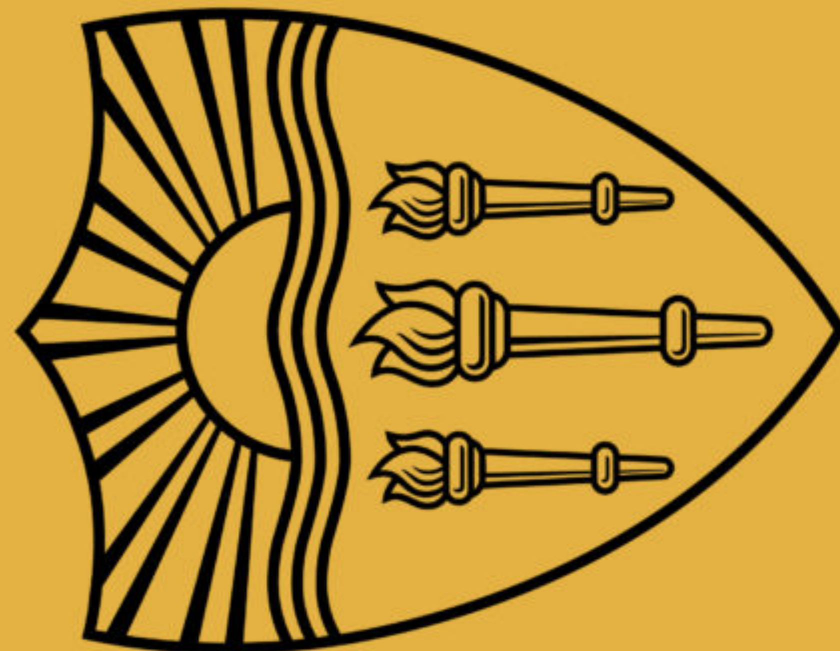
USC



Lecture 12: Transformers: Building Blocks

Instructor: Swabha Swayamdipta

USC CSCI 499 LMs in NLP

Mar 4, Spring 2024

Slides adapted from Dan Jurafsky, Chris Manning, John Hewitt, Anna Goldie

Logistics / Announcements

- Today:
  - Project Progress Report Deadline
  - Graded Quiz 3 sheets will be distributed
  - HW3 will be out
  - Piazza post for final project presentations

- This week:
  - TA lecture on PyTorch
  - Quiz 4 on Wednesday
  - Graded Project Progress Reports will be out

Lecture Outline

- Quiz 3 Answers
- Recap: Transformers as Self-Attention Networks
- Transformers: Multiheaded Attention
- Transformers: Positional Embeddings
- Putting it all together: Transformer Blocks
- Transformers as Encoders, Decoders and Encoder-Decoders

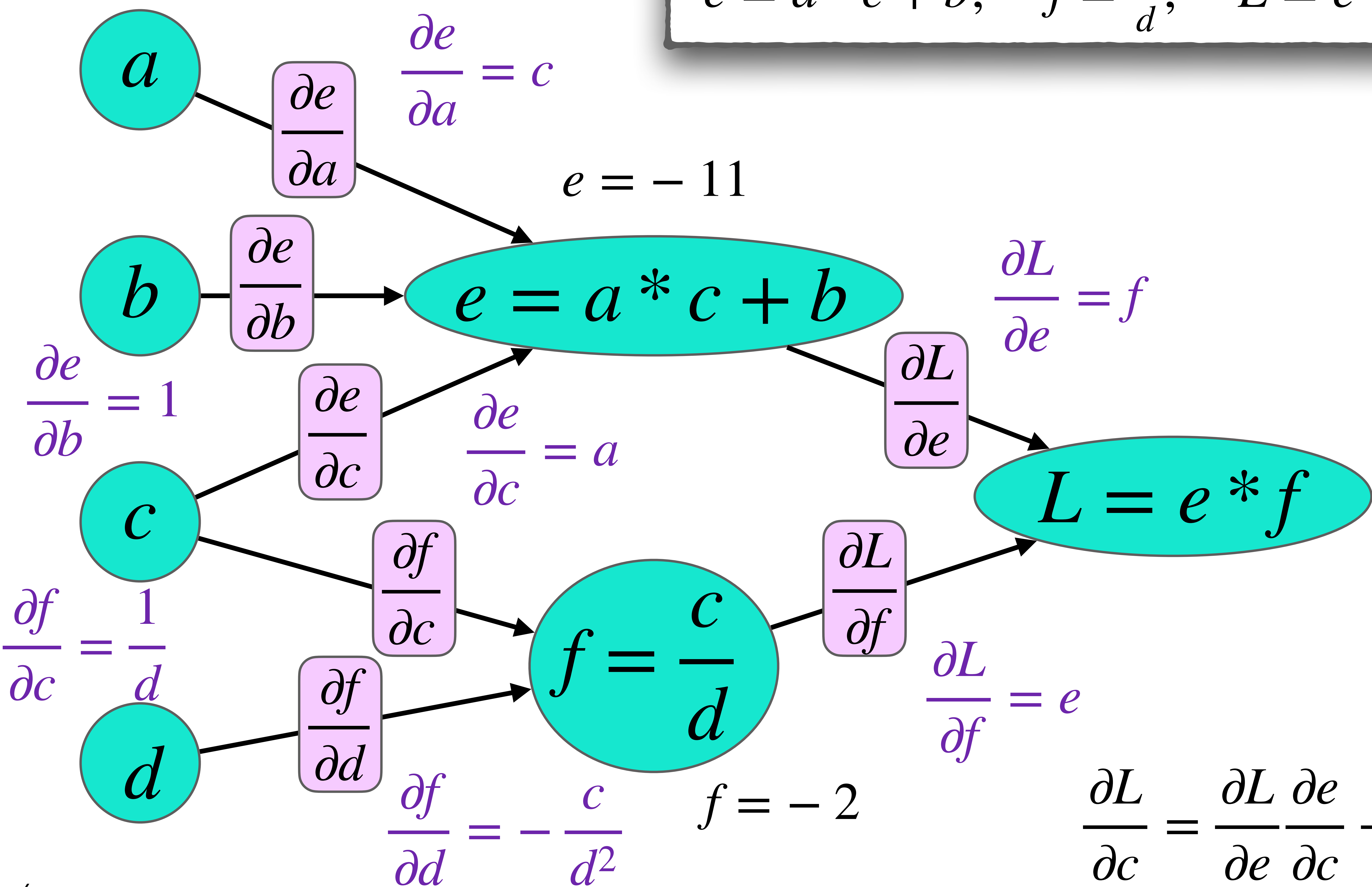
Quiz 3 Answers

Quiz 3

1. Draw the computation graph for the following example and use it to find the gradients for L with respect to all variables, a, b, c, d, e and f.
   $e = a * c + b; \quad f = \frac{c}{d}; \quad L = e * f; \quad a = 3; \quad b = 1; \quad c = -4; \quad d = 2$
2. Which is more expensive: forward propagation or backpropagation? Explain why.
3. Why is non-linearity important in neural networks? What happens if we remove the non-linear activation function in neural networks?
4. Why are vanishing gradients a problem for RNNs? Why is the problem not as severe in feedforward NNs?
5. What's the advantage of weight tying in RNNs? Remember, weight tying is the use of the same weight parameters $\mathbf{W}_h$ and $\mathbf{W}^{[l]}$—representing the lth layer of the RNN—at all time steps.

1. Draw the computation graph for the following example and use it to find the gradients for L with respect to all variables, a, b, c, d, e and f.

$$e = a * c + b; \quad f = \frac{c}{d}; \quad L = e * f; \quad a = 3; \quad b = 1; \quad c = -4; \quad d = 2$$



$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} = fc = 8$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial b} = f = -2$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial d} = -\frac{ec}{d^2} = 11$$

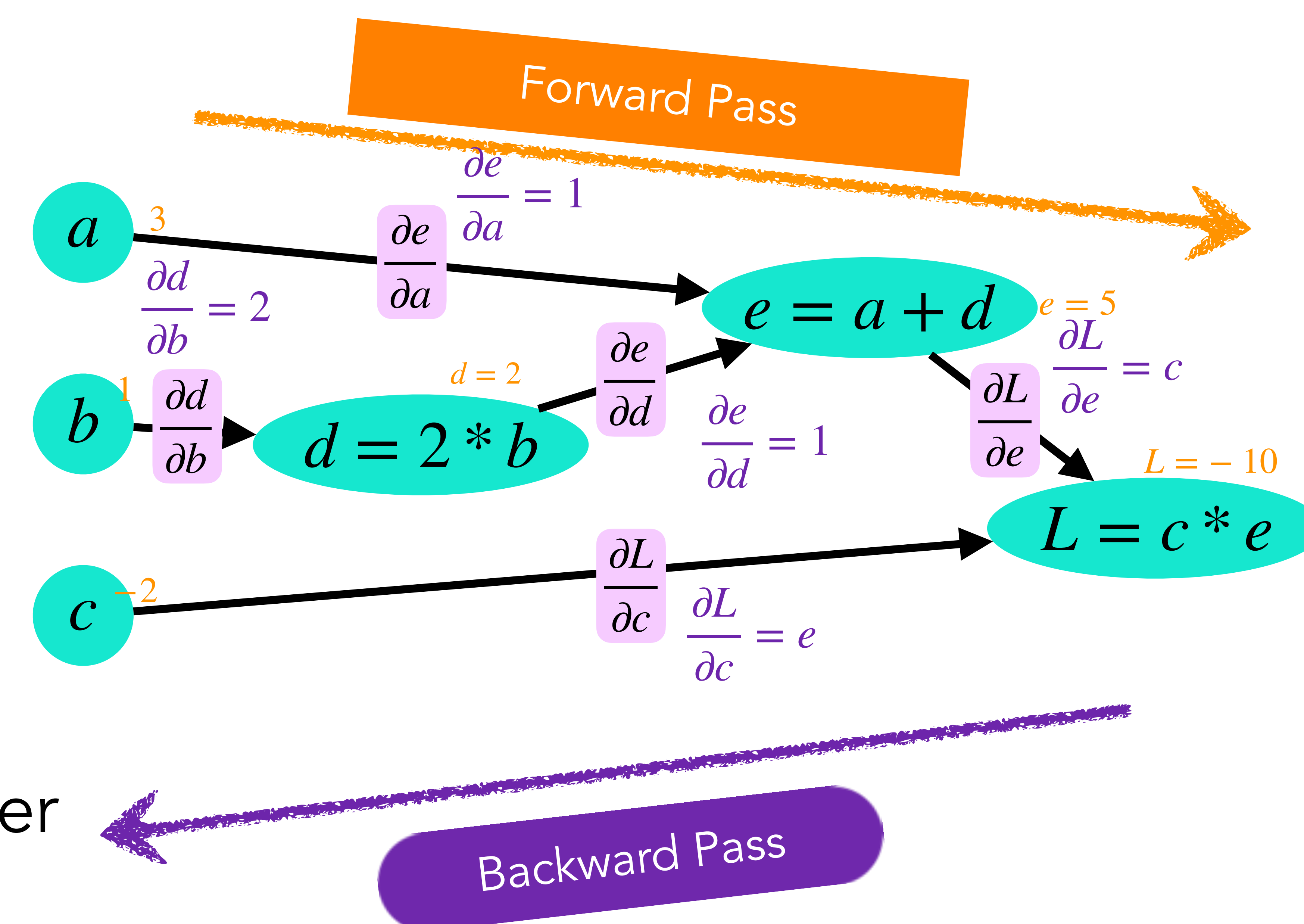
$$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial c} + \frac{\partial L}{\partial f}\frac{\partial f}{\partial c} = af + \frac{e}{d} = -6 - 11/2 = -23/2$$

2. Which is more expensive: forward propagation or backpropagation? Explain why.

A single run of back propagation is more computationally expensive than forward propagation, because it involves computing all gradients and updating the weights.

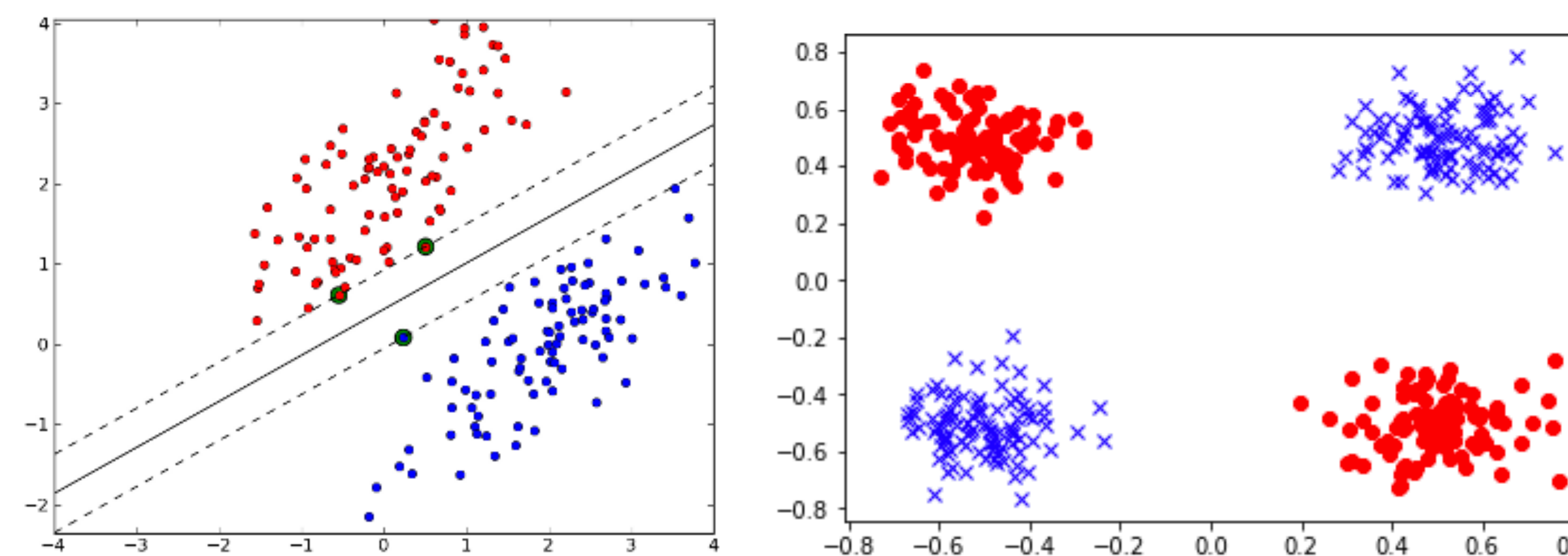
For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer
      - Update the weight
    - For every hidden node
      - Assess how much blame it deserves for the current answer
      - For every weight w from input layer to the hidden layer
        - Update the weight



However, once a model has been trained, we only run forward propagation with it - which amounts to all the costs of running the model.

