

# Attention, Transformer, GPT and BERT

Yuan YAO

HKUST

# Summary

- ▶ We have shown:
  - ▶ CNN Architectures: LeNet5, Alexnet, VGG, GoogleNet, Resnet
  - ▶ Recurrent Neural Networks and LSTM
- ▶ Today:
  - ▶ **Attention**
  - ▶ **Transformer**
  - ▶ **BERT/GPT**
- ▶ Reference:
  - ▶ Feifei Li, Stanford cs231n
  - ▶ Chris Manning, Stanford cs224n

# A Brief History in NLP

- ▶ In 2013-2015, LSTMs started achieving state-of-the-art results
  - ▶ Successful tasks include: handwriting recognition, speech recognition, machine translation, parsing, image captioning
  - ▶ LSTM became the dominant approach
- ▶ Now (2019), other approaches (e.g. Transformers) have become more dominant for Machine Translation.
  - ▶ For example in **WMT** (a MT conference + competition):
  - ▶ In WMT 2016, the summary report contains "RNN" 44 times
  - ▶ In WMT 2018, the report contains "RNN" 9 times and "Transformer" 63 times
- ▶ **Source:** "Findings of the 2016 Conference on Machine Translation (WMT16)", Bojar et al. 2016, <http://www.statmt.org/wmt16/pdf/W16-2301.pdf>
- ▶ **Source:** "Findings of the 2018 Conference on Machine Translation (WMT18)", Bojar et al. 2018, <http://www.statmt.org/wmt18/pdf/WMT028.pdf>



# Neural Machine Translation

Machine Translation using Neural Networks

# Neural Machine Translation (NMT)

The sequence-to-sequence model

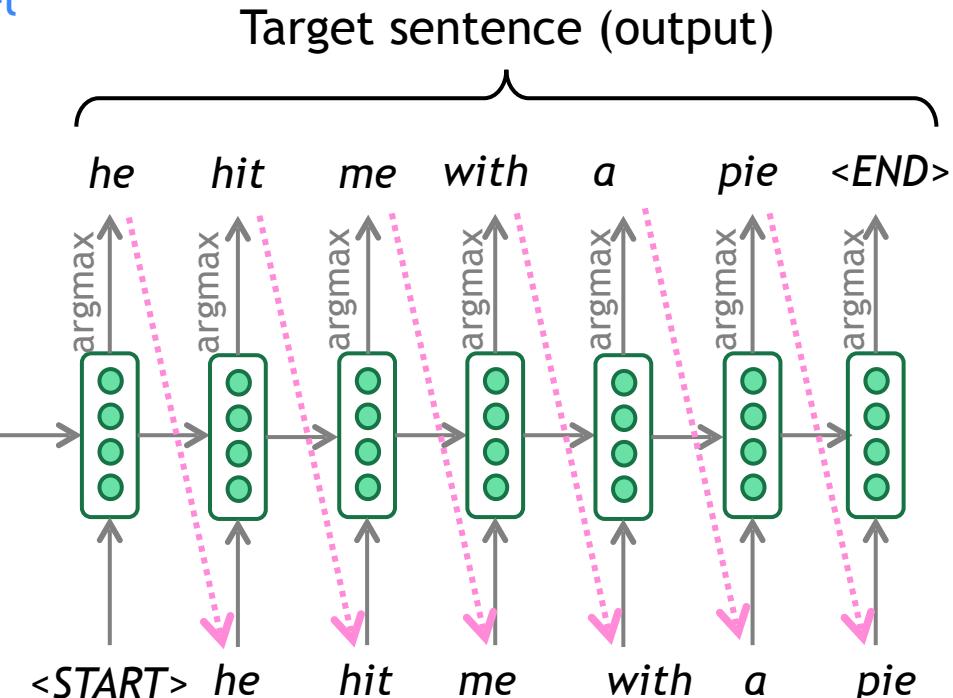
Encoding of the source sentence.  
Provides initial hidden state  
for Decoder RNN.

Encoder RNN

*il a m' entarté*

Source sentence (input)

Encoder RNN produces  
an **encoding** of the  
source sentence.

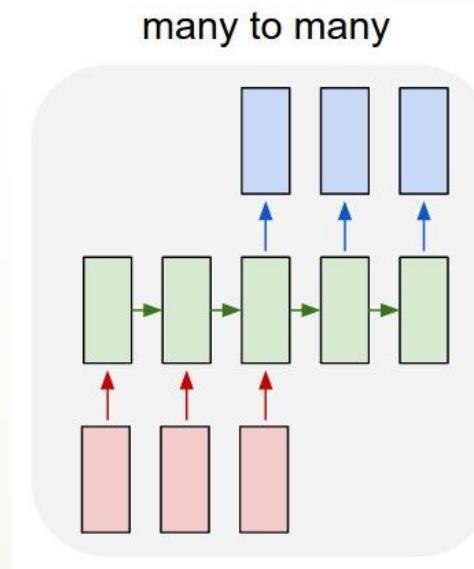


Decoder RNN is a Language Model that generates target sentence, *conditioned on encoding*.

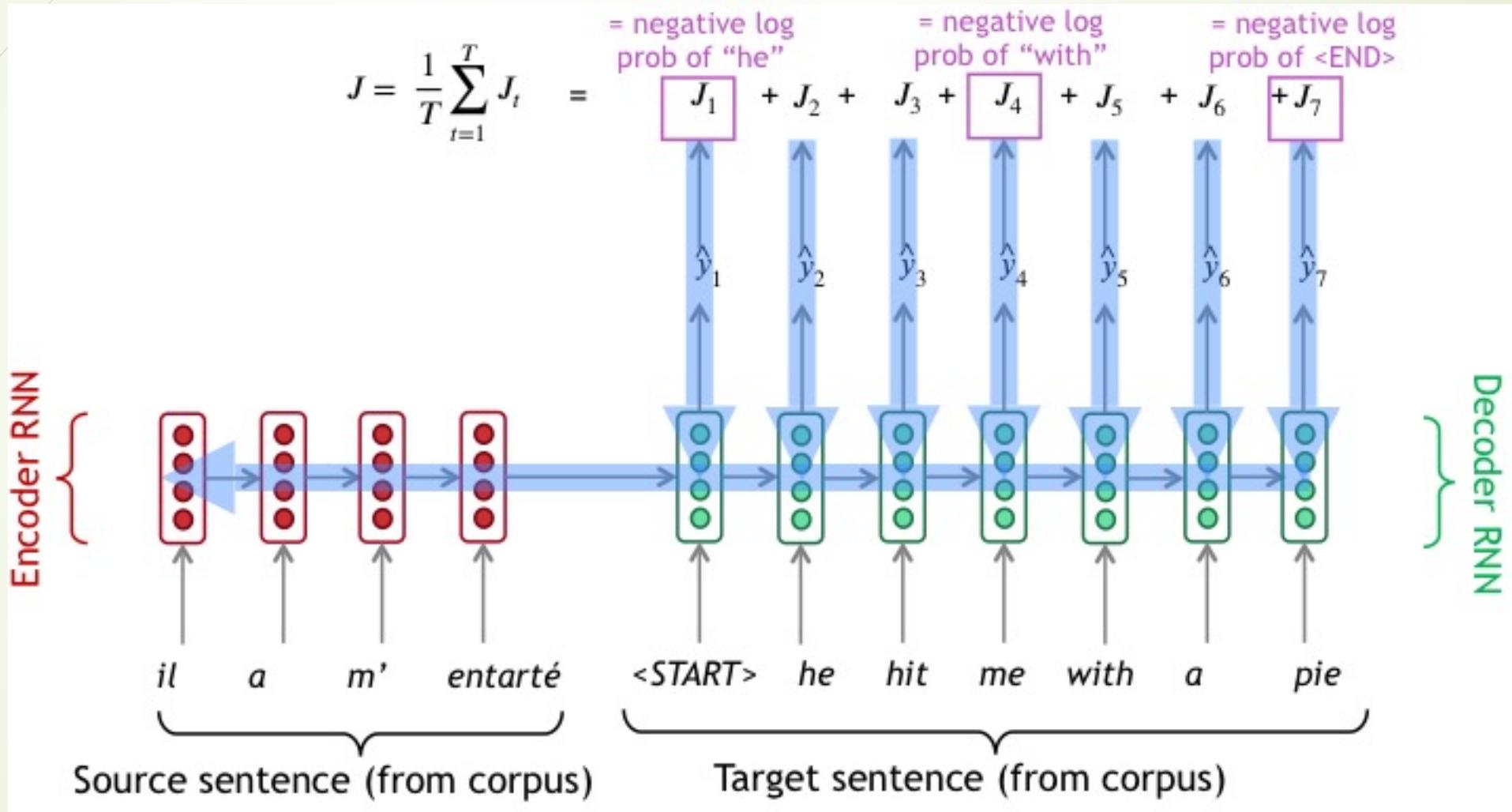
Note: This diagram shows test time behavior:  
decoder output is fed in ..... as next step's input

# Sequence-to-sequence is versatile!

- ▶ Sequence-to-sequence is useful for more than just MT
- ▶ Many NLP tasks can be phrased as sequence-to-sequence:
  - ▶ Summarization (long text → short text)
  - ▶ Dialogue (previous utterances → next utterance)
  - ▶ Parsing (input text → output parse as sequence)
  - ▶ Code generation (natural language → Python code)

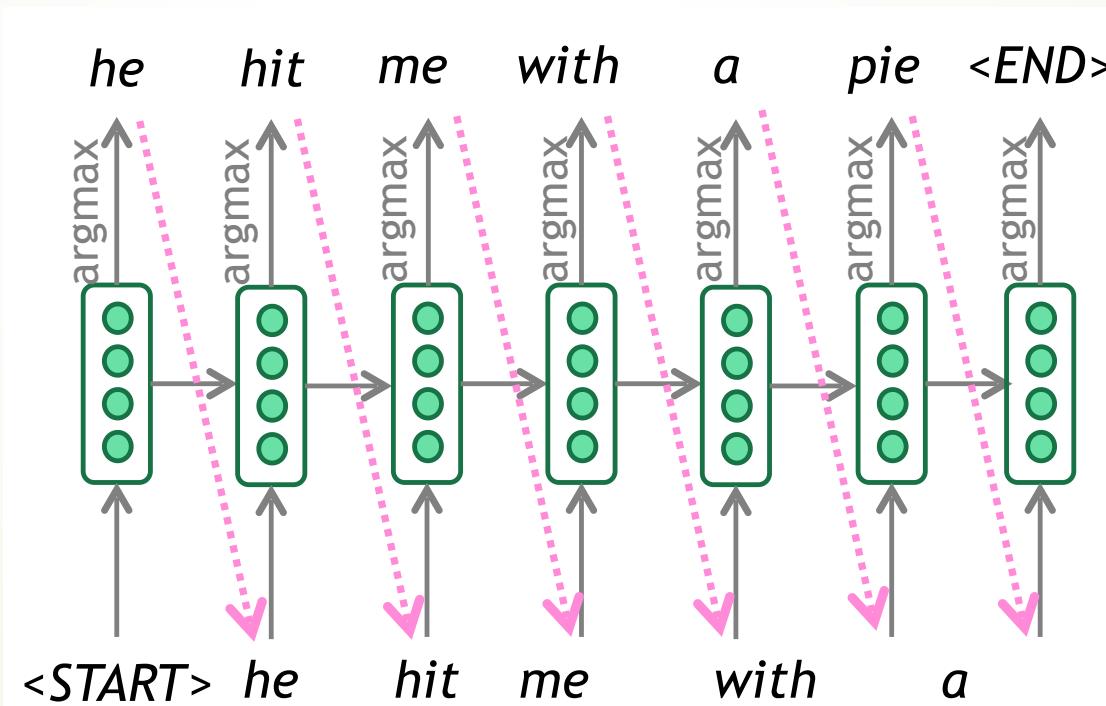


# Training a NMT system by BP



# Greedy Decoding

- ▶ We generate (or “decode”) the target sentence by taking **argmax** on each step of the decoder, called **greedy decoding** (take most probable word on each step)
- ▶ It may not correct once wrong decisions are made



# Beam Search Decoding

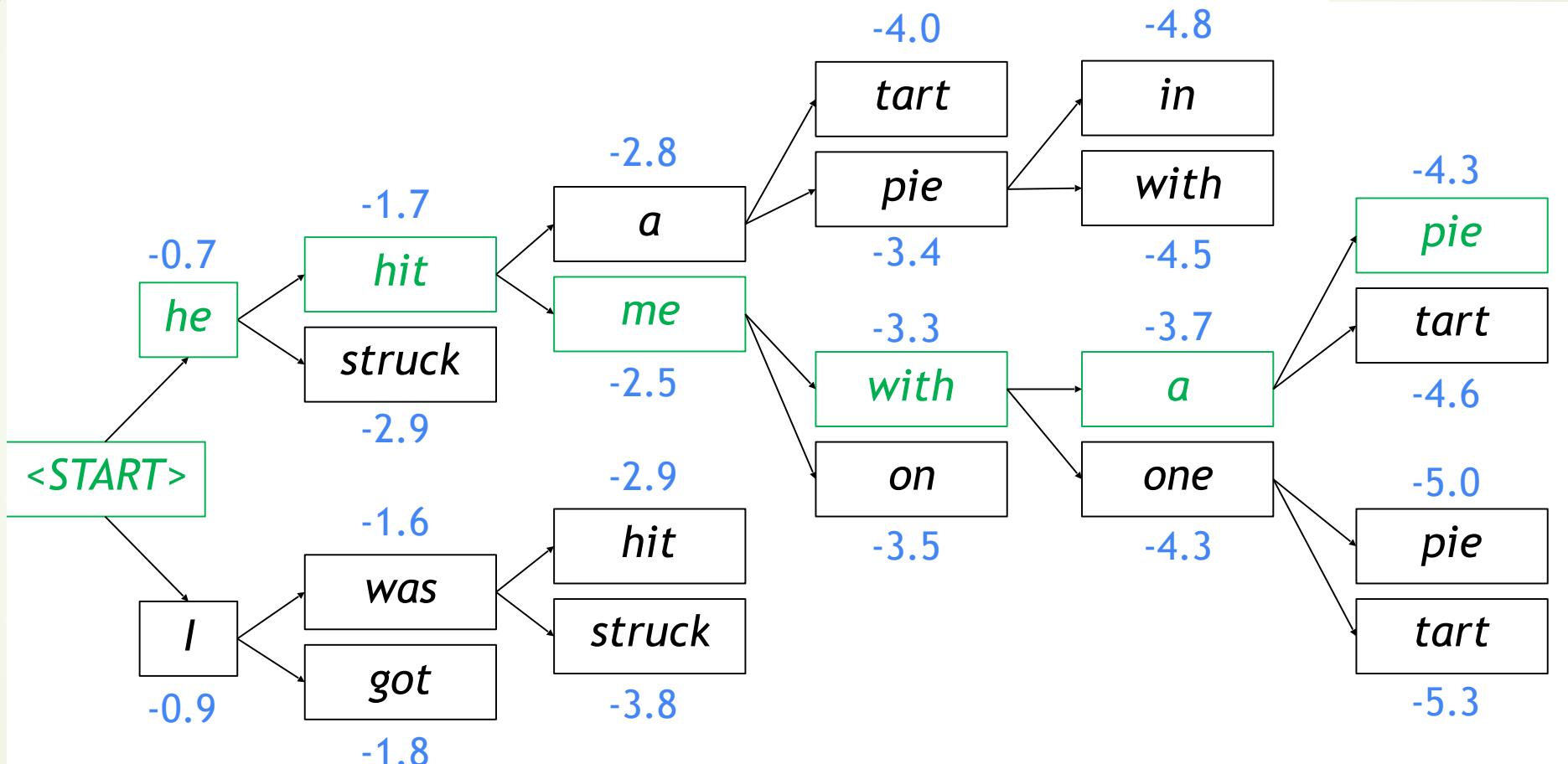
- ▶ Core idea: On each step of decoder, keep track of the  **$k$  most probable** partial translations (which we call *hypotheses*)
  - ▶  $k$  is the beam size (in practice around 5 to 10)
- ▶ A hypothesis  $(y(1), \dots, y(t))$  has a score which is its log probability:

$$\text{score}(y_1, \dots, y_t) = \log P_{\text{LM}}(y_1, \dots, y_t | x) = \sum_{i=1}^t \log P_{\text{LM}}(y_i | y_1, \dots, y_{i-1}, x)$$

- ▶ Scores are all negative, and higher score is better
- ▶ We search for high-scoring hypotheses, tracking top  $k$  on each step
- ▶ Beam search is not guaranteed to find optimal solution
- ▶ But much more efficient than exhaustive search!

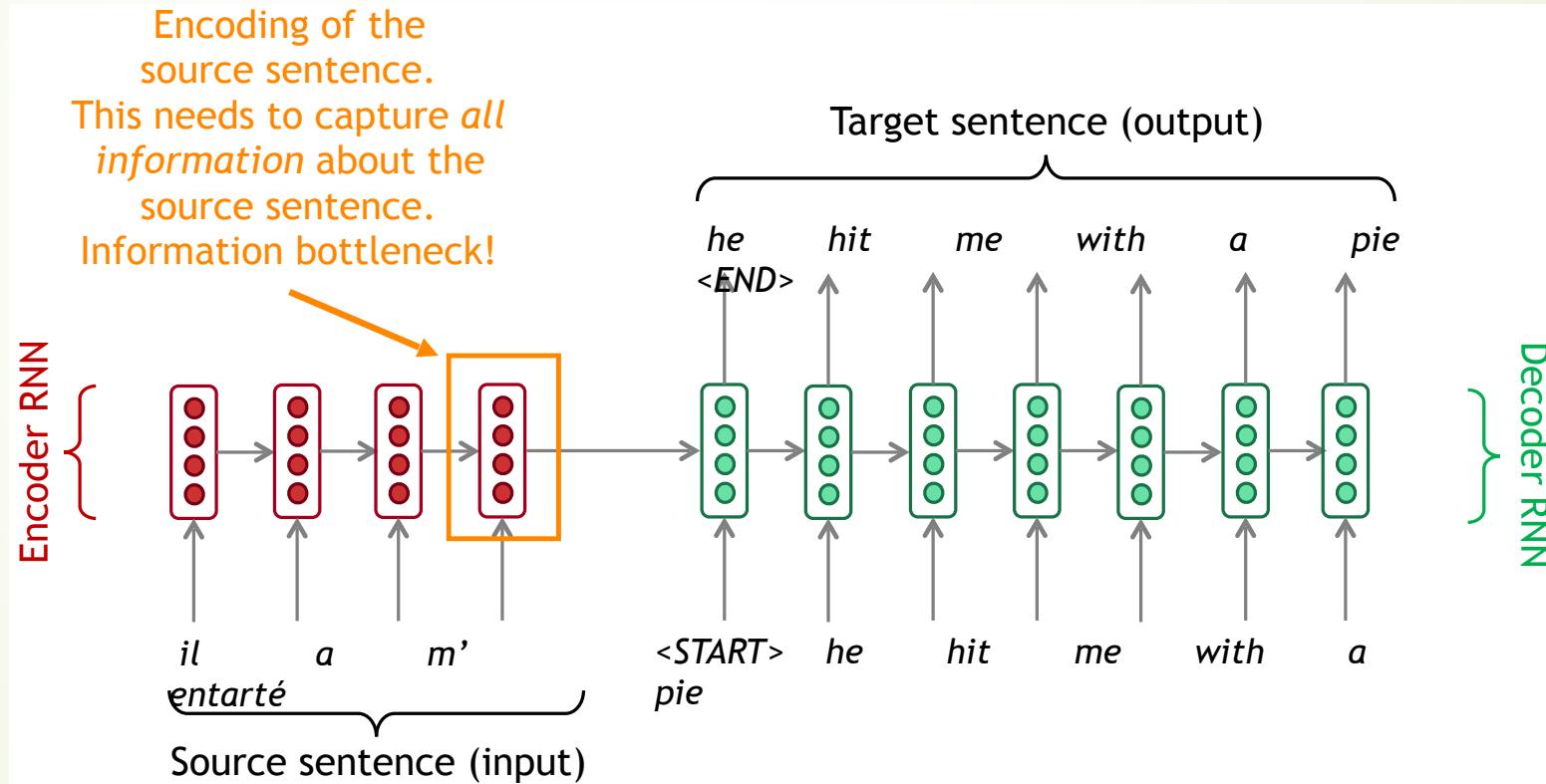
# Beam search decoding example:

Beam size =  $k = 2$ . Blue numbers =  $\text{score}(y_1, \dots, y_t) = \sum_{i=1}^t \log P_{\text{LM}}(y_i | y_1, \dots, y_{i-1}, x)$



