ECE7115 ~~Multimodal VLM~~ LLM

2. Transformer

Spring 2026

Namhyuk Ahn, Inha University

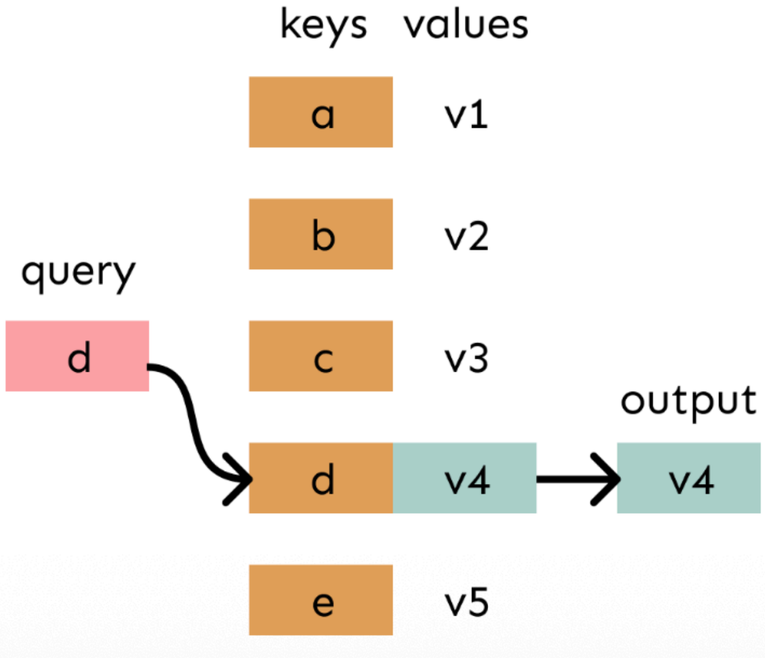
Lecture Overview

- Attention
  - Cross-Attention and Self-Attention
- Tokenizer
- Transformer
  - Model Architecture of the Original Transformer

Attention

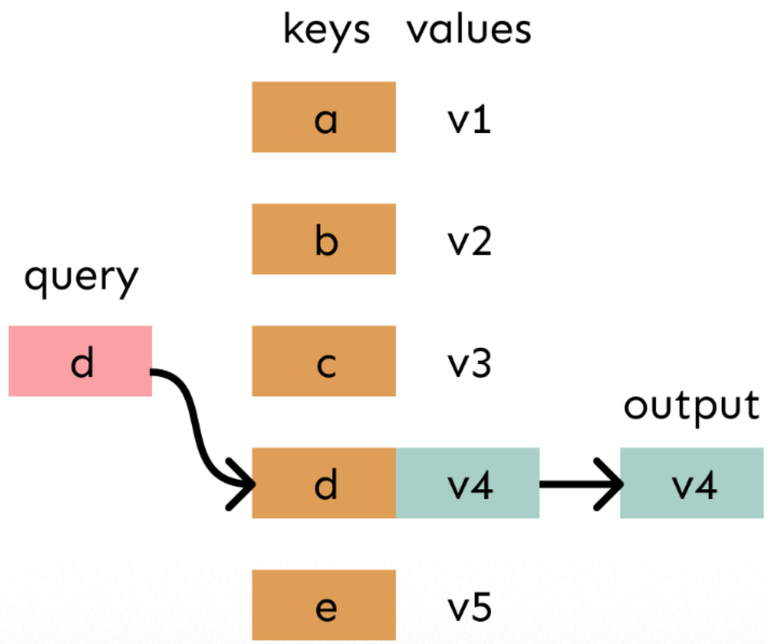
Attention [Bahdanau+ 2015]

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

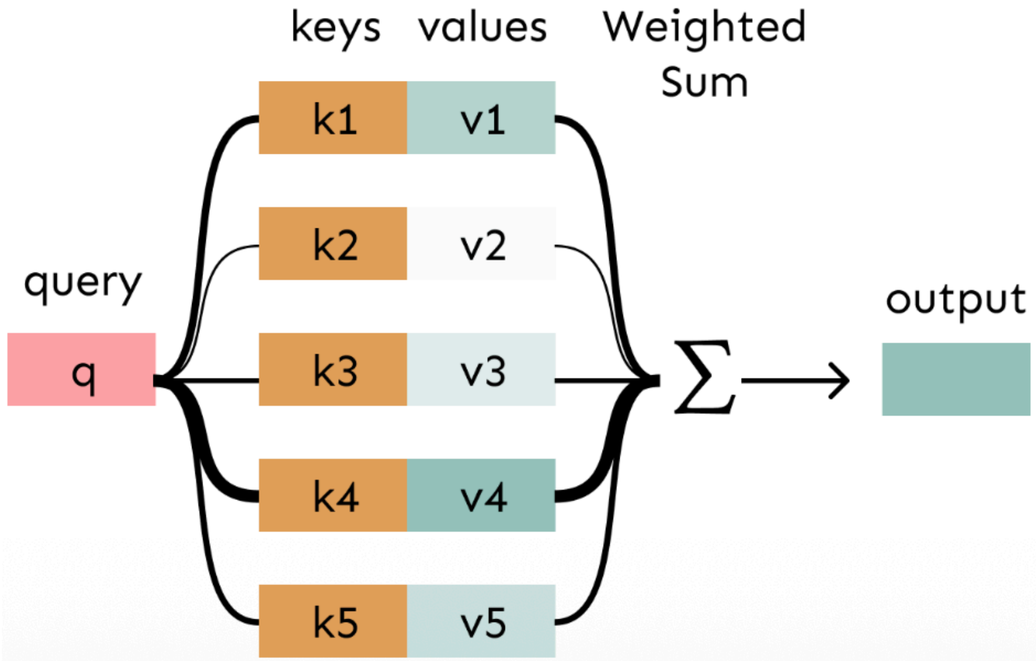


Attention [Bahdanau+ 2015]

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



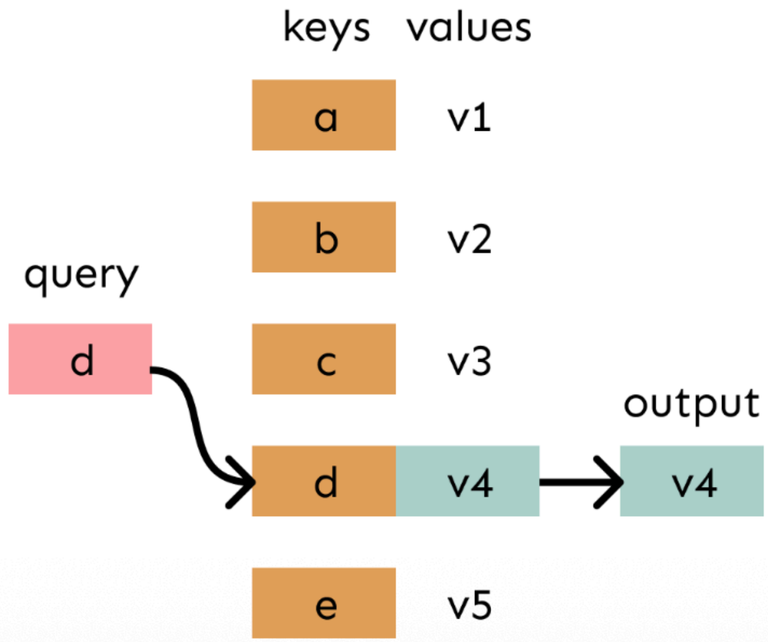
- In attention, the query matches all keys softly, to a weight [0, 1]
- The keys' values are multiplied by the weights and summed



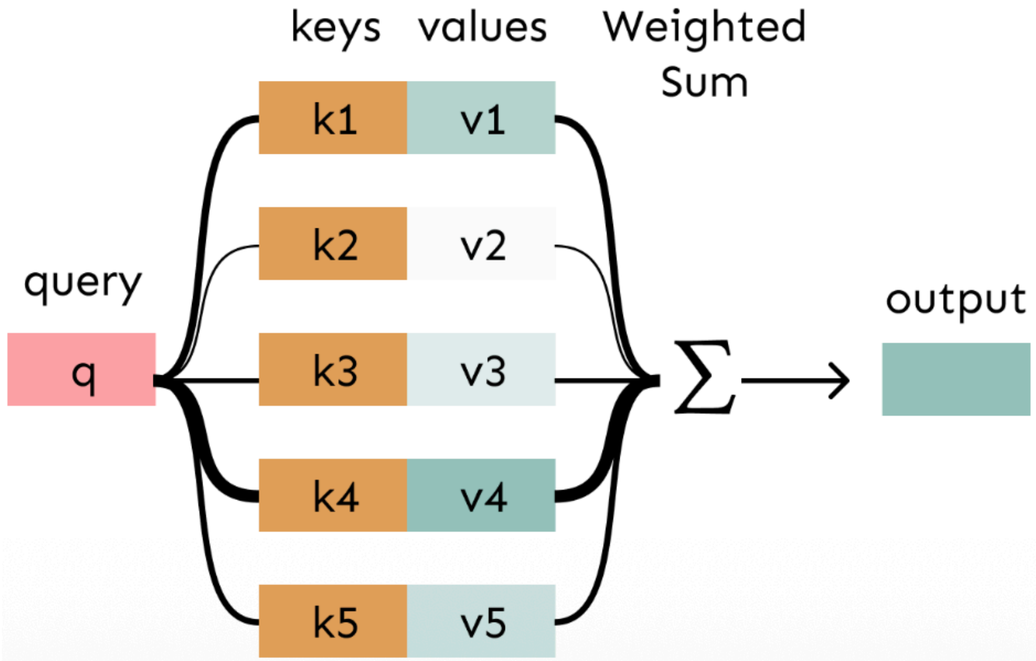
Attention [Bahdanau+ 2015]

Attention is just a soft lookup table with a weighted average!

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

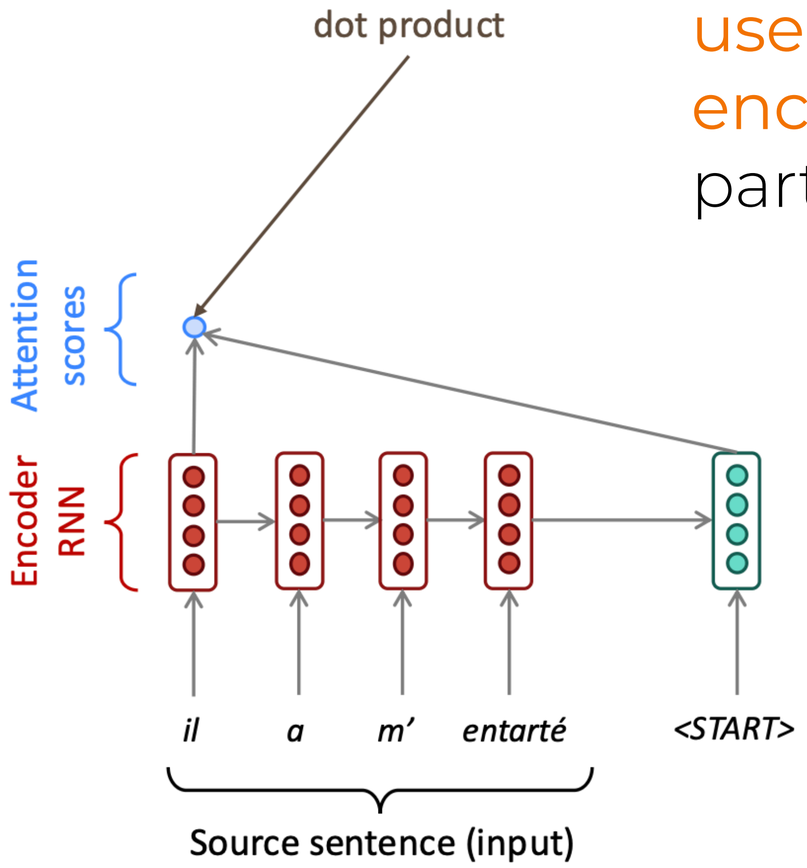


- In attention, the query matches all keys softly, to a weight [0, 1]
- The keys' values are multiplied by the weights and summed



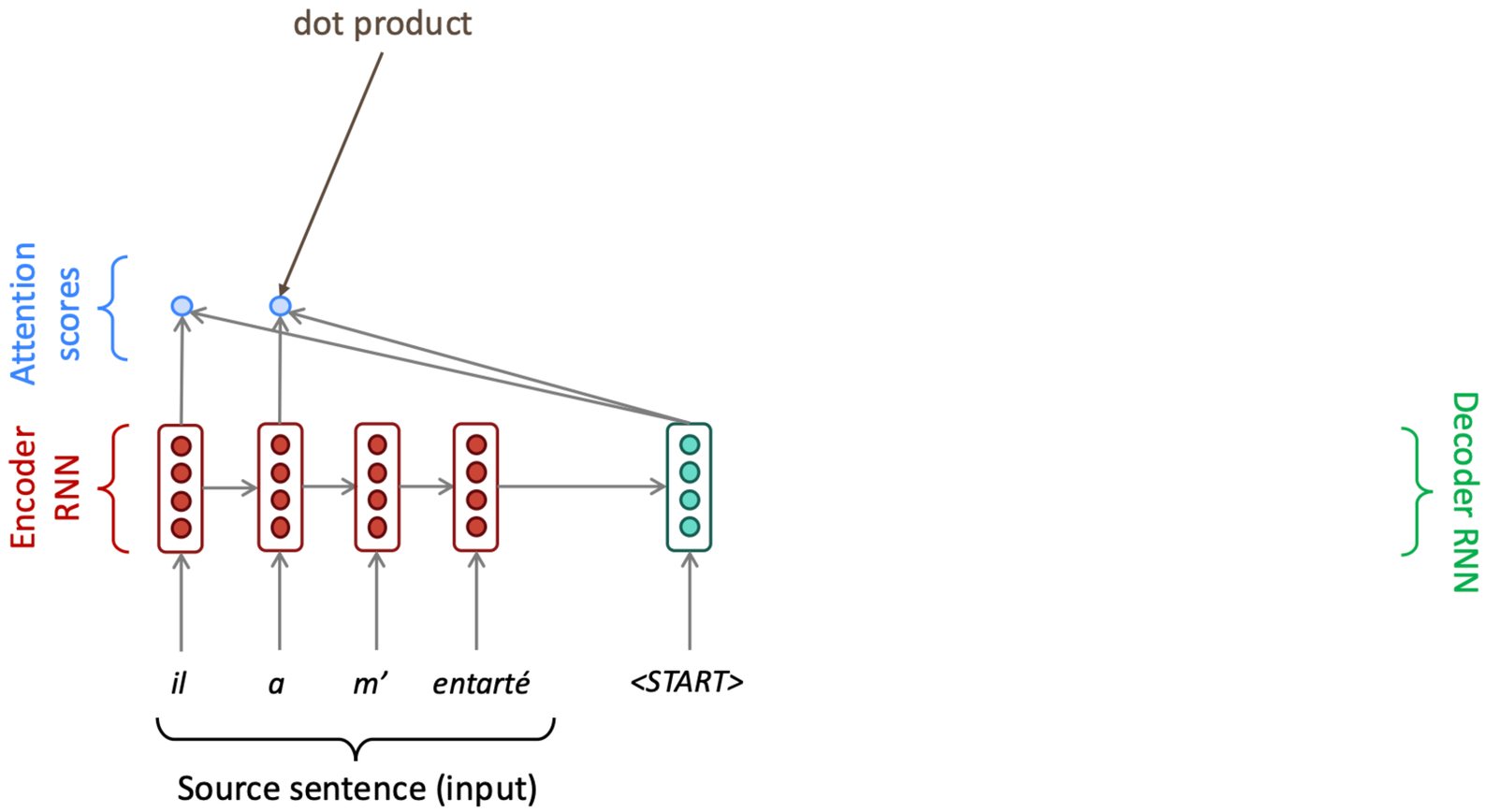
Note: Sequence-to-Sequence (seq2seq)

