Attention, Transformer, and BERT

Yuan YAO

HKUST

Summary

- We have shown:
  - CNN Architectures: LeNet5, Alexnet, VGG, GoogleNet, Resnet
  - Recurrent Neural Networks and LSTM (GRU etc.)
- 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

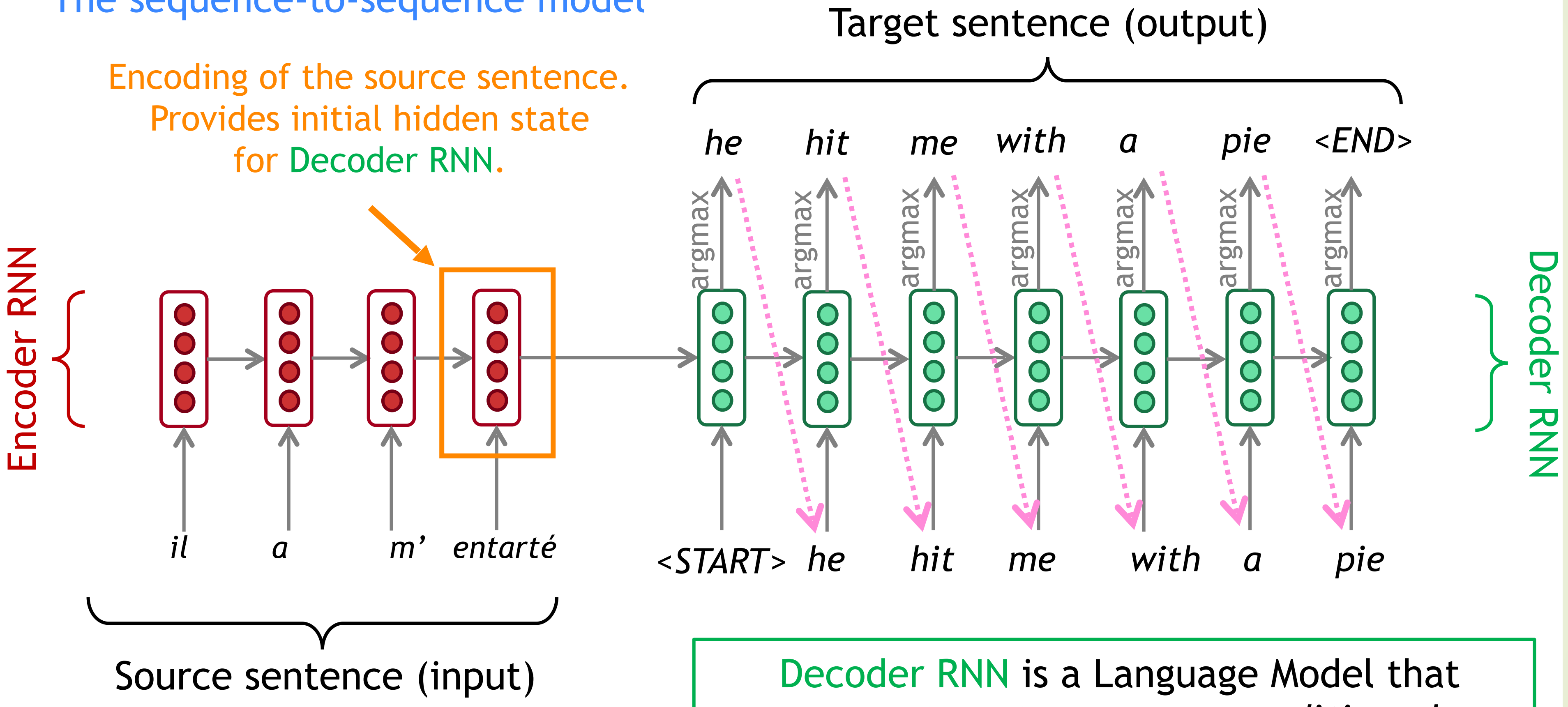
  - **Source:** "Findings of the 2016 Conference on Machine Translation (WMT16)", Bojar et al. 2016, http://www.statmt.org/wmt16/pdf/W16-2301.pdf
  - **Source:** "Findings of the 2018 Conference on Machine Translation (WMT18)", Bojar et al. 2018, http://www.statmt.org/wmt18/pdf/WMT028.pdf

Neural Machine Translation

Machine Translation using Neural Networks

Neural Machine Translation (NMT)

The sequence-to-sequence model



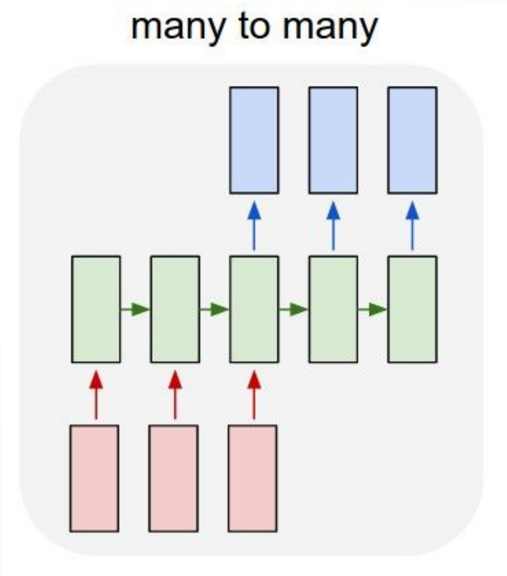
Encoder RNN produces an *encoding* of the source sentence.

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

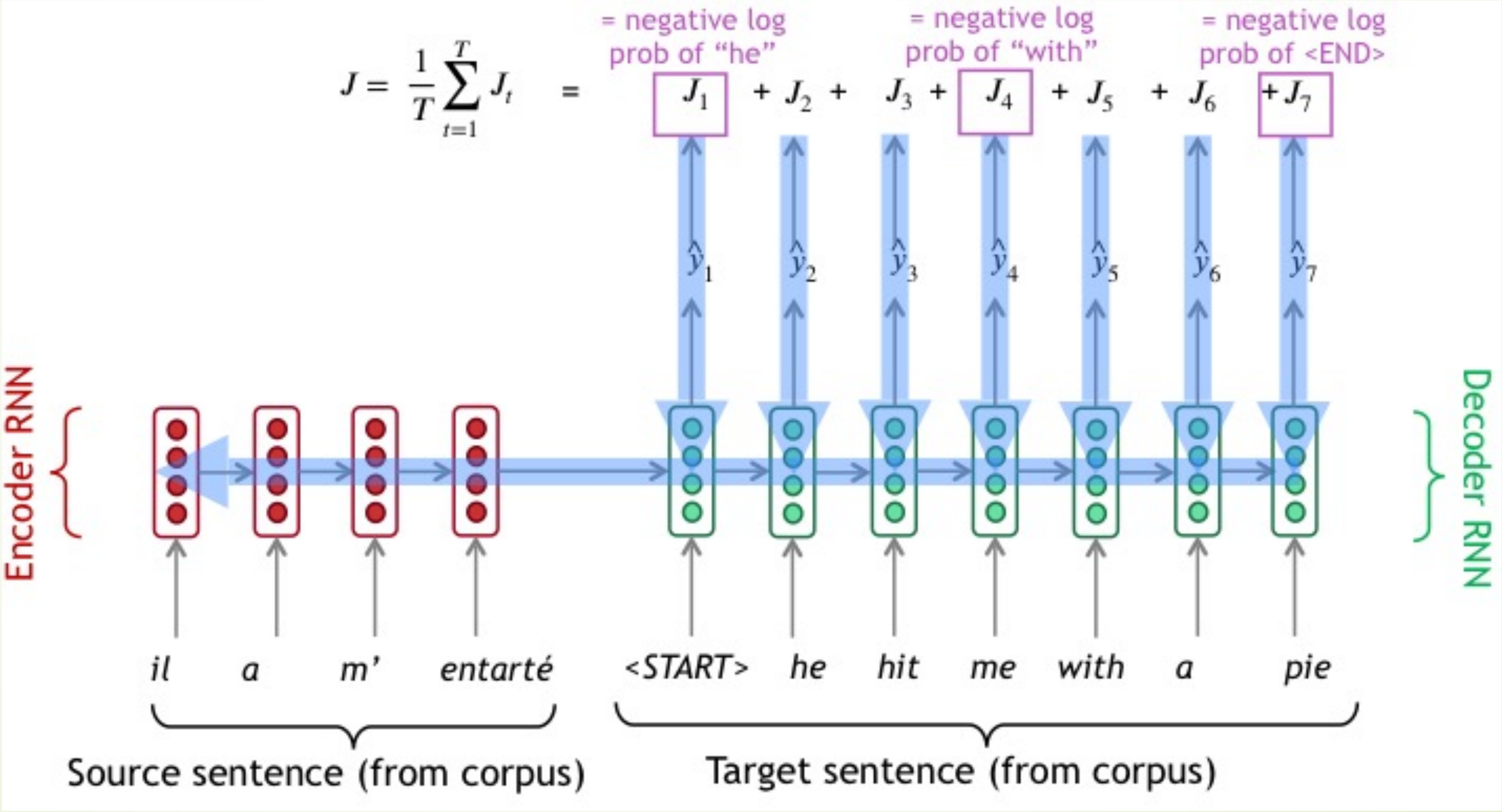
Note: This diagram shows **test time** behavior: decoder output is fed in ····› as next step's input

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)

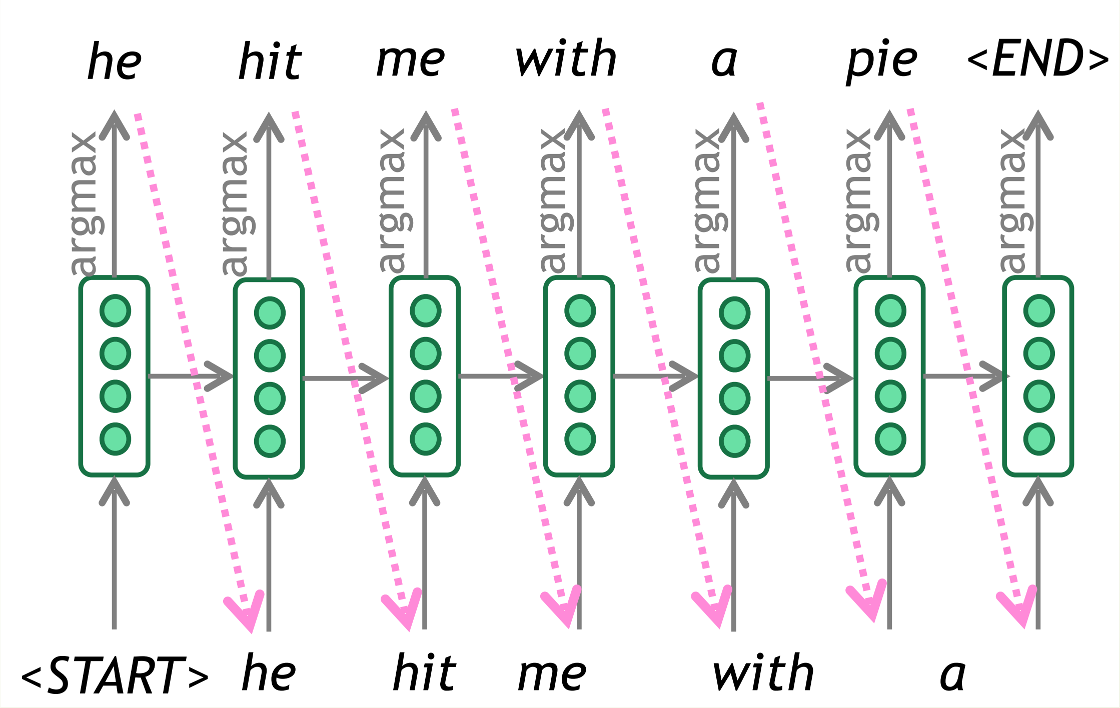


Training a NMT system by BP



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

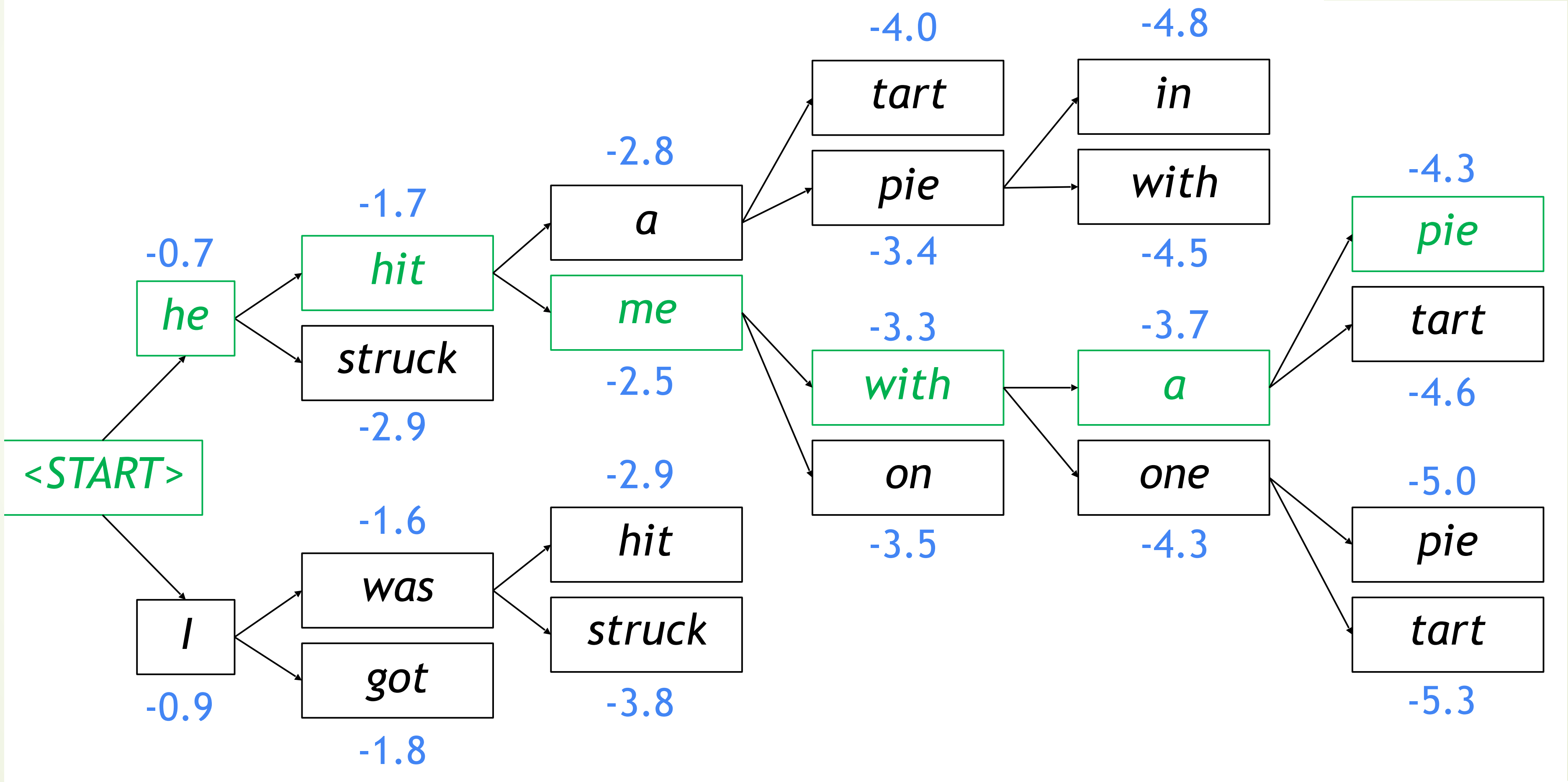
- Core idea: On each step of decoder, keep track of the ***k* most probable** partial translations (which we call *hypotheses*)
  - *k* is the beam size (in practice around 5 to 10)
- A hypothesis (y(1),…,y(t)) has a score which is its log probability:

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

  - 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!

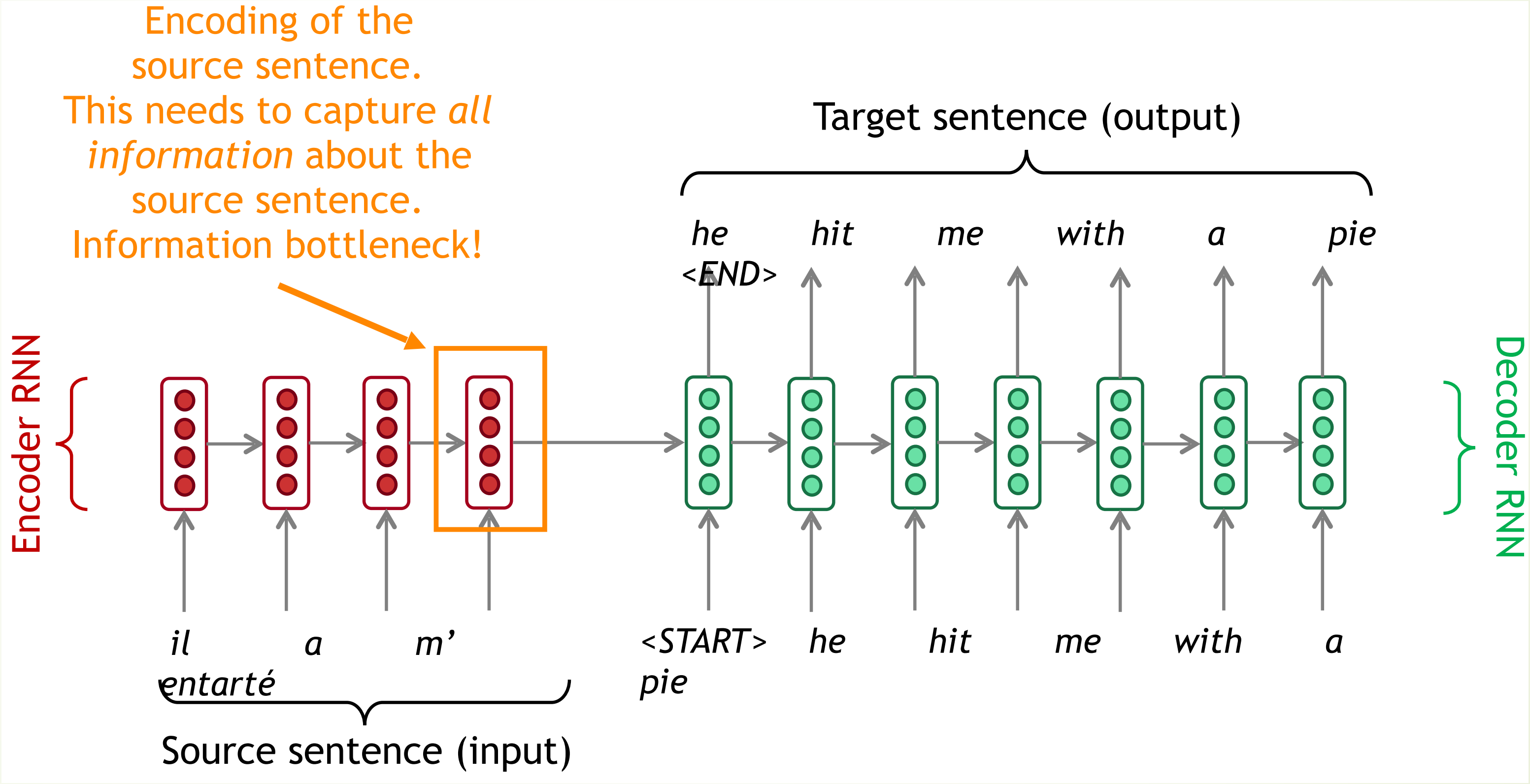
Beam search decoding example:

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



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

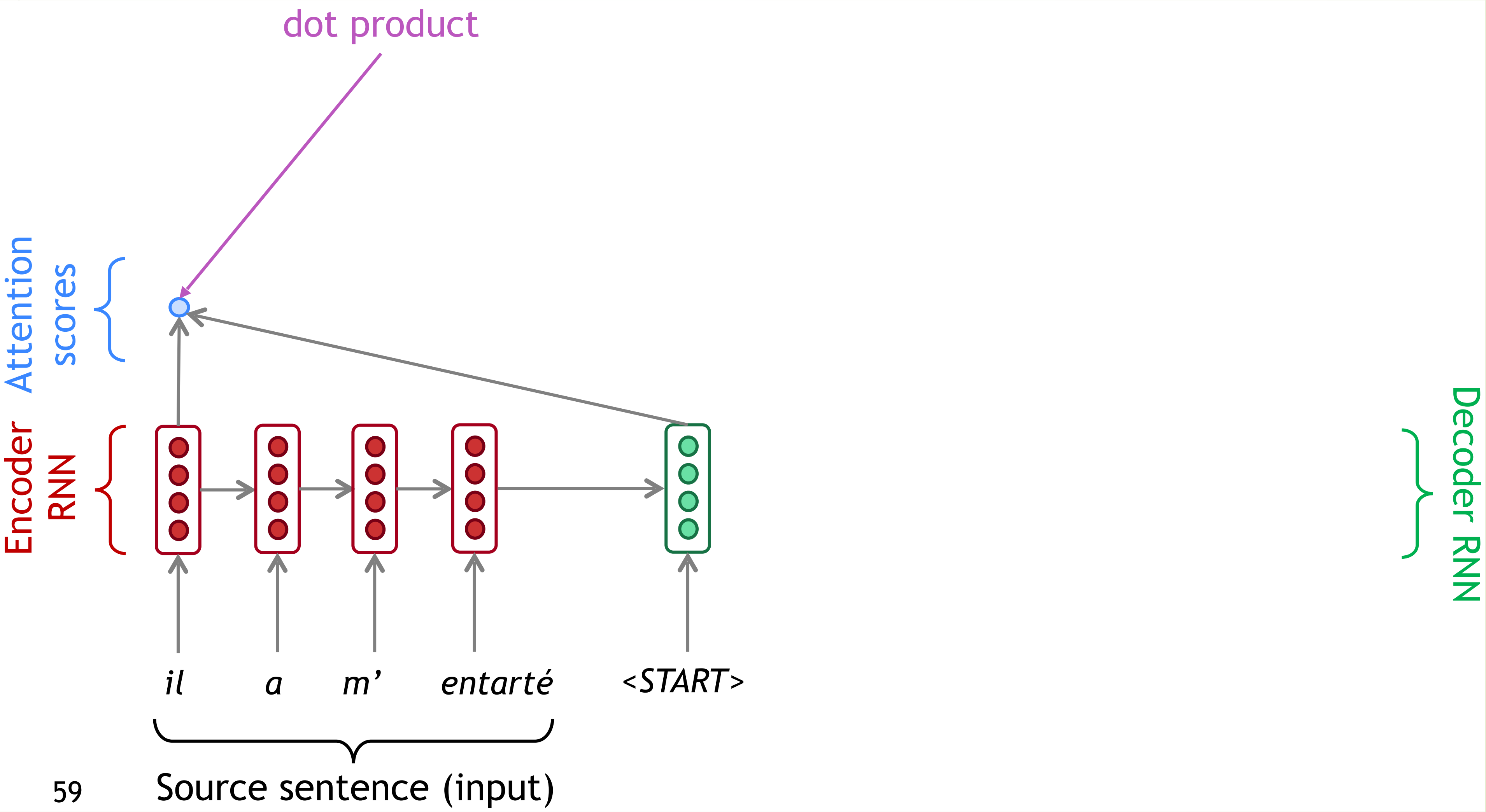
Sequence-to-sequence: the bottleneck problem

