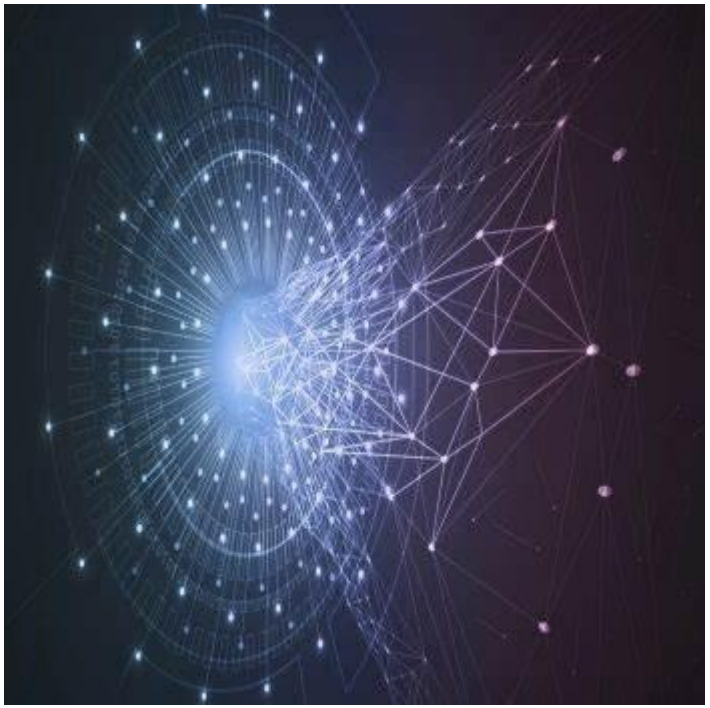
University of Ljubljana, Faculty of Computer and Information Science

Transformers



Prof Dr Marko Robnik-Šikonja

Natural Language Processing, Edition 2025

Contents

- transformer networks

Literature

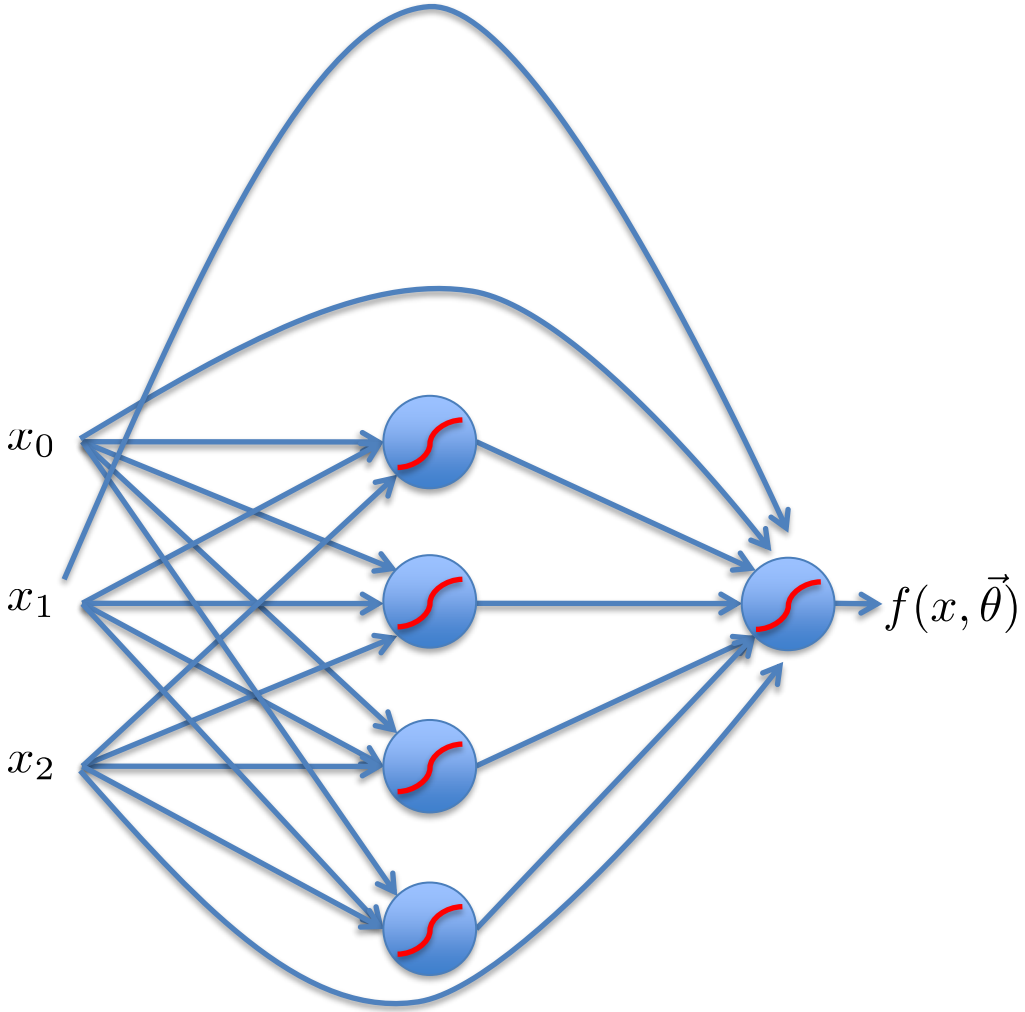
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. In *Advances in neural information processing systems* (pp. 5998-6008).
- Jay Alammar: The Illustrated Transformer. Blog, 2019.
- Sasha Rush: Annotated Transformer. Jupyter Notebook using PyTorch.
- Elvis Saravia: https://github.com/dair-ai/Transformers-Recipe

- some slides by Jay Alammar, Jacob Devlin and Andrej Miščič

Problems with RNNs

- We want parallelization, but RNNs are inherently sequential
- For parallelization we need fixed input size (relatively short)
- Despite GRUs and LSTMs, RNNs still need attention mechanism to deal with long range dependencies – path length between states grows with sequence
- If attention gives us access to any state… maybe we can just use attention and don't need the RNN?

Problems with long sequences: one solution is level jumping

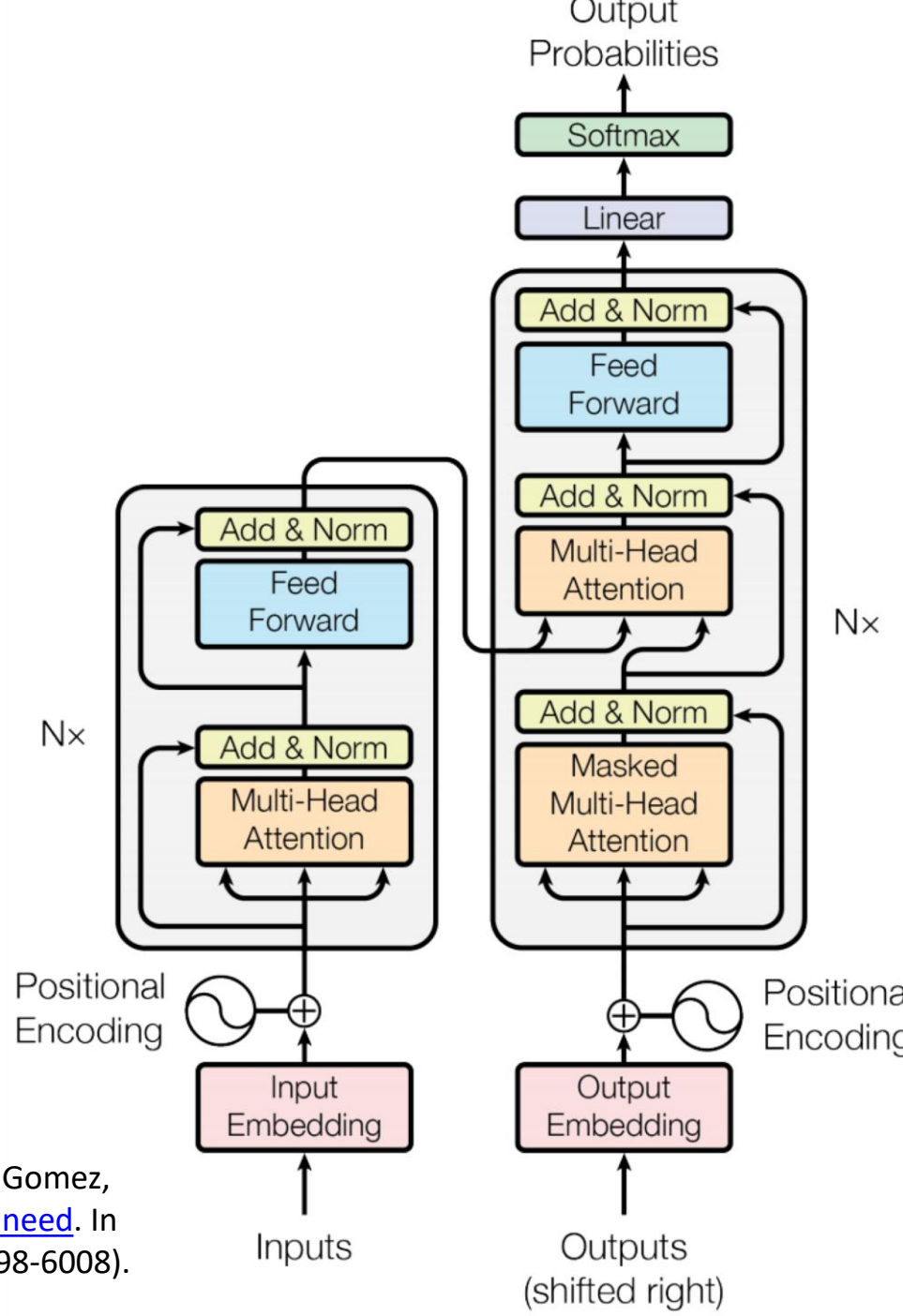


Transformer model

- currently the most successful DNN
- non-recurrent
- architecturally it is an encoder-decoder model
- fixed input length (relatively long but projected to short)
- adapted for parallelization
- adapted for GPU (TPU) processing
- based on extreme use of attention
- well-scalable
- includes level jumping to prevent forgetting
- encoder and decoder can be used independently

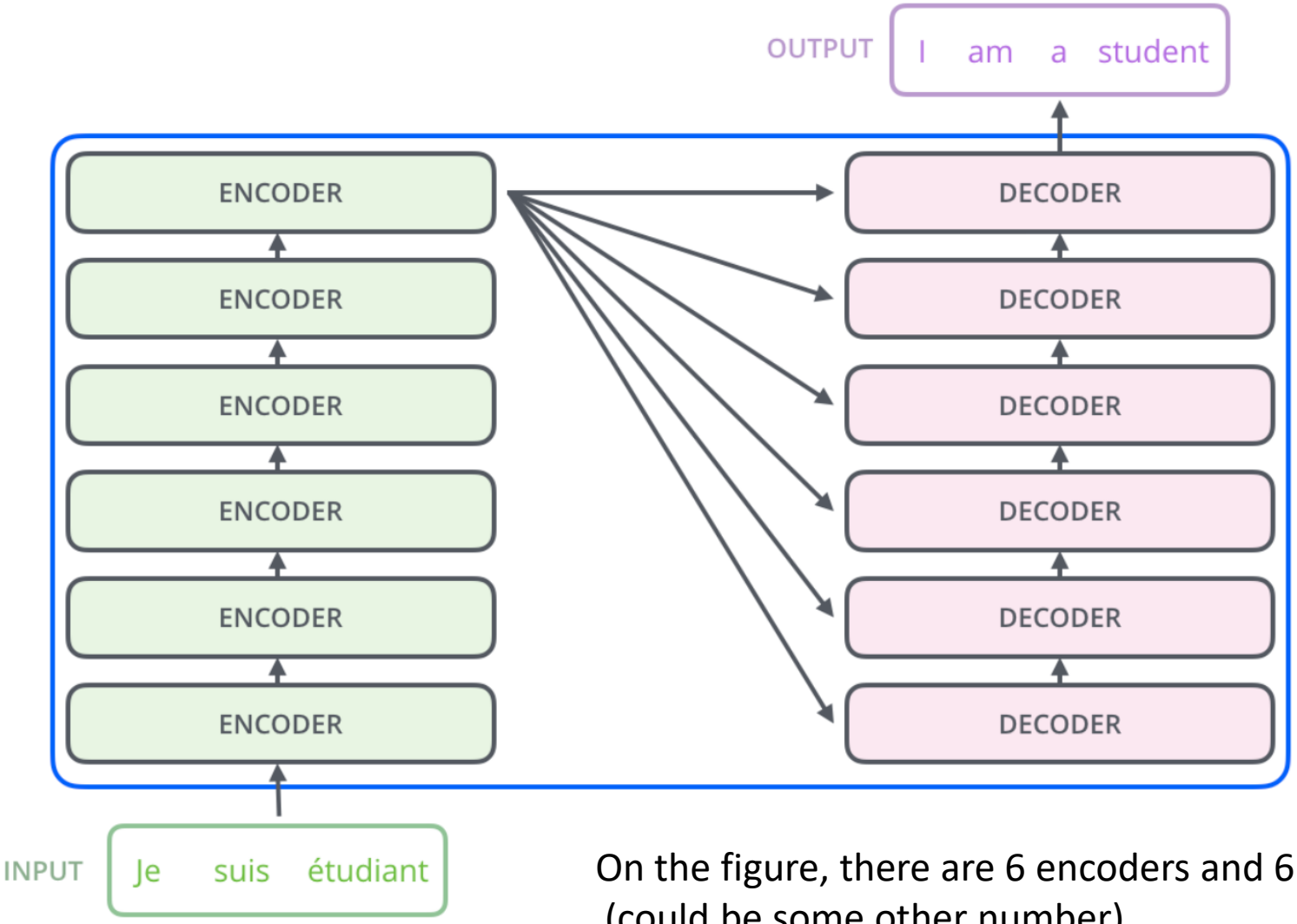
Transformer overview

- Initial task: machine translation with parallel corpus
- Predict each translated word
- Final cost/loss/error function was standard cross-entropy loss on top of softmax



Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. In *Advances in neural information processing systems* (pp. 5998-6008).

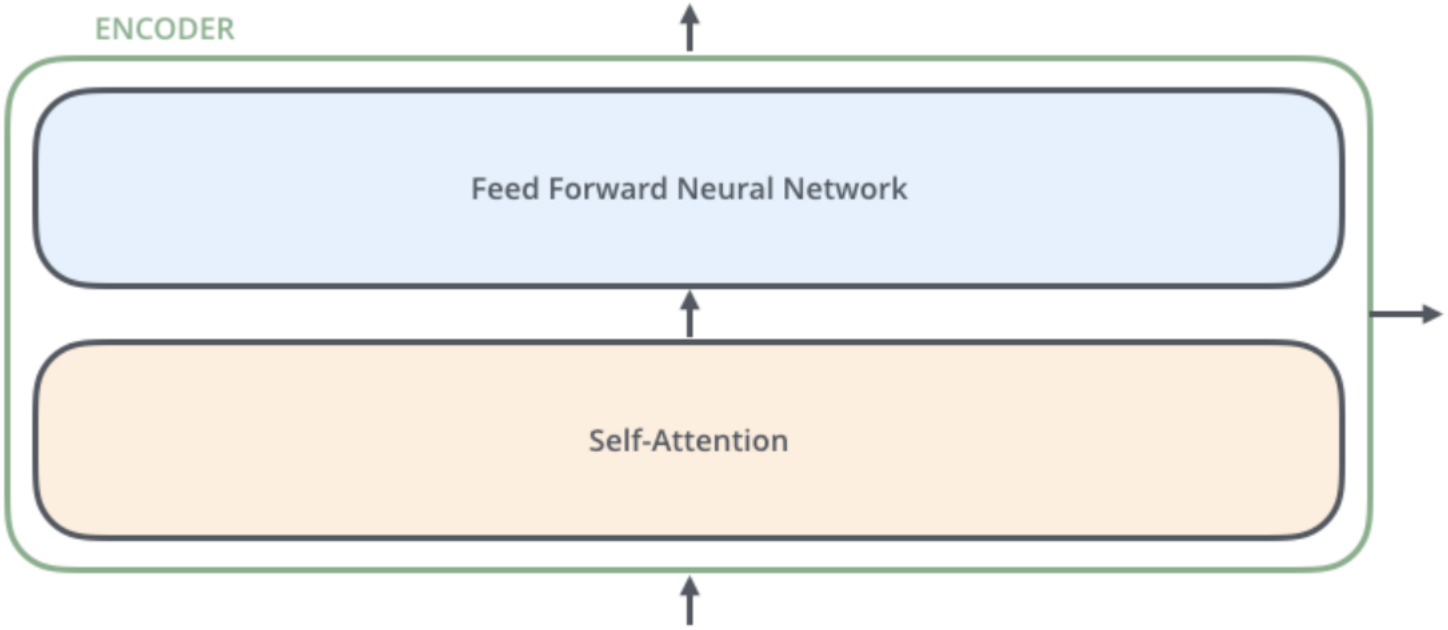
Transformer is an encoder-decoder model



On the figure, there are 6 encoders and 6 decoders (could be some other number).

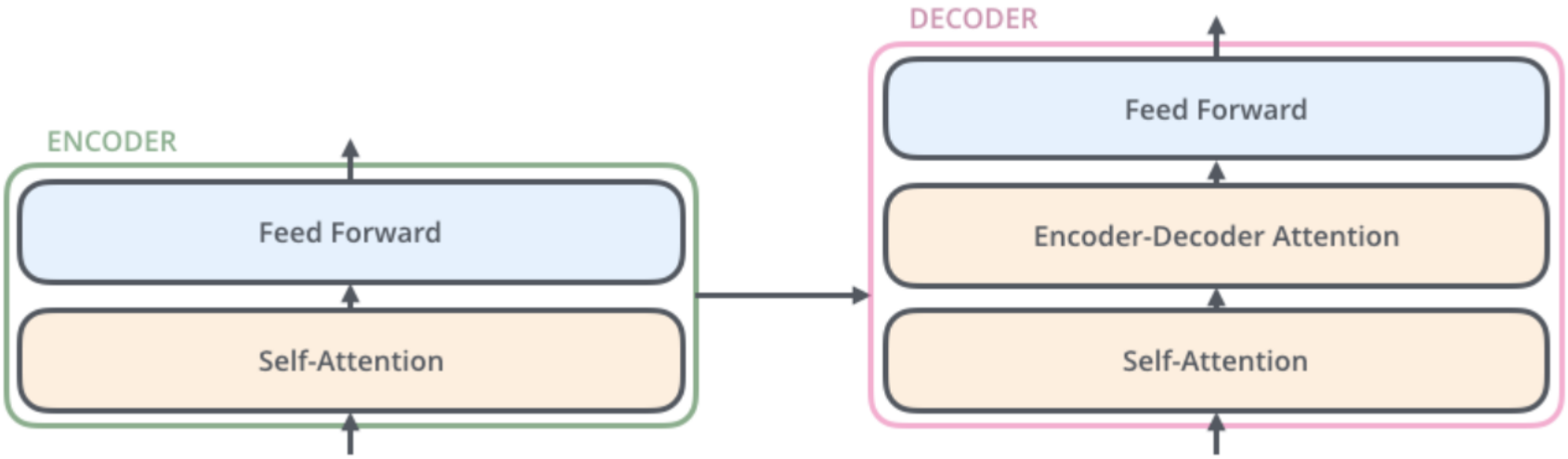
Transformer: encoder

- two layers
- no weight sharing between different encoders
- self-attention helps to focus on relevant part of input

