Image source: Midjourney prompted by THE DECODER

LLaMA from scratch (LLaMA 1 and LLaMA 2)

Umar Jamil

Prerequisites

- Structure of the Transformer model and how the attention mechanism works.
- Training and inference of a Transformer model
- Linear Algebra: matrix multiplication, dot product
- Complex numbers: Euler's formula (not fundamental, nice to have)

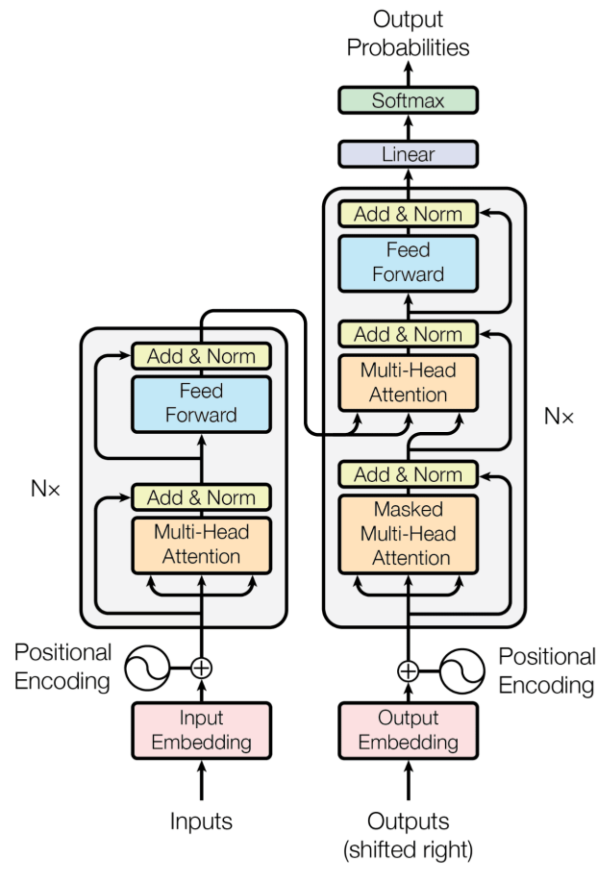
$$e^{ix} = \cos x + i \sin x$$

Topics

- Architectural differences between the vanilla Transformer and LLaMA
- RMS Normalization (with review of Layer Normalization)
- Rotary Positional Embeddings
- KV-Cache
- Multi-Query Attention
- Grouped Multi-Query Attention
- SwiGLU Activation Function

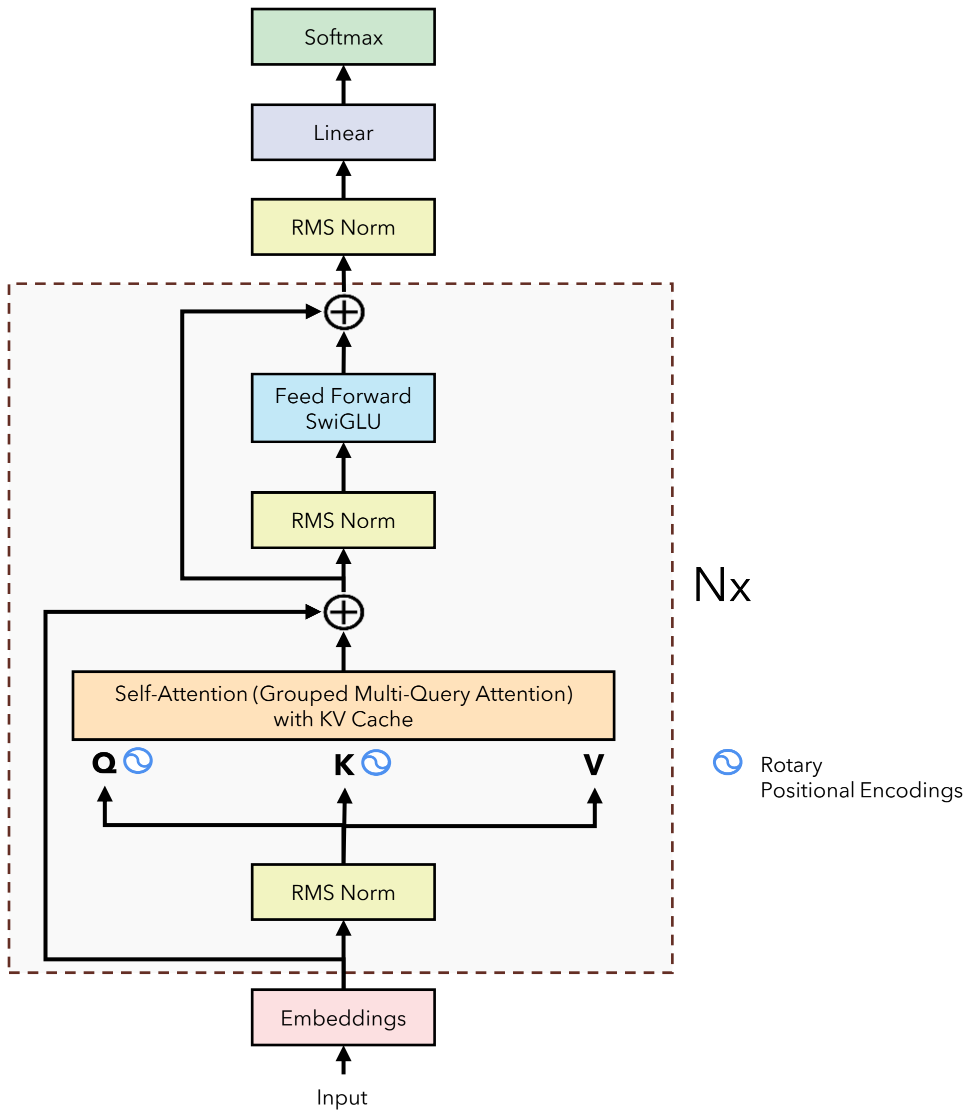
Sometimes, in order to introduce the topic, I will review concepts that you may already be familiar with. Feel free to skip those parts.

Transformer vs LLaMA



Transformer
("Attention is all you need")

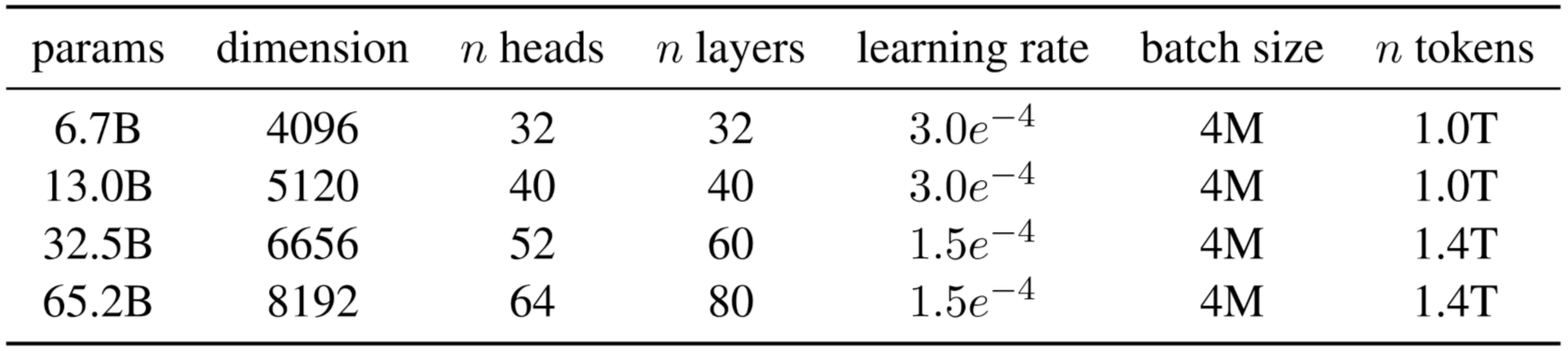
RMS Norm is added before every layer
Rotary positional encoding is applied only for the Q, K


Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

LLaMA

Umar Jamil – https://github.com/hkproj/pytorch-llama-notes

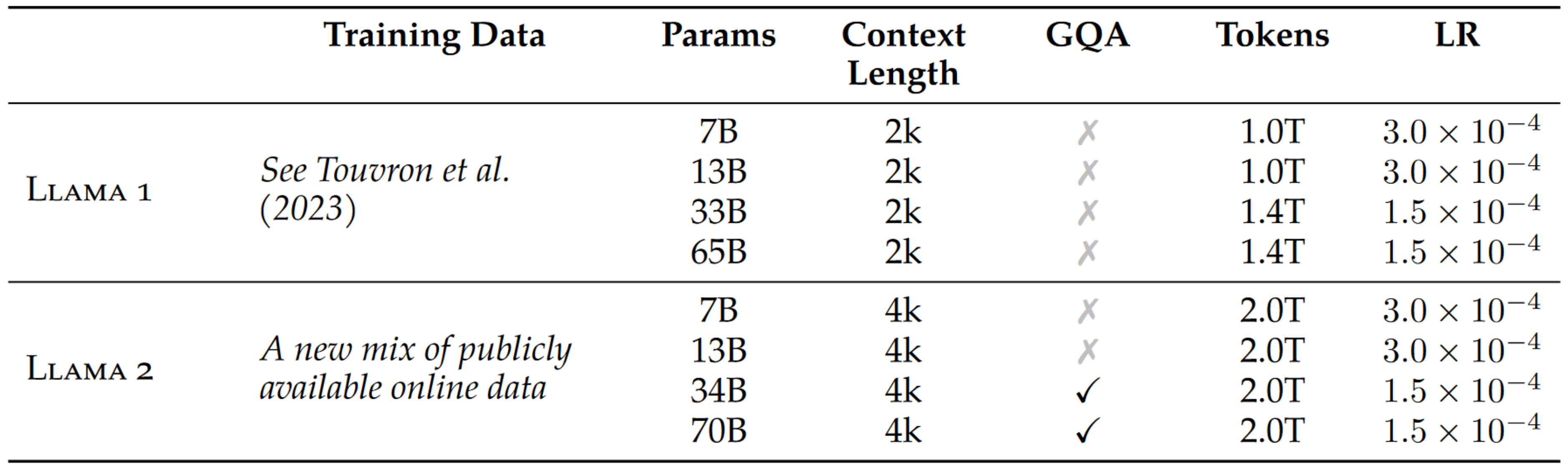
Models (LLaMA 1)



params	dimension	n heads	n layers	learning rate	batch size	n tokens
6.7B	4096	32	32	$3.0e^{-4}$	4M	1.0T
13.0B	5120	40	40	$3.0e^{-4}$	4M	1.0T
32.5B	6656	52	60	$1.5e^{-4}$	4M	1.4T
65.2B	8192	64	80	$1.5e^{-4}$	4M	1.4T

Table 2: **Model sizes, architectures, and optimization hyper-parameters.**

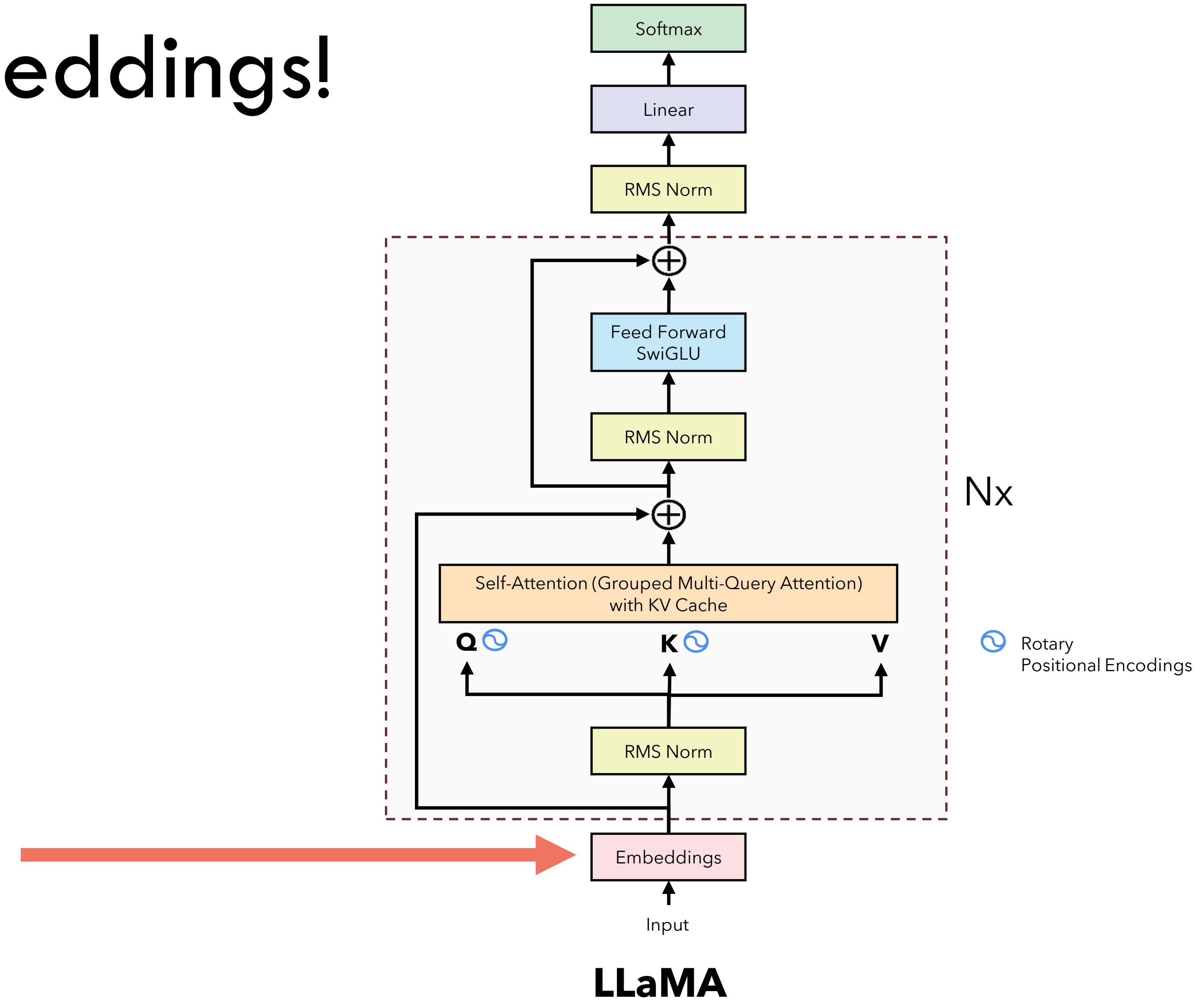
Models (LLaMA 2)



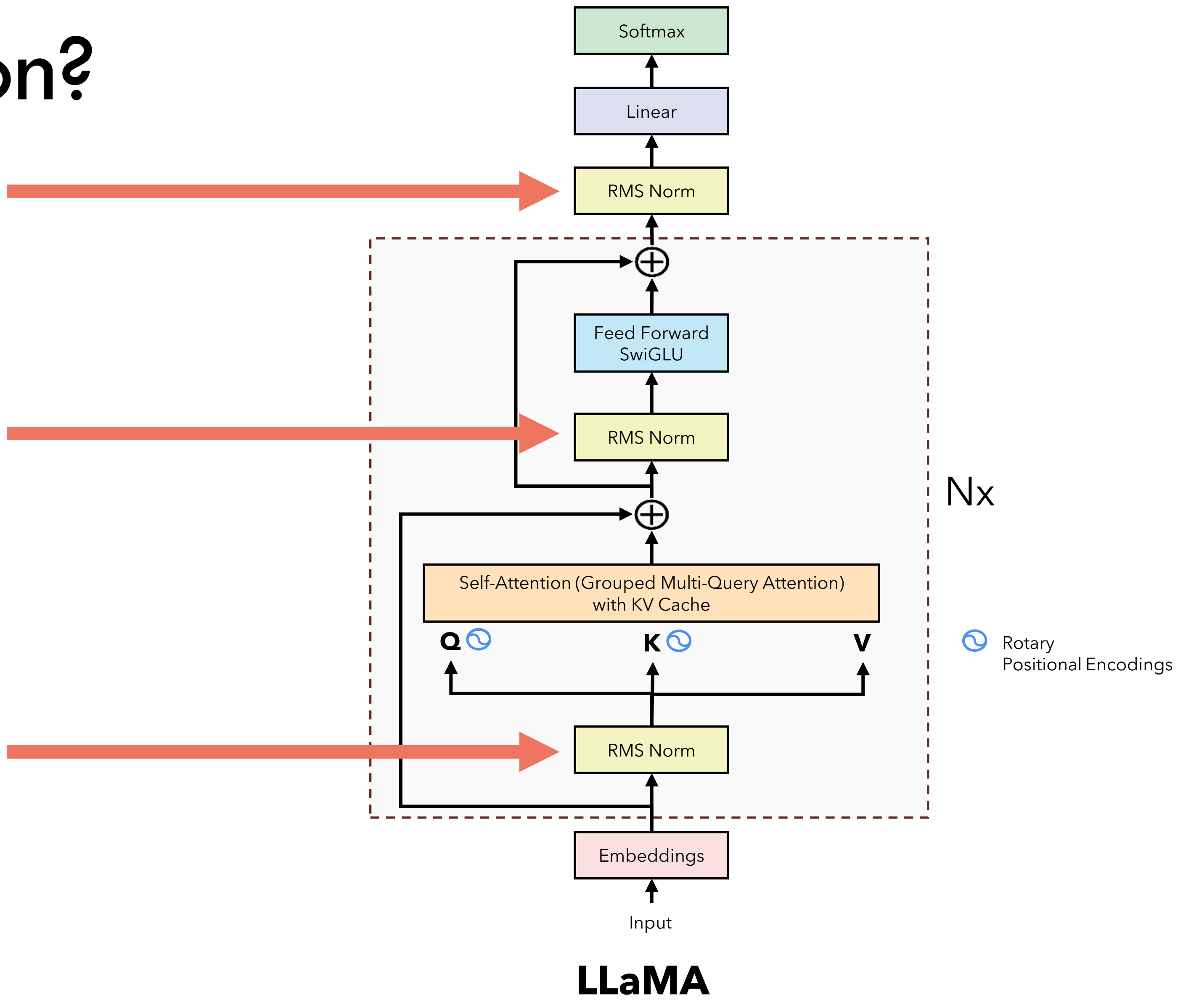
	Training Data	Params	Context Length	GQA	Tokens	LR
LLAMA 1	*See Touvron et al. (2023)*	7B	2k	✗	1.0T	3.0×10^{-4}
		13B	2k	✗	1.0T	3.0×10^{-4}
		33B	2k	✗	1.4T	1.5×10^{-4}
		65B	2k	✗	1.4T	1.5×10^{-4}
LLAMA 2	*A new mix of publicly available online data*	7B	4k	✗	2.0T	3.0×10^{-4}
		13B	4k	✗	2.0T	3.0×10^{-4}
		34B	4k	✓	2.0T	1.5×10^{-4}
		70B	4k	✓	2.0T	1.5×10^{-4}

Table 1: LLAMA 2 family of models. Token counts refer to pretraining data only. All models are trained with a global batch-size of 4M tokens. Bigger models — 34B and 70B — use Grouped-Query Attention (GQA) for improved inference scalability.

Let's review the Embeddings!

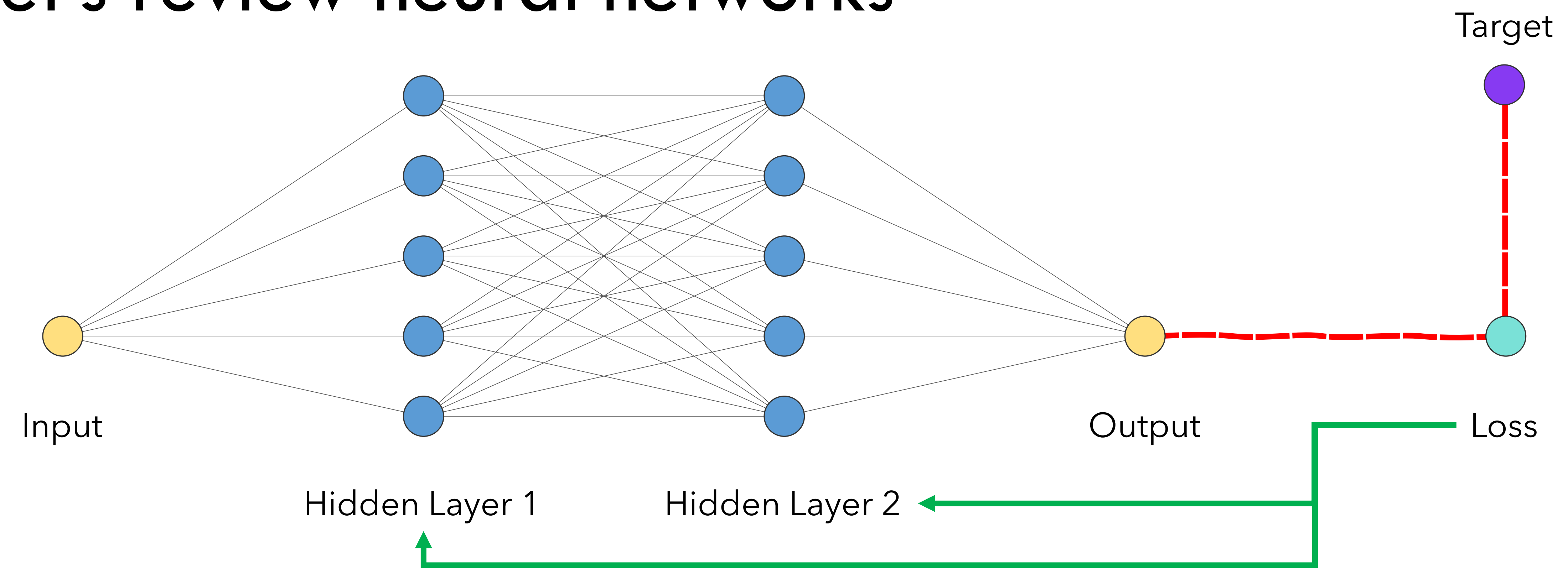

Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA

What is normalization?

