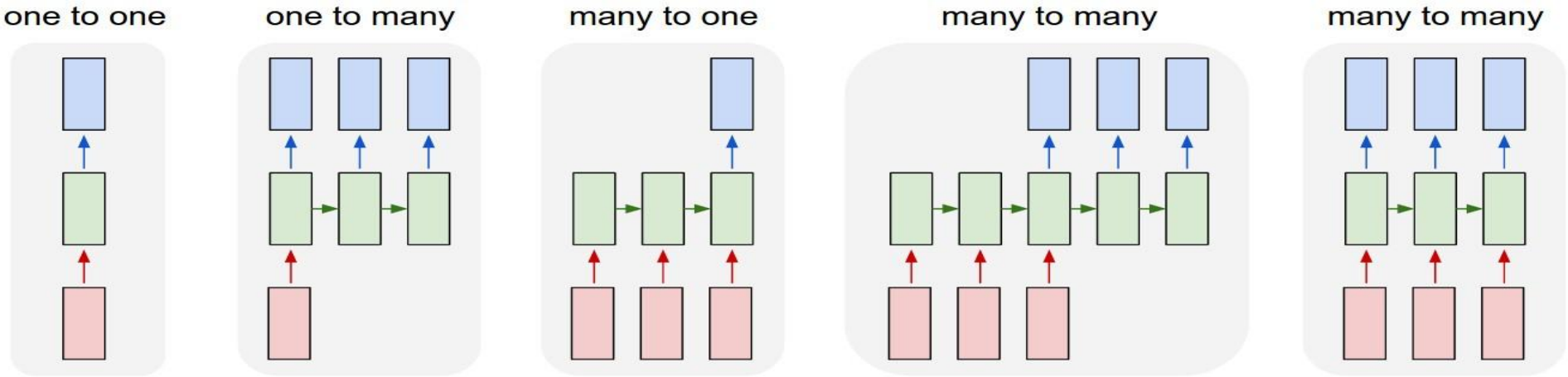
Sequence2Sequence models

Dr. Ahmad El Sallab
AI Senior Expert

Agenda

- **What are seq2seq?**
- Why seq2seq?
- Matched vs Unmatched seq2seq
  - Aligned/Unaligned
- Neural Machine Translation (NMT)
  - Training data: Teacher forcing
  - Basic seq2seq with LSTM
  - Masking
  - Seq2seq + Attention
  - Decoding
  - Word vs Char level
- Convs2s
- Transformers

Sequence models



What makes RNN special vs. DNN/CNN?

- One-One: Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification).
- One-many: Sequence output (e.g. image captioning takes an image and outputs a sentence of words).
- Many-one: Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment).
- Many-Many: Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French).
  - Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified

NLP models meta-architectures

Just like in CV: **Encoder-Decoder**

- **Seq2Class:**
  - Encoder = words vectors aggregation **(How?)**
  - Decoder = None (just classifer=softmax)
  - Analogy to CV: Encoder-Softmax (AlexNet, VGG,...etc)
- **Seq2Seq:**
  - Encoder = words vectors aggregation **(How?)**
  - Decoder = multiple words generation **(How?)**
  - Analogy to CV: Encoder-Decoder in semantic segmentation. But in SS, we have aligned many2many, while in NLP, we have unaligned sequences→ challenge in annotation, model, when to stop, position encoding...etc
- **Word vectors are the input to all the above meta-architectures:**
  - Unlike in CV, where pixels are already digitized
  - Also, in NLP **order matters! = Context**

Why?

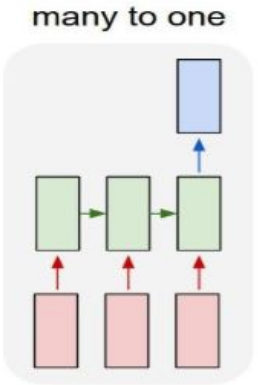
First app→ Language Models

- Predict the **next word**
- Some practical apps: auto-complete
- Indirect apps: TL scenario→ pre-train NLM → transfer to other higher level tasks: classification, NMT, Chatbot, spell correction,...etc

Read ALL first → Generate next

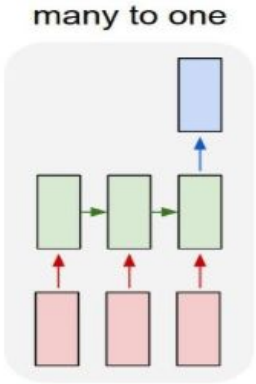
the clouds are in the

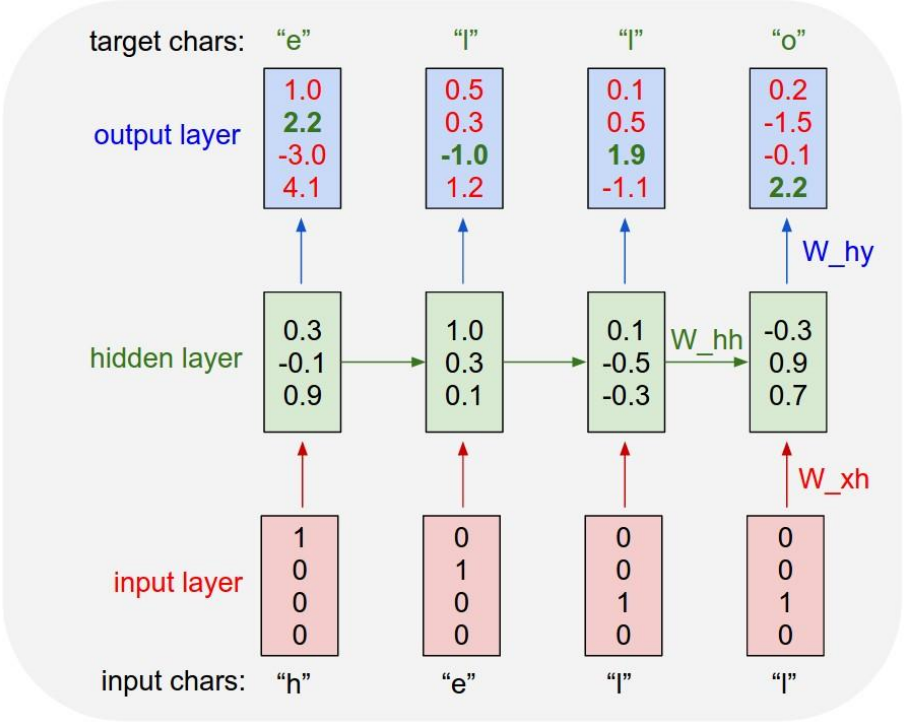
Ground
Garage
Sky
Airport
Oven



Why?

Read ALL first → Generate next

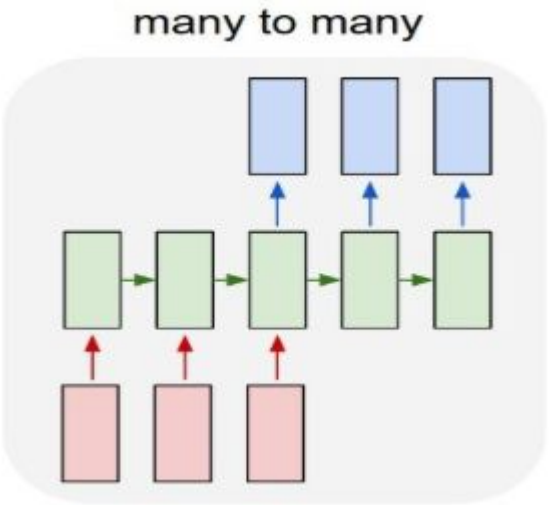




What if we want to predict longer horizon of words/chars?

Why?

Read ALL first → Generate next



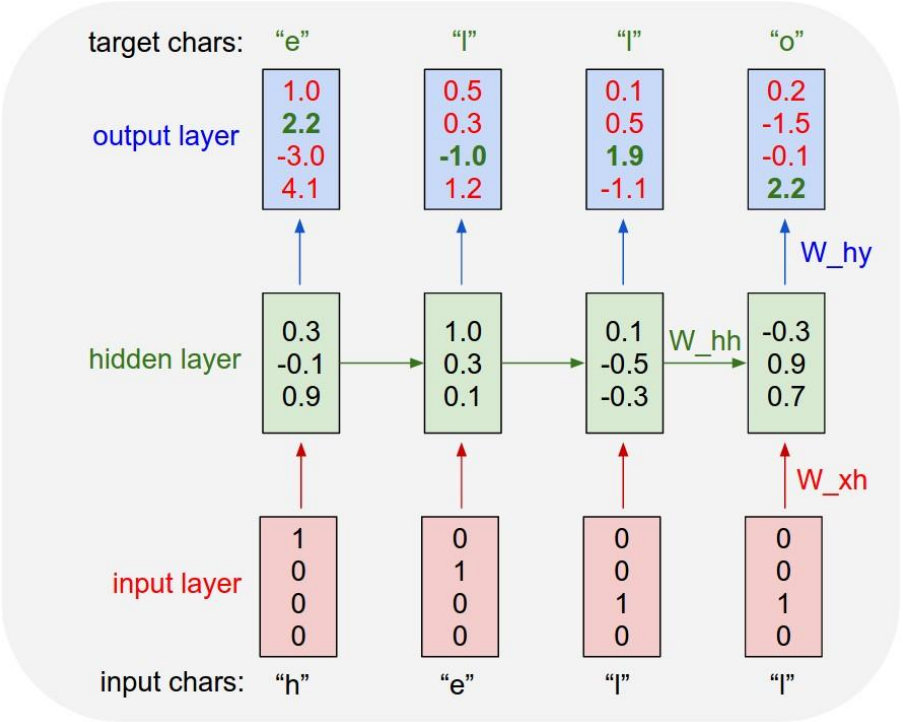
What if we want to predict longer horizon of words/chars?

Why?

Read ALL first → Generate next

First app→ Language Models

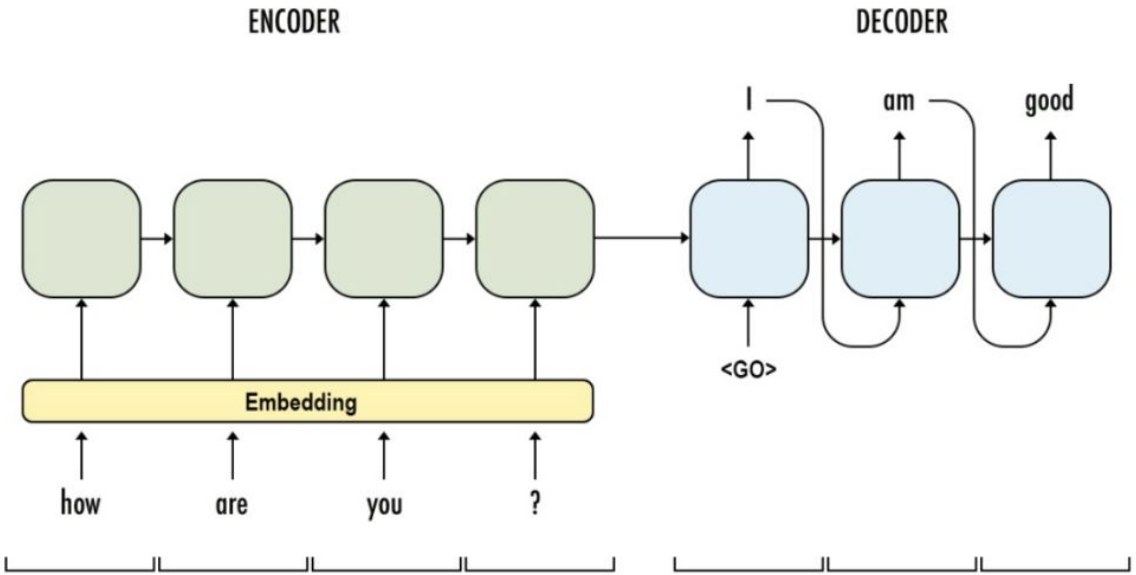
- Predict the next word
- Some practical apps: auto-complete
- Indirect apps: TL scenario→ pre-train NLM → transfer to other higher level tasks: classification, NMT, Chatbot, spell correction,...etc



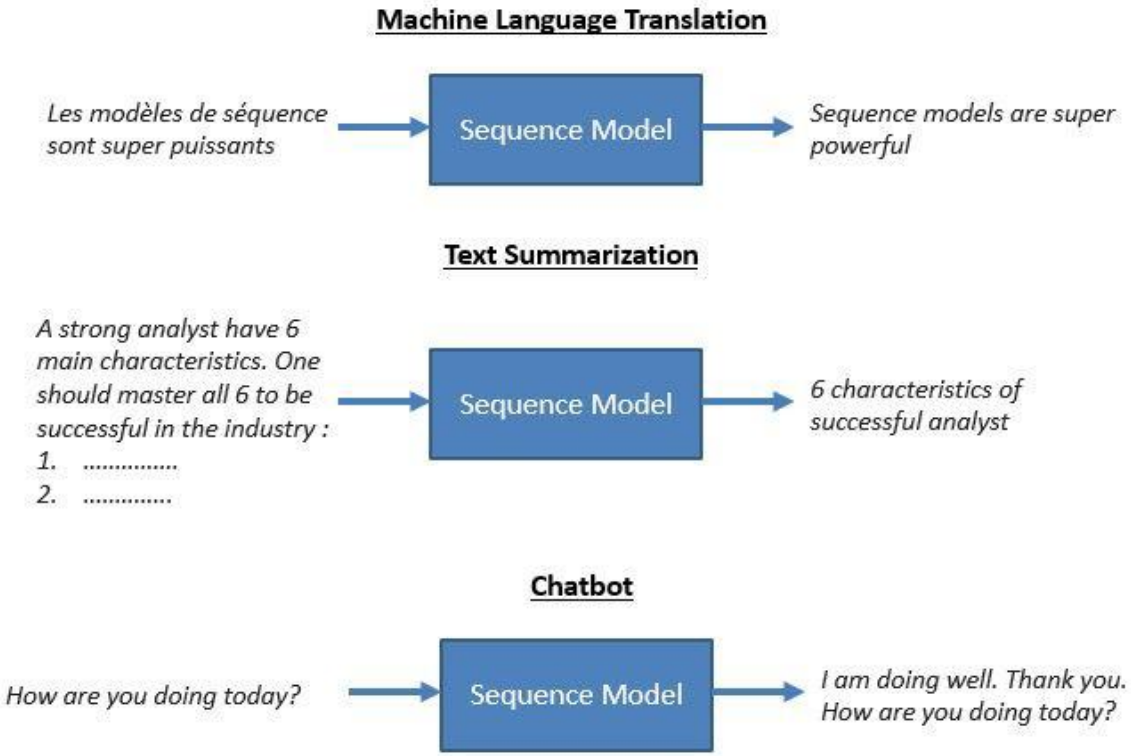
What if we predict longer horizon of words/chars?

Example: Chatbots/QA

Understand the intent first, then reply!



Many others


Machine Language Translation
Les modèles de séquence sont super puissants
Sequence Model
Sequence models are super powerful
Text Summarization
A strong analyst have 6 main characteristics. One should master all 6 to be successful in the industry :
1. .....
2. .....
Sequence Model
6 characteristics of successful analyst
Chatbot
How are you doing today?
Sequence Model
I am doing well. Thank you. How are you doing today?

Seq2Seq is general meta architecture

Unaligned many-many:

NMT

Spelling correction

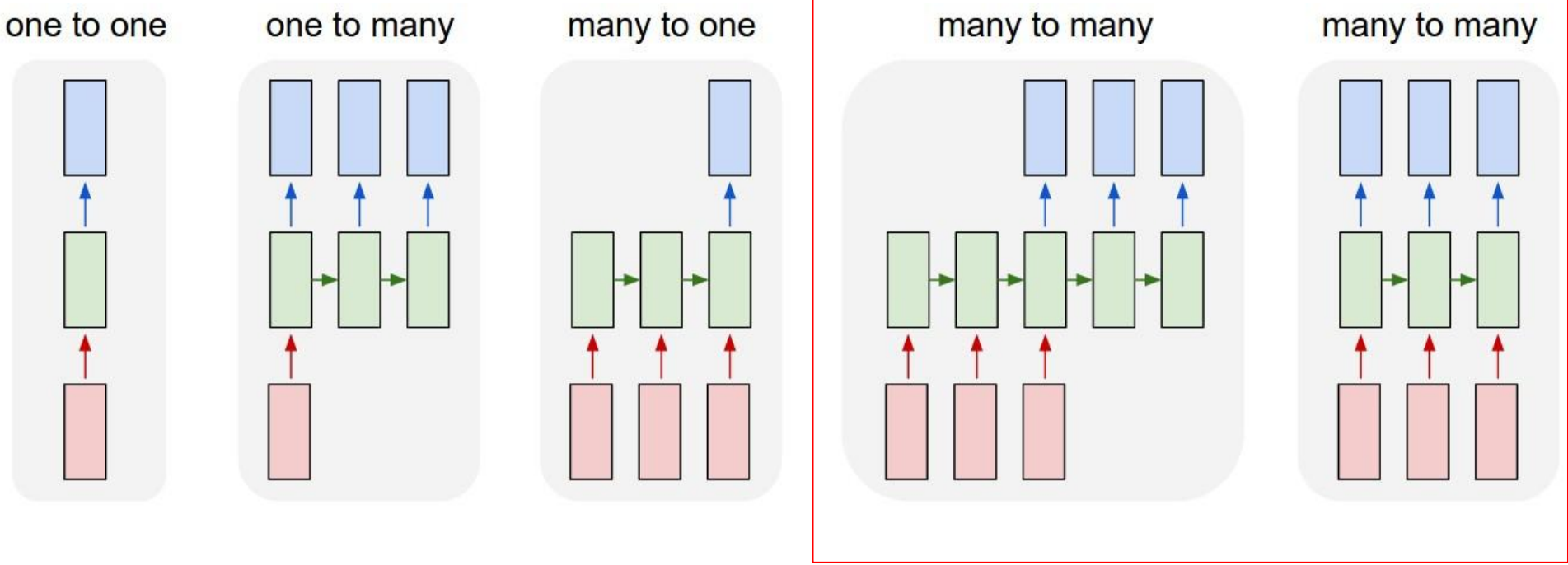
Speech

OCR

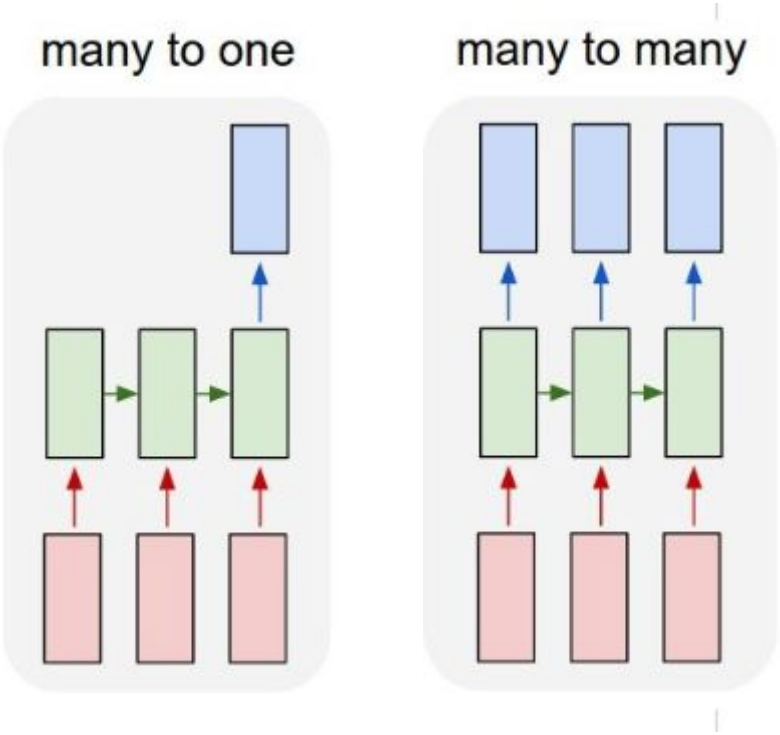
Chatbots

Q&A

Matched vs. Unmatched seq2seq



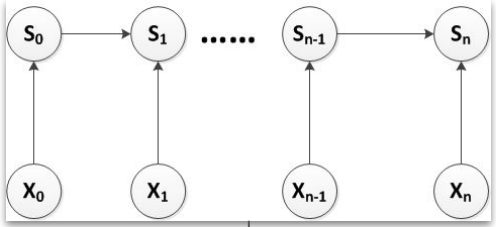
Matched/Aligned case

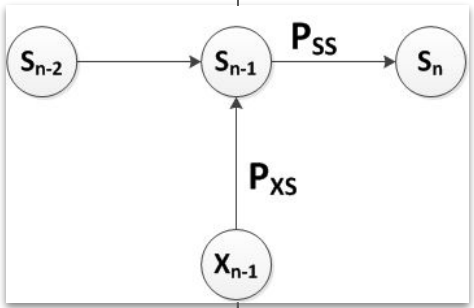


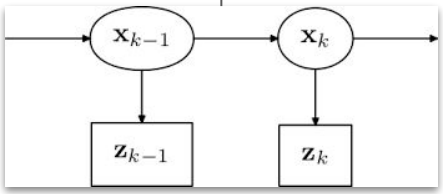
Matched case solutions

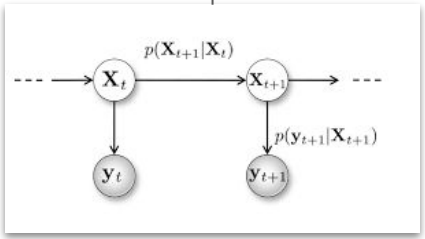
Bayes rule works!
X is directly linked to S ⇒ We can have Pxs=P(S|X) and Pss=P(Sn|Sn-1)

- Bayes filter as recursive state estimation
- Based on Markov Assumption
- Two pdfs: transition + Emission
- Continuous state⟶ Kalman filter
- Discrete state ⟶ HMM



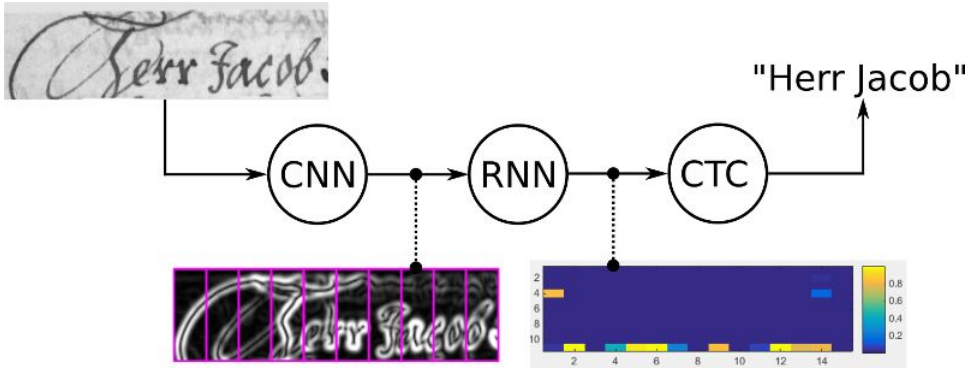


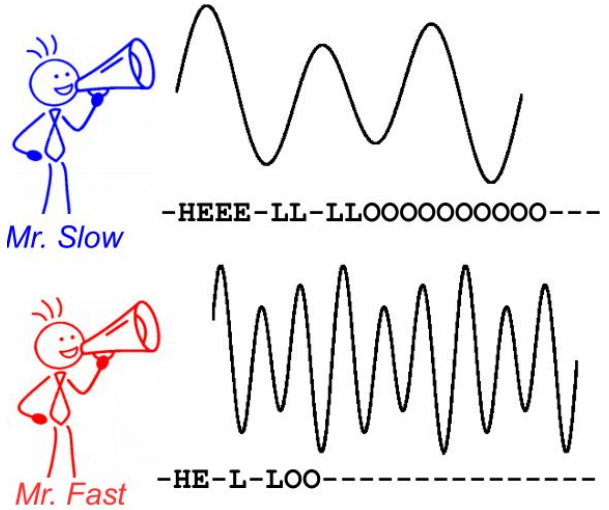




Why we might have Matched/Unaligned cases?

- Some apps has misalignment due to (substitution, deletion, insertion)
  - OCR
  - Speech recognition
  - Spelling correction





Machine Translation

Unmatched/Unaligned

Machine Translation (MT) is the task of translating a sentence *x* from one language (the source language) to a sentence *y* in another language (the target language).

x: *L'homme est né libre, et partout il est dans les fers*

y: *Man is born free, but everywhere he is in chains*

1950s: Early Machine Translation

Machine Translation research began in the early 1950s.

- Russian → English
  (motivated by the Cold War!)

1 minute video showing 1954 MT:
https://youtu.be/K-HfpsHPmvw

- Systems were mostly rule-based, using a bilingual dictionary to map Russian words to their English counterparts

1990s-2010s: Statistical Machine Translation

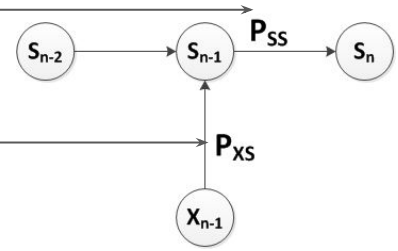
- Core idea: Learn a probabilistic model from data
- Suppose we're translating French → English.
- We want to find best English sentence *y*, given French sentence *x*

$$\text{argmax}_y P(y|x)$$

Form of Bayes Filter!

- Use Bayes Rule to break this down into two components to be learnt separately:

$$= \text{argmax}_y P(x|y)P(y)$$



Translation Model

Models how words and phrases should be translated (*fidelity*). Learnt from parallel data.

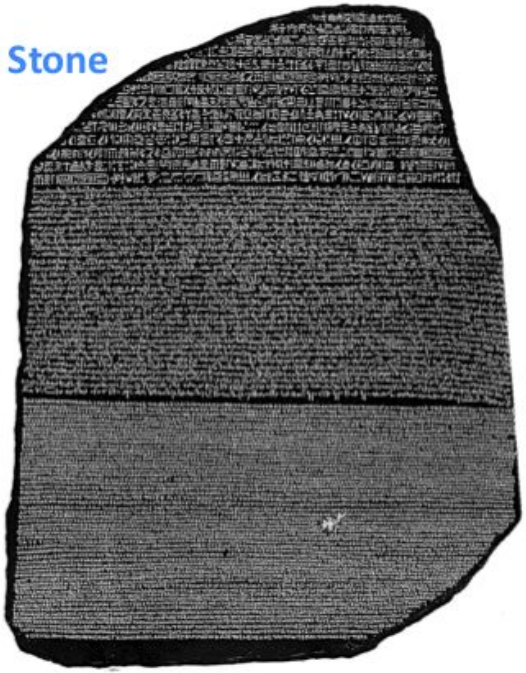
Language Model

Models how to write good English (*fluency*). Learnt from monolingual data.

1990s-2010s: Statistical Machine Translation

- Question: How to learn translation model $P(x|y)$?
- First, need large amount of parallel data
  (e.g. pairs of human-translated French/English sentences)

The Rosetta Stone



Ancient Egyptian

Demotic

Ancient Greek

Form of Bayes Filter! Which x maps to which y?

Learning alignment for SMT

Form of Bayes Filter! Which x maps to which y?

- Question: How to learn translation model $P(x|y)$ from the parallel corpus?

- Break it down further: we actually want to consider

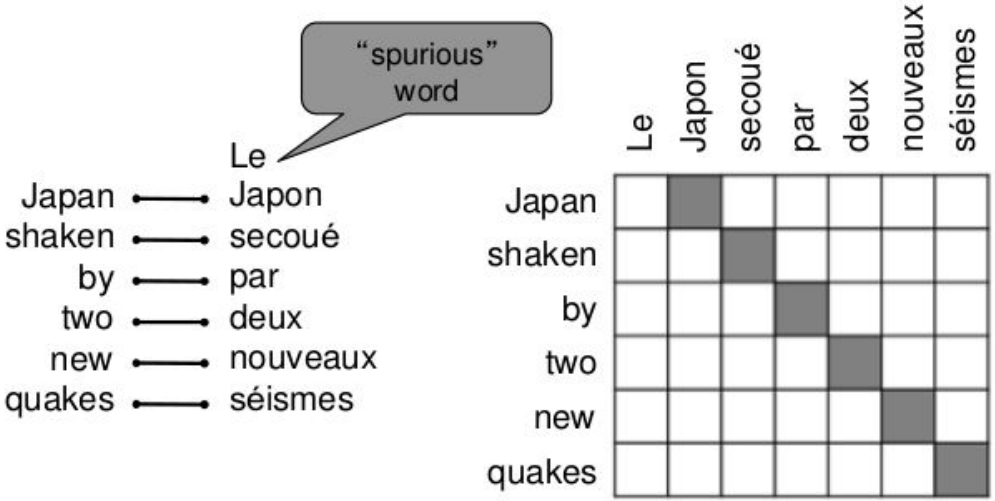
$$P(x, a|y)$$

where *a* is the alignment, i.e. word-level correspondence between French sentence *x* and English sentence *y*

What is alignment?

Alignment is the correspondence between particular words in the translated sentence pair.

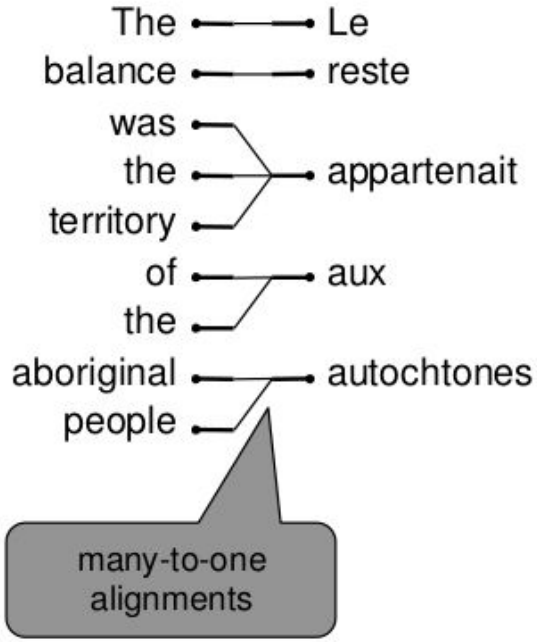
- Note: Some words have no counterpart

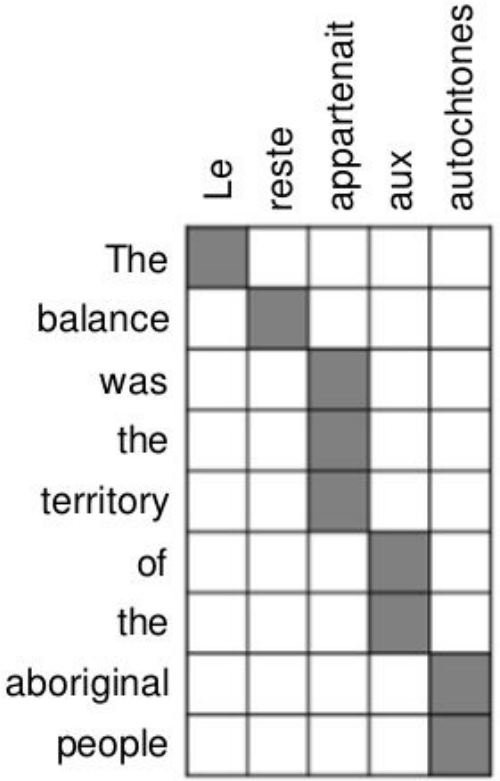


Examples from: "The Mathematics of Statistical Machine Translation: Parameter Estimation", Brown et al, 1993. http://www.aclweb.org/anthology/J93-2003

Alignment is complex

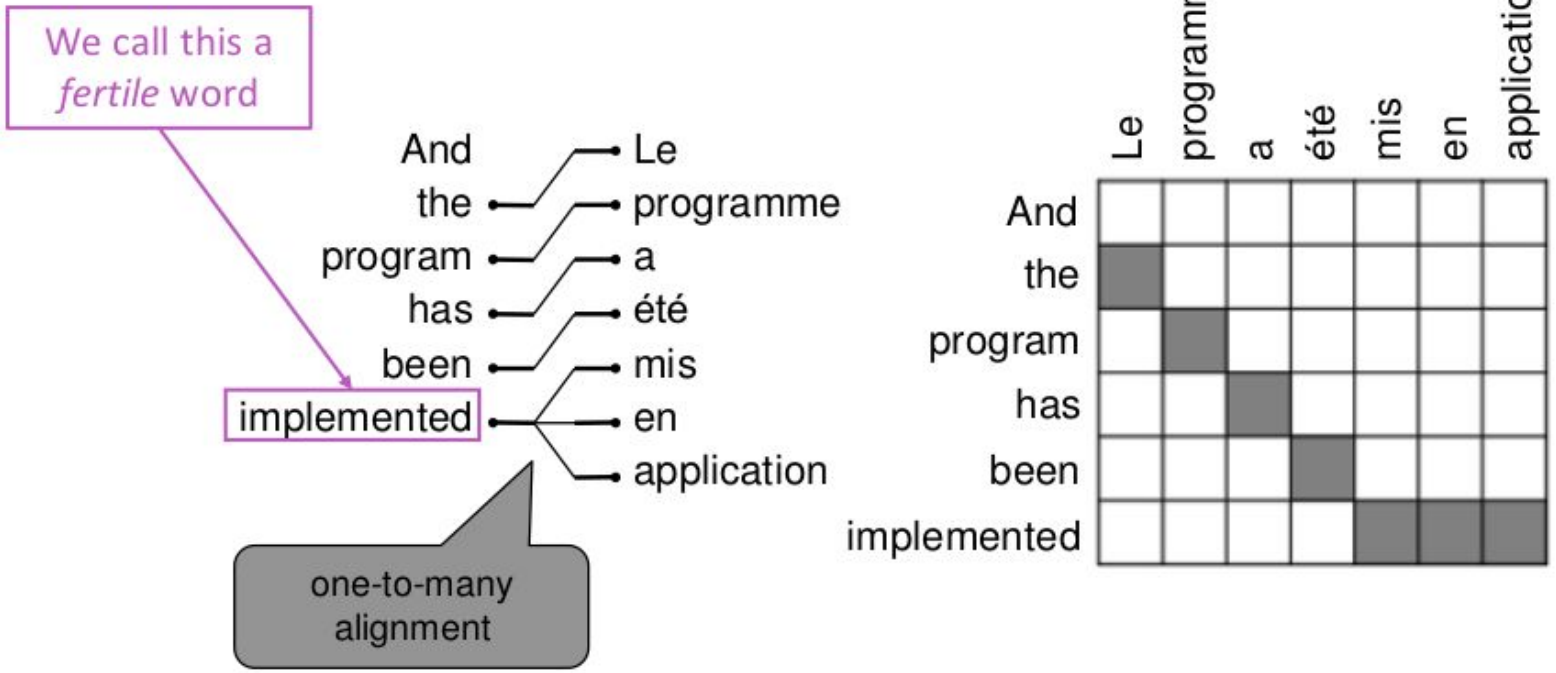
Alignment can be many-to-one





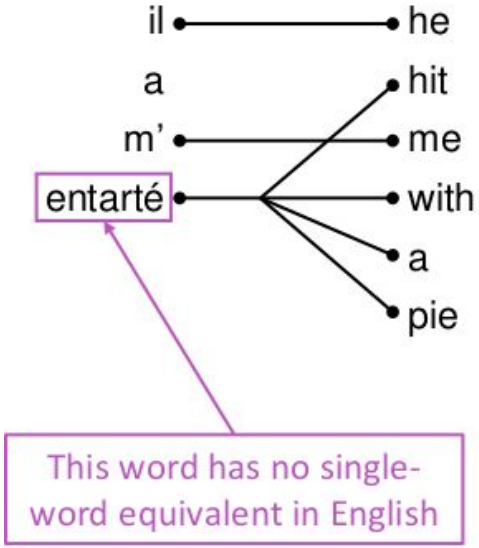
Alignment is complex

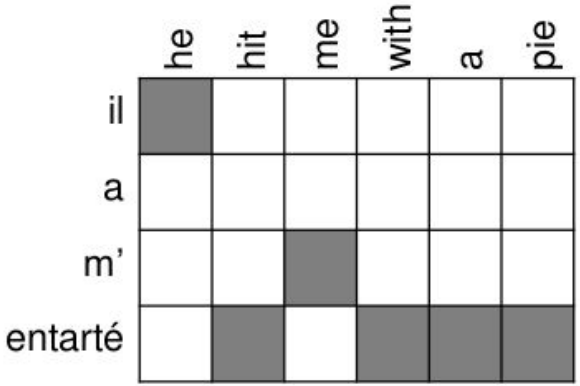
Alignment can be one-to-many



Alignment is complex

Some words are very fertile!

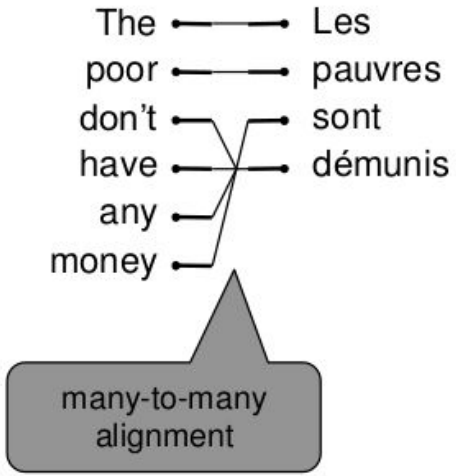


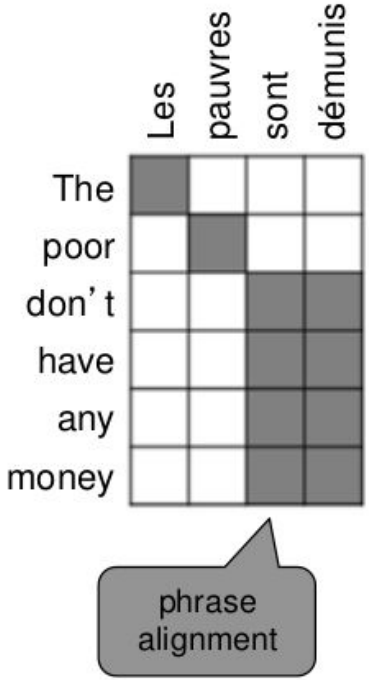


ARABIC!!

Alignment is complex

Alignment can be many-to-many (phrase-level)





Examples from: "The Mathematics of Statistical Machine Translation: Parameter Estimation", Brown et al, 1993. http://www.aclweb.org/anthology/J93-2003

Learning alignment for SMT

- We learn $P(x, a|y)$ as a combination of many factors, including:
  - Probability of particular words aligning (also depends on position in sent)
  - Probability of particular words having particular fertility (number of corresponding words)
  - etc.