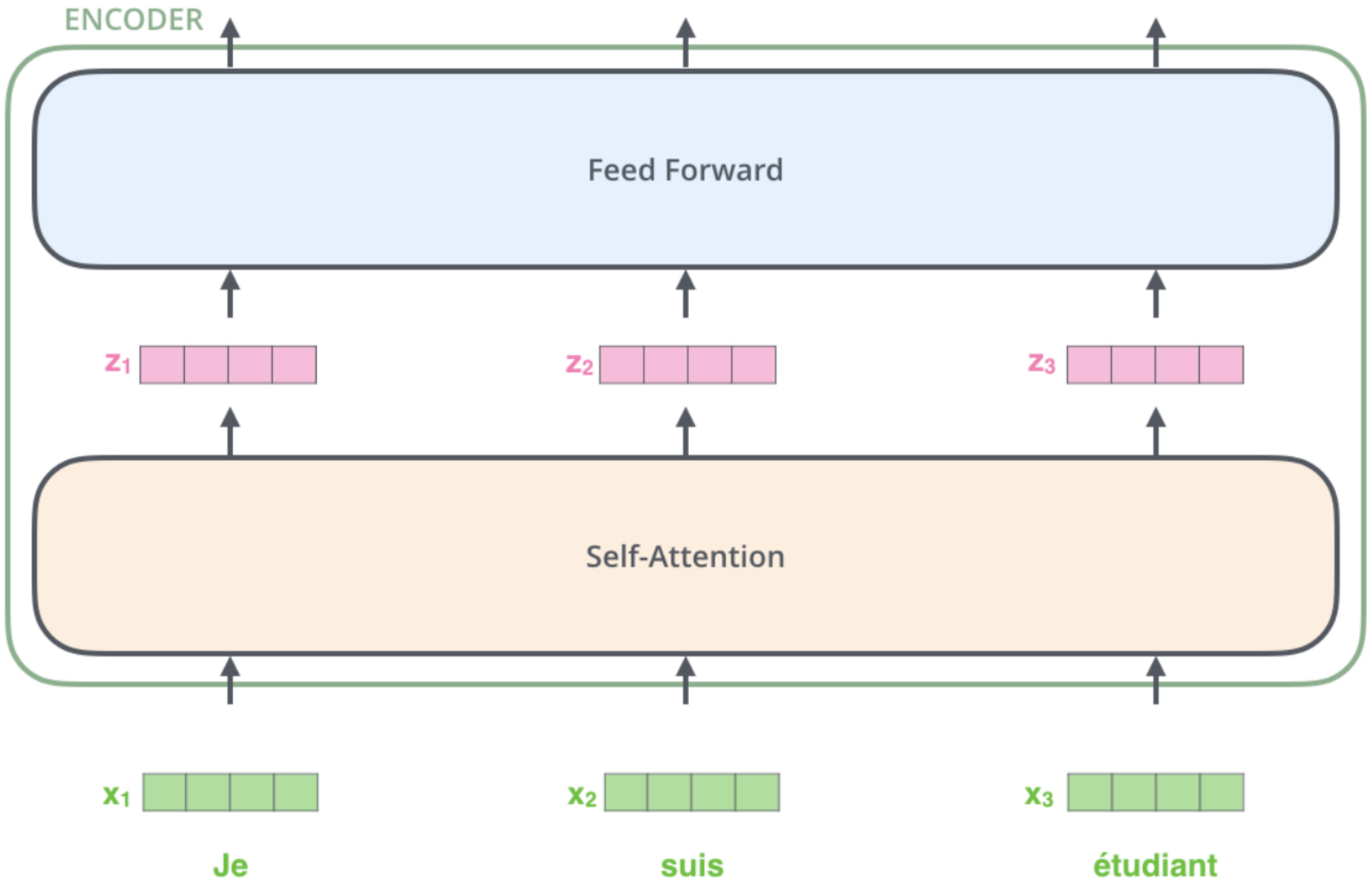


Transformer: decoder

- the same as encoder but with an additional attention layer in between, receiving input fom encoder (called encoder-decoder attention)



Start with embeddings

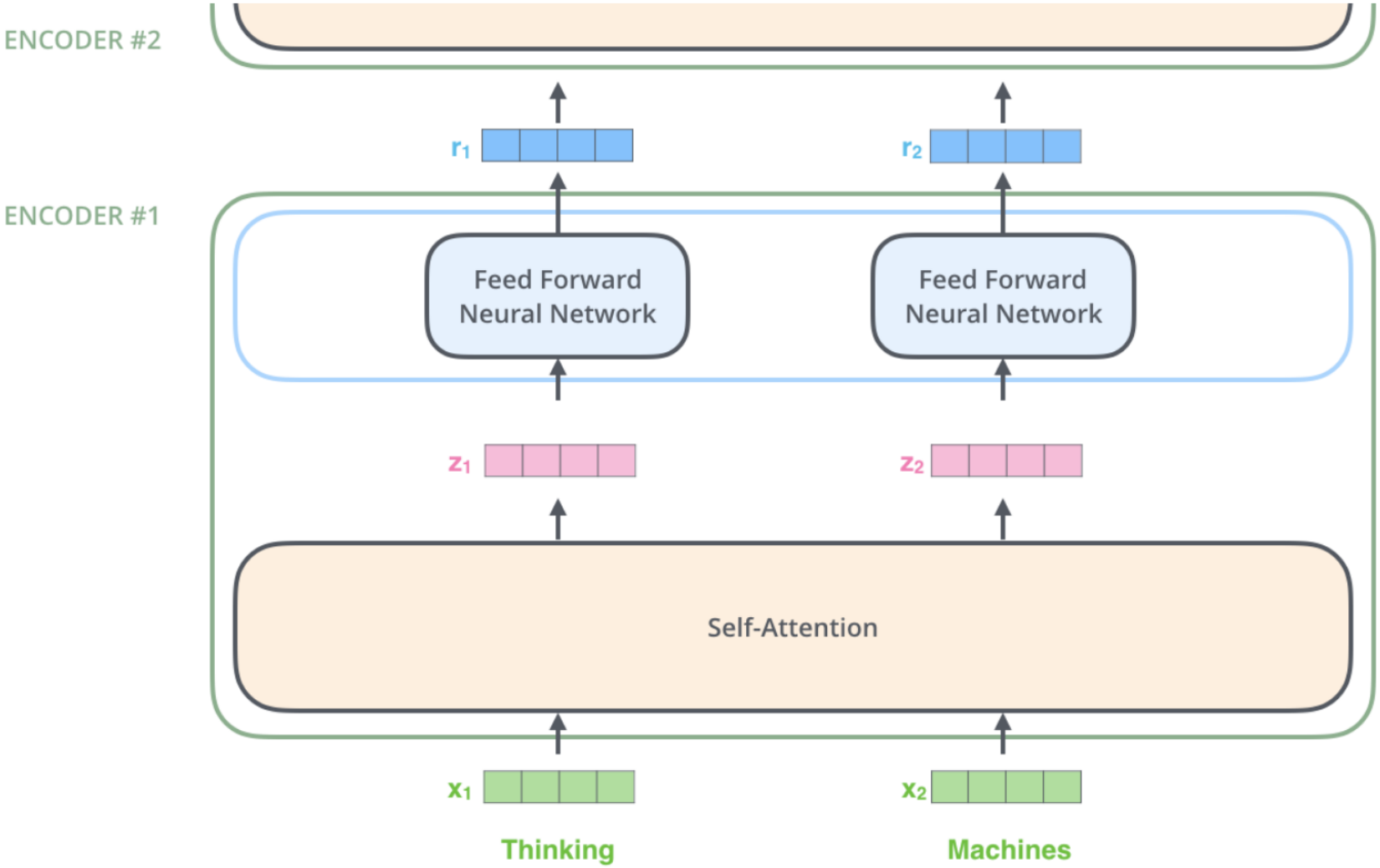


Input to transformer

- embeddings, e.g., 512 dimensional vectors (special, we will discuss that later)
- fixed length, e.g., max 128 tokens
- dependencies between inputs are only in the self-attention layer, no dependencies in feed forward layer – good for parallelization

- **Let us first present the working of the transformer with illustration of the prediction, later we will cover also training.**

Encoding


ENCODER #2
r1
r2
ENCODER #1
Feed Forward Neural Network
Feed Forward Neural Network
z1
z2
Self-Attention
x1
x2
Thinking
Machines

Self-attention

- As the model processes each word (each position in the input sequence), self-attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word

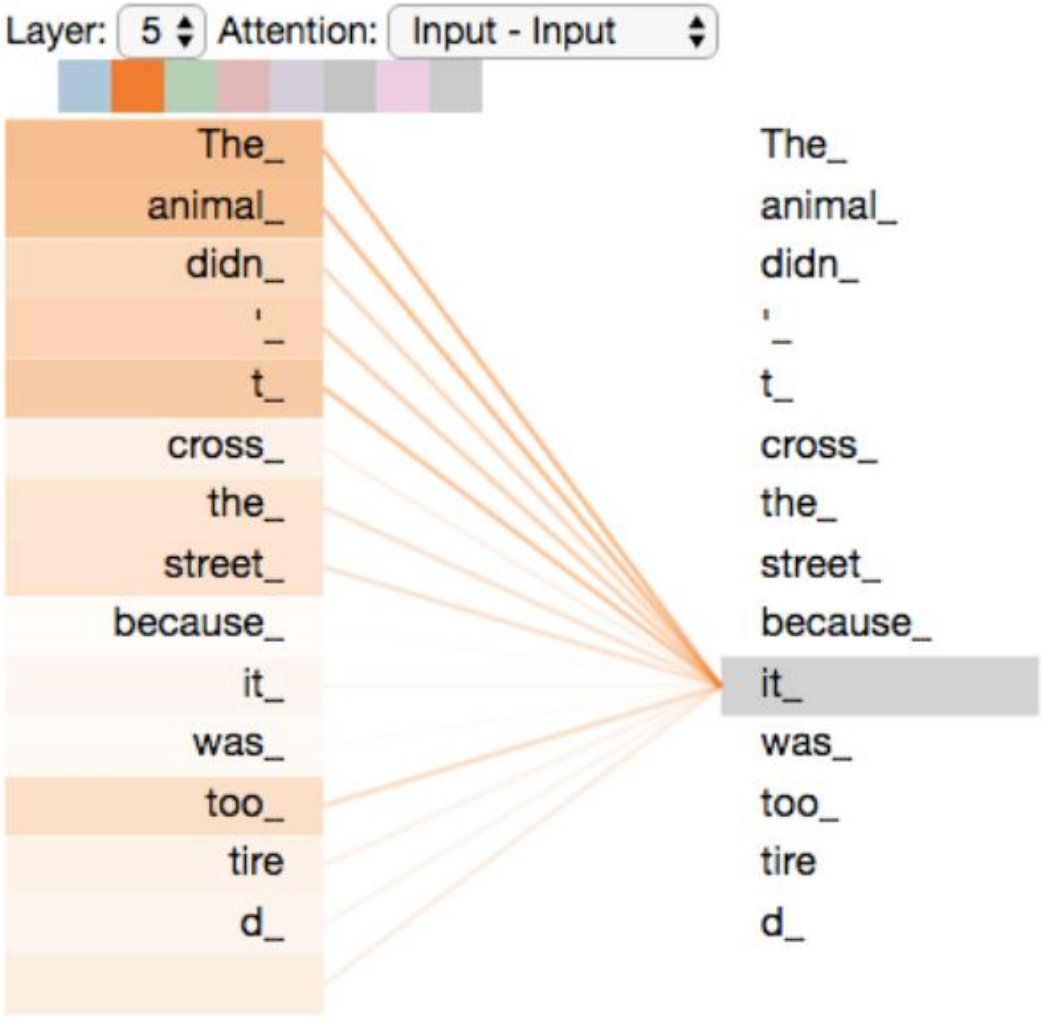
  ”The animal didn't cross the street because it was too tired”

- What does “it” in this sentence refer to? Is it referring to the street or to the animal? It’s a simple question to a human, but not as simple to an algorithm.
- *”The animal didn't cross the street because it was too wide”*

- When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

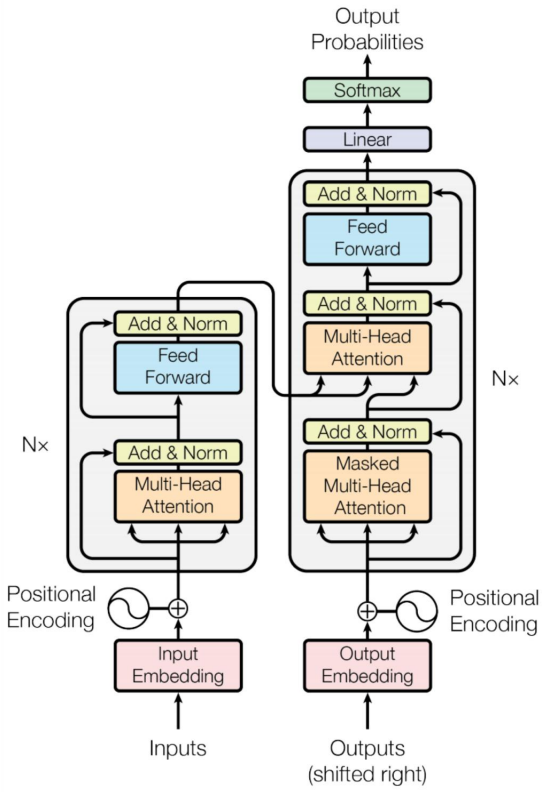
Illustrating self-attention

- As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

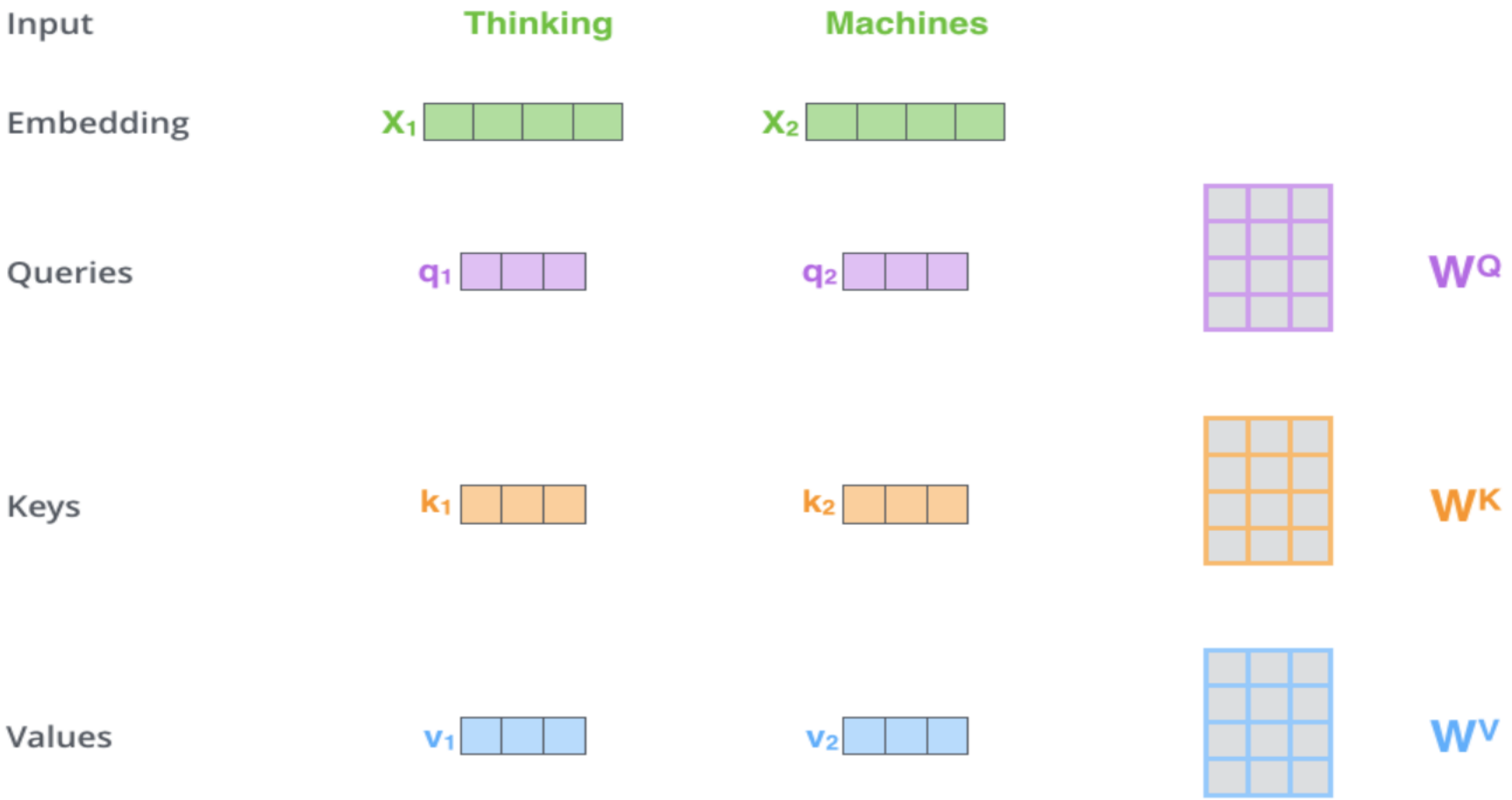


Self-attention details 1/4

- create three vectors from each of the encoder's input vectors (in the first layer, the input vectors are the embedding of each word).
- Query vector Q, Key vector K, Value vector V are created by multiplying the embedding by three matrices that are learned during the training process.
- Q, K, and V are smaller than the embedding vector, typically 64, while the embedding and encoder input/output vectors have a dimensionality of 512.
- They are smaller to make the computation of multiheaded attention (mostly) constant.



Self-attention details 1/4

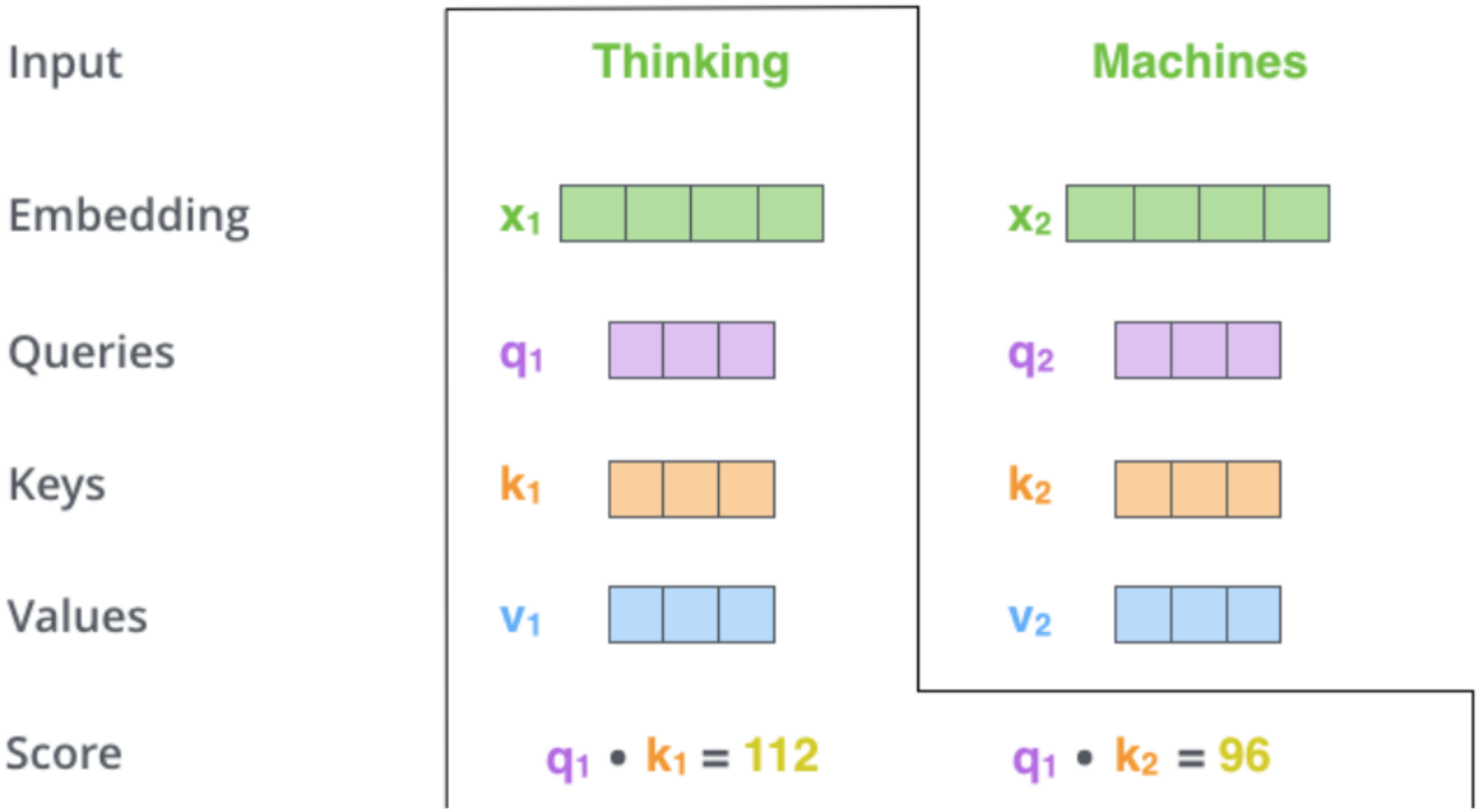


Multiplying x_1 by the W^Q weight matrix produces q_1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

Details 2/4: attention vectors Q, K, V

- The “query” Q, “key” K, and “value” V vectors are abstractions that are useful for calculating and thinking about attention.
- To calculate self-attention for a given word (e.g., “Thinking”), we score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.
- The score is calculated by taking the dot product of the query vector Q with the key vector K of the respective word.
- E.g., on computing the self-attention for the word in position #1, we would compute dot product of q_1 and $k_{1,}$ and q_1 and k_2.