Seq2Seq with Attention

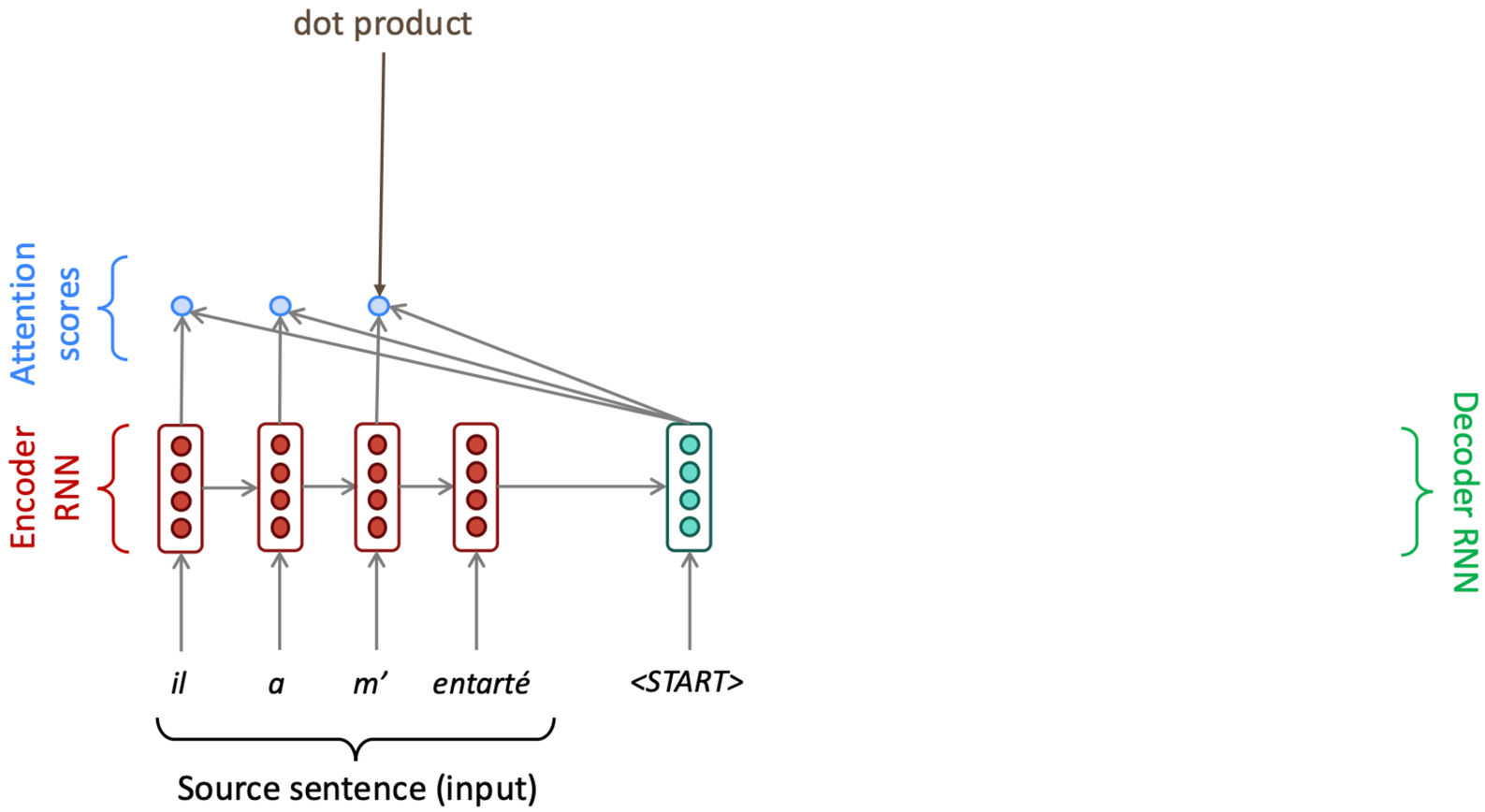


Idea: on each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence

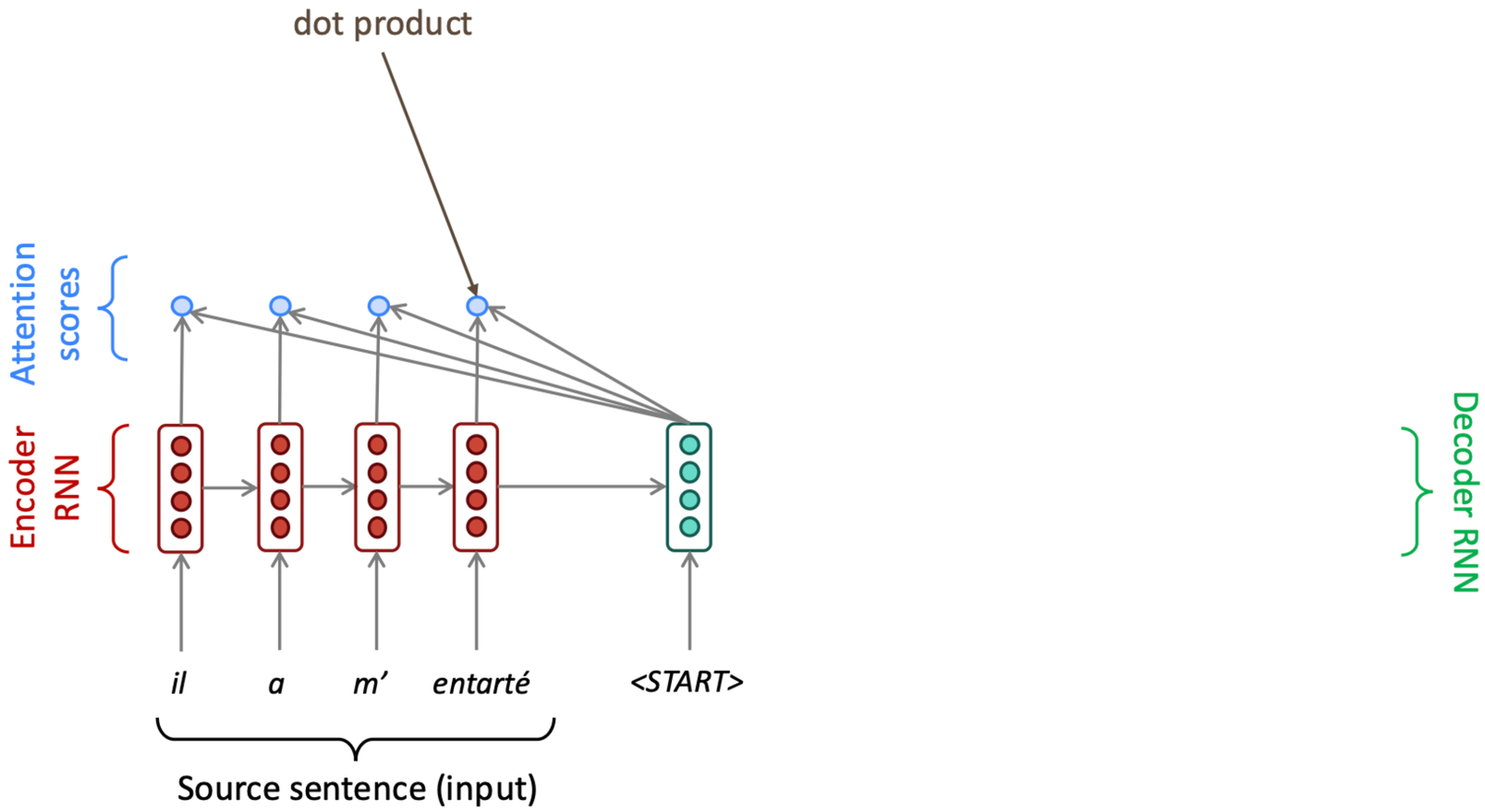
Seq2Seq with Attention



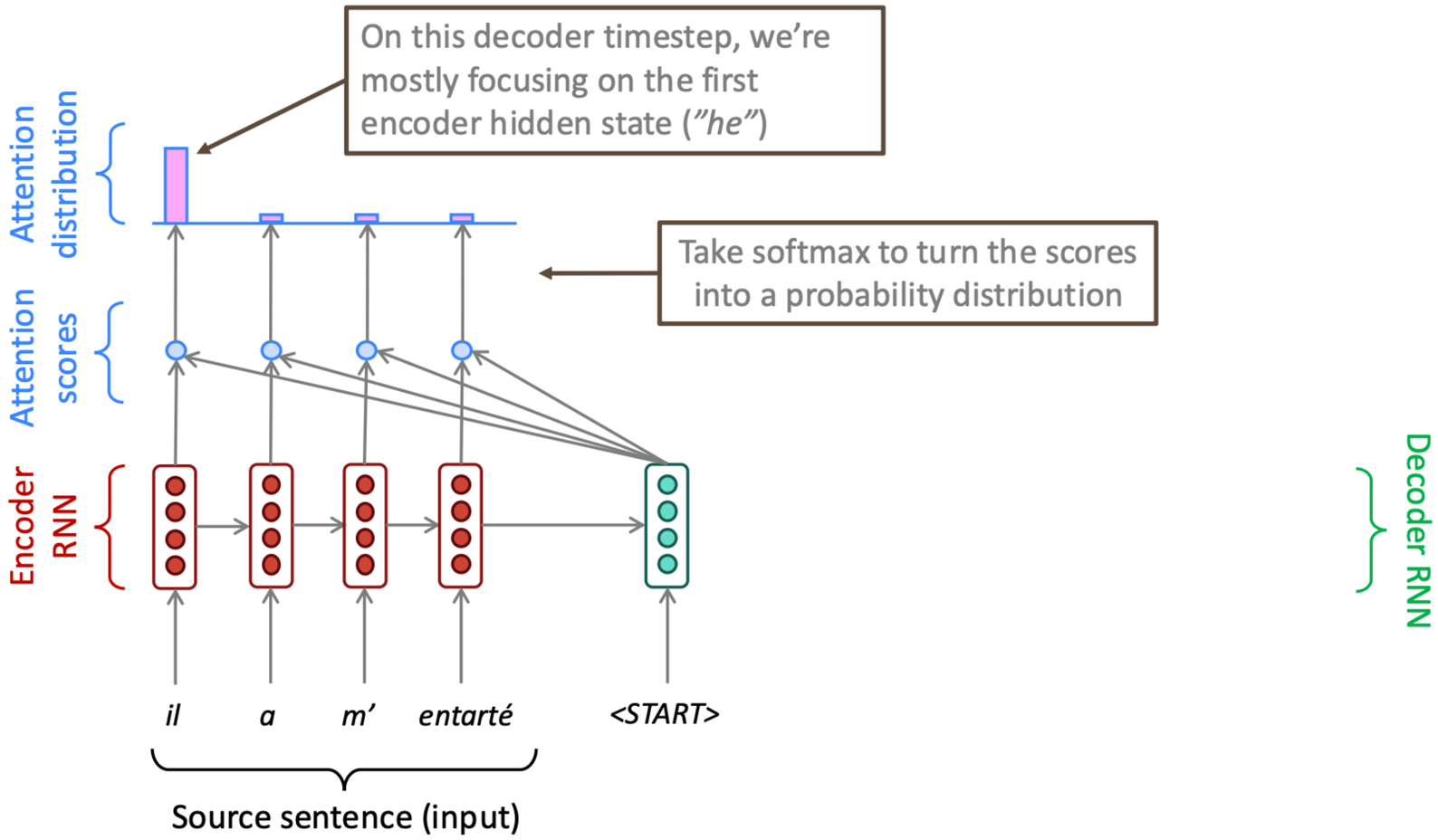
Seq2Seq with Attention



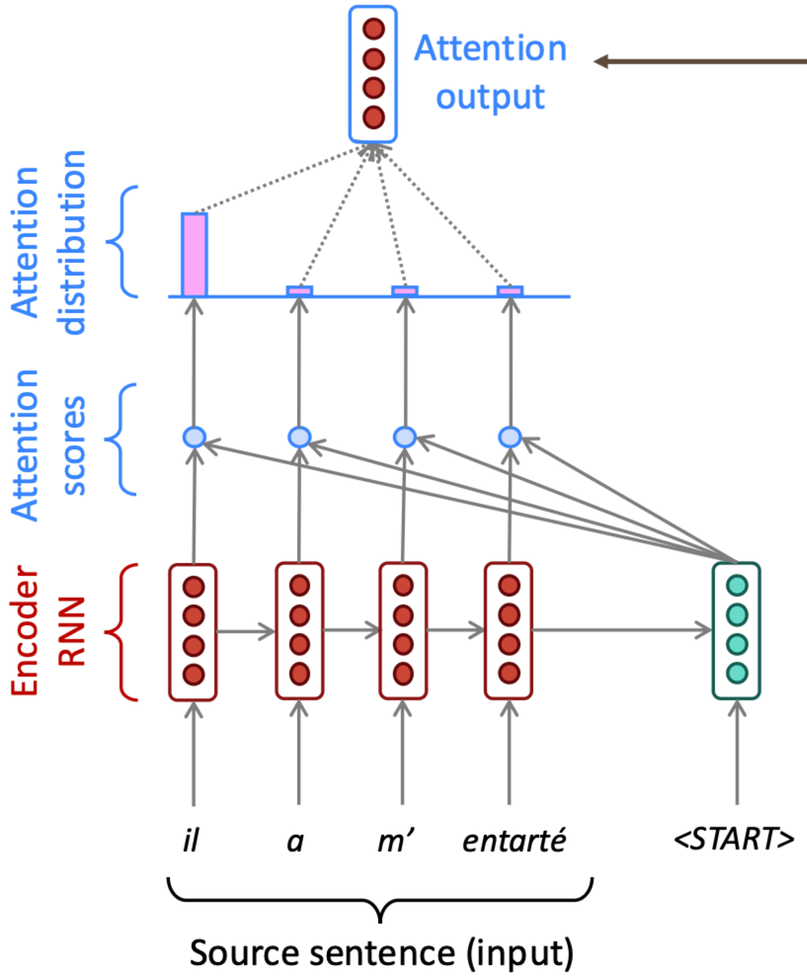
Seq2Seq with Attention



Seq2Seq with Attention



Seq2Seq with Attention

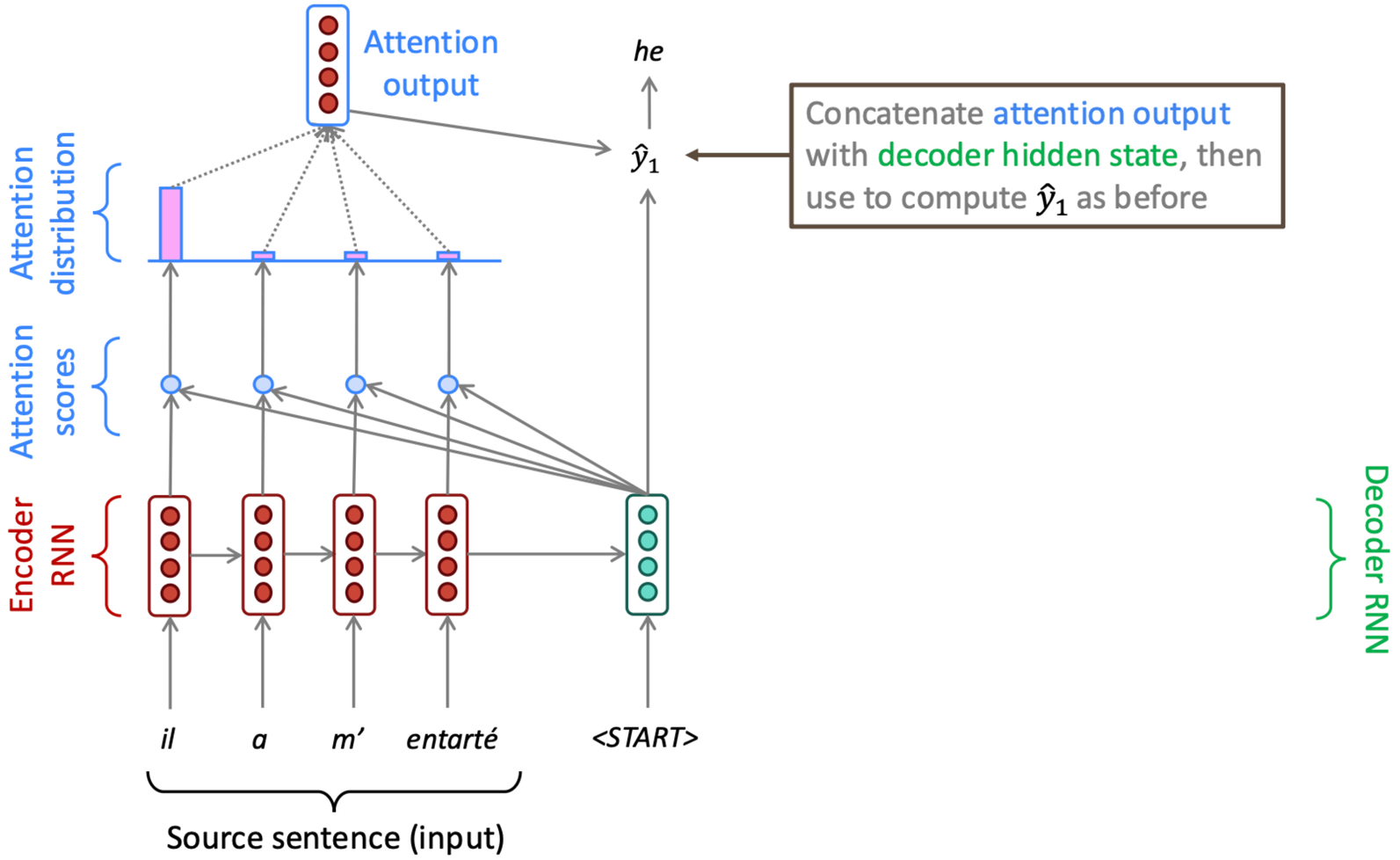


Use the attention distribution to take a **weighted sum** of the encoder hidden states.

