Advanced Natural Language Processing

RECAP: RNNs

RNNs exhibit the following advantages for sequence modeling:

- Handle variable-length sequences
- Keep track of long-term dependencies
- Maintain information about the order as opposed to FFNN
- Share parameters across the network

RNN ISSUES?

$$\mathbf{h}_t = \tanh(\mathbf{V}\mathbf{X}_t + \mathbf{U}\mathbf{h}_{t-1} + \beta_1)$$

Current Input

Previous hidden state

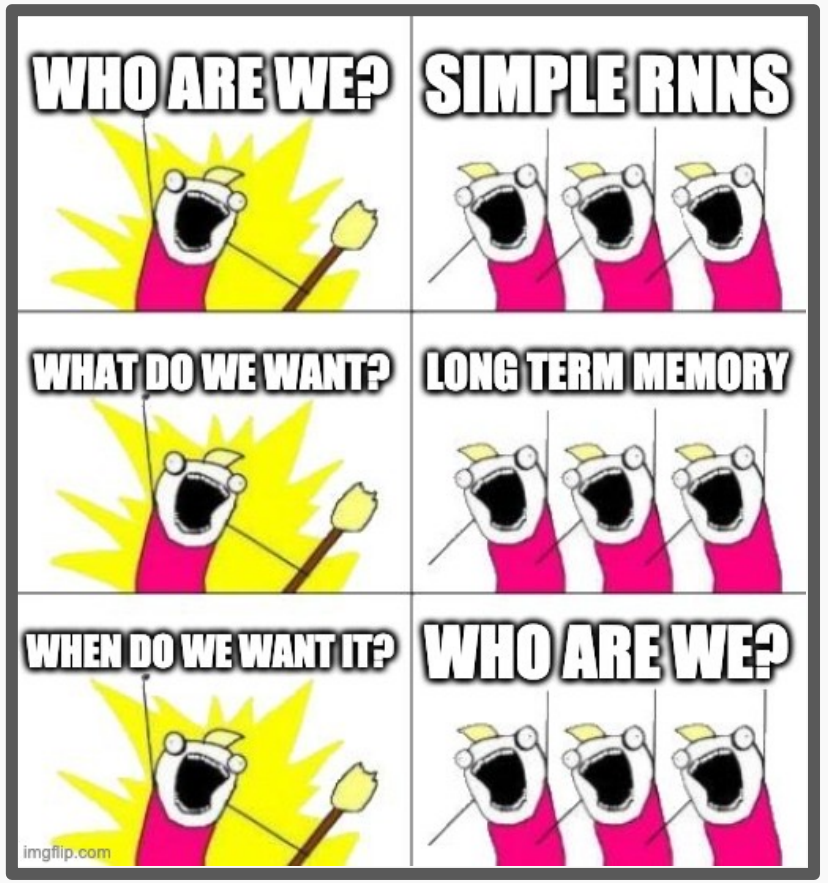
The trainable weights V, U are constants, and they are not a function of input X_t or previous state h_{t-1}.

The simple repeated structure suffers from vanishing/exploding gradients as we move farther away from the target, and hence weights do not learn from initial inputs.

RNN Wishlist?

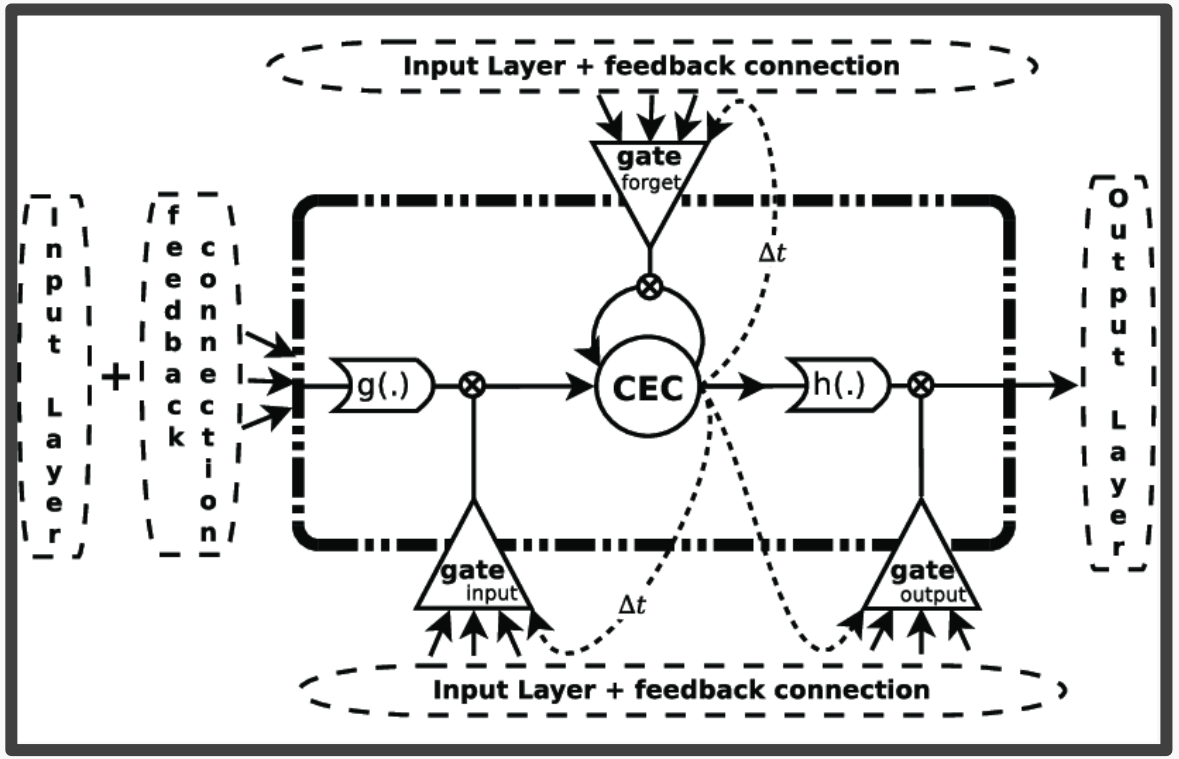
$$\mathbf{h}_t = \tanh\left(\mathbf{V}(h_{t-1}, X_t)\mathbf{X}_t + \mathbf{U}(h_{t-1}, X_t)\mathbf{h}_{t-1} + \beta_1\right)$$

- We want our trainable weights V, U to somehow incorporate the input X_t and the previous state h_{t-1}.
- We need to solve the vanishing gradient problem so that our network also learns from inputs at the beginning of a sequence.



Long short-term memory (LSTM)

- First introduced in **1995** to counter the vanishing gradient problem.
- Training was done using a mixture of Real Time Recurrent Learning and Backpropagation Through Time.
- Underwent several major modifications including addition of *forget gate , peephole connections* etc.
- In the last two decades, several other variants were introduced with mixed results

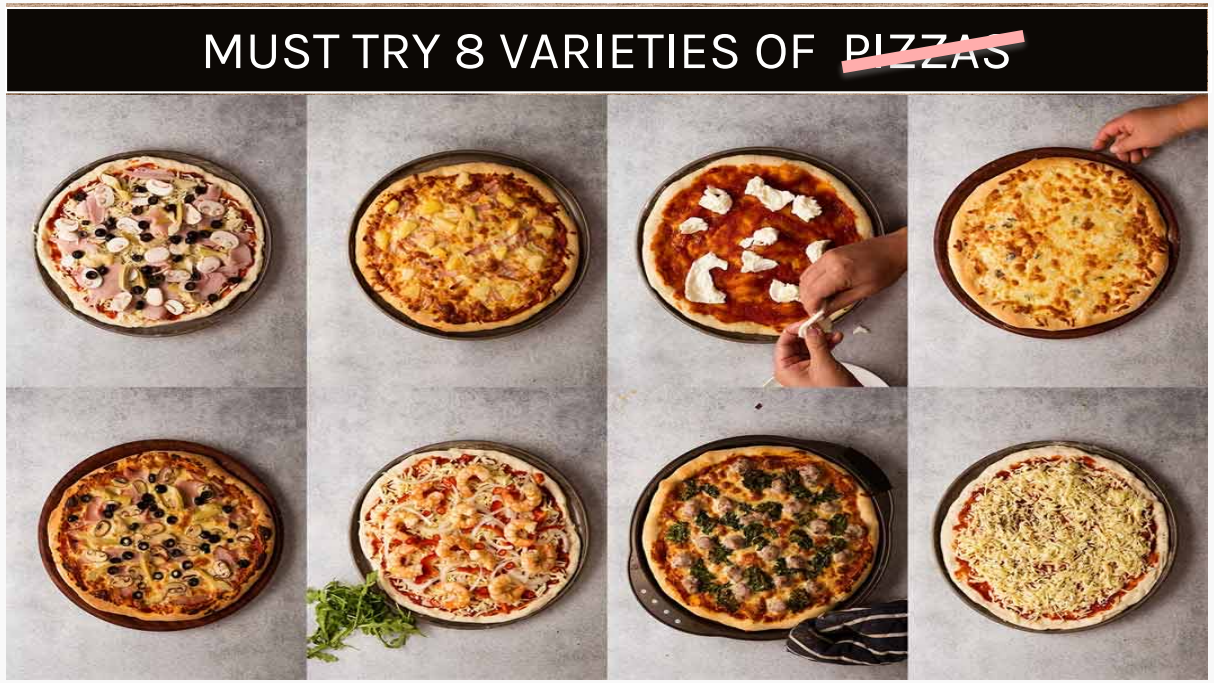


- *First prototype of LSTM cell*

Long short-term memory (LSTM)

After its introduction in 1995, here are the eight popular types of LSTM variants other than the vanilla version:

- **NIG – No input gate**
- **NFG - No forget gate**
- **NOG – No output gate**
- **NIAF – No input activation**
- **NOAF – No output activation**
- **CIFG – Coupled input/forget gate**
- **NP – No peepholes**
- **FGR – Full gate recurrence**



Please refer to the paper ***LSTM: A Search Space Odyssey*** for a thorough analysis of all the variants

ELMo Embeddings

Review of Language Models

A Language Model predicts the next word x_{t+1} in a sentence based on previous words.

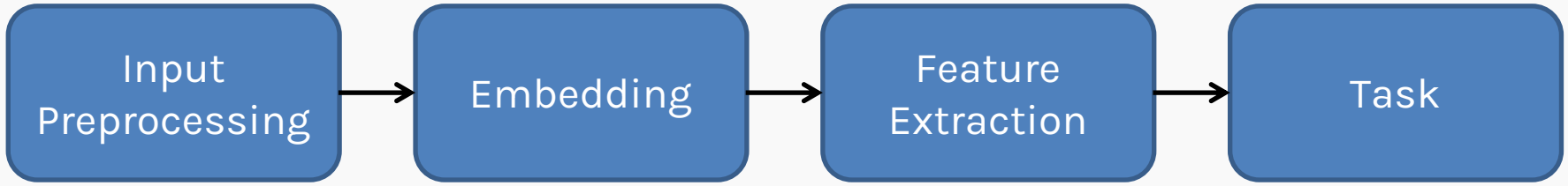
$$P(x_{t+1}|x_t, x_{t-1}, x_{t-2}, \ldots, x_0; \Theta)$$

Remember that in any model, the learned parameters always condition the model.

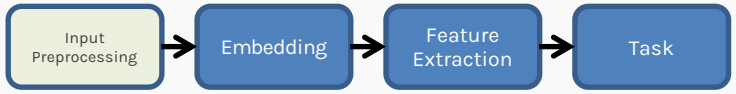
Language Models: Complete Picture

So far, we have been talking about embeddings, neural networks and language models, but without the complete picture.

We can split the language model into four parts:



Input and Preprocessing



Input: The entire collection of data is the corpus in which we train on.

Corpus can be a small dataset such as IMDB, or an enormous such as Wikipedia.

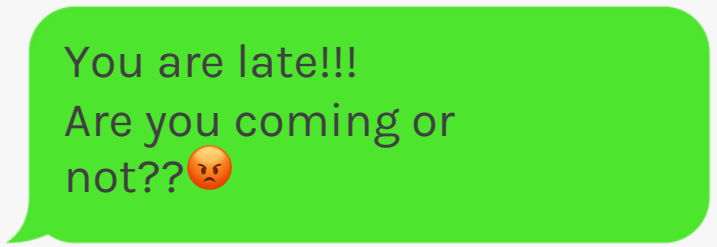
From that corpus we can define a vocabulary, which is composed of the unique words in the corpus. It will influence the complexity of our language model.

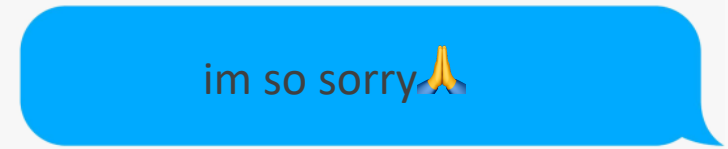
Input and Preprocessing

Preprocessing: The data generally is collected from sources such as the internet.

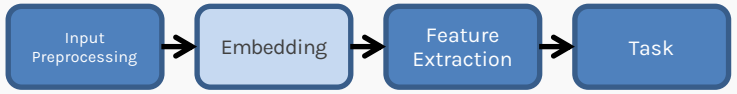
It comes as it is, with punctuation signs, grammatical mistakes and even emojis.

In general, language models often use preprocessed text, without uppercase letters, and where punctuation signs and non-text data are removed.





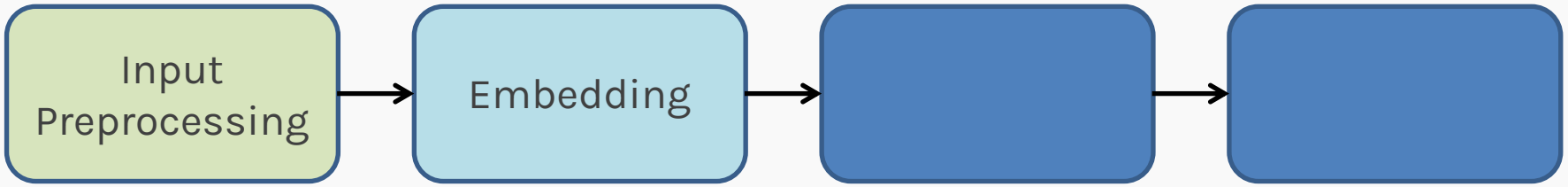
Embeddings



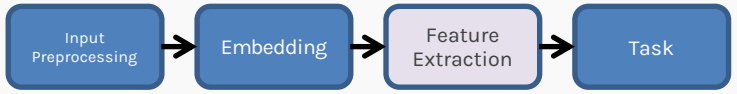
Embedding:

This stage transforms the data to be fed to the network. Embedding layer transforms every word into a fixed length vector with arbitrary dimension.

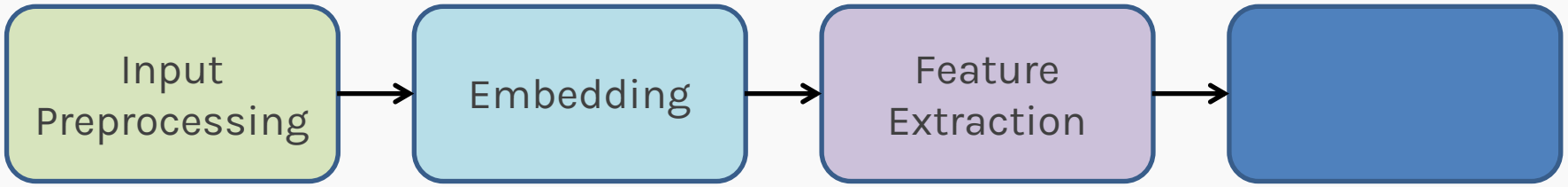
One-hot-encoding or word2vec are some examples.



Feature extraction

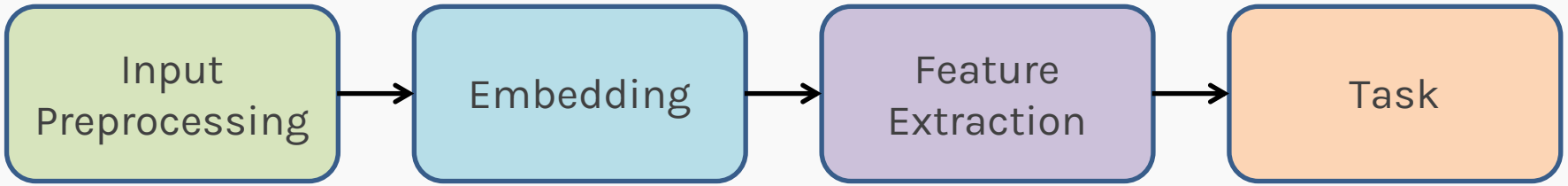


Feature extraction: This stage is the model. Usually a neural network. From the sequence of embeddings, we generate a new representation which will be used in the next stage. Its job is to capture as much contextual information as possible.



Task

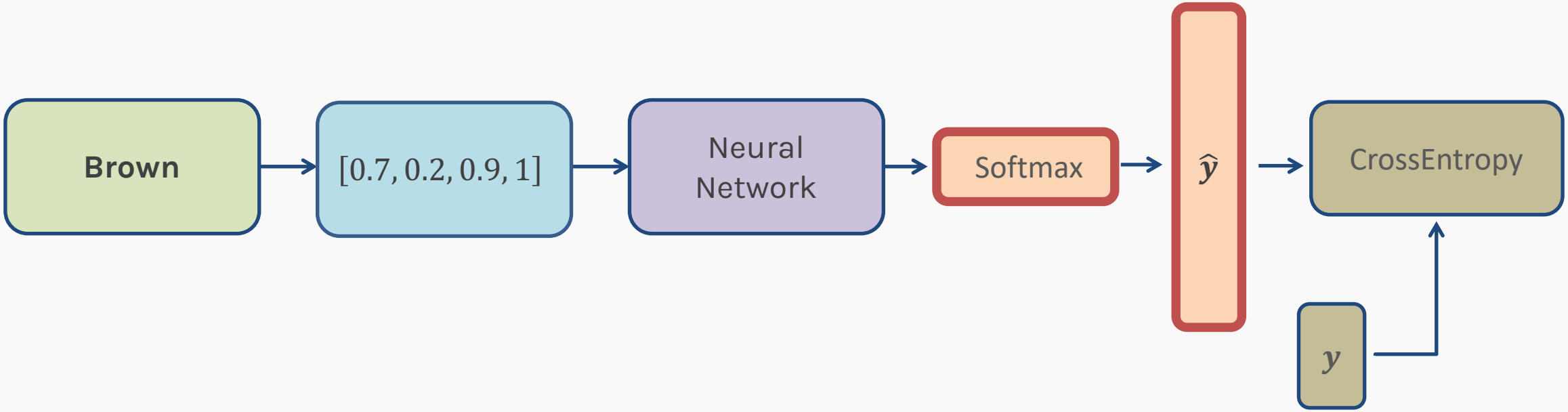
Task: The task of the language model predicts the next word in the sentence.



Training

To train the model, we need to compare our normalized prediction with the ground truth $\boldsymbol{y}$, using the Cross-Entropy loss.

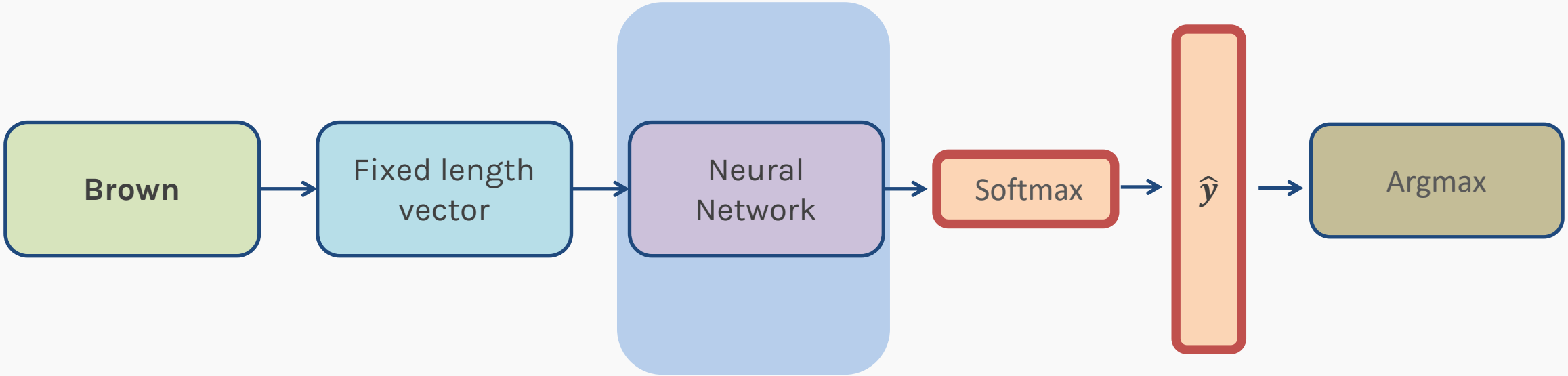
In general, $\boldsymbol{y}$ is one-hot encoded.



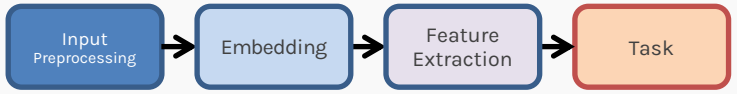
Inference/Prediction

Once the model is trained, we freeze the learned weights of the network. They will not change anymore.

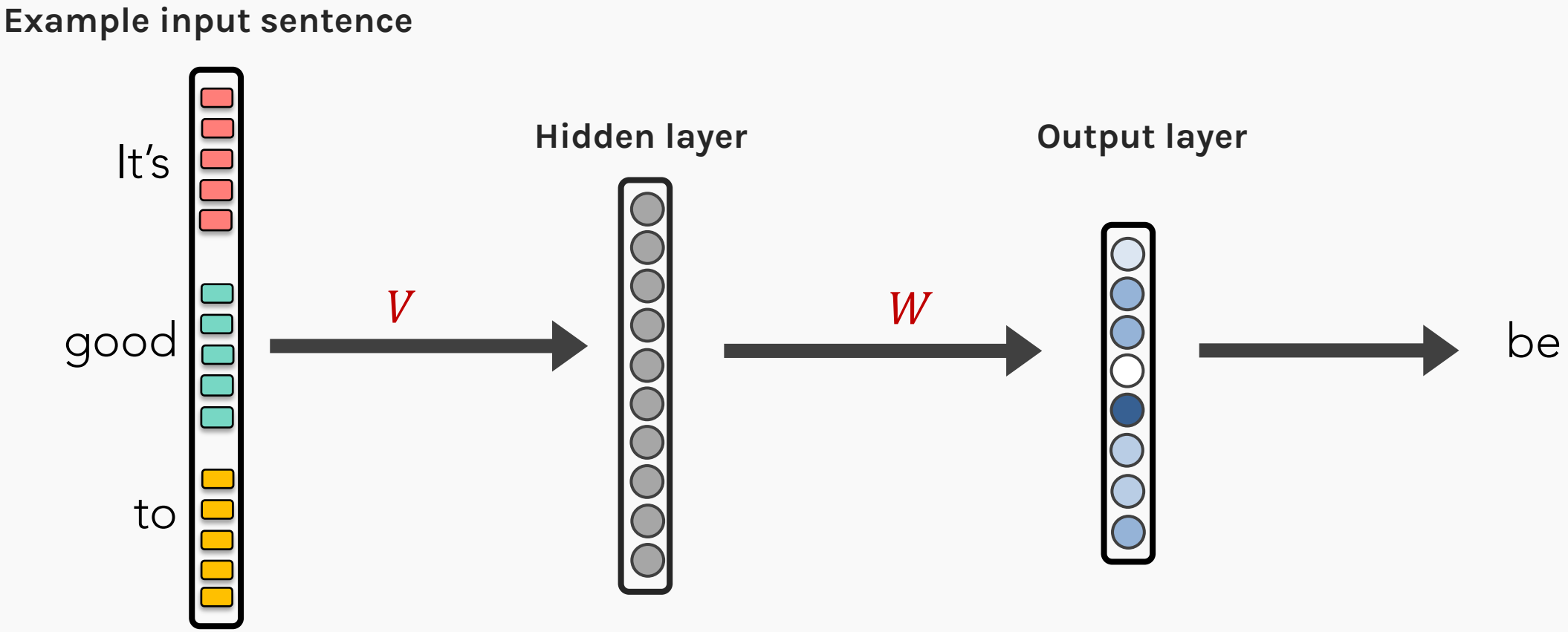
Our predicted word is the one with the maximum probability.



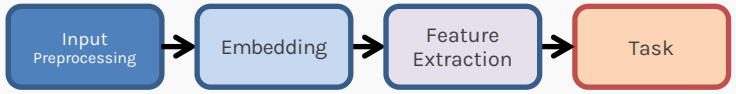
Neural Networks as Language Models



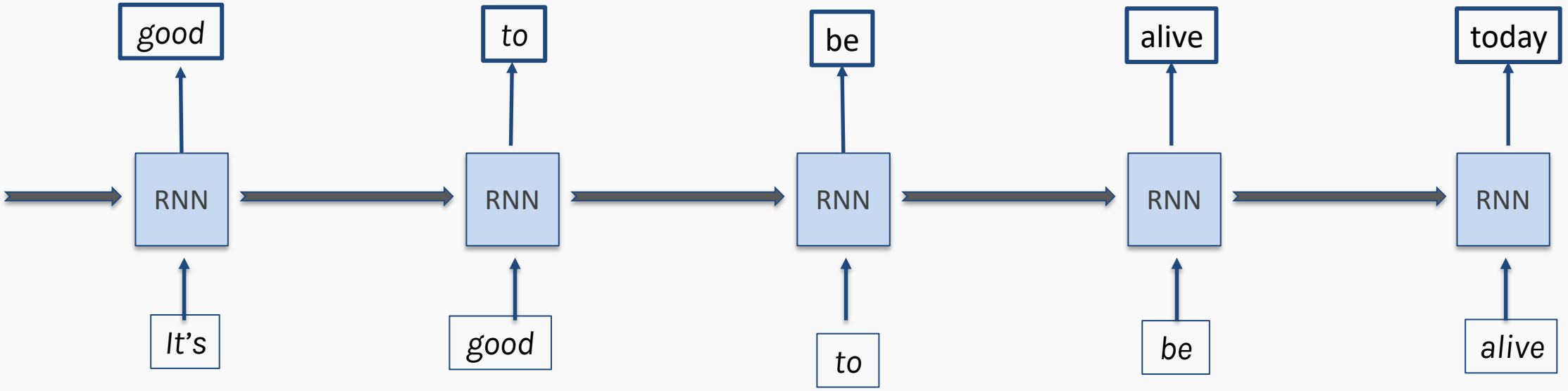
We have seen a FFCC network as a LM in previous lecture.



Neural Networks as Language Models



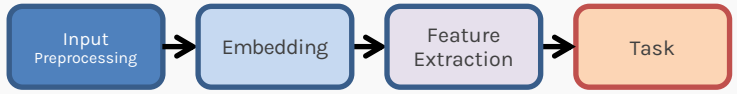
We have used an RNN to process text:



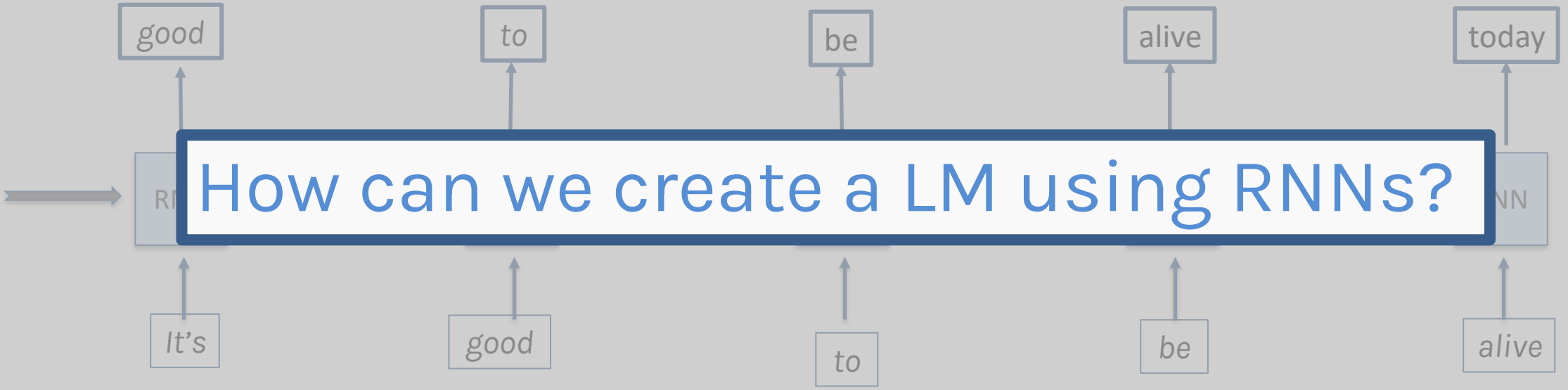
The key advantage of RNN is that they have a memory and and be unrolled to variable length sentences.

In RNN, we use each hidden state to predict the corresponding word.

Neural Networks as Language Models



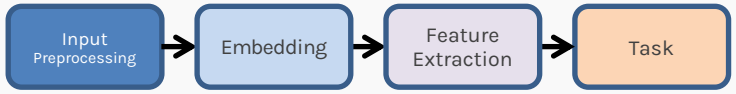
And in session 4, we used an RNN to process text:



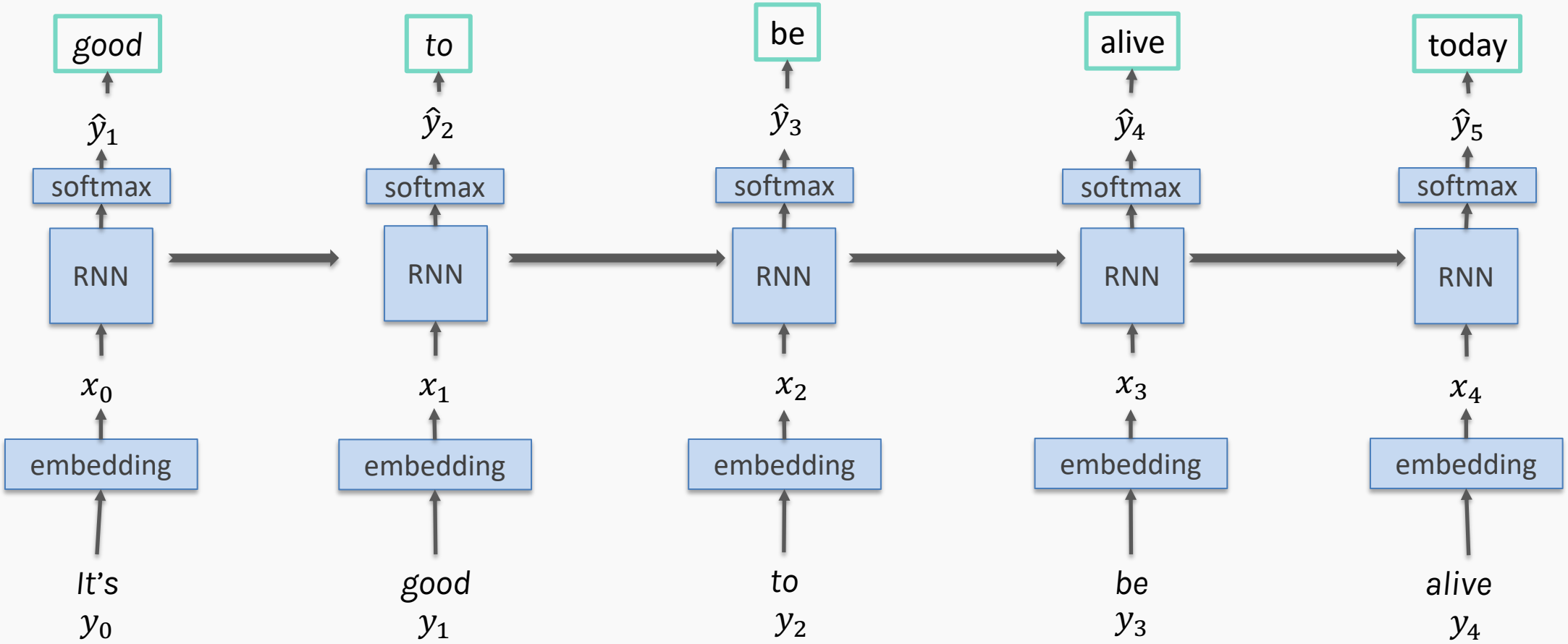
The key advantage of RNN is that they have a memory and and be unrolled to variable length sentences.

In RNN, we use each hidden state to predict the corresponding word.

