北京大学
PEKING UNIVERSITY

Introduction to Large Language Model

Midterm Review

Kun Yuan

北京大学
PEKING UNIVERSITY

PART 01

Optimizer

Convex set and convex functions

北京大学 PEKING UNIVERSITY

Definition 1 (Convex set)

A set $\mathcal{X} \subseteq \mathbb{R}^d$ is called convex, if for $\forall x, y \in \mathcal{X}$, it holds that

$$\theta x + (1-\theta)y \in \mathcal{X}, \quad \forall \theta \in [0,1].$$

Examples:

- Hyperplane $\{x | a^T x = b\}$ and hyperspace $\{x | a^T x \leq b\}$
- Euclidian ball $\{x | \|x - x_c\| \leq r\}$
- Polyhedron $\{x | a_j^T x \leq b_j, j = 1, \cdots, m, c_j^T x = d_j, j = 1, \cdots, p\}$

Convex set and convex functions

北京大学 PEKING UNIVERSITY

Definition 2 (Convex function)

Function $f : \mathcal{X} \to \mathbb{R}$ is said to be convex if $\mathcal{X} \subseteq \mathbb{R}^d$ is a convex set and $\forall x, y \in \mathcal{X}$, it holds that

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta) f(y), \quad \forall \theta \in [0,1].$$

Examples:

- Exponential e^{ax} is convex on $\mathbb{R}$ for any $a \in \mathbb{R}$
- Norms $\|x\|_1$ and $\|x\|_2$ are convex on $\mathbb{R}^d$
- Linear regressioin loss function $\|Ax - b\|^2$ is convex on $\mathbb{R}^d$
- Logistic regression $\frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-b_i a_i^T x))$ is convex on $\mathbb{R}^d$

L-smoothness

Definition 4 (L-smoothness)

A differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be L-smooth if $\forall x, y \in \mathbb{R}^n$,

$$\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2,$$

where $L > 0$ is the Lipschitz constant of ∇f.

In other words, the gradient cannnot vary too quckly

It is easy to show that the above inequality is equivalent to (see notes)

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2}\|y - x\|_2^2$$

which implie that $f(y)$ can be upper bounded by a quadratic function

Gradient descent

- Consider the following **smooth** and **unconstrained** optimization

$$\underset{x \in \mathbb{R}^d}{\text{minimize}} \quad f(x)$$

- **Gradient descent (GD)** is very effective to solve the above problem

$$x_{k+1} = x_k - \gamma \nabla f(x_k), \quad \forall k = 0, 1, \cdots$$

where γ is the learning rate (or step size), and x_0 initializes arbitrarily.

Convergence analysis

北京大学 PEKING UNIVERSITY

Theorem 1

Assume $f(x)$ to be L-smooth. If $\gamma \leq 1/L$, gradient descent converges as

$$\frac{1}{K+1}\sum_{k=0}^{K}\|\nabla f(x_k)\|^2 \leq \frac{2\big(f(x_0)-f^\star\big)}{\gamma(K+1)}.$$

If we further set $\gamma = 1/L$, it holds that

$$\frac{1}{K+1}\sum_{k=0}^{K}\|\nabla f(x_k)\|^2 \leq \frac{2L\big(f(x_0)-f^\star\big)}{K+1}.$$

Its proof is important

Stochastic optimization

北京大学 PEKING UNIVERSITY

- Consider the stochastic optimization problem:

$$\min_{x \in \mathbb{R}^d} \quad f(x) = \mathbb{E}_{\xi \sim \mathcal{D}}[F(x; \xi)]$$

  - ξ is a random variable indicating data samples
  - $\mathcal{D}$ is the data distribution; unknown in advance
  - $F(x; \xi)$ is differentiable in terms of x

Stochastic gradient descent

北京大学 PEKING UNIVERSITY

- Recall the problem

$$\min_{x\in\mathbb{R}^d} \quad f(x) = \mathbb{E}_{\xi\sim\mathcal{D}}[F(x;\xi)]$$

- Closed-form of $f(x)$ is unknown; gradient descent is not applicable

- Stochastic gradient descent (SGD):

$$x_{k+1} = x_k - \gamma \nabla F(x_k;\xi_k), \quad \forall k = 0, 1, \cdots$$

  where ξ_k is a data realization sampled at iteration k.

- Since $\{x_k\}$ are random, all iterates $\{x_k\}$ are also random

Convergence analysis

Let $\mathcal{F}_k = \{x_k, \xi_{k-1}, x_{k-1}, \cdots, \xi_0\}$ be the filtration containing all historical variables at and before iteration k (except for ξ_k).

Assumption 1

Given the filtration $\mathcal{F}_k$, we assume

$$\mathbb{E}[\nabla F(x_k; \xi_k) | \mathcal{F}_k] = \nabla f(x_k)$$

$$\mathbb{E}[\|\nabla F(x_k; \xi_k) - \nabla f(x_k)\|^2 | \mathcal{F}_k] \leq \sigma^2$$

Implying **unbiased** stochastic gradient and **bounded** variance.

Convergence analysis

北京大学 PEKING UNIVERSITY

Theorem 1

Suppose $f(x)$ is L-smooth and Assumption 1 holds. If $\gamma \leq 1/L$, SGD will converge at the following rate

$$\frac{1}{K+1}\sum_{k=0}^{K}\mathbb{E}[\|\nabla f(x_k)\|^2] \leq \frac{2\Delta_0}{\gamma(K+1)} + \gamma L\sigma^2,$$

where $\Delta_0 = f(x_0) - f^\star$.

- SGD **cannot** converge to stationary point with constant learning rate
- Smaller learning rate γ or variance σ^2 leads to smaller convergence error

Momentum SGD

- **Polyak's momentum** method, a.k.a, **heavy-ball** gradient method

$$x_k = x_{k-1} - \gamma \nabla f(x_{k-1}) + \beta(x_{k-1} - x_{k-2})$$

where $\beta \in (0, 1)$ is the momentum parameter

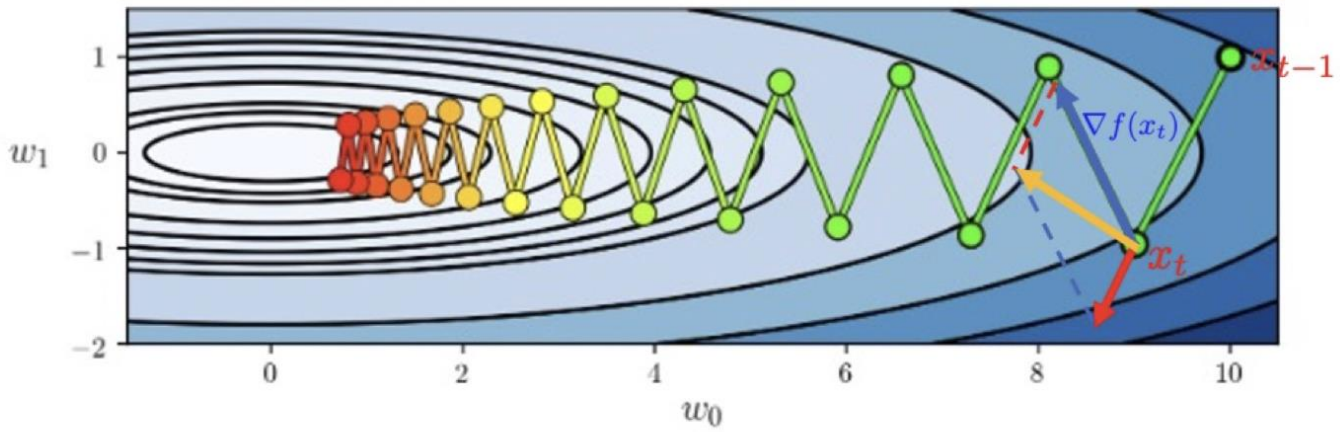


Figure: Momentum can alleviate the "Zig-Zag"

Momentum SGD

- Gradient descent with **Nesterov's momentum**, a.k.a, **Nesterov accelerated gradient (NAG)** method

$$y_{k-1} = x_{k-1} + \beta(x_{k-1} - x_{k-2})$$
$$x_k = y_{k-1} - \gamma \nabla f(y_{k-1})$$

where $\beta \in (0, 1)$ is the momentum parameter

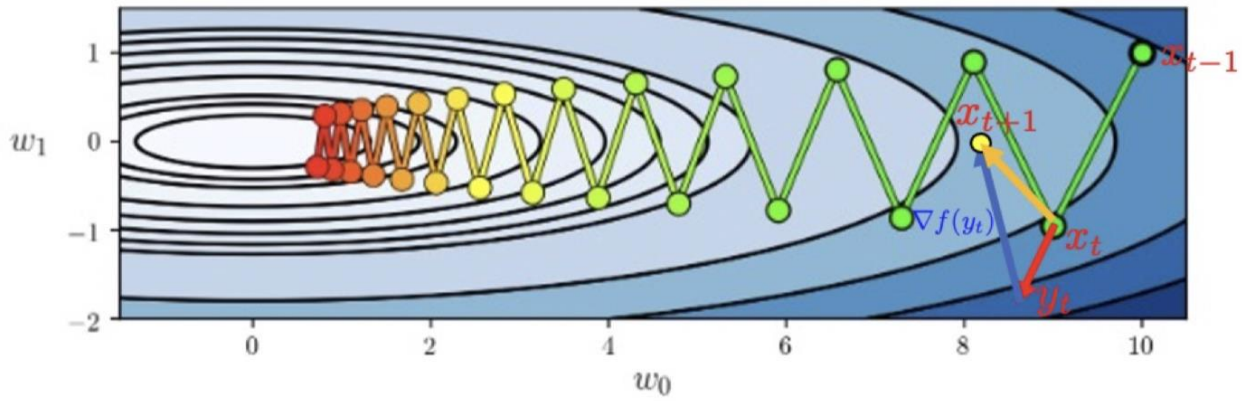


Figure: Nesterov method can alleviate the "Zig-Zag"

Adam

- Adam applies both momentum and adaptive rate to alleviate "Zig-Zag".

$$g_k = \nabla F(x_k; \xi_k)$$

$$m_k = \beta_1 m_{k-1} + (1 - \beta_1) g_k$$

$$s_k = \beta_2 s_{k-1} + (1 - \beta_2) g_k \odot g_k$$

$$x_{k+1} = x_k - \frac{\gamma}{\sqrt{s_k} + \epsilon} \odot m_k$$

where m_0 and s_0 are initialized as 0, and a small ϵ is added for safe-guard.

- Adam needs to store additional variables m and s, which will need more memories

Application: linear regression

- How to get the parameter $\boldsymbol{w}$? We can calculate it with $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$
- A good $\boldsymbol{w}$ will incur the minimum estimation error

$$\boldsymbol{w}^\star = \arg\min_{w \in \mathbb{R}^d} \left\{ \frac{1}{2N} \sum_{i=1}^{N} (\boldsymbol{x}_i^\top \boldsymbol{w} - y_i)^2 \right\} \tag{2}$$

where is called the linear regression problem

- If we introduce

$$X = [\boldsymbol{x}_1^\top; \cdots; \boldsymbol{x}_N^\top] \in \mathbb{R}^{N \times d} \quad y = [y_1; y_2; \cdots; y_N] \in \mathbb{R}^N$$

problem (2) becomes

$$\boldsymbol{w}^\star = \arg\min_{w \in \mathbb{R}^d} \left\{ \frac{1}{2} \|X\boldsymbol{w} - y\|^2 \right\}$$

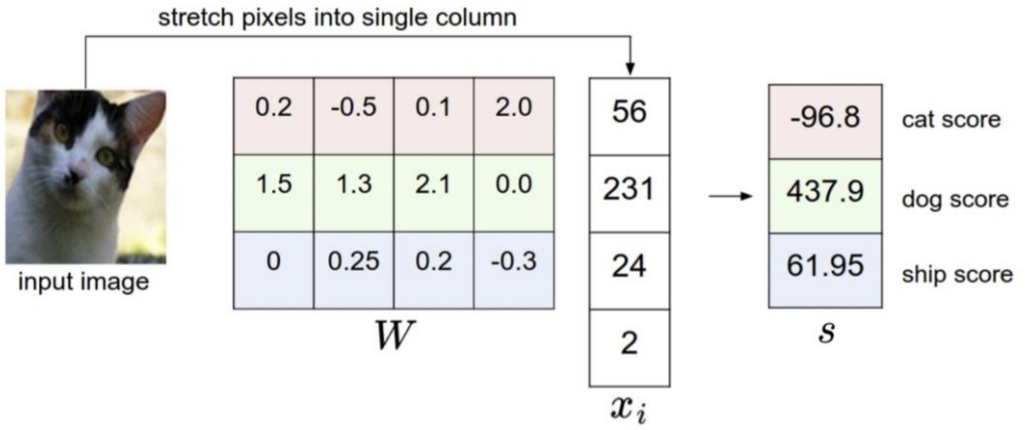
Application: multi-classification

- We first collect the dataset $\{\boldsymbol{x}_i, y_i\}_{i=1}^N$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ is the feature vector and $y_i \in \{1, \cdots, C\}$

- We consider the linear model to predict the score of each class

$$s = W\boldsymbol{x} \in \mathbb{R}^C$$

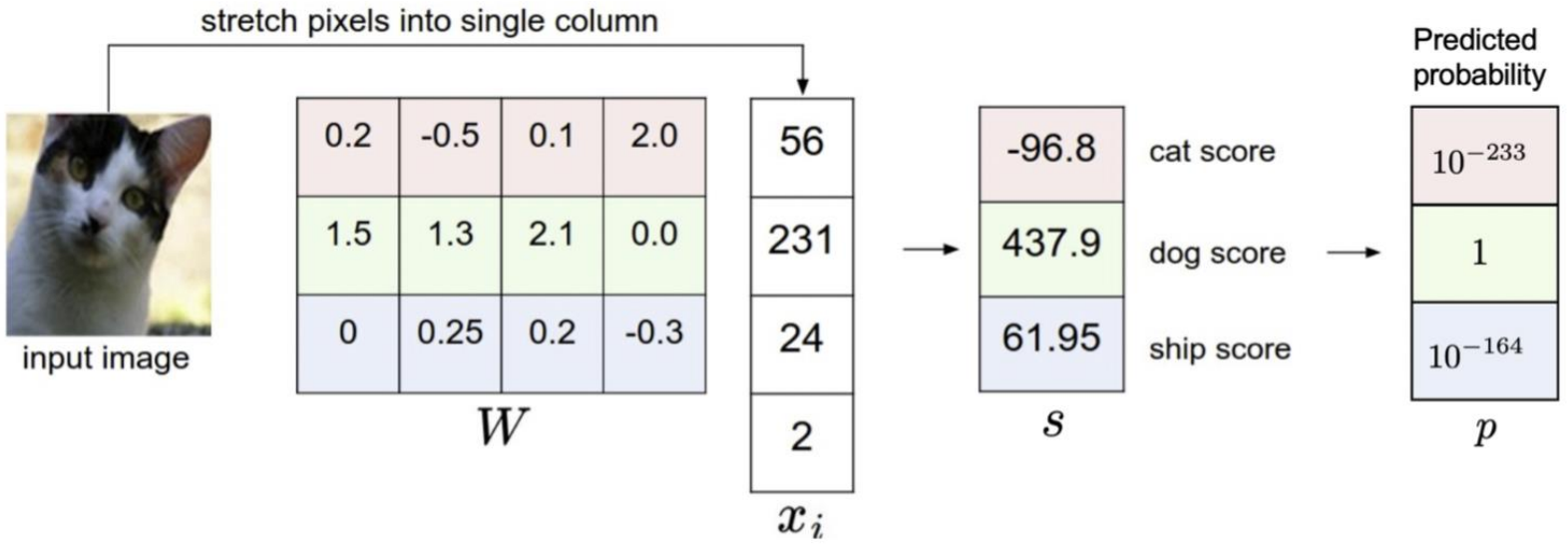
where $W \in \mathbb{R}^{C \times d}$ is the model parameter to learn and s_i indicates the score of the i-th class



Application: multi-classification

- Given the score vector s, we calculate the probability of each class with the **softmax** function as follows

$$p_i = \frac{\exp(s_i)}{\sum_{j=1}^{C} \exp(s_j)} \in (0, 1)$$



Application: multi-classification

- Cross entropy can measure the difference between two probability distributions $\boldsymbol{p} \in \mathbb{R}^d$ and $\boldsymbol{q} \in \mathbb{R}^d$

$$H(\boldsymbol{p}, \boldsymbol{q}) = -\sum_{i=1}^{d} p_i \log(q_i)$$

  Smaller cross entropy indicates smaller difference between $\boldsymbol{p}$ and $\boldsymbol{q}$.

