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

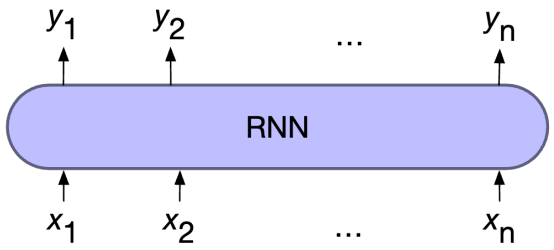
Overview

- Motivation
- Transformer Model
- Achievements and Drawbacks

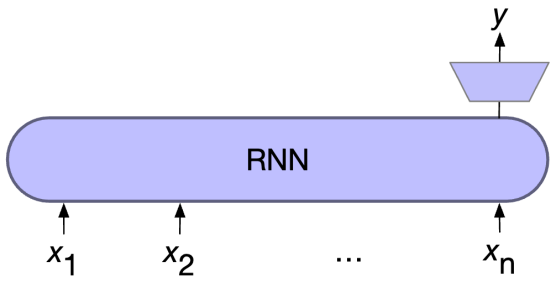
Overview

- **Motivation**
  - **RNNs, Encoder-Decoder, and Attention**
- Transformer Model
- Achievements and Drawbacks

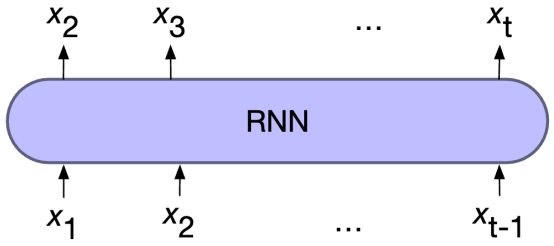
Common RNN Architecture for NLP



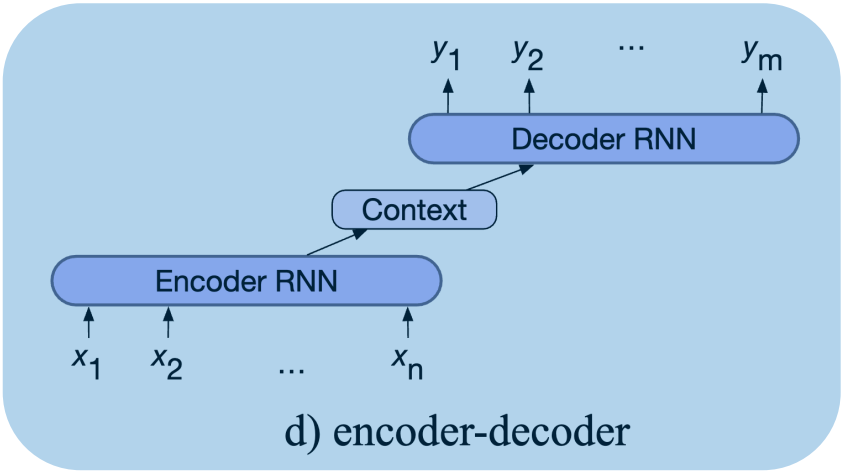
a) sequence labeling



b) sequence classification



c) language modeling



d) encoder-decoder

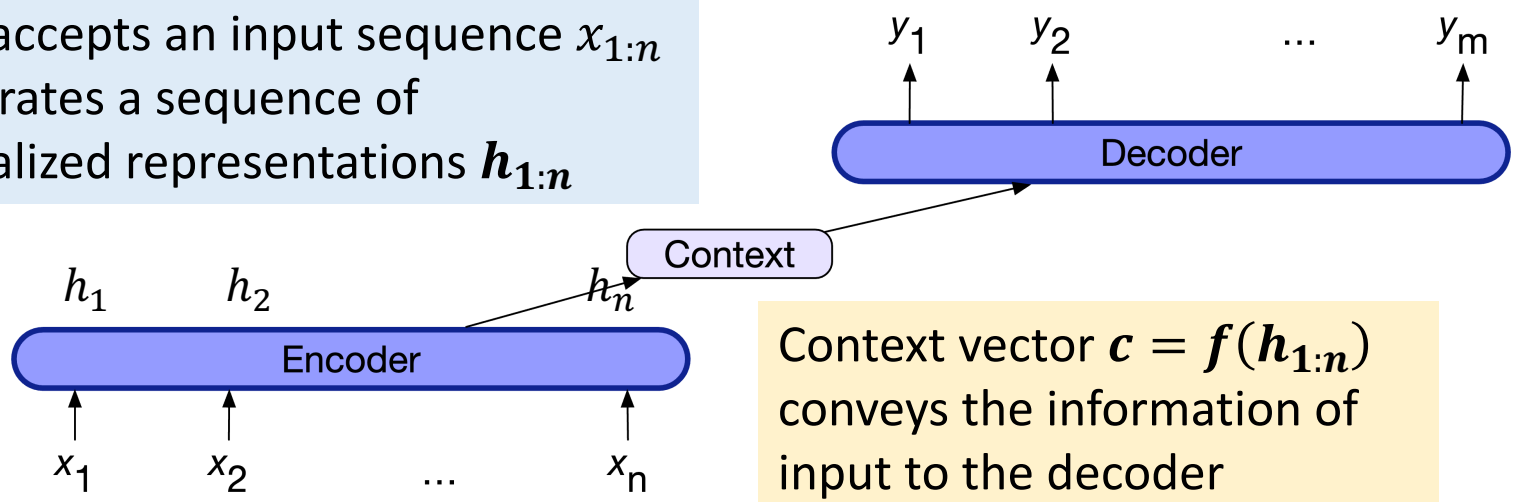
Encoder-decoder: also called **sequence-to-sequence** network

- Take an input sequence and translate it to an output sequence of different length
- Applications: summarization, question answering, dialogue, and particularly popular 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 $\boldsymbol{h_{1:n}}$



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

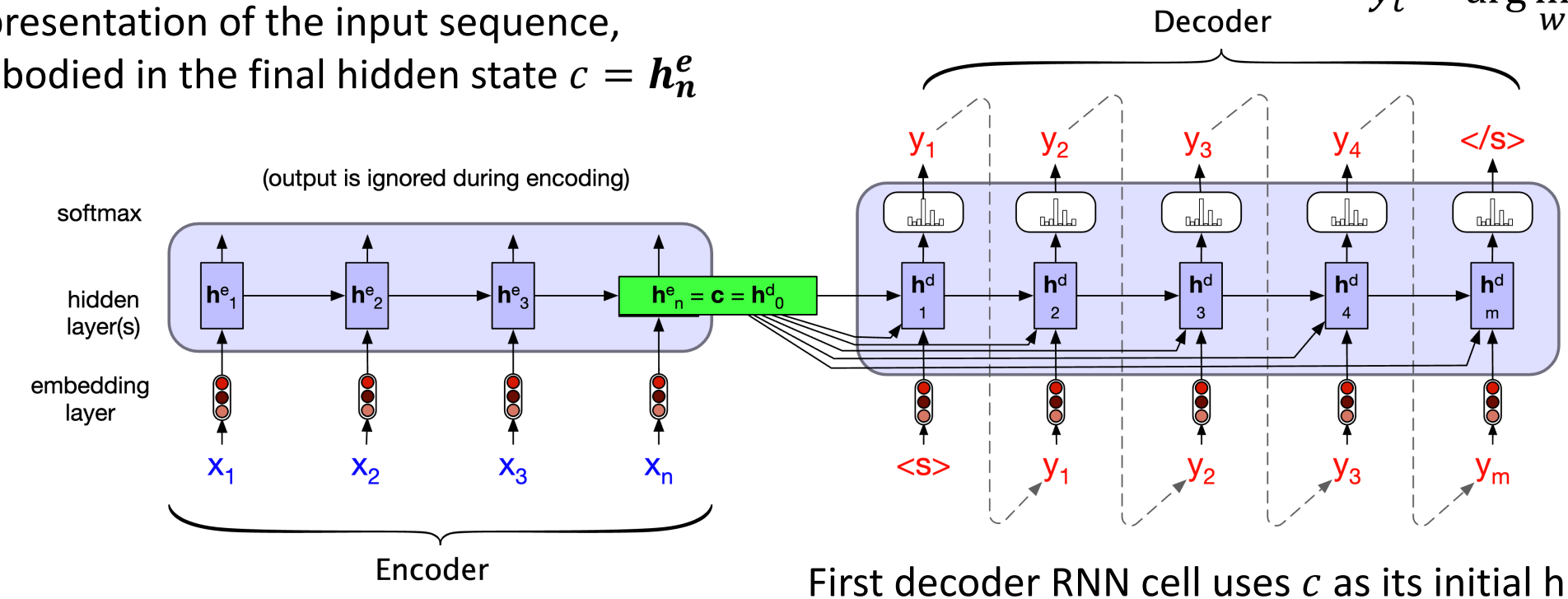
Decoder accepts c as input and generate a sequence of new representations $\boldsymbol{h'_{1:m}}$, from which an output sequence $y_{1:m}$ is obtained

Encoder-Decoder RNN for Machine Translation

$$\boldsymbol{h_t^d} = g\left(y_{t-1}, \boldsymbol{h_{t-1}^d}, \boldsymbol{c}\right) \qquad \hat{y}_t = \text{softmax}\left(f\left(\boldsymbol{h_t^d}\right)\right)$$

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

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

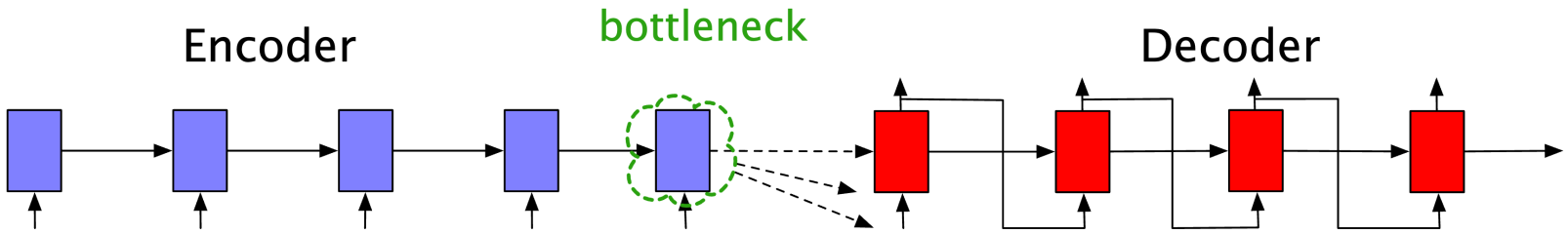


First decoder RNN cell uses c as its initial hidden state h_0^d

c is made available to all time steps to preserve its influence

Attention in Encoder-Decoder

- **Bottleneck problem**: the last hidden state $\boldsymbol{h_n^e}$ 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

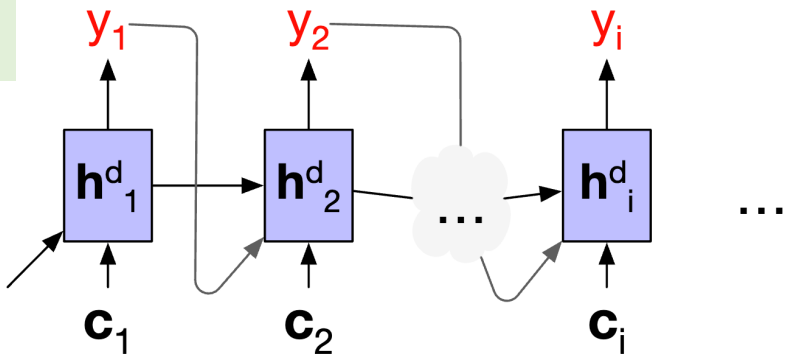


- **Attention mechanism** is a solution: allowing the decoder to get information from *all* the hidden states of encoder, not just the last one
- **Idea**: Create the context vector $\boldsymbol{c}$ by taking a weighted sum of all the encoder states $\boldsymbol{h_1^e, \ldots, h_n^e}$
- The weights attend 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 $\boldsymbol{c}$ with a *dynamically* derived one $\boldsymbol{c}_i$, which is different for each decoding step i

Decoder



The context used for decoding at step 1 ($\boldsymbol{c_1}$) is different from the one for step 2 ($\boldsymbol{c_2}$)

Hidden state used for decoding at step i:

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

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

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

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

Measures relevance by similarity between vectors

Dot-Product Attention

$$\text{score}\left(\boldsymbol{h}_{i-1}^{d}, \boldsymbol{h}_{j}^{e}\right) = \boldsymbol{h}_{i-1}^{d} \cdot \boldsymbol{h}_{j}^{e}$$

