Lecture 5 – NLP. Part 2. Model zoo

Plan

- Language modeling
  - RNN
  - LSTM
  - GRU
- Mashing Translation
  - RNN
  - RNN with Attention
  - Transformer

Language modeling

Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*

Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma ??. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

be a morphism of algebraic spaces over S and Y.

Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

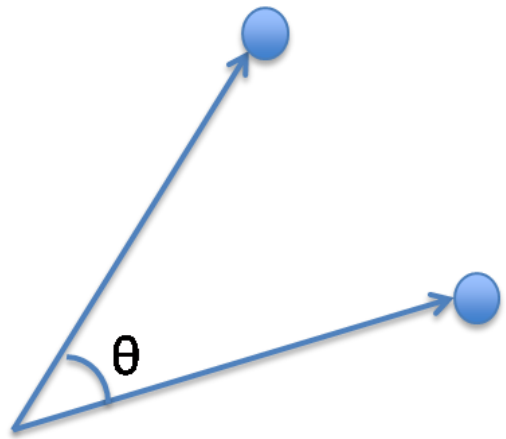
Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □

```c
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECONDS << 12;
  return segtable;
}
```

https://karpathy.github.io/2015/05/21/rnn-effectiveness/

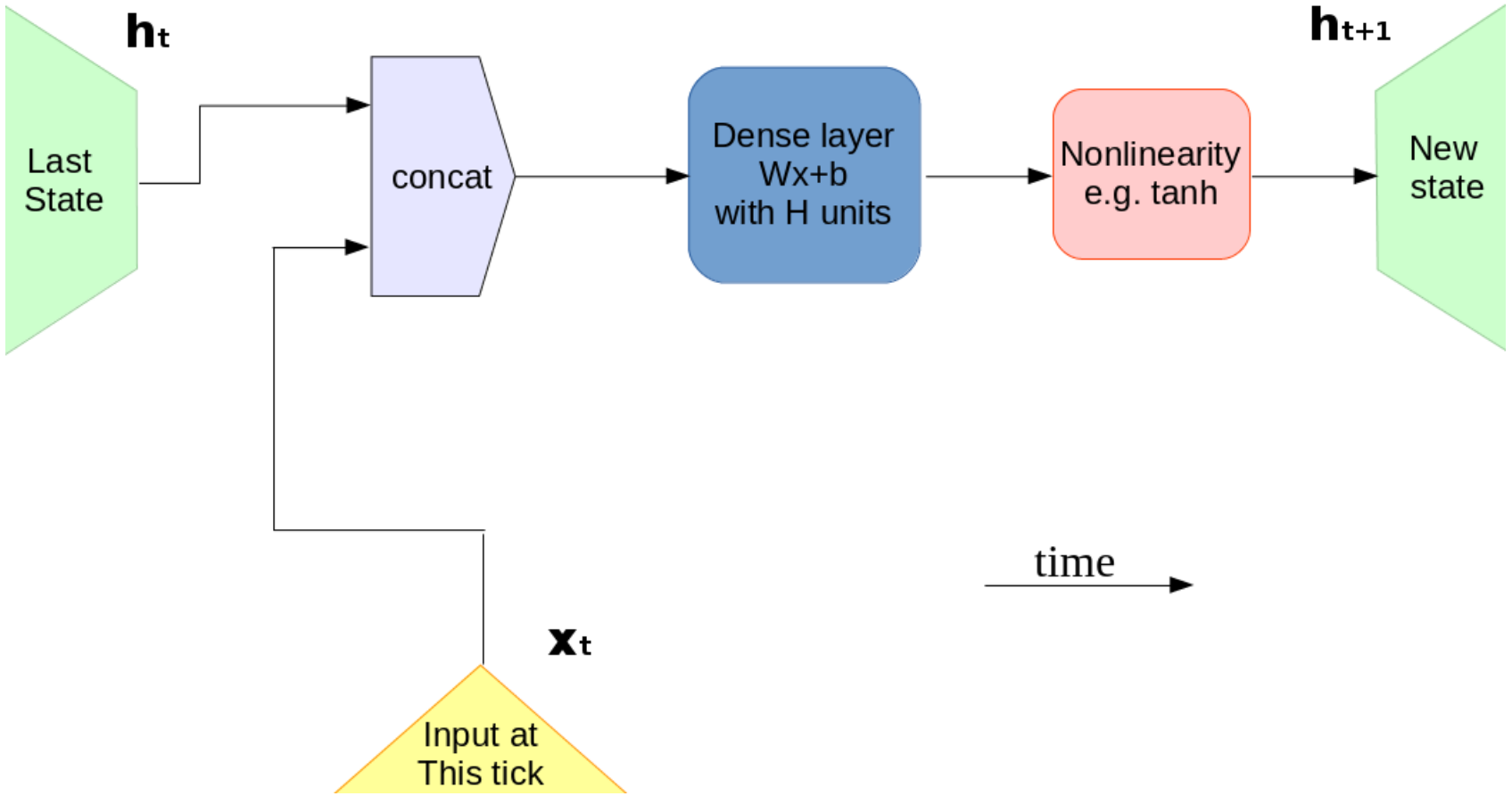
Cosine simularity

$$sim(A,B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$



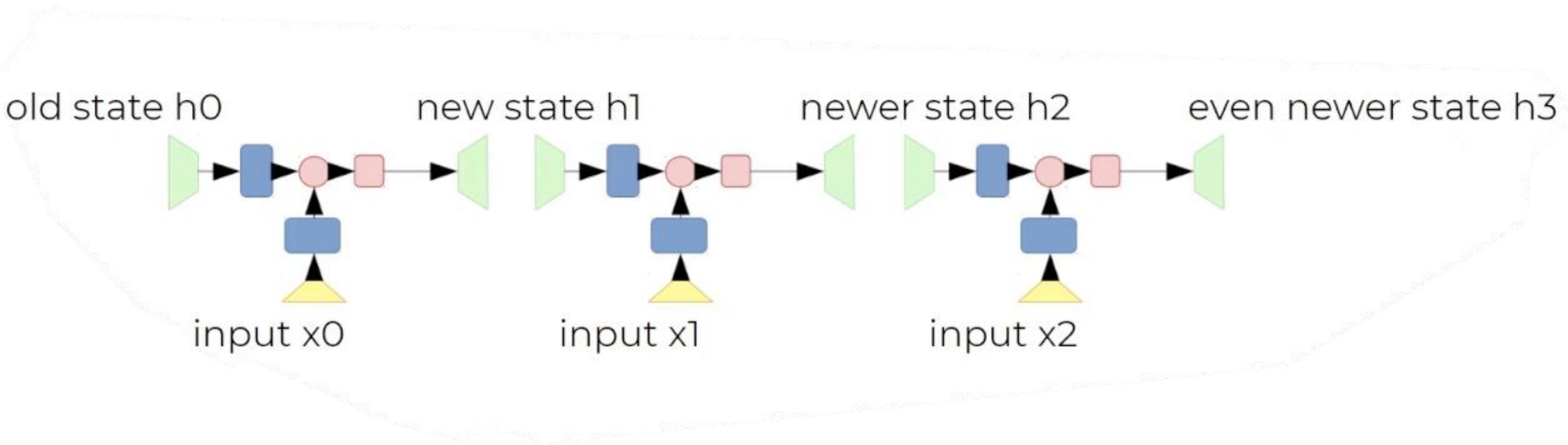
RNN

- Vocabulary is not infinite
- We can make embeddings
- We can combine embeddings



https://github.com/girafe-ai

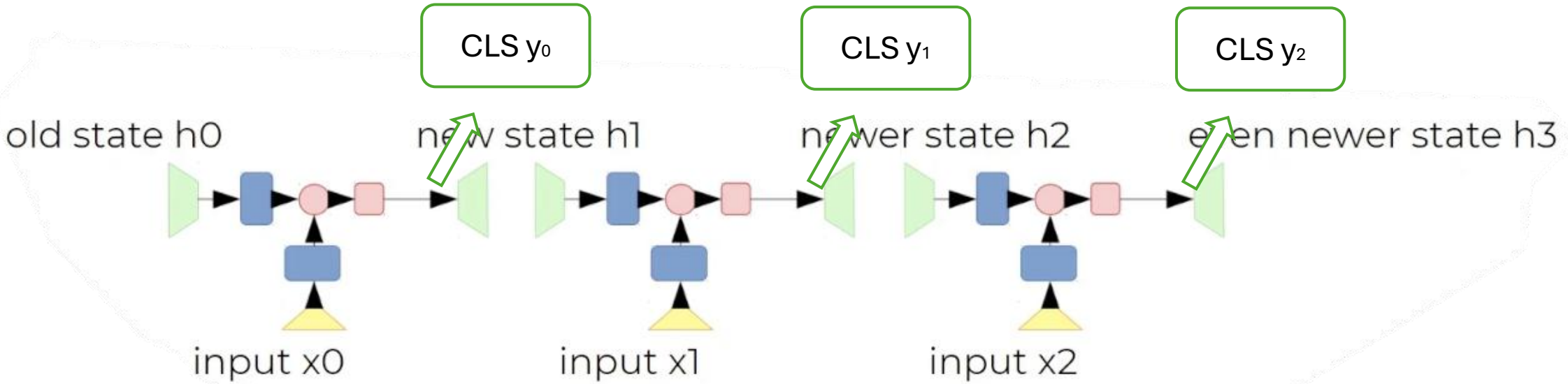
RNN



We use same weight matrices for all steps!

https://github.com/girafe-ai

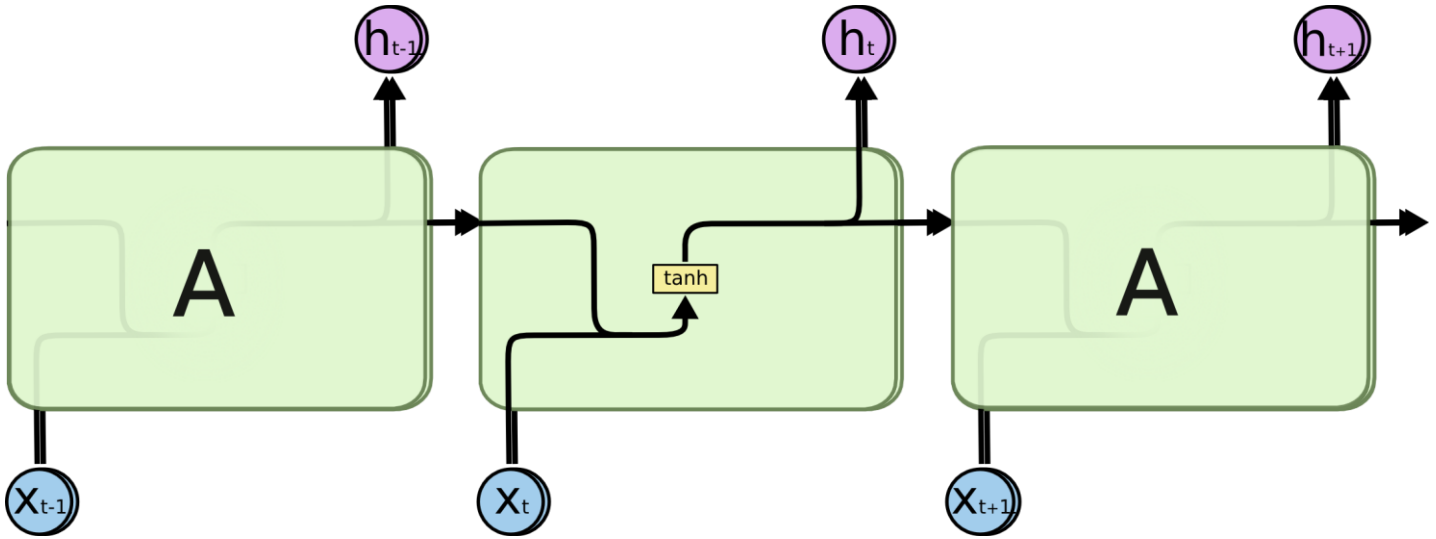
RNN



Vocabular is not infinite, so can easy classify new y token!

https://github.com/girafe-ai

RNN



$$h_0 = \bar{0}$$

$$h_1 = \sigma\big(\langle W_{\text{hid}}[h_0, x_0]\rangle + b\big)$$

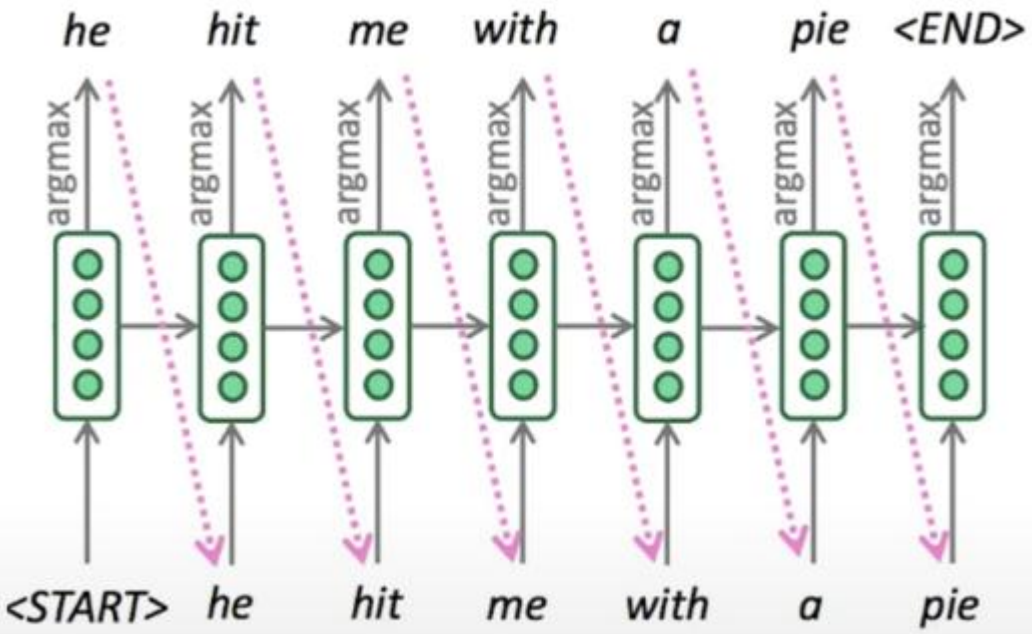
$$h_2 = \sigma\big(\langle W_{\text{hid}}[h_1, x_1]\rangle + b\big) = \sigma\big(\langle W_{\text{hid}}[\sigma\big(\langle W_{\text{hid}}[h_0, x_0]\rangle + b, x_1]\rangle + b\big)$$

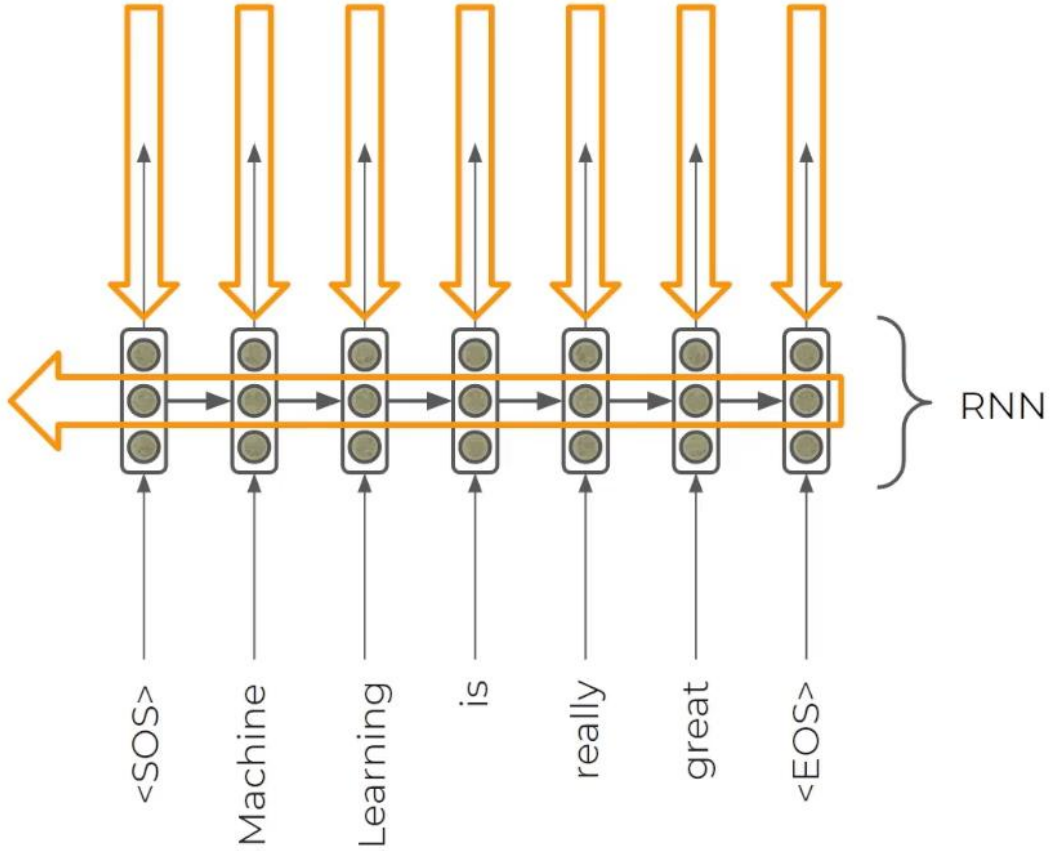
$$h_{i+1} = \sigma\big(\langle W_{\text{hid}}[h_i, x_i]\rangle + b\big)$$

$$P(x_{i+1}) = \text{softmax}\big(\langle W_{\text{out}}, h_i\rangle + b_{\text{out}}\big)$$

How to train?

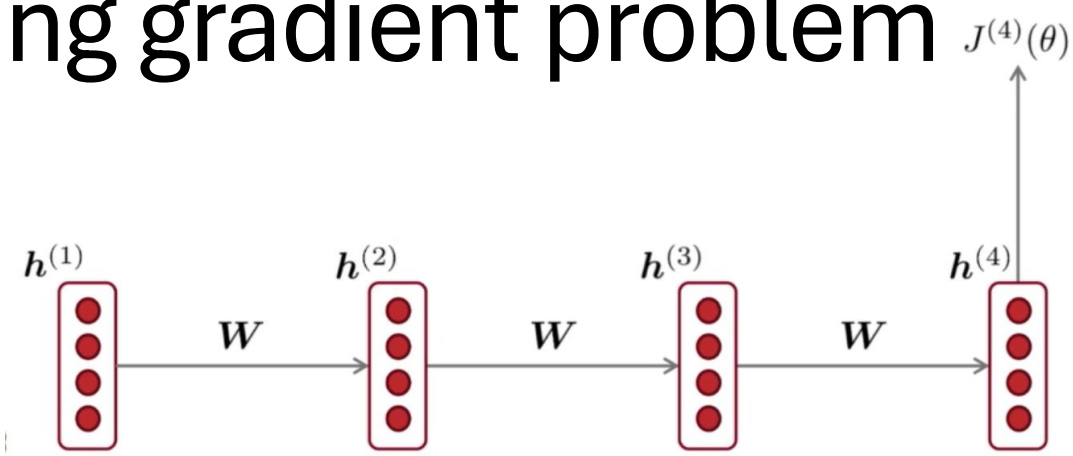
Loss
(e.g. Negative
log-likelihood)

