

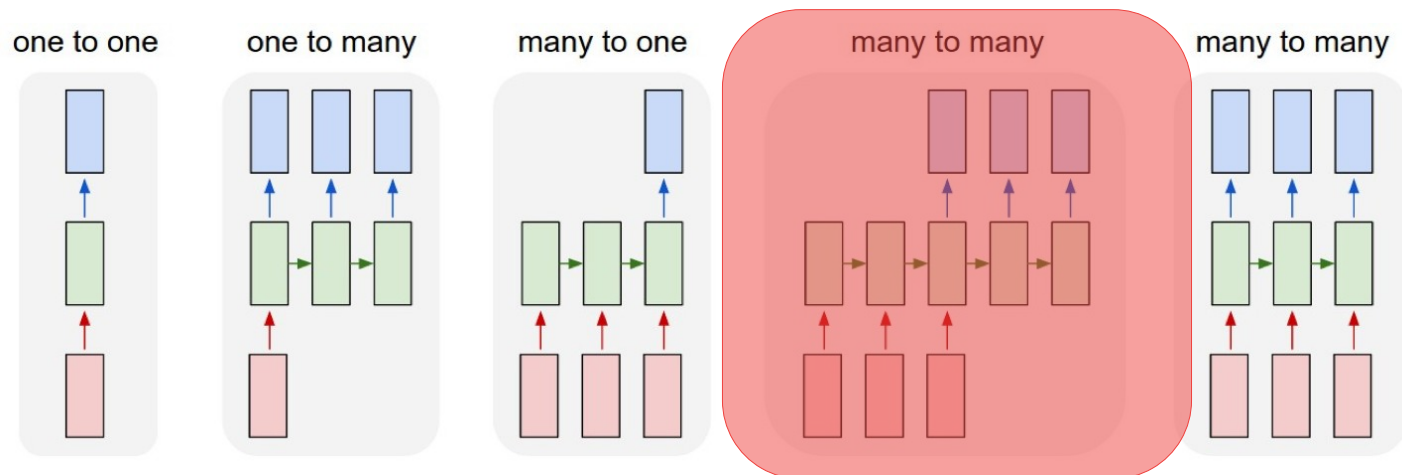
# **Natural Language and Speech Processing**

## Lecture 8: Neural Attention

Tanel Alumäe

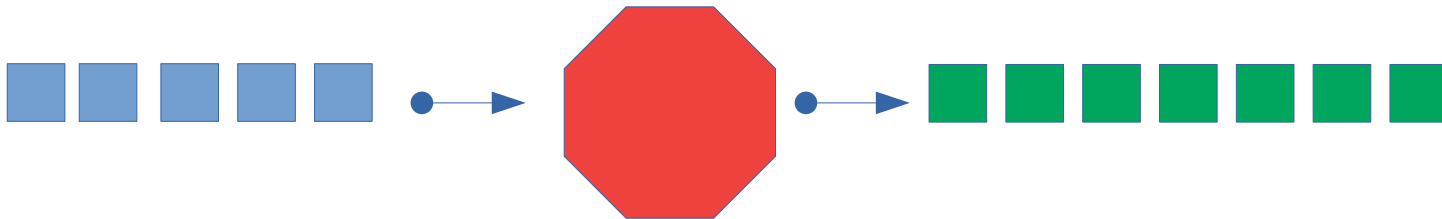
# Recurrent model types

- Using recurrent neural networks, we can:
  - Classify items in a sequence
    - e.g., classify words in a sentence
  - Classify sequences (using hidden state corresponding to the last item of the sequence)
  - Generate sequences from the model
- **But how to deal with a problem when both the input and output are sequences, but of different length?**



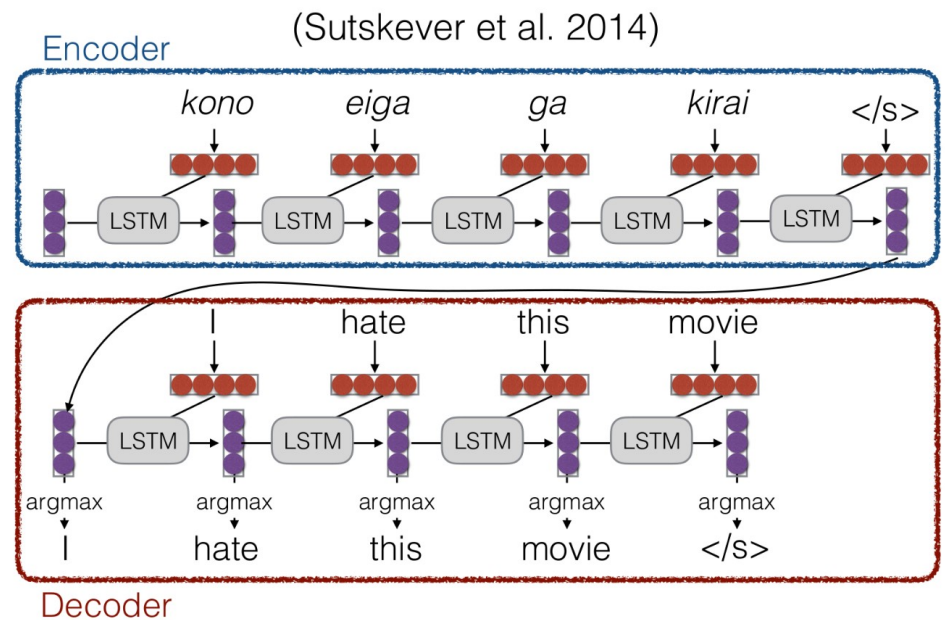
# Sequence-to-sequence

- Encoder-decoder models
- Also known as sequence-to-sequence (seq2seq models)
- Useful for
  - Machine translation
  - Summarization
  - Question answering
  - Chat bots



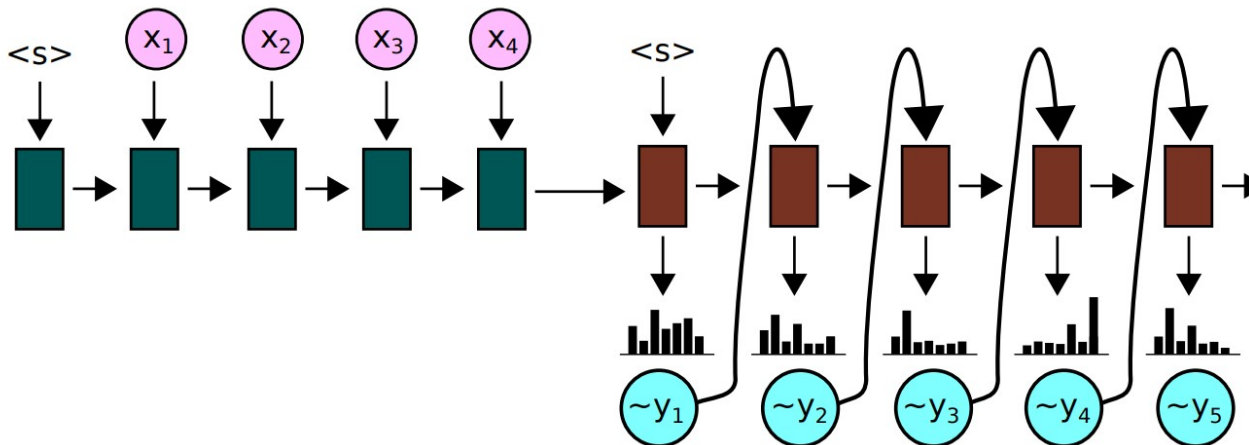
# Basic idea

- Basic idea on RNN-based seq2seq models:
  - Use one RNN (**encoder**) to process the input sequence
  - Take the hidden state after processing the last item in the input
  - Feed the hidden state to the second RNN (**decoder**) as the initial state
  - Generate until  $\langle /s \rangle$ !



# Sequence-to-sequence

- Advantage:
  - Can be trained end-to-end (i.e., there is just one model, and all you need is a lot of input-output pairs as training data)
- Does it work?
  - Kind of

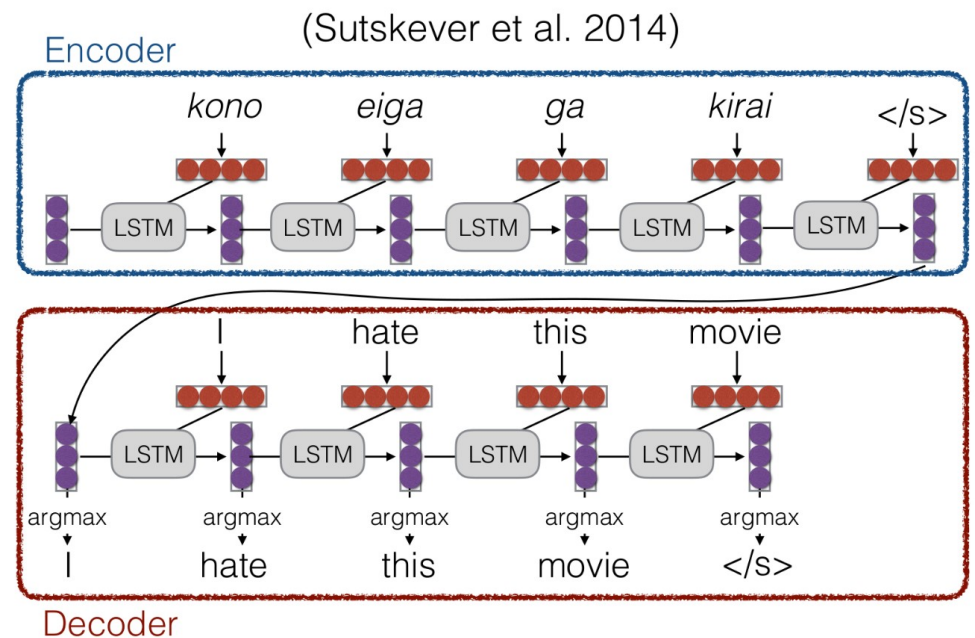


# Early neural machine translation

- Before neural MT, statistical MT was used
- Statistical MT retained most of the meaning, but the syntax of the generated sentence was often horrible
- Output of neural MT is very different from statistical MT
  - Always very fluent
  - But sometimes the meaning is completely lost!

