

CS310 Natural Language Processing

自然语言处理

Lecture 07 - Transformer

Instructor: Yang Xu

主讲人：徐炀

xuyang@sustech.edu.cn

Overview

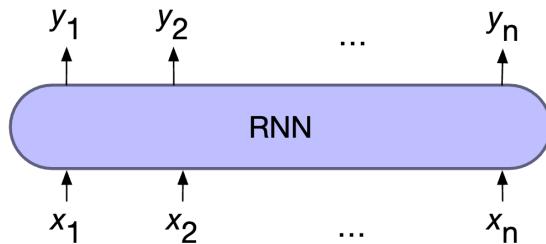
- Motivation
- Transformer Basics
- Transformer Details
- Extensions

Overview

- **Motivation**
 - **RNNs, Encoder-Decoder, and Attention**
- Transformer Basics
- Transformer Details
- Extensions

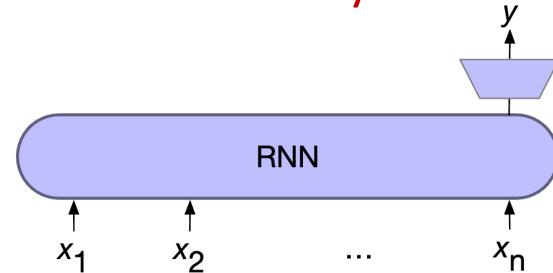
Common RNN Architecture for NLP

POS tagging, NER

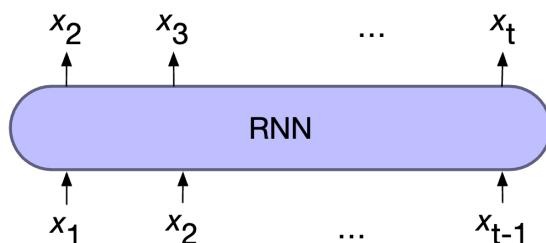


a) sequence labeling

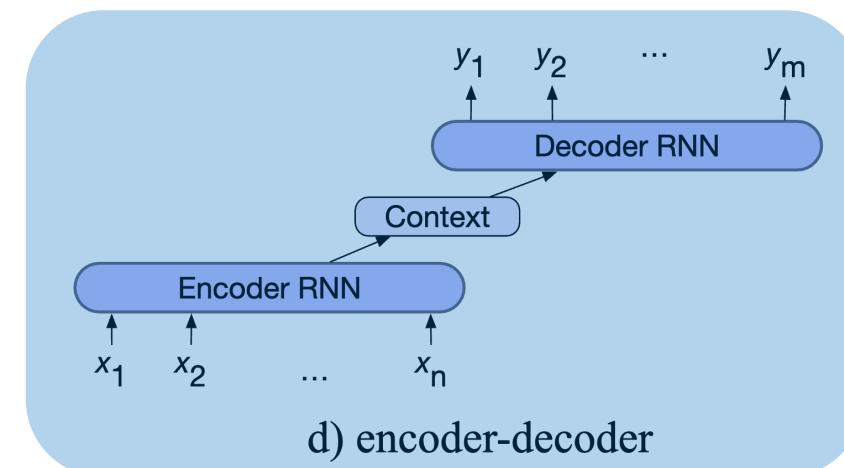
Sentiment analysis



b) sequence classification



c) language modeling



d) encoder-decoder

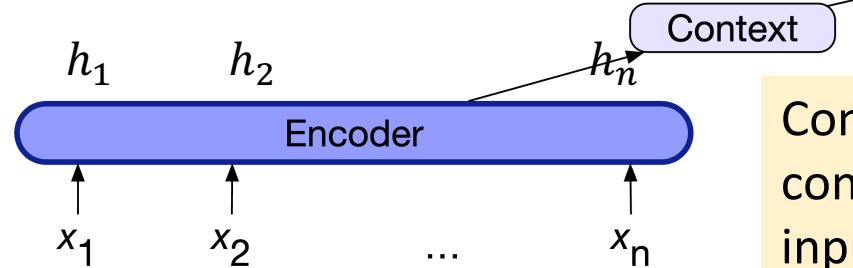
Encoder-decoder: also called **sequence-to-sequence network**

- Take an input sequence and translate it to an output sequence of *different* length
- Applications: summarization, question answering, dialogue, and particularly useful for *machine translation*

Encoder-Decoder

- Three conceptual components: an **encoder**, a **context** vector, and a **decoder**

Encoder accepts an input sequence $x_{1:n}$ and generates a sequence of contextualized representations $\mathbf{h}_{1:n}$



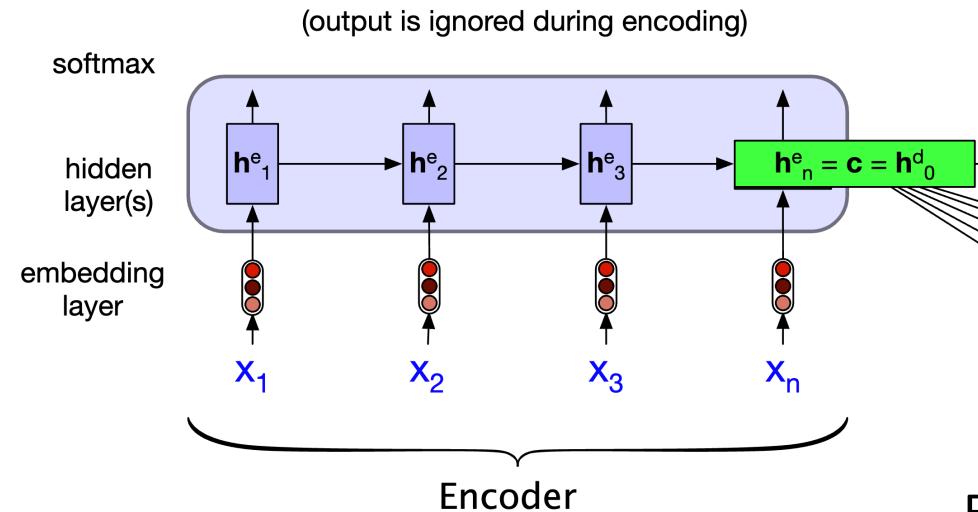
Context vector $\mathbf{c} = f(\mathbf{h}_{1:n})$ conveys the information of input to the decoder



Decoder accepts \mathbf{c} as input and generate a sequence of new representations $\mathbf{h}'_{1:m}$, from which an output sequence $y_{1:m}$ is obtained

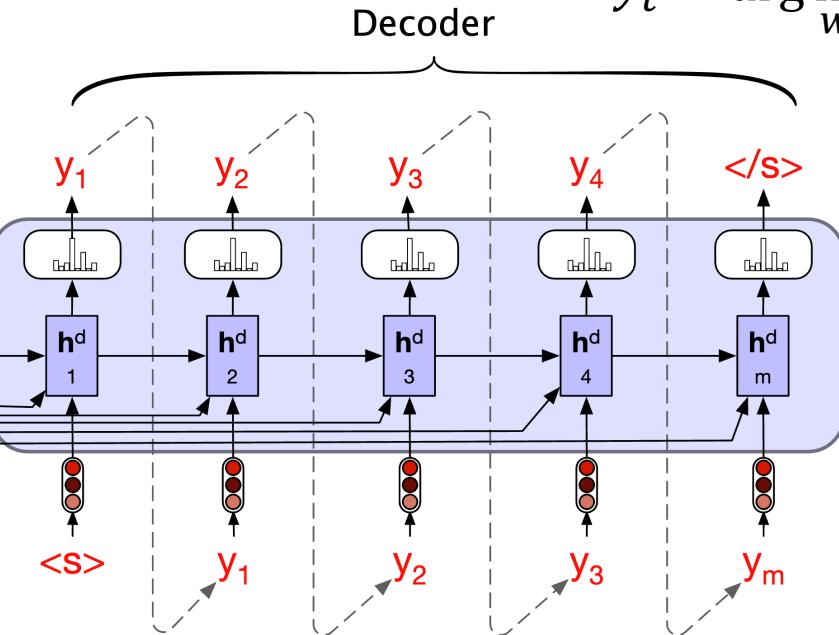
Encoder-Decoder RNN for Machine Translation

Encoder generates a contextualized representation of the input sequence, embodied in the final hidden state $\mathbf{c} = \mathbf{h}_n^e$



$$\mathbf{h}_t^d = g(y_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \quad \hat{y}_t = \text{softmax}(f(\mathbf{h}_t^d))$$

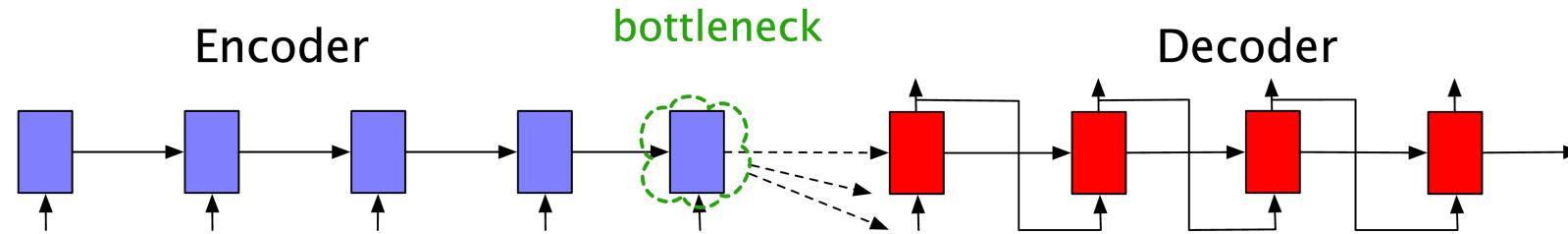
$$y_t = \arg \max_{w \in V} \hat{y}_t$$



First decoder RNN cell uses \mathbf{c} as its initial hidden state \mathbf{h}_0^d
 \mathbf{c} is made available to all time steps to preserve its influence

Attention in Encoder-Decoder

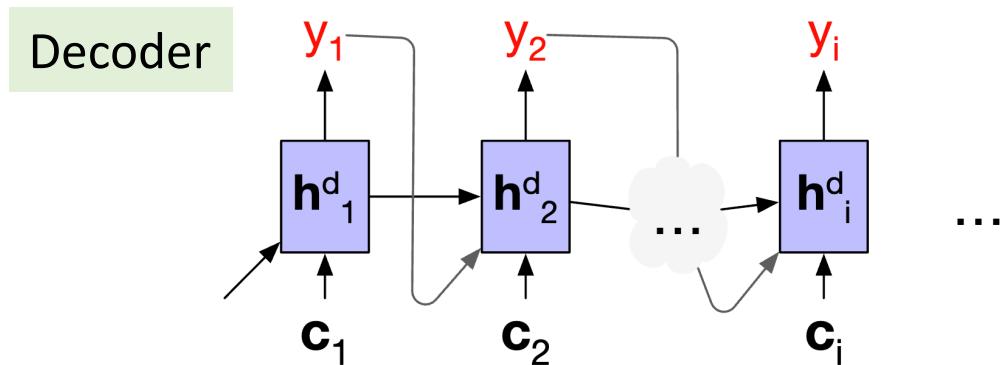
- **Bottleneck problem:** the last hidden state h_n^e from the encoder must represent *everything* about the input meaning -- It is difficult because earlier information at the beginning (especially for long sentence) may not be well represented



- **Attention mechanism** is a solution: allowing the decoder to get information from *all* the hidden states of encoder, not just the last one
- **Idea:** Create the context vector c by taking a weighted sum of all the encoder states h_1^e, \dots, h_n^e
- The weights attend to a particular part of input that is relevant for the token currently being produced by the decoder

Attention in Encoder-Decoder

- **Attention idea:** replaces the *static* context \mathbf{c} with a *dynamically* derived one \mathbf{c}_i , which is different for each decoding step i



The context used for decoding at step 1 (\mathbf{c}_1) is different from the one for step 2 (\mathbf{c}_2)

Hidden state used for decoding at step i:

$$\mathbf{h}_i^d = g(y_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i)$$

\mathbf{c}_i should capture how *relevant* each encoder \mathbf{h}_j^e is to the decoder state \mathbf{h}_{i-1}^d , using some score: $\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)$

Simplest option -- **dot-product attention**:

$$\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e) = \mathbf{h}_{i-1}^d \cdot \mathbf{h}_j^e$$

Measures relevance by similarity between vectors

Dot-Product Attention

$$\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e) = \mathbf{h}_{i-1}^d \cdot \mathbf{h}_j^e$$

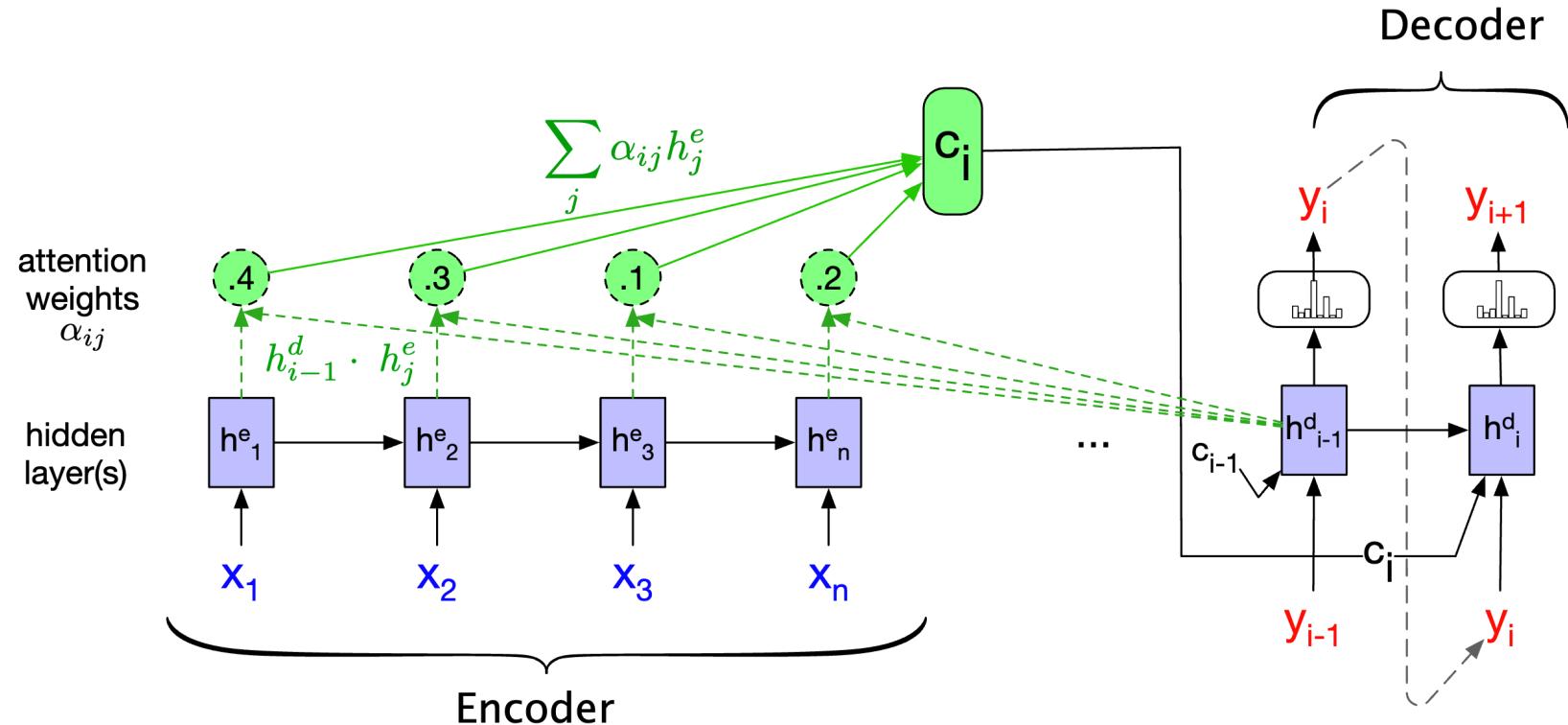
- Score from dot-product reflects the degree of similarity between two vectors
- Normalize the scores with a softmax to obtain vector of weights, α_{ij} , that tells the proportional relevance of each encoder state \mathbf{h}_j^e to the decoder state \mathbf{h}_{i-1}^d :

$$\alpha_{ij} = \text{softmax}\left(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)\right) = \frac{\exp\left(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_j^e)\right)}{\sum_k \exp\left(\text{score}(\mathbf{h}_{i-1}^d, \mathbf{h}_k^e)\right)}$$

- Finally, given the α_{ij} , we can compute the dynamic context vector \mathbf{c}_i by taking a weighted average of all encoder hidden states:

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j^e$$

Encoder-Decoder with Attention



What if h_{i-1}^d, h_j^e are of different dimensions?