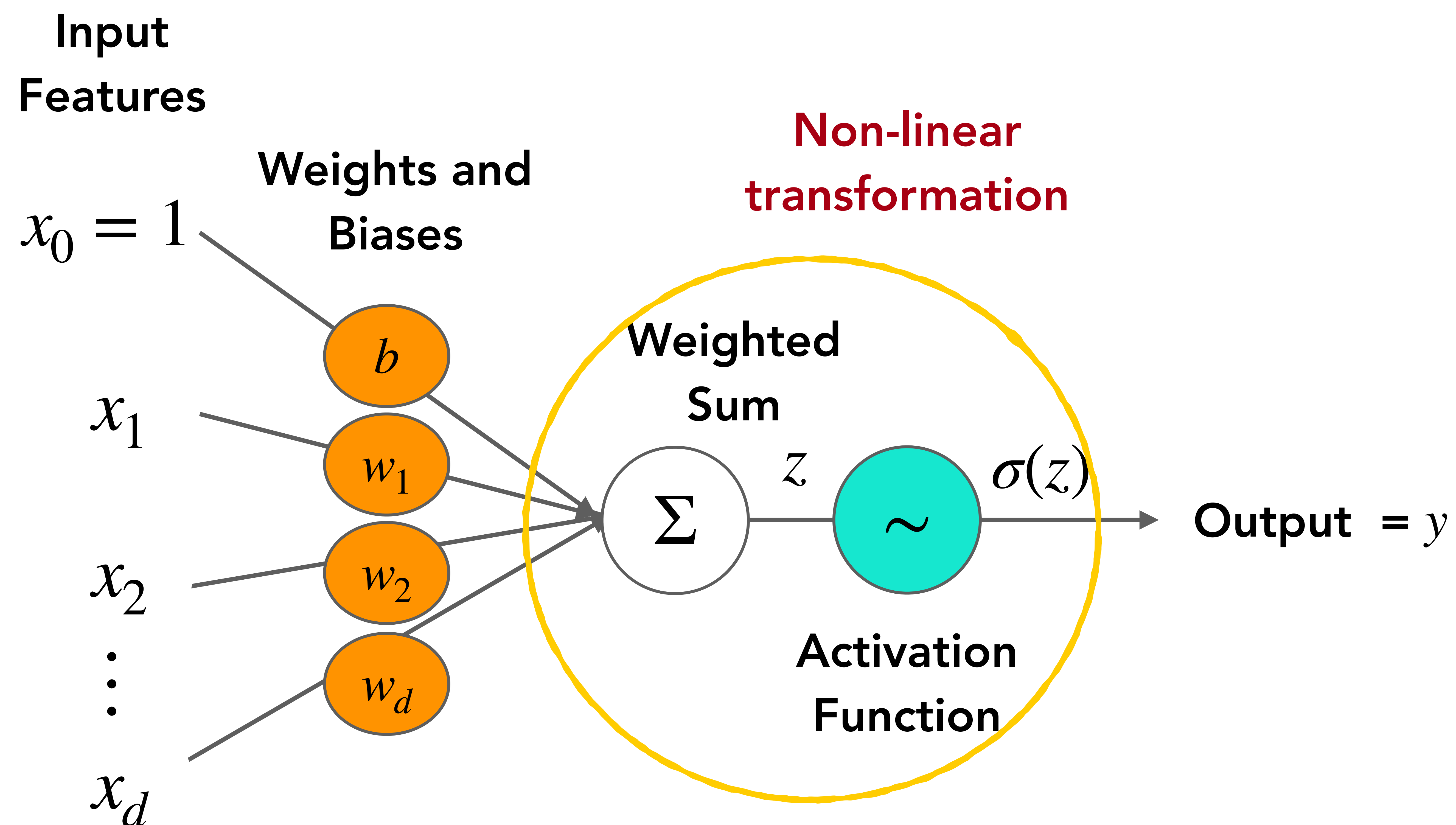
3. Why is non-linearity important in neural networks? What happens if we remove the non-linear activation function in neural networks?



Non-linearity enables arbitrarily complex boundaries between data points; therefore it can represent any function!

Non-linearity enables probabilistic models

If we remove the non- linear activation functions, the resulting (multi-layer) network is exactly equivalent to a single-layer linear network.



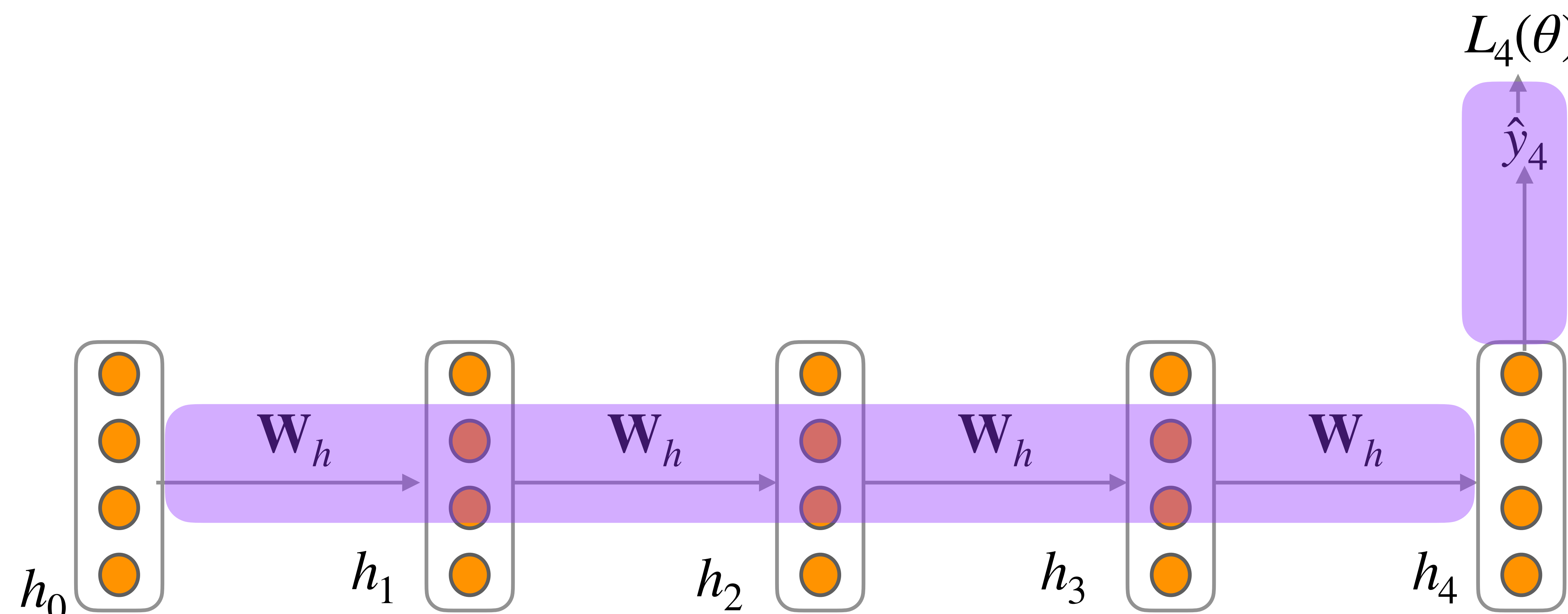
4. Why are vanishing gradients a problem for RNNs? Why is the problem not as severe in feedforward NNs?

$$\frac{\partial L_4}{\partial h_0} = \frac{\partial h_1}{\partial h_0} \times \frac{\partial L_4}{\partial h_1}$$

$$= \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial L_4}{\partial h_2}$$

$$= \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \frac{\partial L_4}{\partial h_3}$$

$$= \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \frac{\partial h_4}{\partial h_3} \times \frac{\partial L_4}{\partial h_4}$$



When these gradients are small, the gradient signal gets smaller and smaller as it backpropagates further…

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

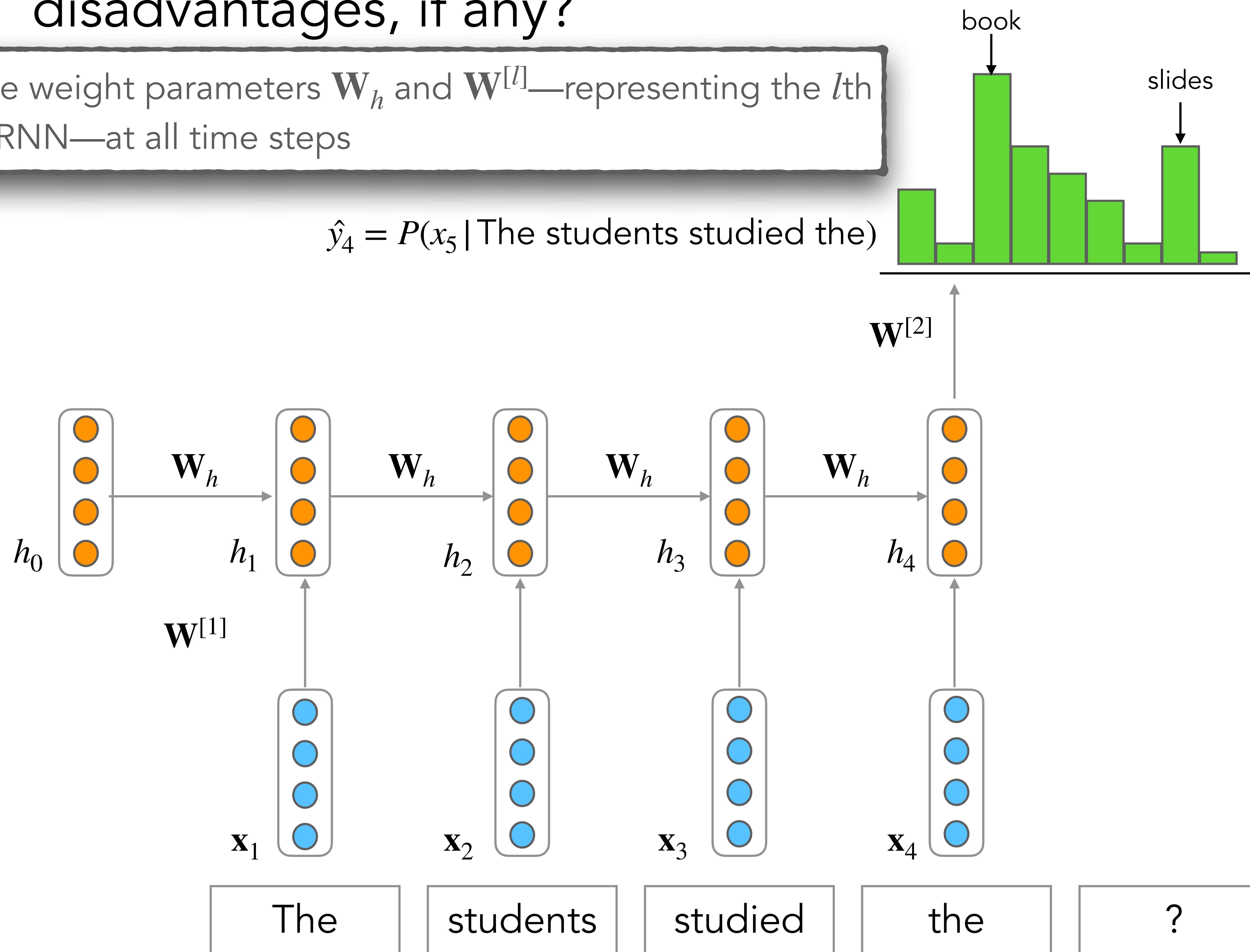
Feedforward networks do not consider arbitrarily long contexts, just a fixed window. Barring a very large window size, there is lower risk of gradient dependence at long ranges

USC Viterbi

5. What's the advantage of weight tying in RNNs? What would be the disadvantages, if any?

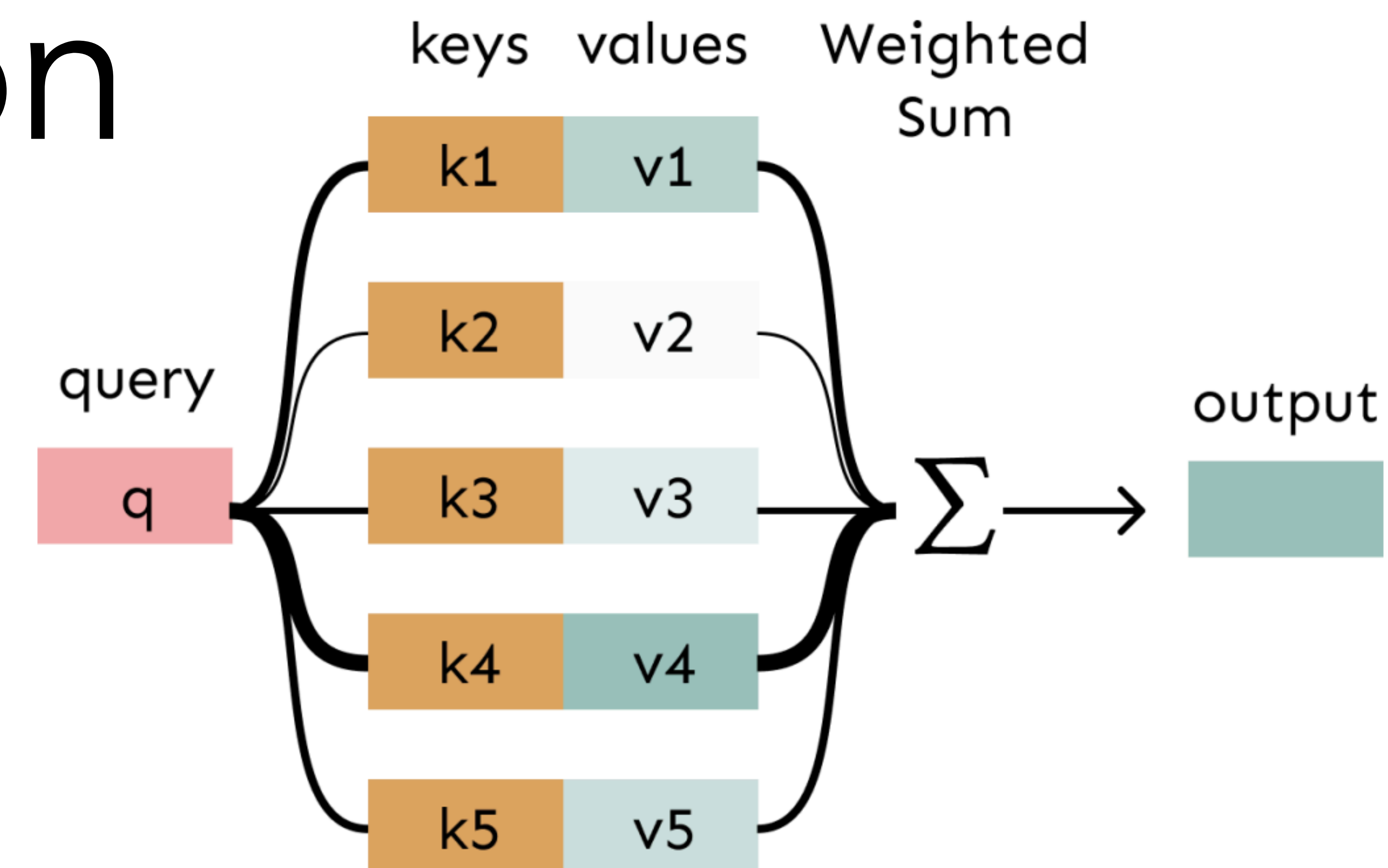
Remember, weight tying is the use of the same weight parameters $\mathbf{W}_h$ and $\mathbf{W}^{[l]}$—representing the lth layer of the RNN—at all time steps

