Deep Learning

Transformer

Lionel Fillatre

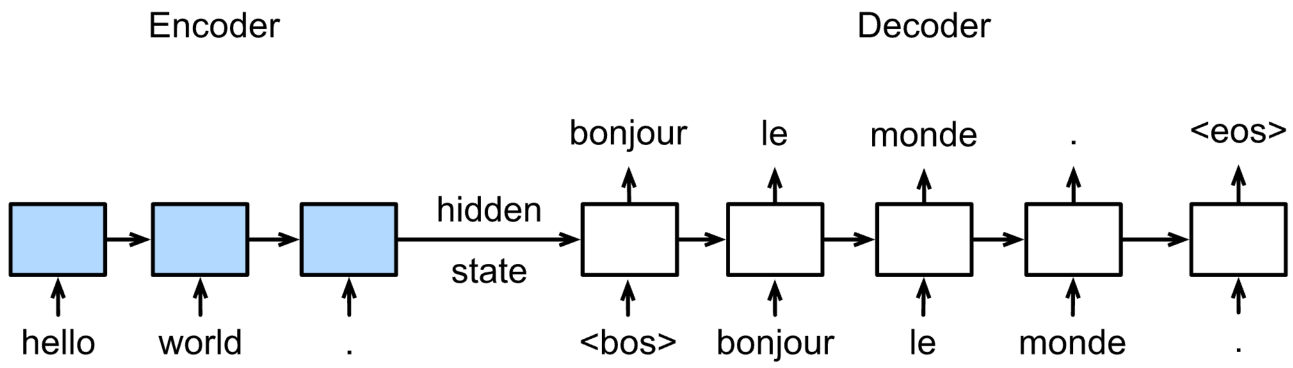
2024-2025

Outline

- Sequence to Sequence
- Attention Mecanism
- Transformer’s Encoder
- Input Encoding
- Classification Transformer
- Transformer for Sequence Transduction
- Conclusion

Sequence to Sequence

Seq2Seq Models

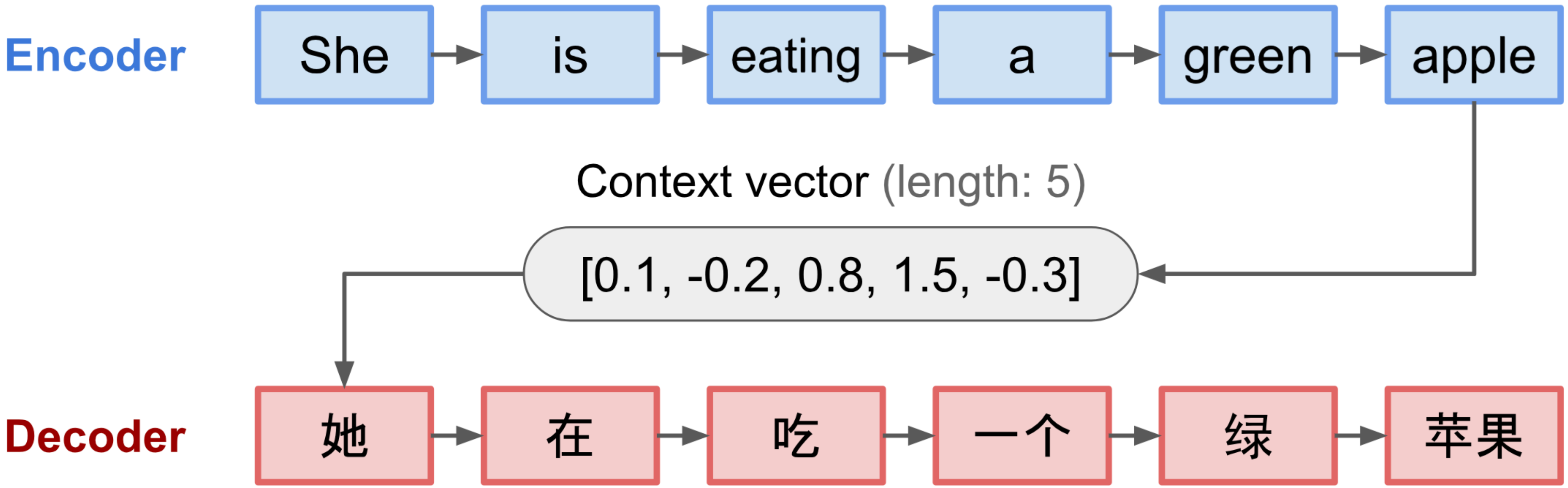


- Note that we require that each sentence ends with a special end-of-sentence symbol <EOS>, which enables the model to define a distribution over sequences of all possible lengths.
- We also use a begin-of-sentence symbol <BOS> for the decoder.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv:*1308.0850

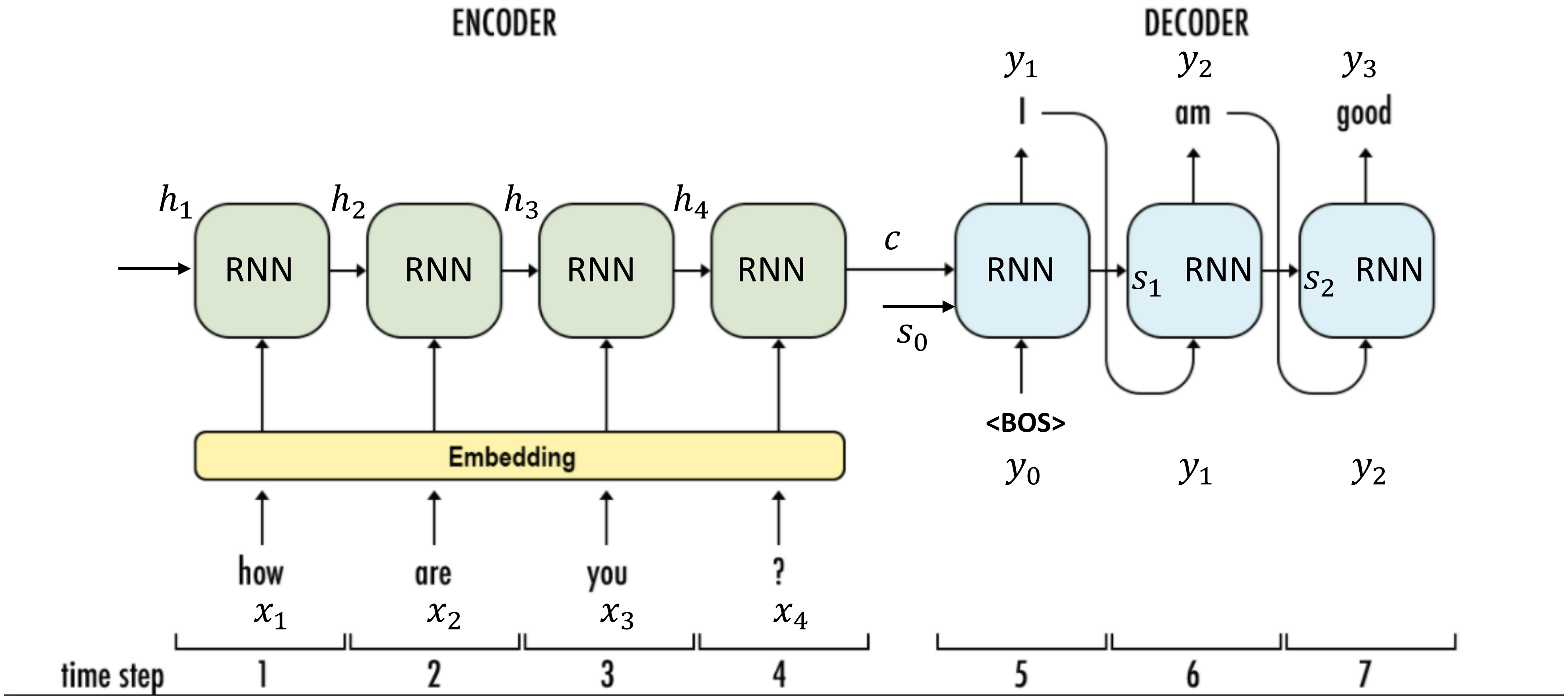
Encoder/Decoder

- 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.



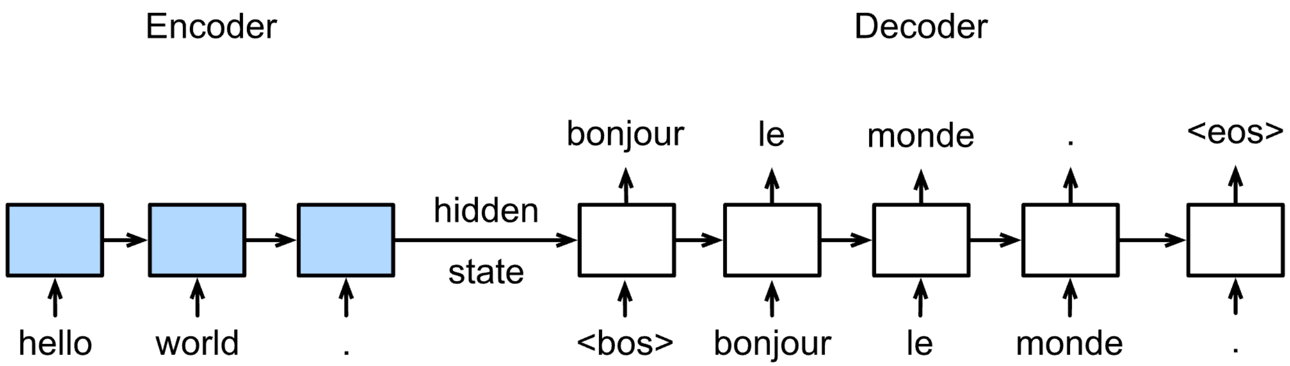
https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

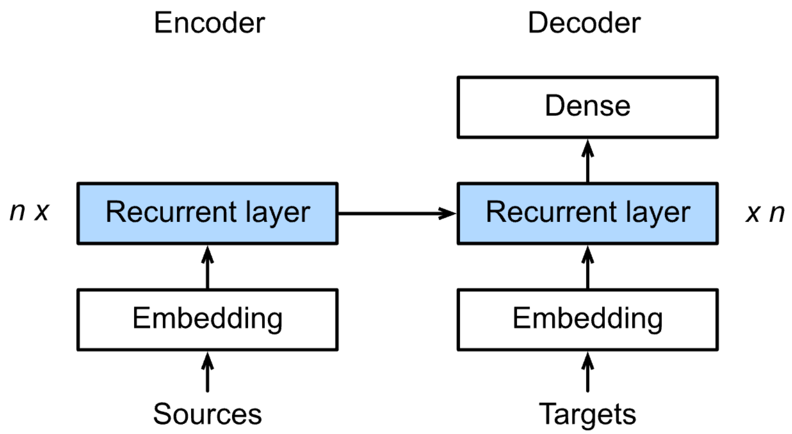
Encoder/Decoder



Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In Proceedings of the ACL Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1700–1709. Association for Computational Linguistics.

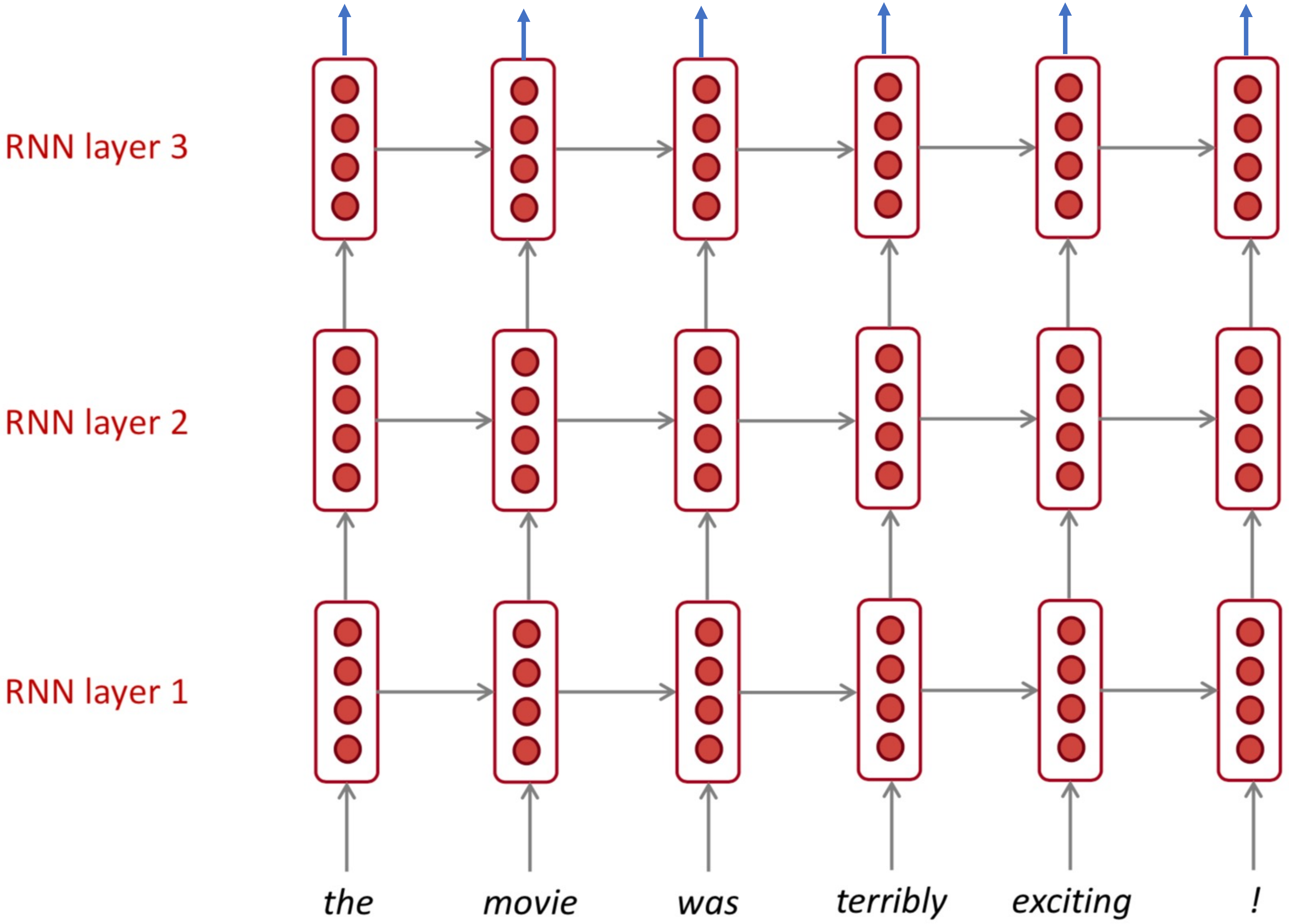
Encoder/Decoder with recurrent layers





Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv:*1308.0850

Multi-layer RNNs



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

Encoder

- From a probabilistic perspective, the encoder/decoder model is a general method to learn the conditional distribution $p(y_1, \dots, y_{T'} | x_1, x_2, \dots, x_T)$
- An encoder reads the input sentence, a sequence of vectors $x = (x_1, x_2, \dots, x_T)$, into a context vector c
- The most common approach is to use an RNN
  - RNN hidden state: $h_t = f(x_t, h_{t-1})$
  - Context vector: $c = q(\{h_1, \dots, h_T\})$

where $h_t \in \mathbb{R}^n$ is a hidden state at time t, and c is a vector generated from the sequence of the hidden states.

