



Attention, Transformer, 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**
- ▶ 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 certain tasks.
 - ▶ 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

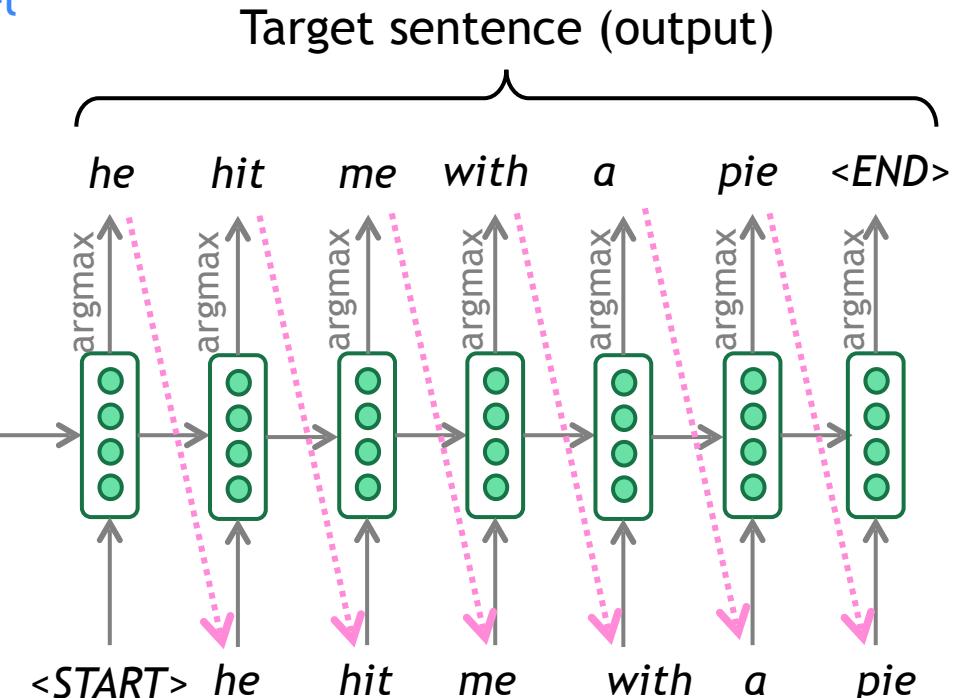
Encoding of the source sentence.
Provides initial hidden state
for Decoder RNN.

Encoder RNN

il a m' entarté

Source sentence (input)

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

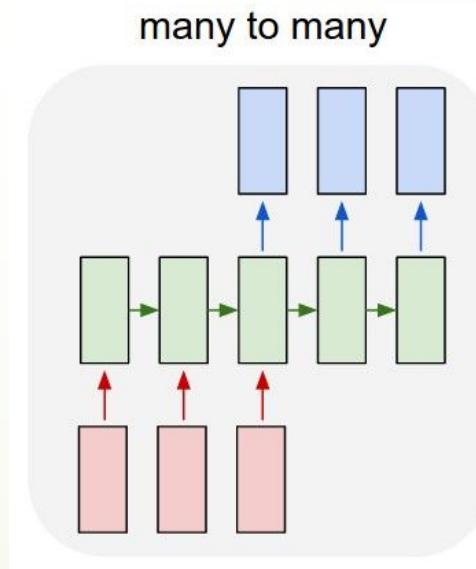


Decoder RNN is a Language Model that generates target sentence, *conditioned on encoding*.

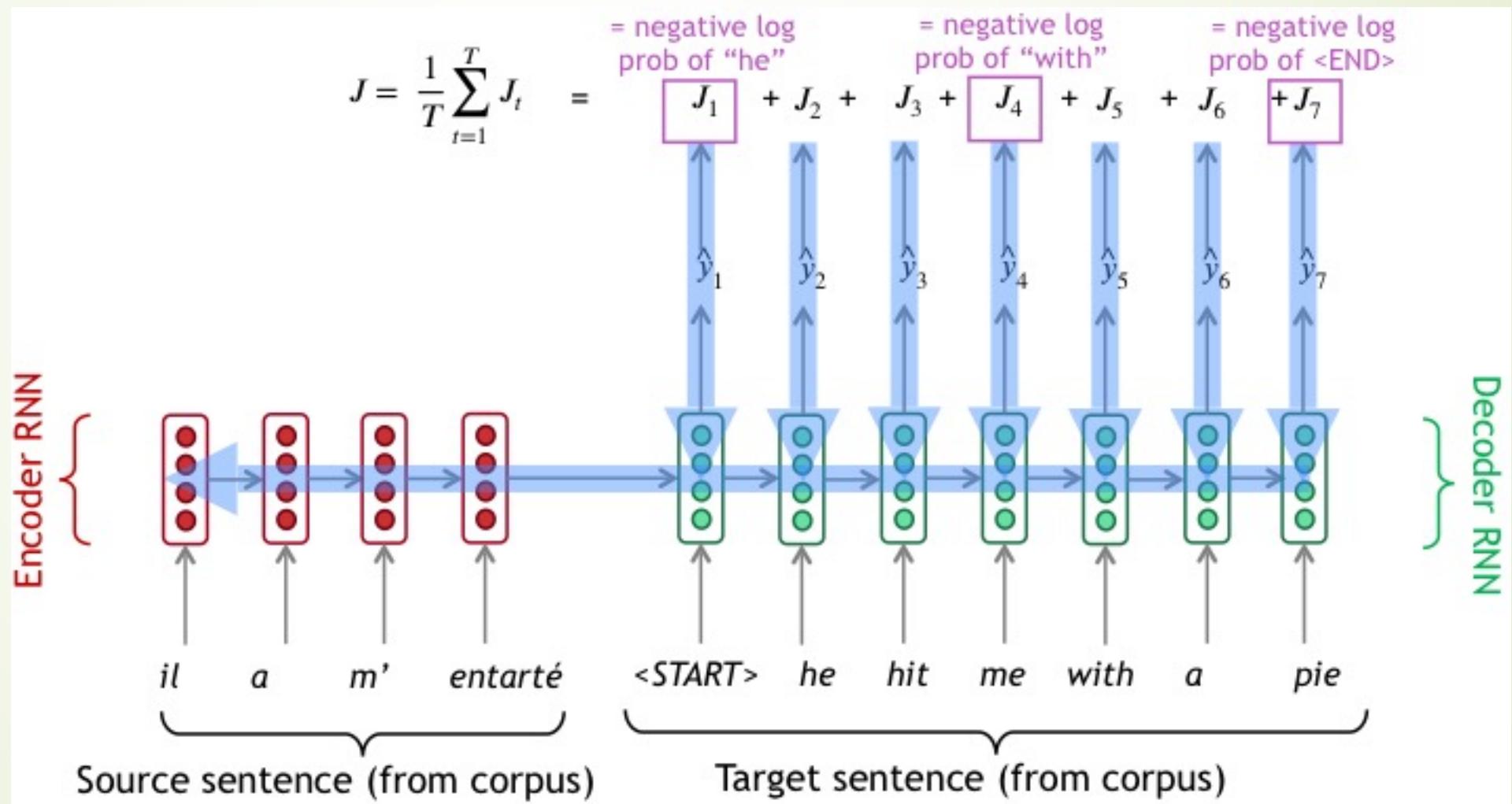
Note: This diagram shows test time behavior:
decoder output is fed in as next step's input

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)

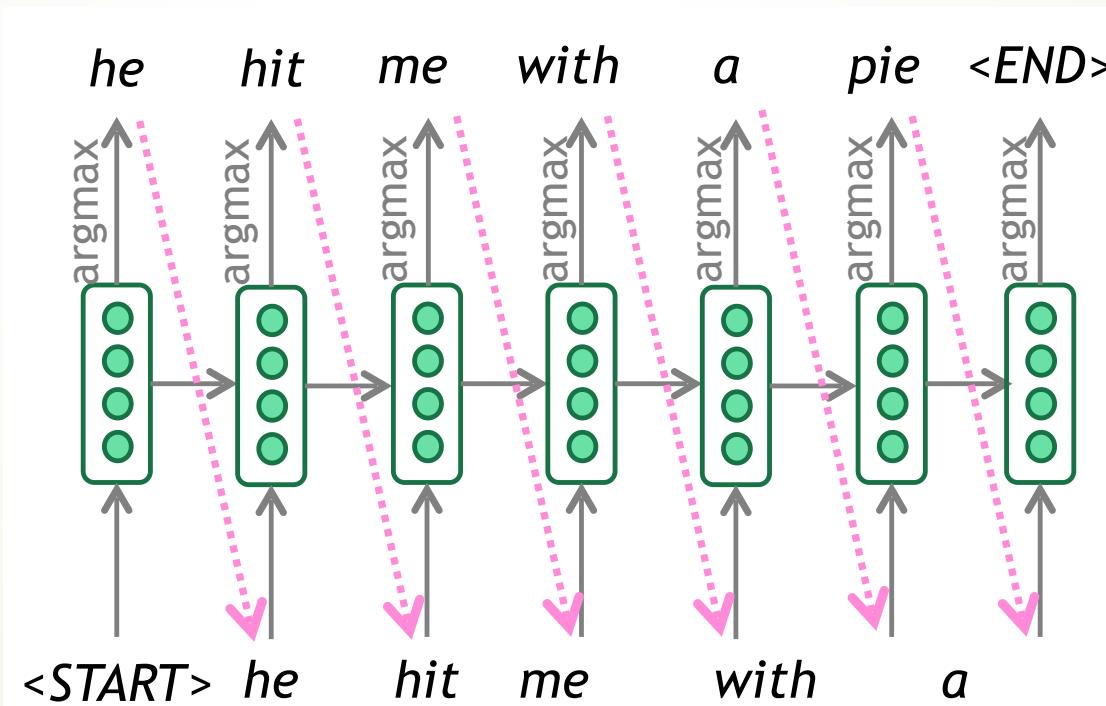


Training a NMT system by BP



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

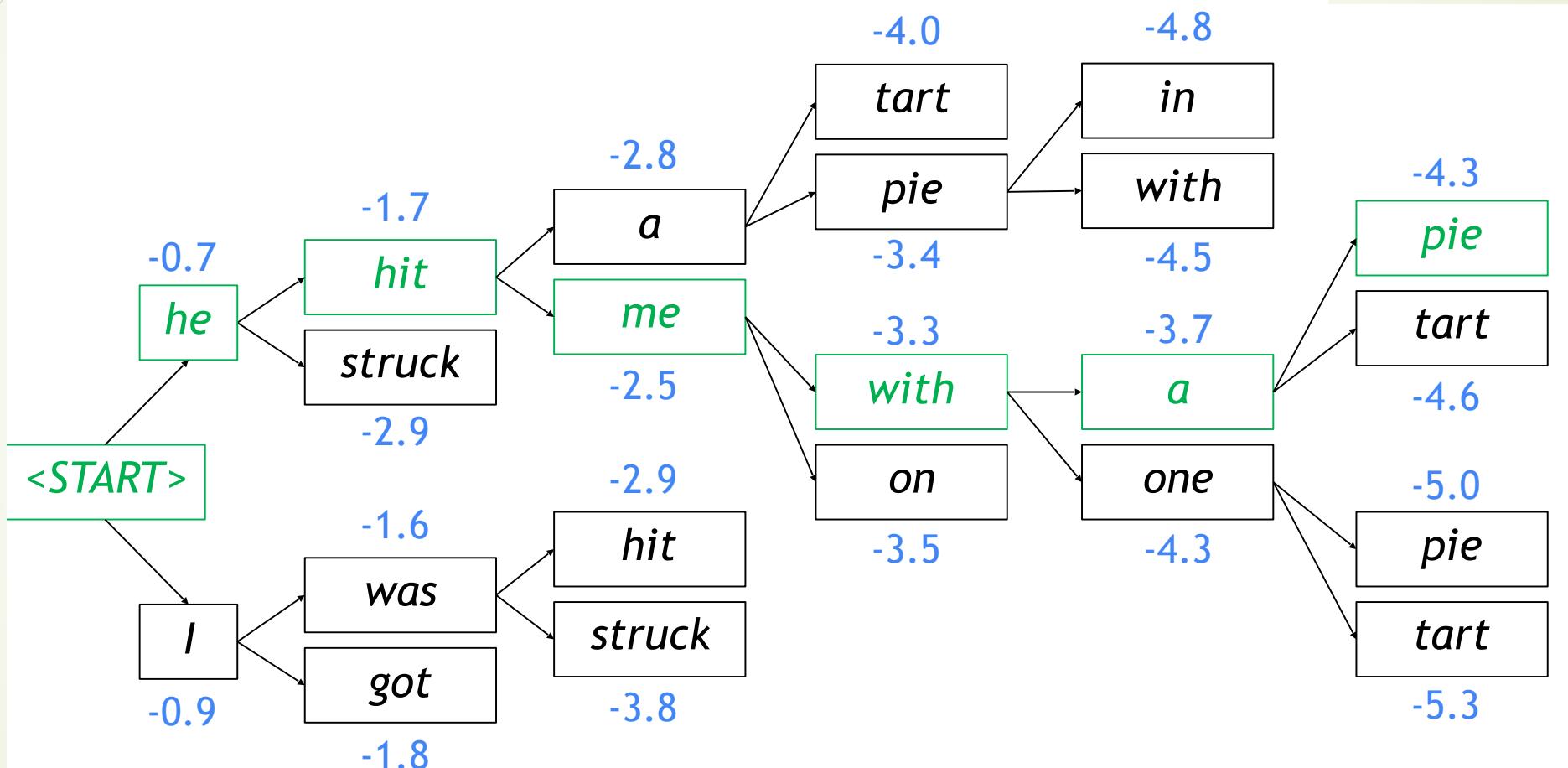
- ▶ Core idea: On each step of decoder, keep track of the **k most probable** partial translations (which we call *hypotheses*)
 - ▶ k is the beam size (in practice around 5 to 10)
- ▶ A hypothesis $(y(1), \dots, y(t))$ has a score which is its log probability:

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

- ▶ 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!

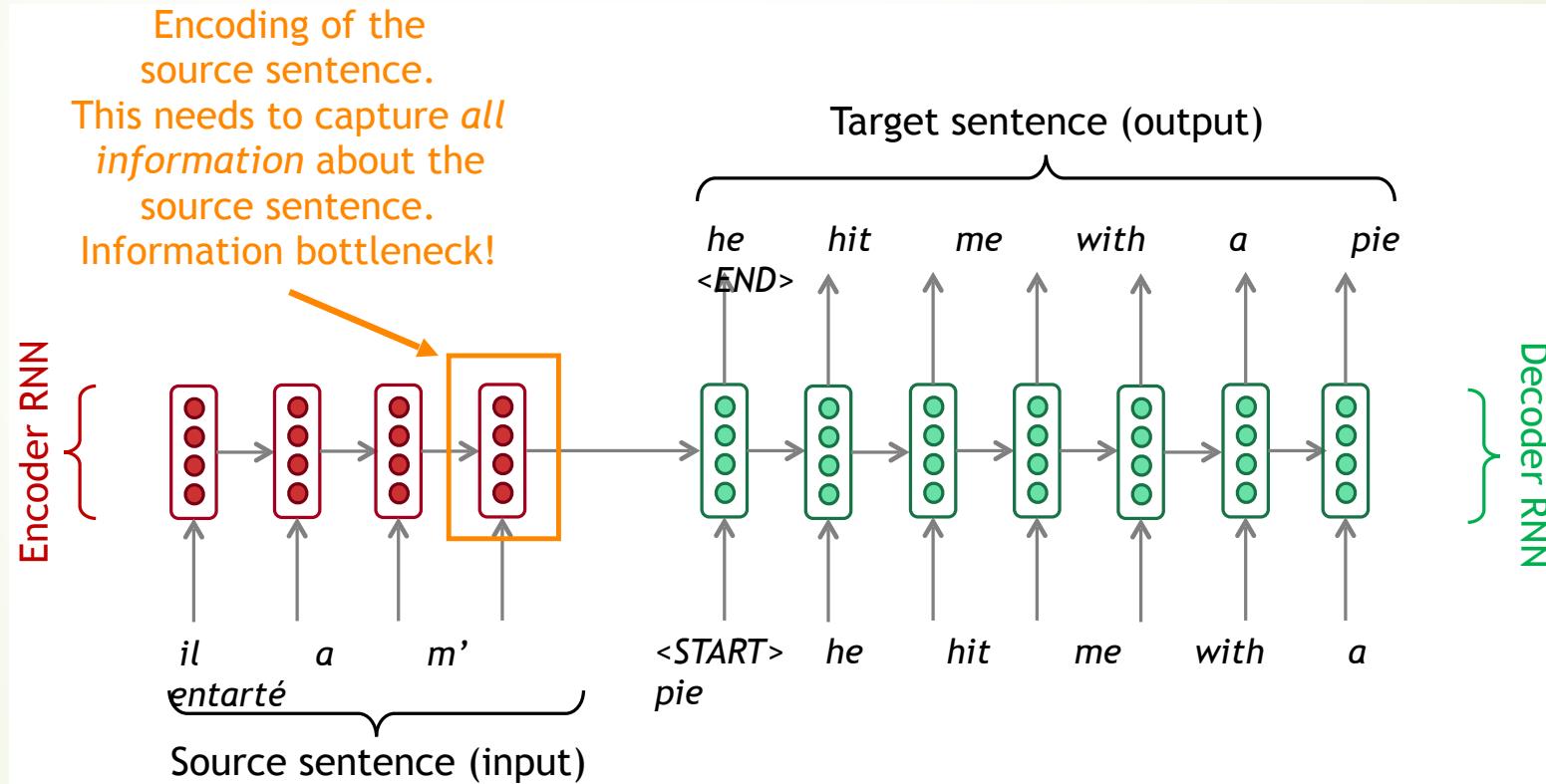
Beam search decoding example:

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

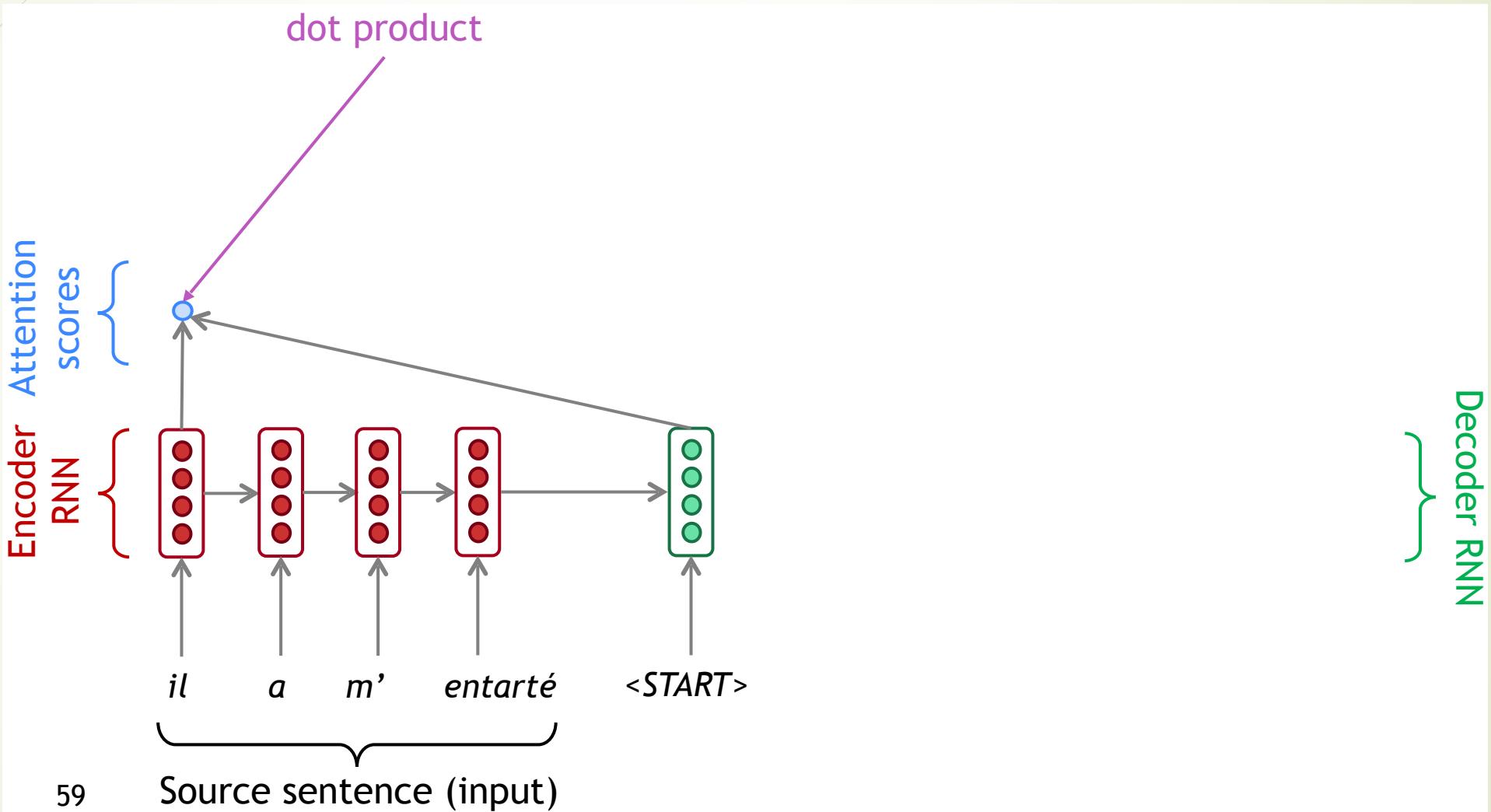


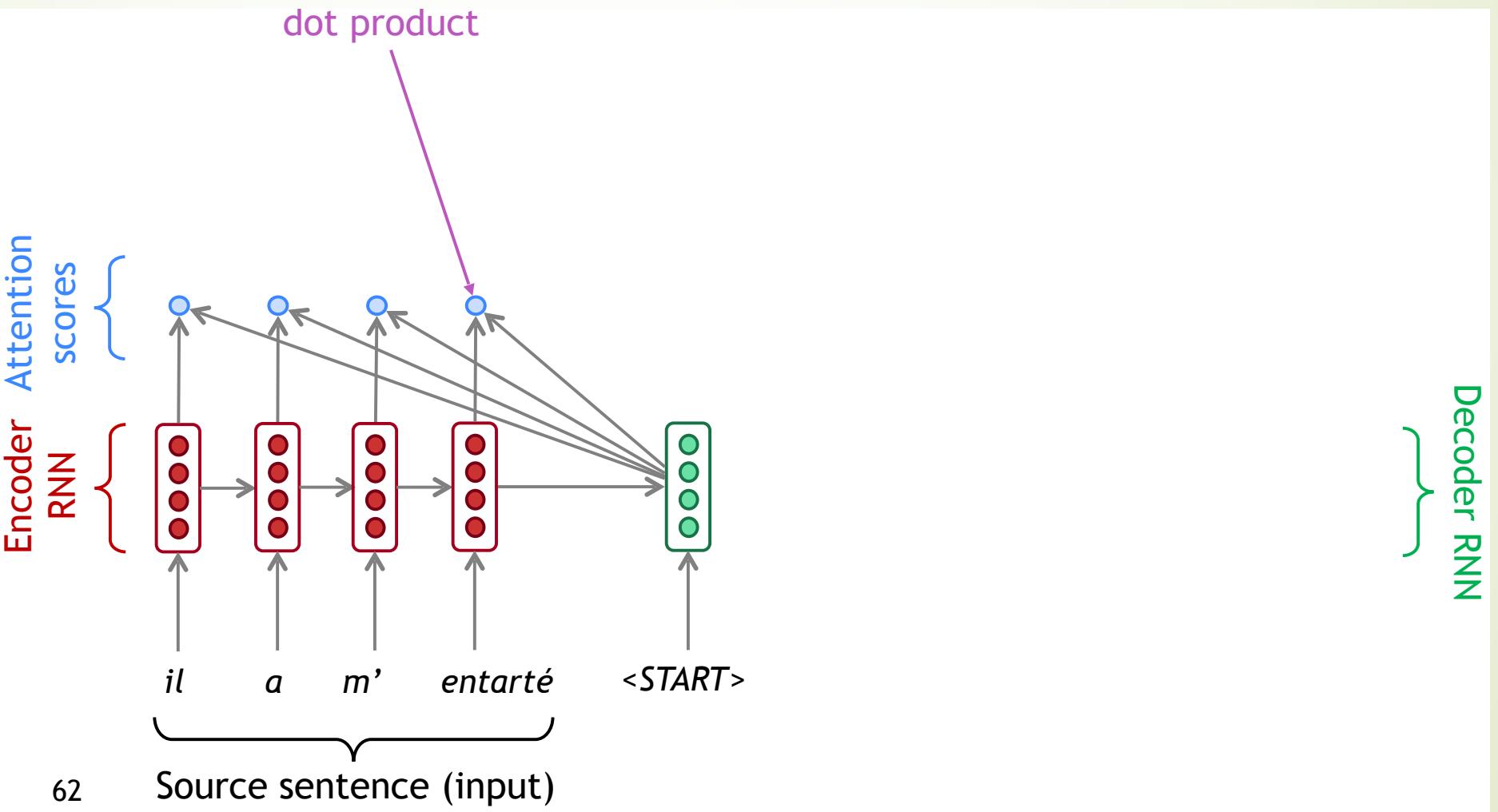


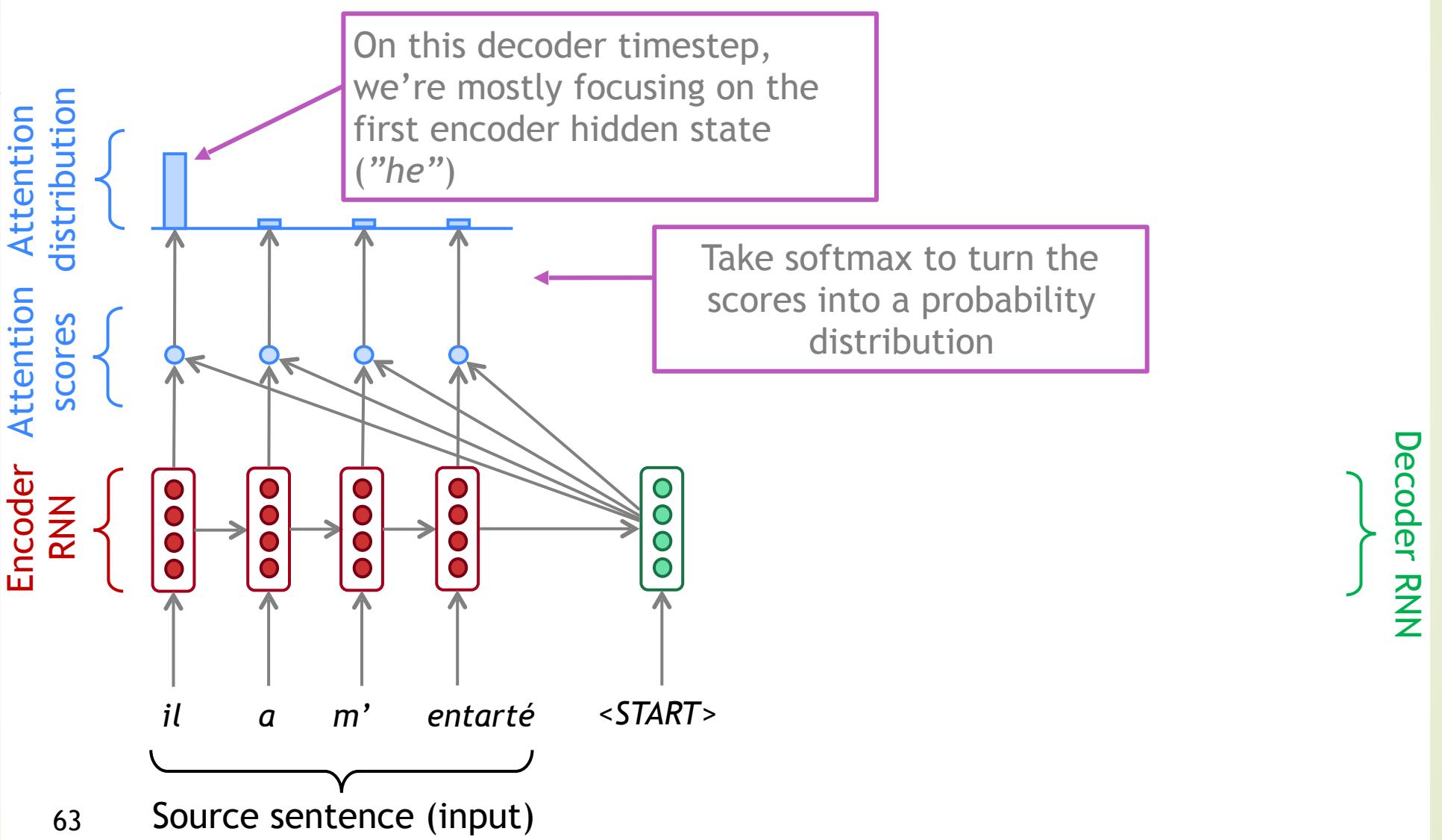
Attention Mechanism

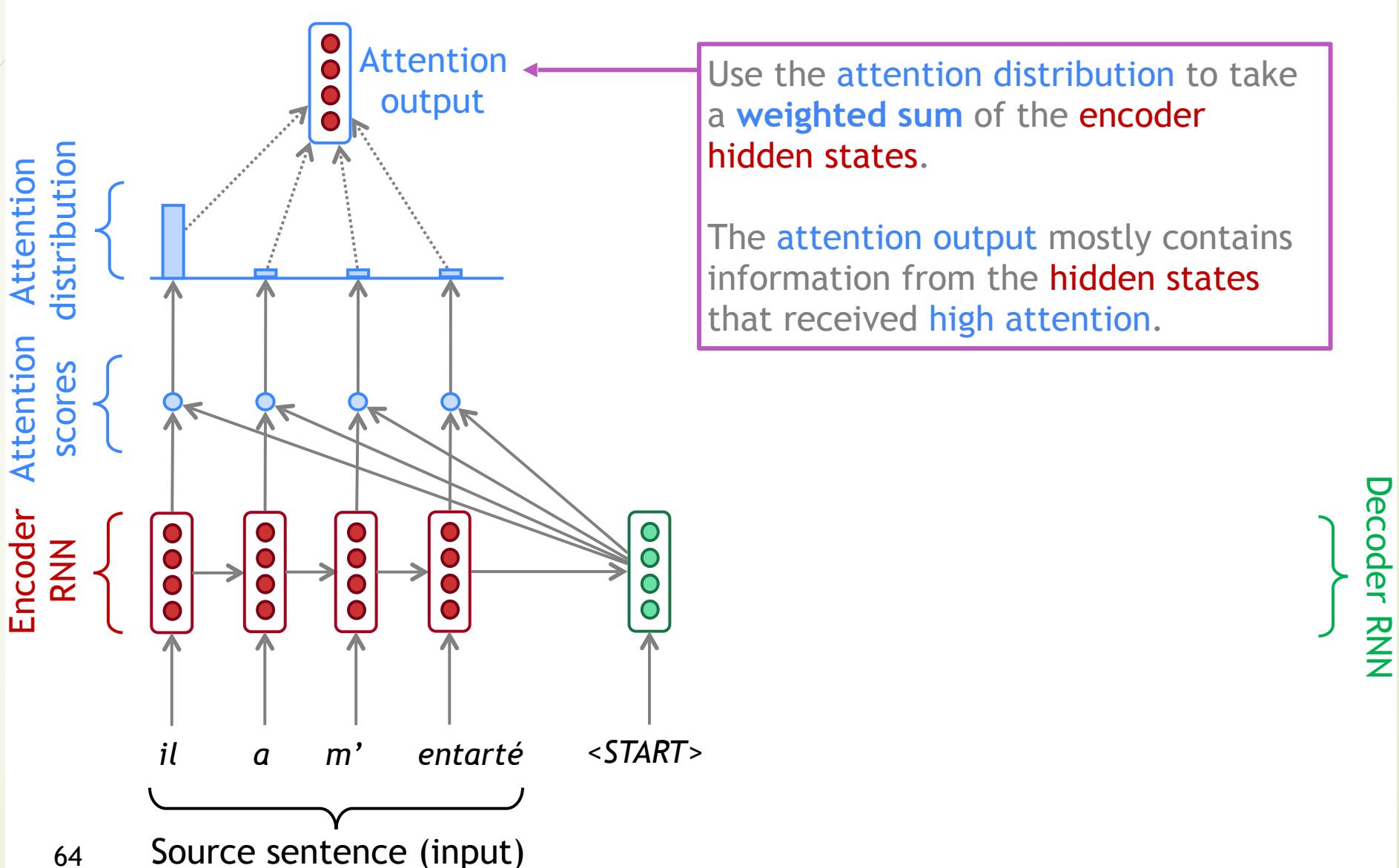
It was firstly invented in computer vision, then to NLP.

