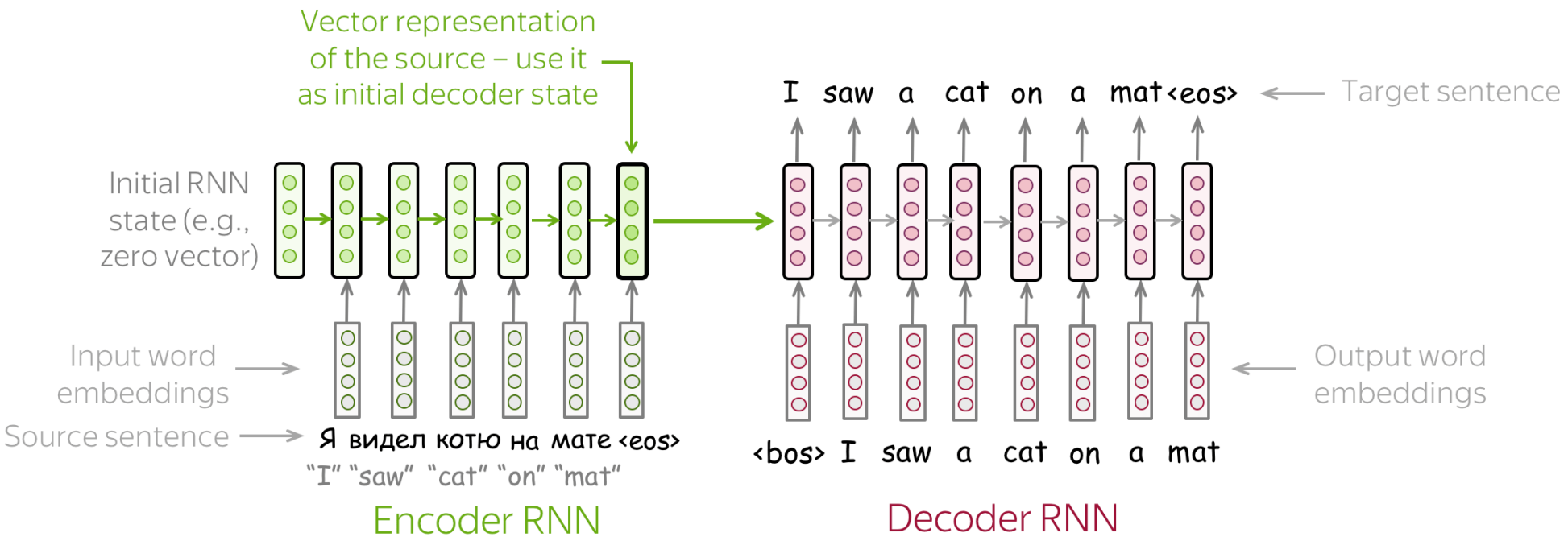
Introduction to transformers

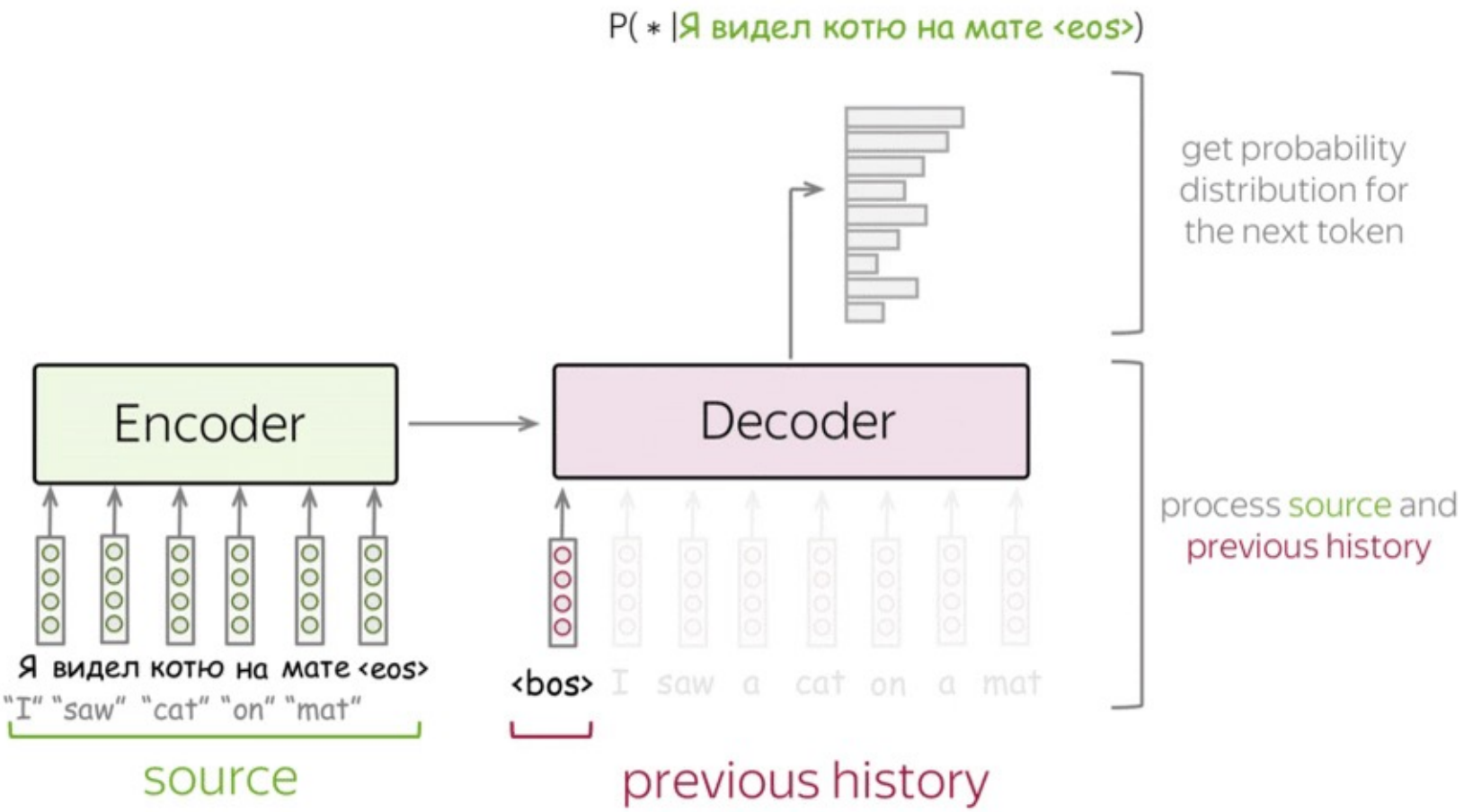
Why Transformers were introduced

- RNNs:
  - Introduce ‘memory’ inside a neuron
  - The output of one node affects the input of next node: time series forcasting
  - Main drawback: RNN is slow - **cannot be parallelized**
- Transformers is a generic architecture with an encoder and decoder part
  - The main difficulties is how to train it.
    - Dataset
    - Task
    - Strategies
  - Originally used (2017) for textual data processing, since 2020 used in vision and today used in all domain
- Popularised by
  - BERT (Bidirectional Encoder Representation from Transformers)
    - https://arxiv.org/pdf/1810.04805.pdf (2019)
    - At inference time BERT use maily the encoder part of the transformer architecture
  - GPT (Generative Pre-trained Transformer)
    - **Language Models are Few-Shot Learners –https://arxiv.org/abs/2005.14165 (2020)**
    - At inference time BERT use maily the decoder part of the transformer architecture