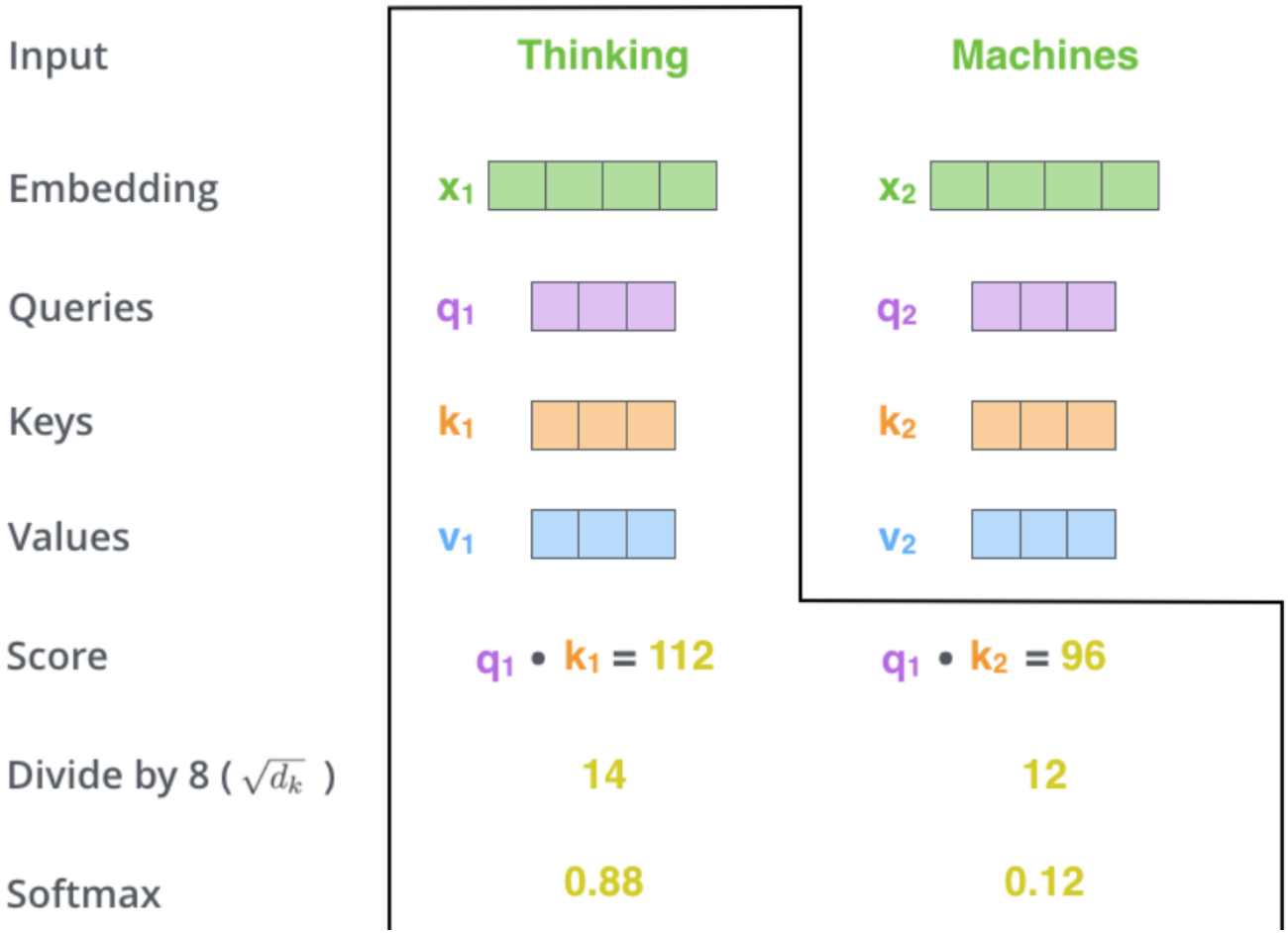
Details 2/4: scoring



Details 3/4: normalize scores

- divide the scores by the square root of the dimension of the key vectors used (in example, the vectors are of dimension 64, therefore divide by 8)
- This leads to more stable gradients.
- Then pass the result through a softmax operation. Softmax normalizes the scores so they’re all positive and add up to 1.

Details 3/4: normalization of scores



Details 4/4: apply attention scores

- The softmax score determines how much each word will be expressed at this position. Usually the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.
- Next multiply each value vector by the softmax score (in preparation to sum them up). The intuition is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them with small scores, e.g., 0.001).
- End computation by summing up the weighted value vectors. This produces the output of the self-attention layer at this position (for the given word – the first one in the example).
- The resulting vector is send to the feed-forward neural network.
- In the actual implementation, however, the calculation is done in matrix form for faster processing.

Details 4/4: self-attention output

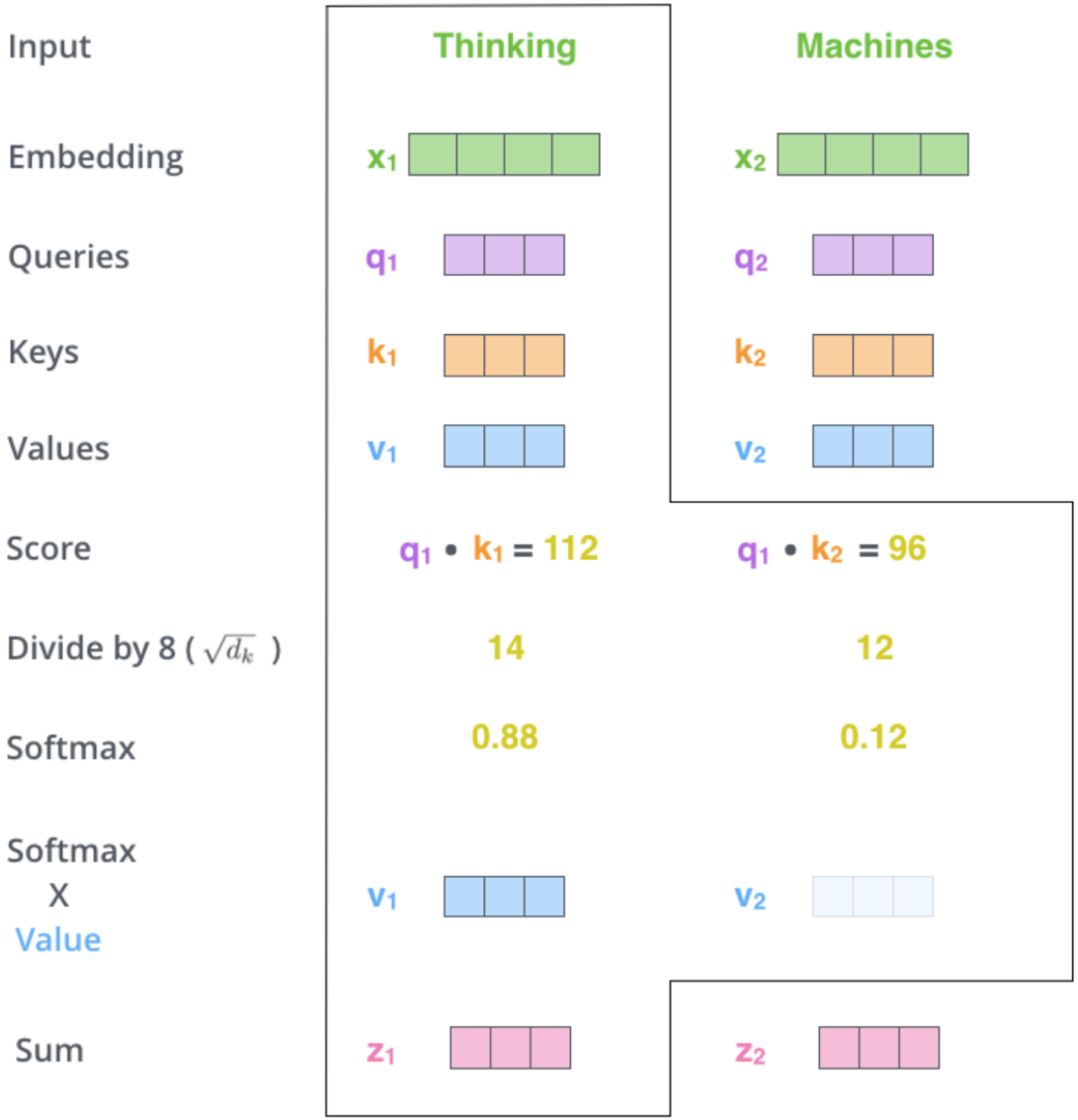
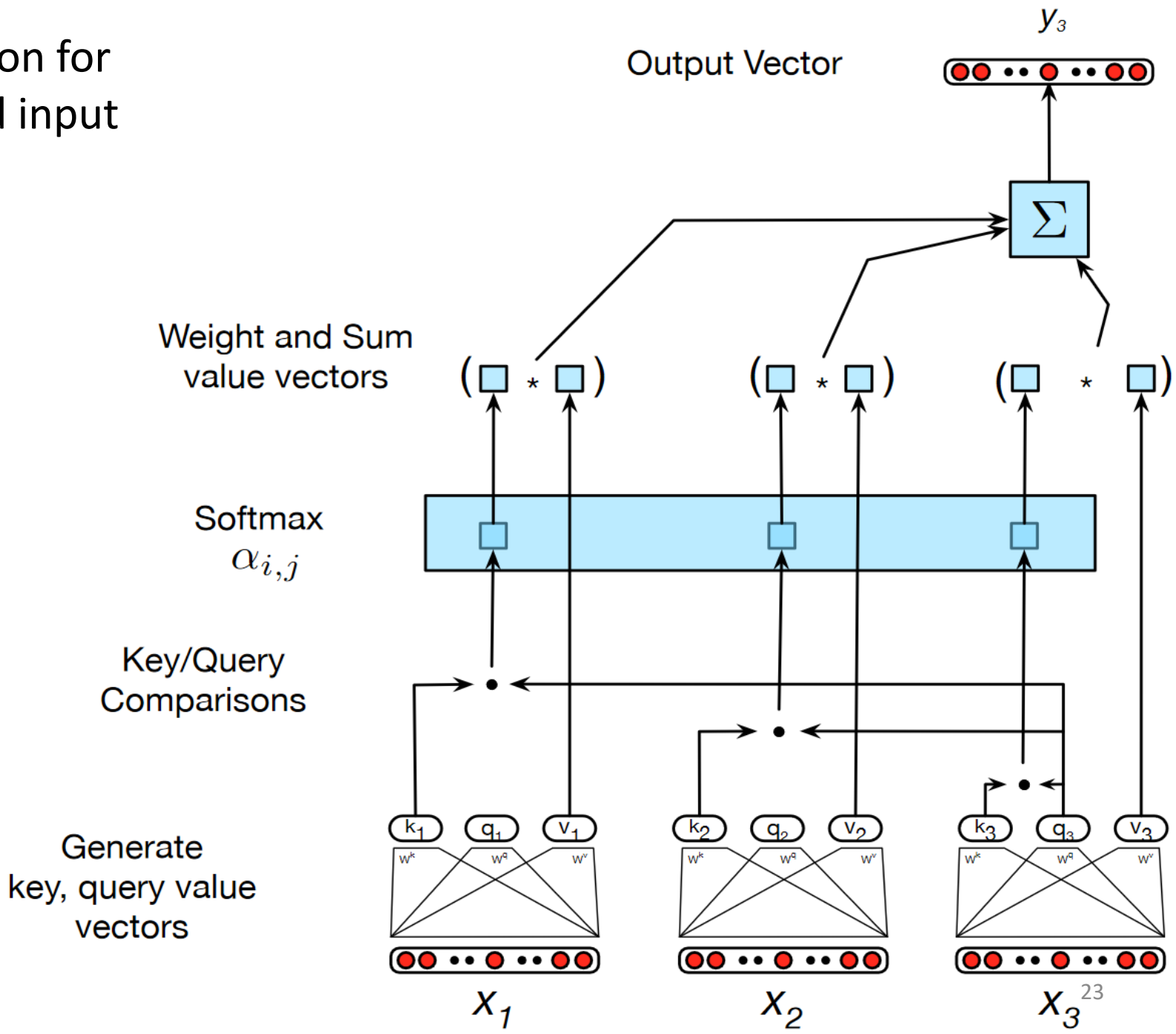
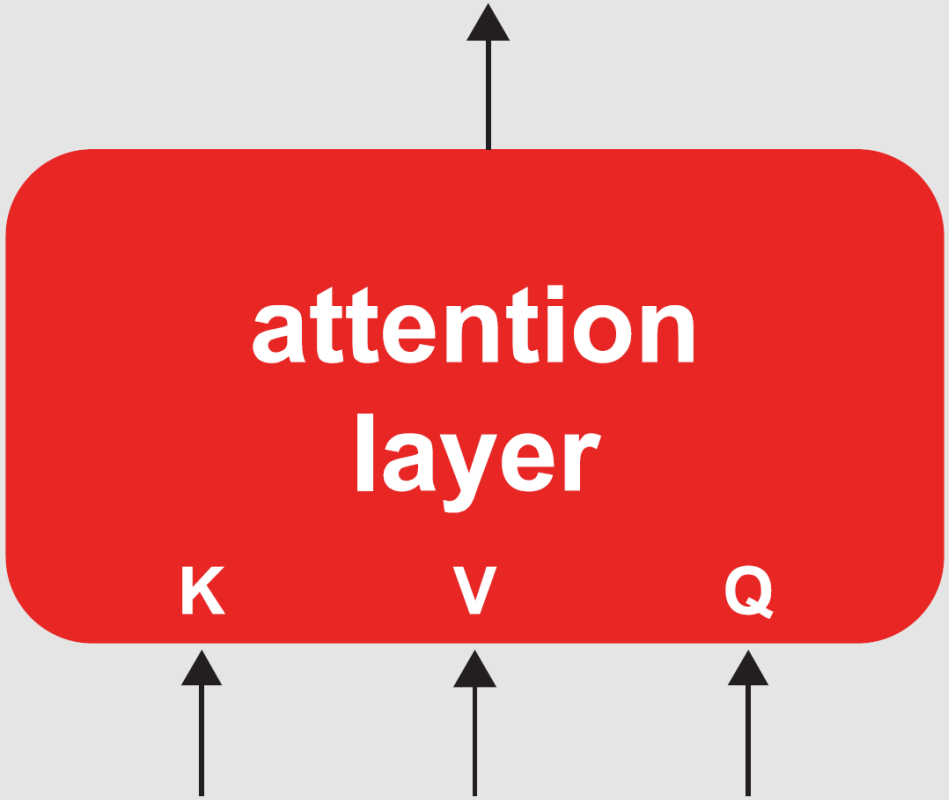


illustration for
the third input



$$Y_i = \sum_{j=1}^{n} \alpha_{ij} V_j$$

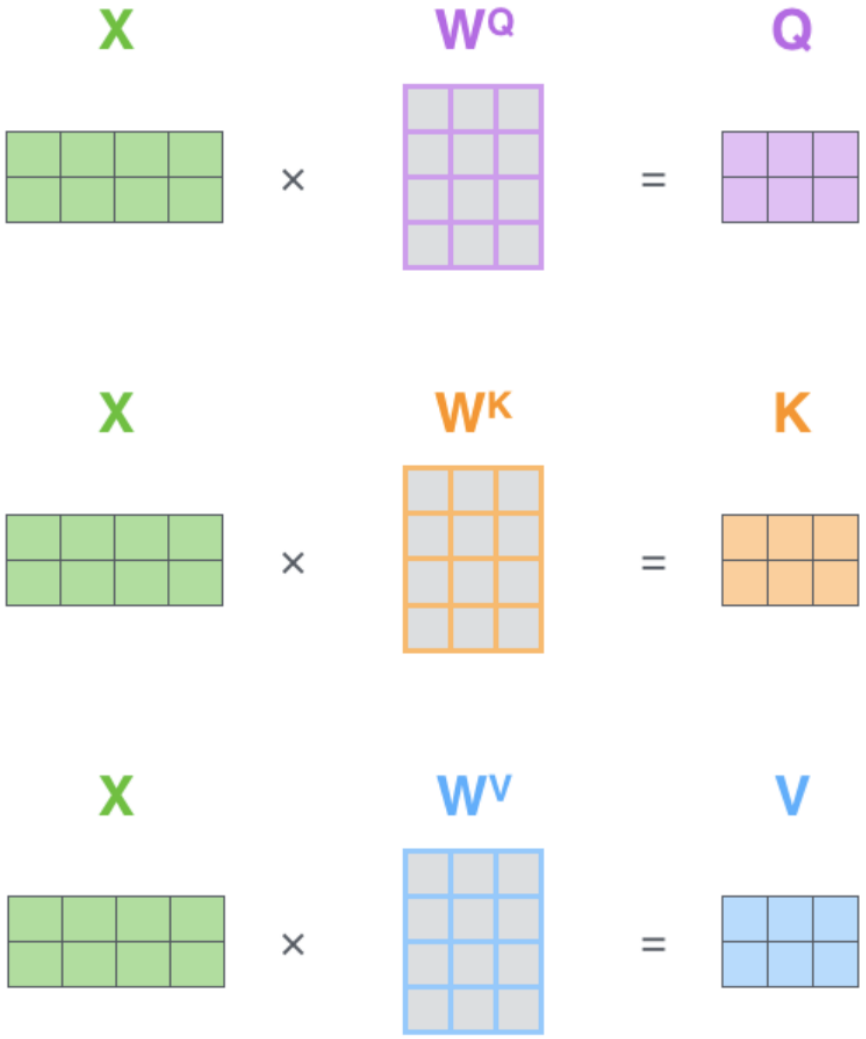


$$sim_{ij} = \phi(Q_i, K_j)$$

$$\alpha_{ij} = \frac{exp(sim_{ij})}{\sum_{k=1}^{n} exp(sim_{ik})}$$

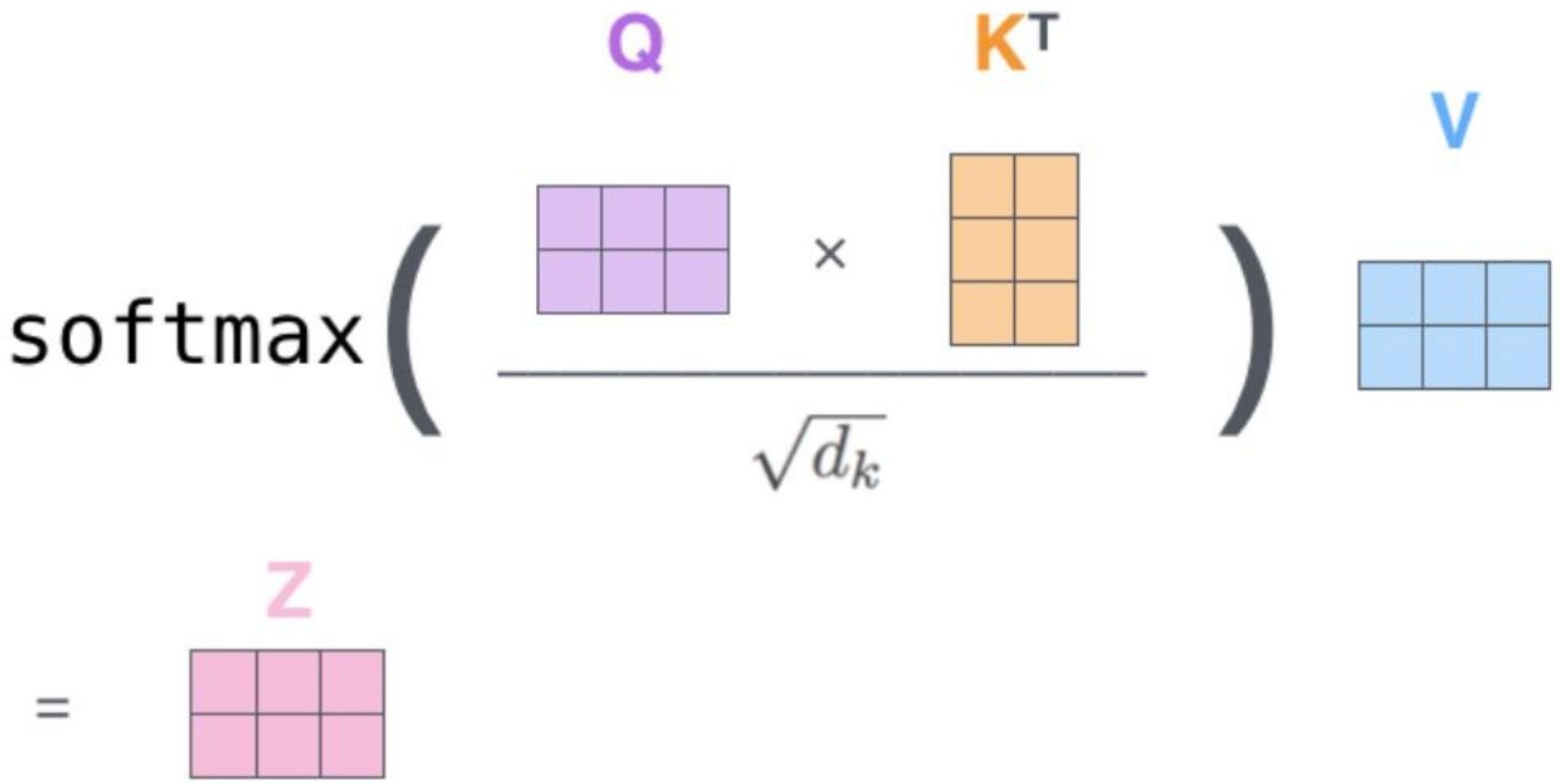
Matrix calculation of self-attention 1/2

