Introduction to Agentic AI

-- Attention and Transformer

Instructor: Guangjing Wang

guangjingwang@usf.edu

Transformer, LLM, NLP and Agentic AI

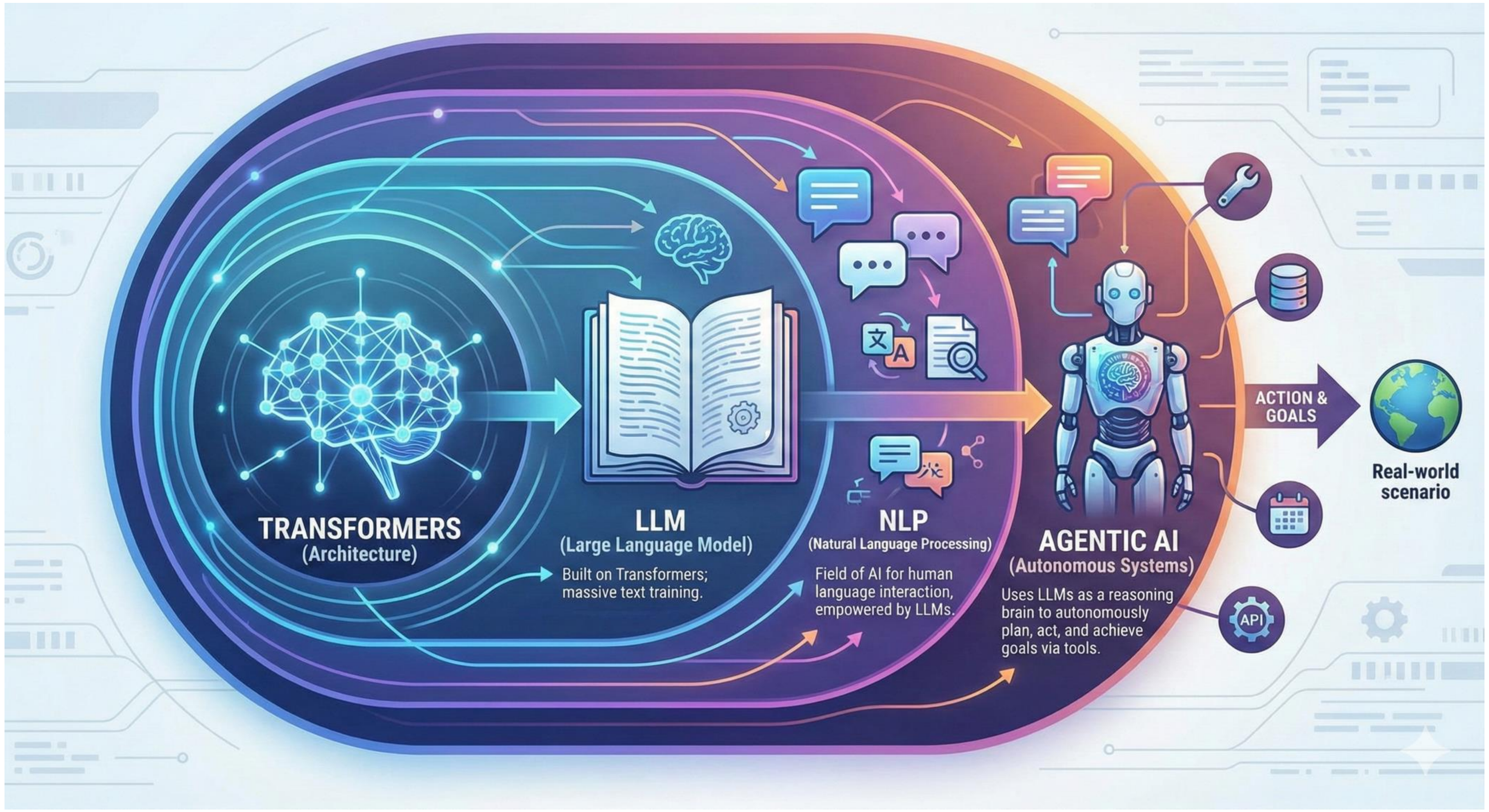
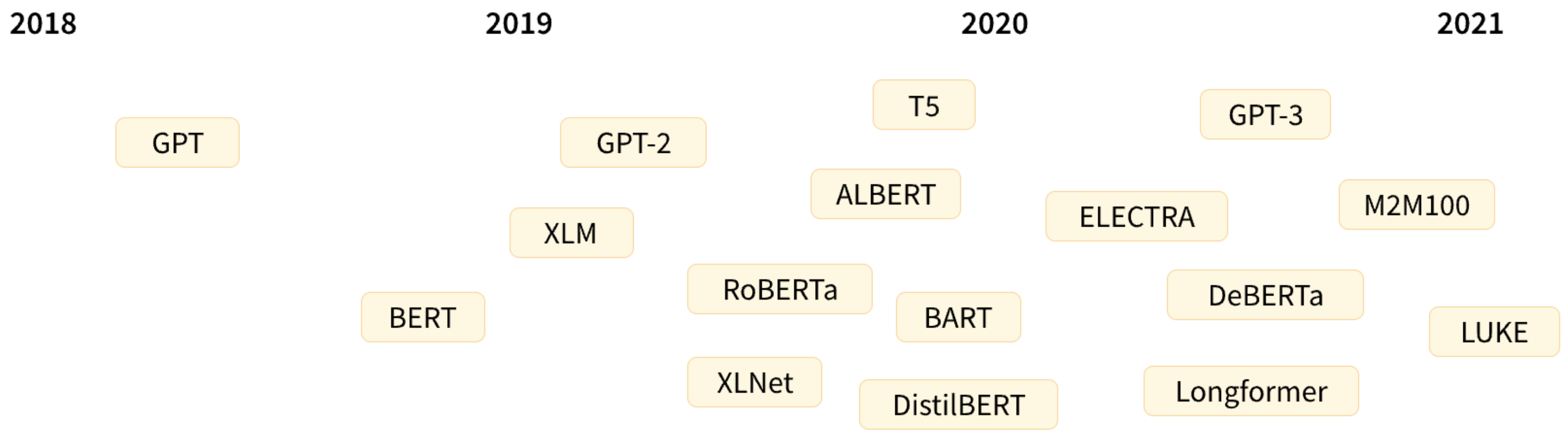

TRANSFORMERS
(Architecture)
LLM
(Large Language Model)
Built on Transformers; massive text training.
NLP
(Natural Language Processing)
Field of AI for human language interaction, empowered by LLMs.
AGENTIC AI
(Autonomous Systems)
Uses LLMs as a reasoning brain to autonomously plan, act, and achieve goals via tools.
ACTION & GOALS
API
Real-world scenario

Image generated by Nano Banana Pro

NLP, LLM and Transformer

- **NLP** is the broader field focused on enabling computers to understand, interpret, and generate human language.
- **LLMs** are a powerful subset of **NLP models** characterized by their massive size, extensive training data, and ability to perform **a wide range of language tasks** with minimal task-specific training.
- **Transformer** is the **foundational** neural network **architecture for LLM** to scale up massively to understand and generate language.

Transformer Development



GPT: Generative Pretrained **Transformer**

BERT: Bidirectional Encoder Representations from **Transformers**

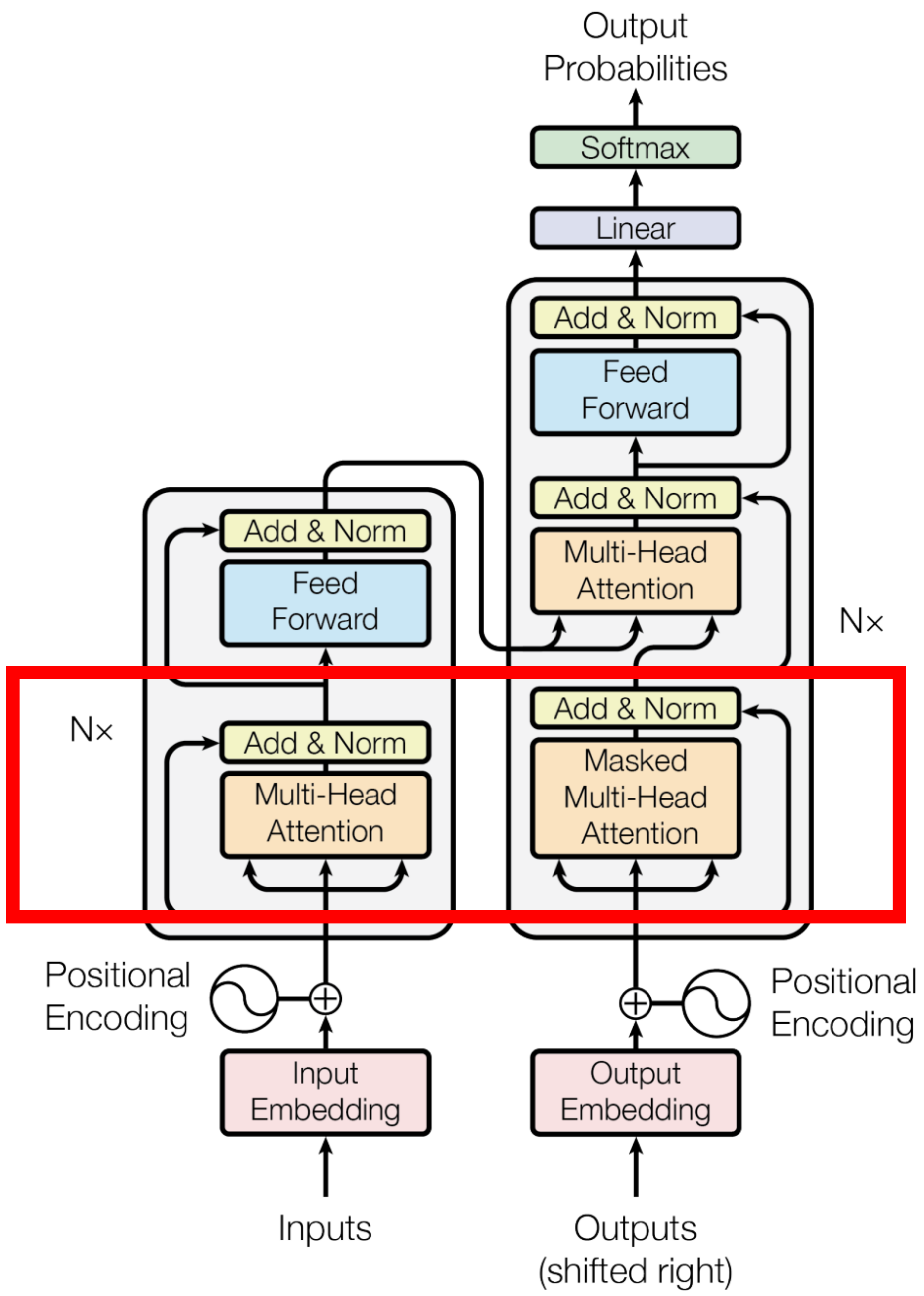
Transformer Is Trained as Language Model

- Why not in the Image Domain?
  - 2012-2018 Deep Learning in Computer Vision is a very popular.
  - Why GPT in 2018 was trained on text?

- Hint:
  - Text data is easier to get from the Internet.
  - Sequence data, such as language, can be self-supervised.
  - Self-supervised learning is a type of training in which the objective is automatically computed from the inputs. No human labeling efforts!

We will talk more about language modeling in Lecture 03.

Transformer



Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

https://arxiv.org/pdf/1706.03762

Some Questions about Transformer

- **Why does the Transformer scale the attention score before softmax?**
- **Why using mask attention?**
- **Why does the Transformer use positional encoding?**
- **Why using residual connections in Transformer?**
- **Why do we use LayerNorm?**
- **What is Q, K and V? What is their dimension?**
- **What are the differences between Transformer Encoder and Decoder?**
- **What are the aspects of Transformer parallelization?**

Attention: Learn the important of different input components

- Transformer is solely based on attention mechanisms, discarding the recurrence and convolutions entirely.

- Dot-product attention is another name for weight: weigh the importance of different input components.