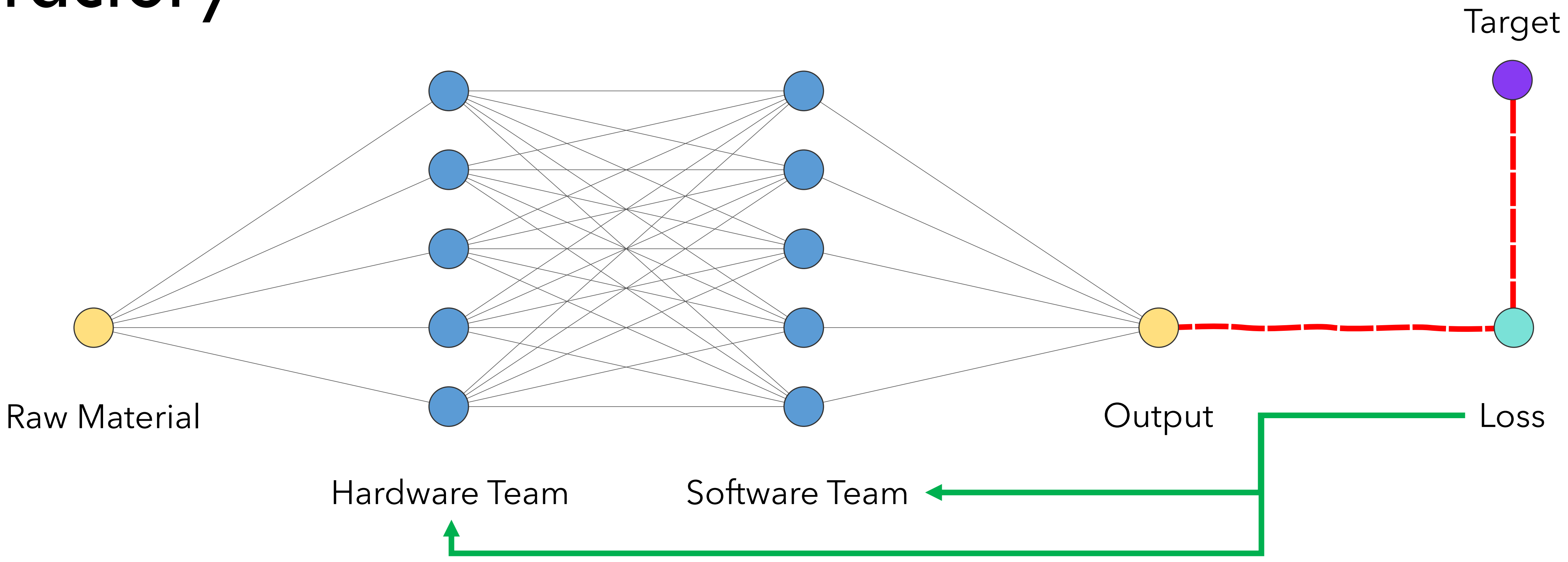


Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA

What is normalization?
Let's review neural networks

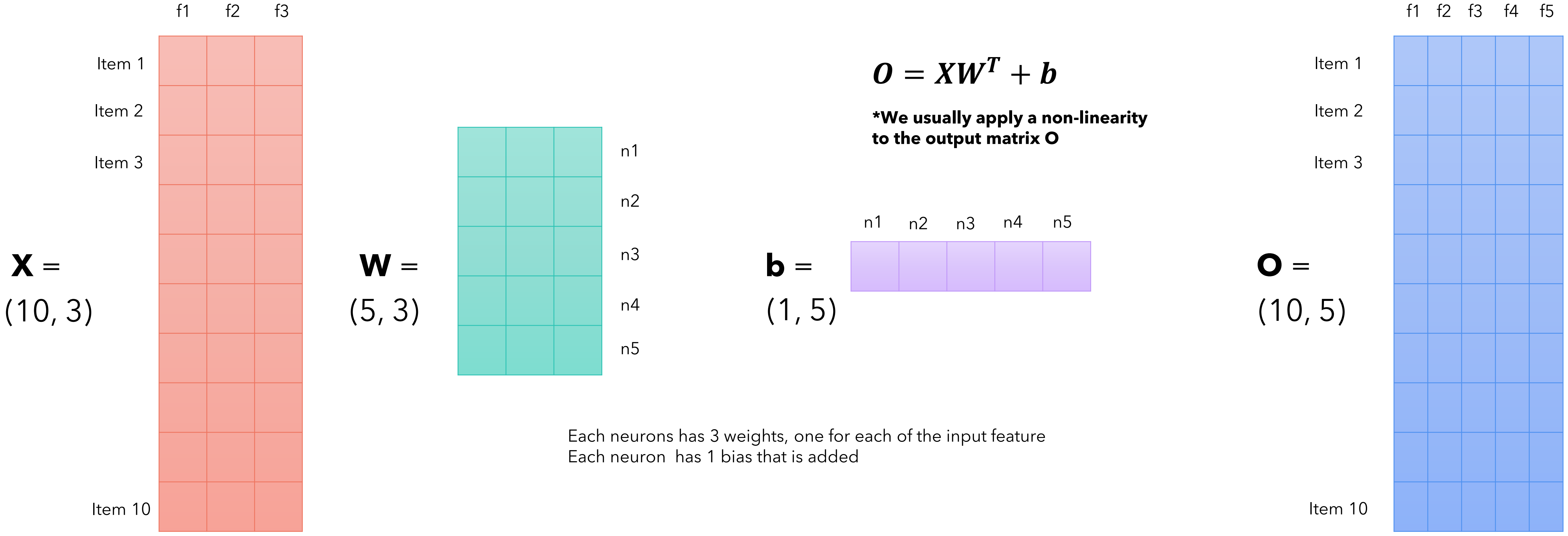


A simple parallel: the bad CEO in a phone factory



What is normalization?
Let's review neural networks' **maths!**

Suppose we have a linear layer, defined as **nn.Linear(in_features=3, out_features=5, bias=True).** This linear layer will create two matrices, called **W** (weight) and **b** (bias). If we have an input **X** of shape (10, 3) the output **O** will be (10, 5). But how does this happen mathematically?



$$O = XW^T + b$$

***We usually apply a non-linearity to the output matrix O**

Each neurons has 3 weights, one for each of the input feature
Each neuron has 1 bias that is added

Let's review neural networks' **maths!**

$$z_1 = (r_1 + b_1) = \left(\sum_{i=1}^{3} a_i w_i + b_1\right)$$

The output of the neuron 1 for the item 1 only depends on the features of the item 1. Usually we apply a non-linearity like the ReLU function to the output z_1. z_1 is referred to as the activation of the neuron 1 w.r.t the data item 1.

$$\mathbf{b} = \begin{array}{c|c|c|c|c} n1 & n2 & n3 & n4 & n5 \\ \hline b_1 & & & & \end{array}$$

(1, 5)

The bias vector will be broadcasted to every row in the XW^T table.

$$+$$

$$O = XW^T + b$$

$$\mathbf{X} = \begin{array}{c|c|c|c} & f1 & f2 & f3 \\ \hline \text{Item 1} & a_1 & a_2 & a_3 \\ \text{Item 2} & & & \\ \text{Item 3} & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ \text{Item 10} & & & \end{array}$$

(10, 3)

$$W^T = \begin{array}{c|c|c|c|c} n1 & n2 & n3 & n4 & n5 \\ \hline w_1 & & & & \\ w_2 & & & & \\ w_3 & & & & \end{array}$$

(3, 5)

$$XW^T = \begin{array}{c|c|c|c|c|c} & f1 & f2 & f3 & f4 & f5 \\ \hline \text{Item 1} & r_1 & & & & \\ \text{Item 2} & & & & & \\ \text{Item 3} & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ \text{Item 10} & & & & & \end{array}$$

(10, 5)

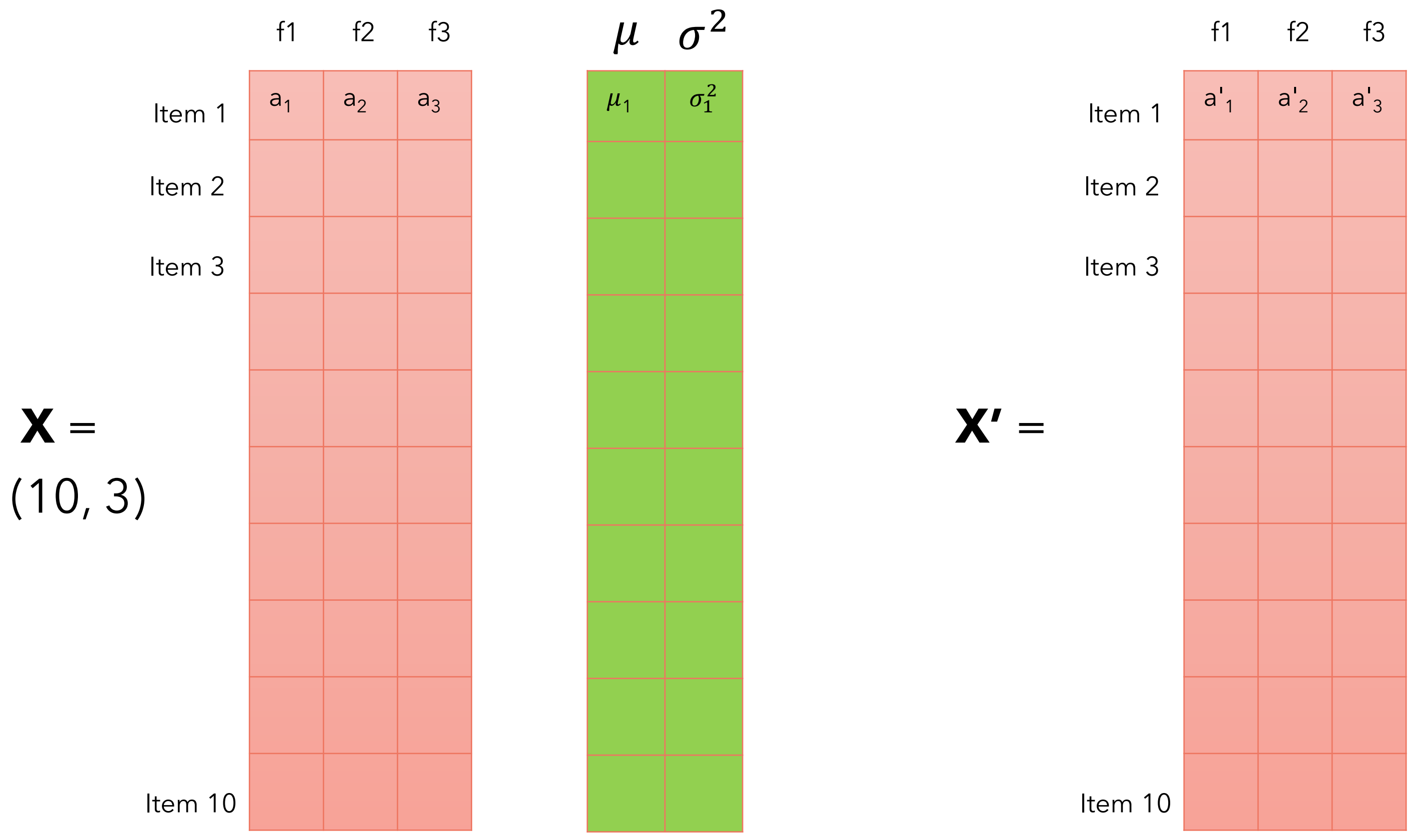
$$O = \begin{array}{c|c|c|c|c|c} & f1 & f2 & f3 & f4 & f5 \\ \hline \text{Item 1} & z_1 & & & & \\ \text{Item 2} & & & & & \\ \text{Item 3} & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ \text{Item 10} & & & & & \end{array}$$

(10, 5)

Let’s review neural networks’ **maths!**

- The output of a neuron for a data item depends on the features of the input data item (and the neuron’s parameters).
- We can think of the input to a neuron as the output of a previous linear.
- If the previous layer, after its weights are updated because of gradient descent, changes drastically its output, the next layer will have its input changed drastically, so it will be forced to re-adjust its weights drastically in turn at the next step of gradient descent.
- The phenomenon by which the distributions of internal nodes (neurons) of a neural network change is referred to as **Internal Covariate Shift**. And we want to avoid it because it makes training the network slower, as the neurons are forced to re-adjust drastically their weights in one direction or another because of drastic changes in the outputs of the previous layers.

A solution to jumping activations: layer normalization!



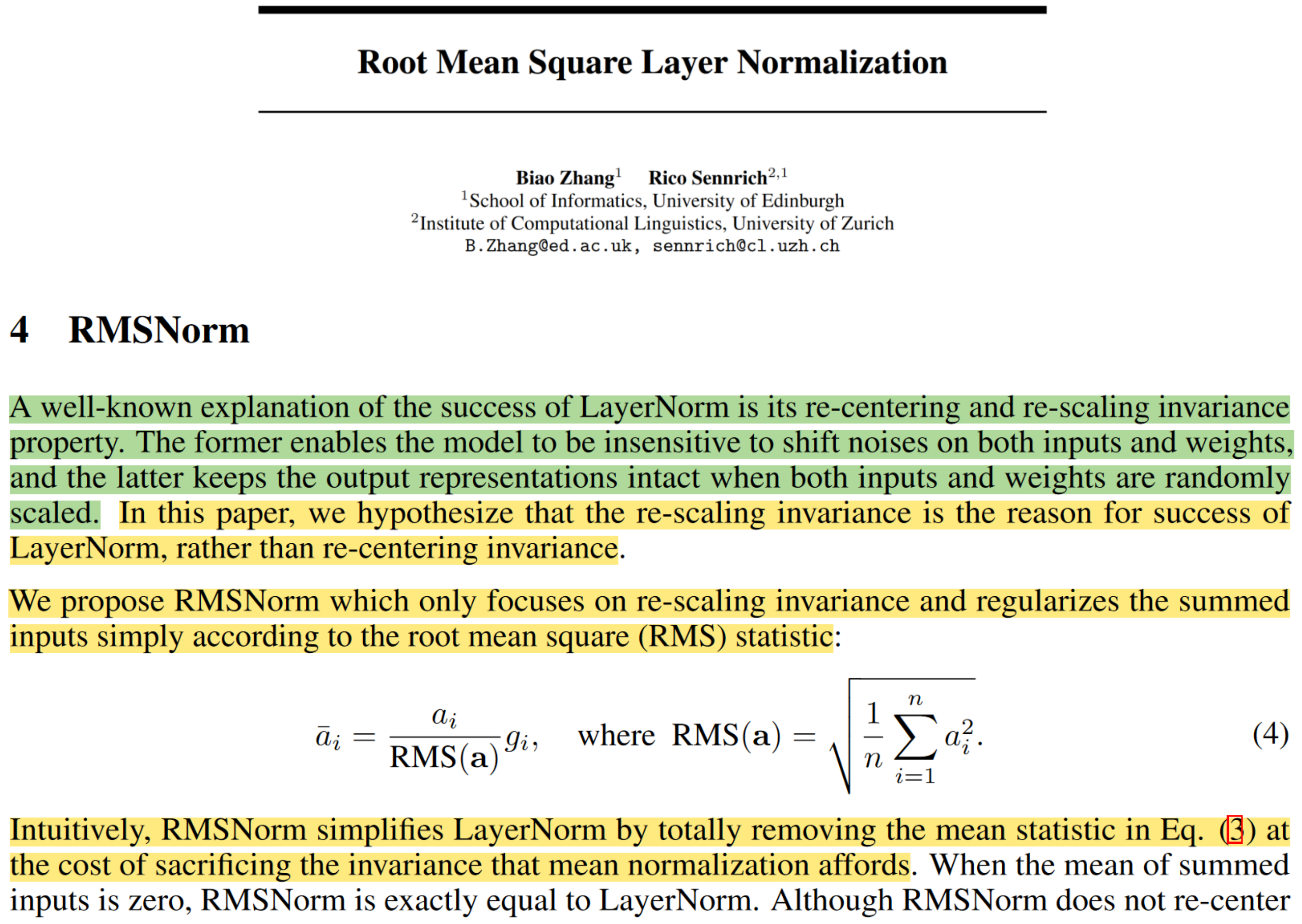
$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

- Each item is updated with its normalized value, which will turn it into a normal distribution with 0 mean and variance of 1.
- The two parameters **gamma** and **beta** are learnable parameters that allow the model to "amplify" the scale of each feature or apply a translation to the feature according to the needs of the loss function.

With batch normalization we normalize by **columns (features)**

With layer normalization we normalize by **rows (data items)**

Root Mean Square Normalization



Root Mean Square Layer Normalization

Biao Zhang[1] **Rico Sennrich**[2,1]
[1]School of Informatics, University of Edinburgh
[2]Institute of Computational Linguistics, University of Zurich
`B.Zhang@ed.ac.uk, sennrich@cl.uzh.ch`

4 RMSNorm

A well-known explanation of the success of LayerNorm is its re-centering and re-scaling invariance property. The former enables the model to be insensitive to shift noises on both inputs and weights, and the latter keeps the output representations intact when both inputs and weights are randomly scaled. In this paper, we hypothesize that the re-scaling invariance is the reason for success of LayerNorm, rather than re-centering invariance.

We propose RMSNorm which only focuses on re-scaling invariance and regularizes the summed inputs simply according to the root mean square (RMS) statistic:

$$\bar{a}_i = \frac{a_i}{\mathrm{RMS}(\mathbf{a})} g_i, \quad \text{where } \mathrm{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}. \tag{4}$$

Intuitively, RMSNorm simplifies LayerNorm by totally removing the mean statistic in Eq. (3) at the cost of sacrificing the invariance that mean normalization affords. When the mean of summed inputs is zero, RMSNorm is exactly equal to LayerNorm. Although RMSNorm does not re-center

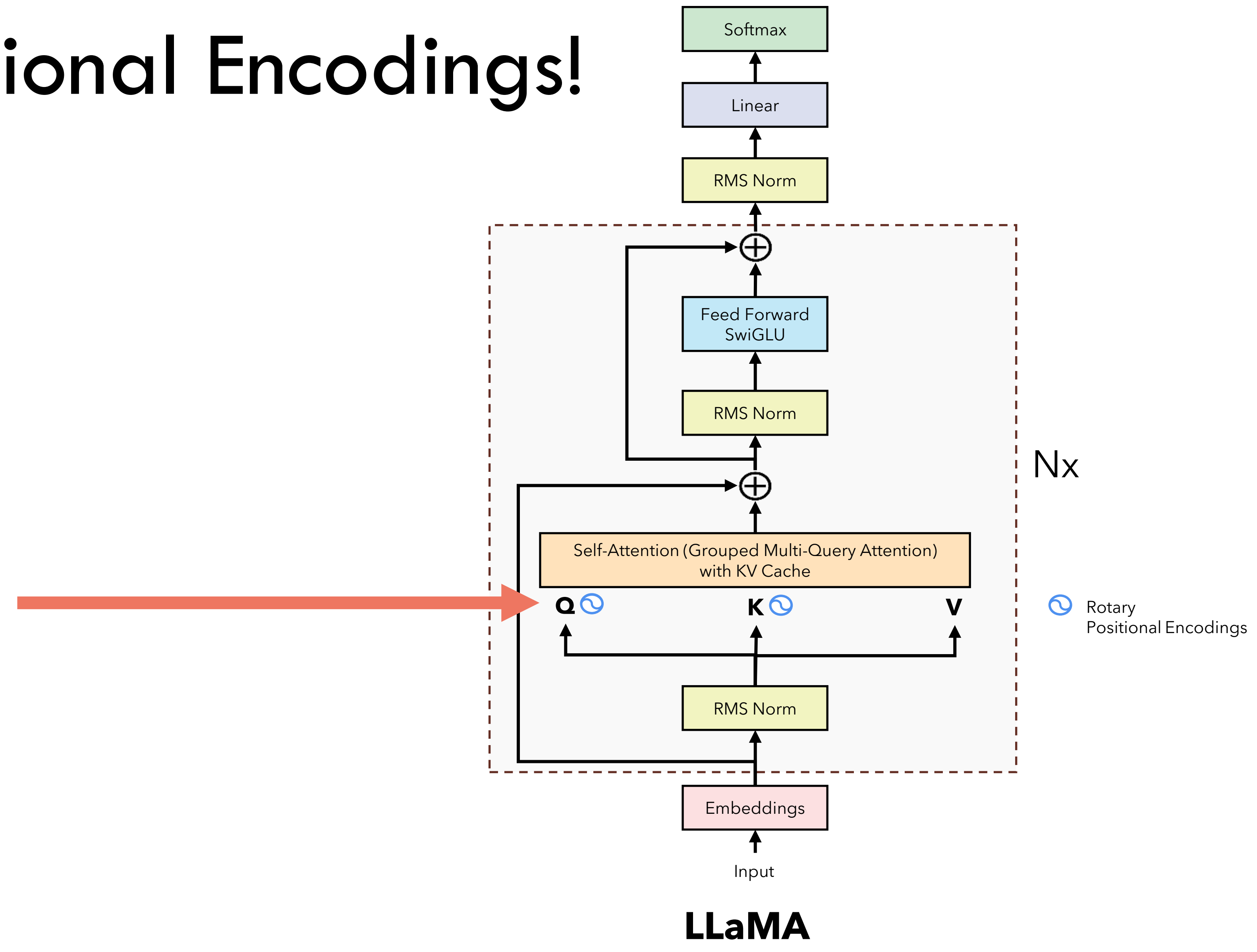
RMSNorm does not depend on the mean but the variance only

Just like Layer Normalization, we also have a learnable parameter **gamma** (**g** in the formula on the left) that is multiplied by the normalized values.

