Deep Learning

Transformer

Lionel Fillatre

2025-2026

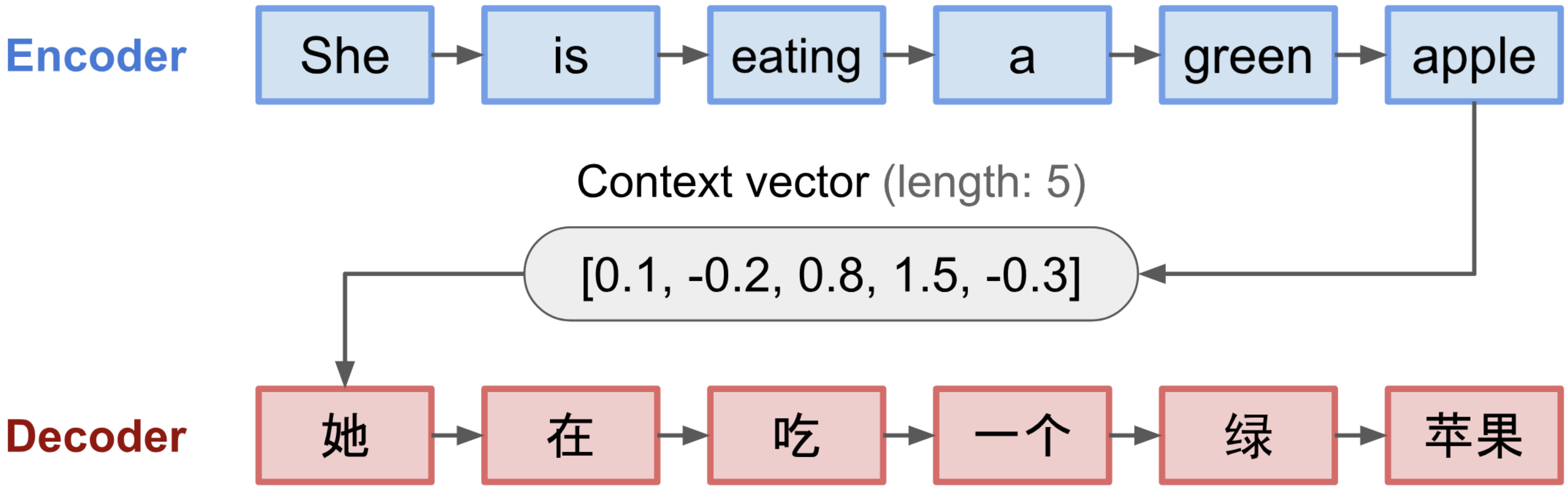
Outline

- Attention Mecanism
- Single-Head Attention
- Multi-Head Attention
- Decode Only Transformer
- Input Encoding
- How does training work?
- Encoder-Decoder Transformer
- Conclusion

Attention Mecanism

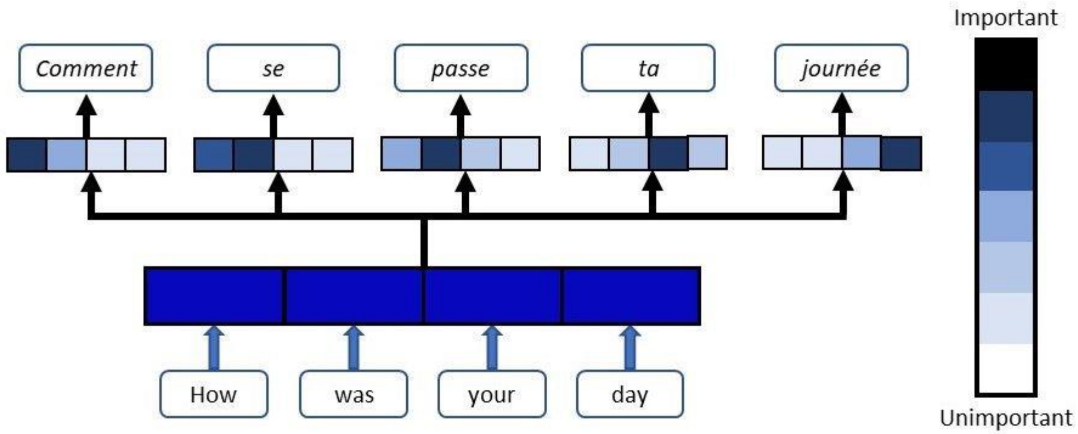
Reminder: Encoder/Decoder

- The encoder-decoder model, translating the sentence “she is eating a green apple” to Chinese.
- The visualization of both encoder and decoder is unrolled in time.



https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Decoder with attention



- The decoder is still a RNN
  - RNN decoder hidden state: $s_{t'} = f(y_{t'-1}, s_{t'-1}, c_{t'})$

- With an RNN, each conditional probability is modeled as

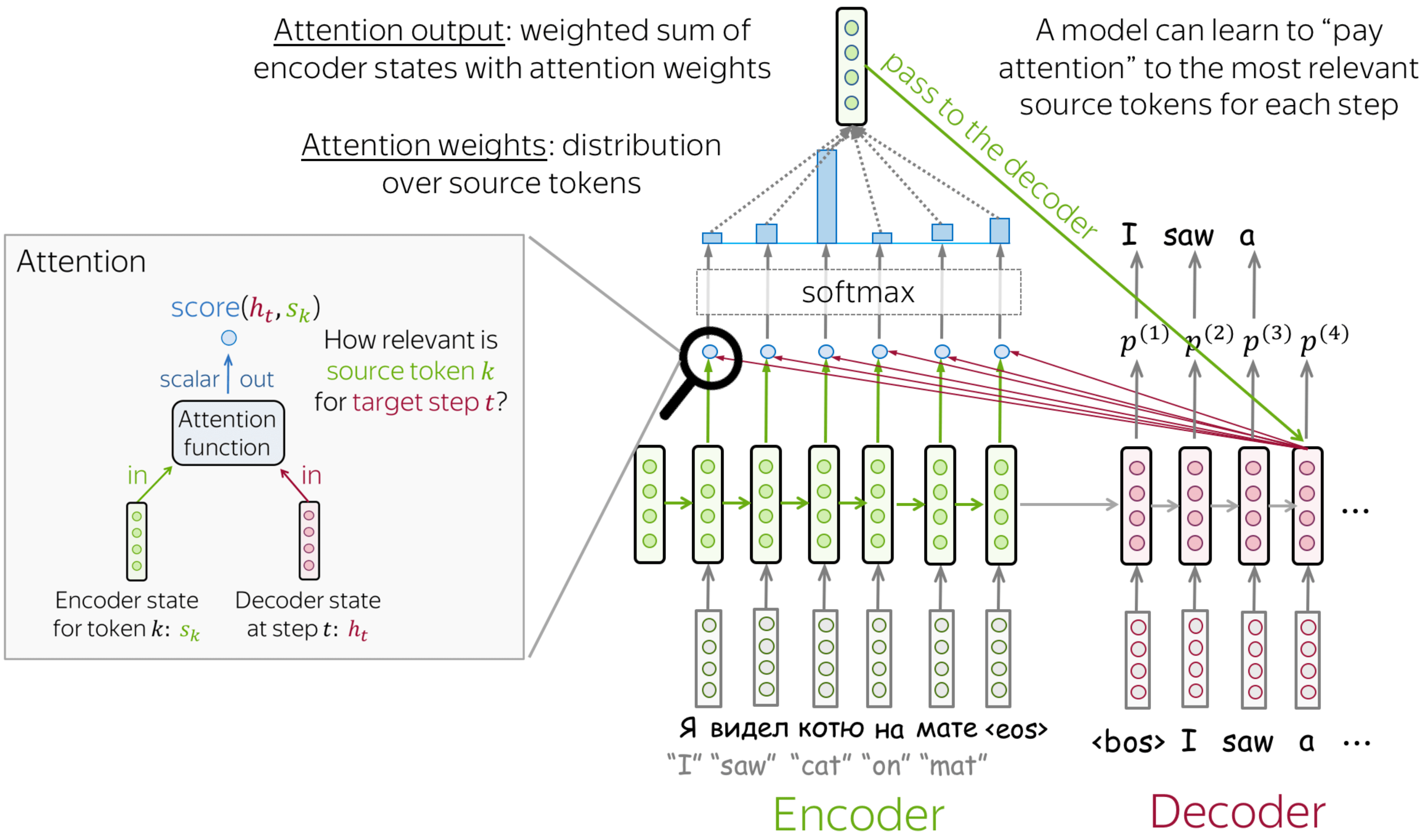
$$p(y_{t'}|y_1, \dots, y_{t'-1}, x) = g(y_{t'-1}, s_{t'}, c_{t'})$$

  - g is a nonlinear, potentially multi-layered, function that outputs the probability of $y_{t'}$

- Principle: the probability is conditioned on a distinct context vector $c_{t'}$ for each target word $y_{t'}$

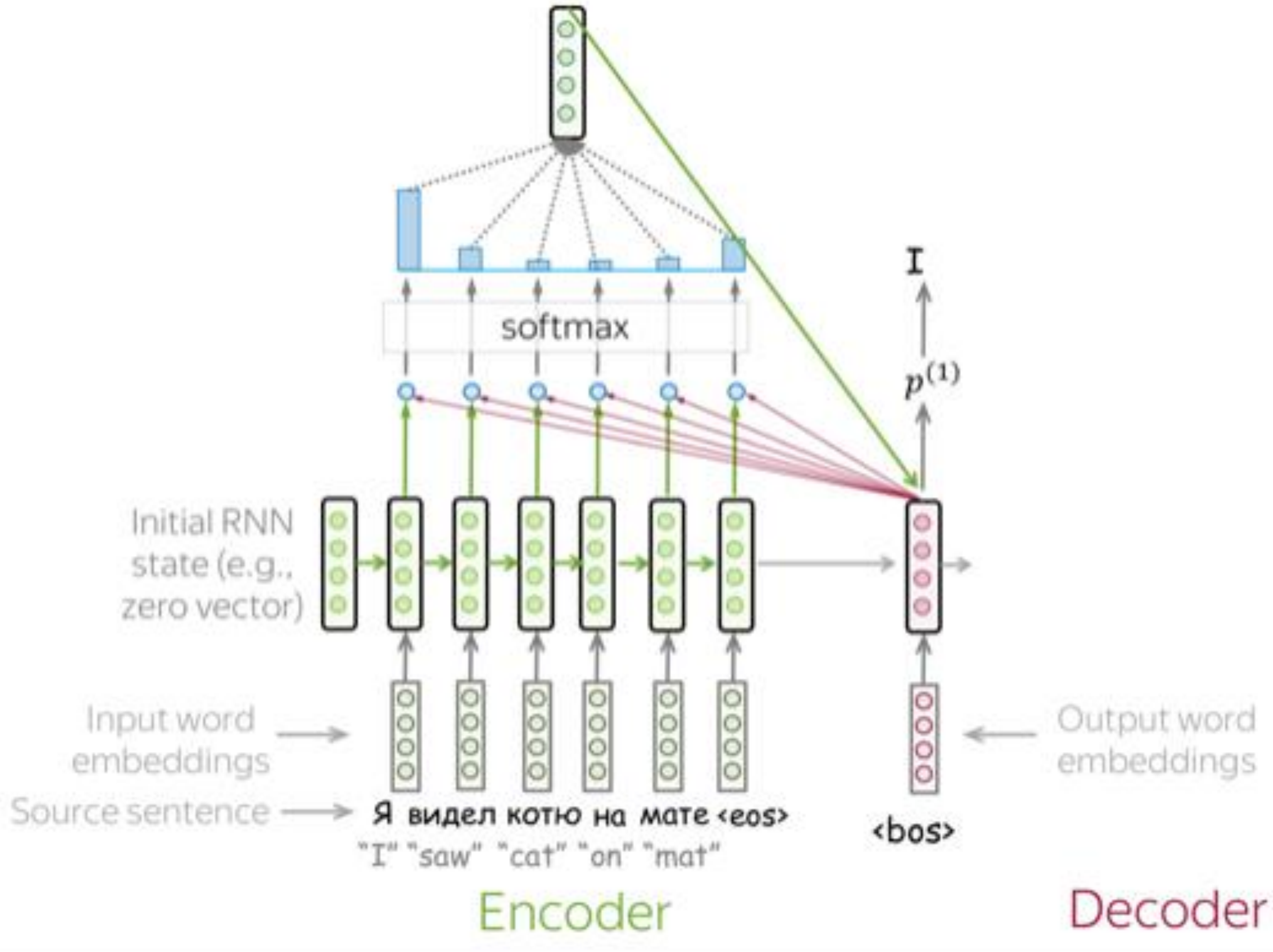
Neural Machine Translation by Jointly Learning to Align and Translate
Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, ICLR, 2015