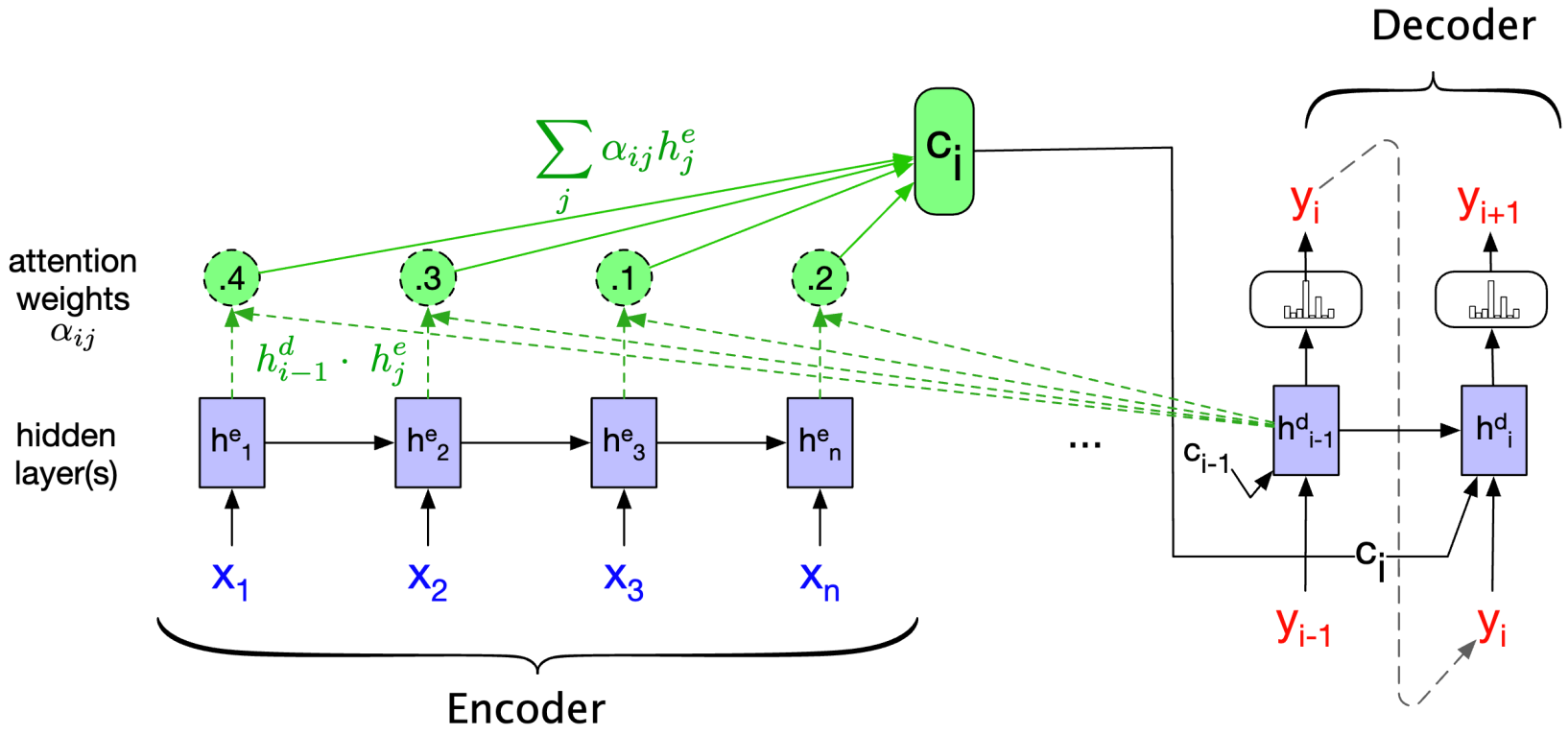
- Score from dot-product reflects the degree of similarity between two vectors
- Normalize the scores with a softmax to obtain vector of weights, α_{ij}, that tells the proportional relevance of each encoder state $\boldsymbol{h}_{j}^{e}$ to the decoder state $\boldsymbol{h}_{i-1}^{d}$:

$$\alpha_{ij} = \text{softmax}\left(\text{score}\left(\boldsymbol{h}_{i-1}^{d}, \boldsymbol{h}_{j}^{e}\right)\right) = \frac{\exp\left(\text{score}\left(\boldsymbol{h}_{i-1}^{d}, \boldsymbol{h}_{j}^{e}\right)\right)}{\sum_{k} \exp\left(\text{score}\left(\boldsymbol{h}_{i-1}^{d}, \boldsymbol{h}_{k}^{e}\right)\right)}$$

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

$$\boldsymbol{c}_i = \sum_{j} \alpha_{ij} \boldsymbol{h}_{j}^{e}$$

Encoder-Decoder with Attention



What if $\boldsymbol{h}_{i-1}^{d}, \boldsymbol{h}_{j}^{e}$ are of different dimensions?

Need a more sophisticated scoring function, such as bilinear:

$$\text{score}\left(\boldsymbol{h}_{i-1}^{d}, \boldsymbol{h}_{j}^{e}\right) = \boldsymbol{h}_{i-1}^{d} \boldsymbol{W}_{\boldsymbol{s}} \boldsymbol{h}_{j}^{e}$$

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

Overview

- Motivation
- **Transformer Model**
  - **Self-Attention; Multihead Attention**
  - **Transformer Blocks**
  - **Positional Embedding**
  - **Language Modeling Head**
  - **Training**
- Achievements and Drawbacks

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 ⇒ 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)

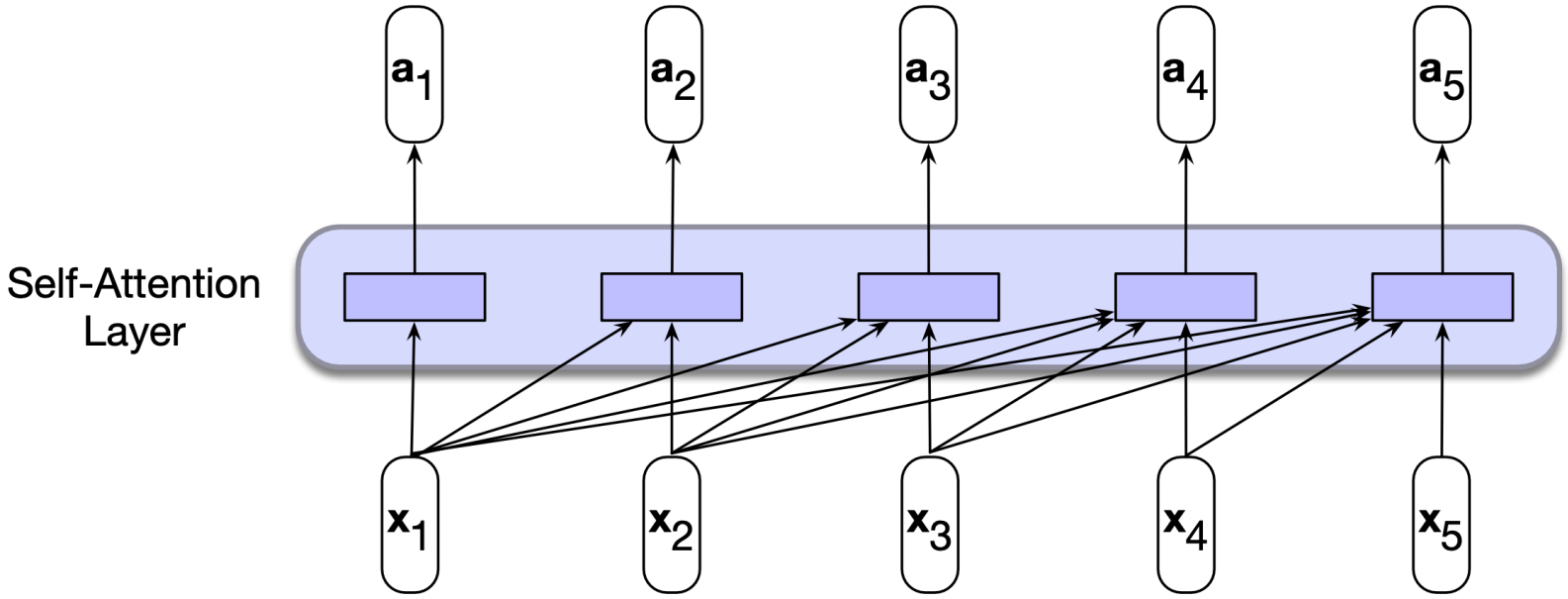
Self-Attention: Motivation

- 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 $x_1, \ldots, x_n$

to output sequence of the same length $a_1, \ldots, a_n$

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

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

- Recall attention in encoder-decoder RNNs: the core 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**:

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

② Normalize to a probability distribution:

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

③ Compute output a_i as the weighted average: $\boldsymbol{a}_i = \sum_{j \leq i} \alpha_{ij} \boldsymbol{x}_j$

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 $\boldsymbol{W^Q}$, $\boldsymbol{W^K}$, and $\boldsymbol{W^V}$ to project each input vector $\boldsymbol{x}_i$ into a representation of its role as a key, query, and value:

$$\boldsymbol{q}_i = \boldsymbol{x}_i \boldsymbol{W}^Q; \quad \boldsymbol{k}_i = \boldsymbol{x}_i \boldsymbol{W}^K; \quad \boldsymbol{v}_i = \boldsymbol{x}_i \boldsymbol{W}^V$$

Dimensions:

input $\boldsymbol{x}_i$, output $\boldsymbol{y}_i$, intermediate $\boldsymbol{a}_i$ $\in \mathbb{R}^d$

$\boldsymbol{q}_i$, $\boldsymbol{k}_i$ $\in \mathbb{R}^{d_k}$

$\boldsymbol{v}_i \in \mathbb{R}^{d_v}$

$\boldsymbol{W}^Q \in \mathbb{R}^{d \times d_k}$

$\boldsymbol{W}^K \in \mathbb{R}^{d \times d_k}$

$\boldsymbol{W}^V \in \mathbb{R}^{d \times d_v}$

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

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

- To compute score between a current focus of attention $\boldsymbol{x}_i$, and a preceding element $\boldsymbol{x}_j$, use the dot-product between its **query** vector $\boldsymbol{q}_i$ and **key** vector $\boldsymbol{k}_j$:

$$\text{score}(x_i, x_j) = \boldsymbol{q}_i \cdot \boldsymbol{k}_j$$

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

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

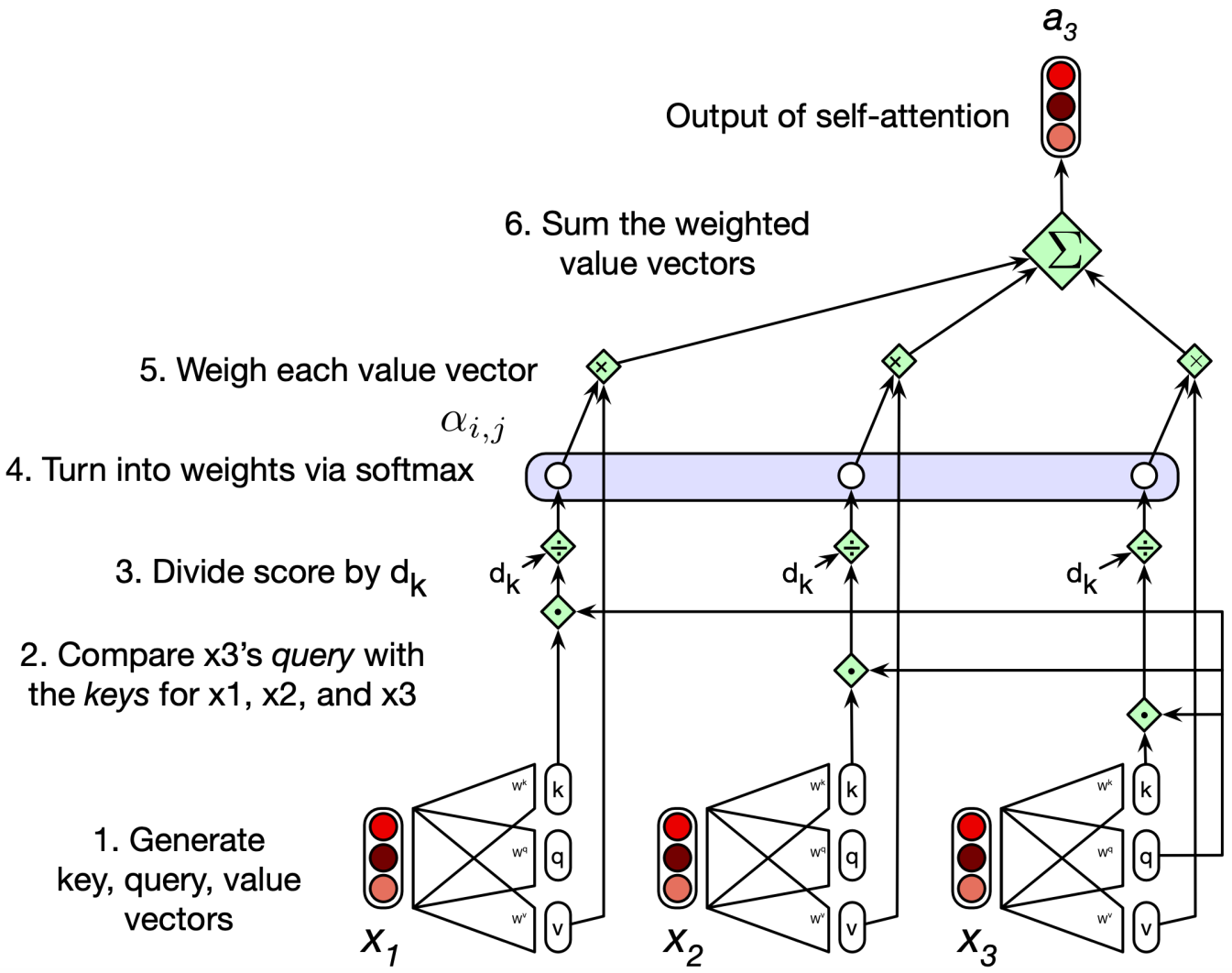
Note: $d = 512,\ d_k = d_v = 64$
How do 64-d vectors (v_j) end up in 512-d (a_i)? -- Concatenation (explained in multihead attention)