- Weights $\mathbf{W}^{[1]}$ are shared (tied) across timesteps → Condition the neural network on all previous words
- Advantage: Keeps the number of parameters manageable
- Disadvantage: The same weights are responsible for all words in the history



Recap: Transformers as Self-Attention Networks

Self-Attention



Keys, Queries, Values from the same sequence

Let $\mathbf{w}_{1:N}$ be a sequence of words in vocabulary V

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

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

$$q_i = Q\boldsymbol{x}_i \text{ (queries)} \qquad k_i = K\boldsymbol{x}_i \text{ (keys)} \qquad v_i = V\boldsymbol{x}_i \text{ (values)}$$

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

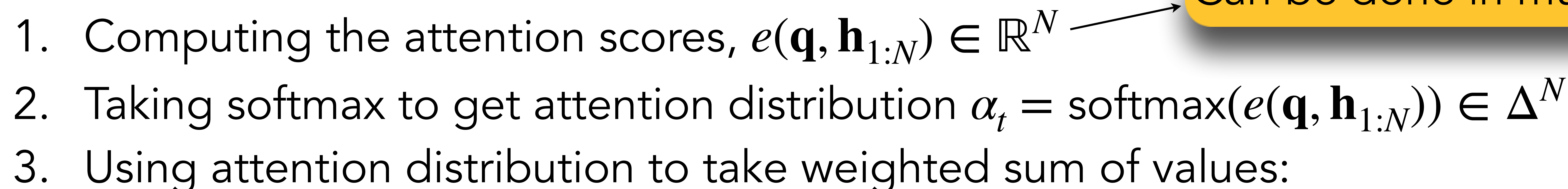
$$\boldsymbol{e}_{ij} = \boldsymbol{q}_i^\top \boldsymbol{k}_j \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'} \exp(\boldsymbol{e}_{ij'})}$$

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

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

Attention Variants

- In general, we have some keys $\mathbf{h}_1, \ldots, \mathbf{h}_N \in \mathbb{R}^{d_1}$ and a query $\mathbf{q} \in \mathbb{R}^{d_2}$
- Attention always involves



1. Computing the attention scores, $e(\mathbf{q}, \mathbf{h}_{1:N}) \in \mathbb{R}^N$ → Can be done in multiple ways!
2. Taking softmax to get attention distribution $\alpha_t = \text{softmax}(e(\mathbf{q}, \mathbf{h}_{1:N})) \in \Delta^N$
3. Using attention distribution to take weighted sum of values:

$$\mathbf{c}_t^{att} = \sum_{i=1}^{N} \alpha_{t,i} \mathbf{h}_i \in \mathbb{R}^{d_1}$$

This leads to the attention output $\mathbf{c}_t^{att}$ (sometimes called the attention context vector)

Attention Variants

- There are several ways you can compute $e(\mathbf{q}, \mathbf{h}_{1:N}) \in \mathbb{R}^N$ from $\mathbf{h}_1 \ldots \mathbf{h}_N \in \mathbb{R}^{d_1}$ and $\mathbf{q} \in \mathbb{R}^{d_2}$
- Basic dot-product attention: $e(\mathbf{q}, \mathbf{h}_{1:N}) = [\mathbf{q} \cdot \mathbf{h}_j]_{j=1:N}$
  - This assumes $d_1 = d_2$
  - We applied this in encoder-decoder RNNs
- Multiplicative attention: $e(\mathbf{q}, \mathbf{h}_{1:N}) = [\mathbf{q}^T \mathbf{W} \mathbf{h}_j]_{j=1:N}$
  - Where $\mathbf{W} \in \mathbb{R}^{d_2 \times d_1}$ is a learned weight matrix.
  - Also called “bilinear attention”

USC Viterbi

More on Attention

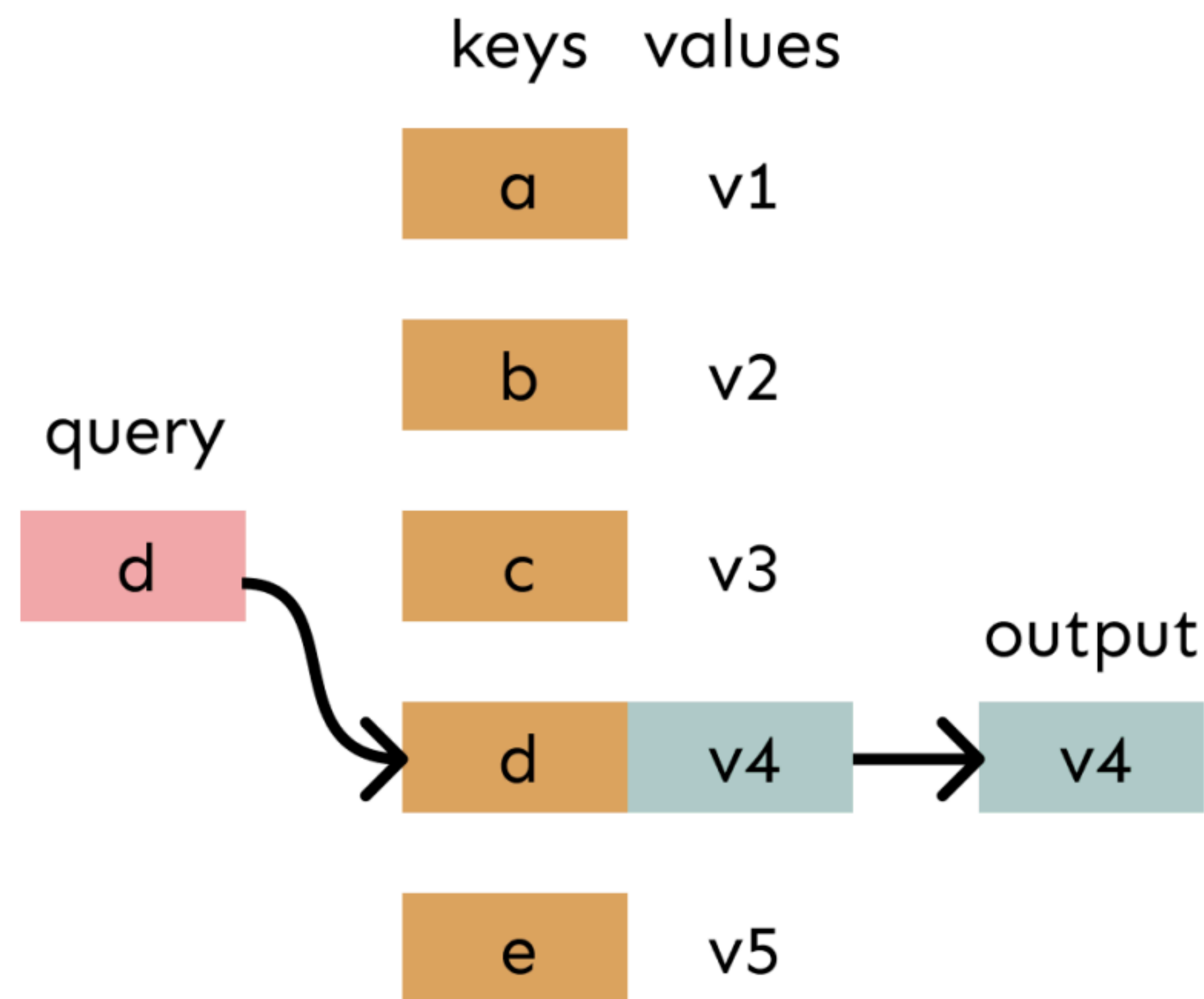
Given a set of vector values, and a vector query, attention is a technique to compute a weighted sum of the values, dependent on the query

- We sometimes say that the query attends to the values.
  - For example, in the seq2seq + attention model, each decoder hidden state (query) attends to all the encoder hidden states (values)
  - You use the keys to get to the values
- The weighted sum is a **selective summary** of the information contained in the values, where the query determines which values to focus on.
- Attention is a way to obtain a **fixed-size representation** of an **arbitrary set of representations** (the values), dependent on some other representation (the query).
- Attention is a powerful, flexible, general deep learning technique in all deep learning models.
  - A new idea from after 2010! Originated in NMT

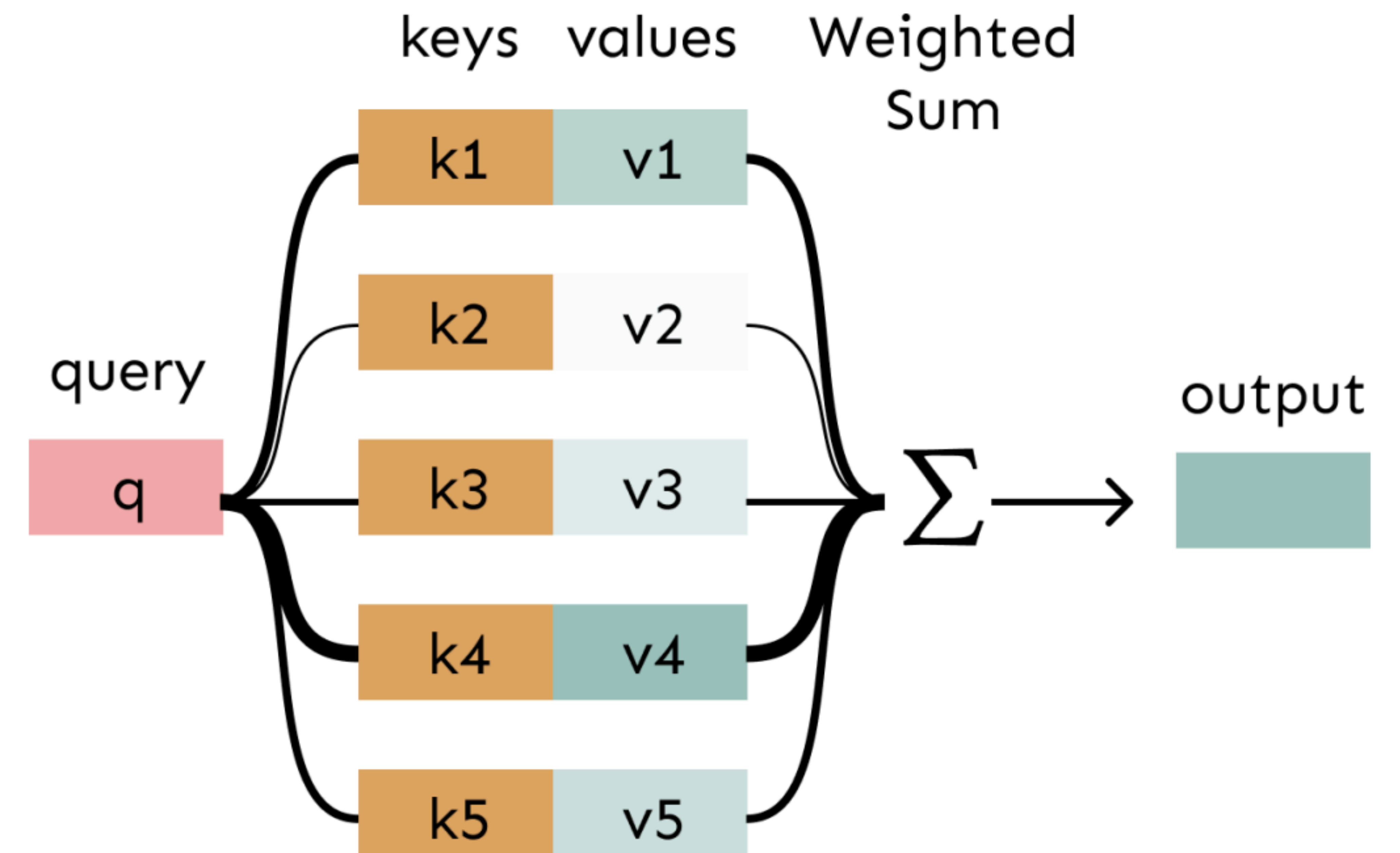
Attention and lookup tables

Attention performs fuzzy lookup in a key-value store

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

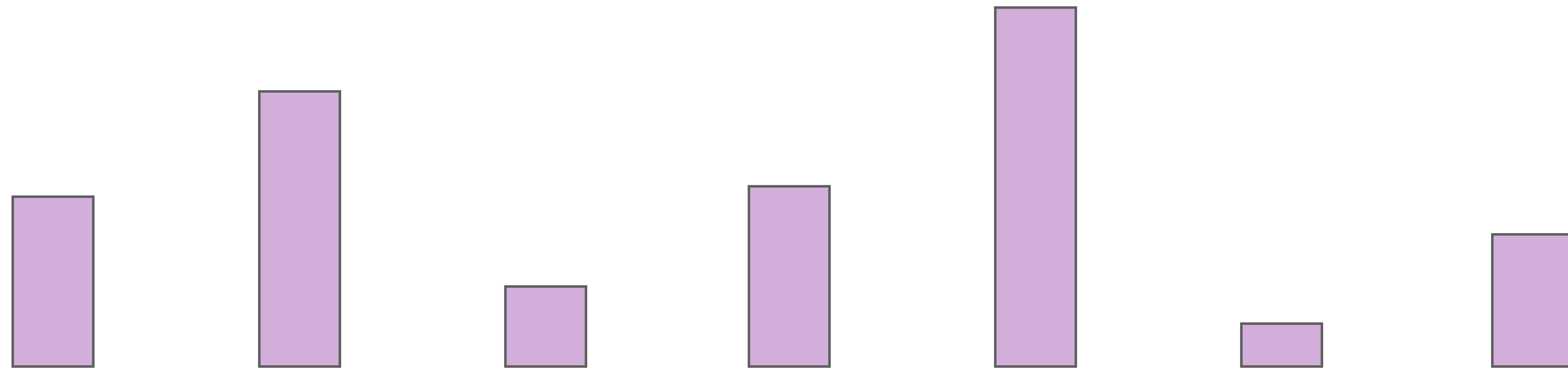


In attention, the query matches all keys softly, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.