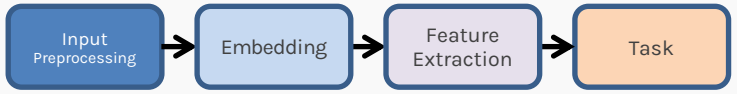
Neural Networks as Language Models



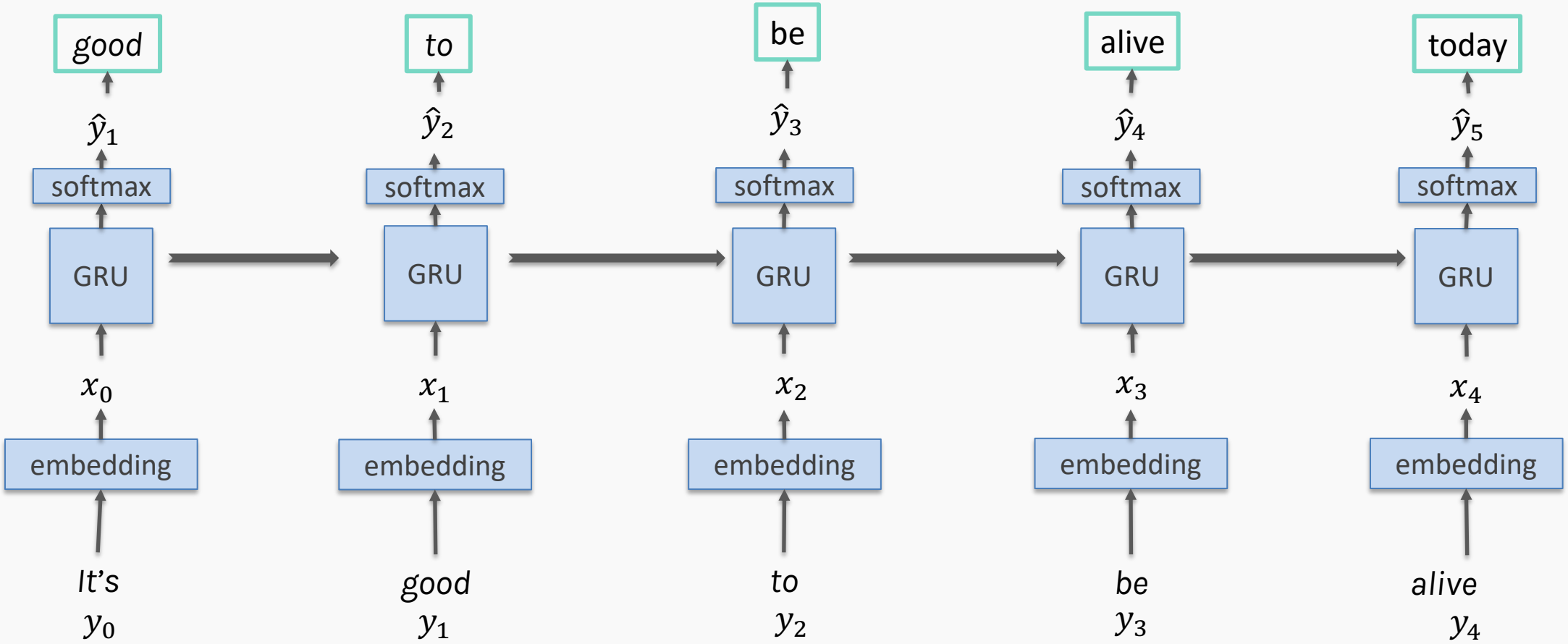
If we want to use the RNN as a LM, it will look like this:



Neural Networks as Language Models



We can use any RNN Cell, such as GRU or LSTMs.



Working towards ELMo

The semantic information is not always contained in the past.

Some words in the future might be important to understand the information in the present.

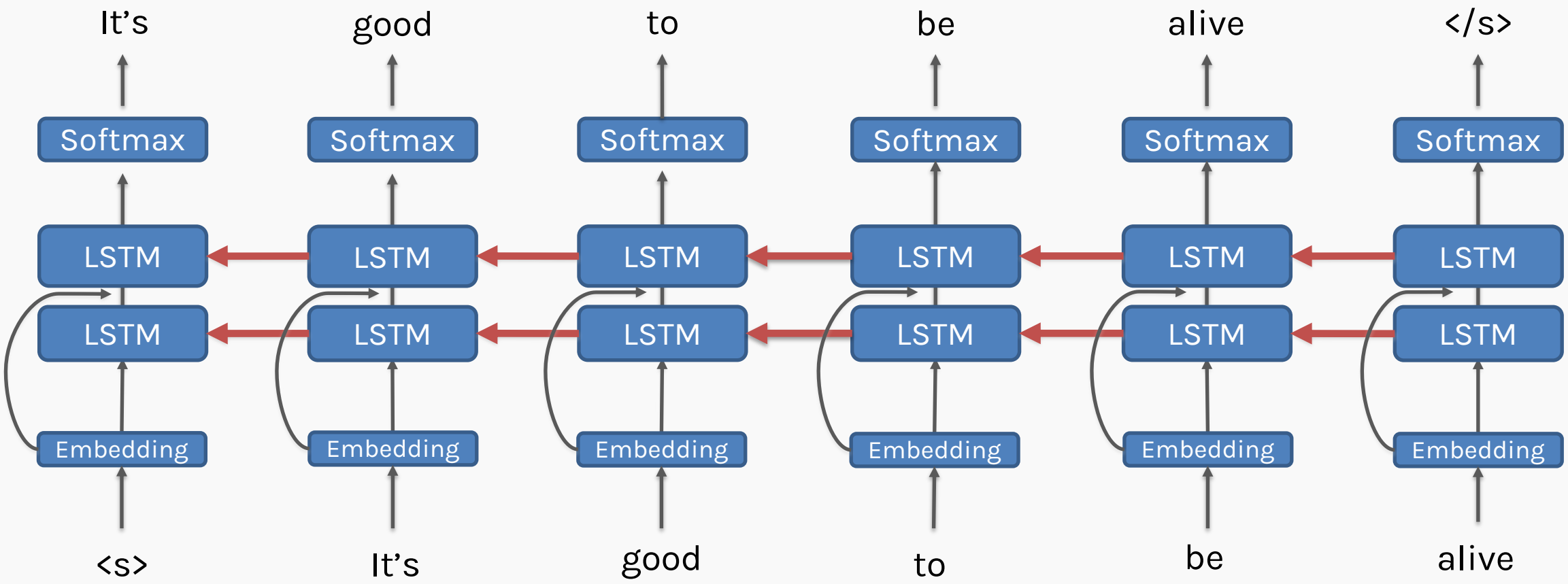
That is why we listen until the other person stops talking!

We can improve the model using another LSTM going in the opposite direction.

We predict the previous word in the sentence, starting from the end.

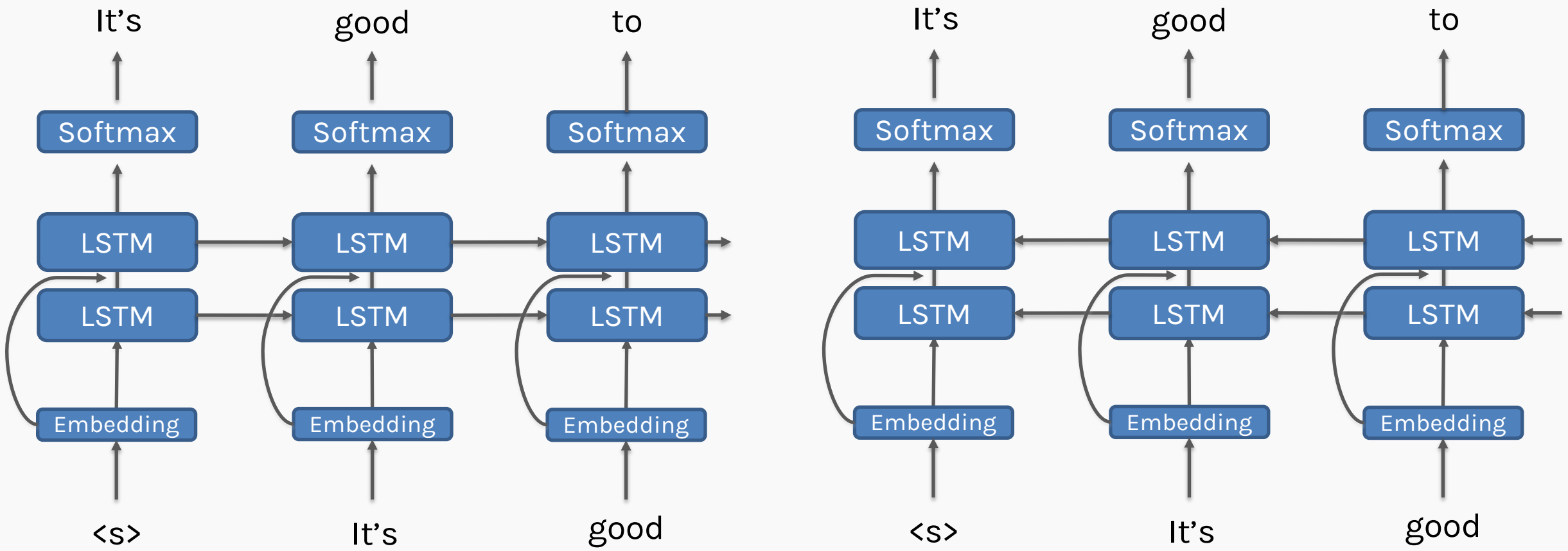
Working towards (backwards) ELMo

We replicate the same structure but in the opposite direction.



Working towards/backwards ELMo

The final model will contain two networks.



Bidirectional LSTM network

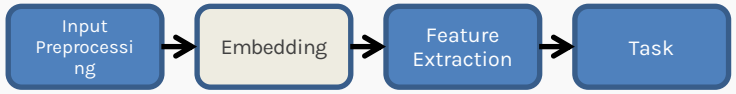
We optimize both networks at the same time. For each word, we optimize both RNNs jointly:

$$L = -\sum y \log(\hat{y}_{right}) - \sum y \log(\hat{y}_{left})$$

Each direction of the LSTM predict the same word, but from different directions.

We will come back to this part later.

Word embeddings: word2vec



Let’s focus on the word2vec embeddings.

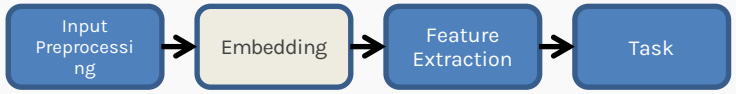
They are constructed with a fixed vocabulary.

Is better than a one-hot vector but it doesn’t consider the context of the entire sentence.

They work but have some drawbacks.

What if we are facing a word that we don’t know? We put it as an unknown.

Character CNN



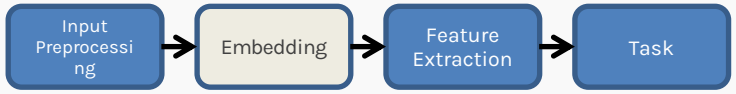
Idea:

What if we use the characters to create an embedding?

We know that all the words in a language are made from a fixed number of characters.

If we use characters and not logograms, like the Japanese language.

Character CNN

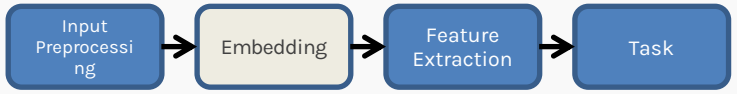


The Benefits?

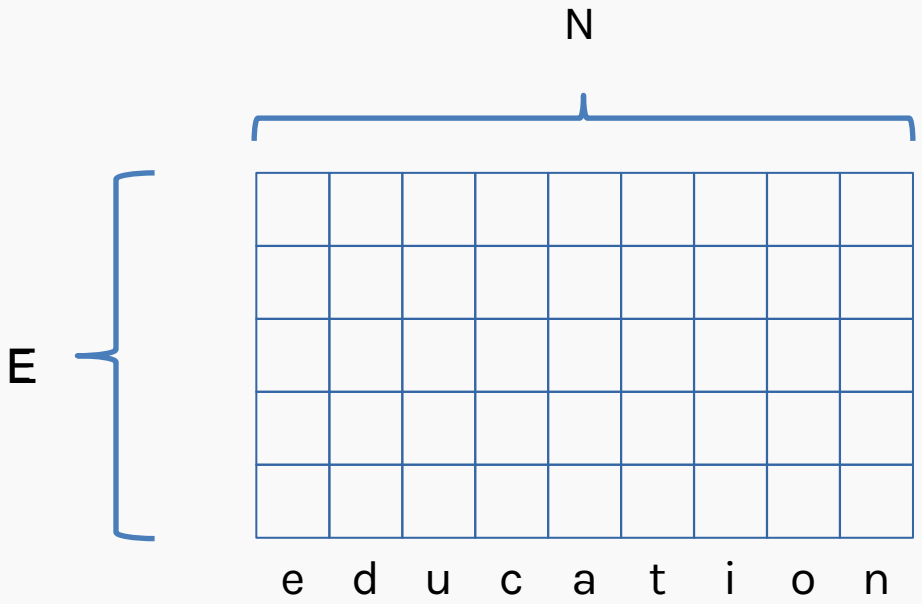
We have less characters than words.

Some words are union of smaller parts, like prefixes or contractions.

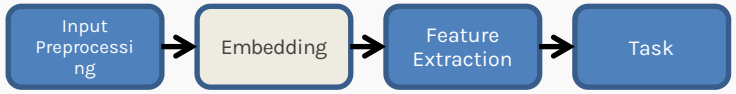
Character CNN



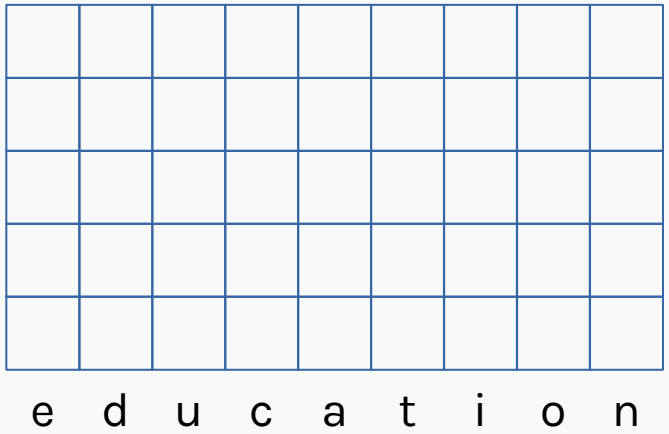
For each character in a word of length **N**, we create an embedding of dimension **E**. It can be a one-hot encoding or a trainable one.



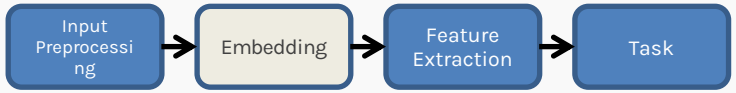
Character CNN



We want to extract information from the matrix. For this reason, we apply a convolutional filter.

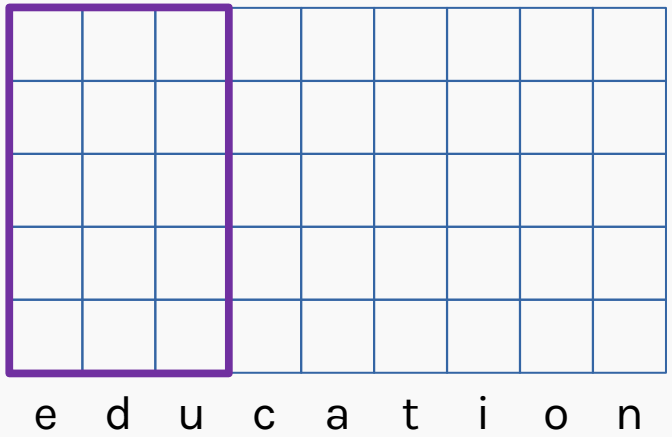


Character CNN

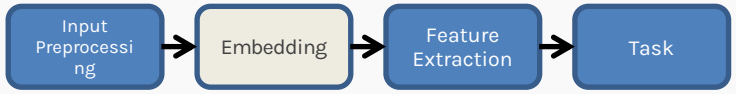


We want to extract information from the matrix. For this reason, we apply a convolutional filter.

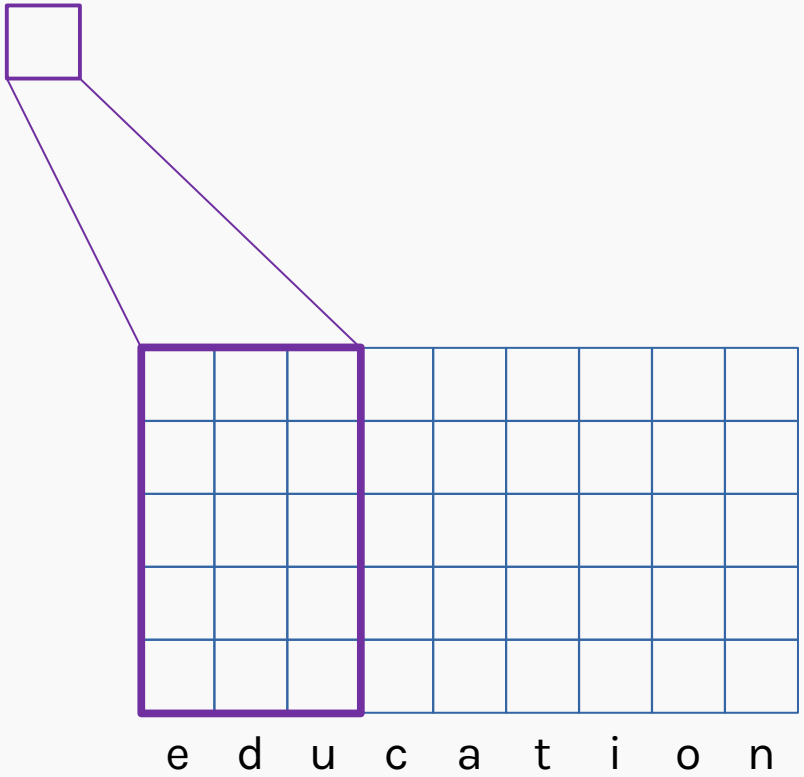
We will apply a 5x3 kernel.



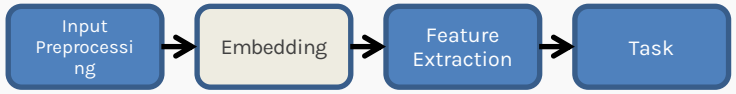
Character CNN



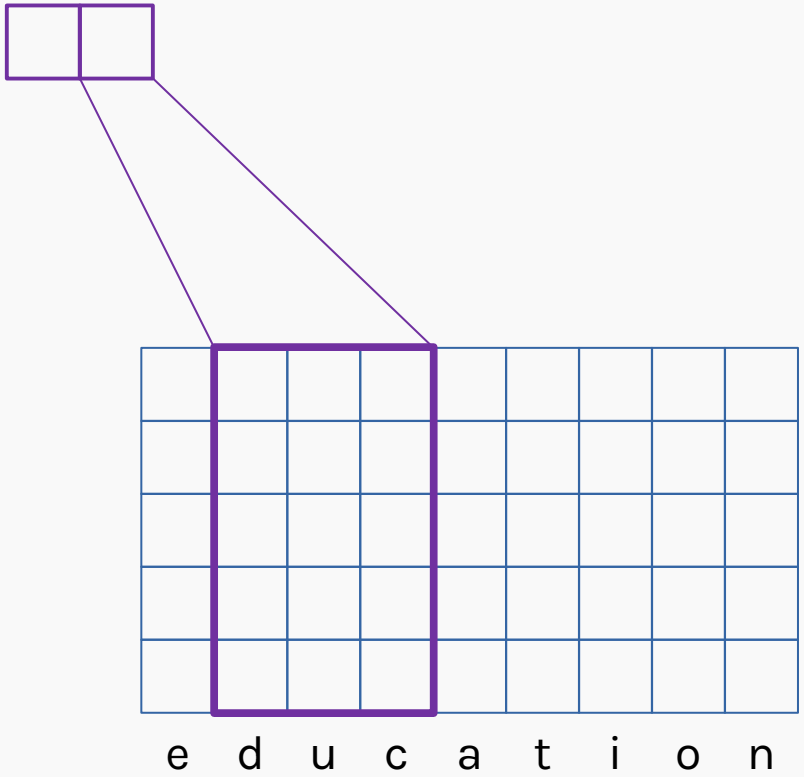
We extract the information to one value.



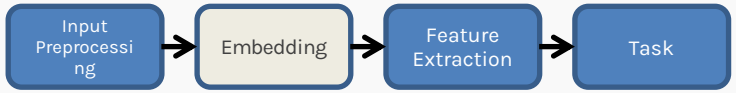
Character CNN



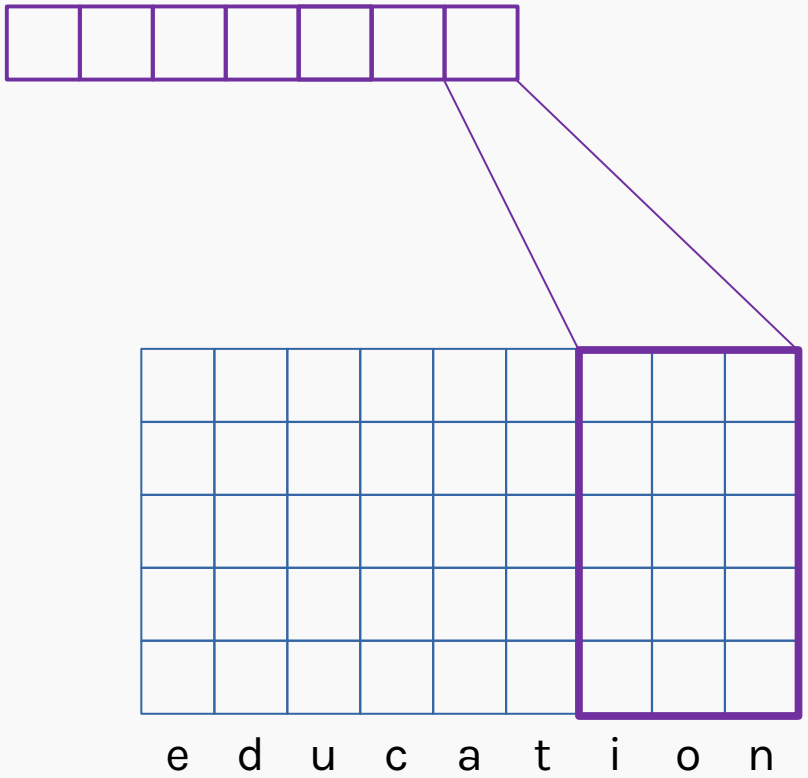
We move the kernel to the side to keep the convolution.



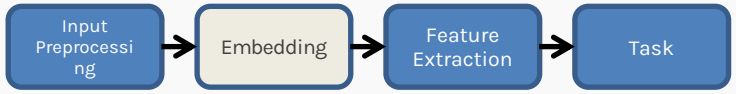
Character CNN



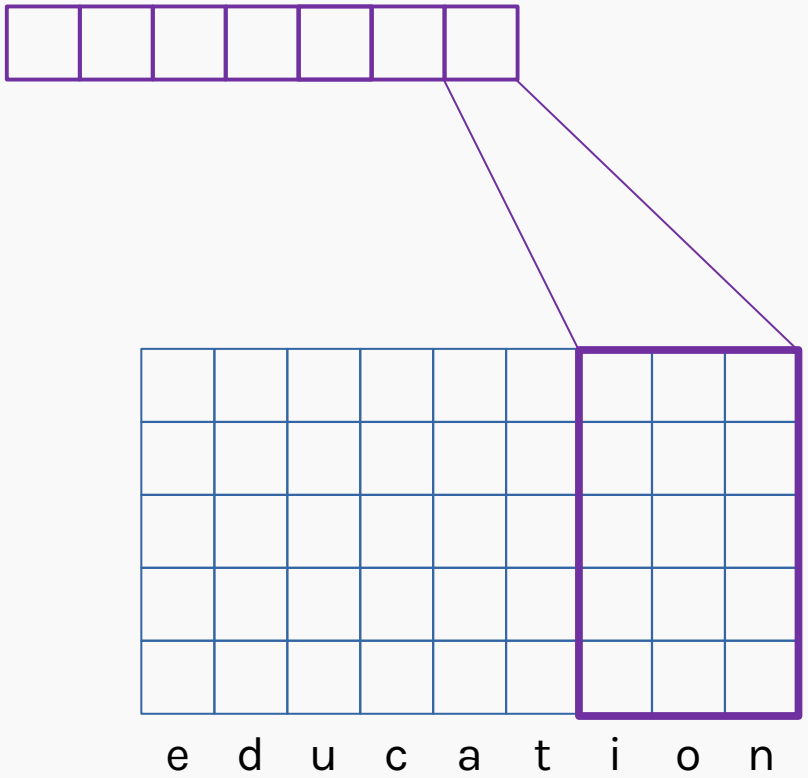
We continue until we complete the sequence.



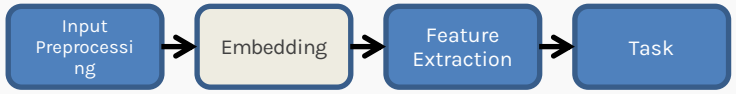
Character CNN



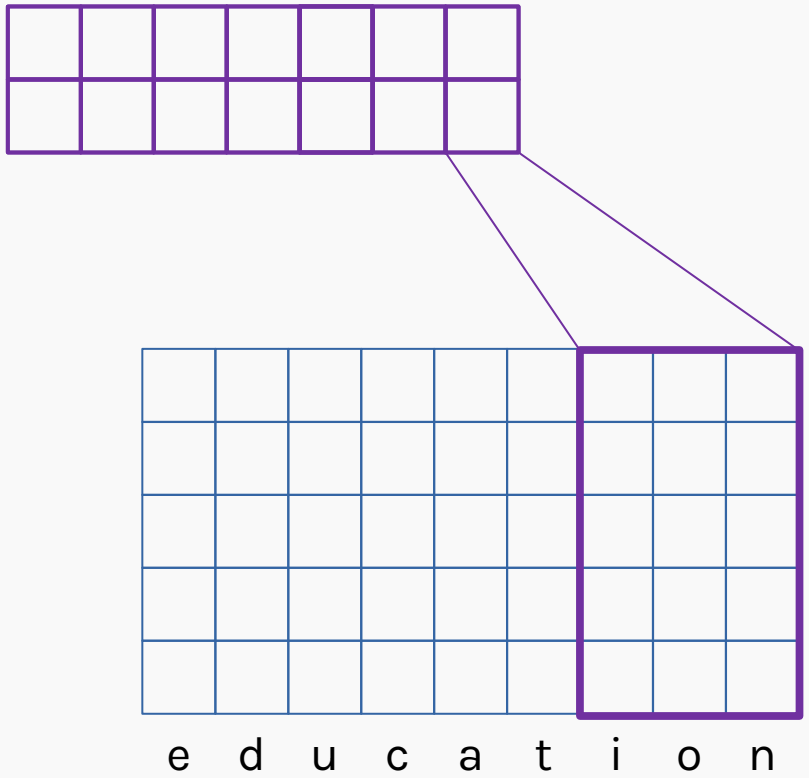
The kernel 5×3 obtains a 1×7 vector.



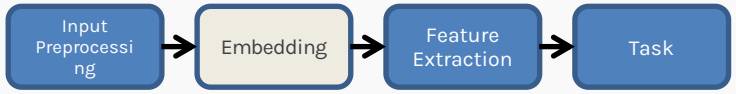
Character CNN



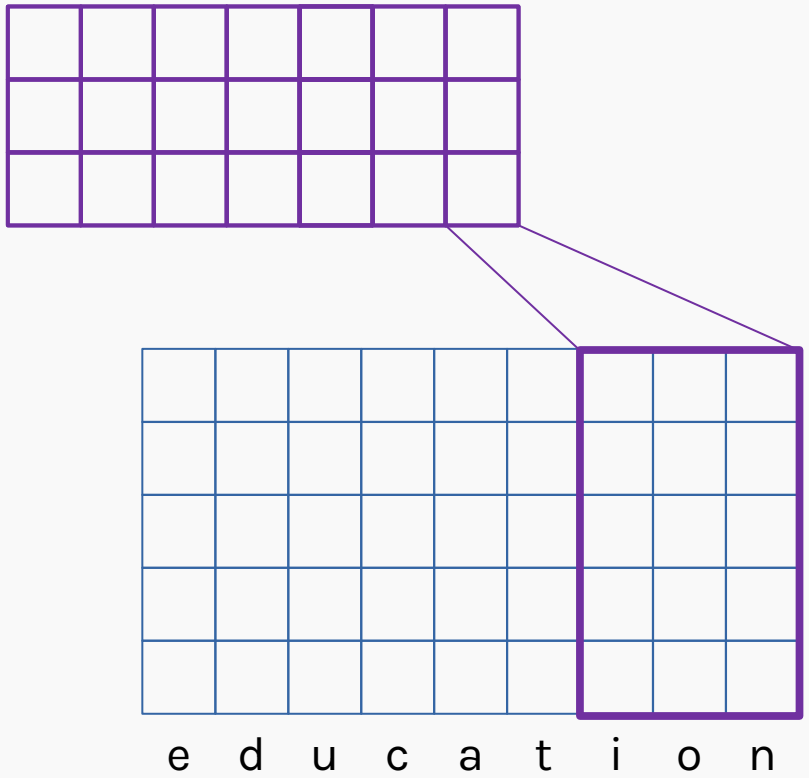
We can add more kernels



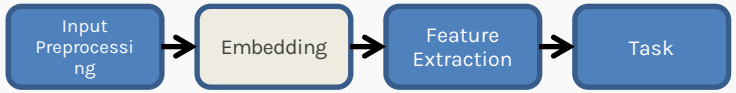
Character CNN



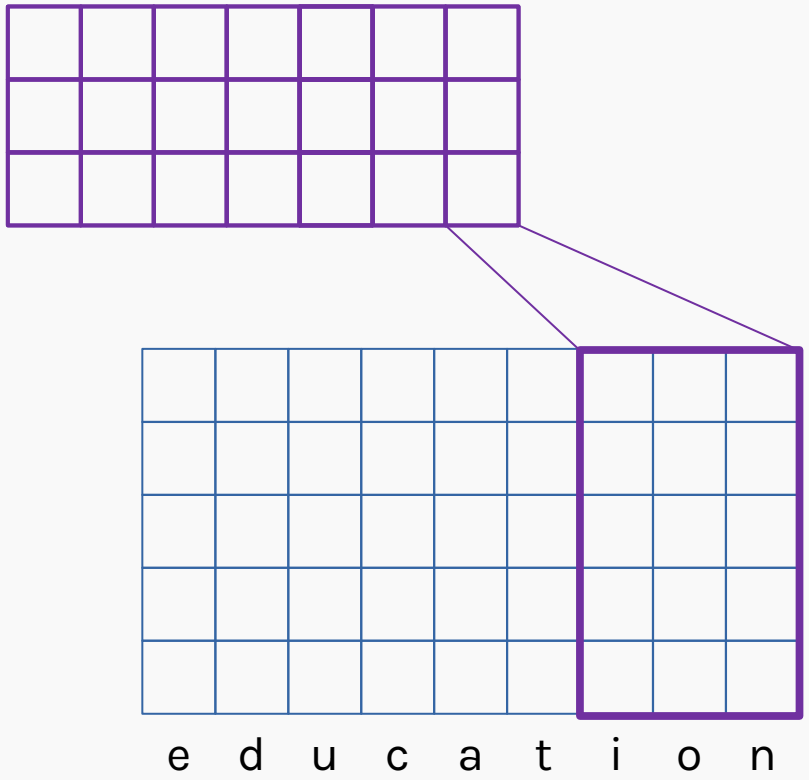
Each kernel adds another row to the result.



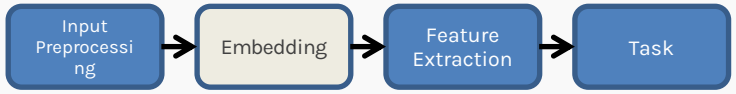
Character CNN



For this filter (which contains 3 kernels), the representation is a matrix of 3×7 .

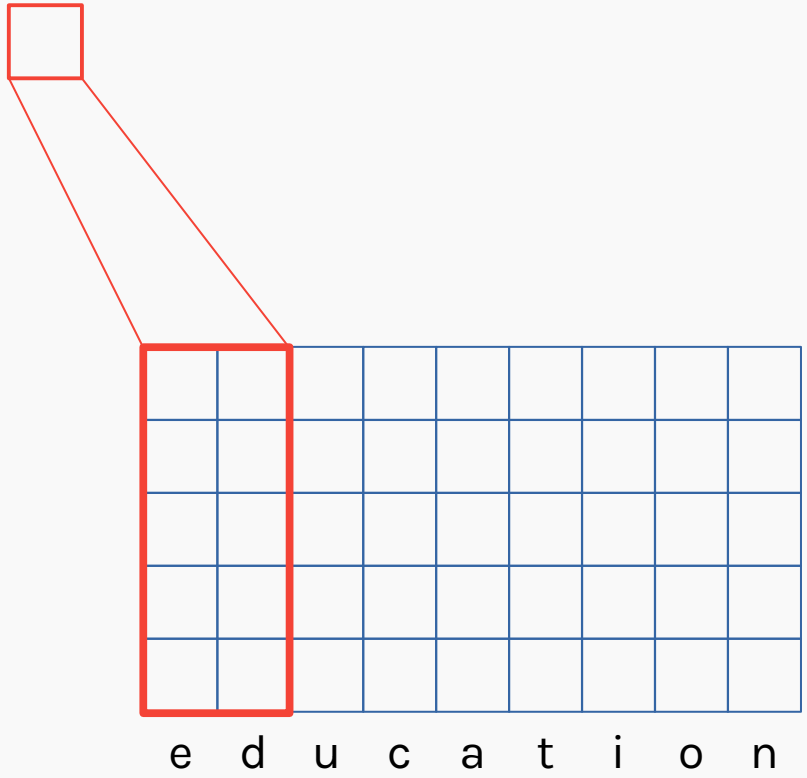


Character CNN

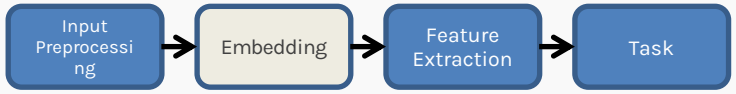


We can add filters with kernels of different widths.