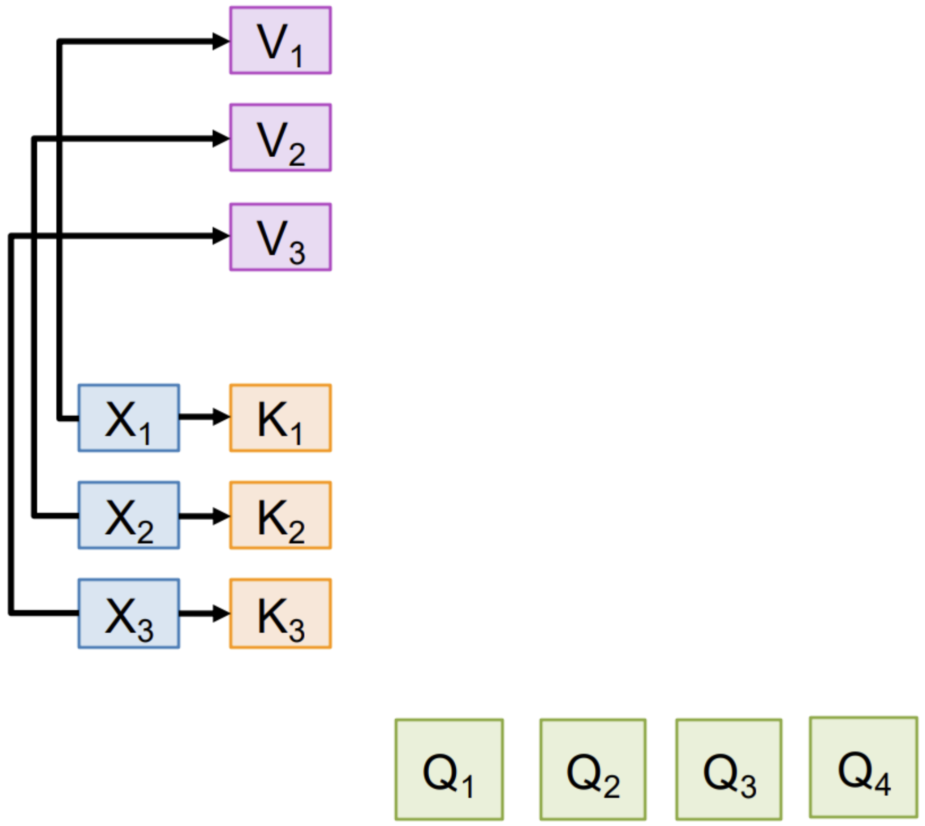
$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

Cross Attention Layer (1)

Output = AttentionFunction(X, Q, W_k, W_v)

Inputs:
Query vector: Q $[N_Q \times D_Q]$
Data vectors: X $[N_X \times D_X]$
Key matrix: W_K $[D_X \times D_Q]$
Value matrix: W_V $[D_X \times D_V]$

Computation:
Keys: $K = XW_K$ $[N_X \times D_Q]$
Values: $V = XW_V$ $[N_X \times D_V]$

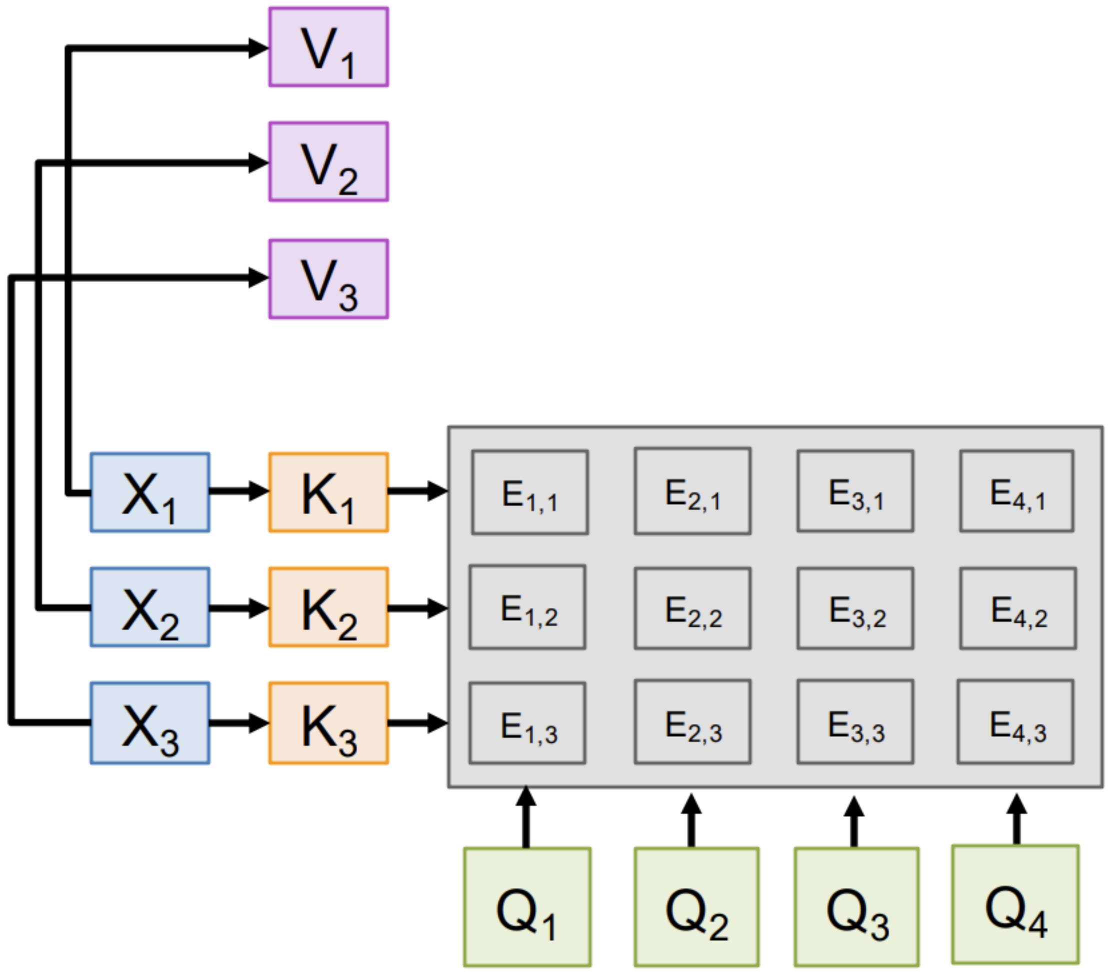


Cross Attention Layer (2)

Inputs:
Query vector: $\mathbf{Q}$ $[N_Q \times D_Q]$
Data vectors: $\mathbf{X}$ $[N_X \times D_X]$
Key matrix: $\mathbf{W_K}$ $[D_X \times D_Q]$
Value matrix: $\mathbf{W_V}$ $[D_X \times D_V]$

Computation:
Keys: $\mathbf{K} = \mathbf{XW_K}$ $[N_X \times D_Q]$
Values: $\mathbf{V} = \mathbf{XW_V}$ $[N_X \times D_V]$
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ $[N_Q \times N_X]$
$E_{ij} = \mathbf{Q_i} \cdot \mathbf{K_j} / \sqrt{D_Q}$

Note the square root of the dimension of the key vectors.



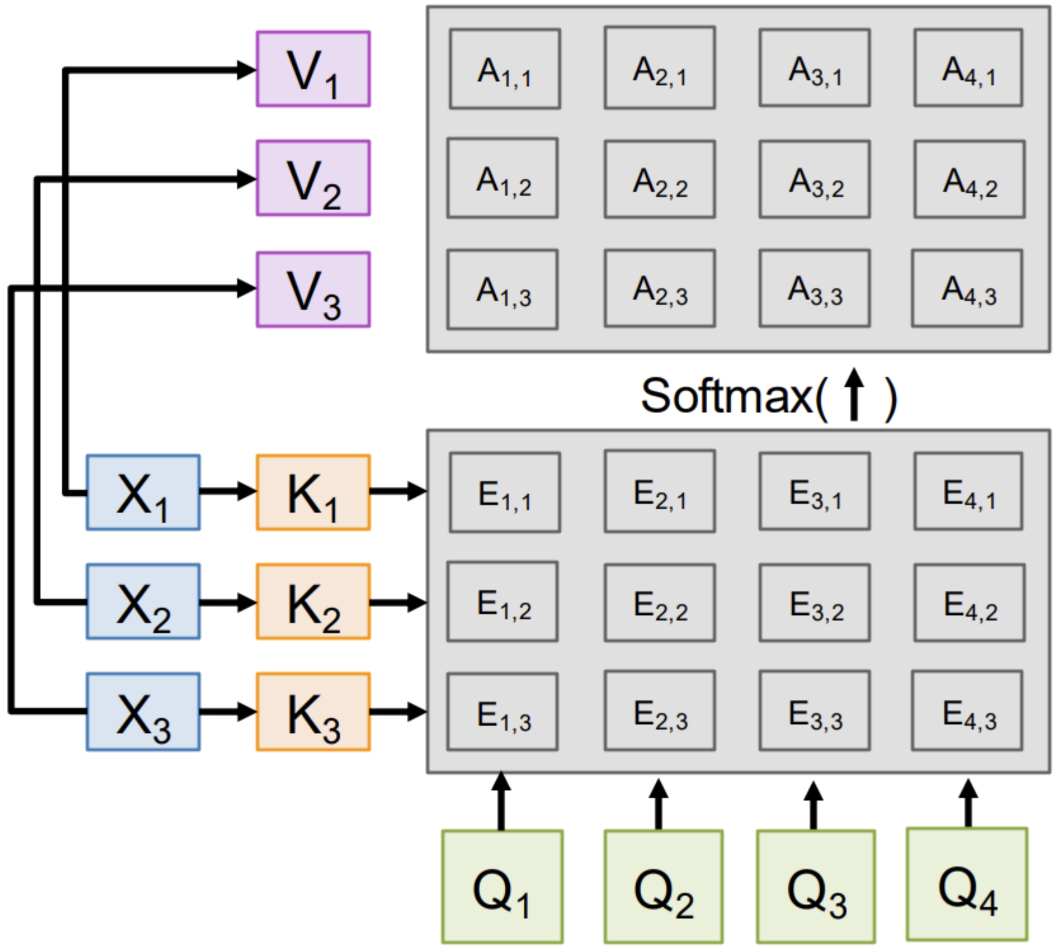
Cross Attention Layer (3)

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

Softmax normalizes each column: each query predicts a distribution over the keys

Inputs:
Query vector: Q $[N_Q \times D_Q]$
Data vectors: X $[N_X \times D_X]$
Key matrix: W_K $[D_X \times D_Q]$
Value matrix: W_V $[D_X \times D_V]$

Computation:
Keys: $K = XW_K$ $[N_X \times D_Q]$
Values: $V = XW_V$ $[N_X \times D_V]$
Similarities: $E = QK^T / \sqrt{D_Q}$ $[N_Q \times N_X]$
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ $[N_Q \times N_X]$


V_1
V_2
V_3
$A_{1,1}$ $A_{2,1}$ $A_{3,1}$ $A_{4,1}$
$A_{1,2}$ $A_{2,2}$ $A_{3,2}$ $A_{4,2}$
$A_{1,3}$ $A_{2,3}$ $A_{3,3}$ $A_{4,3}$
Softmax(↑)
X_1 K_1
X_2 K_2
X_3 K_3
$E_{1,1}$ $E_{2,1}$ $E_{3,1}$ $E_{4,1}$
$E_{1,2}$ $E_{2,2}$ $E_{3,2}$ $E_{4,2}$
$E_{1,3}$ $E_{2,3}$ $E_{3,3}$ $E_{4,3}$
Q_1 Q_2 Q_3 Q_4

Cross Attention Layer (4)

- **Why does the Transformer scale the attention score?**

Similarities: $E = QK^\top / \sqrt{D_Q}$ [N_Q x N_X]

$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$

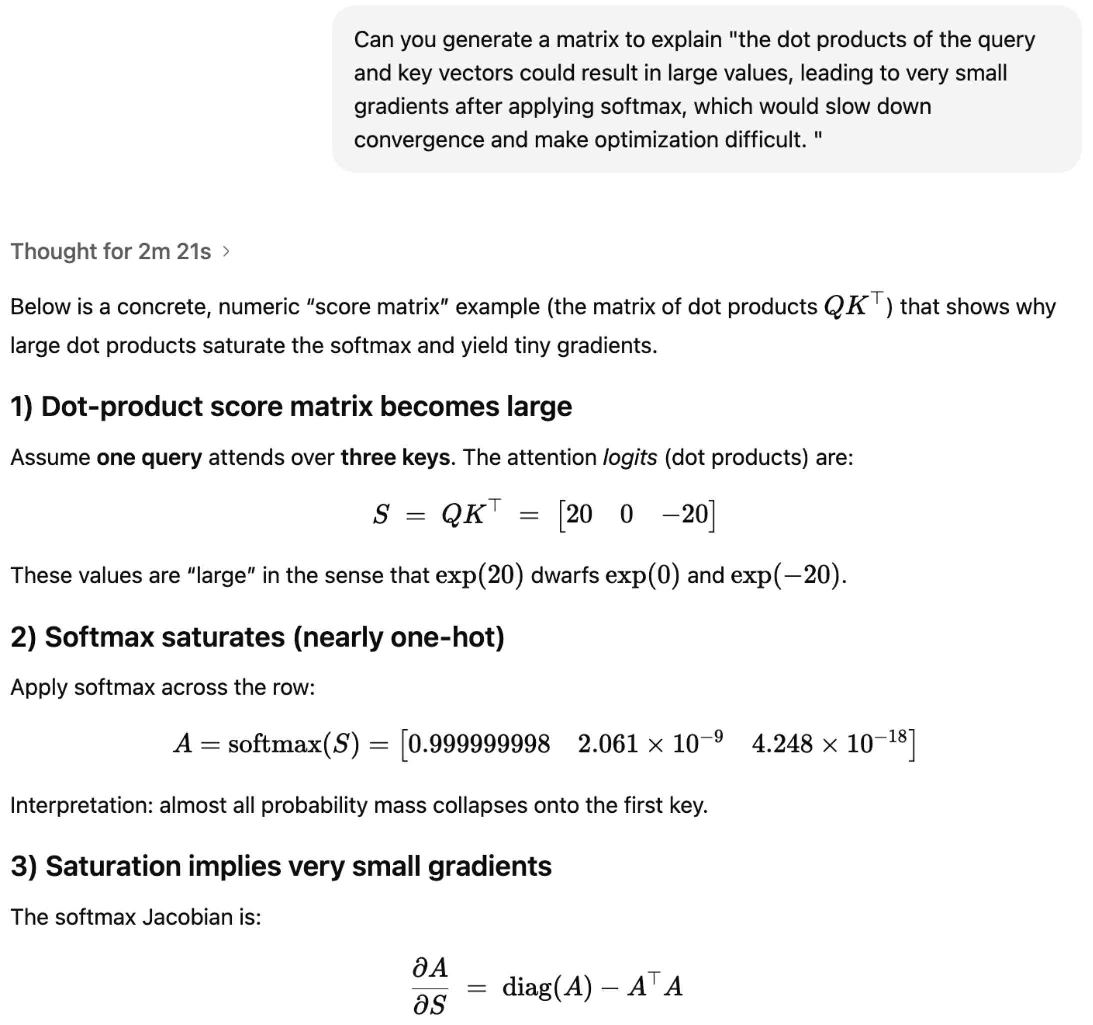
Attention weights: A = softmax(E, dim=1) [N_Q x N_X]

- Stabilize the gradients and improve training performance
- The dot products of the query and key vectors could result in large values, leading to very small gradients after applying softmax.

$$S = QK^\top = \begin{bmatrix} 20 & 0 & -20 \end{bmatrix} \quad A = \text{softmax}(S) = \begin{bmatrix} 0.999999998 & 2.061 \times 10^{-9} & 4.248 \times 10^{-18} \end{bmatrix}$$

$$S_{\text{scaled}} = \frac{S}{8} = \begin{bmatrix} 2.5 & 0 & -2.5 \end{bmatrix} \quad A_{\text{scaled}} = \text{softmax}(S_{\text{scaled}}) = \begin{bmatrix} 0.9184 & 0.0754 & 0.00619 \end{bmatrix}$$

A Top-Down Approach Learning Example (1)



Can you generate a matrix to explain "the dot products of the query and key vectors could result in large values, leading to very small gradients after applying softmax, which would slow down convergence and make optimization difficult. "

Thought for 2m 21s ›

Below is a concrete, numeric “score matrix” example (the matrix of dot products $QK^\top$) that shows why large dot products saturate the softmax and yield tiny gradients.