Every row in the X matrix corresponds to a word in the input sentence.
The embedding vector x (512) is larger then the q/k/v vectors (64)



Matrix calculation of self-attention 2/2

- final calculation

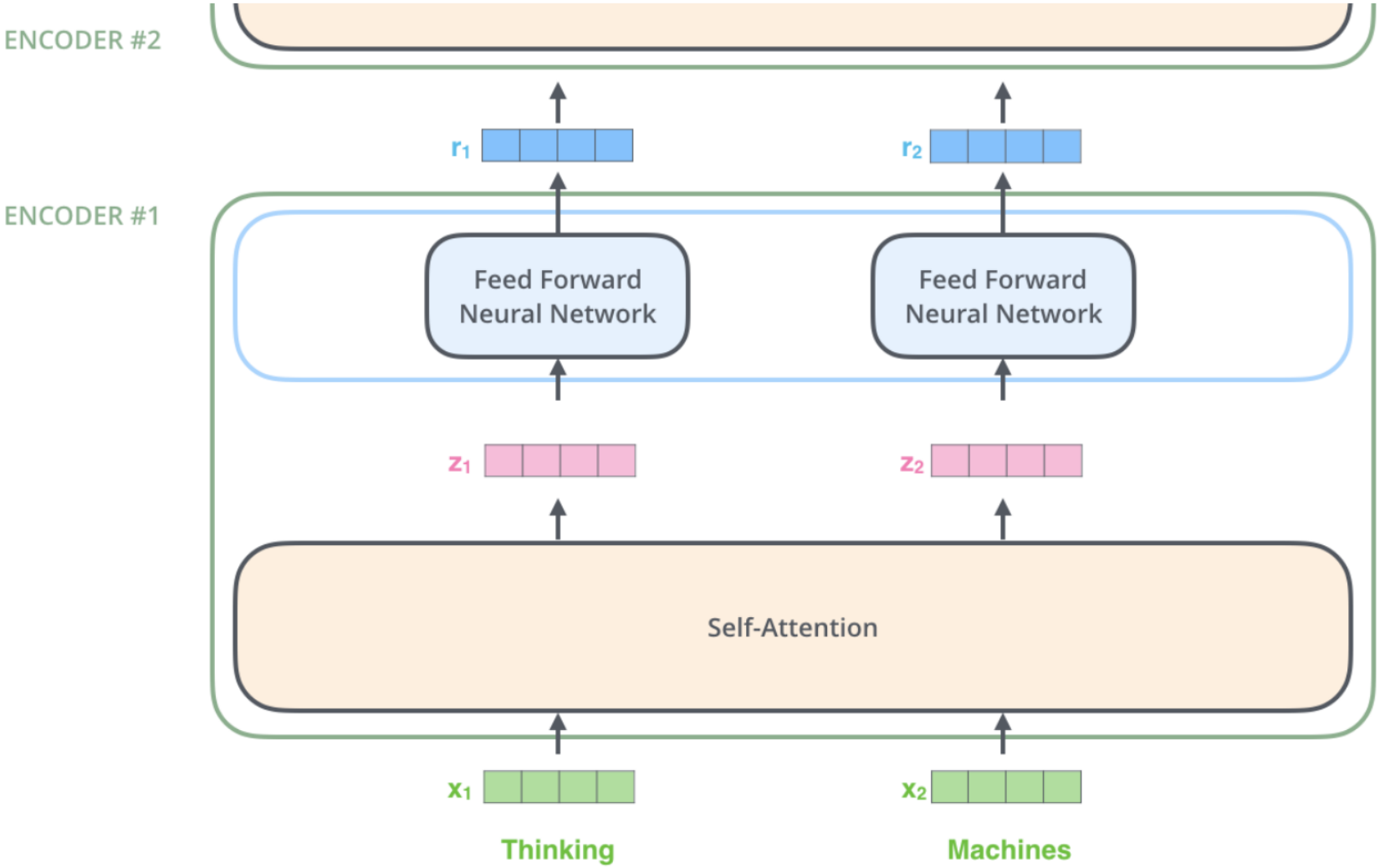


Computing attention head

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

$$A(Q, K, V) = softmax(QK^T)V$$

Encoding

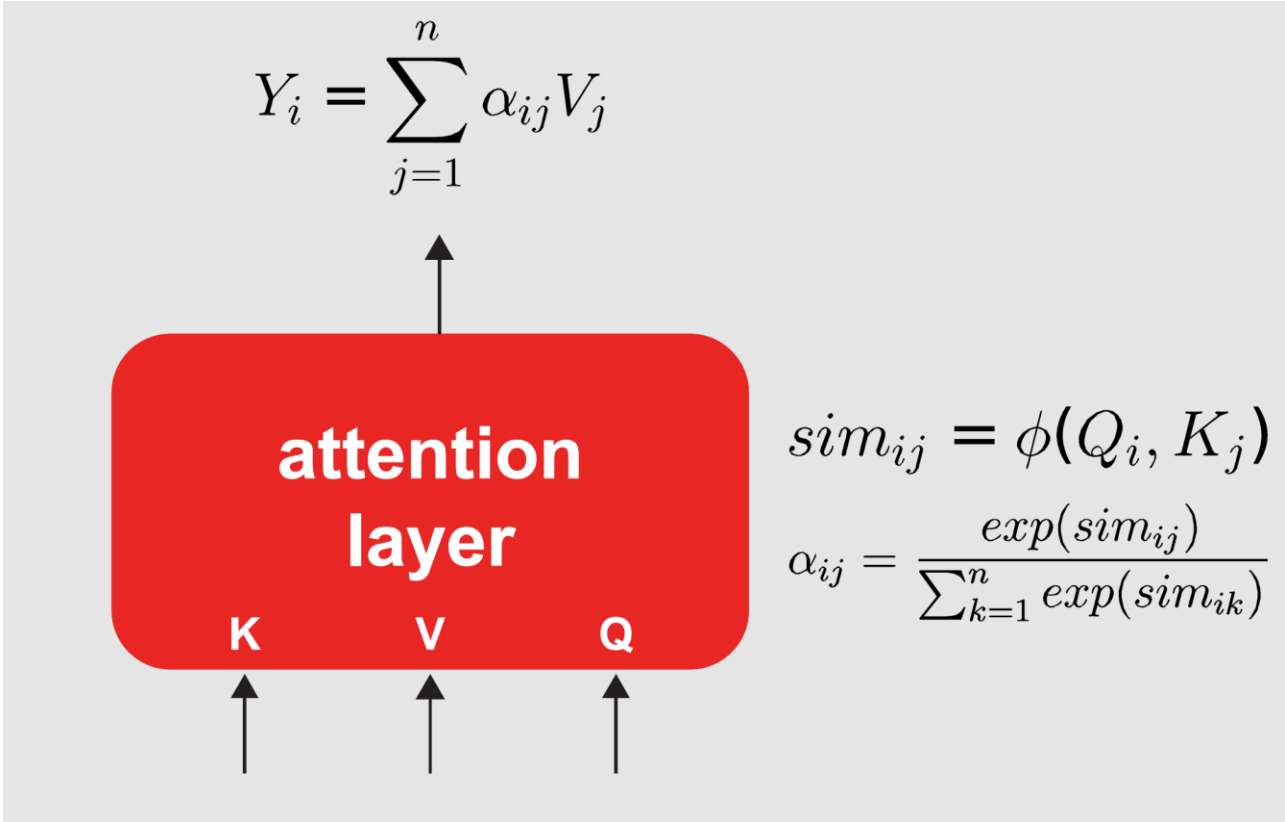

ENCODER #2
ENCODER #1
r1
r2
Feed Forward Neural Network
Feed Forward Neural Network
z1
z2
Self-Attention
x1
x2
Thinking
Machines

In summary 1/3

Self-attention

- recall attention:
summary of encoder states based on similarity between a particular decoder state and encoder states

- Transformer generalizes:
summary of values V based on similarity between a particular query Q_i and keys K



In summary 2/3

Self-attention

Notation:

n - input length

d_m - representation dim.

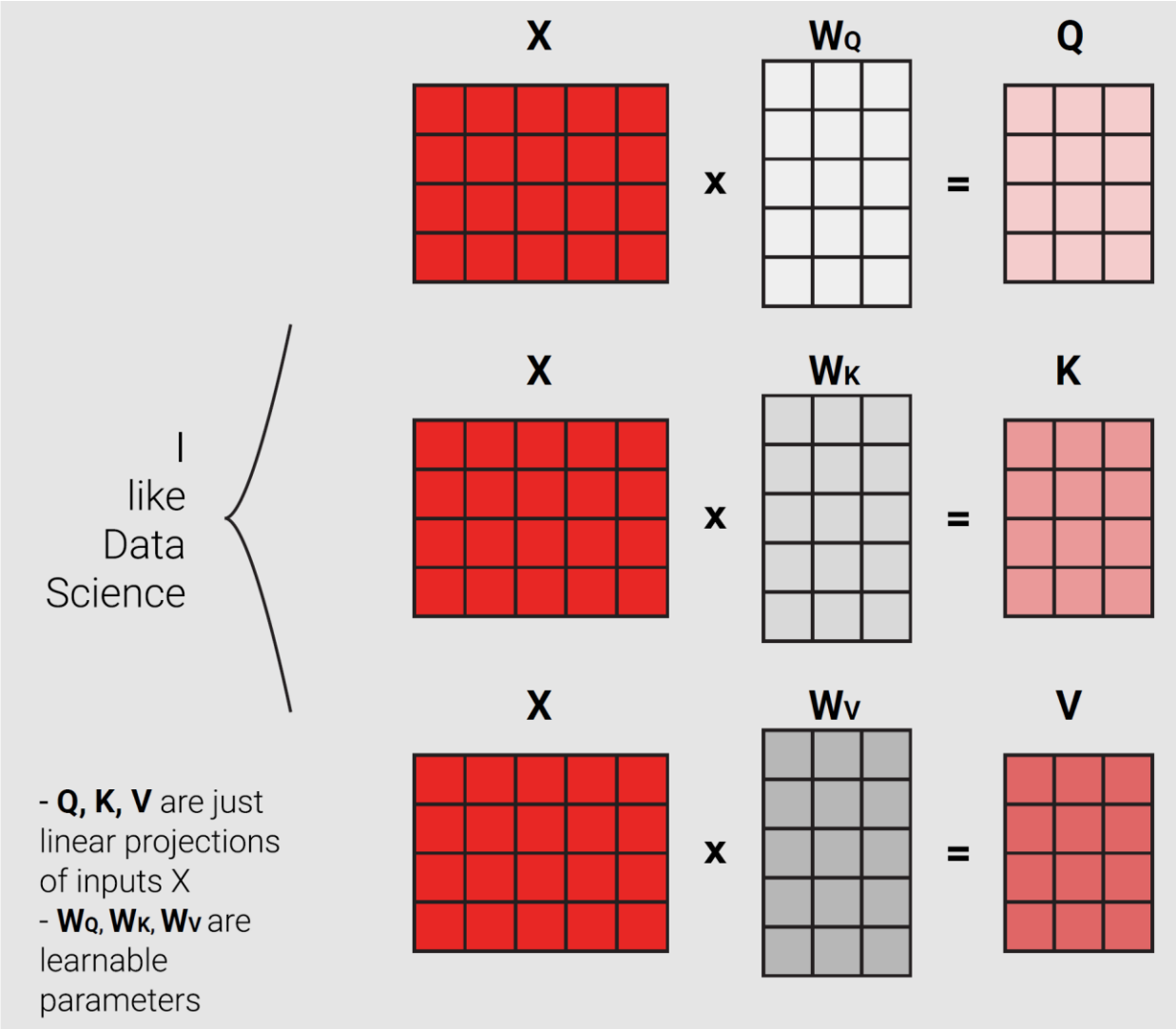
d - dimension of queries, keys, values

Example:

n = 4

d_m = 5

d = 3



In summary 3/3

Self-attention

- scaled dot-product attention;

- with larger d the variance of dot products increases, softmax gets more peaked, its gradient gets smaller;

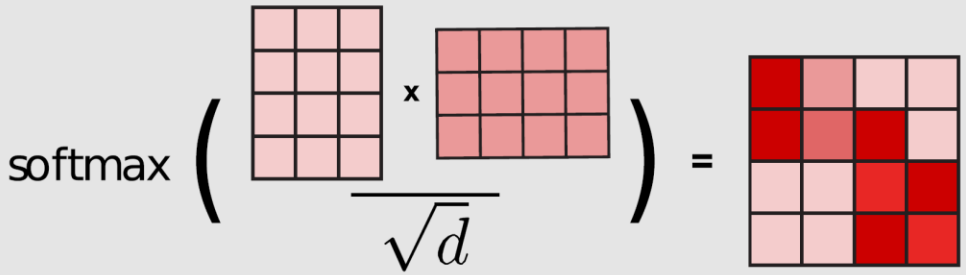
- dividing by $\sqrt{d}$ mitigates this effect.

$$sim_{ij} = \frac{Q_i^T K_j}{\sqrt{d}} \qquad \alpha_{ij} = \frac{exp(sim_{ij})}{\sum_{k=1}^{n} exp(sim_{ik})} \qquad Y_i = \sum_{j=1}^{n} \alpha_{ij} V_j$$