- f and q are some nonlinear functions.

Ilya Sutskever, Oriol Vinyals, Quoc V. Le, 2014, "Sequence to Sequence Learning with Neural Networks, » pp. 3104–311 in NIPS 2014

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio: Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. EMNLP 2014: 1724-1734

Decoder

- The decoder is often trained to predict the next word $y_{t'}$ given the context vector c and all the previously predicted words $\{y_1, y_2, \dots, y_{t'-1}\}$.
- The decoder defines a probability over the translation $y = (y_1, \dots, y_{T'})$ by decomposing the joint probability into the ordered conditionals:

$$p(y|x) = \prod_{t'=1}^{T'} p(y_{t'}|y_1, \dots, y_{t'-1}, x) \approx \prod_{t'=1}^{T'} p(y_{t'}|y_1, \dots, y_{t'-1}, c)$$

Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.
K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. EMNLP, page 1724-1734. ACL (2014)

Decoder with context vector

- The decoder is also an RNN
  - RNN decoder hidden state: $s_{t'} = f(y_{t'-1}, s_{t'-1}, c)$
  - The context vector c can be reused for any hidden state in the decoder

- With an RNN, each conditional probability is modeled as

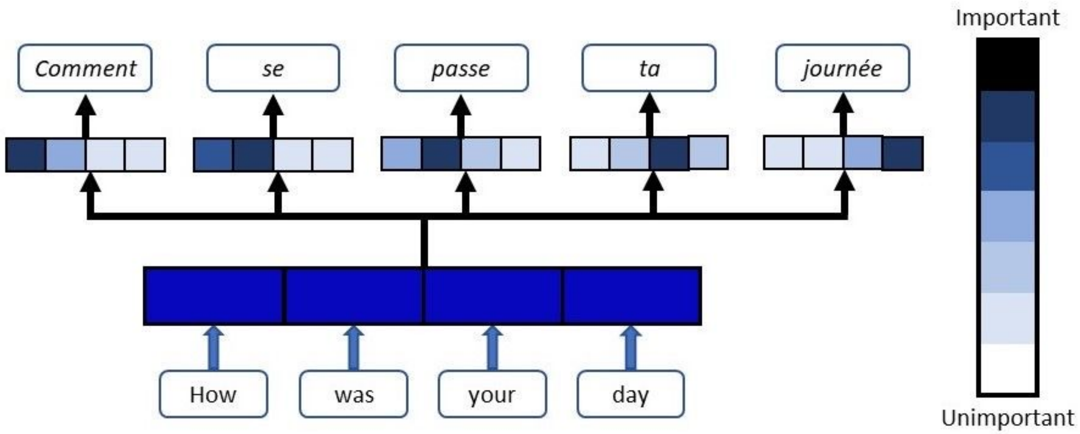
$$p(y_{t'}|y_1, \dots, y_{t'-1}, x) = g(y_{t'-1}, s_{t'}, c)$$

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

- **Drawback of the single context vector:**
  - A critical and apparent disadvantage of this fixed-length context vector design is incapability of remembering long sentences.
  - Often it has forgotten the first part once it completes processing the whole input.
  - The attention mechanism was born to resolve this problem.

Attention Mecanism

Decoder with attention



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

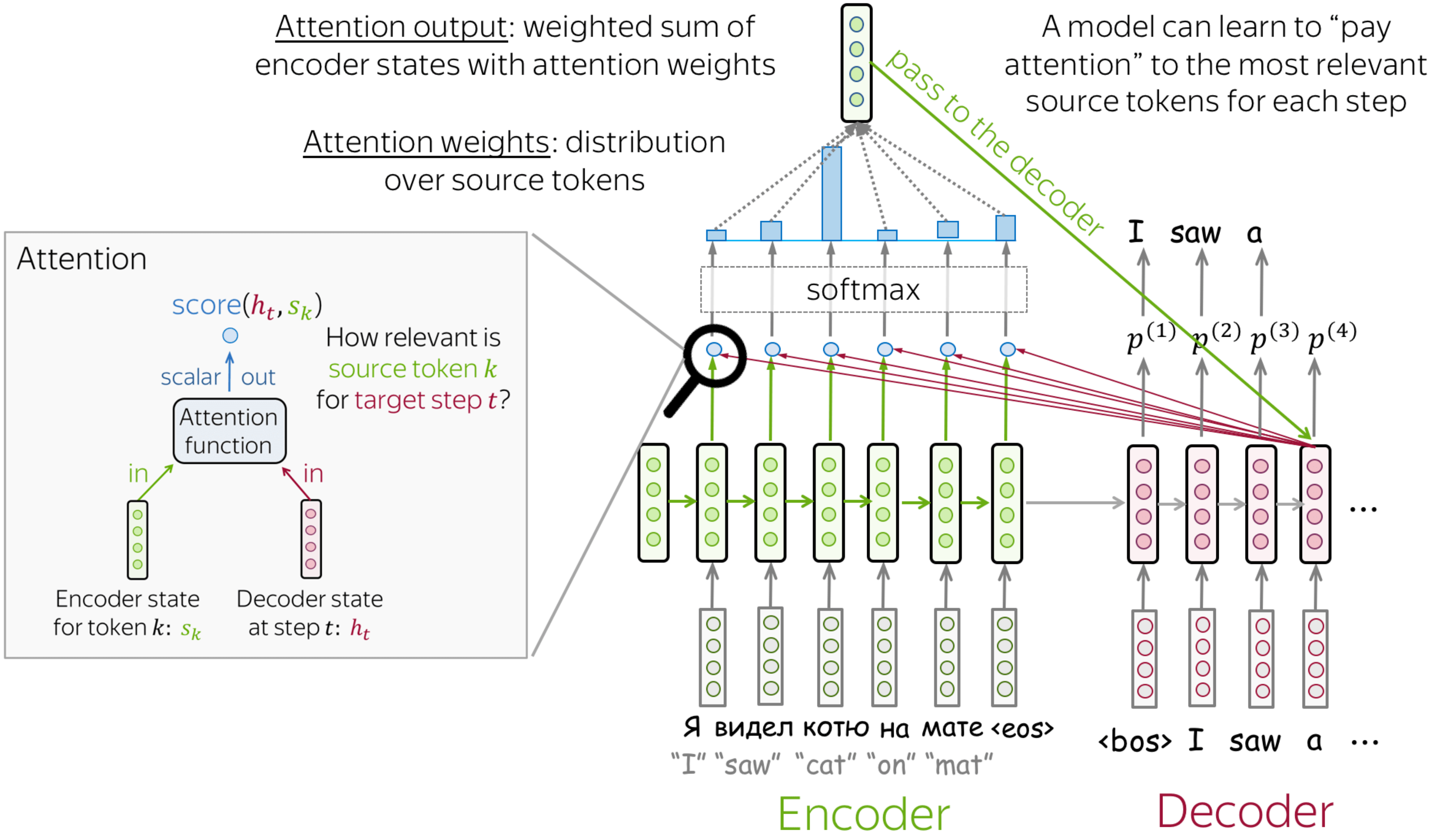
- With an RNN, each conditional probability is modeled as

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

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

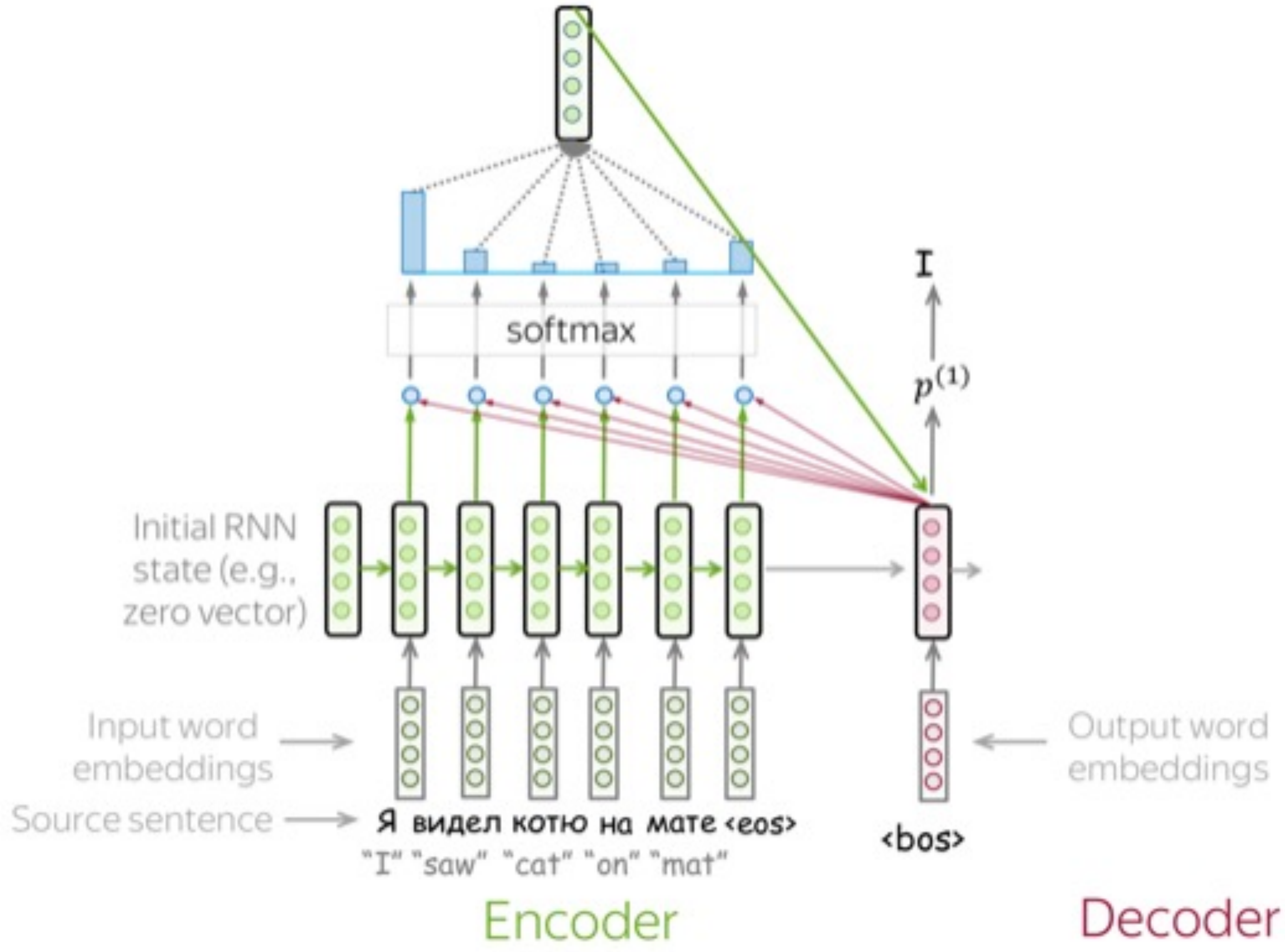
- Principle: the probability is conditioned on a distinct context vector $c_{t'}$ for each target word $y_{t'}$

