



Introduction to Large Language Model

Midterm Review

Kun Yuan

PART 01

Optimizer

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 $\{x | a_j^T x \leq b_j, j = 1, \dots, m, c_j^T x = d_j, j = 1, \dots, p\}$

Definition 2 (Convex function)

Function $f : \mathcal{X} \rightarrow \mathbb{R}$ 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 regression 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

Definition 4 (L -smoothness)

A differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 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 cannot vary too quickly

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

- 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

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

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

Theorem 1

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

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

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

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

Its proof is important

- Consider the stochastic optimization problem:

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

- 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

- Recall the problem

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

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

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

where ξ_k is a data realization sampled at iteration k .

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

Let $\mathcal{F}_k = \{x_k, \xi_{k-1}, x_{k-1}, \dots, \xi_0\}$ 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.

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 $\Delta_0 = f(x_0) - f^*$.

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

- 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

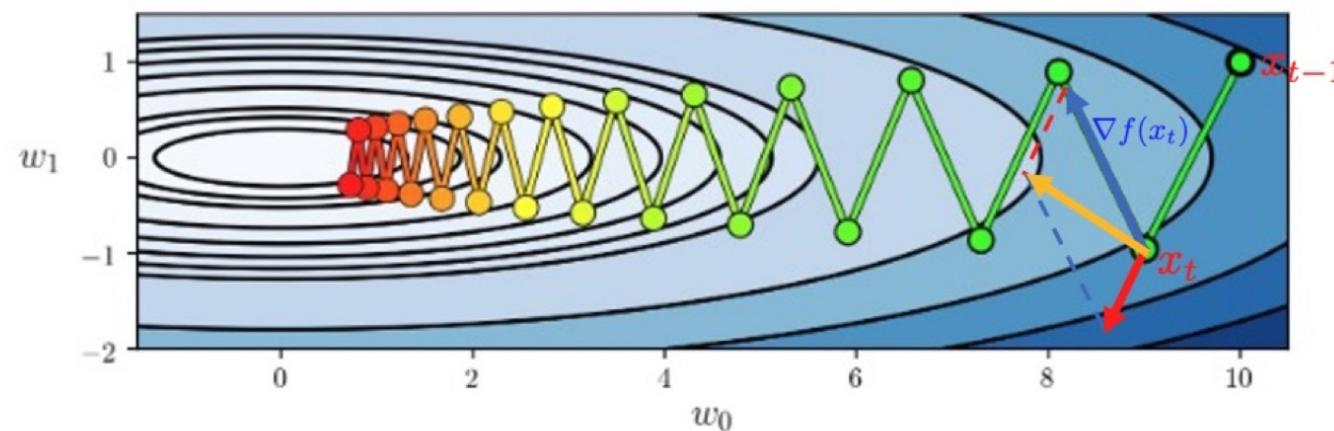


Figure: Momentum can alleviate the “Zig-Zag”

- 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

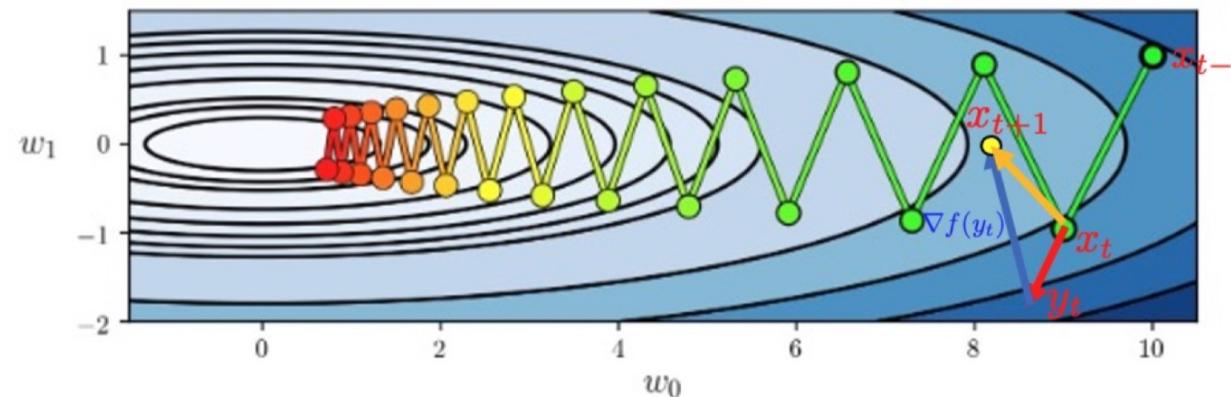


Figure: Nesterov method can alleviate the “Zig-Zag”

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

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

where is called the linear regression problem

- If we introduce

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

problem (2) becomes

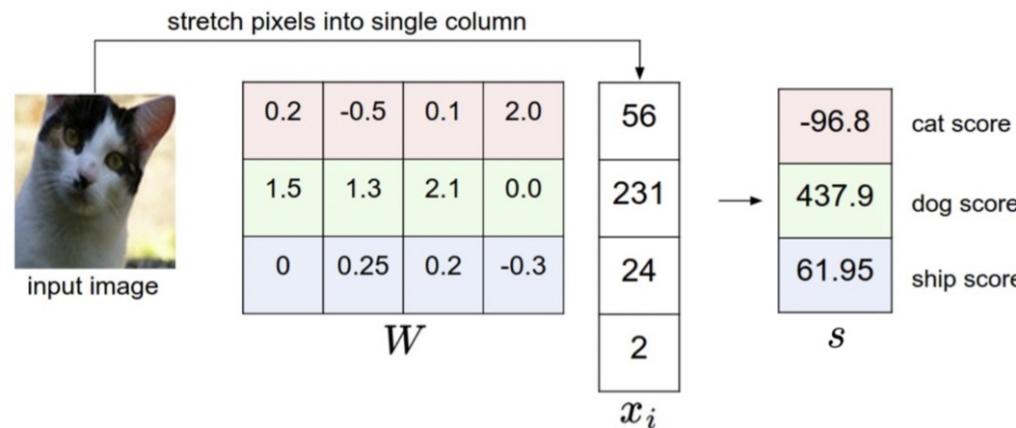
$$\mathbf{w}^* = \arg \min_{\mathbf{w} \in \mathbb{R}^d} \left\{ \frac{1}{2} \|X\mathbf{w} - y\|^2 \right\}$$

Application: multi-classification

- We first collect the dataset $\{\mathbf{x}_i, y_i\}_{i=1}^N$ where $\mathbf{x}_i \in \mathbb{R}^d$ is the feature vector and $y_i \in \{1, \dots, C\}$
- We consider the linear model to predict the score of each class

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

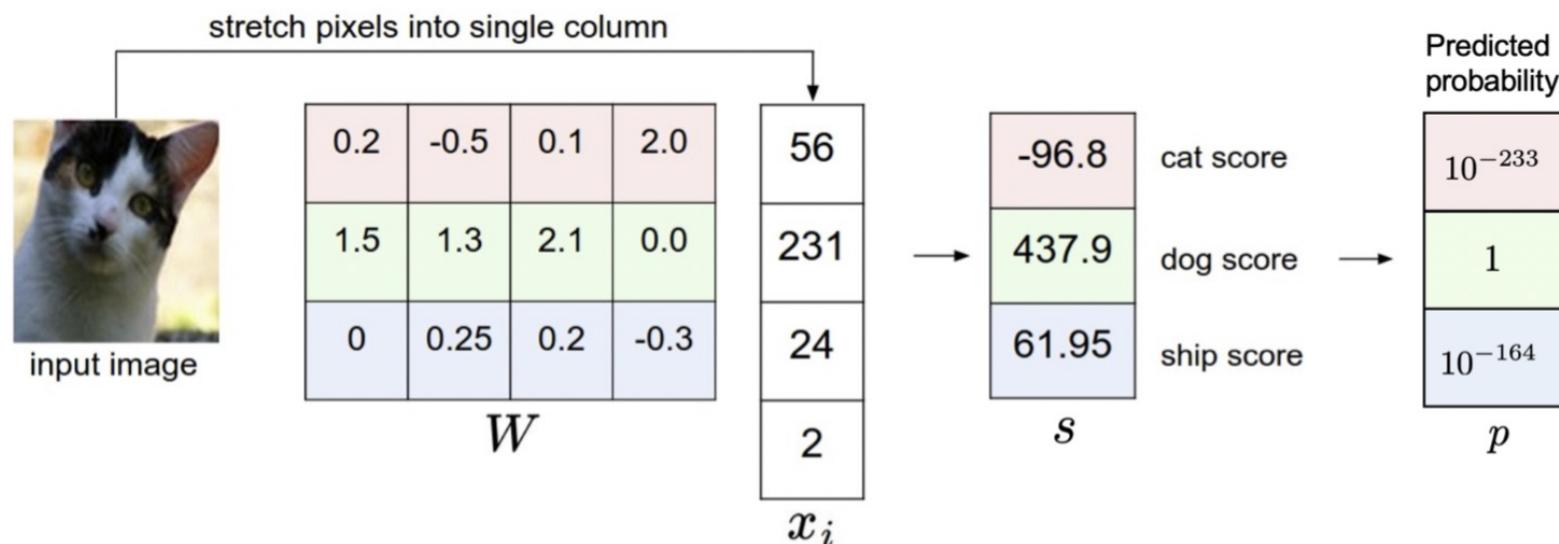
where $W \in \mathbb{R}^{C \times d}$ is the model parameter to learn and s_i indicates the score of the i -th class



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 $\mathbf{p} \in \mathbb{R}^d$ and $\mathbf{q} \in \mathbb{R}^d$

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

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

- Examples:

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

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

Application: multi-classification

- Recall the ground truth probability



Feature vector x

→ Cat →

1
0
0

Label y

Ground-truth
probability

- Now we achieve the loss function in multi-classification

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



PART 02

Language Model

Word semantic representation

- 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

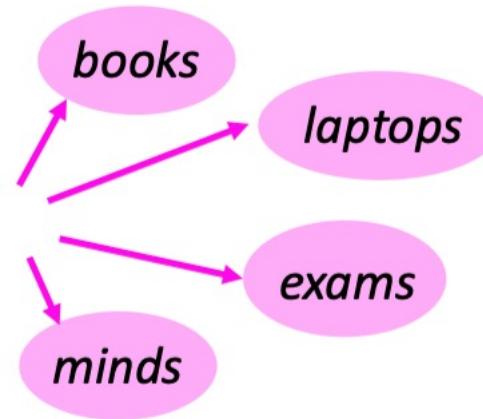
- Given the embedding matrix, we can easily achieve the semantic representation as follows

$$\begin{array}{c} \text{gender} \\ \text{royal} \\ \text{age} \\ \text{food} \\ \vdots \\ \vdots \end{array} \quad \begin{matrix} \text{Embedding matrix} & \times & \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \text{e}_{\text{man}} \end{bmatrix} \end{matrix}$$

Language modeling

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

the students opened their _____



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

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

- 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

$$P(\mathbf{x}^{(t+1)} | \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(1)}) = P(\mathbf{x}^{(t+1)} | \underbrace{\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)}}_{n-1 \text{ words}})$$

(assumption)

prob of a n-gram → $P(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})$
prob of a (n-1)-gram → $P(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})$

(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!

$$\approx \frac{\text{count}(\mathbf{x}^{(t+1)}, \mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})}{\text{count}(\mathbf{x}^{(t)}, \dots, \mathbf{x}^{(t-n+2)})}$$

(statistical approximation)

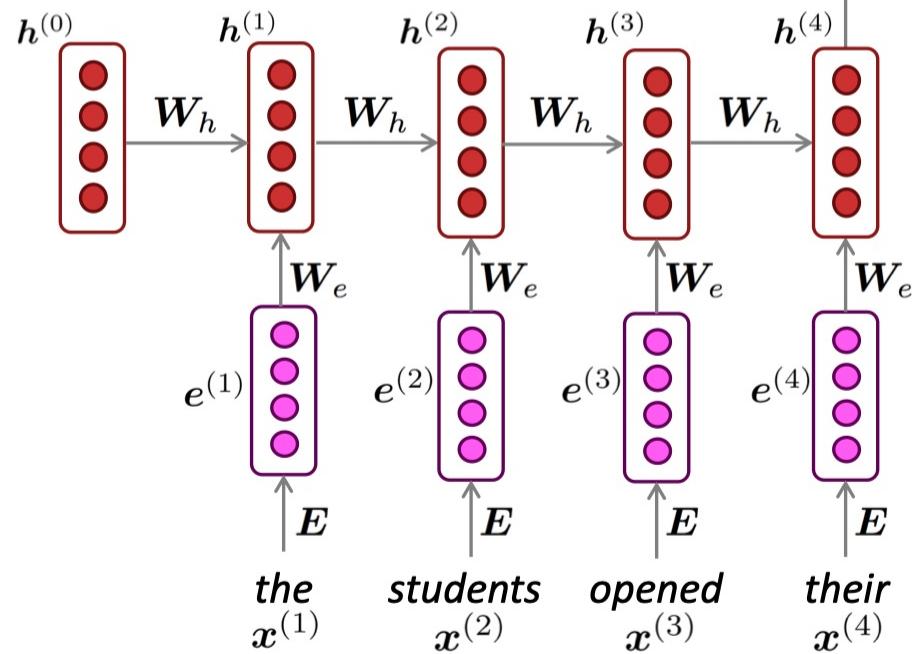
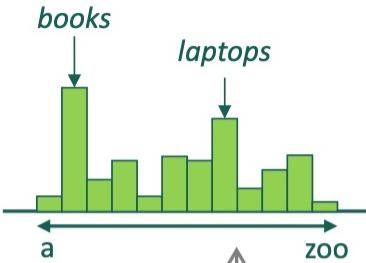


PART 03

RNN and Seq2Seq

Neural network model

$$\hat{\mathbf{y}}^{(4)} = P(\mathbf{x}^{(5)} | \text{the students opened their})$$



