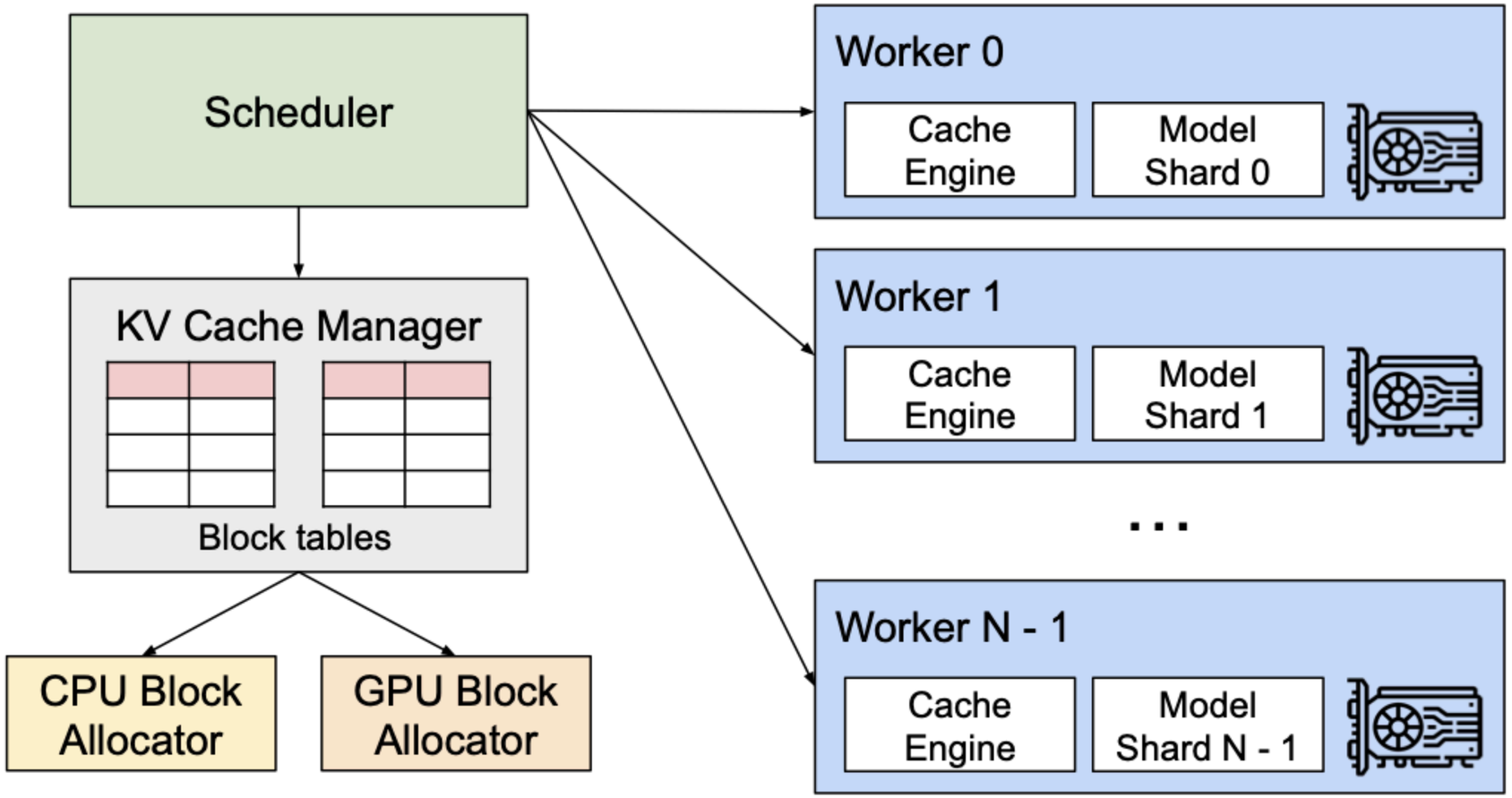


Figure 4. vLLM system overview.

https://arxiv.org/pdf/2309.06180

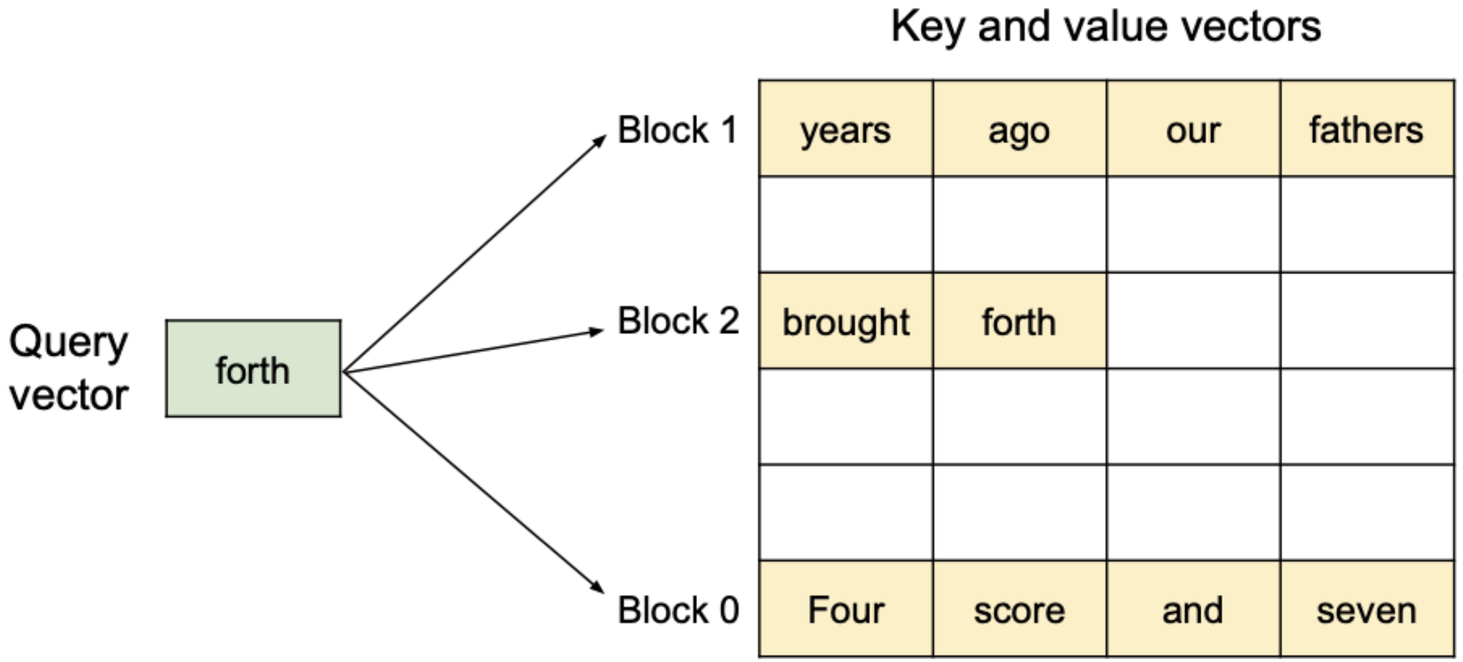


Figure 5. Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory.

https://arxiv.org/pdf/2309.06180

REFORMER: THE EFFICIENT TRANSFORMER

Nikita Kitaev*
U.C. Berkeley & Google Research
kitaev@cs.berkeley.edu

Łukasz Kaiser*
Google Research

Anselm Levskaya
Google Research
{lukaszkaiser,levskaya}@google.com

https://openreview.net/forum?id=rkgNKkHtvB

https://iclr.cc/virtual_2020/poster_rkgNKkHtvB.html

Transformer

From the BERT documentation:

> Using the default training scripts (`run_classifier.py` and `run_squad.py`), we benchmarked the maximum batch size on single Titan X GPU (12GB RAM) with TensorFlow 1.11.0:

System	Seq Length	Max Batch Size
`BERT-Large`	64	12
...	128	6
...	256	2
...	320	1
...	384	0
...	512	0

Outlook

In the near future, it will be impossible to even fine tune state of the art models without datacenter-scale hardware resources.

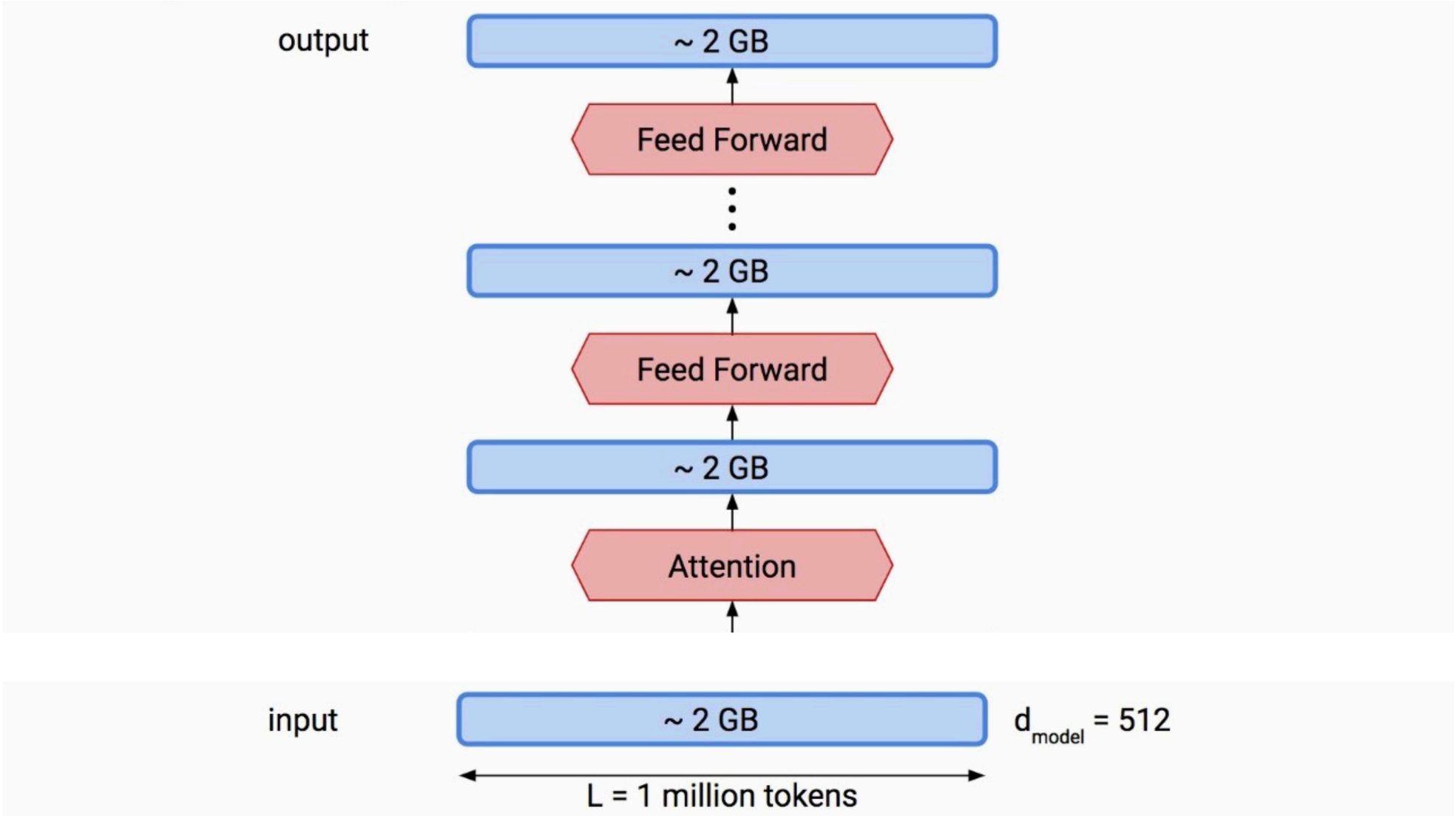
Transformers can be adapted to run on today's hardware over entire chapters or documents of text -- up to 1 million tokens at a time.

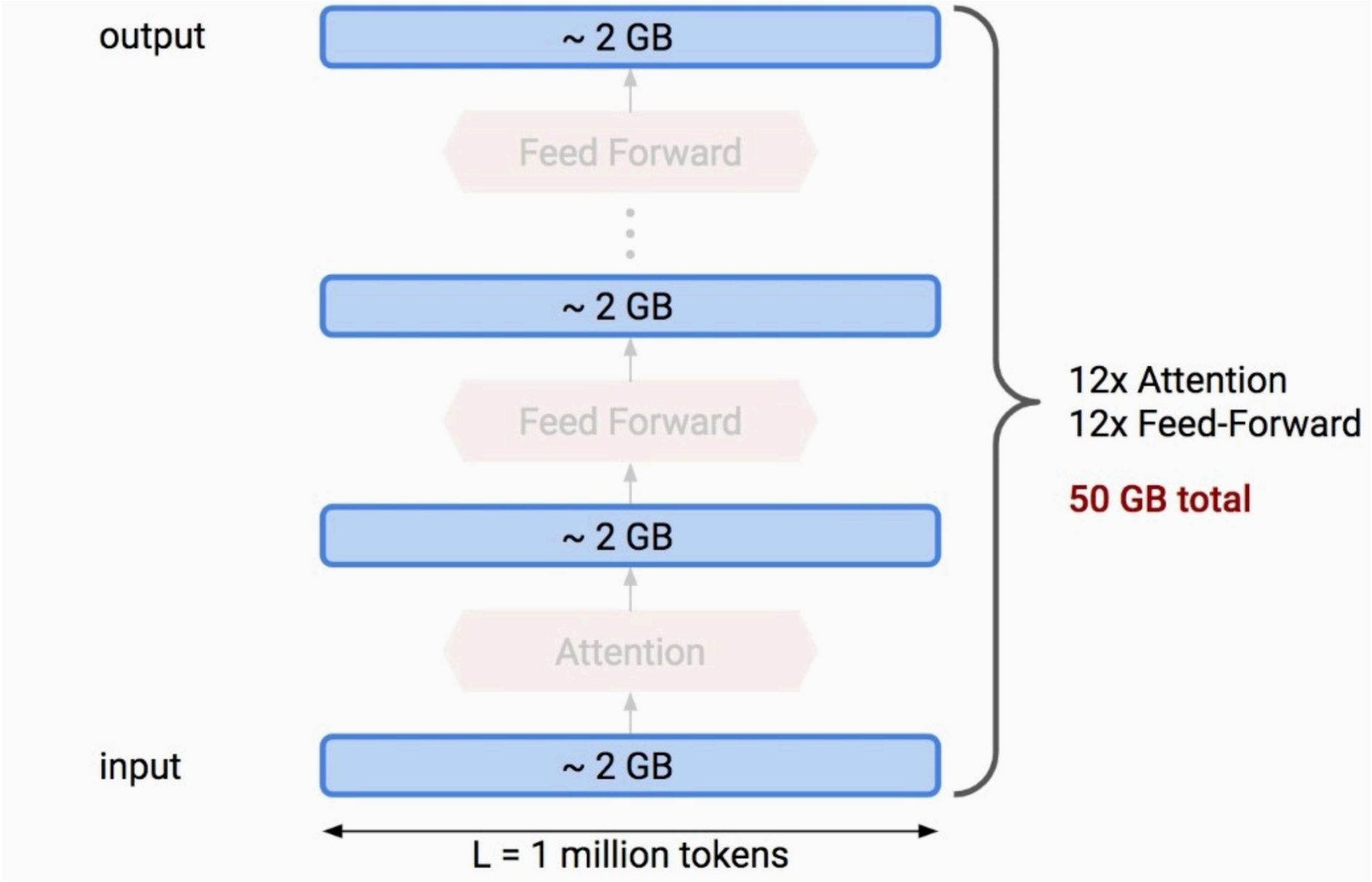
Moreover, the model should run on a single GPU or TPU device.