$$\boldsymbol{a}_i = \sum_{j \le i} \alpha_{ij} \boldsymbol{v}_j$$

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

$$\text{score}(x_i, x_j) = \frac{\boldsymbol{q}_i \cdot \boldsymbol{k}_j}{\sqrt{d_k}}$$

Self-Attention Final Version



$$\boldsymbol{q}_i = \boldsymbol{x}_i \boldsymbol{W}^Q$$
$$\boldsymbol{k}_i = \boldsymbol{x}_i \boldsymbol{W}^K$$
$$\boldsymbol{v}_i = \boldsymbol{x}_i \boldsymbol{W}^V$$

$$\text{score}(x_i, x_j) = \frac{\boldsymbol{q}_i \cdot \boldsymbol{k}_j}{\sqrt{d_k}}$$

$$\alpha_{ij} = \text{softmax}\left(\text{score}(\boldsymbol{x}_i, \boldsymbol{x}_j)\right)$$
$$\forall j \le i$$

$$\boldsymbol{a}_i = \sum_{j \le i} \alpha_{ij} \boldsymbol{v}_j$$

Parallelizing Self-Attention using Matrices

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

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}^{Q}; \quad \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^{K}; \quad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^{V}$$

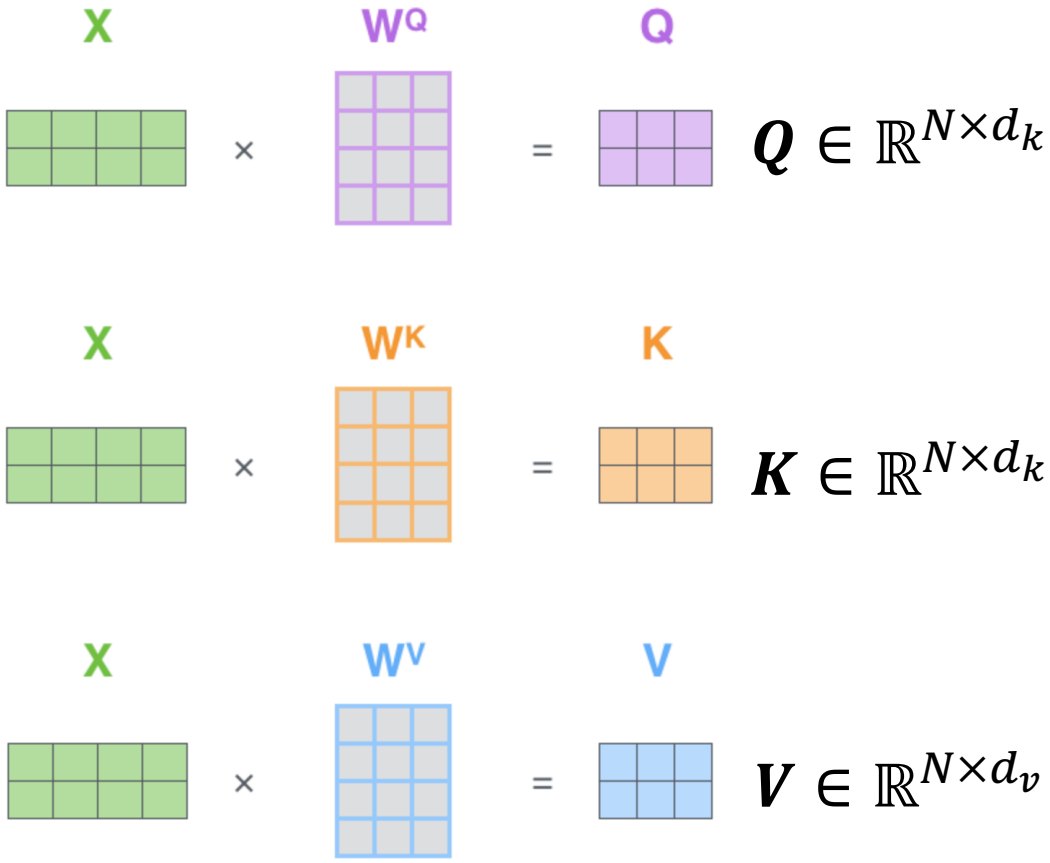
- Resulting in three matrices: $\boldsymbol{Q} \in \mathbb{R}^{N \times d_k}$, $\boldsymbol{K} \in \mathbb{R}^{N \times d_k}$, and $\boldsymbol{V} \in \mathbb{R}^{N \times d_v}$
- Then the attention score can be computed by $\boldsymbol{Q}\boldsymbol{K}^{\top}$, resulting in $N \times N$ scores
- The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

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

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

Parallelizing Self-Attention using Matrices

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



$\boldsymbol{Q} \in \mathbb{R}^{N \times d_k}$

$\boldsymbol{K} \in \mathbb{R}^{N \times d_k}$

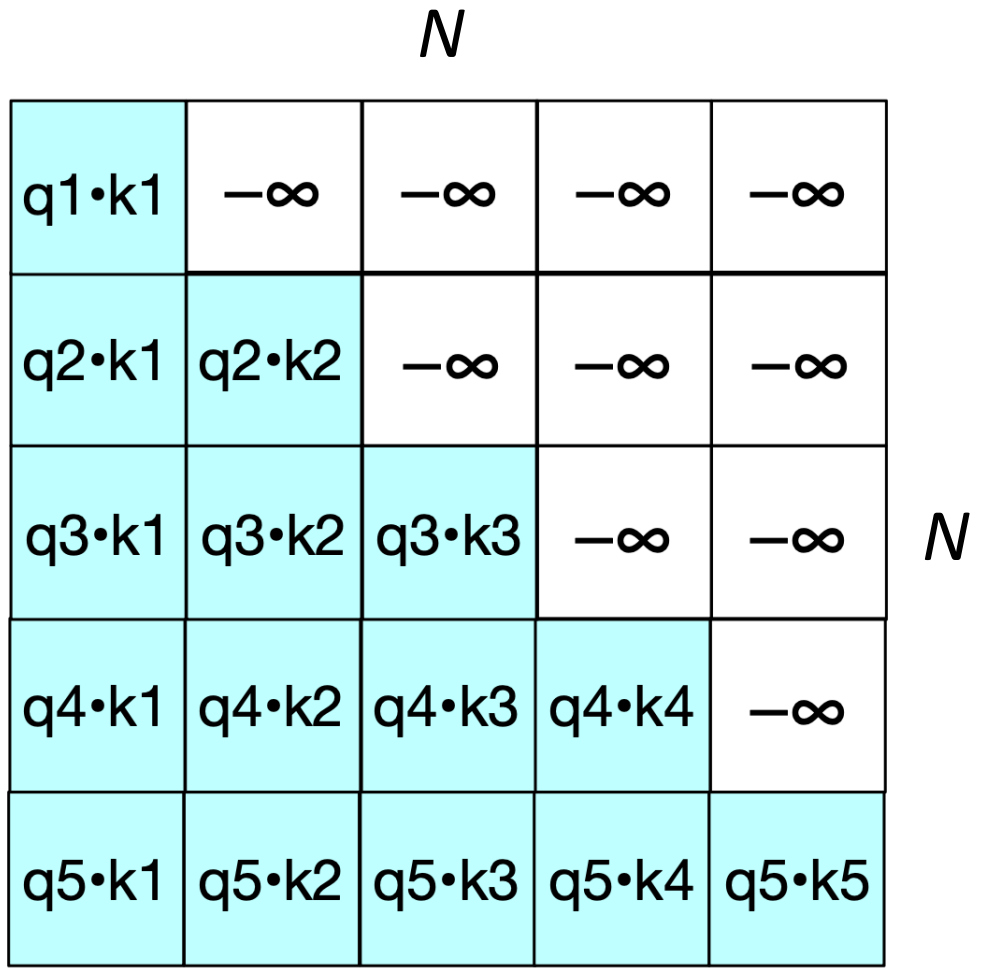
$\boldsymbol{V} \in \mathbb{R}^{N \times d_v}$

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

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

Masking Out the Future

- The self-attention so far has a problem: $\boldsymbol{QK}^\top$ results in a score for each query to every key, including those that come after the query
- Inappropriate for language modeling: guessing the next work 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



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: $\boldsymbol{W}_i^Q \in \mathbb{R}^{N\times d_k}$, $\boldsymbol{W}_i^K \in \mathbb{R}^{N\times d_k}$, and $\boldsymbol{W}_i^V \in \mathbb{R}^{N\times d_v}$
- Given input $\boldsymbol{X} \in \mathbb{R}^{N\times d}$,They produce $\boldsymbol{Q}_i \in \mathbb{R}^{N\times d_k}$, $\boldsymbol{K}_i \in \mathbb{R}^{N\times d_k}$, and $\boldsymbol{V}_i \in \mathbb{R}^{N\times d_v}$
- 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 $N\times hd_v$

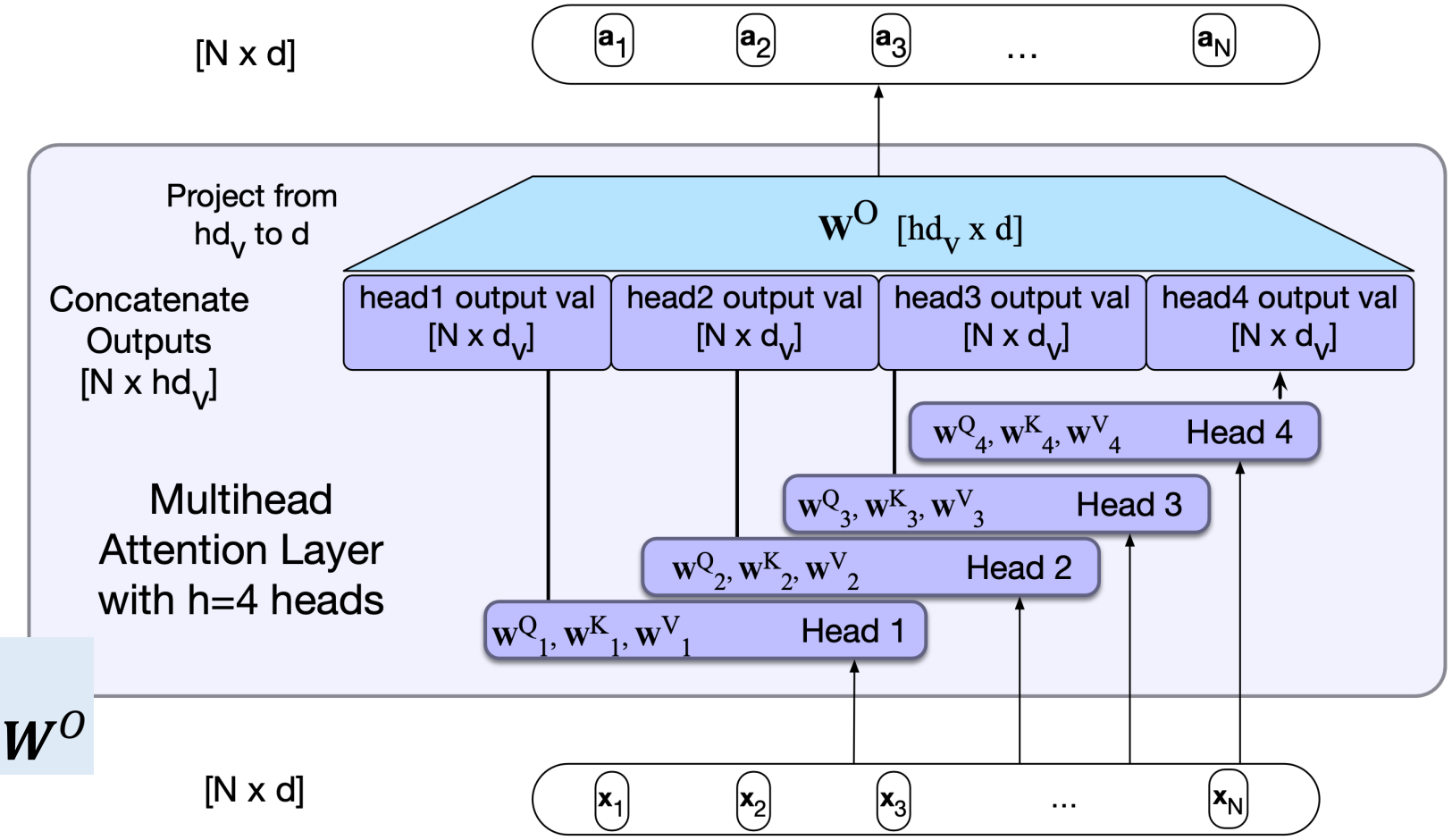
Multihead Attention

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

$$\boldsymbol{Q}_i = \boldsymbol{X}\boldsymbol{W}_i^Q$$
$$\boldsymbol{K}_i = \boldsymbol{X}\boldsymbol{W}_i^K$$
$$\boldsymbol{V}_i = \boldsymbol{X}\boldsymbol{W}_i^V$$