# Problem with this architecture

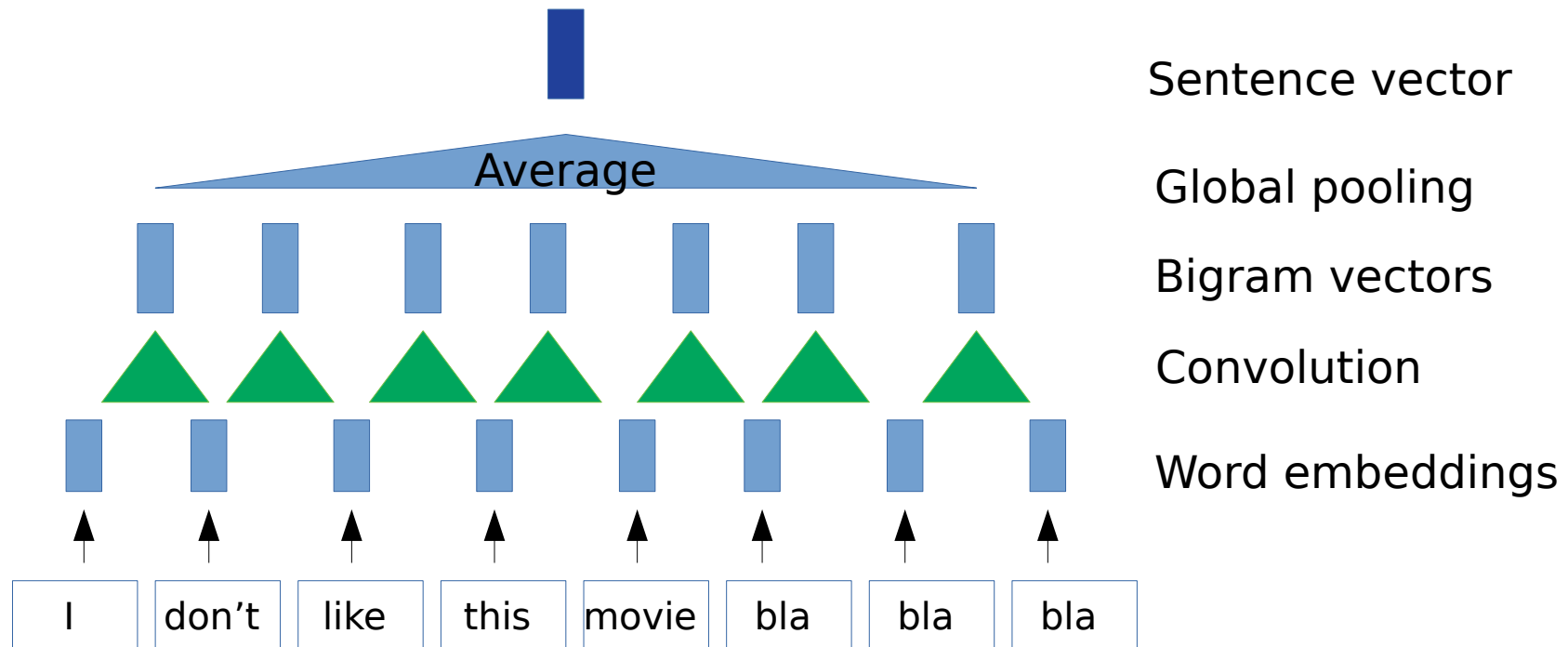
- The whole source sentence is compressed into one vector
- The vector has fixed dimensions, regardless of the sentence
- “You can’t cram the meaning of a whole *%&!\$ing* sentence into a single *\$&!\*ing* vector!” — *Ray Mooney*
- Fixed-length vector makes it difficult to translate longer sequences
- Often it has forgotten the earlier parts of the sequence once it has processed the entire the sequence.



# Neural Attention

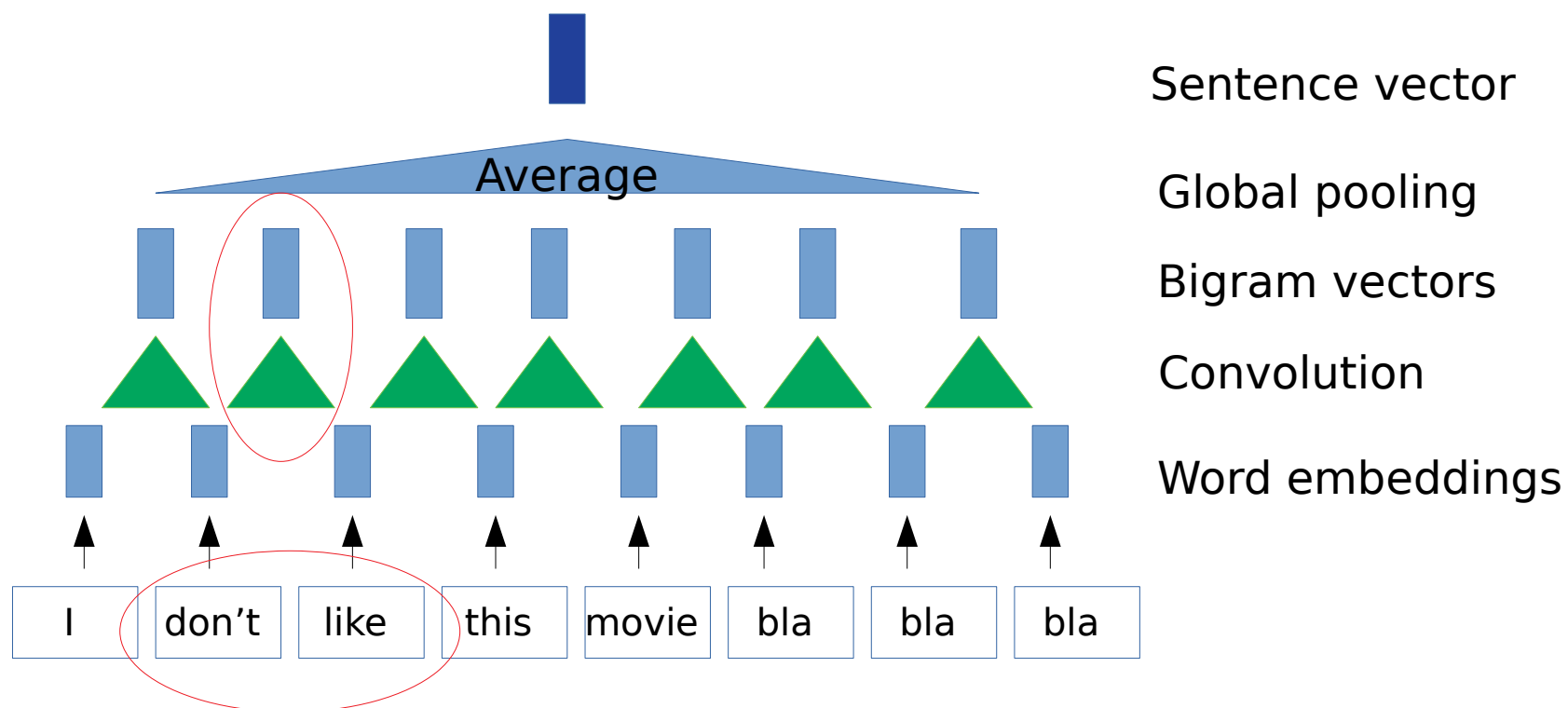
# Global pooling

- Remember convolutional neural networks?
- When applied for sequence classification (e.g. sentiment analysis), global (average or max) pooling is often applied to find the fixed-length vector of the whole text



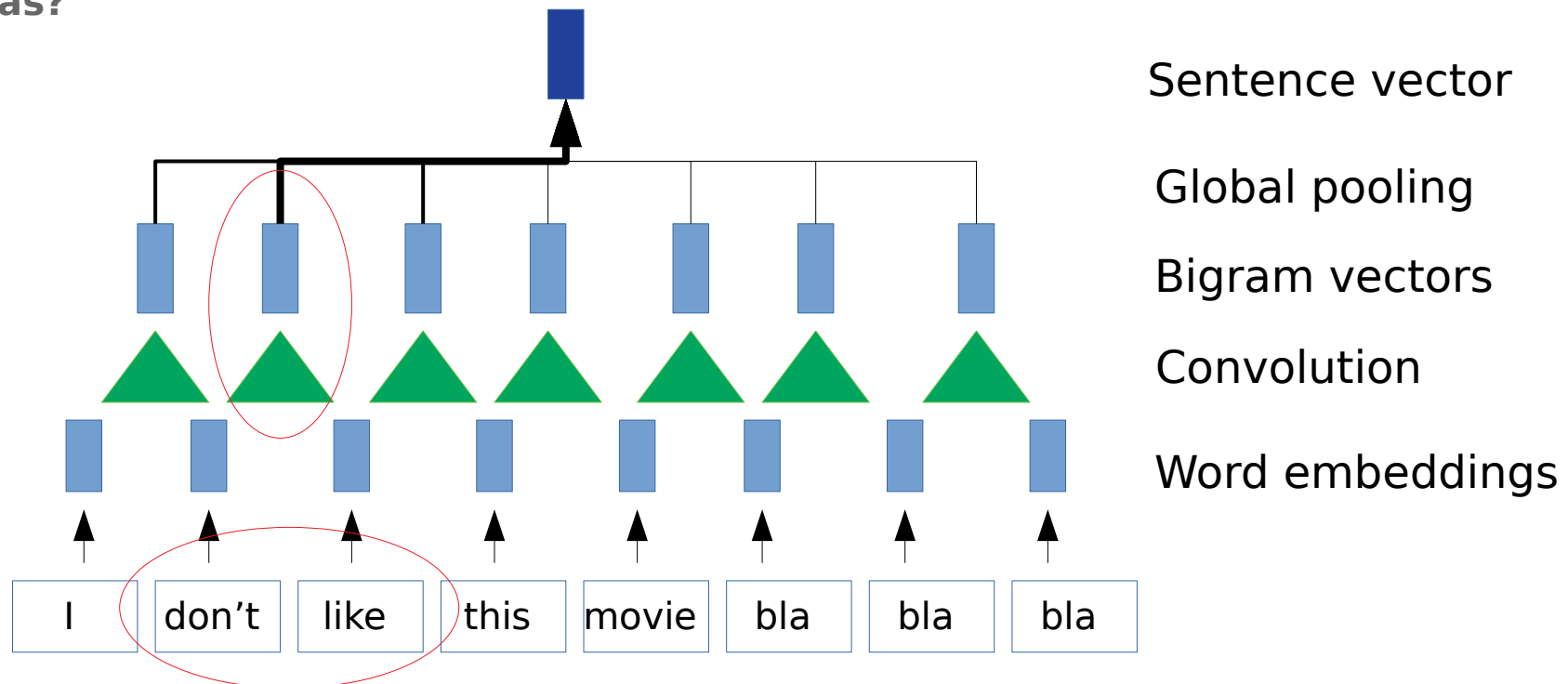
# Problem with global pooling

- Global pooling is too generic:
  - E.g., the bigram „don't like” is really useful for sentiment analysis, but we „bla bla bla” can be discarded
- In other words, we need something like a **weighted average**