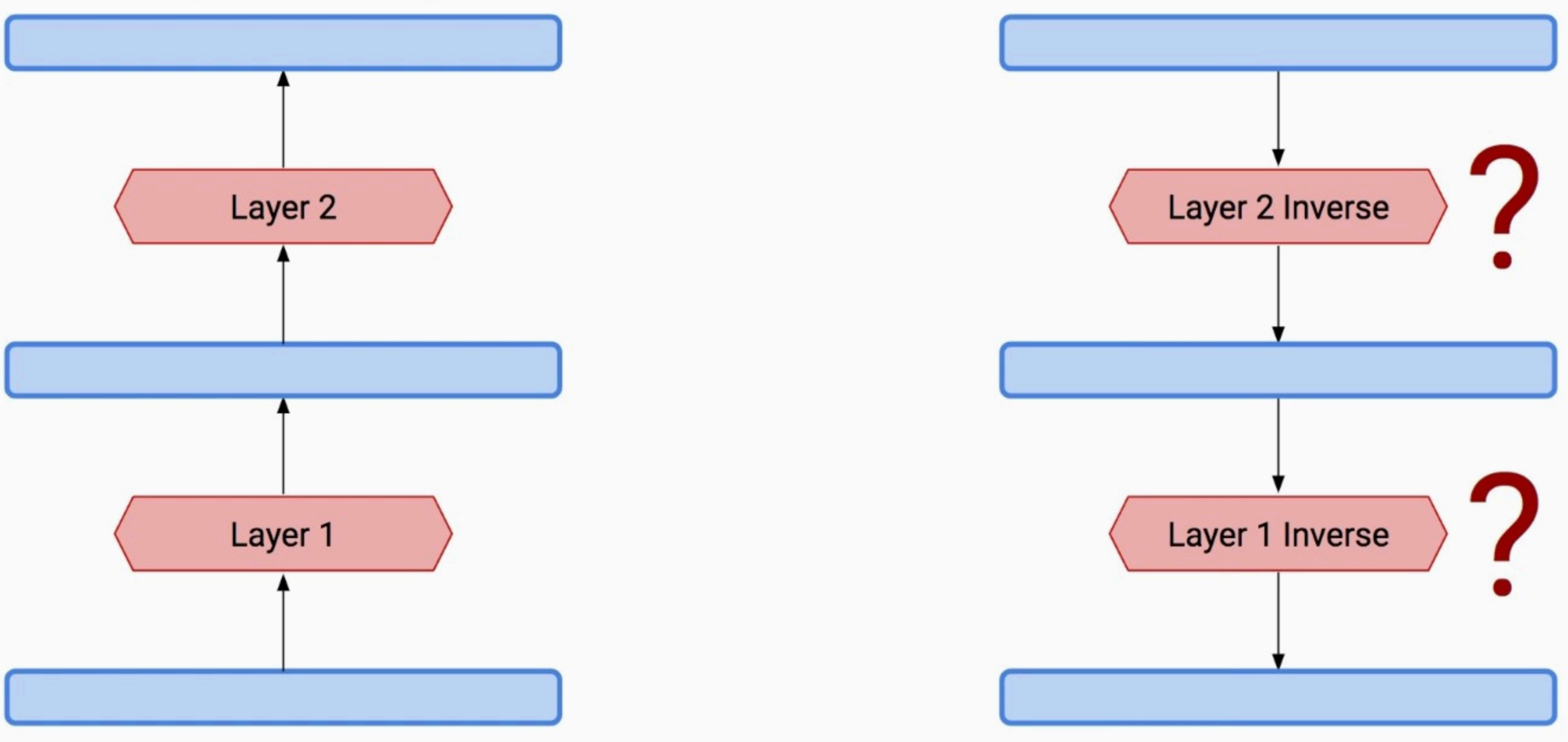
Efficiency Challenges

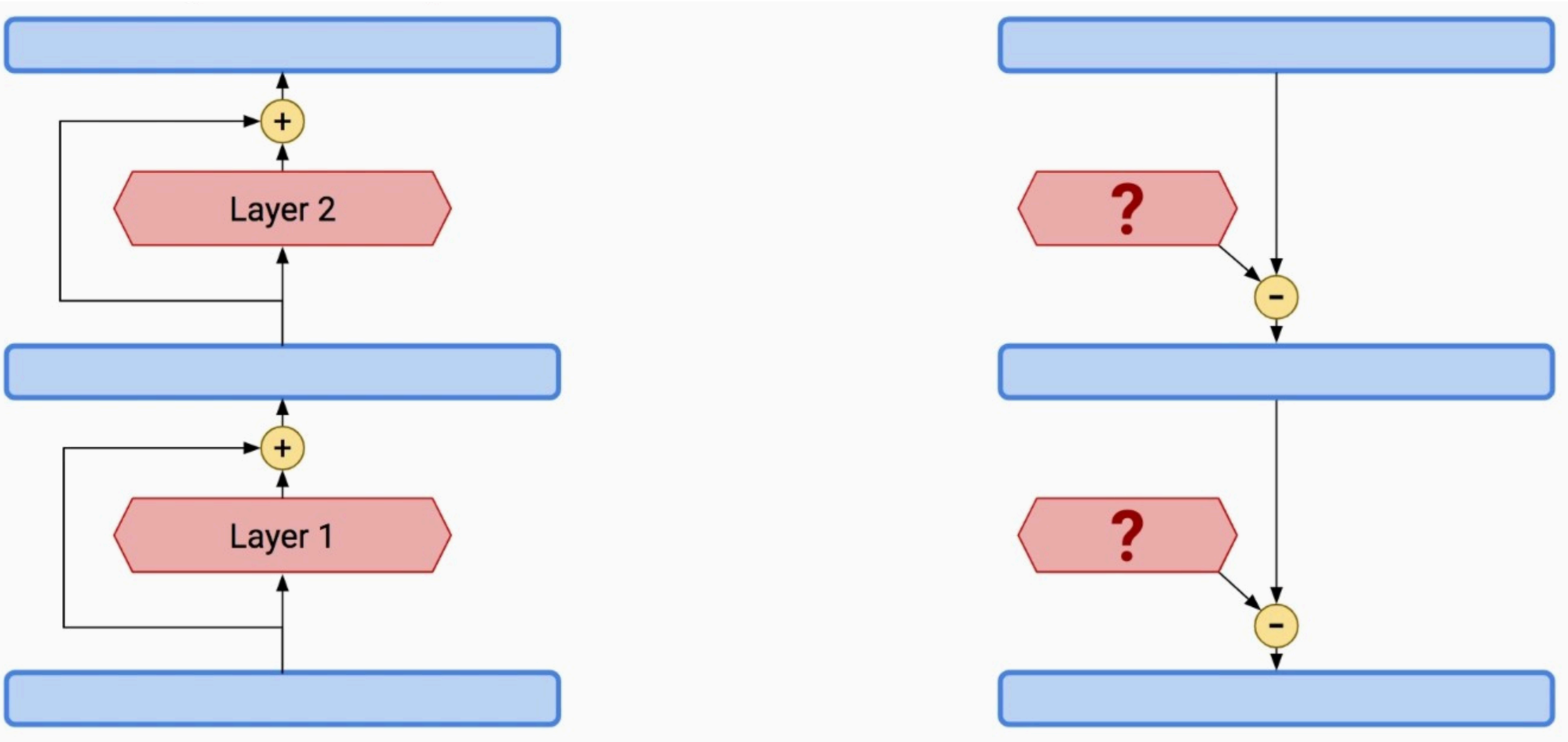
- Memory Efficiency
  - Reduce memory usage with reversible residual layers, as in RevNet [Gomez+ 17]

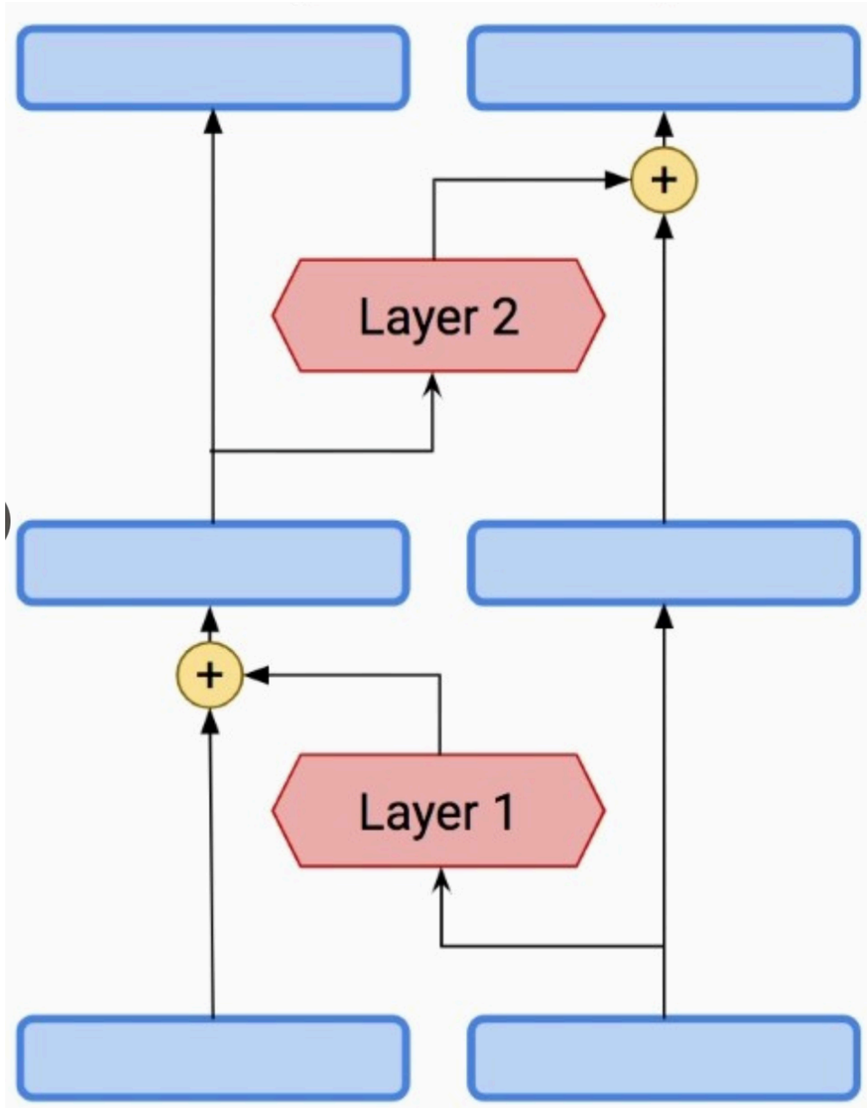
- Time Complexity
  - Introduce fast attention with locality sensitive hashing (LSH)

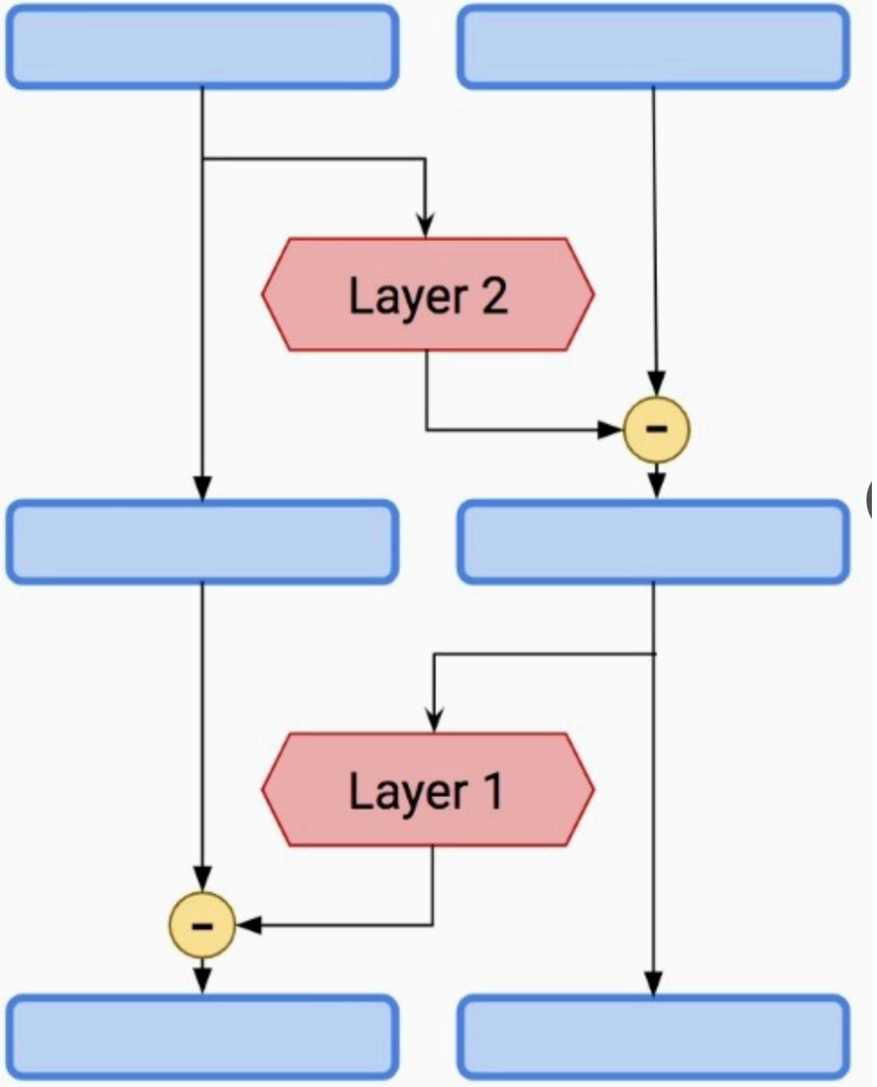

output
~ 2 GB
Feed Forward
~ 2 GB
Feed Forward
~ 2 GB
Attention
input
~ 2 GB
d_{model} = 512
L = 1 million tokens

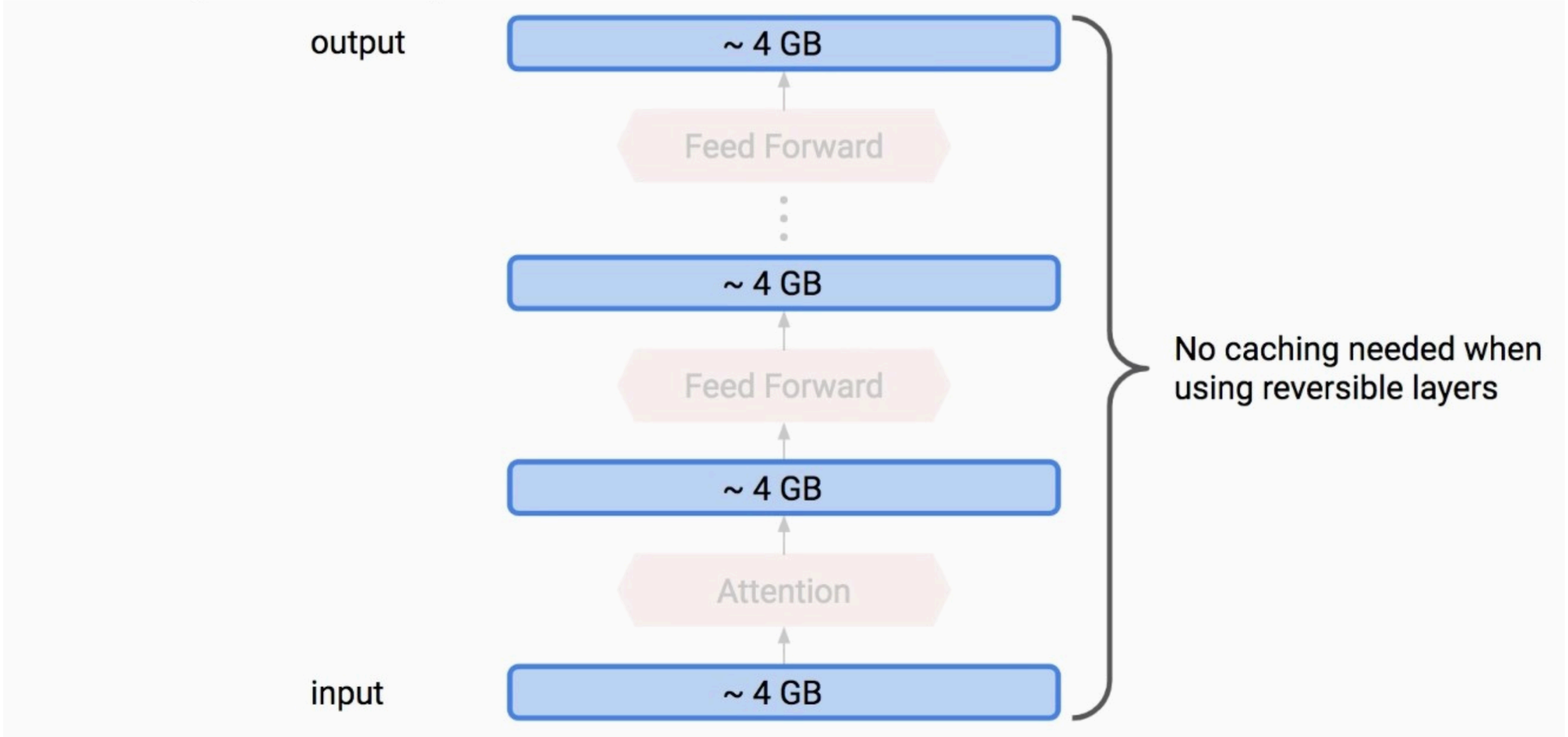

output
~ 2 GB
Feed Forward
~ 2 GB
Feed Forward
~ 2 GB
Attention
input
~ 2 GB
L = 1 million tokens
12x Attention
12x Feed-Forward
50 GB total


Layer 2
Layer 1
Layer 2 Inverse
?
Layer 1 Inverse
?

