

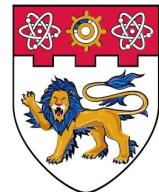


# Natural Language Processing

SC4002 / CE4045 / CZ4045  
by Wang Wenya

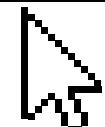
Email: [wangwy@ntu.edu.sg](mailto:wangwy@ntu.edu.sg)

Contents adapted from Dr. Joty Shafiq's notes



NANYANG  
TECHNOLOGICAL  
UNIVERSITY  
**SINGAPORE**

**Click here  
for Lecture 4**





# Modules we will cover

## ML & DL

Introduction to  
machine/deep learning

### Transformer

Attention mechanism,  
encoder/decoder



### Pretraining

Masking, natural  
language generation

## Word

Word vectors,  
language modeling

## Sequence

Sequence modeling,  
seq2seq learning

## Prompting

Prompts, in-context  
learning



# Outline for today

**01** From RNNs to Attentions

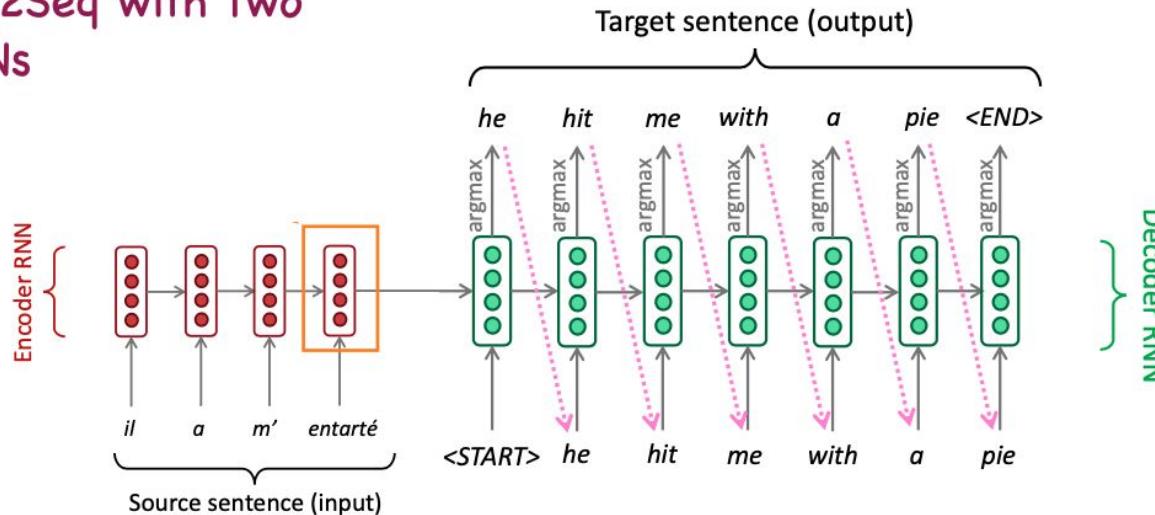
**02** Transformer Model

**03** Applications of Transformers



# Seq2Seq Modeling with RNNs (Revisit)

- Seq2Seq with Two RNNs



Encoder produces an encoding of the source sentence.

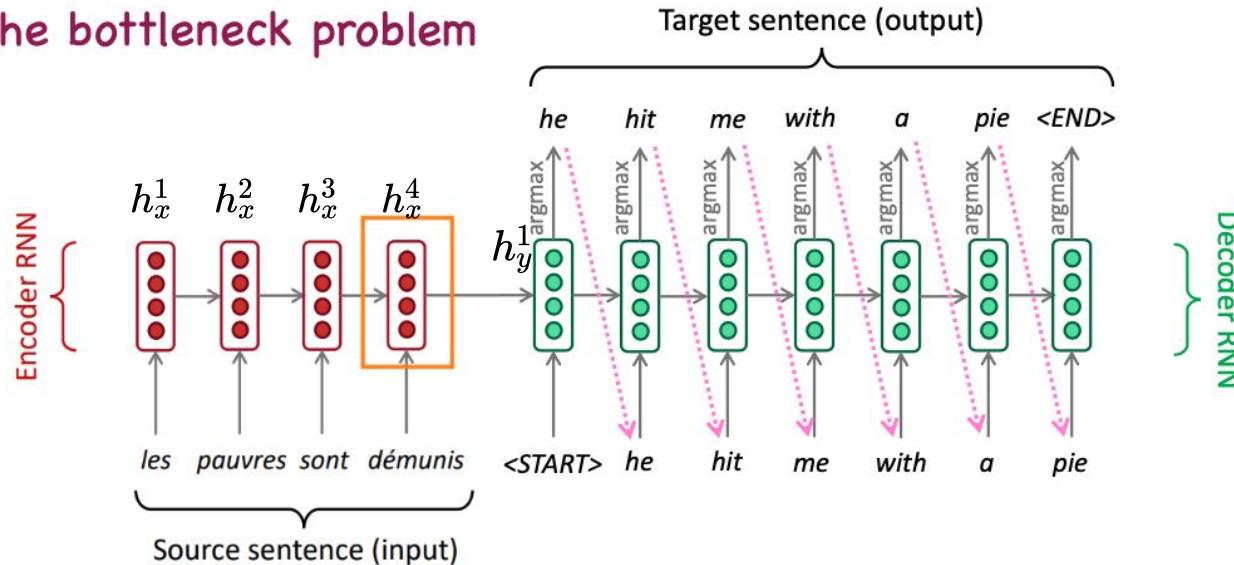
Provides Initial hidden state for Decoder

Decoder RNN is a **Conditional Language Model** that generates target sentence, conditioned on encoding.



# Seq2Seq Modeling with RNNs (Revisit)

## ● The bottleneck problem



Encoding of the source sentence.  
This needs to capture all  
information about the source  
sentence. Information bottleneck!

$$h_y^1 = f(W_{ye} \cdot e_{\text{<START>}} + W_{yh} \cdot h_x^4 + b_y)$$

$$h_x^1 = f(W_{xe} \cdot e_{\text{les}} + W_{xh} \cdot h_x^0 + b_x)$$



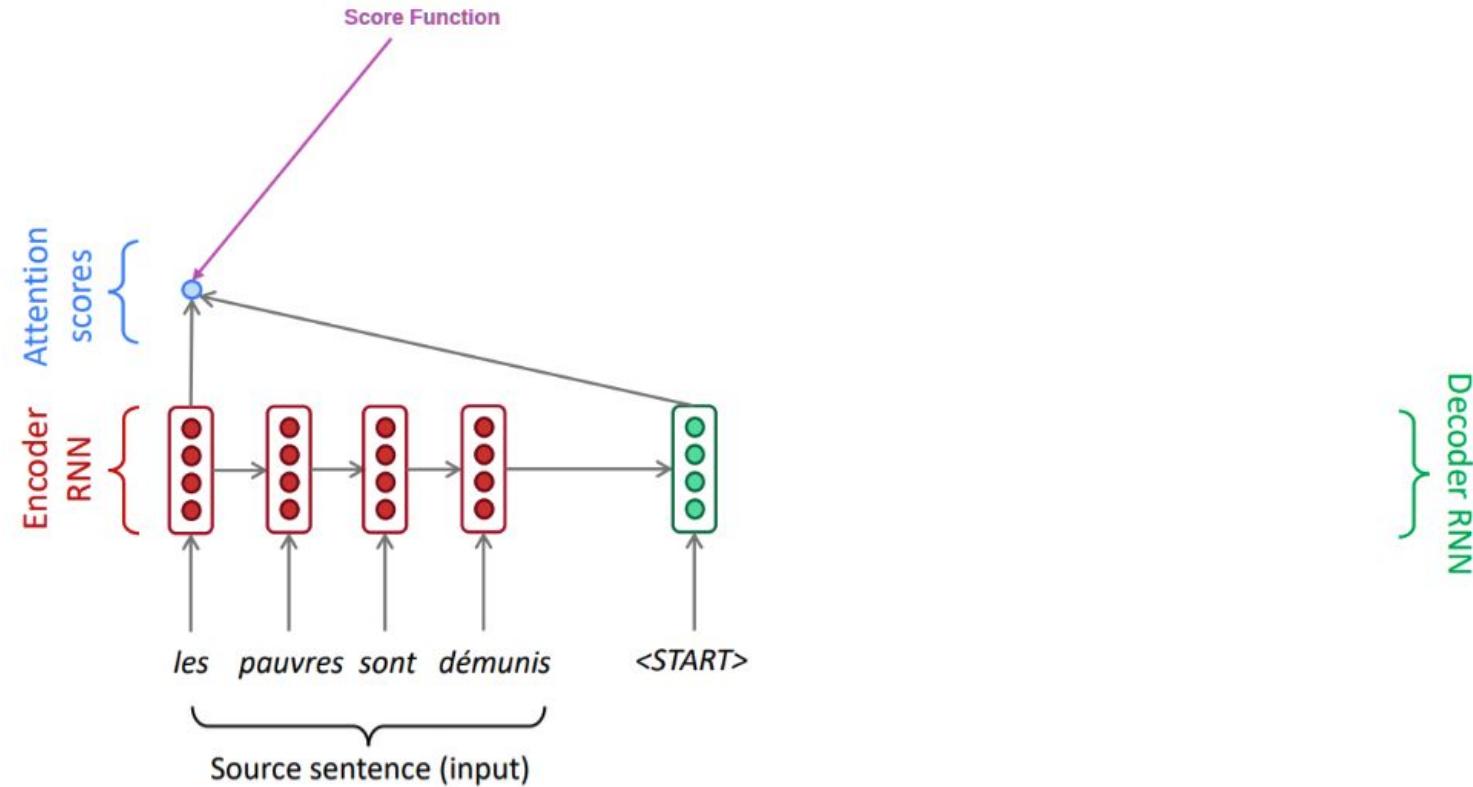
# Solution: Attentions

- **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 the relevant part** of the source sequence