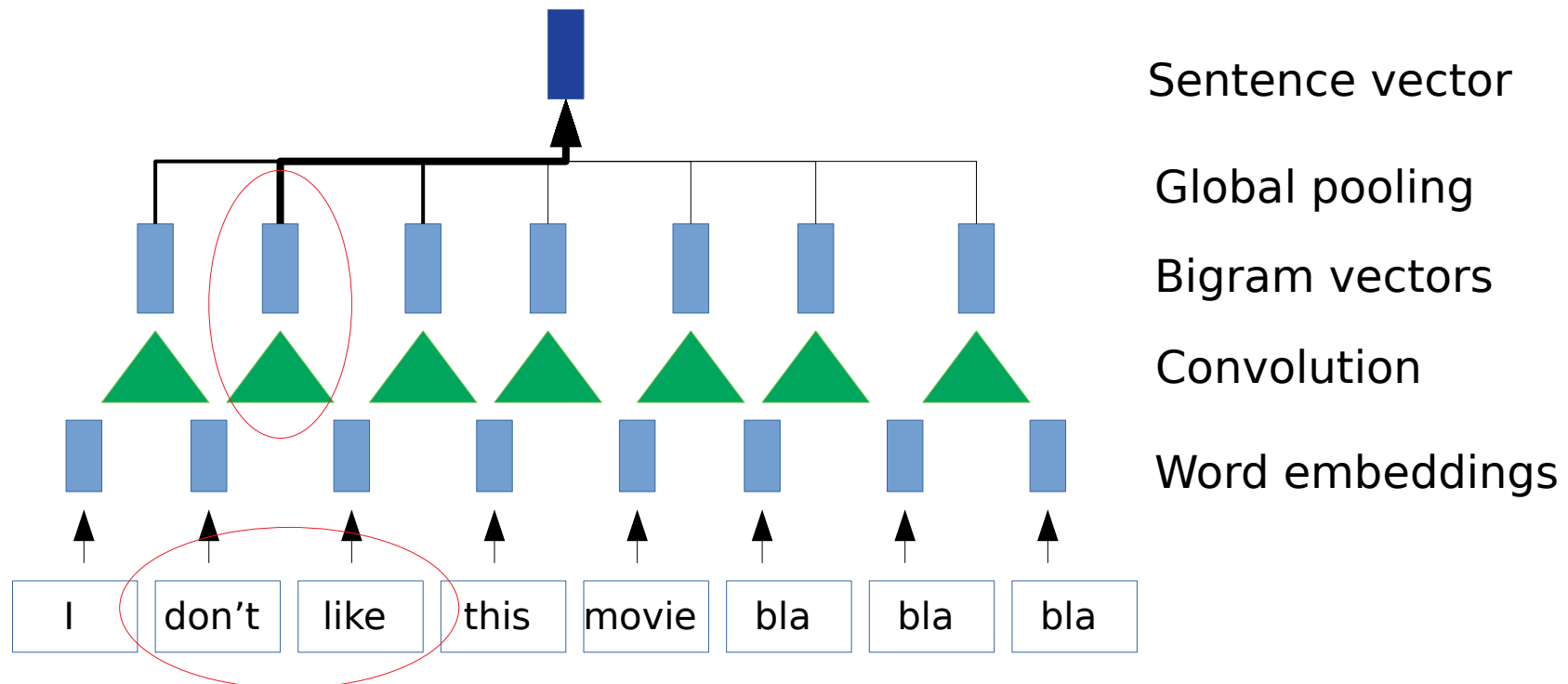
# Weighted average?

- Weighted average would allow us to combine the sentence vector out of only the most relevant parts of the sentence
- **But how to get the weights for the weighted average?**
- The weights must depend on the input sentence
  - i.e., we cannot expect that bigrams 2 and 3 are always most important
- **Any ideas?**

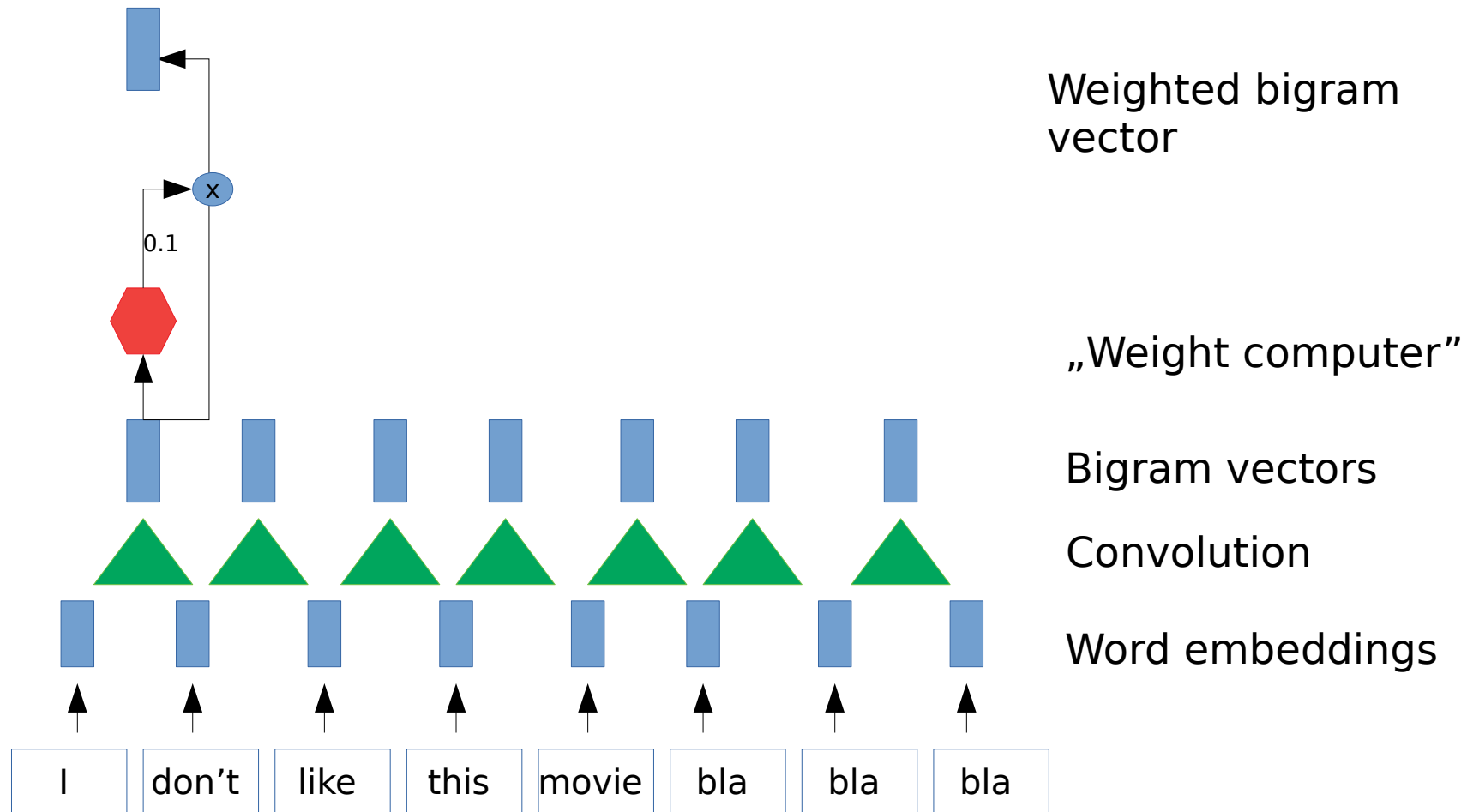


# Weighted average

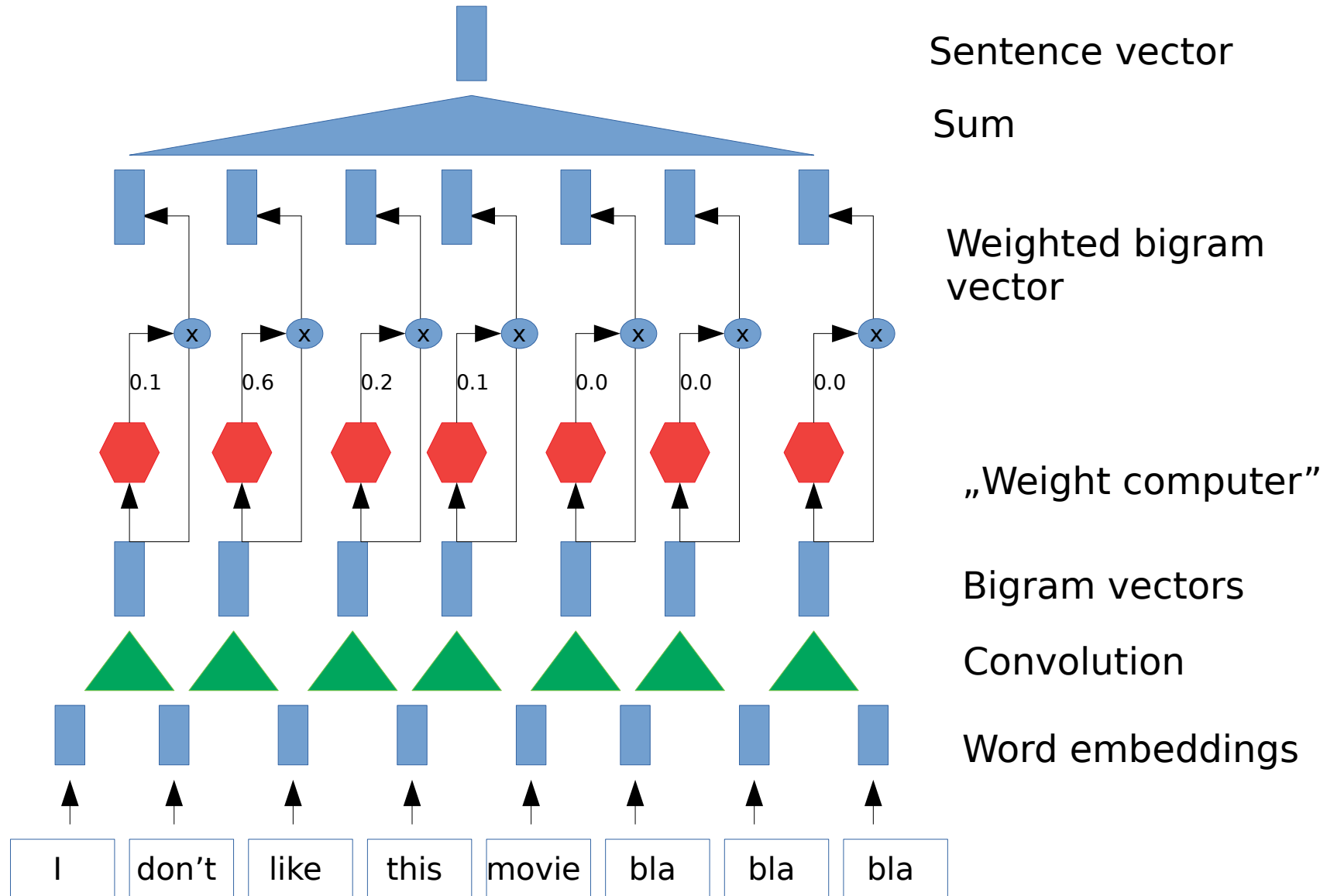
- How to get the weights for the weighted average?
- Solution: use a small neural network!
- We feed each **bigram vector** into the small neural network, and it will tell us the weight