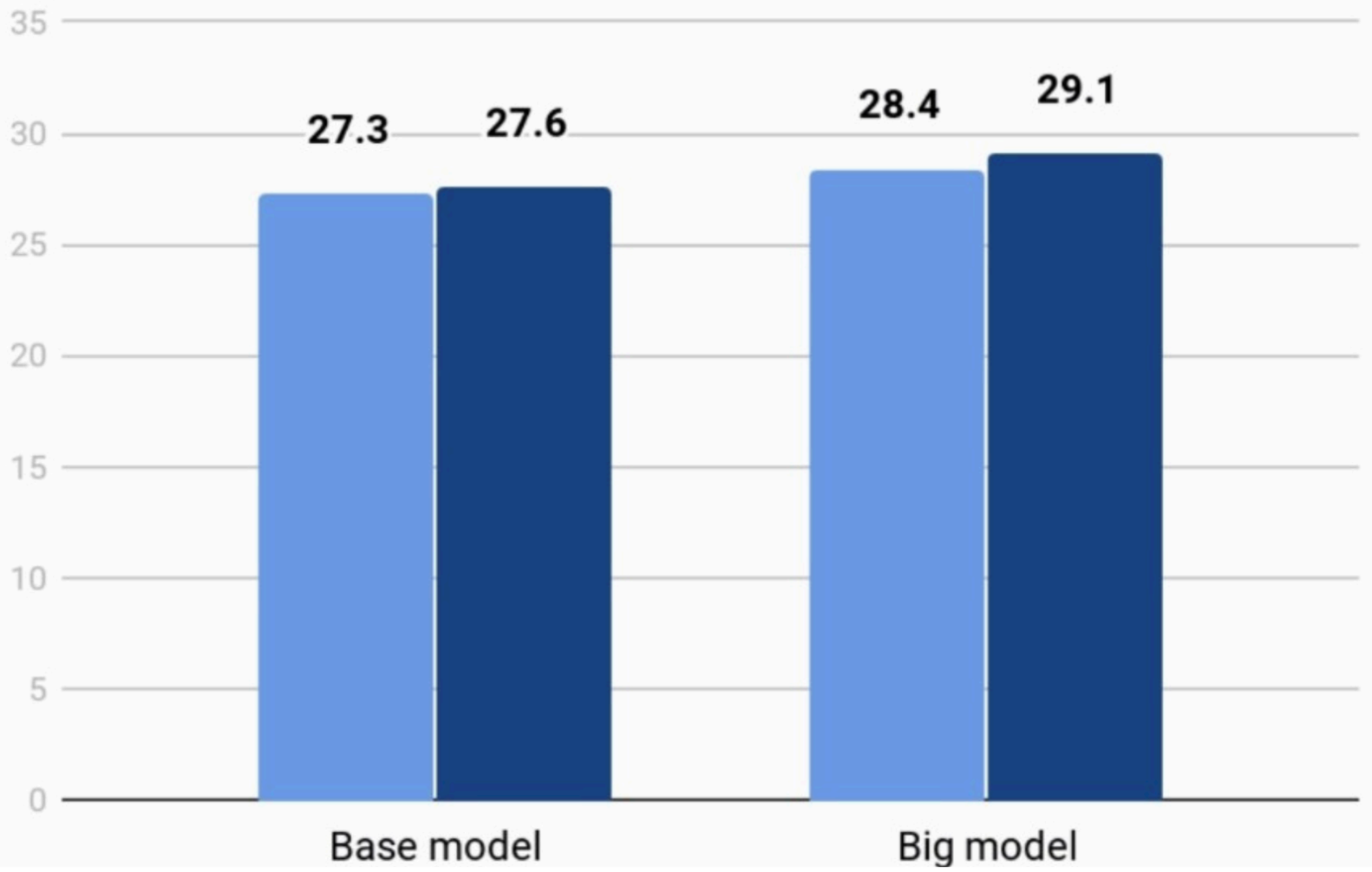

+
Layer 2
+
Layer 1
?
-
?
-


Layer 2
Layer 1

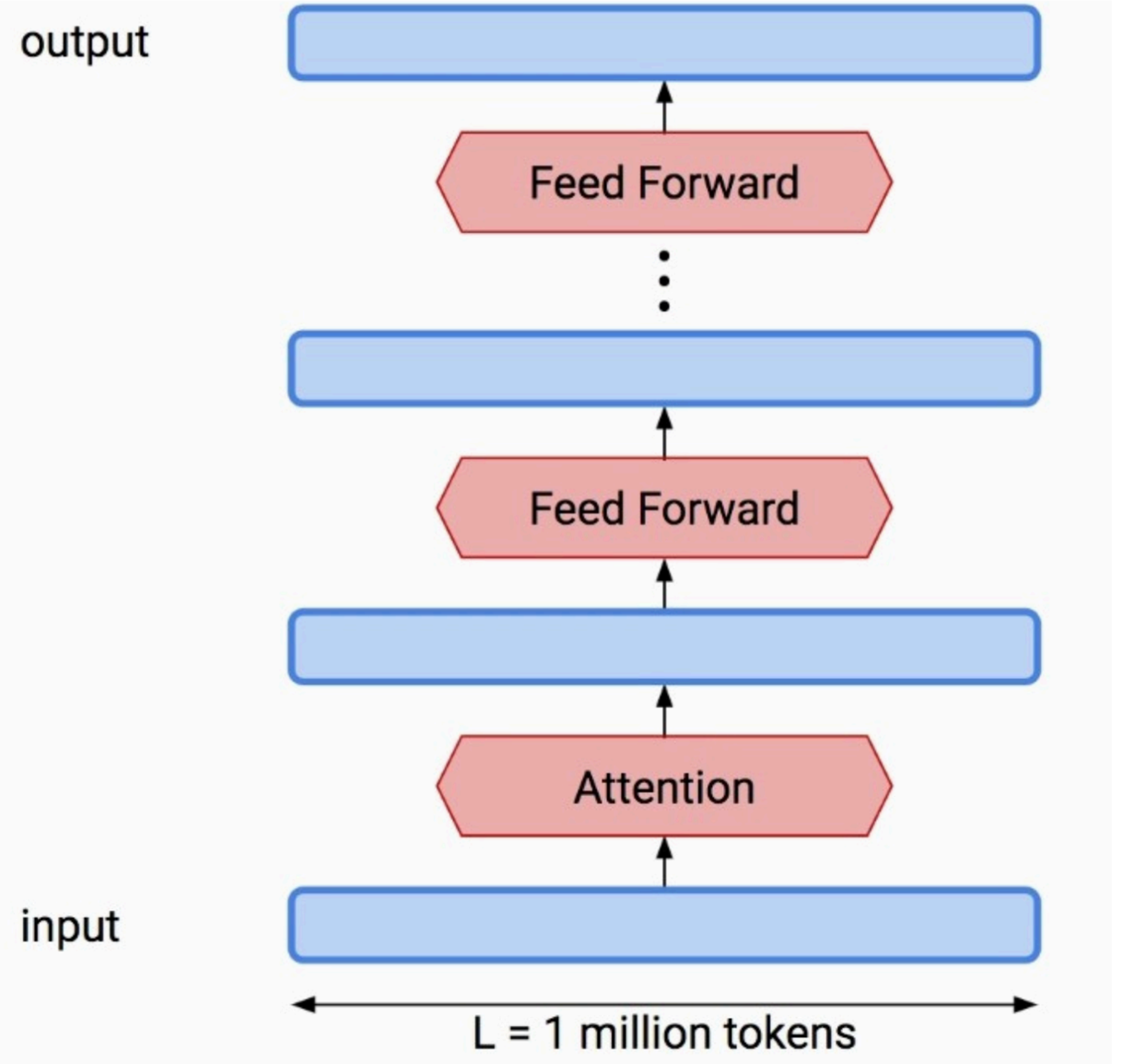

Layer 2
Layer 1

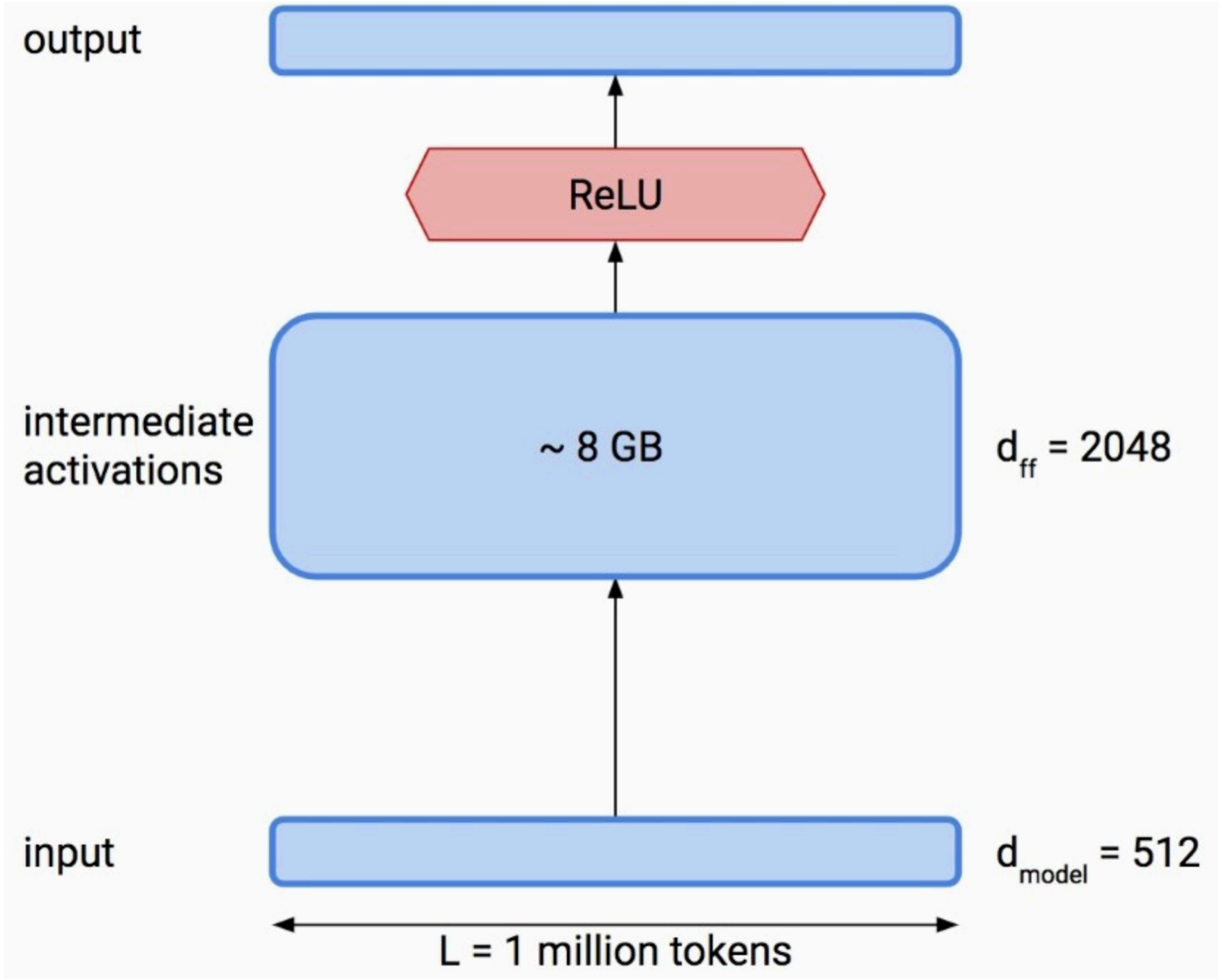

output
~ 4 GB
Feed Forward
~ 4 GB
Feed Forward
~ 4 GB
Attention
input
~ 4 GB
No caching needed when
using reversible layers

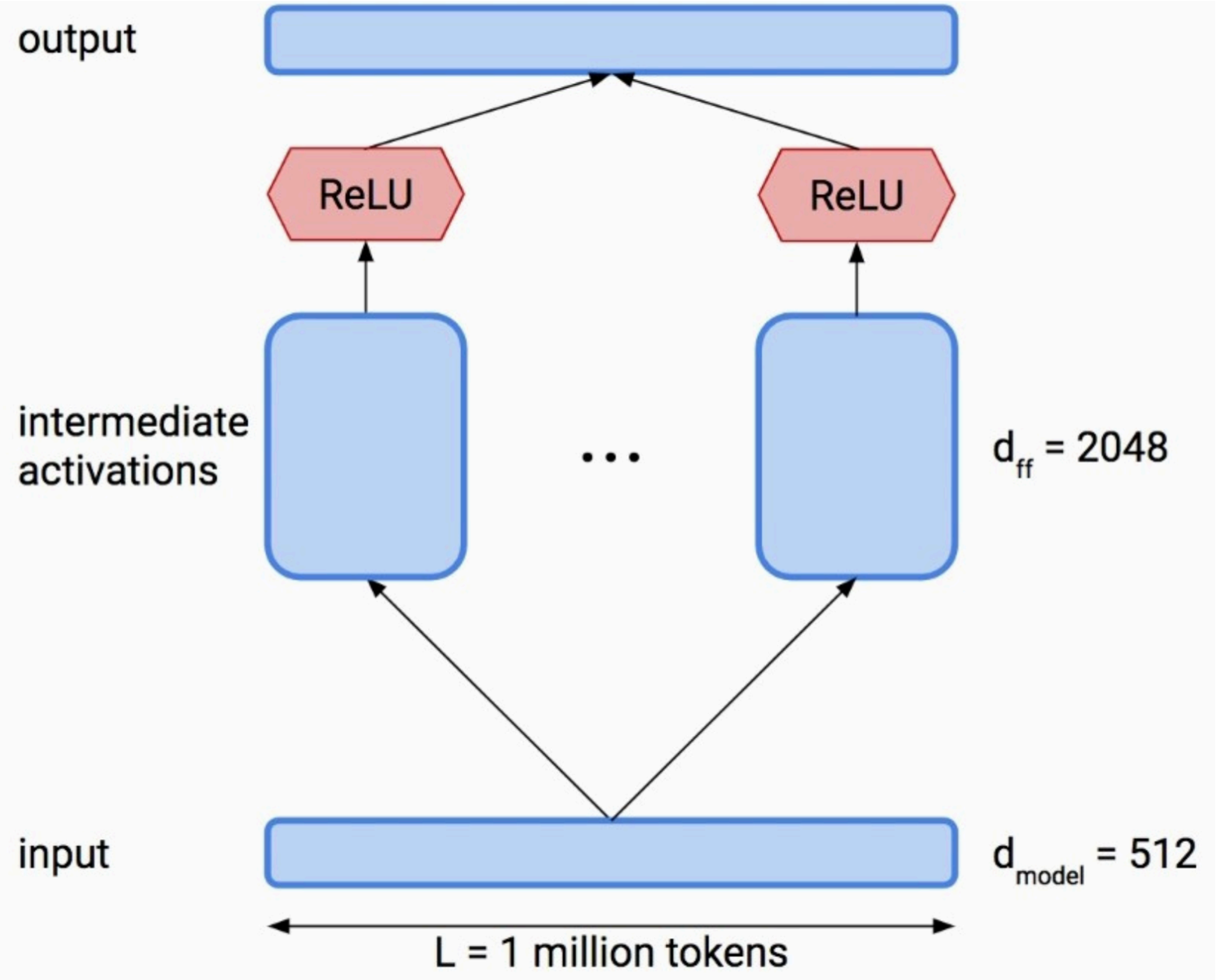
Reversible Transformer: BLEU Scores on WMT English-German

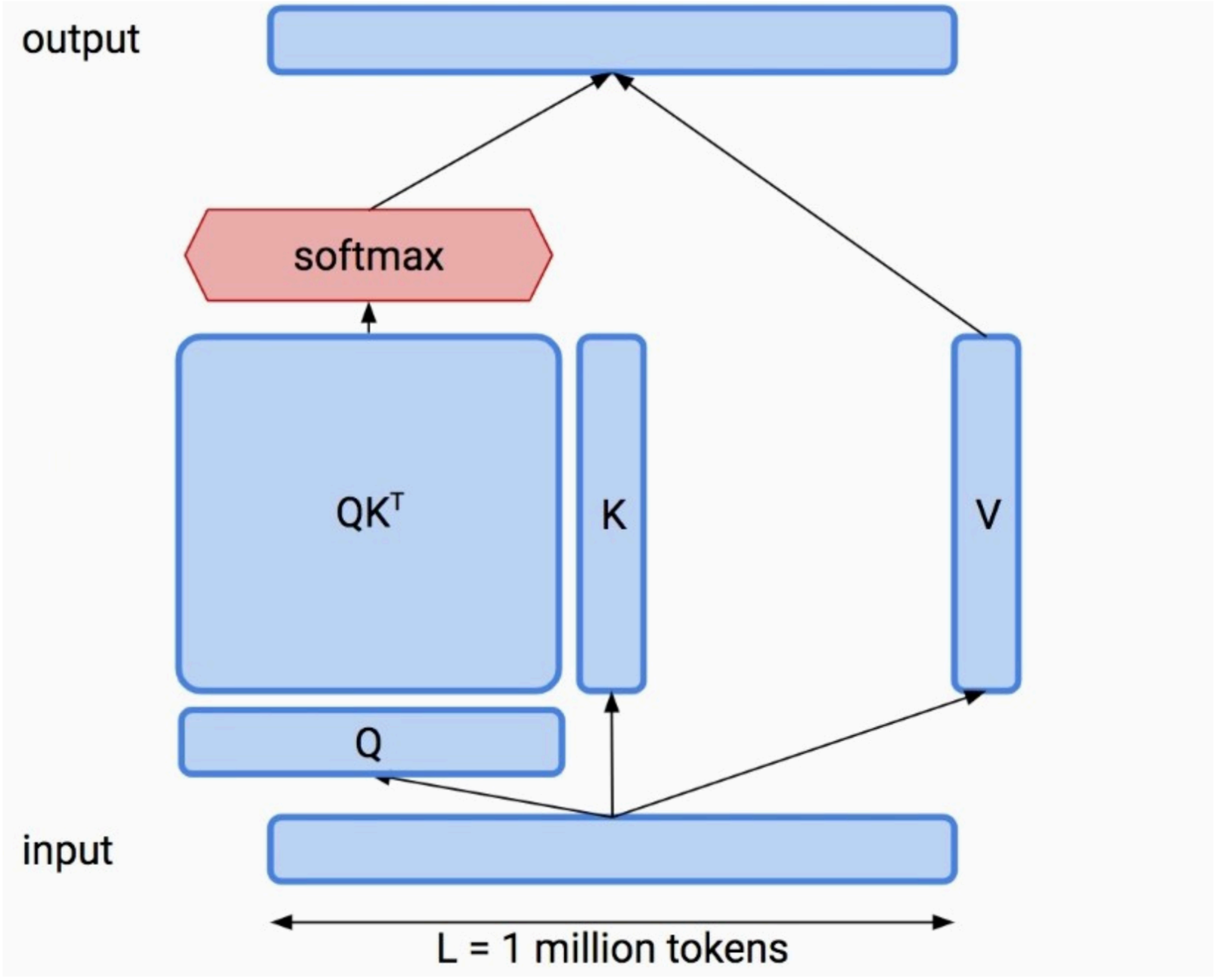


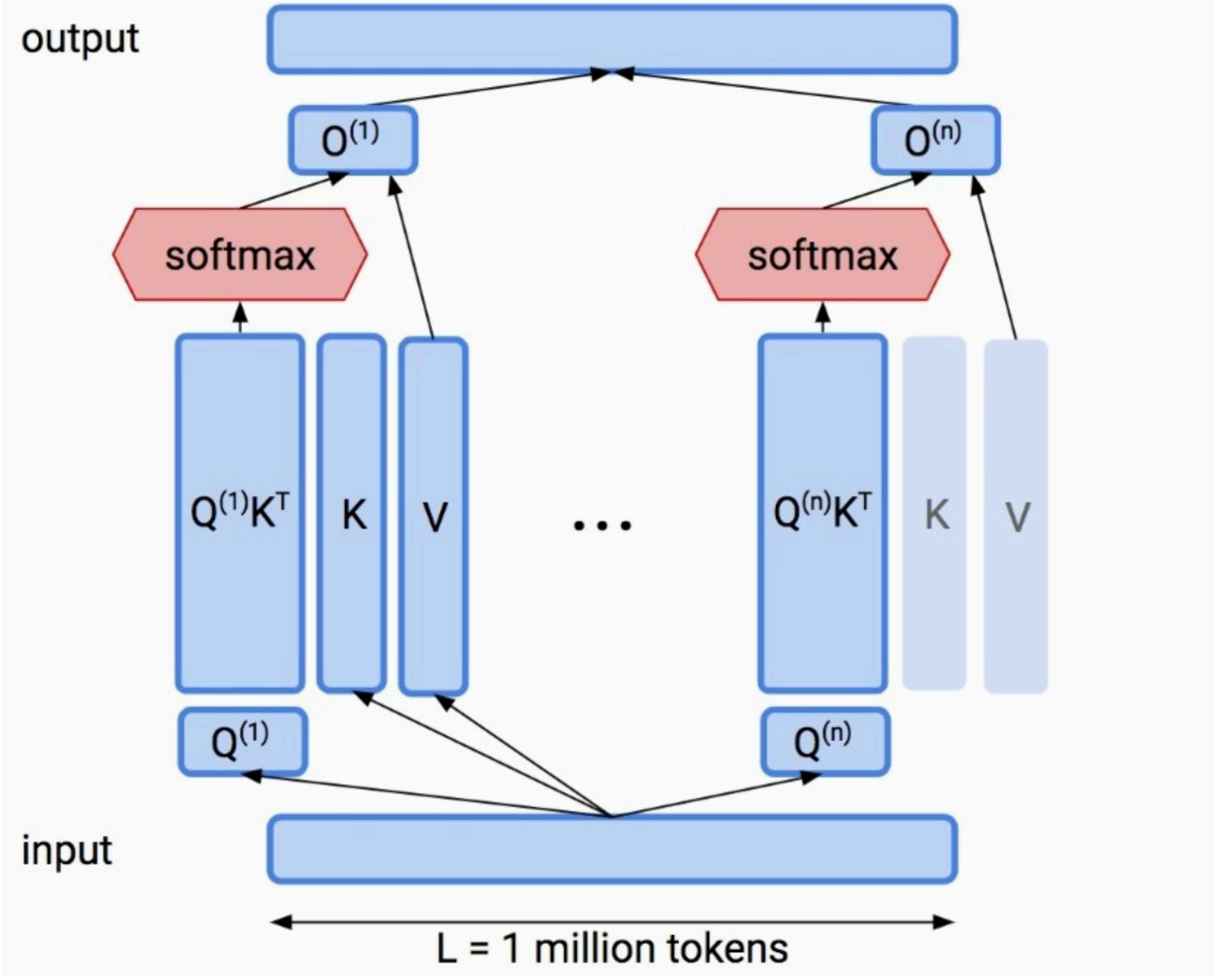
* original reported numbers; differences in BLEU from the Reversible Transformer are likely due to hyperparameter tuning

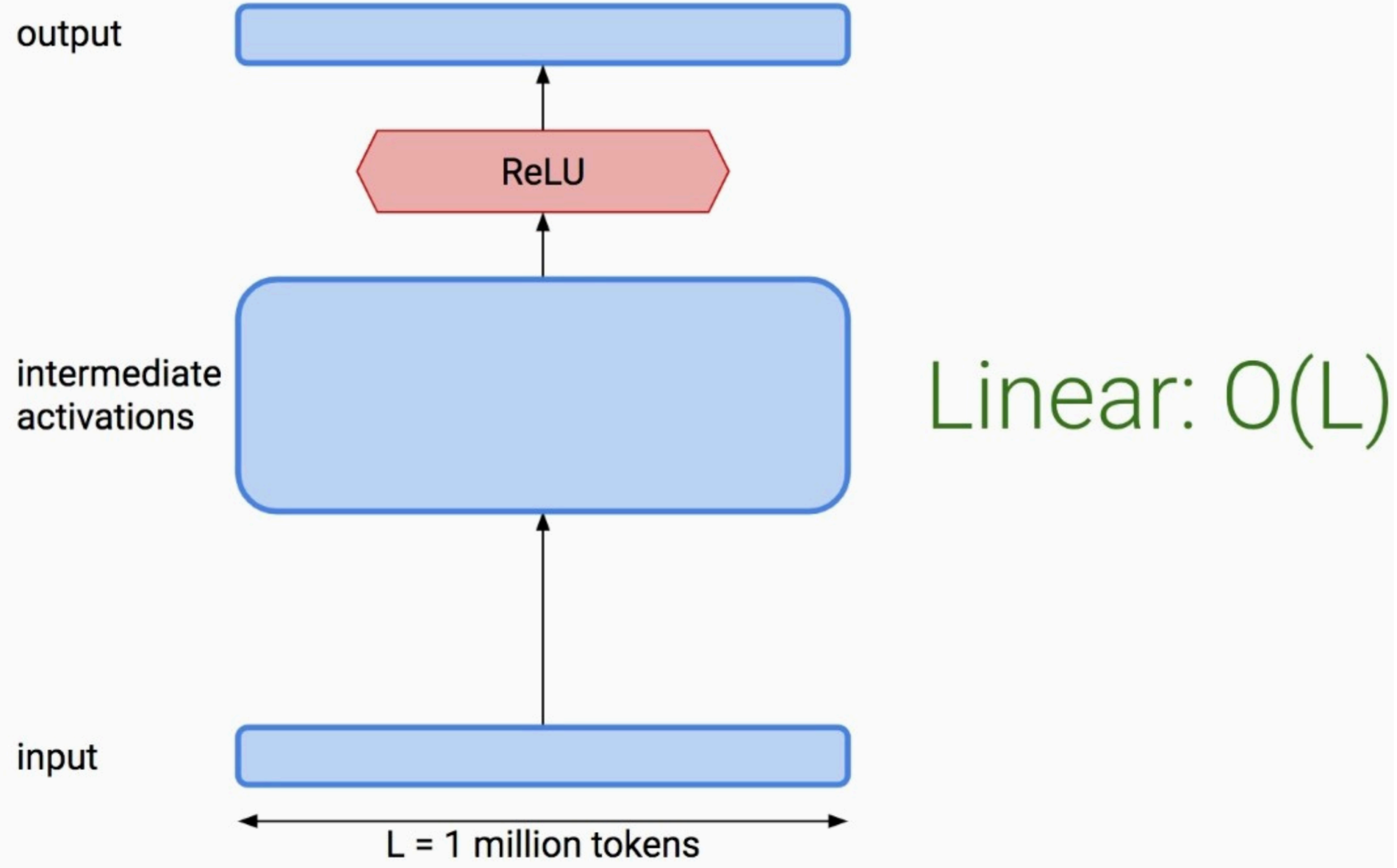

output
Feed Forward
Feed Forward
Attention
input
L = 1 million tokens


output
ReLU
intermediate activations
~ 8 GB
d_{ff} = 2048
input
d_{model} = 512
L = 1 million tokens