For each of the  $k$  hypotheses, find top  $k$  next words and calculate scores

# Sequence-to-sequence: the bottleneck problem

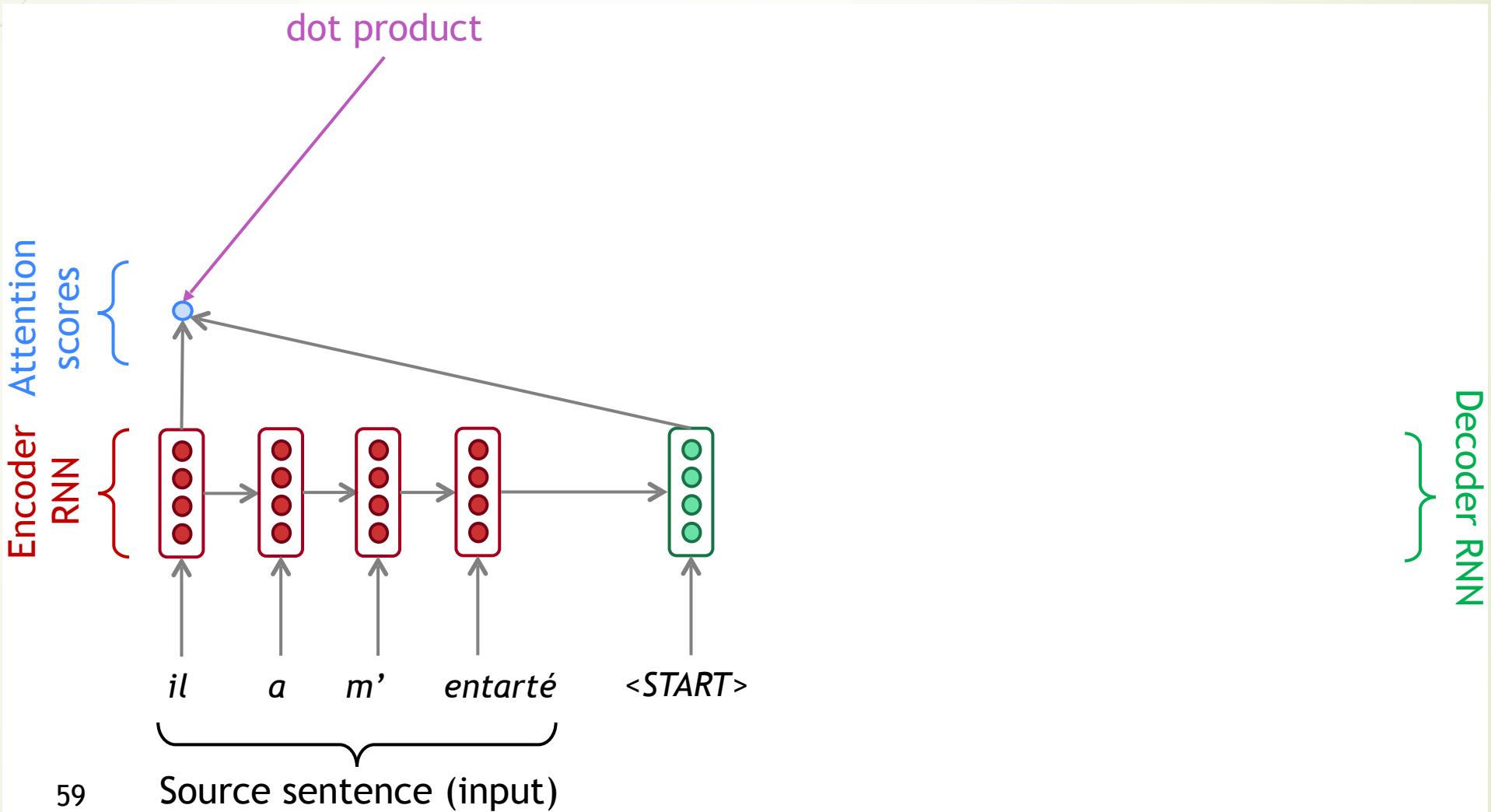


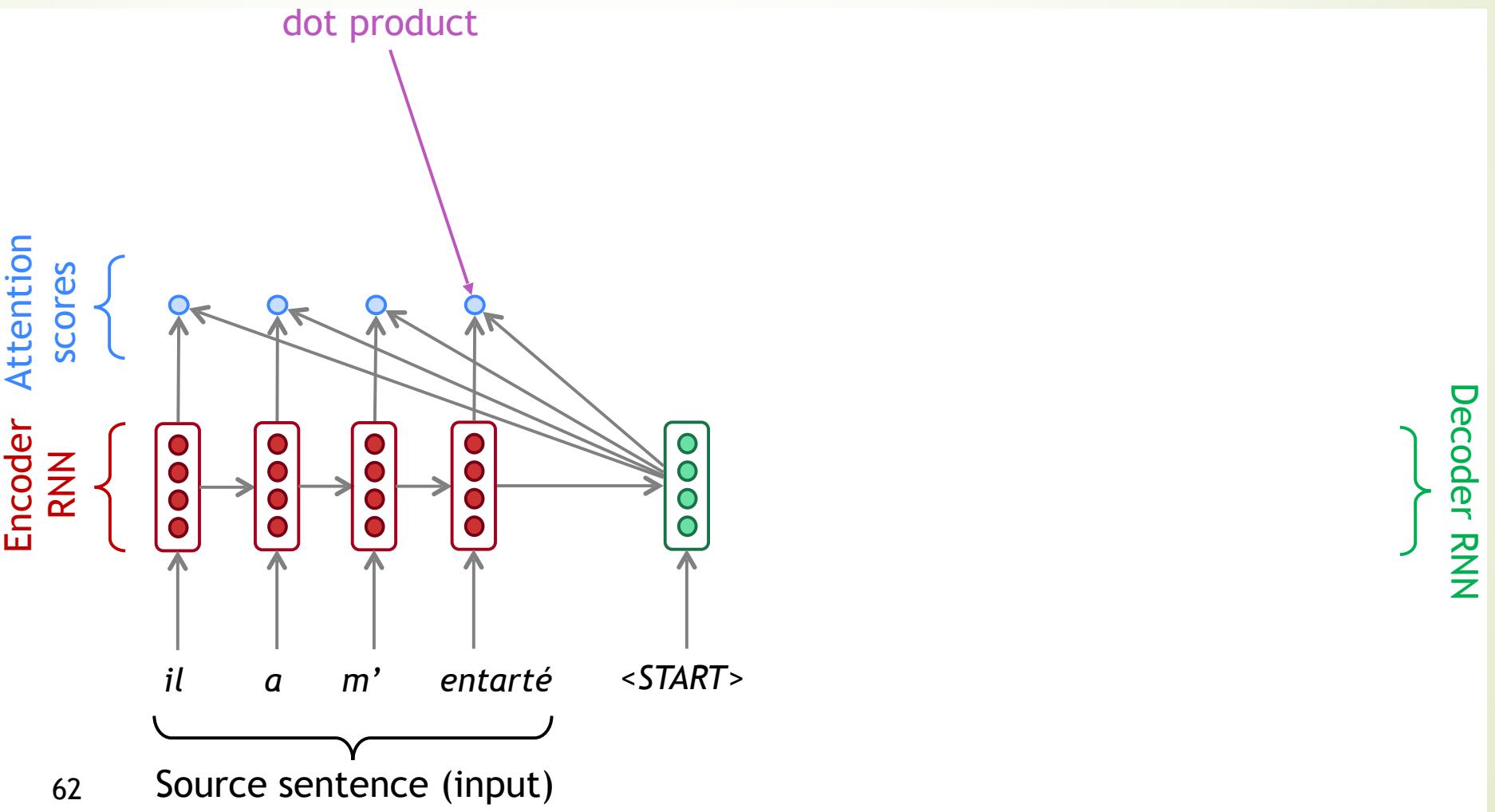


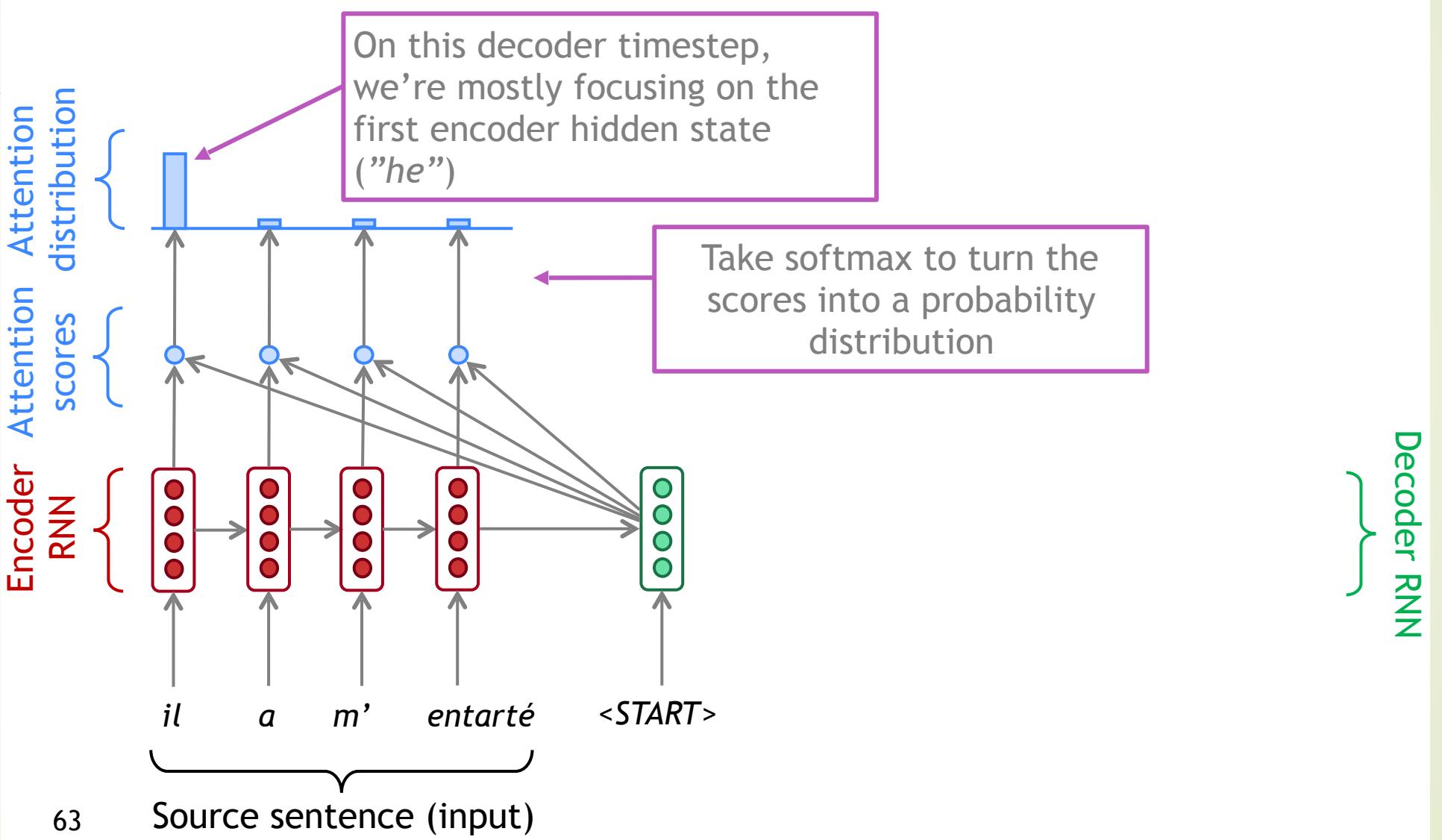
# Attention Mechanism

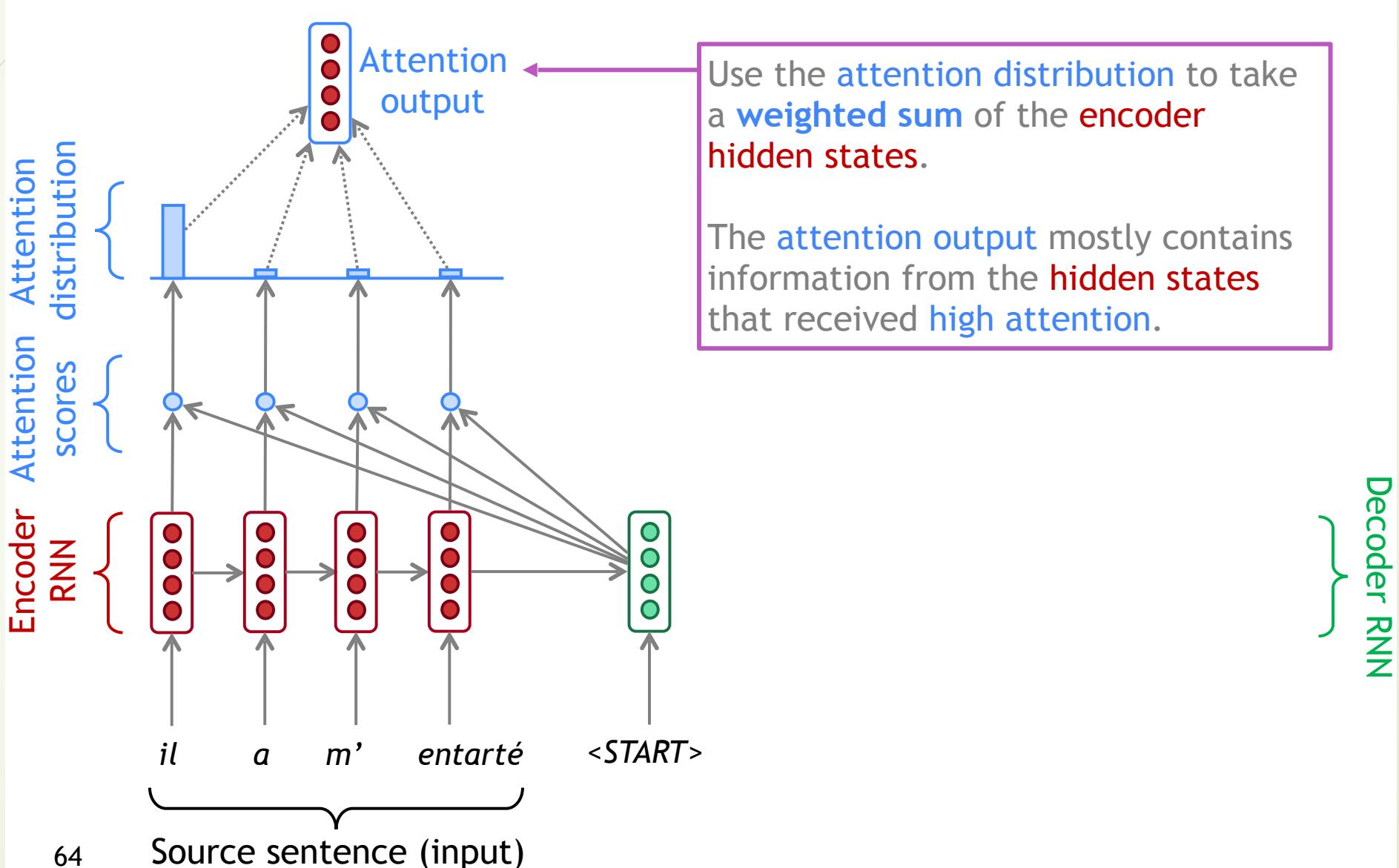
It was firstly invented in computer vision, then to NLP.

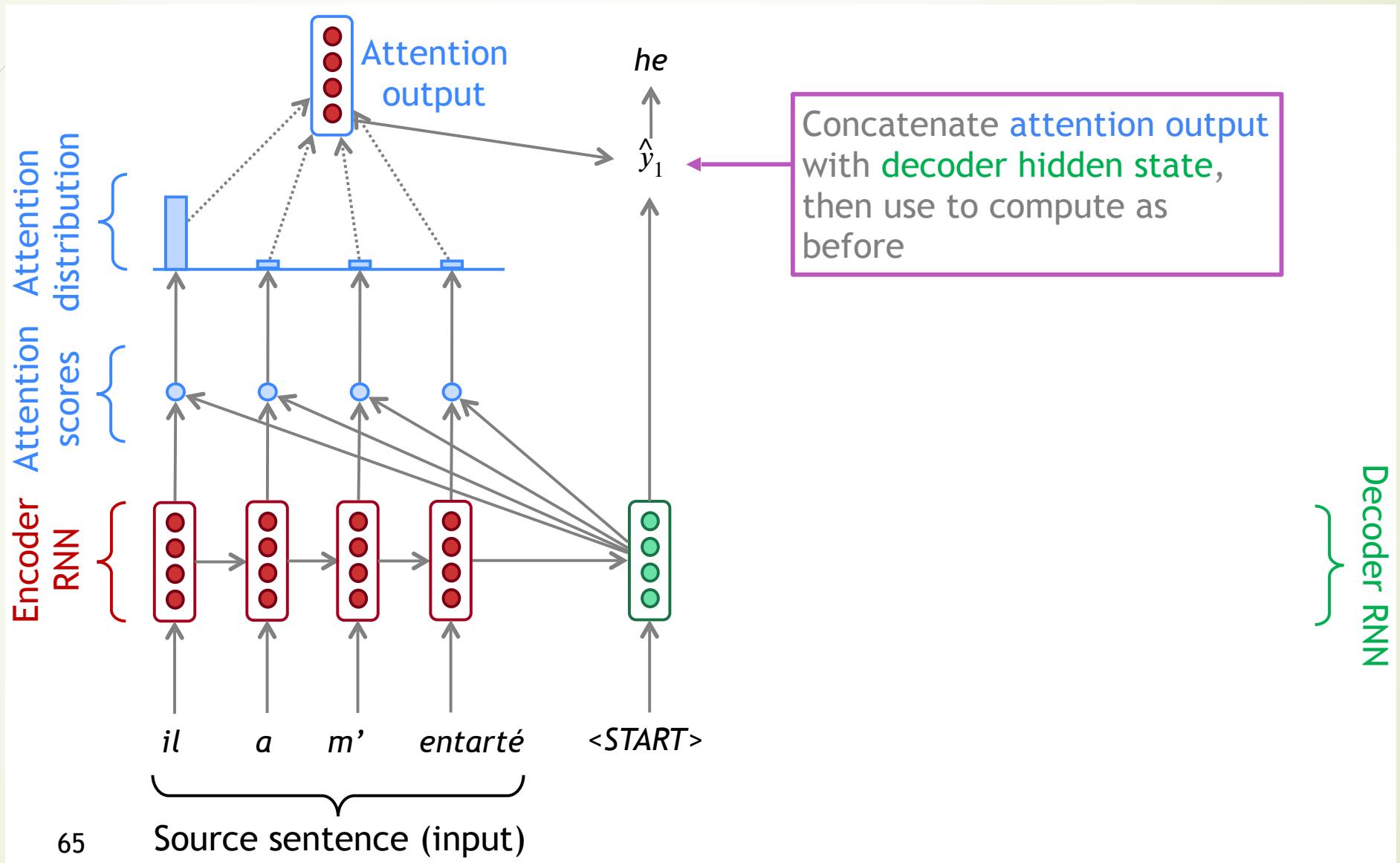
# Sequence-to-sequence with attention

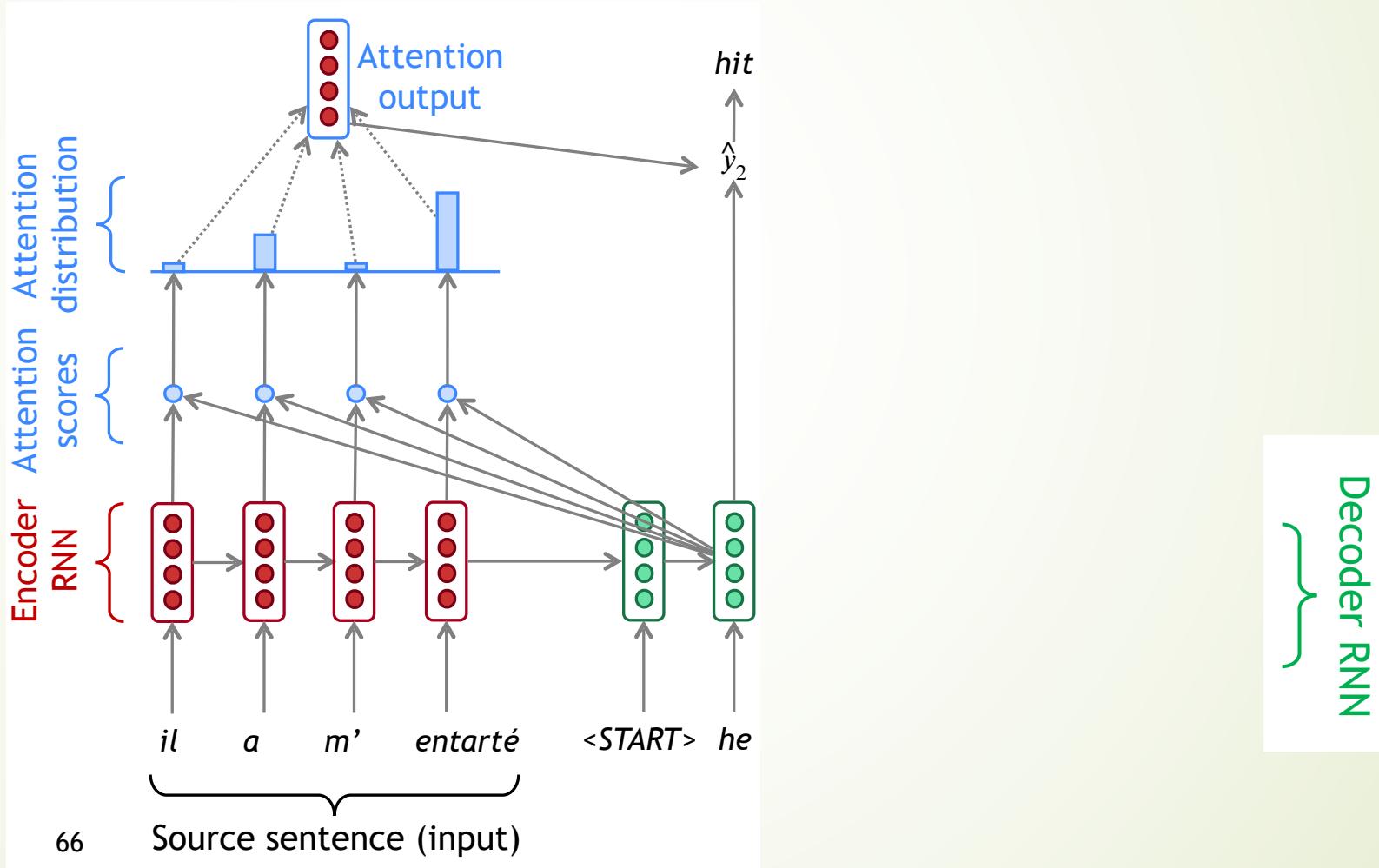


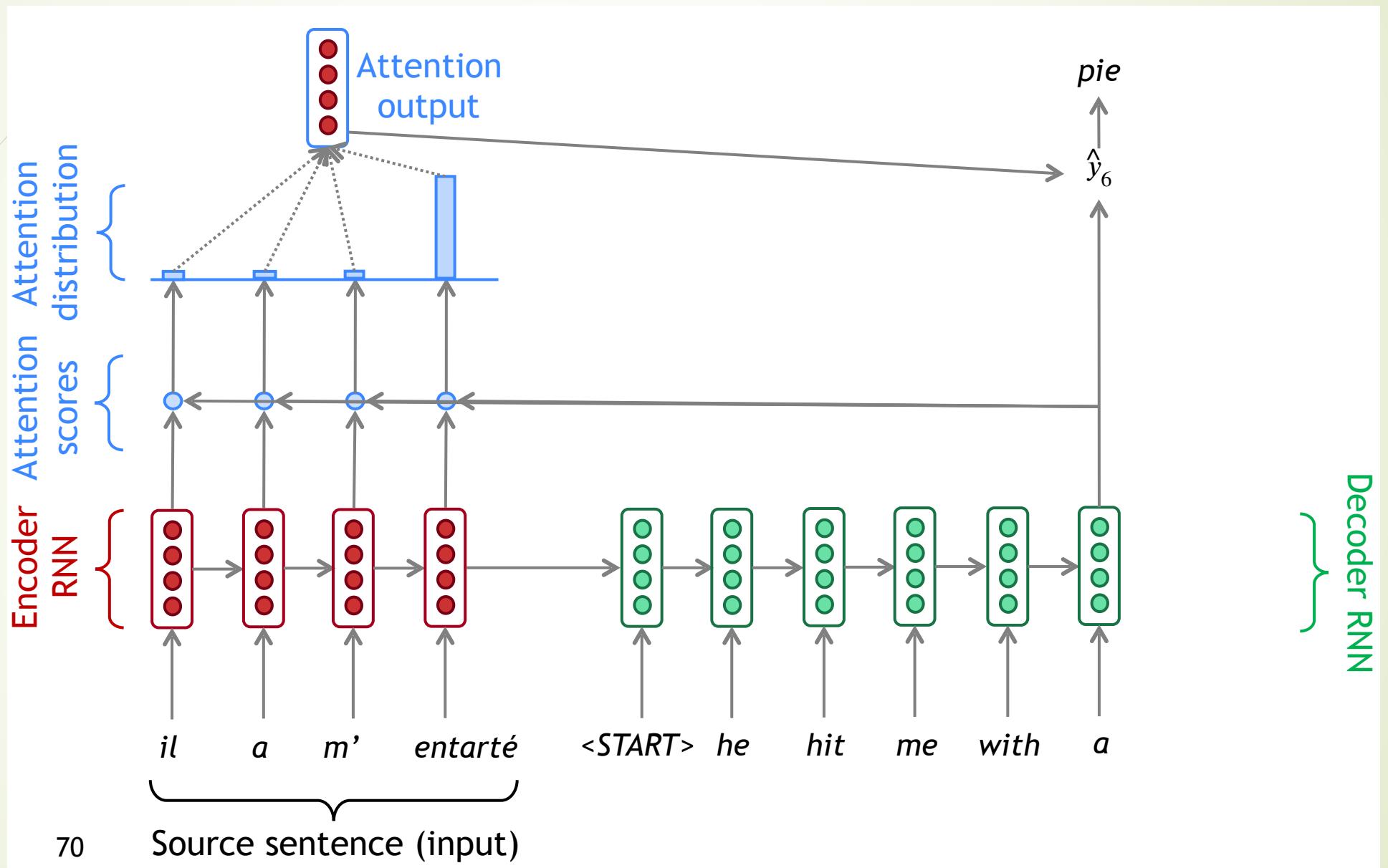












# Attention in Equations

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

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

- We take softmax to get the attention distribution  $\alpha^t$  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

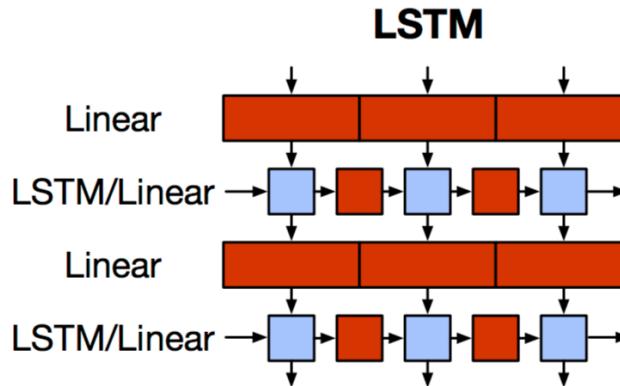
$$a_t = \sum_{i=1}^N \alpha_i^t h_i \in \mathbb{R}^h$$

- Finally we concatenate the attention output  $a_t$  with the decoder hidden state  $s_t$  and proceed as in the non-attention seq2seq model

$$[a_t; s_t] \in \mathbb{R}^{2h}$$

# Motivation of Transformer

- We want **parallelization** but RNNs are inherently sequential



- Despite LSTMs, RNNs generally need attention mechanism to deal with long range dependencies – **path length** between states grows with distance otherwise
- But if **attention** gives us access to any state... maybe we can just use attention and don't need the RNN? 
- And then NLP can have deep models ... and solve our vision envy

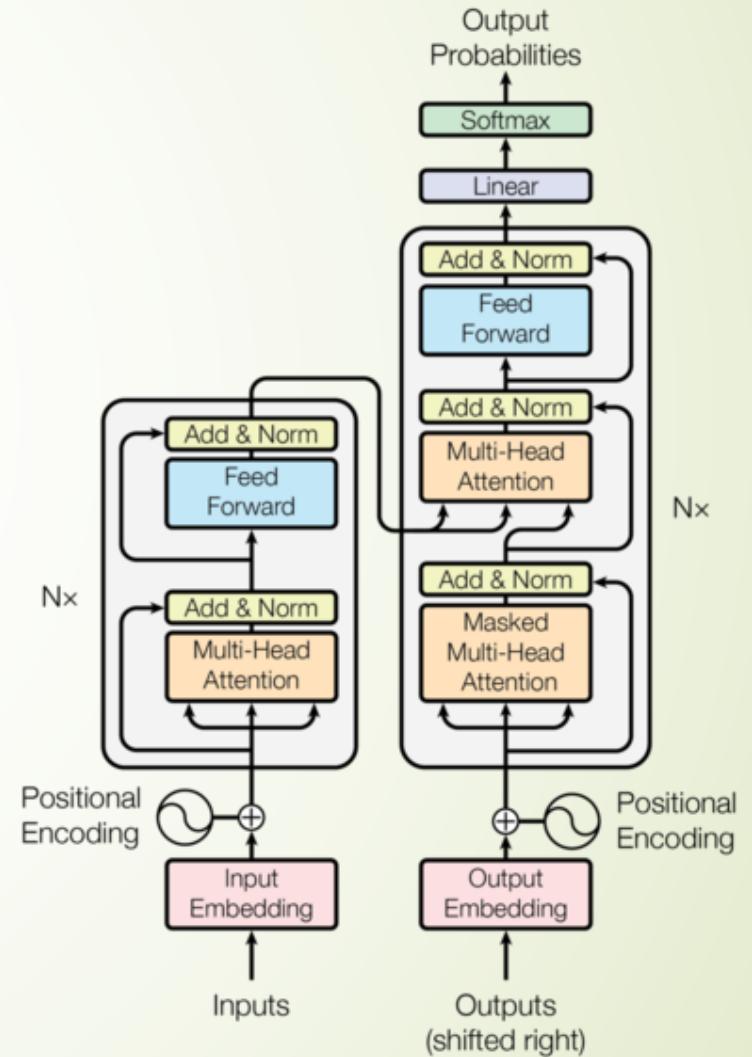


# Transformer

“Attention is all you need”

# Transformer (Vaswani et al. 2017) “Attention is all you need”

- ▶ <https://arxiv.org/pdf/1706.03762.pdf>
- ▶ **Non-recurrent** sequence-to-sequence model
- ▶ A **deep** model with a sequence of **attention**-based transformer blocks
- ▶ Depth allows a certain amount of lateral information transfer in understanding sentences, in slightly unclear ways
- ▶ Final cost/error function is standard cross-entropy error on top of a softmax classifier
- ▶ Initially built for NMT:
  - ▶ Task: machine translation with parallel corpus
  - ▶ Predict each translated word

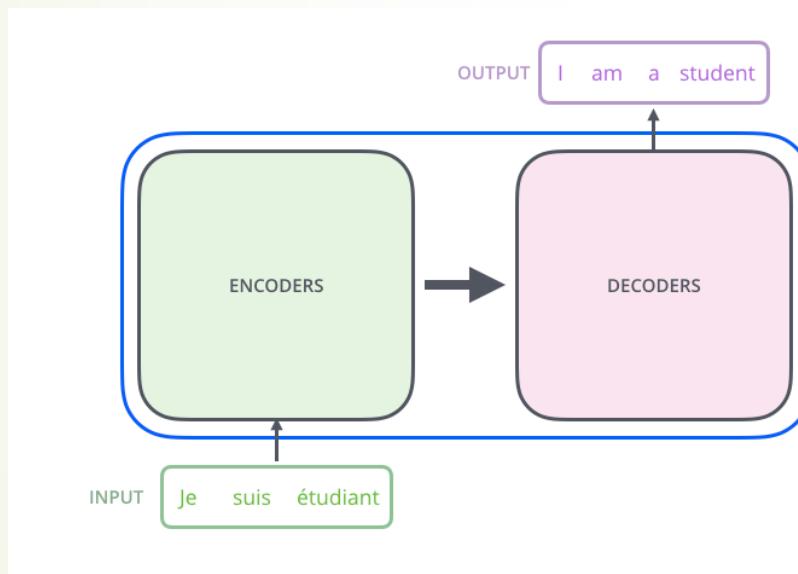


# Transformer Pytorch Notebook

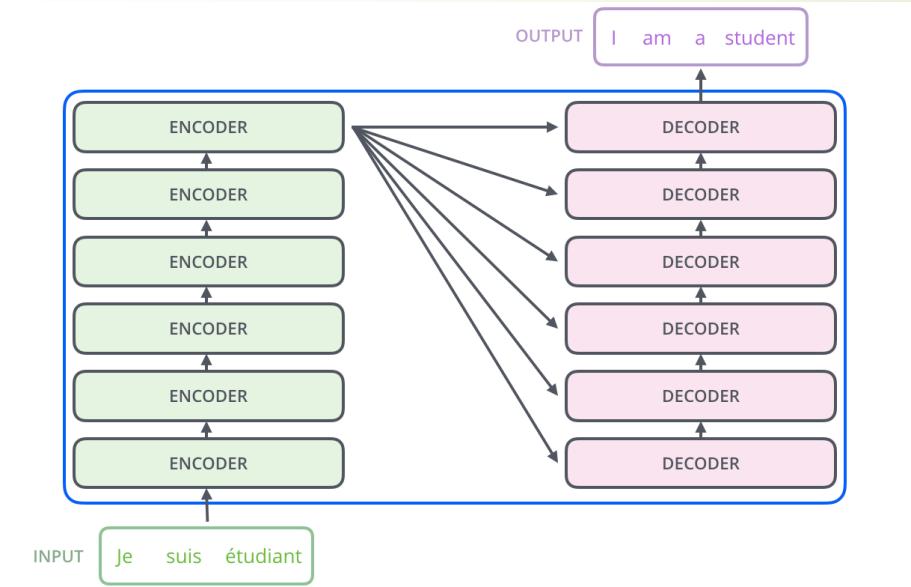
- ▶ Learning about transformers on your own?
- ▶ Key recommended resource:
  - ▶ <http://nlp.seas.harvard.edu/2018/04/03/attention.html>
  - ▶ The Annotated Transformer by Sasha Rush, a Jupyter Notebook using PyTorch that explains everything!
- ▶ <https://jalammar.github.io/illustrated-transformer/>
- ▶ Illustrated Transformer by Jay Alammar, a Cartoon about Transformer with attention visualization notebook based on Tensor2Tensor.