output distribution

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

hidden states

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

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

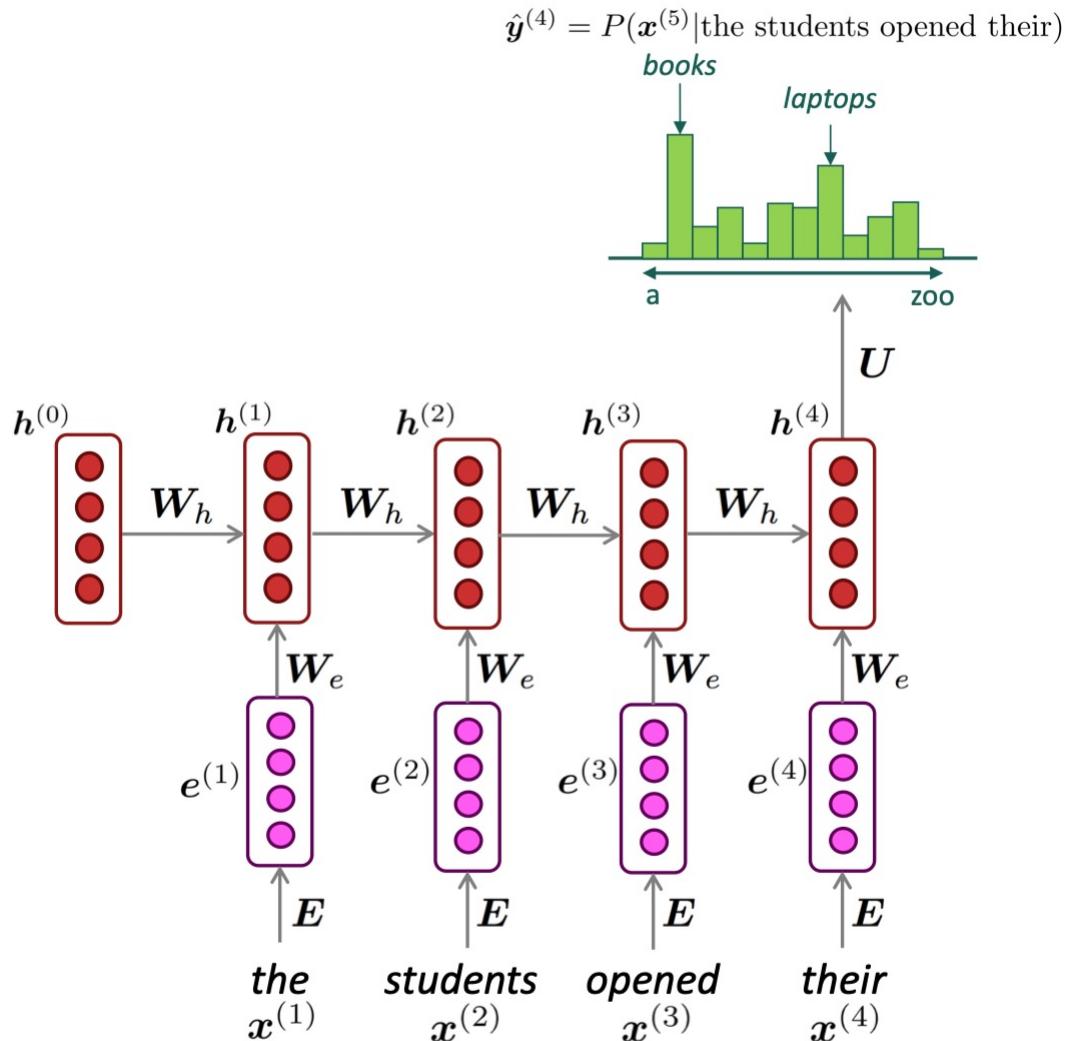
word embeddings

$$\mathbf{e}^{(t)} = \mathbf{E} \mathbf{x}^{(t)}$$

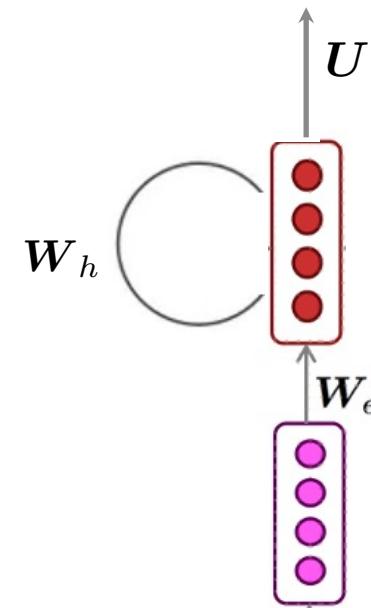
words / one-hot vectors

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

More concise representations

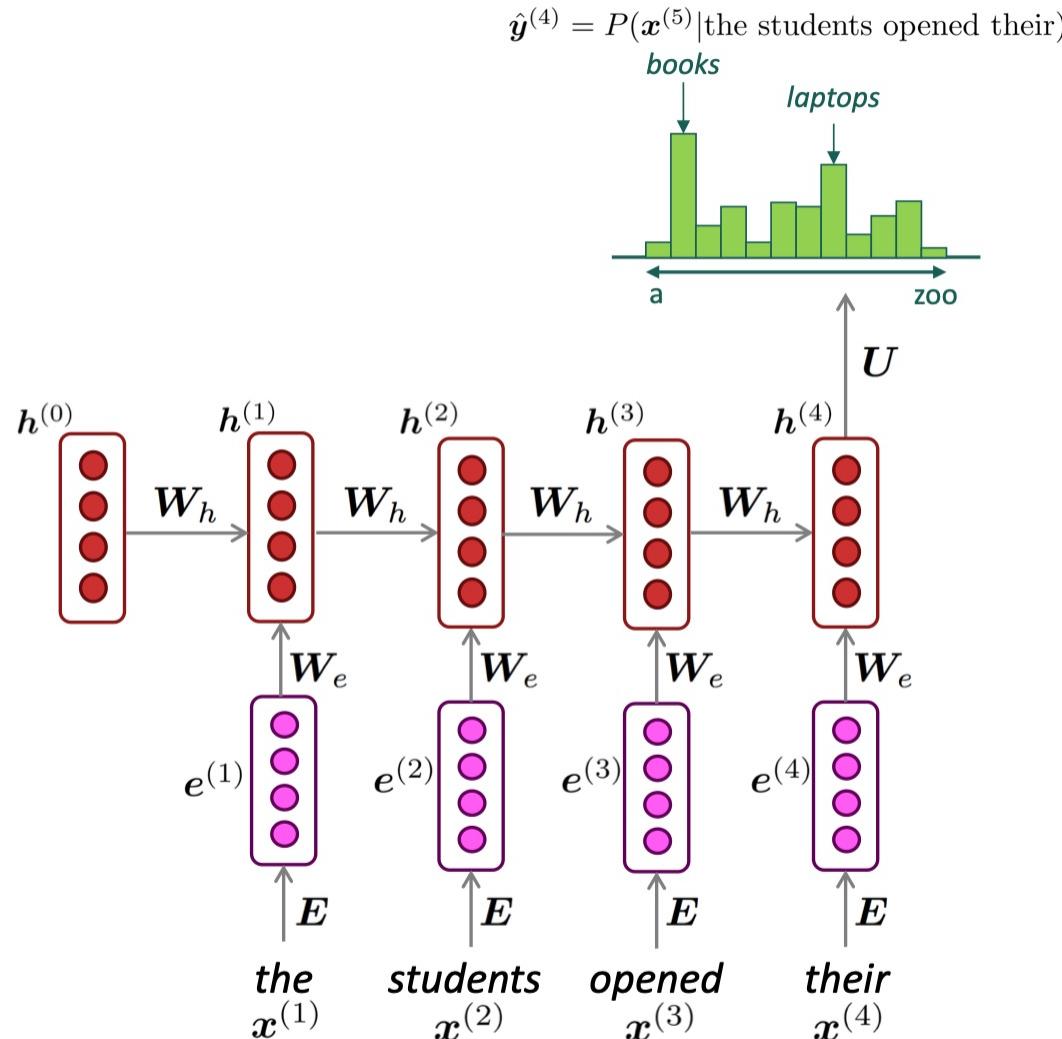


Model weights: $\{\mathbf{W}_h, \mathbf{W}_e, \mathbf{U}, b_1, b_2\}$



$$\begin{aligned}\hat{y}^{(t)} &= \text{softmax}(\mathbf{U}h^{(t)} + \mathbf{b}_2) \in \mathbb{R}^{|V|} \\ h^{(t)} &= \sigma(\mathbf{W}_h h^{(t-1)} + \mathbf{W}_e e^{(t)} + \mathbf{b}_1) \\ h^{(0)} &\text{ is the initial hidden state}\end{aligned}$$

RNN benefits

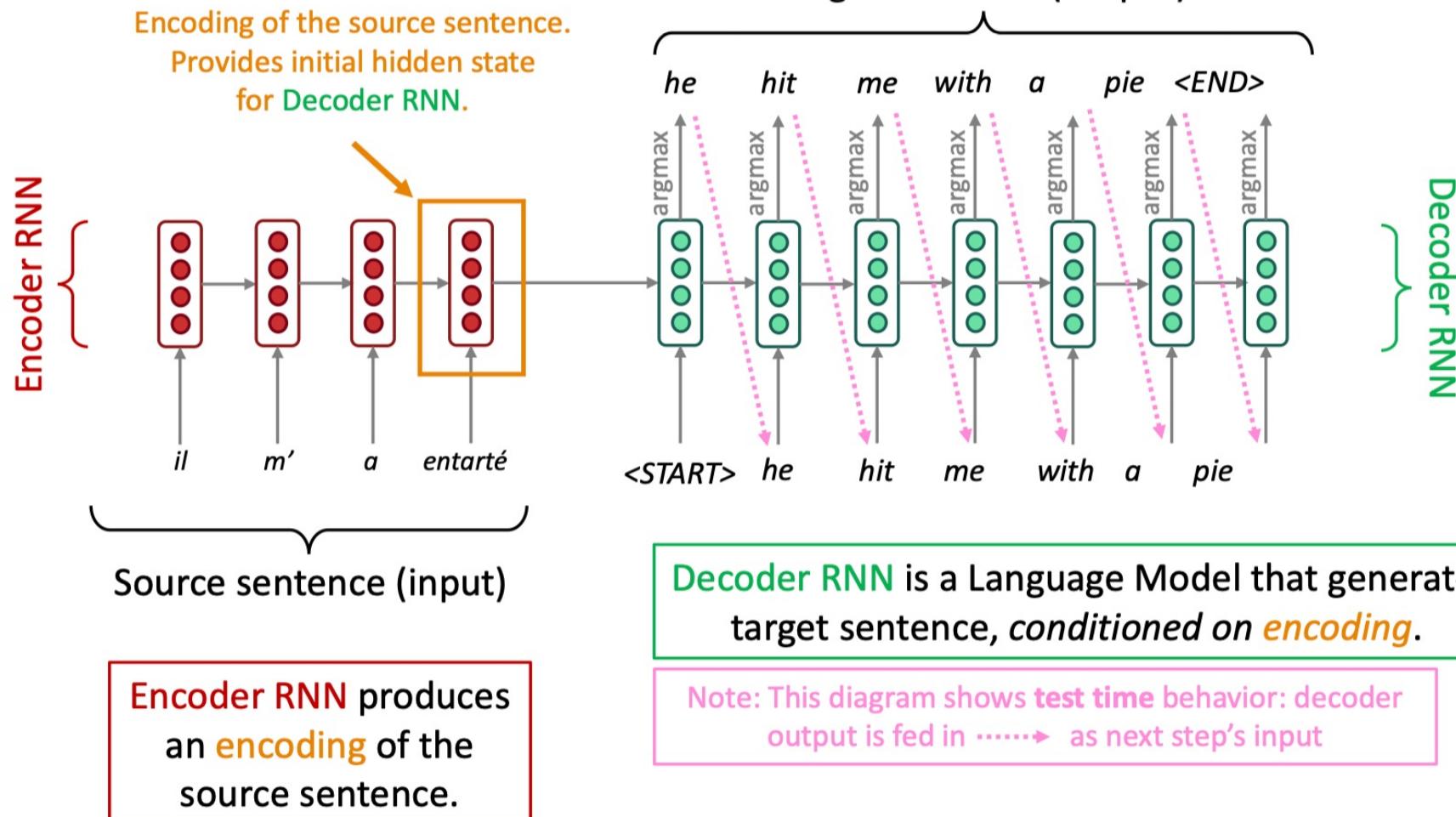


RNN Advantages:

- 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

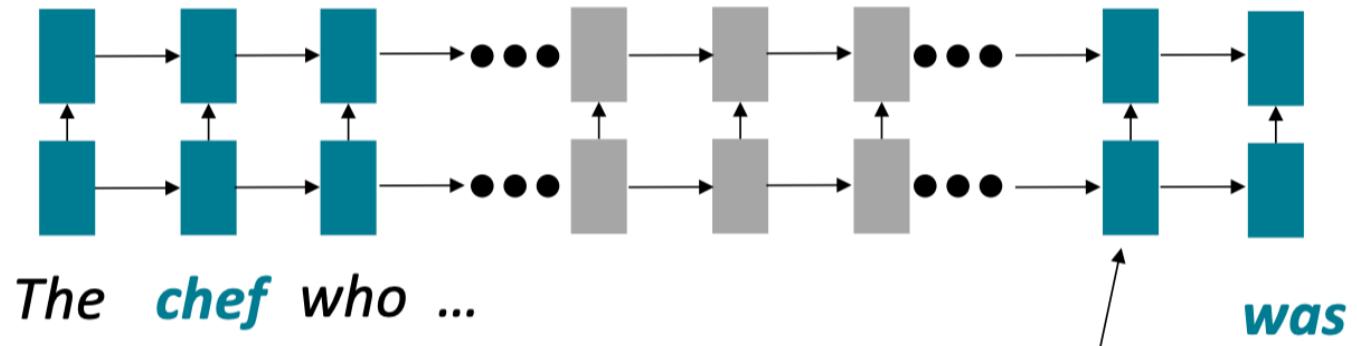
Sequence-to-sequence

- Sequence-to-sequence model



RNN issue I: Linear interaction distance

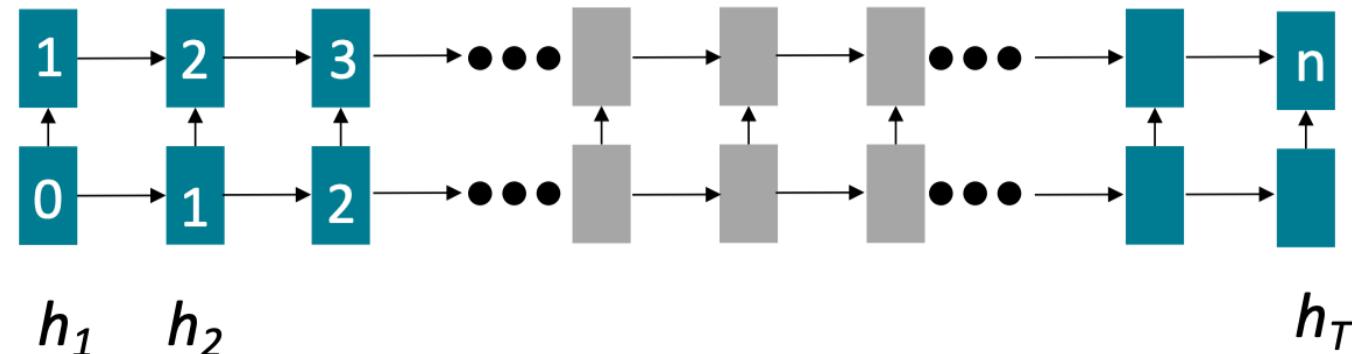
- **O(sequence length)** steps for distant word pairs to interact means:
 - Hard to learn long-distance dependencies (because gradient problems!)
 - Linear order of words is “baked in”; we already know linear order isn’t the right way to think about sentences...



Info of *chef* has gone through
 $O(\text{sequence length})$ many layers!

RNN issue II: Lack of parallelizability

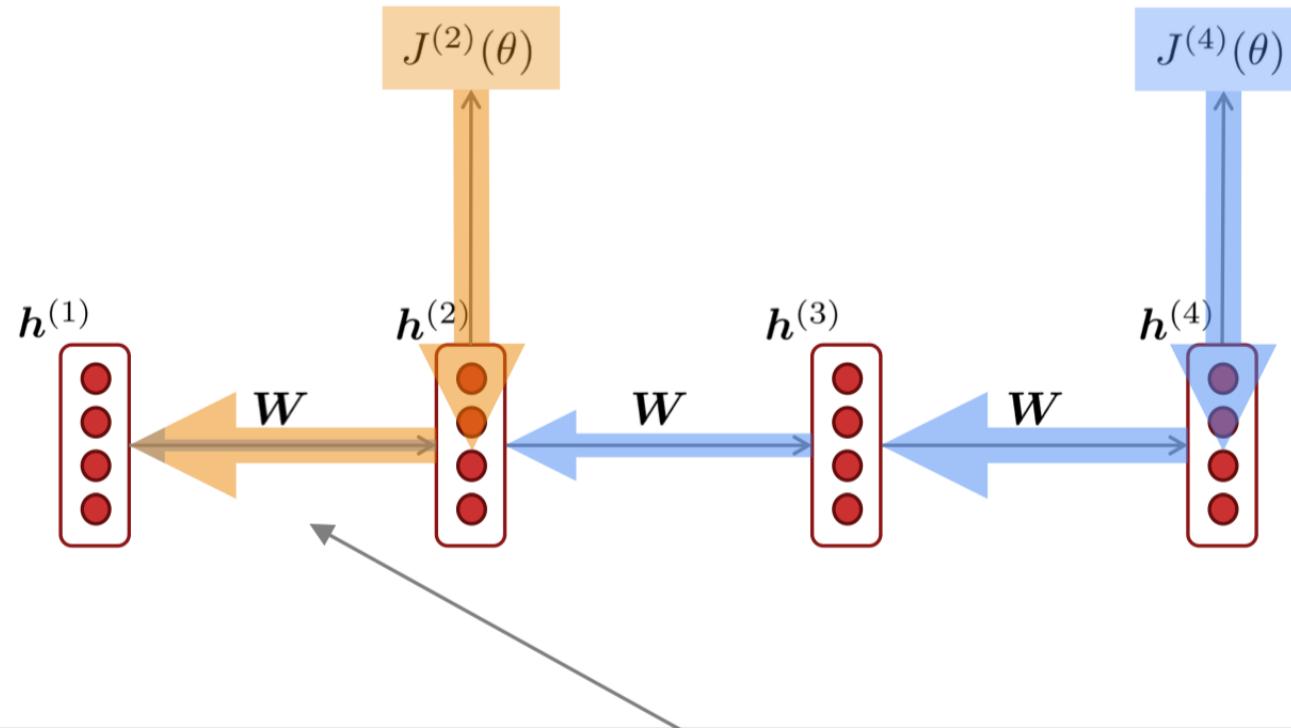
- Forward and backward passes have **O(sequence length)** unparallelizable operations
 - GPUs can perform a bunch of independent computations at once!
 - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
 - Inhibits training on very large datasets!



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

RNN issue III: Gradient vanishing or exploding

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**.