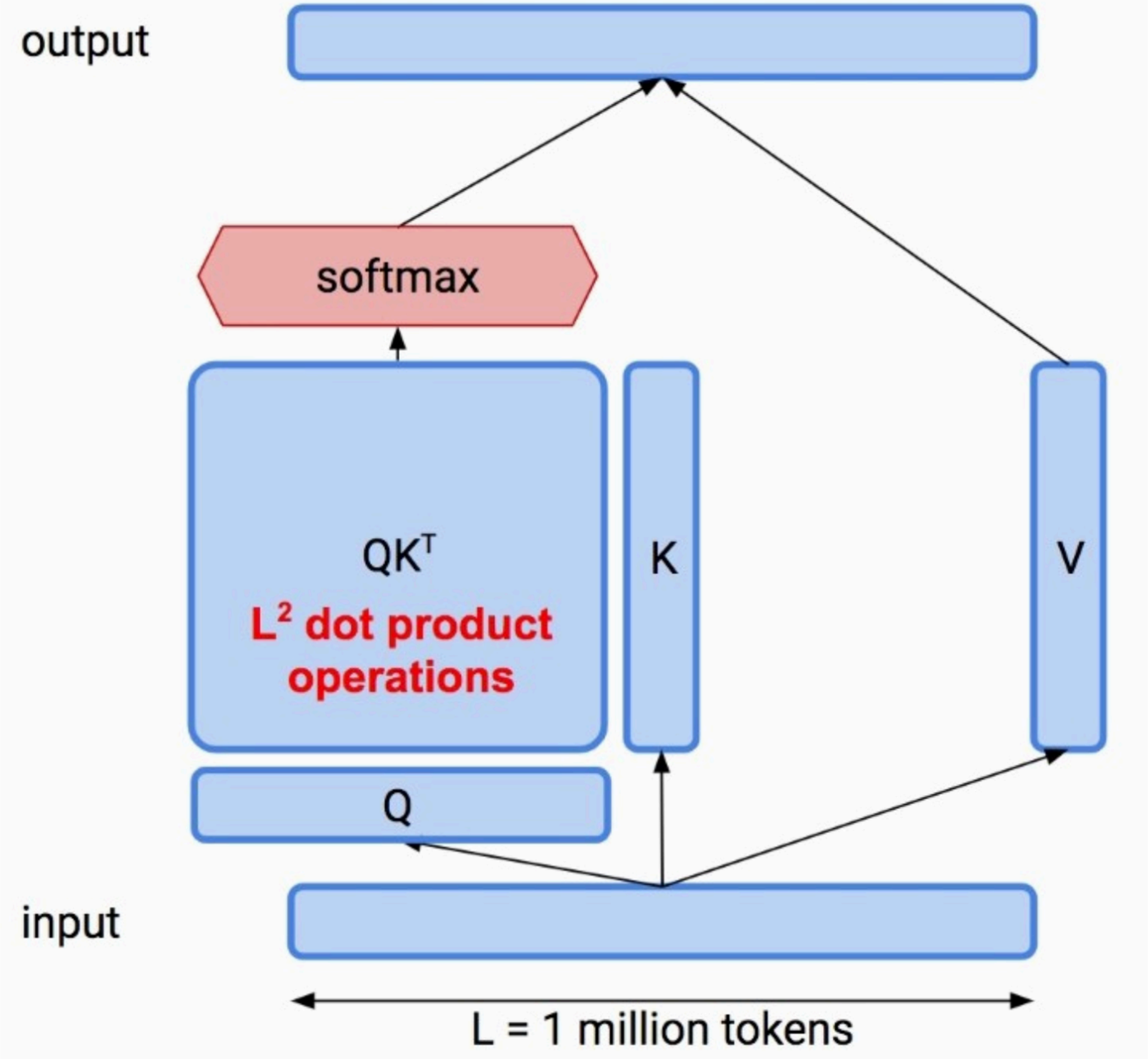

output
ReLU
ReLU
intermediate activations
...
d_{ff} = 2048
input
d_{model} = 512
L = 1 million tokens

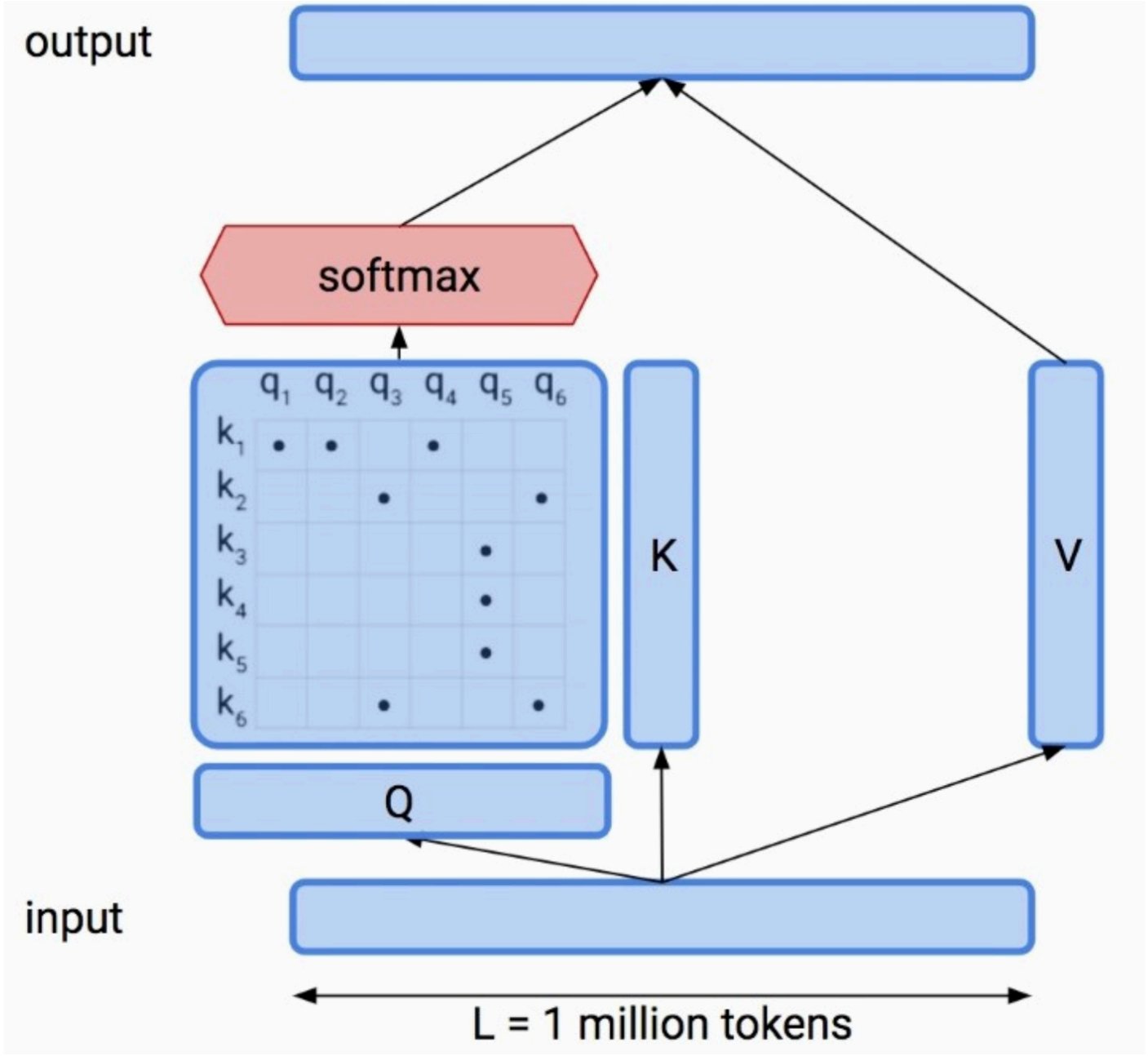

output
softmax
QK^T
K
V
Q
input
L = 1 million tokens

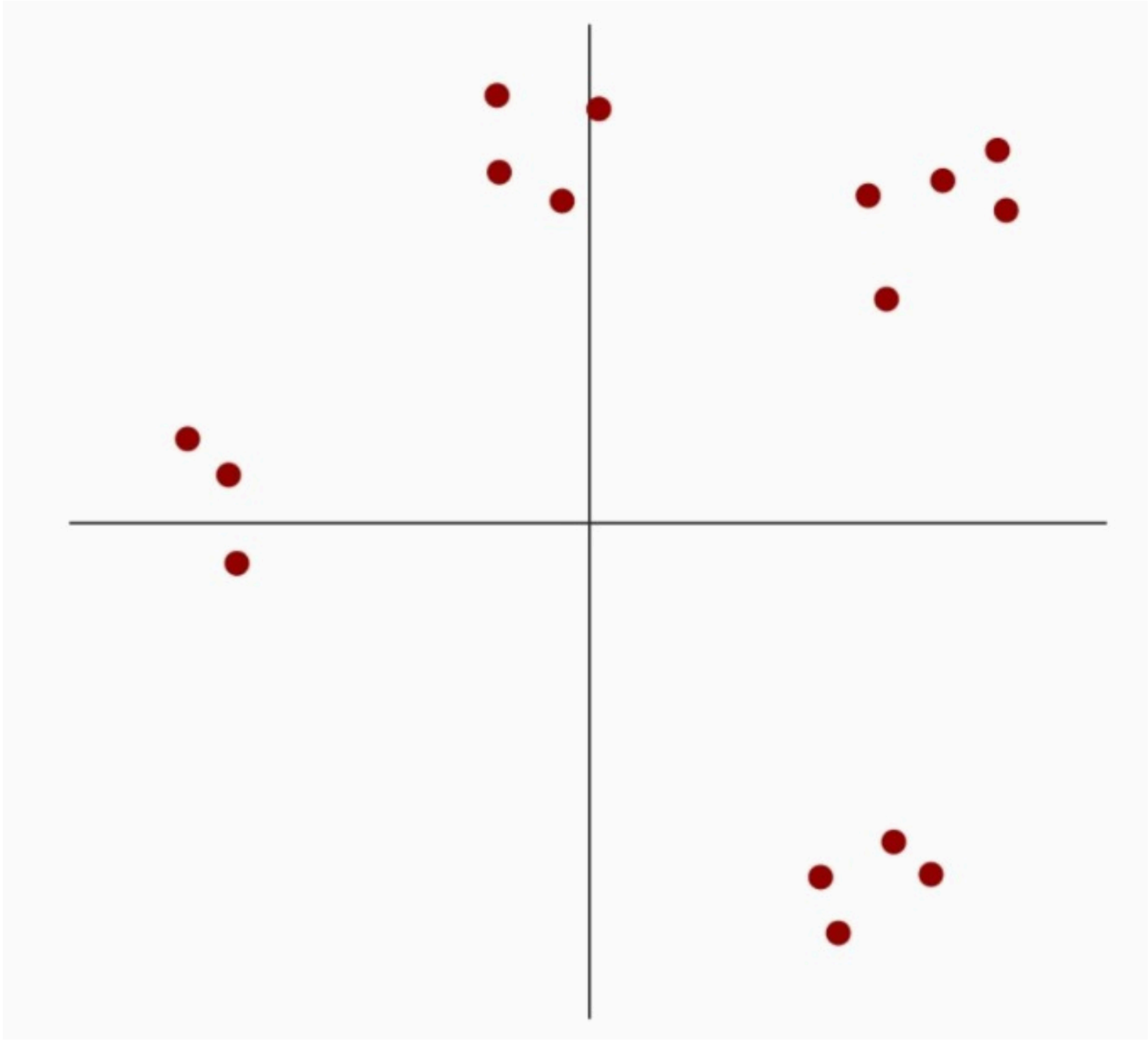

output
$O^{(1)}$
$O^{(n)}$
softmax
softmax
$Q^{(1)}K^T$
K
V
...
$Q^{(n)}K^T$
K
V
$Q^{(1)}$
$Q^{(n)}$
input
L = 1 million tokens

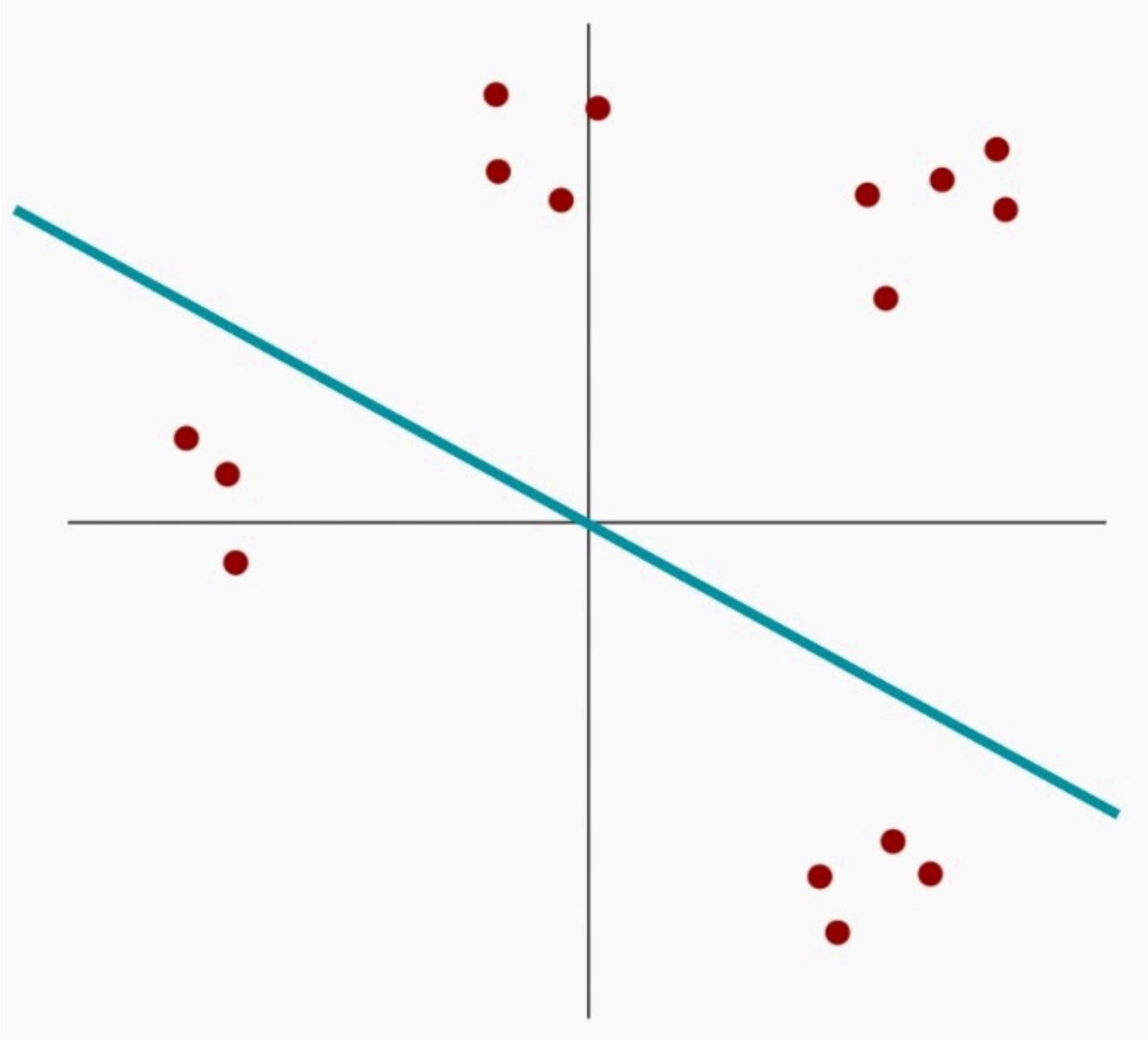

output
ReLU
intermediate activations
input
L = 1 million tokens

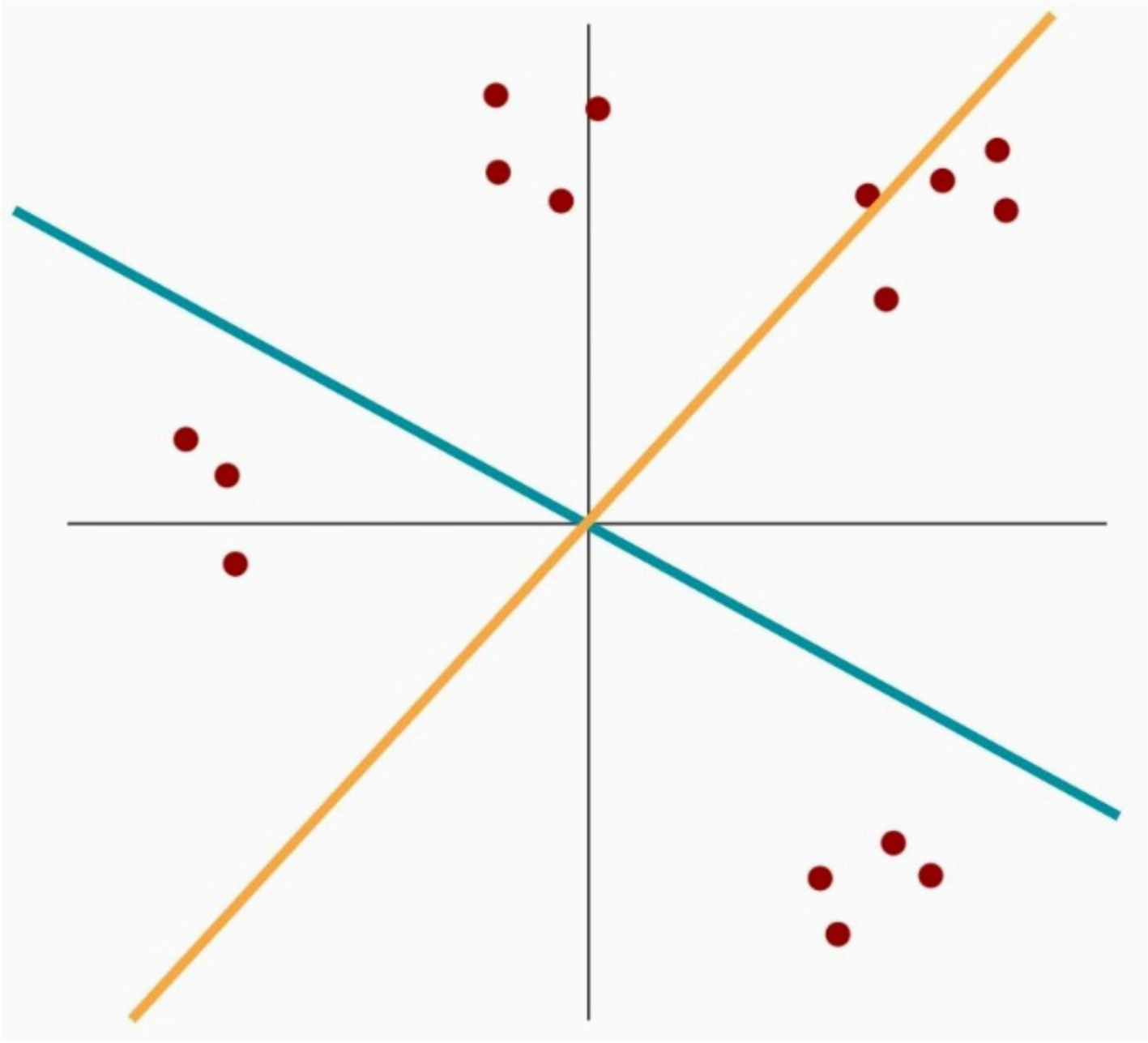
Linear: O(L)