An illustration

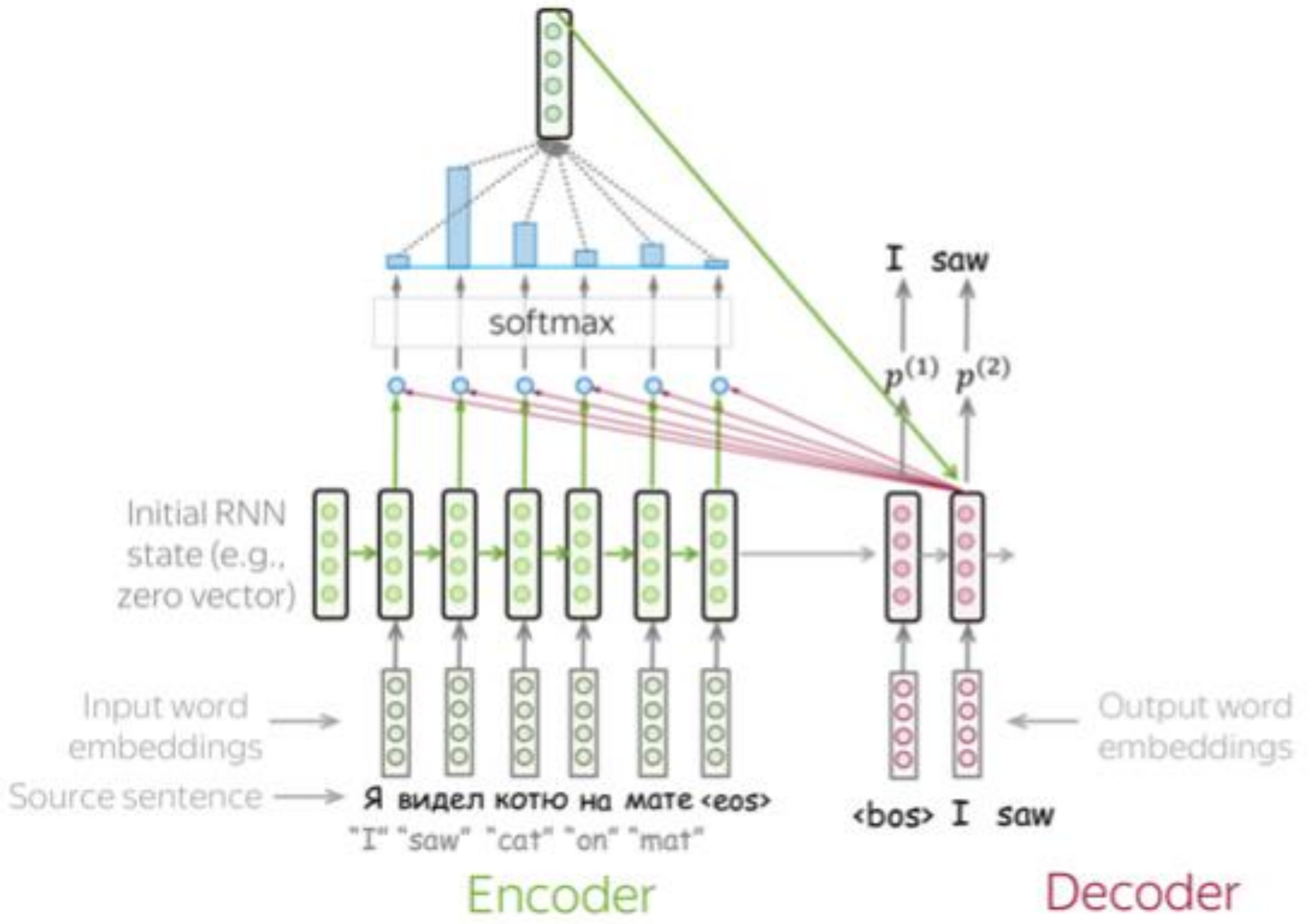


https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

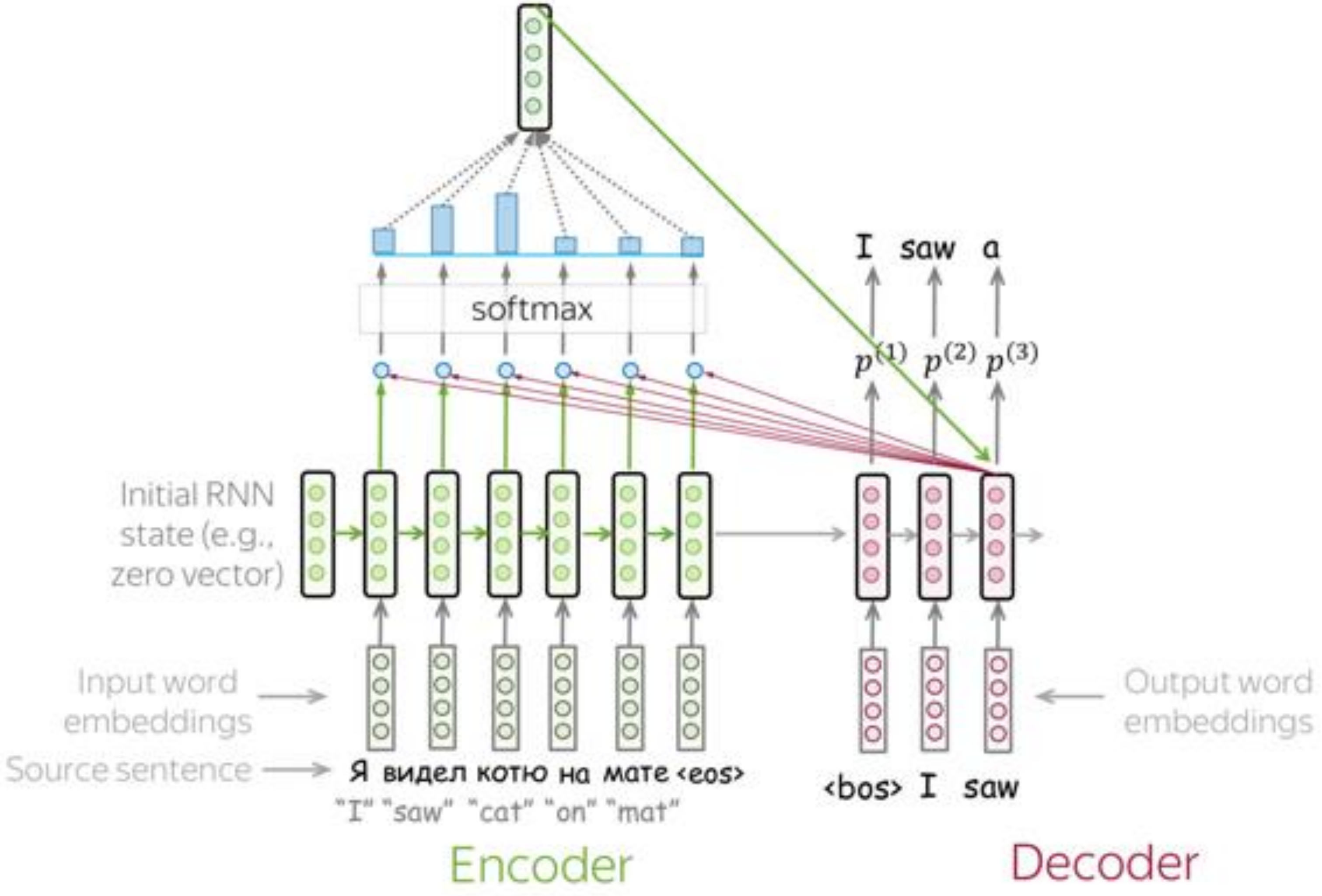
An illustration



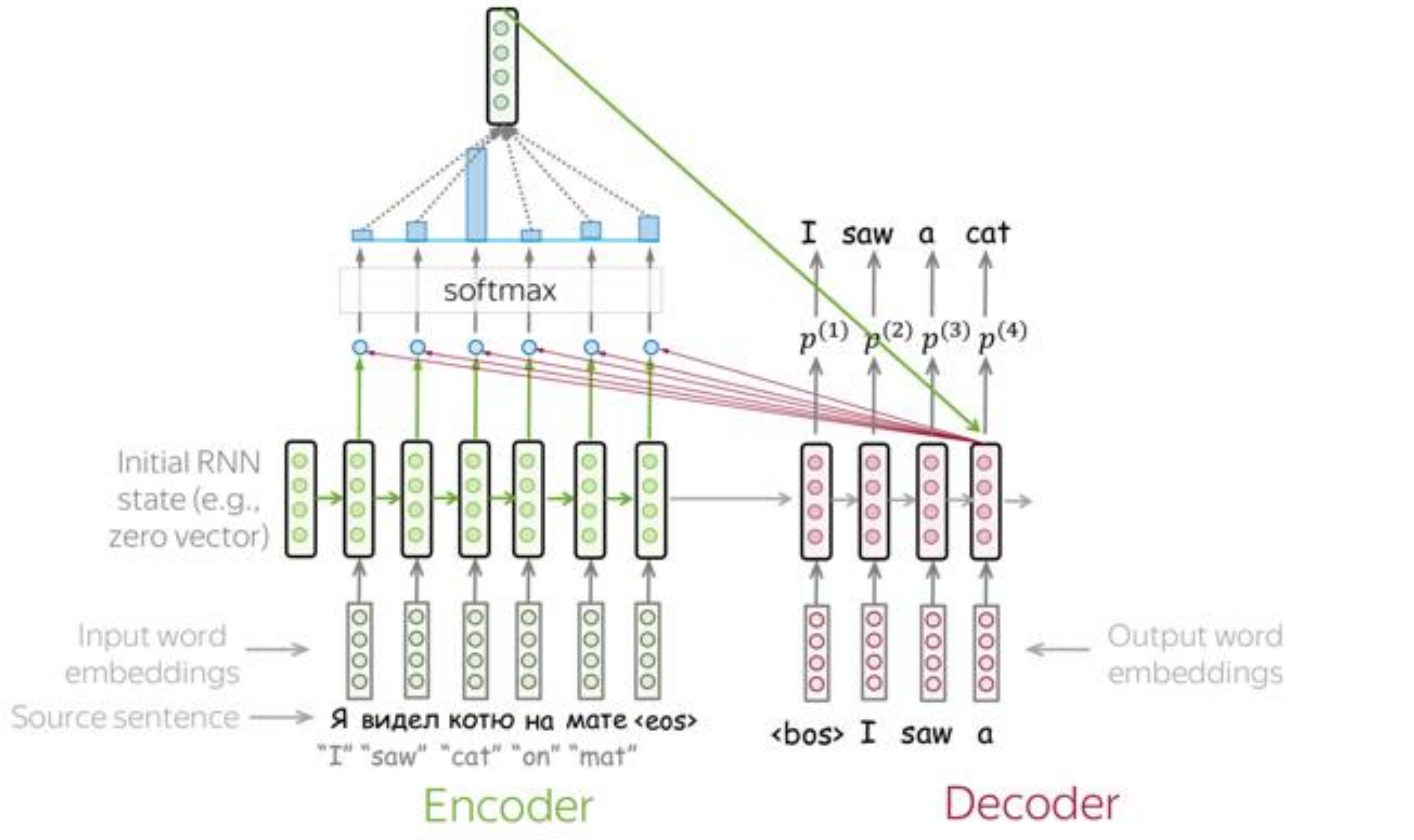
An illustration



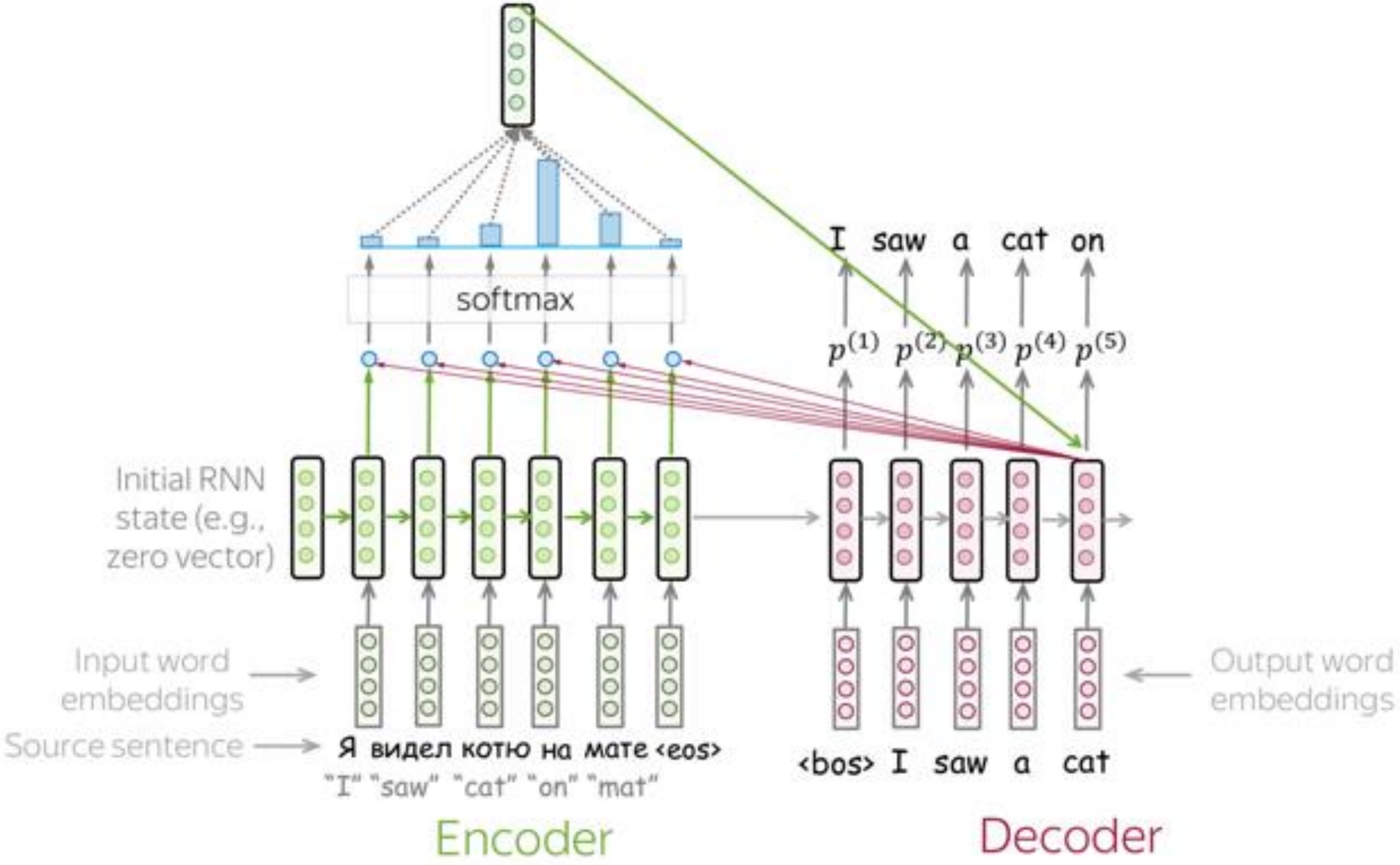
An illustration



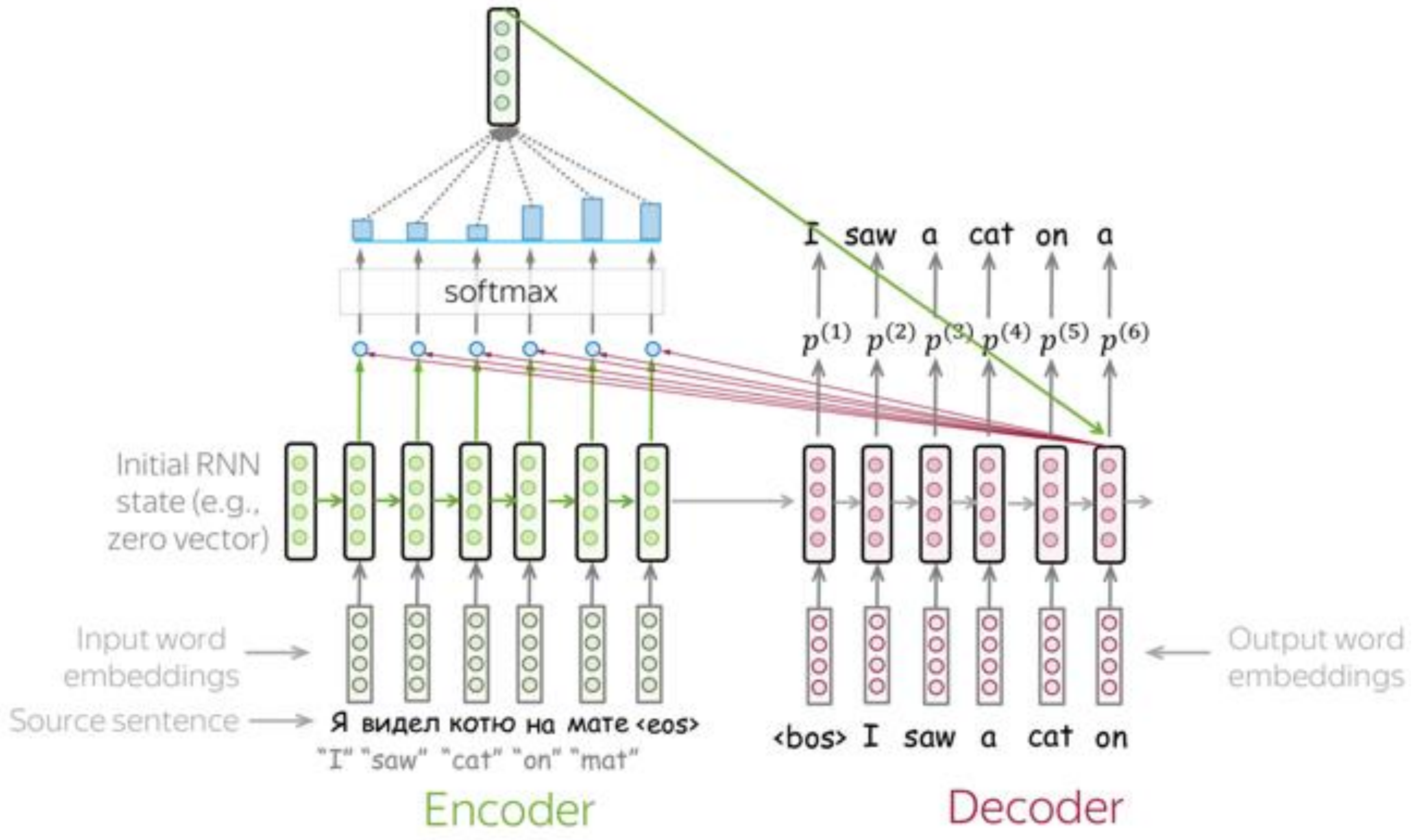
An illustration



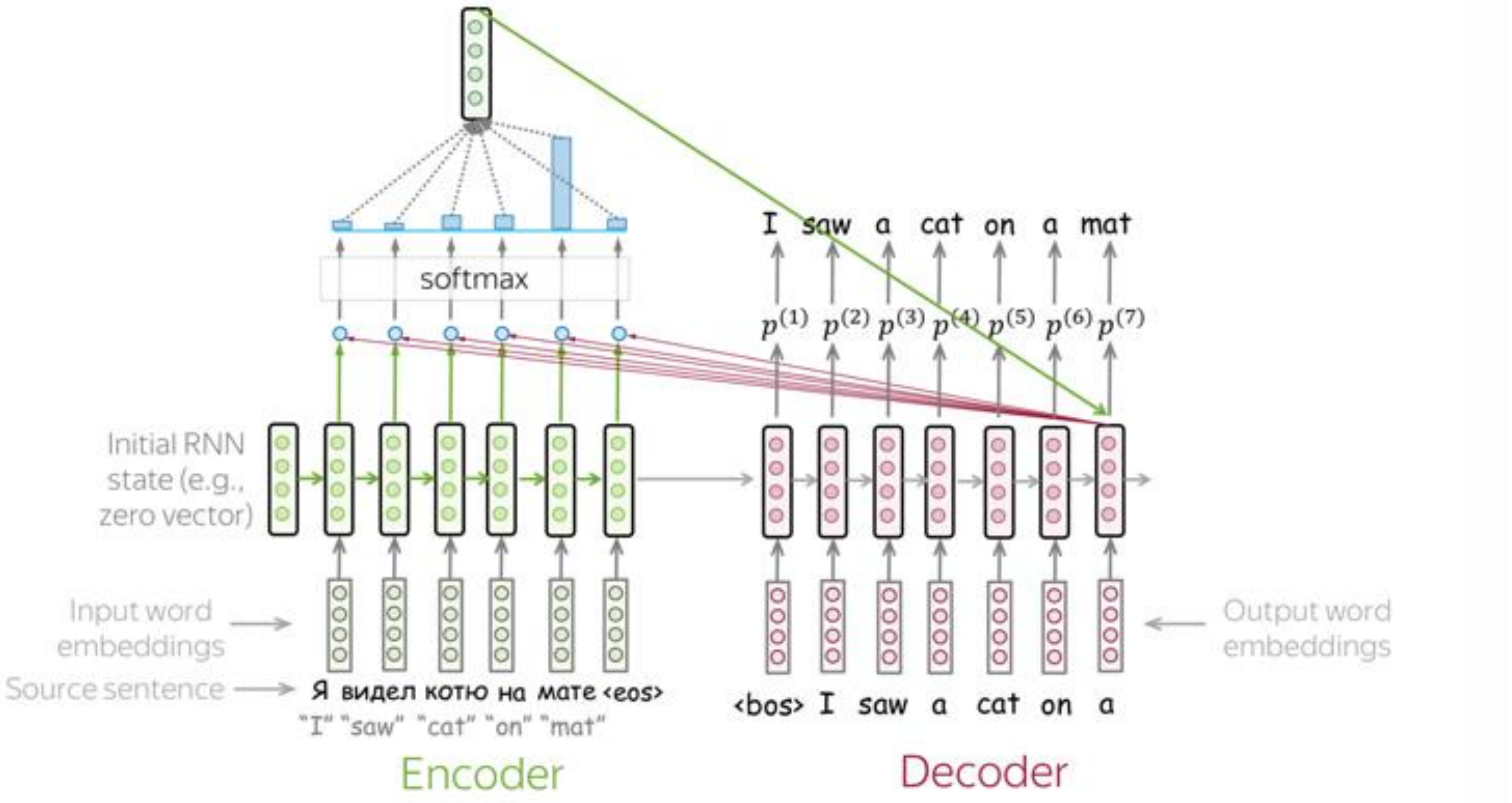
An illustration



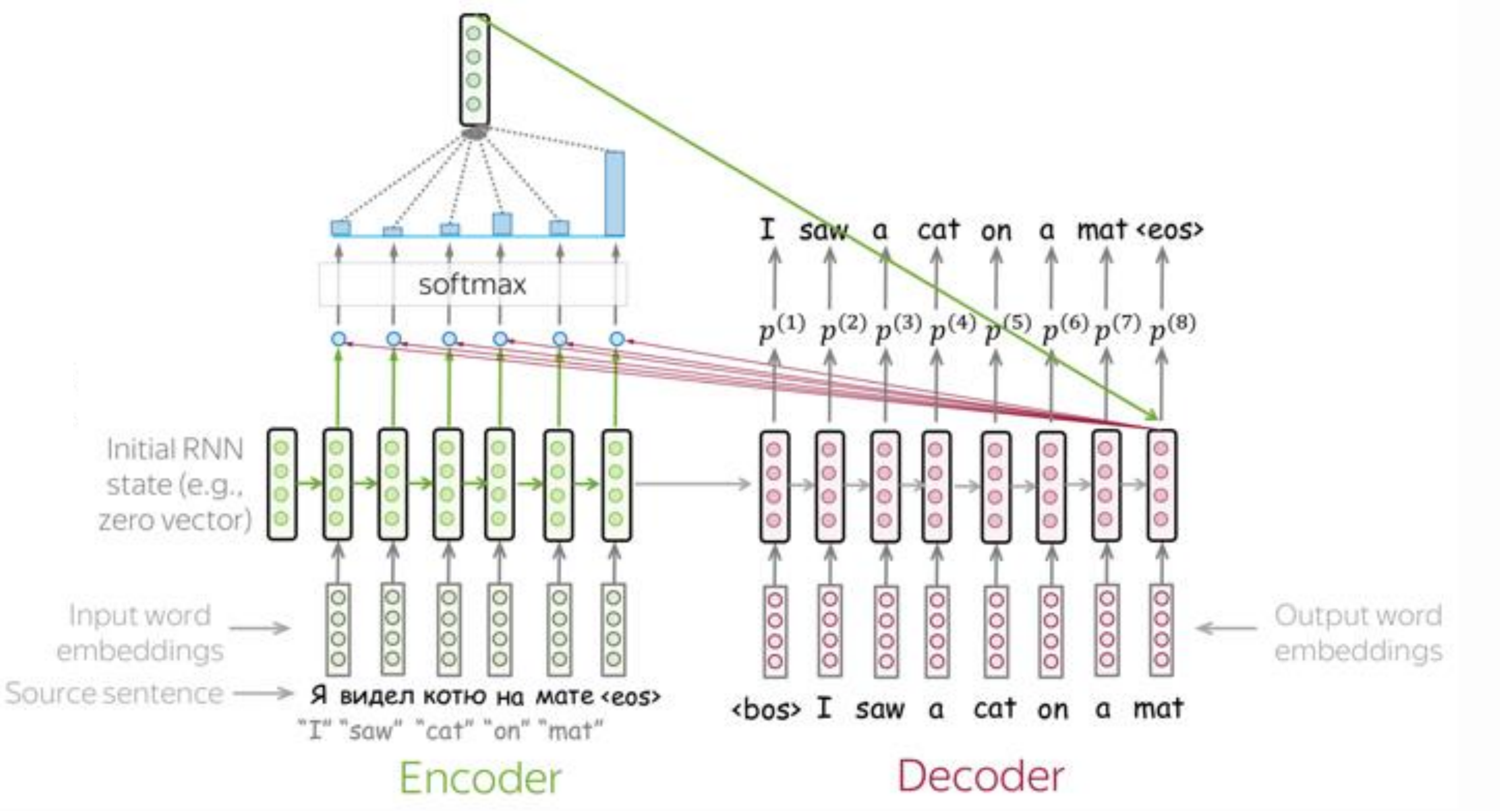
An illustration



An illustration



An illustration



Annotation

- The context vector c_i depends on a sequence of annotations $h = (h_1, h_2, \dots, h_T)$ to which an encoder maps the input sentence.
- Each annotation h_i contains information about the whole input sequence (especially for a bidirectional RNN) with a strong focus on the parts surrounding the i-th word of the input sequence.
- Generally, the annotation is a RNN hidden state
- The context vector c_i is computed as a weighted sum of the annotations h_i:

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

with $0 \leq \alpha_{ij} \leq 1$

Annotation weights

- The weight α_{ij} of each annotation h_j is computed by a softmax function

$$\alpha_{ij} = \mathrm{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}$$

where

$$e_{ij} = \mathrm{score}(s_i, h_j)$$

- The score e_{ij} measures how well the inputs around position j and the output at position i match
- The score is based on the RNN hidden state s_i (just before emitting y_i) and the j-th annotation h_j of the input sentence.

To to compute the score?

- The most popular score are
  - Dot-product: $e_{ij} = \text{score}(s_i, h_j) = s_i^T h_j = h_j^T s_i$
  - Bilinear function: $e_{ij} = \text{score}(s_i, h_j) = s_i^T W h_j = h_j^T W s_i$
  - Multi-Layer Perceptron: $e_{ij} = \text{score}(s_i, h_j) = a(s_i, h_j) = w_2^T \tanh(W_1[s_i, h_j])$

- In the original paper, the authors used the **alignment model** $a(\cdot)$ as a feedforward neural network which is jointly trained with all the other components of the proposed system.

Neural Machine Translation by Jointly Learning to Align and Translate, Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, ICLR, 2015

Interpretation of $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}$

- We can understand the approach of taking a weighted sum of all the annotations as computing an **expected annotation**, where the expectation is over possible alignments.