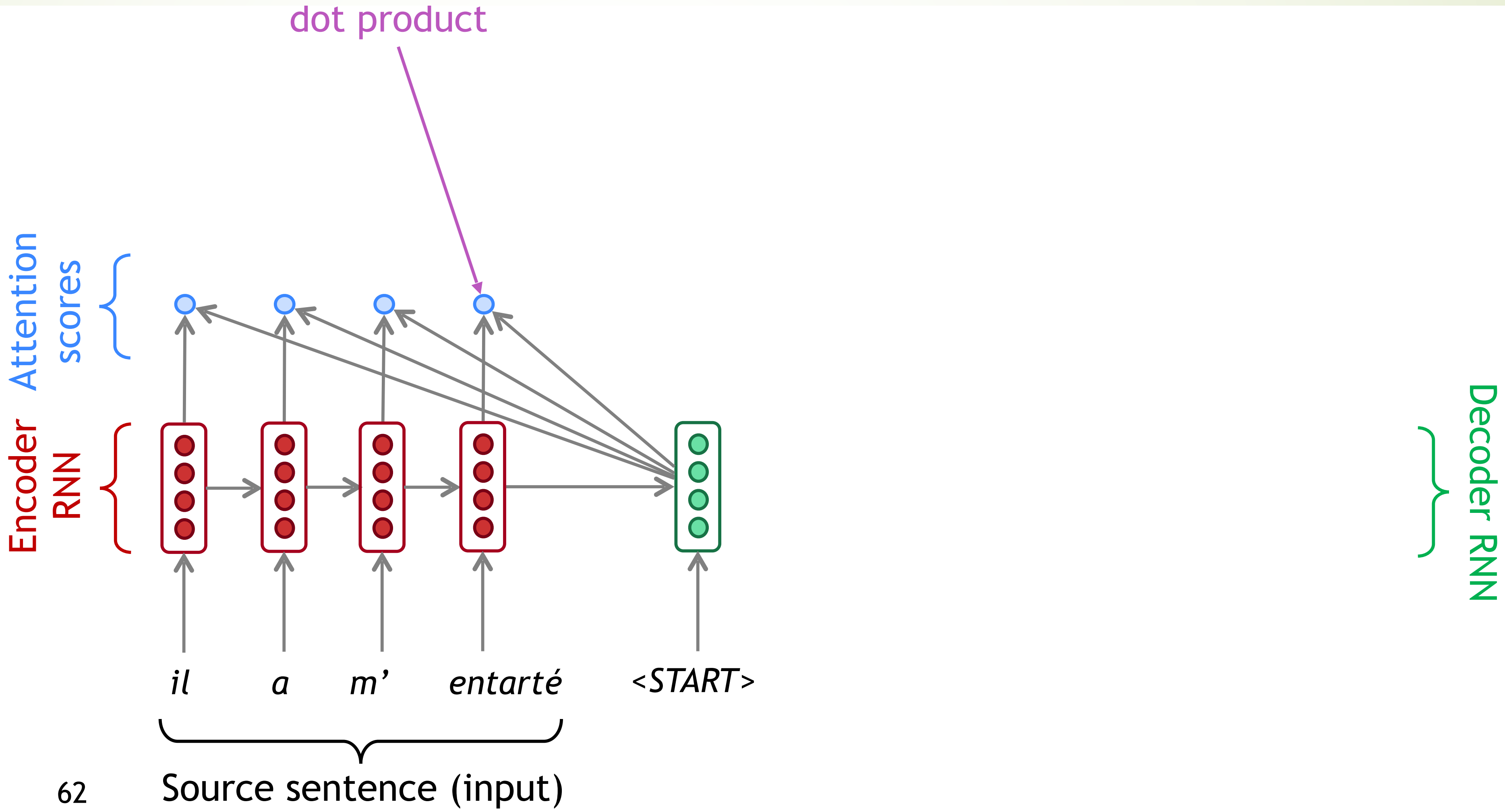


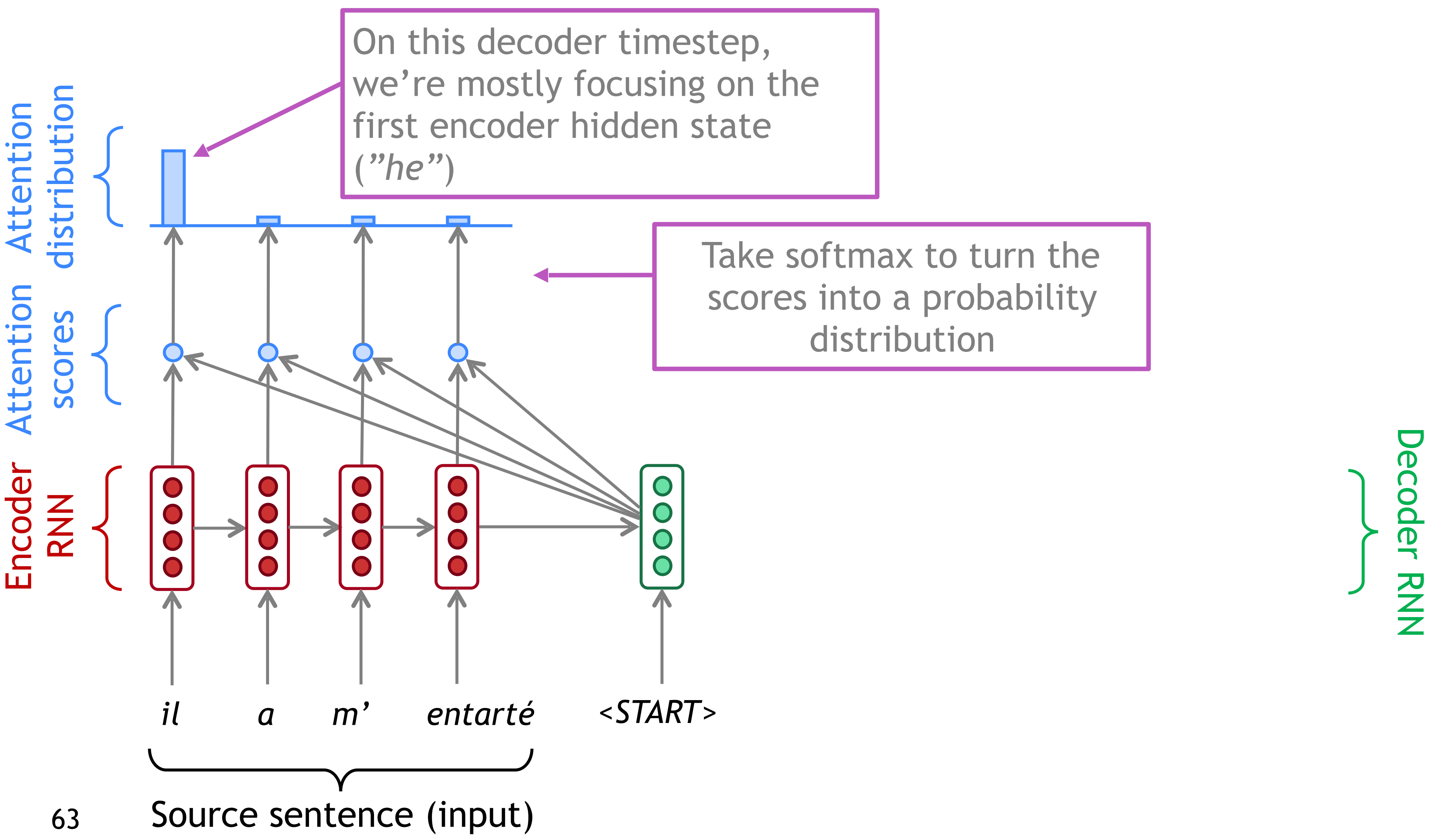
Attention Mechanism

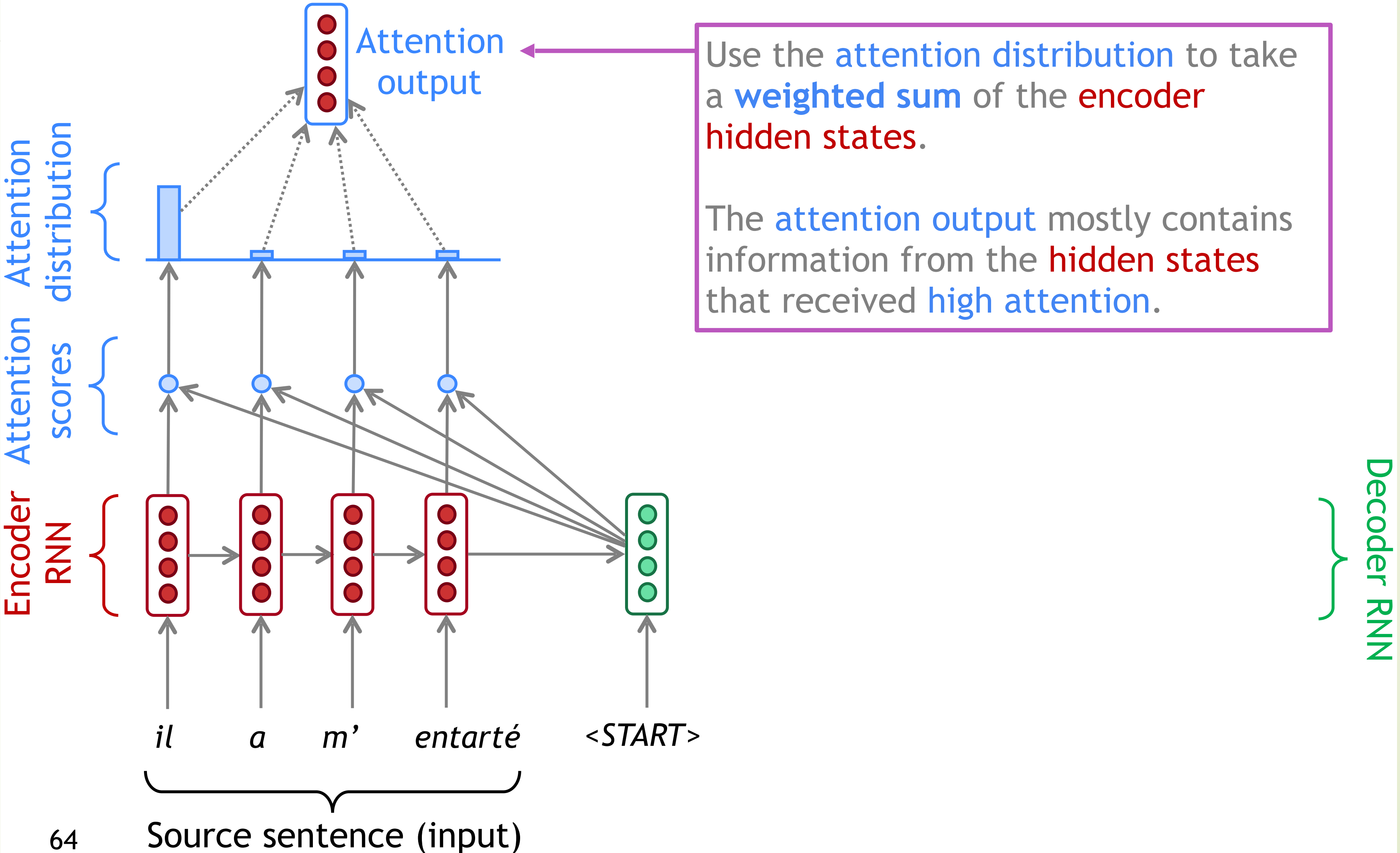
It was firstly invented in computer vision, then to NLP.

Sequence-to-sequence with attention



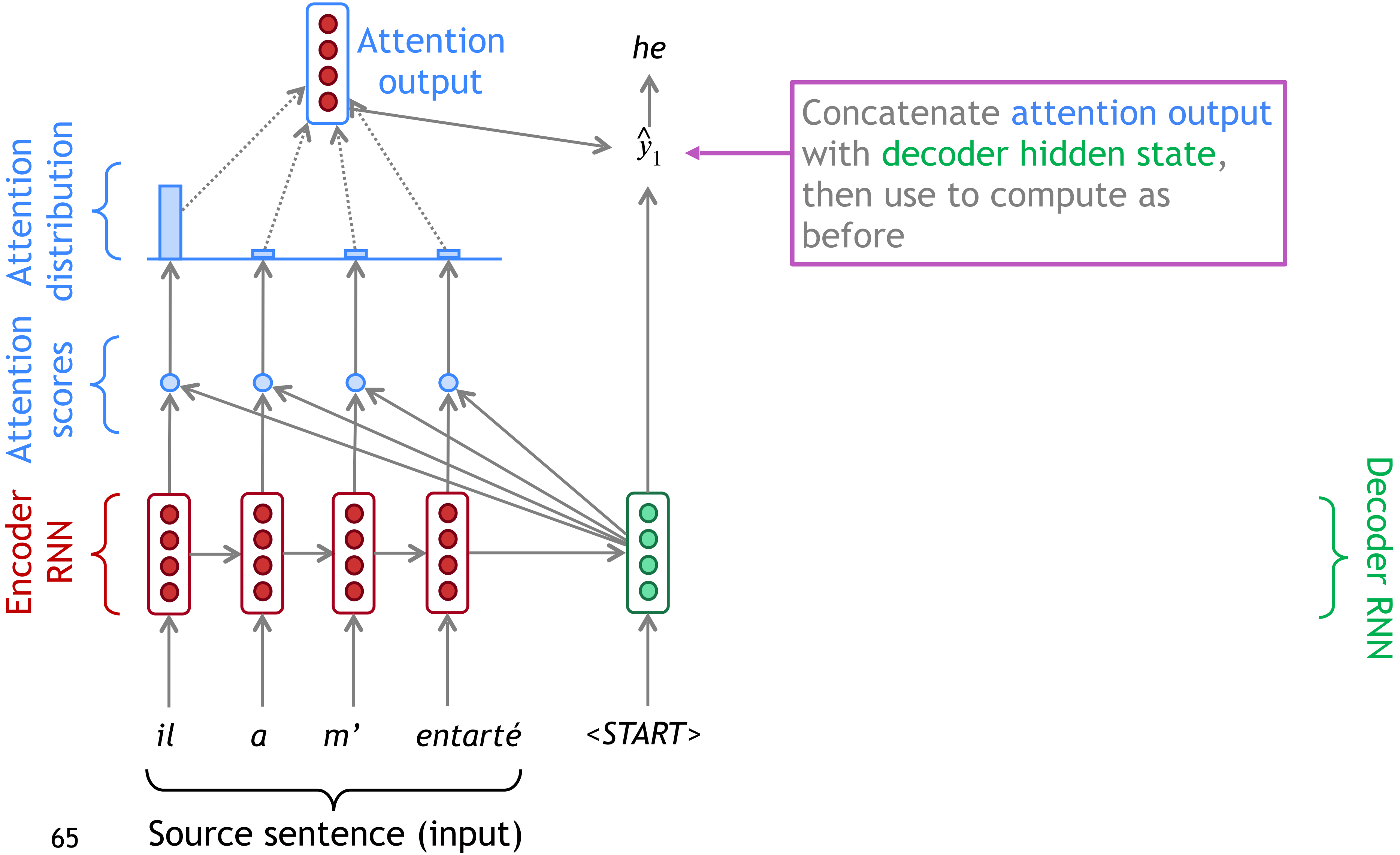

dot product
Attention scores
Encoder RNN
Decoder RNN
il
a
m’
entarté
<START>
Source sentence (input)


On this decoder timestep, we're mostly focusing on the first encoder hidden state ("he")
Take softmax to turn the scores into a probability distribution
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
il
a
m'
entarté
<START>
Source sentence (input)

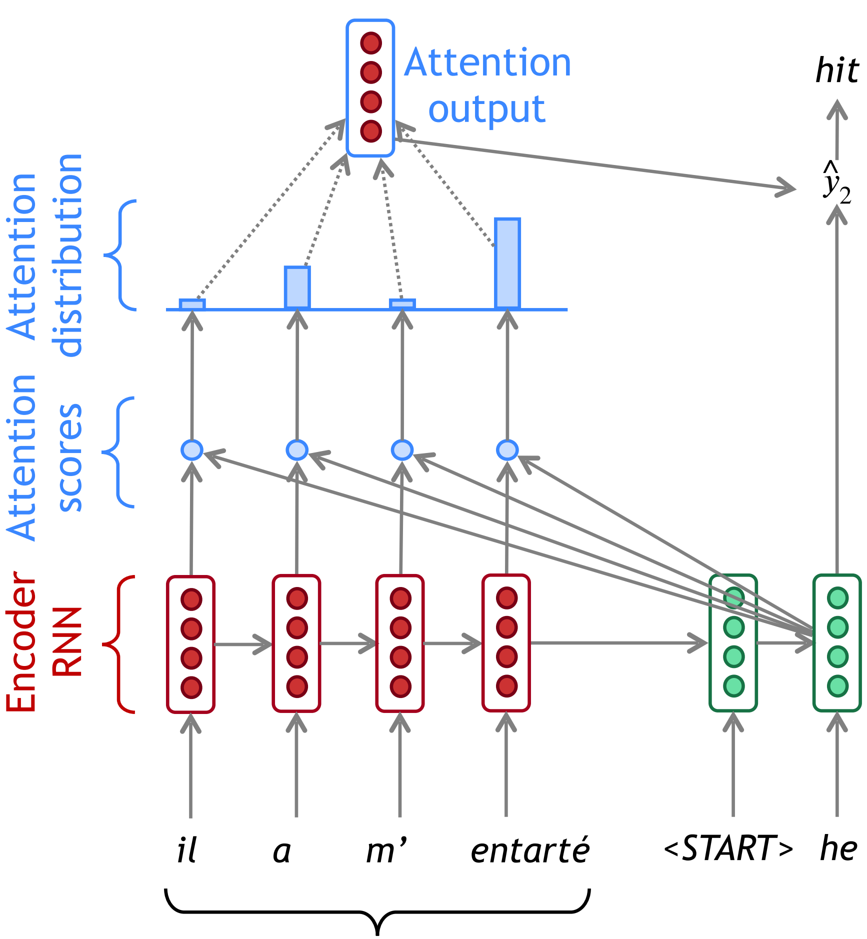


Use the attention distribution to take a **weighted sum** of the encoder hidden states.

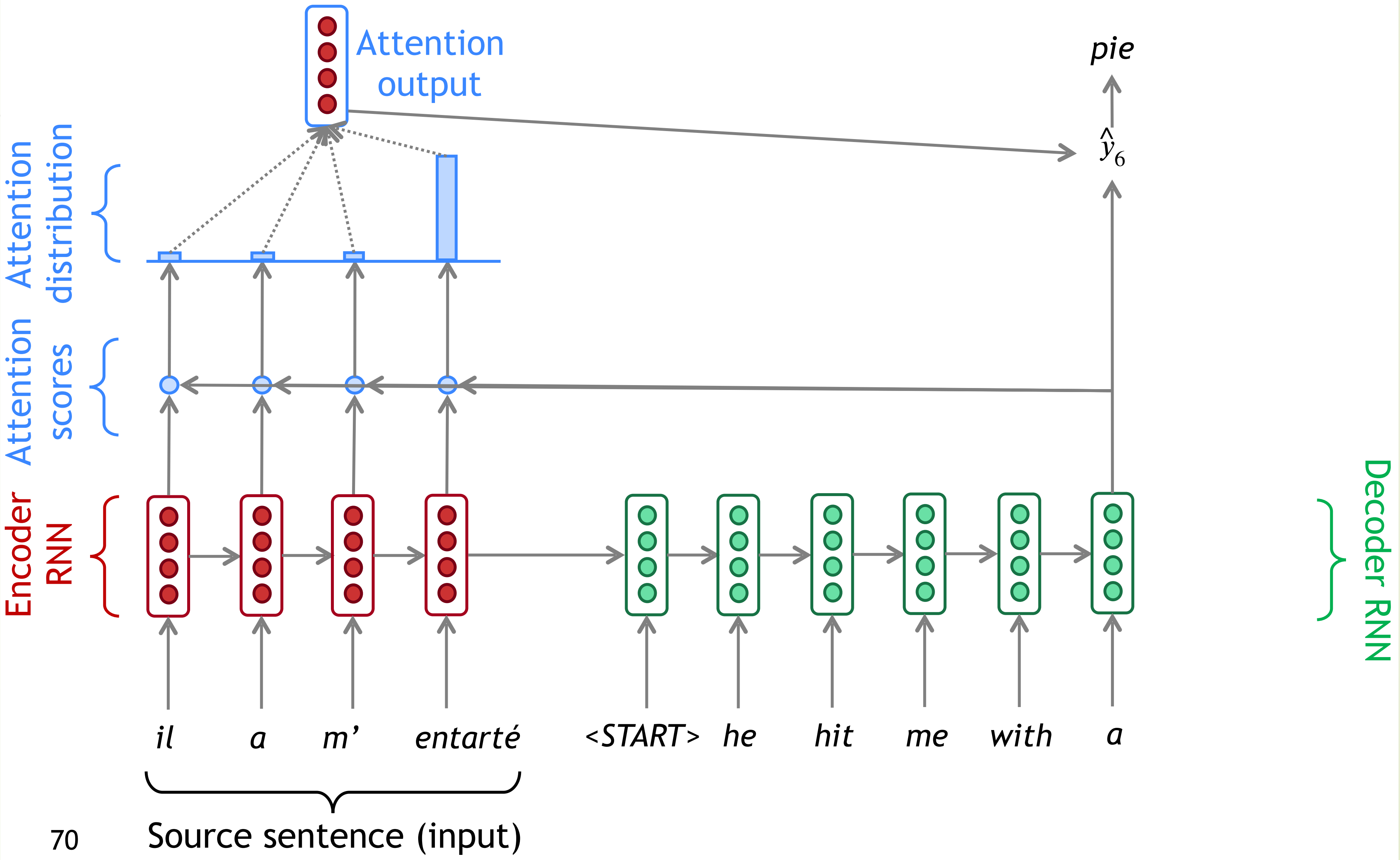
The attention output mostly contains information from the hidden states that received high attention.