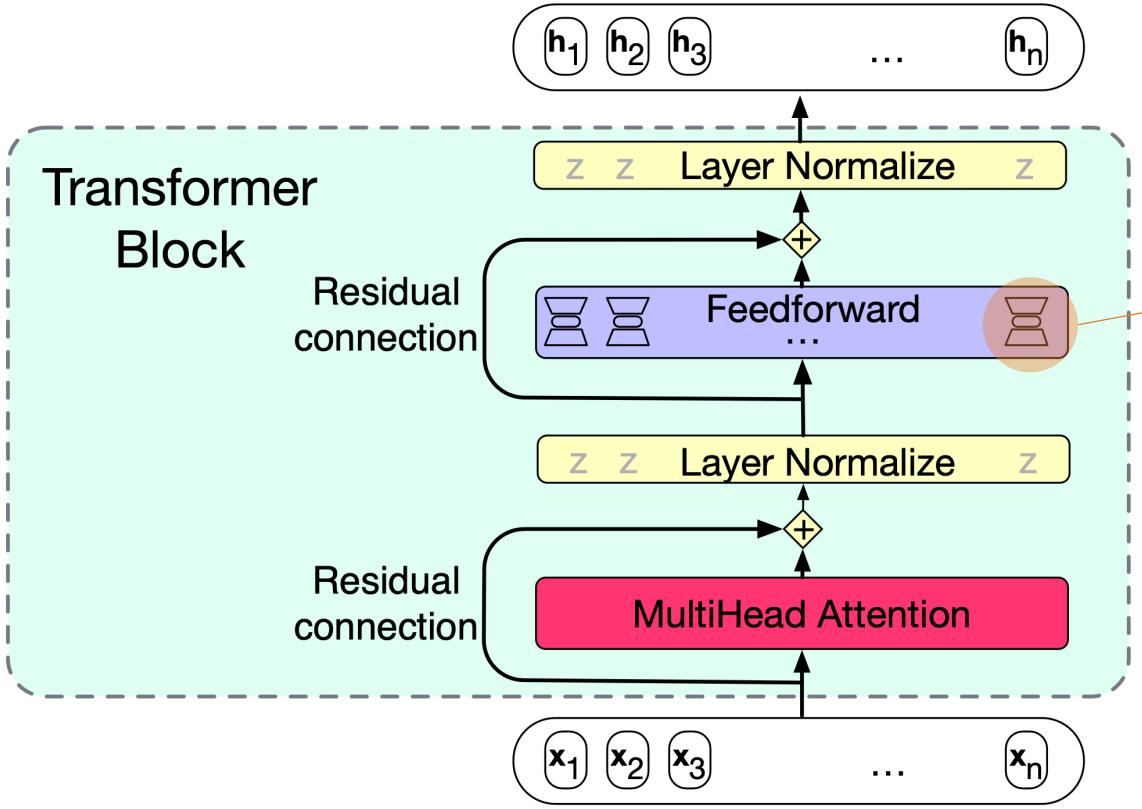
$$\mathbf{head}_i = \text{SelfAttention}(\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i)$$

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



Transformer Blocks

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

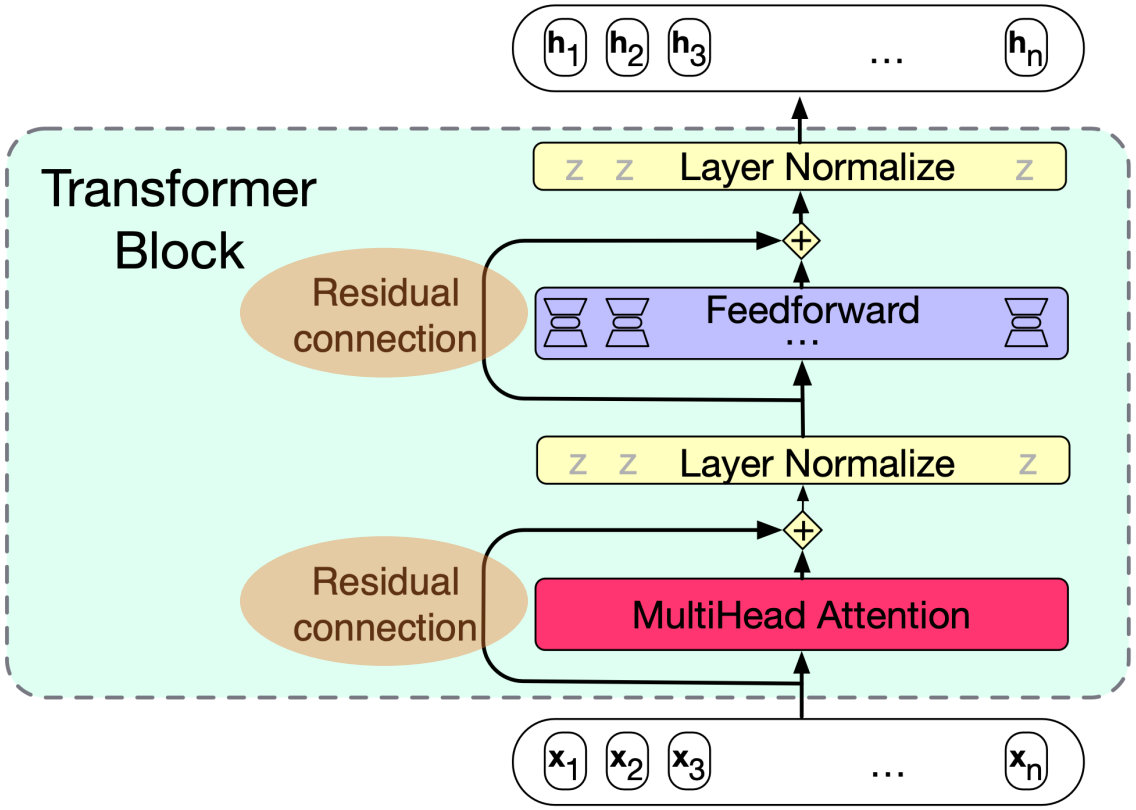


Feedforward layer contains N position-wise networks:

- 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

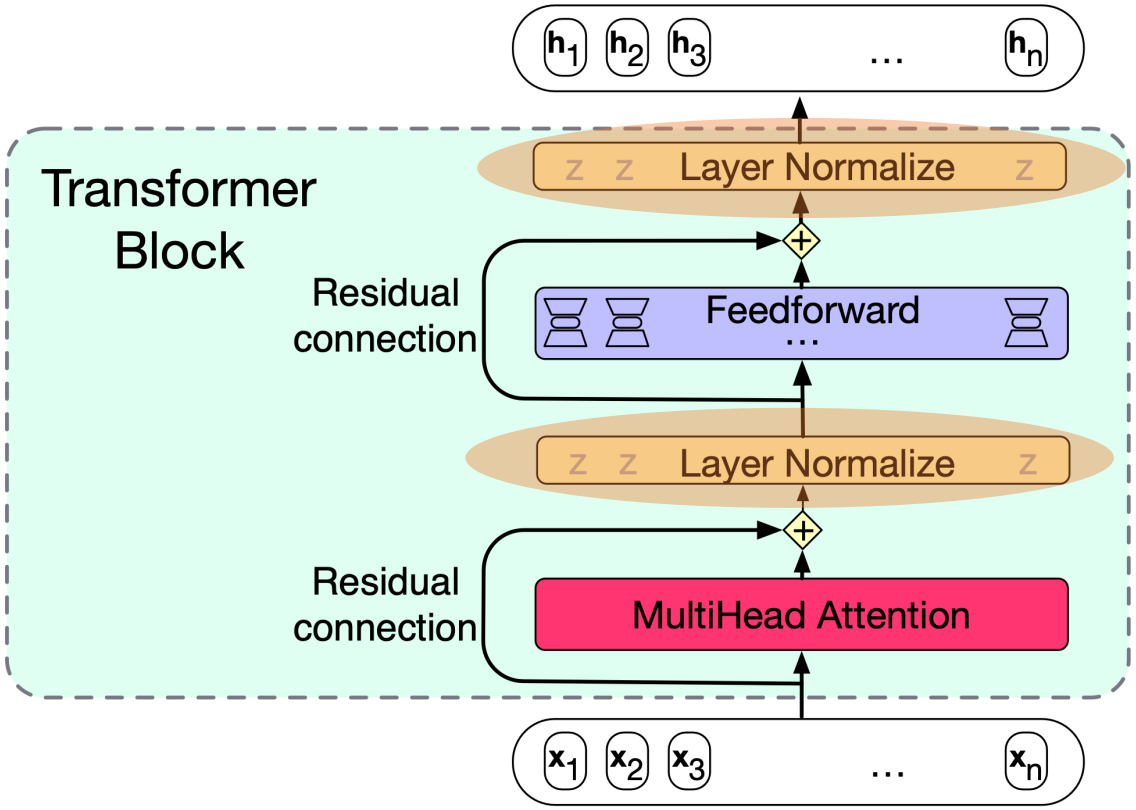
- **Residual connections** pass information from lower layer to higher layer without going through the intermediate layer



- 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, $\boldsymbol{x}$
Output: the vector normalized, $\hat{\boldsymbol{x}}$

First, compute the mean and standard deviation over the elements of $\boldsymbol{x}$:

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

Then, normalize by subtracting and dividing:

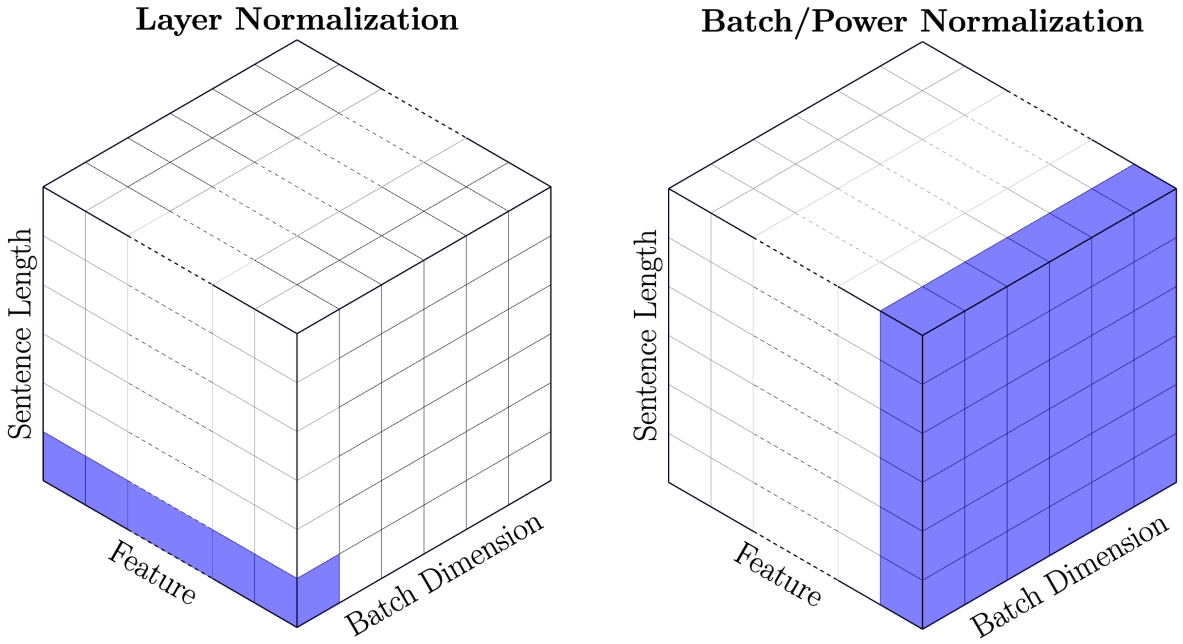
$$\hat{\boldsymbol{x}} = (\boldsymbol{x} - \mu)/\sigma$$