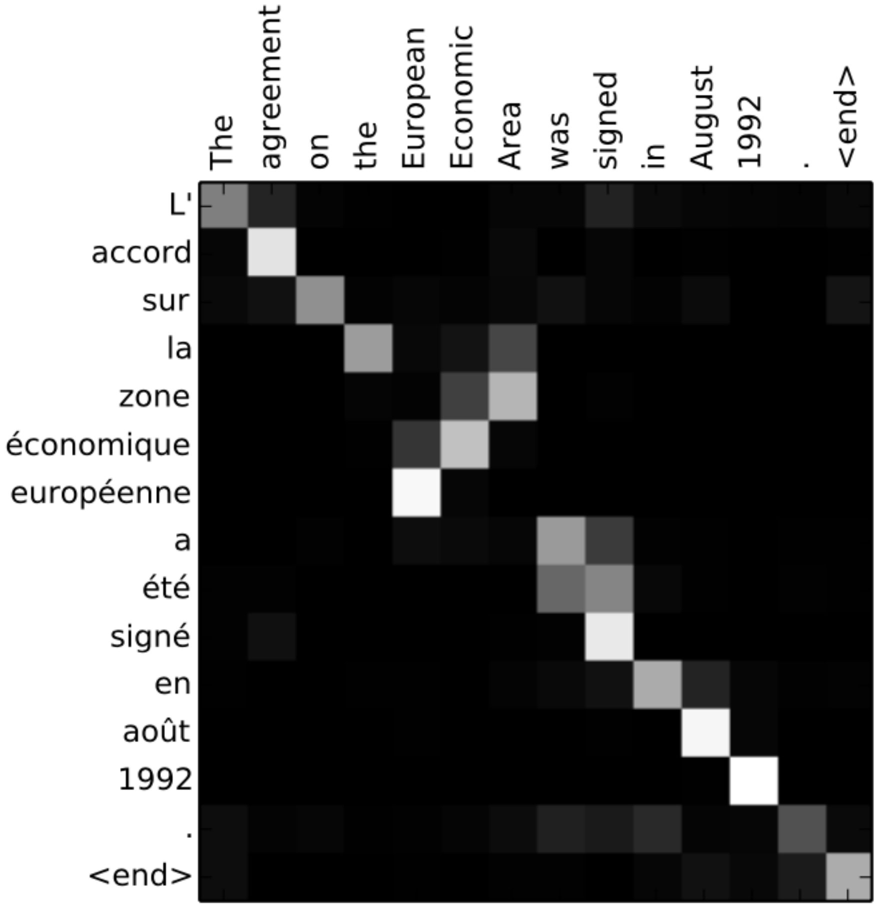
- Let α_{ij} be a probability that the target word y_i is aligned to, or translated from, a source word x_j. Then, the i-th context vector c_i is the expected annotation over all the annotations h_j with probabilities α_{ij}

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

- The probability α_{ij}, or its associated score e_{ij}, reflects the importance of the annotation h_j with respect to the current decoding state s_i in deciding the next prediction y_i.
- Intuitively, this implements a **mechanism of attention** in the decoder.

Illustration of the alignment

- The x-axis and y-axis of the plot correspond to the words in the source sentence (English) and the generated translation (French), respectively.
- Each pixel shows the weight α_{ij} of the annotation of the j-th source word for the i-th target word, in grayscale (0: black, 1: white).



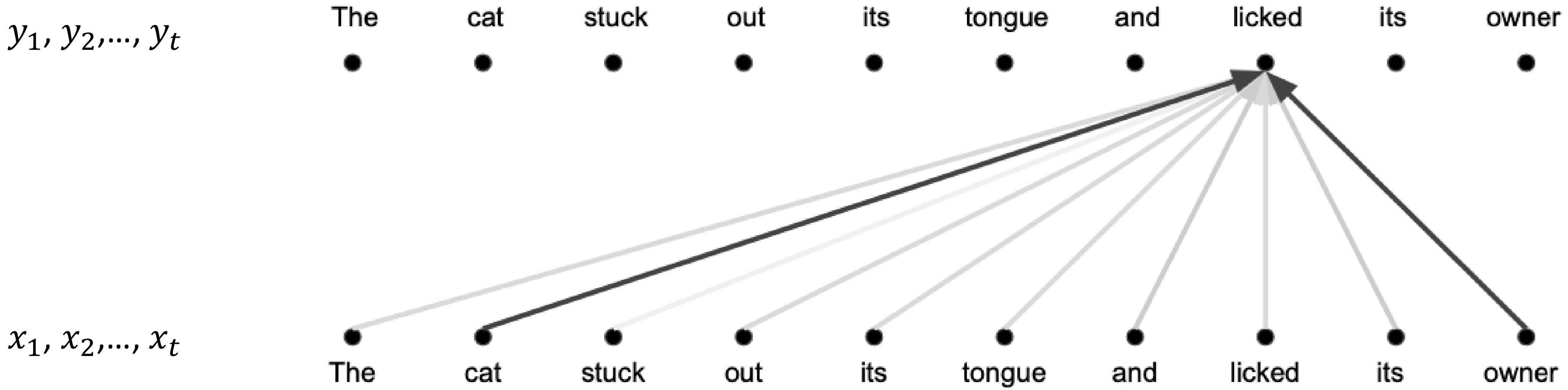
Single-head attention

Transformer’s Encoder: Principle

- Self-attention
- Queries, keys and values
- Scaling the dot product
- Multi-head attention

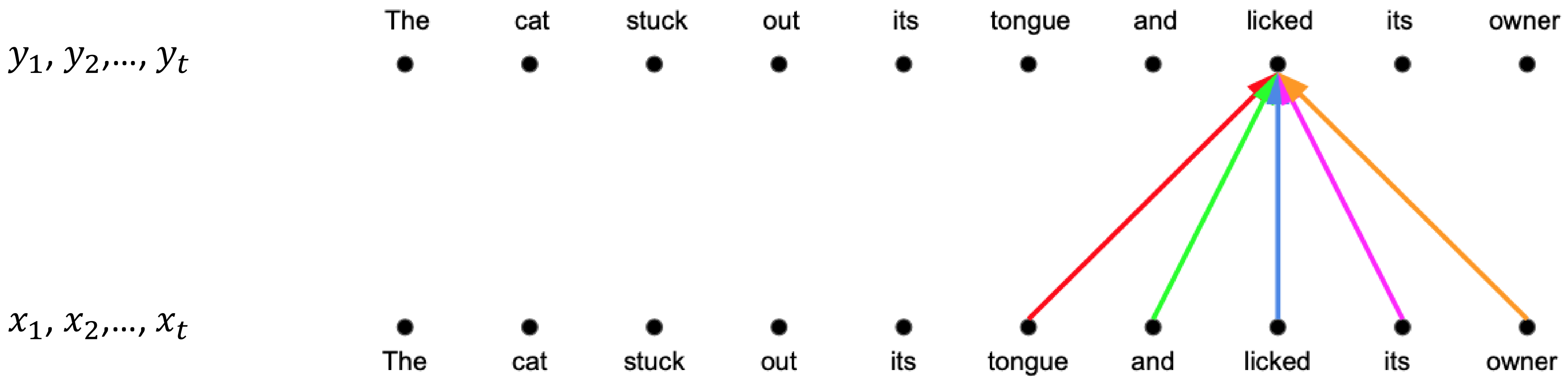
“Deep Contextualized Word Representations” in NAACL, 2018
Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer

Example with a sentence



- The word « licked » is most correlated to « cat » (who?) ans « owner » (to whom?)

Comparison with convolution



- The word « licked » is only correlated to words in a given neighborhood (kernel size)

Self-attention

- Self-attention is a sequence-to-sequence operation:
  - A sequence of vectors goes in, and a sequence of vectors comes out.
  - Let's call the input vectors x_1, x_2,…, x_t and the corresponding output vectors y_1, y_2,…, y_t .
  - The vectors all have dimension d (the inputs are embedded with an embedding layer).

- To produce output vector y_i, the self attention operation simply takes a weighted average over all the input vectors

$$y_i = \sum_{j=1}^{t} w_{ij} x_j$$

where the positive weights w_{ij} sum to one over all j.

Self-attention: basic operation

- The weight

$$w_{ij} = \text{softmax}(e_{ij}) = \text{softmax}\left(\text{score}(x_i, x_j)\right)$$

is not a parameter, as in a normal neural net, but it is derived from a function over x_i and x_j.

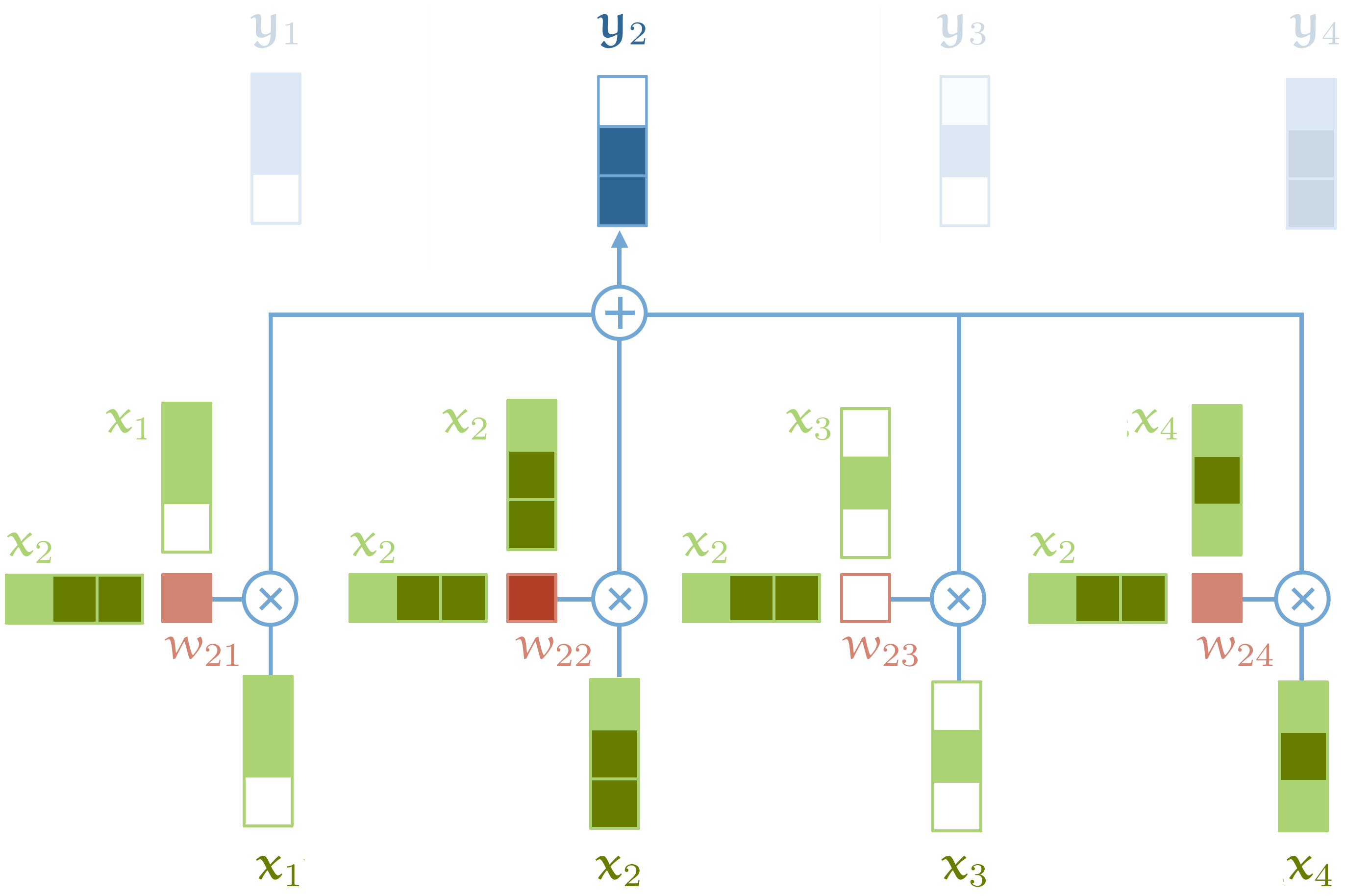
- The simplest option for the score function is the dot product:

$$e_{ij} = x_i^T x_j$$

- The dot product gives us a value anywhere between negative and positive infinity, so we apply a softmax to map the values to $[0,1]$ and to ensure that they sum to 1 over the whole sequence:

$$w_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{t} \exp(e_{ik})}$$

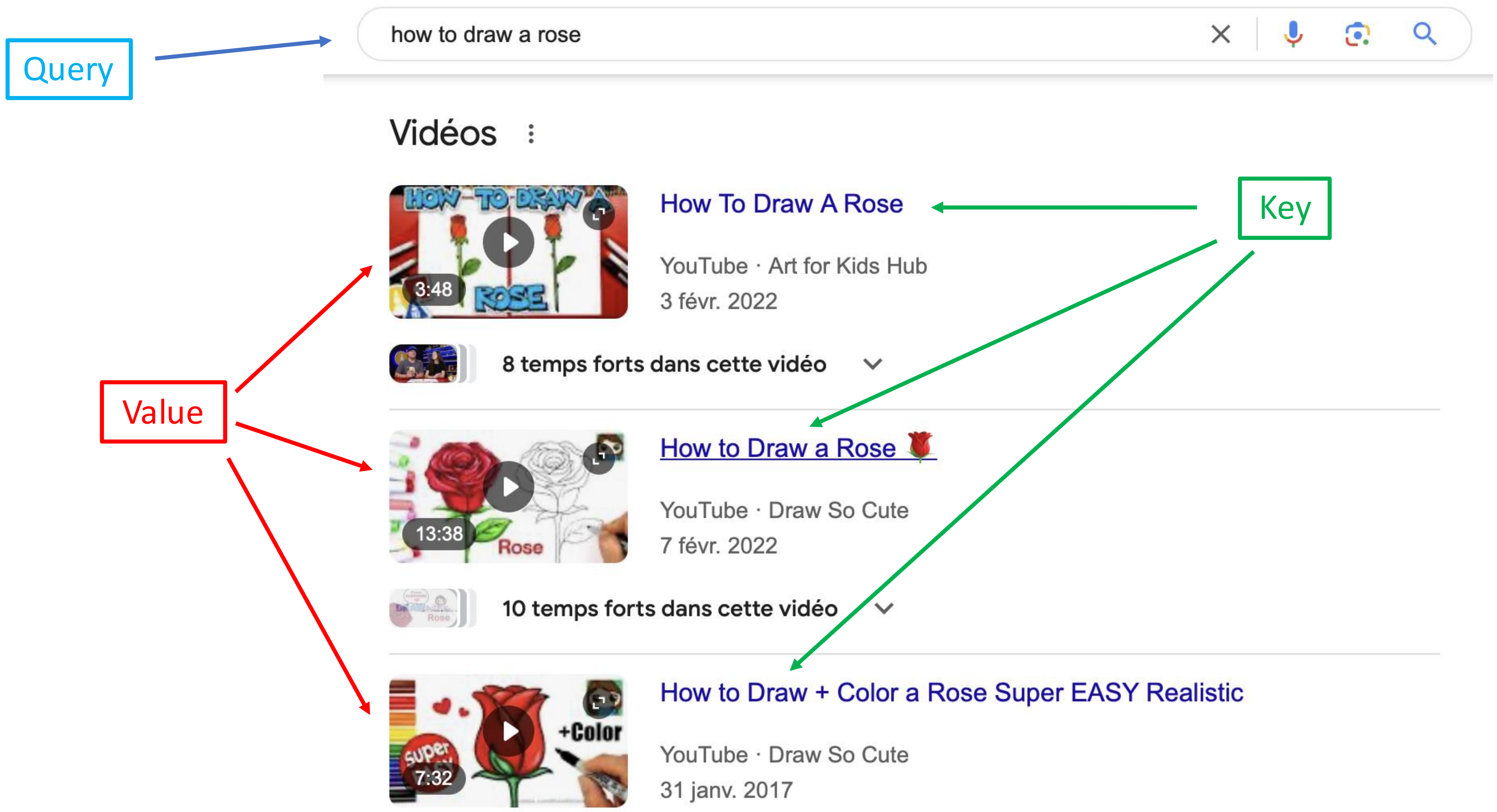
Self-attention



A visual illustration of basic self-attention.
Note that the softmax operation over the weights is not illustrated.

http://peterbloem.nl/blog/transformers

Example: How to draw a rose



Queries, keys and values

$$y_i = \sum_{j=1}^{t} w_{ij} x_j, \quad w_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{t} \exp(e_{ik})}, \quad e_{ij} = x_i^T x_j$$

- Every input vector x_i is used in three different ways in the self attention operation (each role has a name: query, key or value):

$$y_i = \sum_{j=1}^{t} \frac{\exp(x_i^T x_j)}{\sum_{k=1}^{t} \exp(x_i^T x_k)} x_j$$

1. Query: vector from which the attention is looking for its own output y_i
2. Key: It is compared to every other vector at which the query looks to establish the weights
3. Value: It is used as part of the weighted sum to compute each output vector once the weights have been established.

- In the basic self-attention we've seen so far, each input vector must play all three roles.
- In the transformer, new vectors for each role are derived, by applying a linear transformation to the original input vector

Linear transformation for each role

- We can add three $d \times d$ weight matrices W^Q, W^K, W^V to compute three linear transformations of each x_i, for the three different parts of the self attention:

$$q_i = W^Q x_i, \qquad k_i = W^K x_i, \qquad v_i = W^V x_i$$

$$e_{ij} = q_i^T k_j$$

$$w_{ij} = \mathrm{softmax}(e_{ij})$$

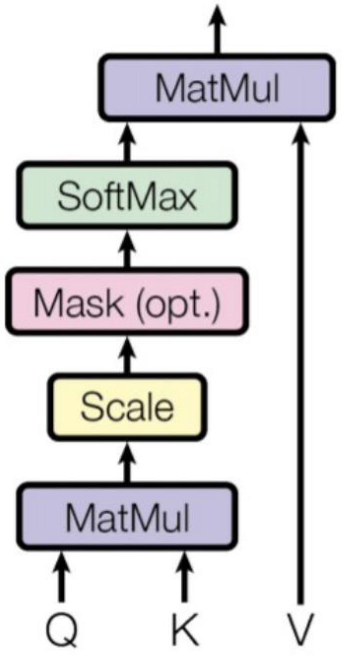
$$y_i = \sum_{j=1}^{t} w_{ij} v_j$$

- This gives the self-attention layer some controllable parameters, and allows it to modify the incoming vectors to suit the three roles they must play.

Attention function: matrix form

- In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q (after the linear transformation if we use it).
  - Initial vectors are the rows of Q
- The keys and values are also packed together into matrices K and V.
- We compute the matrix of outputs as:

$$\mathrm{Attention}(\mathrm{Q}, \mathrm{K}, \mathrm{V}) = \mathrm{softmax}\left(\frac{Q\,K^T}{\sqrt{d}}\right)V$$



Scaling the dot product: why $\sqrt{d}$?

- The softmax function can be sensitive to very large input values.
- These kill the gradient, and slow down learning, or cause it to stop altogether.
- Since the average value of the dot product grows with the embedding dimension k, it helps to scale the dot product back a little to stop the inputs to the softmax function from growing too large:

$$e_{ij} = \frac{q_i^T k_j}{\sqrt{d}}$$

- Why $\sqrt{d}$?
  - Imagine a vector in $\mathbb{R}^d$ with values all c: $(c, c, \cdots, c)$. Its Euclidean length is $c\sqrt{d}$.
  - Therefore, we are dividing out the amount by which the increase in dimension increases the length of the average vectors.

- An other theoretical justification of $\sqrt{d}$:
  - if all the elements q_i and k_j are drawn independently from a $\mathcal{N}(0, \sigma^2)$ then $q_i^T k_j$ would have a variance $d\sigma^4$.
  - But e_{ij} has variance of σ^4.
  - This normalization ensures that the numbers given to softmax are not too dispersed.

Multi-head attention

Multi-head attention

- Finally, we must account for the fact that a word can mean different things to different neighbours.
  - Consider the following example: « Mary gave roses to Susan »
  - We see that the word gave has different relations to different parts of the sentence.
    - Mary expresses who's doing the giving,
    - roses expresses what's being given,
    - and Susan expresses who the recipient is.

- In a single self-attention operation, all this information just gets summed together.
  - If « Susan gave Mary the roses » instead, the output vector y_{gave} would be the same, even though the meaning has changed.

Multi-head attention

- We can give the self attention greater power of discrimination, by combining several self attention mechanisms (which we'll index with i), each with different matrices W_i^Q, W_i^K, W_i^V. These are called attention heads.

- For input x_j each attention head produces a different output vector y_j^i. We concatenate these, and pass them through a linear transformation W^O to reduce the dimension back to d.