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

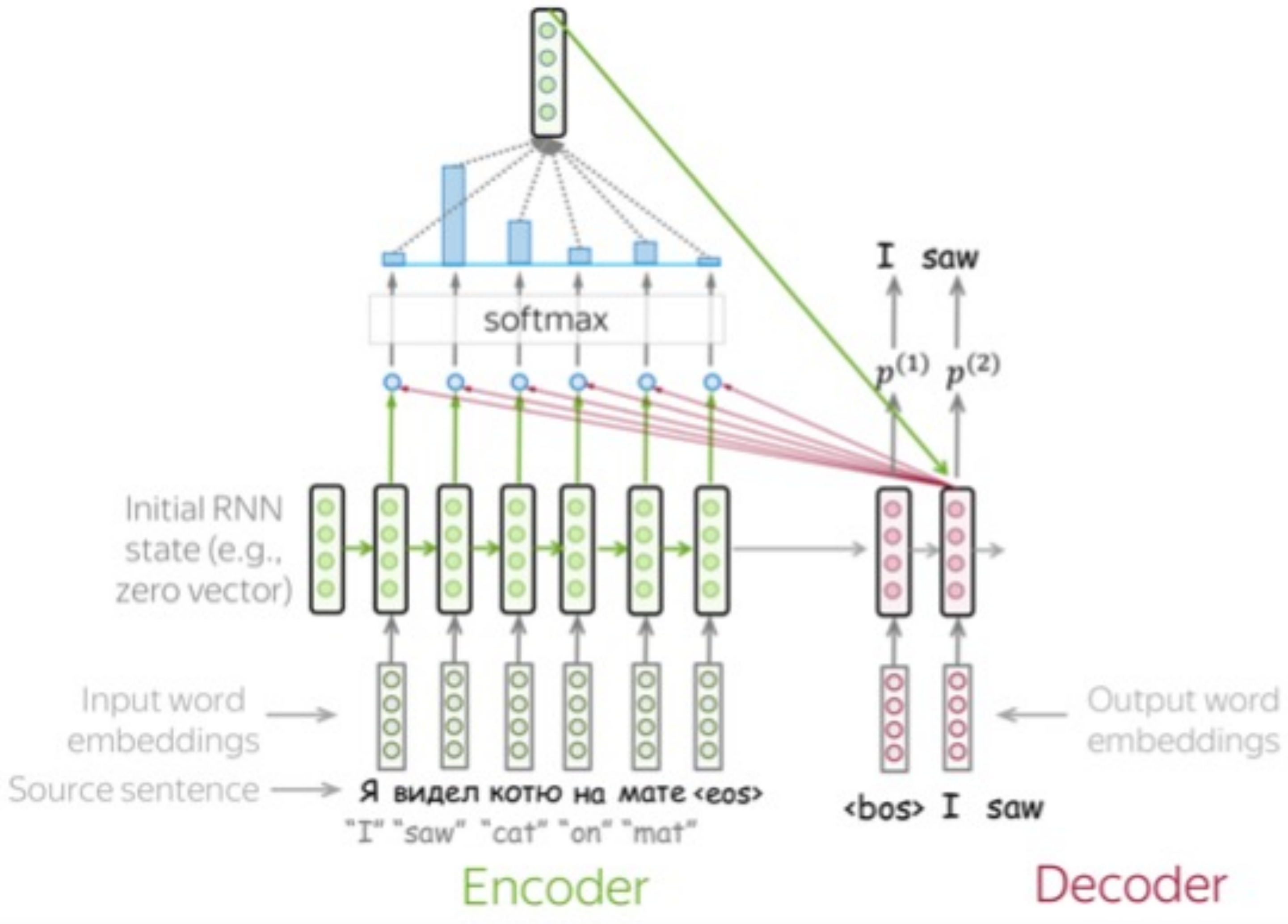


https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

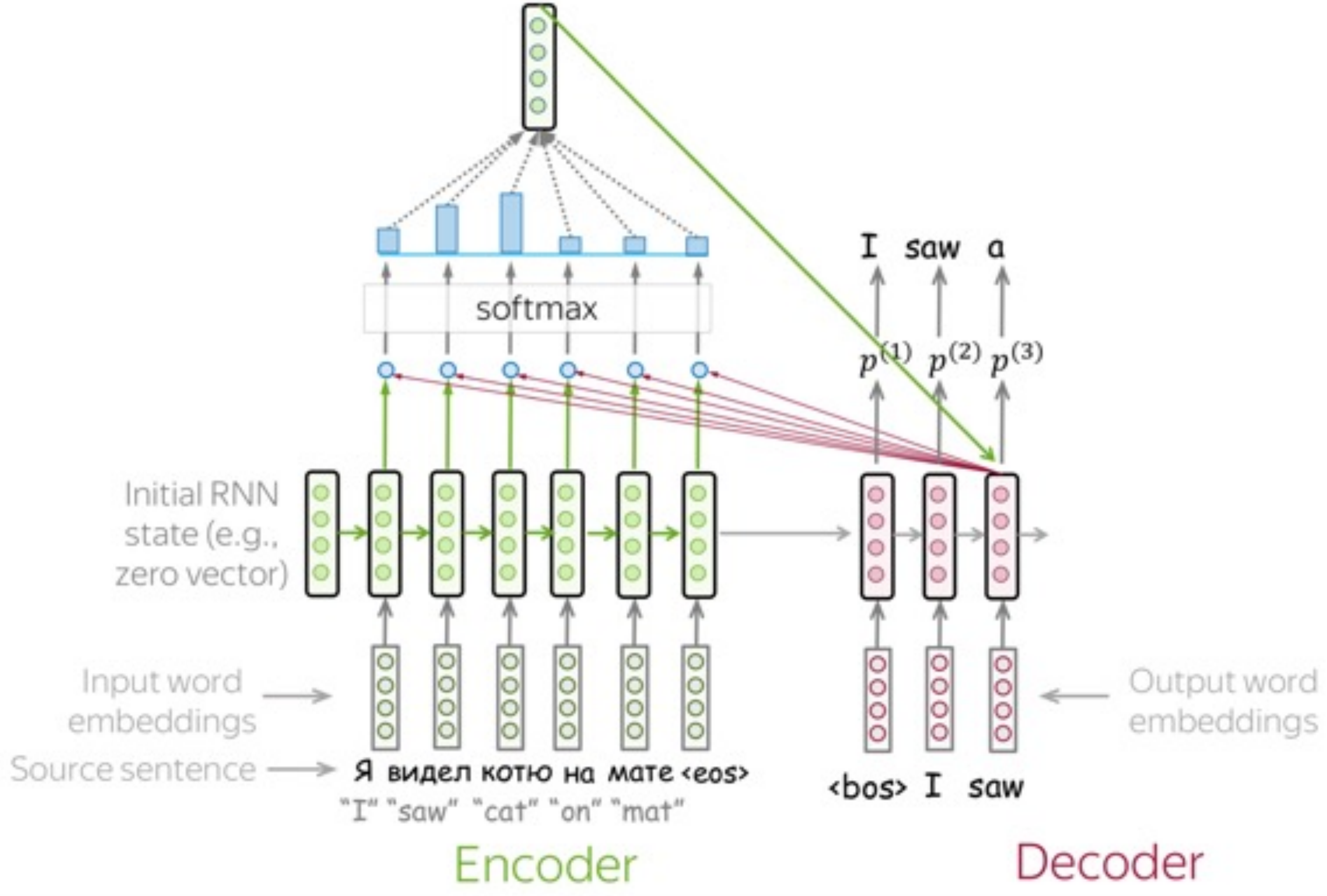
An illustration



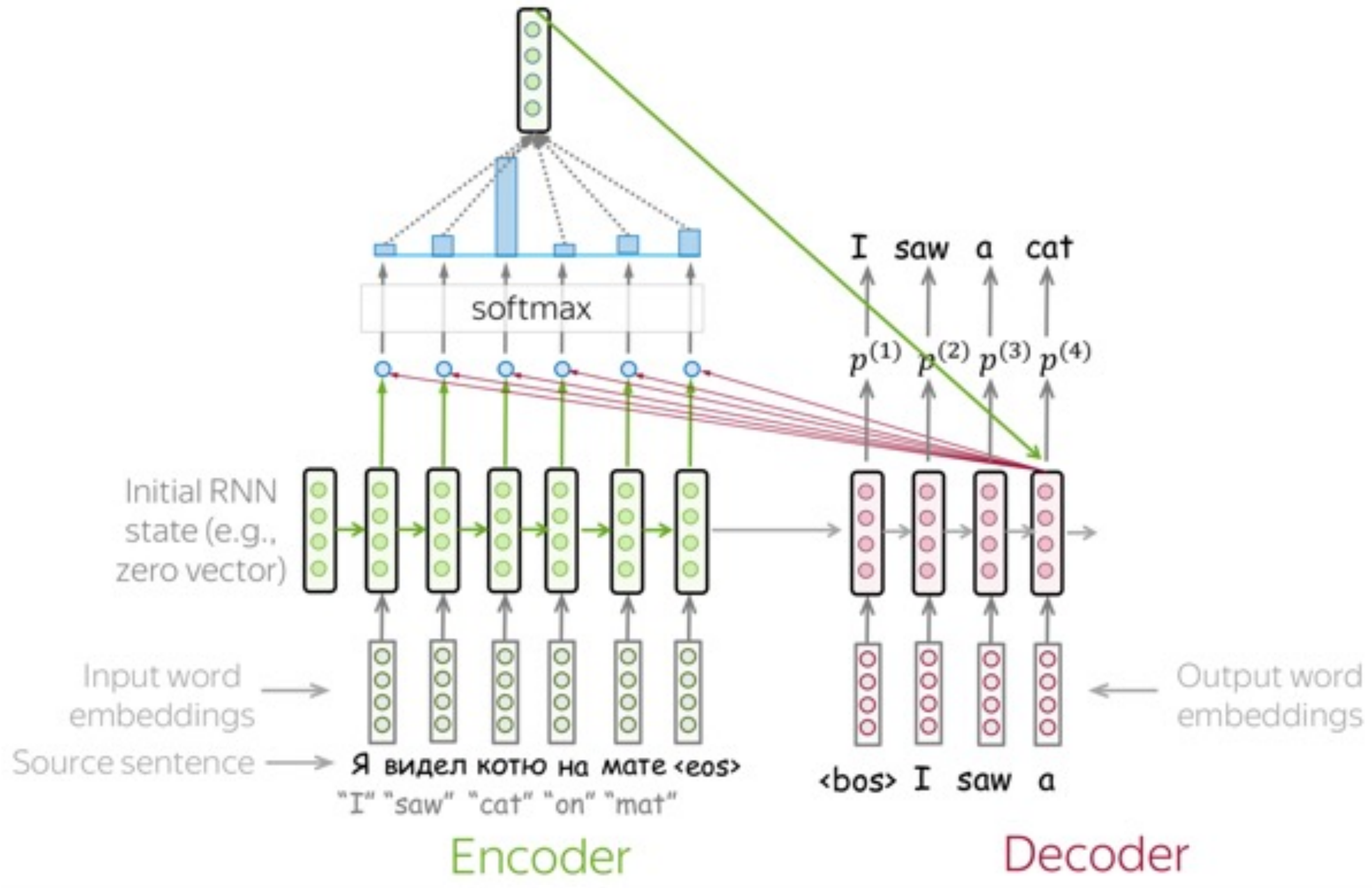
An illustration



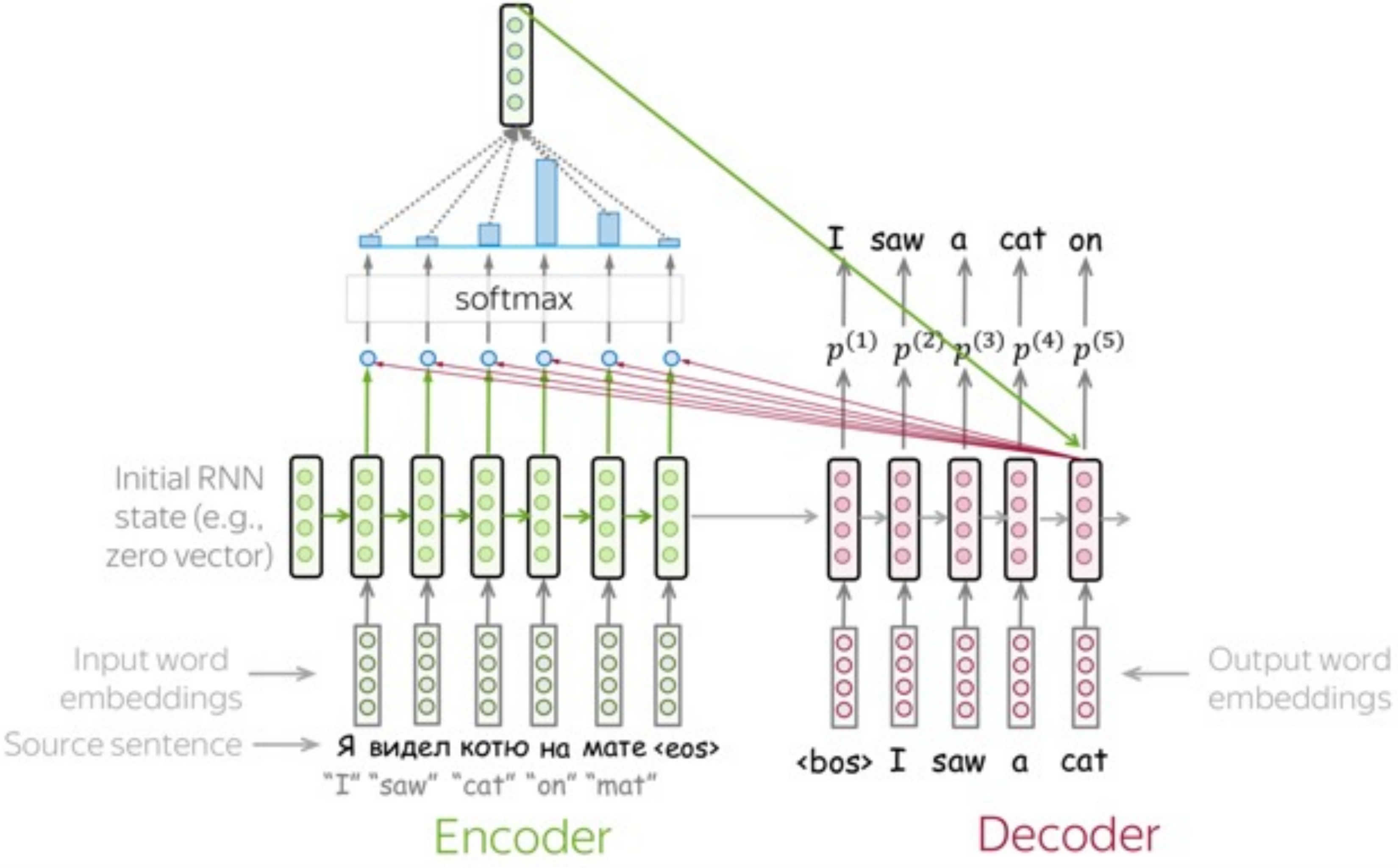
An illustration



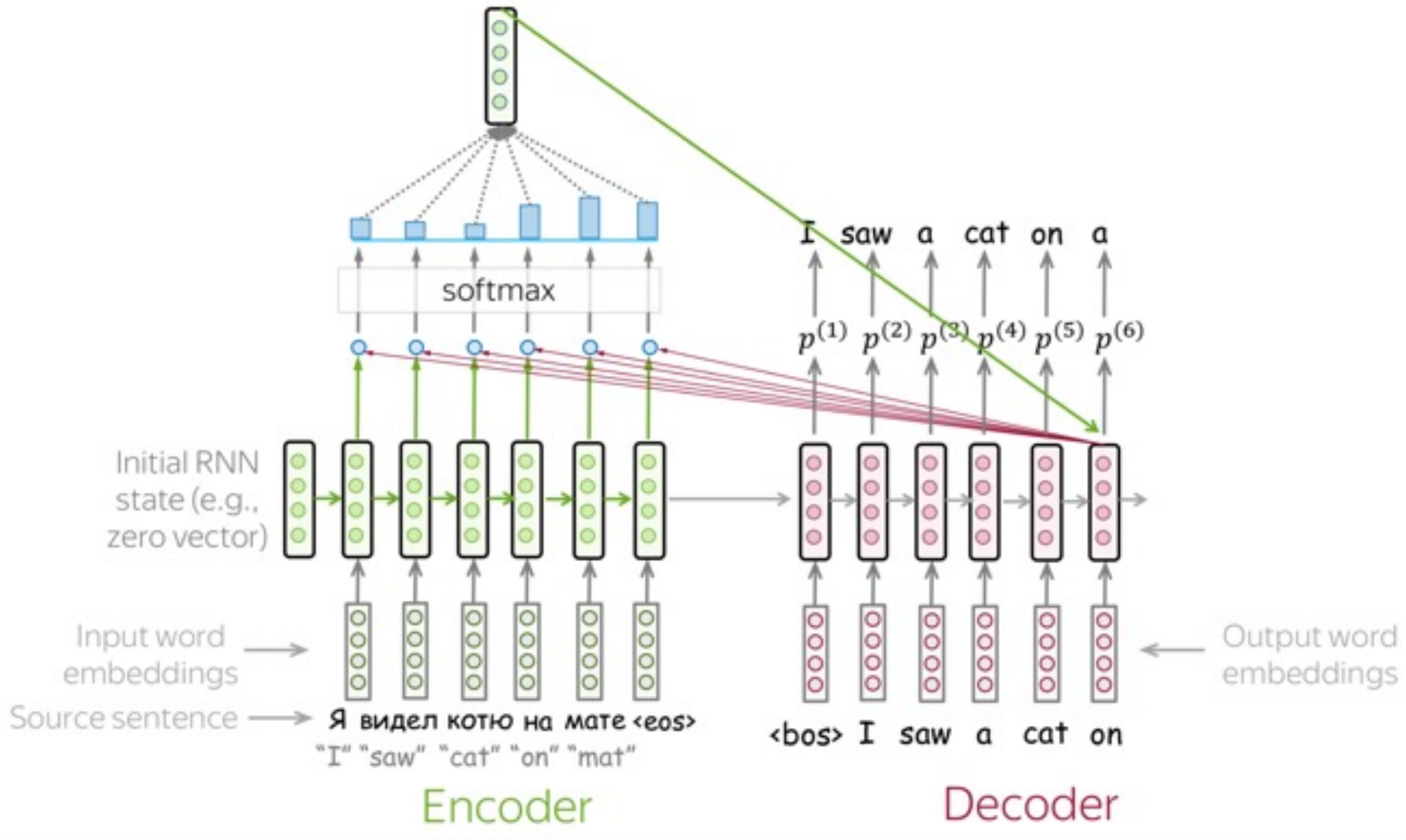
An illustration



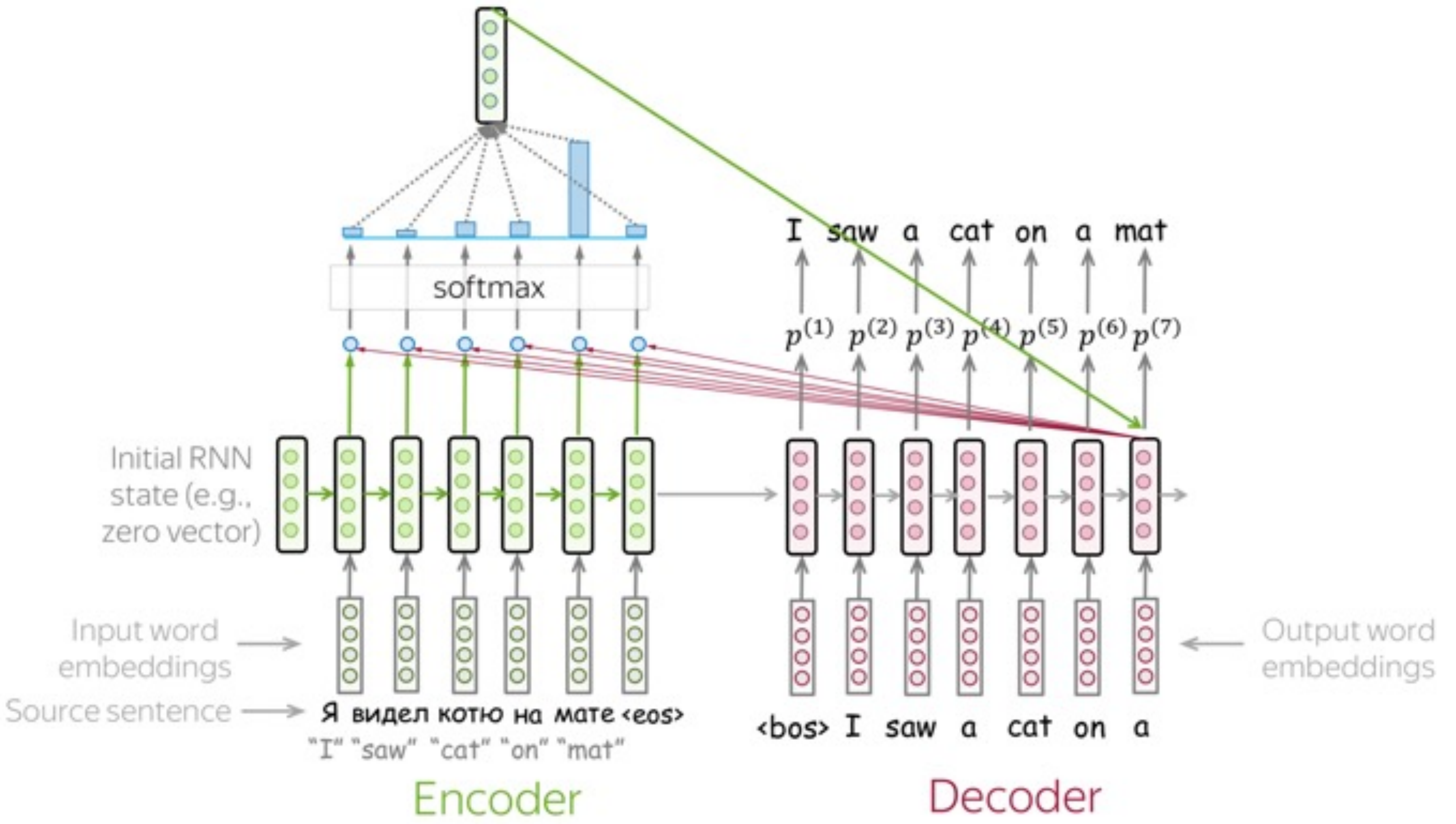
An illustration



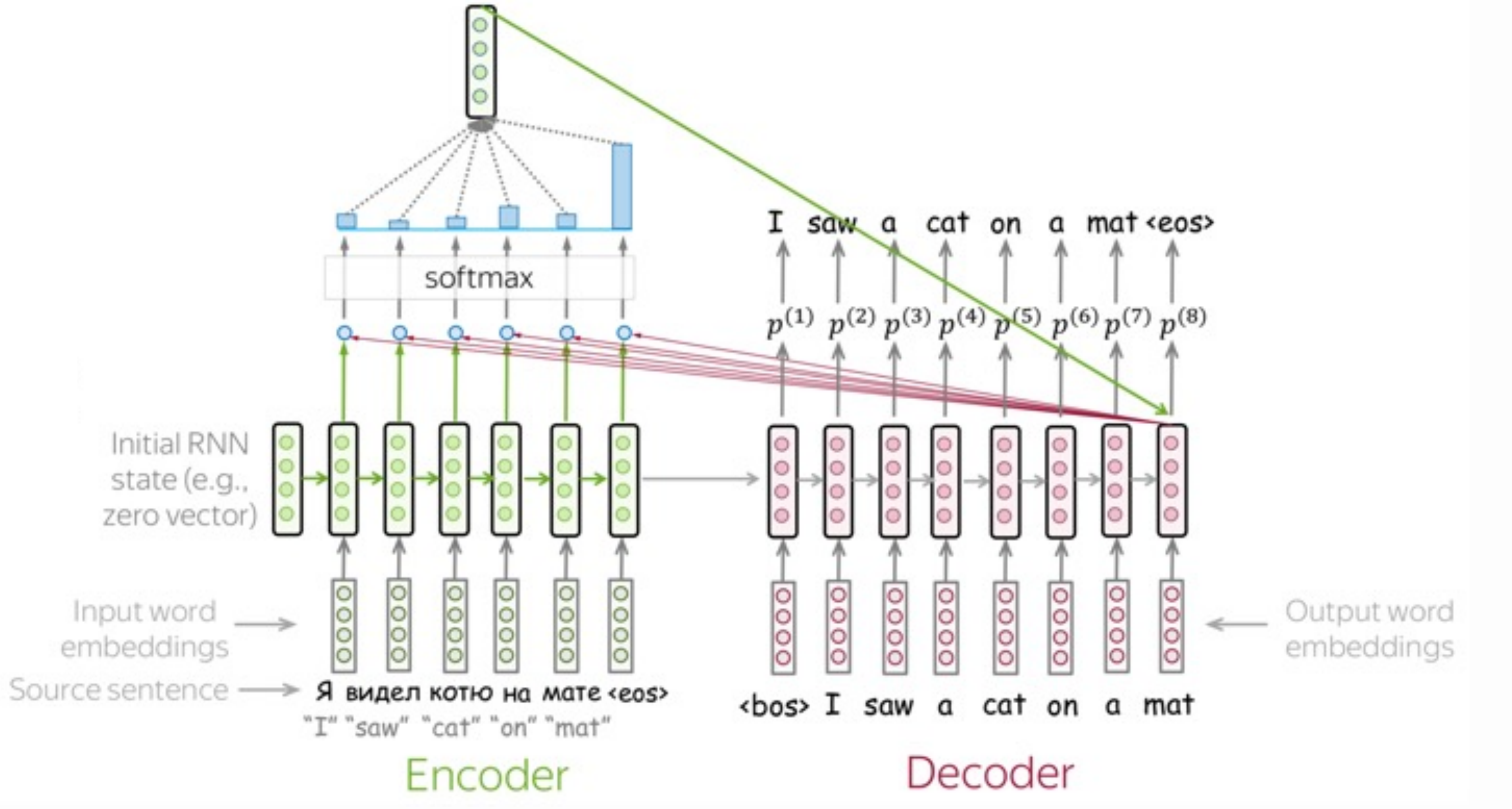
An illustration



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