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


Attention
output
hit
$\hat{y}_2$
Attention
distribution
Attention
scores
Encoder
RNN
il
a
m'
entarté
<START>
he
Source sentence (input)

Decoder RNN


Attention output
pie
$\hat{y}_6$
Attention distribution
Attention scores
Encoder RNN
Decoder RNN
il
a
m'
entarté
<START>
he
hit
me
with
a
Source sentence (input)

Attention in Equations

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

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

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

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

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

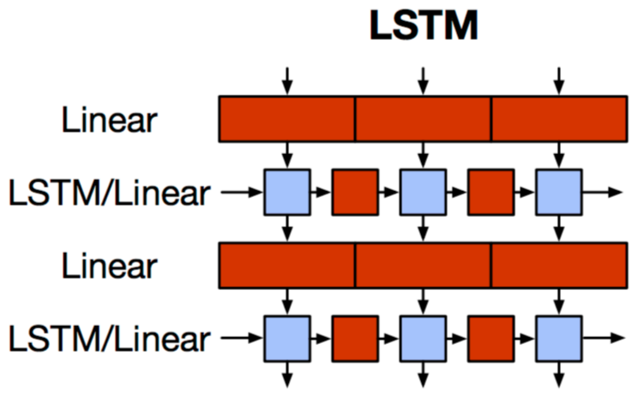
$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

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

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

Motivation of Transformer

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



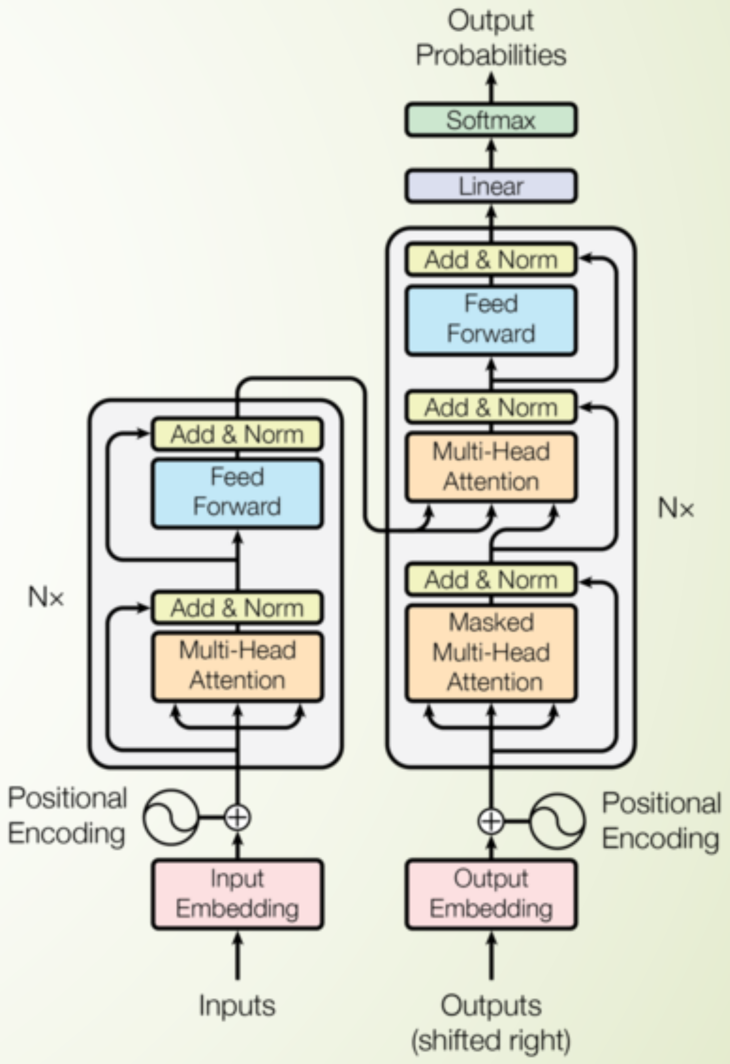
- Despite LSTMs, RNNs generally need attention mechanism to deal with long range dependencies – **path length** between states grows with distance otherwise
- But if **attention** gives us access to any state… maybe we can just use attention and don't need the RNN? 🤔
- And then NLP can have deep models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

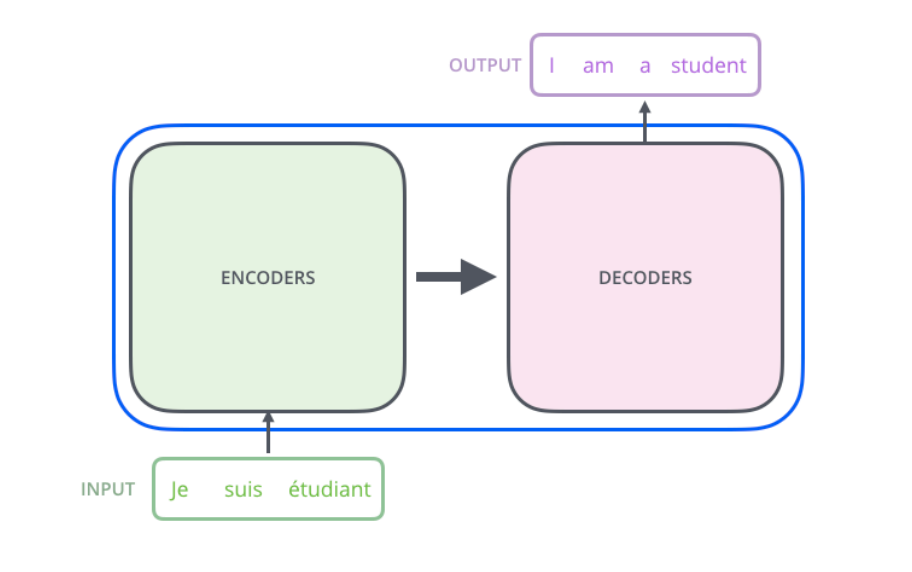


Transformer Pytorch Notebook

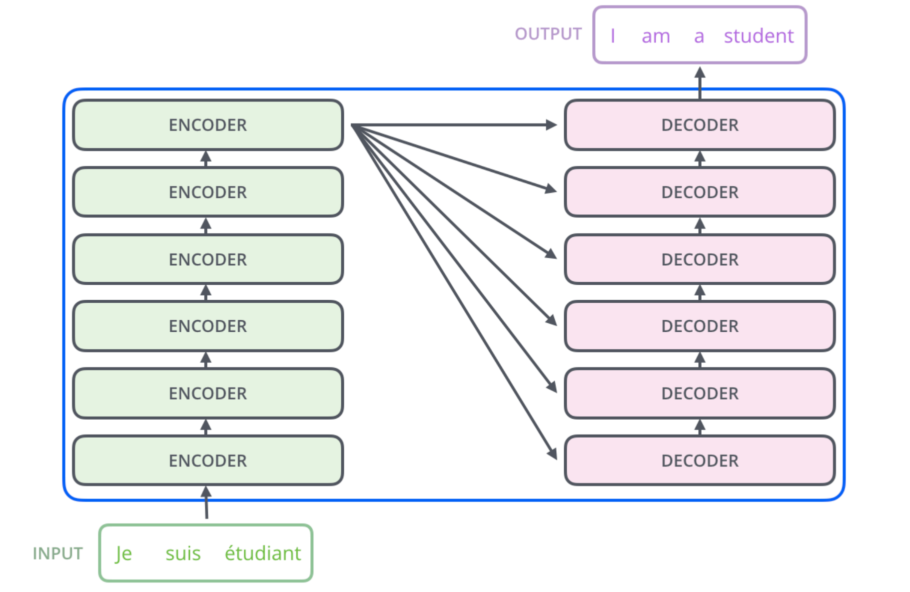
- Learning about transformers on your own?
- Key recommended resource:
  - http://nlp.seas.harvard.edu/2018/04/03/attention.html
  - The Annotated Transformer by Sasha Rush, a Jupyter Notebook using PyTorch that explains everything!
  - https://jalammar.github.io/illustrated-transformer/
  - Illustrated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

Encoder-Decoder Blocks

Encoder-Decoder

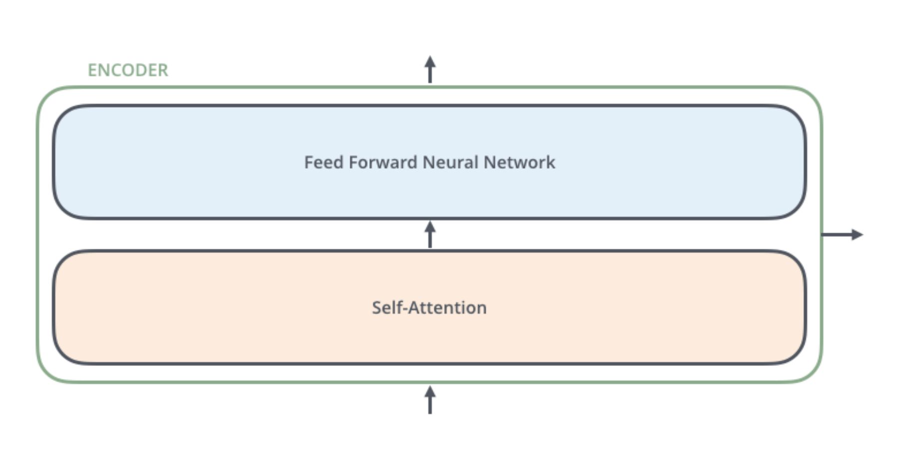


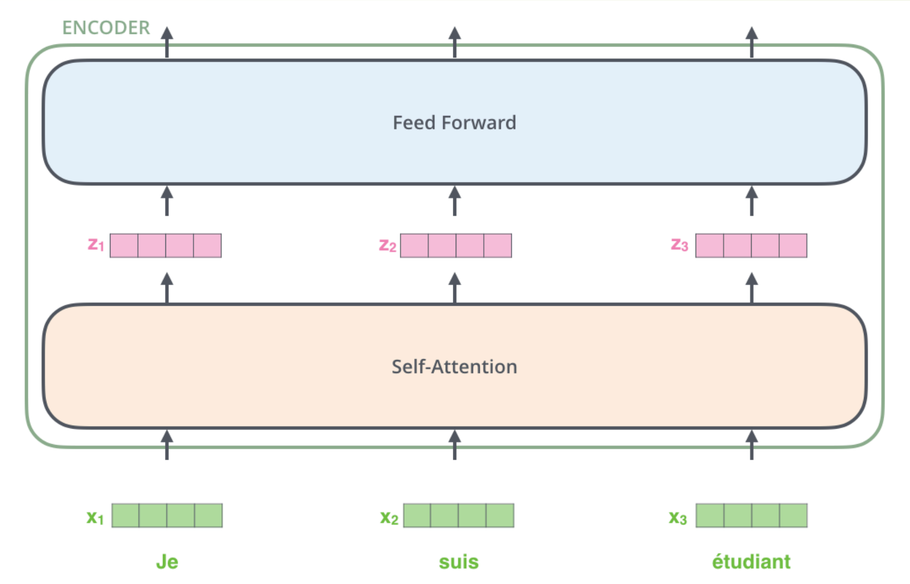
N=6 layers



Encoder has two layers

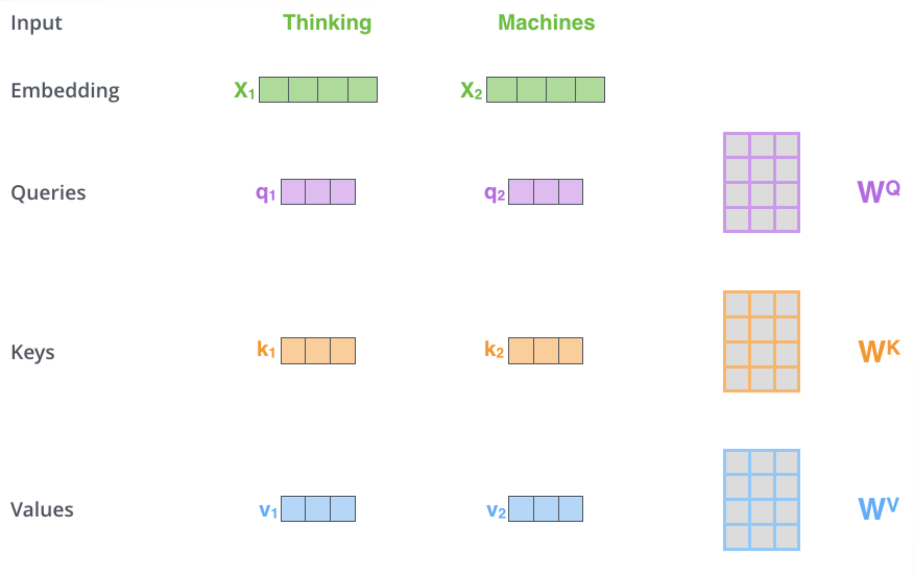
Self-Attention +
FeedForward



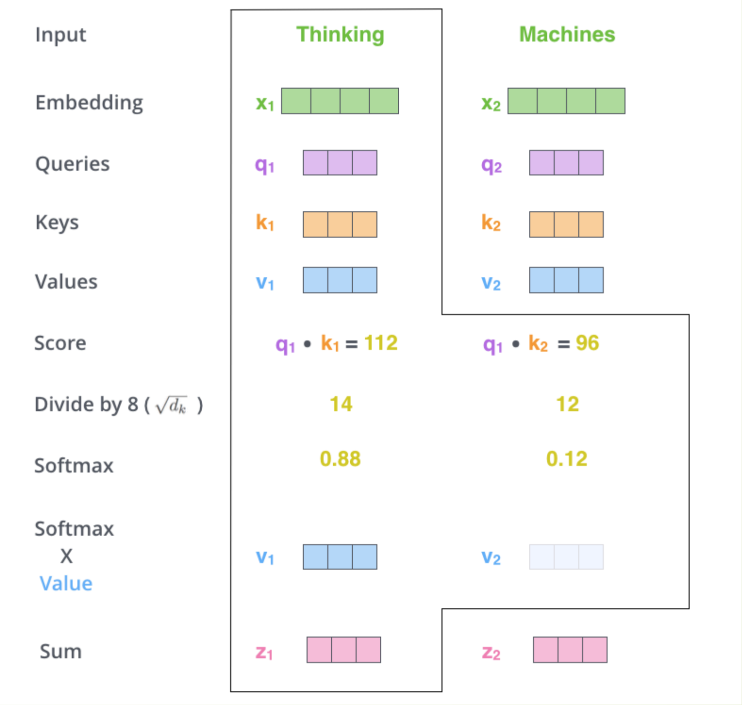


Attention Illustration

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

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

Scaled Dot-Product Attention

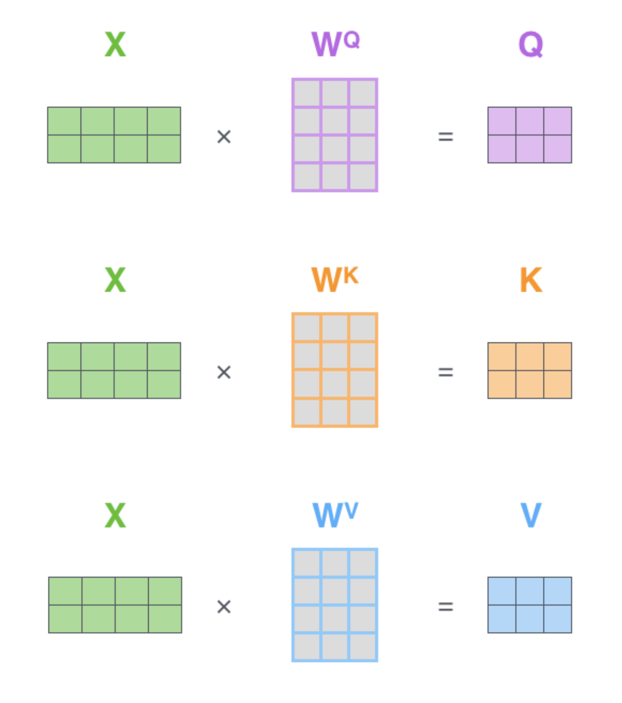
- **Problem**: As d_k gets large, the variance of q^Tk increases
- some values inside the softmax get large
- the softmax gets very peaked
- hence its gradient gets smaller.

- Solution: Scale by length of query/key vectors:

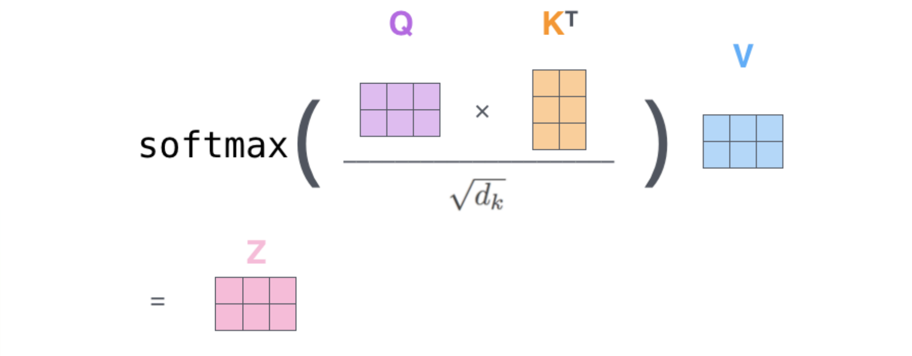
$$A(q, K, V) = \sum_i \frac{e^{\frac{q \cdot k_i}{\sqrt{d_k}}}}{\sum_j e^{\frac{q \cdot k_j}{\sqrt{d_k}}}} v_i$$

Attention: Multiple Inputs

Matrix input



Scaled dot-product



Dot-Product Attention: Matrix Form

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

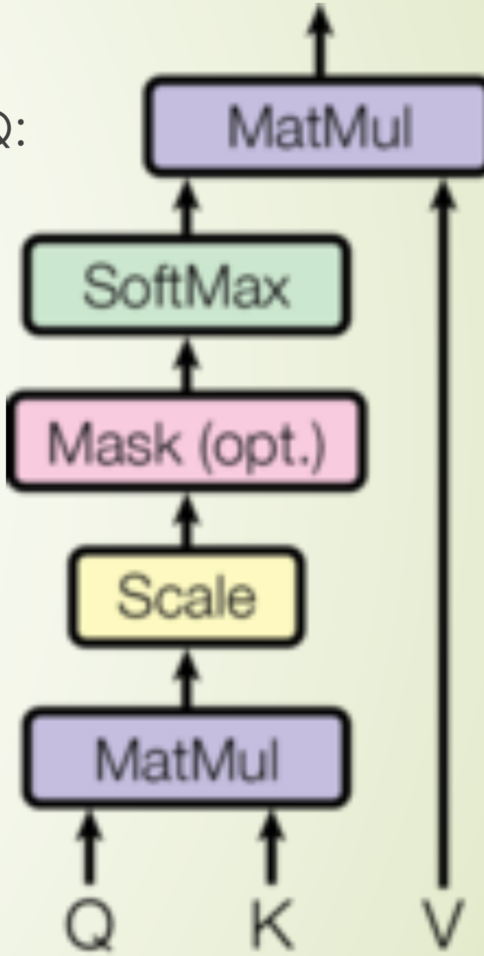
$$A(q, K, V) = \sum_i \frac{e^{\frac{q \cdot k_i}{\sqrt{d_k}}}}{\sum_j e^{\frac{q \cdot k_j}{\sqrt{d_k}}}} v_i$$

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

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

softmax
row-wise

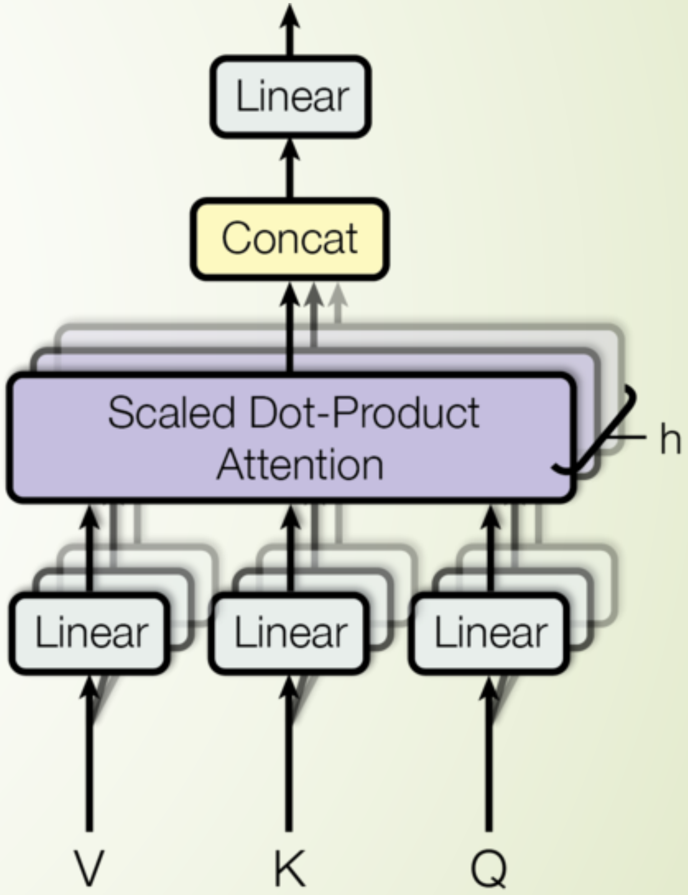
$= [|Q| \times d_v]$



Multi-head Attention

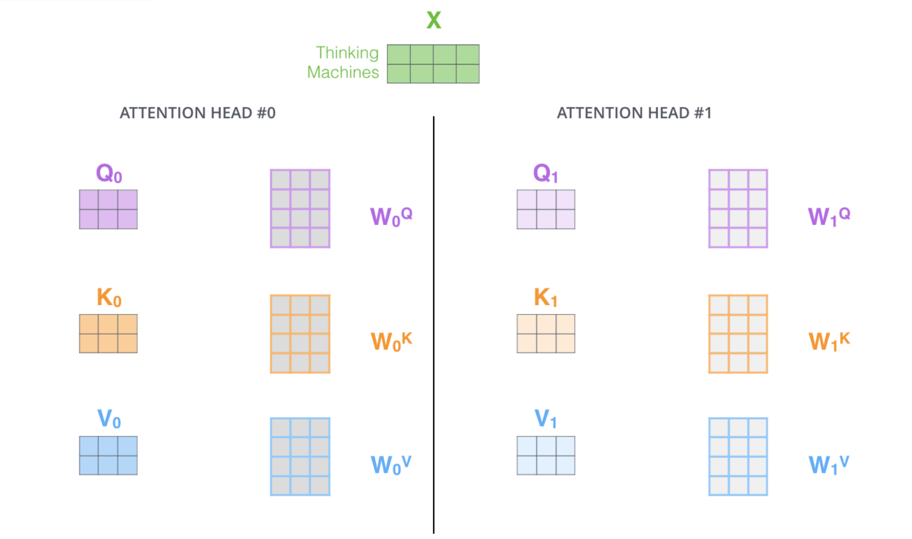
- **Problem** with simple self-attention:
  - Only one way for words to interact with one-another
- **Solution**: Multi-head attention
  - First map Q, K, V into h=8 many lower dimensional spaces via W matrices
  - Then apply attention, then concatenate outputs and pipe through linear layer
  - Multi-head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

$$
\begin{aligned}
\mathrm{MultiHead}(Q, K, V) &= \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_\mathrm{h})W^O \\
\text{where } \mathrm{head}_\mathrm{i} &= \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)
\end{aligned}
$$

