



# Accelerator Architectures for Machine Learning (AAML)

## Lecture 2: Large Language Model (LLM)

Tsung Tai Yeh

Department of Computer Science  
National Yang-Ming Chiao Tung University



# Acknowledgements and Disclaimer

- Slides was developed in the reference with  
Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, ISCA 2019  
tutorial  
Efficient Processing of Deep Neural Network, Vivienne Sze, Yu-Hsin  
Chen, Tien-Ju Yang, Joel Emer, Morgan and Claypool Publisher, 2020  
Yakun Sophia Shao, EE290-2: Hardware for Machine Learning, UC  
Berkeley, 2020  
CS231n Convolutional Neural Networks for Visual Recognition,  
Stanford University, 2020



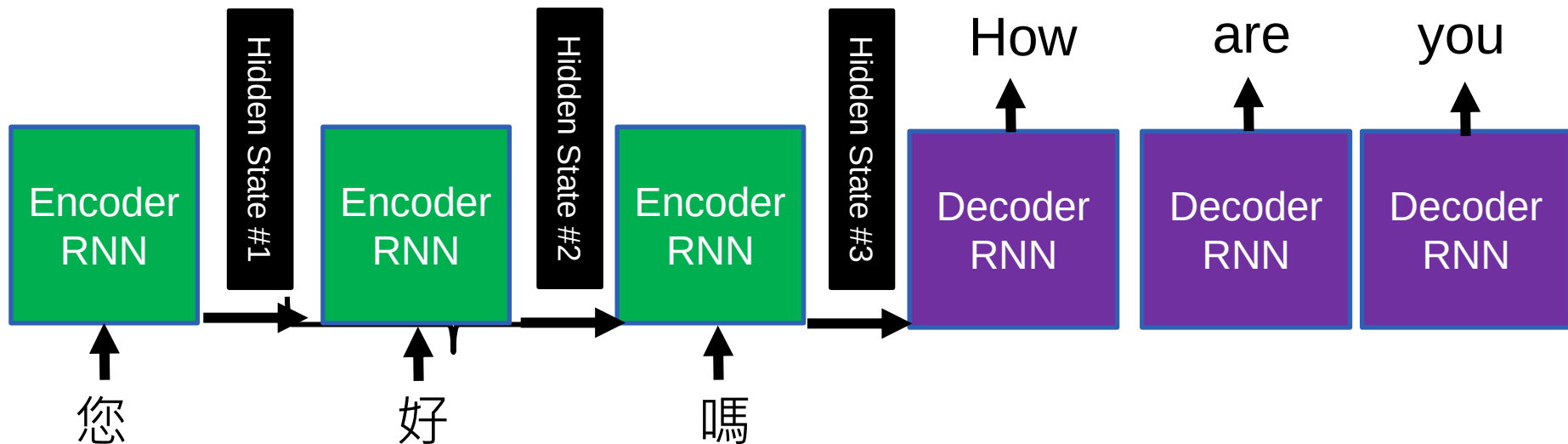
# Outline

- Transformer Model Architecture
  - Encoder-decoder based model
  - Large Language Models (LLMs)
  - FlashAttention vs. PageAttention



# Classical Sequence-to-Sequence Model

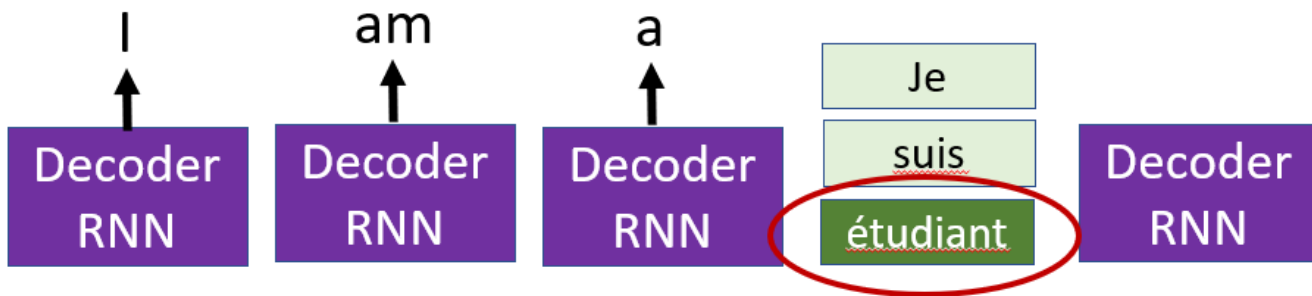
- Pass the last hidden state of the encoding stage
- Decoder uses this last hidden state to do the prediction





# Bottleneck of Sequence-to-Sequence Model

- It is challenging for the model to deal with long sentences
- **Attention**
  - The encoder passes all the hidden states to the decoder
  - The attention enables the decoder to focus on the word before it generates the English translation
  - This ability amplifies the signal from the relevant part of the input sentence

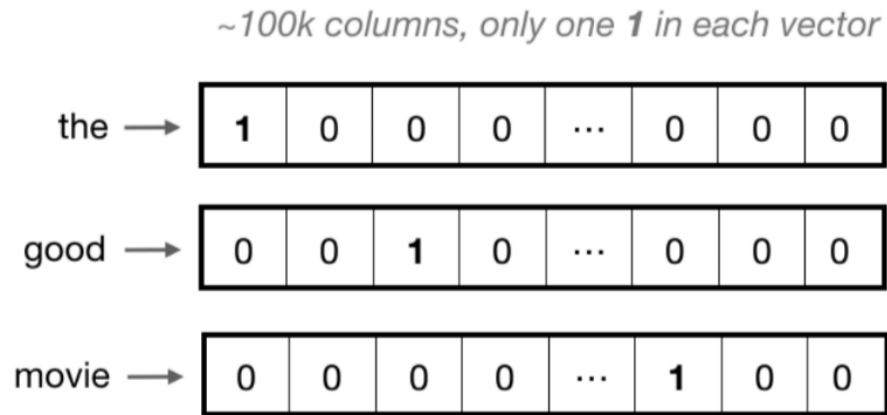




# Word Representation

- **One-Hot Encoding**

- Representing each word as a vector that has as many values in it
- Each column in a vector is one possible word in a vocabulary
- Problem
  - In large vocabularies, these vectors can get very long
  - Contain all 0's except for one value
  - Sparse representation

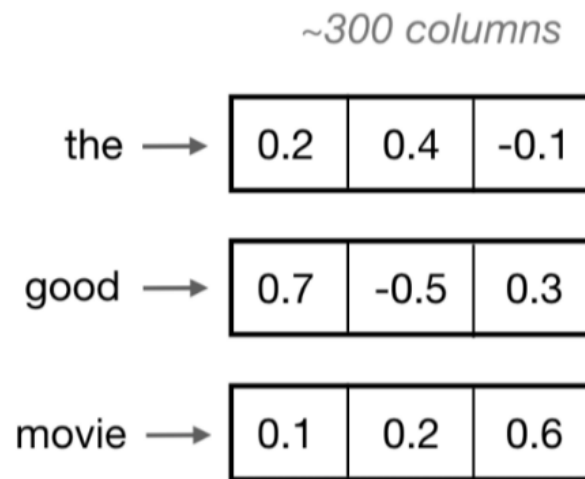




# Word Representation

- **Word Embedding**

- Map the word index to a continuous word embedding through a look-up table
- Popular pre-trained word embeddings
  - Word2Vec, GloVe





# Positional Encoding (PE)

- **Positional encoding (PE)**

- Information to each word about its position in the sentence
- **Unique** encoding for each word's position in a sentence
- Distance between any two positions is **consistent** across sentences with different lengths
- Encode words by using **sin()**, **cos()** with different frequencies
- **Deterministic** and **generalize** to longer sentences

$$\vec{p}_t^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k \cdot t), & \text{if } i = 2k \\ \cos(\omega_k \cdot t), & \text{if } i = 2k + 1 \end{cases} \quad \omega_k = \frac{1}{10000^{2k/d}}$$

- Arguments

- $L$ : maximum # of possible positions
- $d_{\text{model}}$ : dimension of the embeddings
- $n$ : can be set to any value
- $k$ : position
- $i$ : dimension

$$\text{pos}(\mathbf{k}) = \begin{bmatrix} \sin \omega_1 \cdot \mathbf{k} \\ \cos \omega_1 \cdot \mathbf{k} \\ \sin \omega_2 \cdot \mathbf{k} \\ \cos \omega_2 \cdot \mathbf{k} \\ \vdots \\ \sin \omega_{d_{\text{model}}/2} \cdot \mathbf{k} \\ \cos \omega_{d_{\text{model}}/2} \cdot \mathbf{k} \end{bmatrix}_{d_{\text{model}}}, \text{ where } \omega_i = 10000^{\frac{2i}{d_{\text{model}}}}$$

- For each  $k = 0$  to  $L - 1$ 
  - For each  $i = 0$  to  $\frac{d_{model}}{2}$ 
    - $PE_{(k,2i)} = \sin(\frac{k}{2i})$
    - $PE_{(k,2i+1)} = \cos(\frac{k}{2i})$