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

where

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

- 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.

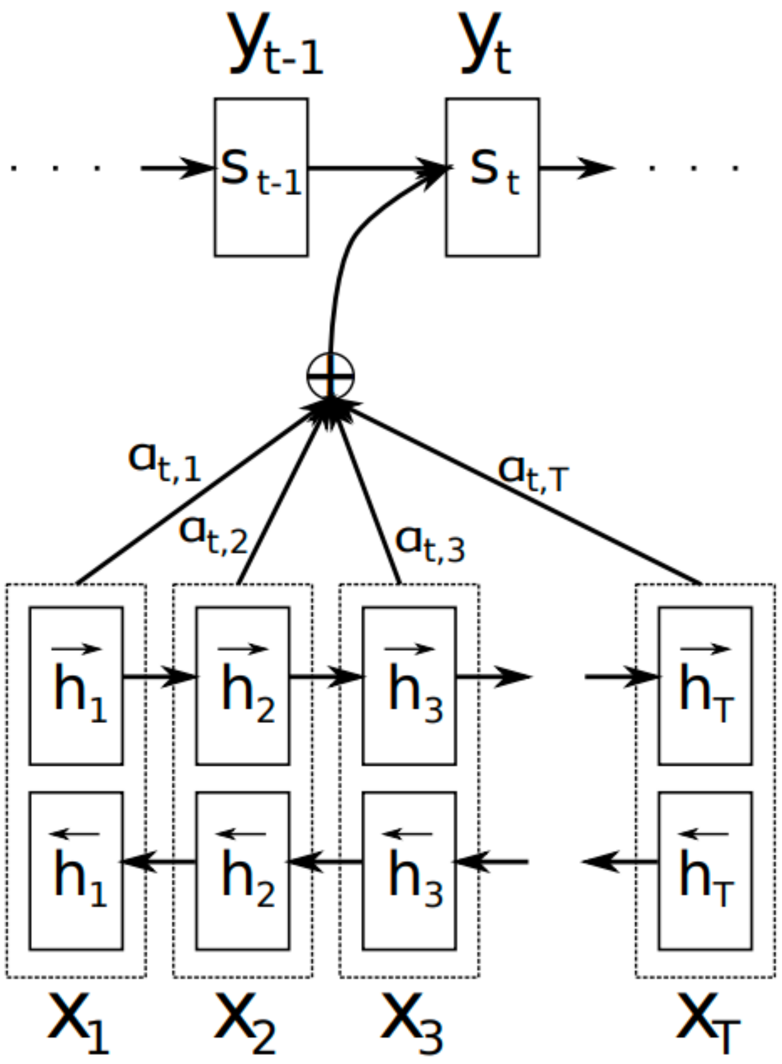
- Let α_{ij} be a probability that the target word y_i is aligned to, or translated from, a source word x_j. Then, the i-th context vector c_i is the expected annotation over all the annotations h_j with probabilities α_{ij}

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

- 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.