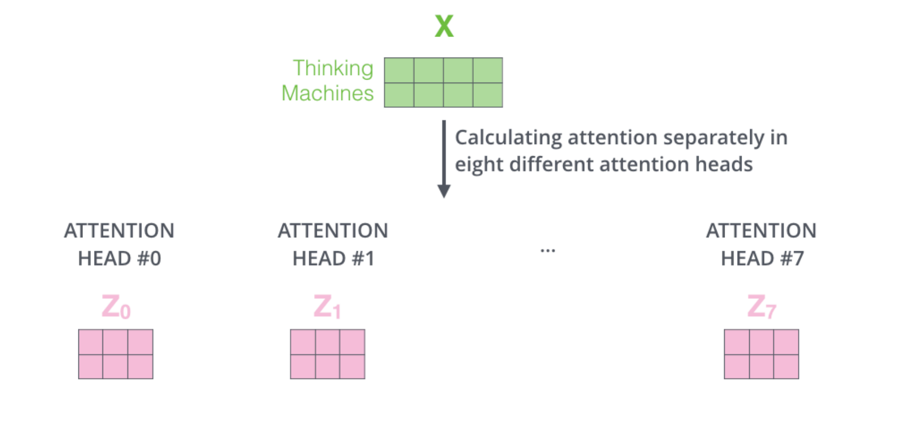


Multihead

2 heads



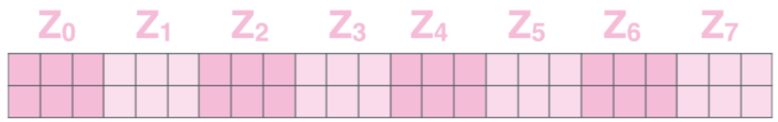
h=8 heads



Concatenation

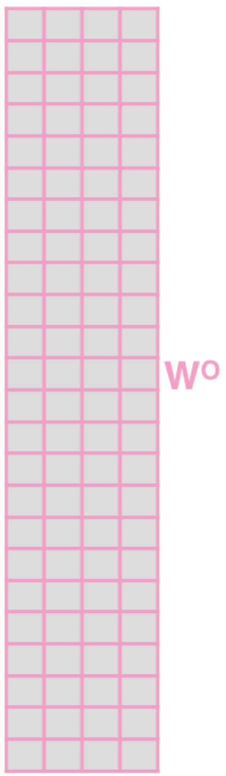
Linear

1) Concatenate all the attention heads

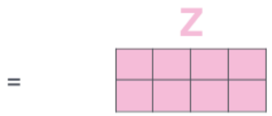


2) Multiply with a weight matrix W^O that was trained jointly with the model

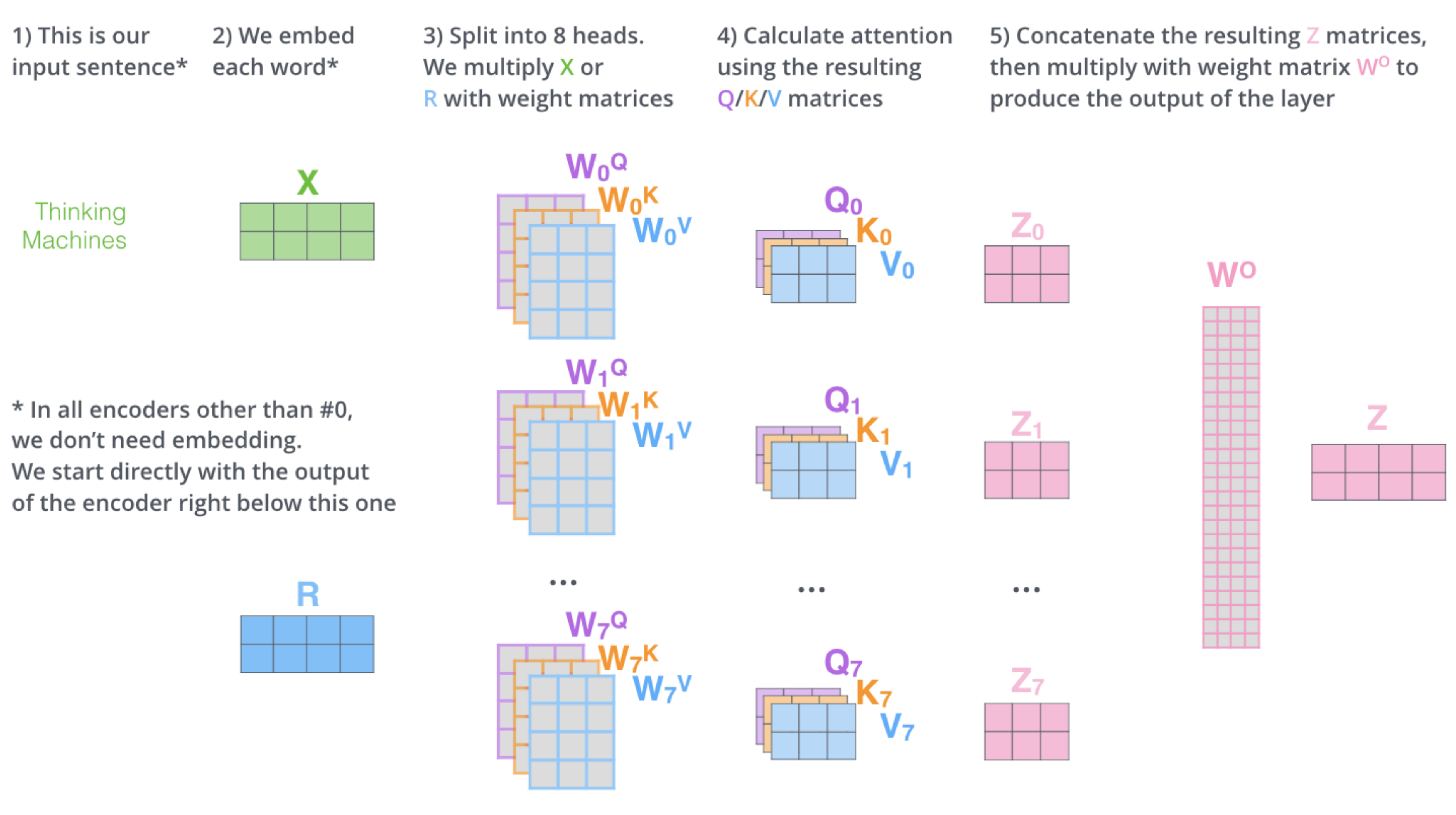
X



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

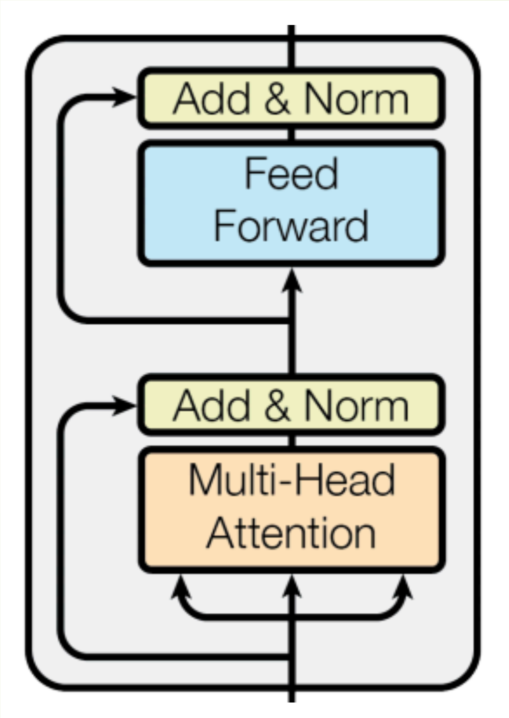


Multi-head Attention



A Transformer block

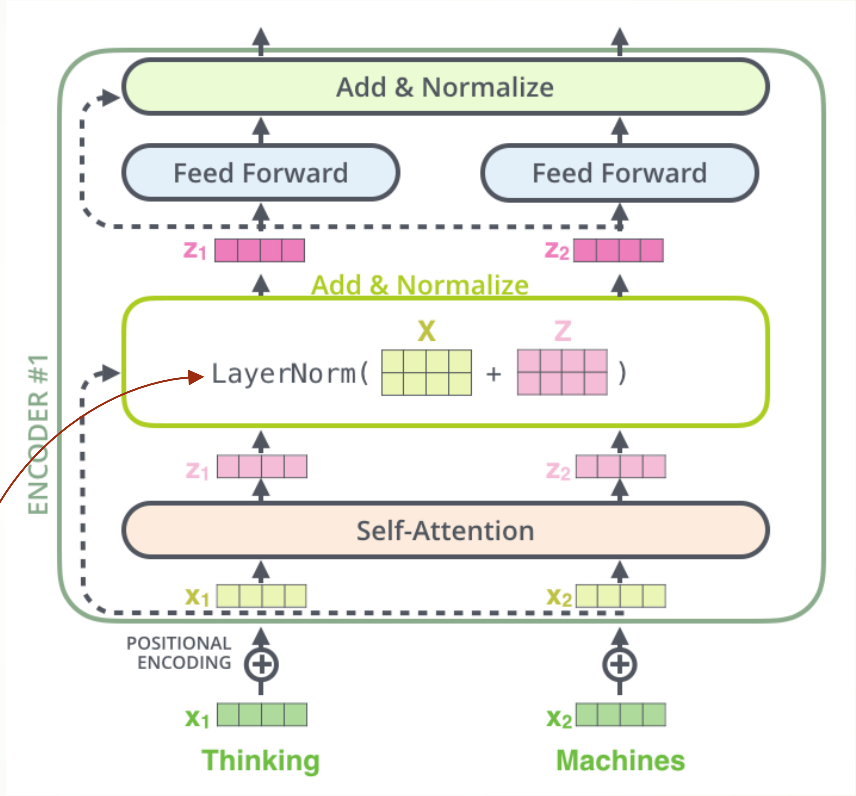
- Each block has two "sublayers"
  - Multihead attention
  - 2-layer feed-forward NNet (with ReLU)
- Each of these two steps also has:
  - Residual (short-cut) connection: x+sublayer(x)
  - LayerNorm(x+sublayer(x)) normalizes features over inputs to be of mean 0, variance 1, and adds two more parameters (Ba-Kiros-Hinton, 2016)



$$\tilde{z}_i^l = f\left(\frac{g^l}{\sigma^l}\left(z_i^l - \mu^l\right) + b^l\right)$$

$$\mu^l = \frac{1}{B}\sum_{i=1}^{B} z_i^l \quad \sigma^l = \sqrt{\frac{1}{B}\sum_{i=1}^{B}\left(z_i^l - \mu^l\right)^2} \quad B \text{ is the number of inputs}$$

Residue (Shortcut)



$$\mu^l = \frac{1}{B}\sum_{i=1}^{B} z_i^l \quad \sigma^l = \sqrt{\frac{1}{B}\sum_{i=1}^{B}\left(z_i^l - \mu^l\right)^2}$$ B is the number of inputs

$$\tilde{z}_i^l = f\left(\frac{g^l}{\sigma^l}\left(z_i^l - \mu^l\right) + b^l\right)$$

Encoder Input

- Actual word representations are word pieces: byte pair encoding
  - Start with a vocabulary of characters
  - Most frequent ngram pairs ↦ a new ngram
  - Example: “es, est” 9 times, “lo” 7 times

Dictionary

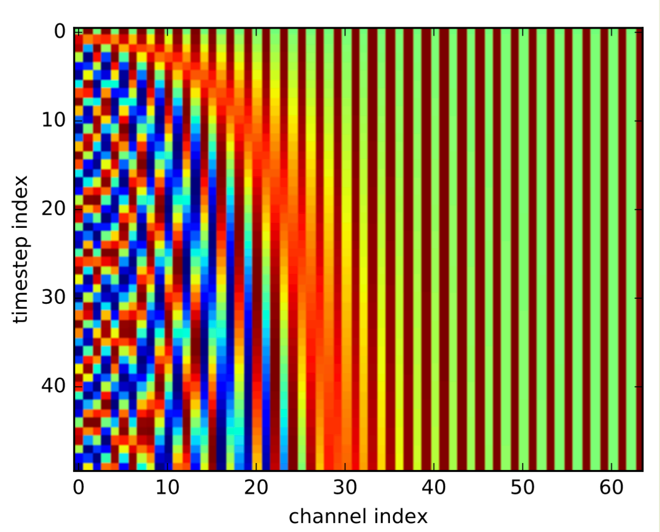
5 **lo** w
2 **lo** w e r
6 n e w est
3 w i d est

Vocabulary

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

- Also added is a **positional encoding** so same words at different locations have different overall representations:

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



where *pos* is the position and *i* is the dimension.
Or learned.

Sin/Cos Position Encoding

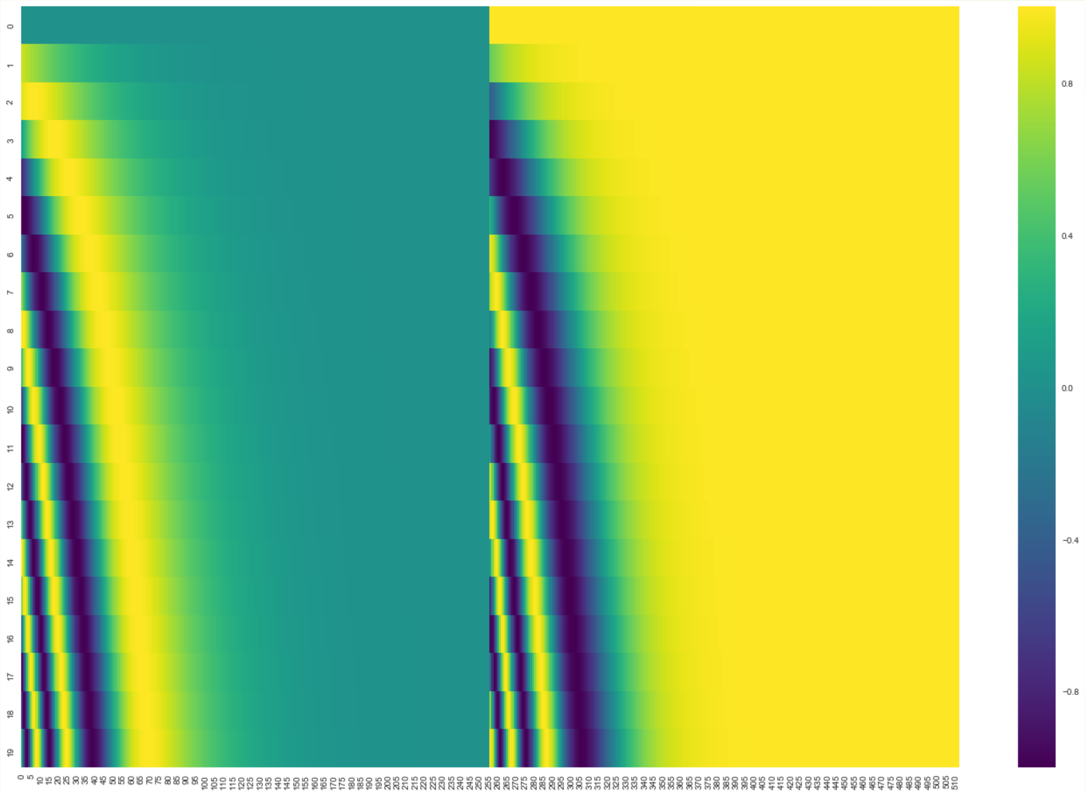
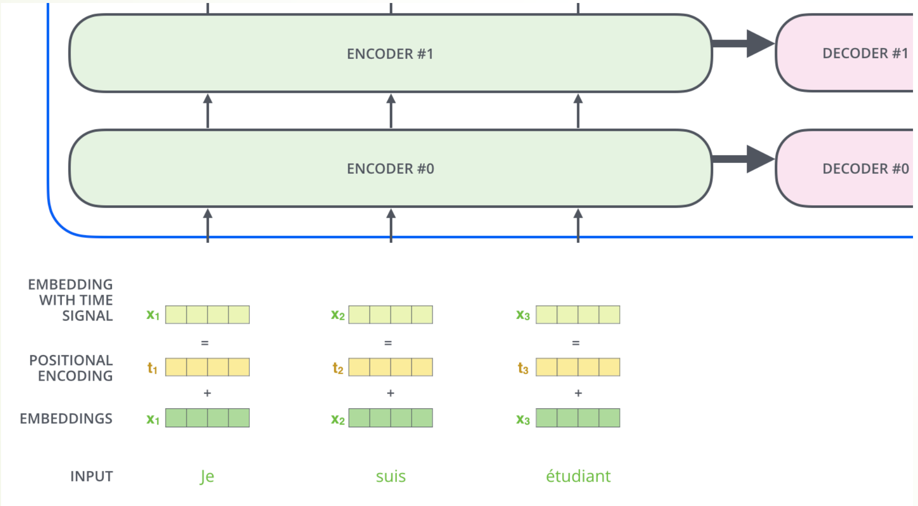
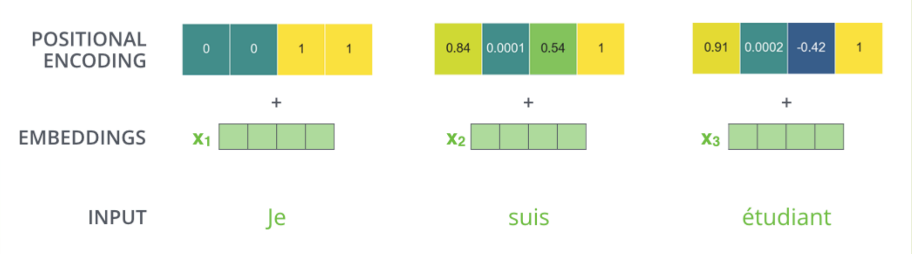


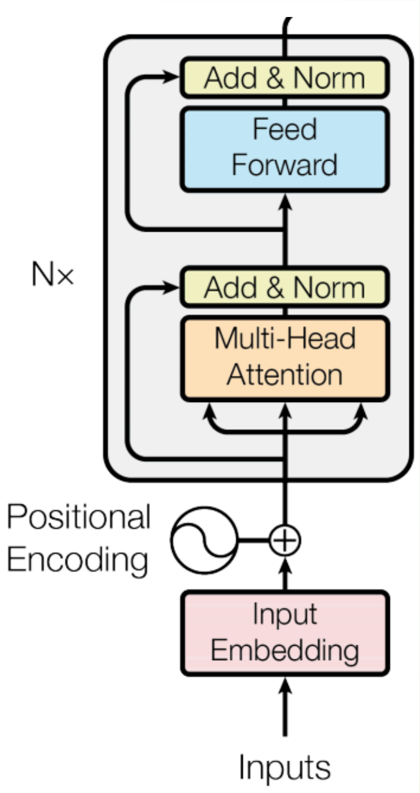
Figure. A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). Each row corresponds the a positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.

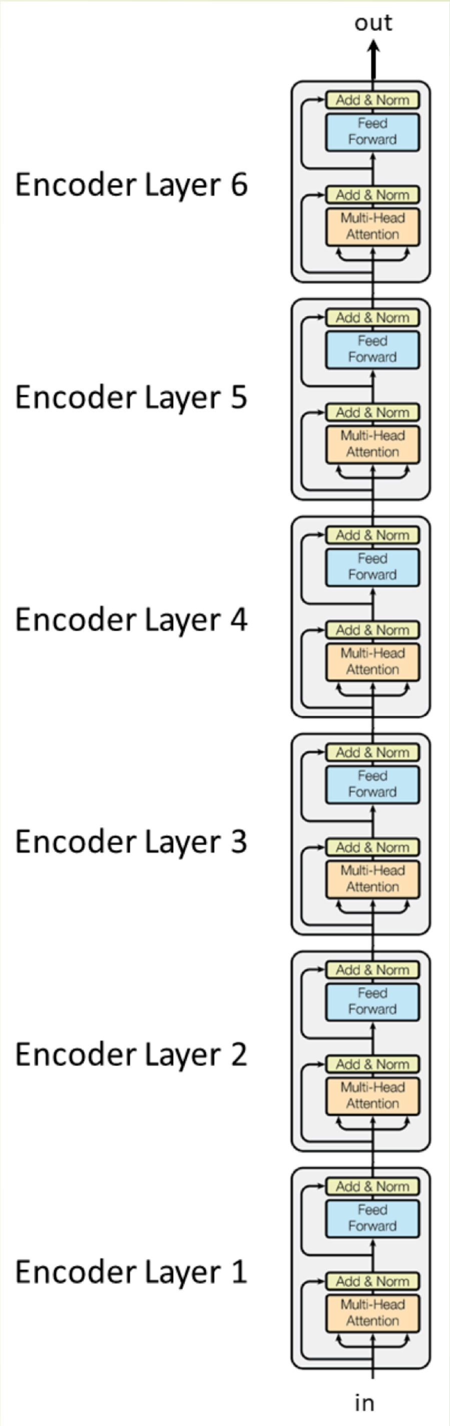

ENCODER #1
DECODER #1
ENCODER #0
DECODER #0
EMBEDDING WITH TIME SIGNAL
POSITIONAL ENCODING
EMBEDDINGS
INPUT
Je
suis
étudiant


POSITIONAL ENCODING
0
0
1
1
0.84
0.0001
0.54
1
0.91
0.0002
-0.42
1
EMBEDDINGS
INPUT
Je
suis
étudiant