Need a more sophisticated scoring function, such as bilinear:

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d W_s h_j^e$$

The **self-attention** defined in transformer architecture is a modification on attention

Overview

- Motivation
- **Transformer Basics**
 - Self-Attention; Multihead Attention
 - Transformer Blocks
 - Positional Embedding
- Transformer Details
- Extensions

Transformer Architecture Overview

- Intuition: across a series of layers, build up richer and richer contextualized representations of the meanings of input words
- Goal: produce a contextualized representation for each word at each position



- Input: sequence of words \Rightarrow Output: sequence of contextual embeddings
- *Like* LSTMs: Neural architecture;
- *Unlike* LSTMs: But not based on recurrent connections
- Made up of stacks of transformer **blocks**
- A block is made of linear layers, feedforward layers, and **self-attention layers** (the key innovation)

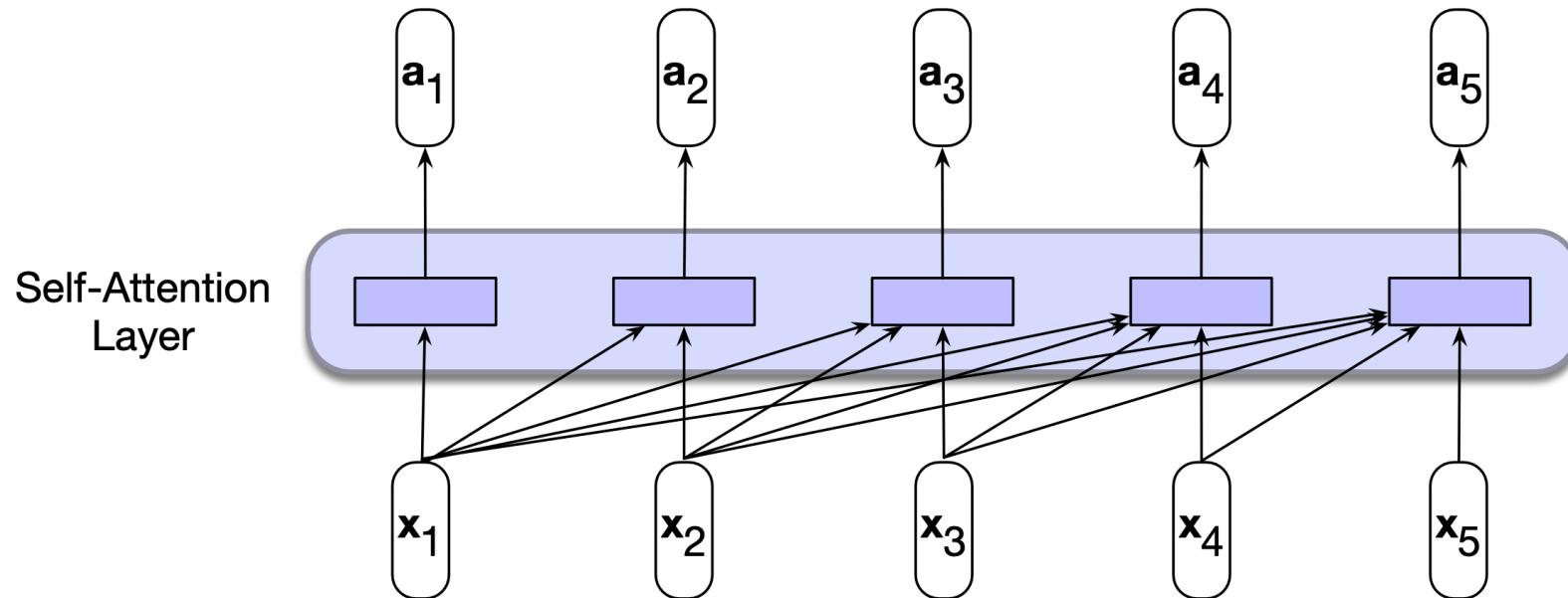
Self-Attention: Motivation

- Sentence: I walked along the **pond**, and noticed that one of the trees along the **bank** had fallen into the **water** after the storm.
- “bank” means 河床 or 银行 here?
- Probably the former because of the presence of “pond” and “water”

- Thus, we need a mechanism that can look broadly in the context to compute representations of words
- **Self-attention** is such a mechanism: given the representations of words from previous layer, look broadly in the context and integrate them into the representations for the next layer

Directions of Context

- In **causal**, or **backward looking** self-attention, context is any of the prior words
- In **bidirectional** self-attention, context can include future words
- We focus on **causal** one for this lecture



Self-attention maps

input sequence x_1, \dots, x_n

to output sequence of the same length a_1, \dots, a_n

When processing x_i , it has access to x_1, x_2, \dots, x_i , but no access to beyond: x_{i+1}, \dots

Self-Attention (version 1): Using Dot-Product

- Recall attention in encoder-decoder RNNs: the intuition of attention is to compare an item with other items in a way to reveal their relevance
- In the case of self-attention (causal), a word is compared to its preceding words; the result is then used to compute the output for current word
- Use our old friend **dot-product**:

② Normalize to a probability distribution:

$$\textcircled{1} \quad \text{score}(x_i, x_j) = x_i \cdot x_j \quad \forall j \leq i$$

$$\alpha_{ij} = \text{softmax}(x_i \cdot x_j) = \frac{\exp(x_i \cdot x_j)}{\sum_k \exp(x_i \cdot x_k)} \quad \forall j \leq i$$

③ Compute output a_i as the weighted average: $a_i = \sum_{j \leq i} \alpha_{ij} x_j$

Note: α_{ij} is likely to be highest for $i = j$, i.e., a word is most similar to itself

Self-Attention (version 2): query, key, and value

- Considering **three different roles** each input plays during the attention process:
- Query**: As the current focus of attention when comparing to all the other preceding inputs;
- Key**: As a preceding input being compared to;
- Value**: Used to compute the output for the current focus of attention.
- Introduce weight matrices \mathbf{W}^Q , \mathbf{W}^K , and \mathbf{W}^V to project each input vector x_i into a representation of its role as a key, query, and value:

$$q_i = x_i \mathbf{W}^Q; \quad k_i = x_i \mathbf{W}^K; \quad v_i = x_i \mathbf{W}^V$$

Dimensions: $\left. \begin{array}{l} \text{input } x_i \\ \text{output } y_i \\ \text{intermediate } a_i \end{array} \right\} \in \mathbb{R}^d$

$\left. \begin{array}{l} q_i \\ k_i \\ v_i \end{array} \right\} \in \mathbb{R}^{d_k}$
 $v_i \in \mathbb{R}^{d_v}$

$\mathbf{W}^Q \in \mathbb{R}^{d \times d_k}$
 $\mathbf{W}^K \in \mathbb{R}^{d \times d_k}$
 $\mathbf{W}^V \in \mathbb{R}^{d \times d_v}$

In the original work
 (Vaswani et al. 2017):
 $d = 512$,
 $d_k = d_v = 64$

Self-Attention (version 2): query, key, and value

- To compute score between a current focus of attention x_i , and a preceding element x_j , use the dot-product between its **query** vector q_i and **key** vector k_j :

$$\text{score}(x_i, x_j) = q_i \cdot k_j$$

- Normalize the scores into α_{ij} using softmax (same as version 1)
- Compute the output a_i as the weighted sum of **value** vectors v :

Note: $d = 512$, $d_k = d_v = 64$
 How do 64-d vectors (v_j) end up in 512-d (a_i)? -- Concatenation (explained in multihead attention)

$$a_i = \sum_{j \leq i} \alpha_{ij} v_j$$

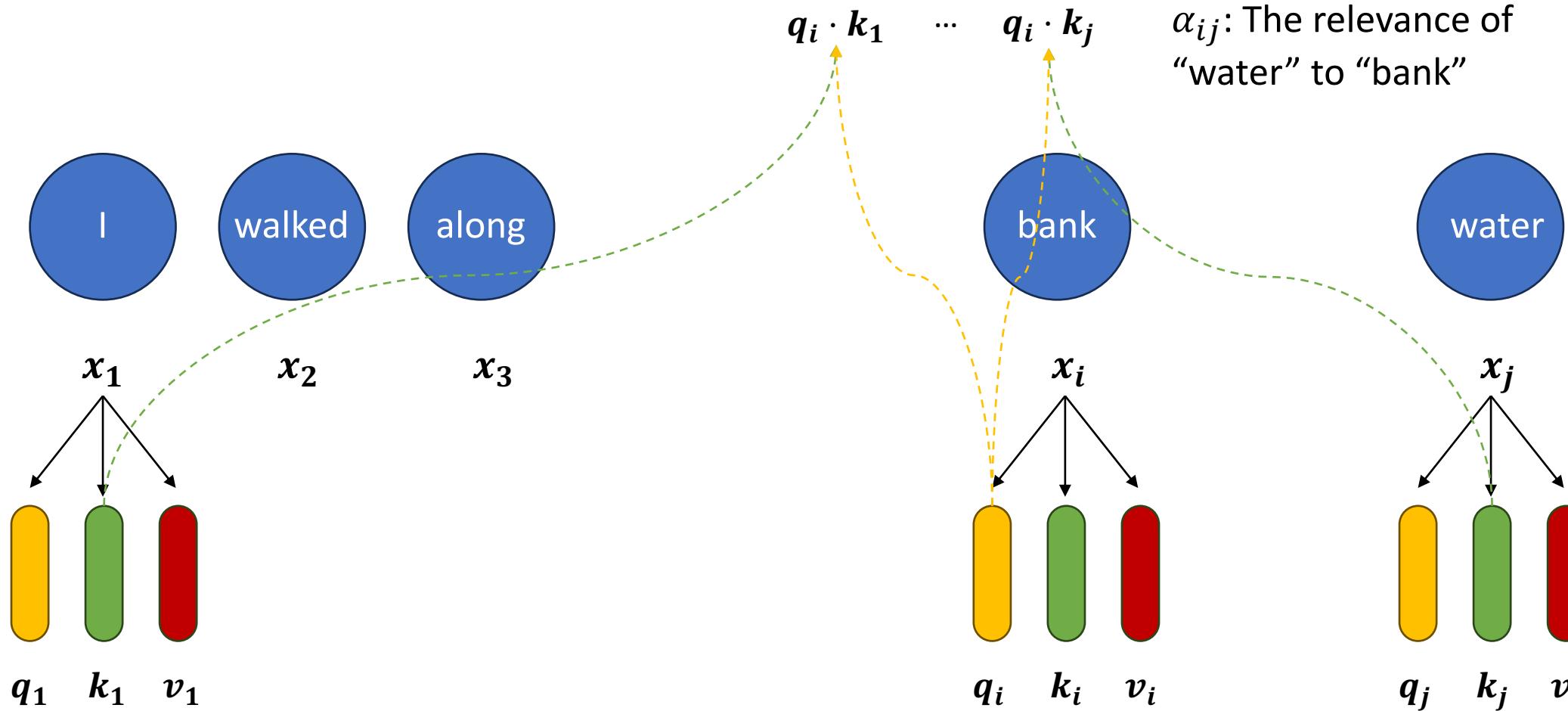
Note: similar to version 1, α_{ij} is also likely to be highest for $i = j$

- To fix numeric issues in exponentiating large values, scale down the dot-product by the size of embedding:

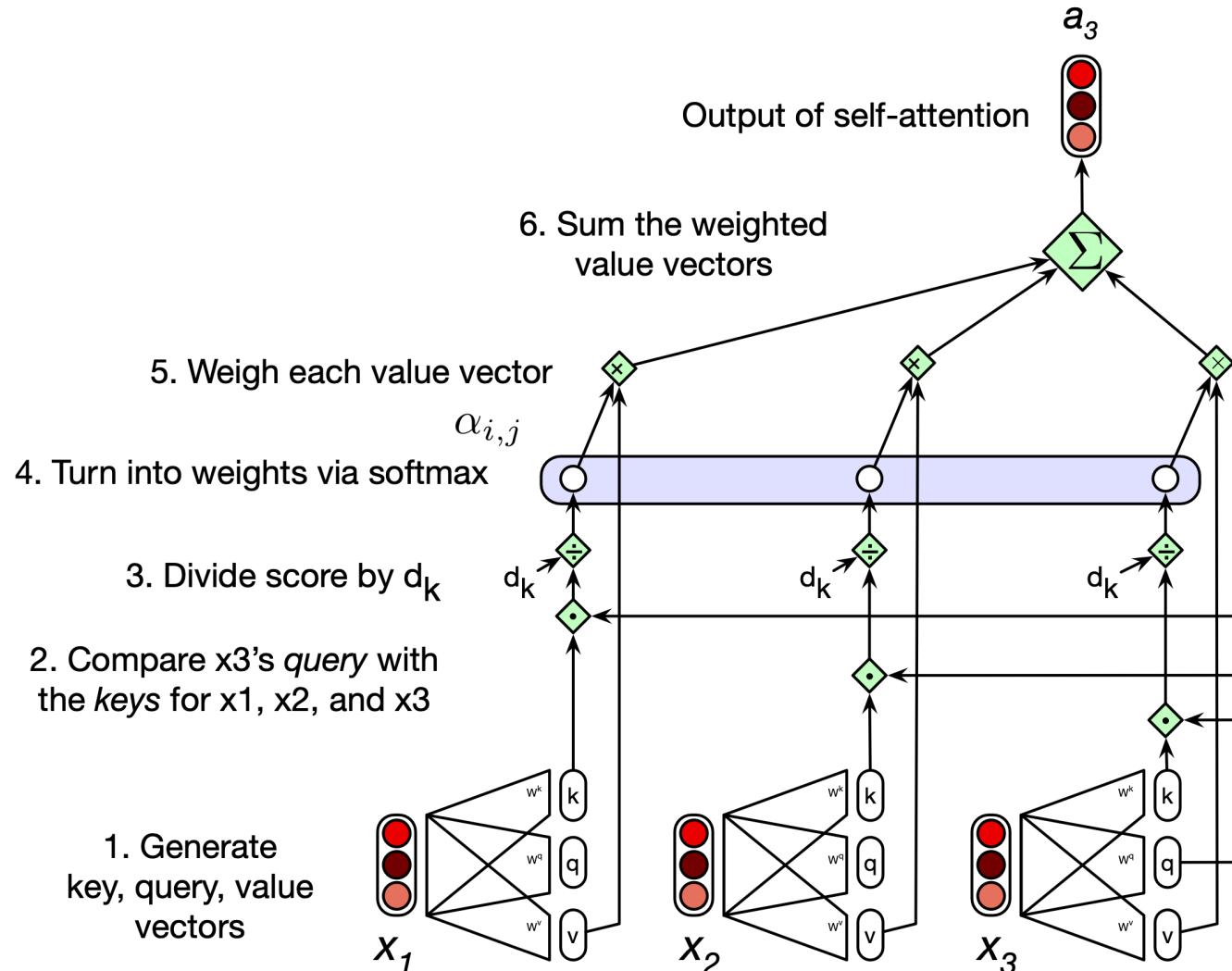
$$\text{score}(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

Self-attention example

α_{i1} : The relevance of “I” to “bank”