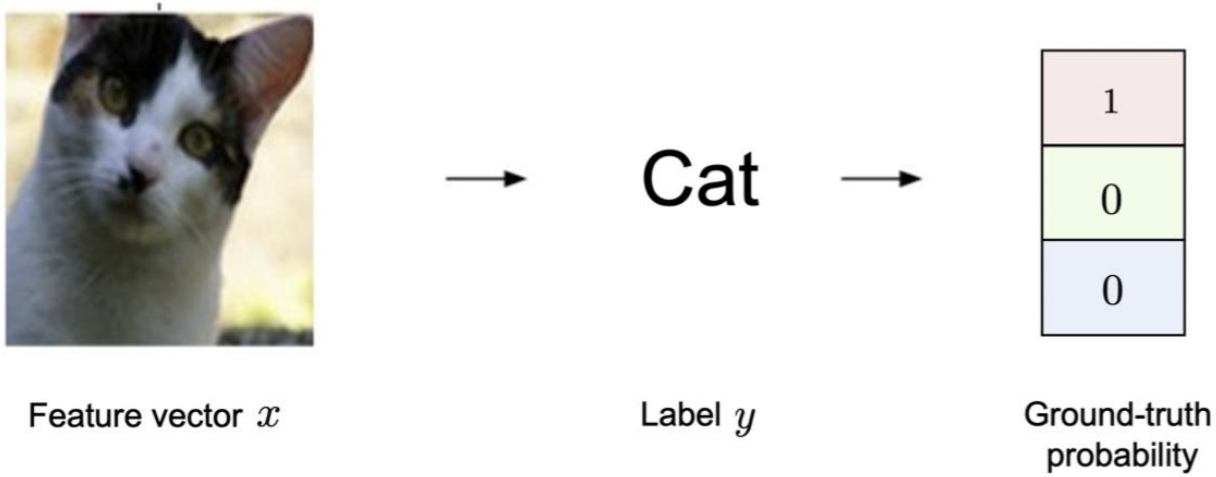
- Examples:

$$\boldsymbol{p} = (1, 0, 0, 0) \quad \boldsymbol{q} = (0.25, 0.25, 0.25, 0.25) \quad \longrightarrow \quad H(\boldsymbol{p}, \boldsymbol{q}) = 2$$

$$\boldsymbol{p} = (1, 0, 0, 0) \quad \boldsymbol{q} = (0.91, 0.03, 0.03, 0.03) \quad \longrightarrow \quad H(\boldsymbol{p}, \boldsymbol{q}) = 0.136$$

Application: multi-classification

- Recall the ground truth probability



- Now we achieve the loss function in multi-classification

$$\min_{W \in \mathbb{R}^{C \times d}} \quad -\frac{1}{N} \sum_{i=1}^{N} \log \left(\frac{\exp(s_{y_i})}{\sum_{j=1}^{C} \exp(s_j)} \right) \quad \text{where} \quad s = W \boldsymbol{x}_i$$

PART 02

Language Model

Word semantic representation

北京大学 PEKING UNIVERSITY

- Each word is represented with vectors that involve semantics

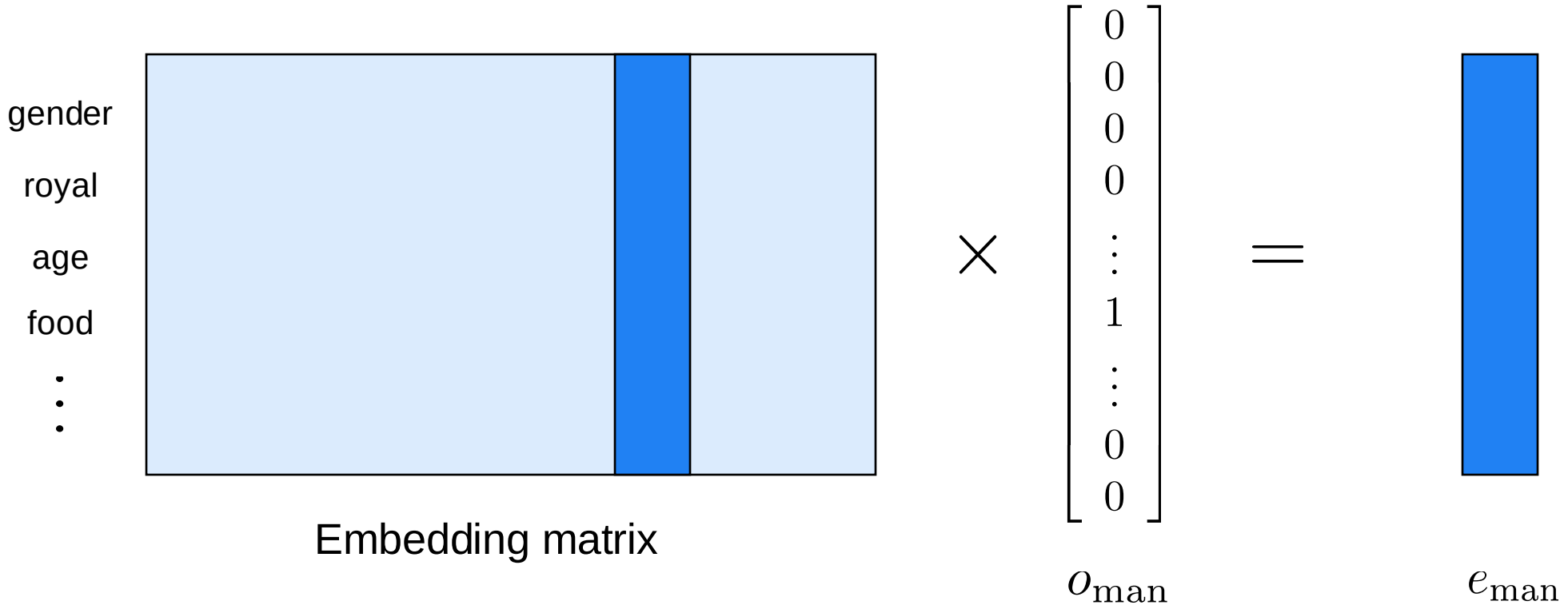
	Man (5391)	Woman (9853)	King (4914)	Queen (7159)	Apple (456)	Orange (6527)
Gender	-1.00	1.00	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.51	0.47	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97

- “Man” is close to “Woman”, “King” is close to “Queen”, and “Apple” is close to “Orange”

- Semantic representation can be much shorter than 1-hot representation

Word embedding

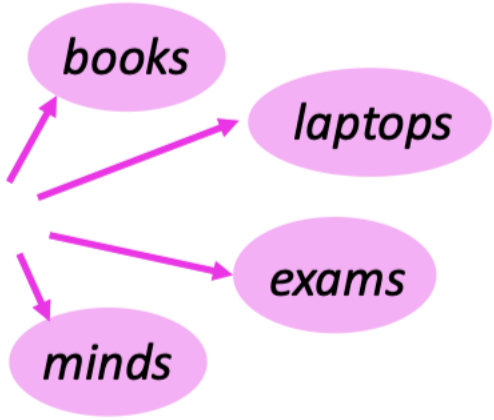
- Given the embedding matrix, we can easily achieve the semantic representation as follows



Language modeling

- **Language Modeling** is the task of predicting what word comes next

the students opened their _____



- More formally: given a sequence of words $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$P(x^{(t+1)} | \, x^{(t)}, \ldots, x^{(1)}) \quad \text{where} \quad x^{(t)} \in \{x_1, \cdots, x_{|V|}\}$$

- A system that does this is called a **Language Model**

n-gram language model

the students opened their _____

- **Question**: How to learn a Language Model?
- **Answer** (pre- Deep Learning): learn an *n*-gram Language Model!

- **Definition:** An *n*-gram is a chunk of *n* consecutive words.
  - unigrams: “the”, “students”, “opened”, ”their”
  - bigrams: “the students”, “students opened”, “opened their”
  - trigrams: “the students opened”, “students opened their”
  - four-grams: “the students opened their”

- **Idea:** Collect statistics about how frequent different n-grams are and use these to predict next word.

n-gram language model

- First we make a Markov assumption: $x^{(t+1)}$ depends only on the preceding *n*-1 words

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)}|\overbrace{\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}}) \quad \text{(assumption)}$$

prob of a n-gram

prob of a (n-1)-gram

$$= \frac{P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{P(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})} \quad \text{(definition of conditional prob)}$$

- **Question:** How do we get these *n*-gram and (*n*-1)-gram probabilities?
- **Answer:** By counting them in some large corpus of text!

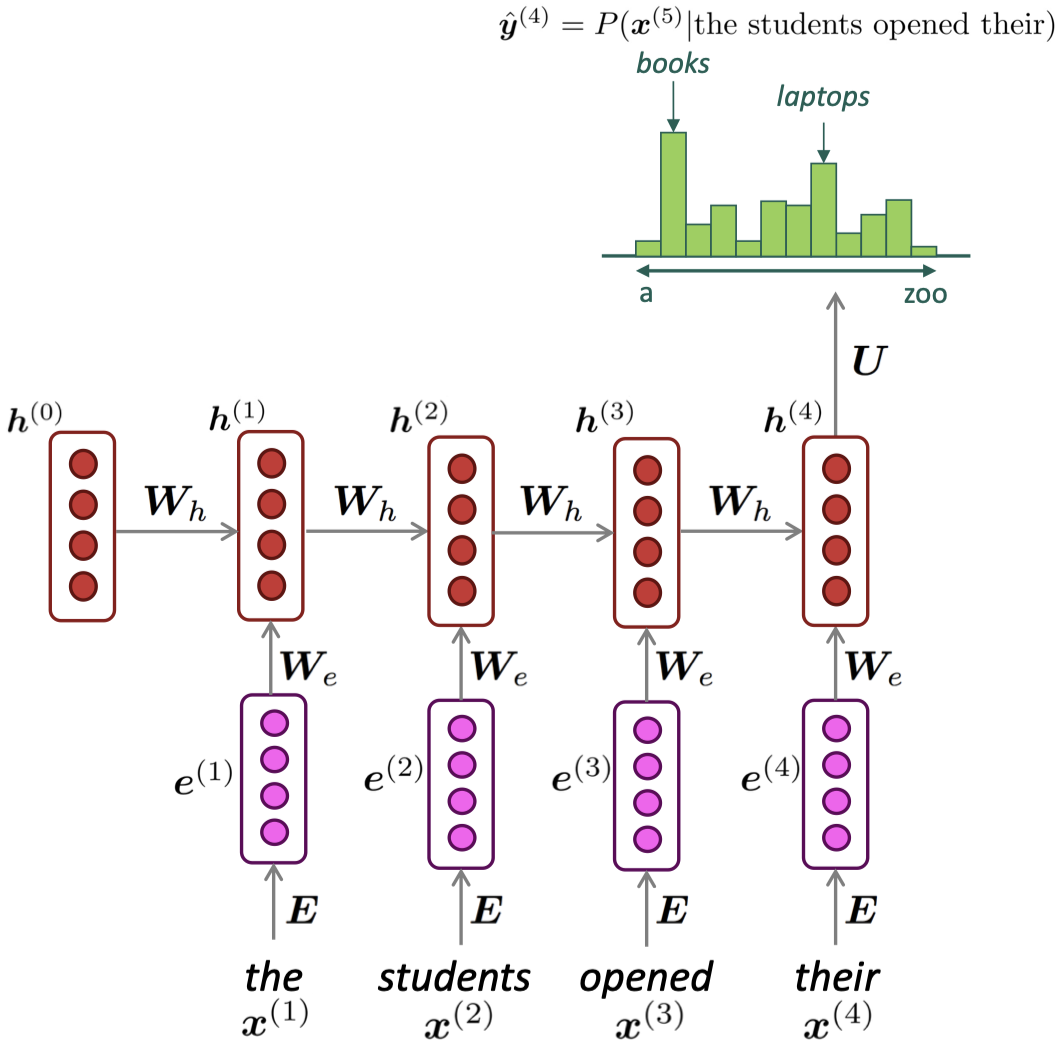
$$\approx \frac{\text{count}(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{\text{count}(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})} \quad \text{(statistical approximation)}$$

北京大学
PEKING UNIVERSITY

PART 03

RNN and Seq2Seq

Neural network model



output distribution

$$\hat{y}^{(t)} = \text{softmax}\left(Uh^{(t)} + b_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$h^{(t)} = \sigma\left(W_h h^{(t-1)} + W_e e^{(t)} + b_1\right)$$

$h^{(0)}$ is the initial hidden state

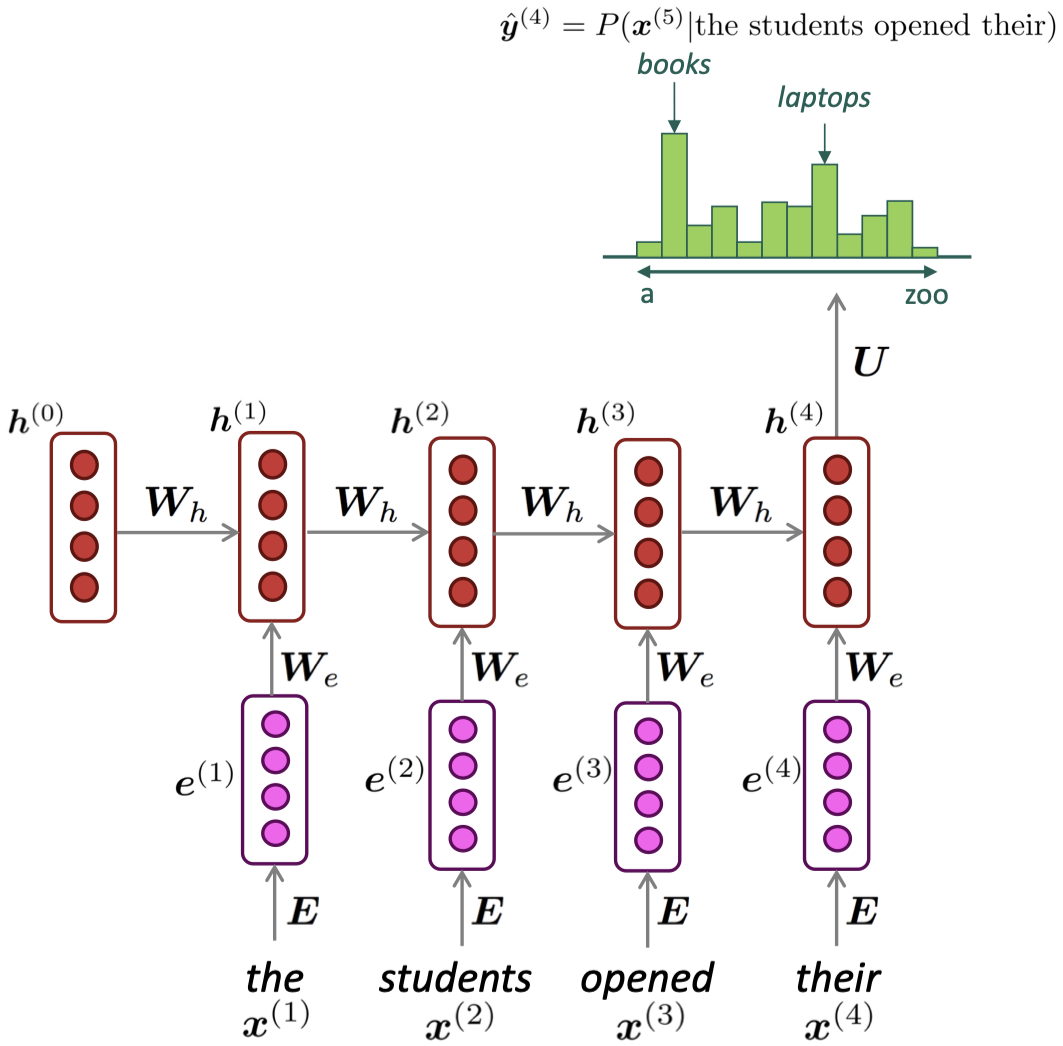
word embeddings

$$e^{(t)} = Ex^{(t)}$$

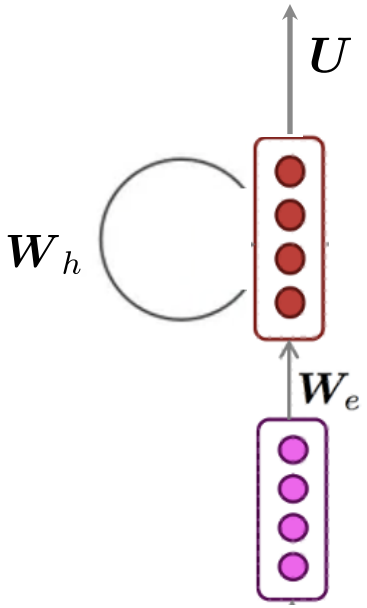
words / one-hot vectors

$$x^{(t)} \in \mathbb{R}^{|V|}$$

More concise representations



Model weights: $\{\boldsymbol{W}_h, \boldsymbol{W}_e, \boldsymbol{U}, b_1, b_2\}$

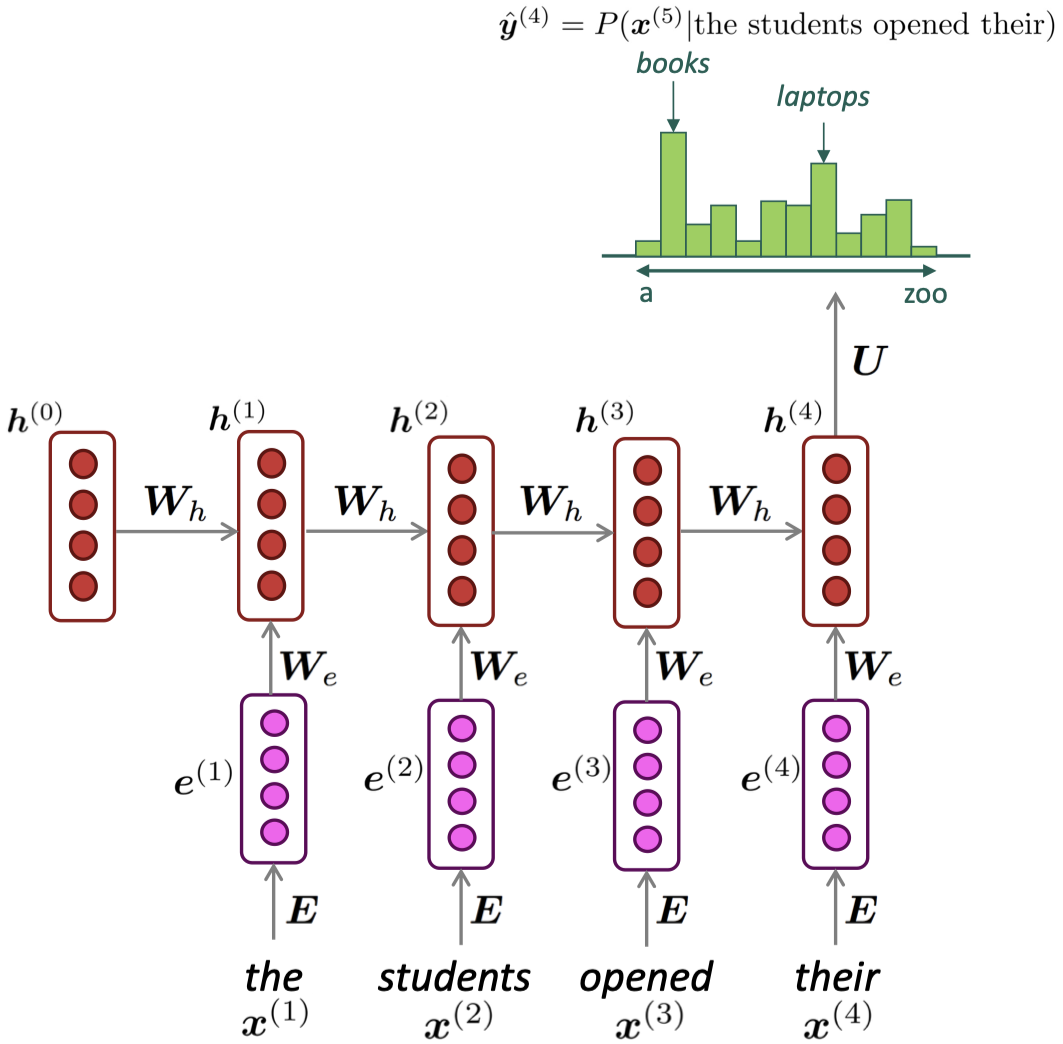


$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$\boldsymbol{h}^{(0)}$ is the initial hidden state