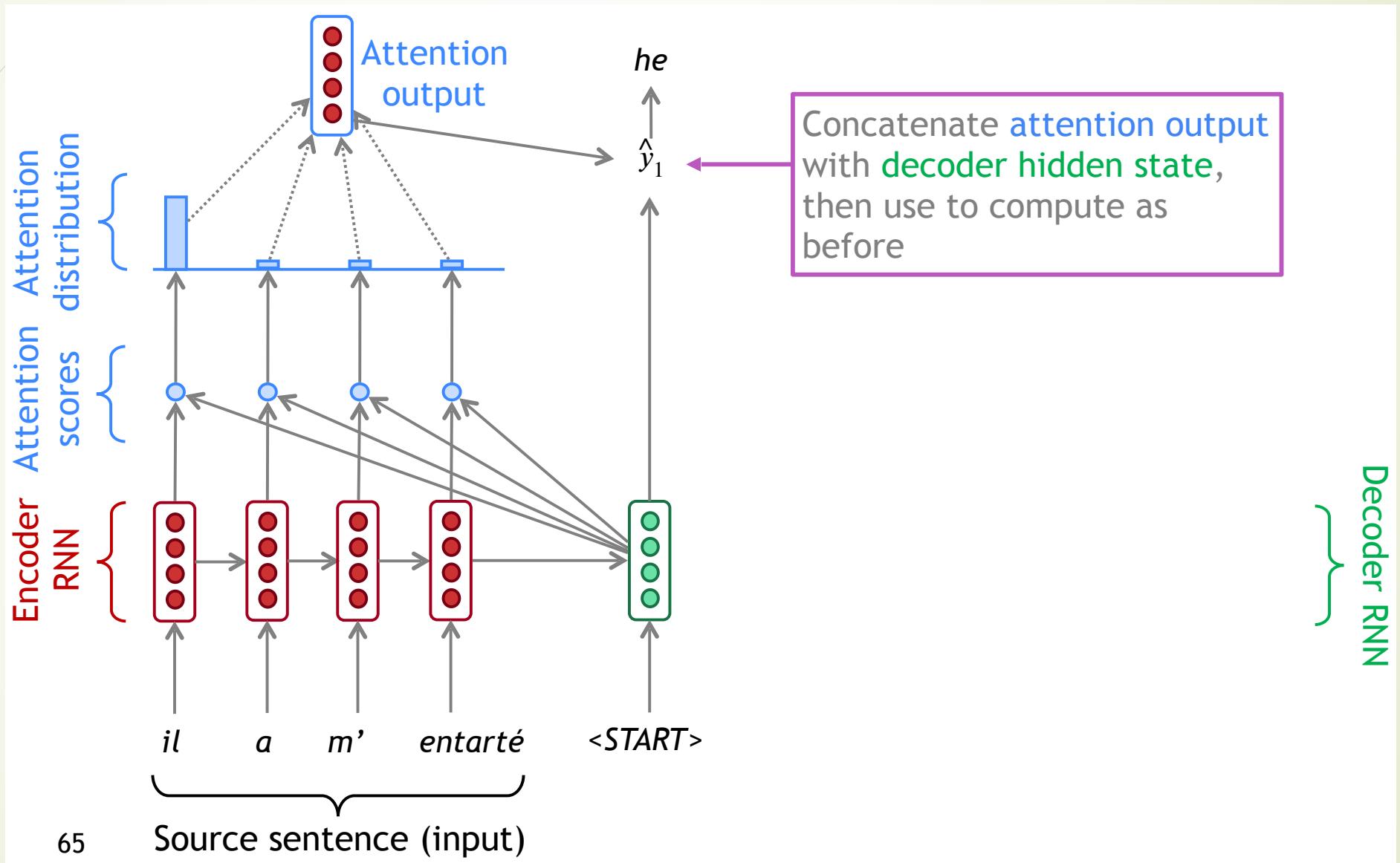
Sequence-to-sequence with attention

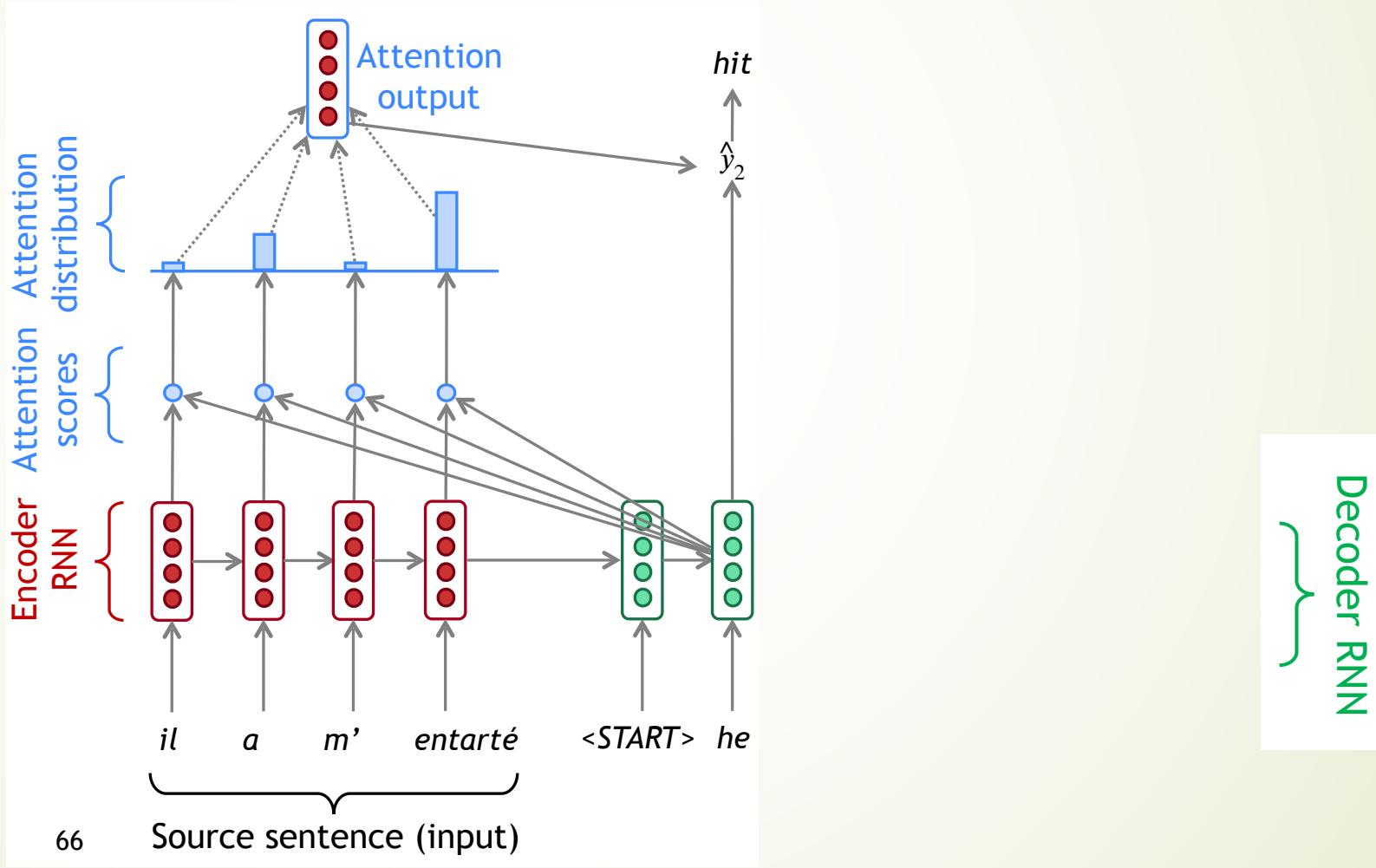


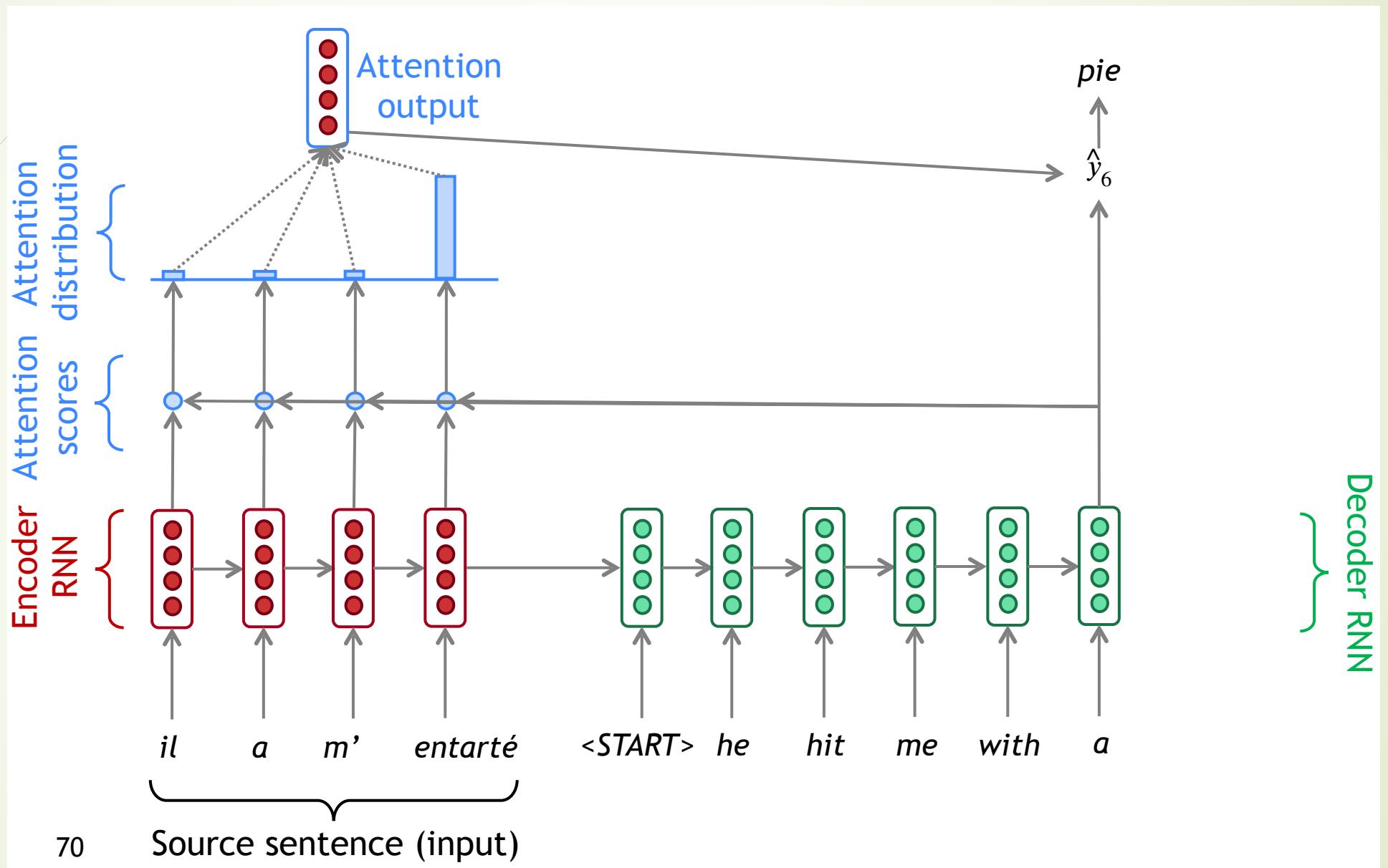












Attention in Equations

- 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 α^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 α^t to take a weighted sum of the encoder hidden states to get the attention output

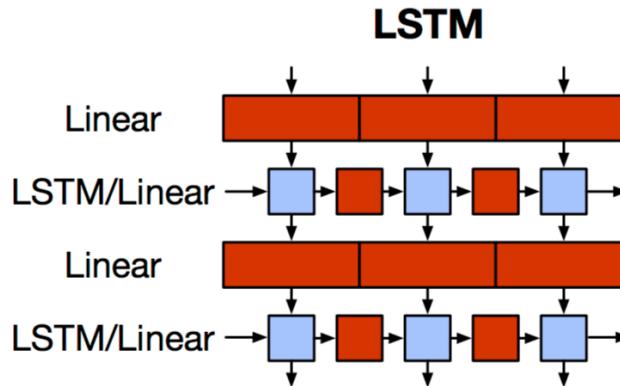
$$a_t = \sum_{i=1}^N \alpha_i^t h_i \in \mathbb{R}^h$$

- 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

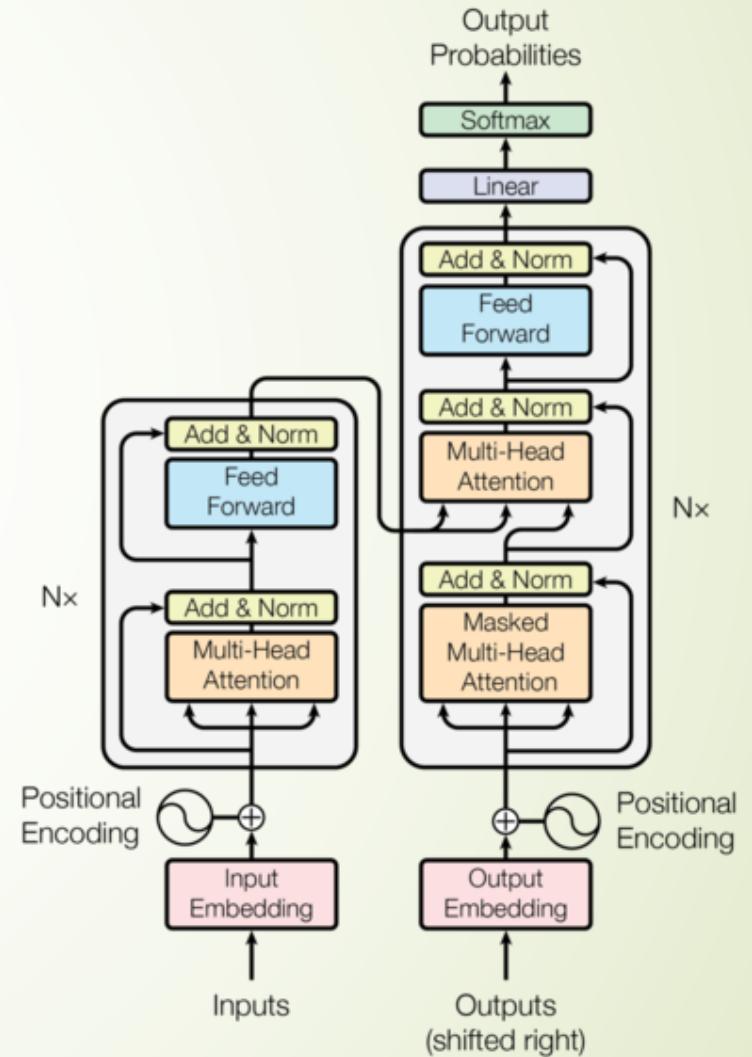


Transformer

“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

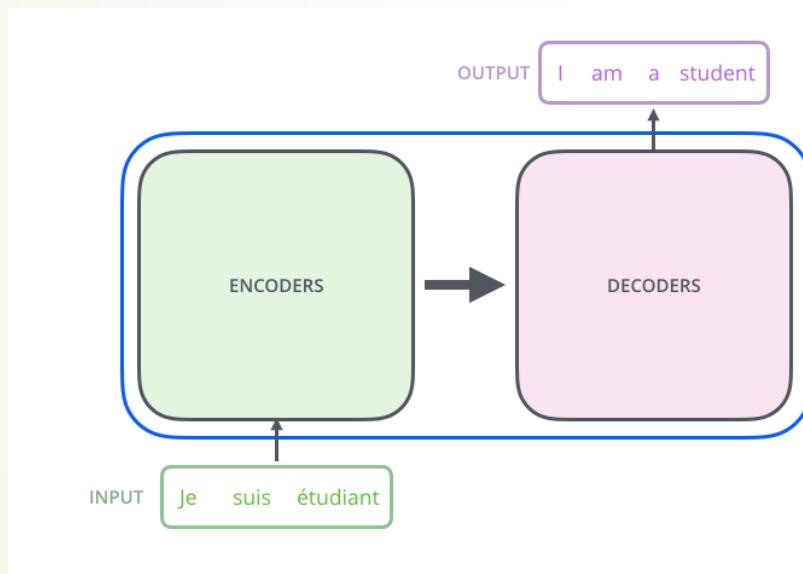


Transformer Pytorch Notebook

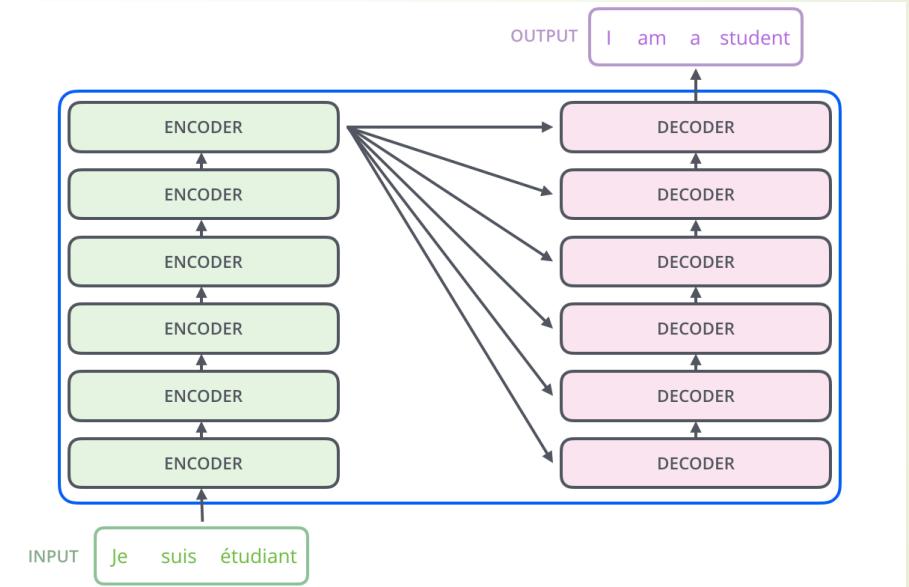
- ▶ Learning about transformers on your own?
- ▶ Key recommended resource:
 - ▶ <http://nlp.seas.harvard.edu/2018/04/03/attention.html>
 - ▶ The Annotated Transformer by Sasha Rush, a Jupyter Notebook using PyTorch that explains everything!
- ▶ <https://jalammar.github.io/illustrated-transformer/>
-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

Encoder-Decoder Blocks

Encoder-Decoder

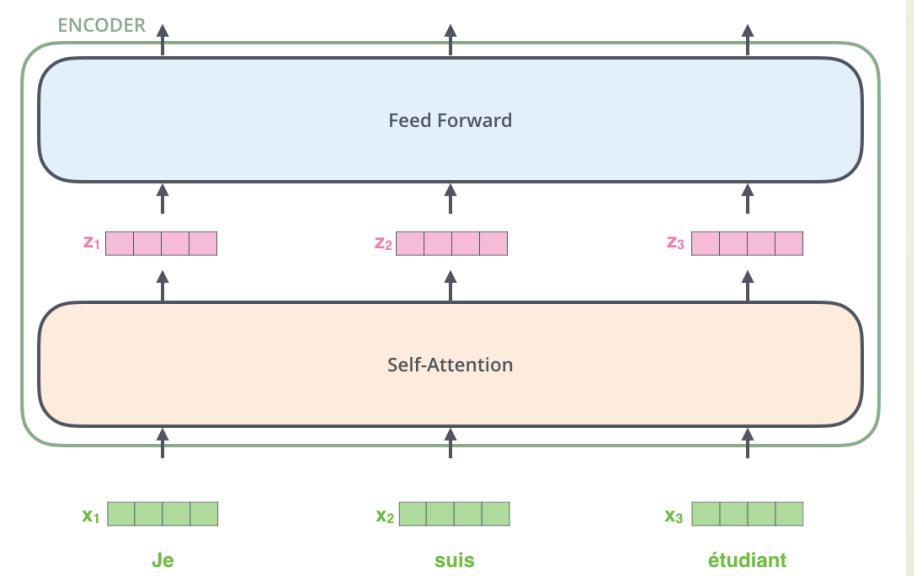
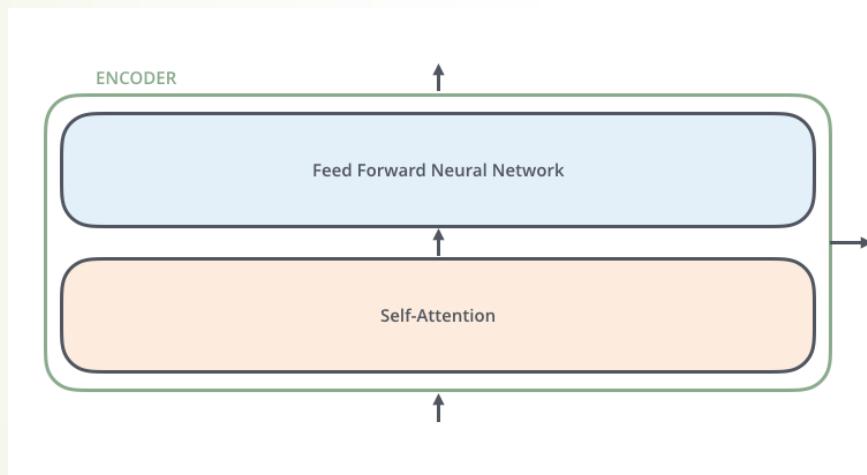


N=6 layers



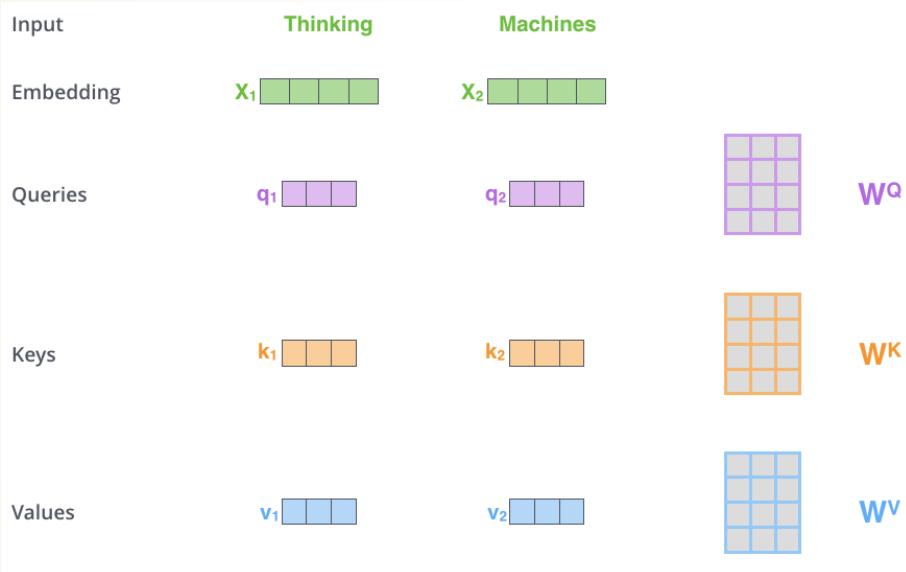
Encoder has two layers

Self-Attention +
FeedForward

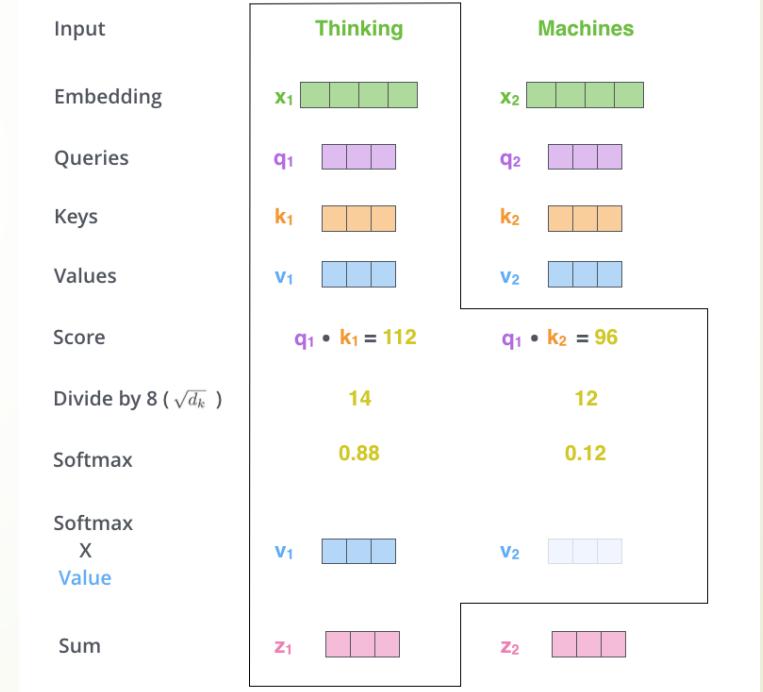


Attention Illustration

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

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

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

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

Dot-Product Attention: Matrix Form

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

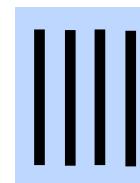
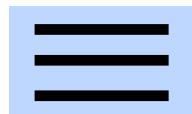
$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$



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

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

softmax
row-wise



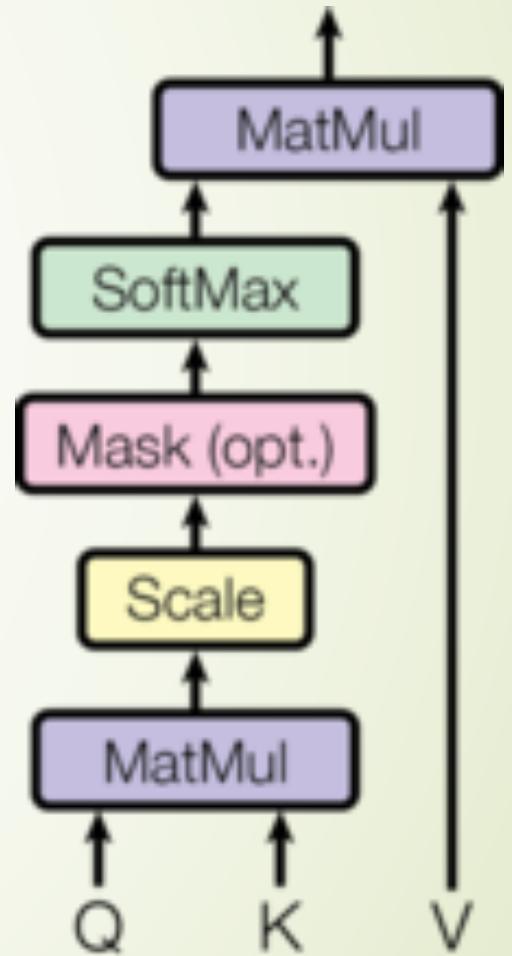
$$= [|Q| \times d_v]$$

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:

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

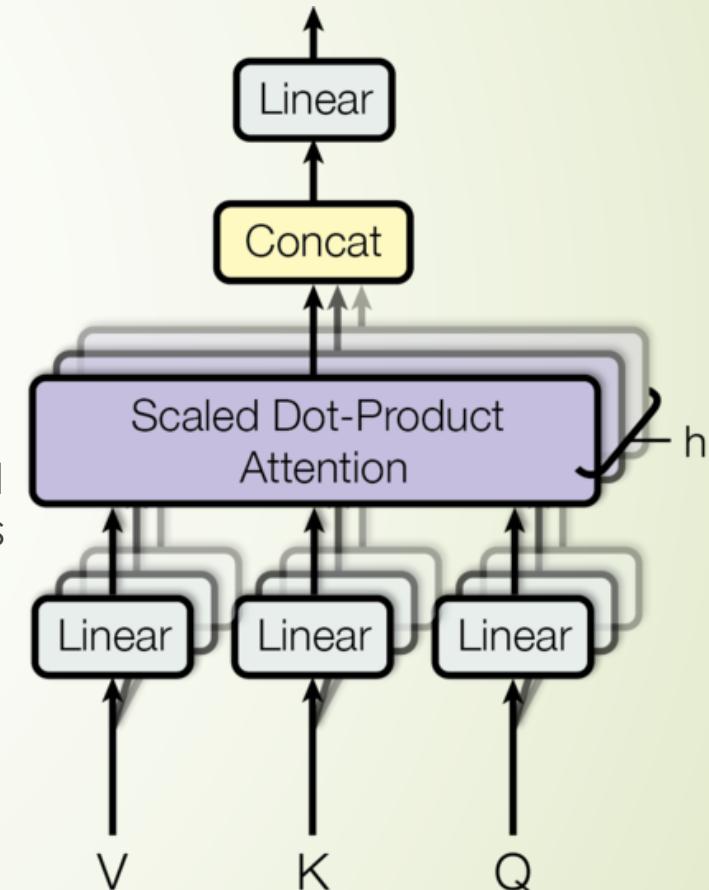


Multi-head Attention

- ▶ **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.