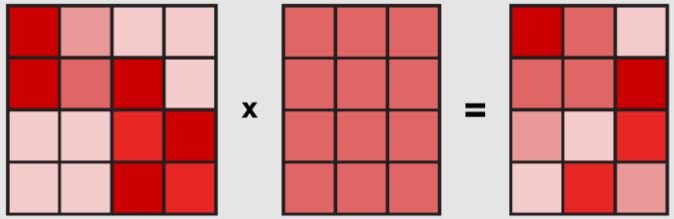
Transformer attention

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d}})V$$

matrix formulation



attention weights

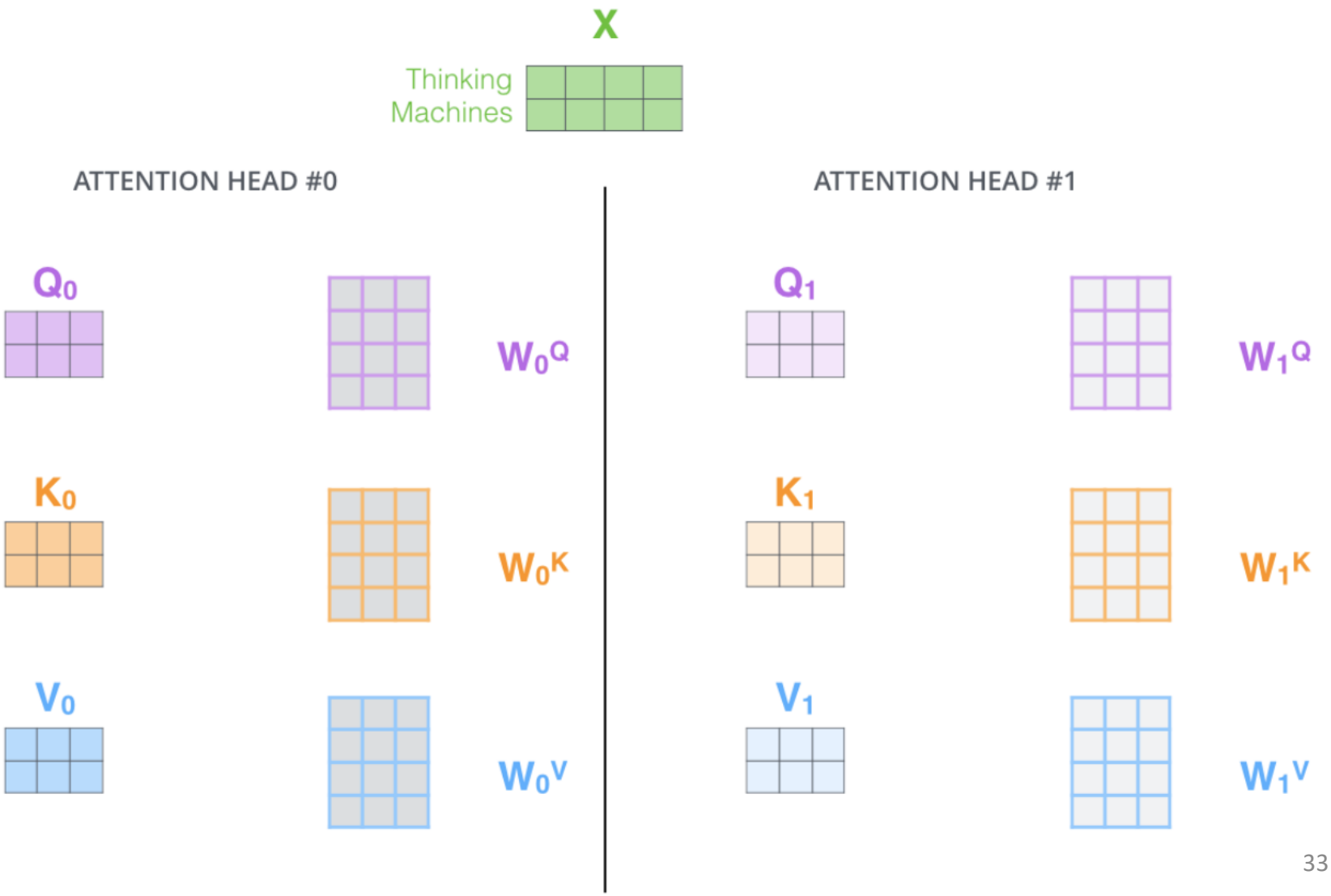


output

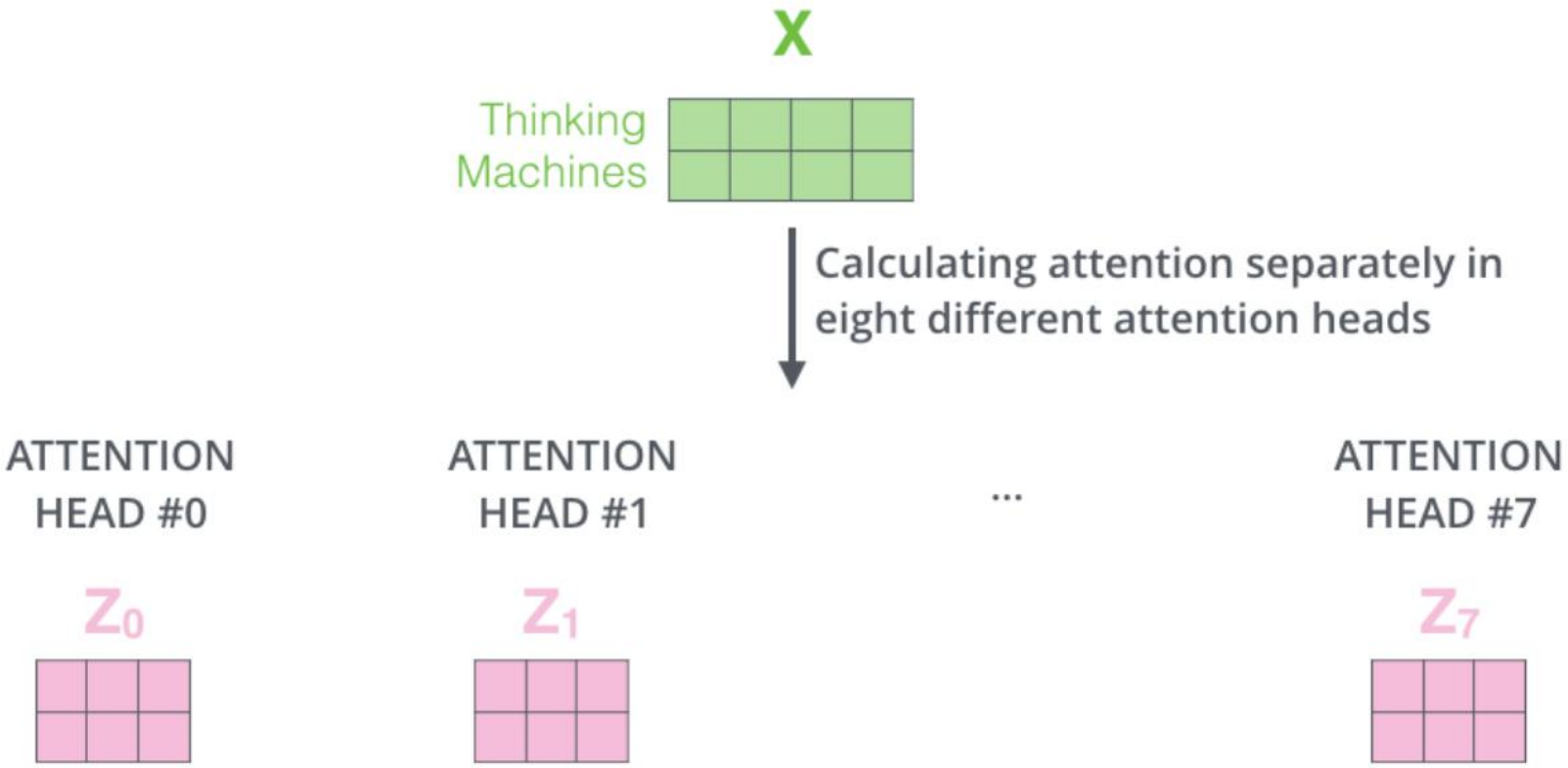
Multi-headed attention

- Self-attention layer is replicated several times, called “multi-headed” attention.
- This expands the model’s ability to focus on different positions. E.g., one attention head might be dominated by the the actual word, but other heads might reveal other important information
- E.g., in translating a sentence like “The animal didn’t cross the street because it was too tired”, we would want to know which word “it” refers to.
- Multi-head attention layer can cover multiple “representation subspaces”
- I.e., we have multiple sets of Query/Key/Value weight matrices (original Transformer uses 8 attention heads)
- Each attention head is randomly initialized. After training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

Example: two attention heads



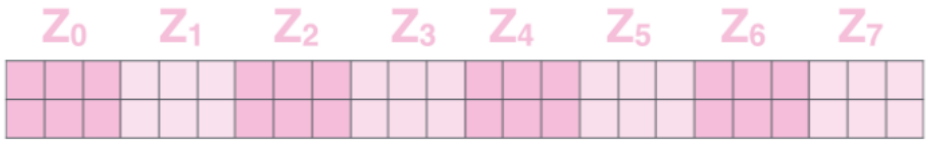
Example: 8 att. heads



- What to do with 8 Z matrices, the feed-forward layer is expecting a single matrix (one vector for each word). We need to condense all attention heads into one matrix.

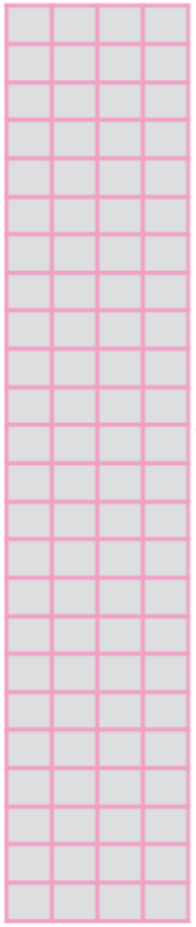
Condensation of attention heads

1) Concatenate all the attention heads



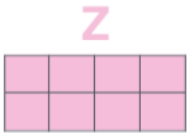
2) Multiply with a weight matrix W^O that was trained jointly with the model

X



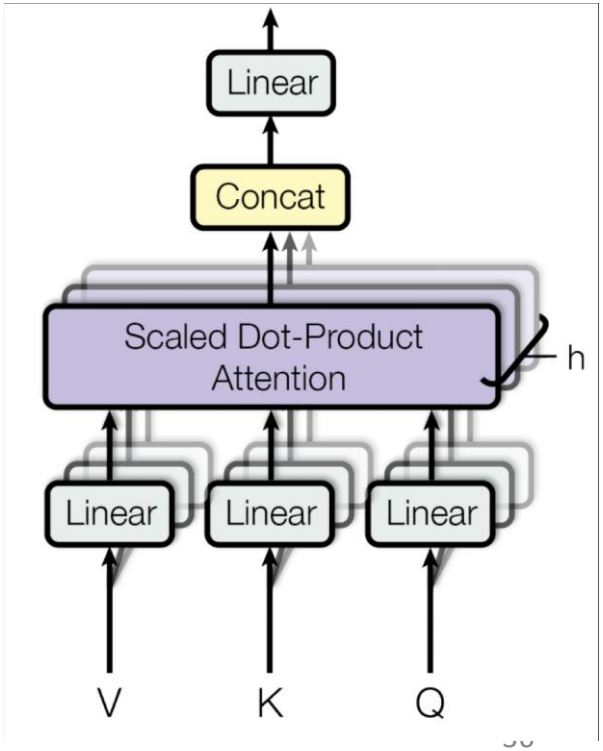
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

=



Computing multi-head attention

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



Summary of self-attention

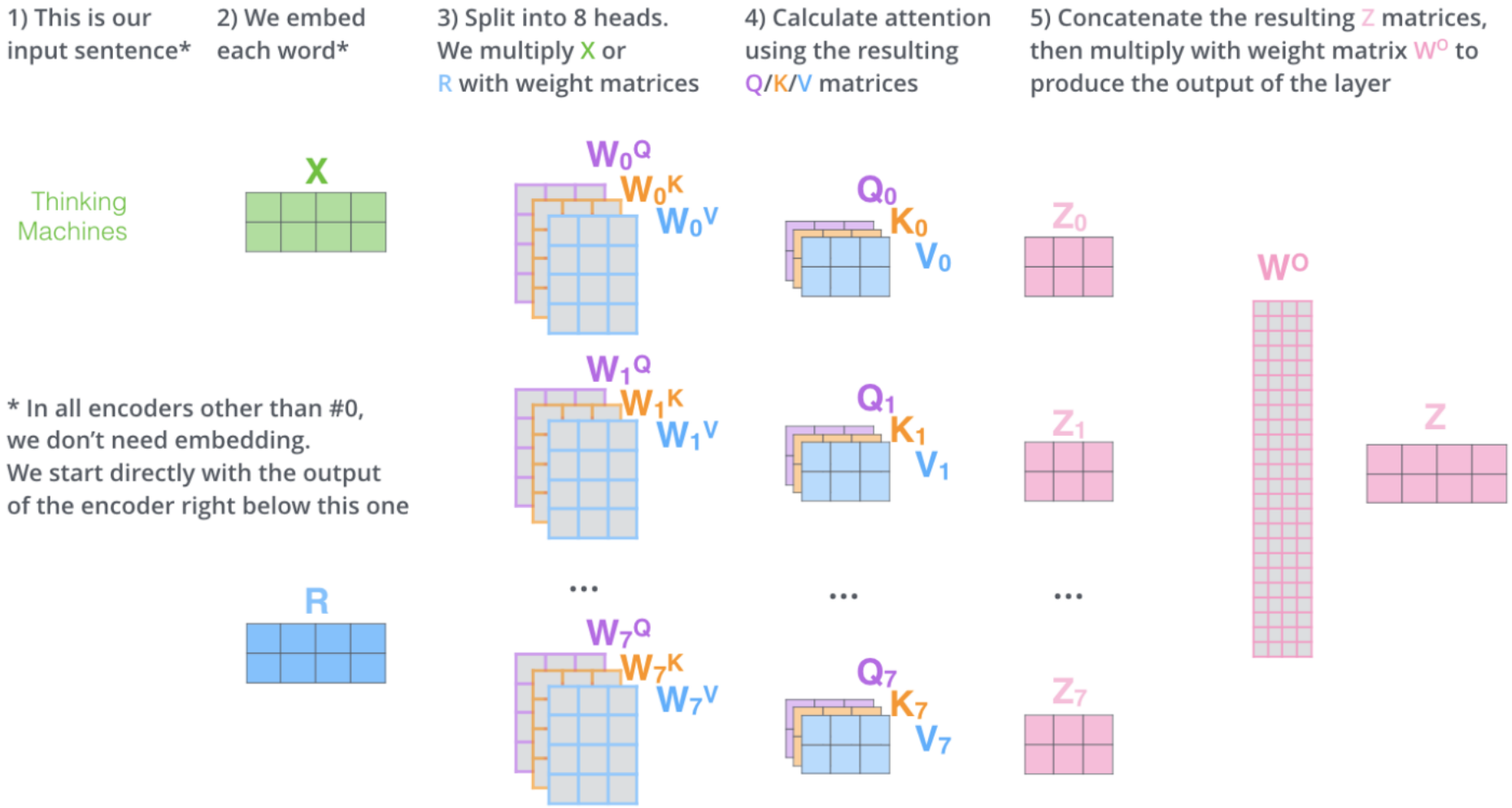


Illustration of self-attention: 1 head

- encoder #5 (the top encoder in the stack)
- As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

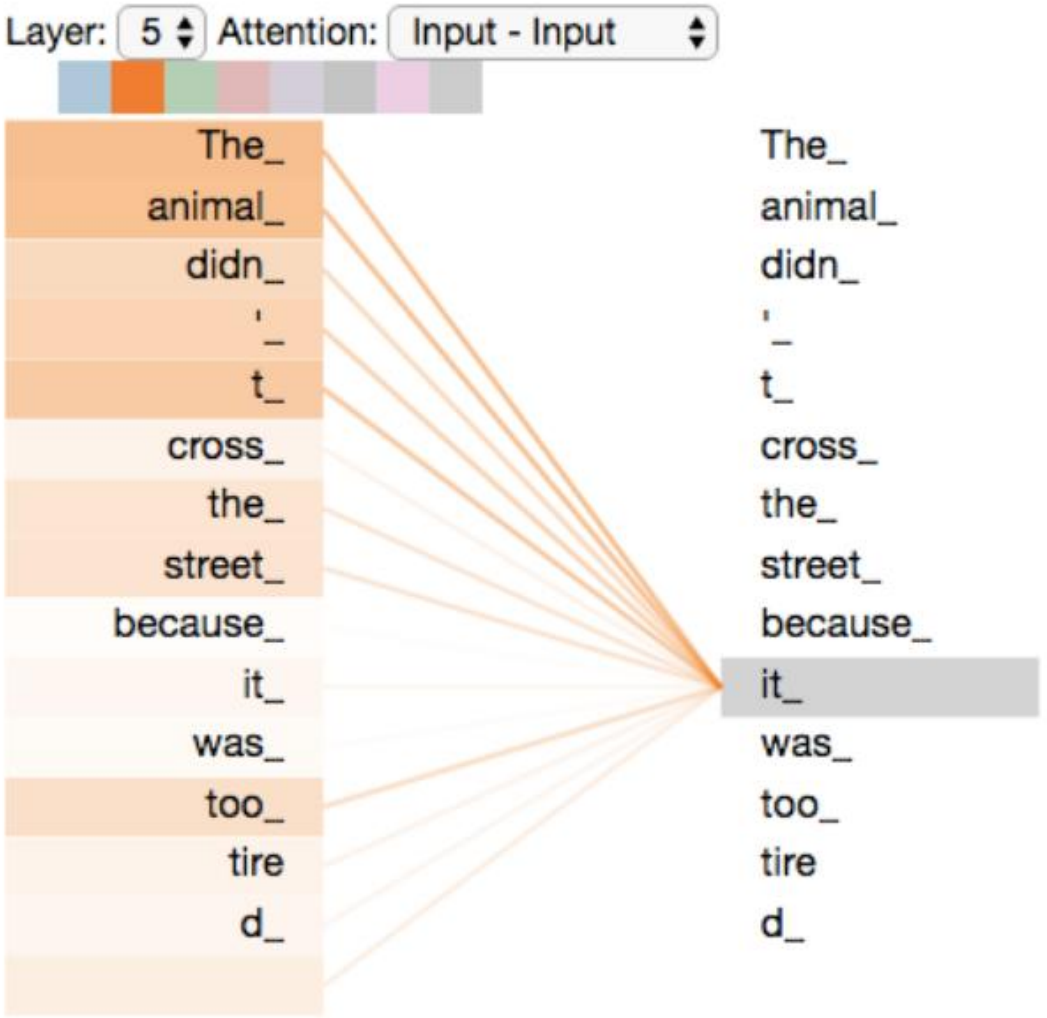
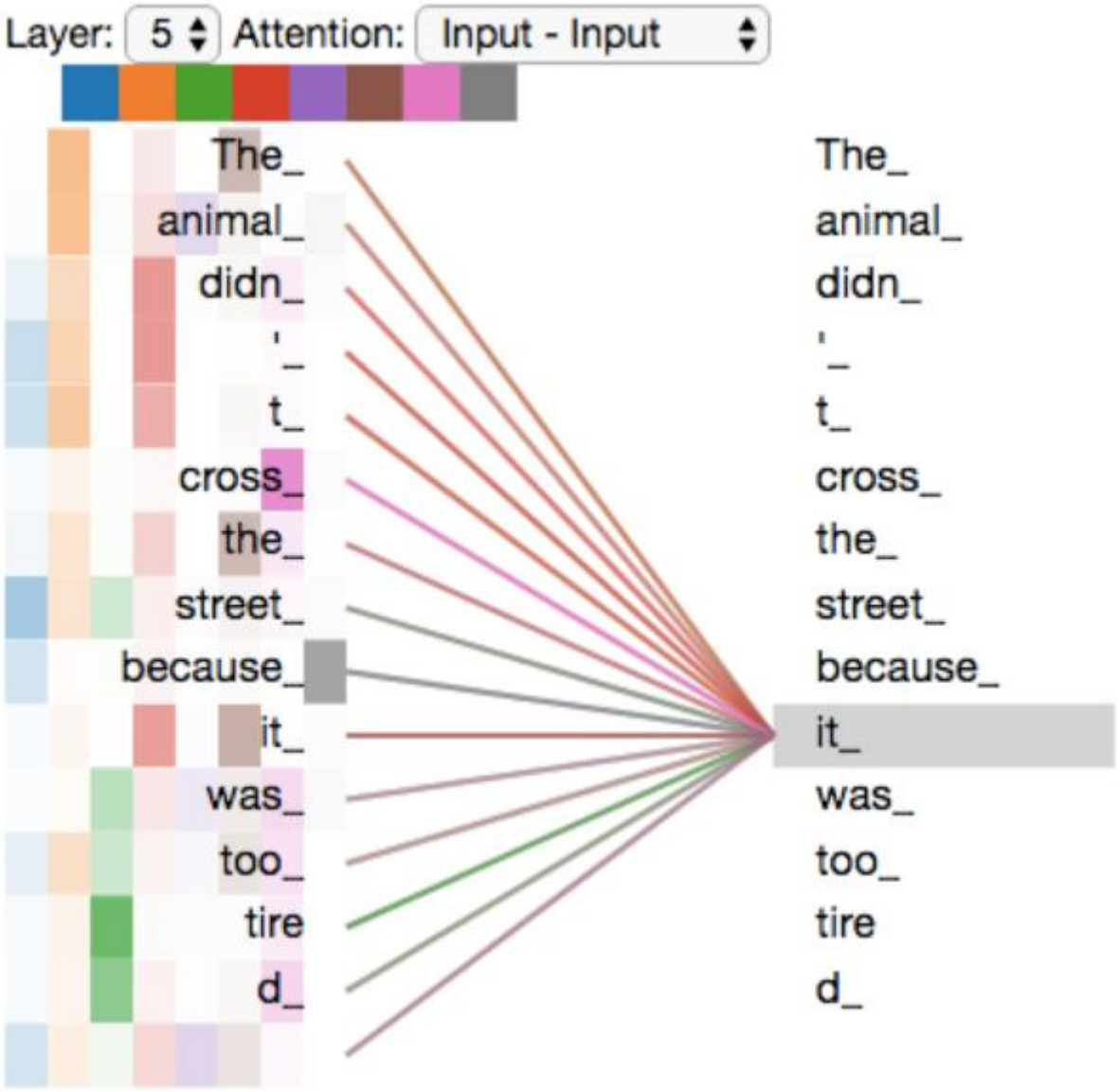


Illustration of self-attention: all heads

- all the attention heads in one picture are harder to interpret

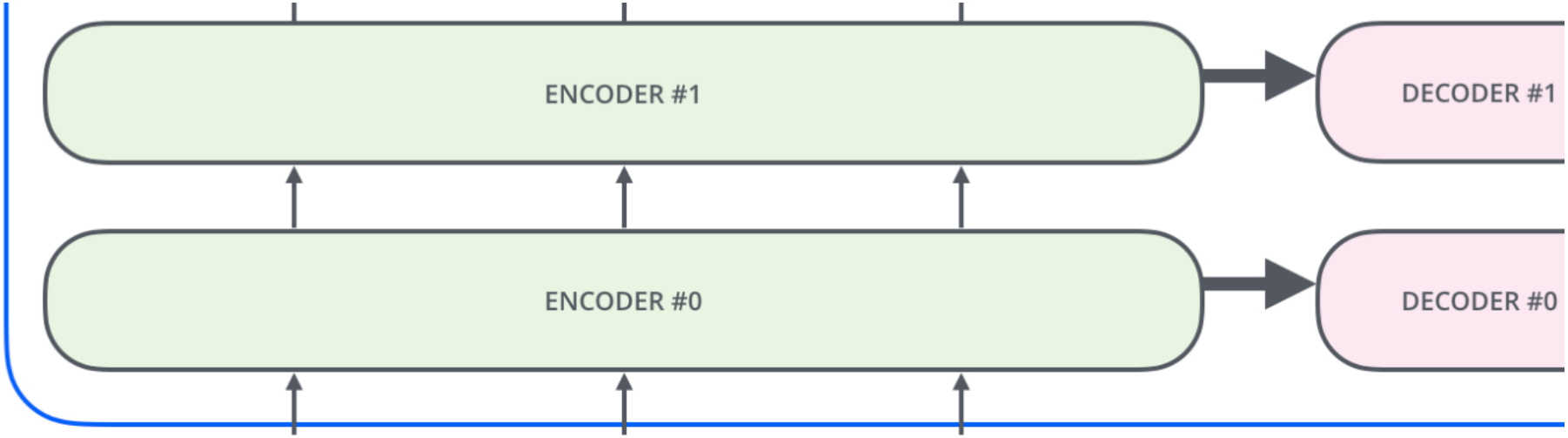


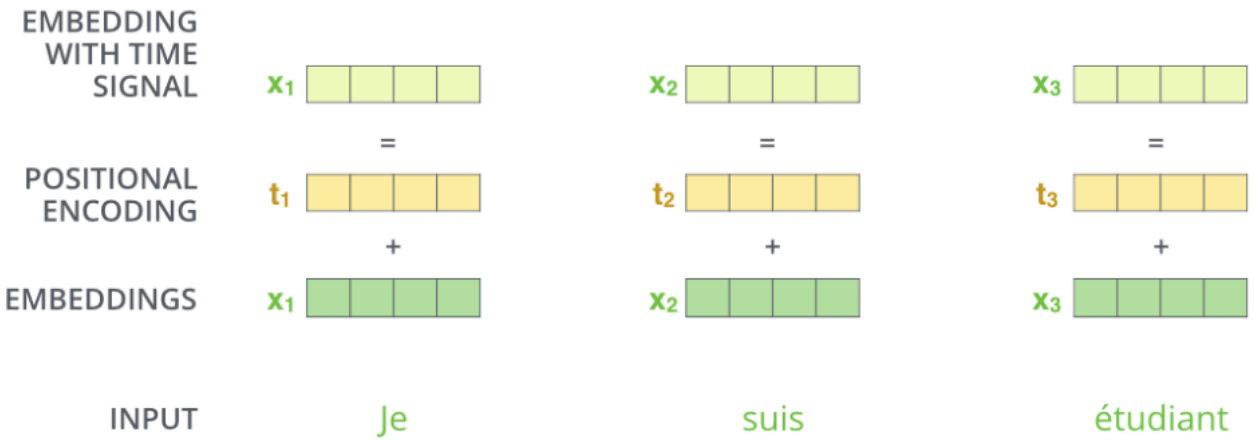
Animated workings of the transformer attention mechanism

Representing the order of the sequence using positional encoding

- Order of the sequence is important, but it is lost with the described transformation, therefore
- The transformer adds a position vector to each input embedding.
- These vectors follow a specific pattern that the model learns, which helps it to determine the position of each word, or the distance between different words in the sequence.
- Adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during the dot-product attention.