1) Dot-product score matrix becomes large

Assume **one query** attends over **three keys**. The attention *logits* (dot products) are:

$$S = QK^\top = \begin{bmatrix} 20 & 0 & -20 \end{bmatrix}$$

These values are “large” in the sense that $\exp(20)$ dwarfs $\exp(0)$ and $\exp(-20)$.

2) Softmax saturates (nearly one-hot)

Apply softmax across the row:

$$A = \text{softmax}(S) = \begin{bmatrix} 0.999999998 & 2.061 \times 10^{-9} & 4.248 \times 10^{-18} \end{bmatrix}$$

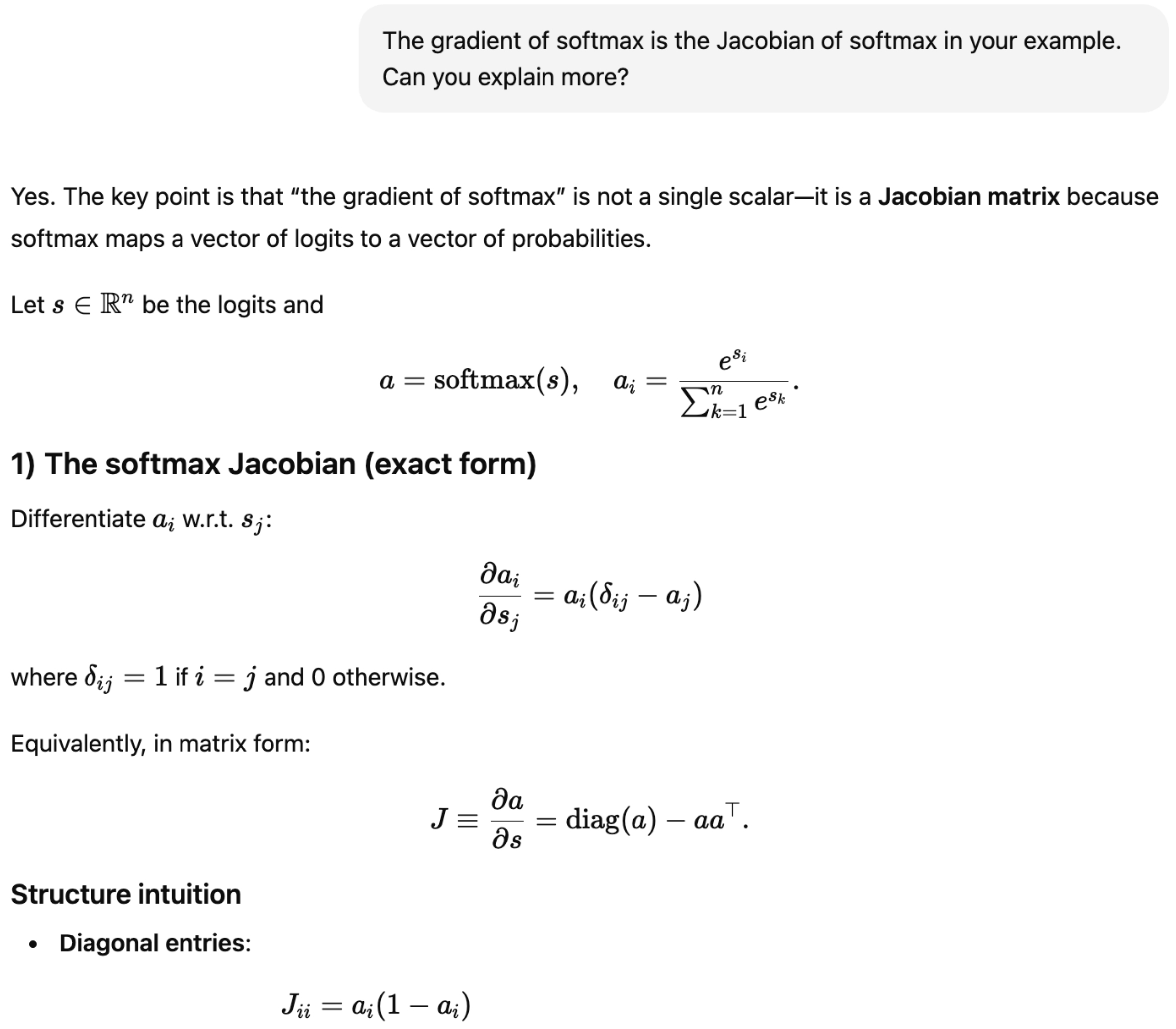
Interpretation: almost all probability mass collapses onto the first key.

3) Saturation implies very small gradients

The softmax Jacobian is:

$$\frac{\partial A}{\partial S} = \text{diag}(A) - A^\top A$$

A Top-Down Approach Learning Example (2)



The gradient of softmax is the Jacobian of softmax in your example. Can you explain more?

Yes. The key point is that “the gradient of softmax” is not a single scalar—it is a **Jacobian matrix** because softmax maps a vector of logits to a vector of probabilities.

Let $s \in \mathbb{R}^n$ be the logits and

$$a = \mathrm{softmax}(s), \quad a_i = \frac{e^{s_i}}{\sum_{k=1}^{n} e^{s_k}}.$$

1) The softmax Jacobian (exact form)

Differentiate a_i w.r.t. s_j:

$$\frac{\partial a_i}{\partial s_j} = a_i(\delta_{ij} - a_j)$$

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise.

Equivalently, in matrix form:

$$J \equiv \frac{\partial a}{\partial s} = \mathrm{diag}(a) - aa^\top.$$

Structure intuition

- **Diagonal entries**:

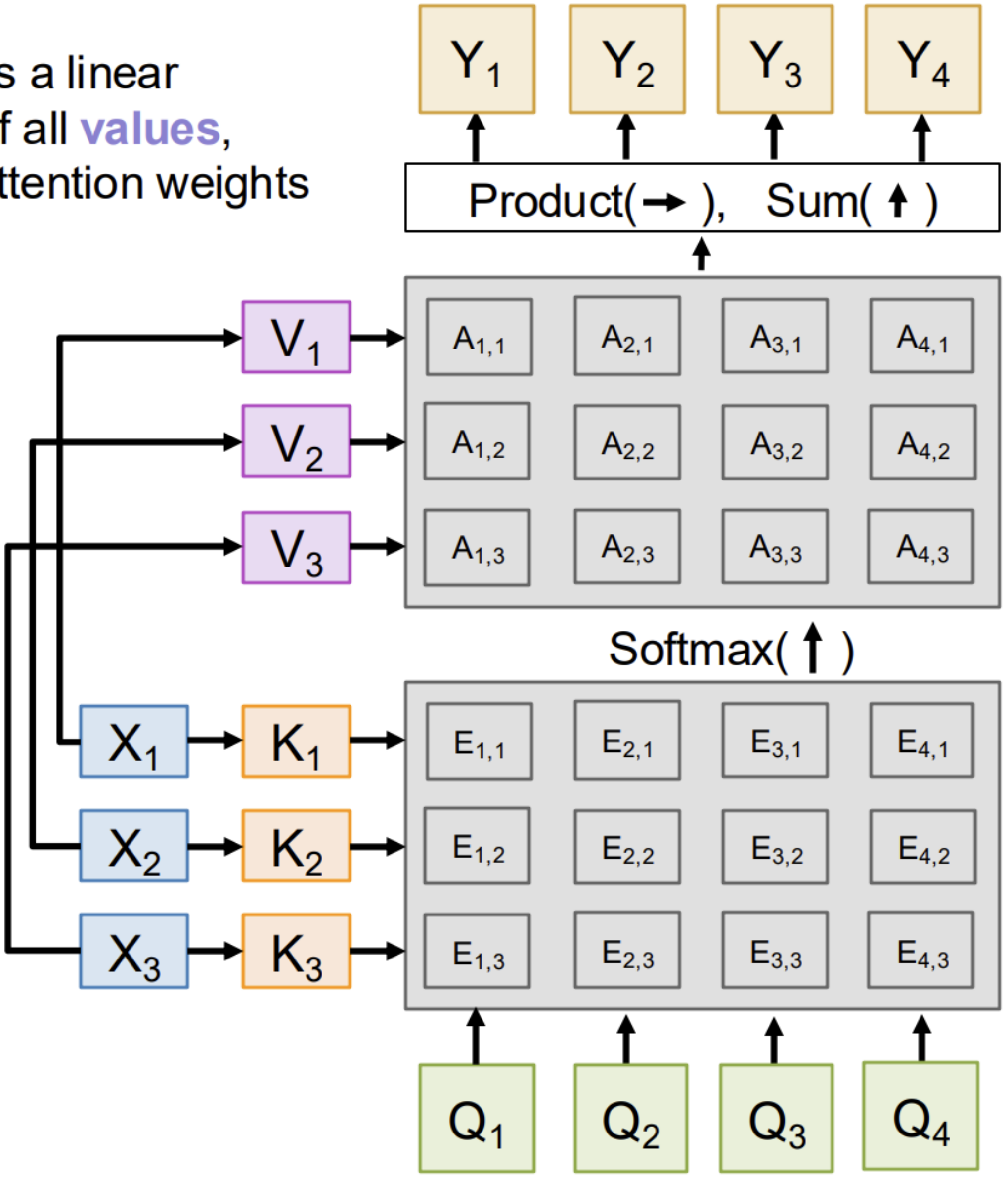
$$J_{ii} = a_i(1 - a_i)$$

Cross Attention (5)

Inputs:
Query vector: Q $[N_Q \times D_Q]$
Data vectors: X $[N_X \times D_X]$
Key matrix: W_K $[D_X \times D_Q]$
Value matrix: W_V $[D_X \times D_V]$

Computation:
Keys: $K = XW_K$ $[N_X \times D_Q]$
Values: $V = XW_V$ $[N_X \times D_V]$
Similarities: $E = QK^T / \sqrt{D_Q}$ $[N_Q \times N_X]$
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) $[N_Q \times N_X]$
Output vector: $Y = AV$ $[N_Q \times D_V]$
$Y_i = \sum_j A_{ij} V_j$

Each **output** is a linear combination of all **values**, weighted by attention weights

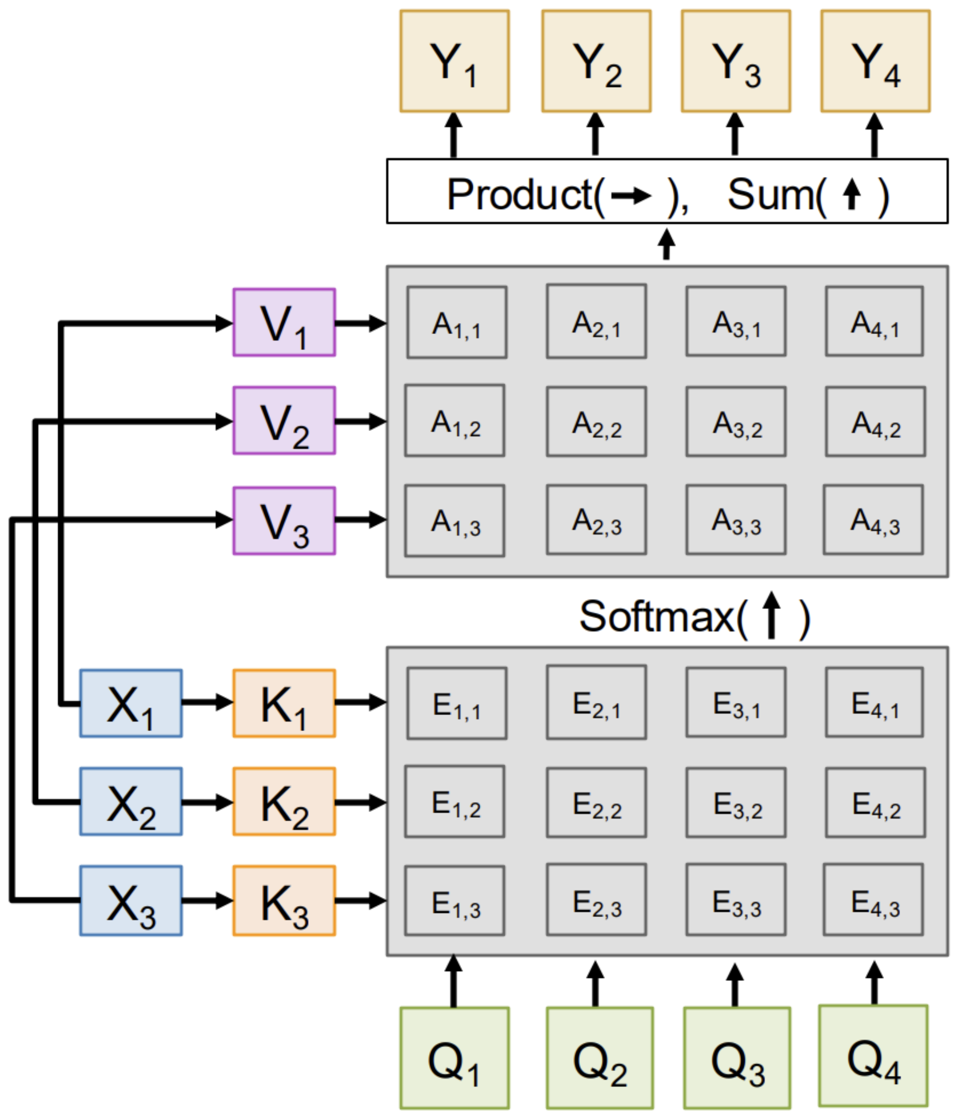

Y_1
Y_2
Y_3
Y_4
Product(→), Sum(↑)
V_1
V_2
V_3
$A_{1,1}$ $A_{2,1}$ $A_{3,1}$ $A_{4,1}$
$A_{1,2}$ $A_{2,2}$ $A_{3,2}$ $A_{4,2}$
$A_{1,3}$ $A_{2,3}$ $A_{3,3}$ $A_{4,3}$
Softmax(↑)
X_1
X_2
X_3
K_1
K_2
K_3
$E_{1,1}$ $E_{2,1}$ $E_{3,1}$ $E_{4,1}$
$E_{1,2}$ $E_{2,2}$ $E_{3,2}$ $E_{4,2}$
$E_{1,3}$ $E_{2,3}$ $E_{3,3}$ $E_{4,3}$
Q_1
Q_2
Q_3
Q_4

Cross Attention (6)

Output = AttentionFunction(X, Q, W_K, W_V)

<u>Inputs</u>:
Query vector: Q [N_Q x D_Q]
Data vectors: X [N_X x D_X]
Key matrix: W_K [D_X x D_Q]
Value matrix: W_V [D_X x D_V]

Each query produces one output, which is a mix of information in the data vectors.