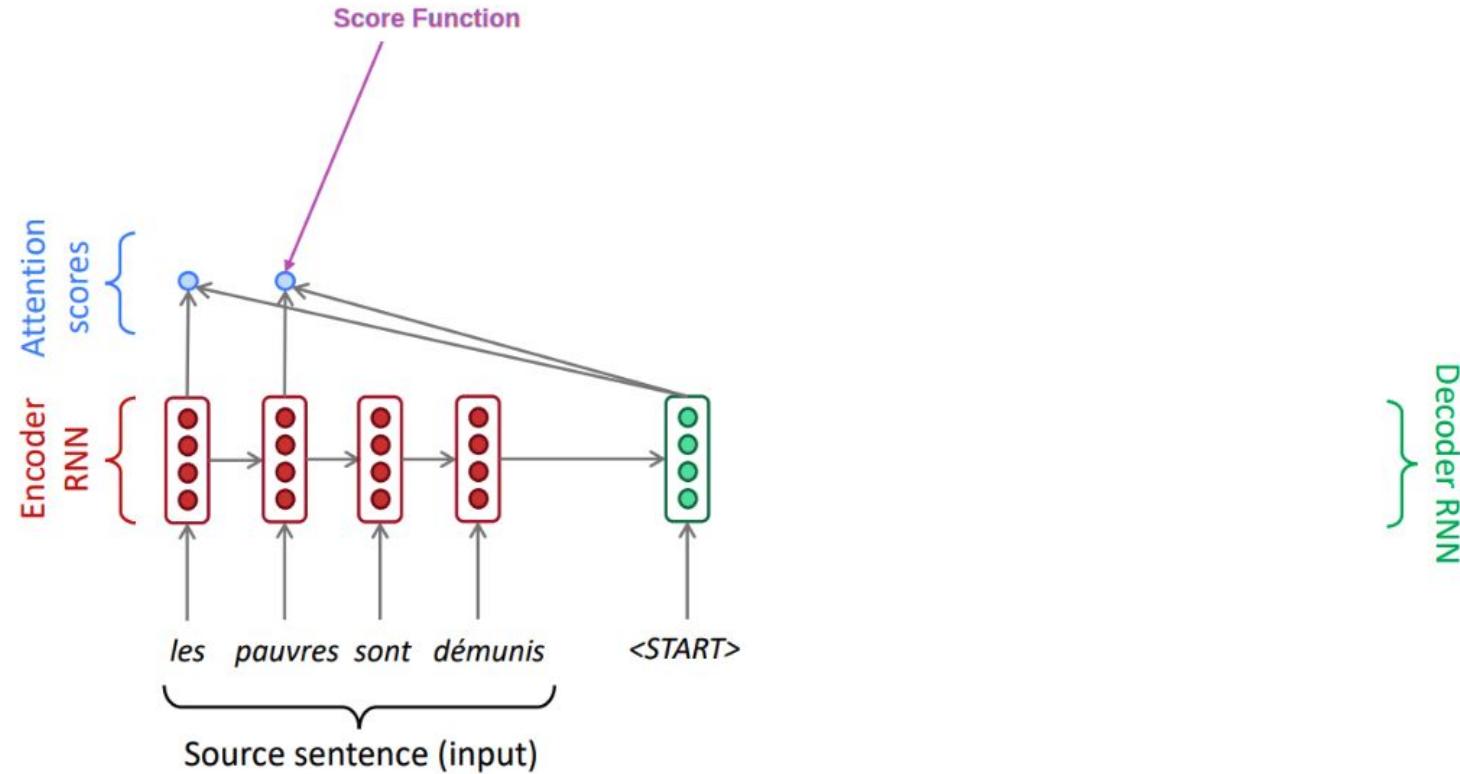


# Solution: Attentions



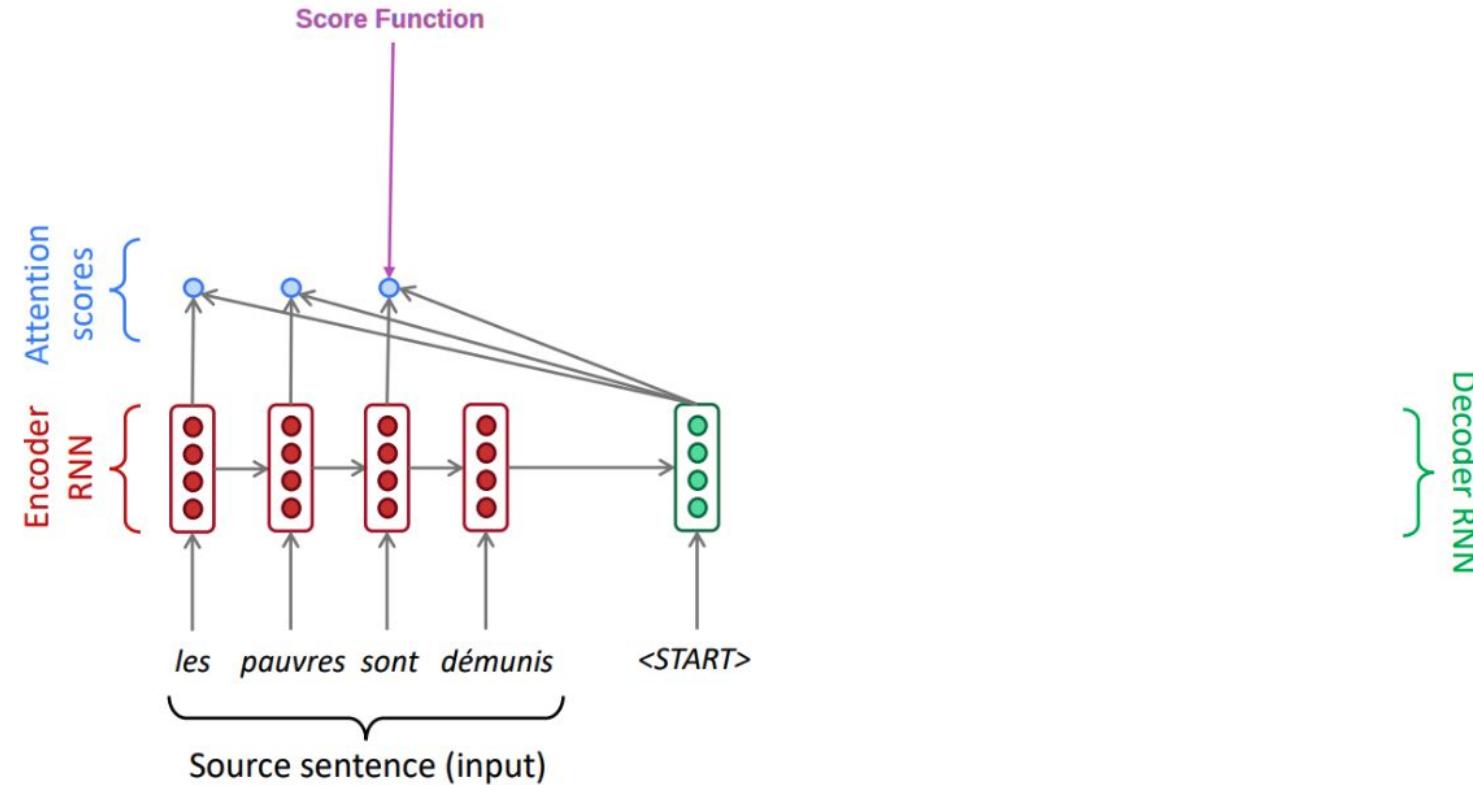


# Solution: Attentions



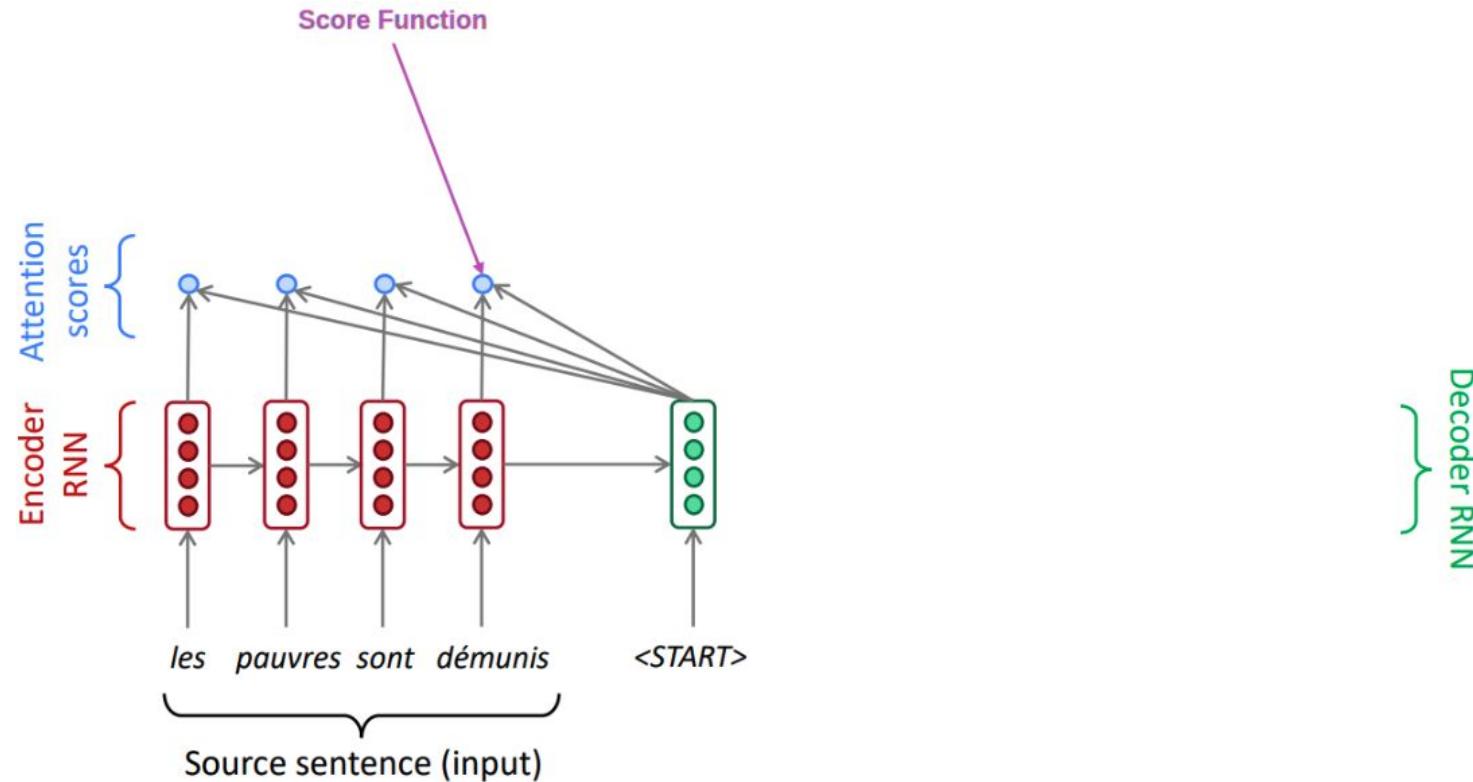


# Solution: Attentions



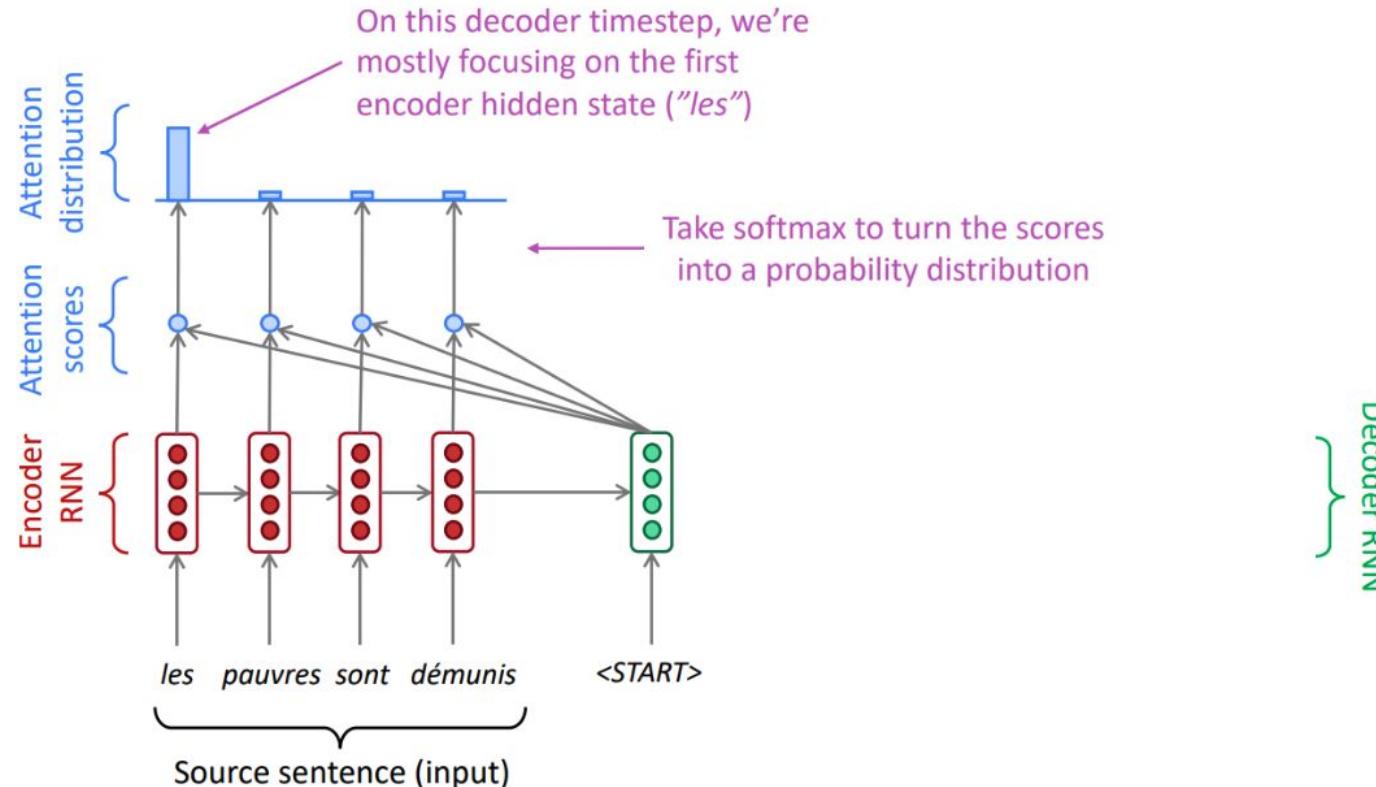


# Solution: Attentions



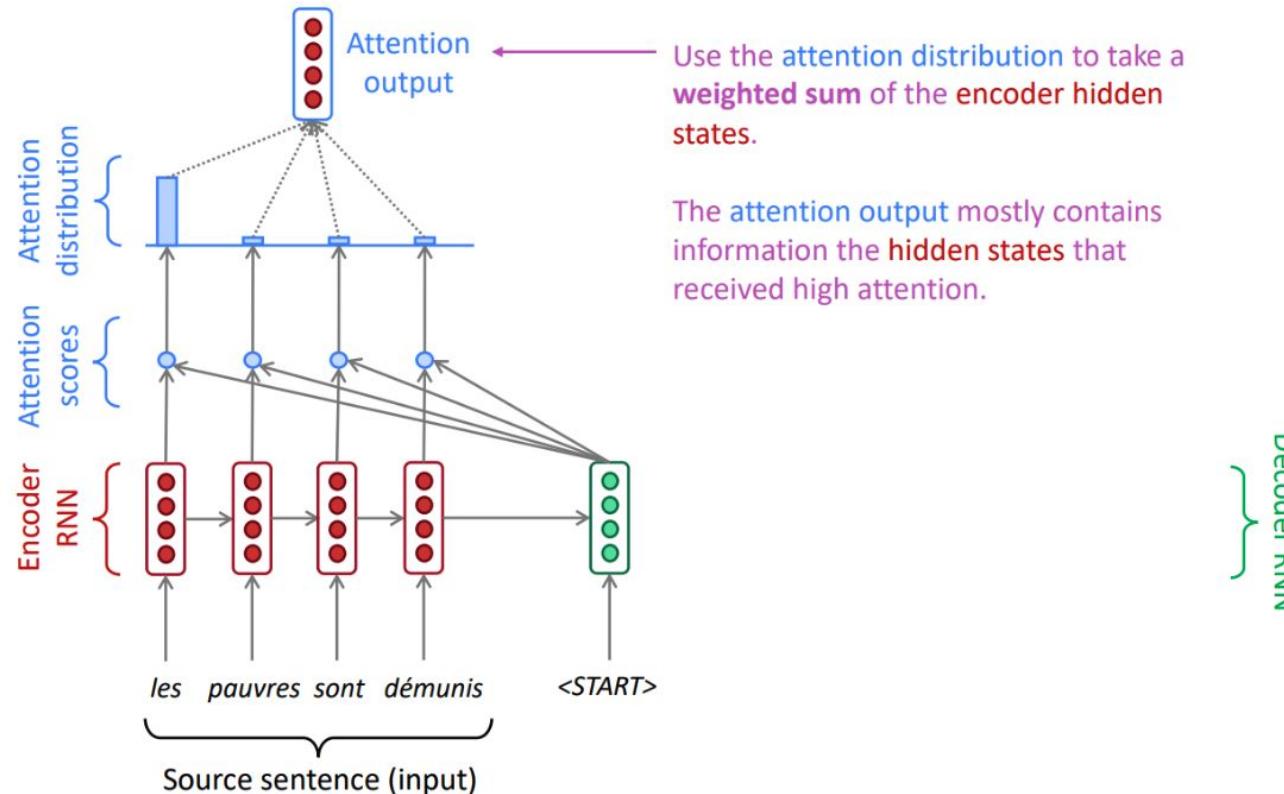


# Solution: Attentions



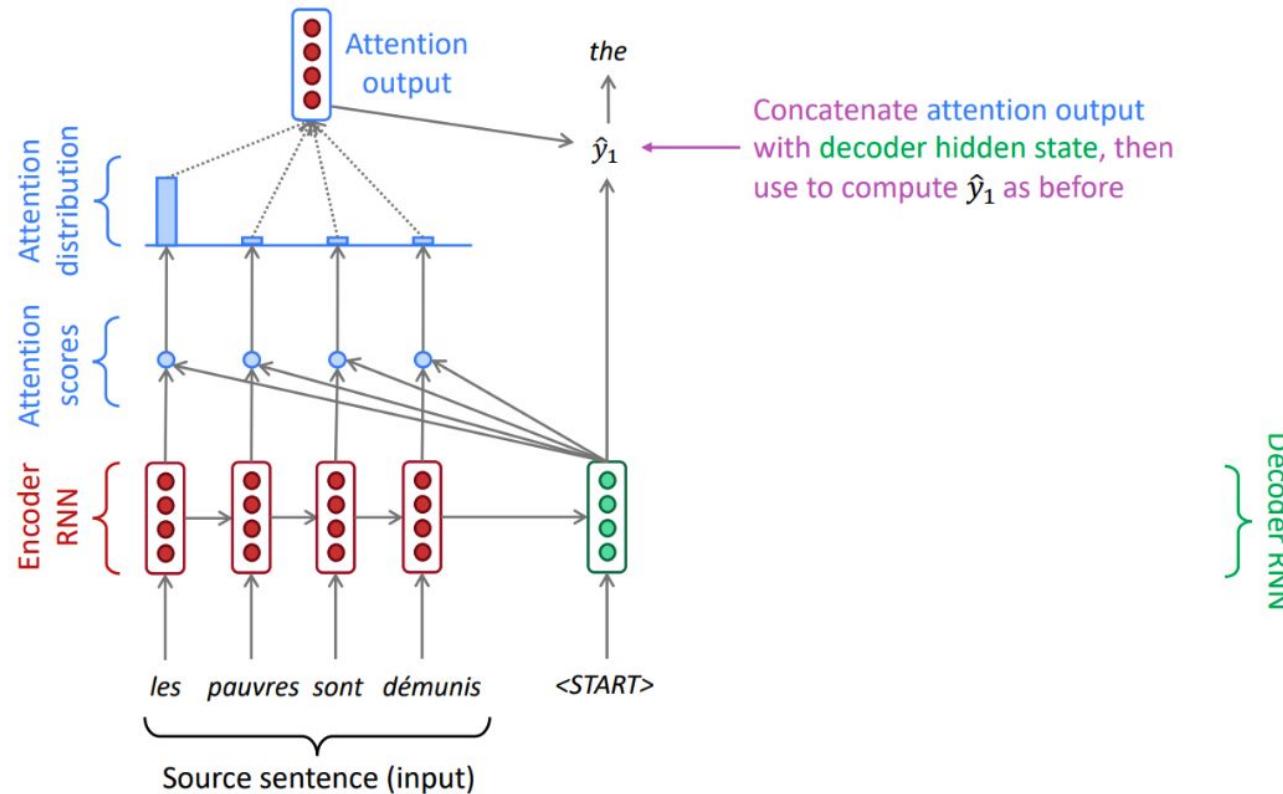


# Solution: Attentions



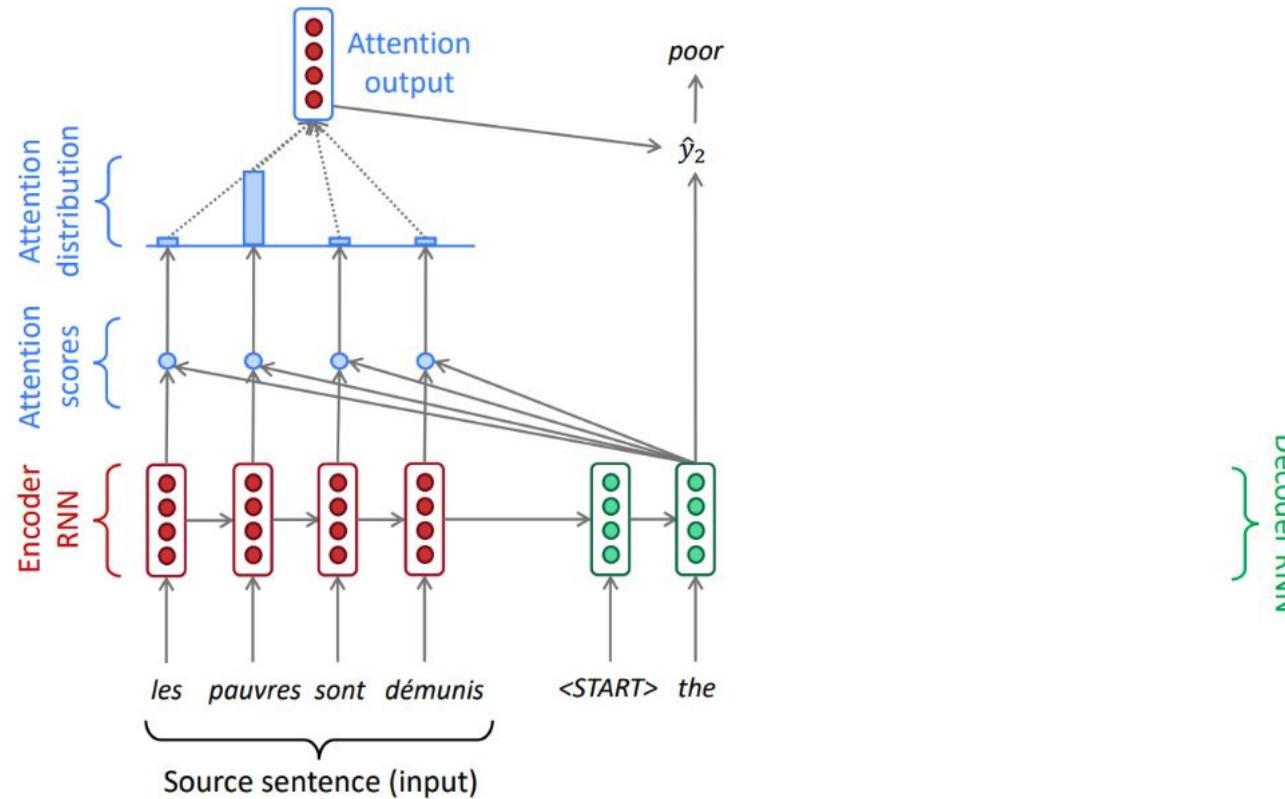


# Solution: Attentions



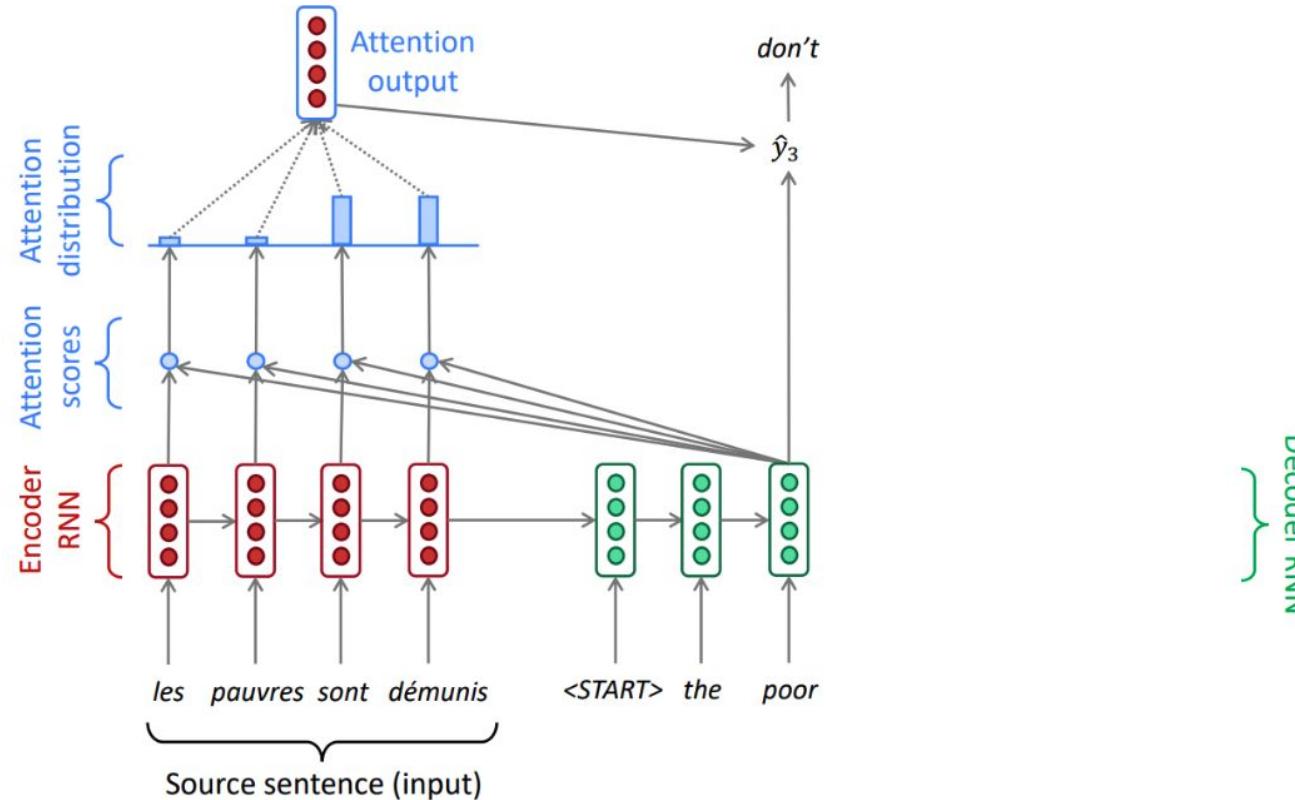


# Solution: Attentions



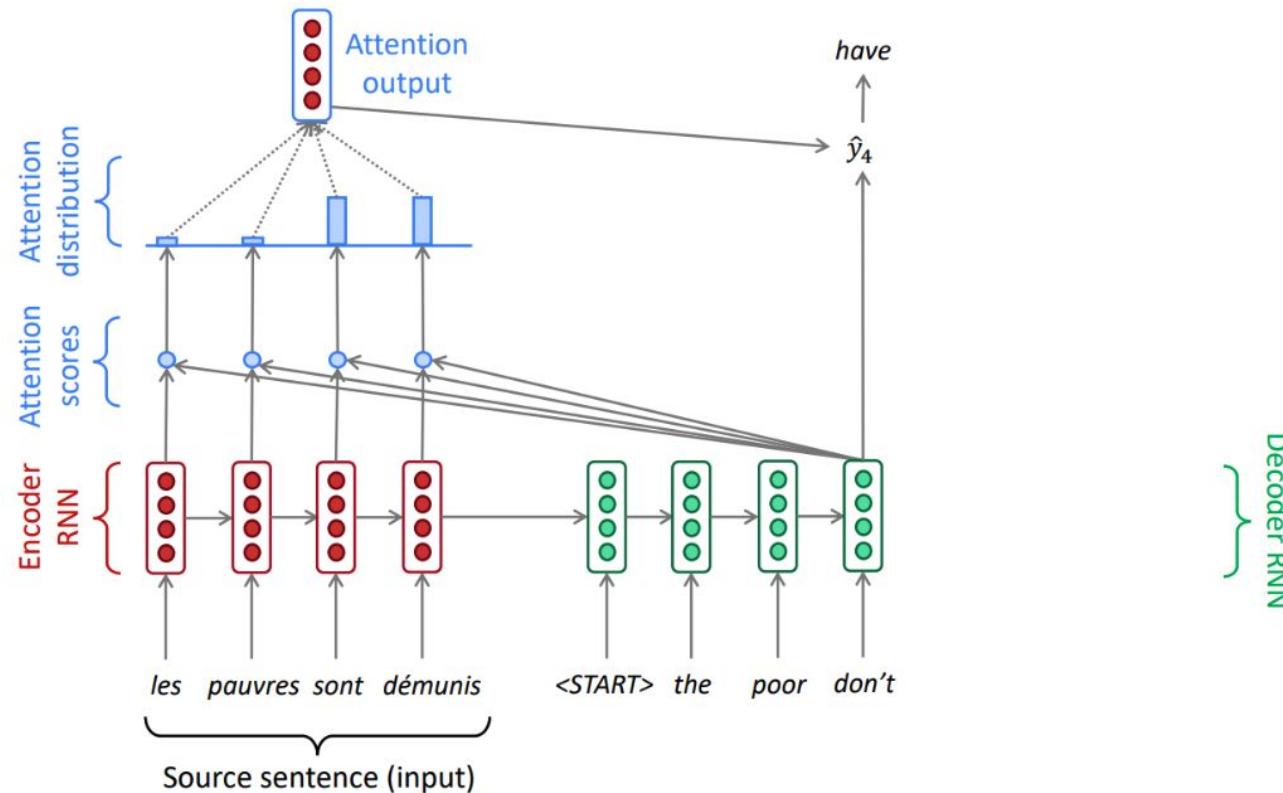


# Solution: Attentions



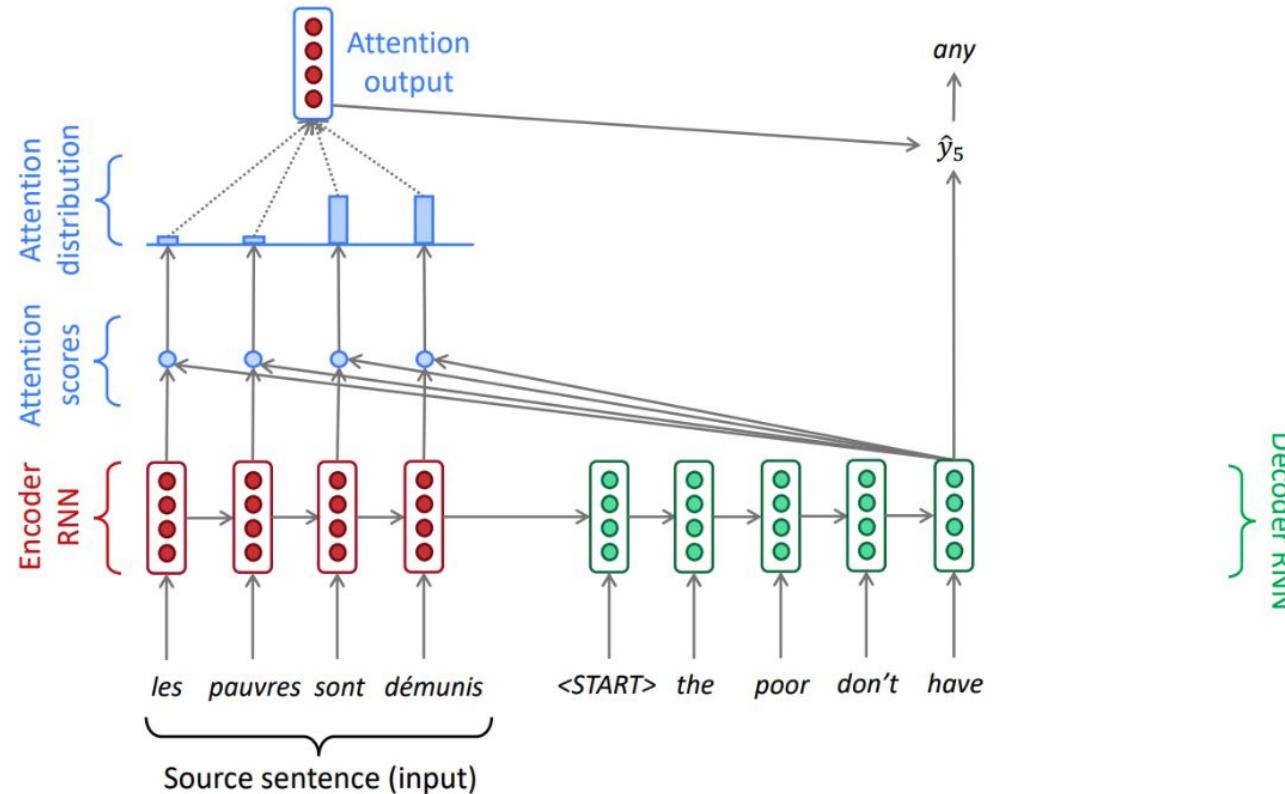