Reminder of the seq2seq model, training step

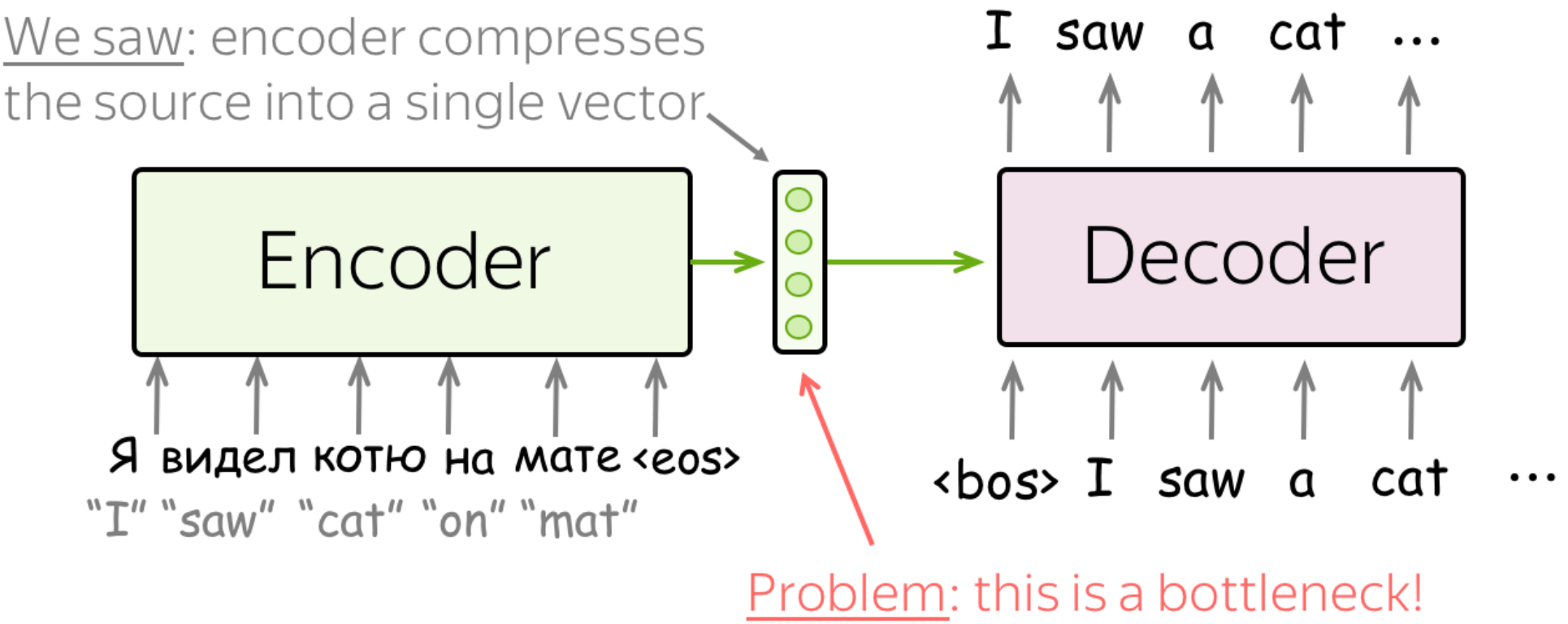


Seq2Seq model, inference step



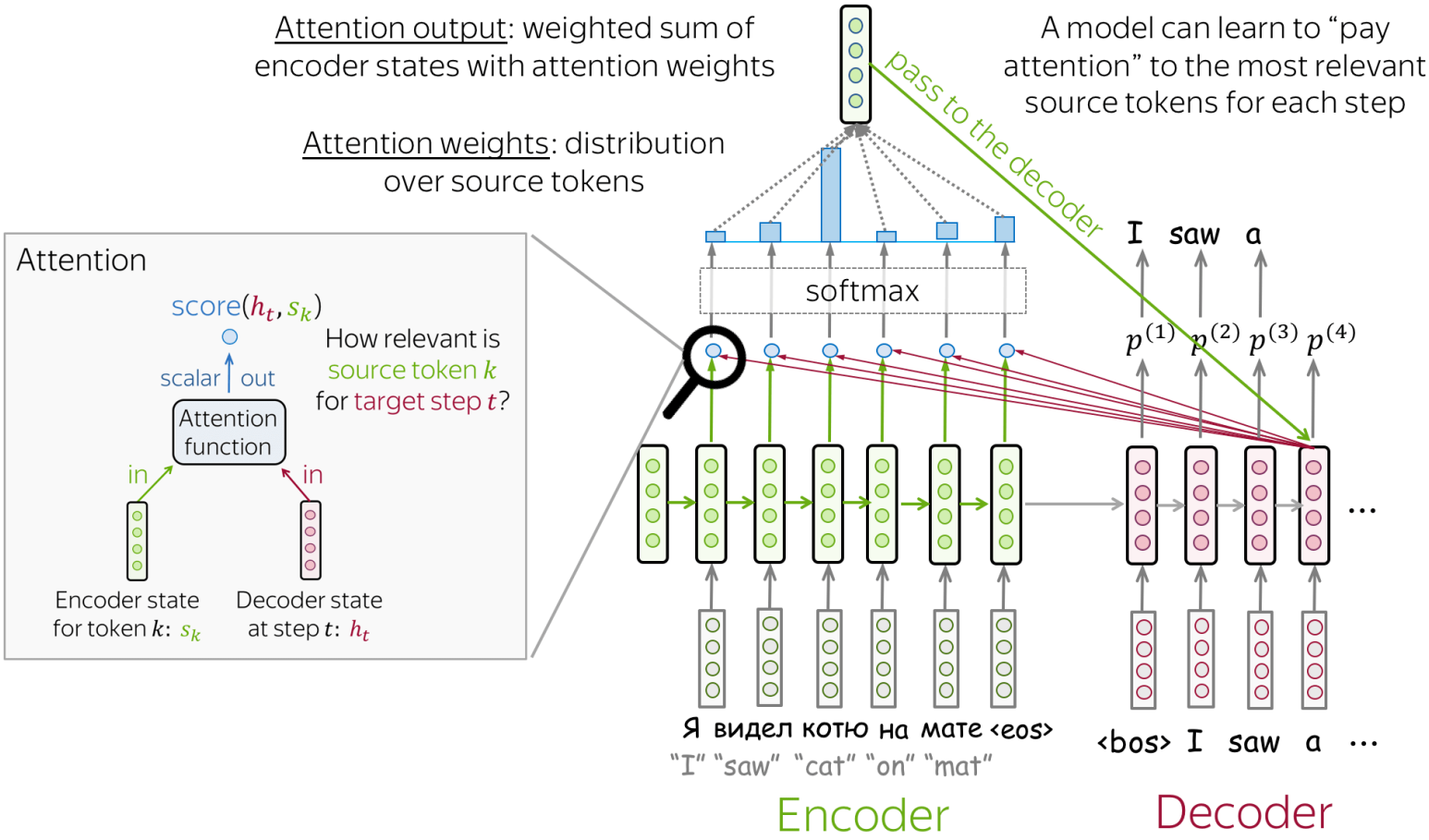
https://wikidocs.net/178419

Reminder of the seq2seq model



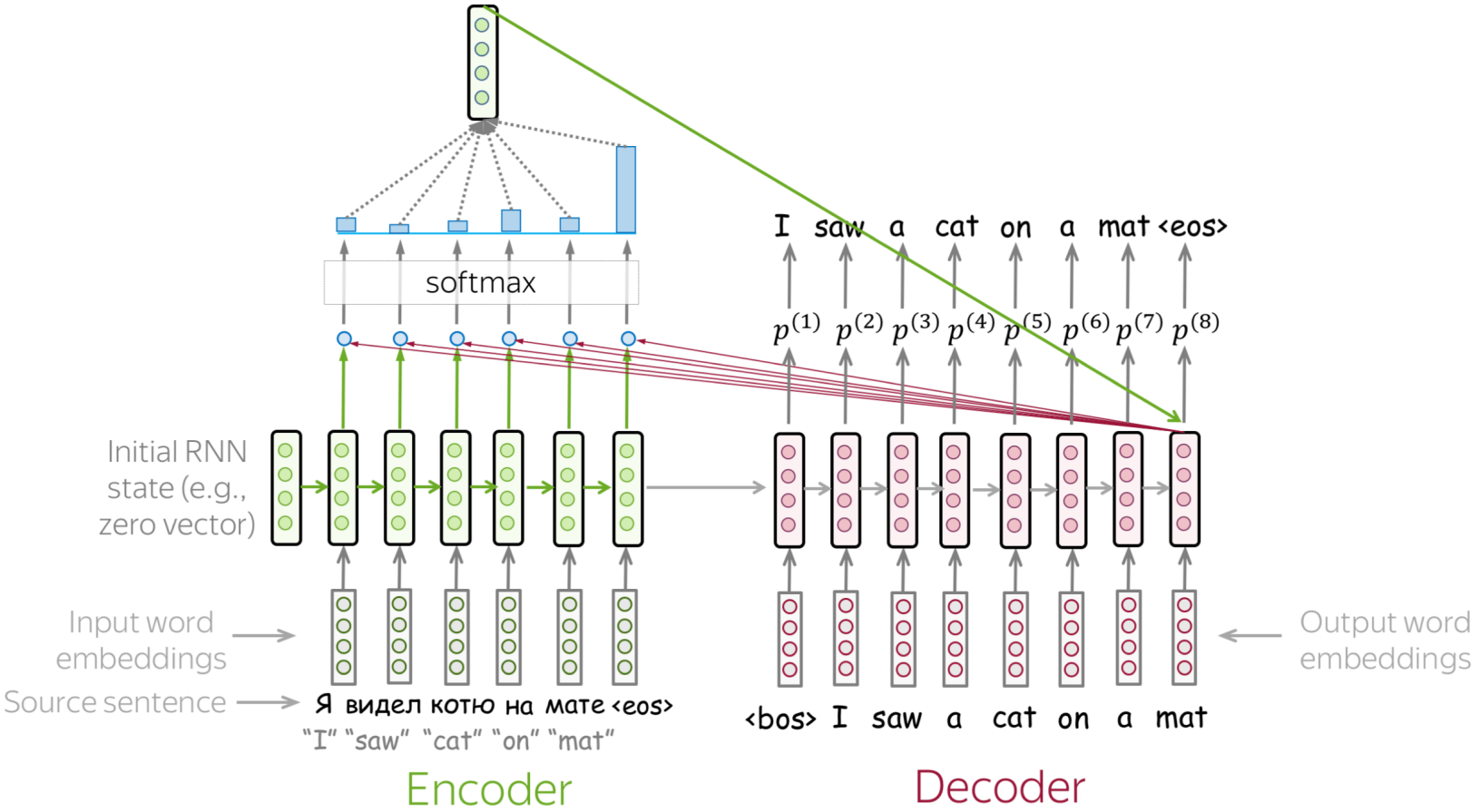
https://wikidocs.net/178419

Attentional Seq2Seq model



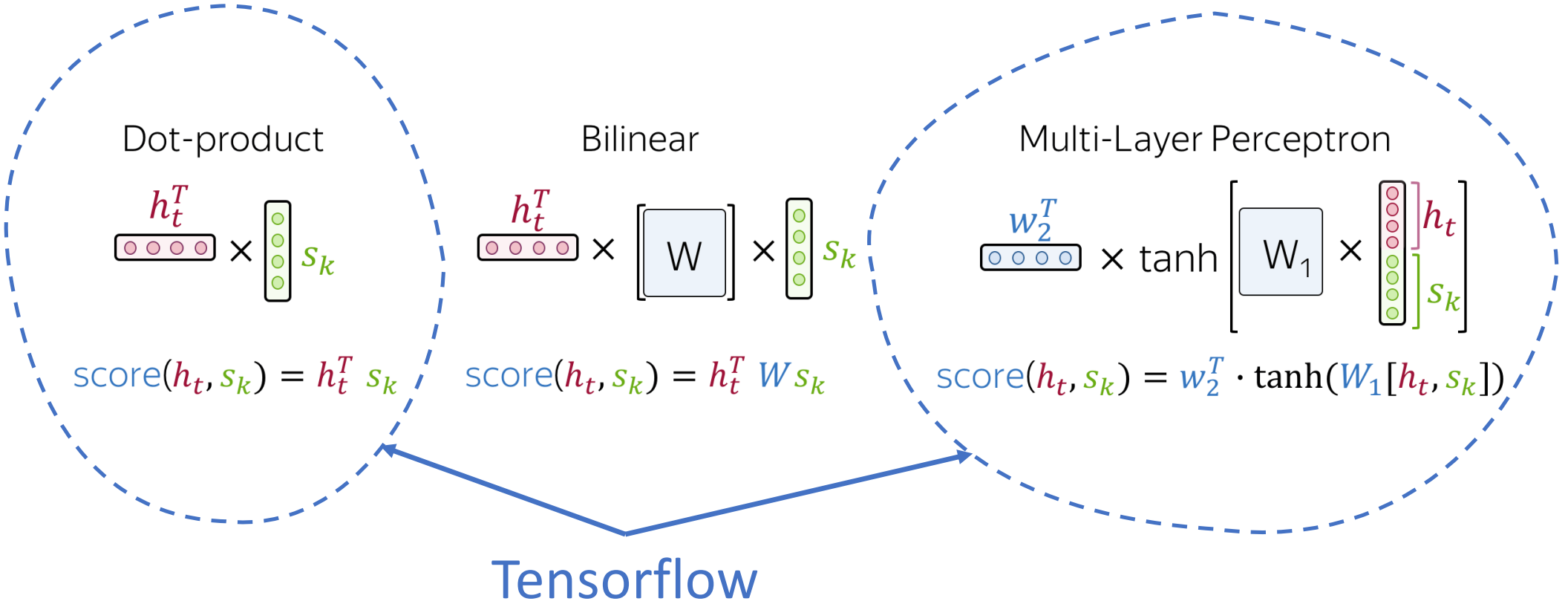
https://wikidocs.net/178419

Attentional Seq2Seq model

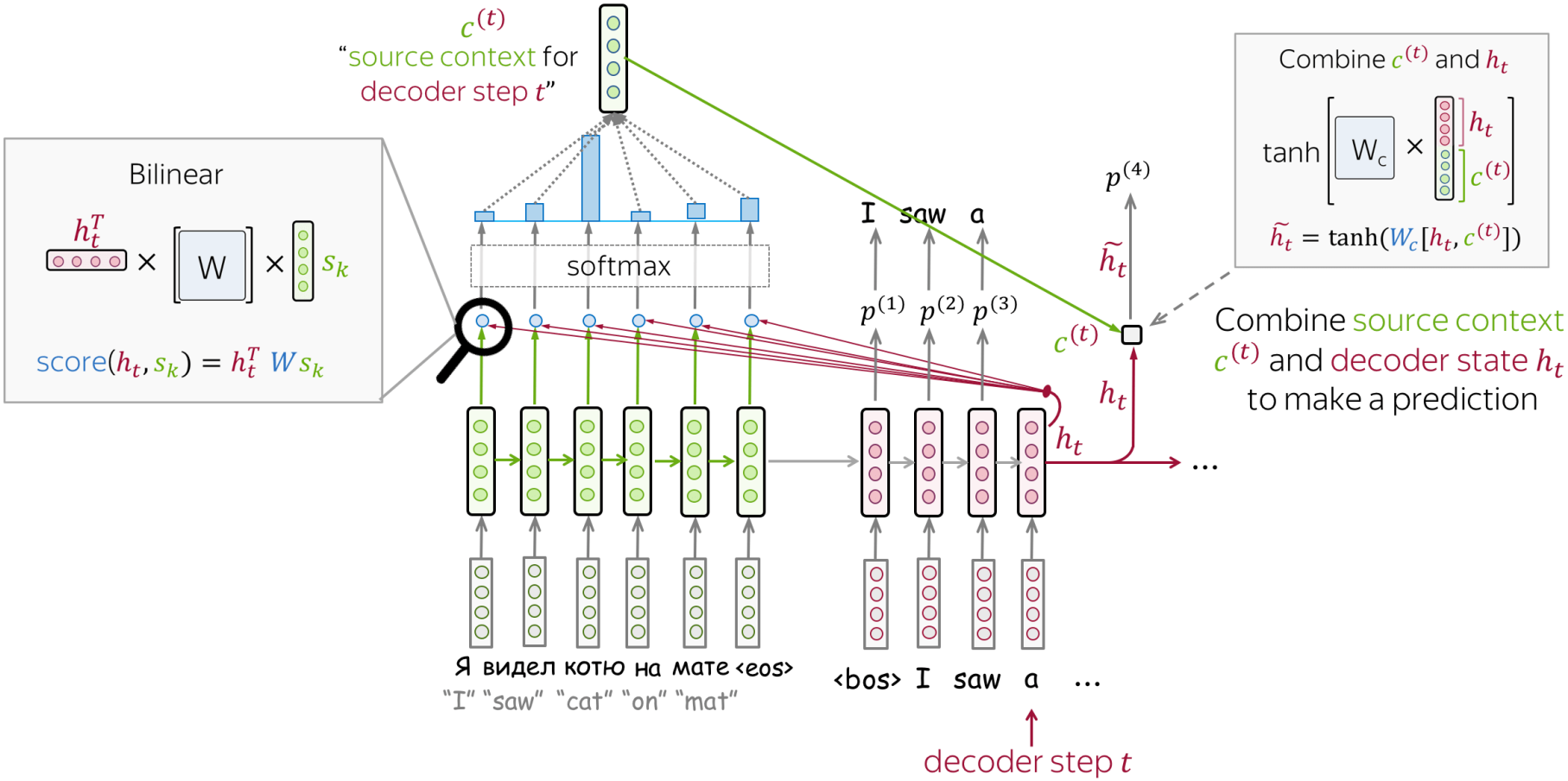


https://wikidocs.net/178419

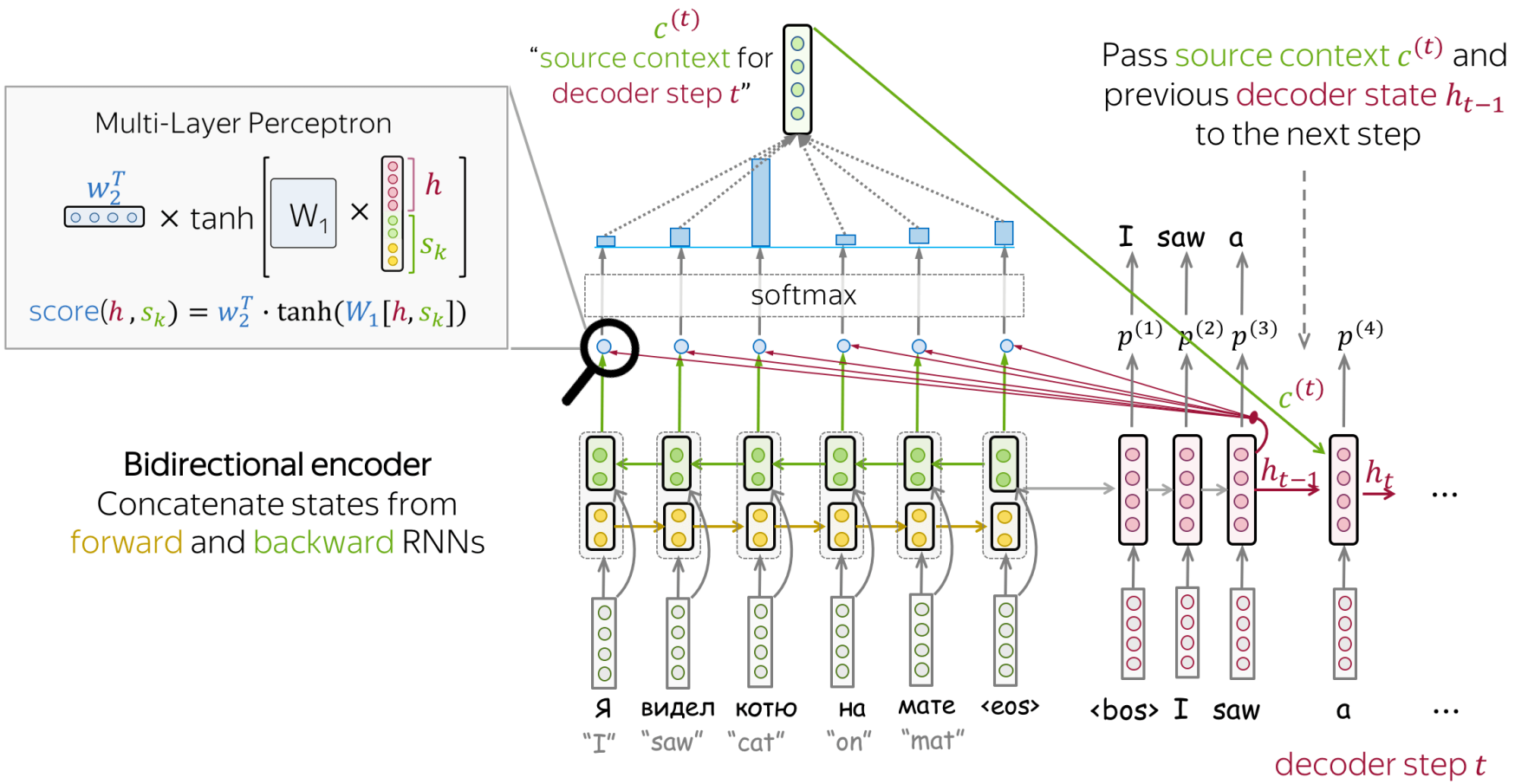
Score function



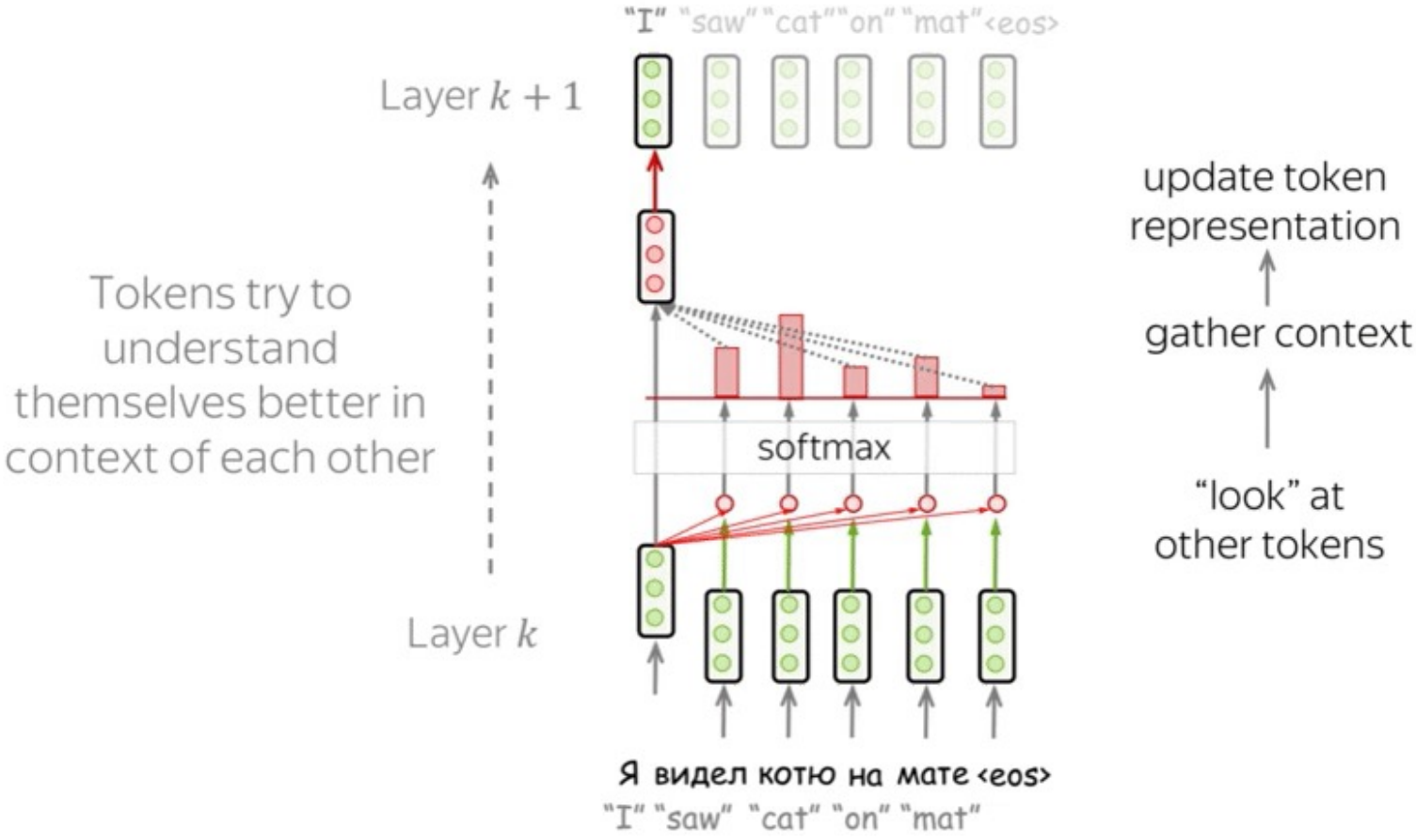
Luong architecture



Bahdanau architecture



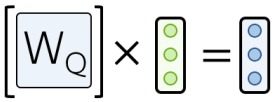
Self-attention



https://wikidocs.net/178419

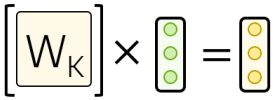
Query, Key, Value – Attentional layer – **available on Tensorflow**

Each vector receives three representations (“roles”)