RNN benefits

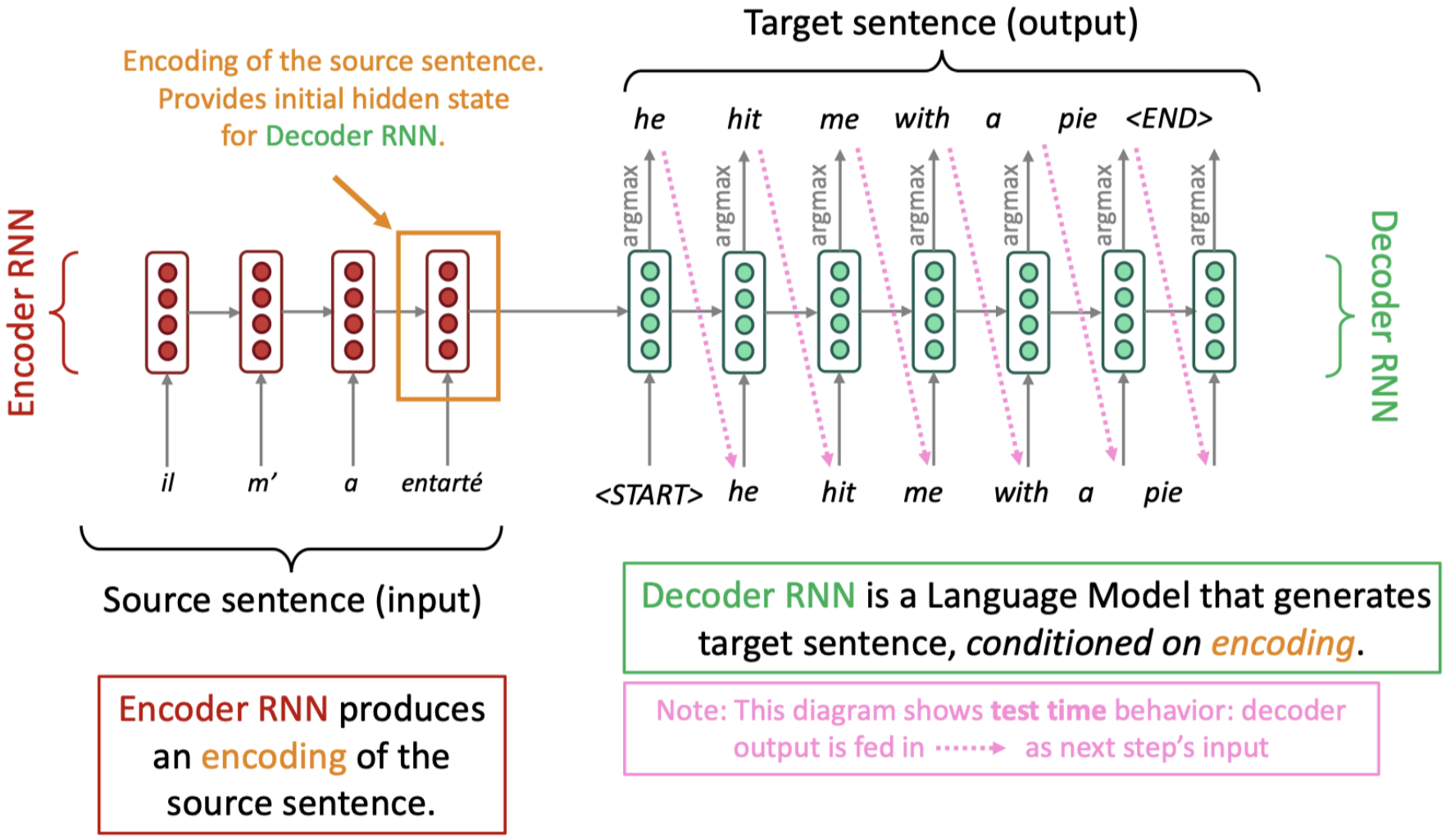


RNN **Advantages**:

- Can process any length input
- Computation for step *t* can (in theory) use information from many steps back
- Model size doesn't increase for longer input context

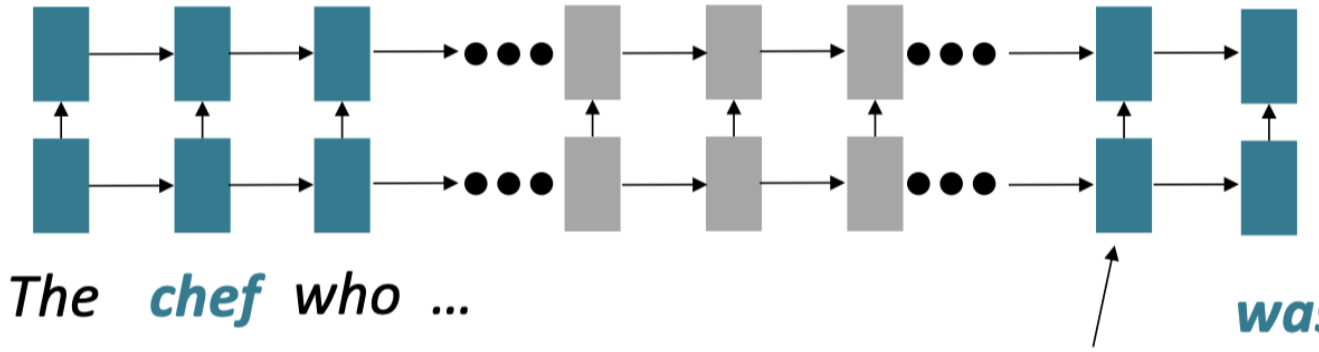
Sequence-to-sequence

- Sequence-to-sequence model



RNN issue I: Linear interaction distance

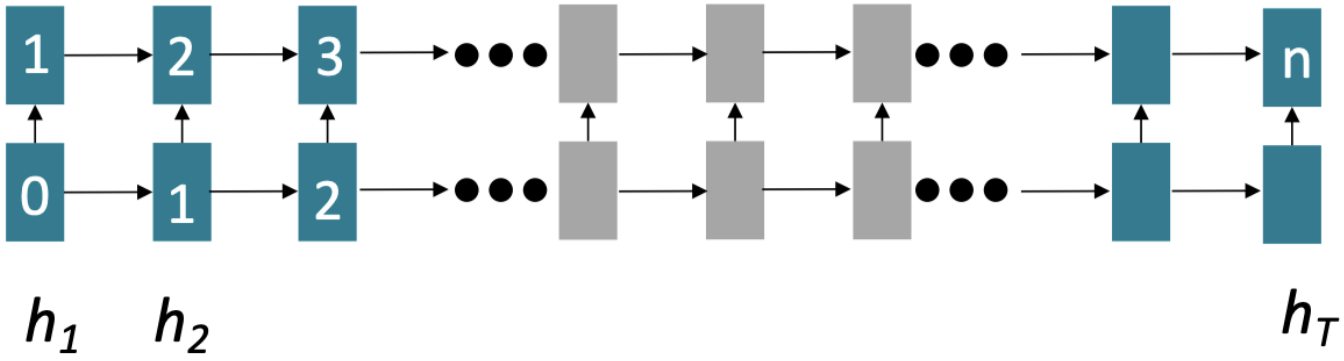
- **O(sequence length)** steps for distant word pairs to interact means:
  - Hard to learn long-distance dependencies (because gradient problems!)
  - Linear order of words is "baked in"; we already know linear order isn't the right way to think about sentences…



Info of ***chef*** has gone through O(sequence length) many layers!

RNN issue II: Lack of parallelizability

- Forward and backward passes have **O(sequence length)** unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!

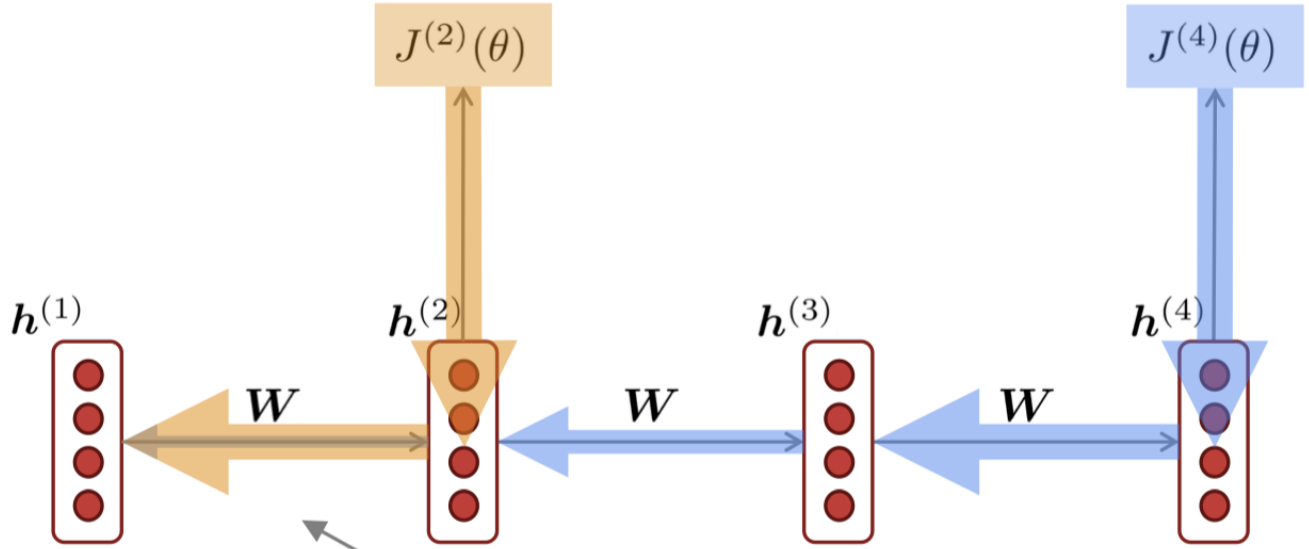


Numbers indicate min # of steps before a state can be computed

RNN issue III: Gradient vanishing or exploding

北京大学 PEKING UNIVERSITY

Vanishing gradient



Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are updated only with respect to near effects, not long-term effects.

北京大学
PEKING UNIVERSITY

PART 04

Transformer

Attention

北京大学 PEKING UNIVERSITY

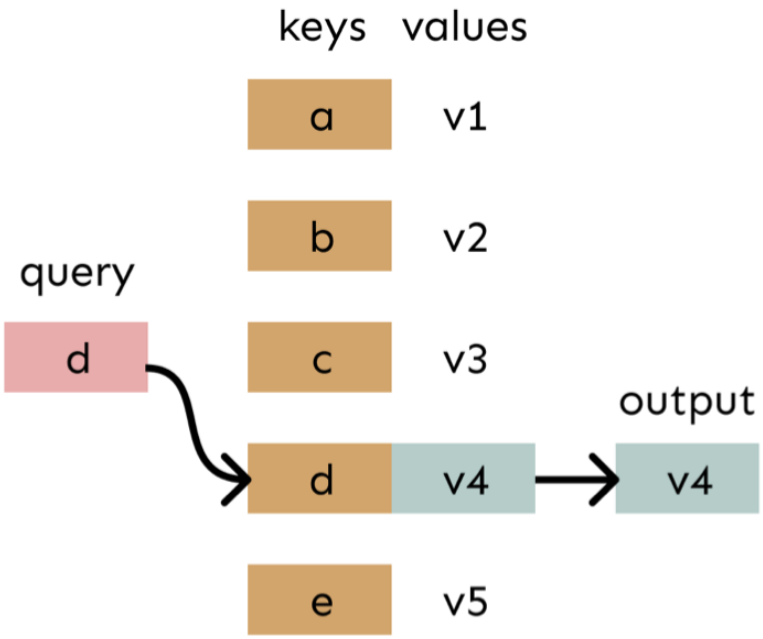
- **Attention** provides a solution to the bottleneck problem.

- **Core idea**: on each step of the decoder, use *direct connection to the encoder* to *focus on a particular part* of the source sequence

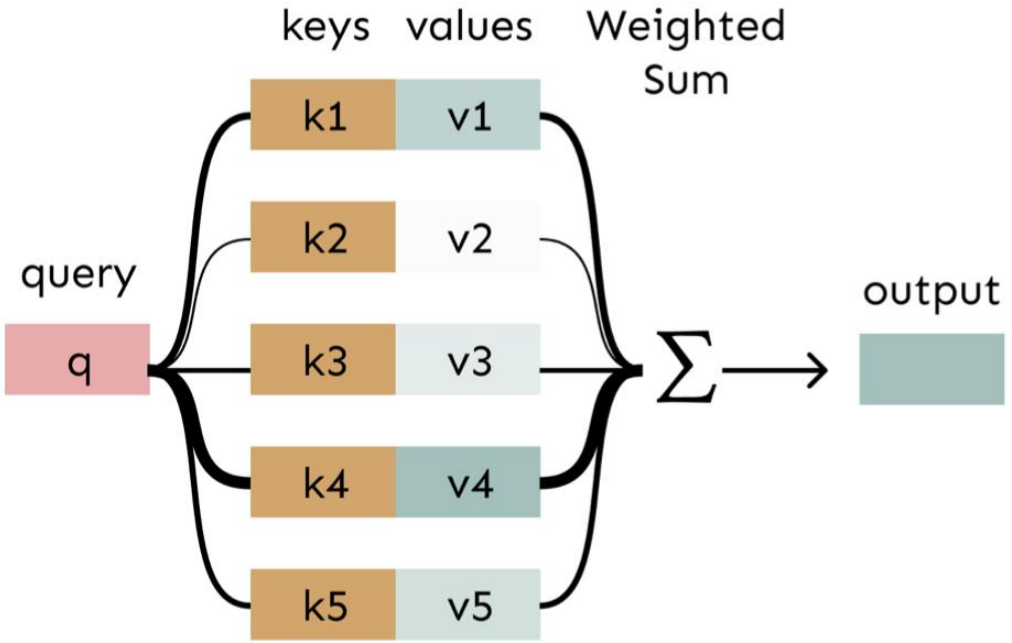
Attention is weighted averaging

Attention is just a **weighted** average – this is very powerful if the weights are learned!

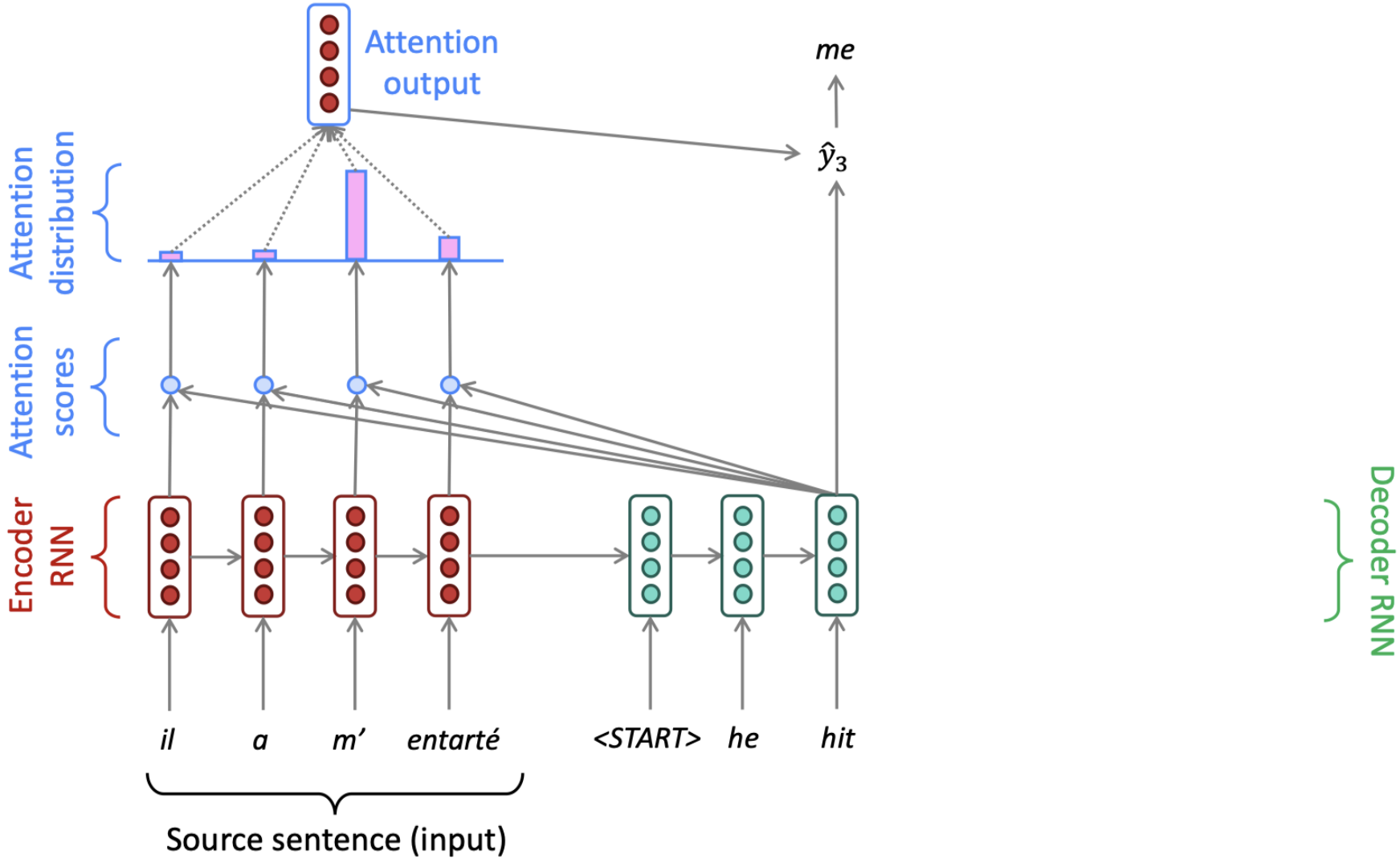
In a **lookup table**, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.



