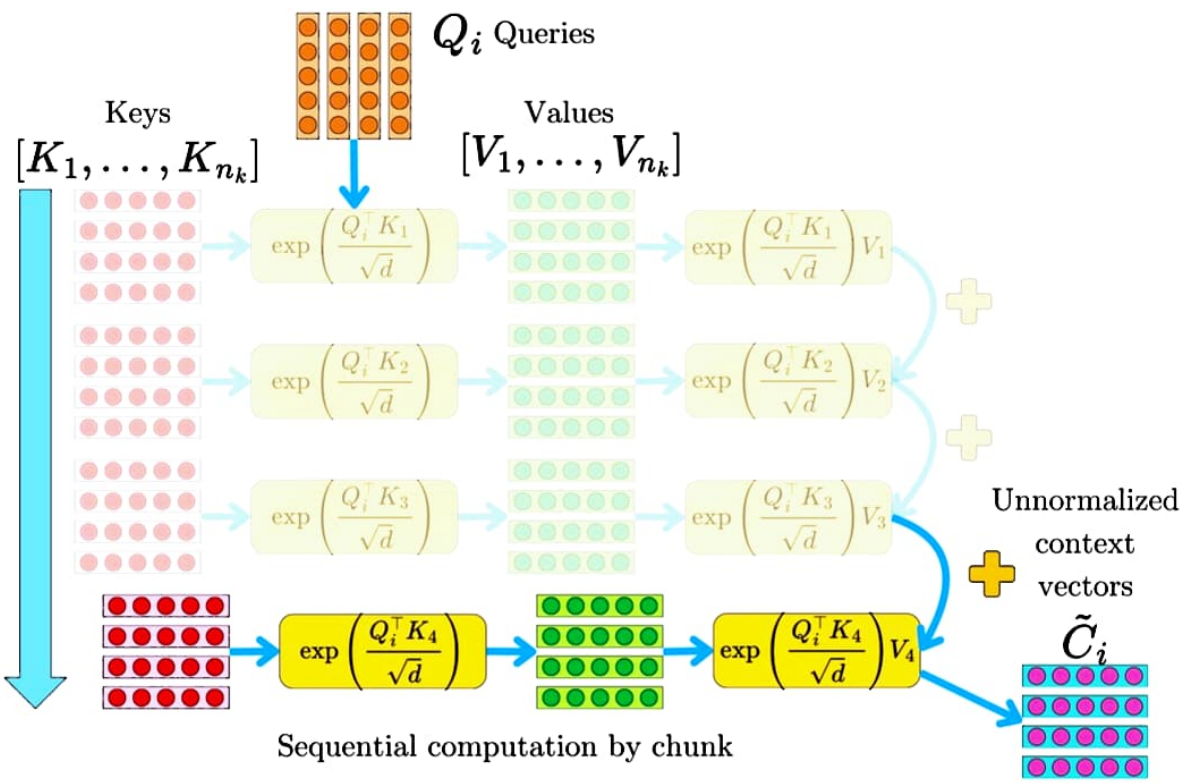
Self-attention Does Not Need $\mathcal{O}(N^2)$ Memory!

Damien Benveniste



Let's consider the computation of the context vectors:

$$\mathbf{c}_i = \frac{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right) \mathbf{v}_j}{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)}. \tag{1}$$

Here is how we arrive to the typical $\sim \mathcal{O}(N^2)$ space complexity:

1. The typical assumption is that we first compute the dot product between the query $\mathbf{q}_i$ and all the keys $[\mathbf{k}_1, \ldots, \mathbf{k}_N]$:

$$\mathbf{e}_i = \left[\frac{\mathbf{q}_i^\top \mathbf{k}_1}{\sqrt{d_{\text{model}}}}, \ldots, \frac{\mathbf{q}_i^\top \mathbf{k}_N}{\sqrt{d_{\text{model}}}}\right] \tag{2}$$

   Where $\mathbf{e}_i$ is the alignment score vector of size N for the query $\mathbf{q}_i$, which leads to the $N \times N$ matrix for the N queries.

