Master Degree in Computer Science

Natural Language Processing

Sequence learning

Prof. Alfio Ferrara

Department of Computer Science, Università degli Studi di Milano
Room 7012 via Celoria 18, 20133 Milano, Italia alfio.ferrara@unimi.it

sed noli modo

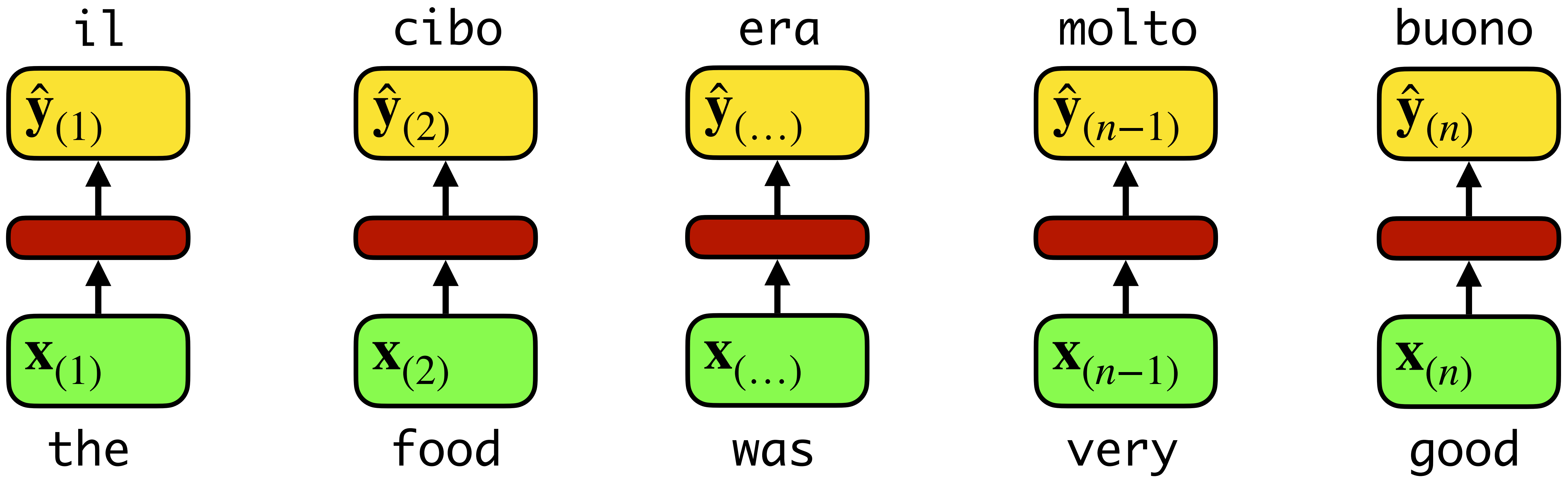
Limits in learning the sequence components in isolation

In this example, we learn to predict the word buono at time (n) from the word good

However, this **is not** sequence learning, in that the same prediction may be derived by learning to map pairs of words not appearing in a sequence

Our goal is to predict the word buono given the previous words `the food was very`

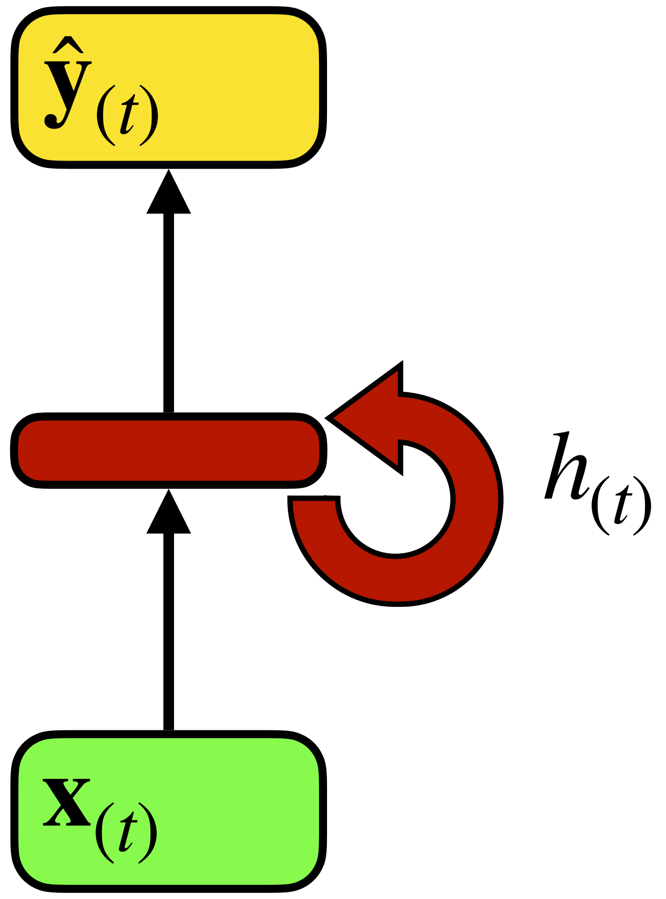
To achieve our goal, we need to link each step of the prediction to the previous ones, keeping a **memory** of what is learned along the sequence

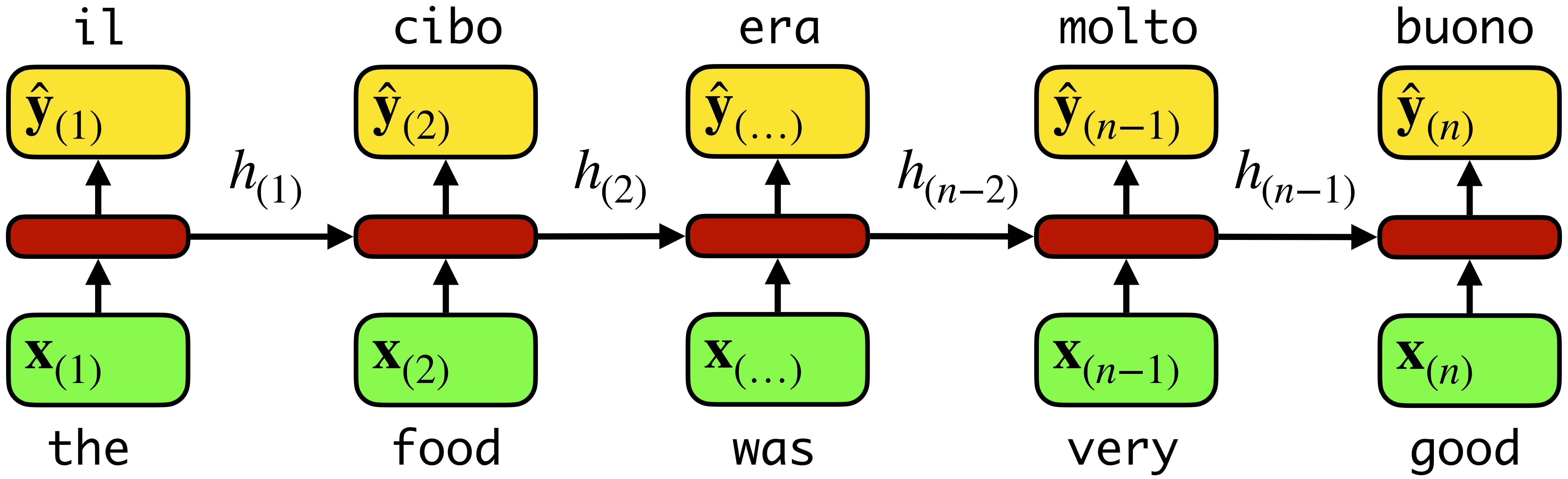


$$\hat{\mathbf{y}}_{(t)} = f\left(\mathbf{x}_{(t)}\right)$$

Recurrent Neural Networks

RNNs exploit an hidden state *h* in order to keep track of the previous learning steps during the learning process



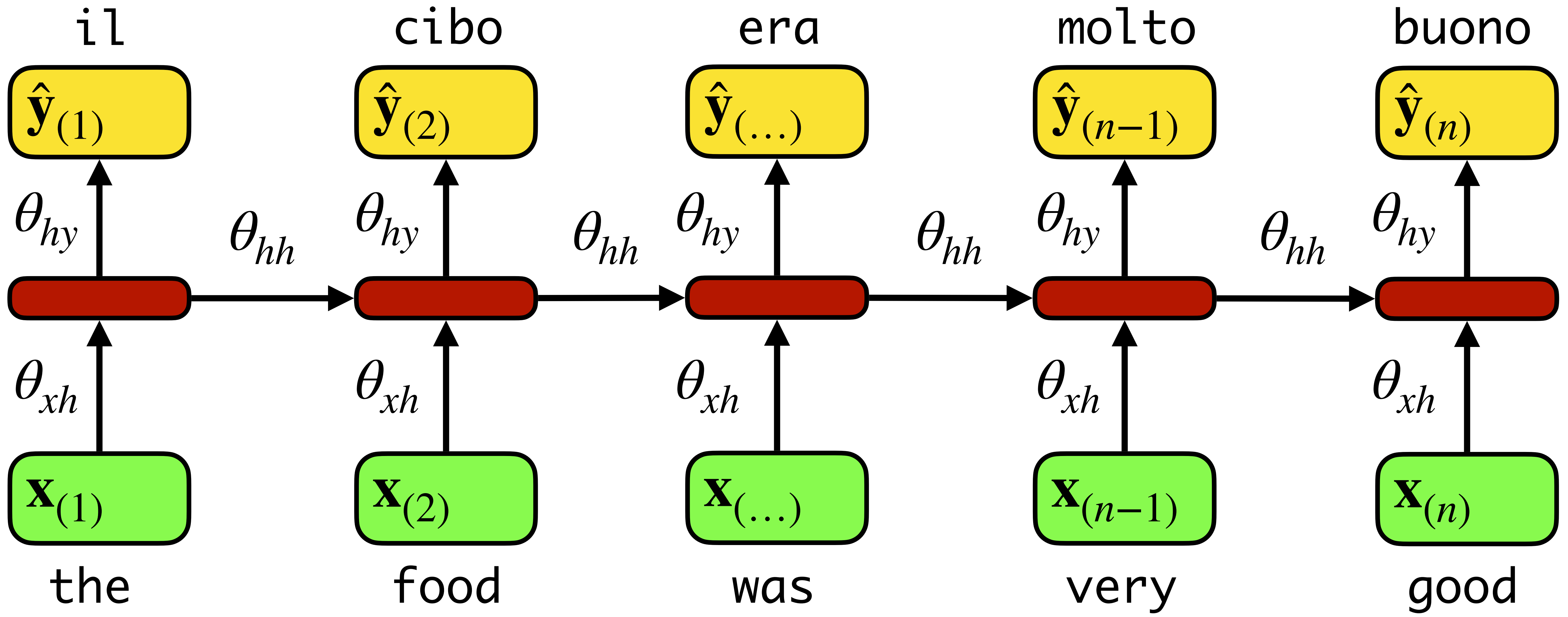


$$\hat{\mathbf{y}}_{(t)} = f\left(\mathbf{x}_{(t)}, h_{(t-1)}\right)$$

Recurrent Neural Networks

In order to implement the RNN architecture, we need three set of parameters: the weights of the transformation from input to hidden state θ_{xh}, the weights of the transformation from hidden state to hidden state θ_{hh}, and the weights of the transformation from the hidden state to the output θ_{hy}