In **attention**, the **query** matches all **keys** *softly*, to a weight between 0 and 1. The keys' **values** are multiplied by the weights and summed.



Sequence-to-sequence with attention



Attention equations

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores e^t for this step:

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \text{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

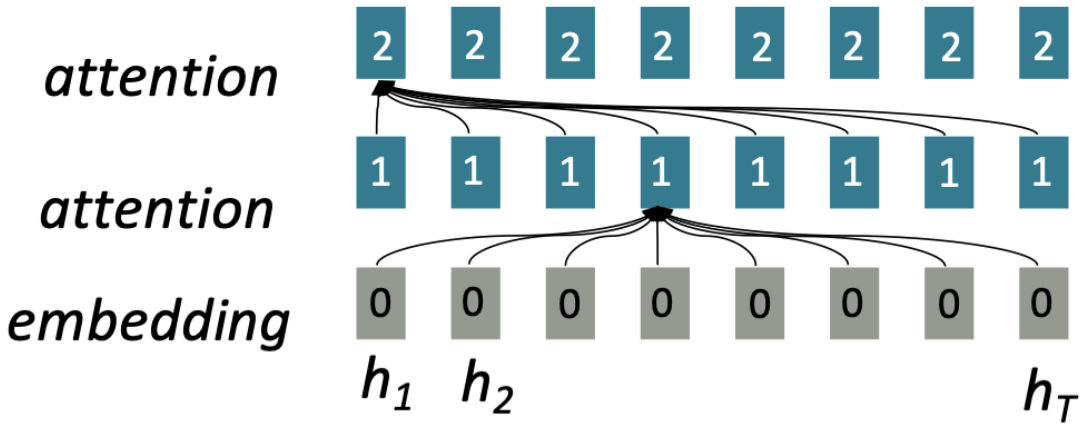
- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

- Concatenate the attention and the decoder hidden state to proceed $[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$

Attention is parallelizable, and solves bottleneck issues.

- **Attention** treats each word's representation as a **query** to access and incorporate information from **a set of values.**
  - We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.
- Number of unparallelizable operations does not increase with sequence length.
- Maximum interaction distance: O(1), since all words interact at every layer!



All words attend to all words in previous layer; most arrows here are omitted

Attention is great

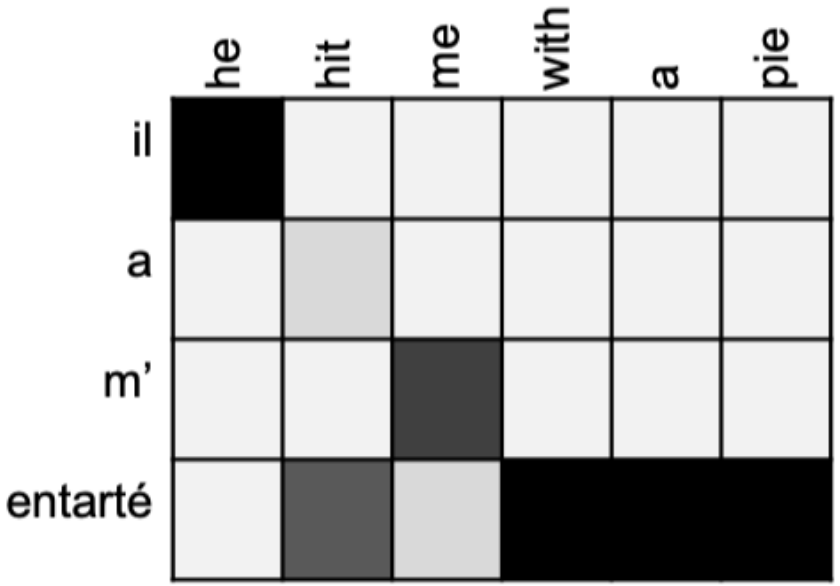
北京大学 PEKING UNIVERSITY

- Attention significantly improves NMT performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention provides a more "human-like" model of the MT process
  - You can look back at the source sentence while translating, rather than needing to remember it all
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with the vanishing gradient problem
  - Provides shortcut to faraway states

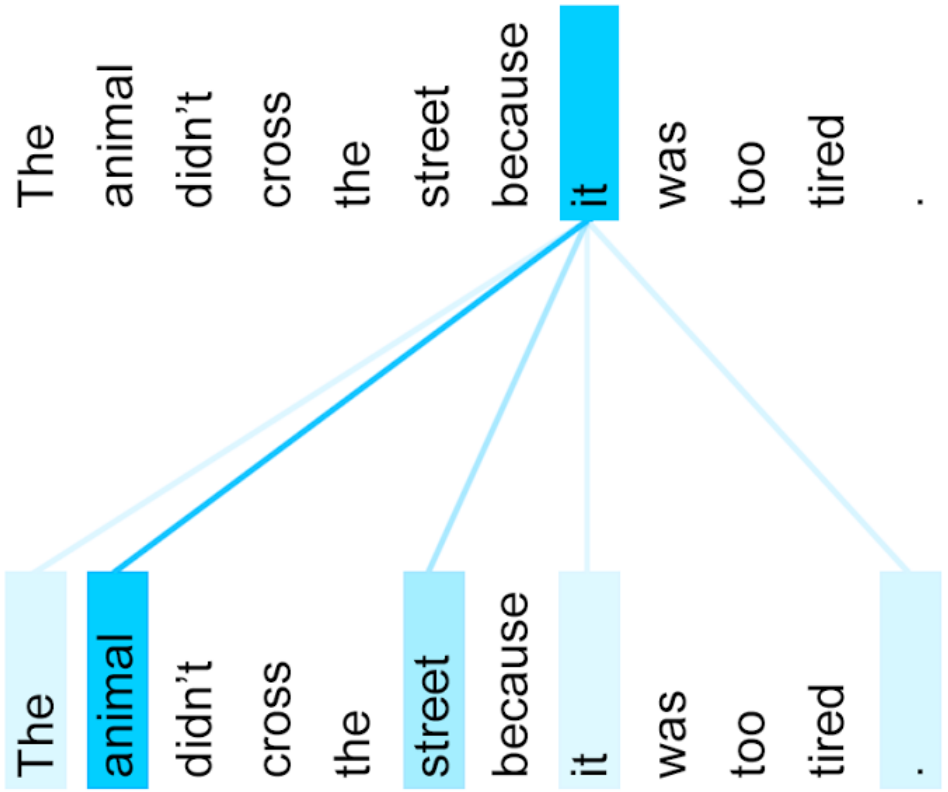
Attention is great

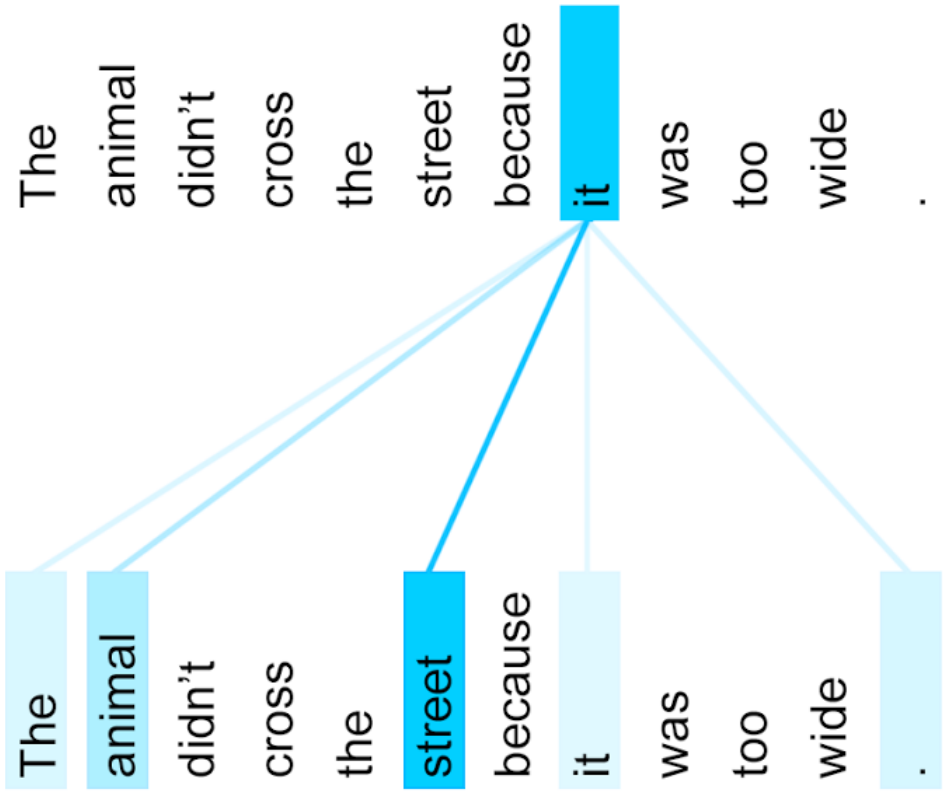
北京大学 PEKING UNIVERSITY

- Attention provides some interpretability
  - By inspecting attention distribution, we see what the decoder was focusing on
  - We get (soft) alignment for free!



Self attention





Self-Attention: keys, queries, values from the same sequence

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary V, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_i$, where $E \in \mathbb{R}^{d \times |V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices Q, K, V, each in $\mathbb{R}^{d \times d}$

$\boldsymbol{q}_i = W_Q \boldsymbol{x}_i$ (queries) $\quad \boldsymbol{k}_i = W_K \boldsymbol{x}_i$ (keys) $\quad \boldsymbol{v}_i = W_V \boldsymbol{x}_i$ (values)

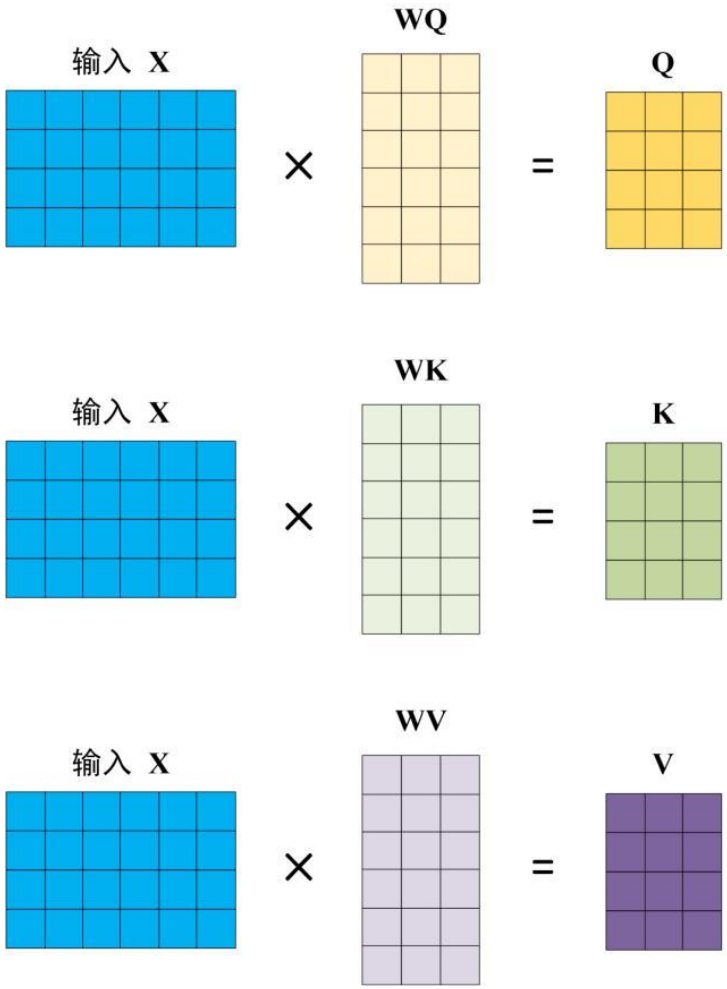
2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \boldsymbol{q}_i^\top \boldsymbol{k}_j \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'} \exp(\boldsymbol{e}_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \boldsymbol{v}_i$$