Self-Attention Final Version



$$\begin{aligned} q_i &= x_i W^Q \\ k_i &= x_i W^K \\ v_i &= x_i W^V \end{aligned}$$

$$\text{score}(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

$$\alpha_{ij} = \text{softmax}(\text{score}(x_i, x_j))$$

$$\forall j \leq i$$

$$a_i = \sum_{j \leq i} \alpha_{ij} v_j$$

Parallelizing Self-Attention using Matrices

- Given a sequence of N input tokens, then the input embedding is $X \in \mathbb{R}^{N \times d}$ (N could be 512, 1024, 2048, 4096, or more)
- Compute the query, key, and value vectors for N tokens all at once:

$$Q = XW^Q; \quad K = XW^K; \quad V = XW^V$$

- Resulting in three matrices: $Q \in \mathbb{R}^{N \times d_k}$, $K \in \mathbb{R}^{N \times d_k}$, and $V \in \mathbb{R}^{N \times d_v}$
- Then the attention score can be computed by QK^\top , resulting in $N \times N$ scores
- The output:

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

Note two issues: 1) QK^\top results in a $N \times N$ score matrix, in which the future needs be masked
 2) $A \in \mathbb{R}^{N \times d}$ and $V \in \mathbb{R}^{N \times d_v}$, if $d \neq d_v$, then need concatenation (multihead)

Parallelizing Self-Attention using Matrices

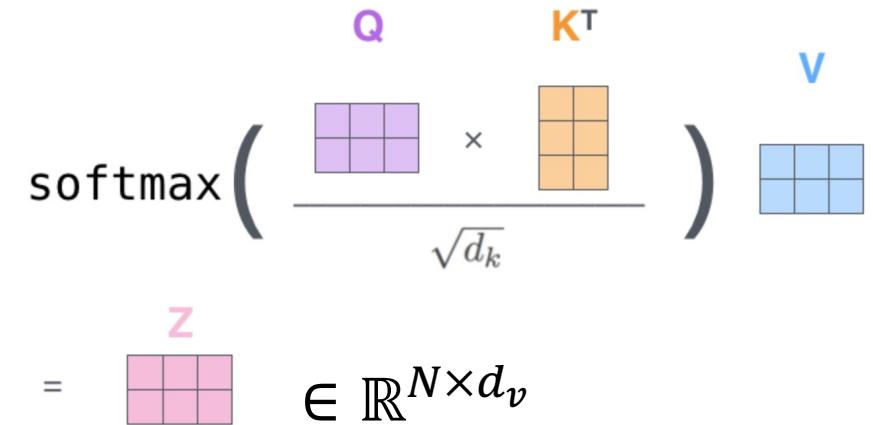
$$X \in \mathbb{R}^{N \times d}$$

$$\begin{array}{ccc} X & \times & W_Q \\ \begin{matrix} \text{green} \\ \boxed{\text{green}} \\ \text{green} \end{matrix} & \times & \begin{matrix} \text{purple} \\ \boxed{\text{purple}} \\ \text{purple} \end{matrix} \\ & = & \begin{matrix} \text{purple} \\ \boxed{\text{purple}} \\ \text{purple} \end{matrix} \quad Q \in \mathbb{R}^{N \times d_k} \end{array}$$

$$\begin{array}{ccc} X & \times & W_K \\ \begin{matrix} \text{green} \\ \boxed{\text{green}} \\ \text{green} \end{matrix} & \times & \begin{matrix} \text{orange} \\ \boxed{\text{orange}} \\ \text{orange} \end{matrix} \\ & = & \begin{matrix} \text{orange} \\ \boxed{\text{orange}} \\ \text{orange} \end{matrix} \quad K \in \mathbb{R}^{N \times d_k} \end{array}$$

$$\begin{array}{ccc} X & \times & W_V \\ \begin{matrix} \text{green} \\ \boxed{\text{green}} \\ \text{green} \end{matrix} & \times & \begin{matrix} \text{blue} \\ \boxed{\text{blue}} \\ \text{blue} \end{matrix} \\ & = & \begin{matrix} \text{blue} \\ \boxed{\text{blue}} \\ \text{blue} \end{matrix} \quad V \in \mathbb{R}^{N \times d_v} \end{array}$$

$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) = Z \in \mathbb{R}^{N \times d_v}$$

$$V$$


Source: <https://jalammar.github.io/illustrated-transformer/>

Masking Out the Future

- The self-attention so far has a problem: QK^T results in a score for each query to every key, including those that come after the query
- Inappropriate for language modeling: guessing the next word is simple if we already know it
- Fix:** Zero out the upper-triangular portion (set to $-\infty$, which softmax will turn to 0), thus eliminating any knowledge of words from the future

	N			
$q1 \cdot k1$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$q2 \cdot k1$	$q2 \cdot k2$	$-\infty$	$-\infty$	$-\infty$
$q3 \cdot k1$	$q3 \cdot k2$	$q3 \cdot k3$	$-\infty$	$-\infty$
$q4 \cdot k1$	$q4 \cdot k2$	$q4 \cdot k3$	$q4 \cdot k4$	$-\infty$
$q5 \cdot k1$	$q5 \cdot k2$	$q5 \cdot k3$	$q5 \cdot k4$	$q5 \cdot k5$

Multihead Attention

- **Motivation:** Different words relate to other words in the sentence in many different ways simultaneously
- Ex. *syntactic*, *semantic*, and *discourse* relationships between verb and its argument
- **Difficult** for a single self-attention model to capture all these relations
- **Solution: multihead self-attention layers**
 - Parallel layers at the same depth; each with its own parameters
 - Each head, i , has its own query, key, and value matrices:
 $\mathbf{W}_i^Q \in \mathbb{R}^{N \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{N \times d_k}$, and $\mathbf{W}_i^V \in \mathbb{R}^{N \times d_v}$
 - Given input $\mathbf{X} \in \mathbb{R}^{N \times d}$, They produce $\mathbf{Q}_i \in \mathbb{R}^{N \times d_k}$, $\mathbf{K}_i \in \mathbb{R}^{N \times d_k}$, and $\mathbf{V}_i \in \mathbb{R}^{N \times d_v}$
 - In the original paper, number of heads $h = 8$, $d = 512$, $d_k = d_v = 64$
 - The output of each head is of shape $N \times d_v$; the outputs from h heads are concatenated to $N \times hd_v$

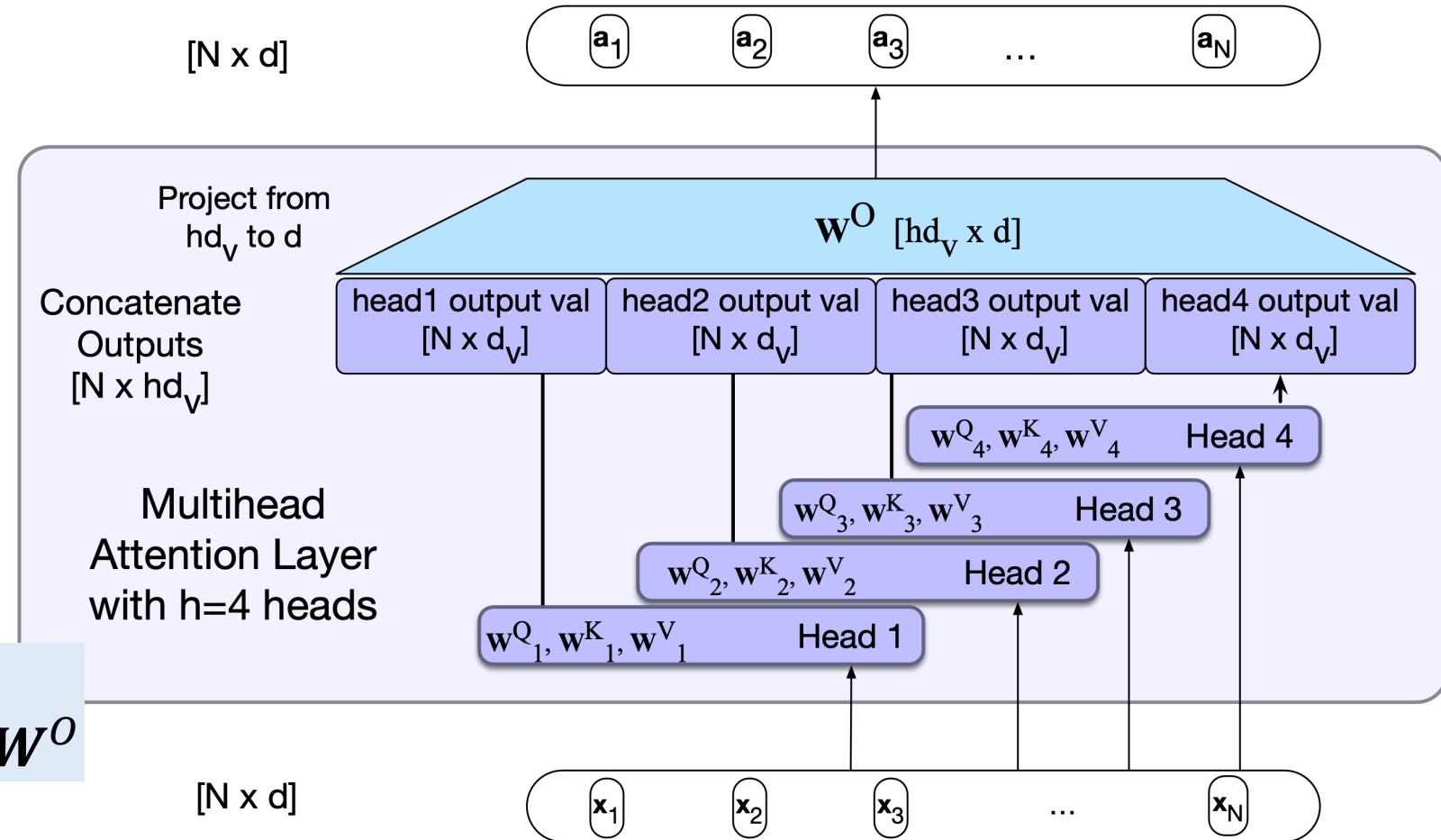
Multihead Attention

- Finally, use yet another linear projection $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d}$ to reshape to original dimension d for each token

$$\begin{aligned} Q_i &= X\mathbf{W}_i^Q \\ K_i &= X\mathbf{W}_i^K \\ V_i &= X\mathbf{W}_i^V \end{aligned}$$

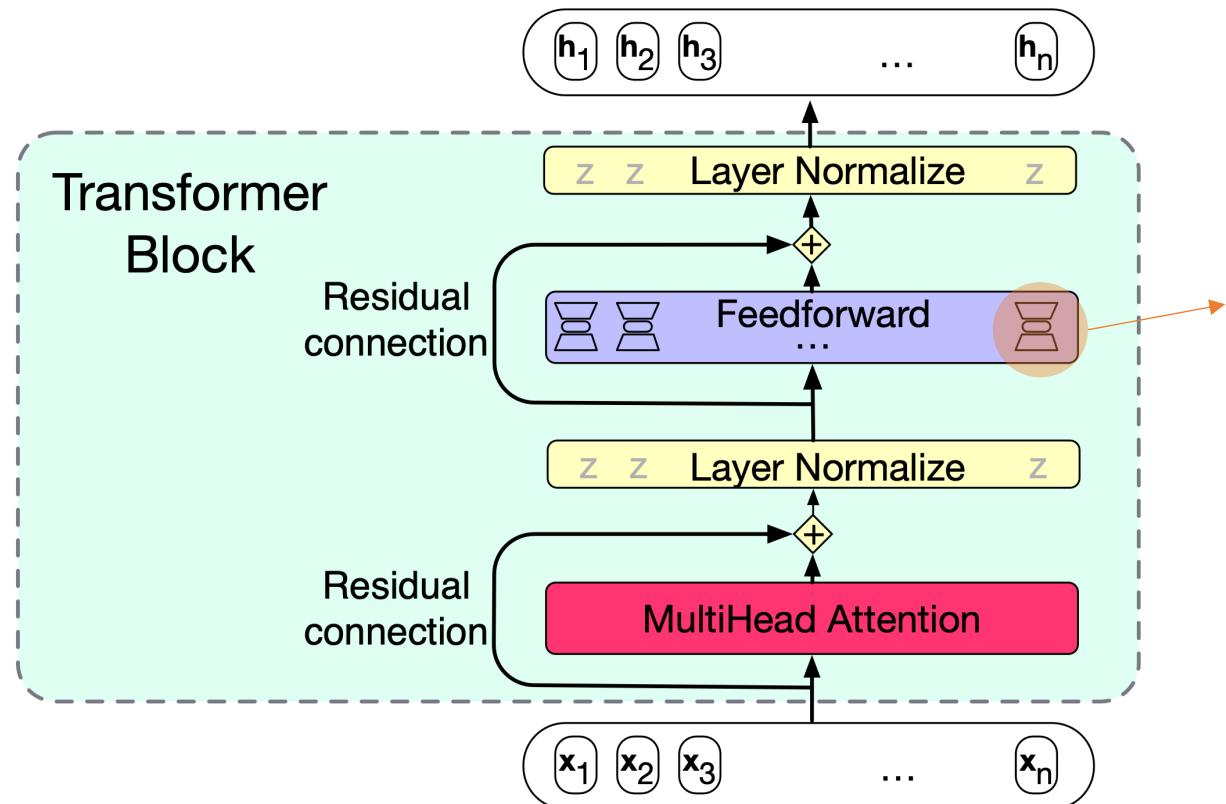
head_i
 $= \text{SelfAttention}(Q_i, K_i, V_i)$

$A = \text{MultiheadAttention}(X)$
 $= (\text{head}_1 \oplus \text{head}_2 \dots \oplus \text{head}_h) \mathbf{W}^O$



Transformer Blocks

- A Transformer block = Self-attention layer +
- 1) Feedforward layer + 2) Residual connections + 3) Layer normalization

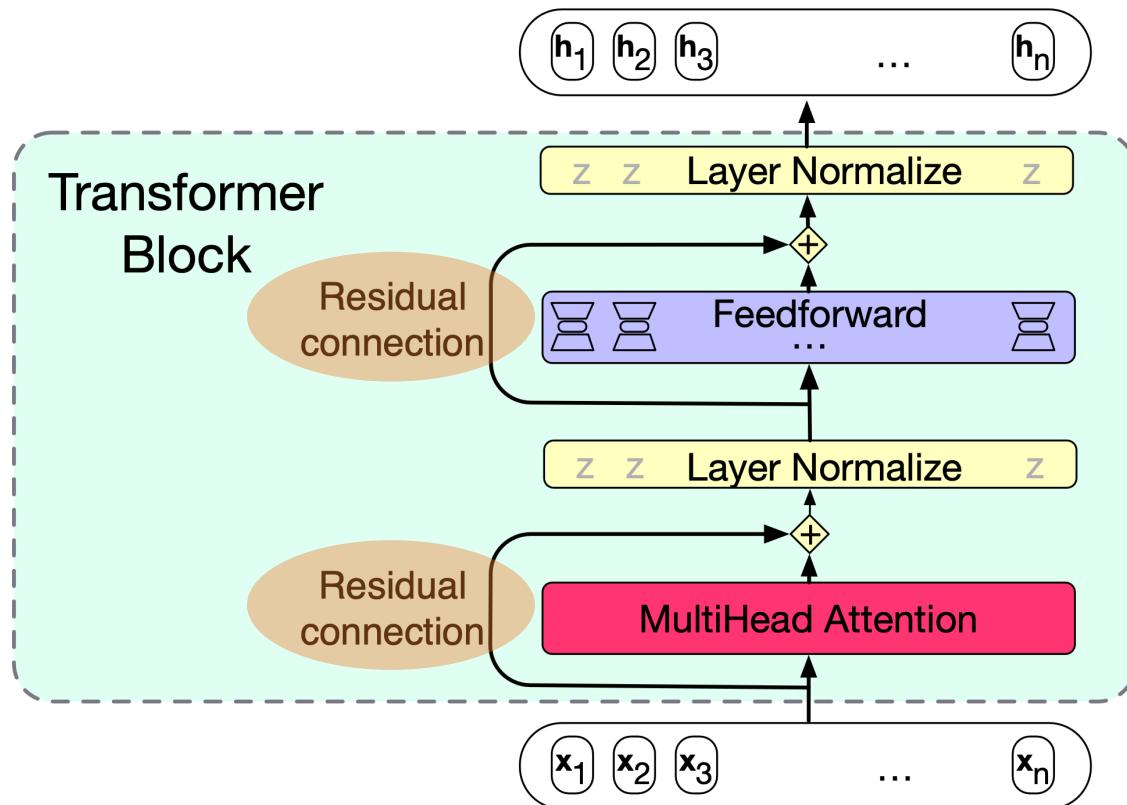


Feedforward layer contains N position-wise networks:

- Each is a 2-layer fully-connected net
- Parameters are different from layer to layer
- Same weights applied to each token position at the same layer
- $d = 512, d_{ff} = 2048$

Transformer Blocks: Residual Connections

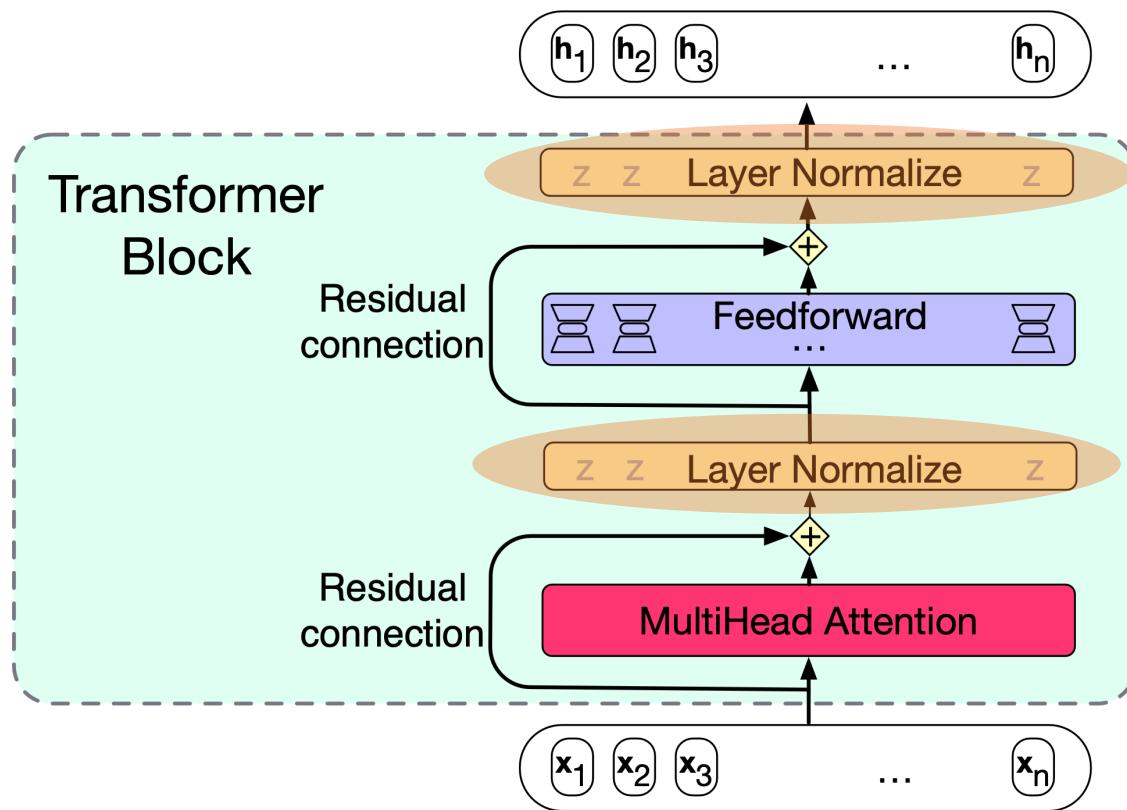
- **Residual connections** pass information from lower layer to higher layer without going through the intermediate layer



- Implemented by adding a layer's input to its output before passing it forward
- Allowing activations to directly go forward and gradients to directly go backward

Transformer Blocks: Layer Normalization

- **Layer normalization** (Ba et al., 2016): keep the values of hidden layer in range that facilitates gradient descent



Input: the vector for a particular position, x
 Output: the vector normalized, \hat{x}

First, compute the mean and standard deviation over the elements of x :

$$\mu = \frac{1}{d} \sum_{i=1}^d x_i \quad \sigma = \sqrt{\frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2}$$

Then, normalize by subtracting and dividing:

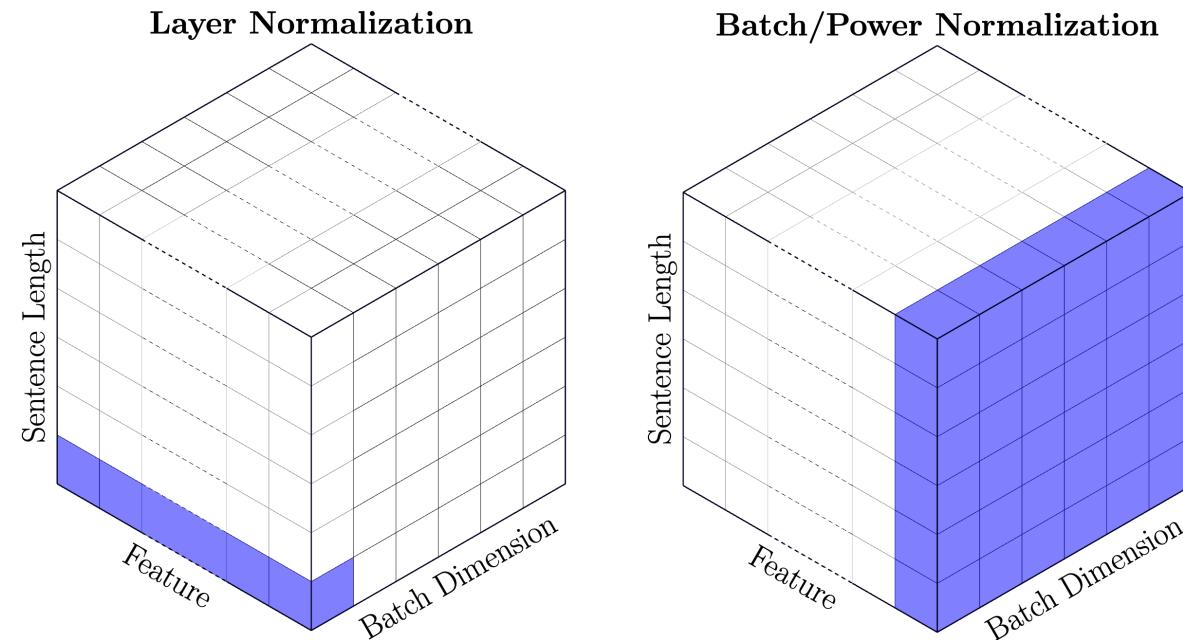
$$\hat{x} = (x - \mu) / \sigma$$

Finally, add learnable parameters γ, β

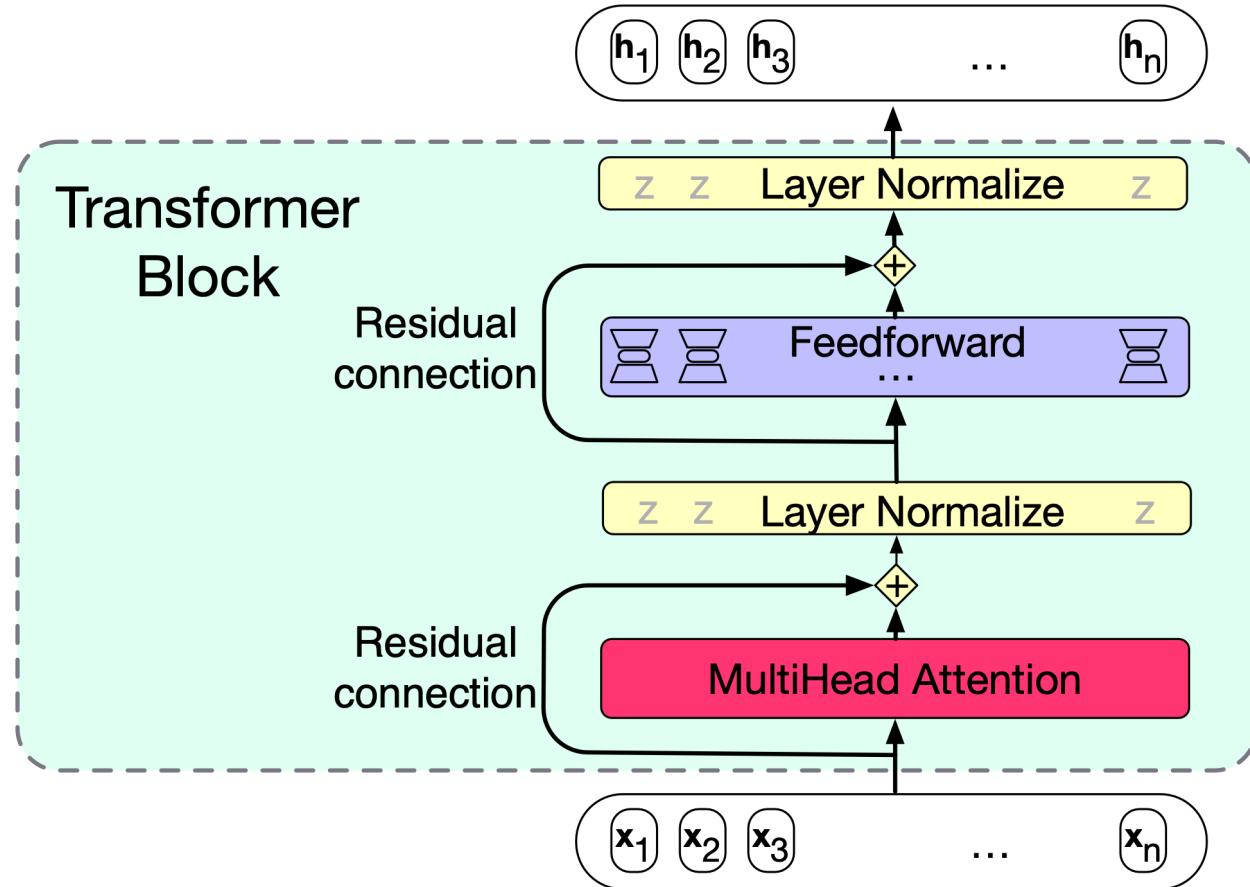
$$\text{LayerNorm} = \gamma \hat{x} + \beta$$

Layer Norm and Batch Norm

- **Why not using batch normalization instead?**
- Perhaps because “statistics of NLP data across the batch exhibit large fluctuations throughout training. This results in instability, if BN is naively implemented.” (Shen et al., 2020)



Transformer Blocks: All Together



Input: X , output: H

$$O = \text{LayerNorm}(X + \text{SelfAttention}(X))$$

$$H = \text{LayerNorm}(O + \text{FFN}(O))$$

break down:

Output H

$$H = \text{LayerNorm}(T^5)$$

$$T^5 = T^4 + T^3$$

$$T^4 = \text{FFN}(T^3)$$

$$T^3 = \text{LayerNorm}(T^2)$$

$$T^2 = X + T^1$$

$$T^1 = \text{SelfAttention}(X)$$

Input X

Transformer Blocks

- A bit difference between encoder and decoder

For encoder:

$$X'_l = \text{LayerNorm}(\text{MHA}(X_{l-1}) + X_{l-1}),$$

$$X_l = \text{LayerNorm}(\text{FFN}(X'_l) + X'_l),$$

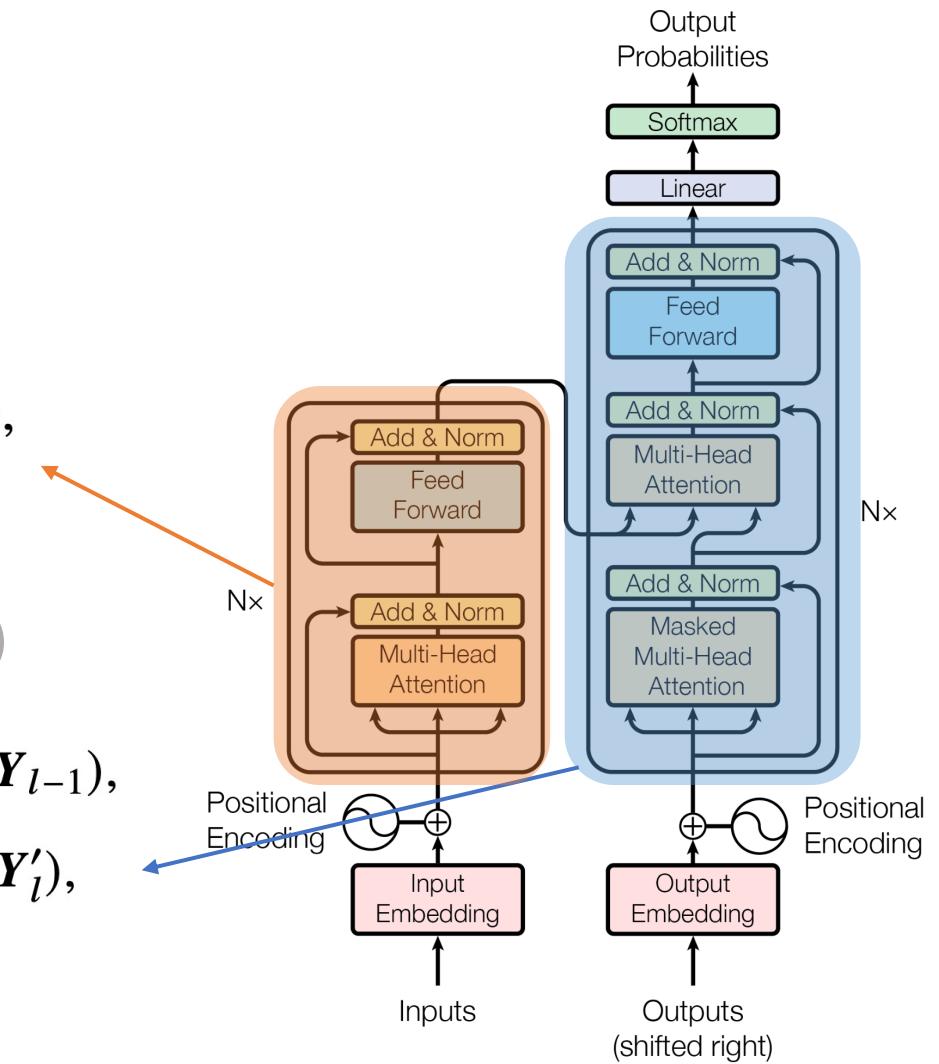
("MHA" for Multi-Head Self-Attention)

For decoder:

$$Y'_l = \text{LayerNorm}(\text{MaskedMHA}(Y_{l-1}) + Y_{l-1}),$$

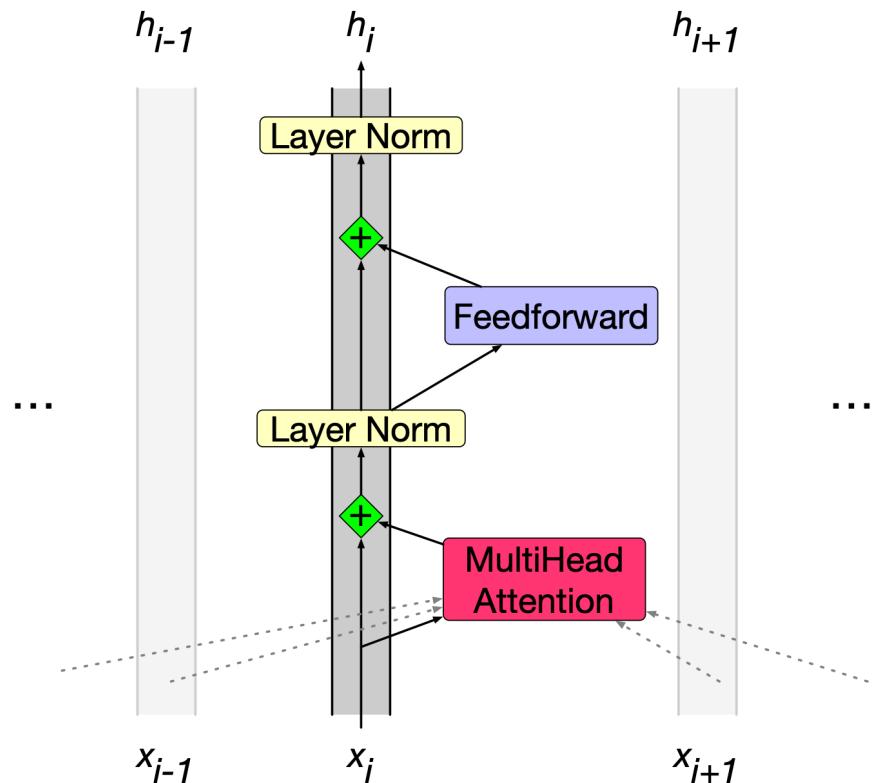
$$Y''_l = \text{LayerNorm}(\text{CrossMHA}(Y'_l, X_L) + Y'_l),$$

$$Y_l = \text{LayerNorm}(\text{FFN}(Y''_l) + Y''_l),$$



Residual Stream View of Transformer Block

- A **residual stream** view: the transformer model processes each individual token through all the layers as a stream of d -dimensional representations