Decoding for SMT

Many possible alignments!

$$\text{argmax}_y P(x|y)P(y)$$

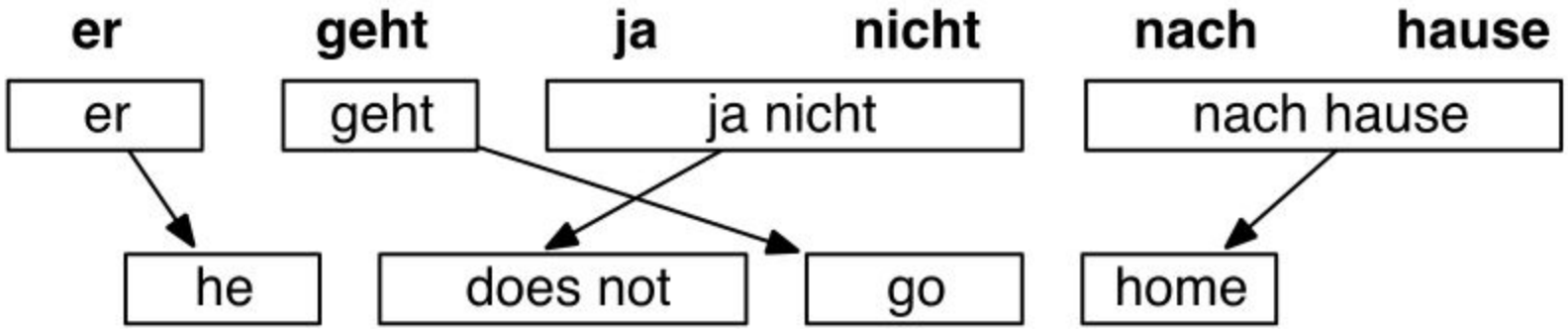
Language Model

Translation Model

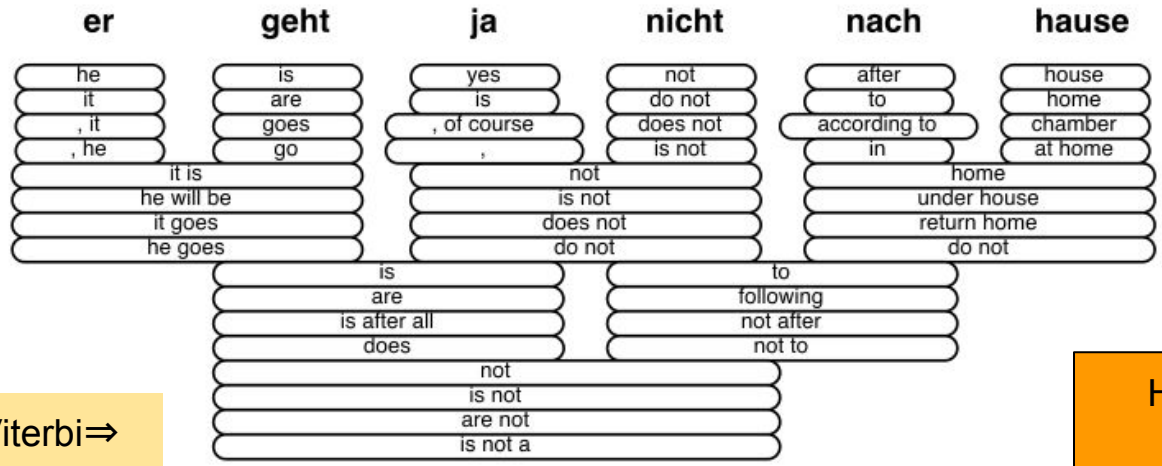
Question: How to compute this argmax?

- We could enumerate every possible *y* and calculate the probability? → Too expensive!
- **Answer:** Use a heuristic search algorithm to search for the best translation, discarding hypotheses that are too low-probability
- This process is called *decoding*

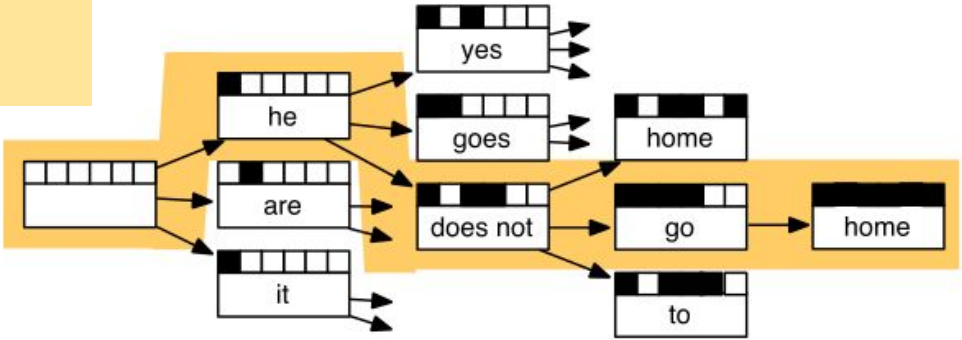
Decoding for SMT



Decoding for SMT



Beam search ⇒ Viterbi⇒ Keep the top-N possibilities at each branch



Huge search space!

Source: "Statistical Machine Translation", Chapter 6, Koehn, 2009.
https://www.cambridge.org/core/books/statistical-machine-translation/94EADF9F680558E13BE759997553CDE5

1990s-2010s: Statistical Machine Translation

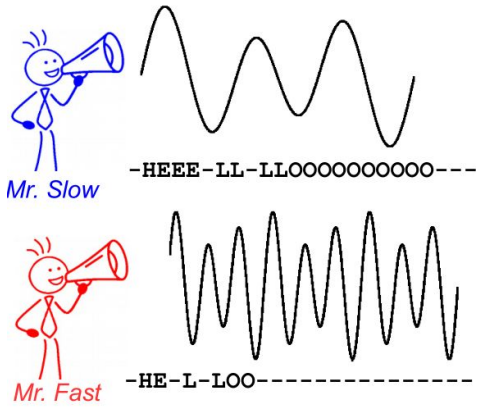
- SMT was a huge research field
- The best systems were extremely complex
  - Hundreds of important details we haven't mentioned here
  - Systems had many separately-designed subcomponents
  - Lots of feature engineering
    - Need to design features to capture particular language phenomena
  - Require compiling and maintaining extra resources
    - Like tables of equivalent phrases
  - Lots of human effort to maintain
    - Repeated effort for each language pair!

Other possible solutions

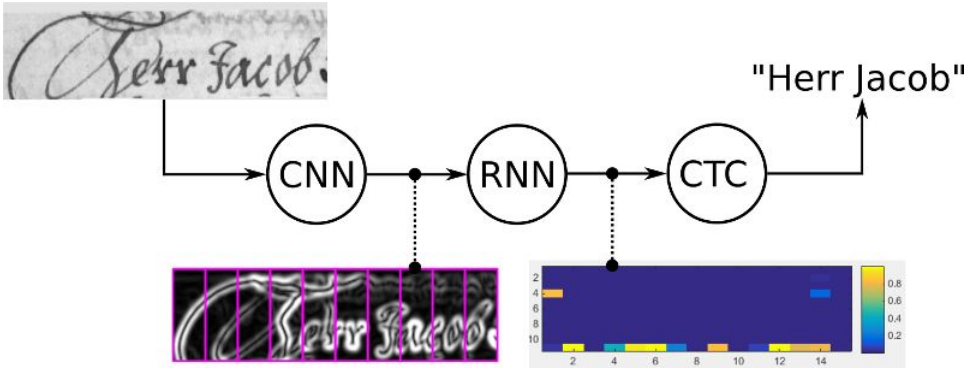
- RNN
- CTC

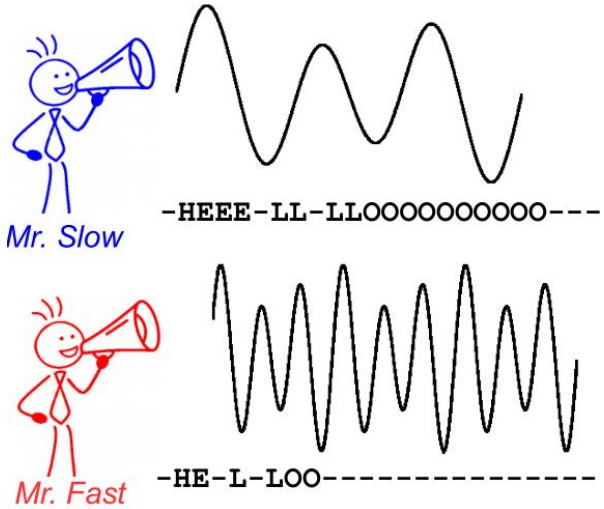
Unaligned-to-Aligned: Connectionist Temporal Classification (CTC)

- Introduce a special character (CTC-blank, denoted as "-" in this text) to indicate that no character is seen at a given time-step
- modify the ground truth text T to T' by inserting CTC-blanks and by repeating characters in **all possible ways**
- we know the image, we know the text, but we don't know where the text is positioned. So, let's just try *all* possible positions of the text "Hi----", "-Hi---", "--Hi--", ...
- we also don't know how much space each character occupies in the image. So let's also try all possible alignments by allowing characters to repeat like "HHi----", "HHHi---", "HHHHi--", ...
- do you see a problem here? Of course, if we allow a character to repeat multiple times, how do we handle *real* duplicate characters like the "l" in "Hello"? Well, just always insert a blank in between in these situations, that is e.g. "Hel-lo" or "Heeellll-------llo"
- calculate score for each possible T' (that is for each transformation and each combination of these), sum over all scores which yields the loss for the pair (I, T)
- decoding is easy: pick character with highest score for each time step, e.g. "HHHHHH-eeeellll-lll--oo---", throw away duplicate characters "H-el-l-o", throw away blanks "Hello", and we are done.



Unaligned-to-Aligned:
Connectionist Temporal Classification (CTC)



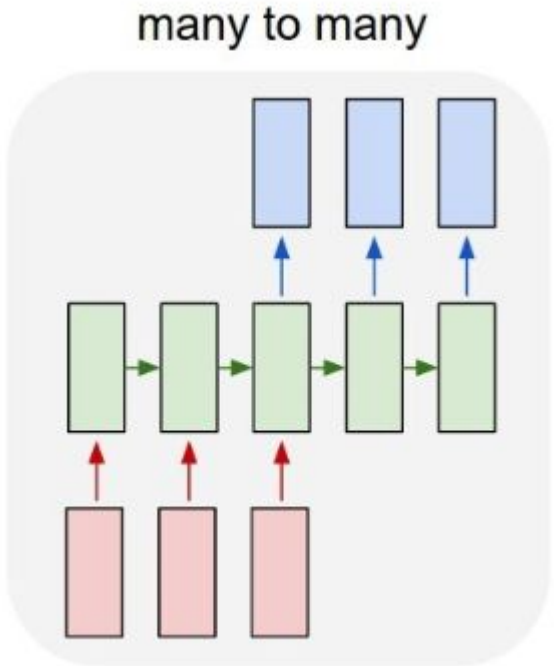


Only when alignment is there, but misaligned by mistakes!
Cannot work with tasks where there is not correspondance like NMT, QA or Chatbots

Unmatched/Unaligned case

Each x can produce many s
Many x can produce one s

- Traditional filtering cannot work, since there's no direct model between Pxs and Pss
- **Recurrent Neural Nets to the rescue**
  - **Ability to have summarizing state in Encoder**
  - **Spit output in separate Decoder**



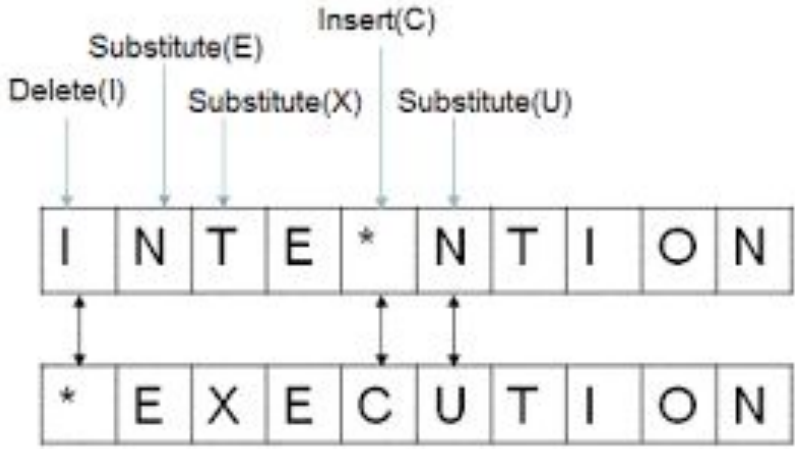
RNNs: separate input X from output S by a "buffer" Encoder

Evaluation of unmatched case

WER

Edit distance (Levenstein)

How to align the 2 sequences?
What are the total “steps” to align = error = **distance**



https://i.pinimg.com/originals/77/65/82/776582769376fa42eeb3cdb898e449d1.gif

There could be more than one “possible” alignment

Evaluation of unmatched case

Ins	_	1
Del	X	1
Sub	#	1

There could be more than one “possible” alignment

Hello
He

Seq 1	Seq 2	Assumed correct	Cost
Hello	He _ _ _	He	3
HeXXX	He	Hello	3
Hello	_H e _ _ _# #_ _	Hello	5
…..			

Evaluation of unmatched case

WER

Edit distance (Levenstein)

How to align the 2 sequences?
What are the total “steps” to align = error = **distance**

Define “cost” per action:
- No action ⇒ 0
- Substitution ⇒ 1,2
- Insertion ⇒ 1
- Deletion ⇒ 1

Count ALL possible sequence of “edits” to align and take the sequence that gives min cost

Solve with Dynamic Programming ⇒
- Find a base case
- Define a recursion formula

Edit distance with DP

Solve with Dynamic Programming ⇒
- Find a base case
- Define a recursion formula

Case 1: All chars are in place: X[i]=Y[i] ⇒ Cost += 0
Case 2: X[i] ≠ Y[i]: Cost += 1 ⇒ Assuming insertion=deletion=substitution=1

We care about the min cost
We don't care "WHAT" happened (ins, del, subs)
In case costs are not equal, we calculate 3 costs and take the min

But we have many possible paths!

All possible “edits” by 2 pointers

Seq 1 = Hello
Seq 2 = Hella

Convention: we want to reach from seq2 to seq1

Keep two pointers on each seq ⇒ i, j
Scan all range of max len of both sequences

If we move i and fix j ⇒ we **insert** one char ⇒ seq 1 len > seq 2 len ⇒ cost increases by 1:
- Example: Hel, He ⇒ cost += 1

If we fix i and move j ⇒ we **delete** one char ⇒ seq 1 len < seq 2 len ⇒ cost increases by 1:
- Example: He, Hel ⇒ cost += 1

If we move both:
- Case 1: both chars are the same = cost remains the same ⇒ cost += 0
- Case 2: different chars ⇒ one char substituted ⇒ cost increases by 1
  - Hello, Hella ⇒ cost +=1

Try all possible moves of i and j
Take the minimum

Initialization

Seq 1 = Hello
Seq 2 = Hella

Convention: we want to reach for the 1st column word to the 1st row word

If we move i many steps and fix j⇒ cost will increase by these steps

```
for i in range(len(r)+1):
    d[i][0] = i
for j in range(len(h)+1):
    d[0][j] = j
```

H, X ⇒ 1, He, X ⇒ 2, Hel, X ⇒ 3, Hell, X ⇒ 4, Hello, X ⇒ 5

Full insertion

	H	e	l	l	o
X	1	2	3	4	5

Initialization

Seq 1 = Hello
Seq 2 = Hella

If we move i many steps and fix j⇒ cost will increase by these steps
If we move j many steps and fix j⇒ cost will increase by these steps

```
for i in range(len(r)+1):
    d[i][0] = i
for j in range(len(h)+1):
    d[0][j] = j
```

X, H ⇒ 1, X, He ⇒ 2, X, Hel ⇒ 3, X, Hell ⇒ 4, X,Hella ⇒ 5

Full deletion

	X
H	1
e	2
l	3
l	4
a	5

Convention: we want to reach for the 1st column word to the 1st row word

Any possible move

Seq 1 = Hello
Seq 2 = Hello

Let's move i 2 steps
j all steps

	H	e
H	0	1
e	1	0
l	2	1
l	3	2
o	4	3

1 deletion

2 deletions

3 deletions

Any possible move

Seq 1 = Hello
Seq 2 = Hello

Let’s move i all steps
j 2 steps

	H	e	l	l	o
H	0	1	2	3	4
e	1	0	1	2	3

1 insertion 2 insertions 3 insertions

Any possible move

Seq 1 = Hello
Seq 2 = Hella

Convention: we want to reach for the 1st column word to the 1st row word

Try all possible moves of i and j
Take the minimum

	H	e	l	l	o
H	0	1	2	3	4
e	1	0	1	2	3
l	2	1	0	1	2
l	3	2	2	0	1
a	4	3	3	1	1

Any possible move

Seq 1 = Hello
Seq 2 = Hello

Try all possible moves of i and j
Take the minimum

```
for i in range(1, len(r)+1):
    for j in range(1, len(h)+1):
        if r[i-1] == h[j-1]:
            d[i][j] = d[i-1][j-1]
        else:
            substitute = d[i-1][j-1] + 1
            insert = d[i][j-1] + 1
            delete = d[i-1][j] + 1
            d[i][j] = min(substitute, insert, delete)
```

Evaluation of unmatched case

Dan Jurafsky

Defining Min Edit Distance (Levenshtein)

- Initialization

  $D(i,0) = i$

  $D(0,j) = j$

- Recurrence Relation:

  For each $i = 1 \ldots M$

  For each $j = 1 \ldots N$

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

- Termination:

  $D(N,M)$ is distance

In this pseudo-code, cost of subs = 2

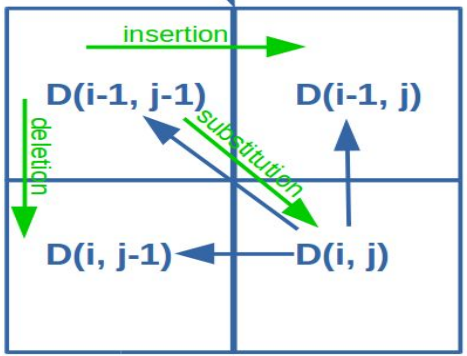
	""	i	a	w	n
""	0	1	2	3	4
f	1	1	2	3	4
l	2	1	2	3	4
a	3	2	1	2	3
w	4	3	2	1	2

Empty Strings

Costs for insertions

Costs for deletions

Total Distance

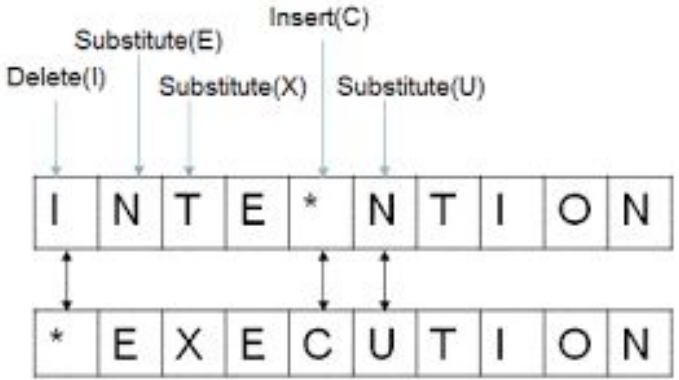


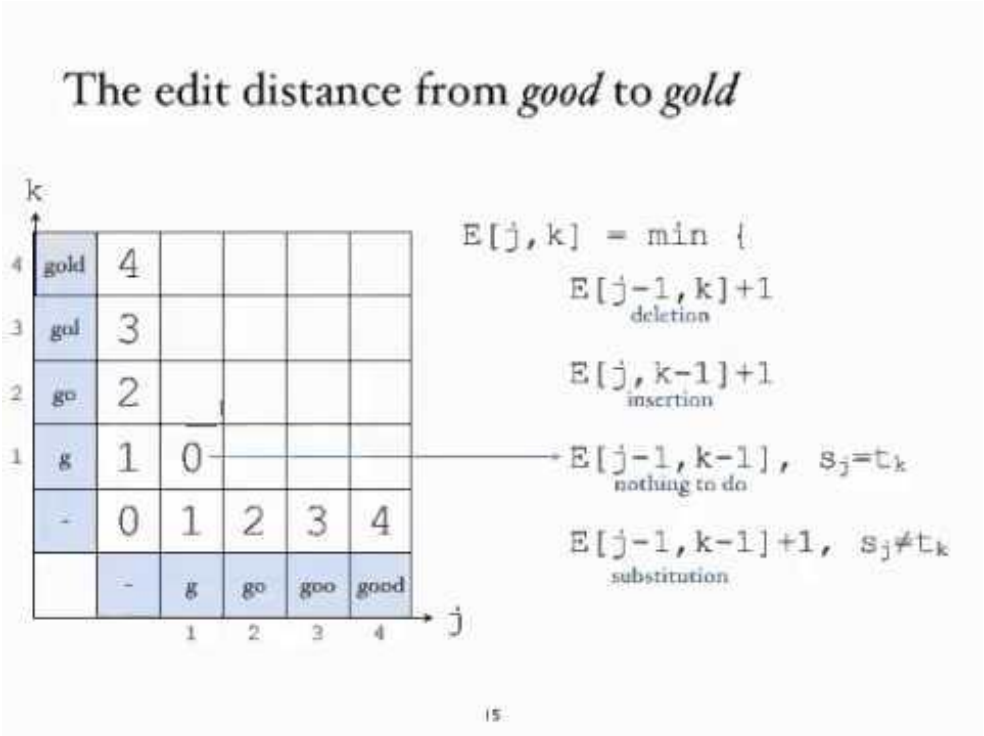
https://i.stack.imgur.com/GZ5lq.jpg

Evaluation of unmatched case

WER

Edit distance (Levenstein)

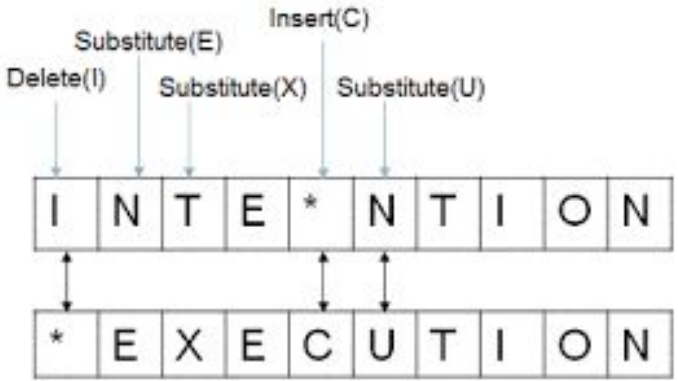




Evaluation of unmatched case

WER

Edit distance (Levenstein)



$$E[j,k] = \min \begin{cases} E[j-1,k]+1 & \text{deletion} \\ E[j,k-1]+1 & \text{insertion} \\ E[j-1,k-1], \ s_j = t_k & \text{nothing to do} \\ E[j-1,k-1]+1, \ s_j \neq t_k & \text{substitution} \end{cases}$$

-	0	E	X	E	C	U	T	I	O	N
0	0	1	2	3	4	5	6	7	8	9
I	1	1	2	3	4	5	6	6	7	8
N	2	2	2	3	4	5	6	7	7	7
T	3	3	3	3	4	5	5	6	7	8
E	4	3	4	3	4	5	6	6	7	8
N	5	4	4	4	4	5	6	7	7	7
T	6	5	5	5	5	5	5	6	7	8
I	7	6	6	6	6	6	6	5	6	7
O	8	7	7	7	7	7	7	6	5	6
N	9	8	8	8	8	8	8	7	6	5

https://i.pinimg.com/originals/77/65/82/776582769376fa42eeb3cdb898e449d1.gif

Evaluation of unmatched case

WER

Edit distance (Levenstein)

$$
\begin{aligned}
E[j,k] = \min \{ & \\
& E[j-1,k]+1 \quad \text{deletion} \\
& E[j,k-1]+1 \quad \text{insertion} \\
\longrightarrow\ & E[j-1,k-1], \quad s_j = t_k \quad \text{nothing to do} \\
& E[j-1,k-1]+1, \quad s_j \neq t_k \quad \text{substitution}
\end{aligned}
$$

		m	o	n	k	e	y
	0	1	2	3	4	5	6
m	1	0	1	2	3	4	5
o	2	1	0	1	2	3	4
n	3	2	1	0	1	2	3
e	4	3	2	1	1	1	2
y	5	4	3	2	2	2	1

https://i0.wp.com/python.gotrained.com/wp-content/uploads/2018/07/edit-distant-1.png?resize=221%2C221&ssl=1

Agenda

- What are seq2seq?
- Why seq2seq?
- Matched vs Unmatched seq2seq
- Word vs Char level
- Training data: Teacher forcing
- Convs2s
- **Basic seq2seq with LSTM**
- Masking
- Seq2seq + Attention
- Decoding
- Transformers

Seq2Seq for NMT

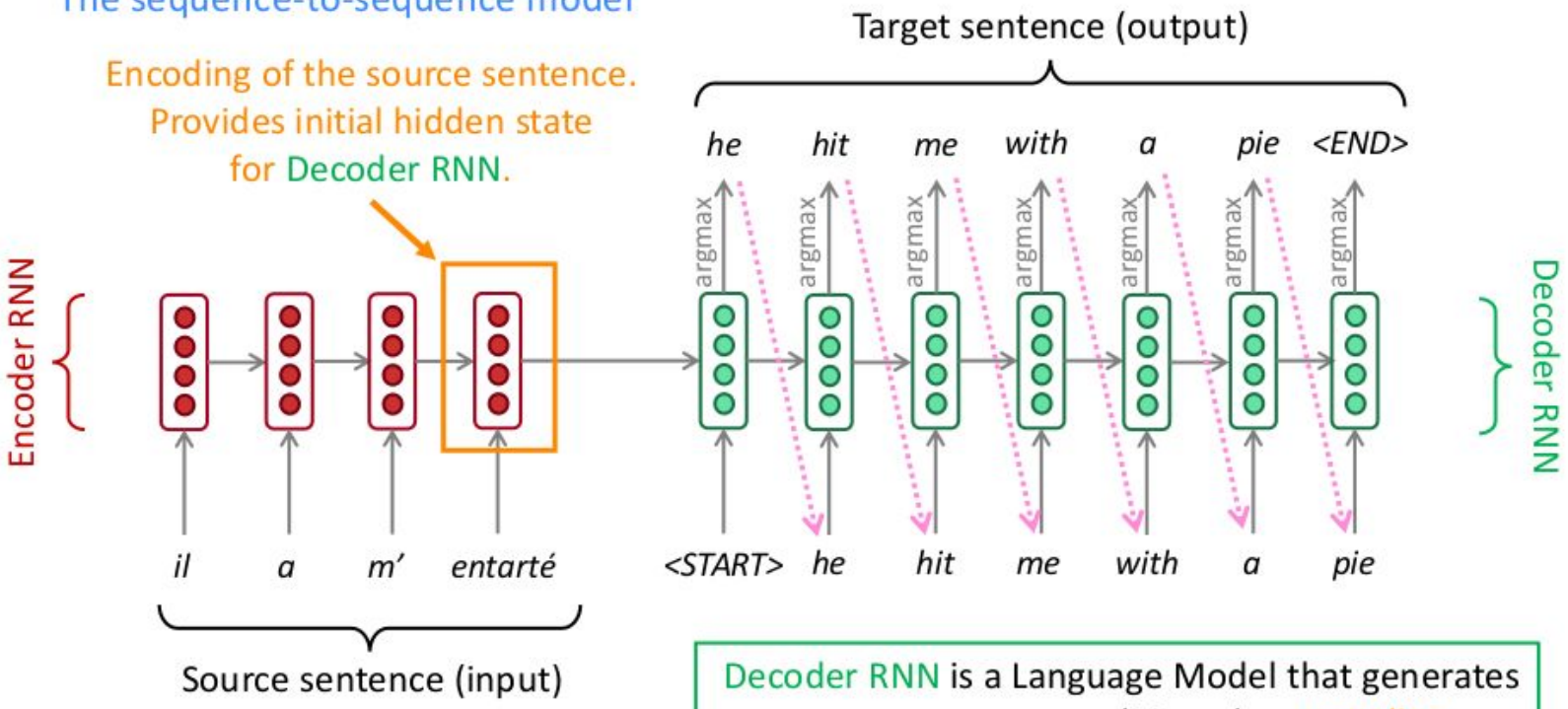
What is Neural Machine Translation?

- Neural Machine Translation (NMT) is a way to do Machine Translation with a *single neural network*

- The neural network architecture is called sequence-to-sequence (aka seq2seq) and it involves *two* RNNs.

Neural Machine Translation (NMT)

The sequence-to-sequence model



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

Encoder RNN produces an encoding of the source sentence.

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

Neural Machine Translation (NMT)

- The sequence-to-sequence model is an example of a **Conditional Language Model**.
  - **Language Model** because the decoder is predicting the next word of the target sentence *y*
  - **Conditional** because its predictions are *also* conditioned on the source sentence *x*

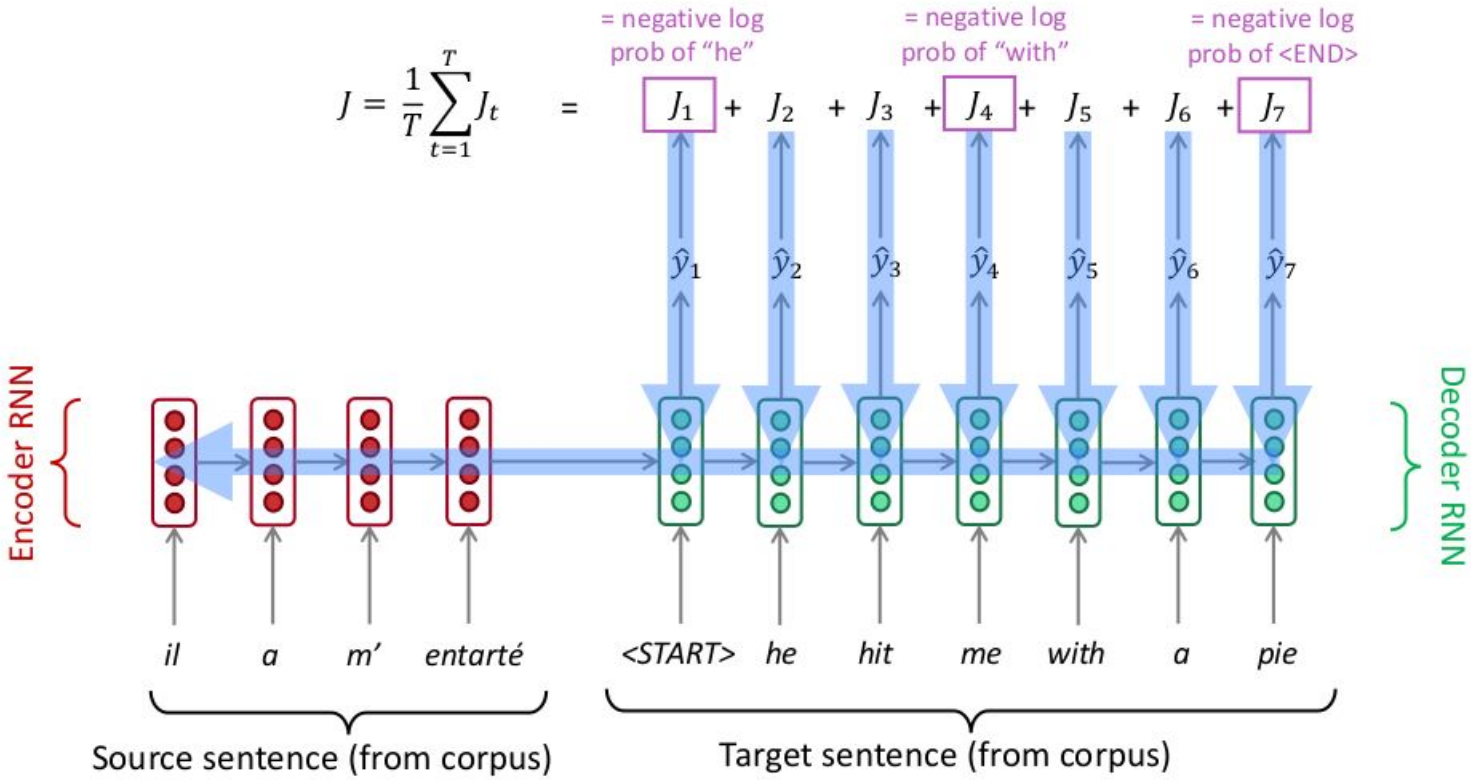
- NMT directly calculates $P(y|x)$:

$$P(y|x) = P(y_1|x)\, P(y_2|y_1, x)\, P(y_3|y_1, y_2, x) \ldots \underbrace{P(y_T|y_1, \ldots, y_{T-1}, x)}$$

Probability of next target word, given target words so far and source sentence *x*

- **Question**: How to train a NMT system?
- **Answer**: Get a big parallel corpus…

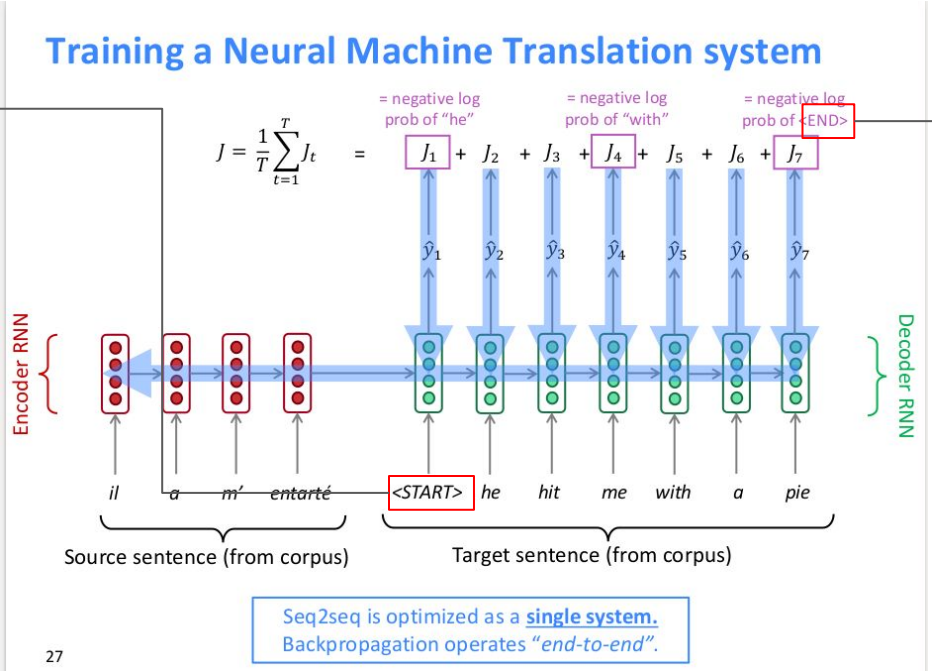
Training a Neural Machine Translation system



Seq2seq is optimized as a **single system.**
Backpropagation operates "*end-to-end*".

Input/output data

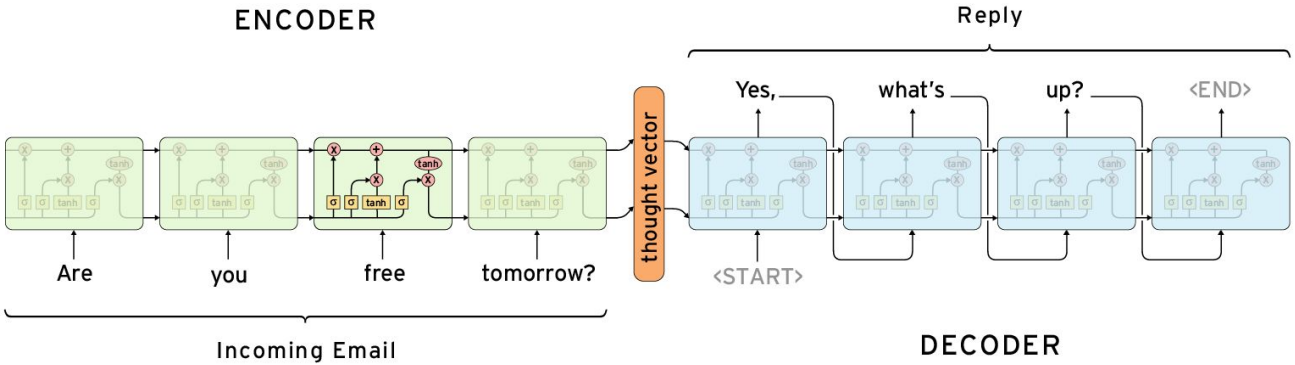
Notice the addition of <START> token to the beginning of target→ To kick start the decoder



Notice the addition of <END> token to the end of target→ To mark end of generation of Decoder. More on that in Decoding

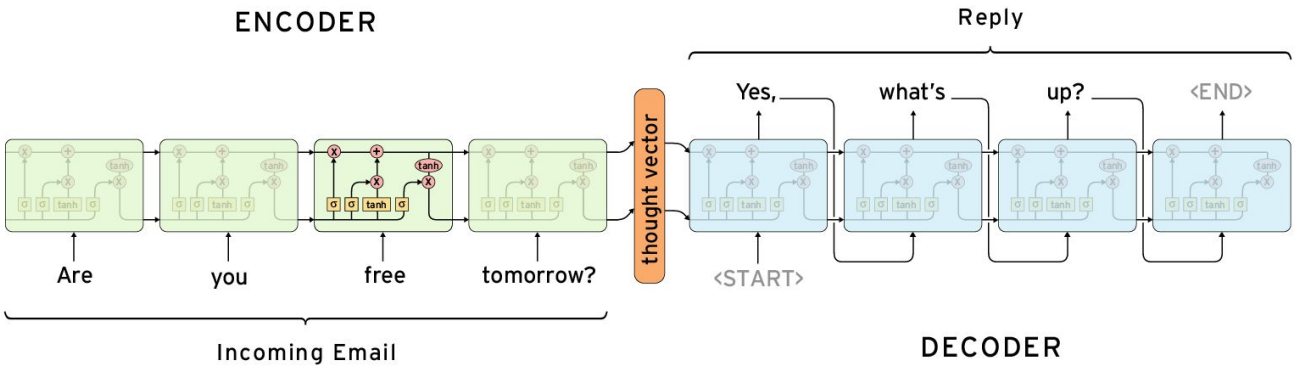
Training

- RNN has 2 inputs: X and h(t-1). LSTM: h=[c;h]
- Encoder: the whole input seq is first fed→ X=true input
- Decoder: The encoder state = decoder init state. X = ?
  - X[0] = <START>
  - Next X[t] = y[t-1] → GT



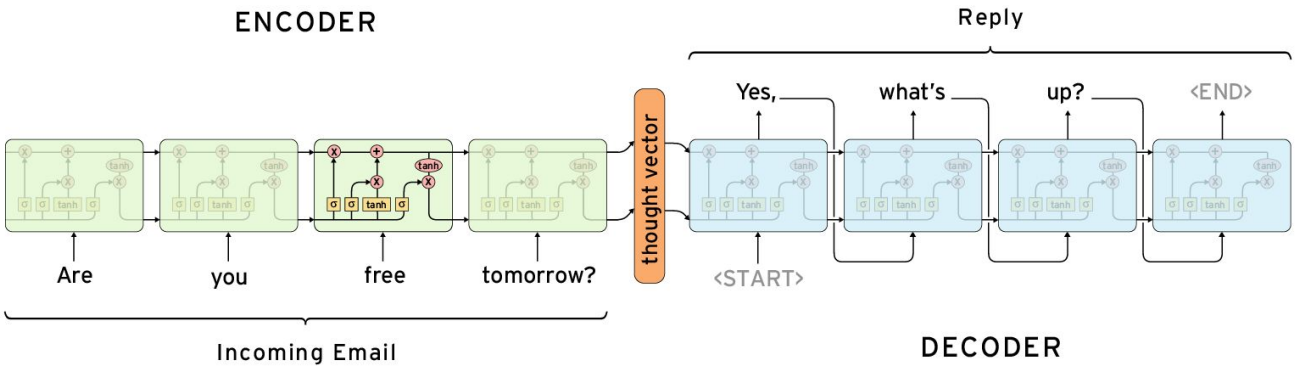
Inference

- RNN has 2 inputs: X and h(t-1). LSTM: h=[c;h]
- Encoder: the whole input seq is first fed→ X=true input
- Decoder: The encoder state = decoder init state. X = ?
  - X[0] = <START>
  - Next X[t] = y[t-1] → Prediction
  - Stop on generating <END>



Teacher forcing → Training

- RNN has 2 inputs: X and h(t-1). LSTM: h=[c;h]
- Encoder: the whole input seq is first fed→ X=true input
- Decoder: The encoder state = decoder init state. X = ?
  - X[0] = <START>
  - Next X[t] = y[t-1] → p → Prediction, 1-p → GT → *Requires dynamic graph (Pytorch or Eager TF)*
  - Stop on generating <END>



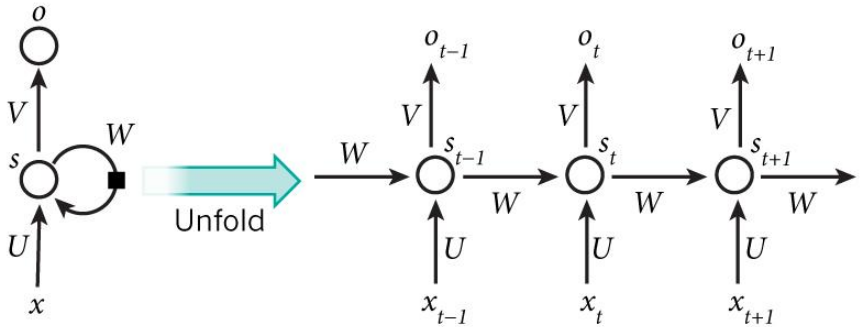
Mask layer

Although RNN support variable length input, static graphs (TF, keras) needs to **unroll** the RNN.

