CS584 Natural Language Processing

Attention and Transformers

Yue Ning
Department of Computer Science
Stevens Institute of Technology

Today's lecture

- Attention models
- Transformers

NMT research continues

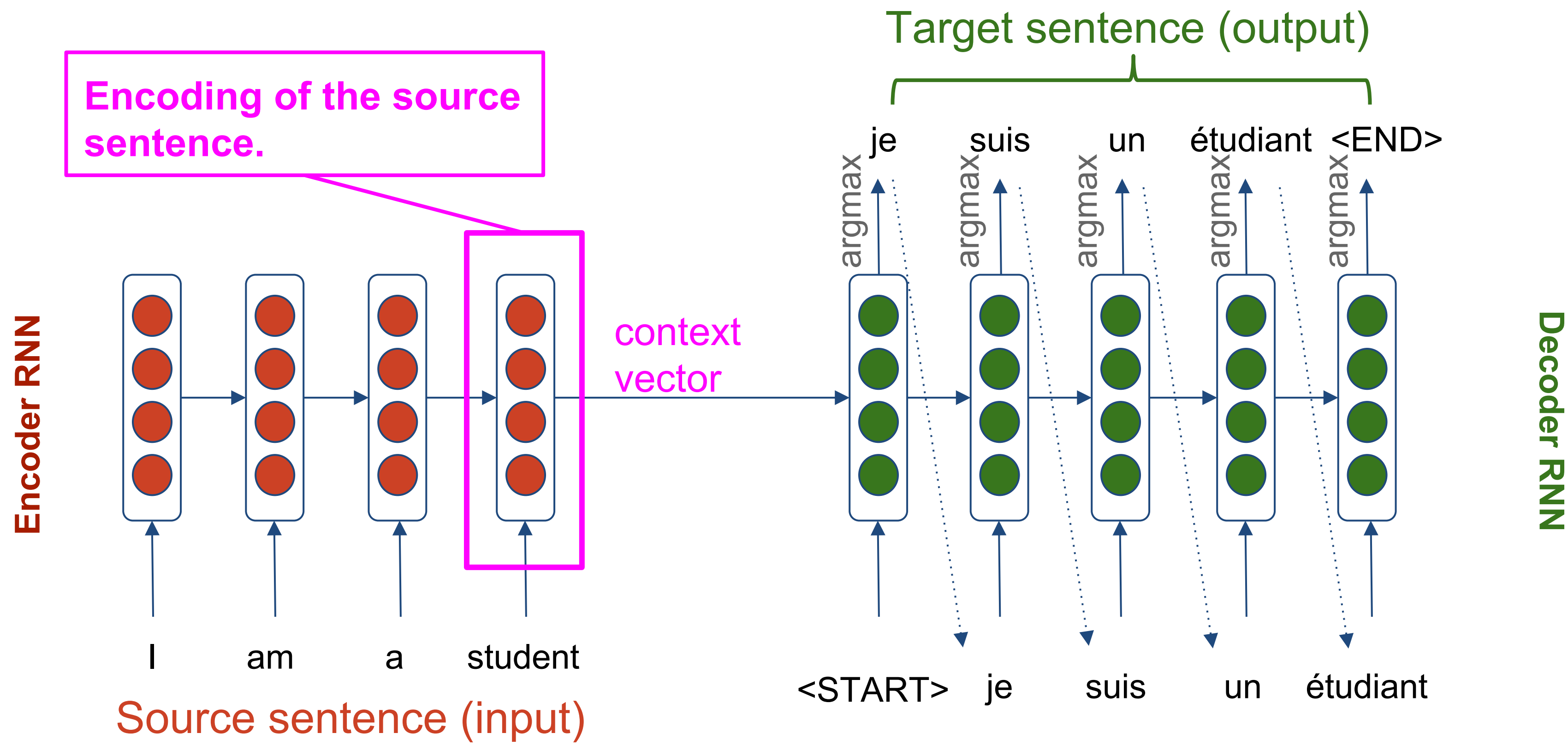
- NMT is the flagship task for NLP Deep Learning

- NMT research has pioneered many of the recent innovations of NLP Deep Learning

- In 2019: NMT research continues to thrive
  - Researchers have found many, many improvements to the “vanilla” seq2seq NMT system we’ve presented.
  - But one improvement is so integral that it is the new vanilla…

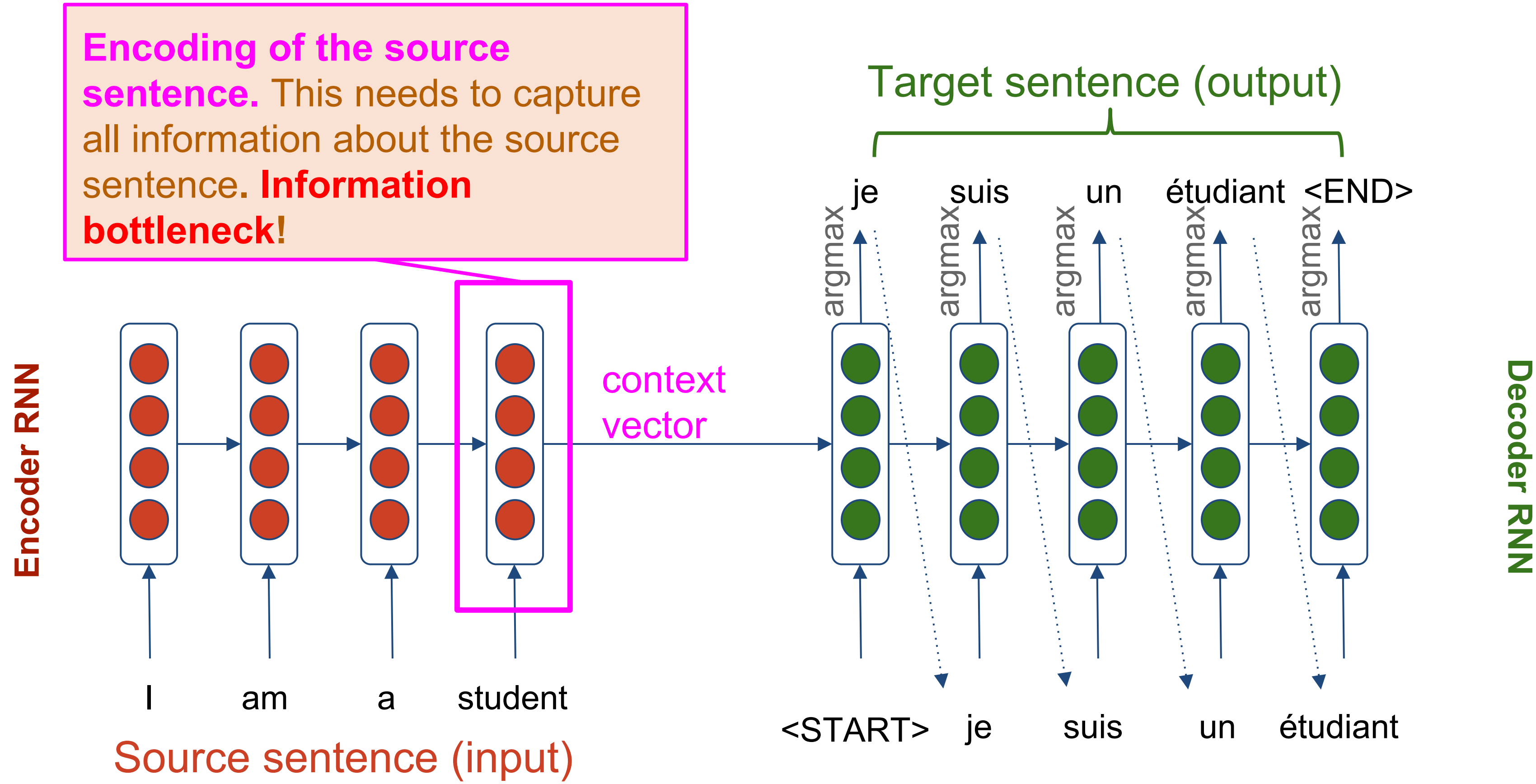
Attention

Seq2Seq: the bottleneck problem



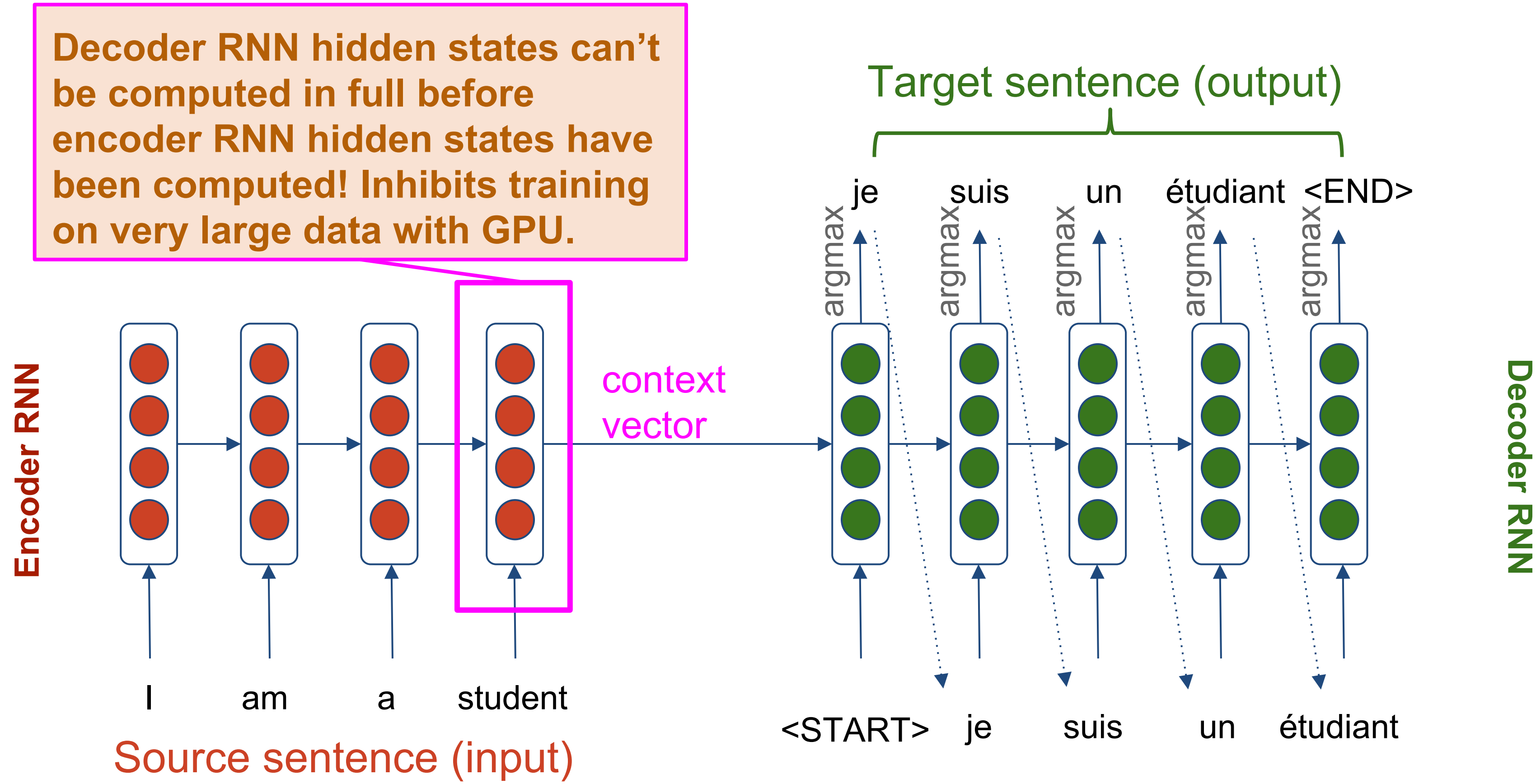
Problems with this architecture?

Seq2Seq: the bottleneck problem



Problems with this architecture?

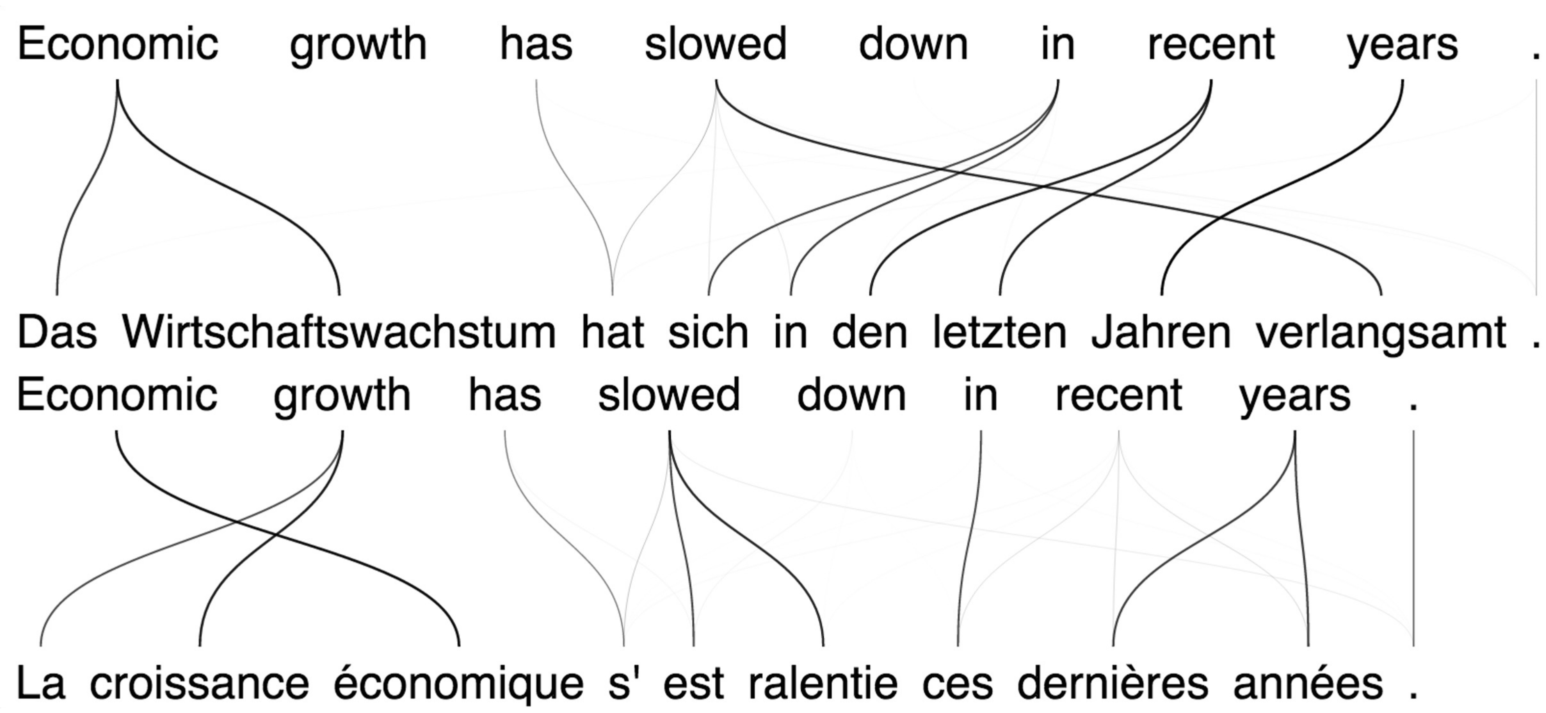
Seq2Seq: the bottleneck problem



Problems with this architecture?