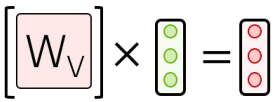
Query: vector **from** which the attention is looking

“Hey there, do you have this information?”



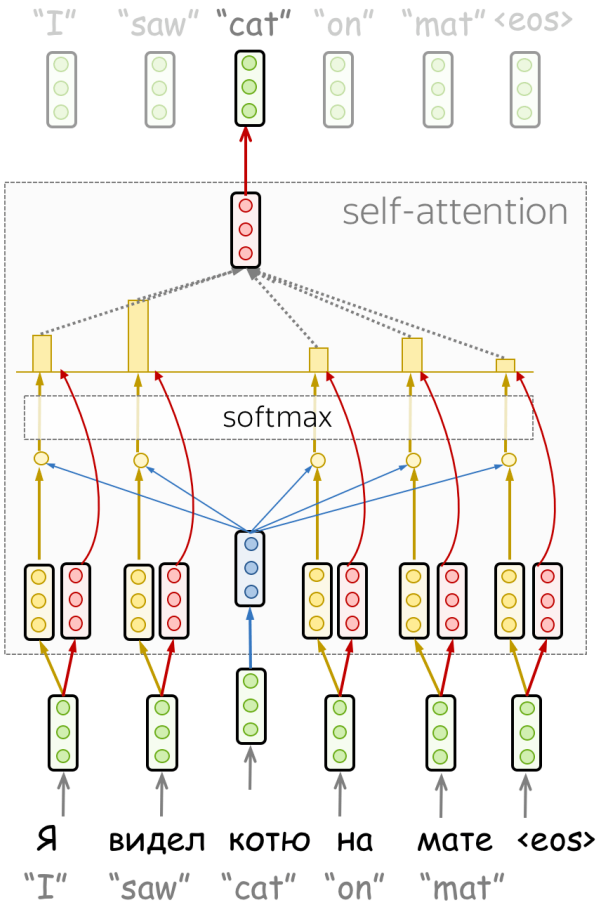
Key: vector **at** which the query looks to compute weights

“Hi, I have this information – give me a large weight!”

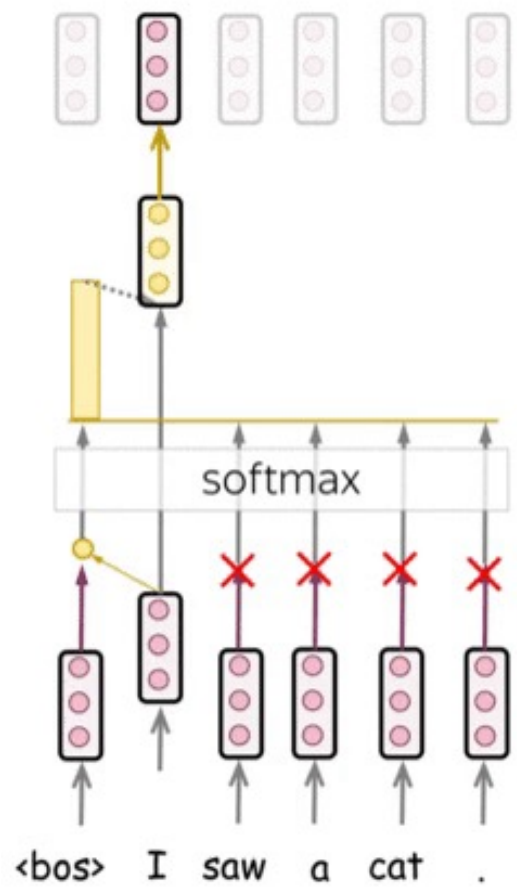


Value: their weighted sum is attention output

“Here’s the information I have!”



Masked head attention - **"Don't Look Ahead" for the Decoder**



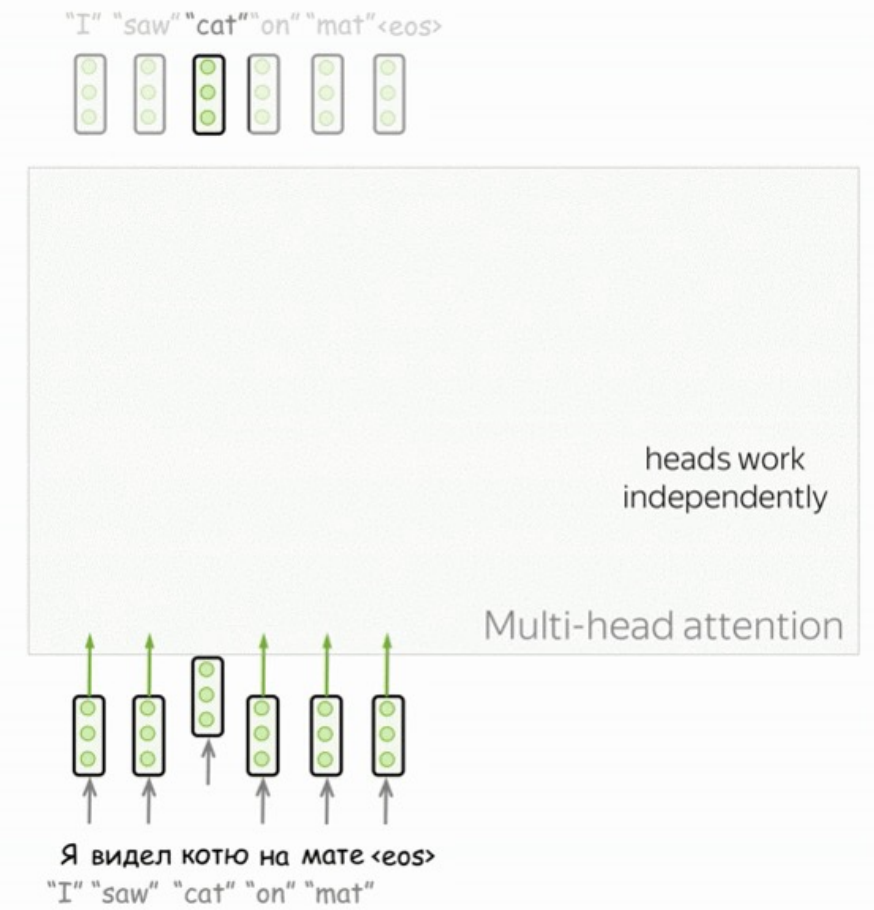
update token representation

gather context

"look" at the previous tokens (future tokens are masked out)

https://wikidocs.net/178419

Multi-Head Attention – available on Tensorflow



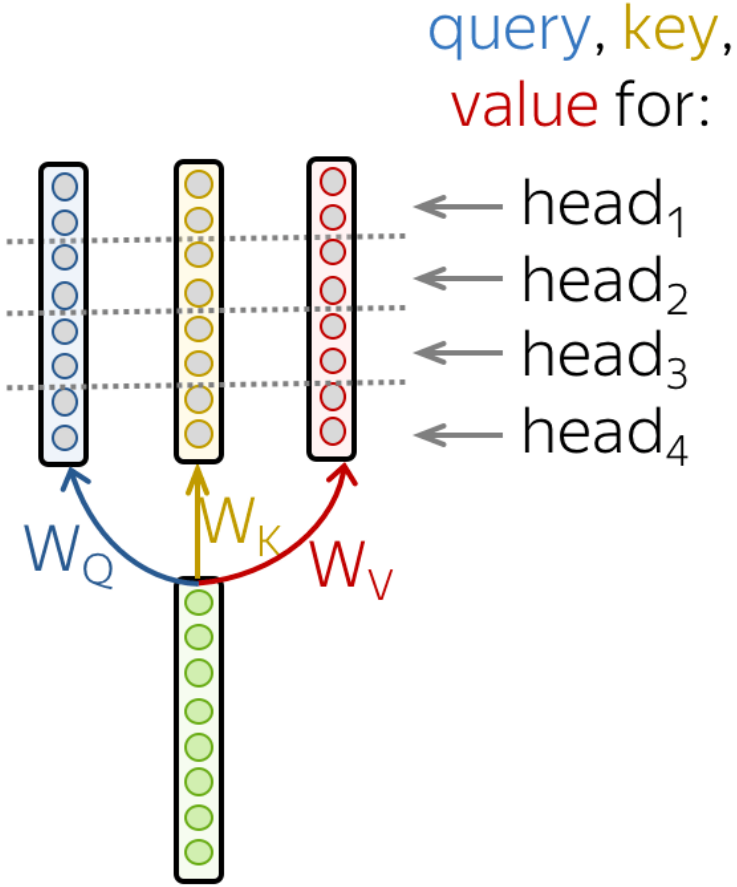
https://wikidocs.net/178419

Multi-head attention

Multiple heads -> Each head is independent —> Each head focuses on a different thing.