Even in dynamic case, or unrolled RNN, at least you need to have matched length batches → (n_samples, n_time_steps, n_features).

You can build the graph with unknown length

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, None, 128)	2560000
lstm_1 (LSTM)	(None, 128)	131584
dense_1 (Dense)	(None, 1)	129



Mask layer

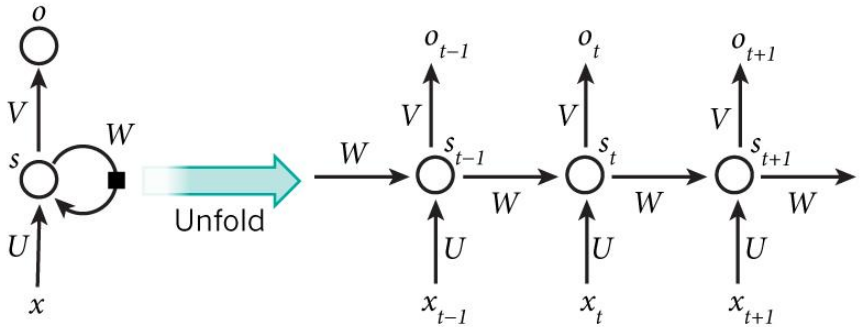
Although RNN support variable length input, static graphs (TF, keras) needs to **unroll** the RNN.

Even in dynamic case, or unrolled RNN, at least you need to have matched length batches → (n_samples, n_time_steps, n_features).

You can build the graph with unknown length ⇒ But must define design matrix at .fit:

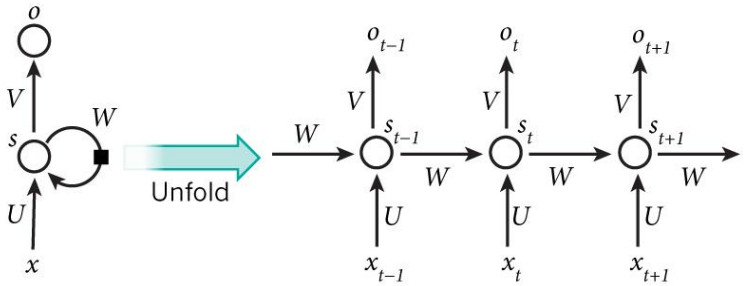
As a numpy matrix, n_time_steps needs

This requires padding!



Mask layer

Although RNN support variable length input, static graphs (TF, keras) needs to **unroll** the RNN.



Even in dynamic case, or unrolled RNN, at least you need to have matched length batches → (n_samples, n_time_steps, n_features). As a numpy matrix, n_time_steps needs to be the same at least per batch!

This requires padding!

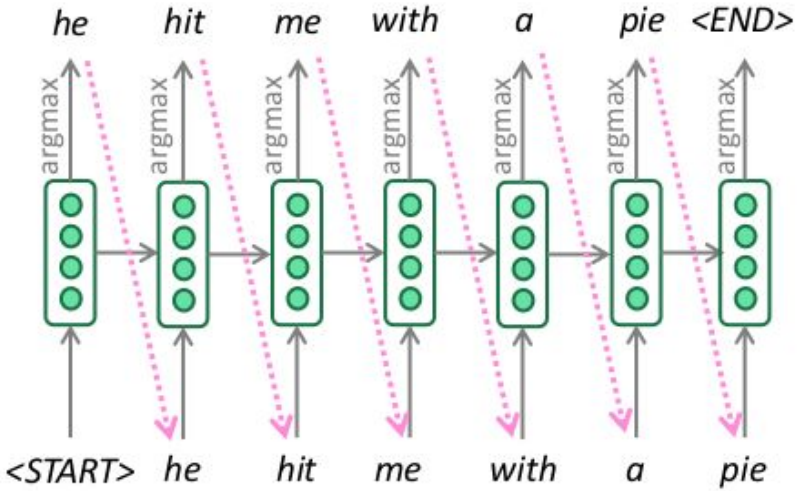
But we can mask the padded tokens from the loss!

You might see higher accuracy with padded tokens. But this is fake, since it's very easy for the network to predict!

```
print('Build model...')
model = Sequential()
model.add(Embedding(max_features, 128, mask_zero=True))
model.add(LSTM(128),return_sequences=True)
model.add(LSTM(128))
model.add(Dense(1, activation='sigmoid'))
```

Greedy decoding

- We saw how to generate (or "decode") the target sentence by taking argmax on each step of the decoder



- This is greedy decoding (take most probable word on each step)
- **Problems with this method?**

Problems with greedy decoding

- Greedy decoding has no way to undo decisions!
  - Input: *il a m'entarté* *(he hit me with a pie)*
  - → *he* ____
  - → *he hit* ____
  - → *he hit a* ____ (whoops! no going back now…)

- How to fix this?

Exhaustive search decoding

- Ideally we want to find a (length *T*) translation *y* that maximizes

$$P(y|x) = P(y_1|x)\,P(y_2|y_1,x)\,P(y_3|y_1,y_2,x)\ldots,P(y_T|y_1,\ldots,y_{T-1},x)$$
$$= \prod_{t=1}^{T} P(y_t|y_1,\ldots,y_{t-1},x)$$

- We could try computing all possible sequences *y*
  - This means that on each step *t* of the decoder, we're tracking V^t possible partial translations, where *V* is vocab size
  - This $O(V^T)$ complexity is far too expensive!

Viterbi

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 $y_1, \ldots, y_t$ 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

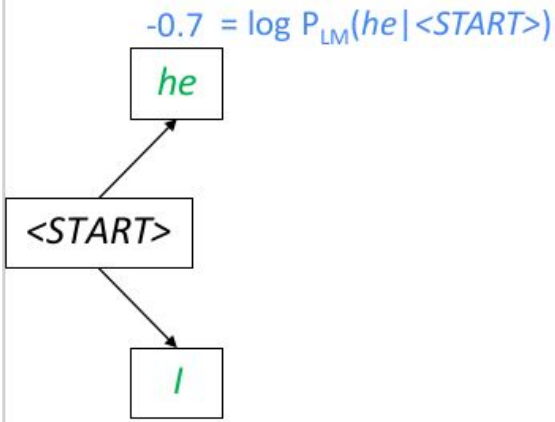
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$

<START>

Calculate prob dist of next word

Beam search decoding: example

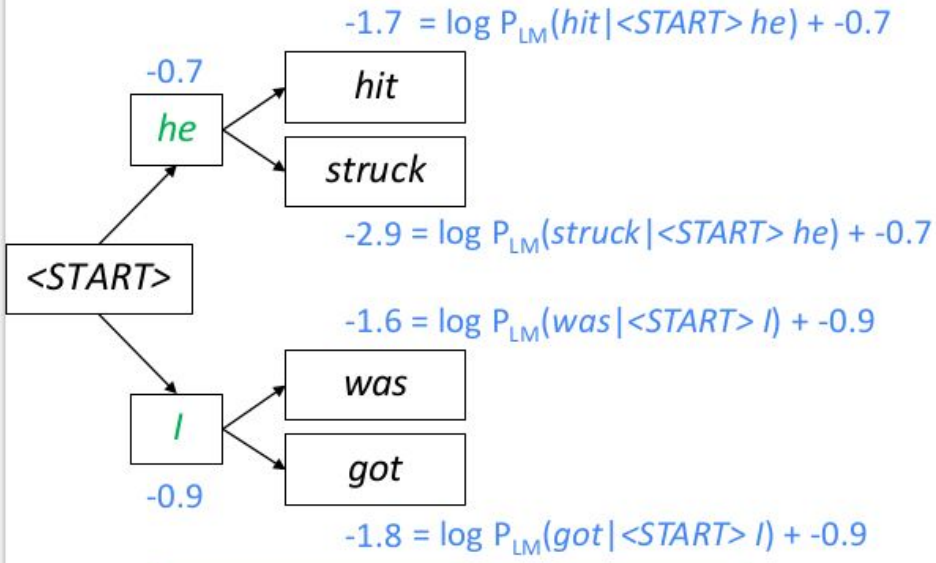
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Take top *k* words and compute scores

Beam search decoding: example

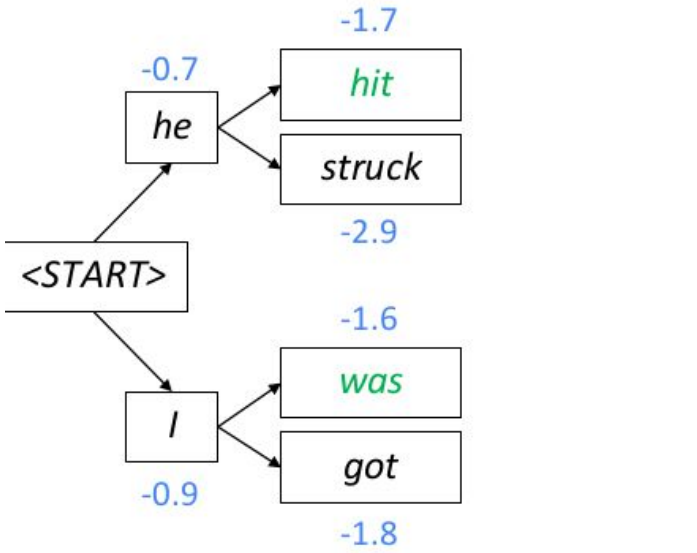
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



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

Beam search decoding: example

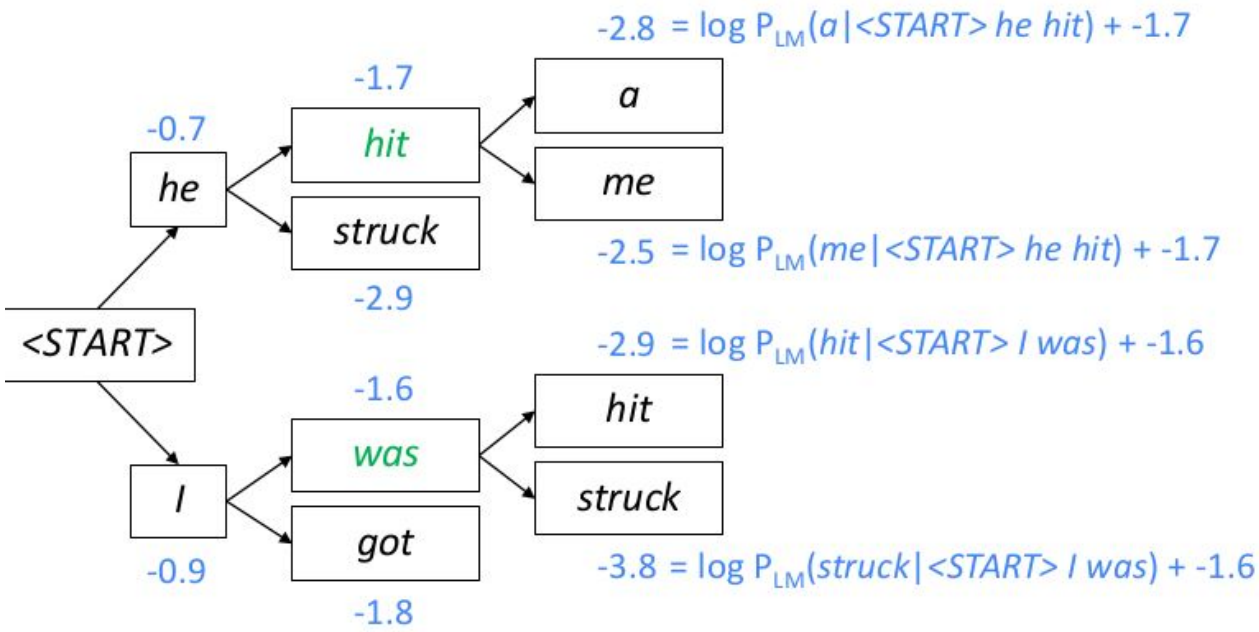
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Of these k^2 hypotheses, just keep k with highest scores

Beam search decoding: example

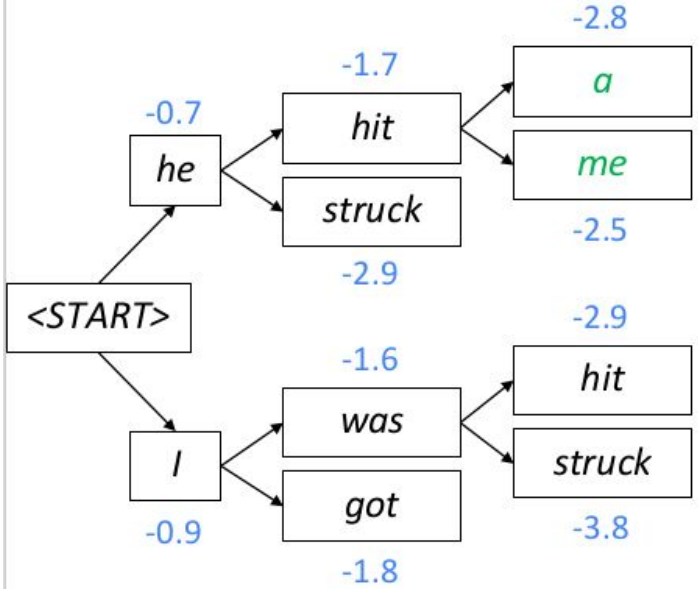
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



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

Beam search decoding: example

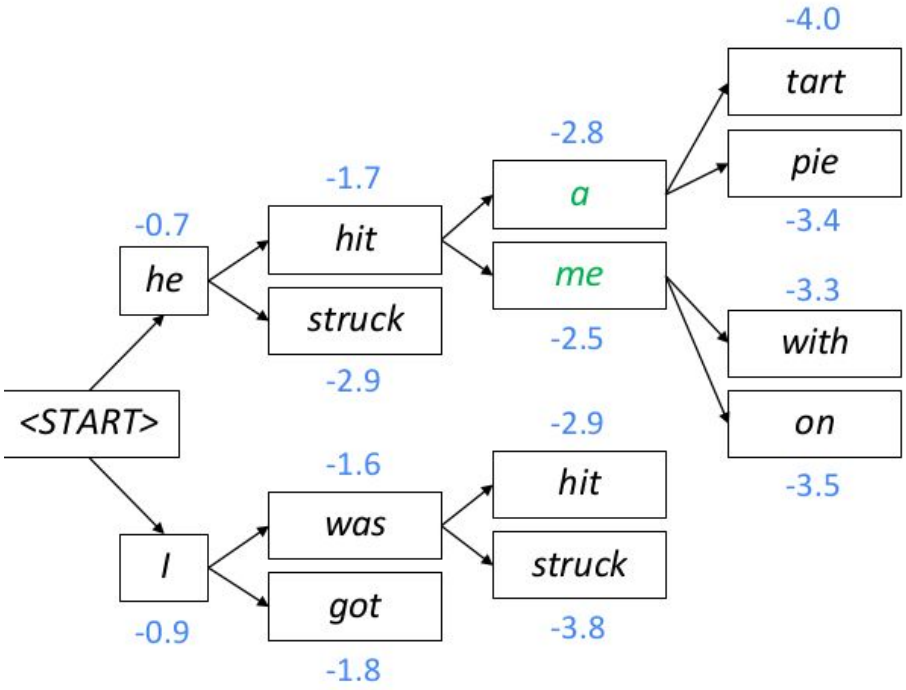
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Of these k^2 hypotheses, just keep k with highest scores

Beam search decoding: example

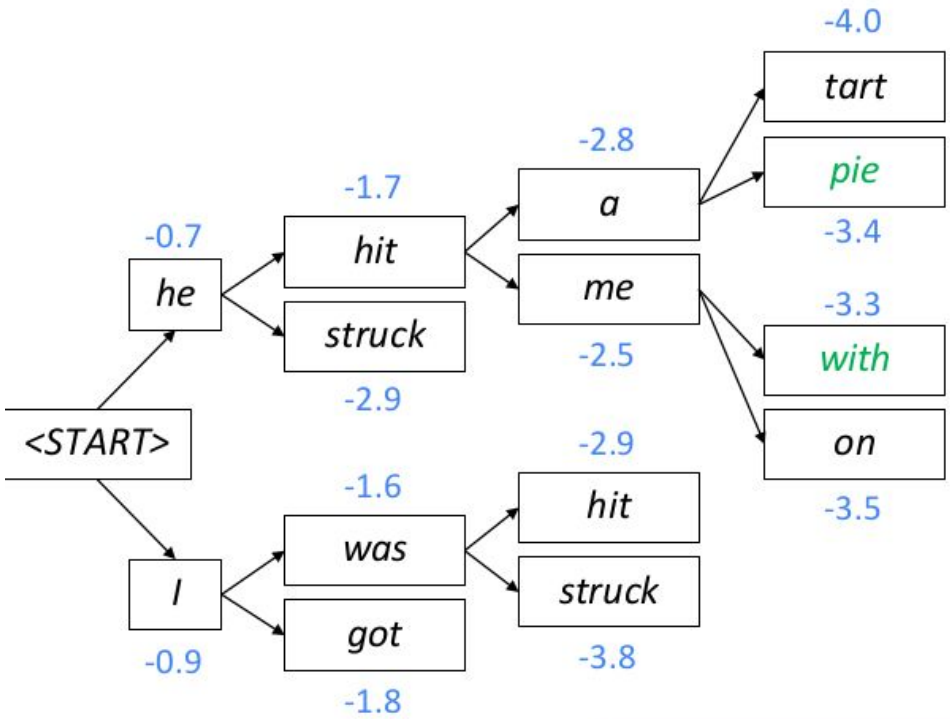
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



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

Beam search decoding: example

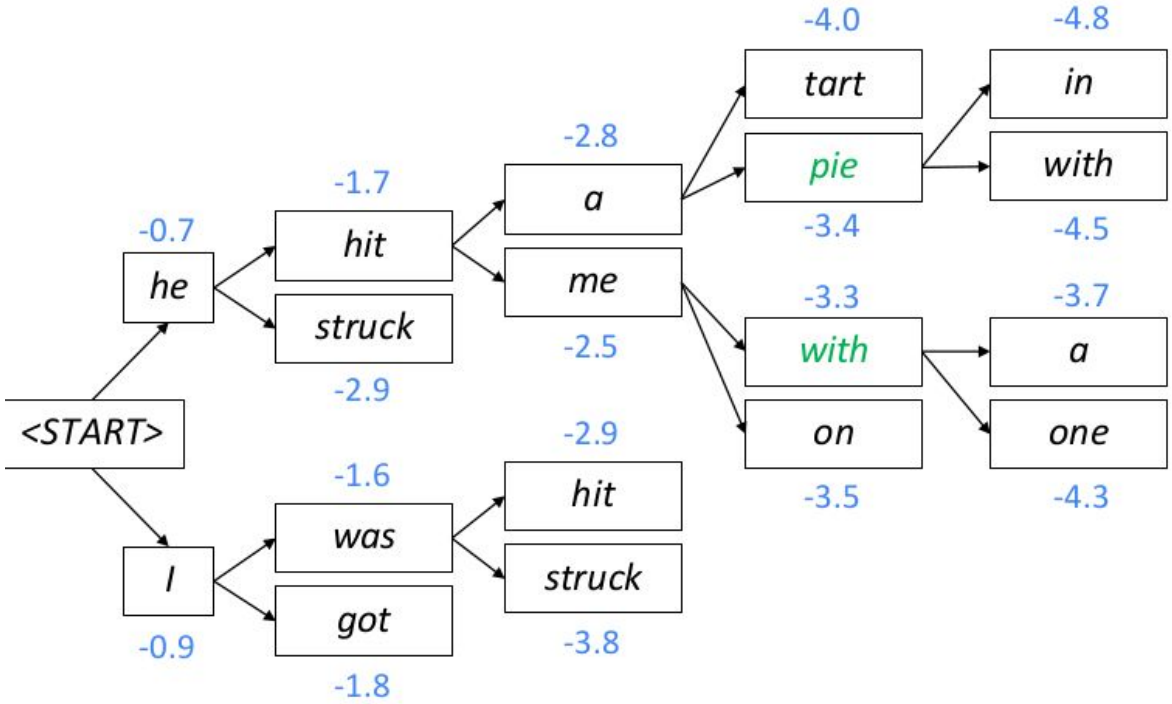
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Of these k^2 hypotheses, just keep k with highest scores

Beam search decoding: example

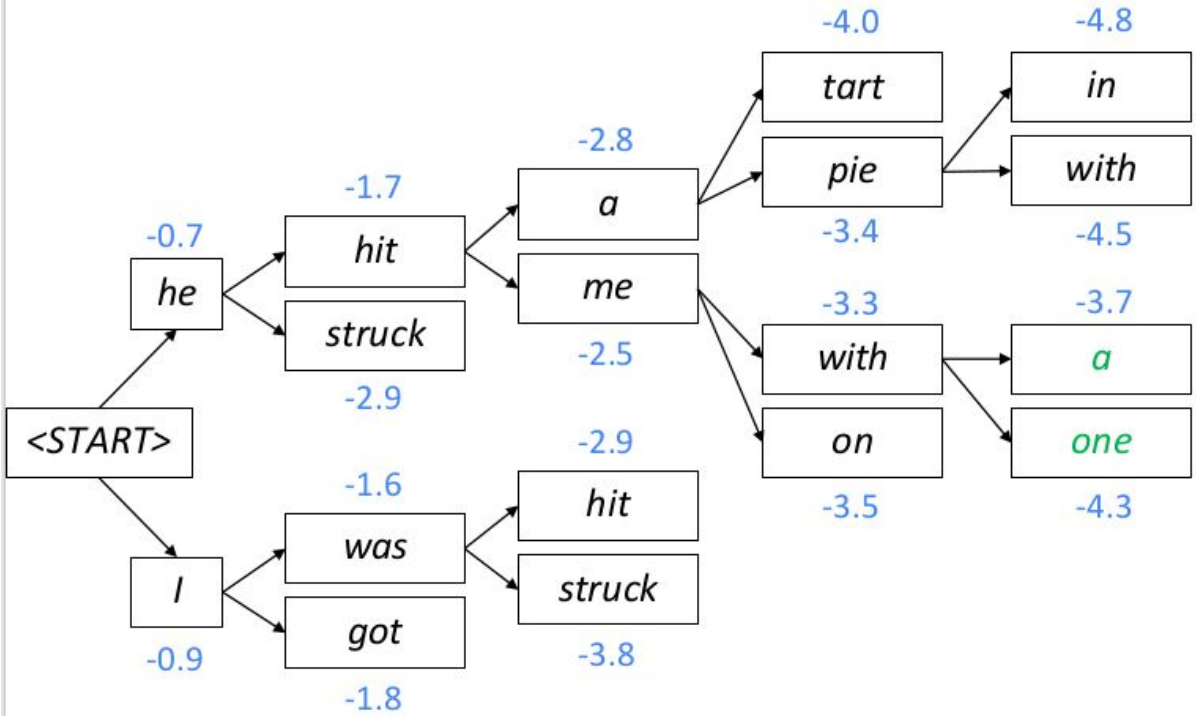
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



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

Beam search decoding: example

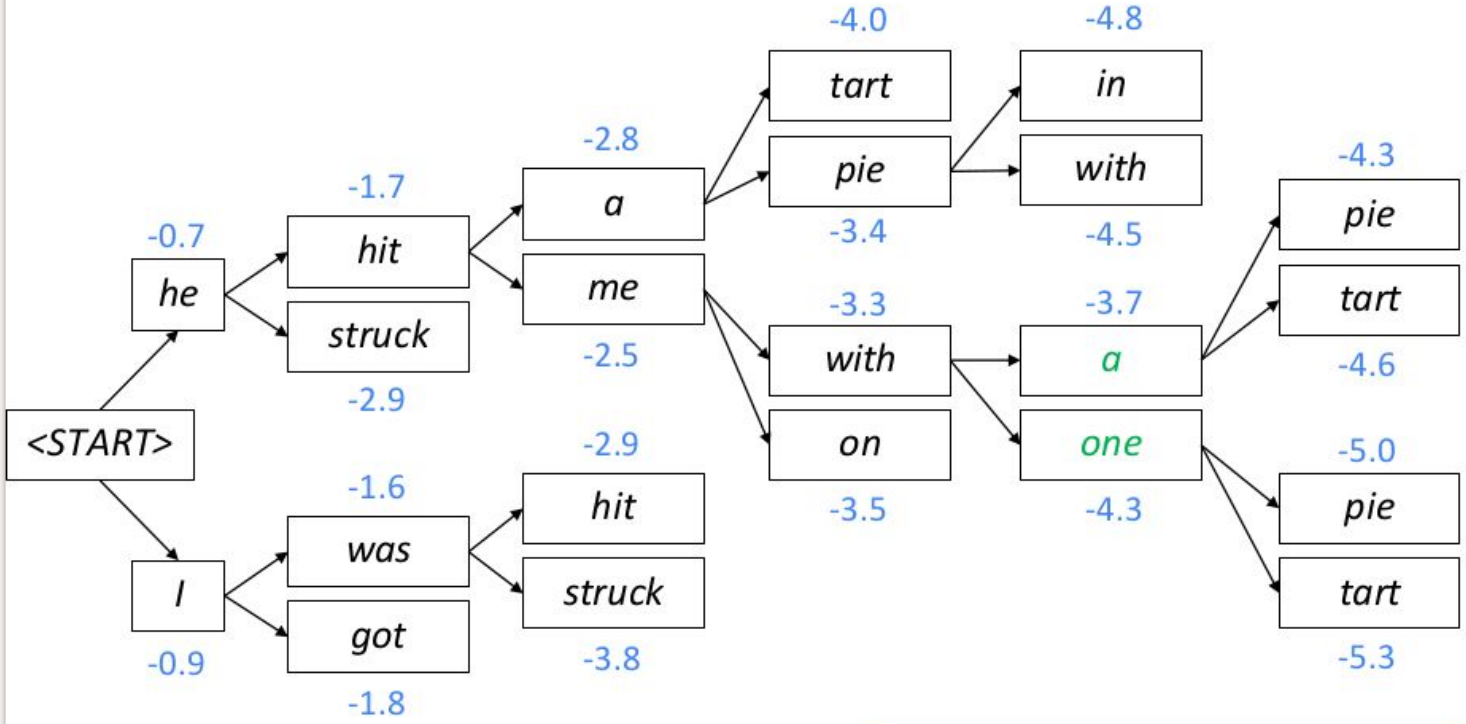
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



Of these k^2 hypotheses, just keep k with highest scores

Beam search decoding: example

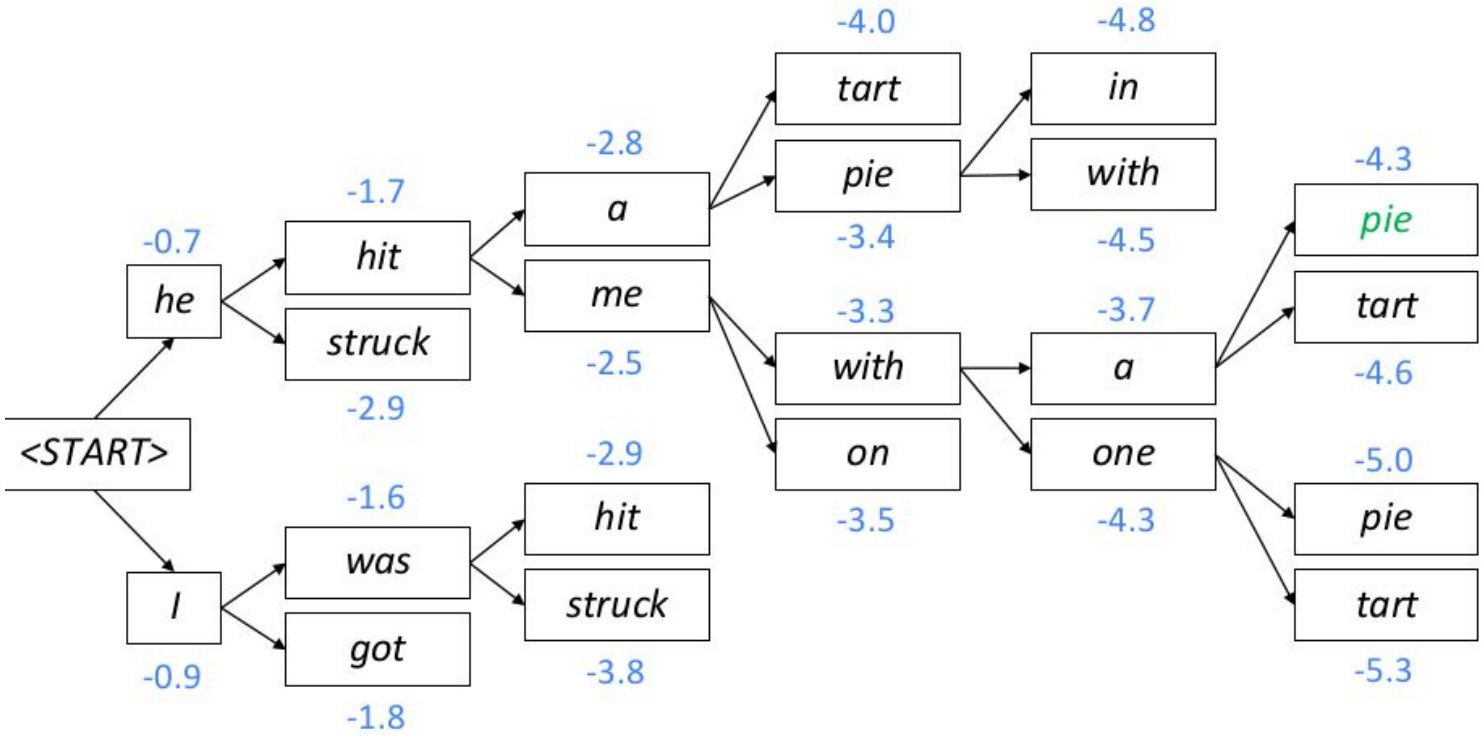
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



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

Beam search decoding: example

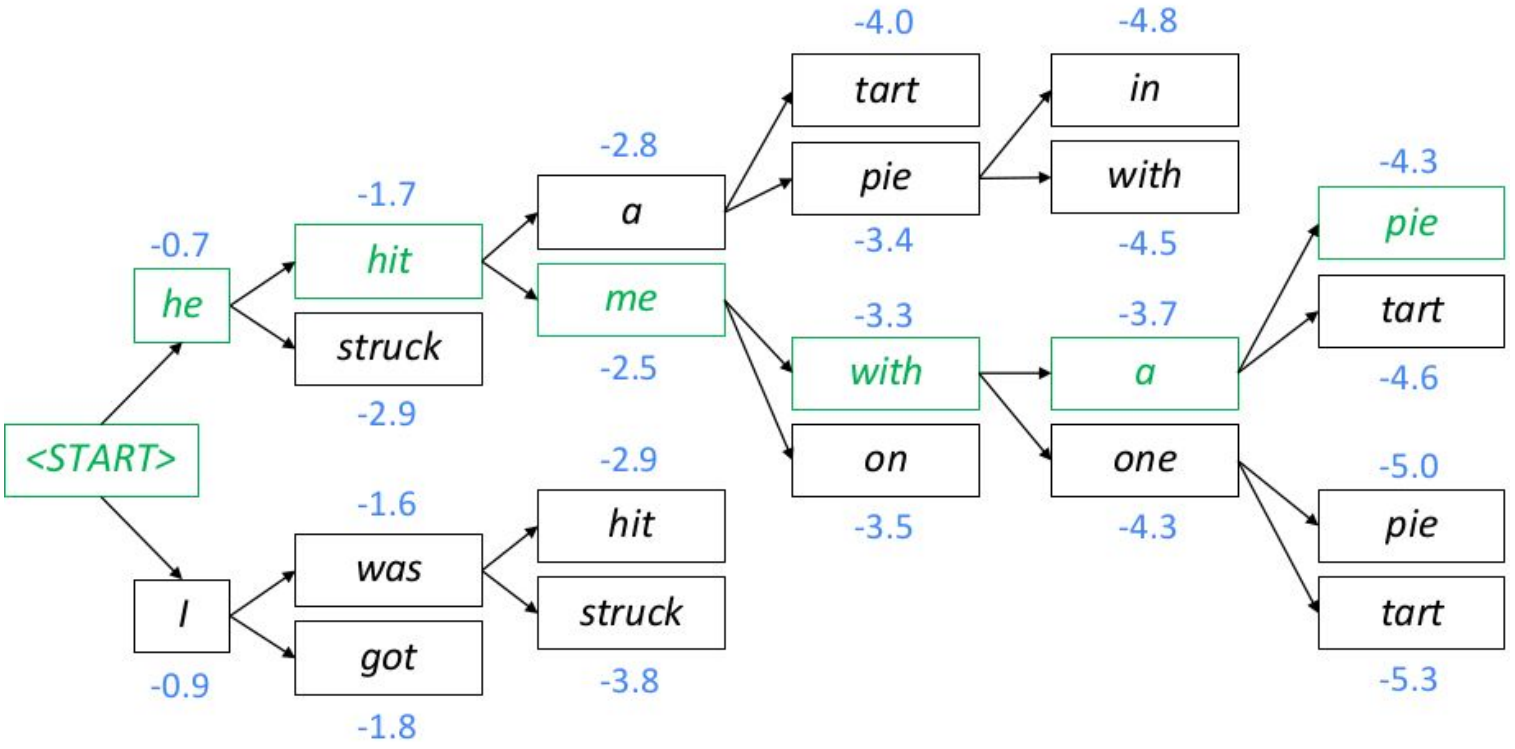
Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$



This is the top-scoring hypothesis!

Beam search decoding: example

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



Backtrack to obtain the full hypothesis

Beam search decoding: stopping criterion

- In greedy decoding, usually we decode until the model produces a <END> token
  - For example: *<START> he hit me with a pie <END>*

- In beam search decoding, different hypotheses may produce <END> tokens on different timesteps
  - When a hypothesis produces <END>, that hypothesis is complete.
  - Place it aside and continue exploring other hypotheses via beam search.

- Usually we continue beam search until:
  - We reach timestep T (where T is some pre-defined cutoff), or
  - We have at least n completed hypotheses (where n is pre-defined cutoff)

Beam search decoding: finishing up

- We have our list of completed hypotheses.
- How to select top one with highest score?

- Each hypothesis $y_1, \ldots, y_t$ on our list has a score

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

- Problem with this: longer hypotheses have lower scores

- Fix: Normalize by length. Use this to select top one instead:

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

Let’s code

- Functional Keras API: https://colab.research.google.com/drive/1dhlc3Nt_LvZcxY5tUd-XLU1fvGt4pPuh?usp=sharing
- See how the random teacher forcing is not possible in static graphs. See the difference with sub classing API and Eager execution: https://colab.research.google.com/drive/1Pne_EbX6rgKfZBIwvJ6OOqVv50UceRAO?usp=sharing
-

NMT: the biggest success story of NLP Deep Learning

Neural Machine Translation went from a fringe research activity in **2014** to the leading standard method in **2016**

- **2014**: First seq2seq paper published
- **2016**: Google Translate switches from SMT to NMT
- This is amazing!
  - **SMT** systems, built by hundreds of engineers over many years, outperformed by NMT systems trained by a handful of engineers in a few months

How do we evaluate Machine Translation?

BLEU (**Bil**ingual **E**valuation **U**nderstudy)

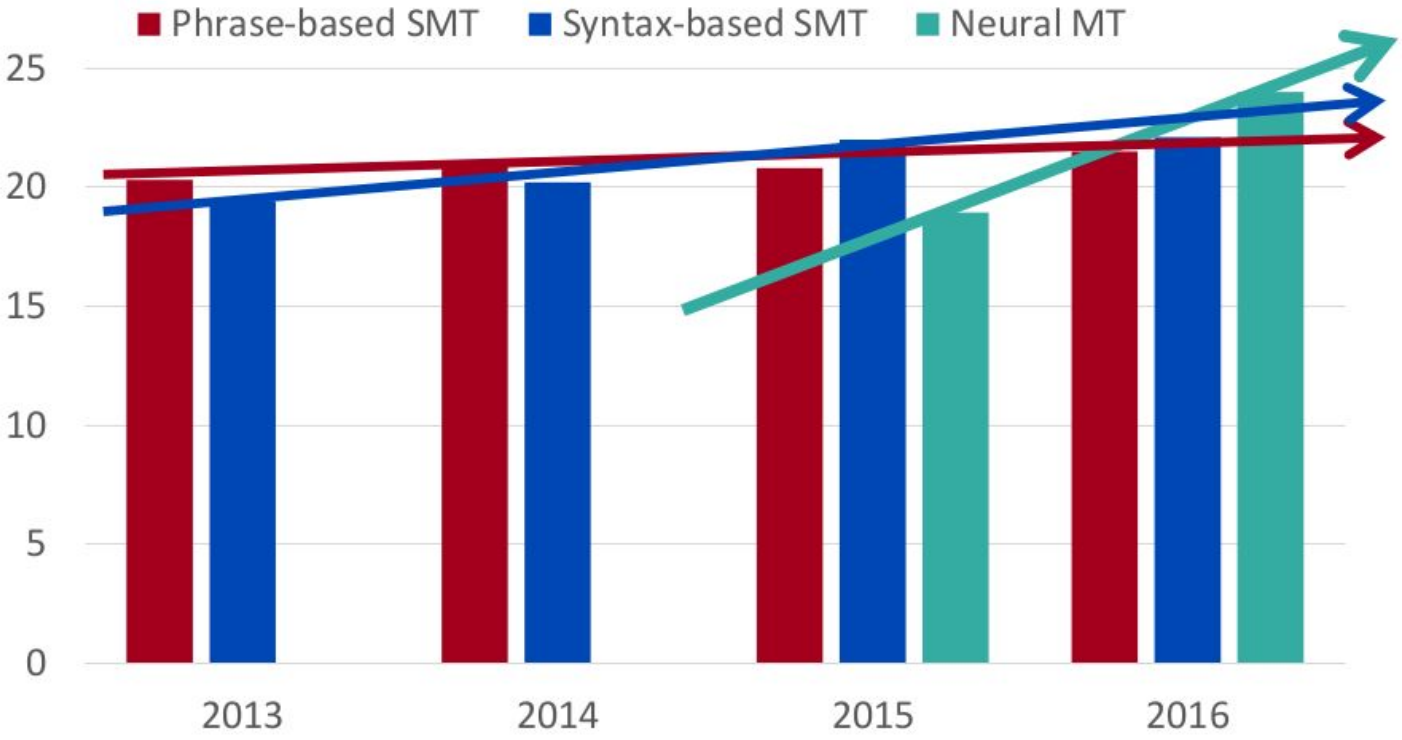
You'll see BLEU in detail in Assignment 4!