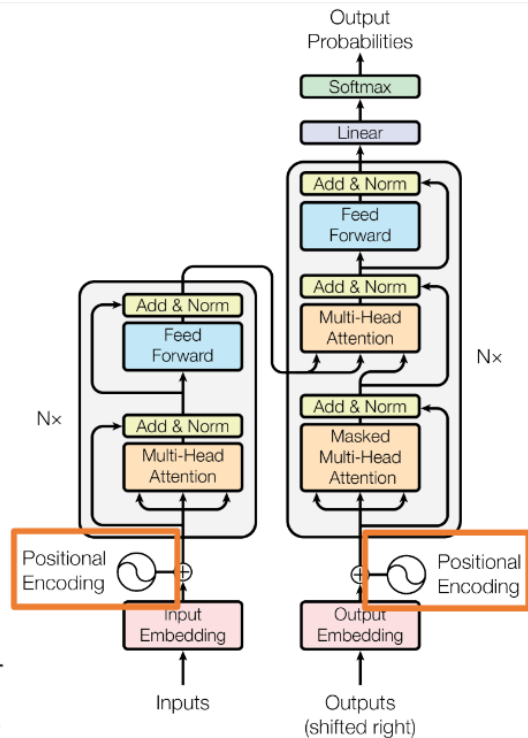


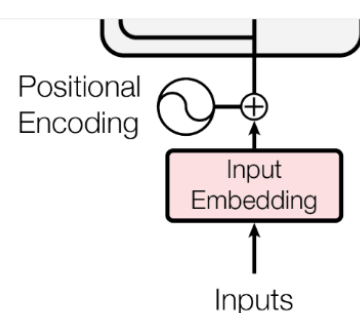
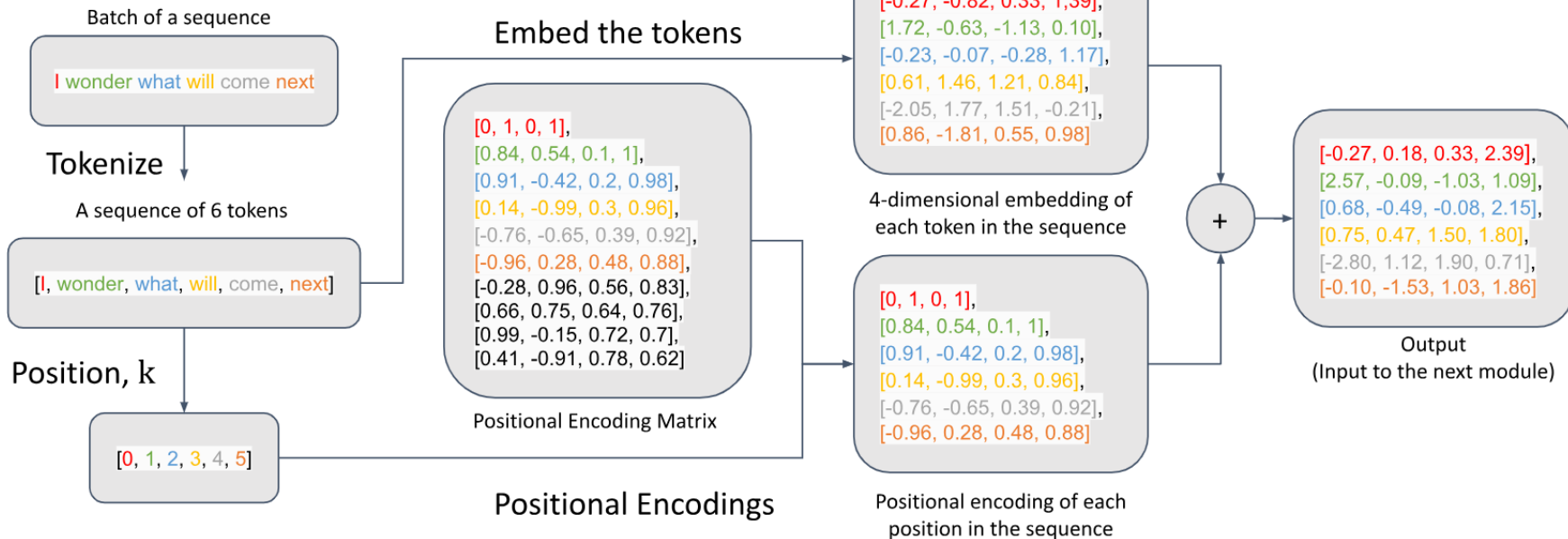
Figure 1: The Transformer - model architecture.



# Positional Encoding (PE)

- Arguments

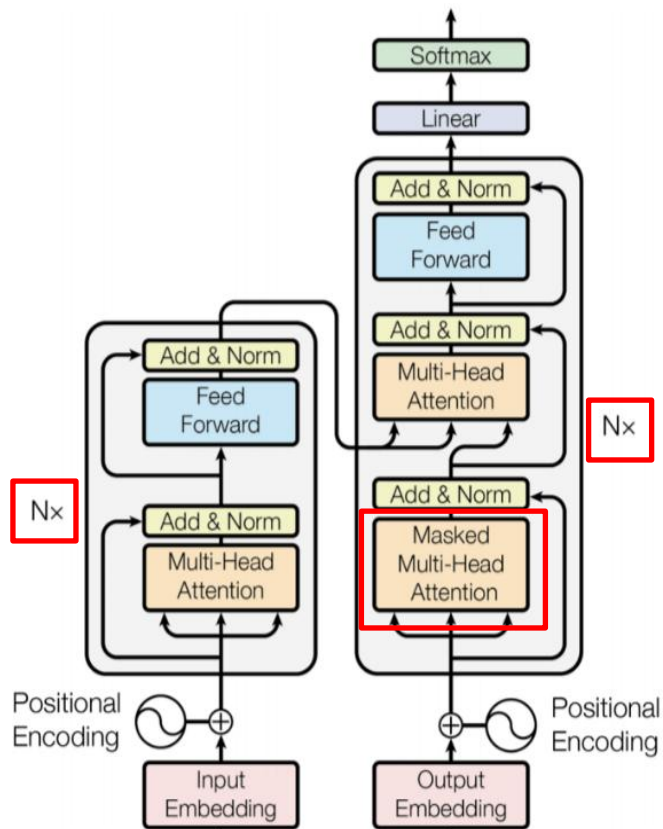
- $L=10, n=100, d_{\text{model}}=4$





# Transformer Model Architecture

- Each **encoder** block has two sub-layers
  - Multi-head self-attention
  - A position-wise fully connected feed-forward
- Each **decoder** block has an additional third sub-layer
  - The third is a masked multi-head attention over the output of the encoder stack
- A **residual connection** is added around each of the two sub-layers
- The decoder yields the output sequence of symbols one element at a time





# Transformer Model Architecture

- Transformer encoder

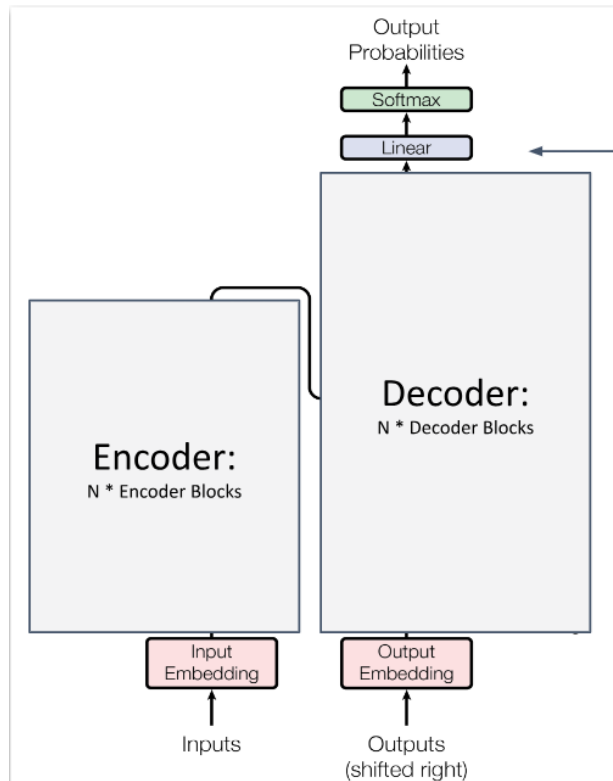
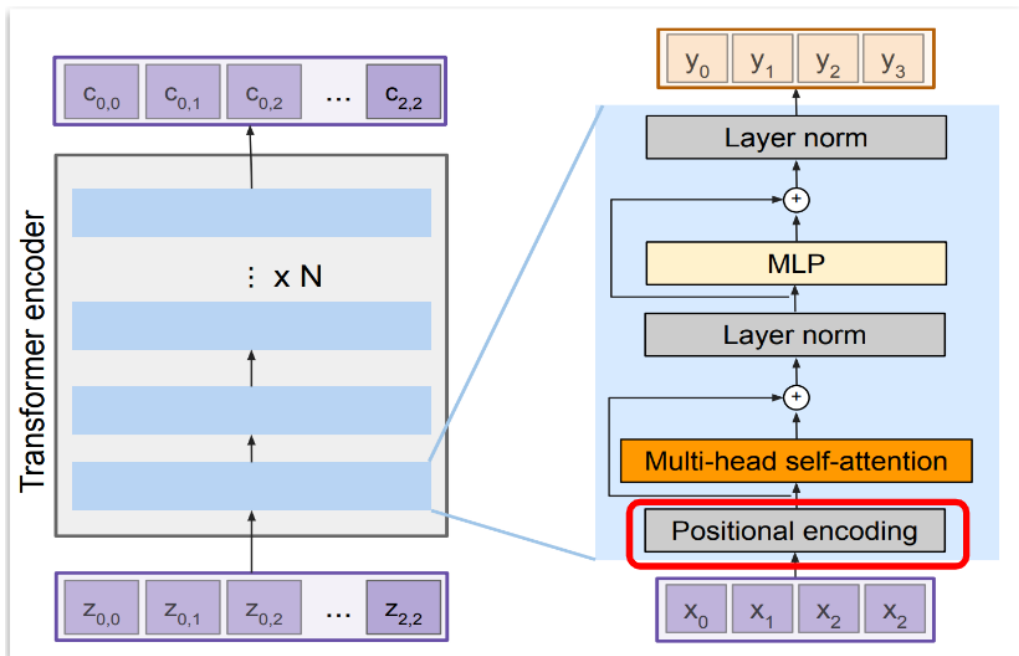


Figure 1: The Transformer - model architecture.