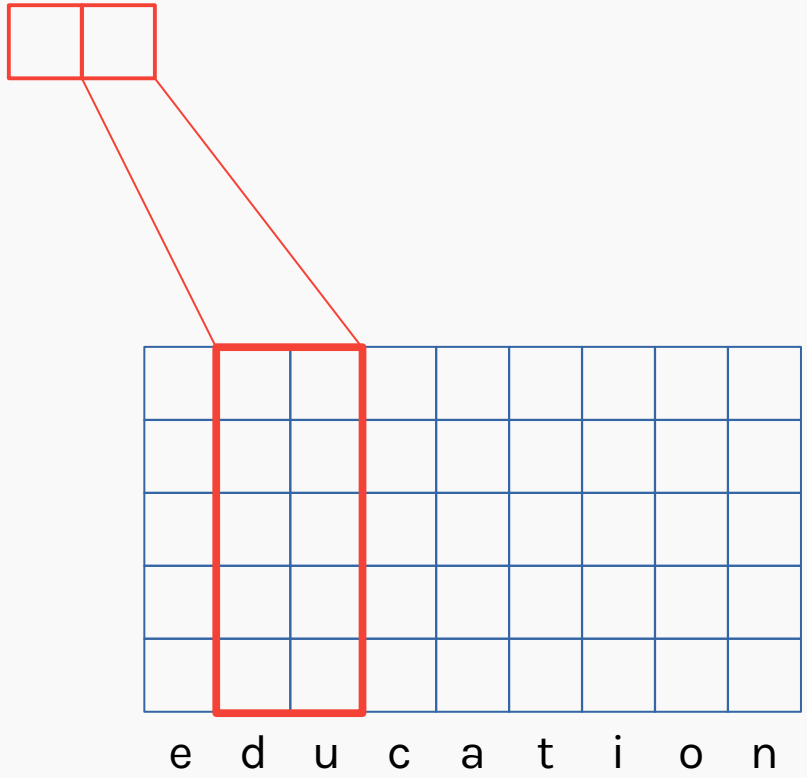
Each kernel size will capture n-gram-like features.



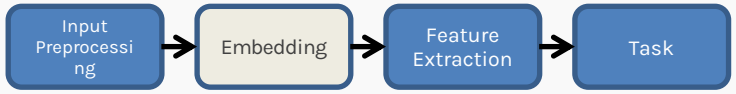
Character CNN



We can add filters with kernels of different widths.
Each kernel size will capture n-gram-like features.

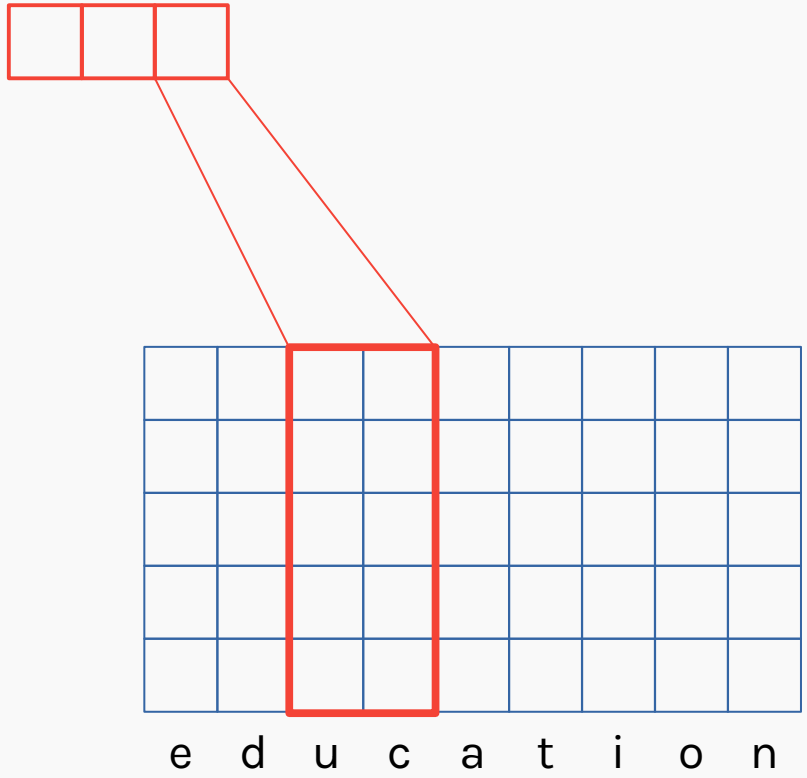


Character CNN

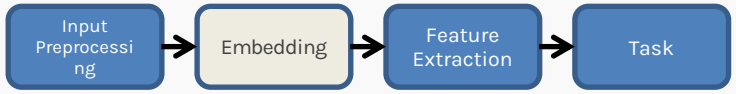


We can add filters with kernels of different widths.

Each kernel size will capture n-gram-like features.

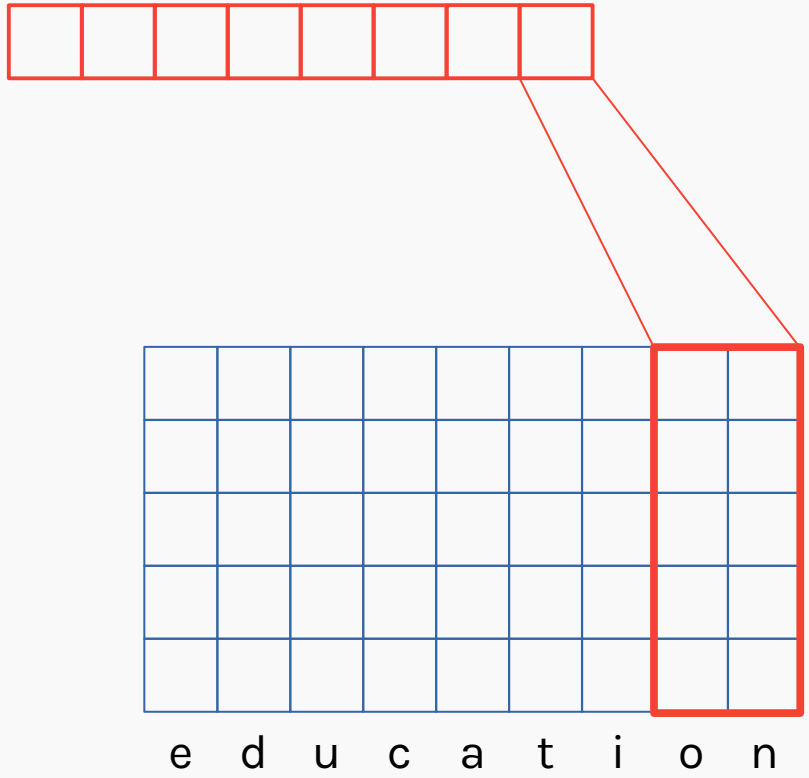


Character CNN

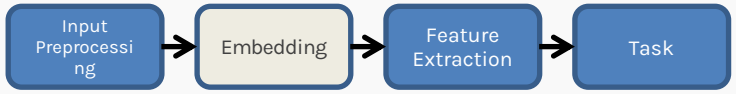


We can add filters with kernels of different widths.

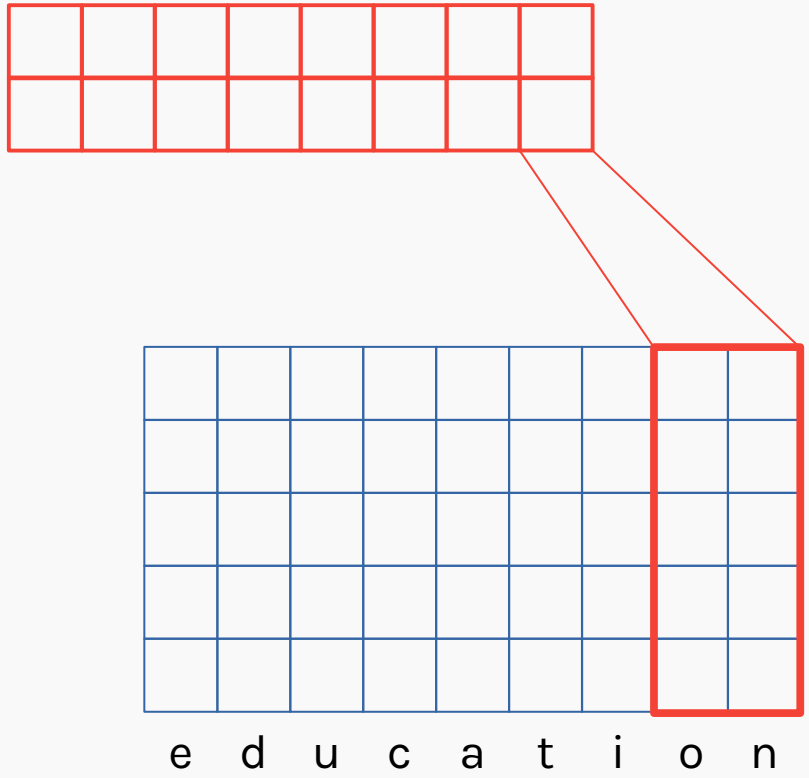
Each kernel size will capture n-gram-like features.



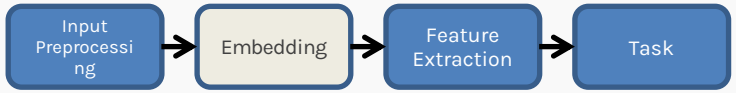
Character CNN



We can add filters with kernels of different widths.
Each kernel size will capture n-gram-like features.

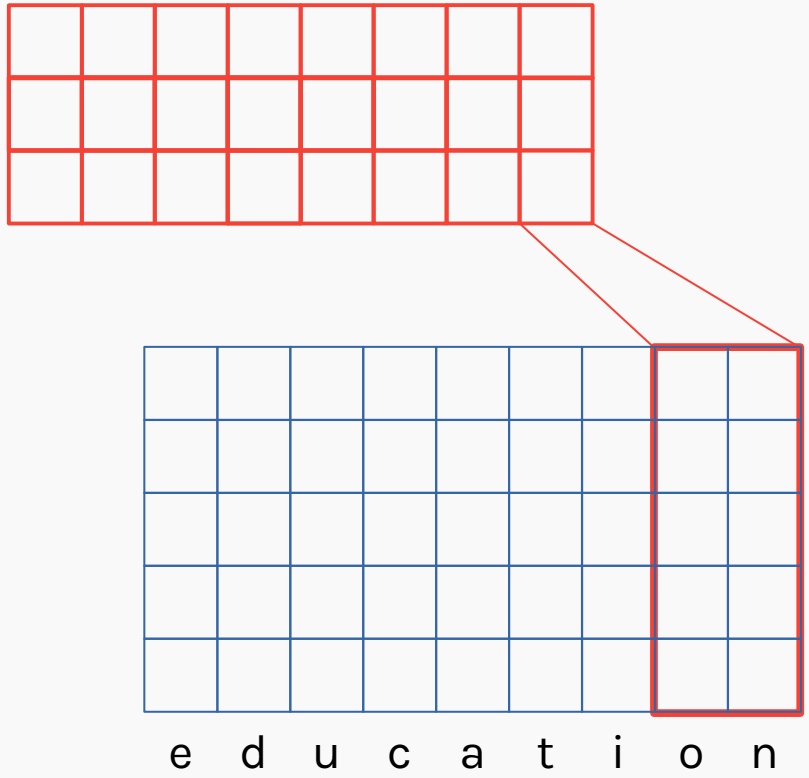


Character CNN

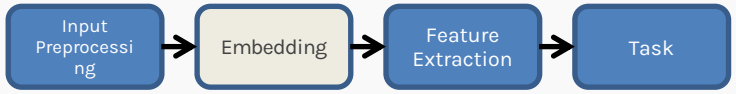


We can add filters with kernels of different widths.

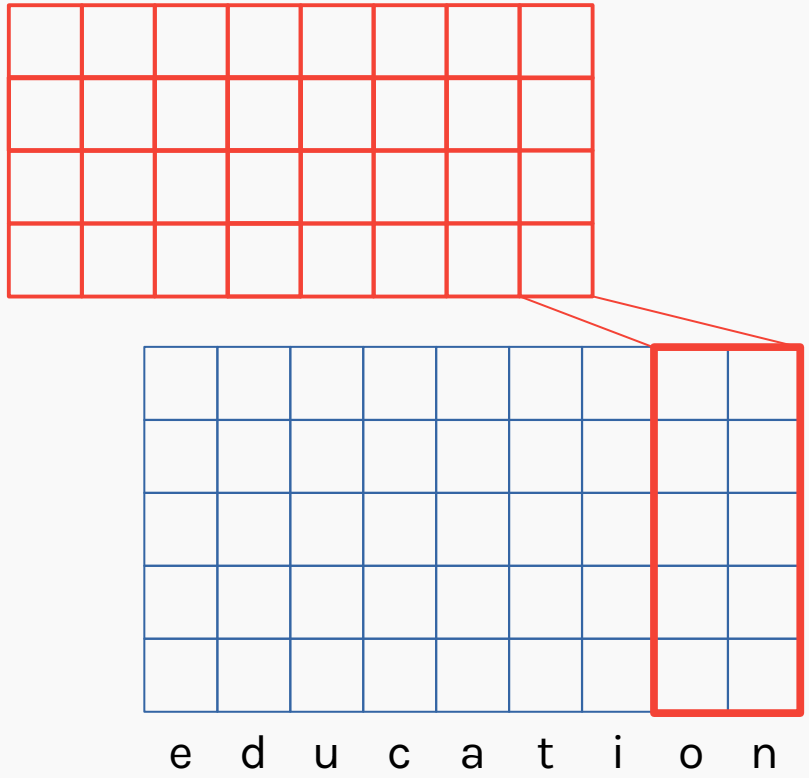
Each kernel size will capture n-gram-like features.



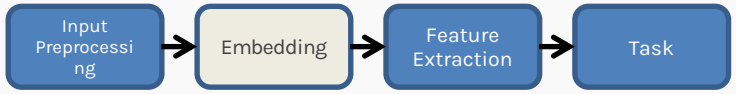
Character CNN



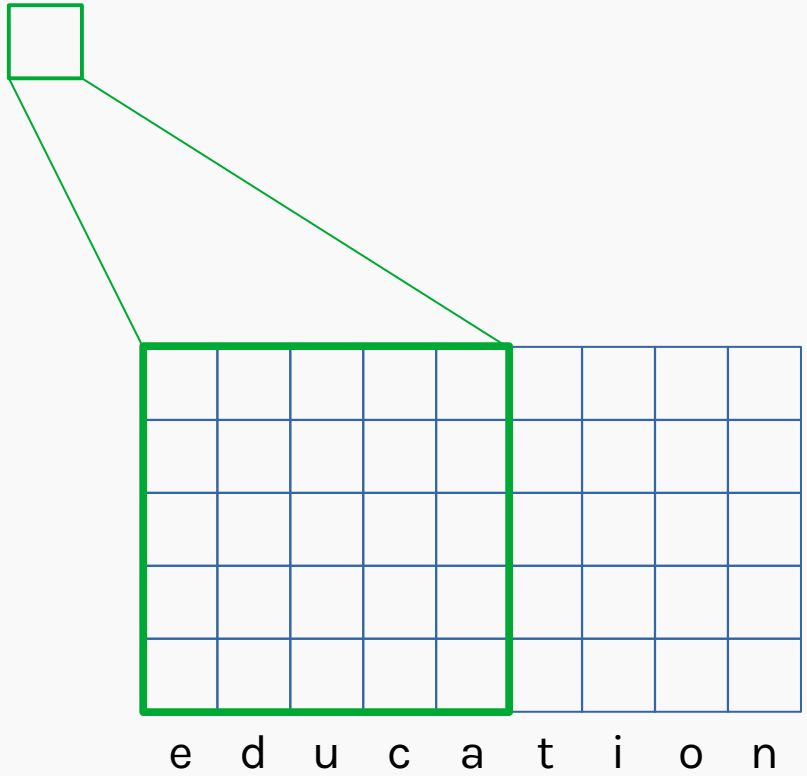
The number of kernels does not need to be the same for all the filters.



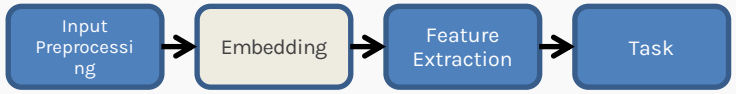
Character CNN



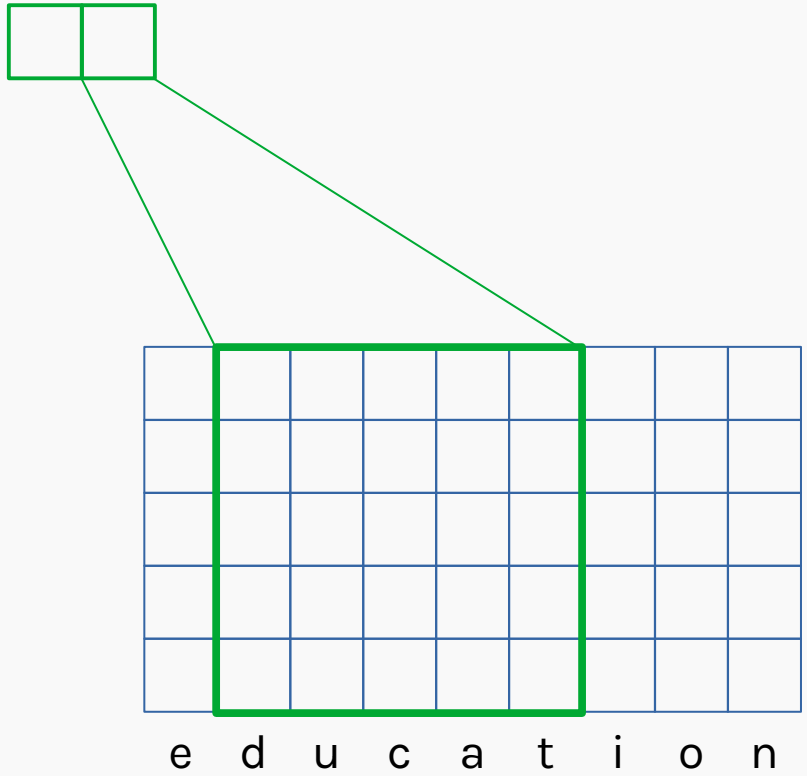
Wider kernels allow to capture long-range relations.



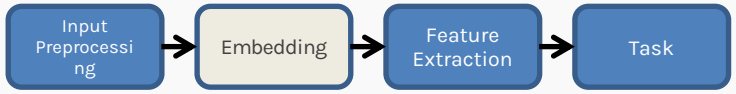
Character CNN



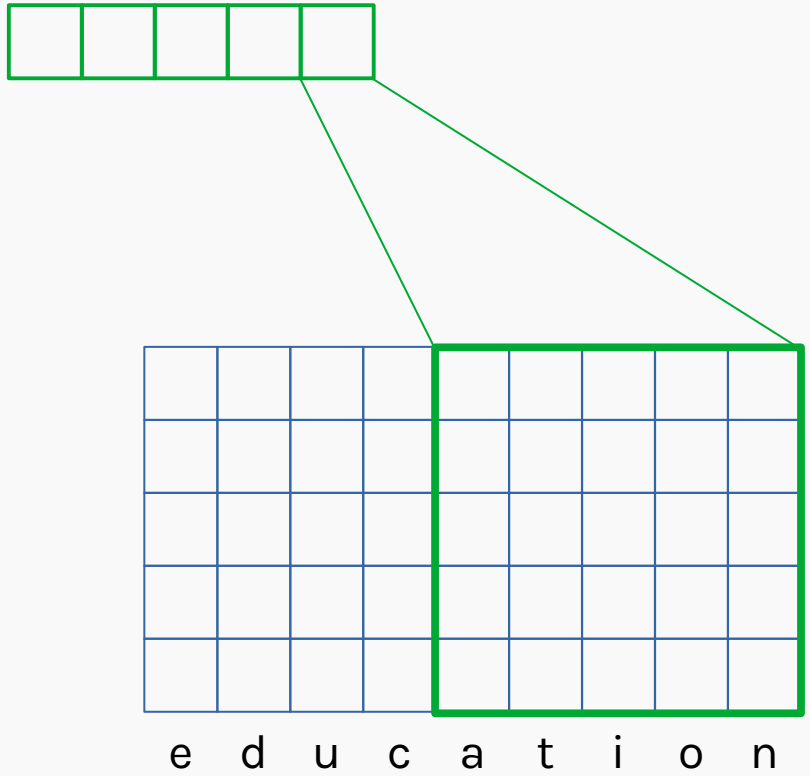
Wider kernels allow to capture long-range relations.



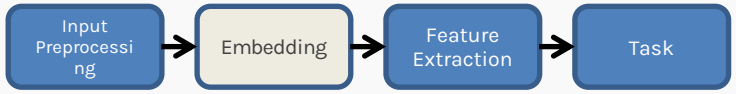
Character CNN



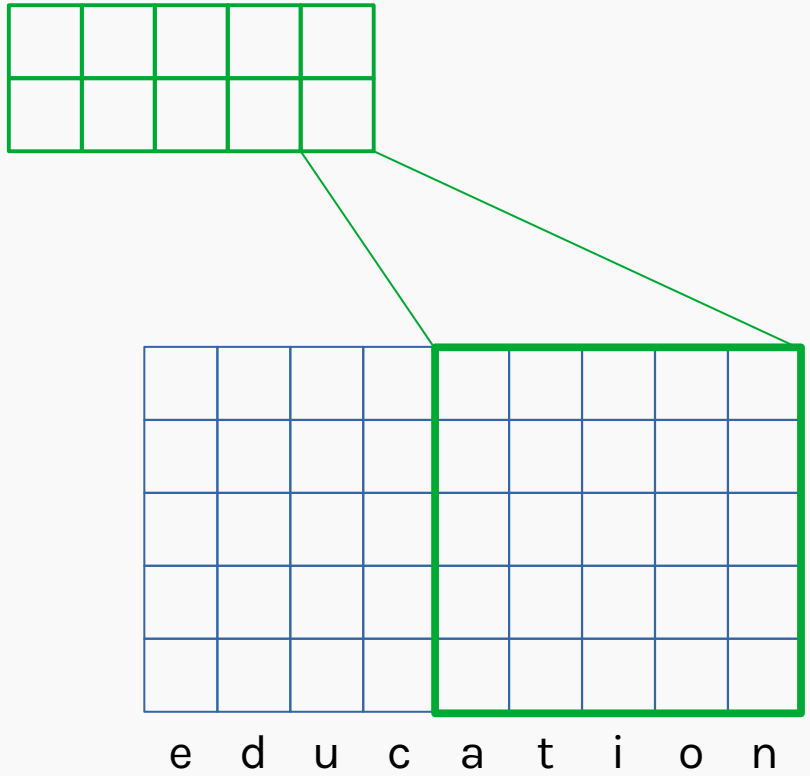
Wider kernels allow to capture long-range relations.



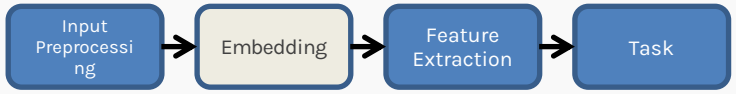
Character CNN



Wider kernels allow to capture long-range relations.

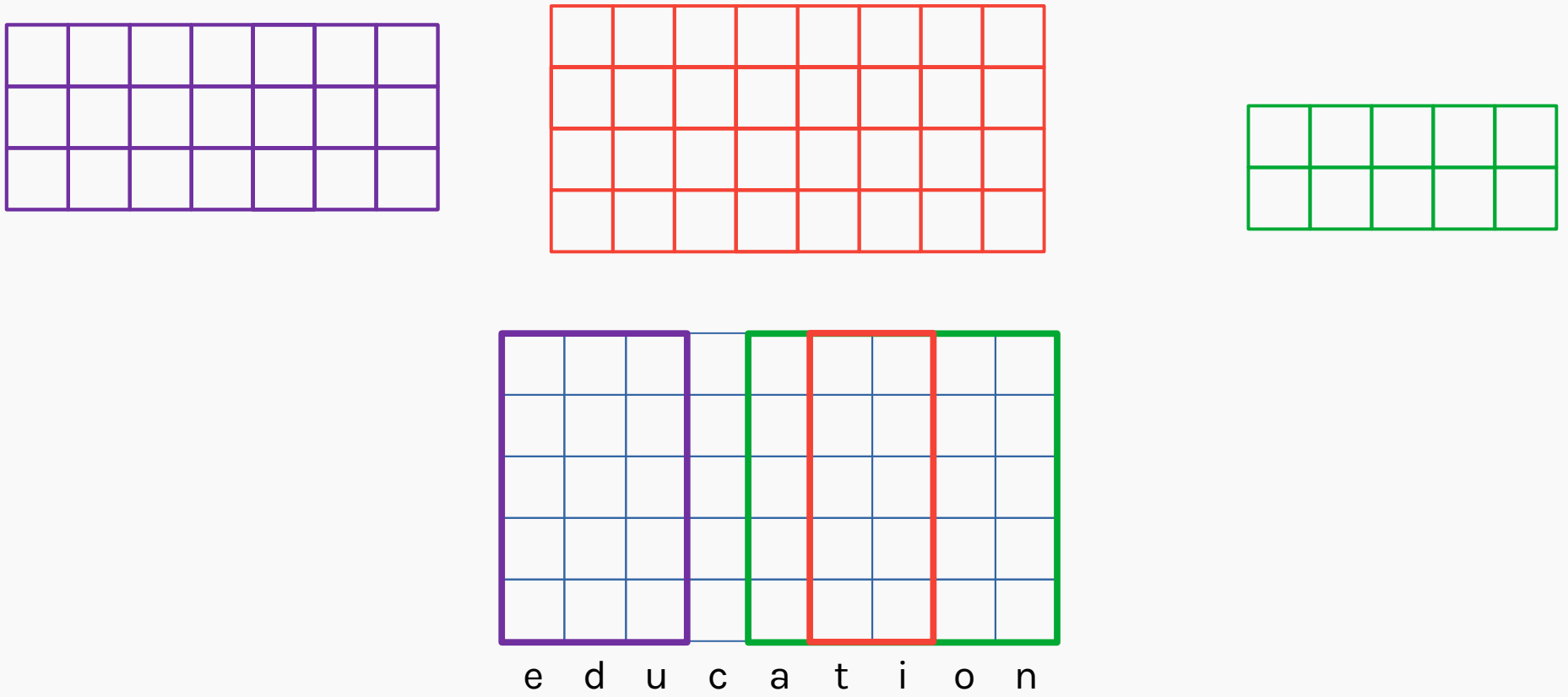


Character CNN

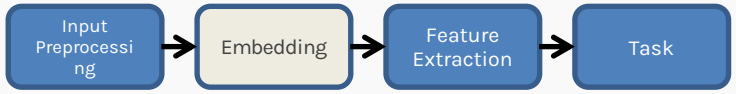


The result is a set of matrices of 3×7 , 4×8 and 2×5 .

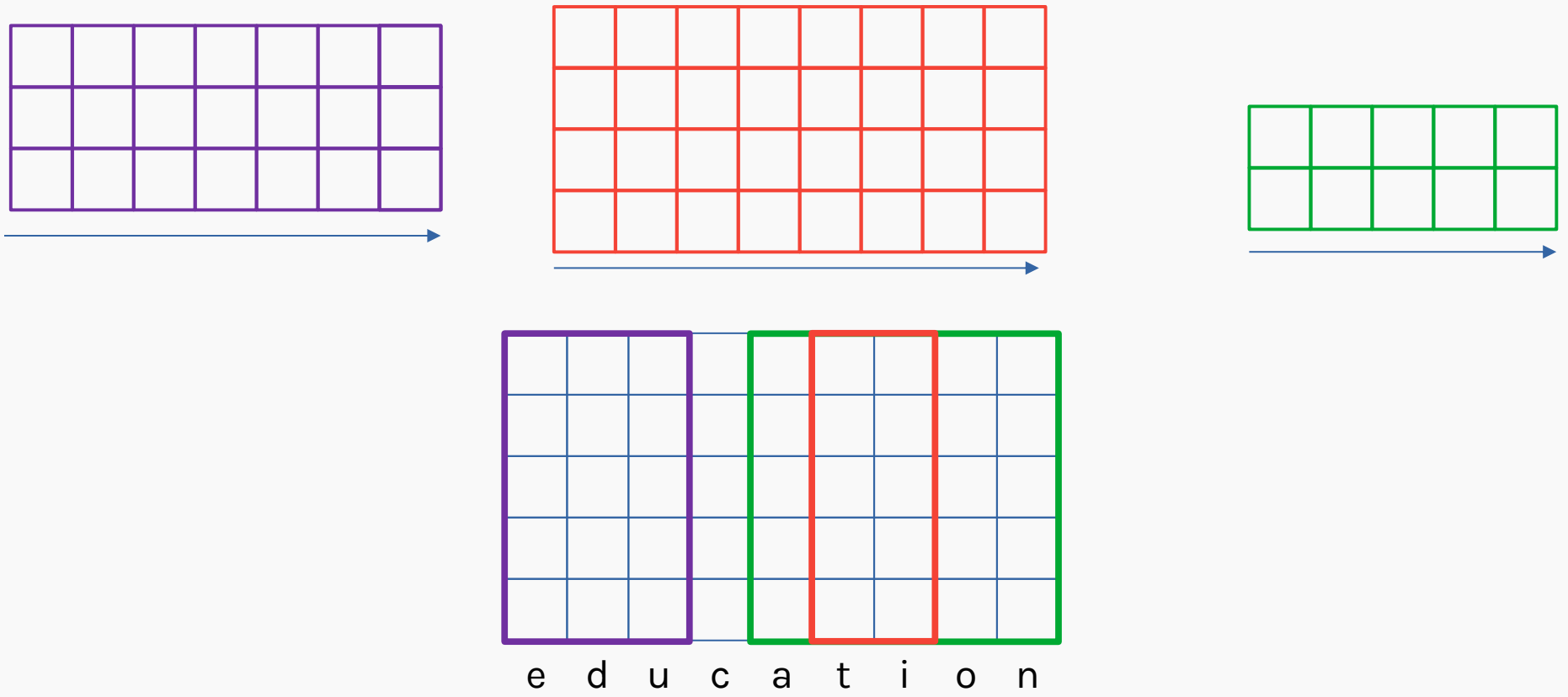
We need to create an embedding of fixed length.



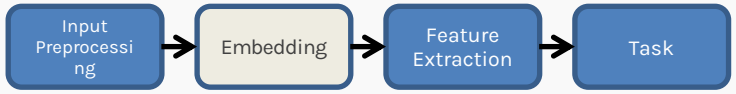
Character CNN



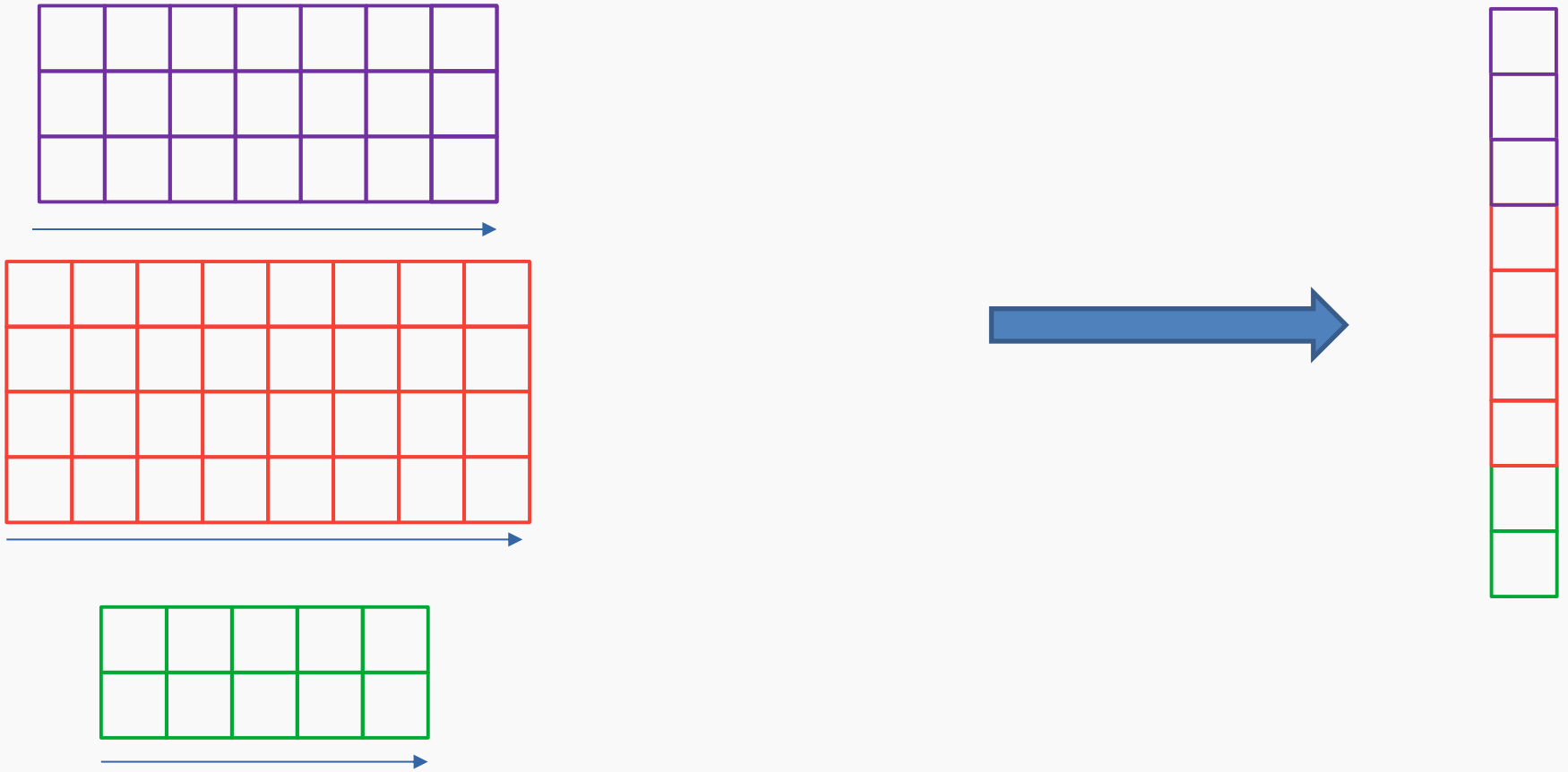
We apply MaxPooling in the temporal direction.
Each word will have the same embedding size.