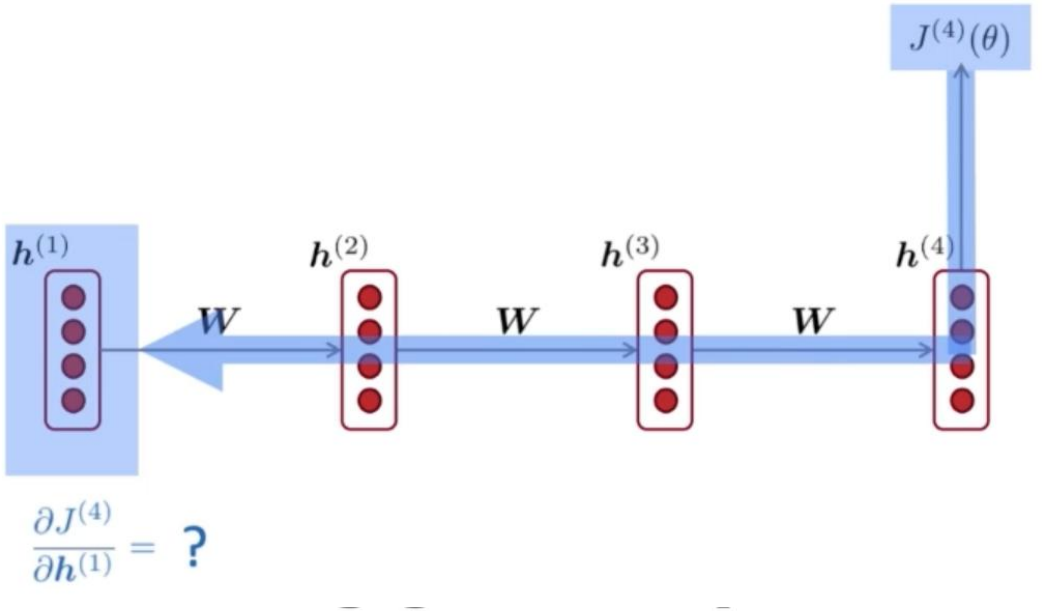


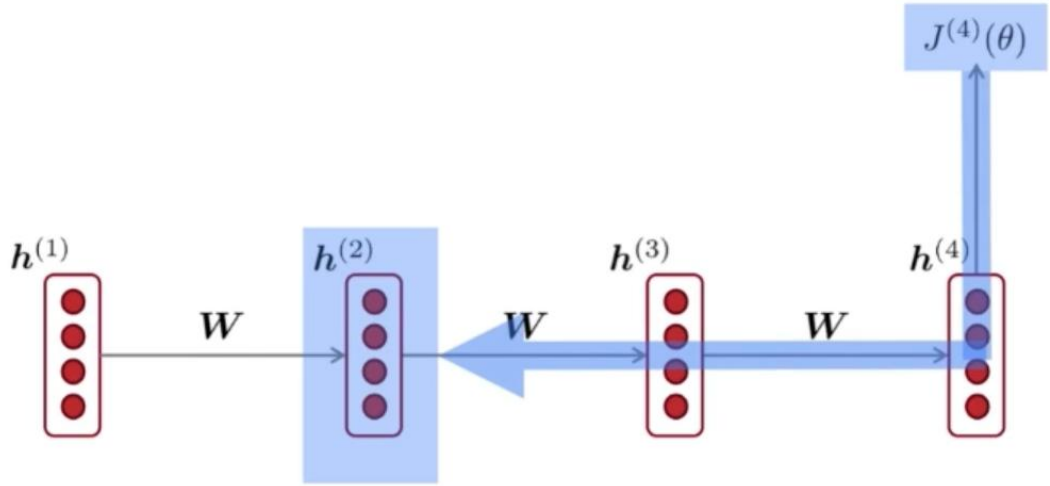


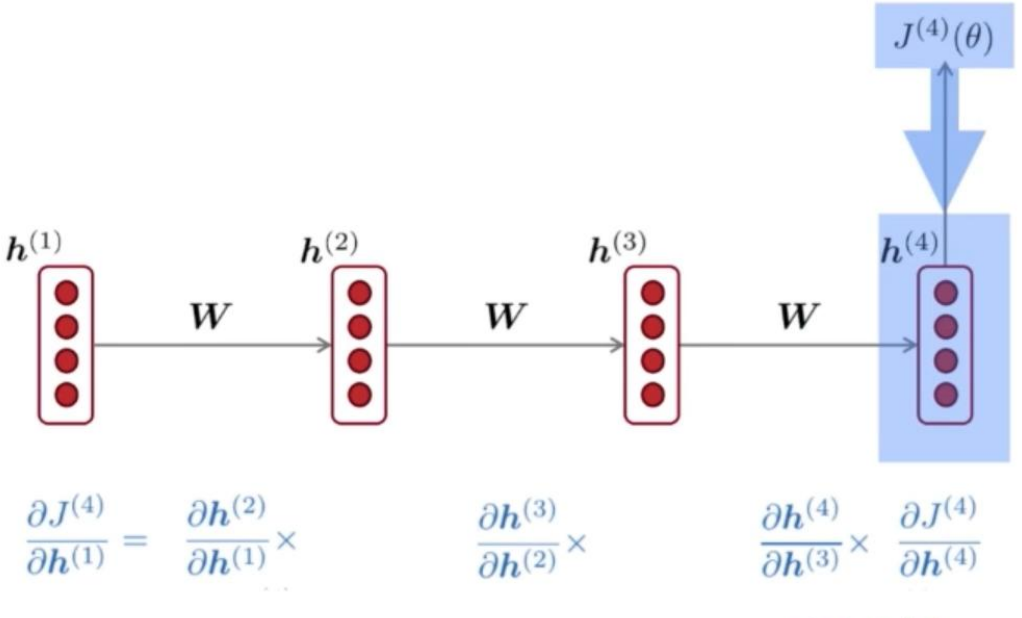
https://github.com/girafe-ai

Vanishing gradient problem





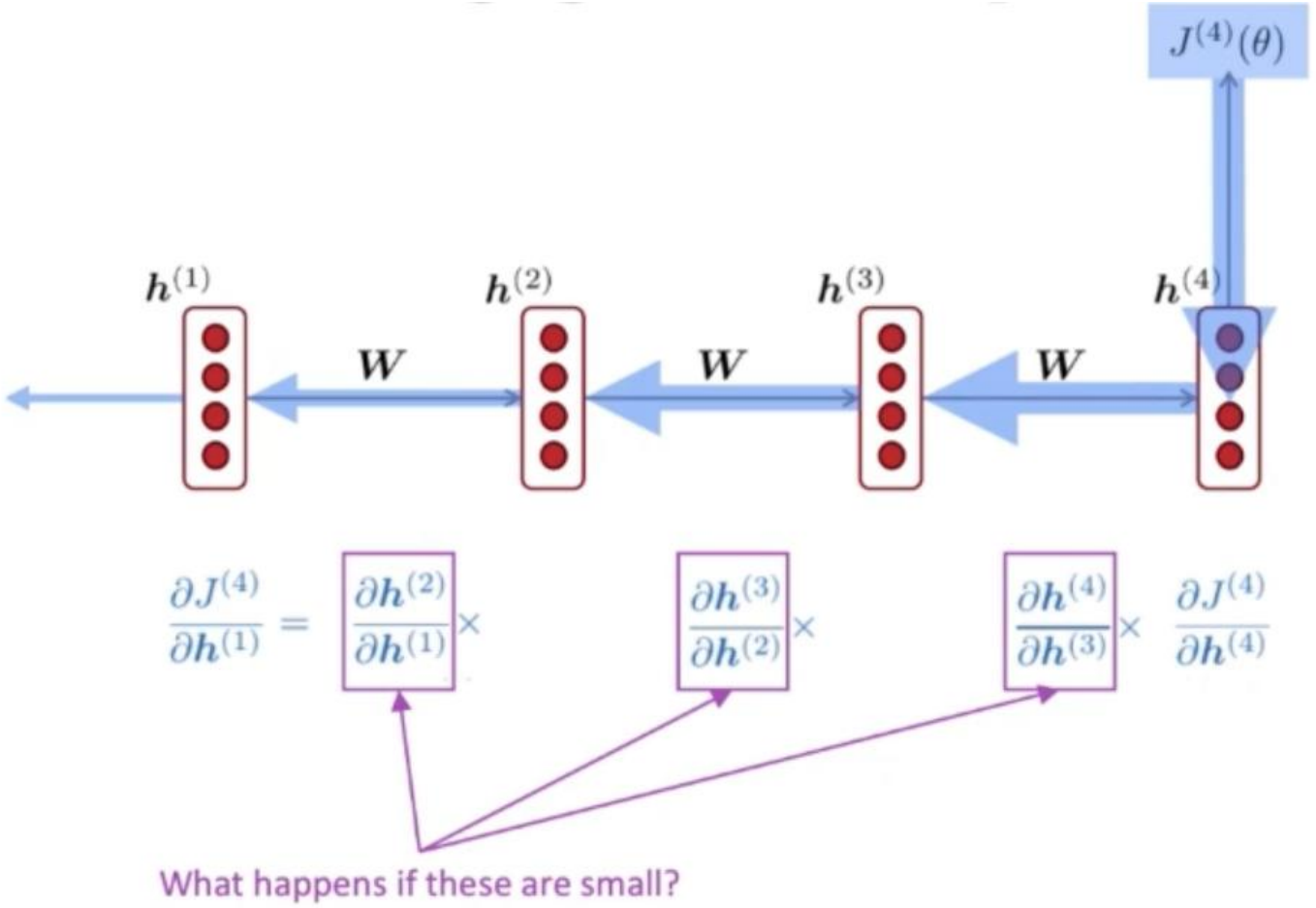




https://github.com/girafe-ai

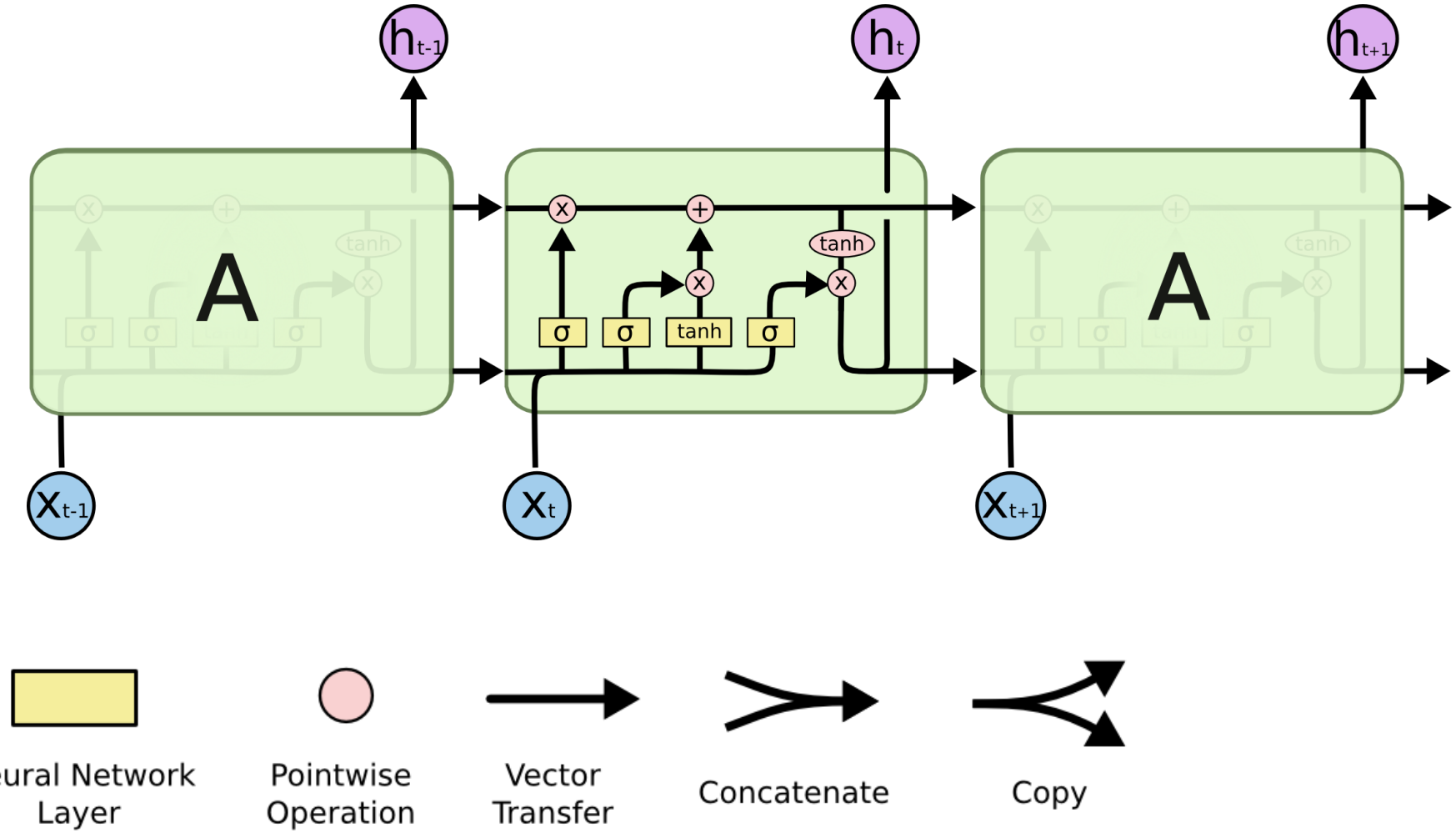
Vanishing gradient problem

When the derivatives are small, the gradient signal gets smaller and smaller as it backpropagates further



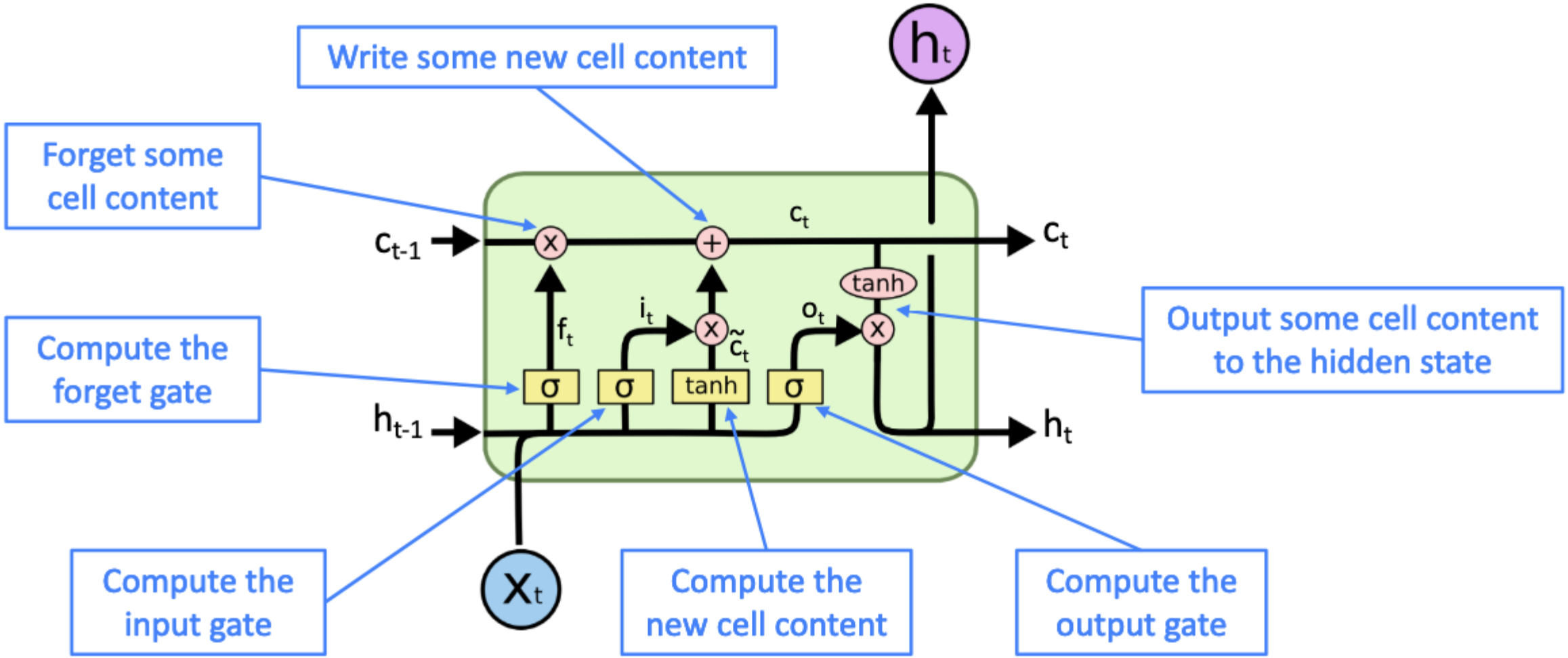
https://github.com/girafe-ai

LSTM



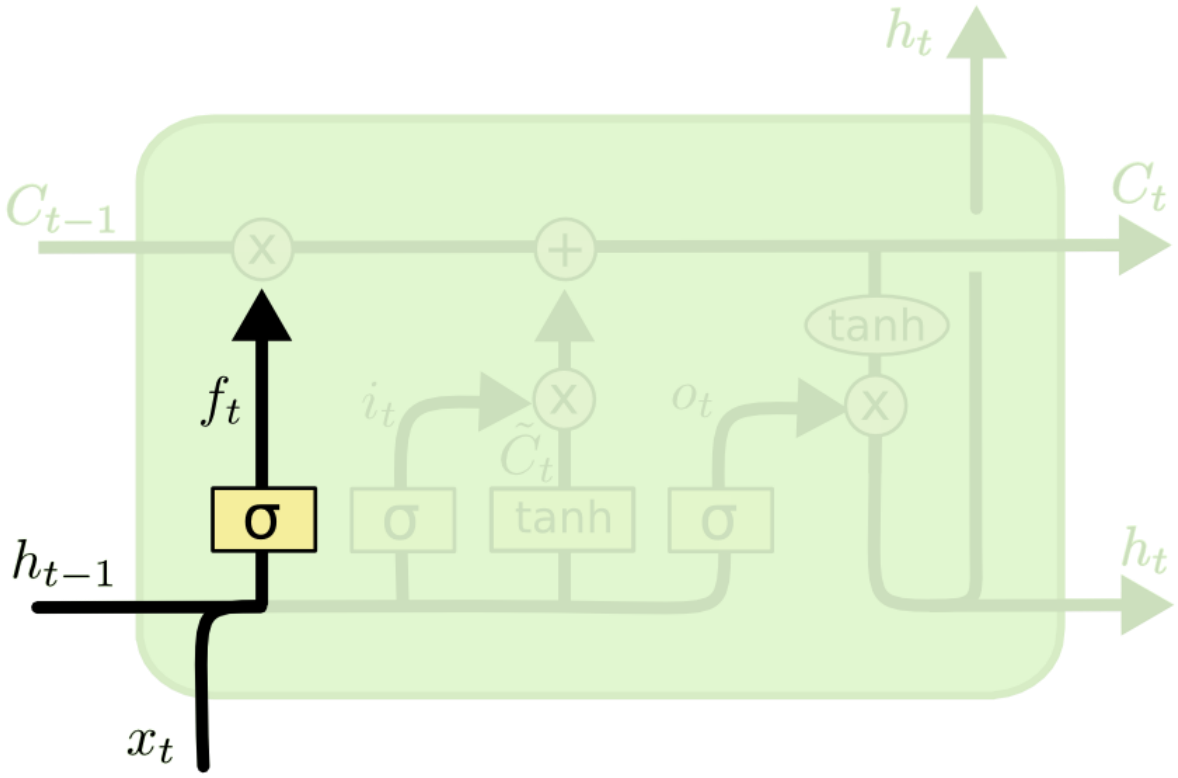
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM



https://github.com/girafe-ai

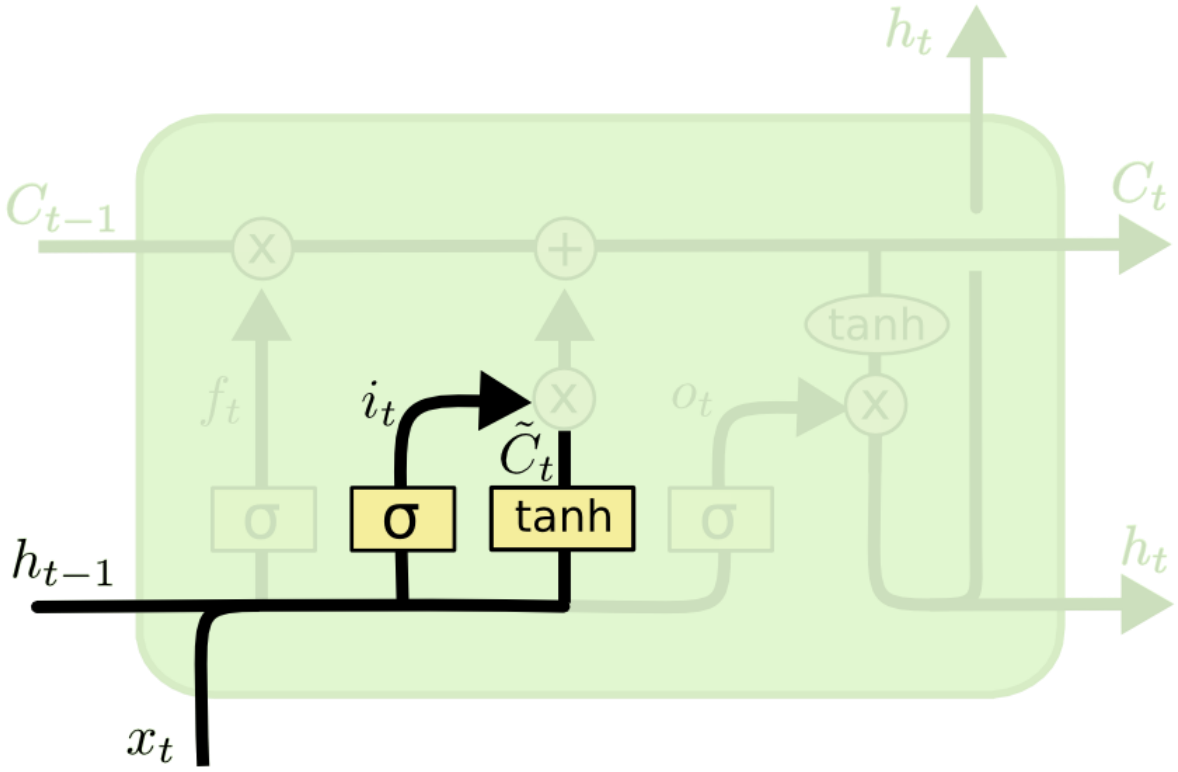
LSTM. Forget gate



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM. Input gate

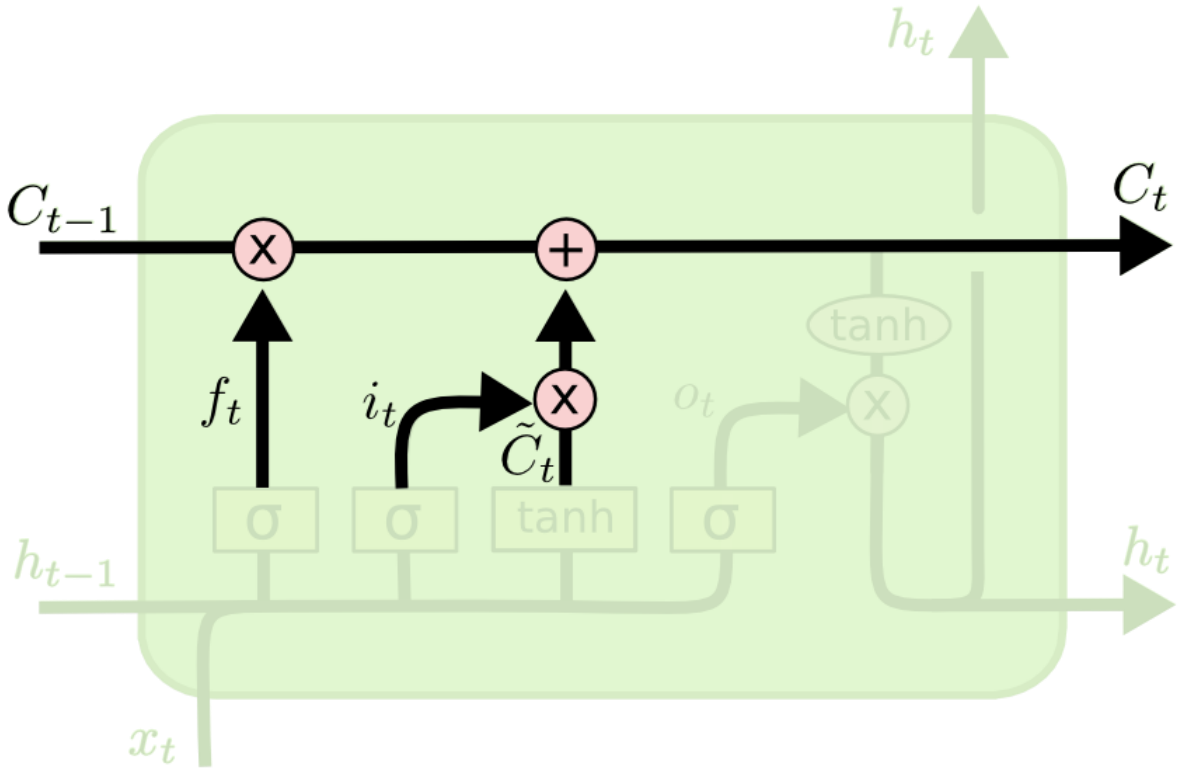


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

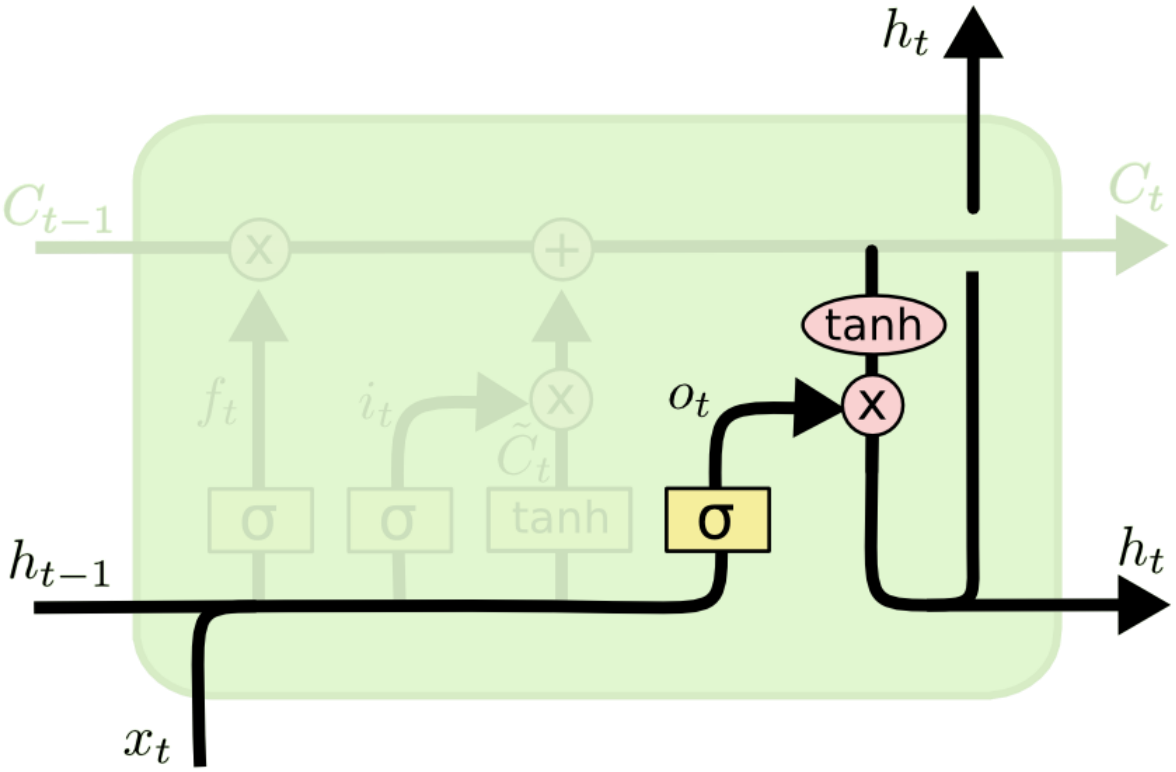
LSTM. Memory update



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM. Output gate and hidden state update

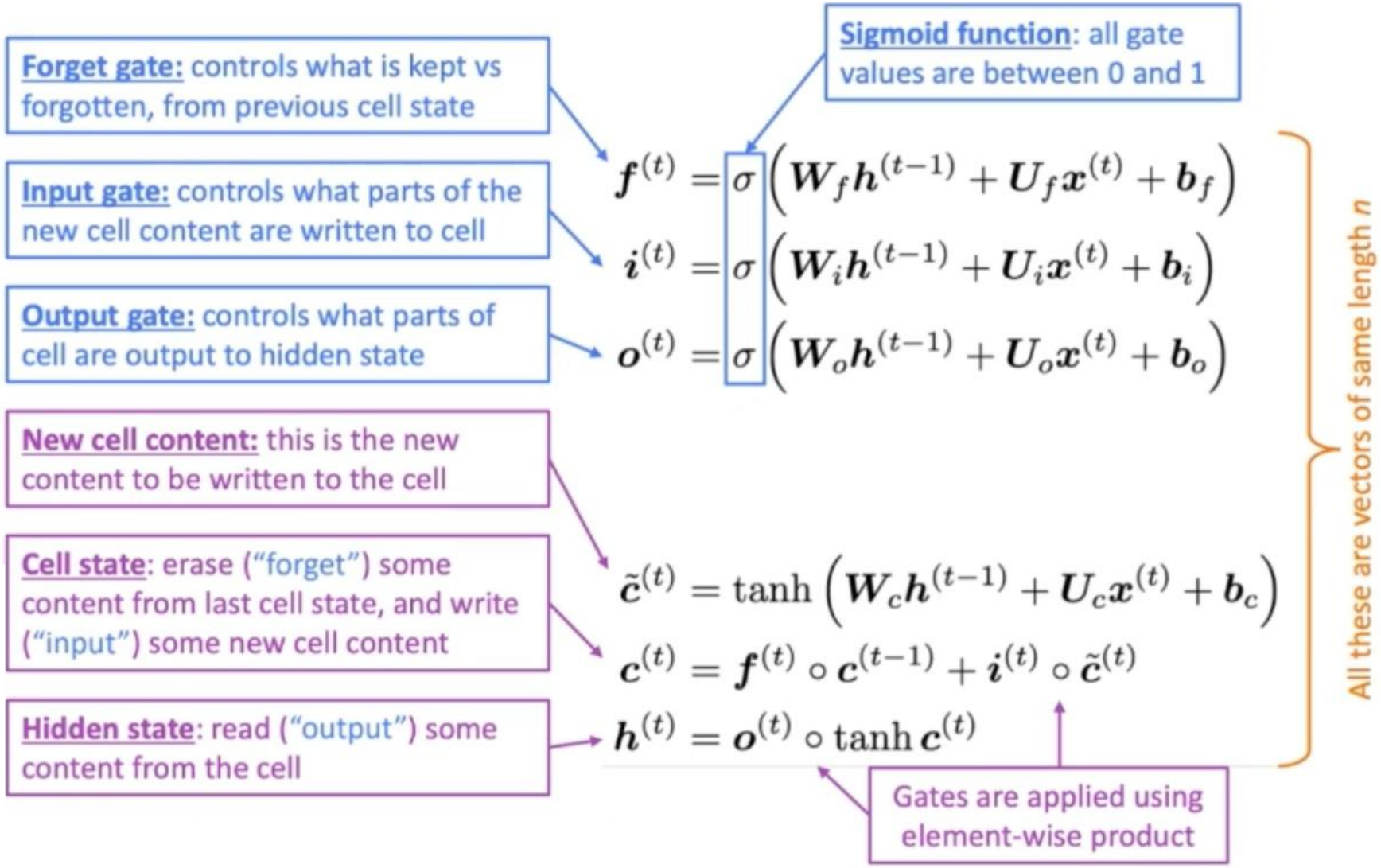


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

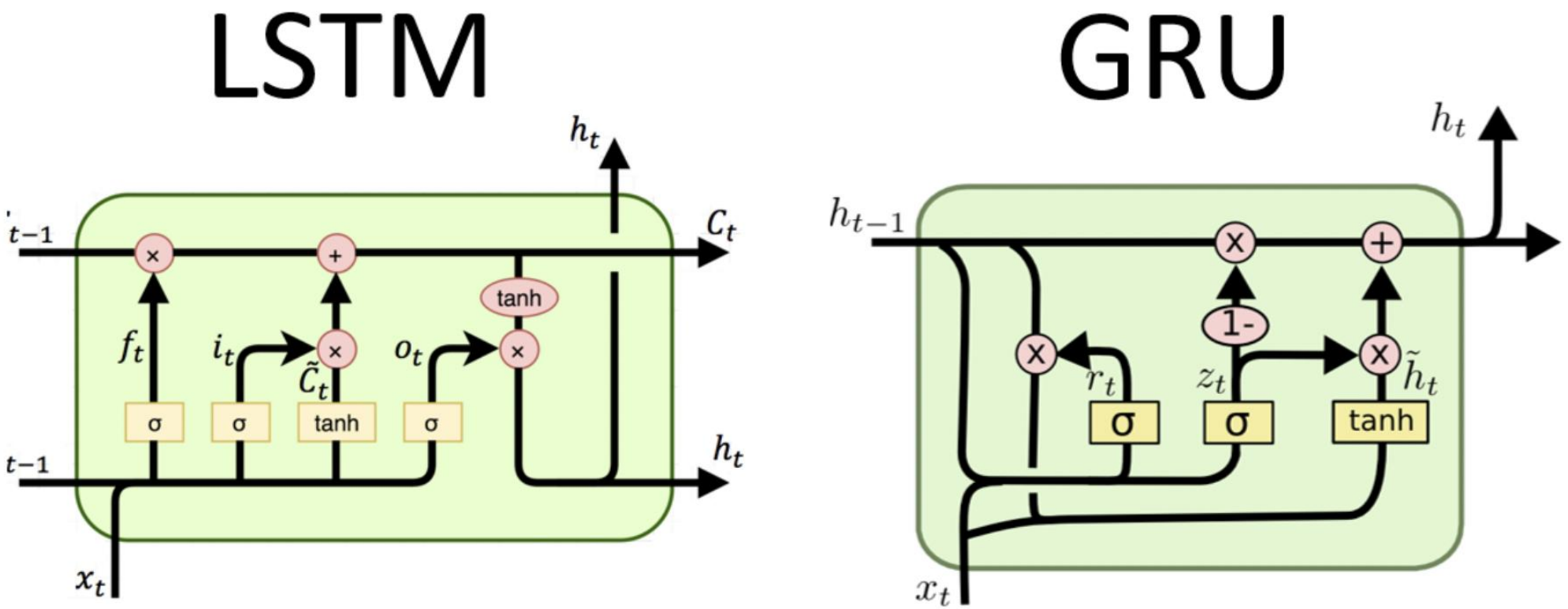
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM.