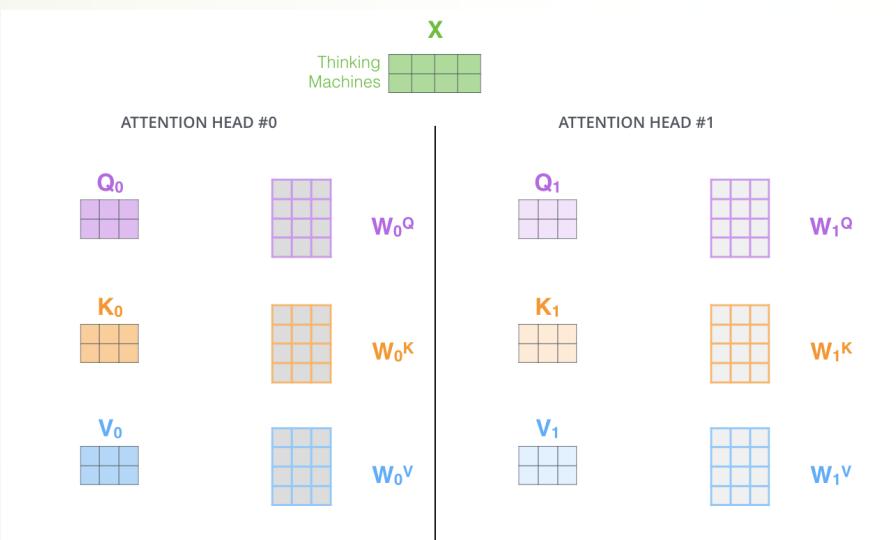
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

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

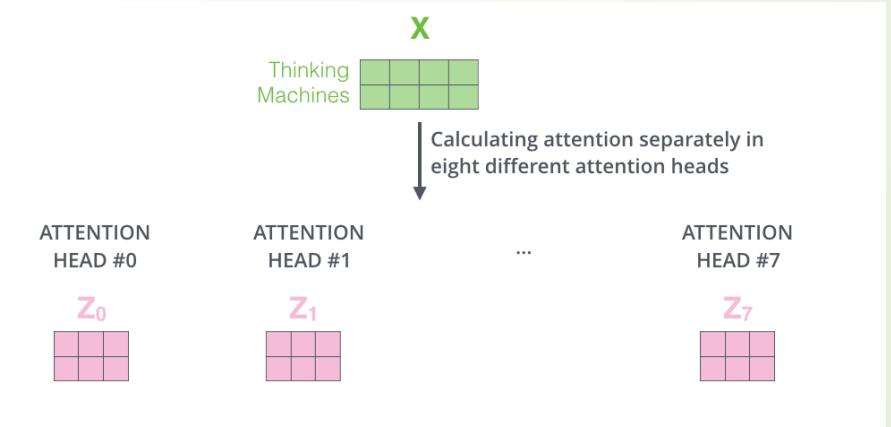


Multihead

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

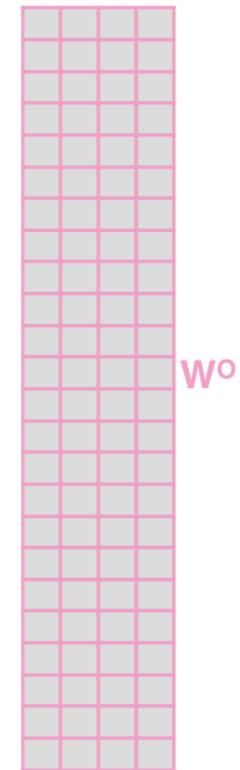
$$= \begin{matrix} Z \\ \hline \end{matrix}$$

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

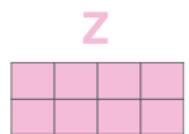
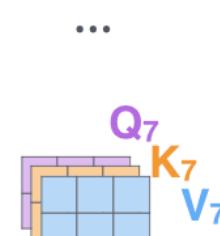
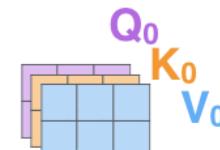
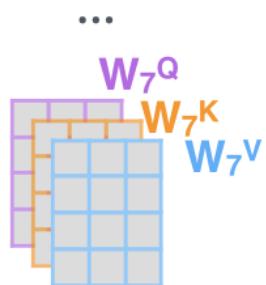
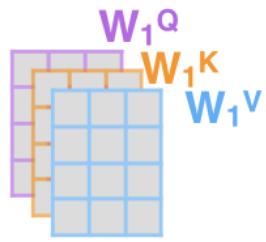
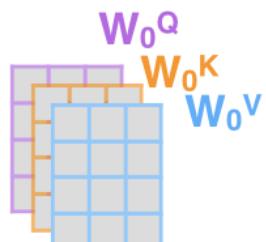
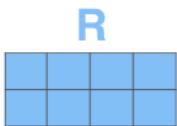
4) Calculate attention using the resulting $Q/K/V$ matrices

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

Thinking
Machines



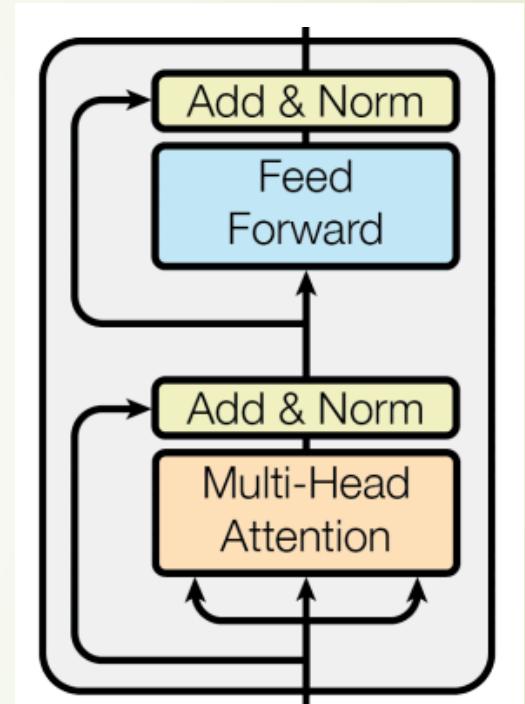
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



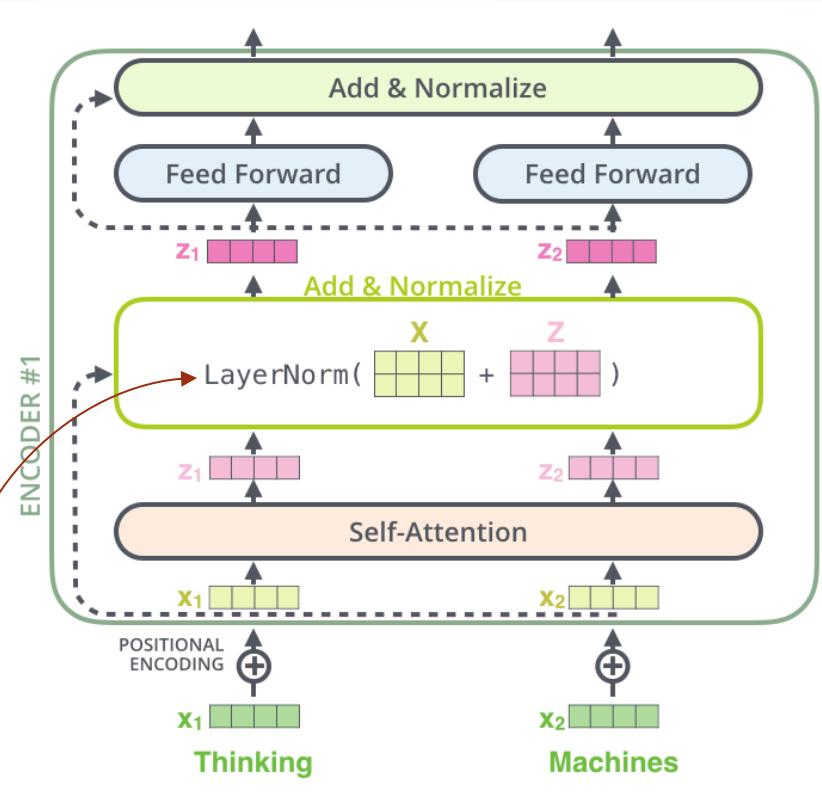
A Transformer block

- ▶ 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)

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2} \quad h_i = f\left(\frac{g_i}{\sigma_i} (a_i - \mu_i) + b_i\right)$$



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:

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

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

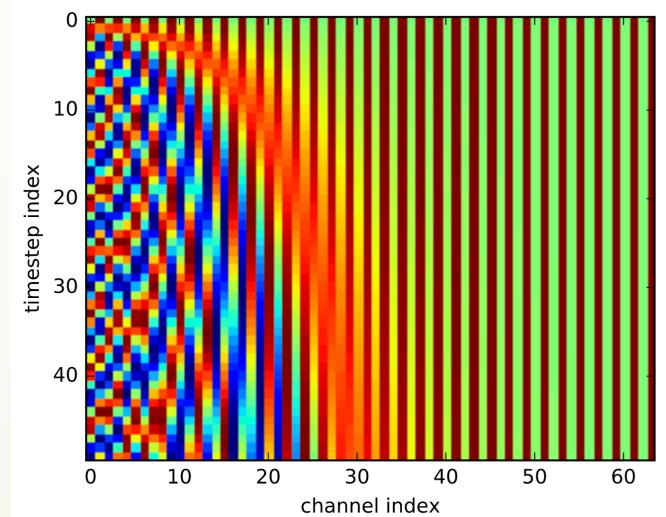
Or learned

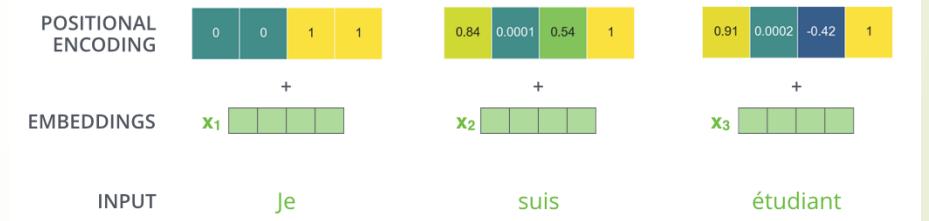
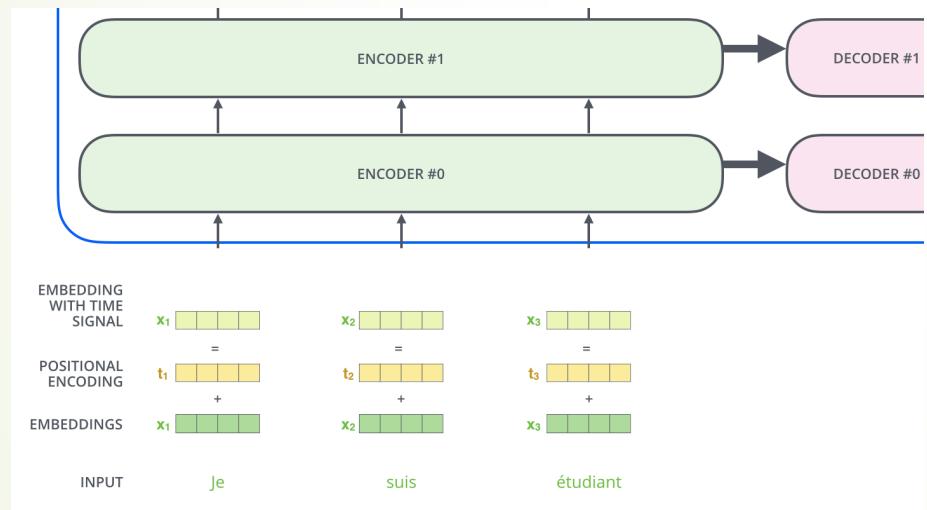
Dictionary

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

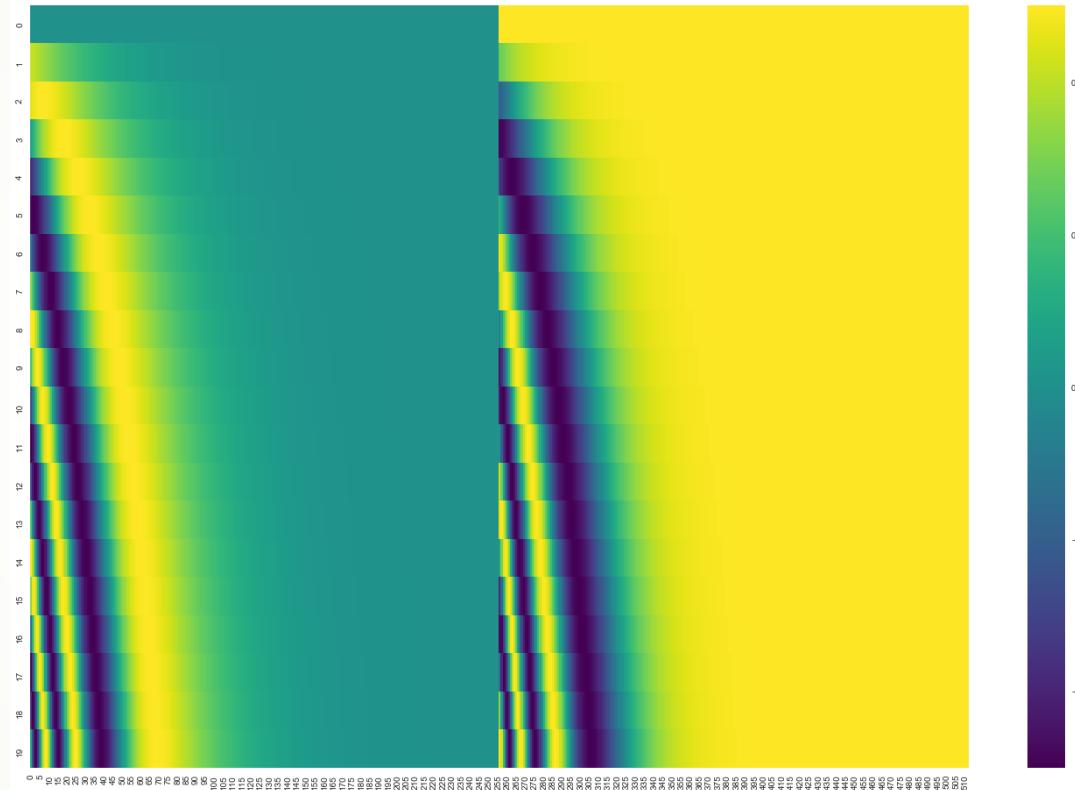
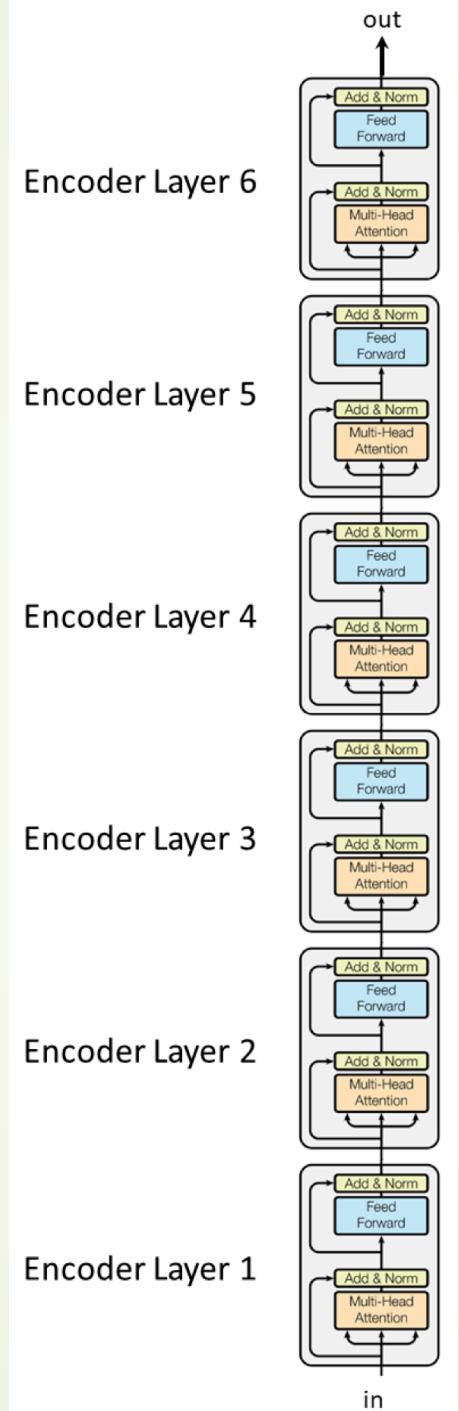
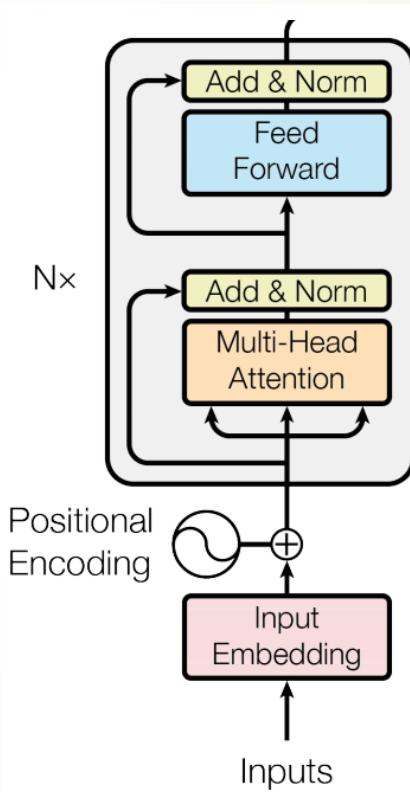


Figure. Each row corresponds to a positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.