How about attention?

- Attention treats each word's representation as a query to access and incorporate information from a set of values.



Attention

- Attention provides a solution to the bottleneck problem.

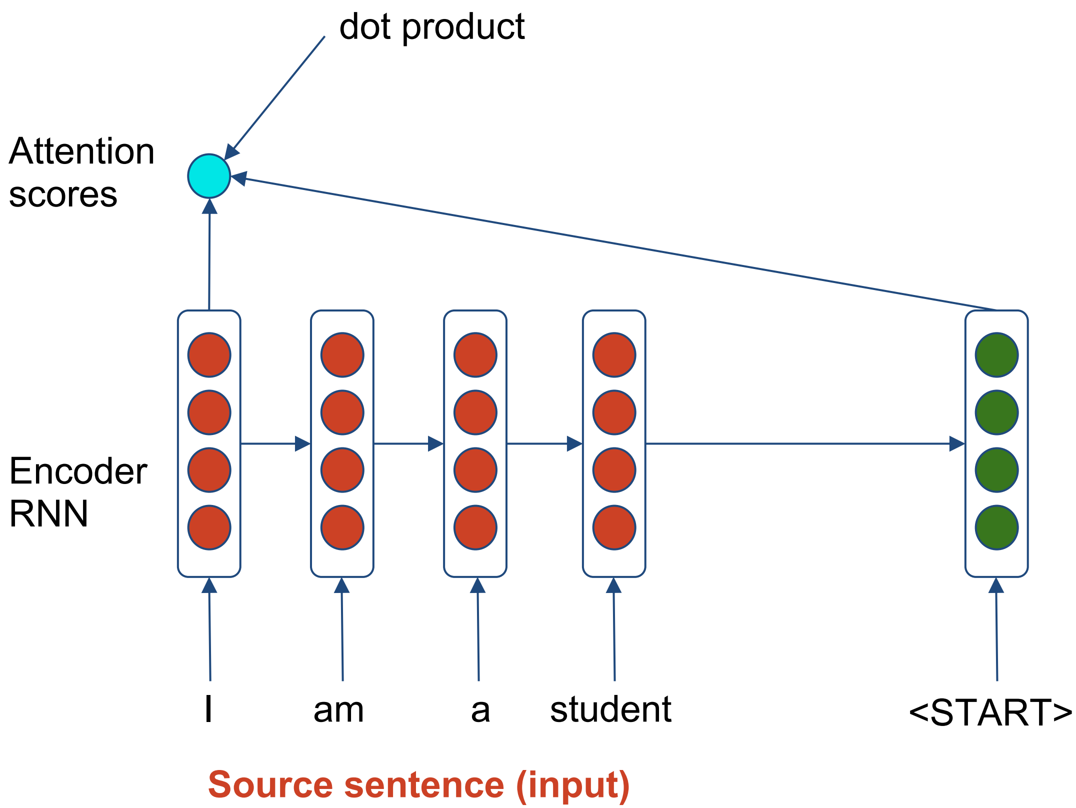
- **Core idea:** on each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence.

- There are many types of attention. We'll examine the encoder mechanisms introduced by Huong et al.[1].

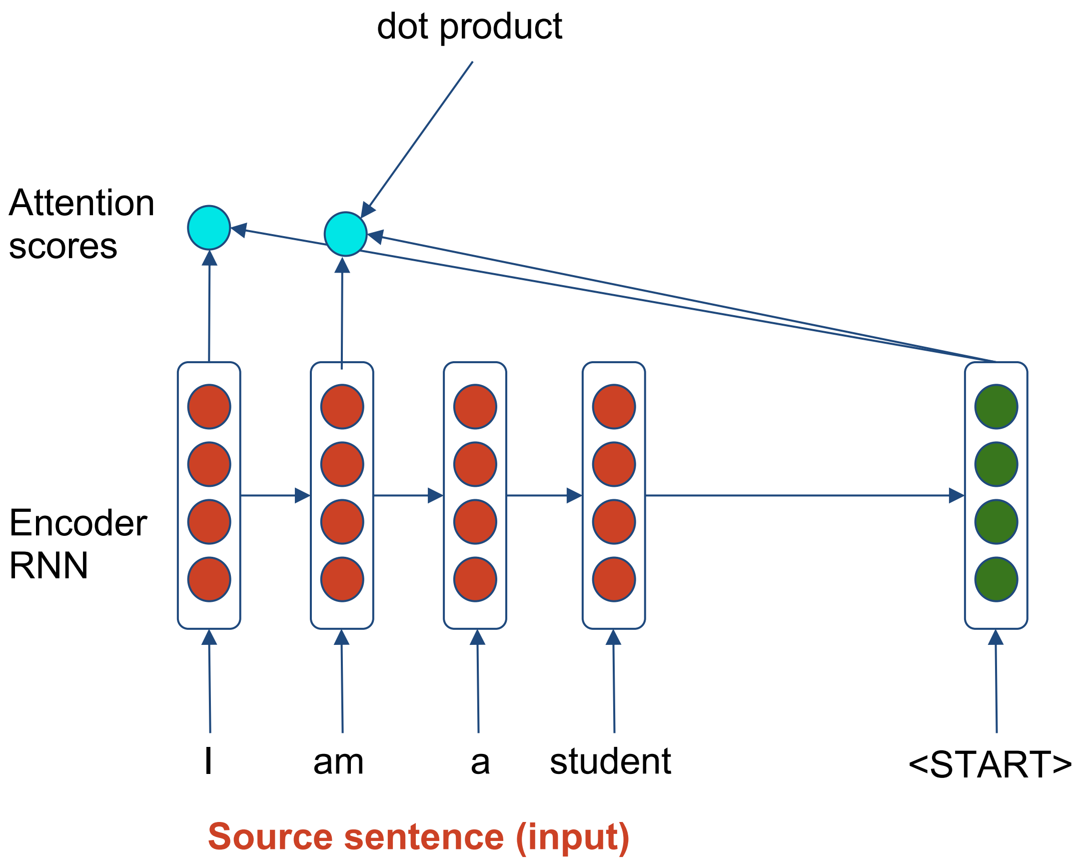
- First, we will show via diagram (no equations), then we will show with equations.

[1]. Huong et al. 2015, Effective Approaches to Attention-based Neural Machine Translation.