Multi-Head Attention

- Just concatenate all the heads and apply an output projection.

$$\text{head}_i = \text{Attention}\left(W_i^Q x, W_i^K x, W_i^V x\right)$$
$$\text{MultiHeadedAttention}(\mathbf{x}) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)\boldsymbol{W}^O$$

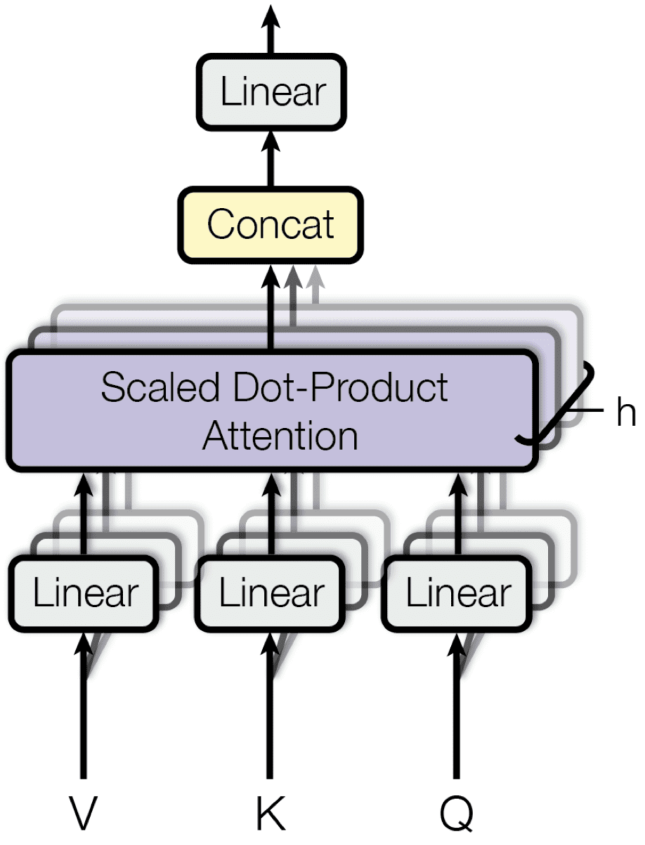
- Previously, we used the following dimensions for **single**-head SA:

$$W^Q \in \mathbb{R}^{d\times d}, \qquad W^K \in \mathbb{R}^{d\times d}, \qquad W^V \in \mathbb{R}^{d\times d},$$

- In practice, we use a reduced dimension for each head.

$$W_i^Q \in \mathbb{R}^{d\times\frac{d}{h}}, \qquad W_i^K \in \mathbb{R}^{d\times\frac{d}{h}}, \qquad W_i^V \in \mathbb{R}^{d\times\frac{d}{h}}, \qquad W^O \in \mathbb{R}^{d\times d}$$

- The total computational cost is similar to that of single-hear attention with full dimensionality.



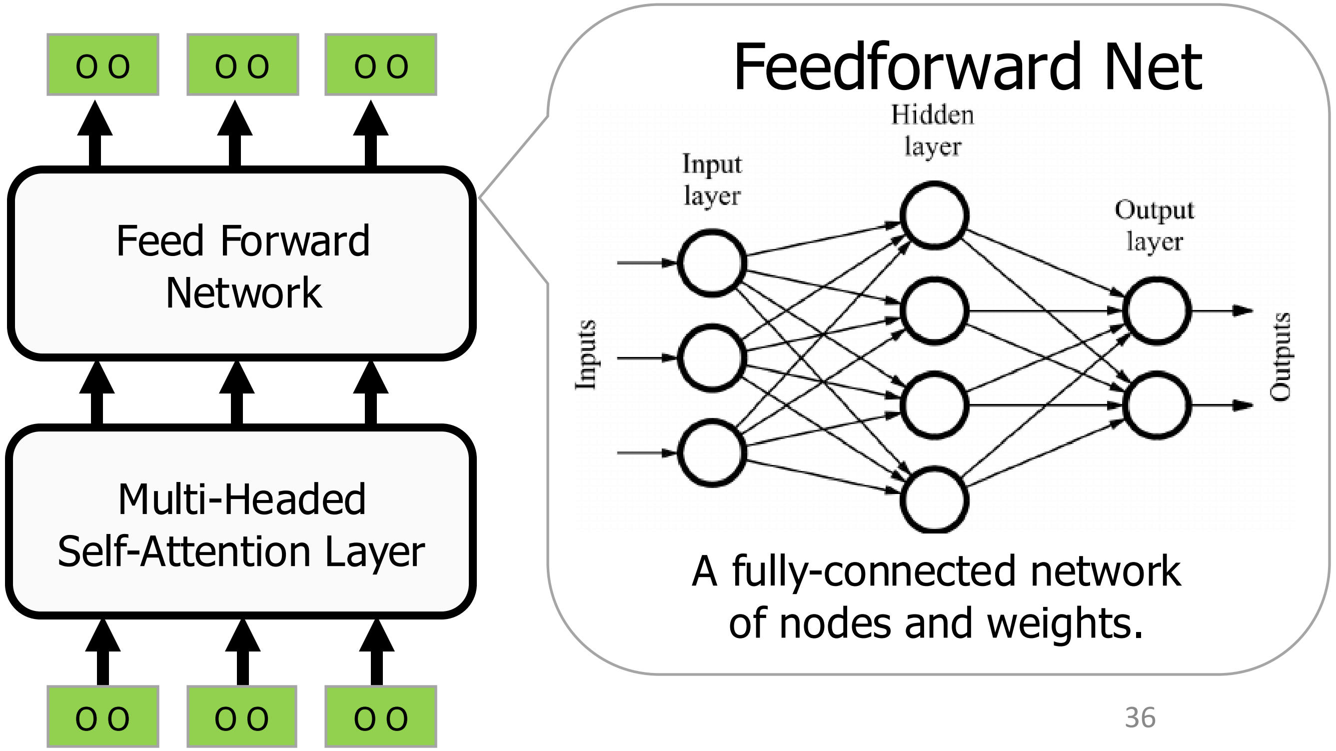
h: number of heads
d: feature dimension in output of SA

Combine with FFN

- Add a feed-forward network to add more expressivity.
  - This applies another nonlinearity to the representations (or "post-process" them).

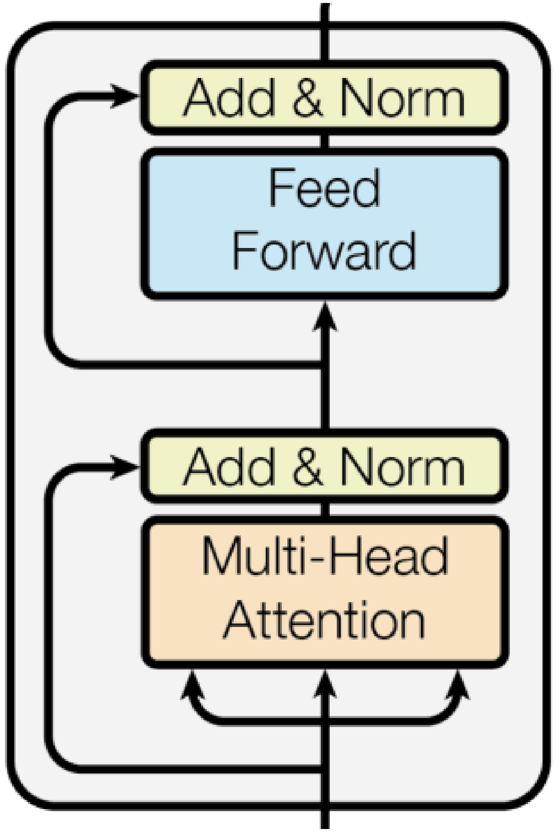
$$\mathrm{FFN}(x) = \sigma(x\boldsymbol{W}_1 + b_1)\boldsymbol{W}_2 + \boldsymbol{b}_2$$
$$\boldsymbol{W}_1 \in \mathbb{R}^{d \times d_{\mathrm{ff}}},$$
$$\boldsymbol{W}_2 \in \mathbb{R}^{d_{\mathrm{ff}} \times d}$$

  - Usually, the dimensionality of the hidden feedforward layer d_{ff} is 2-8 times larger than the input dimension d.



How Do We Prevent Vanishing Gradients?

- Residual connections let the model “skip” layers
  - These connections are particularly useful for training deep networks
- Use layer normalization to stabilize the network and allow for proper gradient flow



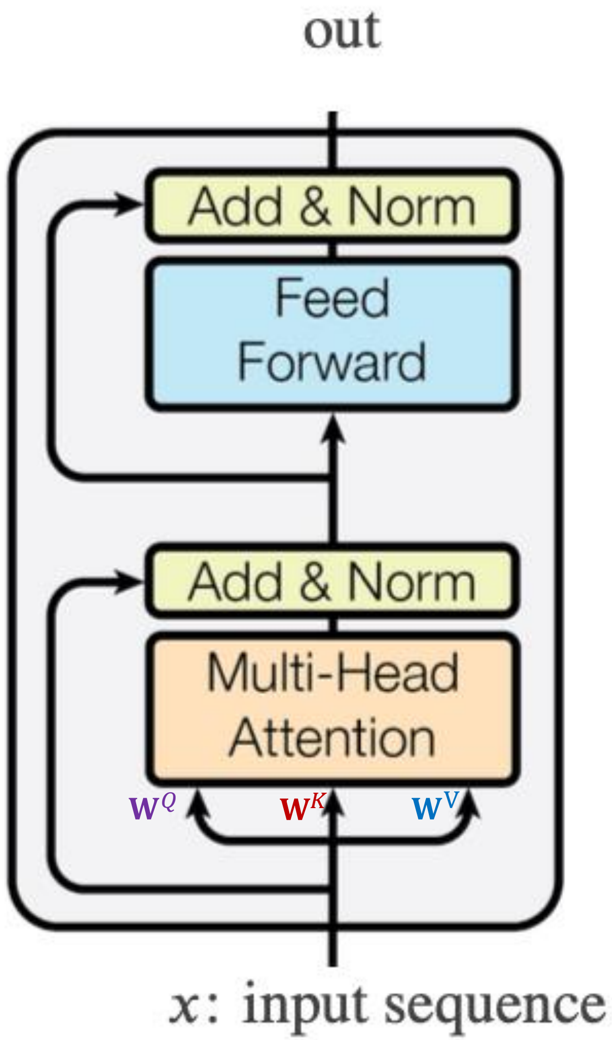
Attention Is All You Need, Vaswani et al. 2017

Putting it Together: Self-Attention Block

- Layer normalization is similar to batch normalization (but it is not strictly speaking a batch normalization)
- Layer normalization prevents the range of values in the layers from changing too much, which allows faster training and better generalization ability
- Given input x:

$$\boldsymbol{z} = LayerNorm\ (\tilde{\boldsymbol{x}} + \boldsymbol{x})\tilde{\boldsymbol{x}}$$
$$= \mathrm{MultiHeadedAttention}(\boldsymbol{x}; \mathbf{W}^{Q}, \mathbf{W}^{K}, \mathbf{W}^{V})$$

$$\tilde{\boldsymbol{z}} = \mathrm{FFN}(\boldsymbol{z}) = \sigma(\boldsymbol{z}W_1 + b_1)W_2 + b_2$$
$$\mathrm{out} = LayerNorm(\tilde{\boldsymbol{z}} + \boldsymbol{z})$$



LayerNorm$\left(x + \text{Sublayer}(x)\right)$

- For a batch $\{x_n\}_{n=1,\dots,N}$ of N vectors $x_n \in \mathbb{R}^K$, also written as $\{x_{n,k}\} \in \mathbb{R}^{N\times K}$, the expectation and variance accros spatial dimensions are « estimated » by

$$\mu_n = \frac{1}{K}\sum_{k=1}^{K} x_{n,k} \in \mathbb{R}, \qquad \sigma_n^2 = \frac{1}{K}\sum_{k=1}^{K}\left(x_{n,k} - \mu_n\right)^2 \in \mathbb{R}$$

- Layer Normalization (LayerNorm in Pytorch)

$$\hat{x}_{n,k} = \frac{x_{n,k} - \mu_n}{\sqrt{\sigma_n^2 + \epsilon}} \in \mathbb{R} \quad \Rightarrow \quad \hat{x}_n = \begin{pmatrix} \hat{x}_{n,1} \\ \vdots \\ \hat{x}_{n,K} \end{pmatrix} \in \mathbb{R}^K$$

$$LN_{\gamma,\beta}(x_n) = \gamma \hat{x}_n + \beta$$

- γ and β are learnable affine transform parameters

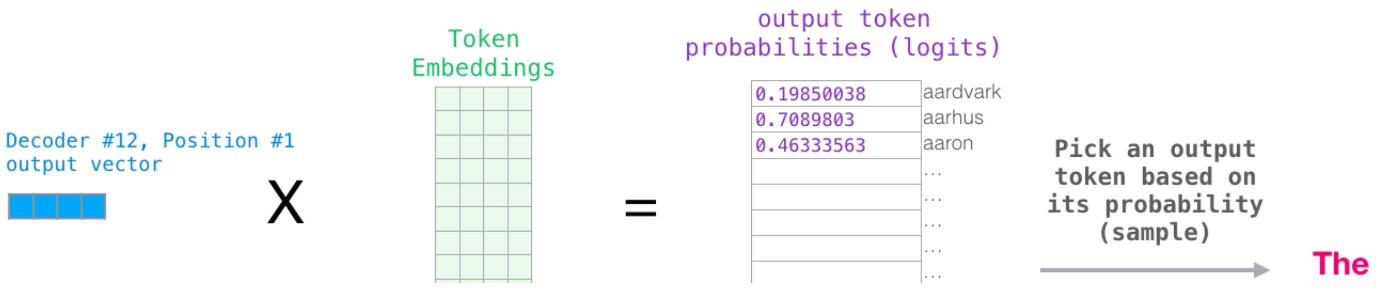
https://theaisummer.com/normalization/

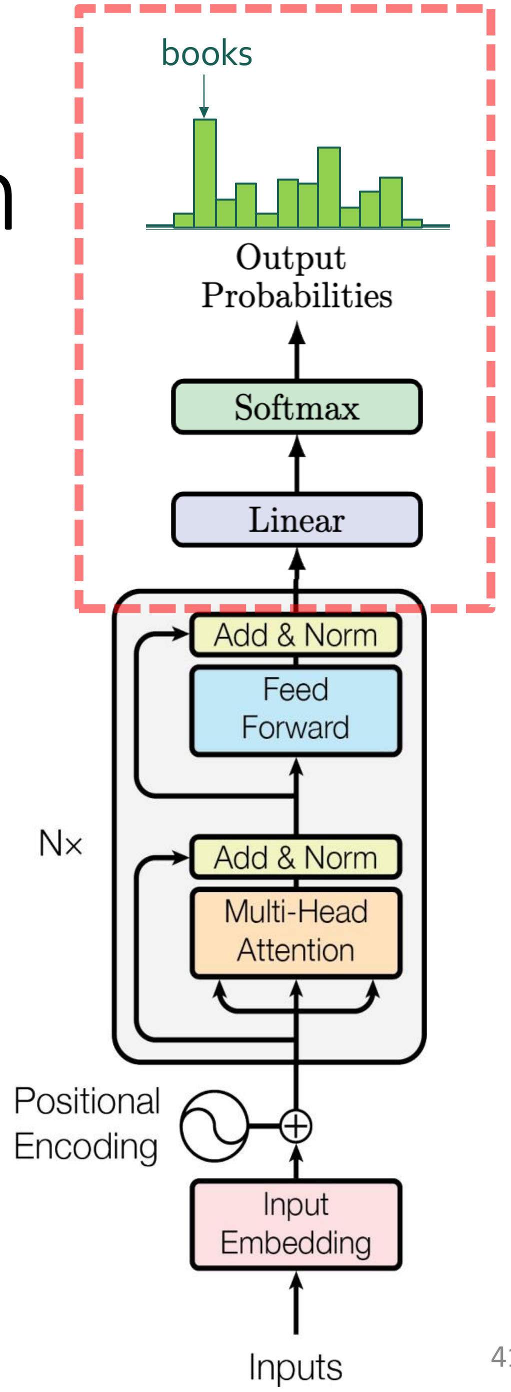
Decoder-Only Transformer

From Representations to Prediction

- Use sequentially N Multi-Head attention modules.
- To perform prediction, add a classification head on top of the final layer of the transformer: $\boldsymbol{x} \in \mathbb{R}^{n \times d}$.
    - n is the length of the input sequence.
- To obtain logits, we can apply a linear transformation with token embedding matrix $W^S \in \mathbb{R}^{d \times V}$
- To obtain probabilities, run this through softmax:

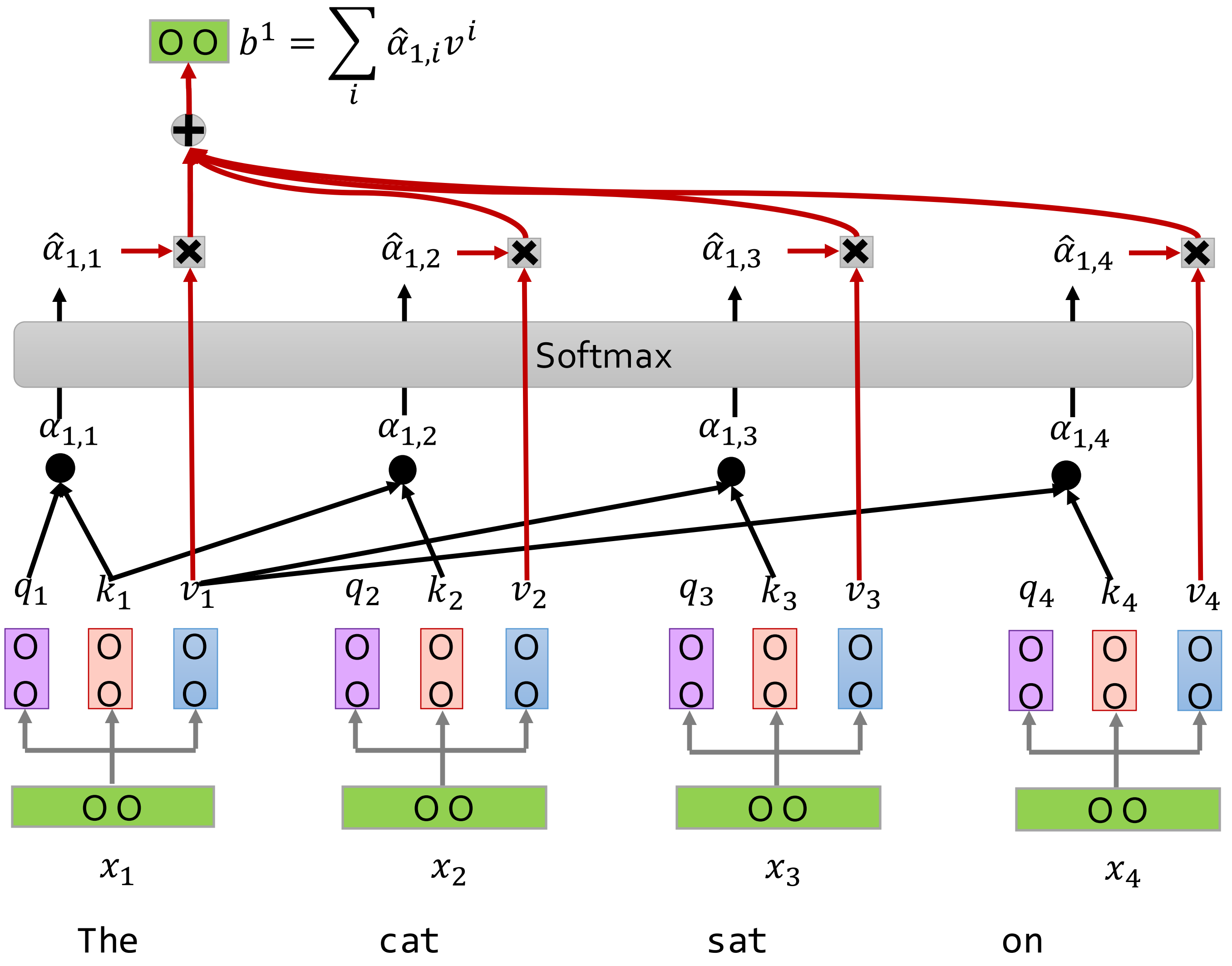
$$\text{softmax}(\boldsymbol{x}W^S) \in \mathbb{R}^{n \times V}$$