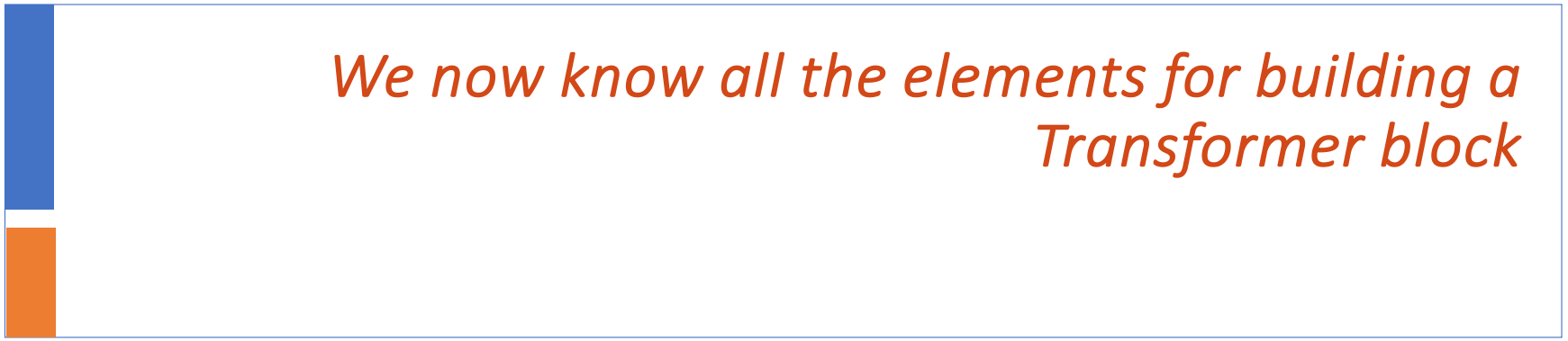
Queries, keys and values can be split into several values -> an attention is calculated on each value -> the result is concatenated.

Split equally into number of heads parts



$$head_i = Attention(QW_Q^i, KW_W^i, VW_W^i)$$
$$MultiHead(Q,V,T) = Concat(head_i, \dots, head_i)$$

In this way, models with a single attention head or several have the same size.

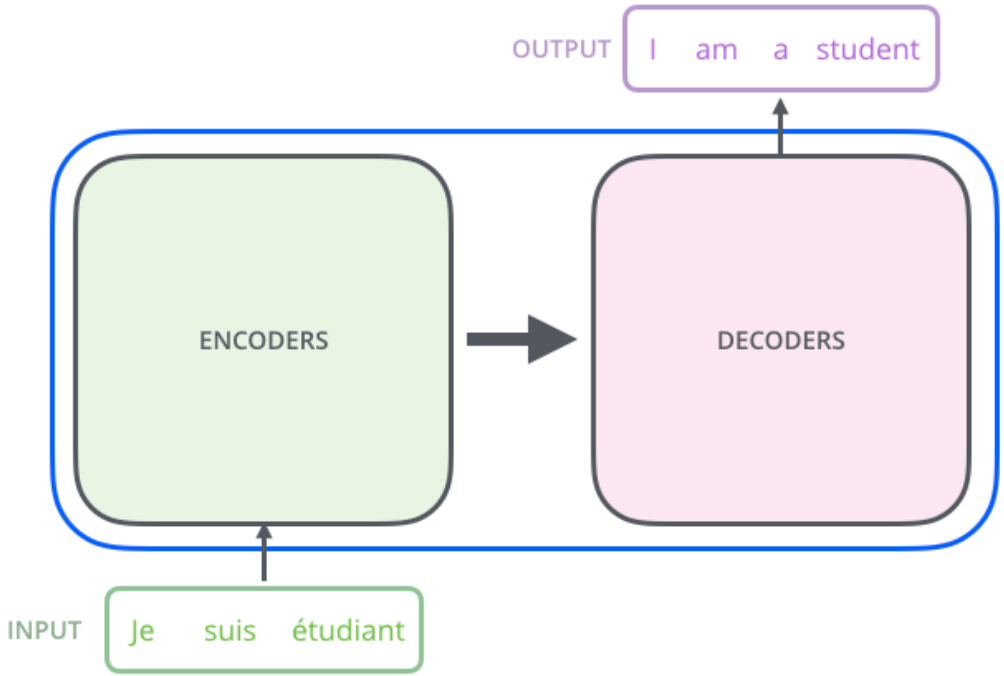

We now know all the elements for building a Transformer block

Transformers

- **The starting point is the paper**
  - Attention is all you need published in 2017
- **Today: just an introduction:** These are two valuable resources to learn more details on the architecture and implementation
  - http://nlp.seas.harvard.edu/2018/04/03/attention.html
  - https://jalammar.github.io/illustrated-transformer/ (some pictures come from this source)
- For SI5 students:
  - The second semester course "From Shallow to Deep Learning" will return to this.
- For M1 DSAI students:
  - You have a full lecture about transformer next year

The full architecture : a set of encoder-decoder pairs

The Transformer consists of two individual modules, namely the **Encoder** stack and the **Decoder** stack.

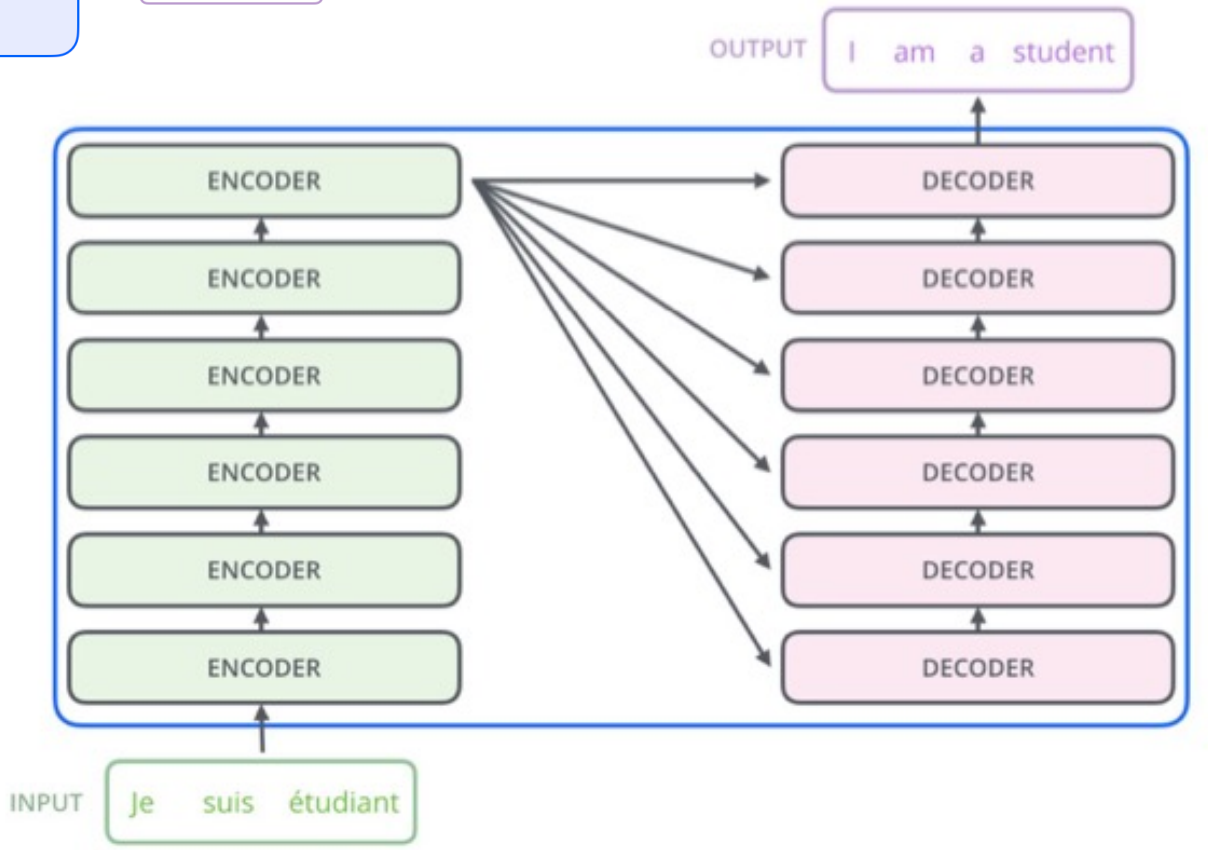


- https://jalammar.github.io/illustrated-transformer/

The full architecture : a set of encoder-decoder pairs

The encoding component is a stack of encoders

The decoding component is a stack of decoders of the same number.



https://jalammar.github.io/illustrated-transformer/

The basic elements : encoder and decoder

Before entering the **encoder** or **decoder** stack, the source tokens are first **embedded** into a high-dimensional space.

Then, a **position encoding** is added to each input embedding

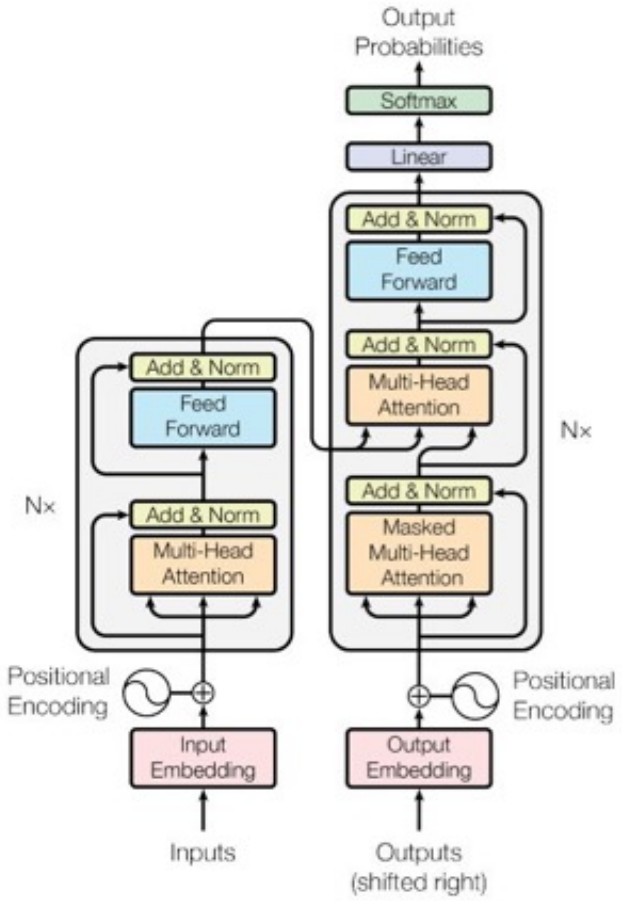


Figure credit: Vaswani et. al

Encoder

*The first layer is a **multi-head self-attention mechanism**,*

*The second layer is a simple, **position-wise, fully connected feed-forward network.***

Each layer employs **residual connection.**

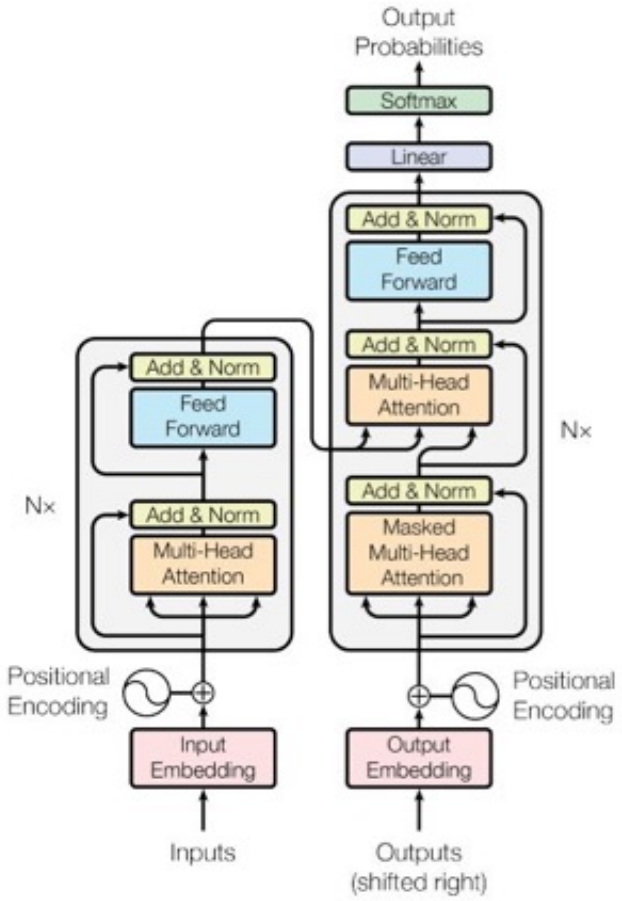
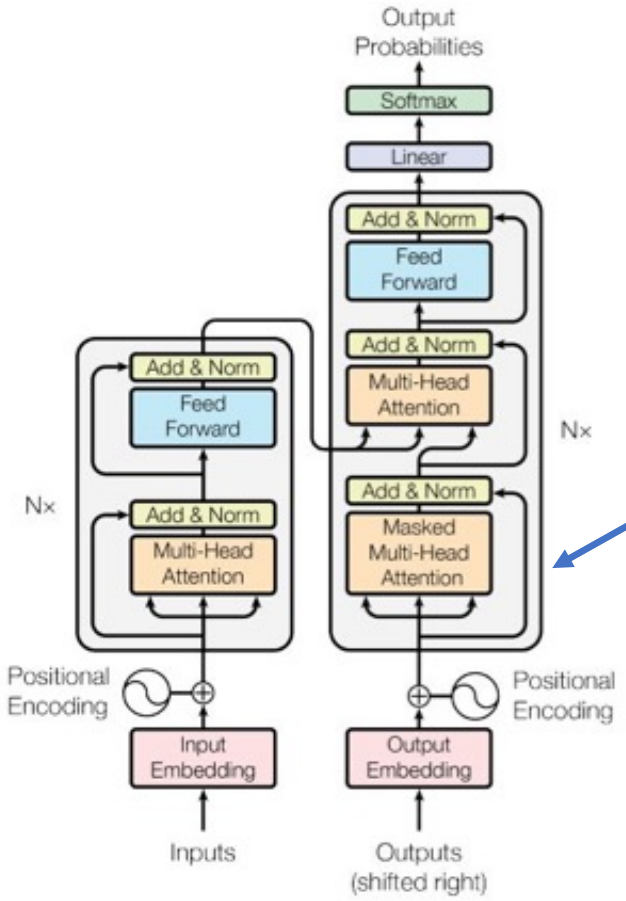