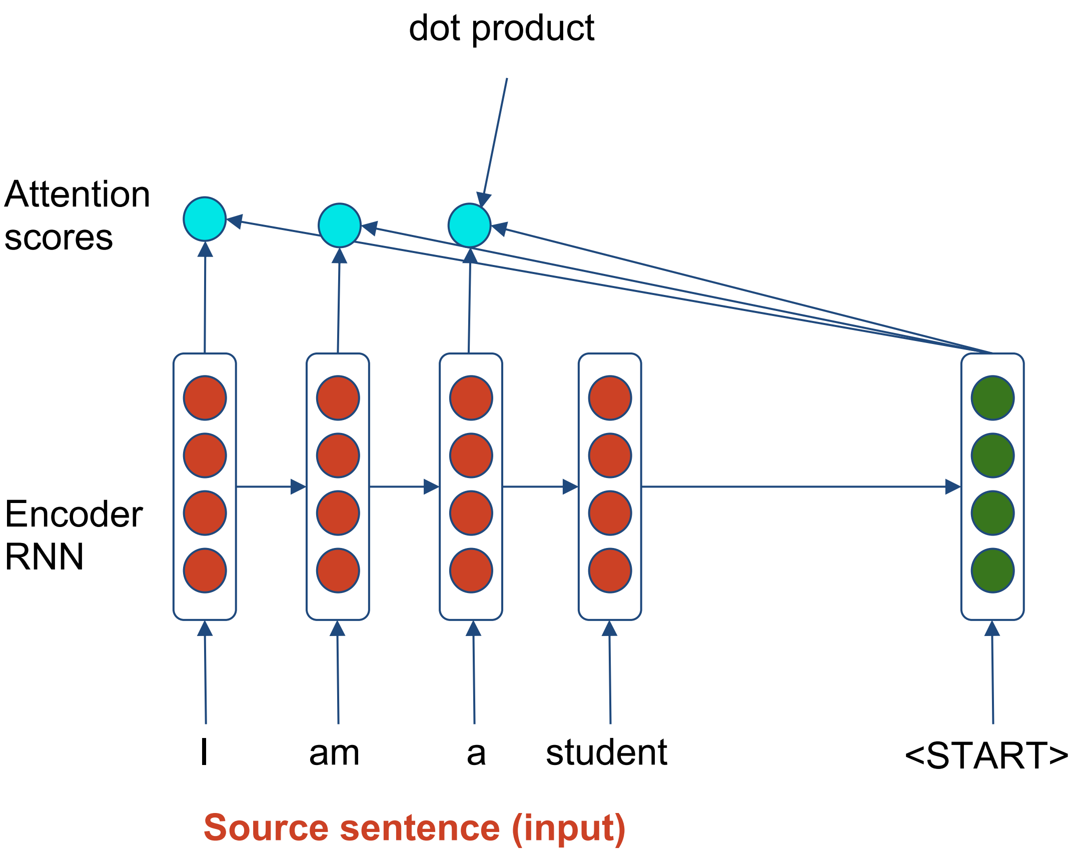
Seq2Seq with attention



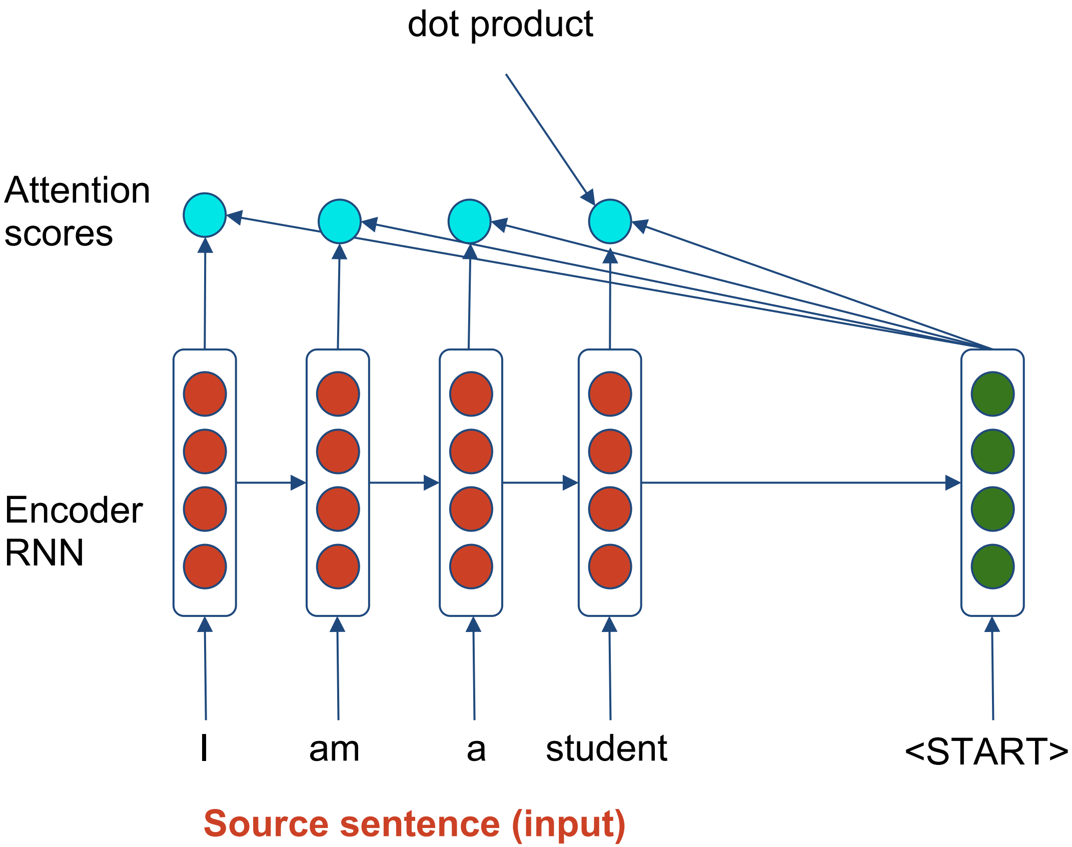
Seq2Seq with attention



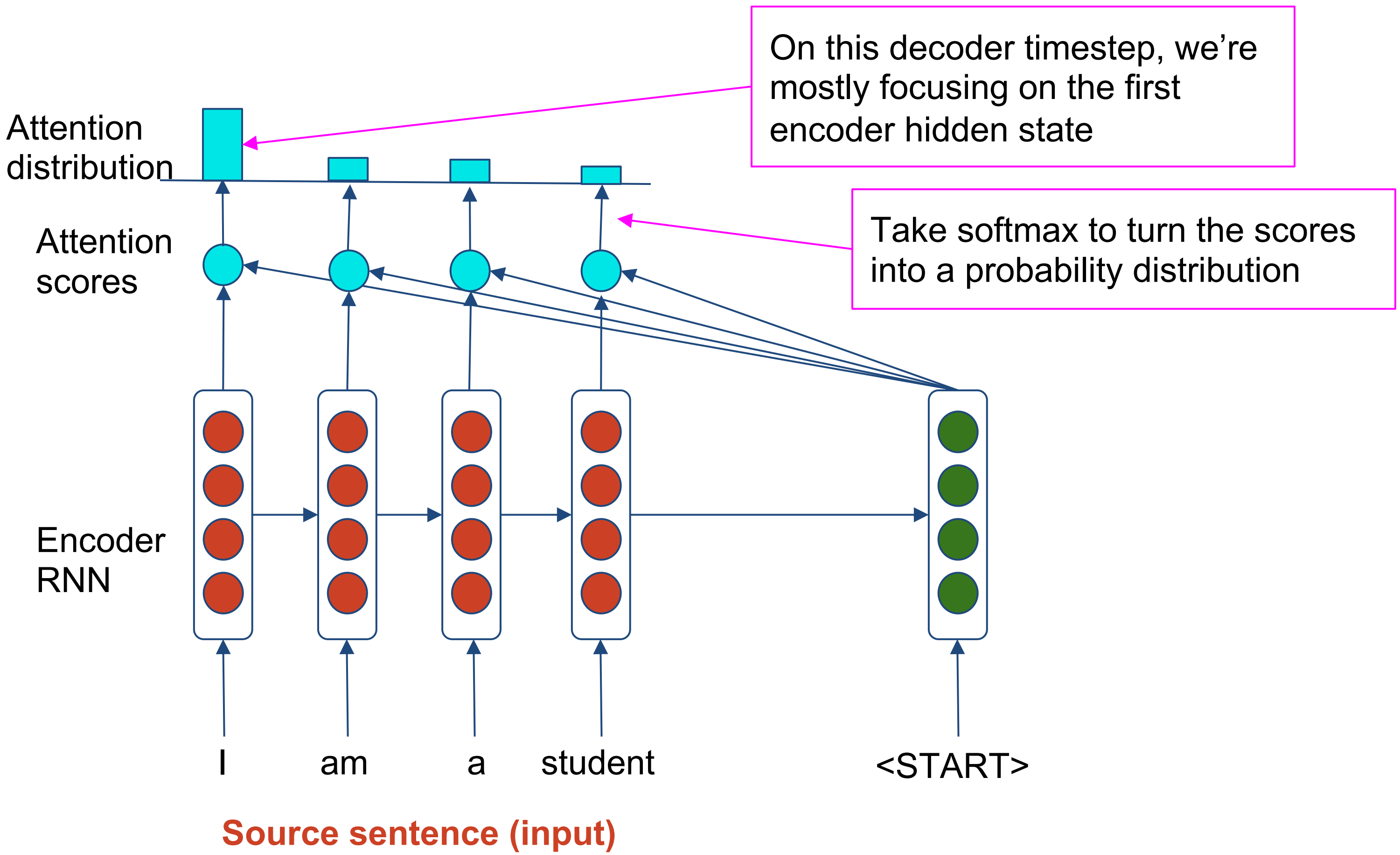
Seq2Seq with attention



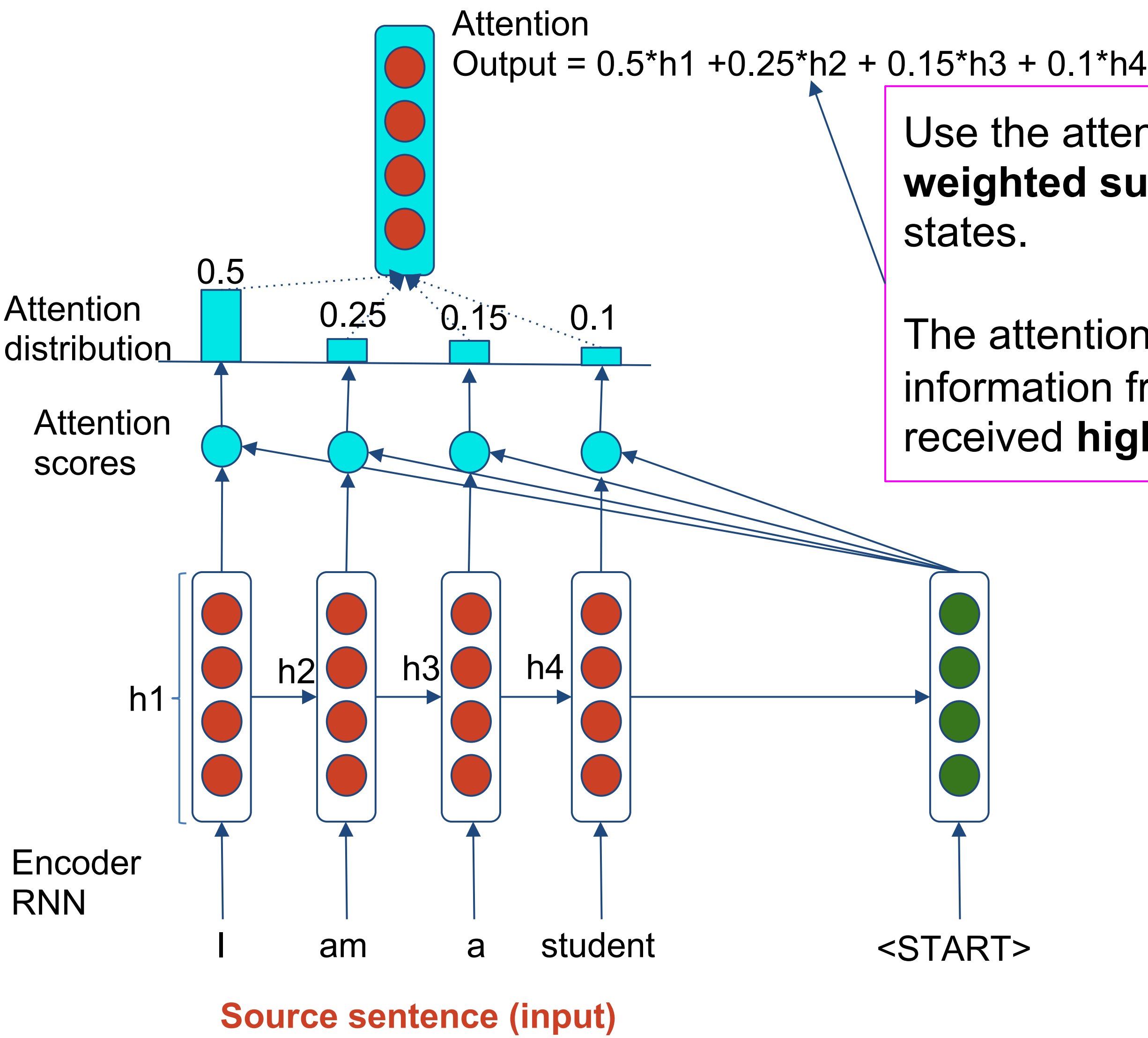
Seq2Seq with attention



Seq2Seq with attention



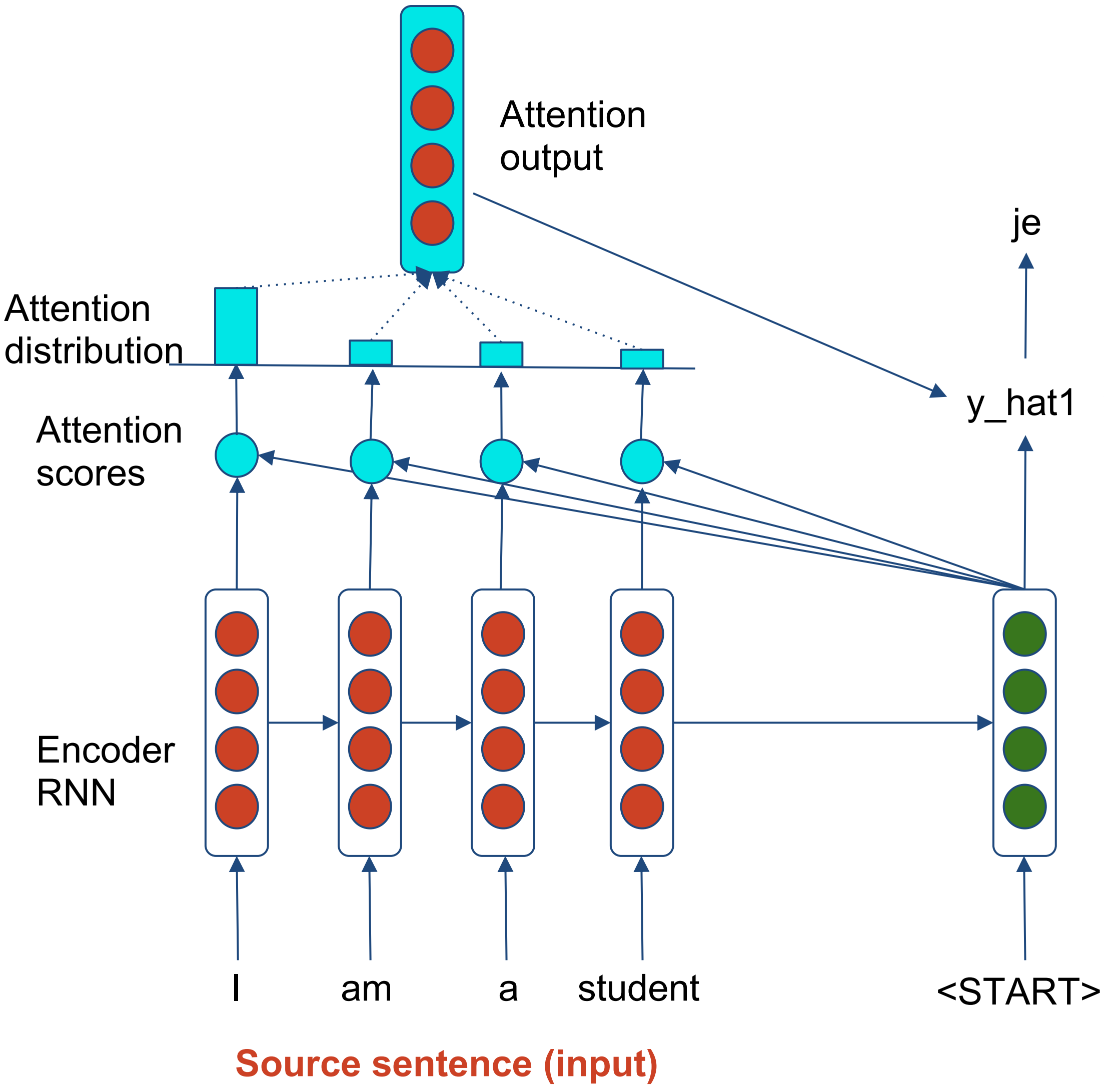
Seq2Seq with attention



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

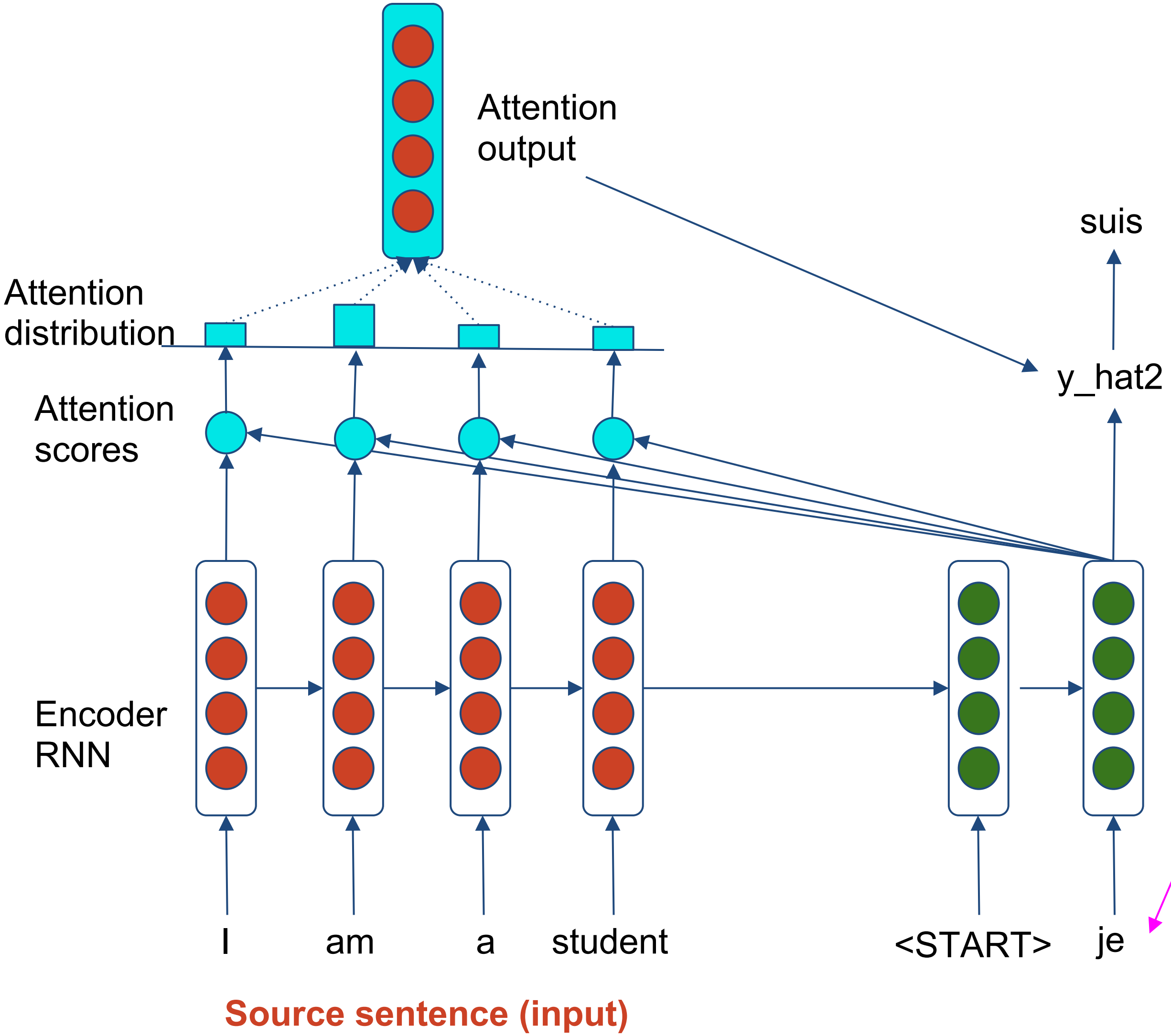
The attention output mostly contains information from the **hidden states** that received **high attention**.

Seq2Seq with attention



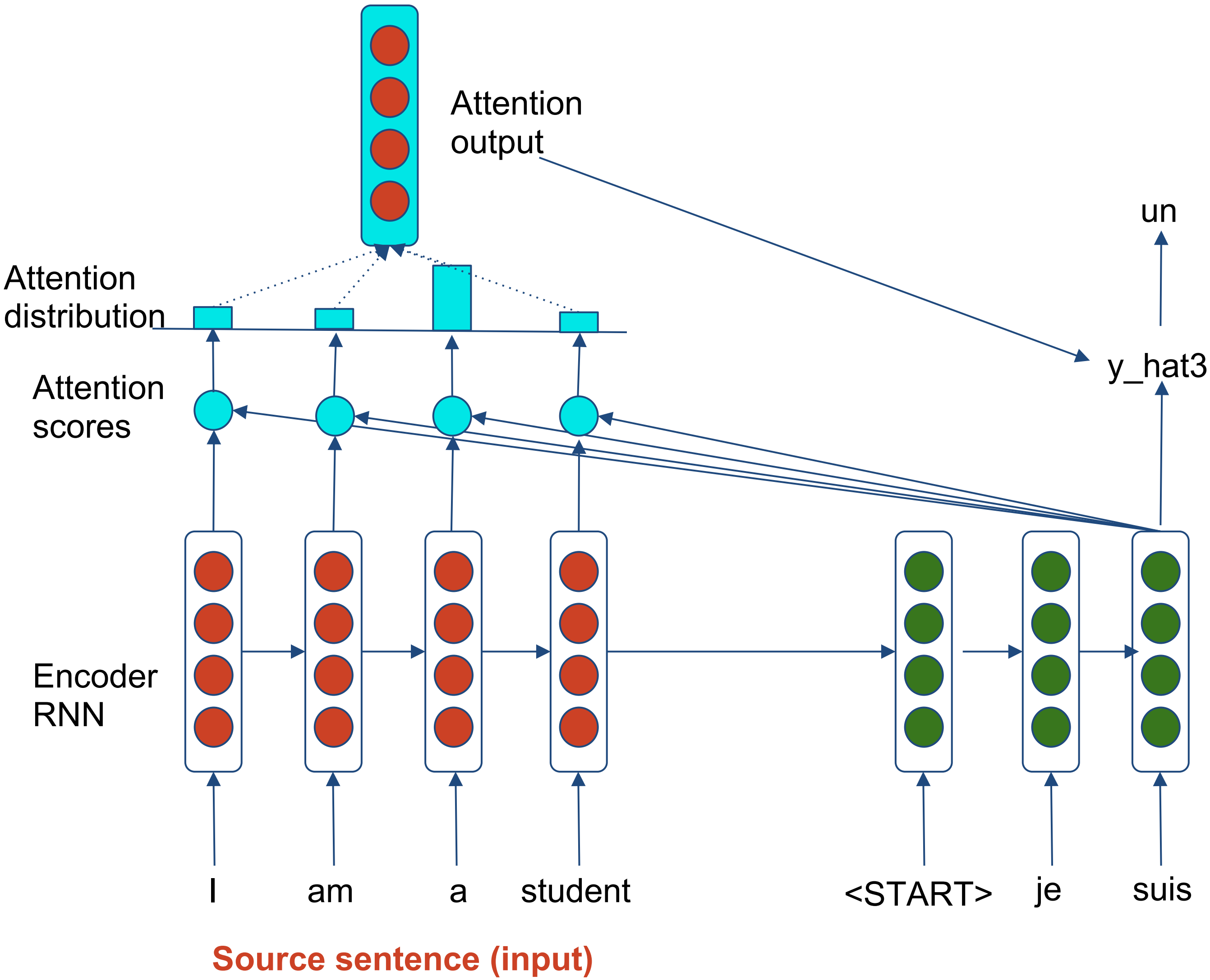
Concatenate attention output with decoder hidden state, then use to compute y_hat1 as before