# Solution: Attentions



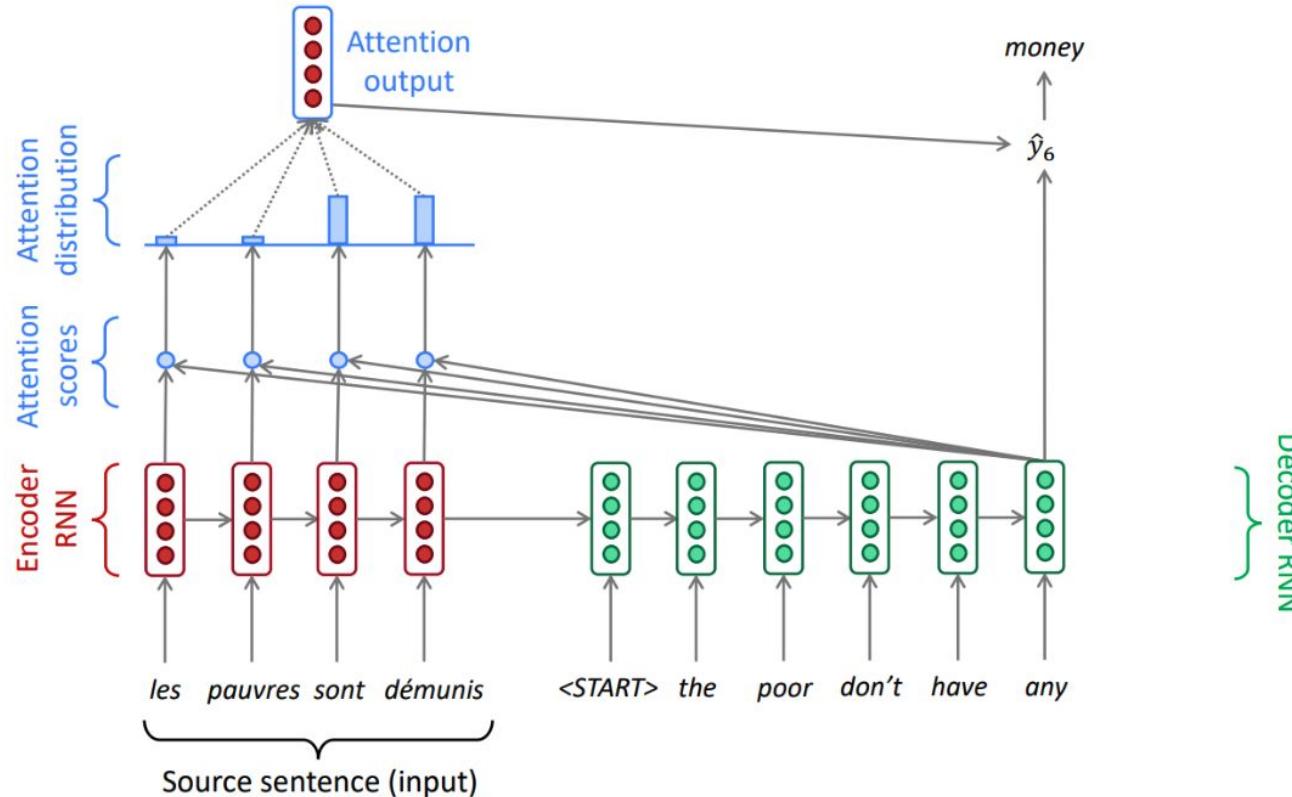


# Solution: Attentions





# Solution: Attentions





# Attentions (Formally)

- 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:

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

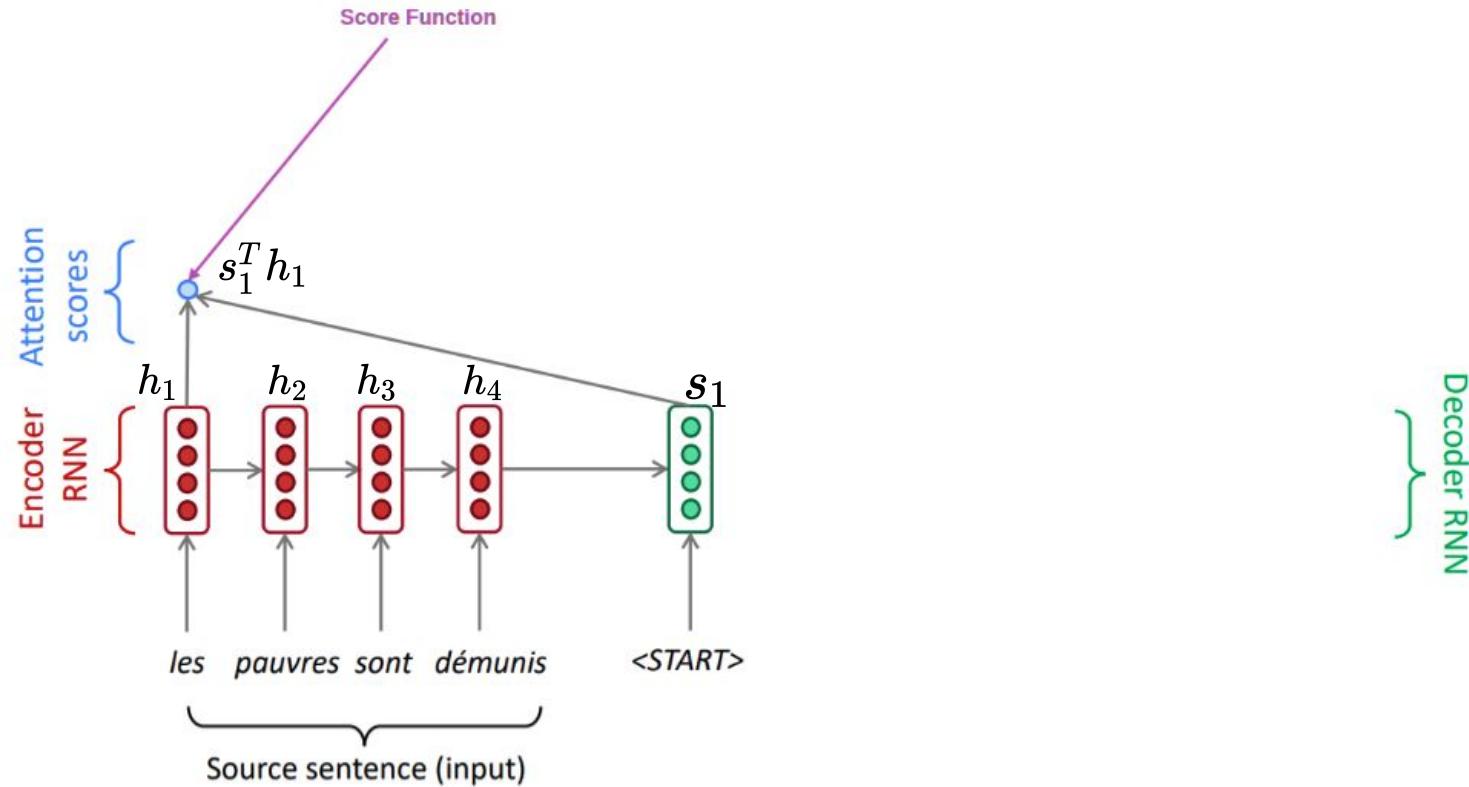
- We take softmax to get the attention distribution  $\alpha^t$  for this step (this is a probability distribution and sums to 1)

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

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

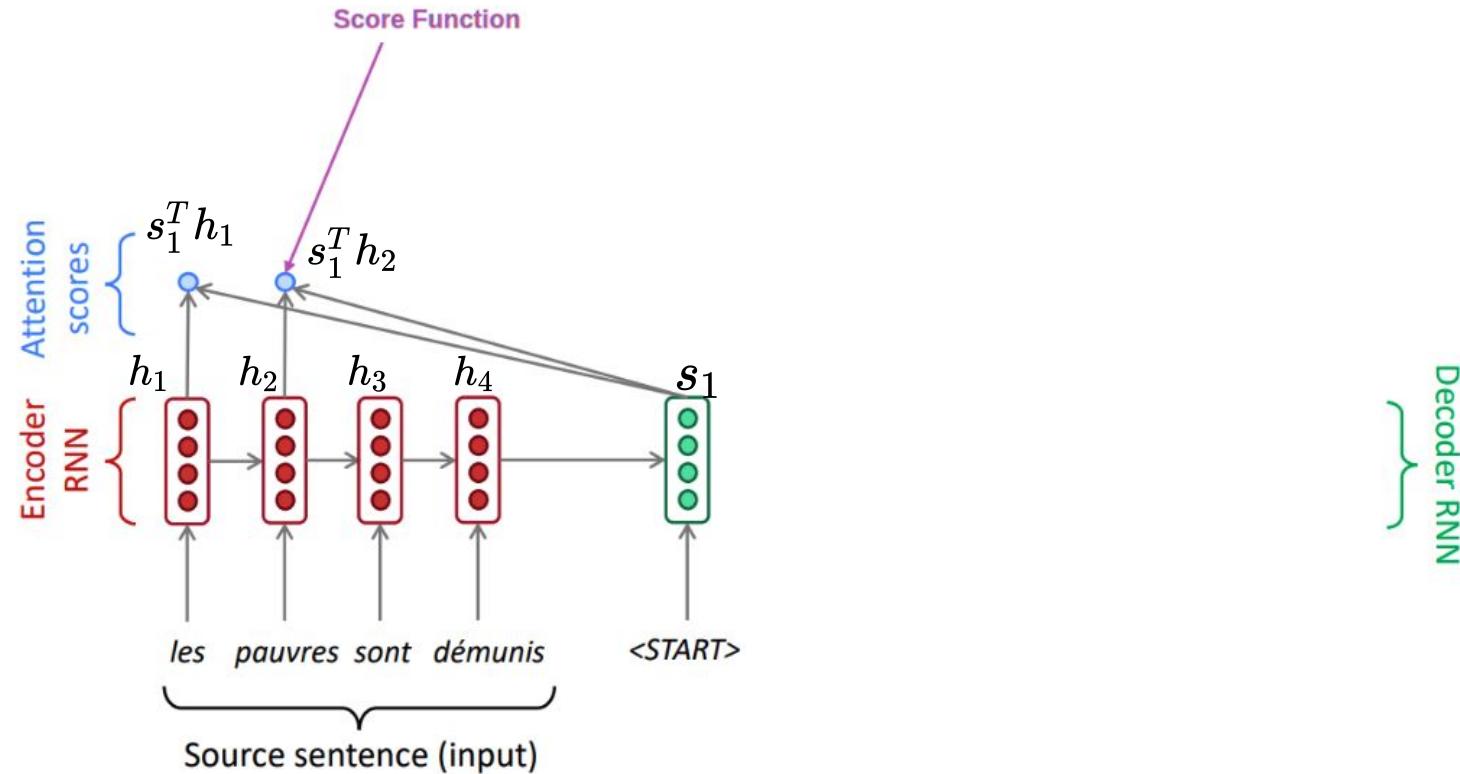
$$a_t = \sum_{i=1}^N \alpha_i^t h_i \in \mathbb{R}^h$$