# Computing the weights

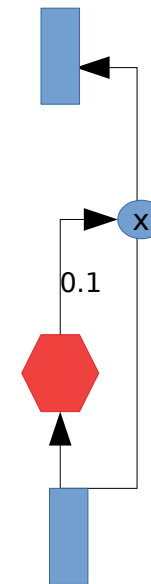


# Computing the weights



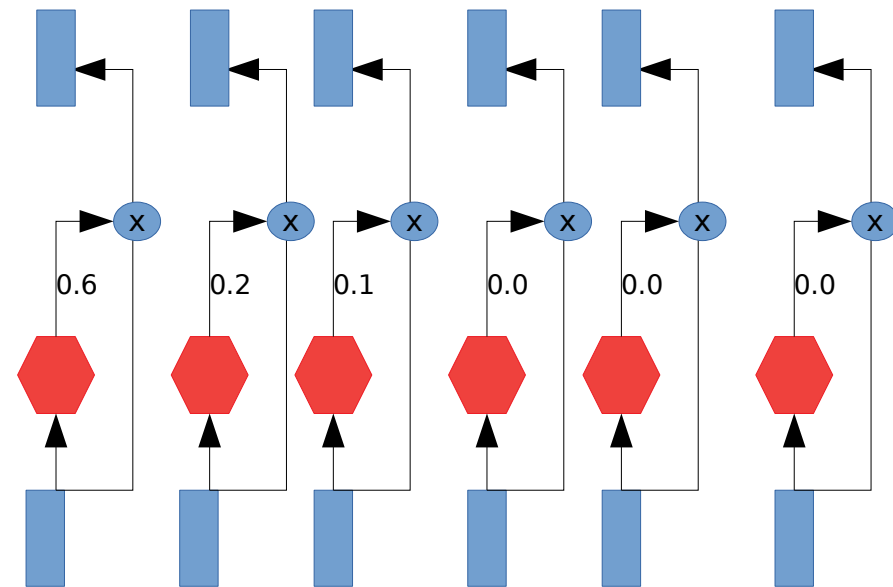
# Weight computer

- What kind of neural network the weight computer (**the red hexagon**) is?
  - Usually just one hidden layer (+ nonlinearity like ReLU) is enough
- But „who” gives this neural network to us?
  - We simply train it, like other parts of the neural network
- Intuition: in the context of sentiment analysis, the weight computer will learn to recognize „emotional” phrases, like „don't like” „loved the plot”, „hated the music”, etc, and assign higher weights to them



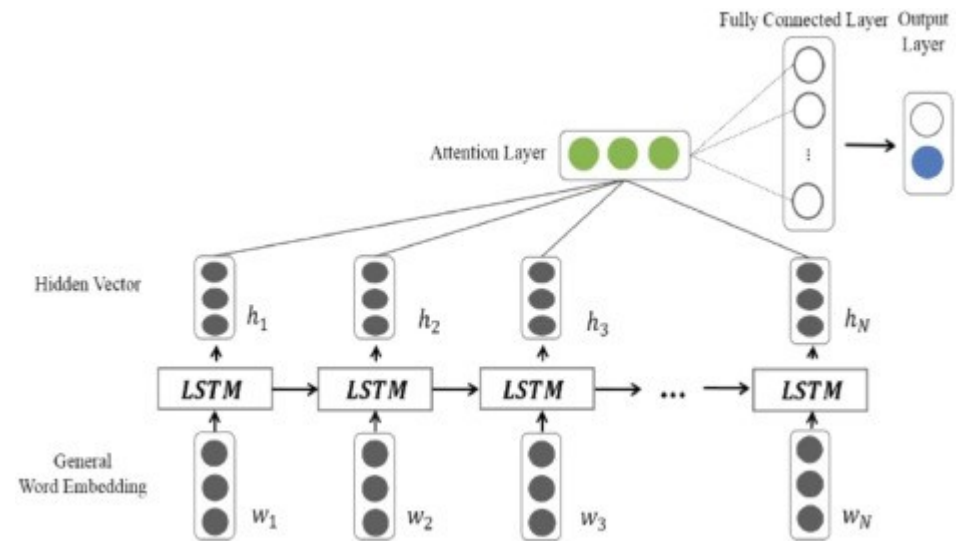
# Weighted average: sum to 1

- Of course, when we compute the weighted average, the weights should sum to 1
- This can be achieved using *softmax*
  - Output of the *weight computer NN* is normalized using softmax over the whole sentence/document
  - This guarantees that the weights/scores are all positive and sum to one



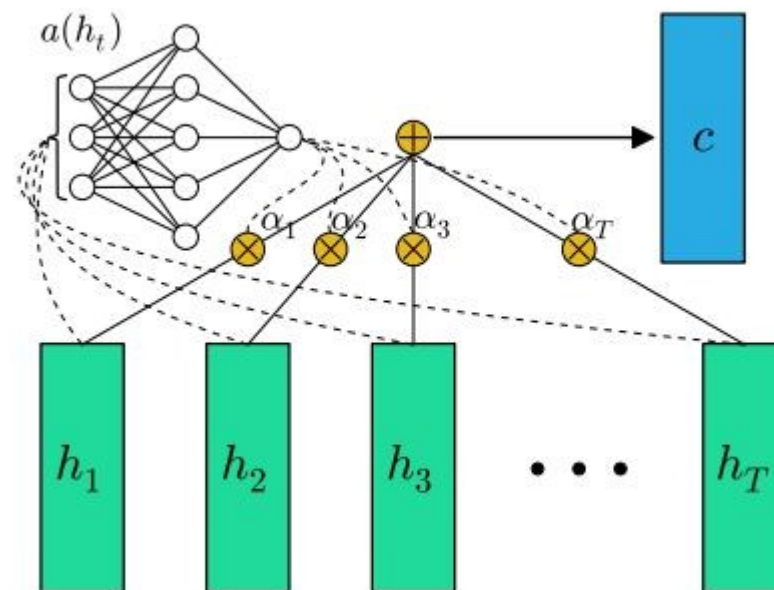
# LSTMs and Attention

- In our example, we used convolutional neural network for getting intermediate representations (bigram vectors)
- We can also use RNN (LSTM) to give us the intermediate vectors, and apply attention to them



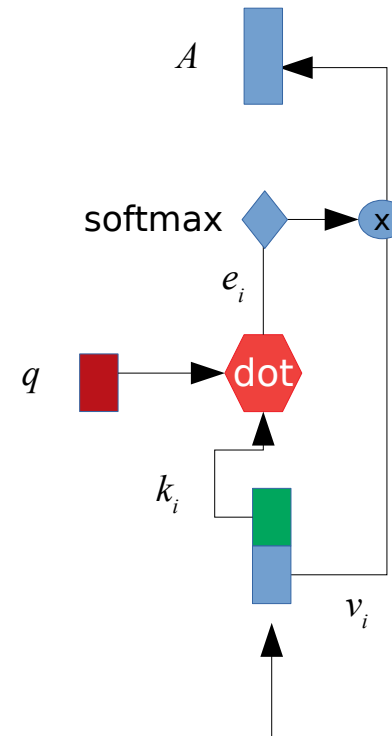
# Attention mechanism

- This kind of weighted average using learned weight computer is called **attention mechanism**
- Attention mechanism is a bit similar to RNN: it also allows compressing a sequence of varying length into a fixed-length vector
  - But it is also quite different



# Attention with query, key and value

- Often attention mechanism is implemented using **query, key and value**:
  - The underlying DNN (CNN or RNN) generates a representation (vector) that is viewed as consisting of two parts: **key** and **value**
  - The weight finder is not a small DNN but a **vector („query”)**, with **same dim as key**
  - Score ( $e_i$ ) is simply computed as a **dot product (similarity) between key and query**
  - Weights over a sequence are softmaxed
  - The softmaxed weight is multiplied with the **value**

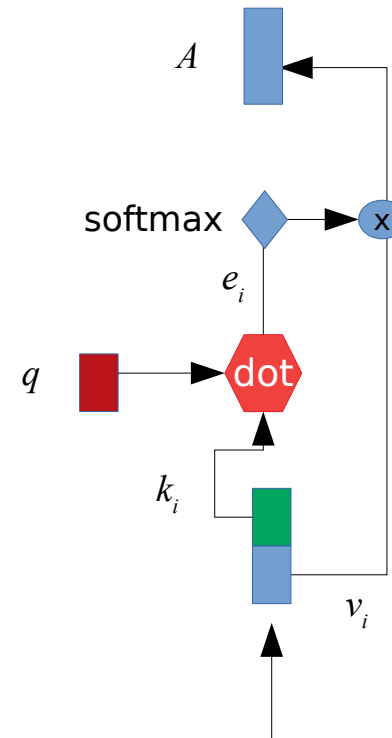


$$e_i = q^T k_i$$

$$A(q, K, V) = \sum_i \frac{\exp(e_i)}{\sum_j \exp(e_j)} v_i$$

# Attention with query, key and value, cont.

- In practice, simple dot product is not the best metric for calculating the score of key, given the query
- Instead, use an additional weight matrix  $W_s$  between query and key
- This allows the model to learn which aspects of similarity between the key and query states are important