Note that we use the same parameters along the whole sequence

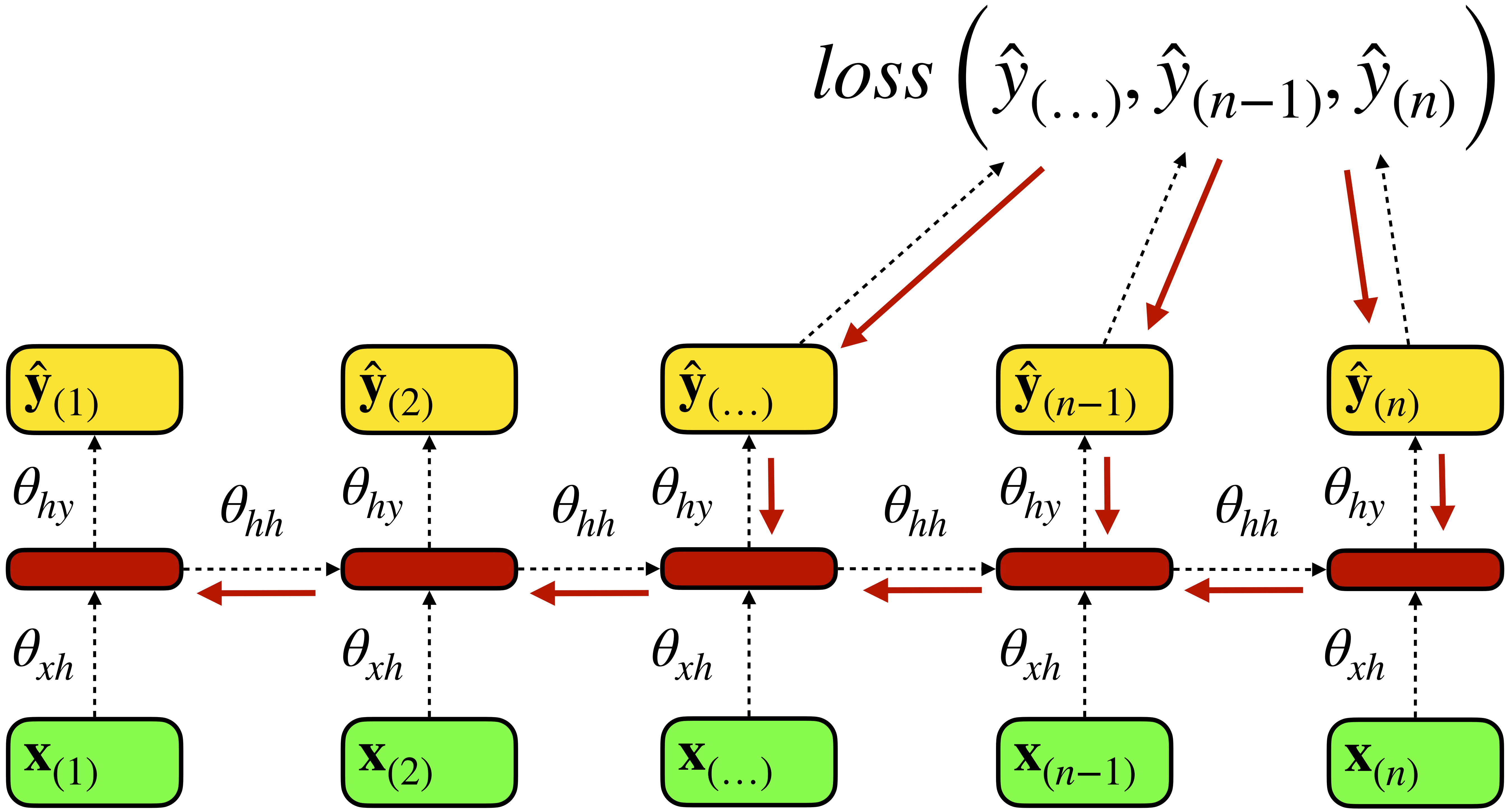


$$h_{(t)} = f\left(\theta_{hh}^T h_{(t-1)} + \theta_{xh}^T x_{(t)}\right)$$

$$\hat{y}_{(t)} = \theta_{hy}^T h_{(t)}$$

Training RNNs

In order to compute a loss function we may select all (or some) of the predictions and compute a single loss for each prediction. Then, we simply back-propagate the gradients of the loss function through all the outputs. This strategy is called **back propagation through time (BPTT)**.



Problem

Back to $h_{(1)}$, BPTT requires a high number of gradient computations and many factors of the product by θ_{hh}

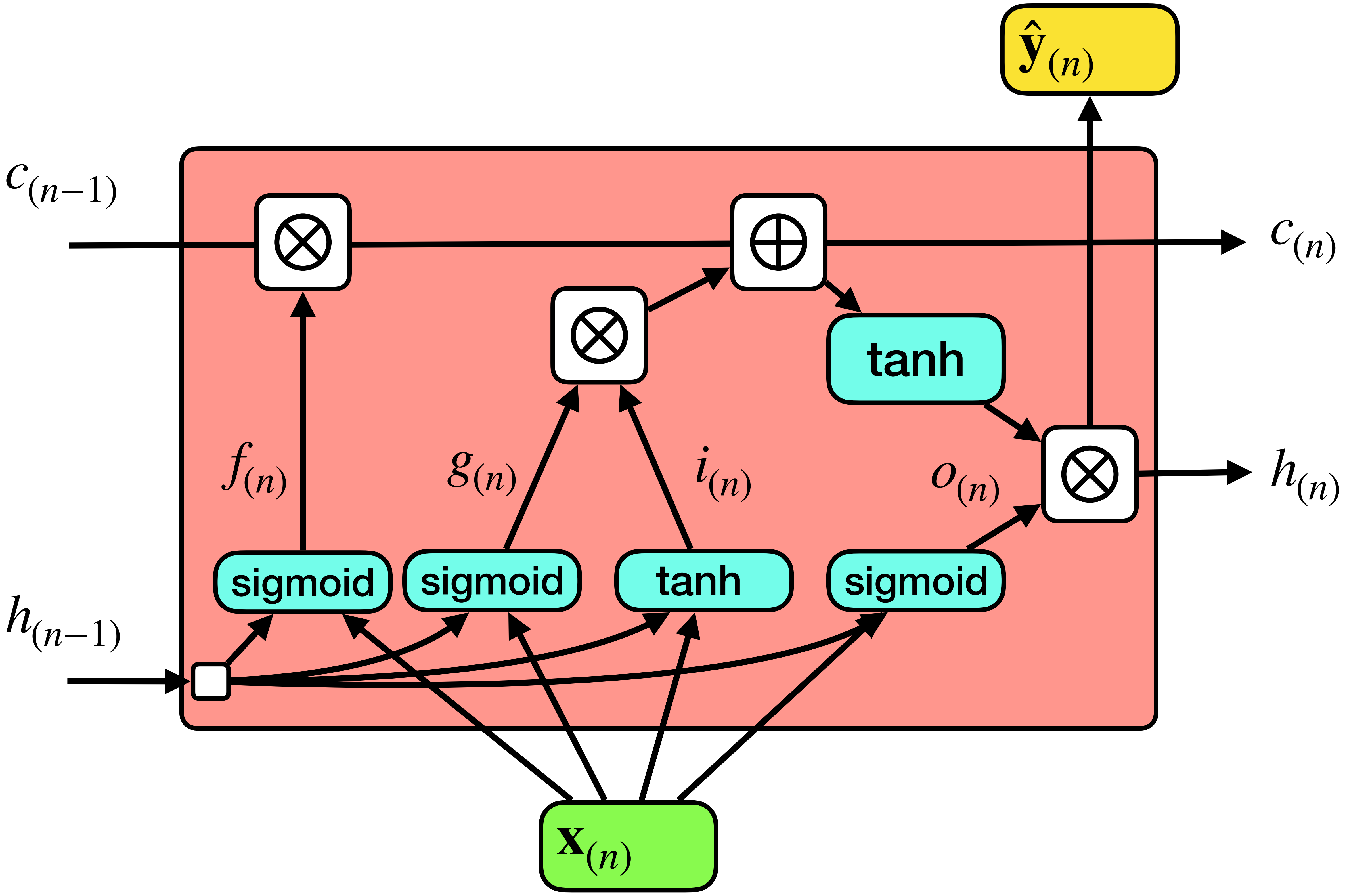
If many values are > 1, the gradients become bigger and bigger (**exploding gradients**)

If many values are < 1, the gradients become smaller and smaller (**vanishing gradients**)

Long term dependencies in sentences are then a problem

Long Short-Term Memory (LSTM)

There are strategies for dealing with the gradients problems of RNN like choosing ReLU as the activation function or initializing weights to the identity matrix. However, the most effective solution is LSTM.



LSTM intuition

The state of a LSTM is represented by two distinct vectors, $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$. Ideally, the first represents a short-term memory and the second a long-term memory.

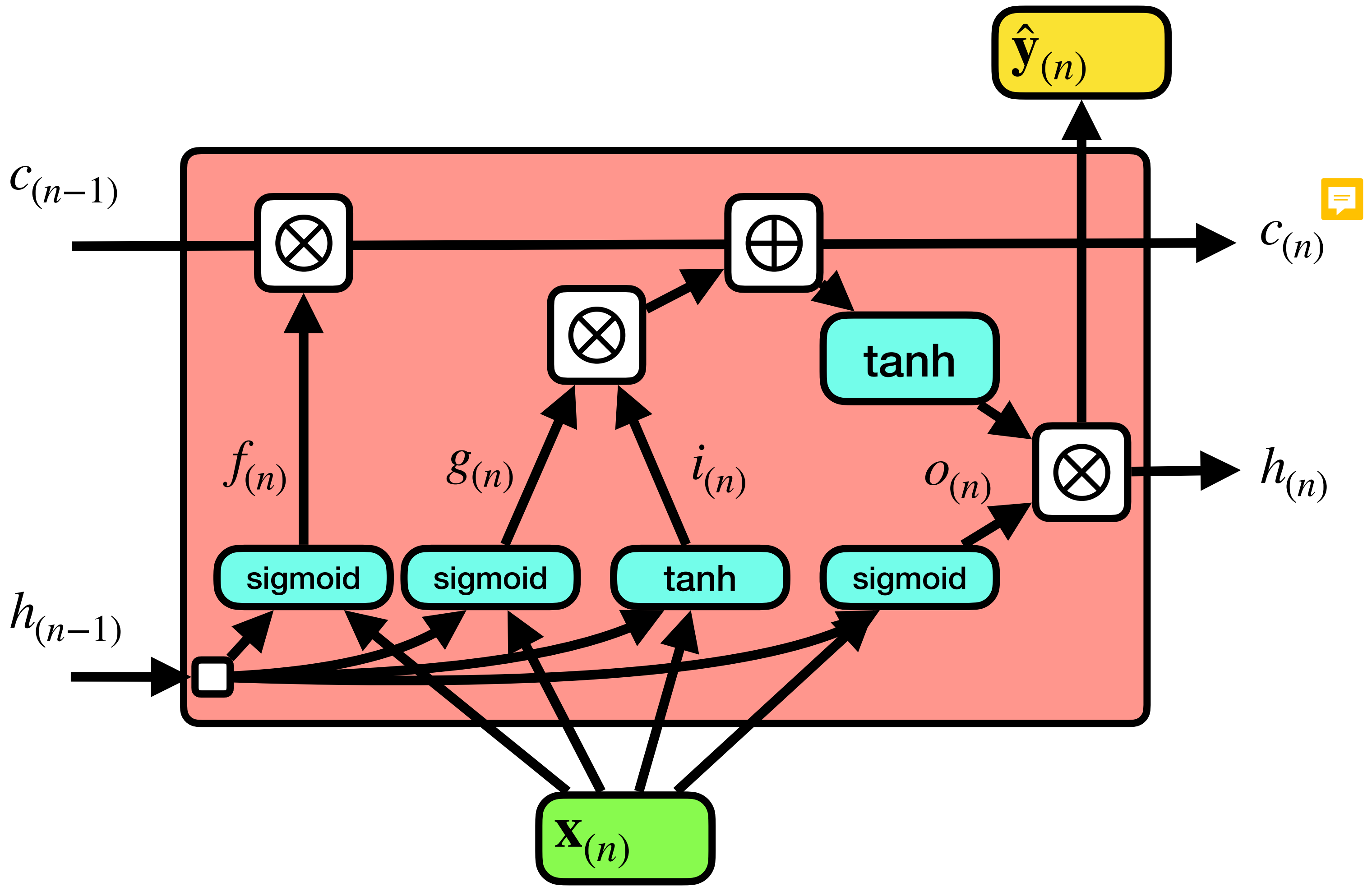
By traversing the network, $\mathbf{c}_{(t)}$ passes through a forget gate that drops part of the memory and then it adds some fresh information coming from the current input.

Long Short-Term Memory (LSTM)

Information flow

The current input vector $\mathbf{x}_{(n)}$ and the previous short-term state $\mathbf{h}_{(n-1)}$ are fed into four different fully connected layers:

- $\mathbf{g}_{(n)}$ analyzes the input and the short-term memory. The output is partially stored in the long-term memory
- **Gate controllers**: the output is between 0 and 1 and it is fed to element-wise multiplications, so that a value of 0 blocks the information
  - $\mathbf{f}_{(n)}$ is a **forget gate controller** that controls which part of the long-term state should be erased
  - $\mathbf{i}_{(n)}$ is a **input gate controller** that controls which part of $\mathbf{g}_{(n)}$ should be added to the long-term memory
  - $\mathbf{o}_{(n)}$ is a **output gate controller** which controls which part of the long-term memory should be exploited to predict the output and determine the subsequent short-term memory



Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. " O'Reilly Media, Inc.".