https://github.com/girafe-ai

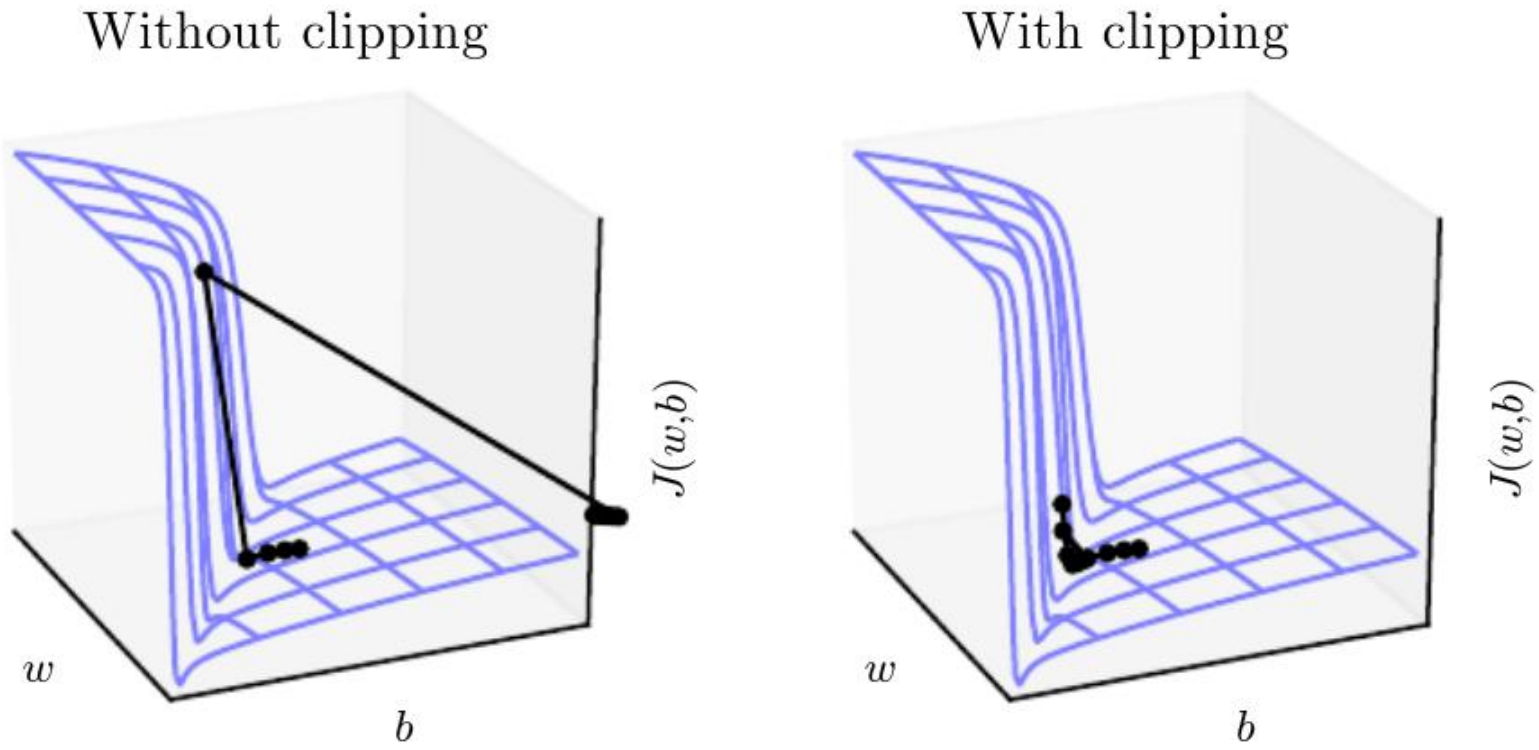
GRU



https://github.com/haksorus/walmart-time-series

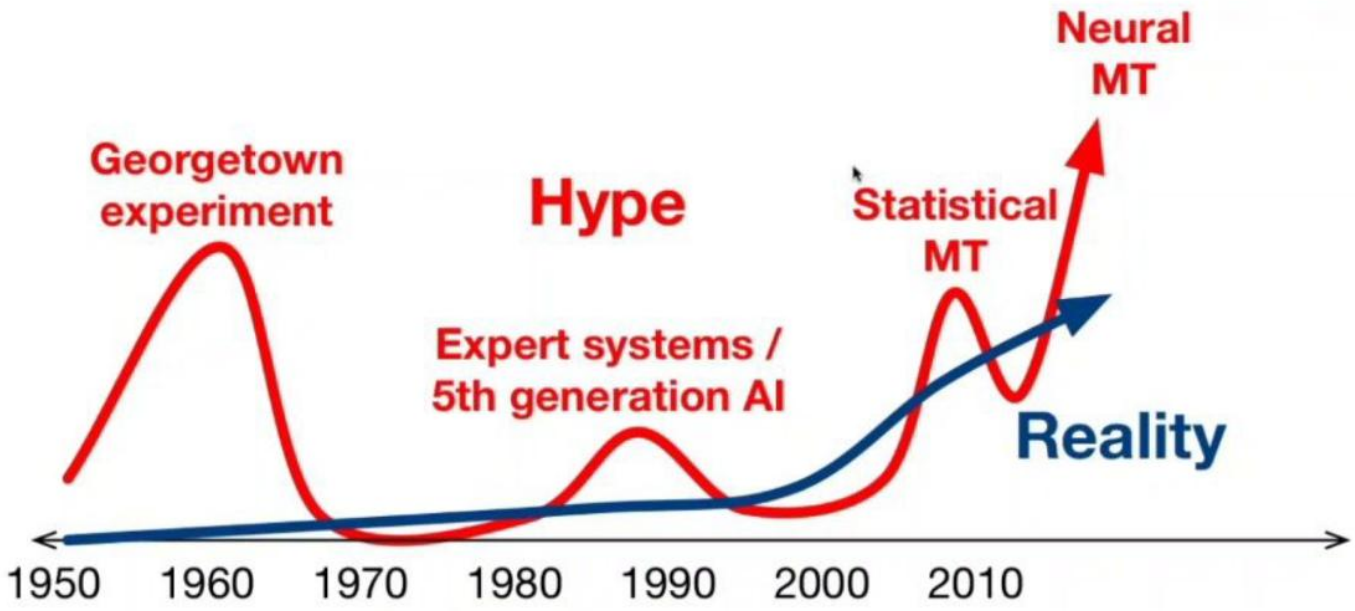
Exploding gradient problem

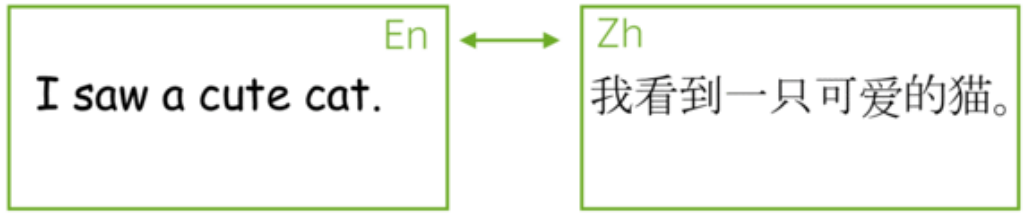
If we see big update step, maybe we can fix it?



https://slides.com/matthieulabeau/update-of-sequence-modeling-mdi341-2022

Mashing translation

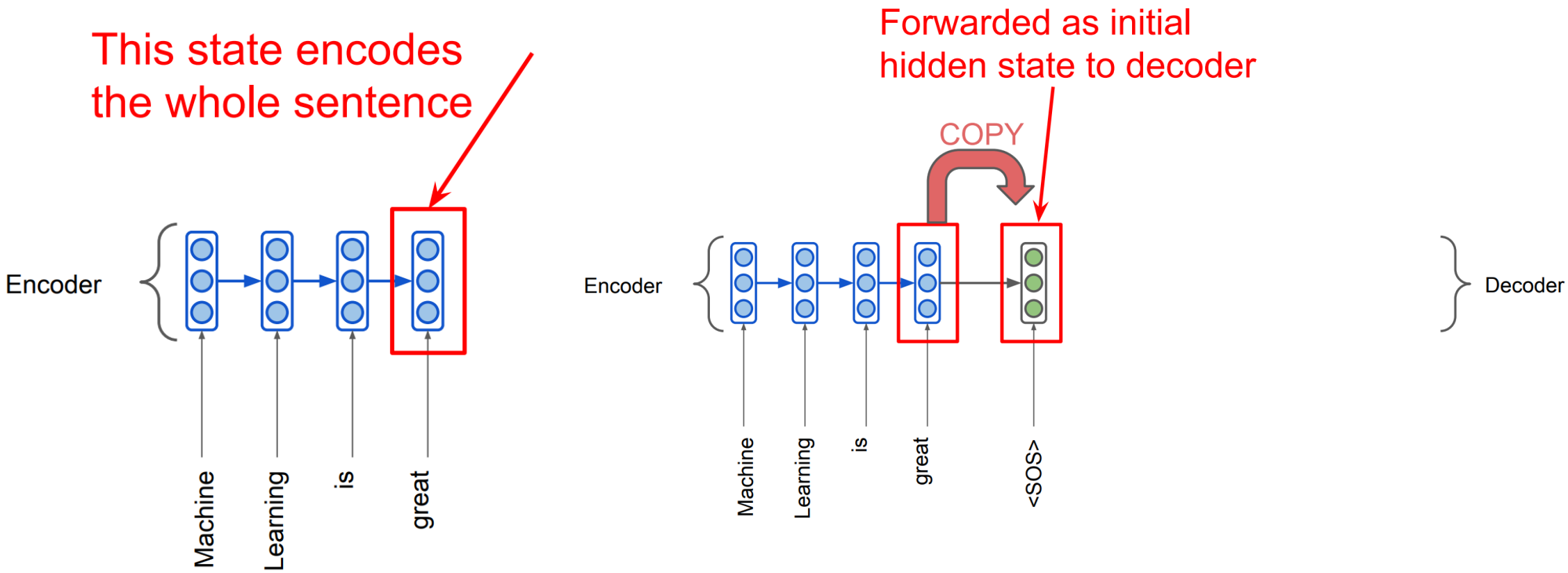




https://github.com/girafe-ai/ml-course/blob/25s_econom/ml-basic_lect10_machine_translation_attention.pdf

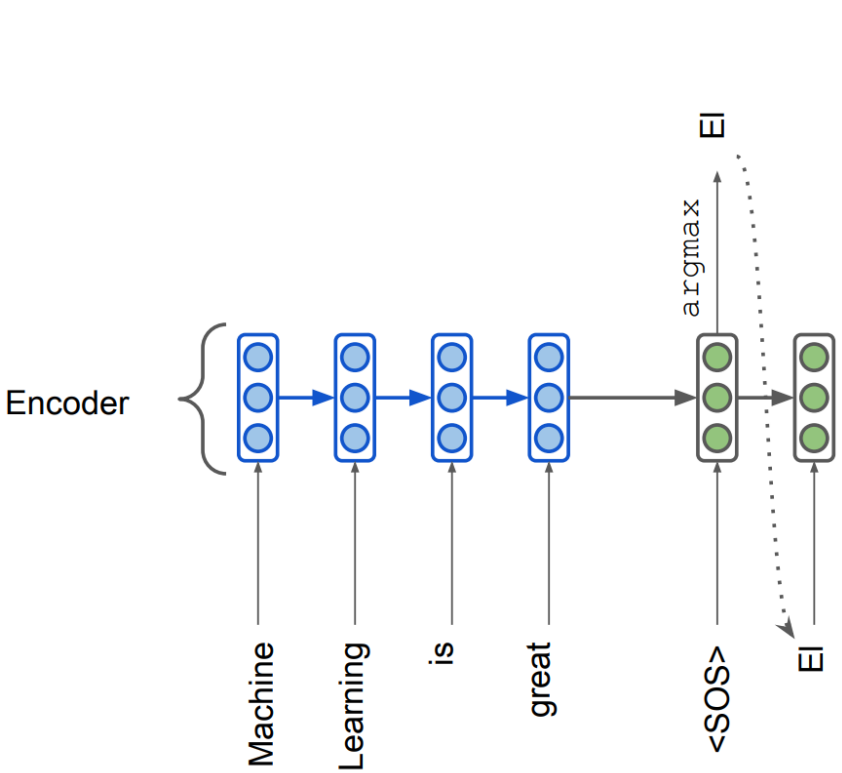
https://lena-voita.github.io/nlp_course

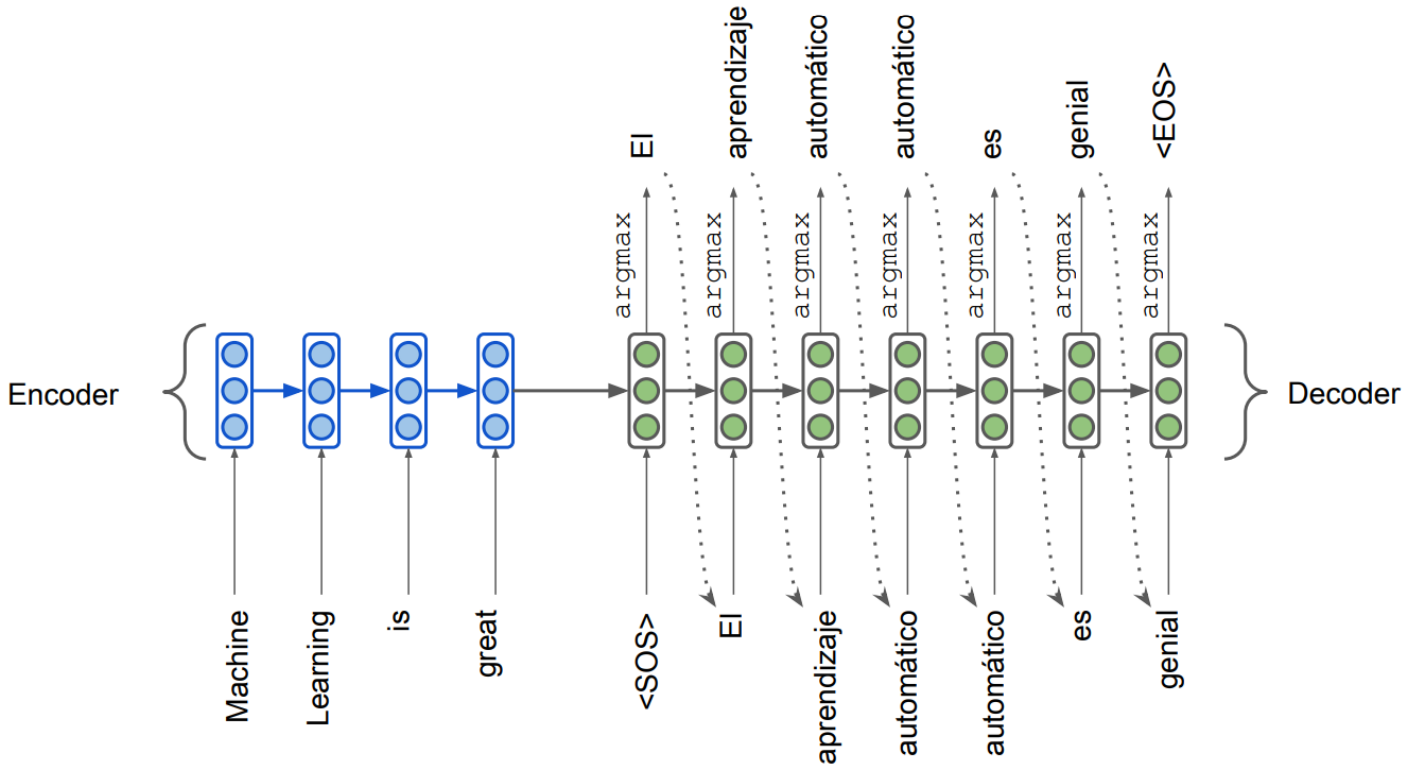
Mashing translation



https://github.com/girafe-ai/ml-course/blob/25s_econom/ml-basic_lect10_machine_translation_attention.pdf

Mashing translation





https://github.com/girafe-ai/ml-course/blob/25s_econom/ml-basic_lect10_machine_translation_attention.pdf

Mashing translation

- NMT directly calculates P(y|x)
  - y – target sentence, x – source sentence

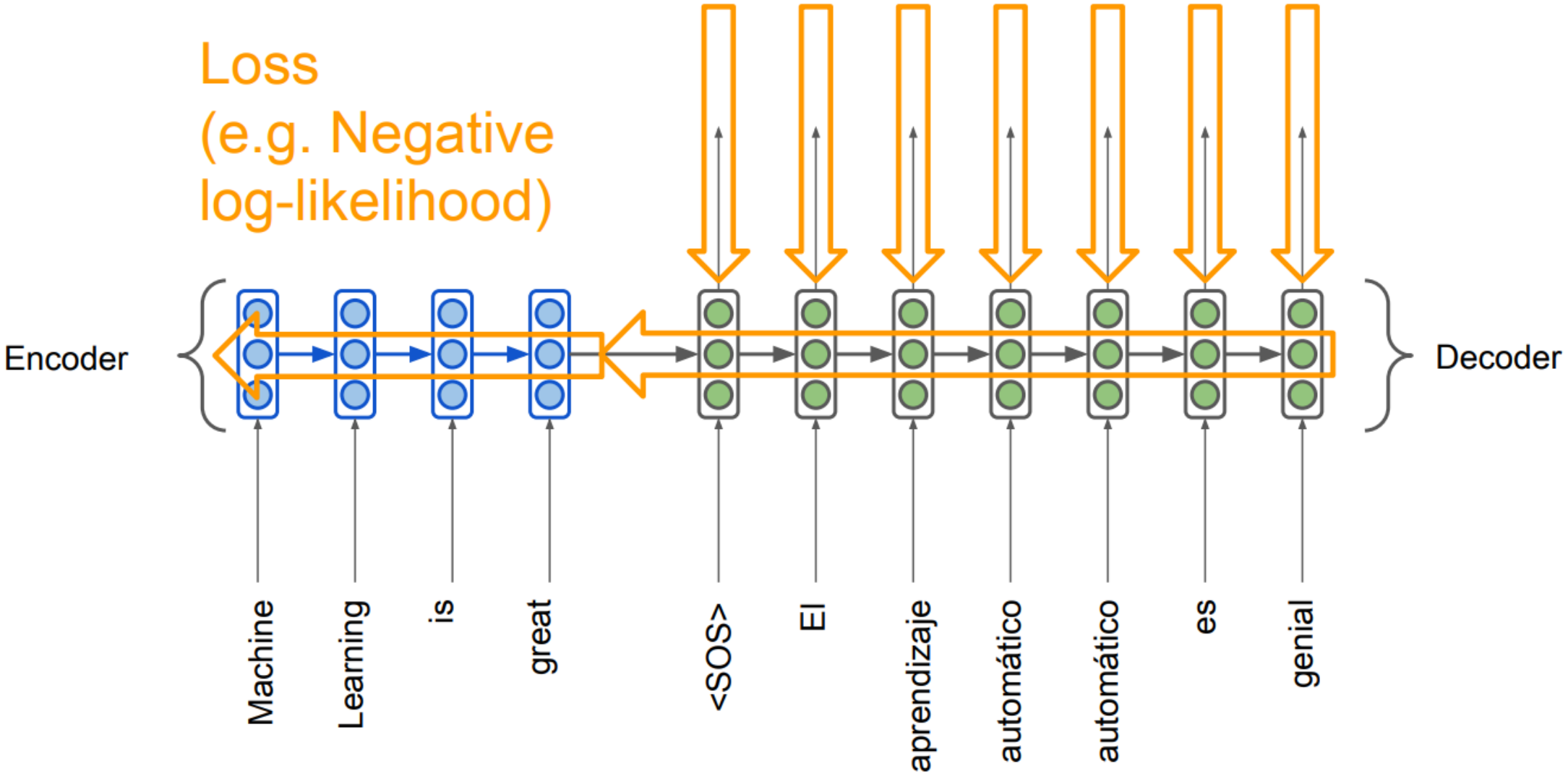
$$P(y|x) = P(y_2|y_1, x)P(y_3|y_1, y_2, x)\ldots \underbrace{P(y_T|y_1, y_2, \ldots, x)}$$

Probability of next word
in target language

- To train it we need a huge parallel corpus.

https://github.com/girafe-ai/ml-course/blob/25s_econom/ml-basic_lect10_machine_translation_attention.pdf

Mashing translation



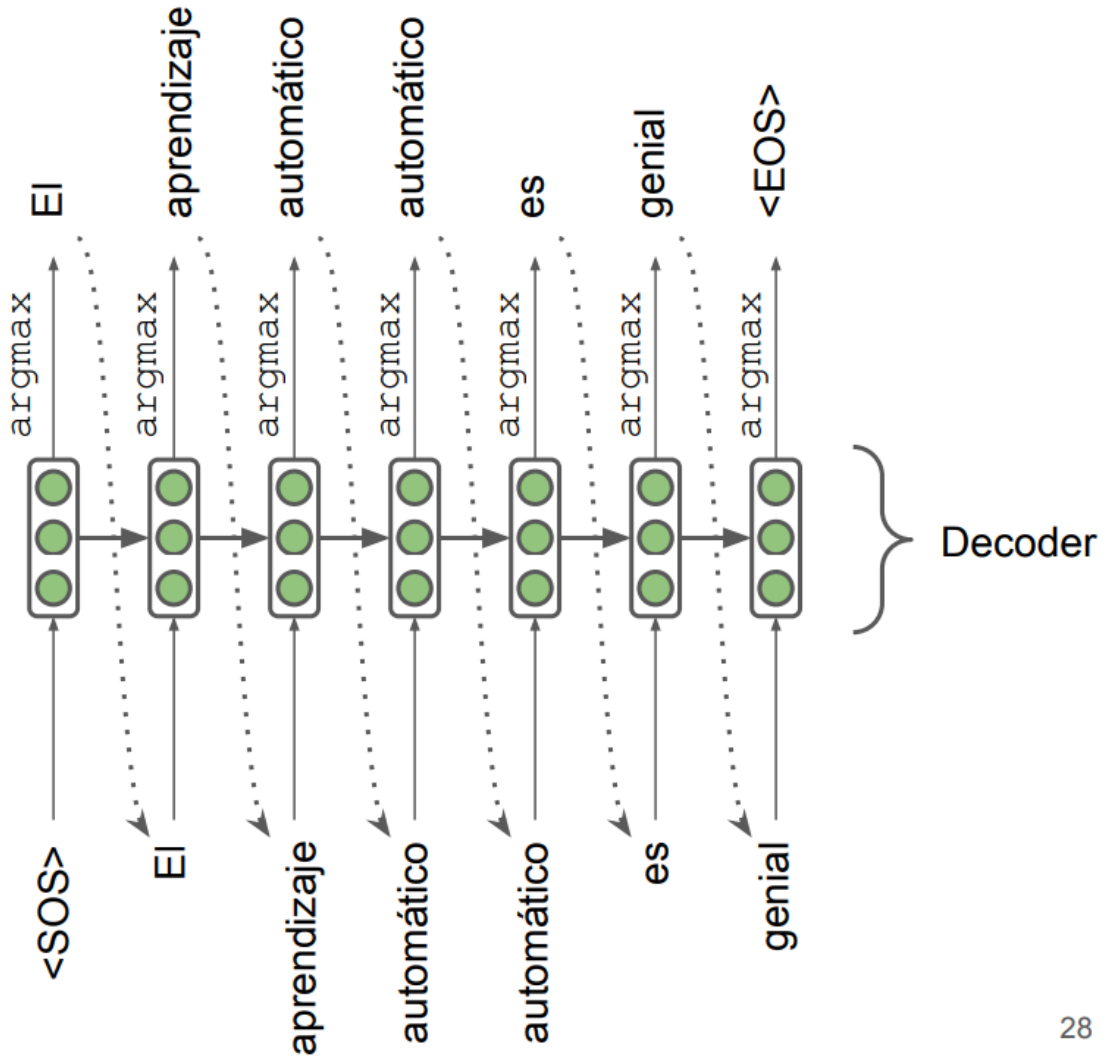
https://github.com/girafe-ai/ml-course/blob/25s_econom/ml-basic_lect10_machine_translation_attention.pdf

Mashing translation

- Decoder predicts the most probable token (argmax) on each step
- The approach is **greedy**

Any problems with it?

Any mistake is treated as input on the next step!



https://github.com/girafe-ai/ml-course/blob/25s_econom/ml-basic_lect10_machine_translation_attention.pdf

Mashing translation

- We want the translation that maximizes the likelihood:

$$P(y|x) = P(y_1|x) \prod_{t=2}^{T} P(y_t|y_1, \dots y_{t-1}, x)$$

- We cannot compute all the possible sequences (exponential complexity)

https://github.com/girafe-ai/ml-course/blob/25s_econom/ml-basic_lect10_machine_translation_attention.pdf

Mashing translation

- 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 has a score which is its log probability:

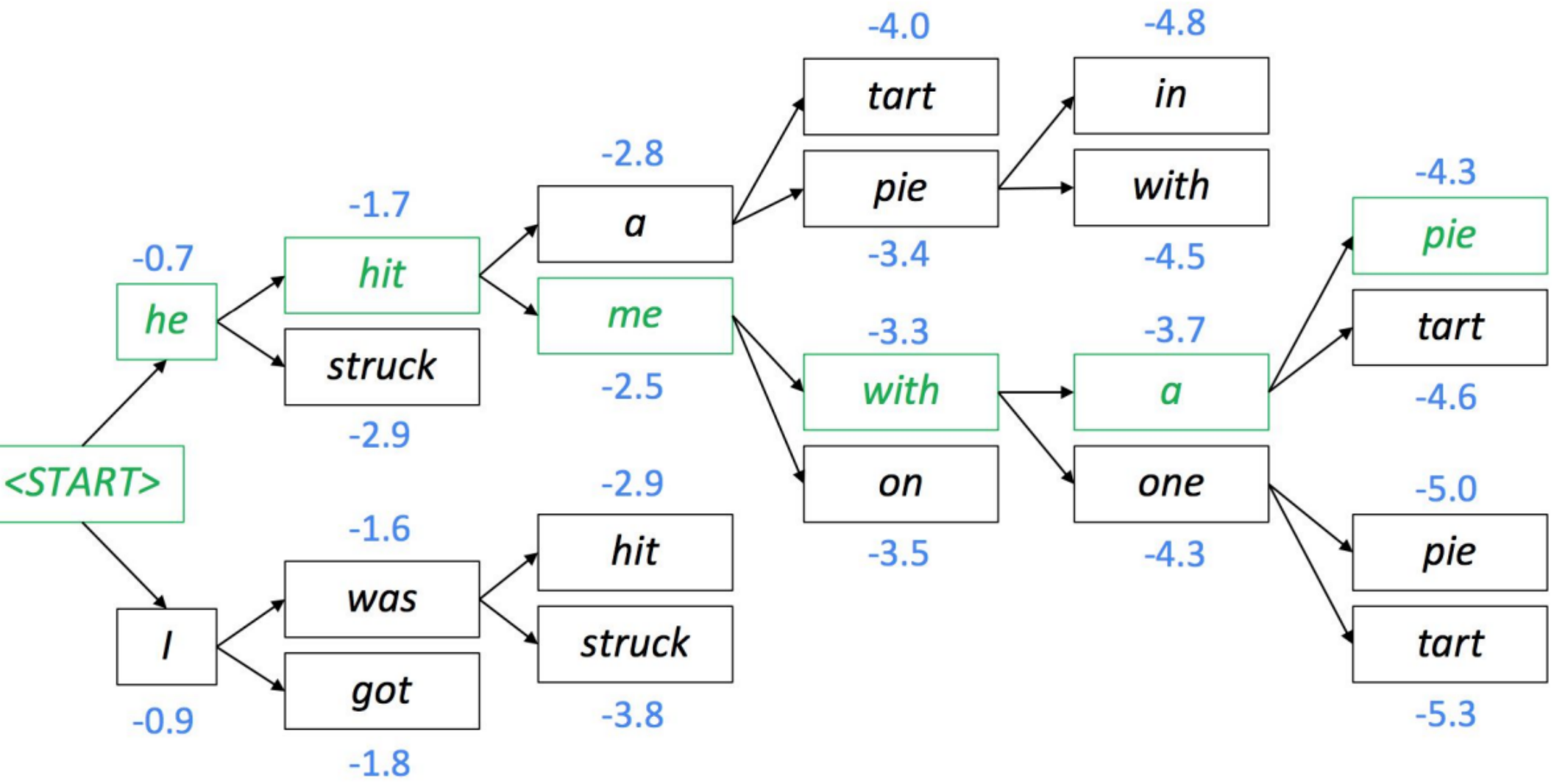
$$\text{score}(y_1, \ldots, y_t) = \log P_{\text{LM}}(y_1, \ldots, y_t | x) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$$

- We search for high-scoring hypotheses, tracking top k on each step
- Beam search does not guarantee finding optimal solution

https://github.com/girafe-ai/ml-course/blob/25s_econom/ml-basic_lect10_machine_translation_attention.pdf

Beam search. Example.