Character CNN



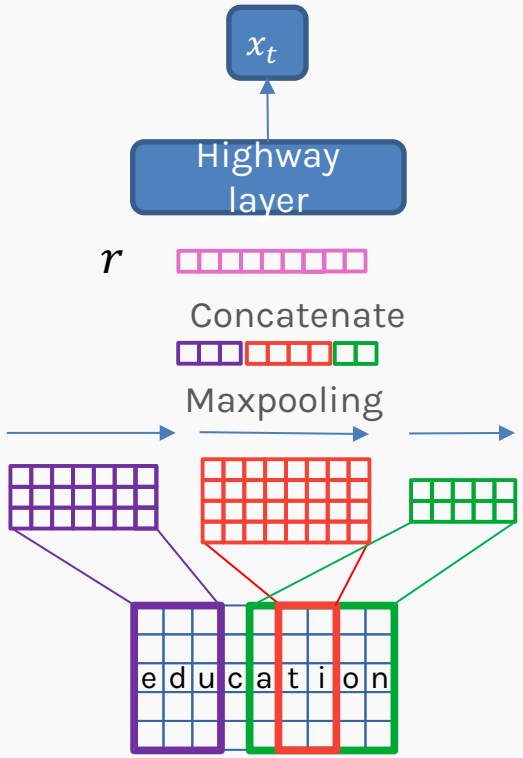
We will get vectors of length 3 , 4 and 2 , that we concatenate to get a 9×1 vector embedding, which we will call $\boldsymbol{r}$.



ELMo embeddings

Our new word embeddings will be the concatenation of all the corresponding hidden states of the network.

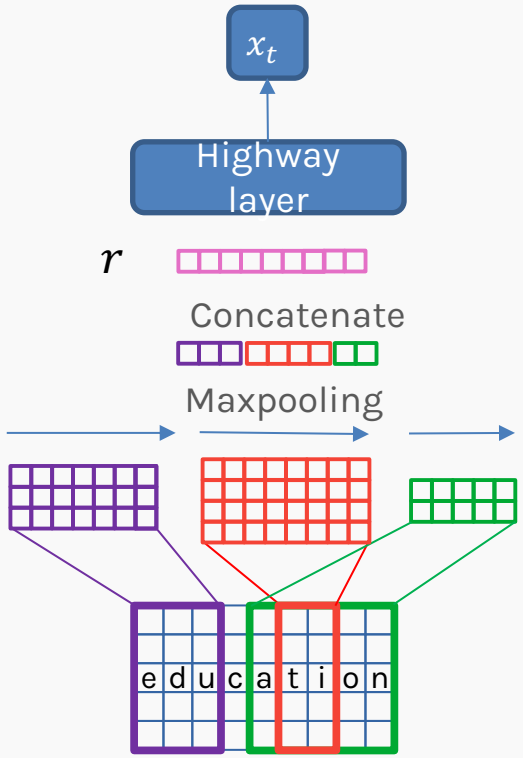
Plus, the character embedding from the Char-CNN.



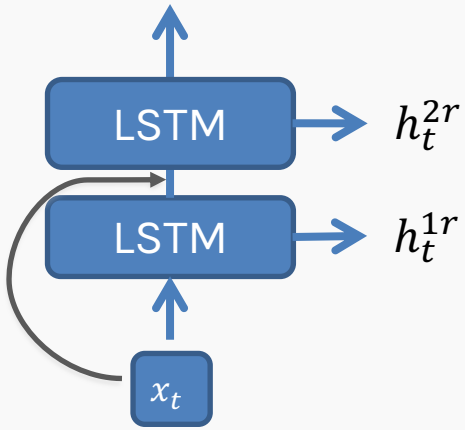
ELMo embeddings

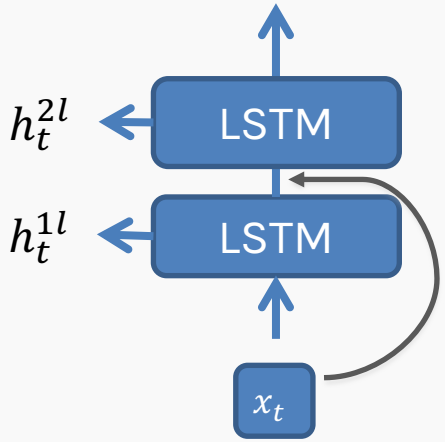
Our new word embeddings will be the concatenation of all the corresponding hidden states of the network.

Plus, the character embedding from the Char-CNN.



$$E_t = \{x_t, h_t^{1l}, h_t^{2l}, h_t^{1r}, h_t^{2r}\}$$

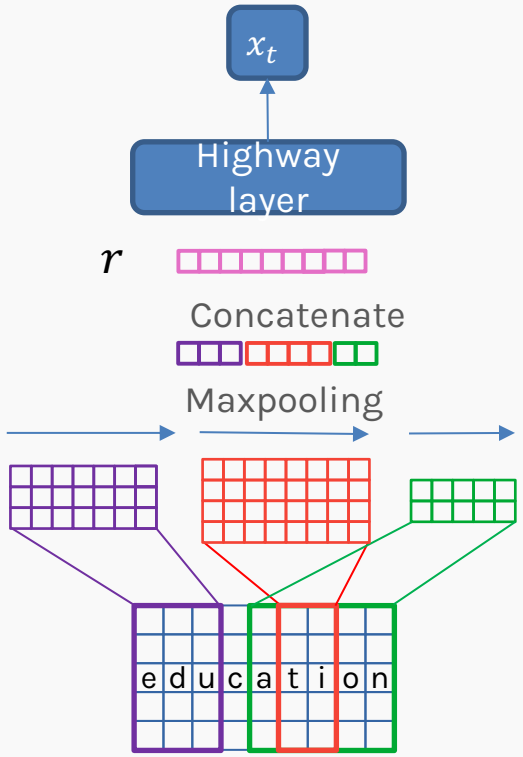


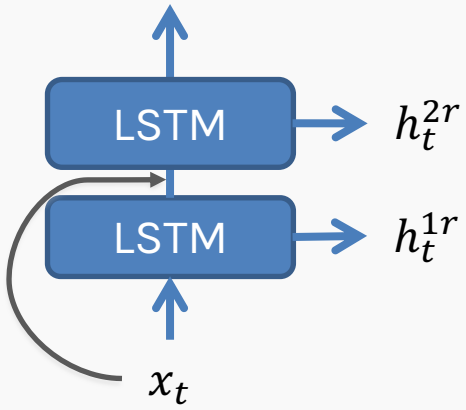


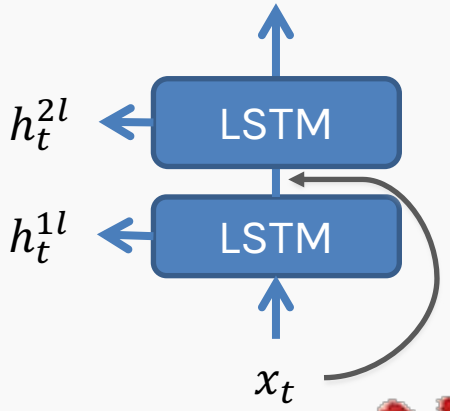
ELMo embeddings

These are the famous "**E**mbeddings from **L**anguage **Mo**dels"

$$E_t = \{x_t, h_t^{1l}, h_t^{2l}, h_t^{1r}, h_t^{2r}\}$$







ELMo embeddings

We can concatenate the embeddings at each recurrent level,

$$h_t^1 = \{h_t^{1l}, h_t^{1r}\}$$
$$h_t^2 = \{h_t^{2l}, h_t^{2r}\}$$

And we rename the Character CNN as

$$h_t^0 = x_t$$

And the new embedding takes the form:

$$E_t = \{h_t^0, h_t^1, h_t^2\}$$

ELMo embeddings

And the new embedding takes the form:

$$E_t = \{h_t^0, h_t^1, h_t^2\}$$

Once trained, we freeze ELMo and use it to extract the multi-level representations, $E_t = \{h_t^0, h_t^1, h_t^2\}$.

For each sentence we can create a context-dependent embedding.

ELMo embeddings: How to use

Now we are ready to replace word2vec.

Why are we so eager to replace something that works?

I can play guitar pretty well.

They went to play in the park.

Romeo and Juliet is a tragic play.

ELMo embeddings: How to use

Word2vec is context-independent.

Which means that the same word used in different contexts will have the same embedding.

I can play guitar pretty well.

They went to play in the park.

Romeo and Juliet is a tragic play.

We must learn all the different meanings

ELMo embeddings: How to use

If I want to use these embeddings as an input.

Do I have to change my model entirely?

ELMo embeddings: How to use

If I want to use these embeddings as an input.

Do I have to change my model entirely?

Not at all!

ELMo embeddings: How to use

We use the embeddings as our sauce.

We concatenate the ELMo embeddings E_t to the ones we use to feed the model for our other task.

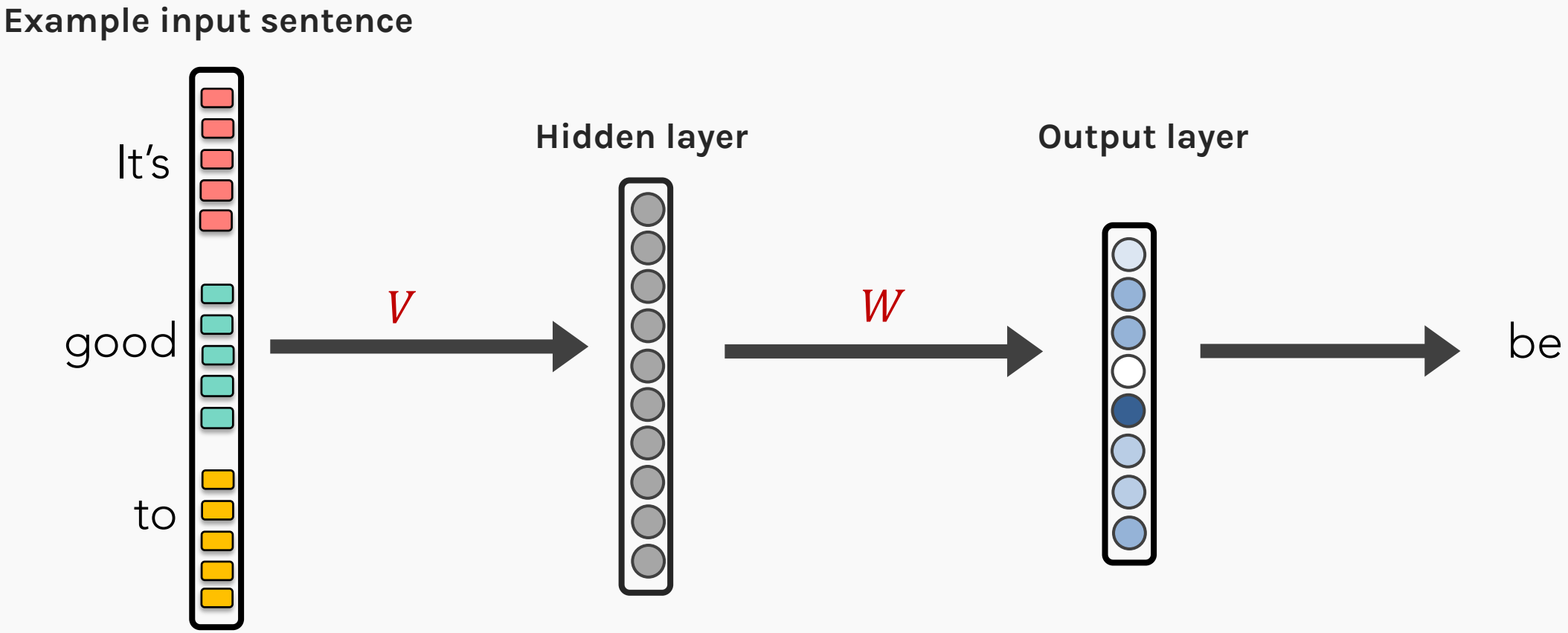
$$e'_t = \{e_t, E_t\}$$

Any other embedding

ELMo embeddings

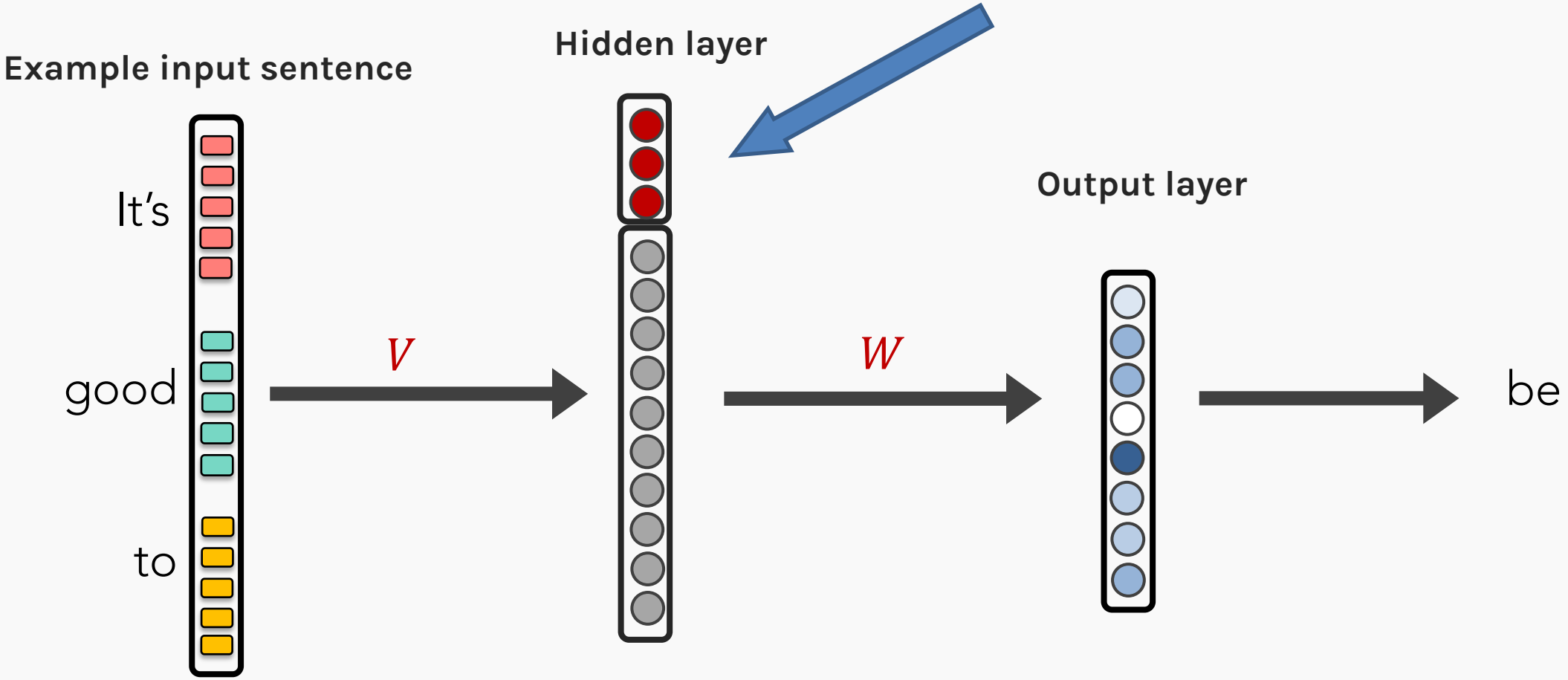
ELMo embeddings: How to use

Going back to a previous example,



ELMo embeddings: How to use

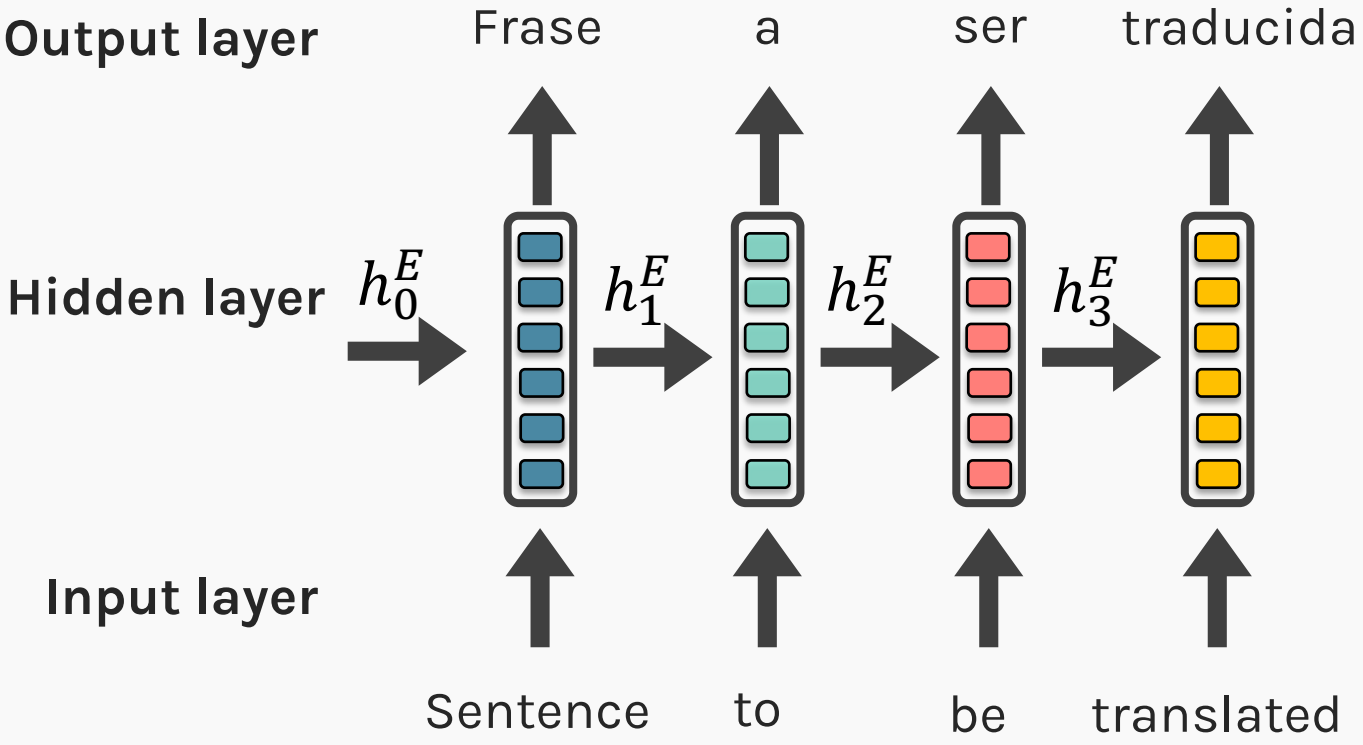
Going back to a previous example, we put the sauce here.



Seq2Seq (with Attention)

Sequence-to-Sequence (seq2seq)

- If our input is a sentence in Language A, and we wish to translate it to Language B, it is clearly sub-optimal to translate word by word (like our current models are suited to do).

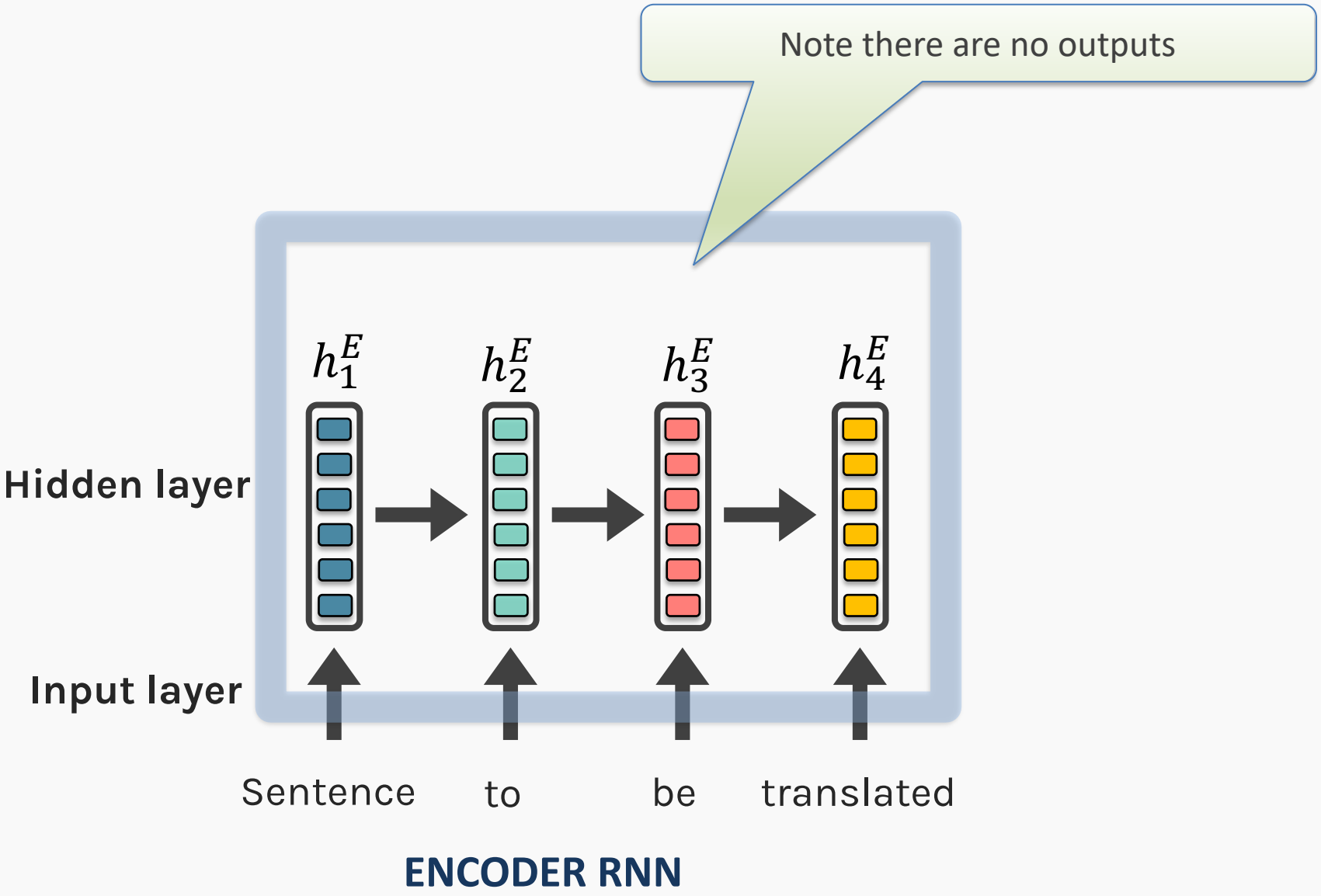


$$h_j^E = \tanh\left(Vx_j^E + Uh_{j-1}^E + b^E\right)$$
$$y_j^E = softmax(h_j^E)$$

Sequence-to-Sequence (seq2seq)

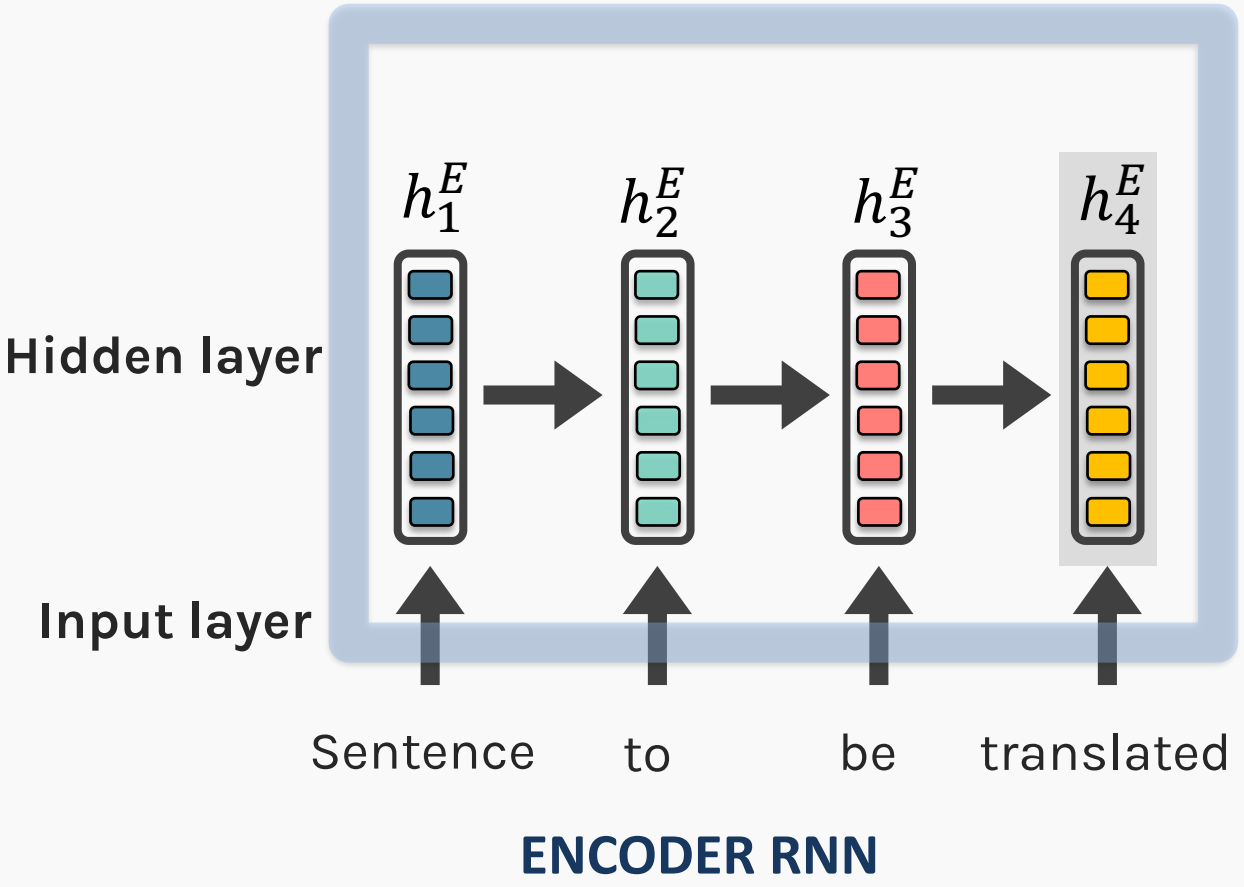
- Instead, let a ***sequence*** of tokens be the unit that we ultimately wish to work with (a sequence of length **N** may emit a sequences of length **M**)
- Seq2seq models are comprised of **2 RNNs**: 1 encoder, 1 decoder

Sequence-to-Sequence (seq2seq)



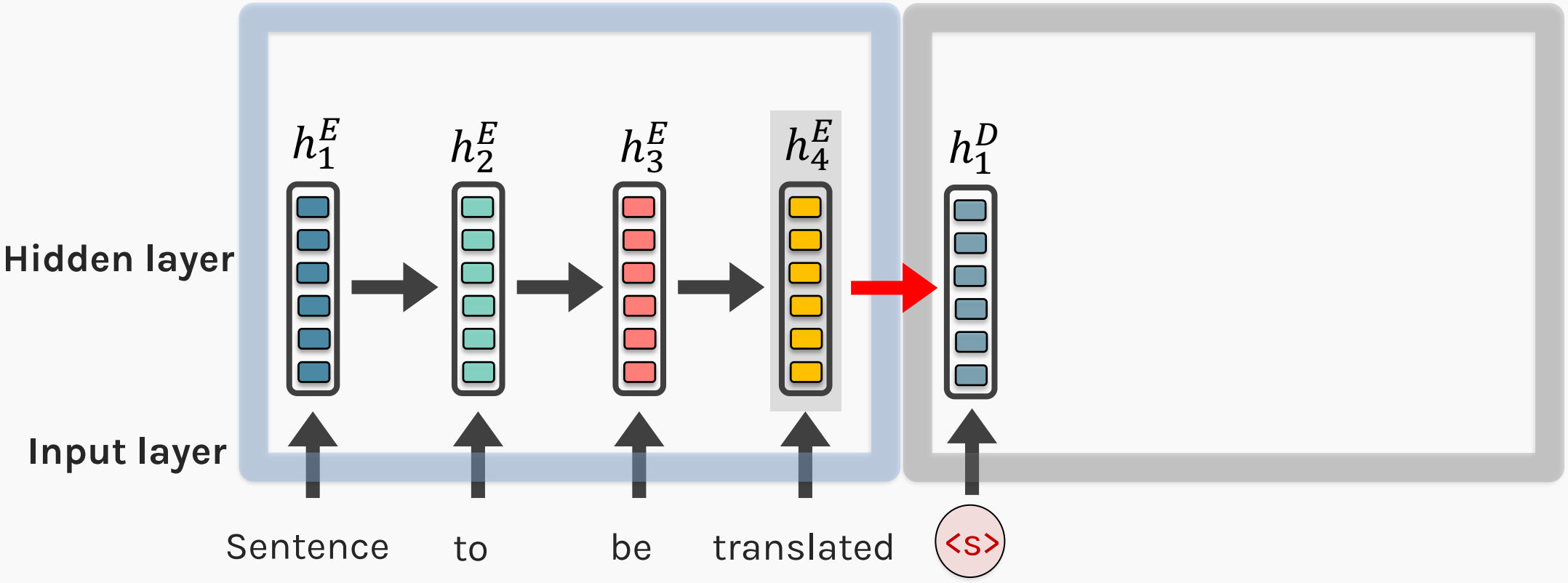
Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN is the initial state of the decoder RNN



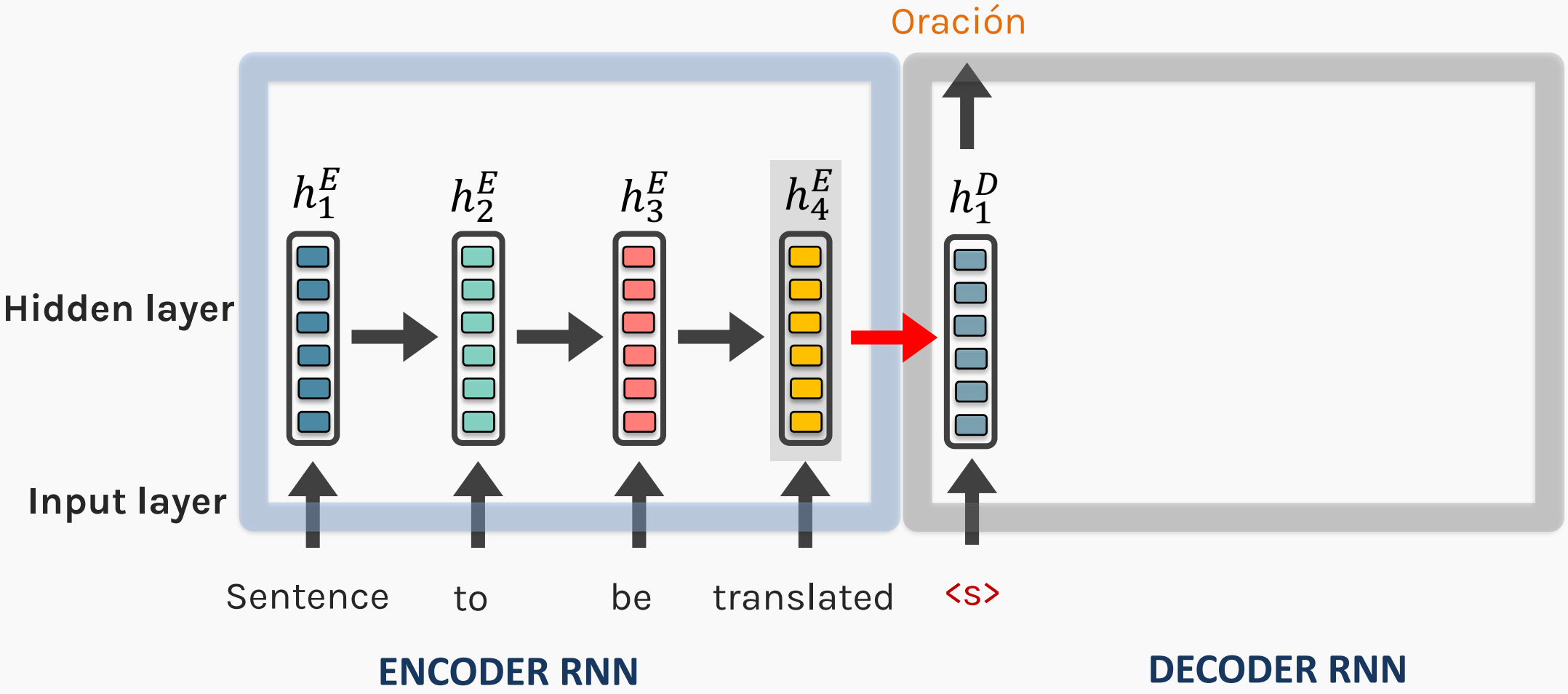
Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN is the initial state of the decoder RNN