PART 04

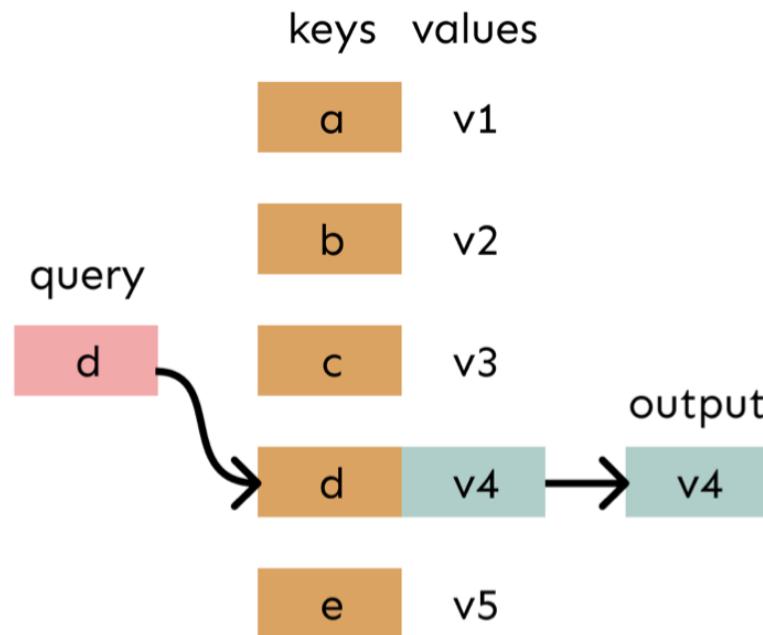
Transformer

- 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

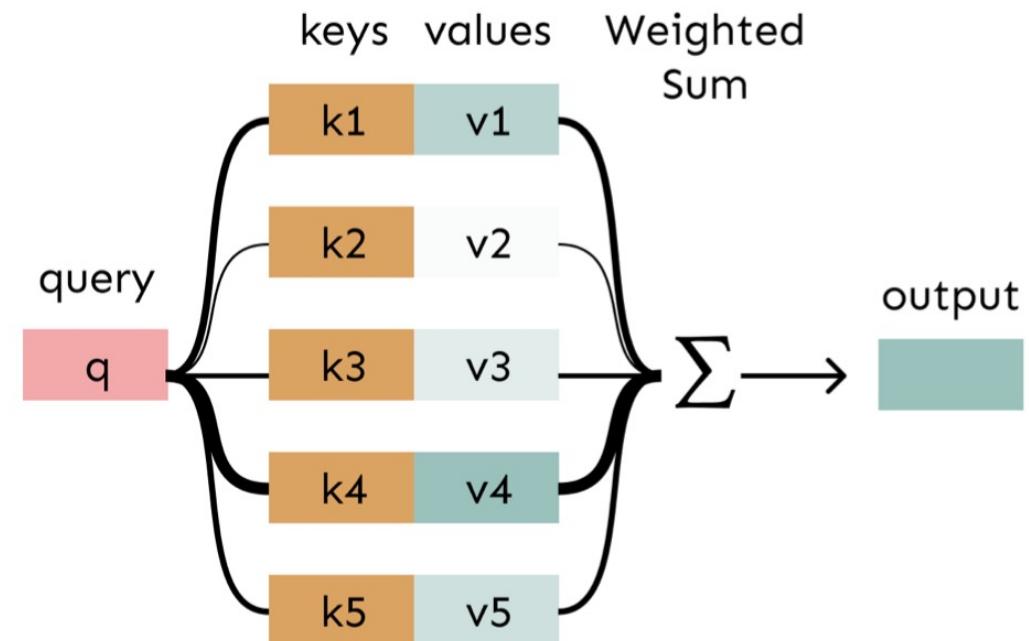
Attention is weighted averaging

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

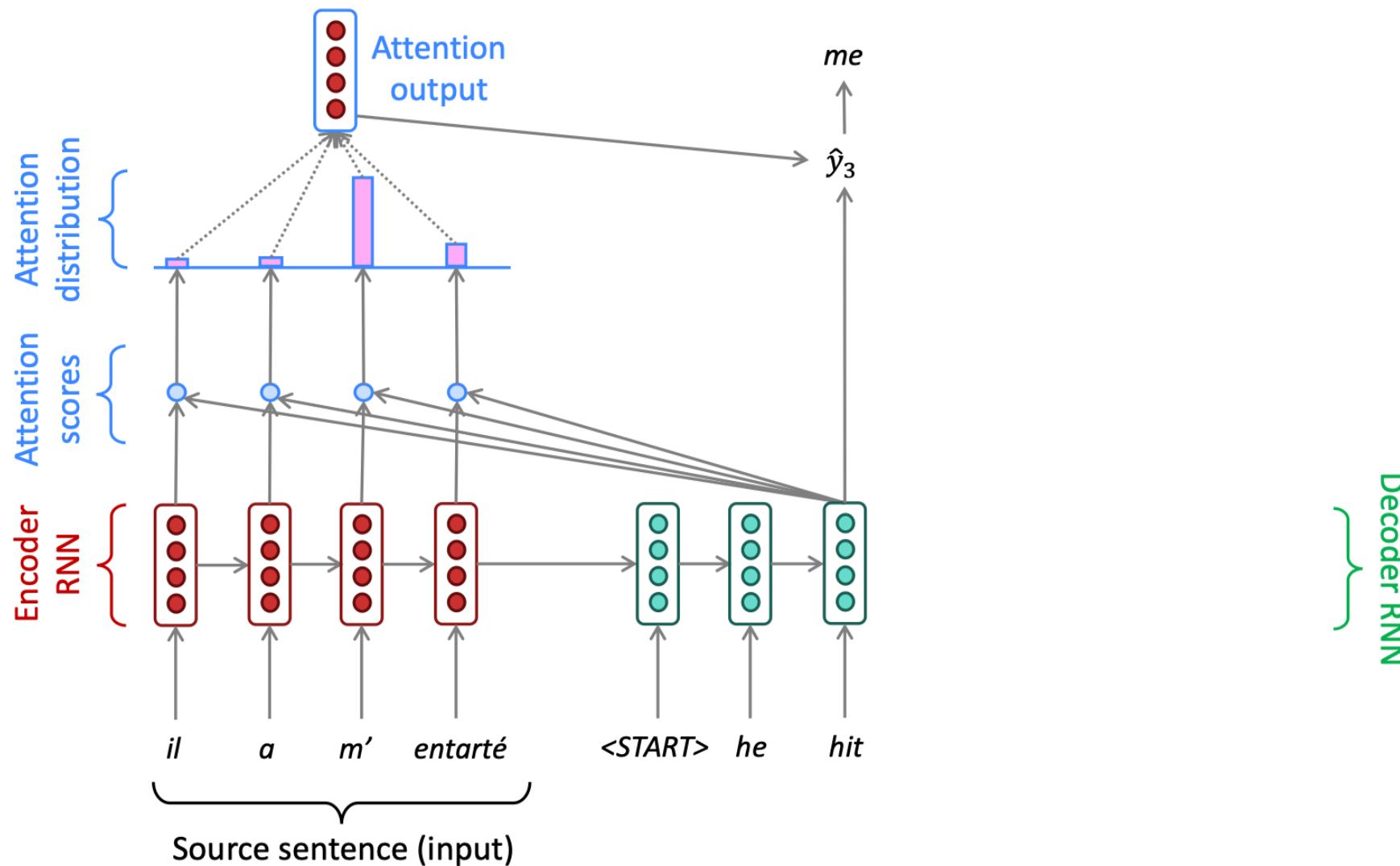
In a **lookup table**, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.



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

$$e^t = [s_t^T h_1, \dots, s_t^T h_N] \in \mathbb{R}^N$$

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

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

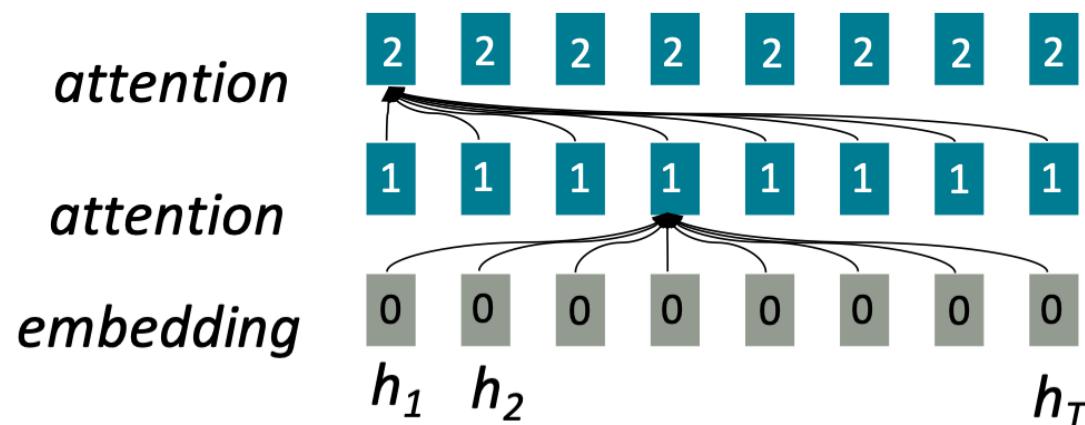
- We use α^t to take a weighted sum of the encoder hidden states to get the attention output a_t

$$a_t = \sum_{i=1}^N \alpha_i^t h_i \in \mathbb{R}^h$$

- Concatenate the attention and the decoder hidden state to proceed $[a_t; s_t] \in \mathbb{R}^{2h}$

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

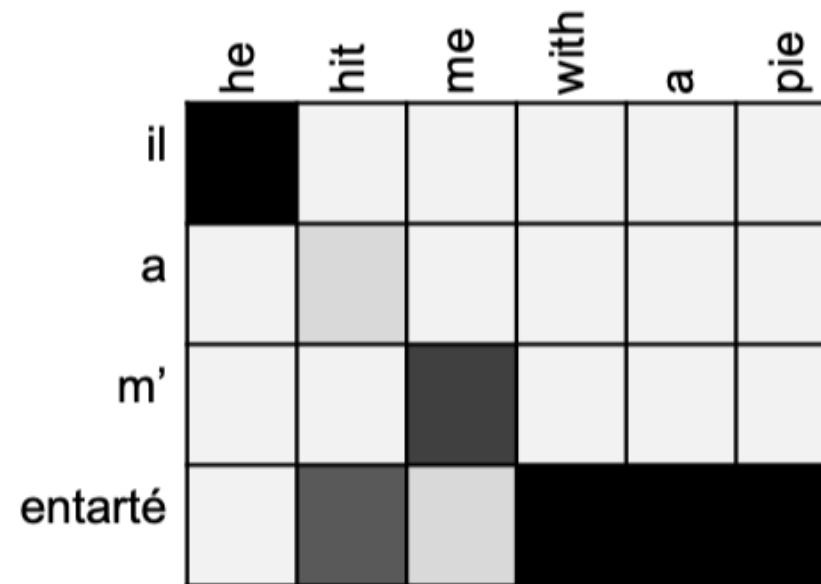
Attention is great



- 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

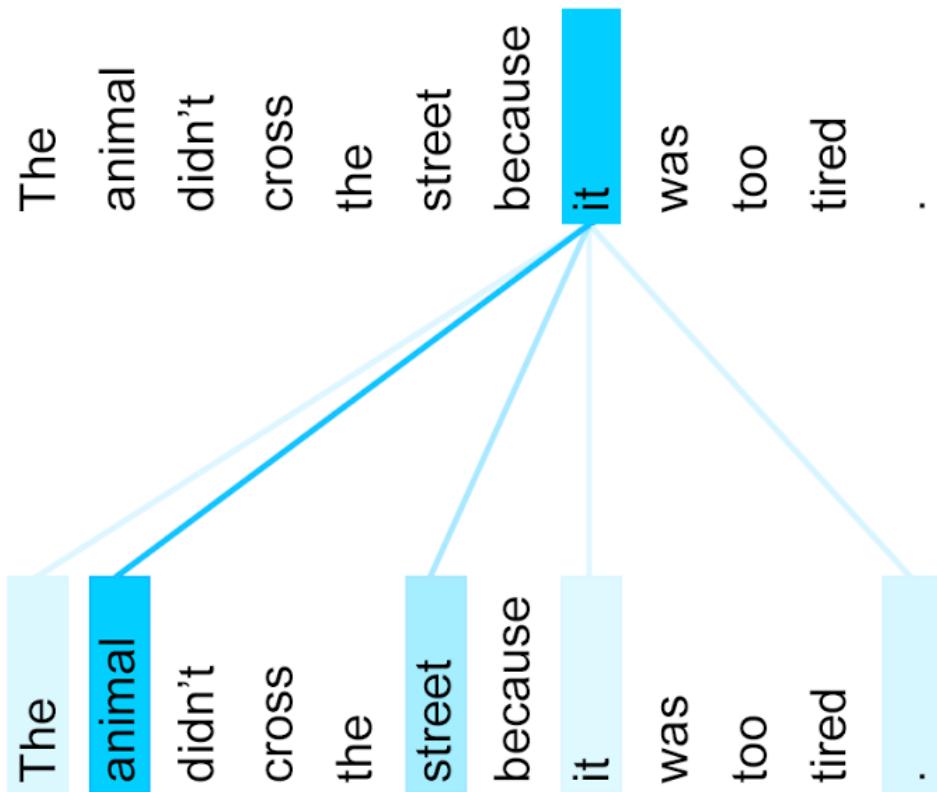
Attention is great

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

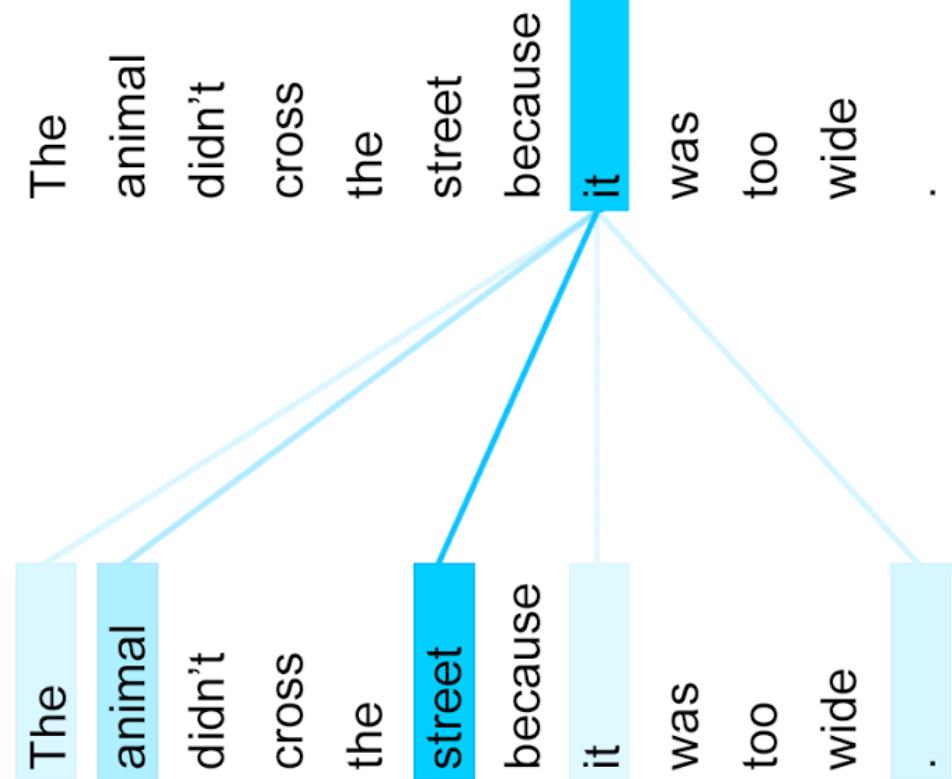


Self attention

The animal didn't cross the street because it was too tired .



The animal didn't cross the street because it was too wide .



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

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

For each \mathbf{w}_i , let $\mathbf{x}_i = E\mathbf{w}_i$, 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 $\mathbb{R}^{d \times d}$

$$\mathbf{q}_i = Q\mathbf{x}_i \text{ (queries)} \quad \mathbf{k}_i = K\mathbf{x}_i \text{ (keys)} \quad \mathbf{v}_i = V\mathbf{x}_i \text{ (values)}$$

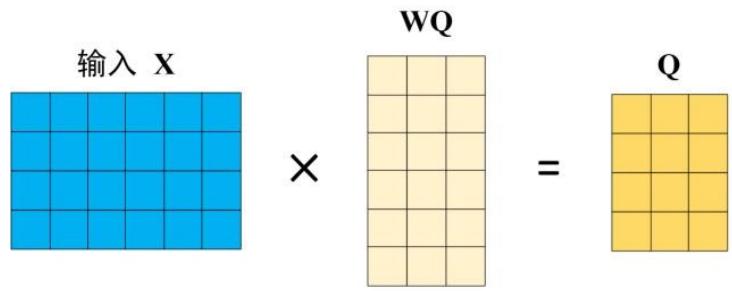
2. Compute pairwise similarities between keys and queries; normalize with softmax

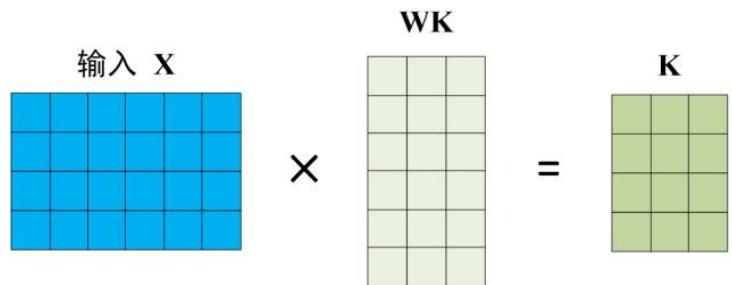
$$\mathbf{e}_{ij} = \mathbf{q}_i^\top \mathbf{k}_j \quad \alpha_{ij} = \frac{\exp(\mathbf{e}_{ij})}{\sum_{j'} \exp(\mathbf{e}_{ij'})}$$

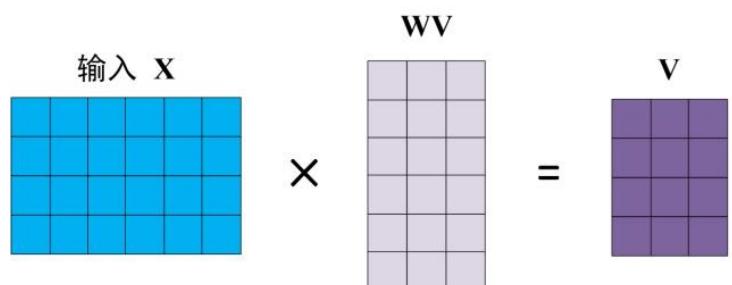
3. Compute output for each word as weighted sum of values

$$\mathbf{o}_i = \sum_j \alpha_{ij} \mathbf{v}_i$$

Self-Attention: matrix representation

$$\text{输入 } \mathbf{X} \times \mathbf{WQ} = \mathbf{Q}$$


$$\text{输入 } \mathbf{X} \times \mathbf{WK} = \mathbf{K}$$


$$\text{输入 } \mathbf{X} \times \mathbf{WV} = \mathbf{V}$$


Self-Attention: matrix representation

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

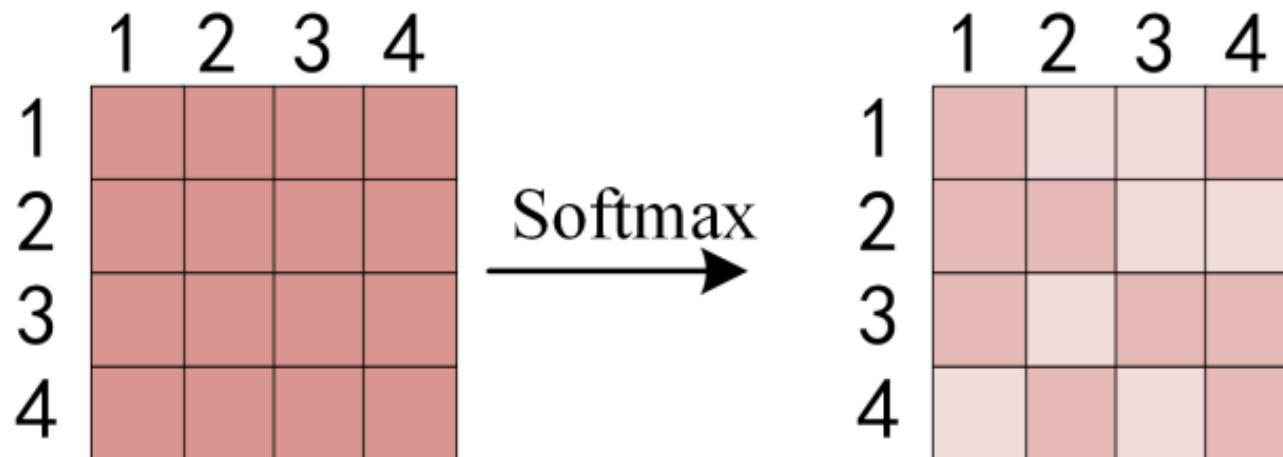
d_k 是 Q, K 矩阵的列数，即向量维度

$$\begin{matrix} & \mathbf{Q} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \times & \mathbf{K}^T \\ & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{matrix} & = & \mathbf{QK}^T \\ & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{matrix} & & \end{matrix}$$