Seq2Seq with attention

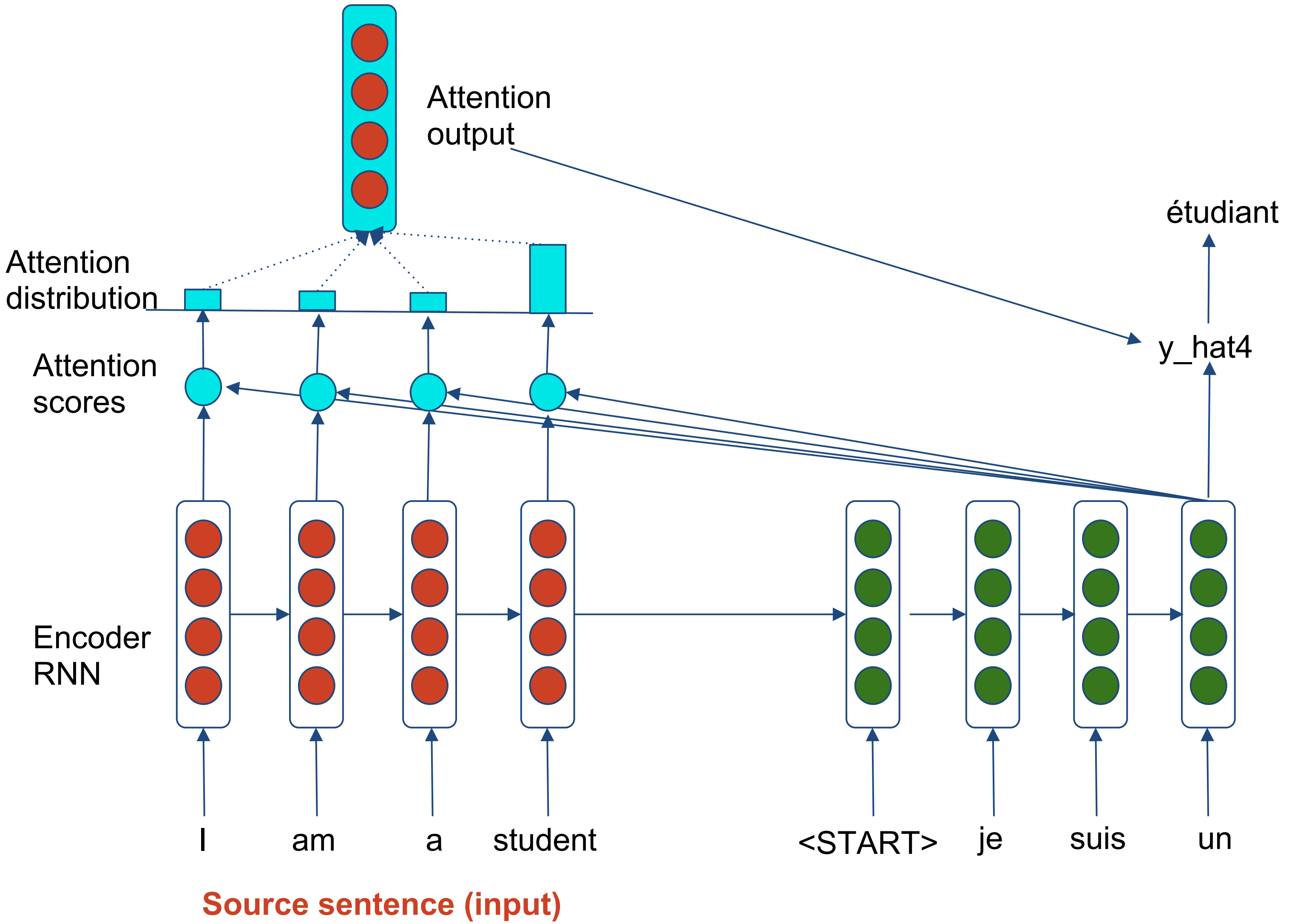


Sometimes we take the attention output from the previous step, and also feed it into the decoder (along with the usual decoder input).

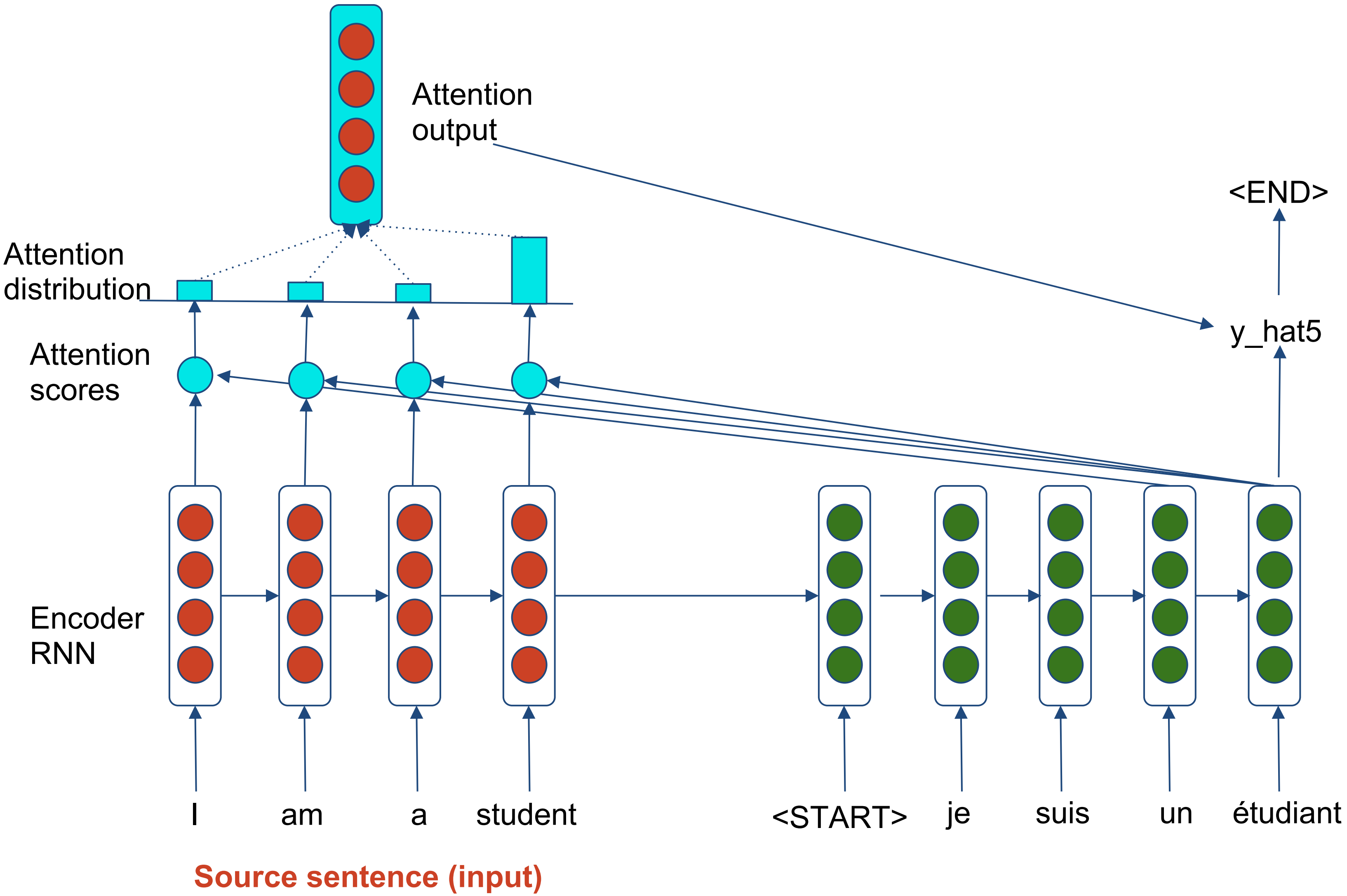
Seq2Seq with attention



Seq2Seq with attention



Seq2Seq with attention



Attention: in equations

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$

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

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

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

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

Attention: in equations

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

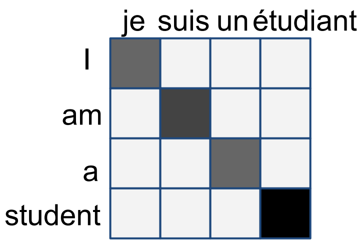
$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

- Finally, we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state $\boldsymbol{s}_t$ and proceed as in the non-attention seq2seq model

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

Attention is great

- Attention significantly improves NMT performance
  - Decoder focuss on certain parts of the source
- Attention solves the bottleneck problem
  - Decoder looks directly at source; bypass bottleneck
- Attention helps with vanishing gradient problem
  - Provides shortcut to far away states
- Attention provides some interpretability
  - Inspecting attention distribution
  - We get (soft) alignment for free!
  - No need to explicitly train an alignment system



Attention is a general Deep Learning technique

- We've seen that attention is a great way to improve the sequence-to-sequence model for Machine Translation.

- We can use attention in many architectures (not just seq2seq) and many tasks (not just MT)

Attention is a general Deep Learning technique

- More general definition of 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).

Attention is a general Deep Learning technique

- More general definition of attention:

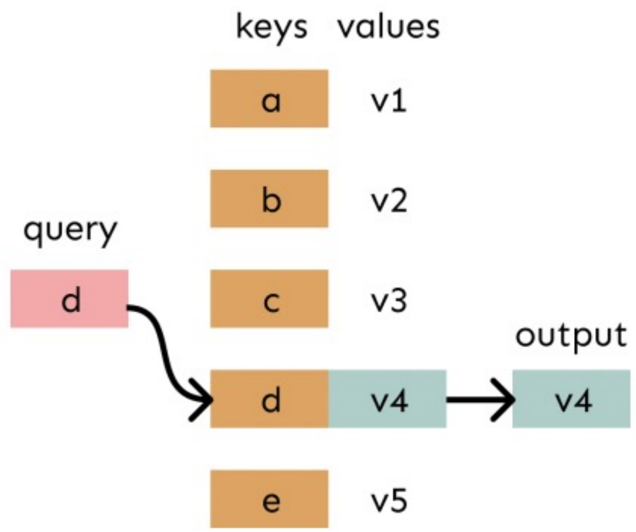
> Given a set of vector *values*, and a vector *query*, attention is a technique to compute a weighted sum of the values, dependent on the query.

- Intuition:
  - A selective summary of the information contained in the values, where the query determines which values to focus on.
  - A fixed-size representation of an arbitrary set of representations (the values), dependent on some other representation (the query).

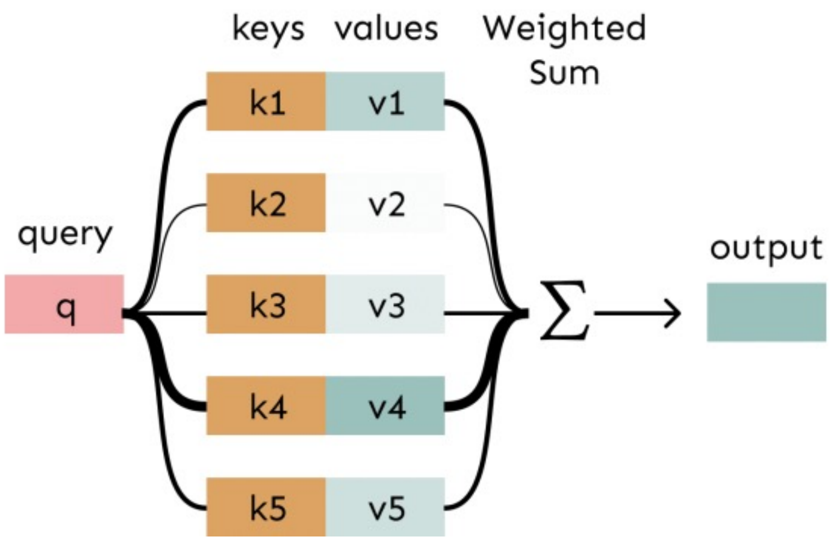
Attention as a soft, averaging lookup table

We can think of attention as performing 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.



Source: https://web.stanford.edu/class/cs224n/slides/cs224n-2023-lecture08-transformers.pdf

Attention variants

There are several ways you can compute $\boldsymbol{e} \in \mathbb{R}^N$ from $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and $\boldsymbol{s} \in \mathbb{R}^{d_2}$:

- Basic dot-product attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{h}_i \in \mathbb{R}$
  - Note: this assumes $d_1 = d_2$
  - This is the version we saw earlier
- Multiplicative attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{W} \boldsymbol{h}_i \in \mathbb{R}$
  - Where $\boldsymbol{W} \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix
- Additive attention: $\boldsymbol{e}_i = \boldsymbol{v}^T \tanh(\boldsymbol{W}_1 \boldsymbol{h}_i + \boldsymbol{W}_2 \boldsymbol{s}) \in \mathbb{R}$
  - Where $\boldsymbol{W}_1 \in \mathbb{R}^{d_3 \times d_1}, \boldsymbol{W}_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $\boldsymbol{v} \in \mathbb{R}^{d_3}$ is a weight vector.

Attention variants

- We have some values $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and a query $\boldsymbol{s} \in \mathbb{R}^{d_2}$

- Attention always involves:
  1. Computing the attention scores $\boldsymbol{e} \in \mathbb{R}^N$
  2. Taking softmax to get attention distribution α:

There are multiple ways to do this

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

3. Using attention distribution to take weighted sum of values:

$$\boldsymbol{a} = \sum_{i=1}^{N} \alpha_i \boldsymbol{h}_i \in \mathbb{R}^{d_1}$$

thus obtaining the attention output **a** (sometimes called the context vector)

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

Every word is both a key and a query simultaneously

Let $w_{1:n}$ be a sequence of words in vocabulary V

e.g., “Zuko Made his uncle tea”

For each w_i, let $x_i = Ew_i$, where $E \in R^{d\times|V|}$is an embedding matrix.