For each token i , at each block and layer, we pass up an embedding of shape $1 \times d$

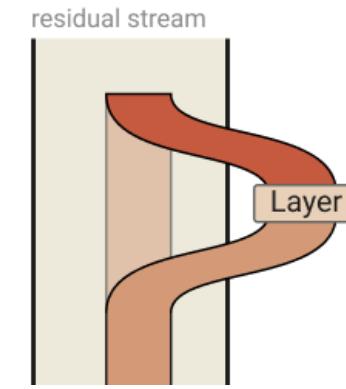
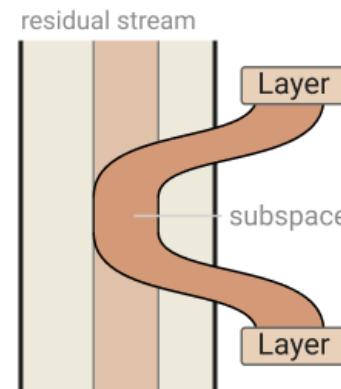
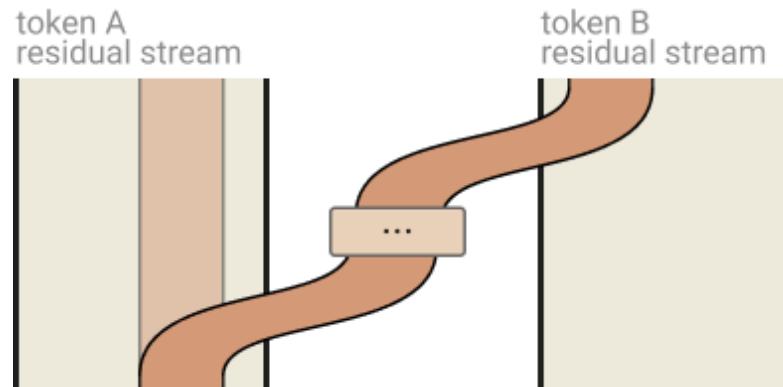
The layers are constantly copying information up from earlier embeddings

Other components add information into this **constant stream**:

- Feedforward adds in a different view of earlier embedding
- Multi-head attention adds information from other tokens

Residual Stream View of Transformer Block

- Attention heads move information from one stream to another



Residual stream can be divided into different subspaces

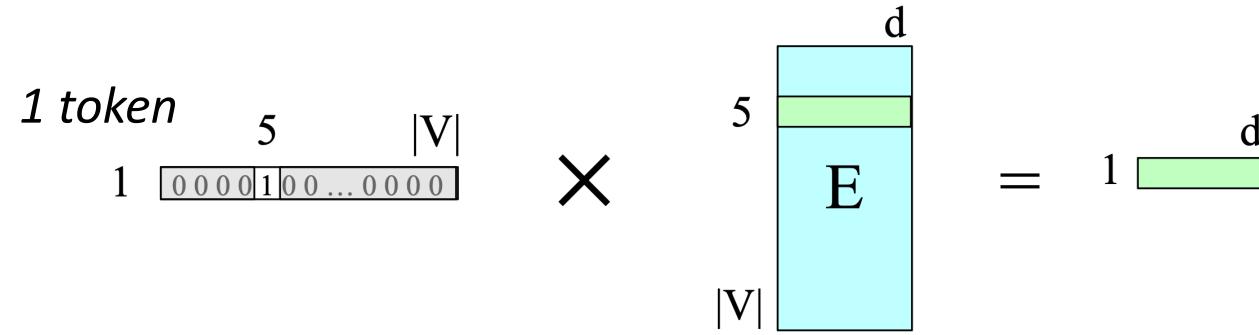
Layers can modify information from residual stream by reading in a subspace

Source: <https://transformer-circuits.pub/2021/framework/index.html>

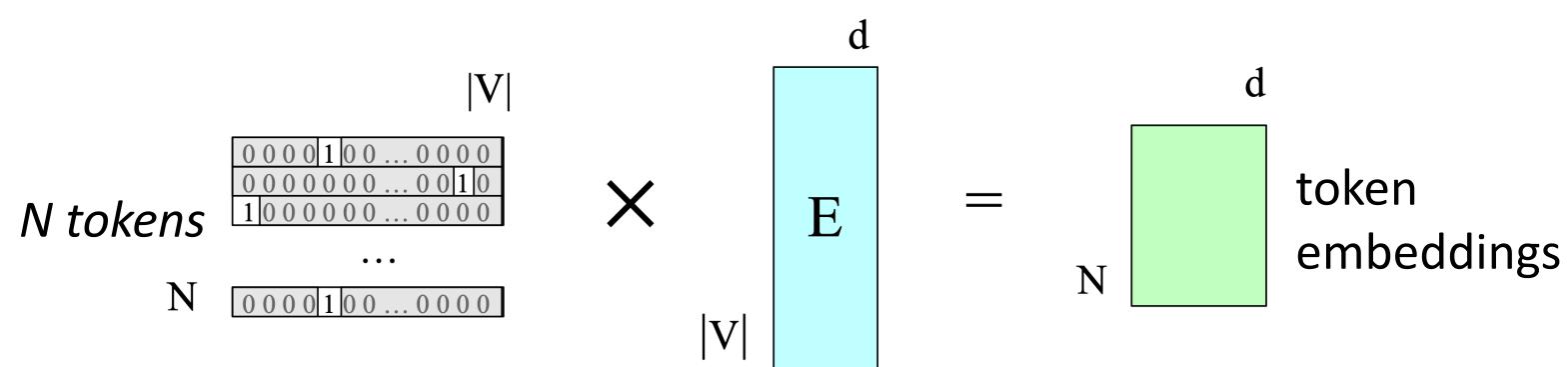
Input Embeddings for Token and Position

- Input embedding to a transformer block X : a matrix of shape $N \times d$
- It has two components: ***token*** embedding + ***positional*** embedding

Token embedding:
One-hot vectors \Rightarrow
dense vectors

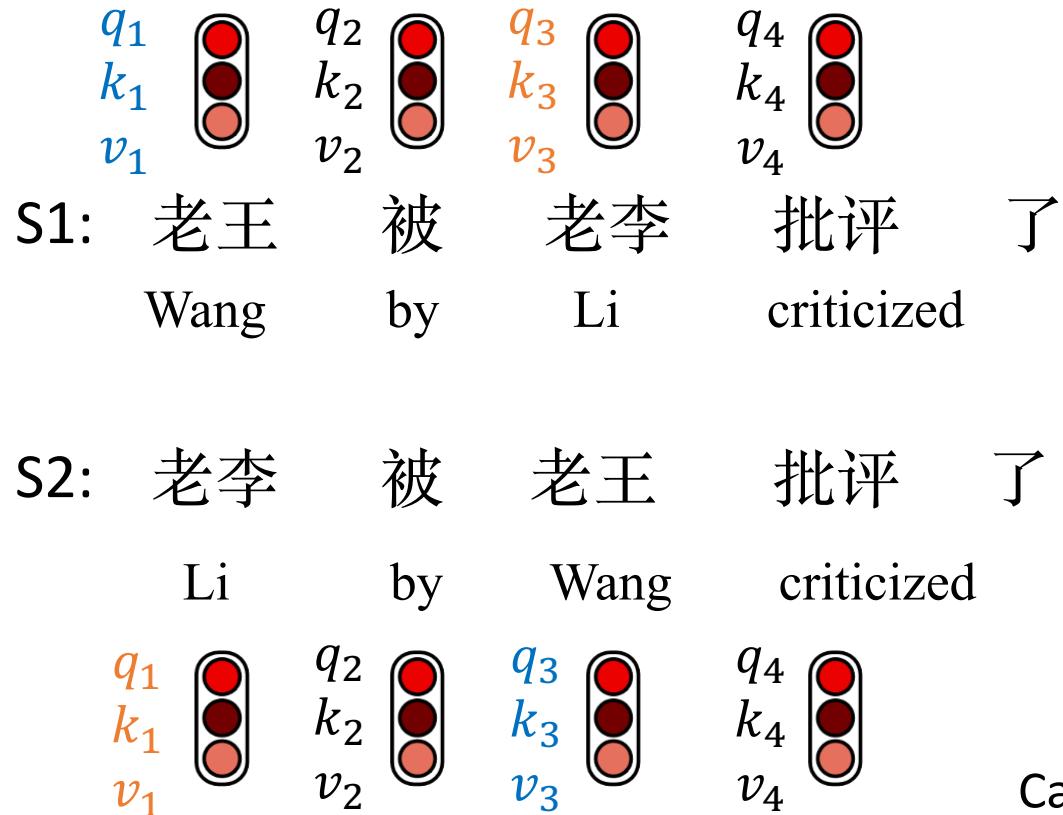


Problem: token embeddings are not position-dependent



Necessity of Positional Embedding

- Word order matters



representation for “批评” in two sentences

$$a_4 = \alpha_{41}v_1 + \alpha_{42}v_2 + \alpha_{43}v_3$$

Exactly the same if
“老王” and “老李” are
represented by token
embeddings only

$$a_4 = \alpha_{41}v_1 + \alpha_{42}v_2 + \alpha_{43}v_3$$

Cannot tell who criticized whom by looking at the output

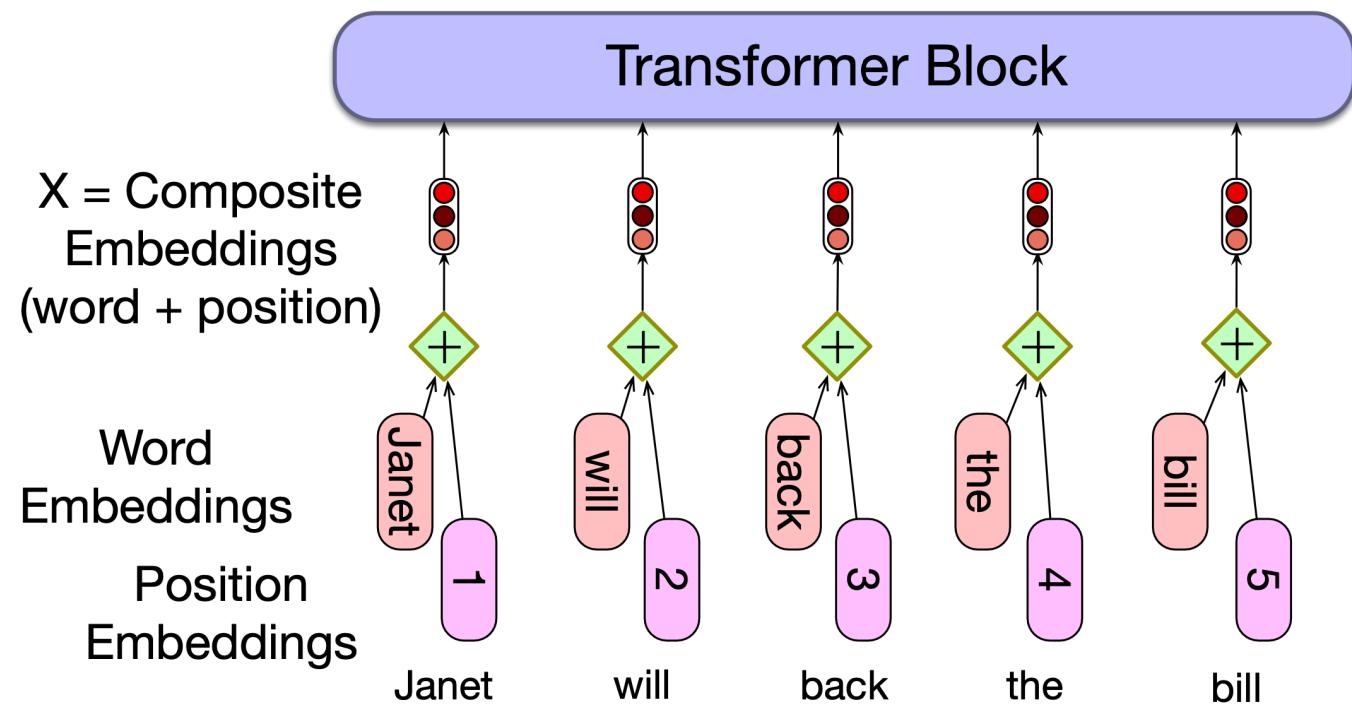
Positional Embedding: Methods

- Simplest method: **absolute position**
- Randomly initialize embeddings for each possible input position

$$\mathbf{X}_{Janet} = \mathbf{E}_{ID(Janet)} + \mathbf{P}_i$$

\mathbf{E}_i and \mathbf{P}_i are of same dimension d

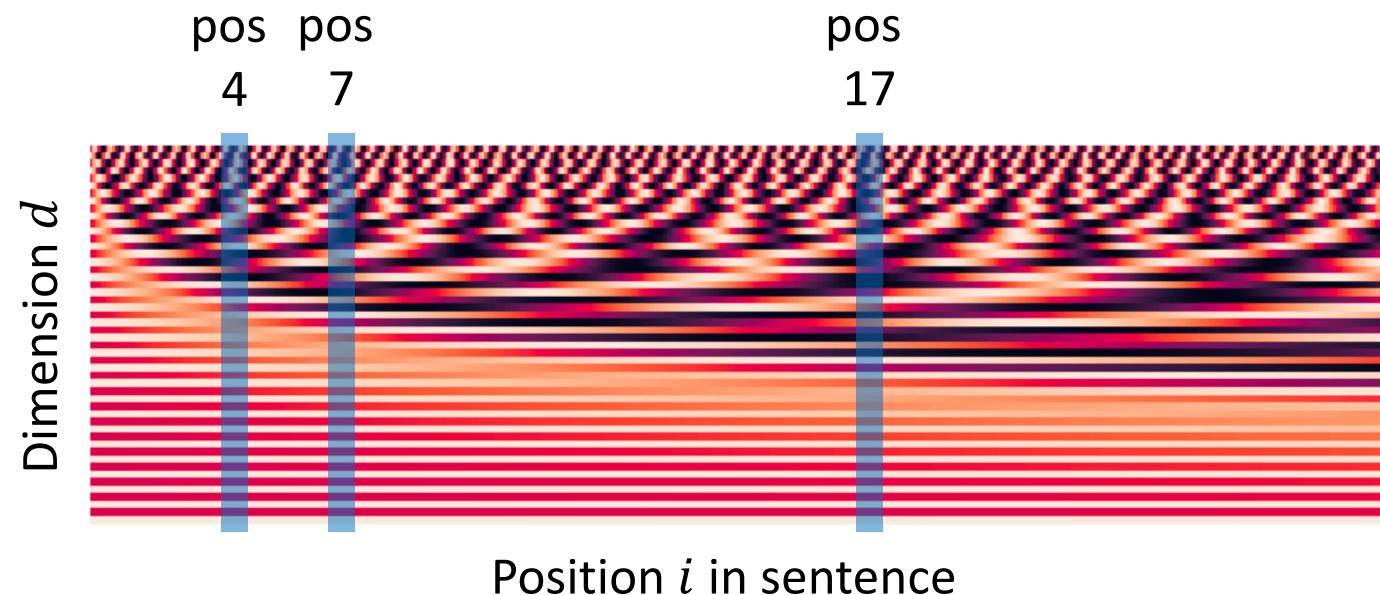
Drawback: Initial positions (small values) dominate the training data; later positions (large values) are poorly trained



Positional Embedding: Absolute Position

- In the original transformer work: **sinusoidal (正弦曲线) positional embedding** by concatenating sine and cosine functions of varying periods

$$\mathbf{P}_i = \left[\begin{array}{c} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \sin(i/10000^{2*2/d}) \\ \cos(i/10000^{2*2/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{array} \right] \quad \left. \right\} d \quad (= 512)$$



Intuition: pos 4 is “closer” to pos 7 than pos 17

Alternative: Rotary Position Embedding (RoPE)

- Construct a rotation matrix to encode relative positional information Su et al., 2021

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

Position i : $\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^Q \mathbf{R}_{\theta,i}$, position j : $\mathbf{k}_i = \mathbf{x}_j \mathbf{W}^K \mathbf{R}_{\theta,j}$

Then attention $A_{ij} = \mathbf{q}_i \mathbf{k}_i^\top = \mathbf{x}_i \mathbf{W}^Q \mathbf{R}_{\theta,i} (\mathbf{x}_j \mathbf{W}^K \mathbf{R}_{\theta,j})^\top = \mathbf{x}_i \mathbf{W}^Q \mathbf{R}_{\theta,i} \mathbf{R}_{\theta,j}^\top \mathbf{W}^{K\top} \mathbf{x}_j^\top$

Rotary Position Embedding (RoPE) cont.