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

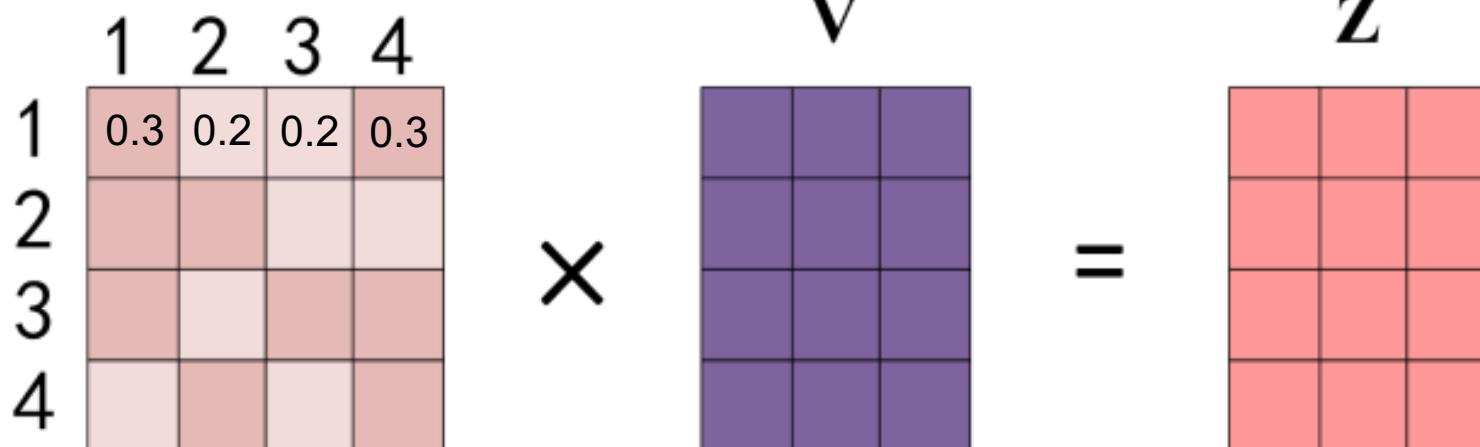


Self-Attention: matrix representation

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

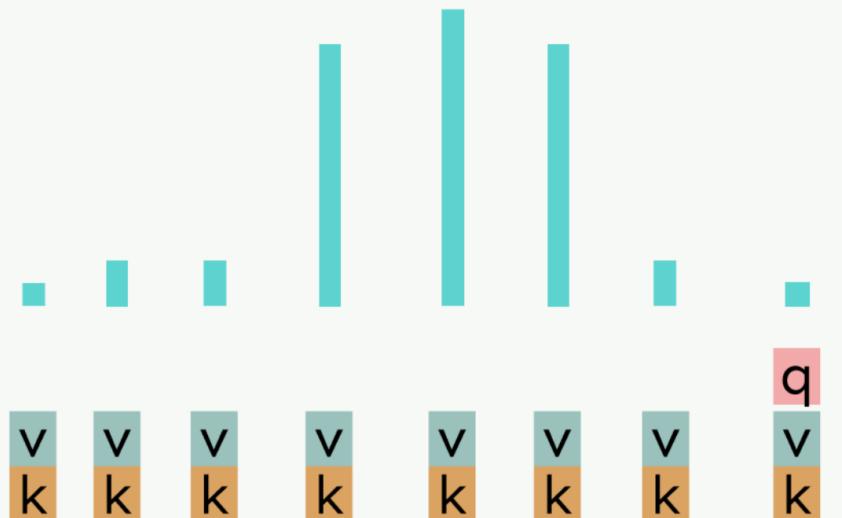
d_k 是 Q, K 矩阵的列数，即向量维度

Very efficient
To calculate

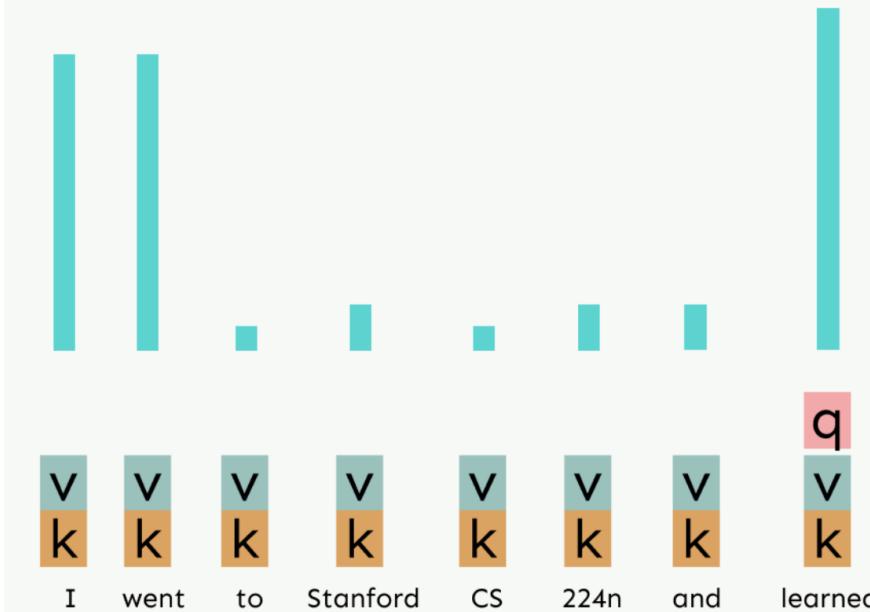


Multi-Head Attention

Attention head 1
attends to entities

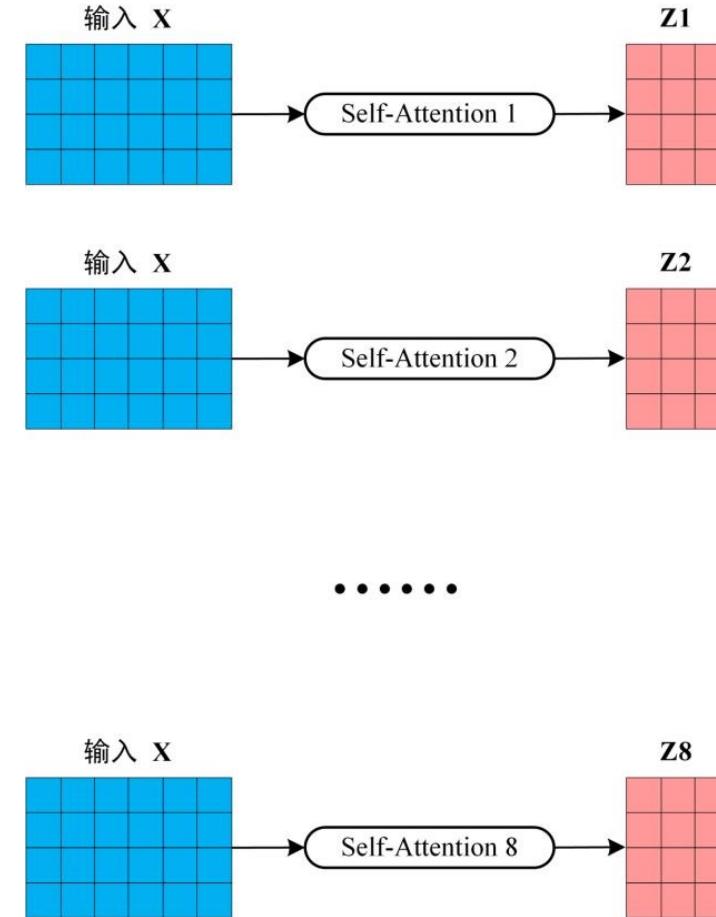
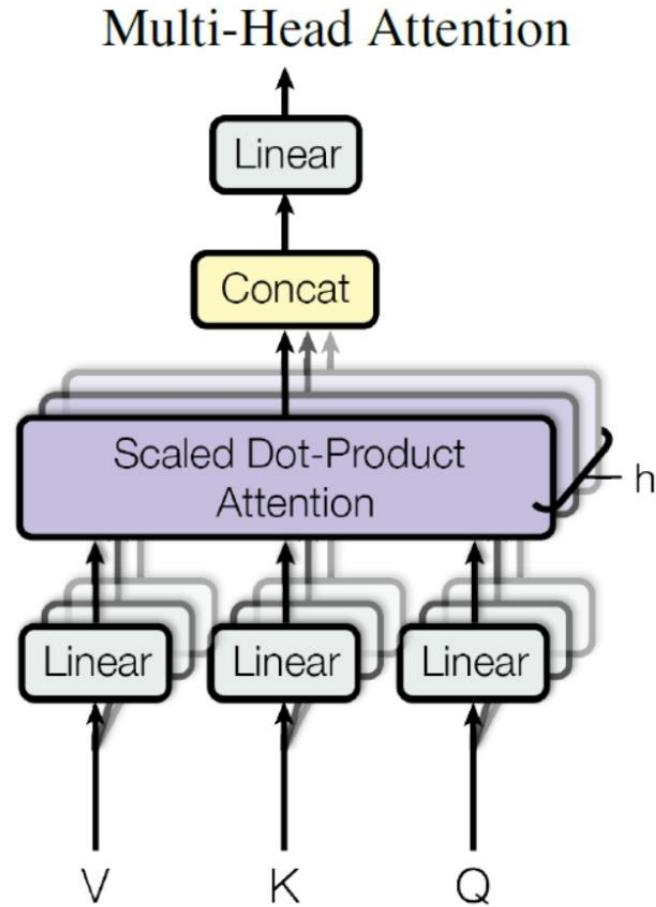


Attention head 2 attends to
syntactically relevant words

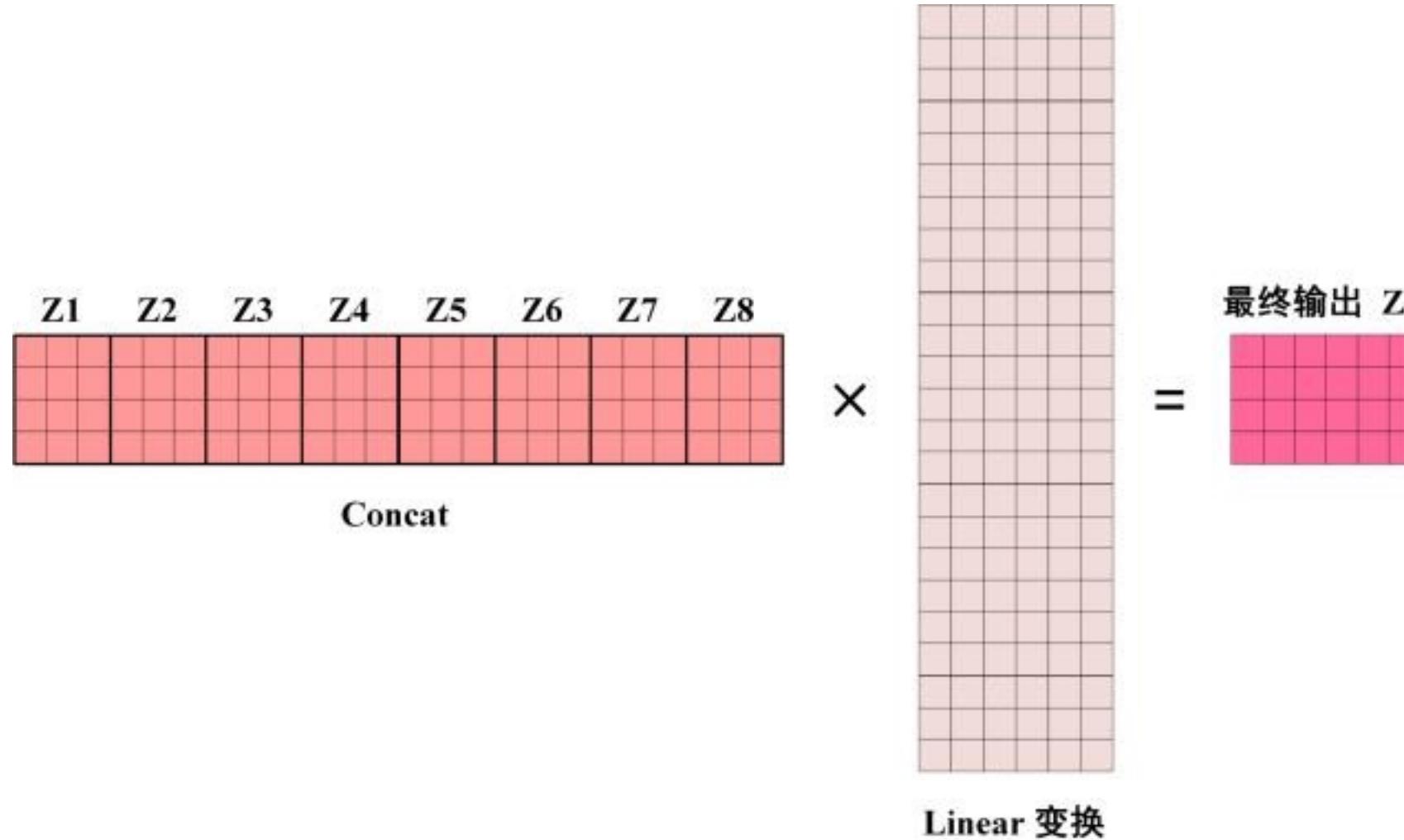


I went to Stanford CS 224n and learned

Use multi-head attention to achieve better representation



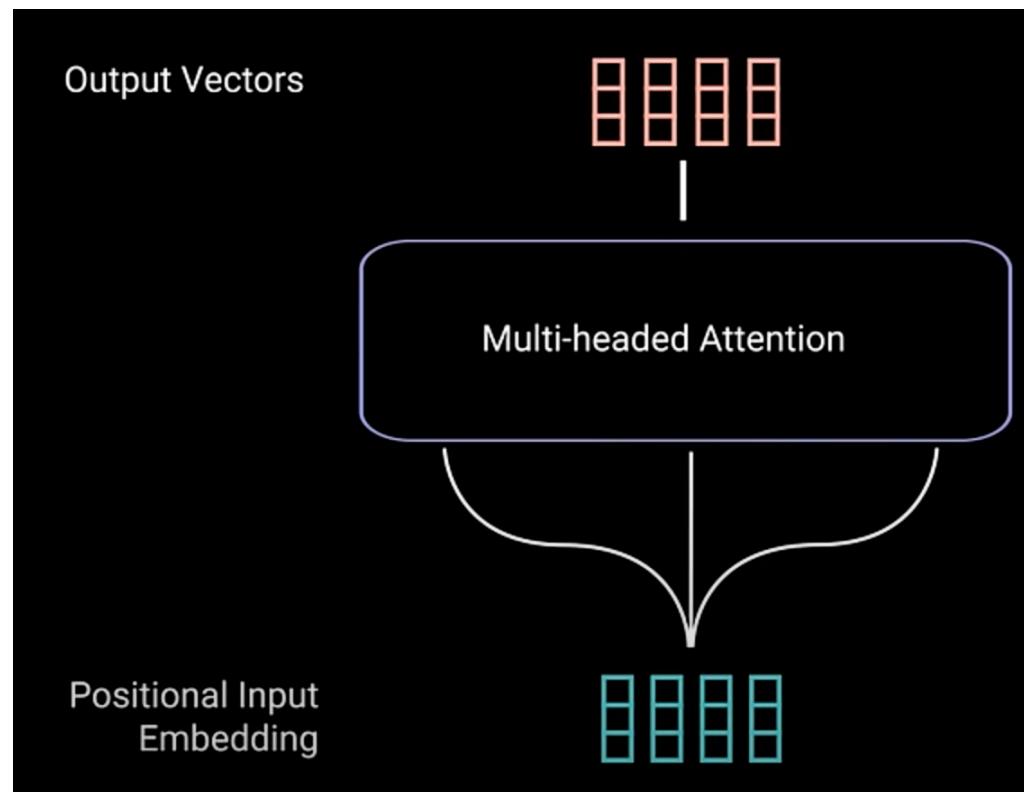
Multi-head attention: concatenation and linear transformation



Multi-head attention recap

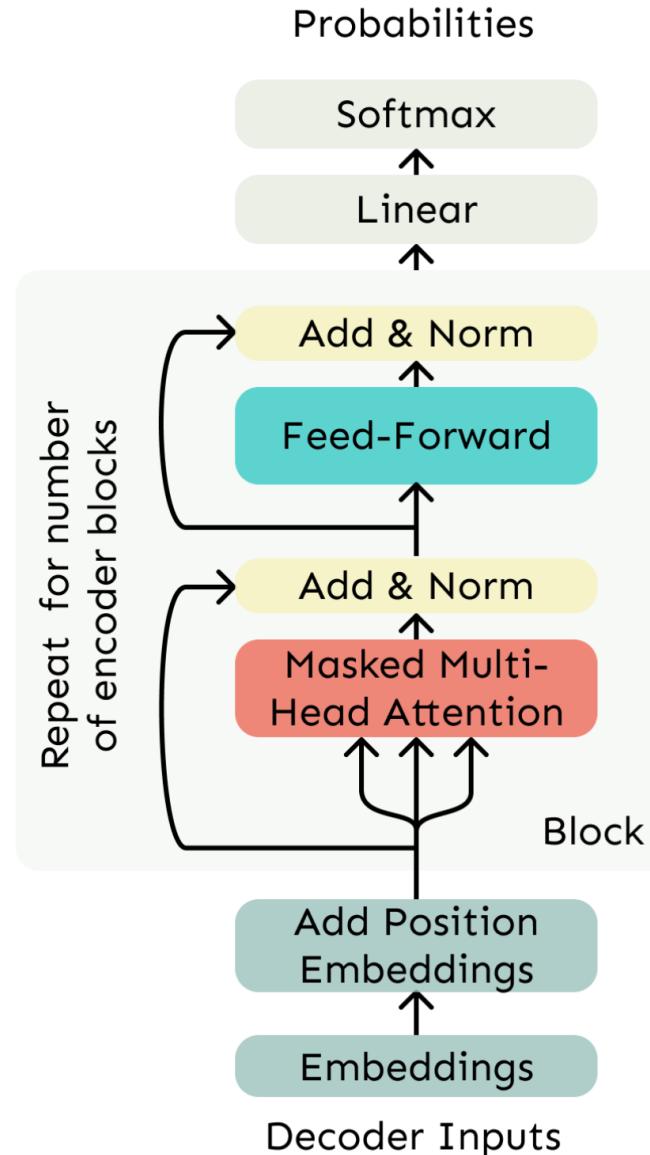
Multi-head attention is to find representations for the input words

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.