Bidirectional RNN

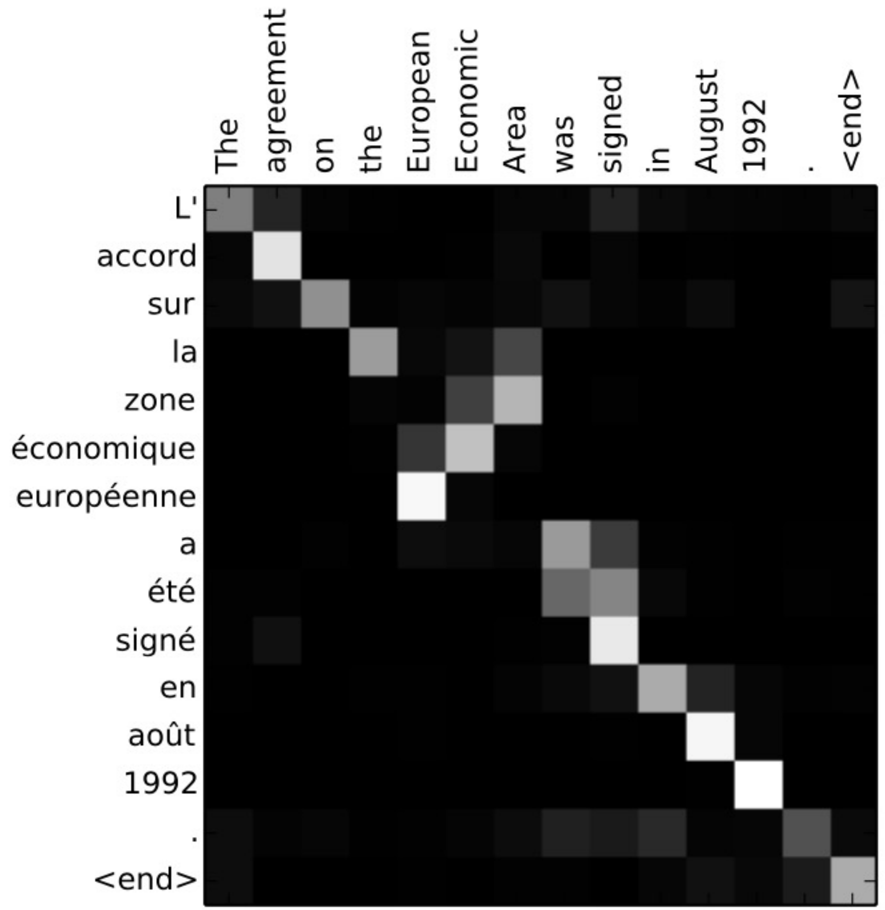
- The hidden state of the encoder is often a bidirectional RNN



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

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).



Transformer’s Encoder

Transformer’s Encoder: Principle

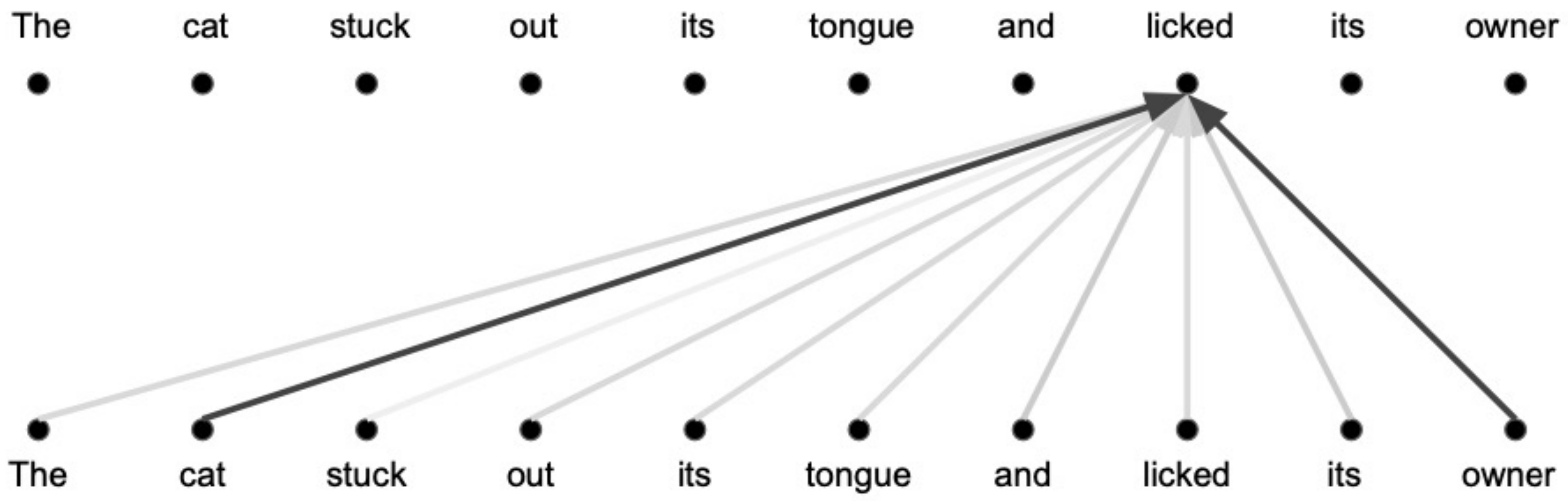
- Self-attention
- Queries, keys and values
- Scaling the dot product
- Multi-head attention

“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

$y_1, y_2, \ldots, y_t$

$x_1, x_2, \ldots, x_t$

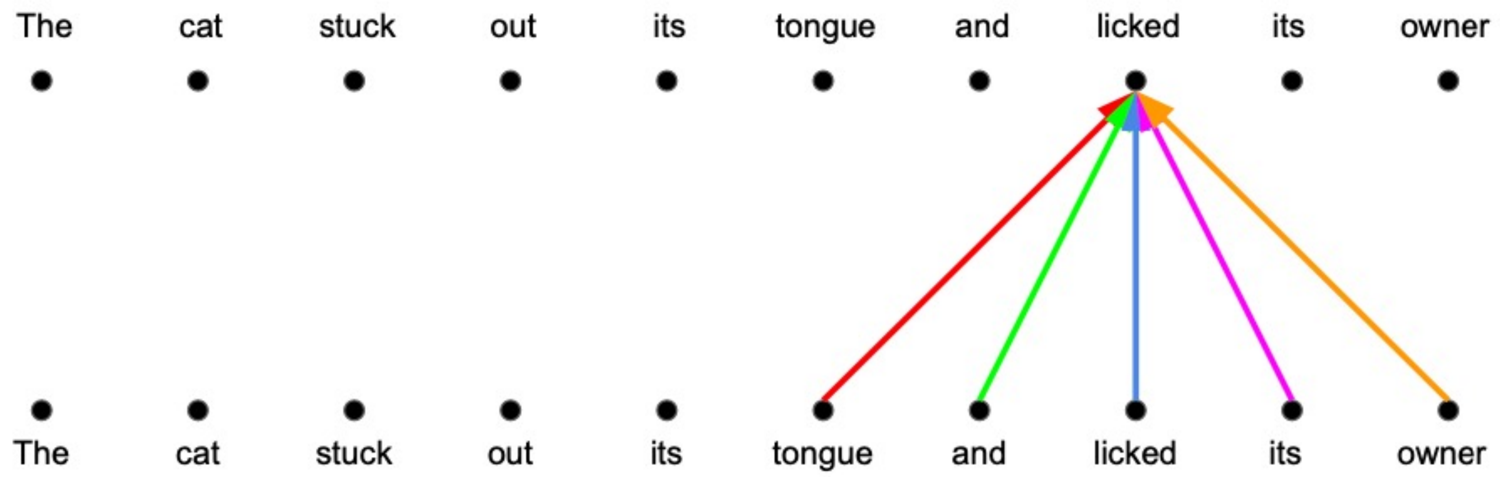


- The word « licked » is most correlated to « cat » (who?) ans « owner » (to whom?)

Comparison with convolution

$y_1, y_2, \ldots, y_t$

$x_1, x_2, \ldots, x_t$



- 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.

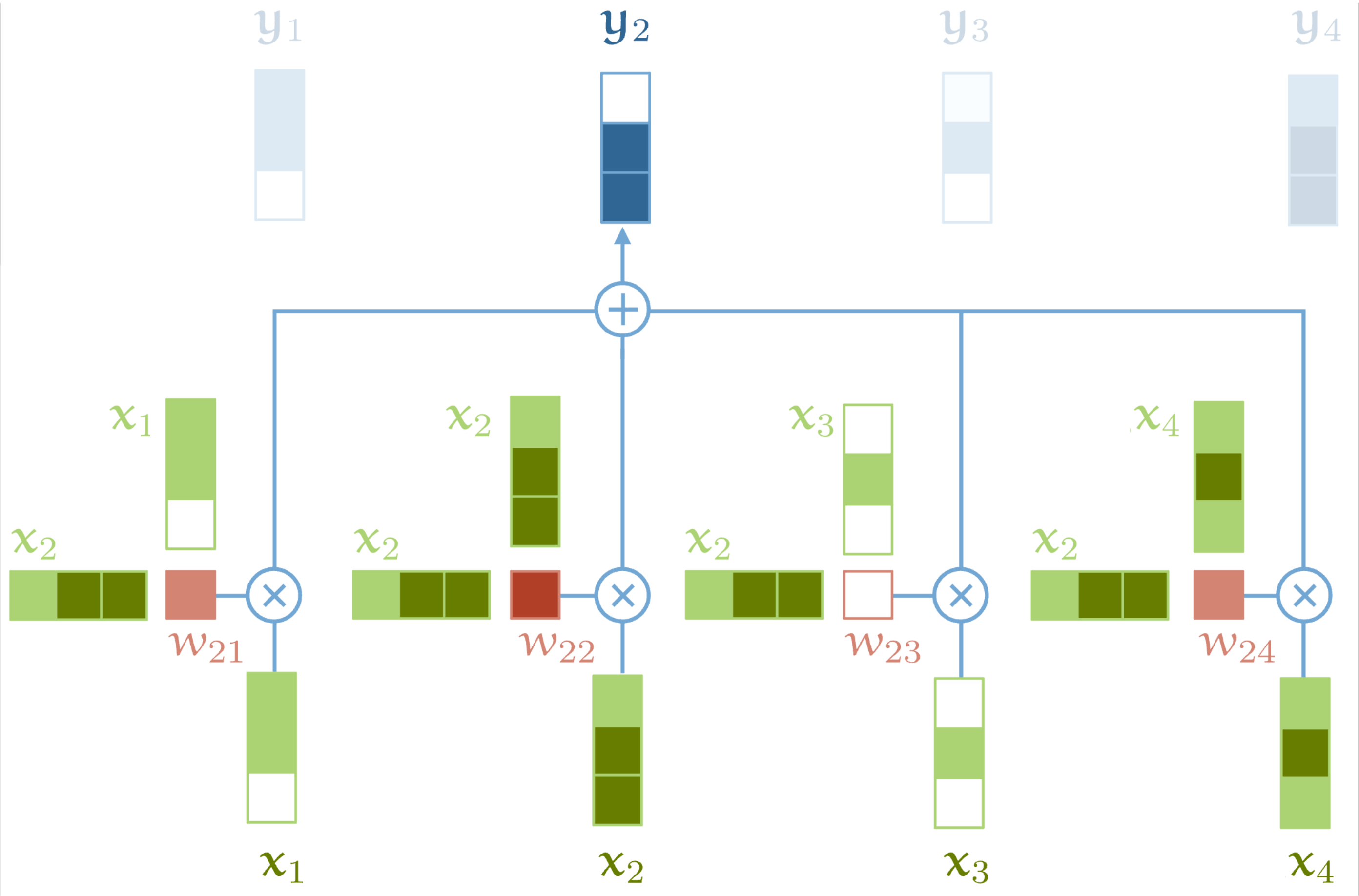
- The simplest option for the score function is the dot product:

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

- 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:

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

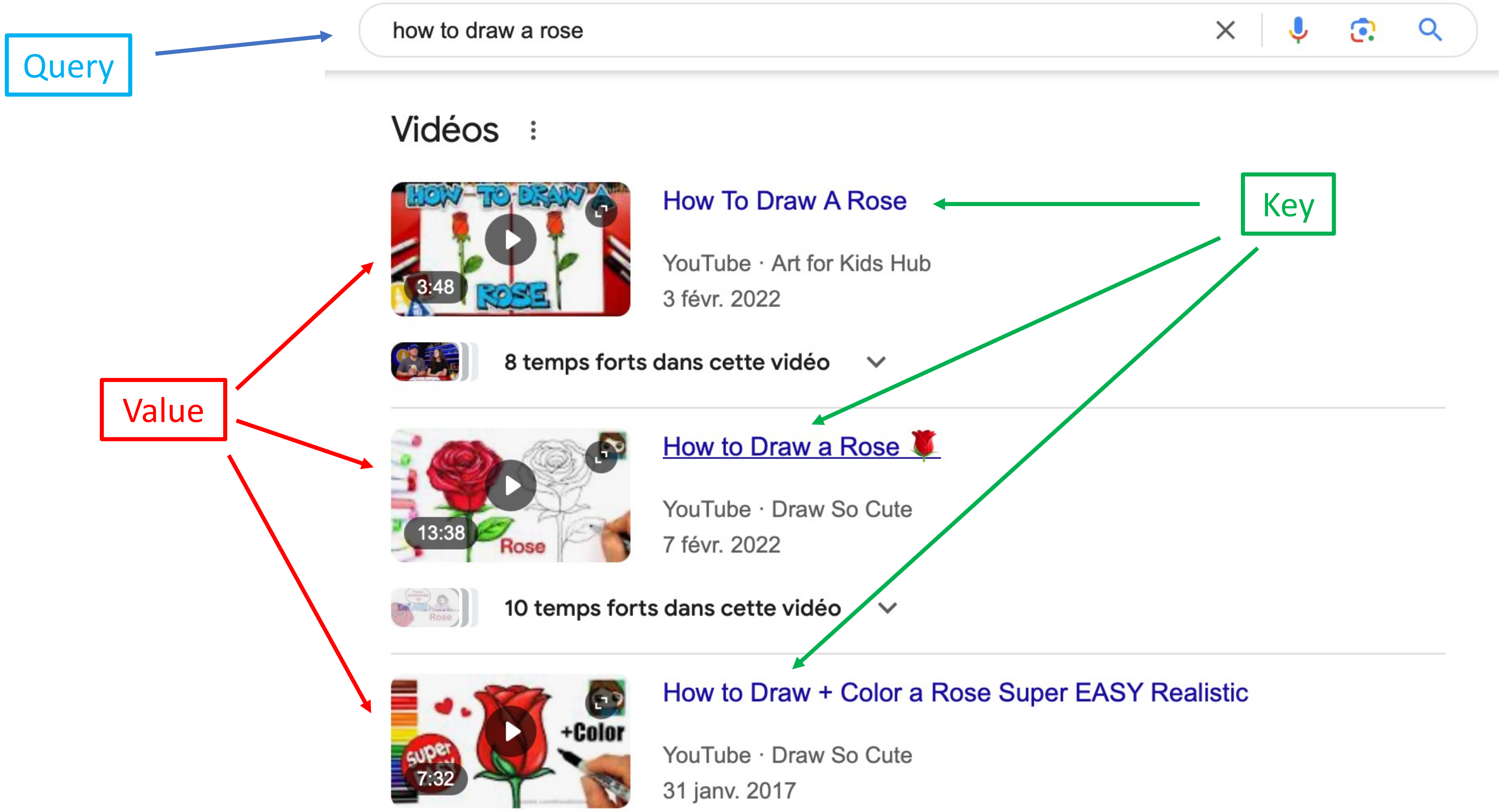
Self-attention



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.