# Encoder-Decoder Blocks

Encoder-Decoder

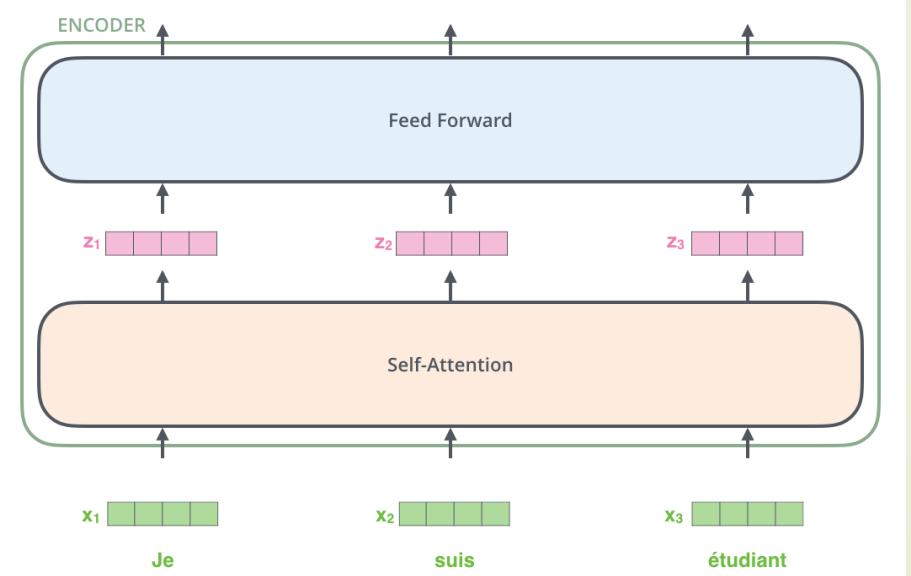
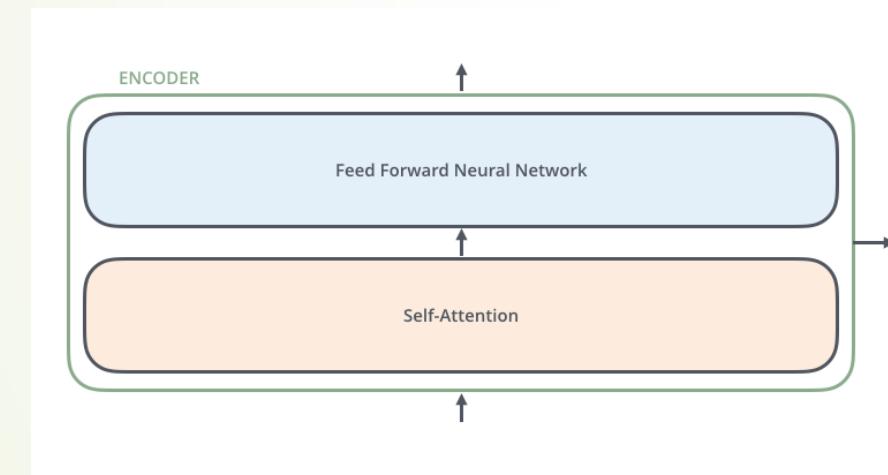


N=6 layers



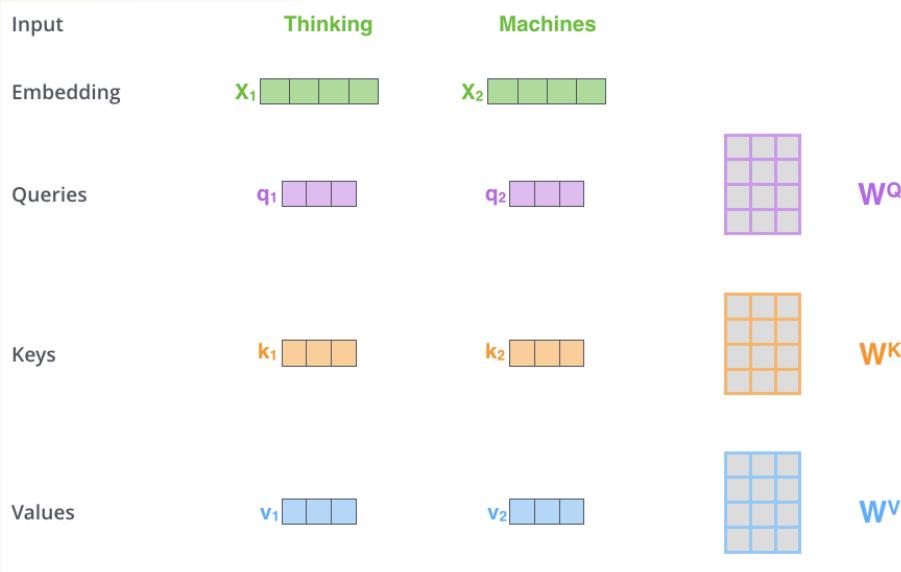
# Encoder has two layers

Self-Attention +  
FeedForward

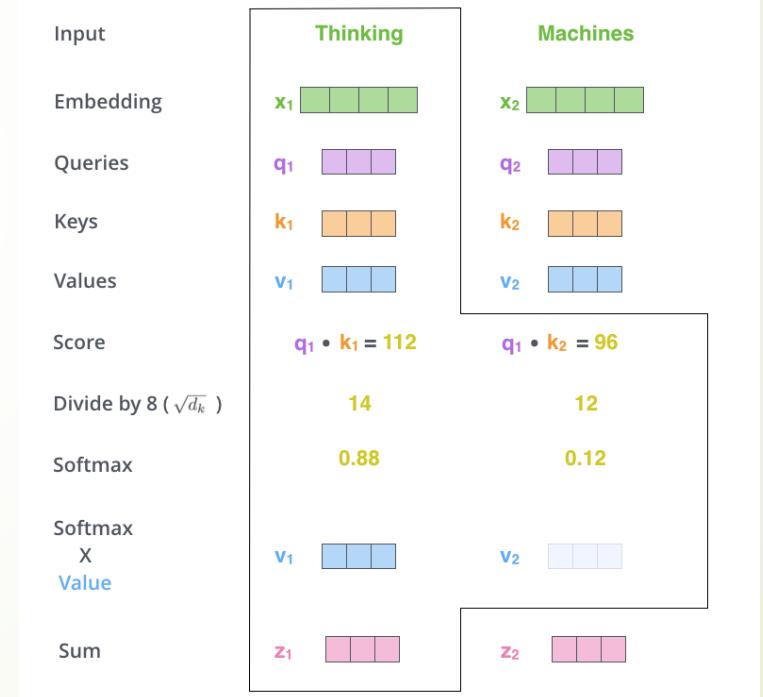


# Attention Illustration

## Embedding->(q,k,v)



## Dot-Product Attention



# Dot-Product Self-Attention: Definition

- ▶ Inputs: a query  $q$  and a set of key-value (k-v) pairs, to an output
- ▶ Query, keys, values, and output are all vectors
- ▶ Output is weighted sum of values, where
  - ▶ Weight of each value is computed by an inner product of query and corresponding key
  - ▶ Queries and keys have same dimensionality  $d_k$ , value have  $d_v$

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

# Dot-Product Attention: Matrix Form

- When we have multiple queries  $q$ , we stack them in a matrix  $Q$ :

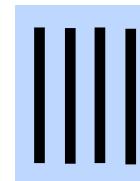
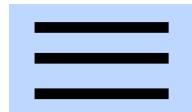
$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$



$$A(Q, K, V) = \text{softmax}(QK^T)V$$

$$[|Q| \times d_k] \times [d_k \times |K|] \times [|K| \times d_v]$$

softmax  
row-wise

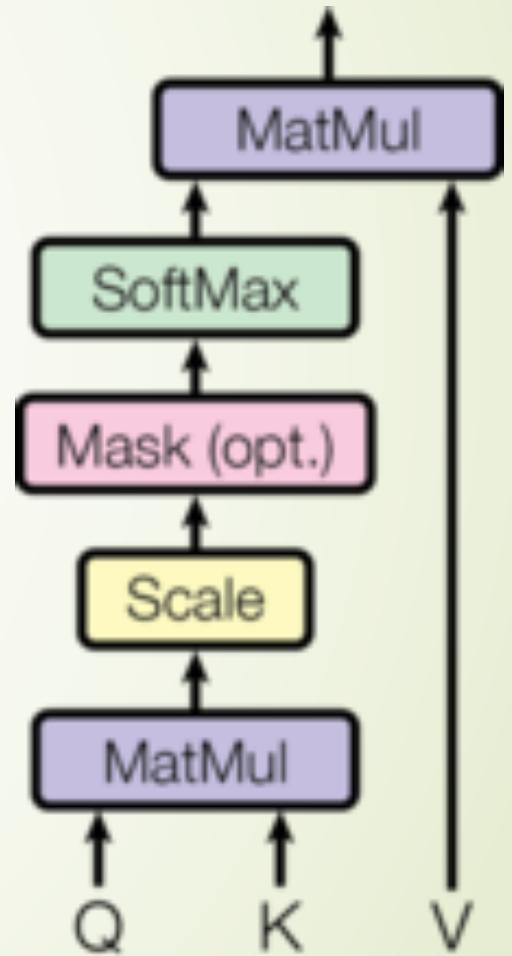


$$= [|Q| \times d_v]$$

# Scaled Dot-Product Attention

- **Problem:** As  $d_k$  gets large, the variance of  $q^T k$  increases
- some values inside the softmax get large
- the softmax gets very peaked
- hence its gradient gets smaller.
  
- Solution: Scale by length of query/key vectors:

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



# Attention: Multiple Inputs

Matrix input

$$\mathbf{X} \times \mathbf{W}^Q = \mathbf{Q}$$

$$\mathbf{X} \times \mathbf{W}^K = \mathbf{K}$$

$$\mathbf{X} \times \mathbf{W}^V = \mathbf{V}$$

Scaled dot-product

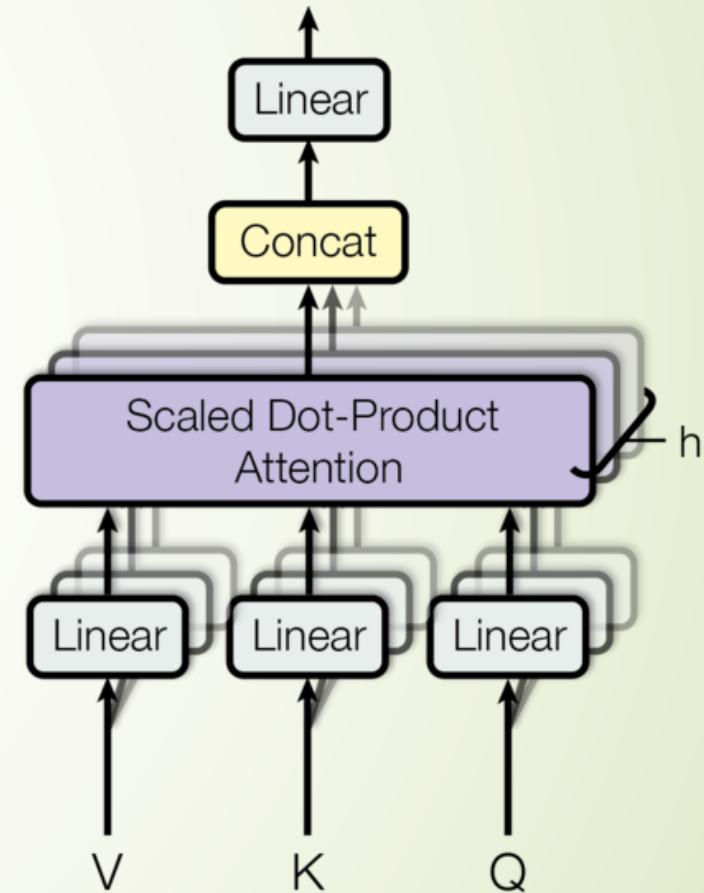
$$\text{softmax}\left(\frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d_k}}\right) \mathbf{V} = \mathbf{Z}$$

# Multi-head Attention

- ▶ **Problem** with simple self-attention:
  - ▶ Only one way for words to interact with one-another
- ▶ **Solution:** Multi-head attention
  - ▶ First map Q, K, V into h=8 many lower dimensional spaces via W matrices
  - ▶ Then apply attention, then concatenate outputs and pipe through linear layer
  - ▶ Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

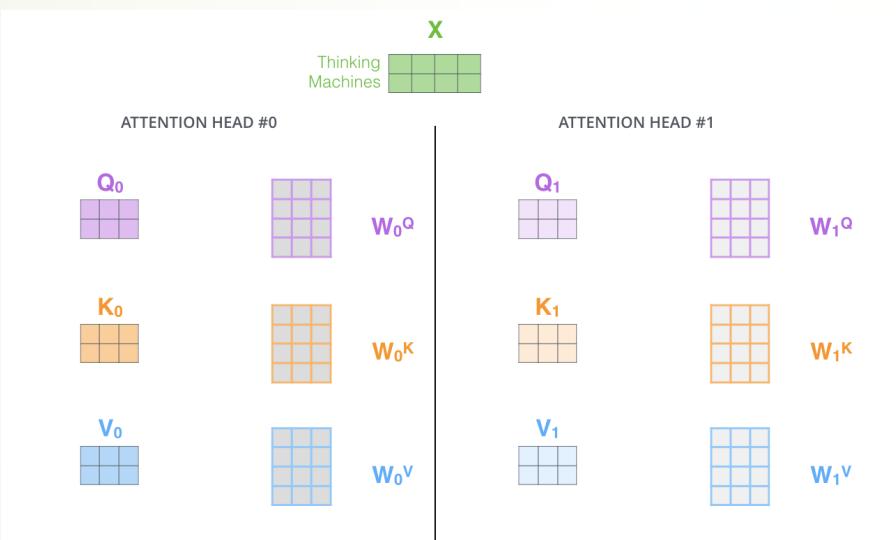
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

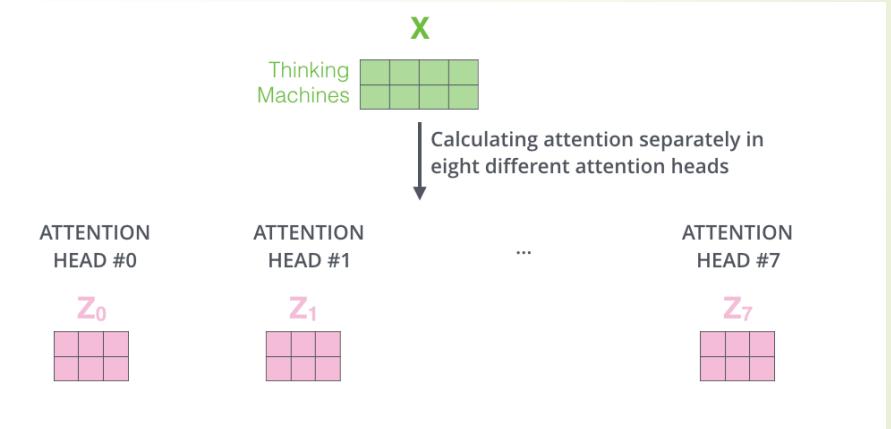


# Multihead

2 heads



$h=8$  heads



## Concatenation

1) Concatenate all the attention heads



3) The result would be the  $Z$  matrix that captures information from all the attention heads. We can send this forward to the FFNN

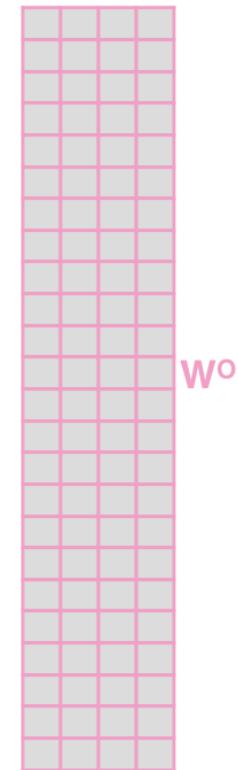
$$= \begin{matrix} Z \\ \hline \end{matrix}$$

A horizontal line with a vertical bar above it, followed by a 4x4 grid of pink squares representing the concatenated matrix  $Z$ .

## Linear

2) Multiply with a weight matrix  $W^o$  that was trained jointly with the model

$X$



# Multi-head Attention

1) This is our input sentence\*

2) We embed each word\*

3) Split into 8 heads.  
We multiply  $X$  or  $R$  with weight matrices

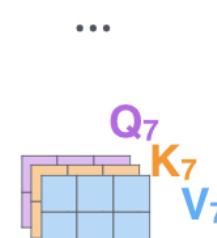
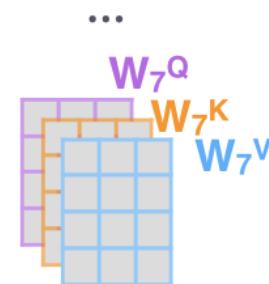
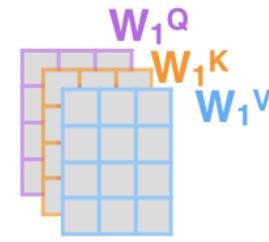
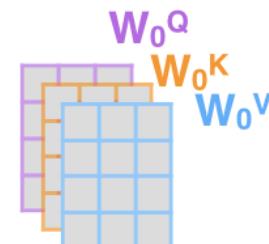
4) Calculate attention using the resulting  $Q/K/V$  matrices

5) Concatenate the resulting  $Z$  matrices, then multiply with weight matrix  $W^O$  to produce the output of the layer

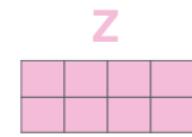
Thinking  
Machines



\* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



$W^O$

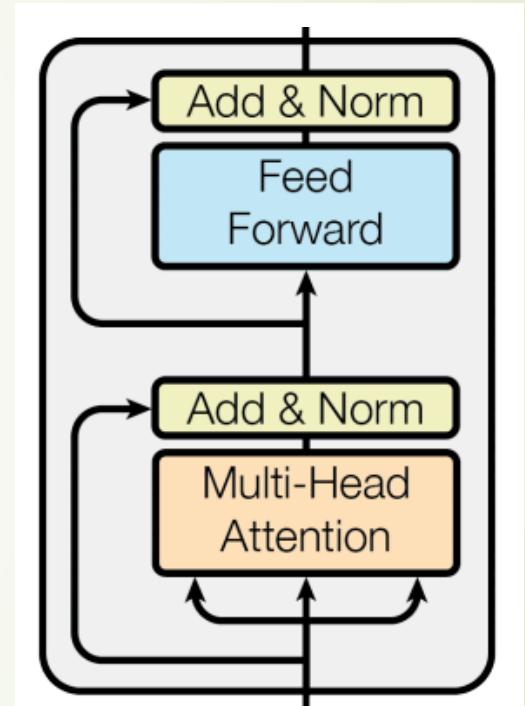


# A Transformer block

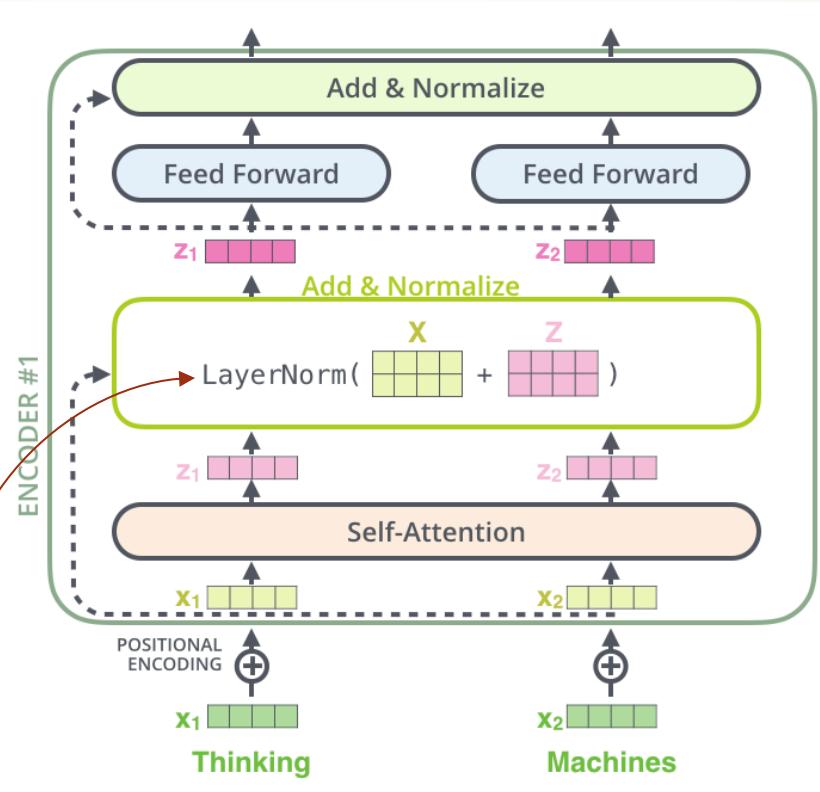
- ▶ Each block has two “sublayers”
  - ▶ Multihead attention
  - ▶ 2-layer feed-forward NNet (with ReLU)
- ▶ Each of these two steps also has:
  - ▶ Residual (short-cut) connection:  $x + \text{sublayer}(x)$
  - ▶ LayerNorm( $x + \text{sublayer}(x)$ ) changes input features to have mean 0, variance 1, and adds two more parameters (Ba et al. 2016)

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

$$h_i = f\left(\frac{g_i}{\sigma_i} (a_i - \mu_i) + b_i\right)$$



# Residue (Shortcut)



$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

$$h_i = f\left(\frac{g_i}{\sigma_i} (a_i - \mu_i) + b_i\right)$$

# Encoder Input

- Actual word representations are word pieces:  
byte pair encoding
  - Start with a vocabulary of characters
  - Most frequent ngram pairs  $\mapsto$  a new ngram
  - Example: "es, est" 9 times, "lo" 7 times
- Also added is a **positional encoding** so same words at different locations have different overall representations:

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

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

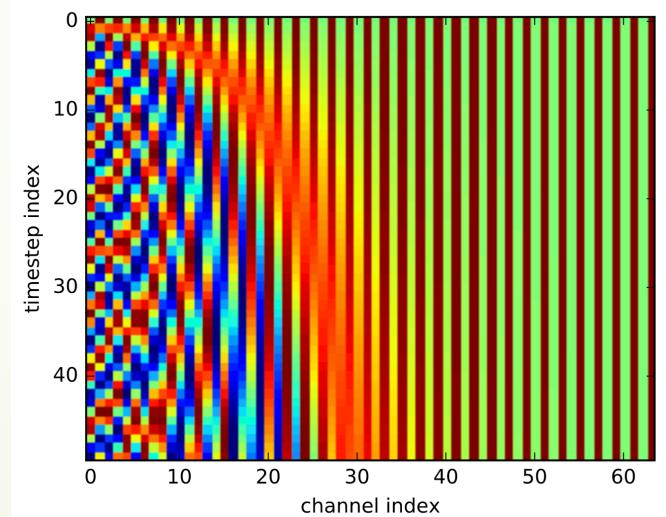
Or learned

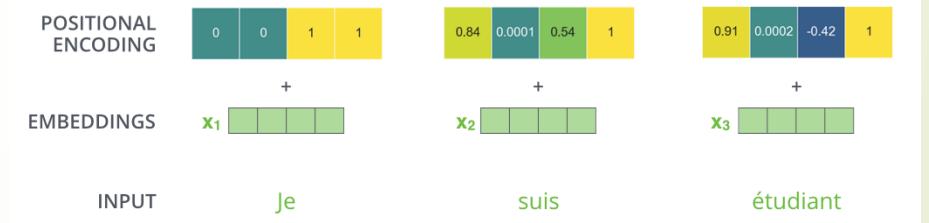
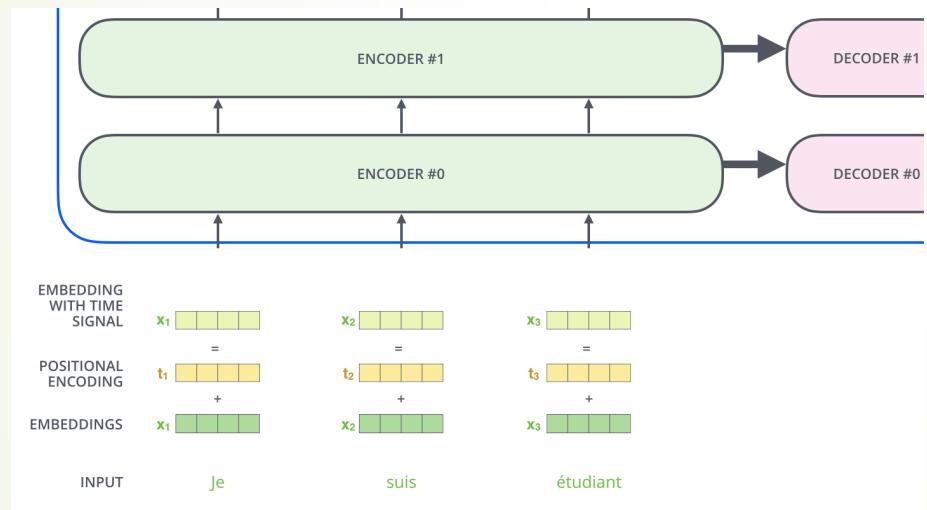
Dictionary

5	l o w
2	l o w e r
6	n e w e s t
3	w i d e s t

Vocabulary

l, o, w, e, r, n, w, s, t, i, d, es, est, lo





# Sin/Cos Position Encoding

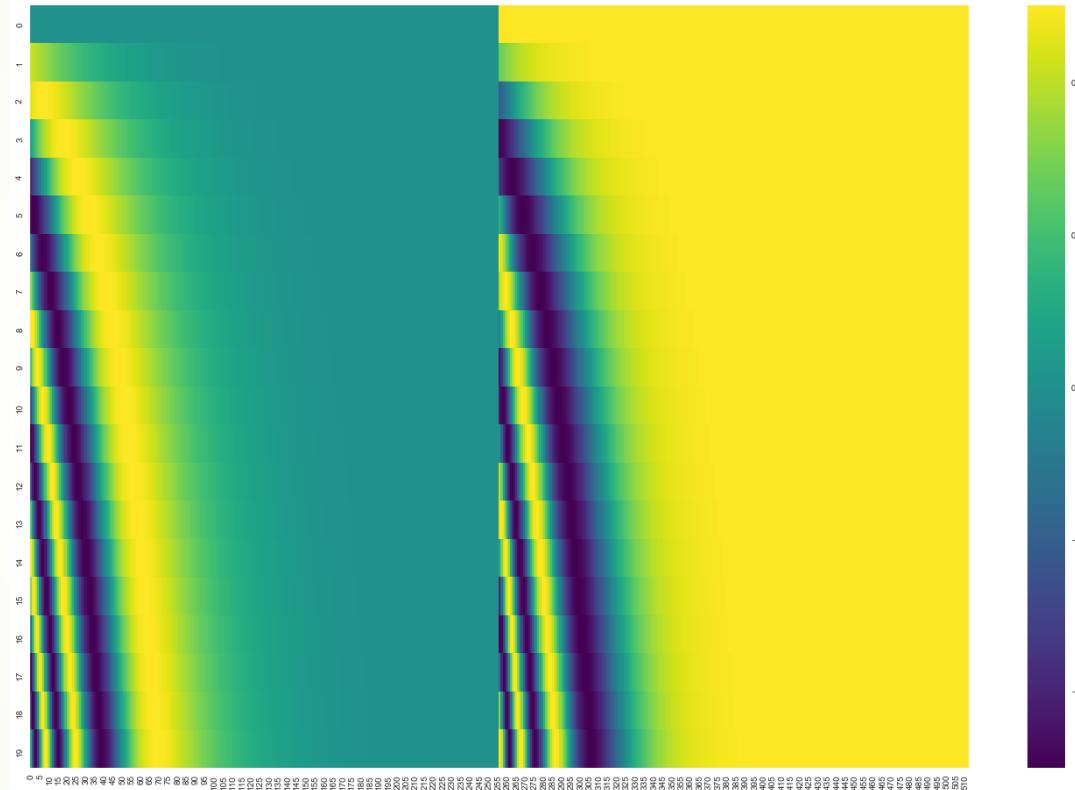
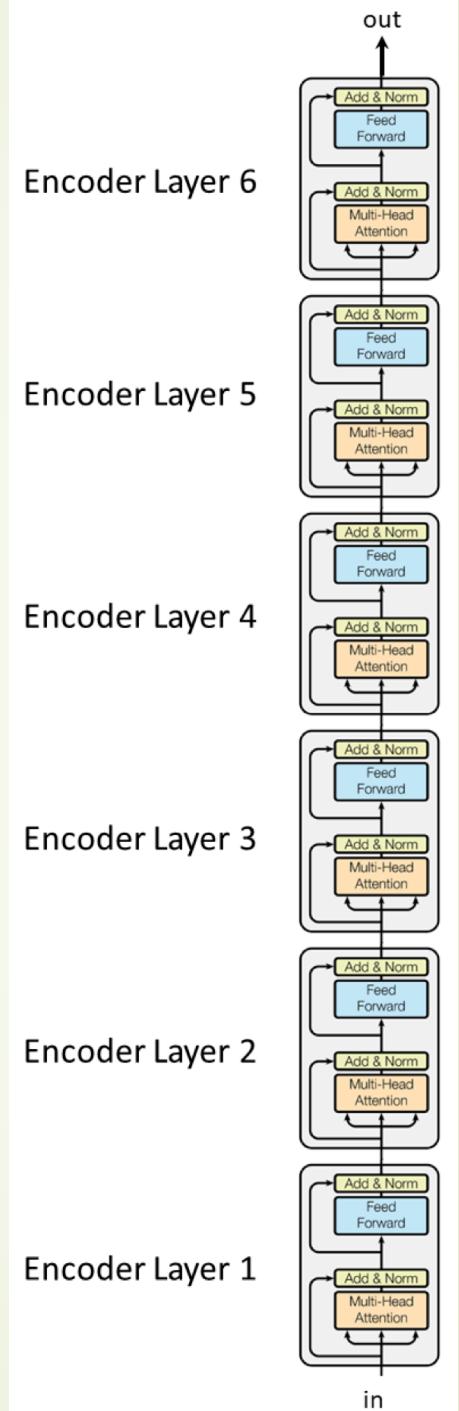
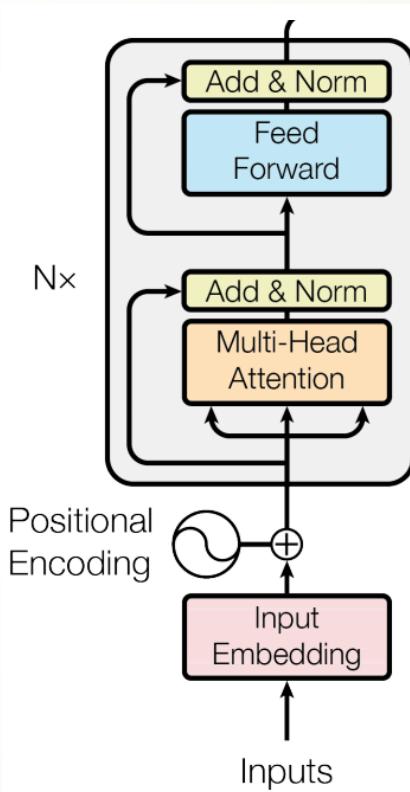


Figure. Each row corresponds to a positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.