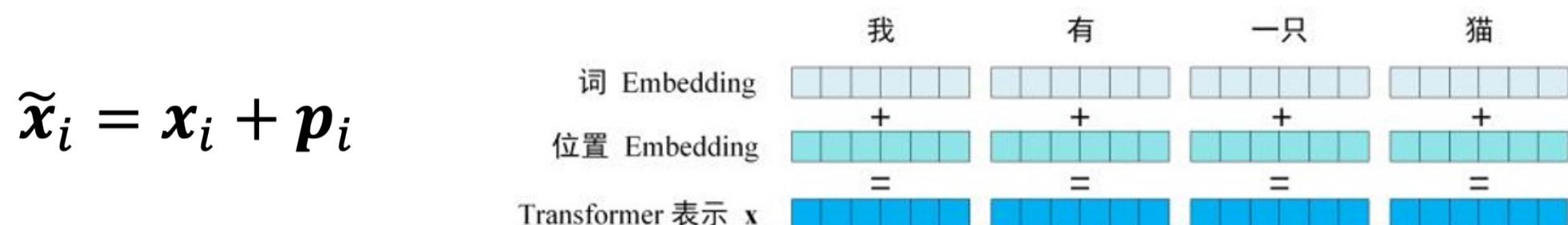


Position embedding

- 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**

$p_i \in \mathbb{R}^d$, for $i \in \{1, 2, \dots, n\}$ are position vectors

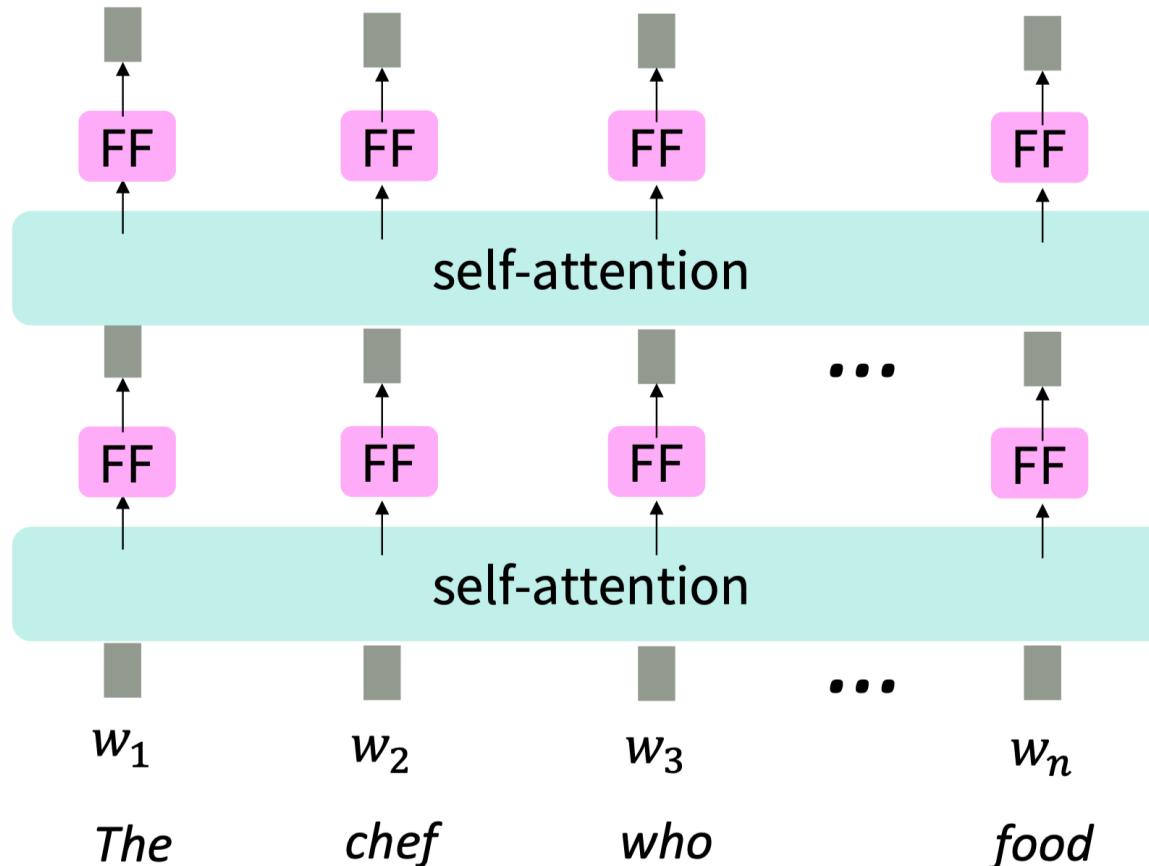
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!
- Recall that x_i is the embedding of the word at index i . The positioned embedding is:



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.

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



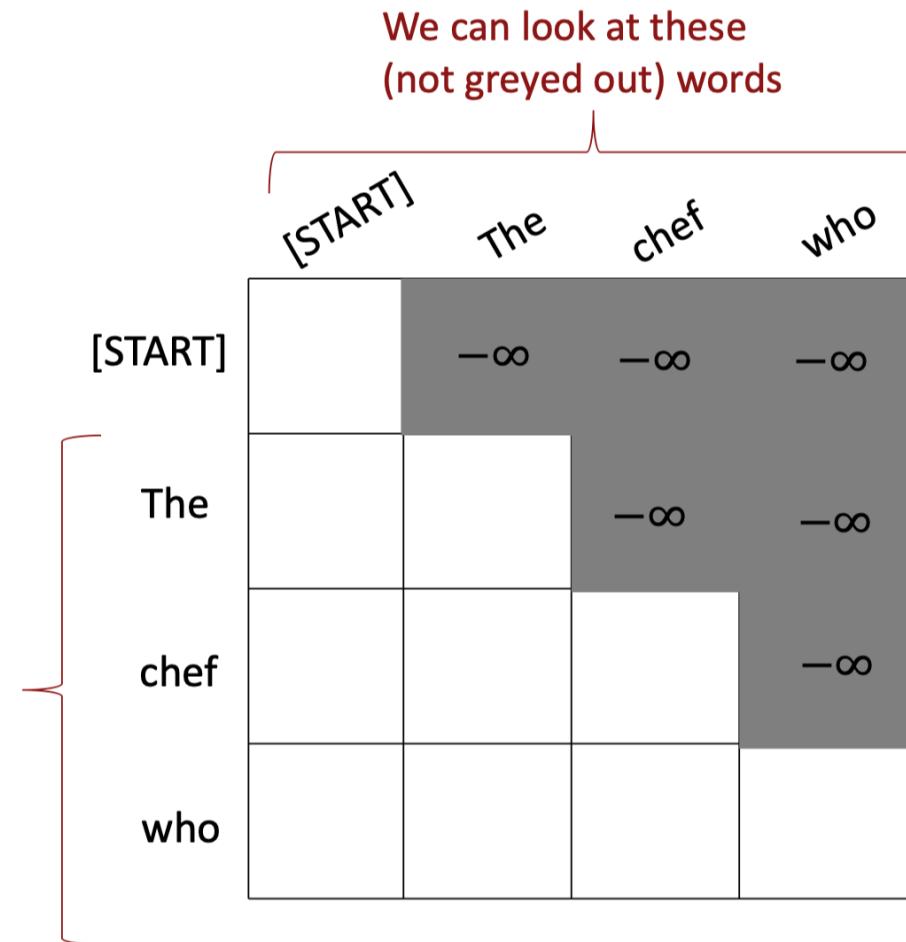
Intuition: the FF network processes the result of attention

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 $-\infty$.

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

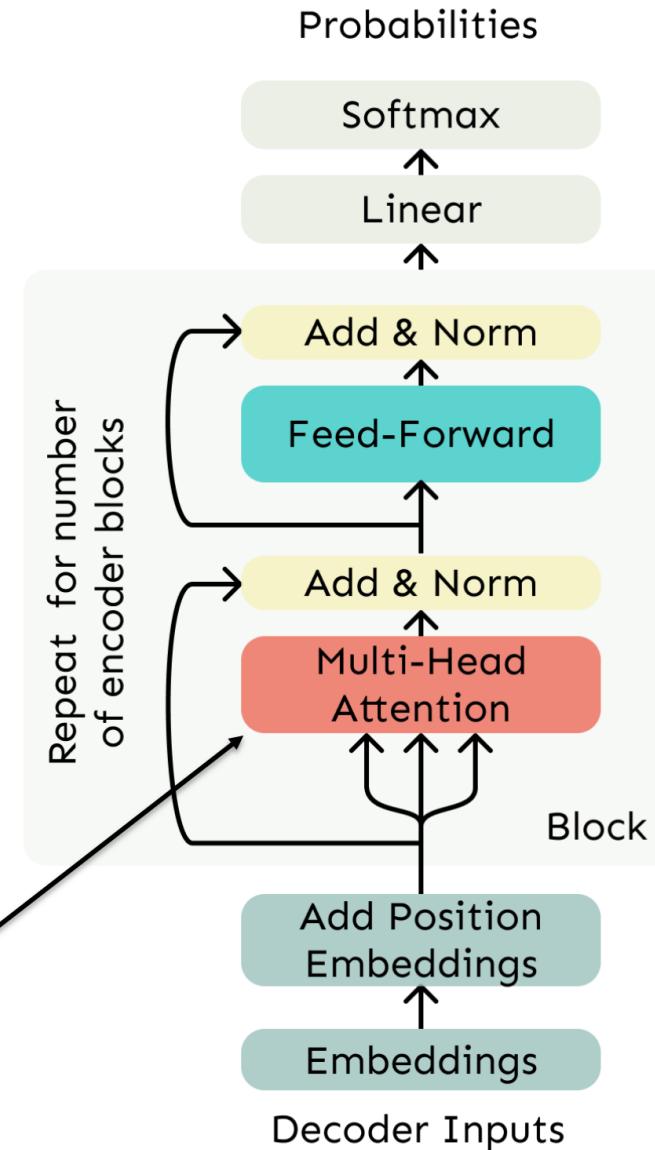
For encoding
these words



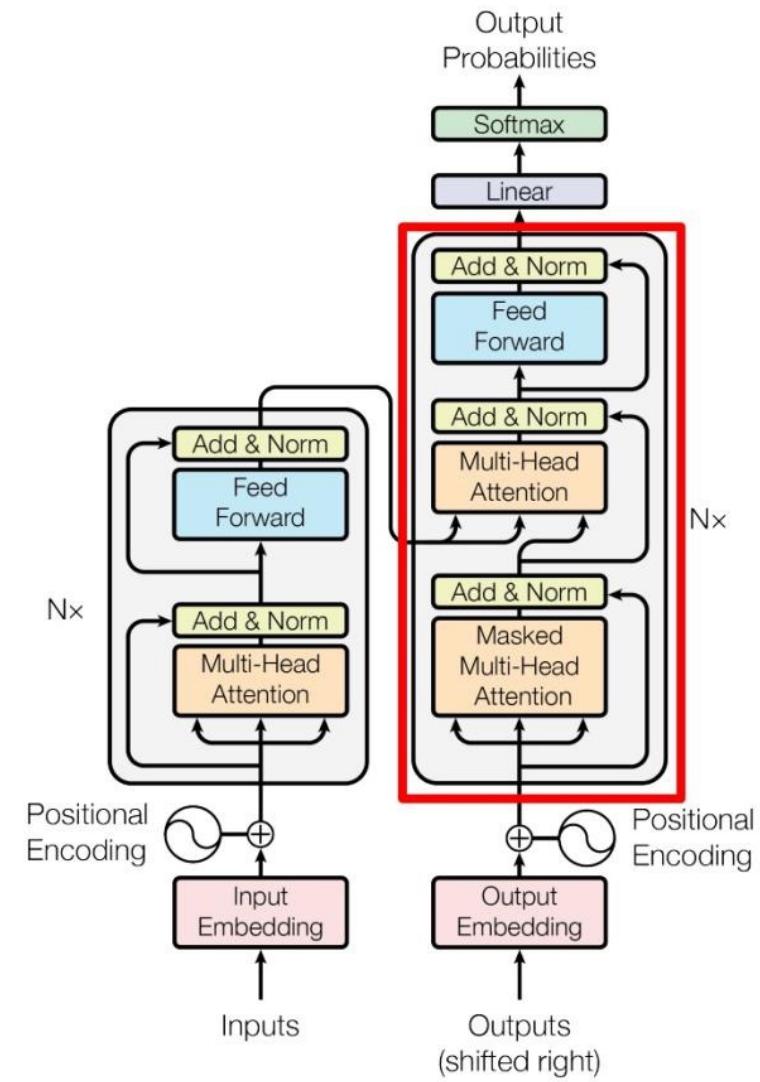
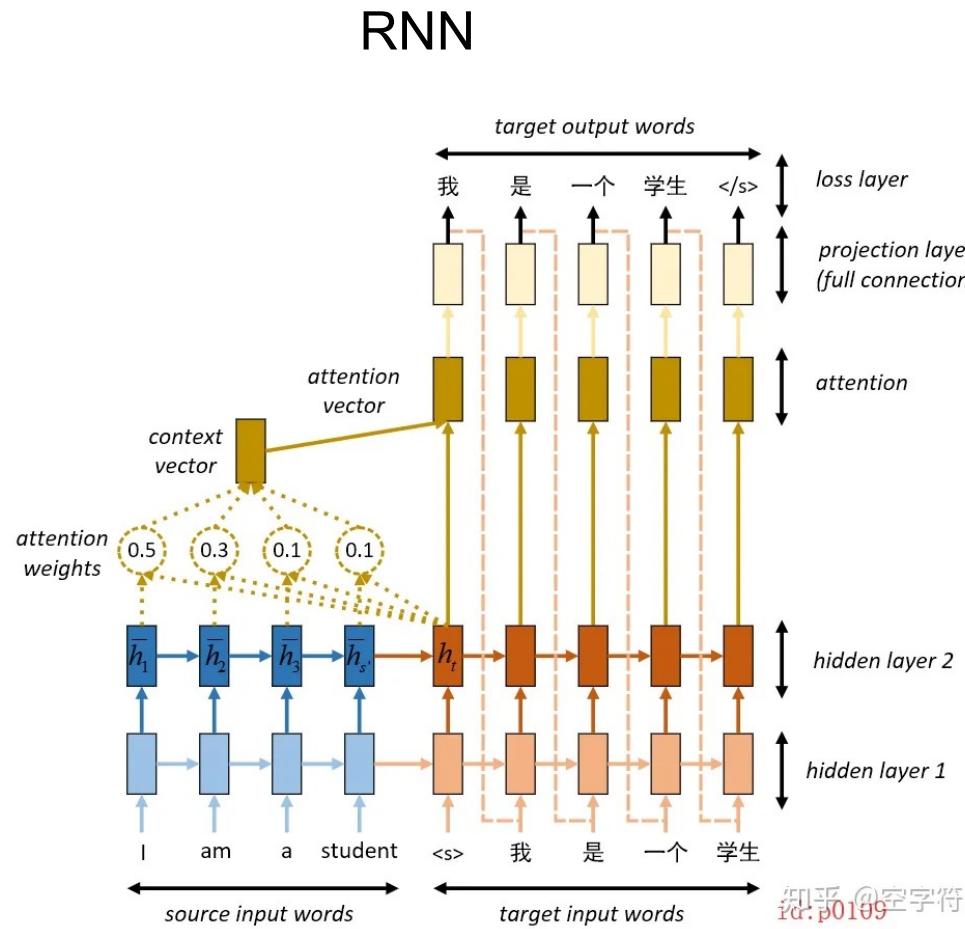
The transformer encoder

- 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.

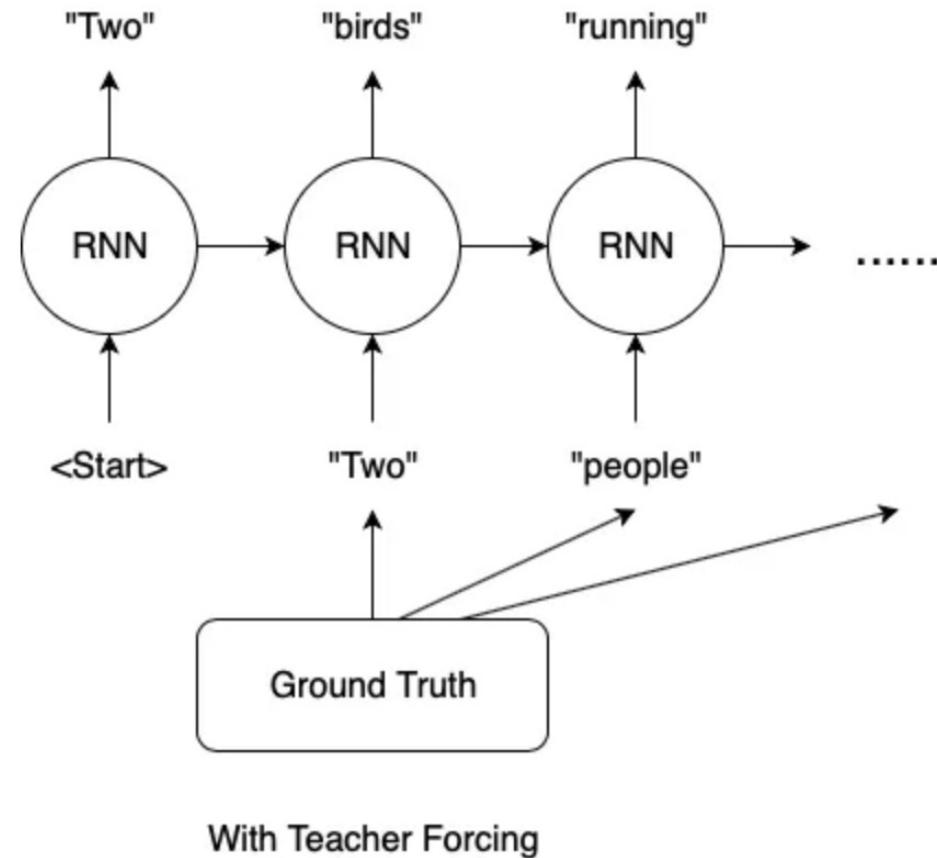
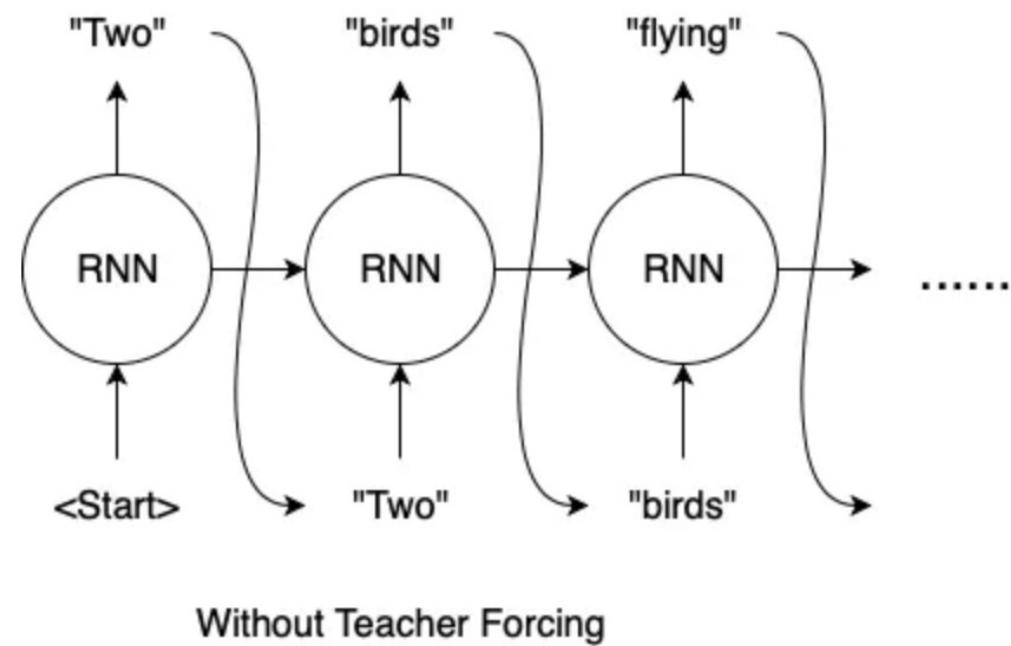
No Masking!