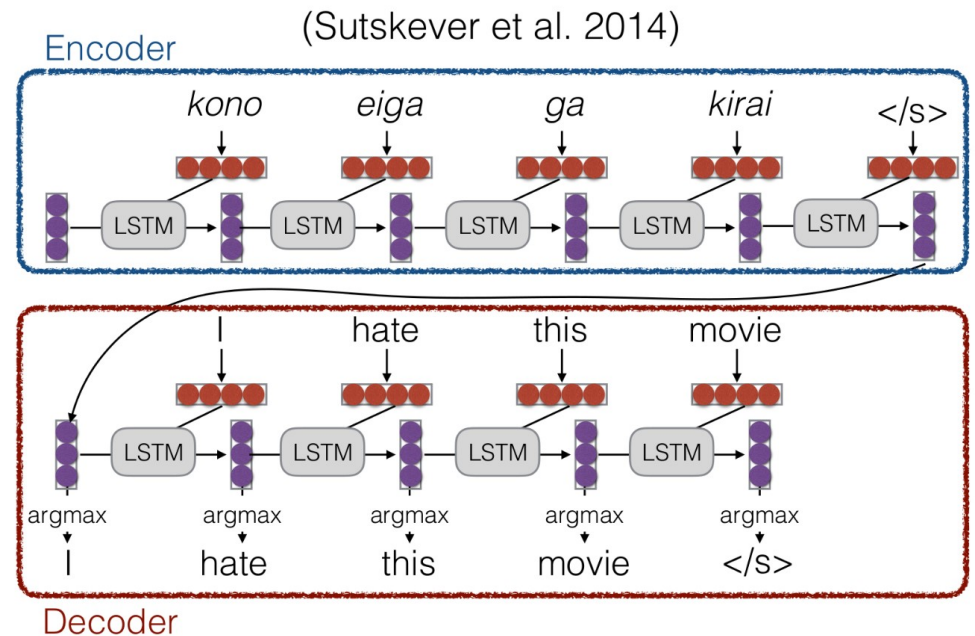


$$e_i = q W_s k_i$$

$$A(q, K, V) = \sum_i \frac{\exp(e_i)}{\sum_j \exp(e_j)} v_i$$

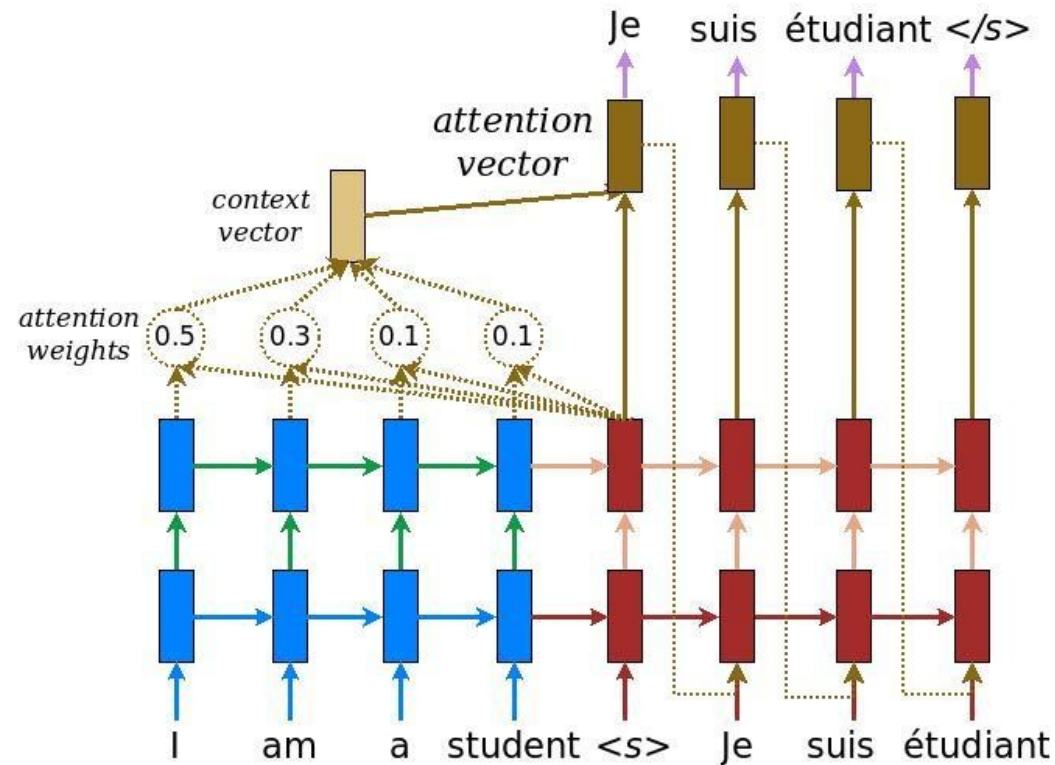
# Attention in seq2seq

- But what does attention have to do with the information bottleneck problem in seq2seq models?
- It turns out that attention can be effectively used to give the decoder a view over the whole input sequence at each decoding step
- Then, we don't have to encode the full source sentence into the hidden state that is passed from encoder to decoder



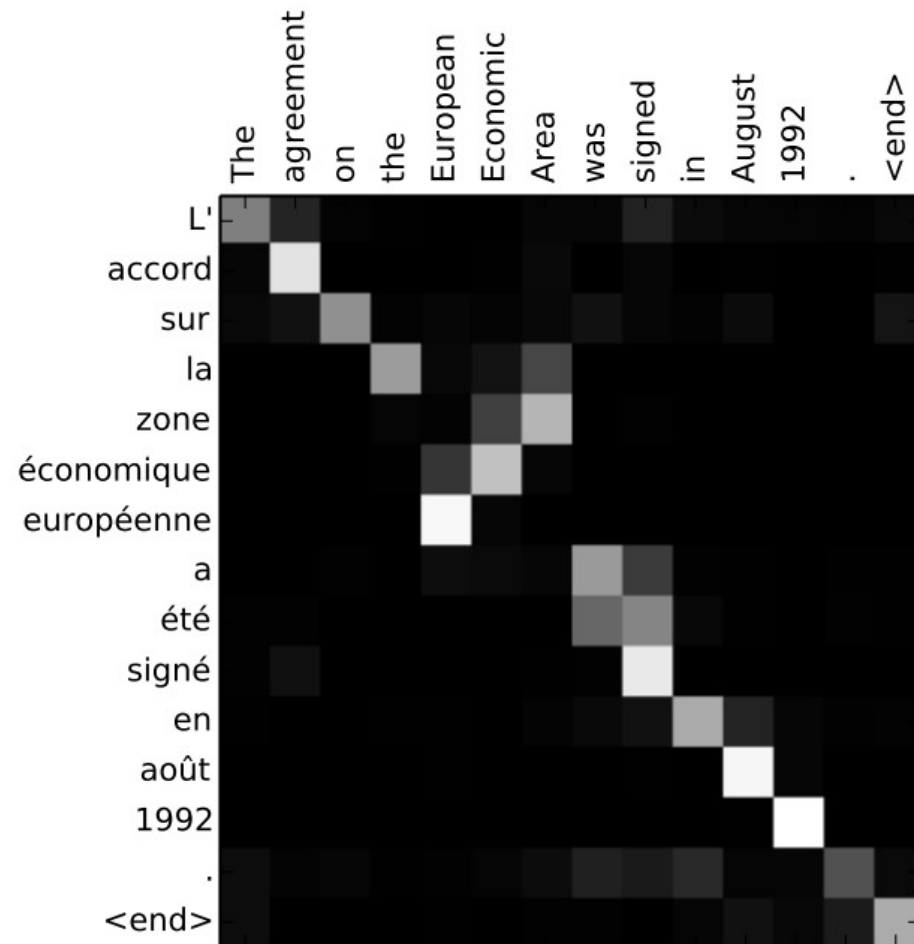
# Attention in seq2seq

- Pass the source sentence through a RNN
- When generating a target sentence, use the **current hidden state of the decoder as the query vector** in attention
- Use the query vector to compute a new context vector from source sentence
- Use the result („attention vector”) as additional input when generating the next word



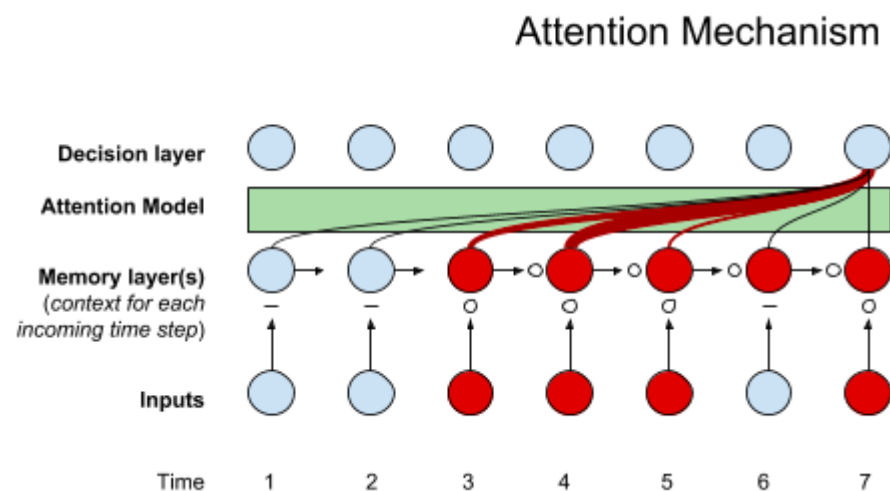
# Visualizing attention

- We can check which words in the source sentence had highest weights for each target word
- In other words, which words were **attended to**
- You can see how the model paid attention correctly when outputting "European Economic Area"
- In French, the order of these words is reversed ("européenne économique zone") as compared to English