Transformer Encoder

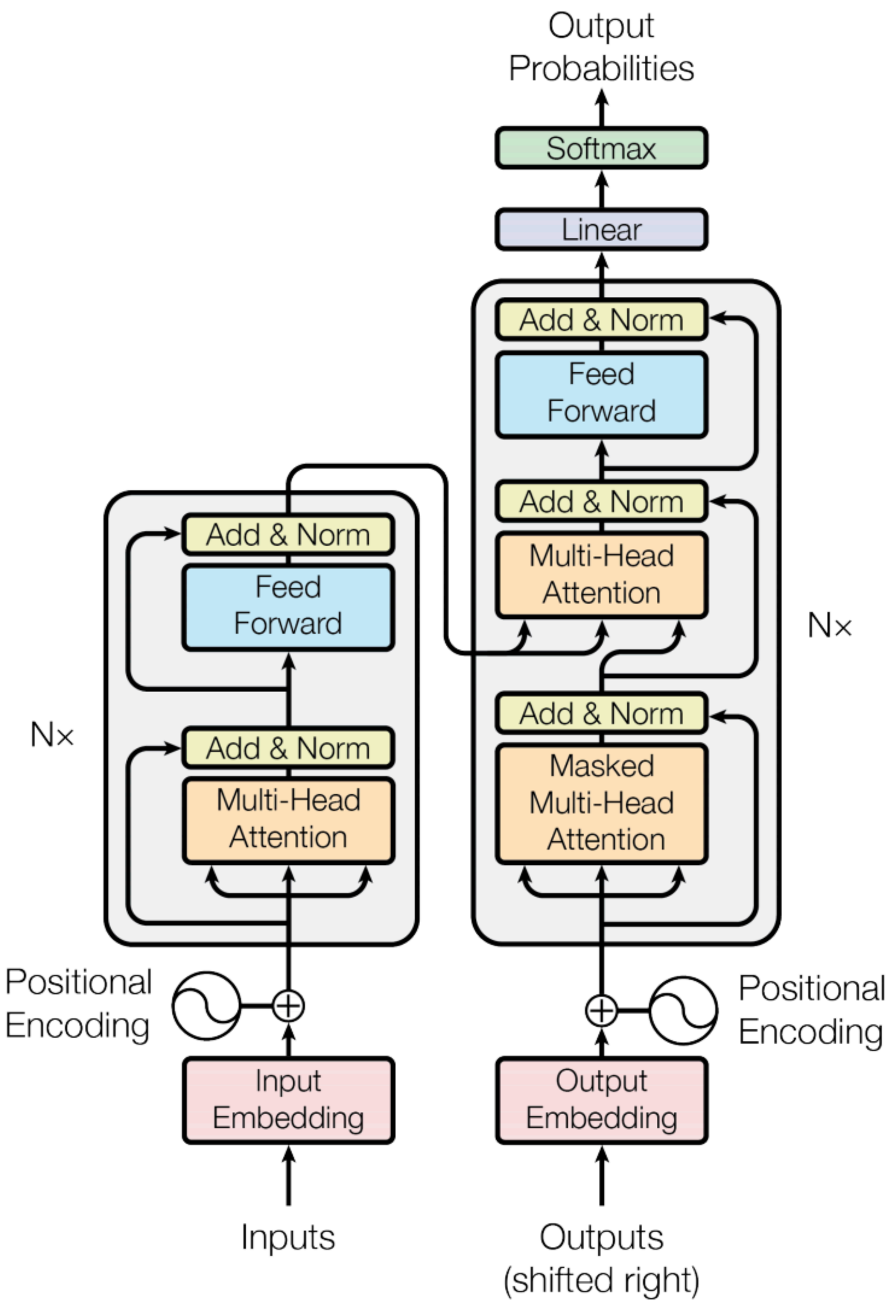
- Blocks are repeated N=6 or more times



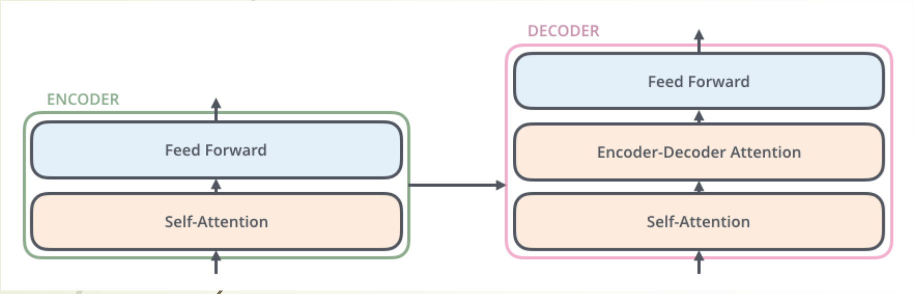


Transformer Decoder

- 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



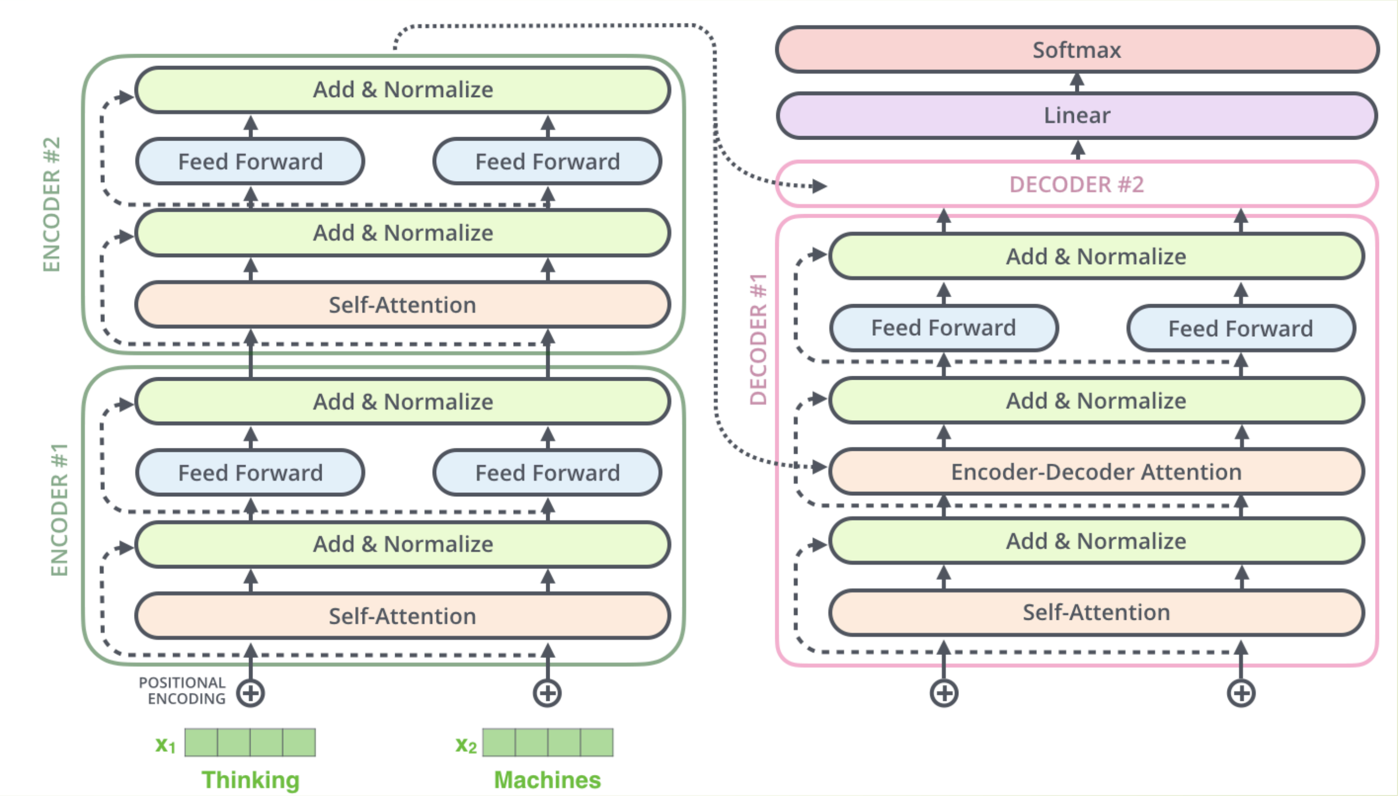


Illustration of Encoder-Decoder

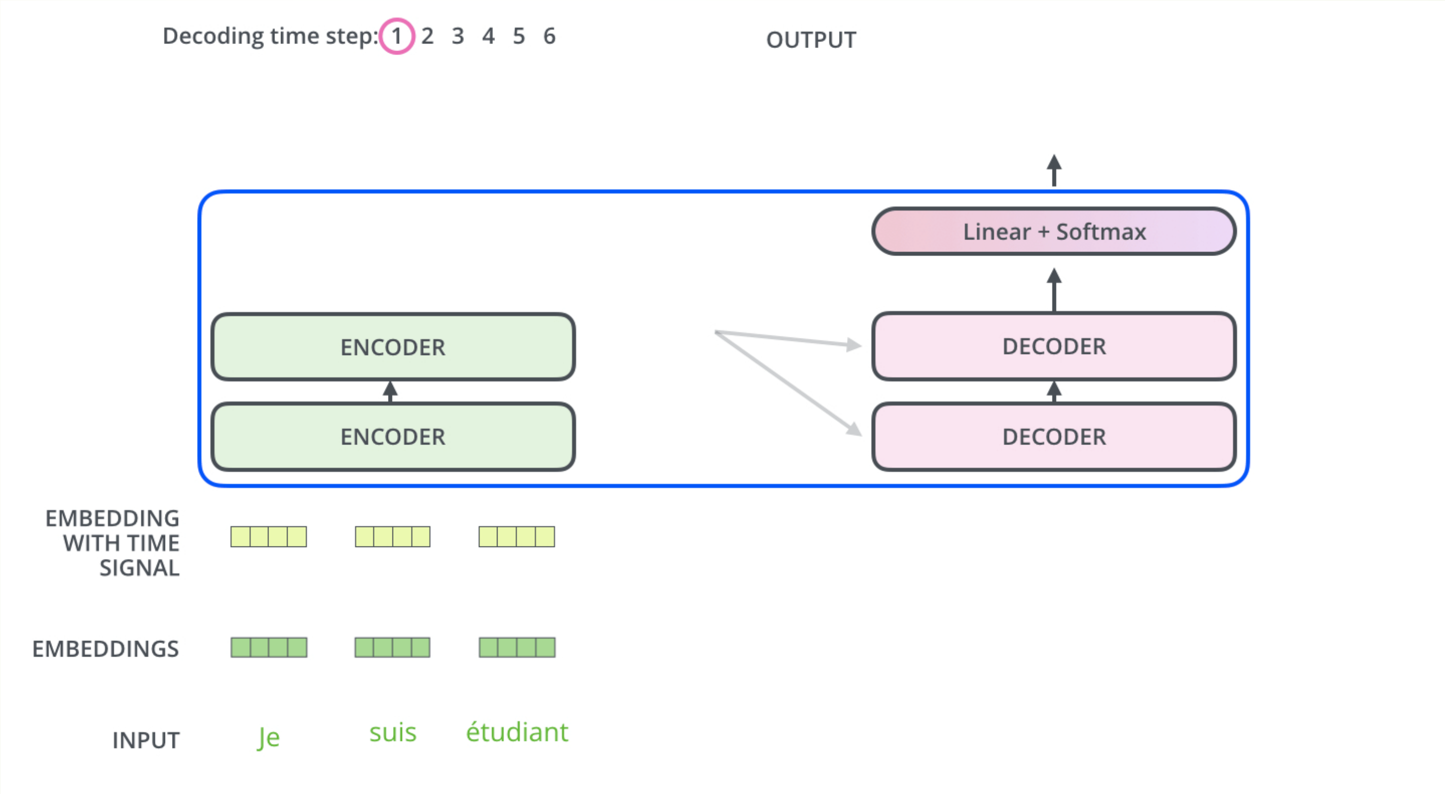
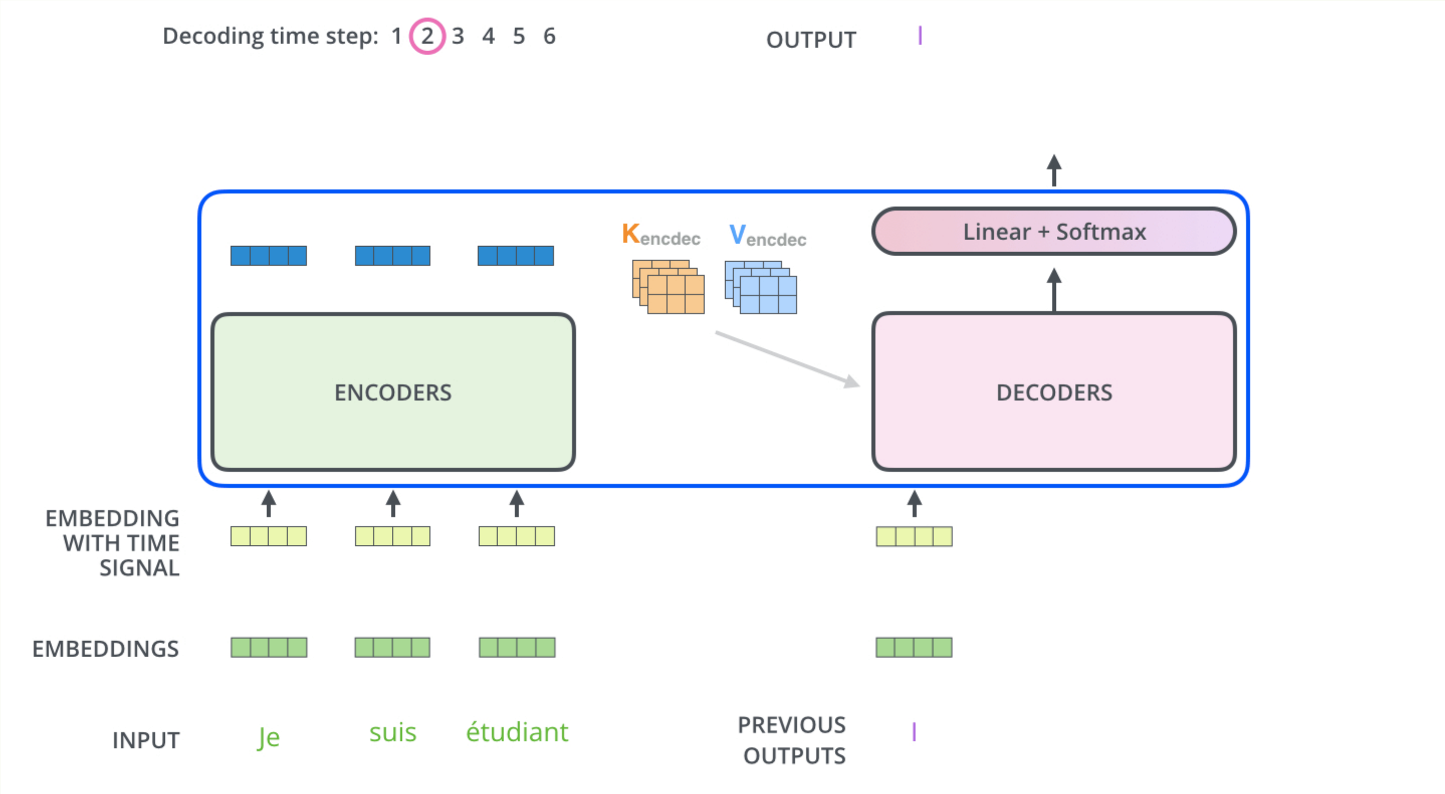
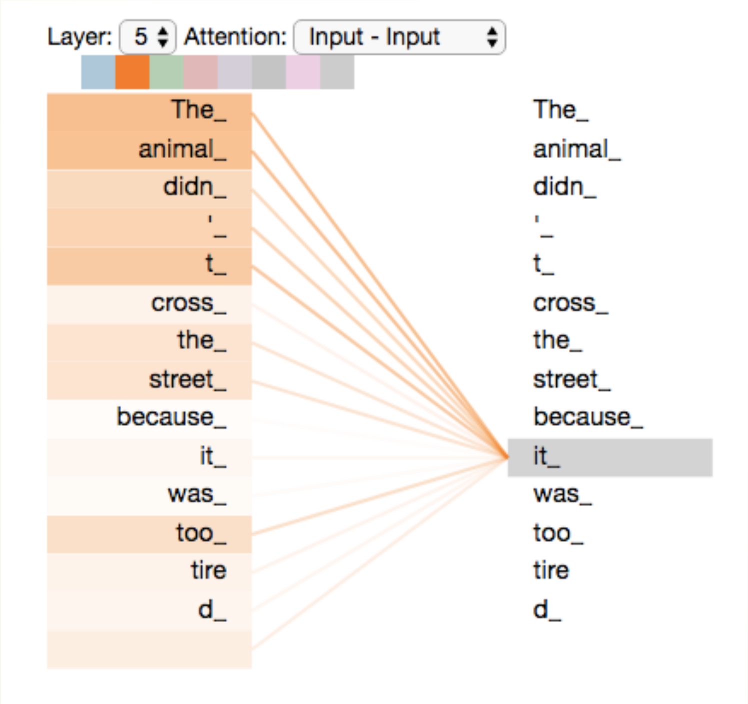


Illustration of Encoder-Decoder

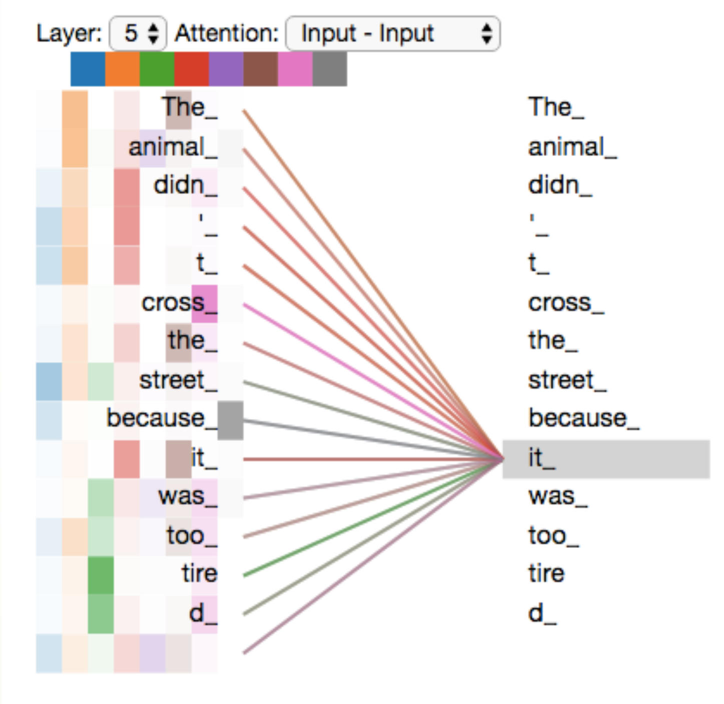


Attention Visualization

Head 2 (yellow) only

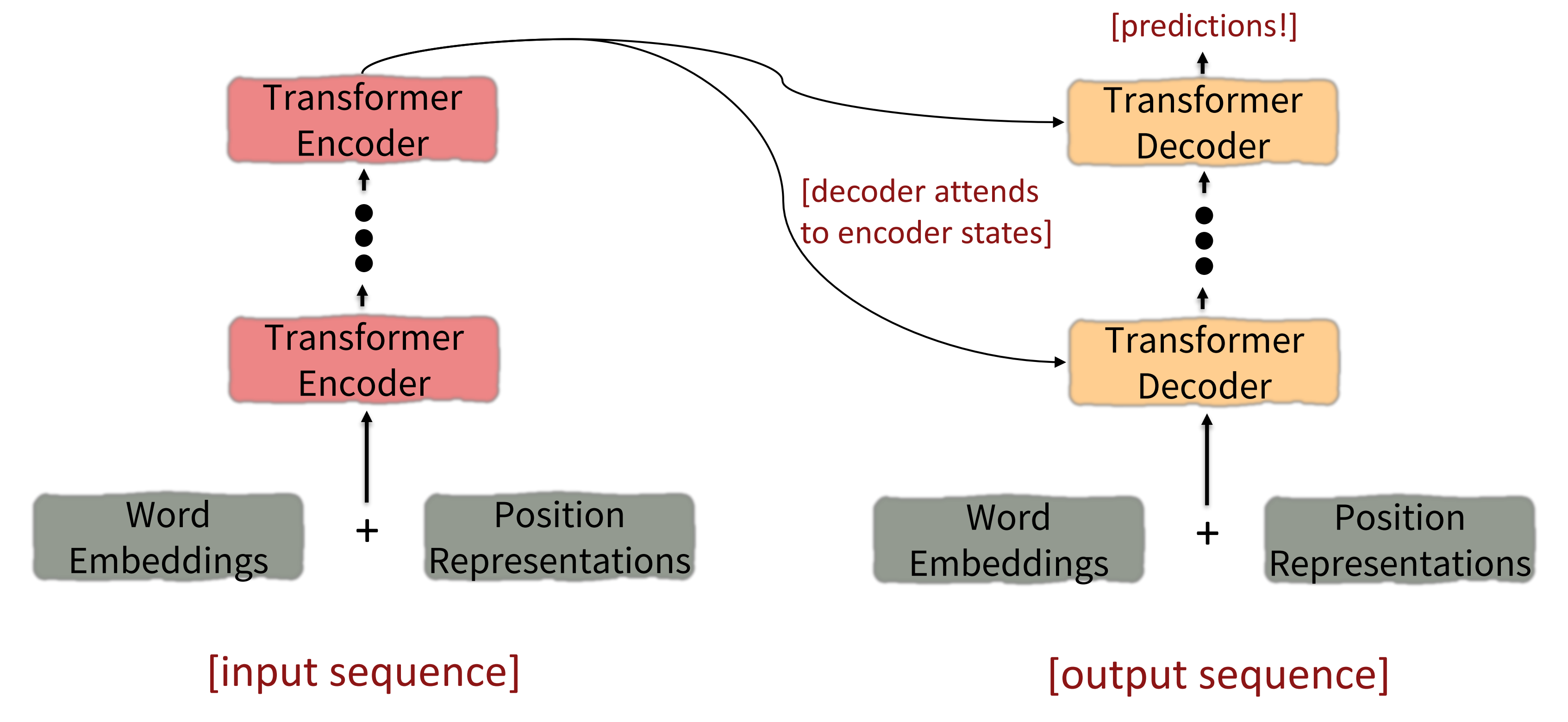


8 heads mixture



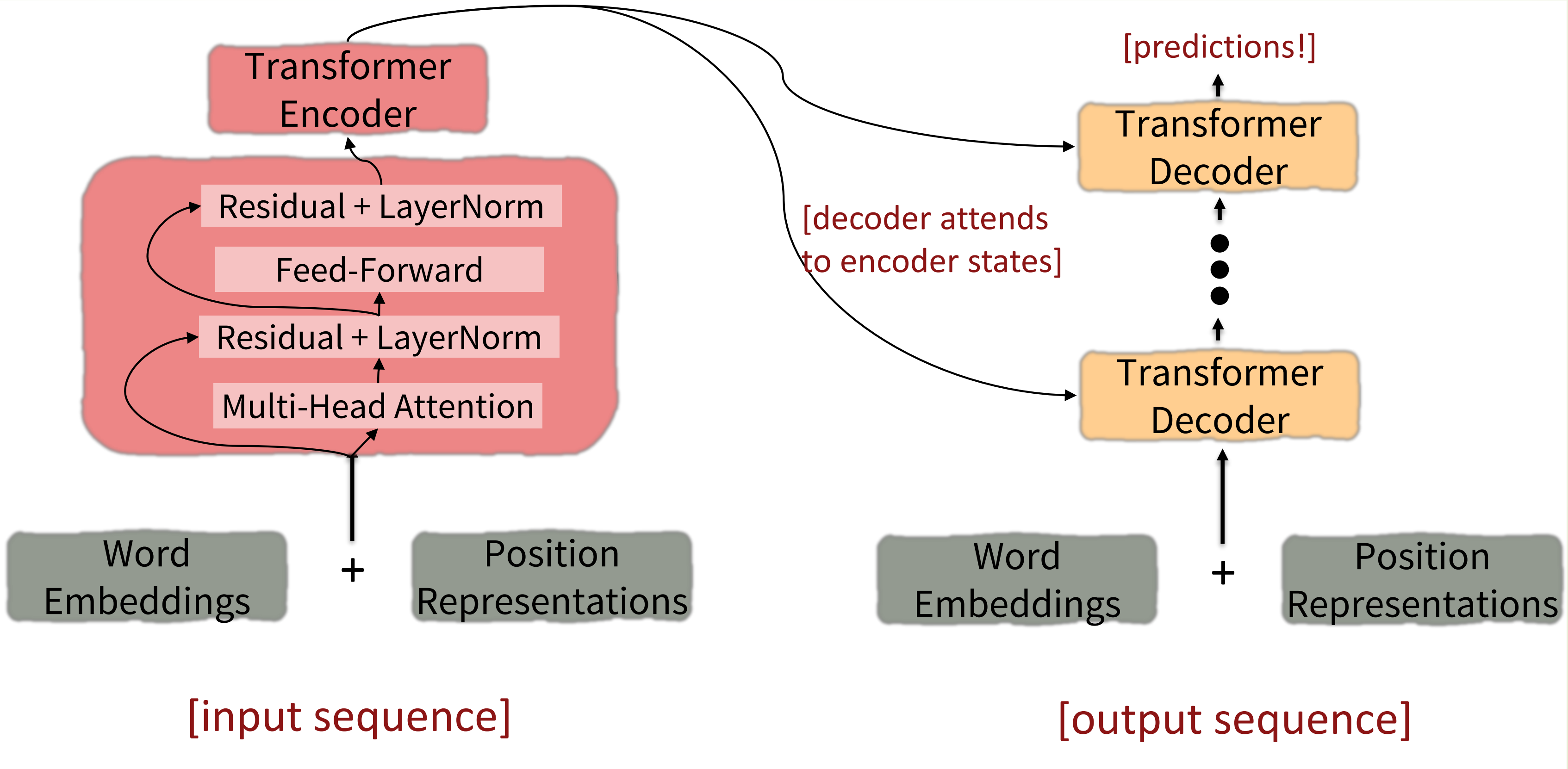
Summary: The Transformer Encoder-Decoder [Vaswani et al. 2017]