The transformer



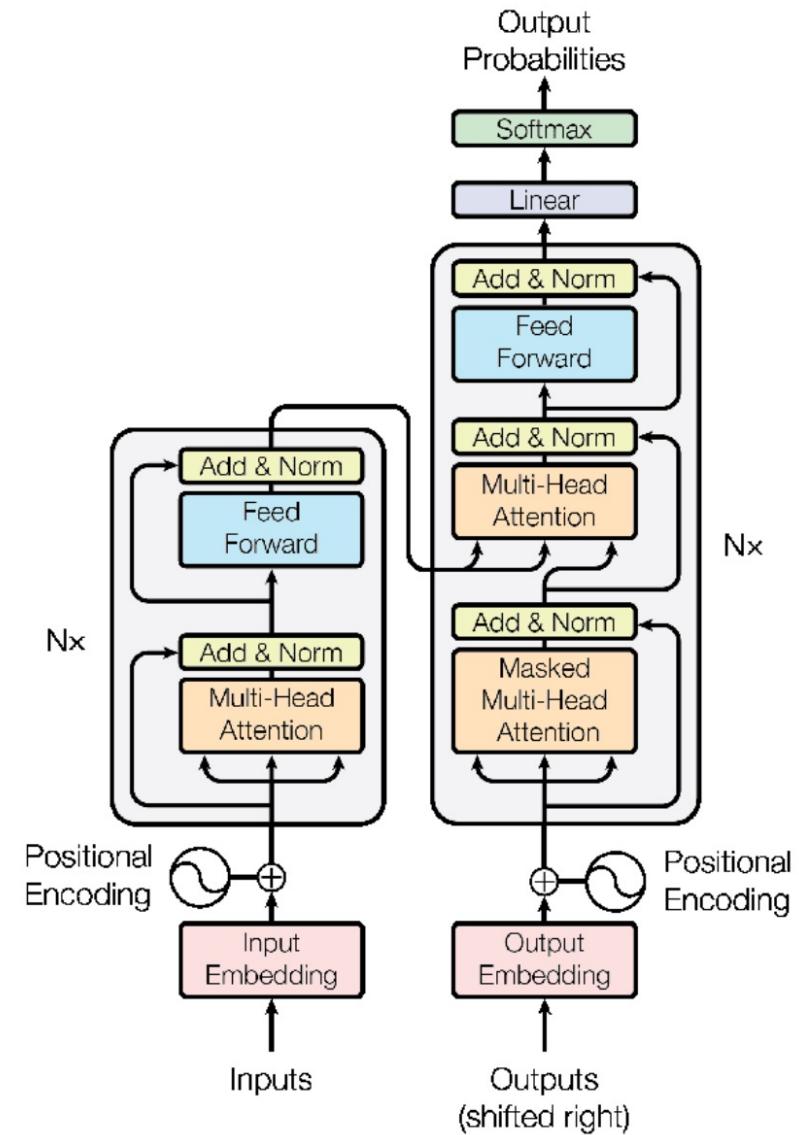
Teacher forcing



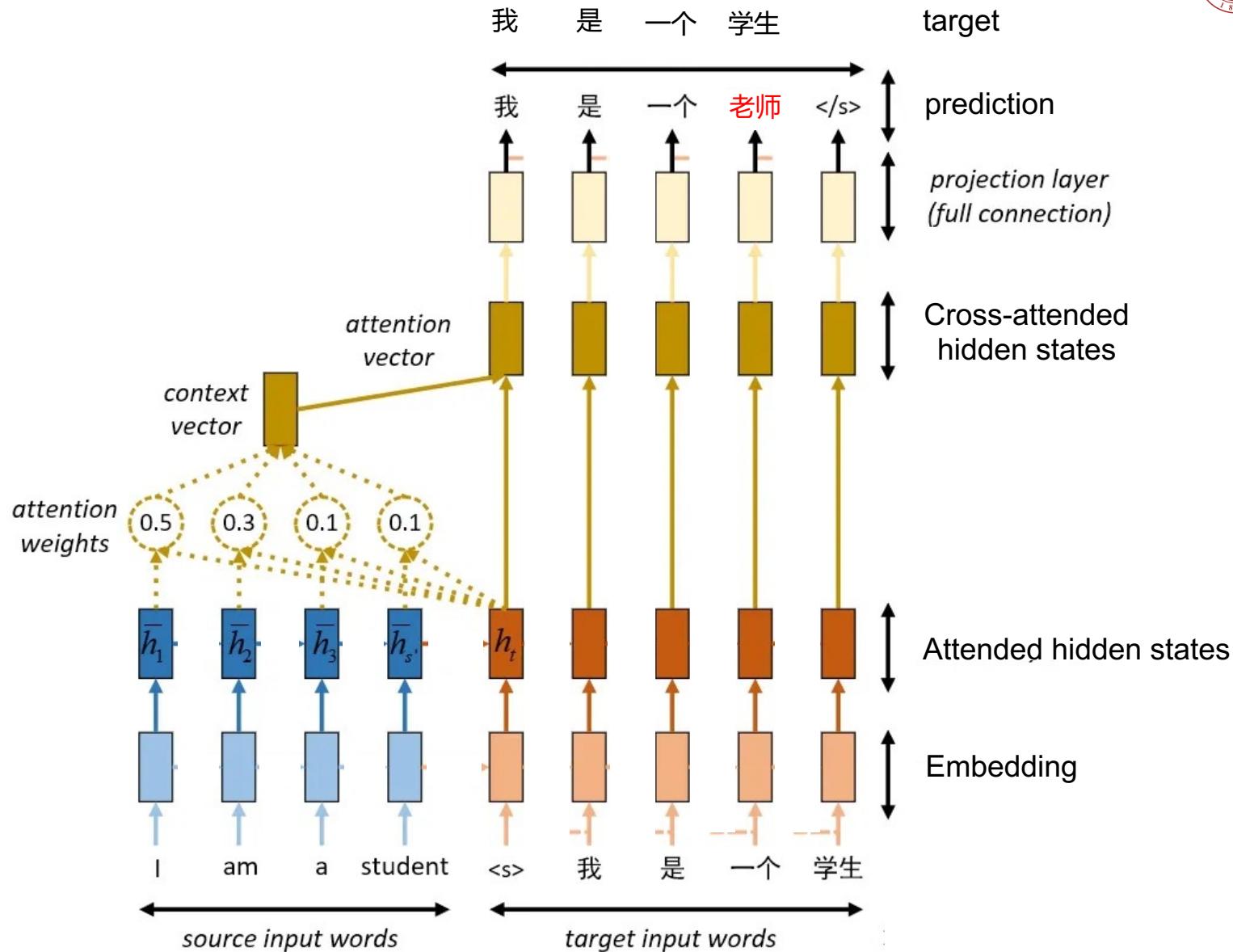
Teacher forcing

Teacher forcing is used when training transformer

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

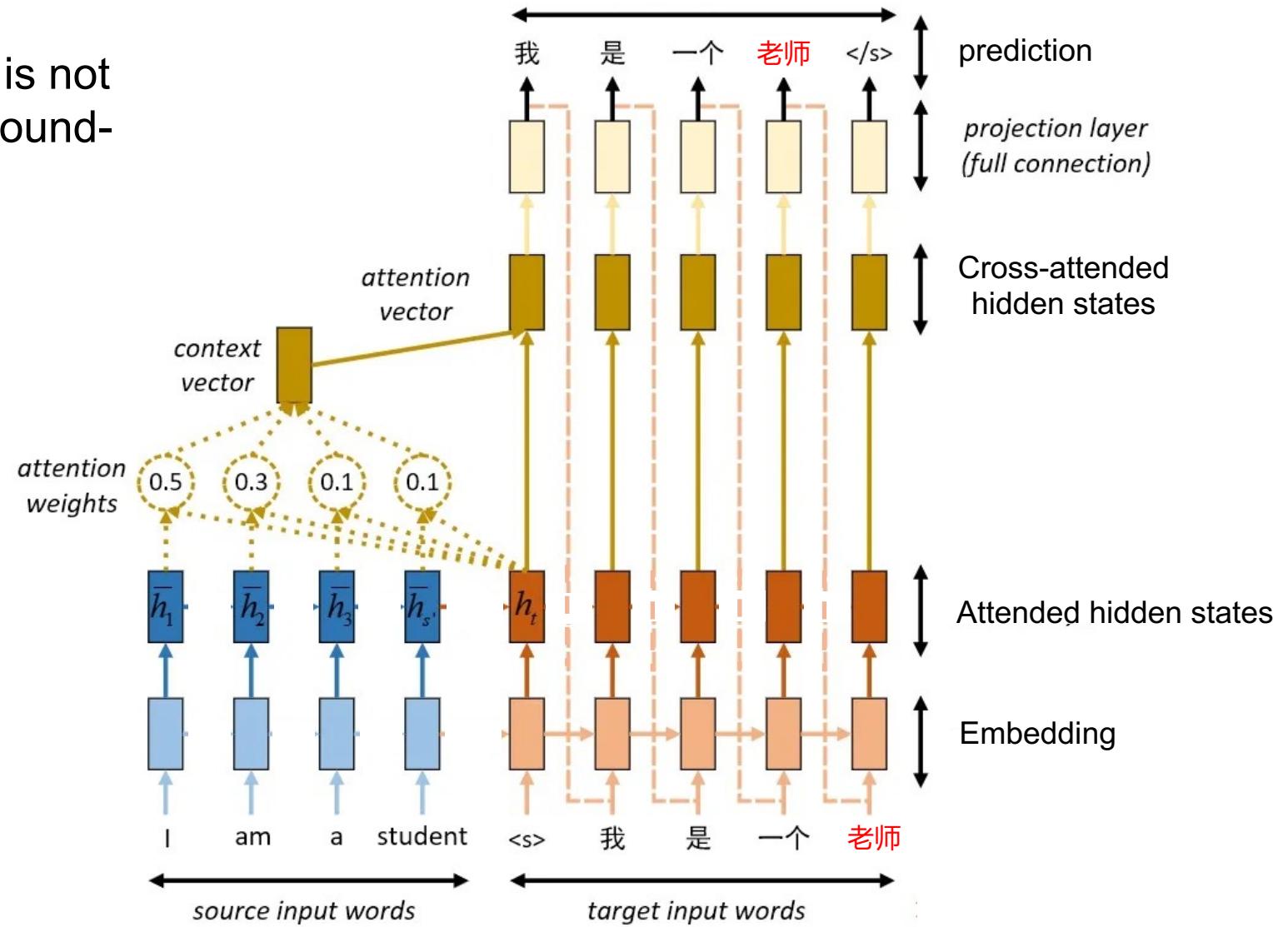


Training transformer with teacher forcing



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

- Cannot be parallel!





PART 05

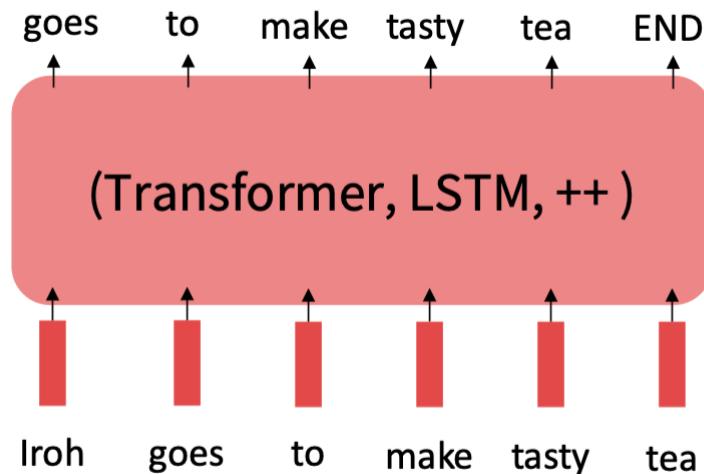
Pretrain and Finetune

The Pretraining / Finetuning Paradigm

Pretraining can improve NLP applications by serving as parameter initialization.

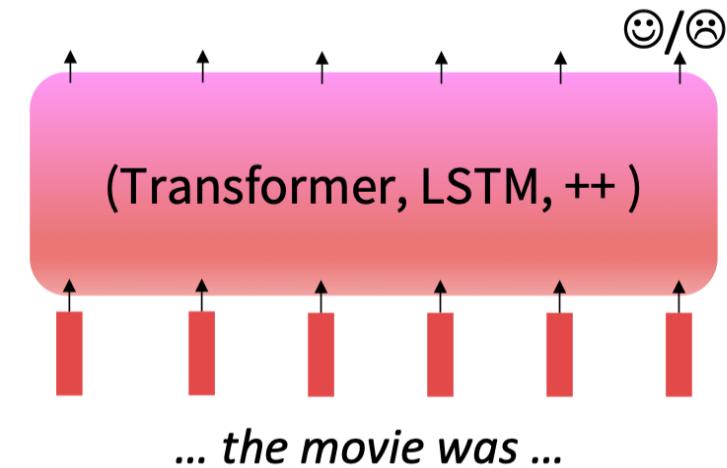
Step 1: Pretrain (on language modeling)

Lots of text; learn general things!



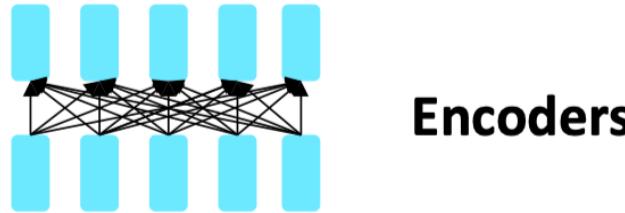
Step 2: Finetune (on your task)

Not many labels; adapt to the task!



[Figure is from Stanford CS224n]

BERT: Bidirectional Encoder Representations from Transformers

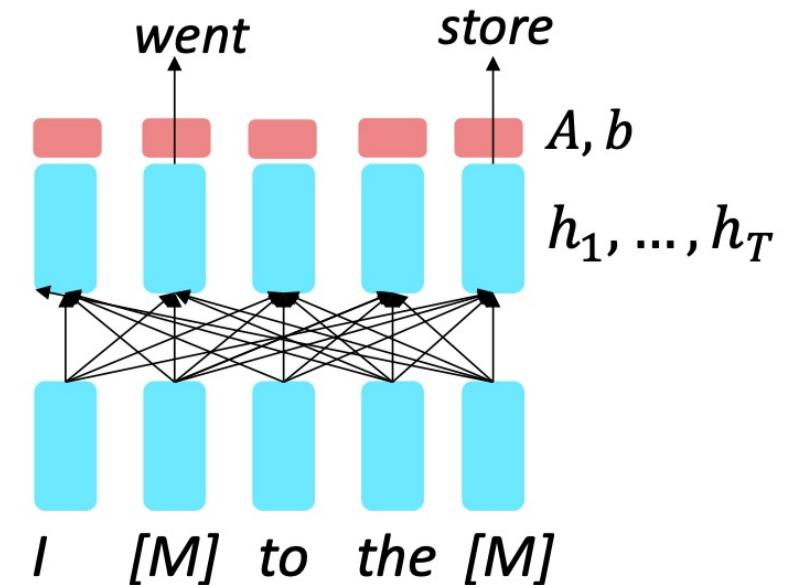


- Gets bidirectional context – can condition on future!
- How do we train them to build strong representations?

Devlin et al., 2018 proposed the “Masked LM” objective and **released the weights of a pretrained Transformer**, a model they labeled BERT.

BERT, roughly speaking, pre-trained the encoder of the transformer.

It replace some fraction of words in the input with a special [MASK] token; predict these words.



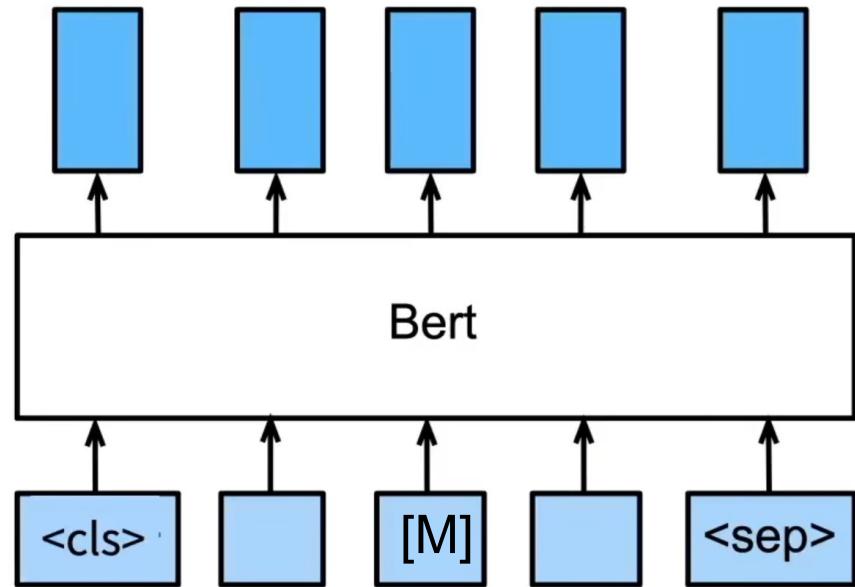
After pre-training, BERT can learn great representations for each word. We can use it as a starting point to finetune downstream tasks. In this way, BERT can perform well in that specific downstream task

To finetune BERT, we will add or change the output layer of the BERT model, and train all parameters with downstream dataset

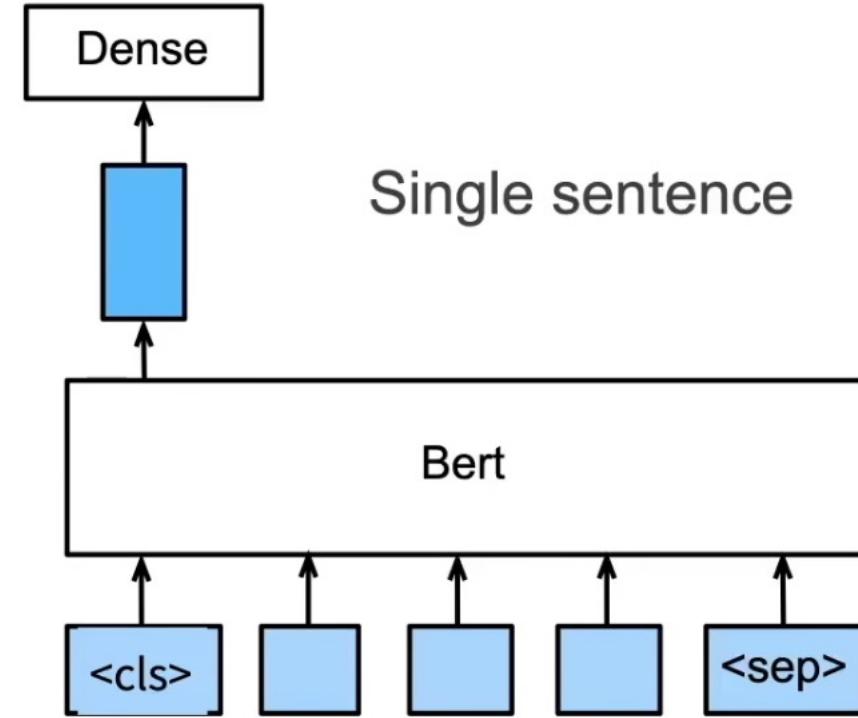
During the finetune stage, only a few downstream dataset is enough. This significantly widens the applications of natural language processing

Pretraining and finetuning become **the standard practices** in most AI fields (NLP, CV, etc.)