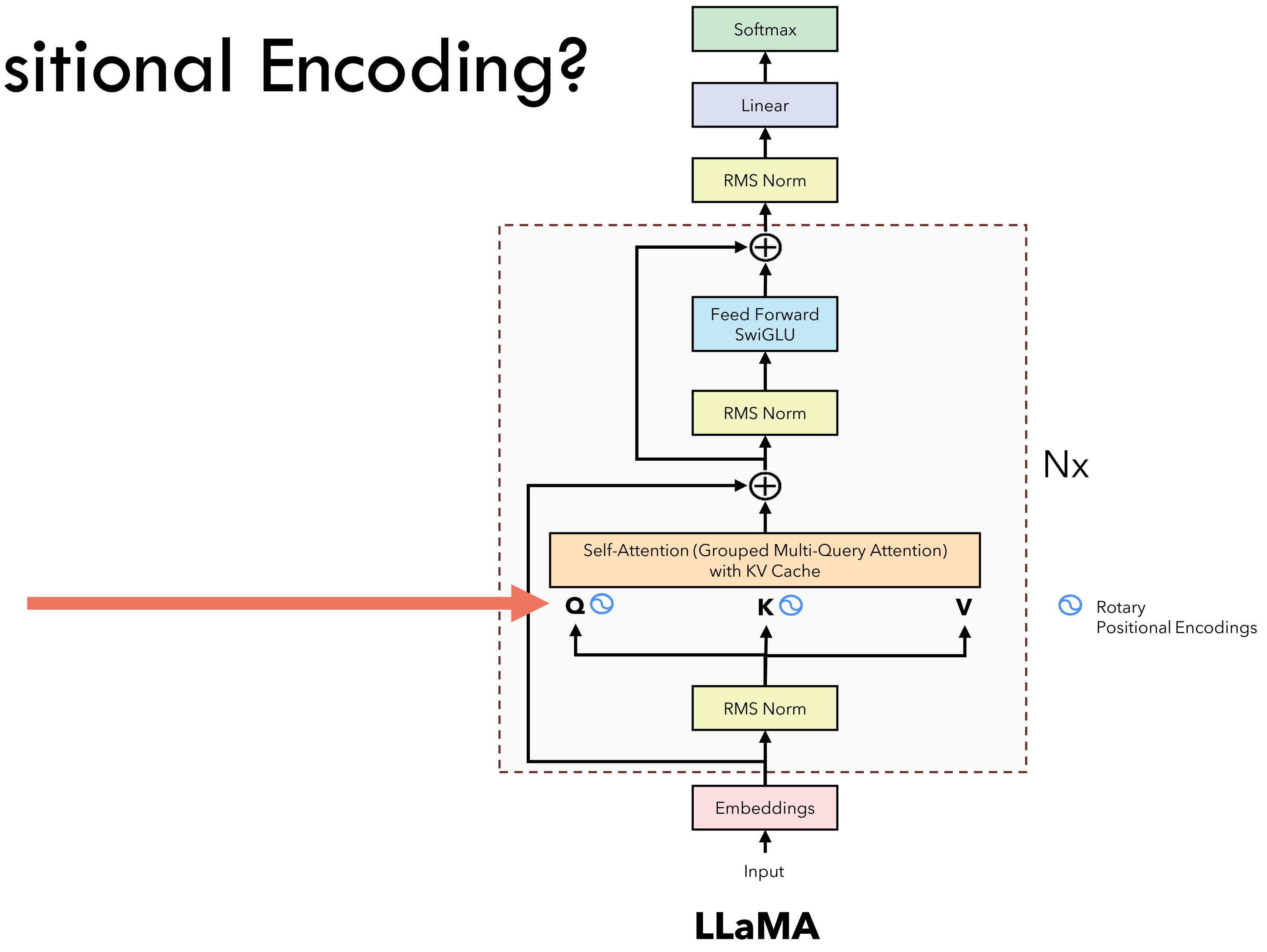
Why RMSNorm?

- Requires less computation compared to Layer Normalization.
- It works well in practice.

Let's review Positional Encodings!

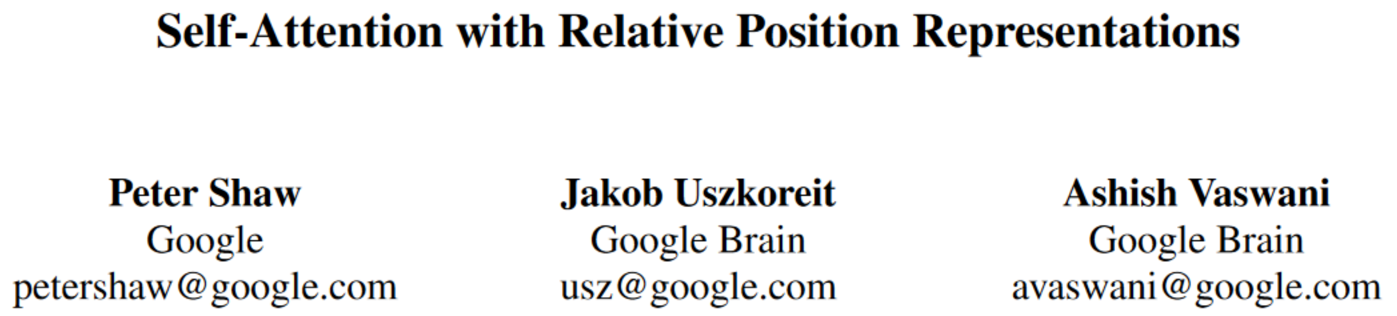

Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA

What is Rotary Positional Encoding?


Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA

What’s the difference between the absolute positional encodings and the relative ones?

- Absolute positional encodings are fixed vectors that are added to the embedding of a token to represent its absolute position in the sentence. So, it deals with **one token at a time**. You can think of it as the pair (latitude, longitude) on a map: each point on earth will have a unique pair.
- Relative positional encodings, on the other hand, deals with two tokens at a time and it is involved when we calculate the attention: since the attention mechanism captures the “intensity” of how much two words are related two each other, relative positional encodings tells the attention mechanism the **distance** between the two words involved in it. So, given **two tokens**, we create a vector that represents their distance.
- Relative positional encodings were introduced in the following paper



Self-Attention with Relative Position Representations

Peter Shaw
Google
petershaw@google.com

Jakob Uszkoreit
Google Brain
usz@google.com

Ashish Vaswani
Google Brain
avaswani@google.com

Absolute Positional Encodings
From "*Attention is all you need*"

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}}$$

Now Kiss!
ATTENTION MECHANISM
TOKEN 1 + ABS. POSITION
TOKEN 2 + ABS. POSITION
imgflip.com

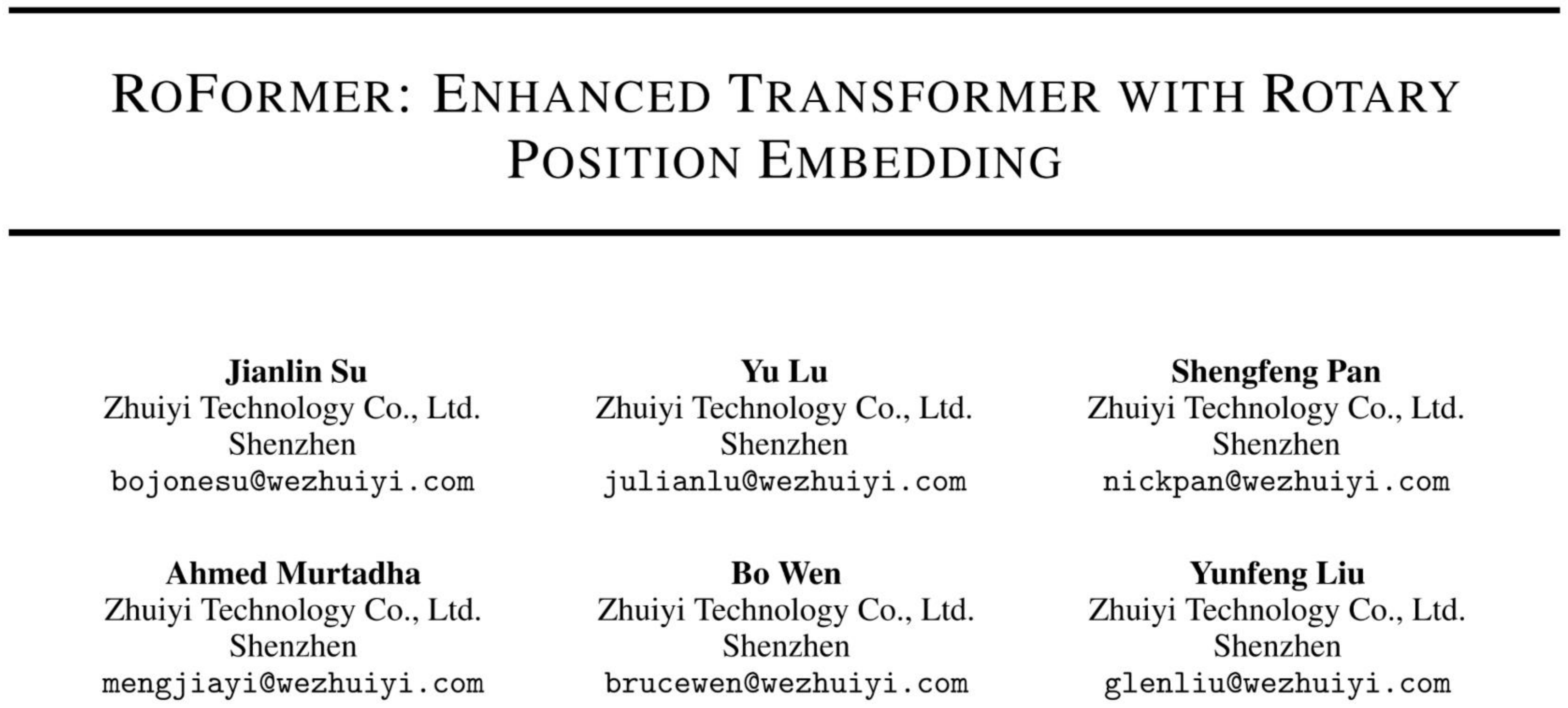
Relative Positional Encodings
From "*Self-Attention with relative positional representations*"

$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$$

Now Kiss!
ATTENTION MECHANISM
TOKEN 1
DISTANCE
TOKEN 2
imgflip.com

*NSFW: Menage a trois

Rotary Position Embeddings



ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING

Jianlin Su
Zhuiyi Technology Co., Ltd.
Shenzhen
`bojonesu@wezhuiyi.com`

Yu Lu
Zhuiyi Technology Co., Ltd.
Shenzhen
`julianlu@wezhuiyi.com`

Shengfeng Pan
Zhuiyi Technology Co., Ltd.
Shenzhen
`nickpan@wezhuiyi.com`

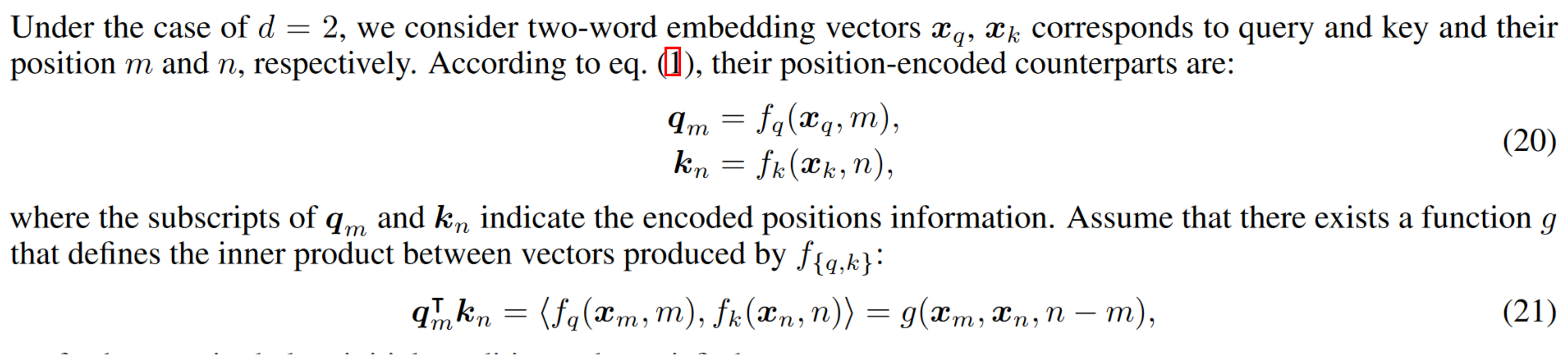
Ahmed Murtadha
Zhuiyi Technology Co., Ltd.
Shenzhen
`mengjiayi@wezhuiyi.com`

Bo Wen
Zhuiyi Technology Co., Ltd.
Shenzhen
`brucewen@wezhuiyi.com`

Yunfeng Liu
Zhuiyi Technology Co., Ltd.
Shenzhen
`glenliu@wezhuiyi.com`

Rotary Position Embeddings: the inner product

- The **dot product** used in the attention mechanism is a type of *inner product*, which can be through of as a generalization of the dot product.
- Can we find an inner product over the two vectors **q** (query) and **k** (key) used in the attention mechanism that only depends on the two vectors and the relative distance of the token they represent?



Under the case of $d = 2$, we consider two-word embedding vectors $\boldsymbol{x}_q$, $\boldsymbol{x}_k$ corresponds to query and key and their position m and n, respectively. According to eq. (1), their position-encoded counterparts are:

$$\begin{aligned} \boldsymbol{q}_m &= f_q(\boldsymbol{x}_q, m), \\ \boldsymbol{k}_n &= f_k(\boldsymbol{x}_k, n), \end{aligned} \tag{20}$$

where the subscripts of $\boldsymbol{q}_m$ and $\boldsymbol{k}_n$ indicate the encoded positions information. Assume that there exists a function g that defines the inner product between vectors produced by $f_{\{q,k\}}$:

$$\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = \langle f_q(\boldsymbol{x}_m, m), f_k(\boldsymbol{x}_n, n) \rangle = g(\boldsymbol{x}_m, \boldsymbol{x}_n, n - m), \tag{21}$$

Rotary Position Embeddings: the inner product

- We can define a function **g** like the following that only depends on the on the two embeddings vector **q** and **k** and their relative distance

$$f_q(\boldsymbol{x}_m, m) = (\boldsymbol{W}_q \boldsymbol{x}_m) e^{im\theta}$$
$$f_k(\boldsymbol{x}_n, n) = (\boldsymbol{W}_k \boldsymbol{x}_n) e^{in\theta}$$
$$g(\boldsymbol{x}_m, \boldsymbol{x}_n, m-n) = \mathrm{Re}[(\boldsymbol{W}_q \boldsymbol{x}_m)(\boldsymbol{W}_k \boldsymbol{x}_n)^* e^{i(m-n)\theta}]$$

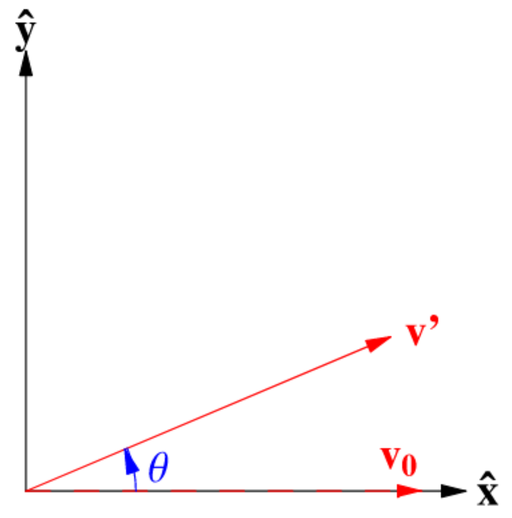
*** = Conjugate** of the complex number

- Using **Euler's formula**, we can write it into its matrix form.

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \underbrace{\begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix}}_{} \begin{pmatrix} W^{(11)}_{\{q,k\}} & W^{(12)}_{\{q,k\}} \\ W^{(21)}_{\{q,k\}} & W^{(22)}_{\{q,k\}} \end{pmatrix} \begin{pmatrix} x^{(1)}_m \\ x^{(2)}_m \end{pmatrix}$$

Rotation matrix in a 2d space, hence the name **rotary** position embeddings

Rotary Position Embeddings: the rotation matrix



In $\mathbb{R}^2$, consider the matrix that rotates a given vector $\mathbf{v}_0$ by a counterclockwise angle θ in a fixed coordinate system. Then

$$\mathsf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \tag{1}$$

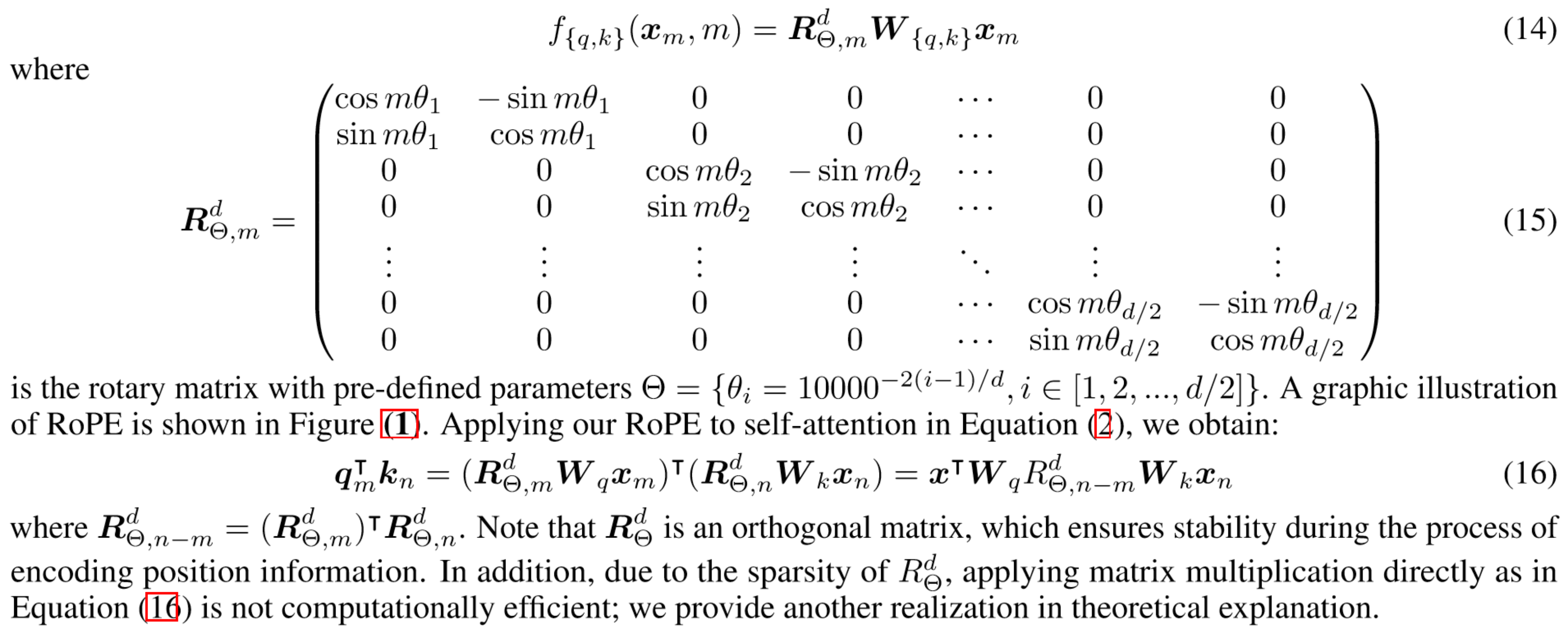
so

$$\mathbf{v}' = \mathsf{R}_\theta \, \mathbf{v}_0. \tag{2}$$

From **Wolfram MathWorld**: https://mathworld.wolfram.com/RotationMatrix.html

Rotary Position Embeddings: the general form

Since the matrix is **sparse**, it is not convenient to use it to compute the positional embeddings



$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m} \boldsymbol{W}_{\{q,k\}} \boldsymbol{x}_m \tag{14}$$

where

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix} \tag{15}$$

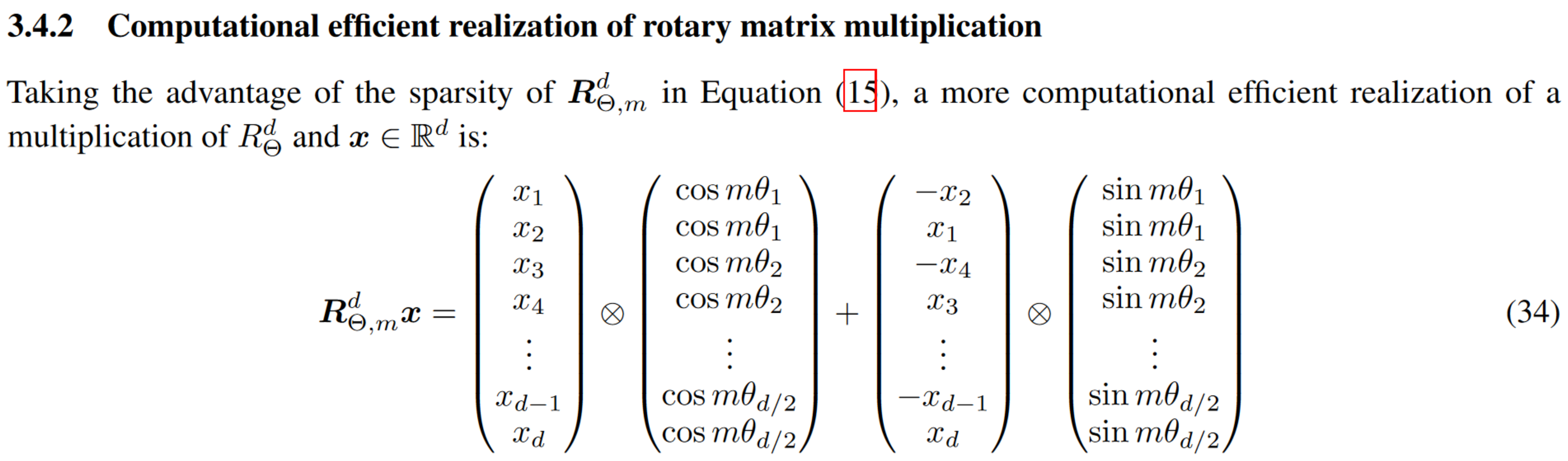
is the rotary matrix with pre-defined parameters $\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, ..., d/2]\}$. A graphic illustration of RoPE is shown in Figure (**1**). Applying our RoPE to self-attention in Equation (2), we obtain:

$$\boldsymbol{q}_m^\intercal \boldsymbol{k}_n = (\boldsymbol{R}^d_{\Theta,m} \boldsymbol{W}_q \boldsymbol{x}_m)^\intercal (\boldsymbol{R}^d_{\Theta,n} \boldsymbol{W}_k \boldsymbol{x}_n) = \boldsymbol{x}^\intercal \boldsymbol{W}_q R^d_{\Theta,n-m} \boldsymbol{W}_k \boldsymbol{x}_n \tag{16}$$

where $\boldsymbol{R}^d_{\Theta,n-m} = (\boldsymbol{R}^d_{\Theta,m})^\intercal \boldsymbol{R}^d_{\Theta,n}$. Note that $\boldsymbol{R}^d_{\Theta}$ is an orthogonal matrix, which ensures stability during the process of encoding position information. In addition, due to the sparsity of R^d_Θ, applying matrix multiplication directly as in Equation (16) is not computationally efficient; we provide another realization in theoretical explanation.

Rotary Position Embeddings: the computational-efficient form

- Given a token with embedding vector **x**, and the position **m** of the token inside the sentence, this is how we compute the position embeddings for the token.



3.4.2 Computational efficient realization of rotary matrix multiplication

Taking the advantage of the sparsity of $\boldsymbol{R}^d_{\Theta,m}$ in Equation (15), a more computational efficient realization of a multiplication of R^d_Θ and $\boldsymbol{x} \in \mathbb{R}^d$ is:

$$\boldsymbol{R}^d_{\Theta,m}\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix} \tag{34}$$

Rotary Position Embeddings: long-term decay

The authors calculated an **upper bound** for the inner product by varying the distance between two tokens and proved that it decays with the growth of the relative distance.
This means that the "intensity" of relationship between two tokens encoded with Rotary Positional Embeddings will be numerically smaller as the distance between them grows.