- BLEU compares the machine-written translation to one or several human-written translation(s), and computes a similarity score based on:
  - *n*-gram precision (usually for 1, 2, 3 and 4-grams)
  - Plus a penalty for too-short system translations

- BLEU is useful but imperfect
  - There are many valid ways to translate a sentence
  - So a good translation can get a poor BLEU score because it has low *n*-gram overlap with the human translation ☹

MT progress over time

[Edinburgh En-De WMT newstest2013 Cased BLEU; NMT 2015 from U. Montréal]



Source: http://www.meta-net.eu/events/meta-forum-2016/slides/09_sennrich.pdf

End-end NMT

Advantages of NMT

Compared to SMT, NMT has many advantages:

- Better performance
  - More fluent
  - Better use of context
  - Better use of phrase similarities
- A single neural network to be optimized end-to-end
  - No subcomponents to be individually optimized
- Requires much less human engineering effort
  - No feature engineering
  - Same method for all language pairs

Disadvantages of NMT?

Compared to SMT:

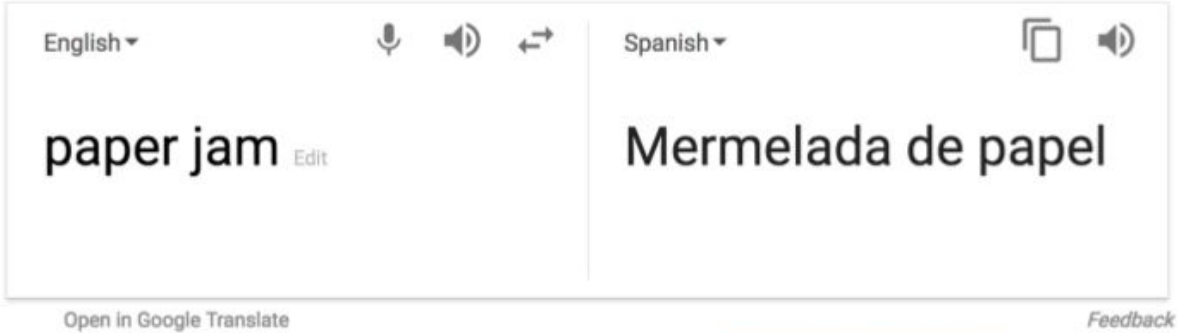
- NMT is less interpretable
  - Hard to debug
- NMT is difficult to control
  - For example, can't easily specify rules or guidelines for translation
  - Safety concerns!

So is Machine Translation solved?

- **Nope!**
- Many difficulties remain:
  - Out-of-vocabulary words
  - Domain mismatch between train and test data
  - Maintaining context over longer text
  - Low-resource language pairs

So is Machine Translation solved?

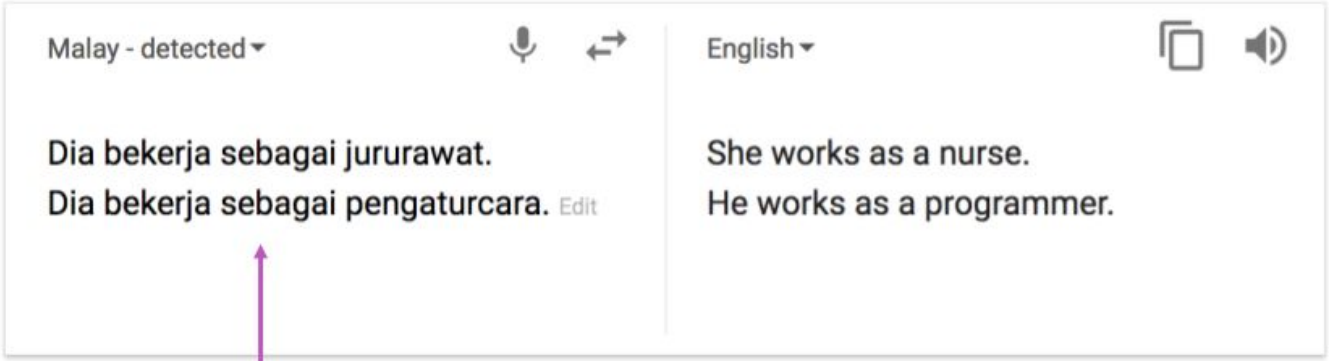
- **Nope!**
- Using common sense is still hard



?

So is Machine Translation solved?

- **Nope!**
- NMT picks up biases in training data

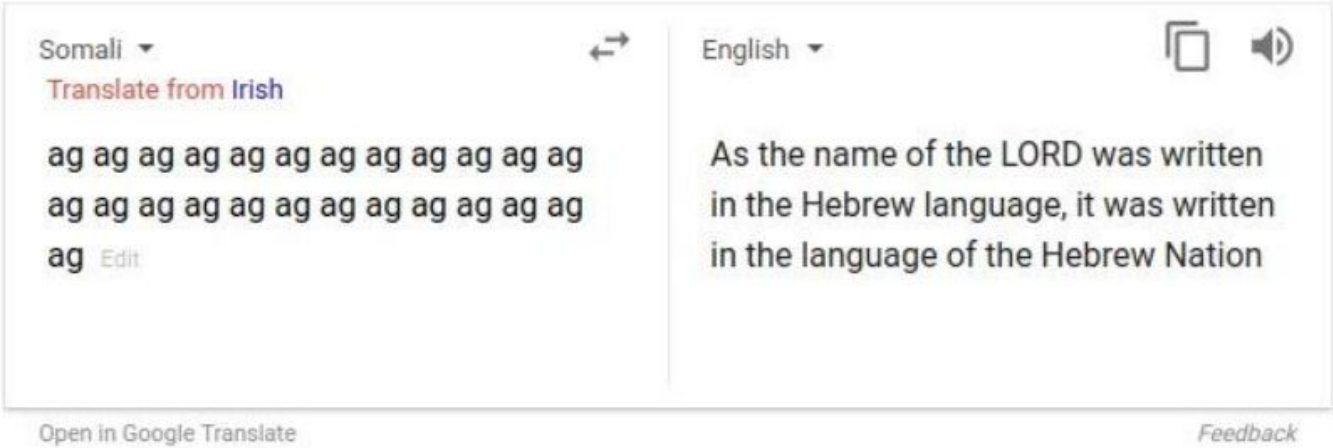


Didn't specify gender

Source: https://hackernoon.com/bias-sexist-or-this-is-the-way-it-should-be-ce1f7c8c683c

So is Machine Translation solved?

- Nope!
- Uninterpretable systems do strange things



Picture source: https://www.vice.com/en_uk/article/j5npeg/why-is-google-translate-spitting-out-sinister-religious-prophecies

Explanation: https://www.skynettoday.com/briefs/google-nmt-prophecies

NMT research continues

NMT is the **flagship task** for NLP Deep Learning

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

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

ATTENTION

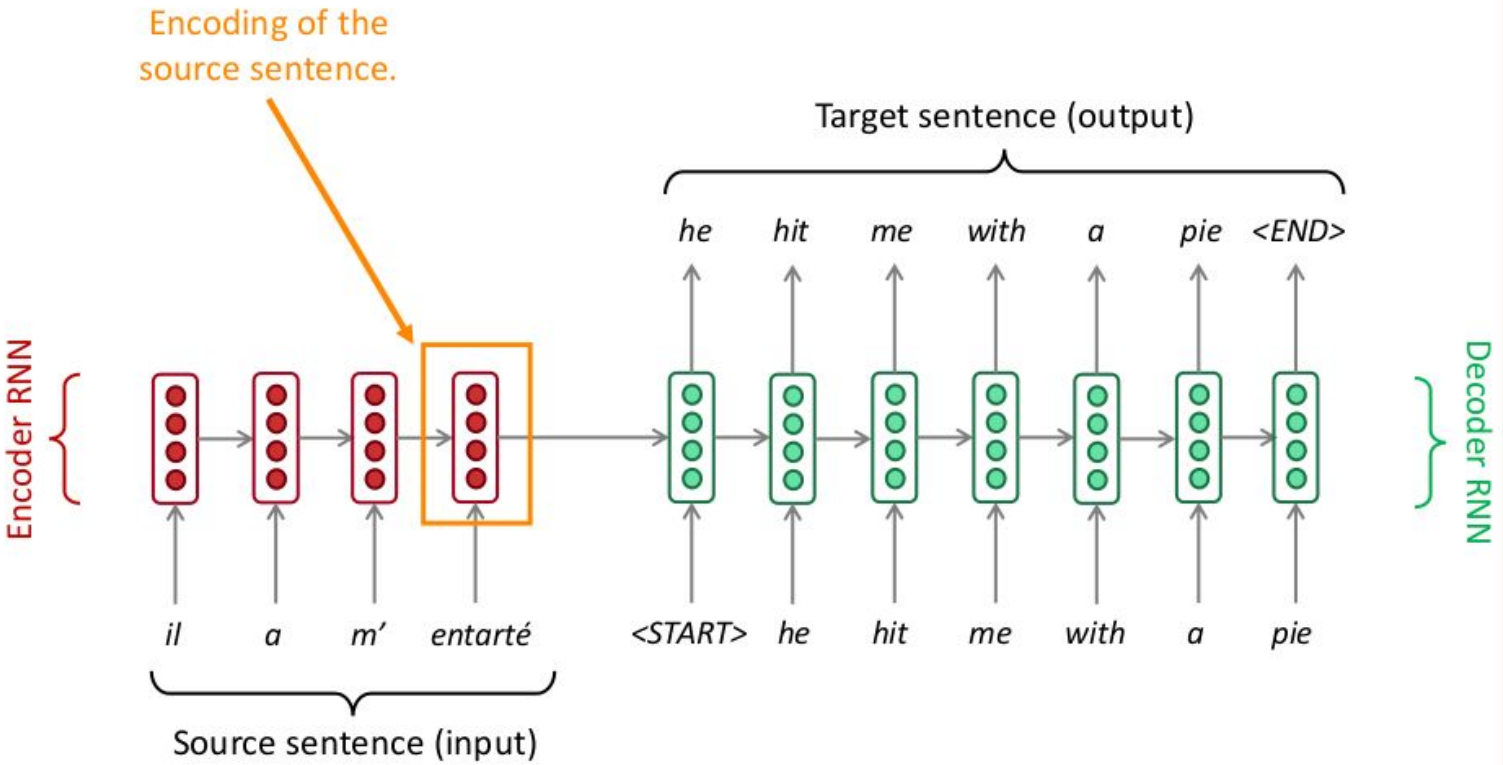
Seq2seq + Attention

- The vanilla seq2seq suffers a severe drawback, all the decoded outputs are initiated from the *final* internal state of the encoder.
- The objective is to mimic the reading process in humans, where the whole sequence is comprehended first, then translation (decoding) happens based on that.
- However, it is clear that, even we as humans will not generate the whole translations without looking back at the *concerned* parts of the input text.
- In other words, every translated word is usually generated by *attending* to certain words in the input, in sequence.

So it is better to *condition* the decoded words on certain words of the input.

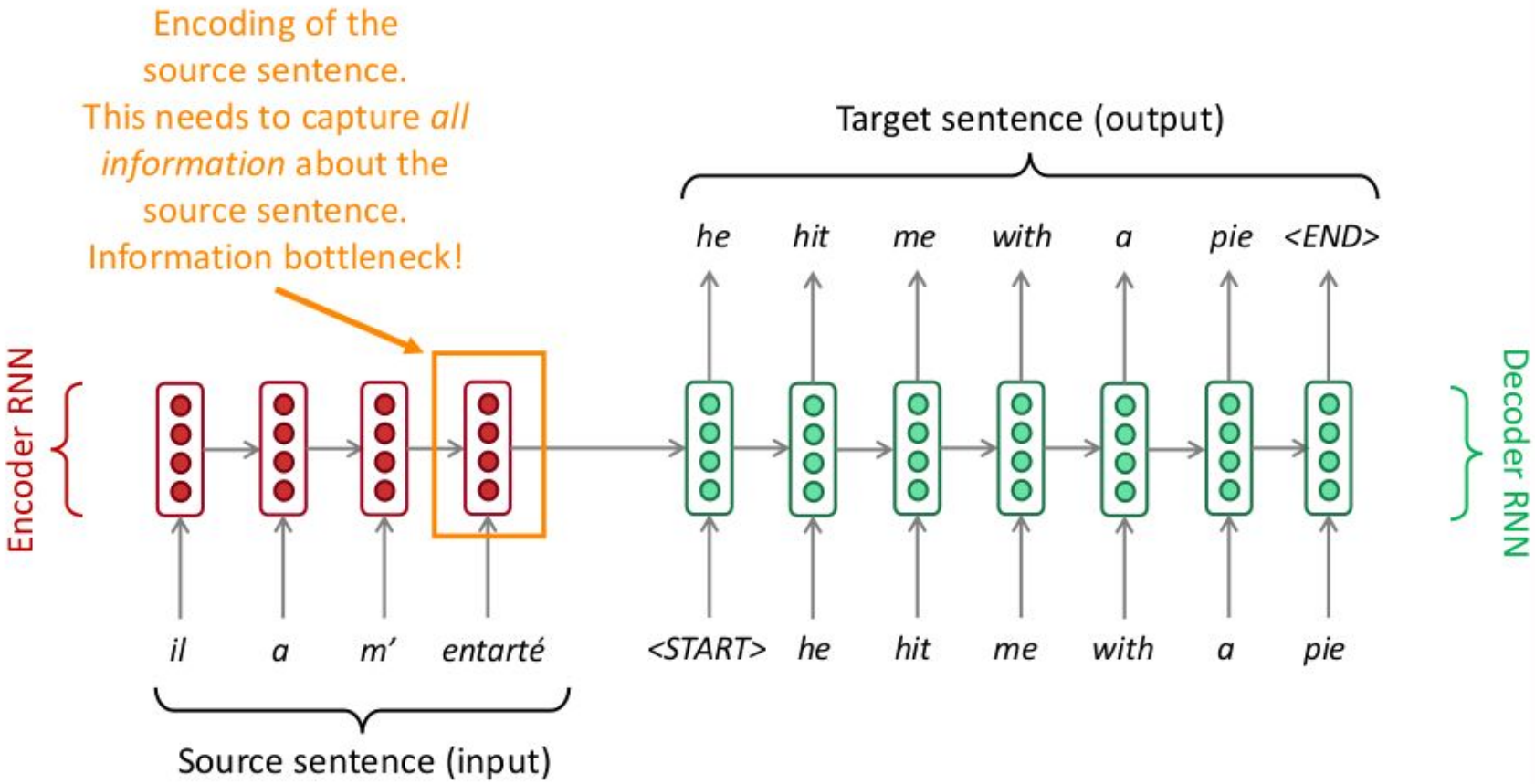
But who decides which words to attend to when generating/decoding output word?

Sequence-to-sequence: the bottleneck problem



Problems with this architecture?

Sequence-to-sequence: the bottleneck problem



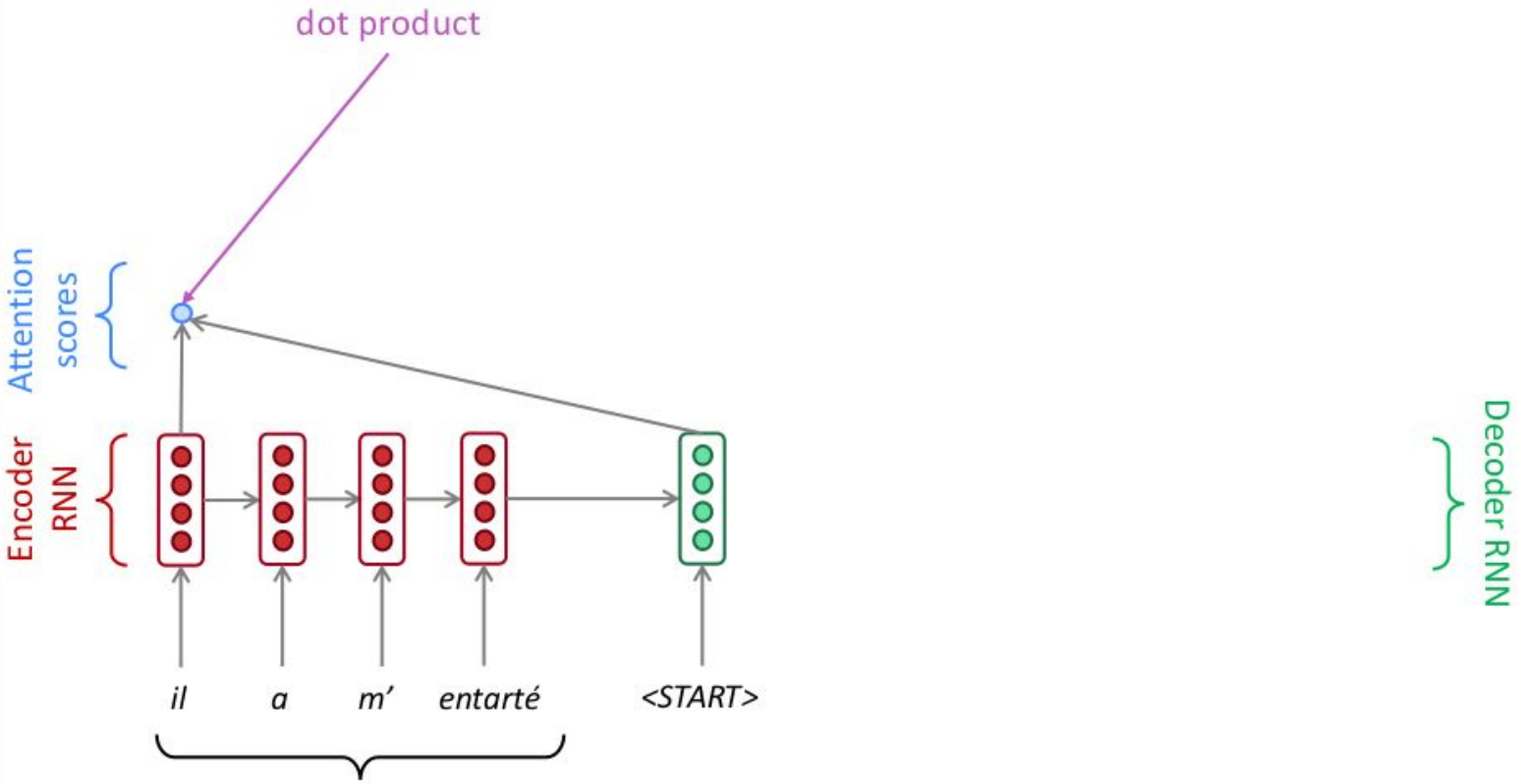
Attention

- **Attention** provides a solution to the bottleneck problem.

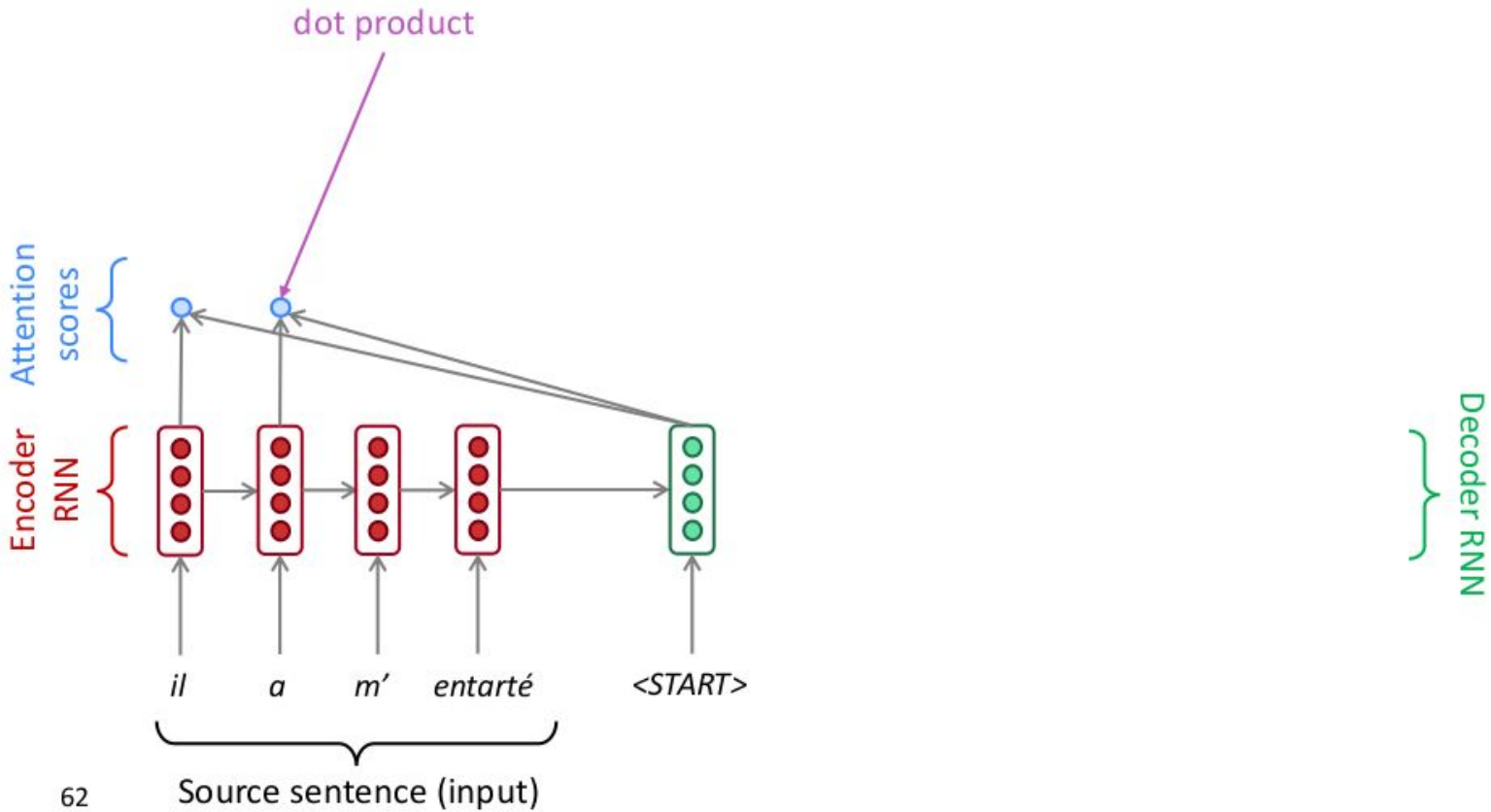
- Core idea: on each step of the decoder, use *direct connection to the encoder* to *focus on a particular part* of the source sequence

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

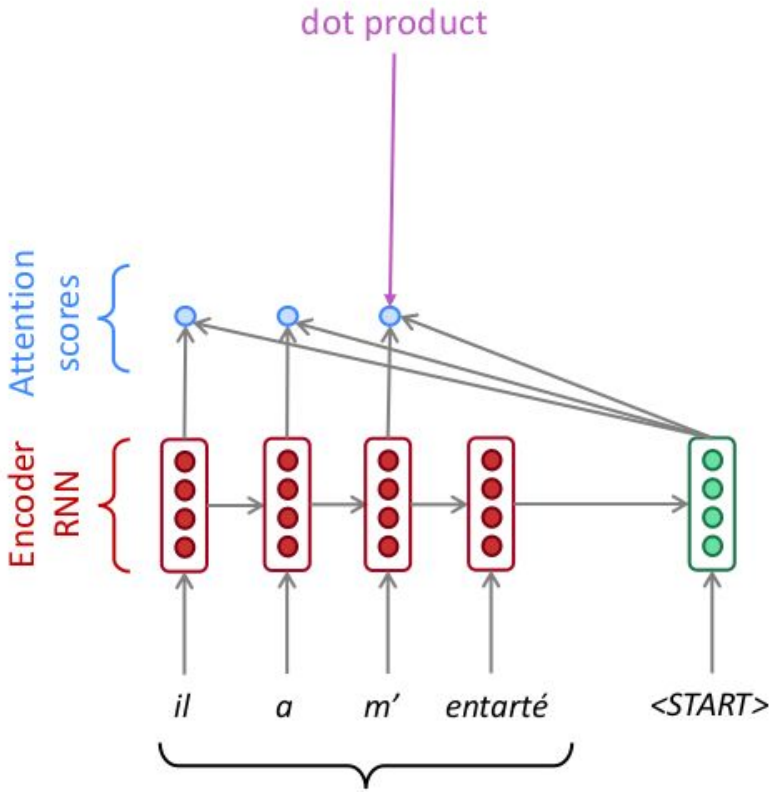
Sequence-to-sequence with attention



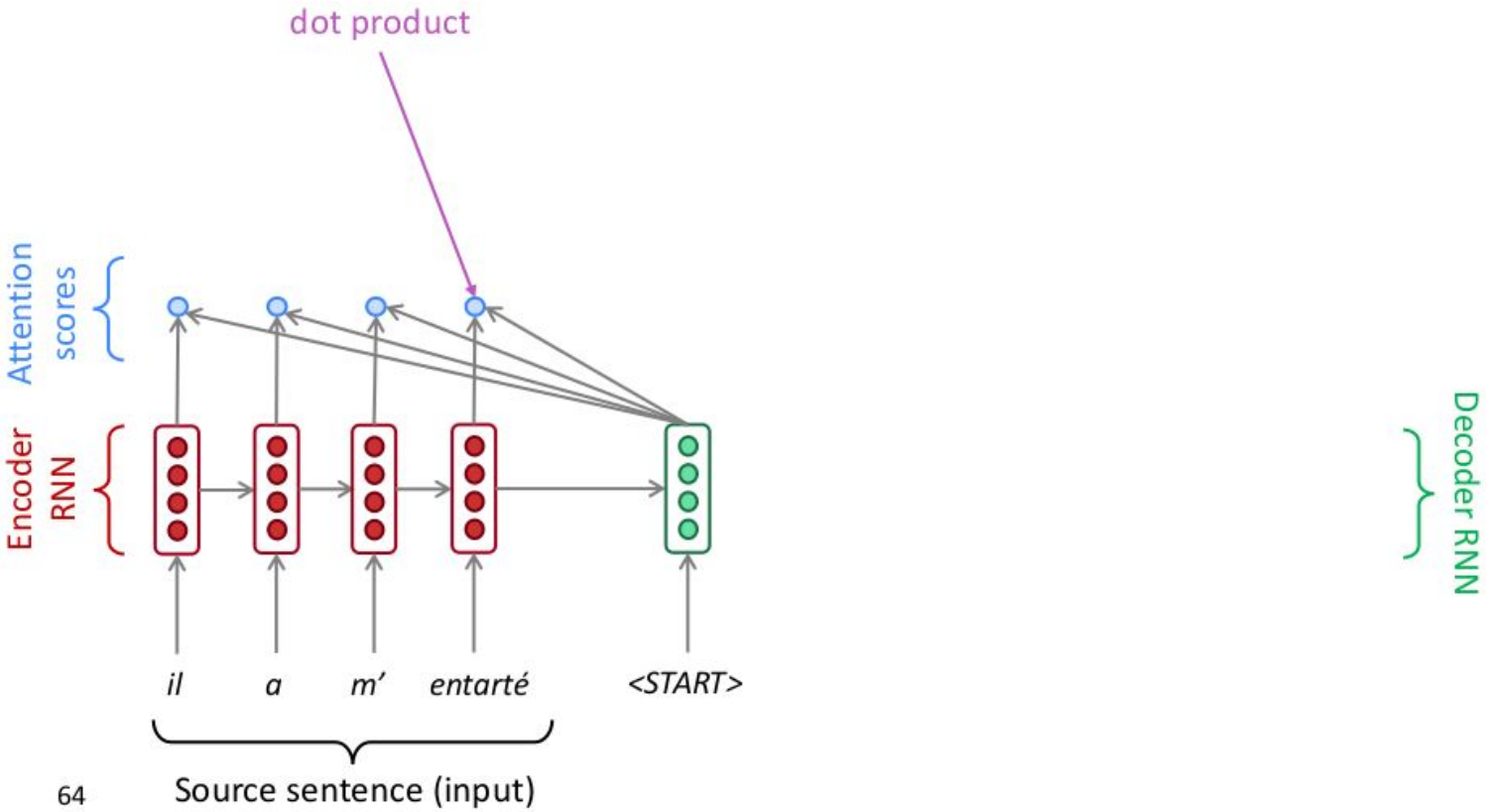
Sequence-to-sequence with attention



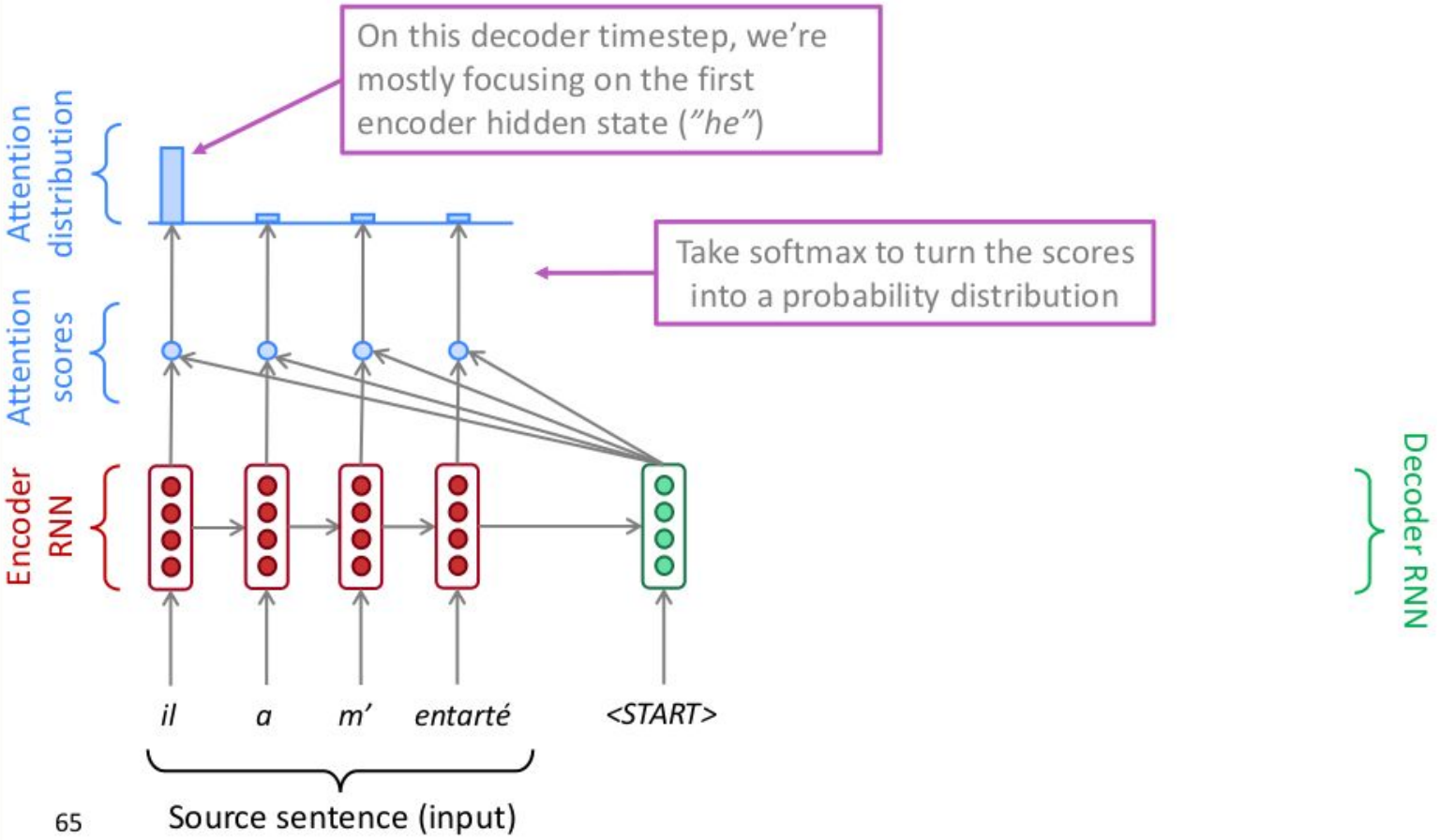
Sequence-to-sequence with attention



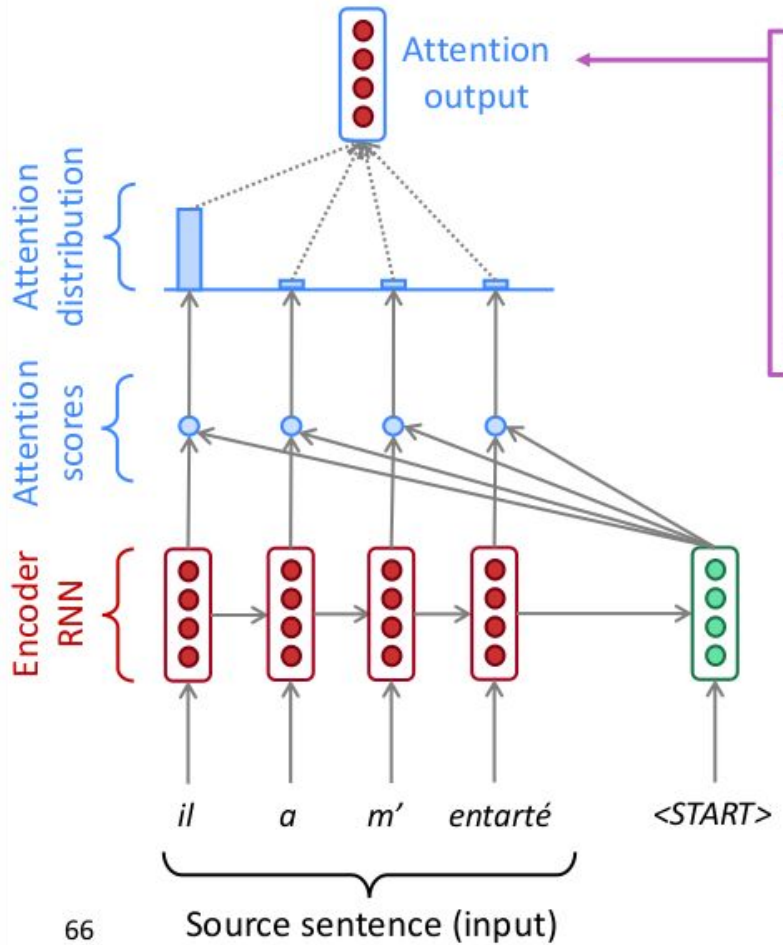
Sequence-to-sequence with attention



Sequence-to-sequence with attention



Sequence-to-sequence with attention

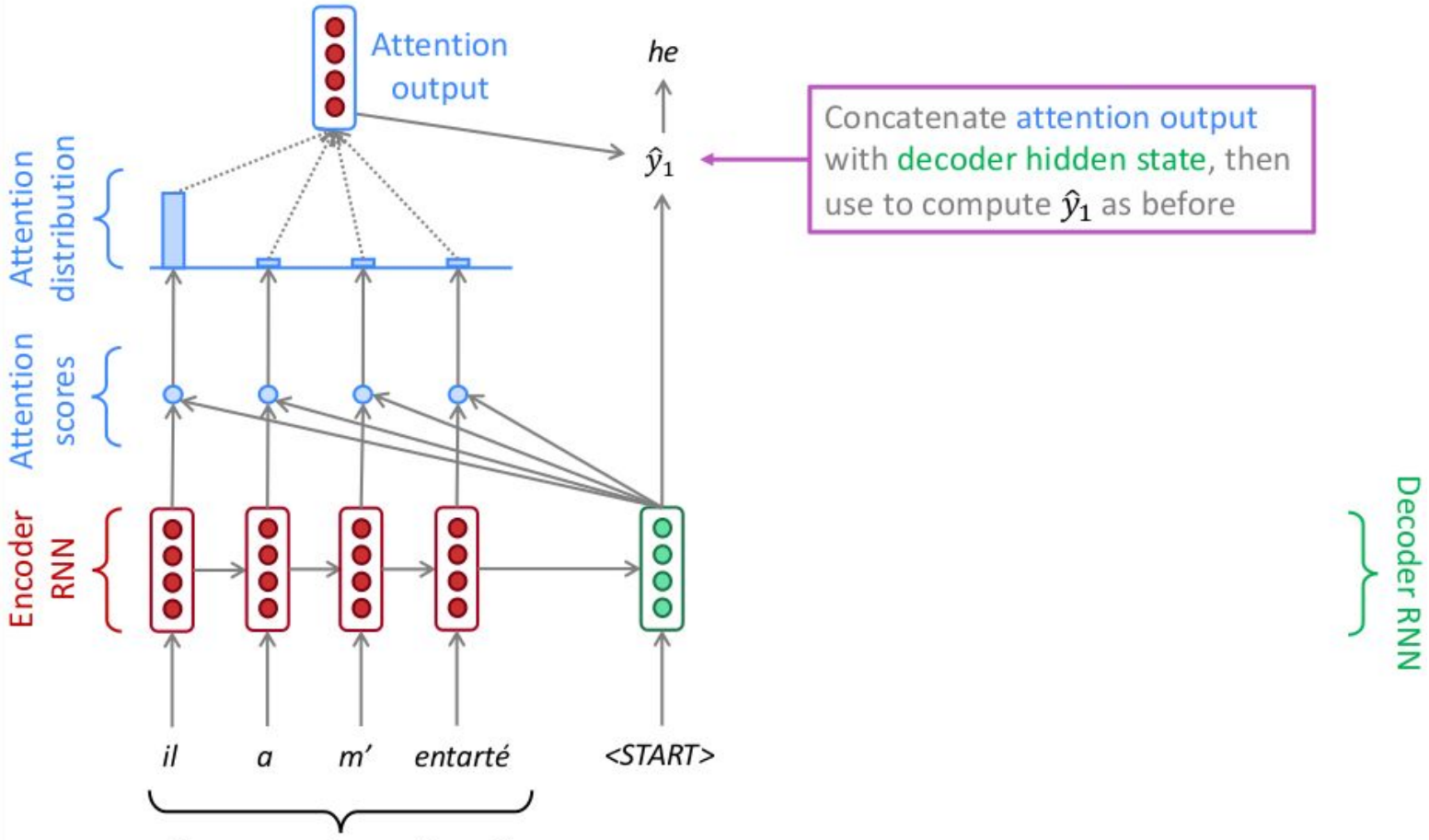


Use the attention distribution to take a **weighted sum** of the encoder hidden states.