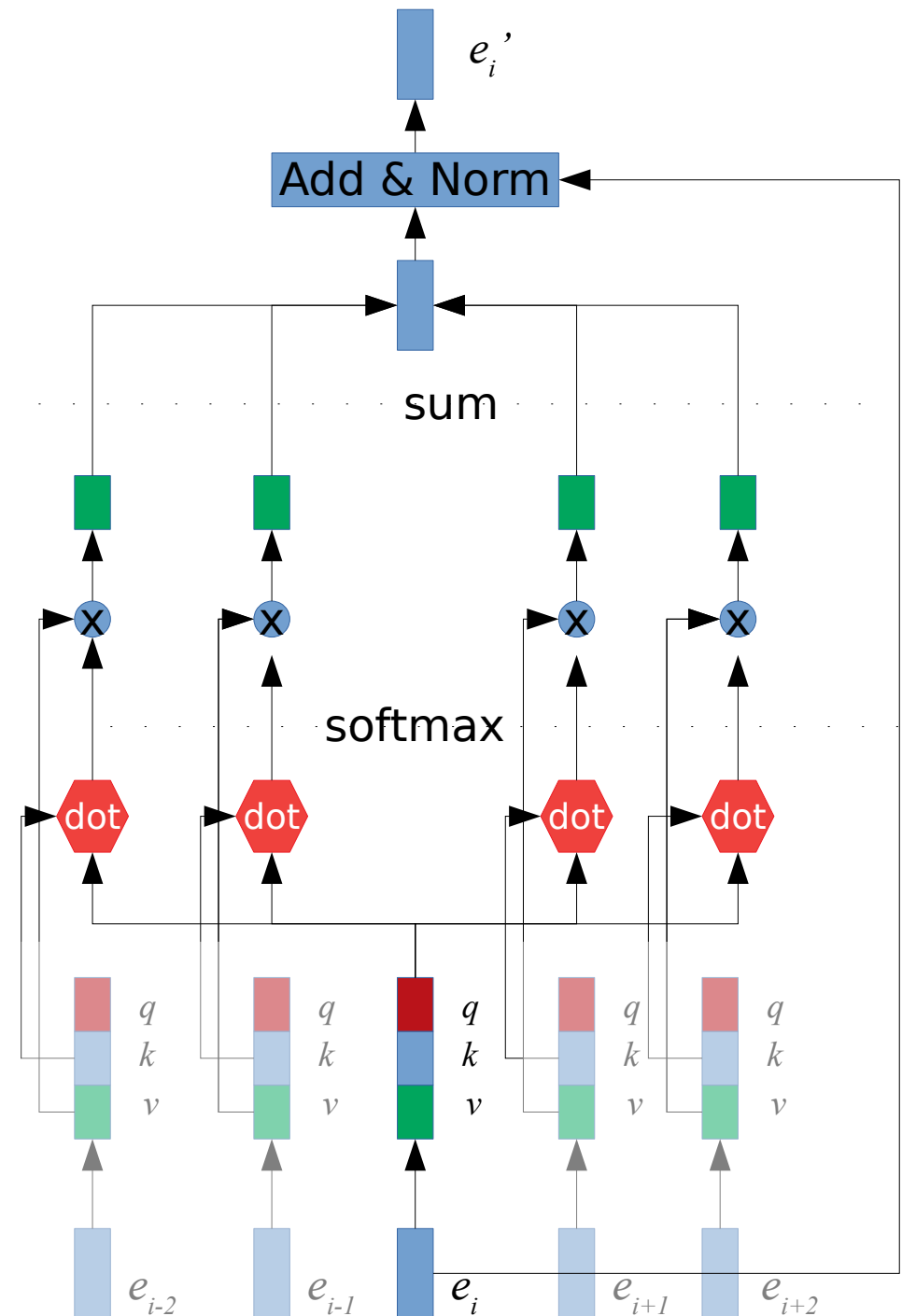
# Self-attention

- So far, we looked at how to use attention for classifying a sequence
  - Attention acts as a weighted average over the sequence
- But attention can be also used for sequence tagging or next-word prediction
  - In this case, attention over the whole sequence is computed for each word independently, using the current word itself as a query vector
- This is called **self-attention**



# Self-attention explained

- Self-attention transforms word embeddings  $e$  to new word embeddings  $e'$ 
  - Each  $e$  is transformed to  $q$ ,  $k$  and  $v$ , using matrix multiplication (using shared matrices for each position)
  - Vector  $q$  of position  $i$  is compared to vectors  $k$  of all other words, using dot product
  - Result is softmaxed over the whole sequence
  - Softmax values are used to compute weighted  $v$  vectors
  - Result is summed
  - Initial  $e$  is added to the new weighted sum, and the result is normalized, producing new  $e'$



# Positional encodings

- Attention loses all information about word order
  - When computing the attention weights (and the resulting weighted average), word order is not taken into account in any way!
- Positional encoding is a way to deal with sequential data when no recurrency or temporal information is available due to the lack of RNNs
- Solution: for each input word, the input embedding is a sum of two vectors:
  - The word embedding
  - A vector describing the word's position



# Positional encodings, cont.

- Usually, sinusoidal embeddings are used
  - Positional embeddings have the same dimensionality as word embeddings (e.g. 200)
  - Each dimension is a sine/cosine of the current timestep, using different frequency
- This allows to encode both absolute and relative position of the word

$$\vec{p}_t^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k \cdot t), & \text{if } i = 2k \\ \cos(\omega_k \cdot t), & \text{if } i = 2k + 1 \end{cases}$$

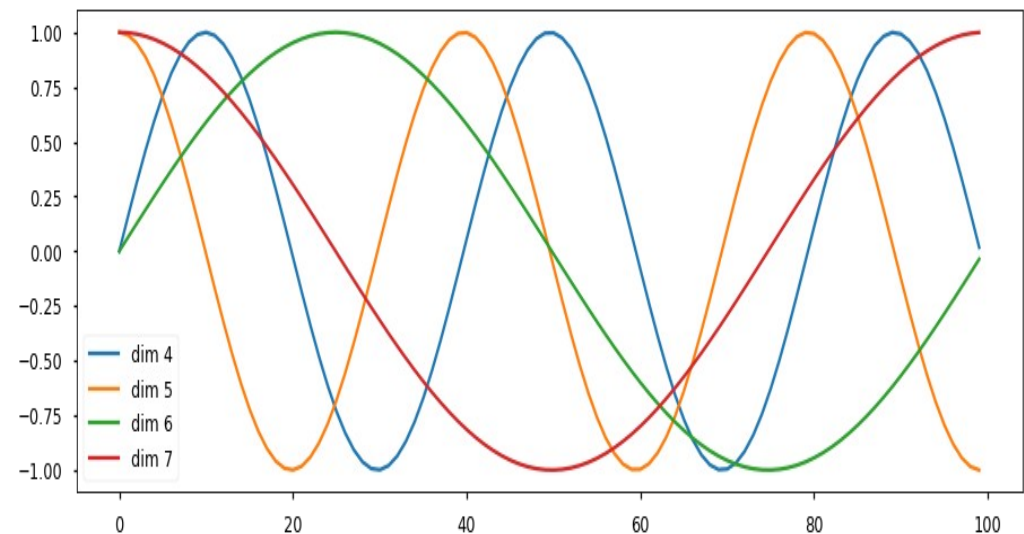
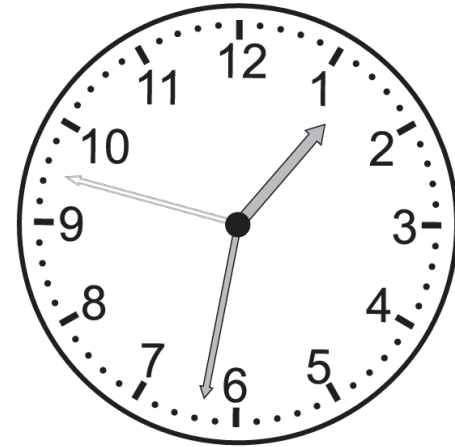
where

$$\omega_k = \frac{1}{10000^{2k/d}}$$

$$\vec{p}_t = \begin{bmatrix} \sin(\omega_1 \cdot t) \\ \cos(\omega_1 \cdot t) \\ \\ \sin(\omega_2 \cdot t) \\ \cos(\omega_2 \cdot t) \\ \\ \vdots \\ \\ \sin(\omega_{d/2} \cdot t) \\ \cos(\omega_{d/2} \cdot t) \end{bmatrix}_{d \times 1}$$

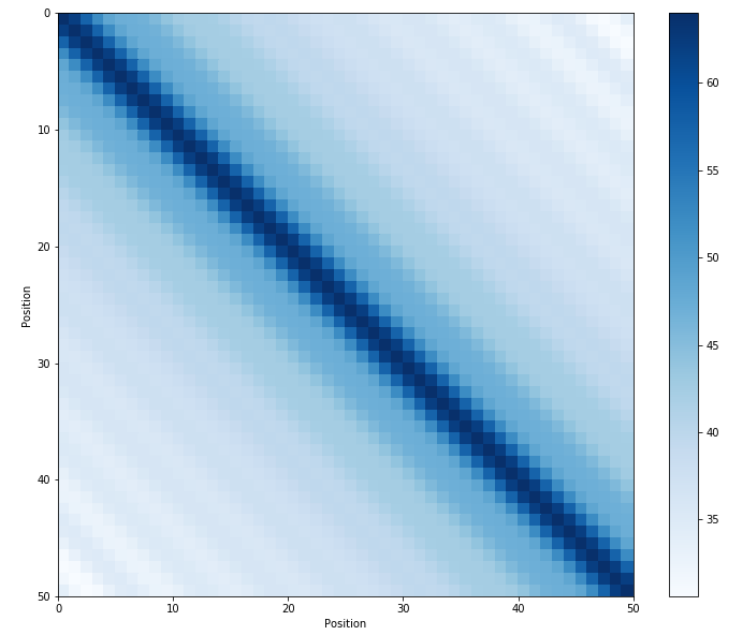
# Positional encodings, cont.

- Intuitive explanation of positional embedding is to think about it as a clock
  - Clock is a positional encoding with **three** dimensions
    - Hours hand, minutes hand, seconds hand
  - Moving from one position to the next position is just rotating those hands at different frequencies
    - (Thanks to Jiaxuan Wang for this great explanation)



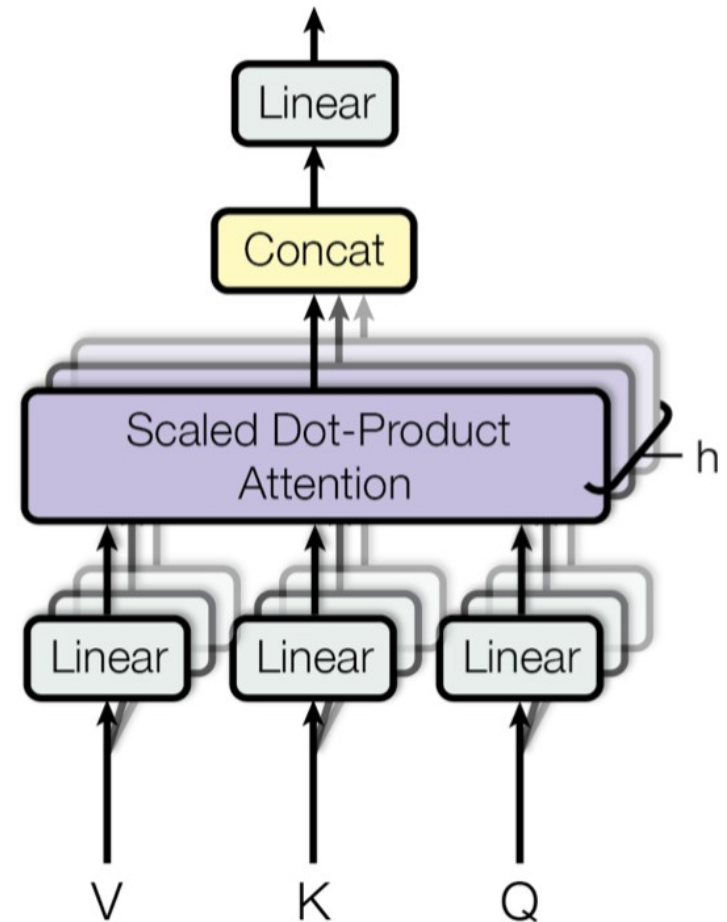
# Positional encodings, more

- Another property of sinusoidal positional encoding is that the **distance** between neighboring time-steps are **symmetrical** and decays nicely with time
- That is, words that are close to each other in the text have embeddings that are closer to each other than words far apart!