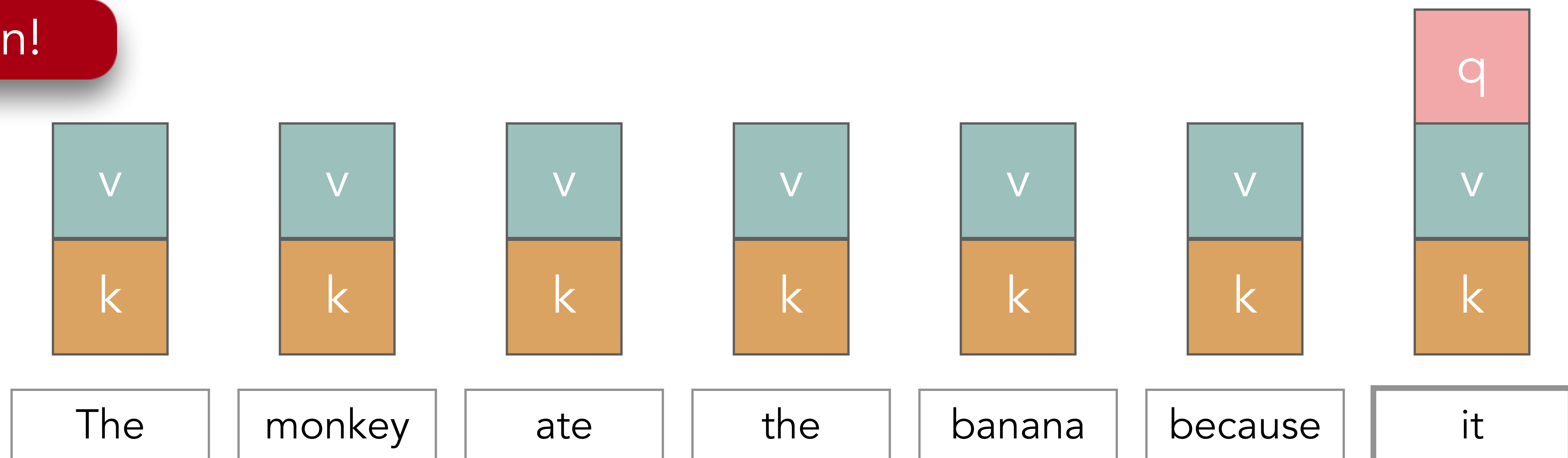


USC Viterbi

Attention in the decoder



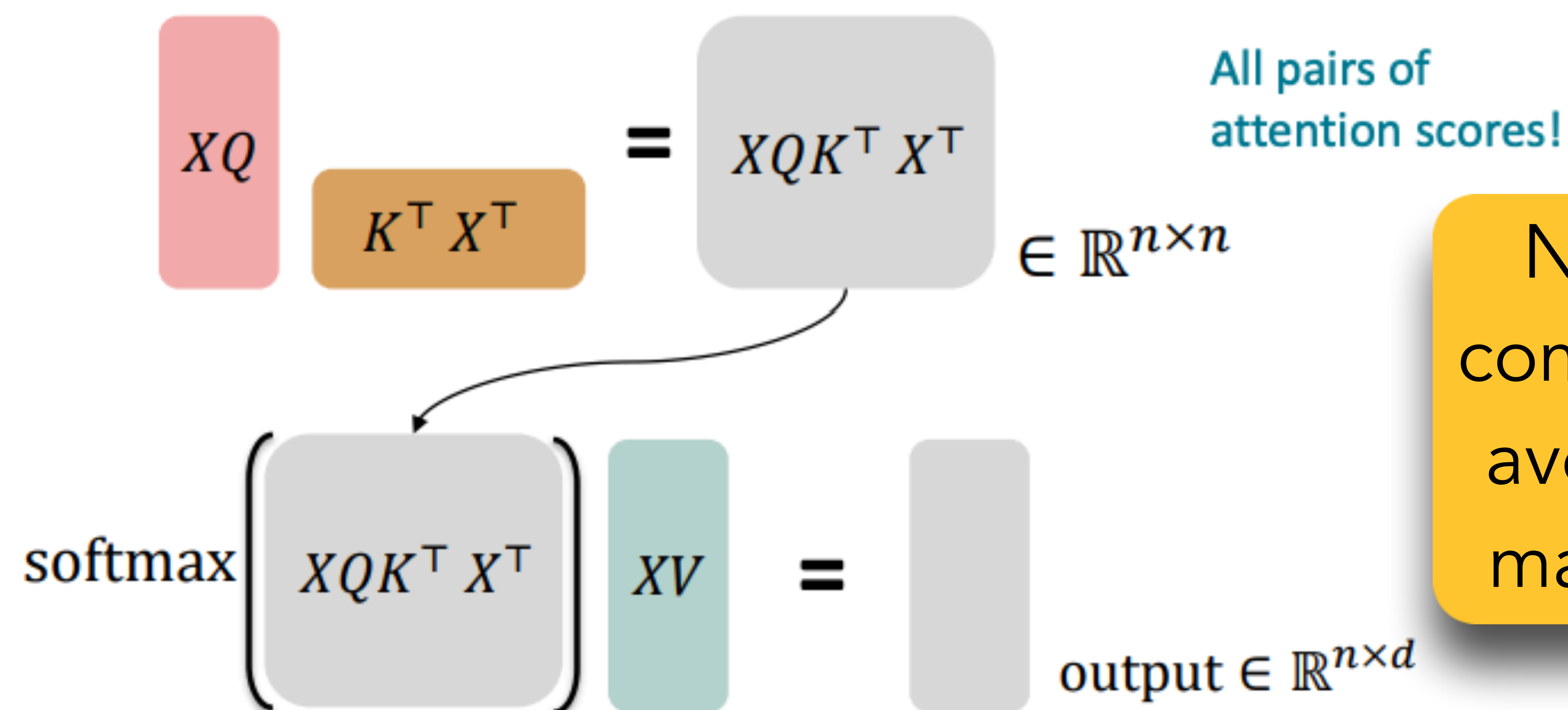
Self-Attention!



Self-Attention as Matrix Multiplications

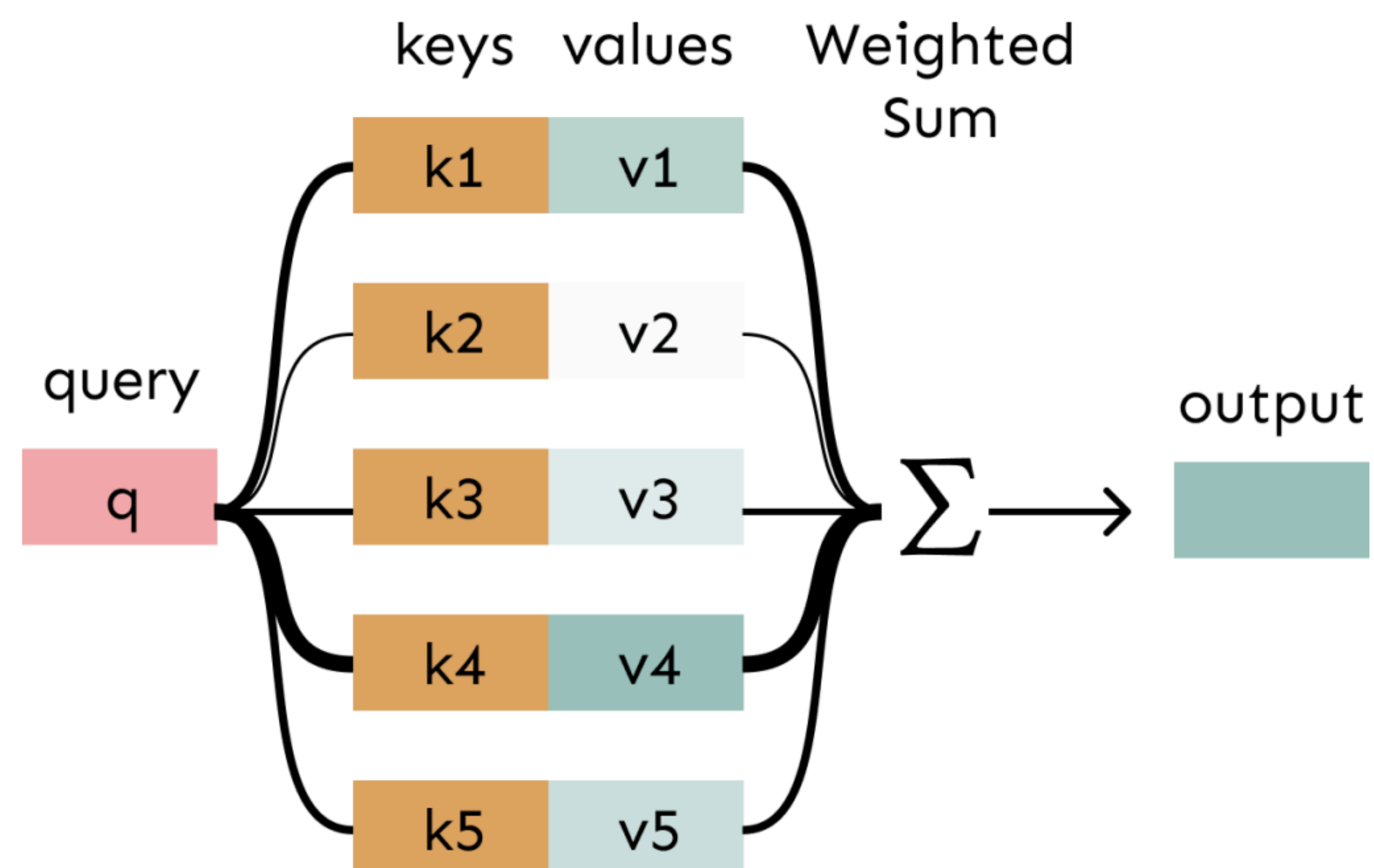
- Key-query-value attention is typically computed as matrices.
  - Let $\mathbf{X} = [\mathbf{x}_1; \ldots; \mathbf{x}_n] \in \mathbb{R}^{n \times d}$ be the concatenation of input vectors
  - First, note that $\mathbf{XK} \in \mathbb{R}^{n \times d}$, $\mathbf{XQ} \in \mathbb{R}^{n \times d}$, and $\mathbf{XV} \in \mathbb{R}^{n \times d}$
  - The output is defined as softmax$(\mathbf{XQ}(\mathbf{XK})^T)\mathbf{XV} \in \mathbb{R}^{n \times d}$

First, take the query-key dot products in one matrix multiplication: $\mathbf{XQ}(\mathbf{XK})^T$



Next, softmax, and compute the weighted average with another matrix multiplication.

Why Self-Attention?



- Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs
- Used often with feedforward networks!

USC Viterbi

Transformers are Self-Attention Networks

- Self-Attention is the key innovation behind Transformers!
- Transformers map sequences of input vectors $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ to sequences of output vectors $(\mathbf{y}_1, \ldots, \mathbf{y}_n)$ of the same length.
- Made up of stacks of Transformer blocks
    - each of which is a multilayer network made by combining
        - simple linear layers,
        - feedforward networks, and
        - self-attention layers

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

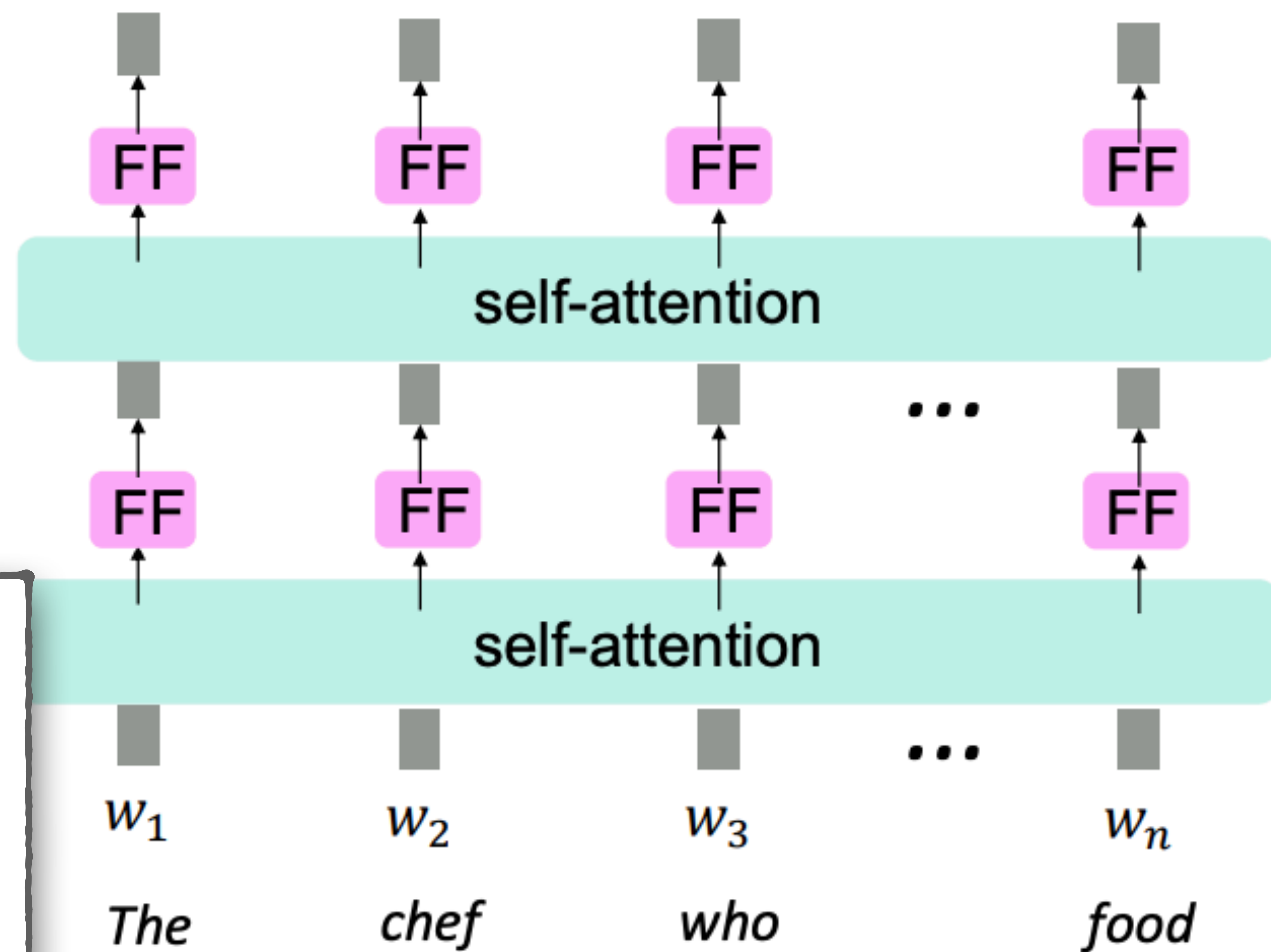
Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