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



Source: http://web.stanford.edu/class/cs224n/slides/cs224n-2019-lecture08-nmt.pdf

Beam search. Stopping.

- In **greedy decoding**, usually we decode until the model produces <EOS> token
- In **beam search decoding**, different hypotheses may produce <EOS> tokens on different timesteps
  - When a hypothesis produces <EOS>, that hypothesis is complete.
  - Place it aside and continue exploring other hypotheses via beam search.
- Usually we continue beam search until:
  - We reach pre-defined timestep T
  - We have at least n completed hypotheses

Beam search. How to find best?

- How to select top one with highest score?
- Each hypothesis on our list has a score:

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

- **Problems?**

  Longer hypotheses have lower scores

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

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

BLEU

BLEU (Bilingual Evaluation Understudy) compares the machine-written translation to human-written translation, and computes a similarity score based on:

- n-gram precision
- penalty for too-short system translations (brevity penalty)

$$BLEU = \text{brevity penalty} \cdot \left(\prod_{i=1}^{n} \text{precision}_i \right)^{1/n} \cdot 100\%$$

$$\text{brevity penalty} = min \left(1, \frac{\text{output length}}{\text{reference length}} \right)$$

https://github.com/girafe-ai

BLEU

BLEU (Bilingual Evaluation Understudy) compares the machine-written translation to human-written translation, and computes a similarity score based on:

- n-gram precision
- brevity penalty

SYSTEM A: Israeli officials (2-GRAM MATCH) responsibility of airport (1-GRAM MATCH) safety

REFERENCE: Israeli officials are responsible for airport security

SYSTEM B: airport security (2-GRAM MATCH) Israeli officials are responsible (4-GRAM MATCH)

Metric	System A	System B
precision (1gram)	3/6	6/6
precision (2gram)	1/5	4/5
precision (3gram)	0/4	2/4
precision (4gram)	0/3	1/3
brevity penalty	6/7	6/7
BLEU	0%	52%

$$BLEU = \text{brevity penalty} \cdot \left(\prod_{i=1}^{n} \text{precision}_i \right)^{1/n} \cdot 100\%$$

https://github.com/girafe-ai

Metrics

- *BLEU*
- *METEOR*
- *ROUGE*
- *BERT-score*
- *BLEURT*
- *XCOMET*

https://llmstudio.ru/blog/bleu-rouge-bert-score

Attention

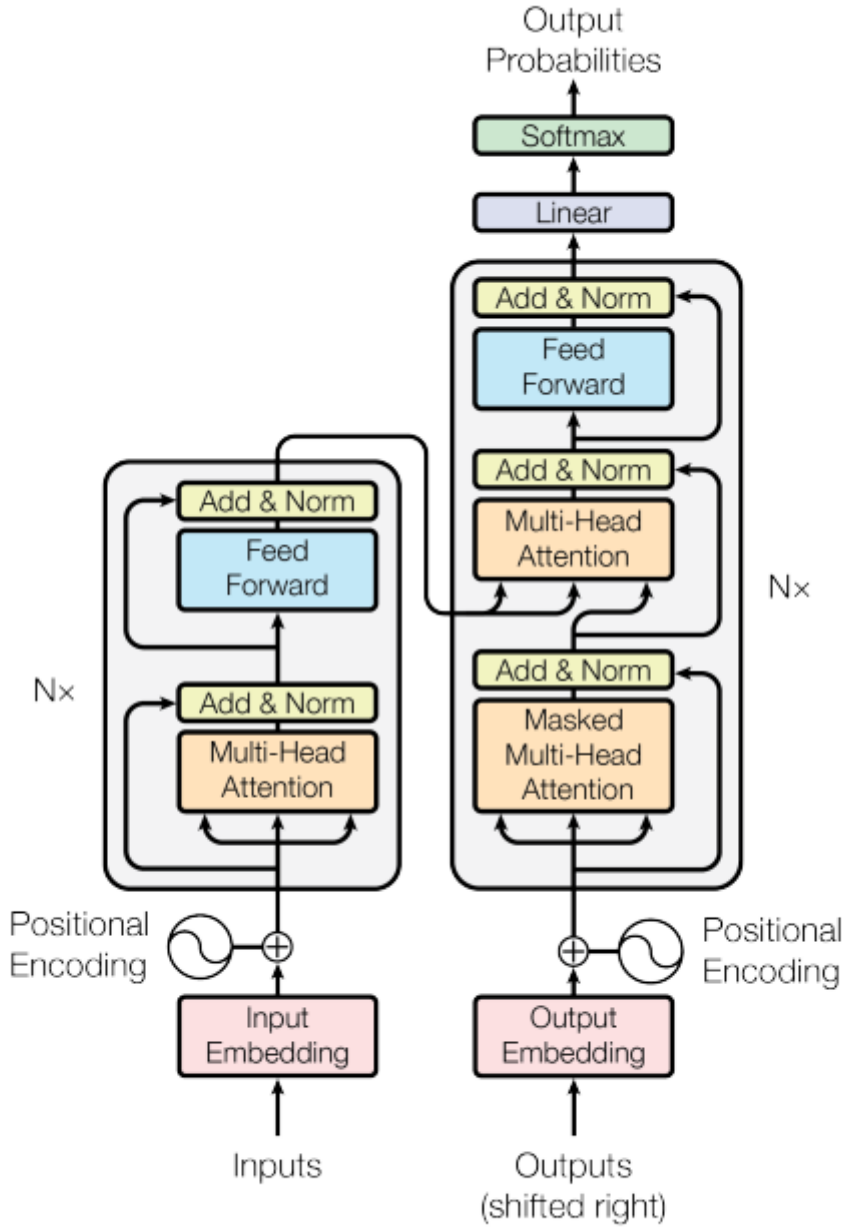
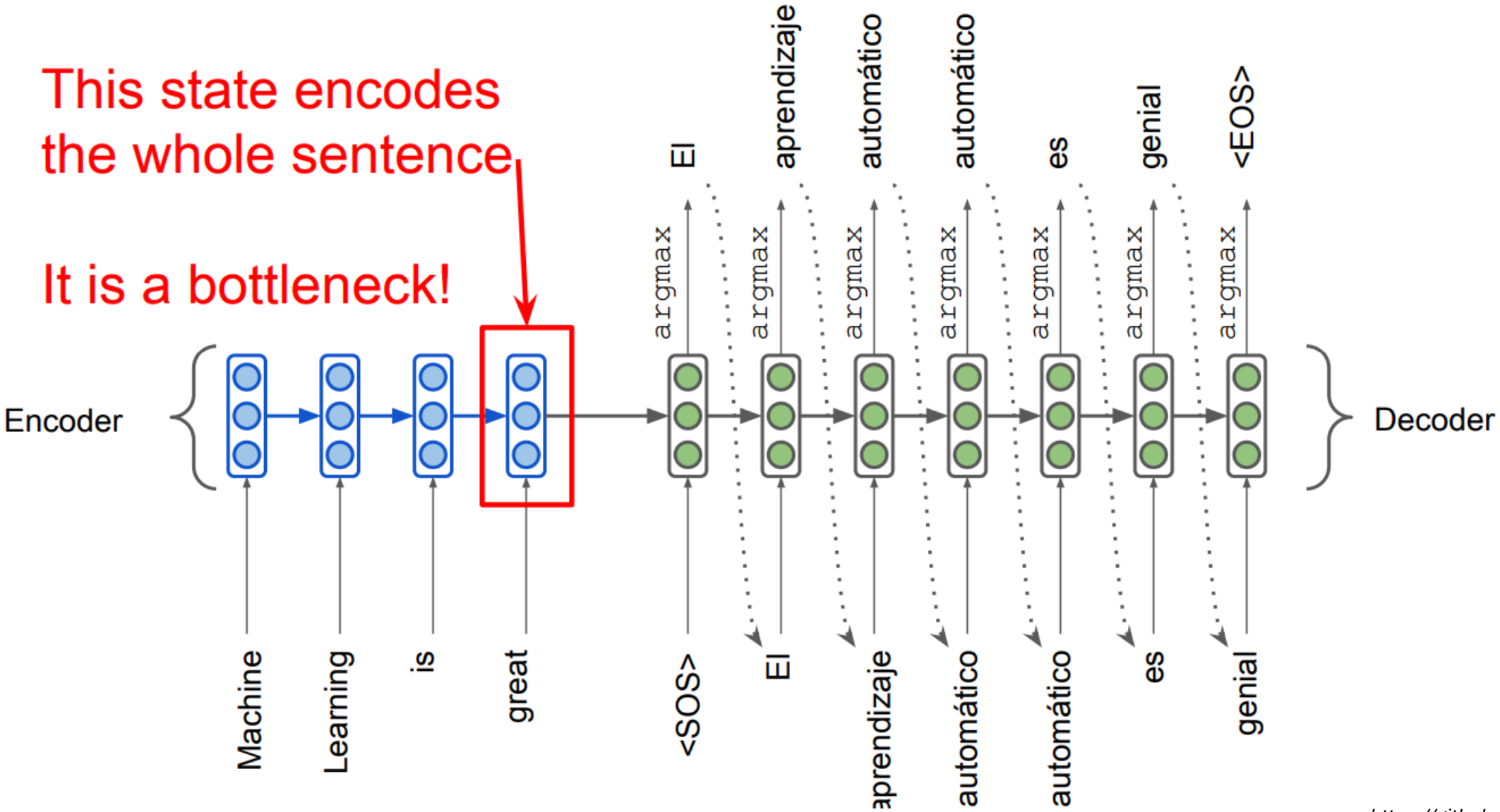


Figure 1: The Transformer - model architecture.