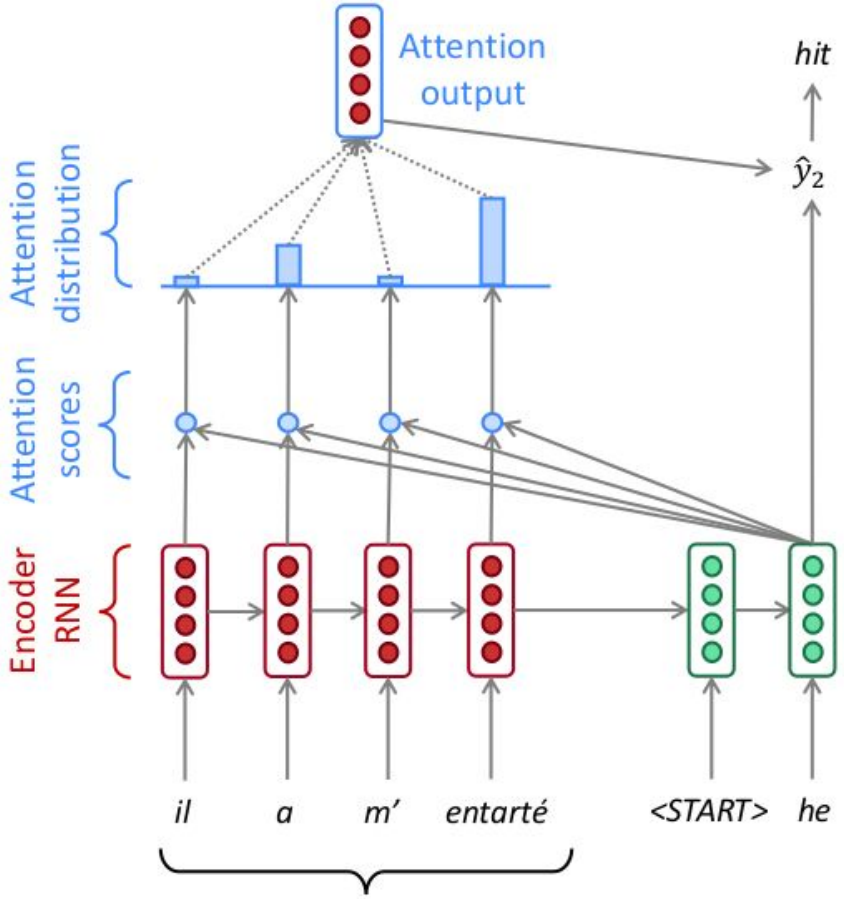
The attention output mostly contains information from the hidden states that received high attention.

Decoder RNN

Sequence-to-sequence with attention



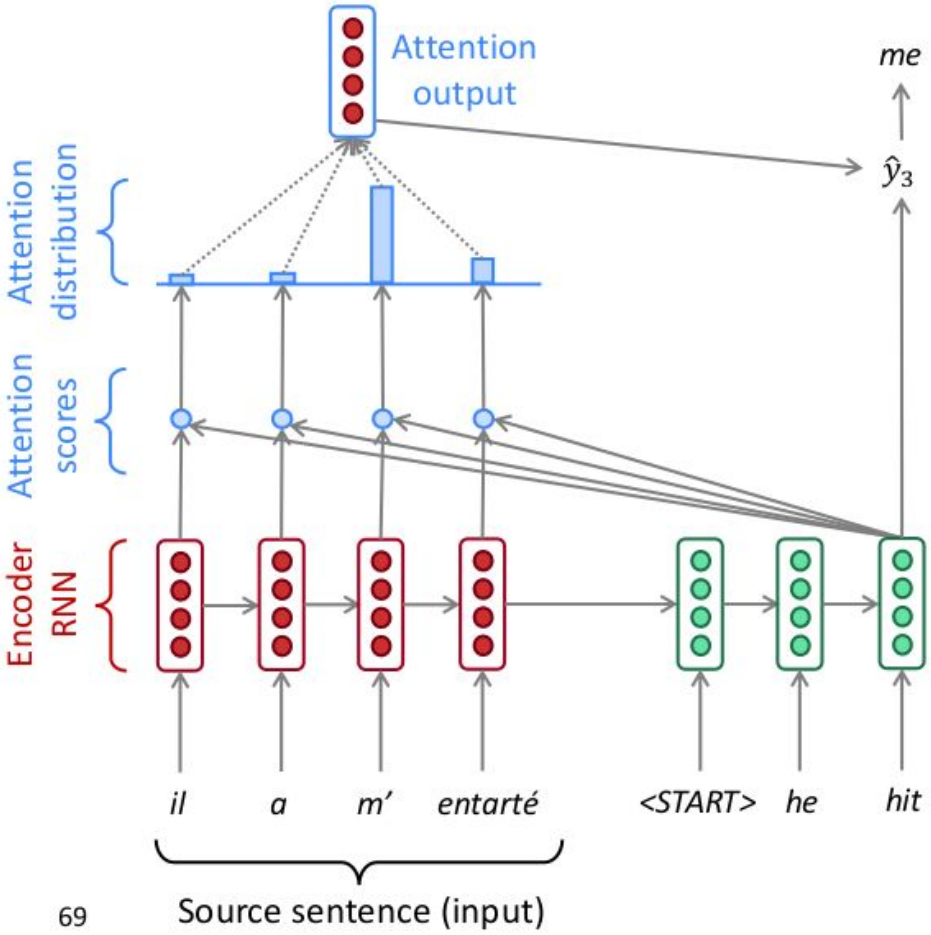
Sequence-to-sequence with attention



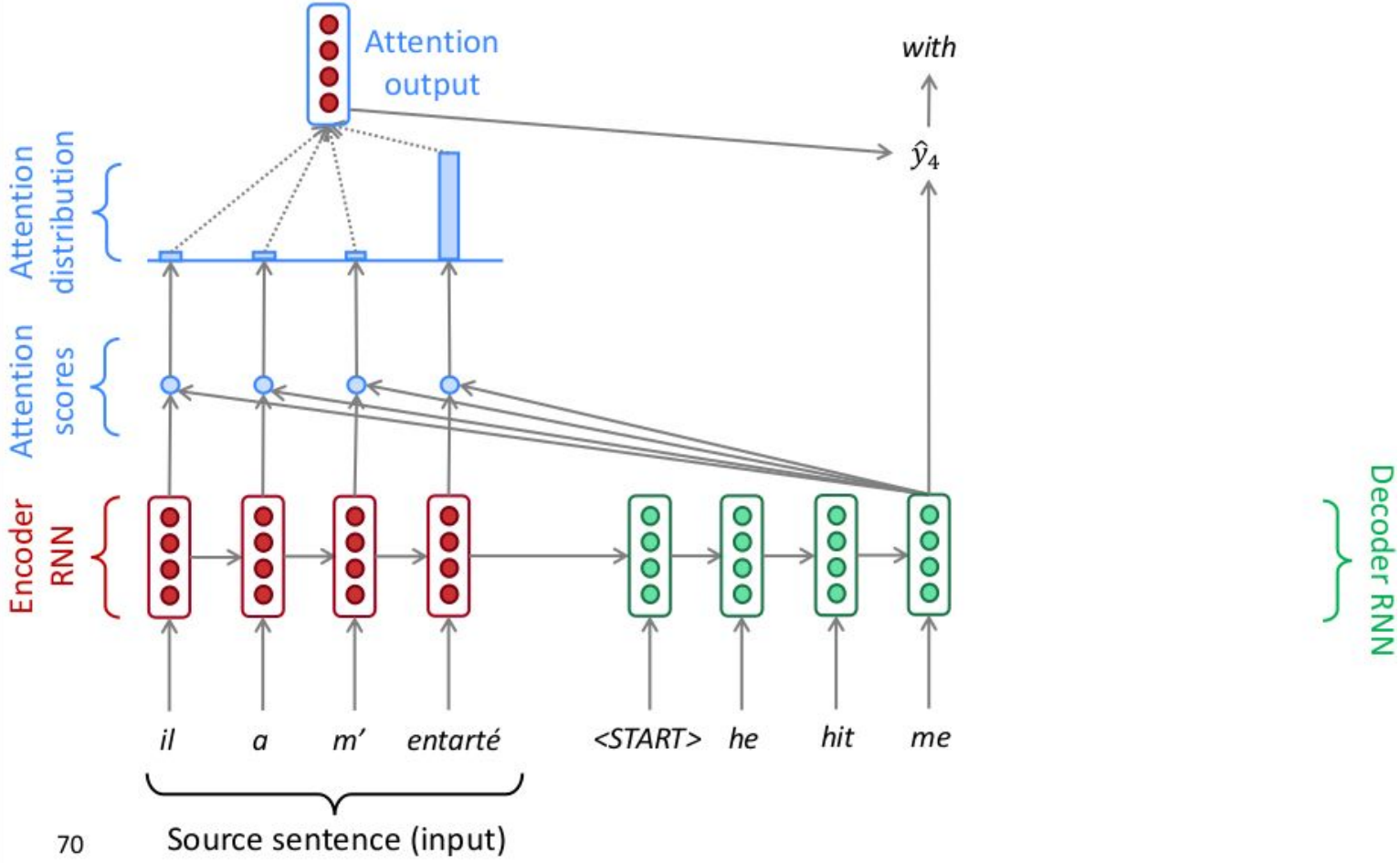
Sometimes we take the attention output from the previous step, and also feed it into the decoder (along with the usual decoder input). We do this in Assignment 4.

Decoder RNN

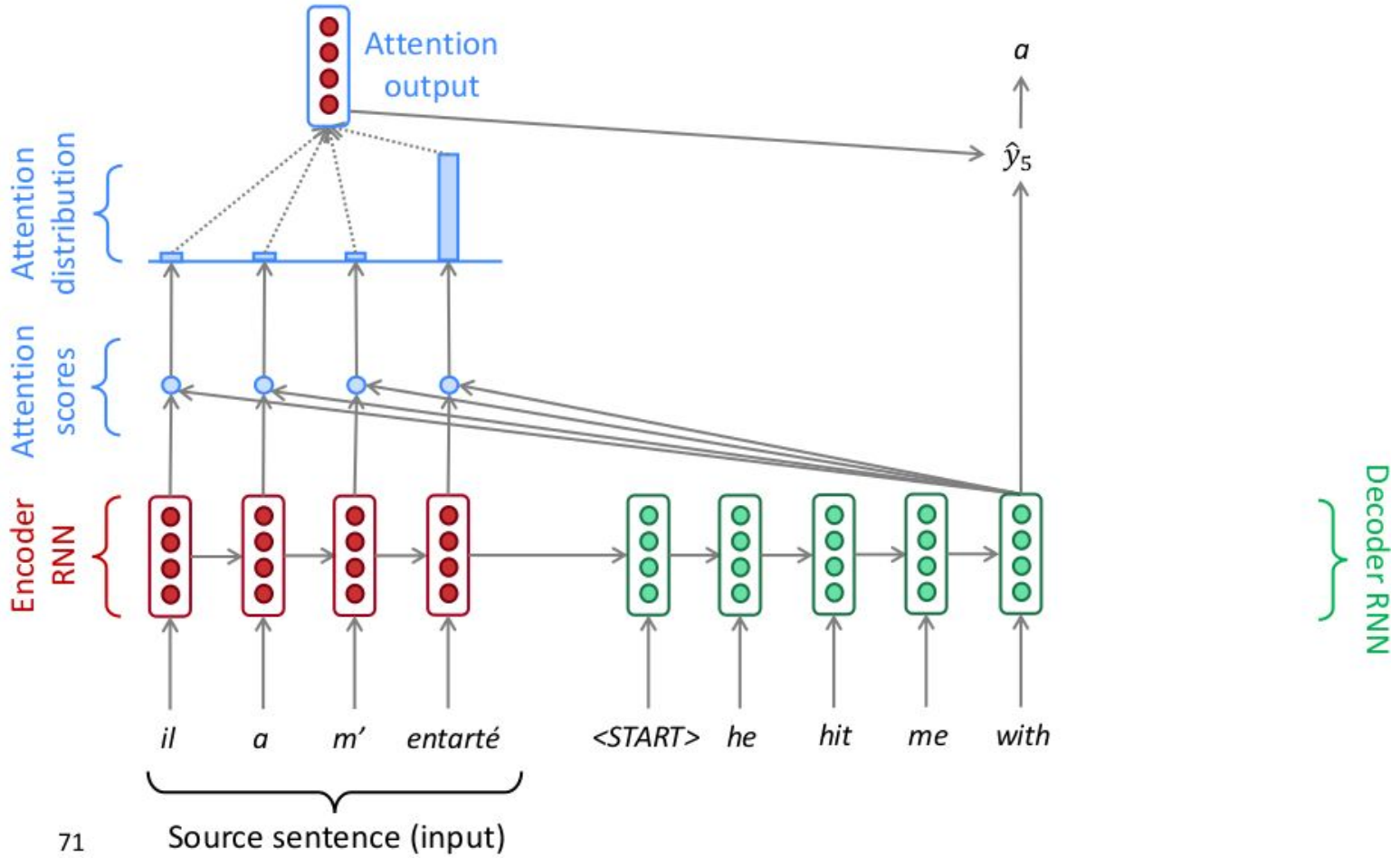
Sequence-to-sequence with attention



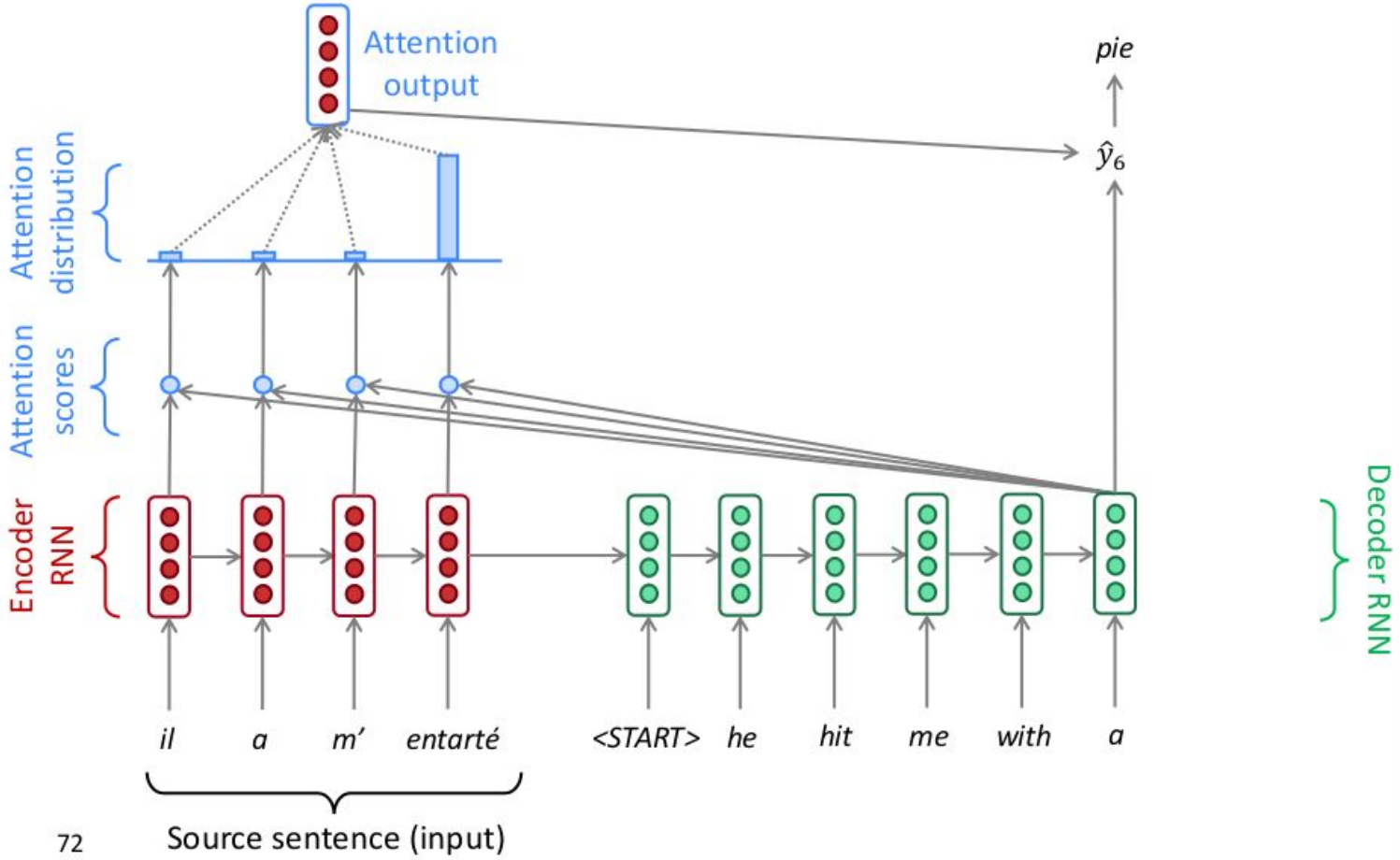
Sequence-to-sequence with attention



Sequence-to-sequence with attention



Sequence-to-sequence with attention



Attention: in equations

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

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

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

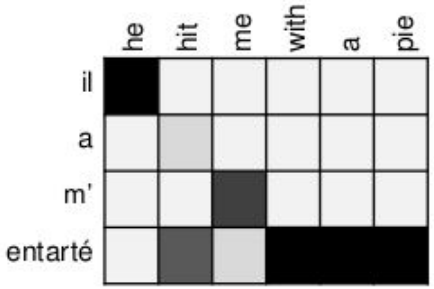
$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

- 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

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

Attention is great

- Attention significantly improves NMT performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with vanishing gradient problem
  - Provides shortcut to faraway states
- Attention provides some interpretability
  - By inspecting attention distribution, we can see what the decoder was focusing on
  - We get (soft) alignment for free!
  - This is cool because we never explicitly trained an alignment system
  - The network just learned alignment by itself



Attention is a *general* Deep Learning technique

- We've seen that attention is a great way to improve the sequence-to-sequence model for Machine Translation.
- However: You can use attention in many architectures (not just seq2seq) and many tasks (not just MT)

- **More general definition of attention**:
  - Given a set of vector *values*, and a vector *query*, **attention** is a technique to compute a weighted sum of the values, dependent on the query.

- We sometimes say that the query *attends to* the values.
- For example, in the seq2seq + attention model, each decoder hidden state (query) *attends to* all the encoder hidden states (values).

Attention is a *general* Deep Learning technique

More general definition of attention:

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

Intuition:

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

There are *several* attention variants

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

- Attention always involves:
  1. Computing the *attention scores* $\boldsymbol{e} \in \mathbb{R}^N$ ← There are multiple ways to do this
  2. Taking softmax to get *attention distribution* α:

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

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

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

thus obtaining the *attention output* $\boldsymbol{a}$ (sometimes called the *context vector*)

Attention variants

You'll think about the relative advantages/disadvantages of these in Assignment 4!

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

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

More information:

"Deep Learning for NLP Best Practices", Ruder, 2017. http://ruder.io/deep-learning-nlp-best-practices/index.html#attention

"Massive Exploration of Neural Machine Translation Architectures", Britz et al, 2017, https://arxiv.org/pdf/1703.03906.pdf

Bahdanau vs Luong Attention

Bahdanau attention:

- The idea of attention in seq2seq was introduced in [Neural Machine Translation by Jointly Learning to Align and Translate]:
-

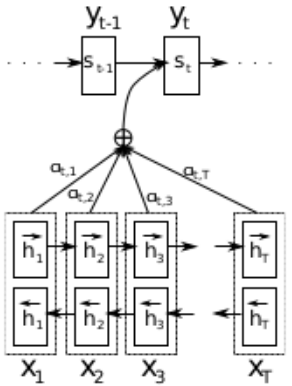


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

The decoder LSTM will have the following inputs:

- previous outputs y_{t-1}
- state composed of concatenation of:

1- previous state s_{t-1} = $[h_{t-1}, C_{t-1}]$

2- context vector c_t = weighted/gated encoder states h_j = $\sum_{j=0}^{j=T} \alpha_{tj} h_j$

α_{tj} are the ***attention*** weights. Those can be visualized as the importance of input word j when decoding output word t:

Bahdanau vs Luong Attention

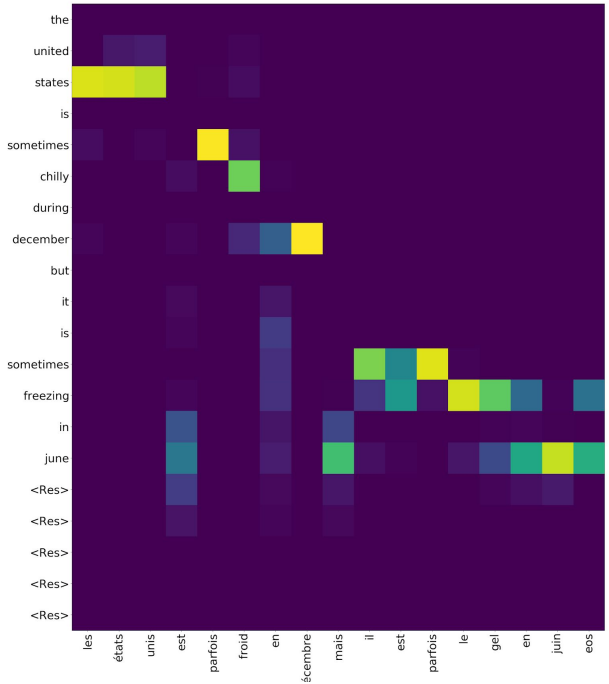
Bahdanau attention:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

$$e_{ij} = a(s_{i-1}, h_j)$$

$$h_j = \left[\overrightarrow{h}_j^\top ; \overleftarrow{h}_j^\top\right]$$



How to find those weights?

The alphas can be learnable. However, they must be related somehow to the simialrity between input word j and output word t.

In the paper Neural Machine Translation by Jointly Learning to Align and Translate, this simialrity measure was *leaned* using a feedforward NN a, which generates a score e_{tj} based on leanable weights over the last decoder state and the encoder state: $e_{tj} = a(s_{t-1}, h_j)$. Those scores are then normalized via simple *softmax*.

Bahdanau vs Luong Attention

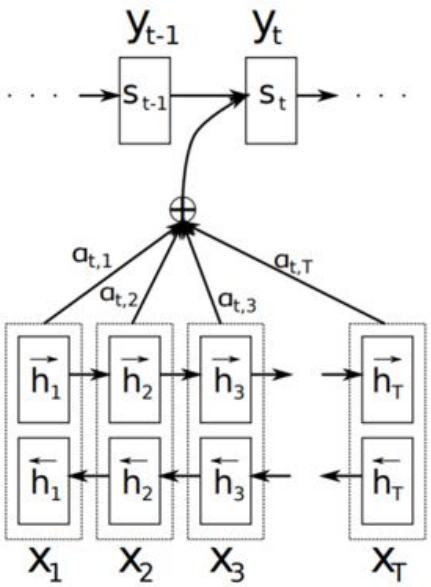
Luong attention:

In Effective Approaches to Attention-based Neural Machine Translation, other simialrity measures are studied to encode the *attention map weights* or α_{tj}. Three simialrity scores e_{tj} are studied:

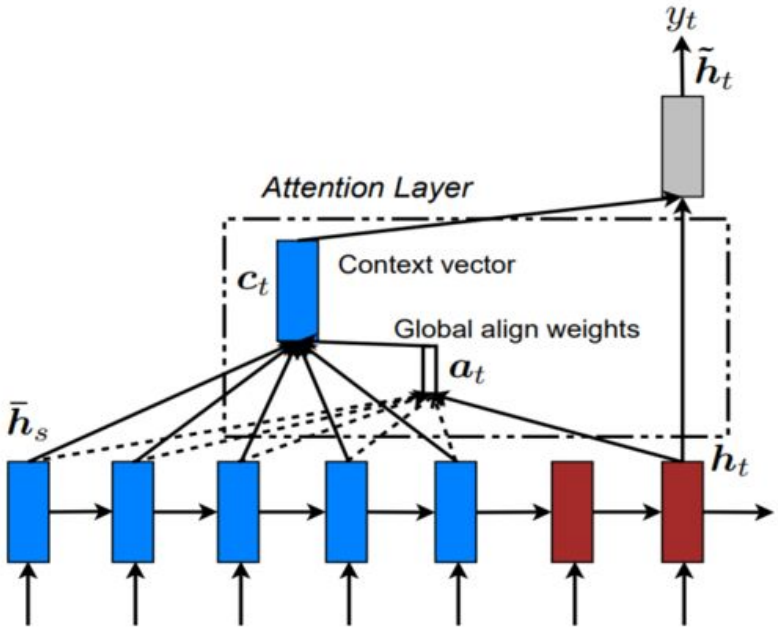
- general = weighted dot product
- concat = Bahdanau (weighted dot + tanh)
- dot = simple dot product

Again, the alphas are just simple softmax: $\alpha_{tj} = softmax(e_{tj})$ over $j \in [0, T]$ in the encoder state h_j Moreover, in Bahdanaus, in Bi-LSTM, both directions states are concatenated. While in Luong, only the top one is used.

Bahdanau vs Luong Attention



Bahdanau attention mechanism



Luong attention mechanism

Bahdanau vs Luong Attention

- Luong is more hierarchical: h[t] = f(ht, ct(hs)) → st=f(h[t-1], h[t]). Context (softmax) weights attends only on current state. LSTM hidden states are then evolving normally as in normal LSTM equations.
- Bahdanau one shot: st=f(st-1, ct(st-1,ht)) → context weights (softmax) is built starting/attending over the last state
- **Additive vs Multiplicative**

$$\alpha_{ts} = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \quad \text{[Attention weights]} \quad (1)$$

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \quad \text{[Context vector]} \quad (2)$$

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W_c}[\boldsymbol{c}_t; \boldsymbol{h}_t]) \quad \text{[Attention vector]} \quad (3)$$

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1} \boldsymbol{h}_t + \boldsymbol{W_2} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases} \quad (4)$$

Let’s code

- Luong attention NMT ⇒ Functional API
- https://colab.research.google.com/drive/1dhlc3Nt_LvZcxY5tUd-XLU1fvGt4pPuh?usp=sharing
- Luong vs. Bahdanau Eager:
  - Luong: https://colab.research.google.com/drive/1IqW7pOS5HXbkIsFQ6BDmVqqL3EAwOmN3?usp=sharing
  - Bahdanau: https://colab.research.google.com/drive/1Pne_EbX6rgKfZBIwvJ6OOqVv50UceRAO?usp=sharing
  -

Let’s code: Spell → Char vs. Word

Char vs. Word Spell:

- Functional API: https://colab.research.google.com/drive/1rq6LcPASrXZptefiAG_GxIJY3znVfAwg?usp=sharing
- Sub classing + Eager:
  - Char+Bahdanau: https://colab.research.google.com/drive/1rq6LcPASrXZptefiAG_GxIJY3znVfAwg?usp=sharing
  - Word + Bahdanau: https://colab.research.google.com/drive/1rq6LcPASrXZptefiAG_GxIJY3znVfAwg?usp=sharing
  - Word + Luong: https://colab.research.google.com/drive/1rq6LcPASrXZptefiAG_GxIJY3znVfAwg?usp=sharing
  -

Let’s code

Char vs. Word level NMT:

https://colab.research.google.com/drive/1dhlc3Nt_LvZcxY5tUd-XLU1fvGt4pPuh?usp=sharing

Let’s code

Char Level (char2char)

Now let's encoder the sentences as sequence of chars instead of words. While it's not clear why we need to do that in NMT application, the need is more clear in other applications that has the possibility of having mistakes in the input, leading to OOV words. Examples:

- Spelling correction
- Chatbots

Let’s code

Char Level (char2char)

Char level models have the following pros:

- No OOV: we cannot encounter a char we don't know. In word level models, there's a risk of having new word not in out vocab.
- number_target_tokens are limited by the length of chars vocab, vs. large number of words vocab, which makes a huge softmax layer. This increase the memory and the model complexity, leading to risk of overfitting. Also, making it hard to learn ALL words softmax weights, since they are rarely encountered as output. Finally, dominant stop words might create class imbalance issue.

Let’s code

Char Level (char2char)

On the other hand, it has the following cons:

- max_encoder_seq_len and max_decoder_seq_len is longer than word level case. This makes it harder for the LSTM to learn the sequence states, leading to the following issue:
- Hallucination: since the sequence is long, the LSTM state capture is harder. Moreover, the model knows nothing about the notion of *word*, leading to: 1) invalid words and 2) invalid words sequence, since the language model now is at char level, and the sequence is long, so the errors in language modeling are higher.
- Error propagation: during inference, seq2seq models are sampled in a sequential order. The decoder inputs are now the feedback of decoder output. This leads to error propagation. The issue exists in both word and char level models. However, it is more severe in char level case, due to longer sequence, making it harder to recover. This also leads to hallucination (invalid words and/or sequence).

Let’s code

Possible fixes:

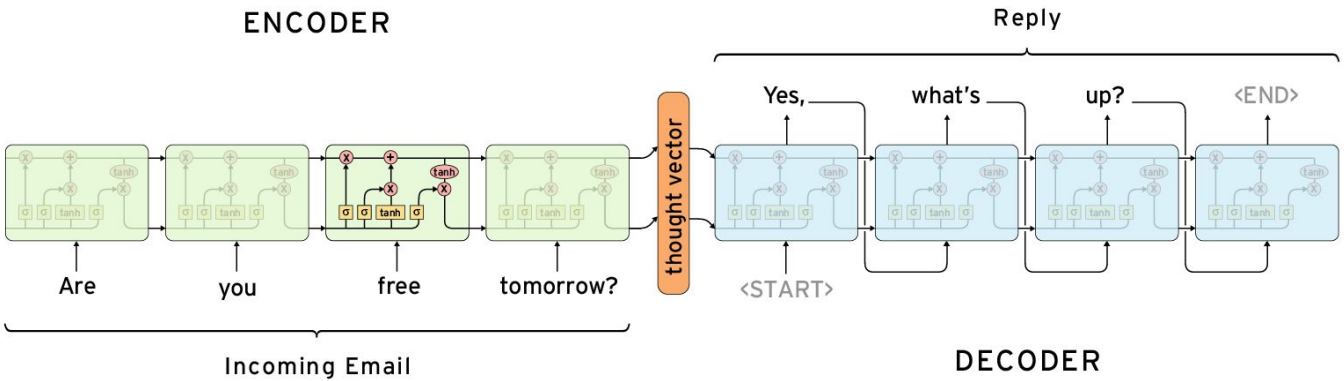
- Custom loss: after the char2char model generates its output, decoder chat by char, then split/tokenize into `decoded_seq` as sequence of words. This enables to calculate a custom loss at the word level (which we care about). Possible losses are:
  - CTC loss: `ctc_loss(gt_seq, decoded_seq)`
  - WER loss: `calculate_WER(gt_seq, decoded_seq)`
- char2word: input=char level, output=word level
  - Learn word boundaries/notion end2end
  - Leading to similar situation as custom loss; the loss will be higher as long as we don't decode the exact needed word.
  - Could be integrated with pre-trained decoder on word level NLM.

Transformers

Attention is all you need!

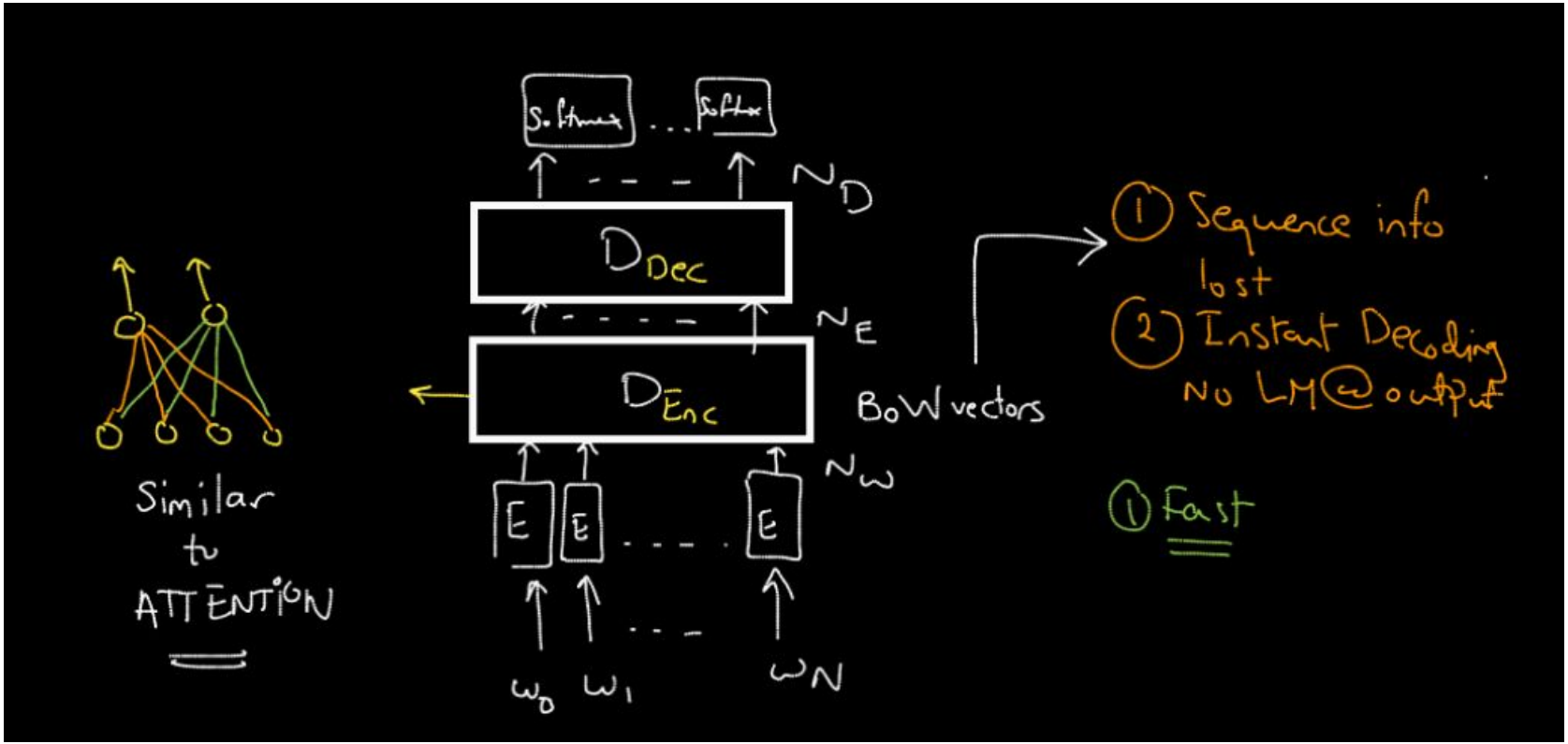
Issues with LSTM seq2seq

- LSTM is slow → No use of parallelization (GPU)
- Sequence information preserved

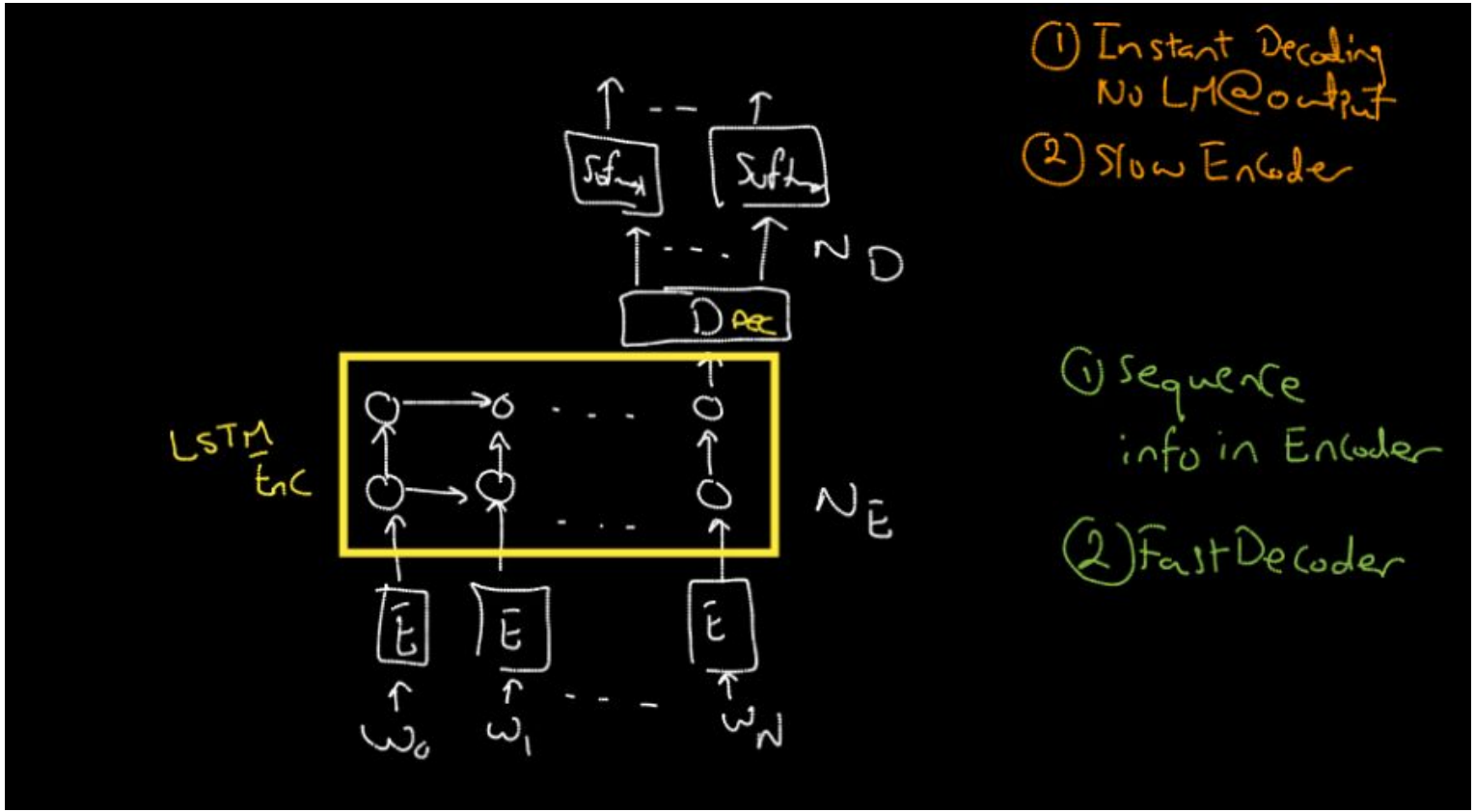


What are other options?

Dense-Dense


Softmax
Softmax
z_D
D_{Dec}
z_E
D_{Enc}
BoW vectors
z_W
E
E
E
w_0 w_1 w_N
Similar to ATTENTION
① Sequence info lost
② Instant Decoding No LM@output
① Fast

LSTM-Dense


① Instant Decoding
No LM@output
② Slow Encoder
Softmax
Softmax
N D
Dec
LSTM Enc
N E
E
E
E
ω0
ω1
ωN
① sequence info in Encoder
② Fast Decoder

Could we do the same with Dense?