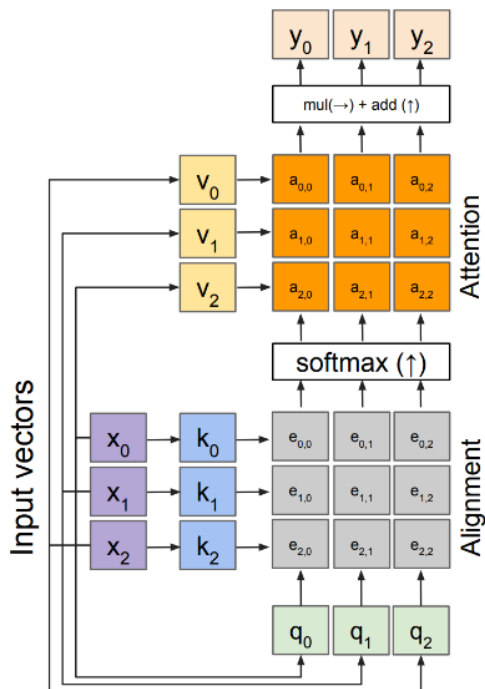


# Transformer Model Architecture

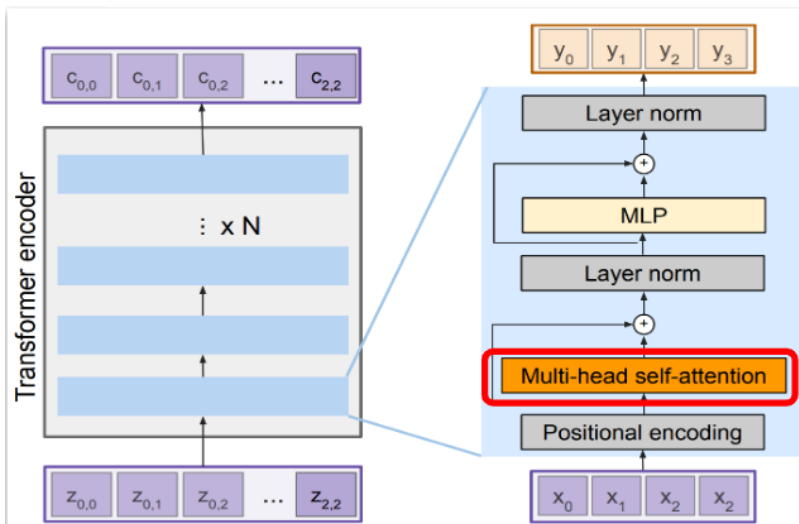
- Encoder – Multi-head self-attention

- $Q = X \cdot W^Q$
- $K = X \cdot W^K$
- $V = X \cdot W^V$

Q: query  
K: key  
V: value



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

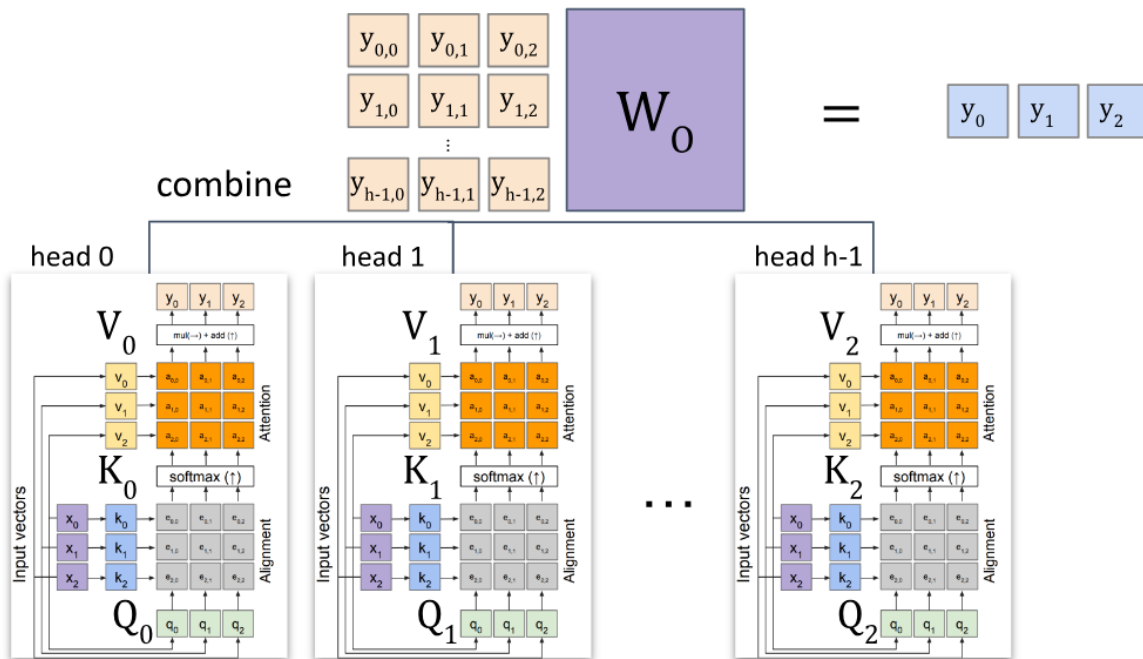




# Transformer Model Architecture

- Encoder – Multi-head self-attention

- $Q_i = X \cdot W^Q_i$
- $K_i = X \cdot W^K_i$
- $V_i = X \cdot W^V_i$



Multi-head Self-attention

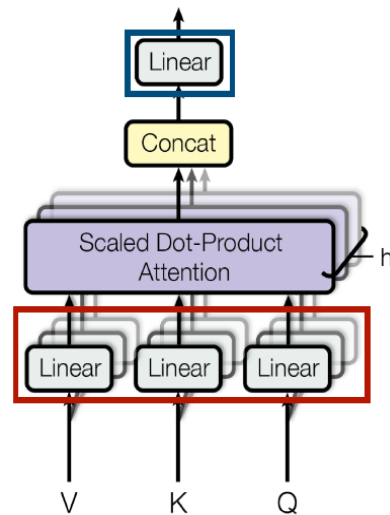


# Multi-Head Self-Attention (MHSA)

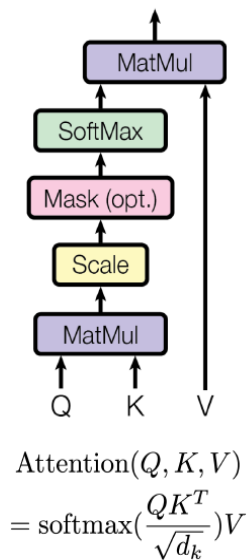
- **Project** Q, K, and V with h **different** learned linear projections
- Perform the **scaled dot-product attention** function on each of Q, K, V **in parallel**
- **Concatenate** the output values
- **Project** the output values again, resulting in the final values

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) W^O$$

where  $\text{head}_i = \text{Attention}(Q W_i^Q, K W_i^K, V W_i^V)$



## Scaled Dot-Product Attention

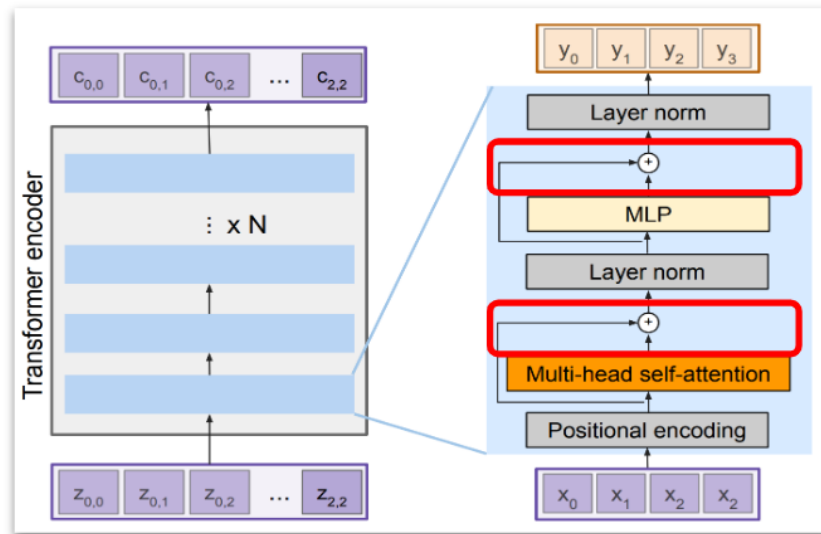




# Transformer Model Architecture

- Encoder – Residual connection
  - Address exploding/vanishing gradients
  - Equation
    - $F(x)$ : output of previous layer
    - $x$ : input of previous layer

$$y = x + F(x)$$





# Transformer Model Architecture

- Encoder – Layer normalization
  - For each input vector  $x = (x_1 .. X_m)$ 
    - Calculate Mean, variance
    - Normalize the vector  $x$ 
      - Small  $\epsilon$  -> avoid dividing 0
    - Scale and shift

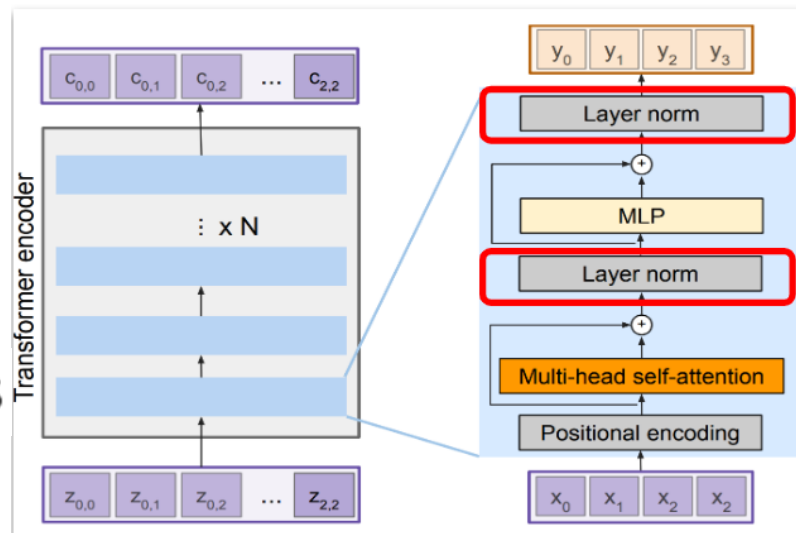
$\gamma, \beta$  are learnable parameters

$$\mu = \frac{\sum_i x_i}{m}$$

$$\sigma^2 = \frac{\sum_i (x_i - \mu)^2}{m}$$

$$\hat{x}_i = \frac{(x_i - \mu)}{\sqrt{\sigma^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$