Finally, add learnable parameters γ, β

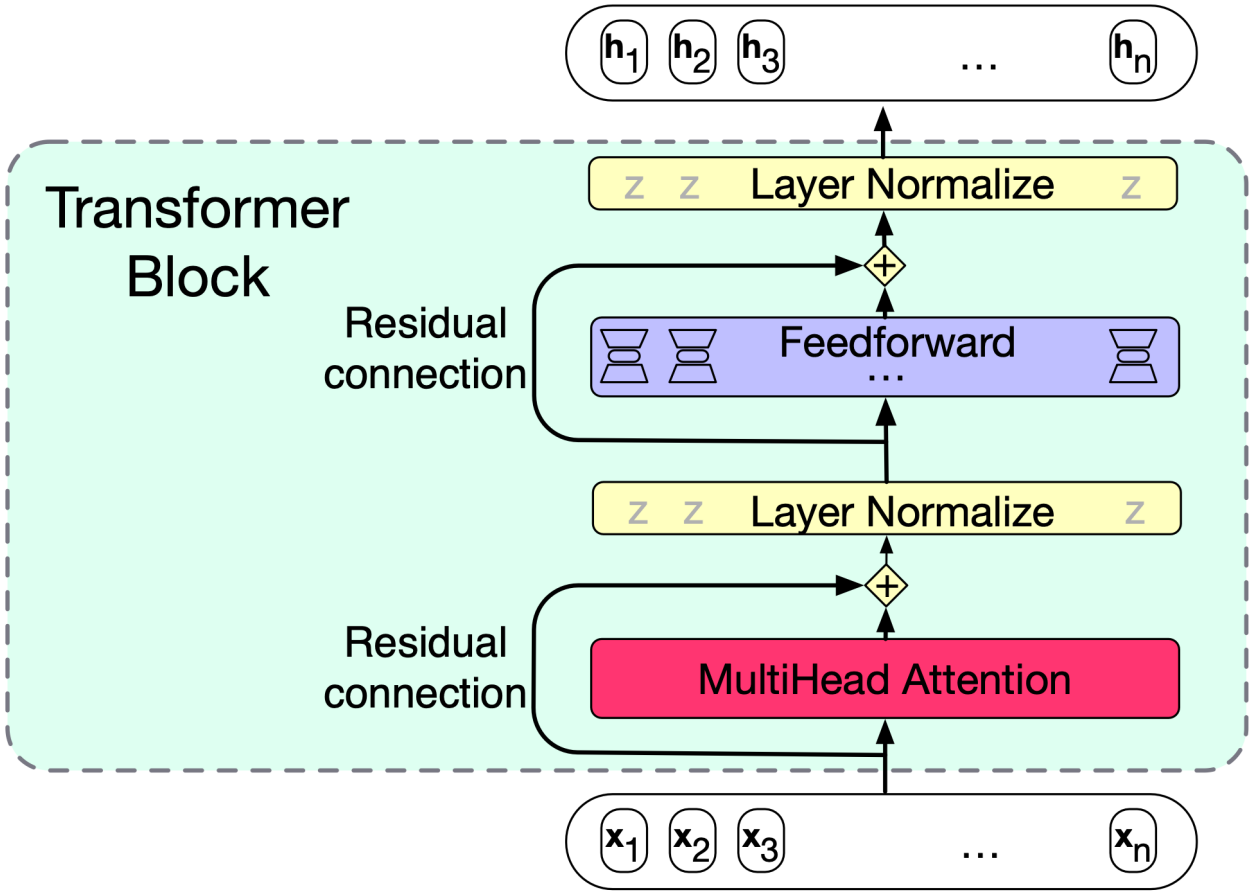
$$\text{LayerNorm} = \gamma\hat{\boldsymbol{x}} + \beta$$

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: $\boldsymbol{X}$, output: $\boldsymbol{H}$

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

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

---

break down:

Output $\boldsymbol{H}$

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

$$\boldsymbol{T}^5 = \boldsymbol{T}^4 + \boldsymbol{T}^3$$

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

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

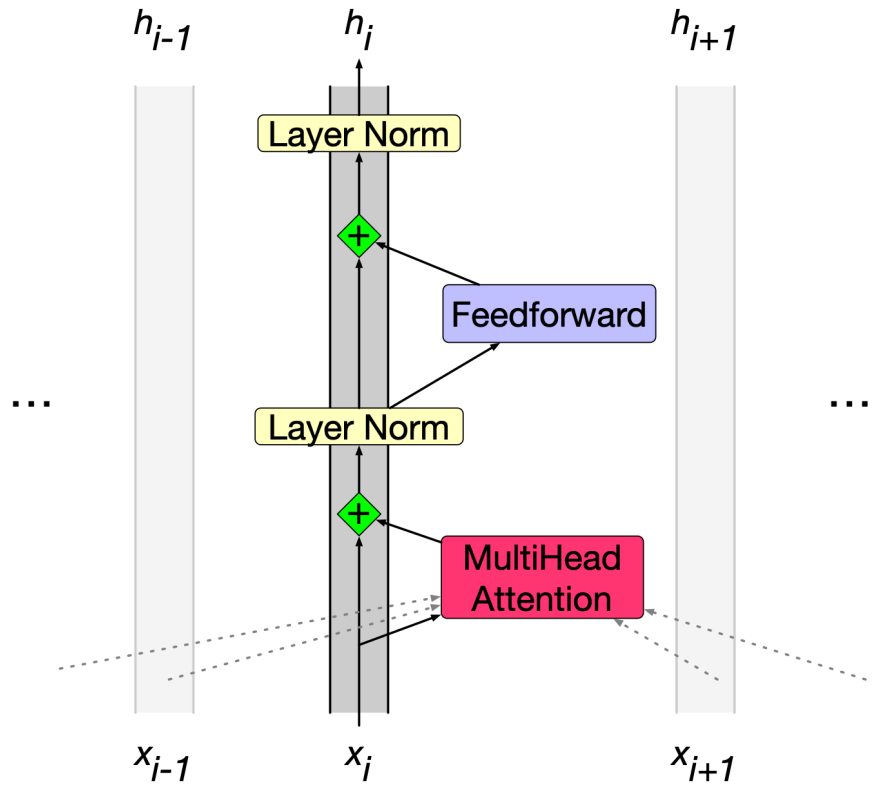
$$\boldsymbol{T}^2 = \boldsymbol{X} + \boldsymbol{T}^1$$

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

Input $\boldsymbol{X}$

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 $1 \times d$

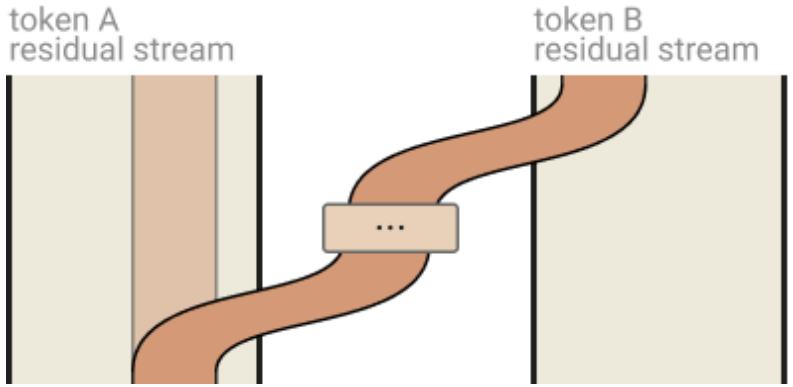
The layers are constantly copying information up from earlier embeddings

Other components add information into this constant stream:

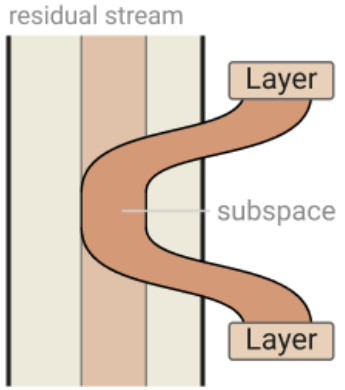
- Feedforward adds in a different view of earlier embedding
- Multihead attention adds information from other tokens

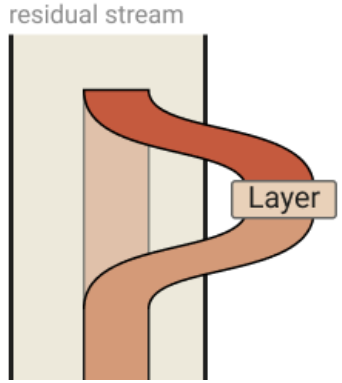
Residual Stream View of Transformer Block

- Attention heads move information from one stream to another



The residual stream is high dimensional, and can be divided into different subspaces.





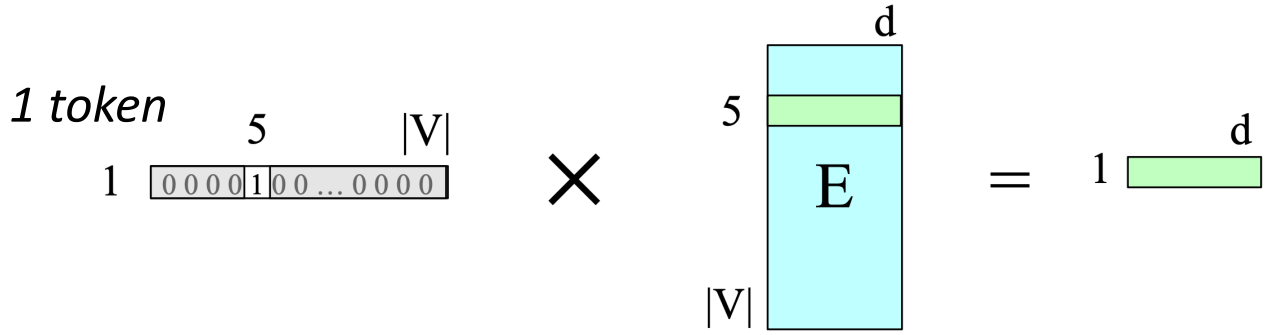
Layers can delete information from the residual stream by reading in a subspace and then writing the negative verison.