Figure credit: Vaswani et. al

Decoder

Encoder-like architecture which includes a ***multi-head attention*** *over the output of the encoder stack.*



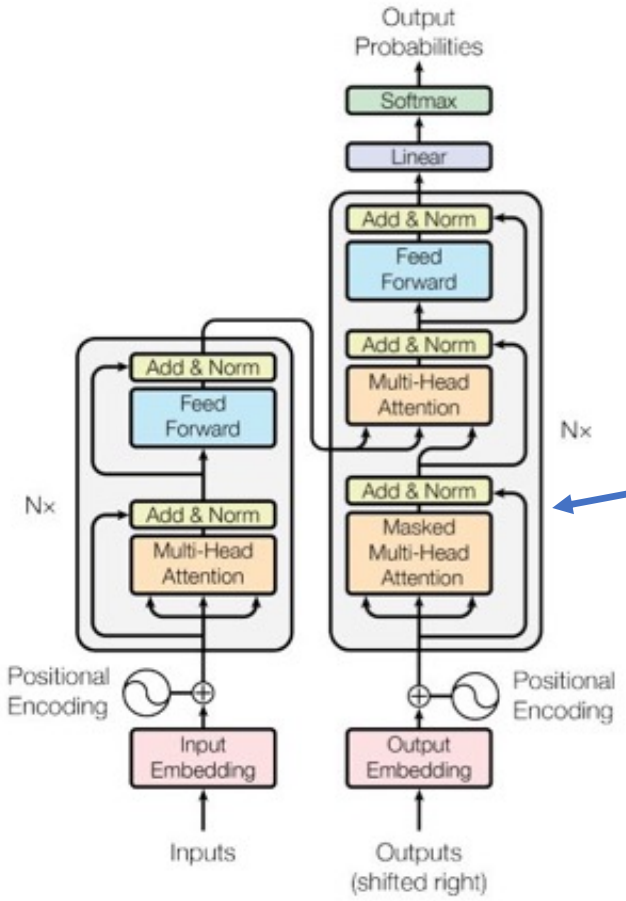
The first layer of multi-head attention, similar to the encoder one, requires masking in order to parallelize learning.

Parallelization is a good thing because it allows the model to train faster.

Figure credit: Vaswani et. al

Decoder

*Encoder-like architecture which includes a **multi-head attention** over the output of the encoder stack.*



The training algorithm knows the entire expected output. It hides (**masks) a part of this known output** sequence for each of the **parallel operations** .

- When it executes #1 - it hides (masks) the entire output.
- When it executes #2 - makes only input 1 visible
- When it executes #3 - makes only input 1 and 2 visible
- Etc.

Figure credit: Vaswani et. al

let's go into a little more detail

- **Step 1:** tokenization (only before first encoder and generally outside the model)
- Formelly : transformer could use 2 tokenizers
  - One for input sentences
  - One for output sentences

	Tokenization is compression																							
Word	Tokenization											is		compression										
Piece	To		##ken			##ization						is		compression										
Char	T	o	k	e	n	i	z	a	t	i	o	n	i	s	c	o	m	p	r	e	s	s	i	o	n

Second step : Embedding (only before first encoder)

- Transform each token (an integer) into a vector (512 bytes)

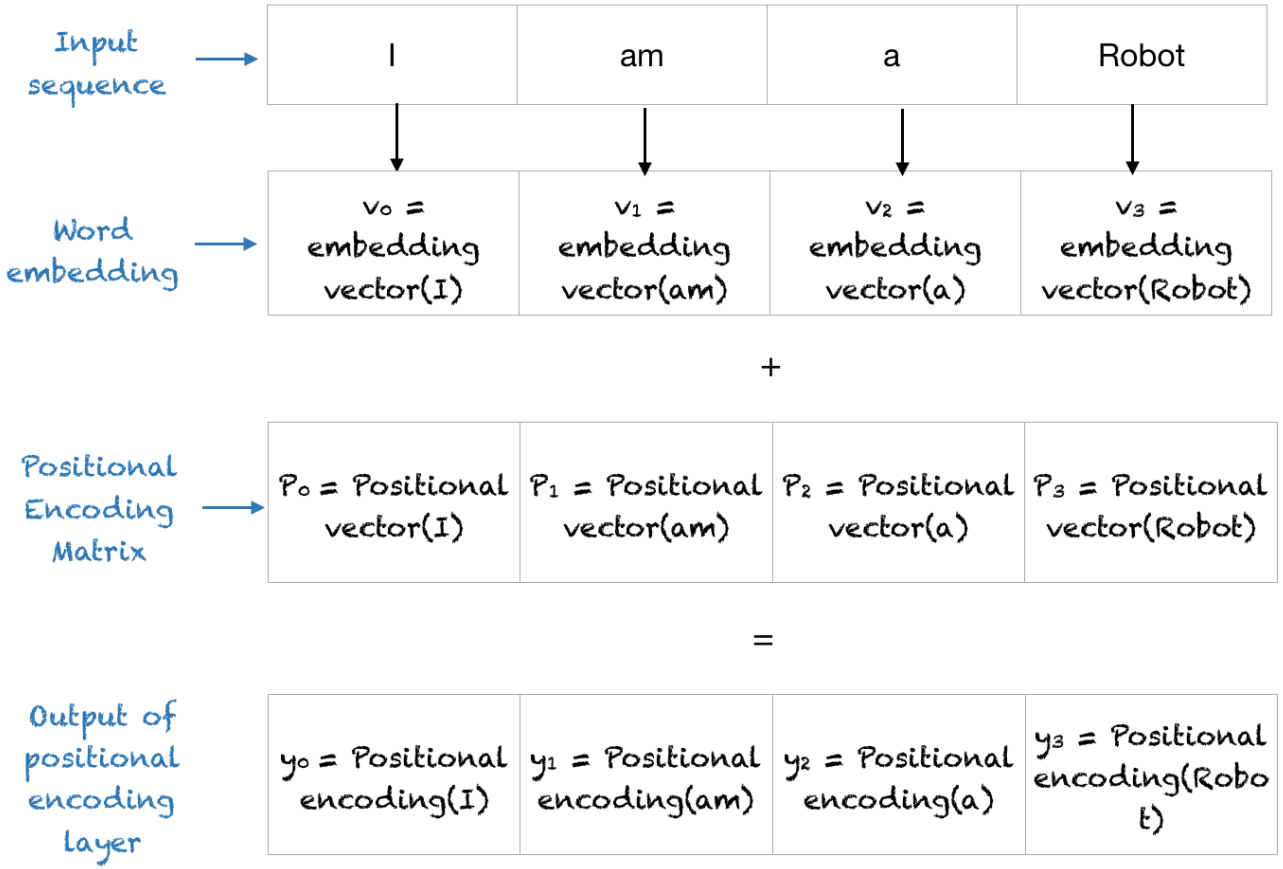
- More information on: https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

Third step : Positional embedding (only before first encoder)

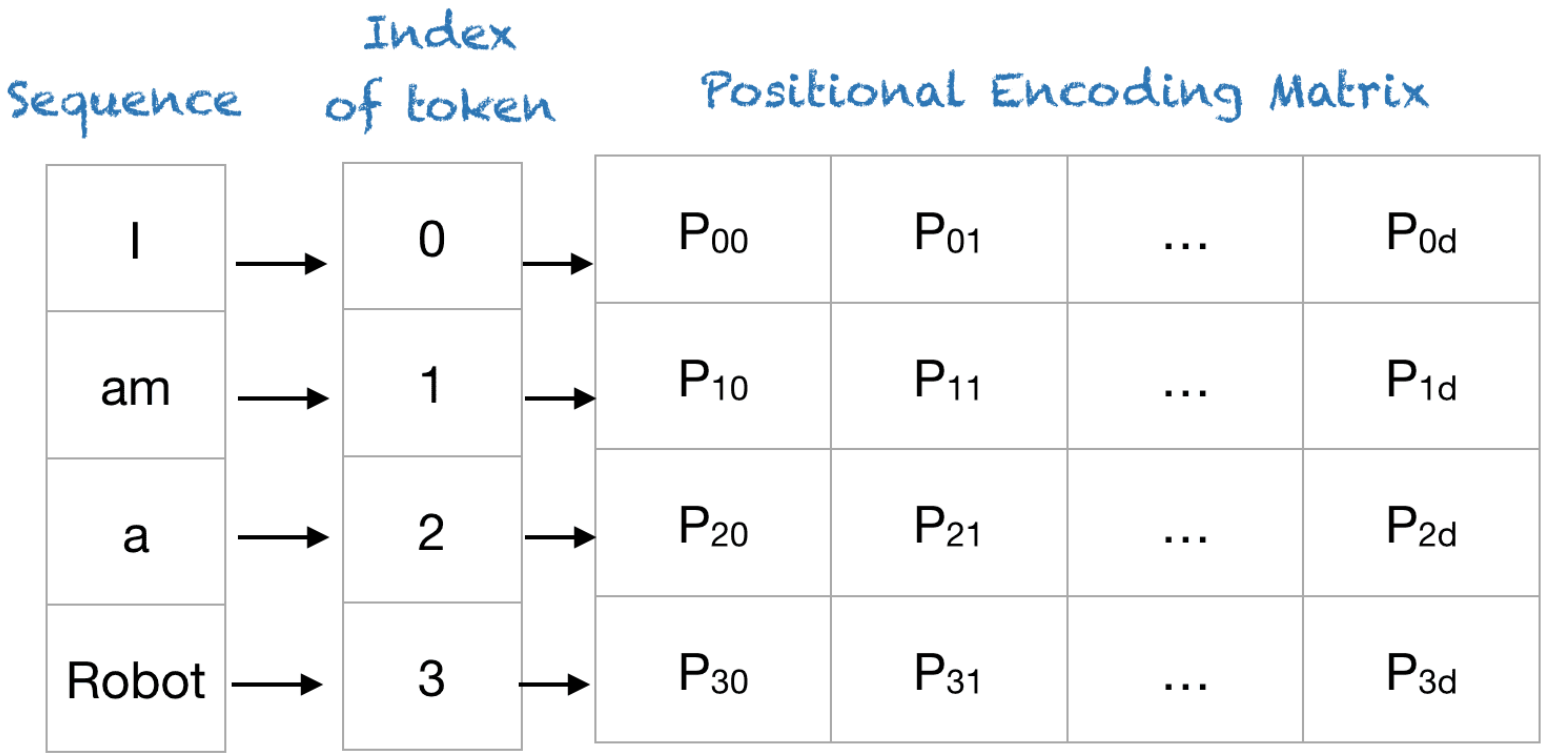
Add positional information to the embedding

If we don't add positional encoding, we get a BOW

Positional encoding goals:
- Removes the sequentiality present in RNNs
- Allows the code to be parallelized



What is positional vector ?



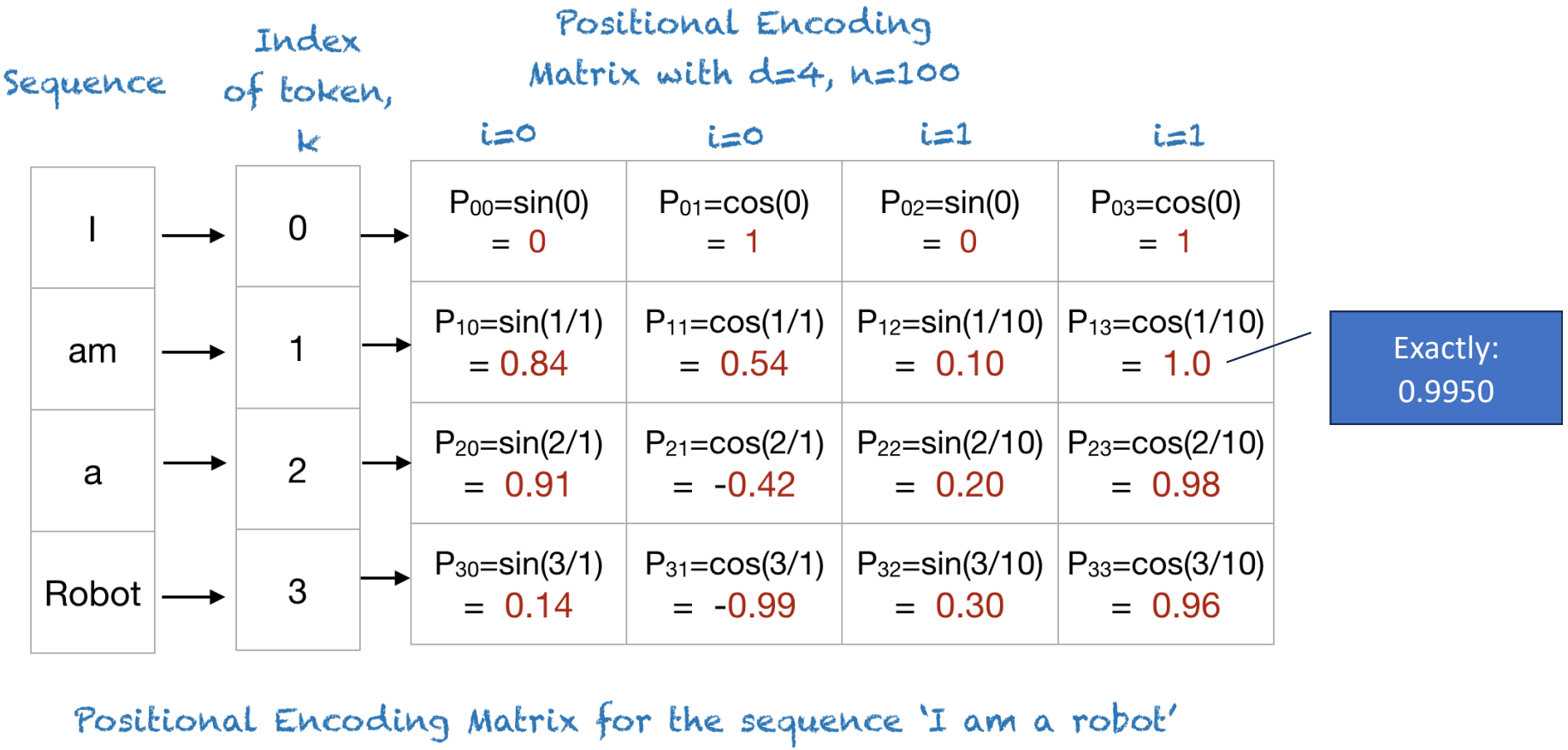
Positional Encoding Matrix for the sequence 'I am a robot'

How each vector is calculated ?