# Transformer Model Architecture

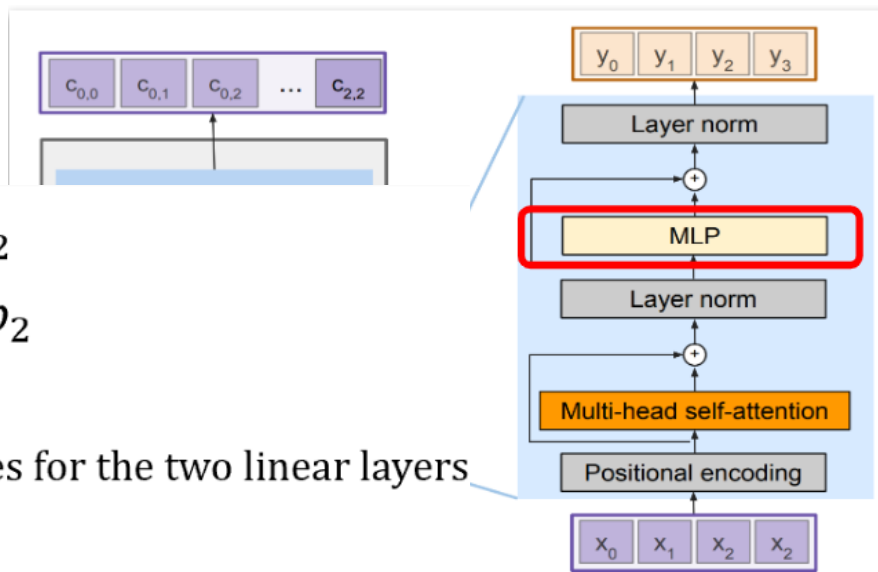
- Encoder – Feed Forward Network (FFN)
  - Contains 2 linear transformations and 1 ReLU

$$\begin{aligned} FFN(x) &= ReLU(W_1x + b_1)W_2 + b_2 \\ &= \max(0, W_1x + b_1)W_2 + b_2 \end{aligned}$$

$x$ : output of the previous layer

$W_{1(d_{model} \times d_{ff})}, W_{2(d_{ff} \times d_{model})}$ : weight matrices for the two linear layers

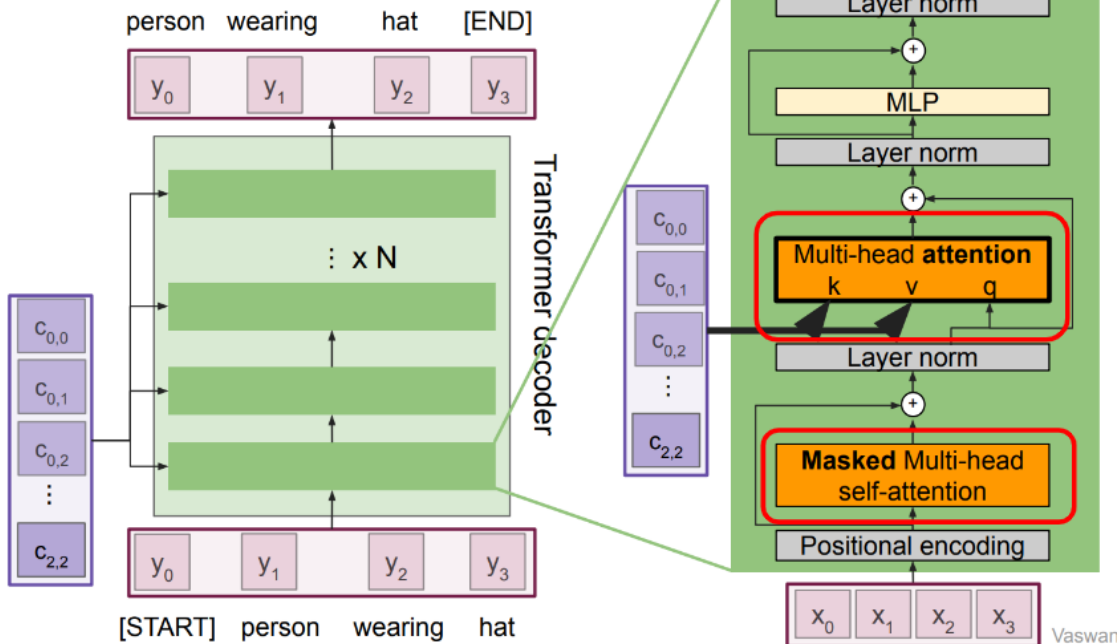
$b_1, b_2$ : biases for the two linear layers





# Transformer Model Architecture

## The Transformer Decoder block





# Transformer Model Architecture

- **Standard attention**

- Each word or token looks at every other word in a sentence simultaneously to calculate relationship scores

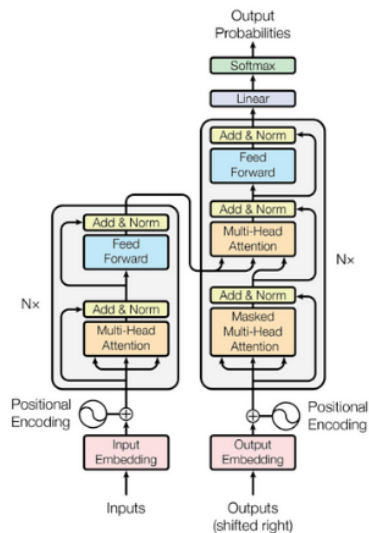
- **Decoder – Mask self-attention**

- Prevents a model from "looking ahead" at future tokens in a sequence
- **The Mask:** A mathematical filter is applied to the attention scores before the softmax
- Without a mask, the model would simply copy the next word instead of learning how to predict it



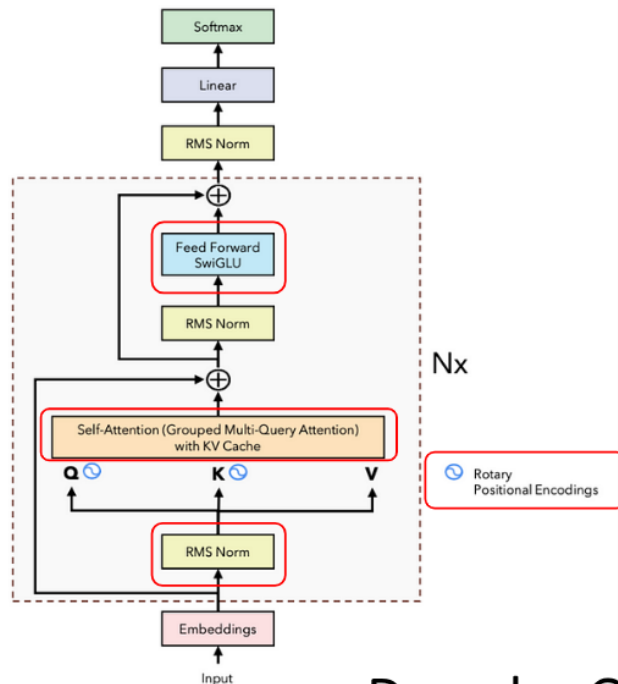
# Large Language Model (LLM) Architecture

## Transformer vs LLaMA



**Transformer**

("Attention is all you need")



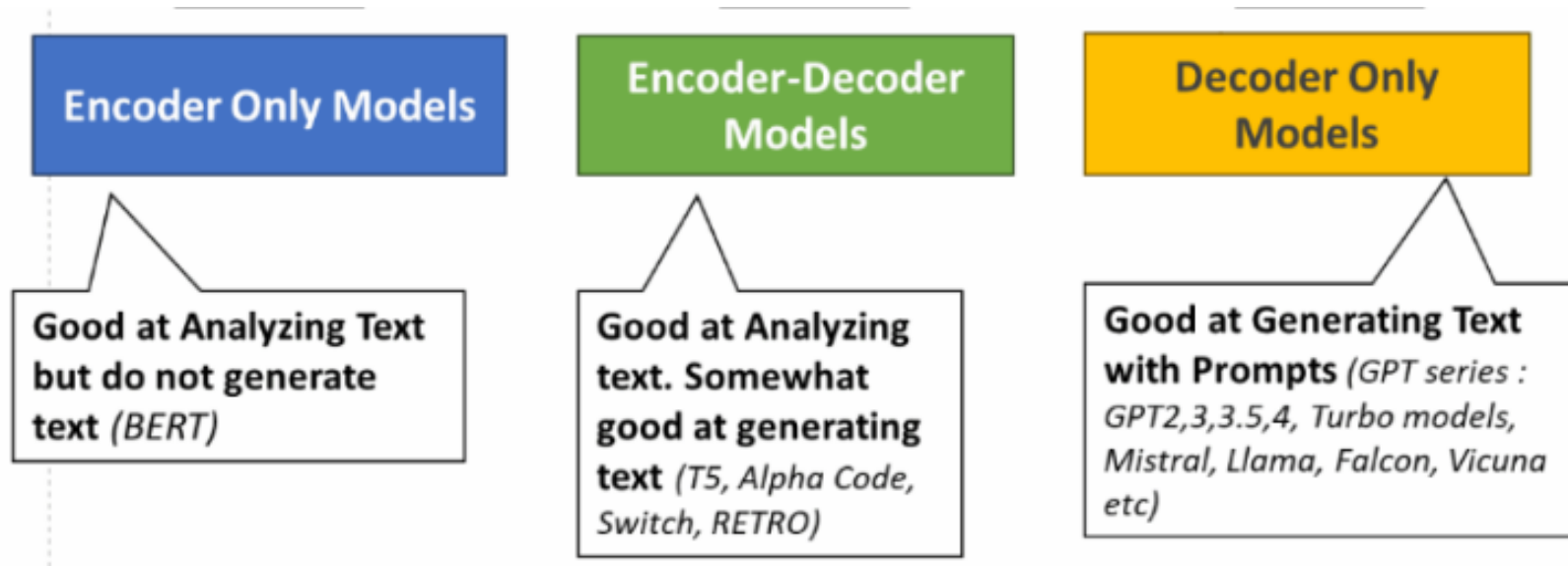
**LLaMA**

Decoder Only!



# Large Language Model (LLM) Architecture

- Why LLaMA is decoder-only model?