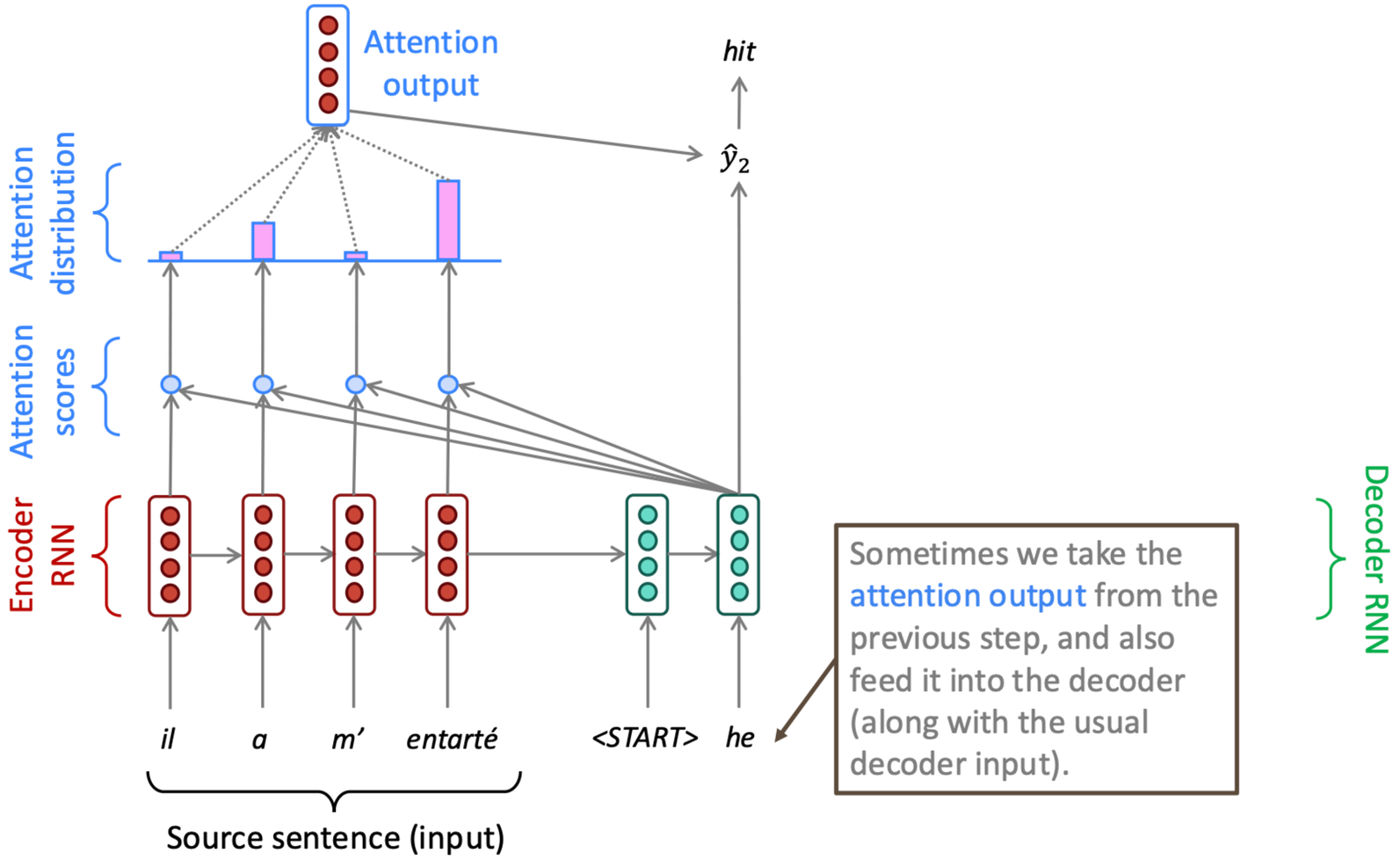
The attention output mostly contains information from the hidden states that received high attention.