Dense-Dense:

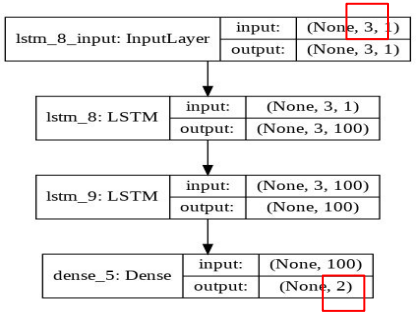
- No summarizing state → Add LSTM Encoder→ Instant decoding → Only for matched case
- Another major issue→ position information is lost!
- LSTM is slow, but Dense is fast

```
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps_in, n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_steps_out))
```

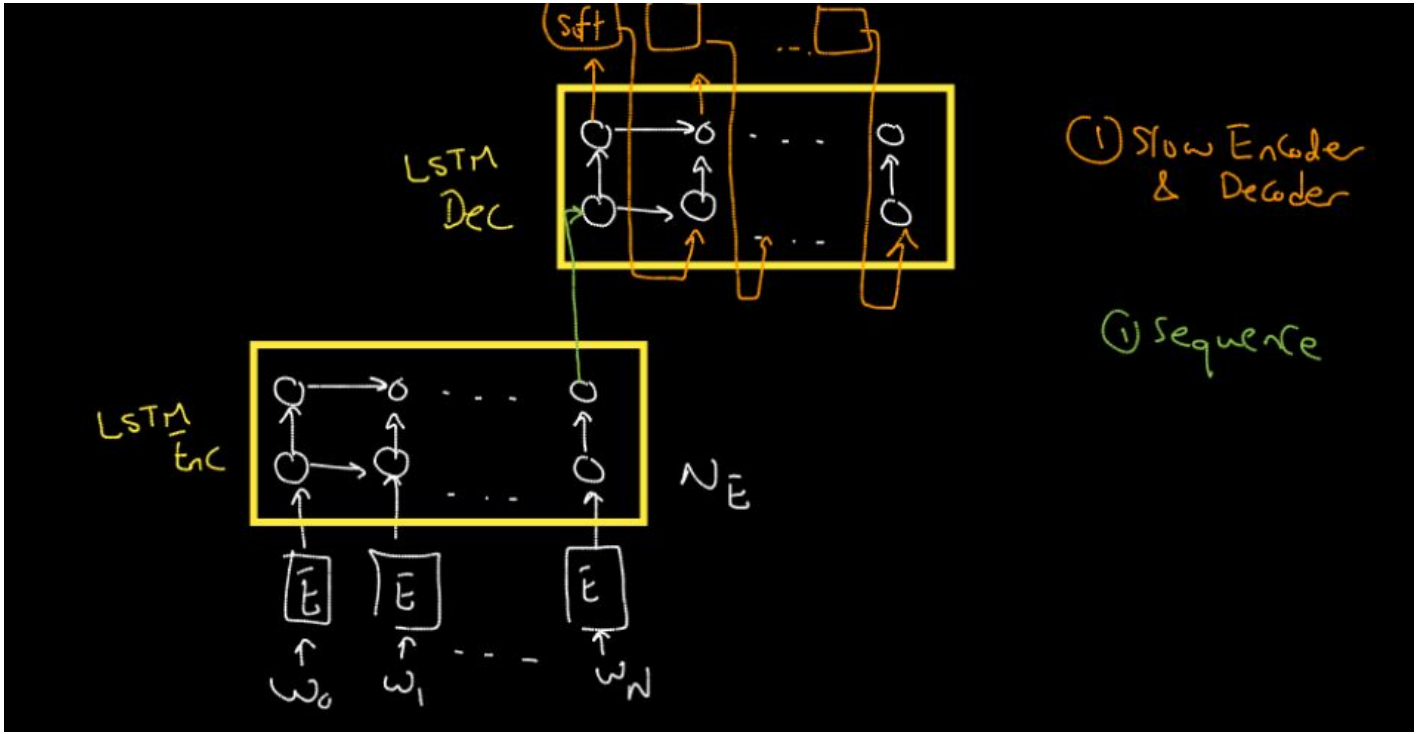
```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_8 (LSTM)                (None, 3, 100)            40800
_________________________________________________________________
lstm_9 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_5 (Dense)              (None, 2)                 202
=================================================================
Total params: 121,402
Trainable params: 121,402
Non-trainable params: 0
_________________________________________________________________
```

```
print(X.shape)
print(y.shape)
```

```
(5, 3, 1)
(5, 2)
```



LSTM-LSTM


Soft
LSTM Dec
LSTM Enc
E
E
E
w_0
w_1
w_N
N_E
① Slow Encoder & Decoder
① Sequence

Pros and Cons

Better (natural) model of temporal info

Longer Horizon of history

Less resources (Conv)

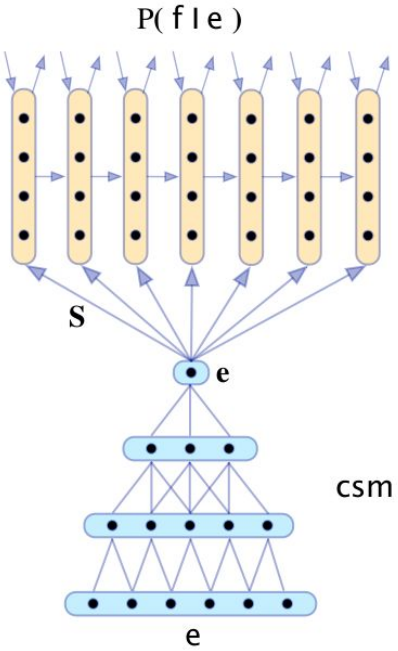
Slow (Sequential)

Sometimes hard to converge

Conv-RNN for MT

CNN application: Translation

- One of the first successful neural machine translation efforts
- Uses CNN for encoding and RNN for decoding
- Kalchbrenner and Blunsom (2013)
  "Recurrent Continuous Translation Models"
- Many more recent models we may cover in future lectures



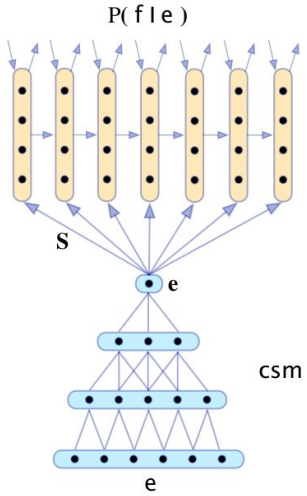
Conv-RNN for MT

- It captures sequence information and summarizes it in a **state**
- Sequence info preserved in **feature maps**

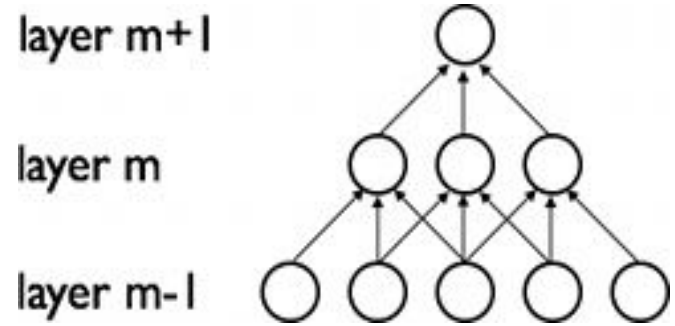
- Network depth scales with max seq (sentence) length!
  - sequence length requires deeper models, it makes it difficult to learn dependencies between distant words
- Slow Decoder (LSTM)
- Inefficient for variable length! Needs padding

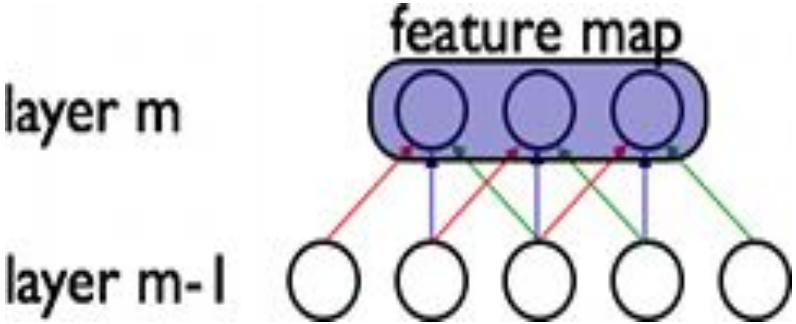
CNN application: Translation

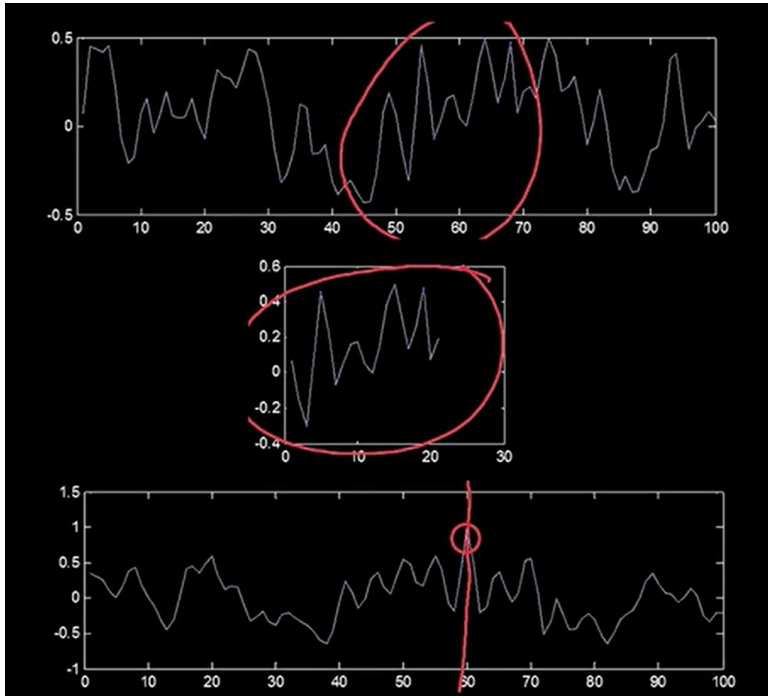
- One of the first successful neural machine translation efforts
- Uses CNN for encoding and RNN for decoding
- Kalchbrenner and Blunsom (2013) "Recurrent Continuous Translation Models"
- Many more recent models we may cover in future lectures



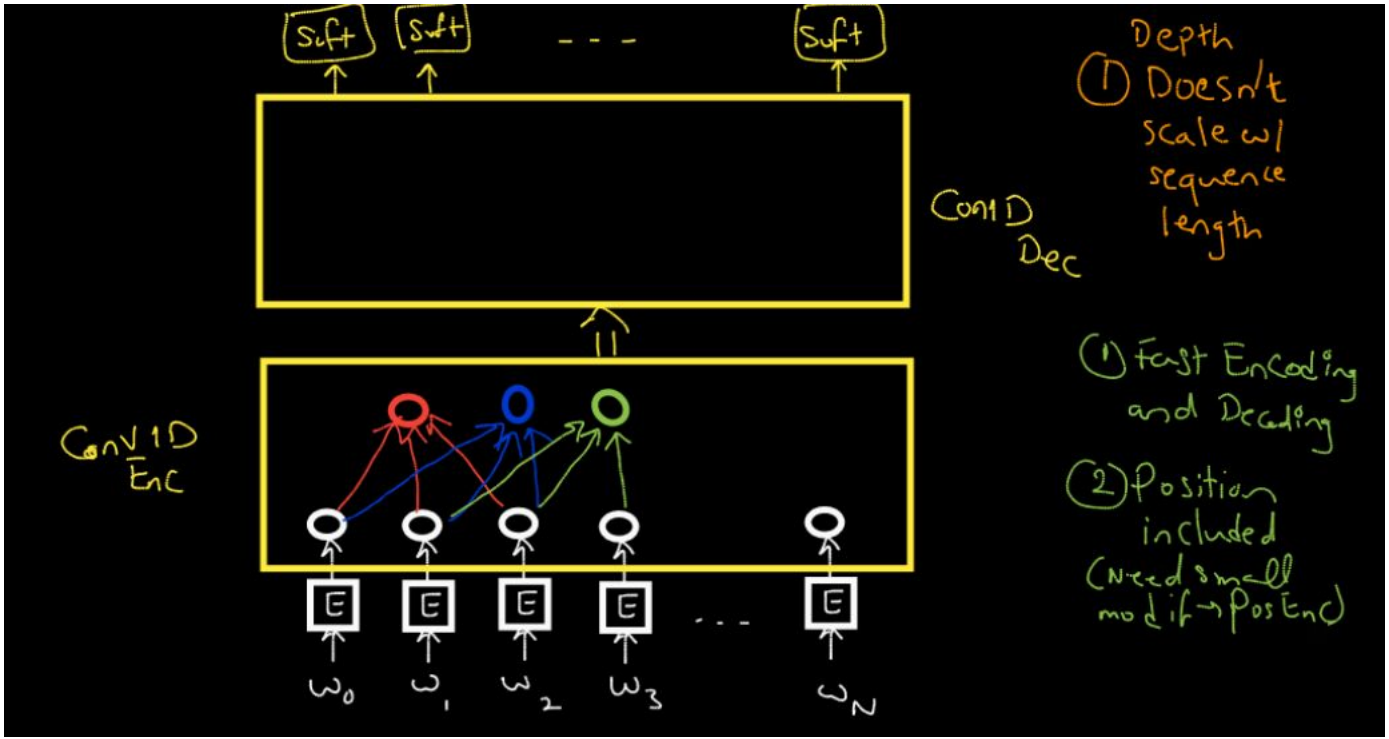
Conv1D


layer m+1
layer m
layer m-1


feature map
layer m
layer m-1

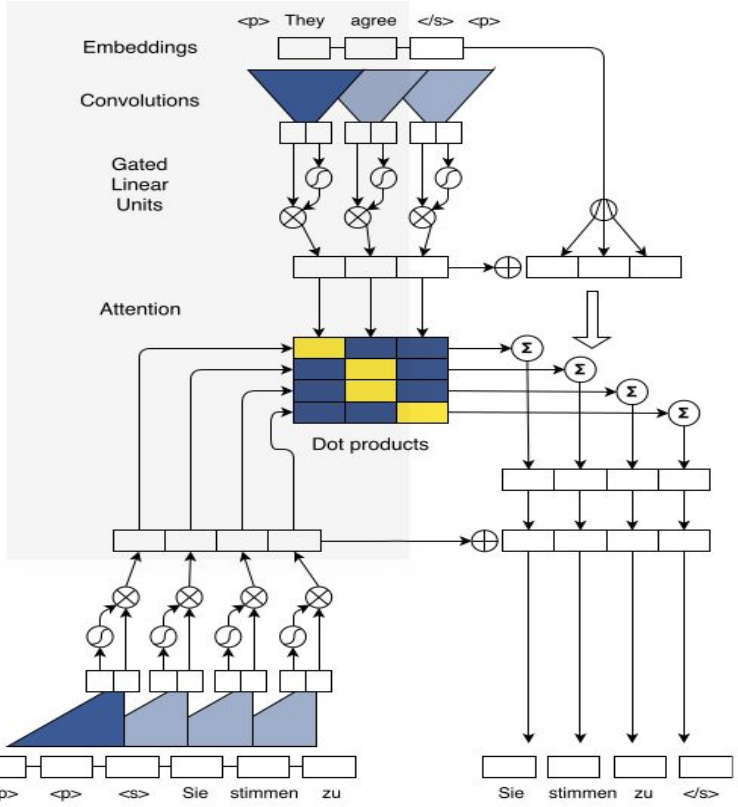

0.5
0
-0.5
0 10 20 30 40 50 60 70 80 90 100
0.6
0.4
0.2
0
-0.2
-0.4
0 10 20 30
1.5
1
0.5
0
-0.5
-1
0 10 20 30 40 50 60 70 80 90 100

Conv-Conv


Soft
Soft
Soft
Depth
① Doesn't scale w/ sequence length
Conv1D Dec
① Fast Encoding and Decoding
② Position included (Need small modif → PosEnc)
Conv1D Enc
E
w_0
w_1
w_2
w_3
w_N

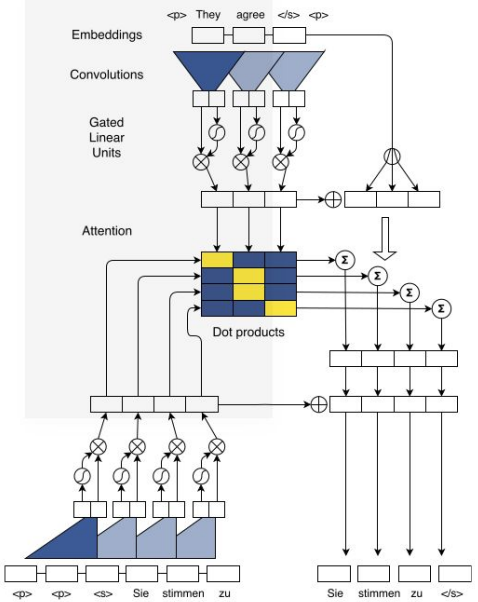
ConvS2S (+Attention)

ConvS2S paper



ConvS2S: ConvNet for Seq2Seq

- It captures sequence information and summarizes it in a **state**
- Sequence info preserved in **feature maps**
- Fast Encoder and Decoder
- Attention provides **learnable alignment**



- Network depth scales with max seq (sentence) length!
- Still needs extra position encoding (more on that later)
  - Although feature maps handles position, but it will be lost as depth increases.

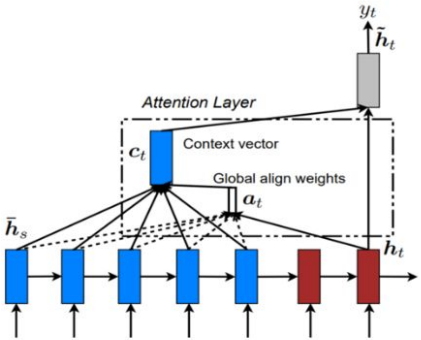
-

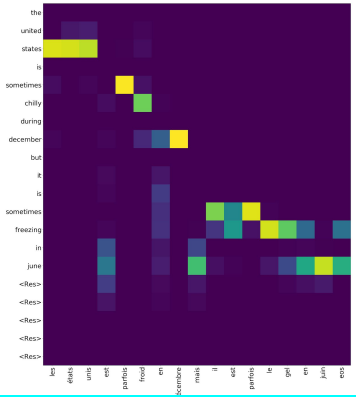
Speed + Performance!

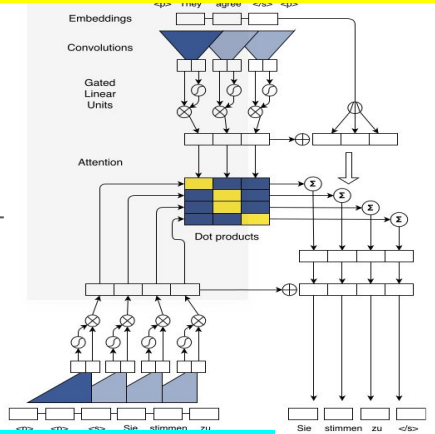
- BoW → Fast → No sequence
- RNN → For sequence! → Slow
- Conv/Dense → For speed! →Conv bad scaling with seq len

Can we take the best out of ALL?

Context is best handled by **attention** in both seq2seq and convs2s
Attention is just **Gating** mechanism → **Fast**



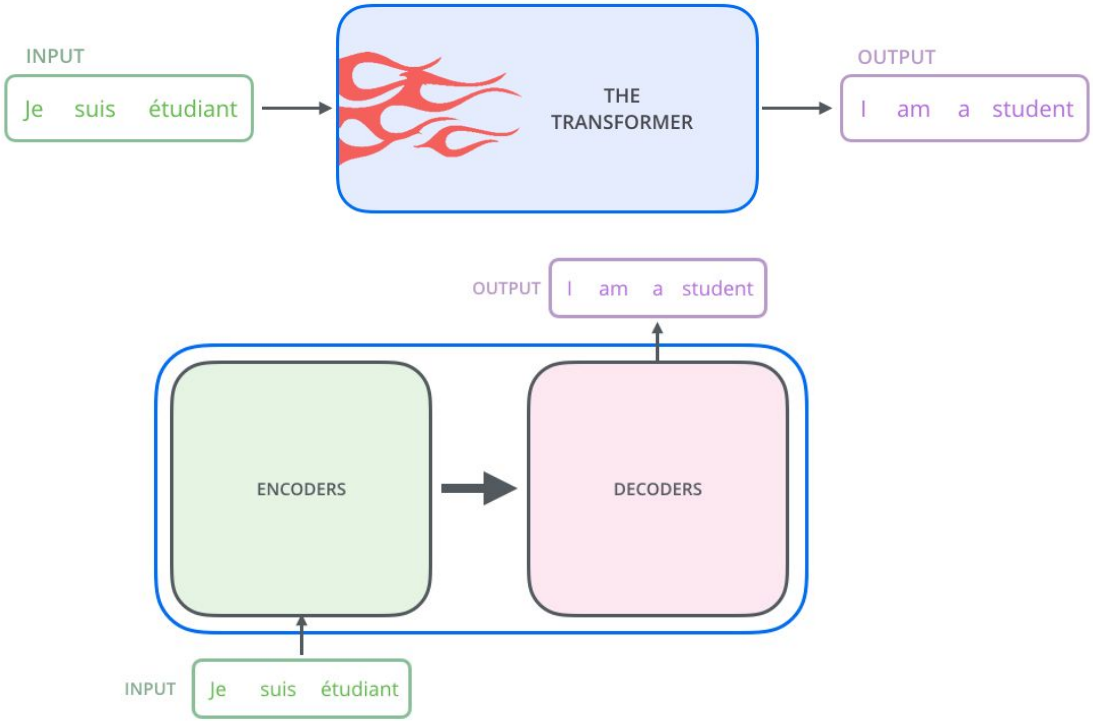




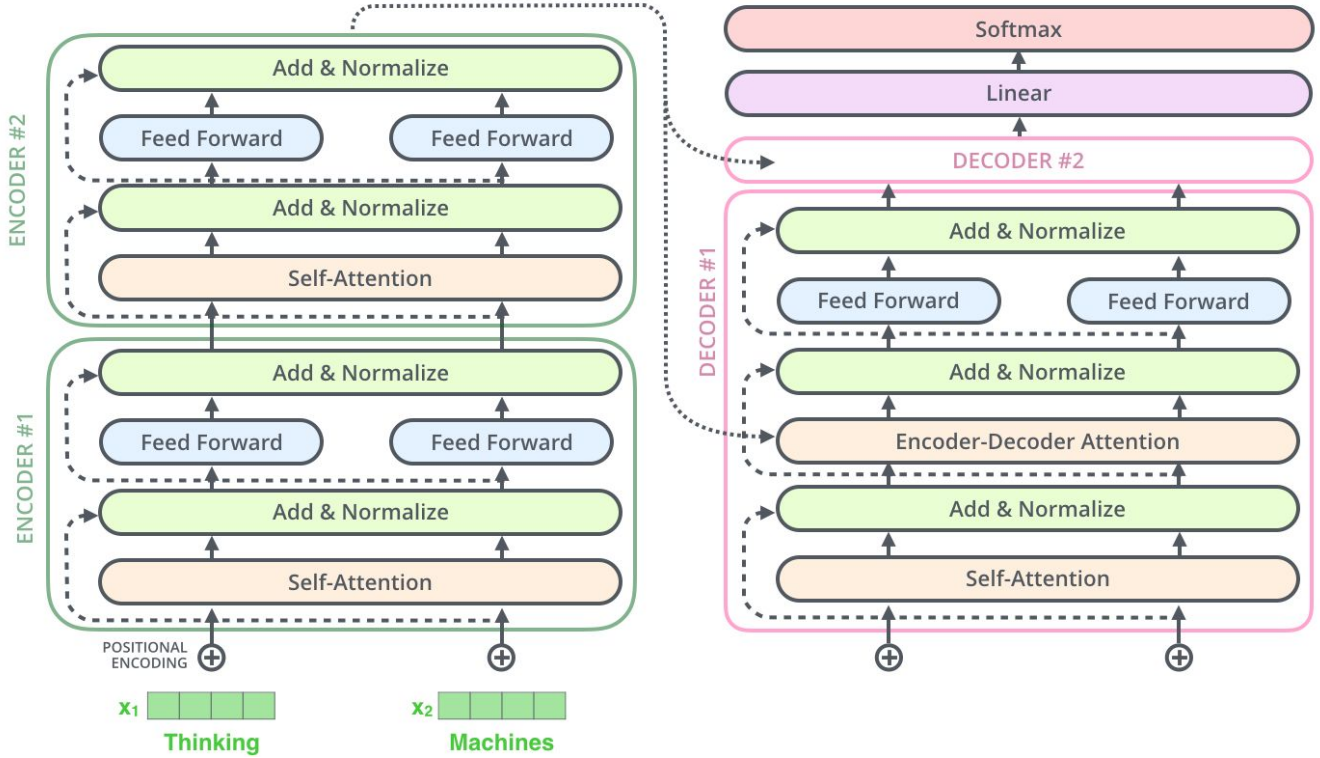
If you have the alignment weights → You have it all !
can efficiently encode and decode based on context

Attention is ALL you need!

Transformers



Transformer



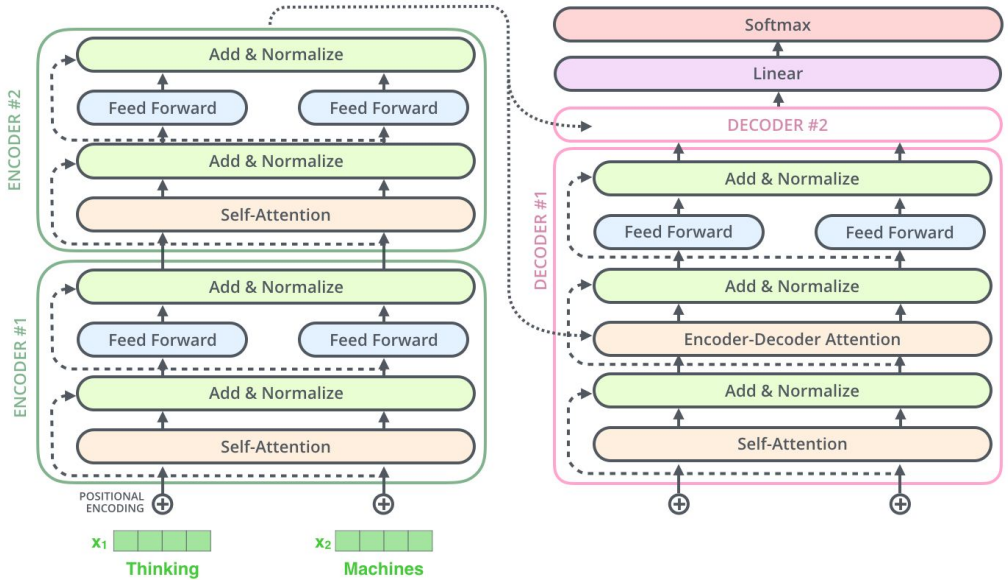
Building blocks step-by-step

Encoder:

- Self attention
- Skip
- Feed fwd
- Position Encoding

Decoder:

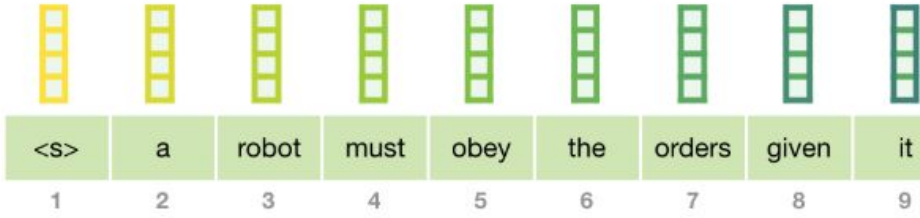
- Encoder-Decoder attention (masking)
- Skip
- Feed fwd
- Position Encoding
- Decoder feedback (teacher forcing)
- Softmax output



Main tricks

- All based on Attention Gating → No conv, No LSTM → Fast (like Dense/Conv)
- Attention handles context → sequence preserved
- Position encoding → Encode the position of the word in the sentence as **explicit feature** → sequence preserved (also in convs2s)
- Conditional decoding → Ability to feedback decoder output into the next token decoding step→ Sequential decoding + Teacher forcing (like RNN)

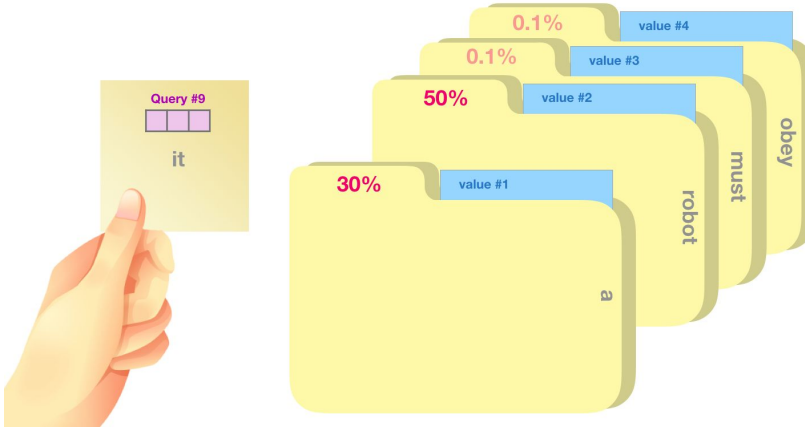
Self-Attention



Self-Attention Process

Self-attention is processed along the path of each token in the segment. The significant components are three vectors:

- Query: The query is a representation of the current word used to score against all the other words (using their keys). We only care about the query of the token we're currently processing.
- Key: Key vectors are like labels for all the words in the segment. They're what we match against in our search for relevant words.
- Value: Value vectors are actual word representations, once we've scored how relevant each word is, these are the values we add up to represent the current word.



http://jalammar.github.io/images/gpt2/self-attention-example-folders-scores-3.png

Self-Attention

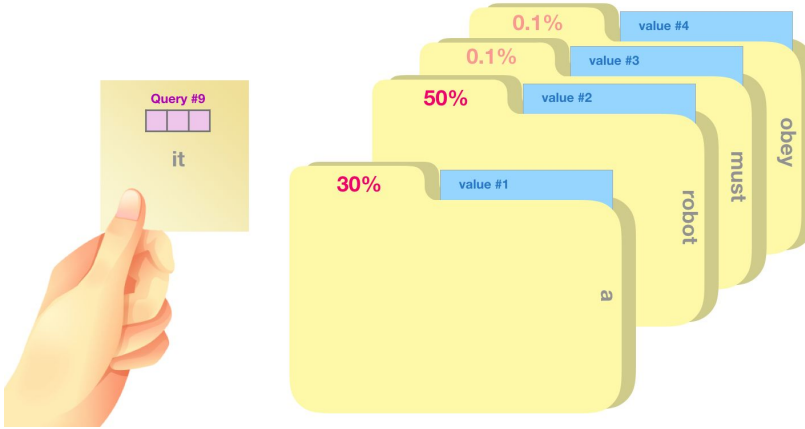
At each level of encoding the sequence, A *_Query_* vector asks all the context *_Key_* word vectors about a certain feature. The encoded feature is an aggregation over all context words vectors *_Values_*, each contributing according to its *_Key_* vector similarity to the *_Query_*.

Self-Attention

Query = A feature detector (is_verb, is_name, ….) **= Word as feature detector**

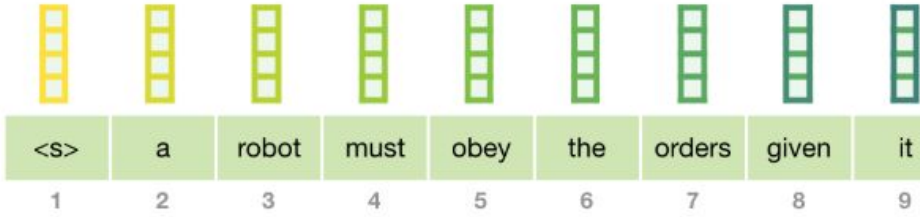
Keys = Context words to extract the feature from **= Word vector as context**

Value = Detected feature(s) value **= Word features (latent factors)**



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

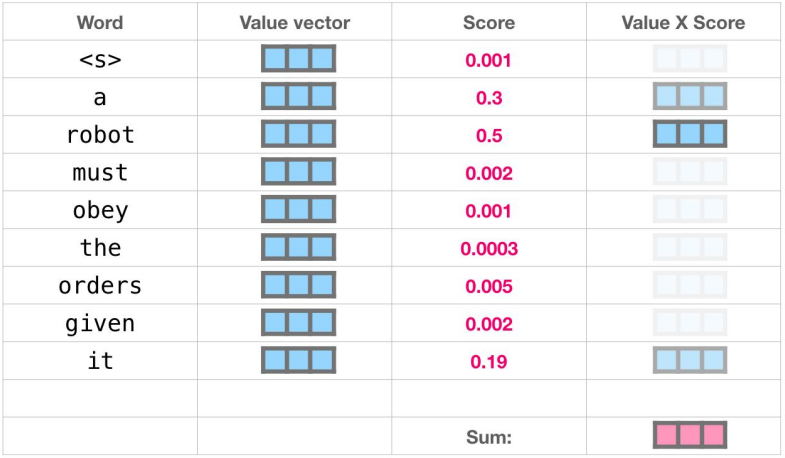
Self-Attention



Self-Attention Process

Self-attention is processed along the path of each token in the segment. The significant components are three vectors:

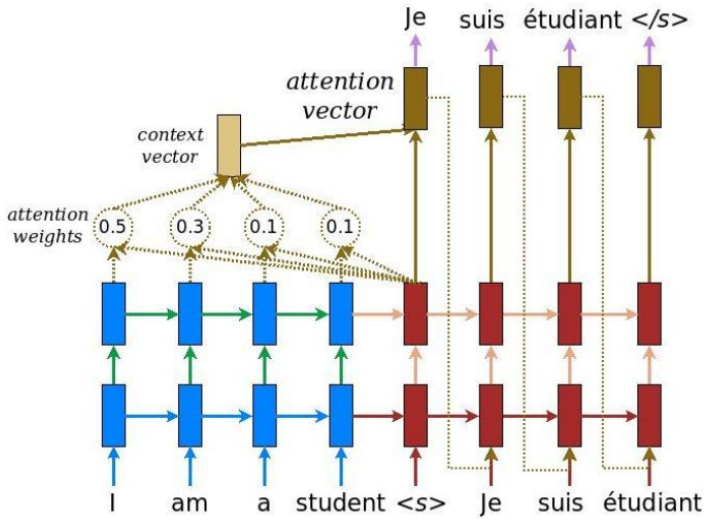
- Query: The query is a representation of the current word used to score against all the other words (using their keys). We only care about the query of the token we're currently processing.
- Key: Key vectors are like labels for all the words in the segment. They're what we match against in our search for relevant words.
- Value: Value vectors are actual word representations, once we've scored how relevant each word is, these are the values we add up to represent the current word.



http://jalammar.github.io/images/gpt2/gpt2-value-vector-sum.png

Self-Attention

Attention in seq2seq is over **hidden states** of **Encoder and Decoder**



$$\alpha_{ts} = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)} \quad \text{[Attention weights]} \quad (1)$$

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s \quad \text{[Context vector]} \quad (2)$$

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_c[\boldsymbol{c}_t; \boldsymbol{h}_t]) \quad \text{[Attention vector]} \quad (3)$$

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1} \boldsymbol{h}_t + \boldsymbol{W_2} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases} \quad (4)$$

Self-Attention

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

Self-Attention is applied over **Encoder Embeddings:**

Q = current word

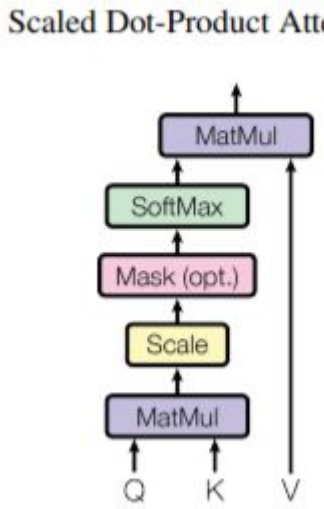
K = All input words

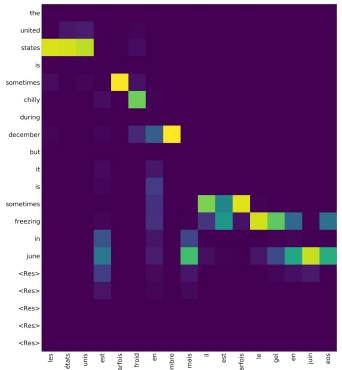
V = All input words (K=V)

QK^T = Outer product → Attention weights

Softmax → Normalization

xV → Scaled dot product → Weighted sum → Highest attention words are included → Extreme => Only current word vector is transfered





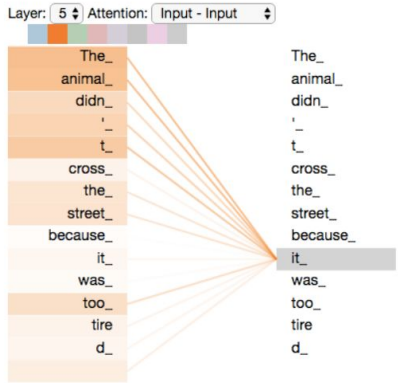
Self-Attention

Example

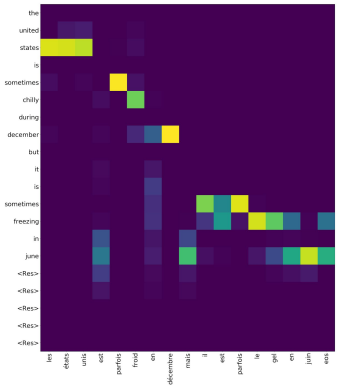
”`The animal didn't cross the street because it was too tired`”

What does “it” in this sentence refer to? Is it referring to the street or to the animal? It’s a simple question to a human, but not as simple to an algorithm.

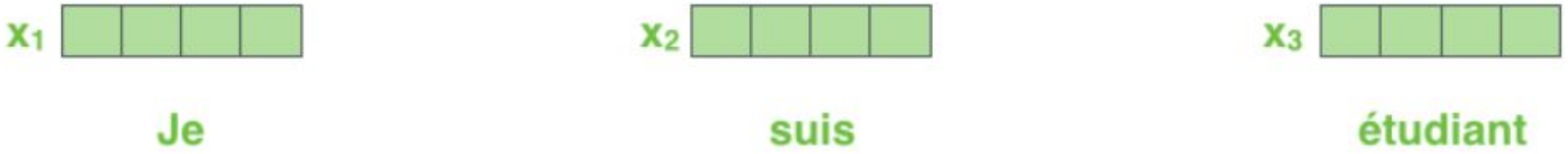
When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.