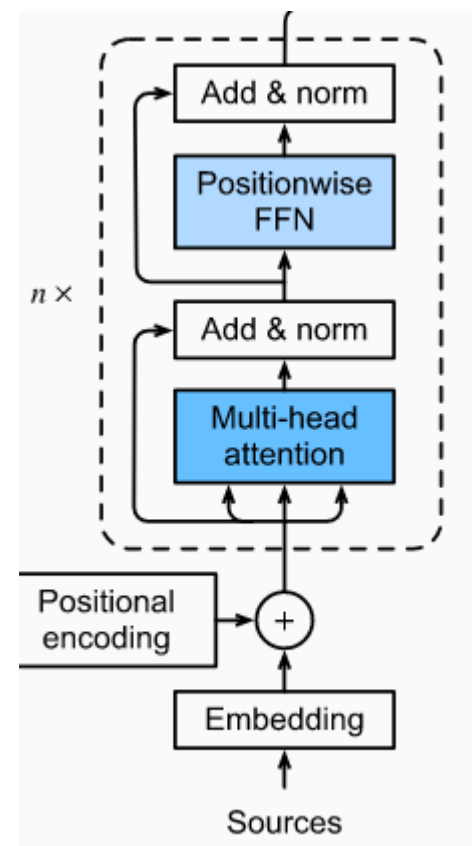
# Multi-head attention

- Attention mechanism is used as a way for the model to focus on relevant information based on what it is currently processing
- It is difficult to capture various different aspects of the input, using a single attention weighted sum
- To solve this problem the Transformer uses the **Multi-Head Attention** block
- This block computes multiple attention weighted sums instead of a single attention pass over the values



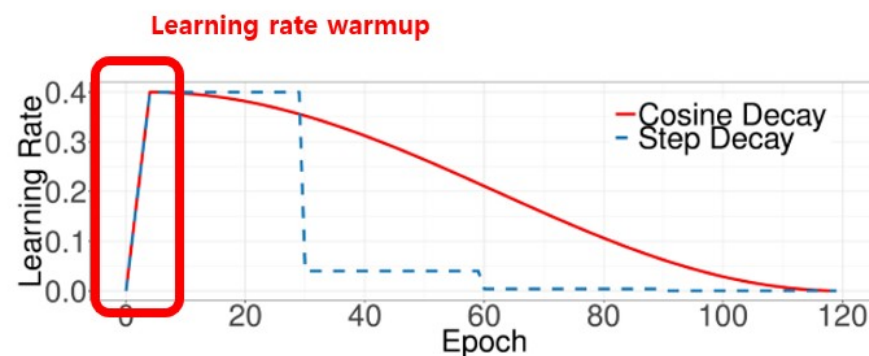
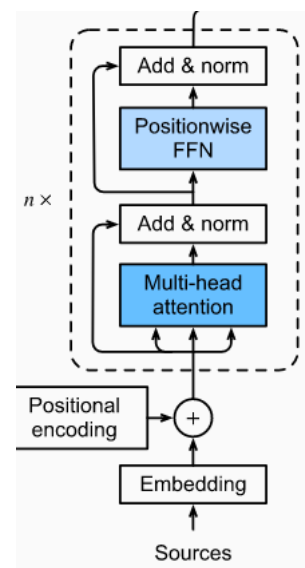
# Sequence processing using attention

- Modern NLP sequence processing:
  - Positional encoding
  - Several layers of:
    - Multi-head self-attention
    - Add & norm
    - Positionwise feed-forward layer (same as convolution with kernel size 1)
    - Add & norm



# Why better/worse than RNN?

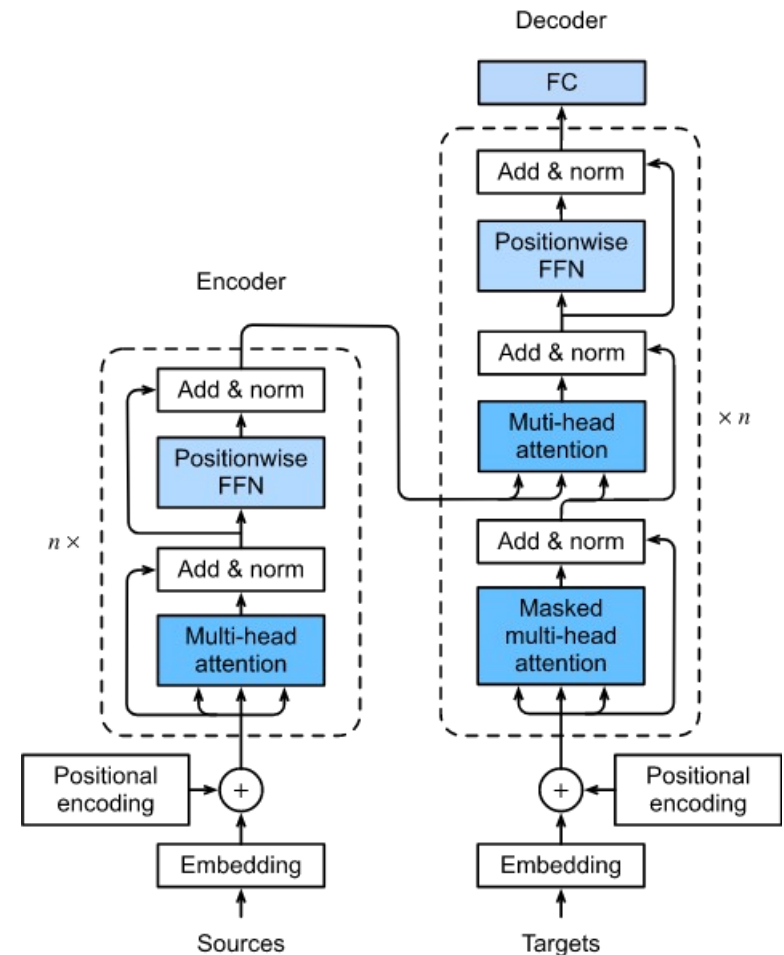
- RNN (incl LSTMs):
  - Long distance dependencies still difficult, despite LSTMs
- Attention:
  - Each step has **direct** access to all the other steps (self-attention)
- However: Attention based models
  - Require more data
  - Training more difficult (often require learning-rate warmup)



(a) Learning Rate Schedule

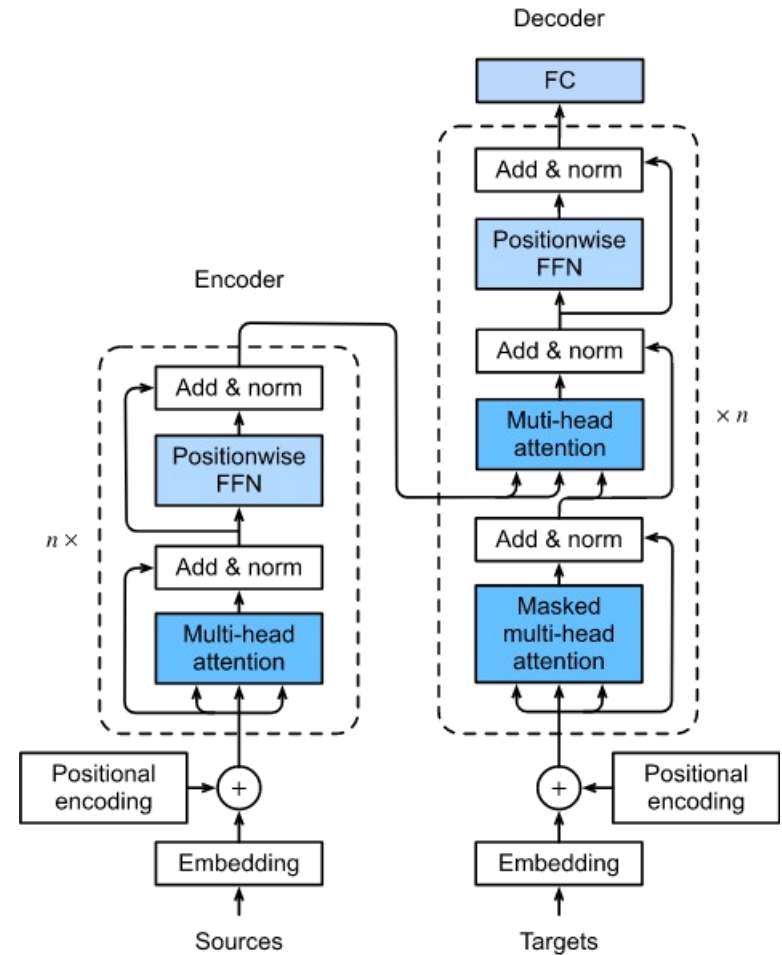
# The Transformer

- Transformer is a sequence-to-sequence model where both encoder and decoder only use attention blocks (i.e., **no RNNs**)
- Proposed in the paper „**Attention is all you need**“ (Vaswani et al., 2017)
- Probably the most important architecture currently in NLP
- Used in machine translation, summarization, speech recognition (with some modifications)