1. Transform each word embedding with weight matrices

$$q_i = W^Q x_i \ (queries), k_i = W^K x_i \ (keys),\ v_i = W^V x_i \ (values)$$

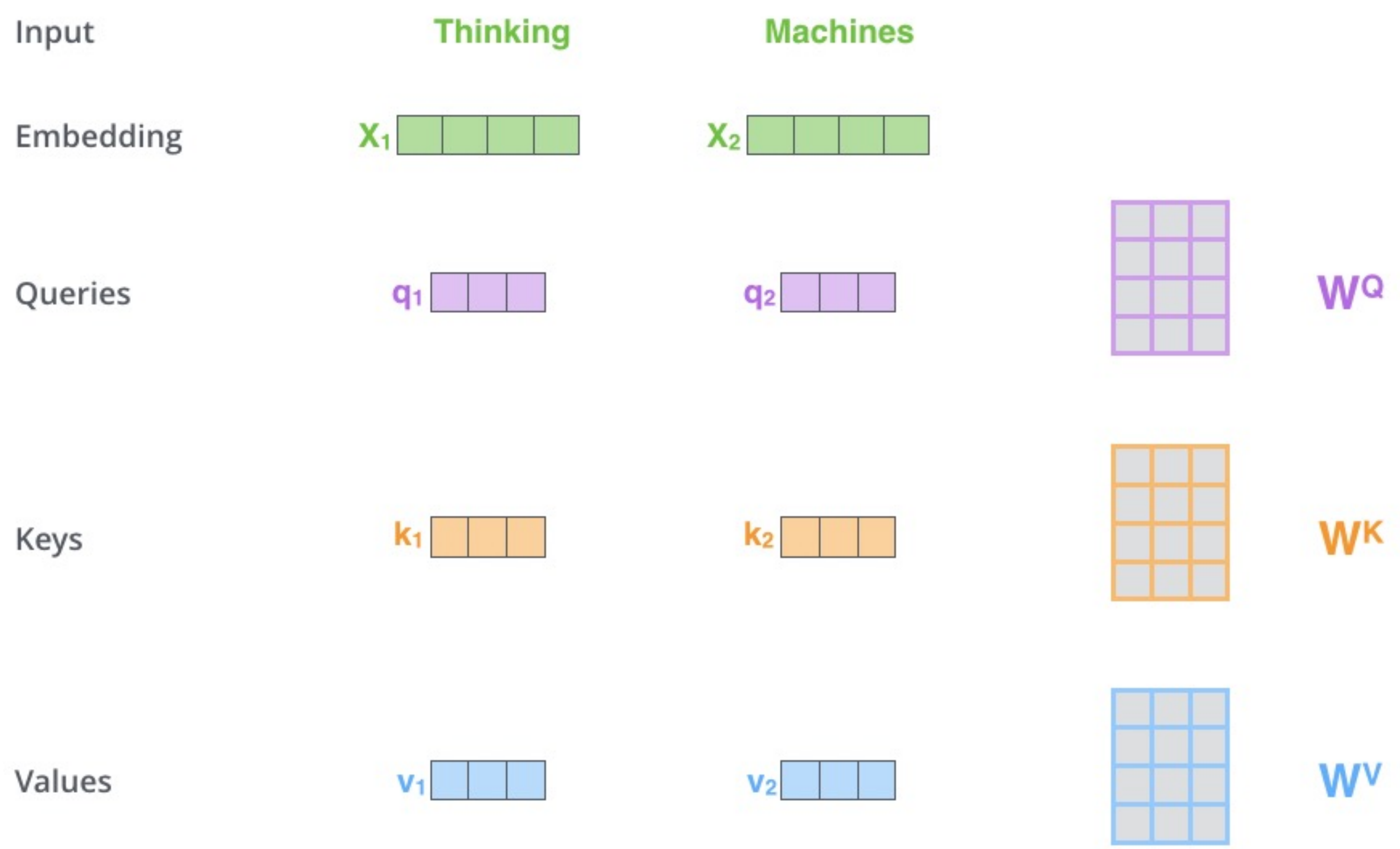
2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \text{inner\_product}(\boldsymbol{q}_i, \boldsymbol{k}_j) \quad \alpha_{ij} = \frac{\exp(eij)}{\sum_{j\prime} \exp(eij\prime)}$$

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

$$\boldsymbol{o}_i = \sum_j \alpha_{ij}\, \boldsymbol{v}_j$$

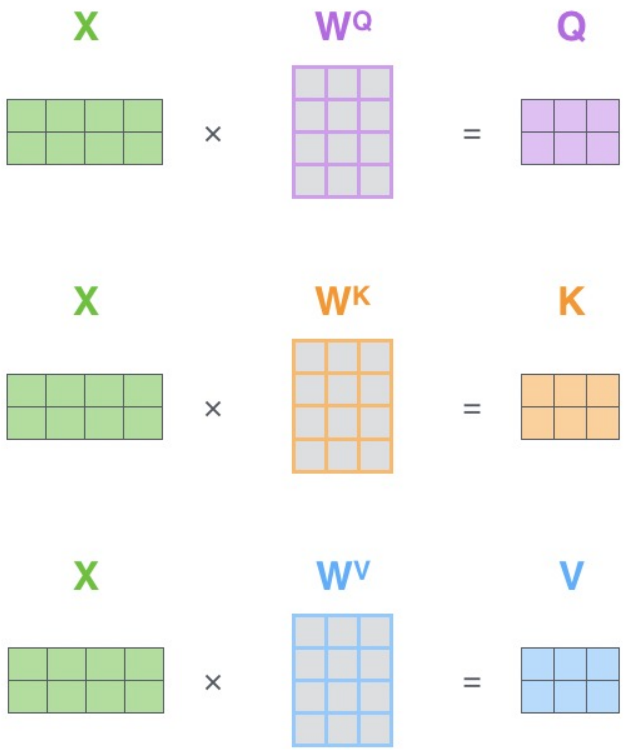
Matrix Calculation of Self-attention



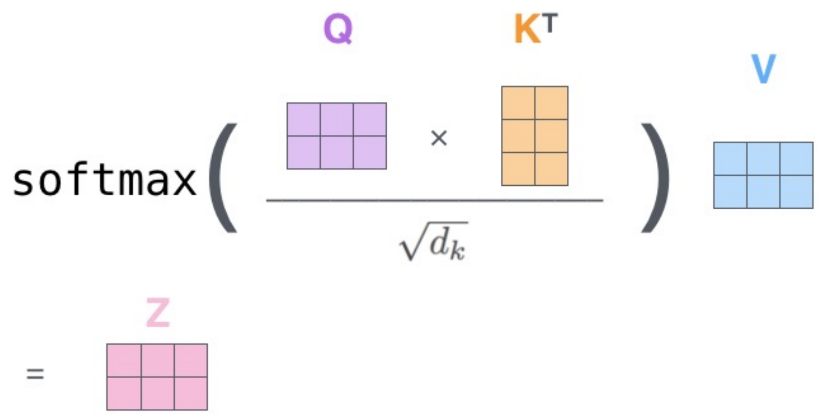
Multiplying X_1 by the W^Q weight matrix produces q_1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

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

Matrix Calculation of Self-attention



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



- Z is a weighted combination of V rows
- Normalizing by $\sqrt{d_k}$ helps control the scale of the softmax, makes it less peaked and have more stable gradients→Scaled dot product attention.
- This is just one head of self-attention — produce multiple heads via randomly initialize parameter matrices.

Barriers/Problems of Self-Attention

1. Doesn't have an inherent notion of order!

2. No nonlinearities for deep learning magic! It's all just weighted averages.

3. Need to ensure we don't "look at the future" when predicting a sequence.
   - Machine translation (target sequence generation)
   - Language modeling

Solution for first problem: sequence order

- Encode the order of the sentence in our keys, queries, and values
- Position vectors

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

- Don't worry about what the p_i are made of yet!
- How to incorporate this info into our self-attention block? just add the $\boldsymbol{p}_i$ to our inputs!
- Recall that $\boldsymbol{x}_i$ is the embedding of the word at index i. The positioned embedding is:

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

Position representation vectors through sinusoids

- **Inject** information about the relative or absolute positions?
- Sinusoidal position representations: concatenate sinusoidal functions of varying periods.
- Each dimension of the positional encoding corresponds to a sinusoid. Here, pos is the position, i is the dimension, and d_{model} is the output dimension.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

- Pros:
  - Periodicity indicates that maybe “absolute position” isn’t as important
  - Maybe can extrapolate to longer sequences as periods restart!
- Cons: Not learnable; the extrapolation doesn’t really work.

Position representation vectors learned from scratch

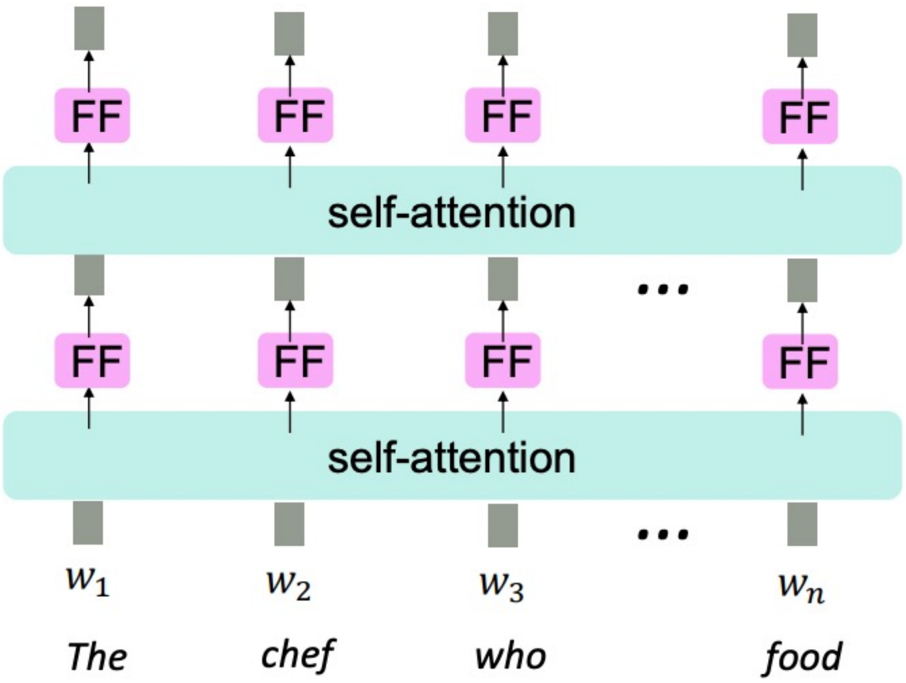
- Learned absolute position representations: Let all p_i be learnable parameters! Learn a matrix $\boldsymbol{p} \in \mathbb{R}^{d \times n}$, and let each $\boldsymbol{p}_i$ be a column of that matrix!
- Pros:
  - Flexibility: each position gets to be learned to fit the data
- Cons:
  - Definitely can't extrapolate to indices outside 1, … , n.
- Most systems use this!
- Sometimes people try more flexible representations of position:
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

Solution for second problem: No nonlinearities

Adding nonlinearities in self-attention:

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

- Easy fix: add a position-wise feedforward network to post-process each output vector to introduce nonlinearities with activation functions.
- Learns position-specific patterns and captures complex relationships in the data.

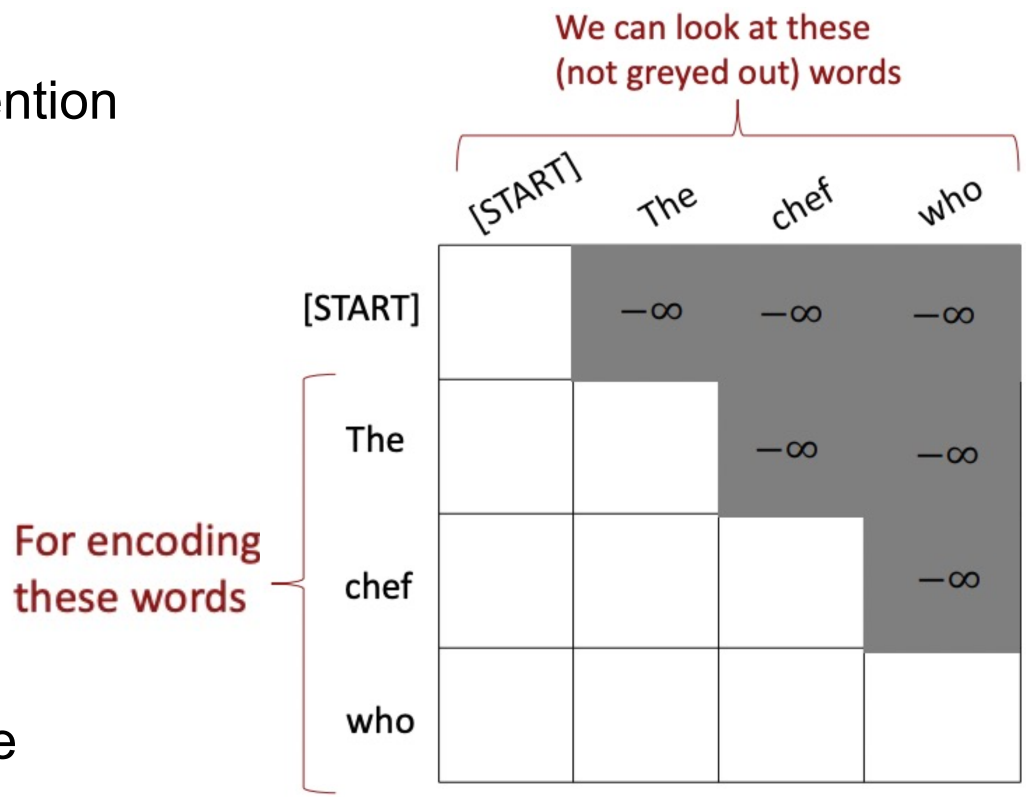


$$
\begin{aligned}
m_i &= MLP(\mathrm{output}_i) \\
&= W_2 * \mathrm{ReLU}(W_1\, \mathrm{output}_i + b_1) + b_2
\end{aligned}
$$

Solution for third problem: Avoid peeking at the future

<u>Masking the future</u> in self-attention

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



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

Barriers and solutions for Self-Attention as a building block

Barriers

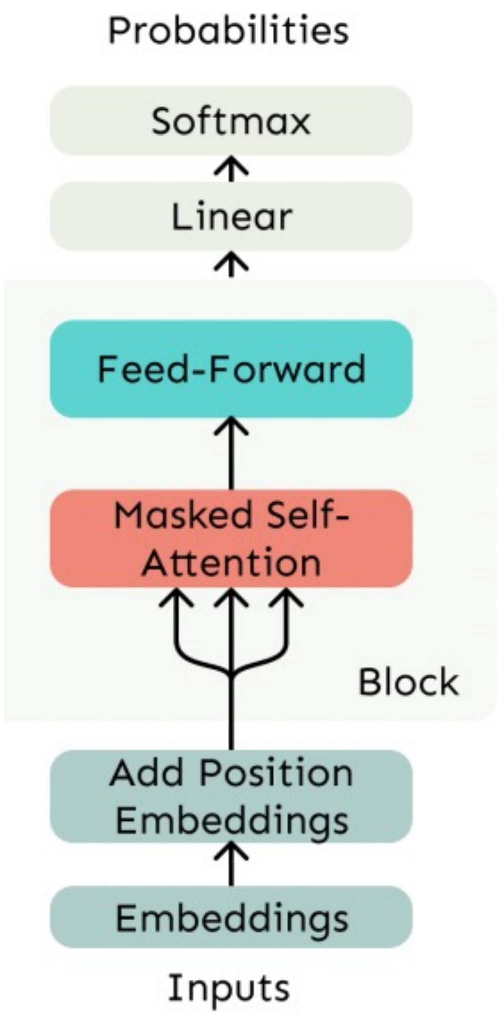
- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning magic! It's all just weighted averages
- Need to ensure we don't "look at the future" when predicting a sequence
  - Machine translation
  - Language modeling

Solutions

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output.
- Mask out the future by artificially setting attention weights!