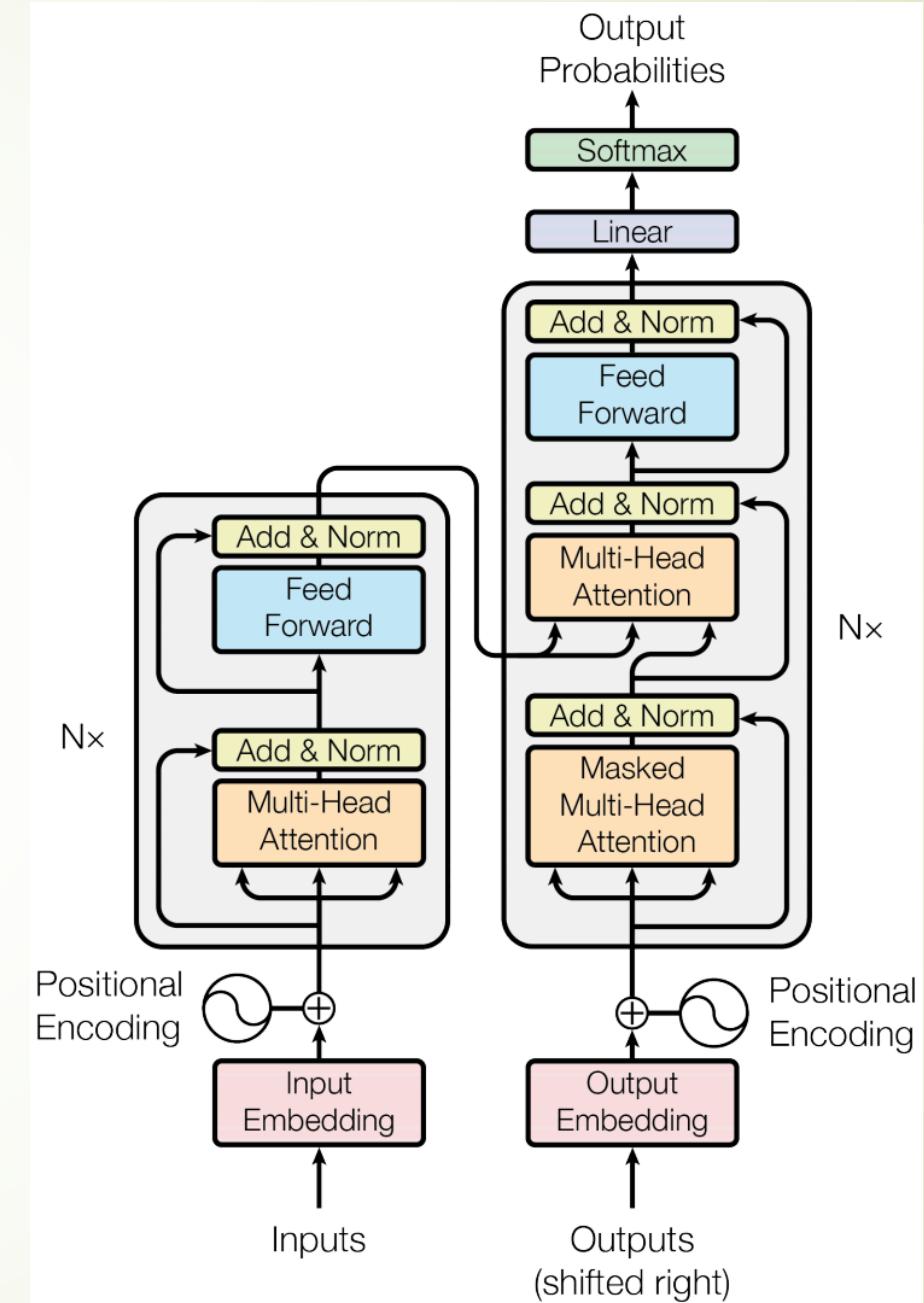
# Transformer Encoder

- Blocks are repeated N=6 or more times

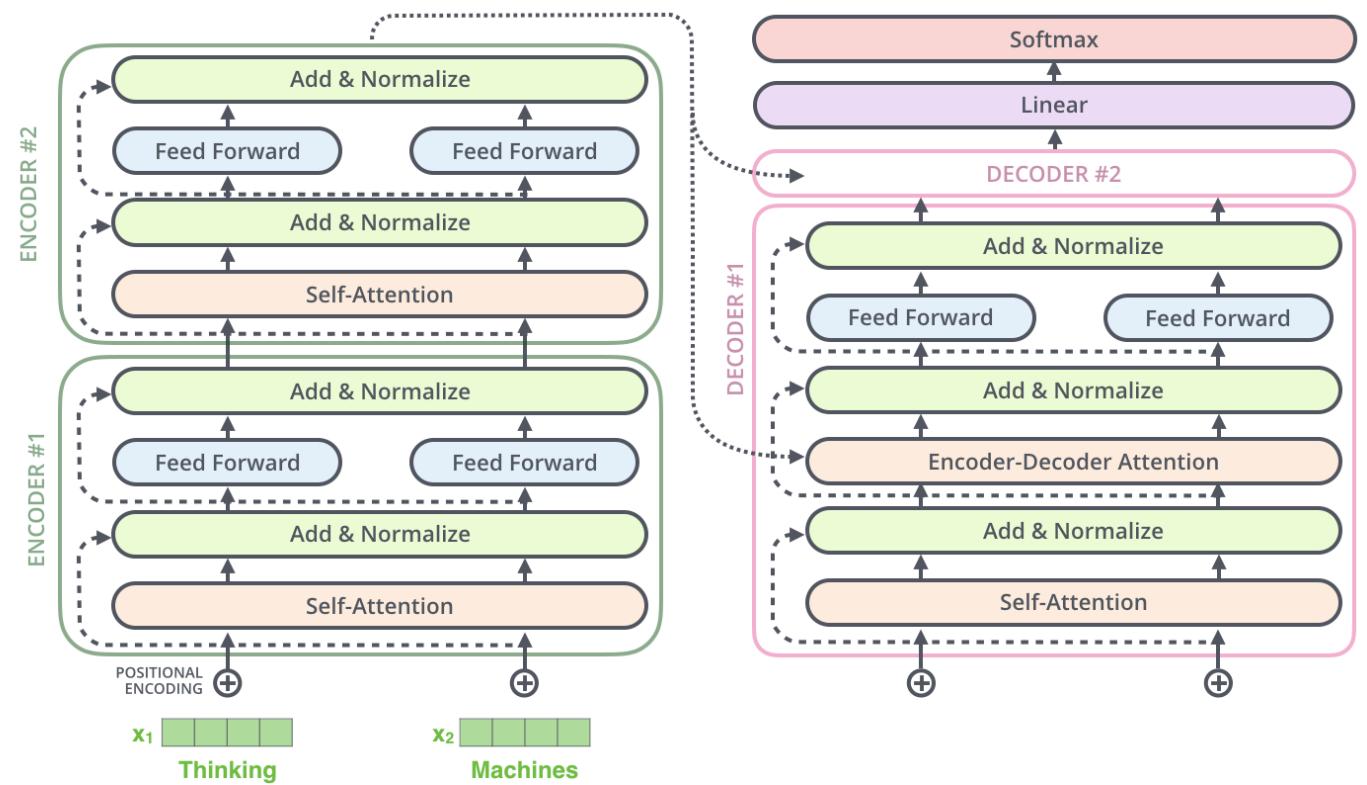
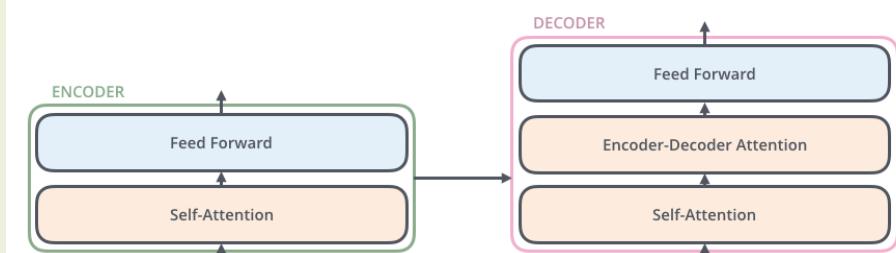


# Transformer Decoder

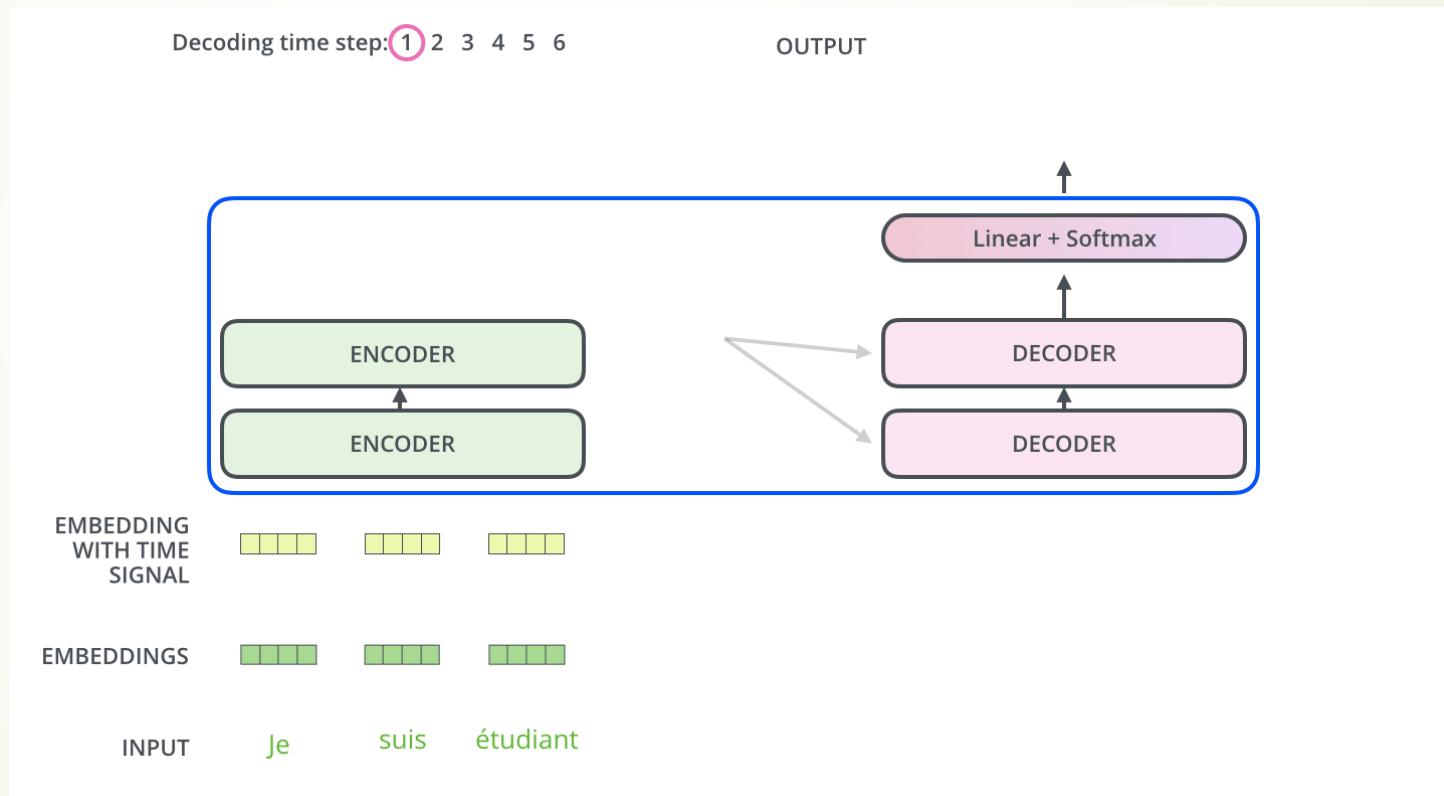
- ▶ 2 sublayer changes in decoder
  - ▶ Masked decoder self-attention on previously generated outputs
  - ▶ Encoder-Decoder Attention, where **queries** come from previous **decoder** layer and **keys and values** come from output of **encoder**
- ▶ Blocks repeated N=6 times also



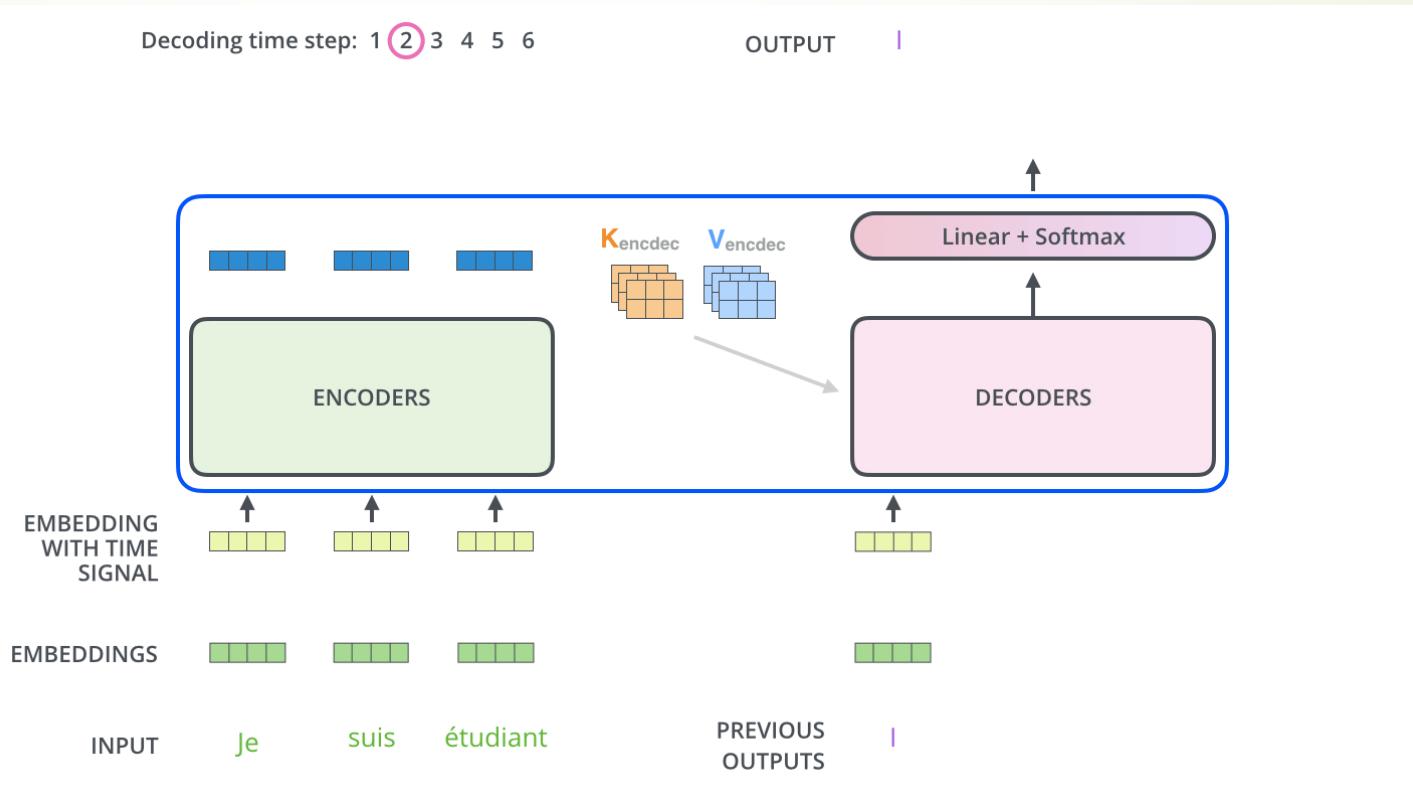
# Encoder-Decoder



# Illustration of Encoder-Decoder

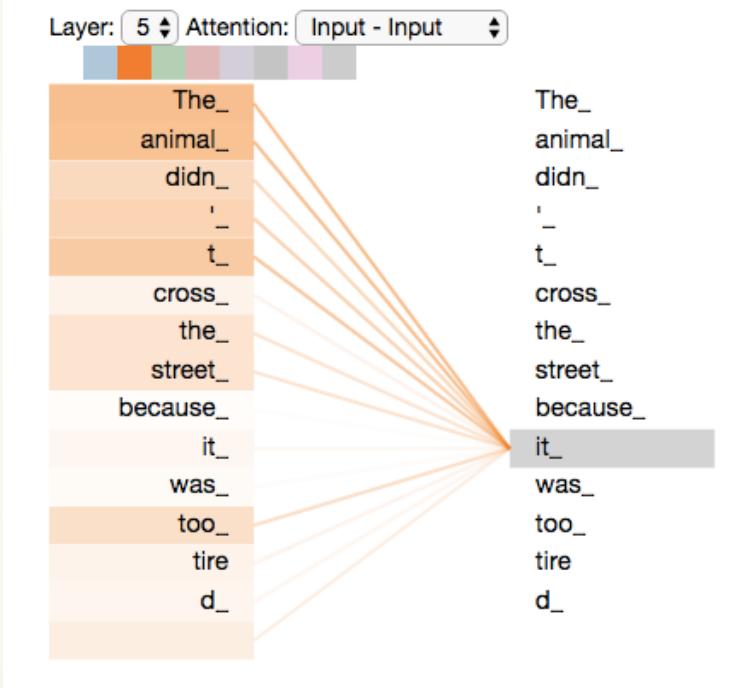


# Illustration of Encoder-Decoder

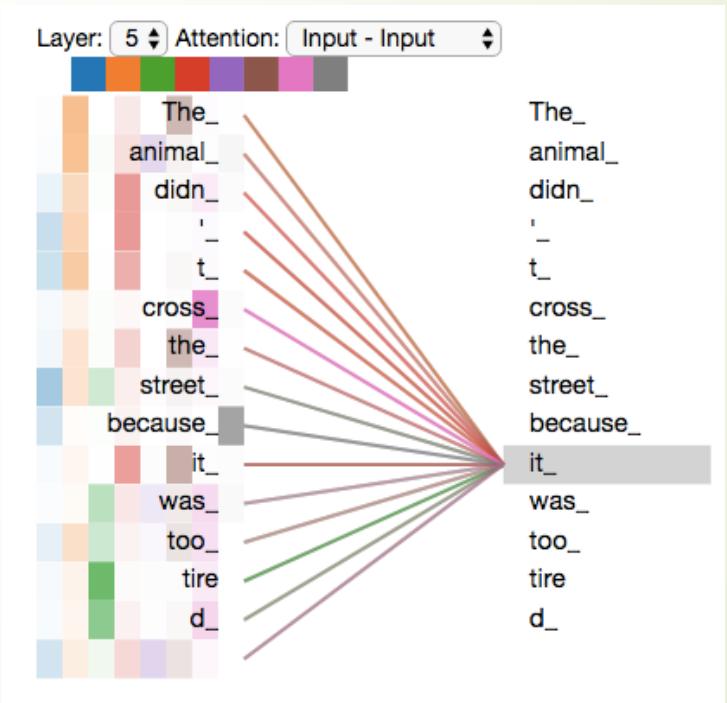


# Attention Visualization

Head 2 (yellow) only

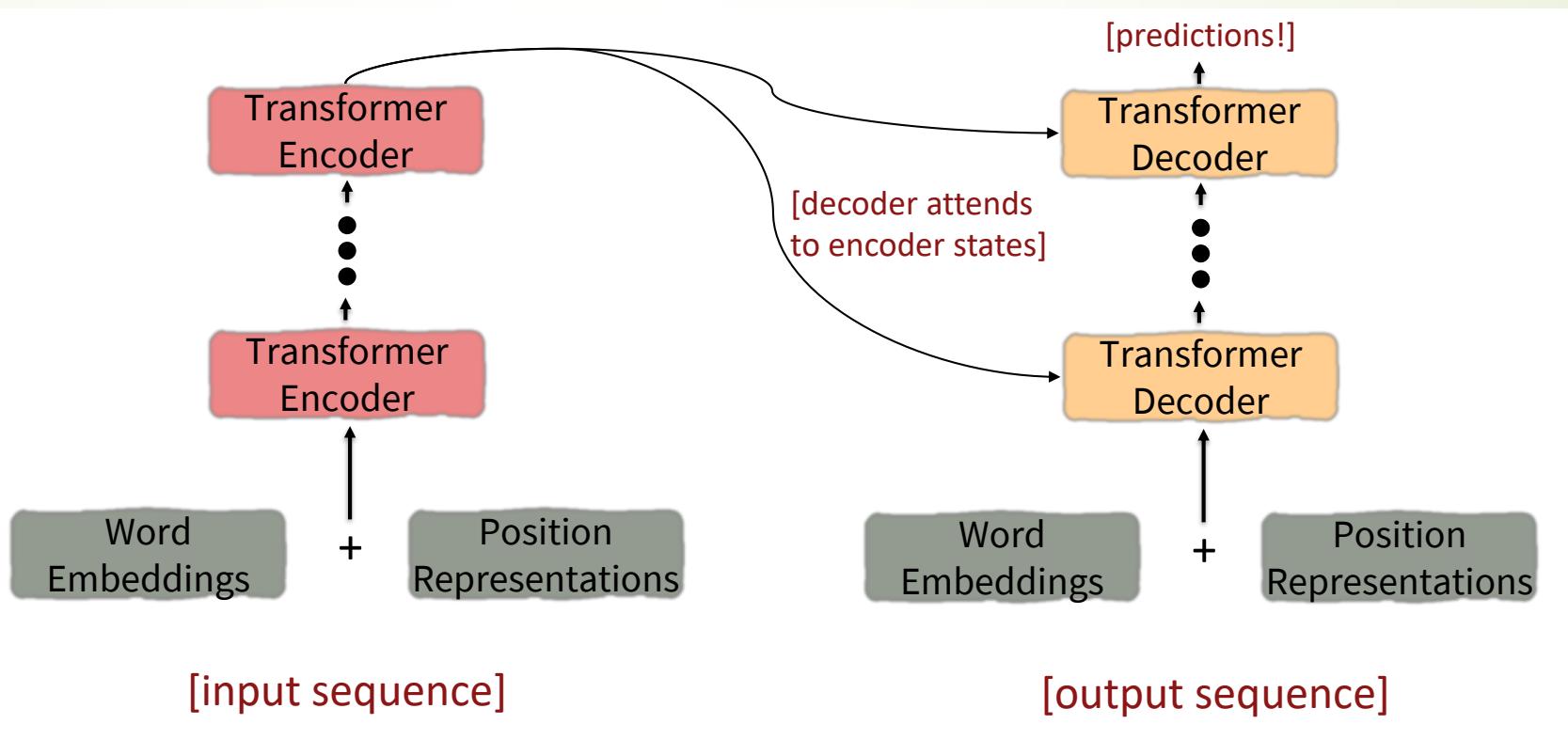


8 heads mixture



# The Transformer Encoder-Decoder [Vaswani et al. 2017]

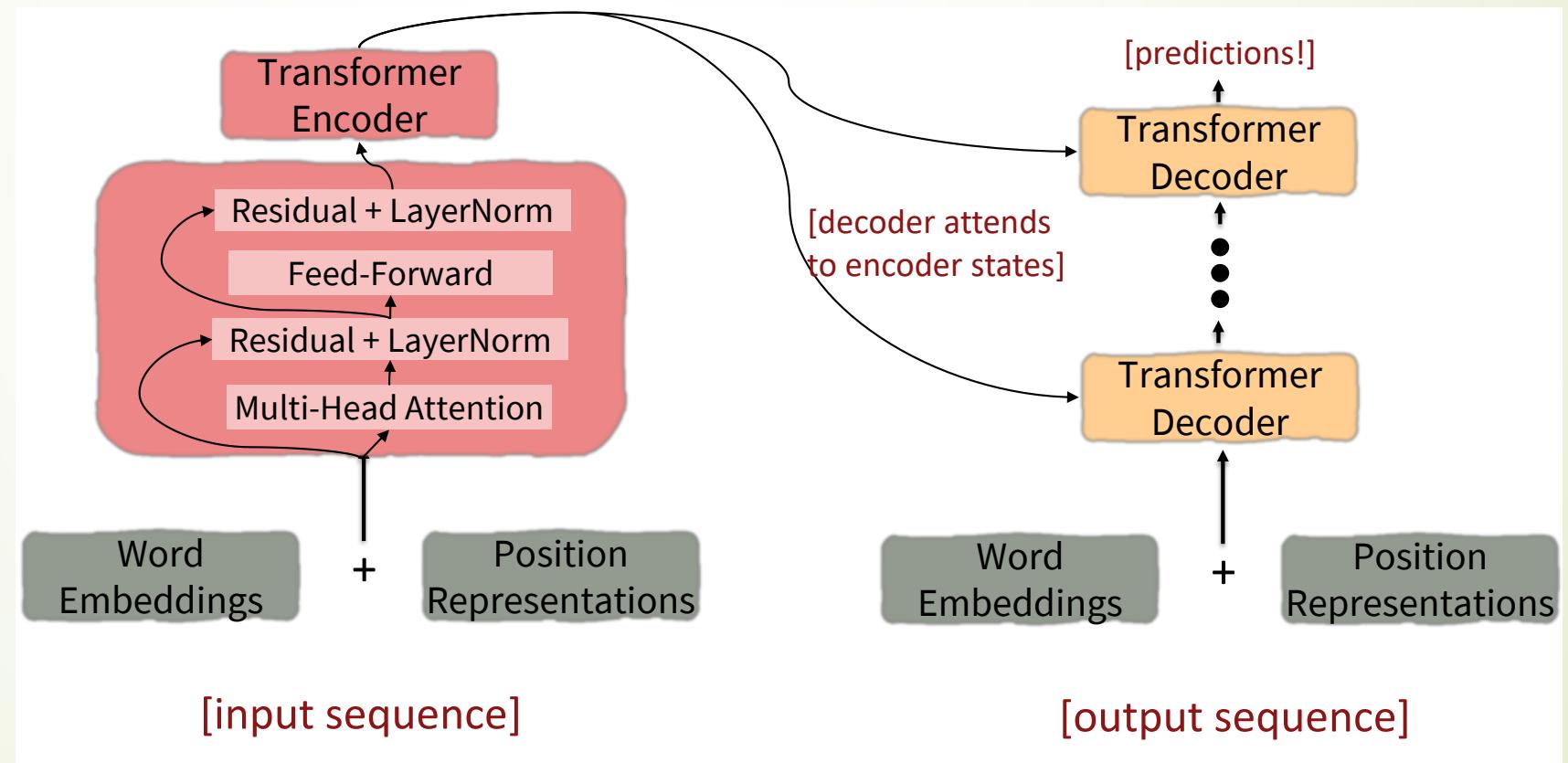
- ▶ Looking back at the whole model



# The Transformer Encoder-Decoder

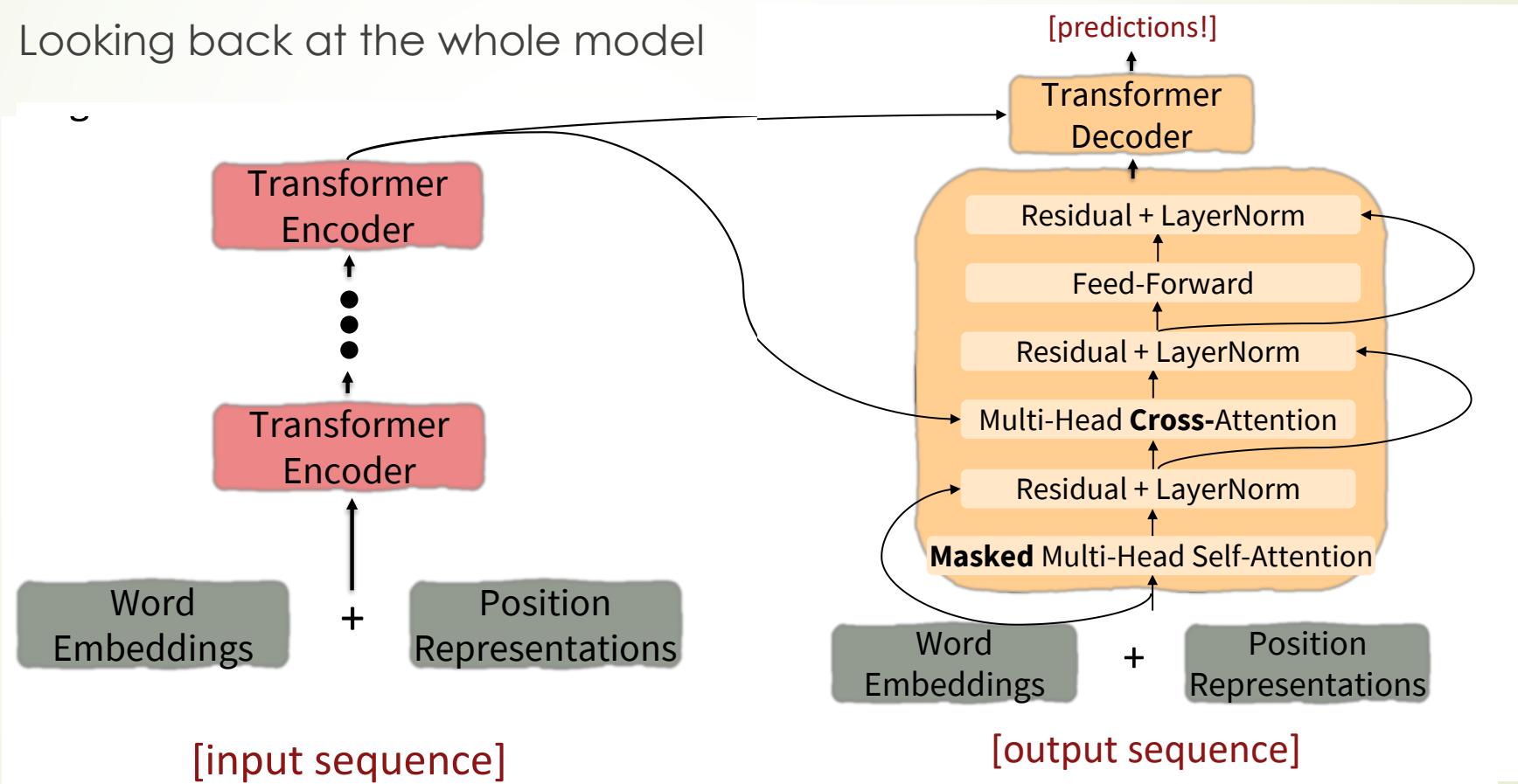
[Vaswani et al. 2017]

- ▶ Looking back at the whole model



# The Transformer Encoder-Decoder [Vaswani et al. 2017]

- ▶ Looking back at the whole model



# Empirical advantages of Transformer vs. LSTM

- ▶ 1. Self-attention == no locality bias
  - ▶ Long-distance context has “equal opportunity”
- ▶ 2. Single multiplication per layer == efficiency on TPU

**Transformer**

X_0_0	X_0_1	X_0_2	X_0_3
X_1_0	X_1_1	X_1_2	X_1_3

$\times$  

**LSTM**

X_0_0	X_0_1	X_0_2	X_0_3
X_1_0	X_1_1	X_1_2	X_1_3

$\times$  

# Major disadvantage of Transformer

- **Quadratic compute in self-attention (today):**

- Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
- For recurrent models, it only grew linearly!