# Solution: Attentions



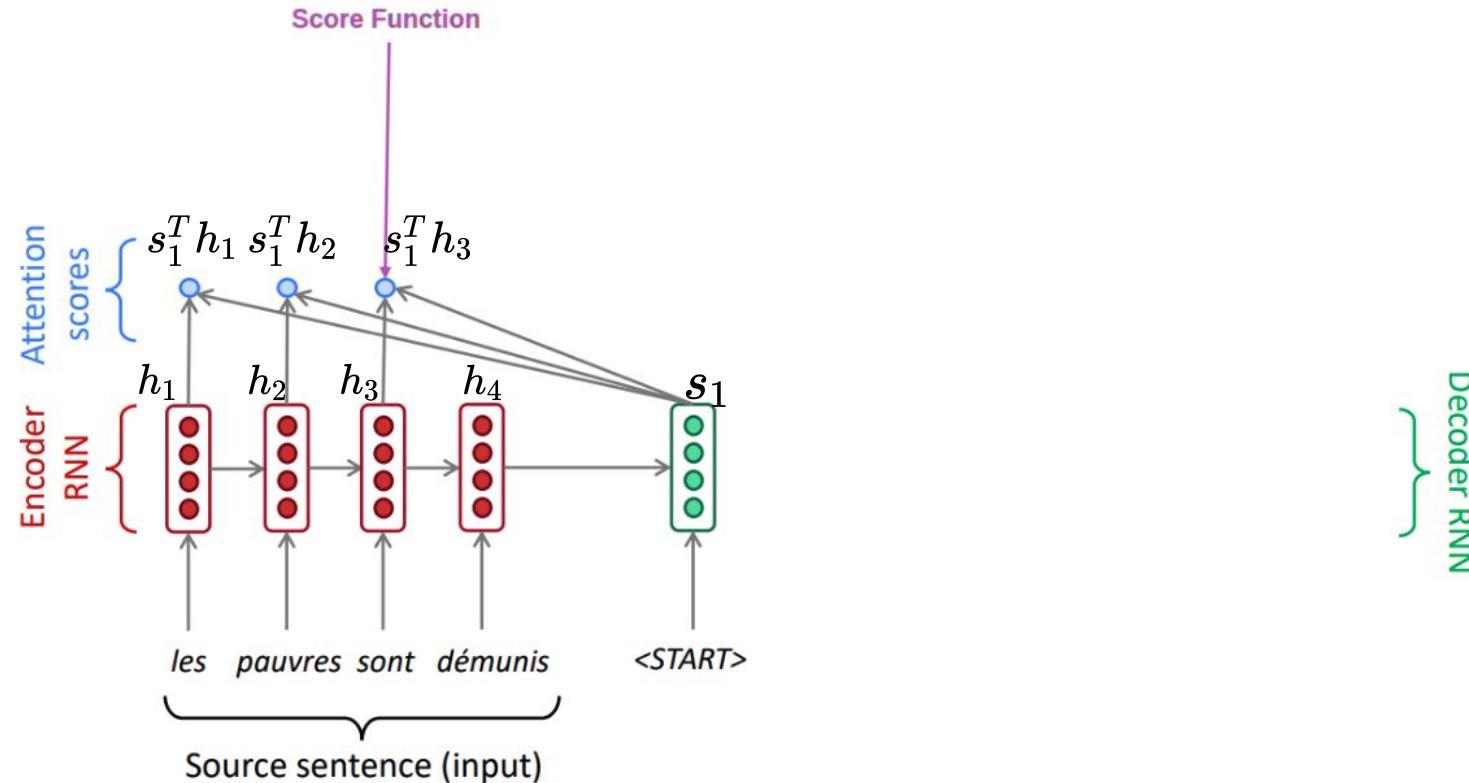


# Solution: Attentions

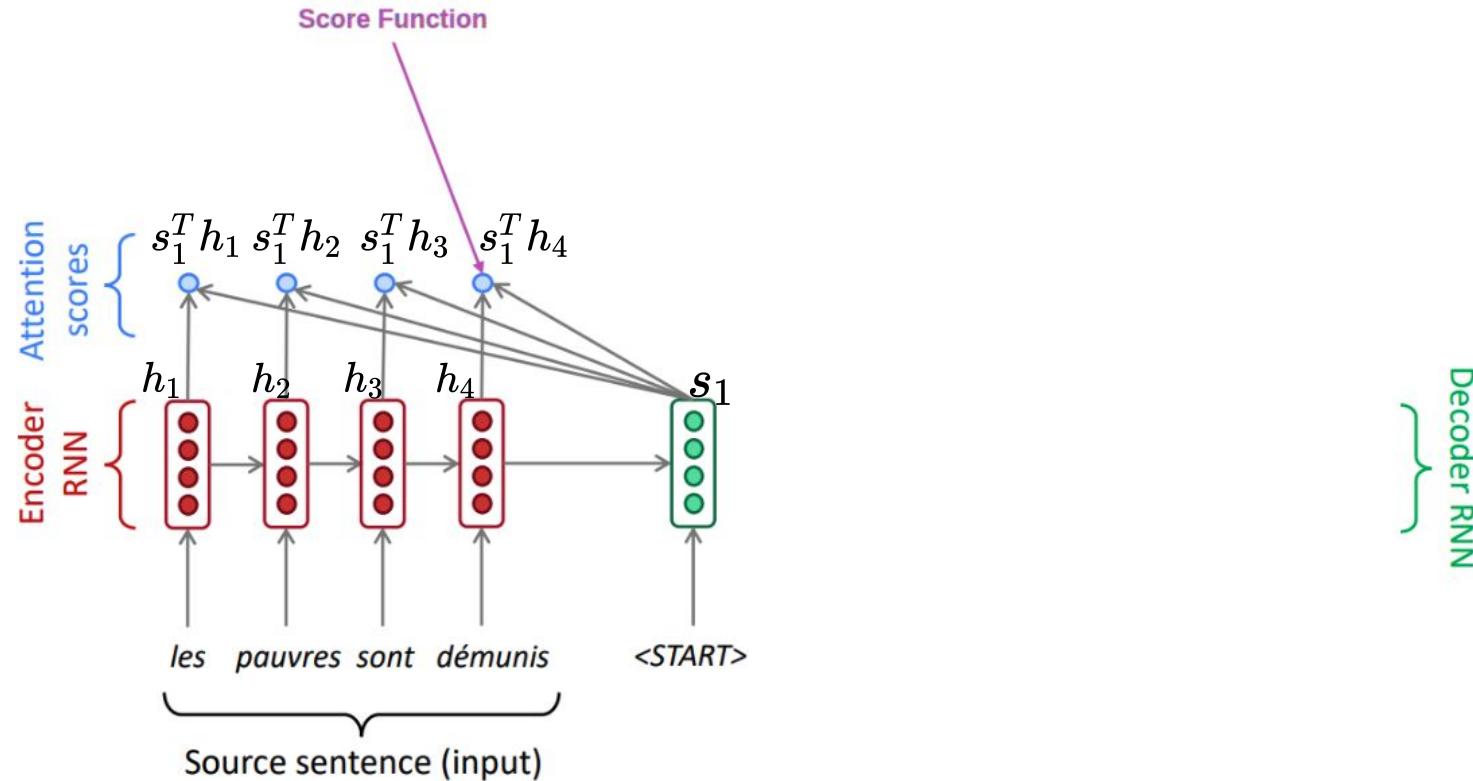




# Solution: Attentions

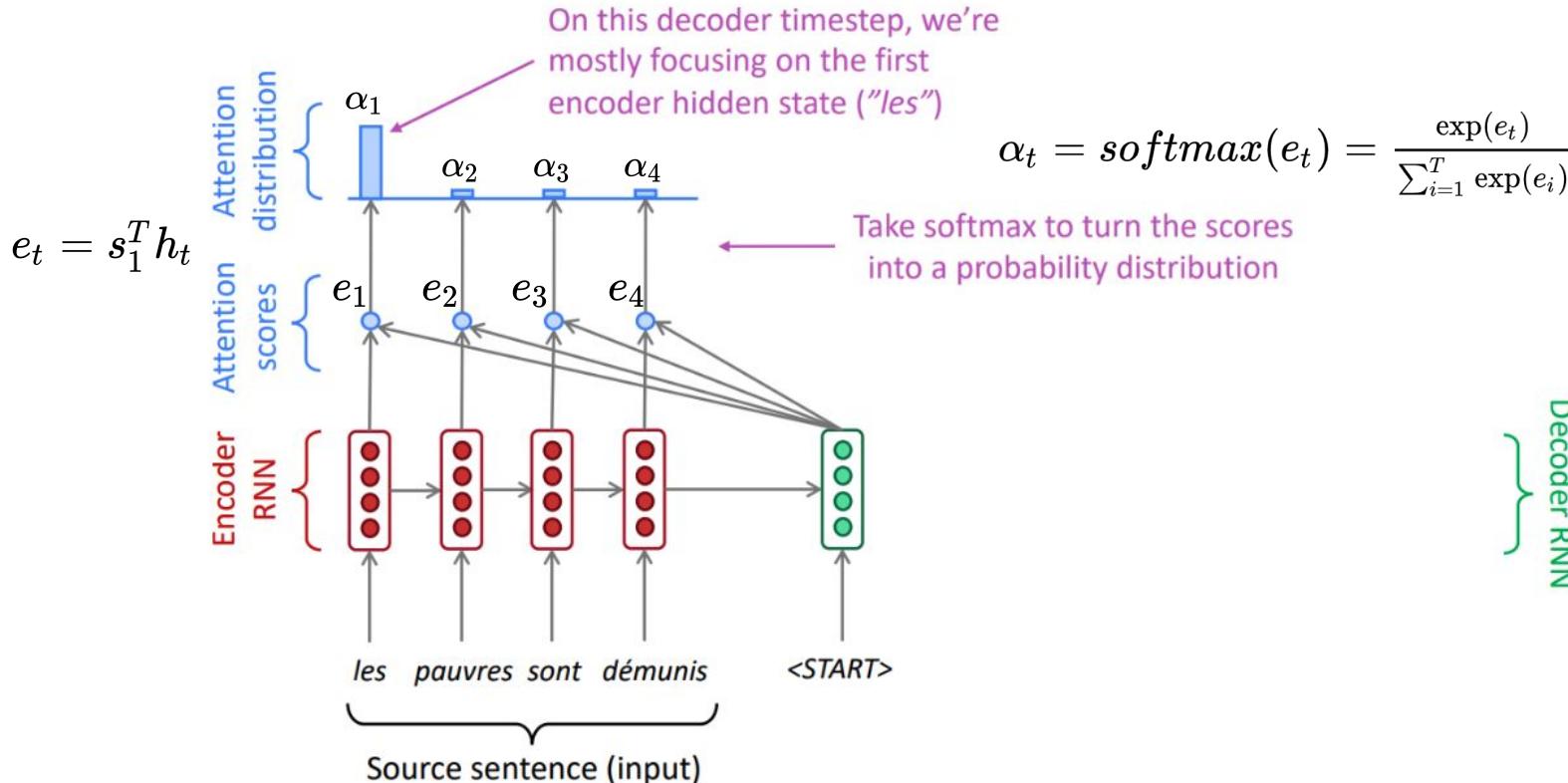


# Solution: Attentions



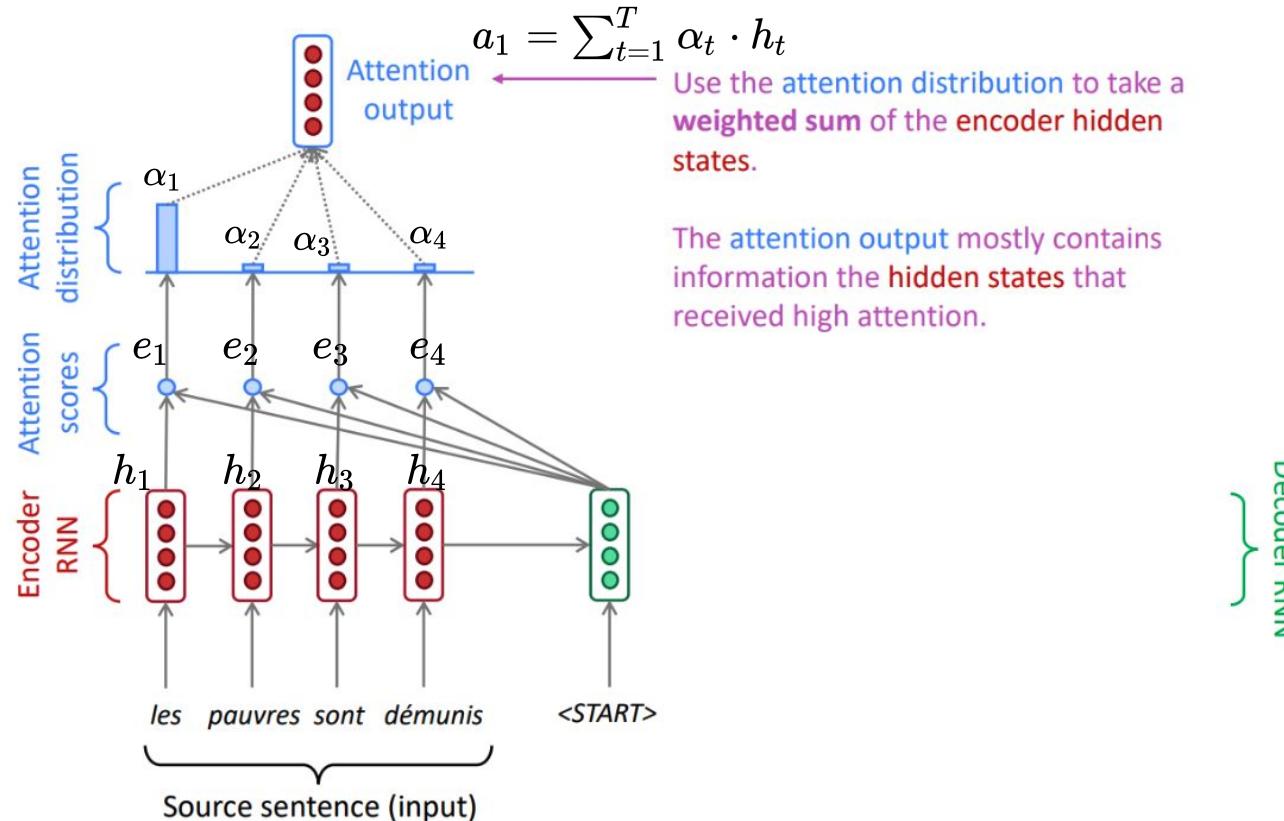


# Solution: Attentions



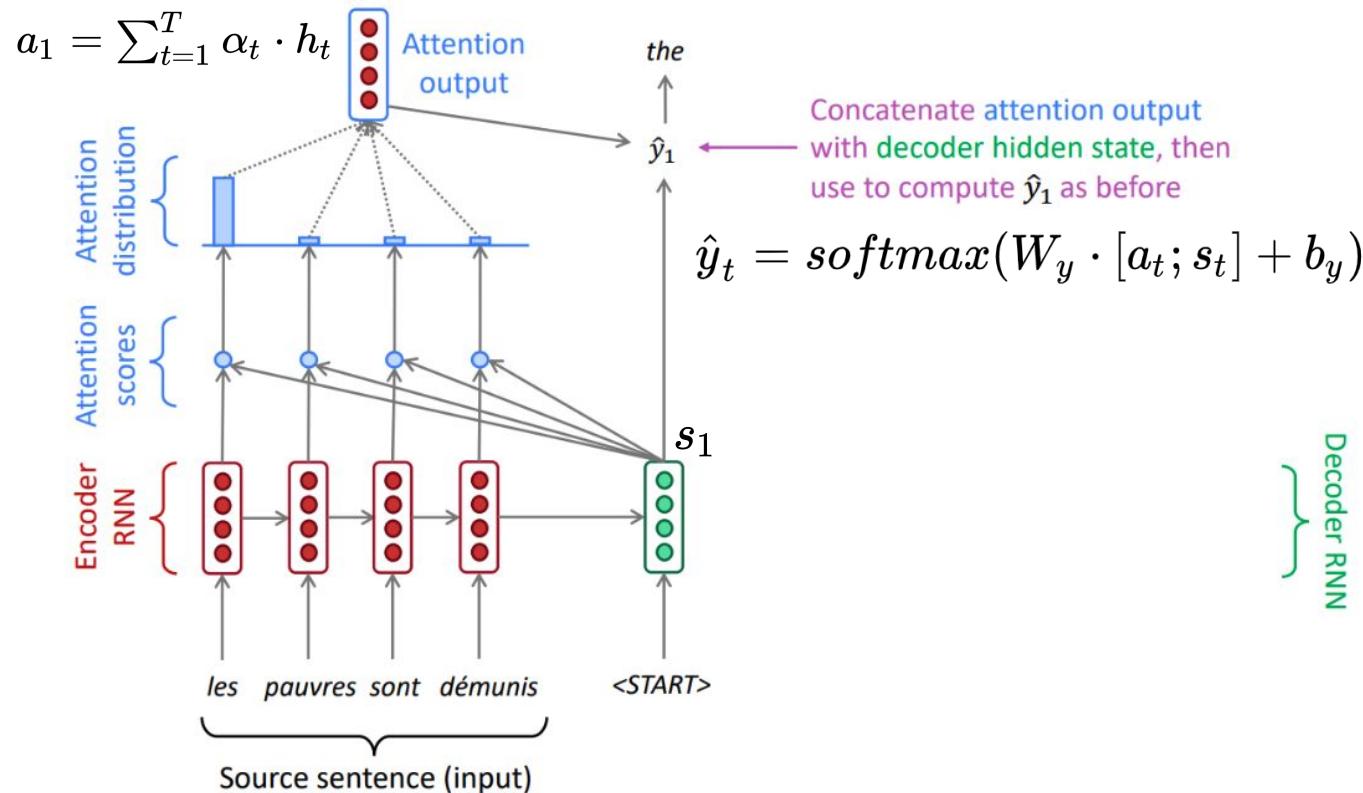


# Solution: Attentions



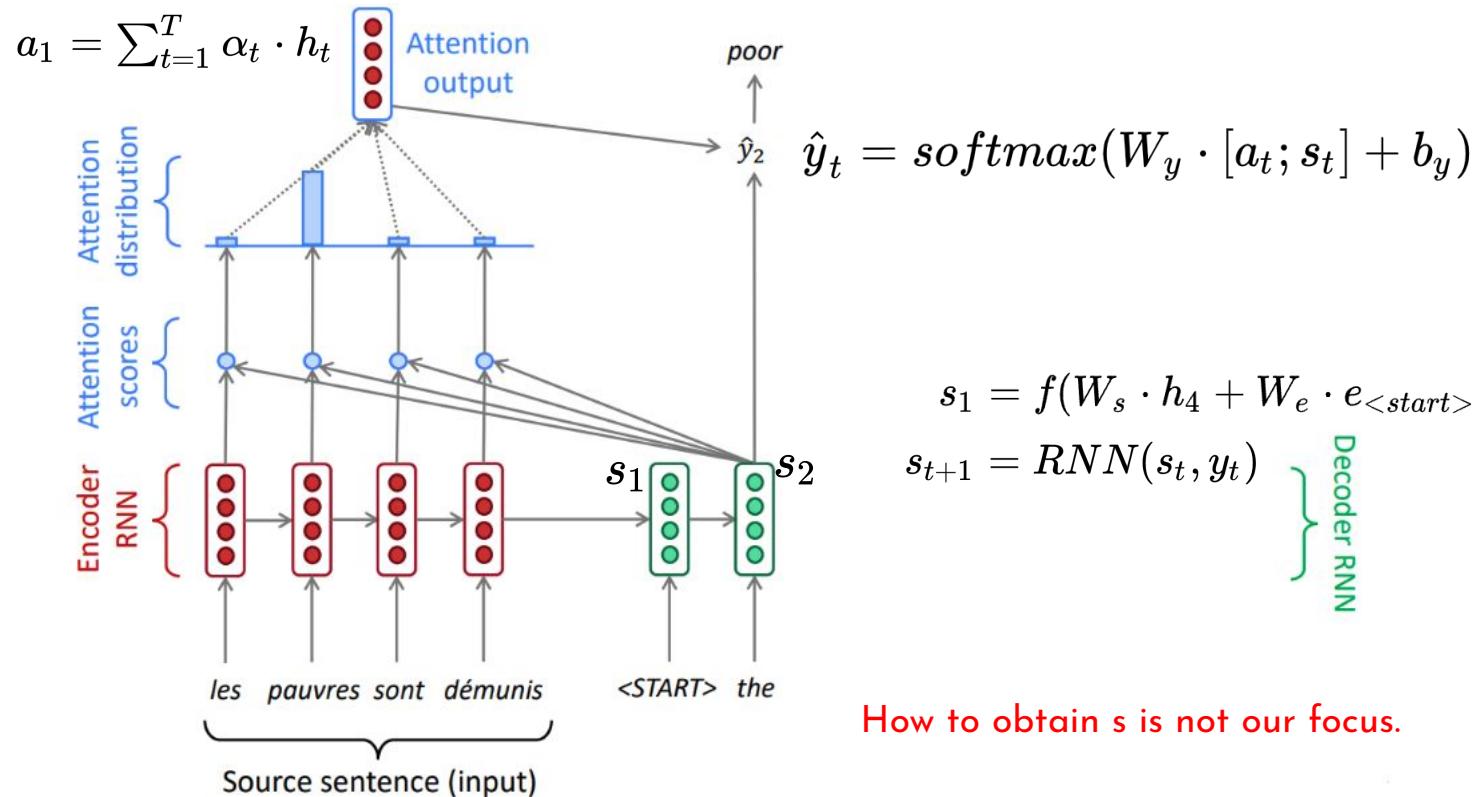


# Solution: Attentions





# Solution: Attentions





# Attentions (Formally)

- 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:

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

There are several ways to do this

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

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

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

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



# Attentions (Formally)

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



# Advantages of Attentions

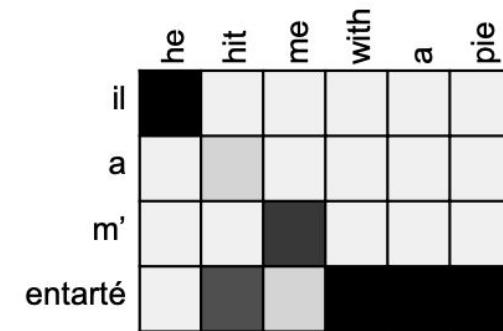
- Attention improves **NMT performance** significantly
  - It's necessary to allow decoder to focus on the relevant parts of the source
- Attention solves the **bottleneck problem**
  - allows decoder to look directly at source; bypass bottleneck
- Attention helps with **vanishing gradient problem**
  - Provides shortcut to faraway states

Refer to <https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3> for more illustrations



# Advantages of Attentions

- Attention provides interpretability
  - By inspecting attention distribution, we can see what the decoder was focusing on
  - (soft) alignment for free
- Attention is a general Deep Learning technique
  - Not just seq2seq model
  - Not just NMT





# Advantages of Attentions

- Attention is a general Deep Learning technique

- We have encoder hidden states  $h_1, \dots, h_N \in \mathbb{R}^h$
- On timestep  $t$ , we have decoder hidden state  $s_t \in \mathbb{R}^h$

Keys/Values

Query

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

Intuition:

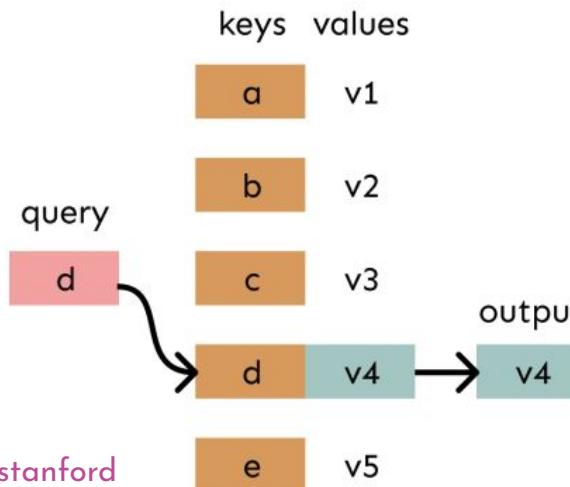
- The weighted sum is a **selective summary** of the information contained in the values, where the query determines which values to focus on.
- Attention is a way to obtain a **fixed-size representation** of an arbitrary set of **representations** (the values), dependent on some other representation (the query).



# Attentions As Query-Key-Value Computation

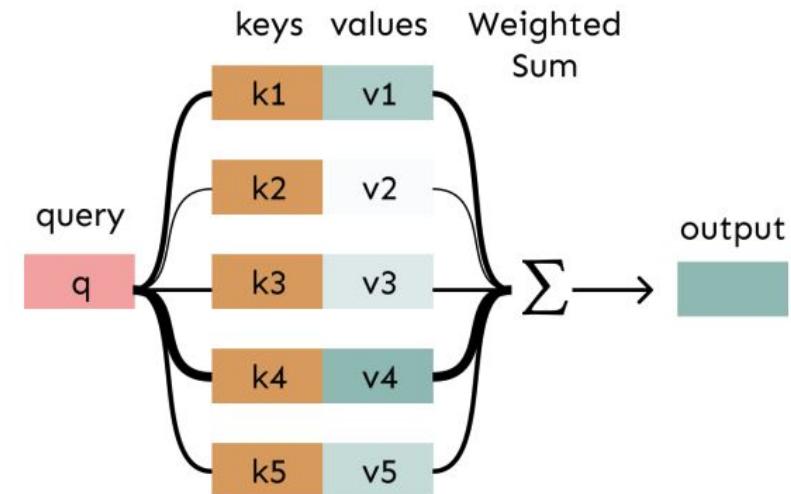
We can think of **attention** as performing fuzzy lookup in a key-value store.