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

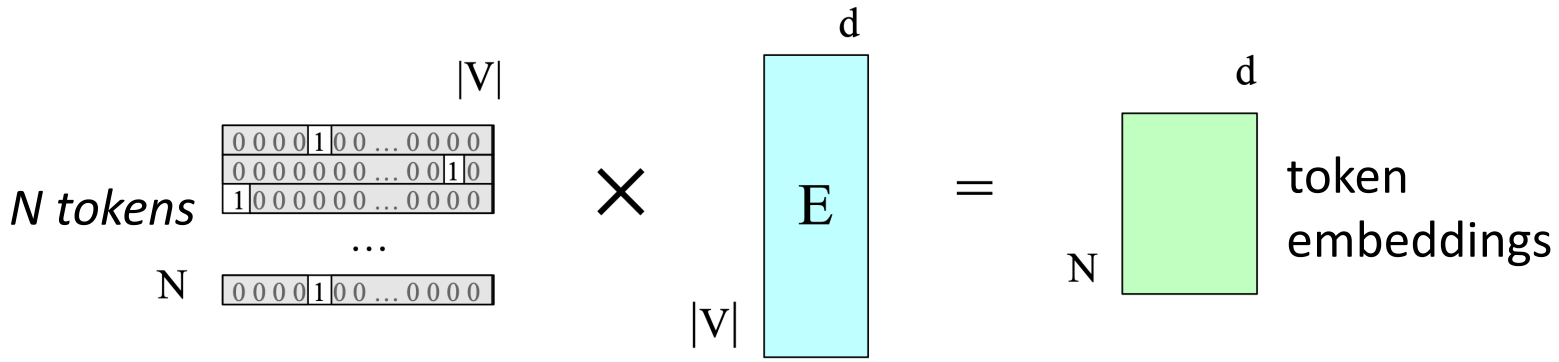
Input Embeddings for Token and Position

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

Token embedding:
One-hot vectors ⇒
dense vectors

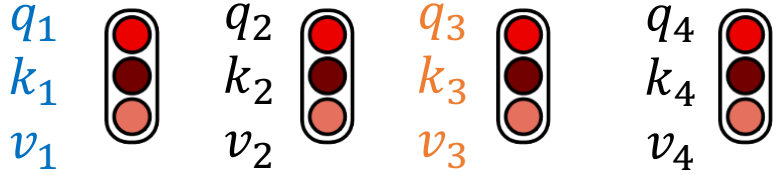


Problem: token embeddings are not position-dependent



Necessity of Positional Embedding

- Word order matters

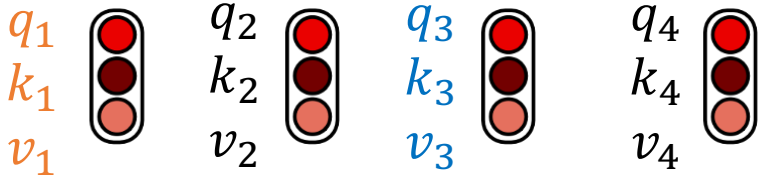


S1: 老王 被 老李 批评 了

Wang by Li criticized

S2: 老李 被 老王 批评 了

Li by Wang criticized



representation for "批评" in two sentences

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

II

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

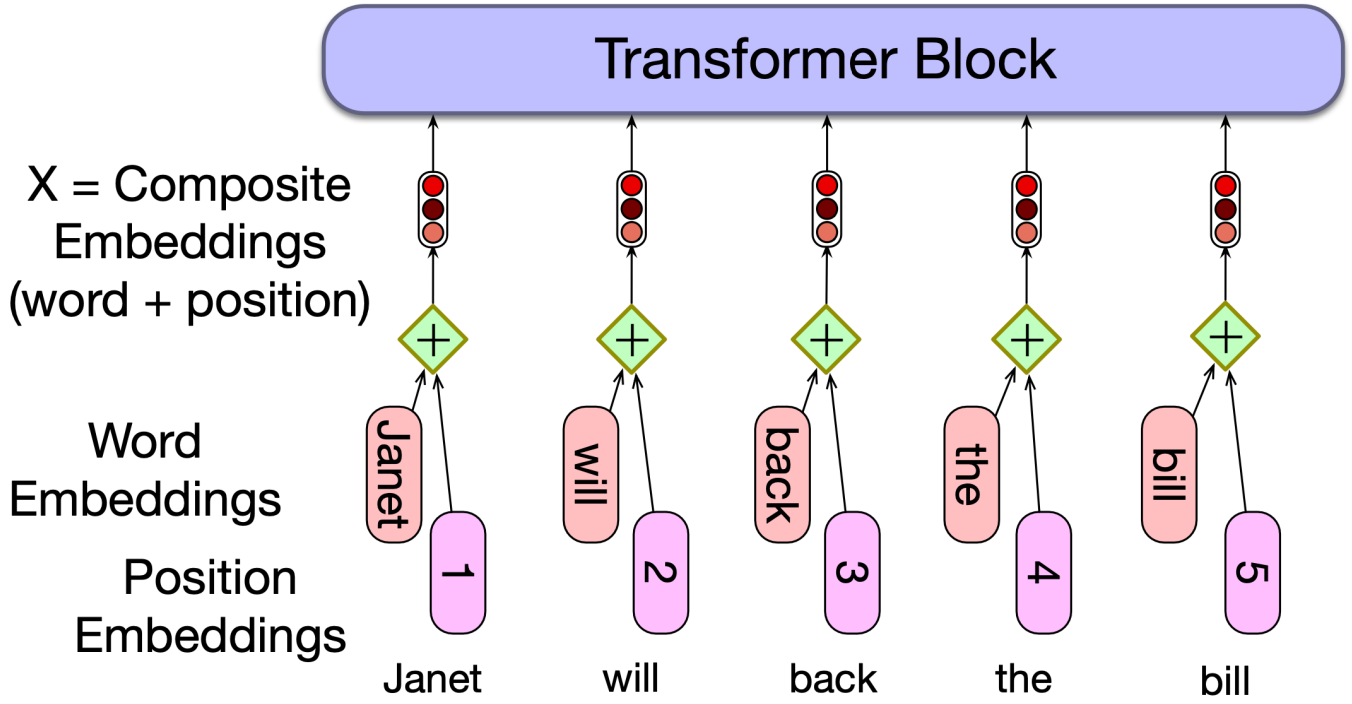
Positional Embedding: Methods

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

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

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

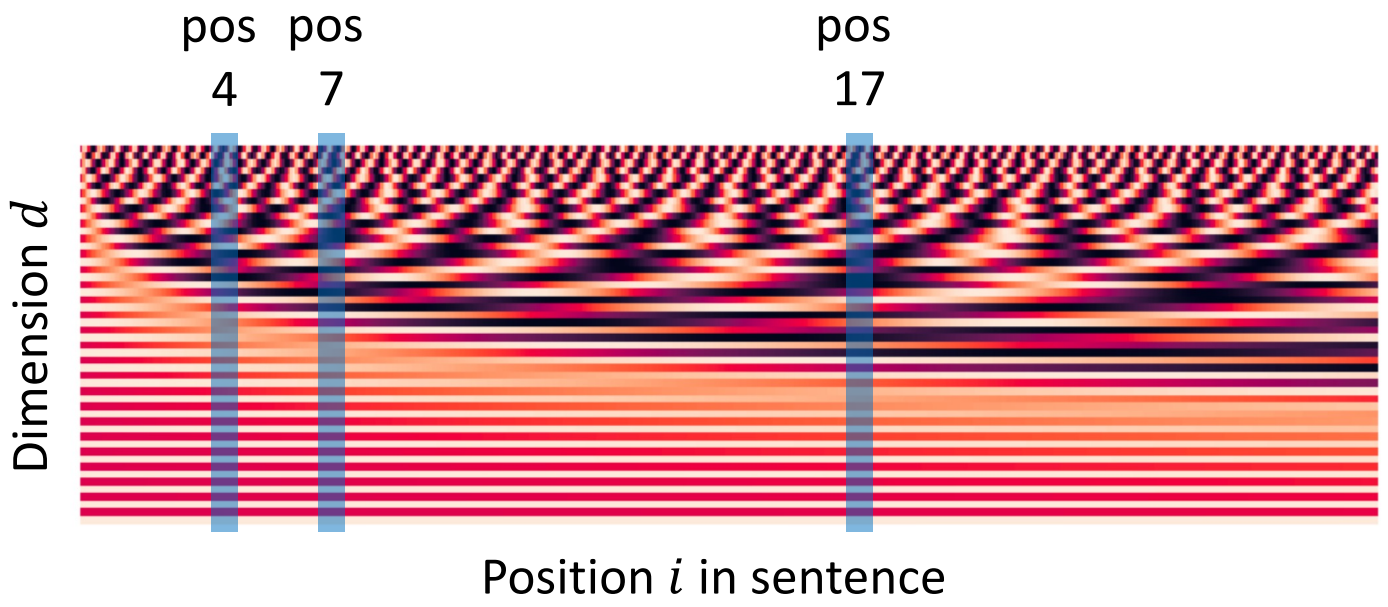
Drawback: Initial positions (small values) dominate the training data; later positions (large values) are poorly trained



Positional Embedding: Methods

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

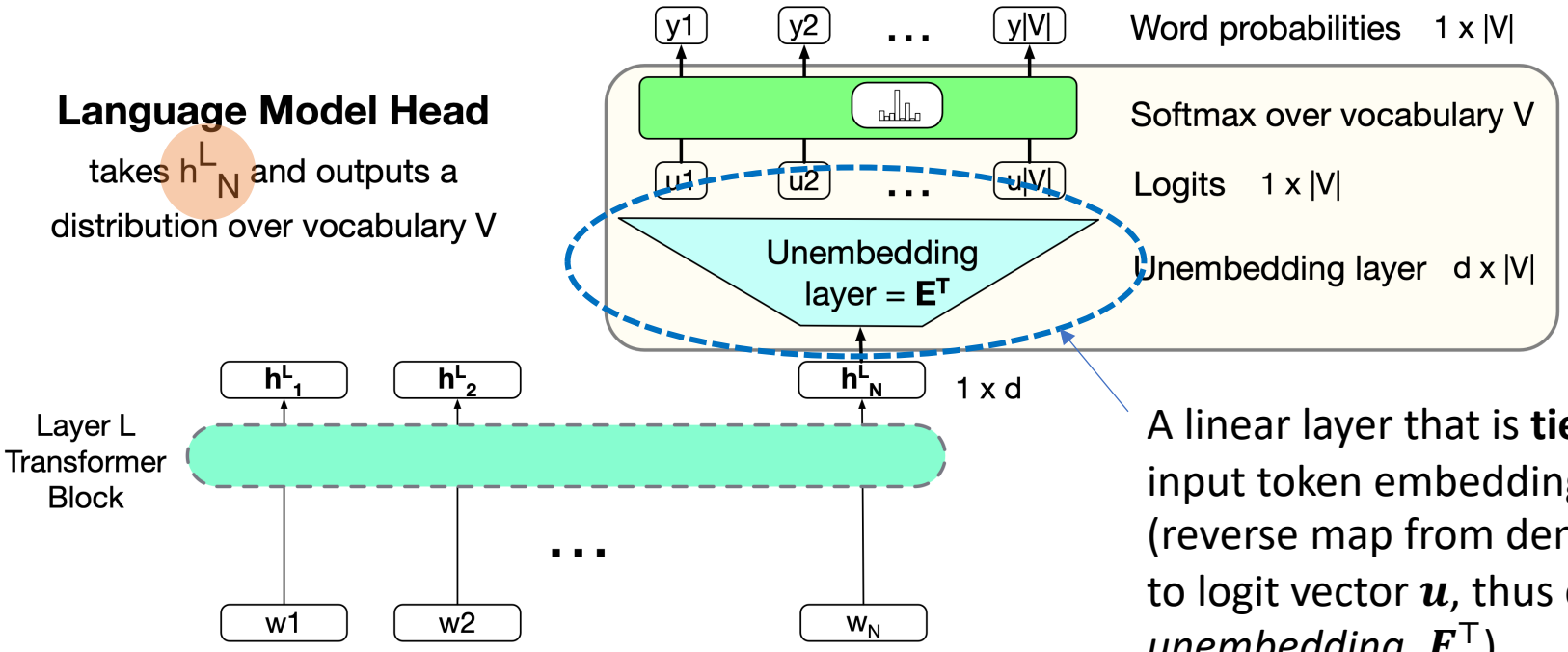
$$\boldsymbol{P}_i = \begin{bmatrix} \sin\left(i/10000^{2*1/d}\right) \\ \cos\left(i/10000^{2*1/d}\right) \\ \sin\left(i/10000^{2*2/d}\right) \\ \cos\left(i/10000^{2*2/d}\right) \\ \vdots \\ \sin\left(i/10000^{2*\frac{d}{2}/d}\right) \\ \cos\left(i/10000^{2*\frac{d}{2}/d}\right) \end{bmatrix} \Bigg\} \begin{matrix} d \\ (= 512) \end{matrix}$$