Long Short-Term Memory (LSTM)

Computation

Let be $\theta_{xi}, \theta_{xf}, \theta_{xo}, \theta_{xg}$ the weights connecting the input to the four layers of the network and $\theta_{hi}, \theta_{hf}, \theta_{ho}, \theta_{hg}$ the weight matrices from the four layers to the previous short-term memory vector

$$\mathbf{i}_{(n)} = \sigma(\theta_{xi}^T \cdot \mathbf{x}_{(n)} + \theta_{hi}^T \cdot \mathbf{h}_{(n-1)})$$

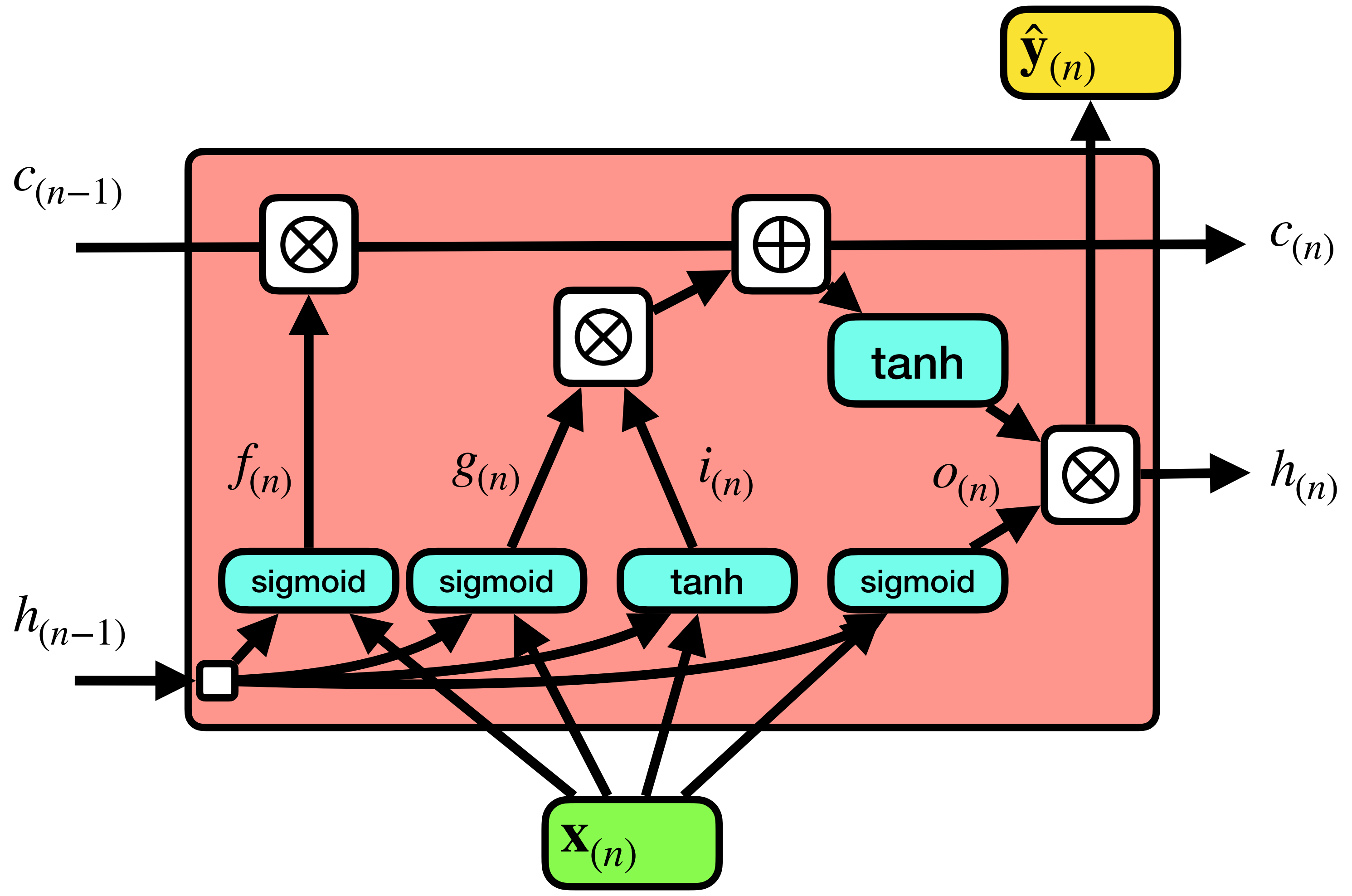
$$\mathbf{f}_{(n)} = \sigma(\theta_{xf}^T \cdot \mathbf{x}_{(n)} + \theta_{hf}^T \cdot \mathbf{h}_{(n-1)})$$

$$\mathbf{o}_{(n)} = \sigma(\theta_{xo}^T \cdot \mathbf{x}_{(n)} + \theta_{ho}^T \cdot \mathbf{h}_{(n-1)})$$

$$\mathbf{g}_{(n)} = tanh(\theta_{xg}^T \cdot \mathbf{x}_{(n)} + \theta_{hg}^T \cdot \mathbf{h}_{(n-1)})$$

$$\mathbf{c}_{(n)} = \mathbf{f}_{(n)} \otimes \mathbf{c}_{(n-1)} + \mathbf{i}_{(n)} \otimes \mathbf{g}_{(n)}$$

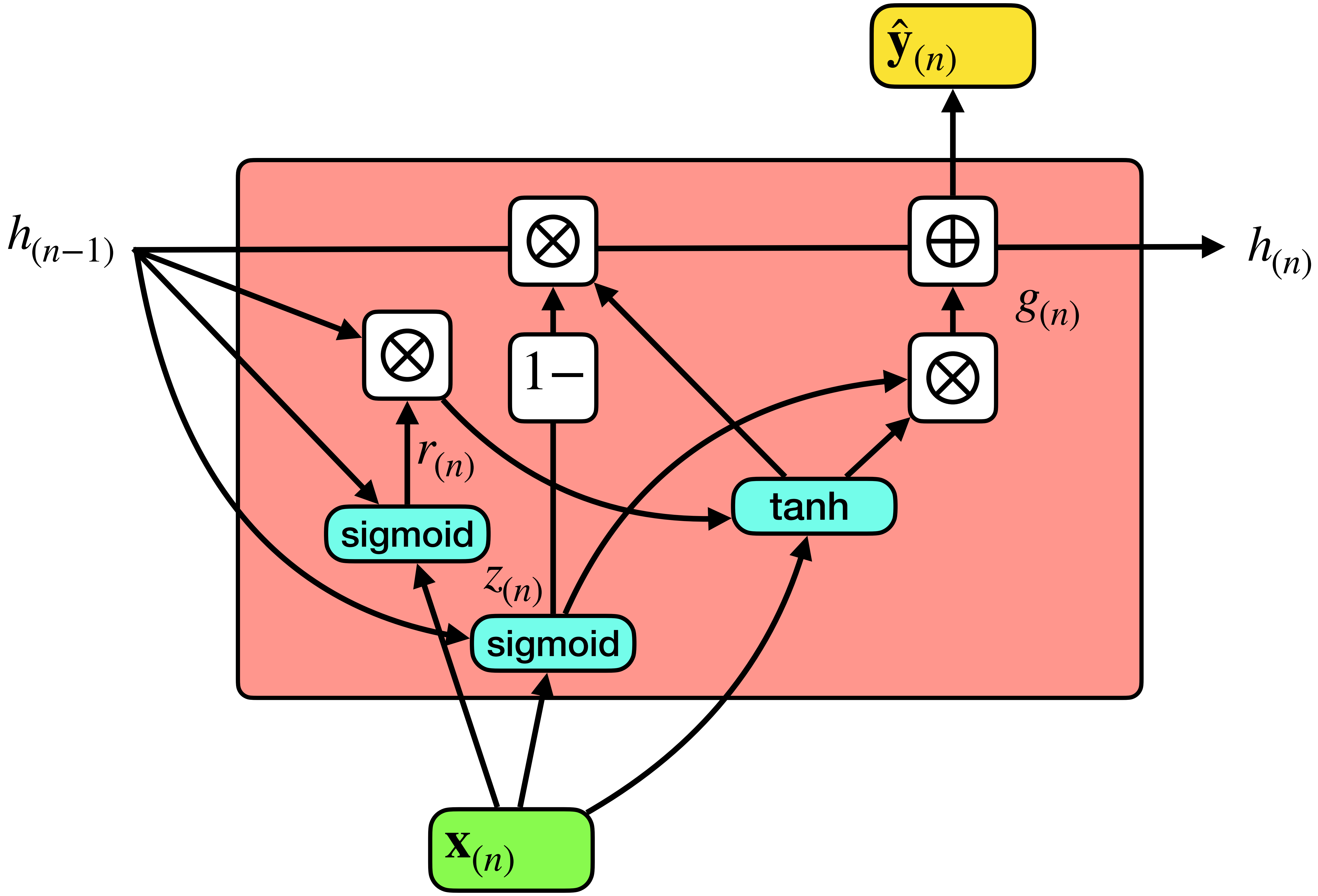
$$\hat{\mathbf{y}}_{(n)} = \mathbf{h}_{(n)} = \mathbf{o}_{(n)} \otimes tanh\left(\mathbf{c}_{(n)}\right)$$



Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. " O'Reilly Media, Inc.".

Gated Recurrent Unit (GRU)

Variant of LSTM that embeds in the same vector the long-term and the short-term memory



$$\mathbf{z}_{(n)} = \sigma(\theta_{xz}^T \cdot \mathbf{x}_{(n)} + \theta_{hz}^T \cdot \mathbf{h}_{(n-1)})$$

$$\mathbf{r}_{(n)} = \sigma(\theta_{xr}^T \cdot \mathbf{x}_{(n)} + \theta_{hr}^T \cdot \mathbf{h}_{(n-1)})$$

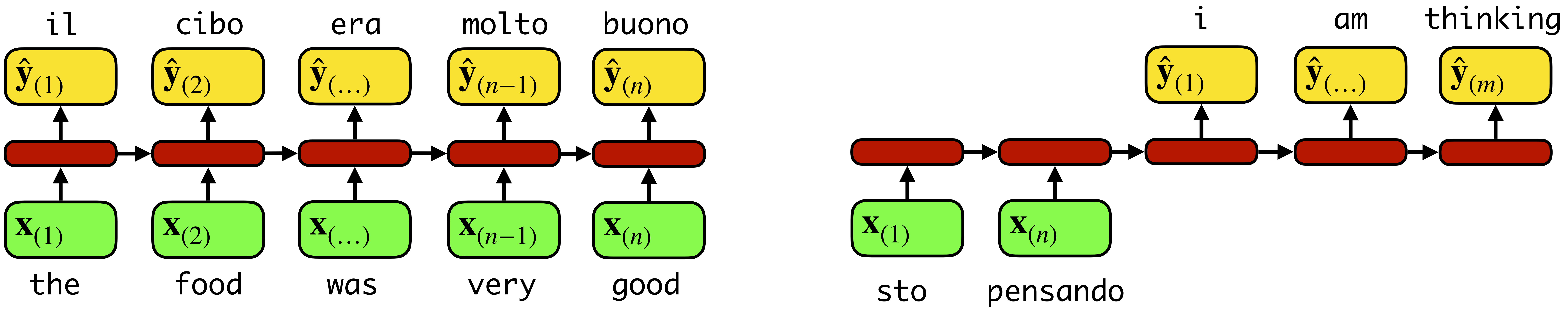
$$\mathbf{g}_{(n)} = tahn\left(\theta_{xg}^T \cdot \mathbf{x}_{(n)} + \theta_{hg}^T \cdot \left(\mathbf{r}_{(n)} \otimes \mathbf{h}_{(n-1)}\right)\right)$$

$$\hat{\mathbf{y}}_{(n)} = \mathbf{h}_{(n)} = \left(1 - \mathbf{z}_{(n)}\right) \otimes \mathbf{h}_{(n-1)} + \mathbf{z}_{(n)} \otimes \mathbf{g}_{(t)}$$

Limits of RNNs

RNNs have some structural limitation when applied to sequence2sequence learning:

- **Sequences are forced to have the same length**
- Sequence learning cannot be parallelized
- We cannot keep a long memory



Sequence to Sequence mapping

Sequence to Sequence (Seq2Seq) is the task of transforming an **input sequence** $x_1, x_2, \ldots, x_n$ into an **output sequence** $y_1, y_2, \ldots, y_m$

Seq2Seq is sometimes presented as the problem of machine translation, because it is the most successful application of Seq2Seq in recent years, but it is in general the task of transforming any kind of sequence into another sequence (e.g., question answering, dialogues, spelling correction, etc.)