- L: length of the input sequence
- k: position of a token in the input sequence
- d: dimension of the output embedding space
- n: User-defined scalar, set to 10,000 by the authors of Attention Is All You Need.
- P(k, j): Position function for mapping a position □ in the input sequence to index (□,□) of the positional matrix

- $P(k, 2i) = \sin(\frac{k}{n^{\frac{2i}{d}}})$ and $P(k, 2i+1) = \cos(\frac{k}{n^{\frac{2i}{d}}})$

An example of values



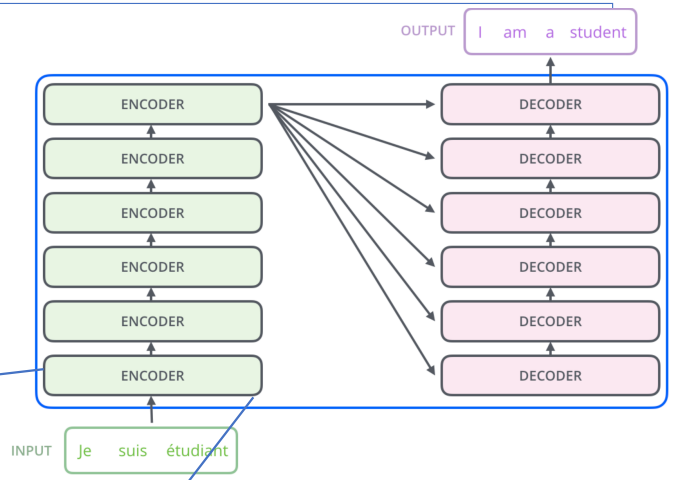
Positional Encoding Matrix for the sequence 'I am a robot'

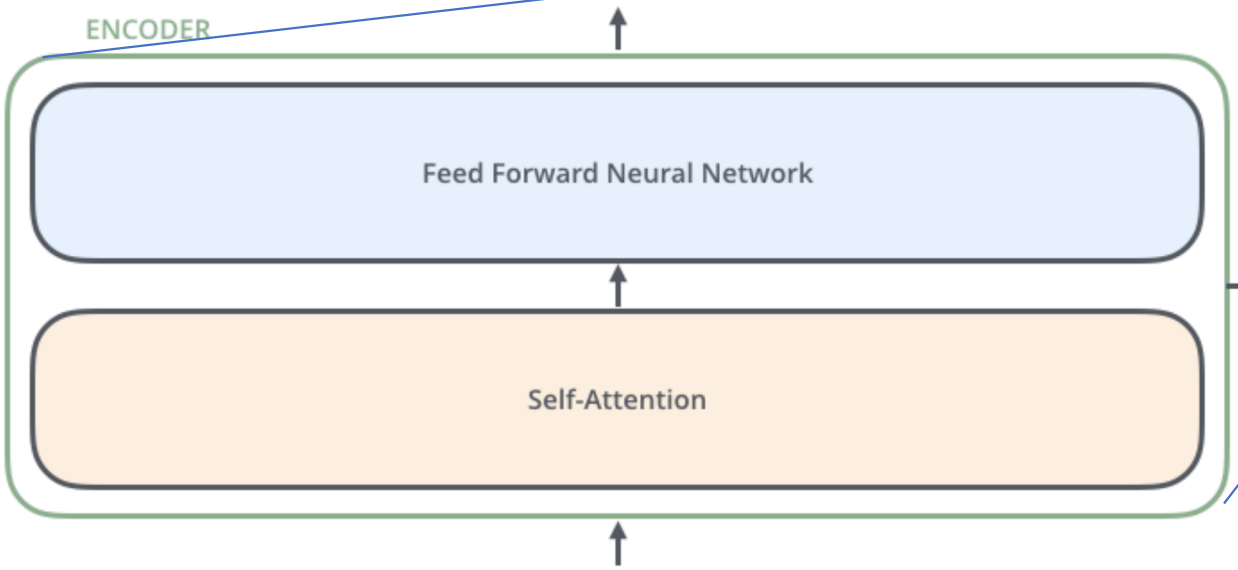
Now, we detail the encoder structure

The **encoders** are **all identical in structure**

- They do not share weights

- Each one is broken down into two sub-layers





outputs of the self-attention are fed to a feed-forward neural network. The exact same one is independently applied to each position.

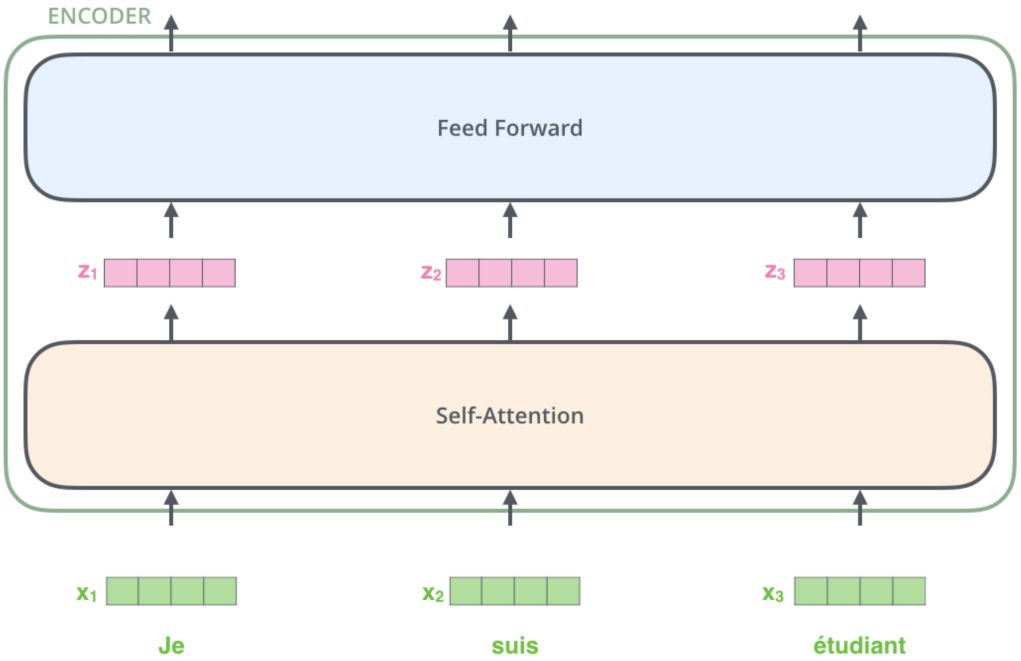
helps the encoder look at other words in the input sentence as it encodes a specific word

https://jalammar.github.io/illustrated-transformer/

Encoder structure

- Key property of Transformer: word in each position flows through its own path in the encoder.
- There are dependencies between these paths in the self-attention layer.
- Feed-forward layer does not have those dependencies

  => various paths can be executed in parallel !

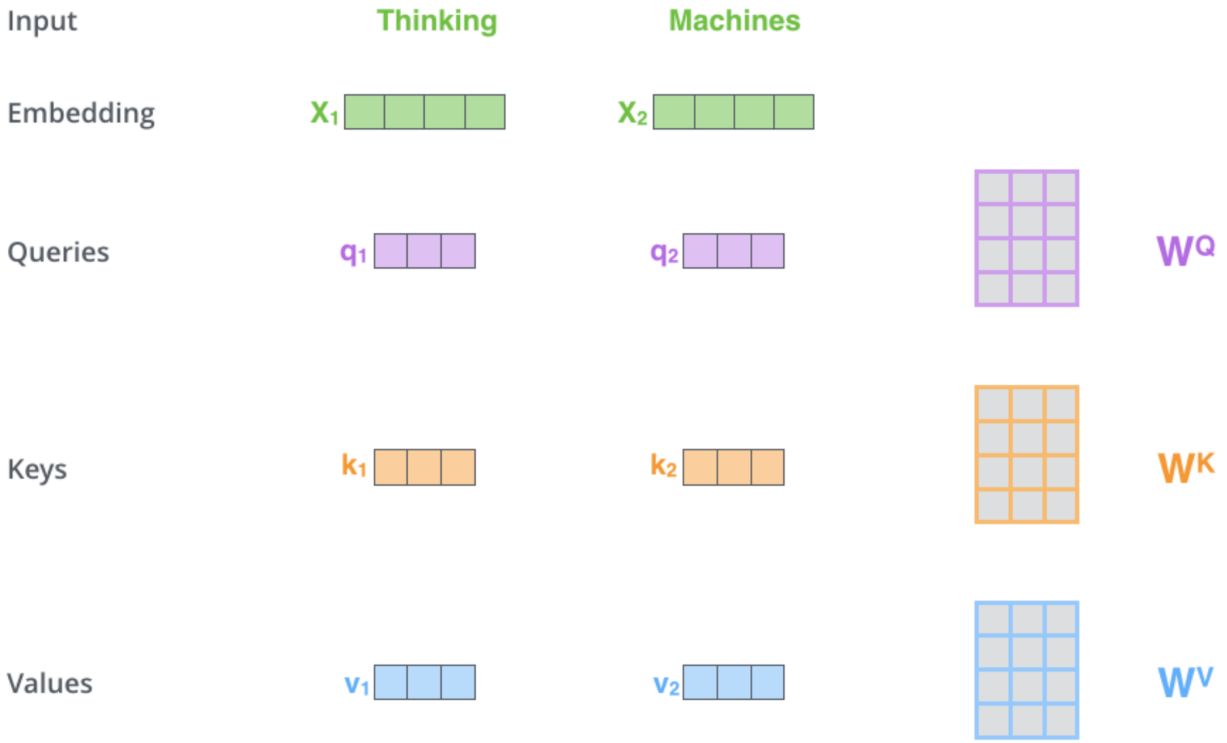


Self-Attention

Step1: create three vectors from each of the encoder's input vectors (typically smaller dimension: 64):

- Query,
- Key,
- Value.

by multiplying the embedding by three matrices that we **trained** during the training process.



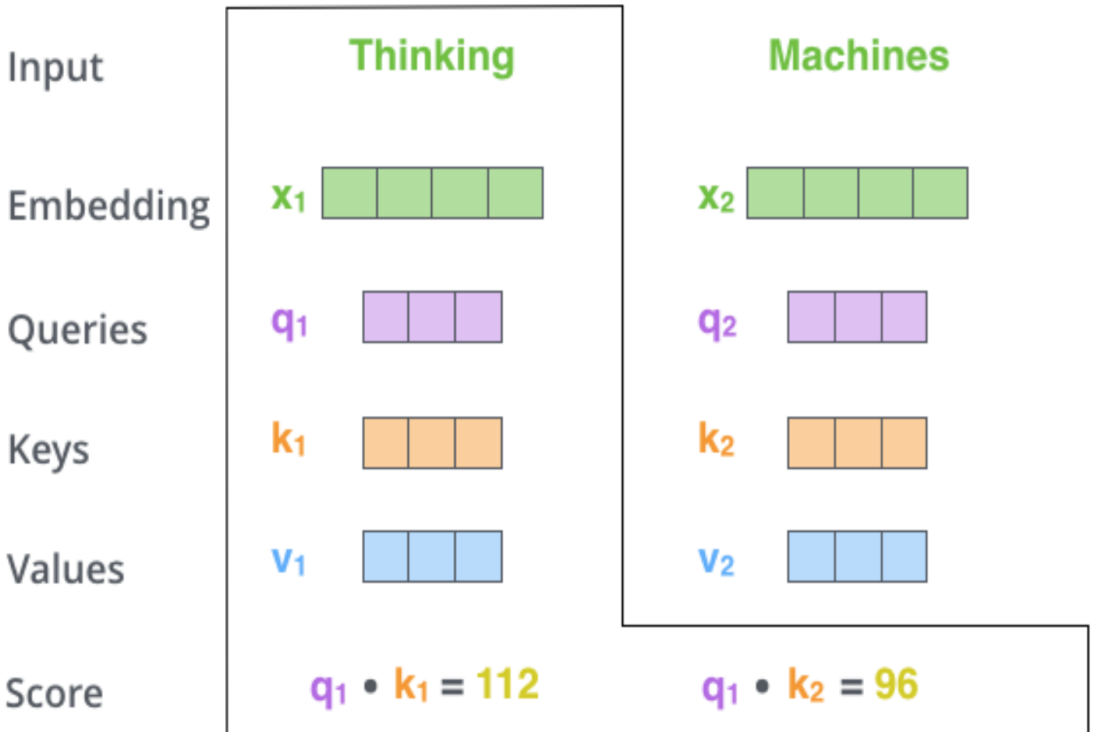
https://jalammar.github.io/illustrated-transformer/

Self-Attention

Step2: calculate a score (like we have seen for regular attention!)

→ calculating the degree of attention to be paid to other words when we encode a word

→ Take dot product of the query vector with the key vector of the respective word we're scoring.