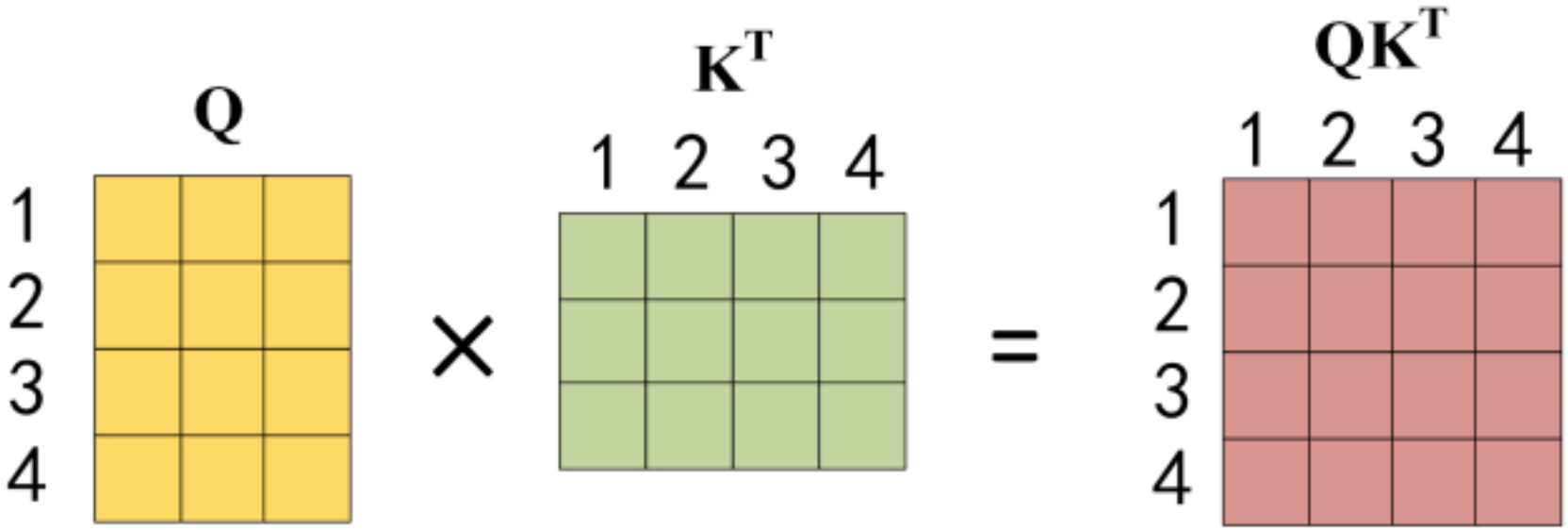
Self-Attention: matrix representation



Self-Attention: matrix representation

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

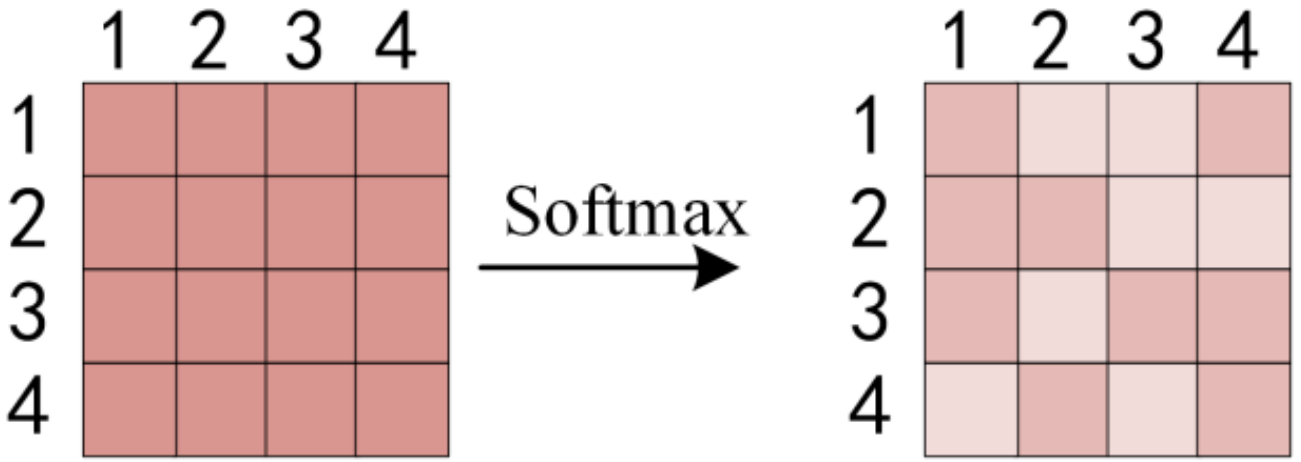
d_k是Q,K矩阵的列数，即向量维度



Self-Attention: matrix representation

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

d_k是Q, K矩阵的列数，即向量维度

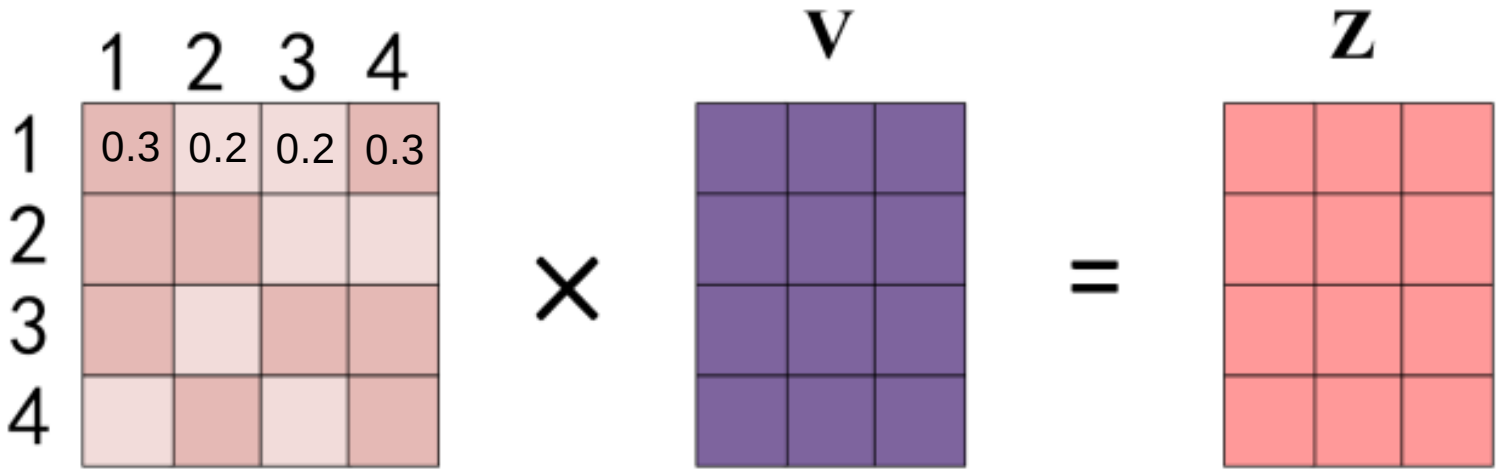


Self-Attention: matrix representation

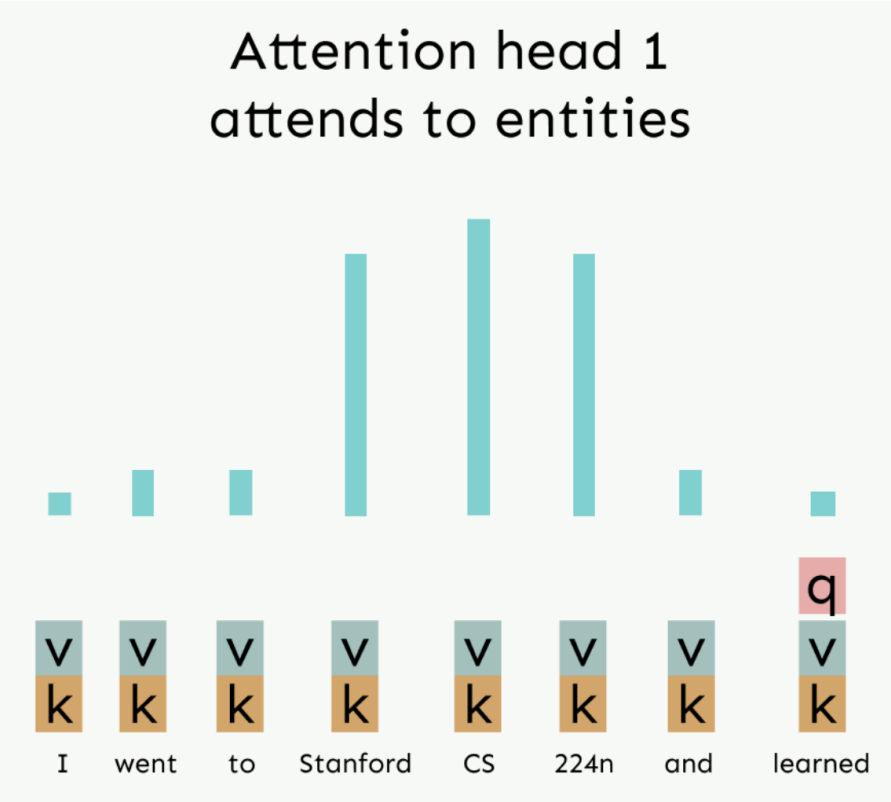
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

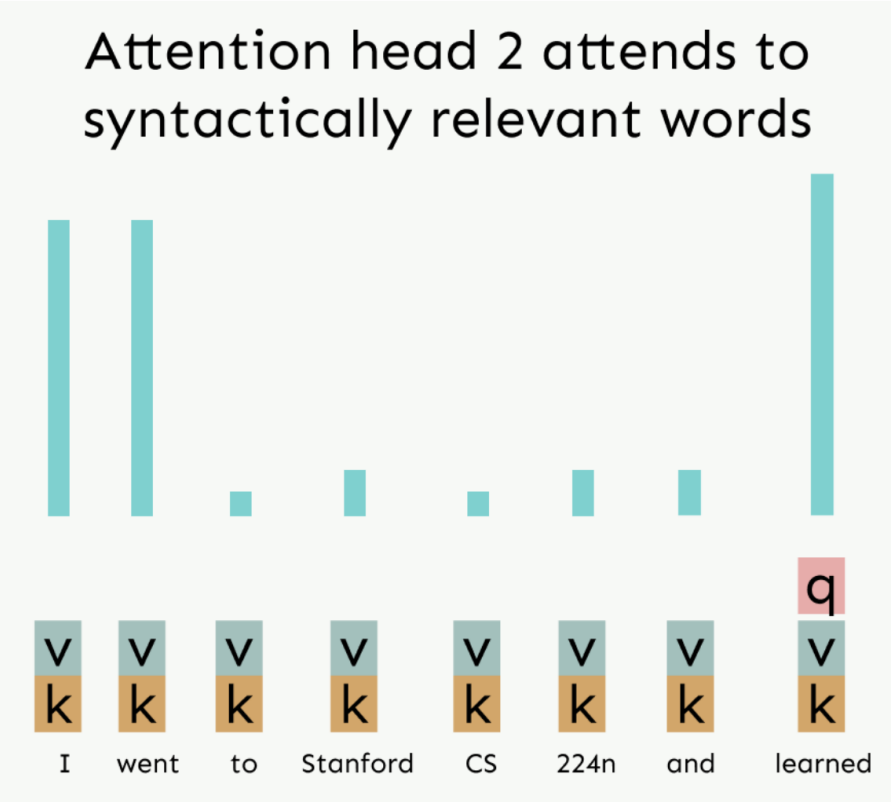
d_k是Q, K矩阵的列数，即向量维度

Very efficient
To calculate



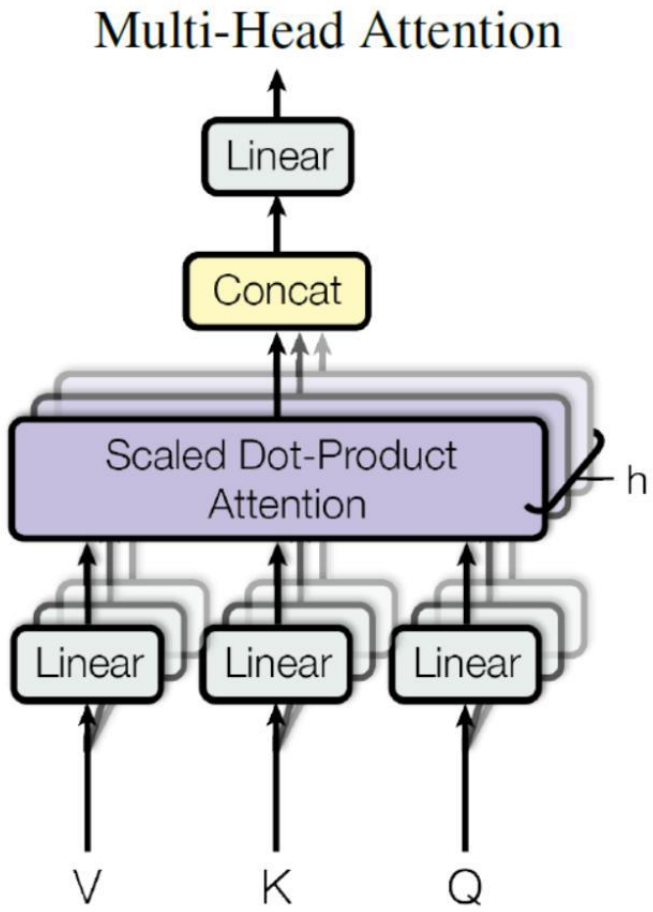
Multi-Head Attention

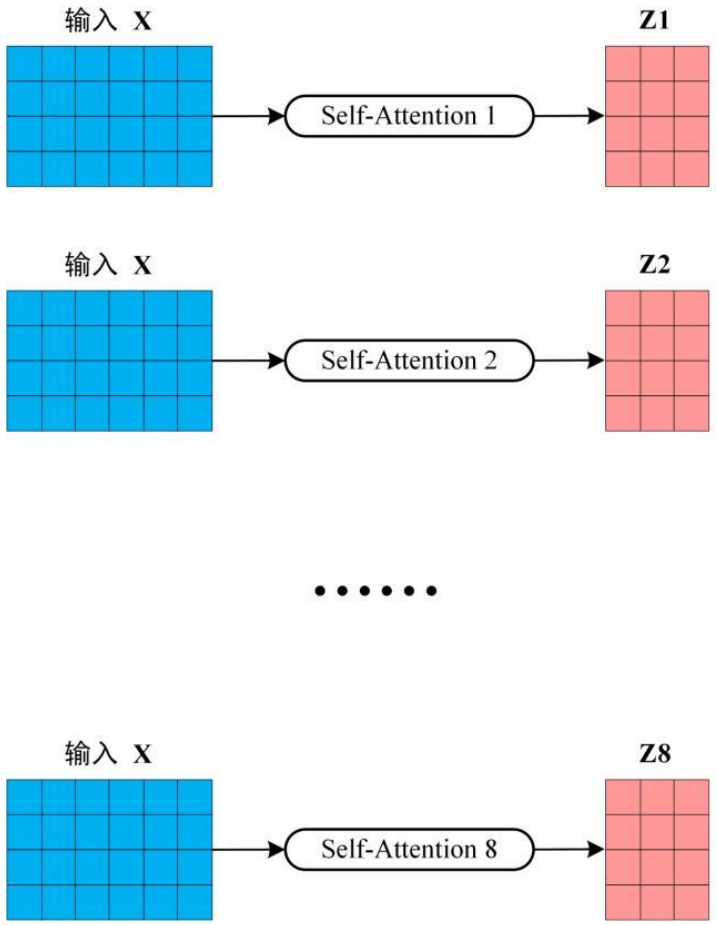




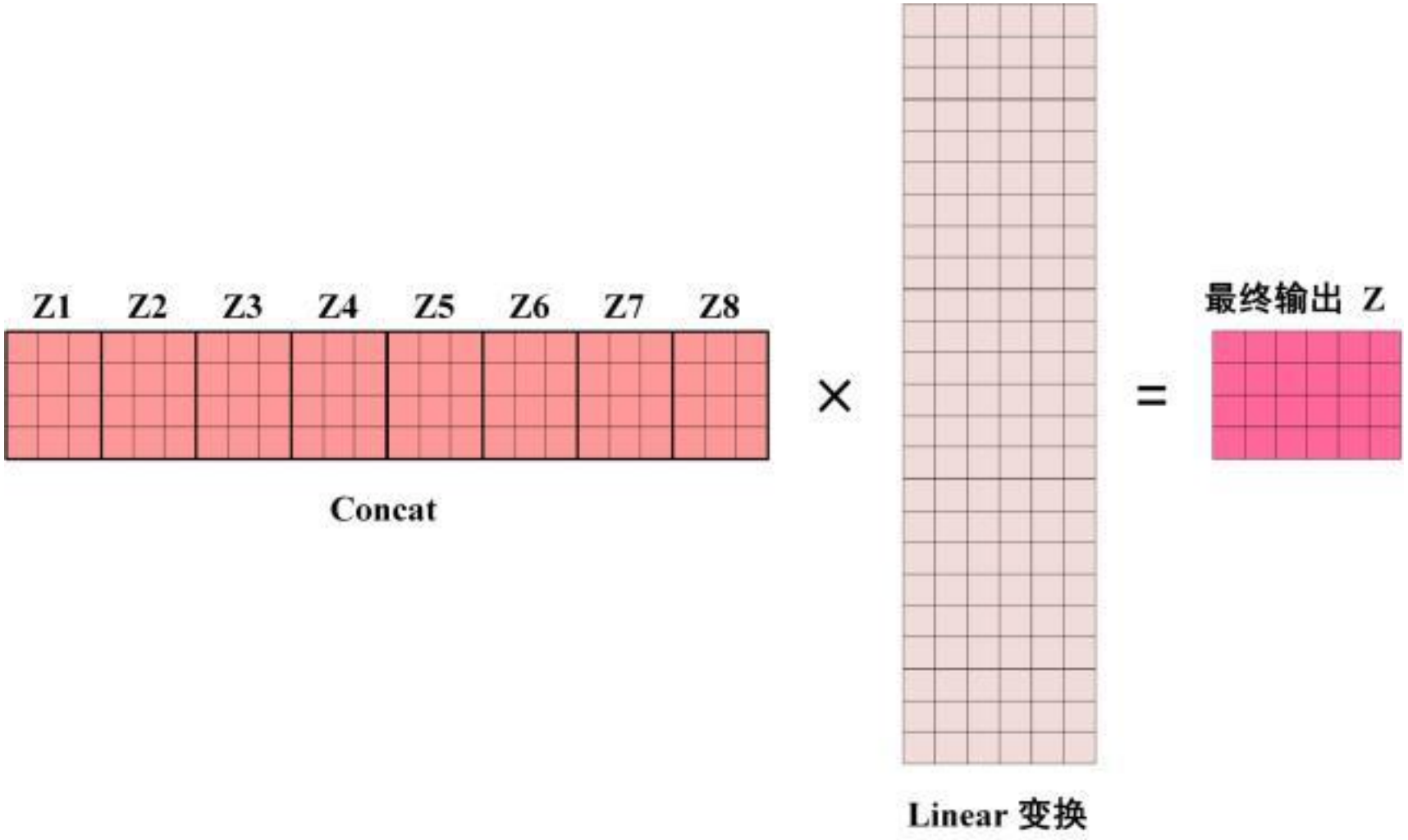
I went to Stanford CS 224n and learned

Use multi-head attention to achieve better representation





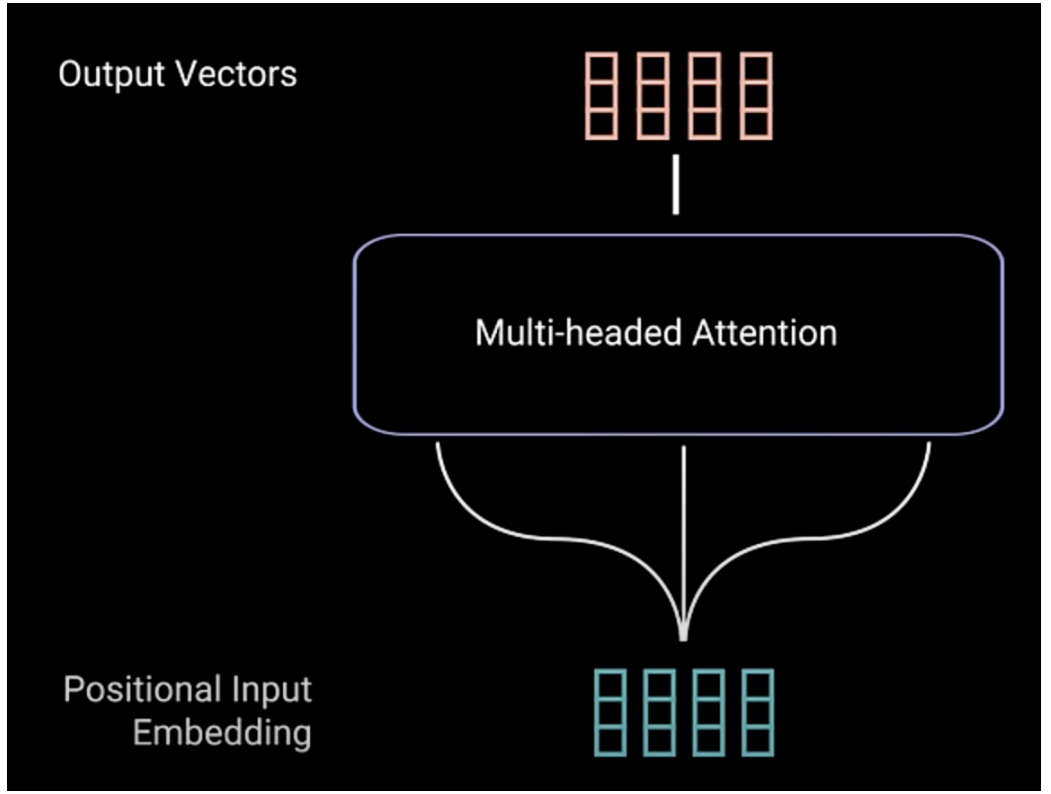
Multi-head attention: concatenation and linear transformation



Multi-head attention recap

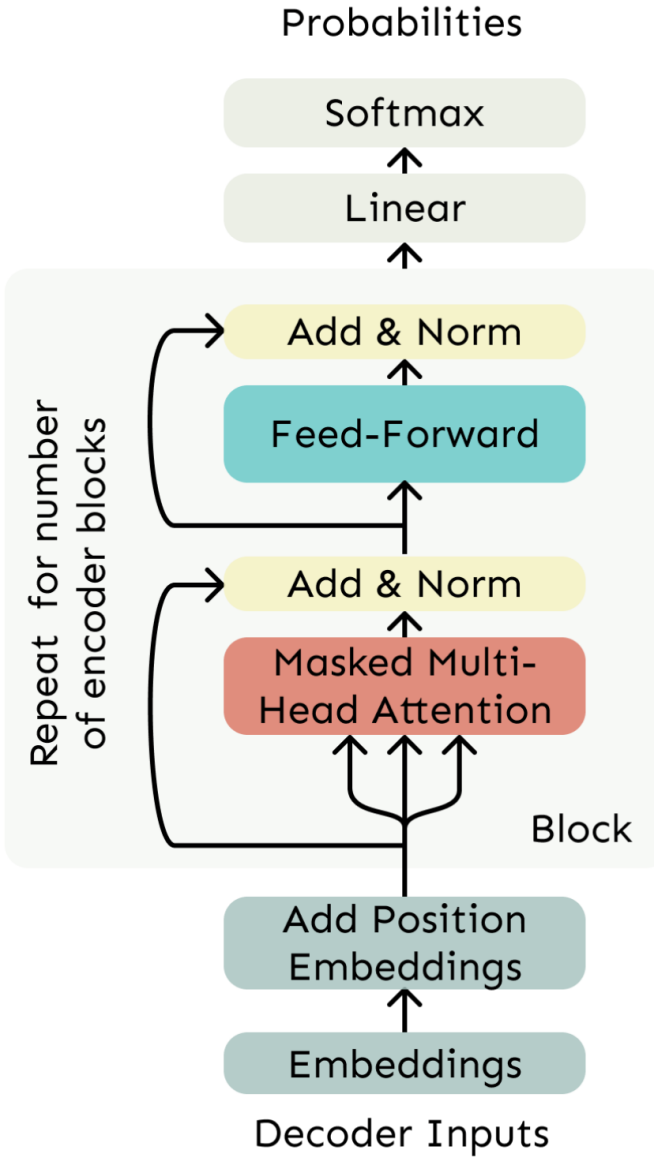
Multi-head attention is to find representations for the input words

It provides better representations than single self-attention



The transformer decoder

- The Transformer Decoder is a stack of Transformer Decoder **Blocks**.
- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- That's it! We've gone through the Transformer Decoder.