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



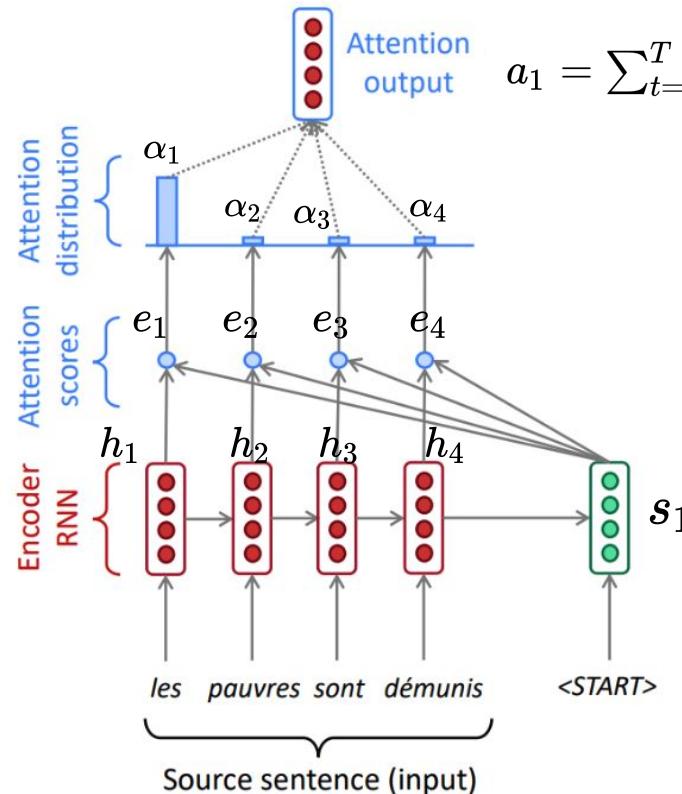
Source: stanford  
224n

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.





# Attentions As Query-Key-Value Computation



$$a_1 = \sum_{t=1}^T \alpha_t \cdot h_t$$

$$\alpha_t = \text{softmax}(e_t) = \frac{\exp(e_1)}{\sum_{i=1}^T \exp(e_i)}$$

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

h is the **key** vector and **value** vector  
s is the **query** vector



# Self Attentions

- Attentions can be generally applied between encoder and decoder for sequence-to-sequence learning.
- How about attentions within a single sequence?

Why not use attention for representations?



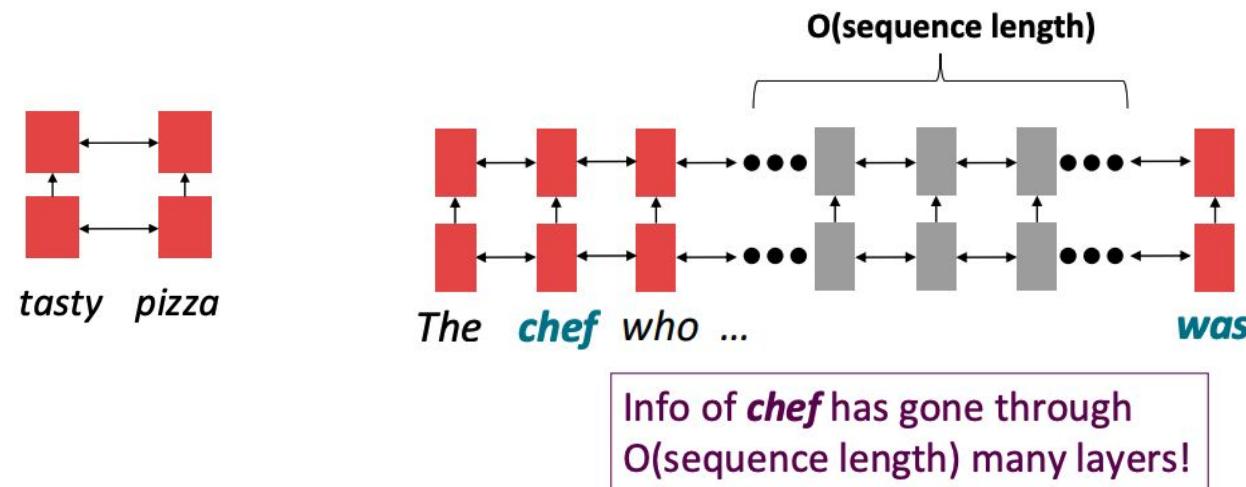
Self Attentions !!!



# Why Self Attentions?

Source: stanford 224n

- RNNs are unrolled “left-to-right” or “right-to-left”
- Only encodes linear locality: nearby words affect more
- Takes several steps for distant word pairs to interact

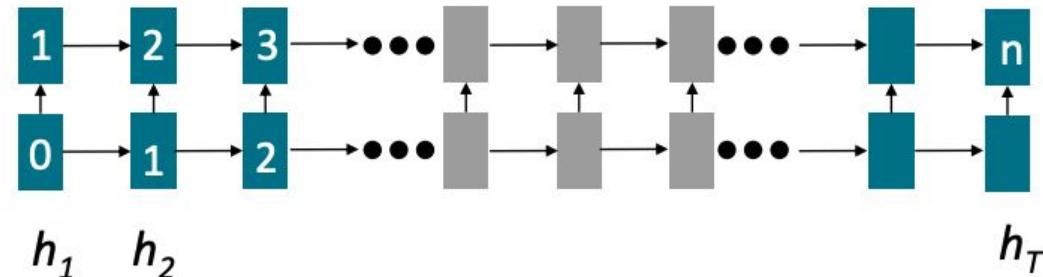




# Why Self Attentions?

Source: stanford 224n

- Forward and backward passes have  $O(\text{sequence length})$  unparallelizable operations
- 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

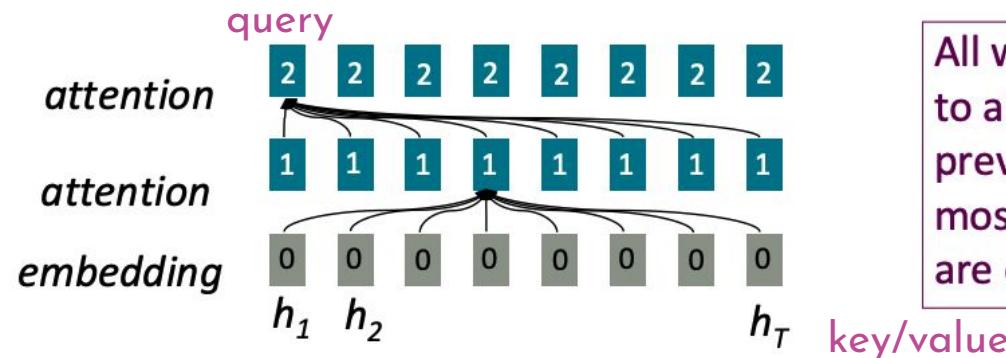


# Why Self Attentions?

Source: stanford 224n

- Treats each word's representation as a query to access and incorporate information from a set of values.
- Easy to parallelize (per layer).
- Maximum interaction distance:  $O(1)$ , since all words interact at every layer!

Each word can  
be query, key,  
value



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



# Self Attention Mechanism

Source: stanford 224n

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

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 weight matrices Q, K, V, each in  $\mathbb{R}^{d \times d}$

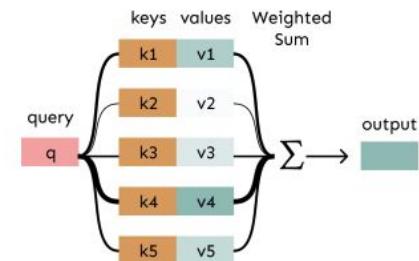
$$\mathbf{q}_i = Qx_i \text{ (queries)} \quad \mathbf{k}_i = Kx_i \text{ (keys)} \quad \mathbf{v}_i = Vx_i \text{ (values)}$$

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

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

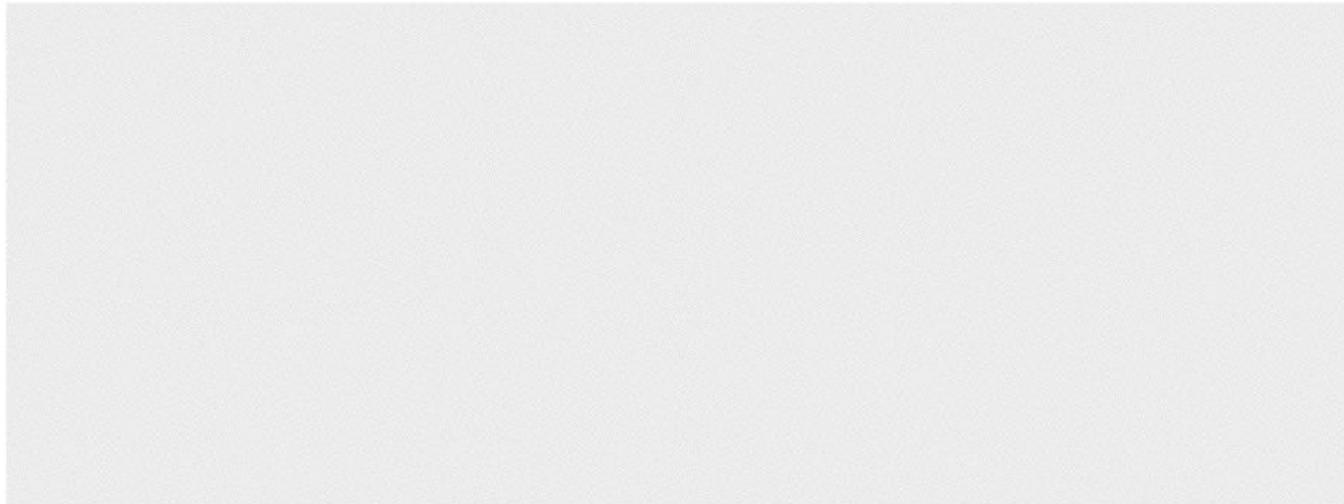
3. Compute output for each word as weighted sum of values

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





# Self Attention Mechanism



input #1

1	0	1	0
---	---	---	---

input #2

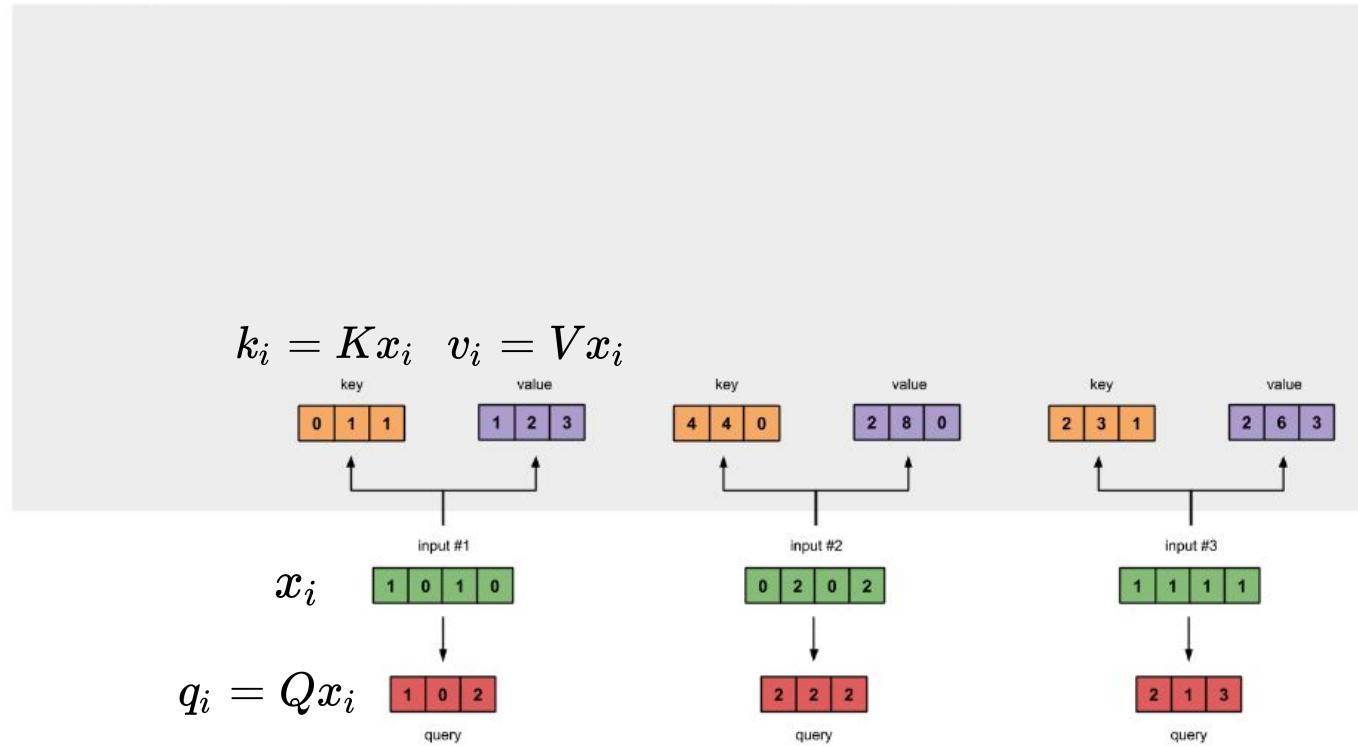
0	2	0	2
---	---	---	---

input #3

1	1	1	1
---	---	---	---



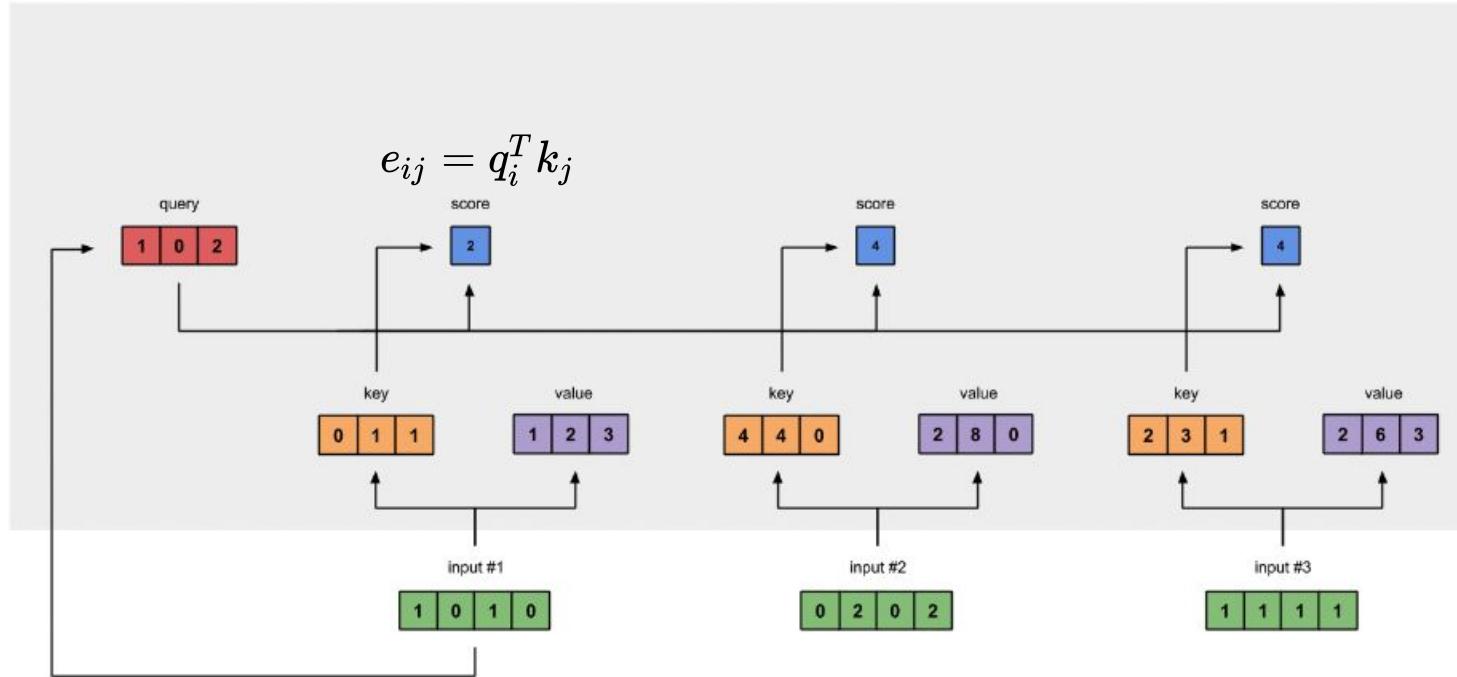
# Self Attention Mechanism



Source: <https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a>



# Self Attention Mechanism

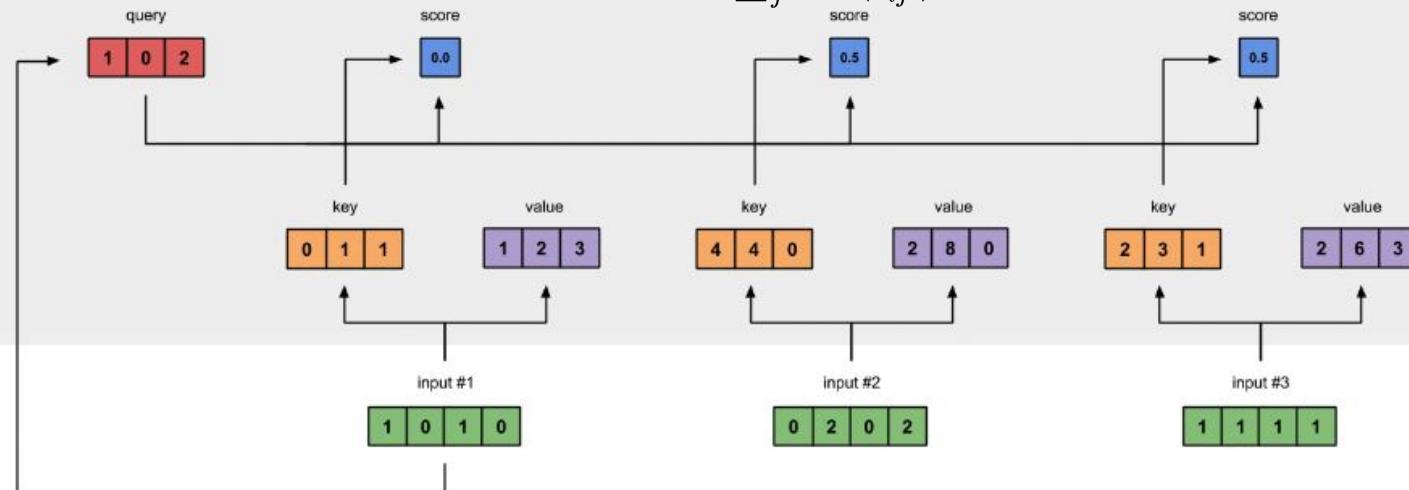


Source: <https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a>



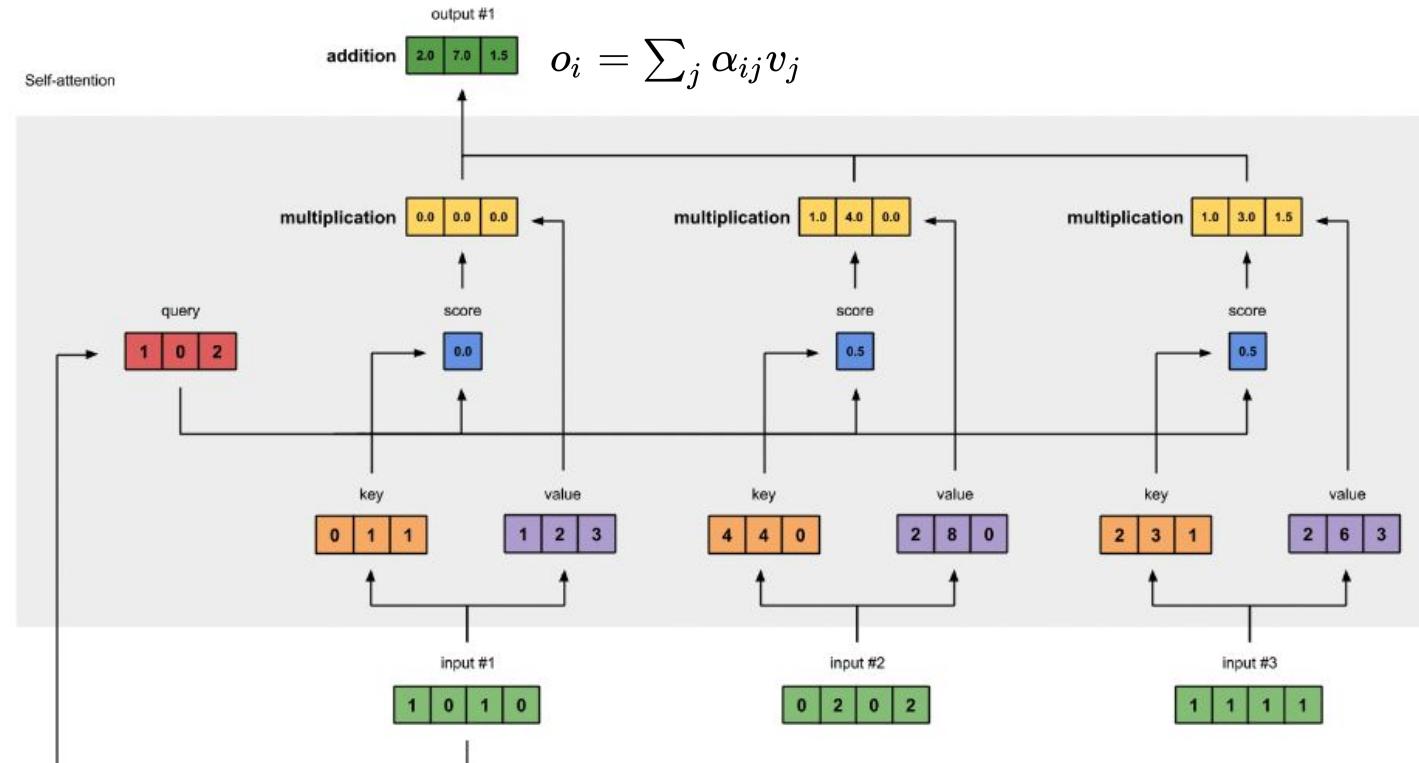
# Self Attention Mechanism

$$e_{ij} = q_i^T k_j \quad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$





# Self Attention Mechanism



Source: <https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a>



# Outline for today

**01** From RNNs to Attentions

**02** Transformer Model

**03** Applications of Transformers



# From Self Attention to Transformer

- Transformers are basic modules commonly adopted in various language models.
- It's a lot like our minimal self attention architecture, but with a few more components.
  - Adding position embeddings
  - Adding non-linearities
  - Single to multi-head self attention
  - Multiple layers



# Position Embedding

## Barriers

- Doesn't have an inherent notion of order!