Illustration

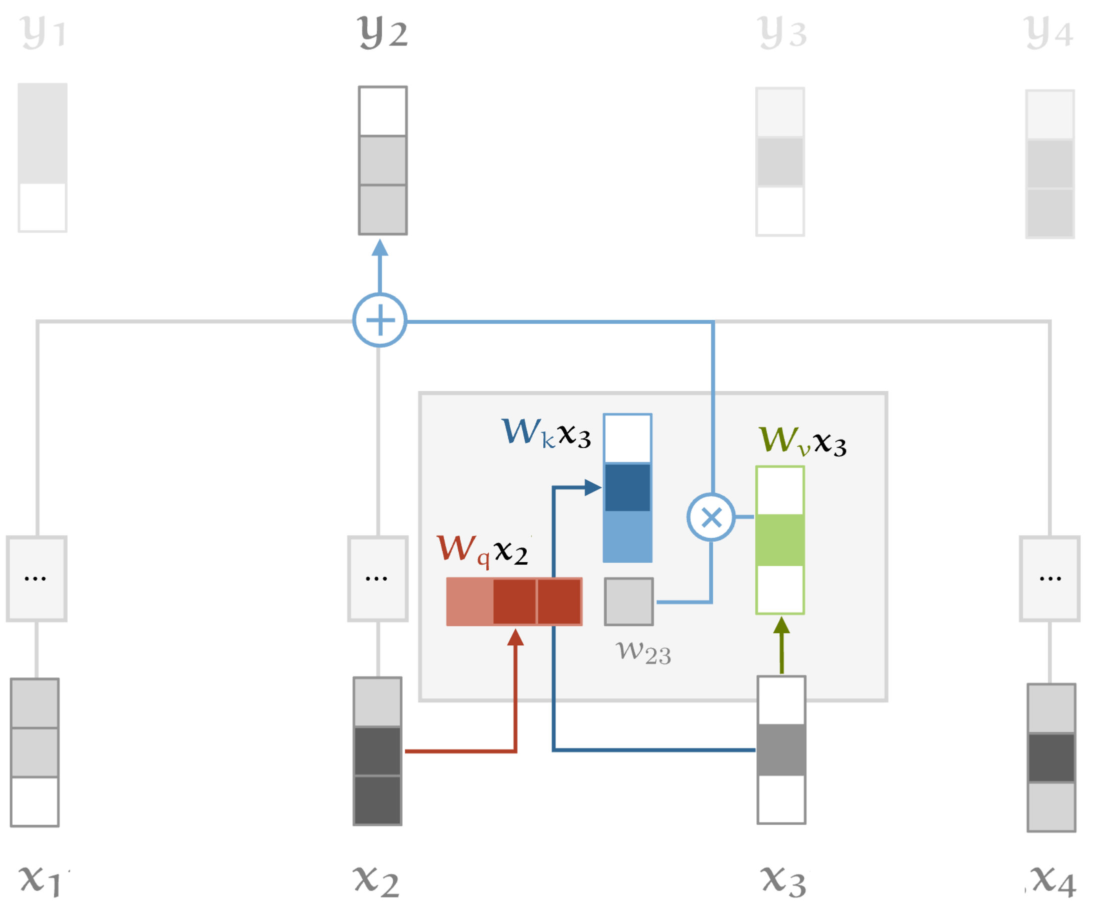


Illustration of the self-attention with key, query and value transformations.

Scaling the dot product

- 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:

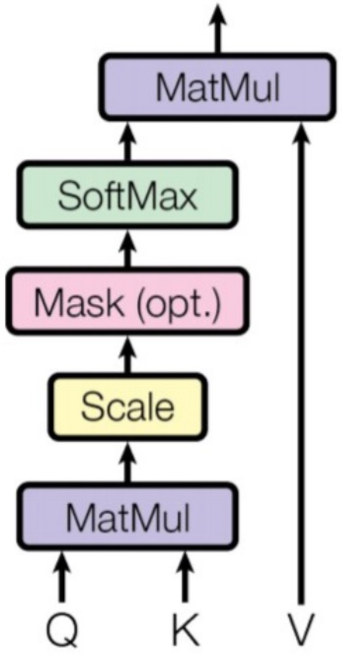
$$e_{ij} = \frac{q_i^T k_j}{\sqrt{d}}$$

- 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.

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:

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



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.

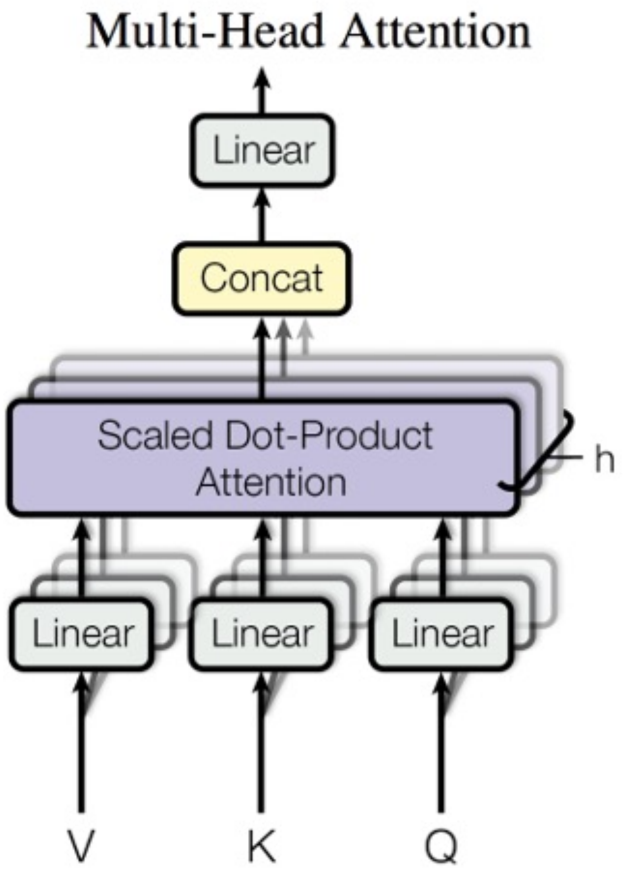
- For input x_j each attention head produces a different output vector y_j^i. We concatenate these, and pass them through a linear transformation W^O to reduce the dimension back to d.

Wide self-attention

- If the embedding vector has 256 dimensions, and we have $h = 8$ attention heads

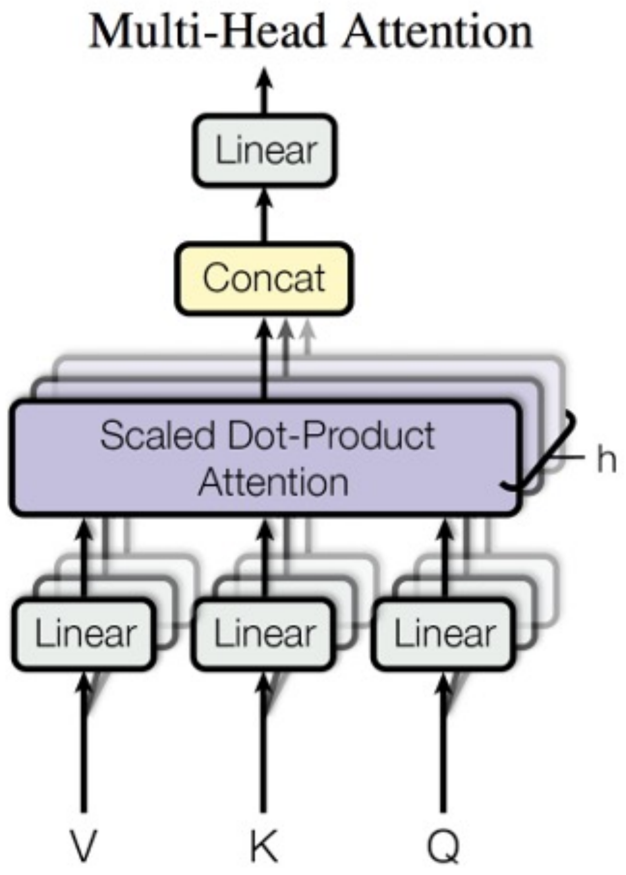
- For each head, we generate keys, values and queries of 256 dimensions each

- This means that the matrices W_i^Q, W_i^K, W_i^V are all 256×256 with $i = 1, \dots, h$



In practice: narrow self-attention

- The standard option is to cut the embedding vector into h chunks:
  - If the embedding vector has 256 dimensions, and we have $h = 8$ attention heads,
  - We cut the embedding vector into 8 chunks of 32 dimensions.
  - For each chunk, we generate keys, values and queries of 32 dimensions each.
  - This means that the matrices W_i^Q, W_i^K, W_i^V are all 32×32 with $i = 1, \dots, h$
- The narrow self-attention is faster, and more memory efficient but all else being equal, the wide self-attention does give better results (at the cost of more memory and time).



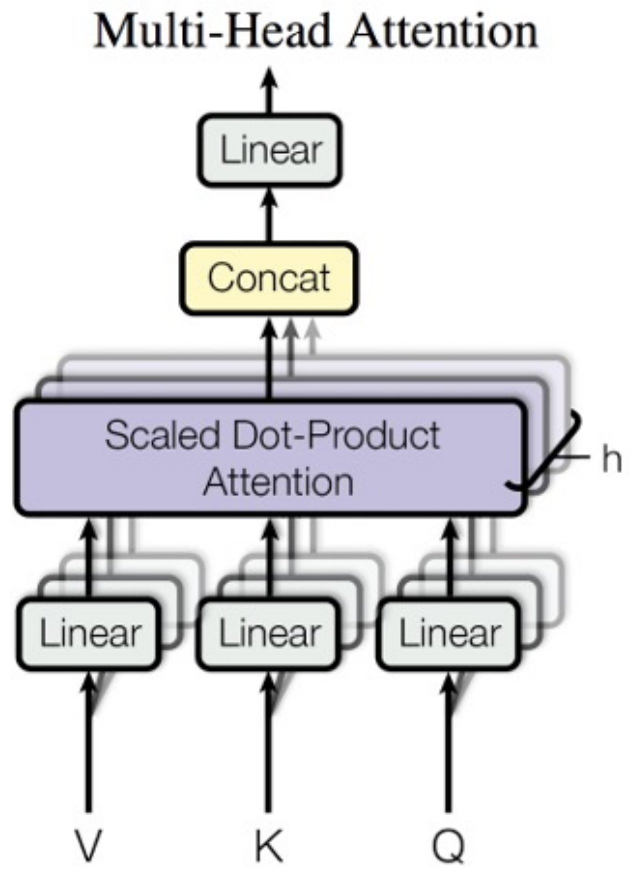
Multi-Head Attention (Matrix Form)

- Matrix model:

$$\text{MultiHead}(Q, K, V) = [\text{head}_1; \text{head}_2; \cdots, \text{head}_h]W^O$$

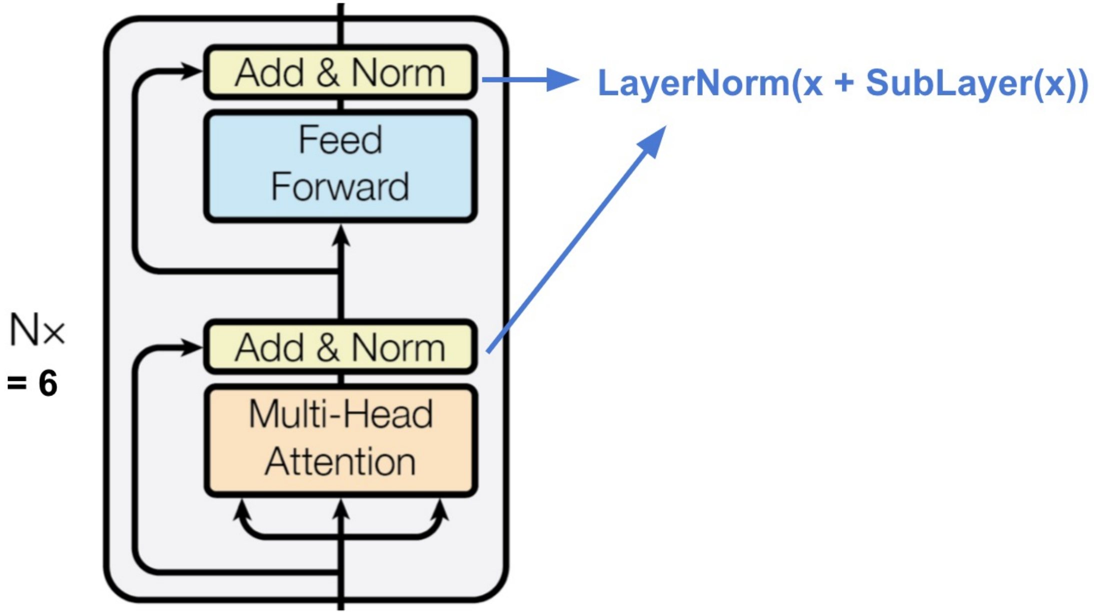
$$\text{head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right)$$

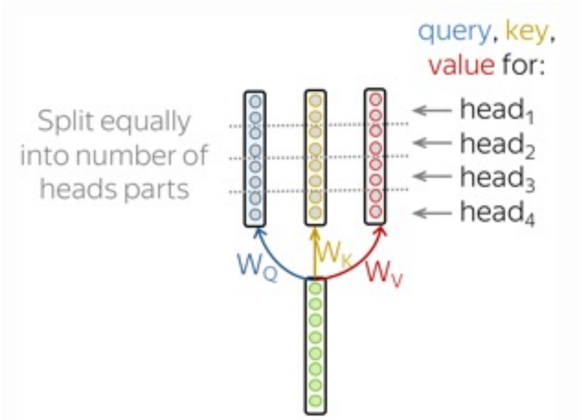
- where
  - $W_i^Q \in \mathbb{R}^{d \times d_Q}$,
  - $W_i^K \in \mathbb{R}^{d \times d_k}$,
  - $W_i^V \in \mathbb{R}^{d \times d_v}$,
  - $W^O \in \mathbb{R}^{hd_v \times d}$ are parameter matrices to be learned.