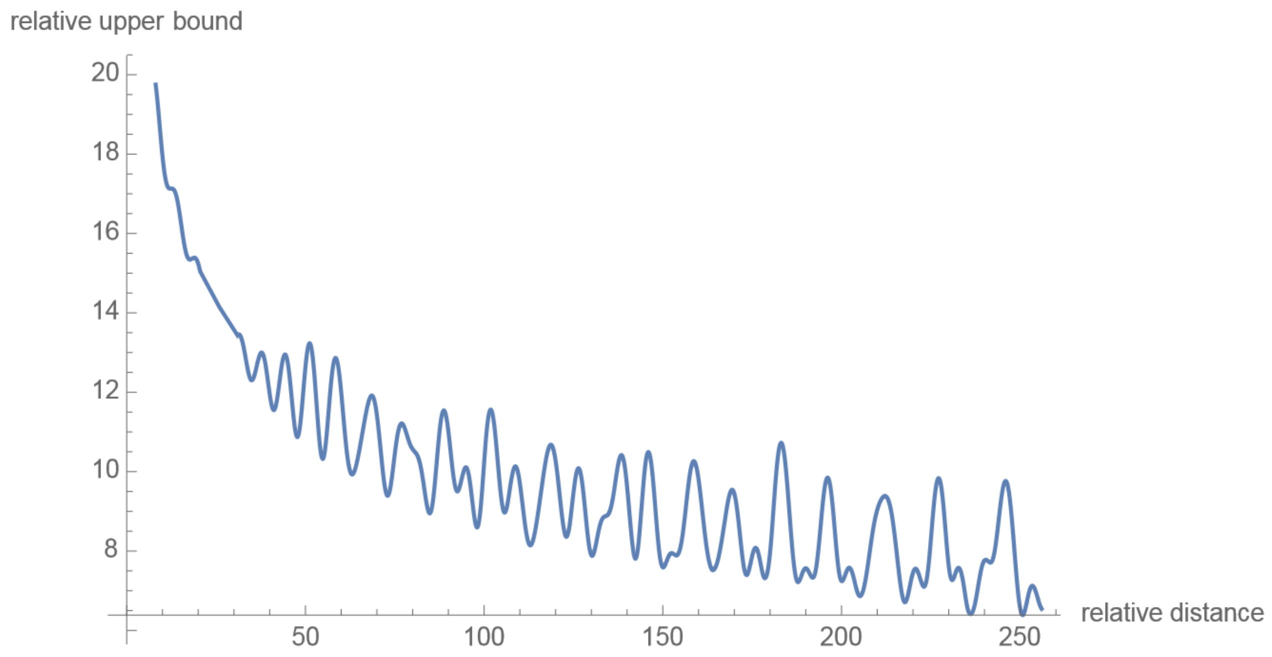
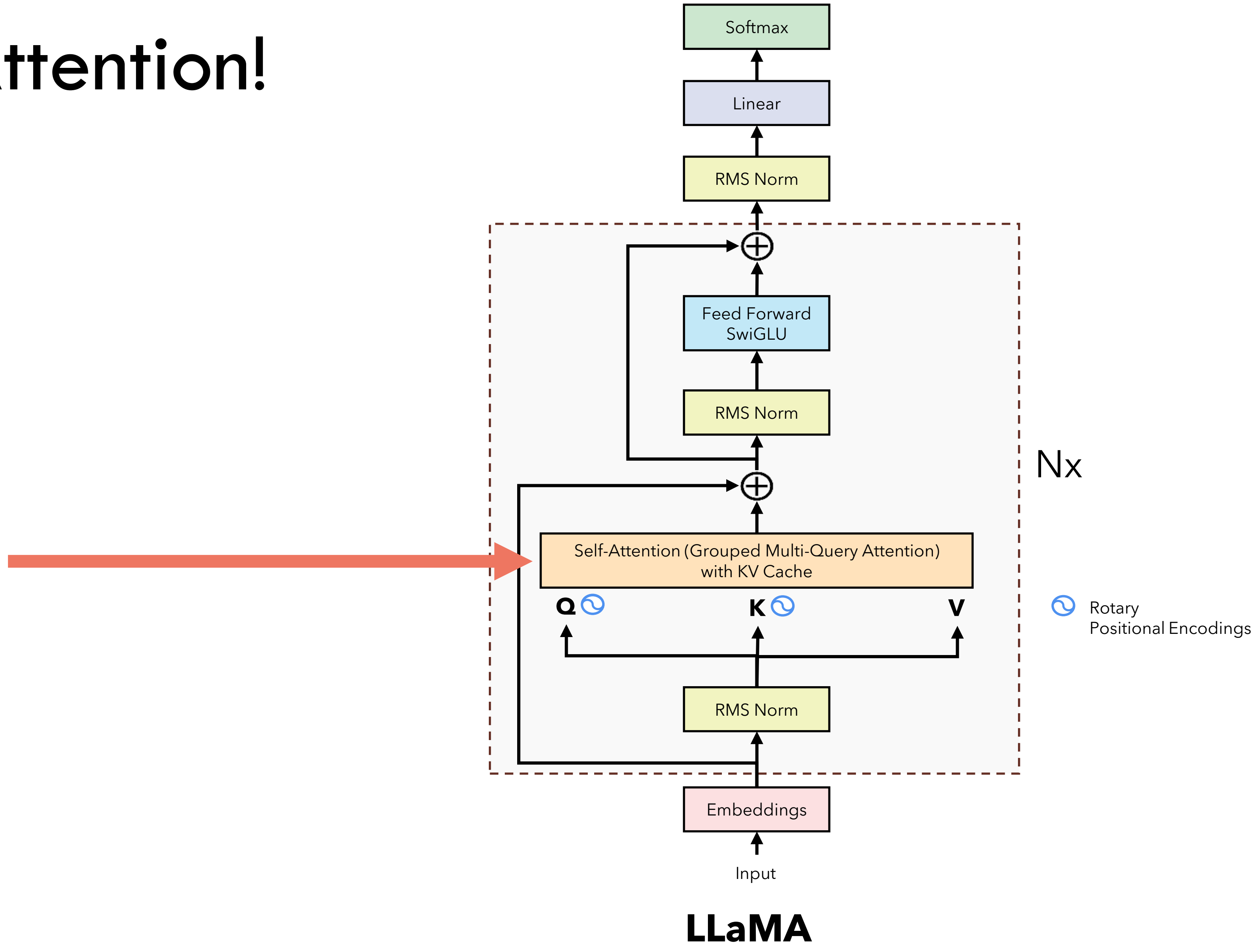


Figure 2: Long-term decay of RoPE.

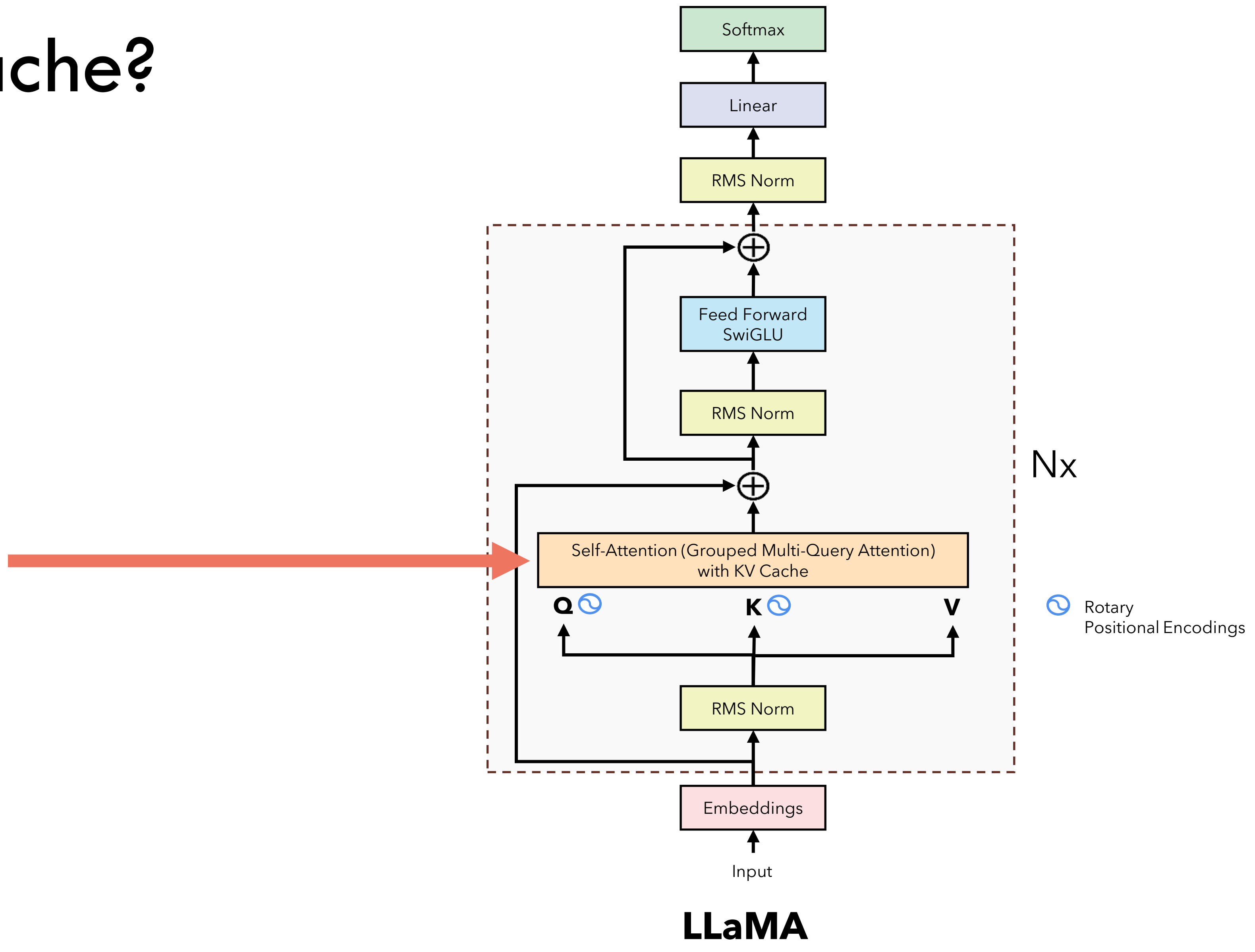
Rotary Position Embeddings: practical considerations

- The rotary position embeddings are **only applied to the query and the keys**, but not the values.
- The rotary position embeddings are applied after the vector **q** and **k** have been multiplied by the **W** matrix in the attention mechanism, while in the vanilla transformer they're applied before.

Let's review Self-Attention!


Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA

What is the KV Cache?


Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
RMS Norm
Embeddings
Input
Nx
Rotary Positional Encodings
LLaMA

Next Token Prediction Task

- Imagine we want to train a model to write Dante Alighieri's Divine Comedy's 5th Canto from the Inferno.

Amor, ch'al cor gentil ratto s'apprende,
prese costui de la bella persona
che mi fu tolta; e 'l modo ancor m'offende.

Amor, ch'a nullo amato amar perdona,
mi prese del costui piacer sì forte,
che, come vedi, ancor non m'abbandona.

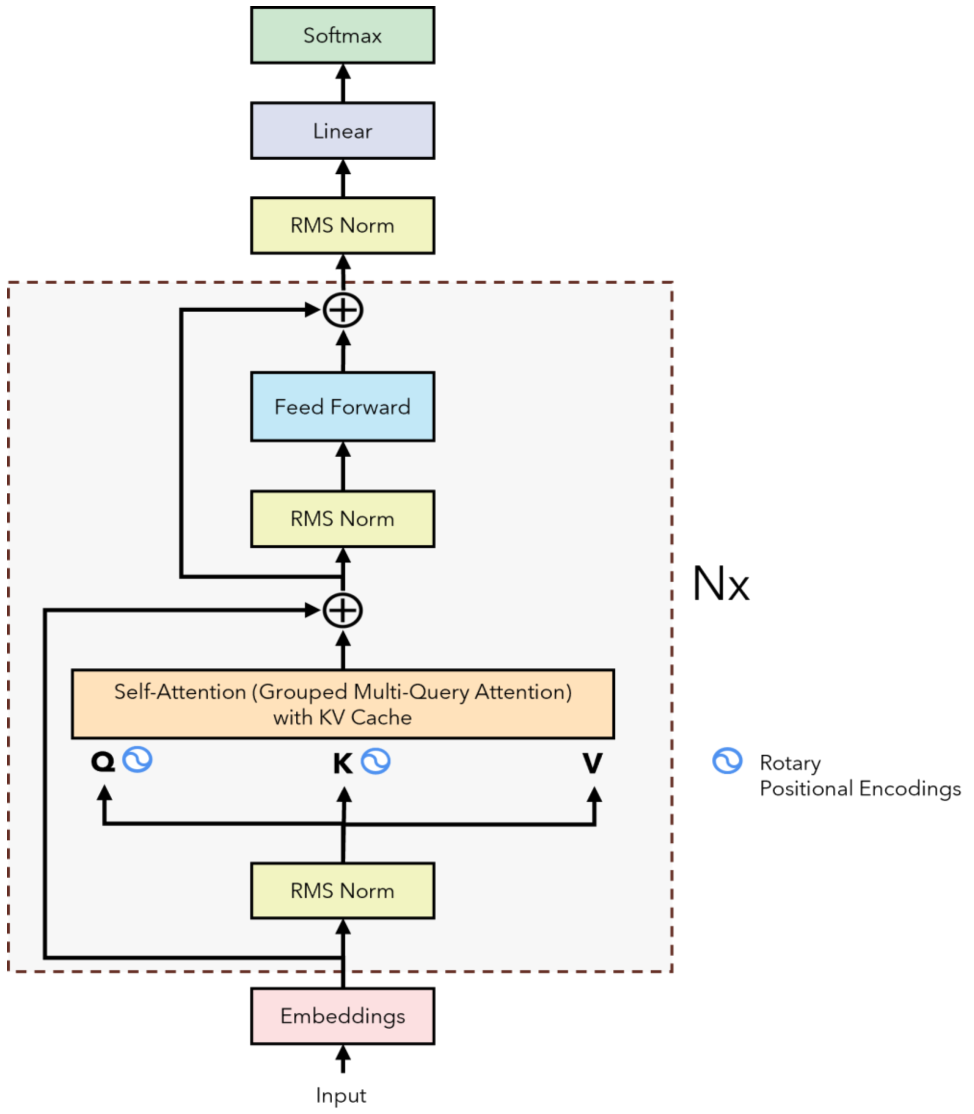
Amor condusse noi ad una morte.
Caina attende chi a vita ci spense.

Love, that can quickly seize the gentle heart,
took hold of him because of the fair body
taken from me—how that was done still wounds me.

Love, that releases no beloved from loving,
took hold of me so strongly through his beauty
that, as you see, it has not left me yet.

Love led the two of us unto one death.
Caina waits for him who took our life."

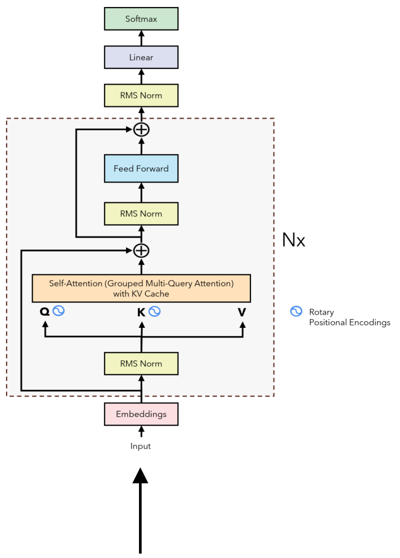
Source: https://digitaldante.columbia.edu/dante/divine-comedy/inferno/inferno-5/


Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Next Token Prediction Task

Target Love that can quickly seize the gentle heart [EOS]

Training

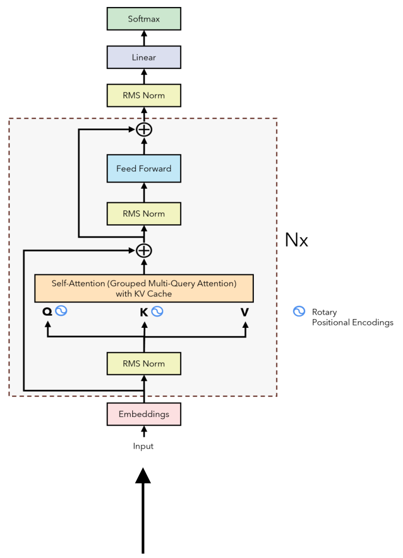

Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Input [SOS] Love that can quickly seize the gentle heart

Next Token Prediction Task: Inference

Output Love

Inference
T = 1

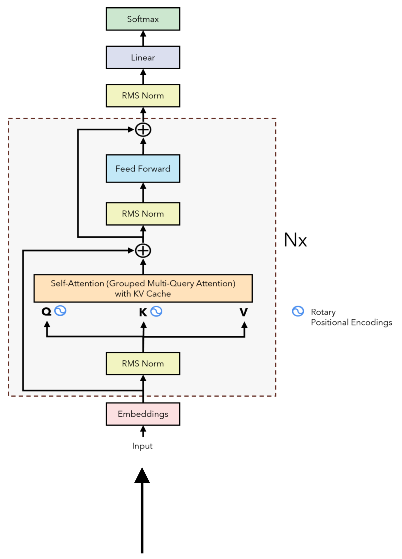

Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Input [SOS]

Next Token Prediction Task: Inference

Output Love that

Inference
T = 2



Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

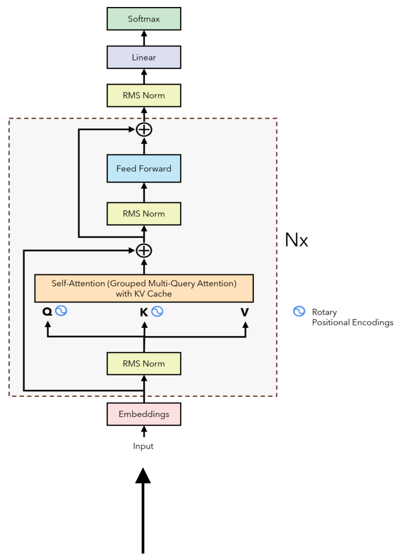
Input [SOS] Love

Next Token Prediction Task: Inference

Output Love that can

Inference

T = 3

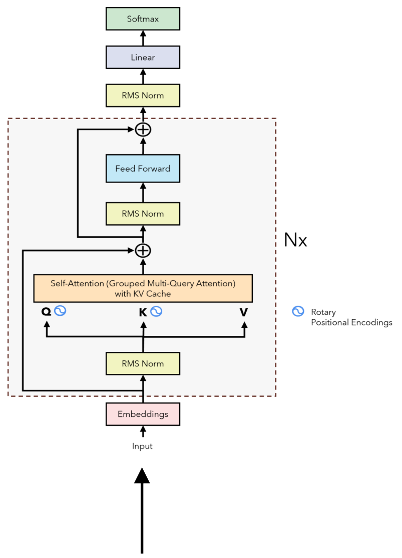

Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Input [SOS] Love that

Next Token Prediction Task: Inference

Output Love that can quickly

Inference
T = 4



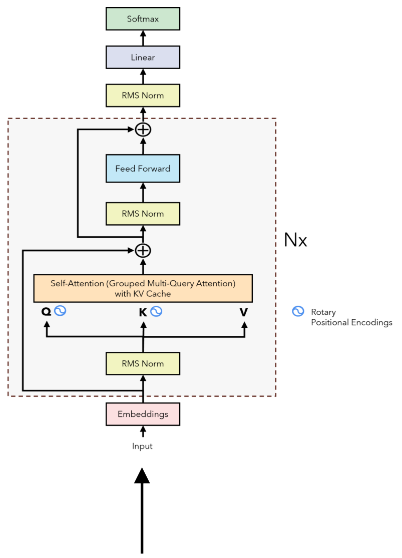
Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Input [SOS] Love that can

Next Token Prediction Task: Inference

Output Love that can quickly seize

Inference
T = 5

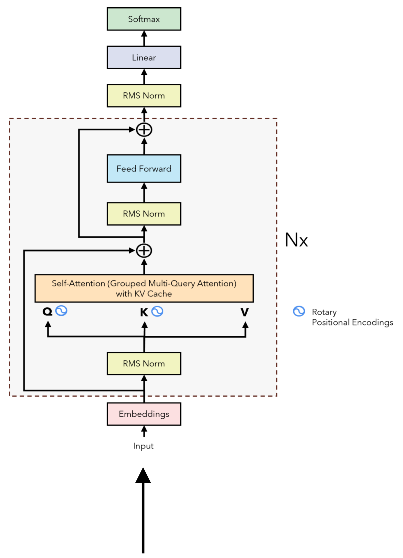

Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Input [SOS] Love that can quickly

Next Token Prediction Task: Inference

Output Love that can quickly seize the

Inference
T = 6

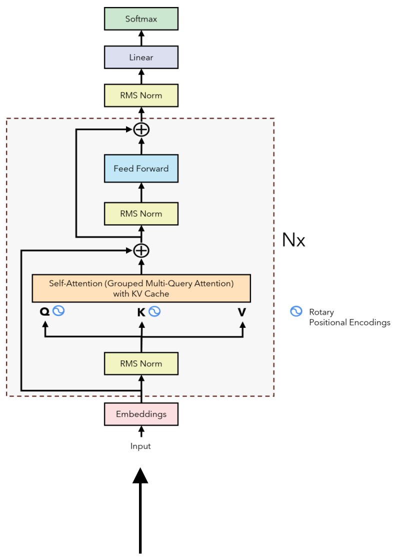

Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Input [SOS] Love that can quickly seize

Next Token Prediction Task: Inference

Output Love that can quickly seize the gentle

Inference
T = 7

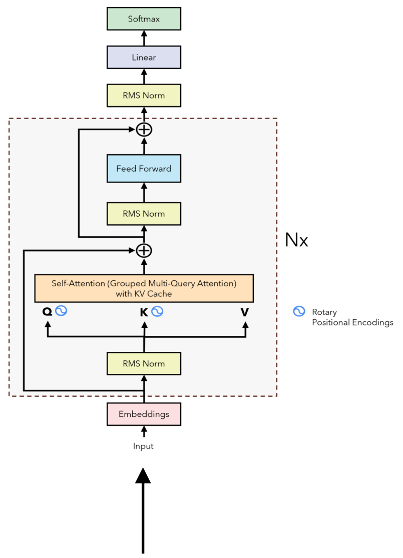

Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Input [SOS] Love that can quickly seize the