Seq2Seq can be seen as the problem of choosing the most probable target sequence y given the input x

Formal definition	Issues
$\hat{y} = \underset{y}{\arg\max}\, \underbrace{p(\overset{\text{Candidate target sequence}}{y} \mid \overset{\text{Input sequence}}{x}, \overset{\text{Parameters}}{\theta})}_{\text{Model}}$	**Learning:** find the parameters θ **Search:** find a way to compute $\arg\max$

Conditional Language Models

We know that we can use RNN's to implement Language Models. A LM is a tool for estimating the probability of a sequence (of words, chars, etc.) as

$$p(y_1, y_2, \ldots, y_m) \approx \prod_{t=1}^{n} p(y_t \mid y_1, \ldots, t_{t-1})$$

Now, we want to do the same, but the estimation is now conditioned by a context x, which, in our case, is the input sequence. We call this models Conditional Language Models (CLM)

Input sequence

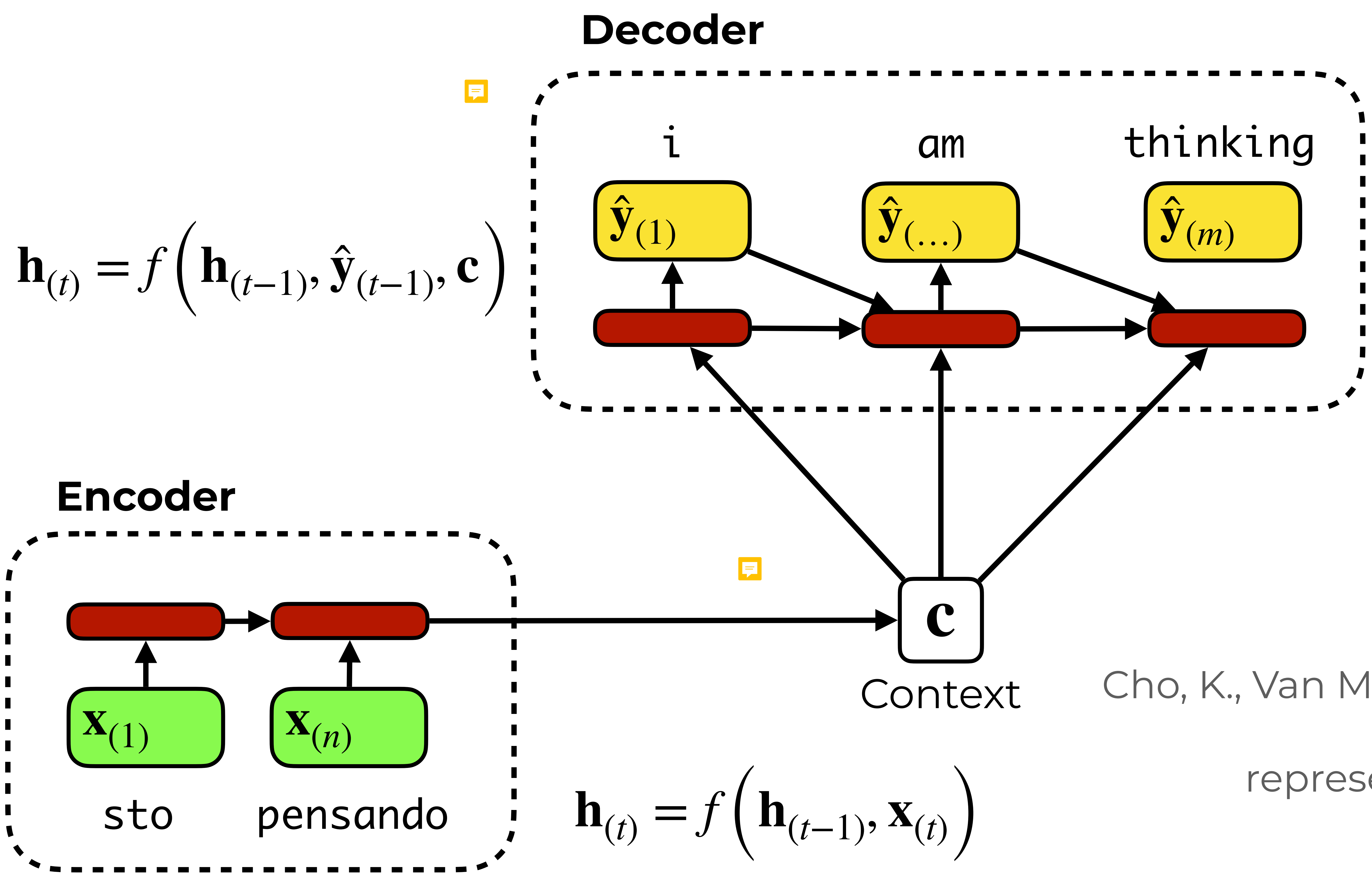
$$p(y_1, y_2, \ldots, y_m \mid x) \approx \prod_{t=1}^{n} p(y_t \mid y_1, \ldots, t_{t-1}, x)$$

I am thinking

sto pensando

Encoder Decoder Networks

A **Encoder Decoder network**, is a model consisting of two RNNs called the encoder and decoder. The encoder reads an input sequence and outputs a single vector, and the decoder reads that vector to produce an output sequence. This frees us from sequence **length and order**.



$$\mathbf{h}_{(t)} = f\left(\mathbf{h}_{(t-1)}, \hat{\mathbf{y}}_{(t-1)}, \mathbf{c}\right)$$

$$\mathbf{h}_{(t)} = f\left(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)}\right)$$

The **encoder** is an RNN that reads each symbol of an input sequence sequentially. After reading the end of the sequence, the hidden state of the RNN is a summary $\mathbf{c}$ of the whole input sequence.

The **decoder** is another RNN which is trained to generate the output sequence by predicting the next symbol $\hat{\mathbf{y}}_{(t)}$ given the hidden state $\mathbf{h}_{(t-1)}$. However, both $\hat{\mathbf{y}}_{(t)}$ and $\mathbf{h}_{(t)}$ are also conditioned on $\hat{\mathbf{y}}_{(t-1)}$ and on the summary $\mathbf{c}$ of the input sequence.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

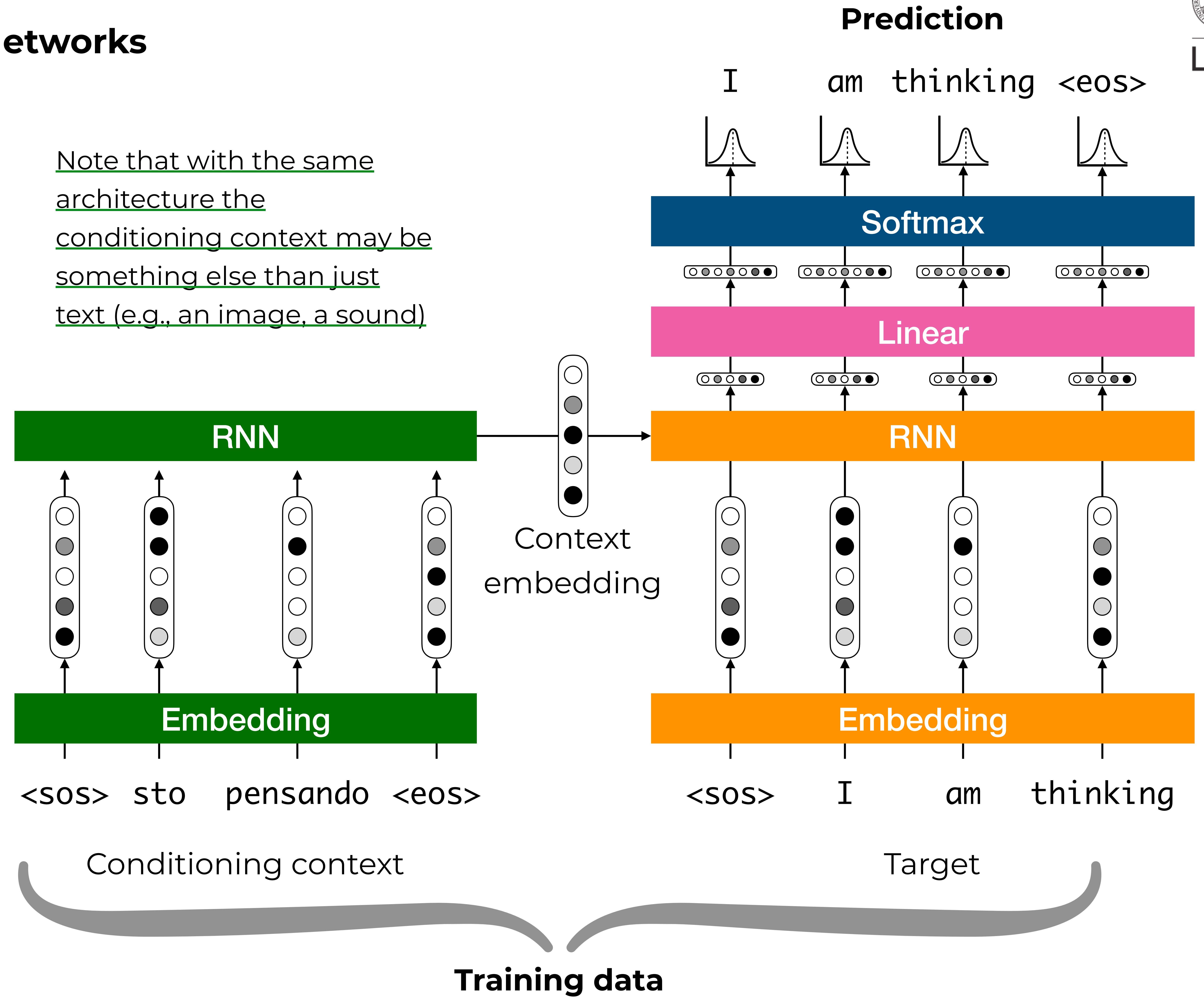
Encoder Decoder RNN Networks

In this model, the **decoder** acts as a **language model** while the **encoder** provides the **conditional context** that affects the LM prediction.

In particular:

- We use the encoder to provide a vector representation of the whole context
- Then we feed the decoder with both the target and the context vectors
- We compute the probability of the next word step-by-step in order to generate the prediction

Note that with the same architecture the conditioning context may be something else than just text (e.g., an image, a sound)



How to select predictions

The encoder-decoder architecture approximates a solution for the equation:

$$\hat{y} = \arg\max_{y} p(y \mid x) = \arg\max_{y} \prod_{t=1}^{n} p(y \mid y_1, \ldots, y_{t-1}, x)$$

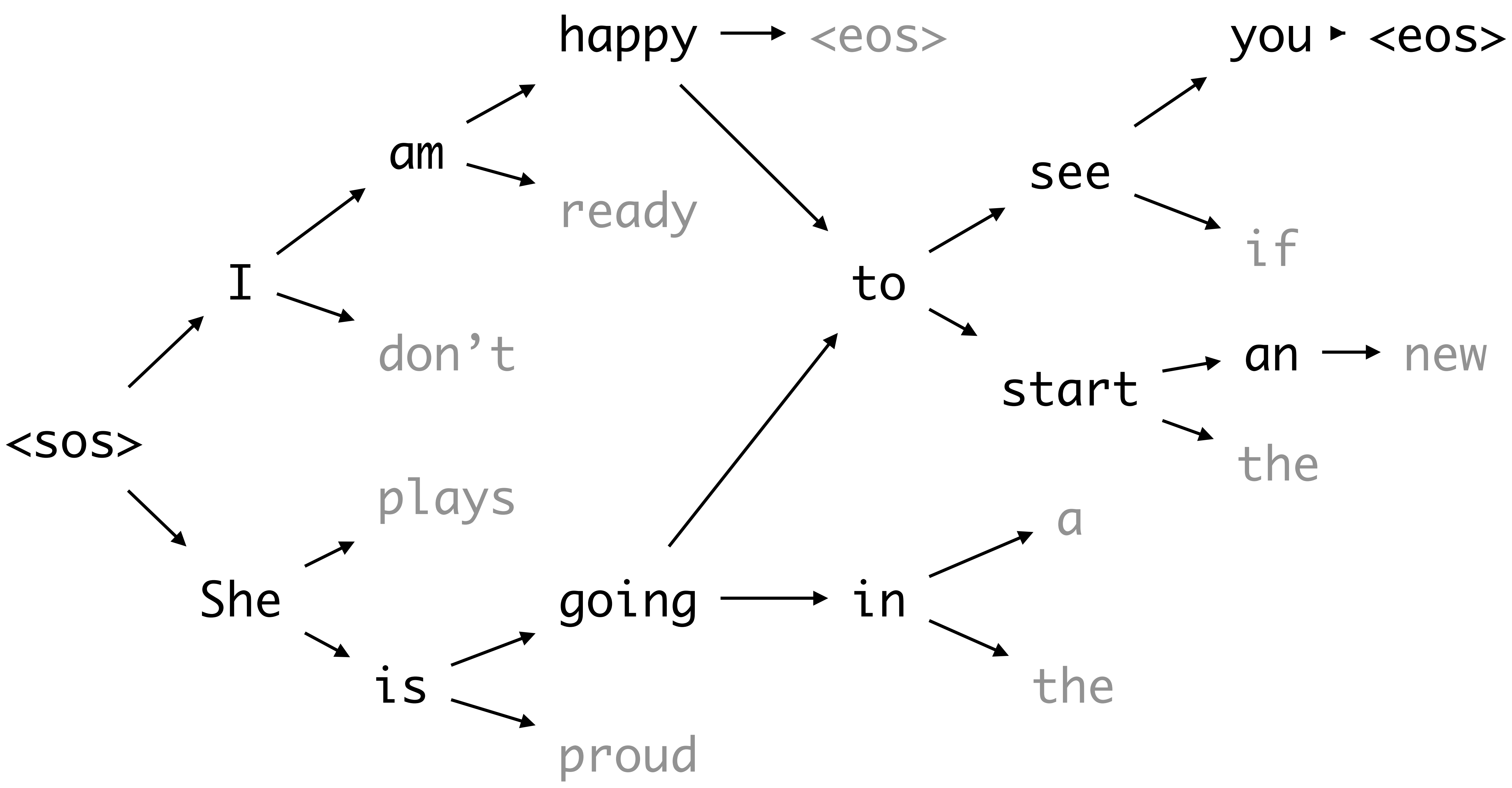
But the computation of the exact solution for the argmax operation is unfeasible because we need to search in a space of size $\mid V \mid^n$

Greedy solution

At each step T, when we predict the Tth word, we select the most probable. However, the best token at step T does not necessarily lead to the best sequence.