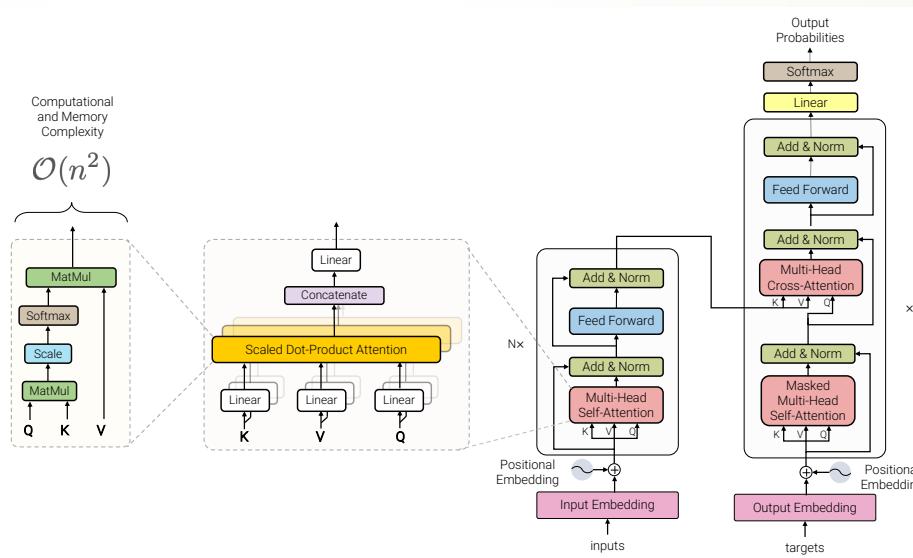


Figure 1: Architecture of the standard Transformer (Vaswani et al., 2017)

# 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  $O(n^2 d)$ , where  $n$  is the sequence length, and  $d$  is the dimensionality.
- ▶ Think of  $d$  as around **1,000** (though for large language models it's much larger!).
  - So, for a single (shortish) sentence,  $n \leq 30$ ;  $n^2 \leq 900$ .
  - In practice, we set a bound like  $n = 512$ .
  - **But what if we'd like  $n \geq 50,000$ ?** For example, to work on long documents?

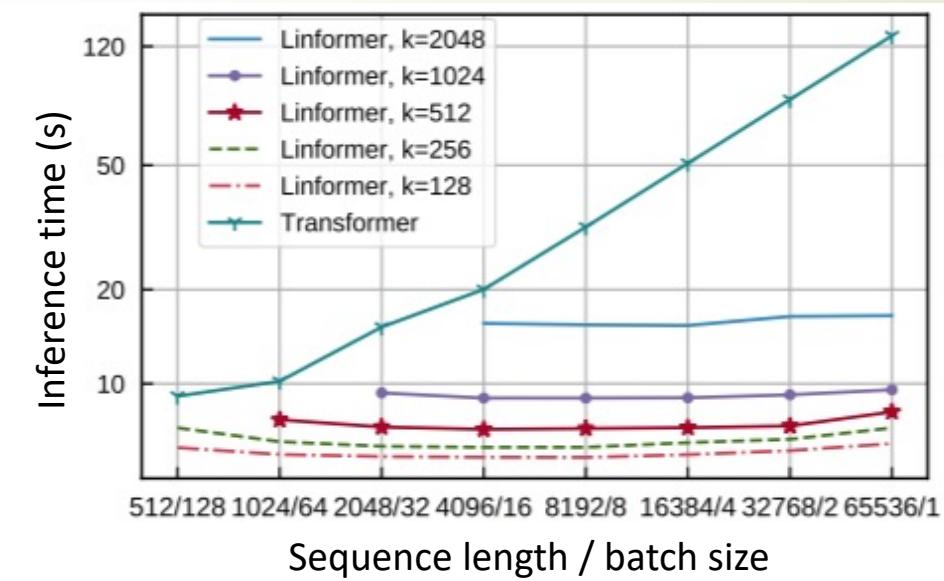
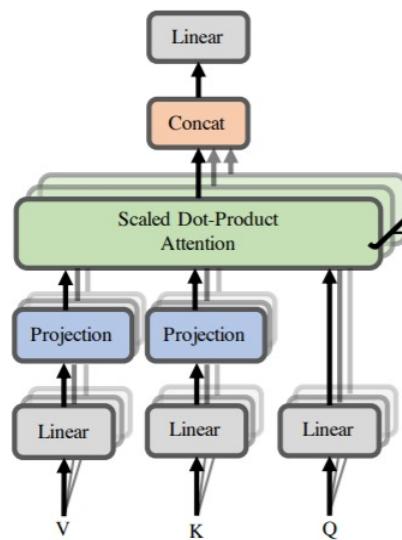
$$\begin{matrix} XQ \\ K^\top X^\top \end{matrix} = \begin{matrix} XQK^\top X^\top \\ \in \mathbb{R}^{n \times n} \end{matrix}$$

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

# Improving quadratic self-attention cost

- ▶ Considerable recent work has gone into the question, Can we build models like Transformers without paying the all-pairs self-attention cost?
- ▶ For example, **Linformer** [Wang et al., 2020, **Linformer: Self-Attention with Linear Complexity**, arXiv:2006.04768]

Key idea: map the sequence length dimension to a lower-dimensional space for values, keys



# Efficient Transformers

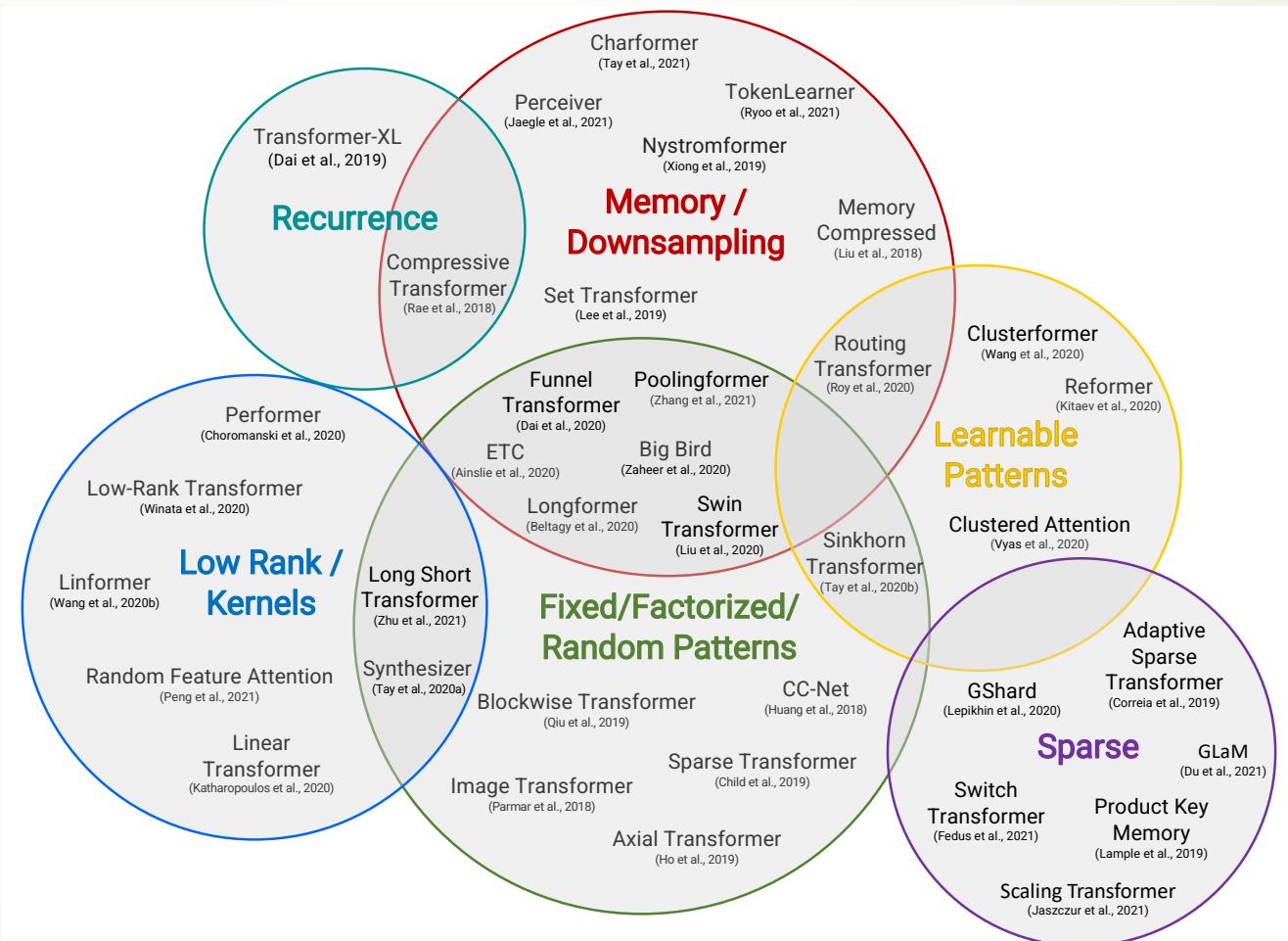


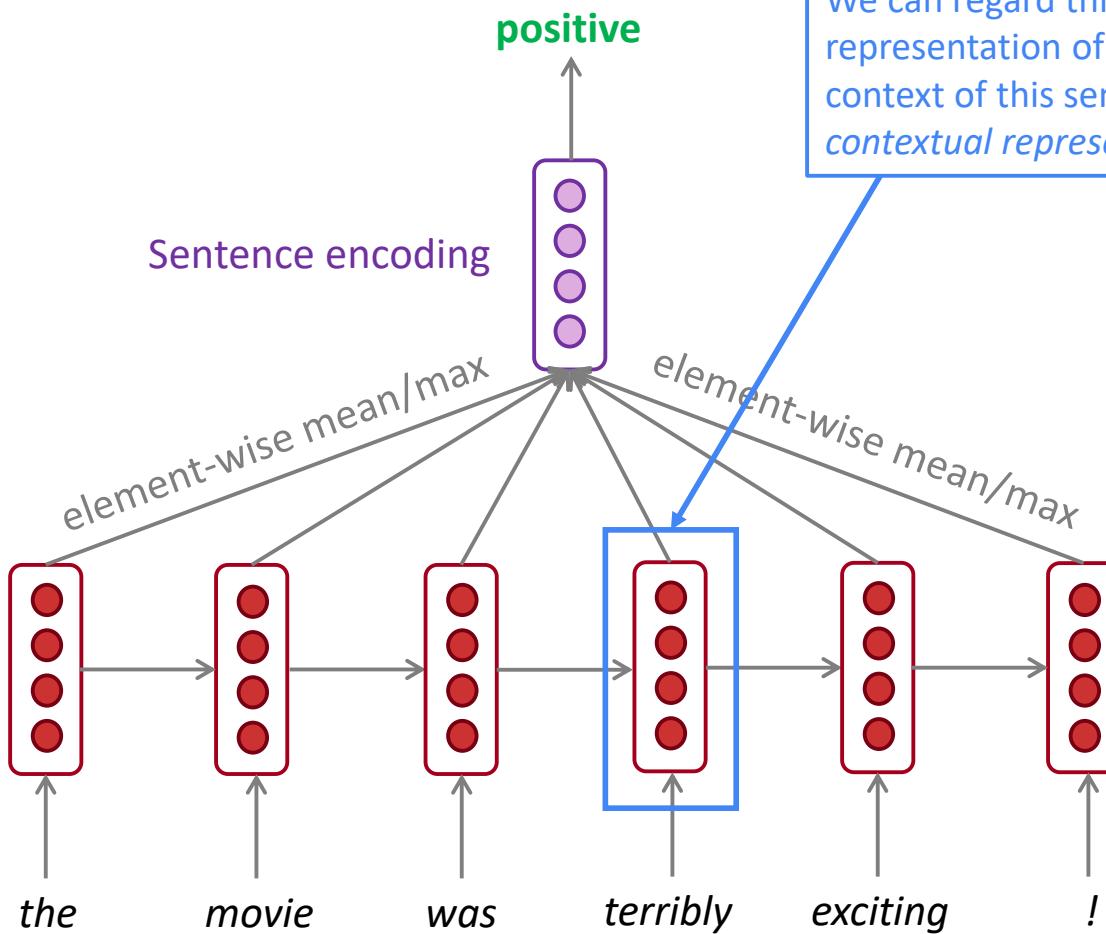
Figure 2: Taxonomy of Efficient Transformer Architectures.

The background features a minimalist design with a vertical brown bar on the left. A large, solid red arrow points from the bottom left towards the center. Overlaid on the background are several thin, light gray curved lines that intersect and curve across the frame.

# Bi-Direction

# Motivation of Bidirection

Task: Sentiment Classification



We can regard this hidden state as a representation of the word “*terribly*” in the context of this sentence. We call this a *contextual representation*.

These contextual representations only contain information about the *left context* (e.g. “*the movie was*”).

What about *right context*?

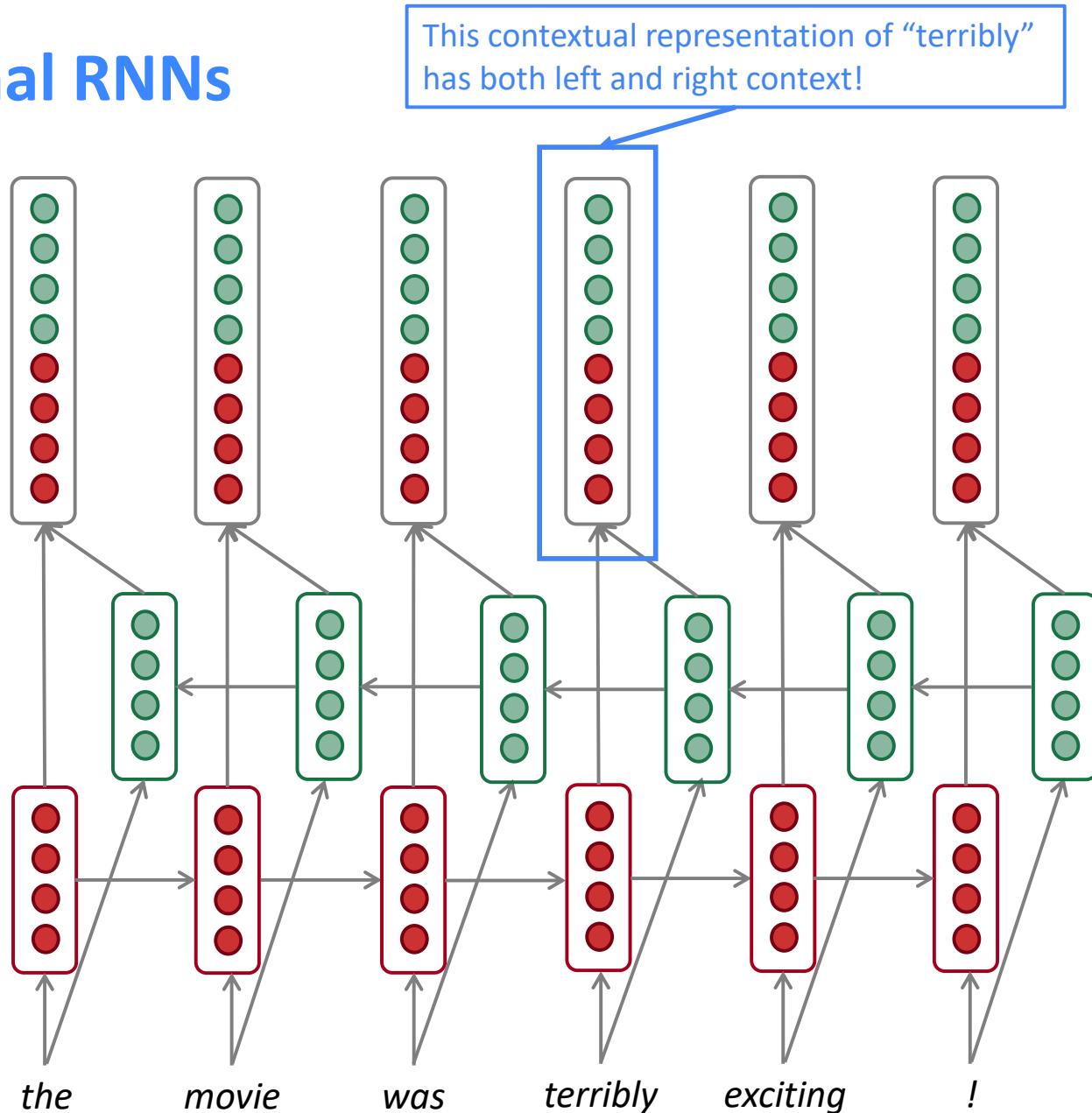
In this example, “*exciting*” is in the right context and this modifies the meaning of “*terribly*” (from negative to positive)

# Bidirectional RNNs

Concatenated  
hidden states

Backward RNN

Forward RNN



# Bidirectional RNN: simplified diagram

On timestep  $t$ :

This is a general notation to mean “compute one forward step of the RNN” – it could be a vanilla, LSTM or GRU computation.

Forward RNN  $\vec{h}^{(t)} = \text{RNN}_{\text{FW}}(\vec{h}^{(t-1)}, \mathbf{x}^{(t)})$

Backward RNN  $\overleftarrow{h}^{(t)} = \text{RNN}_{\text{BW}}(\overleftarrow{h}^{(t+1)}, \mathbf{x}^{(t)})$

Concatenated hidden states

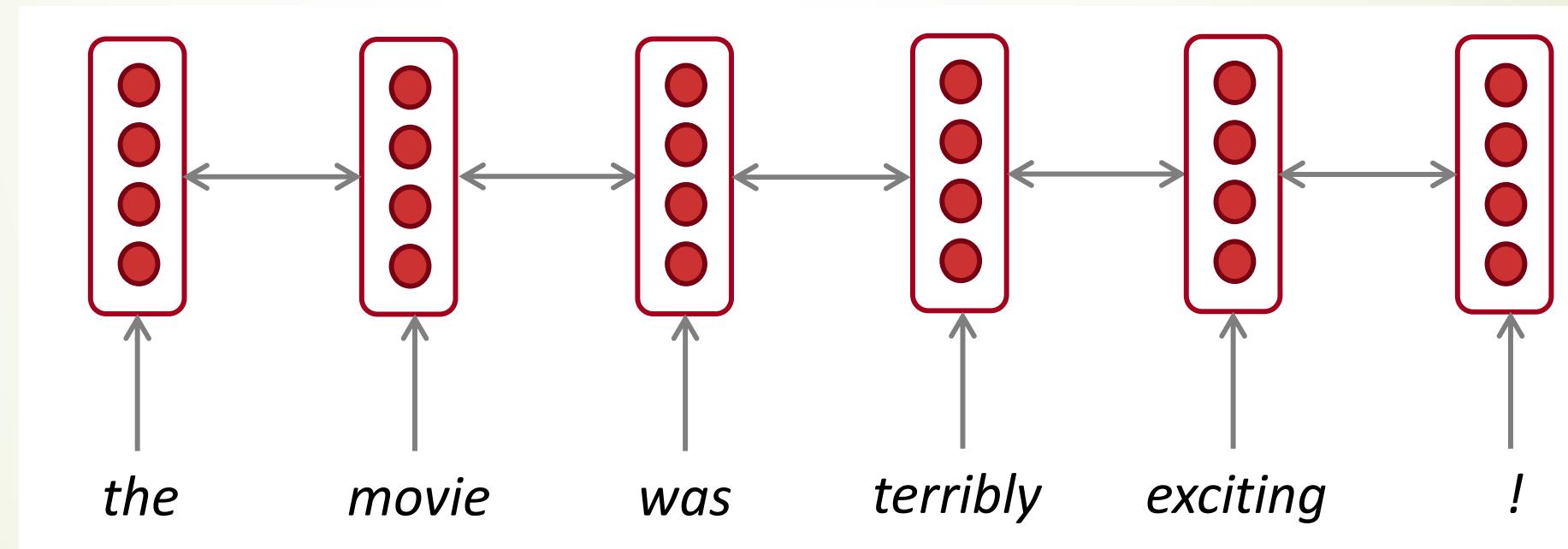
$$\mathbf{h}^{(t)} = [\vec{h}^{(t)}; \overleftarrow{h}^{(t)}]$$

Generally, these two RNNs have separate weights

We regard this as “the hidden state” of a bidirectional RNN. This is what we pass on to the next parts of the network.

# Bidirectional RNN: simplified diagram

- ▶ The two-way arrows indicate bidirectionality and the depicted hidden states are assumed to be the concatenated forwards+backwards states.

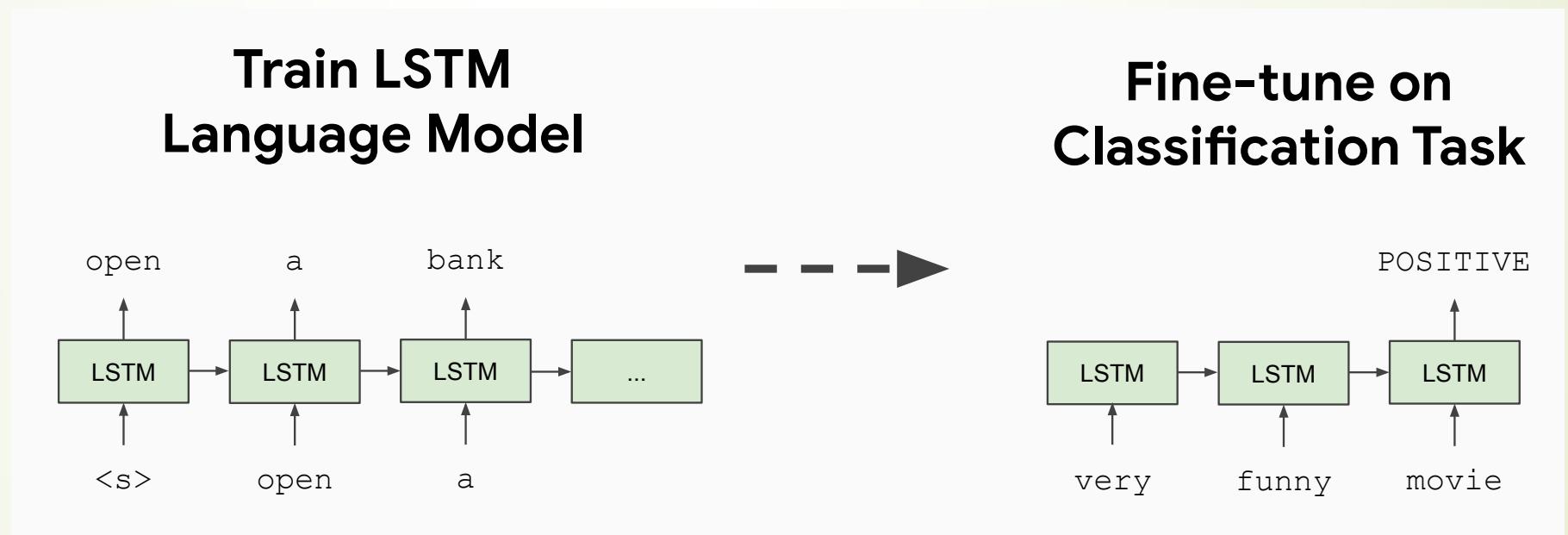


# Bidirectional RNNs

- ▶ Note: bidirectional RNNs are only applicable if you have access to the **entire input sequence**.
  - ▶ For example, **Encoder** of Transformers
  - ▶ They are **not** applicable to Language Modeling, because in LM you *only* have left context available, e.g. **Decoder** of Transformers
- ▶ If you do have entire input sequence (e.g. any kind of encoding), bidirectionality is powerful (you should use it by default).
- ▶ For example, **BERT** (**Bidirectional** Encoder Representations from Transformers) is a powerful pretrained contextual representation system built on bidirectionality.

# Uni-Direction LSTM

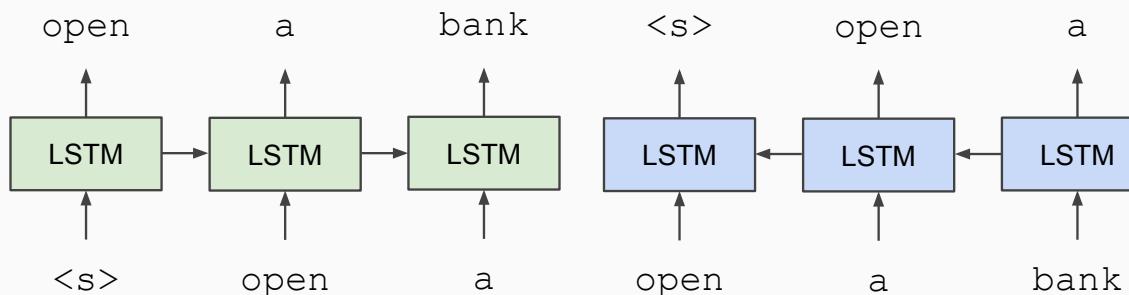
- Semi-Supervised Sequence Learning, Google, 2015



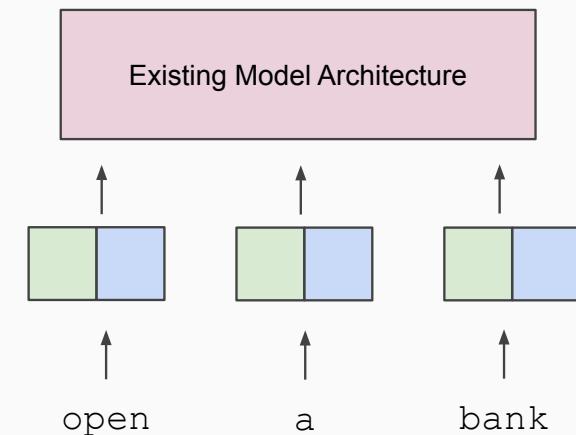
# Bi-Direction LSTM: ELMo -- Embeddings from Language Models

- ▶ Peters et al. (2018) Deep Contextual Word Embeddings, NAACL 2018.  
<https://arxiv.org/abs/1802.05365>
- ▶ Learn a deep Bi-NLM and use all its layers in prediction

**Train Separate Left-to-Right and Right-to-Left LMs**

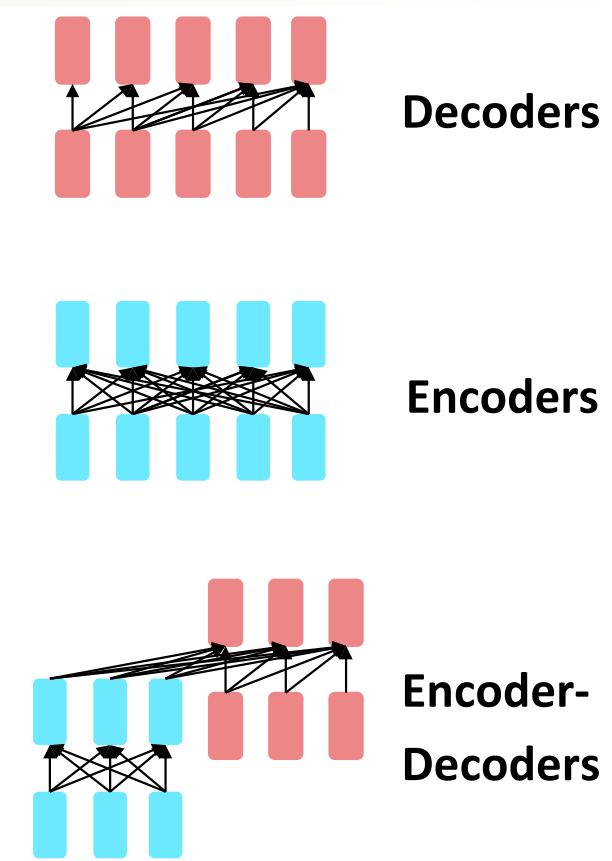


**Apply as “Pre-trained Embeddings”**



# Pretraining for three types of architectures in Transformers

The transformer architecture influences the type of pretraining:

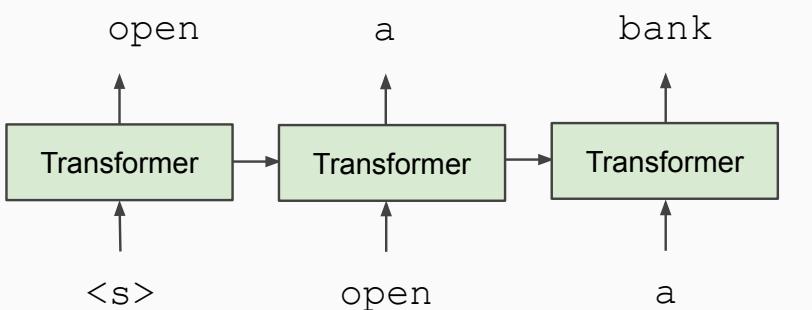


- ▶ Decoders:
  - ▶ **Unidirectional Language models!**
  - ▶ Nice to generate from; can't condition on future words
- ▶ Encoders:
  - ▶ Gets **bidirectional** context (e.g. **classification**) – can condition on future!
  - ▶ Wait, **how do we pretrain them?**
- ▶ Encoder-Decoders:
  - ▶ Good parts of decoders and encoders?
  - ▶ **What's the best way to pretrain them?**

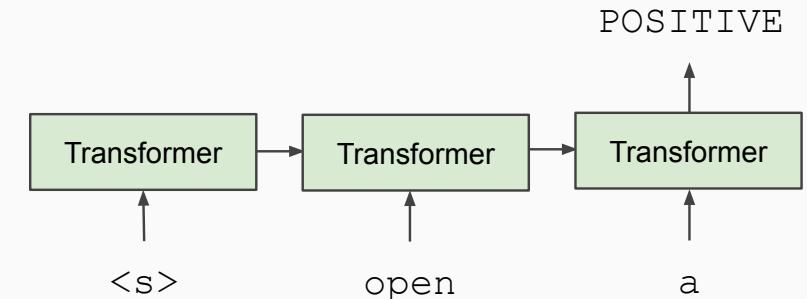
# GPT (Generative Pre-Training): uni-directional transformer

- *Improving Language Understanding by Generative Pre-Training*, OpenAI, 2018

## Train Deep (12-layer) Transformer LM



## Fine-tune on Classification Task



# Pretraining decoders

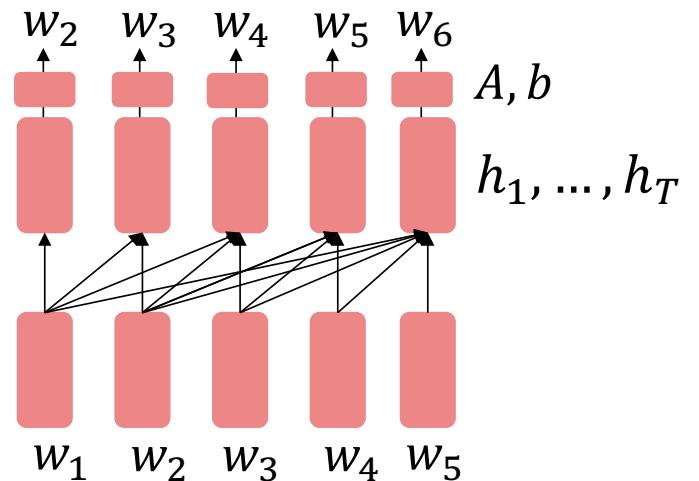
It's natural to pretrain decoders as language models and then use them as generators, finetuning their  $p_\theta(w_t|w_{1:t-1})$ !