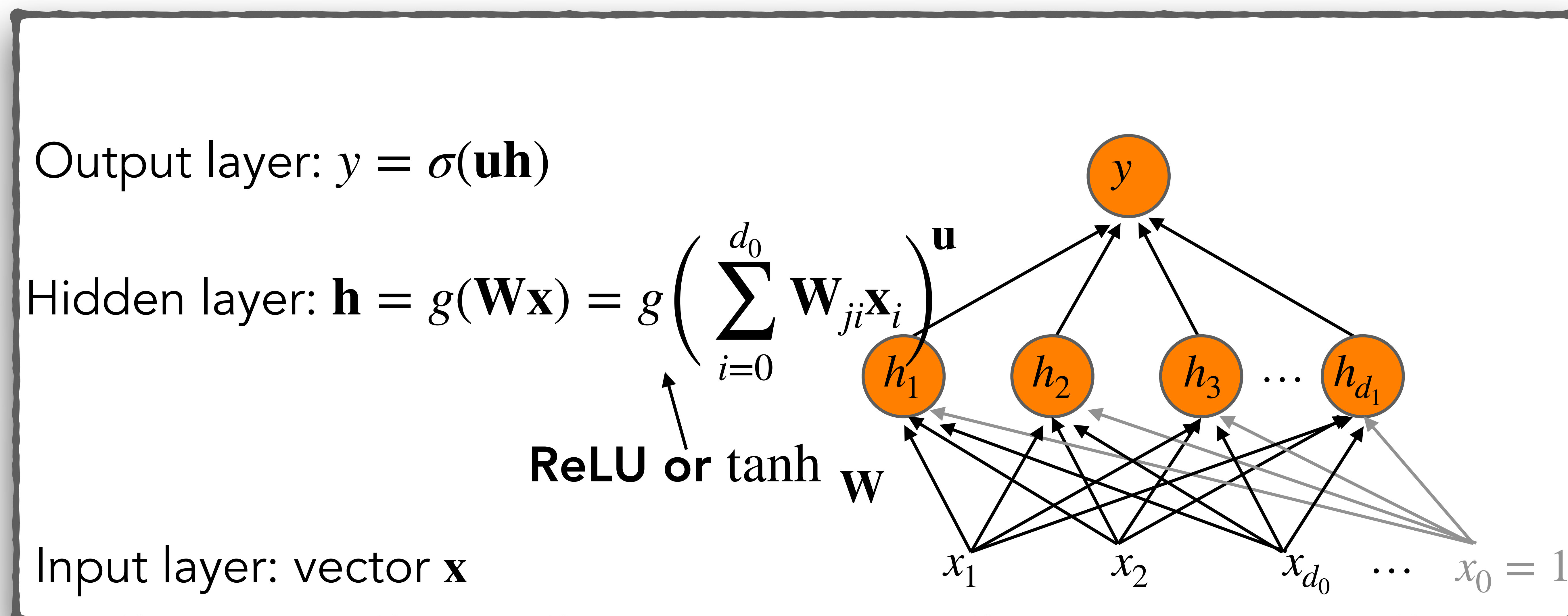
Łukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Self-Attention and Weighted Averages

- **Problem**: there are no element-wise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors
- **Solution**: add a feed-forward network to post-process each output vector.





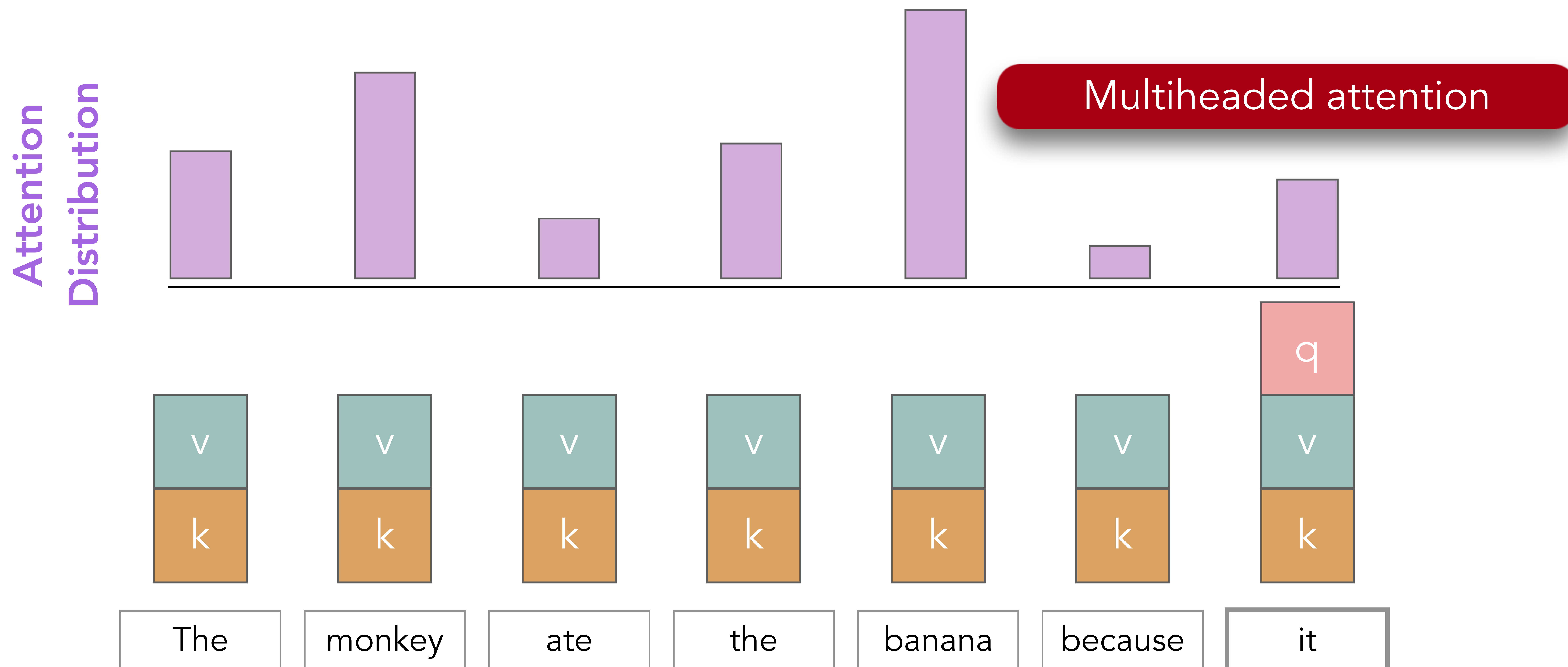
Self Attention and Future Information

- **Problem**: Need to ensure we don't "look at the future" when predicting a sequence
  - e.g. Target sentence in machine translation or generated sentence in language modeling
  - To use self-attention in decoders, we need to ensure we can't peek at the future.
- **Solution (Naïve)**: At every time step, we could change the set of keys and queries to include only past words.
  - (Inefficient!)
- **Solution:** To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$

	[START]	The	chef	who
[START]		$-\infty$	$-\infty$	$-\infty$
The			$-\infty$	$-\infty$
chef				$-\infty$
who				

Self-Attention and Heads

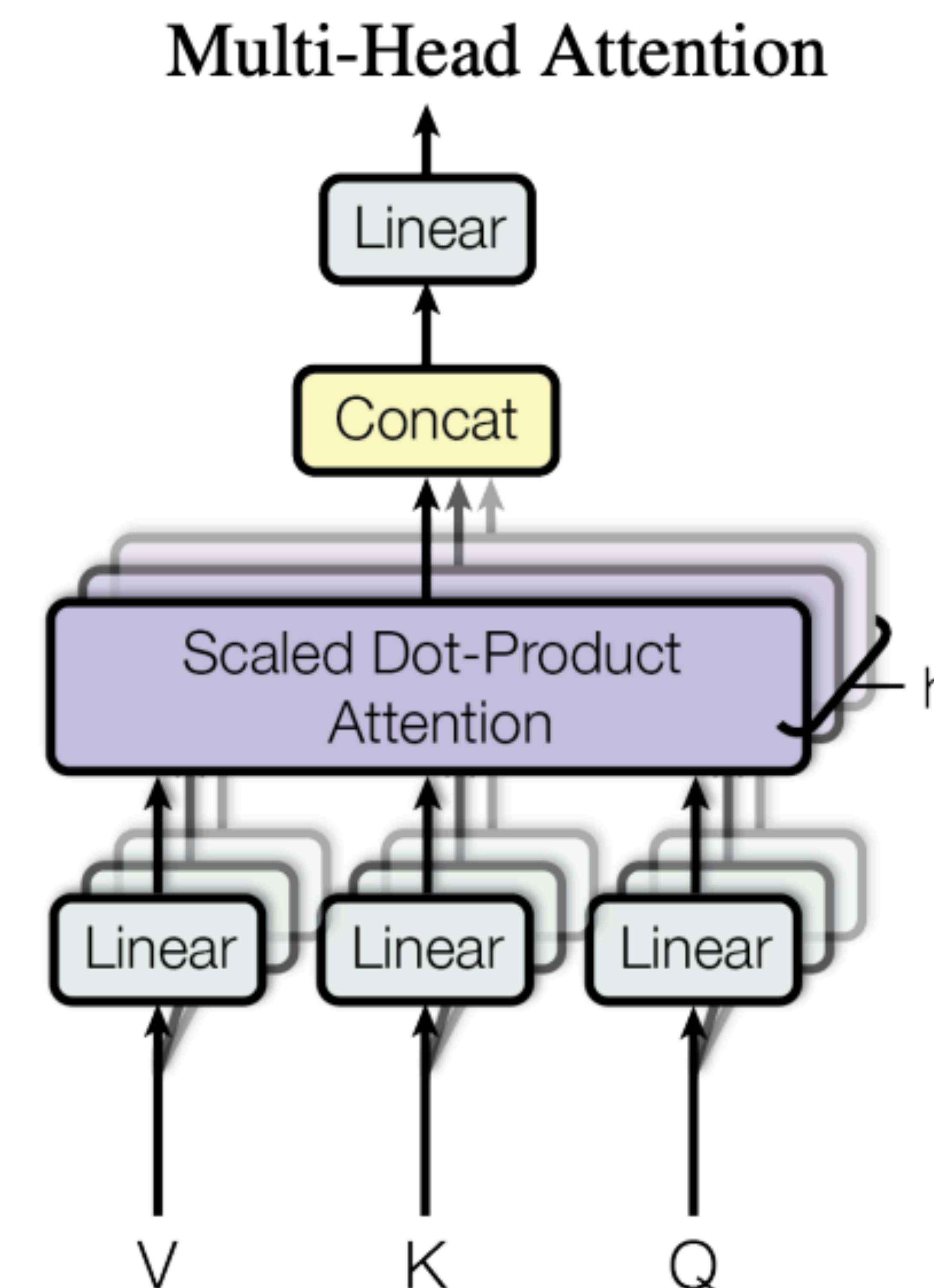
- What if we needed to pay attention to multiple different kinds of things e.g. entities, syntax
- **Solution**: Consider multiple attention computations in parallel



Transformers: Multiheaded Attention

Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $\mathbf{x}_i^T \mathbf{Q}^T (\mathbf{K}\mathbf{x}_j)$ is high, but maybe we want to focus on different j for different reasons?
- We'll define multiple attention "heads" through multiple $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ matrices
- Let $\mathbf{Q}_l, \mathbf{K}_l, \mathbf{V}_l$, each in $\mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and $1 \leq l \leq h$.
- Each attention head performs attention independently:
- Then the outputs of all the heads are combined!



Each head gets to "look" at different things, and construct value vectors differently

USC Viterbi

Multiheaded Attention: Visualization

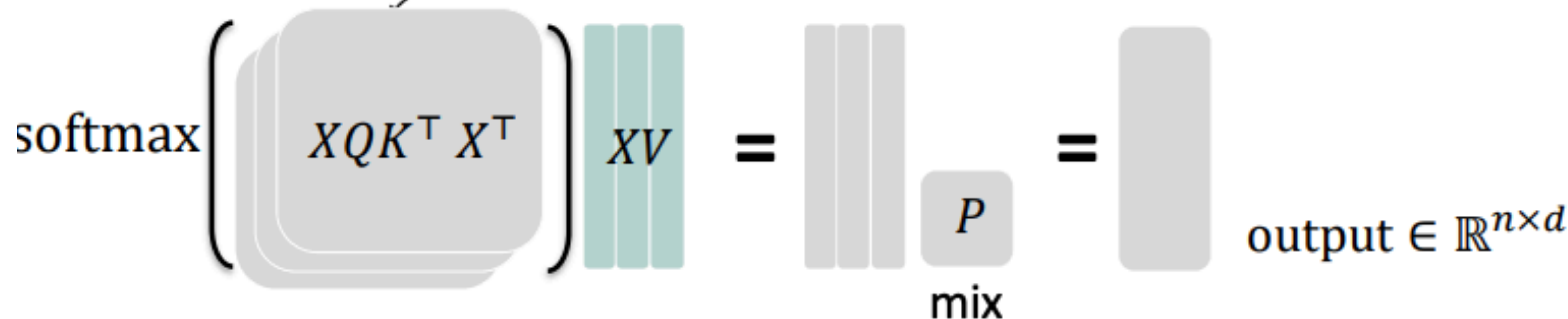
Still efficient, can be parallelized!