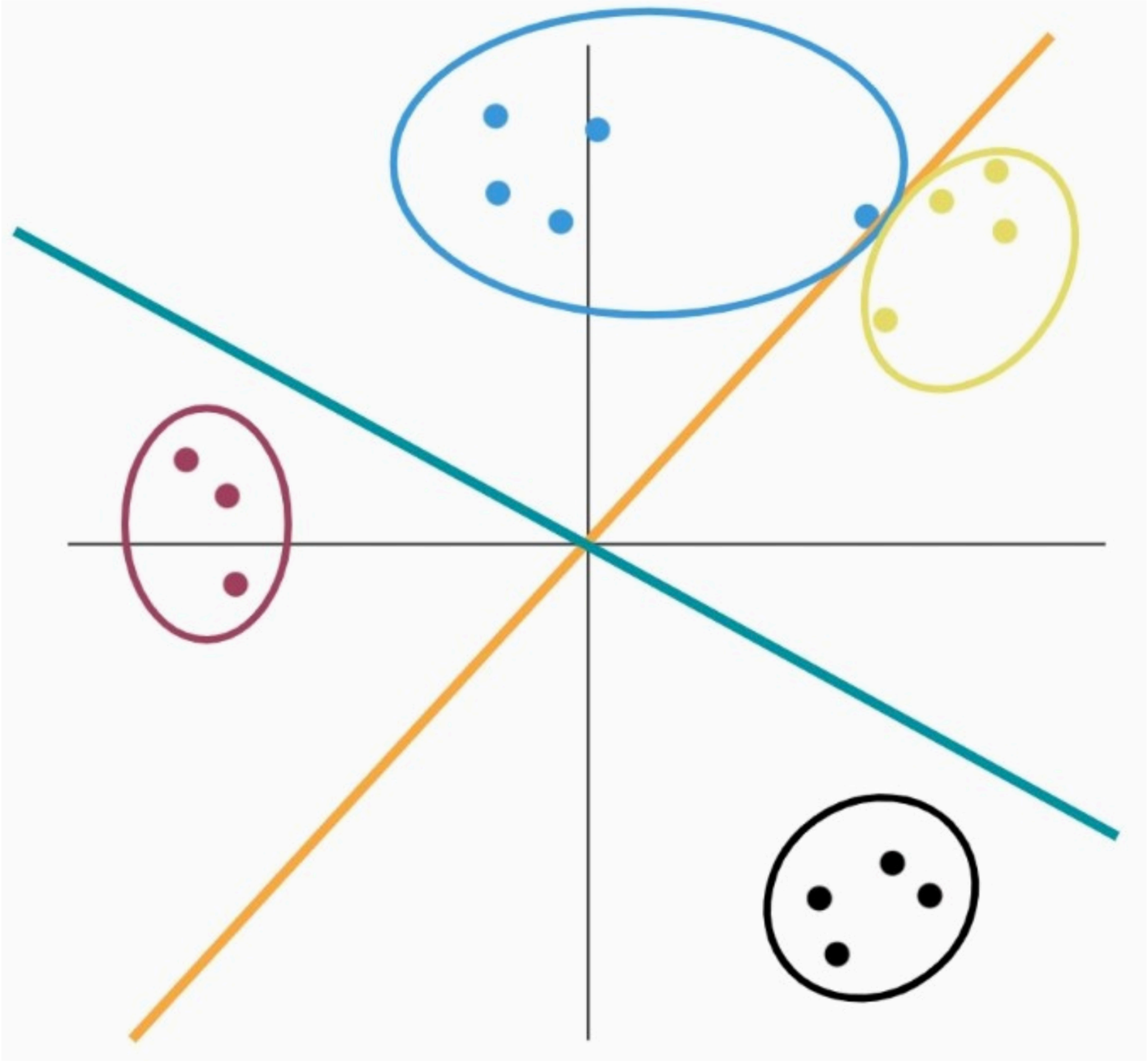

output
softmax
QK^T
L^2 dot product operations
K
V
Q
input
L = 1 million tokens

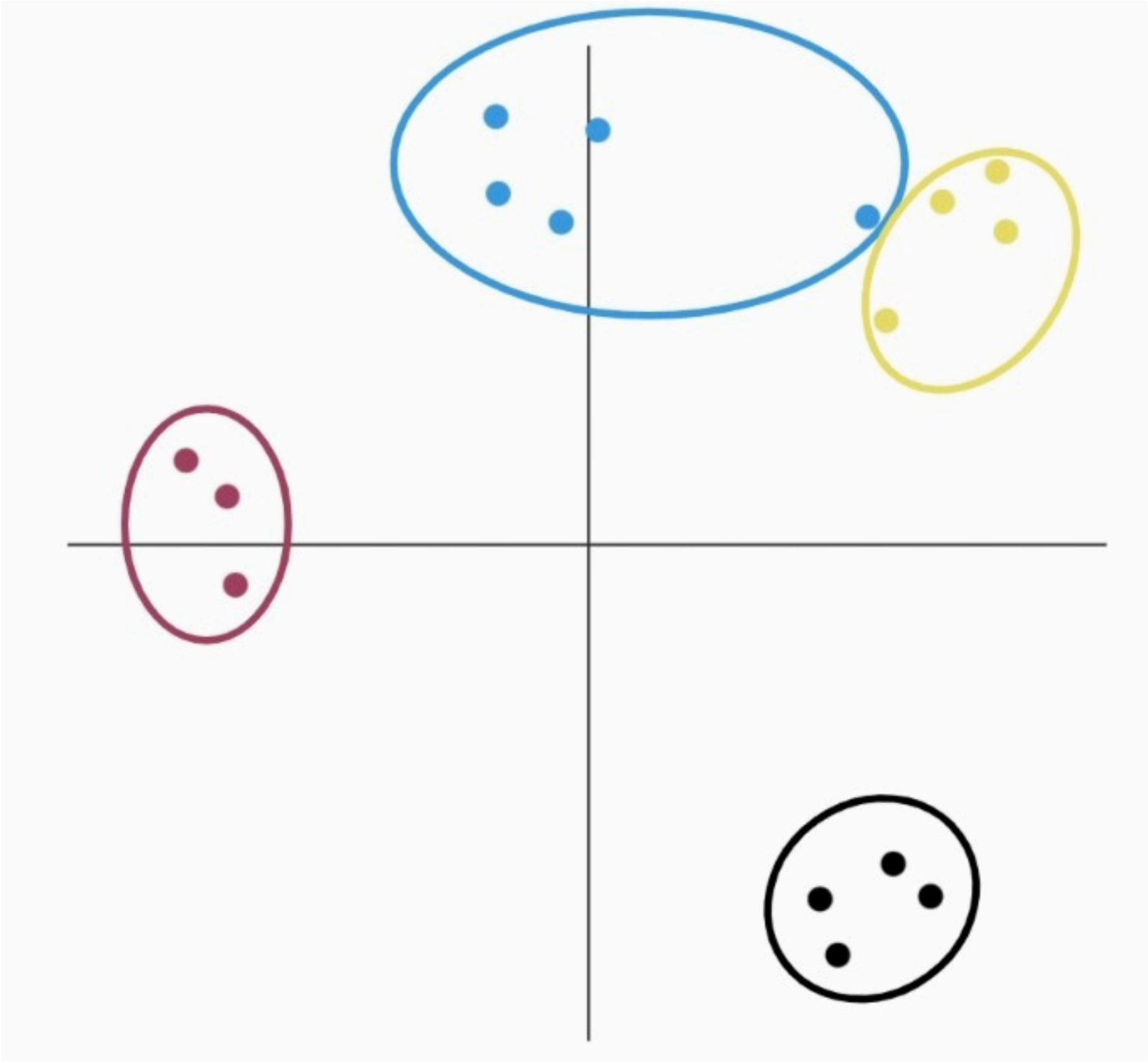

output
softmax
q1 q2 q3 q4 q5 q6
k1
k2
k3
k4
k5
k6
K
V
Q
input
L = 1 million tokens