Finetune BERT to sentence classification

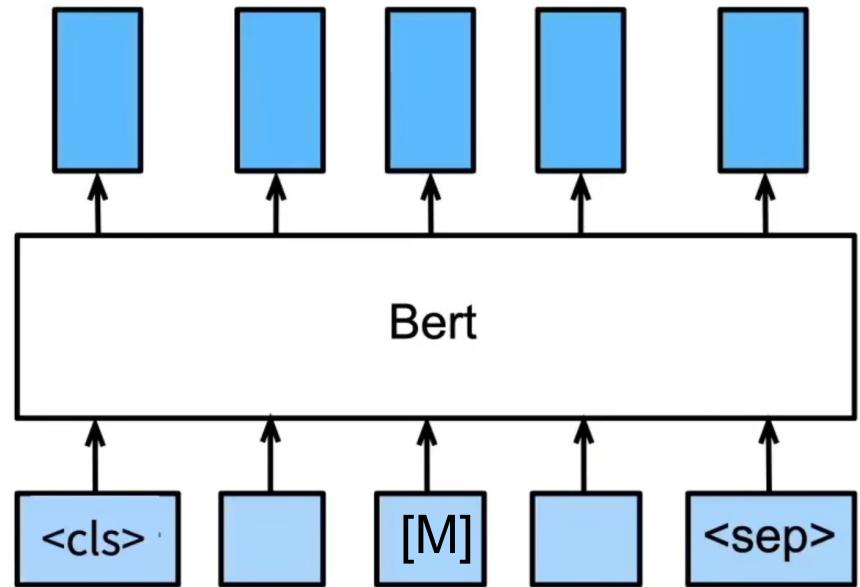


Pre-train

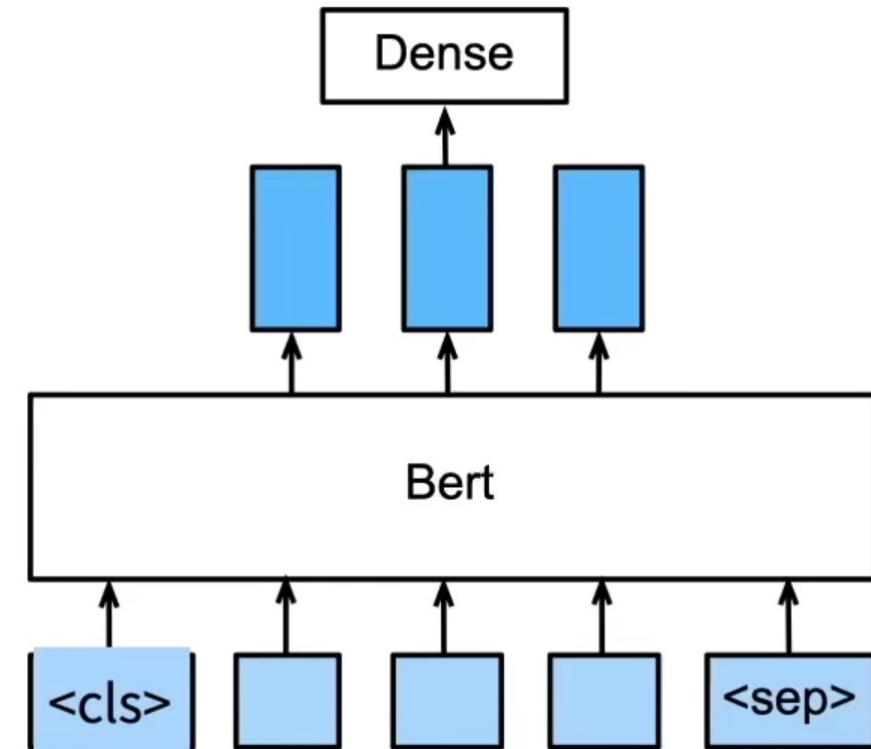


Finetune

Finetune BERT to entity recognition



Pre-train



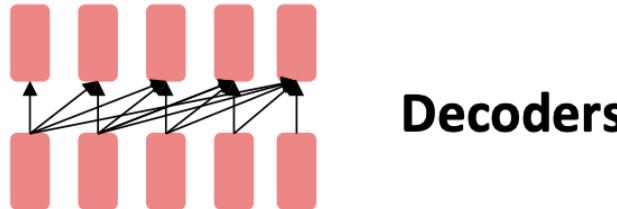
Finetune

BERT limitations

Pretrain encoders such as BERT is great. It particularly suits in information extraction.

However, it is not good at text generation which cannot look into the future

If your task involves generating sequences, consider using a pretrained decoder; BERT and other pretrained encoders don't naturally lead to nice autoregressive (1-word-at-a-time) generation methods.

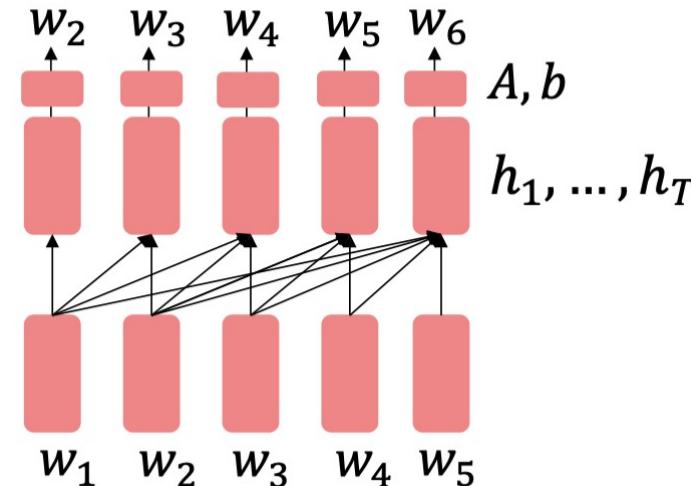


- Language models! What we've seen so far.
- Nice to generate from; can't condition on future words
- All the biggest pretrained models are Decoders.

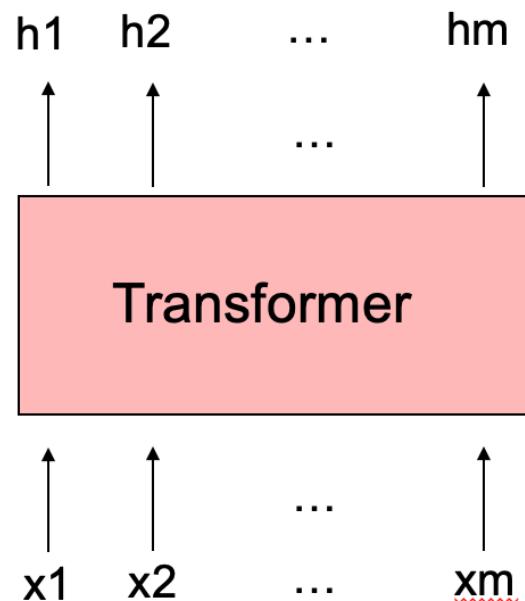
Generative Pretrained Transformer (GPT) [Radford et al., 2018] was a big success in pretraining a decoder!

GPT, roughly speaking, pre-trained the decoder of the transformer.

GPT typically stands for “Generative Pretrained Transformer”



Given a sequence of downstream tokens $\{x_1, x_2, \dots, x_m\}$, and its label y



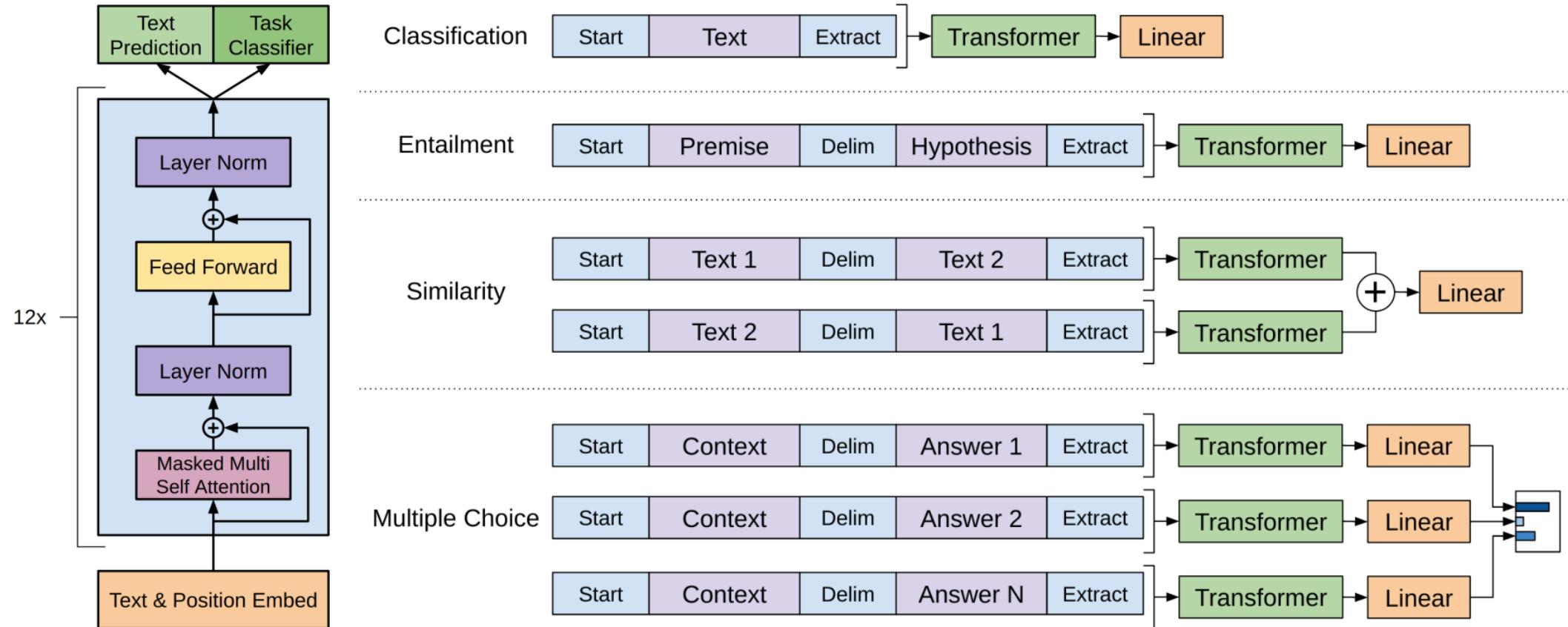
$$\text{Transformer} + \text{softmax}(h_\ell^m W_y) = L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda * L_1(\mathcal{C})$$

$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \dots, u_{i-1}; \Theta)$$

$$P(y|x^1, \dots, x^m) = \text{softmax}(h_l^m W_y).$$

$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P(y|x^1, \dots, x^m).$$

It is very natural to perform fine-tuning with GPT



GPT-2 is a zero-shot learner



No need of any downstream dataset. No need to fine-tune. **Prompt** is enough!



English reference

This re-release, titled The Next Day Extra, was presented in the form of three disks: the original album, unpublished studio sessions and remixes, plus a DVD containing the four clips that have already been unveiled.

(translate to French)



GPT-2 French translation

Les nouvelles re-releases, tout en premier disc, nécessaire de l'album, un studio session et remixes, plus une DVD de l'écran de quelques clips qui ont été déjà échappés.

In contrast, BERT and its variants do not show similar capabilities

GPT-3 gets extremely large. GPT-2 has 1.5B parameters, while GPT-3 has 175B parameters

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

GPT-2 uses zero-shots, while GPT-3 uses few-shots

Zero-shot

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

- 1 Translate English to French: ← *task description*
- 2 cheese => ← *prompt*

One-shot

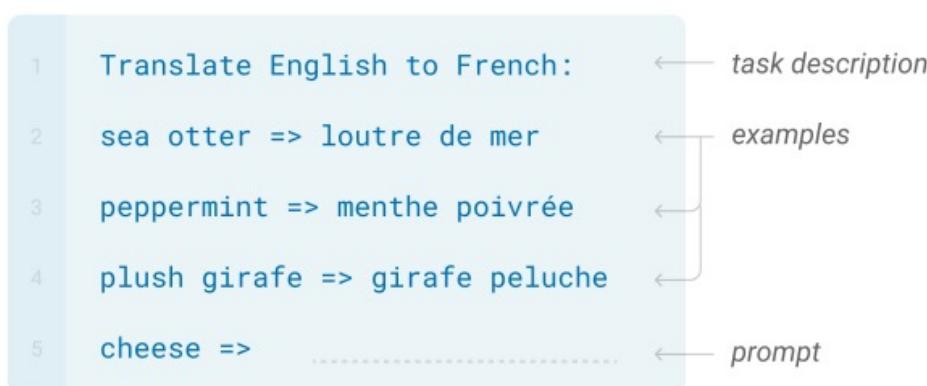
In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

- 1 Translate English to French: ← *task description*
- 2 sea otter => loutre de mer ← *example*
- 3 cheese => ← *prompt*

GPT-2 uses zero-shots but GPT-3 uses few-shots

Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

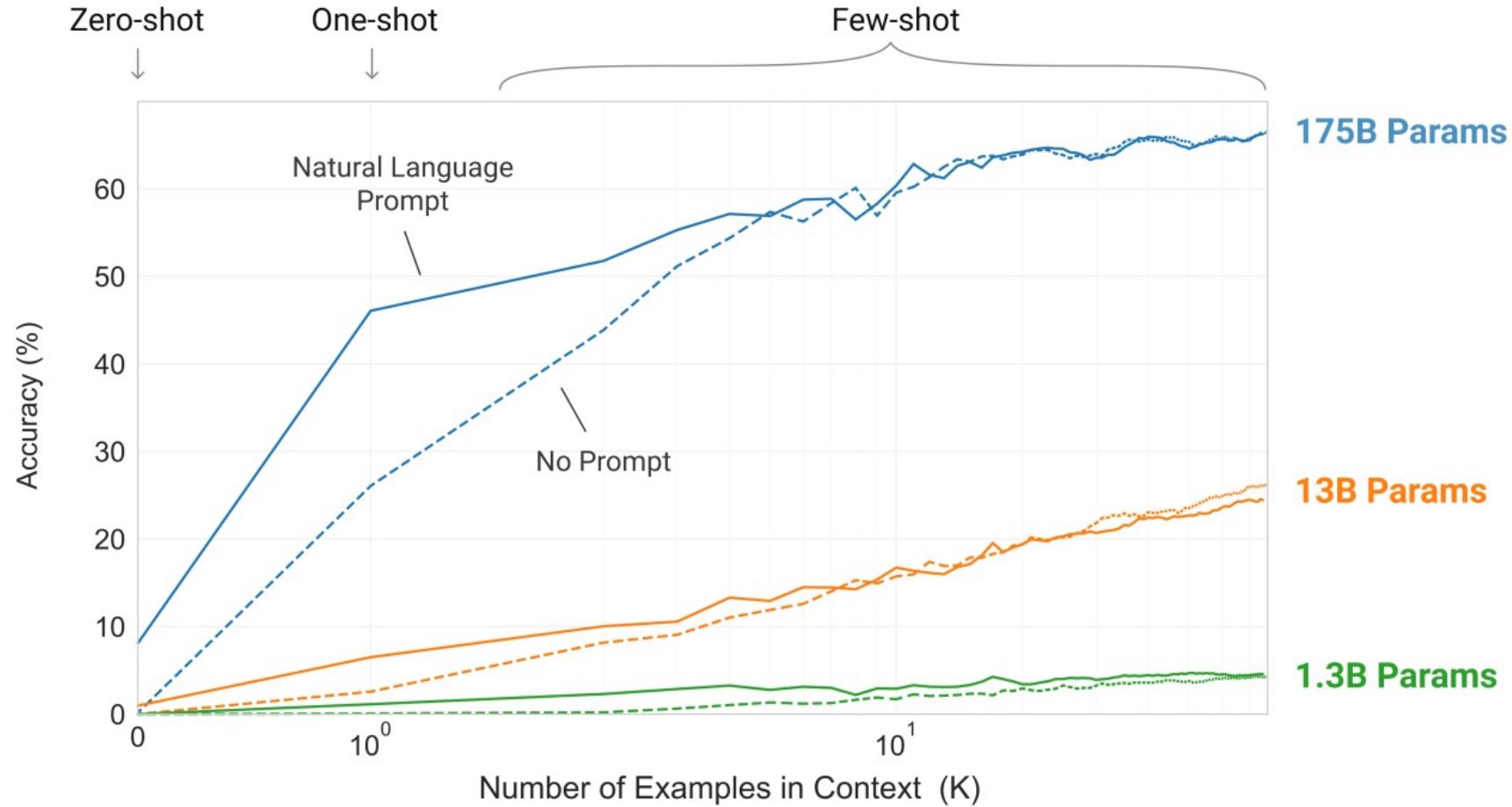


Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.