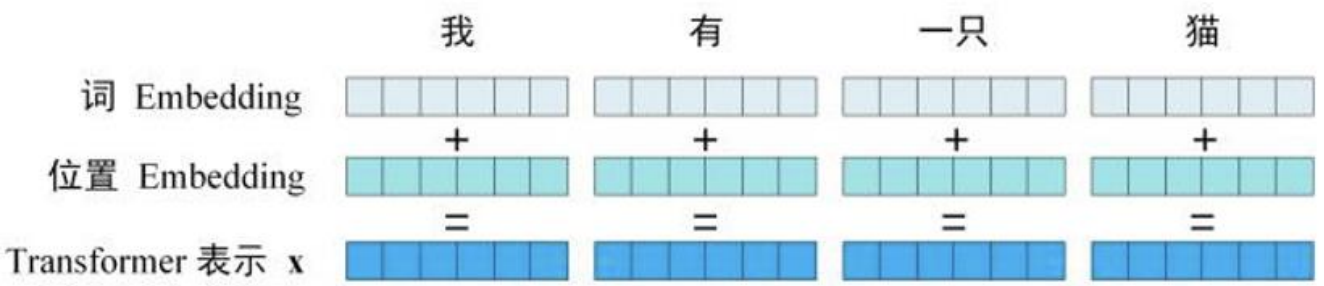
Position embedding

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$$\boldsymbol{p}_i \in \mathbb{R}^d, \text{ for } i \in \{1,2,\ldots,n\} \text{ are position vectors}$$

- Easy to incorporate this info into our self-attention block: just add the $\boldsymbol{p}_i$ to our inputs!
- Recall that $\boldsymbol{x}_i$ is the embedding of the word at index i. The positioned embedding is:

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

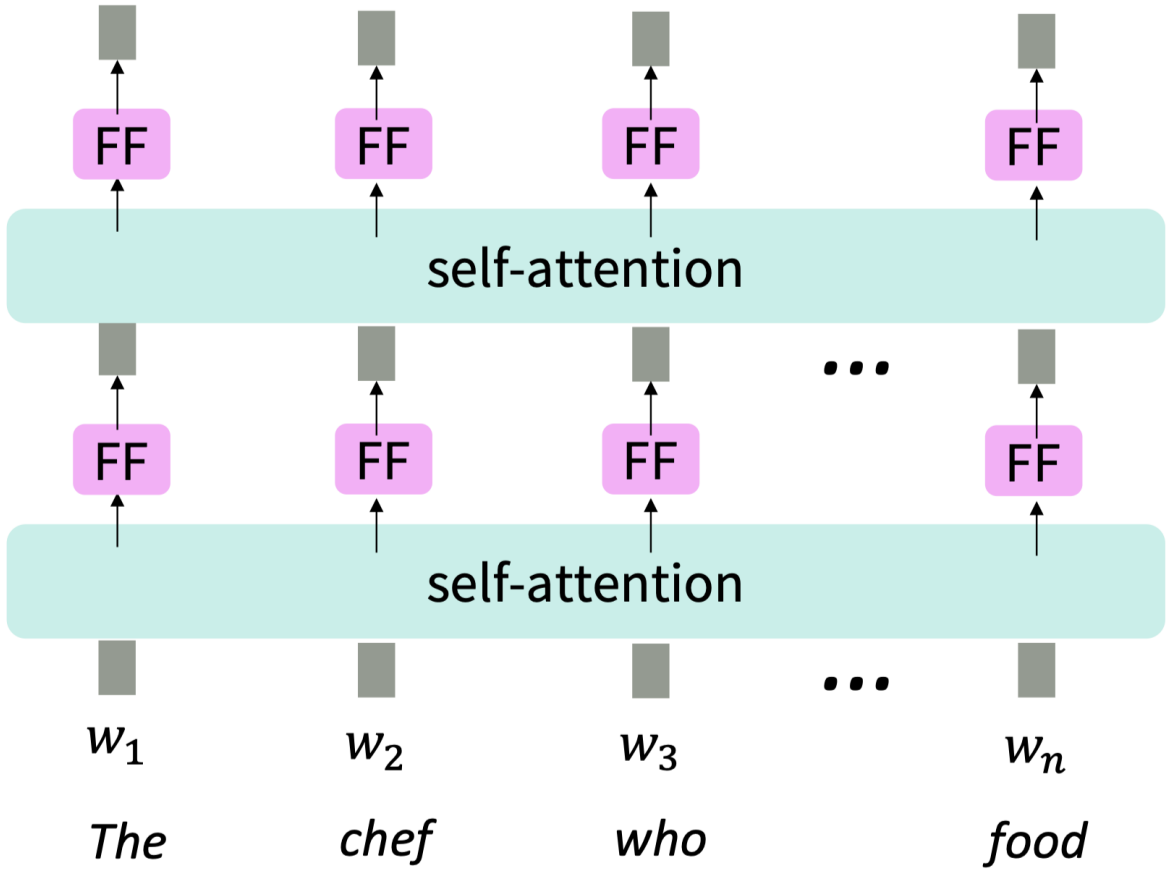


Adding nonlinearities in self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages **value** vectors (Why? Look at the notes!)

- Easy fix: add a **feed-forward network** to post-process each output vector.

$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1\ \text{output}_i + b_1) + b_2$$

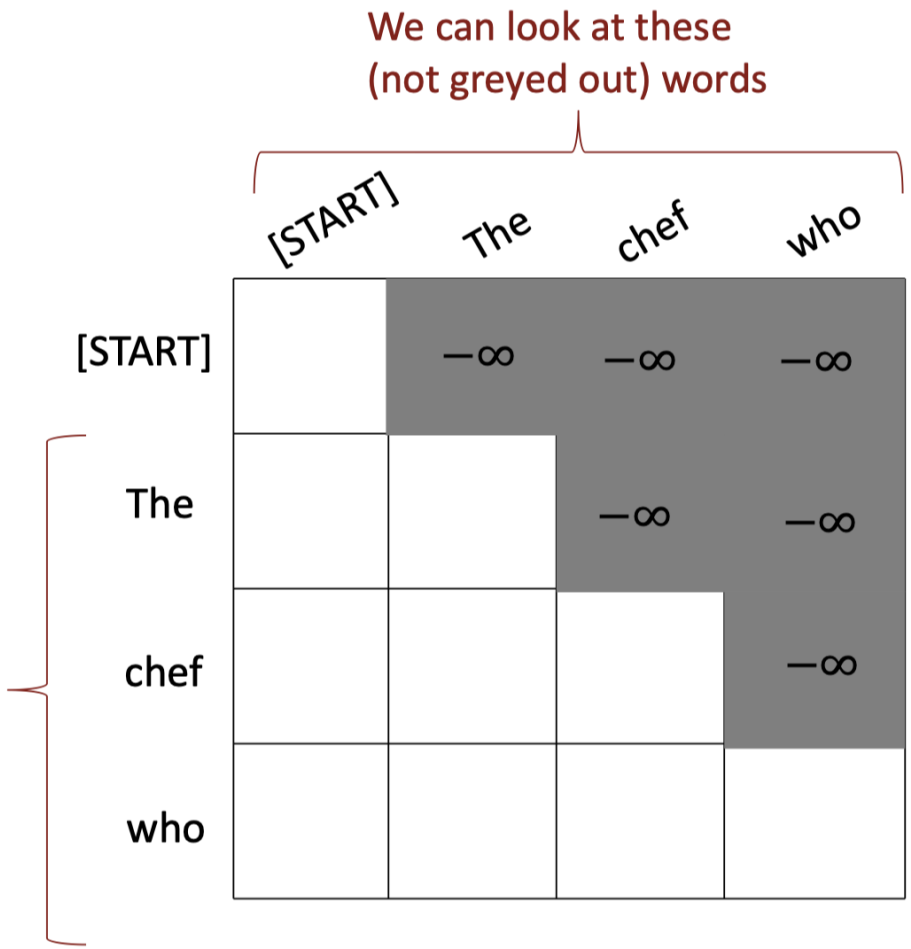


Intuition: the FF network processes the result of attention

Barriers and solutions for Self-Attention as a building block

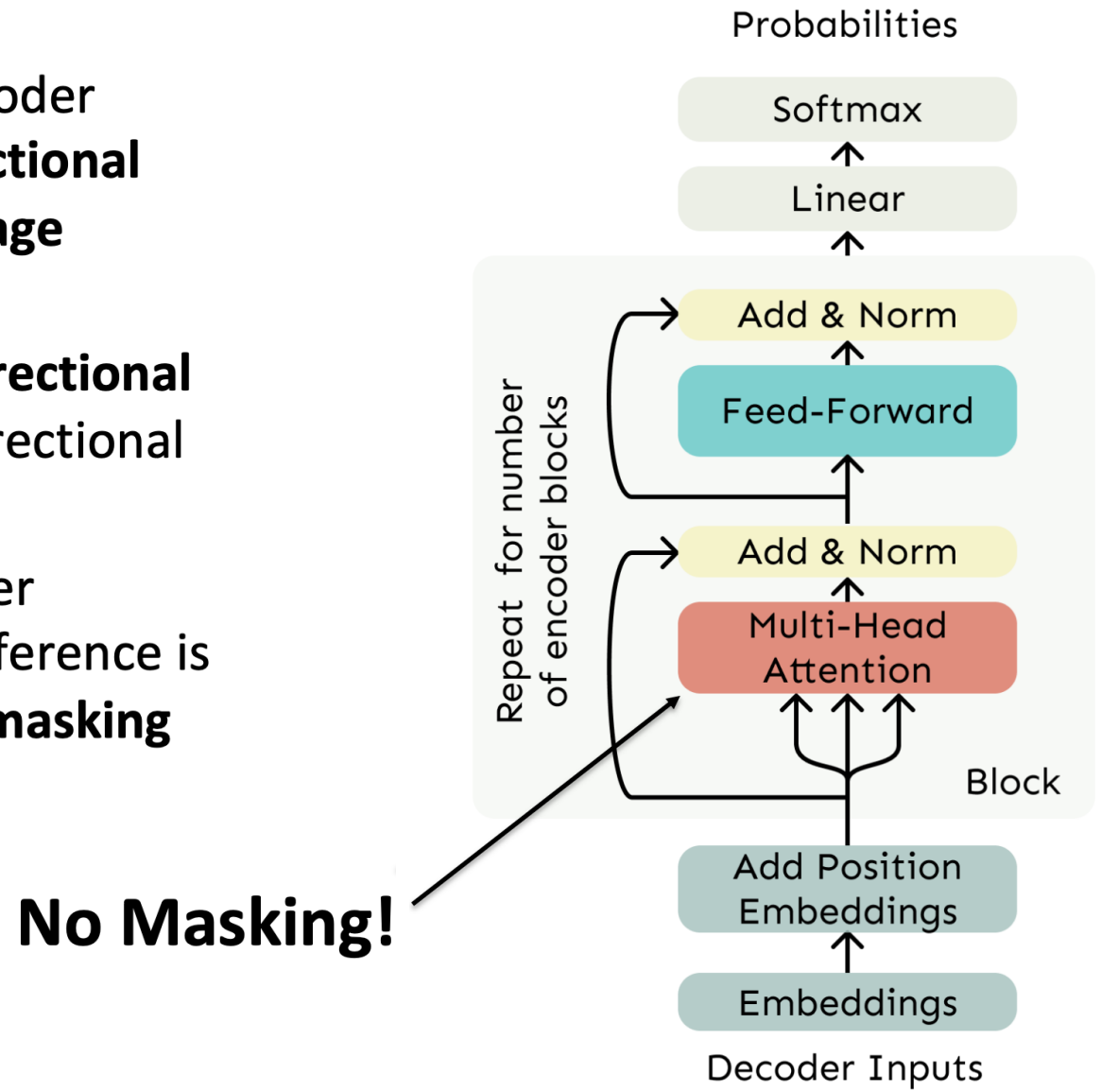
- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^\top k_j, j \leq i \\ -\infty, j > i \end{cases}$$



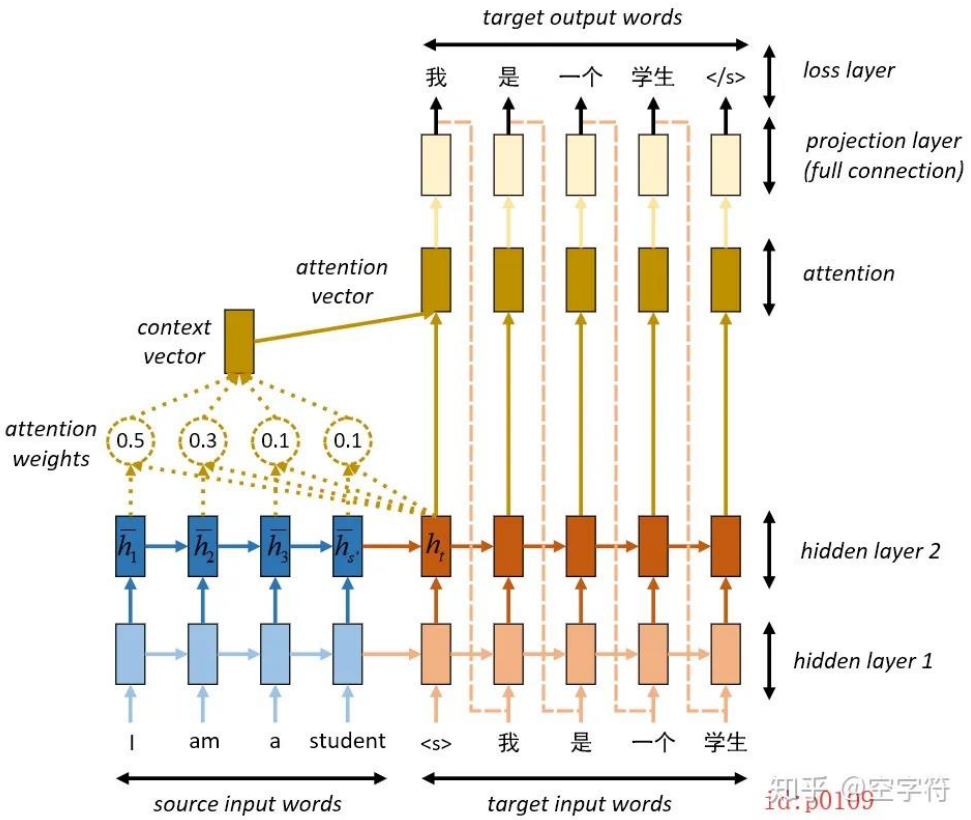
The transformer encoder

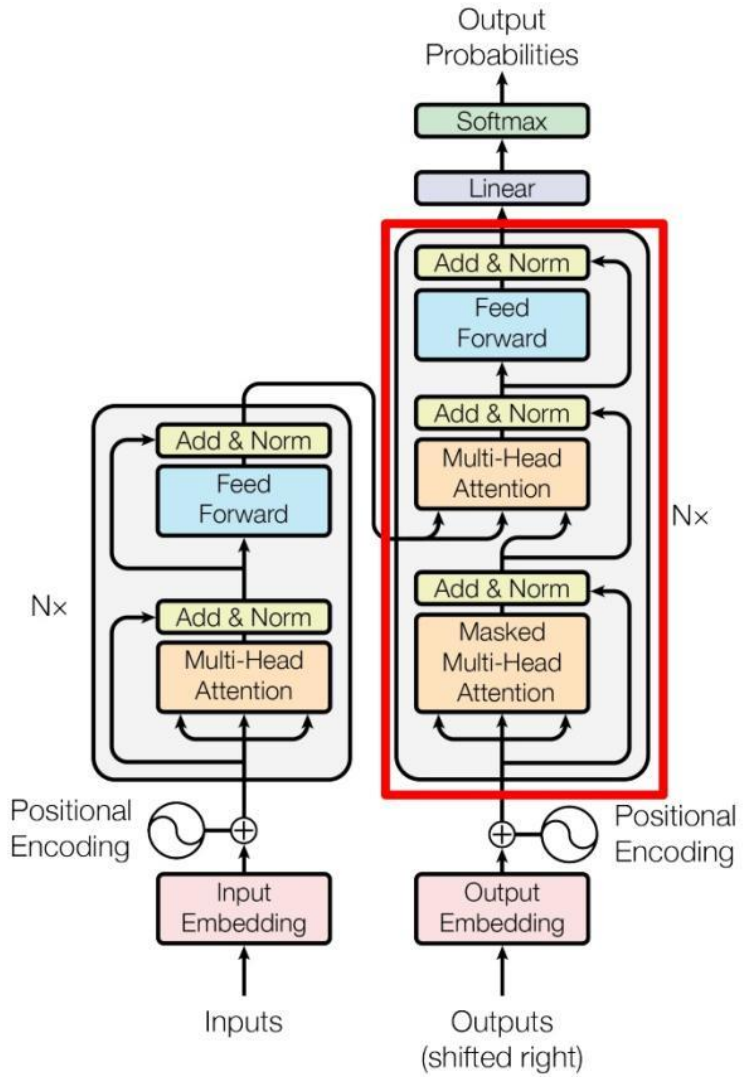
- The Transformer Decoder constrains to **unidirectional context**, as for **language models.**
- What if we want **bidirectional context**, like in a bidirectional RNN?
- This is the Transformer Encoder. The only difference is that we **remove the masking** in the self-attention.