As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

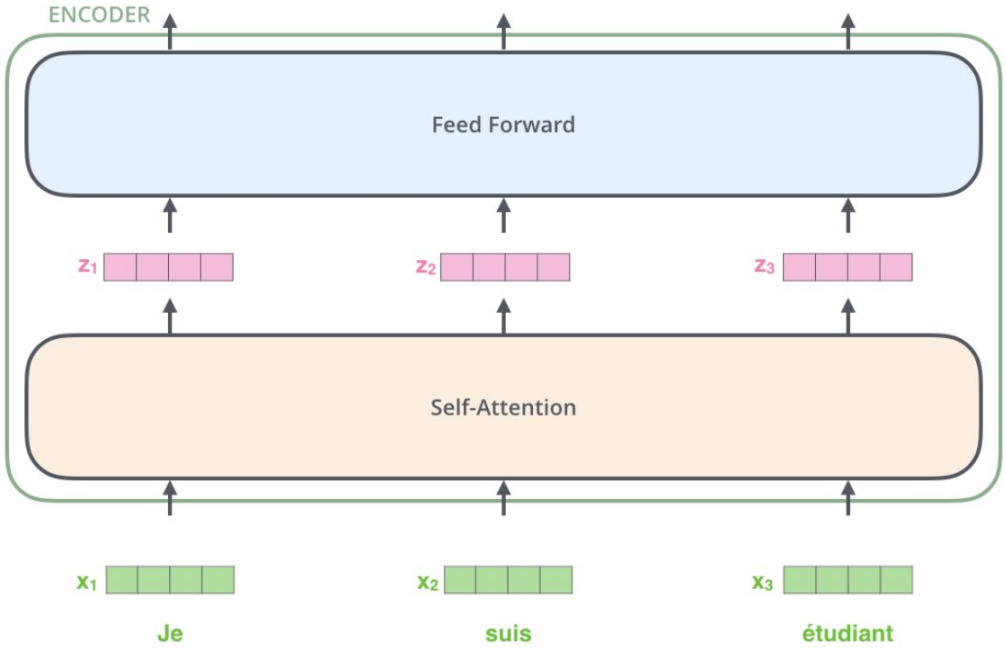


Self-Attention

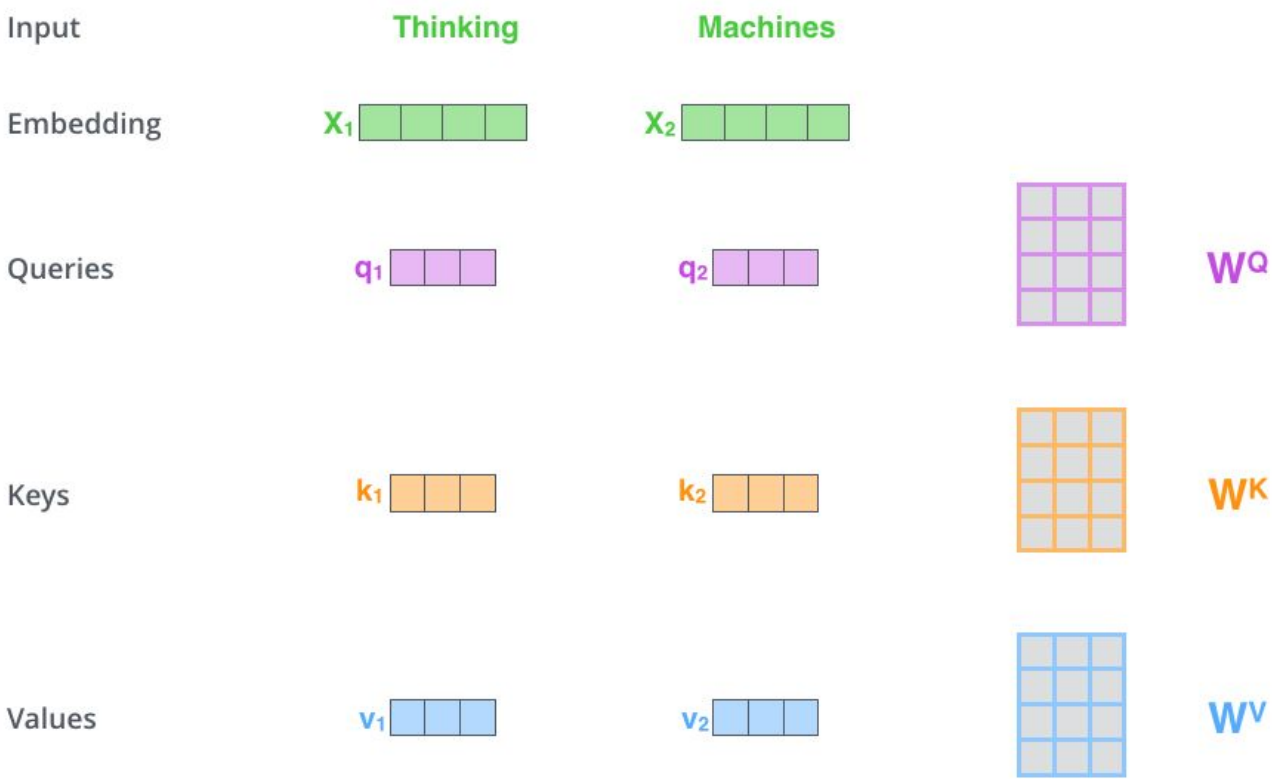


Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

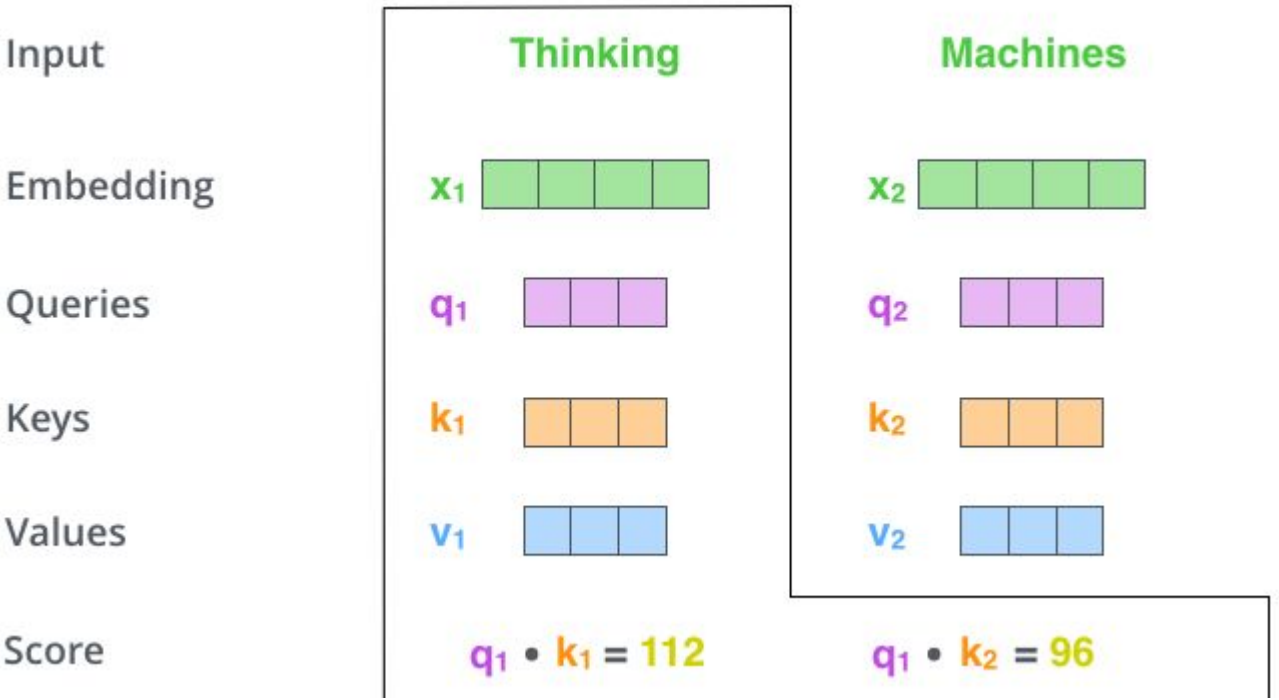
Self-Attention



Self-Attention



Self-Attention



Self-Attention

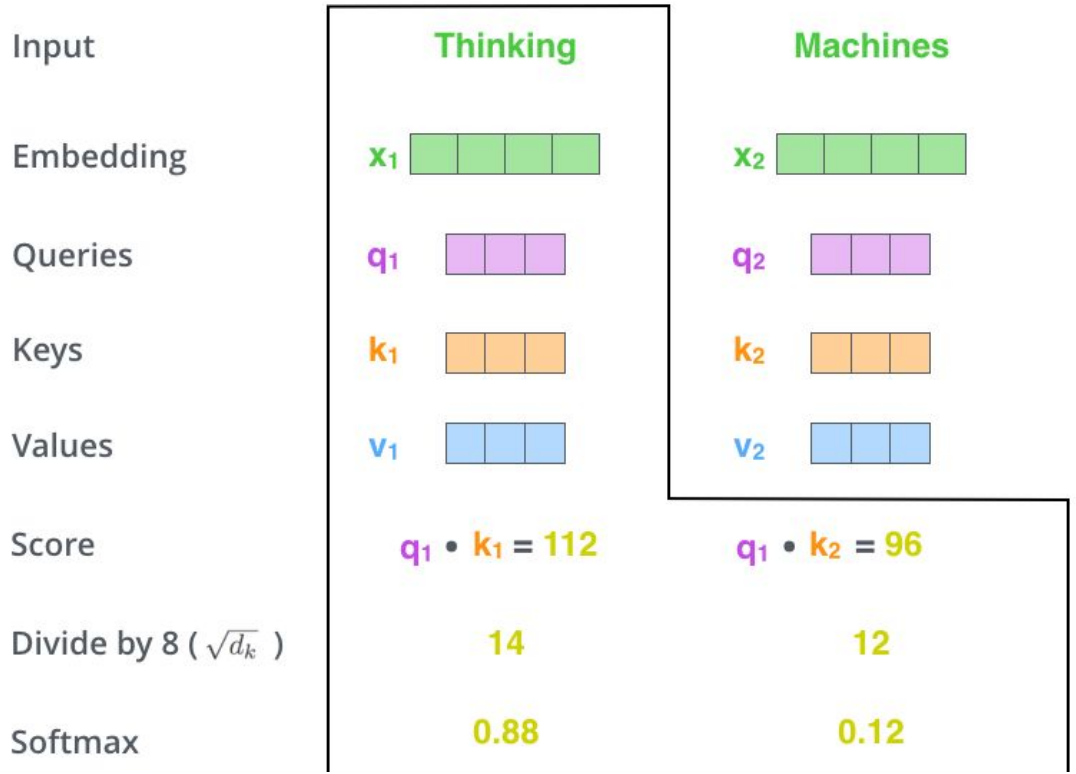
Why divide by dk?

<u>This leads to having more stable gradients.</u>

The Dot could grow large. We will normalize in softmax anyway. But it could overflow before applying the softmax

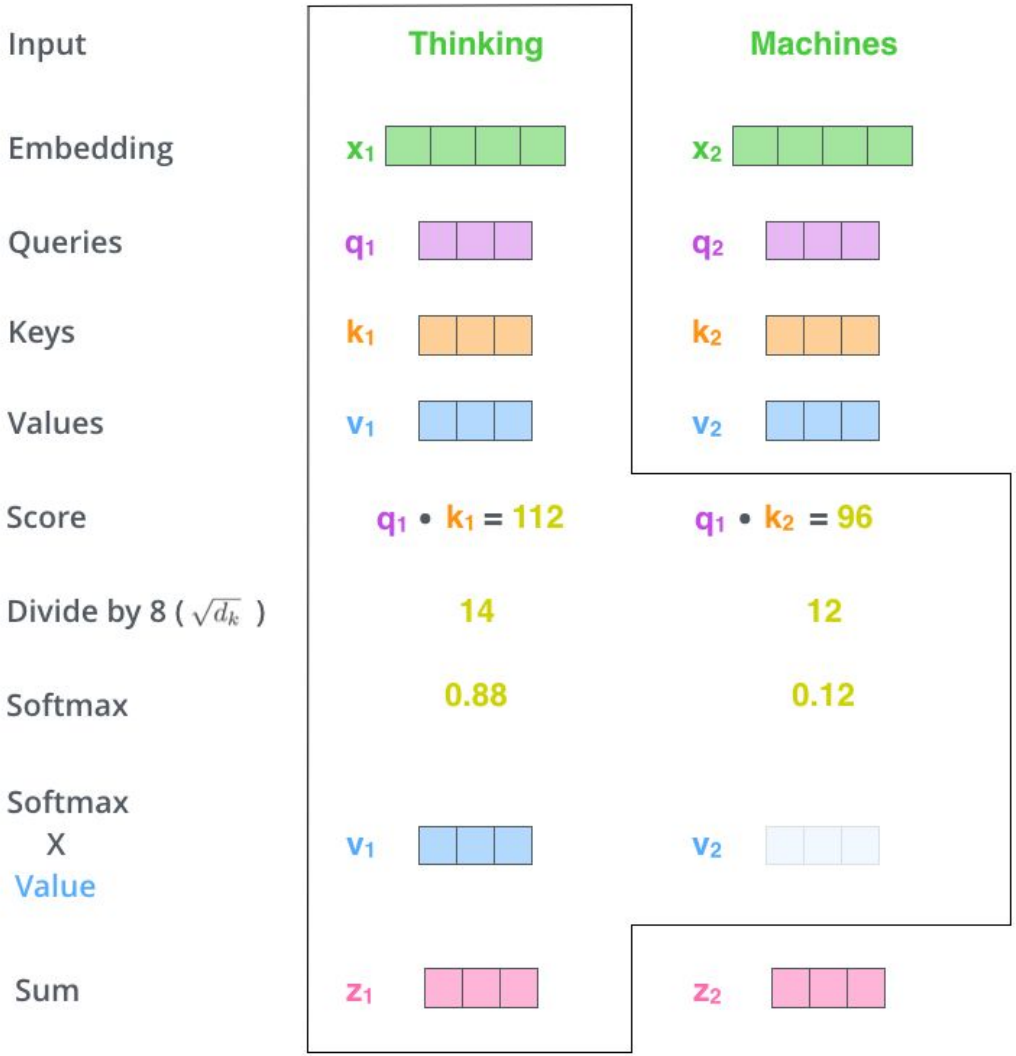
Also, large softmax, means extreme peaks→ smaller gradients

dk = Embedding dimension of keys (64 in the paper, sqrt=8)



Self-Attention

Notice how we get the weighted sum, scaled by the attention weights

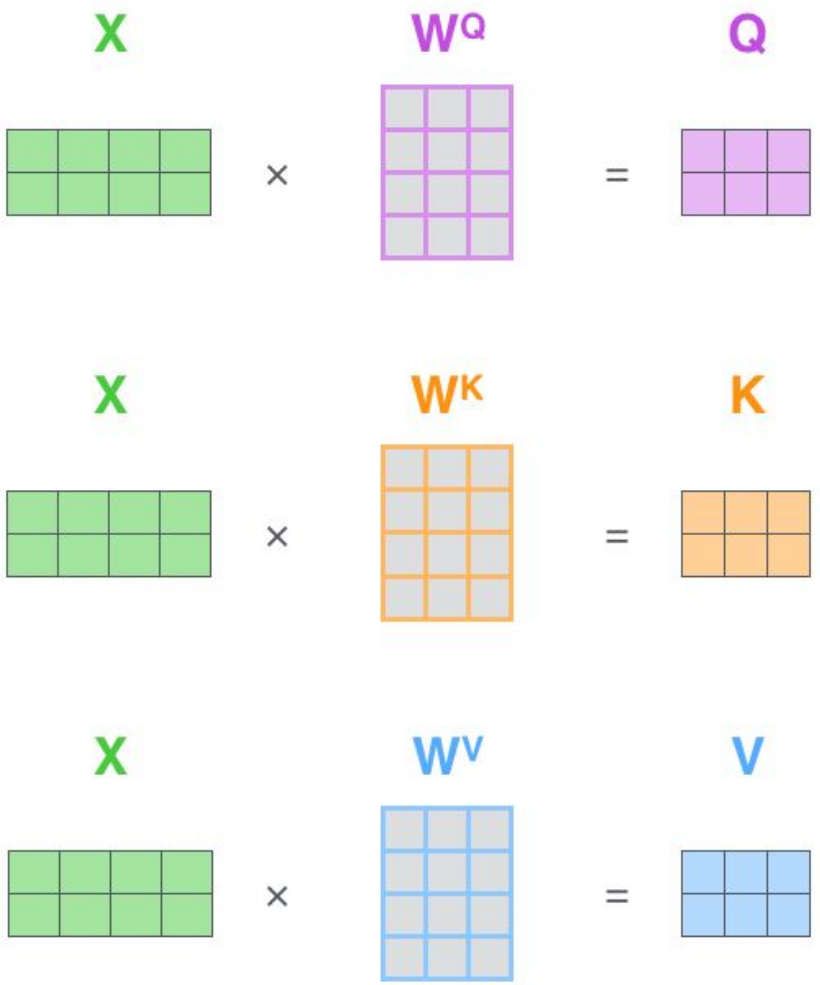


Self-Attention

Matrix equations

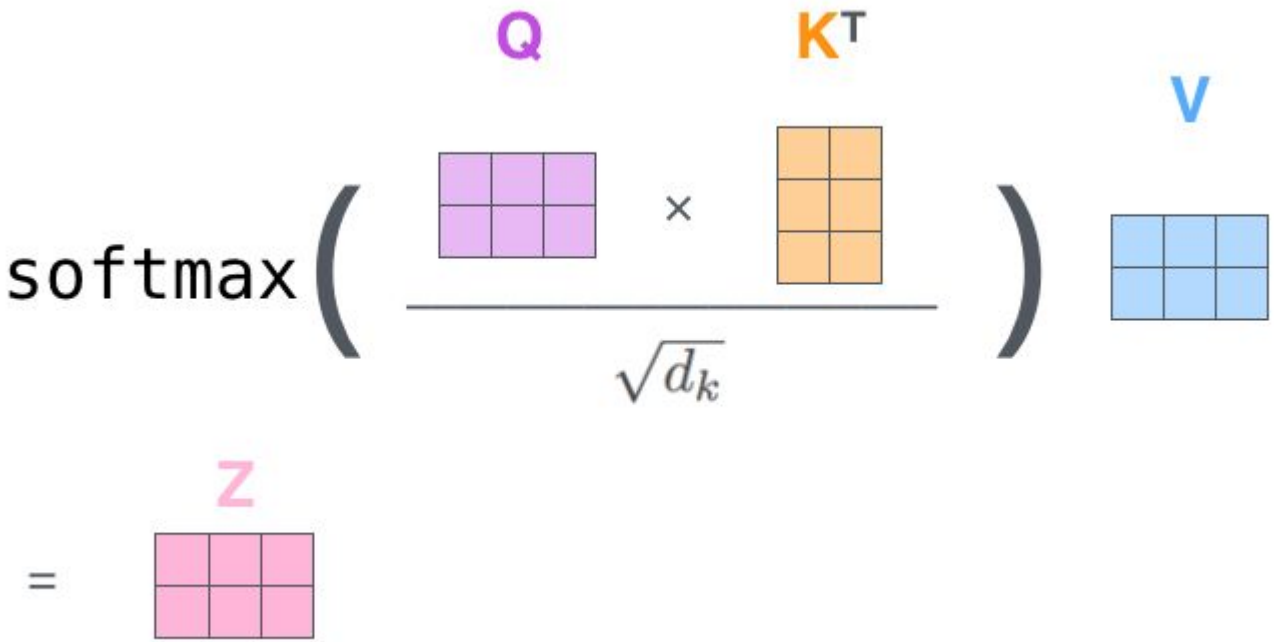
Same as before, but in matrix form:

X is now 2 words (or n_samples or batch)

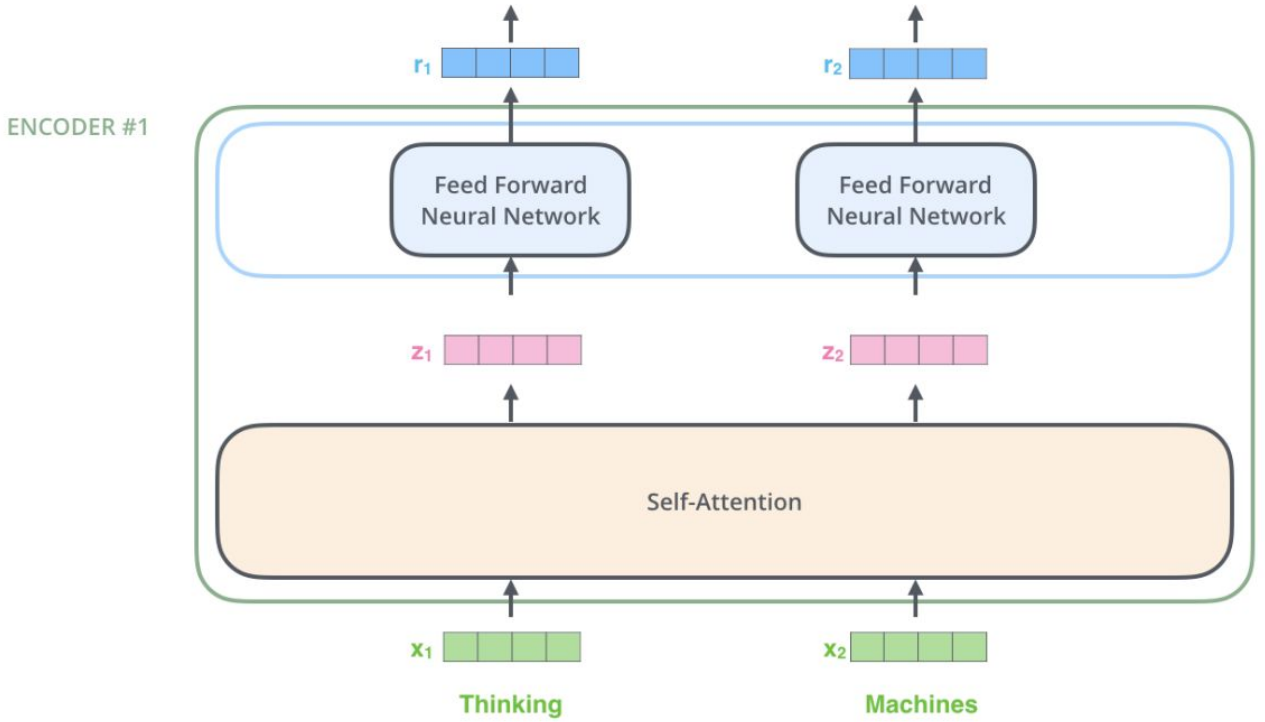


Self-Attention

Matrix equations



Self-Attention



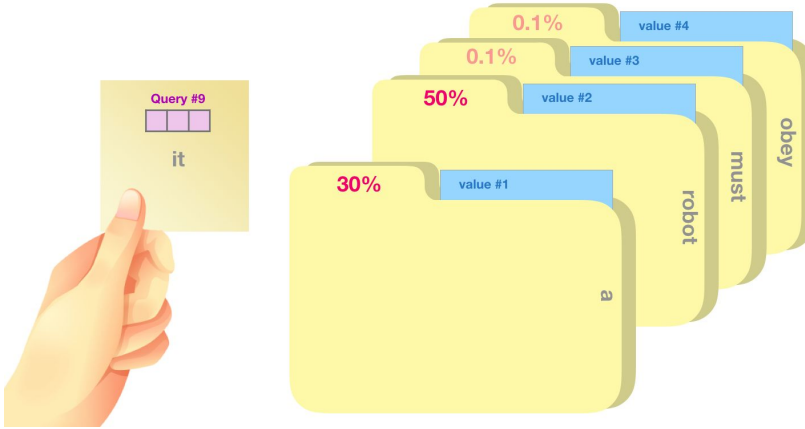
The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

Multi-head attention

Query = A feature detector (is_verb, is_name, ….) **= Word as feature detector**

Keys = Context words to extract the feature from **= Word vector as context**

Value = Detected feature(s) value **= Word features (latent factors)**



But we don't want only one feature | Context!

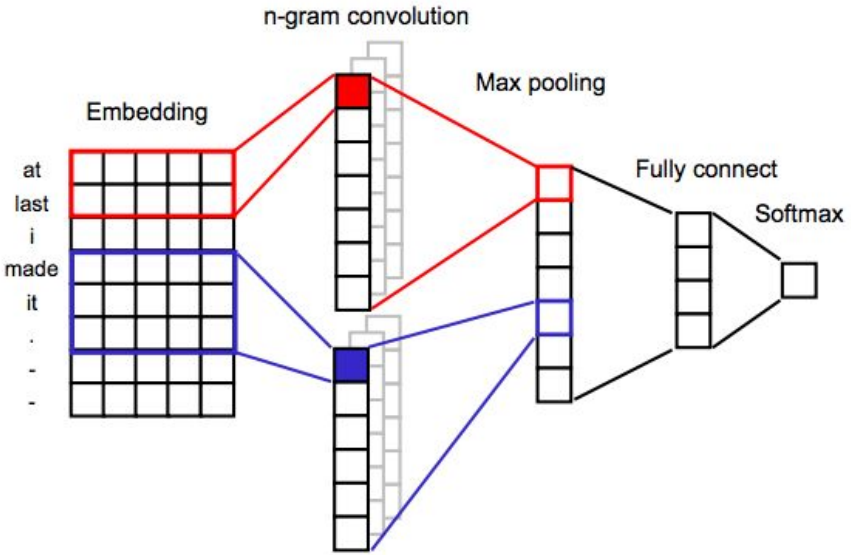
(is_verb, is_name, ….)

→ Make multiple Queries!

Multi-headed CNN

Another popular approach with CNNs is to have a multi-headed model, where each head of the model reads the input time steps using a different sized kernel.

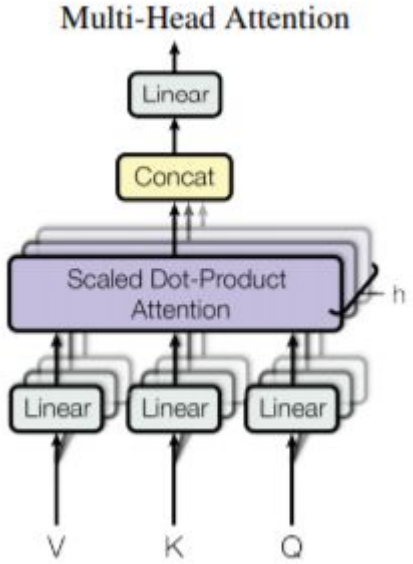
For example, a three-headed model may have three different kernel sizes of 3, 5, 11, allowing the model to read and interpret the sequence data at three different resolutions. The interpretations from all three heads are then concatenated within the model and interpreted by a fully-connected layer before a prediction is made.



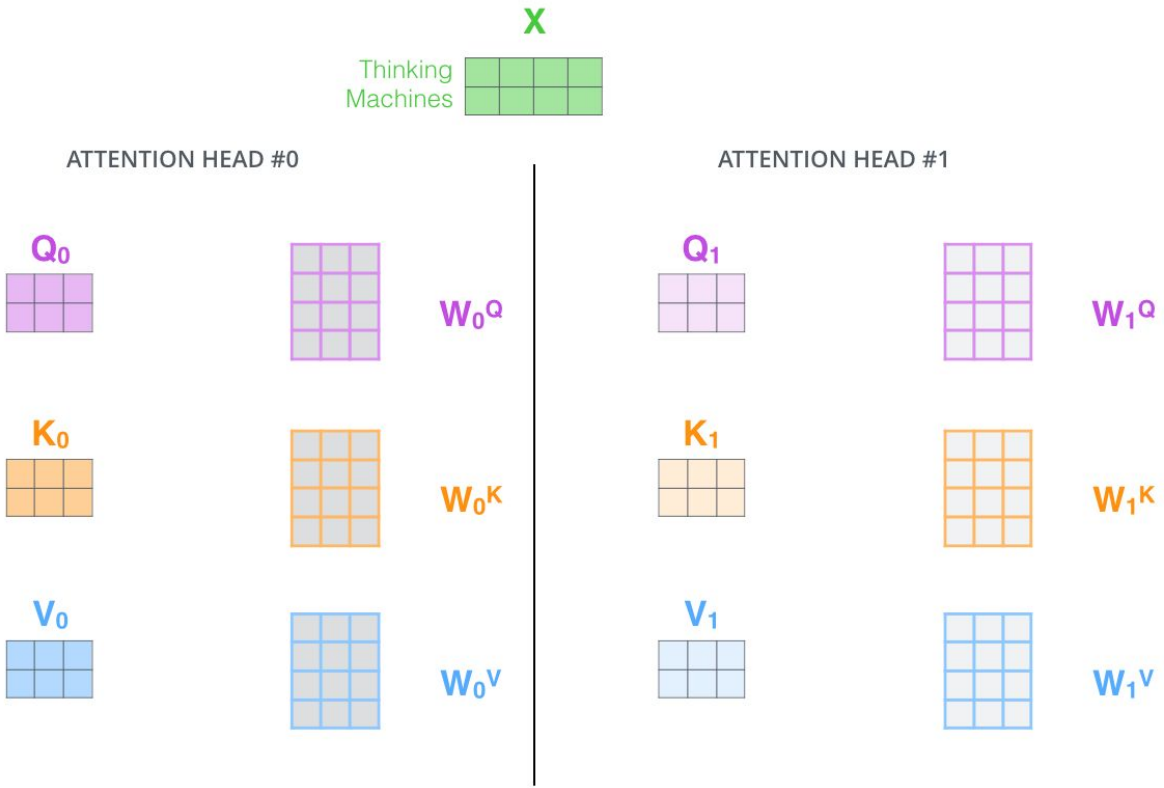
Multi-head attention

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

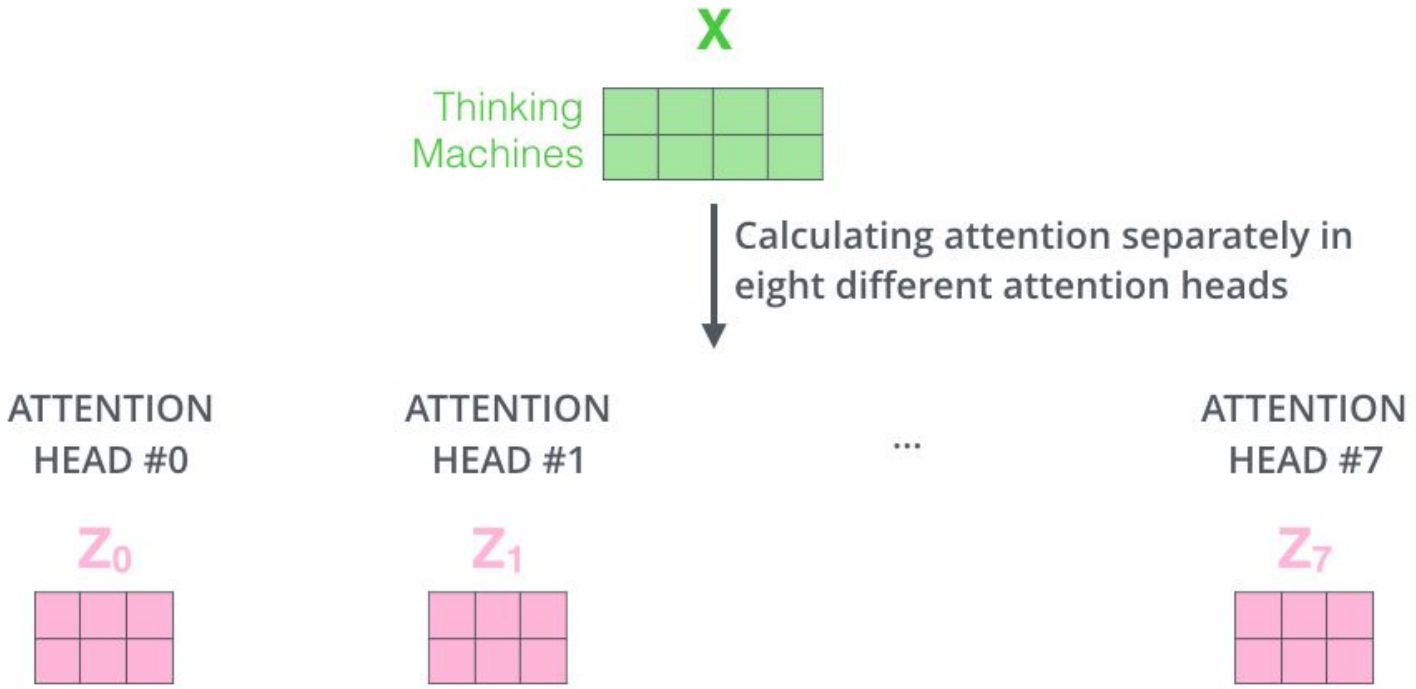
1. It expands the model's ability to **focus on different positions**. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the the actual word itself. It would be useful if we're translating a sentence like "The animal didn't cross the street because it was too tired", we would want to know which word "it" refers to.
2. It gives the attention layer **multiple "representation subspaces"**. As we'll see next, with multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.



Multi-head attention



Multi-head attention

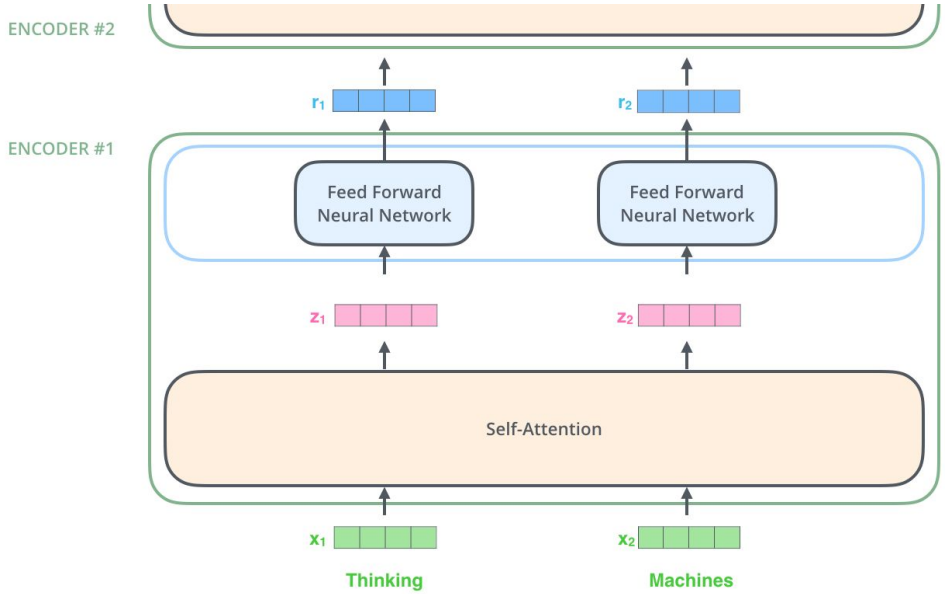


Mutli-head Attention + Feed Fwd

The next stage is Feed Fwd (FFN), which expects one “z”

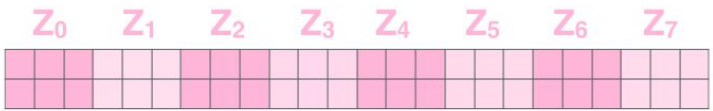
How to handle this, while we now have 8 z’s from 8 heads?

→ **Concatenate**



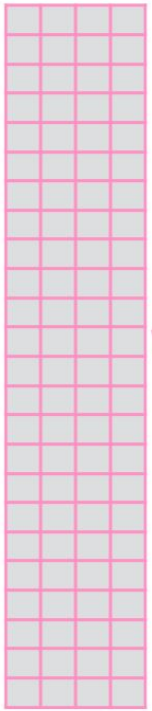
Multi-head attention

1) Concatenate all the attention heads



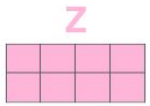
2) Multiply with a weight matrix W^O that was trained jointly with the model

X

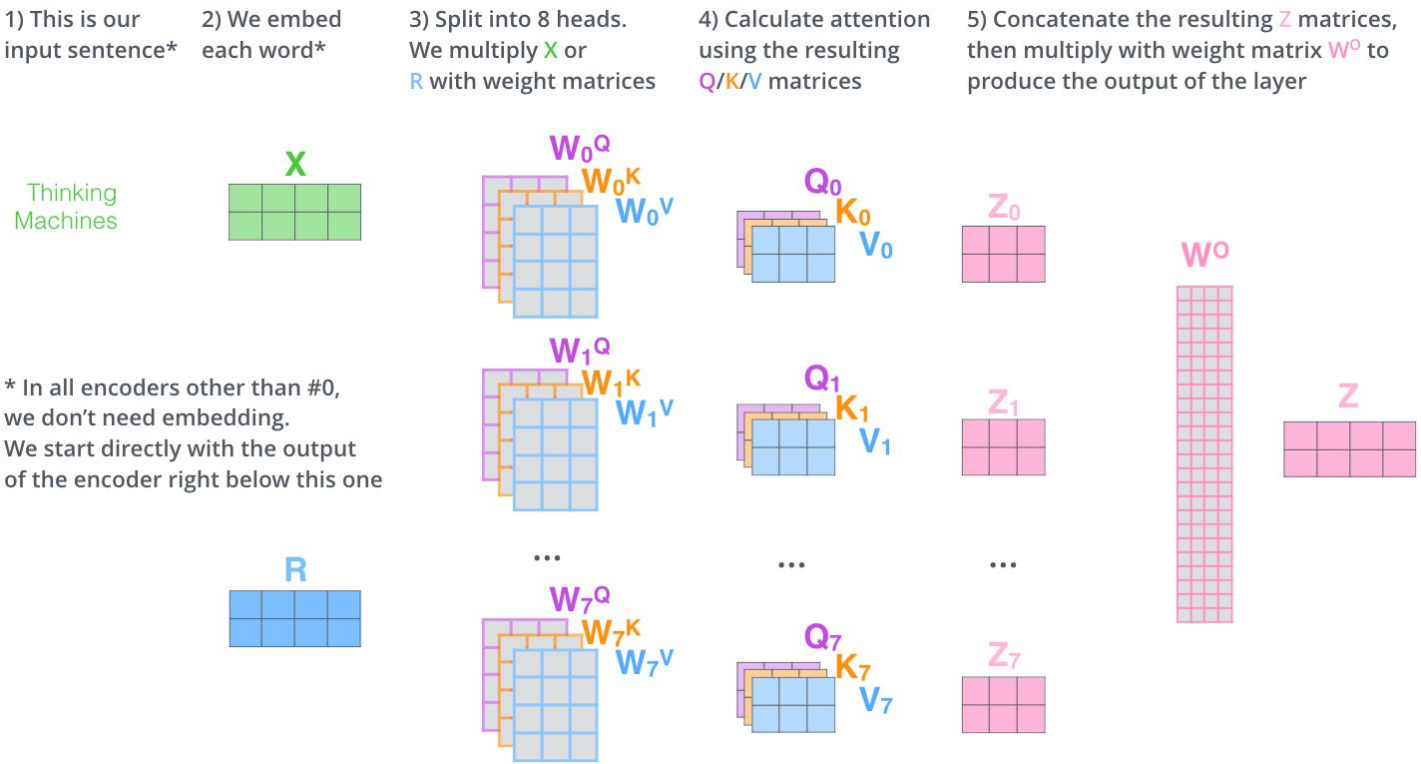


3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

=



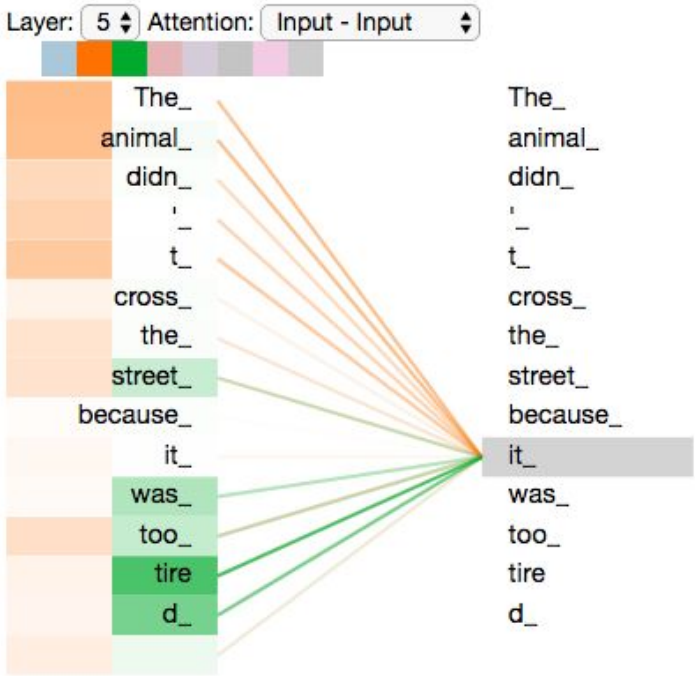
Multi-head attention - Putting all together



Multi-head Attention

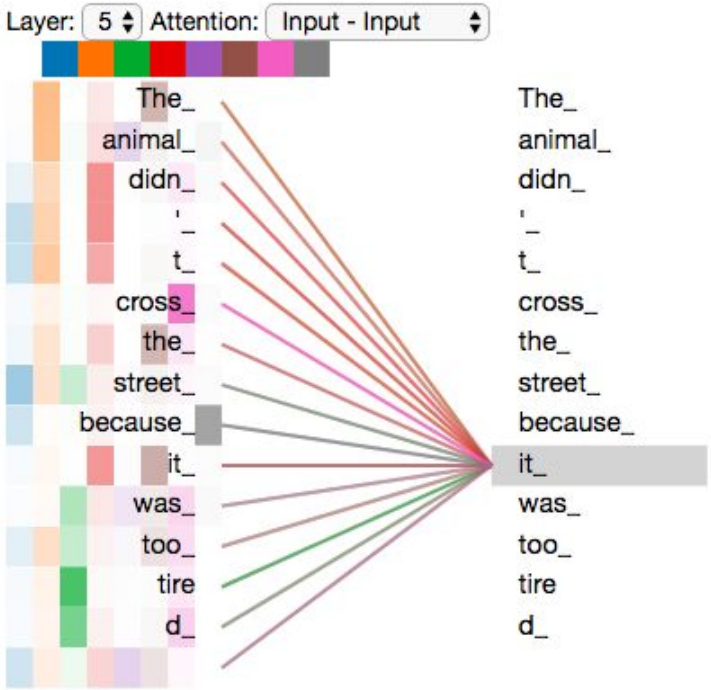
Let’s revisit the example now, with all heads attention. Let’s start only with 2 heads (orange and green):

As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".