## Solutions



# 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**

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

- Don't worry about what the  $\mathbf{p}_i$  are made of yet!
- Easy to incorporate this info into our self-attention block: 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:

$$\mathbf{x}_i = \mathbf{x}_i + \mathbf{p}_i$$

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



# Position Embedding



$$x_i = x_i + p_i$$

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



# Position Embedding

- **Learned absolute position representations:** Let all  $p_i$  be learnable parameters!  
Learn a matrix  $\mathbf{p} \in \mathbb{R}^{d \times n}$ , *and let each  $p_i$  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  $1, \dots, n$ .
- Most systems use this!



# Non-linearities

## Barriers

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning! It's all just weighted averages



## Solutions

- Add position representations to the inputs



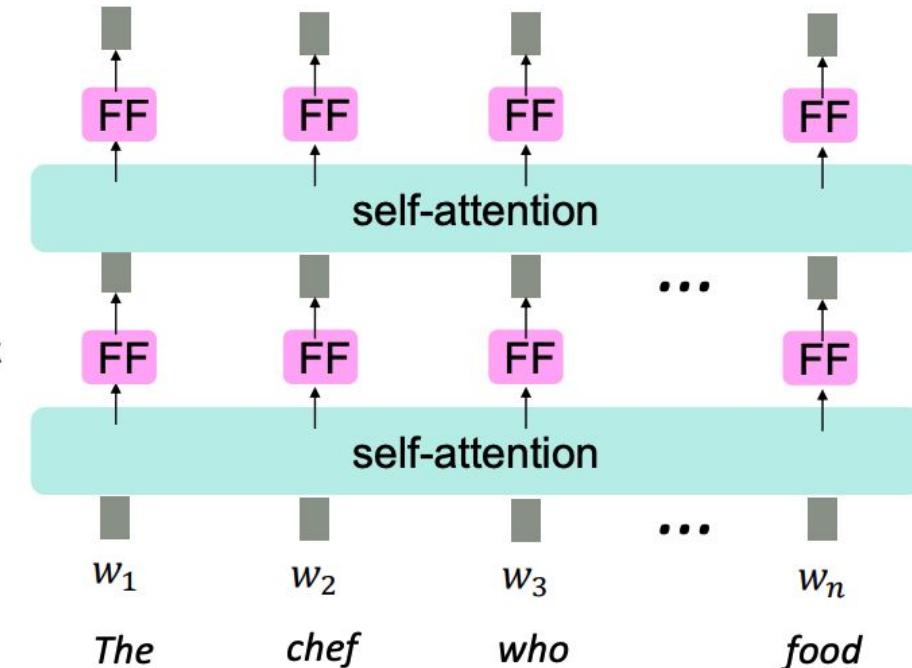
# Nonlinearities

Source: stanford 224n

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages **value** vectors (Why? Look at the notes!)
- Easy fix: add a **feed-forward network** to post-process each output vector.

$$m_i = \text{MLP}(\text{output}_i)$$

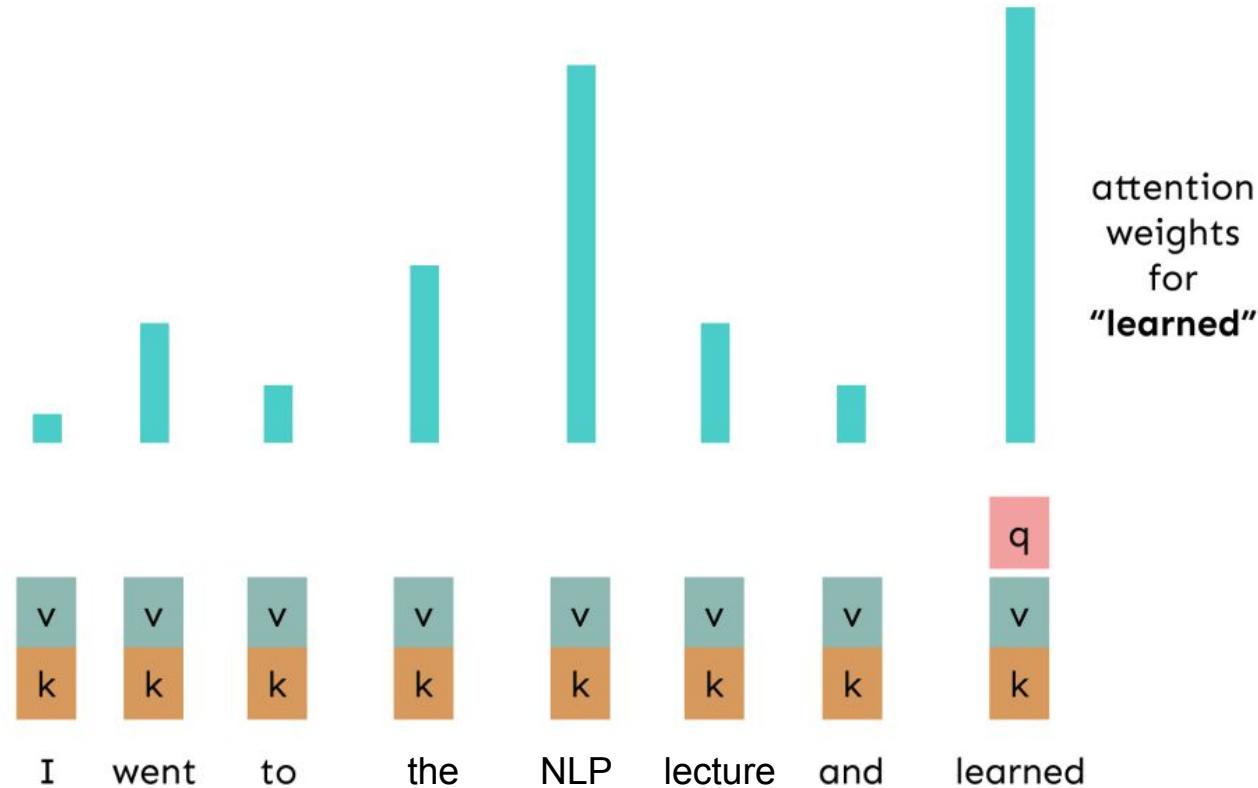
$$= W_2 * \text{ReLU}(W_1 \text{output}_i + b_1) + b_2$$





# Recall a Single Self Attention

Source: stanford 224n

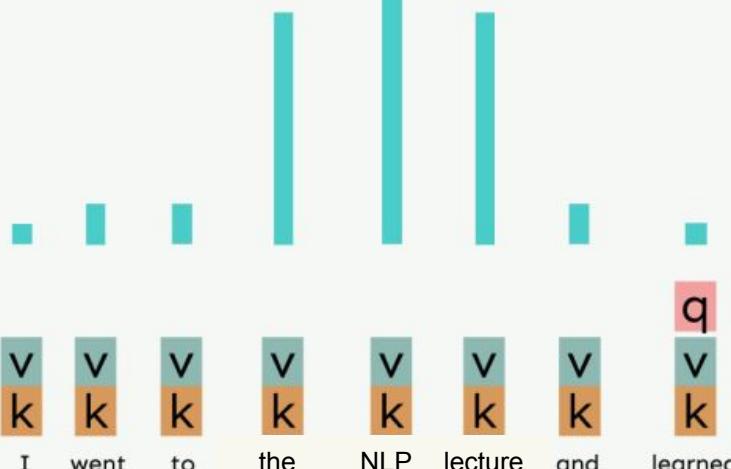




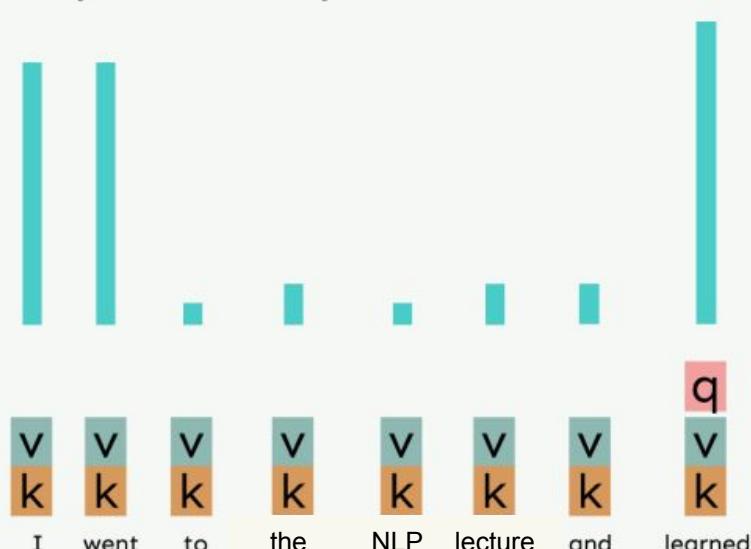
# Multi-head Self Attention

Source: stanford 224n

Attention head 1  
attends to entities



Attention head 2 attends to  
syntactically relevant words





# Matrix-Form Computation (Optional)

- Let's look at how query-key-value attention is computed in matrix form
  - Let  $X = [x_1; \dots; x_n] \in \mathbb{R}^{n \times d}$  be the concatenation of input vectors
  - Compute query, key, value matrices:  $XQ, XK, XV \in \mathbb{R}^{n \times d}$
- Compute the output using self attentions
  - Generate attention score matrix
$$A = \text{softmax}(XQ \cdot (XK)^T) \in \mathbb{R}^{n \times n}$$
  - Aggregate with value matrix
$$\text{output} = A \cdot (XV) \in \mathbb{R}^{n \times d}$$

	Hello	I	love	you
Hello	0.8	0.1	0.05	0.05
I	0.1	0.6	0.2	0.1
love	0.05	0.2	0.65	0.1
you	0.2	0.1	0.1	0.6



# Matrix-Form Computation (Optional)

Source: stanford 224n

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

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

All pairs of attention scores!

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

$$\text{softmax} \left( XQK^\top X^\top \right) XV = \text{output} \in \mathbb{R}^{n \times d}$$



# Matrix-Form Computation (Optional)

Source: stanford 224n

- What if we want to look in multiple places in the sentence at once?
  - For word  $i$ , self-attention “looks” where  $x_i^T Q^T K x_j$  is high, but maybe we want to focus on different  $j$  for different reasons?
- We’ll define **multiple attention “heads”** through multiple Q,K,V matrices
- Let,  $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$ , where  $h$  is the number of attention heads, and  $\ell$  ranges from 1 to  $h$ .
- Each attention head performs attention independently:
  - $\text{output}_\ell = \text{softmax}(X Q_\ell K_\ell^T X^T) * X V_\ell$ , where  $\text{output}_\ell \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
  - $\text{output} = [\text{output}_1; \dots; \text{output}_h] Y$ , where  $Y \in \mathbb{R}^{d \times d}$
- Each head gets to “look” at different things, and construct value vectors differently.



# Matrix-Form Computation (Optional)

Source: stanford 224n

$$Q \in \mathbb{R}^{d \times d/h}$$

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

$$\begin{matrix} XQ \\ K \end{matrix} = \begin{matrix} XQK^\top X^\top \\ K^\top X^\top \end{matrix} \in \mathbb{R}^{3 \times n \times n}$$

3 sets of all pairs of attention scores!

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

$$\text{softmax} \left( \begin{matrix} XQK^\top X^\top \\ XV \end{matrix} \right) V = \begin{matrix} Y_{\text{mix}} \\ Y \end{matrix} = \text{output} \in \mathbb{R}^{n \times d}$$
$$V \in \mathbb{R}^{d \times d/h}$$
$$Y \in \mathbb{R}^{d \times d}$$



# Scaled Dot Product (Optional)

- “Scaled Dot Product” attention aids in training.
- 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.
- Instead of the self-attention function we’ve seen:

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

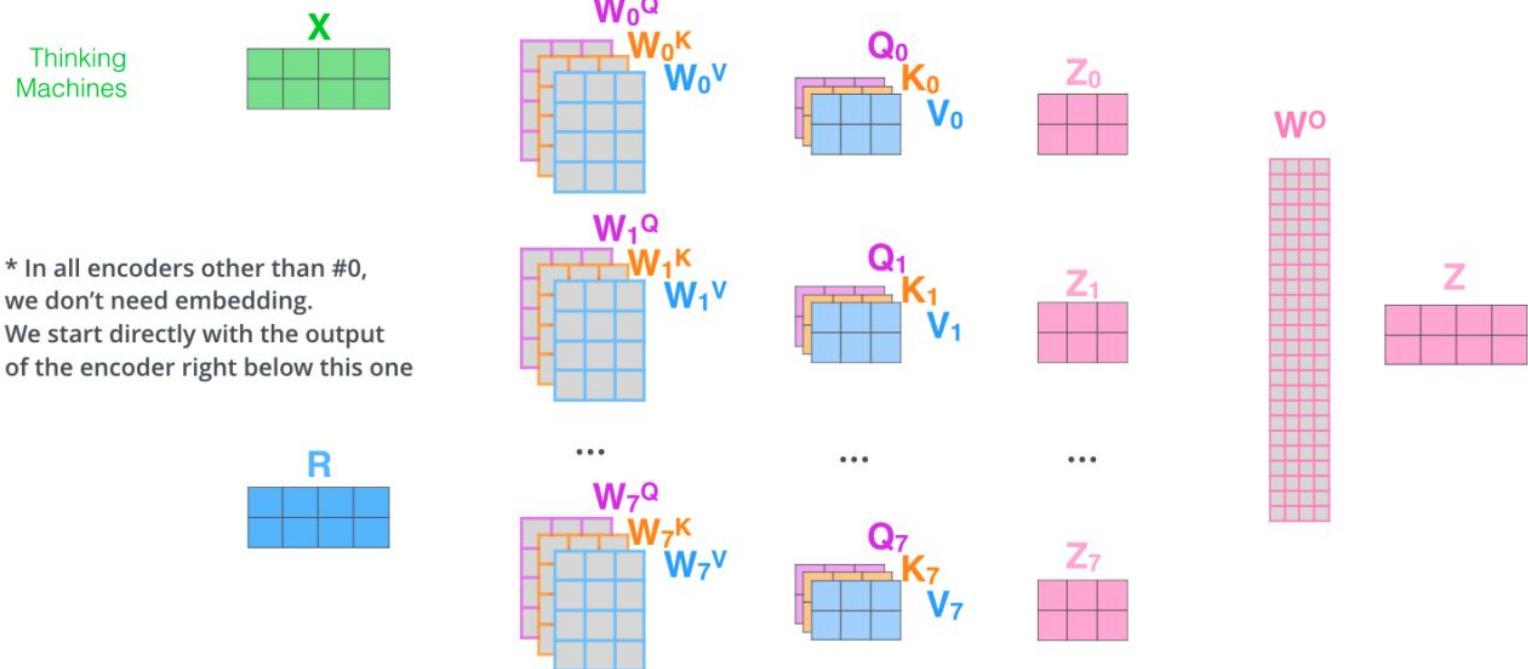
- We divide the attention scores by  $\sqrt{d/h}$ , to stop the scores from becoming large just as a function of  $d/h$  (The dimensionality divided by the number of heads.)

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



# Process

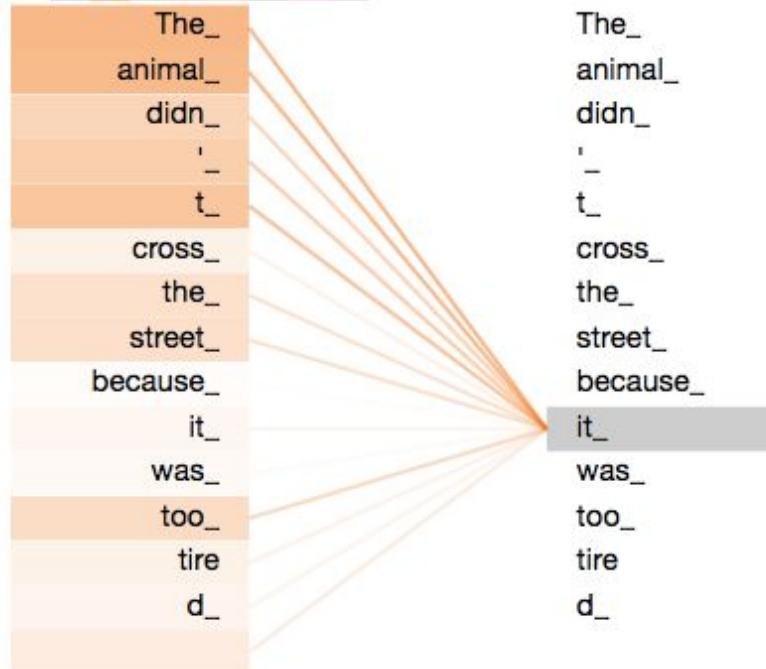
- 1) This is our input sentence\* each word\*
- 2) We embed
- 3) Split into 8 heads.  
We multiply  $X$  or  $R$  with weight matrices
- 4) Calculate attention using the resulting  $Q/K/V$  matrices
- 5) Concatenate the resulting  $Z$  matrices, then multiply with weight matrix  $W^o$  to produce the output of the layer