# Large Language Model (LLM) Architecture

- Difference between Transformer and LLaMA
  - Decoder-only model
  - Pre-Norm (root mean square (RMS) norm)
  - Rotary positional embedding (RoPE)
  - KV cache
  - Grouped multi query attention
  - SwiGLU activation function rather than ReLU in FFN



# Large Language Model (LLM) Architecture

- Rotary positional embedding (RoPE)
  - Combine absolute and relative encoding
    - Absolute positional embedding
      - Assigns a unique vector to each position and doesn't scale well to capture relative position
    - Relative embeddings
      - Focuses on the distance between tokens
      - Enhance the model's understanding of token relationship
  - Rotational mechanism
    - Each position in the sequence is represented by a rotation in the embedding space

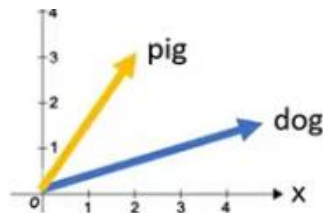


# Large Language Model (LLM) Architecture

- Rotary positional embedding (RoPE)
  - RoPE applies a rotation to the word vector
  - The equation incorporates a rotation matrix that rotates a vector by an angle of  $M_\theta$ , where  $M$  is the absolute position in the sentence.
  - This rotation is applied to the query and key vectors in the self-attention

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

The pig chased the dog





# Large Language Model (LLM) Architecture

- Root Mean Square (RMS) normalization
  - Calculate the RMS of the input vectors rather than the mean and variance
  - Efficient normalization
    - No subtracting mean before squaring
    - No shifting in implementation

$$\text{RMS}(x) = \sqrt{\frac{1}{m} \sum_i x_i^2}$$

$$\hat{x}_i = \frac{x_i}{\text{RMS}(x) + \epsilon}$$

$$y_i = \gamma \hat{x}_i$$

$$\mu = \frac{\sum_i x_i}{m}$$

$$\sigma^2 = \frac{\sum_i (x_i - \mu)^2}{m}$$

$$\hat{x}_i = \frac{(x_i - \mu)}{\sqrt{\sigma^2 + \epsilon}}$$

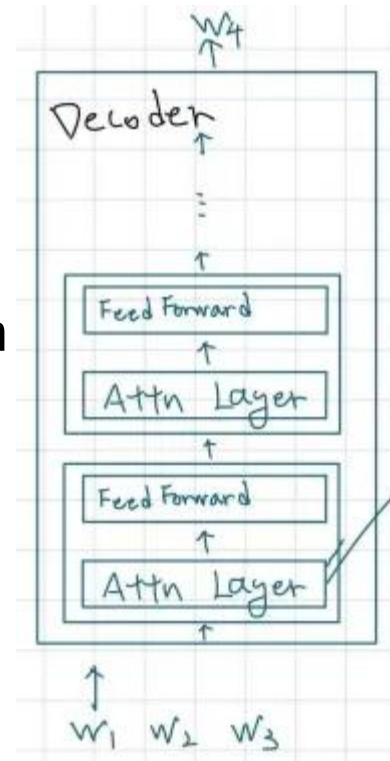
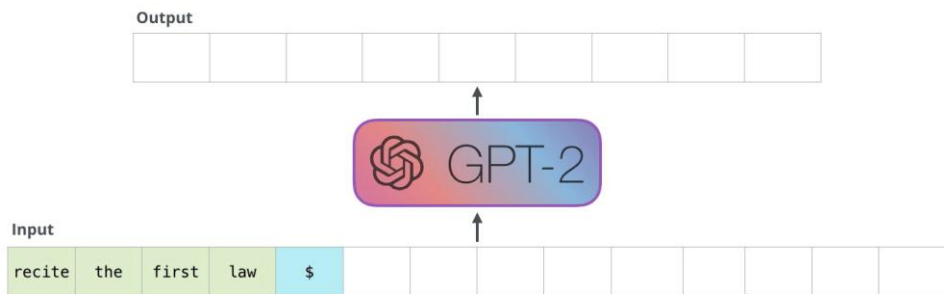
$$y_i = \gamma \hat{x}_i + \beta$$

Layer norm



# Large Language Model (LLM) Architecture

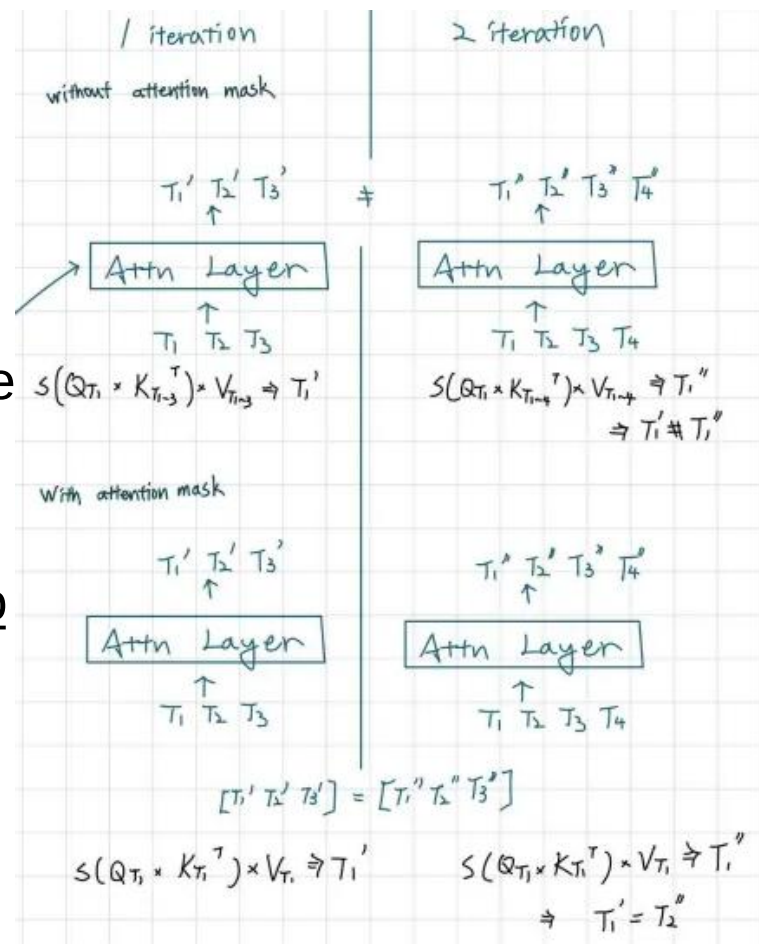
- LLM decoder
  - The decoder works in an auto-regressive fashion
    - Given an input, the model predict the next token
    - Taking the combined input in the next step





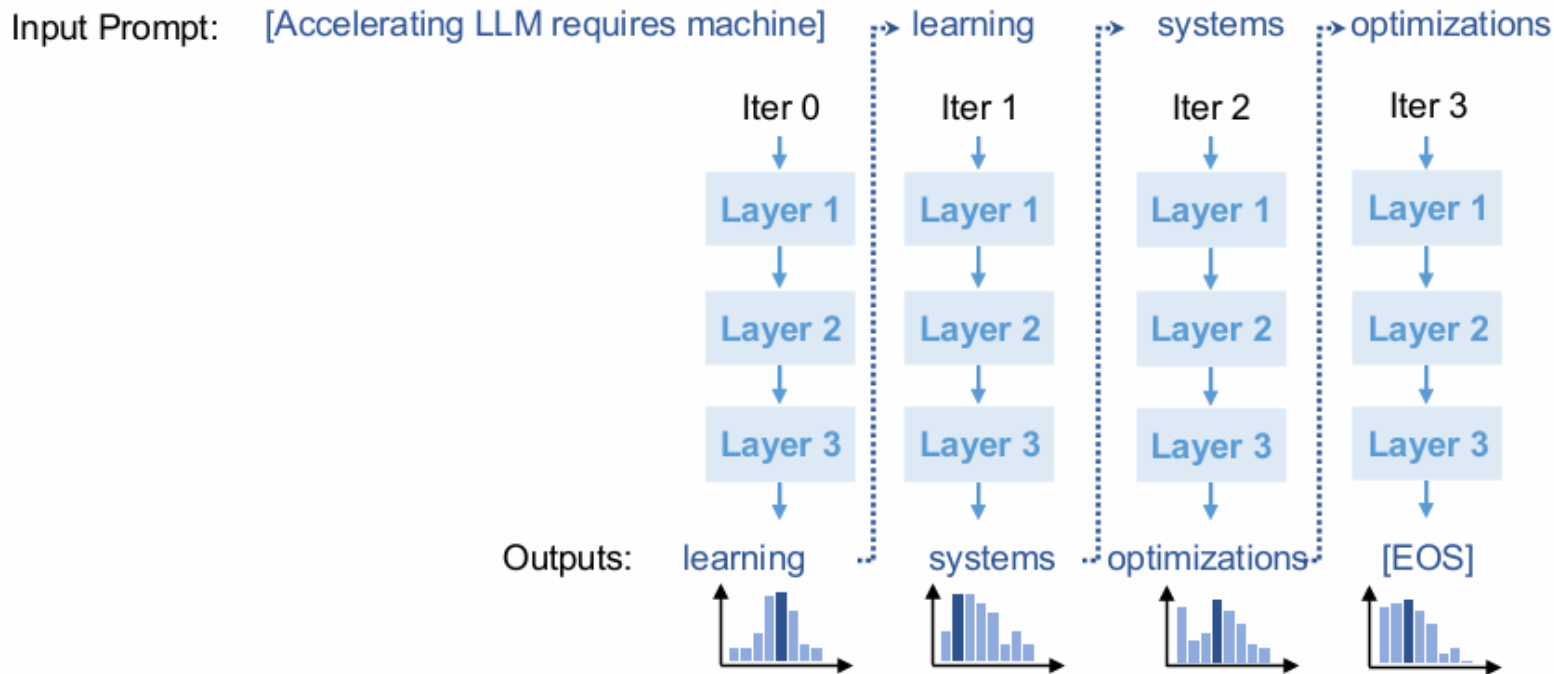
# LLM Architecture

- Sequence mask (in decoder)
  - The decoder cannot see the message in the coming future
  - Use the mask to enable the decoder to only rely on the previous outputs to do the inference → training the decoder