Large model and few-shots are very useful

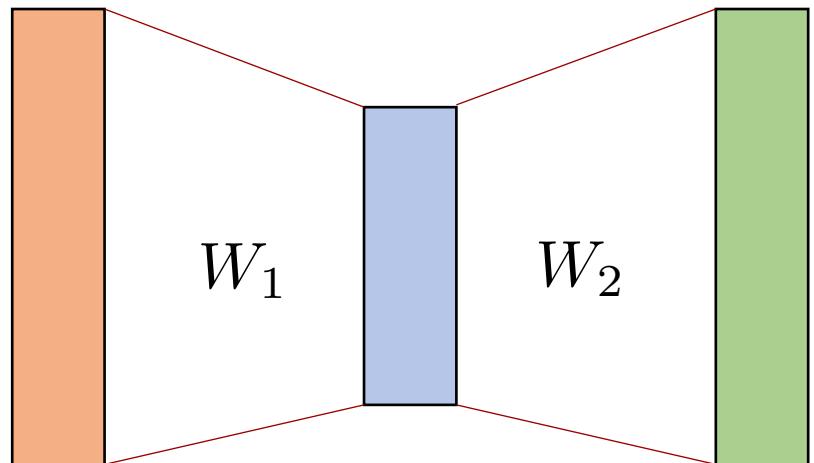




PART 06

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

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

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

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

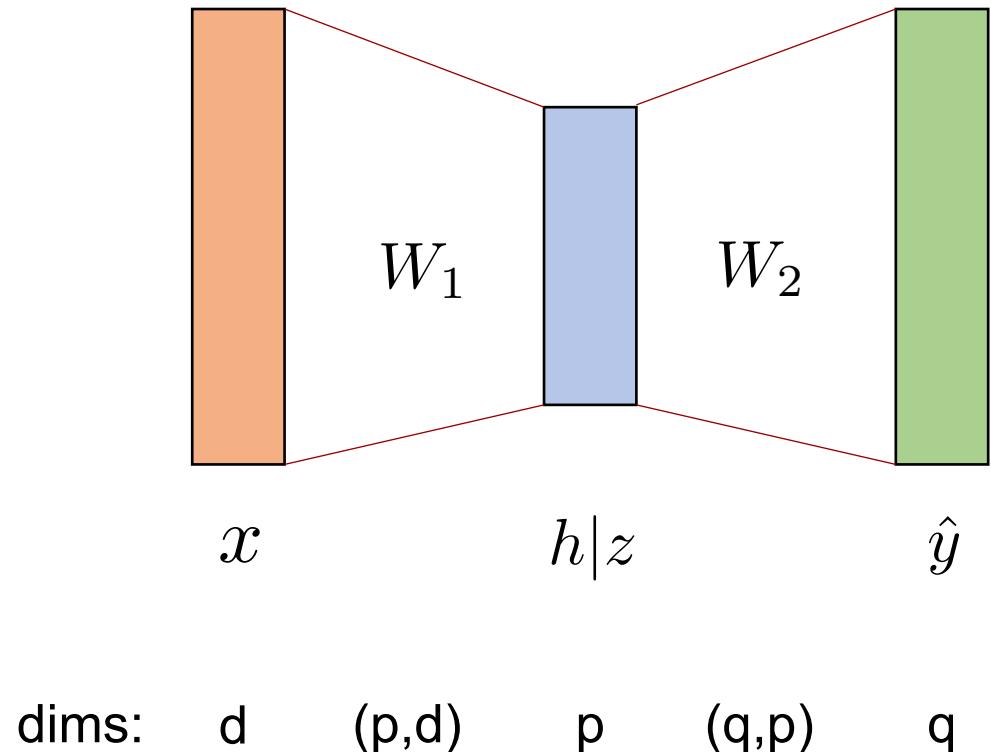
$$\frac{\partial f}{\partial \hat{y}} = \nabla L(\hat{y})$$

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



↓ ↓ ↑

$h_b = W_1 x_b$	$\frac{\partial f}{\partial W_1} = \frac{1}{B} \sum_{b=1}^B \frac{\partial f}{\partial h_b} x_b^T,$
$z_b = \sigma(h_b)$	$\frac{\partial f}{\partial h_b} = \frac{\partial f}{\partial z_b} \odot \nabla \sigma(h_b)$
$\hat{y}_b = W_2 z_b$	$\frac{\partial f}{\partial W_2} = \frac{1}{B} \sum_{b=1}^B \frac{\partial L}{\partial \hat{y}_b} z_b^T, \quad \frac{\partial f}{\partial z_b} = W_2^T \frac{\partial f}{\partial \hat{y}_b}$
$f = \frac{1}{B} \sum_{b=1}^B L(\hat{y}_b)$	$\frac{\partial f}{\partial \hat{y}_b} = \frac{\partial L}{\partial \hat{y}_b}$
Forward	Backward

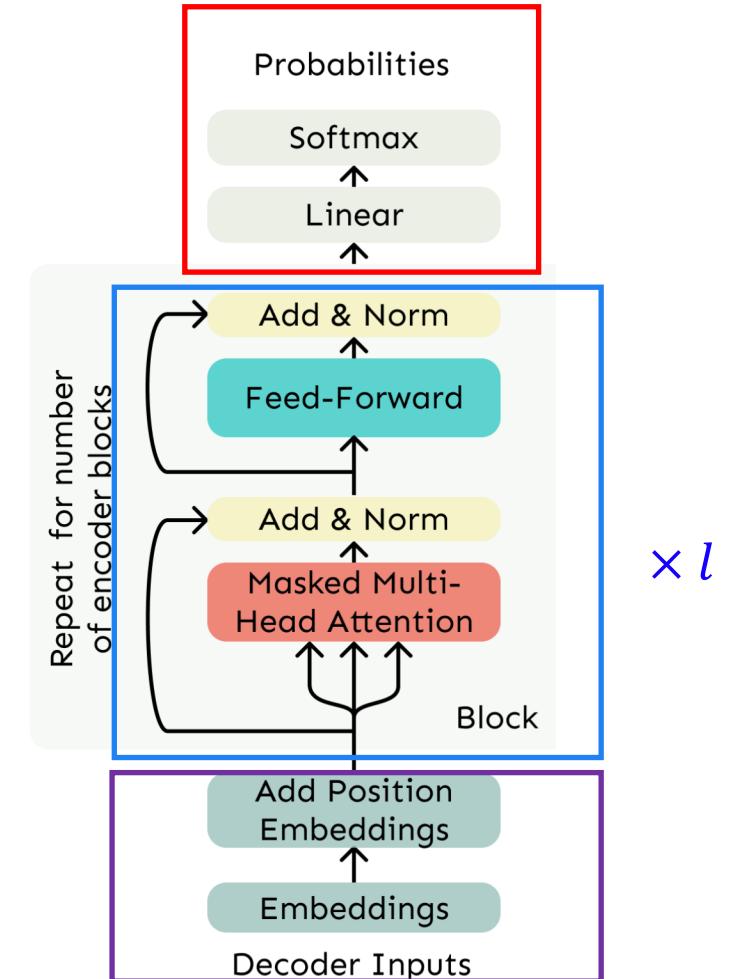
Store $\{\mathbf{h}_b, \mathbf{z}_b, \hat{\mathbf{y}}_b\}_{b=1}^B$ **Store $\nabla_{W_1} f(W_1)$ and $\nabla_{W_2} f(W_2)$**

Total parameters

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

Total parameters:

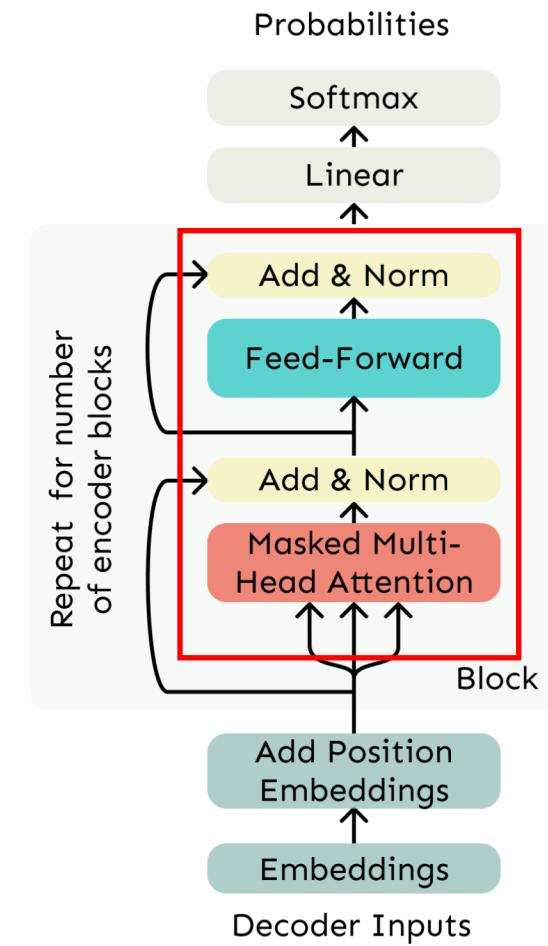
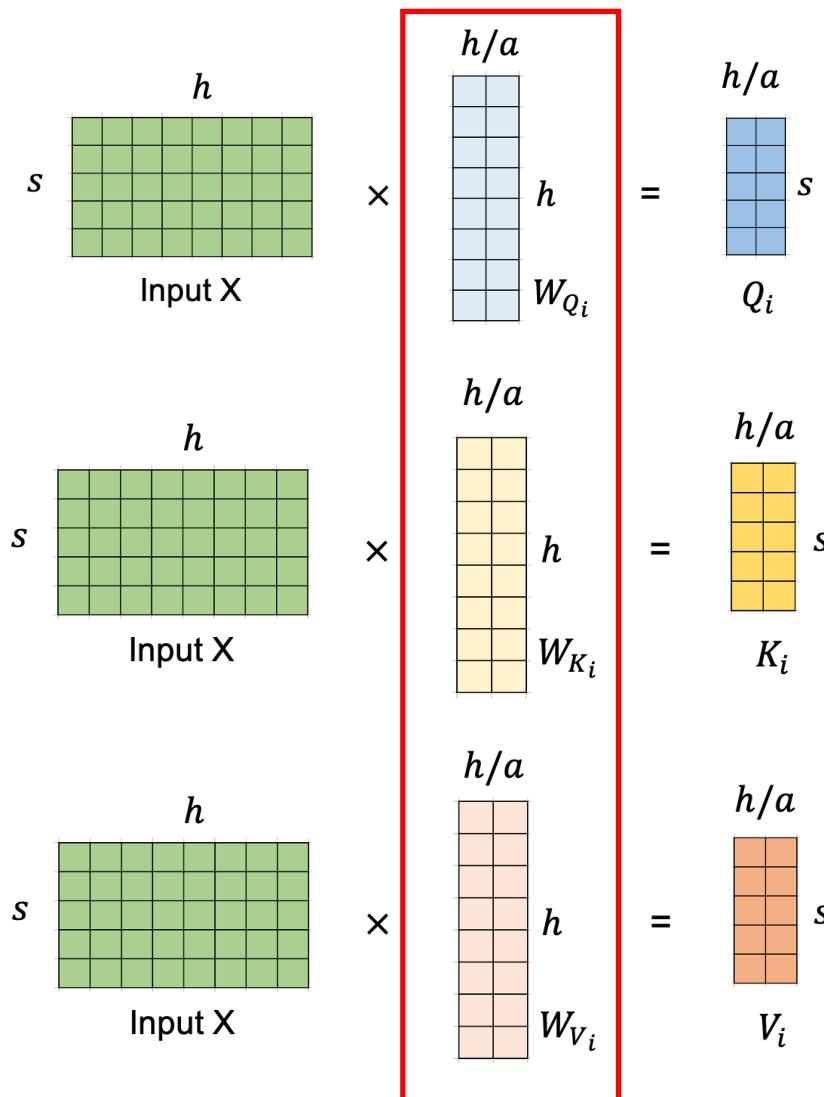
$$12\ell h^2 + 2vh$$



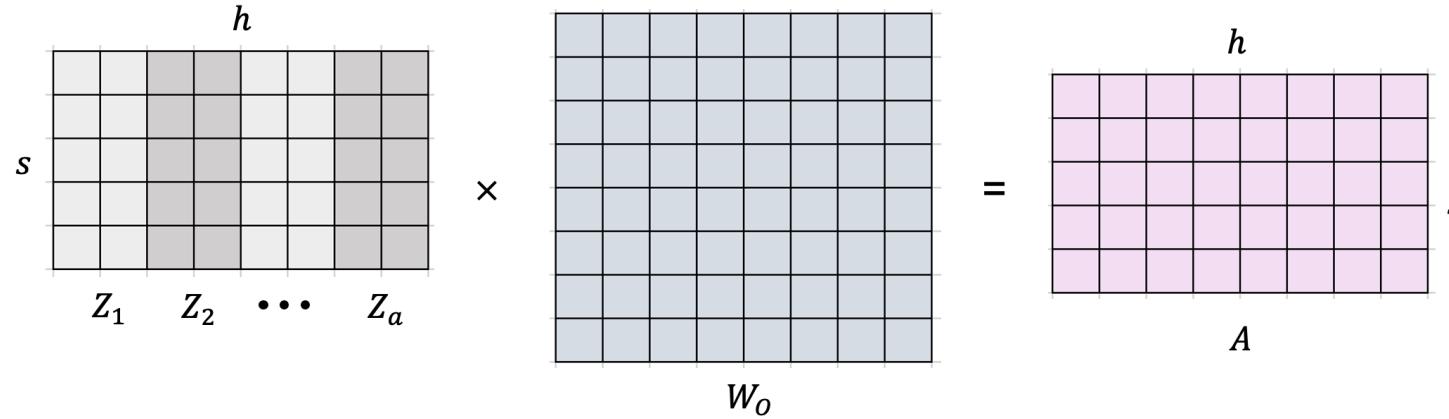
Multi-head attentions

- We need to store W_Q , W_K and W_V

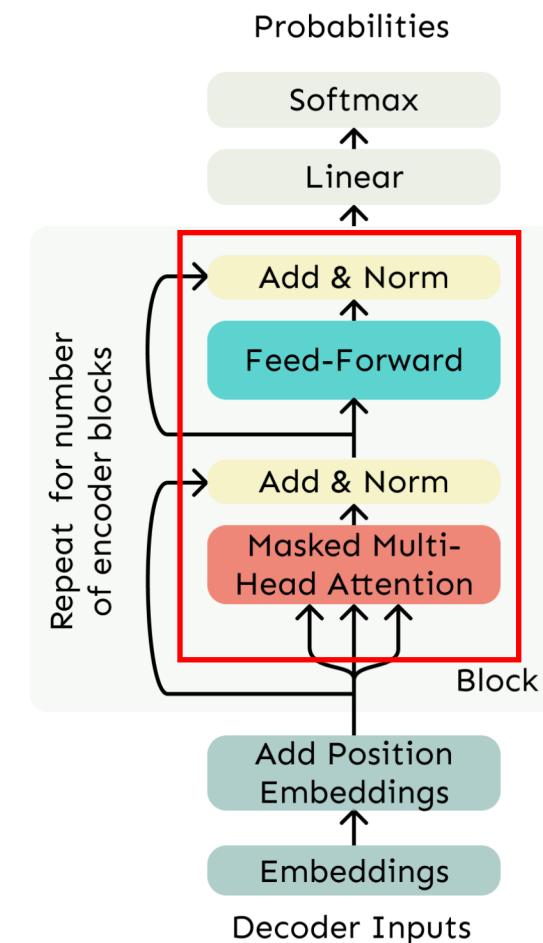
$$3(h^2/a) \times a = 3h^2$$



Multi-head attentions



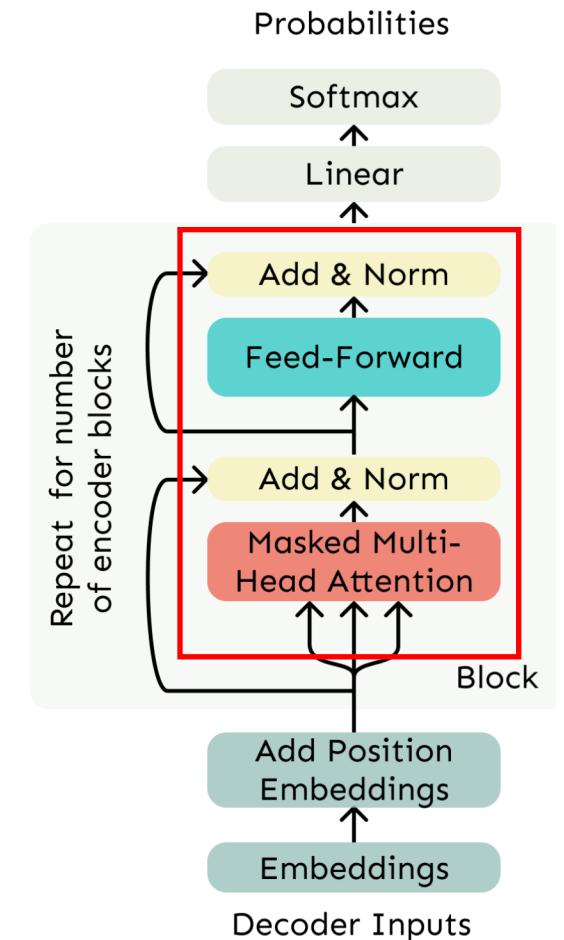
- We need to store W_O : h^2



Feed-forward layers

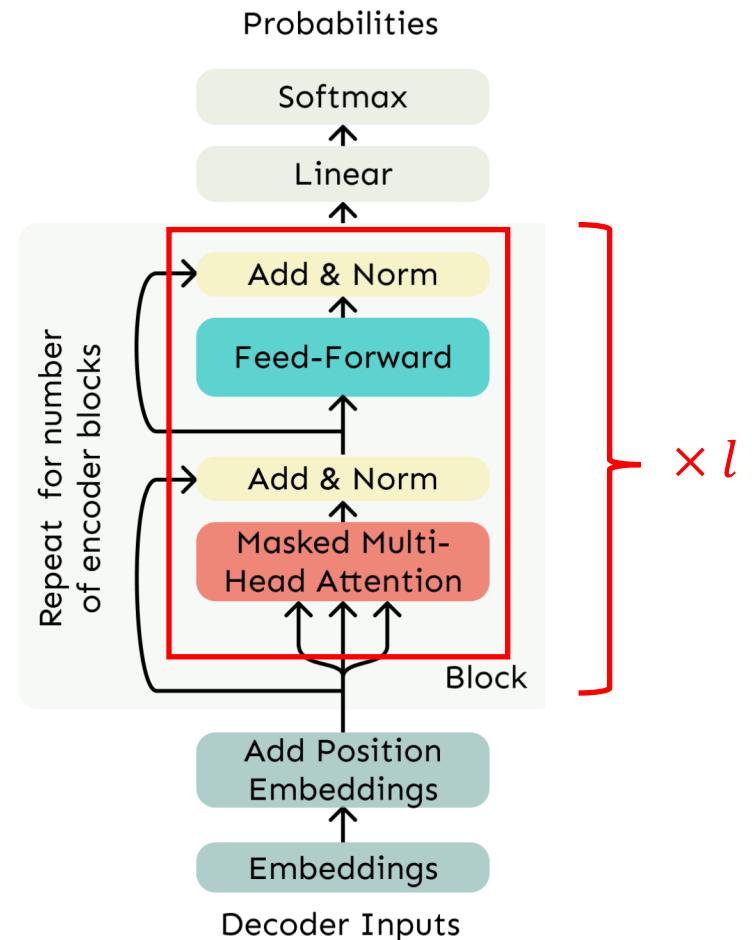
$$X' = \text{ReLU}(A \cdot W_1 + b_1) \cdot W_2 + b_2$$

- 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 = 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

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

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

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

- 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



Try your best and do not leave blanks!

Good Luck!