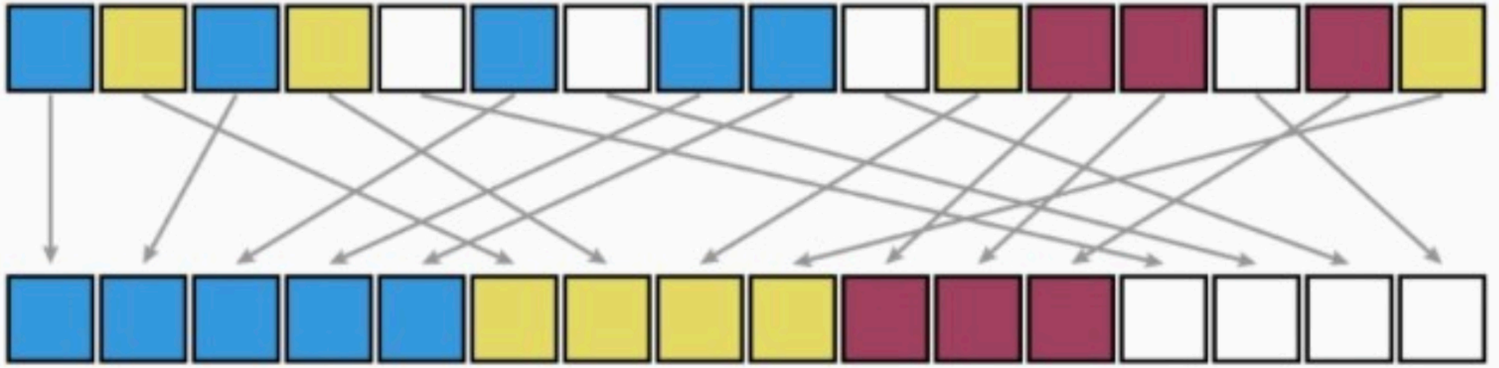
Sequence
of queries=keys

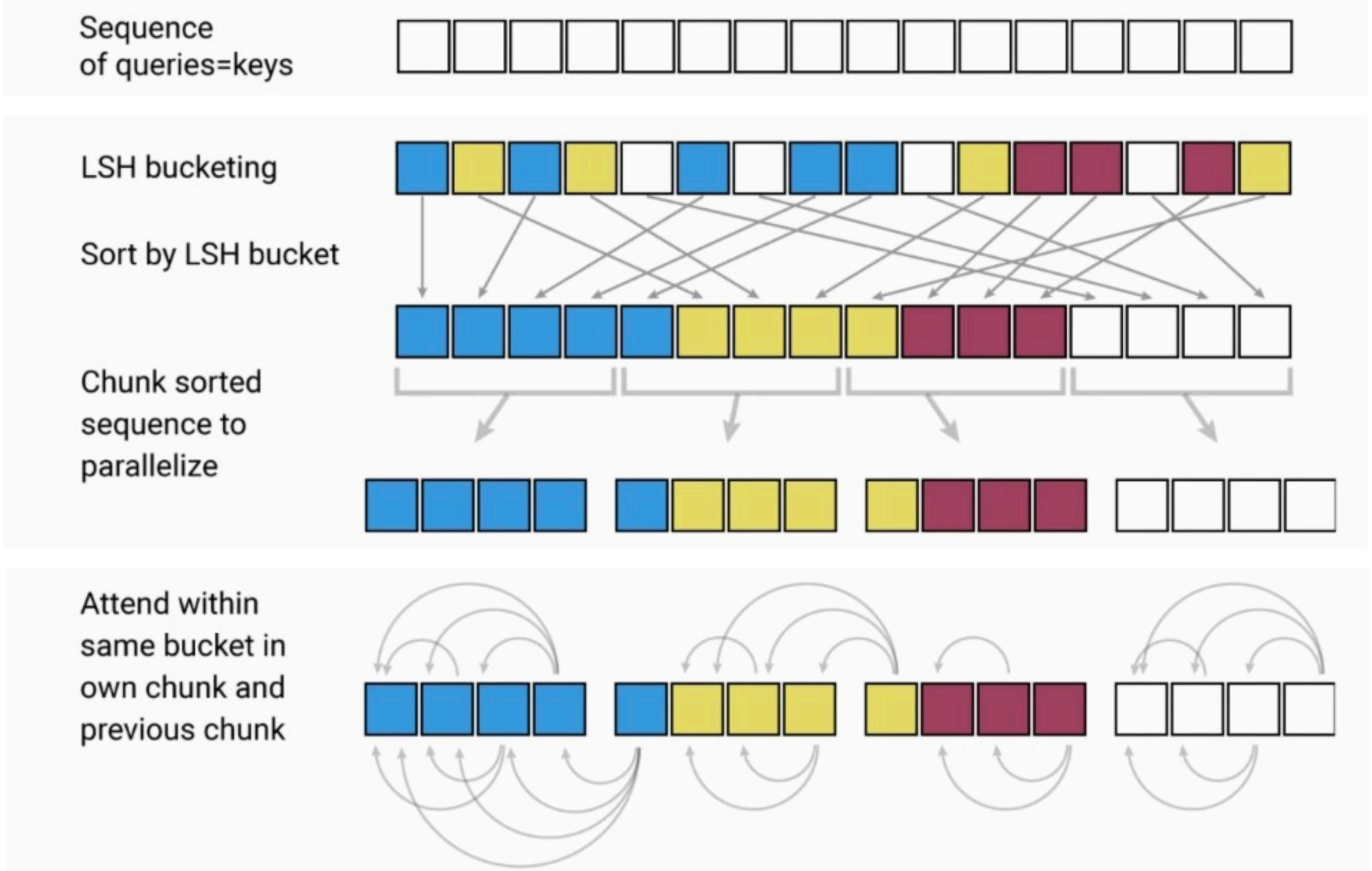
LSH bucketing

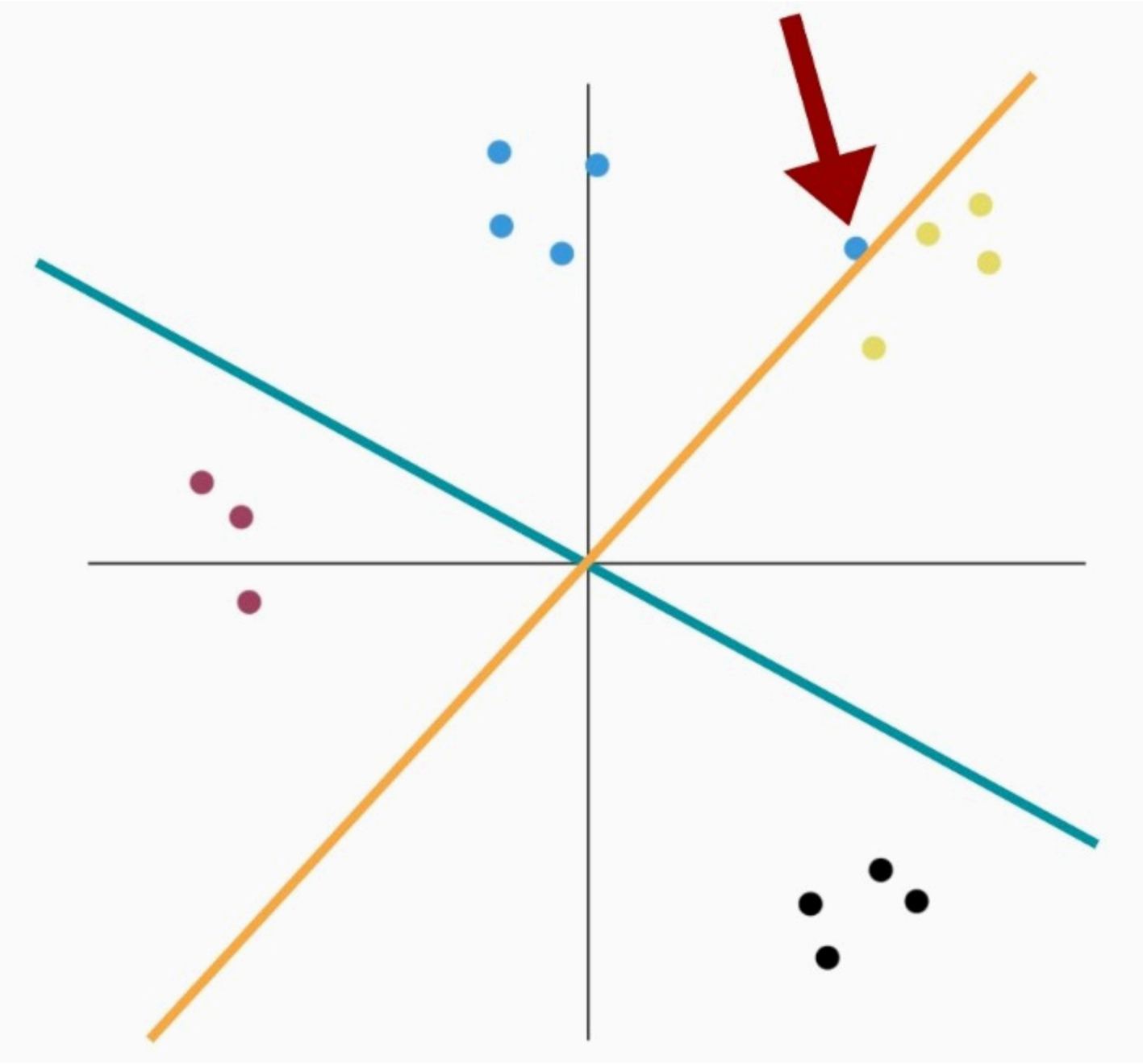
Sequence
of queries=keys

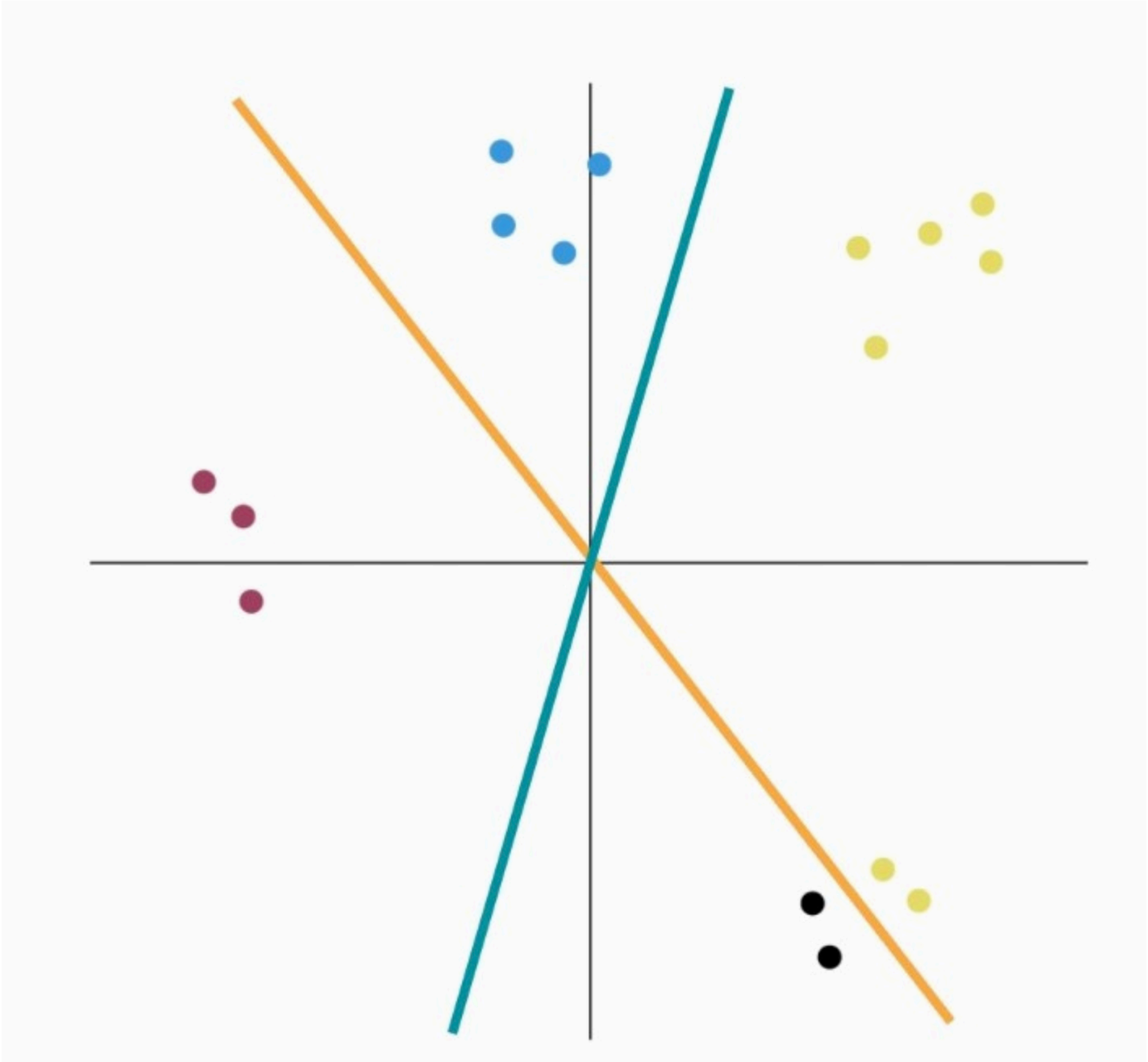
LSH bucketing

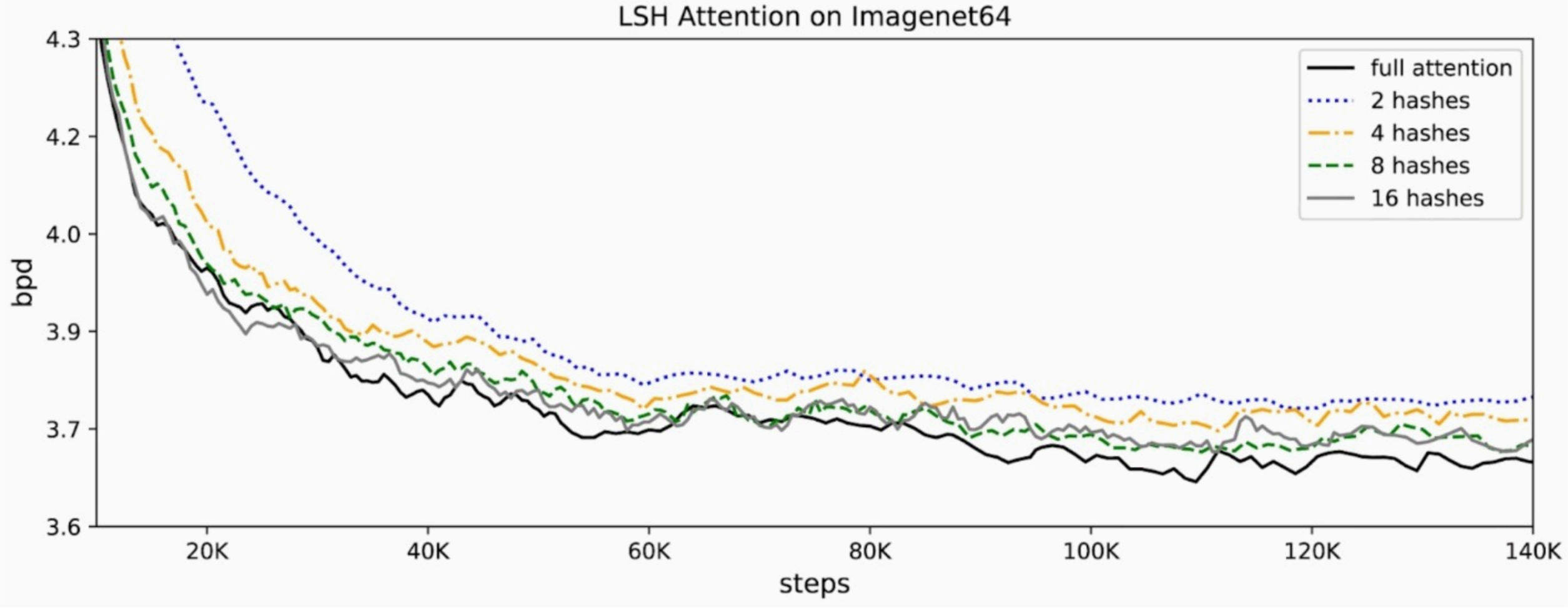
Sort by LSH bucket

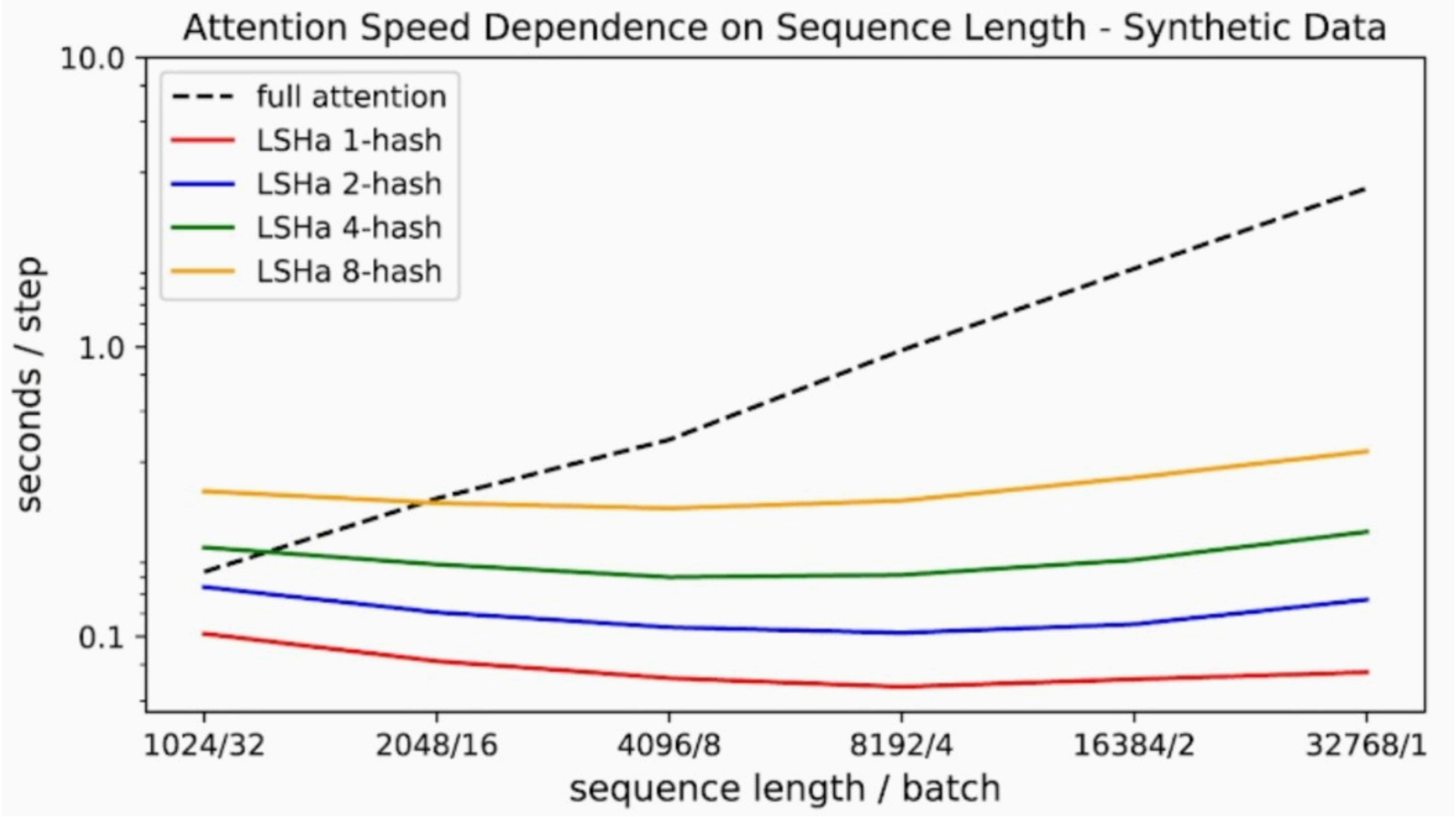



Sequence
of queries=keys
LSH bucketing
Sort by LSH bucket
Chunk sorted
sequence to
parallelize
Attend within
same bucket in
own chunk and
previous chunk






LSH Attention on Imagenet64
4.3
4.2
4.0
3.9
3.7
3.6
bpd
20K
40K
60K
80K
100K
120K
140K
steps
full attention
2 hashes
4 hashes
8 hashes
16 hashes


Attention Speed Dependence on Sequence Length - Synthetic Data
10.0
1.0
0.1
seconds / step
full attention
LSHa 1-hash
LSHa 2-hash
LSHa 4-hash
LSHa 8-hash
1024/32
2048/16
4096/8
8192/4
16384/2
32768/1
sequence length / batch

- Reversible Layers and Chunking make the Transformer memory-efficient without sacrificing accuracy

- LSH Attention approximates *global attention* with O(L log L) time complexity

Efficient Transformers: A Survey

Yi Tay YITAY@GOOGLE.COM
Google Research

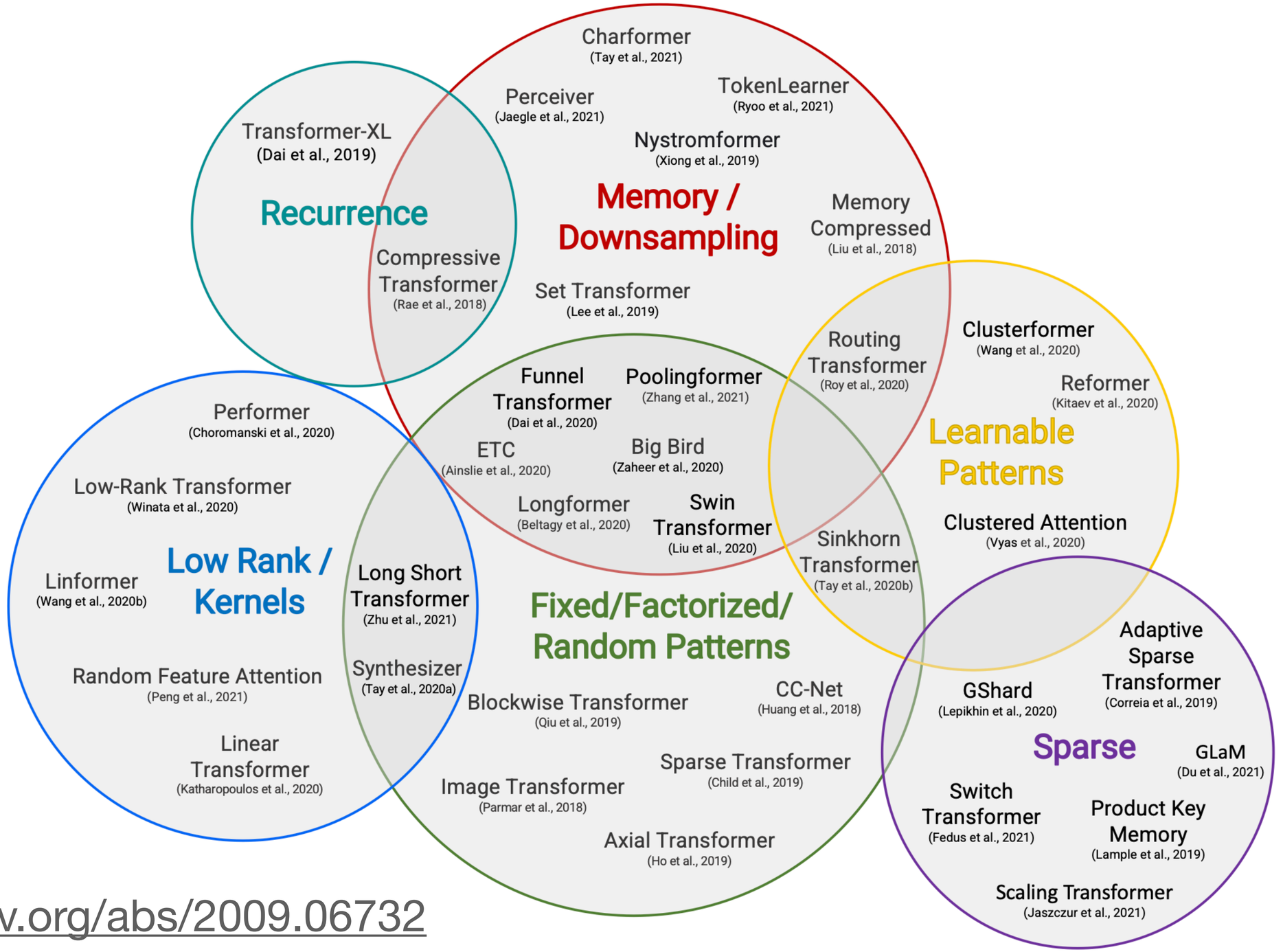
Mostafa Dehghani DEHGHANI@GOOGLE.COM
Google Research, Brain team

Dara Bahri DBAHRI@GOOGLE.COM
Google Research

Donald Metzler METZLER@GOOGLE.COM
Google Research

Editor: Preprint, Version 2, Updated Mar 2022

https://arxiv.org/abs/2009.06732



https://arxiv.org/abs/2009.06732

Model / Paper	Complexity	Decode	Class
Memory Compressed (Liu et al., 2018)	$\mathcal{O}(N_c^2)$	✓	FP+M
Image Transformer (Parmar et al., 2018)	$\mathcal{O}(N.m)$	✓	FP
Set Transformer (Lee et al., 2019)	$\mathcal{O}(kN)$	✗	M
Transformer-XL (Dai et al., 2019)	$\mathcal{O}(N^2)$	✓	RC
Sparse Transformer (Child et al., 2019)	$\mathcal{O}(N\sqrt{N})$	✓	FP
Reformer (Kitaev et al., 2020)	$\mathcal{O}(N \log N)$	✓	LP
Routing Transformer (Roy et al., 2020)	$\mathcal{O}(N\sqrt{N})$	✓	LP
Axial Transformer (Ho et al., 2019)	$\mathcal{O}(N\sqrt{N})$	✓	FP
Compressive Transformer (Rae et al., 2020)	$\mathcal{O}(N^2)$	✓	RC
Sinkhorn Transformer (Tay et al., 2020b)	$\mathcal{O}(B^2)$	✓	LP
Longformer (Beltagy et al., 2020)	$\mathcal{O}(n(k+m))$	✓	FP+M
ETC (Ainslie et al., 2020)	$\mathcal{O}(N_g^2 + NN_g)$	✗	FP+M
Synthesizer (Tay et al., 2020a)	$\mathcal{O}(N^2)$	✓	LR+LP
Performer (Choromanski et al., 2020a)	$\mathcal{O}(N)$	✓	KR
Funnel Transformer (Dai et al., 2020)	$\mathcal{O}(N^2)$	✓	FP+DS

Table 1: Summary of Efficient Transformer Models. Models in the first section are mainly efficient attention methods. Models in the subsequent lower section generally refer to sparse models. Class abbreviations include: FP = Fixed Patterns or Combinations of Fixed Patterns, M = Memory, LP = Learnable Pattern, LR = Low-Rank, KR = Kernel RC = Recurrence, and DS = Downsampling. Furthermore, N generally refers to the sequence length and B is the local window (or block) size. N_g and N_c denote global model memory length and convolutionally-compressed sequence lengths respectively.

Linformer (Wang et al., 2020c)	$\mathcal{O}(N)$	✗	LR
Linear Transformers (Katharopoulos et al., 2020)	$\mathcal{O}(N)$	✓	KR
Big Bird (Zaheer et al., 2020)	$\mathcal{O}(N)$	✗	FP+M
Random Feature Attention (Peng et al., 2021)	$\mathcal{O}(N)$	✓	KR
Long Short Transformers (Zhu et al., 2021)	$\mathcal{O}(kN)$	✓	FP + LR
Poolingformer (Zhang et al., 2021)	$\mathcal{O}(N)$	✗	FP+M
Nyströmformer (Xiong et al., 2021b)	$\mathcal{O}(kN)$	✗	M+DS
Perceiver (Jaegle et al., 2021)	$\mathcal{O}(kN)$	✓	M+DS
Clusterformer (Wang et al., 2020b)	$\mathcal{O}(N \log N)$	✗	LP
Luna (Ma et al., 2021)	$\mathcal{O}(kN)$	✓	M
TokenLearner (Ryoo et al., 2021)	$\mathcal{O}(k^2)$	✗	DS

Table 1: Summary of Efficient Transformer Models. Models in the first section are mainly efficient attention methods. Models in the subsequent lower section generally refer to sparse models. Class abbreviations include: FP = Fixed Patterns or Combinations of Fixed Patterns, M = Memory, LP = Learnable Pattern, LR = Low-Rank, KR = Kernel RC = Recurrence, and DS = Downsampling. Furthermore, N generally refers to the sequence length and B is the local window (or block) size. N_g and N_c denote global model memory length and convolutionally-compressed sequence lengths respectively.

Adaptive Sparse Transformer (Correia et al., 2019)	$\mathcal{O}(N^2)$	✓	Sparse
Product Key Memory (Lample et al., 2019)	$\mathcal{O}(N^2)$	✓	Sparse
Switch Transformer (Fedus et al., 2021)	$\mathcal{O}(N^2)$	✓	Sparse
ST-MoE (Zoph et al., 2022)	$\mathcal{O}(N^2)$	✓	Sparse
GShard (Lepikhin et al., 2020)	$\mathcal{O}(N^2)$	✓	Sparse
Scaling Transformers (Jaszczur et al., 2021)	$\mathcal{O}(N^2)$	✓	Sparse
GLaM (Du et al., 2021)	$\mathcal{O}(N^2)$	✓	Sparse

- **Fixed Patterns (FP)** - The earliest modifications to self-attention simply sparsifies the attention matrix by limiting the field of view to fixed, predefined patterns such as local windows and block patterns of fixed strides.