Tensor!

First, take the query-key dot products in one matrix multiplication: $\mathbf{XQ}_l(\mathbf{XK}_l)^T$

3 sets of all pairs of attention scores!

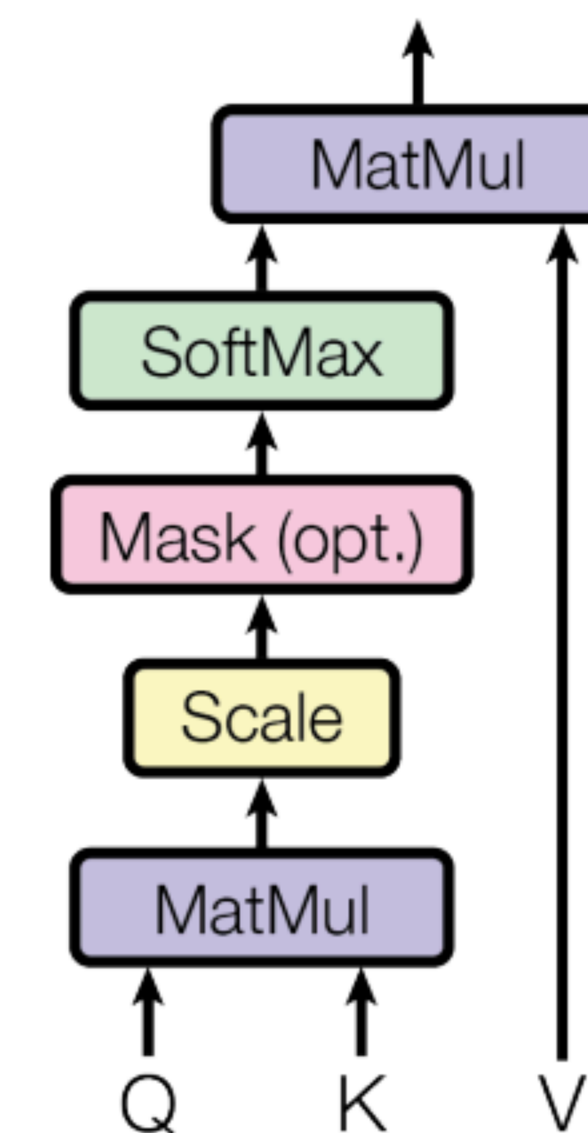
Next, softmax, and compute the weighted average with another matrix multiplication.



Scaled Dot Product Attention

$$\textbf{output}_\ell = \textbf{softmax}(XQ_\ell K_\ell^T X^T) * XV_\ell$$

- So far: Dot product self-attention
- When dimensionality d becomes large, dot products between vectors tend to become large
- Because of this, inputs to the softmax function can be large, making the gradients small
- Now: Scaled Dot product self-attention to aid in training



$$\textbf{output}_\ell = \textbf{softmax}\left(\frac{XQ_\ell K_\ell^T X^T}{\sqrt{d/h}}\right) * XV_\ell$$

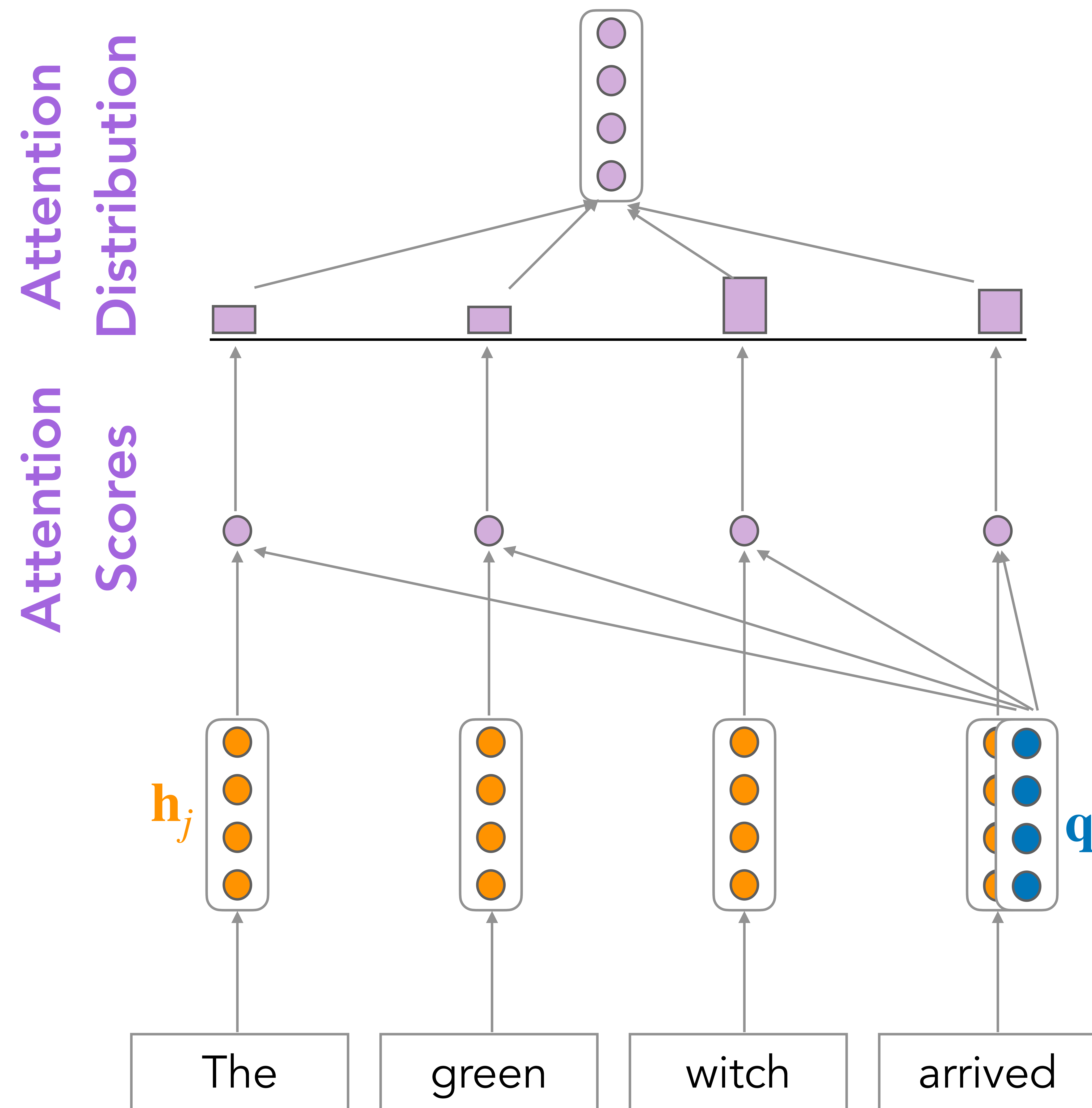
- We divide the attention scores by d/h, to stop the scores from becoming large just as a function of d/h, where h is the number of heads

Attention is all you need (Vaswani et al., 2017)

Self-Attention: Order Information?

- Not necessarily (and not typically) based on Recurrent Neural Nets
- No more order information!
- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.

Do feedforward nets contain order information?



Transformers: Positional Embeddings

Missing: Order Information

- Consider representing each sequence index as a vector
  - $\mathbf{p}_i \in \mathbb{R}^d$, for $i \in \{1,2, \ldots, n\}$ are position vectors
- Don't worry about what the $\mathbf{p}_i$ are made of yet!
- Easy to incorporate this info: just add the $\mathbf{p}_i$ to our inputs!
- Recall that $\mathbf{x}_i$ is the embedding of the word at index i. The positioned embedding is:
  - $\tilde{\mathbf{x}}_i = \mathbf{x}_i + \mathbf{p}_i$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add…

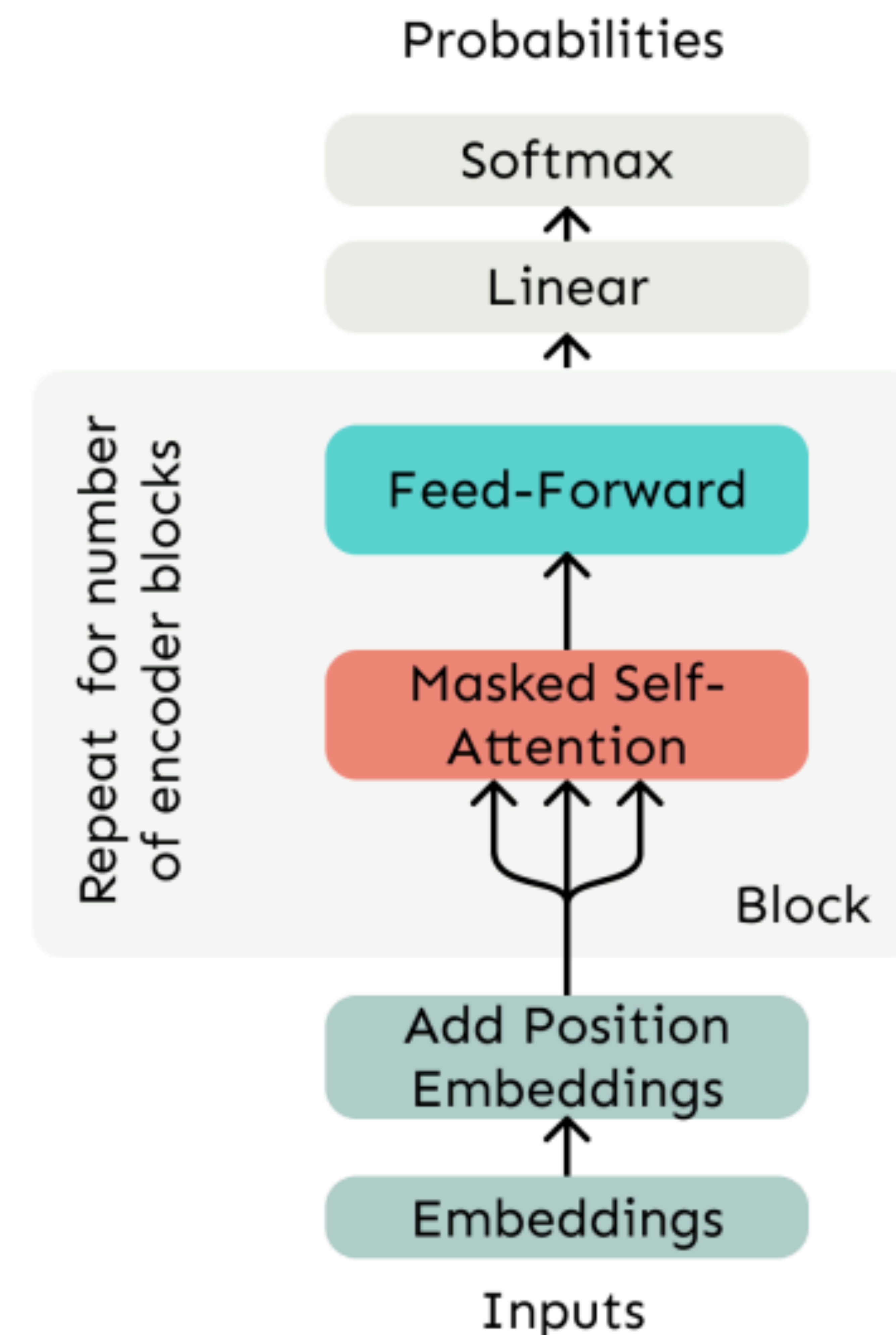
Positional Embeddings

- Maps integer inputs (for positions) to real-valued vectors
    - one per position in the entire context
- Can be randomly initialized and can let all $\mathbf{p}_i$ be learnable parameters (most common)
- Pros:
    - Flexibility: each position gets to be learned to fit the data
- Cons:
    - Definitely can't extrapolate to indices outside $1, \dots, n$.
    - There will be plenty of training examples for the initial positions in our inputs and correspondingly fewer at the outer length limits

Putting it all together: Transformer Blocks

Self-Attention Transformer Building Block

- Self-attention:
  - the basis of the method; with multiple heads
- Position representations:
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- Nonlinearities:
  - At the output of the self-attention block
  - Frequently implemented as a simple feedforward network.
- Masking:
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from "leaking" to the past.



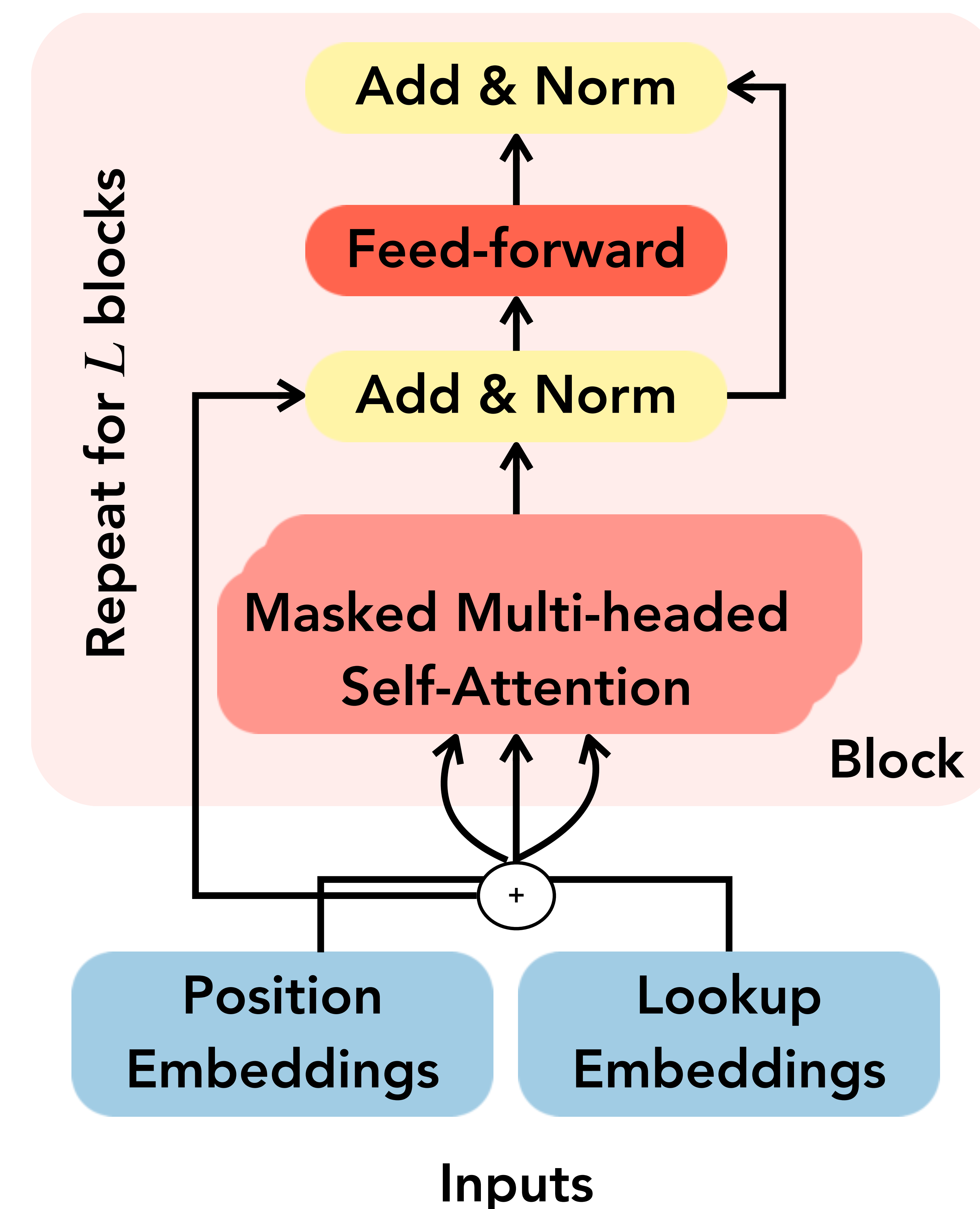
Lecture Outline

- Quiz 3 Answers
- Recap: Transformers as Self-Attention Networks
- Transformers: Multiheaded Attention
- Transformers: Positional Embeddings
- Putting it all together: Transformer Blocks
- Transformers as Encoders, Decoders and Encoder-Decoders