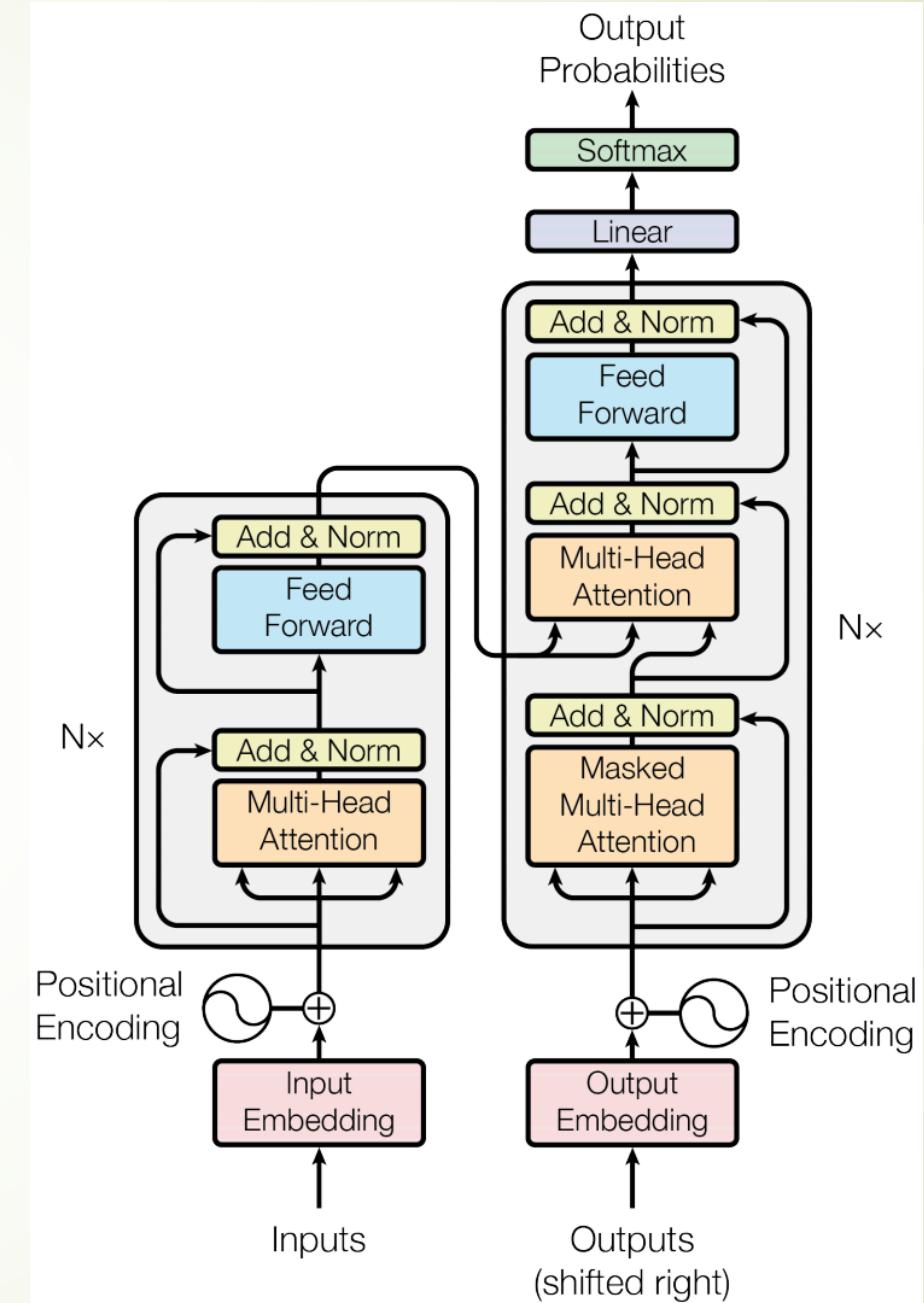
Transformer Encoder

- 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

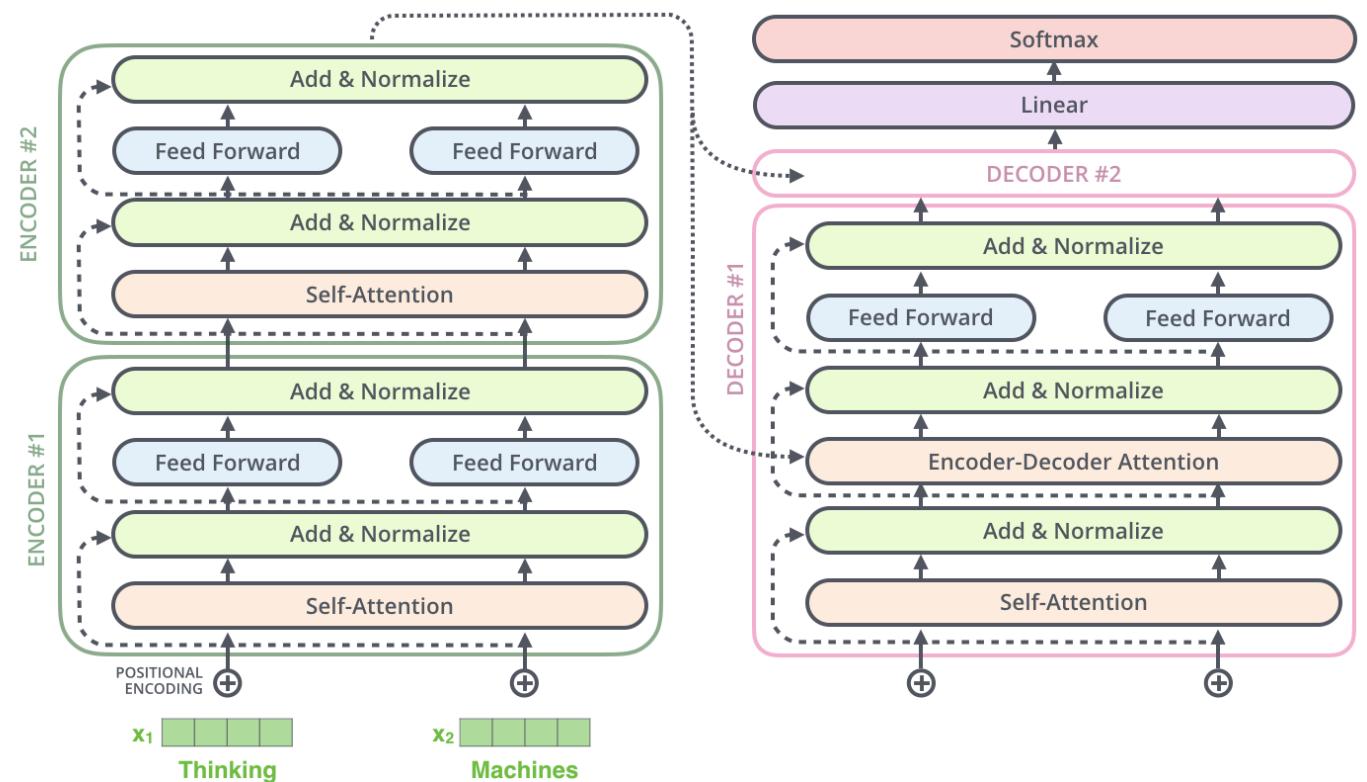
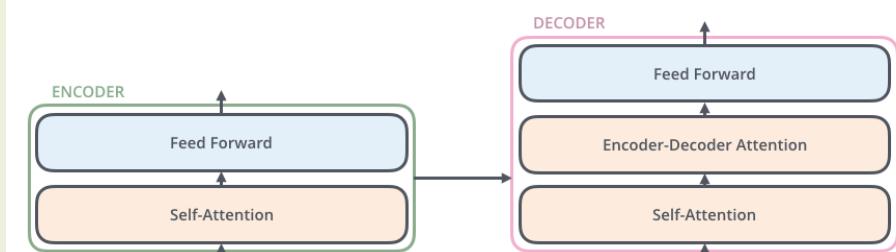


Illustration of Encoder-Decoder

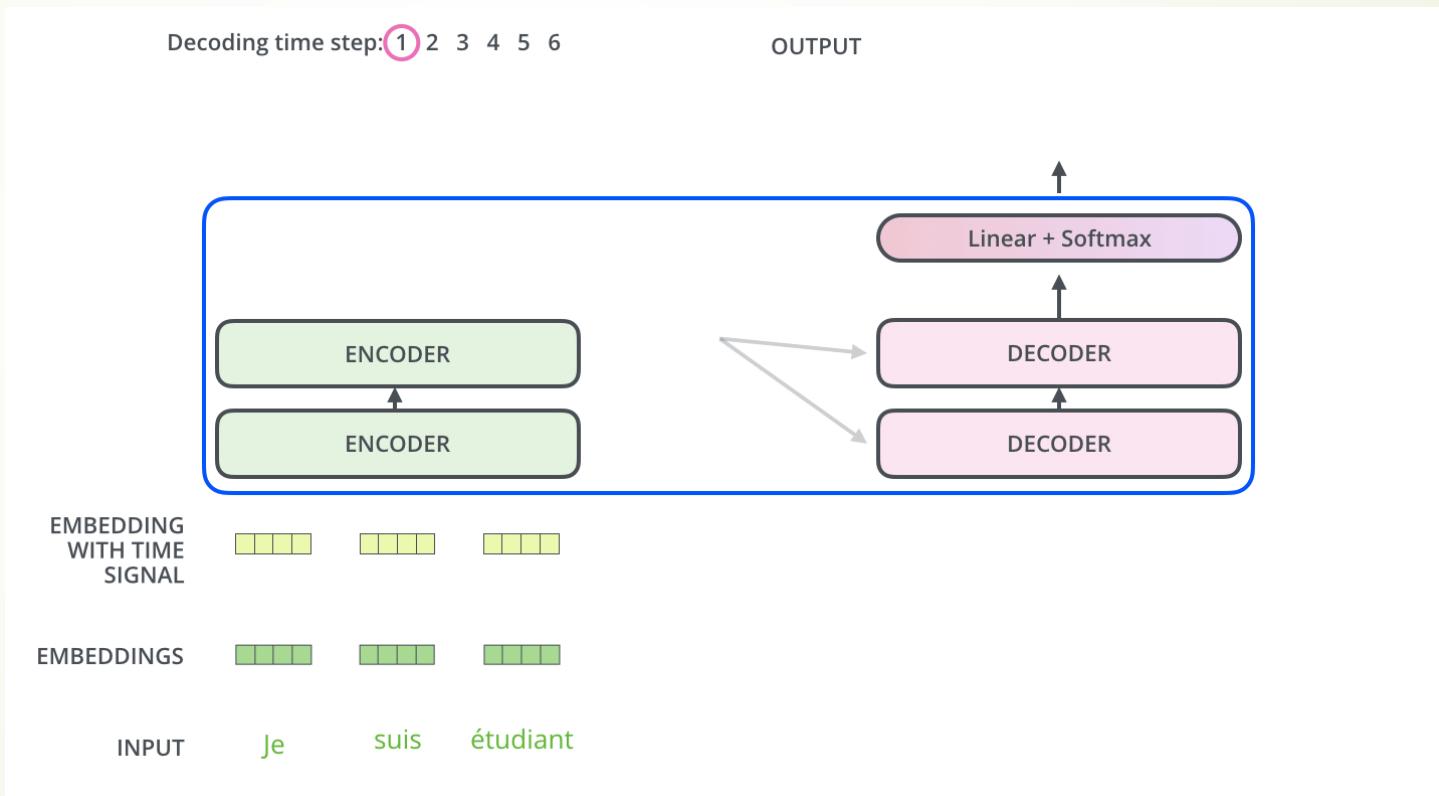
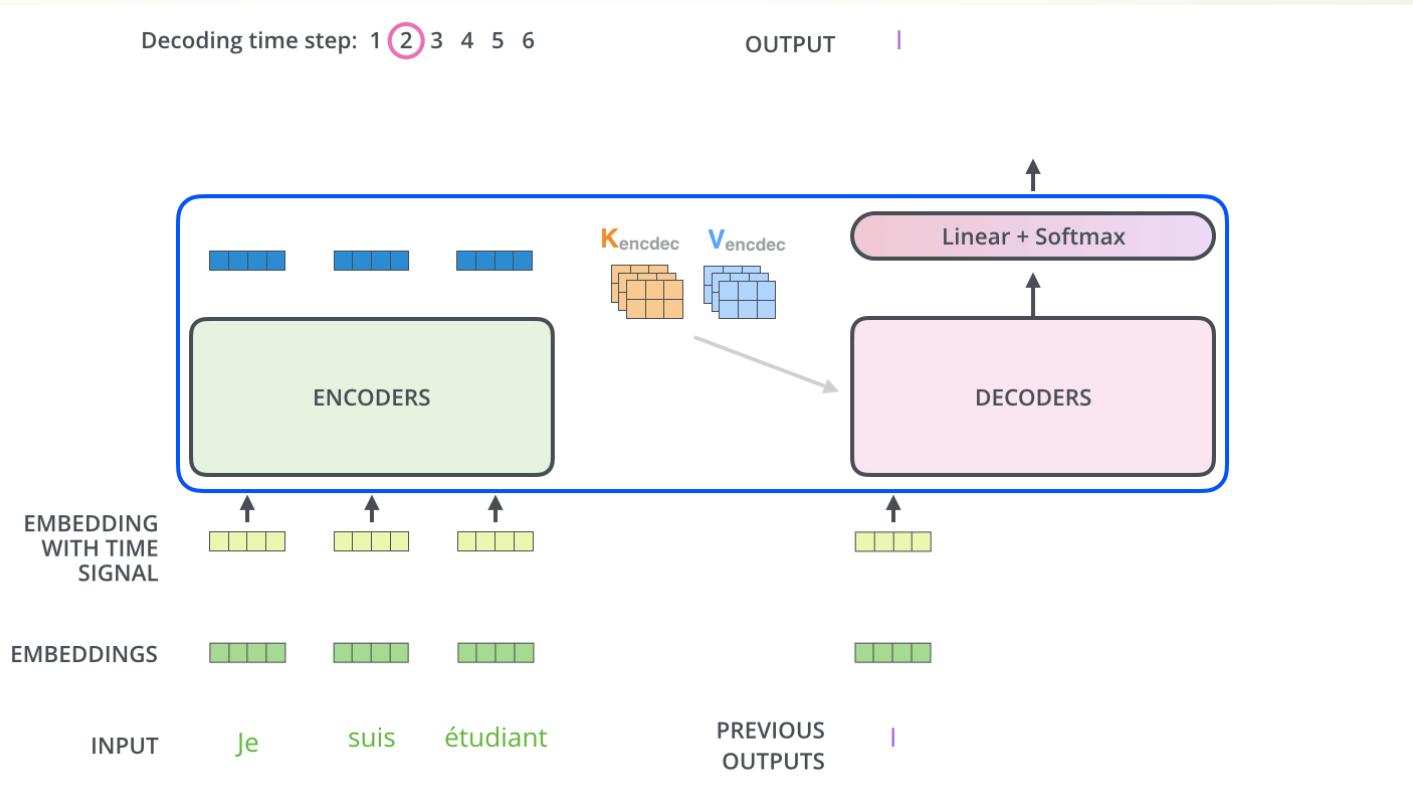
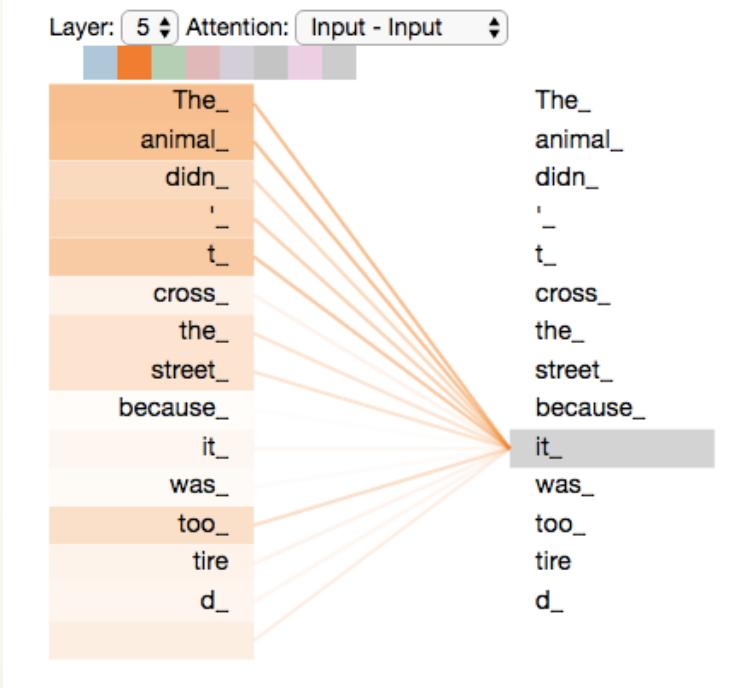