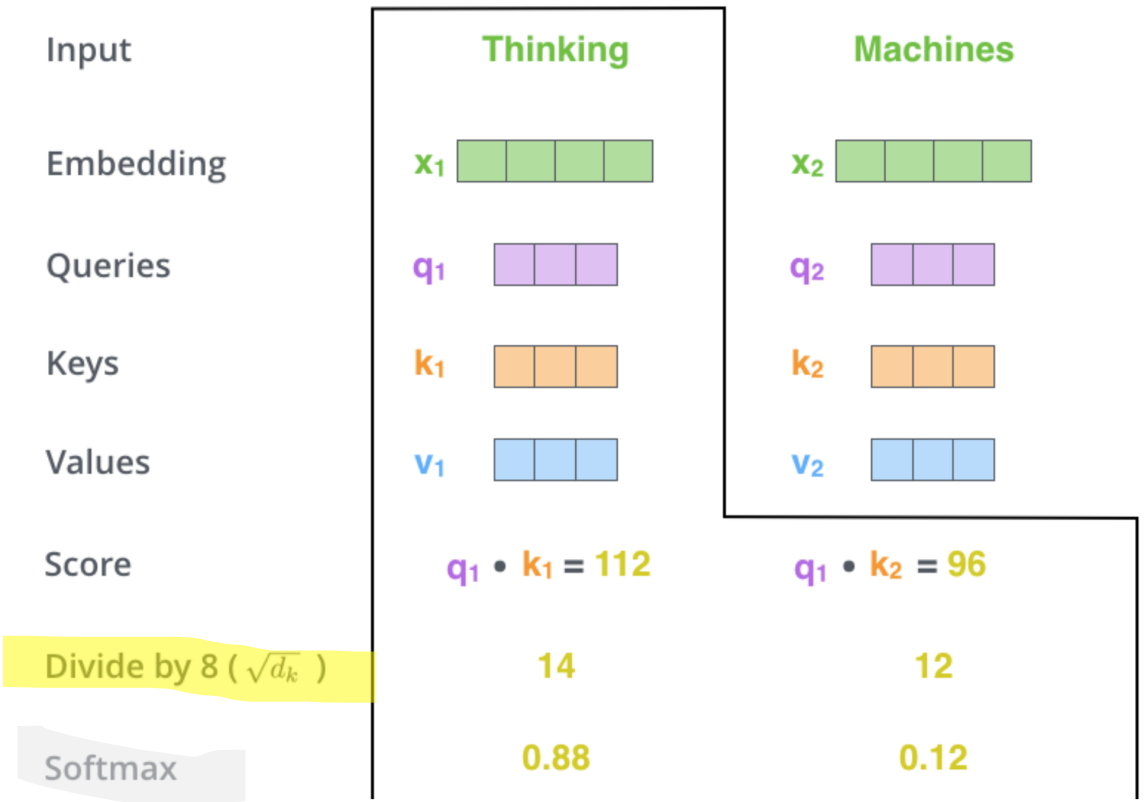


Self Attention

- **Step3** divide scores by the square root of the dimension of the key vectors (more stable gradients).
  - Dimension: 64
  - $\sqrt{64} = 8$

- **Step 4** pass result through a softmax operation (all positive and add up to 1).

- **Intuition**: softmax score determines how much each word will be expressed at this position.



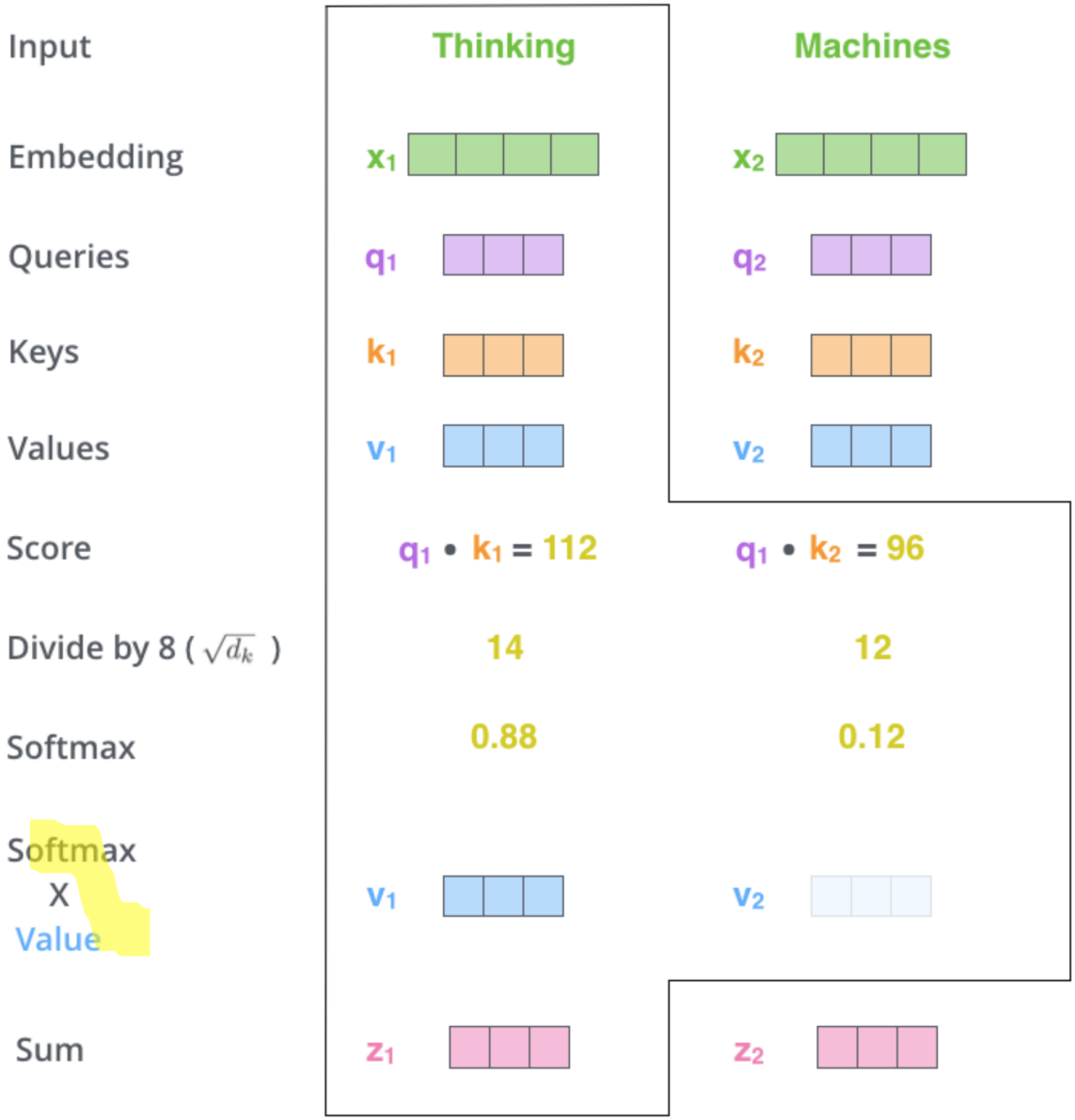
Intuition: softmax score determines how much each word will be expressed at this position.

https://jalammar.github.io/illustrated-transformer/

Self Attention

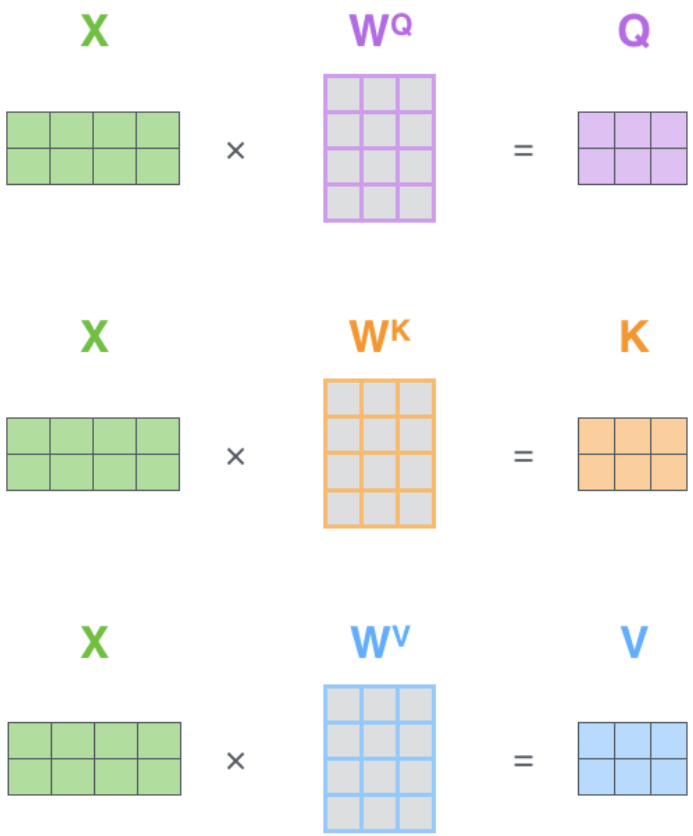
- **Step5:** multiply each value vector by the softmax score
  - **Intuition:** keep the values of the words we want to focus on intact, and drown out the irrelevant words (by multiplying them by lower case numbers like 0.001, for example).
- **Step6**: sum up the weighted value vectors. This produces the output of the self-attention layer at this position

The resulting vector is the one we can send to the feedforward neural network (second step of encoder).



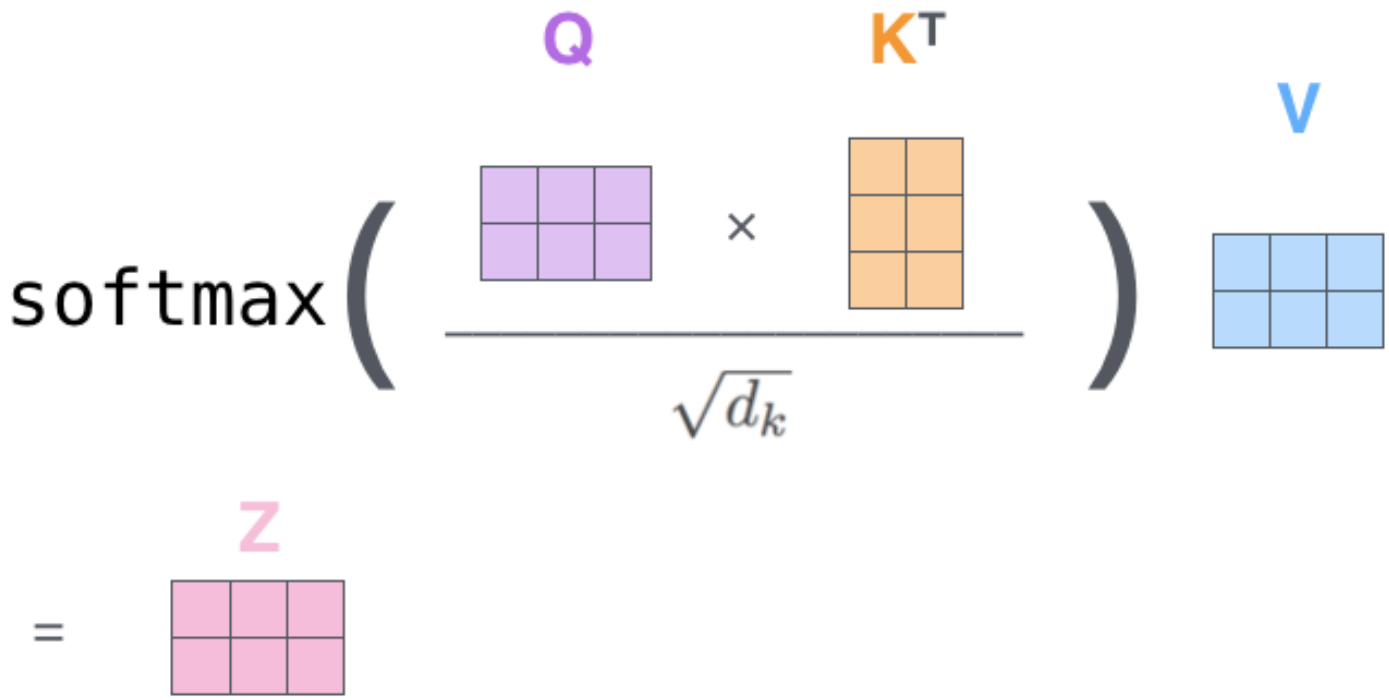
https://jalammar.github.io/illustrated-transformer/

Matrix Calculation of Self-Attention



https://jalammar.github.io/illustrated-transformer/

Matrix Calculation of Self-Attention



https://jalammar.github.io/illustrated-transformer/

What is multi-head attention

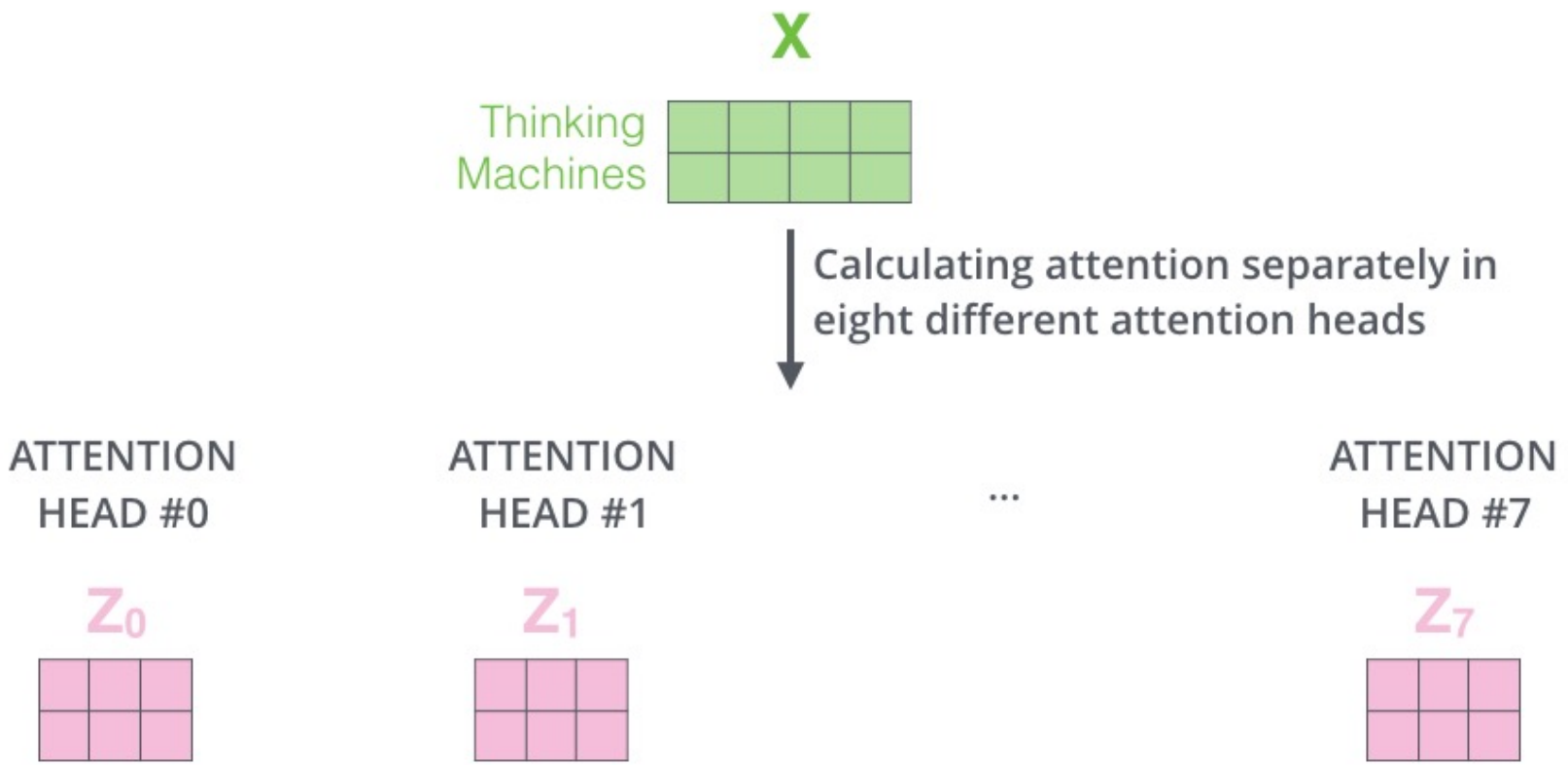
- Multi-head attention:
  - improves the performance of the attention layer by using multiple attention (8 for example)
  - As with CNNs, when several convolution filters are used

- Expands the model's ability to focus on different positions for the same current word using in multiple attentions
  - z1 (previous slide) contains a little bit of every other encoding **(sum operation)**, but it could be dominated by the actual word itself
  - if we are translating a sentence:
    - « The animal did not cross the street because it was too tired »

    → it would be useful to know which word "**it**" refers to.