Next Token Prediction Task: Inference

Output Love that can quickly seize the gentle heart

Inference
T = 8

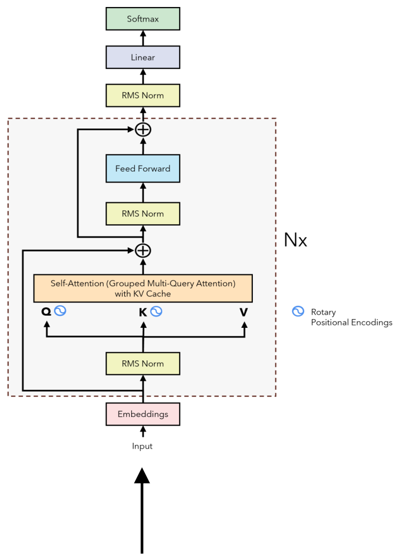

Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Input [SOS] Love that can quickly seize the gentle

Next Token Prediction Task: Inference

Output Love that can quickly seize the gentle heart [EOS]

Inference
T = 9



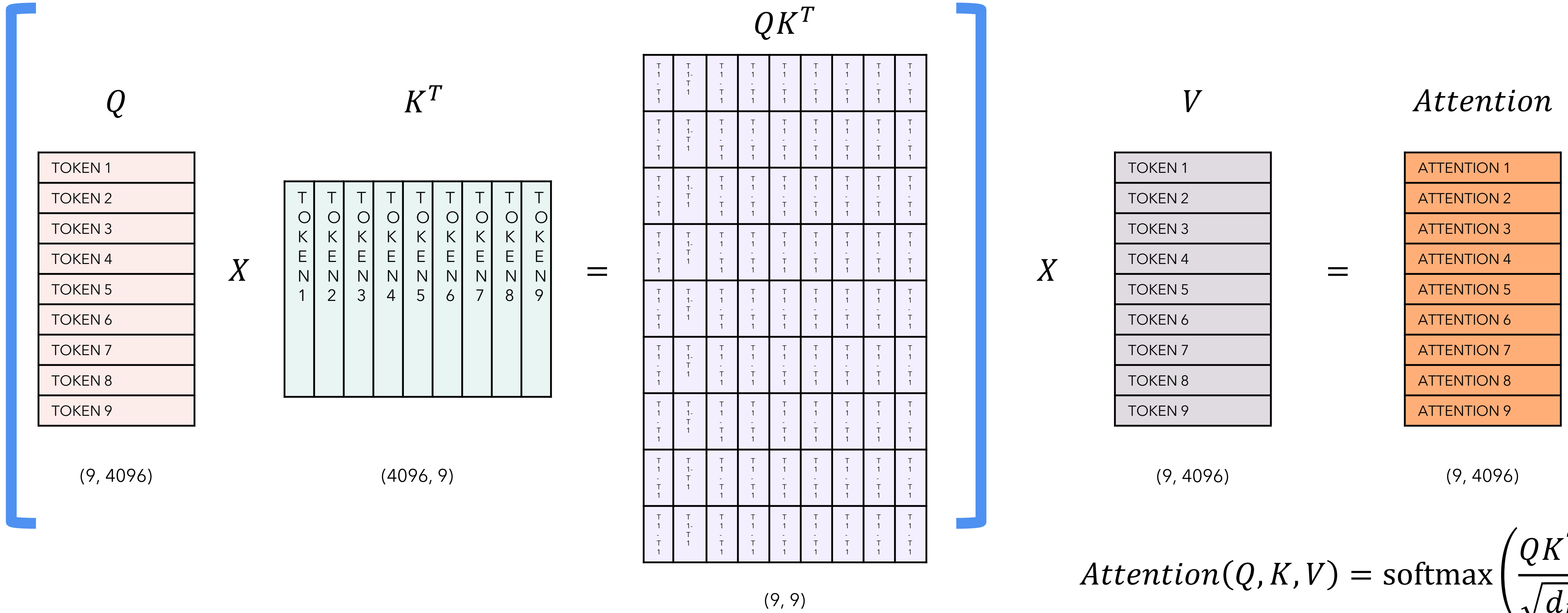
Softmax
Linear
RMS Norm
Feed Forward
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input

Input [SOS] Love that can quickly seize the gentle heart

Next Token Prediction Task: the motivation behind the KV cache

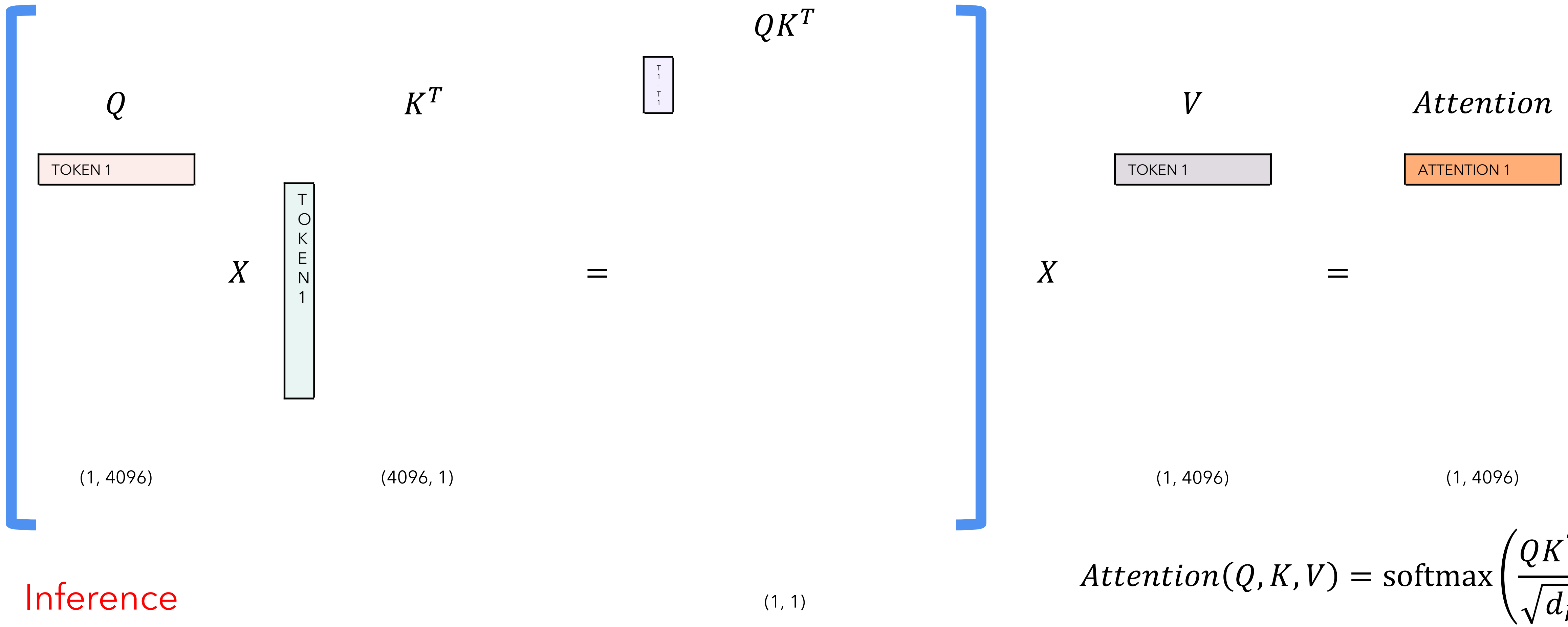
- At every step of the inference, we are only interested in the **last token** output by the model, because we already have the previous ones. However, the model needs access to all the previous tokens to decide on which token to output, since they constitute its context (or the "prompt").
- Is there a way to make the model do less computation on the token it has already seen **during inference**?
  YES! The solution is the **KV cache**!

Self-Attention during Next Token Prediction Task



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

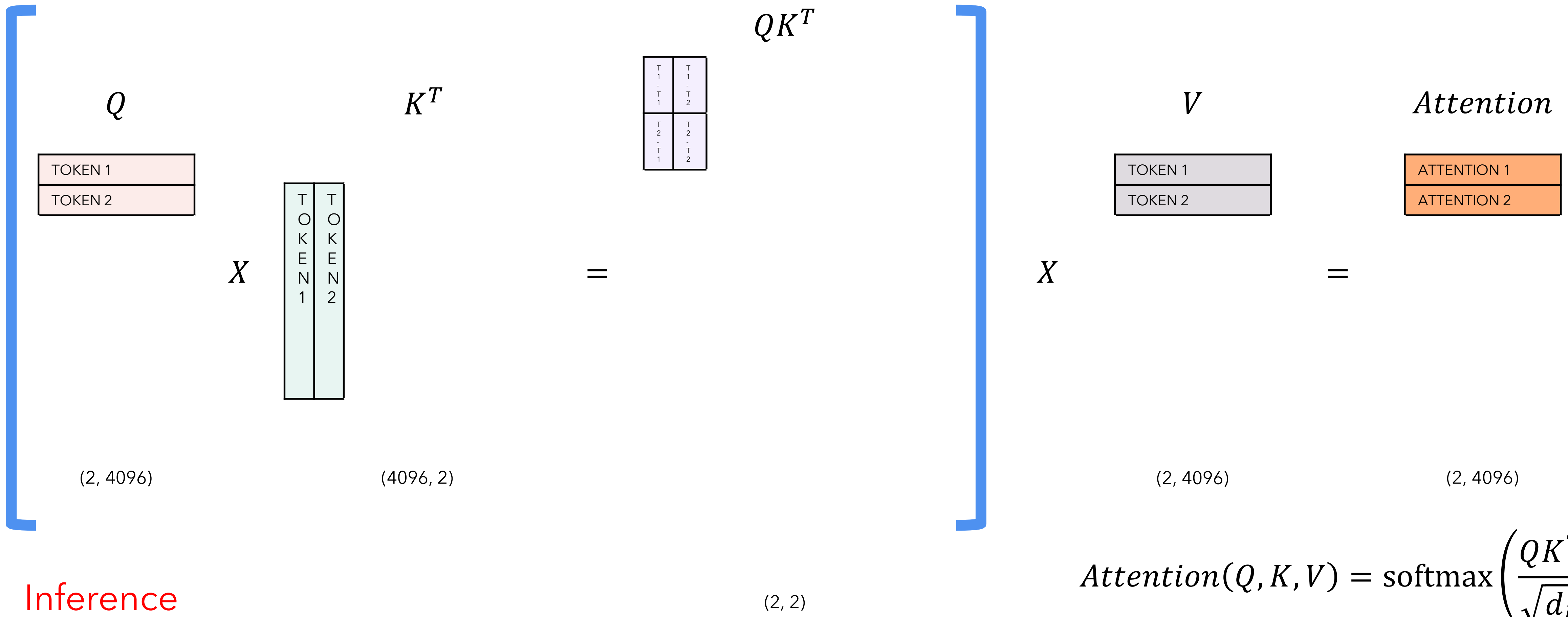
Self-Attention during Next Token Prediction Task



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

Inference
T = 1

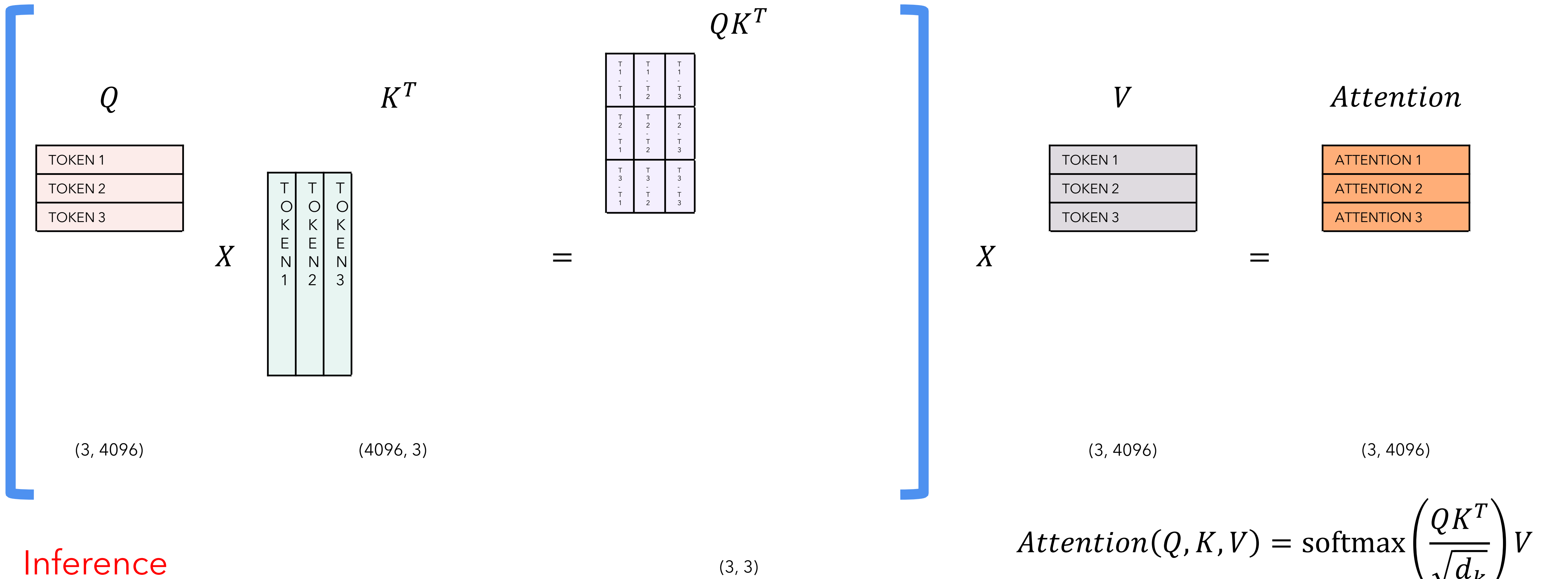
Self-Attention during Next Token Prediction Task



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

Inference
T = 2

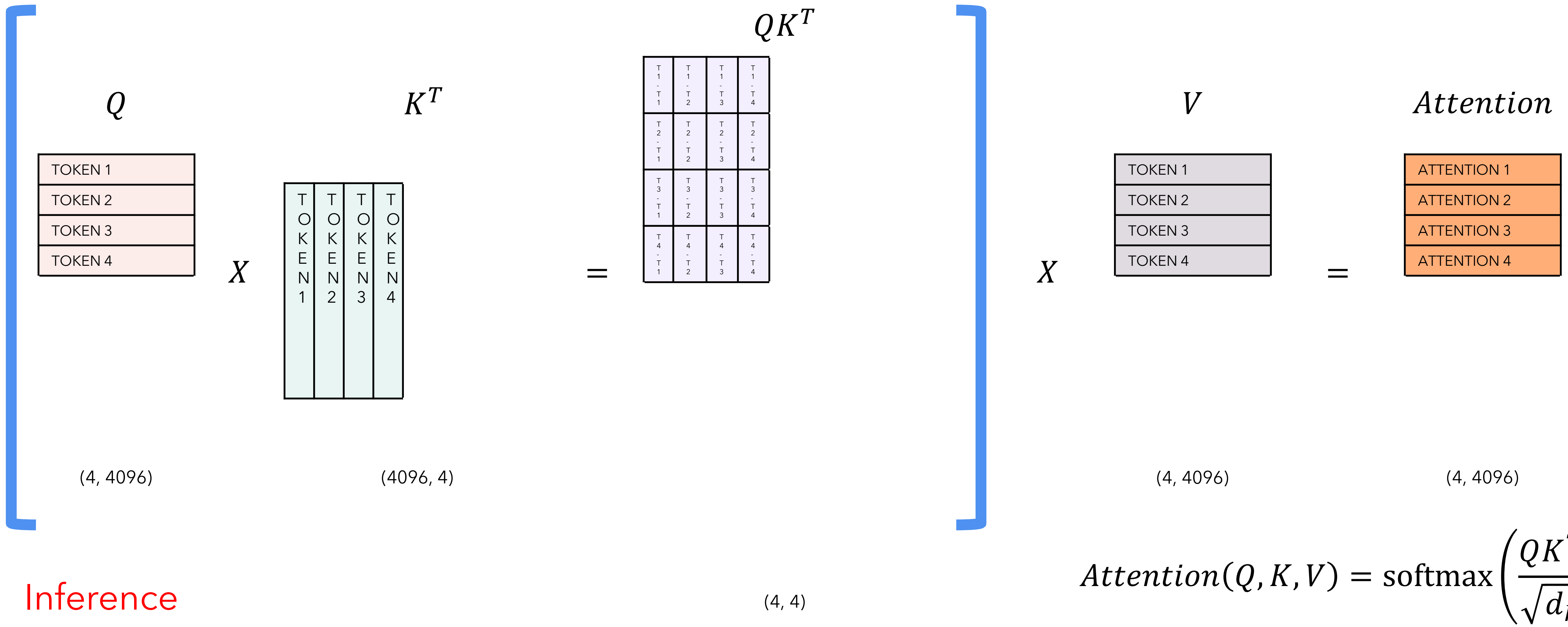
Self-Attention during Next Token Prediction Task



Inference
T = 3

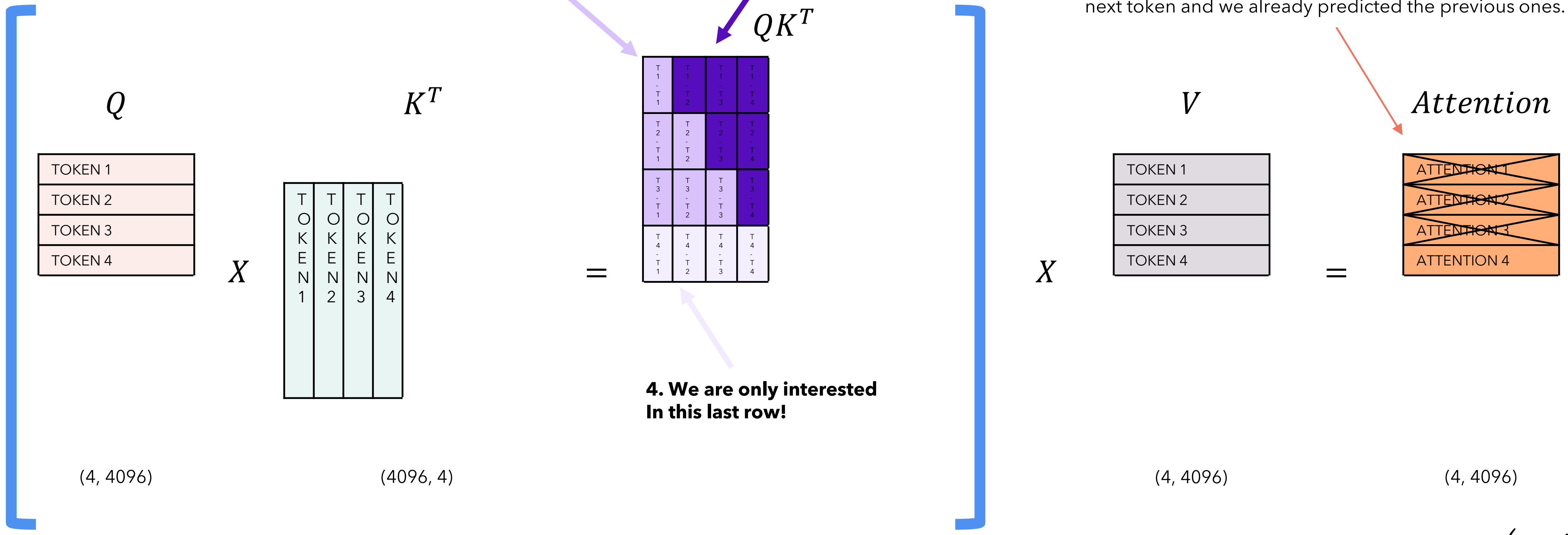
$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Self-Attention during Next Token Prediction Task



Inference
T = 4

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



1. We already computed these dot products In the previous steps. **Can we cache them**?

2. Since the model is causal, **we don't care about the attention of a token with its successors**, but only with the tokens before it.

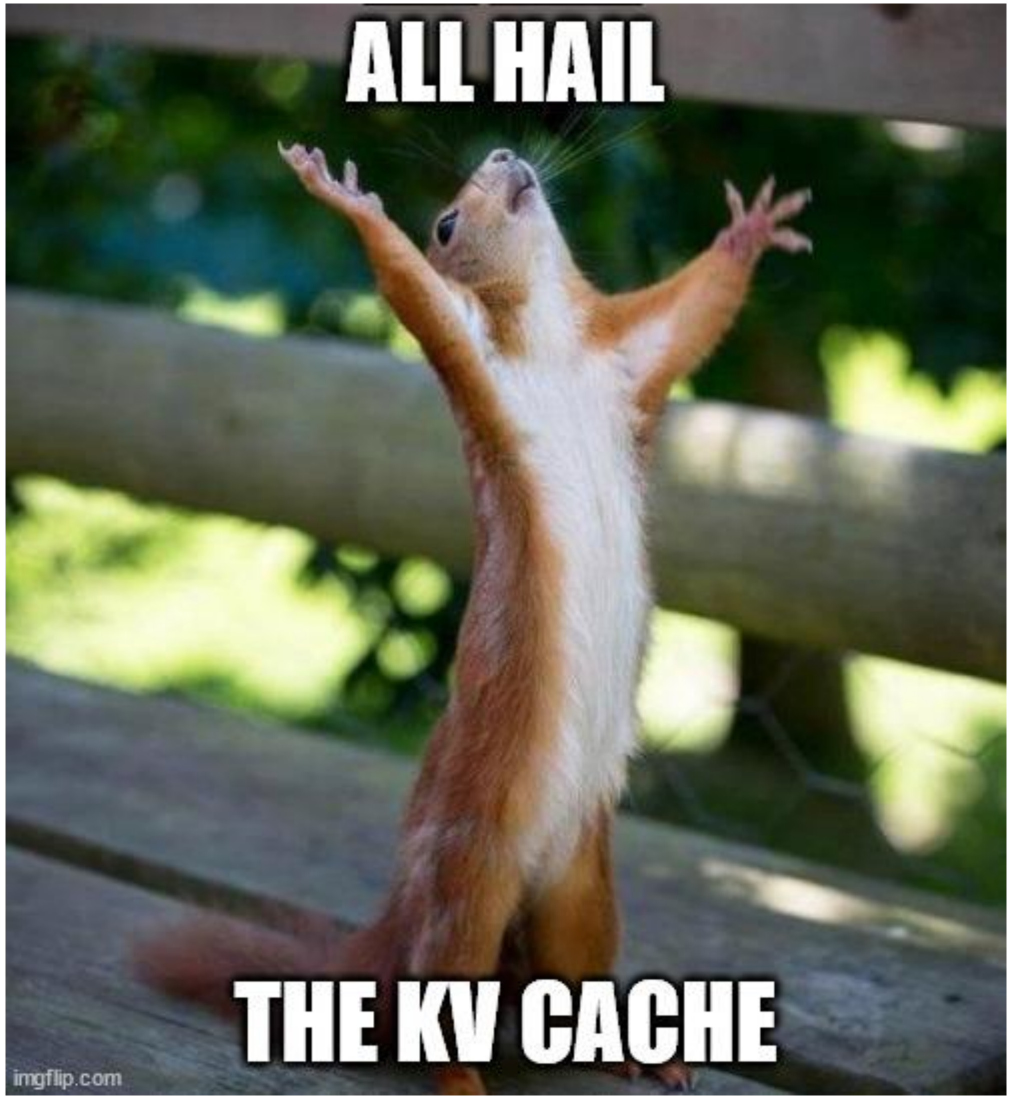
3. We don't care about these, as we want to predict the next token and we already predicted the previous ones.