  - **Blockwise Patterns** The simplest example of this technique in practice is the blockwise (or chunking) paradigm which considers blocks of local receptive fields by chunking input sequences into fixed blocks. Examples of models that do this include Blockwise (Qiu et al., 2019) and/or Local Attention (Parmar et al., 2018). Chunking input sequences into blocks reduces the complexity from N^2 to B^2 (block size) with $B << N$, significantly reducing the cost. These blockwise or chunking methods serve as a basis for many more complex models.
  - **Strided Patterns** Another approach is to consider strided attention patterns, i.e., only attending at fixed intervals. Models such as Sparse Transformer (Child et al., 2019) and/or Longformer (Beltagy et al., 2020) employ strided or "dilated" windows.
  - **Compressed Patterns** - Another line of attack here is to use some pooling operator to down-sample the sequence length to be a form of fixed pattern. For instance, Compressed Attention (Liu et al., 2018) uses strided convolution to effectively reduce the sequence length.

- **Recurrence** - A natural extension to the blockwise method is to connect these blocks via recurrence. Transformer-XL (Dai et al., 2019) proposed a segment-level recurrence mechanism that connects multiple segments and blocks. These models can, in some sense, be viewed as *fixed pattern* models. However, we decided to create its own category due to its deviation from other block / local approaches.

- **Combination of Patterns (CP)** - The key idea of combined[2] approaches is to improve coverage by combining two or more distinct access patterns. For example, the Sparse Transformer (Child et al., 2019) combines strided and local attention by assigning half of its heads to each pattern. Similarly, Axial Transformer (Ho et al., 2019) applies a sequence of self-attention computations given a high dimensional tensor as input, each along a single axis of the input tensor. In essence, the combination of patterns reduces memory complexity in the same way that fixed patterns does. The difference, however, is that the aggregation and combinaton of multiple patterns improves the overall coverage of the self-attention mechanism.

- **Learnable Patterns (LP)** - An extension to fixed, pre-determined pattern are *learnable* ones. Unsurprisingly, models using learnable patterns aim to learn the access pattern in a data-driven fashion. A key characteristic of learning patterns is to determine a notion of token relevance and then assign tokens to buckets or clusters (Vyas et al., 2020; Wang et al., 2020b). Notably, Reformer (Kitaev et al., 2020) introduces a hash-based similarity measure to efficiently cluster tokens into chunks. In a simlar vein, the Routing Transformer (Roy et al., 2020) employs online k-means clustering on the tokens. Meanwhile, the Sinkhorn Sorting Network (Tay et al., 2020b) exposes the sparsity in attention weights by learning to to sort blocks of the input sequence. In all these models, the similarity function is trained end-to-end jointly with the rest of the network. The key idea of learnable patterns is still to exploit fixed patterns (chunked patterns). However, this class of methods learns to sort/cluster the input tokens - enabling a more optimal global view of the sequence while maintaining the efficiency benefits of fixed patterns approaches.

- **Low-Rank Methods** - Another emerging technique is to improve efficiency by leveraging low-rank approximations of the self-attention matrix. The key idea is to assume low-rank structure in the $N \times N$ matrix. The Linformer (Wang et al., 2020c) is a classic example of this technique, as it projects the length dimension of keys and values to a lower-dimensional representation ($N \rightarrow k$). It is easy to see that the low-rank method ameliorates the memory complexity problem of self-attention because the $N \times N$ matrix is now decomposed to $N \times k$.

- **Kernels** - Another recently popular method to improve the efficiency of Transformers is to view the attention mechanism through kernelization. The usage of kernels (Katharopoulos et al., 2020; Choromanski et al., 2020a) enable clever mathematical re-writing of the self-attention mechanism to avoid explicitly computing the $N \times N$ matrix. Since kernels are a form of approximation of the attention matrix, they can be also viewed as a type of low-rank approach (Choromanski et al., 2020a). Examples of recent work in this area include Performers, Linear Transformers and Random Feature Attention (RFA, (Peng et al., 2021))

Long-range Arena (LRA) dataset

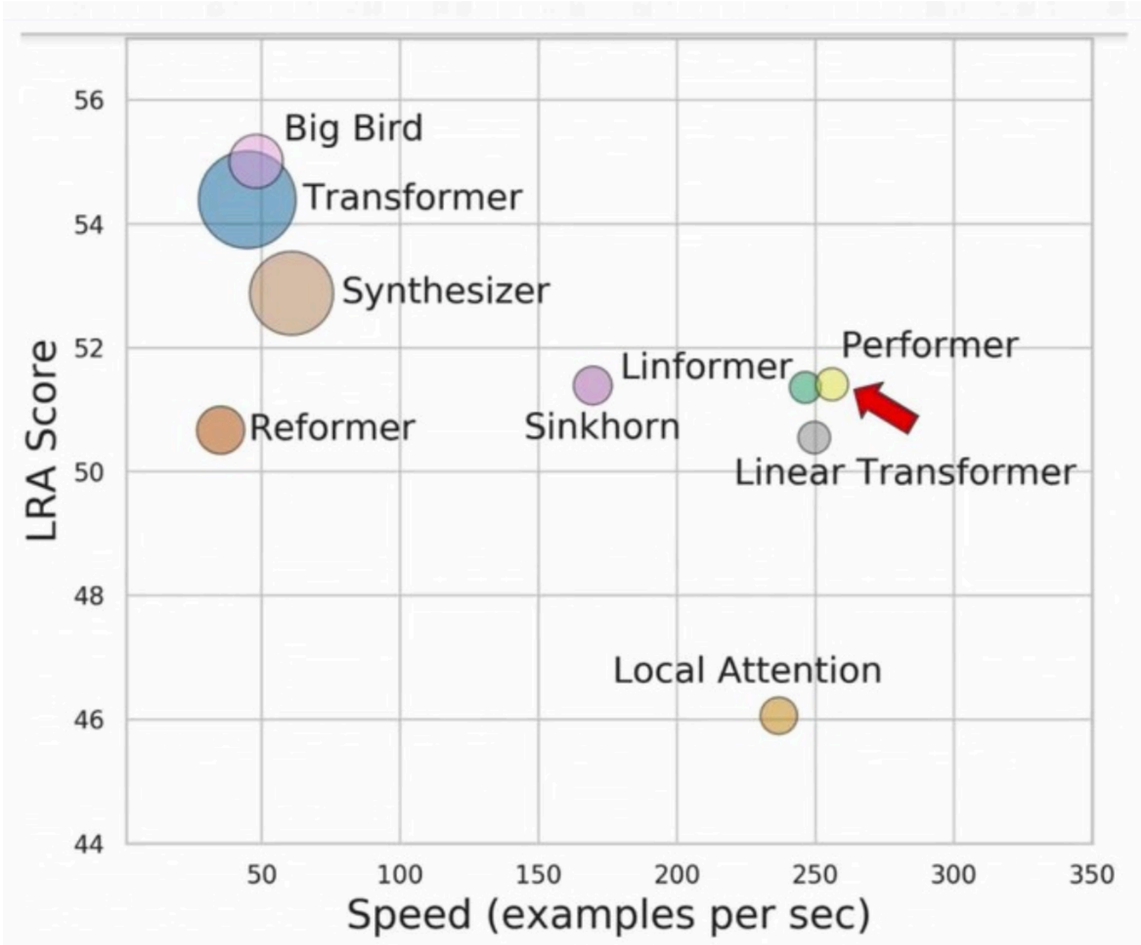
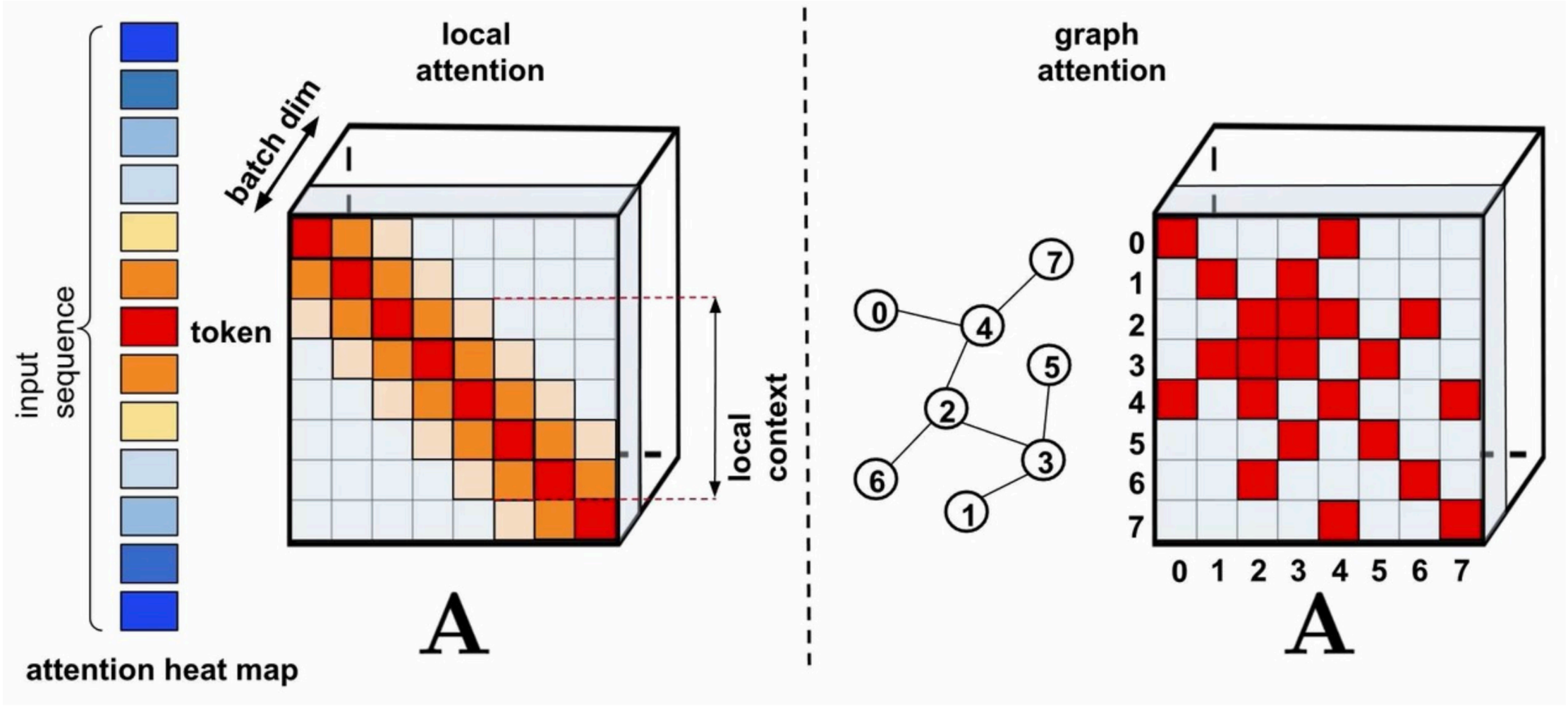
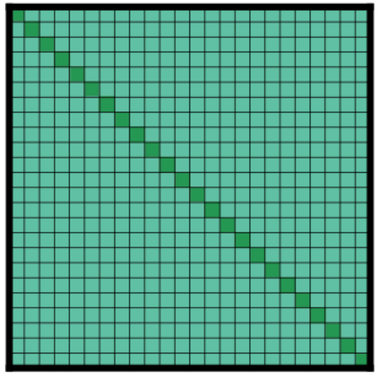


Figure 3: Performance (y axis), speed (x axis), and memory footprint (size of the circles) of different models.

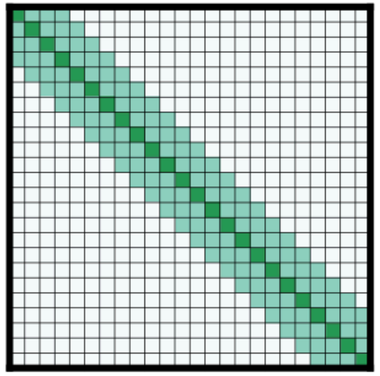
Sparsifying Attention



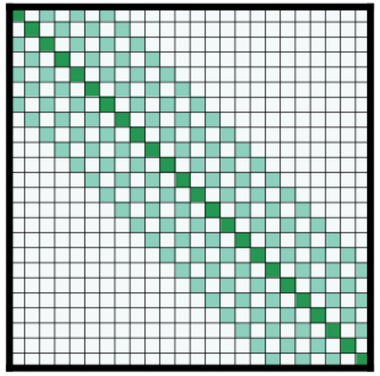
https://iclr.cc/virtual/2021/session/4343



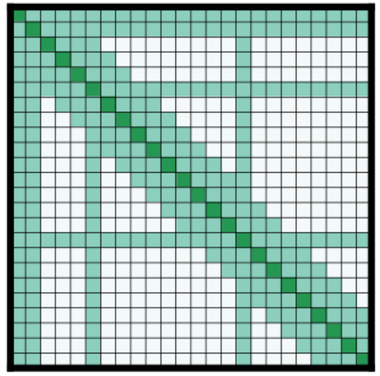
(a) Full n^2 attention



(b) Sliding window attention

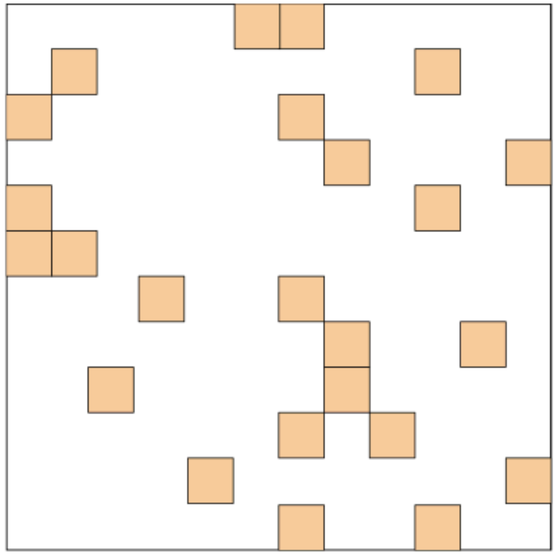


(c) Dilated sliding window

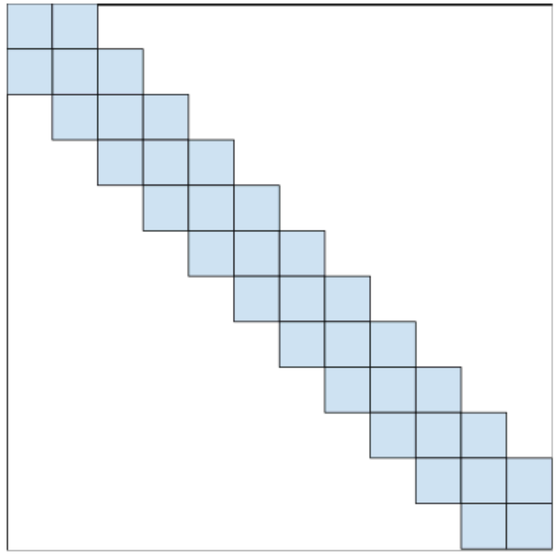


(d) Global+sliding window

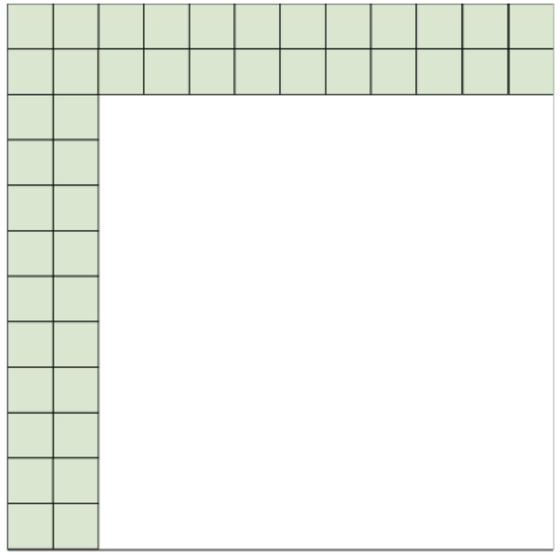
https://arxiv.org/abs/2004.05150



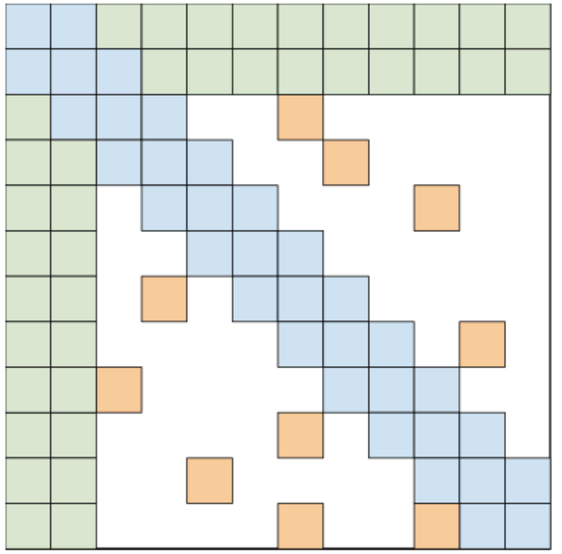
(a) Random attention



(b) Window attention



(c) Global Attention



(d) BIGBIRD

https://arxiv.org/abs/2007.14062

TRAIN SHORT, TEST LONG: ATTENTION WITH LINEAR BIASES ENABLES INPUT LENGTH EXTRAPOLATION

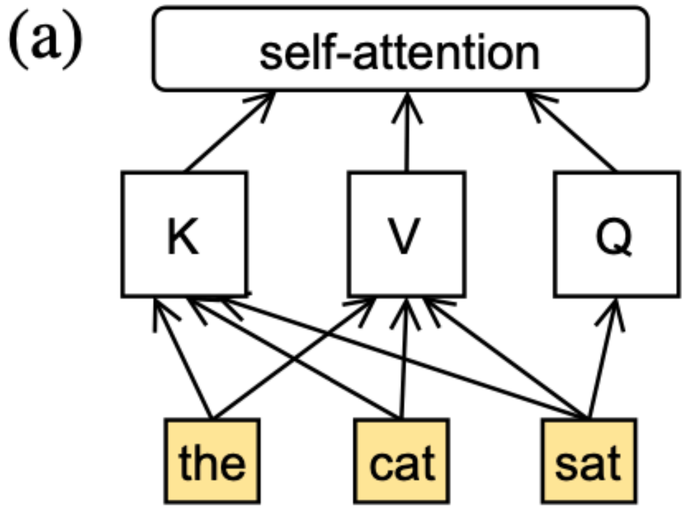
Ofir Press[1,2] **Noah A. Smith**[1,3] **Mike Lewis**[2]

[1]Paul G. Allen School of Computer Science & Engineering, University of Washington
[2]Facebook AI Research
[3]Allen Institute for AI
ofirp@cs.washington.edu

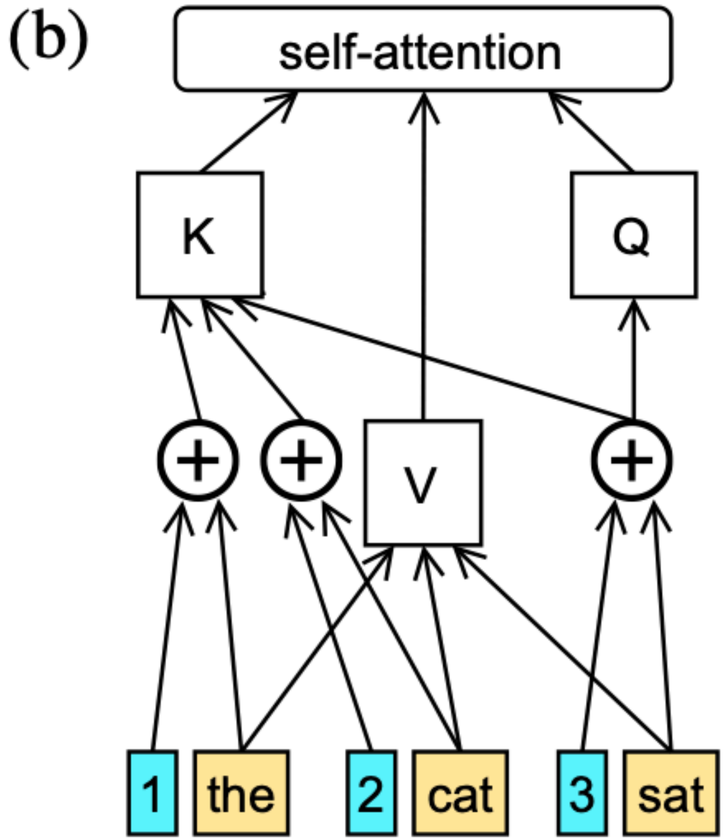
https://arxiv.org/abs/2108.12409

Rotary Position Embeddings The rotary method was introduced by Su et al. (2021) and has recently been popularized by the open source GPT-3 (Brown et al., 2020) implementation GPT-J (Wang & Komatsuzaki, 2021). Instead of adding sinusoidal embeddings at the bottom of the transformer, they multiply the keys and queries of every attention layer by sinusoidal embeddings.