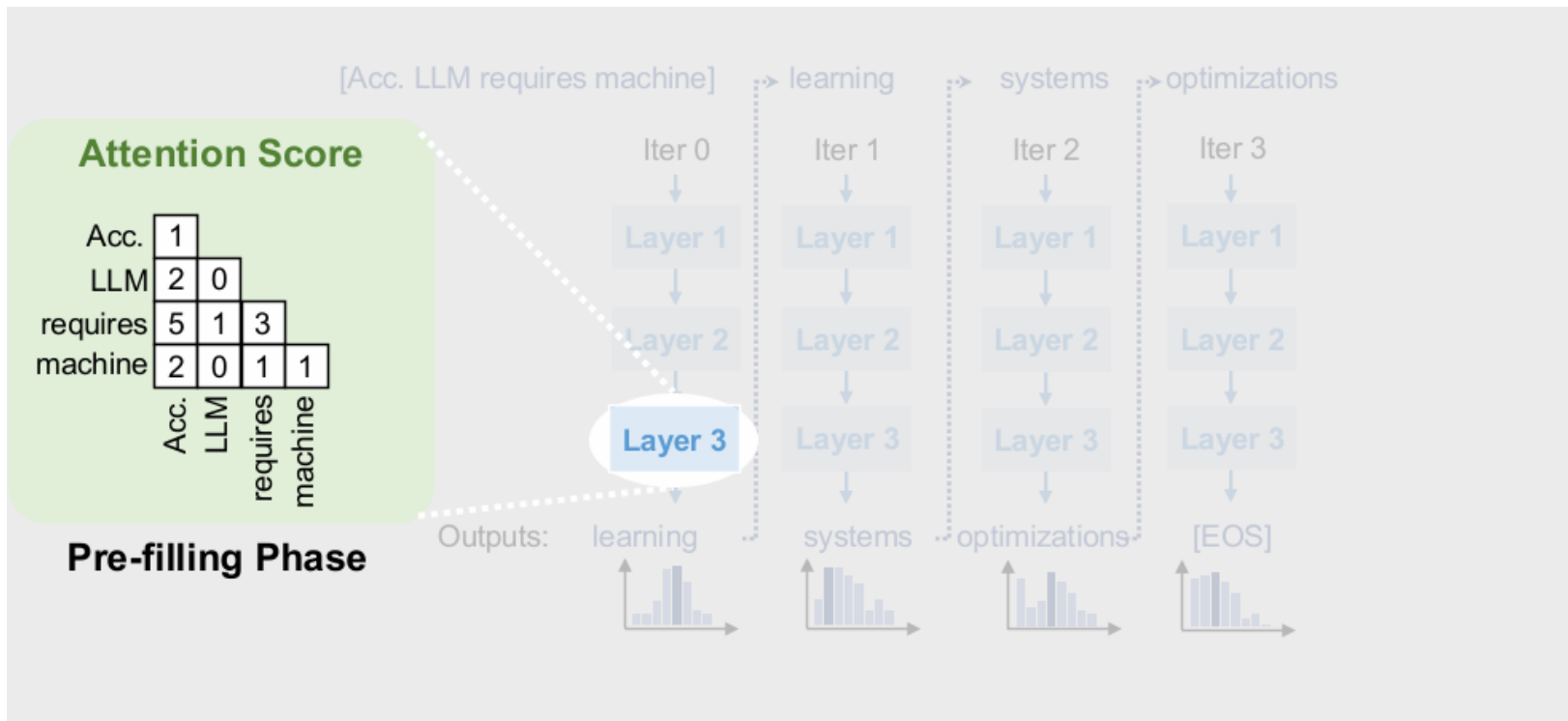


# Generative LLM Inference: Autoregressive Decoding



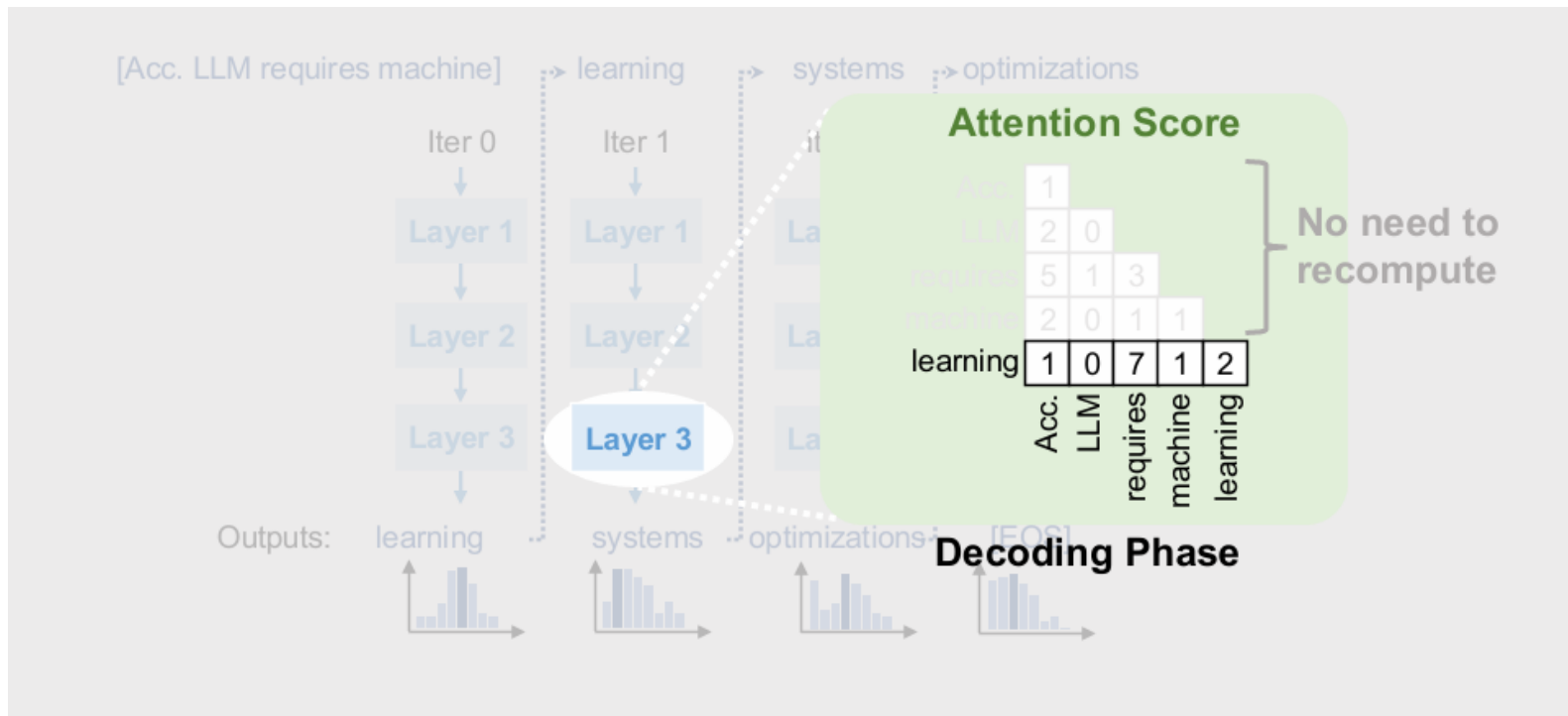


# Generative LLM Inference: Autoregressive Decoding





# Generative LLM Inference: Autoregressive Decoding





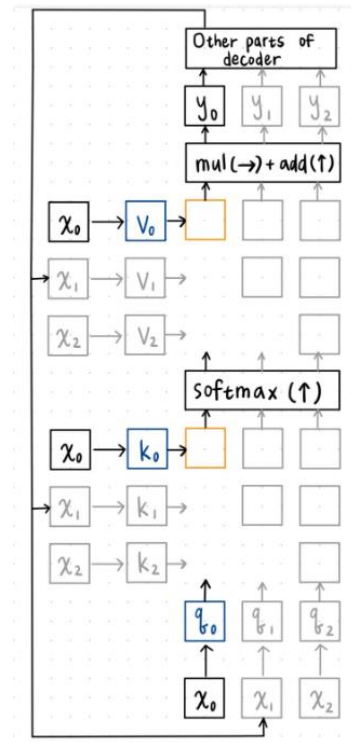
# Generative LLM Inference: Autoregressive Decoding

- **Pre-filling phase (0-th iteration):**
  - Process all input tokens at once
- **Decoding phase (all other iterations):**
  - Process a single token generated from previous iteration
  - Use attention keys & values of all previous tokens
- **Key-value cache**
  - Save attention keys and values for the following iterations to avoid recomputation



# Generative LLM Inference: Autoregressive Decoding

- Key-Value (KV) cache
  - LLM models can generate only one token at a time
  - Each new prediction is dependent on the previous context
    - To predict token number 1000 in the generation, you need information from the previous 999 tokens
  - Optimize the sequential generation process by storing previous calculations to reuse in subsequent tokens, so they don't need to be computed again.