4. We are only interested In this last row!

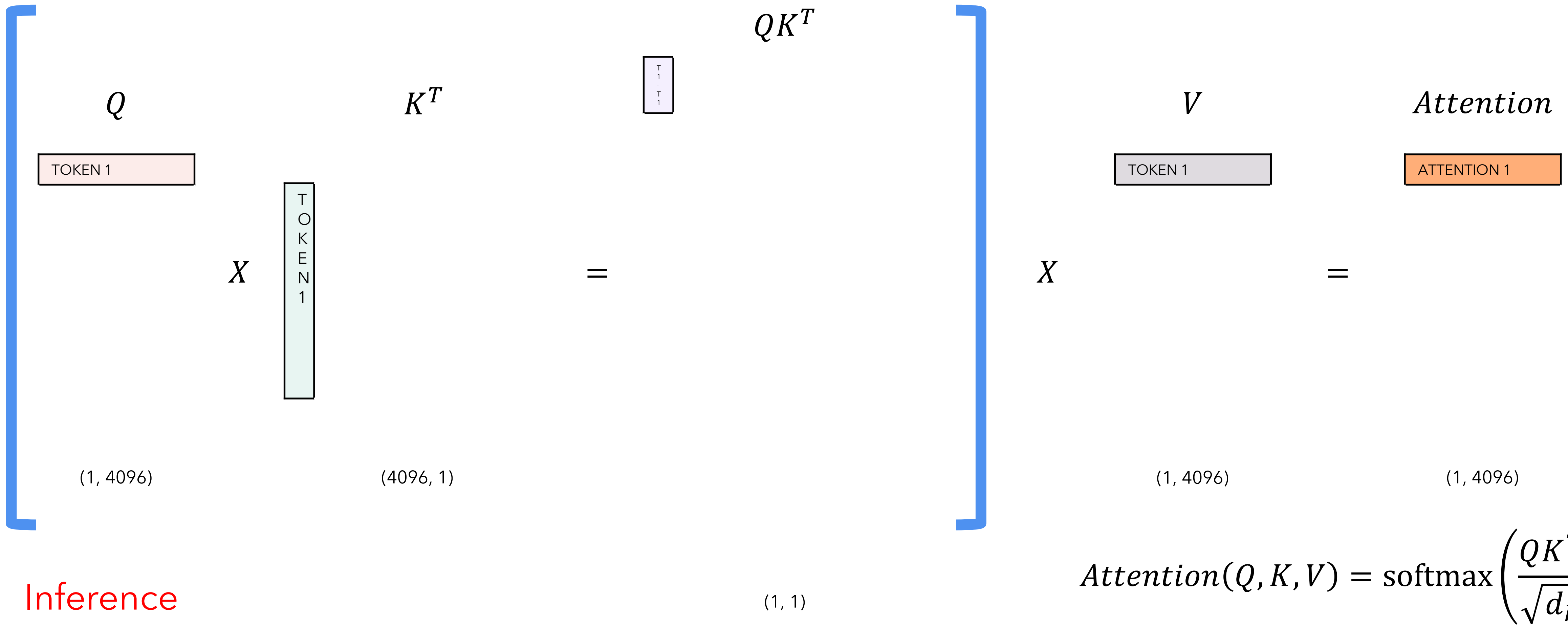
(4, 4)

Inference
T = 4

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


ALL HAIL
THE KV CACHE
imgflip.com

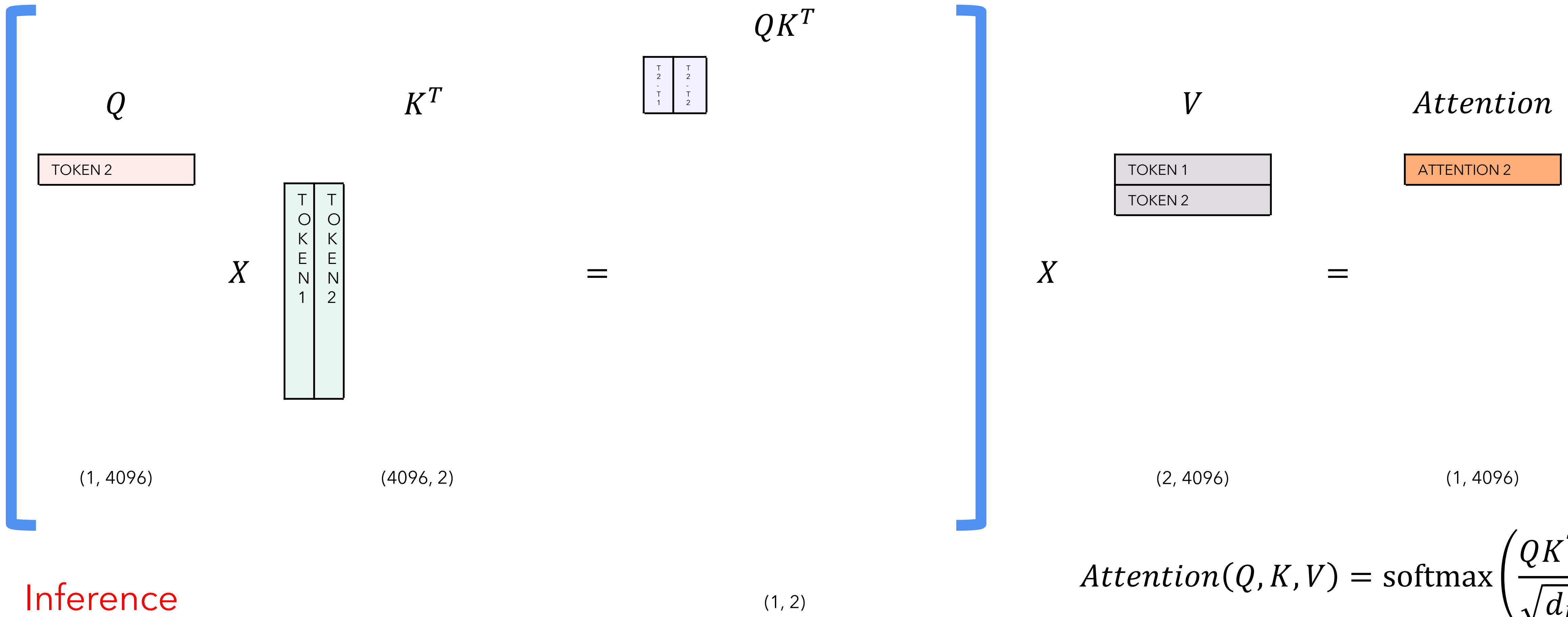
Self-Attention with KV-Cache



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

Inference
T = 1

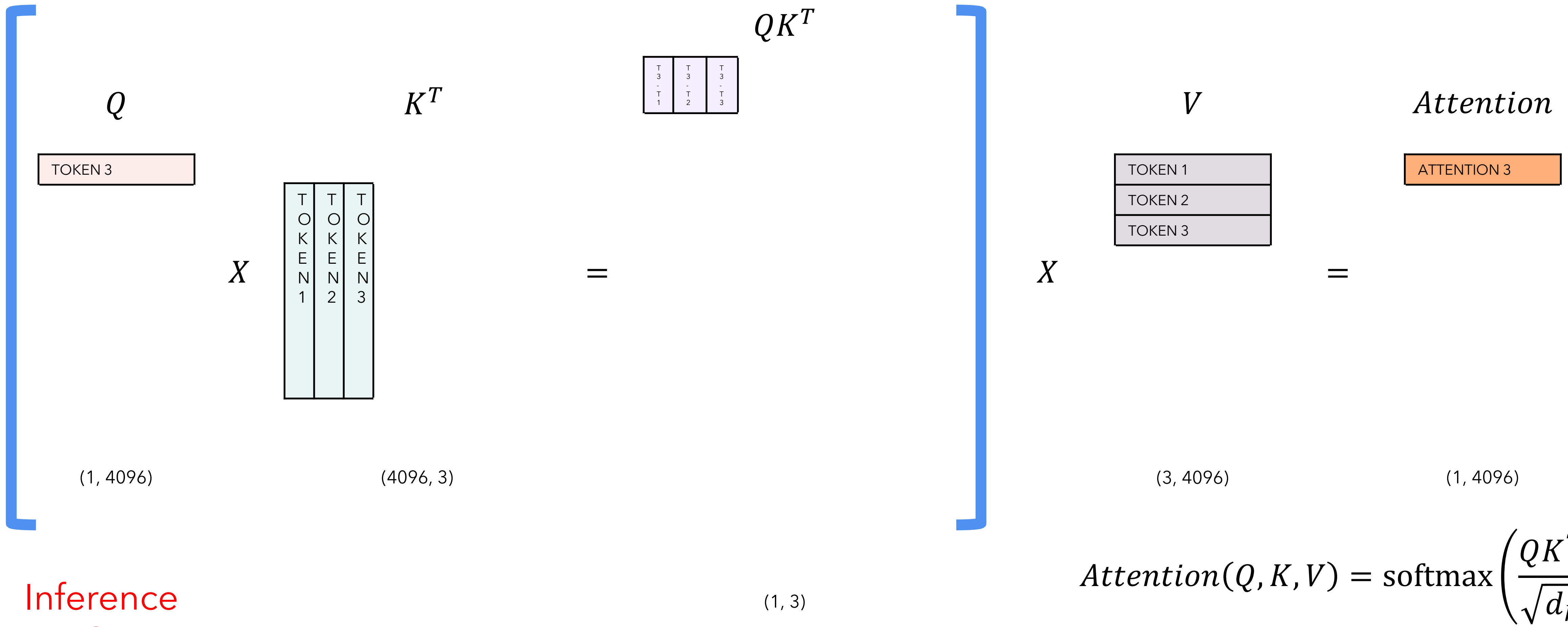
Self-Attention with KV-Cache



Inference
T = 2

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

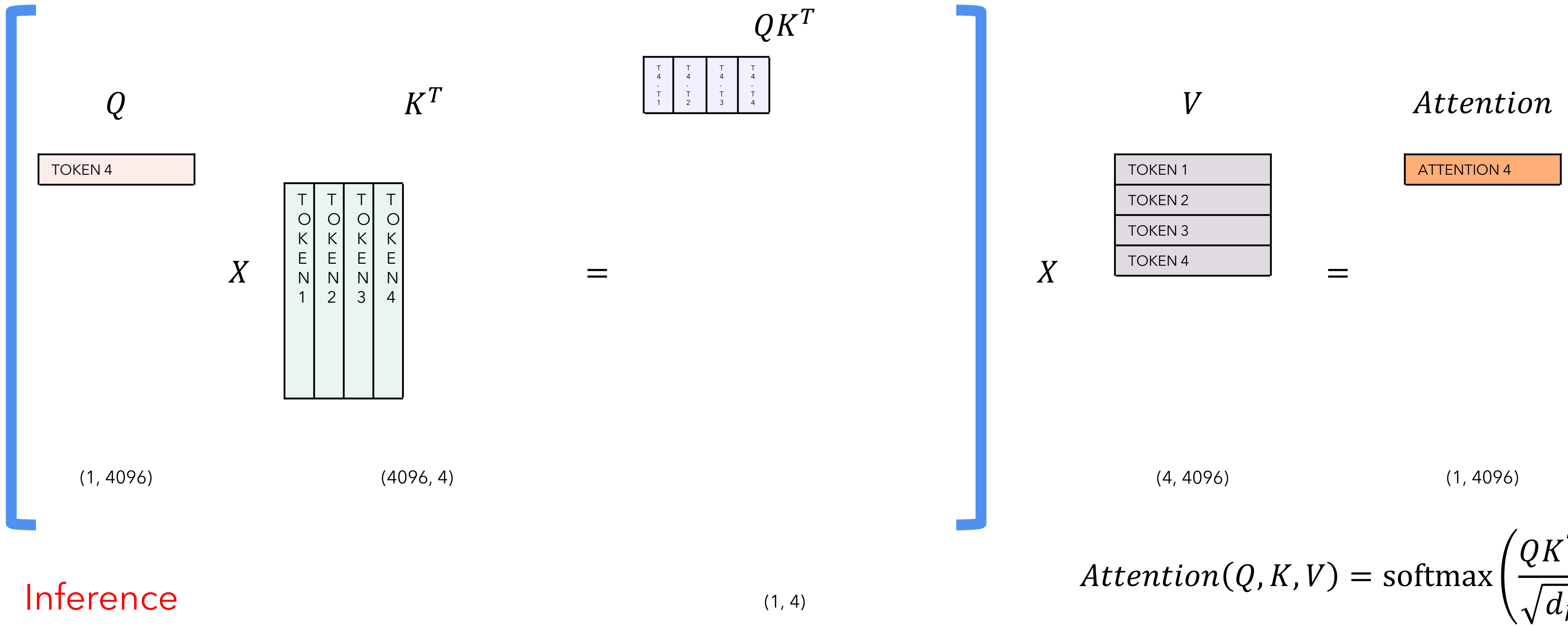
Self-Attention with KV-Cache



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

Inference
T = 3

Self-Attention with KV-Cache



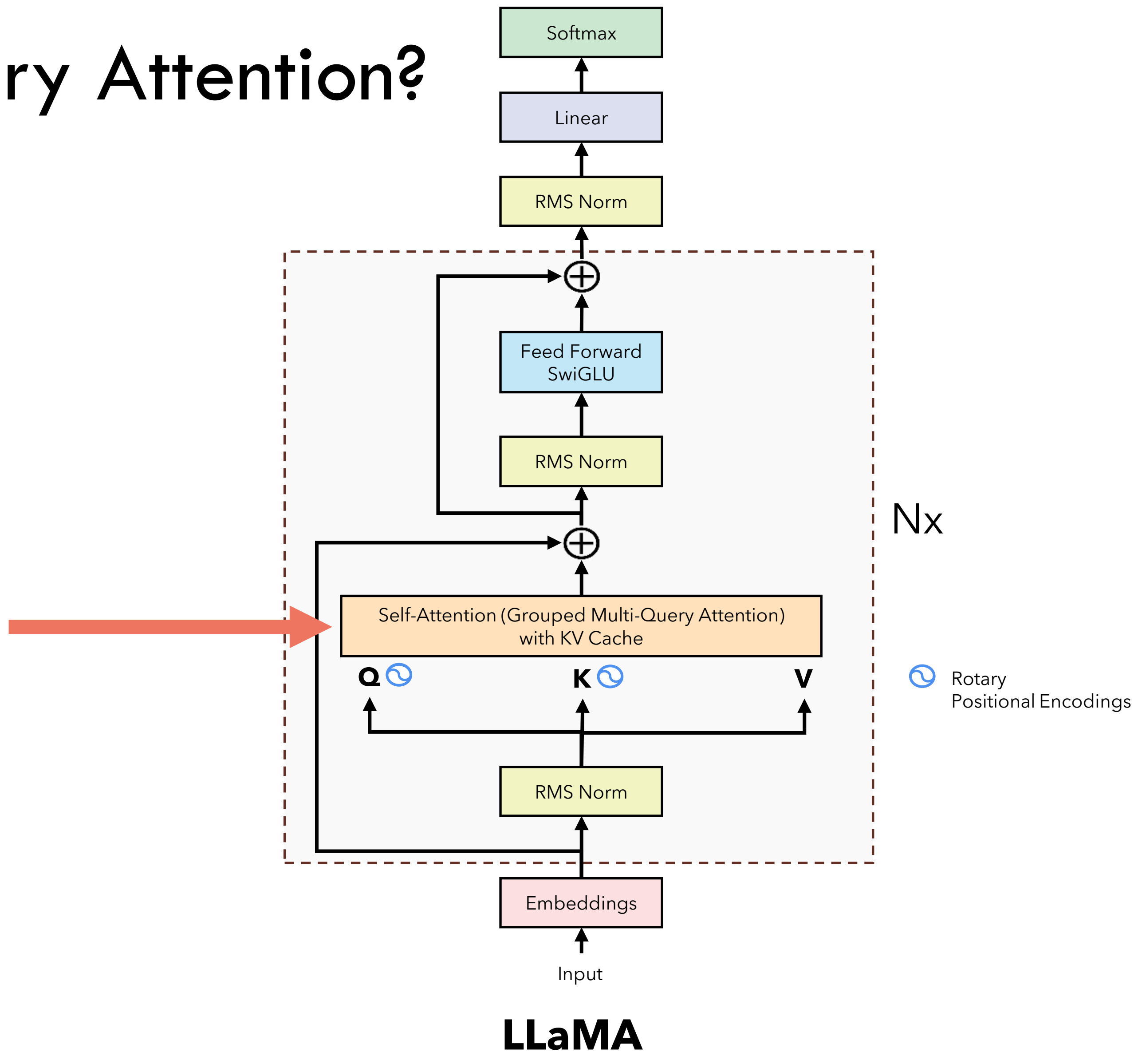
$$Attention(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Inference
T = 4

Umar Jamil – https://github.com/hkproj/pytorch-llama-notes

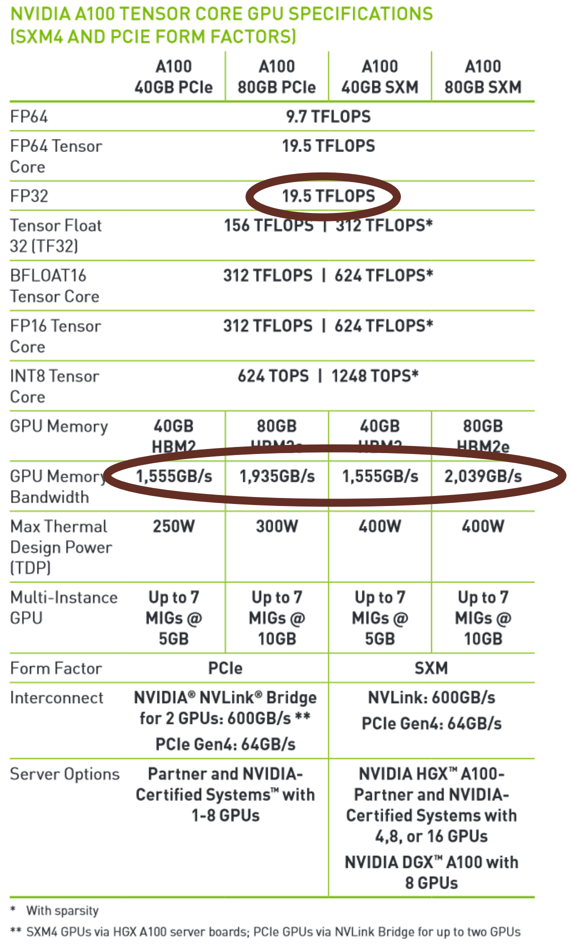
What is Grouped Multi-Query Attention?

Before we talk about Grouped MQA, we need to introduce its predecessor, the Multi-Query Attention (MQA)


Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA

GPUs have a "problem": they're too fast.