Beam search

At each step T, we keep the best *n* options and we continue expanding the selection, by discarding the branches that are not leading to complete sequence



How to evaluate translation and seq2seq systems

Automatic translation (and seq2seq in general) does not have a unique solution and, for this reason, it is very difficult to evaluate

A well known and accepted method is called **BLEU (Bilingual Evaluation Understudy)**

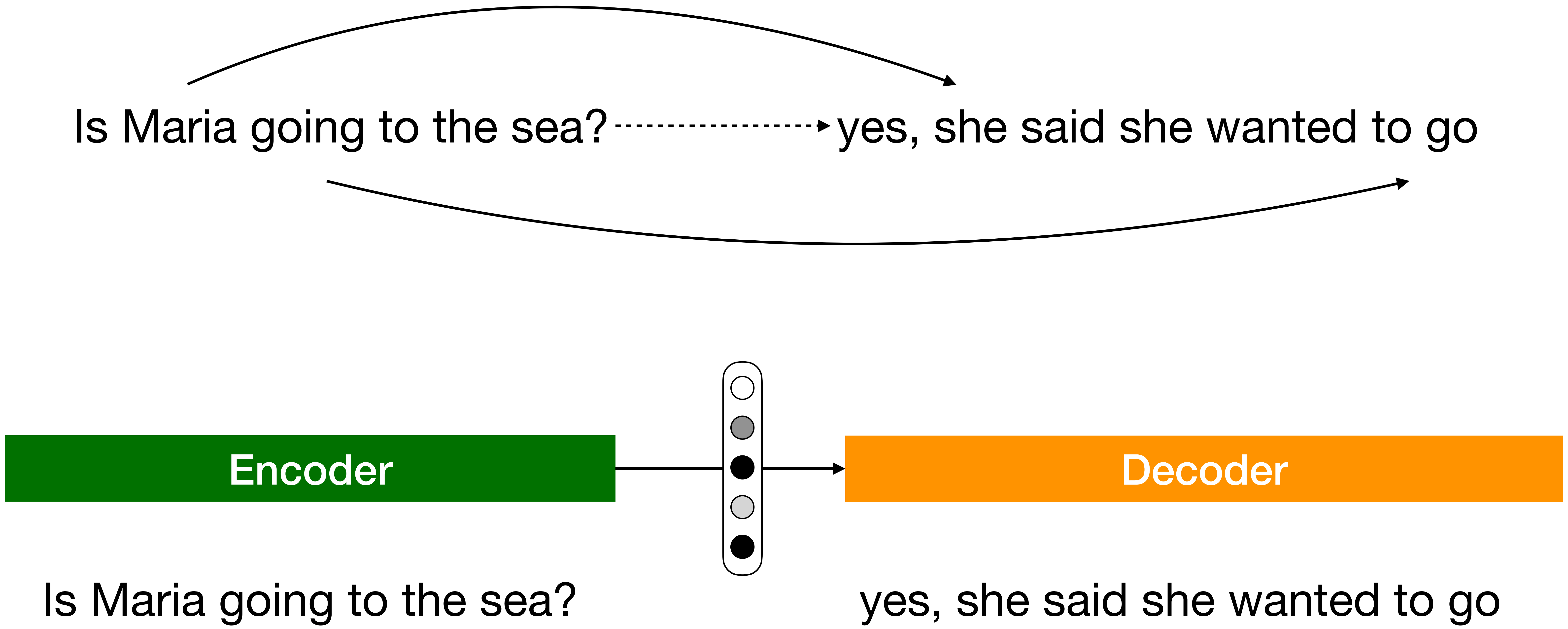
The idea is that for any n-gram in the predicted sequence, we check whether this n-gram appears in the target sequence

$$\exp\left(\min\left(0,1-\frac{len(target)}{len(pred)}\right)\right)\prod_{n=1}^{k} p_n^{\frac{1}{2^n}}$$

Where p_n is the n-gram precision and len(target) and len(pred) denote the length of the target and predicted sentences, respectively. Note that using BLEU longer sequences are preferred to short sequences.

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. Proceedings of the 40th annual meeting of the Association for Computational Linguistics (pp. 311–318).

Potential problems with the encoder-decoder architecture



Is Maria going to the sea? yes, she said she wanted to go

Encoder

Decoder

Is Maria going to the sea?

yes, she said she wanted to go

The encoder compresses all information into a single vector.
The decoder receives only one representation of the context.
Moreover, long term dependencies weakly influence the representation of the context.

Idea: We provide a mechanism that makes it possible for the decoder, at each step, to select which source parts are more relevant

This mechanism is called **Attention**

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

Input: The decoder at step t computes its internal state h_t and receives all the encoder internal states $s_1, \dots, s_n$

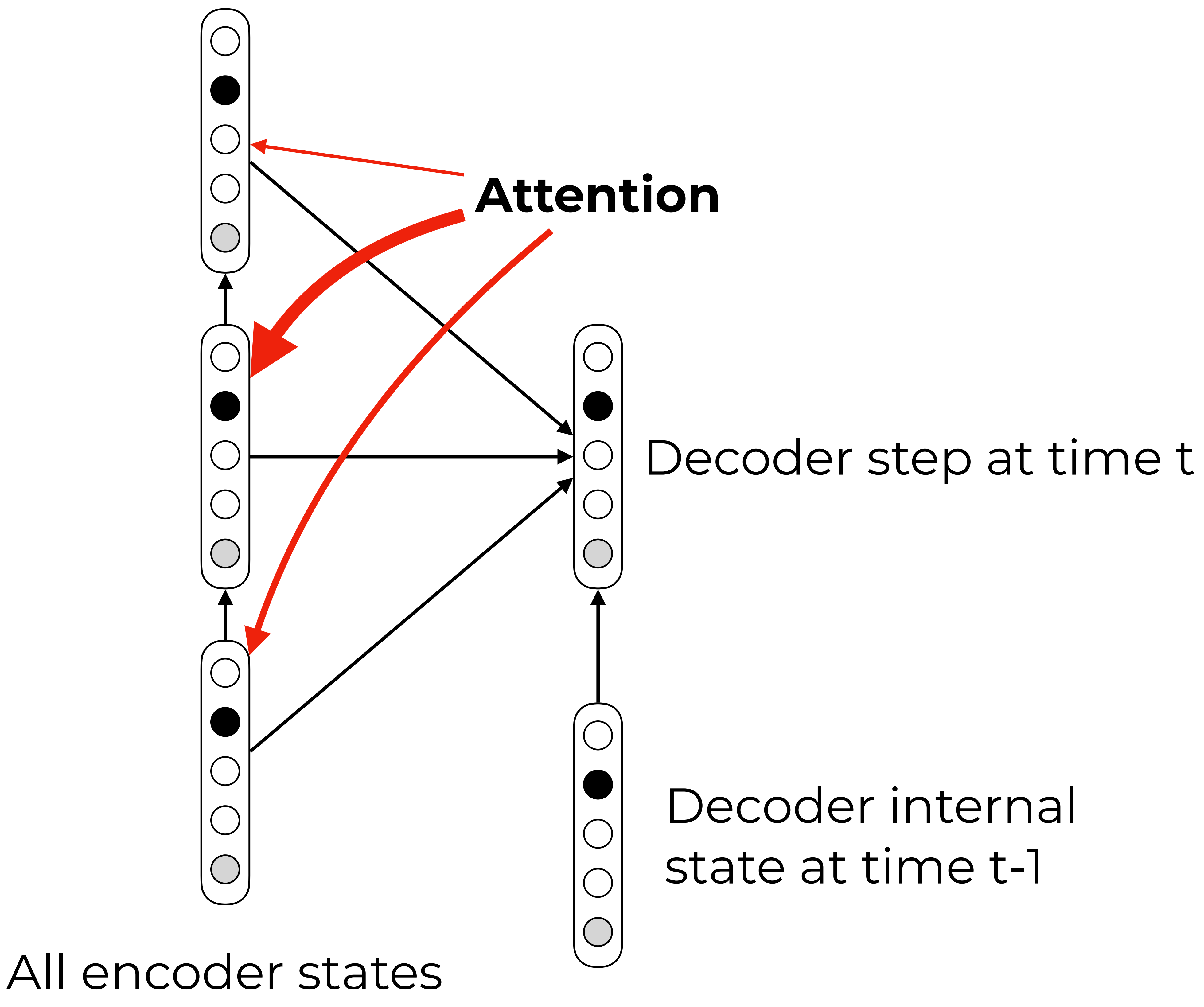
Score: compute how relevant is each encoder state s_k with respect to h_t as a score $\sigma(h_t, s_k)$

Weights: transform scores in weights by softmax

$$a_k = \frac{\exp(\sigma(h_t, s_k))}{\sum_{i=1}^{n} \exp(\sigma(h_t, s_i))}$$

Select: apply weights to the encoder states to discount their relevance

$$\sum_{k=1}^{n} a_k s_k$$



How we instruct the network on what is important in the input source?

No need! Since everything is differentiable, we can learn the attention and everything else!

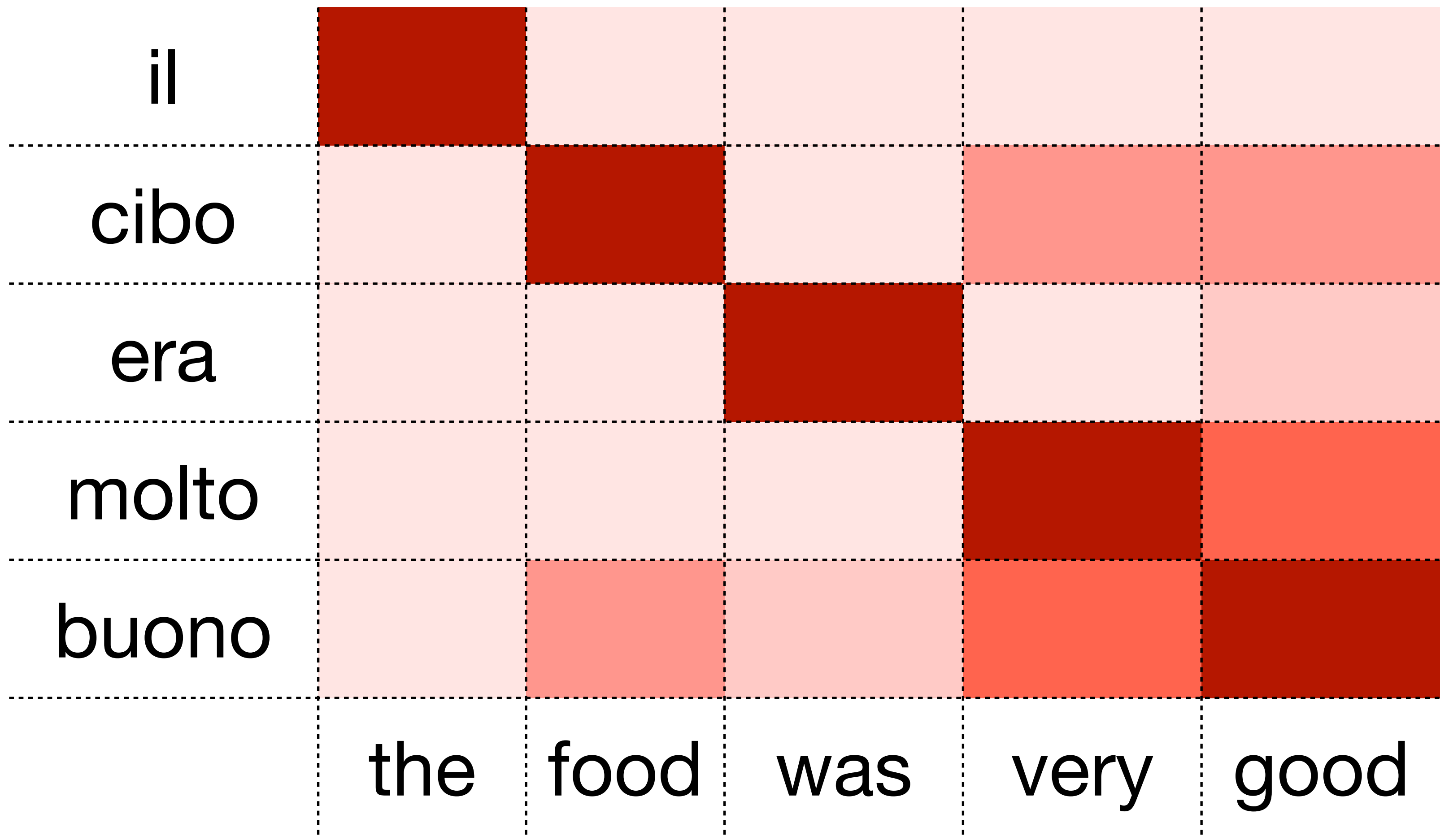
Thus, any differentiable function can be used to compute the attention scores. Examples:

$$\sigma(h_t, s_k) = h_t^T s_k \qquad \sigma(h_t, s_k) = h_t^T \theta \ s_k \qquad \sigma(h_t, s_k) = \theta_1^T \tanh\left(\theta_2 \left(h_t \oplus s_k\right)\right)$$