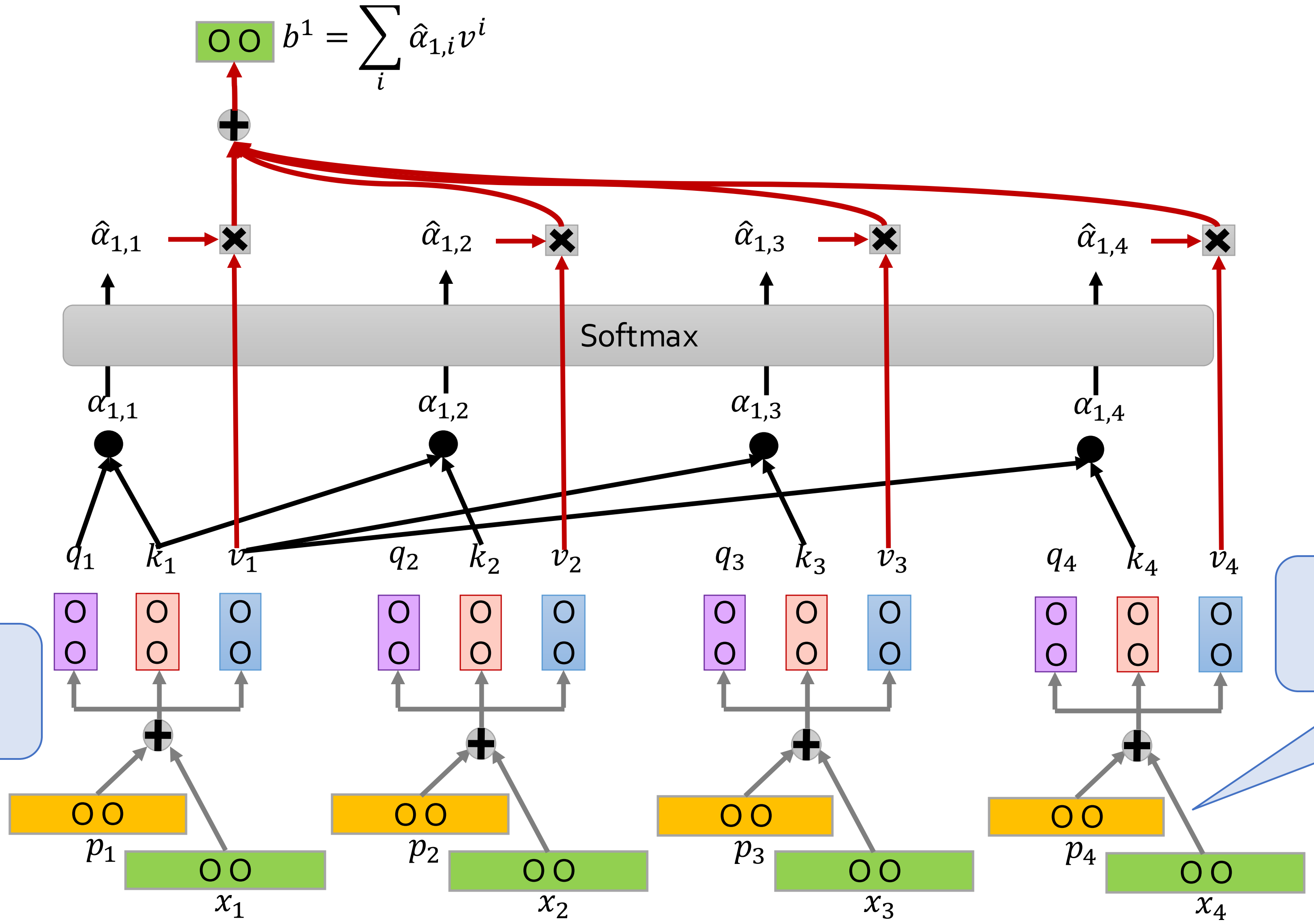


Input encoding

Positional Embeddings



Positional Embeddings



p_i are positional embeddings

Allows model to learn about positions

Criteria for Positional Encodings

- The first idea that might come to mind is to assign a number to each time-step within the $[0, 1]$ range in which 0 means the first word and 1 is the last time-step.
  - One of the problems it will introduce is that you can't figure out how many words are present within a specific range. In other words, time-step doesn't have consistent meaning across different sentences.

- Another idea is to assign a number to each time-step linearly: the first word is given "1", the second word is given "2", and so on.
  - The problem with this approach is that not only the values could get quite large, but also our model can face sentences longer than the ones in training.
  - In addition, our model may not see any sample with one specific length which would hurt generalization of our model.

- Ideally, the following criteria should be satisfied:
  - It should output a unique encoding for each time-step (word's position in a sentence)
  - Distance between any two time-steps should be consistent across sentences with different lengths.
  - The model should generalize to longer sentences without any efforts. Its values should be bounded.
  - It must be deterministic.

Proposed Method for Transformer

- The encoding proposed by the authors satisfies all of those criteria.
  - First of all, it isn't a single number. Instead, it's a d-dimensional (same dimension as word embedding) vector p_t that contains information about a specific position t in a sentence.
  - Secondly, this encoding is not integrated into the model itself. Instead, this vector is used to equip each word with information about its position in a sentence.
  - Accoding to the authors, for any fixed offset s, p_{t+s} can be represented as a linear function of p_t

- Let t the desired position in an input sentence, $p_t = \big(p_t(0), \dots, p_t(d-1)\big) \in \mathbb{R}^d$ be its corresponding encoding. Then,

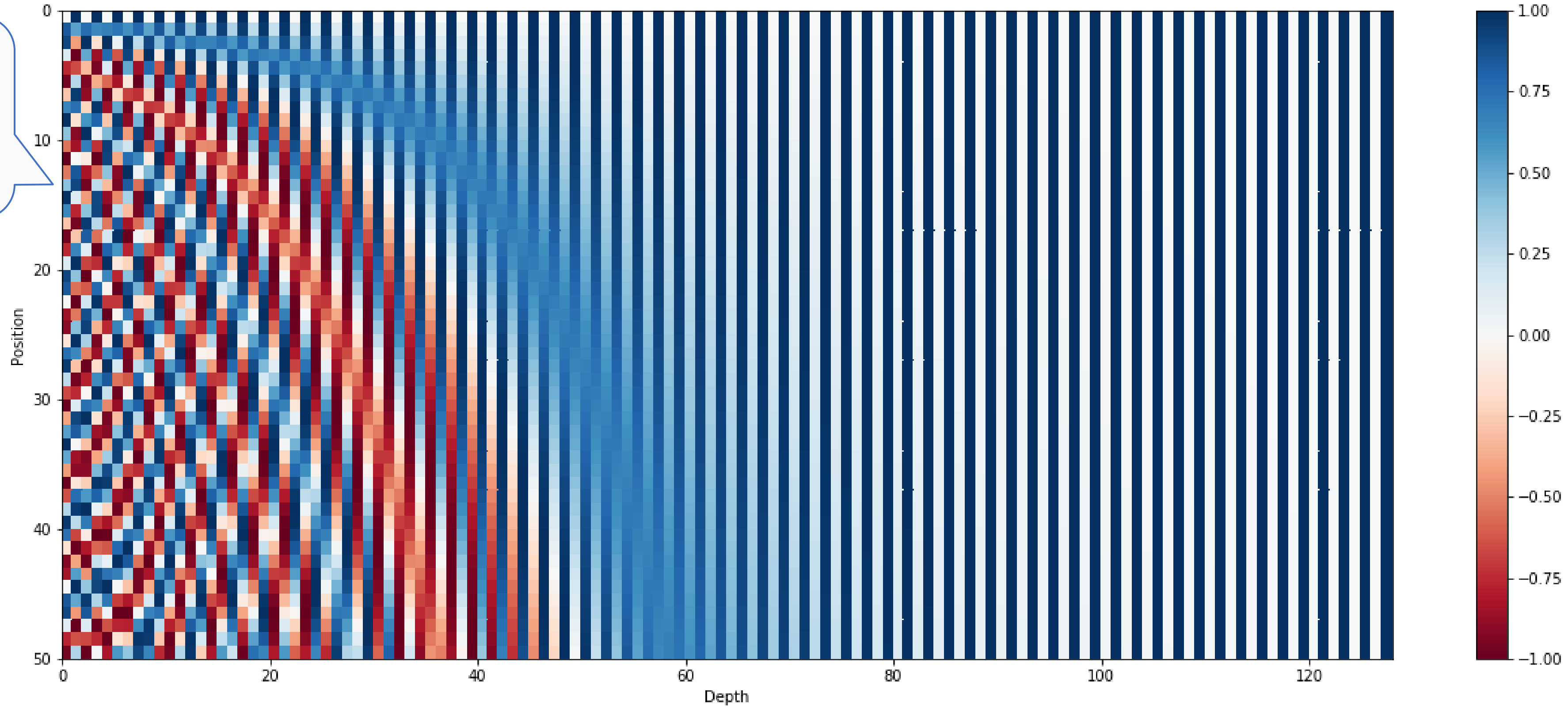
$$p_t(i) = \begin{cases} \sin(w_k t) & \text{if } i = 2k \\ \cos(w_k t) & \text{if } i = 2k+1 \end{cases} \quad \text{with} \quad w_k = \frac{1}{10000^{2k/d}}$$

- Example: $p_t(0) = \sin\left(\frac{t}{10000^{0/d}}\right), p_t(1) = \cos\left(\frac{t}{10000^{0/d}}\right), p_t(2) = \sin\left(\frac{t}{10000^{2/d}}\right), p_t(3) = \cos\left(\frac{t}{10000^{2/d}}\right), \dots$

Visualizing the Positional Encodings

- The 128-dimensional positonal encoding for a sentence with the maximum length of 50.
- Each row represents the embedding vector p_t

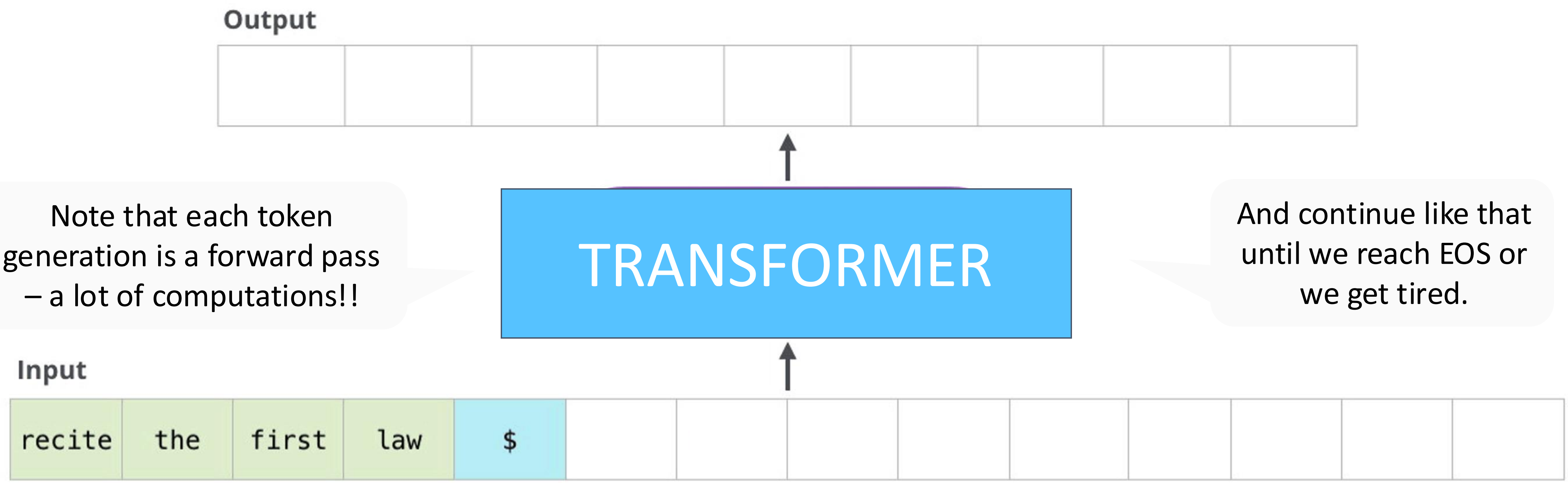
Any pair of rows are different!



https://kikaben.com/transformers-positional-encoding/

How does training work?

Generating text via Transformer



Note that each token generation is a forward pass – a lot of computations!!

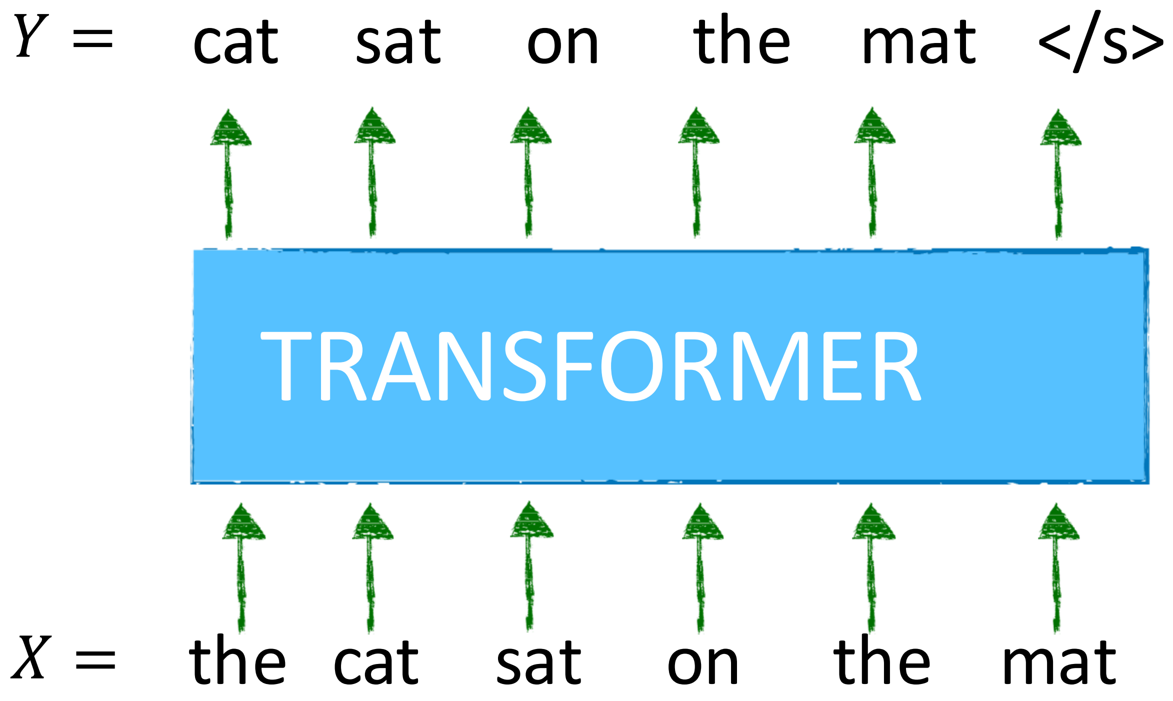
And continue like that until we reach EOS or we get tired.

Image by http://jalammar.github.io/illustrated-gpt2/

Training a Transformer Language Model

- **Goal:** Train a Transformer for language modeling (i.e., predicting the next word).
- **Approach:** Train it so that each position is predictor of the next (right) token.
  - We just shift the input to right by one, and use as labels

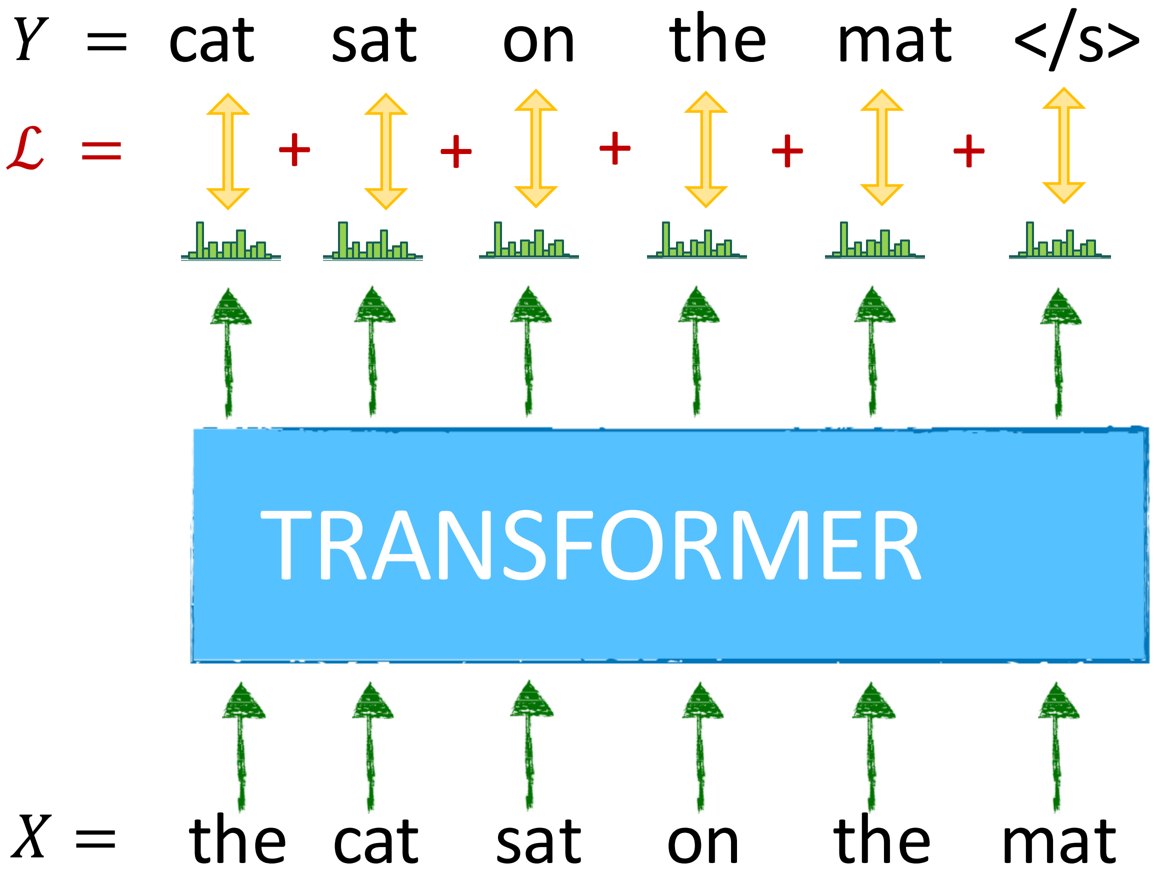
EOS special token




```
X = text[:, :-1]
Y = text[:, 1:]
```

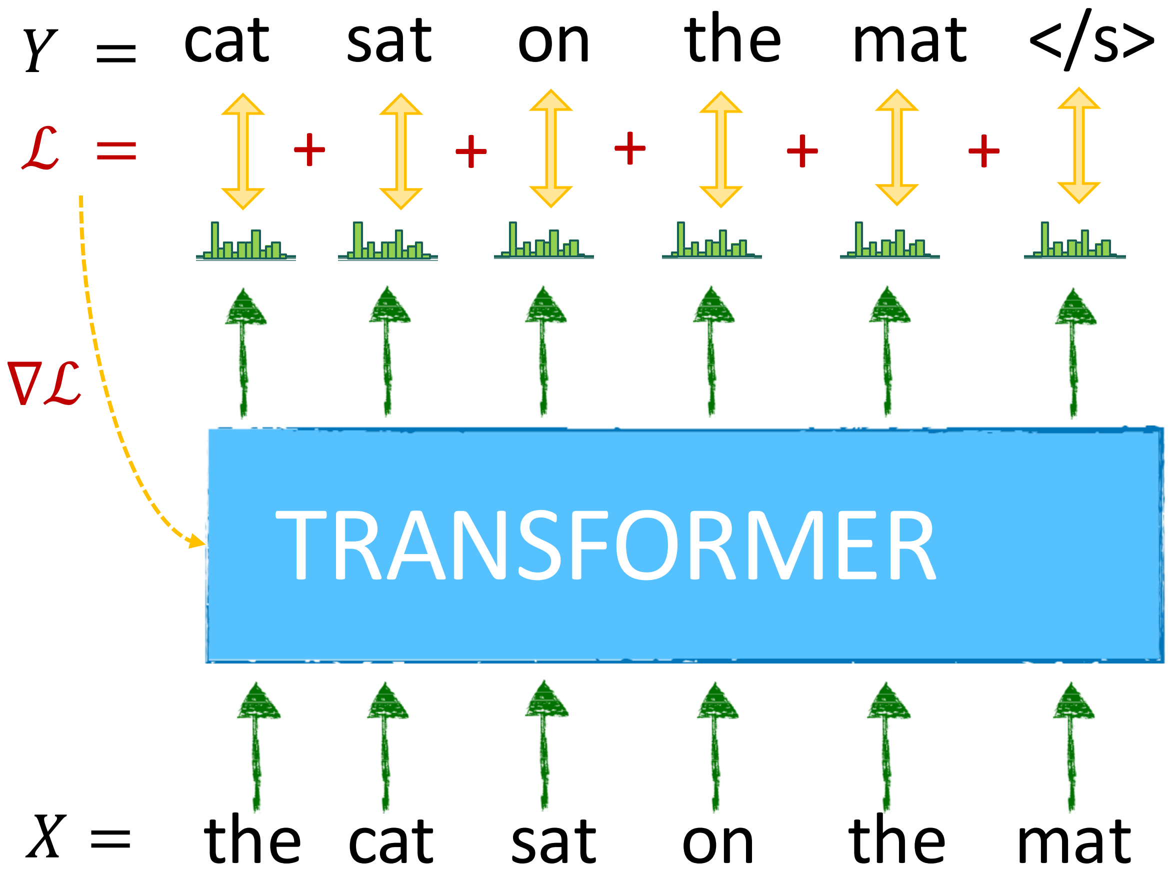
Training a Transformer Language Model

- Sum the position-wise loss values to a obtain a **global loss**.