Y_1 Y_2 Y_3 Y_4
Product(→), Sum(↑)
V_1 V_2 V_3
$A_{1,1}$ $A_{2,1}$ $A_{3,1}$ $A_{4,1}$
$A_{1,2}$ $A_{2,2}$ $A_{3,2}$ $A_{4,2}$
$A_{1,3}$ $A_{2,3}$ $A_{3,3}$ $A_{4,3}$
Softmax(↑)
X_1 X_2 X_3
K_1 K_2 K_3
$E_{1,1}$ $E_{2,1}$ $E_{3,1}$ $E_{4,1}$
$E_{1,2}$ $E_{2,2}$ $E_{3,2}$ $E_{4,2}$
$E_{1,3}$ $E_{2,3}$ $E_{3,3}$ $E_{4,3}$
Q_1 Q_2 Q_3 Q_4

Self-Attention (1)

Output = AttentionFunction(X, W_Q, W_K, W_V)

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

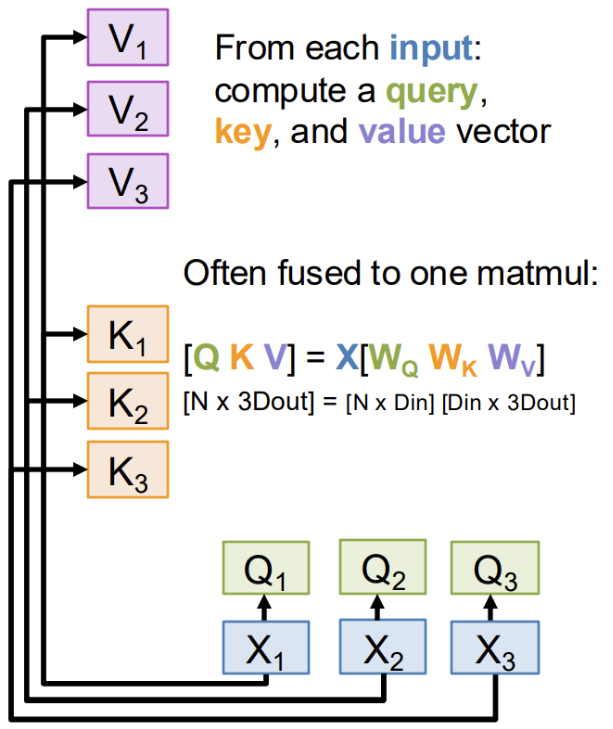
Each input produces one output, which is a mix of information from all inputs

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]


V_1
V_2
V_3
From each input: compute a query, key, and value vector
Often fused to one matmul:
$[Q\ K\ V] = X[W_Q\ W_K\ W_V]$
[N x 3Dout] = [N x Din] [Din x 3Dout]
K_1
K_2
K_3
Q_1
Q_2
Q_3
X_1
X_2
X_3

N: is the length of the input (e.g., the number of tokens)

D_{in}: is the dimension of the feature (e.g., the dimension of the token embedding)

Self-Attention (2)

Inputs:

Input vectors: X [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

Each input produces one output, which is a mix of information from all inputs

Computation:

Queries: $Q = XW_Q$ [N x D_{out}]

Keys: $K = XW_K$ [N x D_{out}]

Values: $V = XW_V$ [N x D_{out}]

Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]

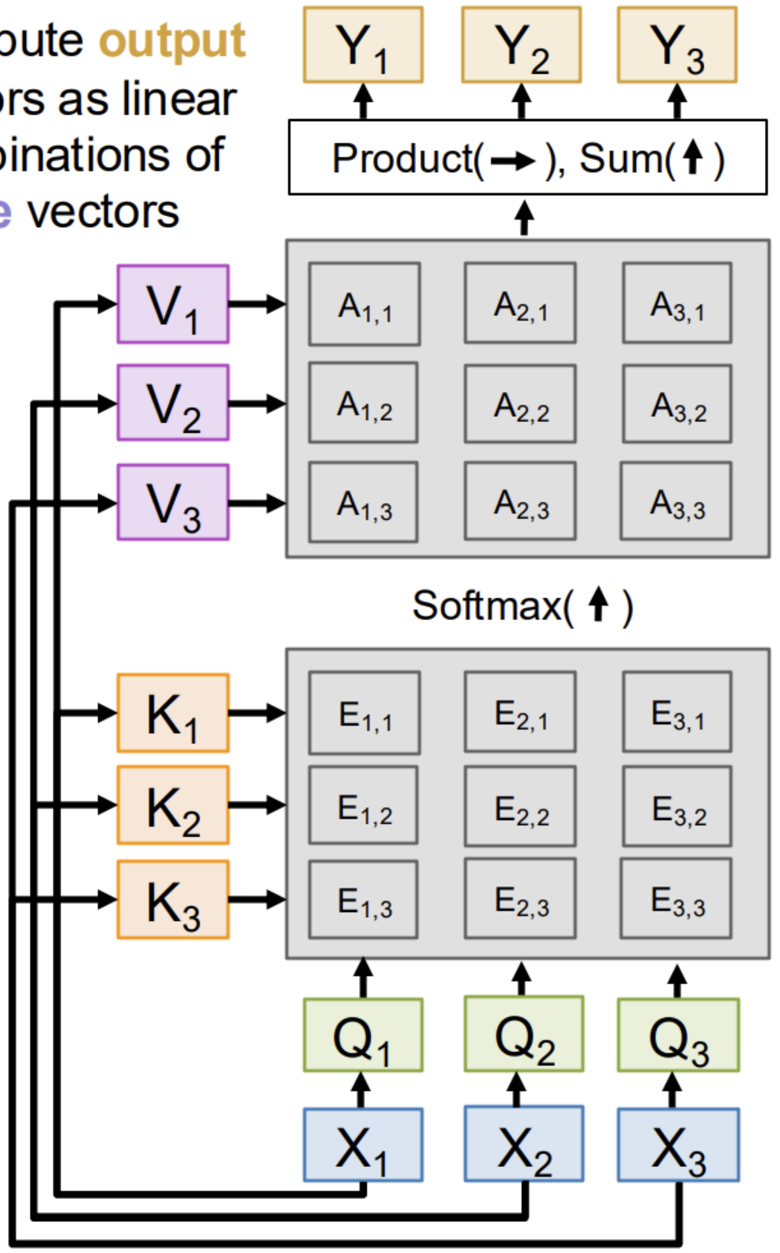
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]

Output vector: $Y = AV$ [N x D_{out}]

$Y_i = \sum_j A_{ij} V_j$

Compute output vectors as linear combinations of value vectors



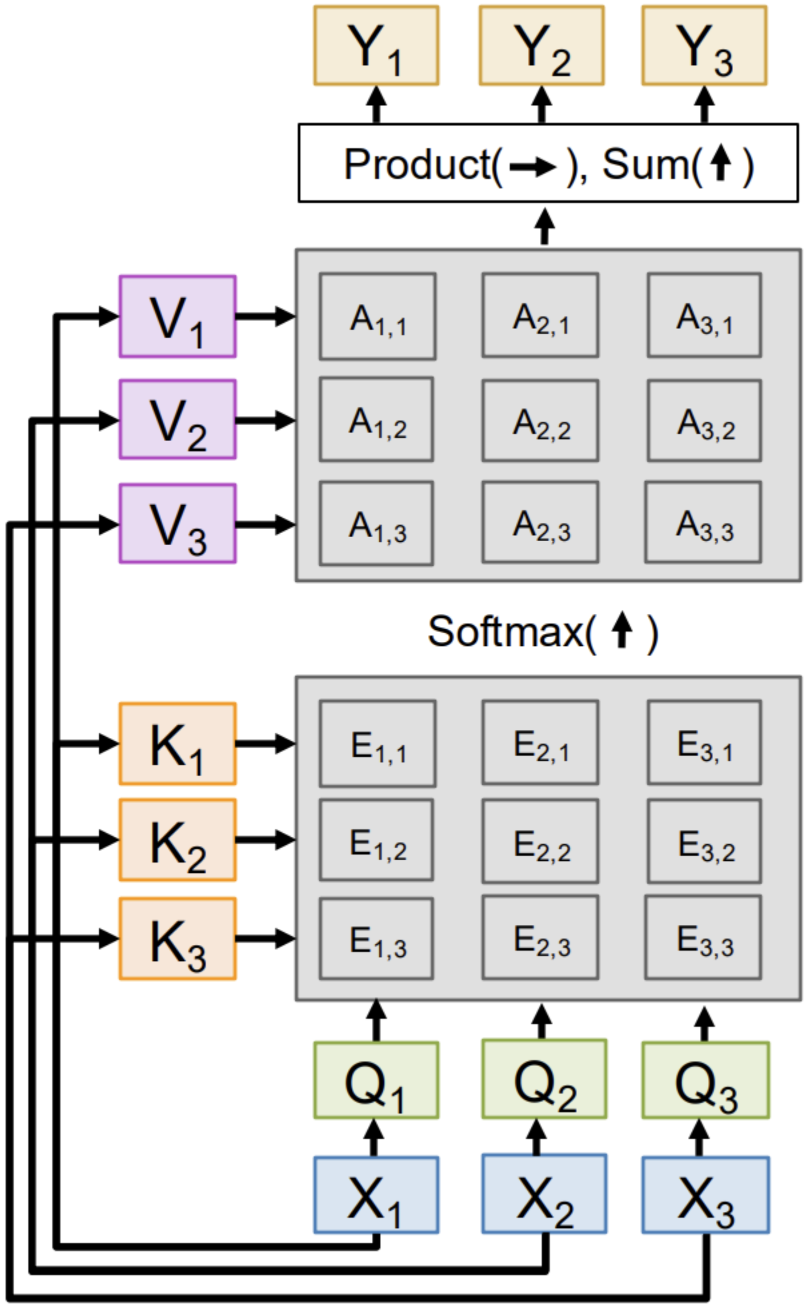
Self-Attention (3)

Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$

Each input produces one output, which is a mix of information from all inputs

Shapes get a little simpler:
- N input vectors, each D_{in}
- Almost always $D_Q = D_V = D_{out}$



Y_1
Y_2
Y_3
Product(→), Sum(↑)
V_1
V_2
V_3
$A_{1,1}$ $A_{2,1}$ $A_{3,1}$
$A_{1,2}$ $A_{2,2}$ $A_{3,2}$
$A_{1,3}$ $A_{2,3}$ $A_{3,3}$
Softmax(↑)
K_1
K_2
K_3
$E_{1,1}$ $E_{2,1}$ $E_{3,1}$
$E_{1,2}$ $E_{2,2}$ $E_{3,2}$
$E_{1,3}$ $E_{2,3}$ $E_{3,3}$
Q_1
Q_2
Q_3
X_1
X_2
X_3

Self-Attention (4)

Inputs:
Input vectors: X [$N \times D_{in}$]
Key matrix: W_K [$D_{in} \times D_{out}$]
Value matrix: W_V [$D_{in} \times D_{out}$]
Query matrix: W_Q [$D_{in} \times D_{out}$]

Computation:
Queries: $Q = XW_Q$ [$N \times D_{out}$]
Keys: $K = XW_K$ [$N \times D_{out}$]
Values: $V = XW_V$ [$N \times D_{out}$]
Similarities: $E = QK^T / \sqrt{D_Q}$ [$N \times N$]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [$N \times N$]
Output vector: $Y = AV$ [$N \times D_{out}$]

$$Y_i = \sum_j A_{ij} V_j$$

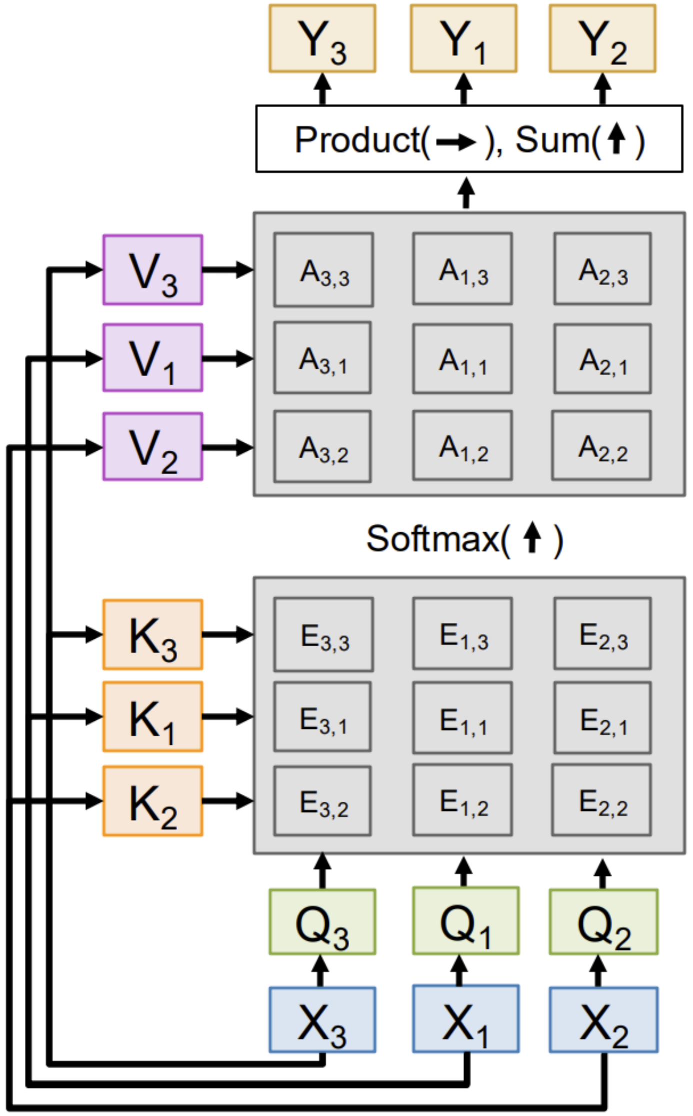
Consider permuting **inputs**:

Queries, **keys**, and **values** will be the same but permuted

Similarities are the same but permuted

Attention weights are the same but permuted

Outputs are the same but permuted

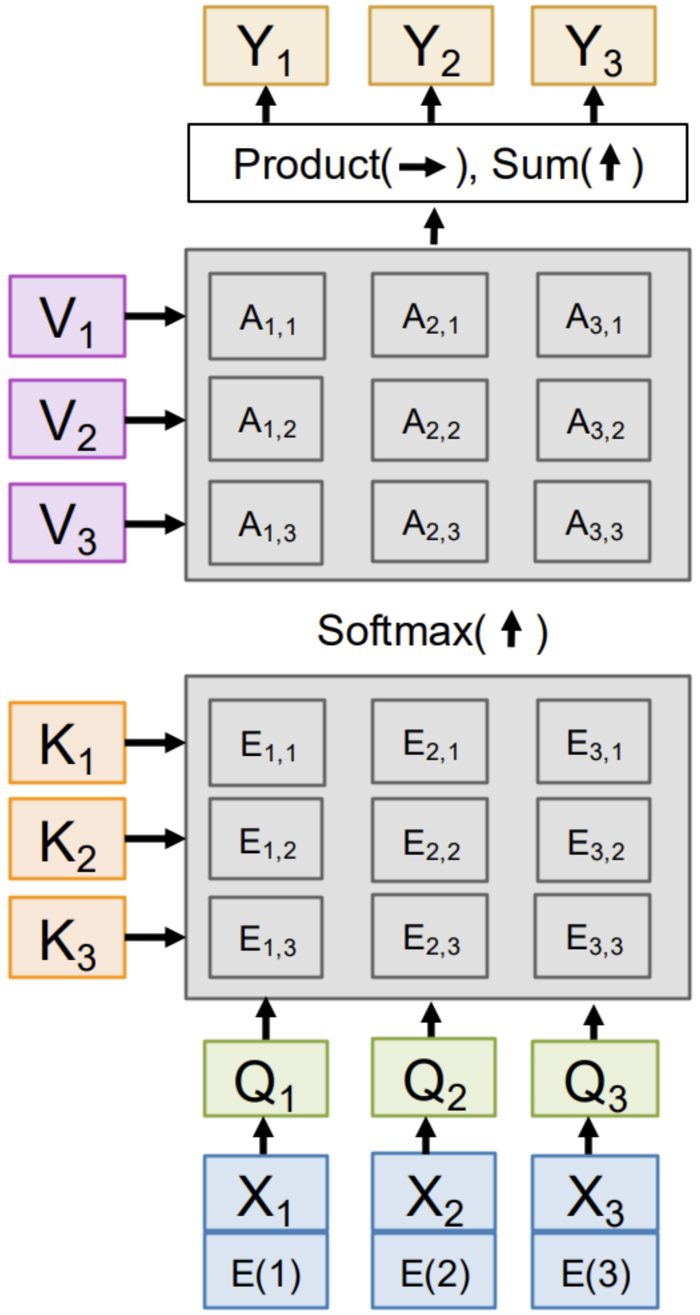


Self-Attention is permutation equivariant!

Self-Attention (5)

- Self-Attention is permutation equivariant!
- Problem: Self-Attention does not know the order of the sequence

- Solution: Add **positional encoding** to each input; this is a vector that is a fixed function of the index



Y_1 Y_2 Y_3
Product(→), Sum(↑)
V_1 V_2 V_3
$A_{1,1}$ $A_{2,1}$ $A_{3,1}$
$A_{1,2}$ $A_{2,2}$ $A_{3,2}$
$A_{1,3}$ $A_{2,3}$ $A_{3,3}$
Softmax(↑)
K_1 K_2 K_3
$E_{1,1}$ $E_{2,1}$ $E_{3,1}$
$E_{1,2}$ $E_{2,2}$ $E_{3,2}$
$E_{1,3}$ $E_{2,3}$ $E_{3,3}$
Q_1 Q_2 Q_3
X_1 X_2 X_3
E(1) E(2) E(3)

Positional Encoding

- Positional encoding is to inject information about the relative or absolute position of the tokens in the sequence.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

pos is the position and i is the i-th dimension of d_model (in token embedding)

Example:

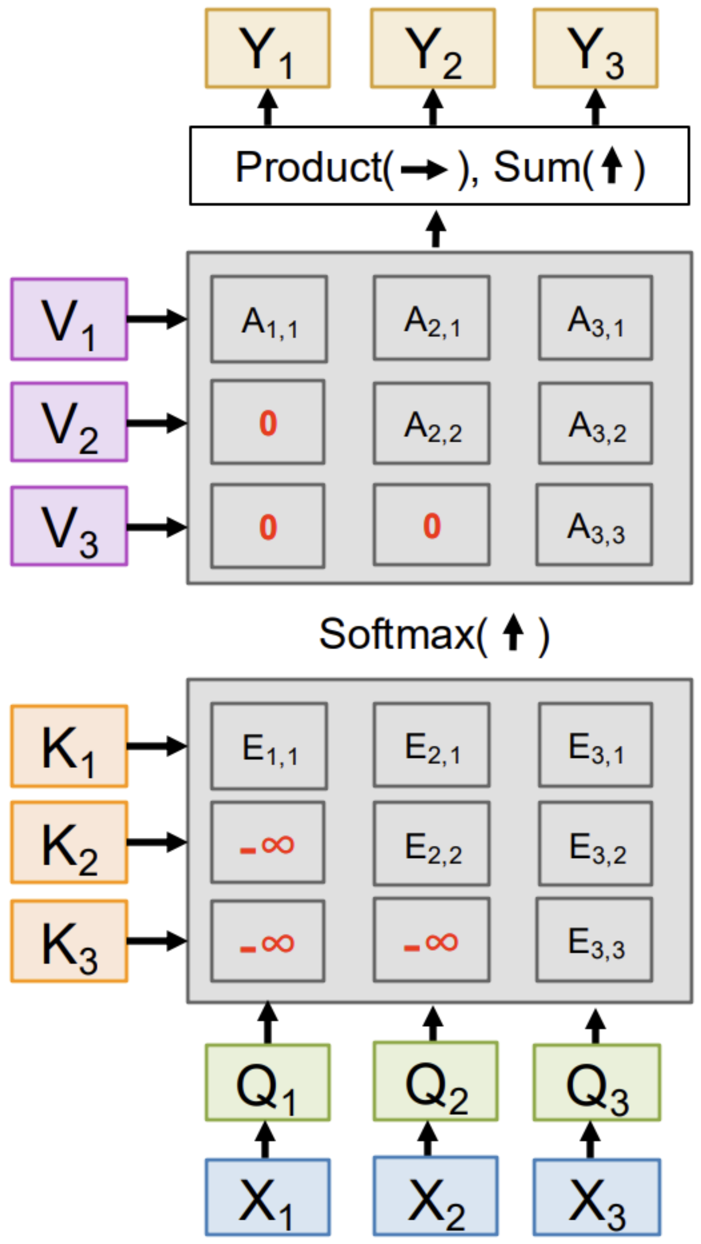
pos	dim0	dim1	dim2	dim3
0	0.0000	1.0000	0.0000	1.0000
1	0.8415	0.5403	0.0100	0.9999
2	0.9093	−0.4161	0.0200	0.9998
3	0.1411	−0.9900	0.0300	0.9996

- The final input embeddings are the sum or concatenation of the learnable embedding and the positional encoding.

Suggested reading: Convolutional sequence to sequence learning https://arxiv.org/abs/1705.03122

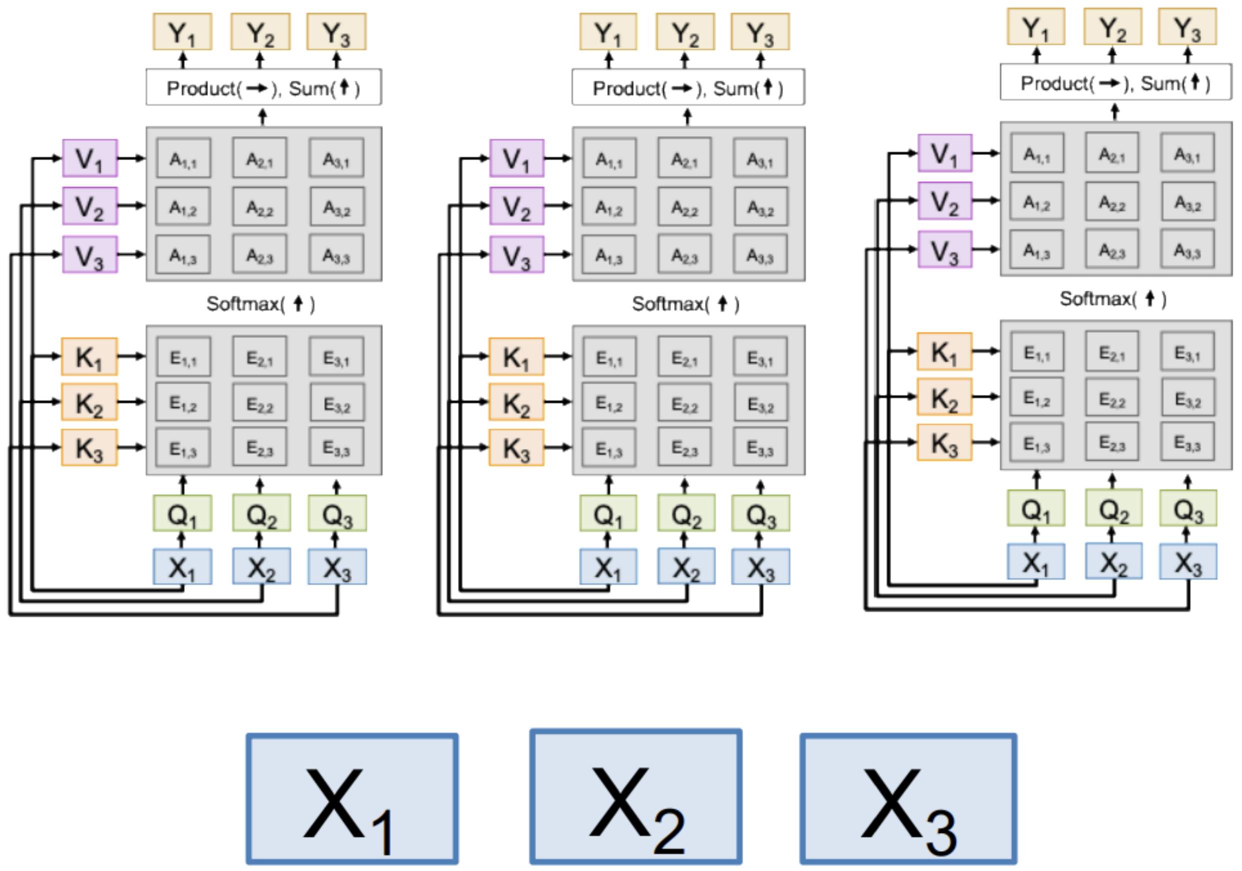
Masked Self-Attention Layer

- Override similarities with -inf;
- Mask controls which inputs each vector is allowed to look at: only previous information, not the future information.
- Used for language modeling where you want to predict the next word.