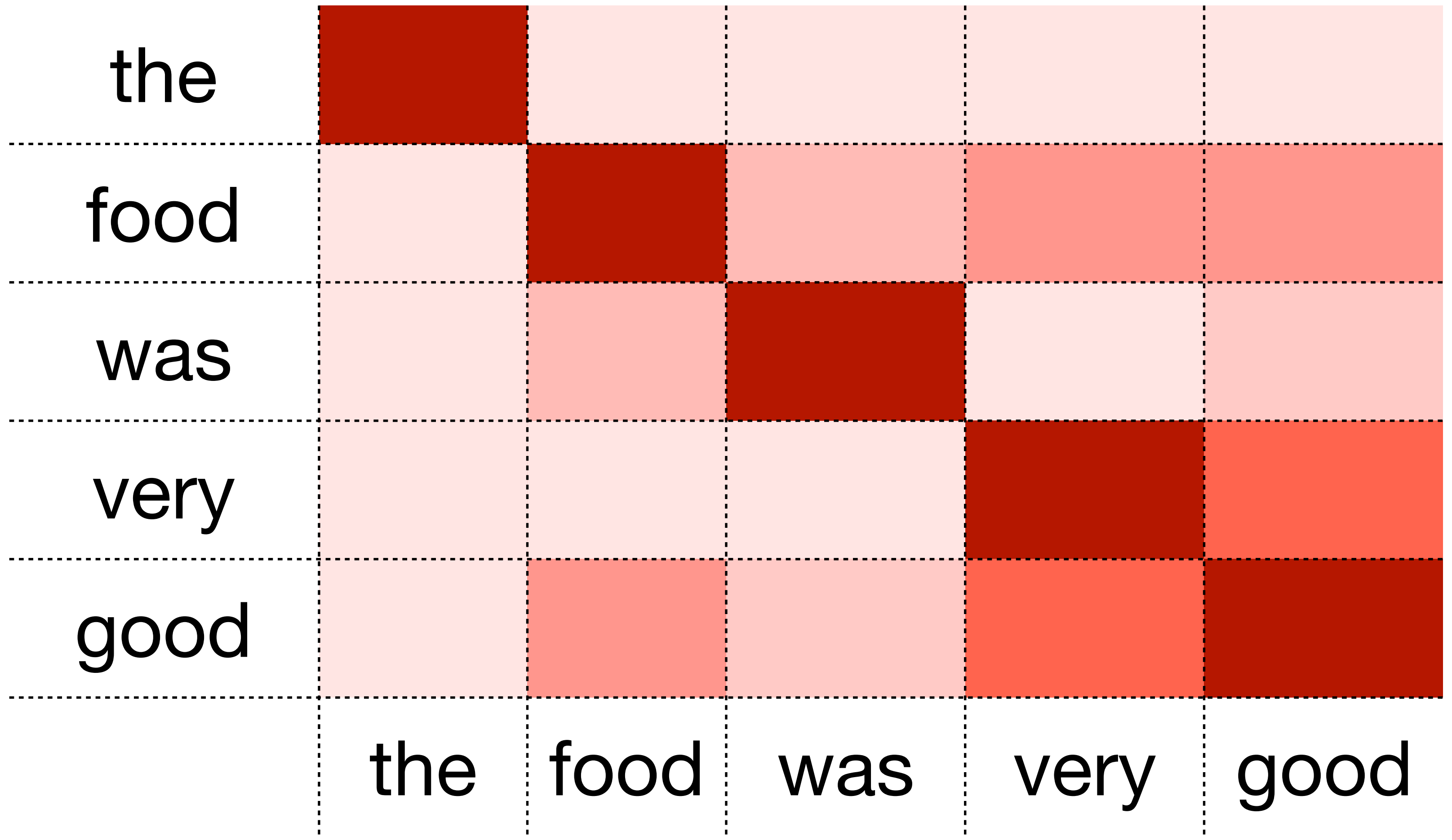
Intuition: not all the word have the same importance for a sequence learning task (for example for translation)

Learning what is important is like having a selecting memory

Attention



Self-Attention



Master Degree in Computer Science

Natural Language Processing

Transformers

Prof. Alfio Ferrara

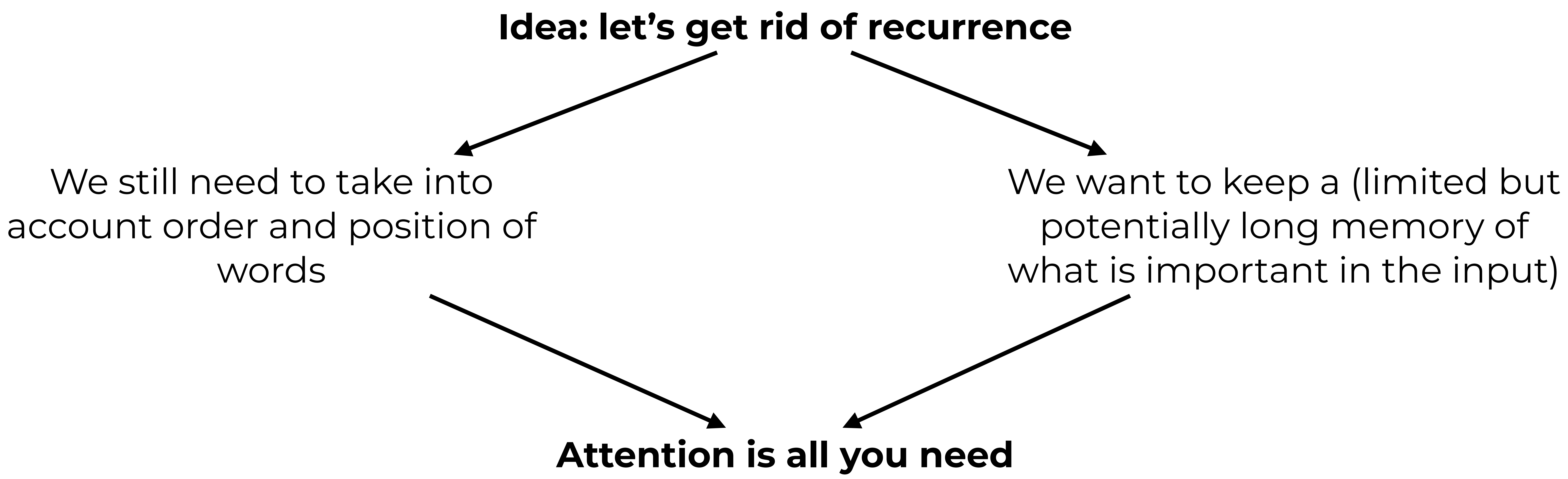
Department of Computer Science, Università degli Studi di Milano
Room 7012 via Celoria 18, 20133 Milano, Italia alfio.ferrara@unimi.it

sed noli modo

Limits of RNNs

RNNs have some structural limitation when applied to sequence2sequence learning:

- Sequences are forced to have the same length
- **Sequence learning cannot be parallelized**
- **We cannot keep a long memory**



Idea: let’s get rid of recurrence

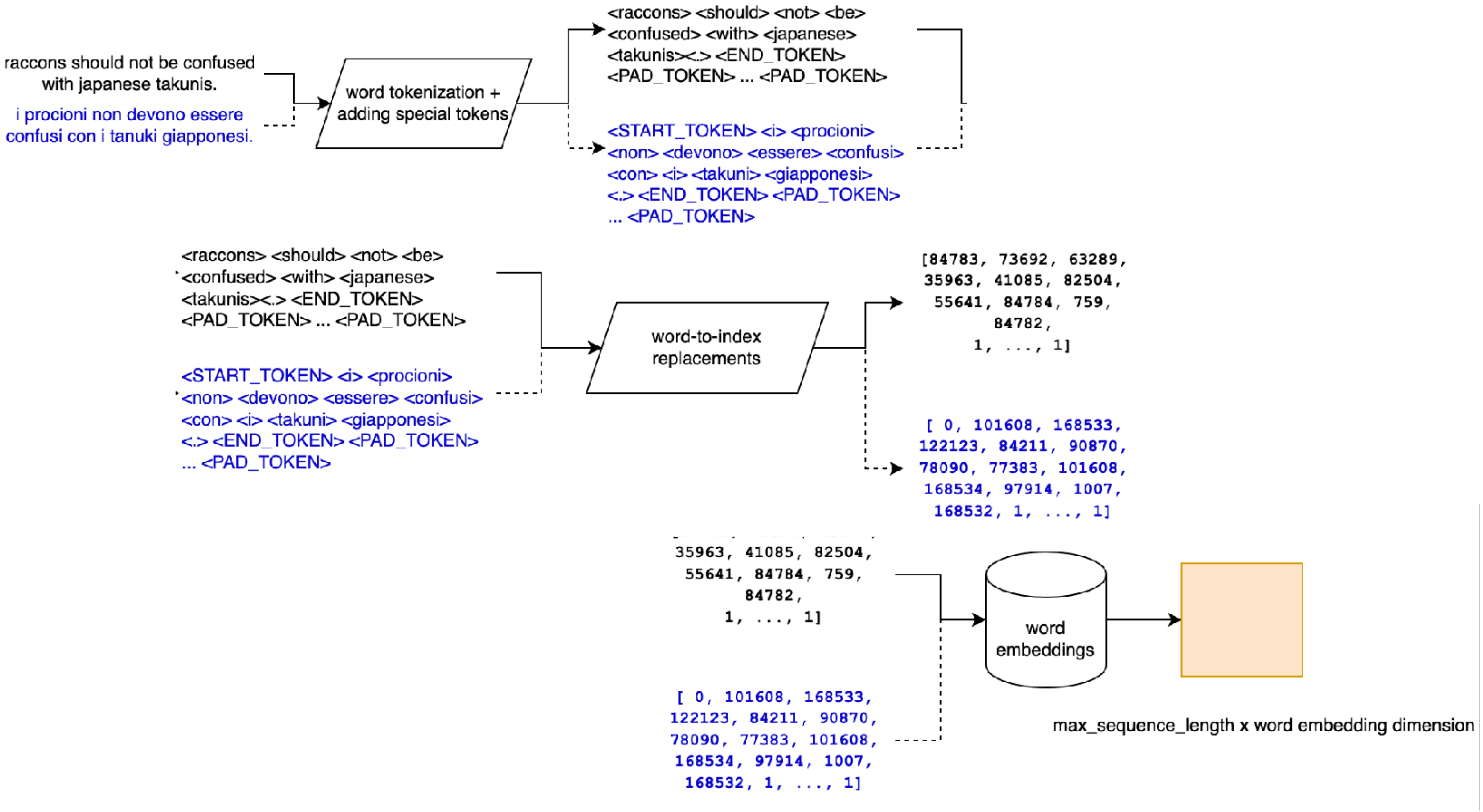
We still need to take into account order and position of words

We want to keep a (limited but potentially long memory of what is important in the input)

Attention is all you need

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.