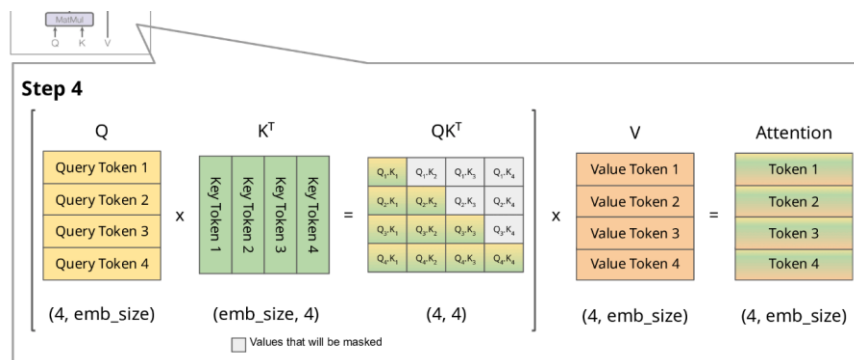
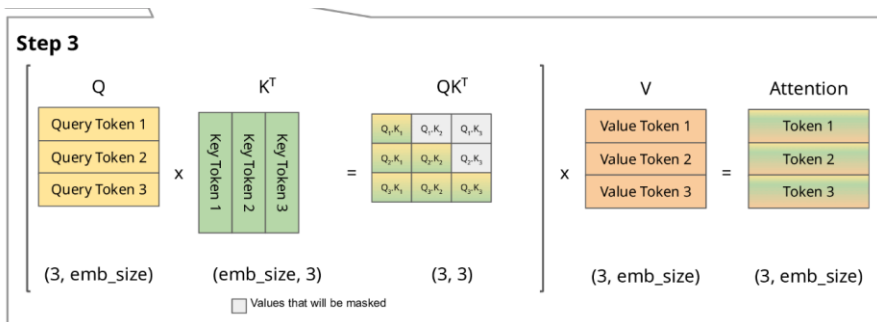
# Generative LLM Inference: Autoregressive Decoding

- Key-Value (KV) cache
  - The matrices obtained with KV caching are way smaller, which leads to faster matrix multiplications
  - The downside of the KV cache is
    - When the length of sequences is becoming long
    - Needs the large memory to cache the Key and Value states



# Generative LLM Inference: Autoregressive Decoding

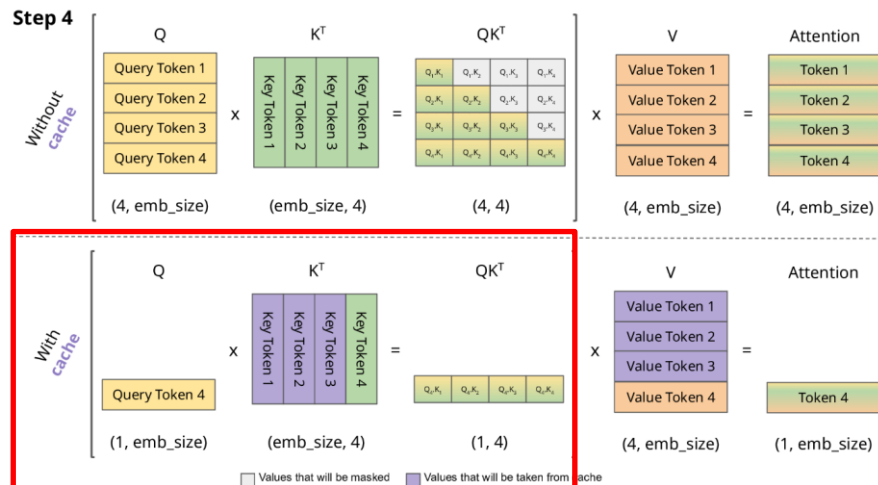
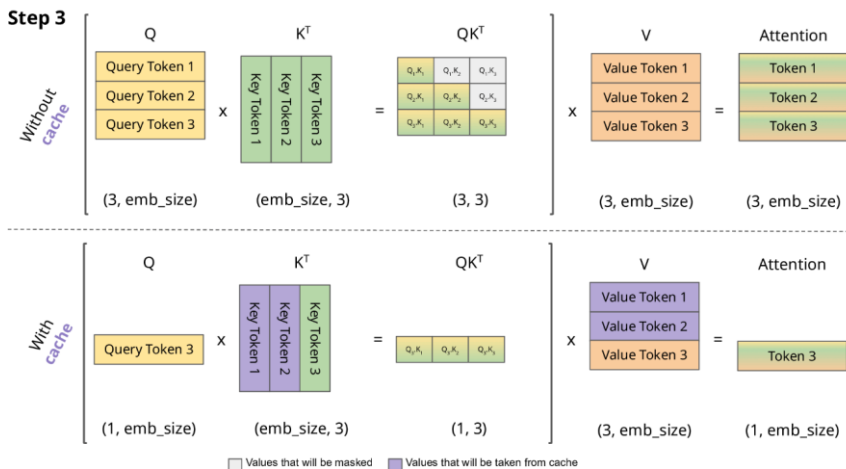
- The scaled dot-product attention
  - The attention of a token only depends on its preceding tokens
  - At each generation step we are recalculating the same previous token attention, when we actually just want to calculate the attention for the new token





# Generative LLM Inference: Autoregressive Decoding

- Key-Value (KV) cache
  - By caching the previous Keys and Values, we can focus on only calculating the attention for the new token.





# Critical Problem of Self-Attention

- **Self-attention problem**

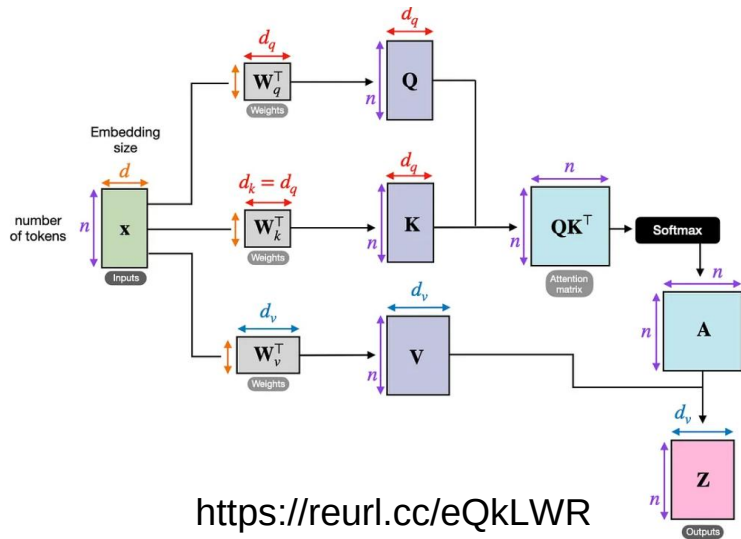
- Scales quadratically with sequence length in both computation and memory usage
- Computes an output for each token in a sequence by attending to all token, requiring an  $O(N^2)$  memory complexity for an input of length  $N$



# Critical Problem of Self-Attention

- **Self-attention problem**

- The key problem is storing each intermediate matrix in the attention calculation during backpropagation
- Quickly exhaust memory for long sequences





# Why we need to save activations in memory?

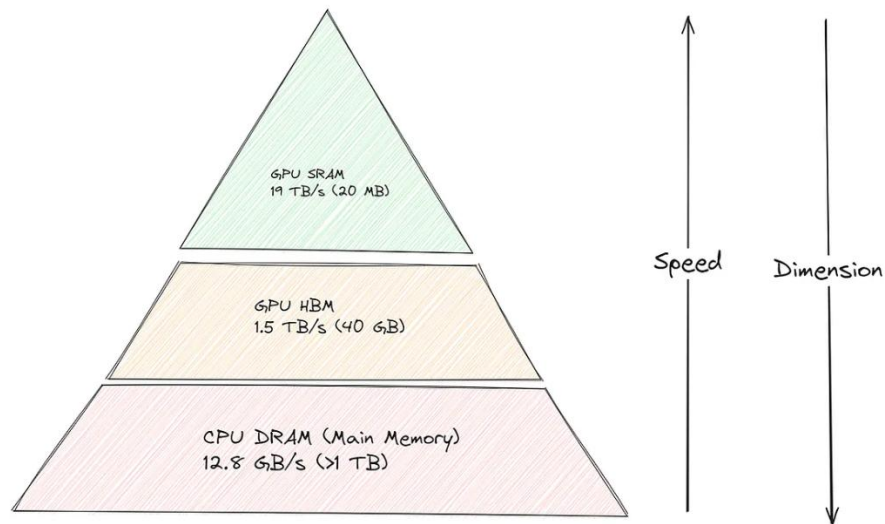
- Neural networks must store intermediate values (activations) for backpropagation
  - **$QK^T$  Matrix**: the raw attention scores before softmax
  - **Softmax outputs (A matrix)**
    - The attention weights before multiplying with Value matrix
  - **Attention Outputs (Z matrix)**
    - The weighted sum of values

```
import torch
def simple_attention(Q, K, V):
    scores = torch.matmul(Q, K.transpose(-2, -1)) #  $QK^T$ 
    weights = torch.nn.functional.softmax(scores, dim=-1) # Softmax( $QK^T$ )
    output = torch.matmul(weights, V) # Weighted sum of V
    return output, weights
```



# GPU Memory Hierarchy

- Every read/write to DRAM comes with serious overhead
  - Load  $Q, K, V \rightarrow$  Compute  $QK^T \rightarrow$  Store attention scores  $\rightarrow$  Softmax  $\rightarrow$  Apply to  $V$   $\rightarrow$  Store results





# FlashAttention

- **FlashAttention**

- Instead of computing the entire  $N \times N$  attention matrix at once
- FlashAttention divides it into smaller tiles or blocks

Standard Attention:

Compute **full**  $QK^T$  matrix ( $N \times N$ )

Store entire matrix **in** HBM/DRAM

FlashAttention:

Divide  $Q, K, V$  **into** blocks **of** size  $B$

Process blocks sequentially **in** fast shared memory

Never materialize the **full**  $N \times N$  matrix



# FlashAttention

- **FlashAttention**

- Tiling and block-sparse attention

|                                   |                                                 |
|-----------------------------------|-------------------------------------------------|
| Sequence length                   | 4096                                            |
| Tile size                         | 64                                              |
| Number of tiles                   | 64 (4096/64)                                    |
| Total attention blocks to process | $64^2 = 4096$                                   |
| Memory savings                    | ~64x (compared to storing full $4096^2$ matrix) |



# FlashAttention

- **Online softmax: computing block by block**
  - Rather than waiting until all logits are in
  - Does a rolling calculation using just the current block

*For each row  $i$ :*

1. Track the current maximum value  $m_i$

2. Track the current sum of exponentials  $\tau_i$

3. For each new block of logits  $x_i^t$

Update max:  $m_i^{new} = \max(m_i, \max(x_i^t))$

Update sum:  $\tau_i^{new} = \tau_i \exp(m_i - m_i^{new}) + \sum \{\exp(x_i^t - m_i^{new})\}$

4. Normalize partial outputs:  $\text{softmax}(x_i^t) = \exp(x_i^t - m_i^{new}) / \tau_i^{new}$



# FlashAttention

- **Online softmax: computing block by block**

- Subtract the maximum value from all logits before exponentiating
  - Prevents overflow

*Block 1: [1.0, 2.0]*

$$max\_1 = 2.0$$

$$sum\_1 = \exp(1.0-2.0) + \exp(2.0-2.0) = 0.368 + 1.0 = 1.368$$

*Block 2: [3.0, 0.5]*

$$max\_2 = \max(2.0, 3.0) = 3.0$$

$$sum\_2 = 1.368 * \exp(2.0-3.0) + \exp(3.0-3.0) + \exp(0.5-3.0) = 0.503 + 1.0 + 0.082 = 1.585$$