Necessities for a self-attention building block

- **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.



The Transformer Model

Transformer

- Based on the <u>encoder-decoder</u> architecture
- Encoder: Converts an input sequence of tokens into a sequence of embedding vectors, often called the hidden state or context.
- Decoder: Uses the encoder's hidden state to iteratively generate an output sequence of tokens, one token at a time.

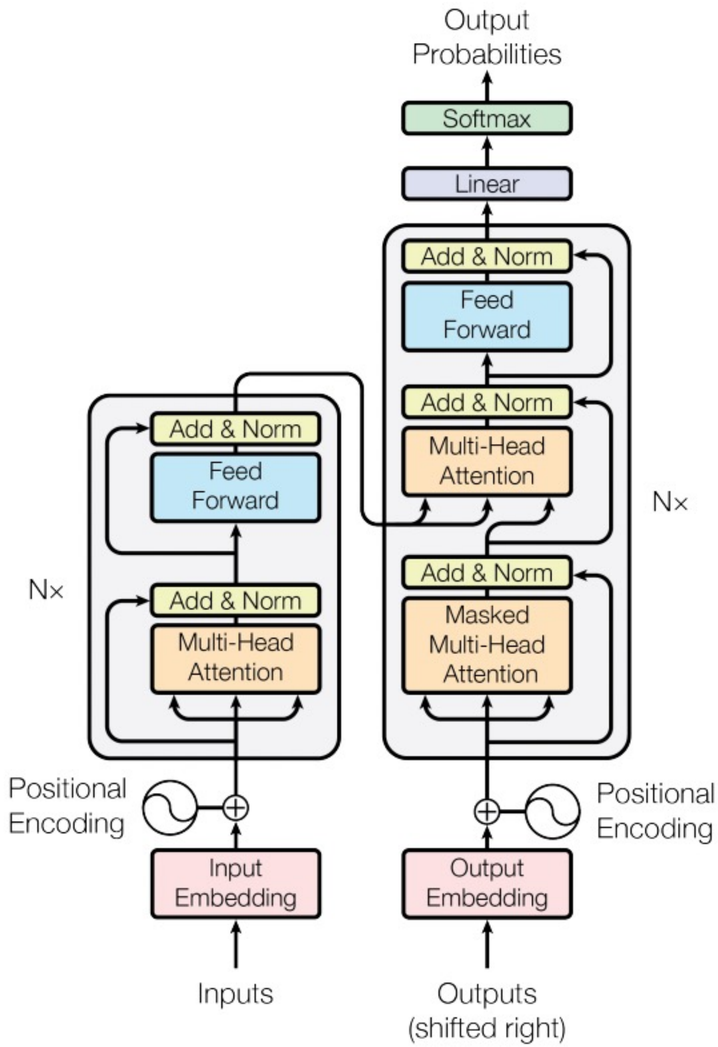


Figure 1: The Transformer - model architecture.

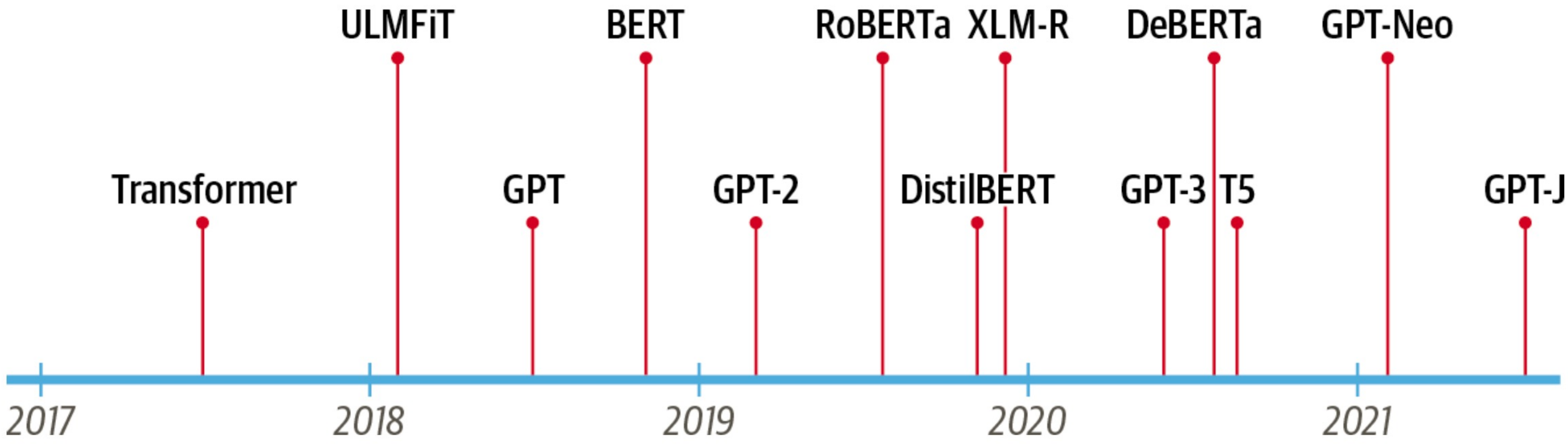
Standalone Transformer-based models

- **Encoder-only**: convert an input sequence into a rich numerical representation that is well suited for tasks like text classification or named entity recognition.
    - BERT[1] and its variants, like RoBERTa[2] and DistilBERT[3]
    - Representation is computed by bidirectional attention

- **Decoder-only**: Given a prompt of text like “Thanks for lunch, I had a...” these models will auto-complete the sequence by iteratively predicting the most probable next word.
    - The family of GPT models (ChatGPT, GPT[7], GPT2[4])
    - The representation is computed only based on the left context by autoregressive attention.

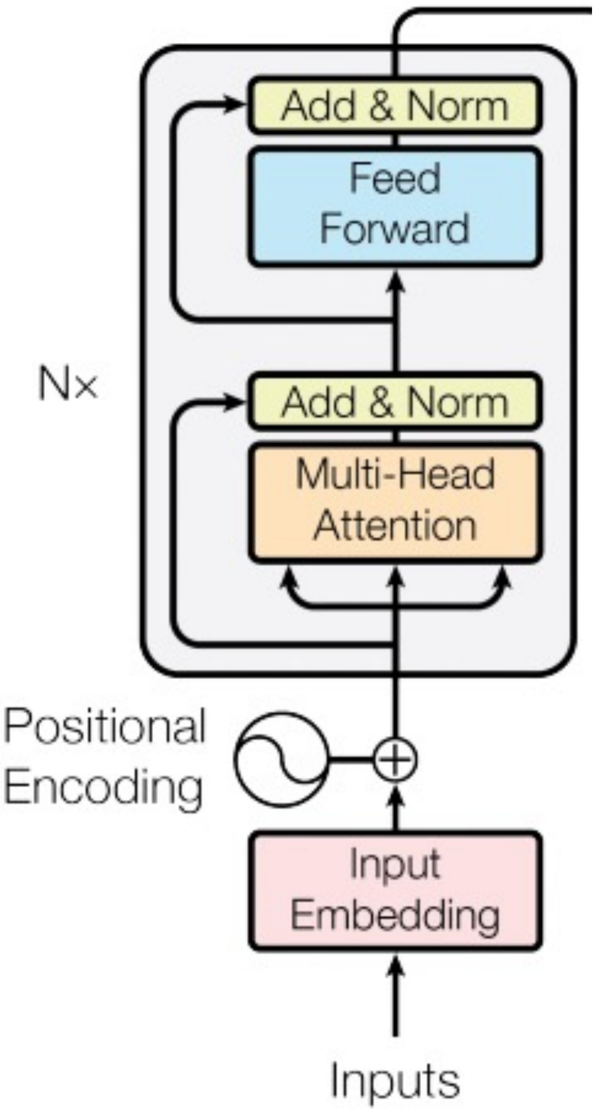
- **Encoder-Decoder**: for modeling complex mappings from one sequence of text to another.
    - BART[5] and T5[6] models

The timeline of transformers



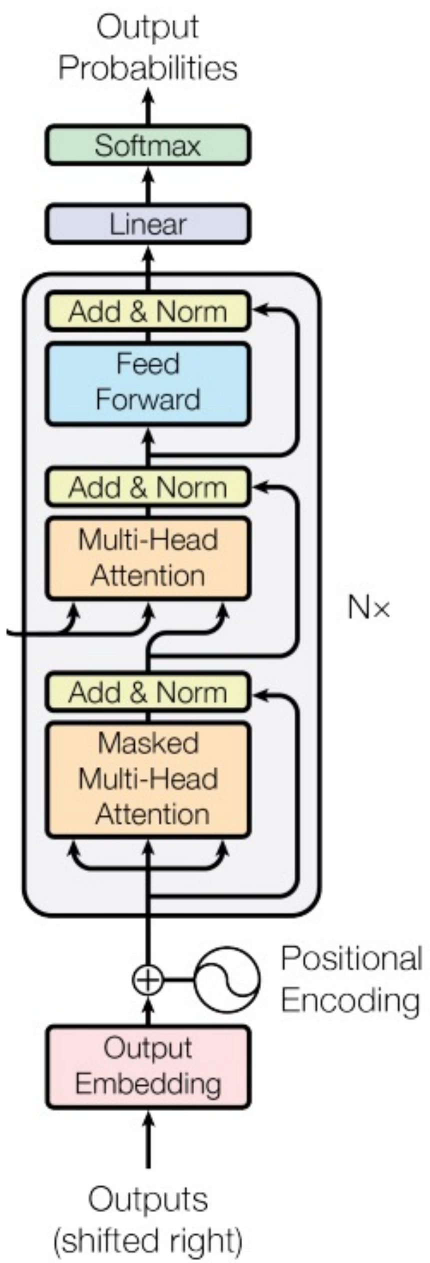
The Transformer Encoder

- The Transformer Encoder is a stack of Transformer Encoder Blocks.
- Each Block consists of:
  - Multi-head self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- The dimensions of both input embeddings and positional embeddings are identical.

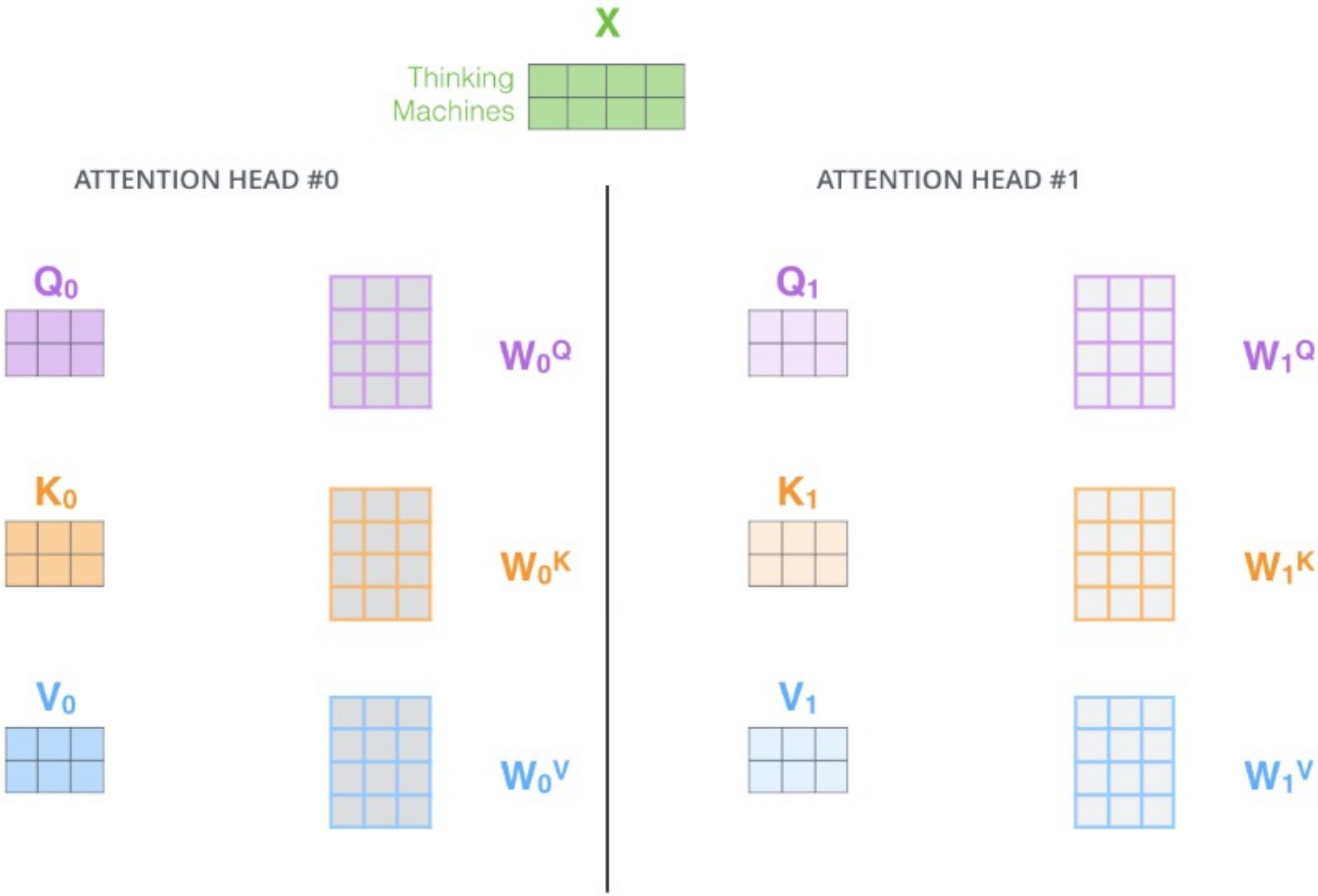


The Transformer Decoder

- A Transformer decoder is how we will build systems like language models.
- It's a lot like our minimal self-attention architecture, but with a few more components.
- In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack.