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

Language Modeling Head

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



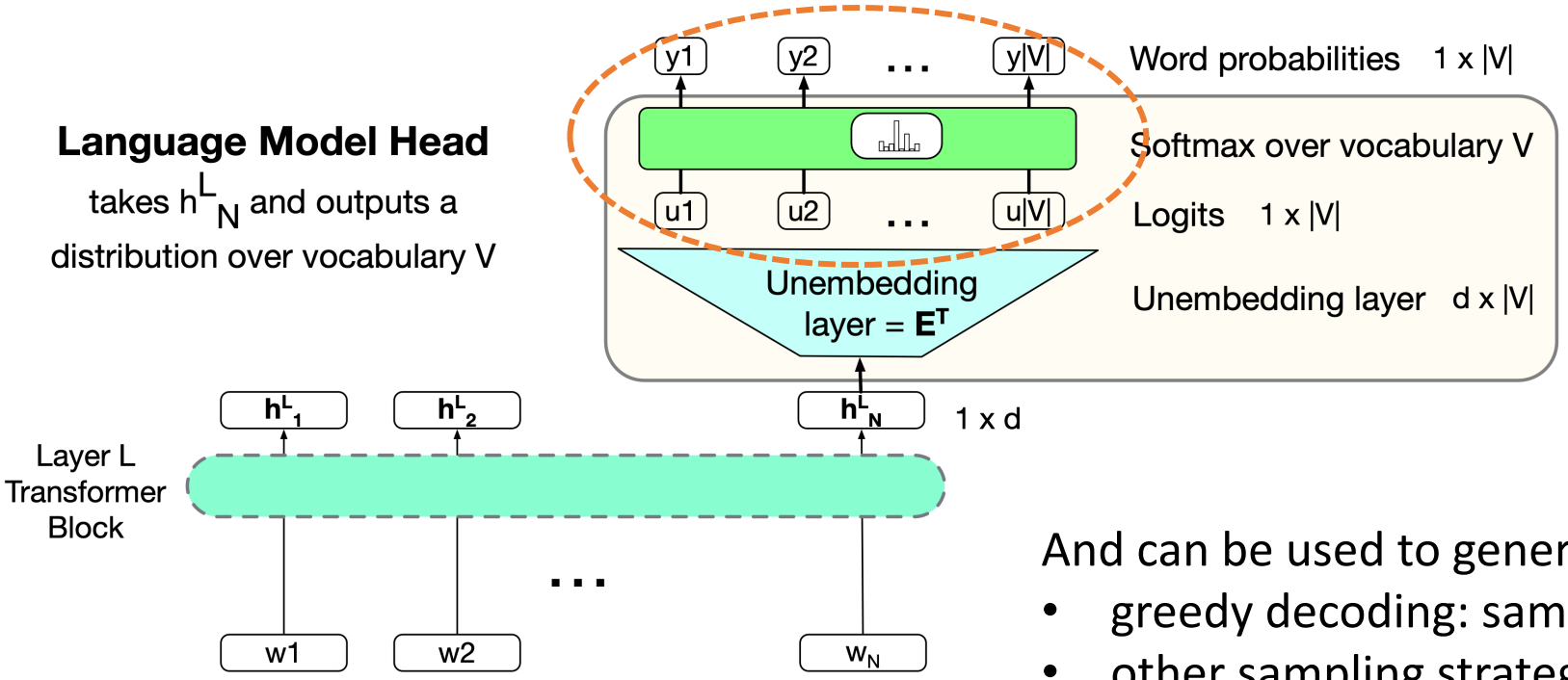
Take the output of the last token position N from the last layer L, $\boldsymbol{h}^L_N$, to predict the next word

A linear layer that is **tied** to the input token embedding matrix E (reverse map from dense vector $\boldsymbol{h}$ to logit vector $\boldsymbol{u}$, thus called *unembedding*, $\boldsymbol{E}^\top$)

$$\boldsymbol{u} = \boldsymbol{h}^L_N \boldsymbol{E}^\top$$

Language Modeling Head

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



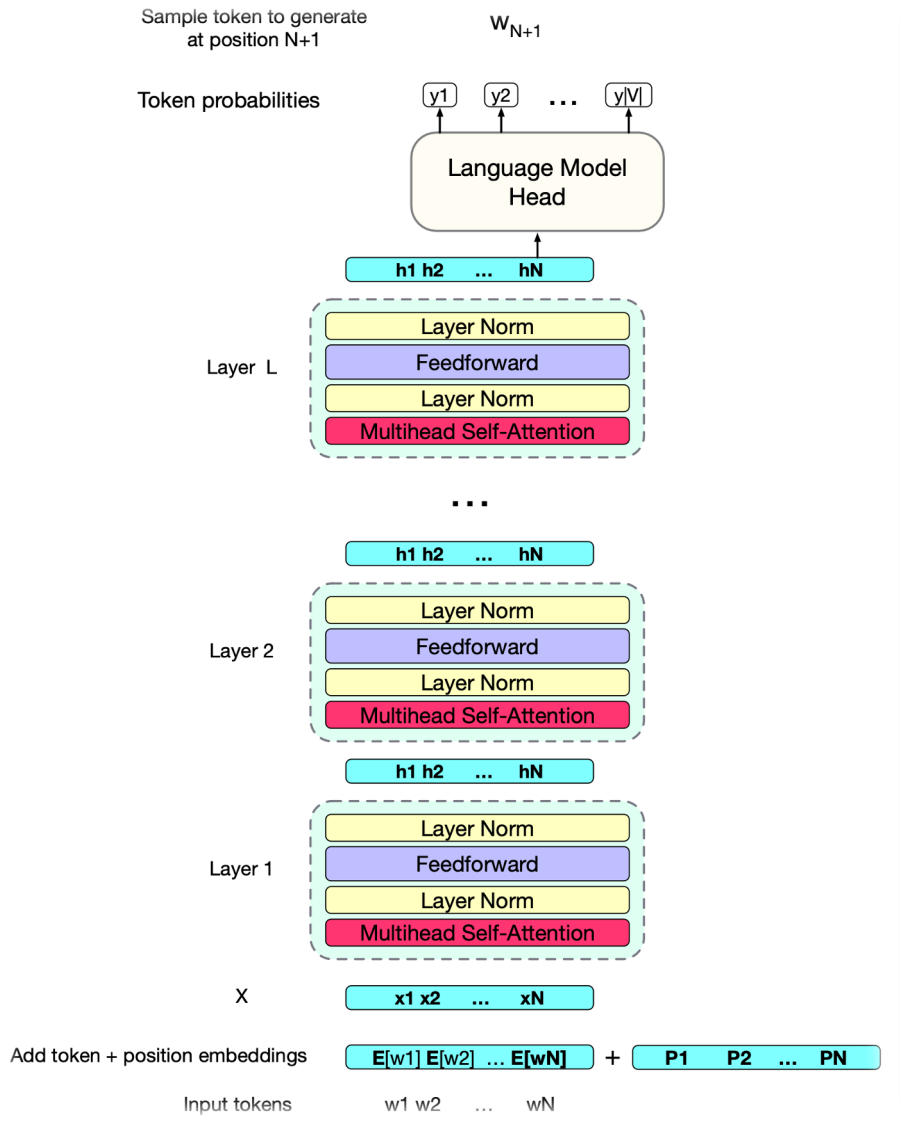
$$\boldsymbol{y} = \text{softmax}(\boldsymbol{u})$$

After being trained, these $\boldsymbol{y}$ distribution can be used to estimate the probability of a given text

And can be used to generate text, by sampling from $\boldsymbol{y}$

- greedy decoding: sample the highest prob word
- other sampling strategies: top-k, top-p, temperature

Full Transformer Model (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: $\boldsymbol{X} = \boldsymbol{E}_w + \boldsymbol{P}_i$

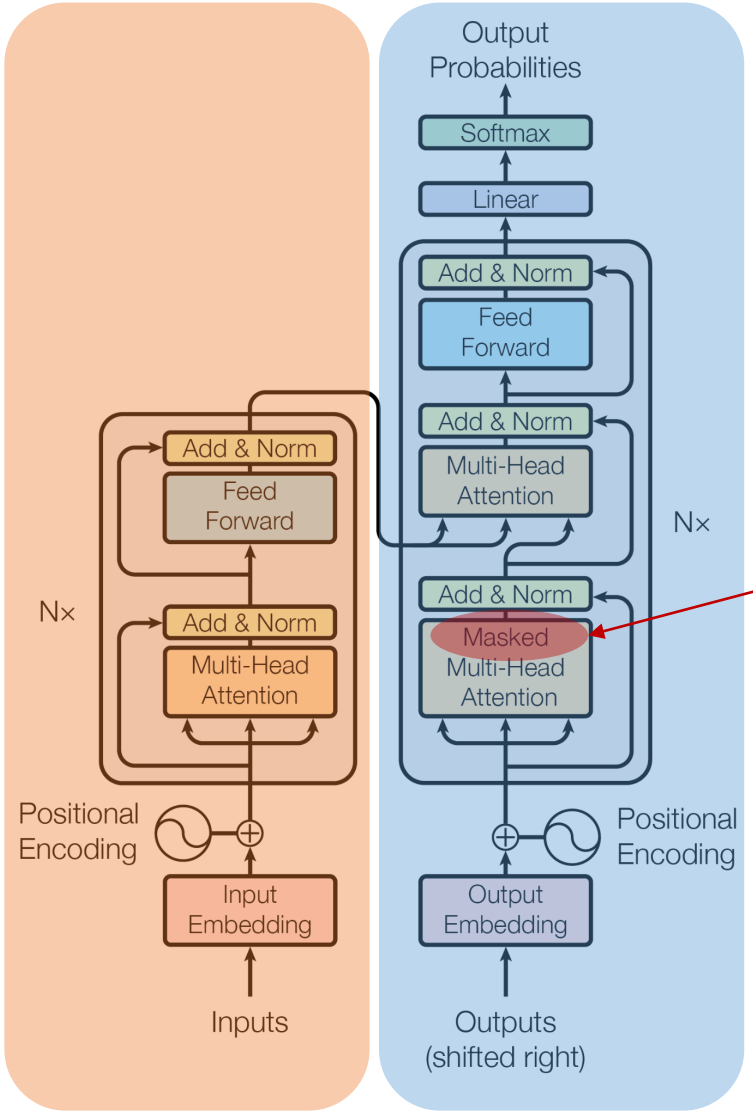
The input to all the other layers is the output $\boldsymbol{H}$ from the layer below

Terminology:

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

Transformer Model (encoder-decoder)

Encoder



Decoder

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

- In their 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

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 $\hat{\boldsymbol{y}}_t$, and then compute the **cross-entropy** loss between this distribution and the actual next word $\boldsymbol{y}_t[w_{t+1}]$

$\boldsymbol{y}_t$ is a one-hot vector of length $|V|$:

- $\boldsymbol{y}_t[w = w_{t+1}] = 1$
- $\boldsymbol{y}_t[w \neq w_{t+1}] = 0$

Cross-entropy loss:

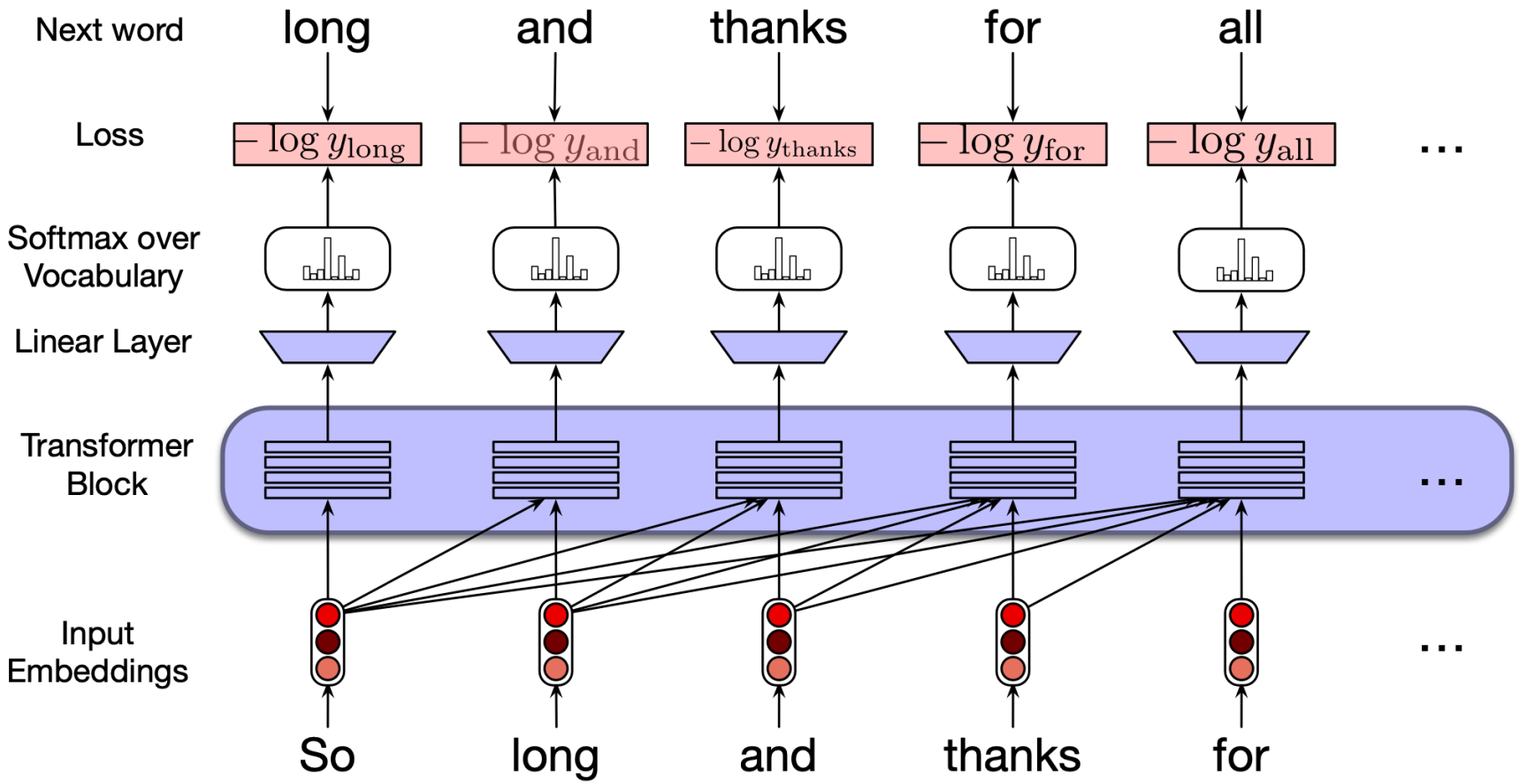
- $L_{CE} = -\sum_{w \in V} \boldsymbol{y}_t[w] \log \hat{\boldsymbol{y}}_t = -\log \hat{\boldsymbol{y}}_t[w_{t+1}]$
- Then average over entire sequence: $\frac{1}{T}\sum_{t=1}^{T} L_{CE}$