Illustration of Encoder-Decoder

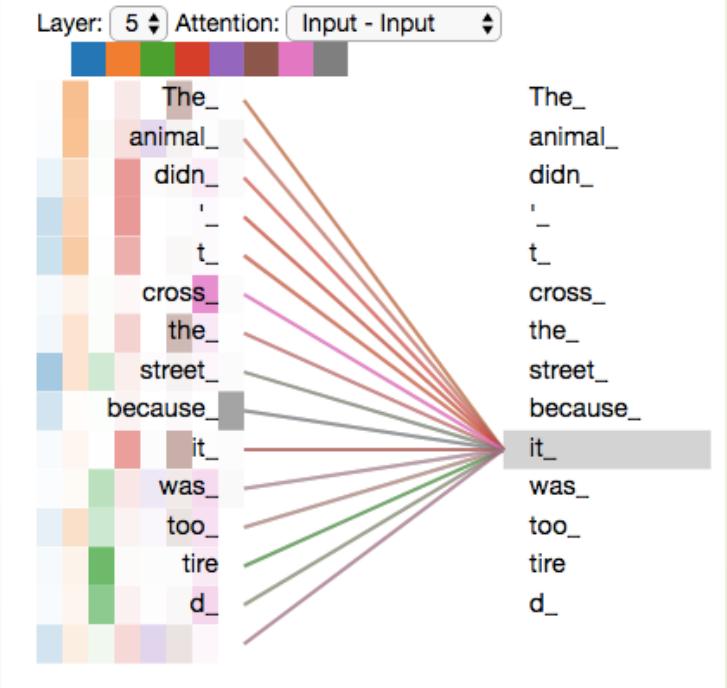


Attention Visualization

Head 2 (yellow) only



8 heads mixture



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 W

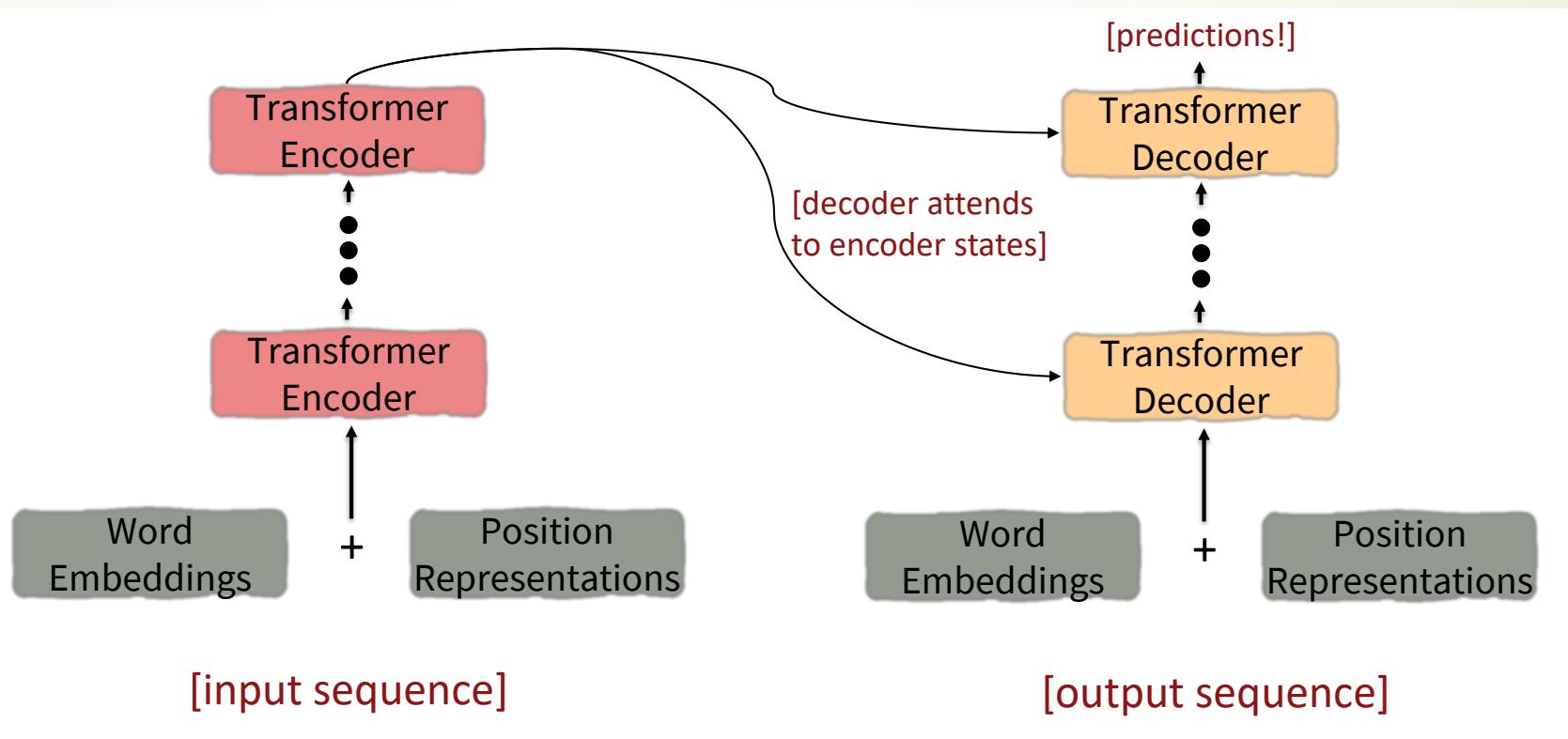
LSTM

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 W

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

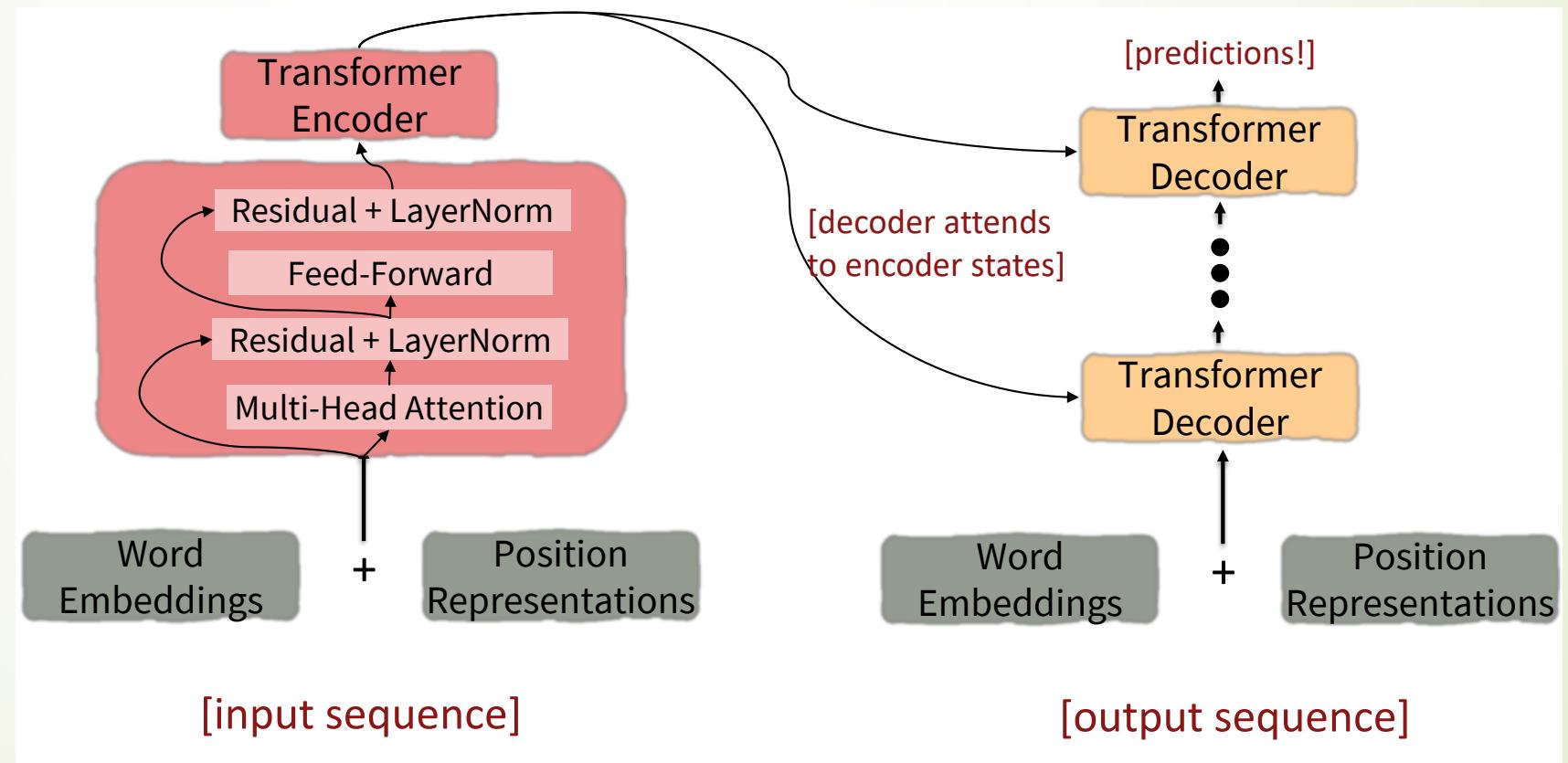
- ▶ Looking back at the whole model



The Transformer Encoder-Decoder

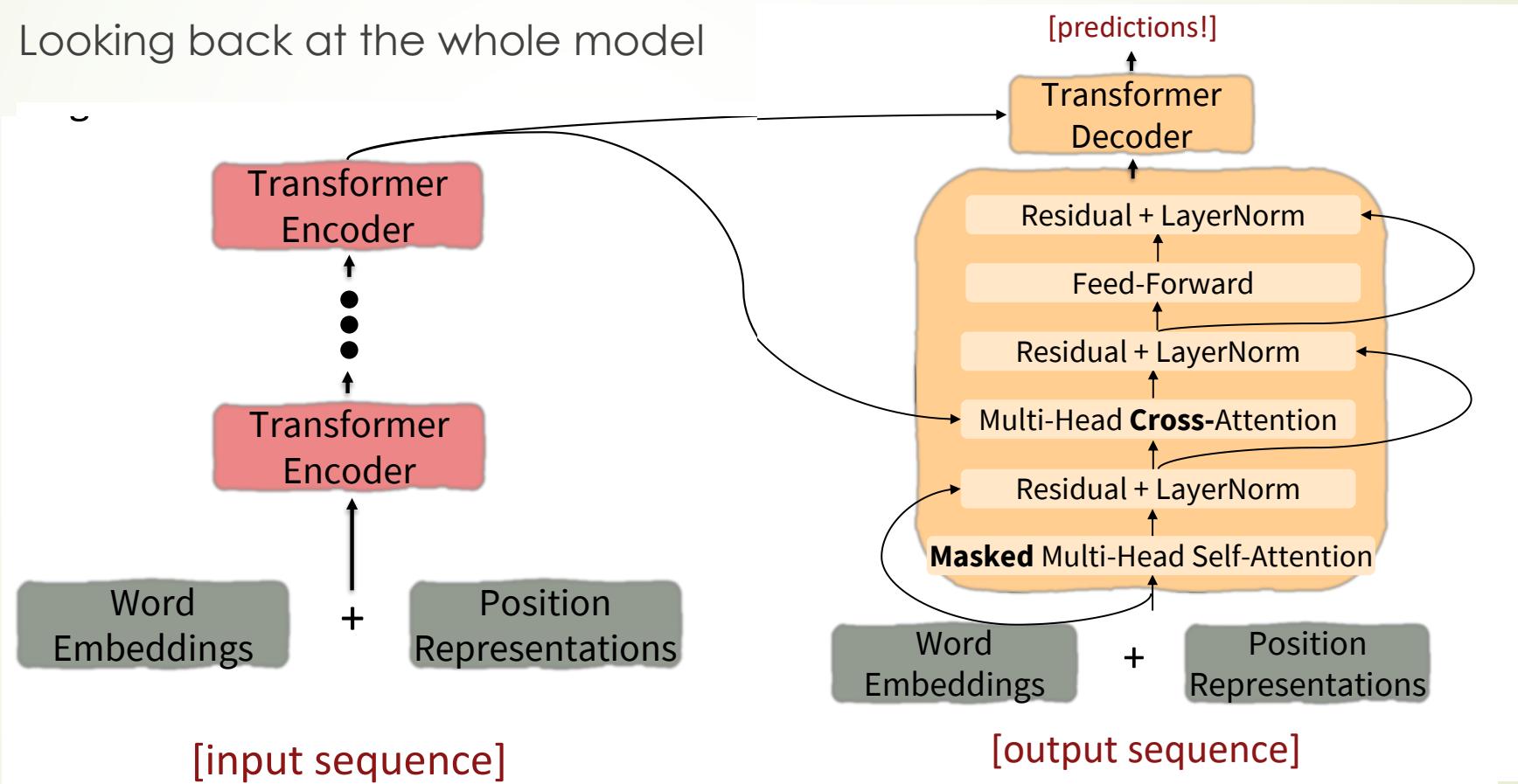
[Vaswani et al. 2017]

- ▶ Looking back at the whole model



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

- ▶ Looking back at the whole model

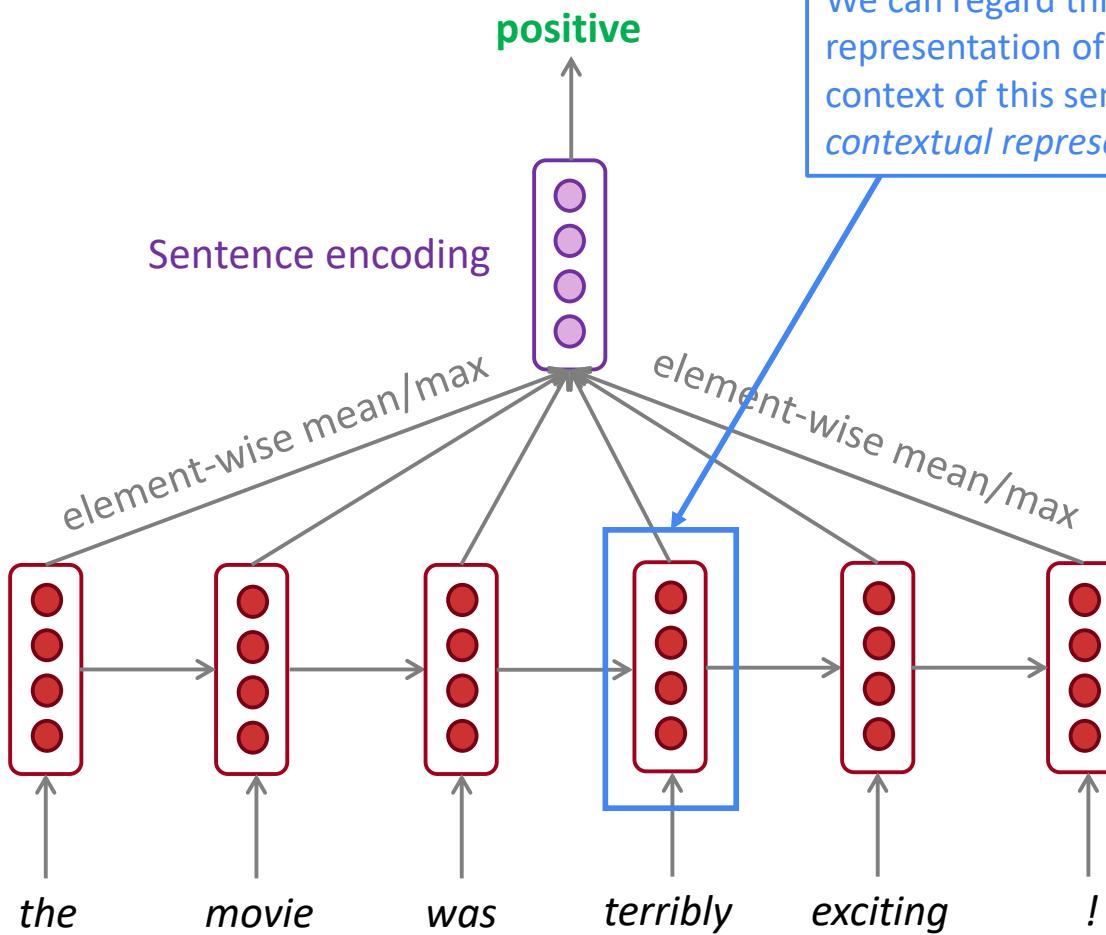


The background features a minimalist design with a vertical brown bar on the left. A large, solid red arrow points from the bottom left towards the center. Overlaid on the arrow are several thin, light gray organic lines that curve and intersect.

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*?

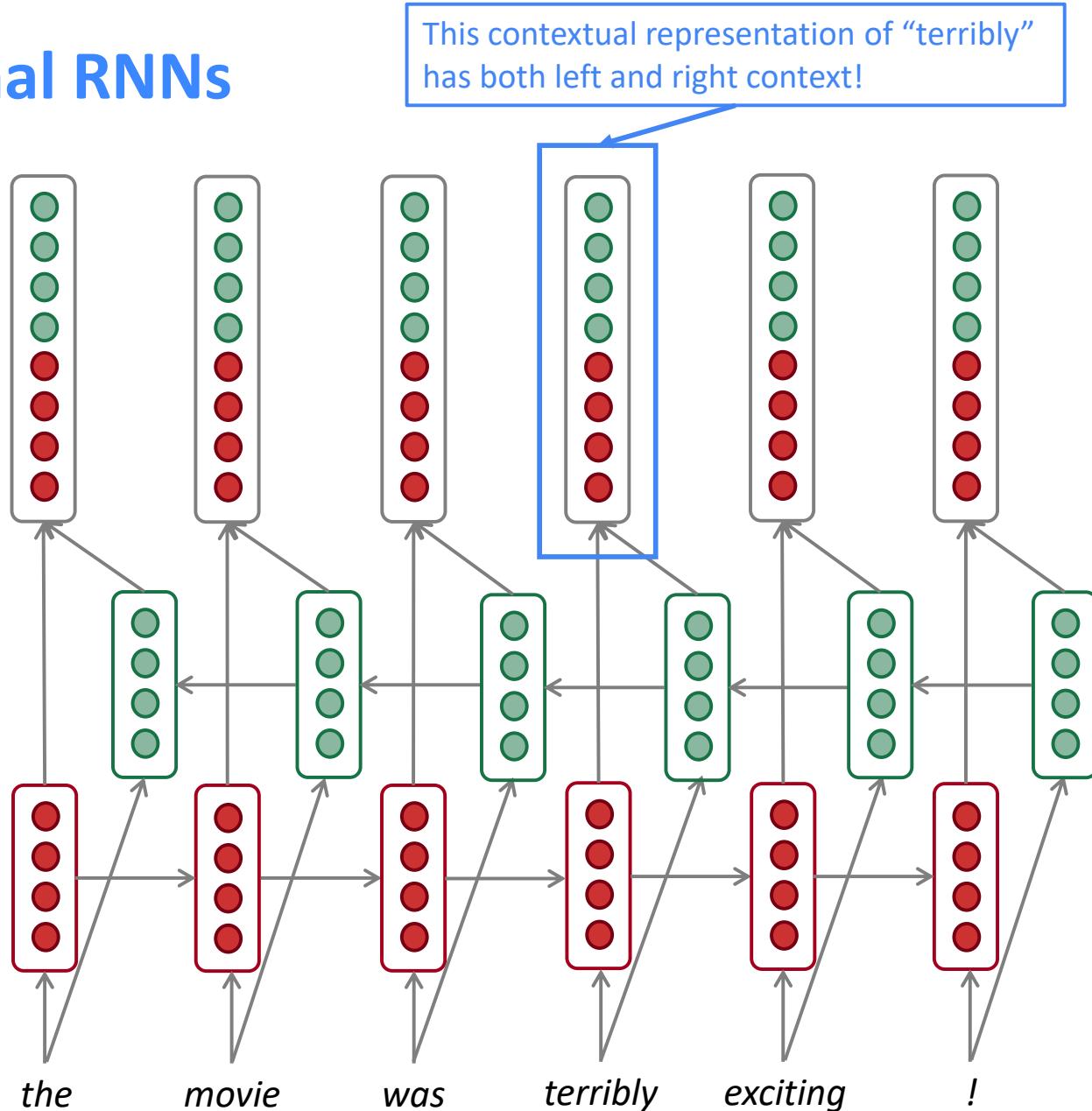
In this example, “*exciting*” is in the right context and this modifies the meaning of “*terribly*” (from negative to positive)

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

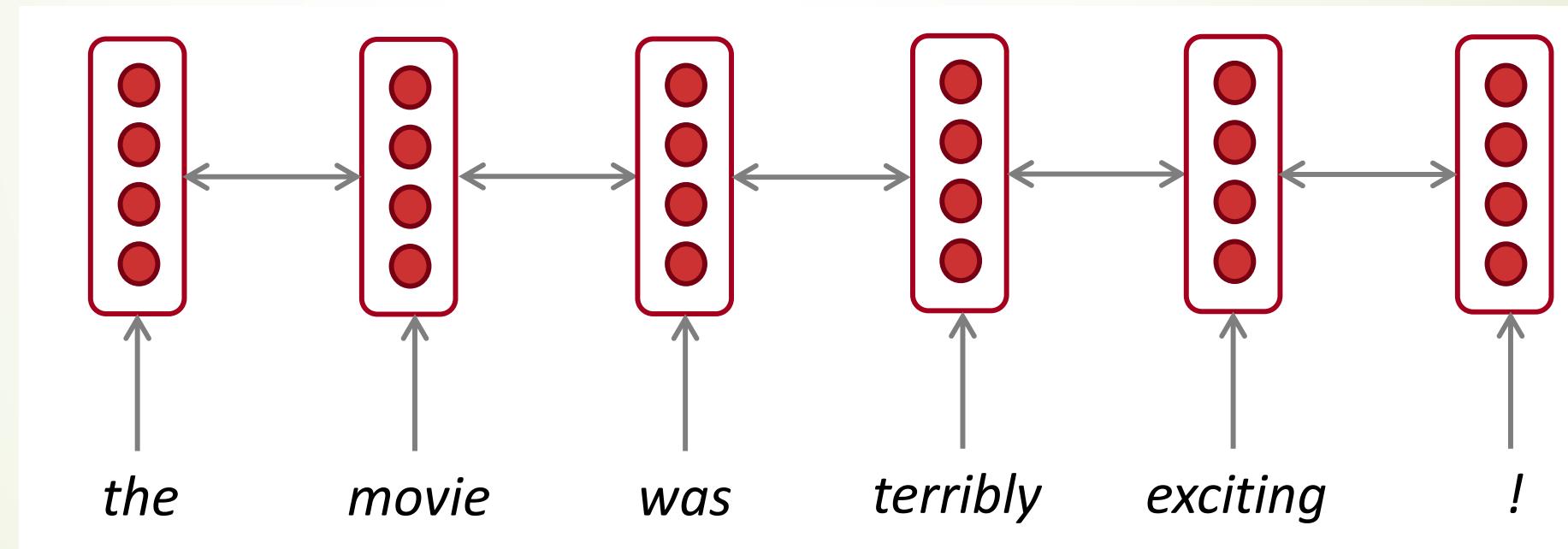
$$\mathbf{h}^{(t)} = [\vec{h}^{(t)}; \overleftarrow{h}^{(t)}]$$

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.



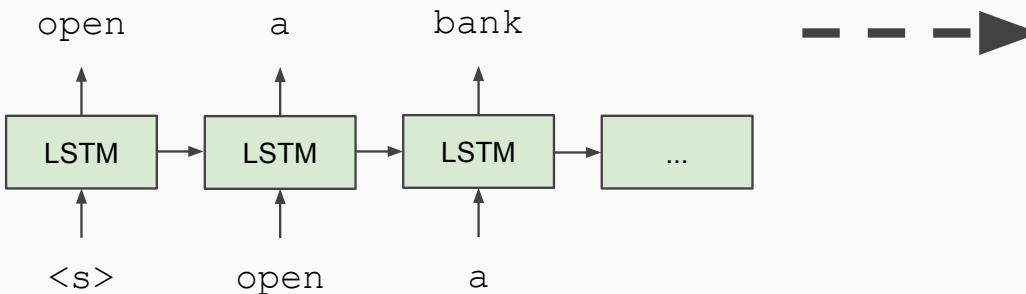
Bidirectional RNNs

- ▶ 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.