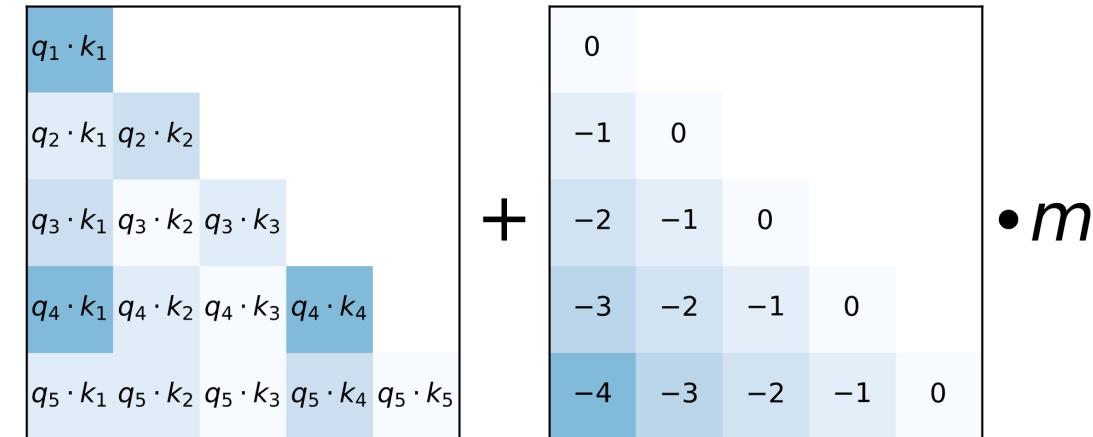
- $A_{ij} = \mathbf{q}_i \mathbf{k}_i^\top = \mathbf{x}_i \mathbf{W}^Q \mathbf{R}_{\theta,i} \mathbf{R}_{\theta,j}^\top \mathbf{W}^{K\top} \mathbf{x}_j^\top = \mathbf{x}_i \mathbf{W}^Q \mathbf{R}_{\theta,i-j} \mathbf{W}^{K\top} \mathbf{x}_j^\top$
- Attention is modeled by the rotation matrix $\mathbf{R}_{\theta,i-j}$, which includes relative positional information
- RoPE has better overall performance than sinusoidal PE
- Used in LLaMA and PaLM

Alternative: ALiBi Position Embedding

- ALiBi: Attention with Linear Biases (Press et al., 2022)
- Enable a model to achieve extrapolation at inference time for sequences that are longer than it saw during training
- by adding penalty to attention scores on longer distances:

$$A_{ij} = \mathbf{q}_i \mathbf{k}_i^\top - m(i - j)$$

m is a head-specific scalar that is set and not learned throughout training

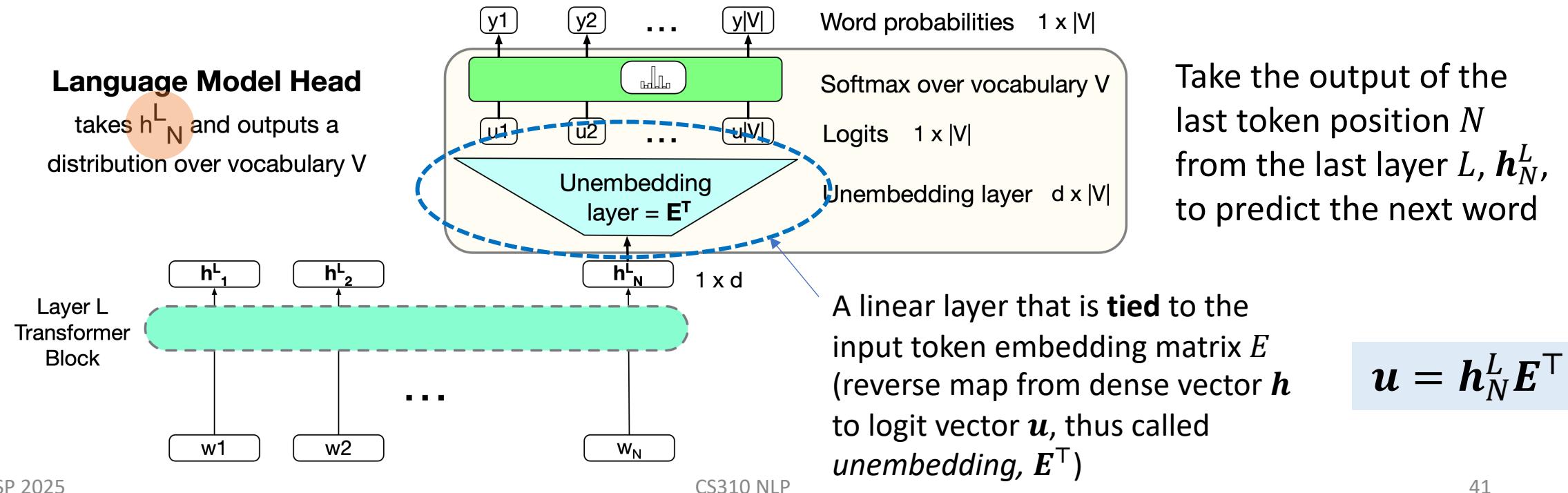


Overview

- Motivation
- Transformer Basics
- **Transformer Details**
 - Language Modeling Head
 - Architectures
 - Training
- Extensions

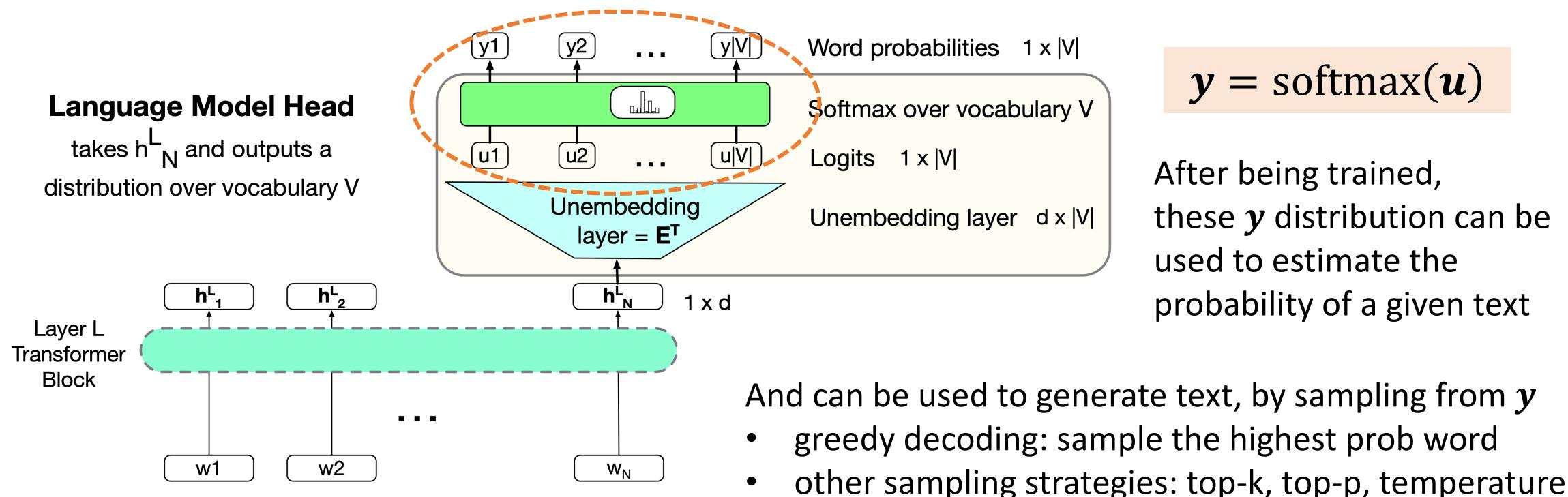
Language Modeling Head

- **Head:** additional neural network added on top of the basic transformer architecture to enable certain task
- LM head \Rightarrow Language Modeling task: Predicting next word from **context**
- **Context size:** transformer's context window -- 512, 1024, 2048, 4096, or even larger



Language Modeling Head

- A softmax layer turns the logits \mathbf{u} into the probabilities \mathbf{y} over vocabulary



Training Transformer as Language Model

- Just like training an RNN language model, we use the same **self-supervision** training method: **predict the next word**
- At each position t , take as input the word sequence $w_{1:t}$, and use them to compute a probability distribution \hat{y}_t , and then compute the **cross-entropy** loss between this distribution and the actual next word $y_t[w_{t+1}]$

y_t is a one-hot vector of length $|V|$:

- $y_t[w = w_{t+1}] = 1$
- $y_t[w \neq w_{t+1}] = 0$

Cross-entropy loss:

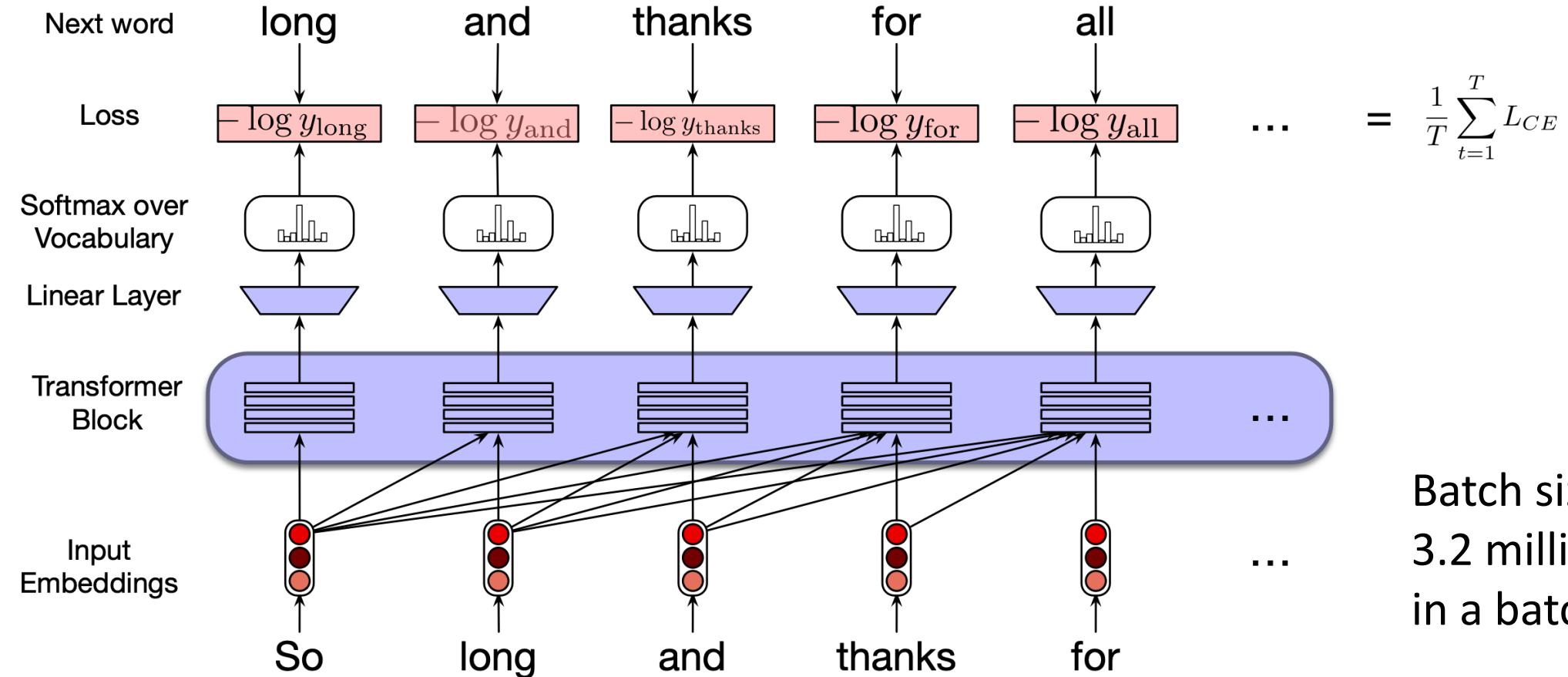
- $L_{CE} = -\sum_{w \in V} y_t[w] \log \hat{y}_t[w_{t+1}] = -\log \hat{y}_t[w_{t+1}]$
- Then average over entire sequence: $\frac{1}{T} \sum_{t=1}^T L_{CE}$

$T = 2048$ or 4096 for GPT3 or GPT3.5

GPT4: 8192; GPT4 Turbo: 128000 (2024.4 data)

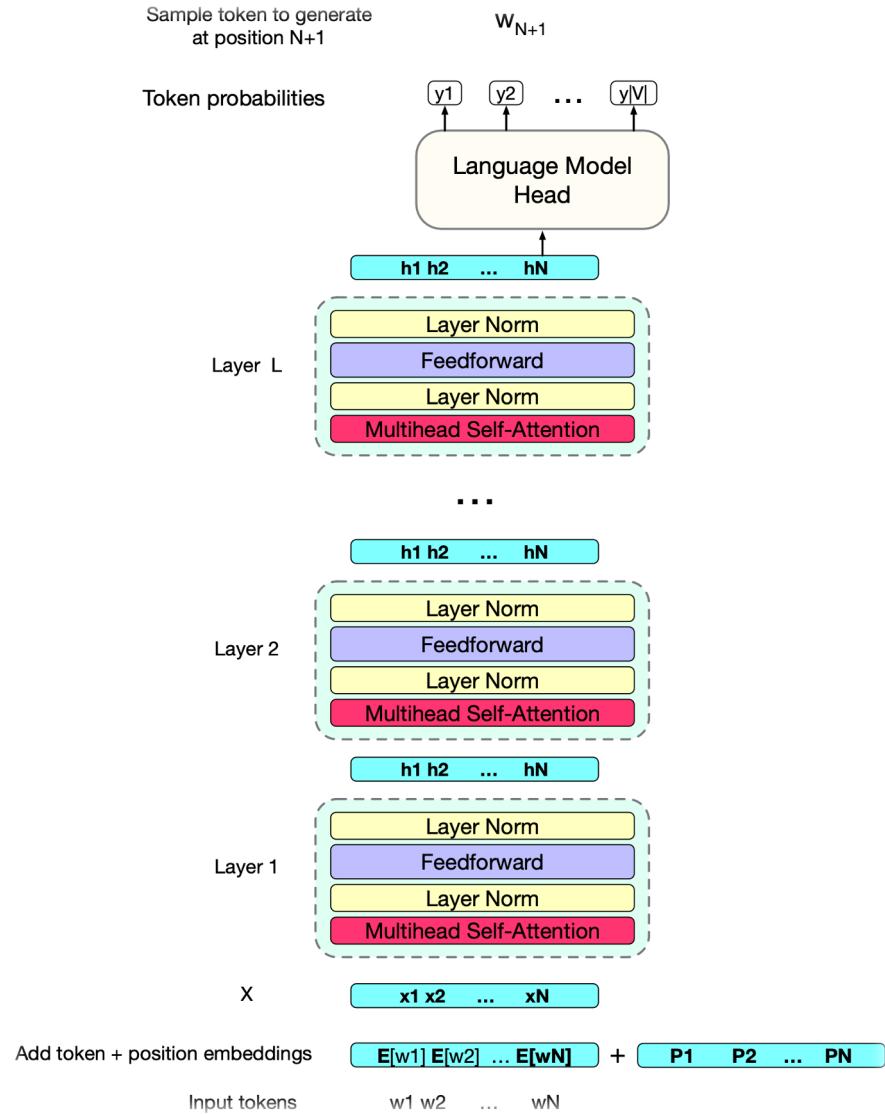
Source: <https://www.scriptbyai.com/token-limit-openai-chatgpt>