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

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
  - Progress on 2024.4.8 ⇒目前总数据量31935GB，目标是达到ChatGPT3.5的40T数据，目前进度79.83%

Scaling Laws

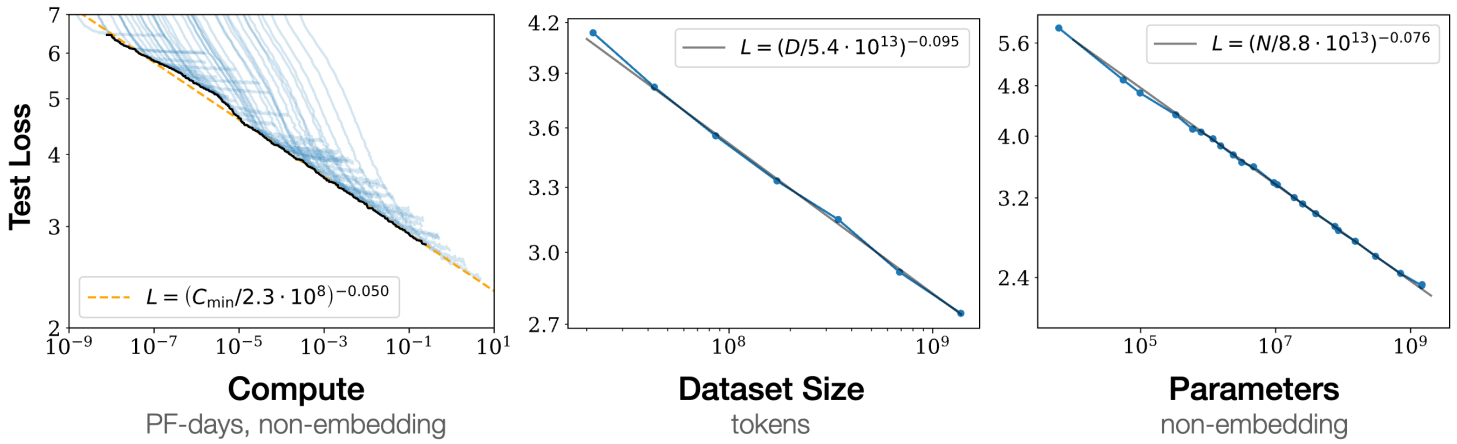
- Performance of LLMs are determined by three factors: model size, dataset size, and training time
- **Scaling laws**: the model's performance (measured by loss) scales as a *power low* with each of the three factors

$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C} \qquad L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} \qquad L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N}$$

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

Scaling Laws: Estimate # of Parameters

- The number of non-embedding parameters N can be estimated:
- $N \approx 2d \cdot n_{\text{layer}} \cdot (2d_{\text{attn}} + d_{\text{ff}}) \approx 12 n_{\text{layer}} d^2$
- (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 Model
- **Achievements and Drawbacks**

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)

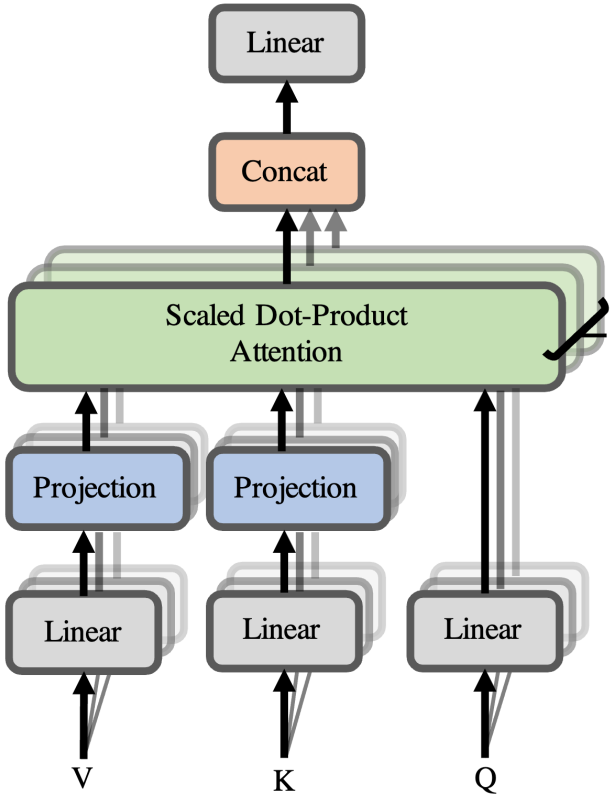
Drawbacks

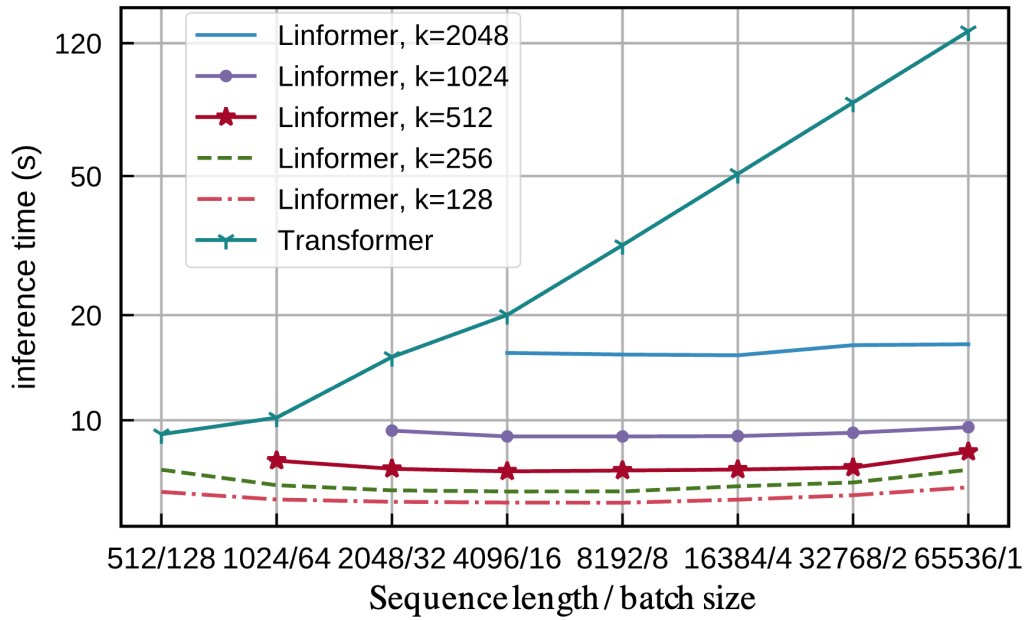
- **Quadratic computing time** in self-attention
- Computing all pairs of interactions means that computation grows quadratically with sequence length
- For RNN models, it only grow linearly

- Total number of operation grows in $O(n^2 d)$
- 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

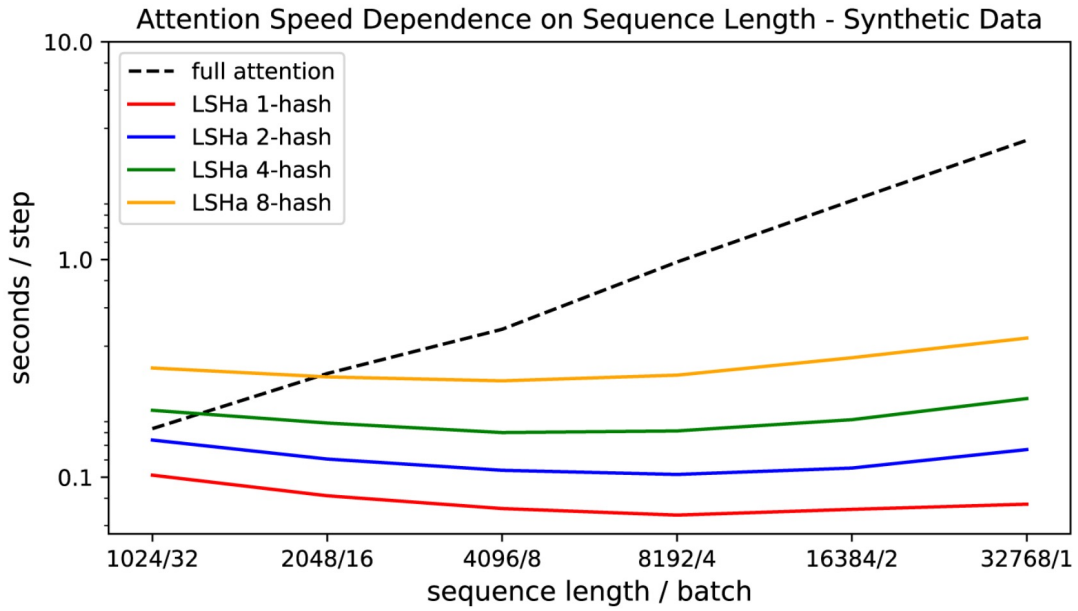
- Linformer (Wang et al., 2020): Approximate self-attention with low rank matrix, which reduces time and memory costs to $O(n)$

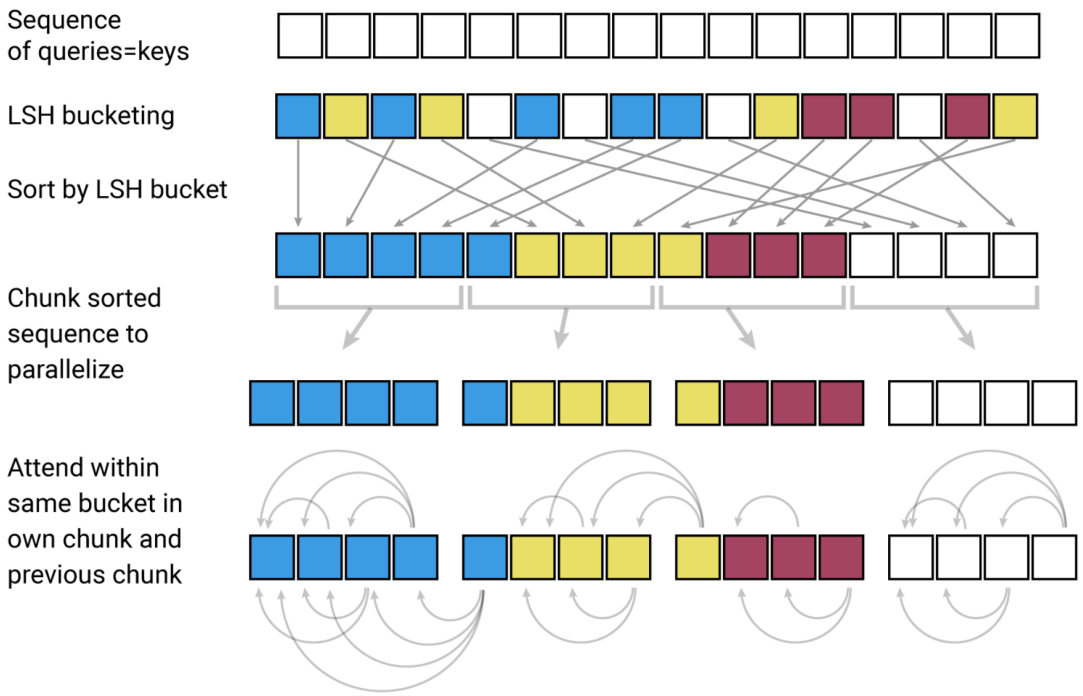




Improving Quadratic Self-Attention

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





To-Do List

- Read Chapter 11 of SLP3: Fine-Tuning and Masked Language Models
- Attend Lab 8
- Keep working on A4

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.