Transformers as Encoders, Decoders and Encoder-Decoders

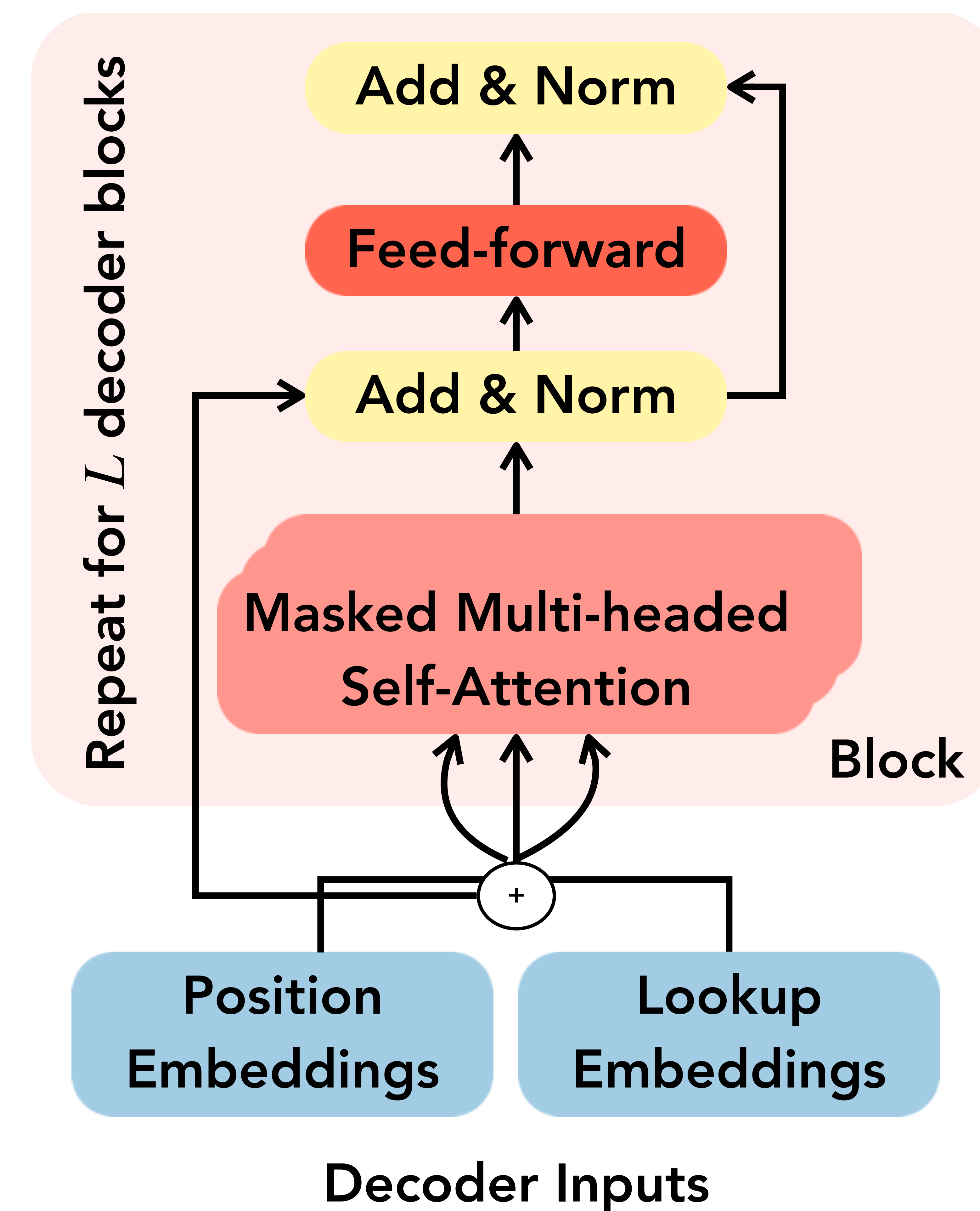
The Transformer Model

- Transformers are made up of stacks of transformer blocks, each of which is a multilayer network made by combining feedforward networks and **self-attention layers**, the key innovation of self-attention transformers
- The Transformer Decoder-only model corresponds to
  - a Transformer language model
- Lookup embeddings can be randomly initialized (more common) or taken from existing resources such as word2vec
  - We will look at tokenization (next week)

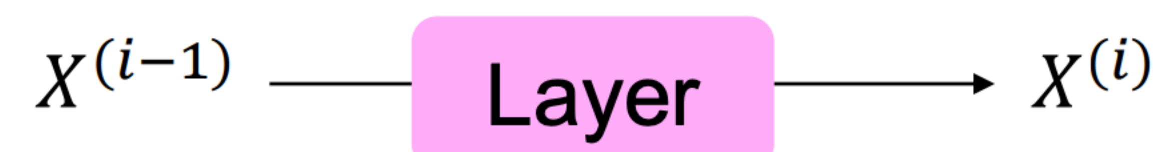


The Transformer Decoder

- Two optimization tricks that help training:
  - Residual Connections
  - Layer Normalization
- In most Transformer diagrams, these are often written together as "Add & Norm"
  - Add: Residual Connections
  - Norm: Layer Normalization



Residual Connections



- Original Connections: $X^{(i)} = \text{Layer}(X^{(i-1)})$ where i represents the layer
- **Residual Connections** : trick to help models train better.
  - We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$
    - so we only have to learn "the residual" from the previous layer

Allowing information to skip a layer improves learning and gives higher level layers **direct access to information** from lower layers (He et al., 2016).

Layer Normalization

- Layer normalization is another trick to help models train faster
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**
- Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.

$$\mu = \frac{1}{d}\sum_{j=1}^{d} x_j; \quad \mu \in \mathbb{R} \qquad \sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2}; \quad \sigma \in \mathbb{R}$$

Result: New vector with zero mean and a standard deviation of one →

$$\hat{x} = \frac{x - \mu}{\sigma}$$

← Component-wise subtraction

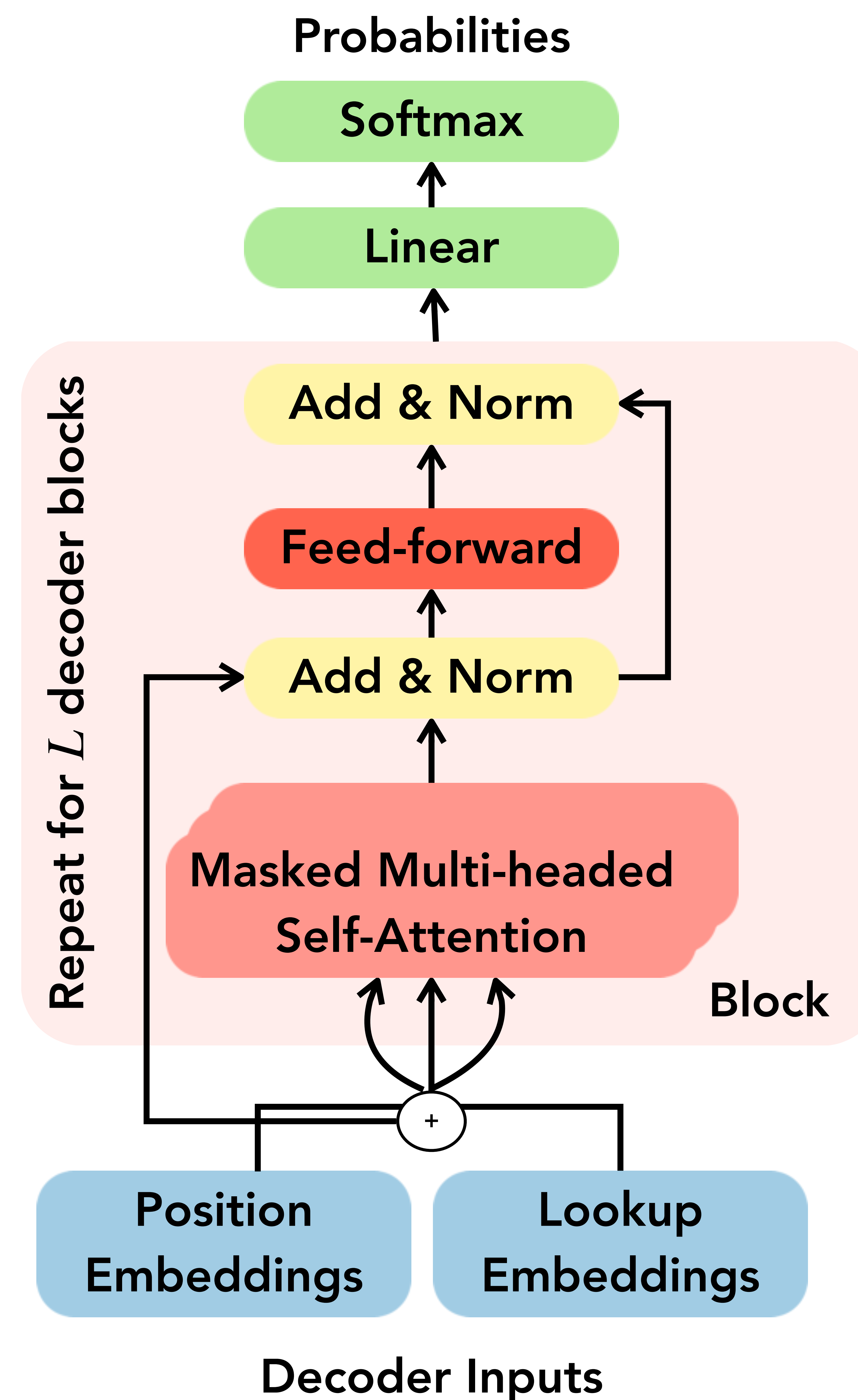
- Let $\gamma \in \mathbb{R}$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters. (Can omit!)

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

Xu et al., 2019

The Transformer Decoder

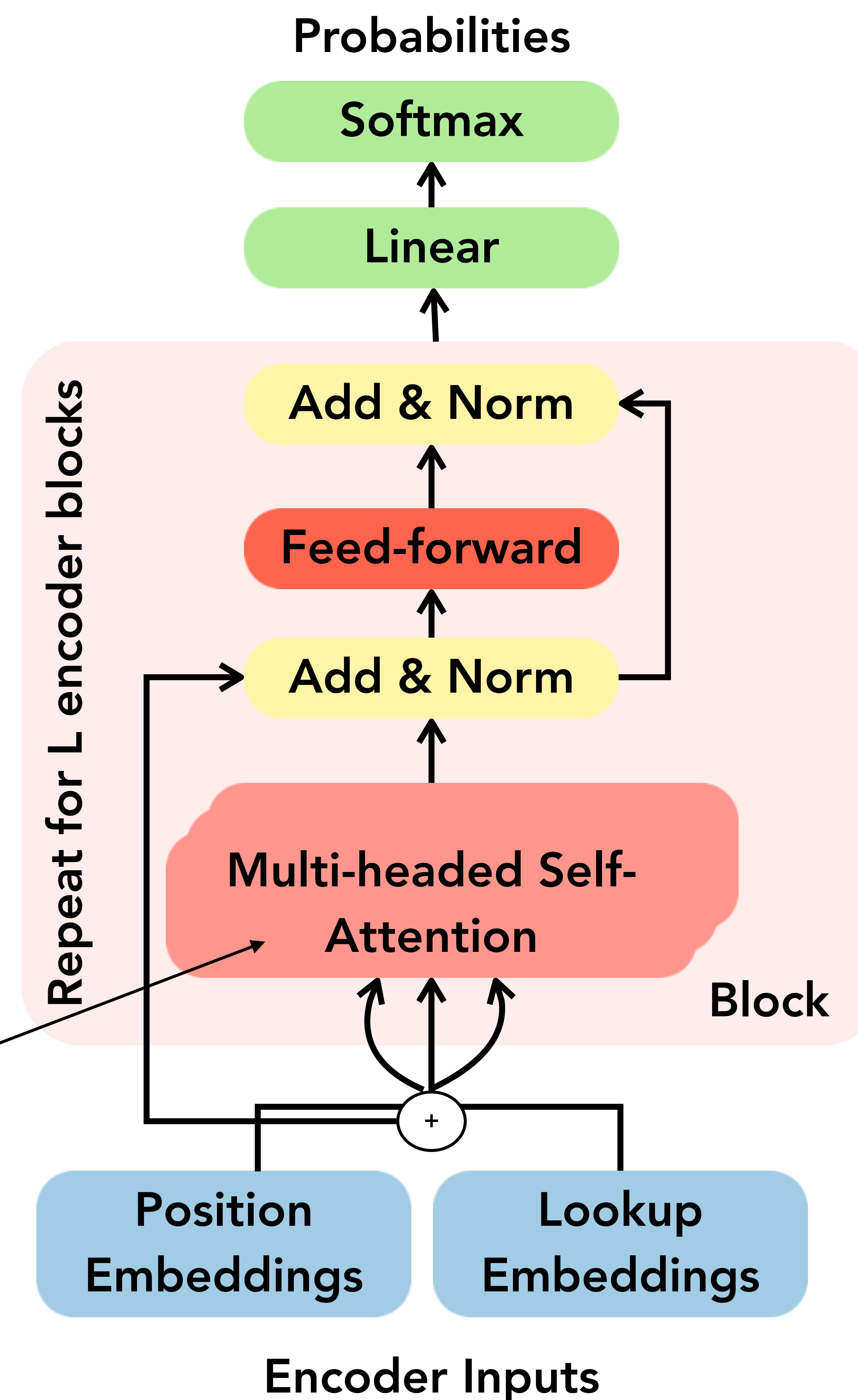
- The Transformer Decoder is a stack of Transformer Decoder Blocks.
- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- Output layer is as always a softmax layer