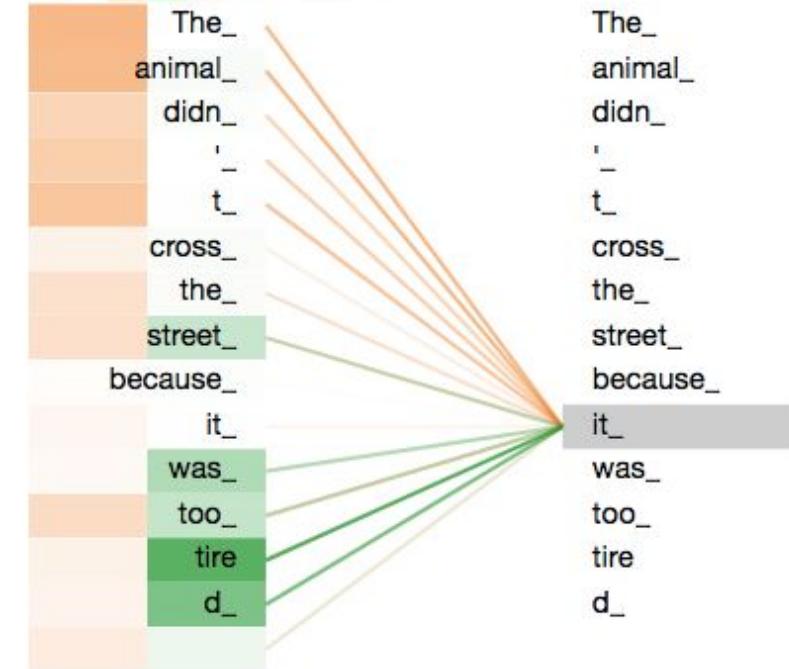


# An Example

Layer: 5 ⬆ Attention: Input - Input ⬆



Layer: 5 ⬆ Attention: Input - Input ⬆



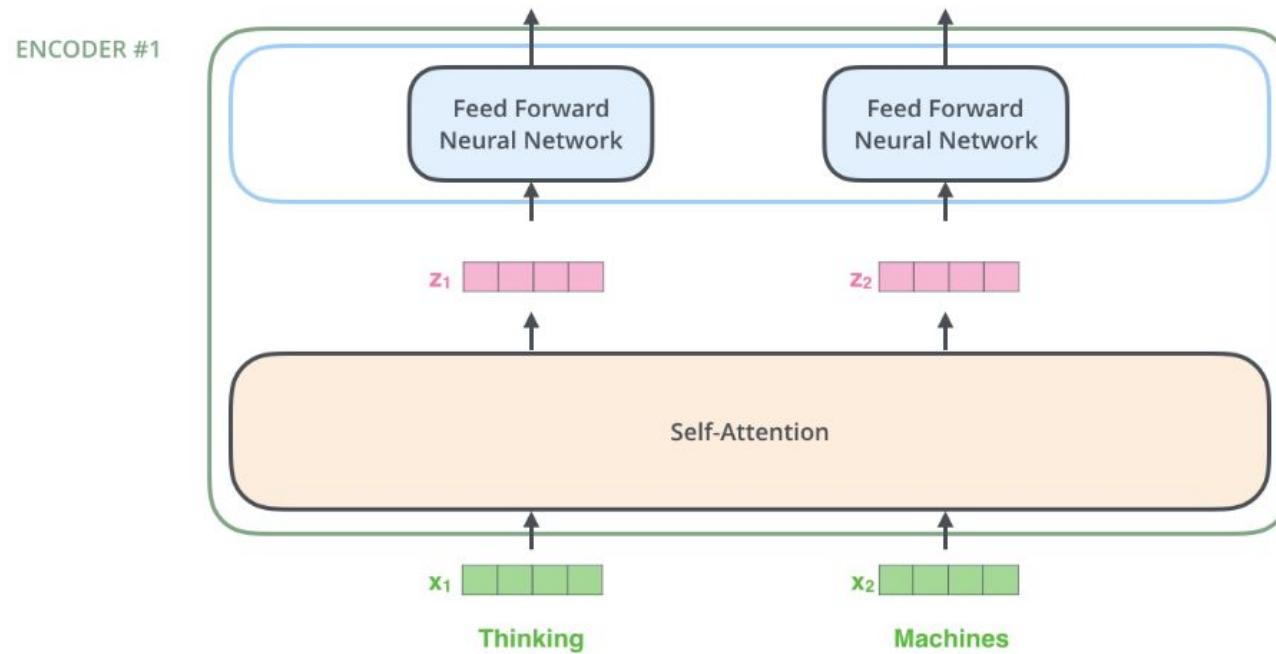


# From Self Attention to Transformer

- Now that we've replaced self attention with multi-head self attention, we'll go through two optimization tricks
  - Residual connections
  - Layer normalizations
- In most Transformer diagrams, these are often written together as "Add & Norm"



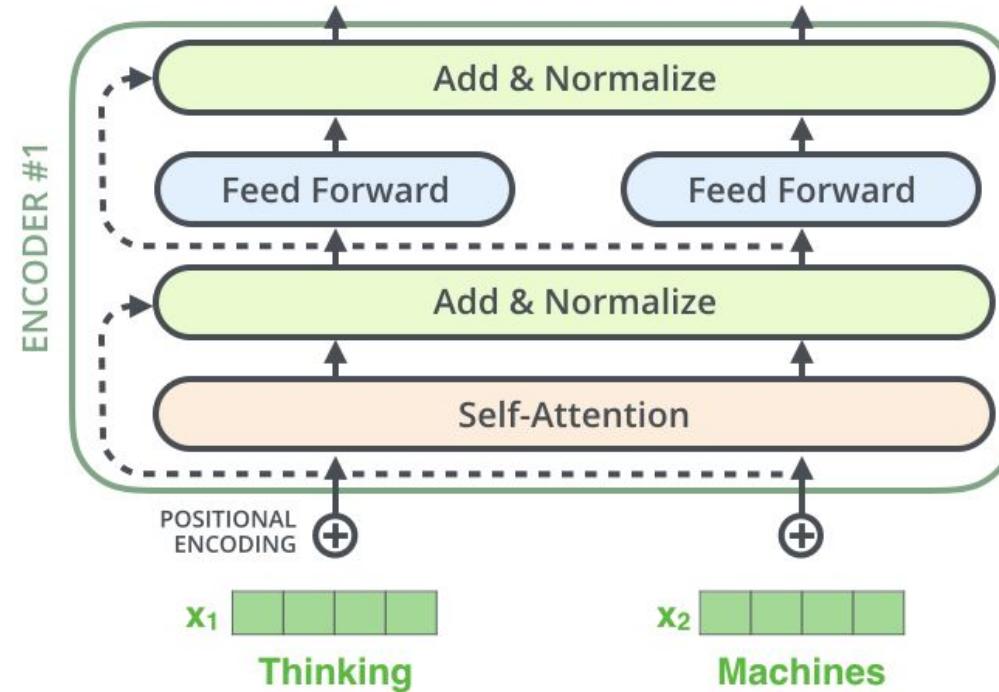
# Self Attention



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



# Self Attention with Add & Norm





# Residual Connections (Recap)

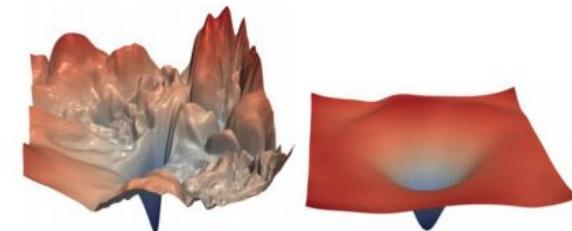
- **Residual connections** are a trick to help models train better.
  - Instead of  $X^{(i)} = \text{Layer}(X^{(i-1)})$  (where  $i$  represents the layer)



- 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!
- Bias towards the identity function!



[no residuals] [residuals]

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



# Layer Normalization [Ba et al. 2016] (Optional)

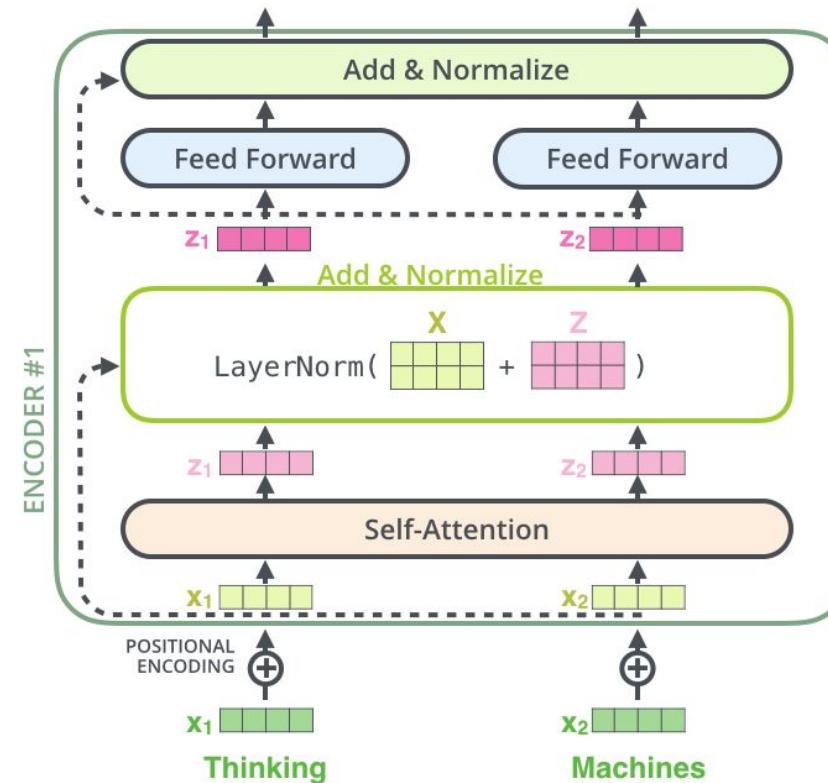
- Layer normalization is a trick to help models train faster.
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**.
  - LayerNorm's success may be due to its normalizing gradients [[Xu et al., 2019](#)]
- Let  $x \in \mathbb{R}^d$  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



# Self Attention with Add & Norm



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

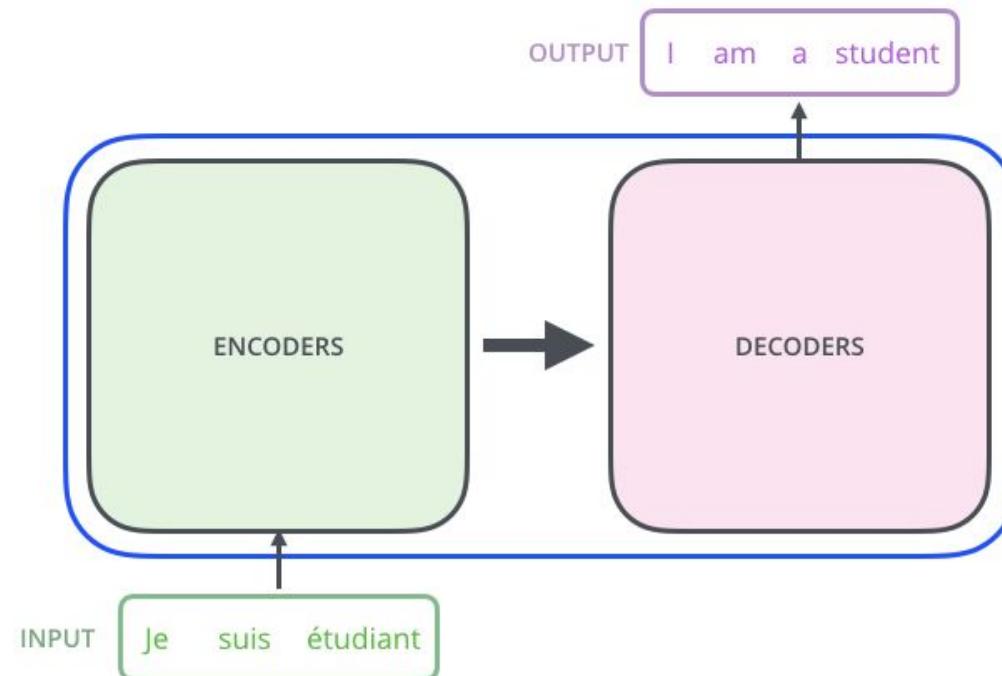


# Transformer Encoder-Decoder



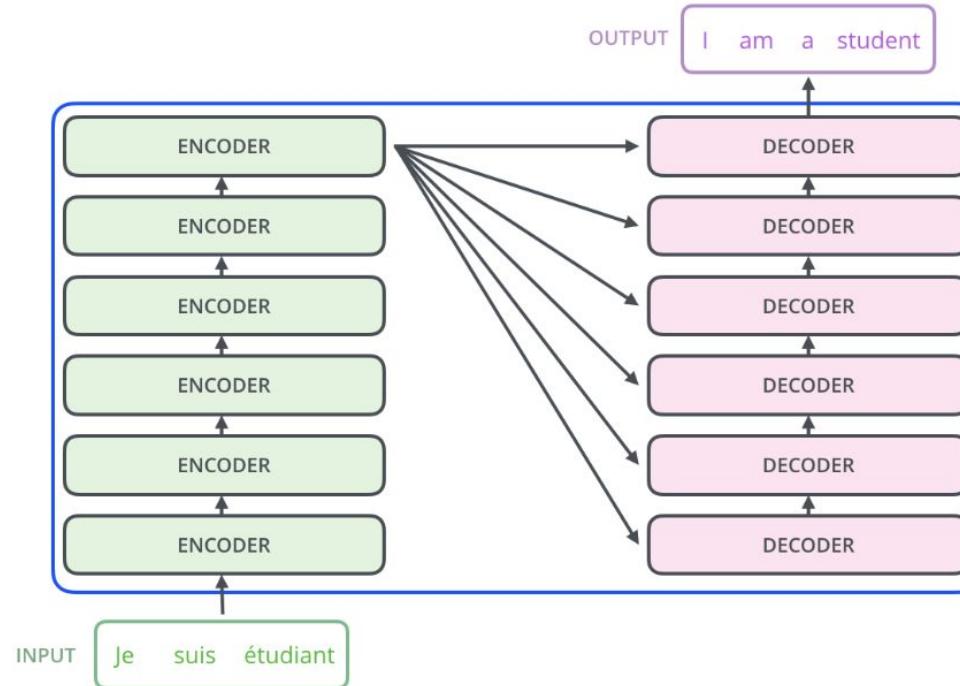


# Transformer Encoder-Decoder





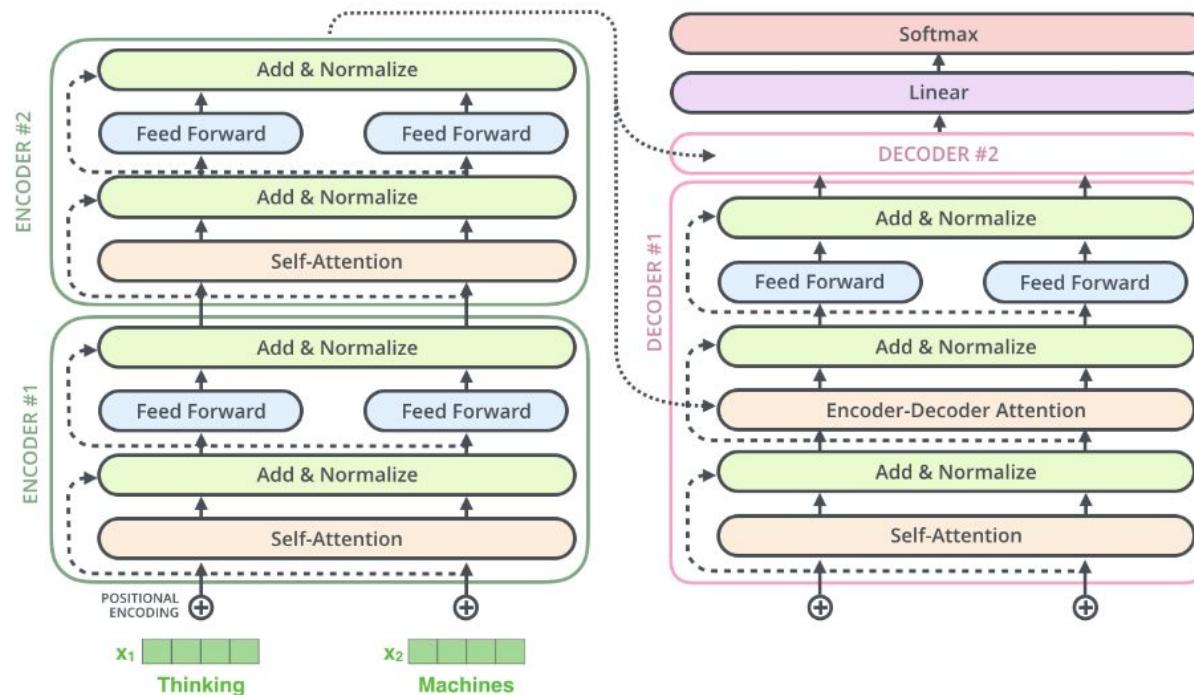
# Transformer Encoder-Decoder



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



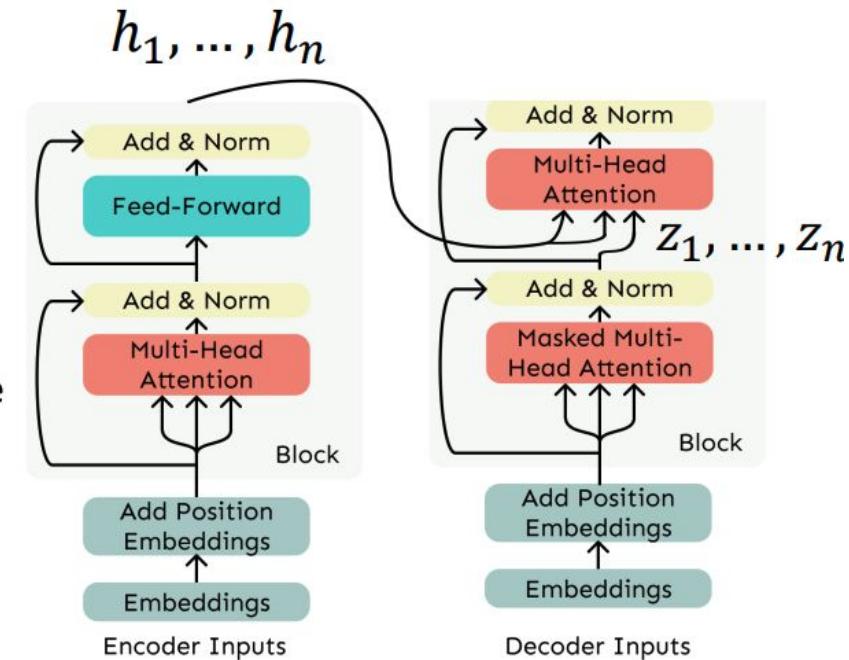
# Transformer Encoder-Decoder (Optional)



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

# Encoder-Decoder Attention (Optional)

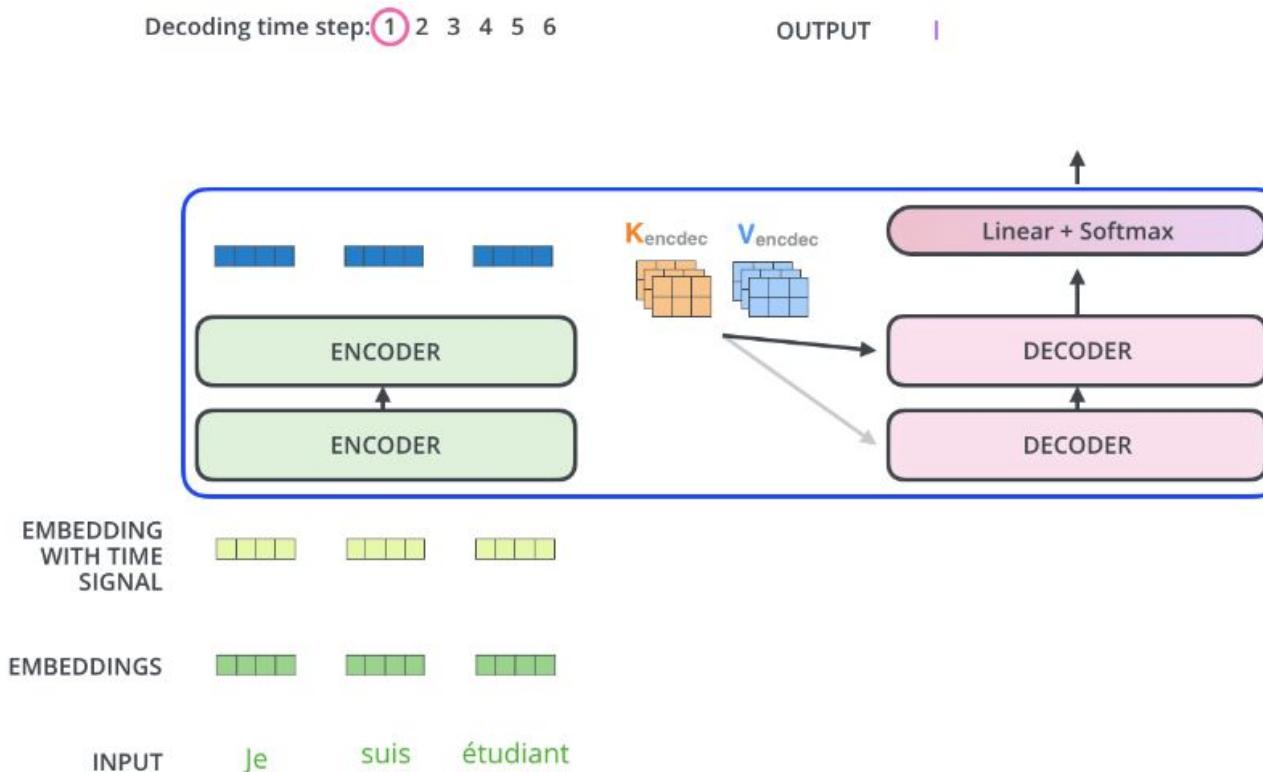
- We saw that self-attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let  $h_1, \dots, h_n$  be **output vectors from the Transformer encoder**;  $x_i \in \mathbb{R}^d$
- Let  $z_1, \dots, z_n$  be 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**,  $q_i = Qz_i$ .