- Looking back at the whole model



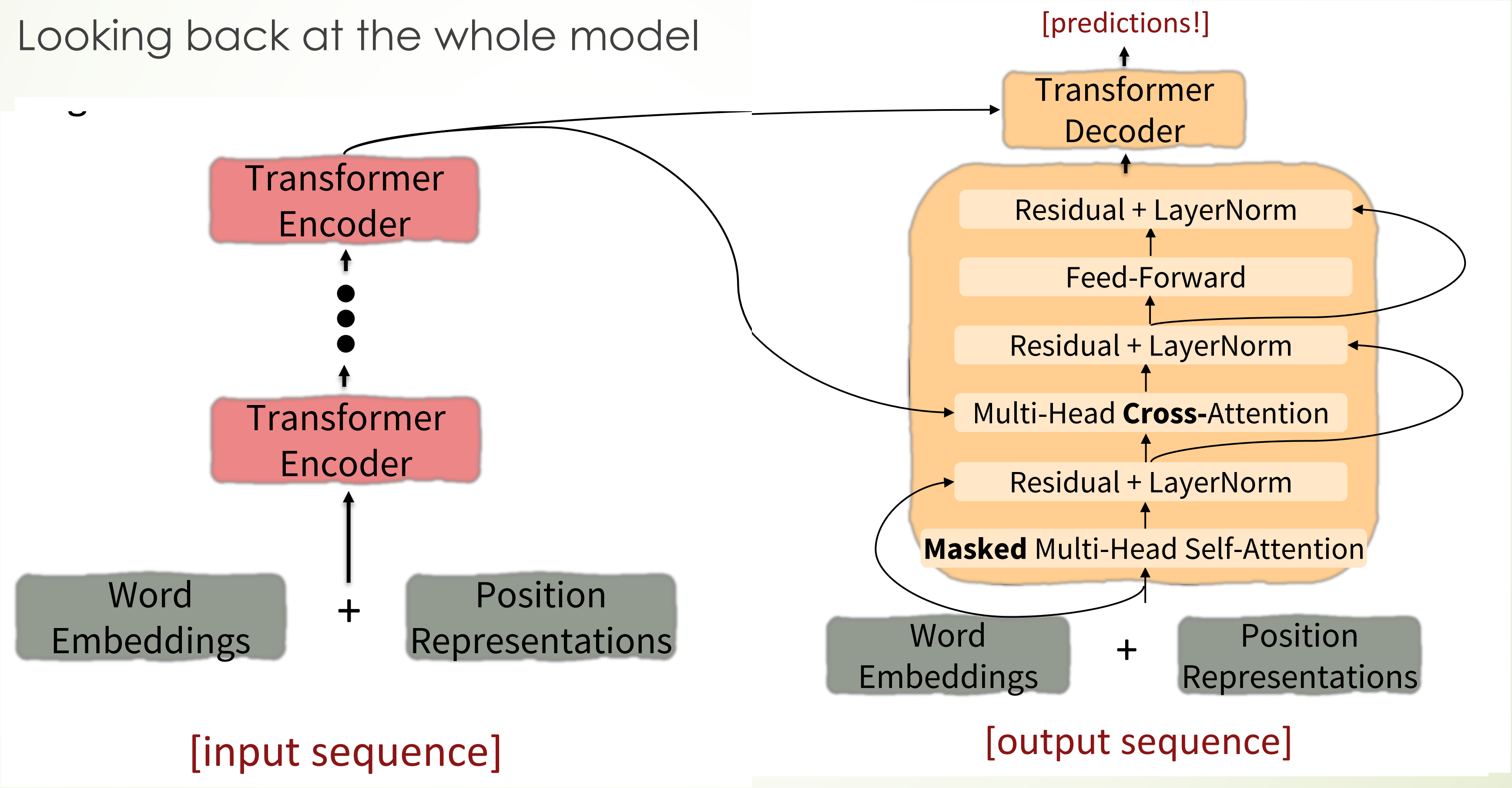
Summary: The Transformer Encoder-Decoder [Vaswani et al. 2017]

- Looking back at the whole model



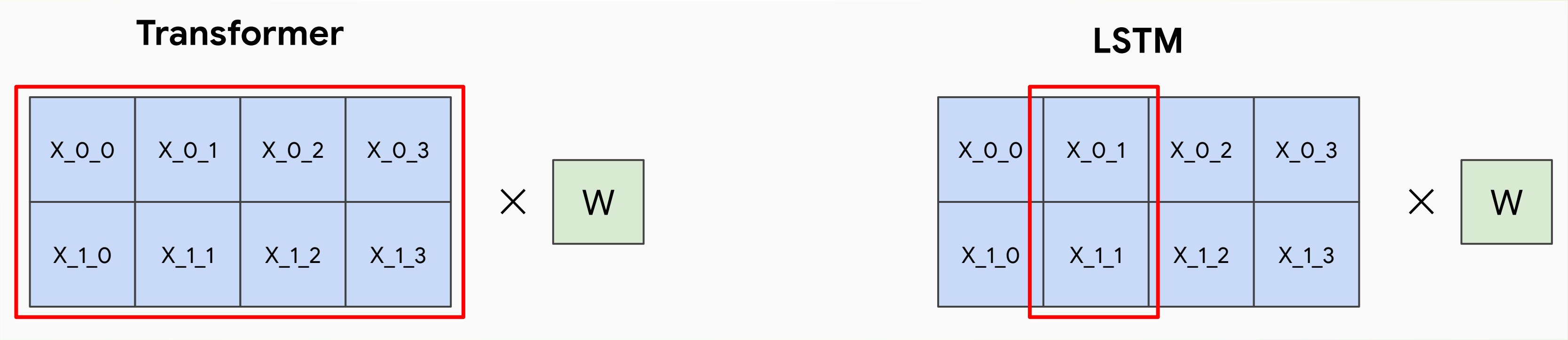
Summary: The Transformer Encoder-Decoder [Vaswani et al. 2017]

- Looking back at the whole model



Empirical advantages of Transformer vs. LSTM

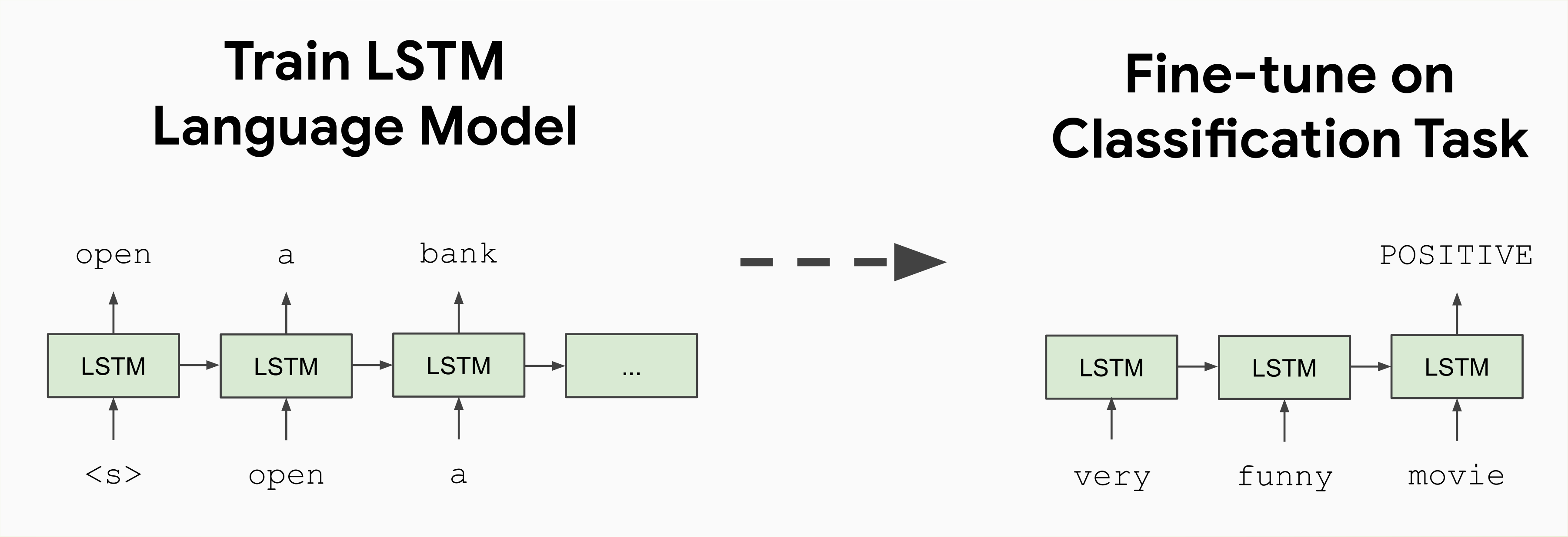
- 1. Self-attention == no locality bias
  - Long-distance context has "equal opportunity"
- 2. Single multiplication per layer == efficiency on TPU



Bi-Directionality

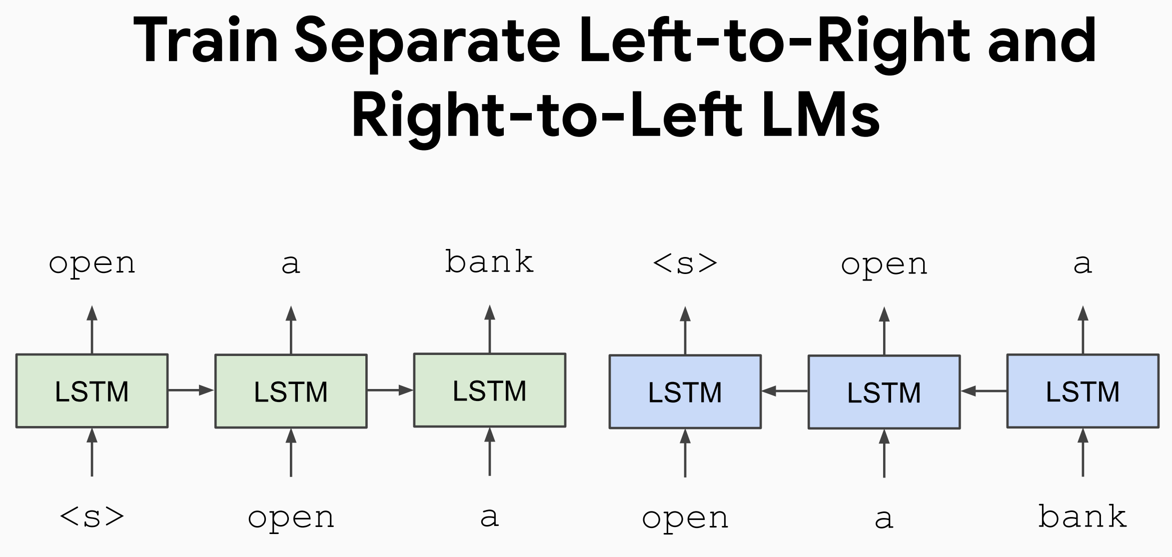
Uni-Direction LSTM

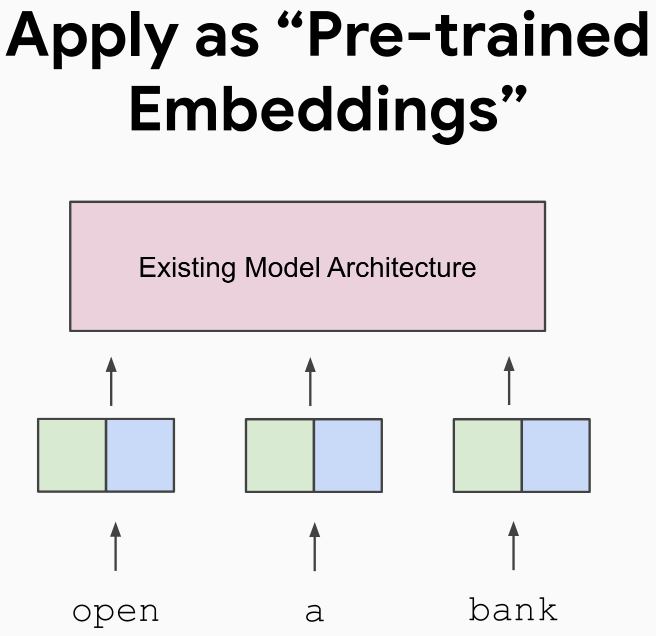
- Semi-Supervised Sequence Learning, Google, 2015



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

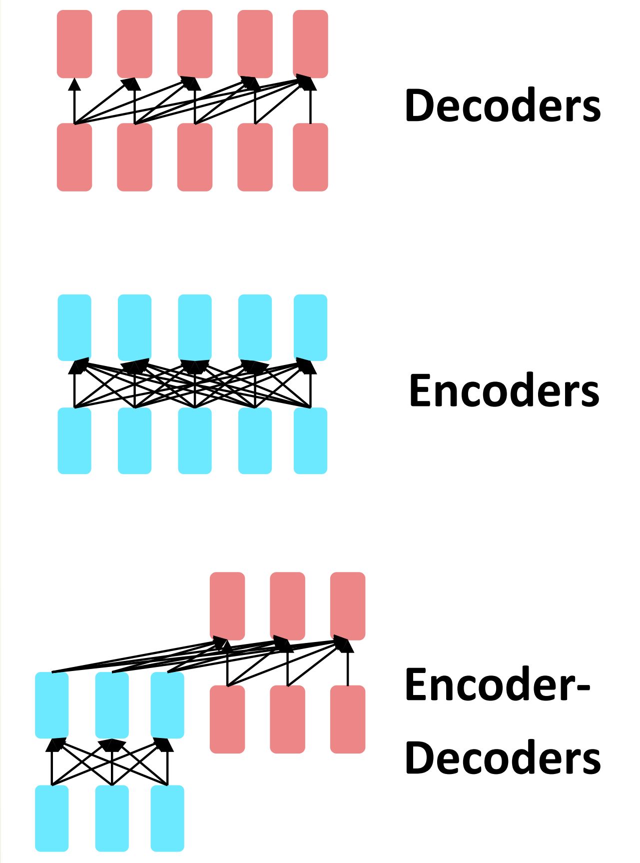
- Peters et al. (2018) Deep Contextual Word Embeddings, NAACL 2018. https://arxiv.org/abs/1802.05365
- Learn a deep Bi-NLM and use all its layers in prediction





Pretraining for three types of architectures in Transformers

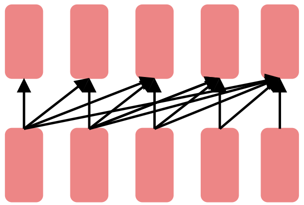
The transformer architecture influences the type of pretraining:



- Decoders:
  - **Unidirectional** Language models! What we've seen so far.
  - Nice to generate from; can't condition on future words
- Encoders:
  - Gets **bidirectional** context – 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-decoder

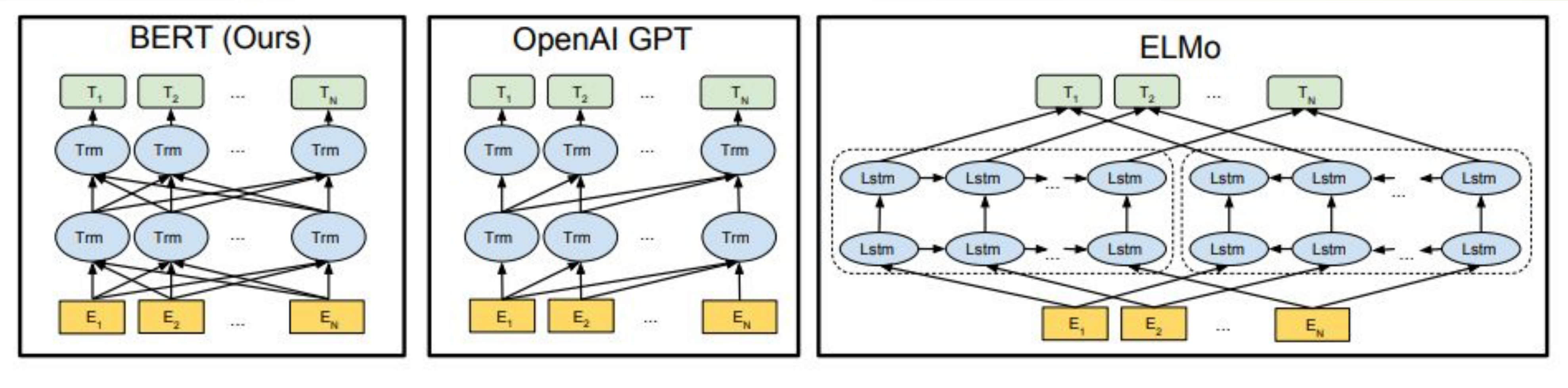
- *Improving Language Understanding by Generative Pre-Training*, OpenAI, 2018
- 2018's GPT was a big success in pretraining a decoder!
- Transformer decoder with 12 layers.
- 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers.
- Byte-pair encoding with 40,000 merges
- Trained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



Decoders

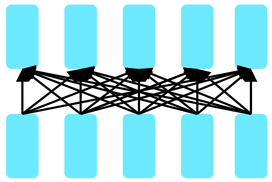
- Language models! What we've seen so far.
- Nice to generate from; can't condition on future words

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%

```
                            store                   gallon
                              ↑                       ↑
the man went to the [MASK] to buy a [MASK] of milk
```

- 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

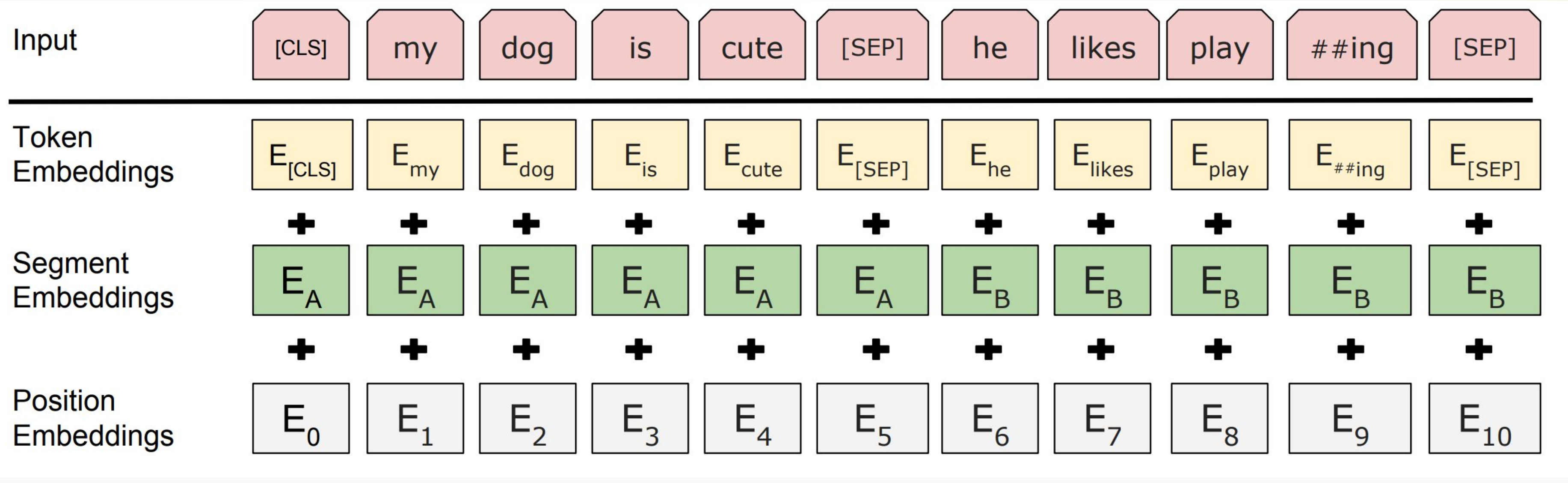
- To learn *relationships* between sentences, predict whether Sentence B is actual sentence that proceeds Sentence A, or a random sentence