The Transformer's Encoder

- A stack of $N = 6$ identical layers.
- Each layer has a multi-head self-attention layer and a simple position-wise fully connected feed-forward network.
  - Attention: "look at other tokens and gather information",
  - FFN: "take a moment to think and process this information").

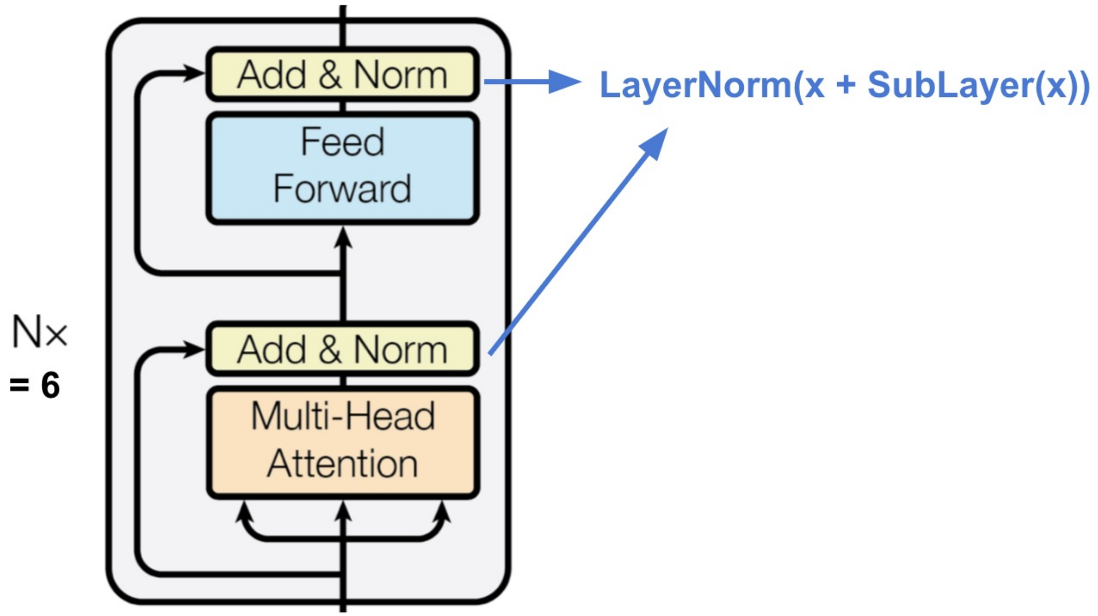




Multi-Head Attention

The Transformer’s Encoder

- Each sub-layer adopts a residual connection and a layer normalization.
    - Equivalent to regularization of the layer
- All the sub-layers output data of the same dimension (for example $d = 512$)



Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Block « Add and Norm »

- 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.
- The transformer employs a residual connection around each of the sublayers, followed by layer normalization.
- That is, the output of each sublayer is

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

where $\text{Sublayer}(x)$ is the function implemented by the sublayer itself.

Layer normalization. Ba, J., Kiros, J. R., & Hinton, G. E. (2016). Advances in nips 2016 deep learning symposium

LayerNorm$\big(x + \text{Sublayer}(x)\big)$

- For a batch $\{x_n\}_{n=1,\ldots,N}$ 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)

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

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

- γ and β are learnable affine transform parameters

https://theaisummer.com/normalization/

Input encoding

Encoding the inputs

- The initial inputs of the encoder are the embeddings of the sequence of inputs (typically Word Embeddings)

- The initial inputs of the decoder are the embeddings of the outputs.

- The order of sequence (position of words in a sentence) information is very important.
  - Since there is no recurrence, this information on the absolute (or relative) position in a sequence is represented by the use of “position encodings”.

Positional Encodings

- Recurrent Neural Networks (RNNs) inherently take the order of word into account
  - They parse a sentence word by word in a sequential manner.

- But the Transformer architecture ditched the recurrence mechanism in favor of multi-head self-attention mechanism. Avoiding the RNNs' method of recurrence will result in massive speed-up in the training time.

- As each word in a sentence simultaneously flows through the Transformer's encoder/decoder stack, the model itself doesn't have any sense of position/order for each word.

- One possible solution to give the model some sense of order is to add a piece of information to each word about its position in the sentence. We call this « piece of information », the positional encoding.

Full input of a transformer = Embedding vector + Positional encoding

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

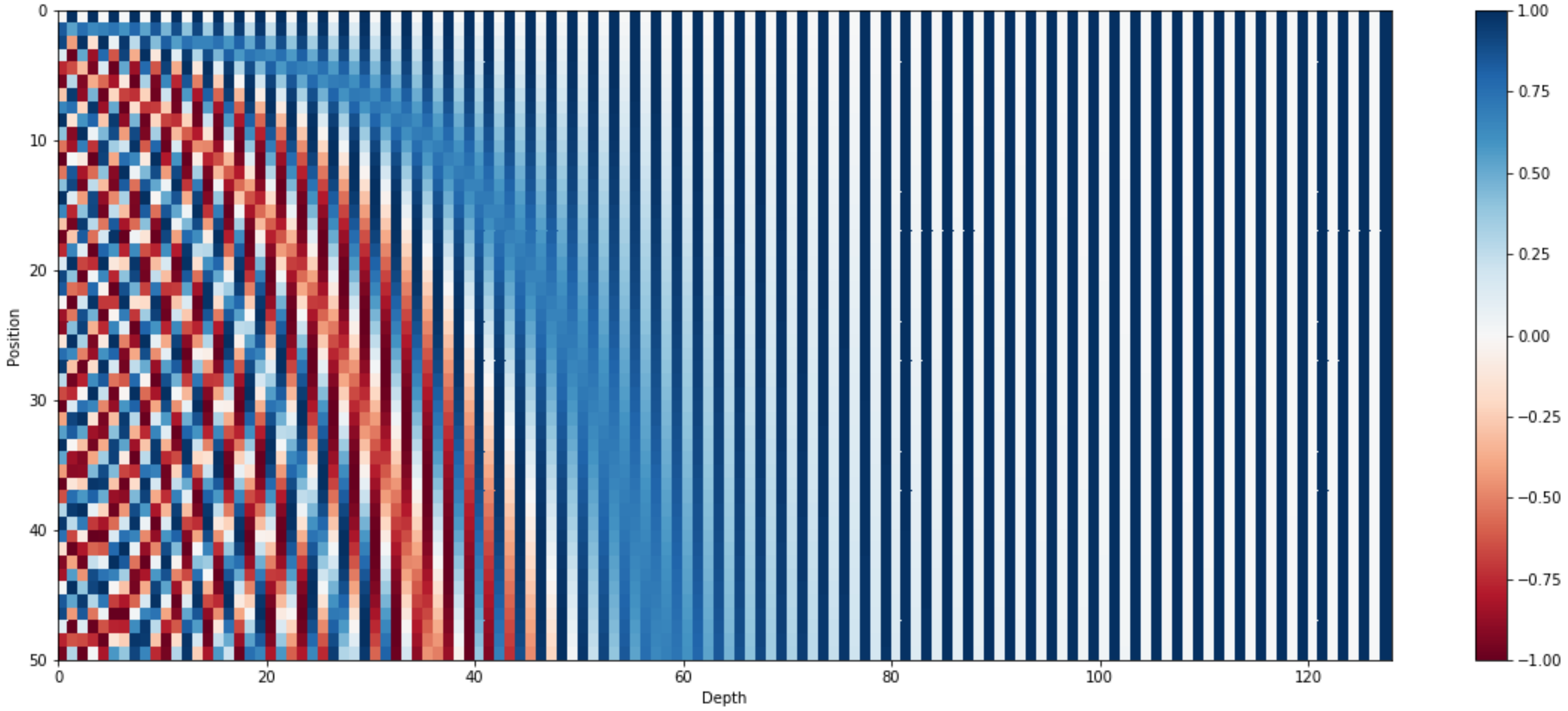
- Let t the desired position in an input sentence, $p_t = \big(p_t(0), \dots, p_t(d-1)\big) \in \mathbb{R}^d$ be its corresponding encoding. Then,

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

- 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



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

Interpretation as Circles

- You could see this encoding as $d/2$ pairs of circles rather than d dimensional vectors.

- When you fix the depth $i = 2k$ of each encoding, you can extract a 2 dimensional vector

$$PE_i(t) = \left(\sin\left(\frac{t}{10000^{i/d}}\right), \cos\left(\frac{t}{10000^{i+1/d}}\right)\right)$$

- If you constantly change the position value t, the vector $PE_i(t)$ rotates clockwise on the unit circle.

- The deeper the dimension i of the embedding is, the smaller the frequency of rotation is.

- Note that a linear transformation (a rotation) allows us to change $PE_i(t)$ into $PE_i(t + s)$ for any offset s

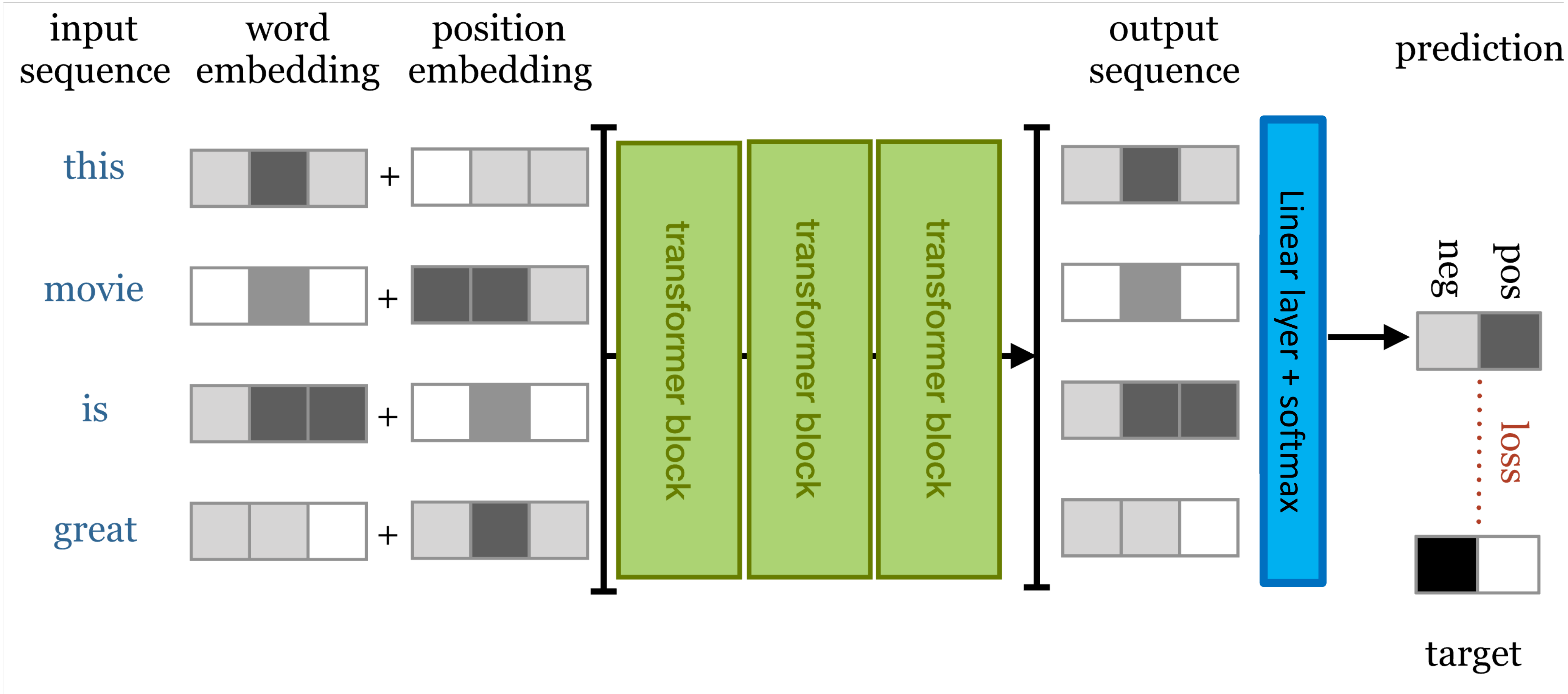
https://data-science-blog.com/blog/2021/04/22/positional-encoding-residual-connections-padding-masks-all-the-details-of-transformer-model/

Classification transformer

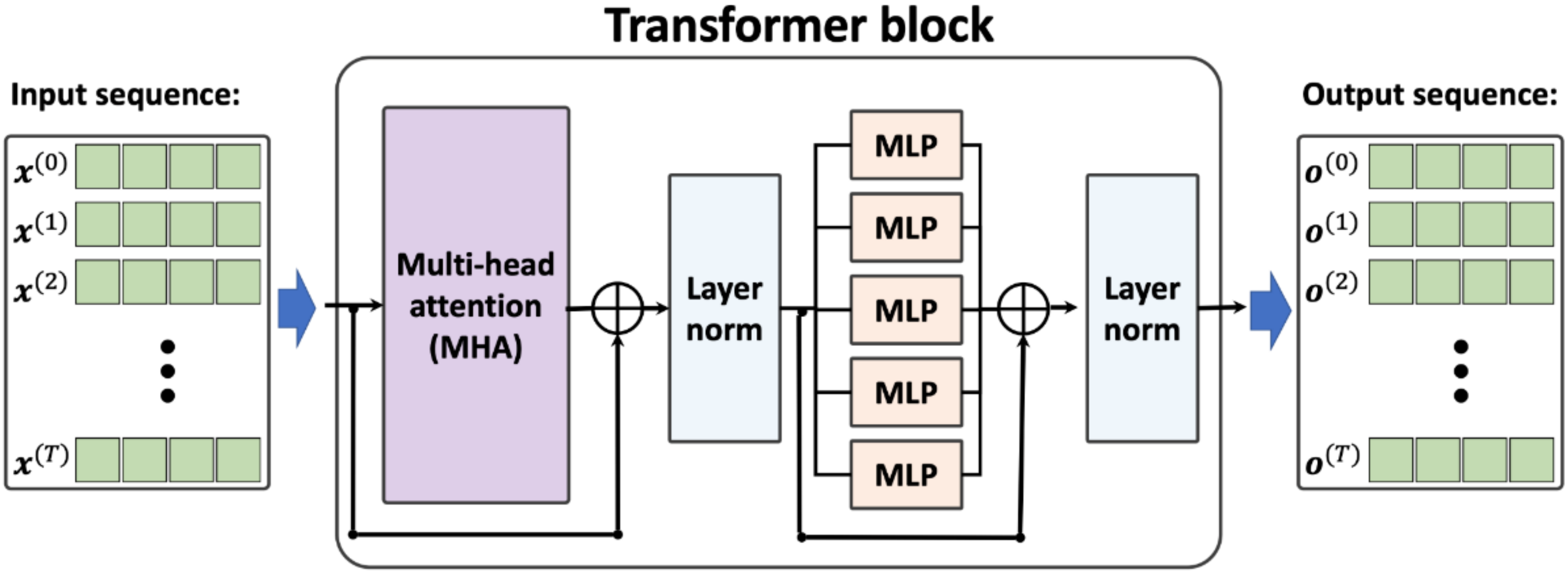
Classification transformer

- The simplest transformer we can build is a sequence classifier.
- The heart of the architecture will simply be a large chain of transformer blocks.
- All we need to do is work out how to feed it the input sequences, and how to transform the final output sequence into a single classification.