Decoder RNN

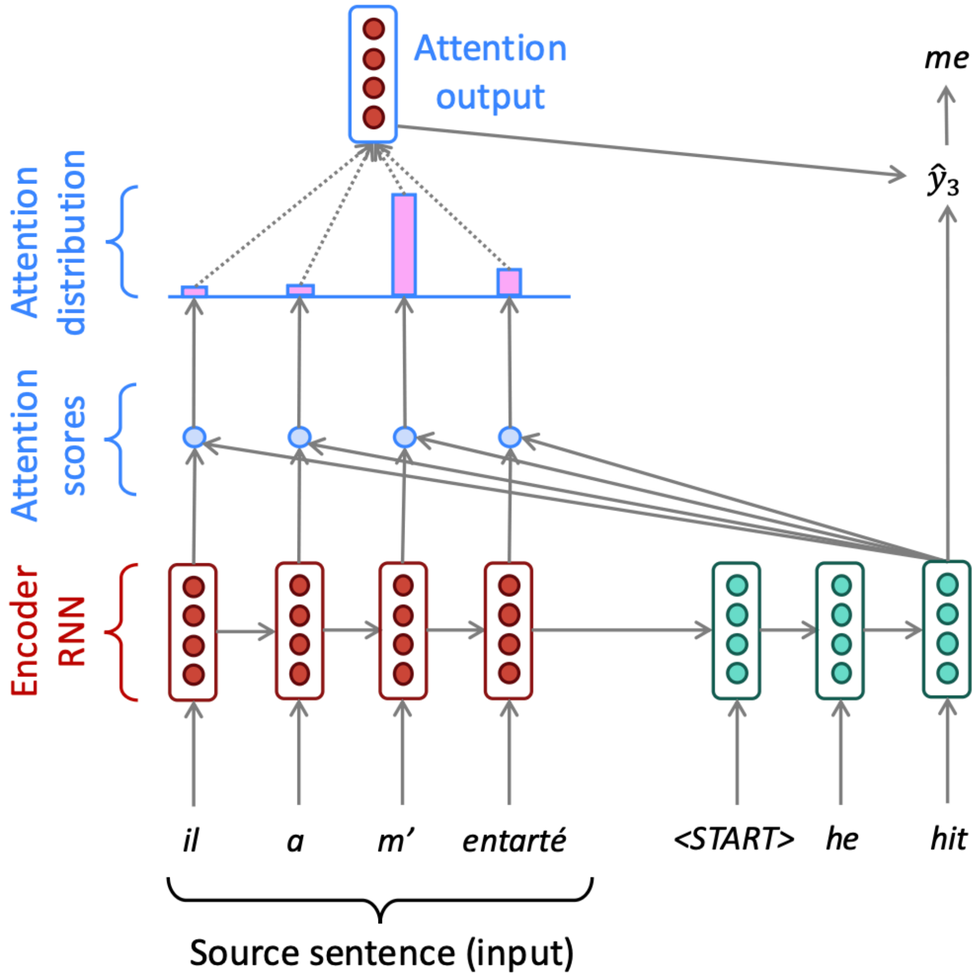
Seq2Seq with Attention



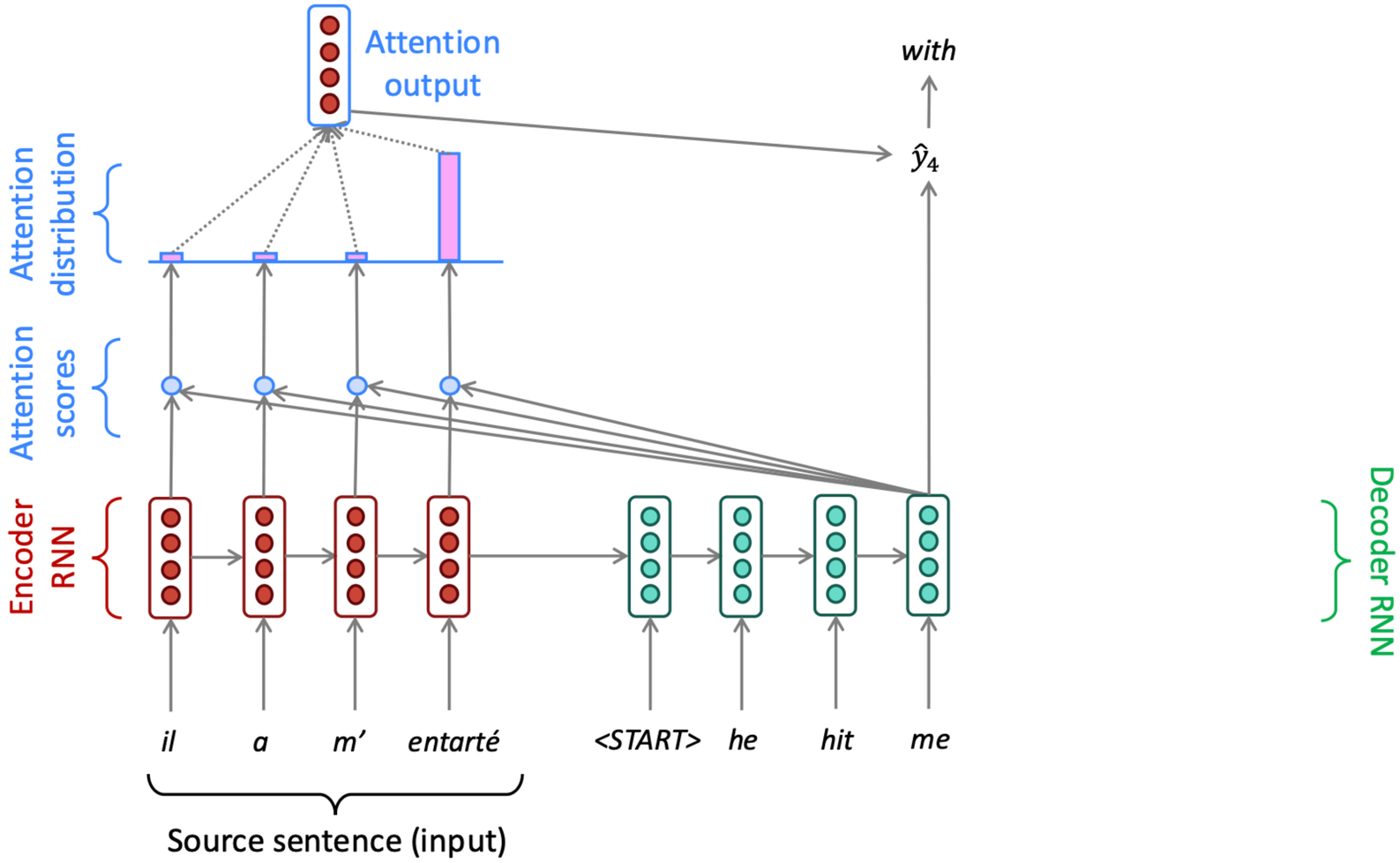
Seq2Seq with Attention



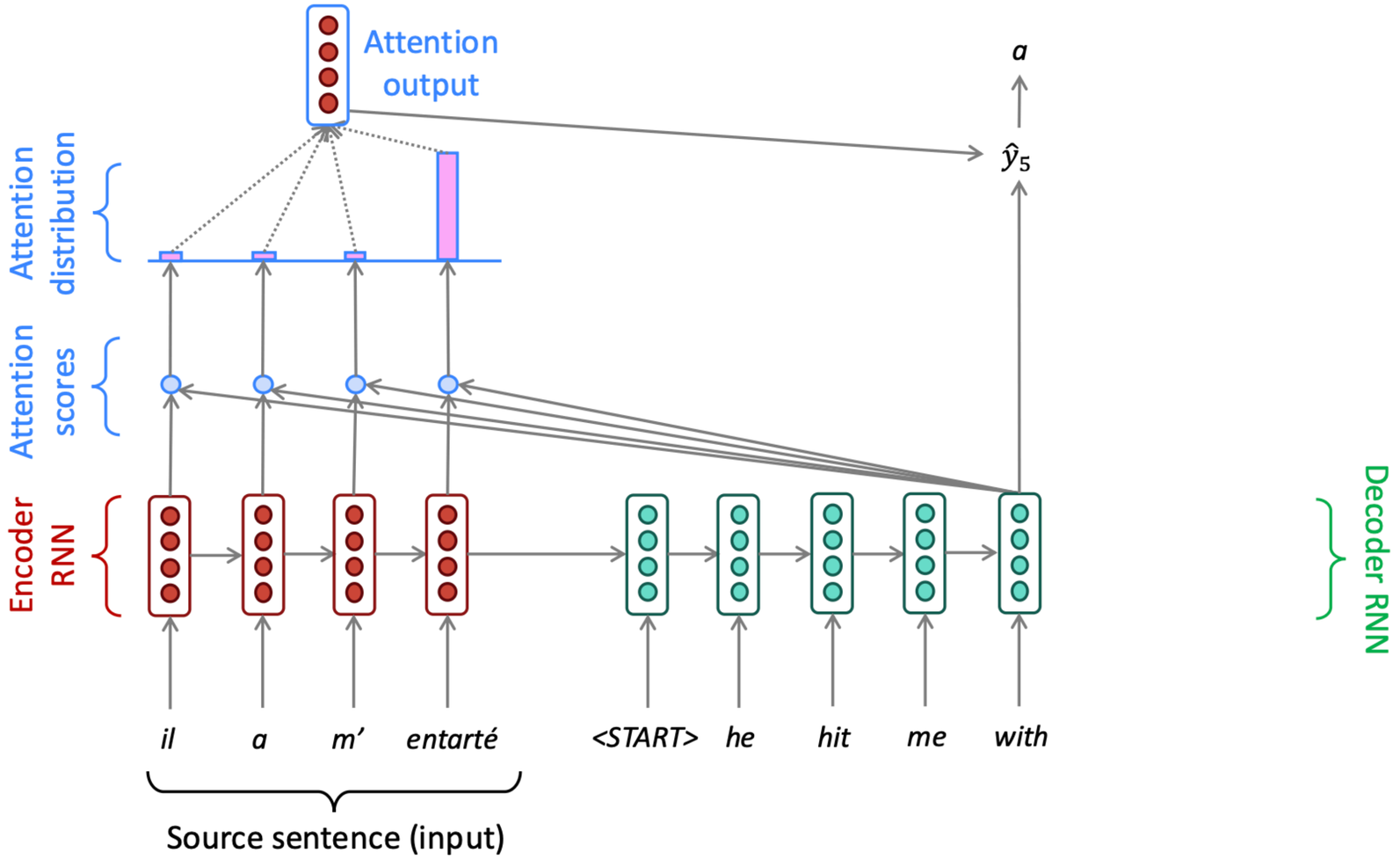
Seq2Seq with Attention



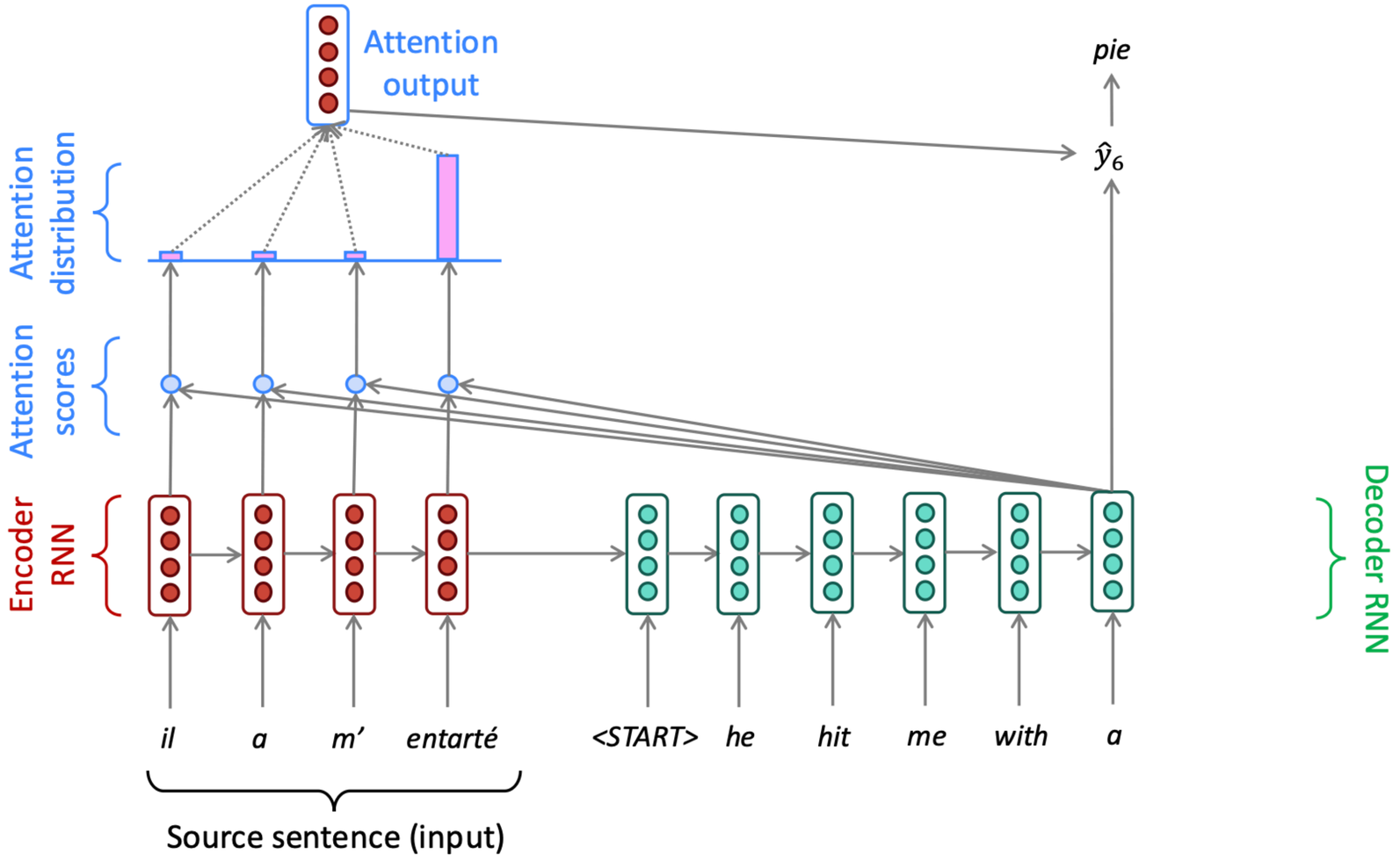
Seq2Seq with Attention



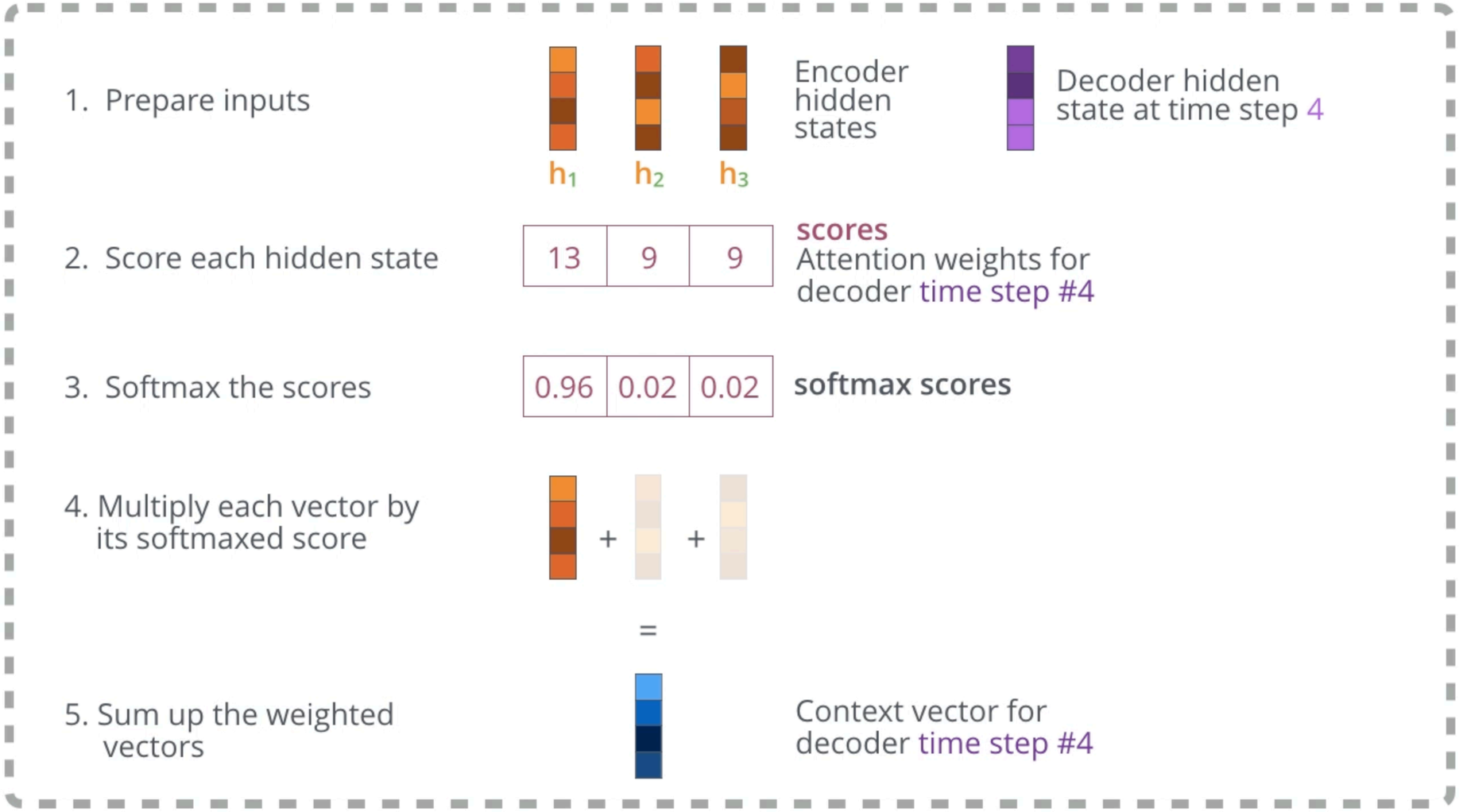
Seq2Seq with Attention



Seq2Seq with Attention

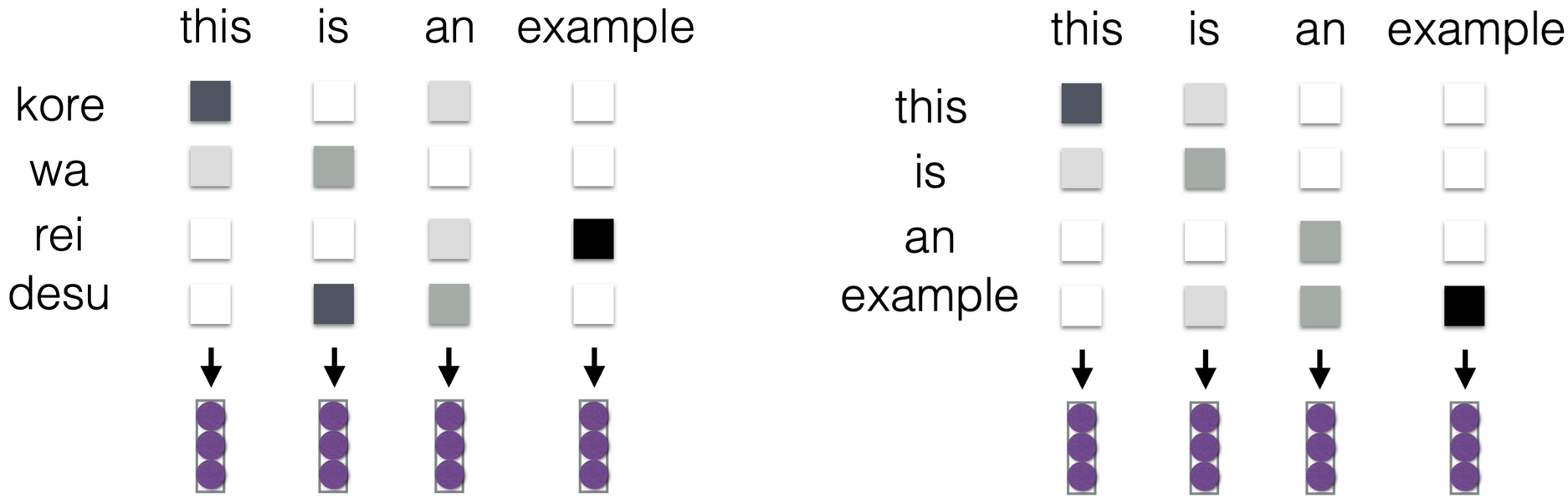


Summary So Far



Self-Attention [Cheng+ 2016, Vaswani+ 2017]

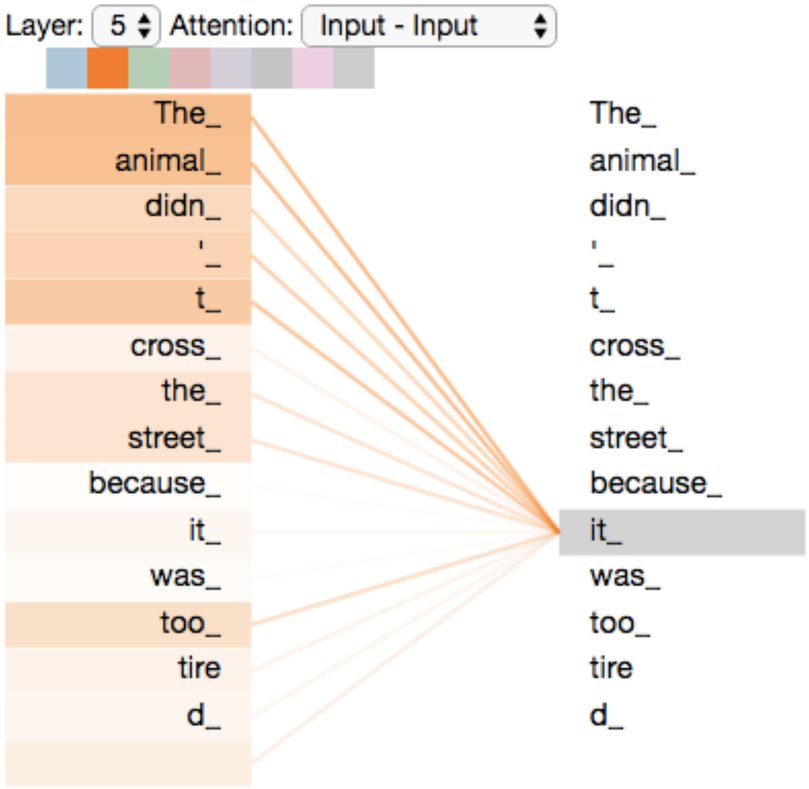
- What we talked about: Cross-attention
  - Paying attention to the input x to generate y_t
- Another type of attention: Self-attention
  - To generate y_t, we pay attention to $y_{<t}$



Self-Attention [Cheng+ 2016, Vaswani+ 2017]

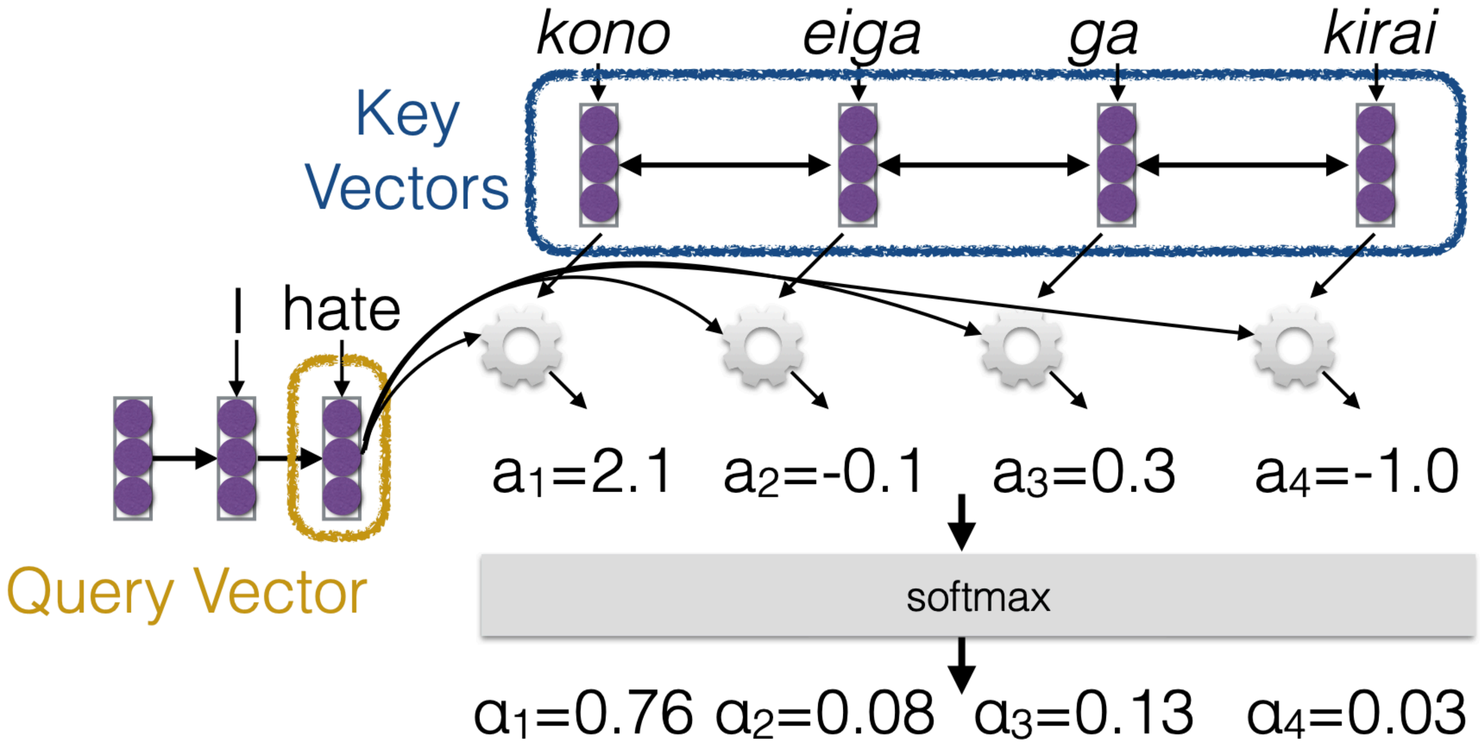
- Why does attending to itself matter?
  - e.g. "The animal didn’t cross the street because it was too tired"
  - What does "it" refer to?