2. We then perform the softmax transformation:

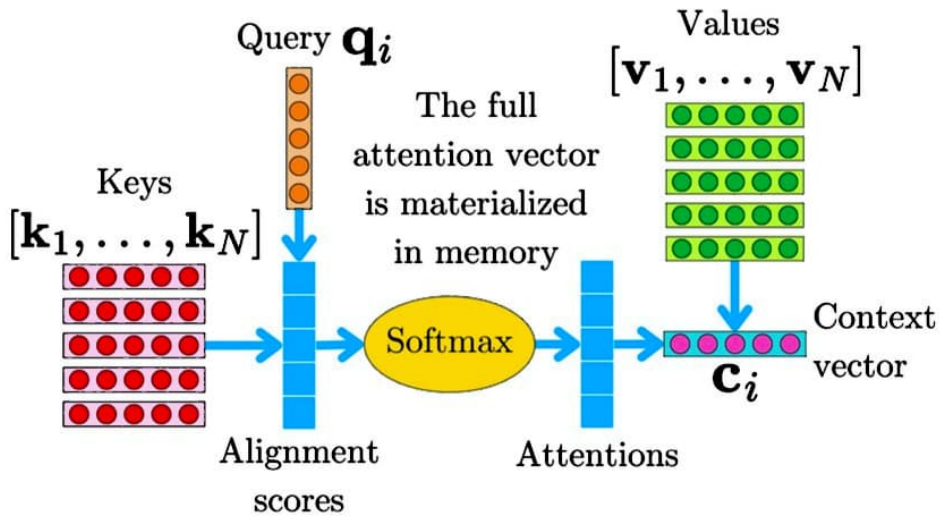
$$\mathbf{a}_i = \frac{1}{\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)} \left[\exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_1}{\sqrt{d_{\text{model}}}}\right), \ldots, \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_N}{\sqrt{d_{\text{model}}}}\right)\right] \tag{3}$$

Here, $\mathbf{a}_i$ is the attention vector of size N for the query $\mathbf{q}_i$. Again, for N queries, it leads to the typical $N \times N$ attention matrix.

3. And finally, we project $\mathbf{a}_i$ onto the different values $V = [\mathbf{v}_1, \ldots, \mathbf{v}_N]$, which leads to $\mathbf{c}_i$:

$$\begin{aligned} \mathbf{c}_i &= \mathbf{a}_i^\top V \\ &= \sum_{j=1}^{N} a_{ij} \mathbf{v}_j \end{aligned} \tag{4}$$

Therefore, naively computing the alignment scores and the attention matrices first forces the materialization of those matrices in memory, which leads to the $\mathcal{O}(N^2)$ space complexity.



However, we do not need to order the computations in this manner! In 2021, Rabe and Staats[2] realized that by reordering the operations, we can greatly reduce the requirements on the memory. The idea is to consider the unnormalized context vector $\tilde{\mathbf{c}}_i$ and the normalization constant $\sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)$ of the softmax transformation separately:

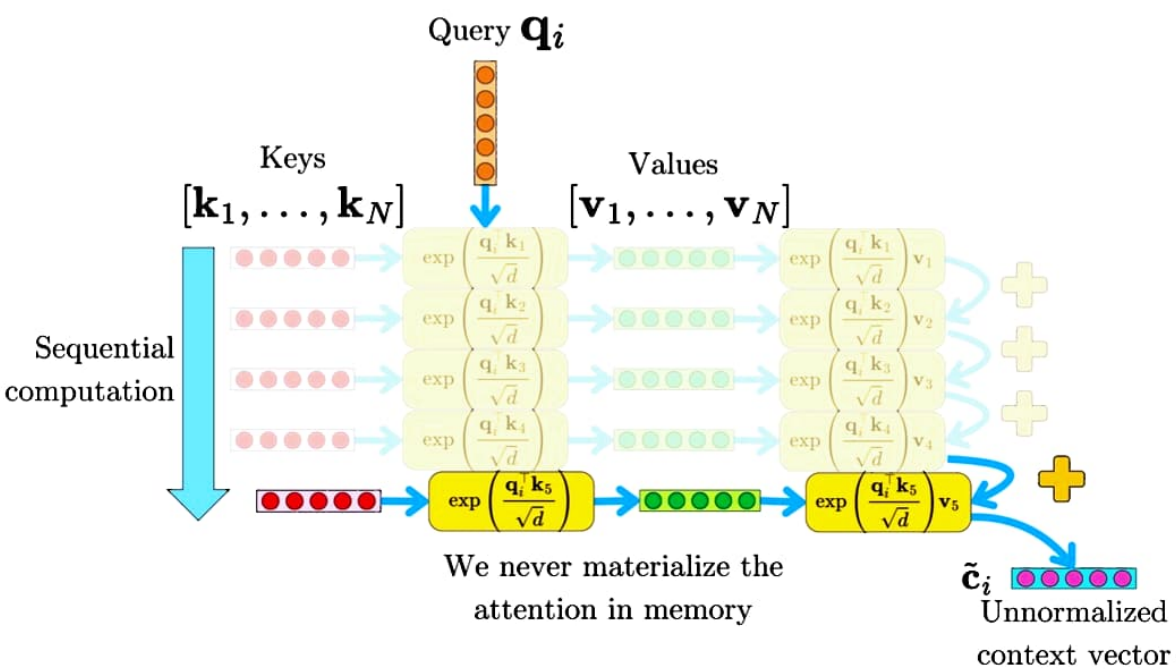
$$\begin{aligned} \tilde{\mathbf{c}}_i &= \sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right) \mathbf{v}_j \\ s_i &= \sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right) \\ \mathbf{c}_i &= \frac{\tilde{\mathbf{c}}_i}{s_i} \end{aligned} \tag{5}$$