This is helpful in tasks **where the output is a sequence** with a vocabulary like that at pretraining time!

- Dialogue (context=dialogue history)
- Summarization (context=document)

$$\begin{aligned} h_1, \dots, h_T &= \text{Decoder}(w_1, \dots, w_T) \\ w_t &\sim Ah_{t-1} + b \end{aligned}$$

Where  $A, b$  were pretrained in the language model!



[Note how the linear layer has been pretrained.]

# Fine-tuning decoders

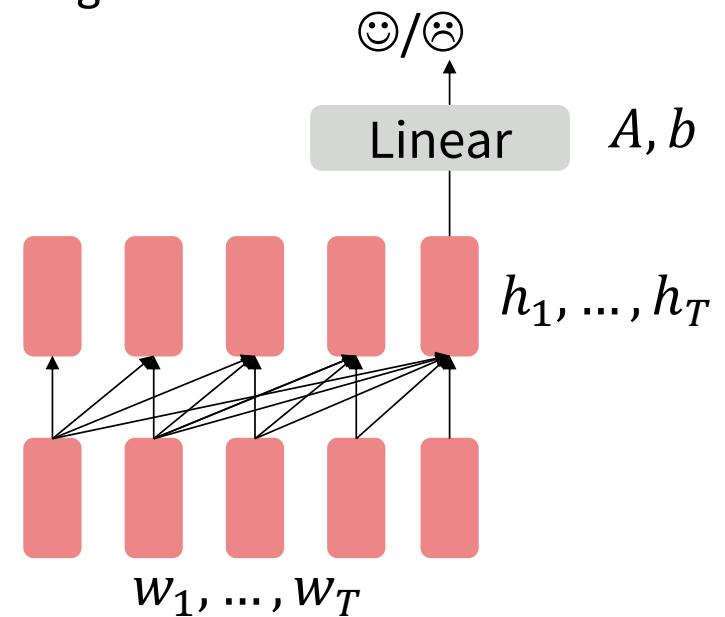
When using language model pretrained decoders, we can ignore that they were trained to model  $p(w_t|w_{1:t-1})$ .

We can finetune them by training a classifier on the last word's hidden state.

$$\begin{aligned} h_1, \dots, h_T &= \text{Decoder}(w_1, \dots, w_T) \\ y &\sim Ah_T + b \end{aligned}$$

Where  $A$  and  $b$  are randomly initialized and specified by the downstream task.

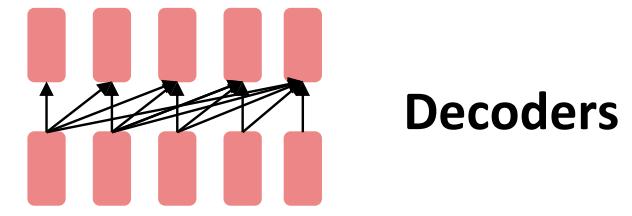
Gradients backpropagate through the whole network.



[Note how the linear layer hasn't been pretrained and must be learned from scratch.]

# GPT (Generative Pre-Trained Transformer): uni-directional transformer-decoder

- ▶ 2018's GPT was a big success in pretraining a decoder!
- Transformer decoder with 12 layers.
- 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers.
- Byte-pair encoding with 40,000 merges
- Trained on BooksCorpus: over 7000 unique books.
  - Contains long spans of contiguous text, for learning long-distance dependencies.



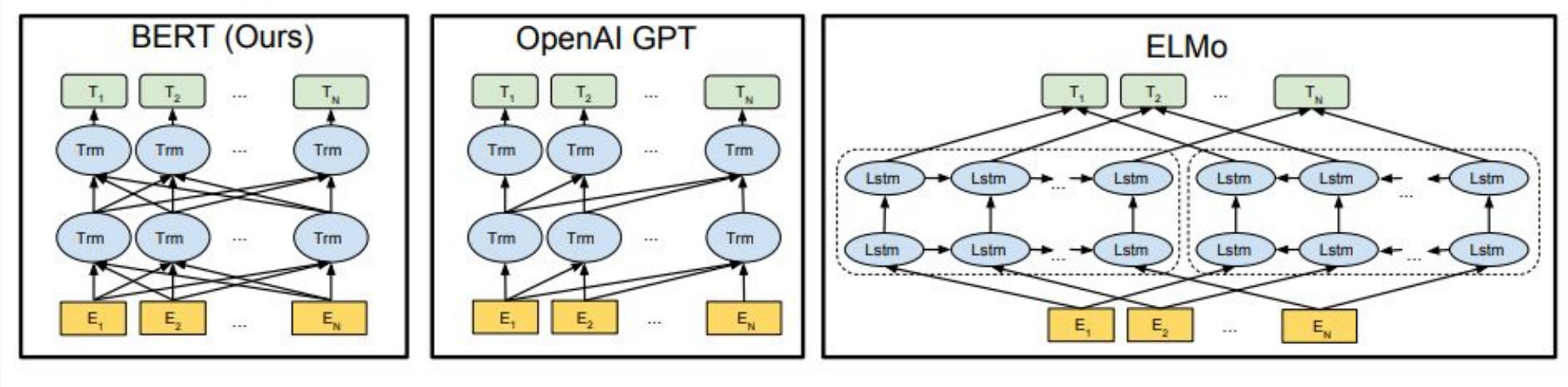
- Language models! What we've seen so far.
- Nice to generate from; can't condition on future words



# GPT-3, In-context learning, and very large models

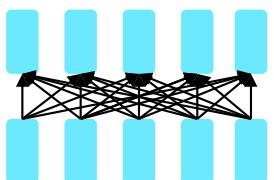
- ▶ So far, we've interacted with pretrained models in two ways:
  - ▶ Sample from the distributions they define (maybe providing a prompt)
  - ▶ Fine-tune them on a task we care about and take their predictions.
- ▶ Very large language models seem to perform some kind of learning **without gradient steps** simply from examples you provide within their contexts.
- ▶ GPT-3 (2020 by OpenAI) is the canonical example of this. The largest T5 model had 11 billion parameters.
- ▶ **GPT-3 has 175 billion parameters.**
  - ▶ 350GB of storage (16-bit precision)
  - ▶ Context window size 2048 tokens

# How about bi-directional transformers? – Yes, BERT!



# BERT: Devlin, Chang, Lee, Toutanova (2018)

- ▶ BERT (**Bidirectional Encoder Representations from Transformers**):
- ▶ Pre-training of Deep Bidirectional Transformers for Language Understanding, which is then fine-tuned for a task
- ▶ Want: truly bidirectional information flow without leakage in a deep model



Encoders

- Gets bidirectional context – can condition on future!
- Wait, how do we pretrain them?

# Masked Language Model

- ▶ **Problem:** How the words see each other in bi-directions?
- ▶ **Solution:** Mask out  $k\%$  of the input words, and then predict the masked words
  - ▶ We always use  $k = 15\%$



the man went to the [MASK] to buy a [MASK] of milk

store  
↑  
gallon  
↑

- ▶ Too little masking: Too expensive to train
- ▶ Too much masking: Not enough context

# Masked LM

- ▶ **Problem:** Masked token never seen at fine-tuning
- ▶ **Solution:** 15% of the words to predict, but don't replace with [MASK] 100% of the time. Instead:
  - ▶ 80% of the time, replace with [MASK]
    - ▶ went to the store → went to the [MASK]
  - ▶ 10% of the time, replace random word
    - ▶ went to the store → went to the running
  - ▶ 10% of the time, keep same
    - ▶ went to the store → went to the store

# Next Sentence Prediction

- To learn *relationships* between sentences, predict whether Sentence B is actual sentence that proceeds Sentence A, or a random sentence

**Sentence A** = The man went to the store.

**Sentence B** = He bought a gallon of milk.

**Label** = IsNextSentence

**Sentence A** = The man went to the store.

**Sentence B** = Penguins are flightless.

**Label** = NotNextSentence

# BERT sentence pair encoding

- ▶ Token embeddings are word pieces (30k)
- ▶ Learned segmented embedding represents each sentence
- ▶ Positional embedding is as for other Transformer architectures

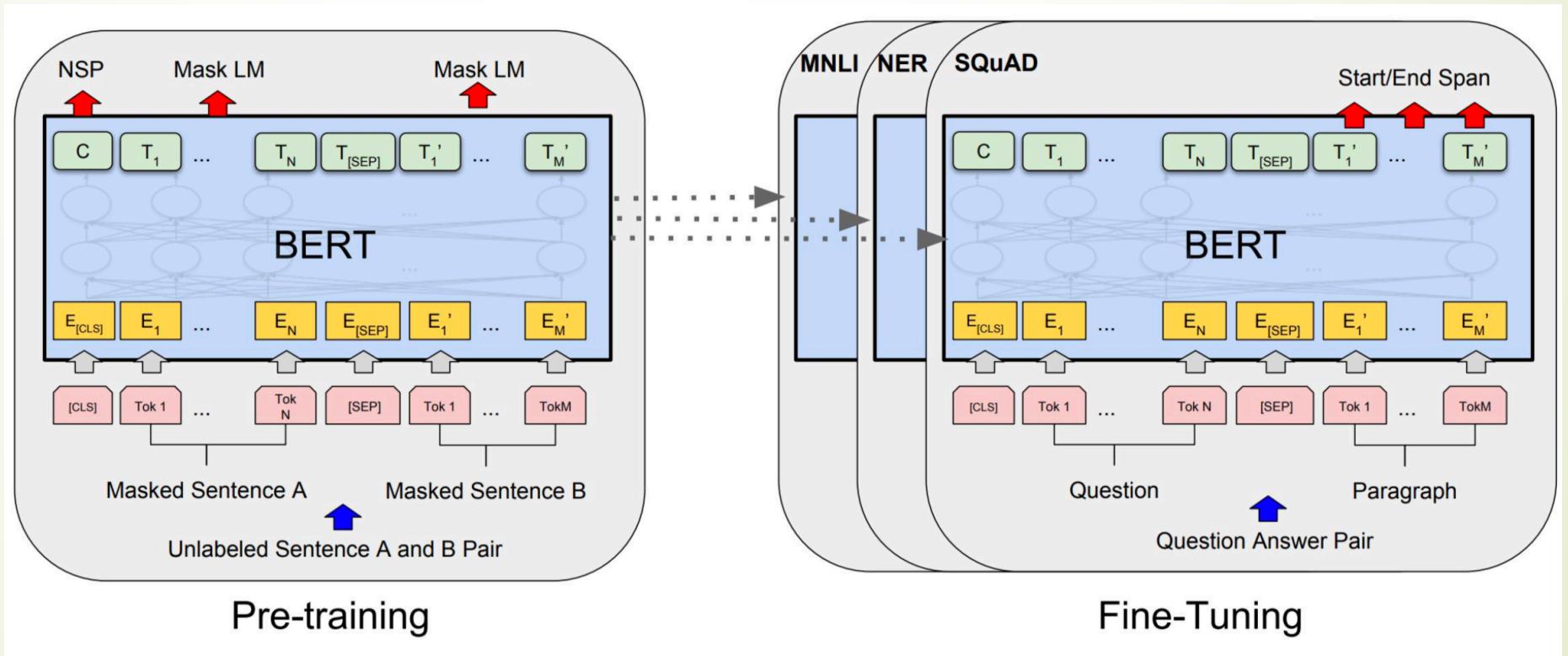
Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	# #ing	[SEP]
Token Embeddings	$E_{[CLS]}$	$E_{my}$	$E_{dog}$	$E_{is}$	$E_{cute}$	$E_{[SEP]}$	$E_{he}$	$E_{likes}$	$E_{play}$	$E_{##ing}$	$E_{[SEP]}$
Segment Embeddings	$E_A$	$E_A$	$E_A$	$E_A$	$E_A$	$E_A$	$E_B$	$E_B$	$E_B$	$E_B$	$E_B$
Position Embeddings	$E_0$	$E_1$	$E_2$	$E_3$	$E_4$	$E_5$	$E_6$	$E_7$	$E_8$	$E_9$	$E_{10}$

# Training

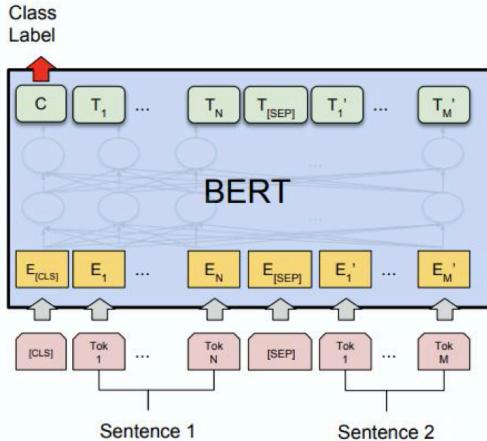
- ▶ 2 models released:
  - ▶ BERT-Base: 12-layer, 768-hidden, 12-head, 110 million params.
  - ▶ BERT-Large: 24-layer, 1024-hidden, 16-head, 340 million params.
- ▶ Training Data:
  - ▶ BookCorpus (800M words)
  - ▶ English Wikipedia (2.5B words)
- ▶ Batch Size: 131,072 words
  - ▶ (1024 sequences \* 128 length or 256 sequences \* 512 length)
- ▶ Training Time: 1M steps (~40 epochs)
- ▶ Optimizer: AdamW, 1e-4 learning rate, linear decay
- ▶ Trained on 4x4 or 8x8 TPU slice for 4 days
- ▶ Pretraining is expensive and impractical on a single GPU; Finetuning is practical and common on a single GPU

# BERT model fine tuning

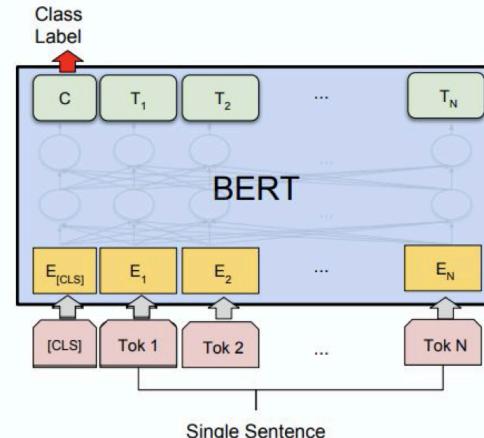
- ▶ Simply learn a classifier built on the top layer for each task that you fine tune for



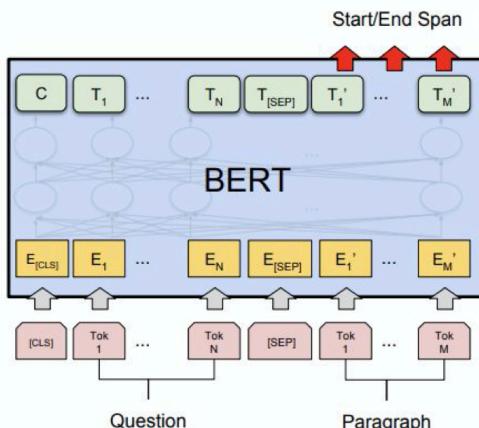
# BERT model fine tuning



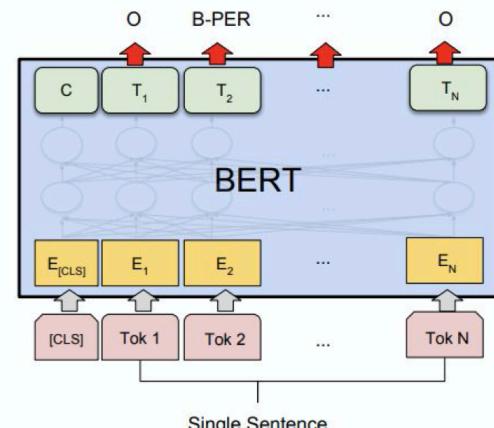
(a) Sentence Pair Classification Tasks:  
MNLI, QQP, QNLI, STS-B, MRPC,  
RTE, SWAG



(b) Single Sentence Classification Tasks:  
SST-2, CoLA

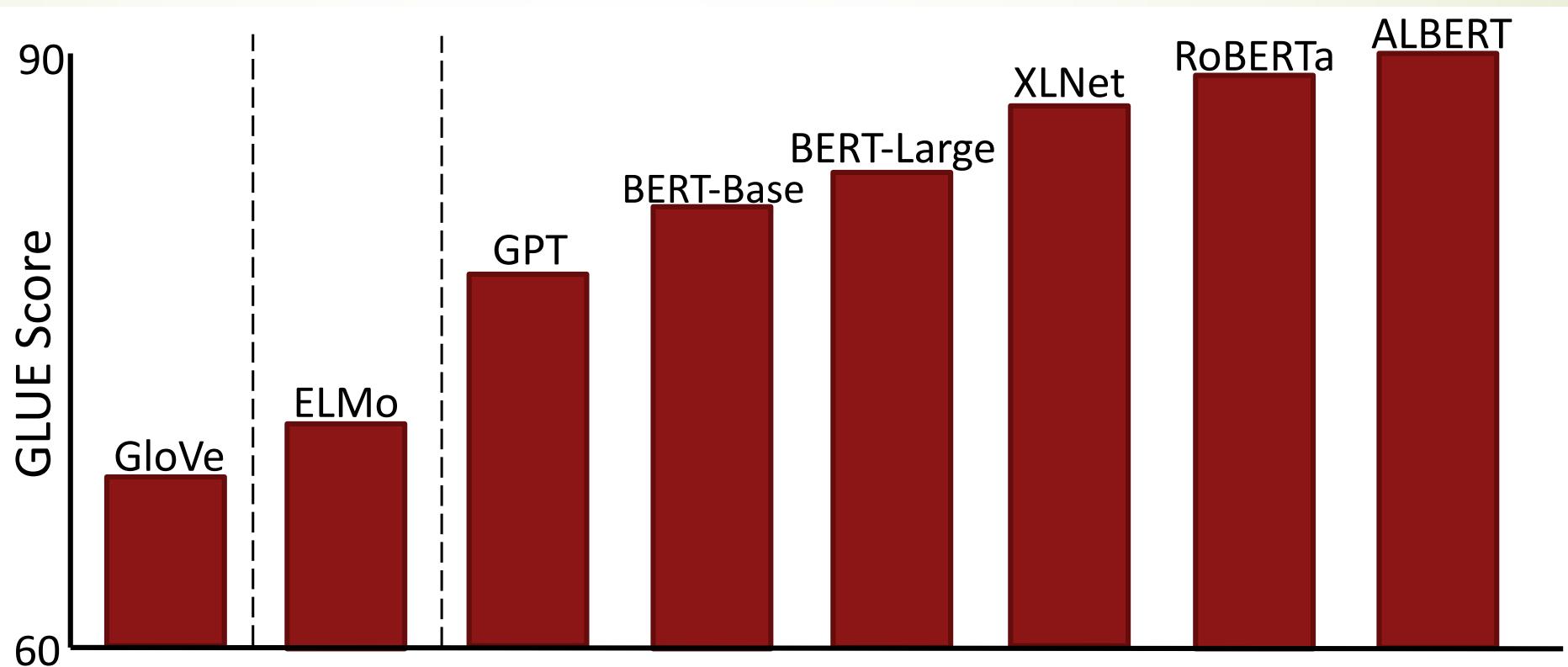


(c) Question Answering Tasks:  
SQuAD v1.1



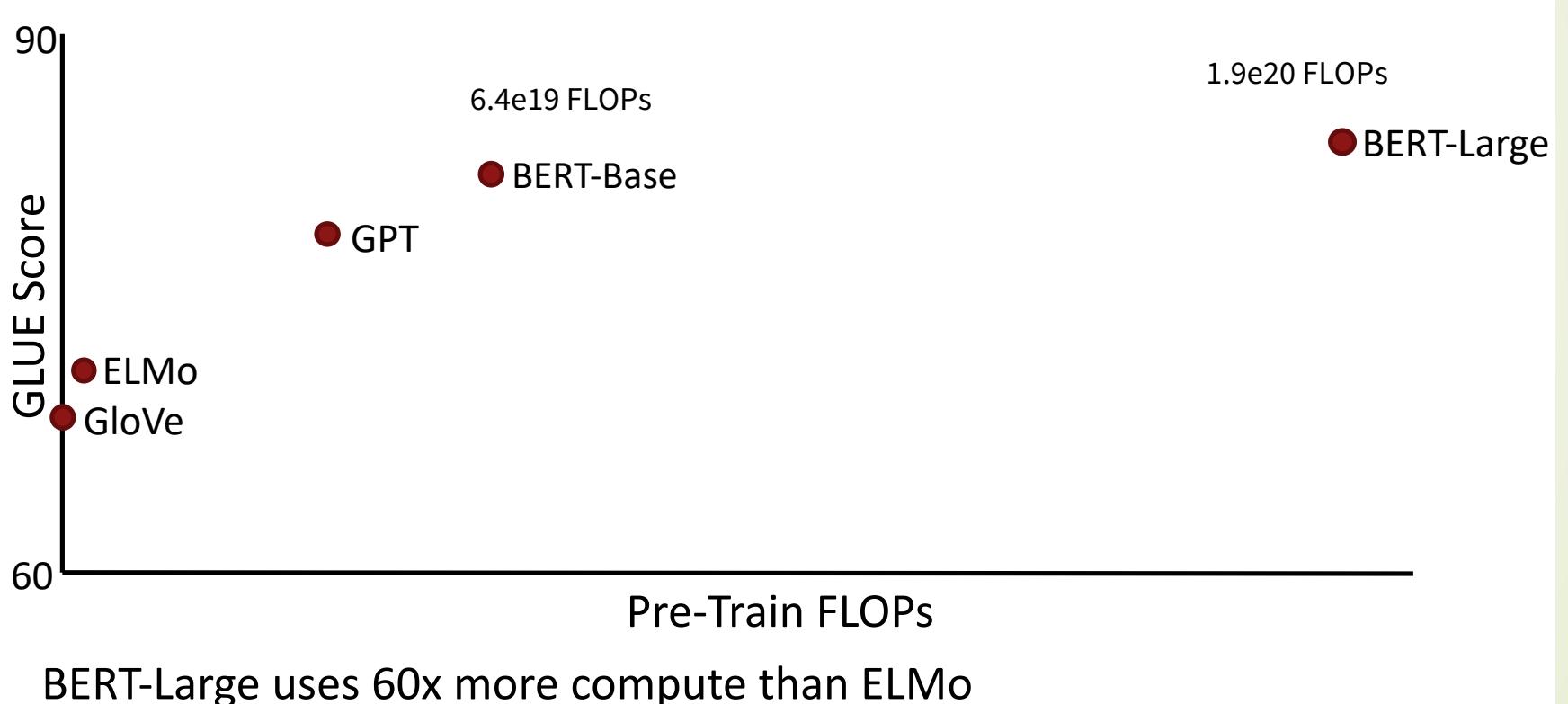
(d) Single Sentence Tagging Tasks:  
CoNLL-2003 NER