- Self-attention can see relevant context within itself



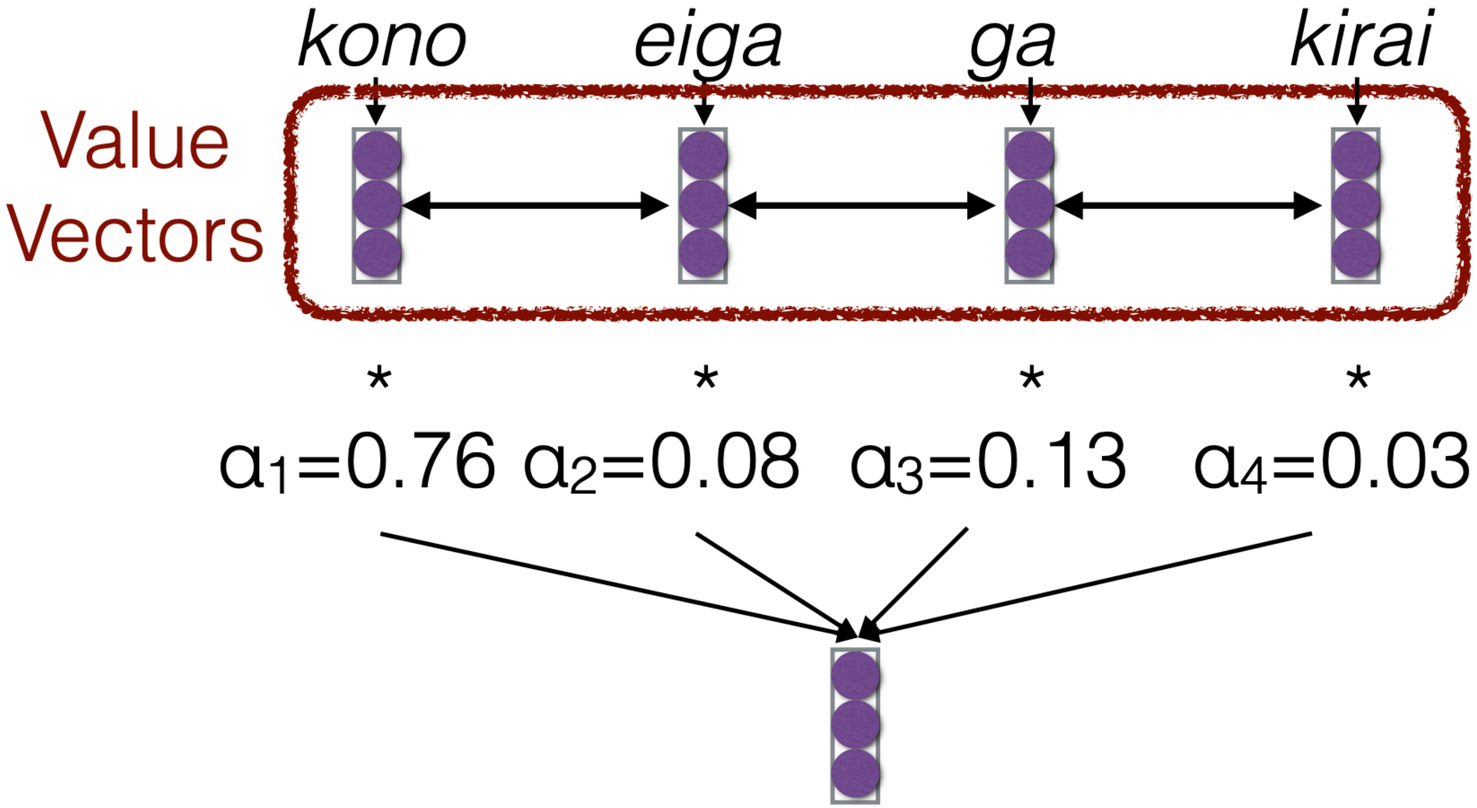
Calculating (Cross-)Attention

- Use query vector (decoder state) and key vectors (all encoder states)
- For each query-key pair, calculate weight
- Normalize attention weights to add to one using Softmax



Calculating (Cross-)Attention

- Combine value vectors (usually encoder states, same as the key vector) together by taking the weighted sum



Attention Score Function

- Notation: $\boldsymbol{q}$ is the query, and $\boldsymbol{k}$ is the key
- Multi-Layer Perceptron [Bahdanau+ 2015]

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{w}_2^\intercal \tanh(W_1[\boldsymbol{q}; \boldsymbol{k}])$$

  - Flexible, often very good with a large dataset

- Bilinear [Luong+ 2015]

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^\intercal W \boldsymbol{k}$$

Attention Score Function

- Notation: q is the query, and k is the key
- Dot Product [Luong+ 2015]

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^\intercal \boldsymbol{k}$$

  - No parameters in Attention! But requires sizes to be the same
  - Problem: scale of the dot product increases as dimensions get larger

- Scaled Dot Product [Vaswani+ 2017]
  - Fix: scale by size of the vector
  - (In practice, # elements of k)

$$a(\boldsymbol{q}, \boldsymbol{k}) = \frac{\boldsymbol{q}^\intercal \boldsymbol{k}}{\sqrt{|\boldsymbol{k}|}}$$

Summary: Calculating Self-Attention

- Let $w_{1:n}$ be a sequence of words in vocabulary V
- For each w_i , let $x_i = Ew_i$, where $E \in \mathbb{R}^{d \times |V|}$ is an embedding matrix

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

$q_i = Qx_i$ (queries) $k_i = Kx_i$ (keys) $v_i = Vx_i$ (values)

- 2. Compute pairwise similarities (here, using dot product) between key and queries. Normalize with softmax

$$e_{ij} = q_i^\top k_j \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

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

$$o_i = \sum_j \alpha_{ij} v_j$$

Transformer

"Attention is All You Need"

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

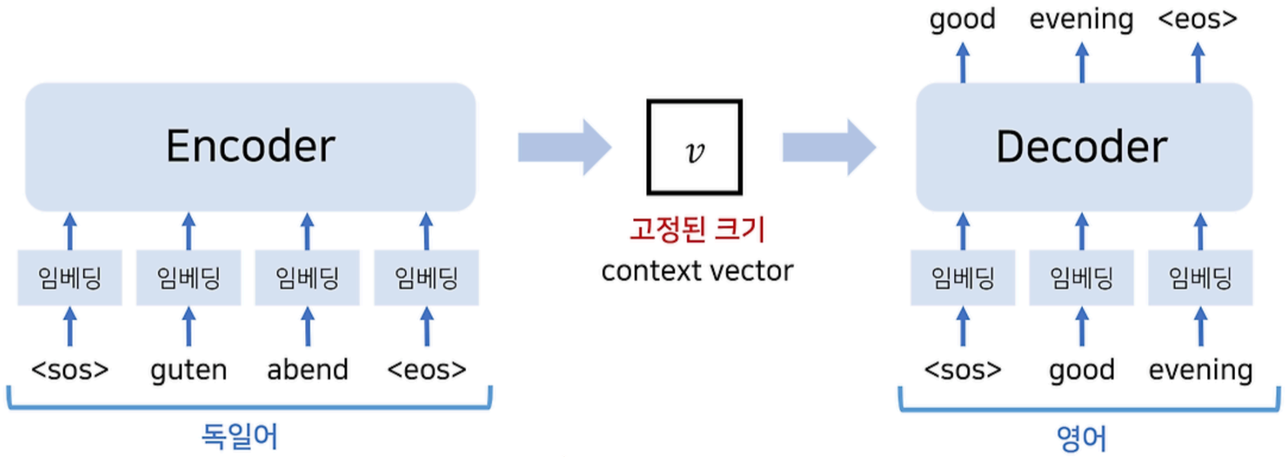
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

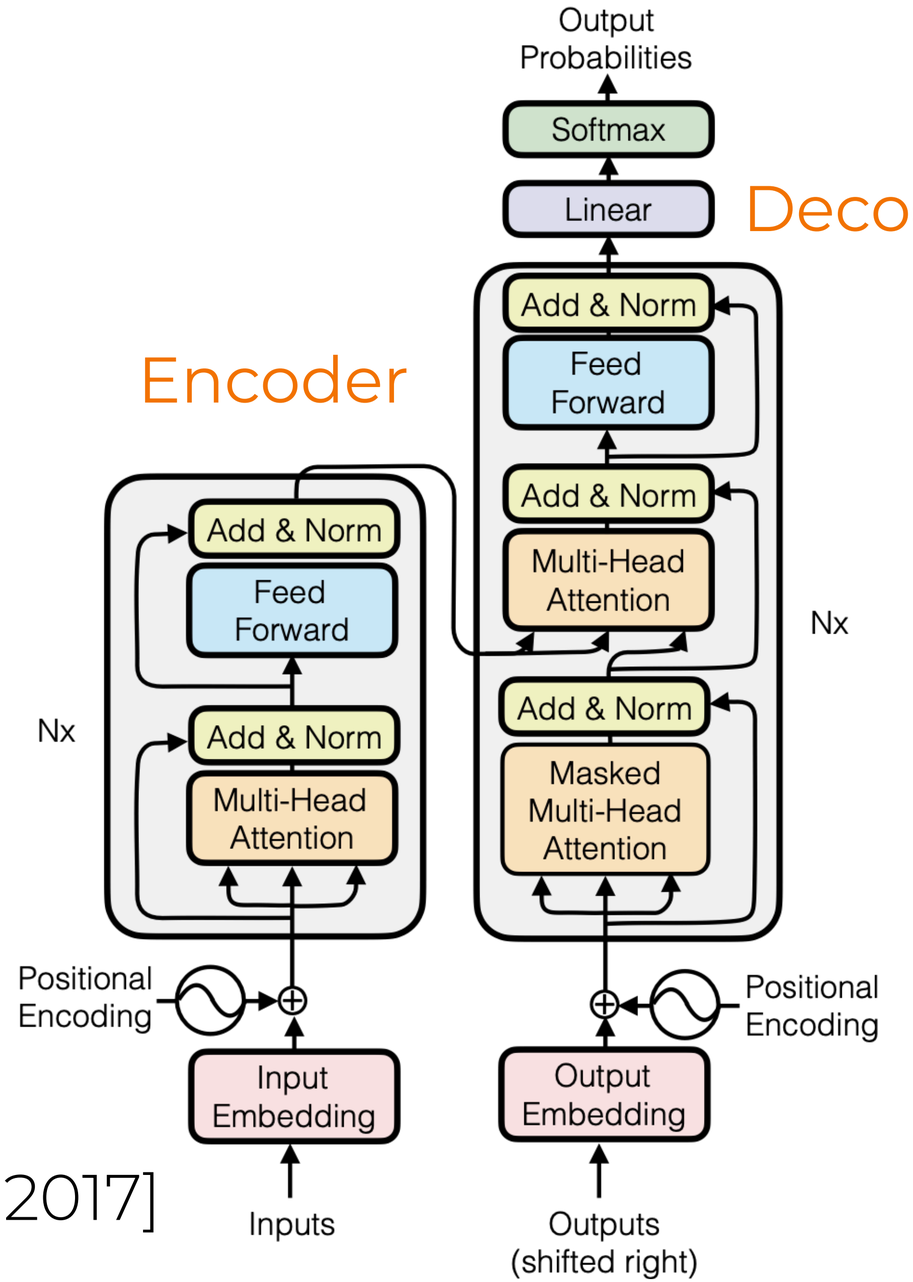
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

"Attention is All You Need"

- A sequence-to-sequence model based entirely on attention
- Strong results on machine translation
- Fast: Only matrix multiplications



previous seq2seq



[Vaswani+ 2017]

Core Transformer Modules

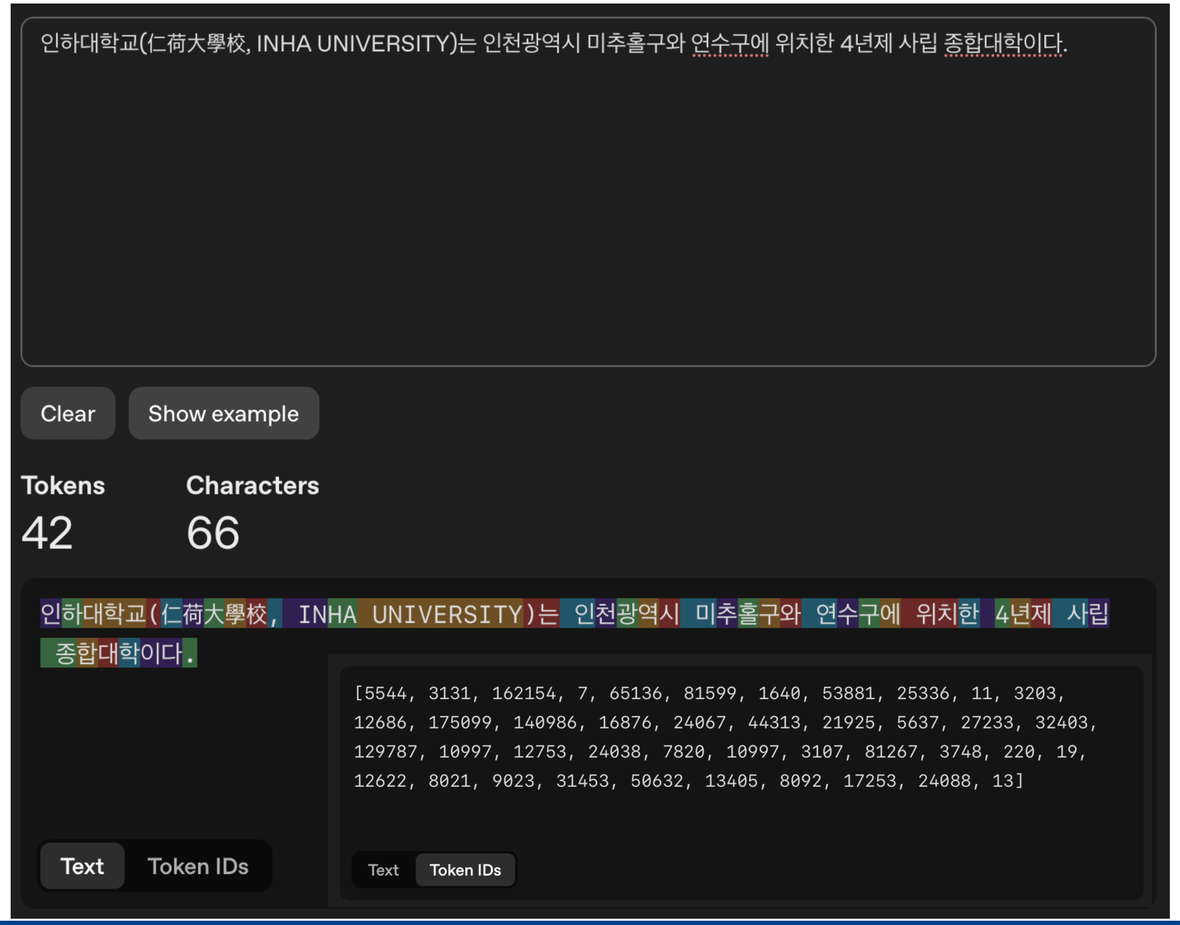
- (Tokenizer)
- Positional encodings
- Multi-headed attention
- Masked attention
- Residual + layer normalization
- Feed-forward network

Tokenizer

Tokenization

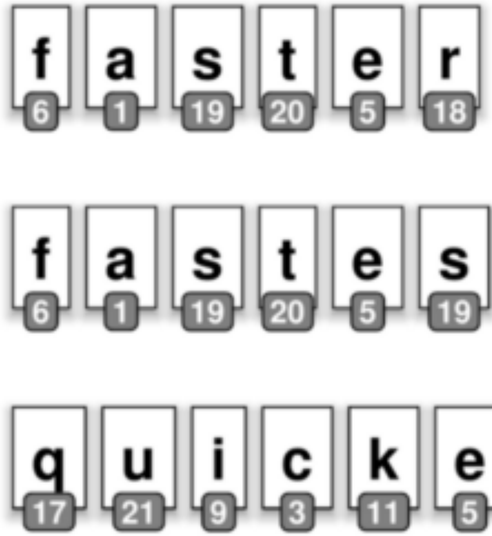
- Tokenizers convert between strings and sequences of integers (tokens)

Encode



Decode

Character-Level Tokenization



- Each character can be converted into a code
  - e.g. ord("a") is 97, ord("🌍") is 127757

- Very simple, but there are ~150K Unicode characters
  - This is a very large vocabulary
  - Many characters are quite rare (e.g. 🌍), which is inefficient

Word-Level Tokenization

- Split strings into words, then assign index

- But there are problems:
  - # words is huge (like for Unicode characters)
  - Many words are rare and the model won't learn much about them
    - e.g. string = "I'll say supercalifragilisticexpialidocious!"
  - This doesn't obviously provide a fixed vocabulary size
    - New words we haven't seen during training get a special <UNK> token, which is ugly and can mess up perplexity calculations

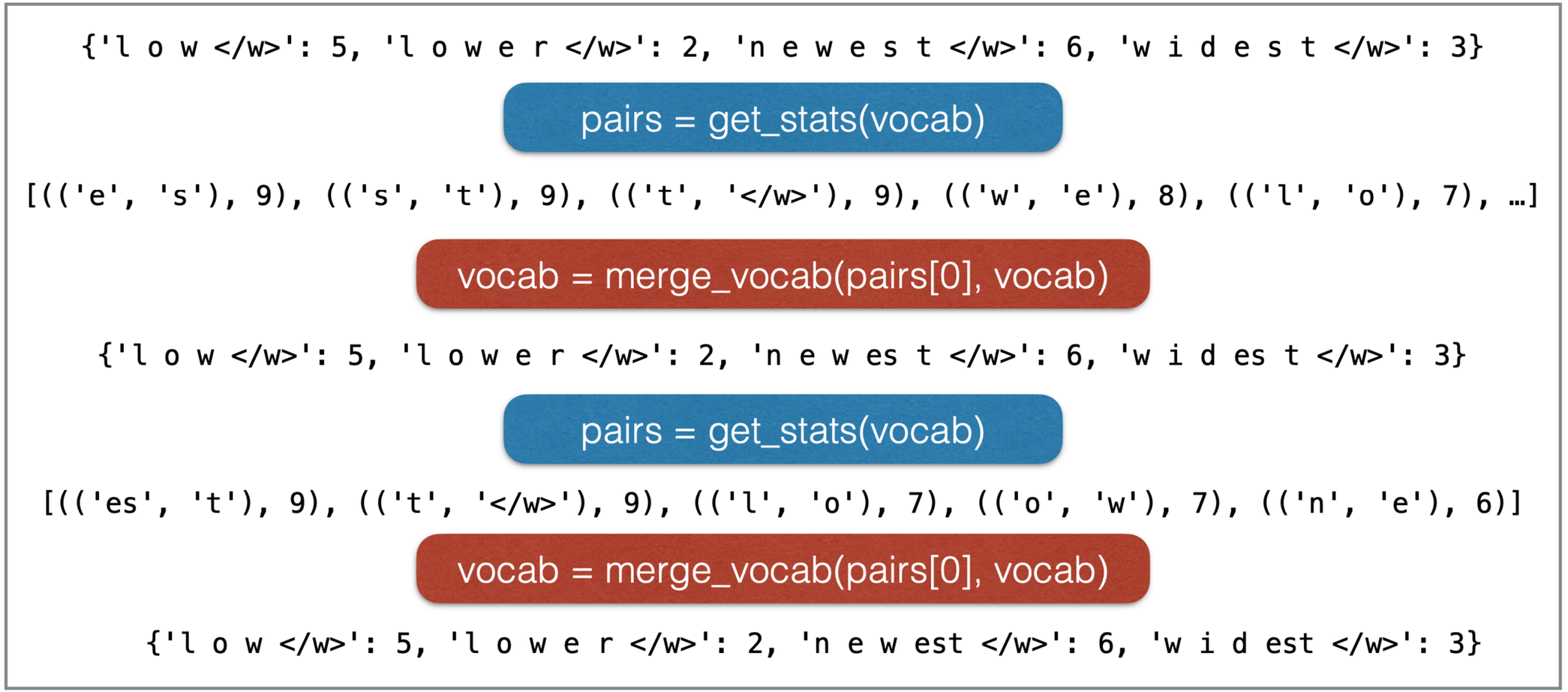
Byte Pair Encoding (BPE)