Unlike the sinusoidal or learned positional embedding approach, the rotary method injects position information into the model at every layer, not just at the initial one. In addition, it adds no position information to the values of the self-attention sublayer. The output of a self-attention sublayer is a linearly transformed, weighted sum of the input value vectors; therefore, by not inserting position information into the values, the outputs of each transformer-layer contain no explicit position information. We suspect that this segregation of position information may be beneficial for extrapolation, and we draw inspiration from it in the design of our method (§3).



Standard Attention



Position Infused Attention

T5 Bias Though most models use trained or sinusoidal position embeddings, the T5 model of Raffel et al. (2020) uses a relative position method (Shaw et al., 2018; Huang et al., 2019) that adds no position information to word embeddings (as in the previous method). Instead, it modifies the way attention values are computed. We refer to this as the "T5 bias" method.[6] To compute attention values in the unmodified transformer, we compute the dot product of every query with every relevant key and then softmax these attention values. In this method, we compute the attention values as before, but then we add a learned, shared bias to each query-key score that is dependent on just the distance between the query and key. Therefore, all query-key scores where the query and key distance are zero (i.e., the query and key represent the same token) get a specific learned bias, all scores where the query and key are one word away get a different learned bias, and so on, up to a certain point, from where multiple different distances share the same learned bias (which might be beneficial for extrapolation). As in the rotary method, the T5 bias injects position information into the model at every layer and integrates no explicit position information into the self-attention value vectors.

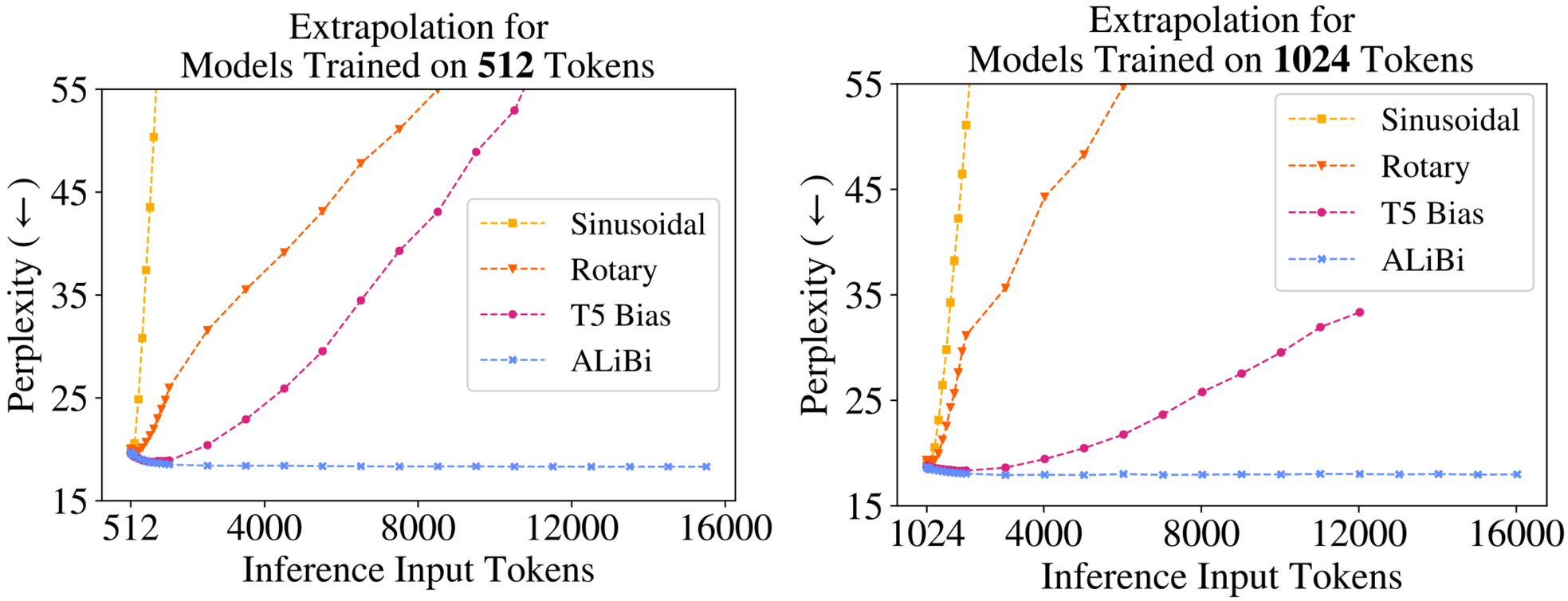


Figure 1: Extrapolation: as the (validation-set's) input sequence gets longer (x-axis), current position methods (sinusoidal, rotary, and T5) show degraded perplexity (y-axis, lower is better), but our method (§3) does not. Models were trained on WikiText-103 with sequences of $L = 512$ (left) or $L = 1{,}024$ (right) tokens. T5 ran out of memory on our 32GB GPU. For more detail on exact perplexities and runtimes, see Tables 2 and 3 in the appendix.

We therefore introduce Attention with Linear Biases (ALiBi) to facilitate efficient extrapolation. ALiBi negatively biases attention scores with a linearly decreasing penalty proportional to the distance between the relevant key and query. Our simple approach eliminates position embeddings.

Compared to a sinusoidal model trained on the same input length, our method requires no additional runtime or parameters and incurs a negligible (0–0.7%) memory increase. ALiBi can be implemented by changing only a few lines of existing transformer code.

Using ALiBi, a transformer LM can be trained on short-L sequences and therefore at much lower cost, and it can still be reliably applied to long sequences at runtime. For example, a 1.3 billion parameter LM trained on $L = 1024$ tokens with ALiBi achieves the same perplexity as a sinusoidal model trained on $L = 2048$ when both are tested on sequences of 2048 tokens, even though *our model is 11% faster and uses 11% less memory.*

Attention with Linear Biases (ALiBi)

- Attention for i^{th} query $\mathbf{q}_i \in \mathbb{R}^{1\times d}$ for i from 1 to L in each head and d is the head dimension
- Given the first i keys $\mathbf{K} \in \mathbb{R}^{i\times d}$ (for an auto-regressive LM)
- Attention weights are softmax($\mathbf{q}_i\mathbf{K}^\top$)
- ALiBi introduces two differences:

  1. Do not add position embeddings at any point in the network

  2. softmax($\mathbf{q}_i\mathbf{K}^\top + m \cdot [-(i-1), \ldots, -2, -1, 0]$)

     Set scalar m for each head as a geometric sequence, e.g. $\frac{1}{2^1}, \frac{1}{2^2}, \ldots, \frac{1}{2^8}$

Attention with Linear Biases (ALiBi)

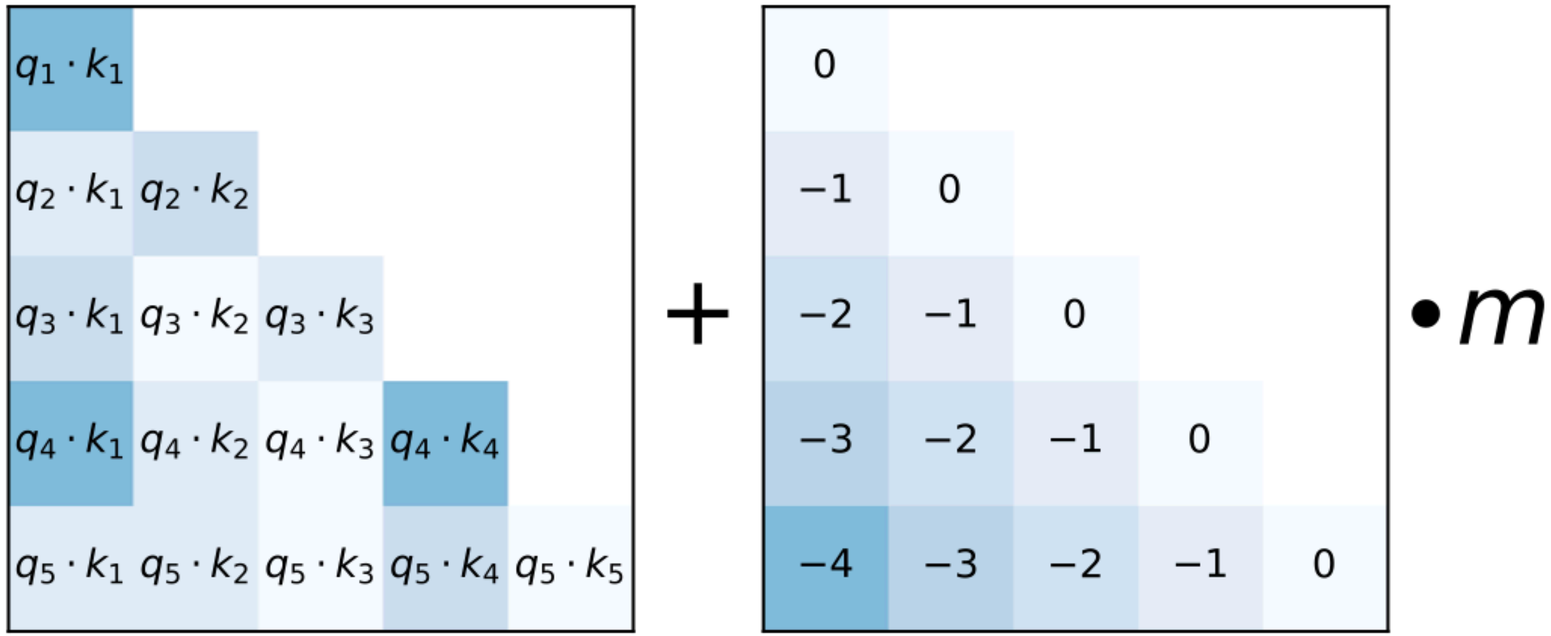
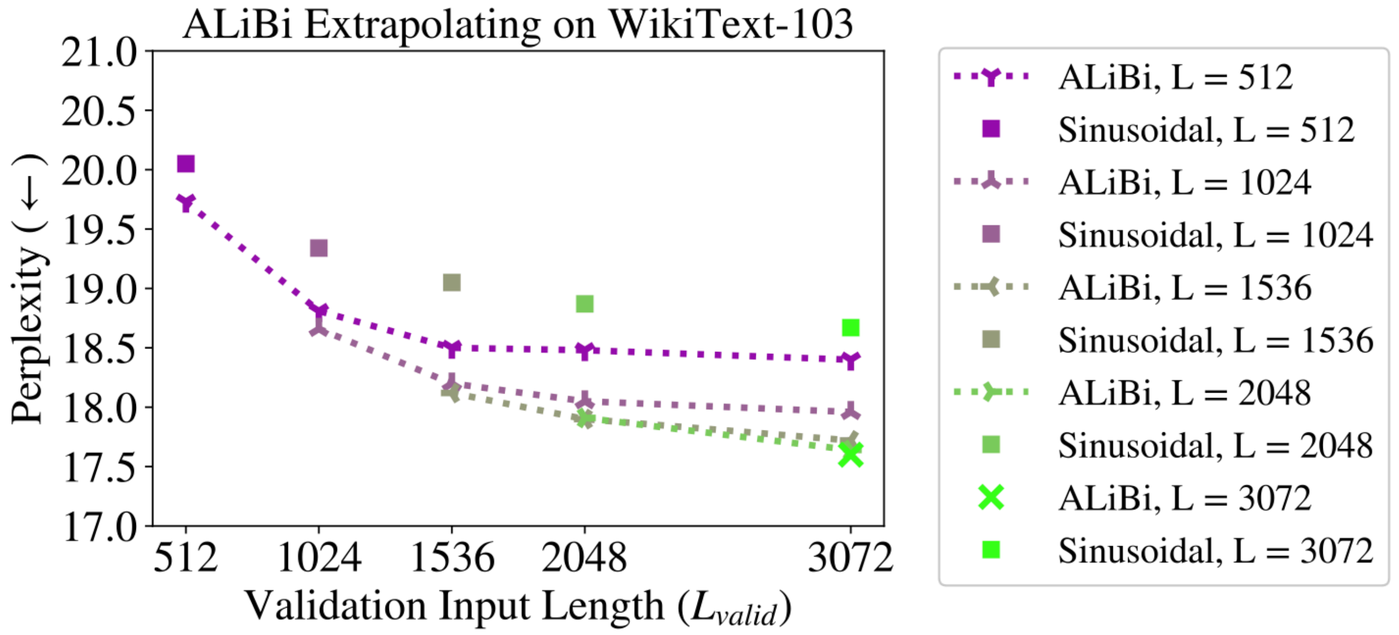
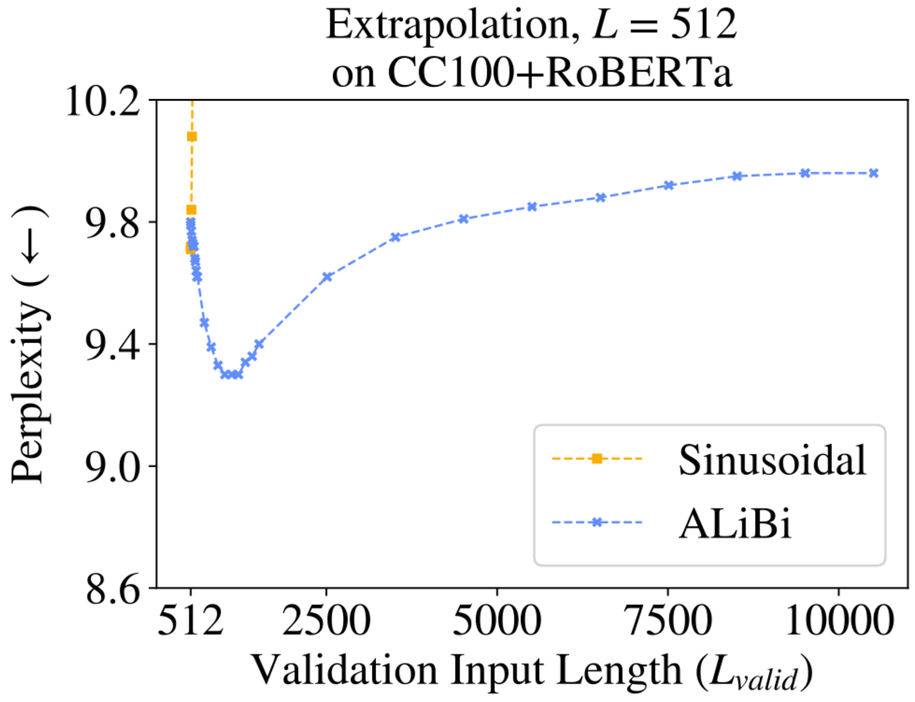
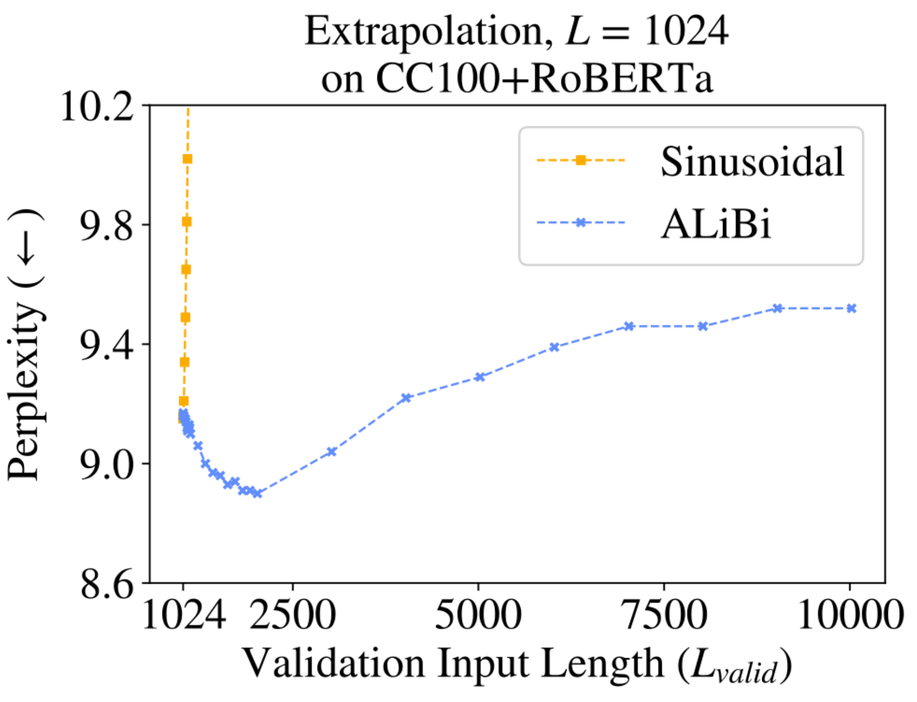


Figure 3: When computing attention scores for each head, our linearly biased attention method, ALiBi, adds a constant bias (right) to each attention score ($\mathbf{q}_i \cdot \mathbf{k}_j$, left). As in the unmodified attention sublayer, the softmax function is then applied to these scores, and the rest of the computation is unmodified. **m is a head-specific scalar** that is set and not learned throughout training. We show that our method for setting m values generalizes to multiple text domains, models and training compute budgets. When using ALiBi, we do *not* add positional embeddings at the bottom of the network.


ALiBi Extrapolating on WikiText-103
Perplexity (←)
21.0
20.5
20.0
19.5
19.0
18.5
18.0
17.5
17.0
512
1024
1536
2048
3072
Validation Input Length (L_{valid})
ALiBi, L = 512
Sinusoidal, L = 512
ALiBi, L = 1024
Sinusoidal, L = 1024
ALiBi, L = 1536
Sinusoidal, L = 1536
ALiBi, L = 2048
Sinusoidal, L = 2048
ALiBi, L = 3072
Sinusoidal, L = 3072


Extrapolation, $L = 512$
on CC100+RoBERTa
Perplexity (←)
10.2
9.8
9.4
9.0
8.6
512
2500
5000
7500
10000
Validation Input Length (L_{valid})
Sinusoidal
ALiBi


Extrapolation, $L = 1024$
on CC100+RoBERTa
Perplexity (←)
10.2
9.8
9.4
9.0
8.6
1024
2500
5000
7500
10000
Validation Input Length (L_{valid})
Sinusoidal
ALiBi