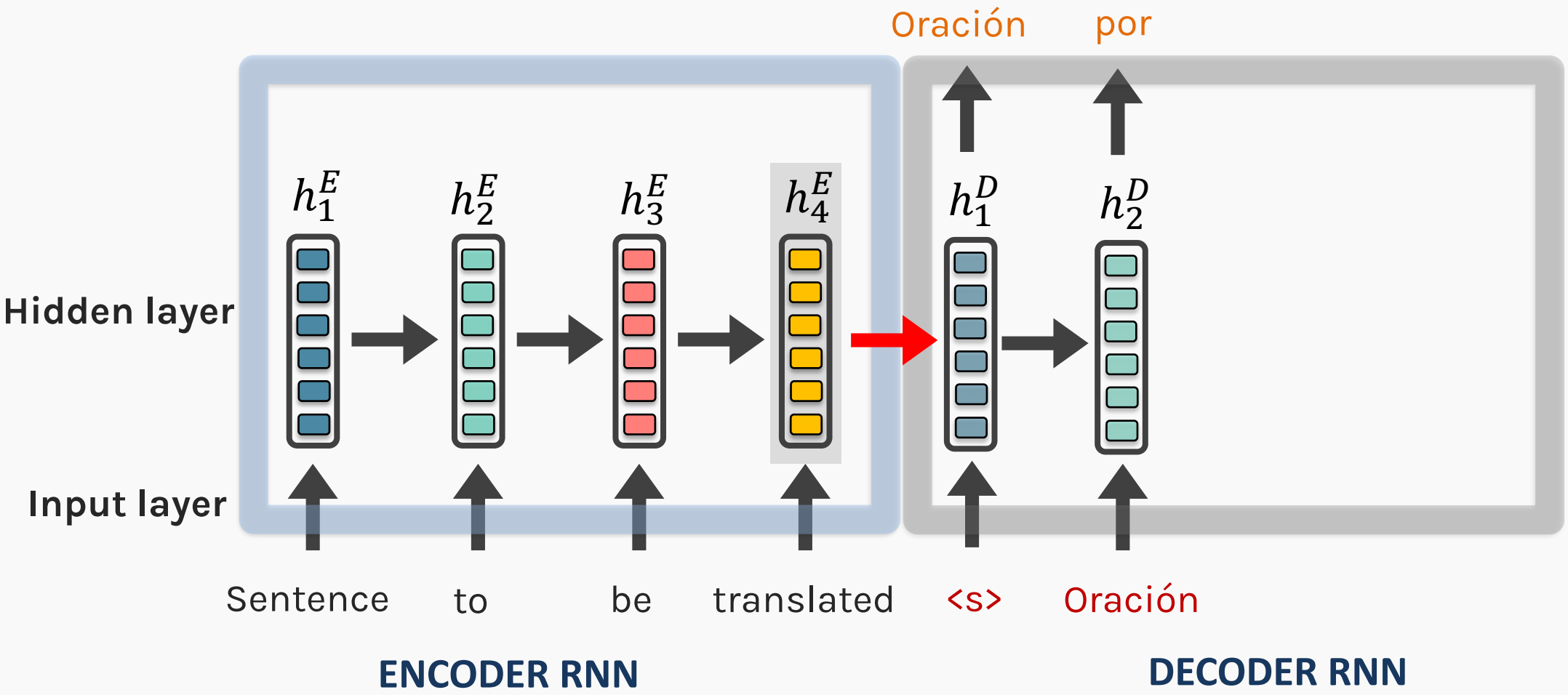


Sequence-to-Sequence (seq2seq)

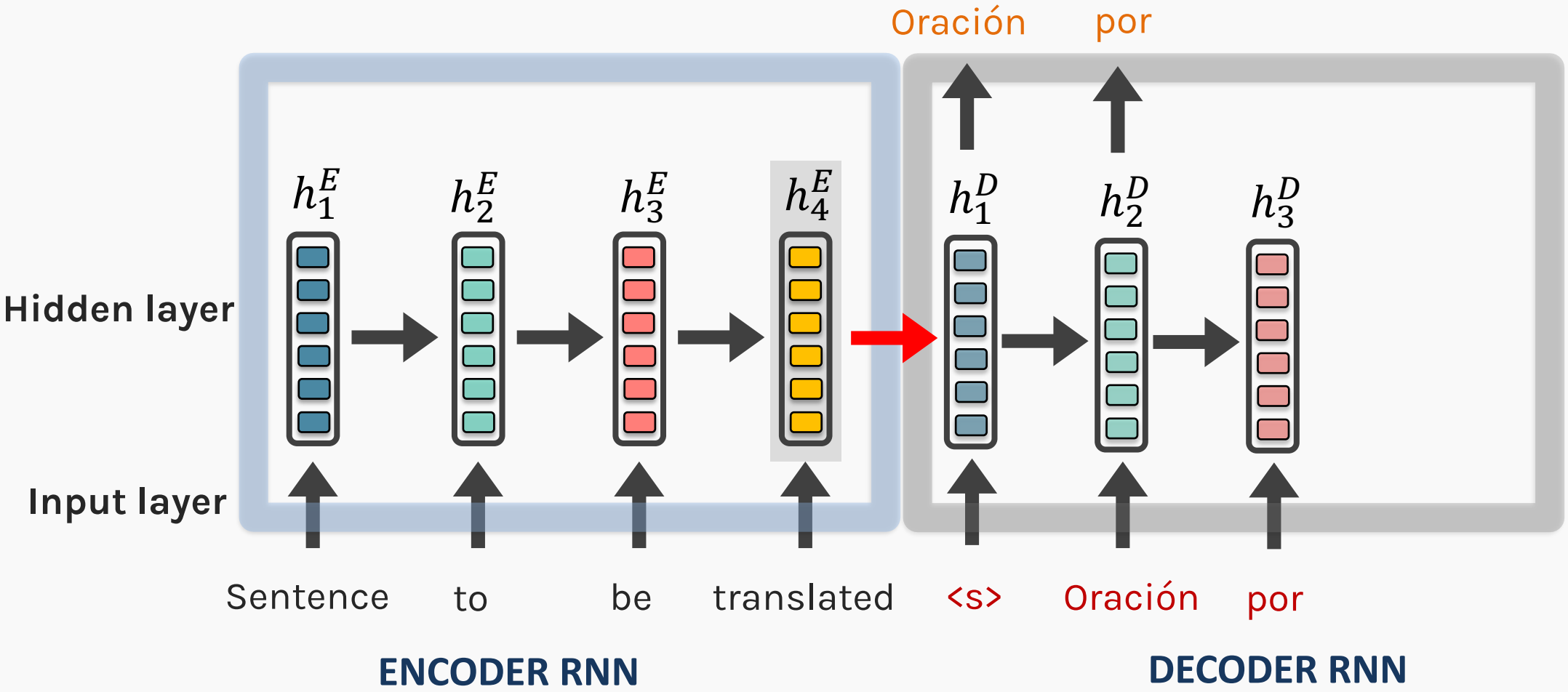
The final hidden state of the encoder RNN is the initial state of the decoder RNN



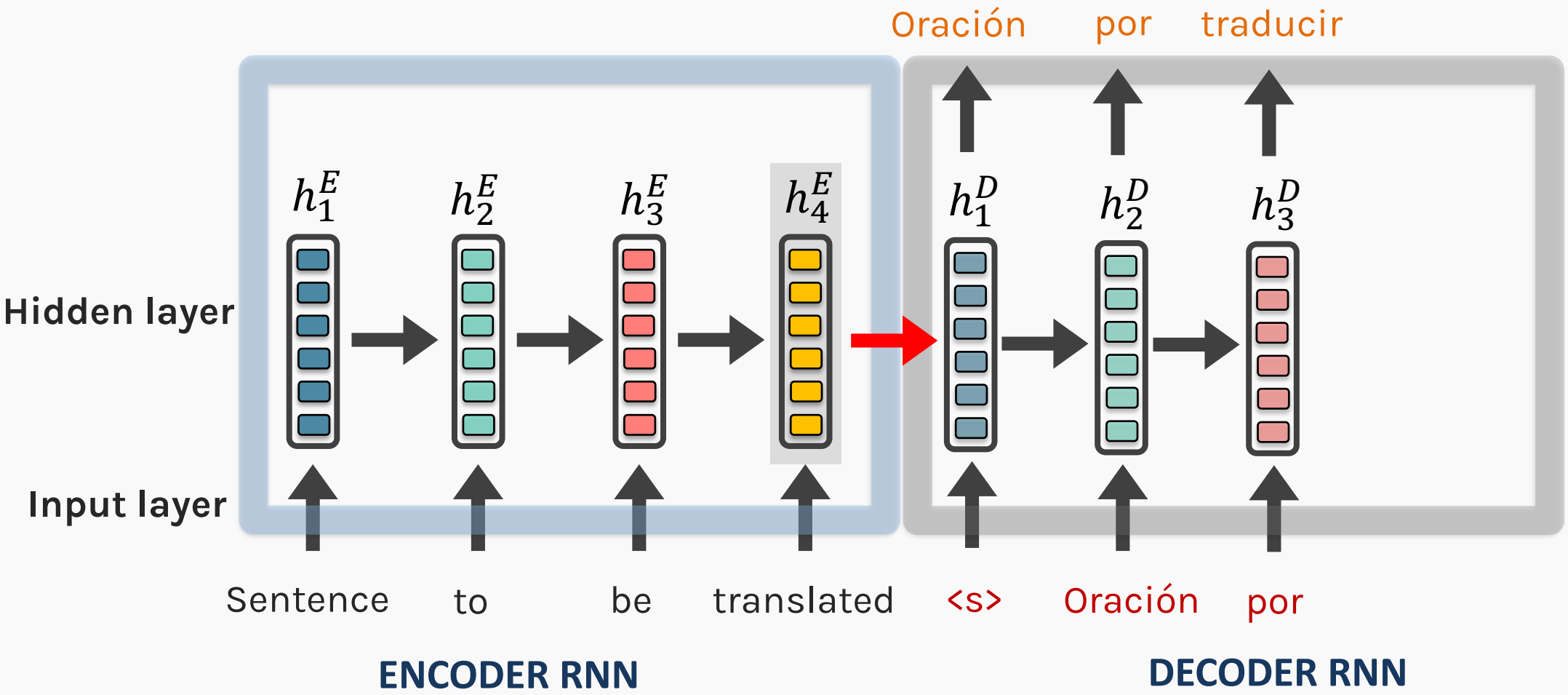
Sequence-to-Sequence (seq2seq)



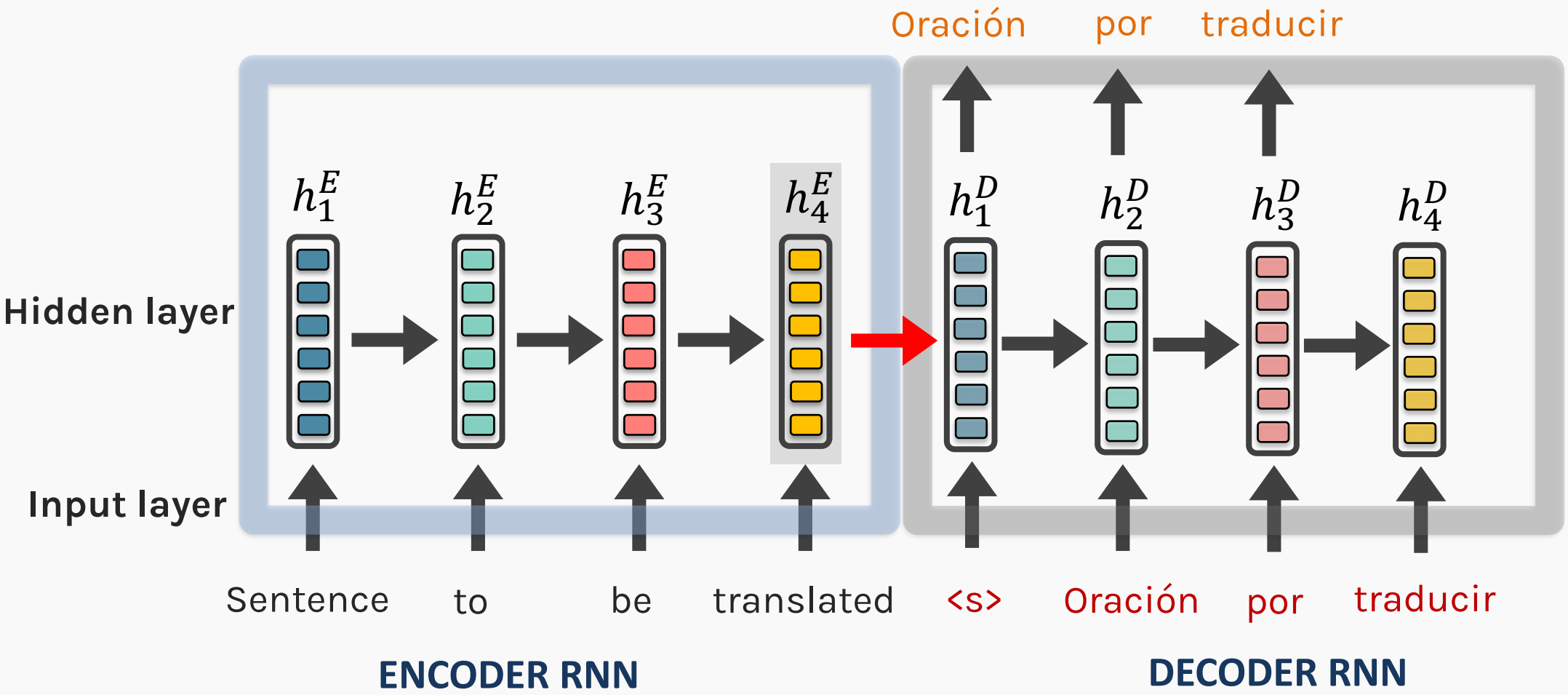
Sequence-to-Sequence (seq2seq)



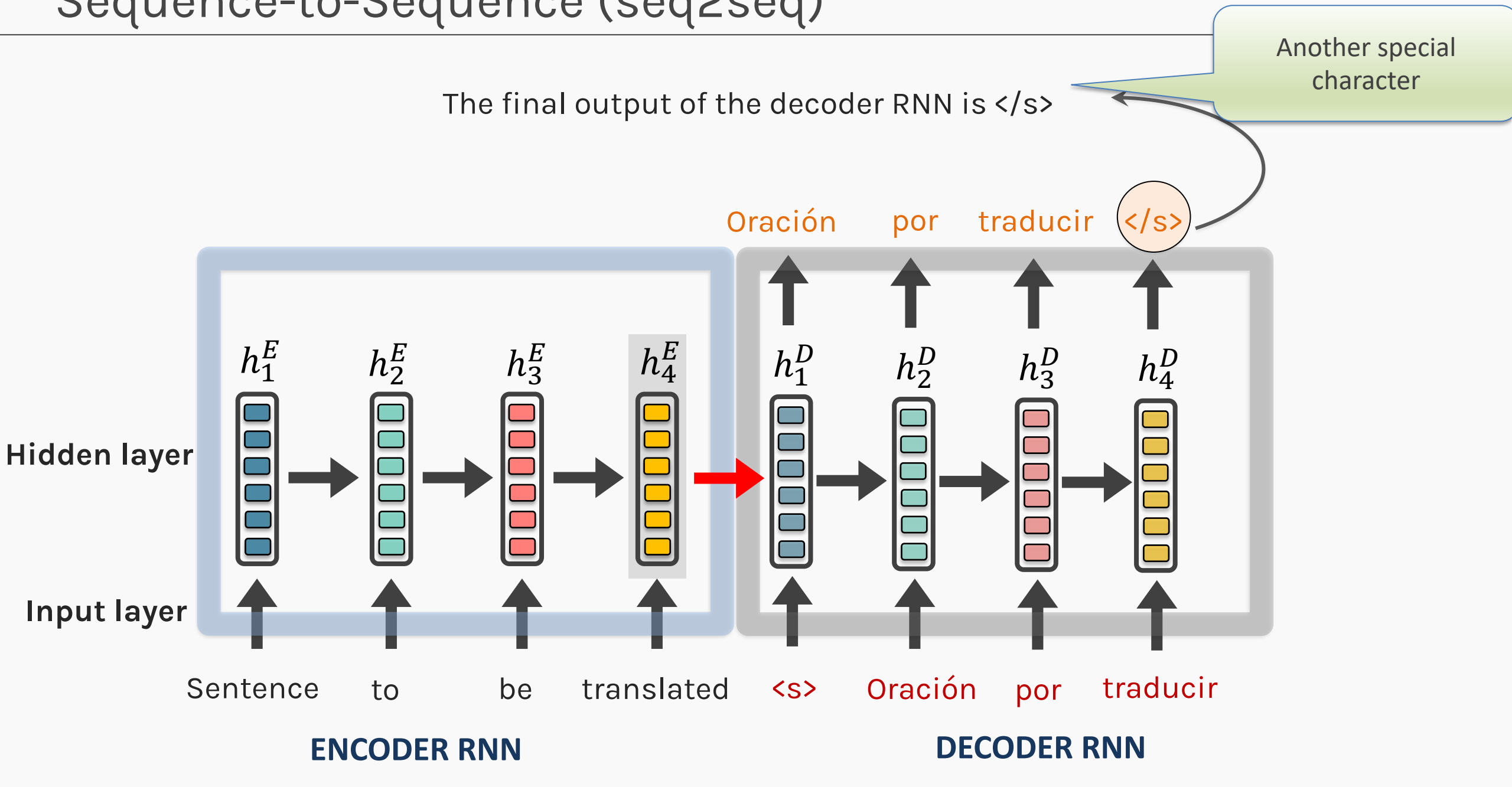
Sequence-to-Sequence (seq2seq)



Sequence-to-Sequence (seq2seq)

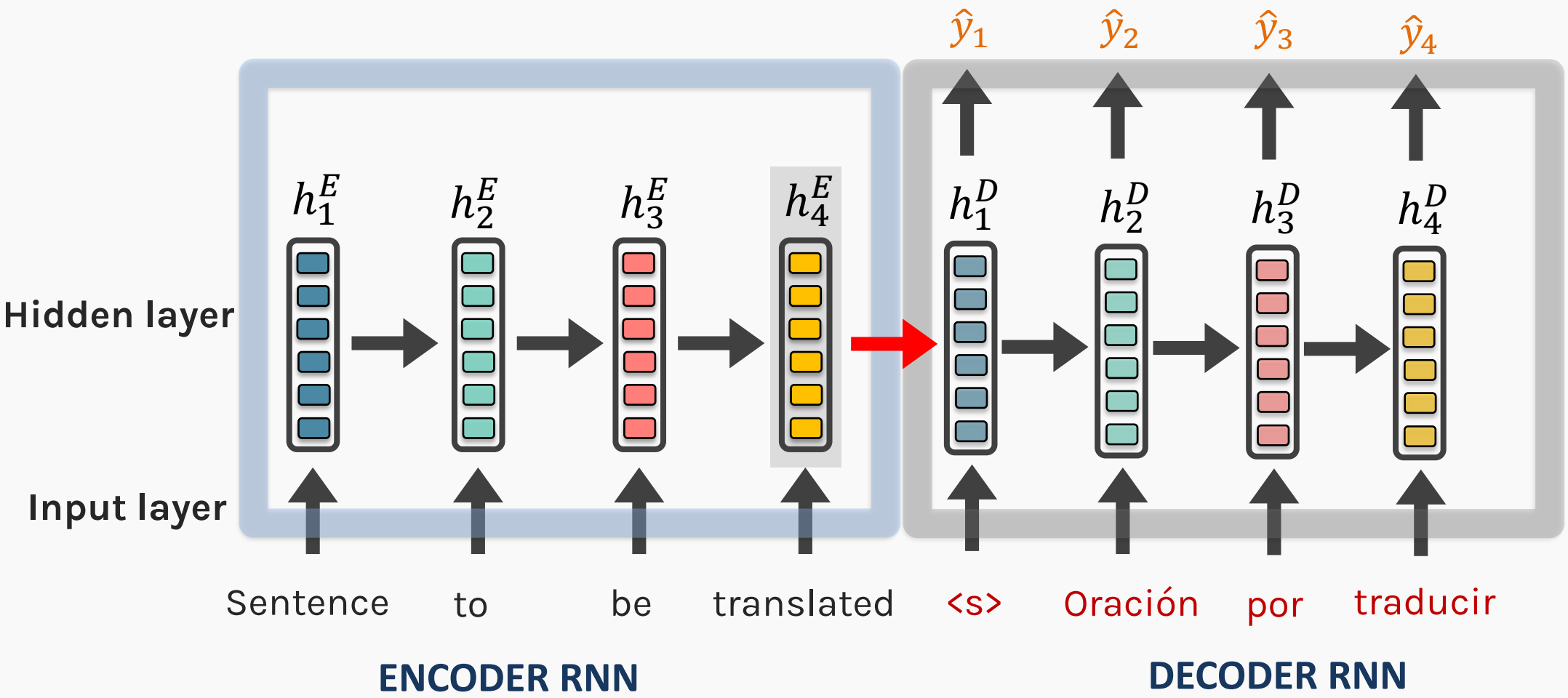


Sequence-to-Sequence (seq2seq)



Sequence-to-Sequence (seq2seq): Training

Training occurs like RNNs typically do. Decoder generates outputs one word at a time, until we generate the </s> token.



Sequence-to-Sequence (seq2seq)

See any issues with this traditional **seq2seq** paradigm?

Sequence-to-Sequence (seq2seq)

It's crazy that the entire "meaning" of the 1st sequence is expected to be packed into one embedding, and that the encoder then never interacts w/ the decoder again. Hands free!

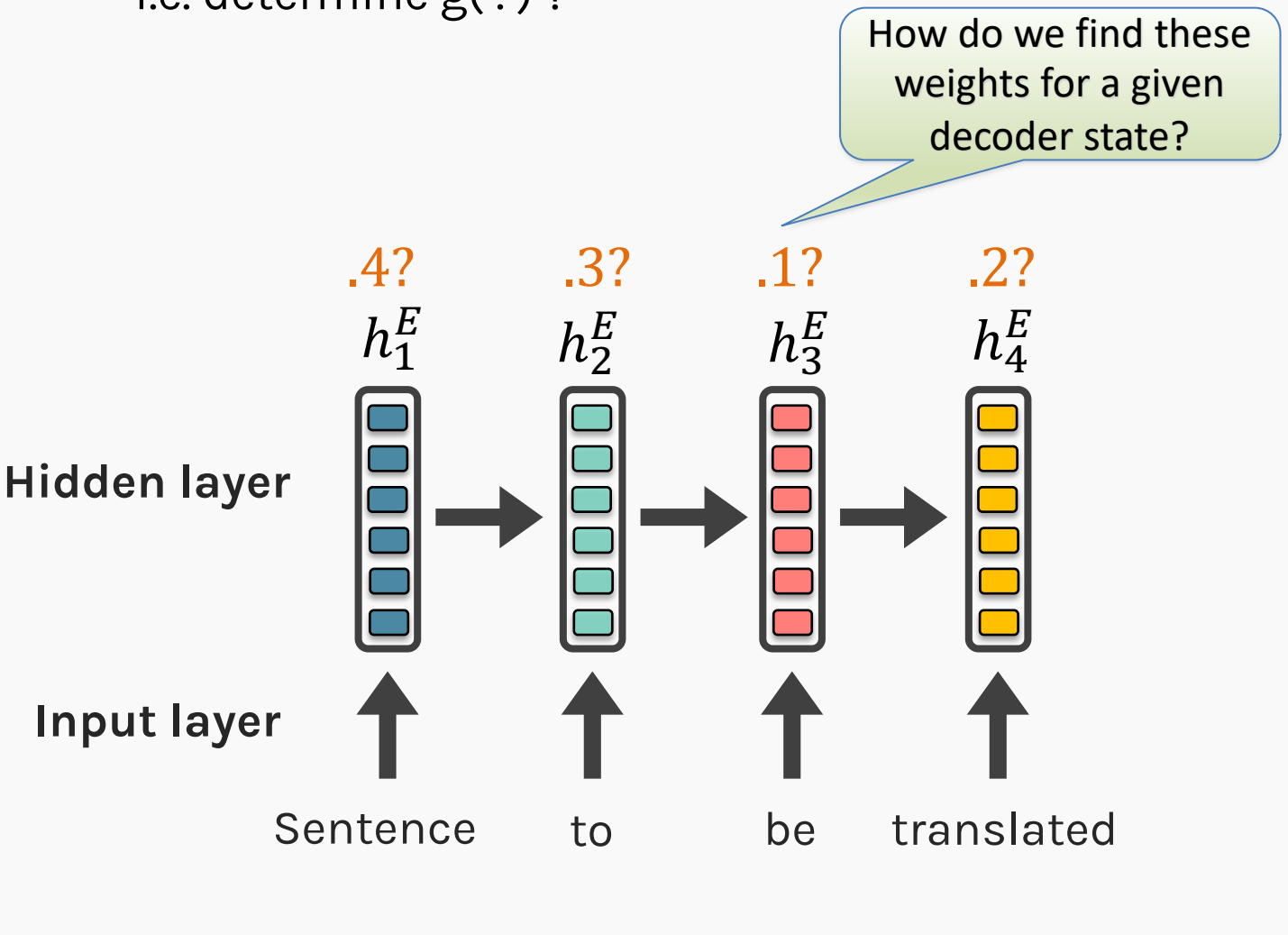
What other alternatives can we have?

Seq2Seq + Attention

Q: How do we determine how much attention to pay to each of the encoder's hidden states i.e. determine $g(\,.\,)$?

Seq2Seq + Attention

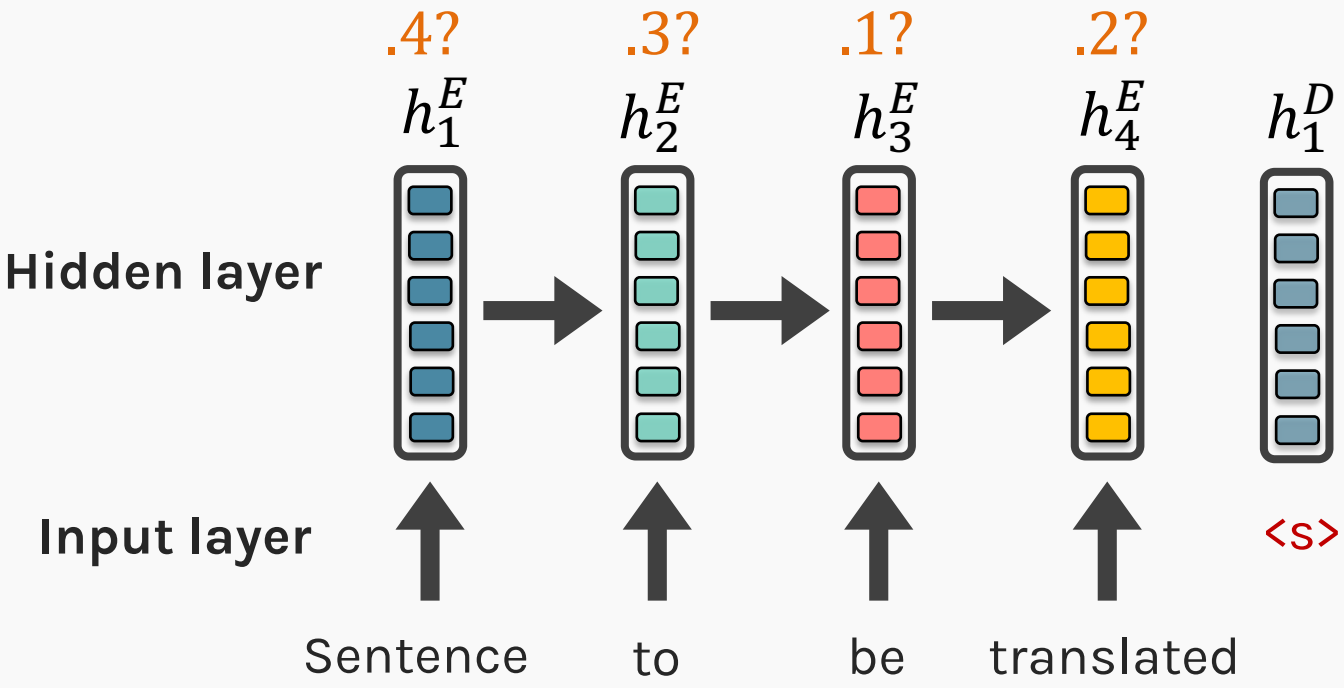
Q: How do we determine how much attention to pay to each of the encoder's hidden states i.e. determine g(.) ?



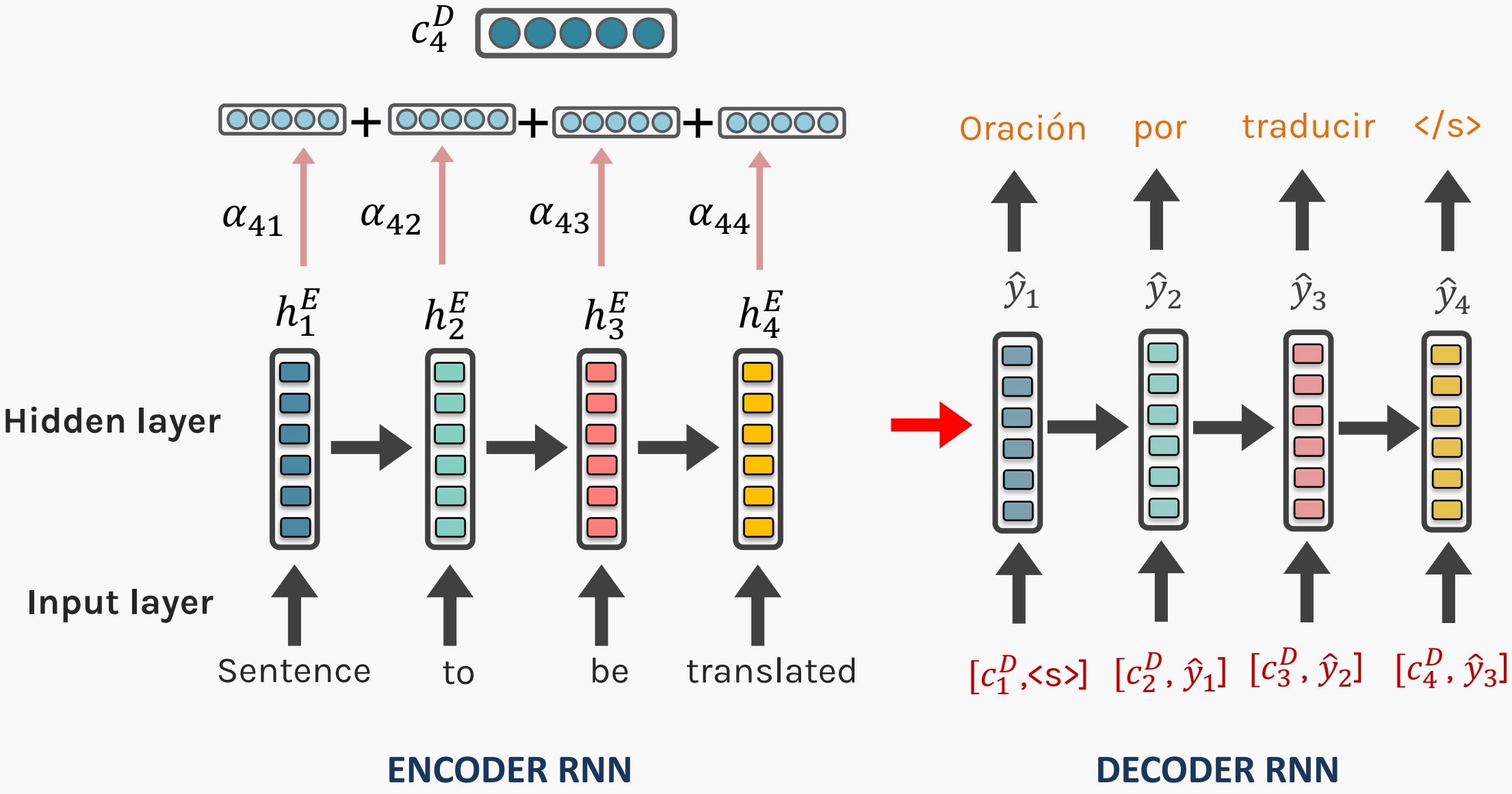
Seq2Seq + Attention

Q: How do we determine how much attention to pay to each of the encoder's hidden states i.e. determine g(.) ?

A: Let's base it on our decoder's previous hidden state (our latest representation of meaning) and all of the encoder's hidden states!