- The BPE algorithm was introduced in 1994 for data compression
  - It was adapted to NLP for NMT and then also used by GPT-2
  - BPE is subword-level tokenization

- Basic idea: train the tokenizer on raw text to automatically determine the vocabulary
- Intuition: common sequences of characters are represented by a single token, rare sequences are represented by many tokens
- Method: from byte, iteratively merges the most frequent "byte pair"

Byte Pair Encoding (BPE)

- Incrementally combine together the most frequent token pairs



Transformer: Positional Encoding

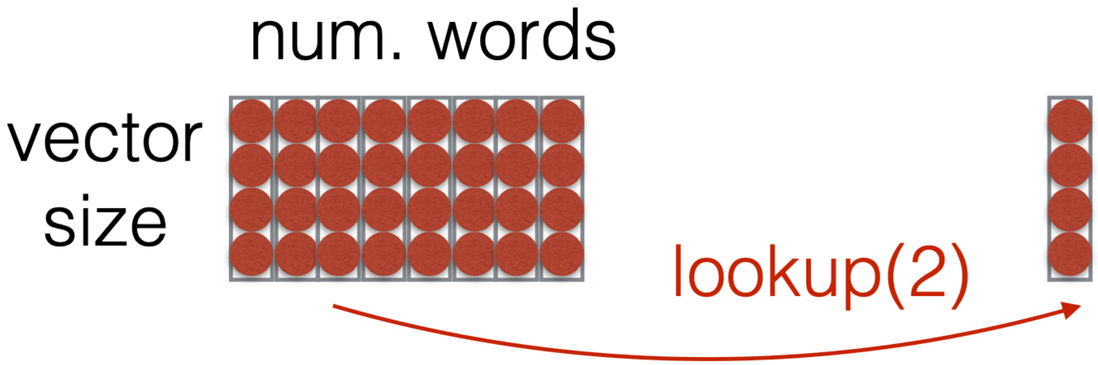
Recap: Inputs and Embeddings

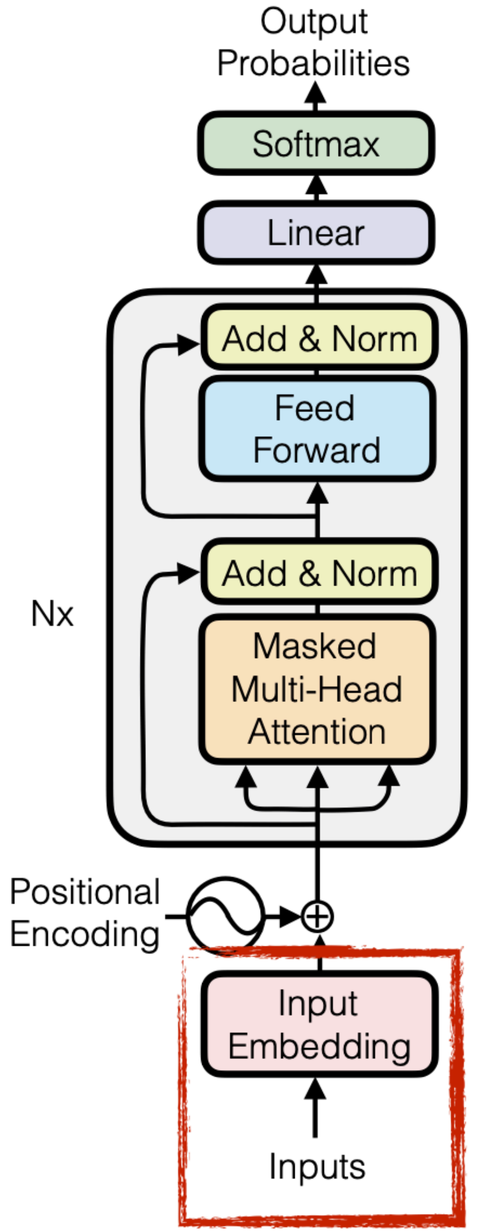
- Inputs: Generally split using subwords (tokenizer)
  - words → tokens

the books were improved

the book _s were **improv _ed**

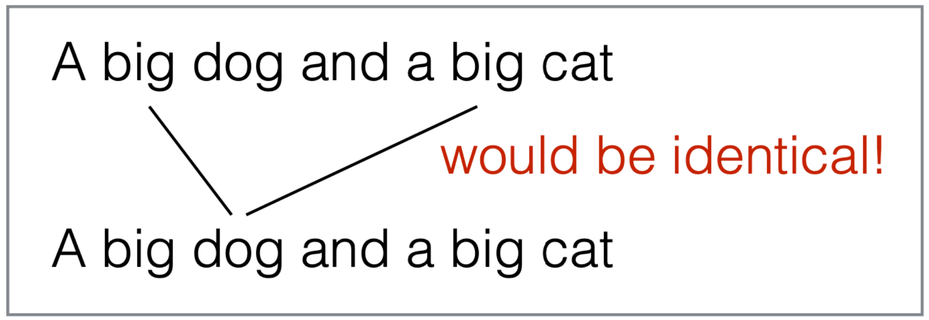
- Input embeddings: Looked up from input tokens
  - Token indices are converted to dense matrix





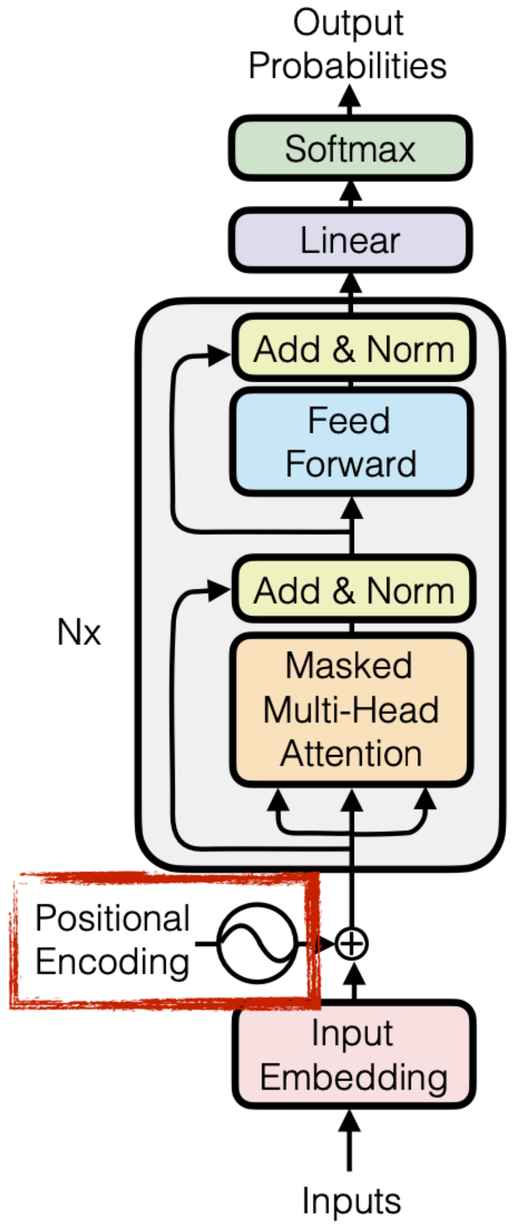
Positional Encoding

- The transformer model is *purely* attentional
  - There is no way to figure out sequential order
- If only embeddings were used, there would be no way to distinguish between identical words



- Positional encodings add a sequence order to a word embedding

$W_{big} + W_{pos2}$ $W_{big} + W_{pos6}$

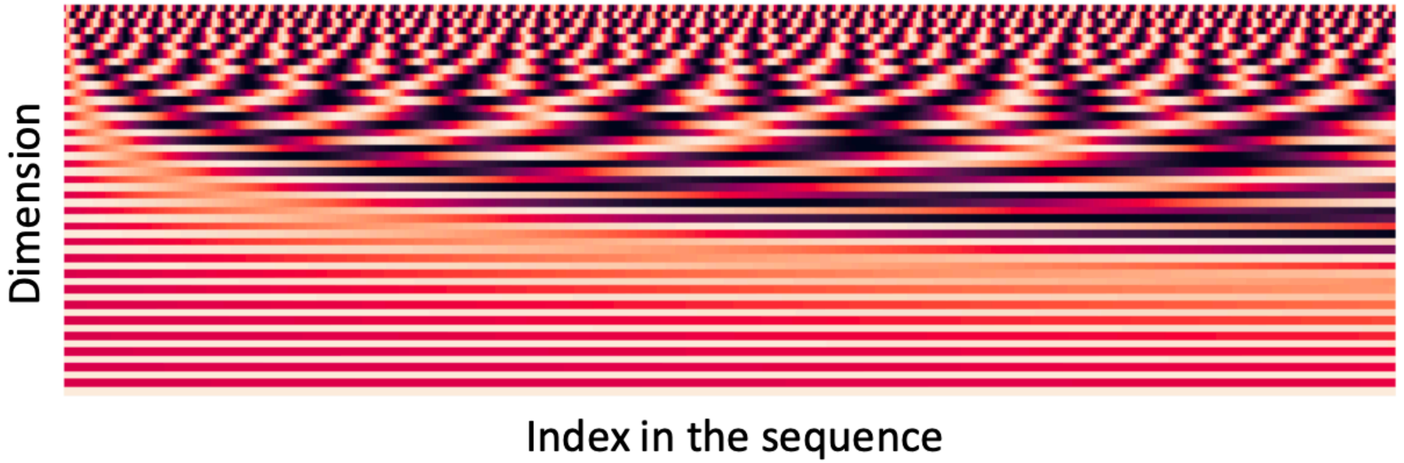


Sinusoidal Positional Encoding

[Vaswani+ 2017, Kazemnejad+ 2019]

- Calculate each dimension with a sinusoidal function of varying period

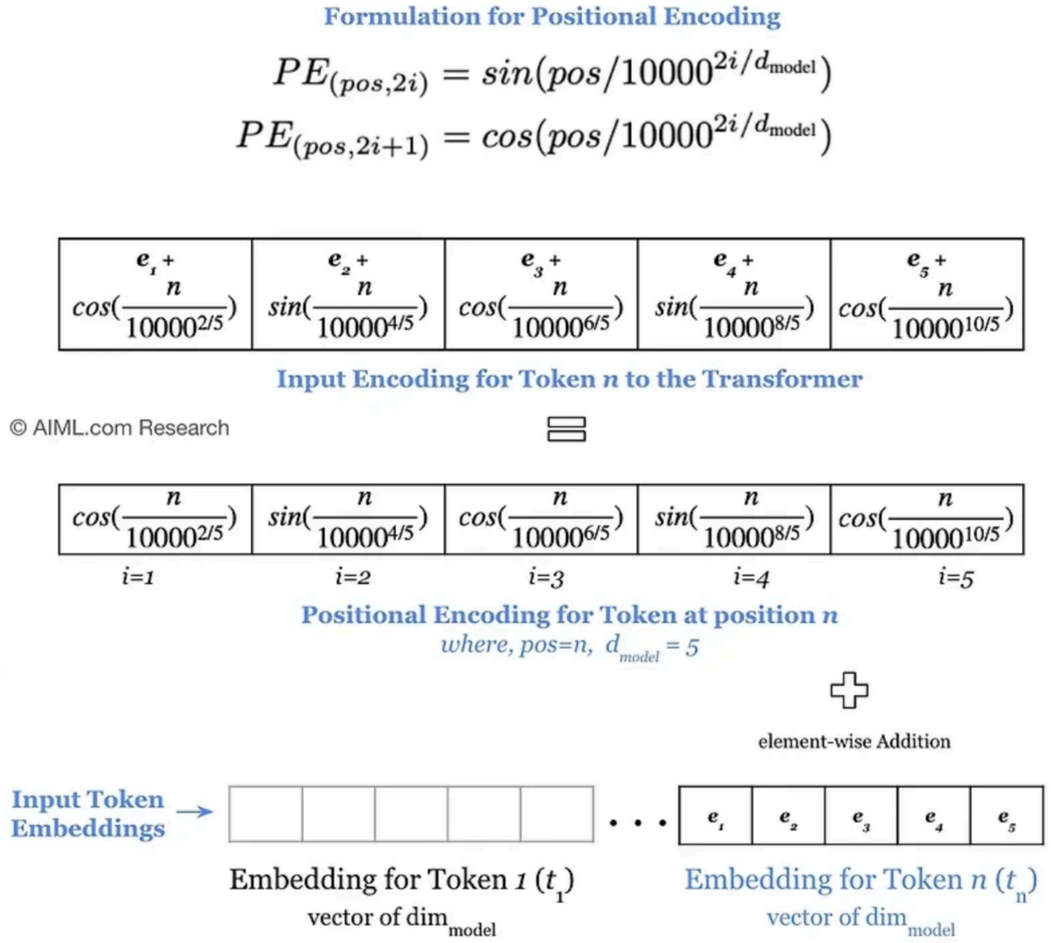
$$\boldsymbol{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



- 👍 Length generalization: (maybe) handle arbitrary length sequences
- 👍 Multi-scale:
  - High-frequency components are used to differentiate near tokens
  - Low-frequency components are used to grasp far-distance tokens

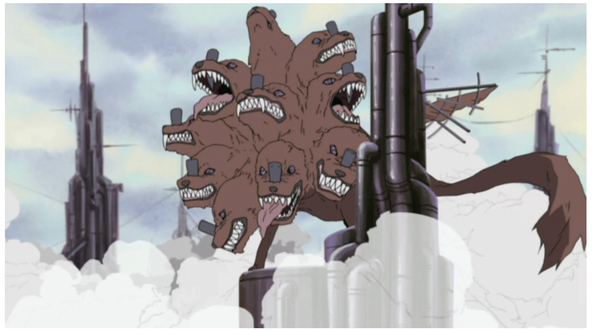
Sinusoidal Positional Encoding

[Vaswani+ 2017, Kazemnejad+ 2019]



Transformer: Multi-Head Attention

Why Multi-Head?