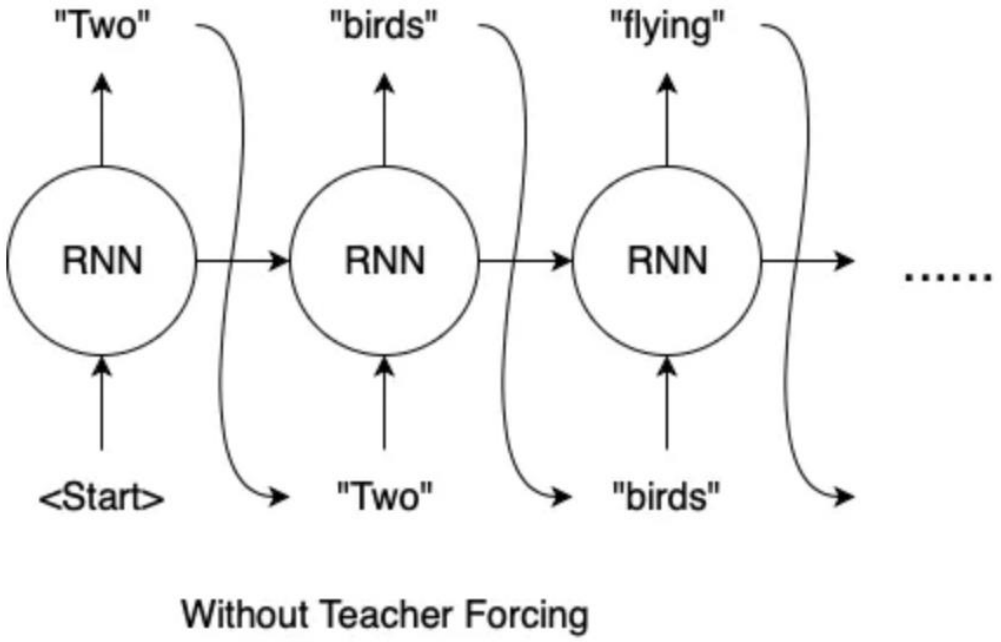
The transformer

RNN

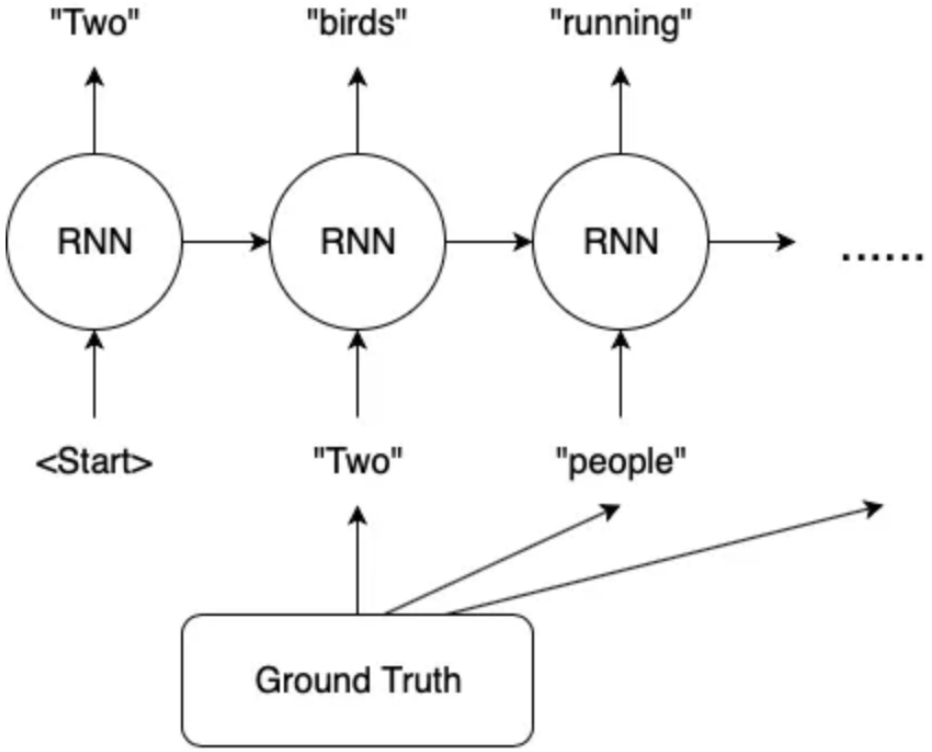




Teacher forcing



Without Teacher Forcing

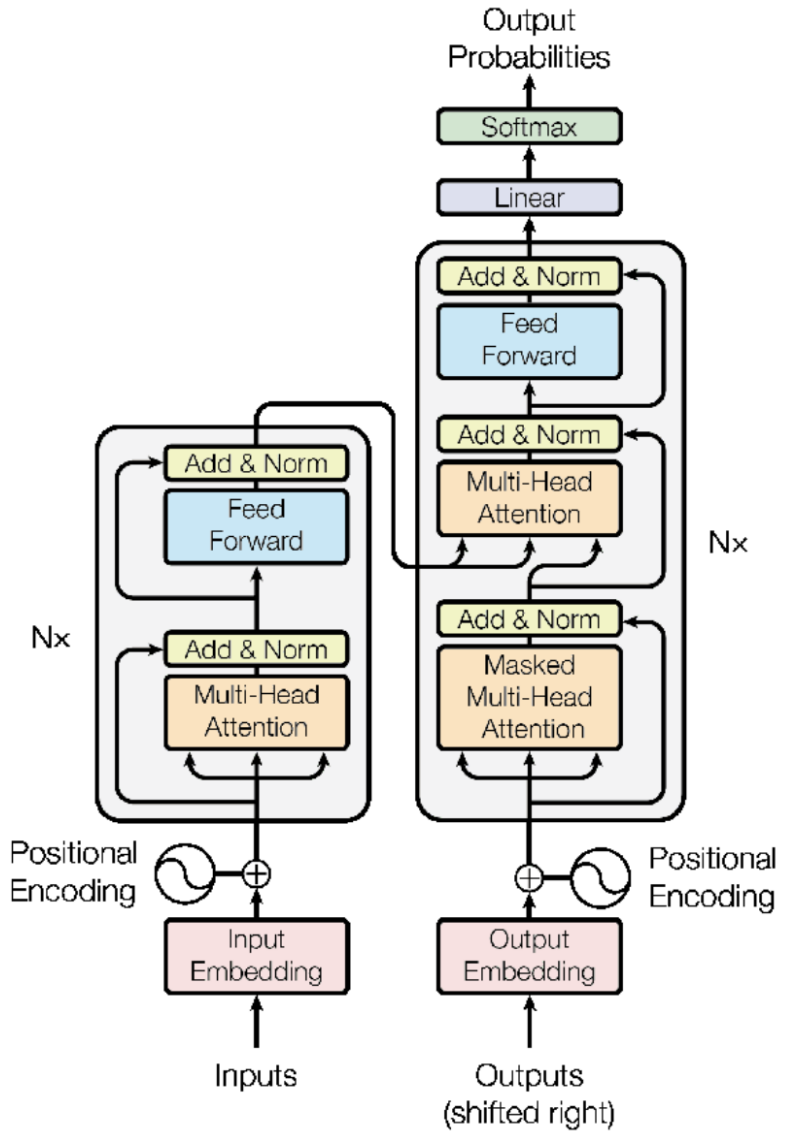


With Teacher Forcing

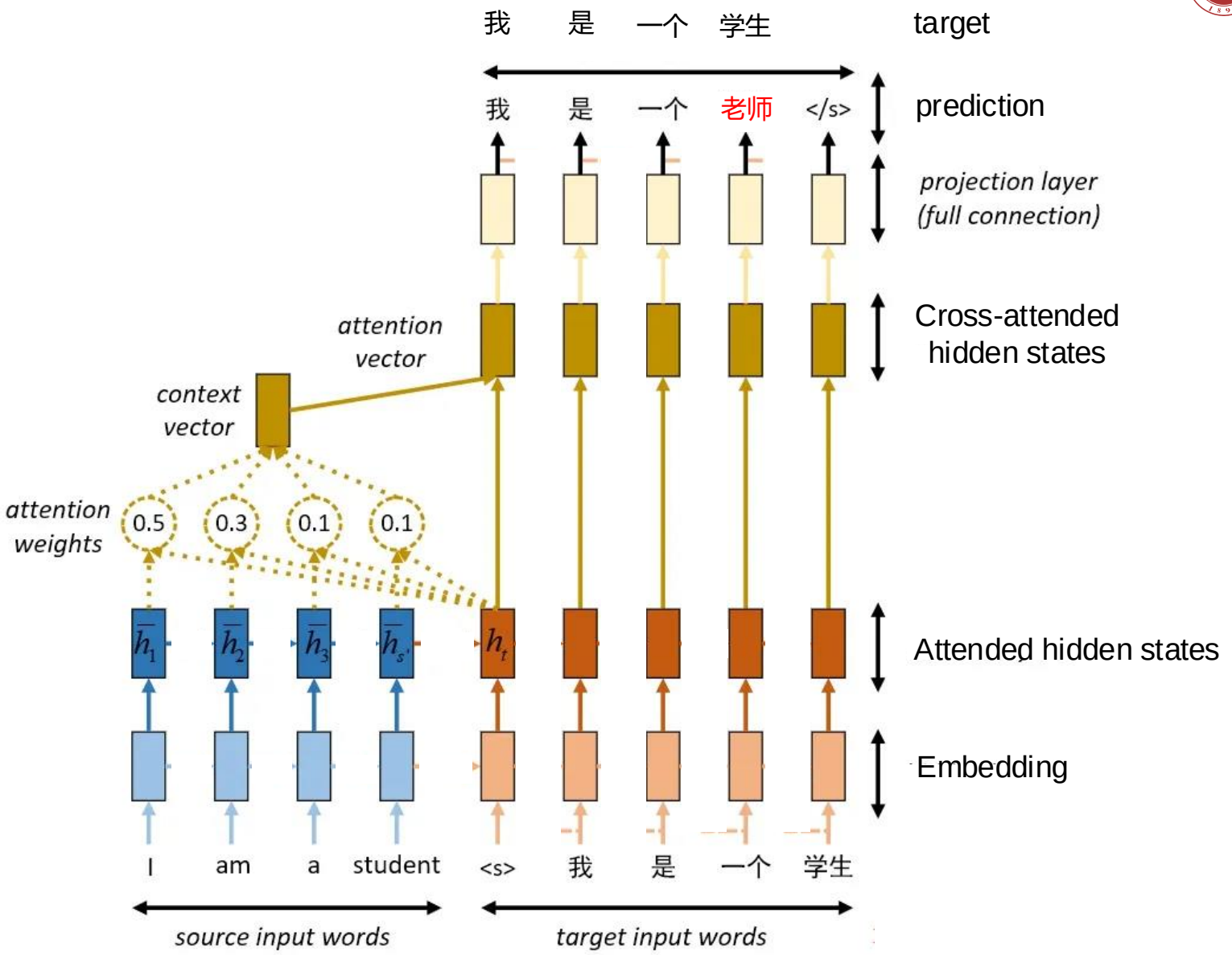
Teacher forcing

Teacher forcing is used when training transformer

- To avoid error accumulation
- To accelerate training due to parallel computing endowed by masked attention



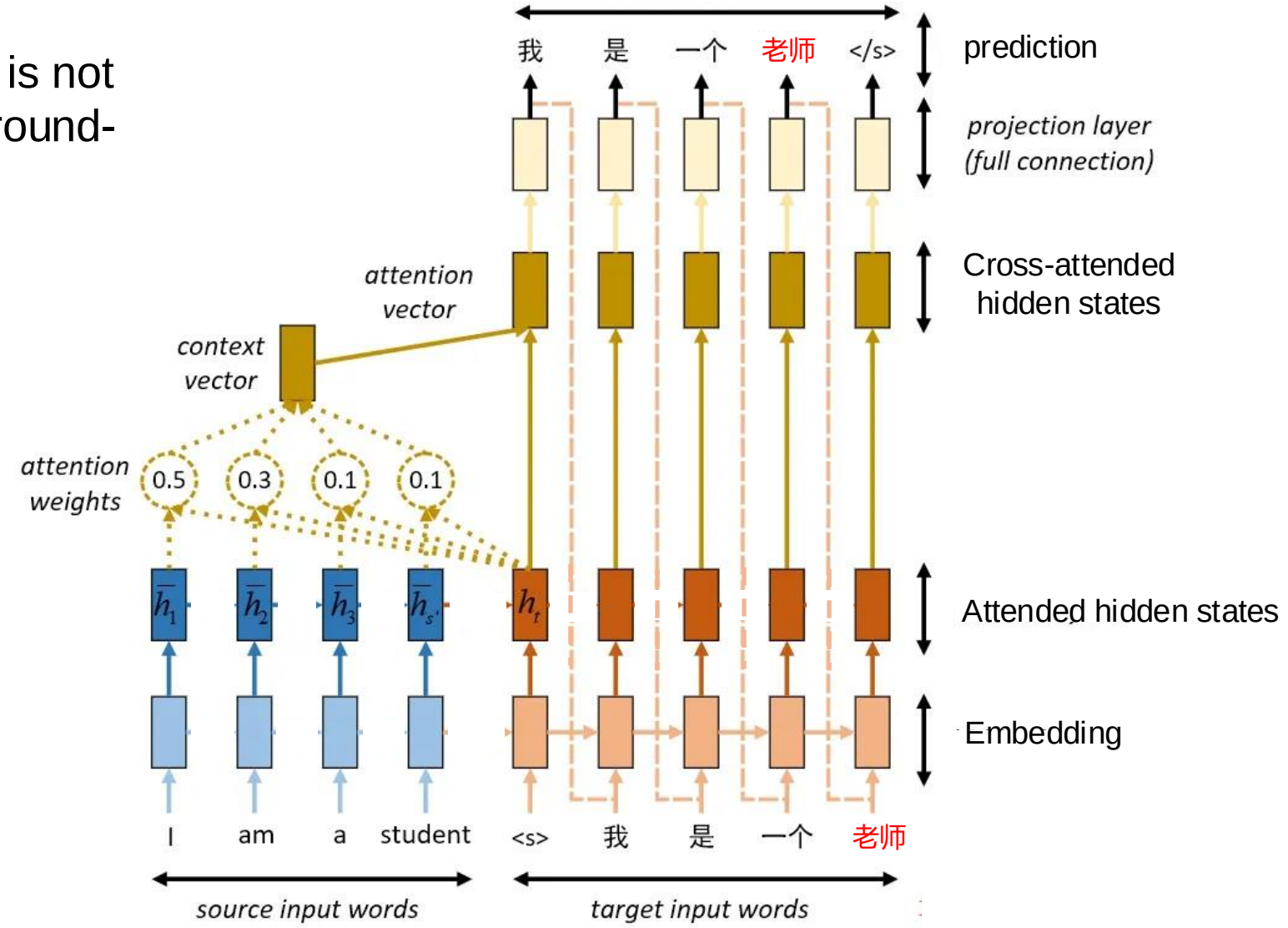
Training transformer with teacher forcing



Inference

During inference, teacher forcing is not used since wo do not have the ground-truth label

- Cannot be parallel!

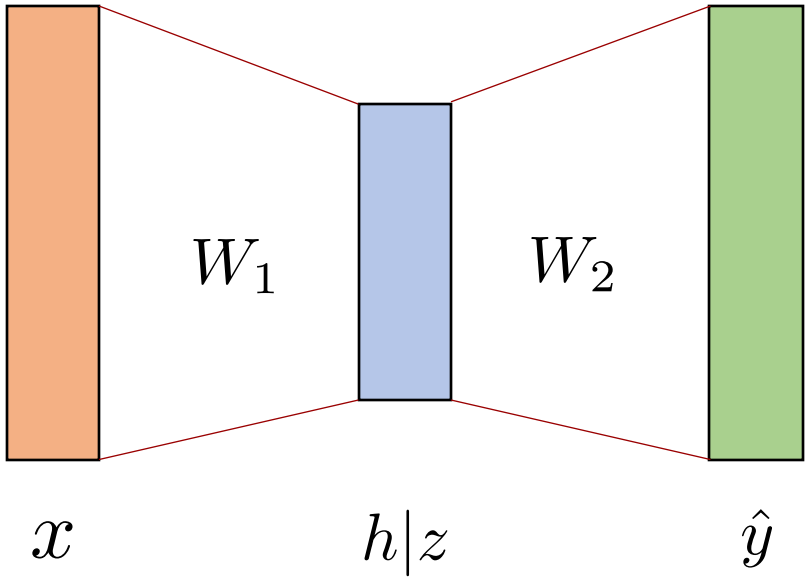


北京大学
PEKING UNIVERSITY

PART 05

Forward and backward propagation

Linear neural network



dims: d (p,d) p (q,p) q

$$h = W_1 x$$

$$z = \sigma(h)$$

$$\hat{y} = W_2 z$$

$$f = L(\hat{y})$$

Forward

$$\frac{\partial f}{\partial W_1} = \frac{\partial f}{\partial h} x^T$$

$$\frac{\partial f}{\partial h} = \frac{\partial f}{\partial z} \odot \nabla\sigma(h)$$

$$\frac{\partial f}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} z^T, \quad \frac{\partial f}{\partial z} = W_2^T \frac{\partial L}{\partial \hat{y}}$$

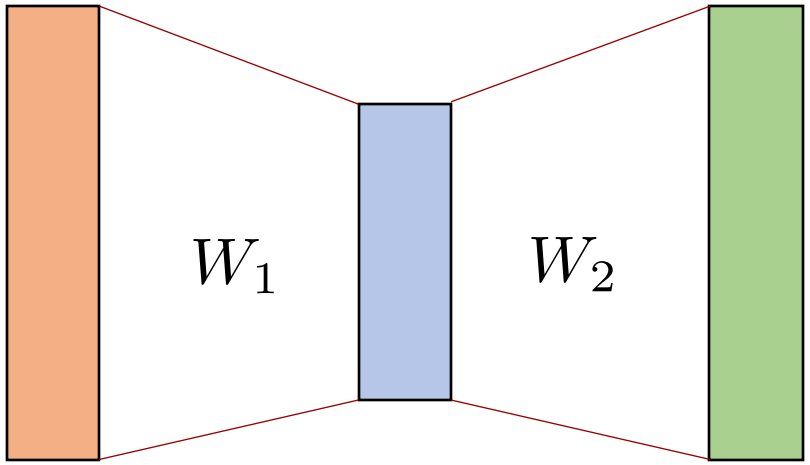
$$\frac{\partial f}{\partial \hat{y}} = \nabla L(\hat{y})$$