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

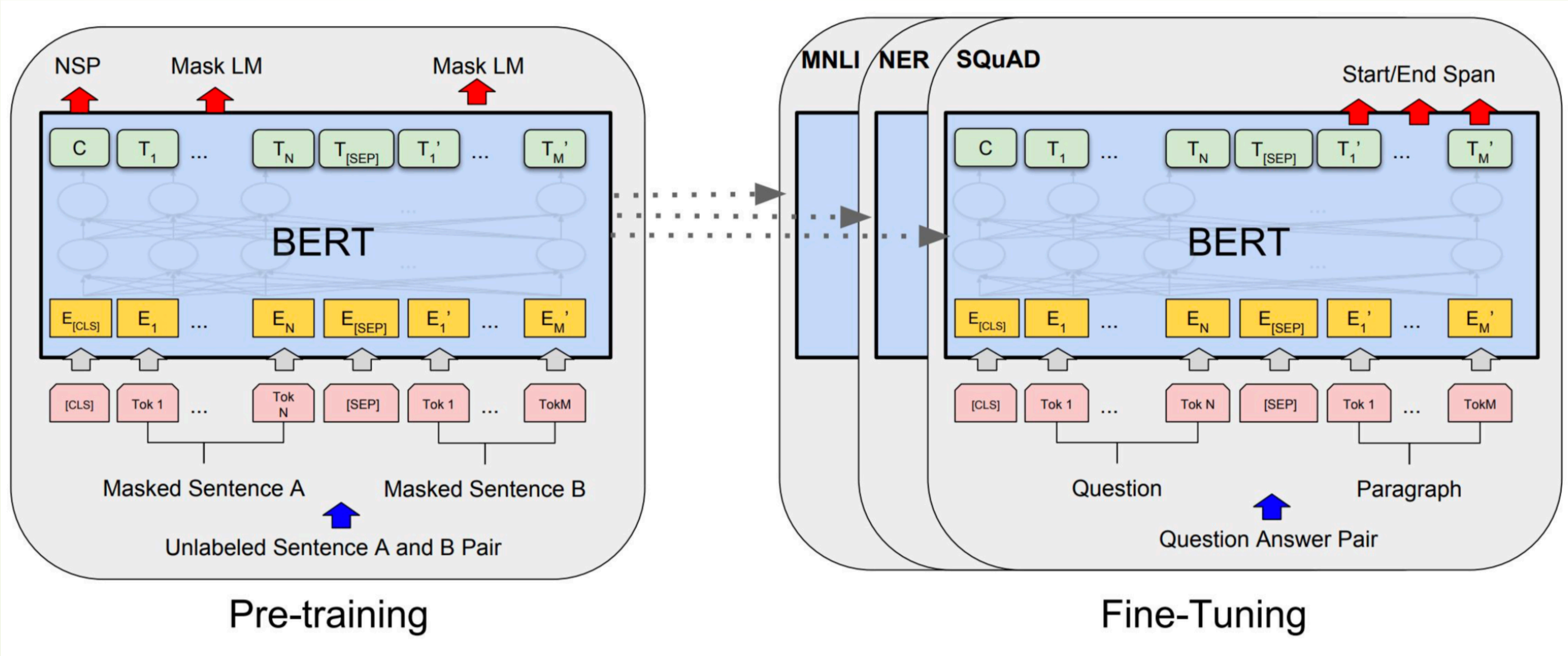


Training

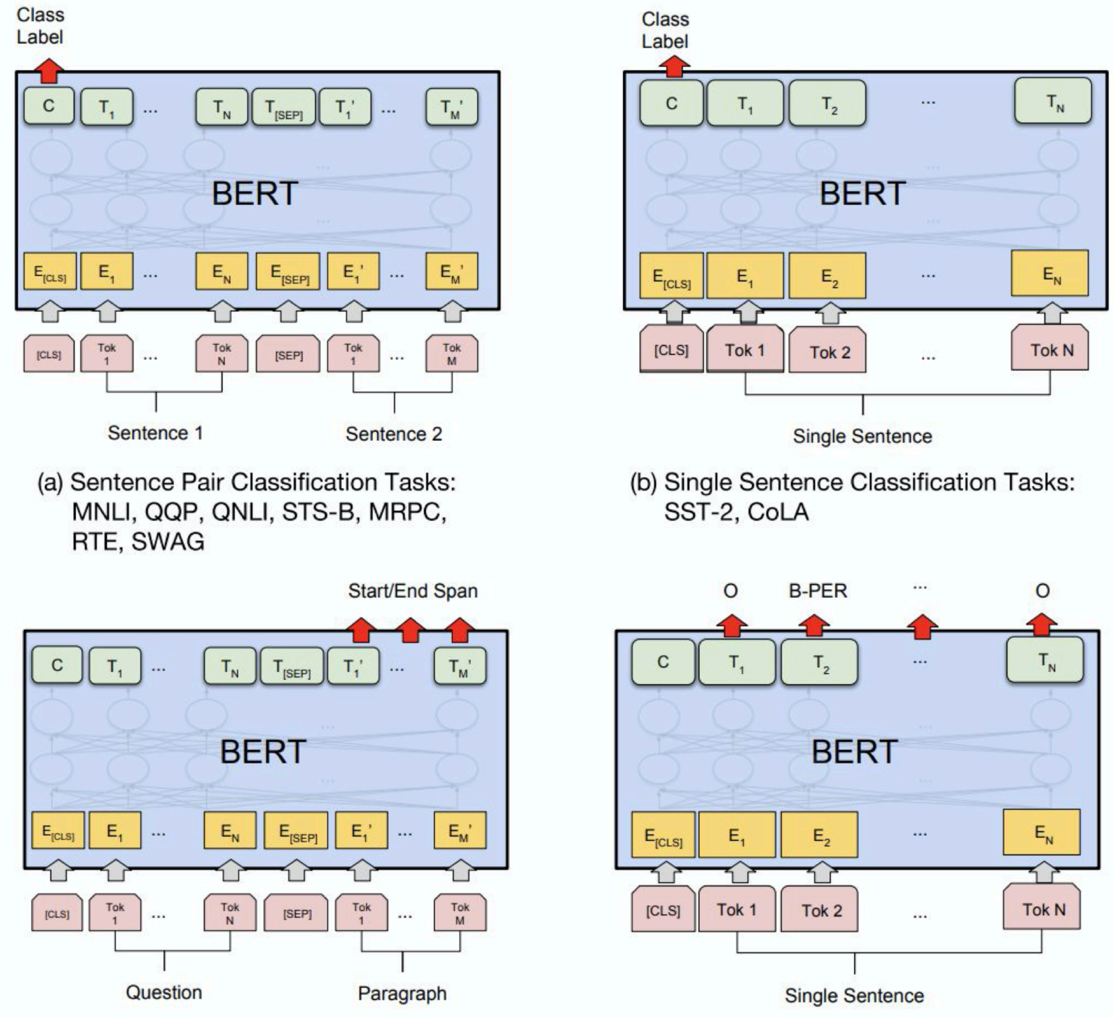
- 2 model released:
  - BERT-Base: 12-layer, 768-hidden, 12-head, 110 million params.
  - BERT-Large: 24-layer, 1024-hidden, 16-head, 340 million params.
- Training Data:
  - BookCorpus (800M words)
  - English Wikipedia (2.5B words)
- Batch Size: 131,072 words
  - (1024 sequences * 128 length or 256 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



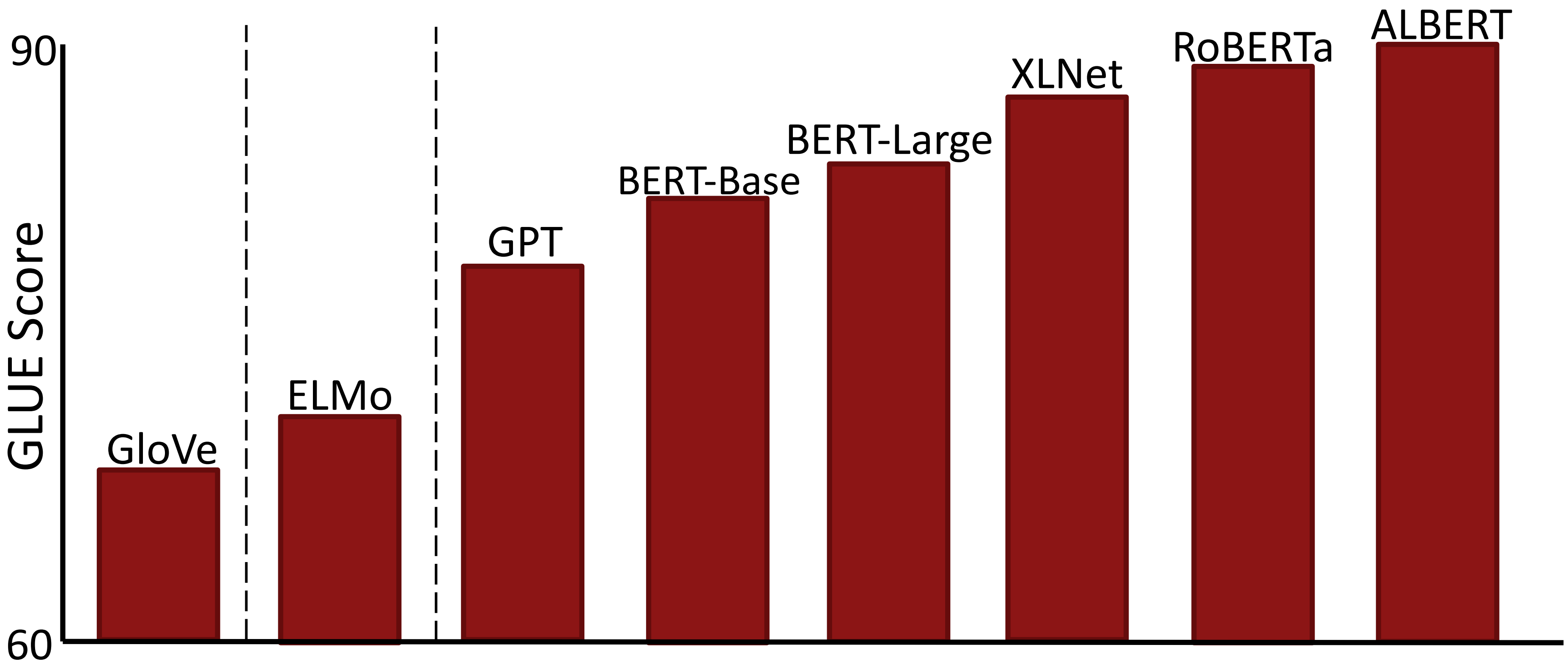
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

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

(c) Question Answering Tasks: SQuAD v1.1

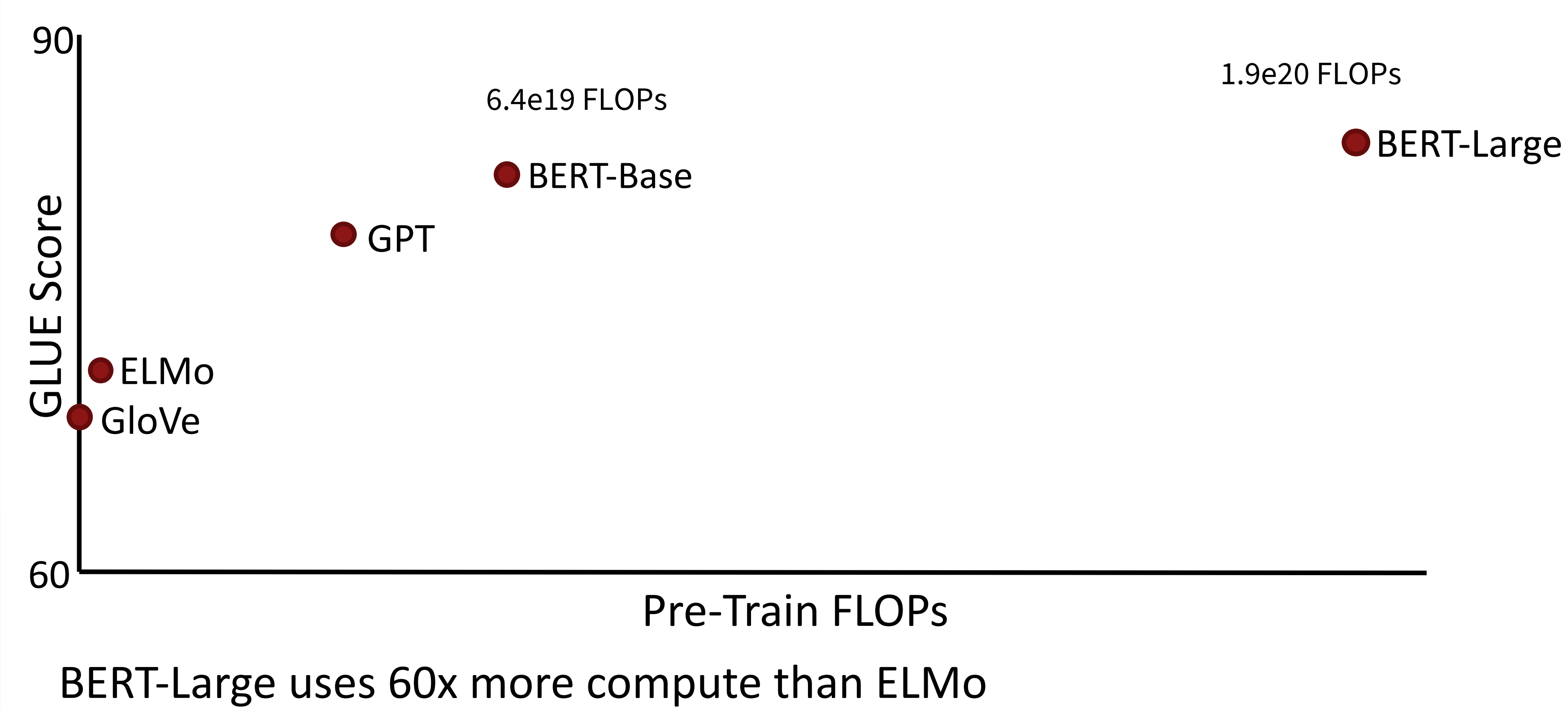
(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

Rapid Progress for Pre-training (GLUE Benchmark)

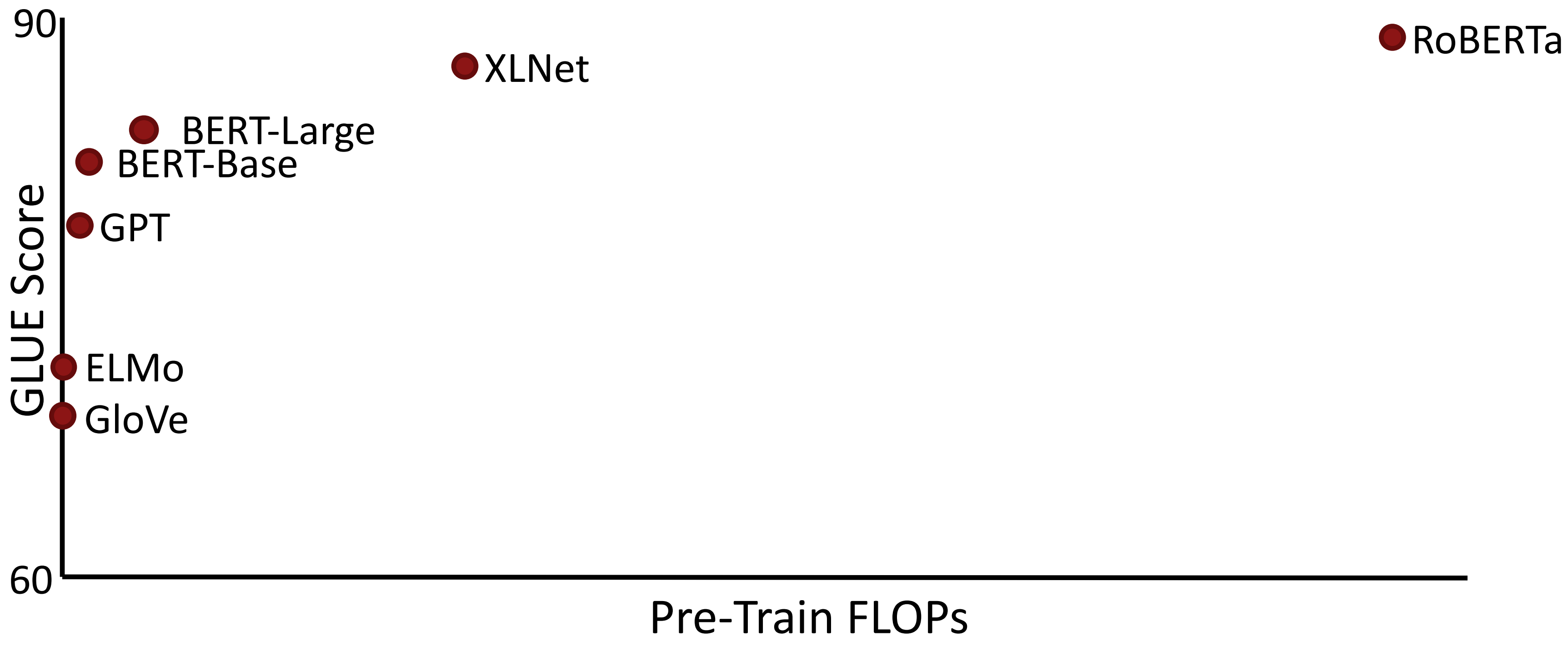


Over 3x reduction in error in 2 years, “superhuman” performance

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

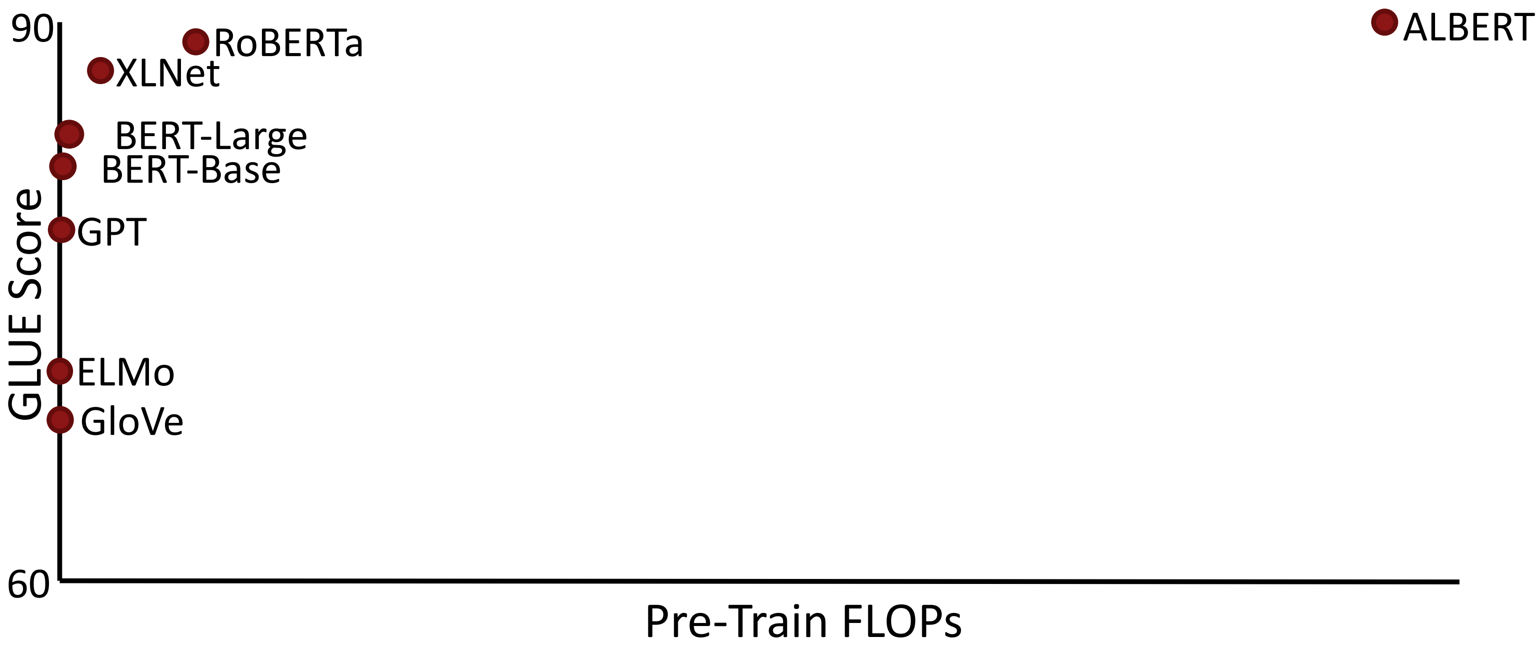


But let's change the x-axis to computational cost…



RoBERTa uses 16x more compute than BERT-Large

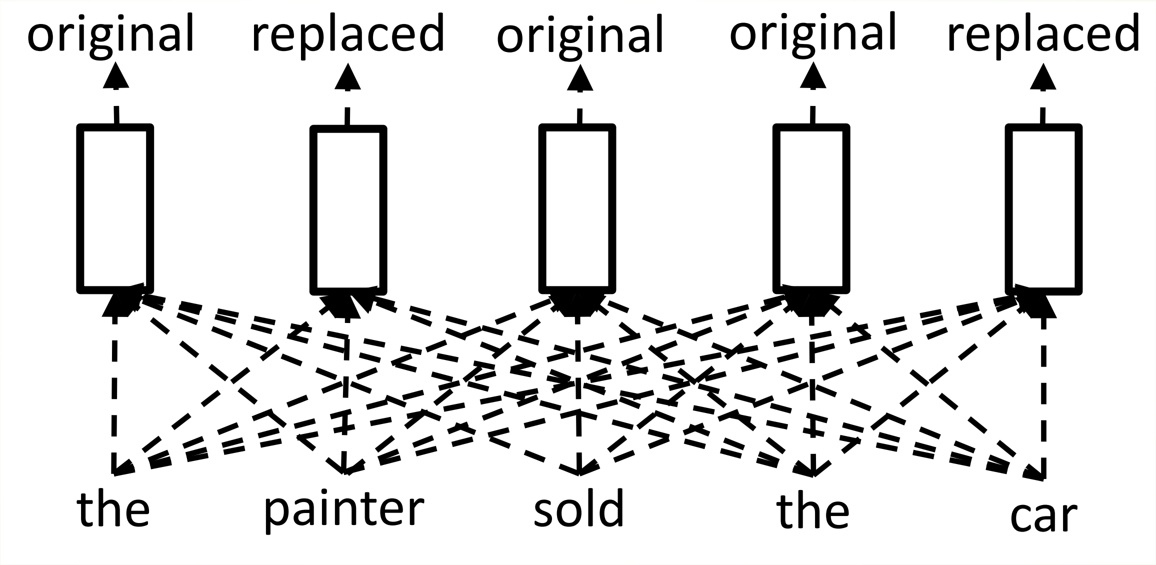
More compute, more better?