Training Transformer as Language Model



Batch size is large:
3.2 million tokens
in a batch for GPT3

Architecture: Decoder-only



Transformer model has a stacked architecture of multiple layers of transformer blocks

The input to the first block is word embedding + positional embedding: $X = E_w + P_i$

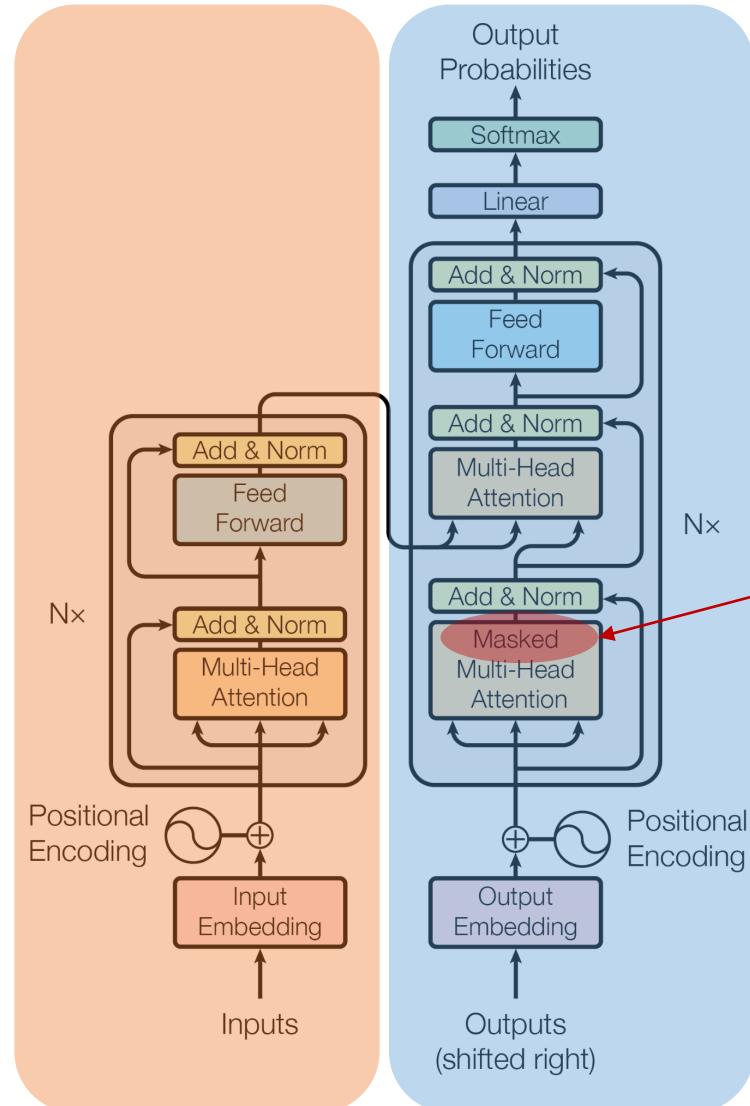
The input to all the other layers is the output H from the layer below

Terminology:

- **decoder-only model:** it is only half of the encoder-decoder model in the original work

Architecture: Encoder-decoder

Encoder



Decoder

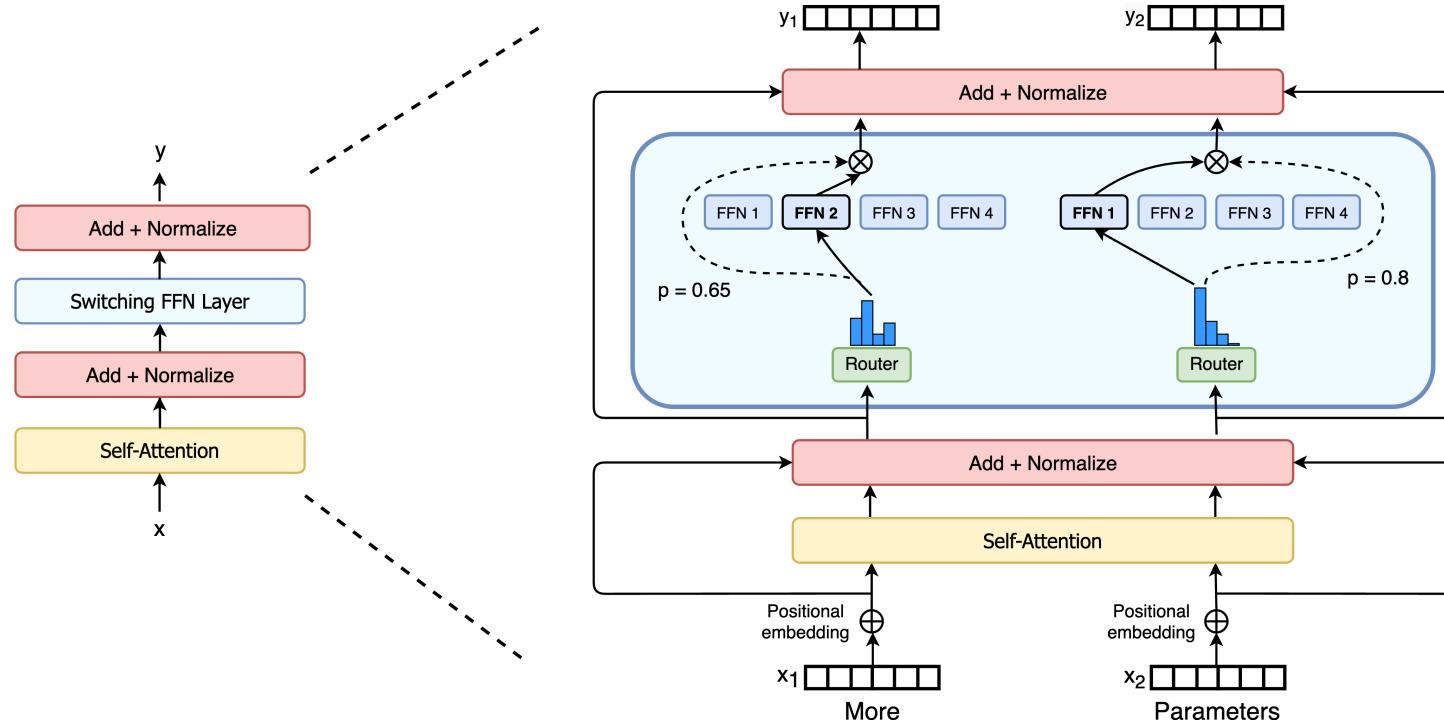
Vaswani et al. (2017)'s original work is for machine translation task

- In the encoder, the self-attention is bidirectional;
- In the decoder, the self-attention is causal, i.e., future words are masked out

It was only later that the paradigm for causal language model was defined using only the decoder part

Architecture: Mixture of Experts (MoE)

- How to train faster? -- Replace every FFN layer of the transformer model with an MoE layer



Each MoE layer has K experts:

$$E_1, E_2, \dots, E_K$$

Each E_i is a FFN

Each token x_t is passed through a **router**, or **gate** G , computing the weight of each expert

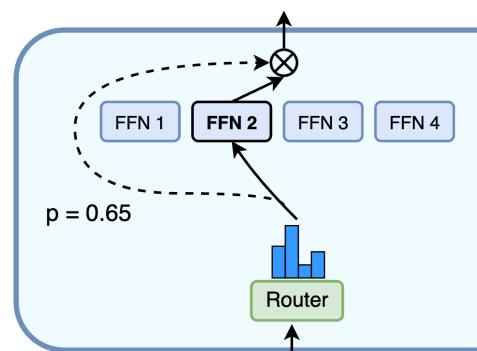
$$G(x_t) = \text{softmax}(\text{topk}(x_t \cdot W^G))$$

Architecture: Mixture of Experts (MoE) cont.

- Only top k experts will be selected (other non-selected experts will be placed with weight 0)

$$G(\mathbf{x}_t) = \text{softmax}(\text{topk}(\mathbf{x}_t \cdot \mathbf{W}^G))$$

- $G(\mathbf{x}_t)$ is in the form: $[G(\mathbf{x}_t)_1, \dots, G(\mathbf{x}_t)_k]$
- The output is the weighted sum of all selected experts



$$\text{MoELayer}(\mathbf{x}_t) = \sum_{i=1}^K G(\mathbf{x}_t)_i \cdot E_i(\mathbf{x}_t)$$

Comparison of Model Architecture

	LLaMA-3.1 (405B)	DeepSeek (67B)	DS-V2 (236B)	DS-V3 (671B)
MoE	No	No	162 experts	257 expert
# of layers	126	95	60	61
Hidden dimension d	16384	8192	5120	7168
# of attention heads	128	64	128	128

Types of Normalization

- Original transformer: Layer normalization

$$\mu = \frac{1}{d} \sum_{i=1}^d x_i \quad \sigma = \sqrt{\frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2}$$

$$\text{LayerNorm}(\mathbf{x}) = \gamma \hat{\mathbf{x}} + \beta \quad \hat{\mathbf{x}} = (\mathbf{x} - \mu) / \sigma$$

- More advanced methods:
- Root Mean Square Layer Normalization: **RMSNorm**

$$\text{RMSNorm}(\mathbf{x}) = \gamma \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})}$$

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2}$$

- RMSNorm has better speed and performance
- Used in Gopher, Chinchilla, LLaMA 3.1, DeepSeek

Types of Normalization cont.

- DeepNorm
- Scale the residual connection with some factor α

$$\text{DeepNorm}(\mathbf{x}) = \text{LayerNorm}(\alpha \cdot \mathbf{x} + \text{Sublayer}(\mathbf{x}))$$

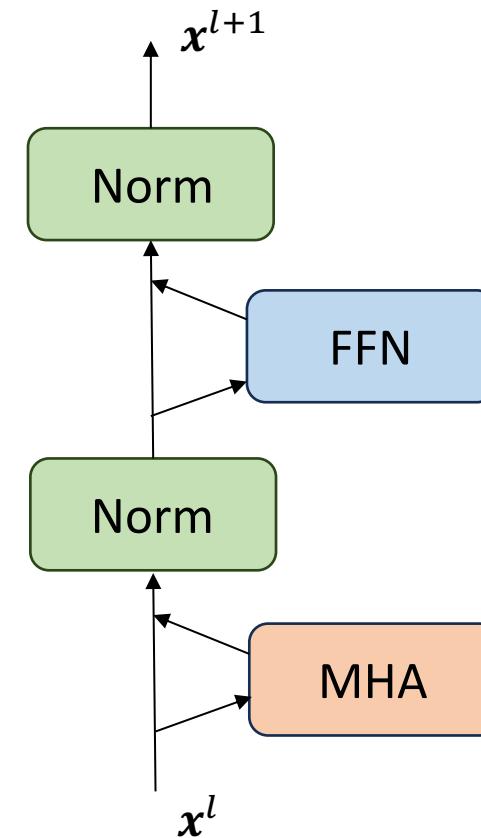
- Where Sublayer represents FFN or Self-Attention (MHA)
- More robust training effect, deeper networks (>1000)
- Used in GLM-130B

Position of Normalization

- **Pre-Norm:** normalization placed before residual connection, to every sublayer

$$\text{PreNorm}(x) = x + \text{Sublayer}(\text{Norm}(x))$$

- Pros: Training is more robust
- Cons: Less performant than Post-Norm

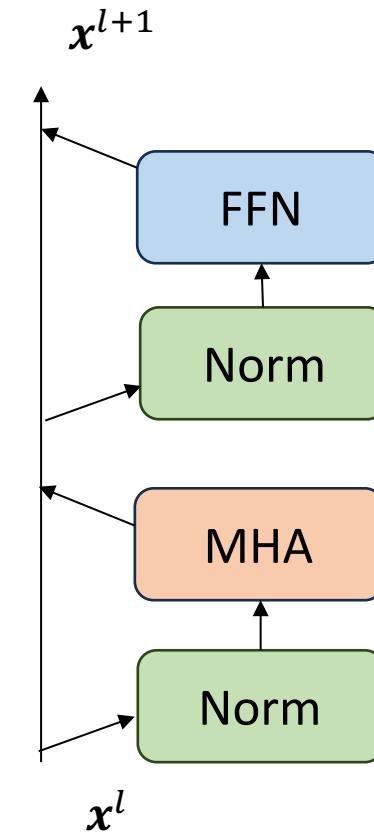


Position of Normalization

- **Post-Norm:** normalization placed after residual connection

$$\text{PostNorm}(x) = \text{Norm}(x + \text{Sublayer}(x))$$

- Used in original transformer
- Pros: Training is faster to converge
- Cons: Sometimes makes training less robust



Comparison of Model Architecture cont.