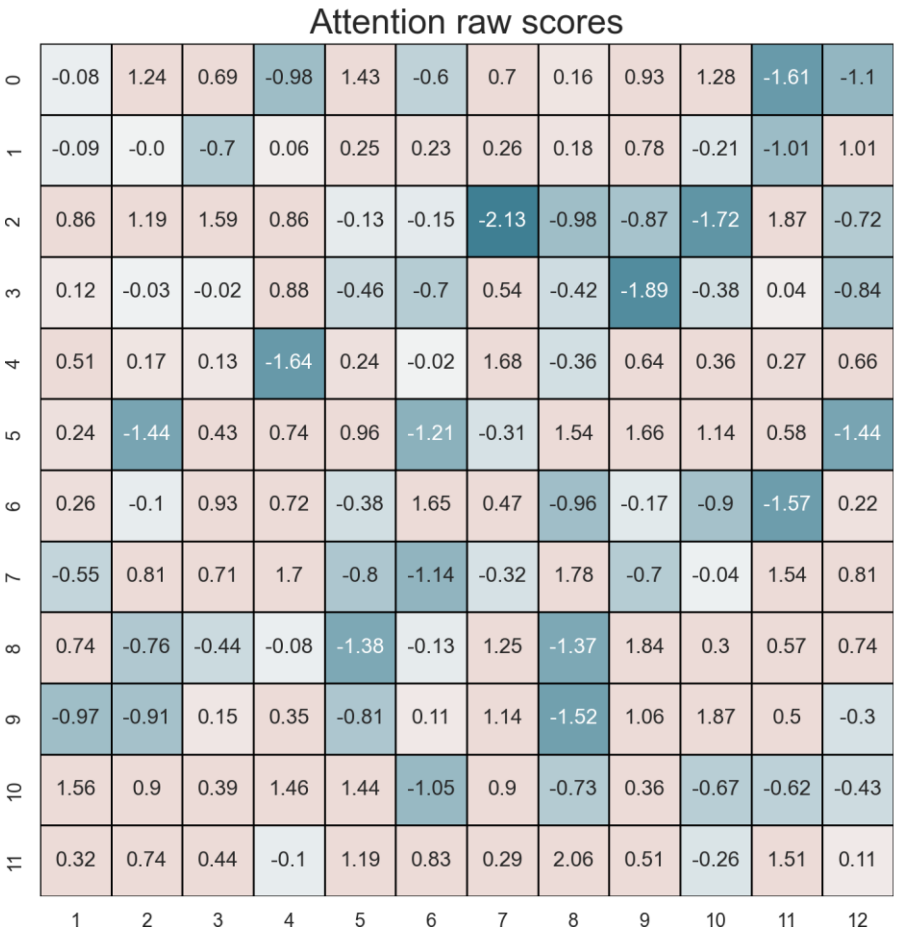


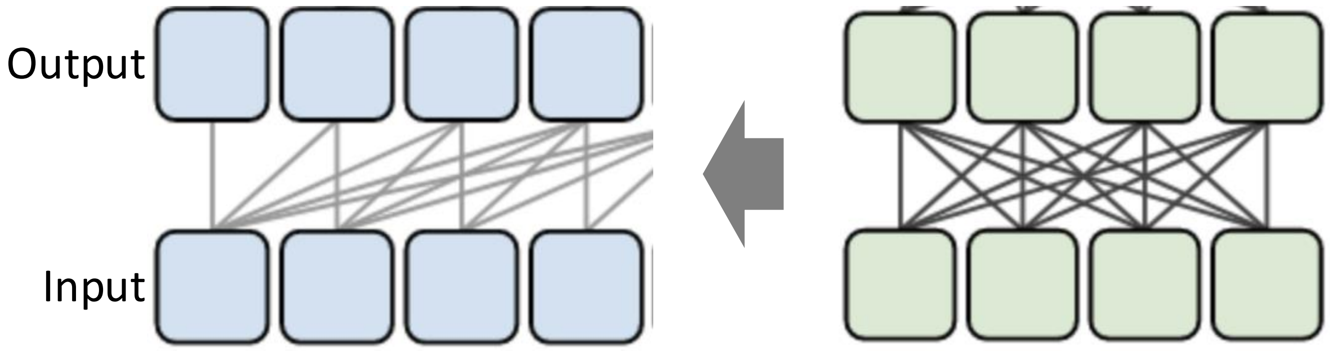
Training a Transformer Language Model

- The model would solve the task by copying the next token to output (data leakage) since we process the input sequence as a whole.
- Does not learn anything useful



Attention mask

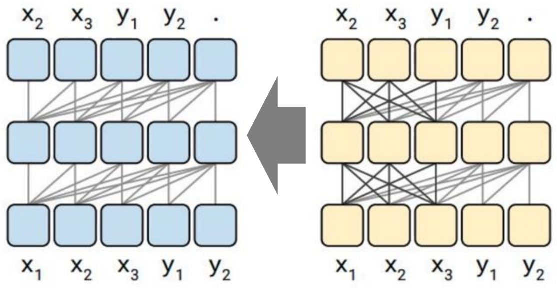




What we want

What we have

Attention mask



Attention mask

SCALED SCORES

Output \ Input	First token			Last token
First token	0.7	0.1	0.1	0.1
	0.1	0.6	0.2	0.1
	0.1	0.3	0.6	0.1
Last token	0.1	0.3	0.3	0.3

Output

Input

X

Element-wise product

LOOK-AHEAD MASK

1	-inf	-inf	-inf
1	1	-inf	-inf
1	1	1	-inf
1	1	1	1

=

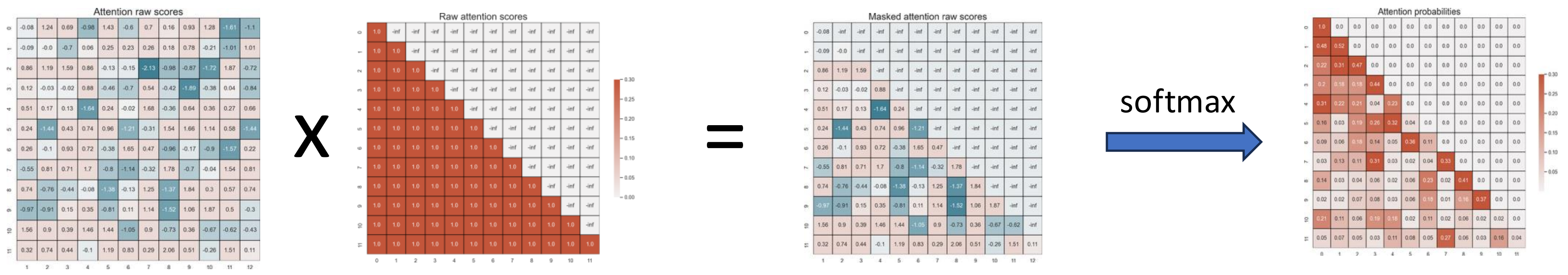
MASKED SCORES

0.7	-inf	-inf	-inf
0.1	0.6	-inf	-inf
0.1	0.3	0.6	-inf
0.1	0.3	0.3	0.3

Input

Attention masking: Why Before Softmax?

- We applied attention masking before softmax. Why not after?

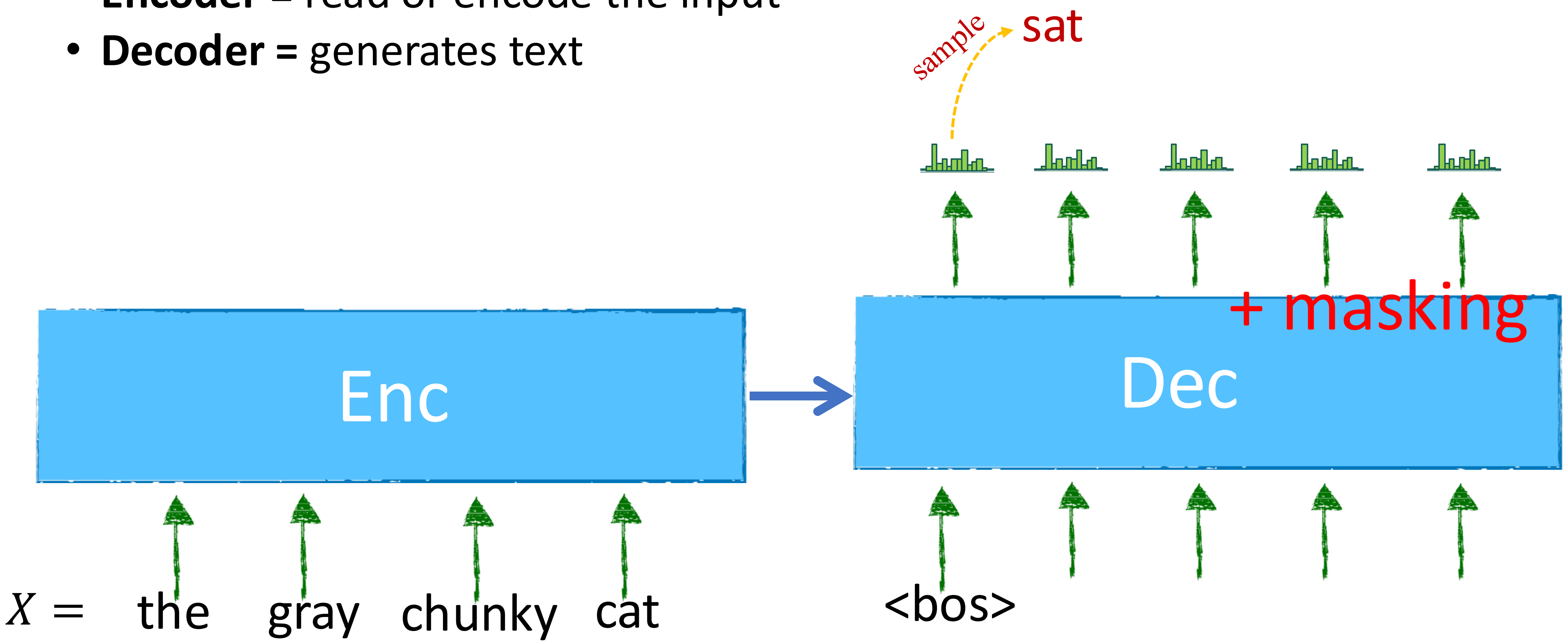


- Softmax normalizes the scores so that it’s a probability.
- Masking after softmax, would lead to an unnormalized probability distribution.

Encoder-Decoder Transformer

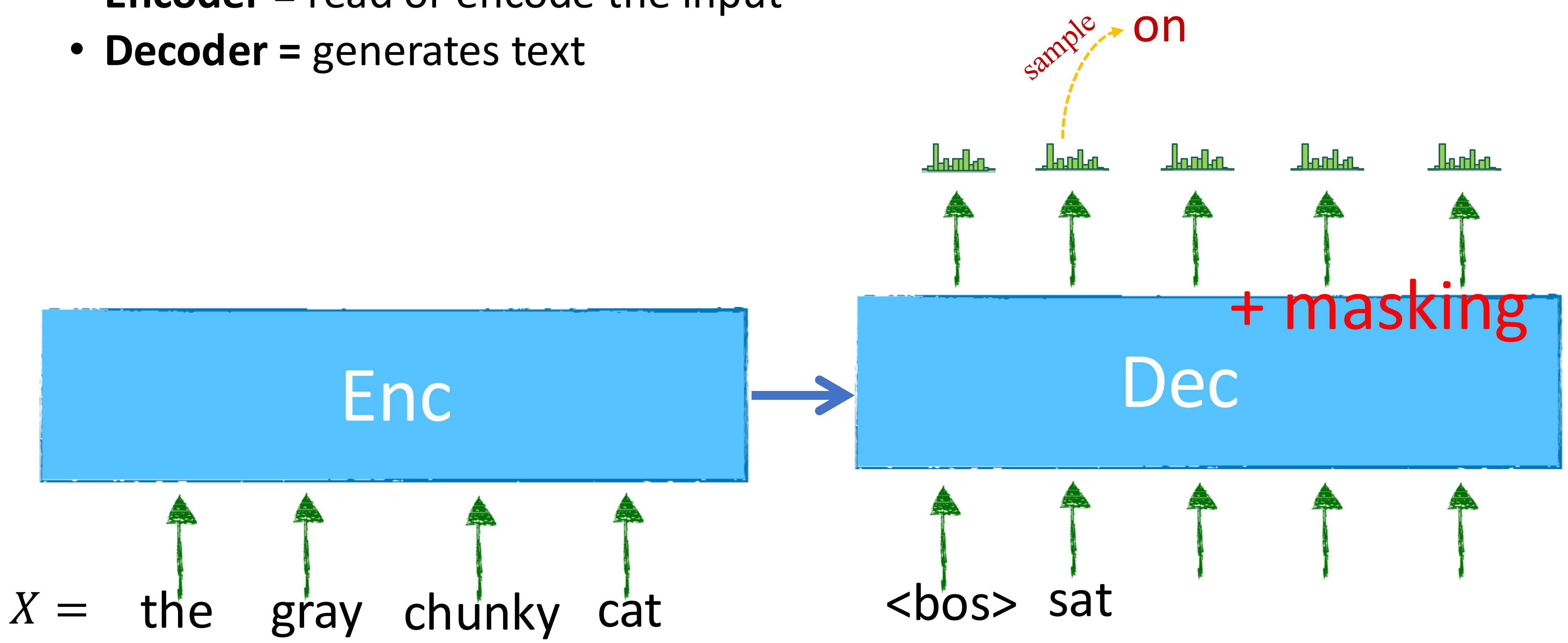
Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder =** read or encode the input
  - **Decoder =** generates text



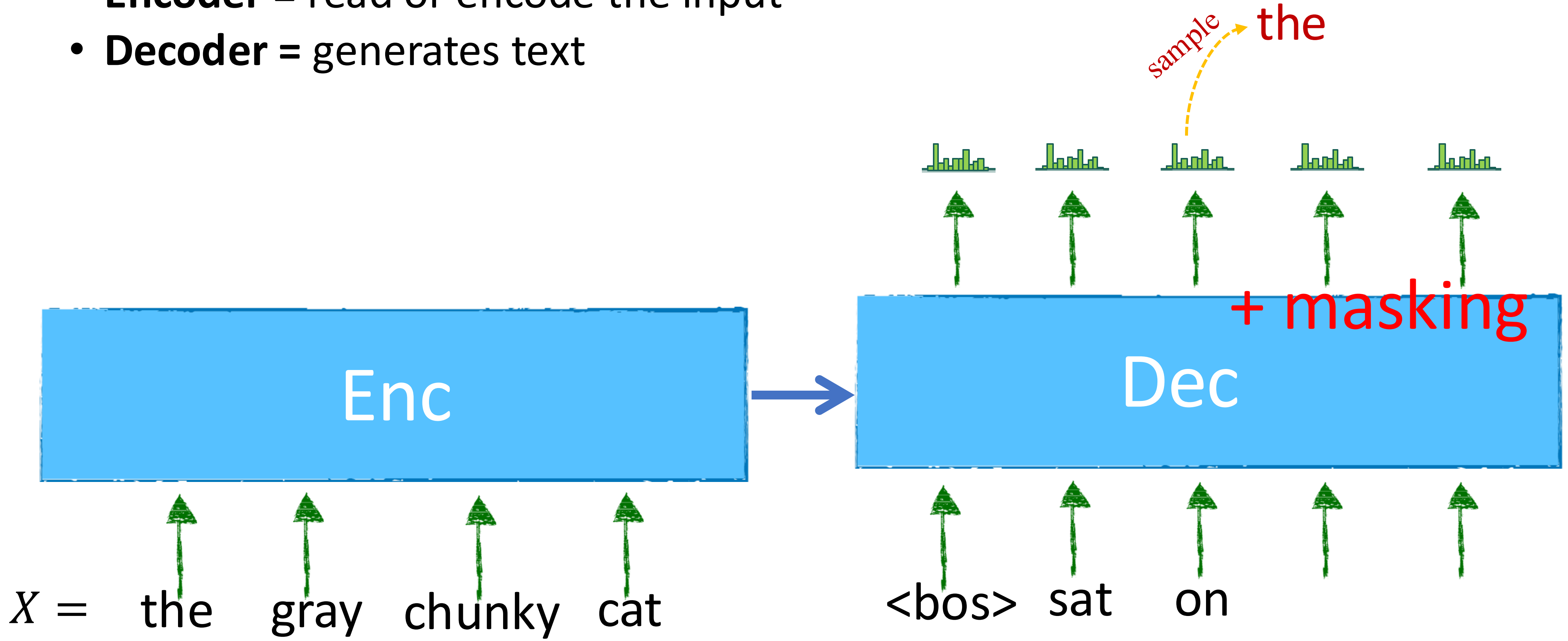
Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder =** read or encode the input
  - **Decoder =** generates text



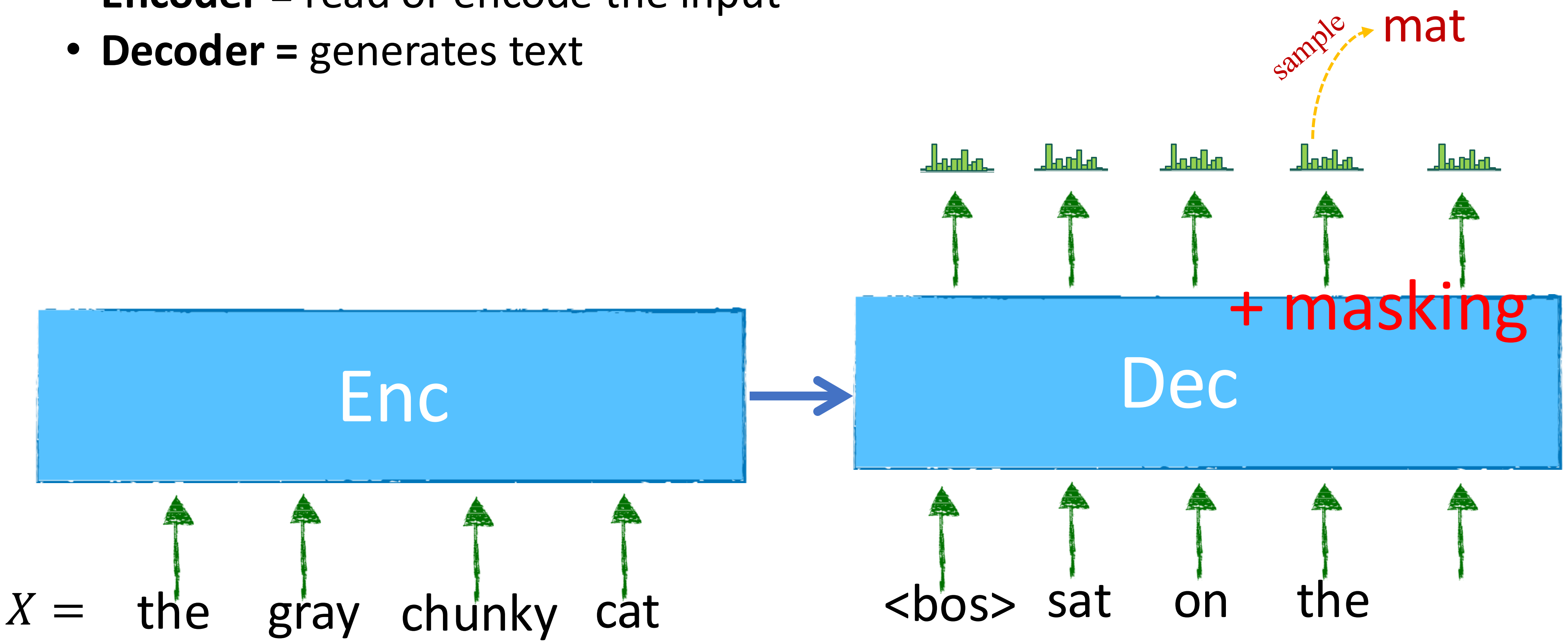
Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder =** read or encode the input
  - **Decoder =** generates text