The Transformer Encoder

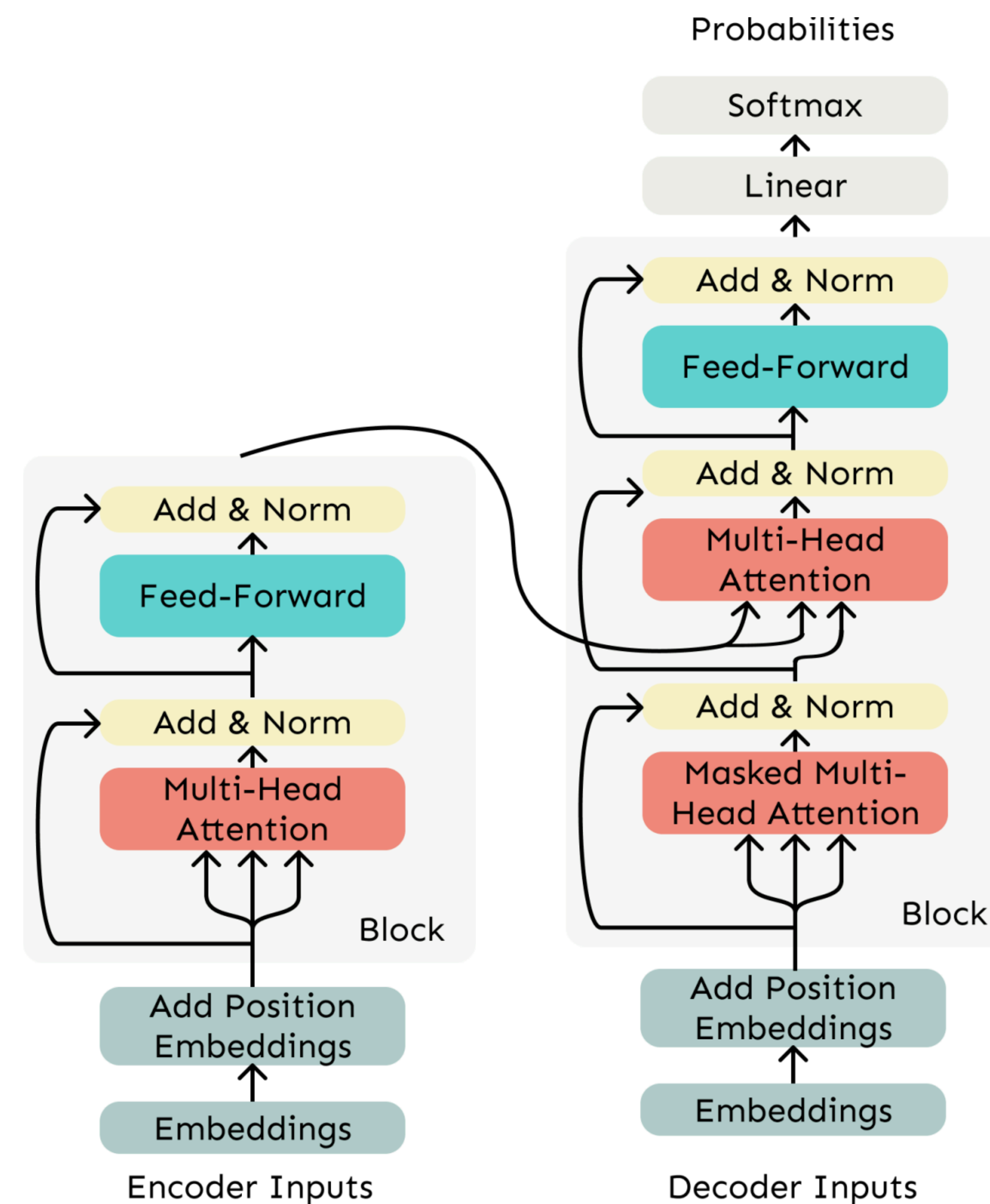
- The Transformer Decoder constrains to unidirectional context, as for language models.
- What if we want bidirectional context, i.e. both left to right as well as right to left?
- The only difference is that we remove the masking in the self-attention.
- Commonly used in sequence prediction tasks such as POS tagging
  - One output token y per input token x

No Masking!



The Transformer Encoder-Decoder

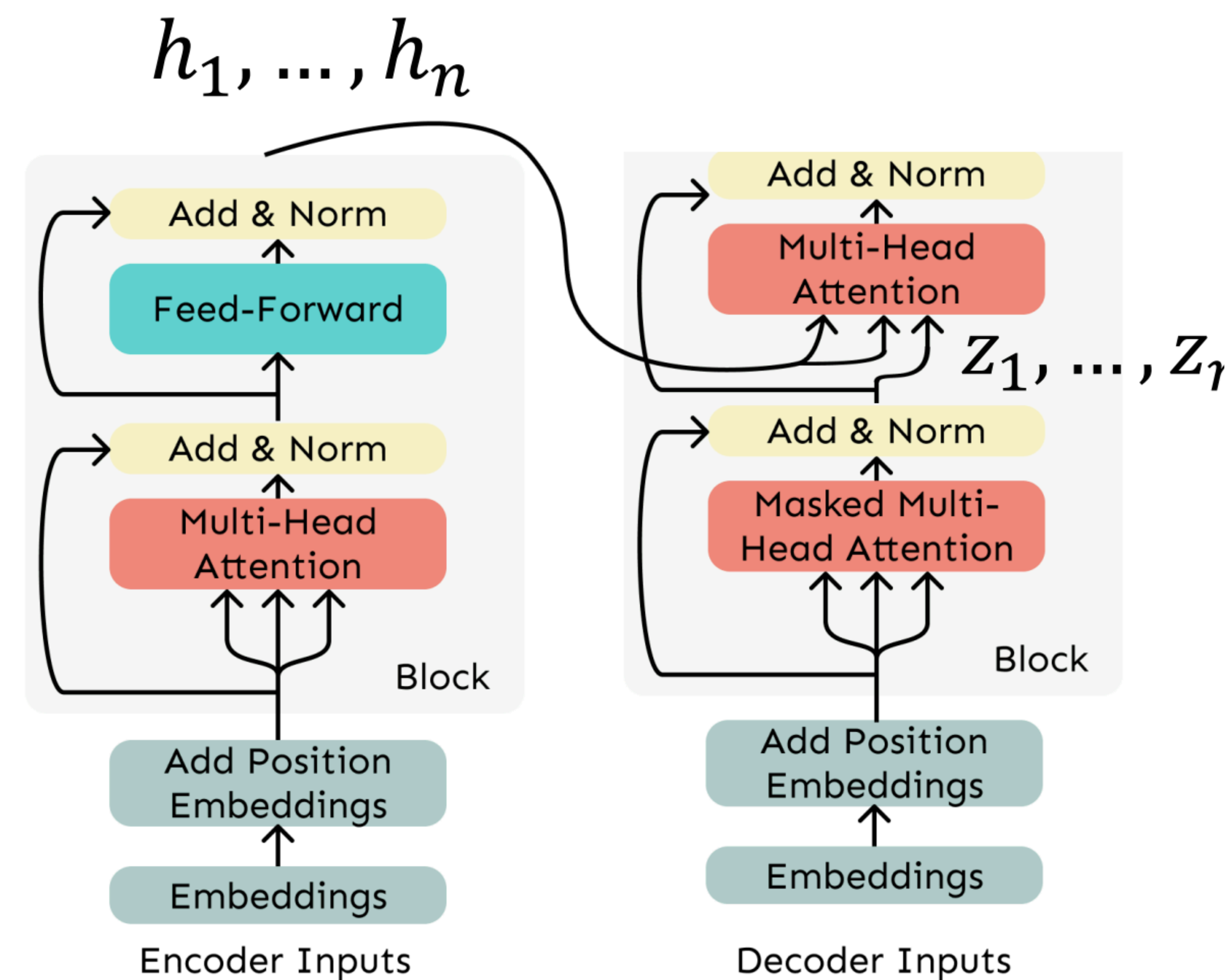
- Recall that in machine translation, we processed the source sentence with a bidirectional model and generated the target with a unidirectional model.
- For this kind of seq2seq format, we often use a Transformer Encoder-Decoder.
- We use a normal Transformer Encoder.
- Our Transformer Decoder is modified to perform **cross-attention** to the output of the Encoder.



USC Viterbi

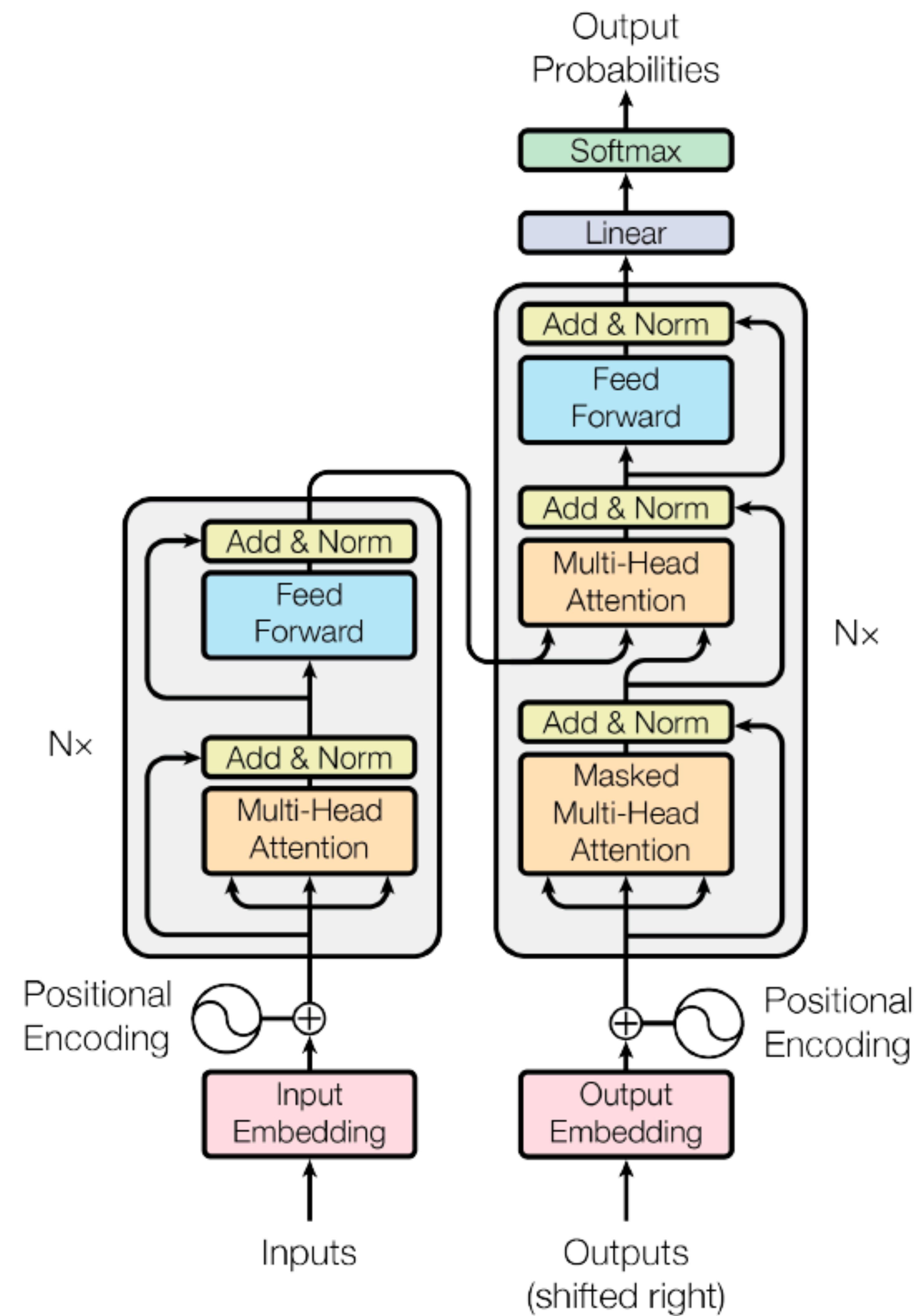
Cross Attention

- We saw that self -attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let $\mathbf{h}_1, \ldots, \mathbf{h}_n$ be output vectors from the Transformer encoder; $\mathbf{h}_i \in \mathbb{R}^d$
- Let $\mathbf{z}_1, \ldots, \mathbf{z}_n$ be input vectors from the Transformer decoder, $\mathbf{h}_i \in \mathbb{R}^d$
- Then keys and values are drawn from the encoder (like a memory):
  - $\mathbf{k}_i = \mathbf{K}\mathbf{h}_i, \mathbf{v}_i = \mathbf{V}\mathbf{h}_i$
- And the queries are drawn from the decoder, $\mathbf{q}_i = \mathbf{Q}\mathbf{z}_i$



USC Viterbi

Transformer Diagram



Attention is all you need (Vaswani et al., 2017)

Transformers: Performance

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 [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Language Modeling

Model	Test perplexity	ROUGE-L
seq2seq-attention, $L = 500$	5.04952	12.7
Transformer-ED, $L = 500$	2.46645	34.2
Transformer-D, $L = 4000$	2.22216	33.6
Transformer-DMCA, no MoE-layer, $L = 11000$	2.05159	36.2
Transformer-DMCA, MoE-128, $L = 11000$	1.92871	37.9
Transformer-DMCA, MoE-256, $L = 7500$	1.90325	38.8

The real power of Transformers comes from pretraining language models which are then adapted for different tasks

After Spring Break!