# FlashAttention

- **Online softmax: computing block by block**
  - The small difference between offline and online softmax results is due to the numerical stability techniques used in online softmax
  - This changes the individual values slightly but preserves their relative proportions

*With online softmax:  $[exp(1.0-3.0)/1.585, exp(2.0-3.0)/1.585, exp(3.0-3.0)/1.585, exp(0.5-3.0)/1.585] = [0.050, 0.137, 0.631, 0.052]$*

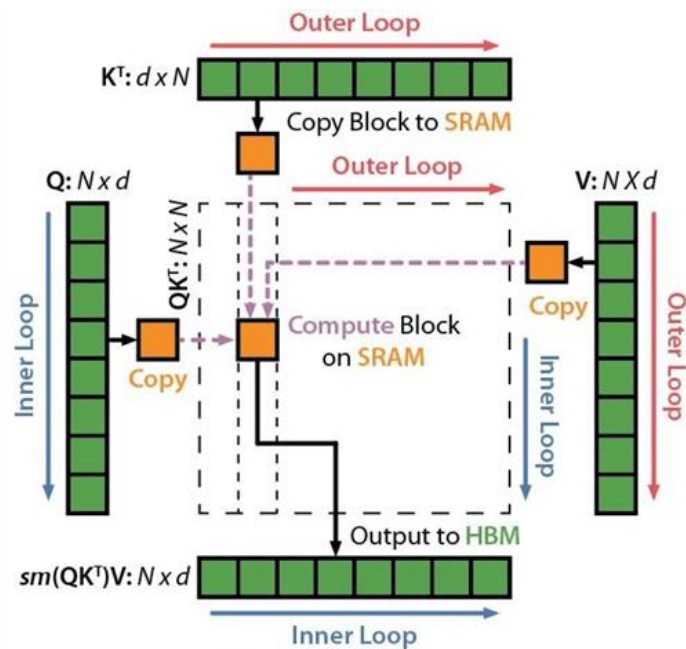
*without online softmax :  $softmax([1.0, 2.0, 3.0, 0.5]) =: [0.085, 0.232, 0.631, 0.052]$*



# FlashAttention

- **FlashAttention**

- Tiling: decompose large softmax into smaller ones by scaling
- 1. Load inputs by blocks from global to share memory
- 2. On chip, compute attention output wrt the block
- 3. Update output in device memory by scaling



**FlashAttention**

<https://reurl.cc/eQkLWR>



# FlashAttention

- **Fused GPU Kernels**

- Merges several GPU operations into one kernel
- No more unnecessary memory shuffling

```
# Standard implementation (multiple kernel launches)
scores = torch.matmul(Q, K.transpose(-2, -1)) # Kernel 1
scores = scores / math.sqrt(d) # Kernel 2
attn = F.softmax(scores, dim=-1) # Kernel 3
output = torch.matmul(attn, V) # Kernel 4
```

```
# FlashAttention (single fused kernel)
output = flash_attention(Q, K, V) # All operations in one kernel!
```



# FlashAttention

- **Activation Checkpointing**

- Instead of storing intermediate results like the attention matrix during forward pass
- FlashAttention recomputes them during backward pass

```
# Standard Backprop:
```

```
Store  $QK^T$ ,  $\text{softmax}(QK^T)$ .. during forward pass
```

```
Use stored values directly during backward pass
```

```
# FlashAttention Backprop:
```

```
Don't store intermediate values
```

```
Recompute  $QK^T$ ,  $\text{softmax}(QK^T)$  during backward pass
```

```
Trade slight increase in computation for massive memory savings
```



# FlashAttention

- **Evolution of FlashAttention**

- **FlashAttention 1 (2022)**

- Introduce the core tiling approach and memory optimization techniques

- **FlashAttention 2 (2023)**

- Better memory access patterns, reduce redundant computation, and improved numerical stability

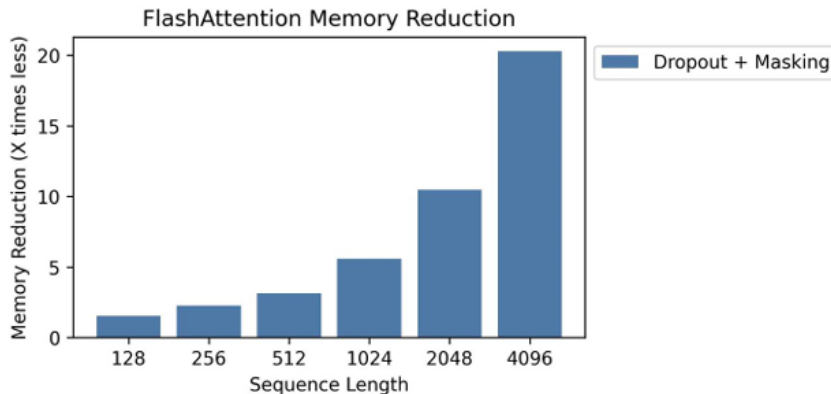
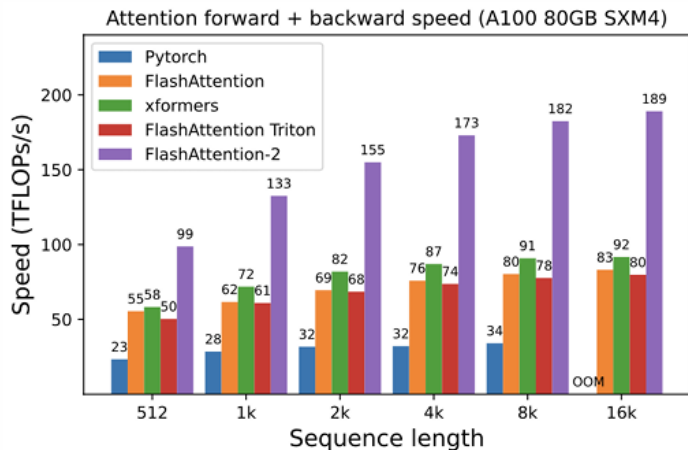
- **FlashAttention 3 (2024)**

- With enhanced parallelization, better memory footprint management



# FlashAttention

- **FlashAttention Performance Analysis**
  - 2-4X speedup
  - 10-20X memory reduction





# PageAttention

- **PageAttention: OS-Inspired memory management**
  - LLM computes and store key (K) and value (V) vectors for all previously generated tokens
  - This KV cache grows linearly with sequence length

*[Token 1]  $\rightarrow$  Compute  $K_1$ ,  $V_1 \rightarrow$  Store in cache*

*[Token 2]  $\rightarrow$  Compute  $K_2$ ,  $V_2 \rightarrow$  Store in cache*

*...*

*[Token N]  $\rightarrow$  Compute  $K_n$ ,  $V_n \rightarrow$  Store in cache*

For token N+1, reuse all cached  $K_1...K_n$  and  $V_1...V_n$  instead of recomputing them.



# PageAttention

- **In traditional implementations**
  - KV caches use contiguous memory blocks
  - Each sequence element allocates a fixed-size memory block
  - Each memory block must be large enough to hold maximum possible sequence length
  - This simple approach leads to significantly memory inefficiencies



# PageAttention

- **Internal Fragmentation**

- When we pre-allocate memory for the maximum possible sequence length (4096 tokens), but only use a portion (512 tokens)



## Internal Fragmentation Example:

*Allocated: 4096 tokens per sequence*

*Actually used: 512 tokens*

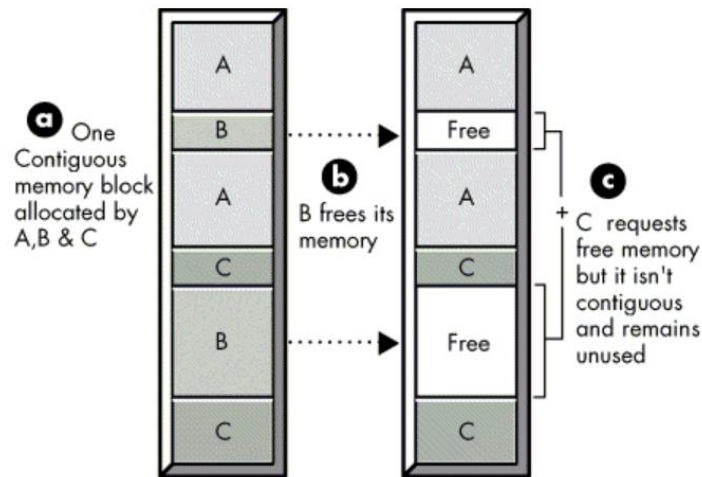
*Wasted memory: 87.5% of allocation!*



# PageAttention

- **External Fragmentation**

- As sequences are created and completed at different times
- Memory becomes fragmented into small, non-contiguous blocks that cannot fit new requests





# PageAttention

- **How PageAttention solves memory fragmentation**
  - **1. Memory paging**
    - Divides KV cache into uniform “pages” (fixed-size blocks)
  - **2. Page Tables**
    - Maintains mappings from logical positions to physical memory pages
  - **3. Non-contiguous storage**
    - Allows storing a sequence’s KV cache across multiple non-contiguous pages



# PageAttention

- **Benefits of paged attention**
  - **Reduce memory waste**
    - Page are allocated precisely as needed
  - **Dynamic memory management**
    - Pages can be recycled when tokens are no longer needed
  - **Efficient resource utilization**
    - Memory can be shared across different sequences
  - **Support for variable-length sequences**
    - No need to pre-allocate for maximum length



# Takeaway Questions

- What's problem the “Attention” aiming to solve?
  - (A) Gradient vanishing
  - (B) Message passing in the long sequence of data
  - (C) Over-fitting
- What are benefits of the “Transformer” ?
  - (A) Large hidden layer
  - (B) The amount of computation is small
  - (C) More data parallelism



# Takeaway Questions

- How does the “self-attention” help the encoder?
  - (A) Looking at other words in the input sentence
  - (B) Memorizing the more messages within a network
  - (C) Focus on a specific word