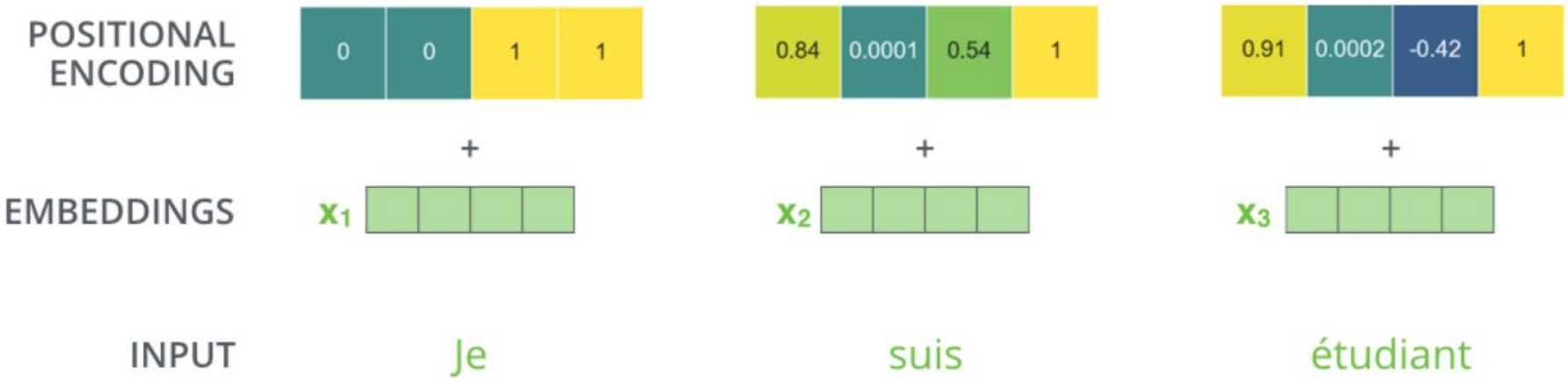
Adding position encoding





Example: encoding position

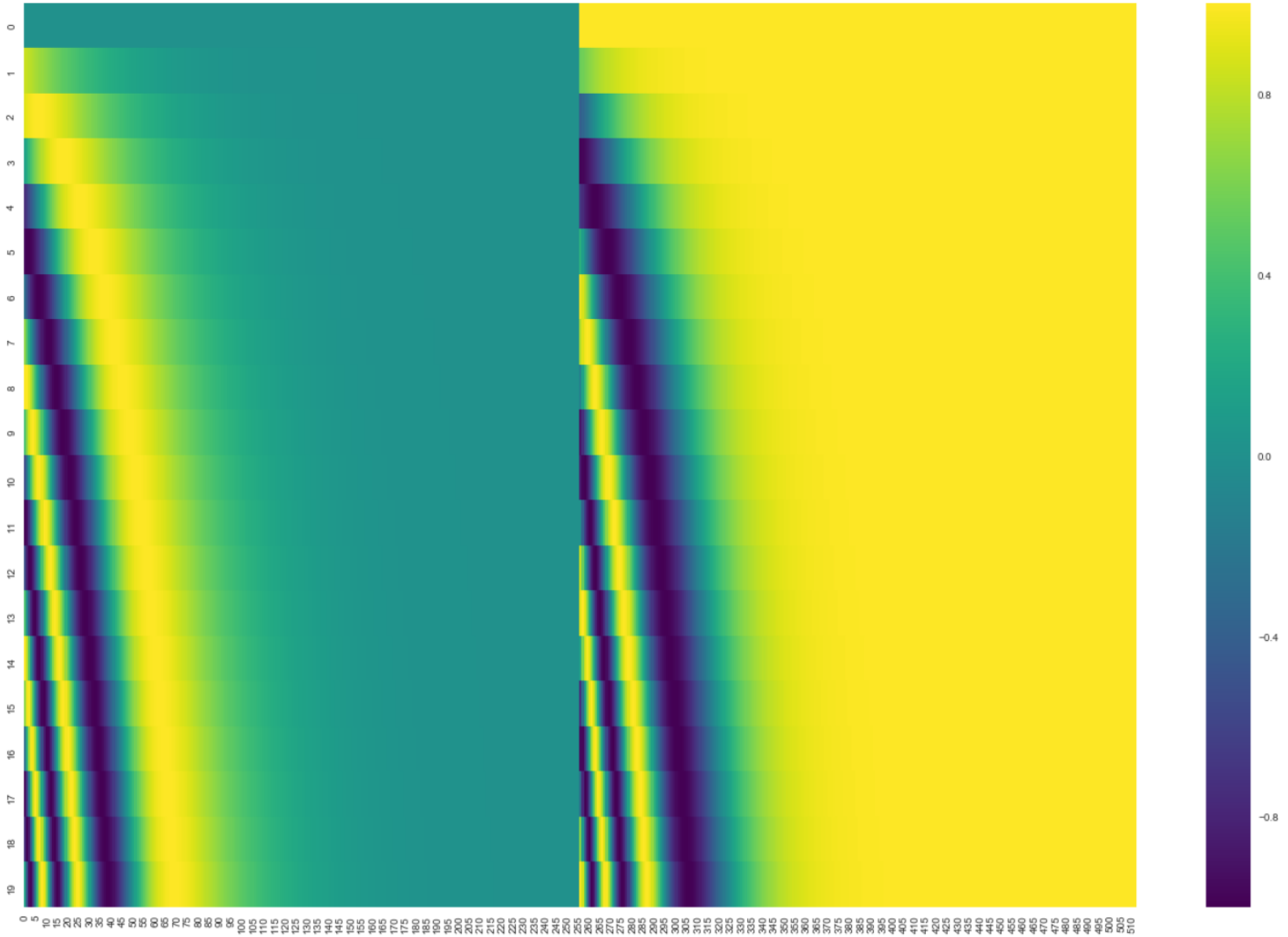
- the values of positional encoding vectors follow a specific pattern.



Patterns of positional encodings

- many different possibilities how to generate a position pattern
- The next slide contains an example of positional encoding for 20 words (rows) with an embedding size of 512 (columns).
- the values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors

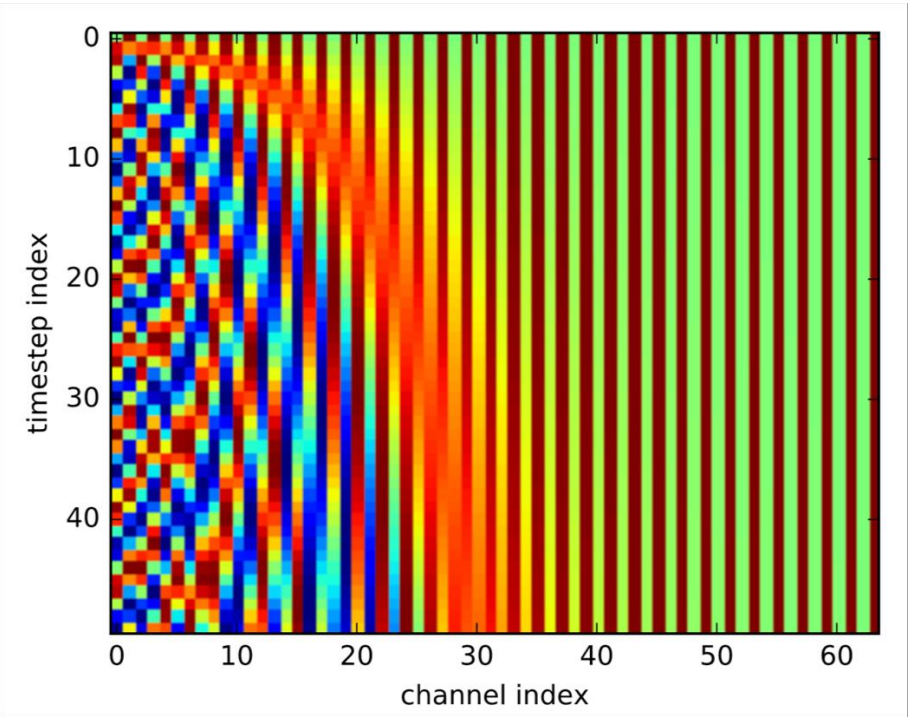
Example of positional encoding



Another positional encoding

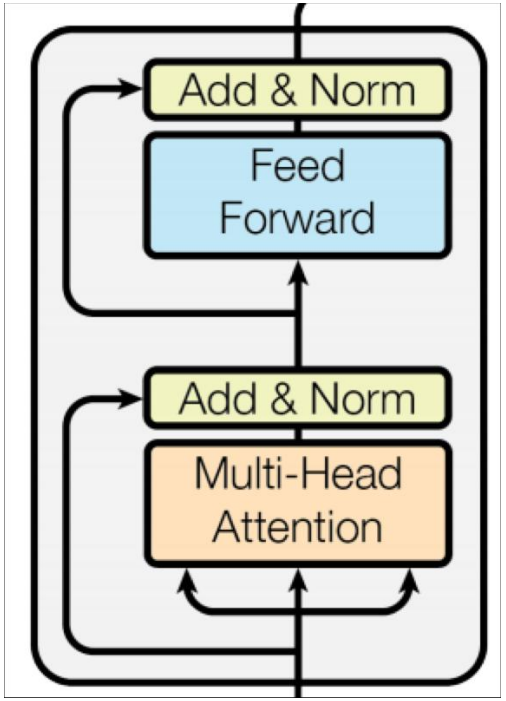
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$



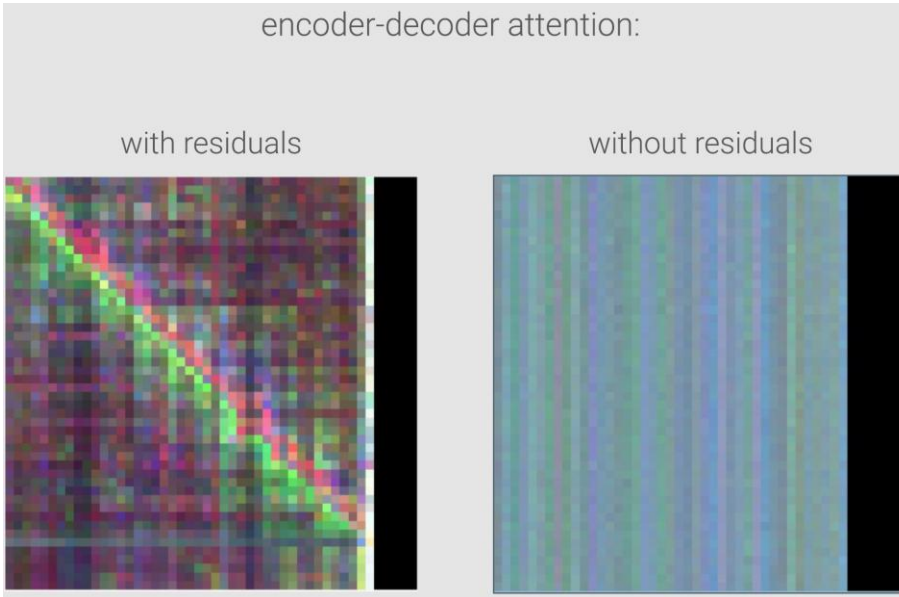
Encoder blocks

- Each block has two “sublayers”
  - Multihead attention
  - 2-layer feed-forward neural network (with ReLU)
- Each of these two steps also has a residual (short-circuit) connection and LayerNorm, i.e.:
  - LayerNorm(x + Sublayer(x))

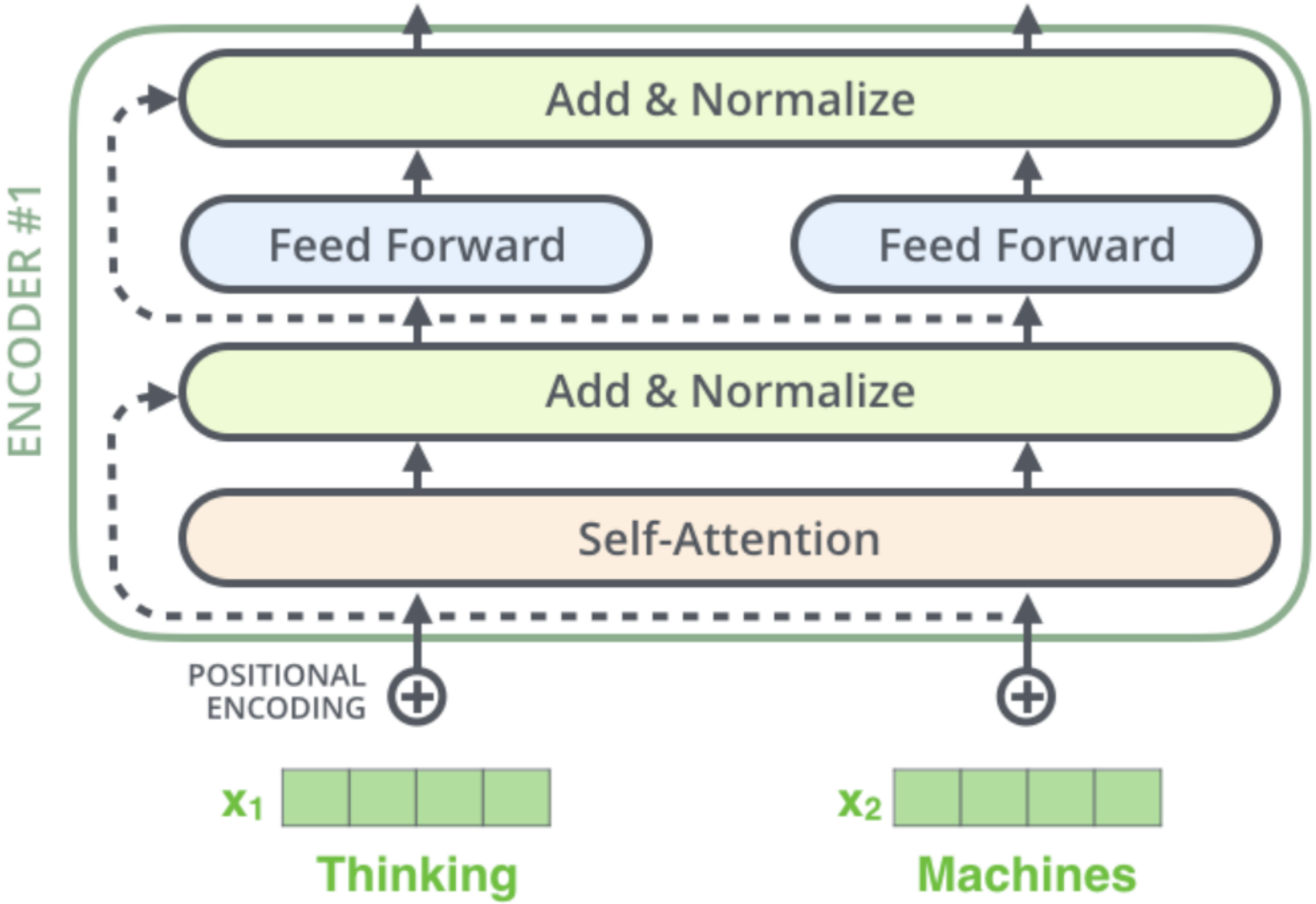


The Residual connections

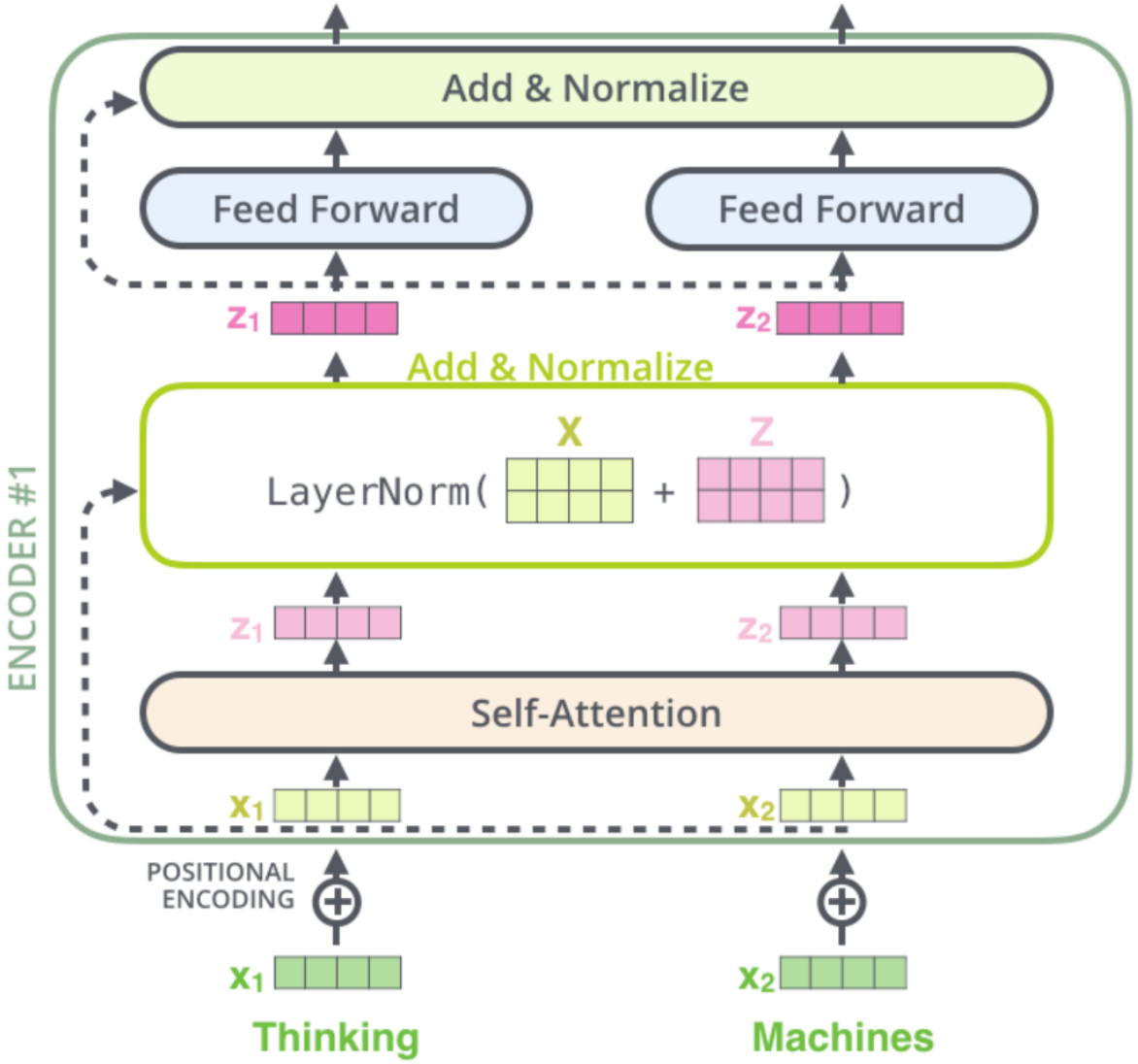
- each sub-layer of transformer (self-attention and feed-forward NN) in each encoder has a residual connection around it, and is followed by a layer-normalization step.
- the same for decoder sub-layers
- enables learning of deeper networks by improving a gradient flow
- in transformers, residual connections also help to maintain positional information in higher layers