Backward

Store h, z and $\hat{y}$

Store $\nabla_{W_1} f(W_1)$ and $\nabla_{W_2} f(W_2)$

Linear neural network with batch size B



dims: d (p,d) p (q,p) q

$$h_b = W_1\, x_b \qquad z_b = \sigma(h_b) \qquad \hat{y}_b = W_2\, z_b$$

$$f = \frac{1}{B}\sum_{b=1}^{B} L(\hat{y}_b)$$

Forward

$$\frac{\partial f}{\partial W_1} = \frac{1}{B}\sum_{b=1}^{B} \frac{\partial f}{\partial h_b} x_b^T, \qquad \frac{\partial f}{\partial h_b} = \frac{\partial f}{\partial z_b} \odot \nabla\sigma(h_b)$$

$$\frac{\partial f}{\partial W_2} = \frac{1}{B}\sum_{b=1}^{B} \frac{\partial L}{\partial \hat{y}_b} z_b^T, \qquad \frac{\partial f}{\partial z_b} = W_2^{\mathrm{T}} \frac{\partial f}{\partial \hat{y}_b}$$

$$\frac{\partial f}{\partial \hat{y}_b} = \frac{\partial L}{\partial \hat{y}_b}$$

Backward

Store $\{\mathbf{h_b}, \mathbf{z_b}, \hat{\boldsymbol{y}}_b\}_{b=1}^{B}$

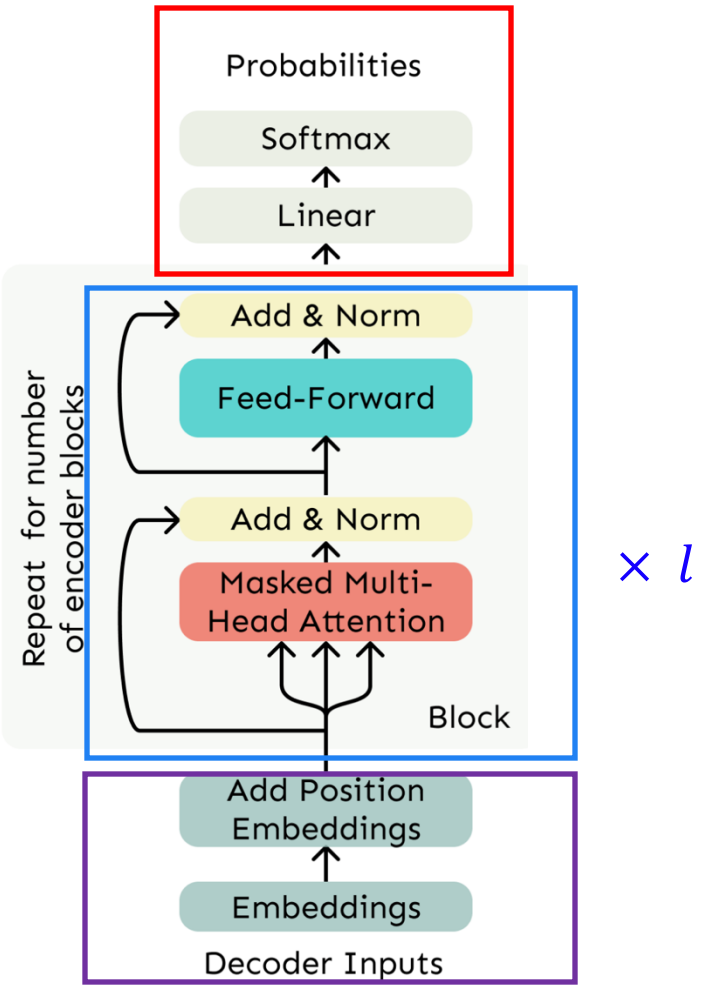
Store $\nabla_{\mathbf{W_1}} \mathbf{f}(\mathbf{W_1})$ and $\nabla_{\mathbf{W_2}} \mathbf{f}(\mathbf{W_2})$

Total parameters

- Embeddings: vh
- Attention blocks: $12lh^2$
- Probability predictions: vh

Total parameters:

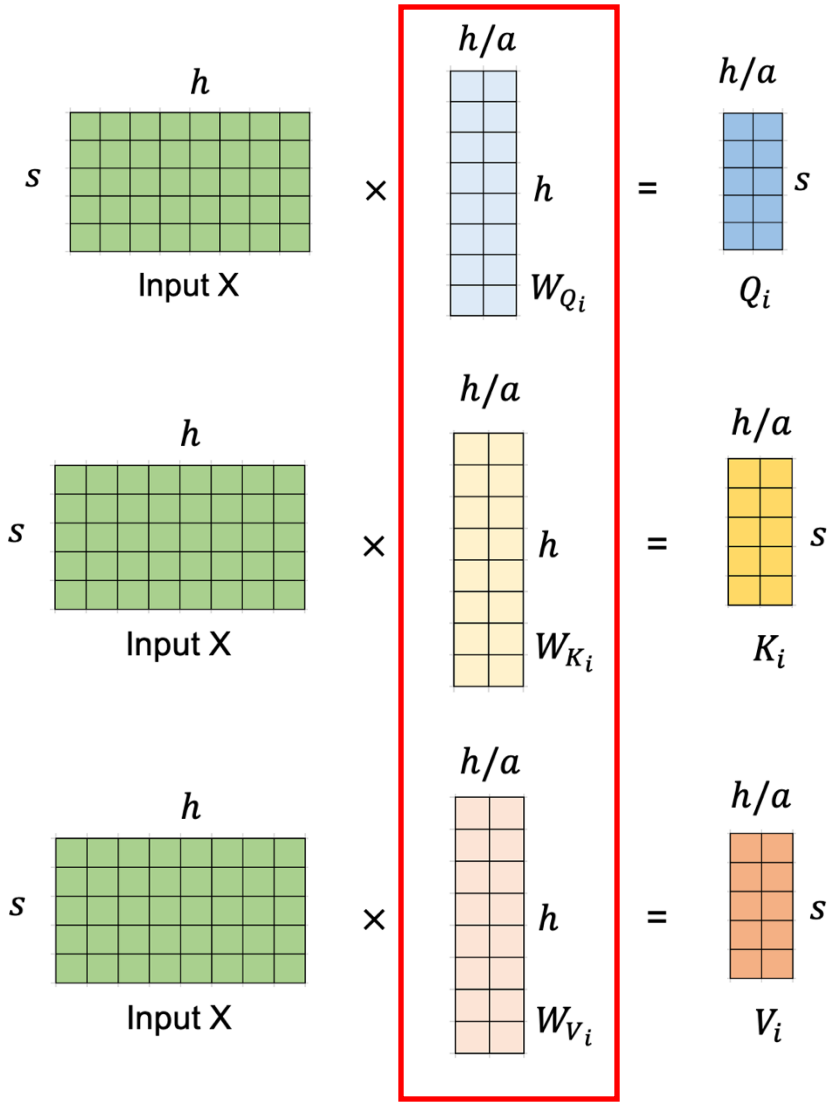
$$12\ell h^2 + 2vh$$

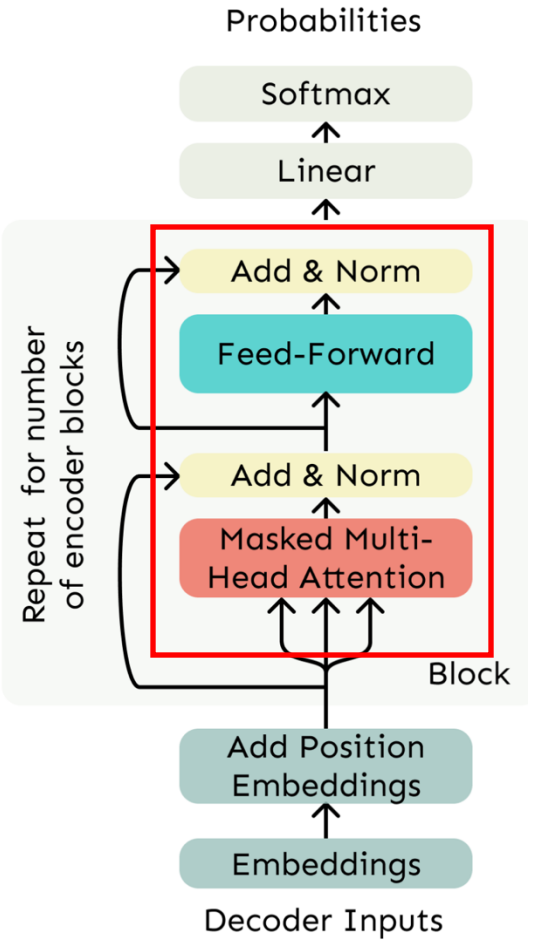


Multi-head attentions

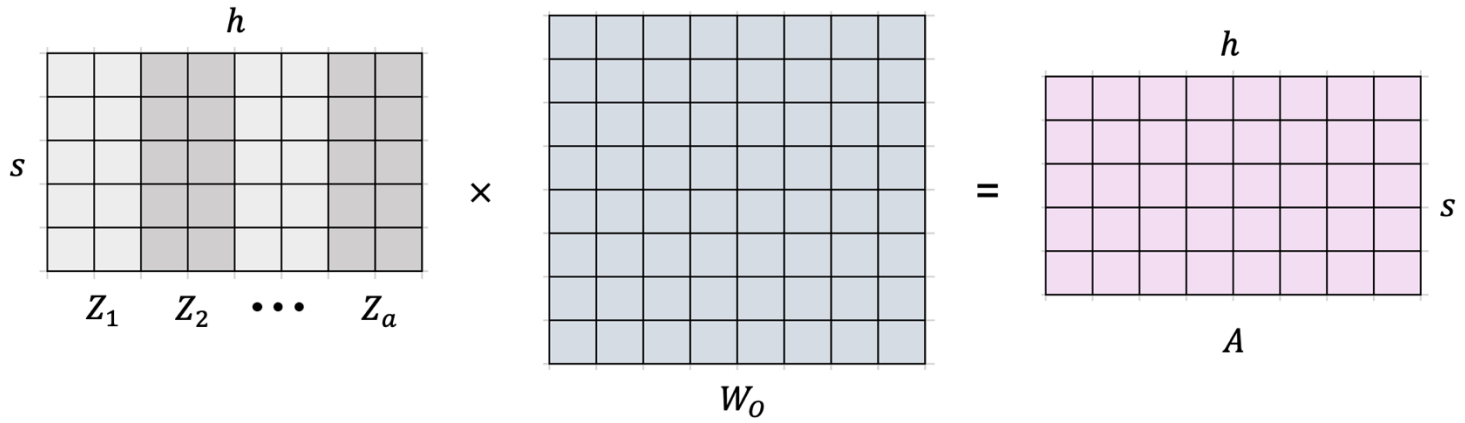
- We need to store W_Q, W_K and W_V

$$3(h^2/a) \times a = 3h^2$$

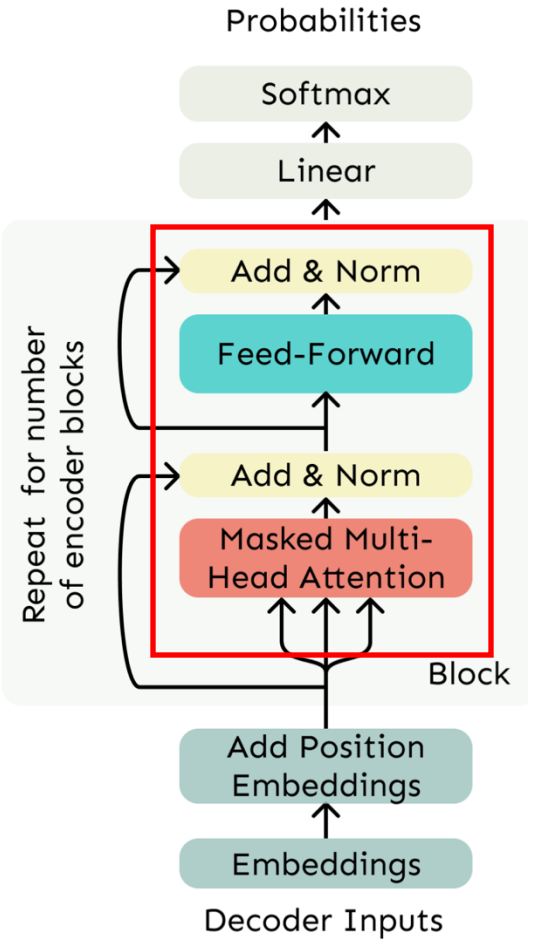




Multi-head attentions



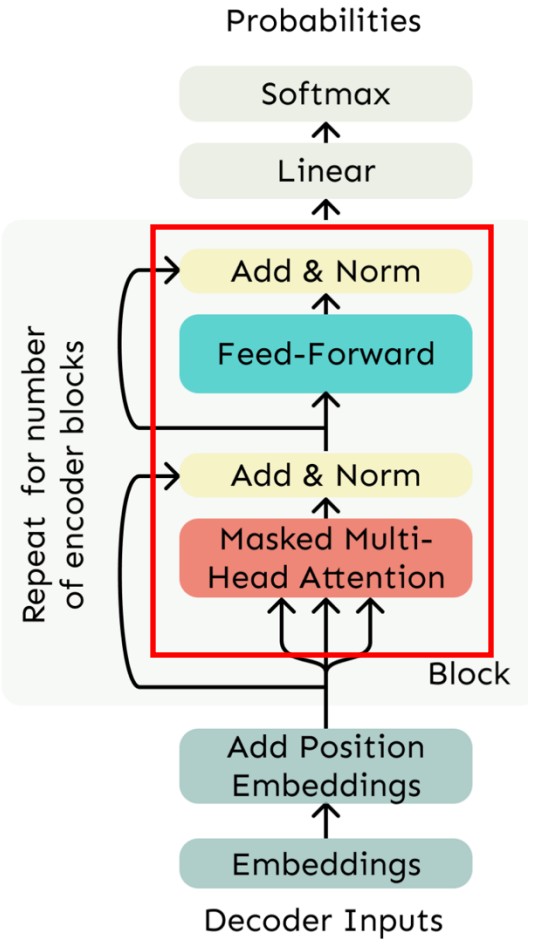
- We need to store W_O: h^2



Feed-forward layers

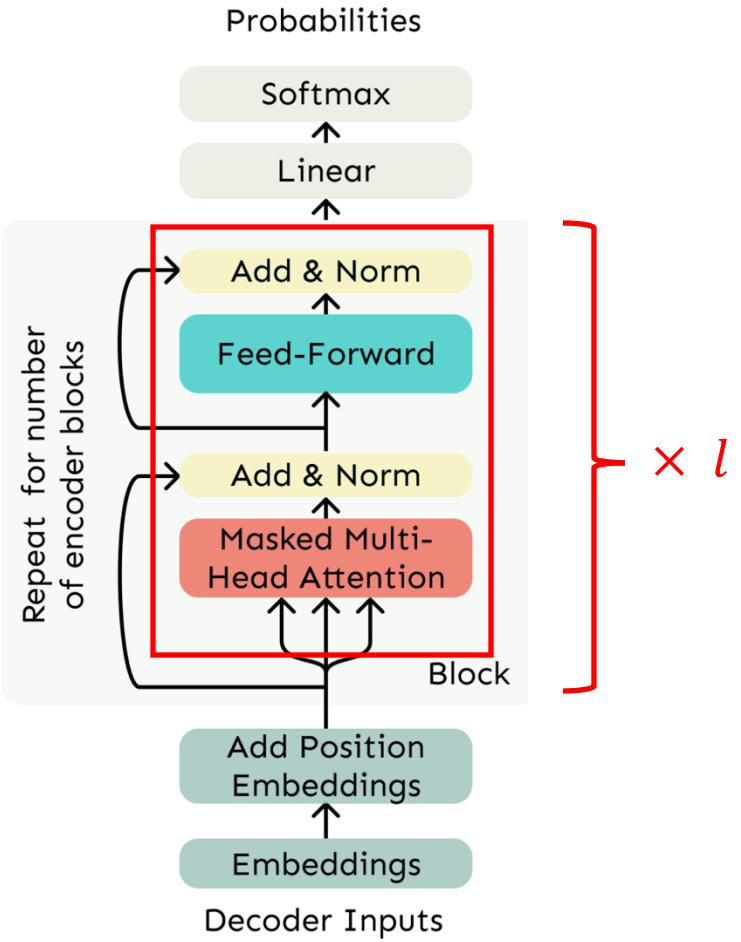
$$X' = \mathrm{ReLU}(A \cdot W_1 + b_1) \cdot W_2 + b_2$$

- Dims of W_1: $h \times 4h$
- Dims of each W_2: $4h \times h$
- We need to store W_1 and W_2 : $8h^2$
- The storage of b_1 and b_2 can be ignored



Transformer block

- Multi-head attentions: $4h^2$
- Feed-forward layers : $8h^2$
- l layers of attentions : $(4h^2 + 8h^2) \times l = 12lh^2$



Total computations

Total forward-backward FLOPs

$$\left(b\ell(24sh^2 + 4s^2h) + 4bsvh\right) \times 3 = 3\left(b\ell(24sh^2 + 4s^2h) + 4bsvh\right)$$

Batch-size b and sequence length s will influence the the computations

Computations increase with s^2

Memory

Memory = Model + Gradient + Optimizer states + Activations

- Given a model with P parameters, gradient will consume P parameters, and Optimizer states will consume 2P parameters; **4P parameters in total.**
- When using FP32 to store parameters, each parameter takes **4** Bytes
- When using FP16 or BF16 to store parameters, each parameter takes **2** Bytes

Total activations

Ignoring 2sv and sh and using batch-size b:

$$(2s^2a + 14sh) \times l \times b$$

- The activation increases fast with sequence length s
- Number of the multi-head influences the activations
- Batch size influences the activations

Total memory

Memory = **Model + Gradient + Optimizer states** + **Activations**

$$(48\,l\,h^2 + bl(2s^2a + 14sh)) \times 4 \text{ Bytes}$$

- When hidden state h is large, the model parameters dominate the memory
- When batch-size b or sequence length s is large, the activation dominates the memory

北京大学
PEKING UNIVERSITY

Try your best and do not leave blanks!

Good Luck!