Input embedding



raccons should not be confused with japanese takunis.

i procioni non devono essere confusi con i tanuki giapponesi.

word tokenization + adding special tokens

<raccons> <should> <not> <be> <confused> <with> <japanese> <takunis><.> <END_TOKEN> <PAD_TOKEN> ... <PAD_TOKEN>

<START_TOKEN> <i> <procioni> <non> <devono> <essere> <confusi> <con> <i> <takuni> <giapponesi> <.> <END_TOKEN> <PAD_TOKEN> ... <PAD_TOKEN>

<raccons> <should> <not> <be> <confused> <with> <japanese> <takunis><.> <END_TOKEN> <PAD_TOKEN> ... <PAD_TOKEN>

<START_TOKEN> <i> <procioni> <non> <devono> <essere> <confusi> <con> <i> <takuni> <giapponesi> <.> <END_TOKEN> <PAD_TOKEN> ... <PAD_TOKEN>

word-to-index replacements

```
[84783, 73692, 63289,
 35963, 41085, 82504,
 55641, 84784, 759,
 84782,
 1, ..., 1]
```

```
[ 0, 101608, 168533,
122123, 84211, 90870,
78090, 77383, 101608,
168534, 97914, 1007,
168532, 1, ..., 1]
```

```
35963, 41085, 82504,
55641, 84784, 759,
84782,
1, ..., 1]
```

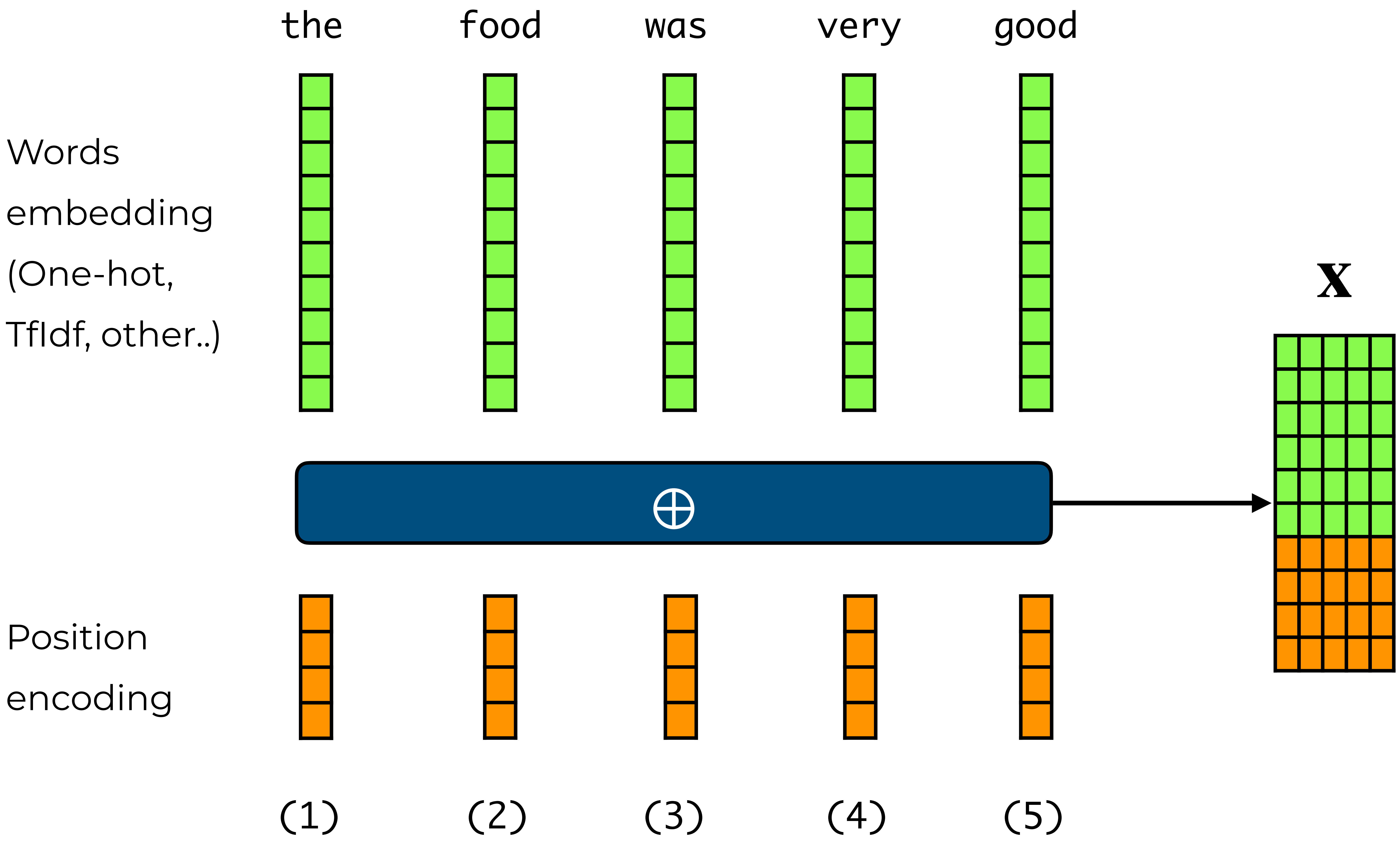
```
[ 0, 101608, 168533,
122123, 84211, 90870,
78090, 77383, 101608,
168534, 97914, 1007,
168532, 1, ..., 1]
```

word embeddings

max_sequence_length x word embedding dimension

Positional encoding

The next step to get rid of recurrence is to find another way to keep track of word position

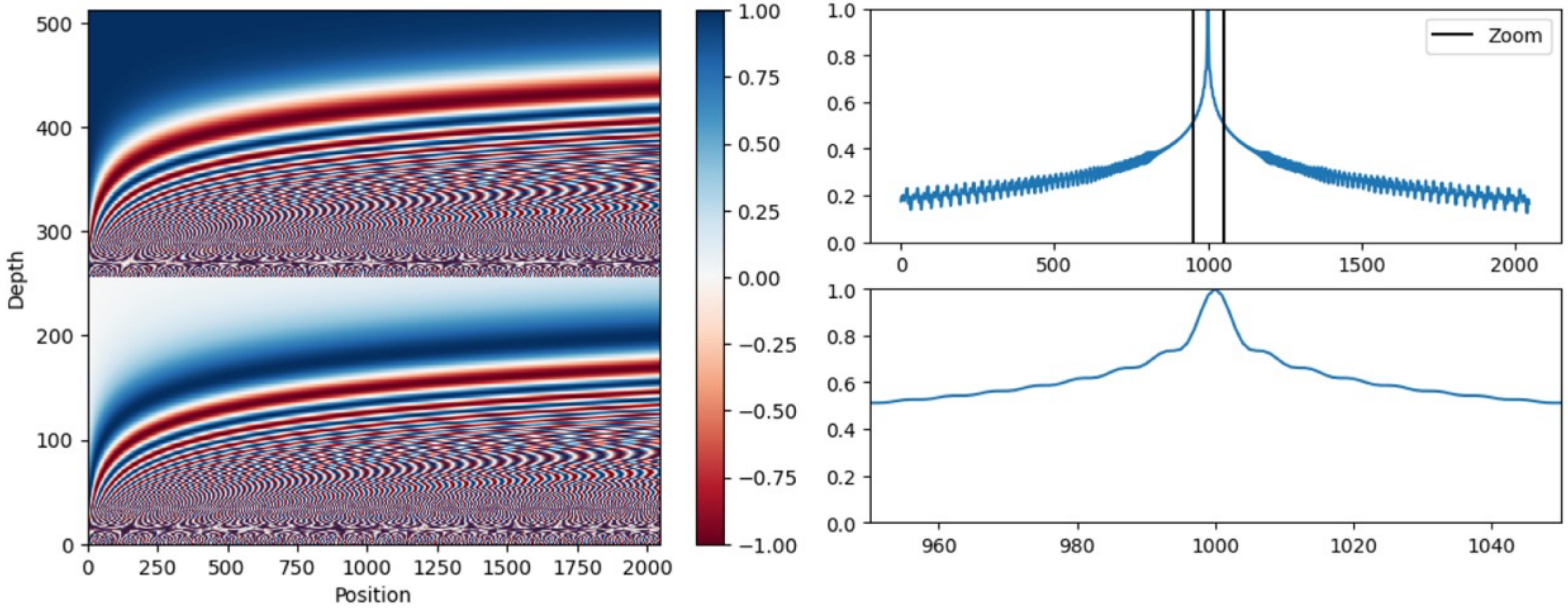


By encoding the position of words this way, we keep the order but we loose the memory.

However, with the attention mechanism we can control the impact of the important parts.

$$PE(\textbf{position}, 2i) = \sin\left(\frac{\textbf{position}}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(\textbf{position}, 2i+1) = \cos\left(\frac{\textbf{position}}{10000^{\frac{2i}{d_{model}}}}\right)$$



Positional encoding values (left) and the value of the dot product of position 1000 with neighbouring positions (right). Figure adapted from TensorFlow's Transformers tutorial.

By encoding the position of words this way, we keep the order but we loose the memory. However, with the attention mechanism we can control the impact of the important parts. The position vectors are **added** to the word vectors.

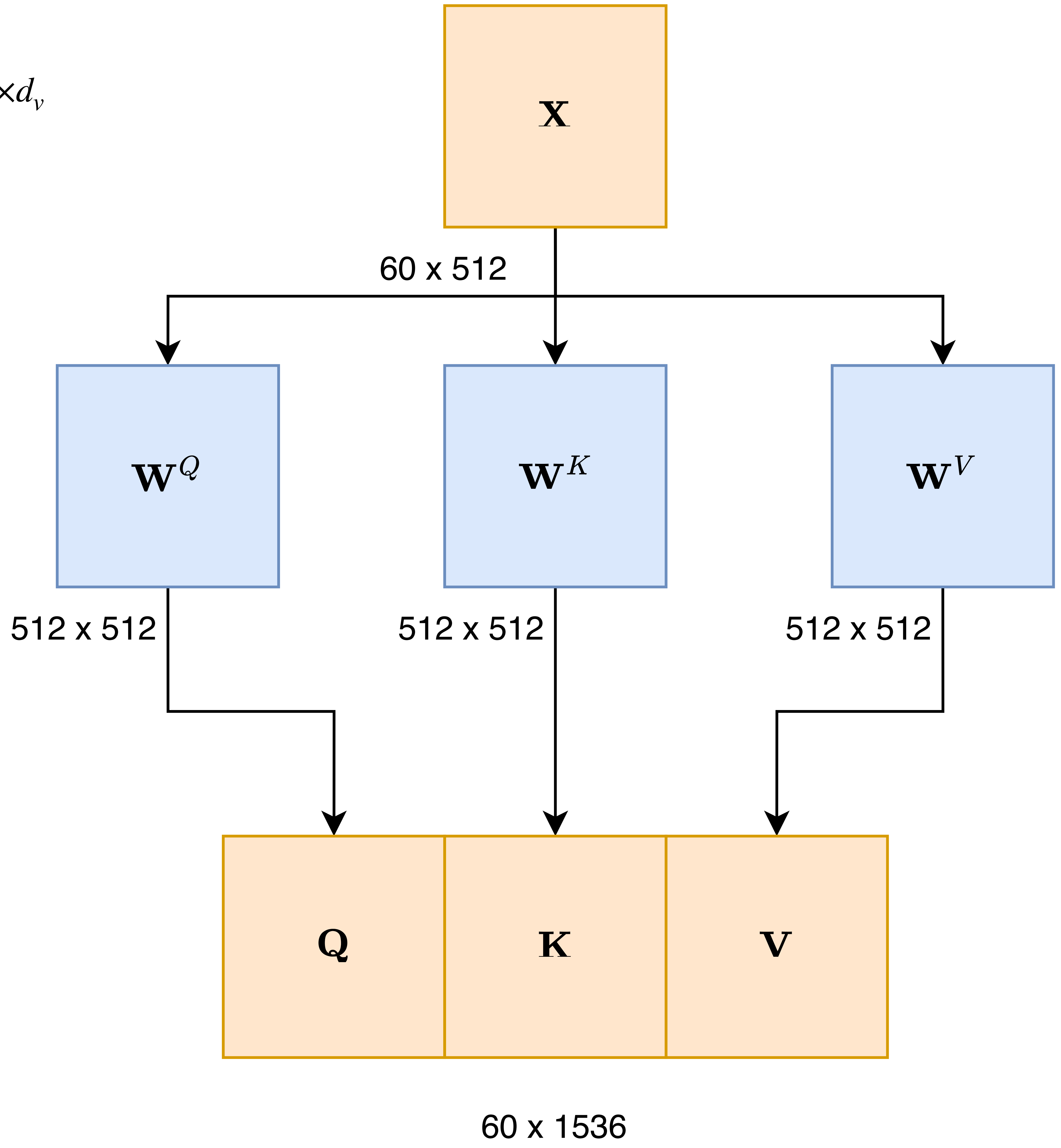
Learn attention

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{Q}, \quad \mathbf{K} = \mathbf{X}\mathbf{W}^{K}, \quad \mathbf{V} = \mathbf{X}\mathbf{W}^{V}; \quad \mathbf{Q}, \mathbf{K} \in \mathbb{R}^{N\times d_k}, \mathbf{V} \in \mathbb{R}^{N\times d_v}$$

$$\textsf{score}(\mathbf{Q}, \mathbf{K}) = \mathbf{Q}\mathbf{K}^{\top}$$

$$\textsf{Attention}(\mathbf{Q}, \mathbf{K}) = \textsf{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\top}}{\sqrt{d_k}}\right)$$