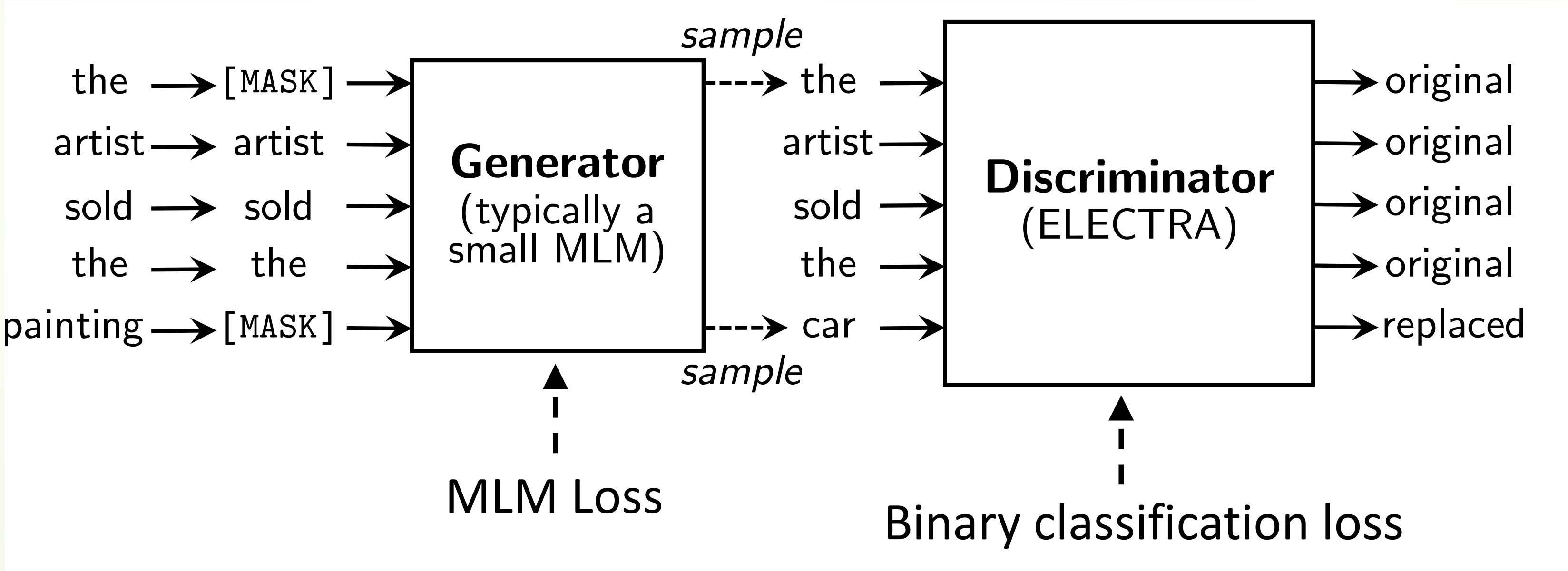
ALBERT uses 10x more compute than RoBERTa

ELECTRA: "Efficiently Learning an Encoder to Classify Token Replacements Accurately"

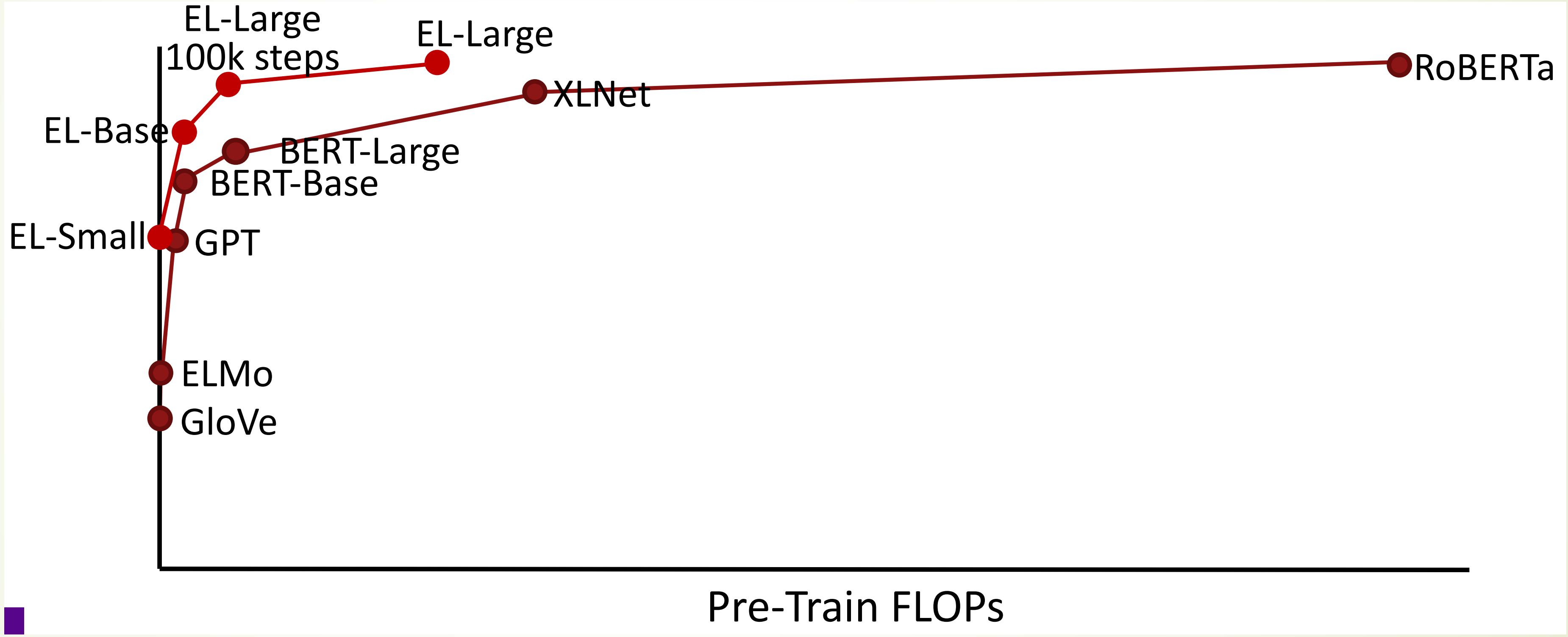
- Clark, Luong, Le, and Manning, ICLR 2020. https://openreview.net/pdf?id=r1xMH1BtvB
- Bidirectional model but learn from all tokens



Generating Replacements

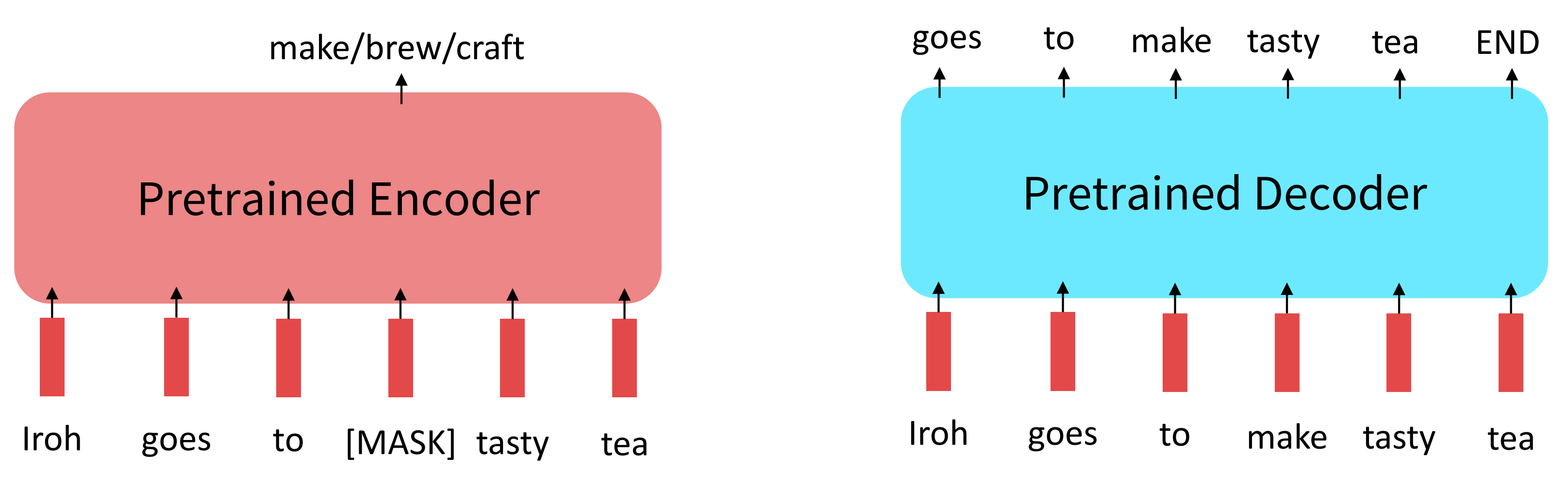


Results: GLUE Score vs Compute



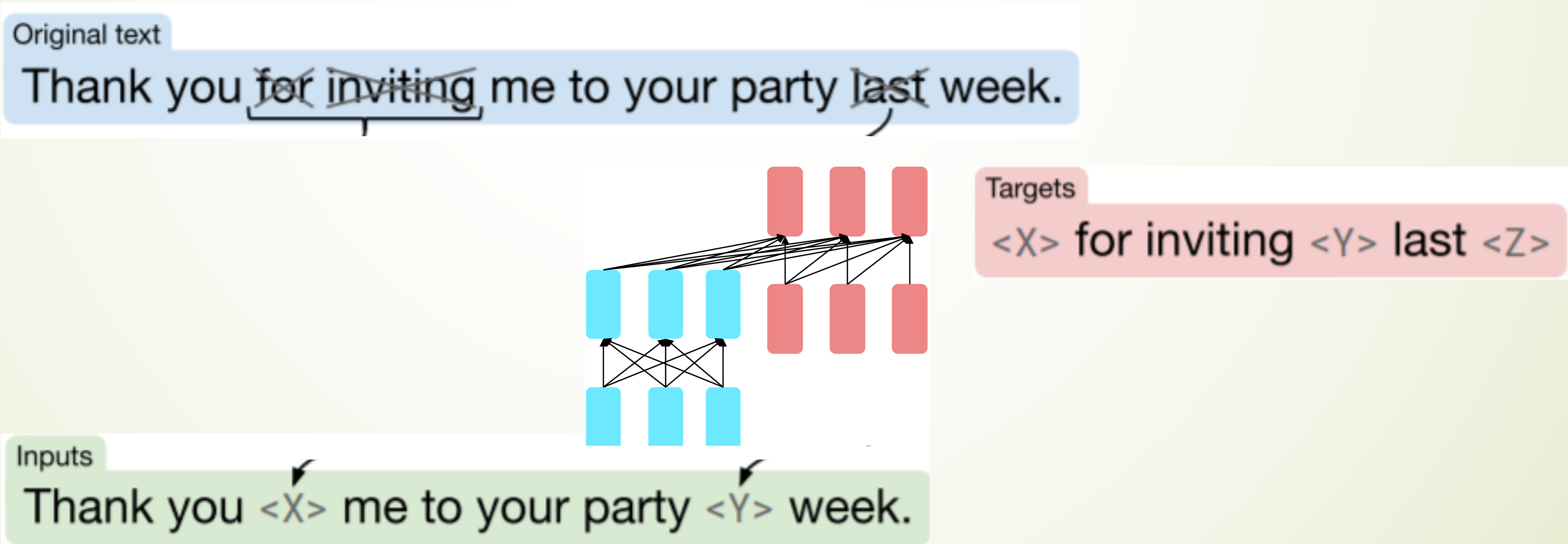
Limitations of Pretrained Encoders

- 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.

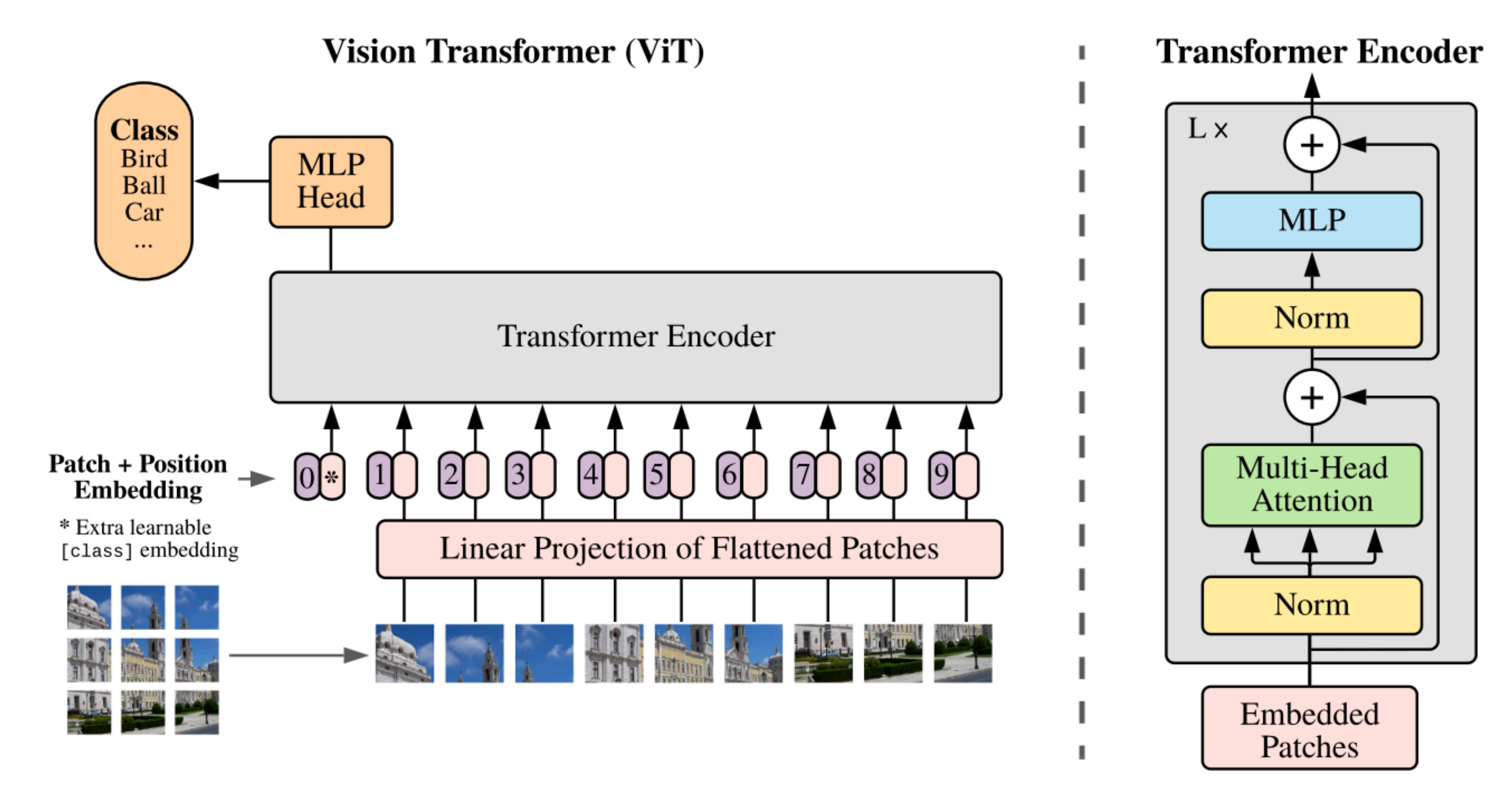


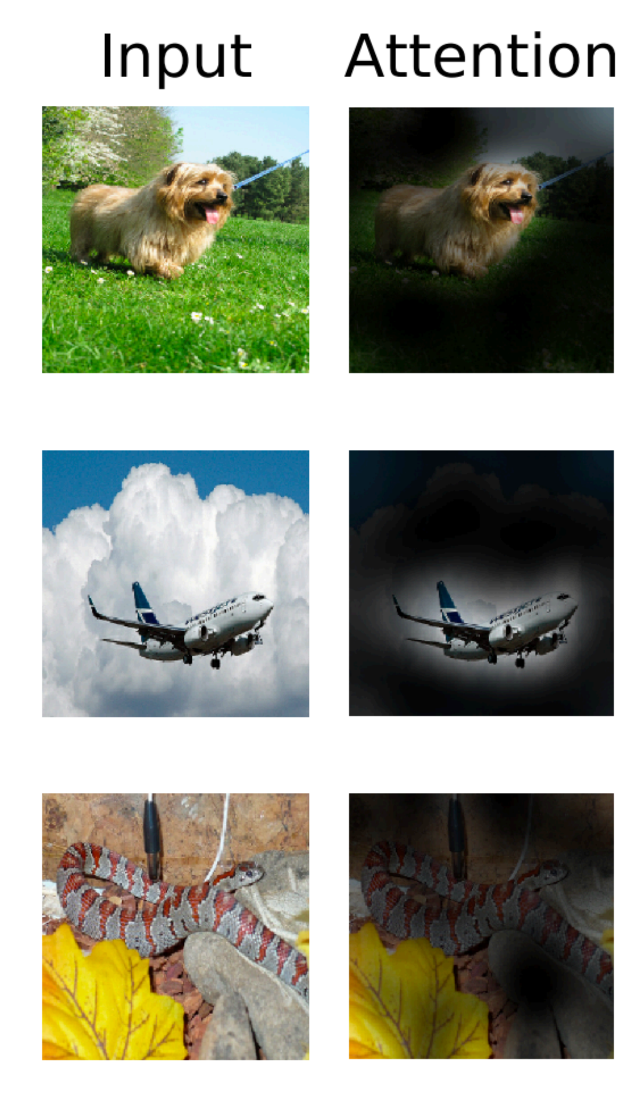
GPT-3, In-context learning, and very large models

- 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.**

Transformers for Vision

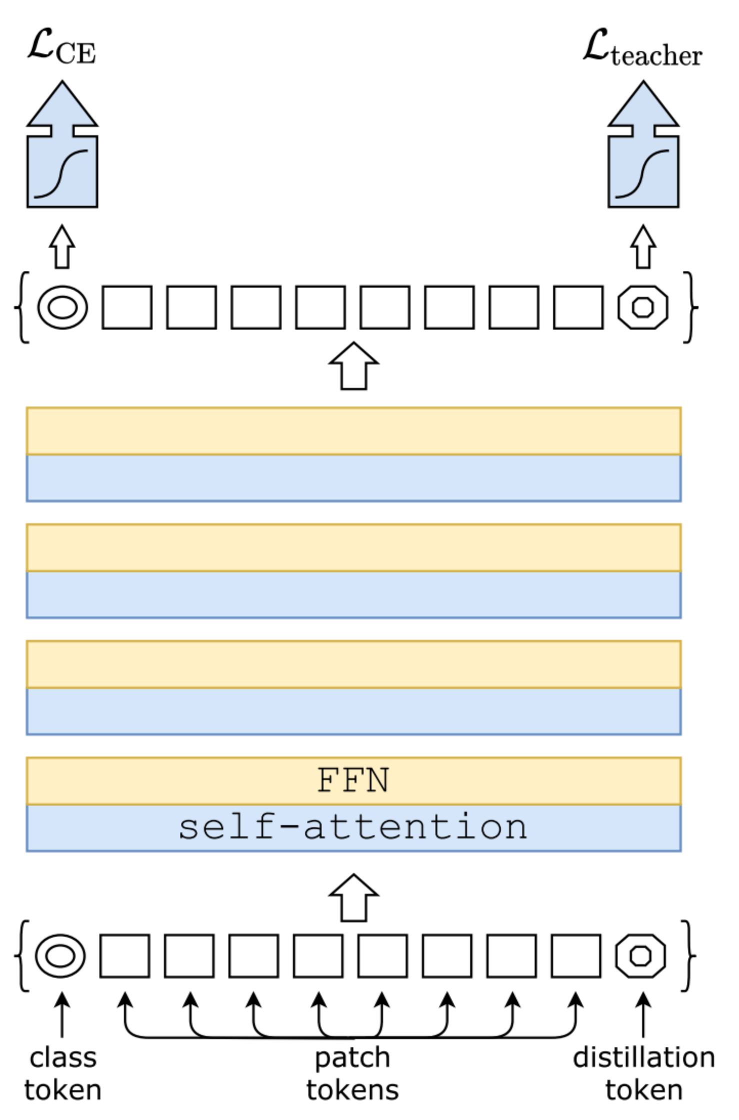
Vision Transformer (VIT)



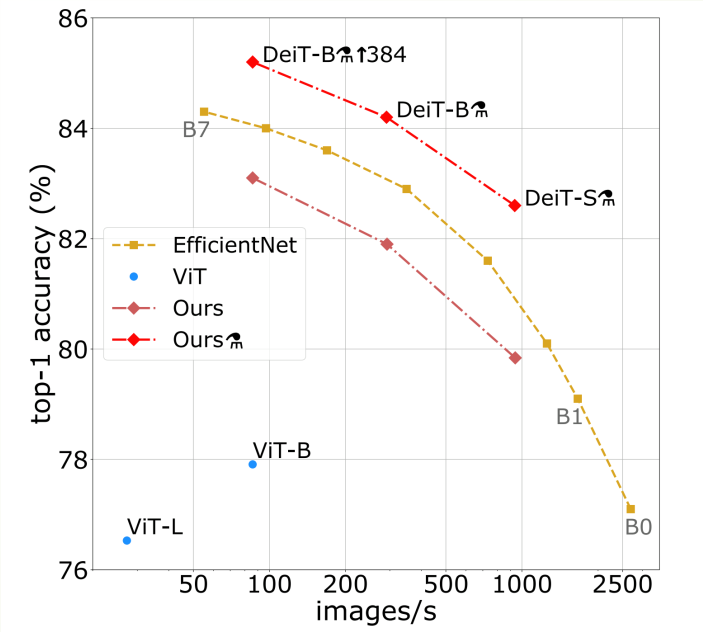


Dosovitskiy et al. ICLR 2021

Data-efficient image transformer (DeiT)

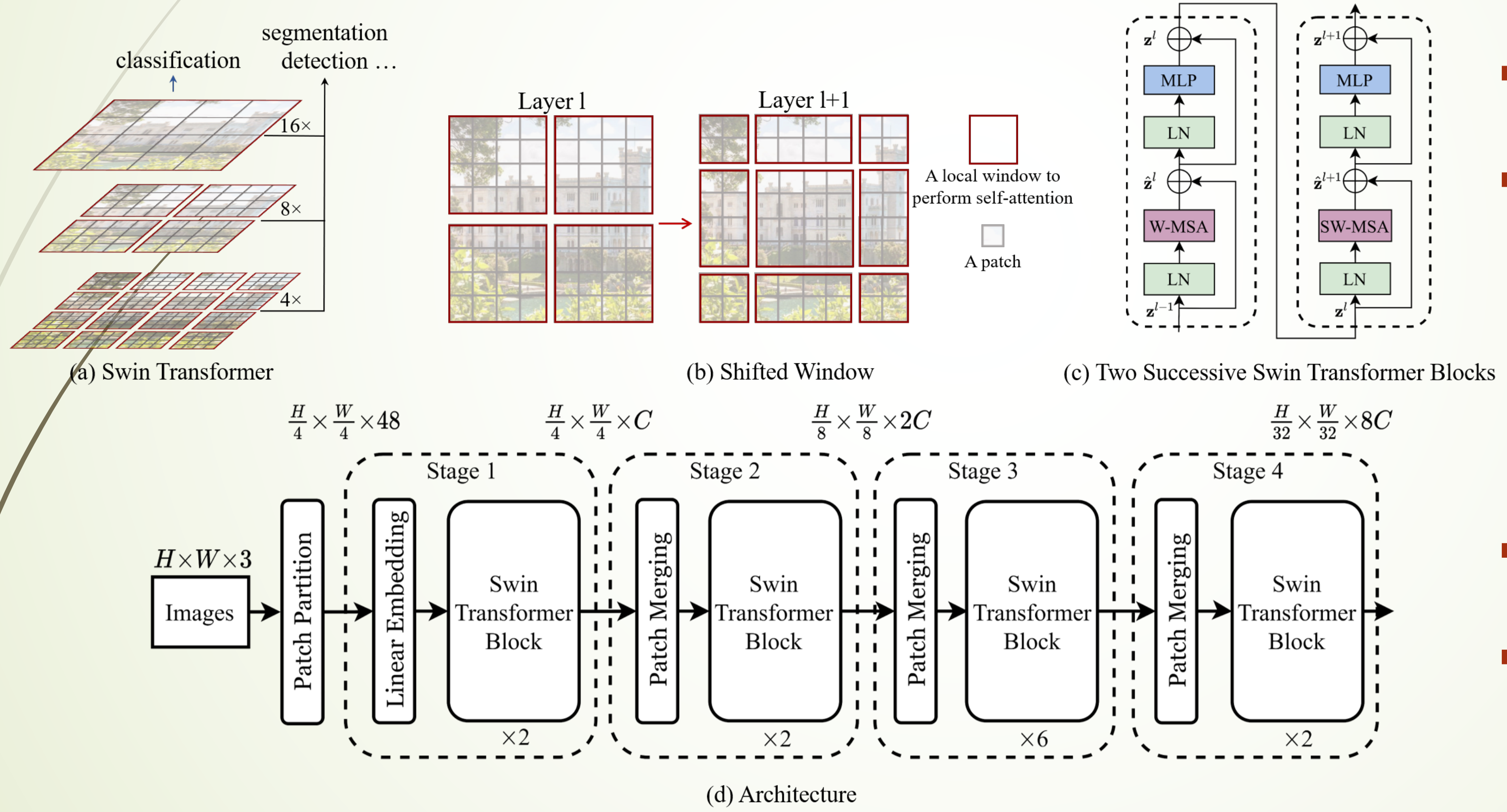


- DeiT introduce ViT a distillation mode with a teacher network, and a richer data augmentation.



Touvron et al. arXiv: 2012.12877 (Jan 2021)

Swin Transformer: Hierarchical Vision Transformer using **S**hifted **Win**dows



(a) Swin Transformer (b) Shifted Window (c) Two Successive Swin Transformer Blocks (d) Architecture

- Multiscale shifted windows as patches
- Correlation (attention) across scales

- Liu et al. ICCV 2021 (Marr award)
- https://github.com/microsoft/Swin-Transformer

Thank you!