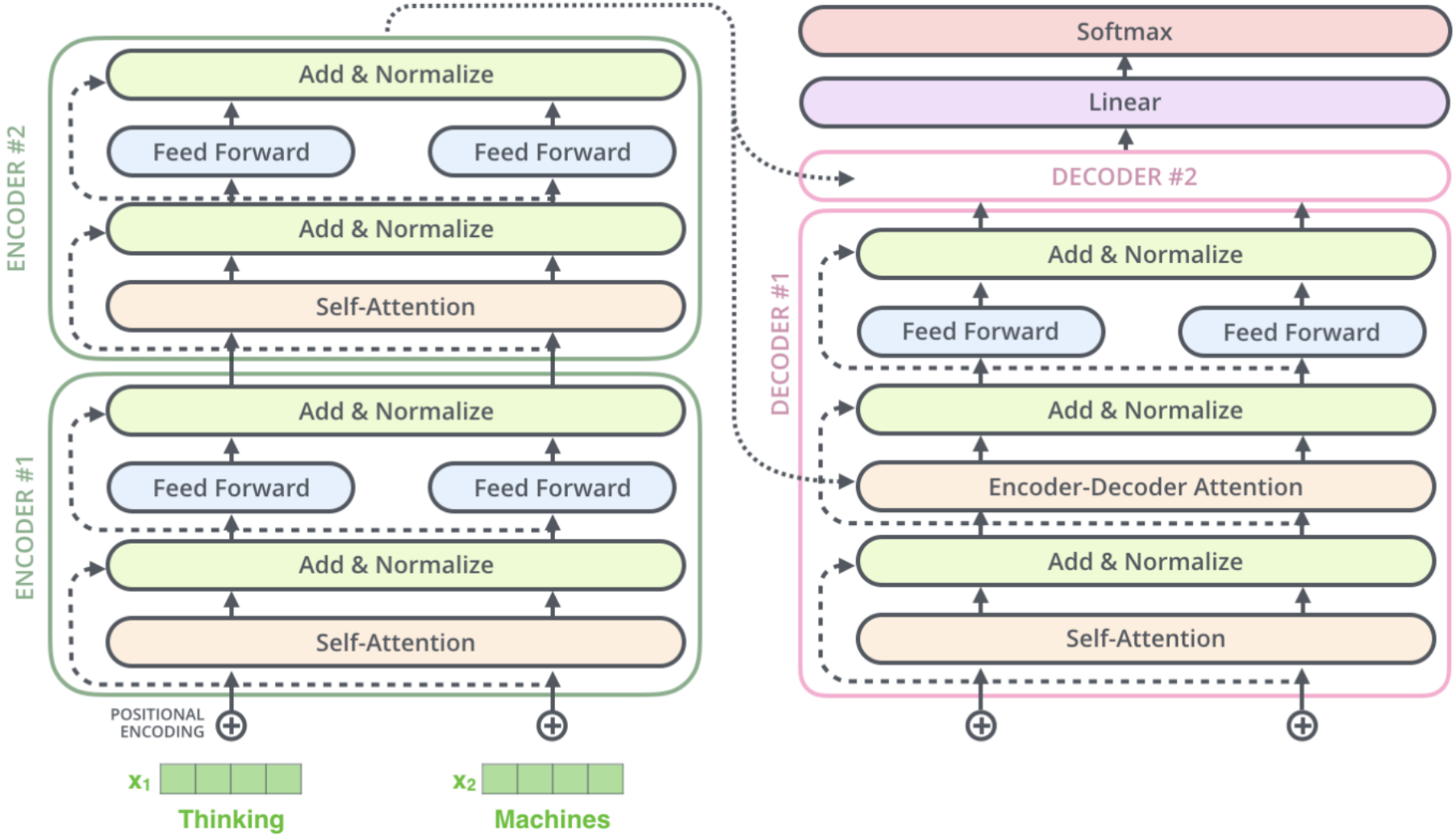
Architecture with residual connection – top level view



Architecture with residual connection – example

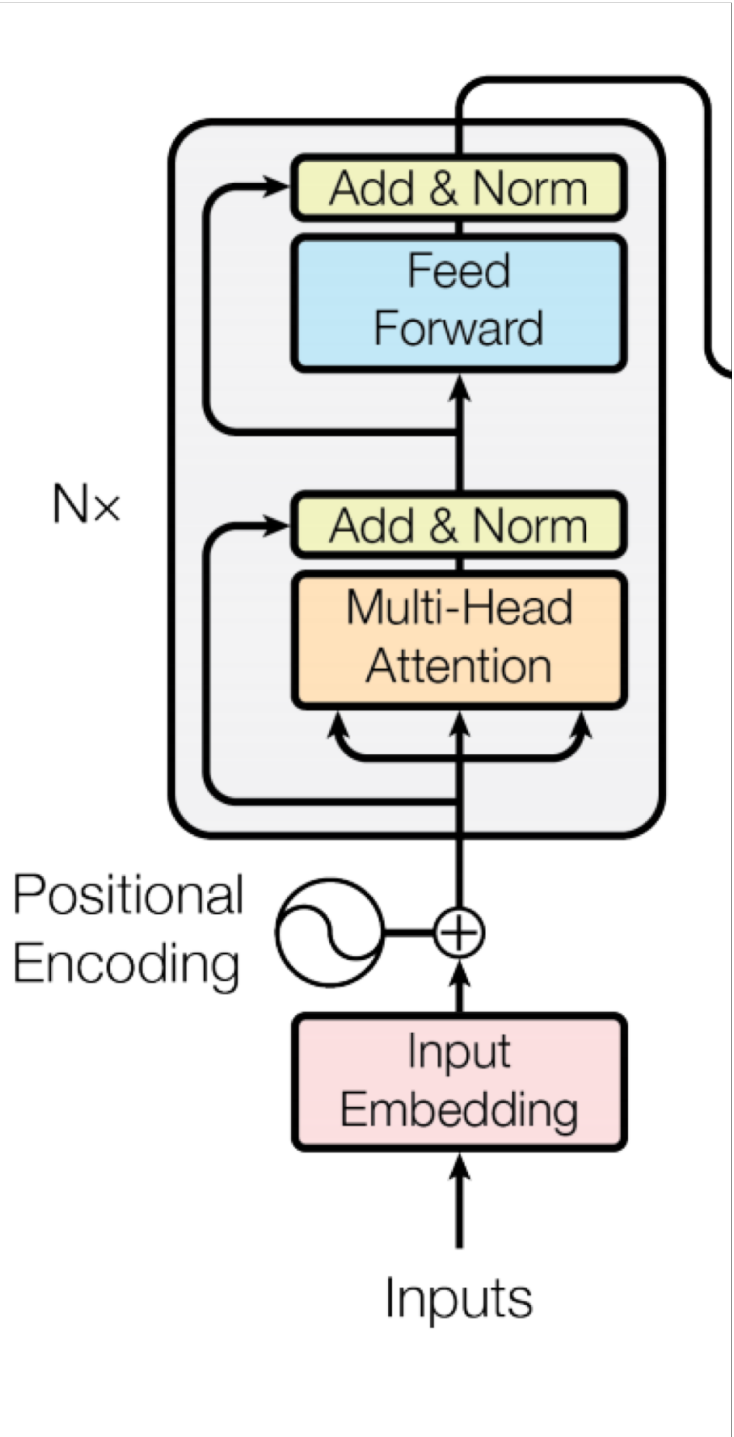


Example: 2 stacked transformer



Complete encoder

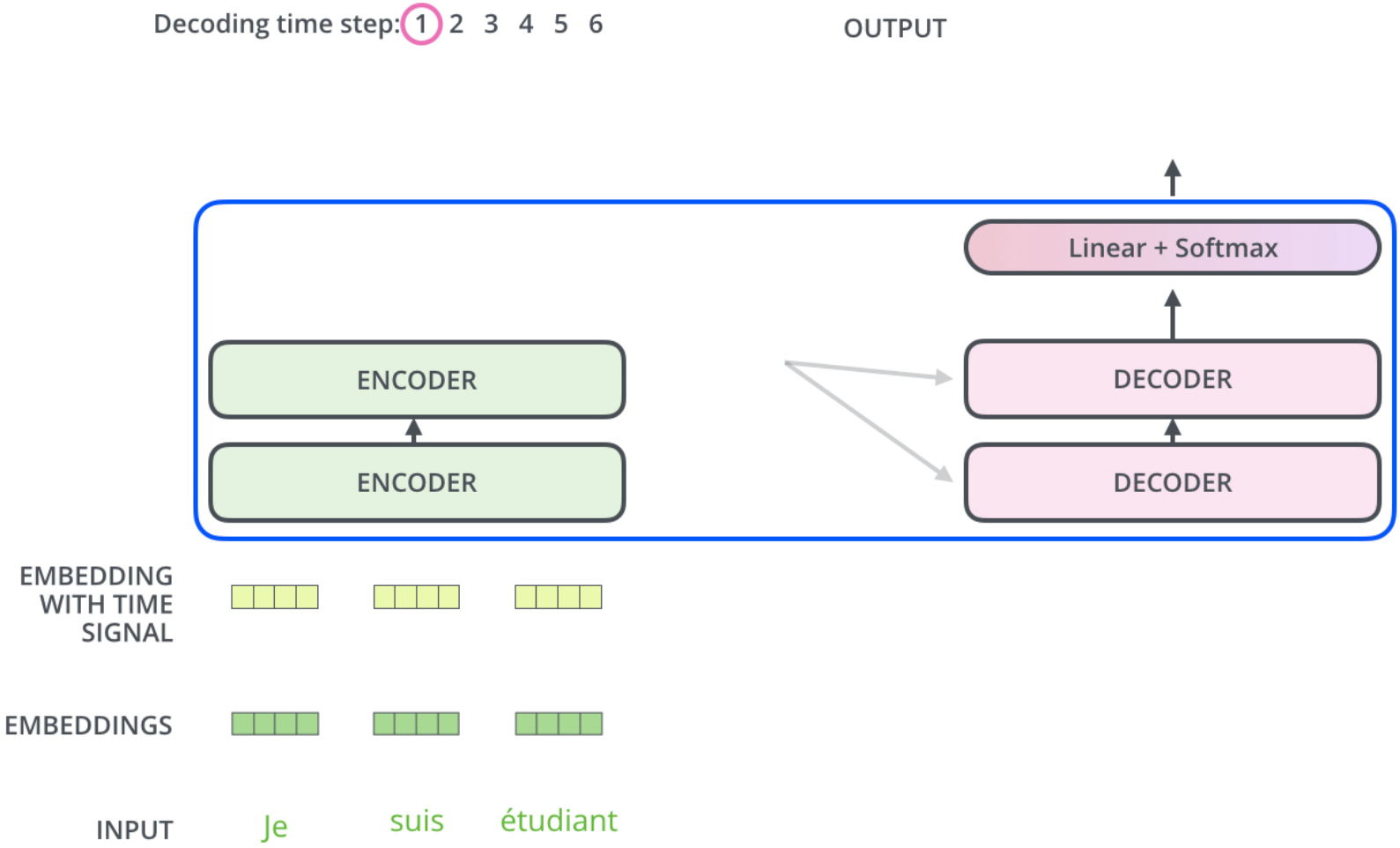
- each block is repeated several times, e.g., 6 times



Decoder

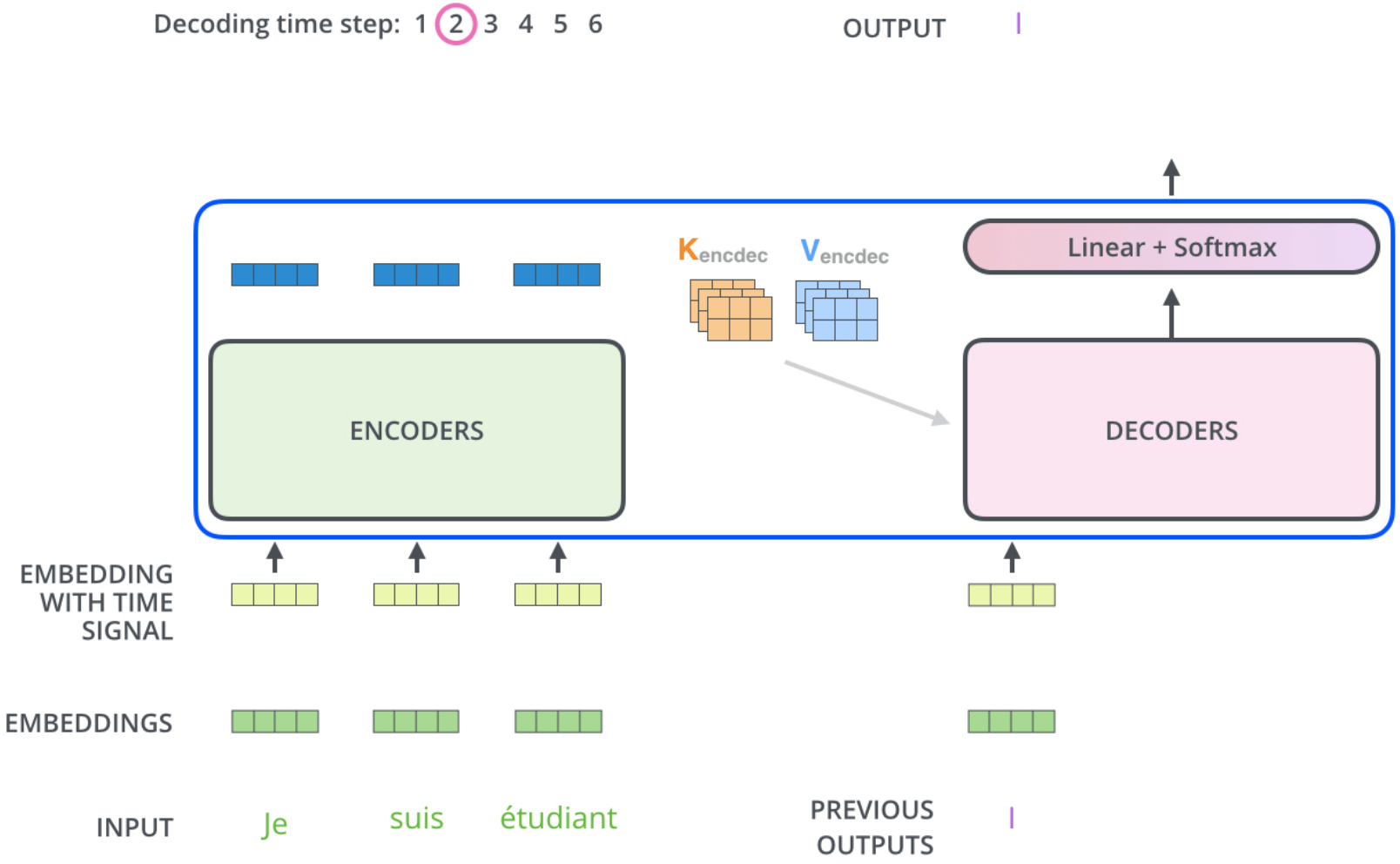
- Decoders have the same components as encoders
- An encoder starts by processing the input sequence.
- The output of the top encoder is transformed into a set of attention vectors K and V.
- These are used by each decoder in its “encoder-decoder attention” layer which helps the decoder to focus on appropriate places in the input sequence.

Encoder-decoder in action 1/2



After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

Encoder-decoder in action 2/2



The steps repeat until a special symbol indicating the end of output is generated. The output of each step is fed to the bottom decoder in the next time step. We add positional encoding to decoder inputs to indicate the position of each word.

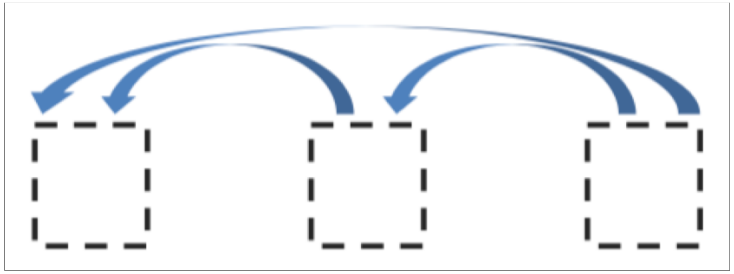
Self-attention and encoder-decoder attention in the decoder

- In the decoder, the self-attention layer is only allowed to attend to itself and earlier positions in the output sequence (to maintain the autoregressive property).
- This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.

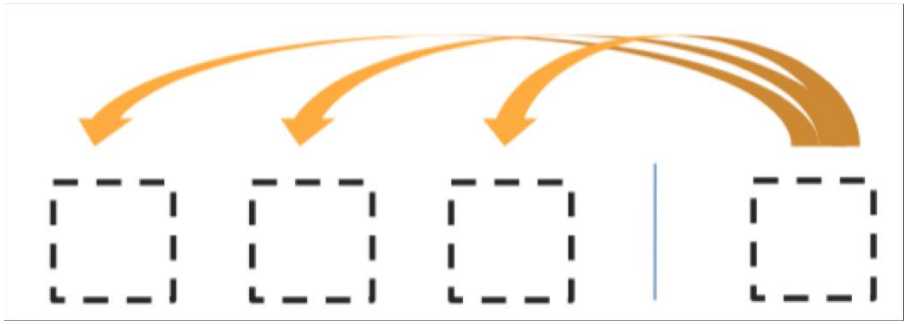
- The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Q (queries) matrix from the layer below it, and takes the K (keys) and V (values) matrix from the output of the encoder stack.