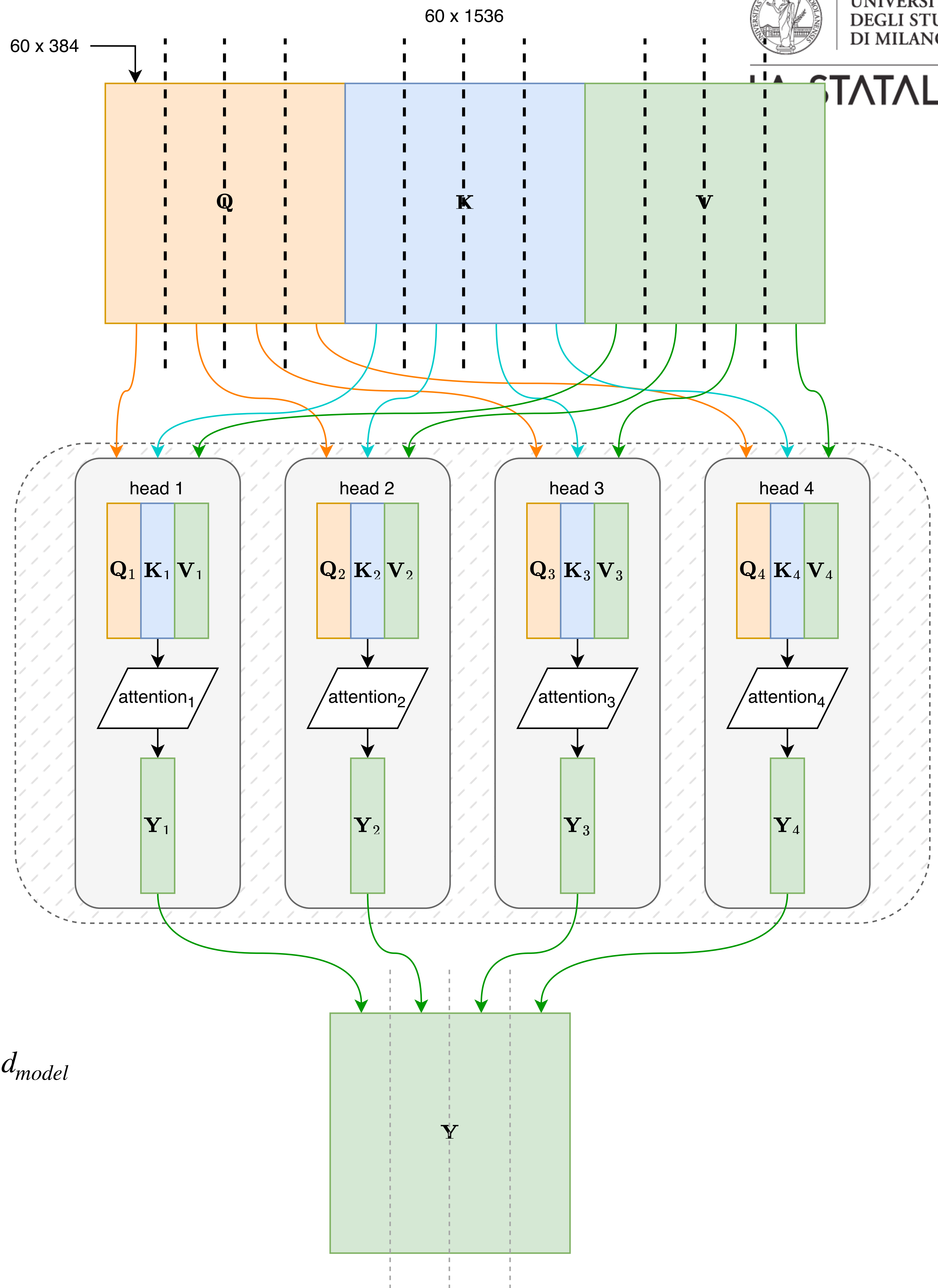
$$\textsf{Output} = \textsf{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\top}}{\sqrt{d_k}}\right)\mathbf{V}$$



Multi head attention

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.
They give enriched contextual representation: how do different words relate to each other simultaneously?
These are sets of self-attention layers, called **heads**, that reside in parallel layers at the same depth in a model, each with its own set of parameters. Given these distinct set of parameters, each head can learn different aspects of the relationships that exist among inputs at the same level of abstraction.

Multi-head Attention



$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q, \mathbf{K} = \mathbf{X}\mathbf{W}_i^K, \mathbf{V} = \mathbf{X}\mathbf{W}_i^V; \quad \mathbf{W}_i^Q \in \mathbb{R}^{d_{model}\times d_k}, \mathbf{W}_i^K \in \mathbb{R}^{d_{model}\times d_k}, \mathbf{W}_i^V \in \mathbb{R}^{d_{model}\times d_v}$$

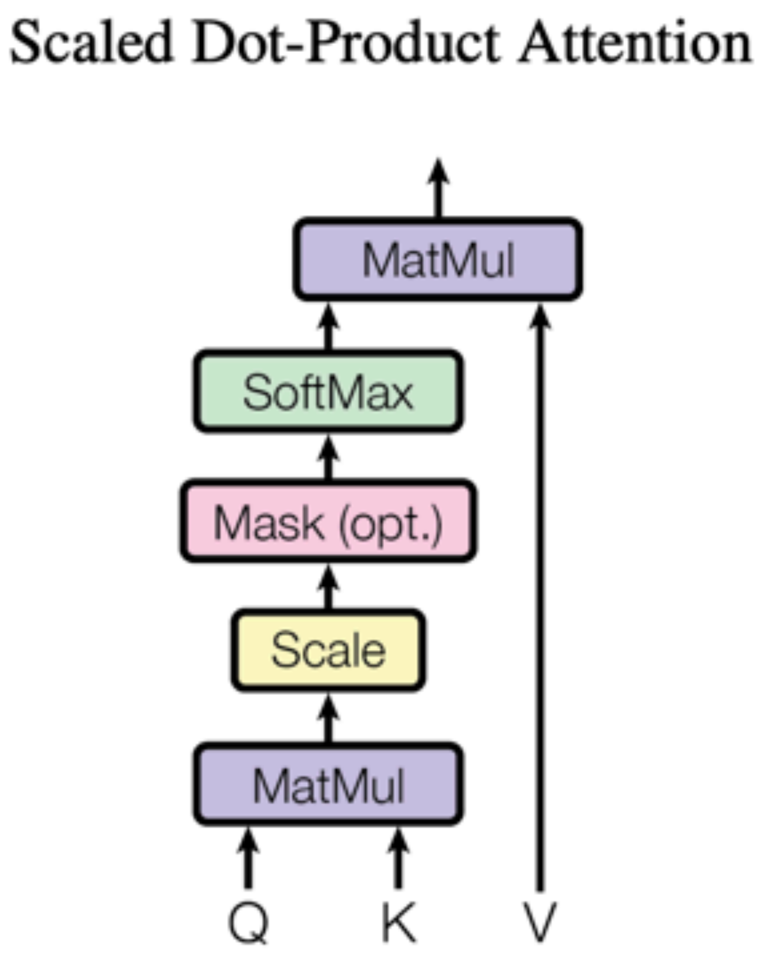
$$\mathbf{head}_i = \textbf{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$$

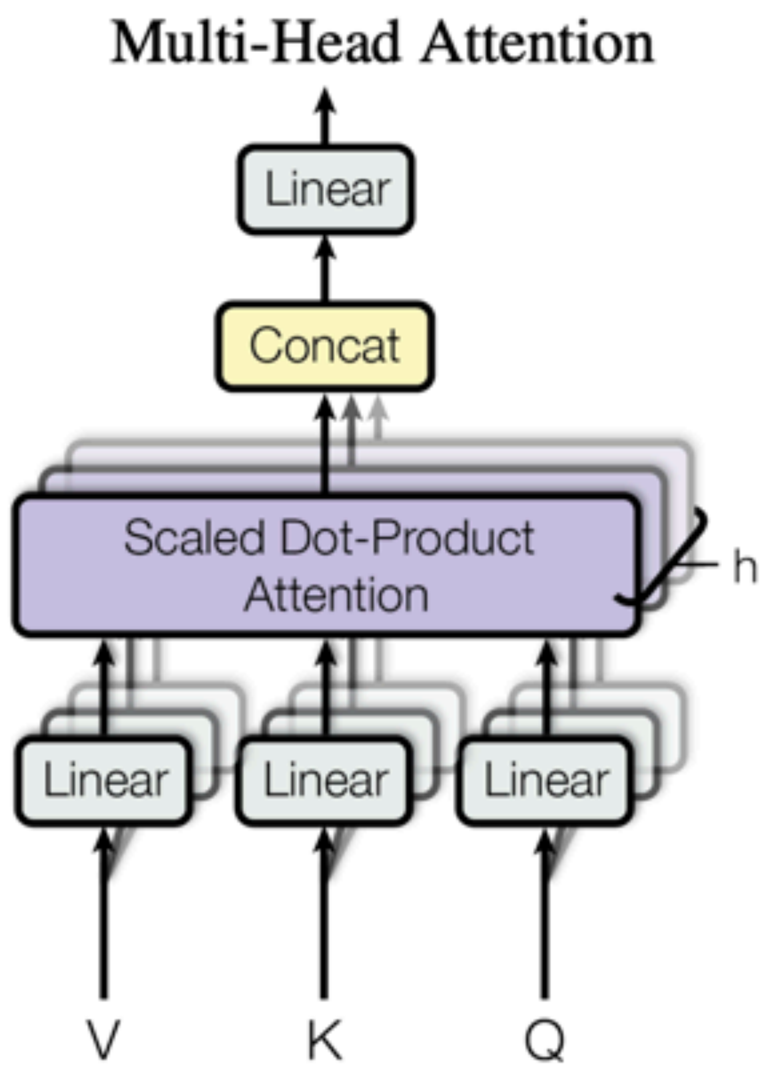
$$\textbf{MultiHeadAttn}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2, \ldots, \oplus\, \mathbf{head}_h)\mathbf{W}^O, \quad \mathbf{W}^O \in \mathbb{R}^{hd_v \times d_{model}}$$

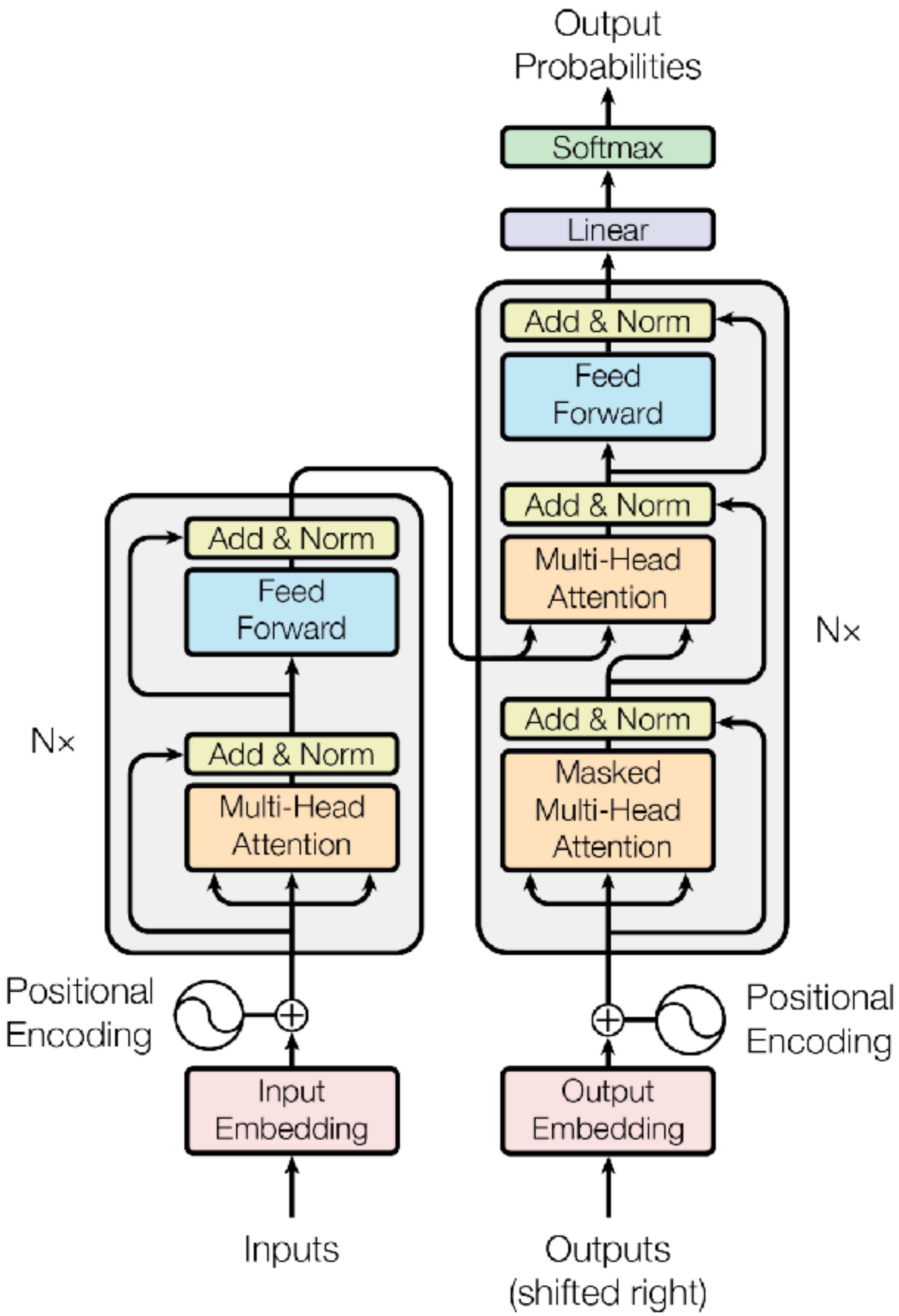
Transformer = Encoder decoder with attention + positional encoding

In summary we have:

1. Encoder-decoder architecture
2. Positional encoding
3. MultiHead (parallel) Attention







Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.