Seq2Seq + Attention



Seq2Seq + Attention

Attention:

- greatly improves seq2seq results
- allows us to visualize the contribution each word gave during each step of the decoder

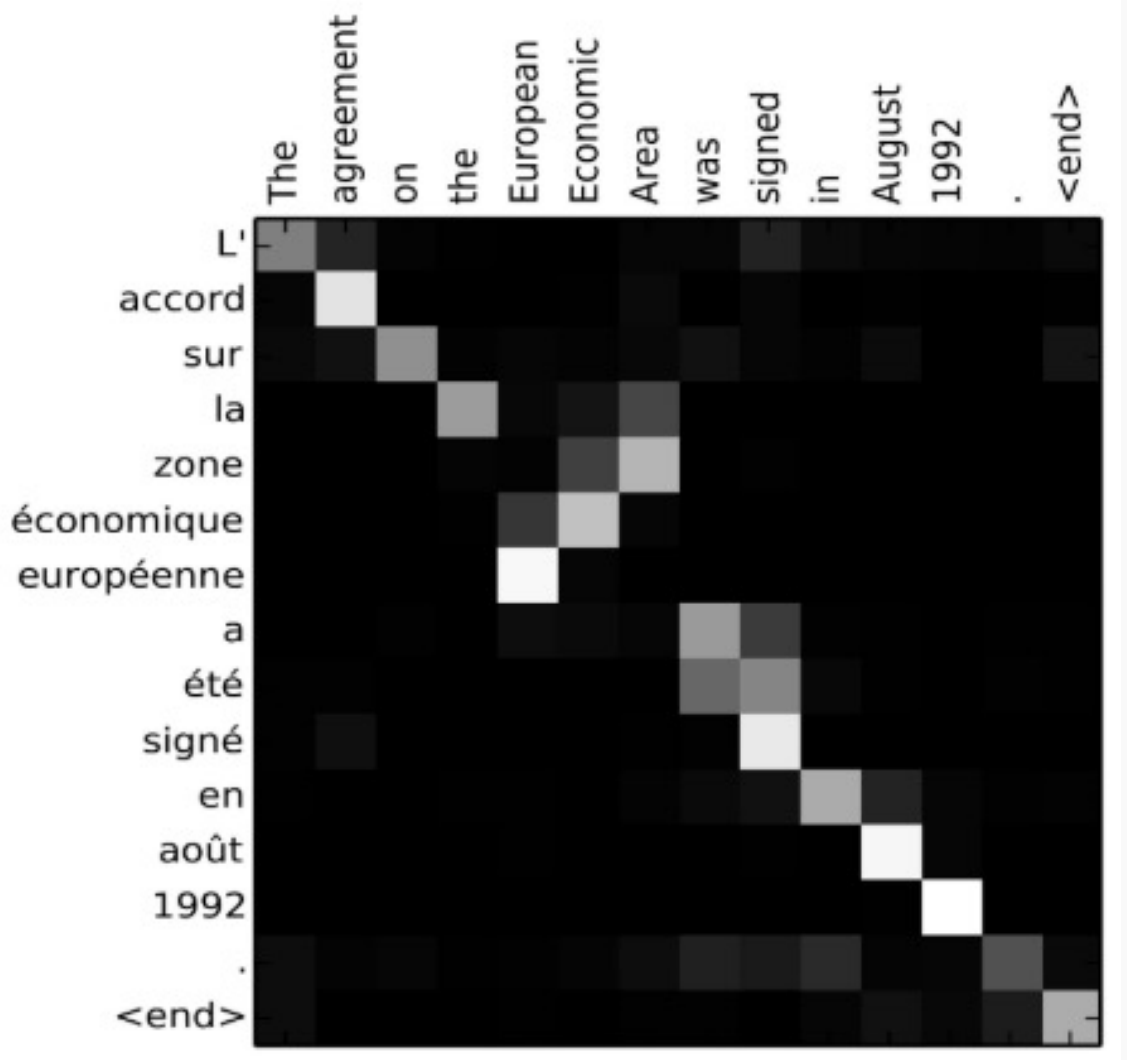
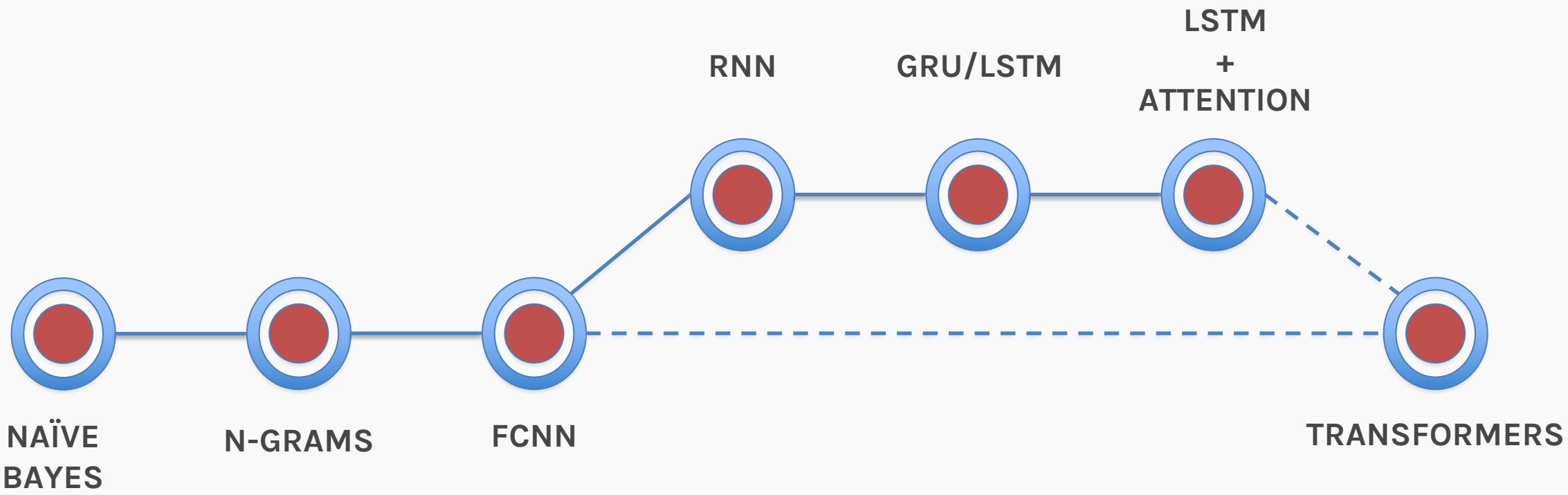


Image source: Fig 3 in [Bahdanau et al., 2015]

ATTENTION IS ALL YOU NEED

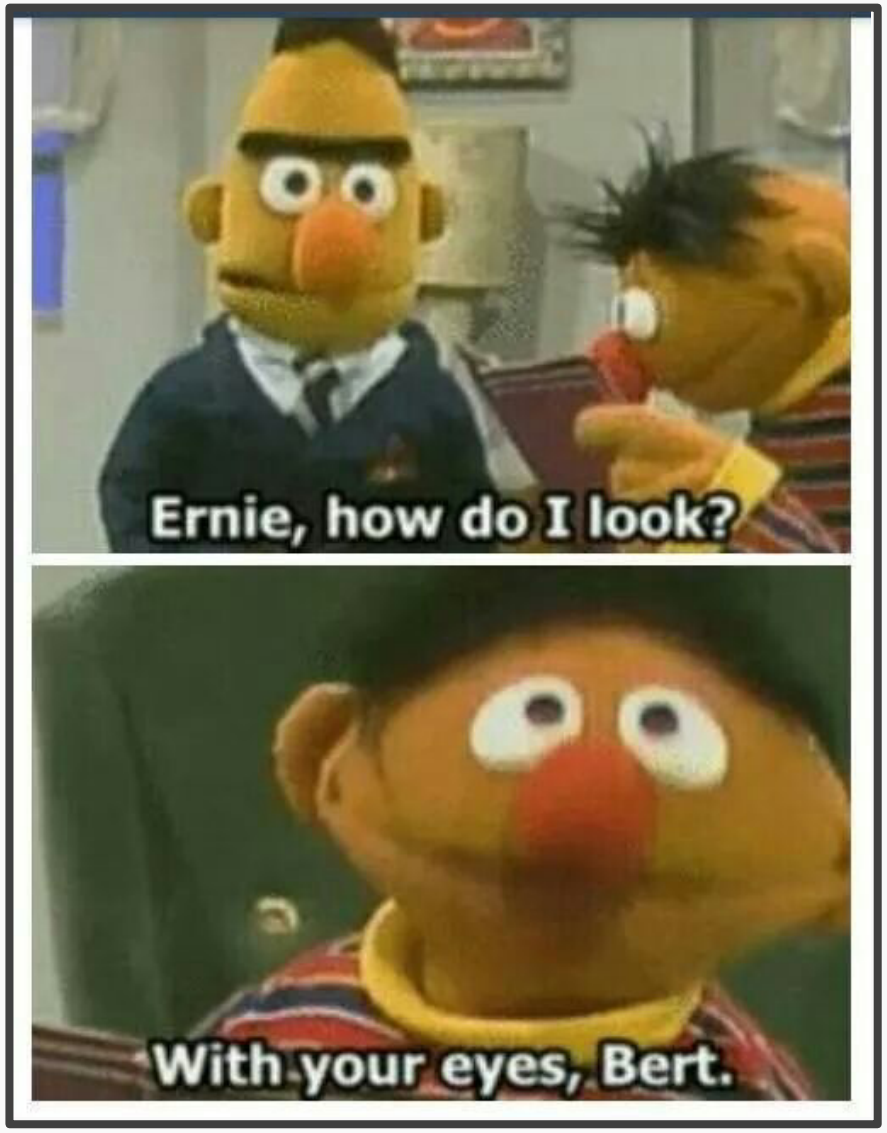
Language modeling



What we want

Language Model Wishlist?

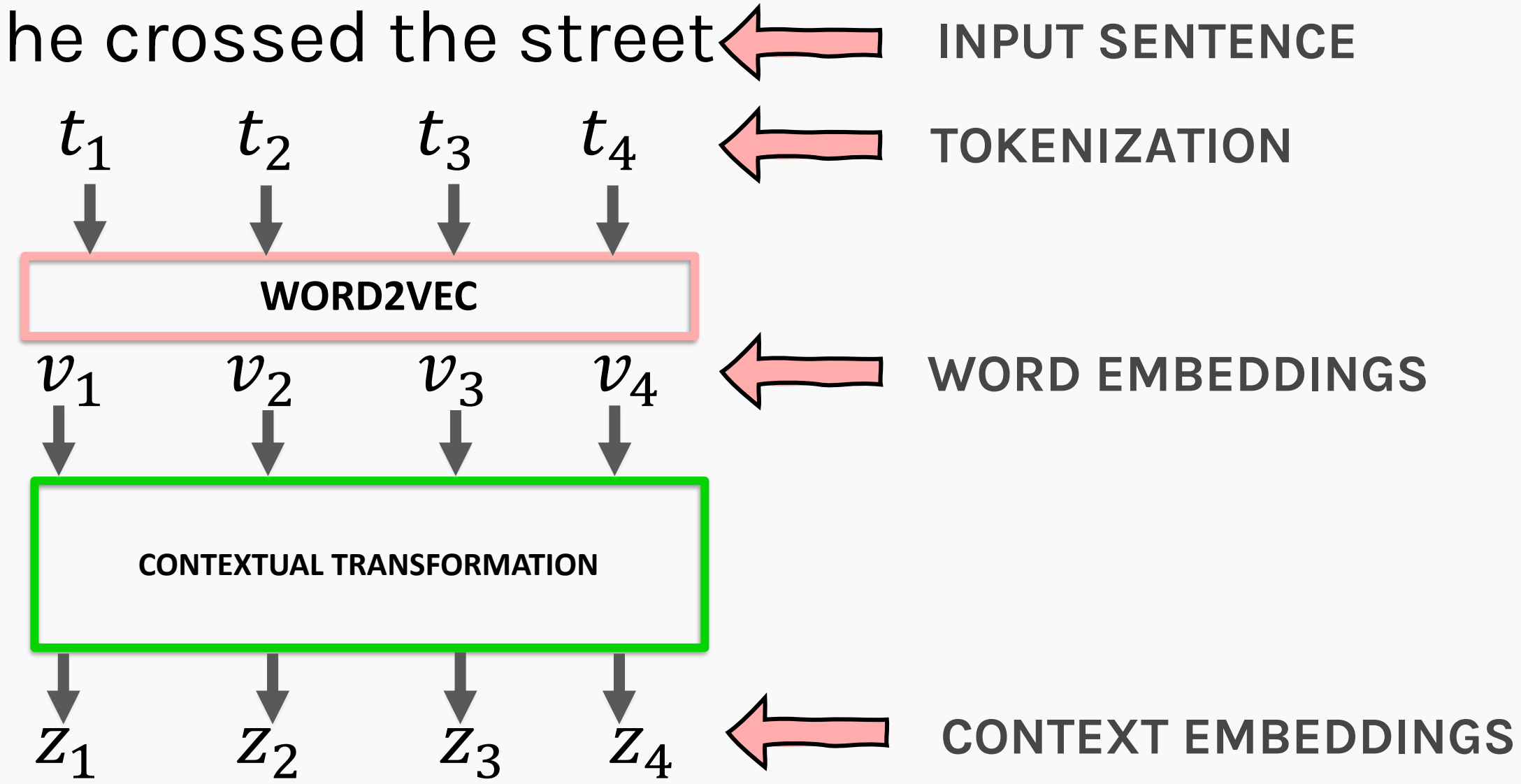
- We want to have strong contextual relations between words
- We want words to have sequential information
- We need an architecture that can be trained in parallel (non-Markovian property)



We’ve already seen an approach of relative importance

Attention

Attention – Where to add weights?



Attention – The basics

- We can still use the idea of transforming the word embeddings to get more context
- However, the transformation matrix *A* must place some importance to the relative importance of words

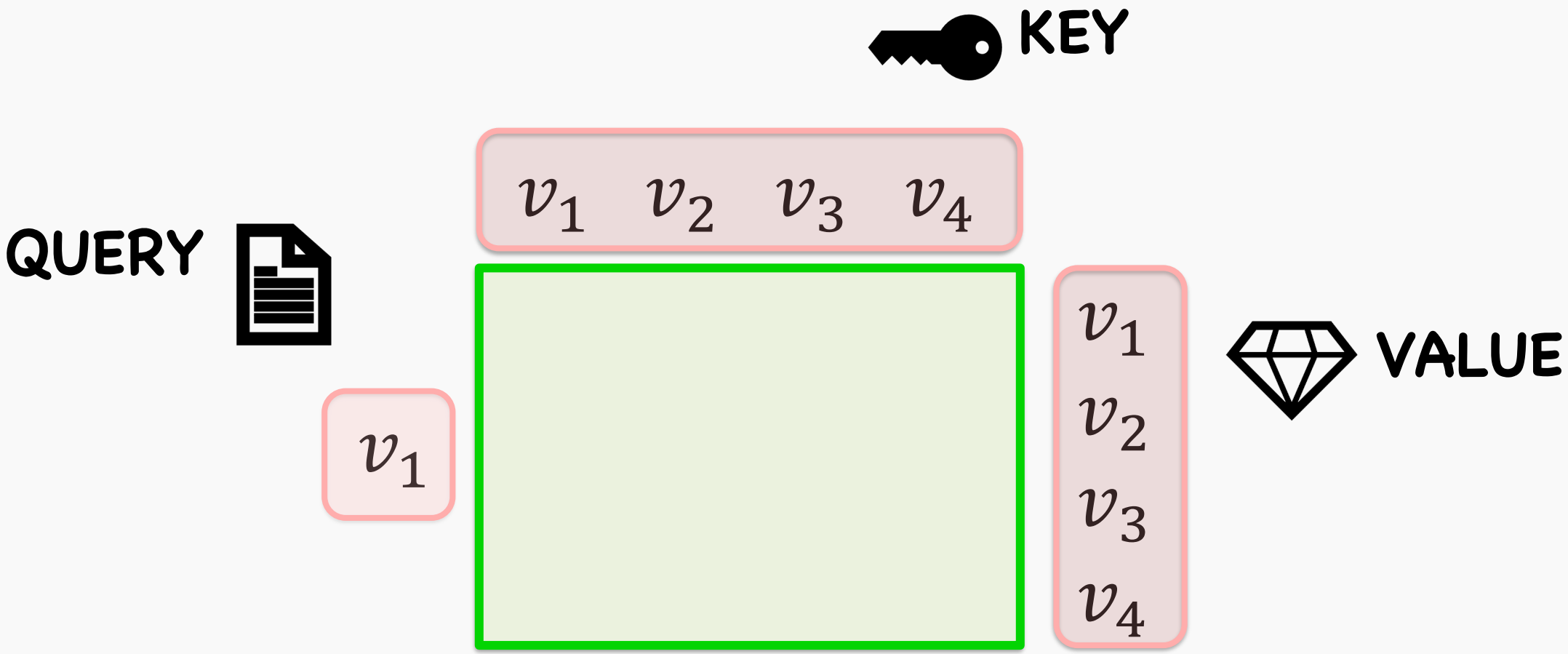
$$y = [y_1, y_2, y_3, \cdots, y_n]$$

$$z = A\, y^T$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \dots & \dots & a_{ij} & \dots & a_{in} \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}$$

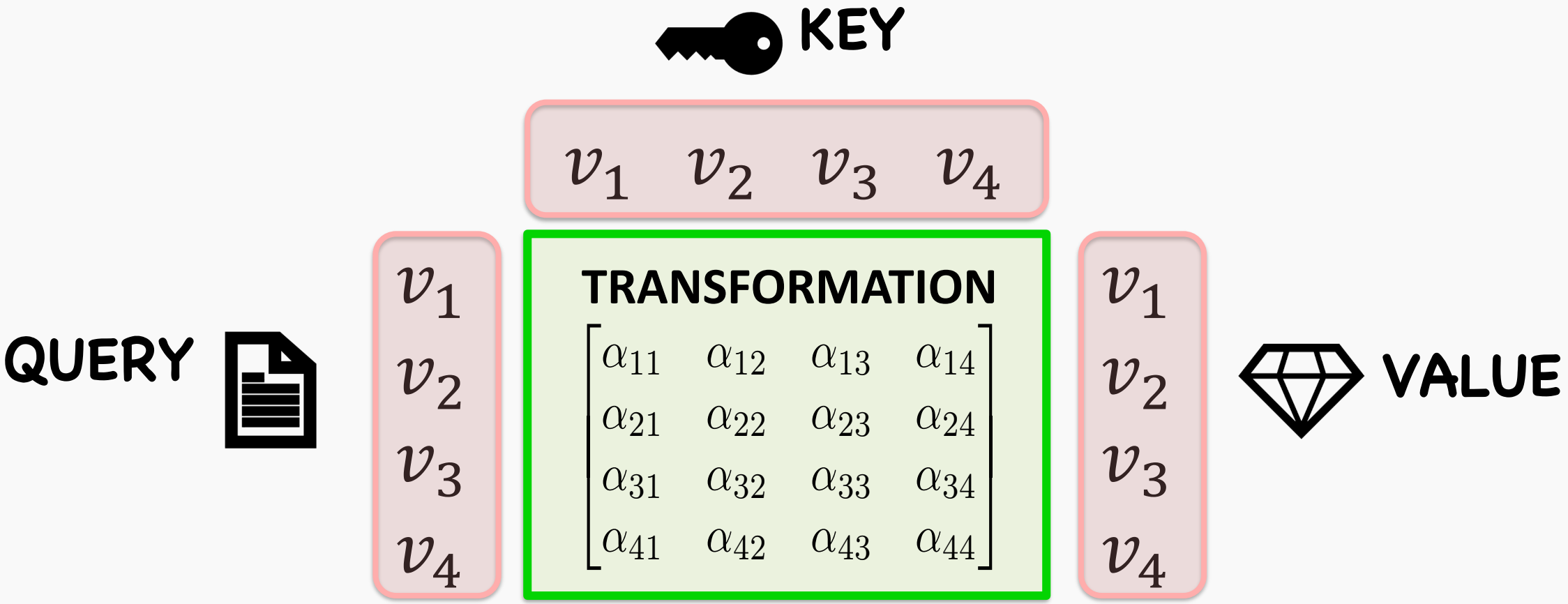
a_{ij} must account for relative importance between word i & j

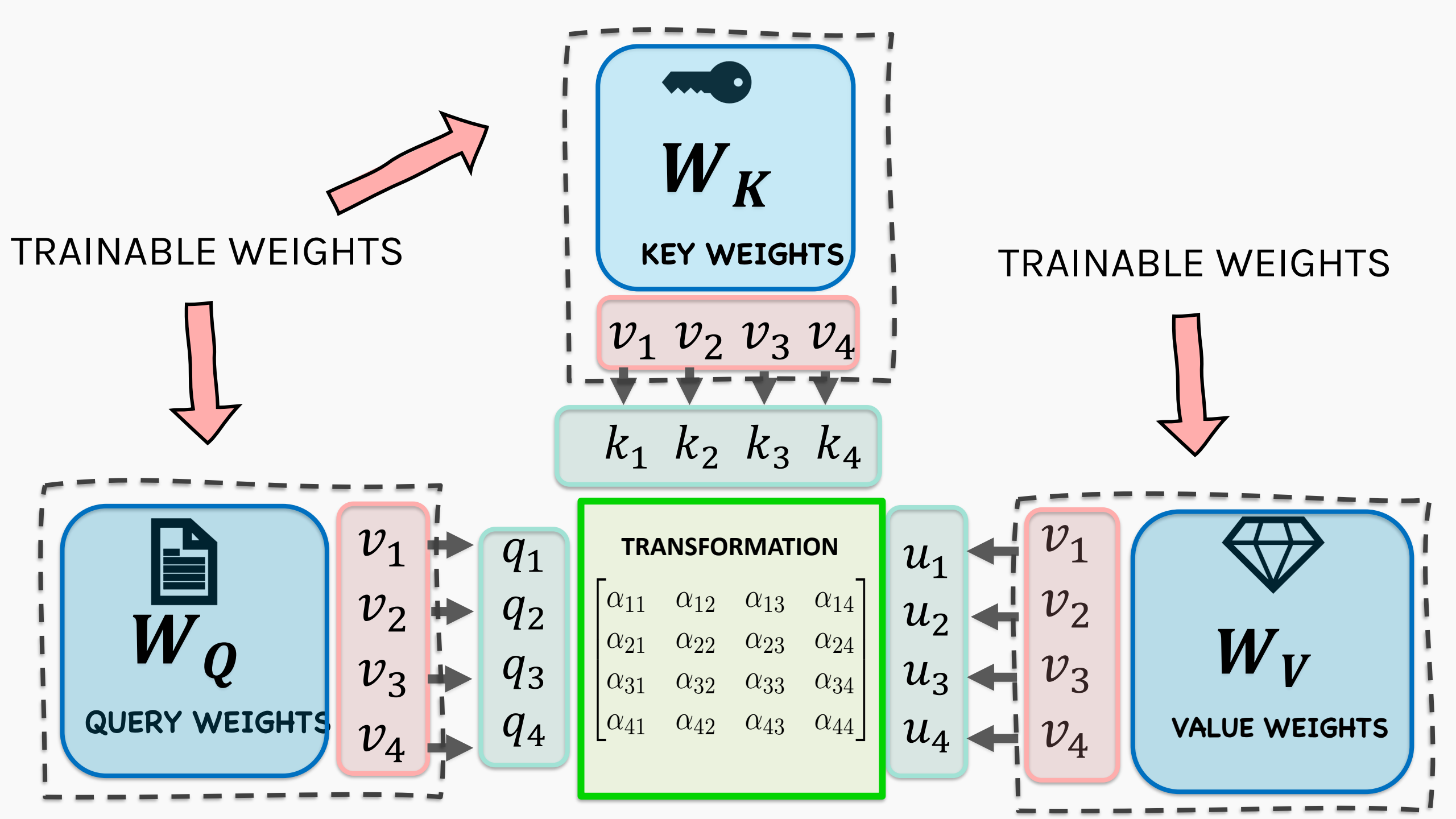
Database Analogy



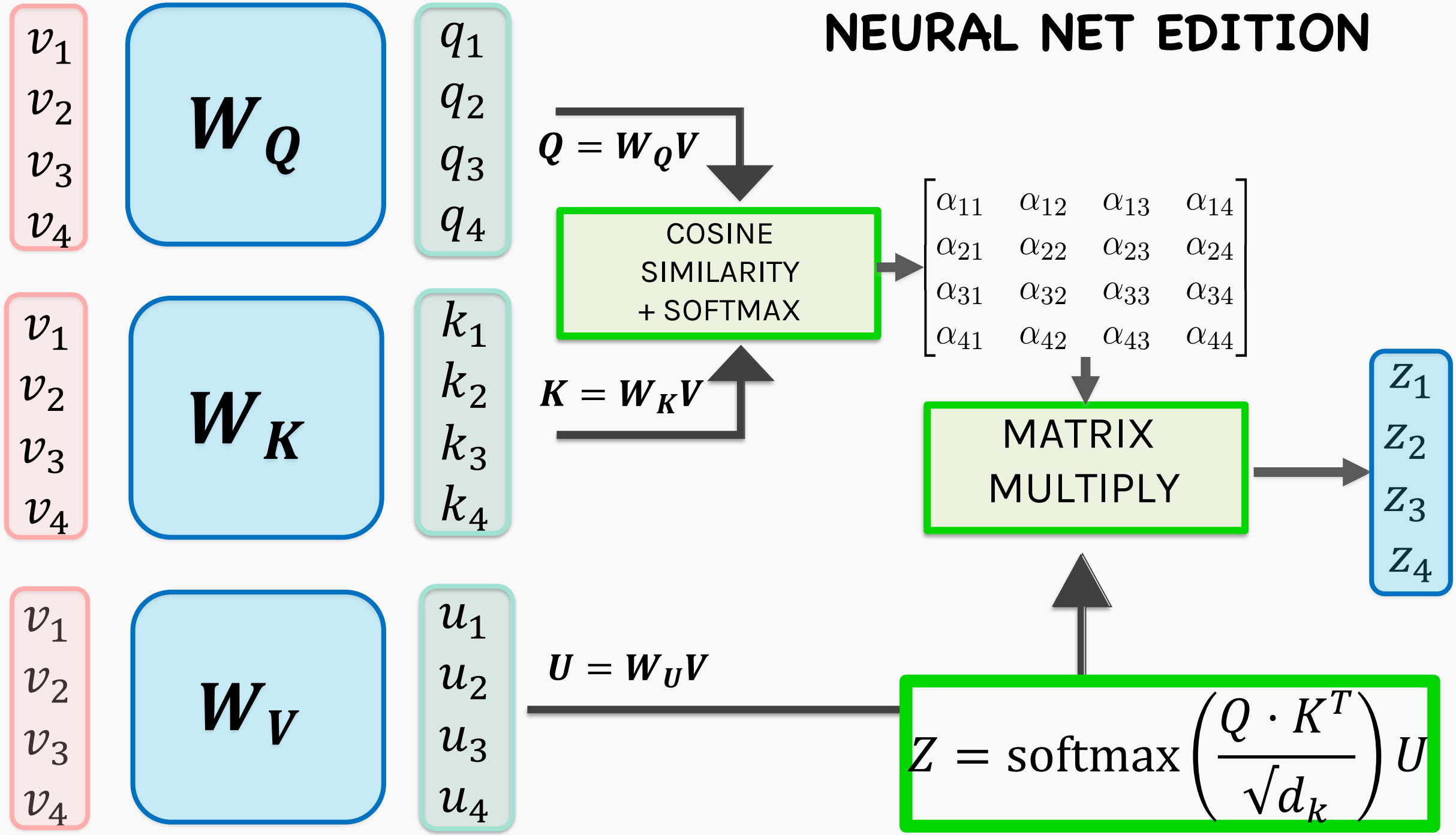
Database Analogy

For simplicity, we stick to our database analogy of **QUERY, KEY & VALUE**

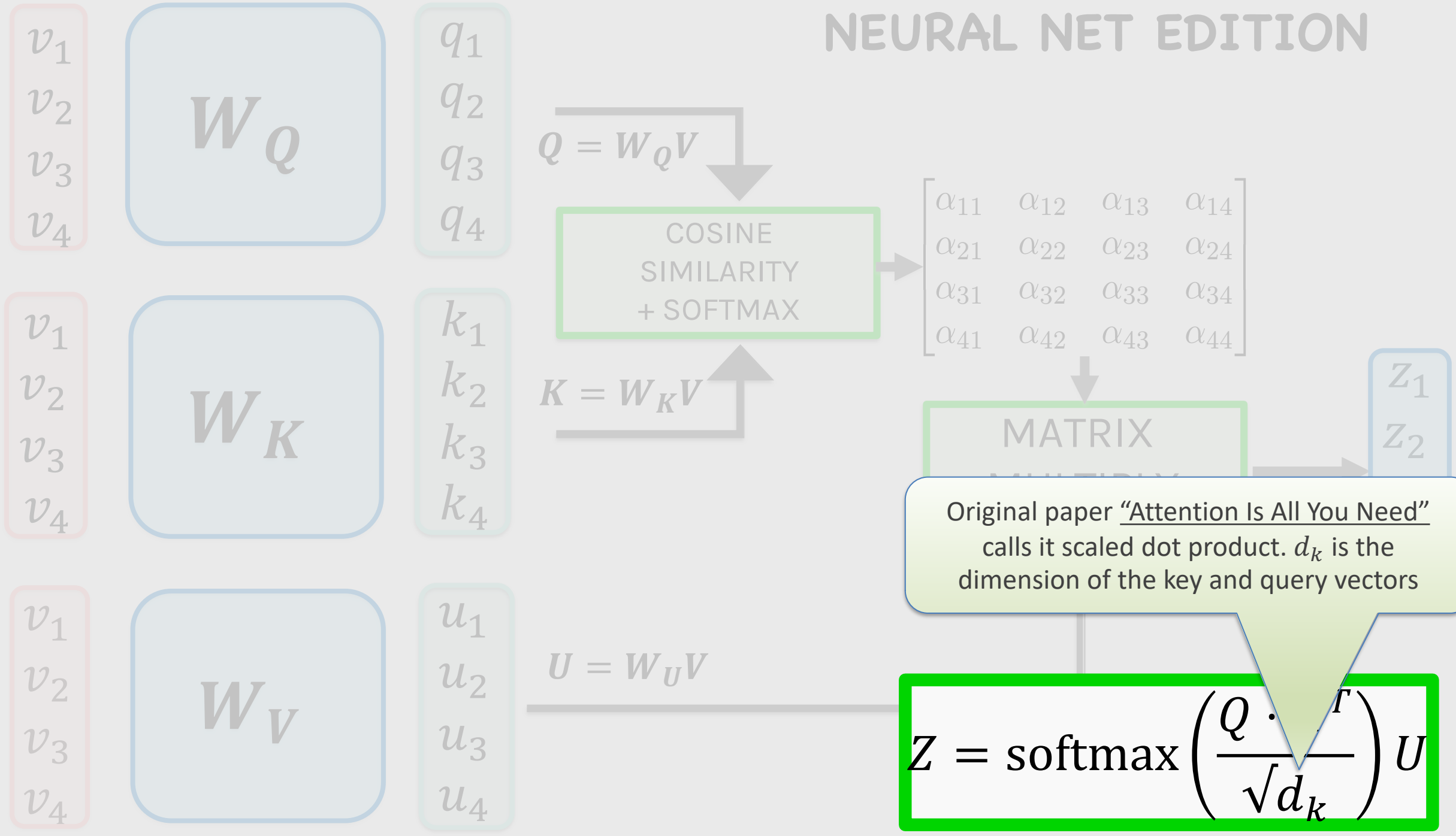


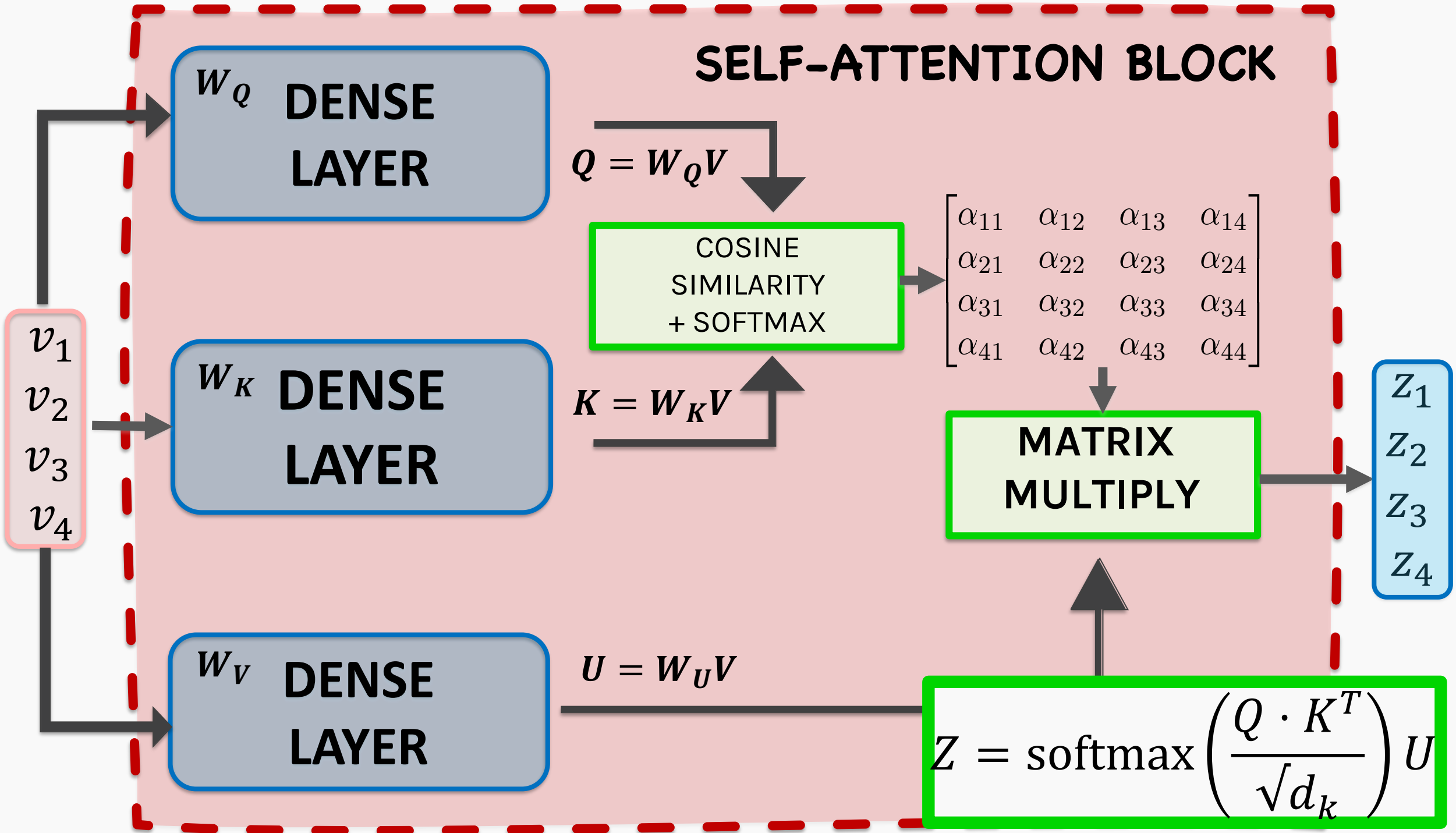

TRAINABLE WEIGHTS
W_K
KEY WEIGHTS
v_1 v_2 v_3 v_4
k_1 k_2 k_3 k_4
TRAINABLE WEIGHTS
W_Q
QUERY WEIGHTS
v_1 v_2 v_3 v_4
q_1 q_2 q_3 q_4
TRANSFORMATION
$\begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix}$
u_1 u_2 u_3 u_4
v_1 v_2 v_3 v_4
W_V
VALUE WEIGHTS