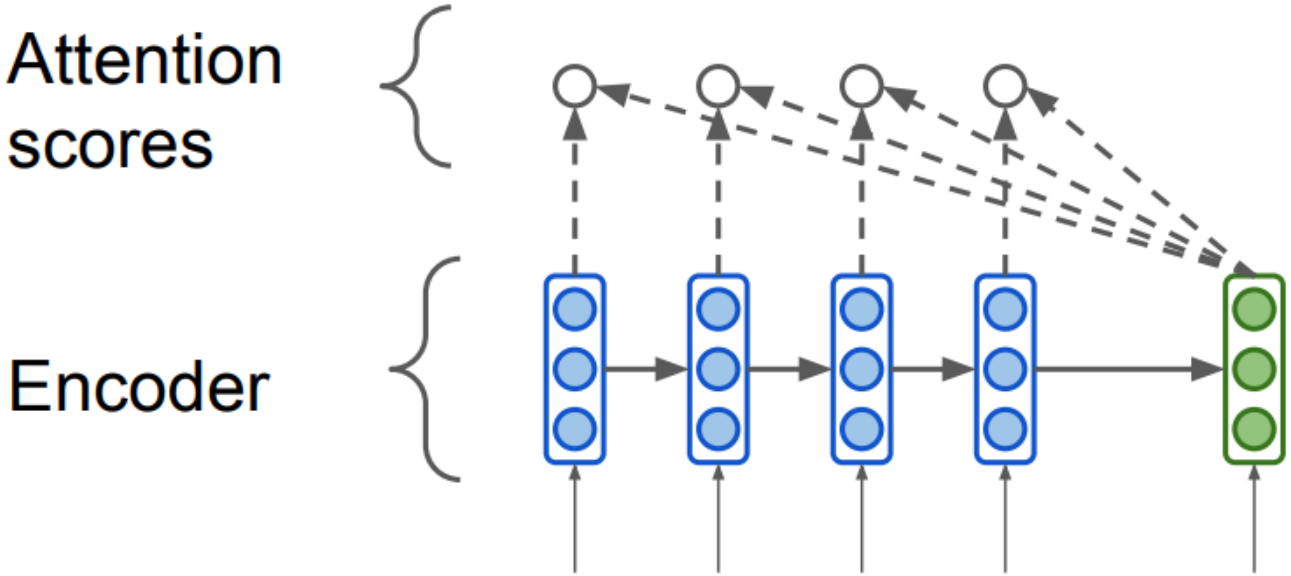
https://arxiv.org/pdf/1706.03762

Attention



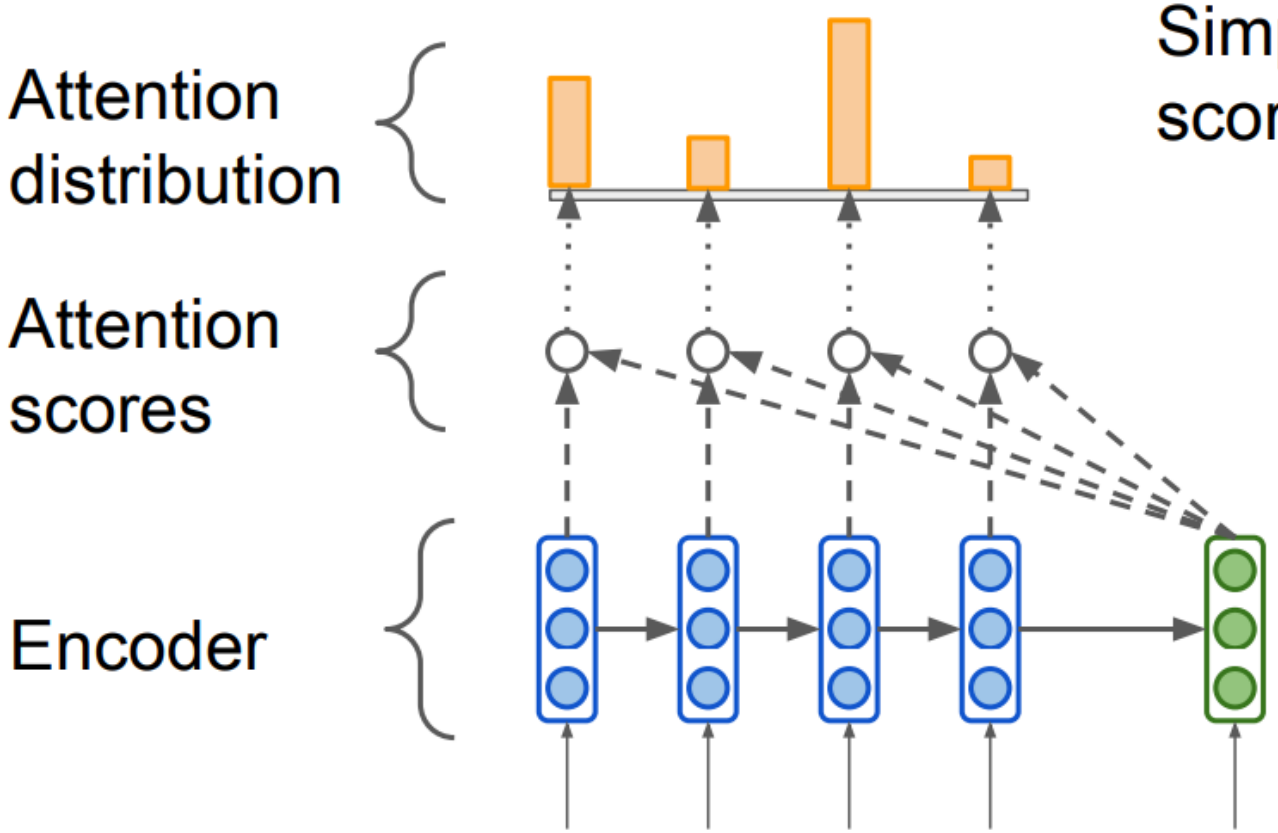
https://github.com/girafe-ai

Attention



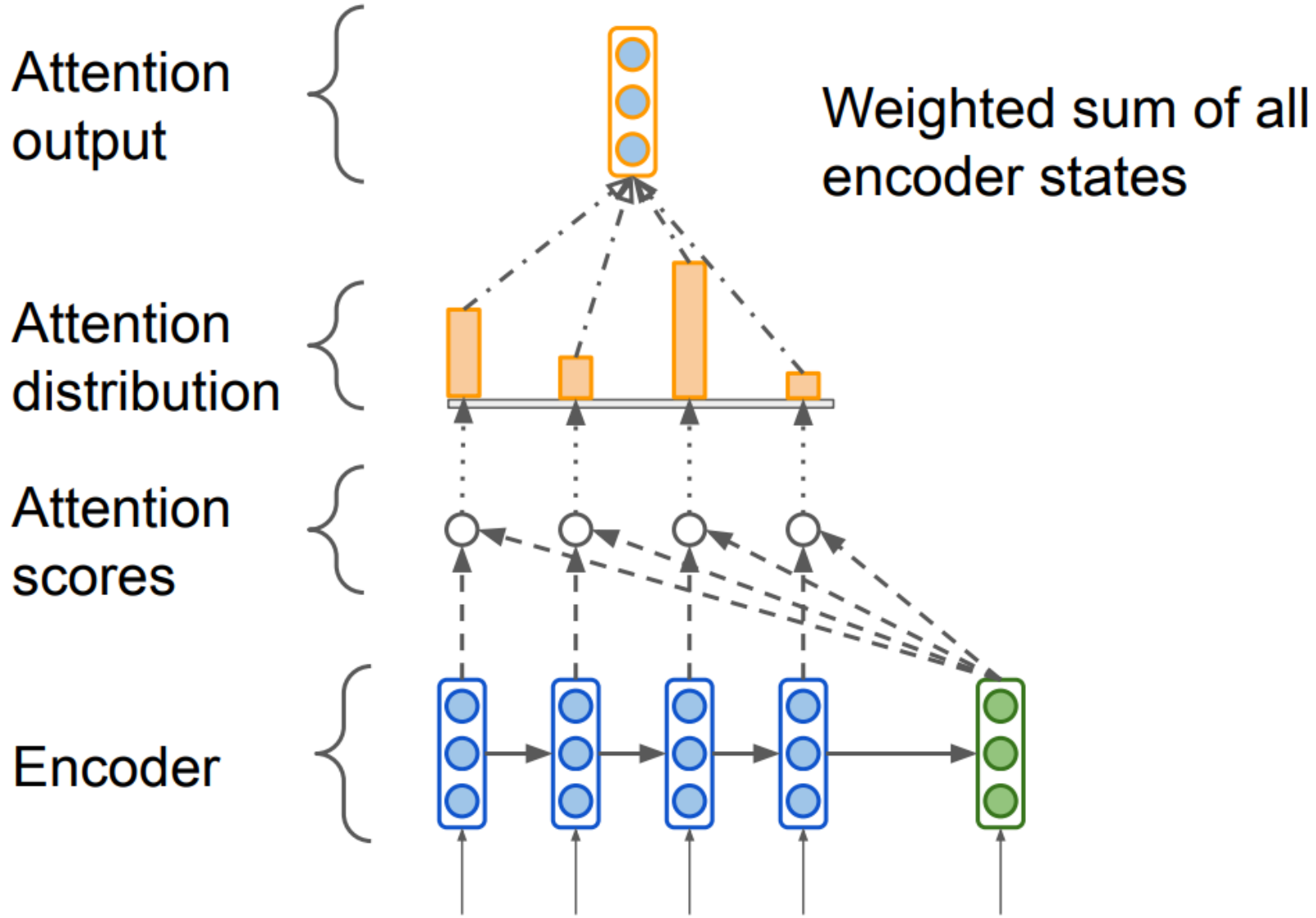
https://github.com/girafe-ai

Attention



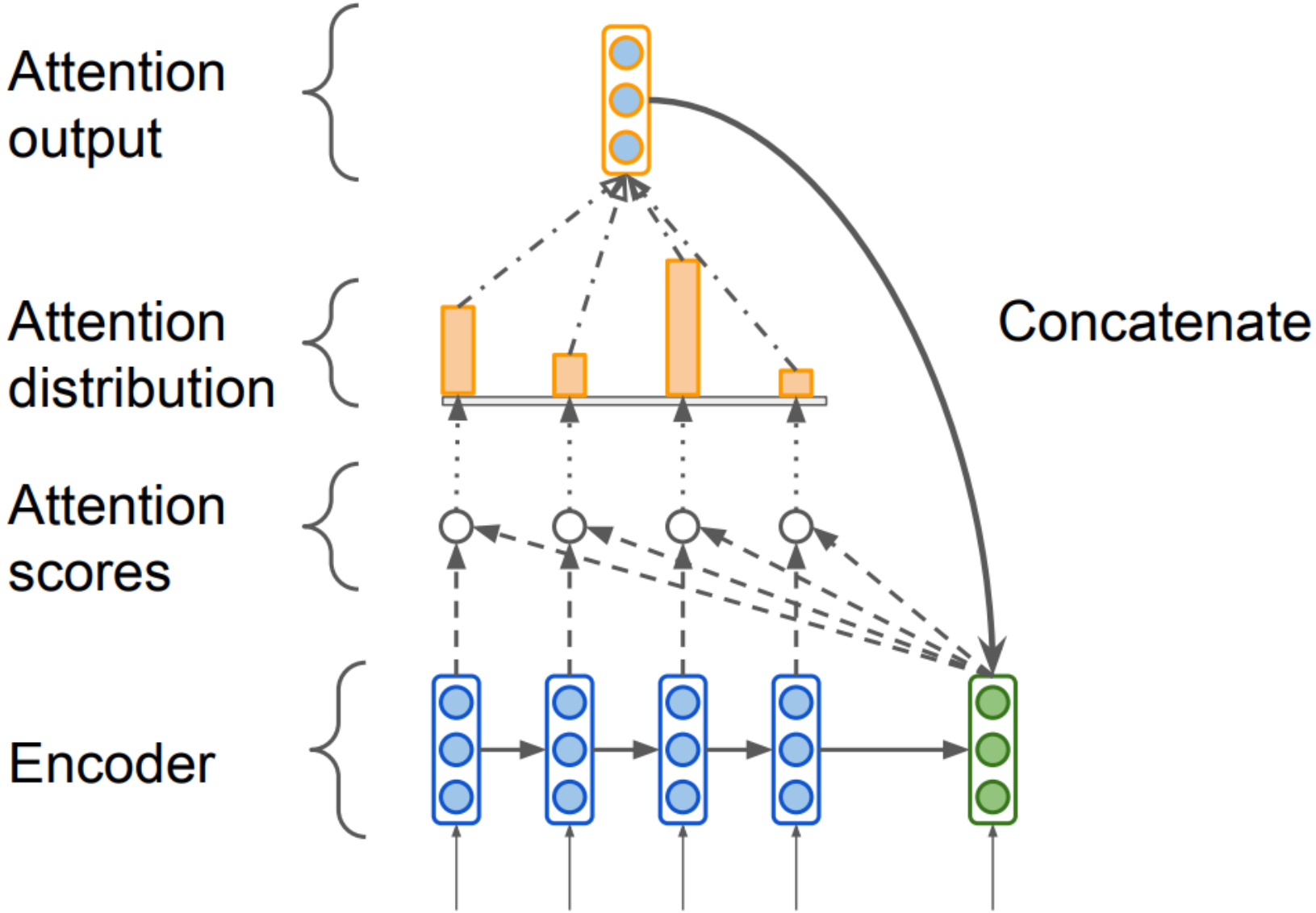
https://github.com/girafe-ai

Attention



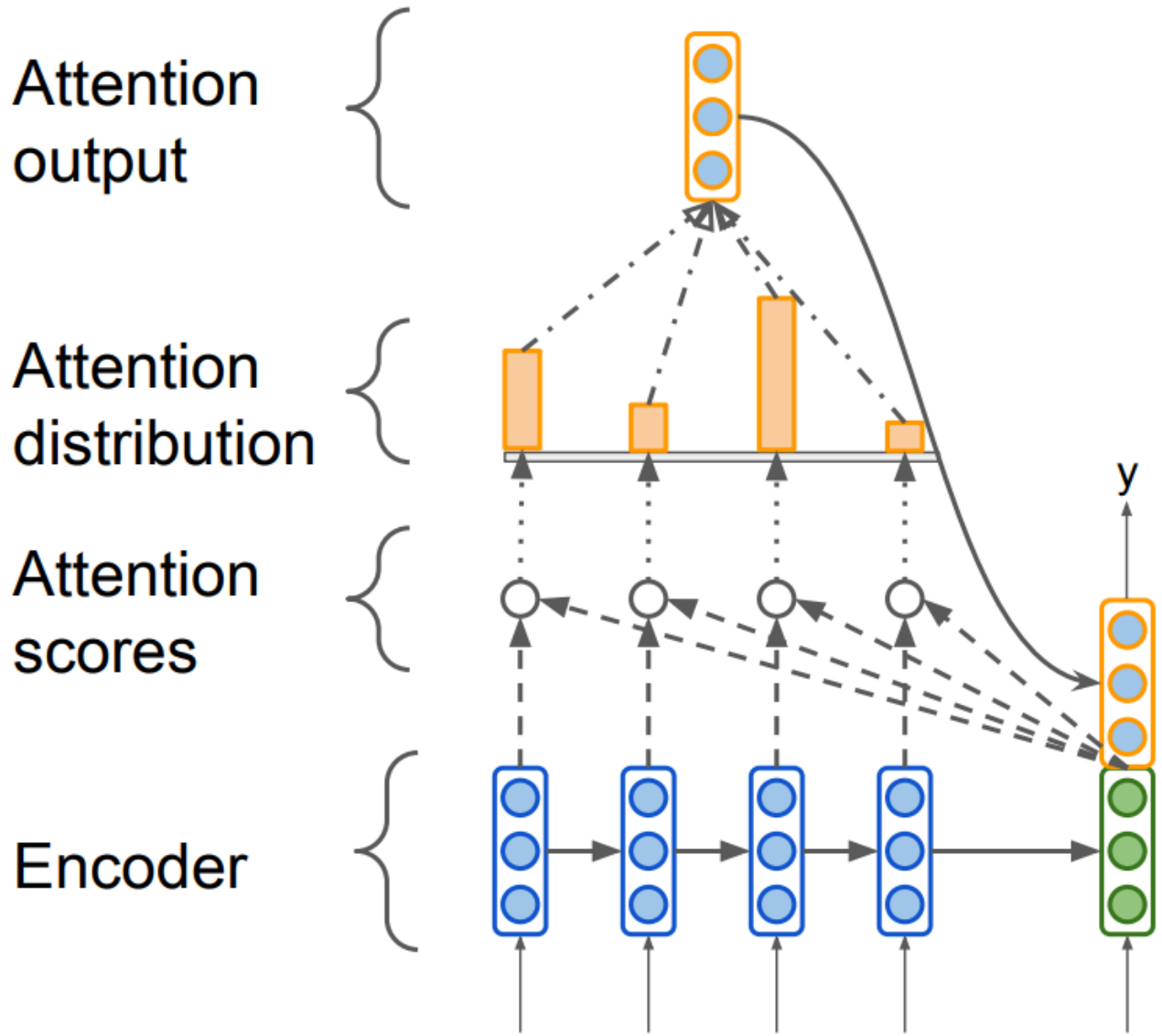
https://github.com/girafe-ai

Attention



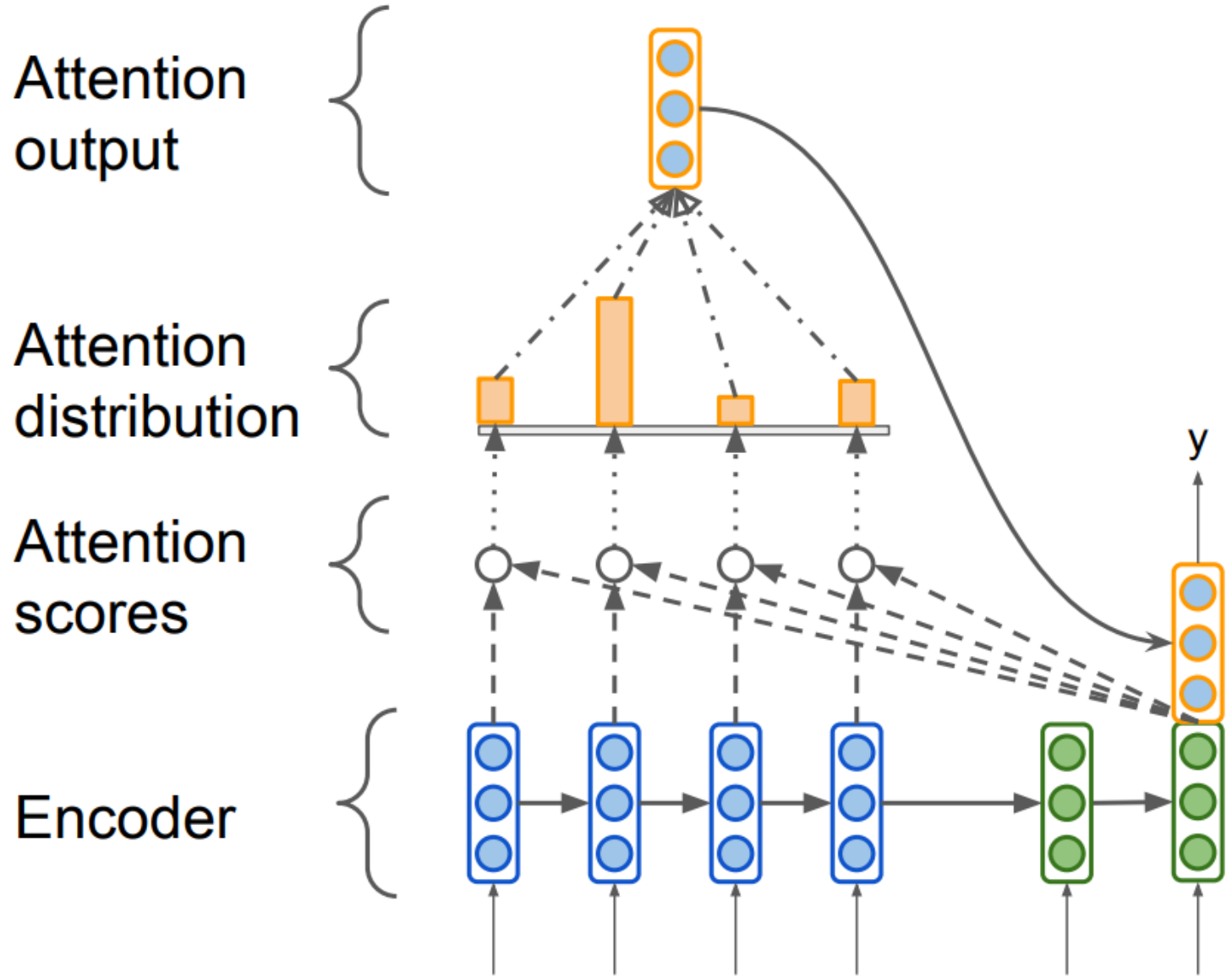
https://github.com/girafe-ai

Attention



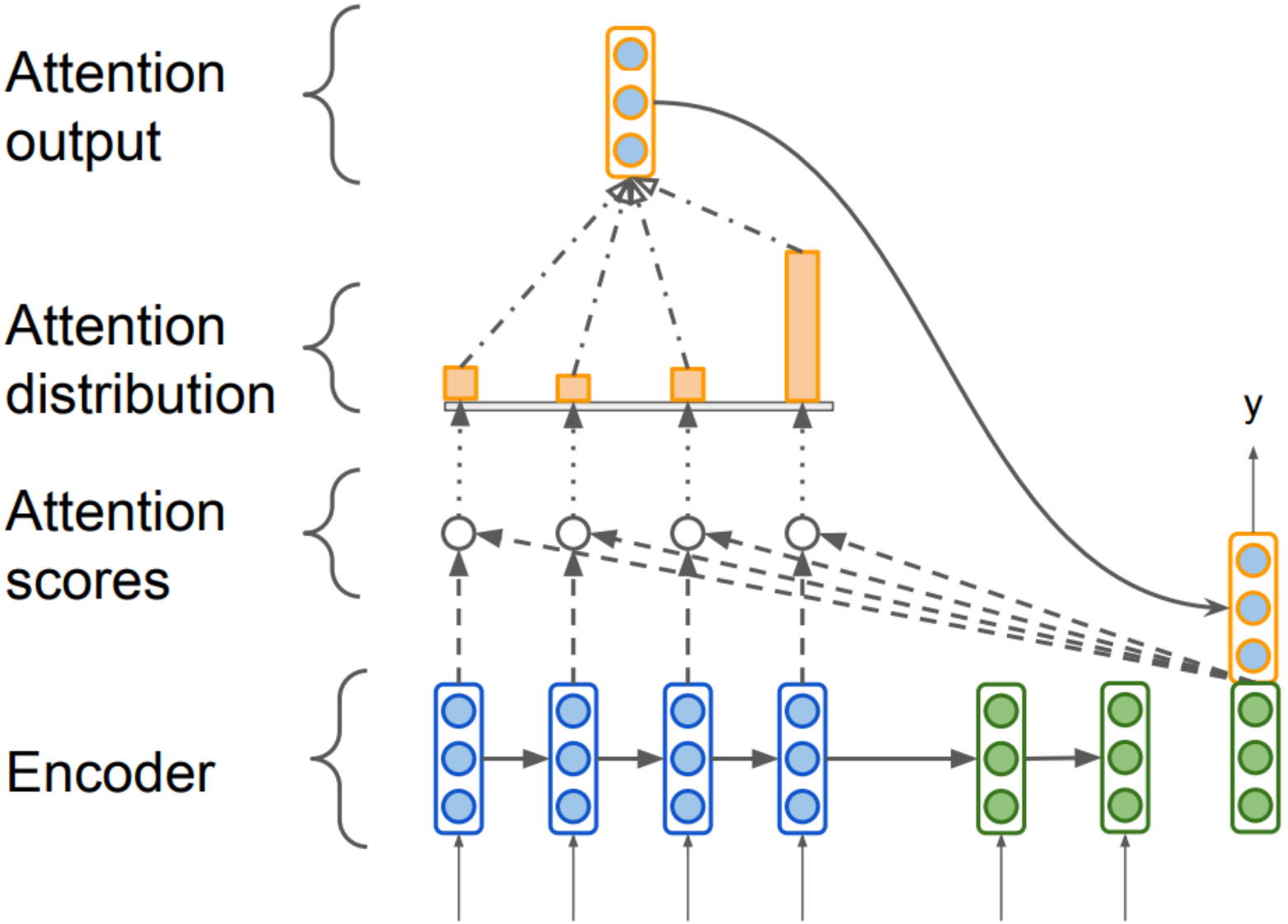
https://github.com/girafe-ai

Attention



https://github.com/girafe-ai

Attention



https://github.com/girafe-ai

Attention in formulas

Denote encoder hidden states $\mathbf{h}_1, \ldots, \mathbf{h}_N \in \mathbb{R}^k$

and decoder hidden state at time step t $\mathbf{s}_t \in \mathbb{R}^k$

The attention scores $\mathbf{e}^t$ can be computed as dot product

$$\mathbf{e}^t = [\mathbf{s}^T\mathbf{h}_1, \ldots, \mathbf{s}^T\mathbf{h}_N]$$

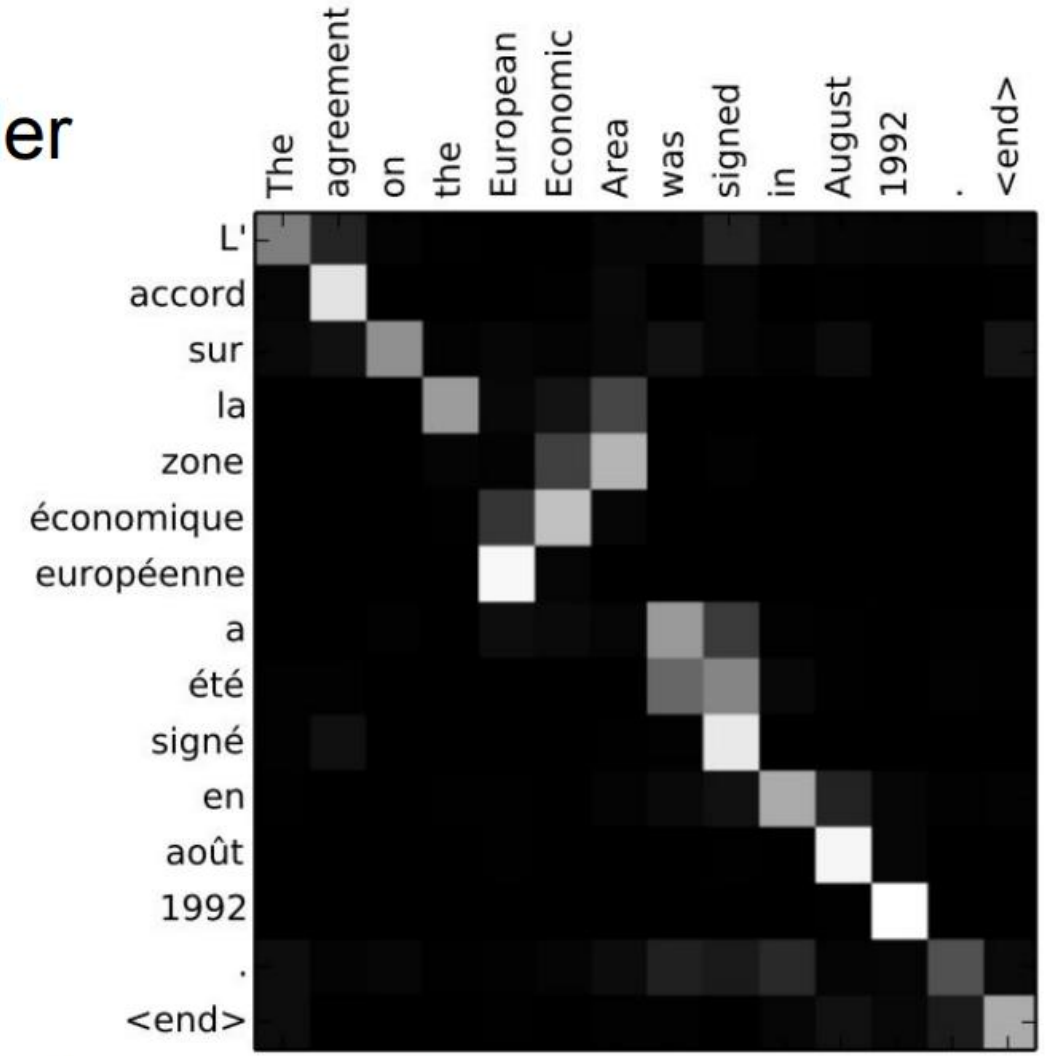
Then the attention vector is a linear combination of encoder states

$$\mathbf{a}_t = \sum_{i=1}^{N} \boldsymbol{\alpha}_i^t \mathbf{h}_i \in \mathbb{R}^k \text{, where } \boldsymbol{\alpha}_t = \mathrm{softmax}(\mathbf{e}_t)$$

https://github.com/girafe-ai

Attention alignment

- We may see what the decoder was focusing on
- We get word alignment for free!



Bahdanau et al. "Neural Machine Translation by Jointly Learning to Align and Translate", 2014

https://github.com/girafe-ai

Attention types

- Basic dot-product (the one discussed before): $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{h}_i \in \mathbb{R}$
- Multiplicative attention: $\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{W} \boldsymbol{h}_i \in \mathbb{R}$
  - $\boldsymbol{W} \in \mathbb{R}^{d_2 \times d_1}$ - weight matrix
- Additive attention: $\boldsymbol{e}_i = \boldsymbol{v}^T \tanh(\boldsymbol{W}_1 \boldsymbol{h}_i + \boldsymbol{W}_2 \boldsymbol{s}) \in \mathbb{R}$
  - $\boldsymbol{W}_1 \in \mathbb{R}^{d_3 \times d_1}, \boldsymbol{W}_2 \in \mathbb{R}^{d_3 \times d_2}$ - weight matrices
  - $\boldsymbol{v} \in \mathbb{R}^{d_3}$ - weight vector

Attention demonstration

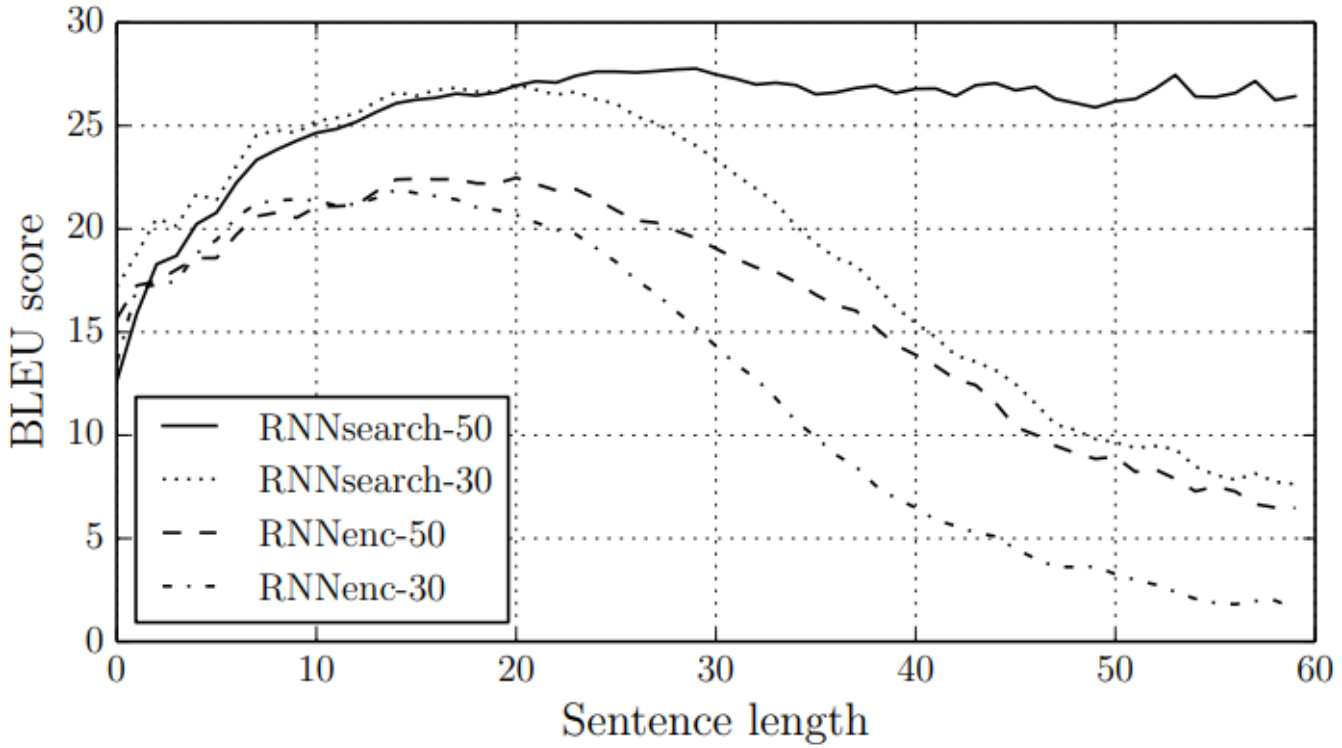
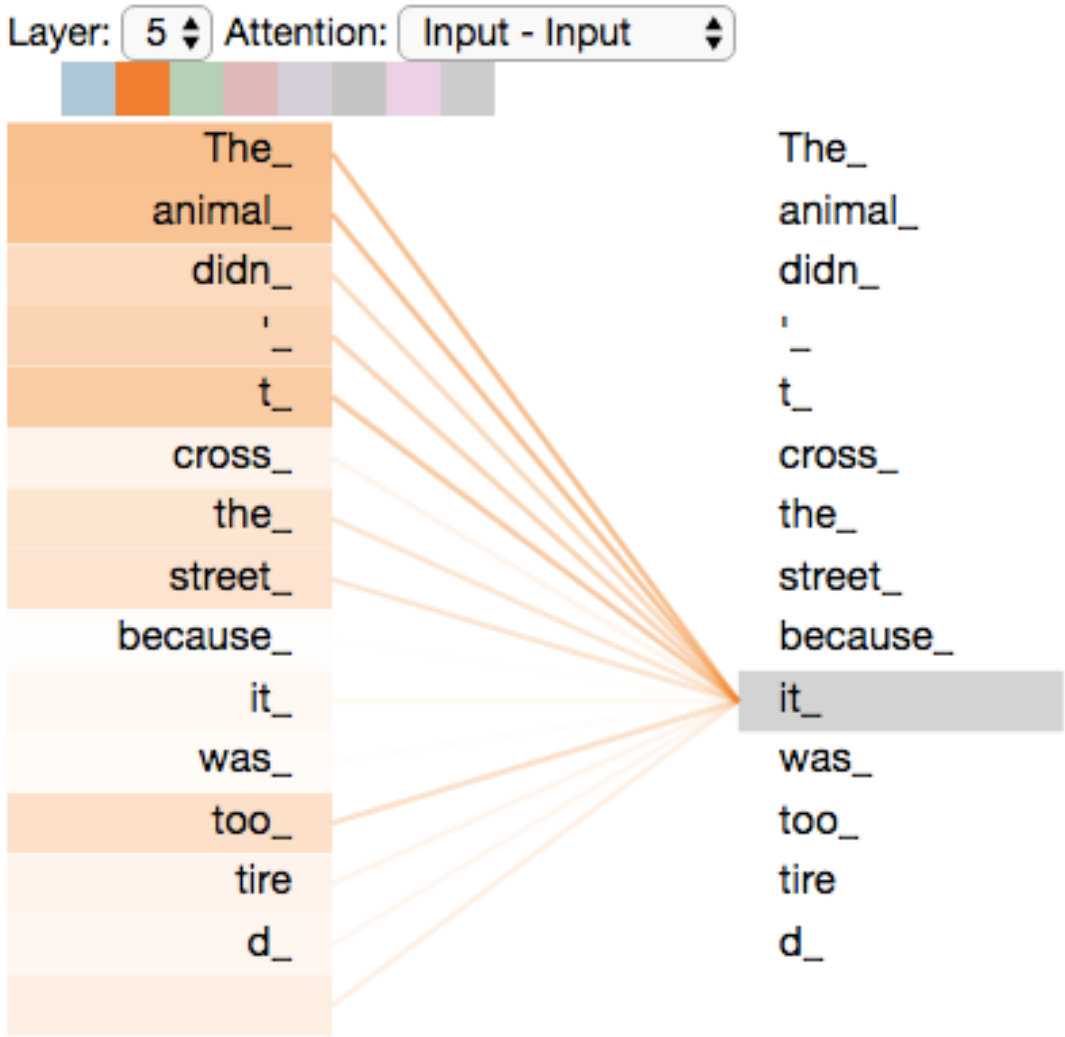


Figure 2: The BLEU scores of the generated translations on the test set with respect to the lengths of the sentences. The results are on the full test set which includes sentences having unknown words to the models.

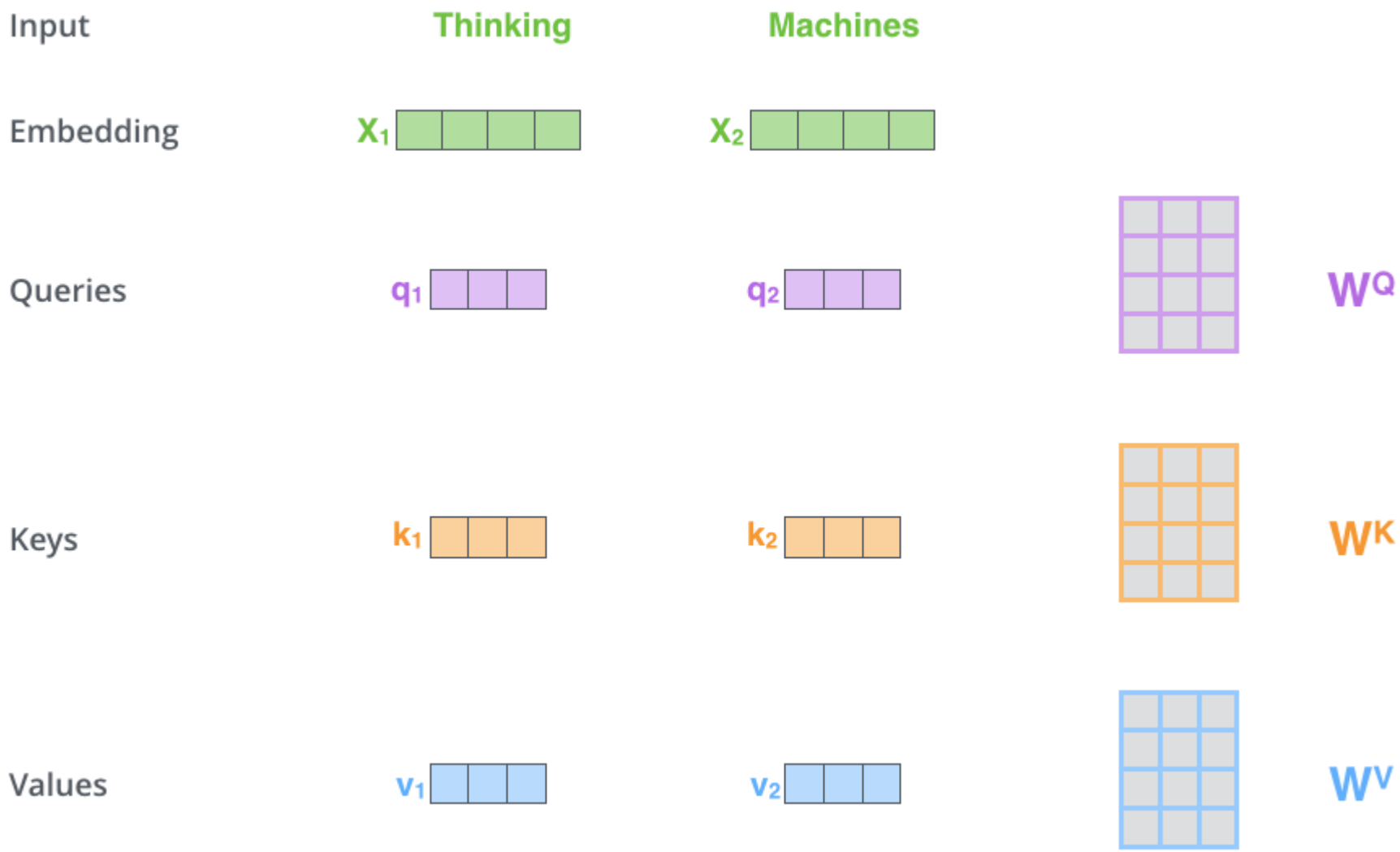
https://arxiv.org/pdf/1409.0473

Self attention



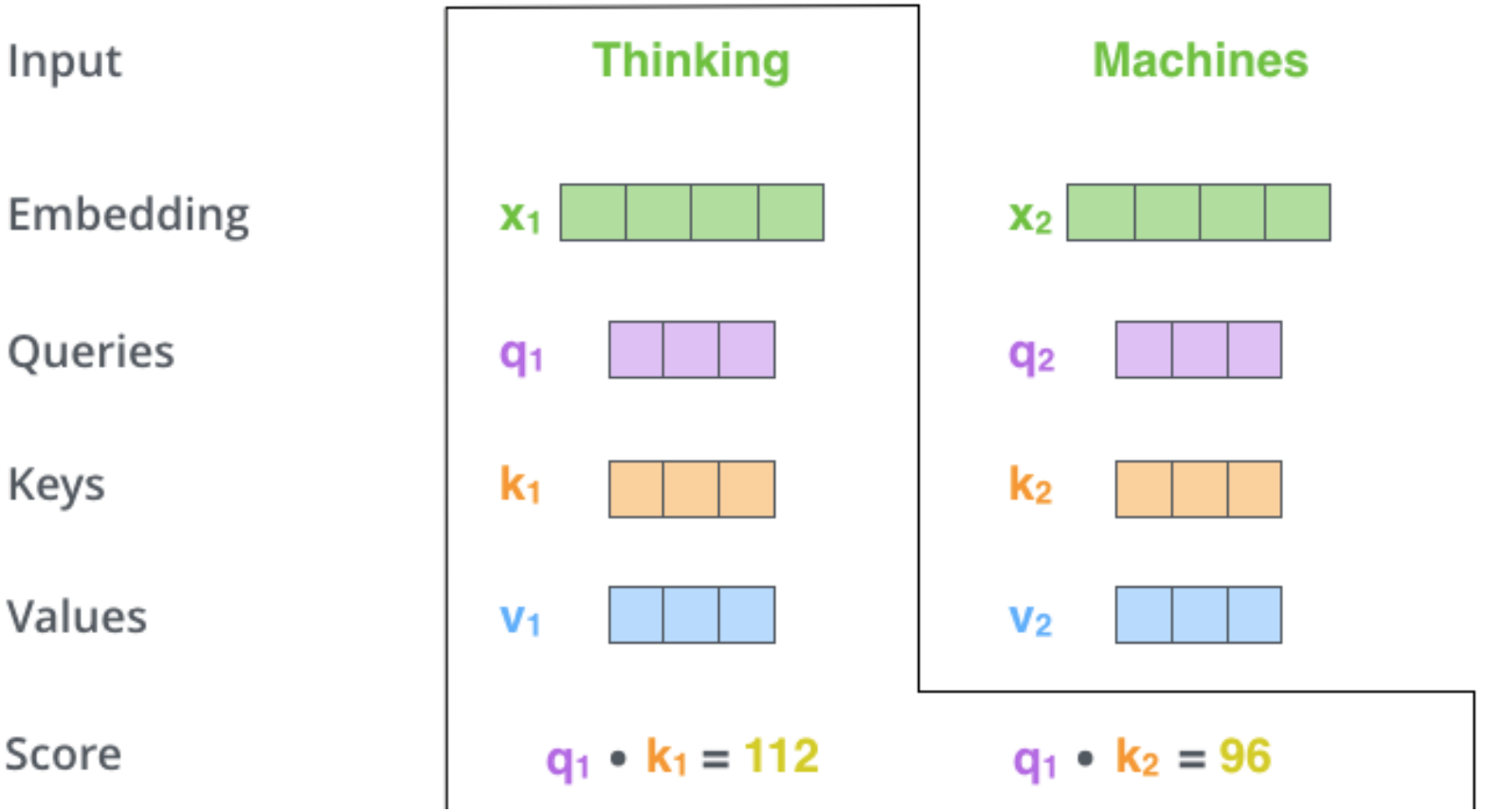
https://jalammar.github.io/illustrated-transformer/