- In recent years, GPUs have become very fast at performing calculations, insomuch that the speed of computation (FLOPs) is much higher than the memory bandwidth (GB/s) or speed of data transfer between memory areas. For example, an NVIDIA A100 can perform 19.5 TFLOPs while having a memory bandwidth of 2TB/s.
- This means that sometimes the bottleneck is not how many operations we perform, but how much data transfer our operations need, and that depends on the size and the quantity of the tensors involved in our calculations.
- For example, computing the same operation on the same tensor N times may be faster than computing the same operation on N different tensors, even if they have the same size, this is because the GPU may need to move the tensors around.
- **This means that our goal should not only be to optimize the number of operations we do, but also minimize the memory access/transfers that we perform.**

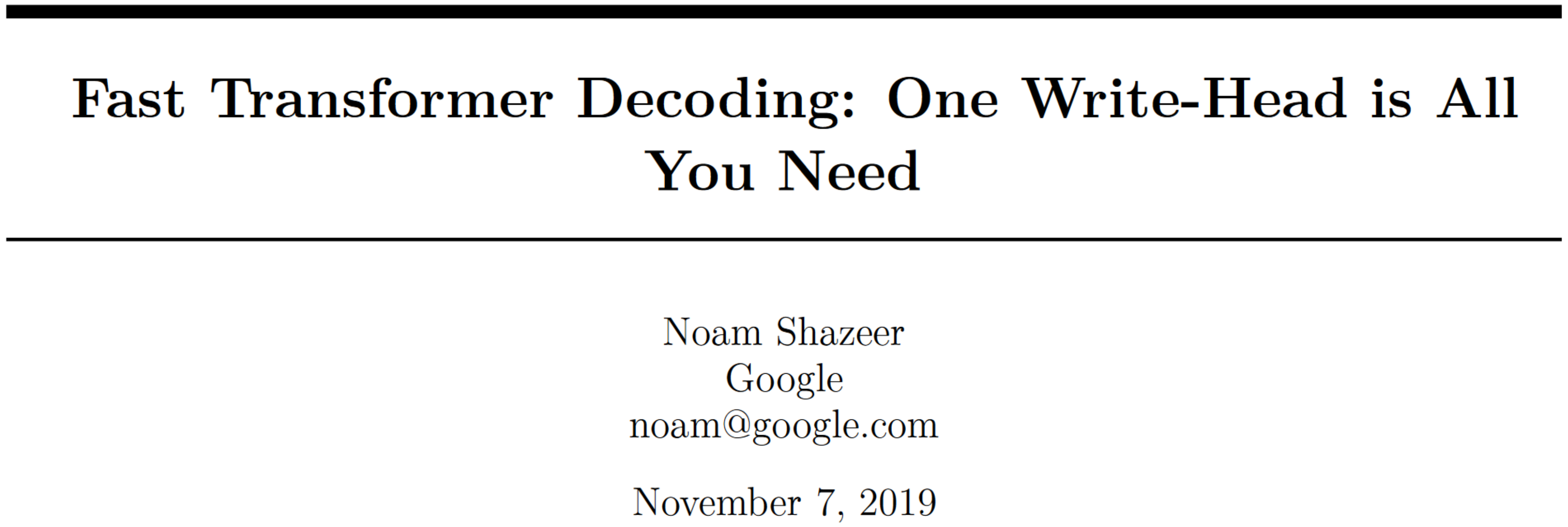


NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

* With sparsity
** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

Introducing Multi-Query Attention



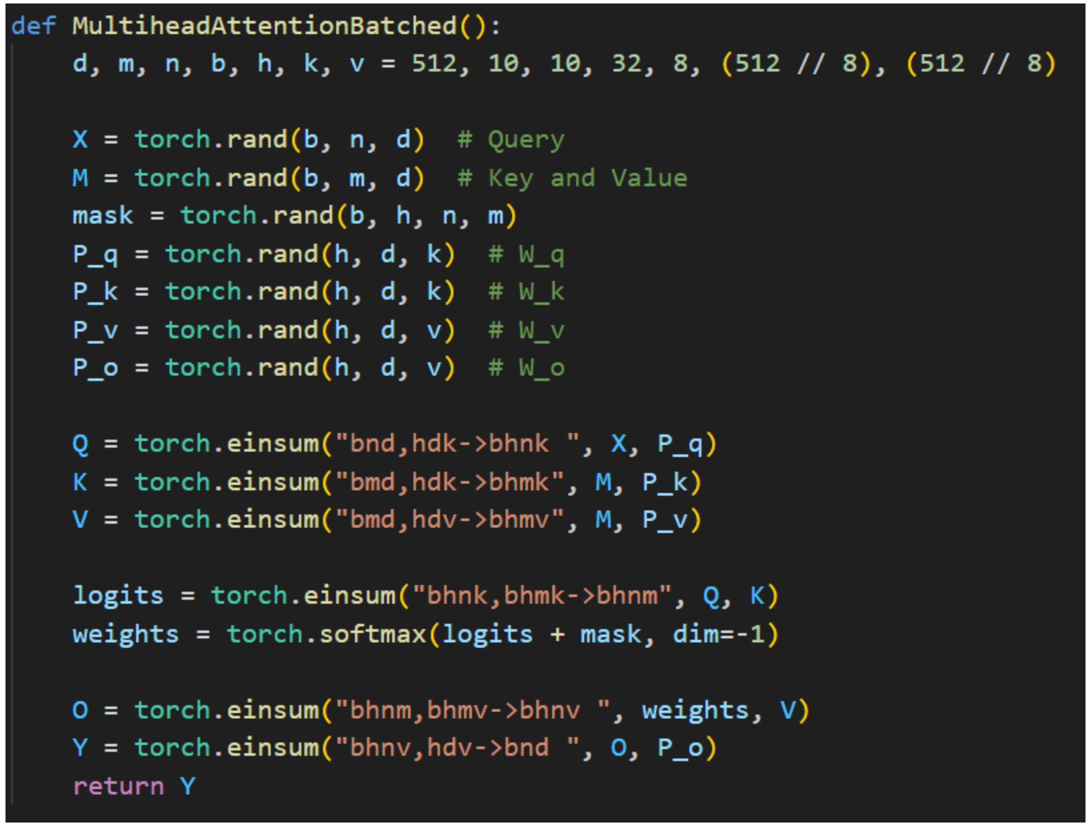
Fast Transformer Decoding: One Write-Head is All You Need

Noam Shazeer
Google
noam@google.com

November 7, 2019

Comparing different attention algorithms: **vanilla batched multi-head attention**

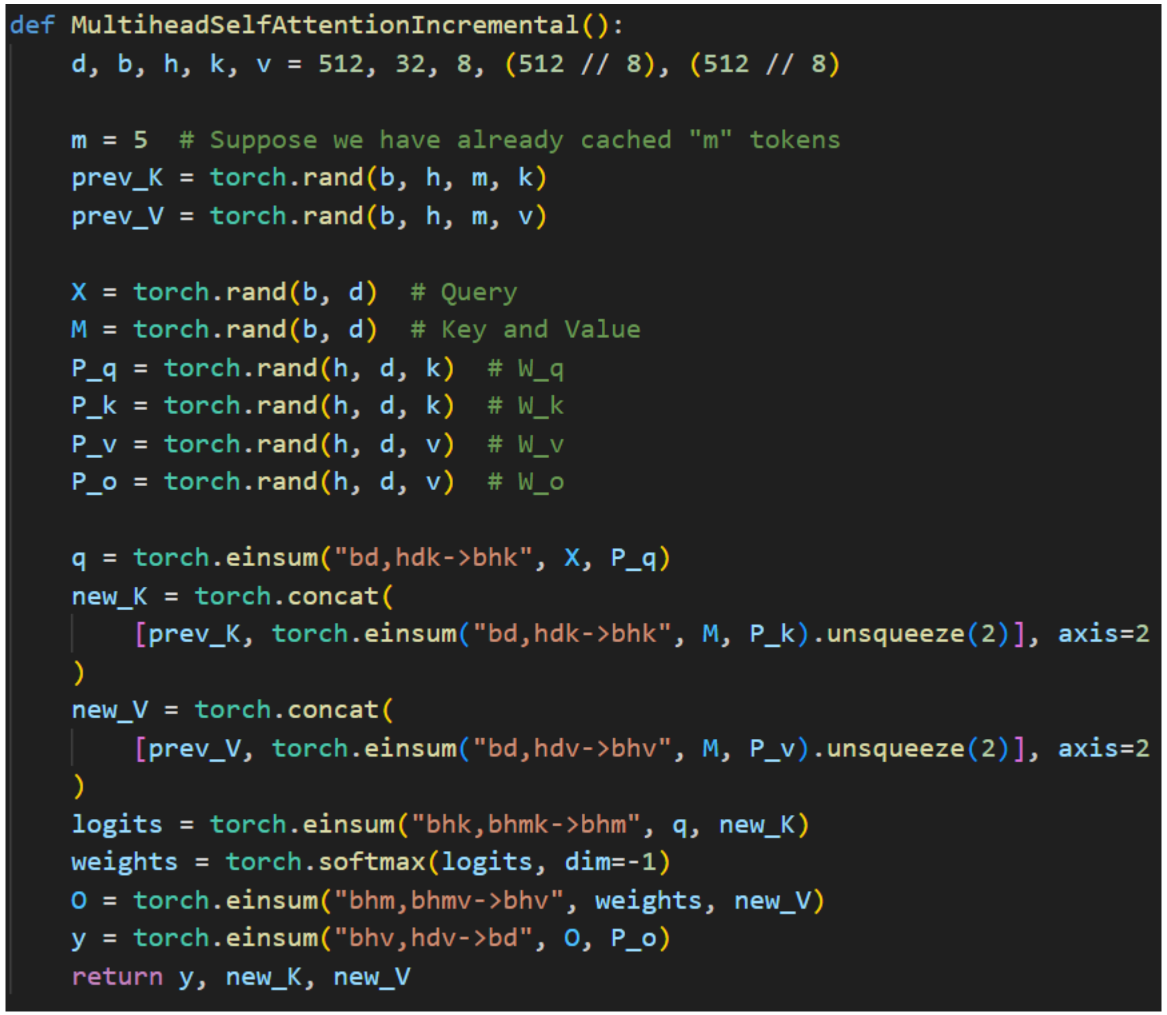
- Multihead Attention as presented in the original paper "Attention is all you need".
- By setting $m = n$ (sequence length of query = seq. length of keys and values)
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O(bnd + bhn^2 + d^2)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{1}{k} + \frac{1}{bn})$
- In this case, the ratio is much smaller than 1, which means that the number of memory access we are performing is much less than the number of arithmetic operations, so the memory access is **not** the bottleneck here.



```python
def MultiheadAttentionBatched():
    d, m, n, b, h, k, v = 512, 10, 10, 32, 8, (512 // 8), (512 // 8)

    X = torch.rand(b, n, d)  # Query
    M = torch.rand(b, m, d)  # Key and Value
    mask = torch.rand(b, h, n, m)
    P_q = torch.rand(h, d, k)  # W_q
    P_k = torch.rand(h, d, k)  # W_k
    P_v = torch.rand(h, d, v)  # W_v
    P_o = torch.rand(h, d, v)  # W_o

    Q = torch.einsum("bnd,hdk->bhnk ", X, P_q)
    K = torch.einsum("bmd,hdk->bhmk", M, P_k)
    V = torch.einsum("bmd,hdv->bhmv", M, P_v)

    logits = torch.einsum("bhnk,bhmk->bhnm", Q, K)
    weights = torch.softmax(logits + mask, dim=-1)

    O = torch.einsum("bhnm,bhmv->bhnv ", weights, V)
    Y = torch.einsum("bhnv,hdv->bnd ", O, P_o)
    return Y
```

Comparing different attention algorithms: **batched multi-head attention with KV cache**

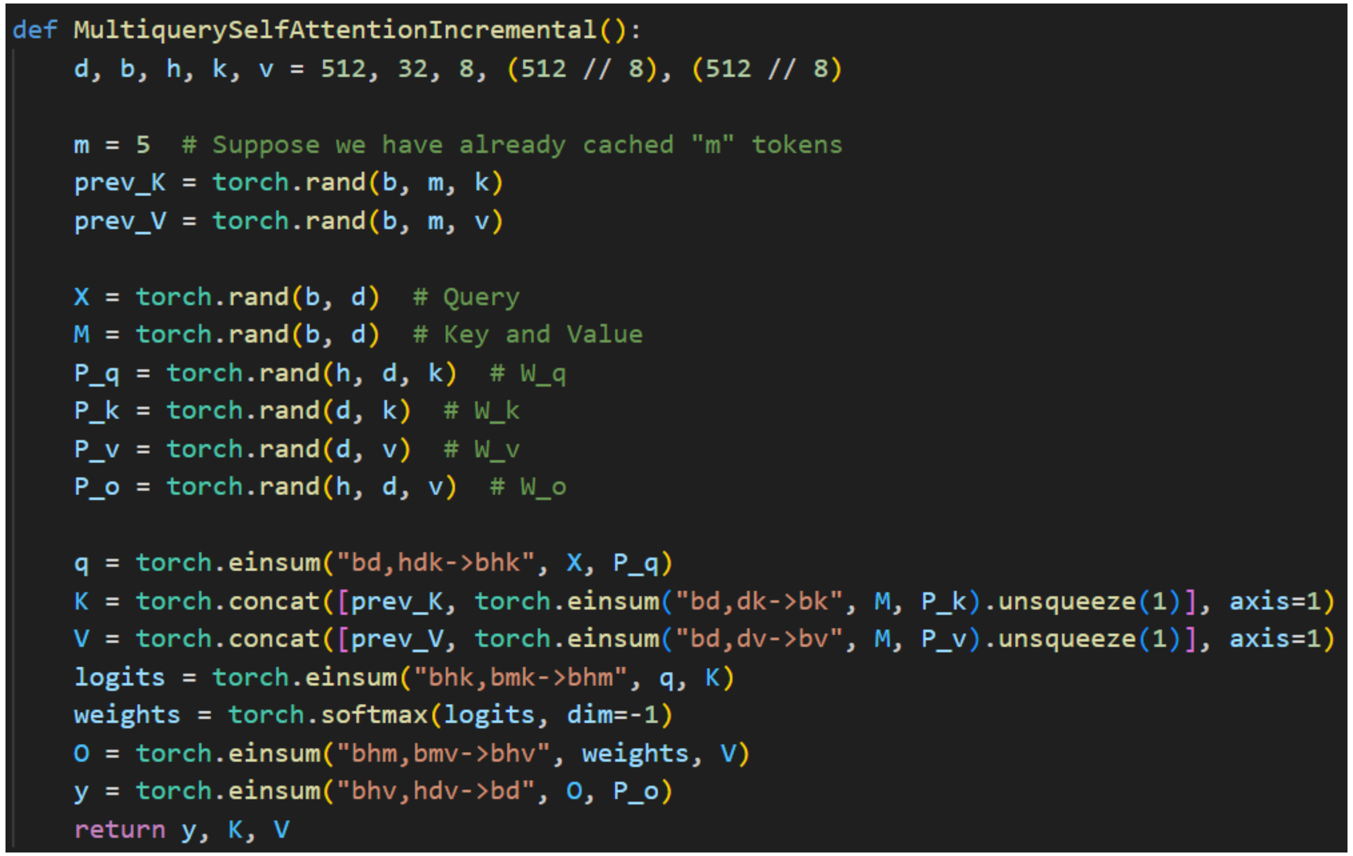
- Uses the KV cache to reduce the number of operations performed.
- By setting $m = n$ (sequence length of query = seq. length of keys and values)
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O(bn^2d + nd^2)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{n}{d} + \frac{1}{b})$
- When $n \approx d$ (the sequence length is close to the size of the embedding vector) or $b \approx 1$ (the batch size is 1), the ratio becomes 1 and the memory access now becomes the bottleneck of the algorithm. For the batch size is not a problem, since it is generally much higher than 1, while for the $\frac{n}{d}$ term, we need to reduce the sequence length. **But there's a better way…**



```python
def MultiheadSelfAttentionIncremental():
    d, b, h, k, v = 512, 32, 8, (512 // 8), (512 // 8)

    m = 5  # Suppose we have already cached "m" tokens
    prev_K = torch.rand(b, h, m, k)
    prev_V = torch.rand(b, h, m, v)

    X = torch.rand(b, d)  # Query
    M = torch.rand(b, d)  # Key and Value
    P_q = torch.rand(h, d, k)  # W_q
    P_k = torch.rand(h, d, k)  # W_k
    P_v = torch.rand(h, d, v)  # W_v
    P_o = torch.rand(h, d, v)  # W_o

    q = torch.einsum("bd,hdk->bhk", X, P_q)
    new_K = torch.concat(
        [prev_K, torch.einsum("bd,hdk->bhk", M, P_k).unsqueeze(2)], axis=2
    )
    new_V = torch.concat(
        [prev_V, torch.einsum("bd,hdv->bhv", M, P_v).unsqueeze(2)], axis=2
    )
    logits = torch.einsum("bhk,bhmk->bhm", q, new_K)
    weights = torch.softmax(logits, dim=-1)
    O = torch.einsum("bhm,bhmv->bhv", weights, new_V)
    y = torch.einsum("bhv,hdv->bd", O, P_o)
    return y, new_K, new_V
```

Comparing different attention algorithms: **multi-query attention with KV cache**

- We remove the h dimension from the K and the V, while keeping it for the Q. This means that all the different query heads will share the same keys and values.
- The number of arithmetic operations performed is $O(bnd^2)$
- The total memory involved in the operations, given by the sum of all the tensors involved in the calculations (including the derived ones!) is $O(bnd + bn^2k + nd^2)$
- The ratio between the total memory and the number of arithmetic operations is $O(\frac{1}{d} + \frac{n}{dh} + \frac{1}{b})$
- Comparing with the previous approach, we have reduced the expensive term $\frac{n}{d}$ by a factor of h.
- The performance gains are important, while the model's quality degrades only a little bit.



```python
def MultiquerySelfAttentionIncremental():
    d, b, h, k, v = 512, 32, 8, (512 // 8), (512 // 8)

    m = 5  # Suppose we have already cached "m" tokens
    prev_K = torch.rand(b, m, k)
    prev_V = torch.rand(b, m, v)

    X = torch.rand(b, d)  # Query
    M = torch.rand(b, d)  # Key and Value
    P_q = torch.rand(h, d, k)  # W_q
    P_k = torch.rand(d, k)  # W_k
    P_v = torch.rand(d, v)  # W_v
    P_o = torch.rand(h, d, v)  # W_o

    q = torch.einsum("bd,hdk->bhk", X, P_q)
    K = torch.concat([prev_K, torch.einsum("bd,dk->bk", M, P_k).unsqueeze(1)], axis=1)
    V = torch.concat([prev_V, torch.einsum("bd,dv->bv", M, P_v).unsqueeze(1)], axis=1)
    logits = torch.einsum("bhk,bmk->bhm", q, K)
    weights = torch.softmax(logits, dim=-1)
    O = torch.einsum("bhm,bmv->bhv", weights, V)
    y = torch.einsum("bhv,hdv->bd", O, P_o)
    return y, K, V
```

Speed & Quality comparisons

BLEU score on a translation task (English – German)

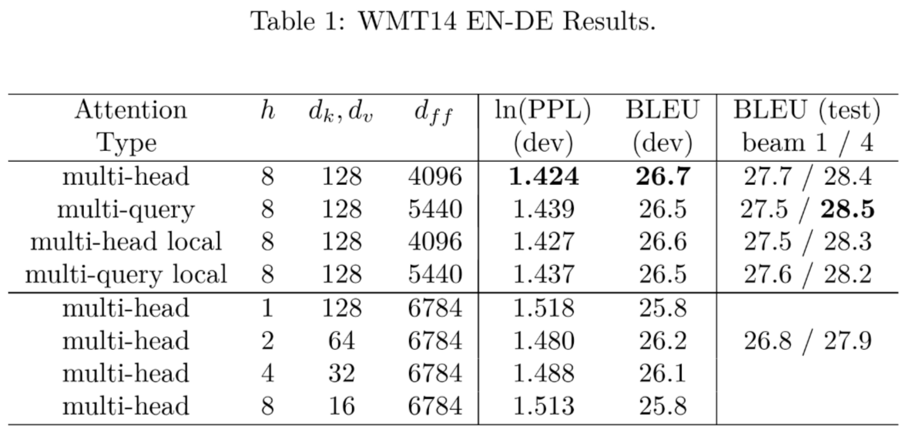


Table 1: WMT14 EN-DE Results.

Attention Type	h	d_k, d_v	d_{ff}	ln(PPL) (dev)	BLEU (dev)	BLEU (test) beam 1 / 4
multi-head	8	128	4096	**1.424**	**26.7**	27.7 / 28.4
multi-query	8	128	5440	1.439	26.5	27.5 / **28.5**
multi-head local	8	128	4096	1.427	26.6	27.5 / 28.3
multi-query local	8	128	5440	1.437	26.5	27.6 / 28.2
multi-head	1	128	6784	1.518	25.8	
multi-head	2	64	6784	1.480	26.2	26.8 / 27.9
multi-head	4	32	6784	1.488	26.1	
multi-head	8	16	6784	1.513	25.8	

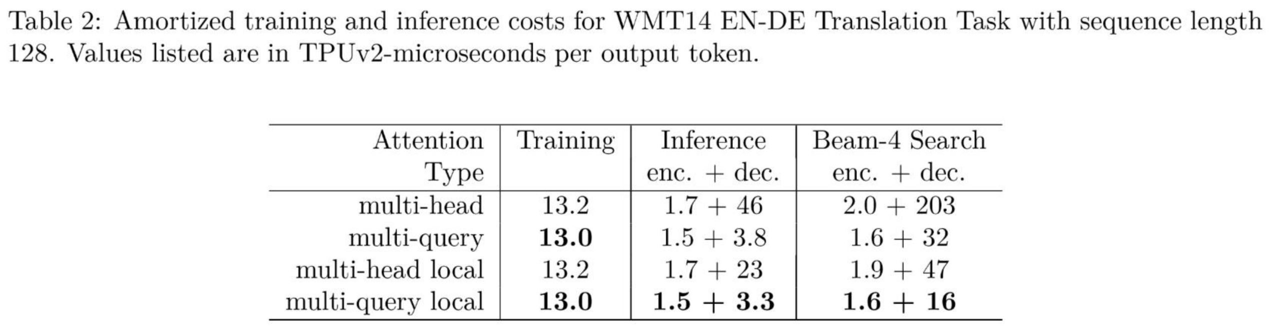
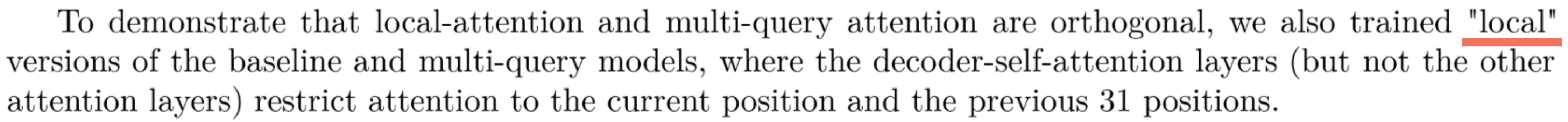


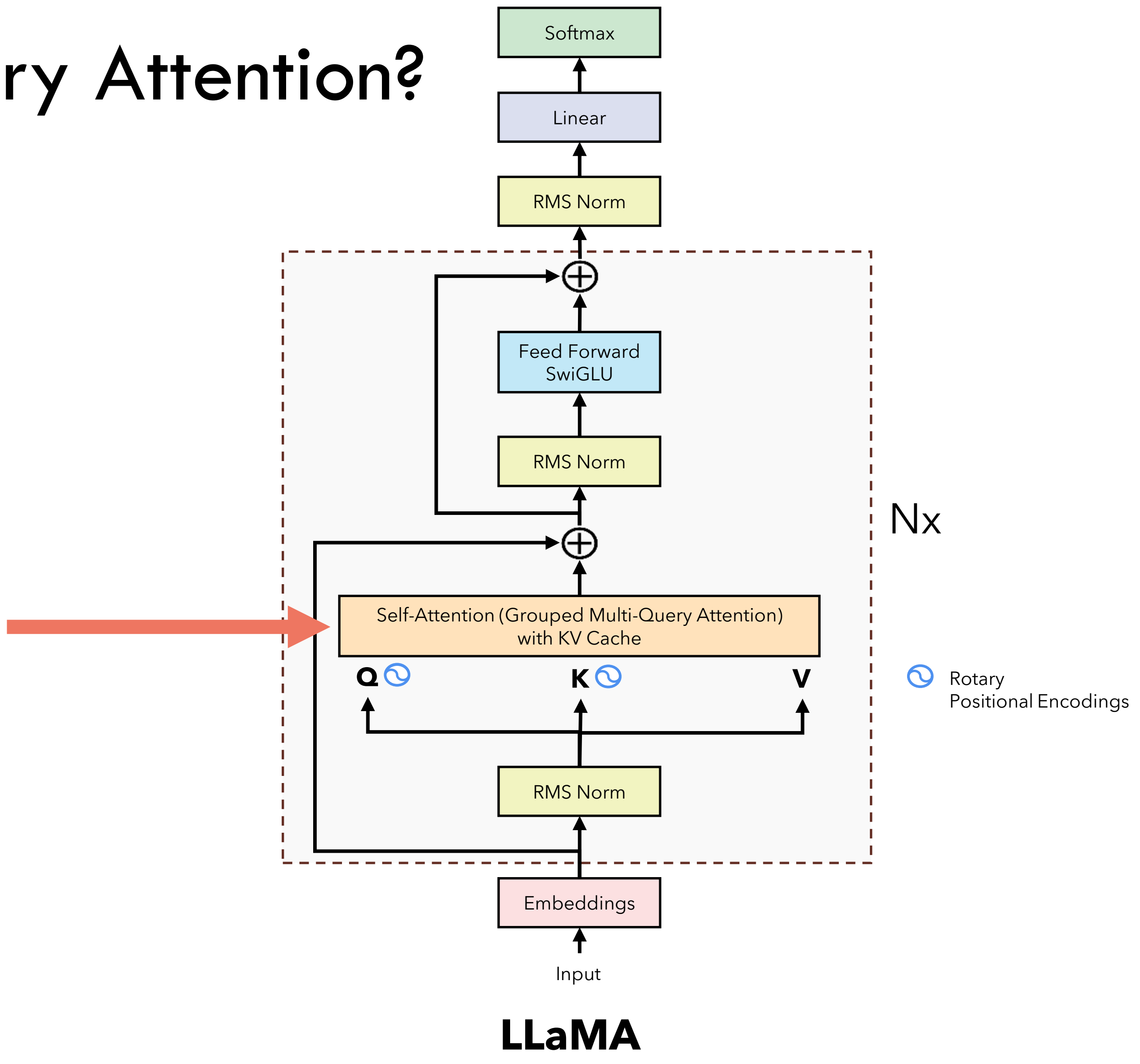
Table 2: Amortized training and inference costs for WMT14 EN-DE Translation Task with sequence length 128. Values listed are in TPUv2-microseconds per output token.