Multiheaded Self-Attention (1)

H = 3 independent self-attention layers (called heads), each with their own weights

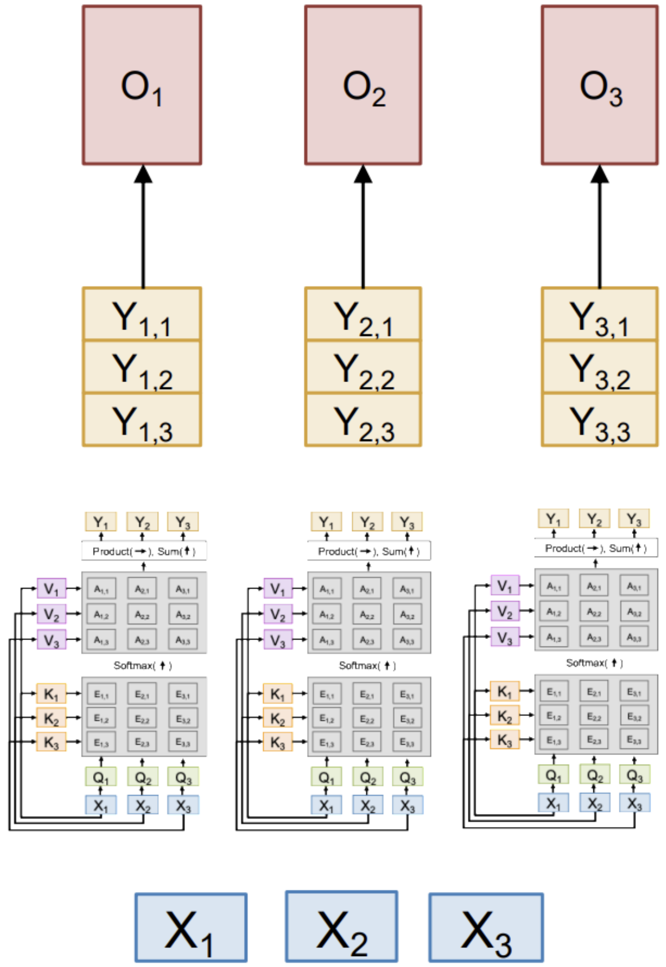


Multiheaded Self-Attention (2)

- Output projection fuses data from each head

- Stack up the H independent outputs for each input X

- H = 3 independent self-attention layers (**called heads**), each with their own weights



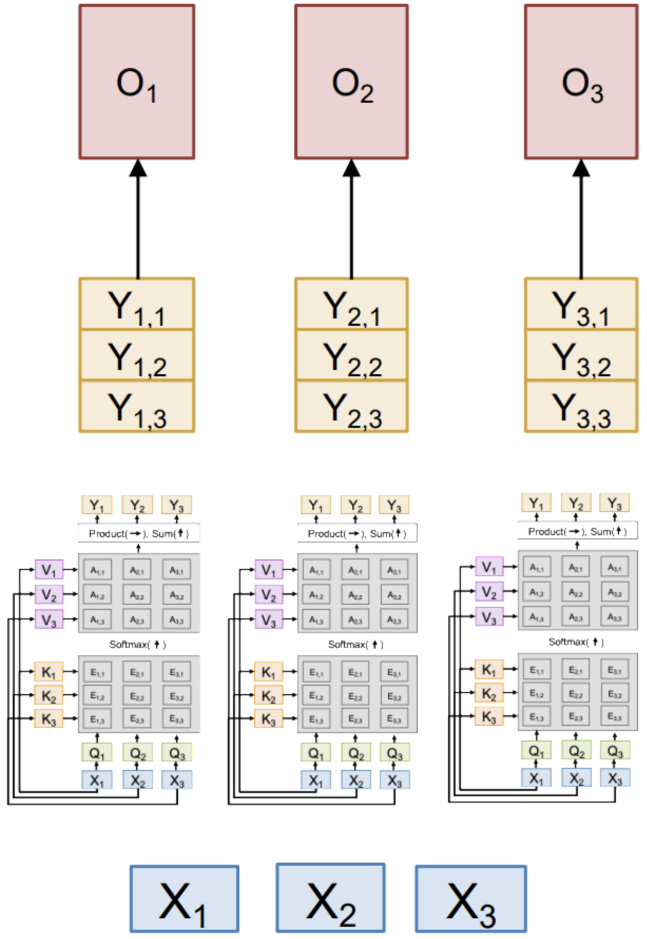
Multiheaded Self-Attention (3)

Inputs:
Input vectors: X [N x D]
Key matrix: W_K [D x HD_H]
Value matrix: W_V [D x HD_H]
Query matrix: W_Q [D x HD_H]
Output matrix: W_O [HD_H x D]

Computation:
Queries: $Q = XW_Q$ [H x N x D_H]
Keys: $K = XW_K$ [H x N x D_H]
Values: $V = XW_V$ [H x N x D_H]
Similarities: $E = QK^T / \sqrt{D_Q}$ [H x N x N]
Attention weights: A = softmax(E, dim=2) [H x N x N]
Head outputs: $Y = AV$ [H x N x D_H] => [N x HD_H]
Outputs: $O = YW_O$ [N x D]

Each of the H parallel layers use a qkv dim of D_H = "head dim"

Usually $D_H = D / H$, so inputs and outputs have the same dimension



Self-Attention is Four Matrix Multiplies!

Inputs:

Input vectors: X [N x D]
Key matrix: W_K [D x HD_H]
Value matrix: W_V [D x HD_H]
Query matrix: W_Q [D x HD_H]
Output matrix: W_O [HD_H x D]

Computation:

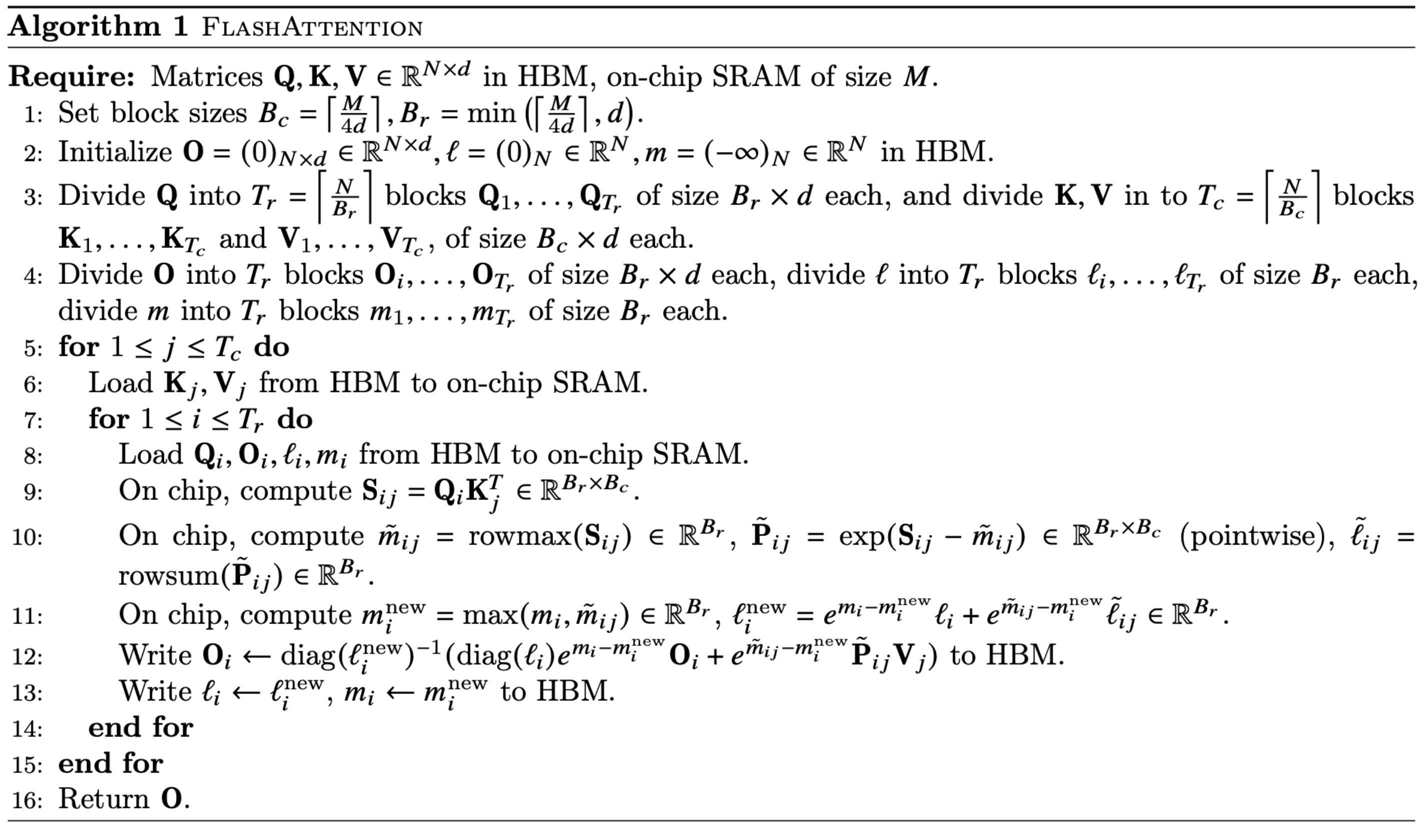
Queries: $Q = XW_Q$ [H x N x D_H]
Keys: $K = XW_K$ [H x N x D_H]
Values: $V = XW_V$ [H x N x D_H]
Similarities: $E = QK^T / \sqrt{D_Q}$ [H x N x N]
Attention weights: A = softmax(E, dim=2) [H x N x N]
Head outputs: $Y = AV$ [H x N x D_H] => [N x HD_H]
Outputs: $O = YW_O$ [N x D]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get **Q**, **K**, **V** each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] **[H x D_H x N]** => **[H x N x N]**
3. V-Weighting
   [H x N x N] **[H x N x D_H]** => **[H x N x D_H]**
   Reshape to **[N x HD_H]**
4. Output Projection
   [N x HD_H] **[HD_H x D]** => [N x D]

In practice, compute all H heads **in parallel** using batched matrix multiply operations!

Algorithmic complexity of Self-Attention

- Time Complexity?
  - O(N^2)
- Memory Complexity?
  - O(N^2)
  - O(N) with **Flash Attention**

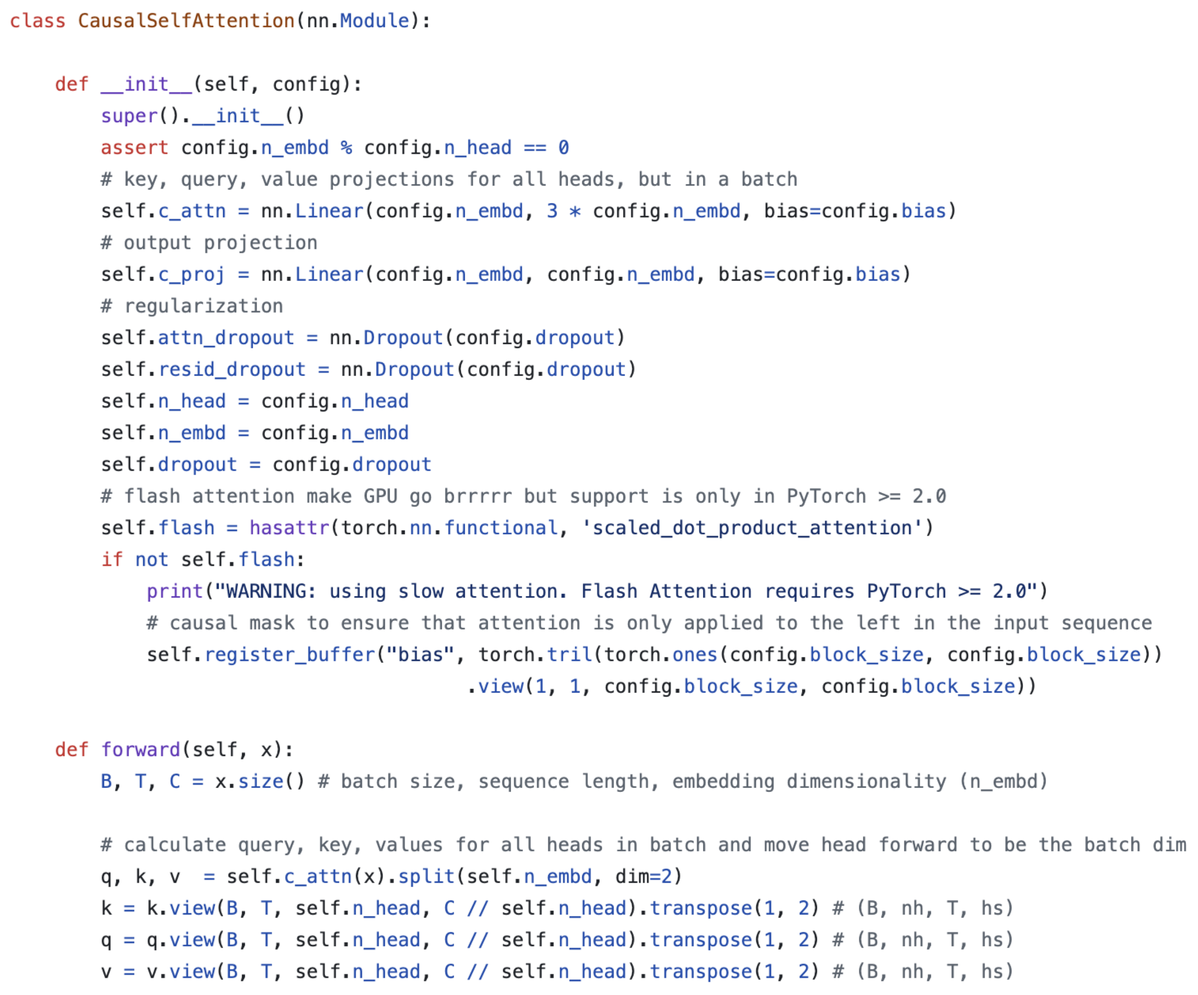


Algorithm 1 FlashAttention

```
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 $B_c = \left\lceil \frac{M}{4d} \right\rceil, B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.
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 $\mathbf{Q}_1, \dots, \mathbf{Q}_{T_r}$ 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 $\mathbf{K}_1, \dots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \dots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.
4: Divide $\mathbf{O}$ into $T_r$ blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide $\ell$ into $T_r$ blocks $\ell_i, \dots, \ell_{T_r}$ of size $B_r$ each, divide $m$ into $T_r$ blocks $m_1, \dots, m_{T_r}$ 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 $\tilde{m}_{ij} = \mathrm{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \mathrm{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.
11:    On chip, compute $m_i^{\mathrm{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\mathrm{new}} = e^{m_i - m_i^{\mathrm{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.
12:    Write $\mathbf{O}_i \leftarrow \mathrm{diag}(\ell_i^{\mathrm{new}})^{-1}(\mathrm{diag}(\ell_i) e^{m_i - m_i^{\mathrm{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\mathrm{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.
13:    Write $\ell_i \leftarrow \ell_i^{\mathrm{new}}$, $m_i \leftarrow m_i^{\mathrm{new}}$ to HBM.
14:  end for
15: end for
16: Return $\mathbf{O}$.
```

GPU high bandwidth memory (HBM) and GPU on-chip SRAM
https://arxiv.org/pdf/2205.14135 (Not required but suggested reading!)