Attentions in the decoder

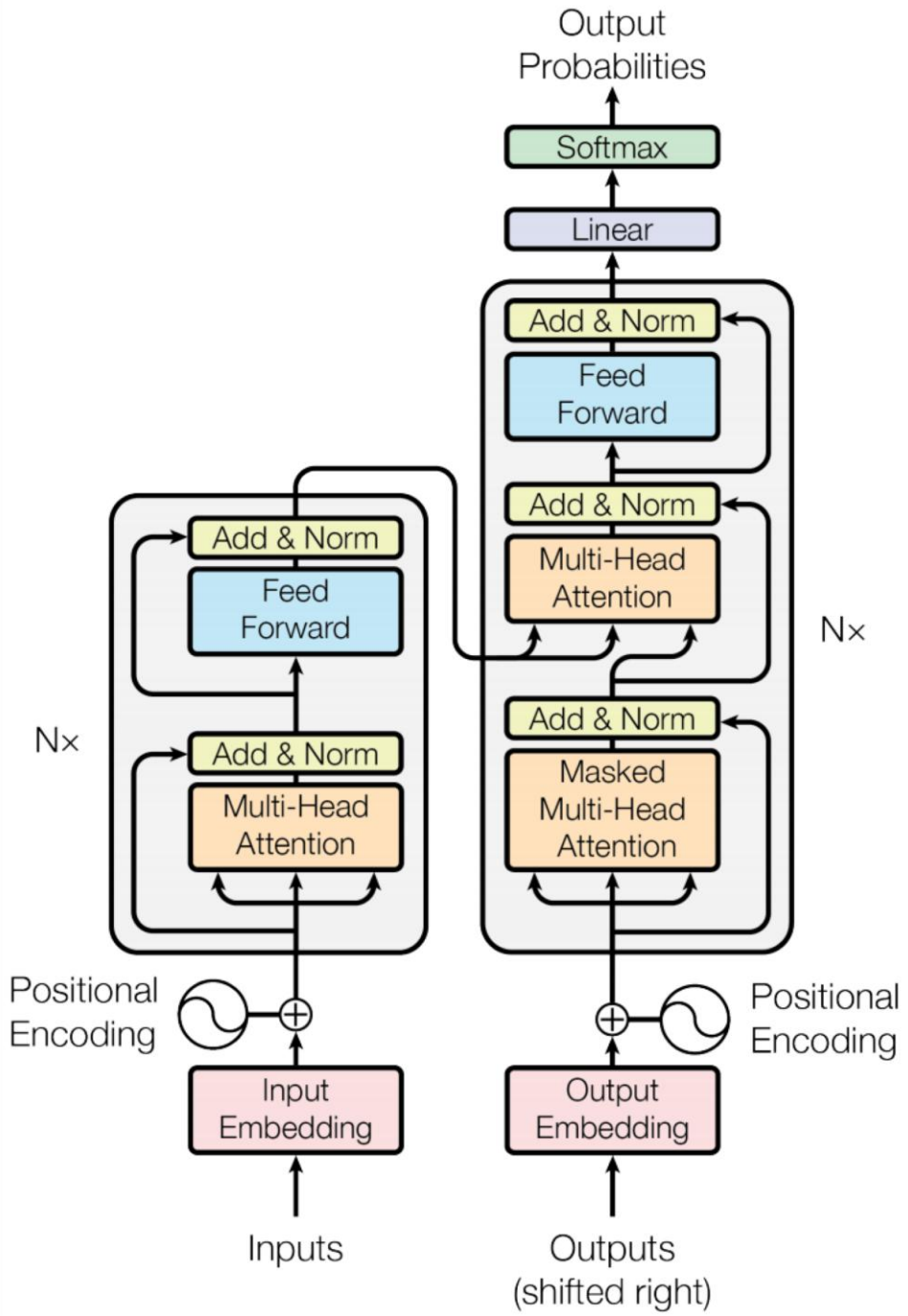
1. Masked decoder self-attention on previously generated outputs



2. Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of the encoder



One encoder-decoder block



Final Linear and Softmax Layer

- The decoder stack outputs a vector of floats.
- The final linear layer which is followed by a softmax layer turns them into words.
- The Linear layer is a simple fully connected neural network that projects the vector, produced by the stack of decoders, into a much larger vector called a logits vector (probability scores for each word).
- Example: the model knows 10,000 unique English words (“output vocabulary”) that it has learned from its training dataset. Therefore, the logits vector is 10,000 cells wide – each cell corresponding to the score of a unique word.
- The softmax layer turns those scores into probabilities (all positive, between 0 and 1, sum to 1.0).
  The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

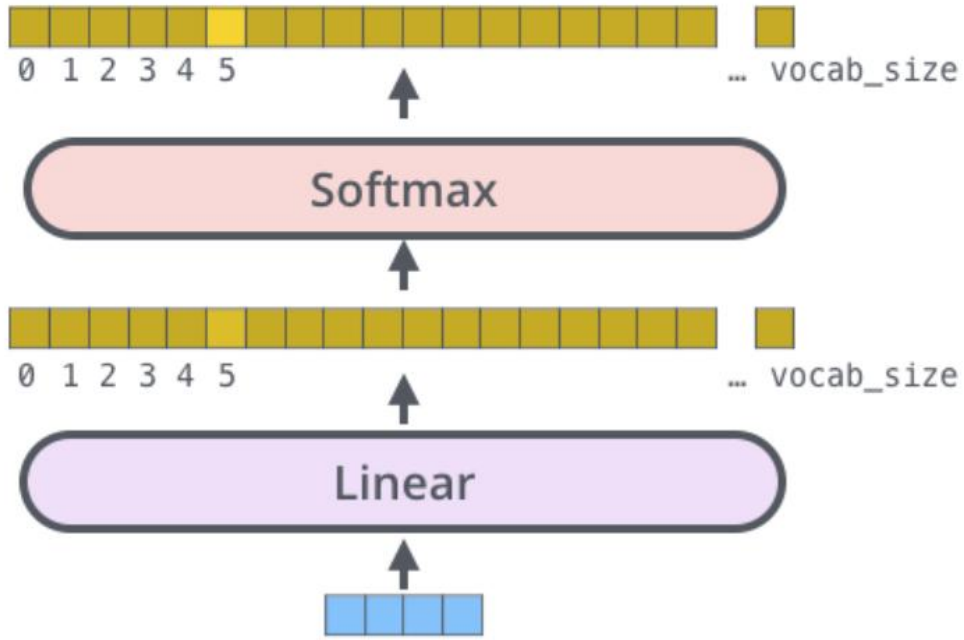
Producing the output words

Which word in our vocabulary is associated with this index?

am

Get the index of the cell with the highest value (**argmax**)

5



Training the transformer

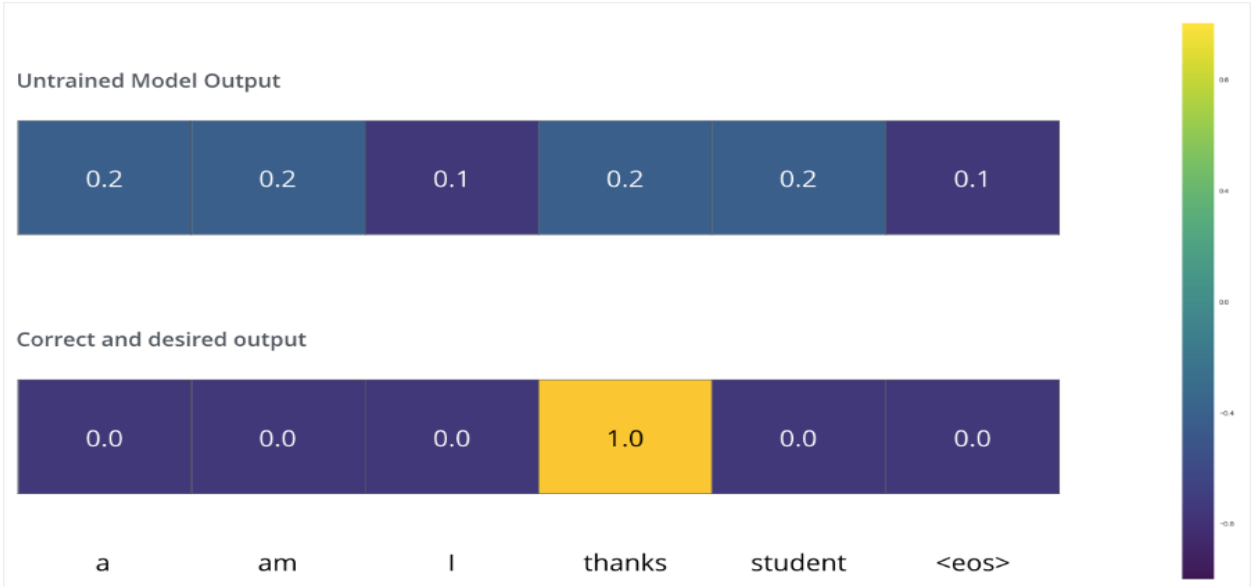
- During training, an untrained model would go through the exactly the same forward pass. But since we are training it on a labeled training dataset, we can compare its output with the actual correct output.
- For illustration, let's assume that our output vocabulary only contains six words(a, am, i, thanks, student, <eos>)
- The input is typically in the order of 10^4 (e.g., 30 000)

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

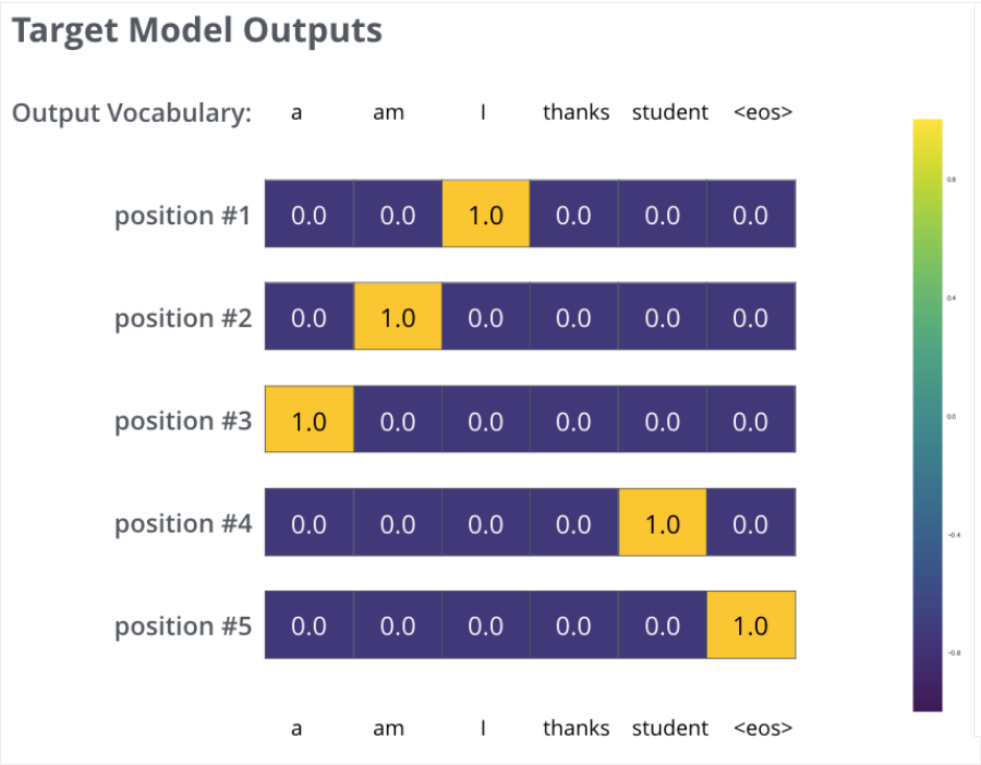
The Loss Function

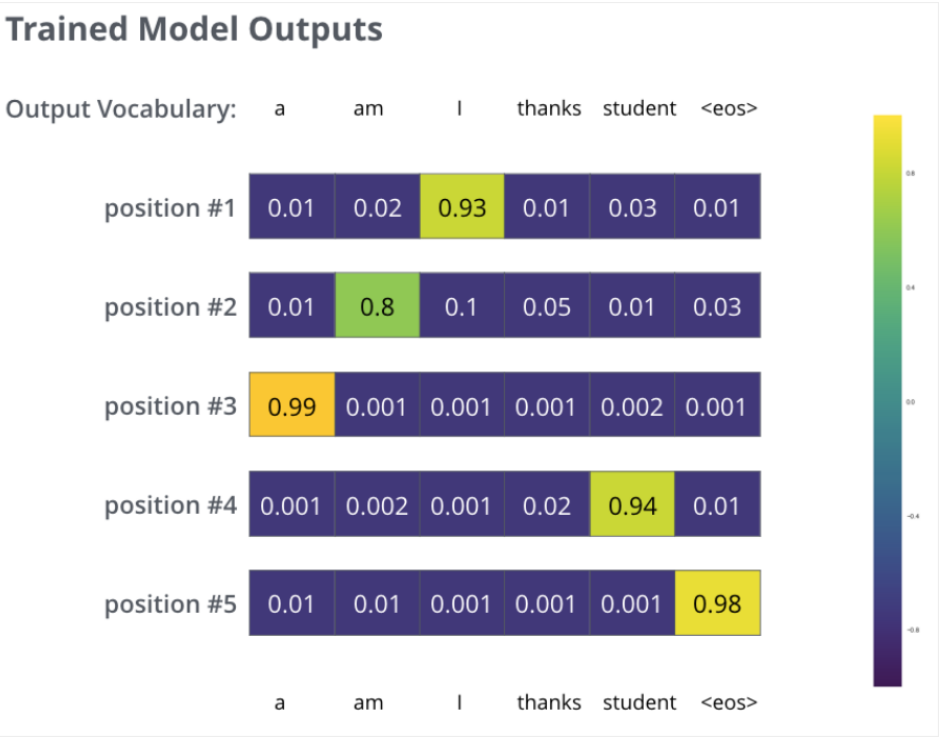
- evaluates the difference between the true output and the returned output
- transformer typically uses cross-entropy or Kullback–Leibler divergence.
- The model output is a probability distribution, the true output is 1-hot encoded, e.g.,



Loss evaluation for sequences

- loss function has to be evaluated for the whole sentence, not just a single word
- transformers use greedy decoding or beam search





Subword Encoding tokenization

- Learn tokenization based on statistics
- Relevant for modern neural networks
- Use the data to guide the tokenization
- **Subword tokenization** (because tokens are often parts of words)
- Can include common morphemes like *-est* or *-er*.
  - (A morpheme is the smallest meaning-bearing unit of a language; *unlikeliest* has morphemes *un-*, *likely*, and *-est*.)
- Relevant for all languages, but crucial for morphologically rich languages such as Slovene
- What happens if subword tokenization is inadequate?

Subword tokenization

- Common algorithms:
  - Byte-Pair Encoding (BPE) (Sennrich et al., 2016)
  - WordPiece (Schuster and Nakajima, 2012)
- Both have 2 parts:
  - A token learner that takes a raw training corpus and induces a vocabulary (a set of tokens).
  - A token segmenter that takes a raw test sentence and tokenizes it according to that vocabulary

Byte Pair Encoding (BPE)

Let vocabulary be the set of all individual characters

= {A, B, C, D,…,a, b, c, d….}

- Repeat:
  - choose the two symbols that are most frequently adjacent in training corpus (say ‘A’, ‘B’),
  - adds a new merged symbol ‘AB’ to the vocabulary
  - replace every adjacent ’A’ ’B’ in corpus with ‘AB’.
- Until *k* merges have been done.

BPE token learner algorithm

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

BPE in use

- Most subword algorithms are run inside white-space separated tokens.
- So first add a special end-of-word symbol '__' before whitespace in training corpus
- Next, separate into letters.

BPE token learner

An example corpus :(

low low low low low lowest lowest newer newer newer newer newer newer wider wider wider new new

Add end-of-word tokens and segment:

corpus		vocabulary
5	l o w _	_, d, e, i, l, n, o, r, s, t, w
2	l o w e s t _	
6	n e w e r _	
3	w i d e r _	
2	n e w _	

BPE token learner

corpus

5 l o w _
2 l o w e s t _
6 n e w e r _
3 w i d e r _
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w

Merge e r to er

corpus

5 l o w _
2 l o w e s t _
6 n e w er _
3 w i d er _
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w, er

BPE

corpus

5 l o w _
2 l o w e s t _
6 n e w er _
3 w i d er _
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w, er

Merge er _ to er_

corpus

5 l o w _
2 l o w e s t _
6 n e w er_
3 w i d er_
2 n e w _

vocabulary

_, d, e, i, l, n, o, r, s, t, w, er, er_

BPE

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2 l o w e s t _	
6 n e w er_	
3 w i d er_	
2 n e w _	

Merge n e to ne

corpus	vocabulary
5 l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2 l o w e s t _	
6 ne w er_	
3 w i d er_	
2 ne w _	

BPE

The next merges are:

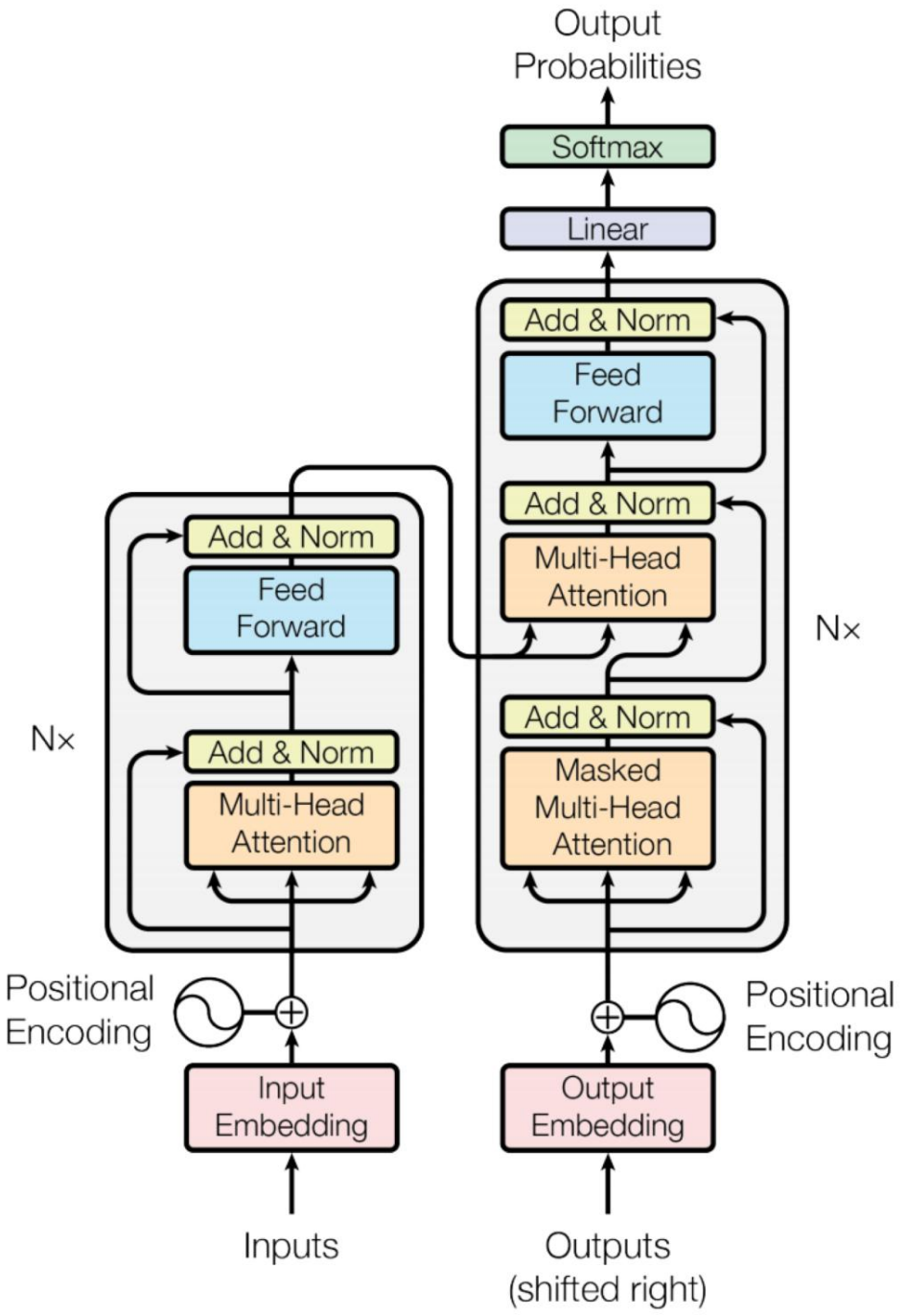
Merge	Current Vocabulary
(ne, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

BPE token learner algorithm

- On the test data, run each merge learned from the training data:
  - Greedily
  - In the order we learned them
  - (test frequencies don't play a role)
- So: merge every e r to er, then merge er _ to er_, etc.
- Result:
  - Test set "n e w e r _" would be tokenized as a full word
  - Test set "l o w e r _" would be two tokens: "low er_"

Transformer architecture

- typically, the input is first tokenized with subword encoding
- what is the alternative?



Transformer hints

- Byte-pair/sentence pair encodings for input tokens
- Checkpoint averaging
- ADAM optimizer with learning rate changes
- Dropout during training at every layer just before adding residual
- Label smoothing
- Auto-regressive decoding with beam search and length penalties
- Use of transformers is widespread in the form of pretrained models
- Without pretraining, they are hard to optimize and unlike LSTMs don’t usually just work out of the box and might not work well with other building blocks on tasks.

Transformers are everywhere

- music: LLM where the vocabulary consists of MIDI pitches, pauses, velocity
- object detection (attention to objects)
- time series, where signals are discretized and values are treated as letters