Attention Type	Training	Inference enc. + dec.	Beam-4 Search enc. + dec.
multi-head	13.2	1.7 + 46	2.0 + 203
multi-query	**13.0**	1.5 + 3.8	1.6 + 32
multi-head local	13.2	1.7 + 23	1.9 + 47
multi-query local	**13.0**	**1.5 + 3.3**	**1.6 + 16**


To demonstrate that local-attention and multi-query attention are orthogonal, we also trained "local" versions of the baseline and multi-query models, where the decoder-self-attention layers (but not the other attention layers) restrict attention to the current position and the previous 31 positions.

What is Grouped Multi-Query Attention?

Now, let's talk about Grouped MQA!


Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA

Grouped Multi-Query Attention: a compromise between two extremes.

Multi-Head Attention

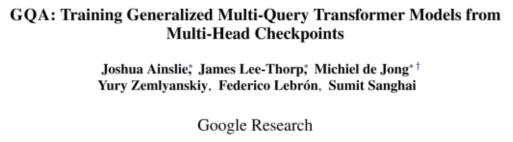
- High quality
- Computationally slow

Grouped Multi-Query Attention

- A good compromise between quality and speed

Multi-Query Attention

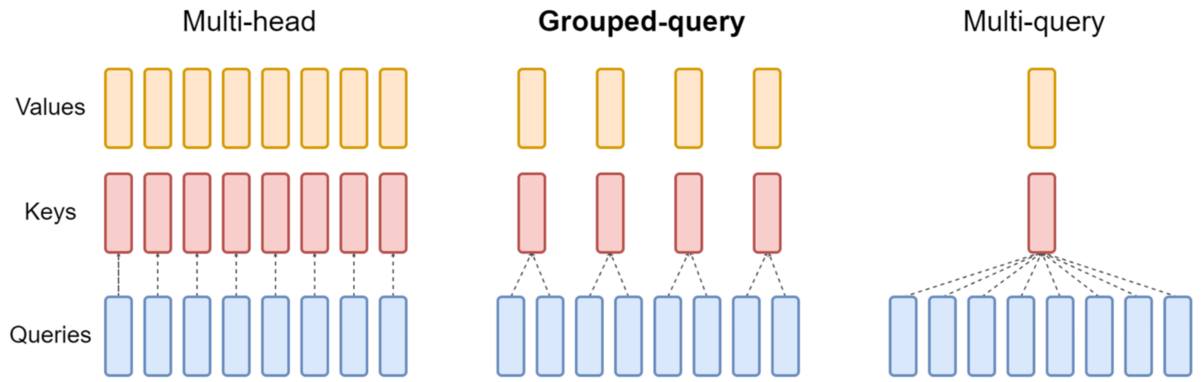
- Loss in quality
- Computationally fast



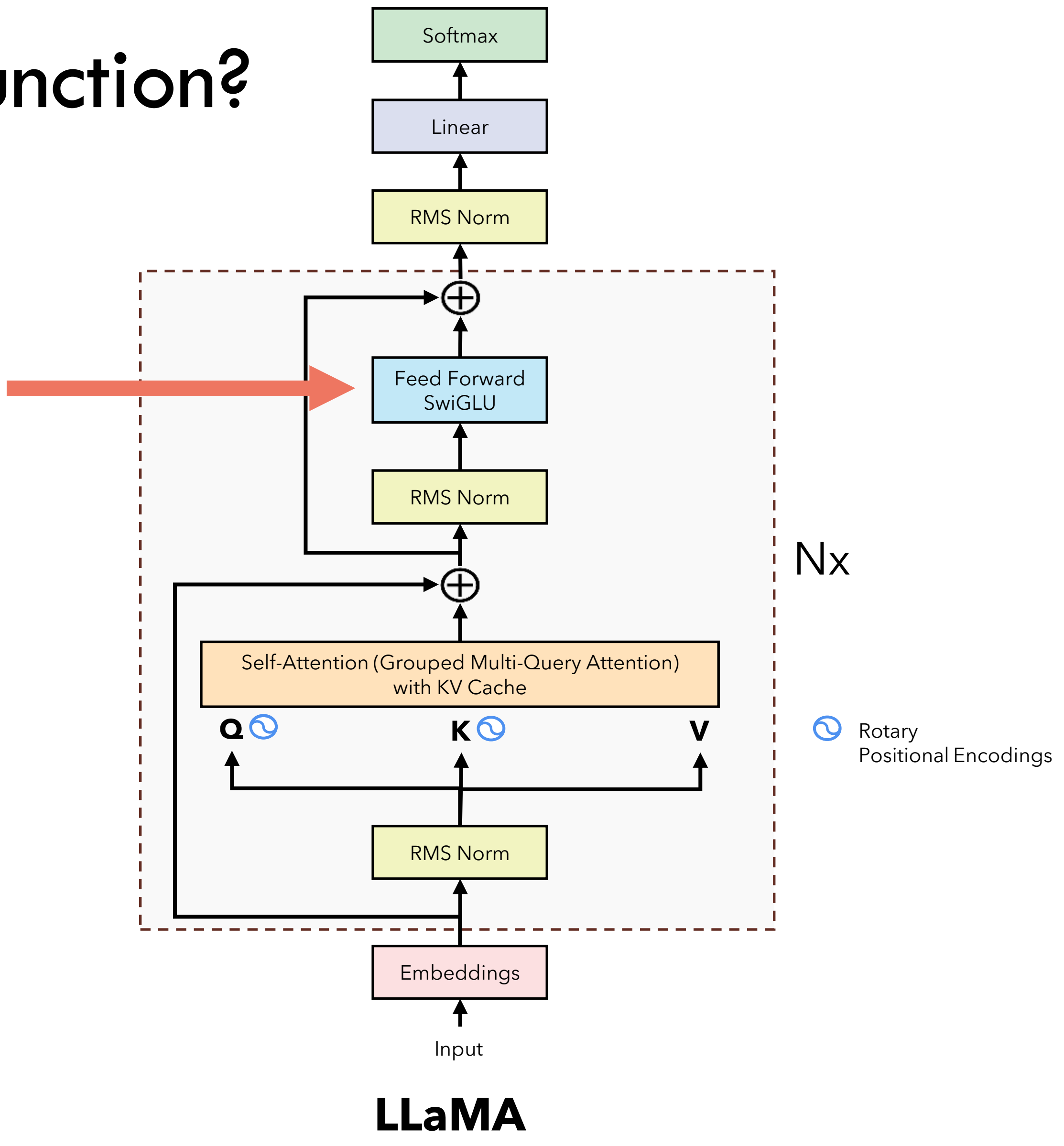
GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints

Joshua Ainslie*, James Lee-Thorp*, Michiel de Jong* †
Yury Zemlyanskiy, Federico Lebrón, Sumit Sanghai

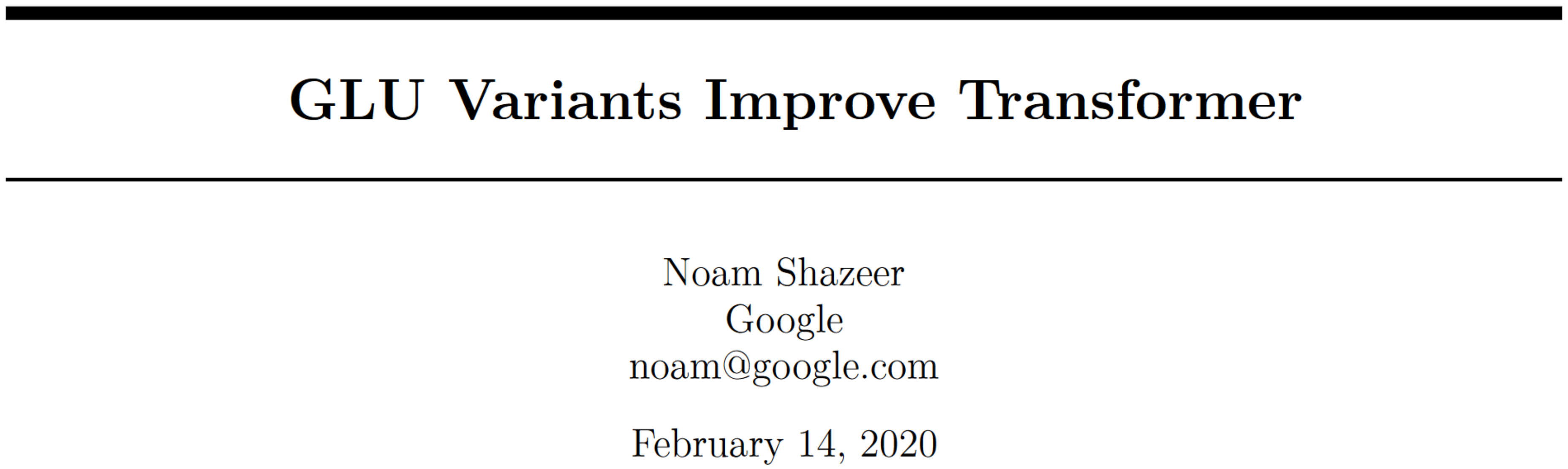
Google Research



What is the SwiGLU activation function?


Softmax
Linear
RMS Norm
Feed Forward SwiGLU
RMS Norm
Nx
Self-Attention (Grouped Multi-Query Attention) with KV Cache
Q
K
V
Rotary Positional Encodings
RMS Norm
Embeddings
Input
LLaMA

SwiGLU Activation Function



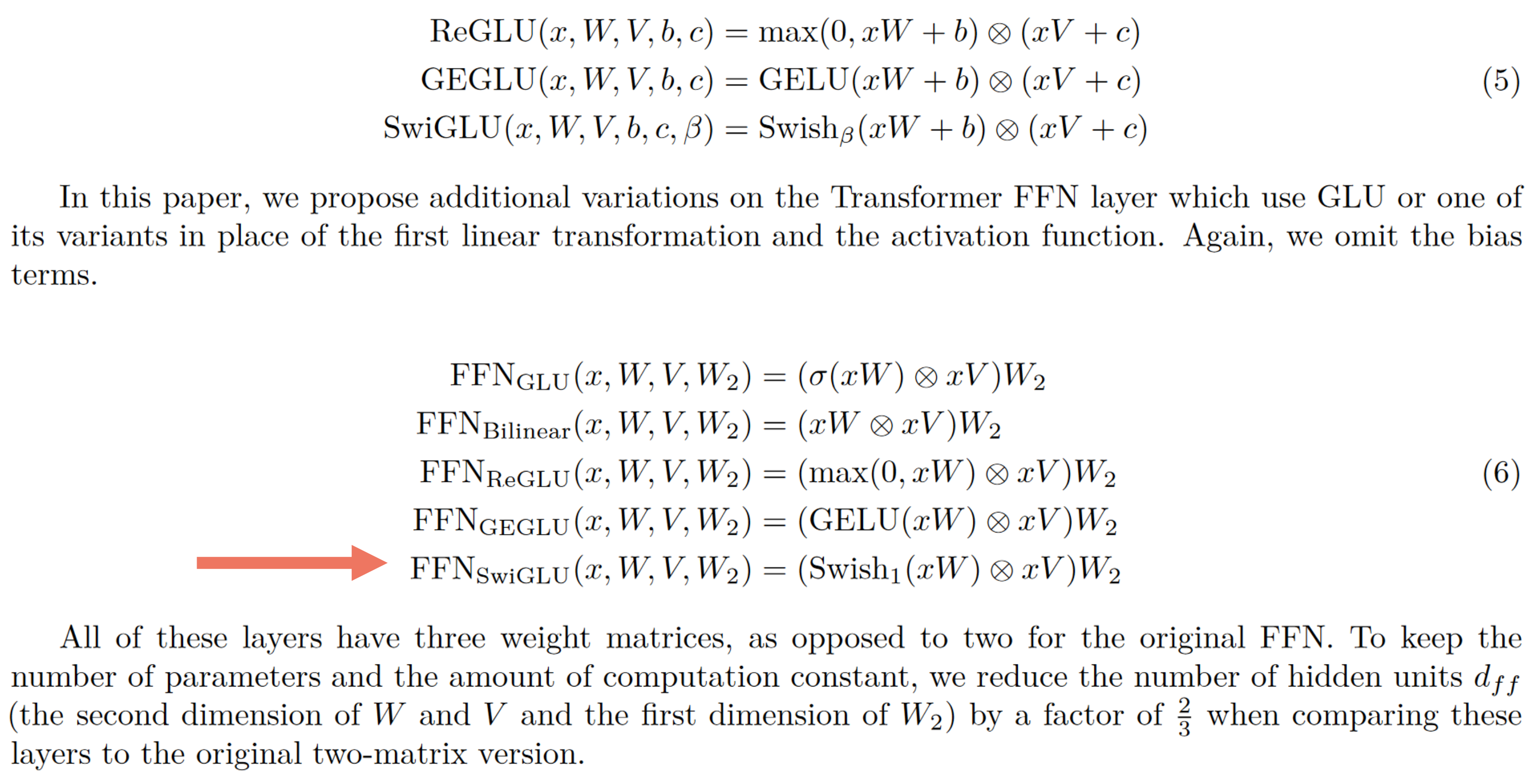
GLU Variants Improve Transformer

Noam Shazeer
Google
noam@google.com

February 14, 2020

SwiGLU Activation Function

- The author compared the performance of a Transformer model by using different activation functions in the Feed-Forward layer of the Transformer architecture.



$$
\begin{aligned}
\mathrm{ReGLU}(x, W, V, b, c) &= \max(0, xW + b) \otimes (xV + c) \\
\mathrm{GEGLU}(x, W, V, b, c) &= \mathrm{GELU}(xW + b) \otimes (xV + c) \\
\mathrm{SwiGLU}(x, W, V, b, c, \beta) &= \mathrm{Swish}_\beta(xW + b) \otimes (xV + c)
\end{aligned} \tag{5}
$$

In this paper, we propose additional variations on the Transformer FFN layer which use GLU or one of its variants in place of the first linear transformation and the activation function. Again, we omit the bias terms.

$$
\begin{aligned}
\mathrm{FFN}_{\mathrm{GLU}}(x, W, V, W_2) &= (\sigma(xW) \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{Bilinear}}(x, W, V, W_2) &= (xW \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{ReGLU}}(x, W, V, W_2) &= (\max(0, xW) \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{GEGLU}}(x, W, V, W_2) &= (\mathrm{GELU}(xW) \otimes xV)W_2 \\
\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) &= (\mathrm{Swish}_1(xW) \otimes xV)W_2
\end{aligned} \tag{6}
$$

All of these layers have three weight matrices, as opposed to two for the original FFN. To keep the number of parameters and the amount of computation constant, we reduce the number of hidden units d_{ff} (the second dimension of W and V and the first dimension of W_2) by a factor of $\frac{2}{3}$ when comparing these layers to the original two-matrix version.

SwiGLU Activation Function

Transformer (“Attention is all you need”)

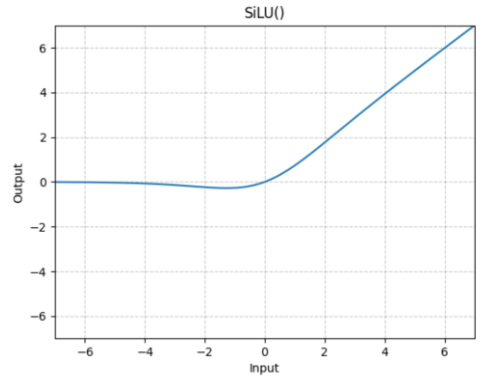
$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

LLaMA

$$\mathrm{FFN}_{\mathrm{SwiGLU}}(x, W, V, W_2) = (\mathrm{Swish}_1(xW) \otimes xV)W_2$$

We use the swish function with β = 1. In this case it’s called the **Sigmoid Linear Unit** (**SiLU**) function.

$$\mathrm{swish}(x) = x\,\mathrm{sigmoid}(\beta x) = \frac{x}{1 + e^{-\beta x}}$$



```
class FeedForward(nn.Module):
    def __init__(
        self,
        dim: int,
        hidden_dim: int,
        multiple_of: int,
        ffn_dim_multiplier: Optional[float],
    ):
        super().__init__()
        hidden_dim = int(2 * hidden_dim / 3)
        # custom dim factor multiplier
        if ffn_dim_multiplier is not None:
            hidden_dim = int(ffn_dim_multiplier * hidden_dim)
        hidden_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)

        self.w1 = ColumnParallelLinear(
            dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
        )
        self.w2 = RowParallelLinear(
            hidden_dim, dim, bias=False, input_is_parallel=True, init_method=lambda x: x
        )
        self.w3 = ColumnParallelLinear(
            dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
        )

    def forward(self, x):
        return self.w2(F.silu(self.w1(x)) * self.w3(x))
```

How well does it perform?

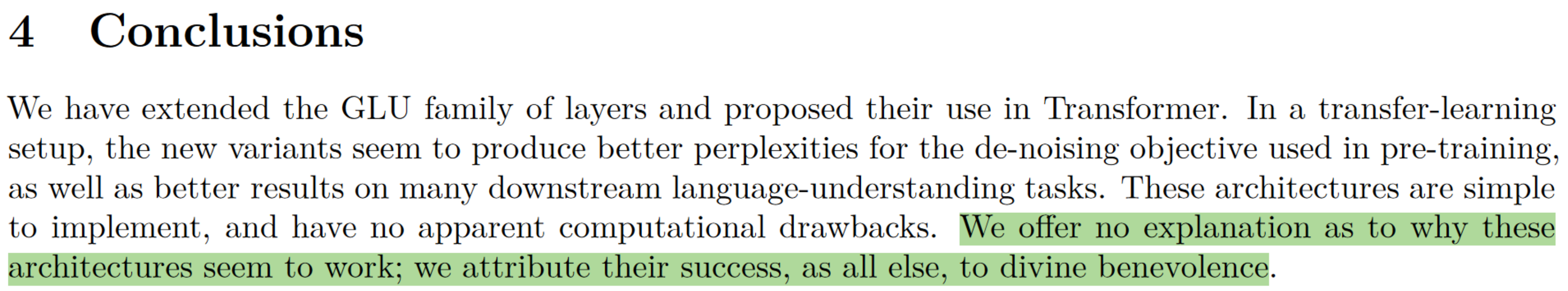
Table 1: Heldout-set log-perplexity for Transformer models on the segment-filling task from [Raffel et al., 2019]. All models are matched for parameters and computation.

Training Steps	65,536	524,288
FFN_{ReLU} (*baseline*)	1.997 (0.005)	1.677
FFN_{GELU}	1.983 (0.005)	1.679
$\text{FFN}_{\text{Swish}}$	1.994 (0.003)	1.683
FFN_{GLU}	1.982 (0.006)	1.663
$\text{FFN}_{\text{Bilinear}}$	1.960 (0.005)	1.648
$\text{FFN}_{\text{GEGLU}}$	**1.942** (0.004)	**1.633**
$\text{FFN}_{\text{SwiGLU}}$	**1.944** (0.010)	**1.636**
$\text{FFN}_{\text{ReGLU}}$	1.953 (0.003)	1.645

Table 2: GLUE Language-Understanding Benchmark [Wang et al., 2018] (dev).

	Score Average	CoLA MCC	SST-2 Acc	MRPC F1	MRPC Acc	STSB PCC	STSB SCC	QQP F1	QQP Acc	MNLIm Acc	MNLImm Acc	QNLI Acc	RTE Acc
FFN_{ReLU}	83.80	51.32	94.04	**93.08**	**90.20**	89.64	89.42	89.01	91.75	85.83	86.42	92.81	80.14
FFN_{GELU}	83.86	53.48	94.04	92.81	**90.20**	89.69	89.49	88.63	91.62	85.89	86.13	92.39	80.51
$\text{FFN}_{\text{Swish}}$	83.60	49.79	93.69	92.31	89.46	89.20	88.98	88.84	91.67	85.22	85.02	92.33	81.23
FFN_{GLU}	84.20	49.16	94.27	92.39	89.46	89.46	89.35	88.79	91.62	86.36	86.18	92.92	**84.12**
$\text{FFN}_{\text{GEGLU}}$	84.12	53.65	93.92	92.68	89.71	90.26	90.13	89.11	91.85	86.15	86.17	92.81	79.42
$\text{FFN}_{\text{Bilinear}}$	83.79	51.02	**94.38**	92.28	89.46	90.06	89.84	88.95	91.69	**86.90**	**87.08**	92.92	81.95
$\text{FFN}_{\text{SwiGLU}}$	84.36	51.59	93.92	92.23	88.97	**90.32**	**90.13**	**89.14**	**91.87**	86.45	86.47	**92.93**	83.39
$\text{FFN}_{\text{ReGLU}}$	**84.67**	**56.16**	**94.38**	92.06	89.22	89.97	89.85	88.86	91.72	86.20	86.40	92.68	81.59
[Raffel et al., 2019]	83.28	53.84	92.68	92.07	88.92	88.02	87.94	88.67	91.56	84.24	84.57	90.48	76.28
ibid. stddev.	0.235	1.111	0.569	0.729	1.019	0.374	0.418	0.108	0.070	0.291	0.231	0.361	1.393

Why SwiGLU works so well?



4 Conclusions

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

Thanks for watching!
Don’t forget to subscribe for more amazing content on AI and Machine Learning!