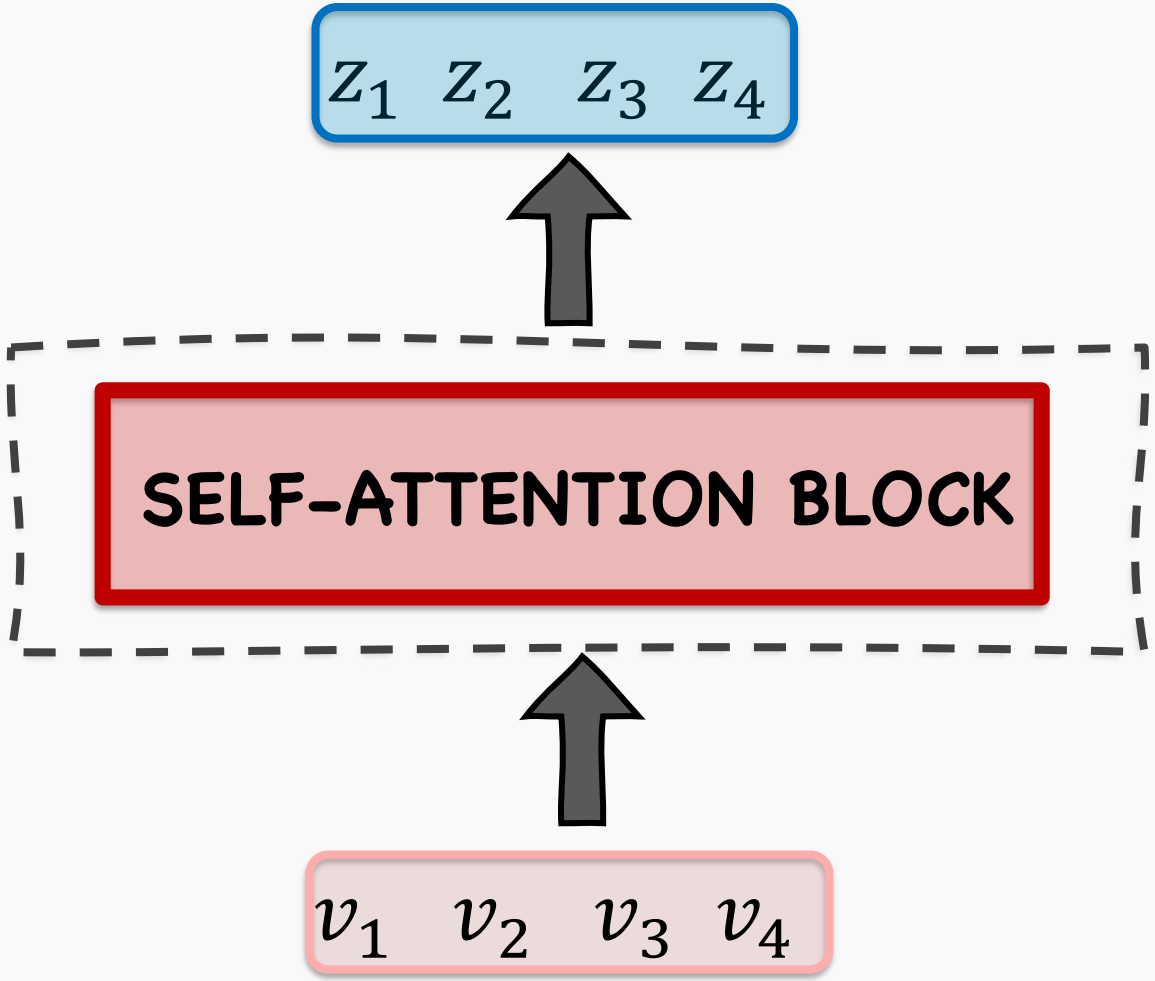
NEURAL NET EDITION

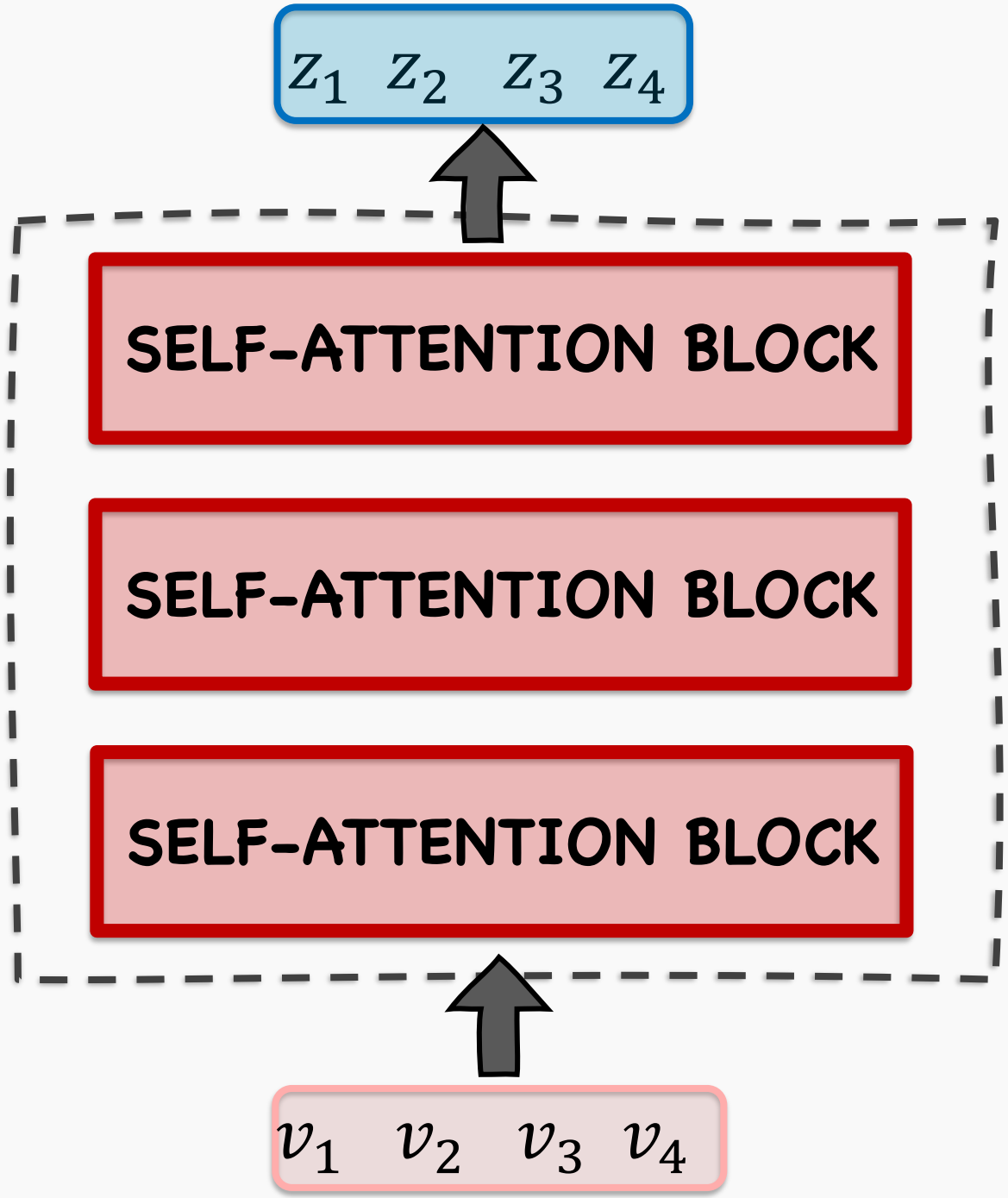


NEURAL NET EDITION



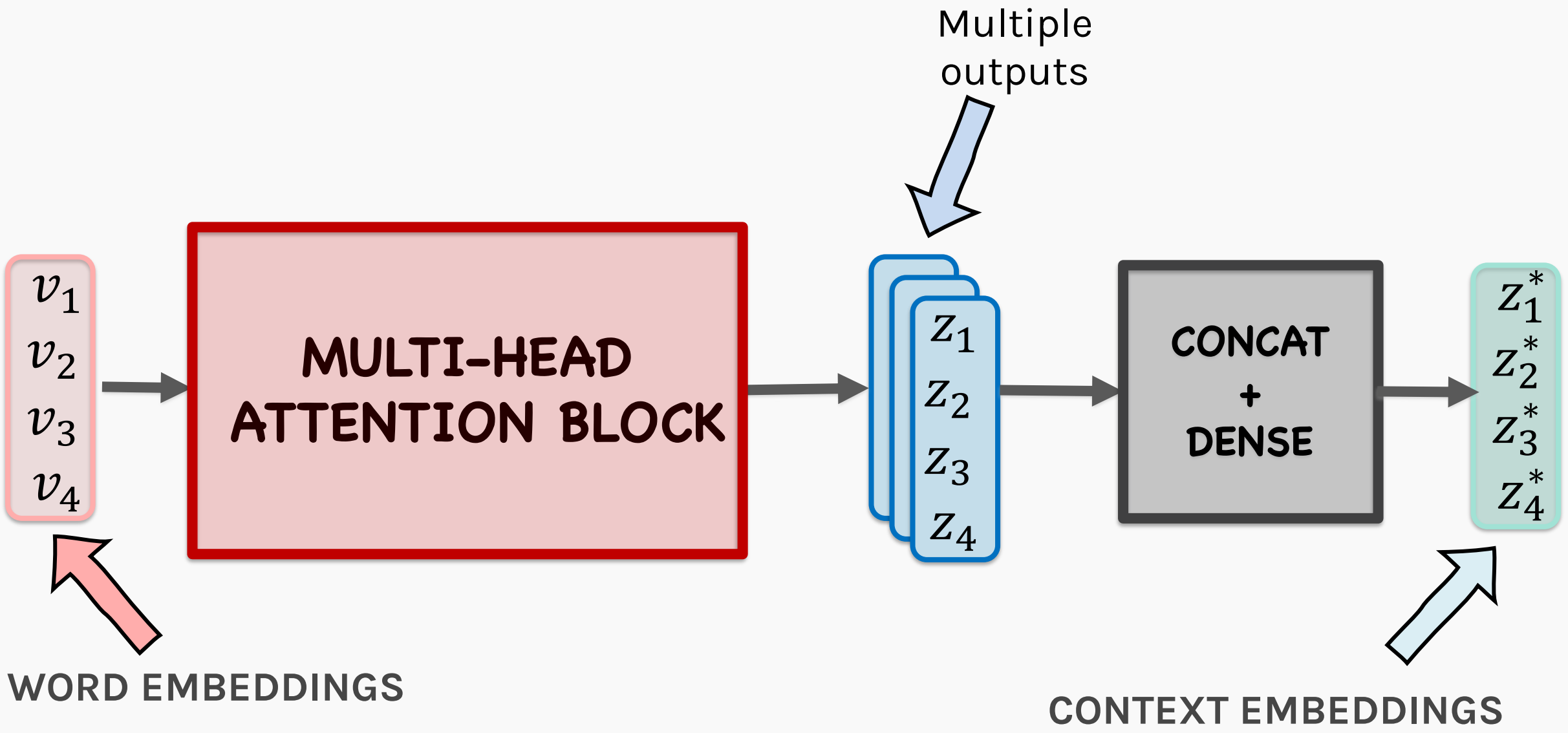

SELF-ATTENTION BLOCK
W_Q DENSE LAYER
W_K DENSE LAYER
W_V DENSE LAYER
v_1
v_2
v_3
v_4
$Q = W_Q V$
$K = W_K V$
$U = W_U V$
COSINE SIMILARITY + SOFTMAX
$\begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix}$
MATRIX MULTIPLY
z_1
z_2
z_3
z_4
$Z = \text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right) U$


z_1 z_2 z_3 z_4
SELF-ATTENTION BLOCK
v_1 v_2 v_3 v_4

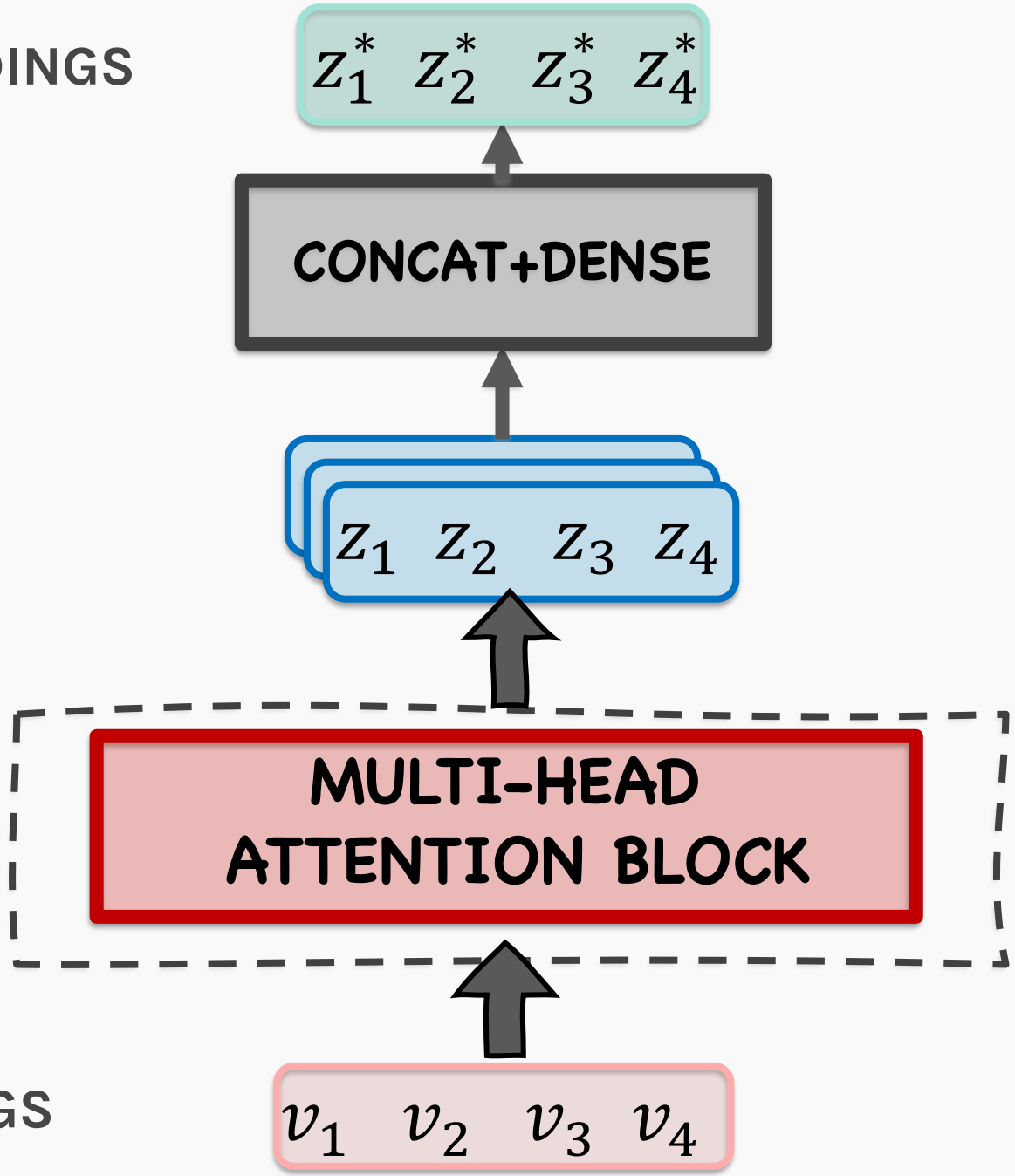

z_1 z_2 z_3 z_4
SELF-ATTENTION BLOCK
SELF-ATTENTION BLOCK
SELF-ATTENTION BLOCK
v_1 v_2 v_3 v_4

RECAP: CNNs

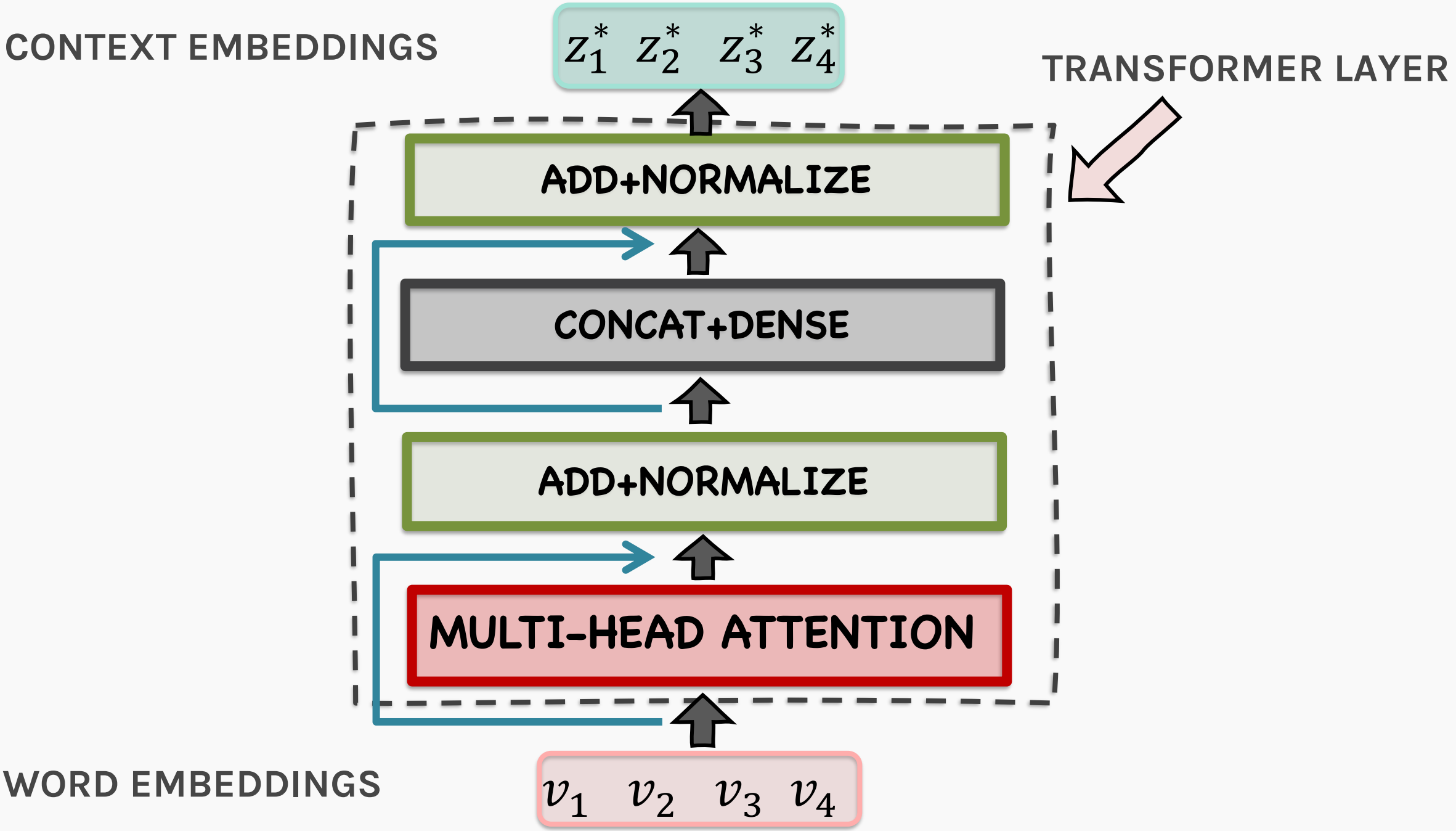
Imagine that we want to recognize swans in an image:


Multiple outputs
v_1
v_2
v_3
v_4
MULTI-HEAD ATTENTION BLOCK
z_1
z_2
z_3
z_4
CONCAT + DENSE
z_1^*
z_2^*
z_3^*
z_4^*
WORD EMBEDDINGS
CONTEXT EMBEDDINGS

CONTEXT EMBEDDINGS

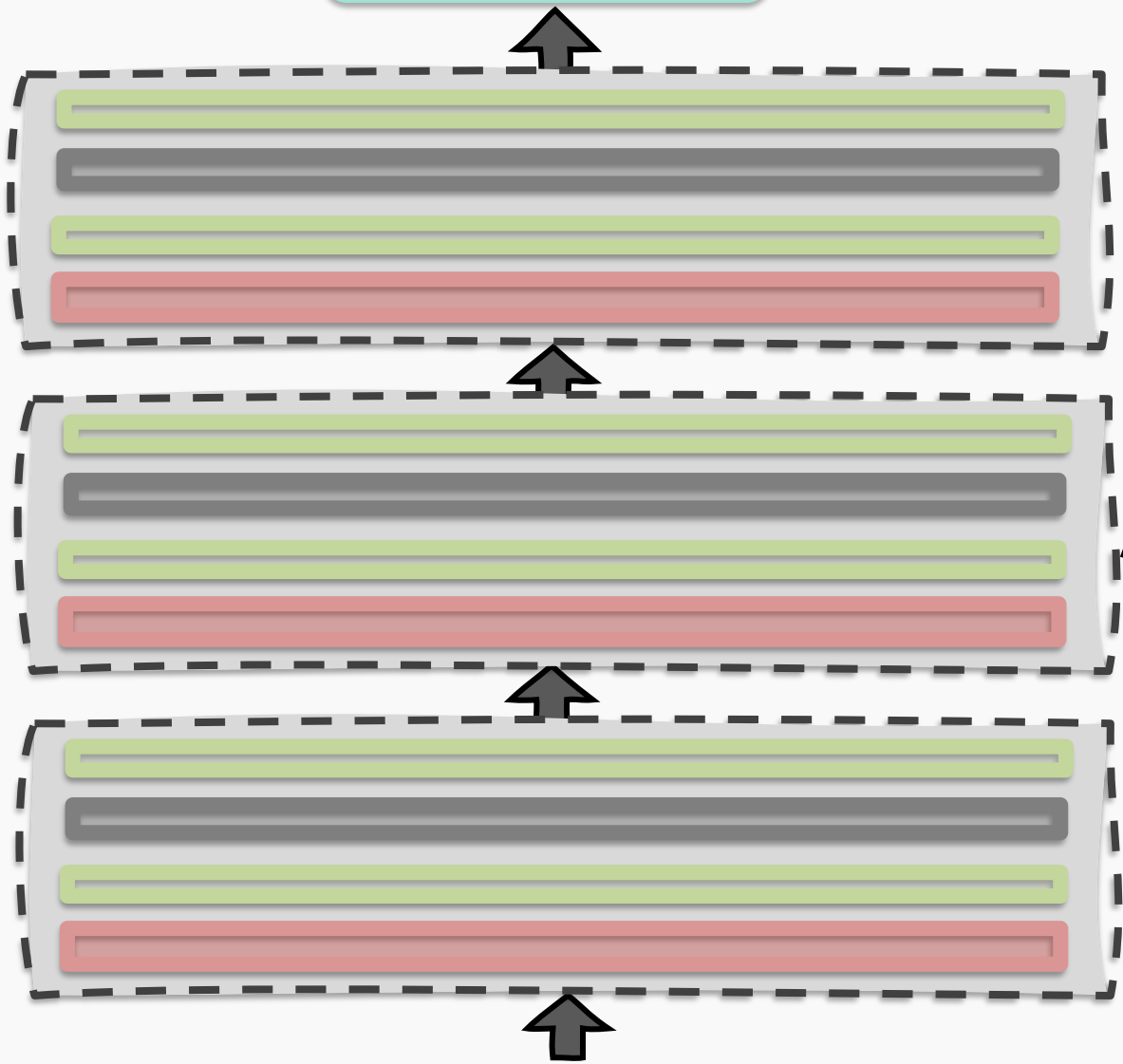


WORD EMBEDDINGS


CONTEXT EMBEDDINGS
z_1^* z_2^* z_3^* z_4^*
TRANSFORMER LAYER
ADD+NORMALIZE
CONCAT+DENSE
ADD+NORMALIZE
MULTI-HEAD ATTENTION
WORD EMBEDDINGS
v_1 v_2 v_3 v_4

CONTEXT EMBEDDINGS

z_1^* z_2^* z_3^* z_4^*



TRANSFORMER LAYERS

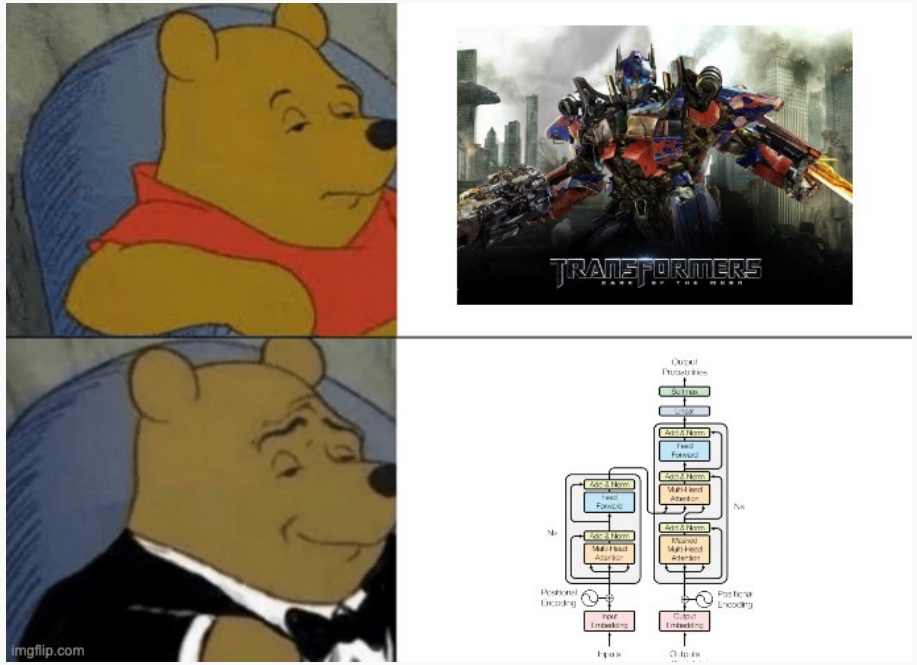
WORD EMBEDDINGS

v_1 v_2 v_3 v_4

What we want

LANGUAGE MODEL WISHLIST

- Position and order of words are the essential parts of any language
- Recurrent Neural Networks (RNNs) inherently take the order of word into account
- Multi-head attention blocks do not take such an order by design, so there's the need to incorporate the order of the words separately

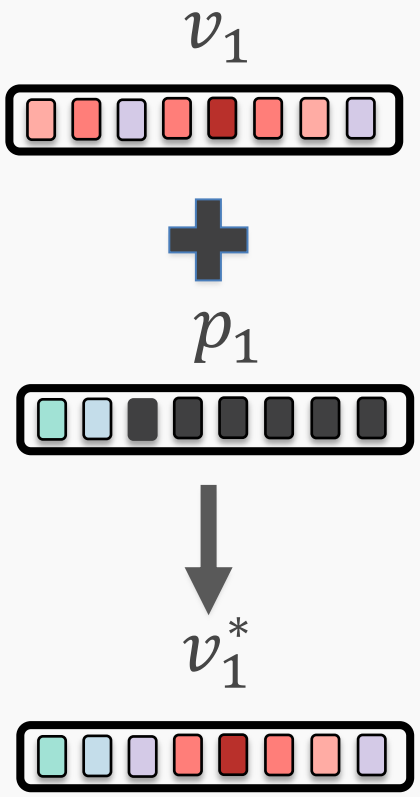


💡 **IDEA #255:** Positional Encoding

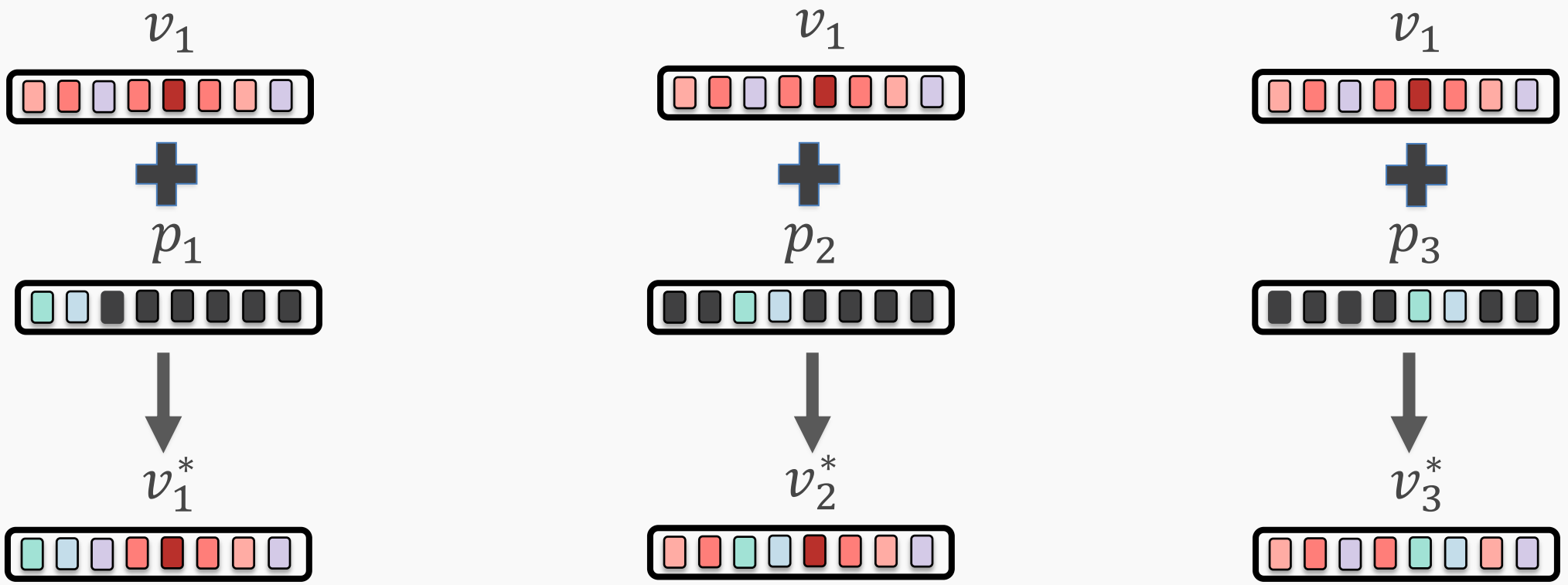
Positional Encoding

- A very simple way to do this would be to modify the input embedding v_i with a positional vector p_i which encodes some information of the position of the input embedding

- Thus, the same input embedding v_i will be a different value v_i^* depending on the position of the embedding in the sentence