Producing a classification



Transformer block



Transformer block

- The block applies, in sequence:
  - a self attention layer,
  - a layer normalization,
  - a feed forward layer (a single MLP applied independently to each vector),
  - and another layer normalization.
  - residual connections are added around both, before the normalization.
- The order of the various components is not set in stone

Transformer block in Pytorch

```
class TransformerBlock(nn.Module):
  def __init__(self, k, heads):
    super().__init__()
    self.attention = SelfAttention(k, heads=heads)
    self.norm1 = nn.LayerNorm(k)
    self.norm2 = nn.LayerNorm(k)
    self.ff = nn.Sequential(
      nn.Linear(k, 4 * k),
      nn.ReLU(),
      nn.Linear(4 * k, k)
    )
```

```
def forward(self, x):
   attended = self.attention(x)
   x = self.norm1(attended + x)
   feedforward = self.ff(x)
   return self.norm2(feedforward + x)
```

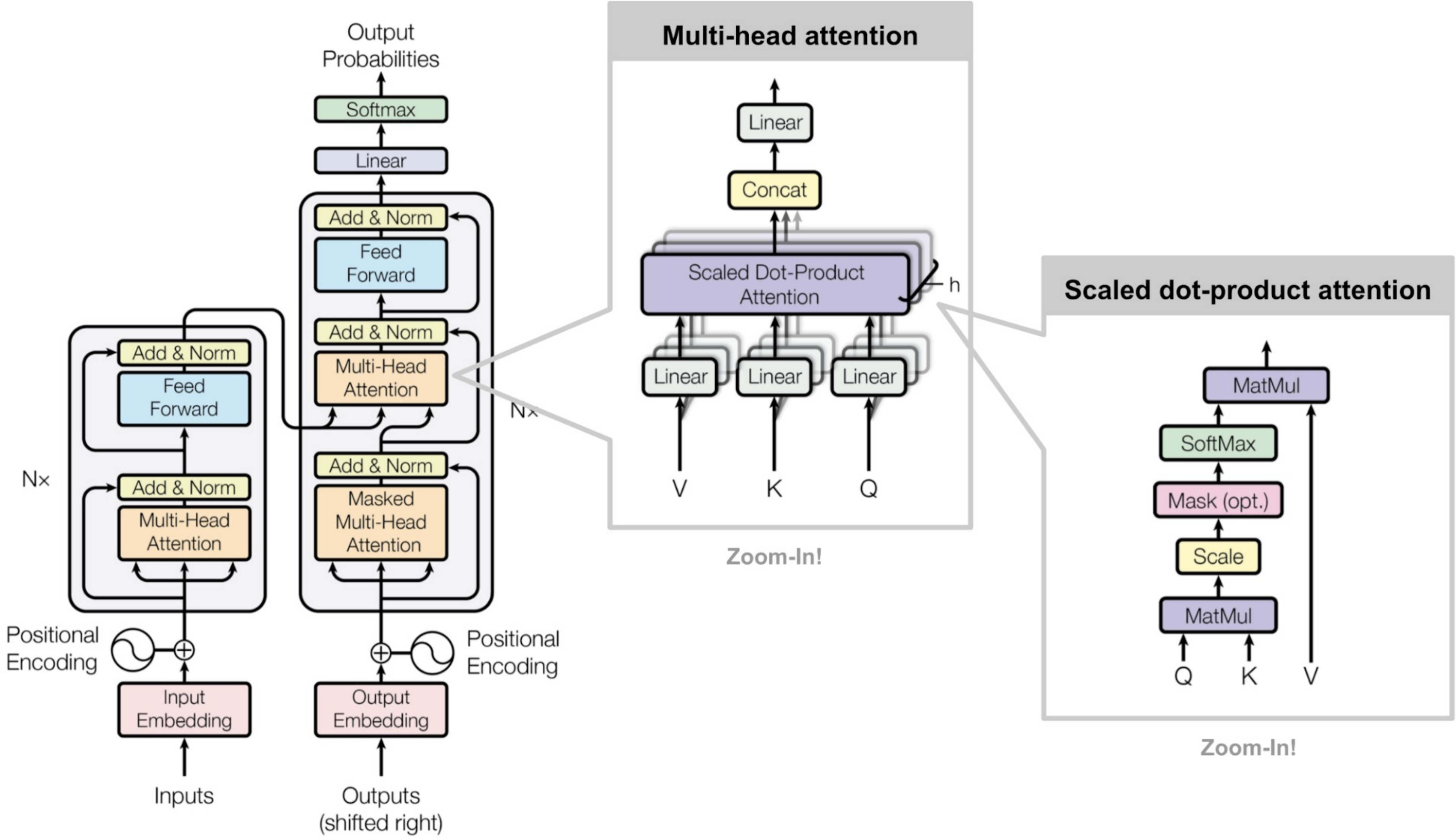
k: input size (word embedding + positional encoding)

Transformer for sequence transduction

Tranformer for sequence transduction

- The encoder maps an input sequence of symbol representations $x = (x_1, x_2, \dots, x_T)$ to a sequence of continuous representations $z = (z_1, z_2, \dots, z_T)$.
- Given z, the decoder then generates an output sequence $y = (y_1, y_2, \dots, y_{T'})$ of symbols one element at a time.
- At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.

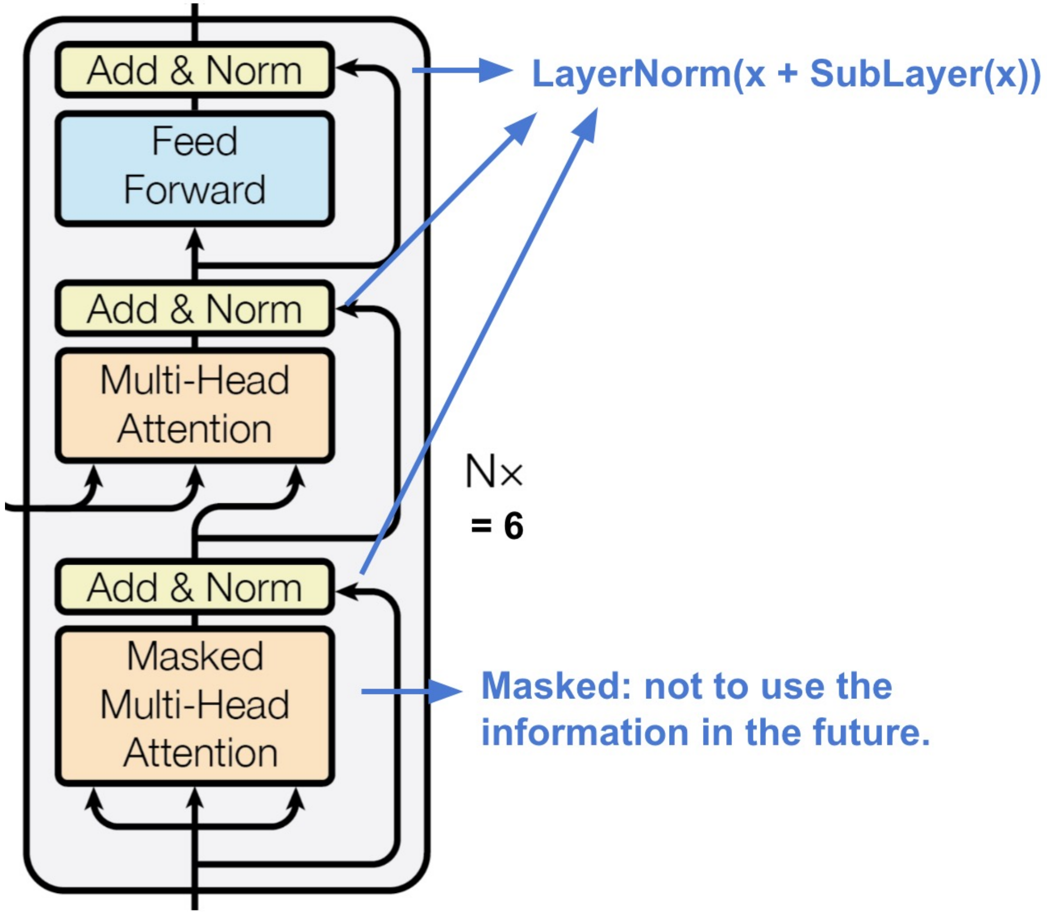
Full Architecture



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

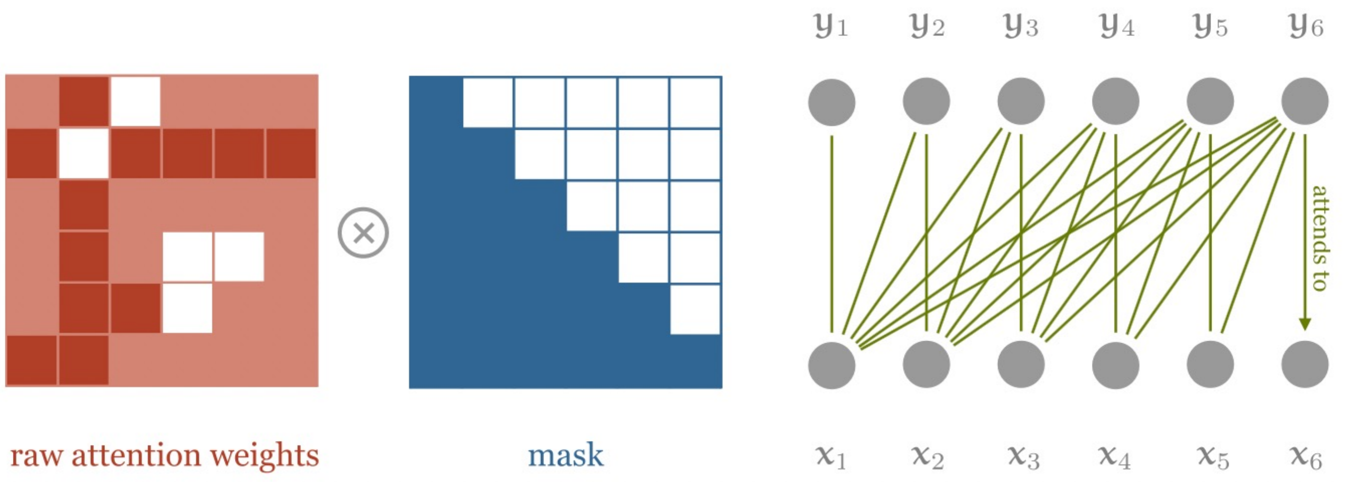
The Transformer’s Decoder

- A stack of $N = 6$ identical layers
- Each layer has two sub-layers of multi-head attention mechanisms and one sub-layer of fully-connected feed-forward network.
- Similar to the encoder, each sub-layer adopts a residual connection and a layer normalization.
- The first multi-head attention sub-layer is modified to prevent positions from attending to subsequent positions, as we don’t want to look into the future of the target sequence when predicting the current position.



Masked Multi-head Attention

- With a transformer, the output depends on the entire input sequence, so prediction of the next character becomes vacuously easy, just retrieve it from the input.
- To use self-attention as an autoregressive model, we'll need to ensure that it cannot look forward into the sequence. We do this by applying a mask to the matrix of dot products, before the softmax is applied.
- This mask disables all elements above the diagonal of the matrix.



- Note that the multiplication symbol is slightly misleading: we actually set the masked out elements (the white squares) to $-\infty$ to get very negative raw weights, then the softmax returns 0

https://peterbloem.nl/blog/transformers

Conclusion

Conclusion

- Dealing with sequence of unknown size is a hot topic
- Attention is nowadays a crucial mechanism
- Transformers are exploiting self-attention
- Transformers are powerful and generic
- A transformer is a significant alternative to a recurrent neural network
- Currently extended to computer vision

Train XLNet (2020)

- XLNet is a language model developed by CMU and Google Research which outperforms the previous SOTA model BERT (Bidirectional Encoder Representations from Transformers) on 20 language tasks including SQuAD, GLUE, and RACE; and has achieved SOTA (State Of The Art) results on 18 of these tasks.
- The auhors said: “We train XLNet-Large on 512 TPU v3 chips for 500K steps with an Adam optimizer, linear learning rate decay and a batch size of 2048, which takes about 2.5 days.”
- A Cloud TPU v3 device, which costs US$8 per hour on Google Cloud Platform, has eight independent embedded chips.
- The total training cost should be 512 (chips) * (US$8/8) (cost per chip) * 24 (hours) * 2.5 (days) = $30,720.”
- Hence, the training cost might be significant for SOTA methods.