# Rapid Progress for Pre-training (GLUE Benchmark)

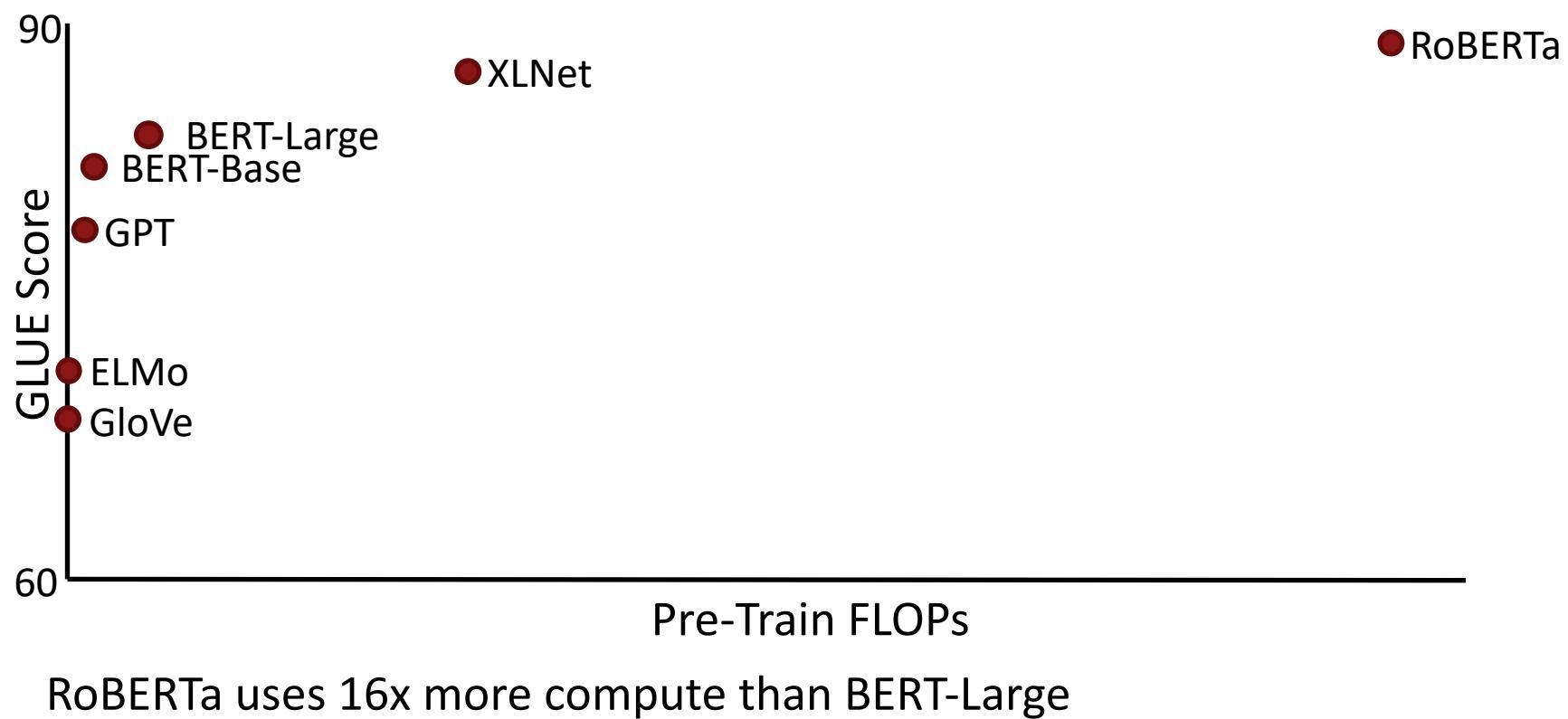


Over 3x reduction in error in 2 years, “superhuman” performance

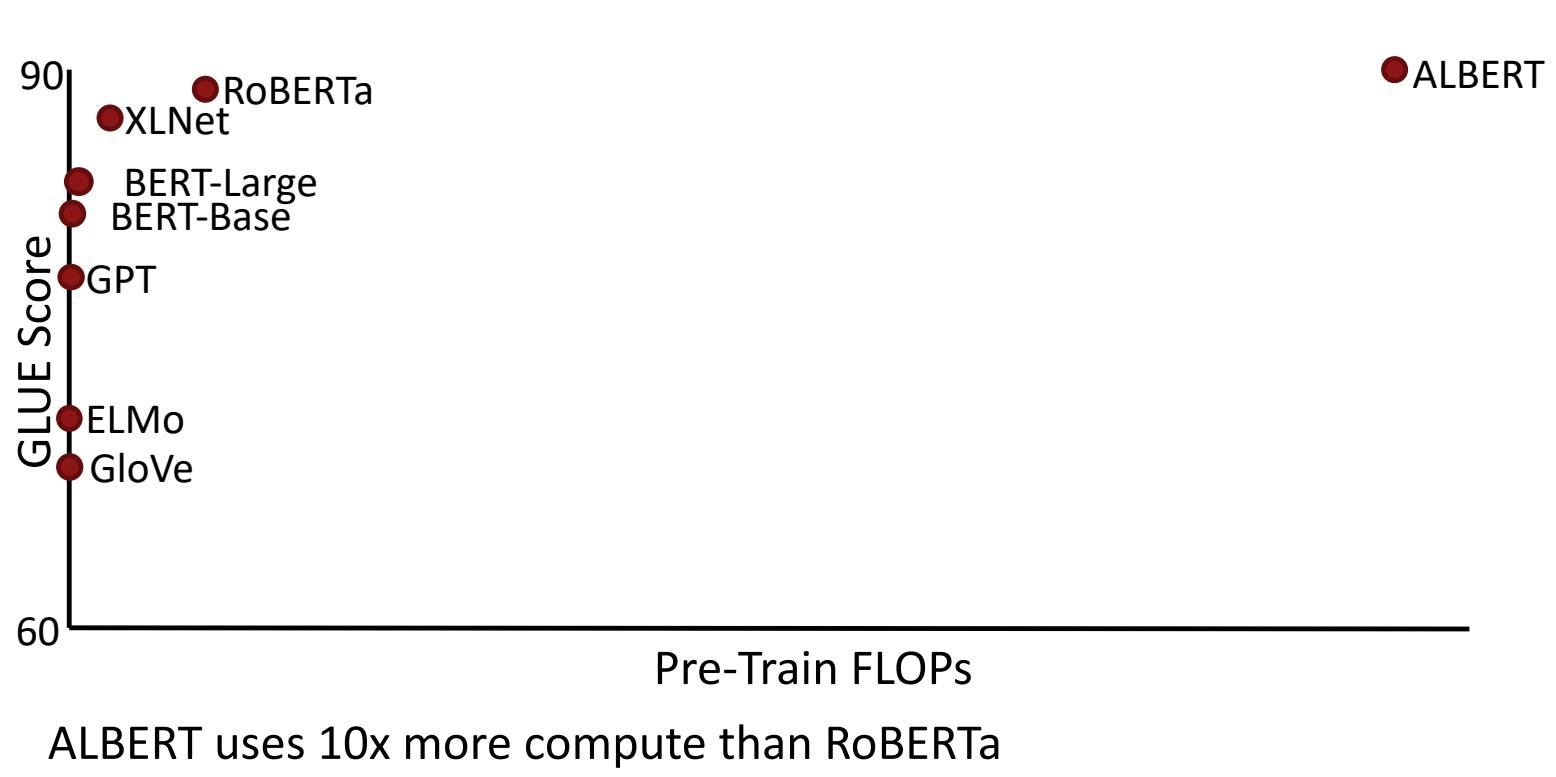
# But let's change the x-axis to computational cost...



But let's change the x-axis to computational cost...

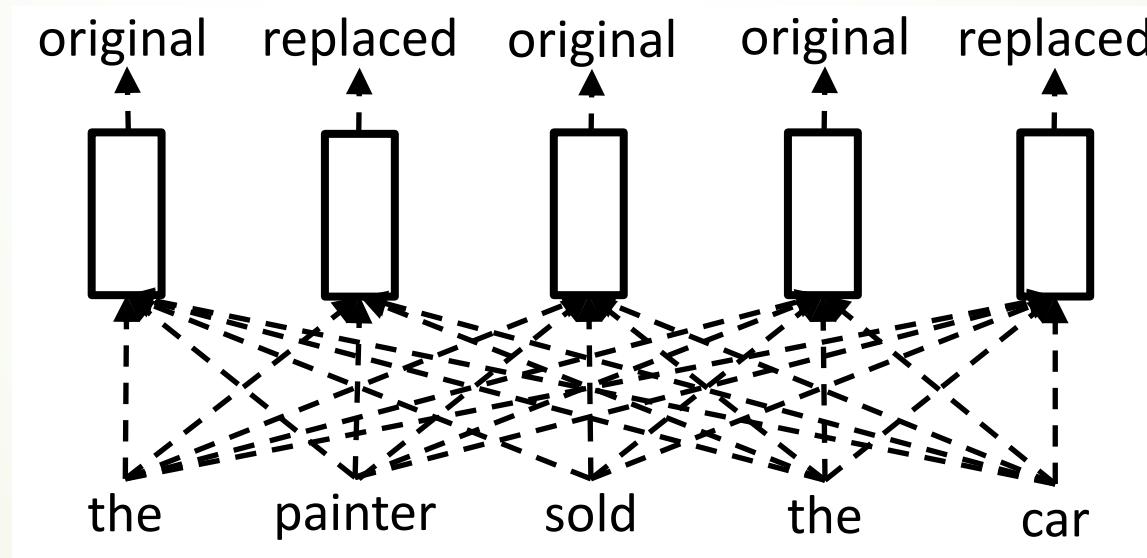


# More compute, more better?

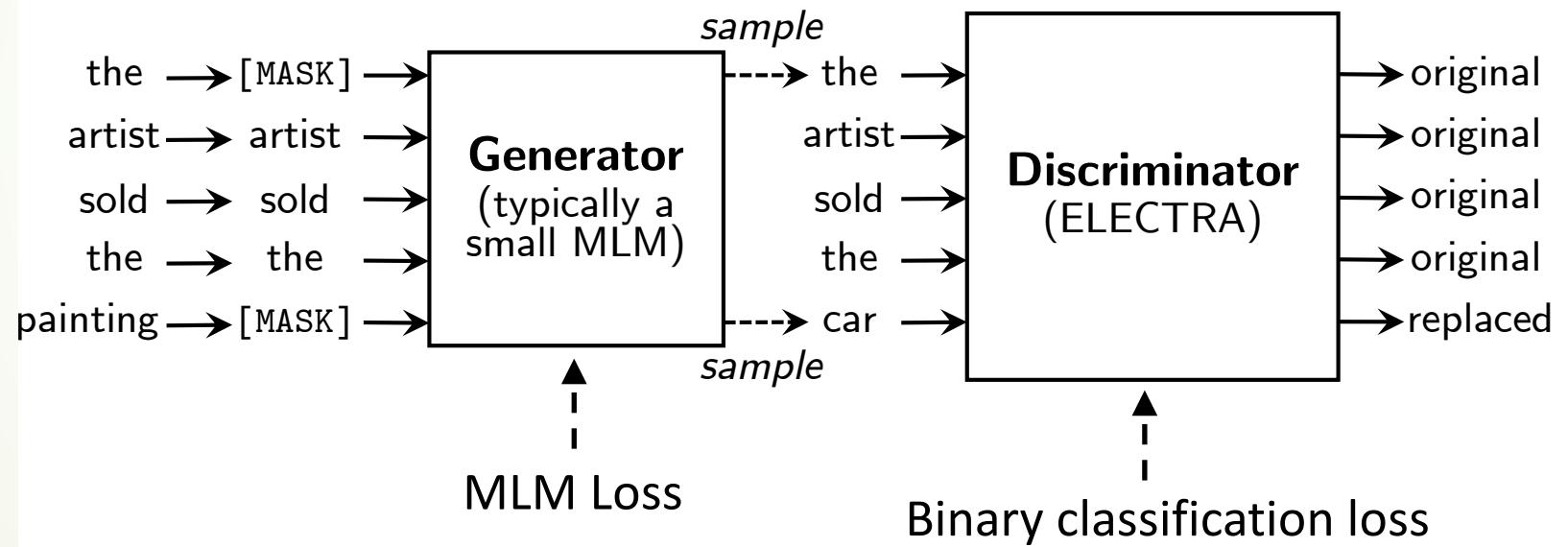


# ELECTRA: “Efficiently Learning an Encoder to Classify Token Replacements Accurately”

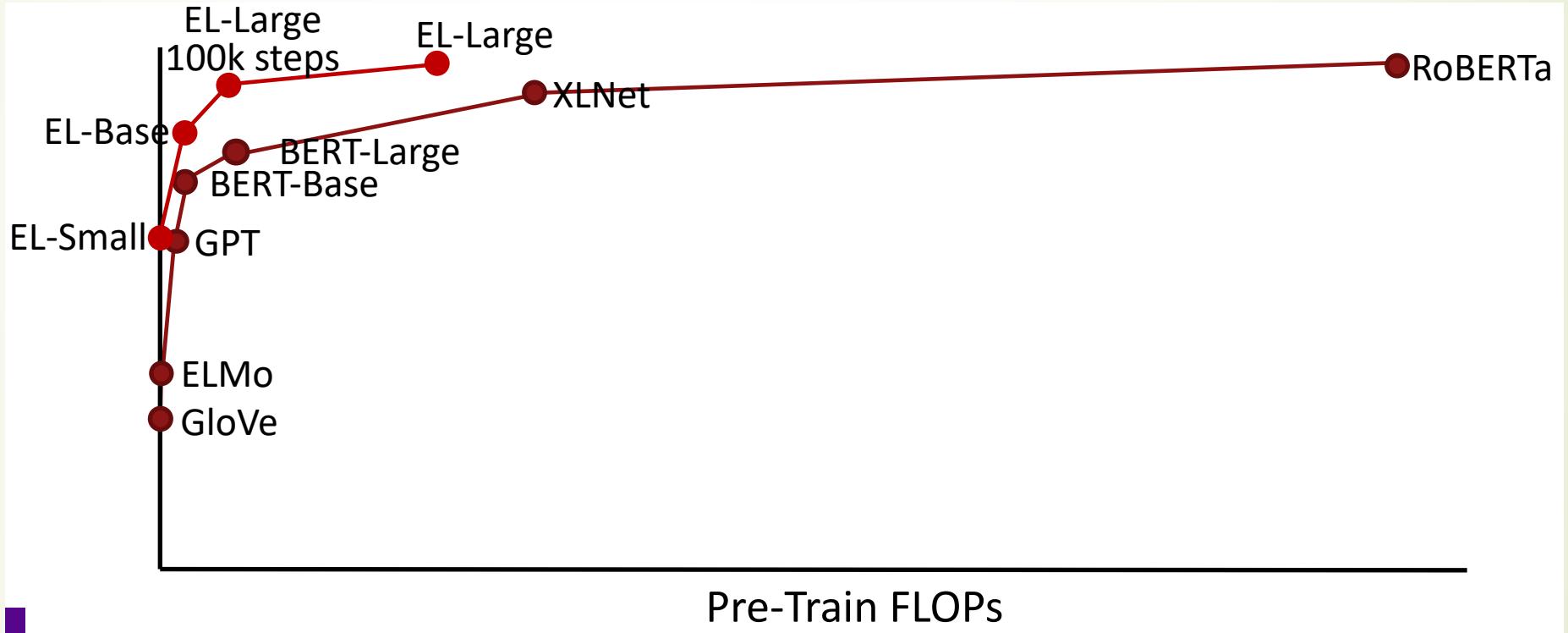
- ▶ Clark, Luong, Le, and Manning, ICLR 2020.  
<https://openreview.net/pdf?id=r1xMH1BtvB>
- ▶ Bidirectional model but learn from all tokens



# Generating Replacements

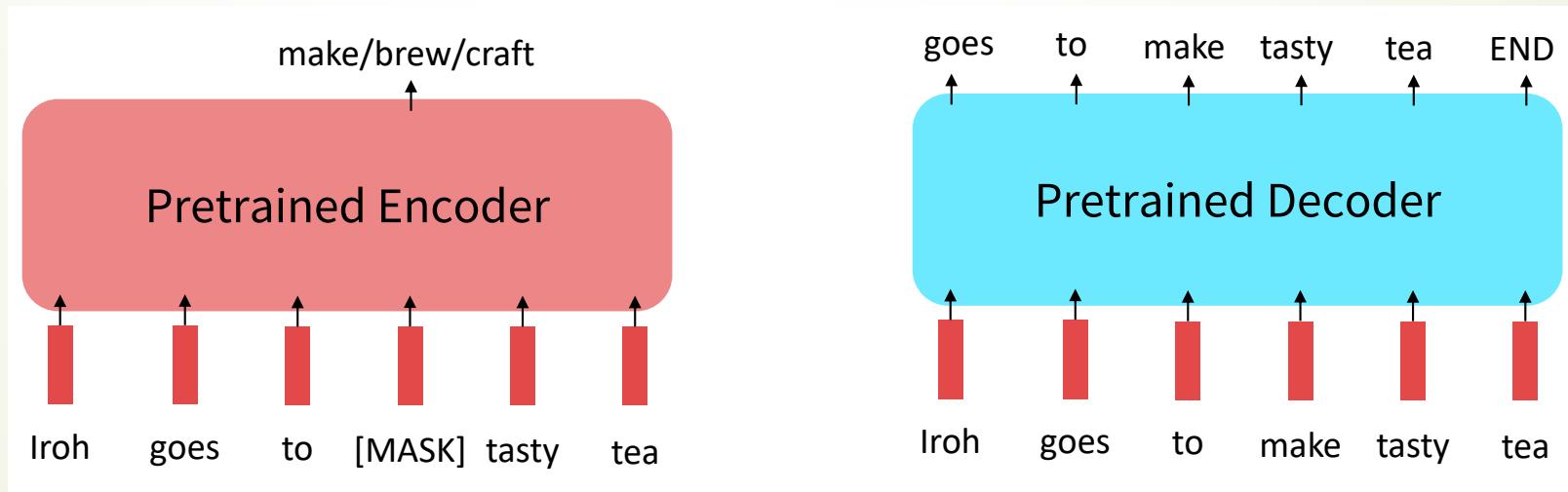


# Results: GLUE Score vs Compute



# Limitations of Pretrained Encoders

- ▶ Those results looked great! Why not used pretrained encoders for *everything*?
- ▶ If your task involves generating sequences, consider using a pretrained **decoder**; BERT and other pretrained encoders don't naturally lead to nice autoregressive (1-word-at-a-time) generation methods.

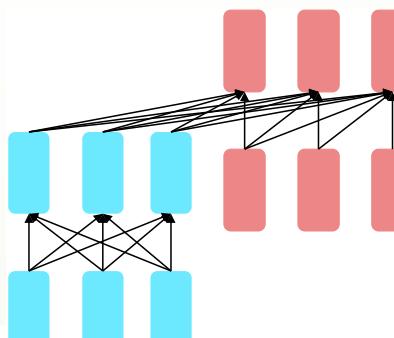


# Pretraining encoders-decoders: T5

- ▶ Pretraining encoder-decoders: what pretraining objective to use?
- ▶ What Raffel et al., 2018 found to work best was **span corruption: T5**.
- ▶ Replace different-length spans from the input with unique placeholders; decode out the spans that were removed!
- ▶ A fascinating property of T5: it can be finetuned to answer a wide range of questions, retrieving knowledge from its parameters (11 billion).

Original text

Thank you ~~for inviting~~ me to your party ~~last~~ week.



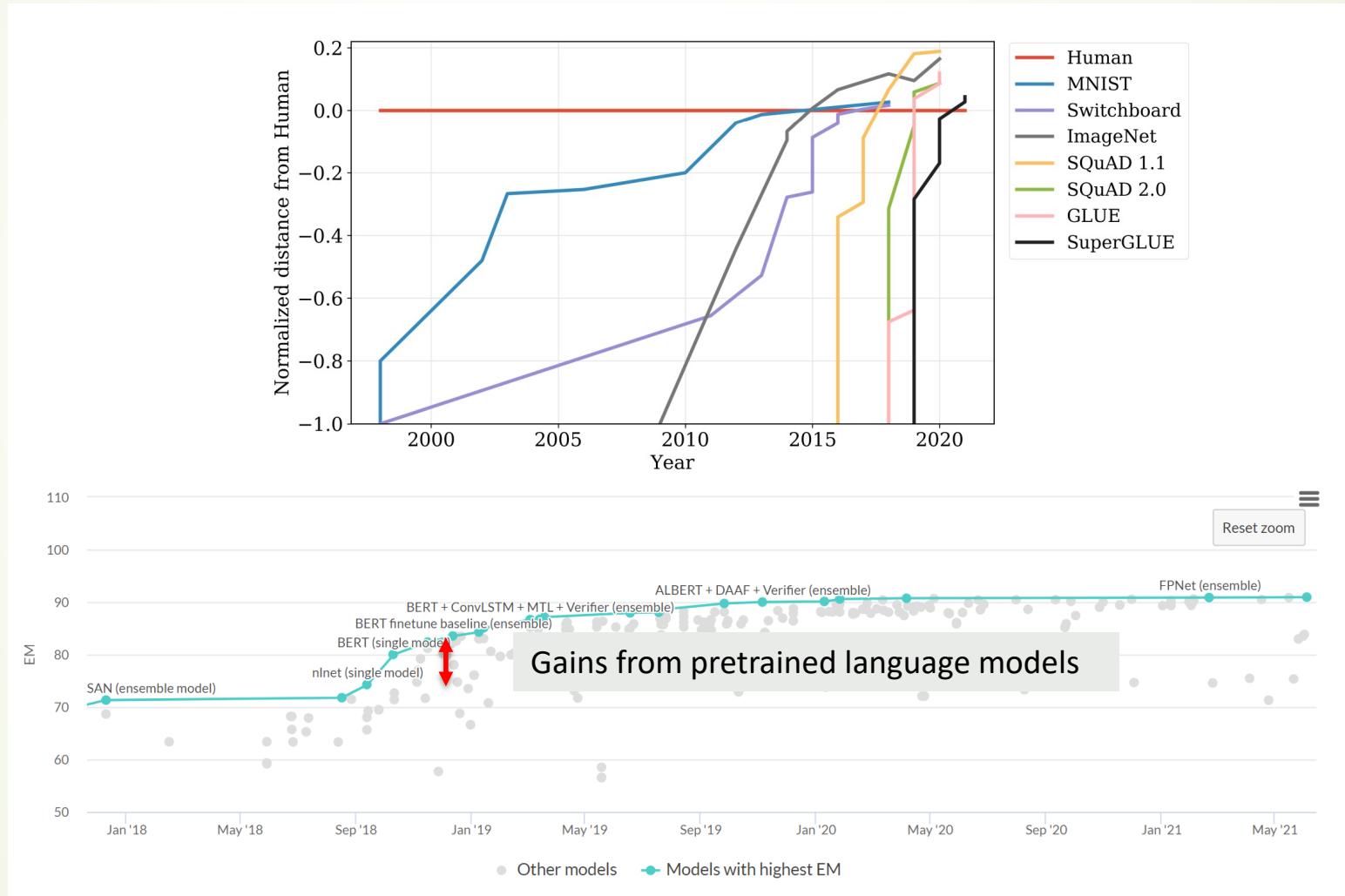
Targets

<X> for inviting <Y> last <Z>

Inputs

Thank you <X> me to your party <Y> week.

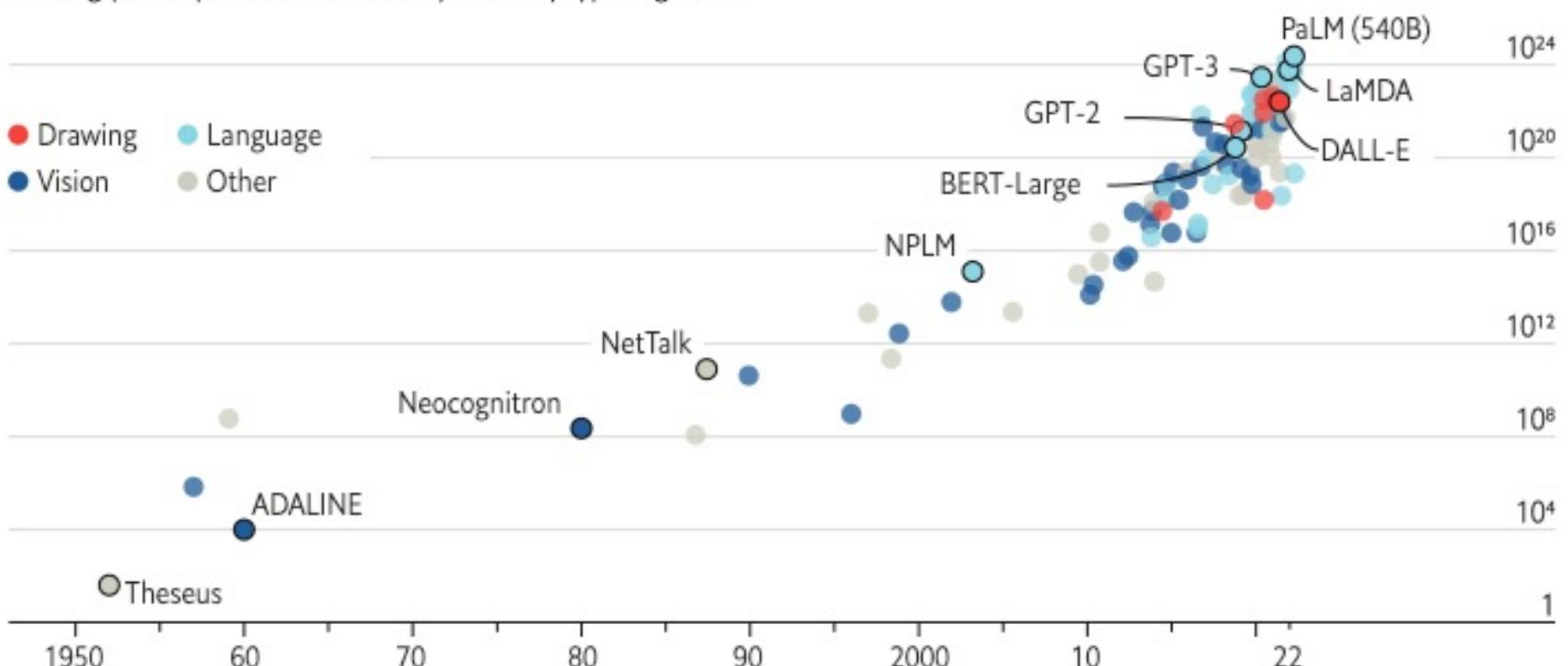
# Pretraining revolution



## The blessings of scale

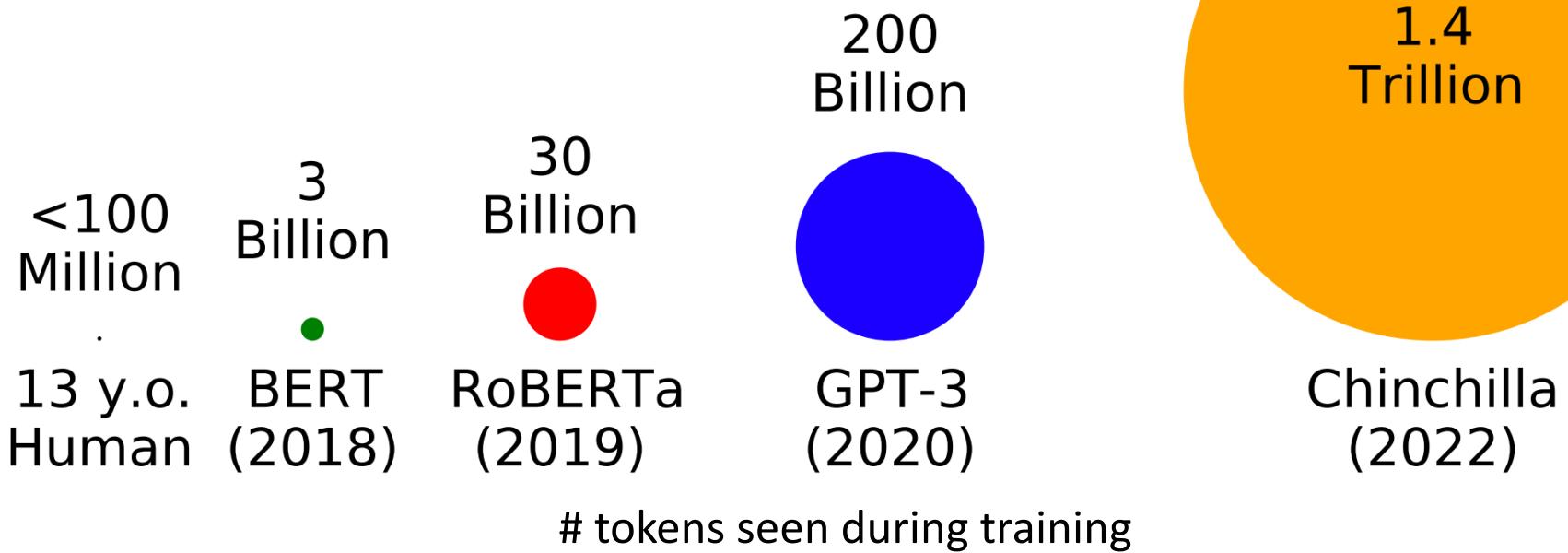
AI training runs, estimated computing resources used

Floating-point operations, selected systems, by type, log scale



Sources: "Compute trends across three eras of machine learning", by J. Sevilla et al., arXiv, 2022; Our World in Data

# More and more data

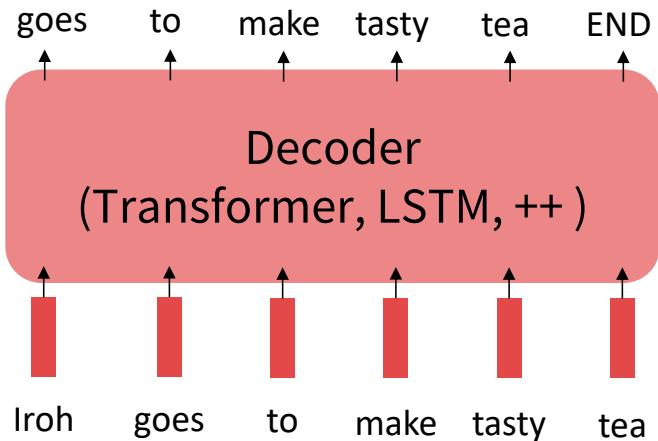


<https://babylm.github.io/>

# Scaling up finetuning

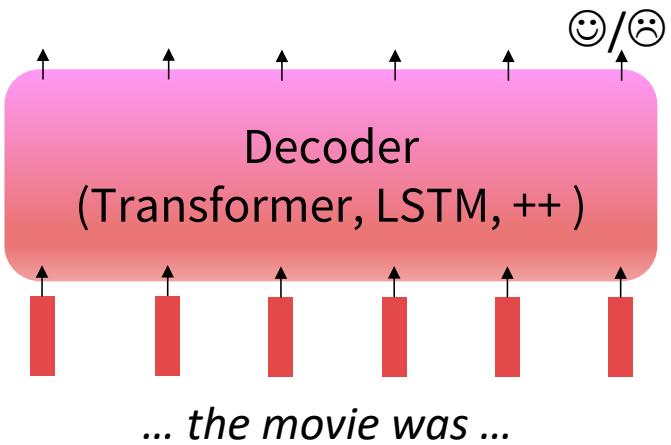
## Step 1: Pretrain (on language modeling)

Lots of text; learn general things!



## Step 2: Finetune (on many tasks)

Not many labels; adapt to the tasks!



# Instruction Finetuning

- Collect examples of (instruction, output) pairs across many tasks and finetune an LM

Please answer the following question.  
What is the boiling point of Nitrogen?