Self attention



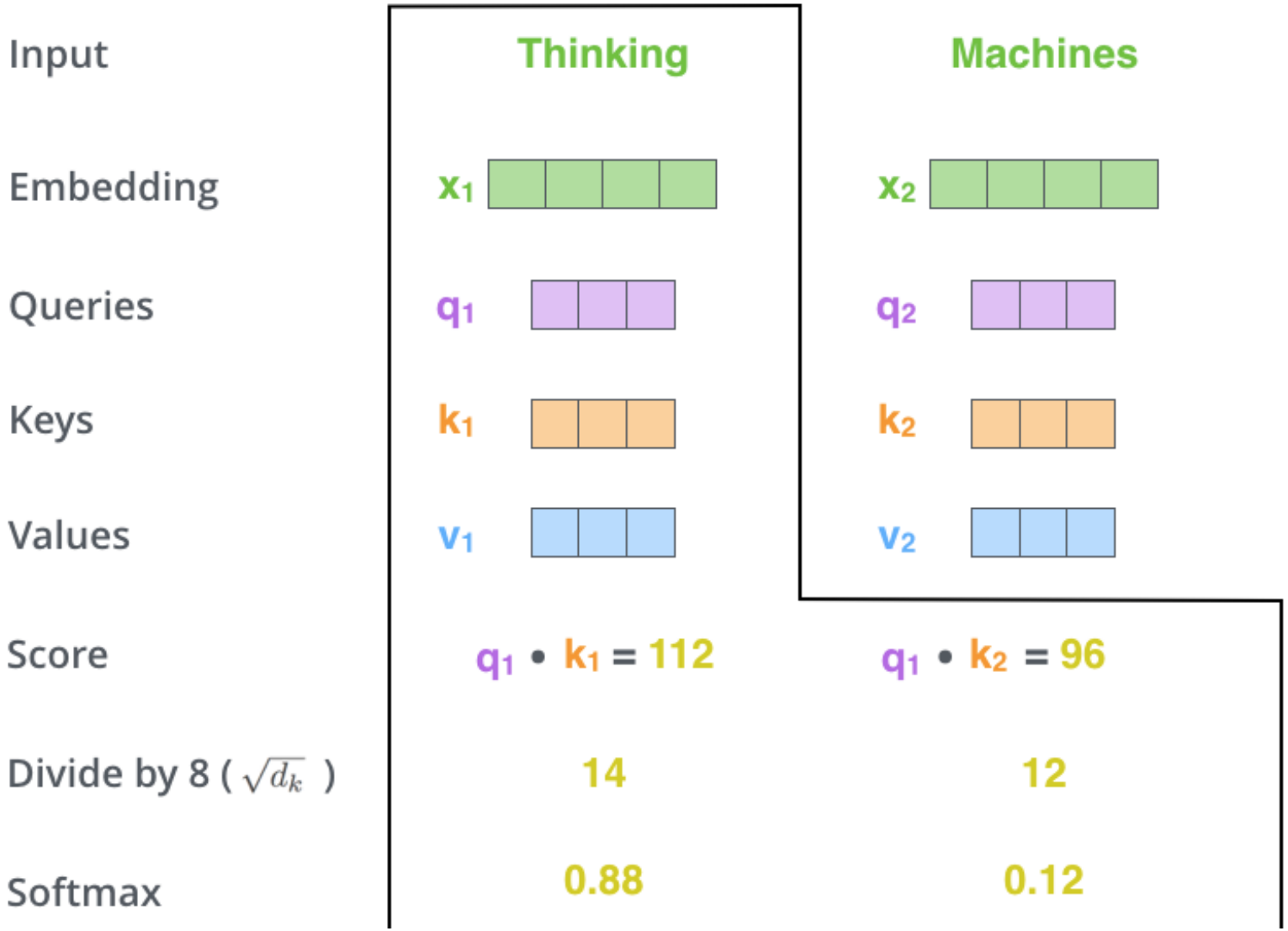
https://jalammar.github.io/illustrated-transformer/

Self attention



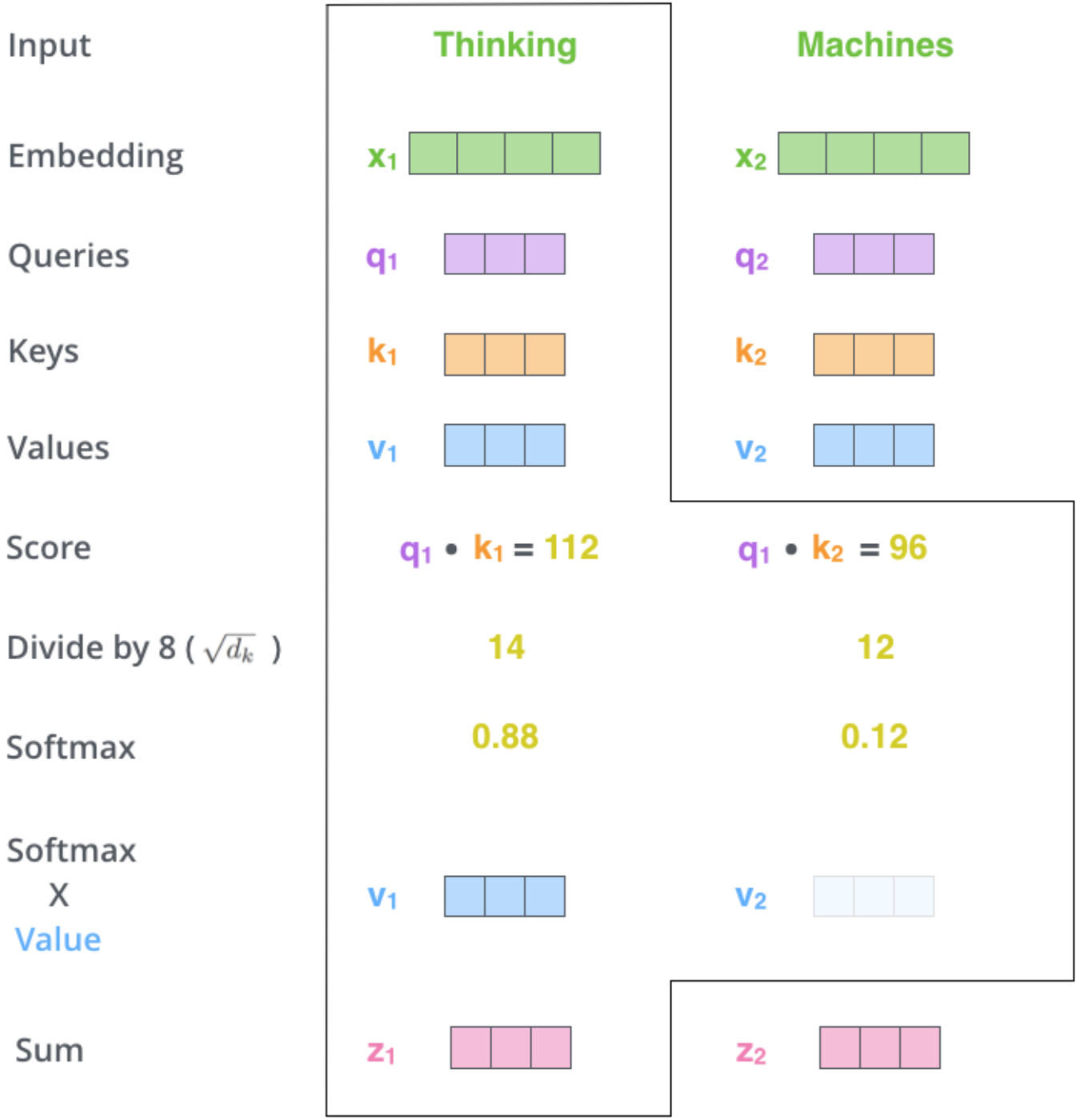
https://jalammar.github.io/illustrated-transformer/

Self attention



https://jalammar.github.io/illustrated-transformer/

Self attention



STEP 1: create Query, Key, Value

STEP 2: calculate scores

STEP 3: divide by $\sqrt{d_k}$

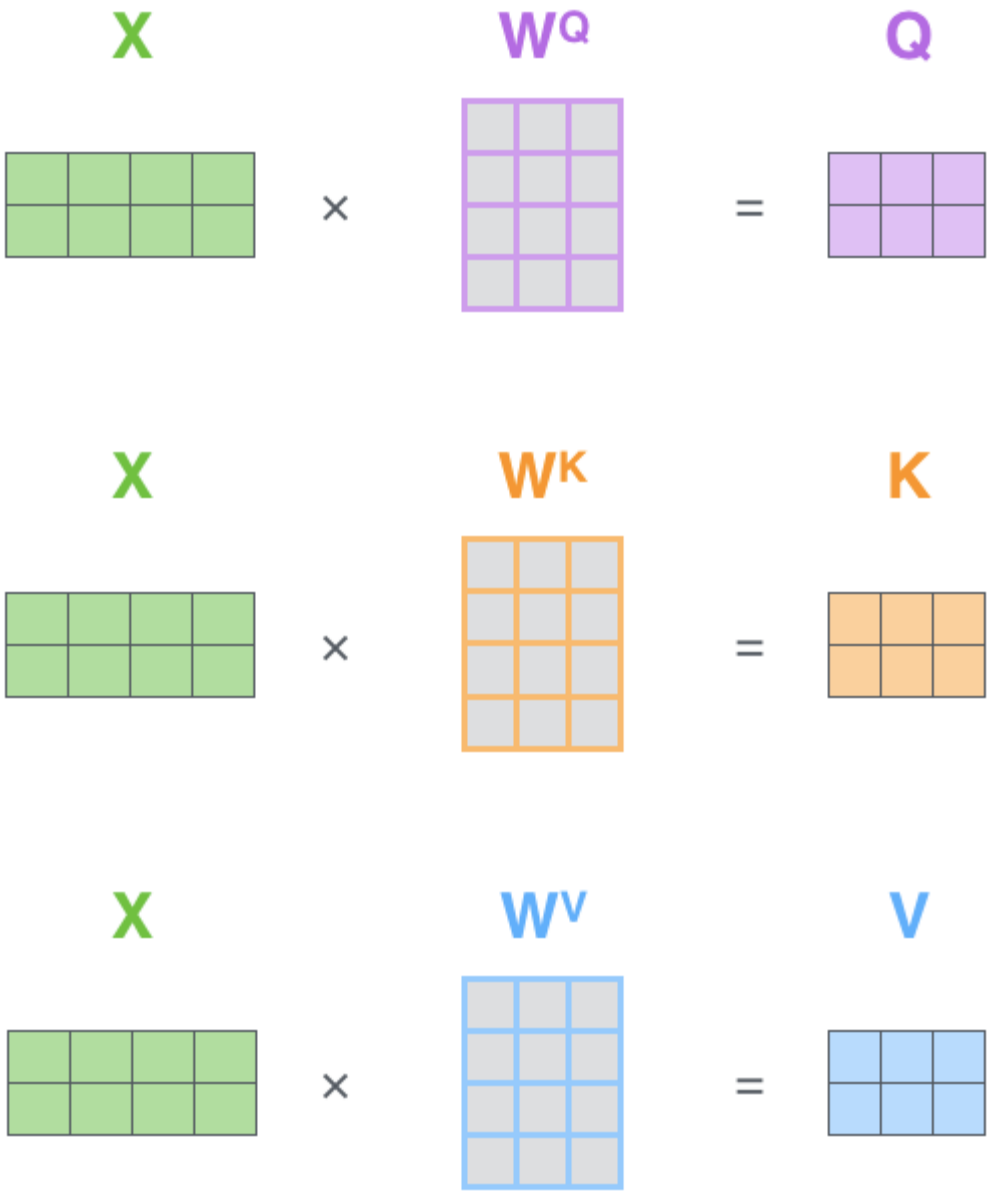
STEP 4: softmax

STEP 5: multiply each value vector by the softmax score

STEP 6: sum up the weighted value vectors

https://jalammar.github.io/illustrated-transformer/

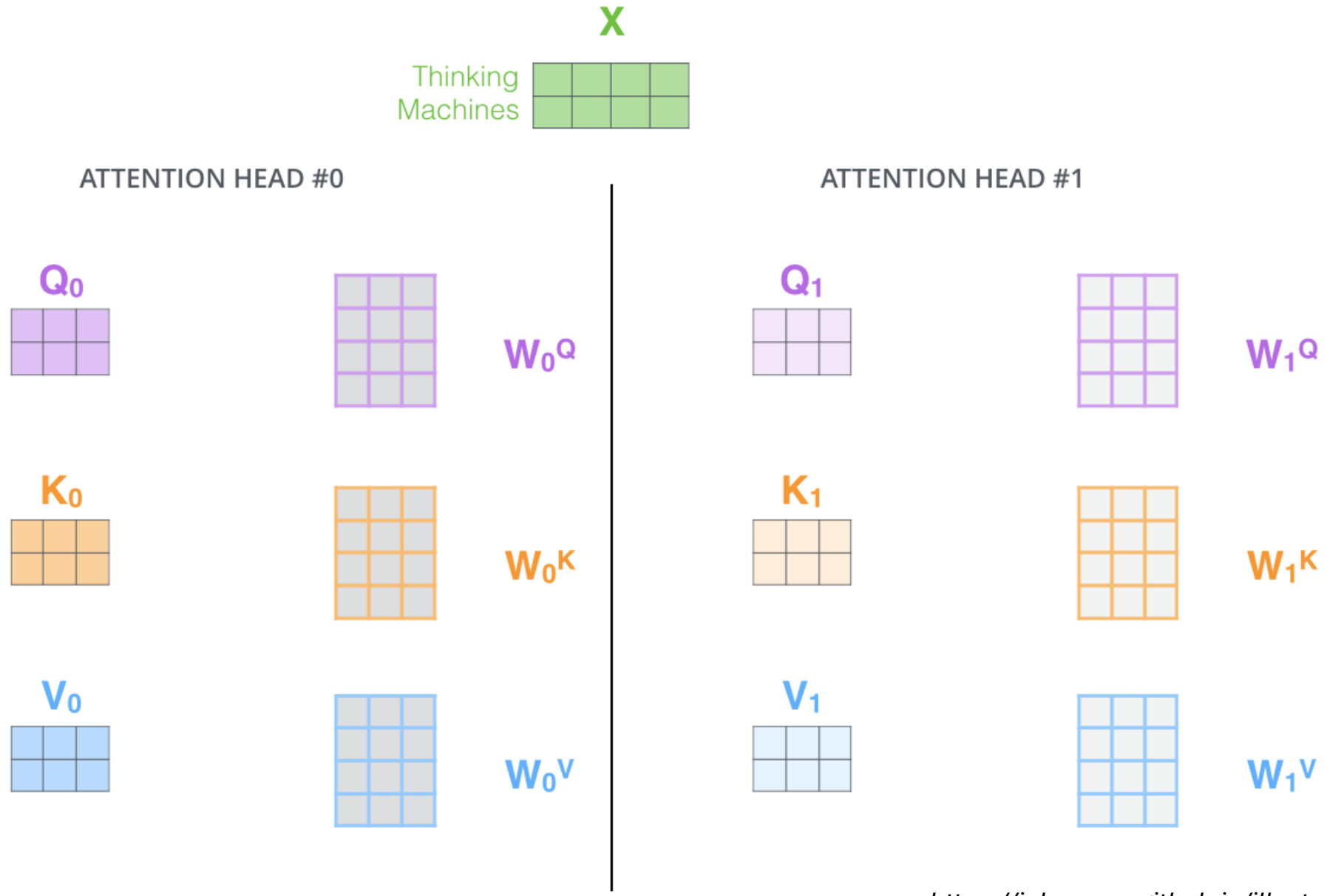
Self attention



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

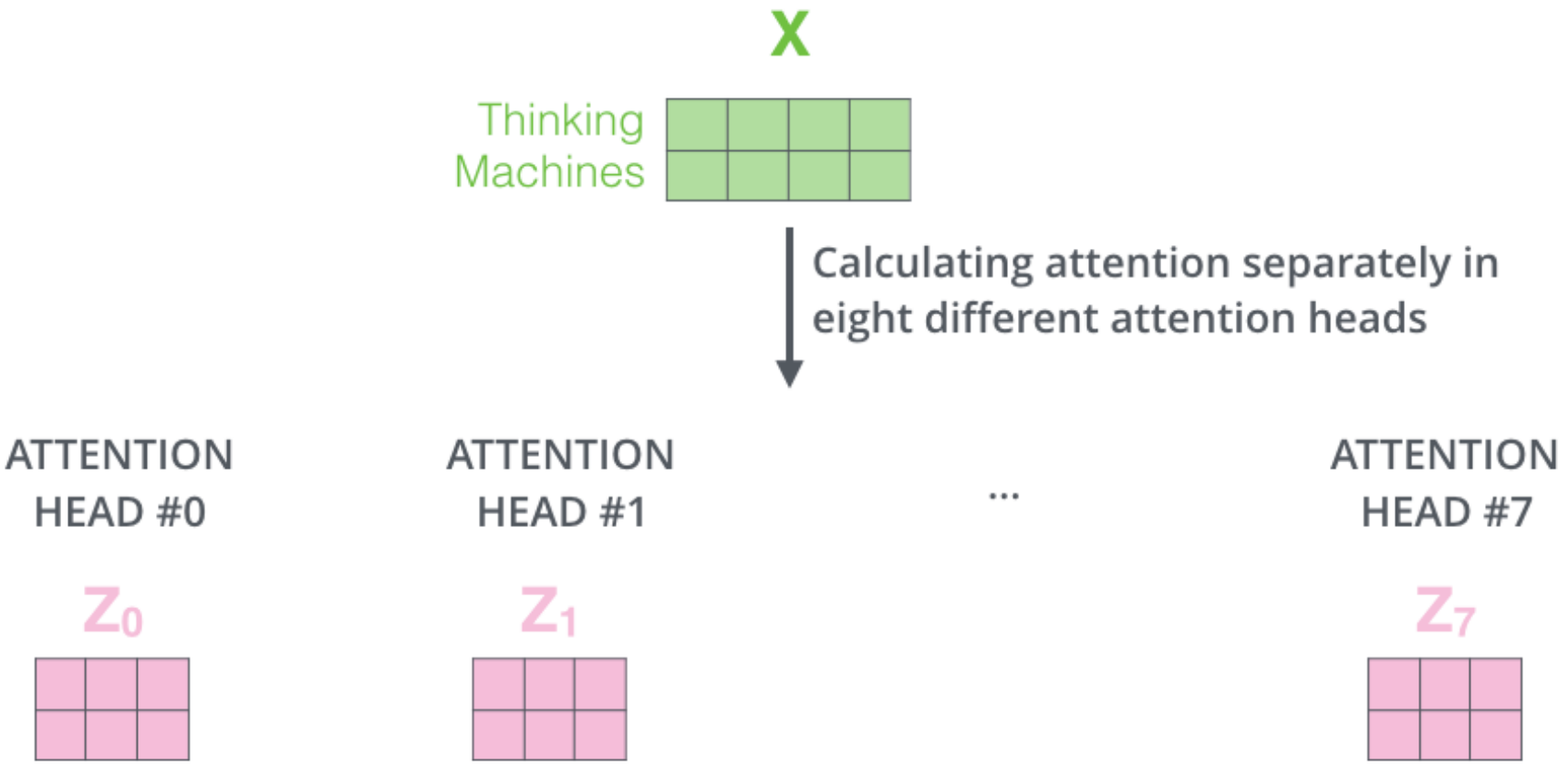
https://jalammar.github.io/illustrated-transformer/

Self attention



https://jalammar.github.io/illustrated-transformer/

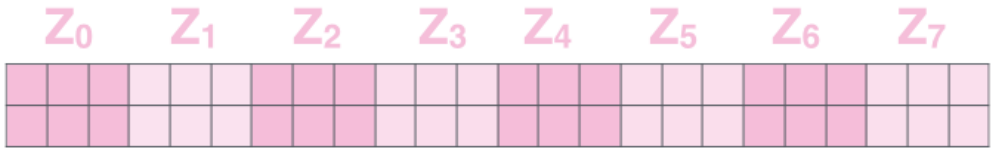
Self attention



https://jalammar.github.io/illustrated-transformer/

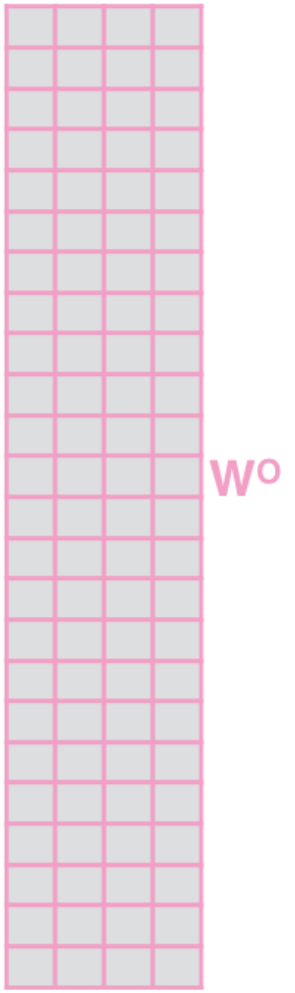
Self attention

1) Concatenate all the attention heads



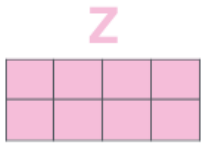
2) Multiply with a weight matrix W^O that was trained jointly with the model

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

=



https://jalammar.github.io/illustrated-transformer/

Self attention

1) This is our input sentence*

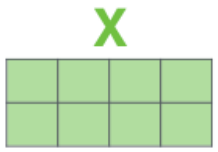
2) We embed each word*

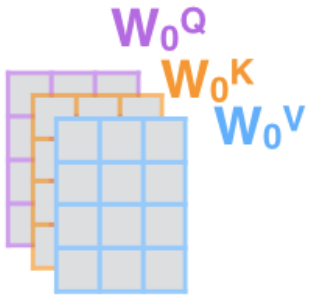
3) Split into 8 heads. We multiply X or R with weight matrices

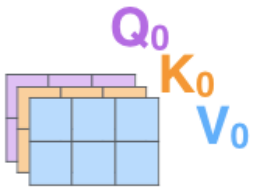
4) Calculate attention using the resulting Q/K/V matrices

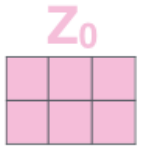
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

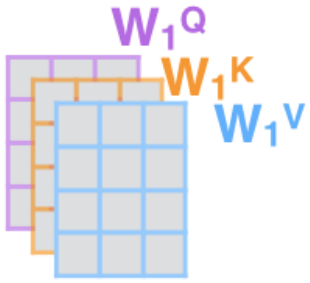
Thinking Machines

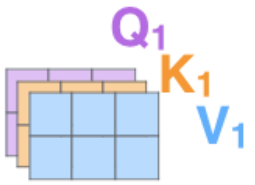


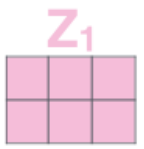


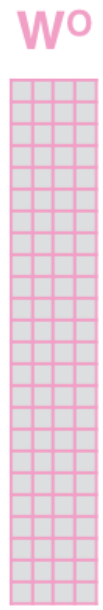


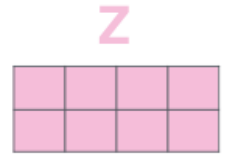




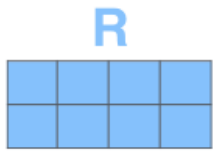




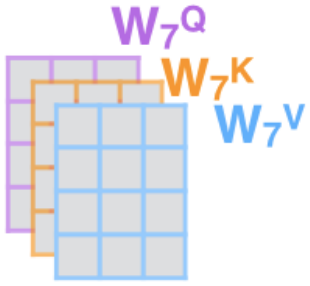




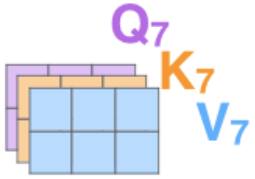
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



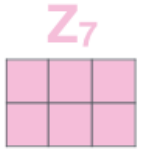
...



...