Because $\tilde{\mathbf{c}}_i$ and s_i are just sums, we can easily loop through the key-value pairs to compute the context vector:

$\tilde{\mathbf{c}}_i = 0, s_i = 0$ ▷ We initialize $\tilde{\mathbf{c}}_i$ and s_i
for $j = 1$ to N **do** ▷ We loop through the key-value pairs $(\mathbf{k}_j, \mathbf{v}_j)$
  $\tilde{\mathbf{c}}_i \leftarrow \tilde{\mathbf{c}}_i + \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)\mathbf{v}_j$
  $s_i \leftarrow s_i + \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)$
end for
$\mathbf{c}_i = \frac{\tilde{\mathbf{c}}_i}{s_i}$ ▷ We compute the final context vector
Return: $\mathbf{c}_i$

At any point during the for-loop, we only need to store the intermediary values of $\tilde{\mathbf{c}}_i$ and s_i. $\tilde{\mathbf{c}}_i$ is a vector of size d_{model} and s_i is a scalar. Therefore, for one query, we need constant space complexity $\mathcal{O}(1)$ to compute one context vector. Even iterating through all the queries, we never need to capture more than the intermediary values of $\tilde{\mathbf{c}}_i$ and s_i, so we can compute the full attention mechanism in $\mathcal{O}(1)$ space complexity:

for $i = 1$ to N **do** ▷ We loop through the queries
  $\tilde{\mathbf{c}}_i = 0$
  $s_i = 0$
  for $j = 1$ to N **do** ▷ We loop through the key-value pairs $(\mathbf{k}_j, \mathbf{v}_j)$
    $\tilde{\mathbf{c}}_i \leftarrow \tilde{\mathbf{c}}_i + \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)\mathbf{v}_j$
    $s_i \leftarrow s_i + \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{model}}}}\right)$
  end for
  $\mathbf{c}_i = \frac{\tilde{\mathbf{c}}_i}{s_i}$
end for
Return: $[\mathbf{c}_1, \ldots, \mathbf{c}_N]$



In reality, this is not a practical solution because sequential operations are not adapted to the parallelization capability of the CPU, GPU, or TPU hardware that is commonly used for neural network computations. In practice, the queries, keys, and values are partitioned into chunks to allow for a high degree of parallelization while keeping the memory requirement low. Let's assume that we partition the queries into n_q chunks and the keys and values into n_k chunks:

$$
\begin{aligned}
Q &= [Q_1, Q_2, \ldots, Q_{n_q}] \\
K &= [K_1, K_2, \ldots, K_{n_k}] \\
V &= [V_1, V_2, \ldots, V_{n_k}]
\end{aligned} \tag{6}
$$