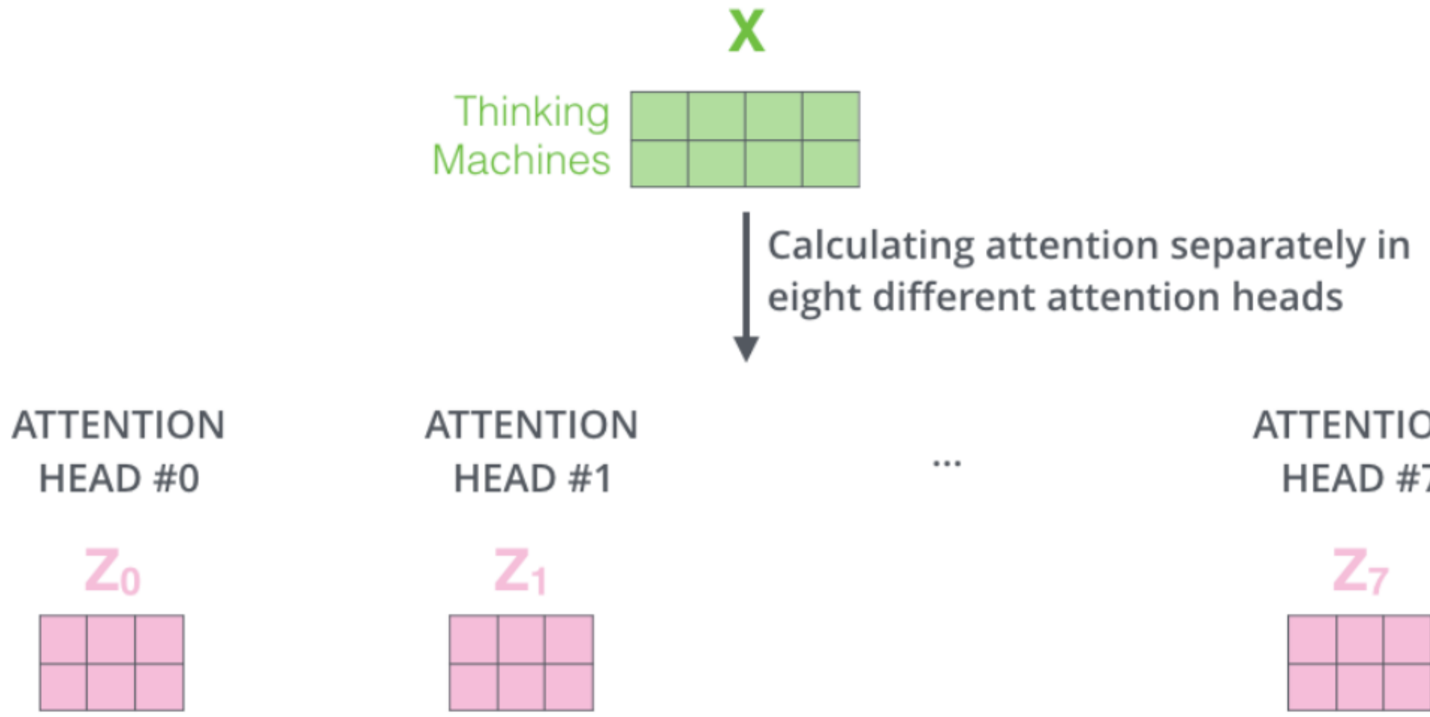


Multi-headed attention



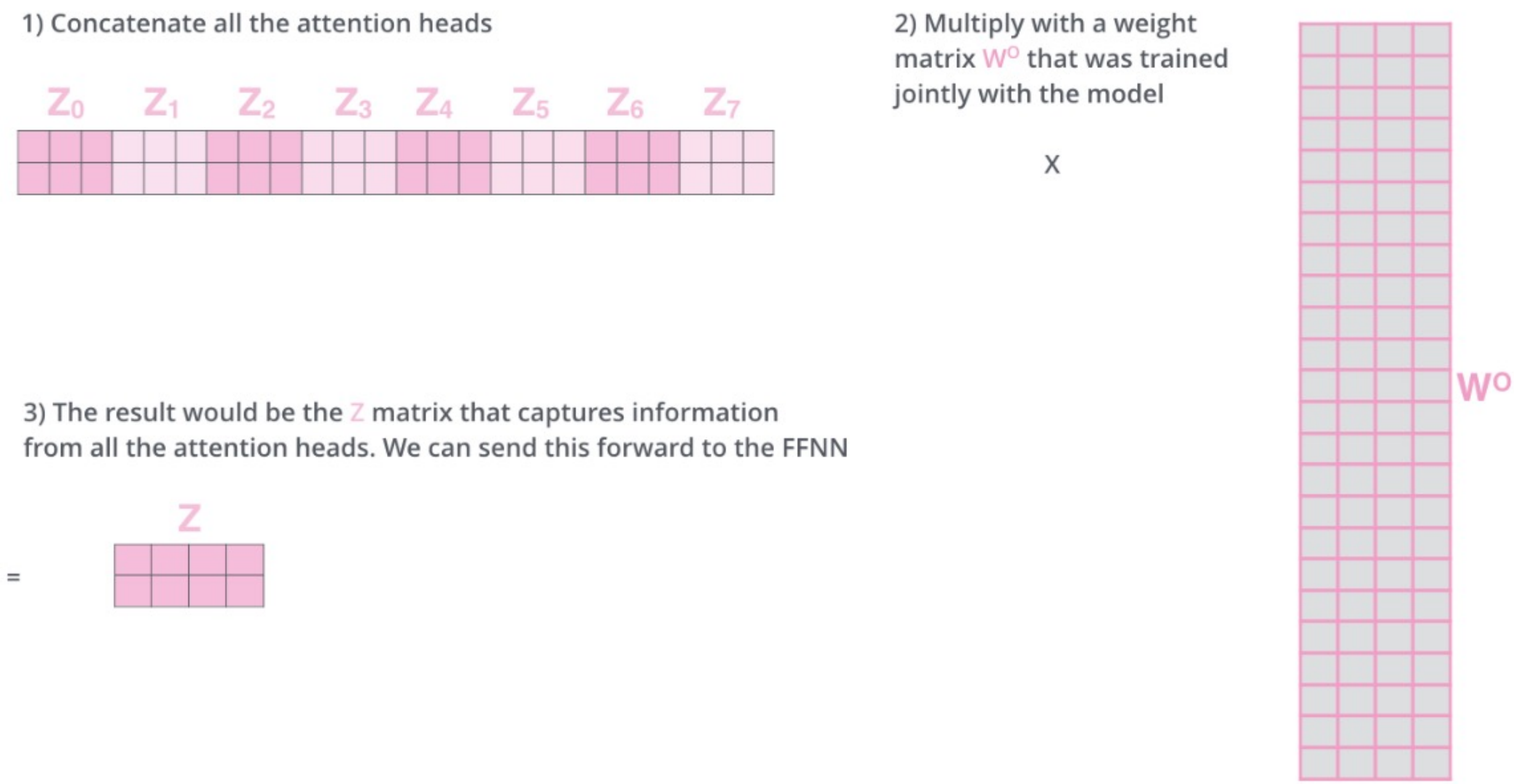
Source: http://jalammar.github.io/illustrated-transformer/

Multi-headed attention - 8 heads



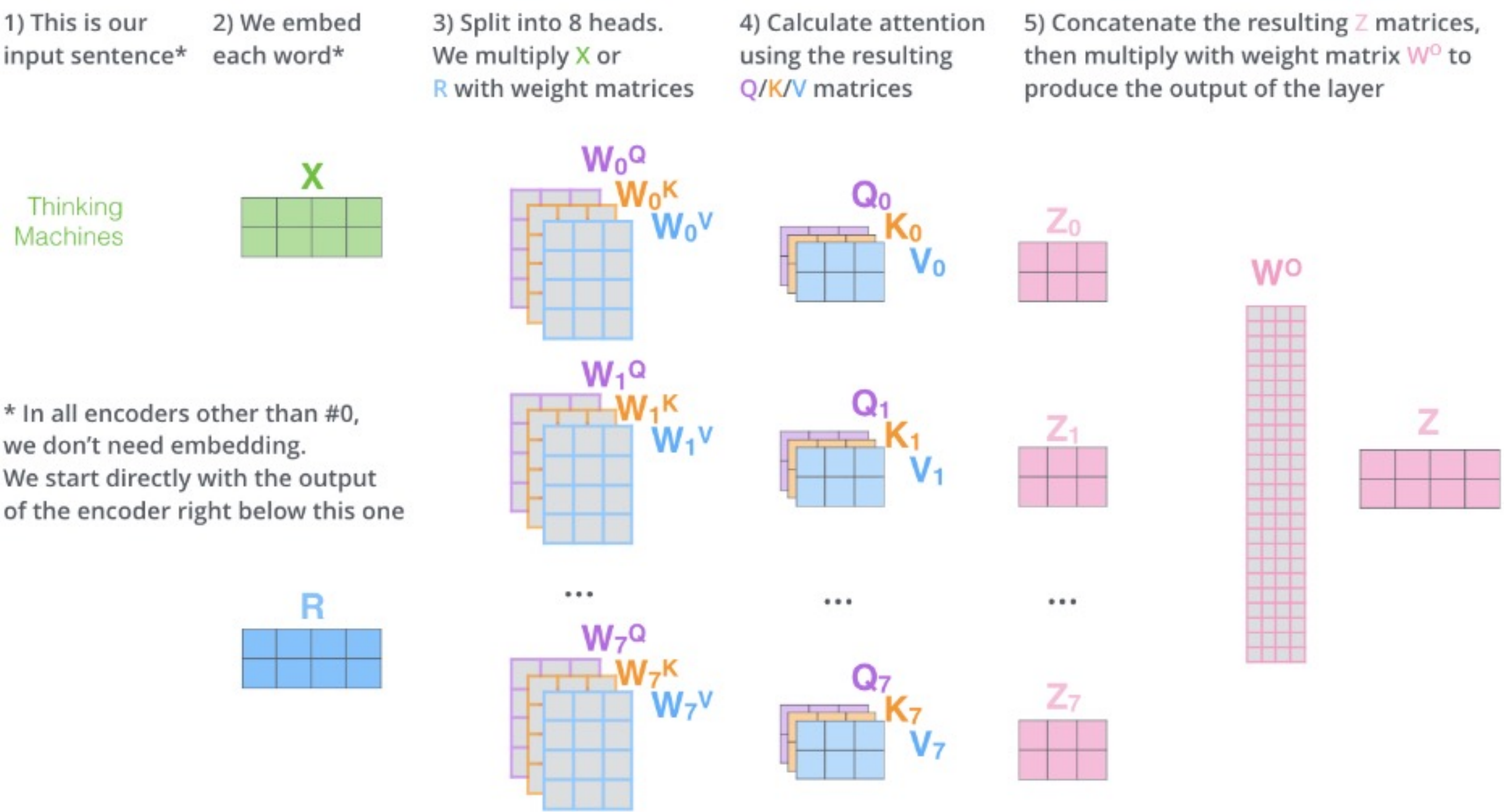
Source: http://jalammar.github.io/illustrated-transformer/

Multi-headed attention - aggregation



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

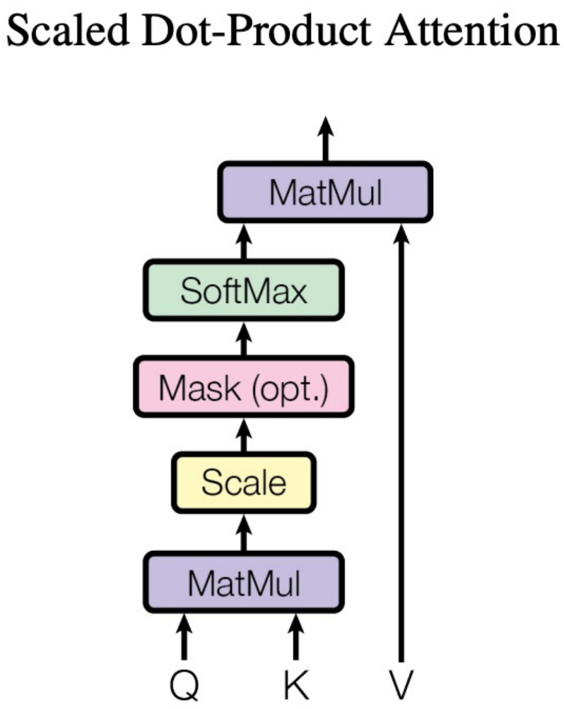
Multi-headed attention - summary

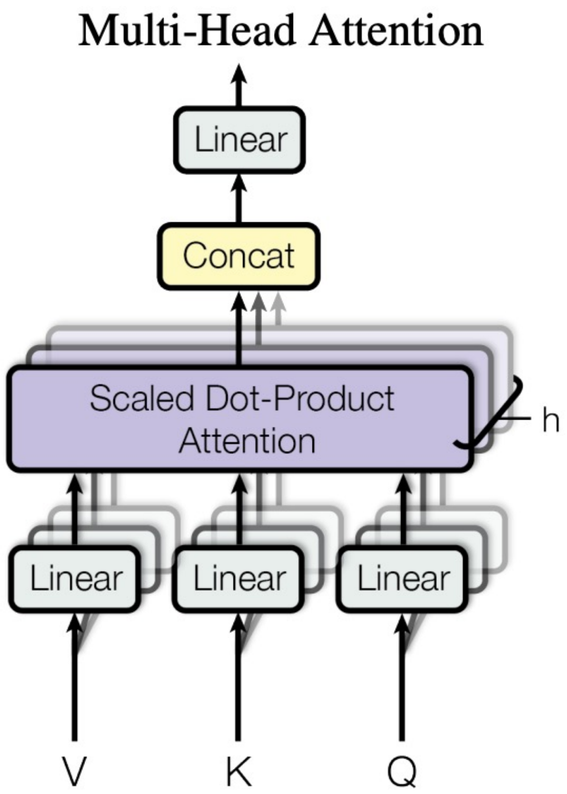


Multi-headed attention

- We define multiple attention “heads” through multiple Q, K, V matrices.
- 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.

Multi-headed attention



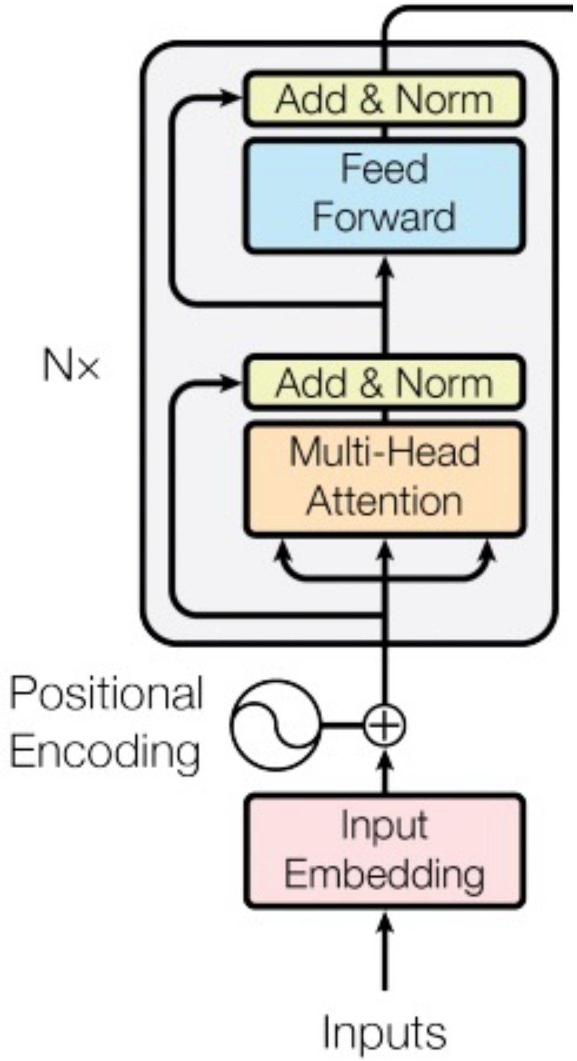


Multi-Head Attention consists of several attention layers running in parallel.