	LLaMA-3.1 (405B)	DeepSeek (67B)	DS-V2 (236B)	DS-V3 (671B)
MoE	No	No	162 experts	257 expert
# of layers	126	95	60	61
Hidden dimension d	16384	8192	5120	7168
# of attention heads	128	64	128	128
Normalization	Pre RMSNorm	Pre RMSNorm	Pre RMSNorm	Pre RMSNorm

Training: Data

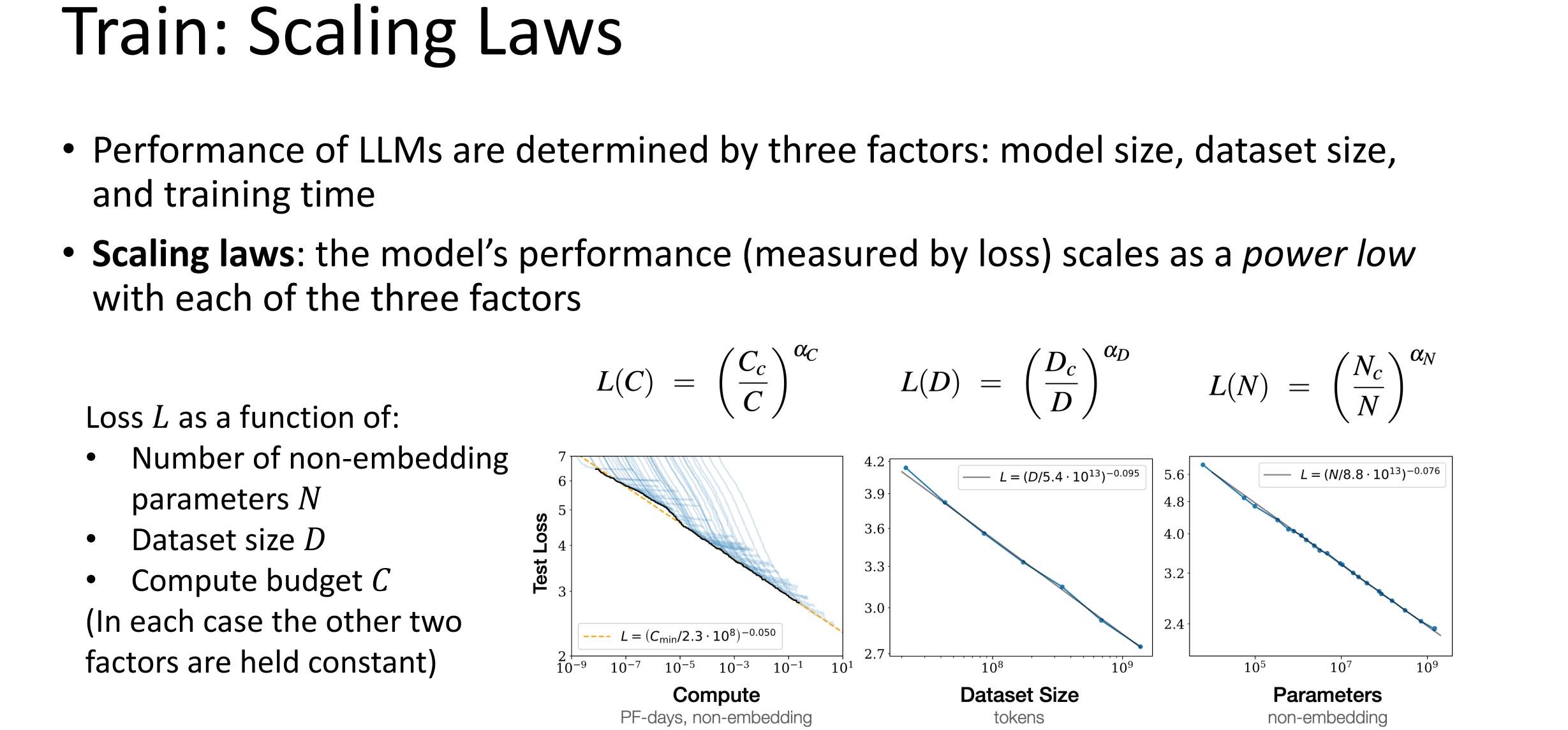
- Large Language Models (LLMs) are mainly trained on text scraped from the web, augmented by more carefully curated data
- Such large data are likely to contain many natural examples that are helpful to NLP tasks: question and answer pairs, translation, summaries, etc.
- **Common Crawl:** A series of snapshots of the entire web with billions of webpages
- Needs a LOT of cleaning work!
- Ex. Colossal Clean Crawled Corpus (C4; Raffel et al., 2020): 156 billion tokens
 - Mostly patent text documents, Wikipedia, and news sites (according to Dodge et al., 2021)
- MNBVC(Massive Never-ending BT Vast Chinese corpus)超大规模中文语料集

Train: Scaling Laws

- Performance of LLMs are determined by three factors: model size, dataset size, and training time
- **Scaling laws:** the model's performance (measured by loss) scales as a *power law* with each of the three factors

Loss L as a function of:

- Number of non-embedding parameters N
- Dataset size D
- Compute budget C
 (In each case the other two factors are held constant)



Ref: Kaplan et al. (2020) Scaling Laws for Neural Language Models. arxiv.org/abs/2001.08361

Training: Estimate # of Parameters

- The number of non-embedding parameters N can be estimated:
- $N \approx 2d \cdot n_{\text{layer}} \cdot (2d_{\text{attn}} + d_{\text{ff}}) \approx 12n_{\text{layer}}d^2$
- (Assuming $d = d_{\text{attn}} = d_{\text{ff}}$)
- For GPT-3, with $n = 96$ layers and $d = 12288$, $N \approx 12 \times 96 \times 12288^2 \approx 1.74 \times 10^{11}$
- That is 174 billion parameters (千亿)

Overview

- Motivation
- Transformer Basics
- Transformer Details
- **Extensions**

Achievement: Machine Translation

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$3.3 \cdot 10^{18}$	
Transformer (big)	28.4	41.0	$2.3 \cdot 10^{19}$	

FLOPS =
 Floating point
 operations per
 second

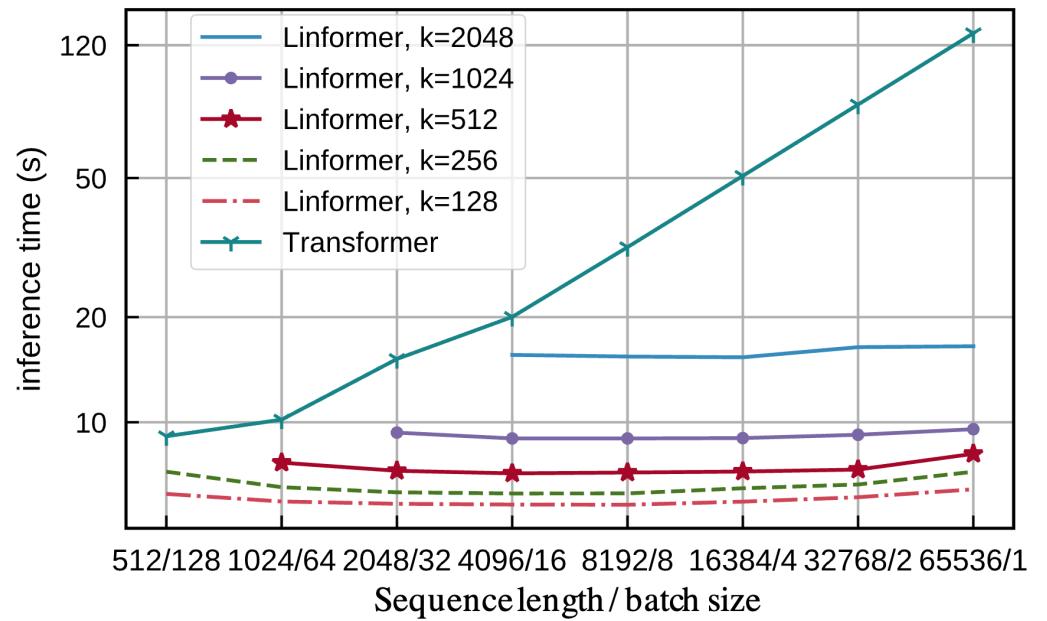
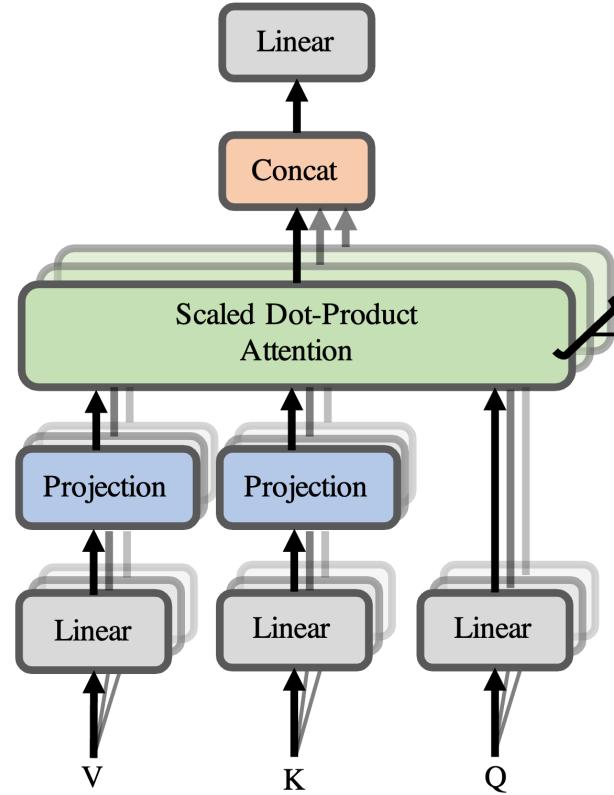
Table from Vaswani et al. (2017)

Drawbacks

- **Quadratic computing time** in self-attention
 - Computing all pairs of interactions means that computation grows quadratically with sequence length
 - For RNN models, it only grow linearly
-
- Total number of operation grows in $O(n^2d)$
 - For smaller models, $d \approx 1000$; for larger models, $d > 10000$
 - For short input, $n \approx 512$
 - For long input (e.g., long documents), $n > 50000$

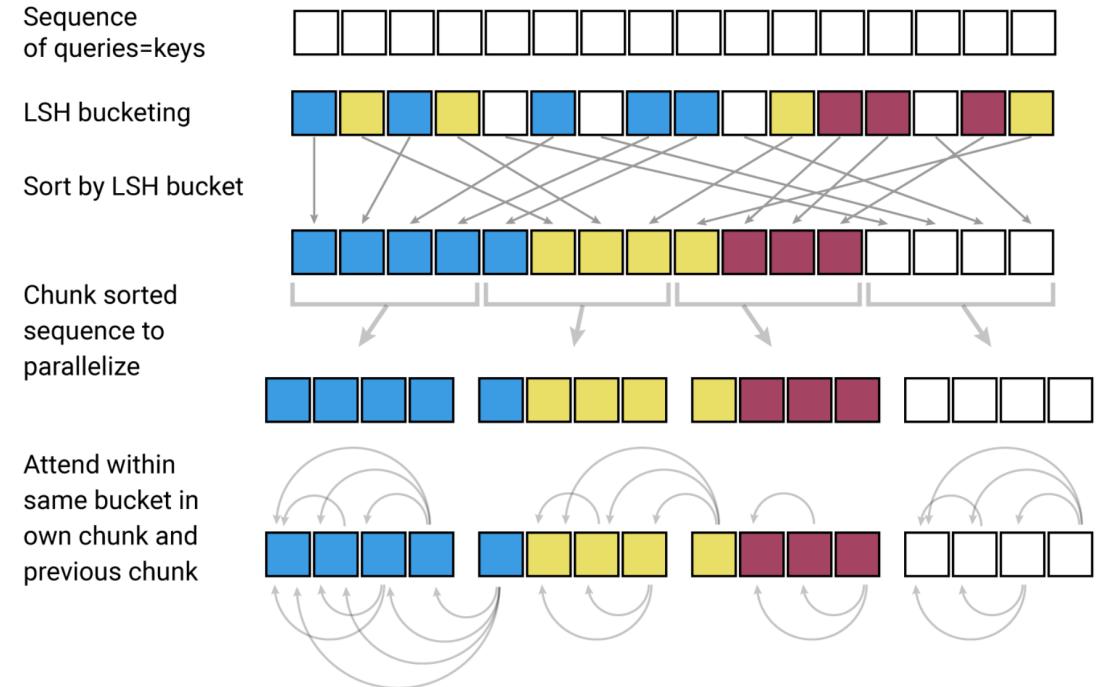
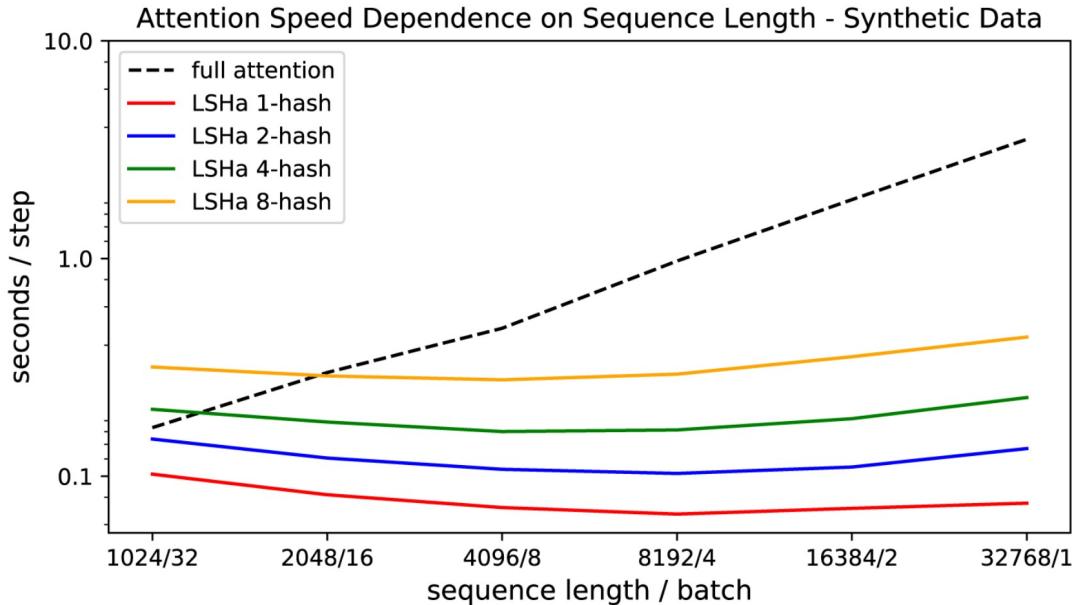
Improving Quadratic Self-Attention

- Linformer (Wang et al., 2020): Approximate self-attention with low rank matrix, which reduces time and memory costs to $O(n)$



Improving Quadratic Self-Attention

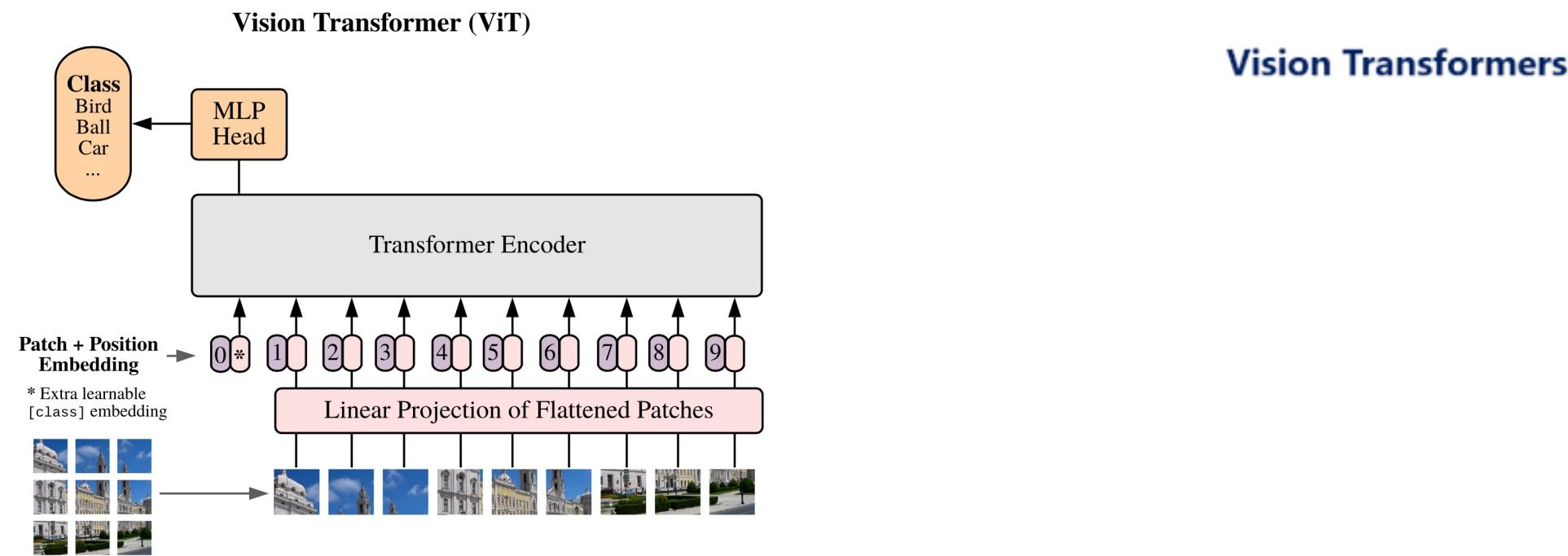
- Reformer (Kitaev et al., 2020): Replace dot-product with locality sensitive hashing; use reversible residual networks



Visual Transformer (ViT)

Dosovitskiy et al., 2020

- Vision Transformer (ViT) is a transformer designed for vision tasks



Reference

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.
- Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.
- Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.
- Kitaev, N., Kaiser, Ł., & Levskaya, A. (2020). Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*.
- Shen, S., Yao, Z., Gholami, A., Mahoney, M., & Keutzer, K. (2020, November). Powernorm: Rethinking batch normalization in transformers. In *International conference on machine learning* (pp. 8741-8751). PMLR.