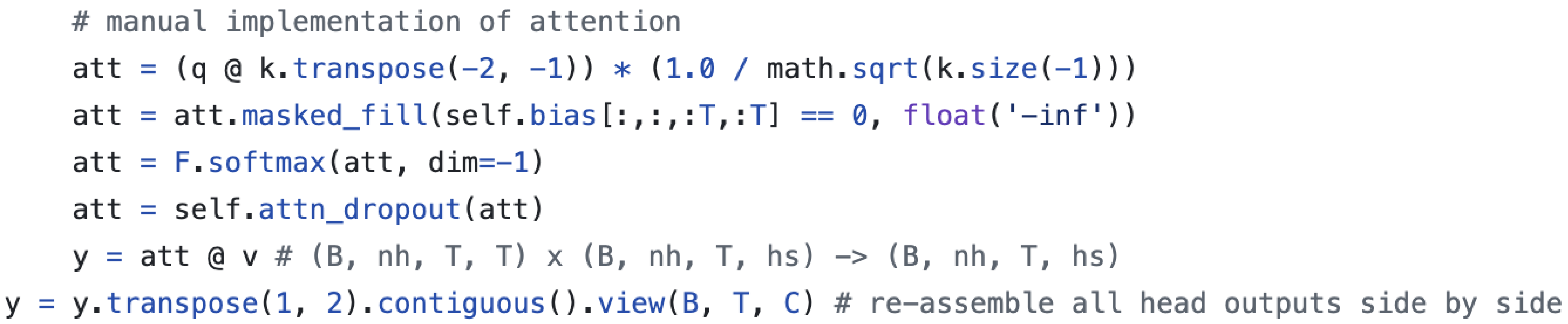
Peek the Code



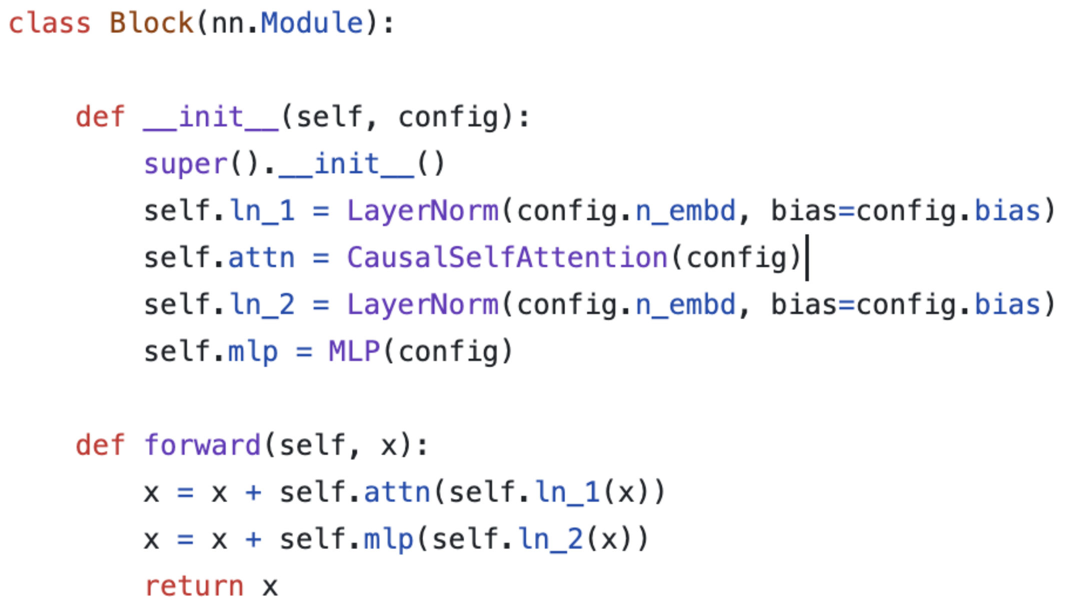
```
class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd, bias=config.bias)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd, bias=config.bias)
        # regularization
        self.attn_dropout = nn.Dropout(config.dropout)
        self.resid_dropout = nn.Dropout(config.dropout)
        self.n_head = config.n_head
        self.n_embd = config.n_embd
        self.dropout = config.dropout
        # flash attention make GPU go brrrrr but support is only in PyTorch >= 2.0
        self.flash = hasattr(torch.nn.functional, 'scaled_dot_product_attention')
        if not self.flash:
            print("WARNING: using slow attention. Flash Attention requires PyTorch >= 2.0")
            # causal mask to ensure that attention is only applied to the left in the input sequence
            self.register_buffer("bias", torch.tril(torch.ones(config.block_size, config.block_size))
                                        .view(1, 1, config.block_size, config.block_size))

    def forward(self, x):
        B, T, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)

        # calculate query, key, values for all heads in batch and move head forward to be the batch dim
        q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
```

https://github.com/karpathy/nanoGPT/blob/master/model.py

Peek the Code



```
    # manual implementation of attention
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)
    y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
y = y.transpose(1, 2).contiguous().view(B, T, C) # re-assemble all head outputs side by side
```



```
class Block(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.ln_1 = LayerNorm(config.n_embd, bias=config.bias)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = LayerNorm(config.n_embd, bias=config.bias)
        self.mlp = MLP(config)

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x
```

Peek the Code

```python
class GPT(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.vocab_size is not None
        assert config.block_size is not None
        self.config = config

        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size, config.n_embd),
            wpe = nn.Embedding(config.block_size, config.n_embd),
            drop = nn.Dropout(config.dropout),
            h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = LayerNorm(config.n_embd, bias=config.bias),
        ))
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
        # with weight tying when using torch.compile() some warnings get generated:
        # "UserWarning: functional_call was passed multiple values for tied weights.
        # This behavior is deprecated and will be an error in future versions"
        # not 100% sure what this is, so far seems to be harmless. TODO investigate
        self.transformer.wte.weight = self.lm_head.weight # https://paperswithcode.com/method/weight-tying

        # init all weights
        self.apply(self._init_weights)
        # apply special scaled init to the residual projections, per GPT-2 paper
        for pn, p in self.named_parameters():
            if pn.endswith('c_proj.weight'):
                torch.nn.init.normal_(p, mean=0.0, std=0.02/math.sqrt(2 * config.n_layer))
```

```python
def _init_weights(self, module):
    if isinstance(module, nn.Linear):
        torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
        if module.bias is not None:
            torch.nn.init.zeros_(module.bias)
    elif isinstance(module, nn.Embedding):
        torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
```

```python
# forward the GPT model itself
tok_emb = self.transformer.wte(idx) # token embeddings of shape (b, t, n_embd)
pos_emb = self.transformer.wpe(pos) # position embeddings of shape (t, n_embd)
x = self.transformer.drop(tok_emb + pos_emb)
for block in self.transformer.h:
    x = block(x)
x = self.transformer.ln_f(x)
```

The Transformer (1)

- Transformer Block
  - Input: Set of vectors X

Recall **Layer Normalization**:

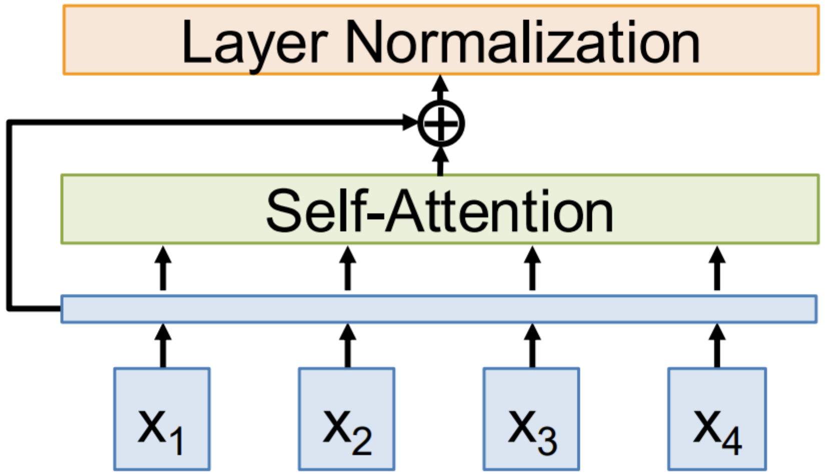
Given $h_1, \ldots, h_N$	(Shape: D)
scale: γ	(Shape: D)
shift: β	(Shape: D)
$\mu_i = (\sum_j h_{i,j})/D$	(scalar)
$\sigma_i = (\sum_j (h_{i,j} - \mu_i)^2/D)^{1/2}$	(scalar)
$z_i = (h_i - \mu_i) / \sigma_i$	
$y_i = \gamma * z_i + \beta$	

Ba et al, 2016

Layer normalization normalizes all vectors

Residual connection

All vectors interact through (multiheaded) Self-Attention



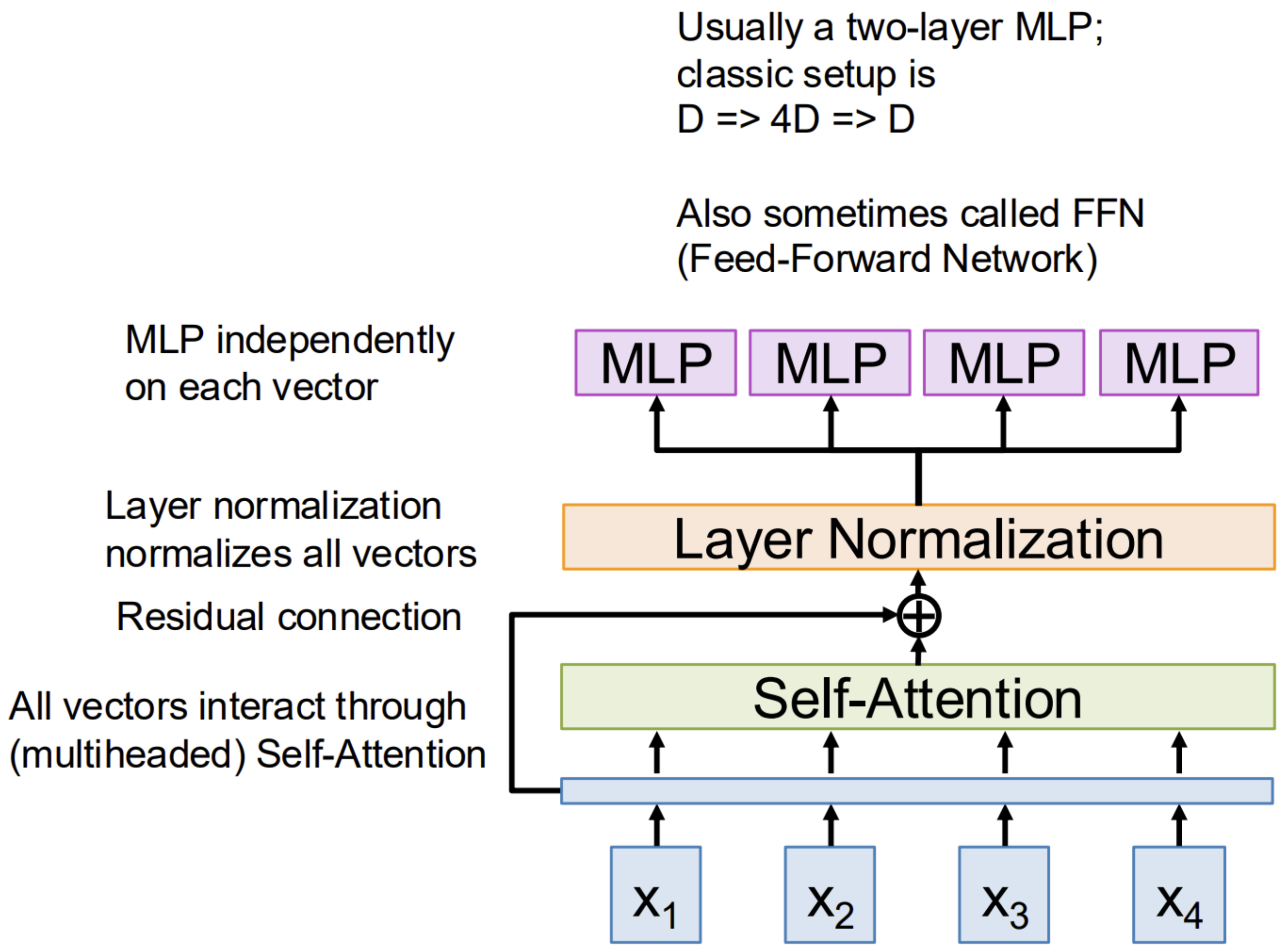
The Transformer Block (2)

- Transformer Block
  - Input: Set of vectors X

TikTok Interview Question:

Can we replace FFN to Self-Attention?