-320.4F

Answer the following question by reasoning step-by-step.  
The cafeteria had 23 apples. If they used 20 for lunch and bought 6 more, how many apples do they have?

The cafeteria had 23 apples originally. They used 20 to make lunch. So they had  $23 - 20 = 3$ . They bought 6 more apples, so they have  $3 + 6 = 9$ .

Language model

- Evaluate on unseen tasks

Q: Can Geoffrey Hinton have a conversation with George Washington?  
Give the rationale before answering.

Geoffrey Hinton is a British-Canadian computer scientist born in 1947. George Washington died in 1799. Thus, they could not have had a conversation together. So the answer is "no".

# Limitations of instruction finetuning

- One limitation of instruction finetuning is obvious: it's **expensive** to collect ground-truth data for tasks.
- But there are other, subtler limitations too. Can you think of any?
- **Problem 1:** tasks like open-ended creative generation have no right answer.
  - *E.g. Write me a story about a dog and her pet grasshopper.*
- **Problem 2:** language modeling penalizes all token-level mistakes equally, but some errors are worse than others.
- Even with instruction finetuning, there a mismatch between the LM objective and the objective of "satisfy human preferences"!
- Can we **explicitly attempt to satisfy human preferences? (alignment)**



# **Reinforcement Learning from Human Feedback (RLHF)**

-- Alignment

# Reinforcement learning to the rescue

- ▶ The field of **reinforcement learning (RL)** has studied these (and related) problems for many years now [[Williams, 1992](#); [Sutton and Barto, 1998](#)]
- ▶ Circa 2013: resurgence of interest in RL applied to deep learning, game-playing [[Mnih et al., 2013](#)]
- ▶ But the interest in applying RL to modern LMs is an even newer phenomenon [[Ziegler et al., 2019](#); [Stiennon et al., 2020](#); [Ouyang et al., 2022](#)]. **Why?**
  - ▶ RL w/ LMs has commonly been viewed as very hard to get right (still is!)
  - ▶ Newer advances in RL algorithms that work for large neural models, including language models (e.g. PPO; [[Schulman et al., 2017](#)])



# Reward maximization from human

- Let's say we were training a language model on some task (e.g. summarization).
- For each LM sample  $s$ , imagine we had a way to obtain a *human reward* of that summary:  $R(s) \in \mathbb{R}$ , higher is better.
- Now we want to maximize the expected reward of samples from our LM:

$$\mathbb{E}_{\hat{s} \sim p_\theta(s)}[R(\hat{s})]$$

SAN FRANCISCO,  
California (CNN) --  
A magnitude 4.2  
earthquake shook the  
San Francisco

...  
overturn unstable  
objects.

An earthquake hit  
San Francisco.  
There was minor  
property damage,  
but no injuries.

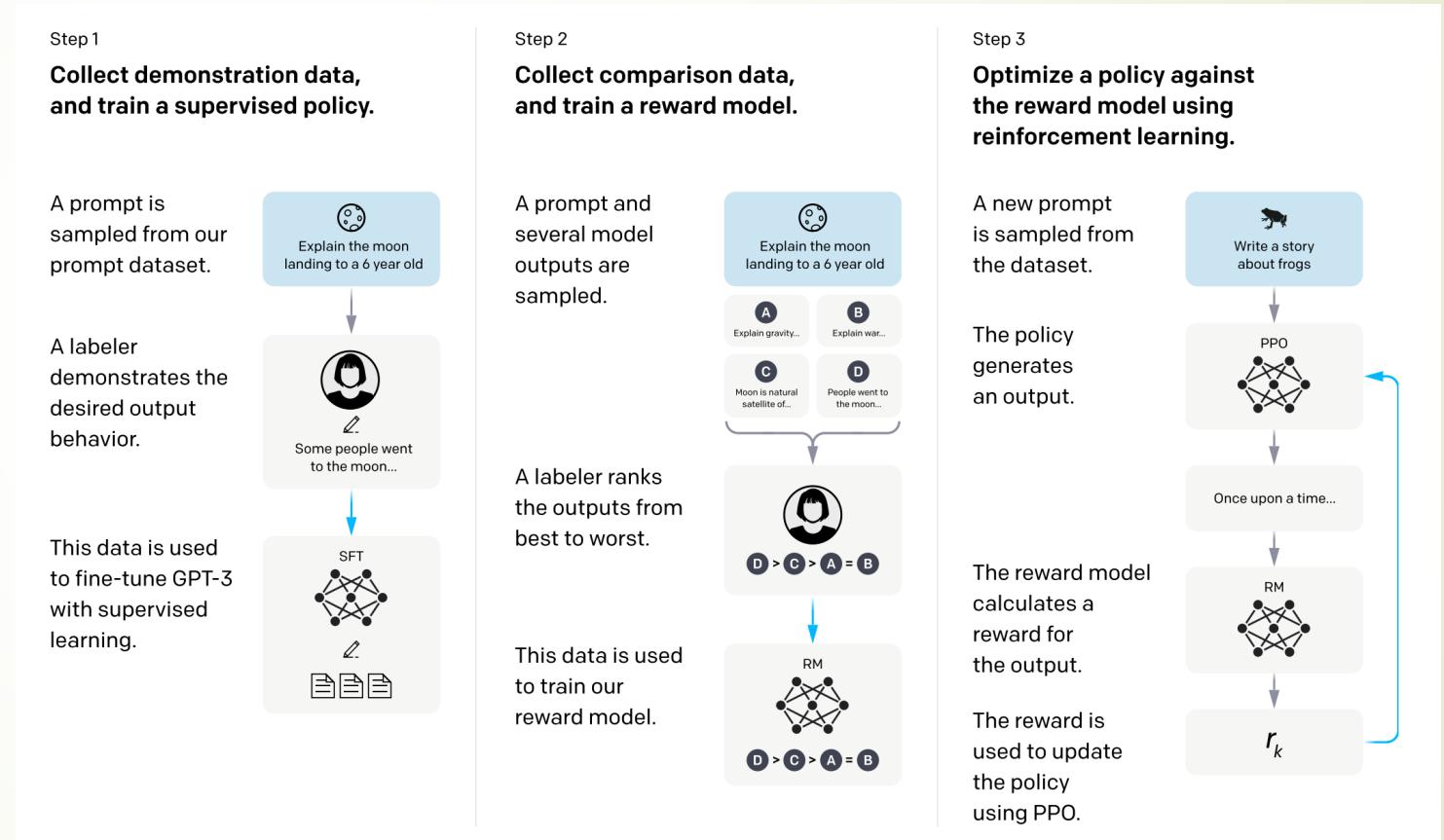
$$s_1 \\ R(s_1) = 8.0$$

The Bay Area has  
good weather but is  
prone to  
earthquakes and  
wildfires.

$$s_2 \\ R(s_2) = 1.2$$

# High-level instantiation: ‘RLHF’ pipeline

- ▶ First step: instruction tuning!
- ▶ Second + third steps: maximize reward (but how??)



# Optimizing for human preferences

- ▶ How do we actually change our LM parameters  $\theta$  to maximize this?

$$\mathbb{E}_{\hat{s} \sim p_{\theta}(s)}[R(\hat{s})]$$

- ▶ Let's try doing gradient ascent!

$$\theta_{t+1} := \theta_t + \alpha \nabla_{\theta_t} \mathbb{E}_{\hat{s} \sim p_{\theta_t}(s)}[R(\hat{s})]$$

How do we estimate  
this expectation??

What if our reward  
function is non-  
differentiable??

- ▶ **Policy gradient** methods in RL (e.g., [Williams, 1992]) give us tools for estimating and optimizing this objective.

# A very brief introduction to Policy Gradient

- We want to obtain

(defn. of expectation) (linearity of gradient)

$$\nabla_{\theta} \mathbb{E}_{\hat{s} \sim p_{\theta}(s)} [R(\hat{s})] = \nabla_{\theta} \sum_s R(s) p_{\theta}(s) = \sum_s R(s) \nabla_{\theta} p_{\theta}(s)$$

- Here we'll use a very handy trick known as the **log-derivative trick**. Let's try taking the gradient of  $\log p_{\theta}(s)$

$$\nabla_{\theta} \log p_{\theta}(s) = \frac{1}{p_{\theta}(s)} \nabla_{\theta} p_{\theta}(s) \quad \Rightarrow \quad \nabla_{\theta} p_{\theta}(s) = p_{\theta}(s) \nabla_{\theta} \log p_{\theta}(s)$$

(chain rule)

- Plug back in:

This is an  
expectation      of this

$$\sum_s R(s) \nabla_{\theta} p_{\theta}(s) = \sum_s p_{\theta}(s) R(s) \nabla_{\theta} \log p_{\theta}(s)$$

$$= \mathbb{E}_{\hat{s} \sim p_{\theta}(s)} [R(\hat{s}) \nabla_{\theta} \log p_{\theta}(\hat{s})]$$

# A very brief introduction to Policy Gradient

- Now we have put the gradient “inside” the expectation, we can approximate this objective with Monte Carlo samples:

$$\nabla_{\theta} \mathbb{E}_{\hat{s} \sim p_{\theta}(s)} [R(\hat{s})] = \mathbb{E}_{\hat{s} \sim p_{\theta}(s)} [R(\hat{s}) \nabla_{\theta} \log p_{\theta}(\hat{s})] \approx \frac{1}{m} \sum_{i=1}^m R(s_i) \nabla_{\theta} \log p_{\theta}(s_i)$$

This is why it’s called “**reinforcement learning**”: we **reinforce** good actions, increasing the chance they happen again.

- Giving us the update rule:

$$\theta_{t+1} := \theta_t + \alpha \frac{1}{m} \sum_{i=1}^m R(s_i) \nabla_{\theta_t} \log p_{\theta_t}(s_i)$$

If  $R$  is +++

If  $R$  is ---

Take gradient steps to maximize  $p_{\theta}(s_i)$

This is **heavily simplified!** There is a *lot* more needed to do RL w/ LMs. **Can you see any problems with this objective?**

Take steps to minimize  $p_{\theta}(s_i)$

# How do we model human preferences?

- ▶ Awesome: now for any **arbitrary, non-differentiable reward function**  $R(s)$ , we can train our language model to maximize expected reward.
- ▶ Not so fast! (Why not?)
- ▶ **Problem 1:** human-in-the-loop is expensive!
  - ▶ **Solution:** instead of directly asking humans for preferences, **model their preferences** as a separate (NLP) problem! [Knox and Stone, 2009]

An earthquake hit San Francisco. There was minor property damage, but no injuries.

$$s_1 \\ R(s_1) = 8.0$$


The Bay Area has good weather but is prone to earthquakes and wildfires.

$$s_2 \\ R(s_2) = 1.2$$


Train an LM  $RM_\phi(s)$  to predict human preferences from an annotated dataset, then optimize for  $RM_\phi$  instead.

# How do we model human preferences?

- ▶ **Problem 2:** human judgments are noisy and miscalibrated!
  - ▶ **Solution:** instead of asking for direct ratings, ask for **pairwise comparisons**, which can be more reliable [Phelps et al., 2015; Clark et al., 2018]

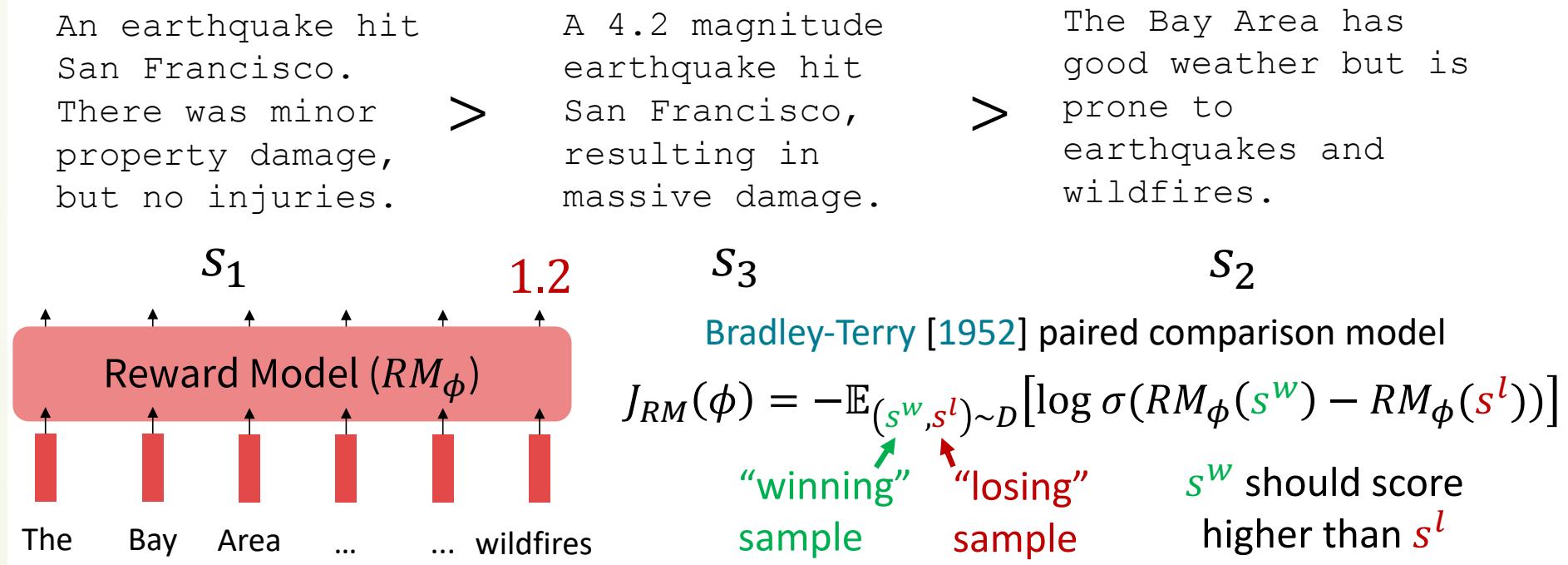
A 4.2 magnitude  
earthquake hit  
San Francisco,  
resulting in  
massive damage.

$s_3$

$R(s_3) = \text{ } 4.1? \text{ } 6.6? \text{ } 3.2?$

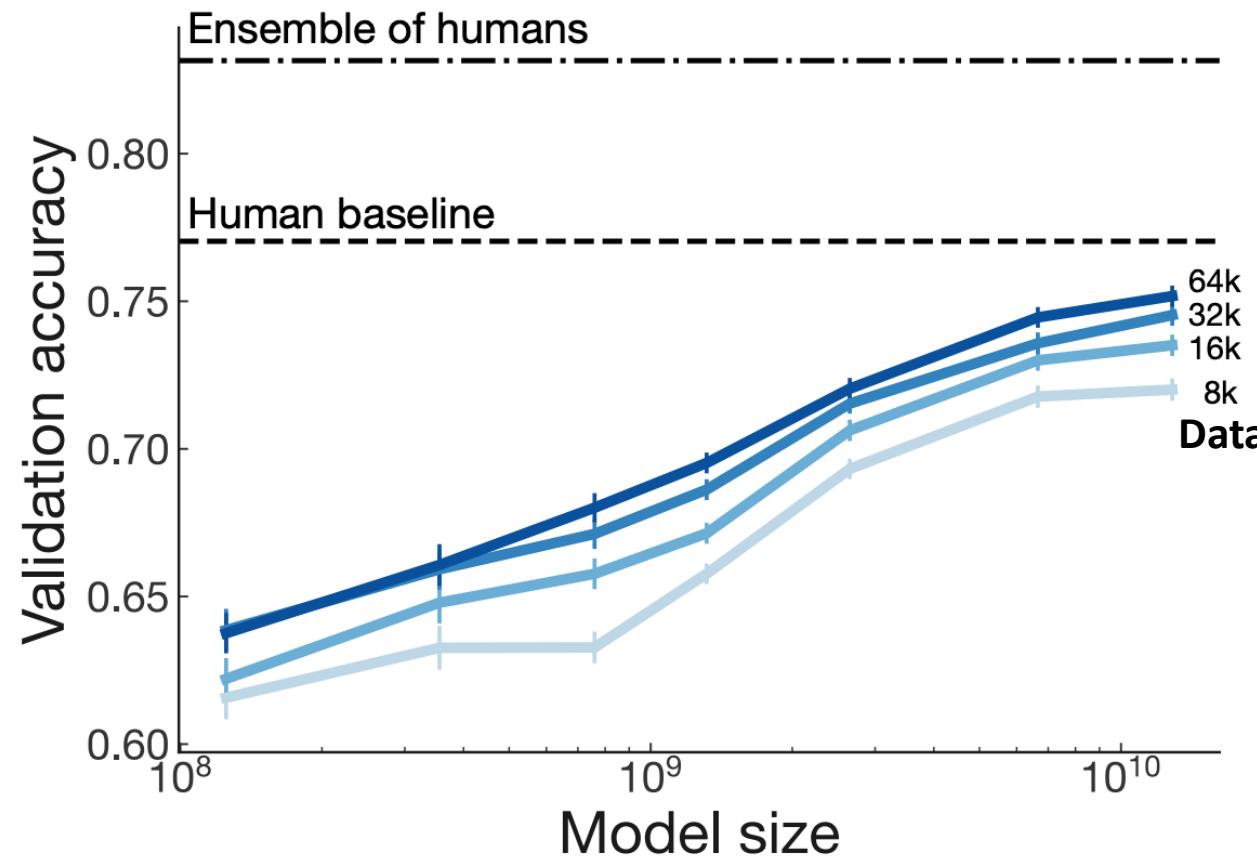
# How do we model human preferences?

- ▶ **Problem 2:** human judgments are noisy and miscalibrated!
  - ▶ **Solution:** instead of asking for direct ratings, ask for **pairwise comparisons**, which can be more reliable [Phelps et al., 2015; Clark et al., 2018]



# Training a reward model first

Evaluate RM on predicting outcome of held-out human judgments



Large enough RM  
trained on enough  
data approaching  
single human perf

[Stiennon et al., 2020]

# RLHF: Putting it all together

[Christiano et al., 2017; Stiennon et al., 2020]

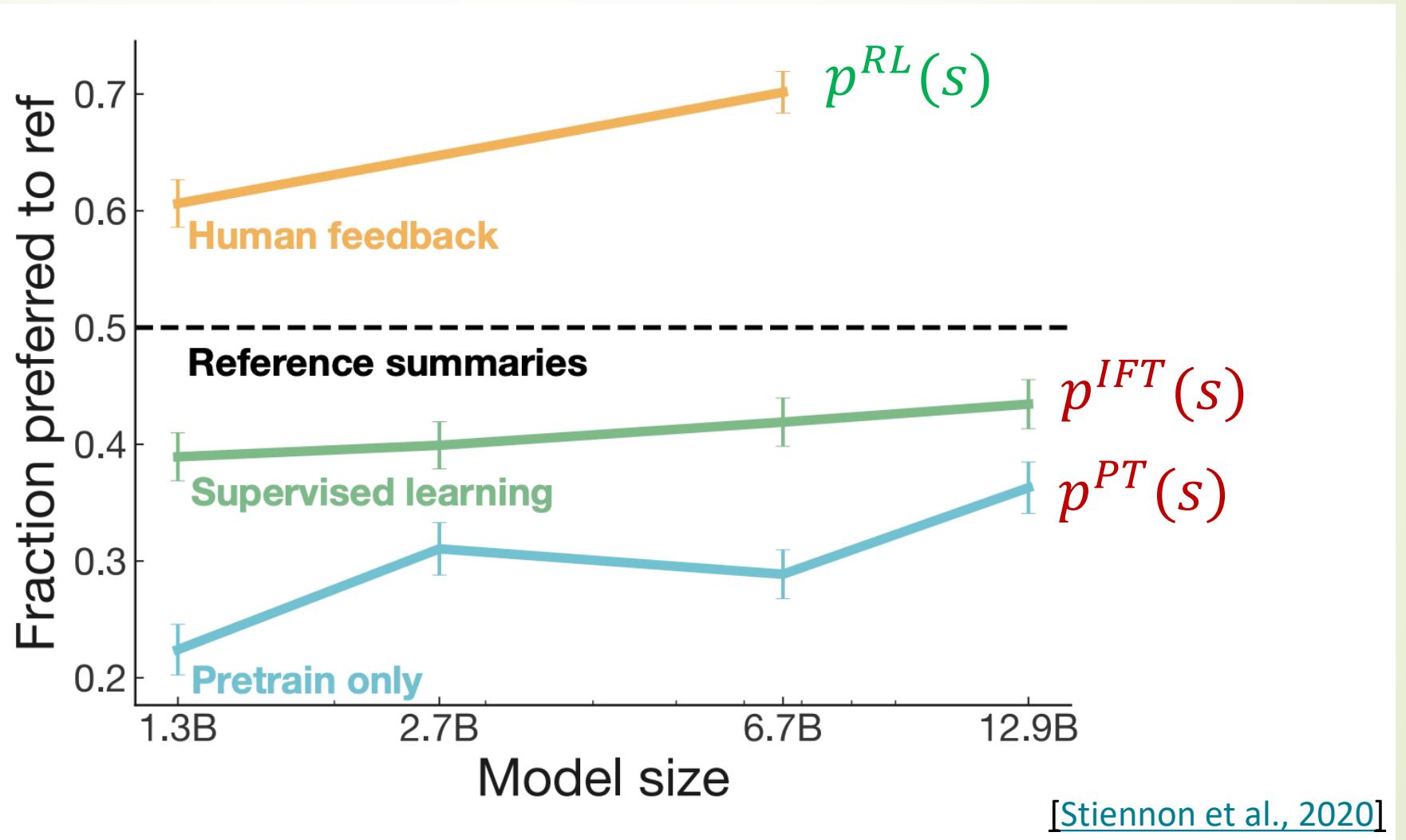
- ▶ Finally, we have everything we need:
  - ▶ A pretrained (possibly instruction-finetuned) LM  $p^{\text{PT}}(s)$
  - ▶ A reward model  $RM(s)$  that produces scalar rewards for LM outputs, trained on a dataset of human comparisons
  - ▶ A method for optimizing LM parameters towards an arbitrary reward function.
- ▶ Now to do RLHF:
  - ▶ Initialize a copy of the model  $p_{\theta}^{\text{RL}}(s)$ , with parameters  $\theta$  we would like to optimize
  - ▶ Optimize the following reward with RL:

$$R(s) = RM_{\phi}(s) - \beta \log \left( \frac{p_{\theta}^{\text{RL}}(s)}{p^{\text{PT}}(s)} \right)$$

Pay a price when  
 $p_{\theta}^{\text{RL}}(s) > p^{\text{PT}}(s)$

This is a penalty which prevents us from diverging too far from the pretrained model. In expectation, it is known as the **Kullback-Leibler (KL)** divergence between  $p_{\theta}^{\text{RL}}(s)$  and  $p^{\text{PT}}(s)$ .

# RLHF improves over pretraining and finetuning



# ChatGPT: Instruction Finetuning + RLHF for dialog agents [<https://openai.com/blog/chatgpt/>]

## ChatGPT: Optimizing Language Models for Dialogue

Note: OpenAI (and similar companies) are keeping more details secret about ChatGPT training (including data, training parameters, model size)—perhaps to keep a competitive edge...

## Methods

We trained this model using Reinforcement Learning from Human Feedback (RLHF), using the same methods as InstructGPT, but with slight differences in the data collection setup. We trained an initial model using supervised fine-tuning: human AI trainers provided conversations in which they played both sides—the user and an AI assistant. We gave the trainers access to model-written suggestions to help them compose their responses. We mixed this new dialogue dataset with the InstructGPT dataset, which we transformed into a dialogue format.

**(Instruction finetuning!)**

# ChatGPT: Instruction Finetuning + RLHF for dialog agents

[<https://openai.com/blog/chatgpt/>]

## ChatGPT: Optimizing Language Models for Dialogue

Note: OpenAI (and similar companies) are keeping more details secret about ChatGPT training (including data, training parameters, model size)—perhaps to keep a competitive edge...

## Methods

To create a reward model for reinforcement learning, we needed to collect comparison data, which consisted of two or more model responses ranked by quality. To collect this data, we took conversations that AI trainers had with the chatbot. We randomly selected a model-written message, sampled several alternative completions, and had AI trainers rank them. Using these reward models, we can fine-tune the model using Proximal Policy Optimization. We performed several iterations of this process.

(RLHF!)

# Yet, Limitations of RL + Reward Modeling

- ▶ **Human preferences are unreliable!**
  - ▶ "Reward hacking" is a common problem in RL
  - ▶ Chatbots are rewarded to produce responses that seem authoritative and helpful, *regardless of truth*
  - ▶ This can result in making up facts + hallucinations
- ▶ **Models** of human preferences are even *more* unreliable!
  - ▶ RLHF labels are often obtained from overseas, low-wage workers
  - ▶ We also need to be quite careful about how annotator biases might creep into LMs

# What's next?

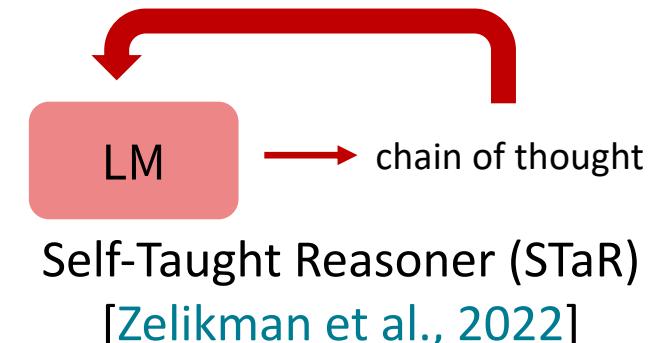
- ▶ RLHF is still a very underexplored and fastmoving area!
- ▶ RLHF gets you further than instruction finetuning, but is (still!) data expensive.
- ▶ Recent work aims to alleviate such data requirements:
  - ▶ RL from **AI feedback** [Bai et al., 2022]
  - ▶ Finetuning LMs on their own outputs [Huang et al., 2022; Zelikman et al., 2022]
  - ▶ However, there are still many limitations of large LMs (size, hallucination) that may not be solvable with RLHF!

## LARGE LANGUAGE MODELS CAN SELF-IMPROVE

Jiaxin Huang<sup>1\*</sup> Shixiang Shane Gu<sup>2</sup> Le Hou<sup>2†</sup> Yuexin Wu<sup>2</sup> Xuezhi Wang<sup>2</sup>  
Hongkun Yu<sup>2</sup> Jiawei Han<sup>1</sup>

<sup>1</sup>University of Illinois at Urbana-Champaign <sup>2</sup>Google  
<sup>1</sup>{jiaxinh3, hanj}@illinois.edu <sup>2</sup>{shanegu, lehou, crickwu, xuezhiw, hongkuny}@google.com

[Huang et al., 2022]



# What's next?

- ▶ RLHF is too sophisticated
- ▶ **DPO, Rafailov et al. NeurIPS 2023**

▶ RLHF optimizes  $\max_{\pi_\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_\theta(y|x)} [r_\phi(x, y)] - \beta \mathbb{D}_{\text{KL}} [\pi_\theta(y|x) || \pi_{\text{ref}}(y|x)]$

▶ whose optimizer is  $\pi_r(y|x) = \frac{1}{Z(x)} \pi_{\text{ref}}(y|x) \exp\left(\frac{1}{\beta} r(x, y)\right)$

▶ which leads to a representation  $r(x, y) = \beta \log \frac{\pi_r(y|x)}{\pi_{\text{ref}}(y|x)} + \beta \log Z(x).$

▶ Then the Bradley-Terry model becomes

▶ Maximizing the Likelihood:

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} \left[ \log \sigma \left( \beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)} \right) \right]$$

$$\nabla_\theta \mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\beta \mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} \left[ \underbrace{\sigma(\hat{r}_\theta(x, y_l) - \hat{r}_\theta(x, y_w))}_{\text{higher weight when reward estimate is wrong}} \left[ \underbrace{\nabla_\theta \log \pi(y_w|x)}_{\text{increase likelihood of } y_w} - \underbrace{\nabla_\theta \log \pi(y_l|x)}_{\text{decrease likelihood of } y_l} \right] \right], \quad \hat{r}_\theta(x, y) = \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$$

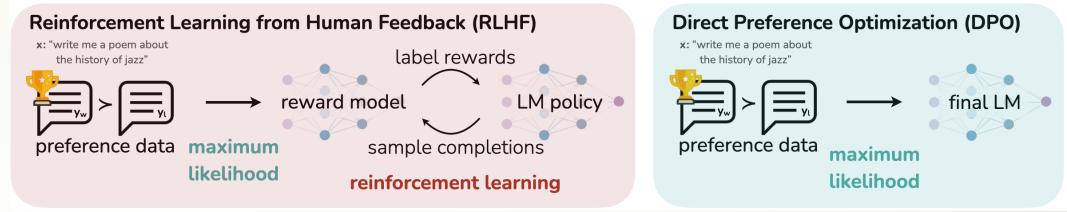


Figure 1: **DPO optimizes for human preferences while avoiding reinforcement learning.** Existing methods for fine-tuning language models with human feedback first fit a reward model to a dataset of prompts and human preferences over pairs of responses, and then use RL to find a policy that maximizes the learned reward. In contrast, DPO directly optimizes for the policy best satisfying the preferences with a simple classification objective, without an explicit reward function or RL.

Thank you!