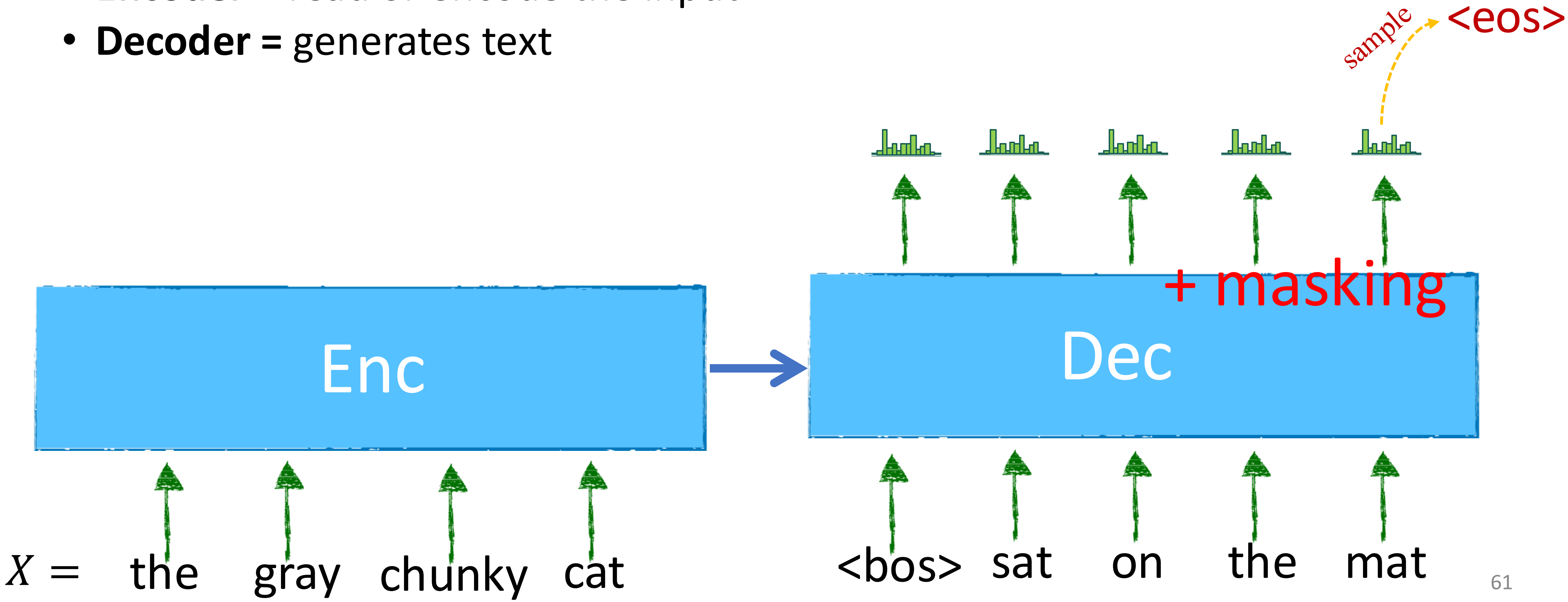
Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder =** read or encode the input
  - **Decoder =** generates text

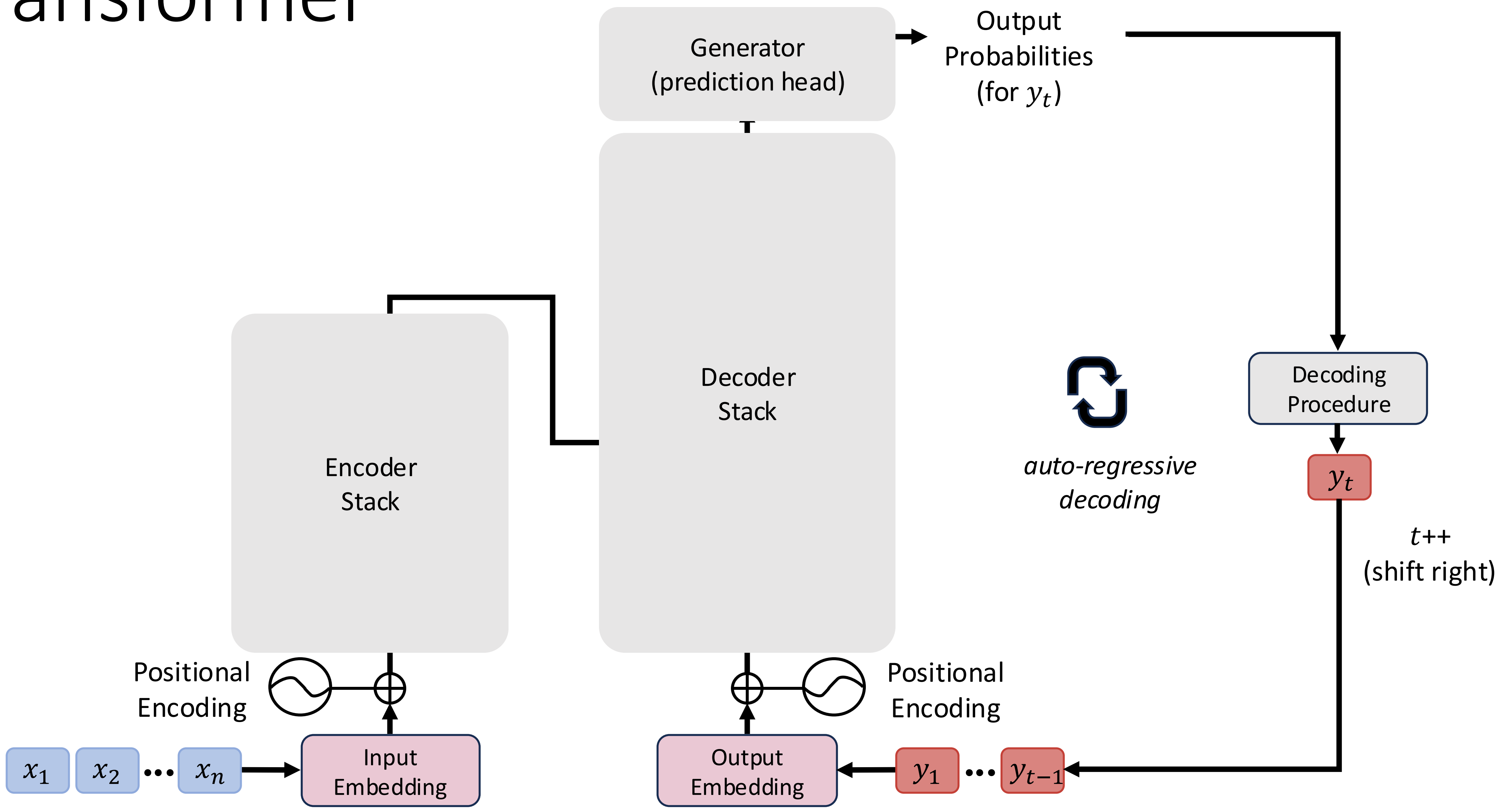


Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder =** read or encode the input
  - **Decoder =** generates text

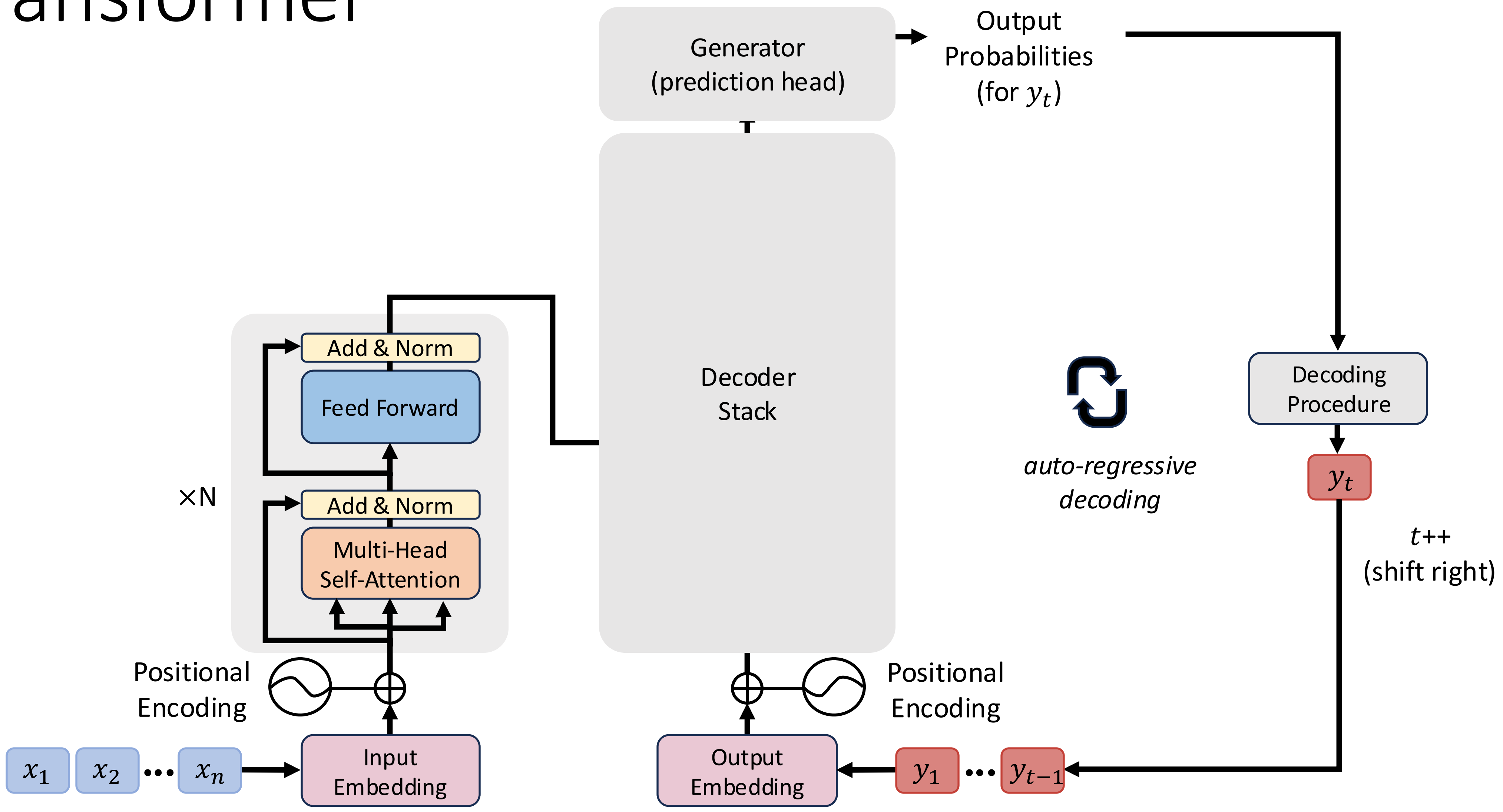


Transformer

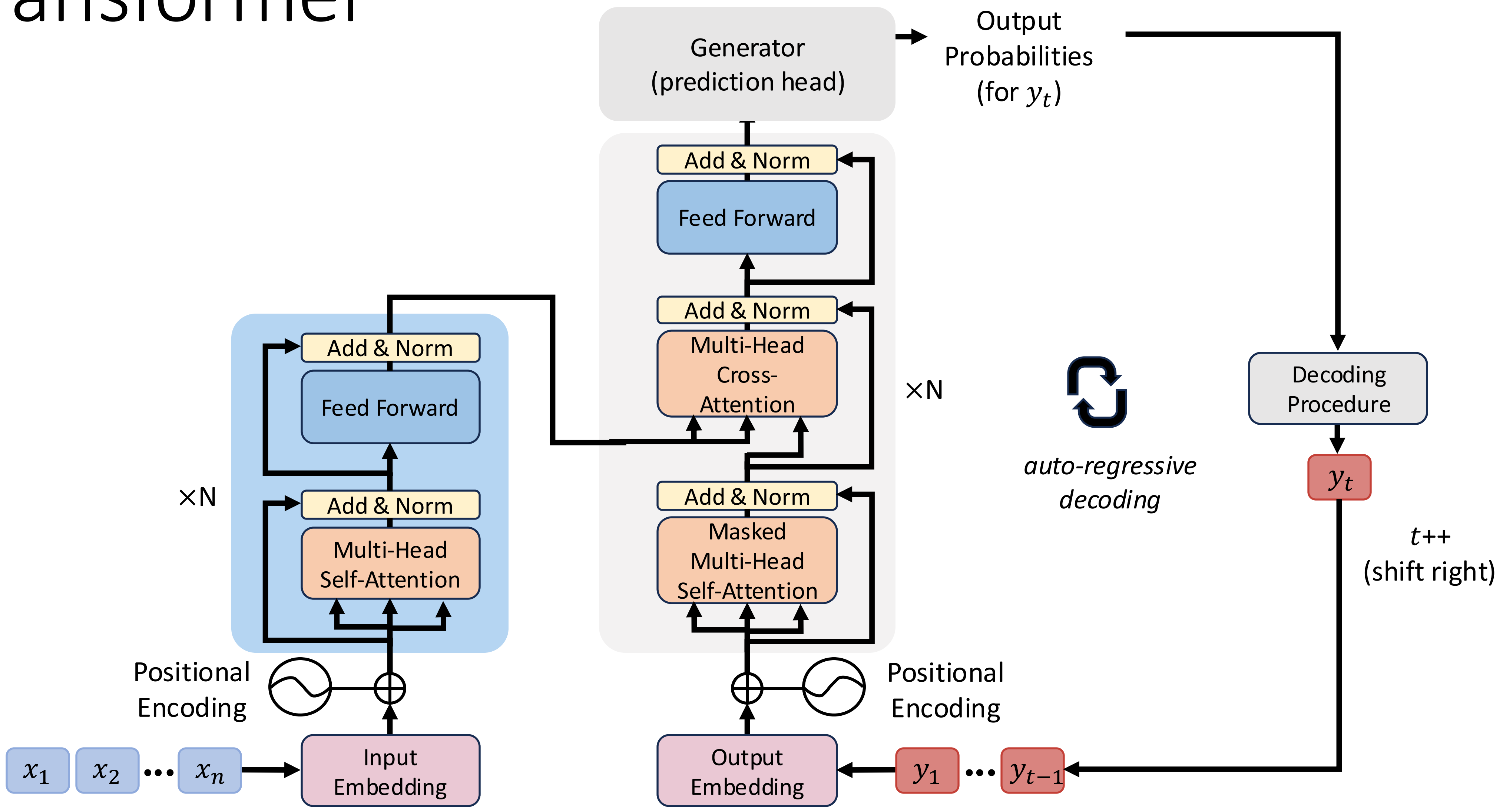


[Vaswani et al. 2017]