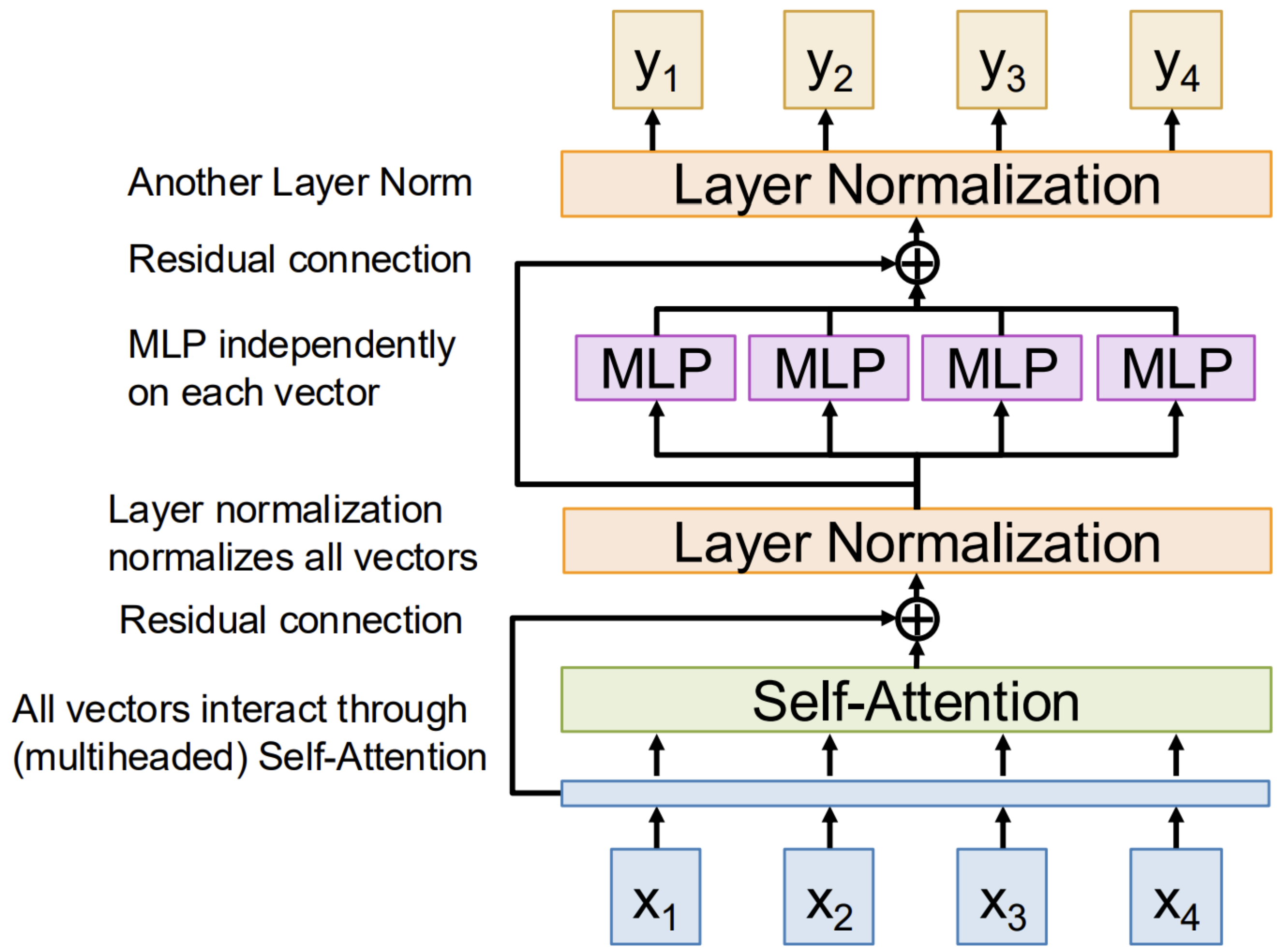
Check the paper: **Attention is NOT all you need: pure attention loses rank…** https://proceedings.mlr.press/v139/dong21a/dong21a.pdf


Usually a two-layer MLP; classic setup is D => 4D => D
Also sometimes called FFN (Feed-Forward Network)
MLP independently on each vector
MLP
MLP
MLP
MLP
Layer normalization normalizes all vectors
Layer Normalization
Residual connection
All vectors interact through (multiheaded) Self-Attention
Self-Attention
x_1
x_2
x_3
x_4

The Transformer Block (3)

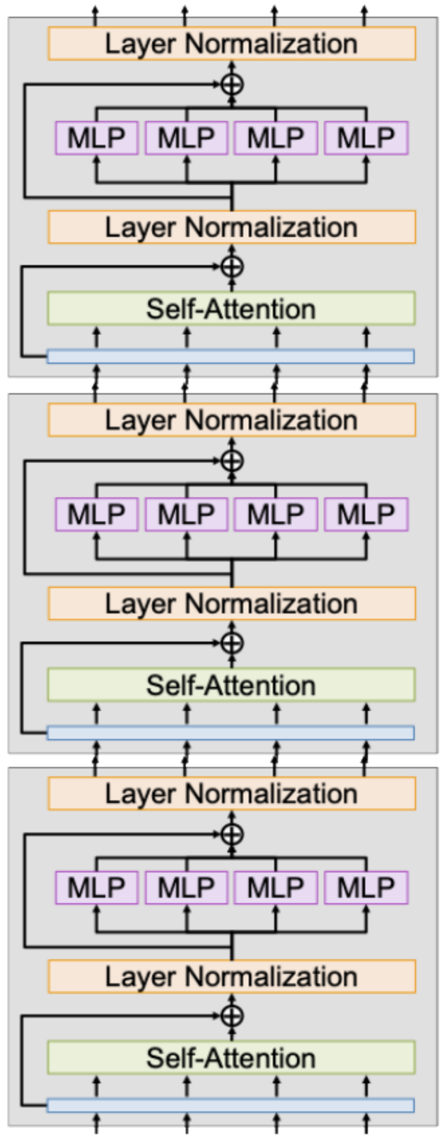
- Transformer Block
  - Input: Set of vectors X
  - Output: Set of vectors Y

- Self-Attention is the only interaction between vectors
- LayerNorm and MLP work on each vector independently

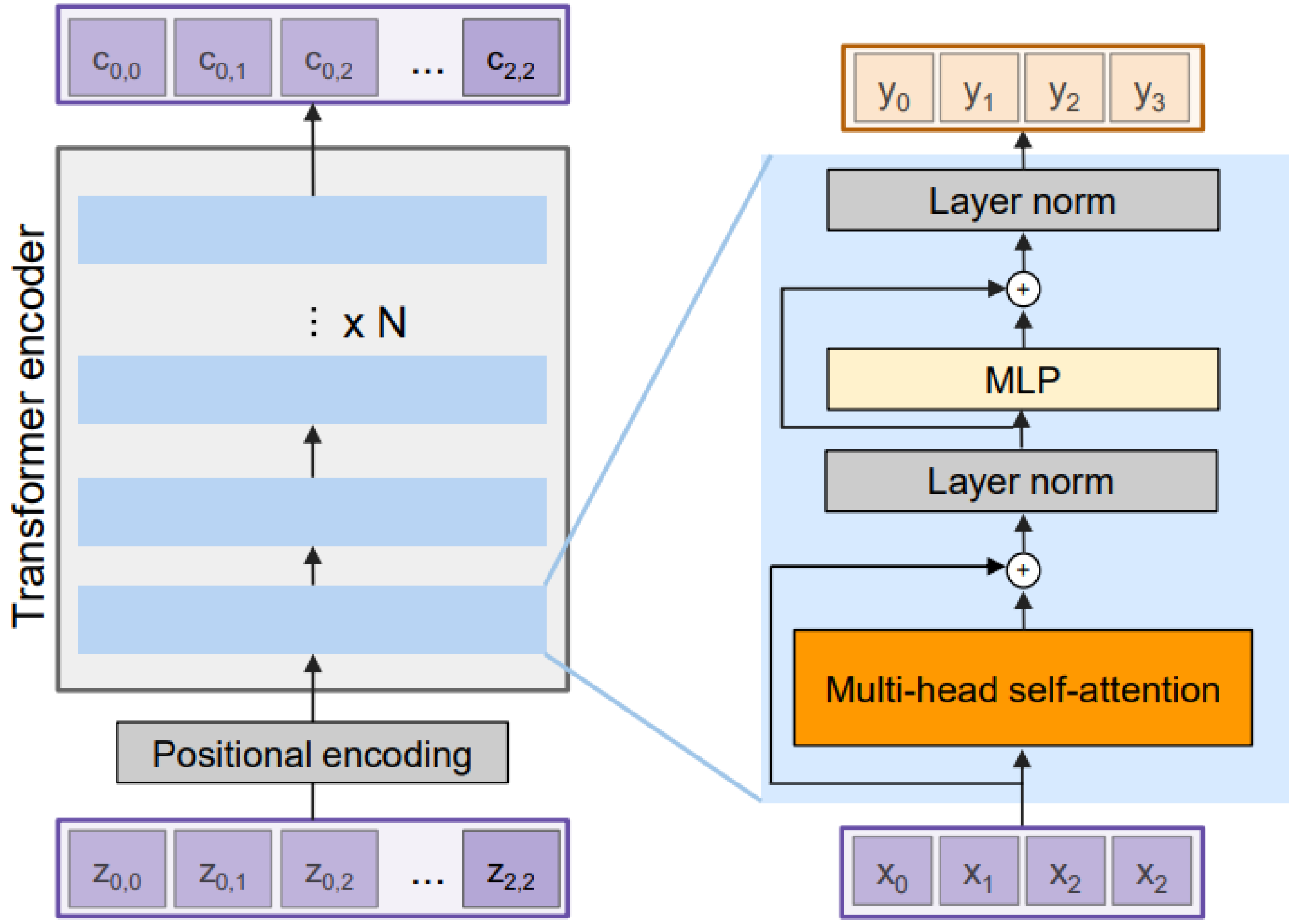


The Transformer (4)

- A Transformer is just a stack of identical Transformer blocks!
- They have not changed much since 2017… but have gotten a lot bigger!
- Original:12 blocks, D=1024, H=16, N=512
  213M params
- GPT-3: 96 blocks, D=12288, H=96, N=2048
  175B params



The Transformer encoder block



Transformer Encoder Block:

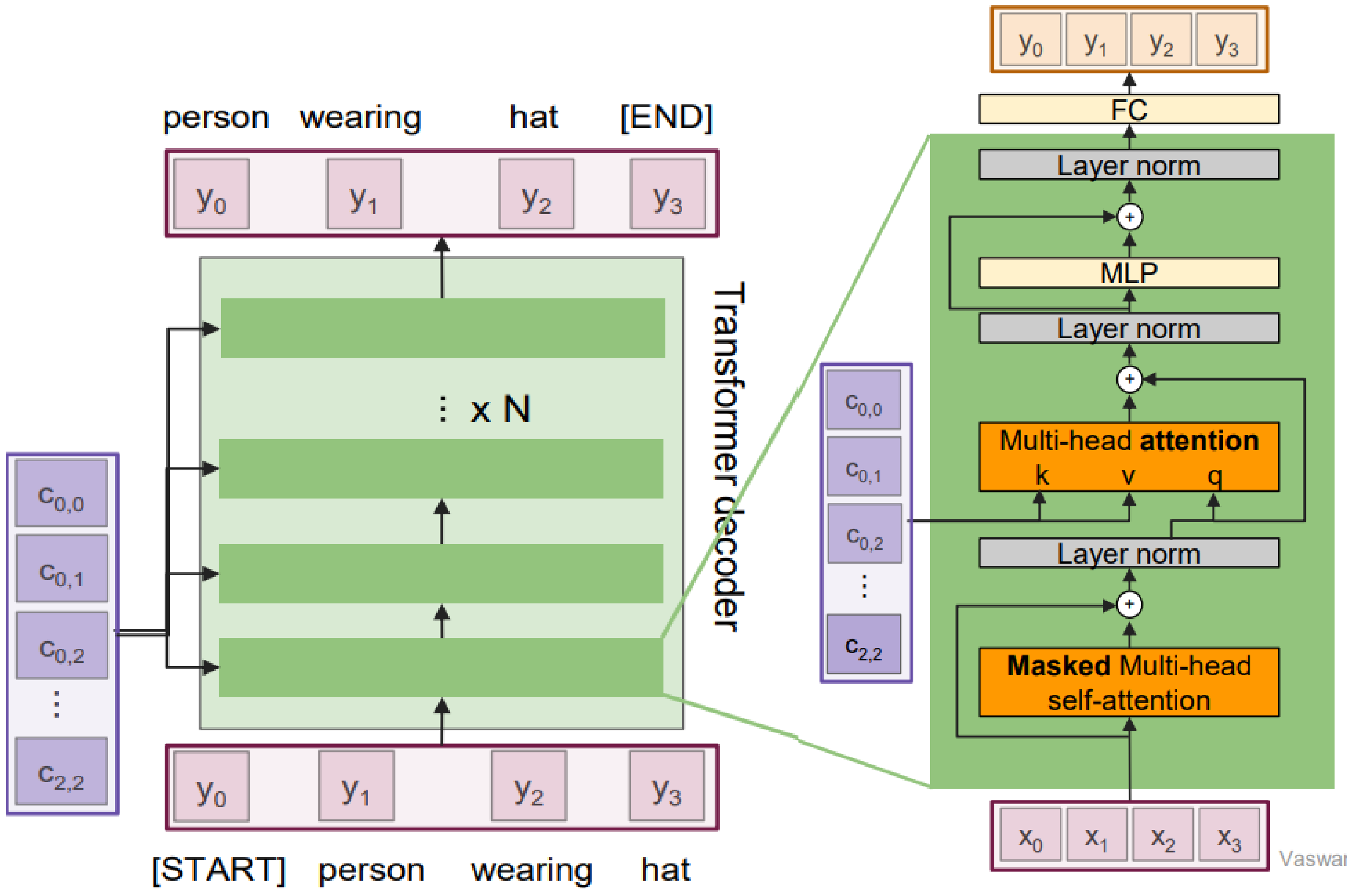
Inputs: Set of vectors **x**
Outputs: Set of vectors **y**

Self-attention is the only interaction between vectors.

Layer norm and MLP operate independently per vector.

Highly scalable, highly parallelizable, but high memory usage.

The Transformer decoder block



Transformer Decoder Block:

Inputs: Set of vectors **x** and Set of context vectors **c.**
Outputs: Set of vectors **y.**

Masked Self-attention only interacts with past inputs.

Multi-head attention block is NOT self-attention. It attends over encoder outputs.

Highly scalable, highly parallelizable, but high memory usage.

Vaswani et al, "Attention is all you need", NeurIPS 2017

Thinking these questions!

- **Why does the Transformer scale the attention score before softmax?**
- **Why using mask attention?**
- **Why does the Transformer use positional encoding?**
- **Why using residual connections in Transformer?**
- **Why do we use LayerNorm?**
- **What is Q, K and V? What is their dimension?**
- **What are the differences between Transformer Encoder and Decoder?**
- **What are the aspects of Transformer parallelization?**

References

- https://cs231n.stanford.edu/slides/2025/lecture_8.pdf
- https://huggingface.co/learn/llm-course/chapter1/4