Multi-head Attention

Let’s revisit the example now, with all heads attention. All 8 are hard to interpret.



Positional Encoding

Transformers in their current form lack essential feature→ word position in the sentence → **sequence information**

Position encoding → Encode the position of the word in the sentence as **explicit feature** → sequence preserved (also in convs2s)

Position is just another **categorical feature**, just like word index (or any cat feature in tabular data).

Categorical features are encoded with **Embedding** in structured DL.

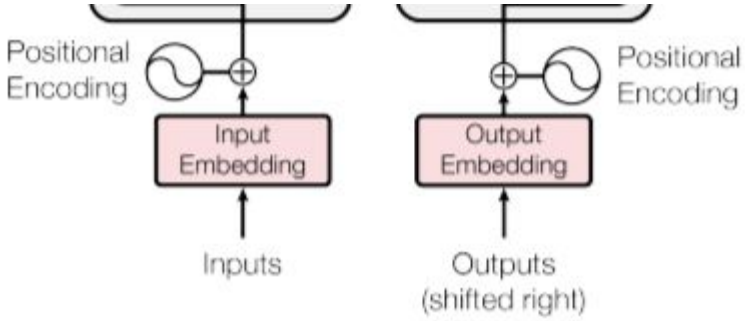
How to merge with word embeddings?

Positional Encoding

How to merge with word embeddings?

As with any fusion model, the merge layer could be: concat, sum, average,...etc

In the paper they choose **sum**



But this Embedding is not necessarily learnable!

Positional Encoding

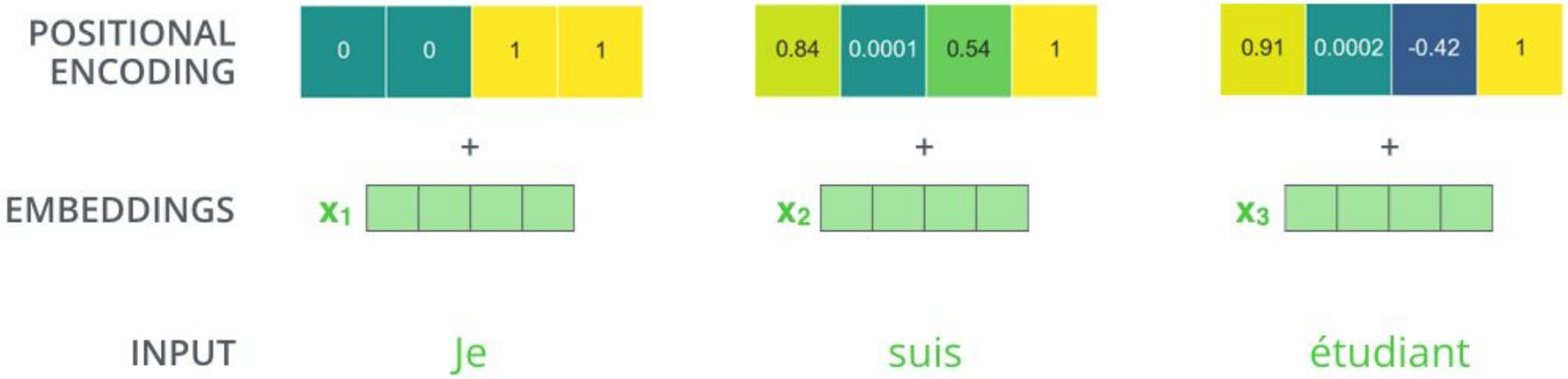
But this Embedding is not necessarily learnable!

We just need a vector representation with the following characteristics:

- Size comparable to word embeddings→ It will be added to it
- Similarity between neighboring positions is encoded in the vector space

Any continuous function pattern will do the job.

Positional Encoding - Example with dimension=4



Positional Encoding

But this Embedding is not necessarily learnable!

-

Any continuous function pattern will do the job.

We want every neighbor position (say the one to the right of the current), to be a unit larger/smaller from the current. The Embedding table length = max sequence length (we are encoding positions).

We don’t know apriori the max sequence length. If we choose a linear or continously increasing function, the number could overflow with long sequence.

Best solution is to choose **periodic function**

Positional Encoding

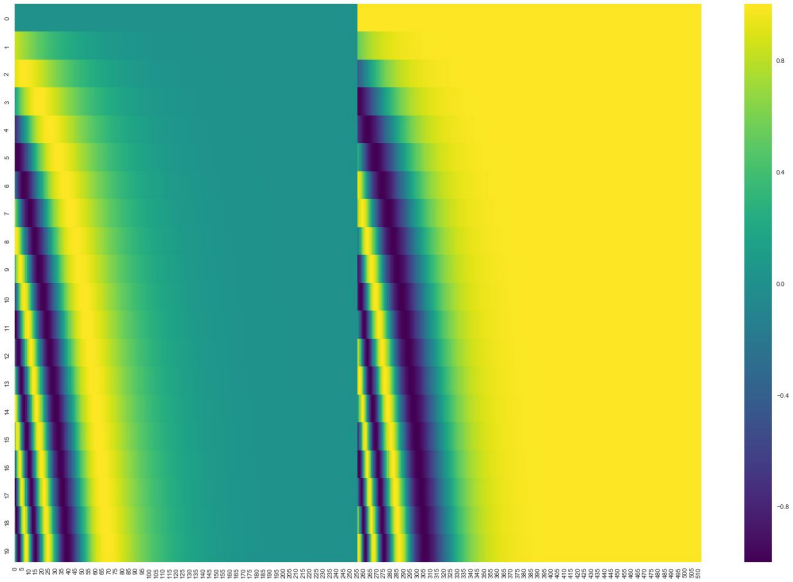
But this Embedding is not necessarily learnable!

We just need a vector representation with the following characteristics:

- Size comparable to word embeddings→ It will be added to it
- Similarity between neighboring positions is encoded in the vector space

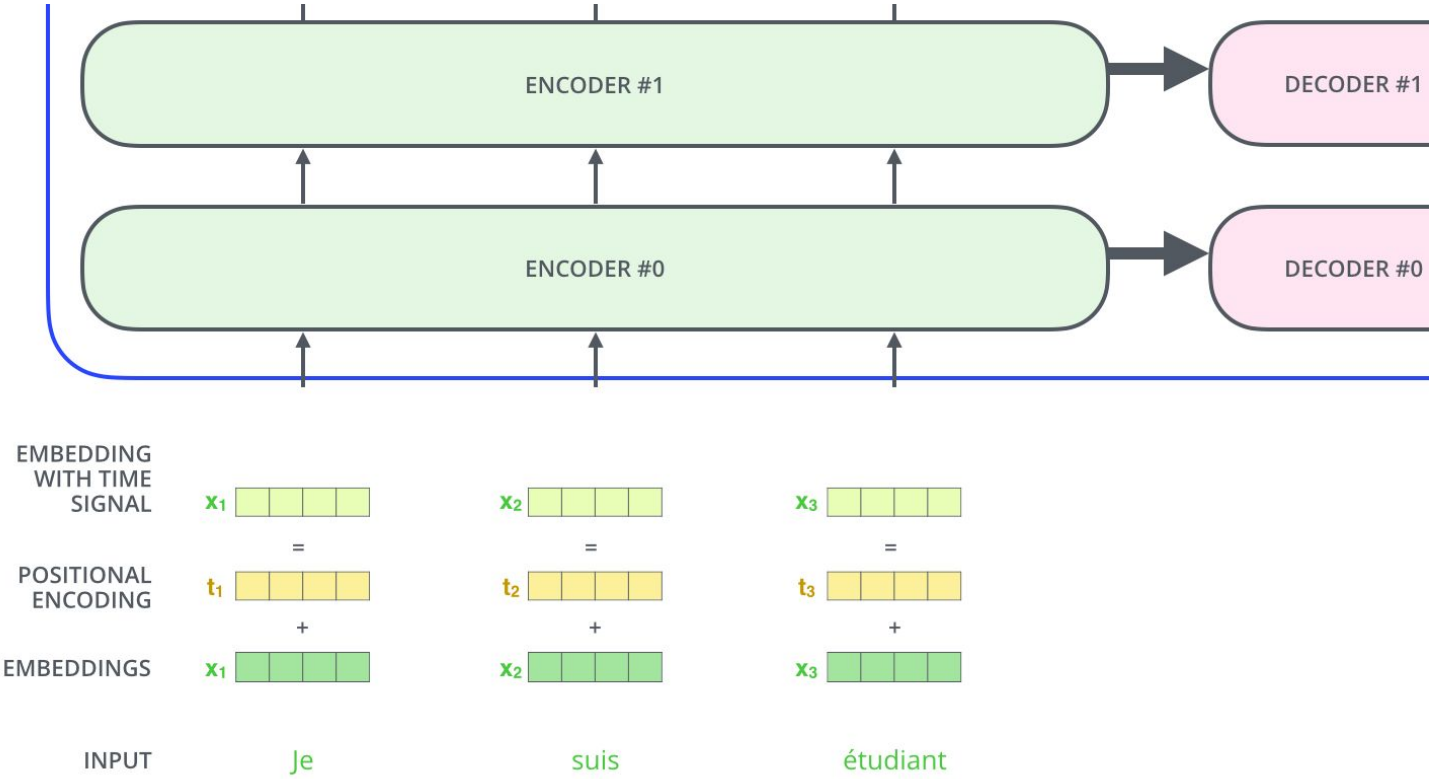
Any continuous function pattern will do the job.

In the paper they choose sinusoidal function



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

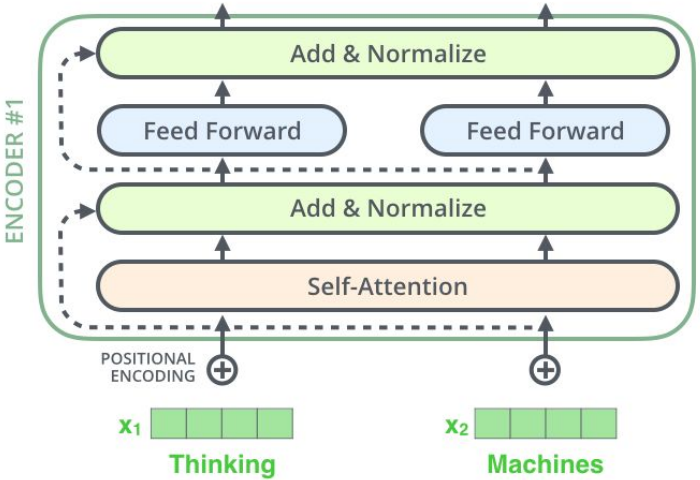
Positional Encoding - Putting all together



Extras

Residuals = Skip connections

For vanishing gradients (ResNet→ Deep Arch)



Extras

Residuals = Skip connections

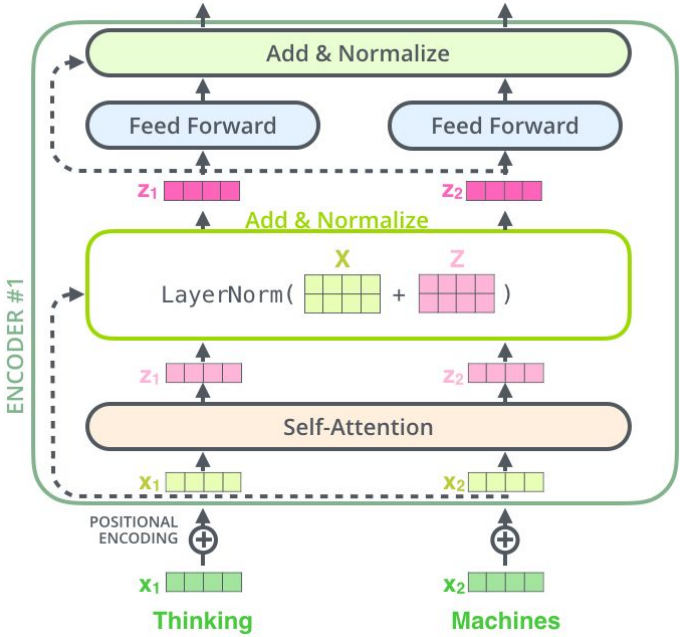
For vanishing gradients (ResNet→ Deep Arch)

+LayerNorm (https://arxiv.org/abs/1607.06450) → BatchNorm for RNN

One way to reduce the training time is to normalize the activities of the neurons.

batch normalization is dependent on the mini-batch size and it is not obvious how to apply it to recurrent neural networks → Batch is variable size in RNN

batch normalization into layer normalization by computing the mean and variance used for normalization from all of the summed inputs to the neurons in a layer on a **single training case**

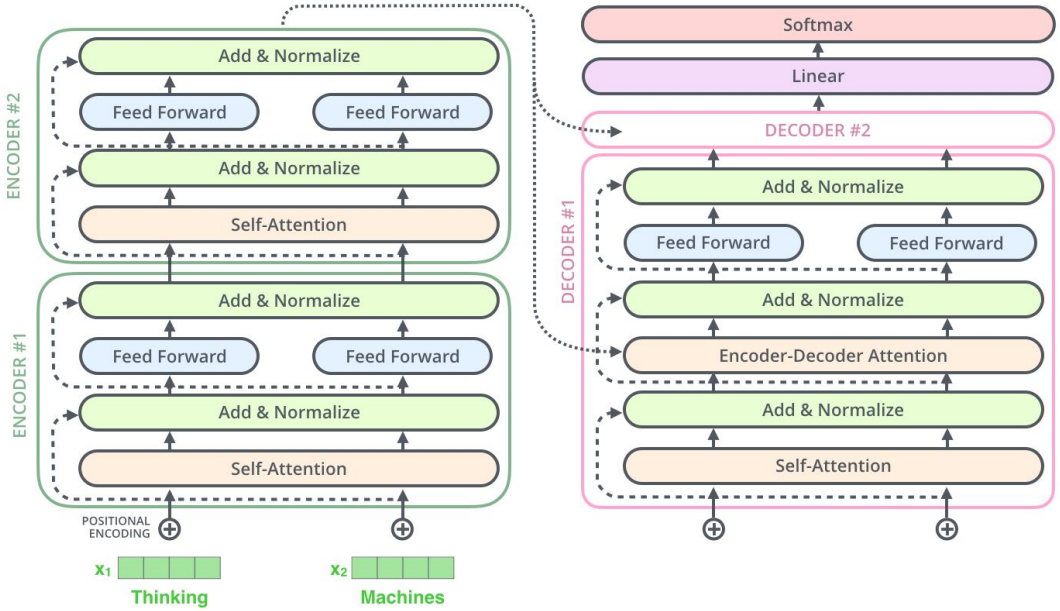


Decoder

Same as Encoder, but will have Self-Attention

But it cannot include the future decoder output like in the Encoder

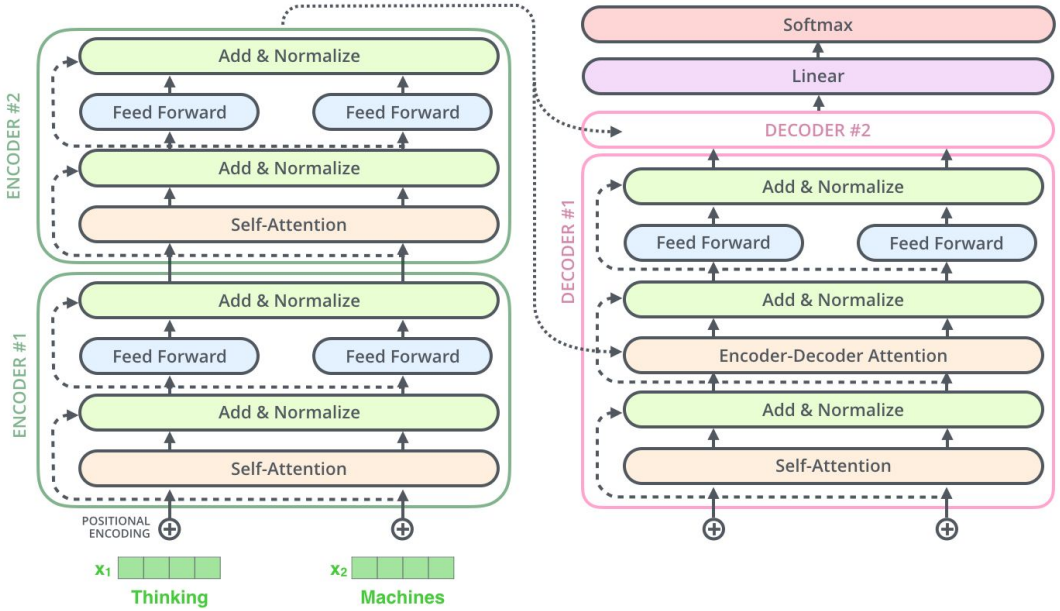
In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to `-inf`) before the softmax step in the self-attention calculation.



Decoder

Same as Encoder, but will have Self-Attention + Encoder-Decoder Attention: will now include the encoder layer outputs + the Decoder output so far:

The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.



Decoder - Look-ahead Masking layer

But it cannot include the future decoder output like in the Encoder

Mask all the pad tokens in the batch of sequence. It ensures that the model does not treat padding as the input.

The look-ahead mask is used to mask the future tokens in a sequence. In other words, the mask indicates which entries should not be used.

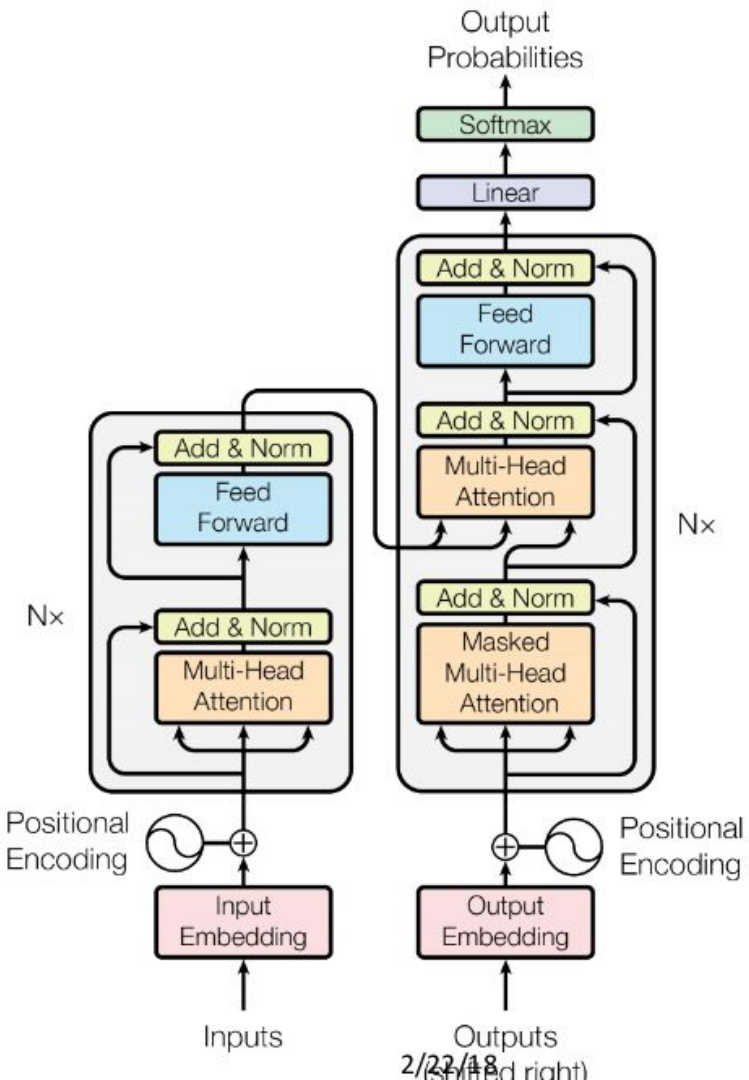
This means that to predict the third word, only the first and second word will be used. Similarly to predict the fourth word, only the first, second and the third word will be used and so on.

Transformer Decoder

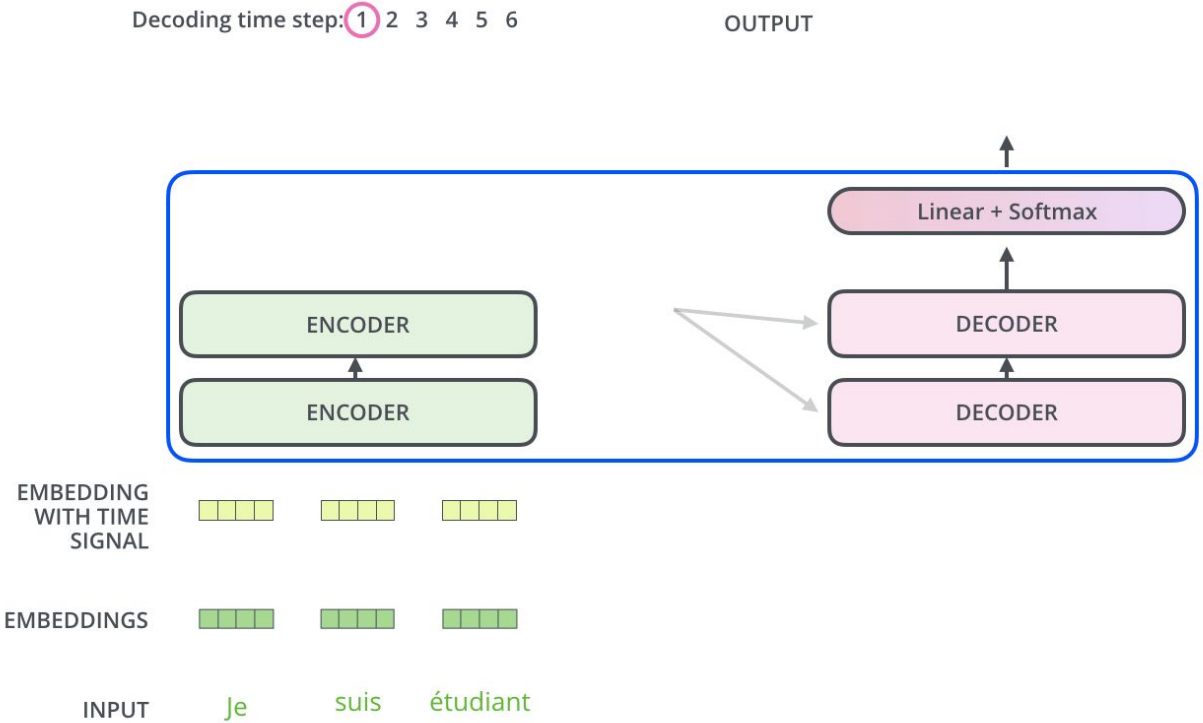
- 2 sublayer changes in decoder
- Masked decoder self-attention on previously generated outputs:

- Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of encoder

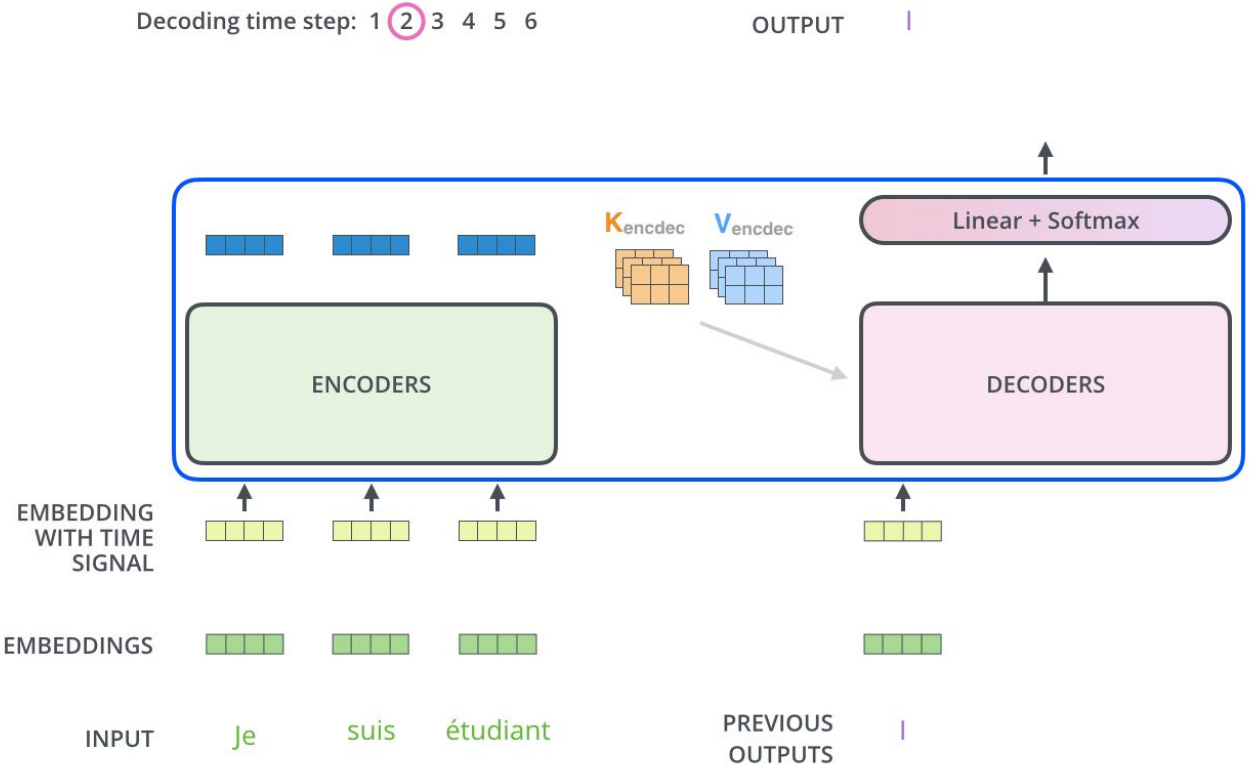
- Blocks repeated 6 times also



Encoding process



Decoding process



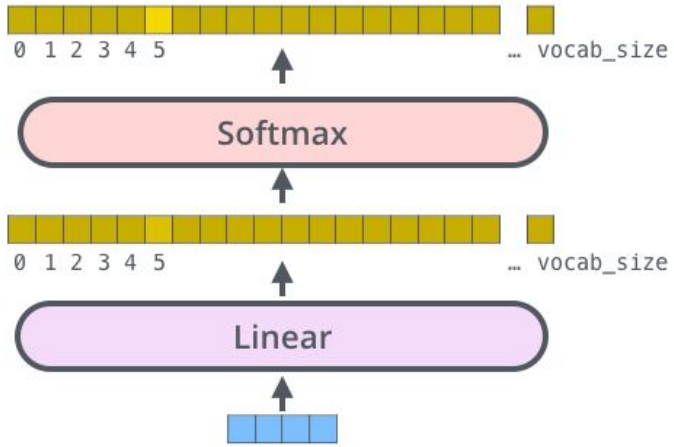
Output softmax

Which word in our vocabulary is associated with this index?

am

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

5



Scalability

[Attention is all you need paper]

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

d (~100) >> n (~10)

Parallelization

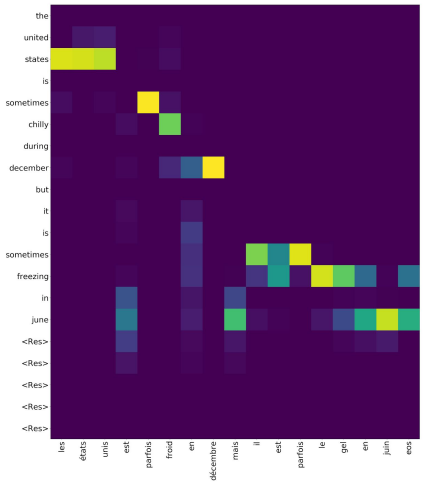
Scalability

[Attention is all you need paper]

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

No sequential operation
ALL in parallel



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

alignment/attention weight matrix→ nxn

Weighted sum over V's → d

Scalability

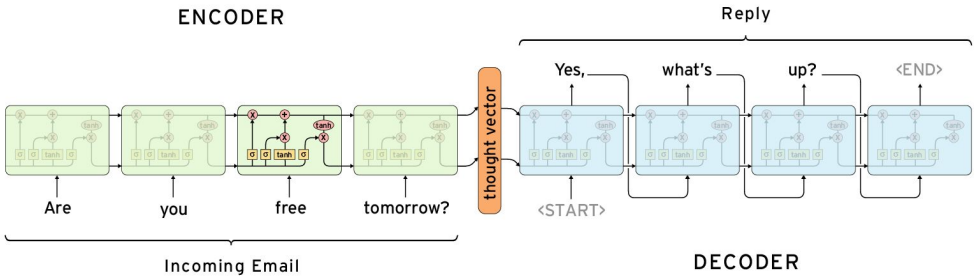
[Attention is all you need paper]

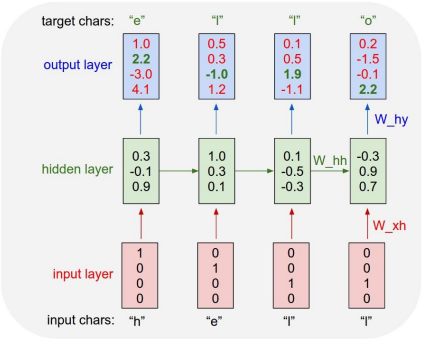
Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Each W is dxd

Although W's are shared overtime, however, the operation is repeated n times → **Sequential**





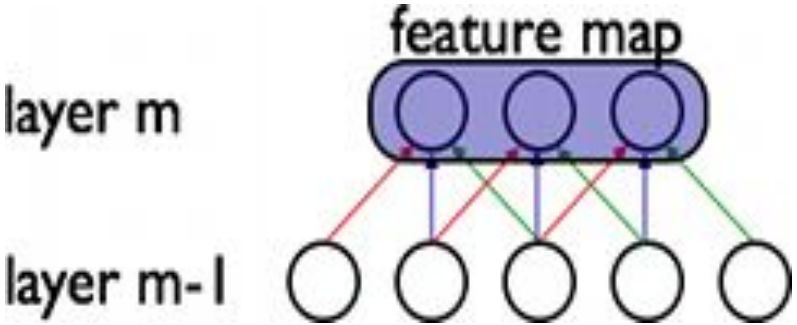
Scalability

[Attention is all you need paper]

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

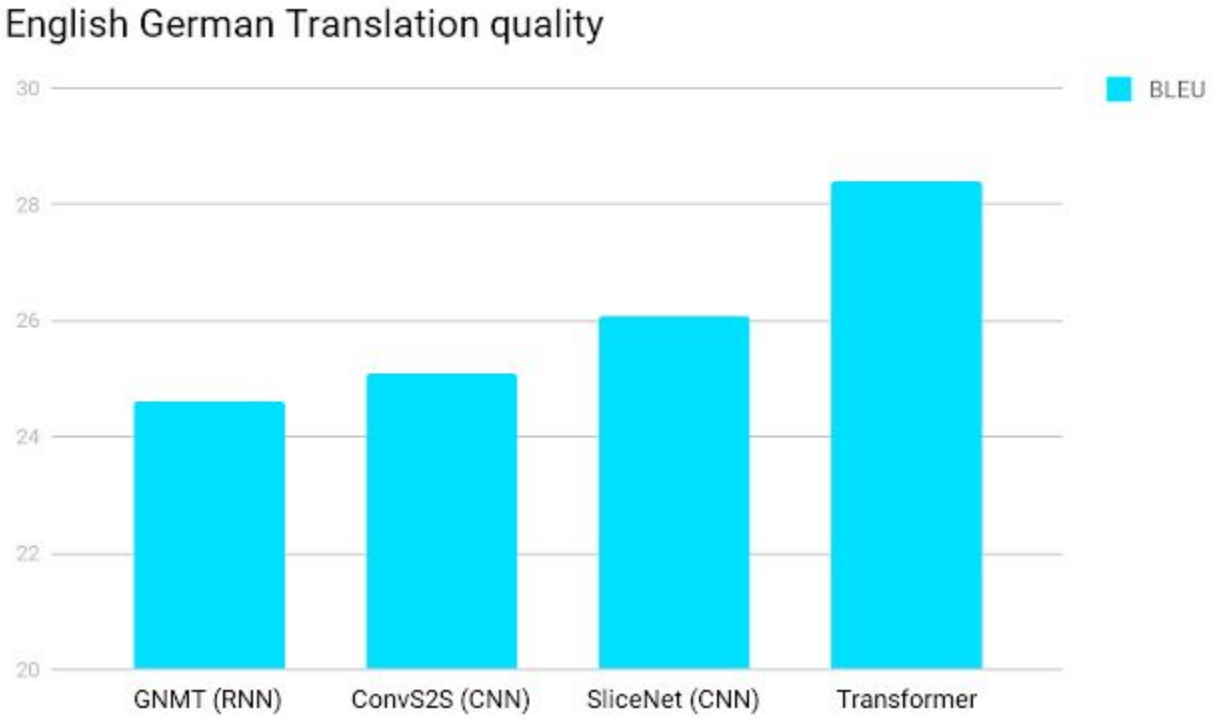
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Still O(1) since all feature maps are done in parallel

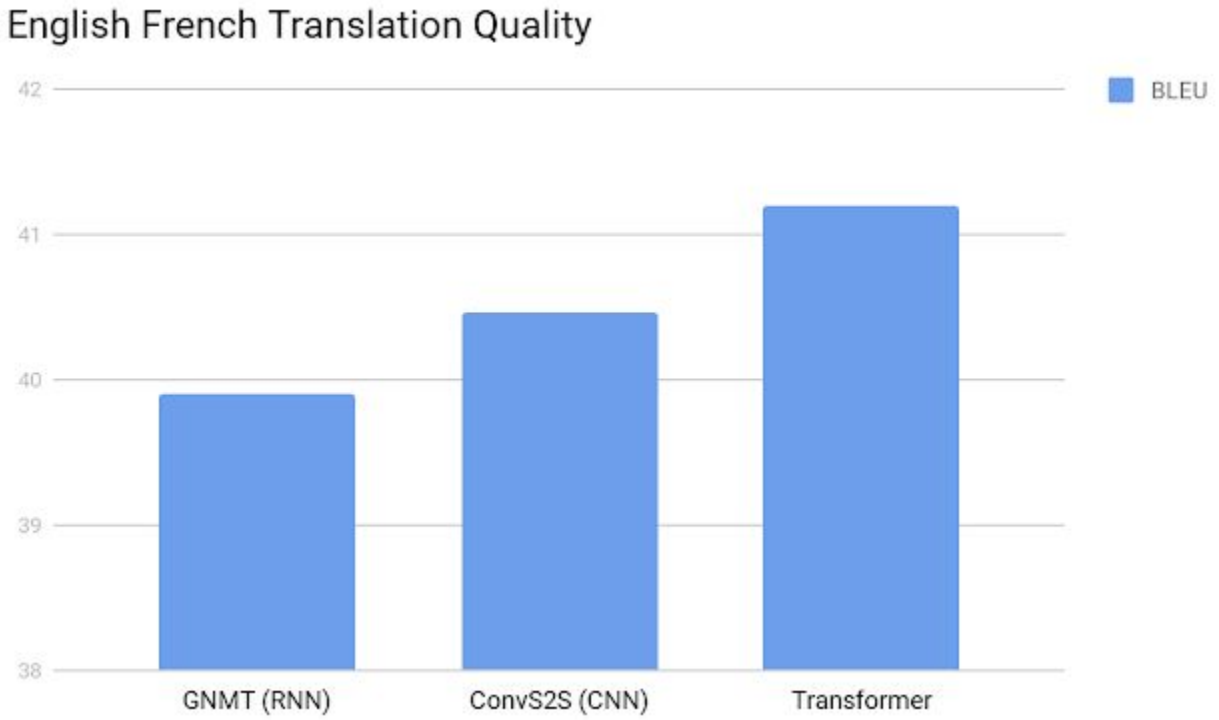


For each kernel→ k operations
Repeat ~ n times
We have multiple feature maps per input → dxd

Performance



Performance



Let’s code

https://colab.research.google.com/drive/1vG_FKAI9FcrPJf4wis070rTeuclqdHrl?usp=sharing

References

- Sequence to Sequence Learning with Neural Networks
- Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
- Neural Machine Translation by Jointly Learning to Align and Translate
- Effective Approaches to Attention-based Neural Machine Translation
- https://github.com/devm2024/nmt_keras/blob/master/base.ipynb
- https://keras.io/examples/lstm_seq2seq/
- https://machinelearningmastery.com/return-sequences-and-return-states-for-lstms-in-keras/
- https://arxiv.org/abs/1706.03762
- http://jalammar.github.io/illustrated-transformer/

- https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html