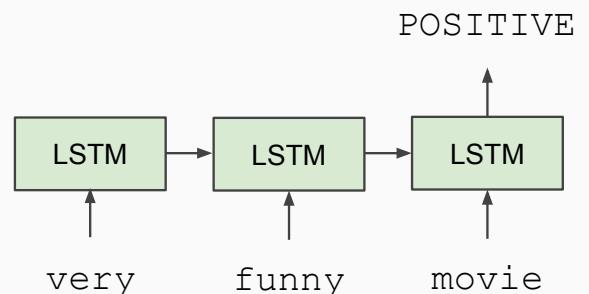
Uni-Direction LSTM

- Semi-Supervised Sequence Learning, Google, 2015

Train LSTM Language Model



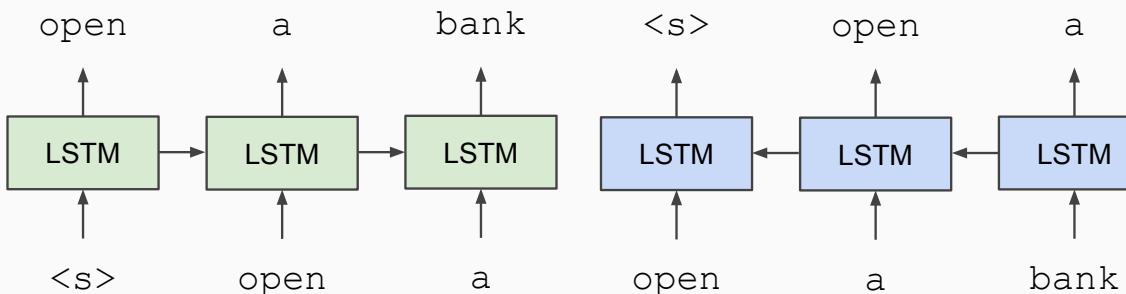
Fine-tune on Classification Task



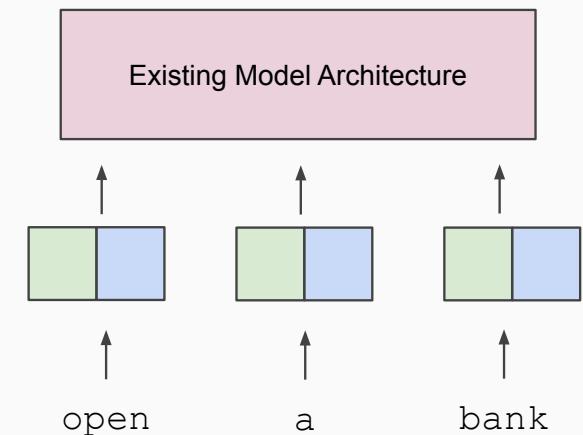
Bi-Direction: ELMo -- Embeddings from Language Models

- ▶ 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



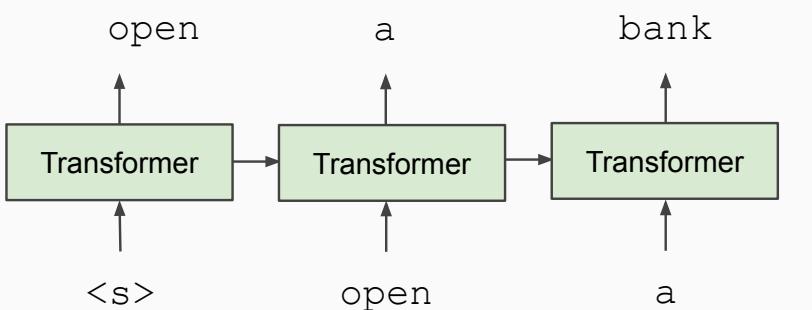
Apply as “Pre-trained Embeddings”



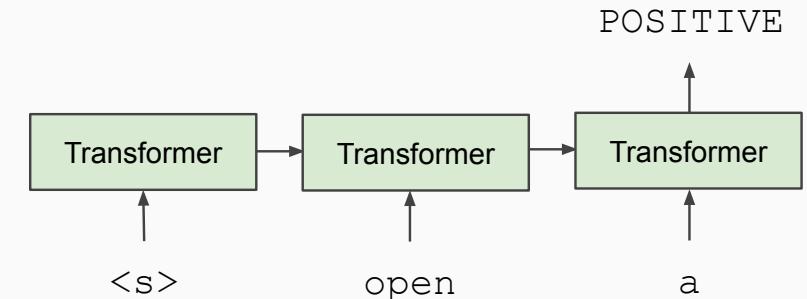
GPT (Generative Pre-Training): uni-directional transformer

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

Train Deep (12-layer) Transformer LM

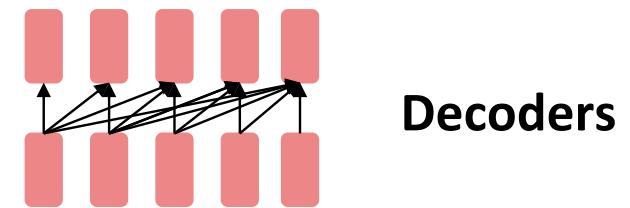


Fine-tune on Classification Task



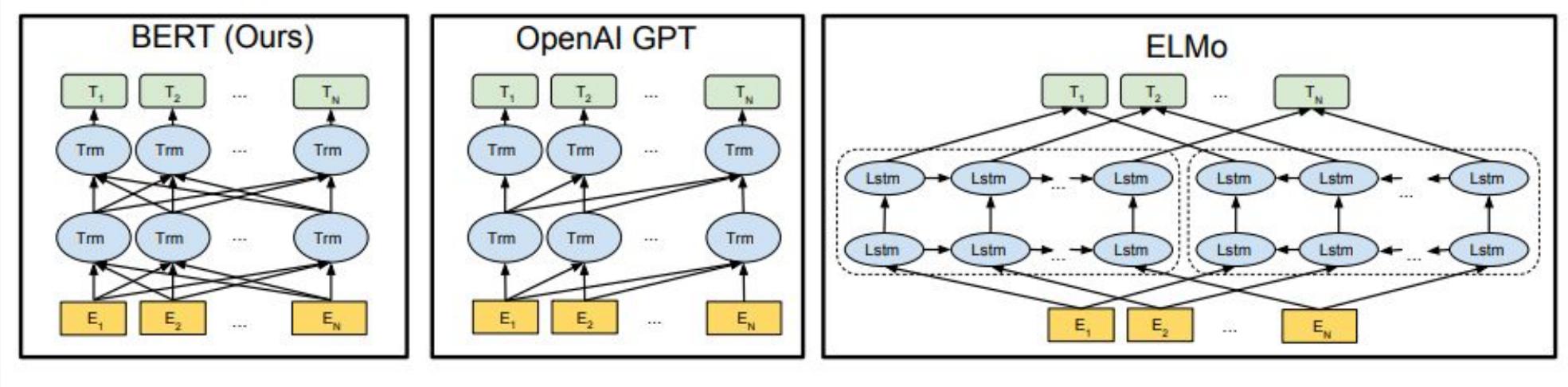
GPT (Generative Pre-Training): uni-directional transformer-decoder

- ▶ 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.



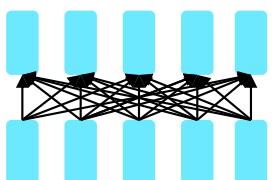
- Language models! What we've seen so far.
- Nice to generate from; can't condition on future words

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 $k = 15\%$

- ▶ 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

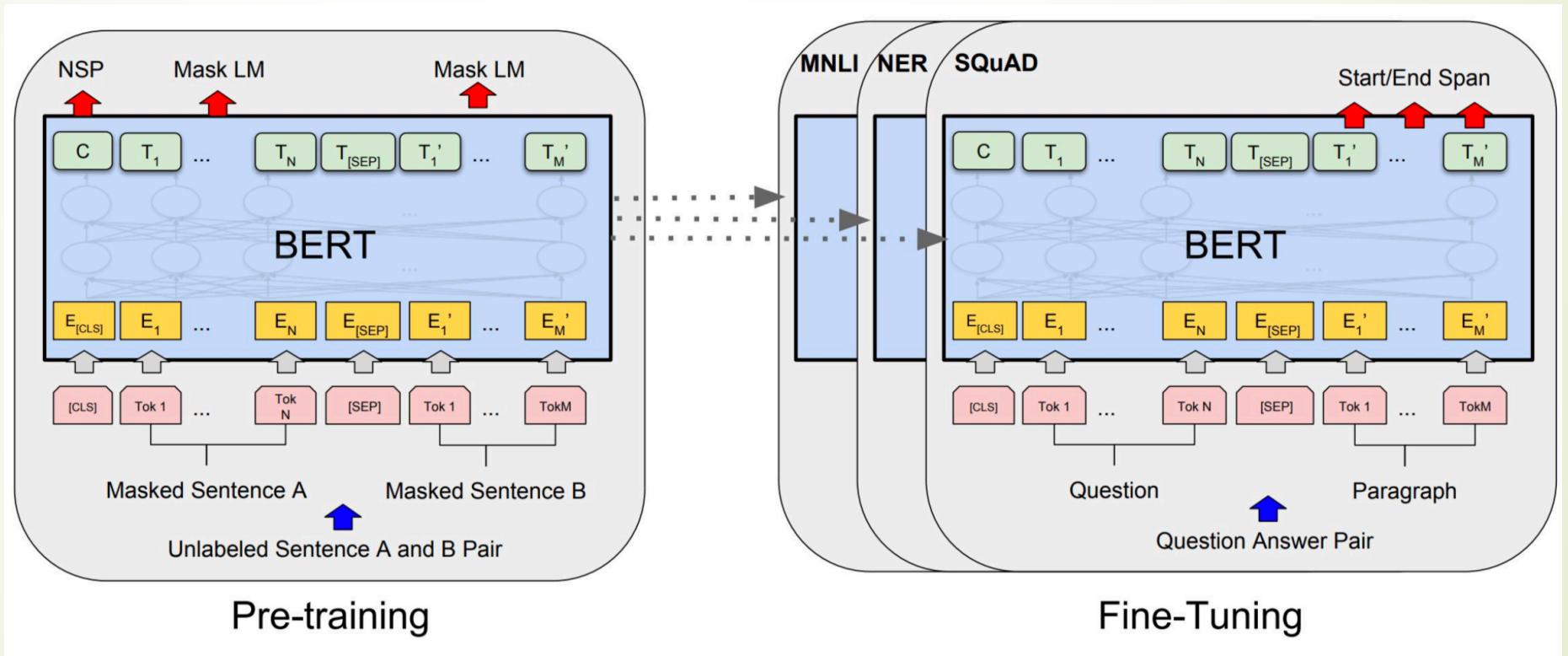
Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	# #ing	[SEP]
Token Embeddings	$E_{[CLS]}$	E_{my}	E_{dog}	E_{is}	E_{cute}	$E_{[SEP]}$	E_{he}	E_{likes}	E_{play}	$E_{##ing}$	$E_{[SEP]}$
Segment Embeddings	E_A	E_A	E_A	E_A	E_A	E_A	E_B	E_B	E_B	E_B	E_B
Position Embeddings	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8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

Training

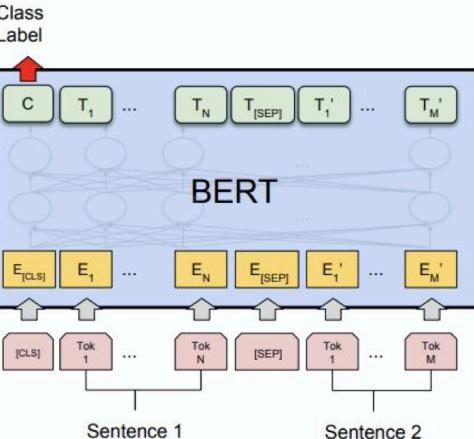
- ▶ 2 models released:
 - ▶ BERT-Base: 12-layer, 768-hidden, 12-head, 110 million params.
 - ▶ BERT-Large: 24-layer, 1024-hidden, 16-head, 340 million params.
- ▶ Training Data:
 - ▶ BookCorpus (800M words)
 - ▶ English Wikipedia (2.5B words)
- ▶ Batch Size: 131,072 words
 - ▶ (1024 sequences * 128 length 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

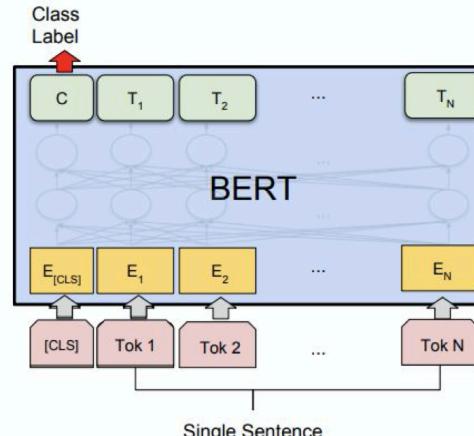
- ▶ Simply learn a classifier built on the top layer for each task that you fine tune for



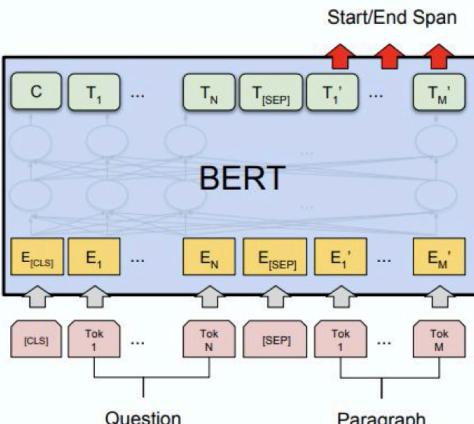
BERT model fine tuning



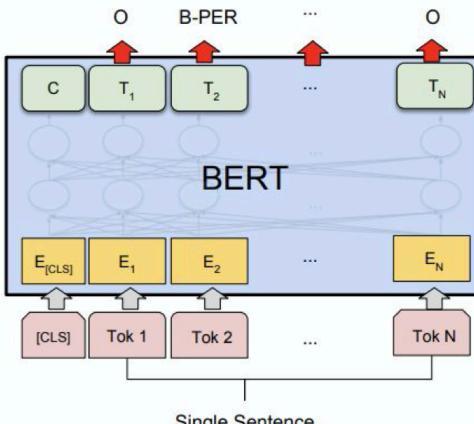
(a) Sentence Pair Classification Tasks:
MNLI, QQP, QNLI, STS-B, MRPC,
RTE, SWAG



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

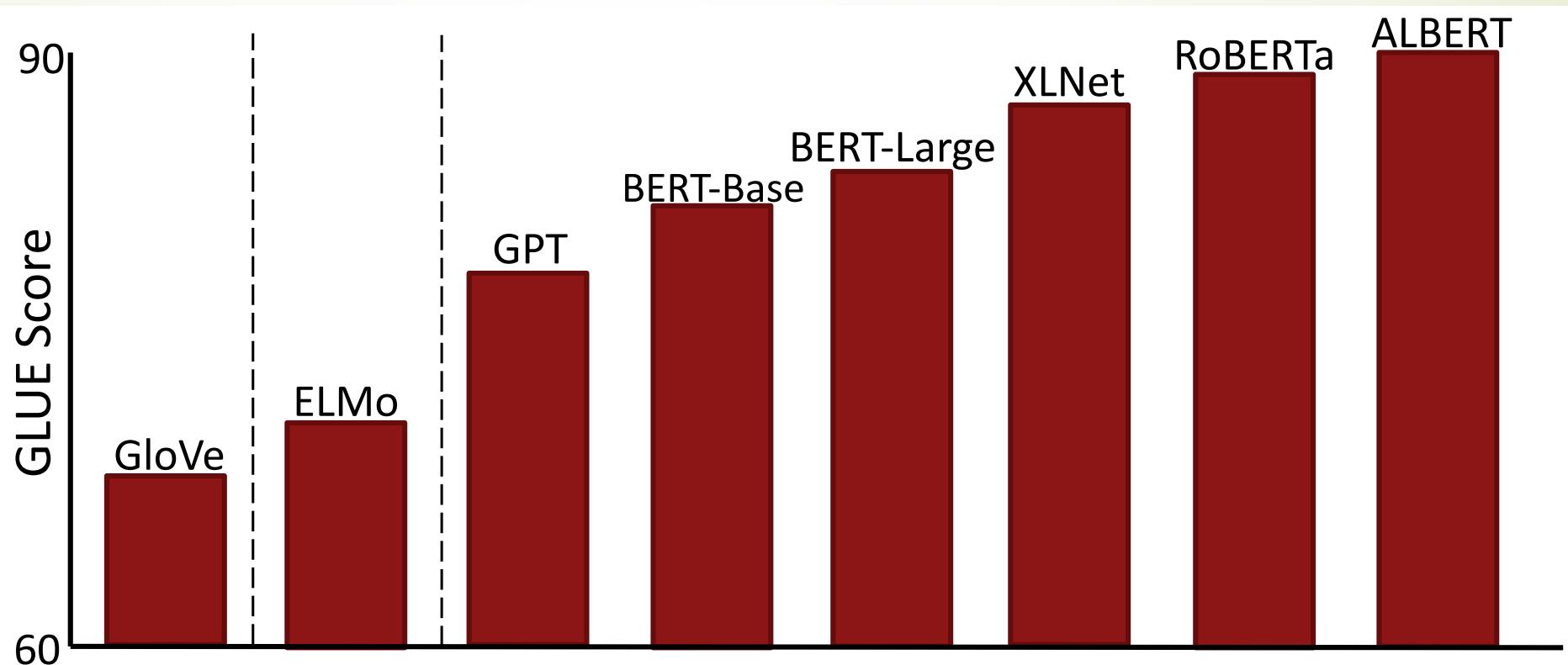


(c) Question Answering Tasks:
SQuAD v1.1



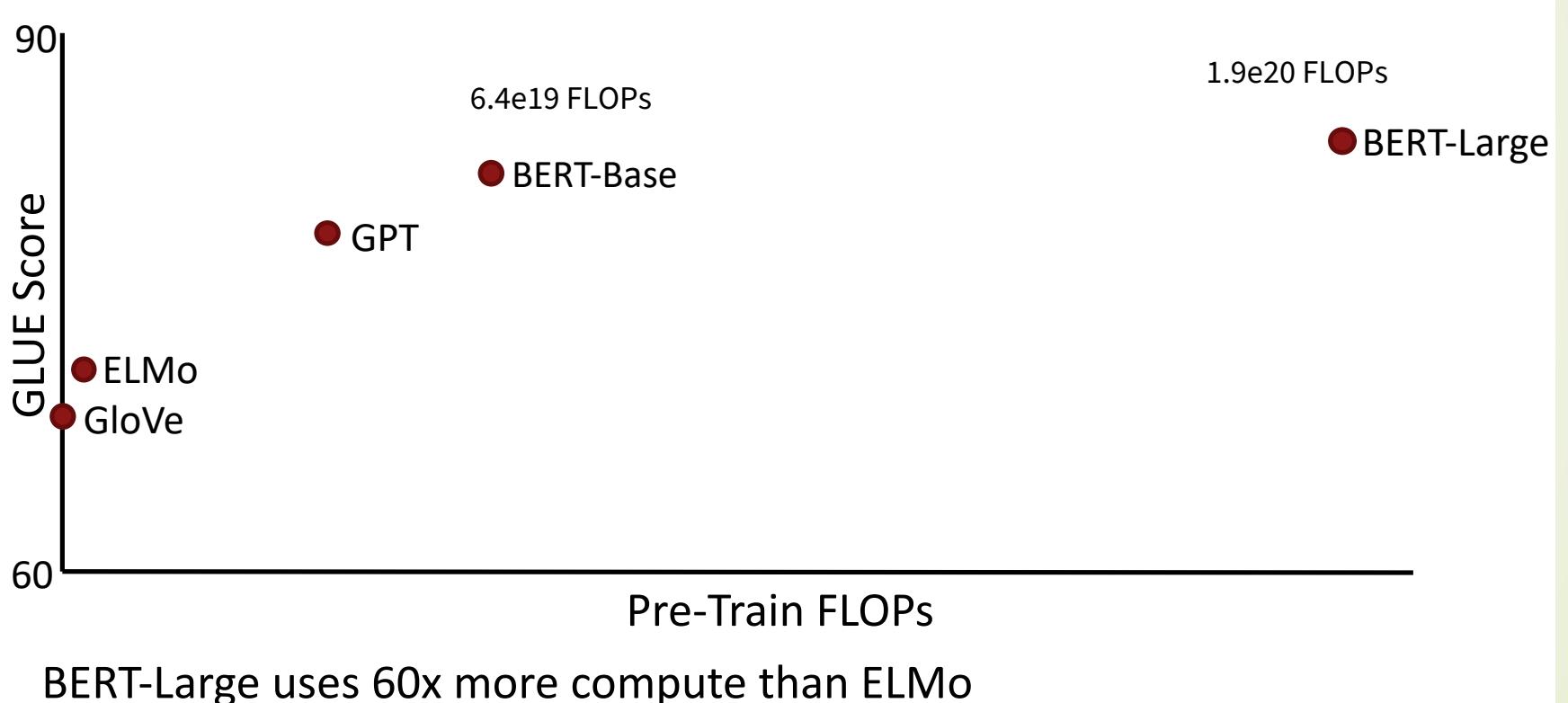
(d) Single Sentence Tagging Tasks:
CoNLL-2003 NER