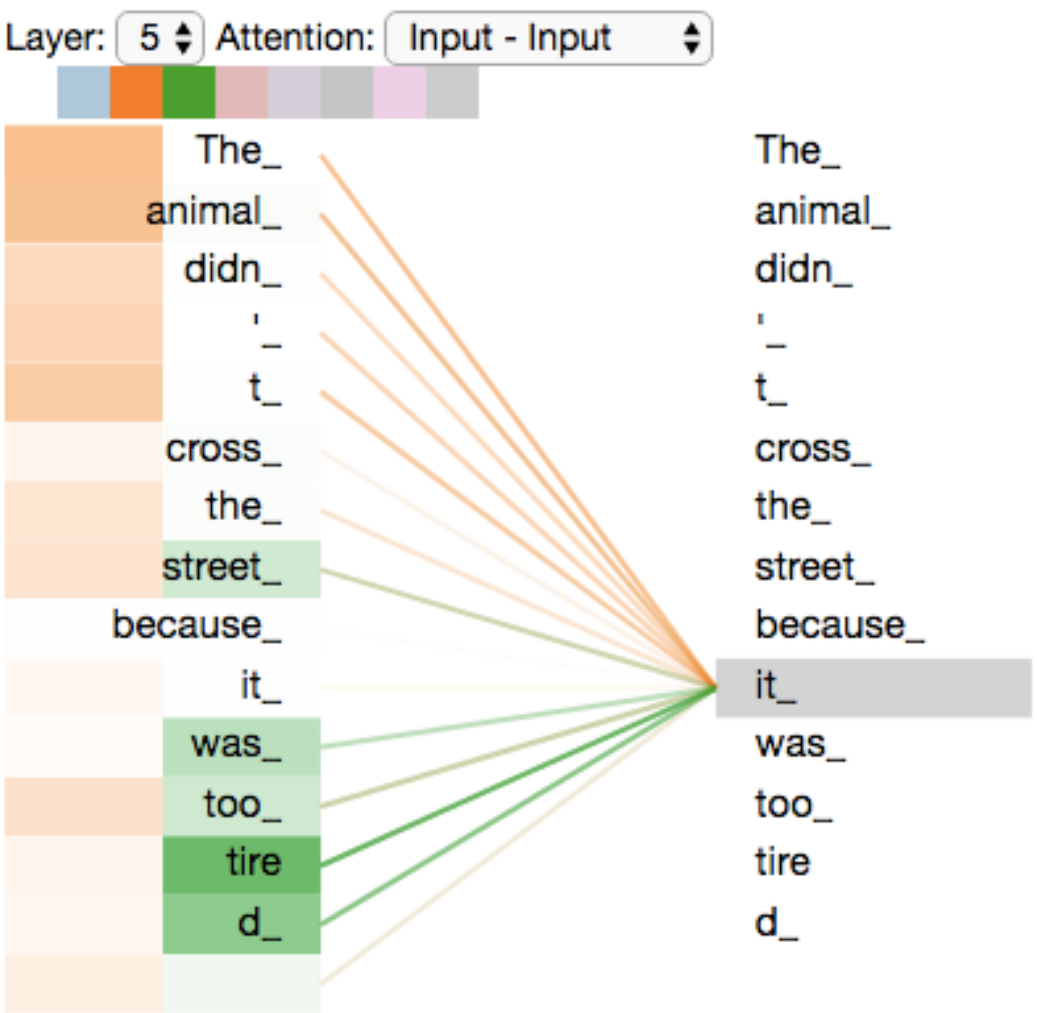


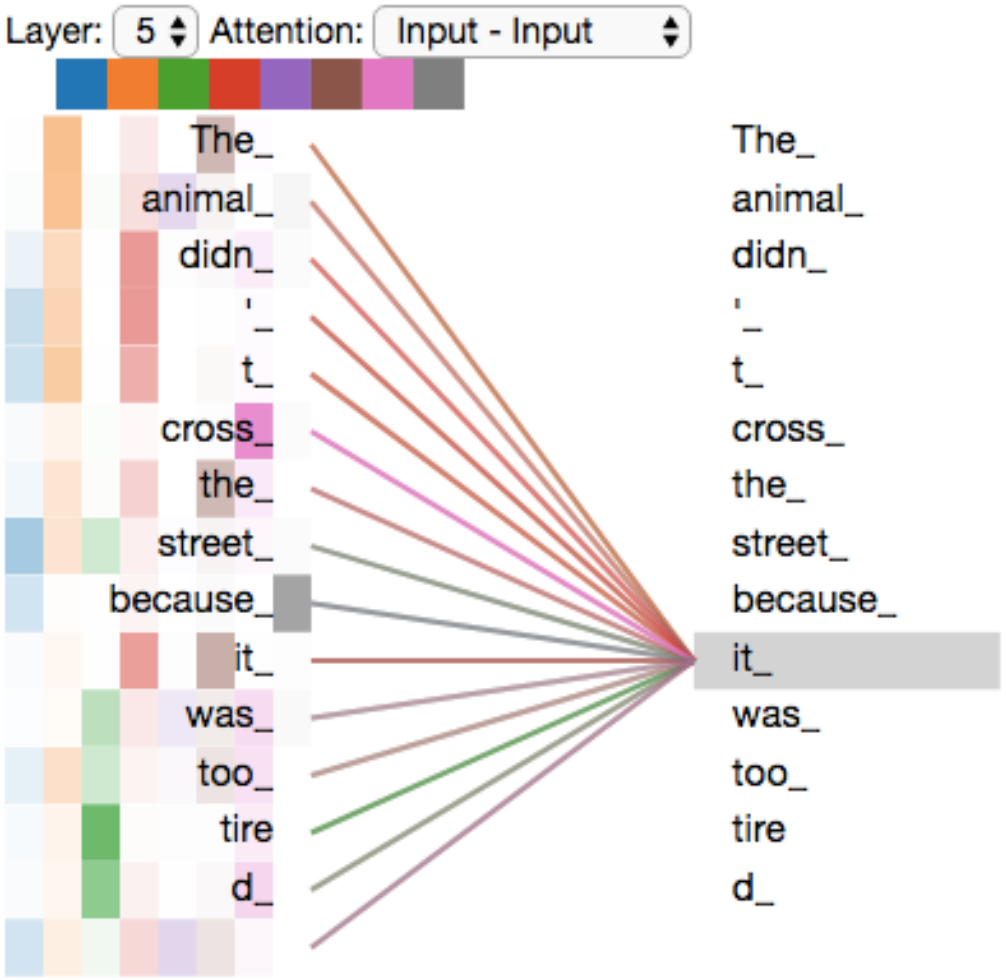
...



https://jalammar.github.io/illustrated-transformer/

Self attention



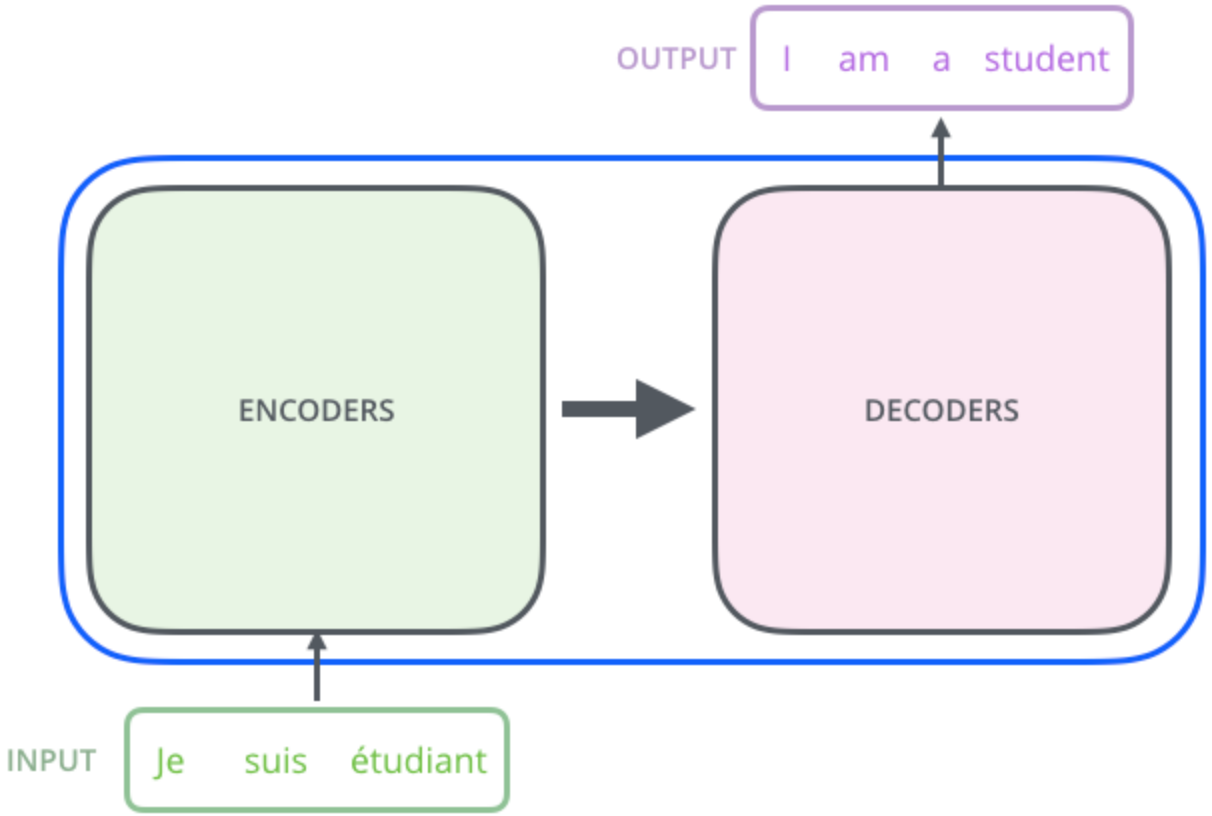


https://jalammar.github.io/illustrated-transformer/

The Transformer

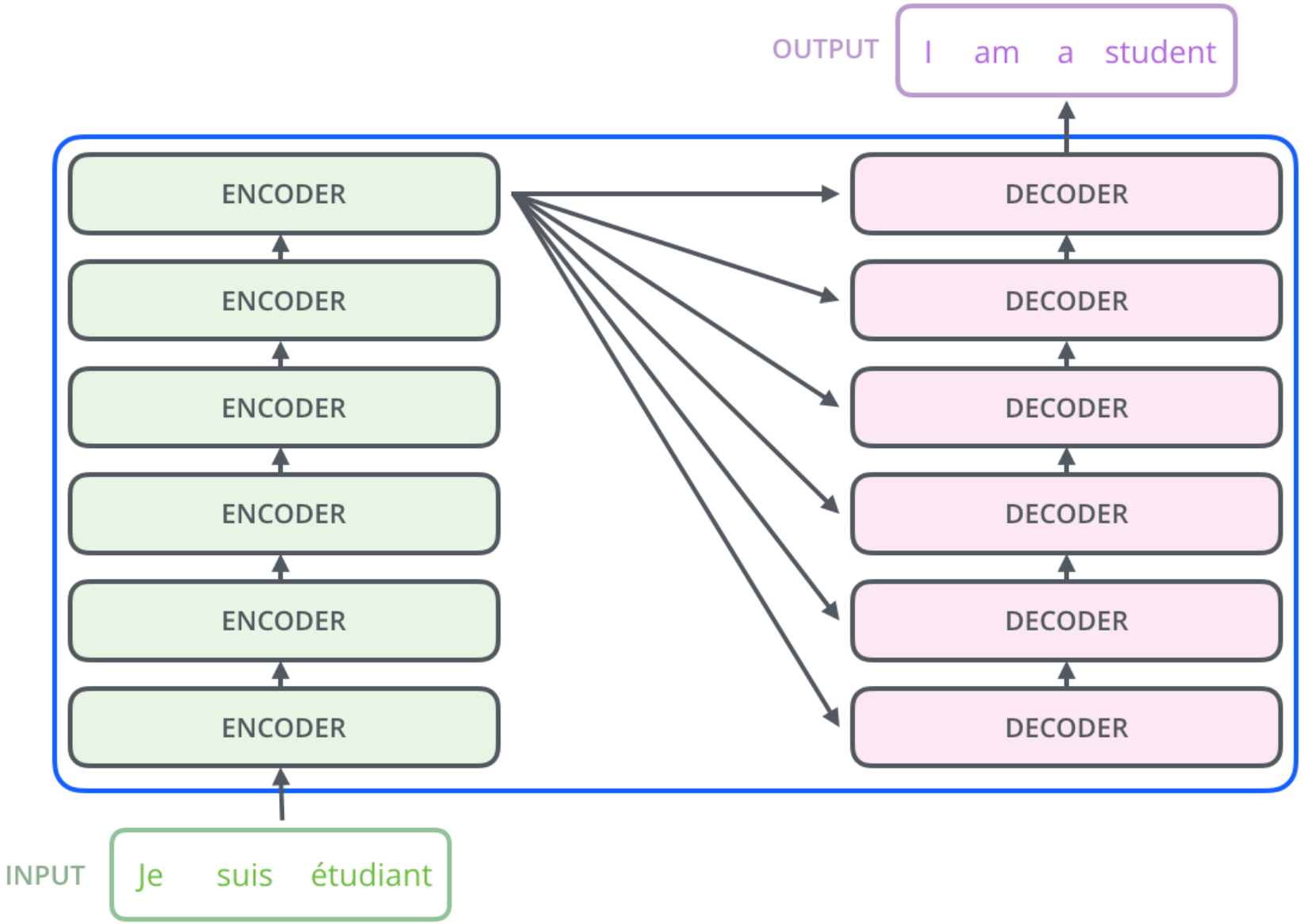
https://jalammar.github.io/illustrated-transformer/

The Transformer



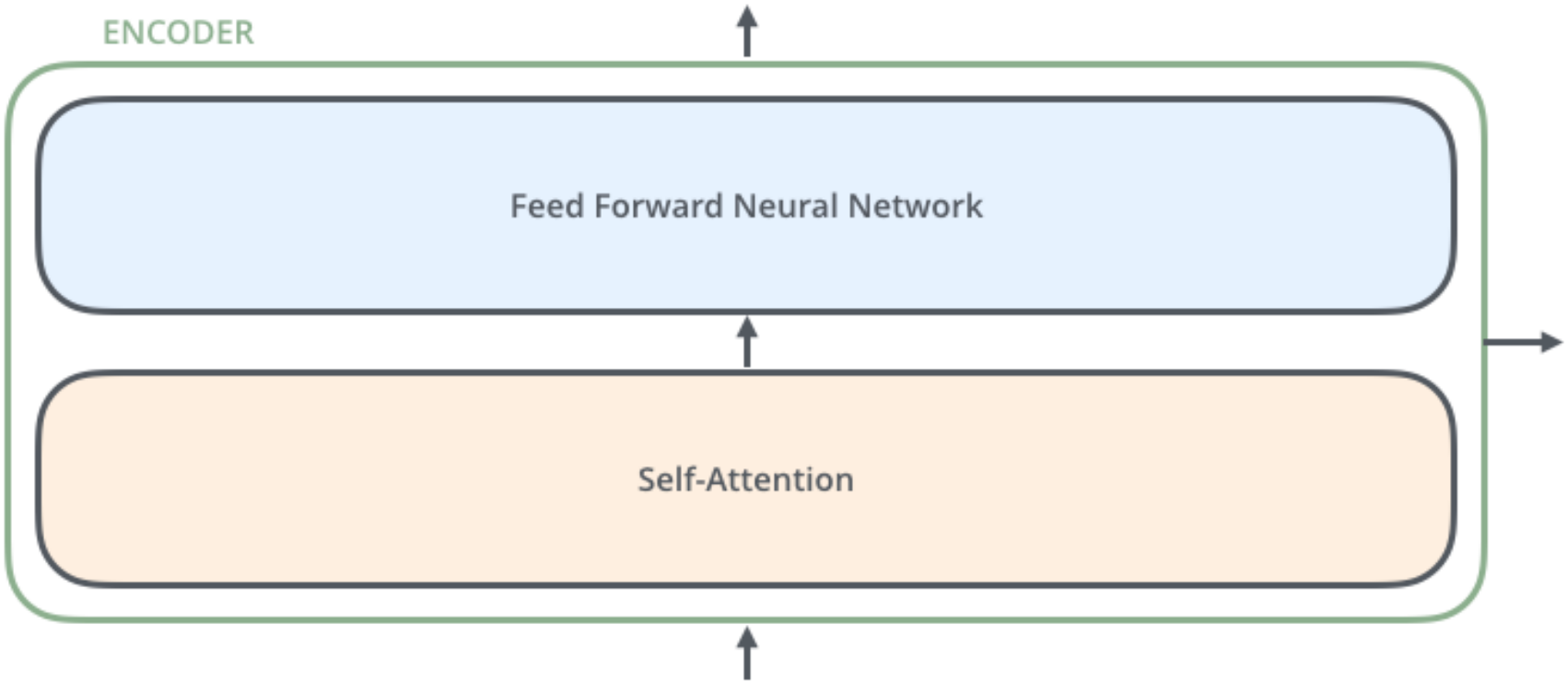
https://jalammar.github.io/illustrated-transformer/

The Transformer



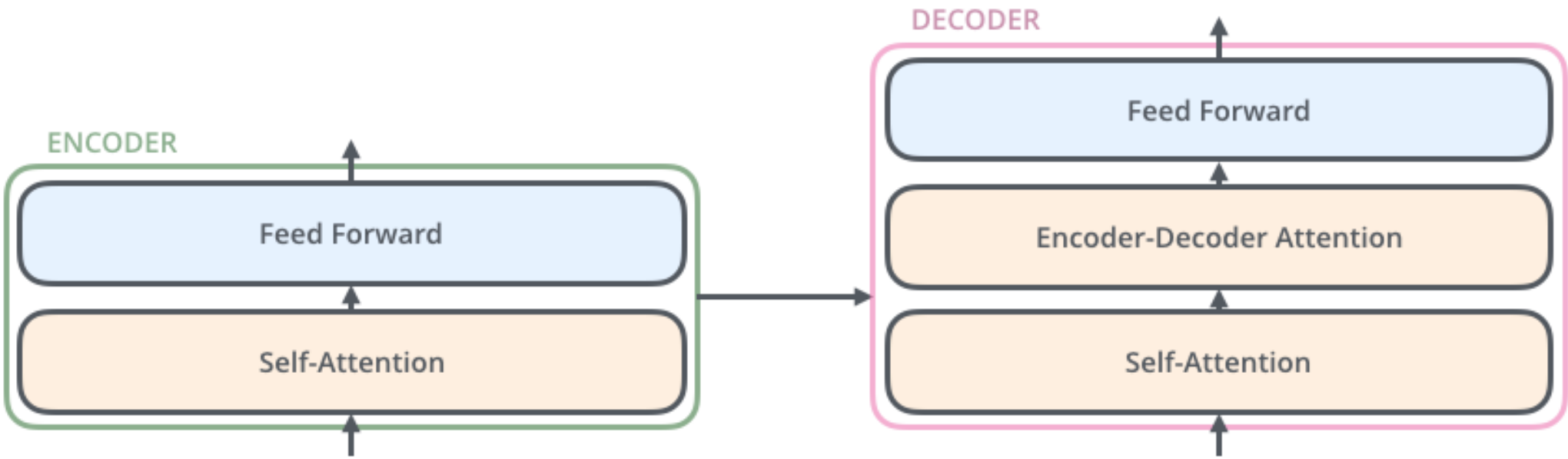
https://jalammar.github.io/illustrated-transformer/

The Transformer



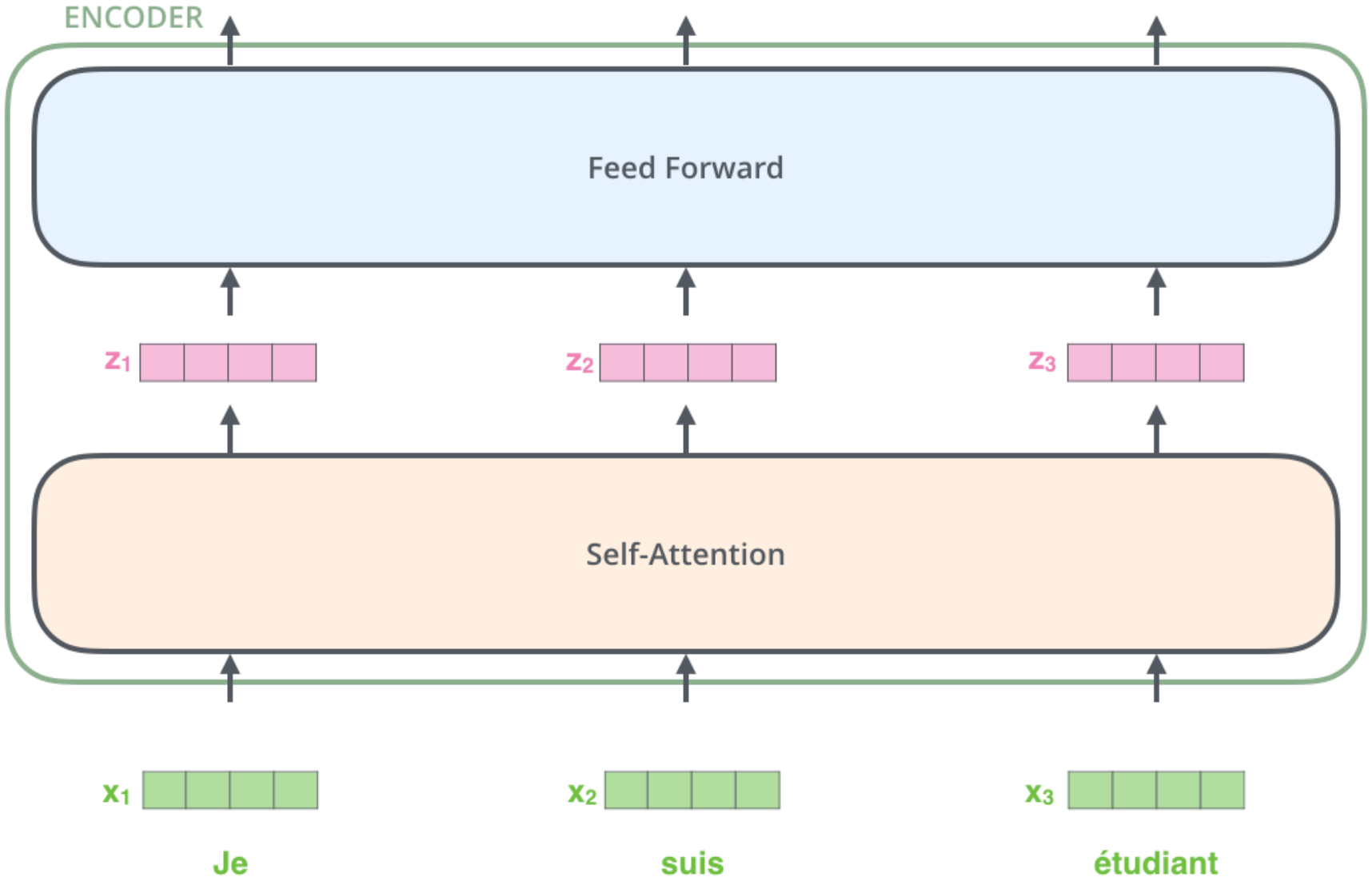
https://jalammar.github.io/illustrated-transformer/

The Transformer



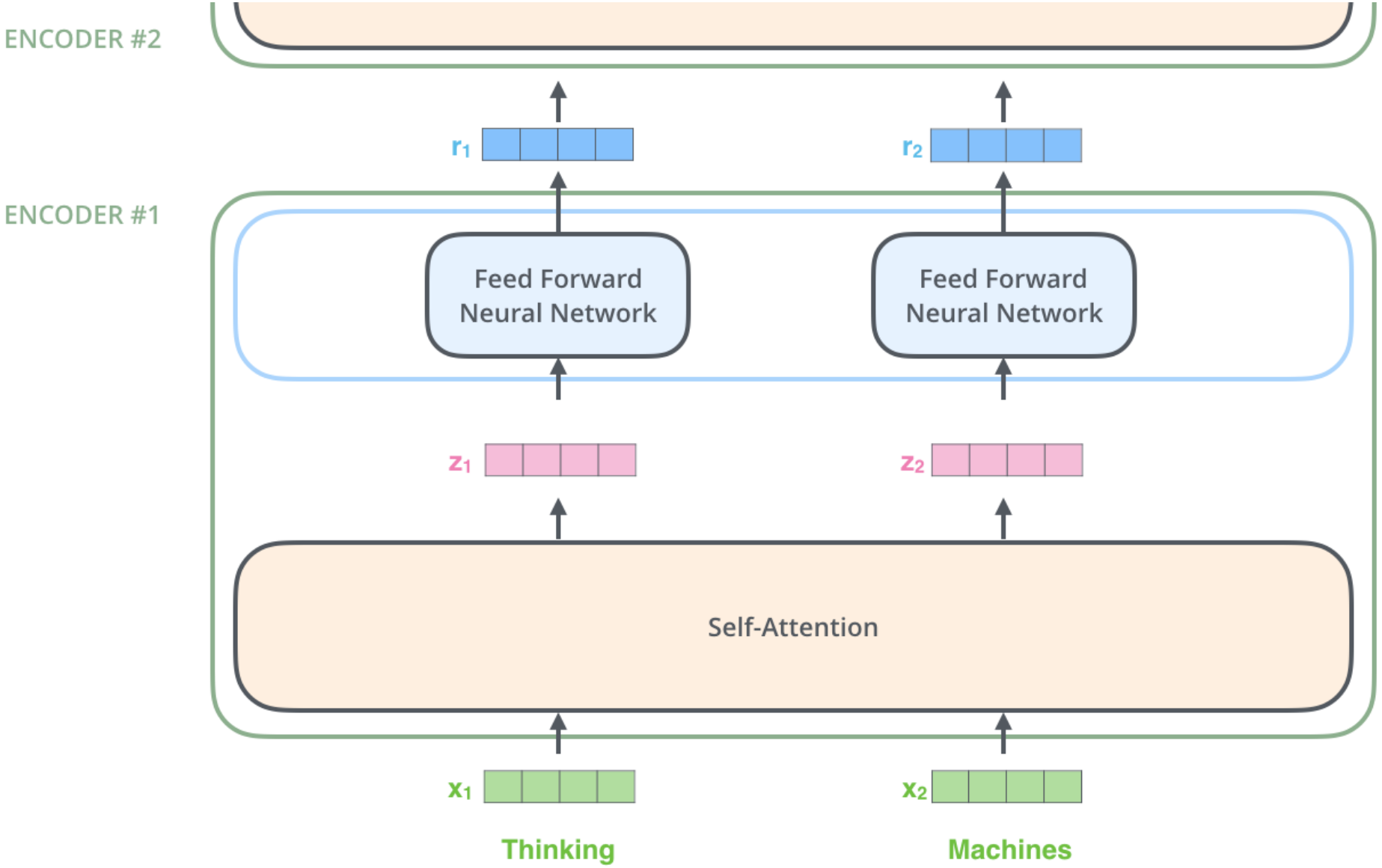
https://jalammar.github.io/illustrated-transformer/

The Transformer



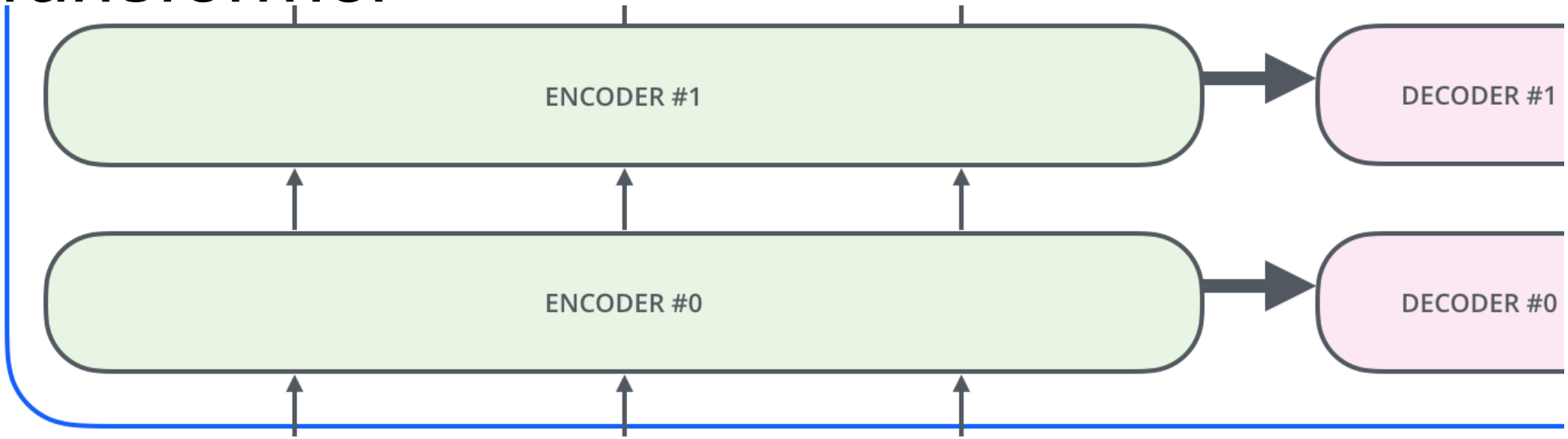
https://jalammar.github.io/illustrated-transformer/