The Transformer Encoder

- Two optimization tricks that end up being :
  - **Residual Connections**
  - **Layer Normalization**

- In most Transformer diagrams, these are often written together as “Add & Norm”

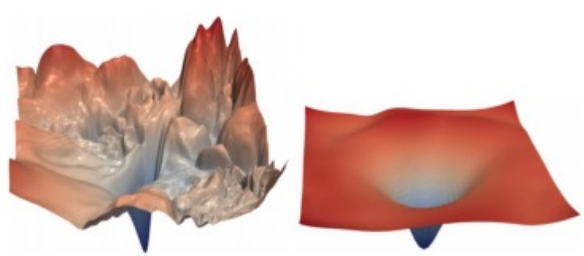


The Transformer: Residual connections [He et al., 2016]

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

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

- Gradient is great through the residual connection; it’s **1**!



[no residuals] [residuals]

Loss landscape visualization, [Li et al., 2018, on a ResNet]

The Transformer: Layer normalization [Ba et al., 2016]

- Layer normalization is a trick to help models train faster.
- Idea: cut down on uninformative variation in hidden vector values by normalizing to zero mean and standard deviation within each layer.
  - LayerNorm's success may be due to its normalizing gradients [Xu et al., 2019]

- Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.
- Let $\mu = \sum_{j=1}^{d} x_j$; this is the mean; $\mu \in \mathbb{R}$.
- Let $\sigma = \sqrt{\frac{1}{d} \sum_{j=1}^{d} (x_j - \mu)^2}$; this is the standard deviation; $\sigma \in \mathbb{R}$.
- Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters. (Can omit!)
- Then layer normalization computes:

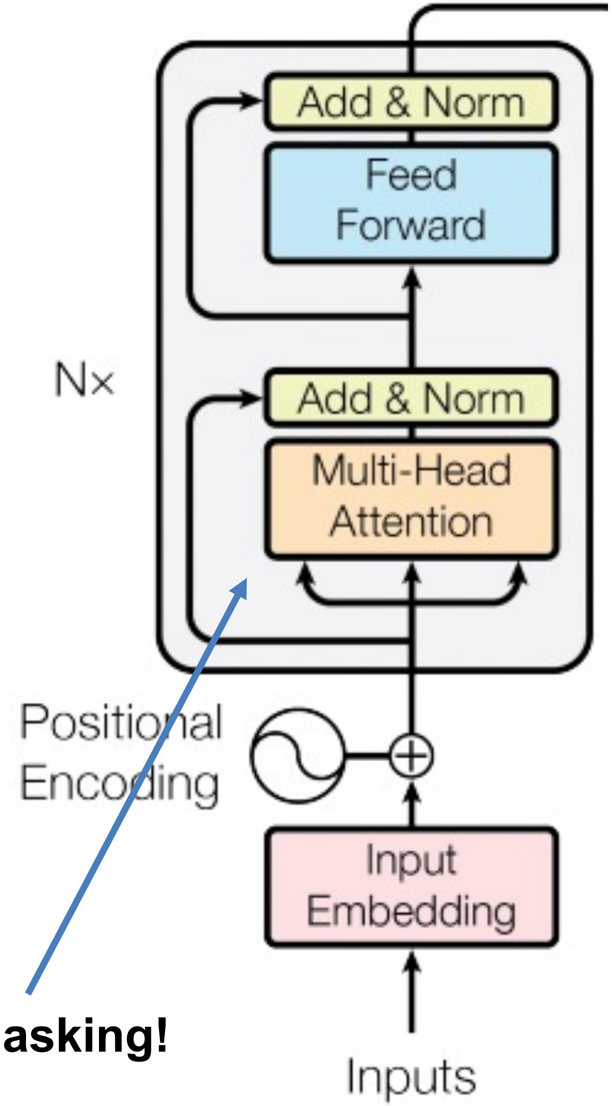
$$\text{output} = \frac{x - \mu}{\sqrt{\sigma} + \epsilon} * \gamma + \beta$$

Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias

The Transformer Encoder

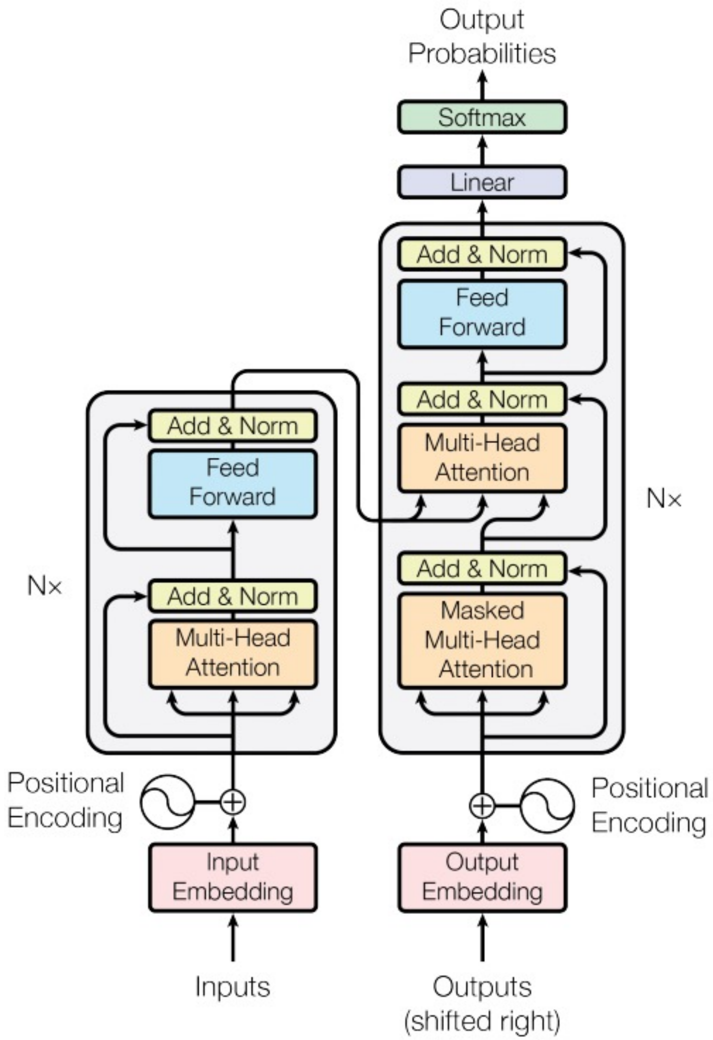
- The Transformer Decoder constrains to uni-directional context, as for language models.
- What if we want bi-directional context, like in a bidirectional RNN?
- This is the Transformer Encoder. Note that we don't have the masking in the multi-head attention in Encoder.



No masking!

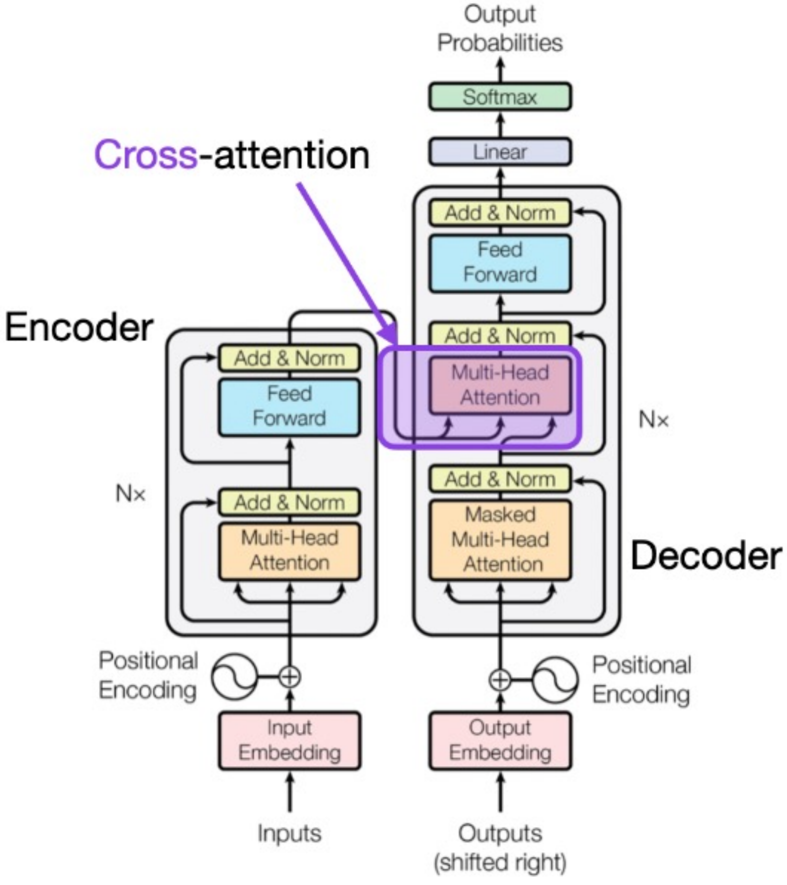
The Transformer Encoder-Decoder

- For seq2seq formats, we often use a Transformer Encoder-Decoder:
  - A normal Transformer Encoder.
  - A modified decoder to perform cross-attention to the output of the Encoder.



Cross-attention

- We saw that self-attention is when keys, queries, and values come from the same source.
- Let $h_1, \dots, h_n$ be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathbb{R}^d$
- Let $z_1, \dots, z_n$ be input vectors from the Transformer **decoder**, $z_i \in \mathbb{R}^d$
- Then keys and values are drawn from the **encoder** (like a memory):
  - $k_i = Kh_i, v_i = Vh_i$.
- And the queries are drawn from the **decoder**, $q_i = Qz_i$.



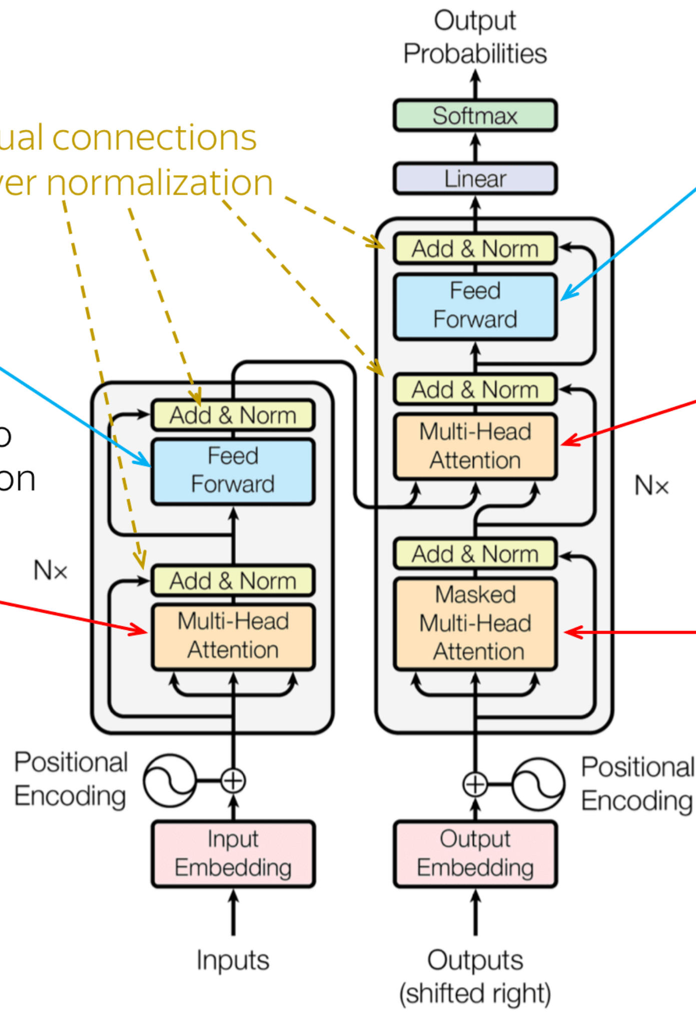
The Transformer Encoder-Decoder

Residual connections and layer normalization

Feed-forward network: after taking information from other tokens, take a moment to think and process this information

Encoder self-attention: tokens look at each other

queries, keys, values are computed from encoder states



Feed-forward network: after taking information from other tokens, take a moment to think and process this information

Decoder-encoder attention: target token looks at the source

queries – from decoder states; keys and values from encoder states

Decoder self-attention (masked): tokens look at the previous tokens

queries, keys, values are computed from decoder states

Great results with Transformers

First, Machine Translation from the original Transformers paper!

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**	$2.3 \cdot 10^{19}$	

Great results with Transformers

Next, document generation!

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 old standard

Transformers all the way down.

What would we like to fix about the Transformer?

- Quadratic computation in self-attention:
  - Computing all pairs of interactions means our computation grows quadratically with the sequence length!
  - For recurrent models, it only grew linearly!
- Position representations:
  - Are simple absolute indices the best we can do to represent position?
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(n^2d)$, where n is the sequence length, and d is the dimensionality.

$$XQ \quad K^\top X^\top = XQK^\top X^\top \in \mathbb{R}^{n\times n}$$

Need to compute all pairs of interactions!
$O(n^2d)$

- Think of d as around $\mathbf{1{,}000}$ (though for large language models it's much larger!).
  - So, for a single (shortish) sentence, $n \leq 30$; $n^2 \leq \mathbf{900}$.
  - In practice, we set a bound like $n = 512$.
  - But what if we'd like $\boldsymbol{n} \geq \mathbf{50,\ 000}$? For example, to work on long documents?

Do we need to remove the quadratic cost of attention?

- As Transformers grow larger, a larger and larger percent of compute is outside the self-attention portion, despite the quadratic cost. Such as the matrix multiplication, multi-head.

- In practice, almost no large Transformer language models use anything but the quadratic cost attention we've presented here.
    - The cheaper methods tend not to work as well at scale.

- So, is there no point in trying to design cheaper alternatives to self-attention?

- Or would we unlock much better models with much longer contexts (>100k tokens?) to make it more efficient and scalable for processing long sequences?

Do Transformer Modifications Transfer?

- *"The research community has proposed copious modifications to the Transformer architecture since it was introduced over three years ago, relatively few of which have seen widespread adoption.*

- *In this paper, we comprehensively evaluate many of these modifications in a shared experimental setting that covers most of the common uses of the Transformer in natural language processing.*

- *Surprisingly, we find that most modifications do not meaningfully improve performance."*

Do Transformer Modifications Transfer Across Implementations and Applications?

Sharan Narang[*] Hyung Won Chung Yi Tay William Fedus
Thibault Fevry[†] Michael Matena[†] Karishma Malkan[†] Noah Fiedel
Noam Shazeer Zhenzhong Lan[†] Yanqi Zhou Wei Li
Nan Ding Jake Marcus Adam Roberts Colin Raffel[†]

Readings

1. Attention Is All You Need
2. The Illustrated Transformer
3. Transformer (Google AI blog post)
4. Layer Normalization

Your feedback is important!

Please participate in the below survey (3-5 minutes) to help provide feedback for this lecture:
https://www.surveymonkey.com/r/GM8KWRV

References

[1] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2018

[2] RoBERTa: A Robustly Optimized BERT Pretraining Approach. 2019

[3] DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. 2019

[4] GPT2: Language Models are Unsupervised Multitask Learners. 2018

[5] BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. 2019

[6] T5: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. 2019

[7] GPT: Improving Language Understanding by Generative Pre-Training. 2018

STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®
1870

stevens.edu

Thank You