Source: stanford 224n

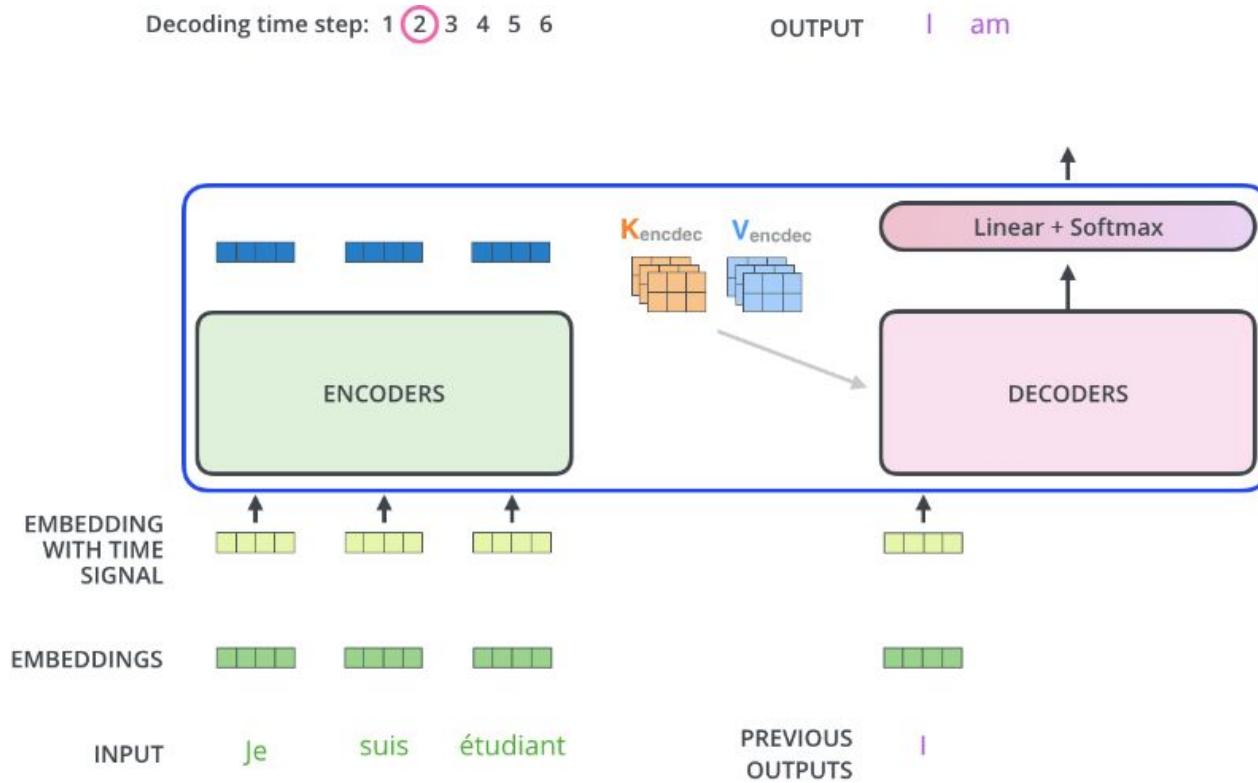


# Transformer Encoder-Decoder



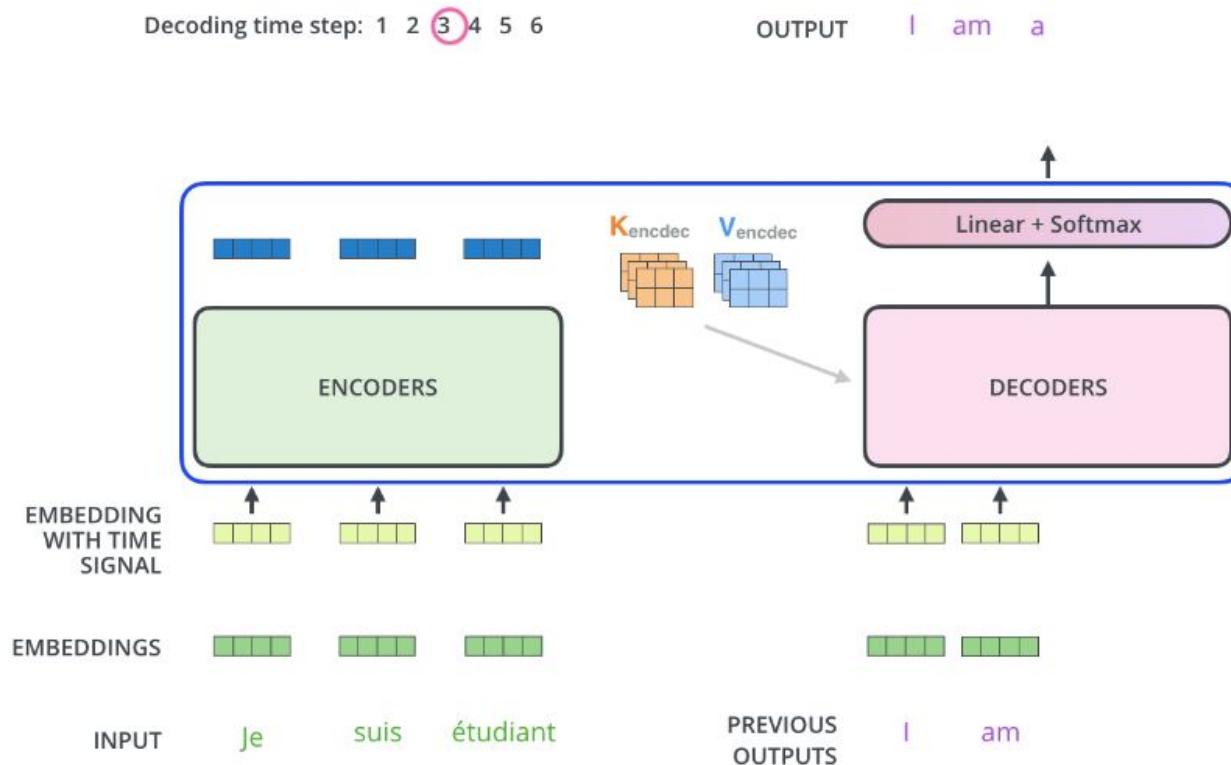


# Transformer Encoder-Decoder





# Transformer Encoder-Decoder



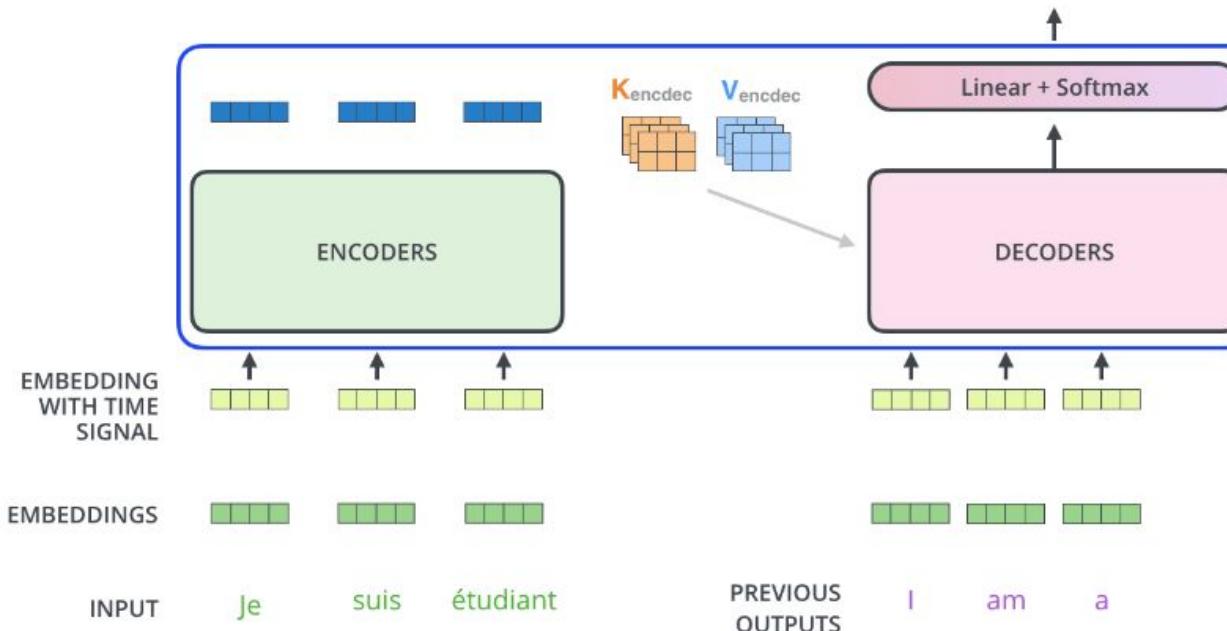


# Transformer Encoder-Decoder

Decoding time step: 1 2 3 4 5 6

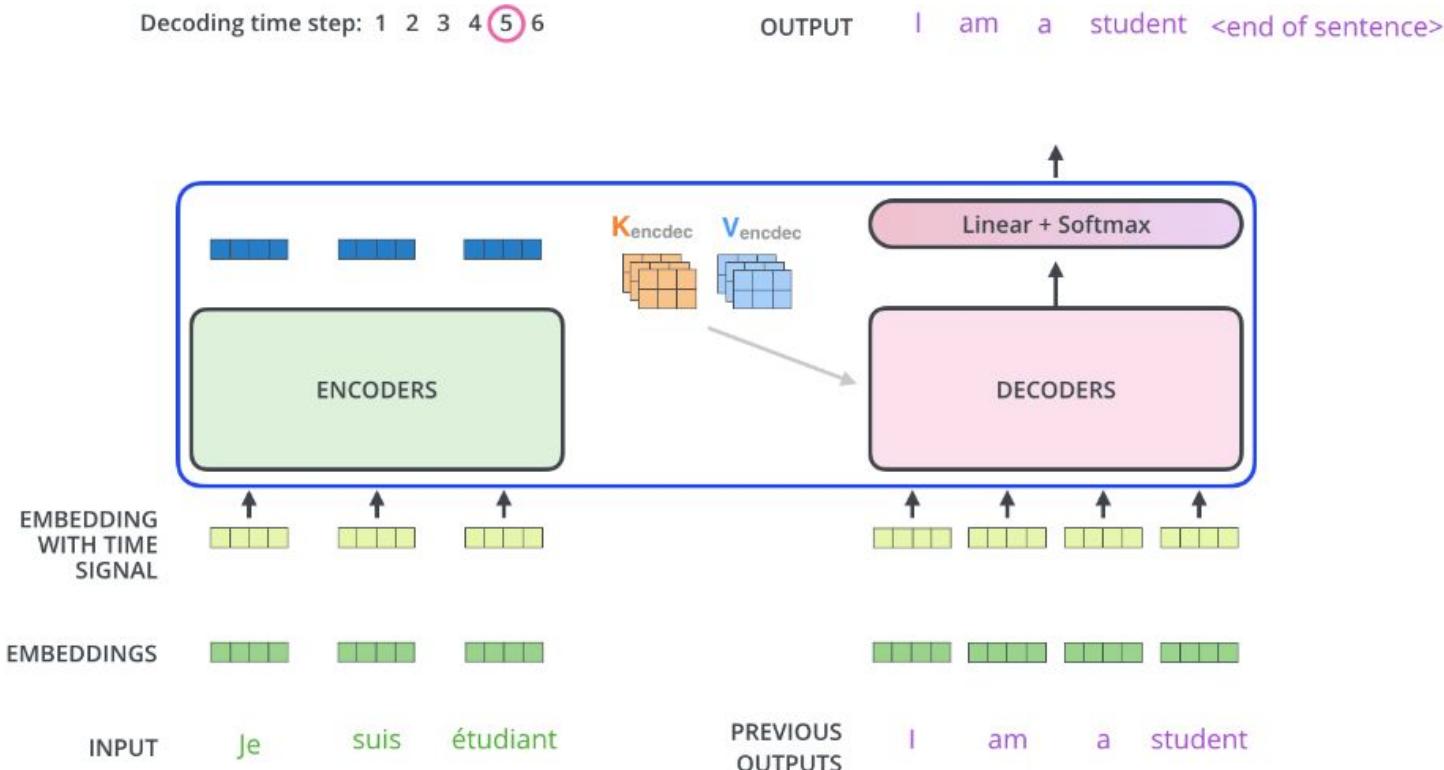
OUTPUT

I am a student





# Transformer Encoder-Decoder

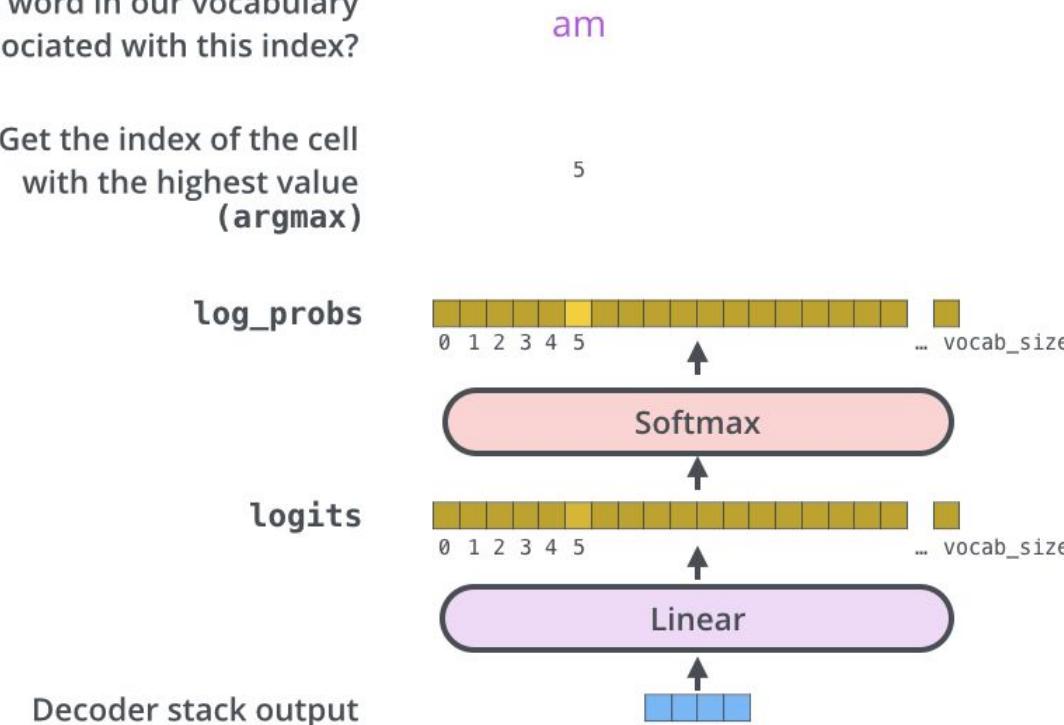




# Transformer Encoder-Decoder

Which word in our vocabulary  
is associated with this index?

Get the index of the cell  
with the highest value  
(`argmax`)





# Outline for today

**01** From RNNs to Attentions

**02** Transformer Model

**03** Applications of Transformers



# Great Results with Transformers

Source: stanford 224n

First, Machine Translation from the original Transformers paper!

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	<b>41.29</b>	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1		<b><math>3.3 \cdot 10^{18}</math></b>
Transformer (big)	<b>28.4</b>	<b>41.8</b>		$2.3 \cdot 10^{19}$



# Great Results with Transformers

Source: stanford 224n

Next, document generation!

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

The old standard

Transformers all the way down.



# Great Results with Transformers

Source: stanford 224n

Transformers' parallelizability allows for efficient pretraining, and have made them the de-facto standard.

On this popular aggregate benchmark, for example:



All top models are Transformer (and pretraining)-based.

Rank	Name	Model	URL Score
1	DeBERTa Team - Microsoft	DeBERTa / TuringNLv4	90.8
2	HFL iFLYTEK	MacALBERT + DKM	90.7
3	+ Alibaba DAMO NLP	StructBERT + TAPT	90.6
4	+ PING-AN Omni-Sinitic	ALBERT + DAAF + NAS	90.6
5	ERNIE Team - Baidu	ERNIE	90.4
6	T5 Team - Google	T5	90.3

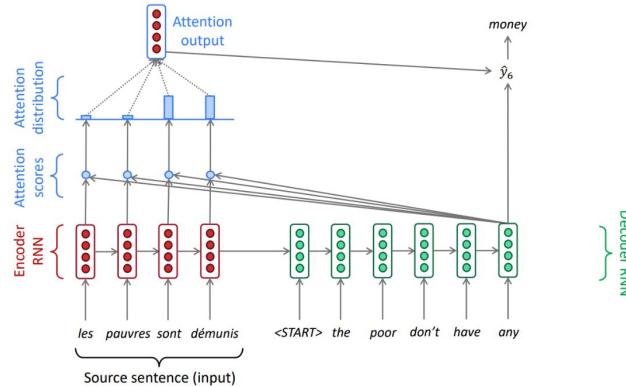


# Learn Transformers (for your interest)

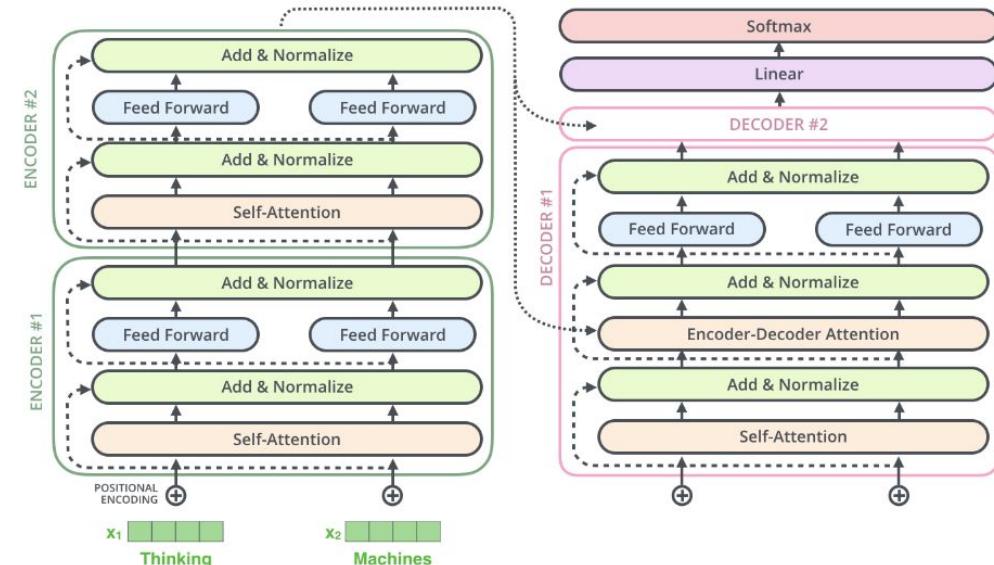
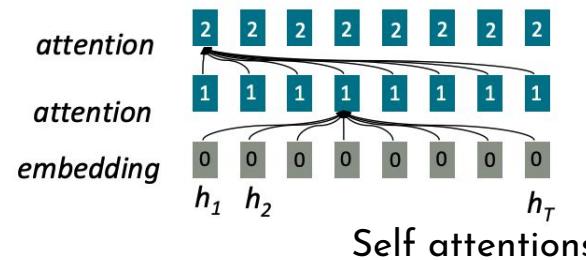
- Key recommended resource:
  - The annotated transformer  
<http://nlp.seas.harvard.edu/2018/04/03/attention.html>
  - The illustrated transformer  
<https://jalammar.github.io/illustrated-transformer/>



# Summary



Attention-based encoder-decoder



Transformers