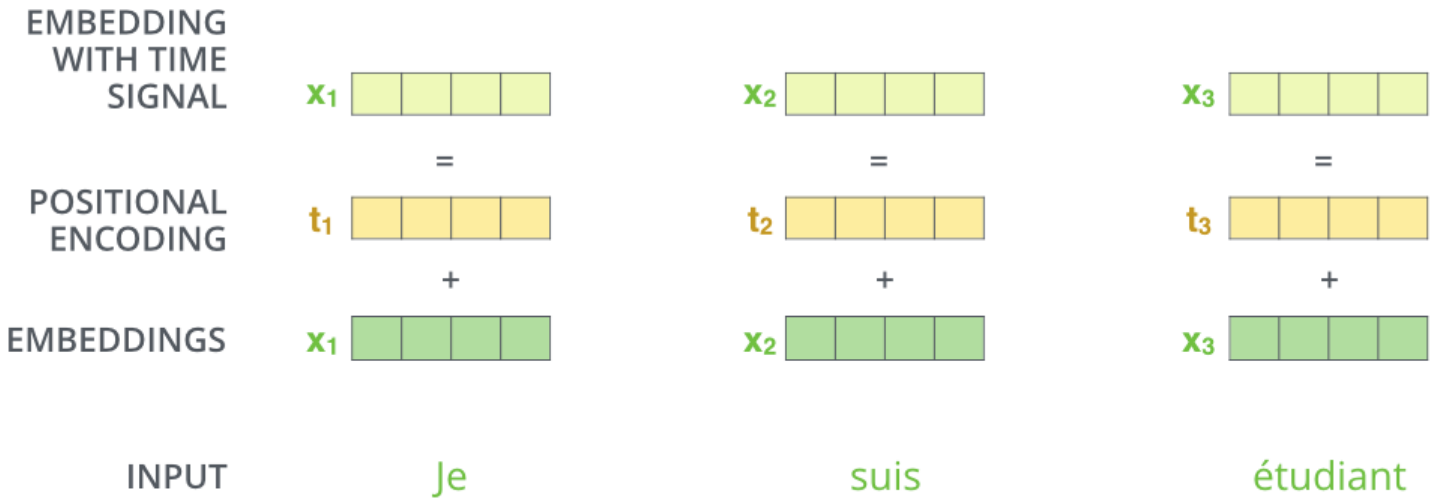
The Transformer



https://jalammar.github.io/illustrated-transformer/

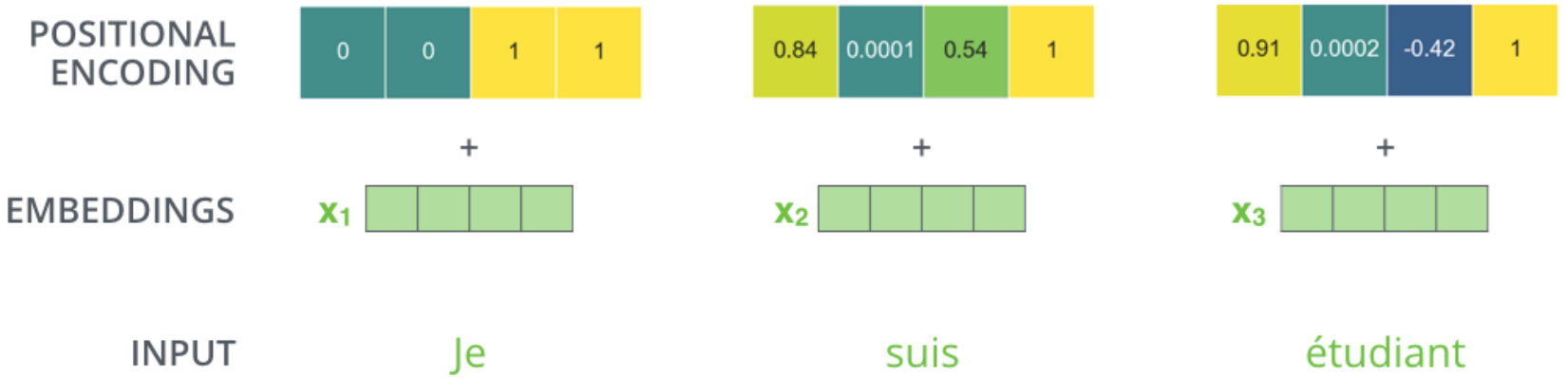
The Transformer

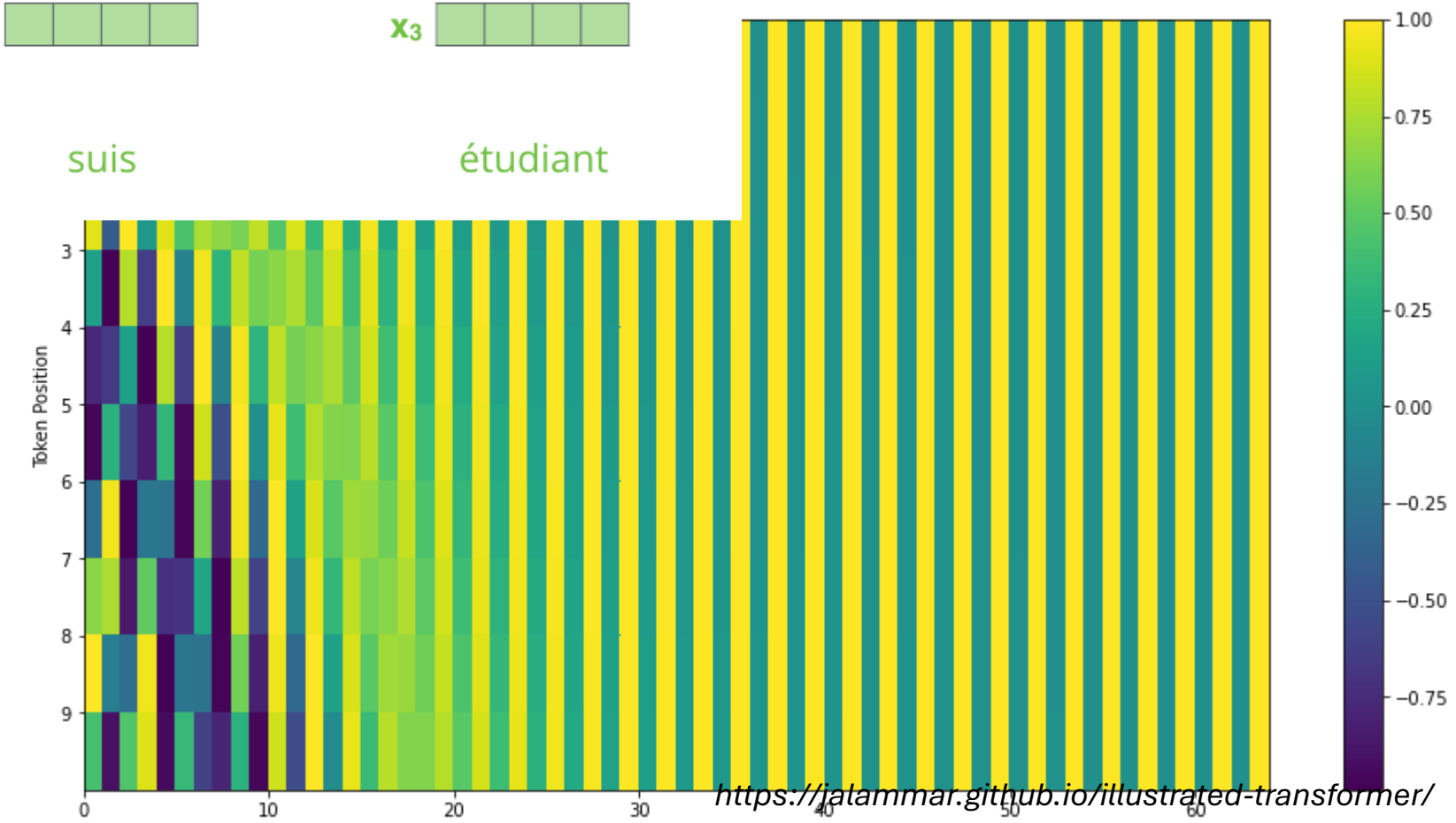




https://jalammar.github.io/illustrated-transformer/

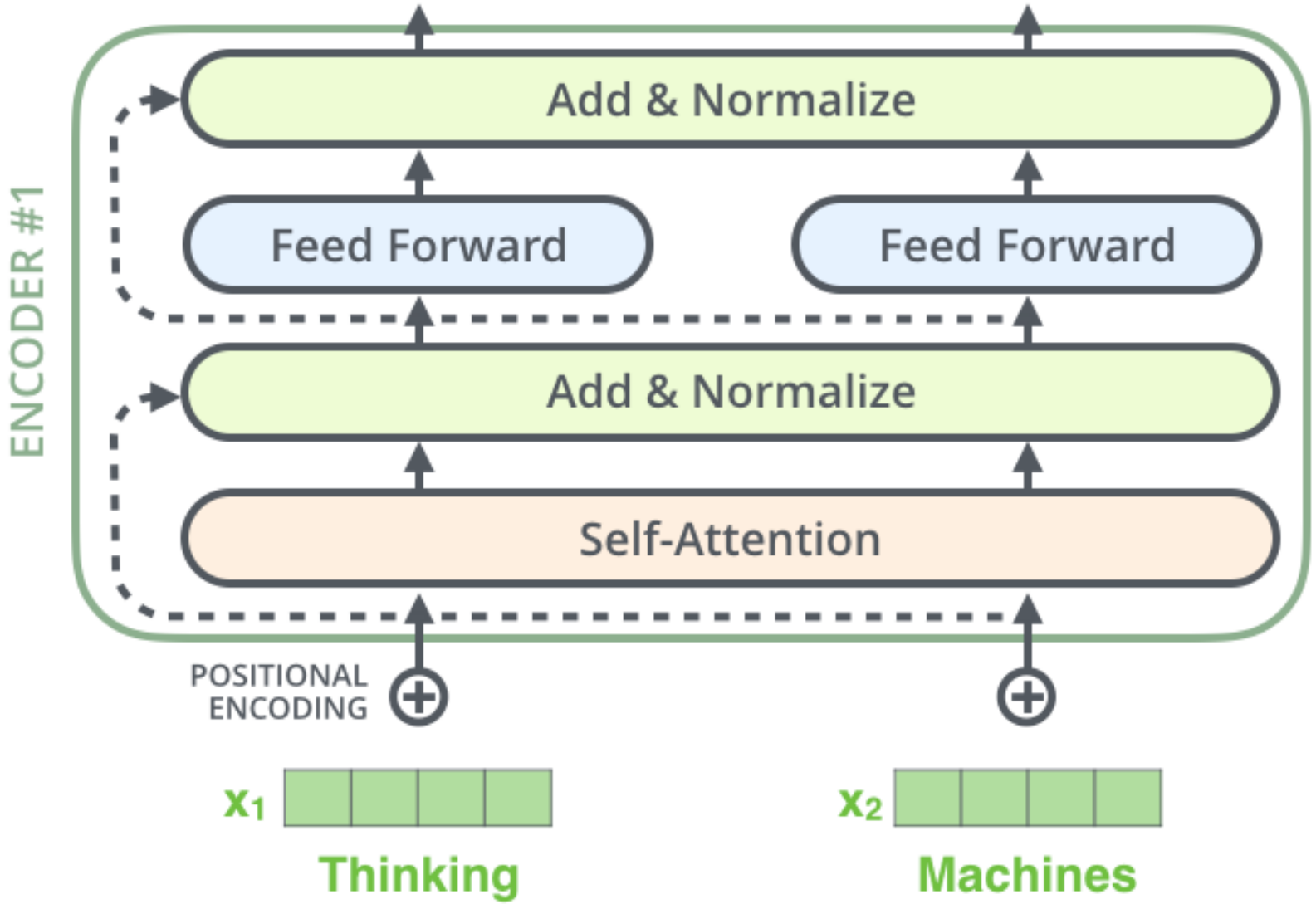
The Transformer





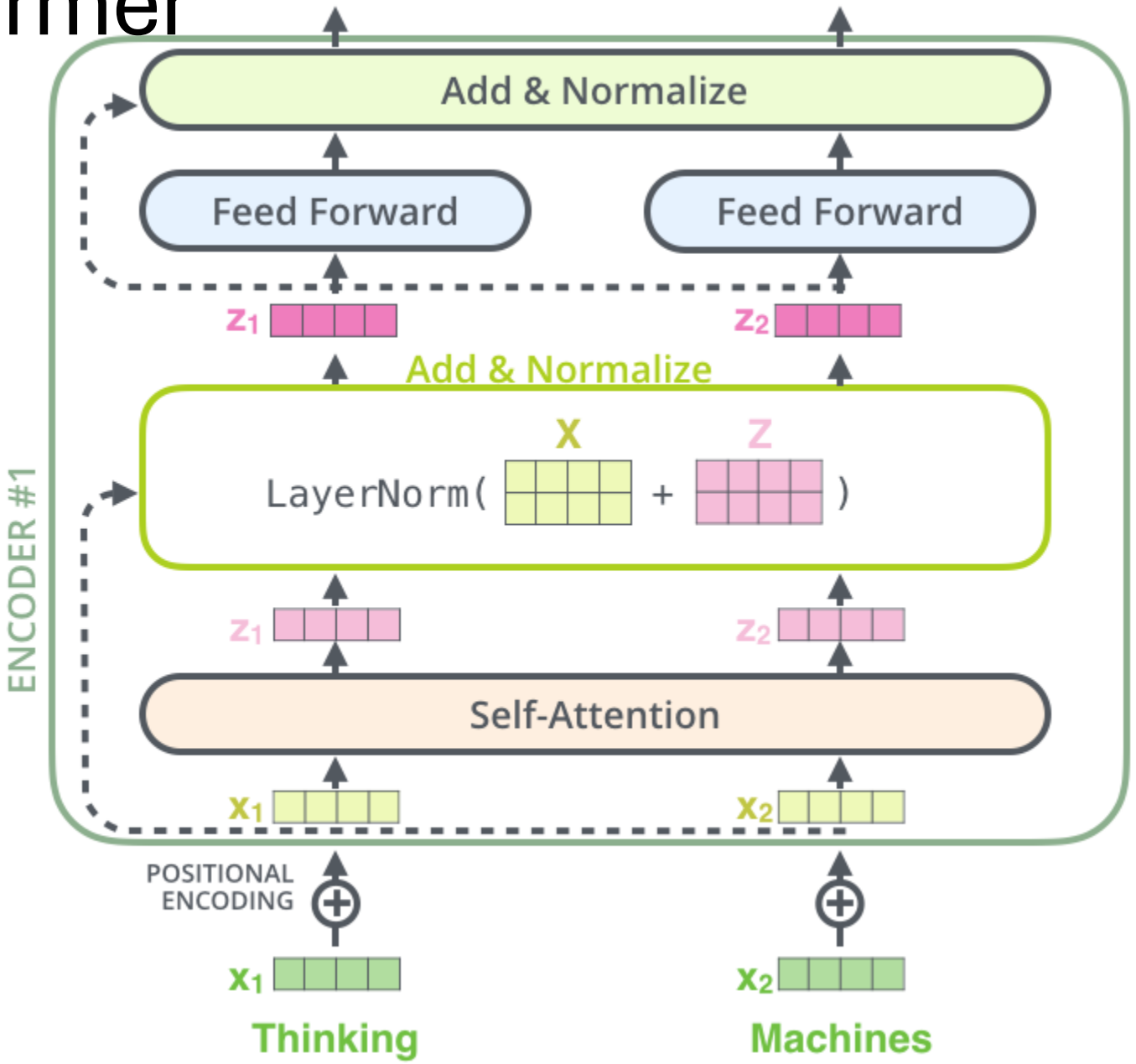
https://jalammar.github.io/illustrated-transformer/

The Transformer



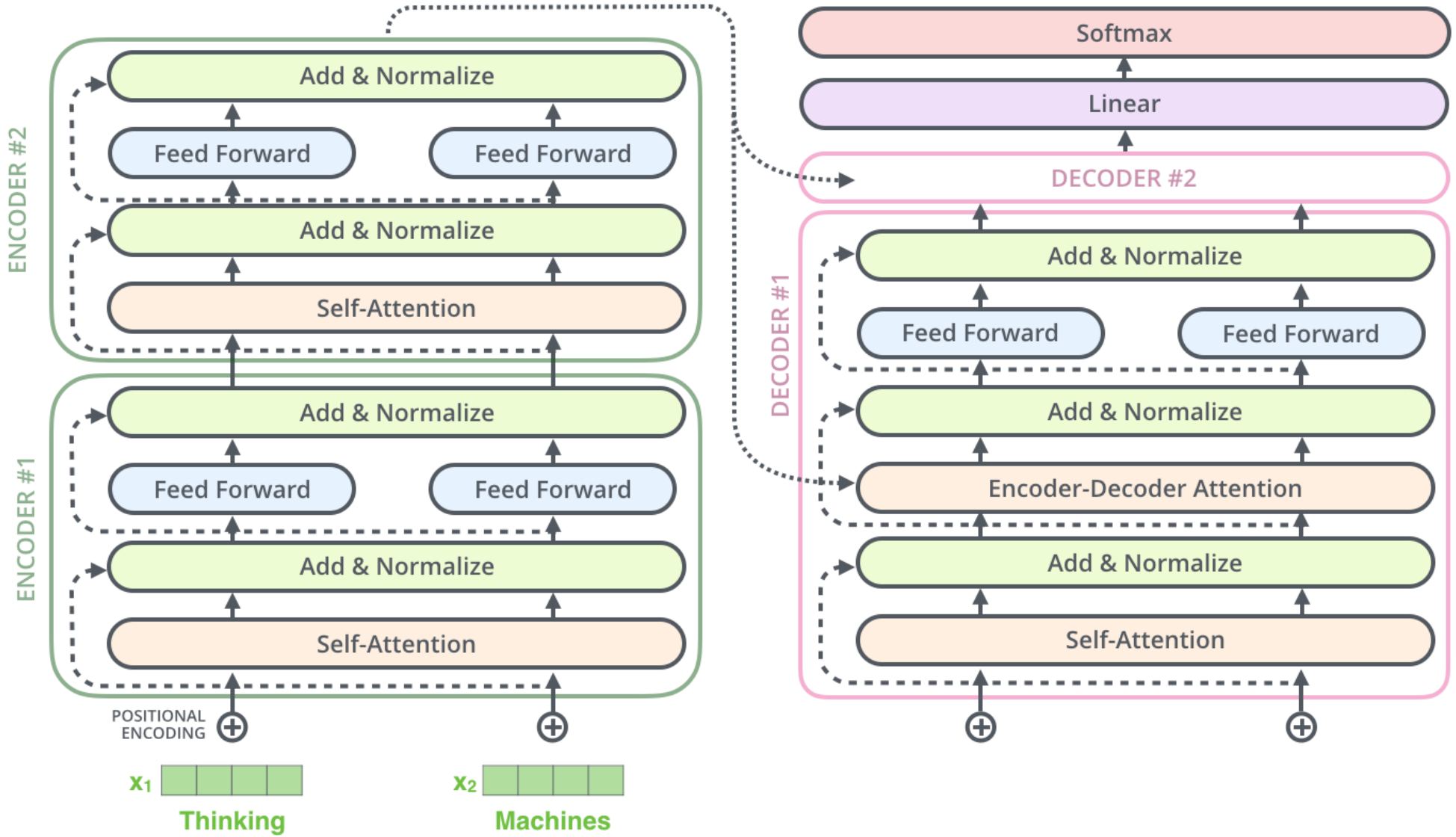
https://jalammar.github.io/illustrated-transformer/

The Transformer



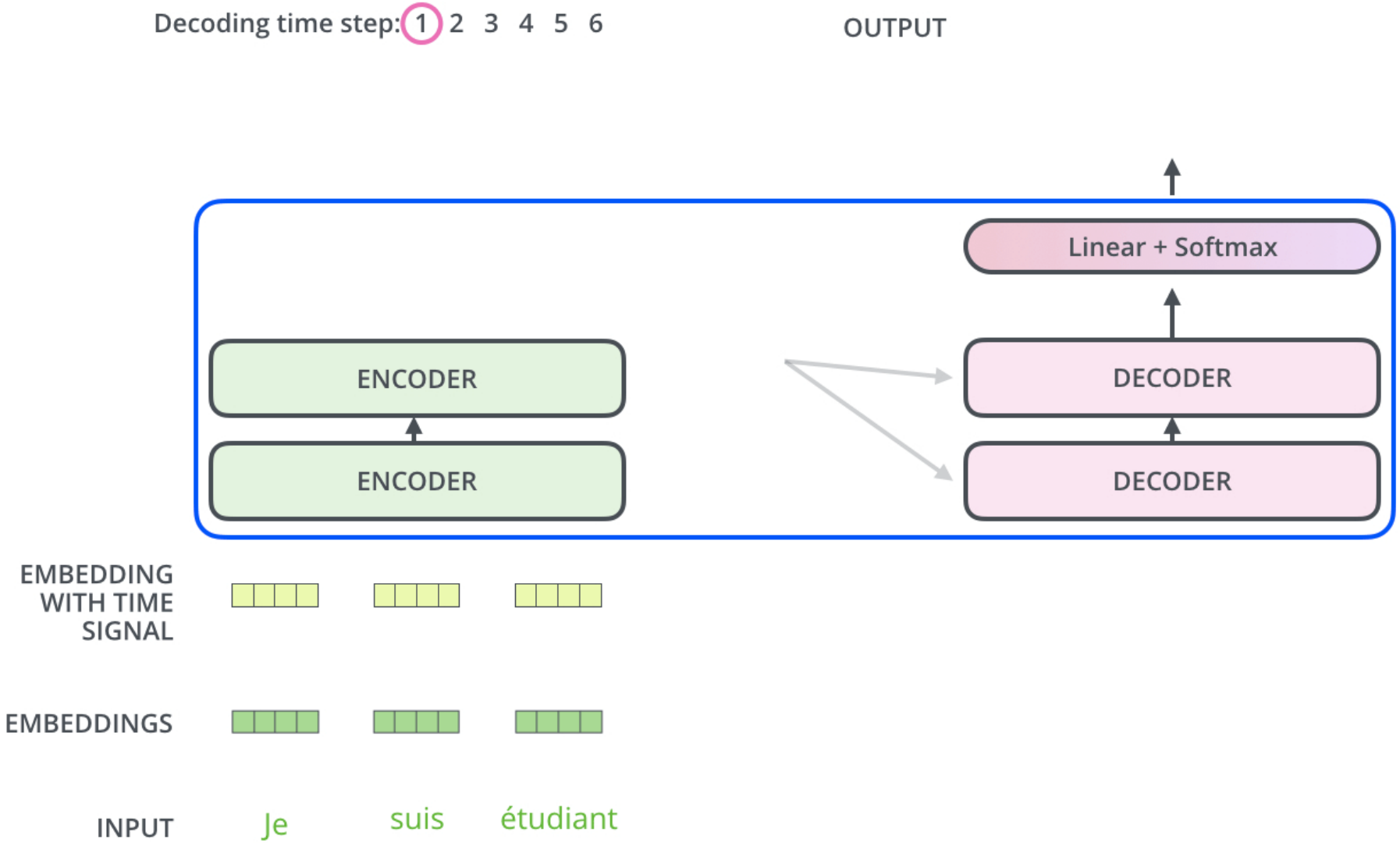
https://jalammar.github.io/illustrated-transformer/

The Transformer



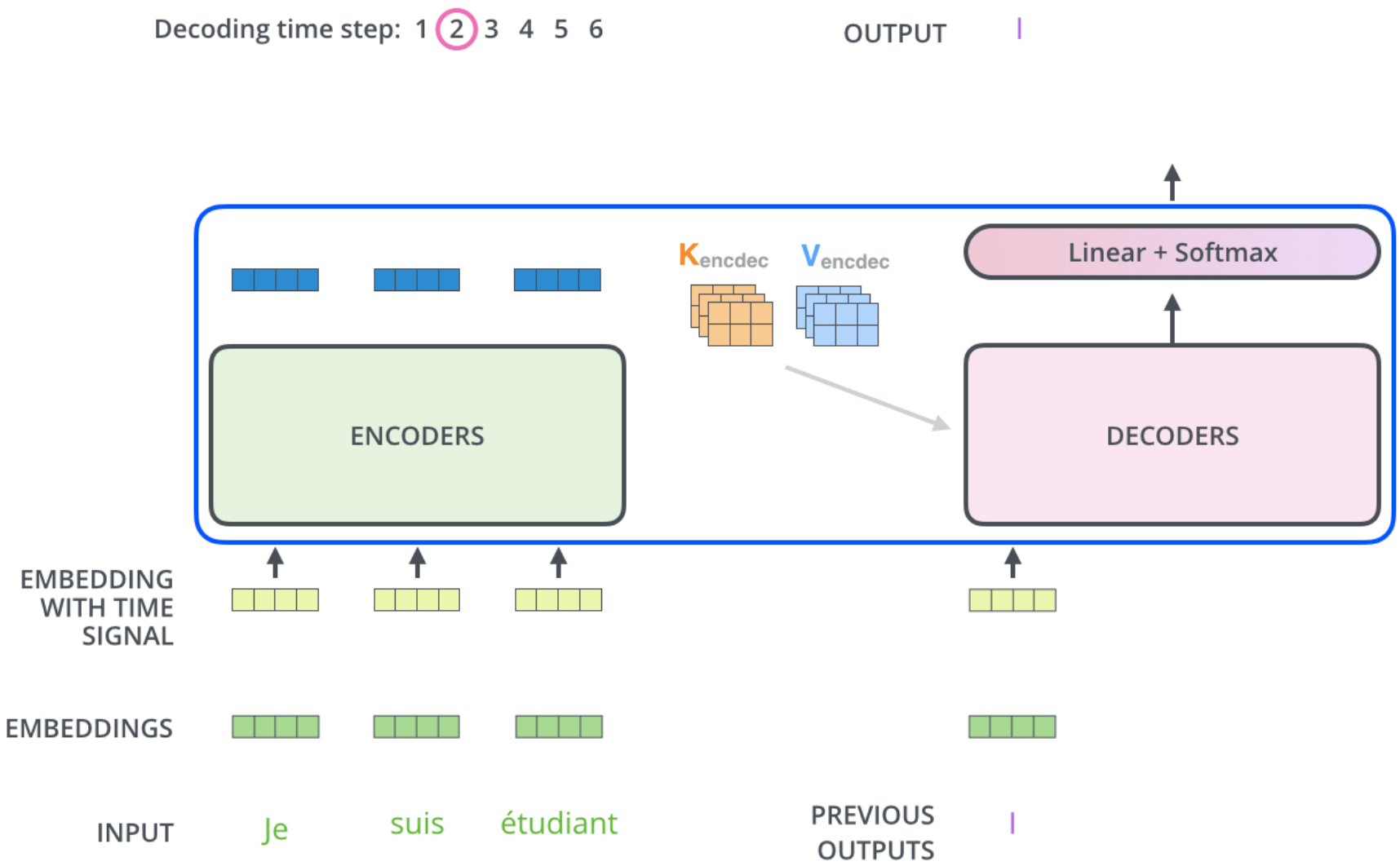
https://jalammar.github.io/illustrated-transformer/

The Transformer



https://jalammar.github.io/illustrated-transformer/

The Transformer



https://jalammar.github.io/illustrated-transformer/