Positional Encoding

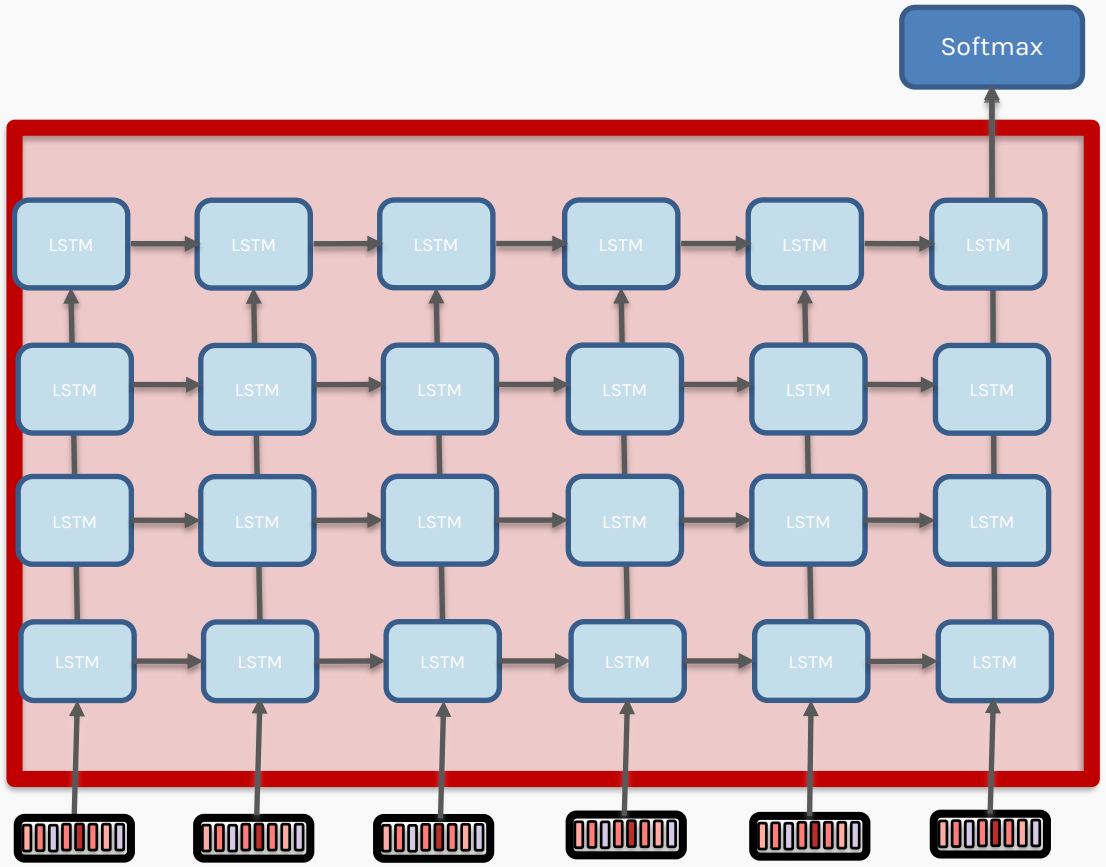


In the above case, $v_1^* \neq v_2^* \neq v_3^*$

Bringing it all together

Comparison – RNN v/s Transformer

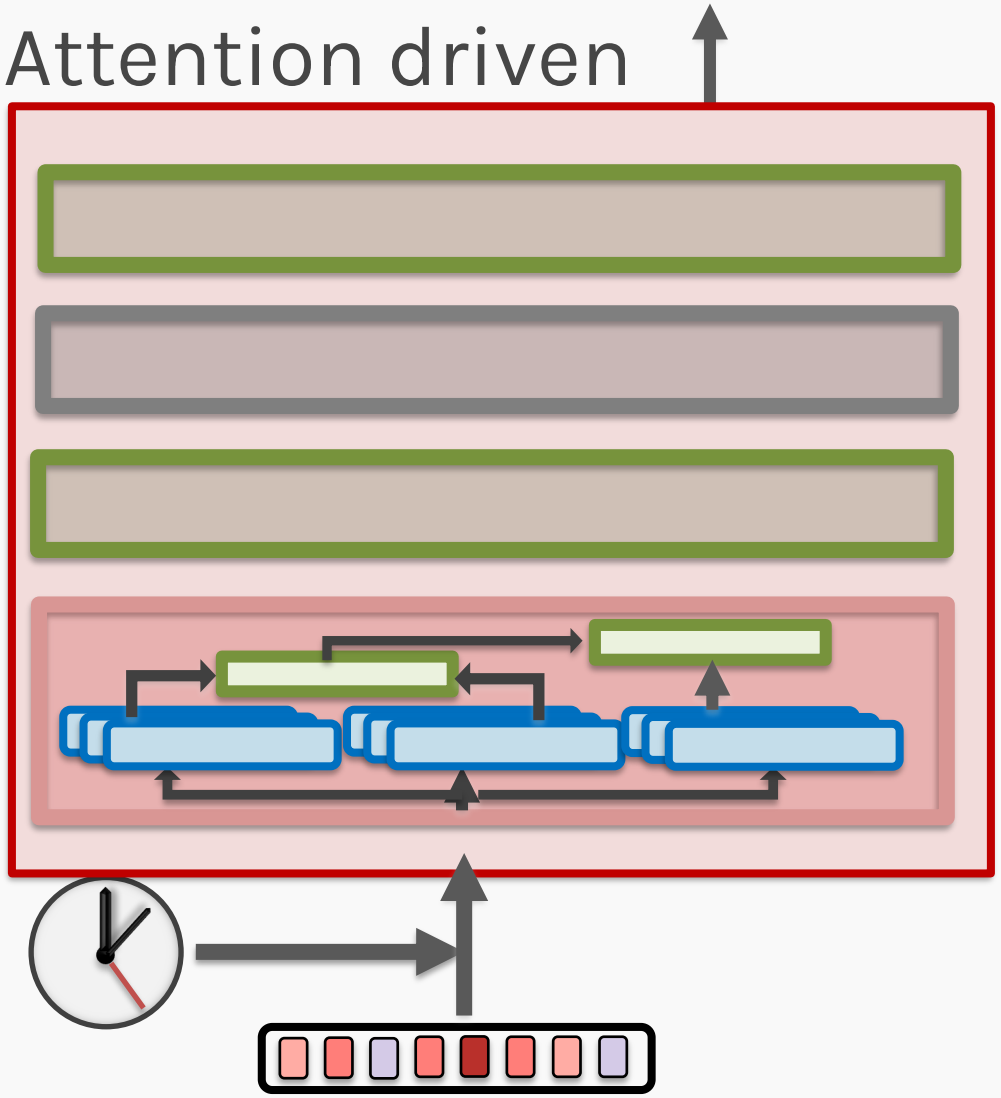
BEFORE



Recurrent units

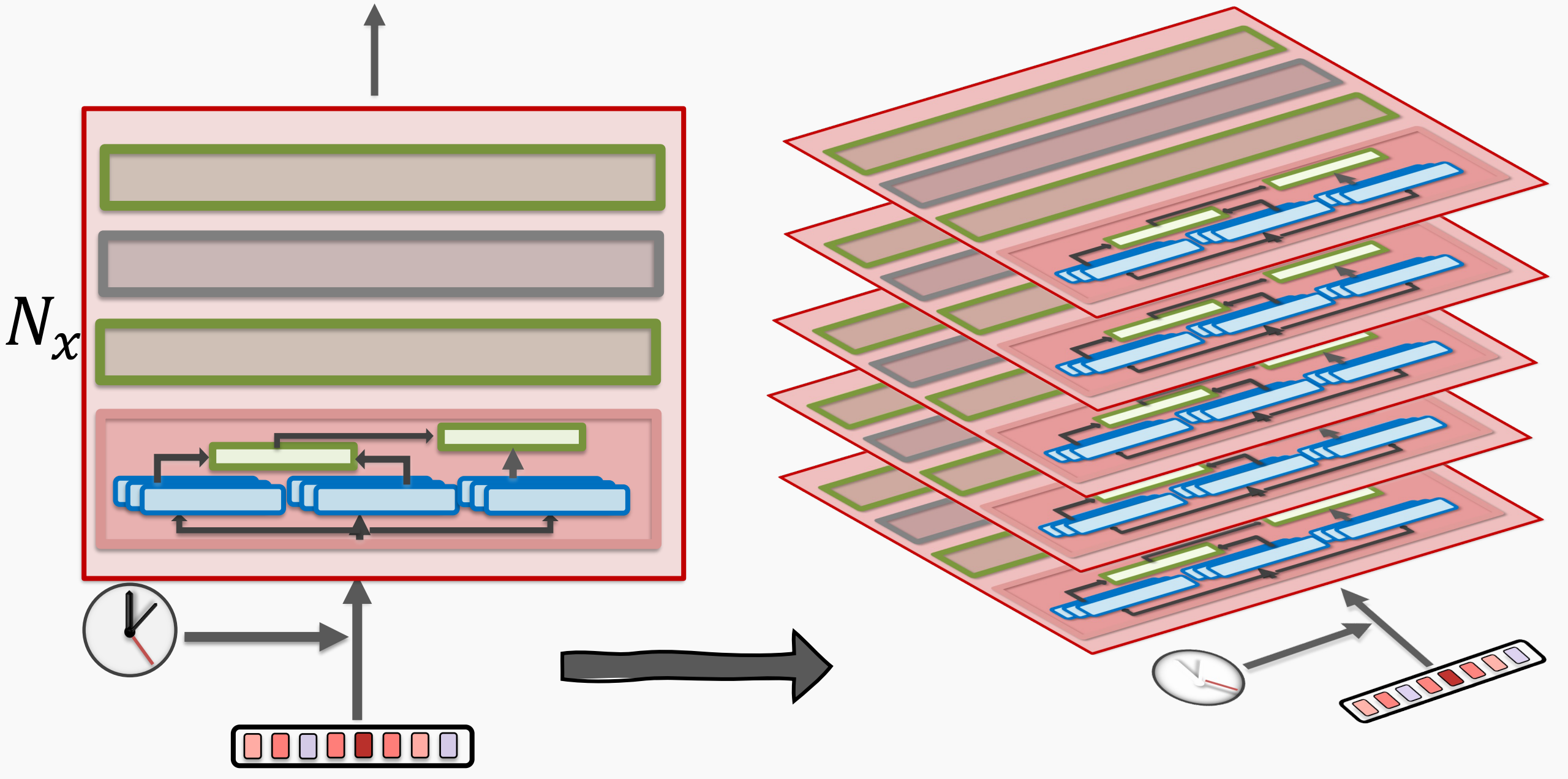
AFTER

Attention driven



N_x

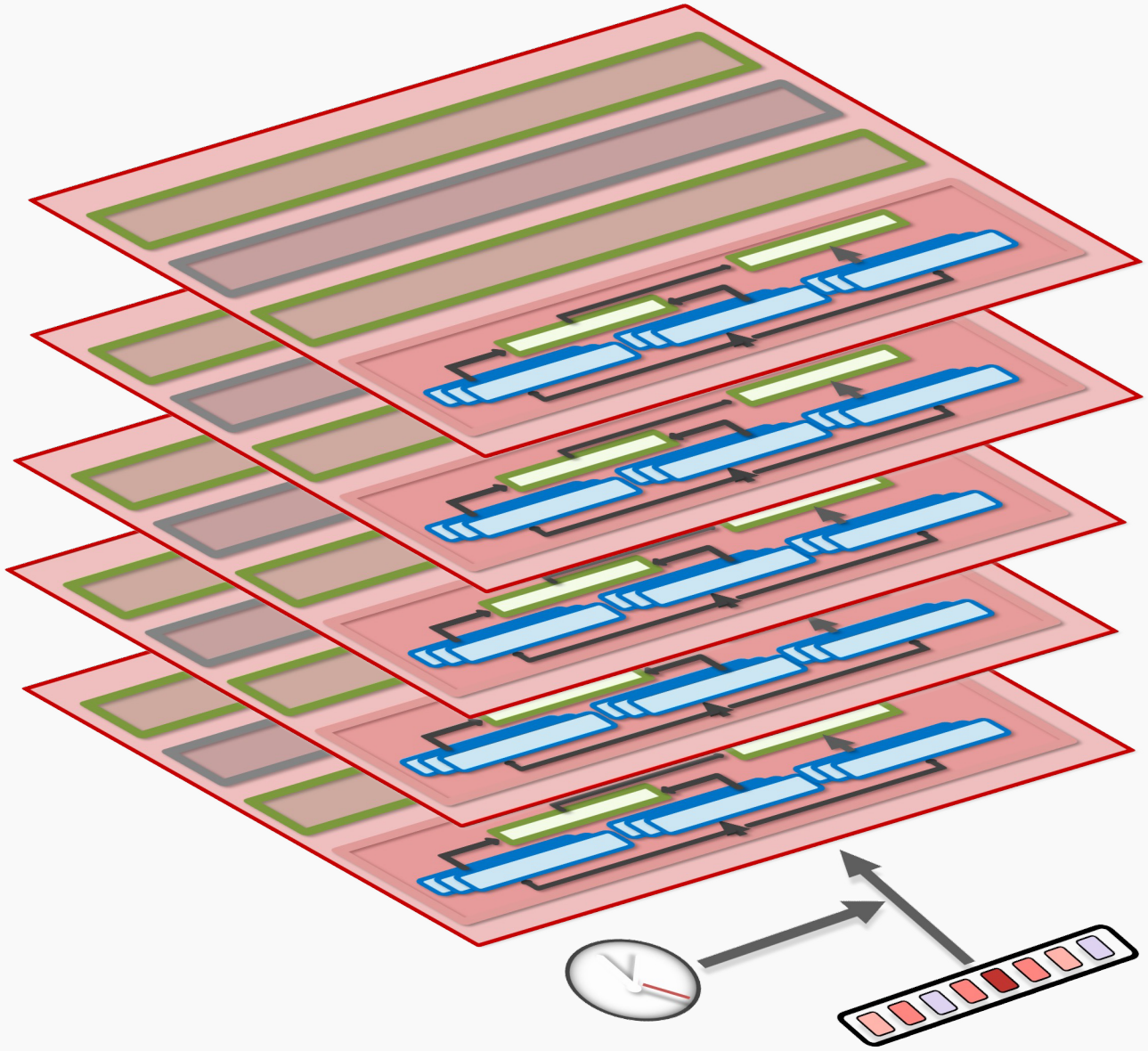
Transformer as a 3-D model



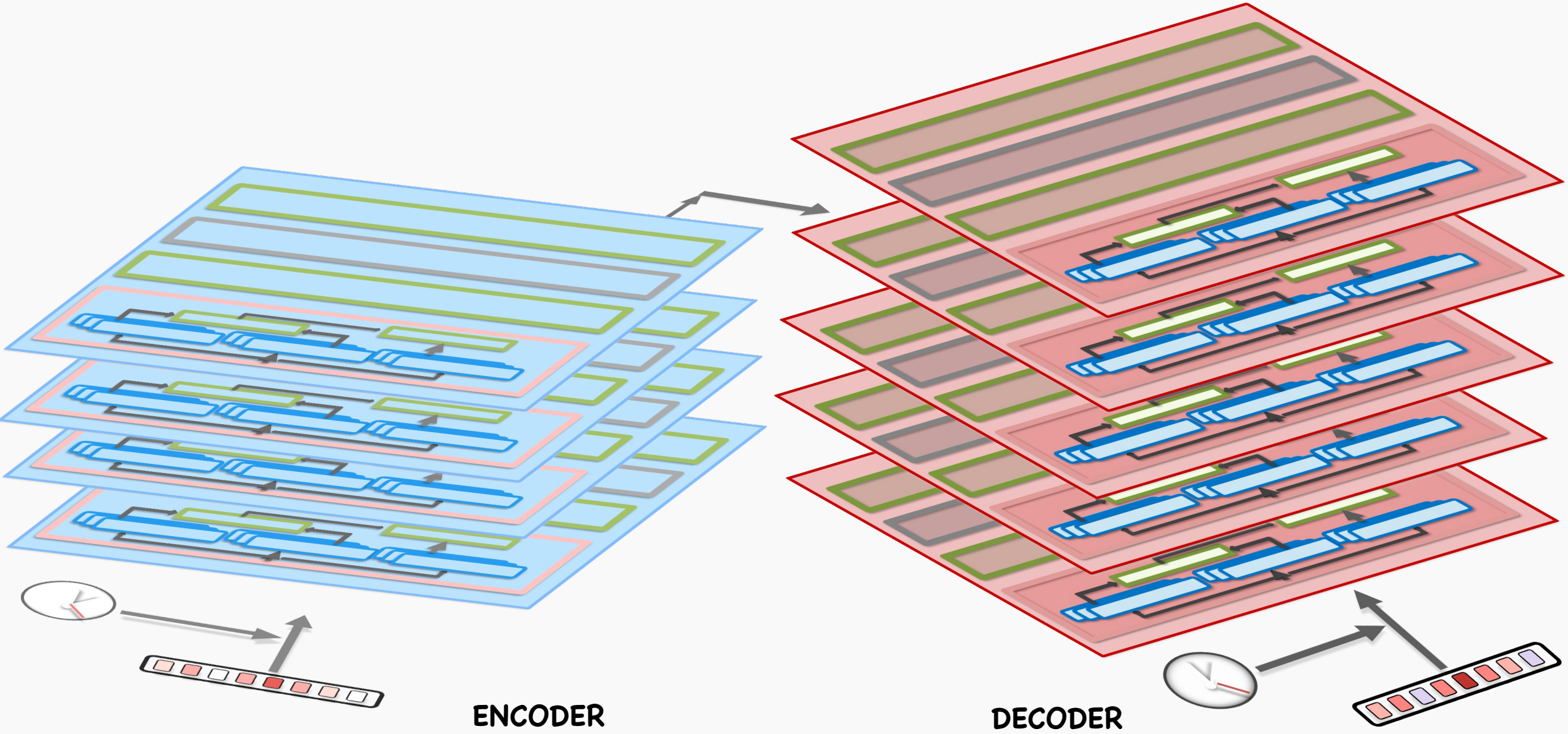
Transformers - Summary

Language Model Wishlist?

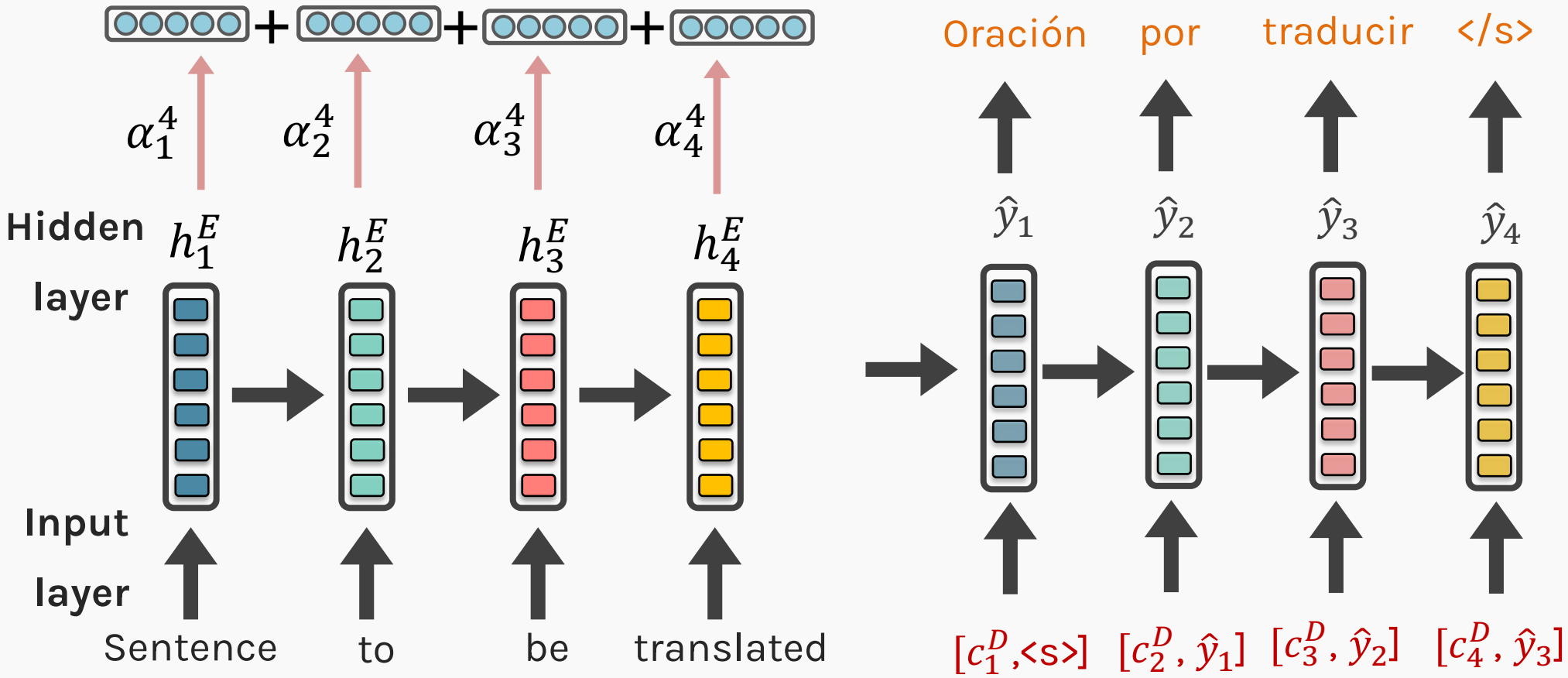
- ☑ We want to have strong contextual relations between words - **DONE**
- ☑ We want words to have sequential information - **DONE**
- ☑ We need an architecture that can be trained in parallel (non-Markovian property) - **DONE**



Transformers - Summary



RECAP: Seq2Seq + Attention

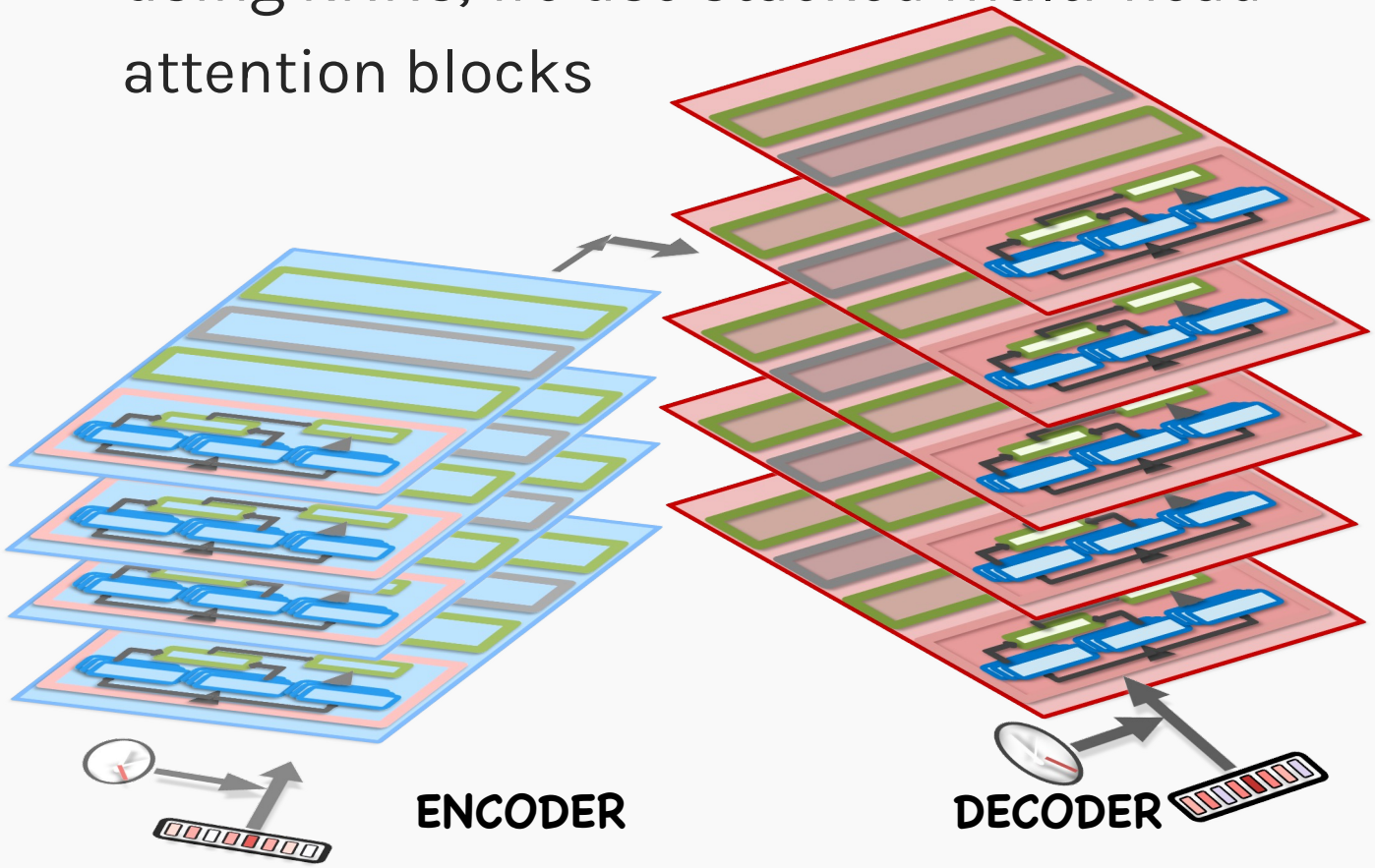


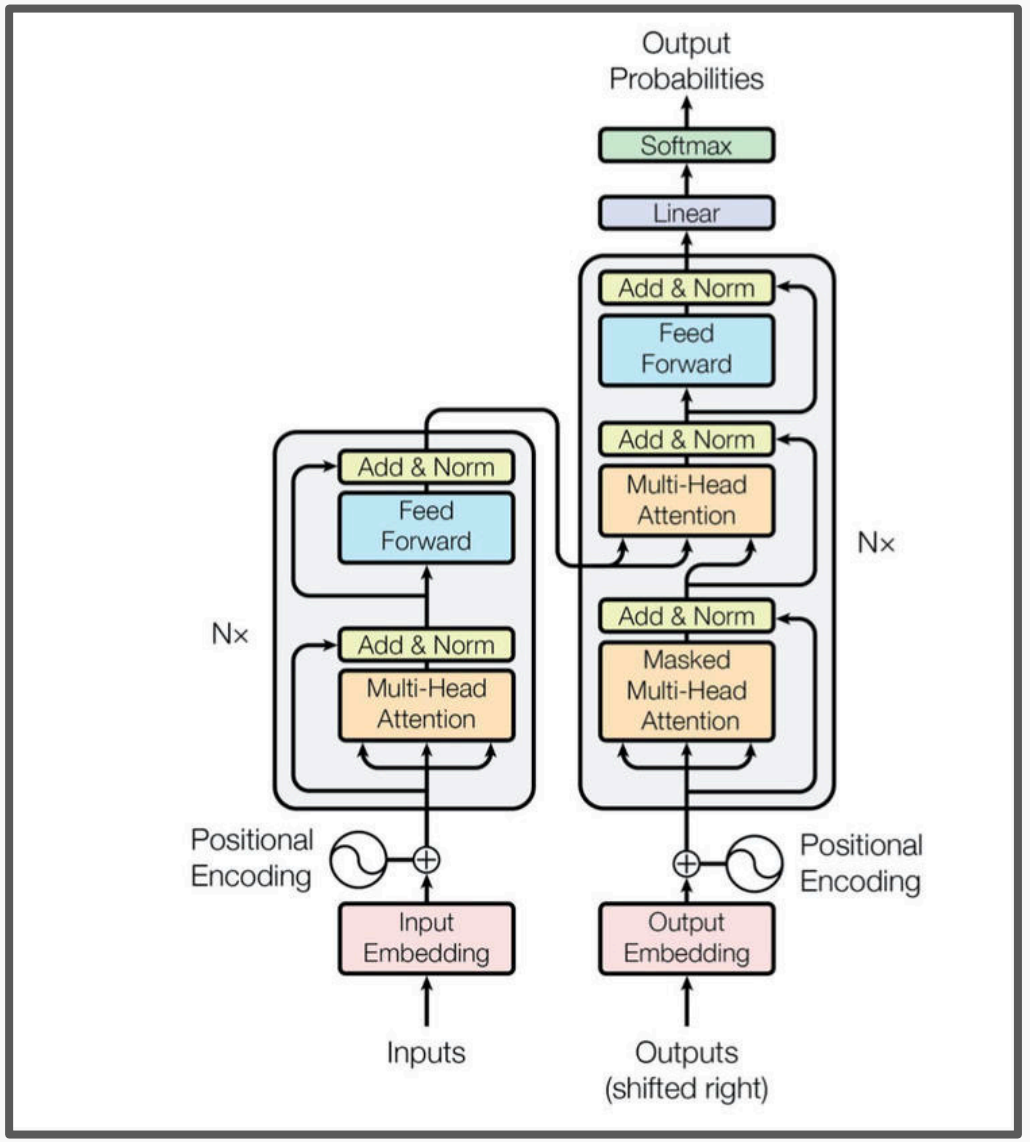
ENCODER RNN

DECODER RNN

Transformers - Summary

Transformers consist of an Encoder-Decoder architecture, but instead of using RNNs, we use stacked multi-head attention blocks

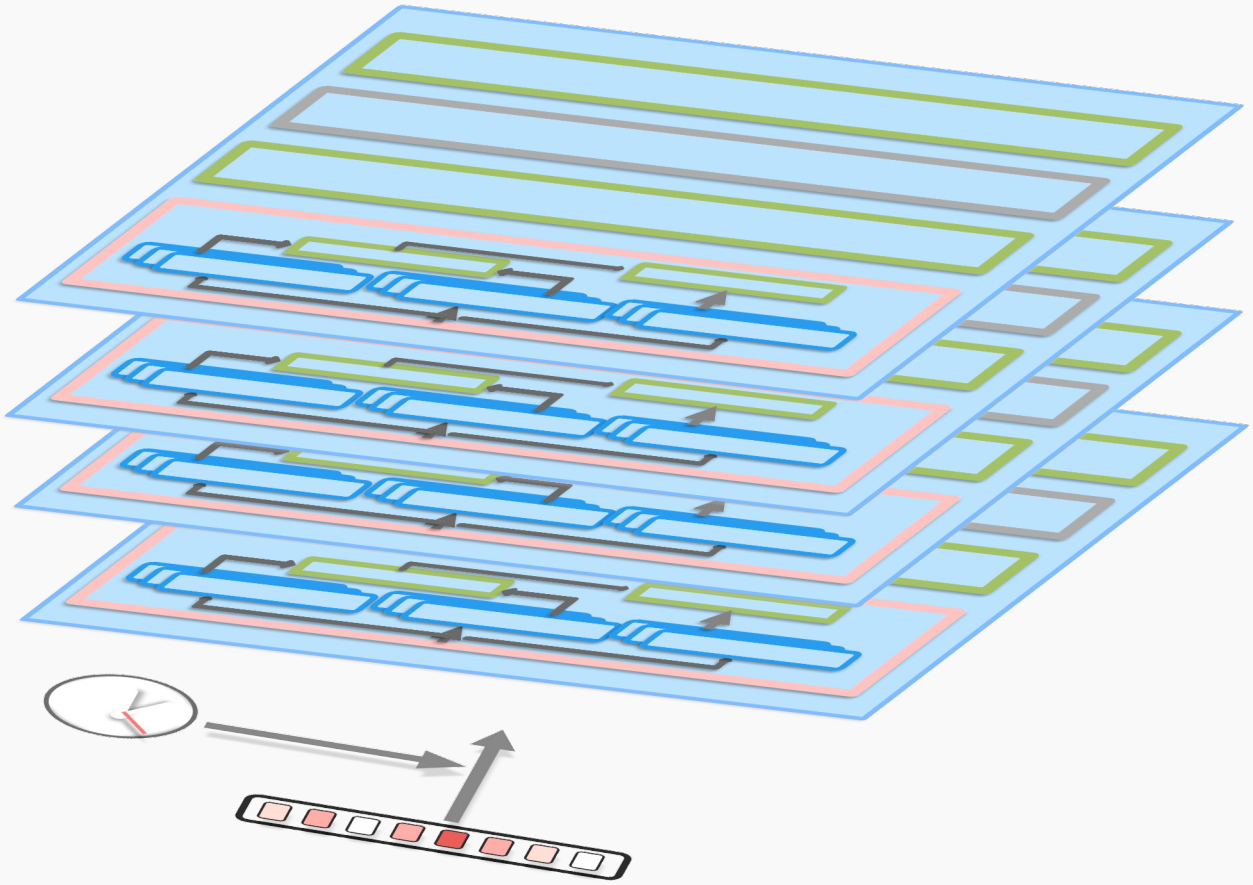




From Transformers to **BERT**

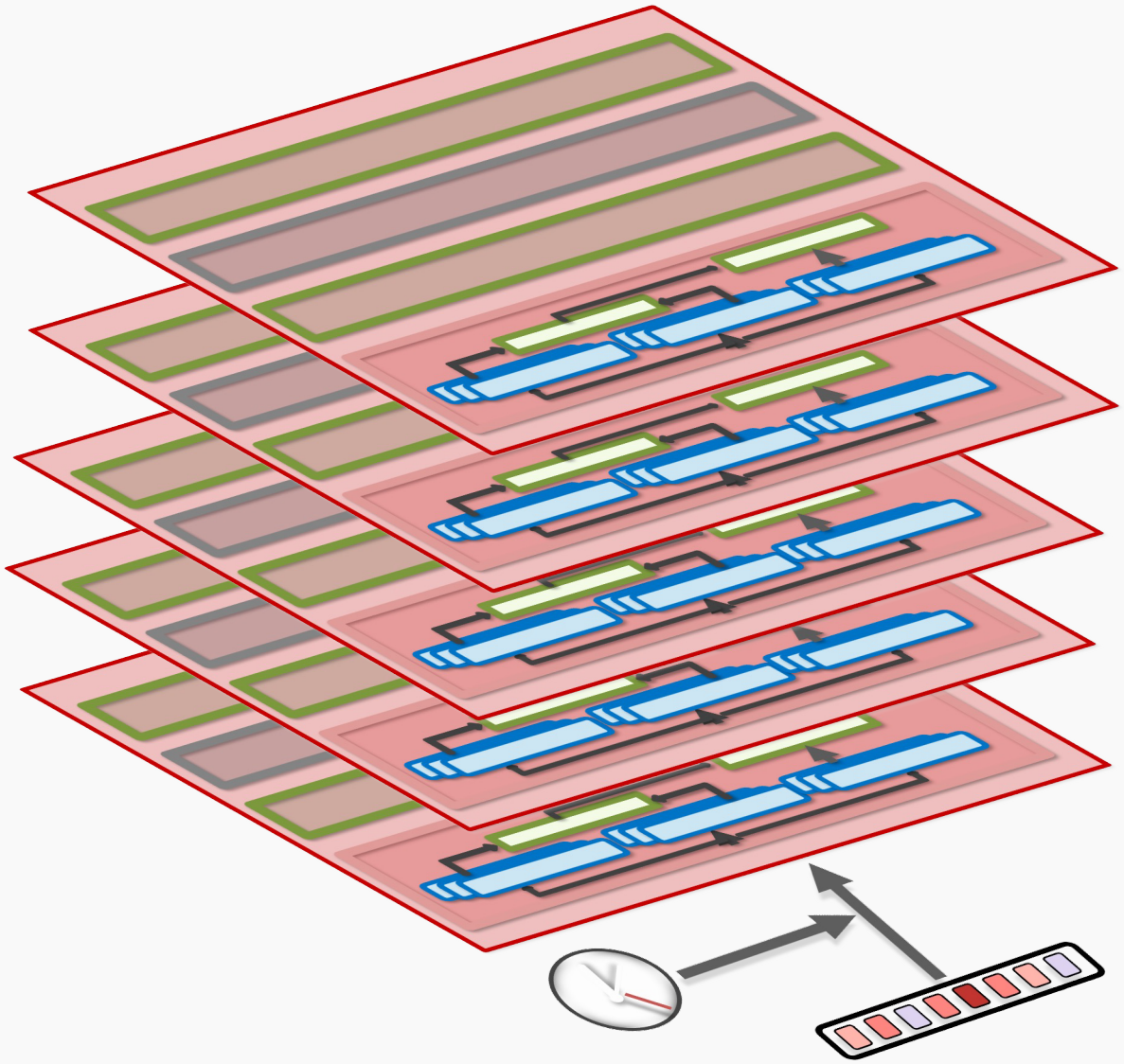
- Instead of an **Encoder-Decoder** architecture for machine translation, what if we just use the encoder for Language Model and other NLP tasks

- This led to the new architecture called Bidirectional Encoder Representations from Transformers, or more commonly known as BERT



From Transformers to **GPT**

- Now if we use just the decoder for Language Model
- We get a Causal Language Model (A word is predicted using words from its left context)
- Also known as autoregressive model
- Generative Pre-trained Transformer, or more commonly known as GPT



BERT vs GPT

BERT

- Masked Language Model
- Bidirectional language model
- Made up of only the Encoder with stacked transformer blocks.
- Good for a language model

GPT

- Auto-regressive Language Model
- Unidirectional language model
- Made up of only the Decoder with stacked transformer blocks.
- Good for generating text