# The Transformer

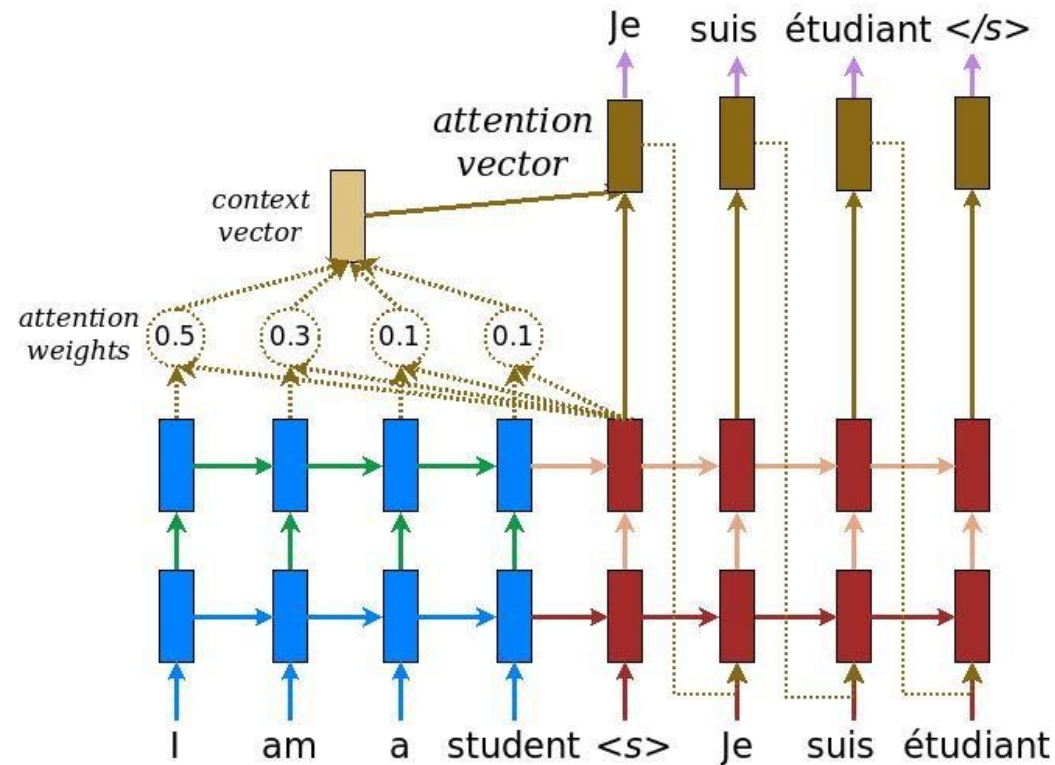
- Encoder is a sequence processing model that applies a series of multi-head attention layers (+ add & norm) to position-encoded input embeddings
- Decoder is a language model that uses attention over both inputs (that is, encoder outputs) as well as already generated tokens (self-attention)
  - Decoder is exactly as encoder, but with a intermediate block that attend over the encoder outputs



# Beam Search

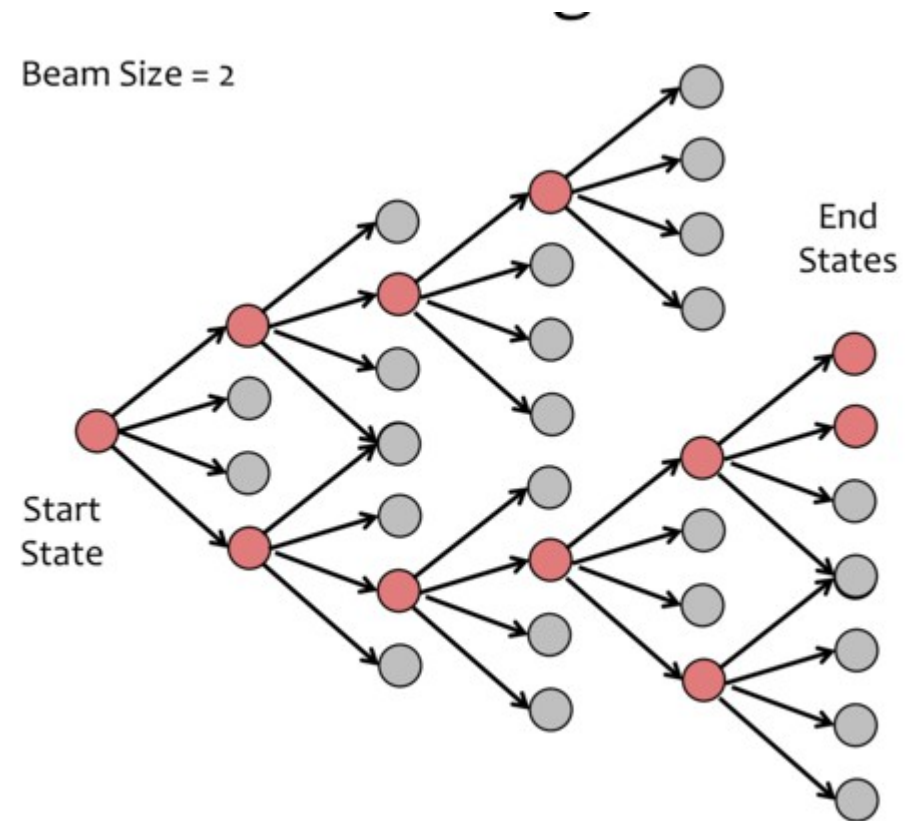
# Greedy Search

- How to generate the best output sequence from the decoder?
- Greedy search: at each time step, select the word with the highest probability
- It doesn't guarantee that the individual choices being made make sense together and combine into a coherent whole
- Can we sample from the output probability distribution instead?
  - In the case of MT, this leads to strange and usually incorrect output
- We also cannot track all alternatives, since this leads to  $|V|^k$  alternatives, where  $|V|$  is the vocabulary size and  $k$  length of the sentence



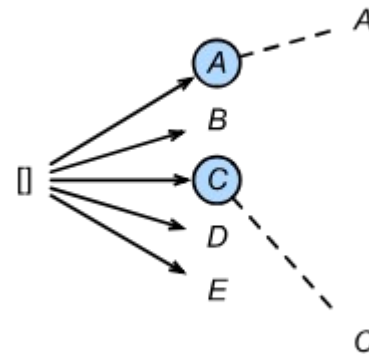
# Beam Search

- Beam search tracks  $k$  (beam size) best alternatives in parallel
- At first time step,  $k$  best alternatives are kept, together with their probabilities
- This is called the *frontier*
- At each next timestep, we expand each candidate in the frontier, and multiply the probabilities along each path
- After each expansion step, we *prune* the candidates to  $k$  best alternatives



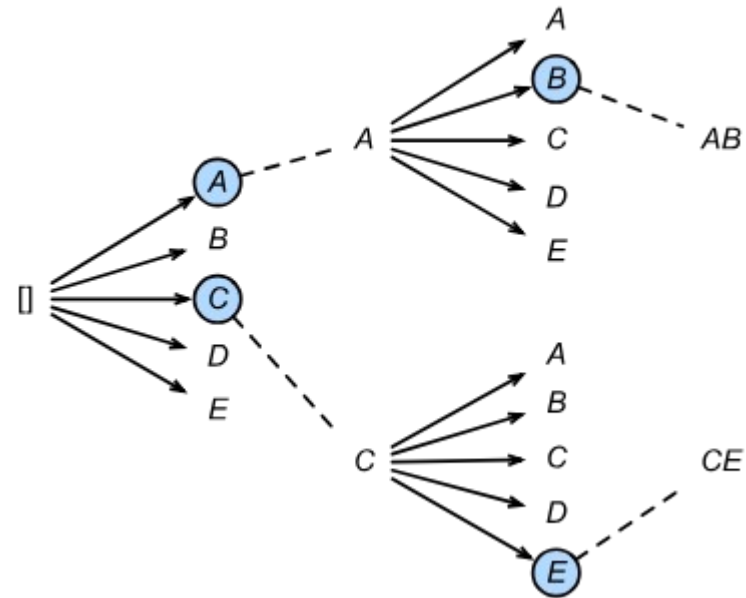
# Beam Search, example

- $k=2$
- Time step 1:
  - Keep A, C



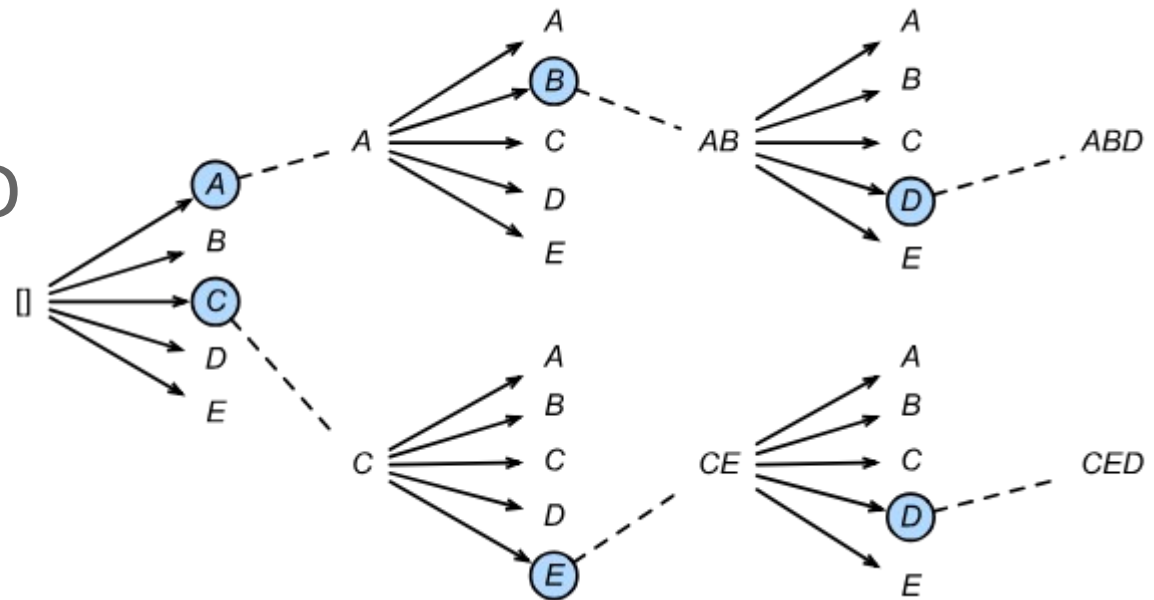
# Beam Search, example

- $k=2$
- Time step 2:
  - Keep AB, CE



# Beam Search, example

- $k=2$
- Time step 3:
  - Keep ABD, CED



# Beam Search, cont.

- If  $\epsilon$  is generated at certain branch, this branch is not expanded further
- Beam search continues until only hypotheses ending with  $\epsilon$  are among the best  $k$  hypotheses
- Different hypotheses could have different length
  - Longer hypotheses will naturally look worse than shorter ones just based on their length
  - The usual solution to this is to apply some form of length normalization to each of the hypotheses

Questions?