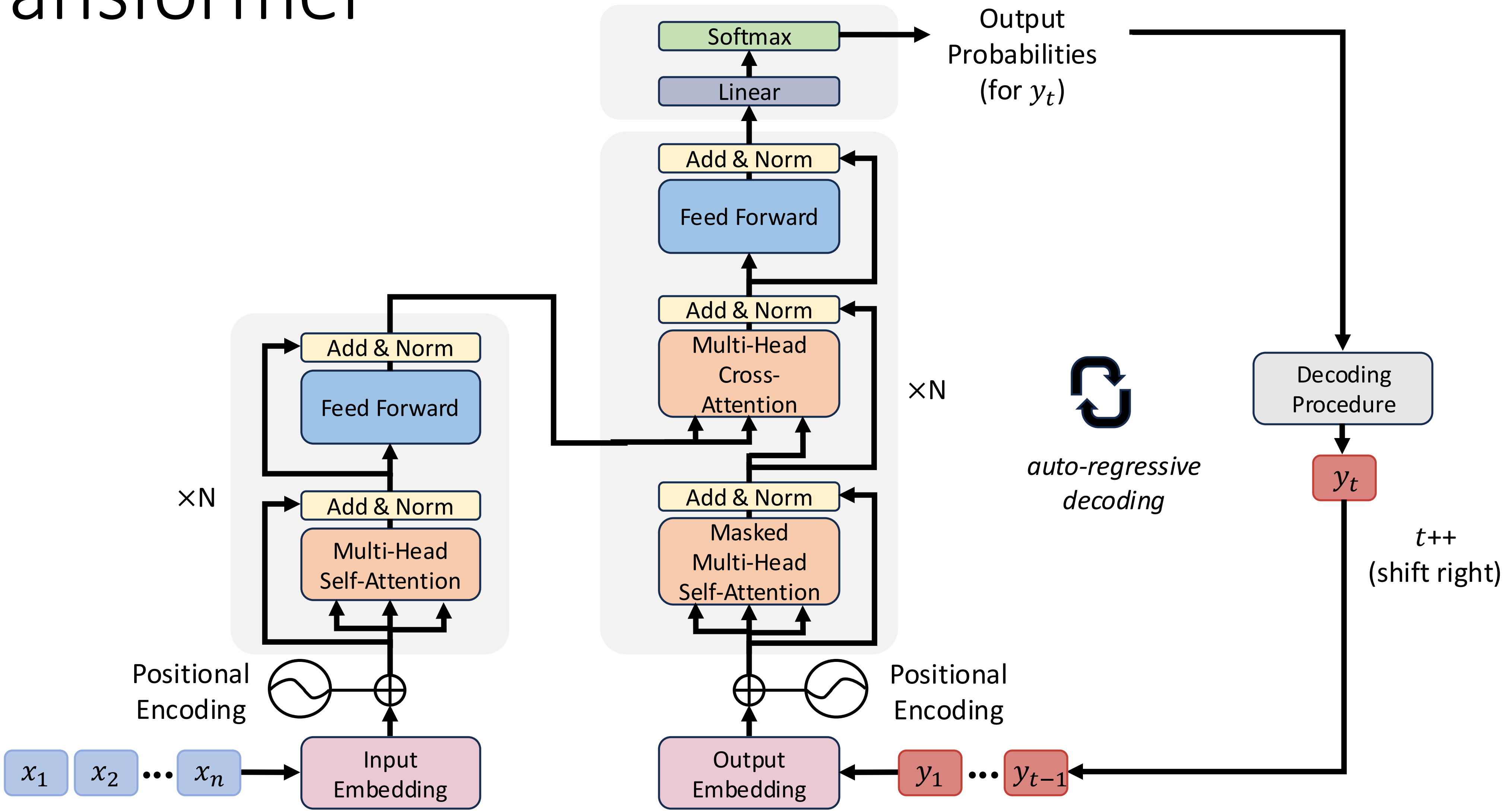
Transformer



Transformer

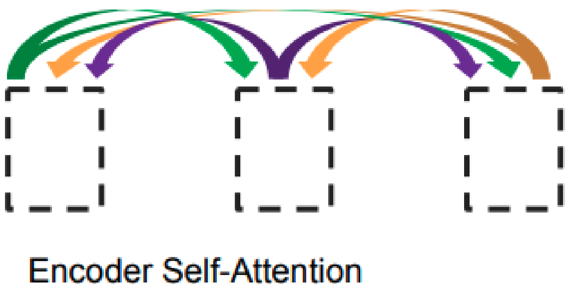


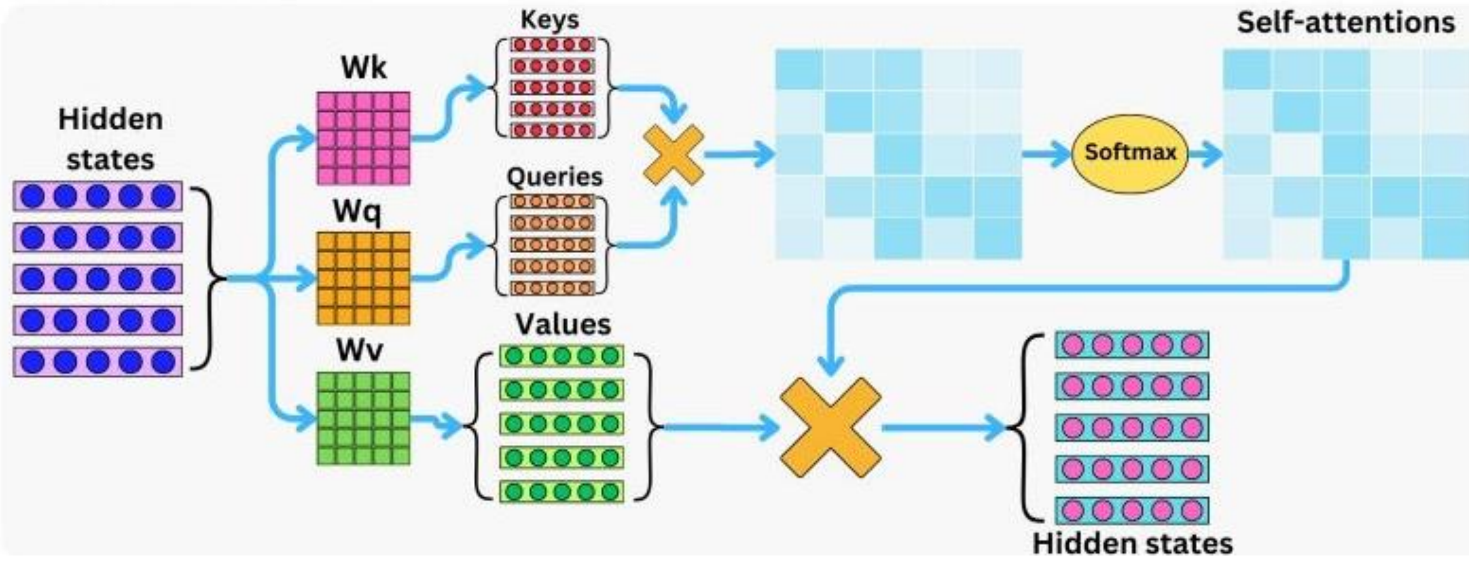
Transformer

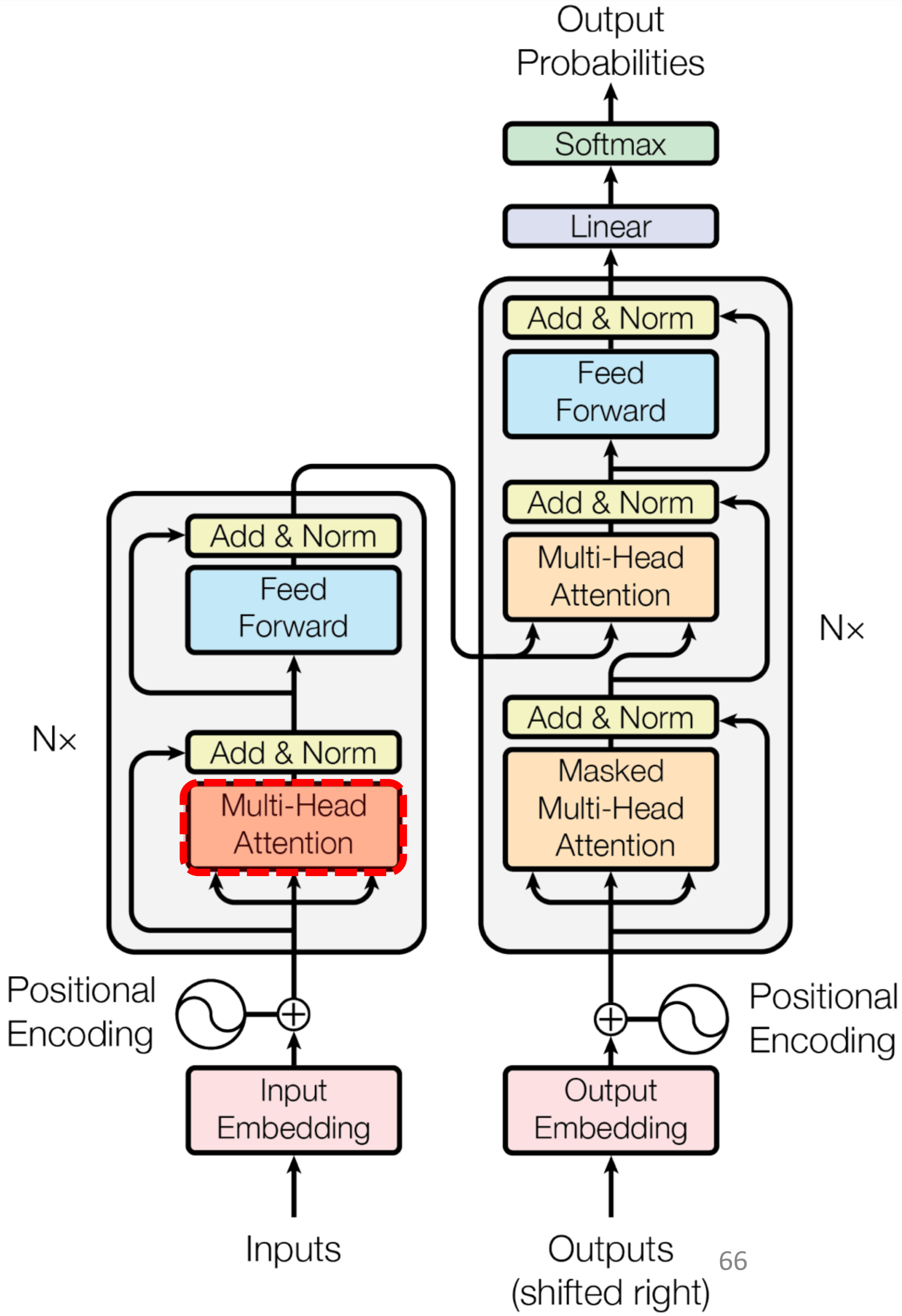


Transformer

- Computation of **encoder** attends to both sides.

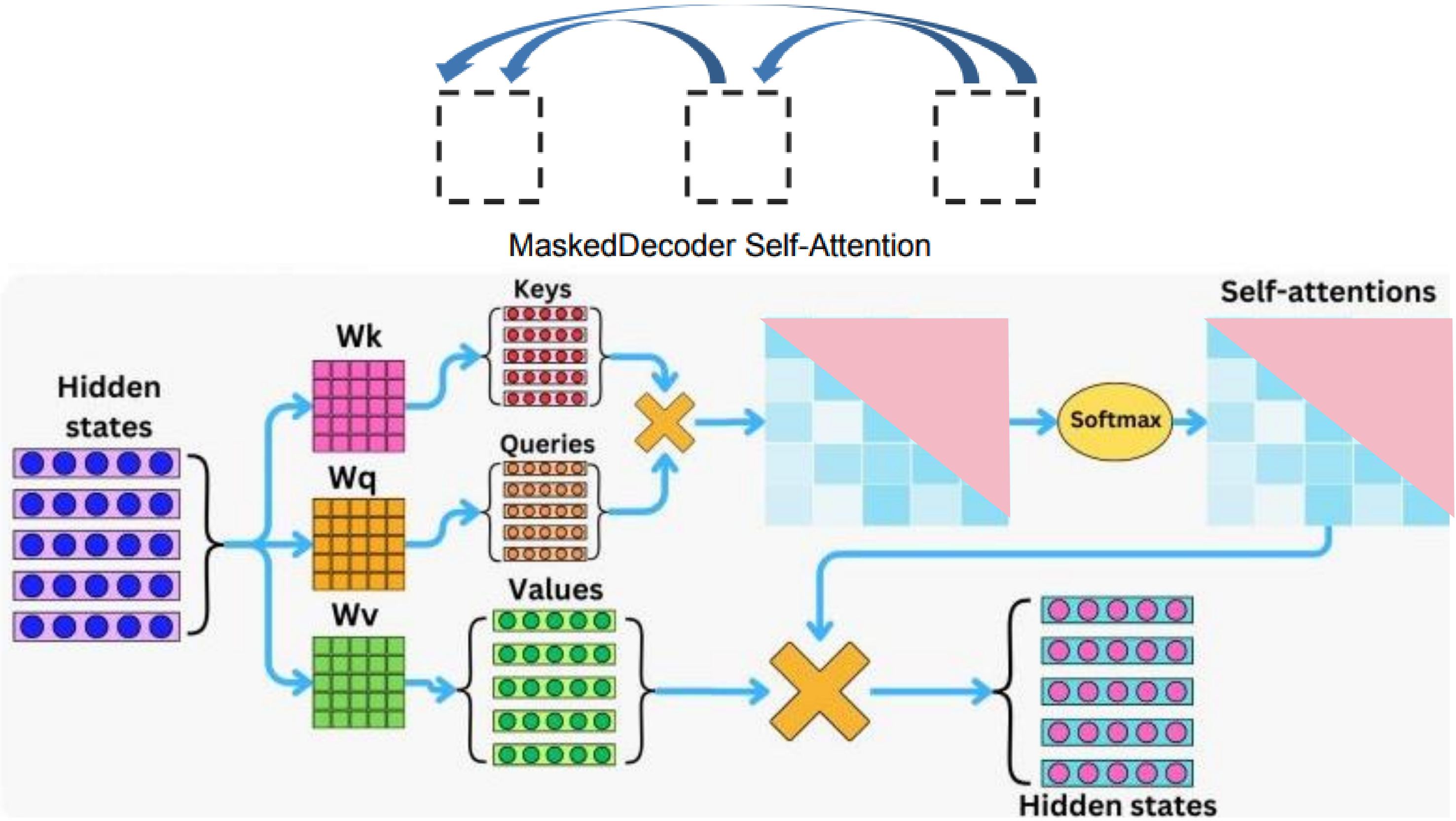


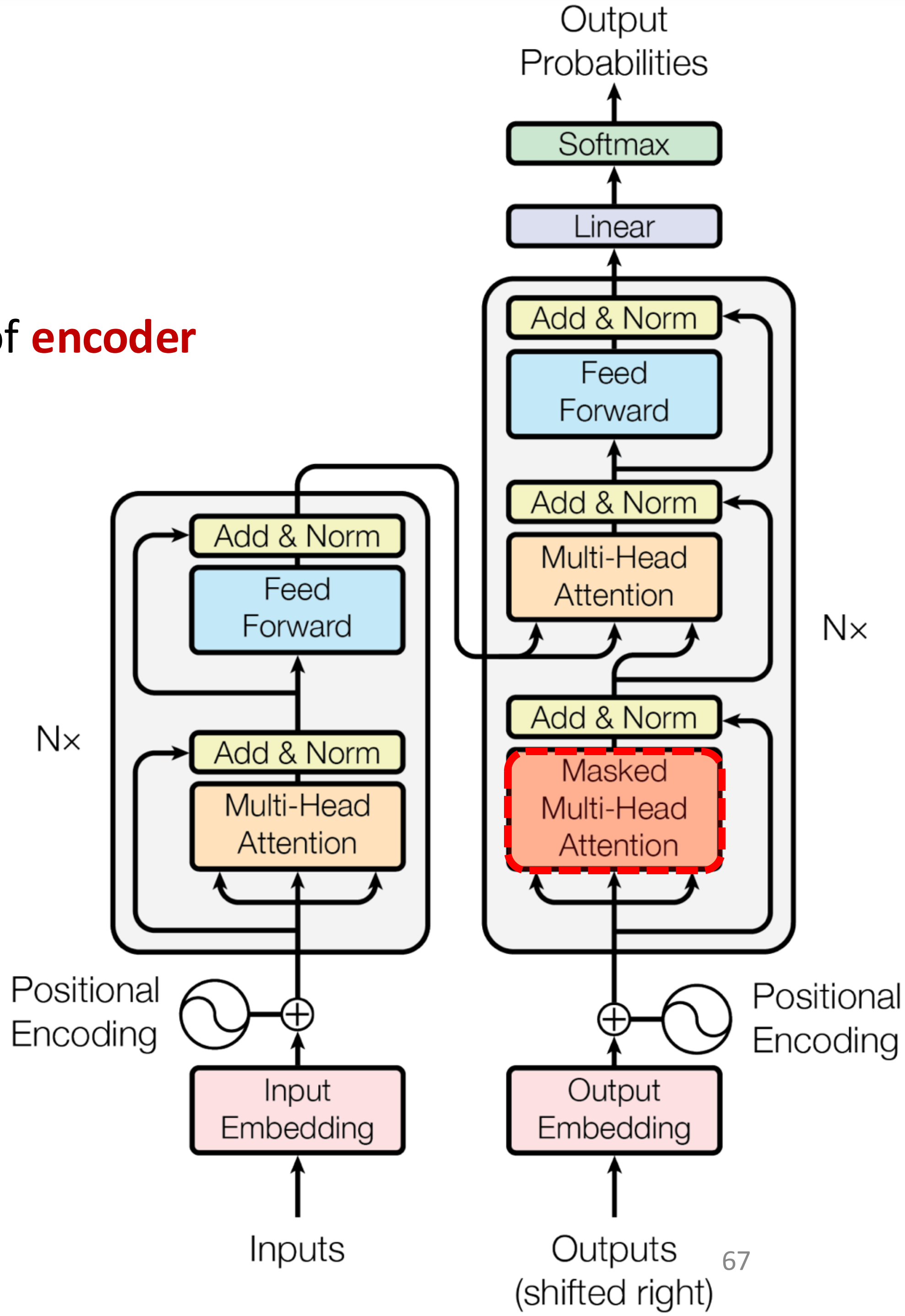




Transformer

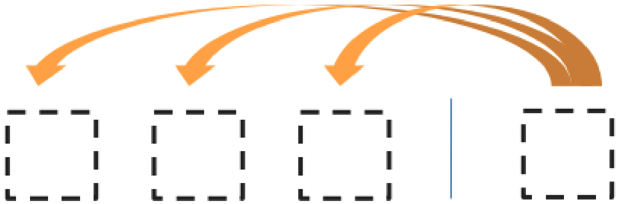
- At any step of **decoder**, it attends to previous computation of **encoder** as well as **decoder's** own generations.



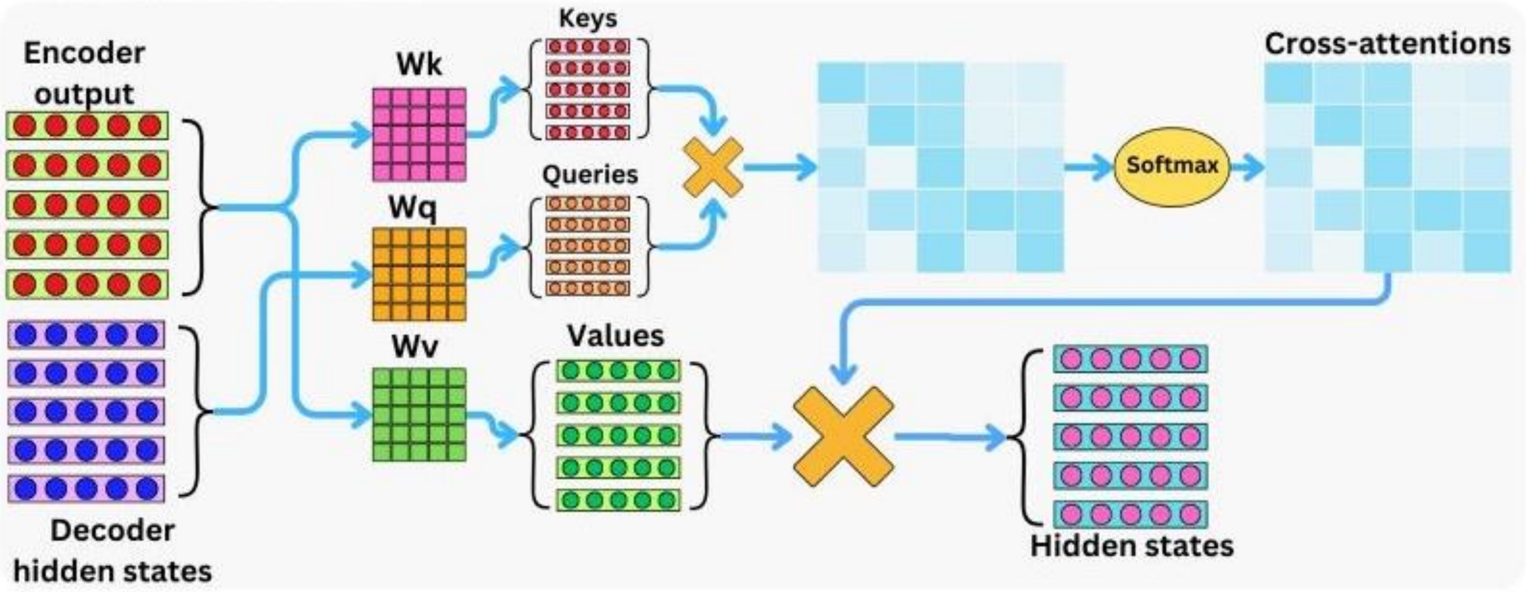


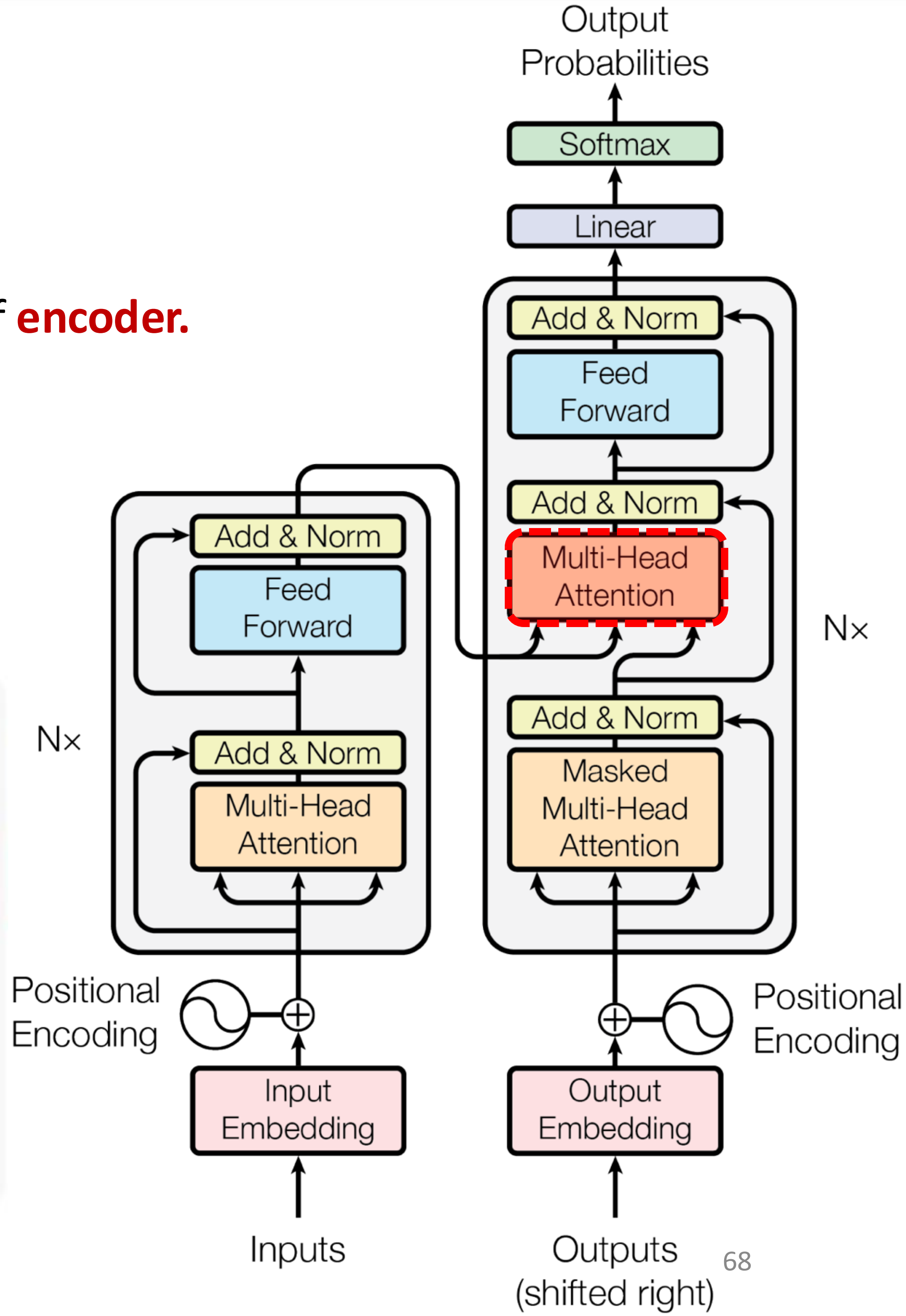
Transformer

- At any step of **decoder**, it attends to previous computation of **encoder.**



Encoder-Decoder Attention





Conclusion

Conclusion

- Attention is nowadays a crucial mechanism for deep neural networks
- Transformers are exploiting self-attention
- Transformers are powerful and generic
- There are many ways to aggregate transformers.
- Each method has advantages and disadvantages.