- Multi-head
  - Use multiple attention pathways (and outputs)

- Why? Information from different parts can be useful to disambiguate in different ways

I **run** a small business

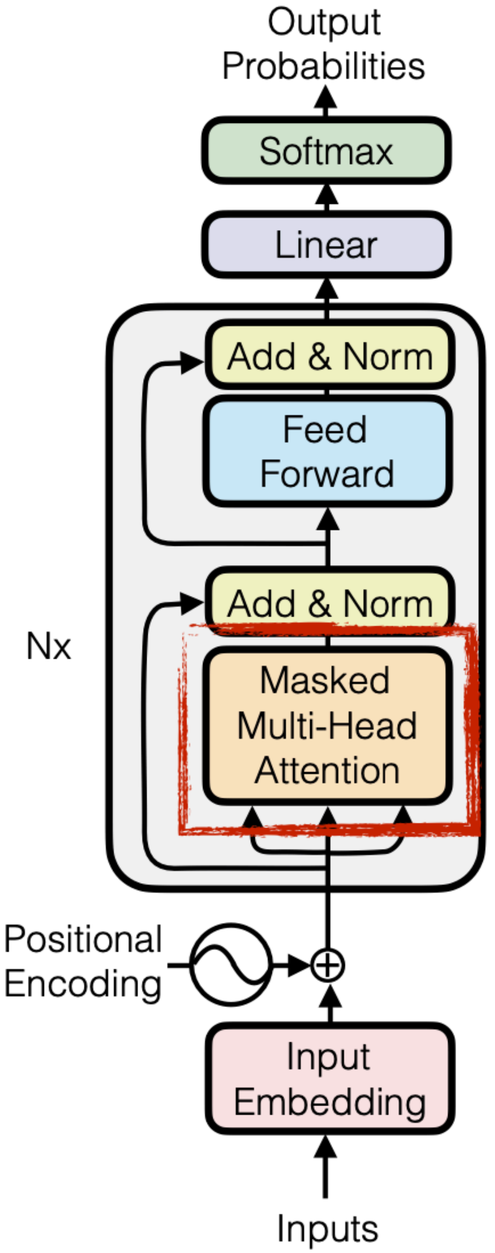
I **run** a mile in 10 minutes

The robber made a **run** for it

The stocking had a **run**

syntax
(nearby context)

semantics
(farther context)



Why Multi-Head?

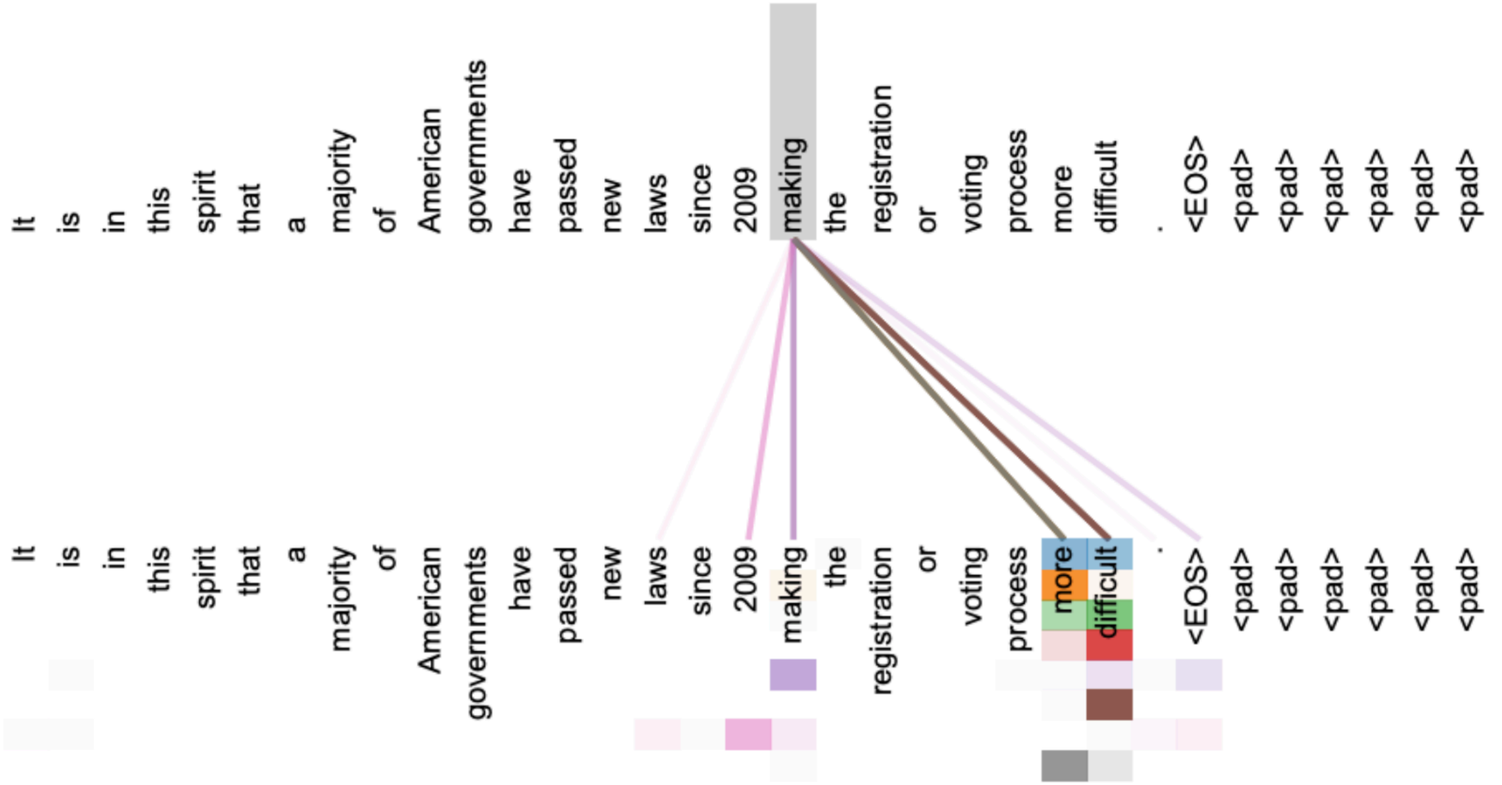
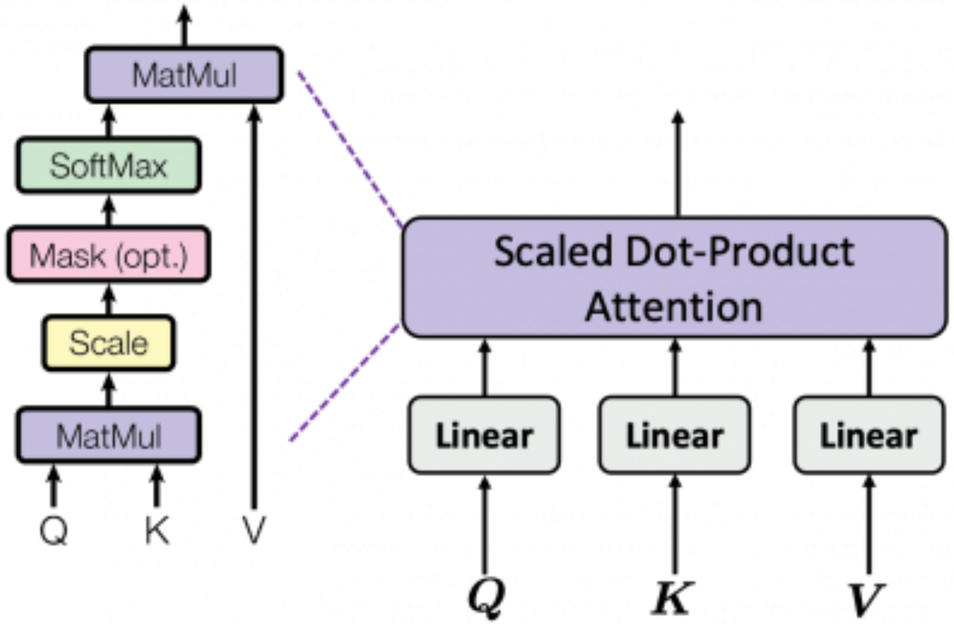
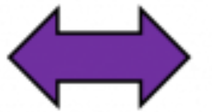


Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb 'making', completing the phrase 'making...more difficult'. Attentions here shown only for the word 'making'. Different colors represent different heads. Best viewed in color.

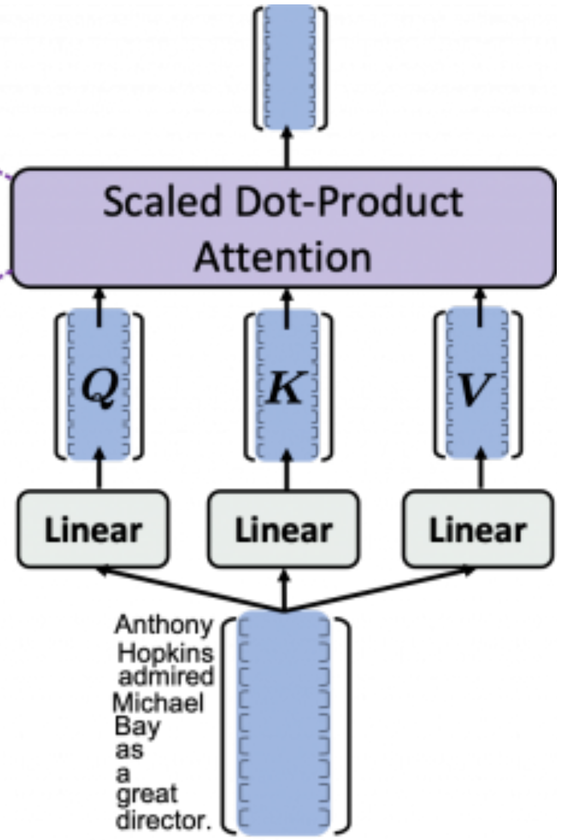
Scaled Dot-Product Attention



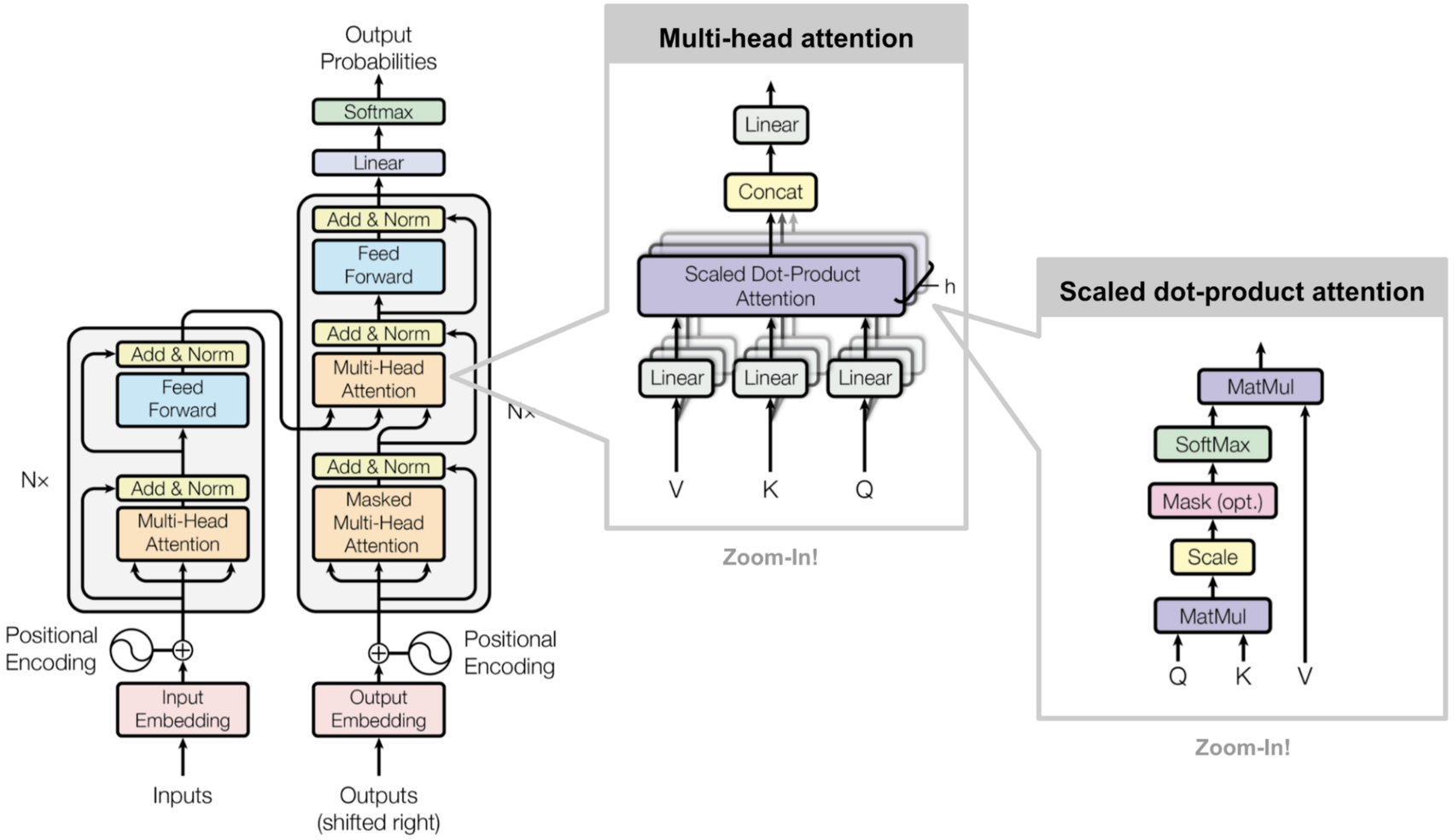
$$softmax\left(\frac{Q \times K}{\sqrt{d_k}}\right) \times V$$



Note: $a(\boldsymbol{q}, \boldsymbol{k}) = \dfrac{\boldsymbol{q}^\intercal \boldsymbol{k}}{\sqrt{|\boldsymbol{k}|}}$