Rapid Progress for Pre-training (GLUE Benchmark)

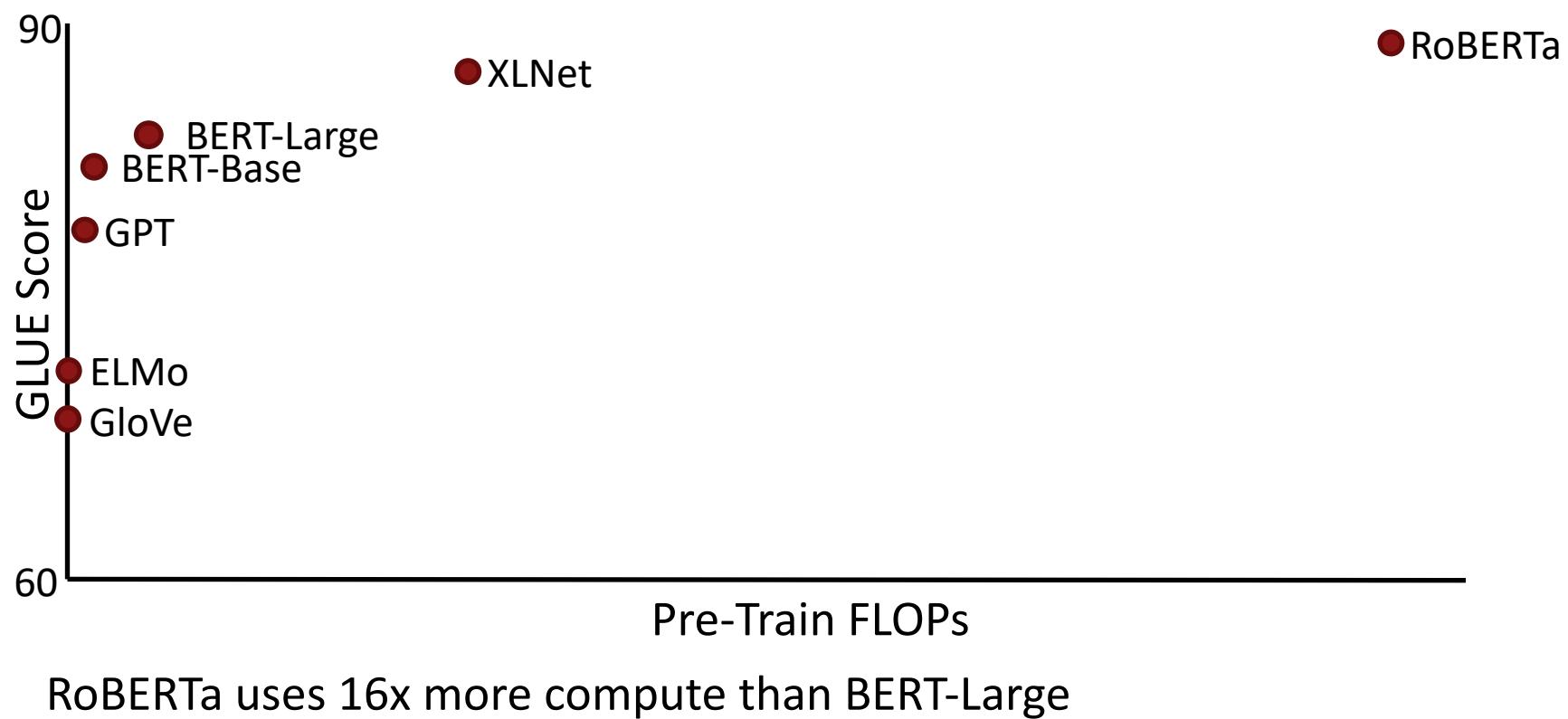


Over 3x reduction in error in 2 years, “superhuman” performance

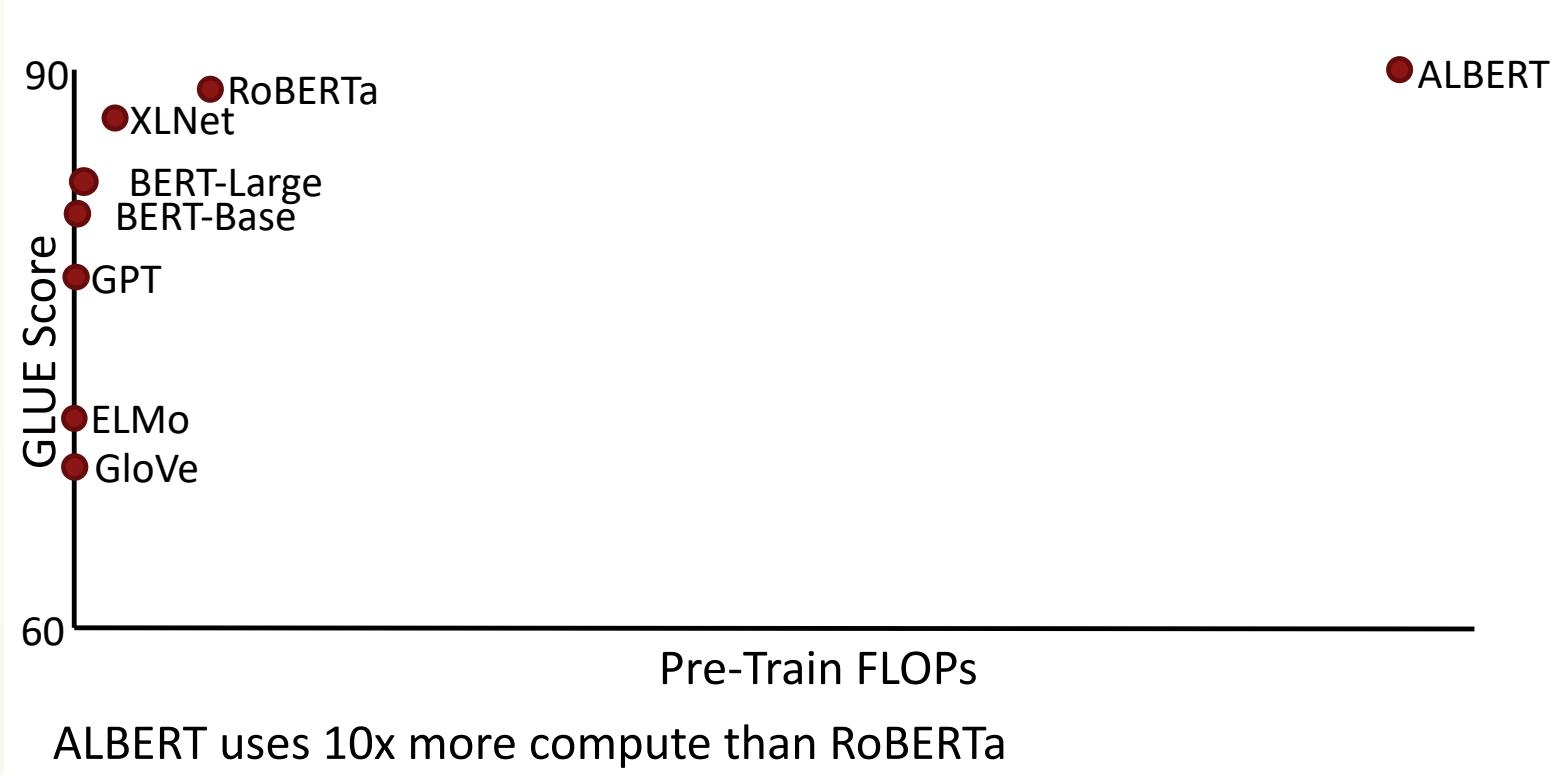
But let's change the x-axis to computational cost...



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

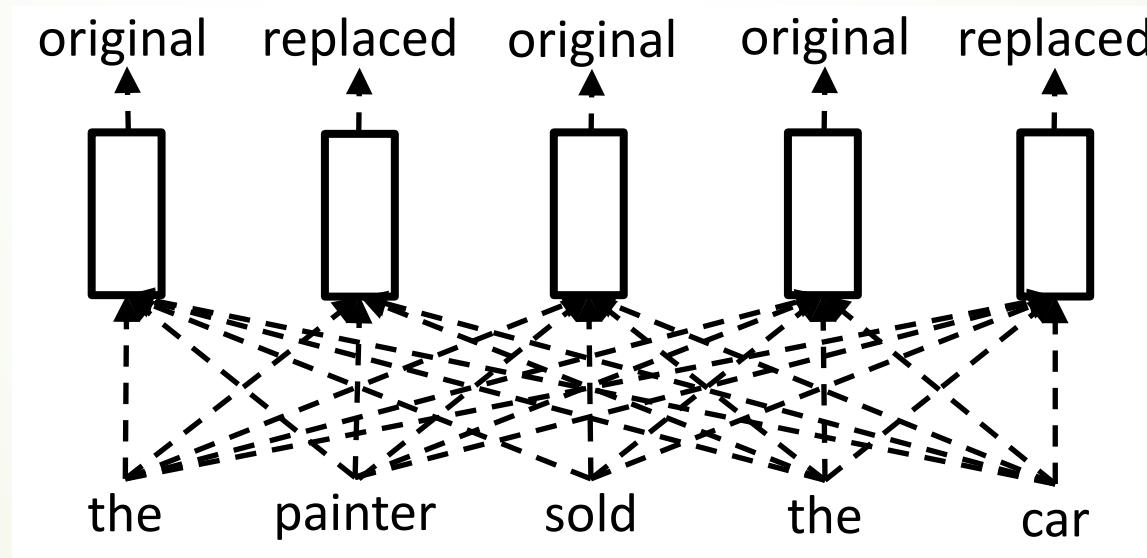


More compute, more better?

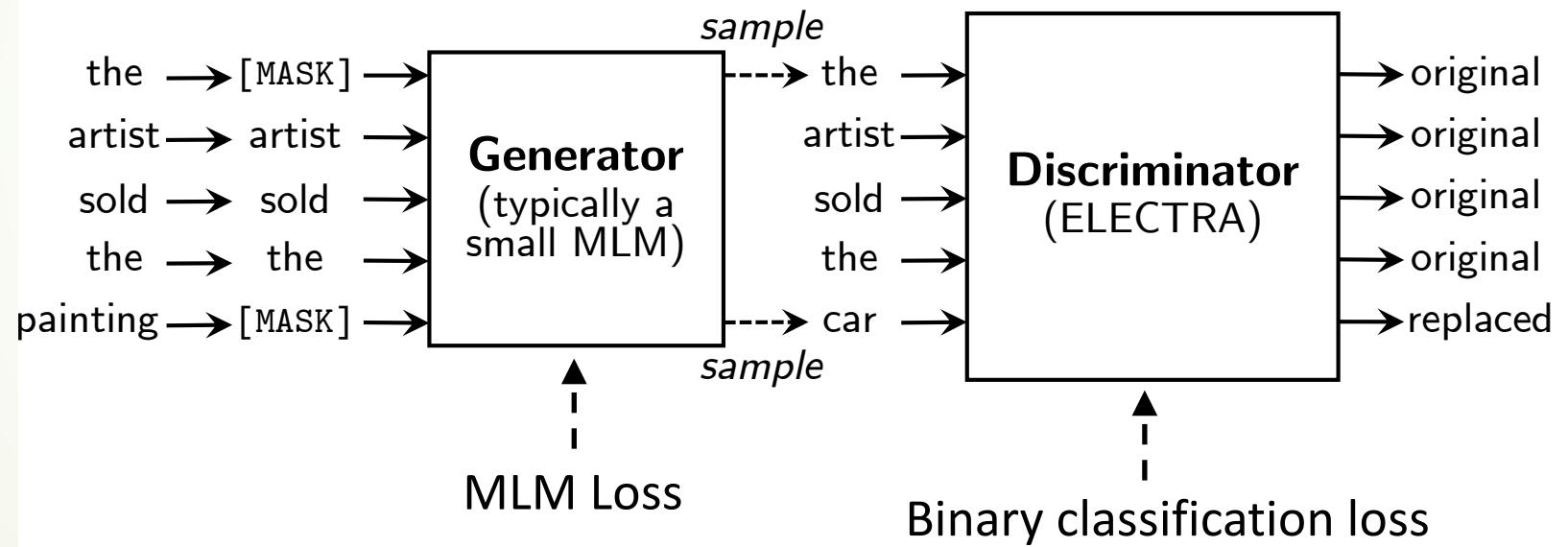


ELECTRA: “Efficiently Learning an Encoder to Classify Token Replacements Accurately”

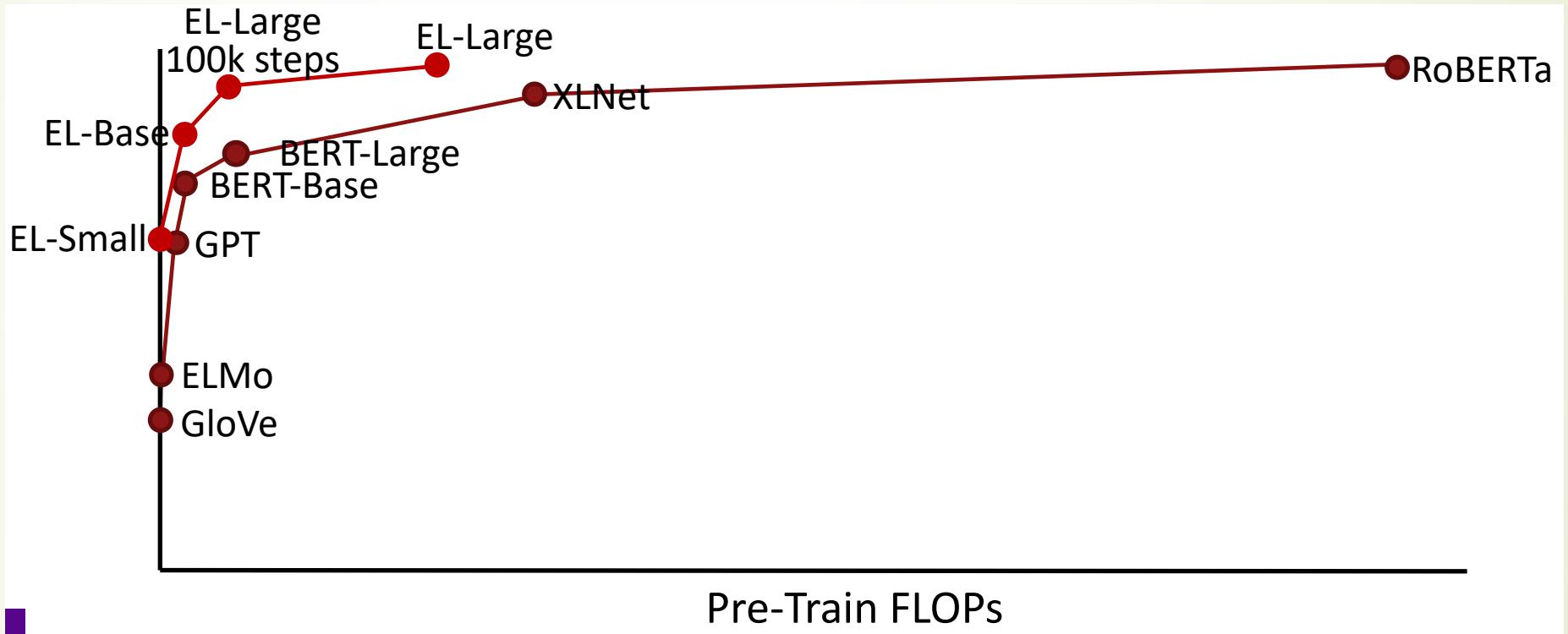
- ▶ Clark, Luong, Le, and Manning, ICLR 2020.
<https://openreview.net/pdf?id=r1xMH1BtvB>
- ▶ Bidirectional model but learn from all tokens



Generating Replacements

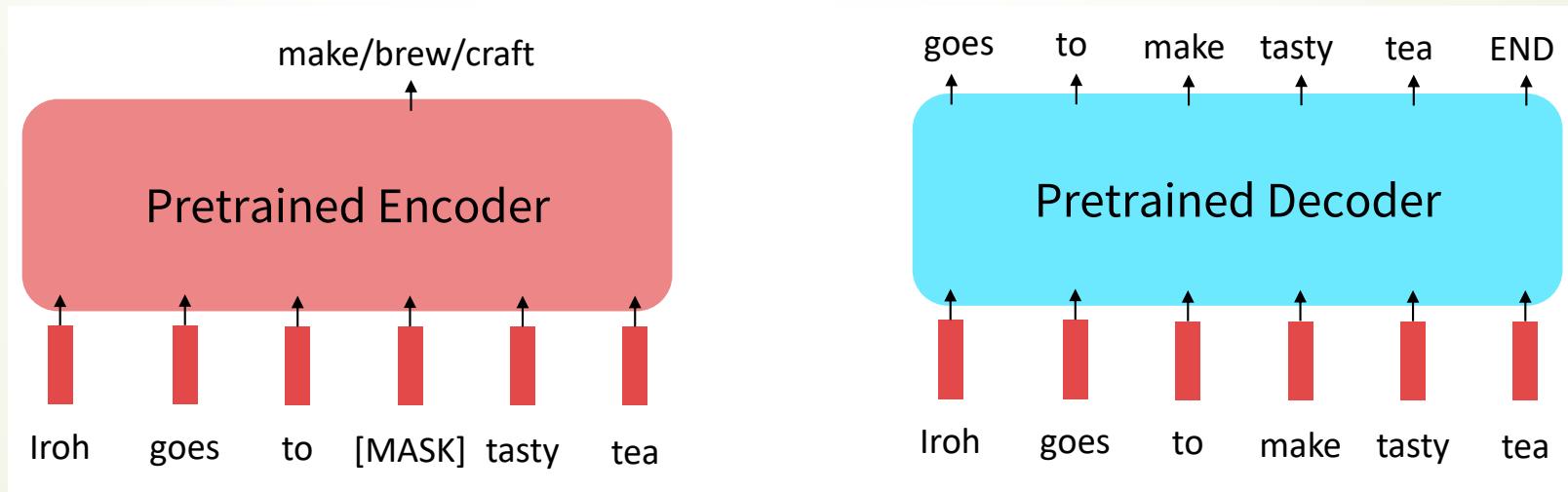


Results: GLUE Score vs Compute



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.

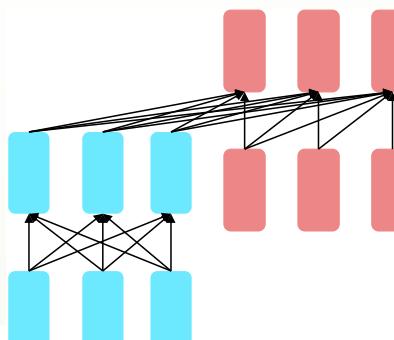


Pretraining encoders-decoders: T5

- ▶ 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.

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.



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 is the canonical example of this. The largest T5 model had 11 billion parameters.
- ▶ **GPT-3 has 175 billion parameters.**

Thank you!