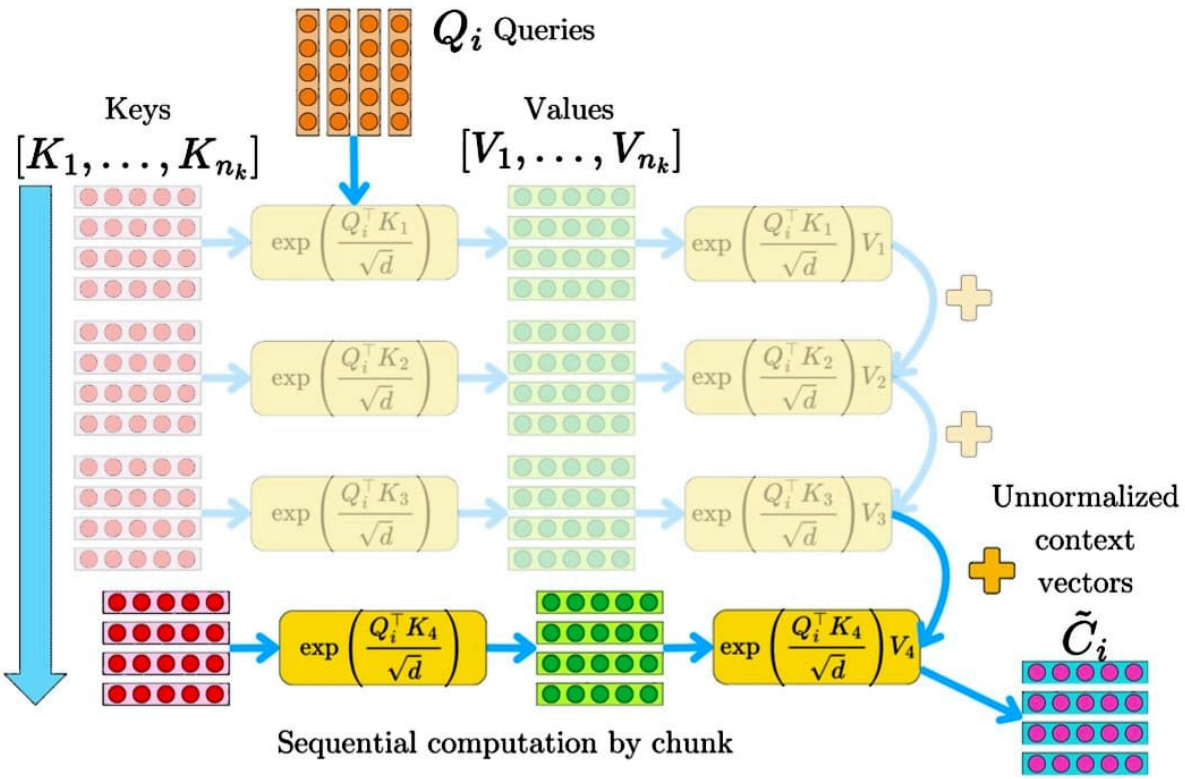
where each Q_i is a $\frac{N}{n_q} \times d_{\text{model}}$ matrix and K_i, V_i are $\frac{N}{n_k} \times d_{\text{model}}$ matrices. Let's call $N_q = \frac{N}{n_q}$, the number of queries per chunk, and $N_k = \frac{N}{n_k}$ the number of key-value pairs per chunk. We can now iterate through the chunks exactly in the same way:

for $i = 1$ to n_q **do** ▷ Process each query chunk
  $\tilde{C}_i \leftarrow \mathbf{0}$ ▷ Initialize output accumulator
  $S_i \leftarrow \mathbf{0}$ ▷ Initialize denominator accumulator
  for $j = 1$ to n_k **do** ▷ Process each key-value chunk
    $A_{ij} \leftarrow \exp\left(\frac{Q_i K_j^\top}{\sqrt{d_{\text{model}}}}\right)$
    $\tilde{C}_i \leftarrow \tilde{C}_i + A_{ij} V j$
    $S_i \leftarrow S_i + \sum_{j=1}^{N_k} A_{ij}$
  end for
  $C_i \leftarrow \tilde{C}_i \oslash S_i$ ▷ Normalize output (element-wise division)
end for
Return: $C = [C_1; C_2; \ldots; C_{n_q}]$ ▷ Concatenate results

As before, we need to store intermediary values of $\tilde{C}_i$ and S_i. In this context, $Q_i K_j^\top$ is a matrix of size $N_q \times N_k$, and so is A_{ij}. $\tilde{C}_i$ is a matrix of size $N_q \times d_{\text{model}}$ and S_i is a vector of size N_q. Therefore the space complexity is $\mathcal{O}(N_q \times N_k + N_q \times d_{\text{model}})$. To balance the number of chunks and the number of key-value pairs per chunk, they chose $N_k = n_k = \sqrt{N}$ and fixed $N_q = 1024$. This results in a space complexity:

$$
\mathcal{O}(1024\sqrt{N} + 1024 d_{\text{model}}) = \mathcal{O}(\sqrt{N}) \tag{7}
$$

This approach allows for efficient tensor operations within each chunk while dramatically reducing the peak memory requirements. Note that no approximation has been made, and it is mathematically equivalent to the vanilla attention mechanism. However, this approach is slower (8-13% slower during the forward pass and 30-35% slower during the backward pass) due to the sequential computations, but it enables the processing of much longer sequences that would otherwise be impossible due to memory constraints. It is one of the memory optimization strategies used in the *xFormers* packaged developed by Meta[1] and used in the development of the Llama models.



References

[1] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[2] Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.