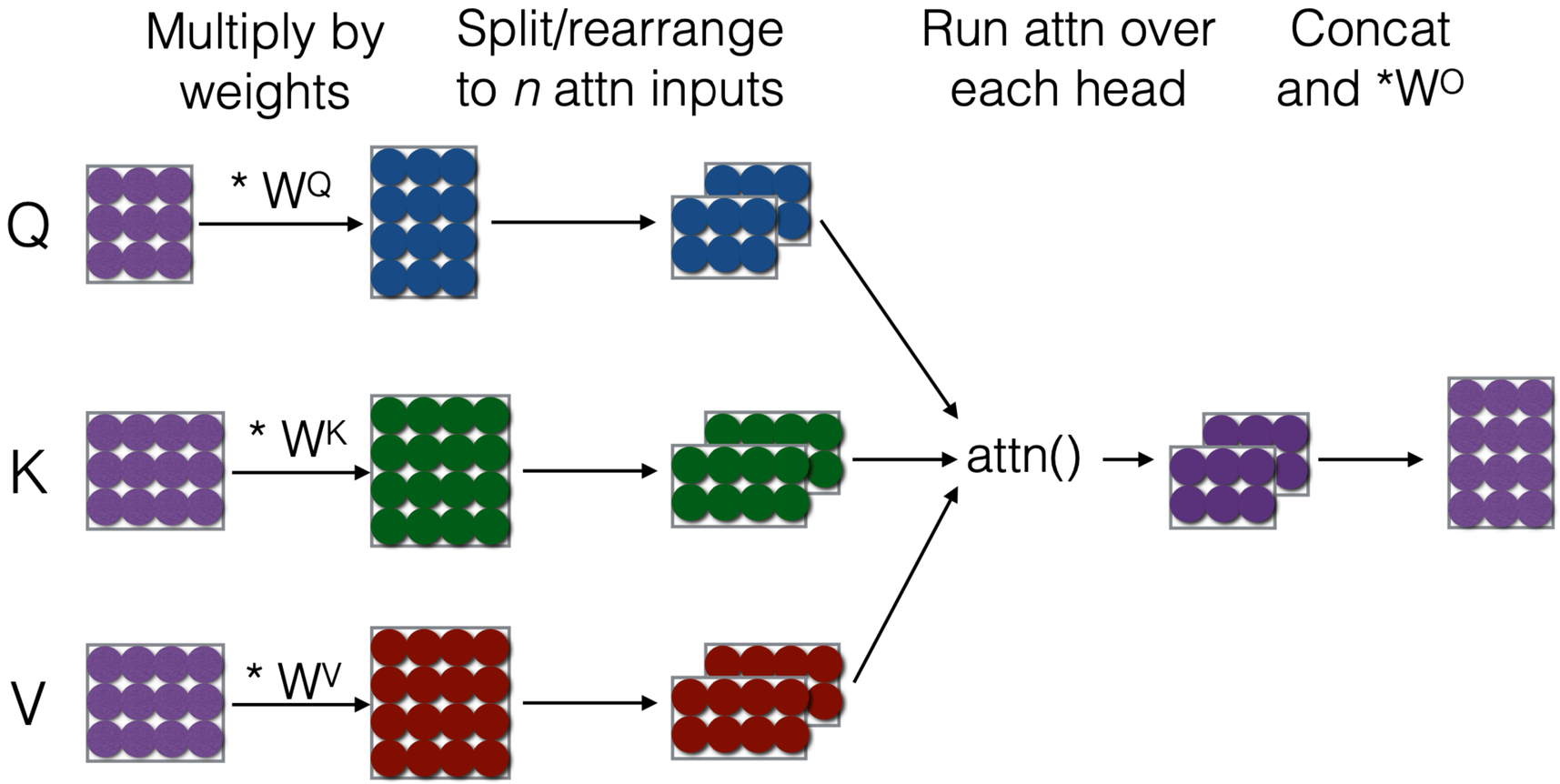
Multi-Head Attention



Multi-Head Attention

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

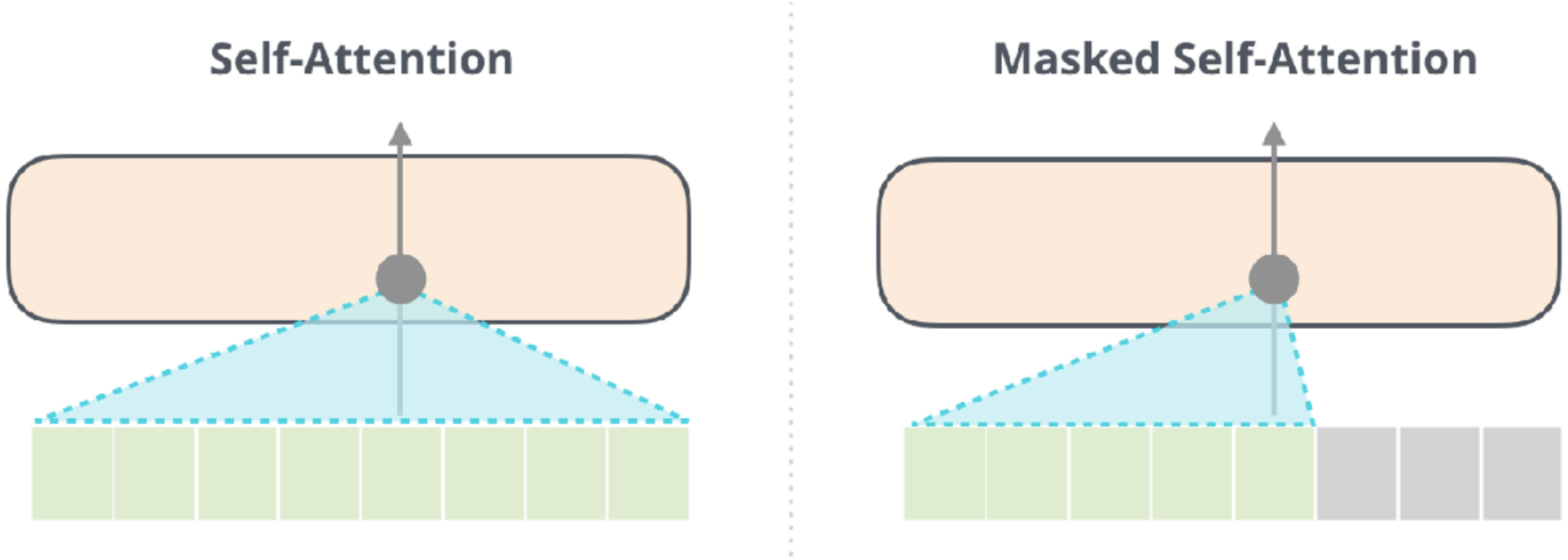
- When implementing multi-head attention (MHA)



Transformer: Masked Multi-Head Attention

Attention Can Look at the Future

- In decoders (generating the English part of Korean → English translation), we need to ensure we don't peek at the future
  - Masking the future sequence (tokens)



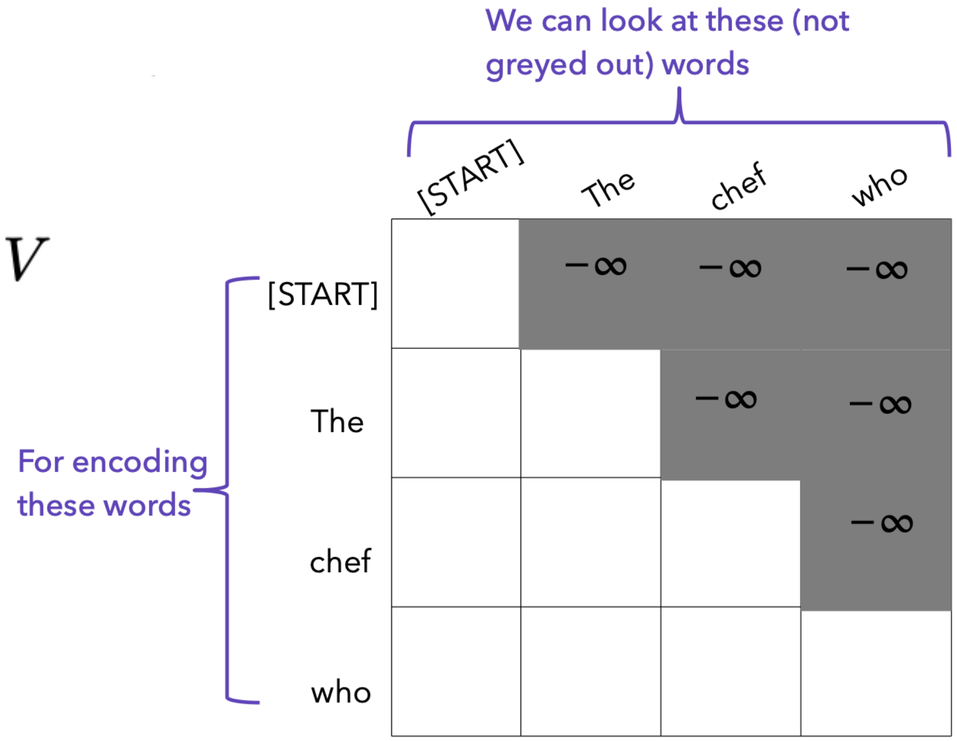
Masked Attention

- More efficiently (parallelization-friendly), we mask-out attention to future words by setting attention score to -inf

- Attention is now:

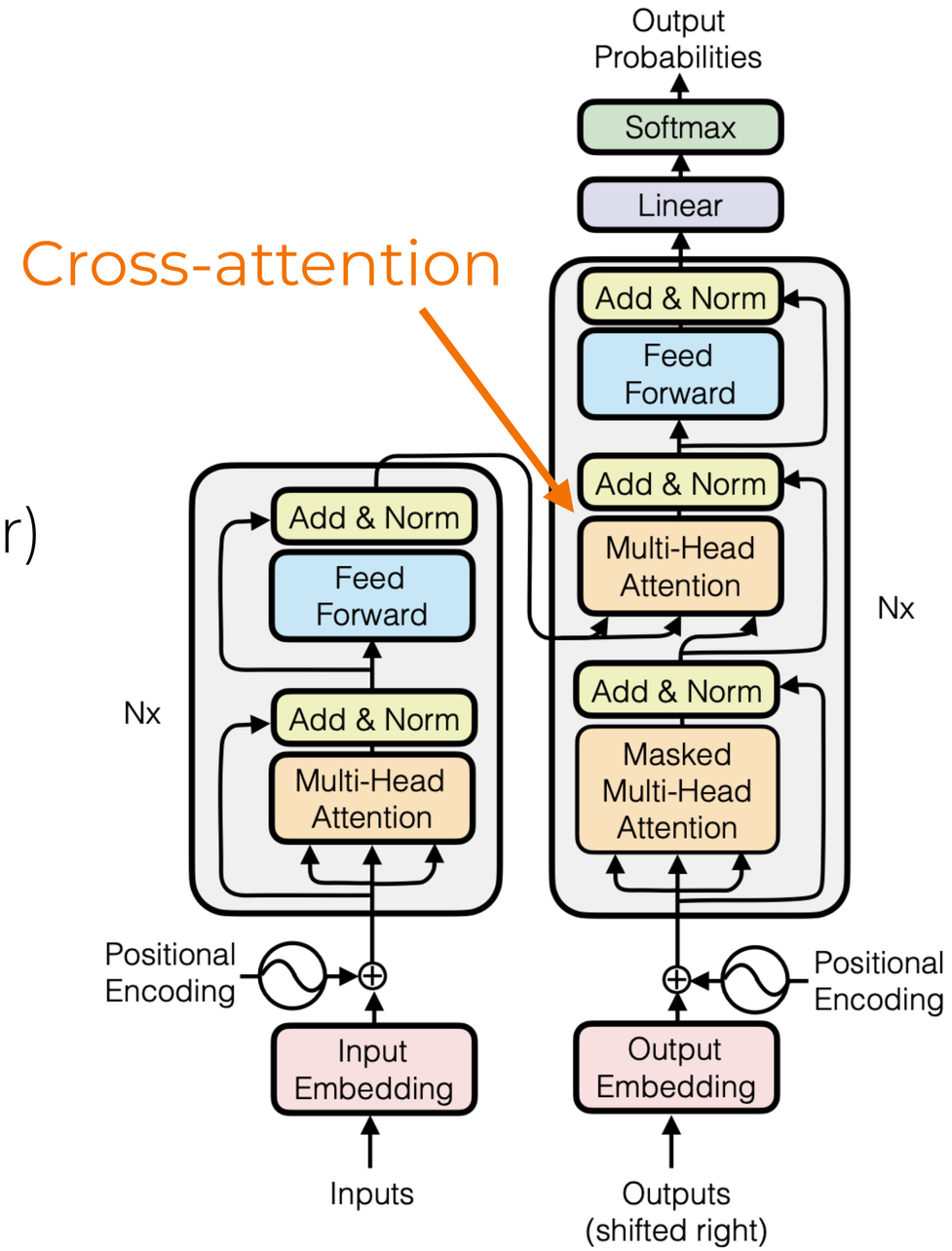
$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

$$M_{ij} = \begin{cases} 0 & \text{if position } j \text{ is allowed to attend} \\ -\infty & \text{if position } j \text{ is masked} \end{cases}$$



Recap

- We learned
  - Tokenization (word strings to indices)
  - Embedding (indices to matrix)
  - Positional Encoding (add sequence order)
  - Attentions:
    - Multi-Head Attention
    - Masked Multi-Head Attention

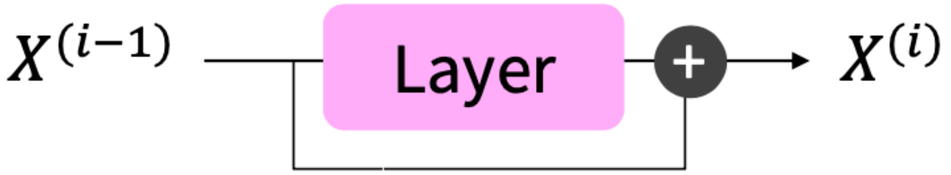


Transformer:
Residual Connection & Layer Normalization

Residual Connection [He+ 2016]

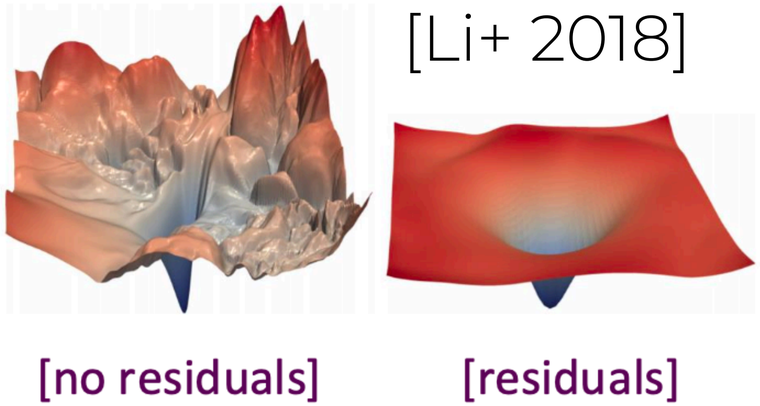
- Residual connections are a trick to help models train better
  - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$

  - We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$
    - We only have to learn the residual from the previous layer



Loss landscape visualization
w/, w/o residual

[Li+ 2018]



Layer Normalization [Ba+ 2016]

- Normalizes the outputs to be within a consistent range, preventing too much variance in scale of outputs

gain

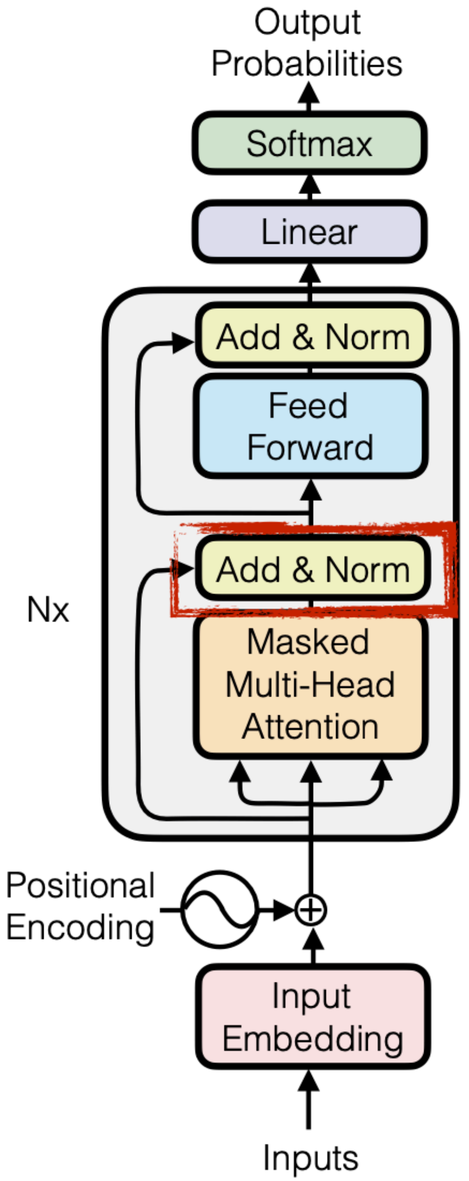
bias

$$\text{LayerNorm}(\mathbf{x}; \mathbf{g}, \mathbf{b}) = \frac{\mathbf{g}}{\sigma(\mathbf{x})} \odot (\mathbf{x} - \mu(\mathbf{x})) + \mathbf{b}$$

vector stddev

vector mean

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



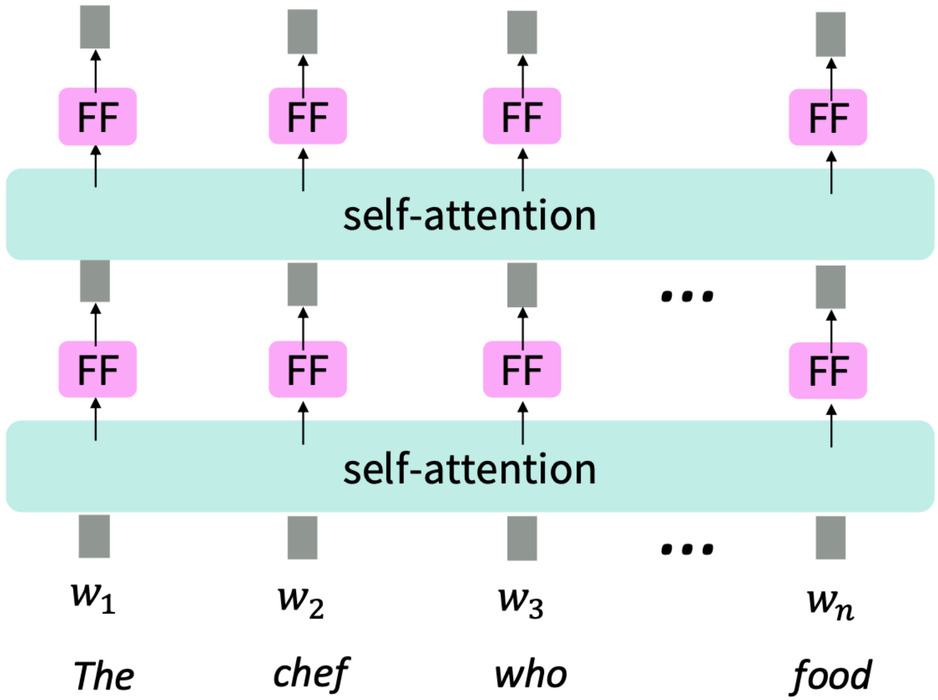
Transformer: Pointwise FFN

Pointwise Feed Forward Network

- No elementwise nonlinearities in self-attention
- Stacking more self-attention layers just re-averages value vectors

- Solution: Adding non-linearities (FFN) to post-process each output vector

$$\text{FFN}(x; W_1, \mathbf{b}_1, W_2, \mathbf{b}_2) = f(\mathbf{x}W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$$



Summary

- We learned
  - Tokenization (word strings to indices)
  - Embedding (indices to matrix)
  - Positional Encoding (add sequence order)
  - Attentions:
    - Multi-Head Attention
    - Masked Multi-Head Attention
  - Residual Connections (standard approach)
  - Layer Normalization (increase stability)
  - Pointwise FFN (add nonlinearities)