  - With multi-headed attention, we have not just one, but several sets of matrices of weight Request/Key/Value

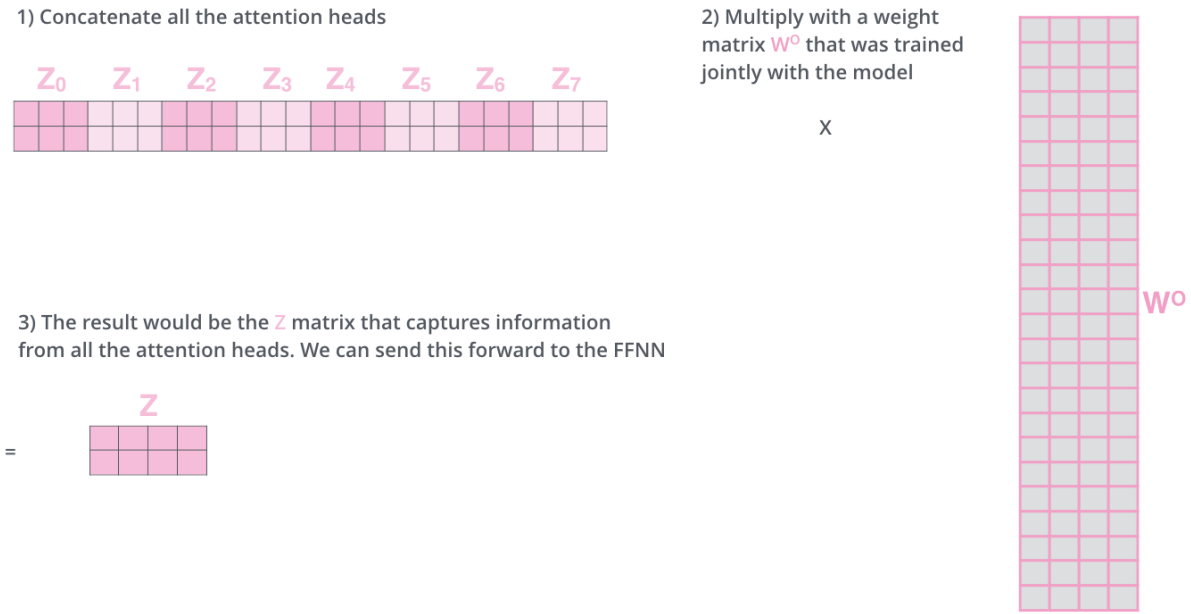
Multi-head attention



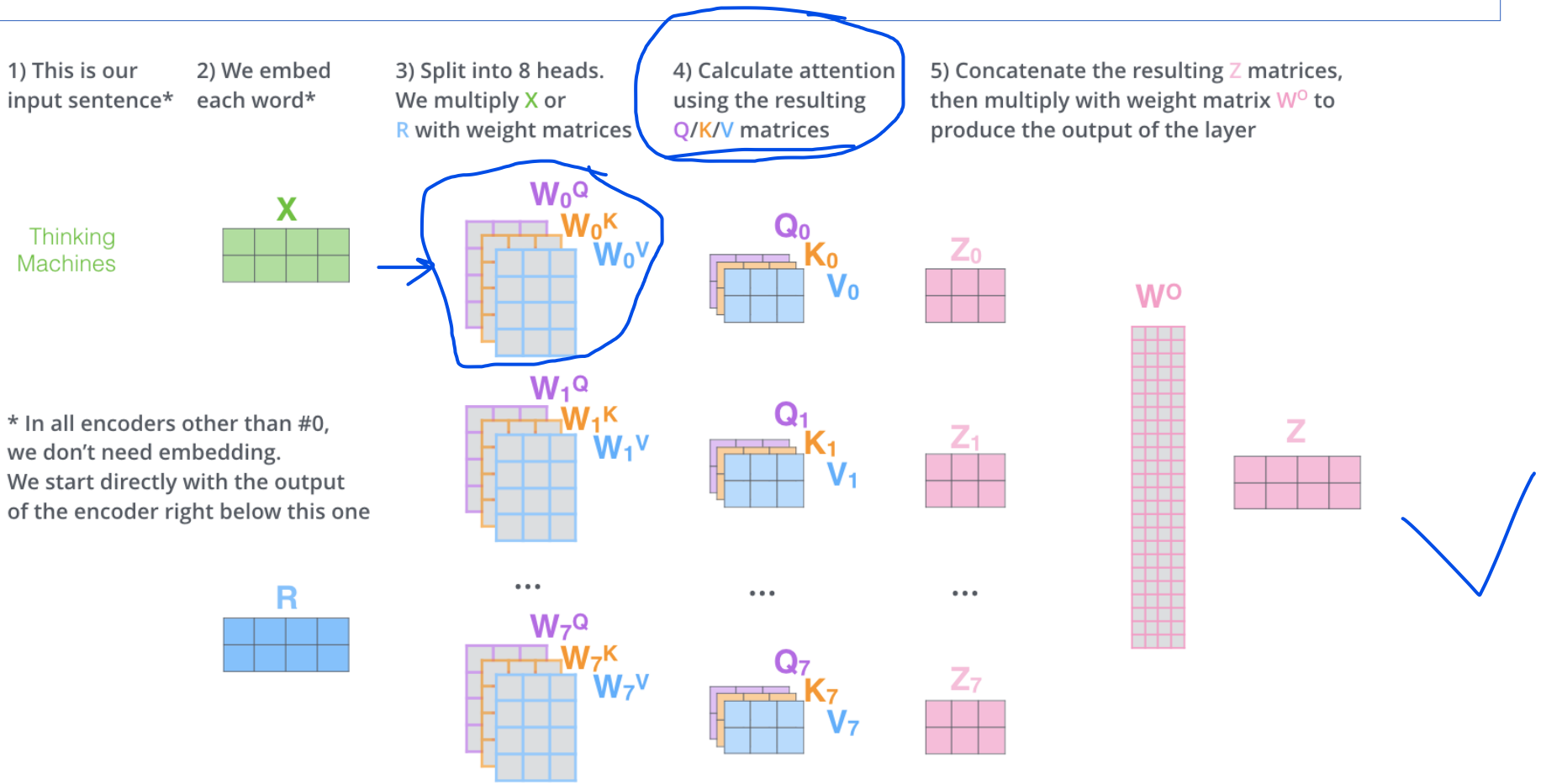
https://jalammar.github.io/illustrated-transformer/

From multi-head attention to « one attention »

- The feed-forward layer is not expecting eight matrices
  - Expects a single matrix (a vector for each word).
  - We need a way to condense these eight down into a single matrix.

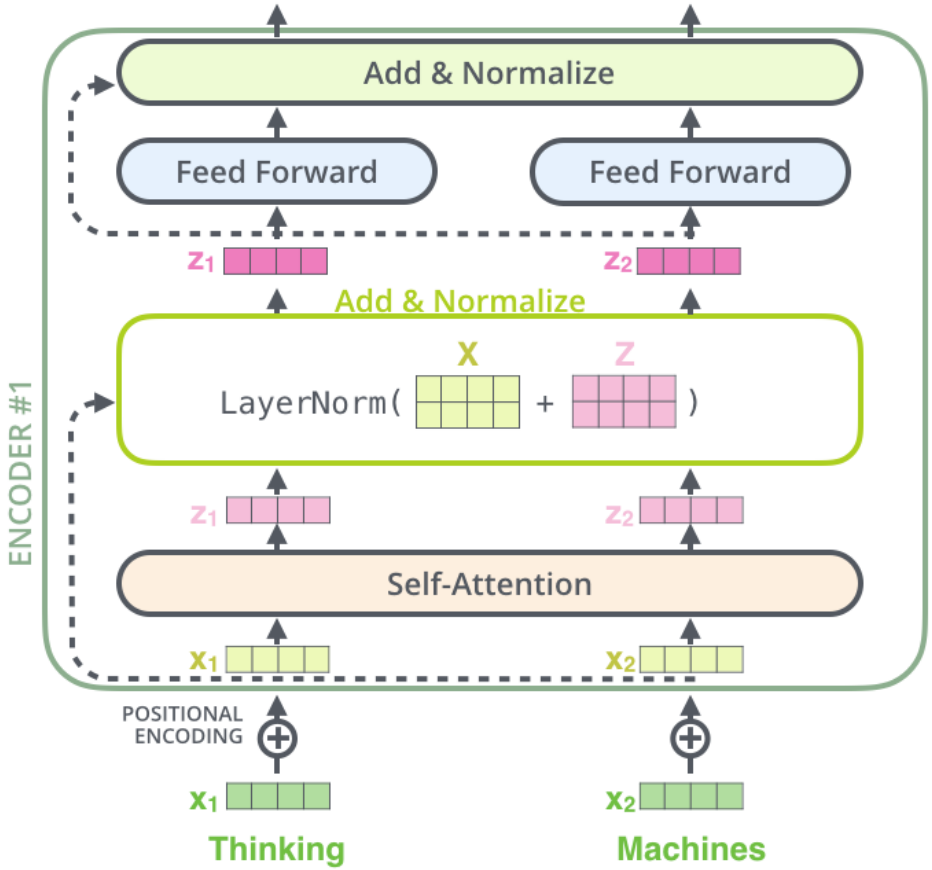


Multi-head attention layer



https://jalammar.github.io/illustrated-transformer/

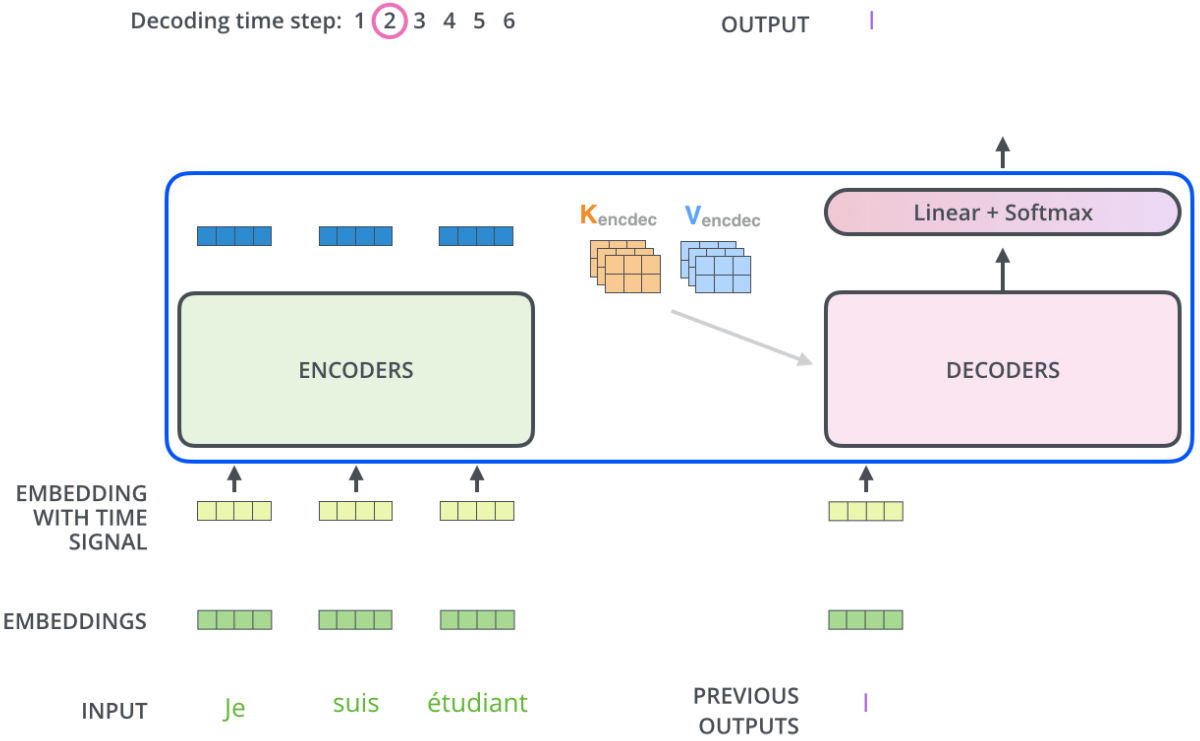
Encoder layer



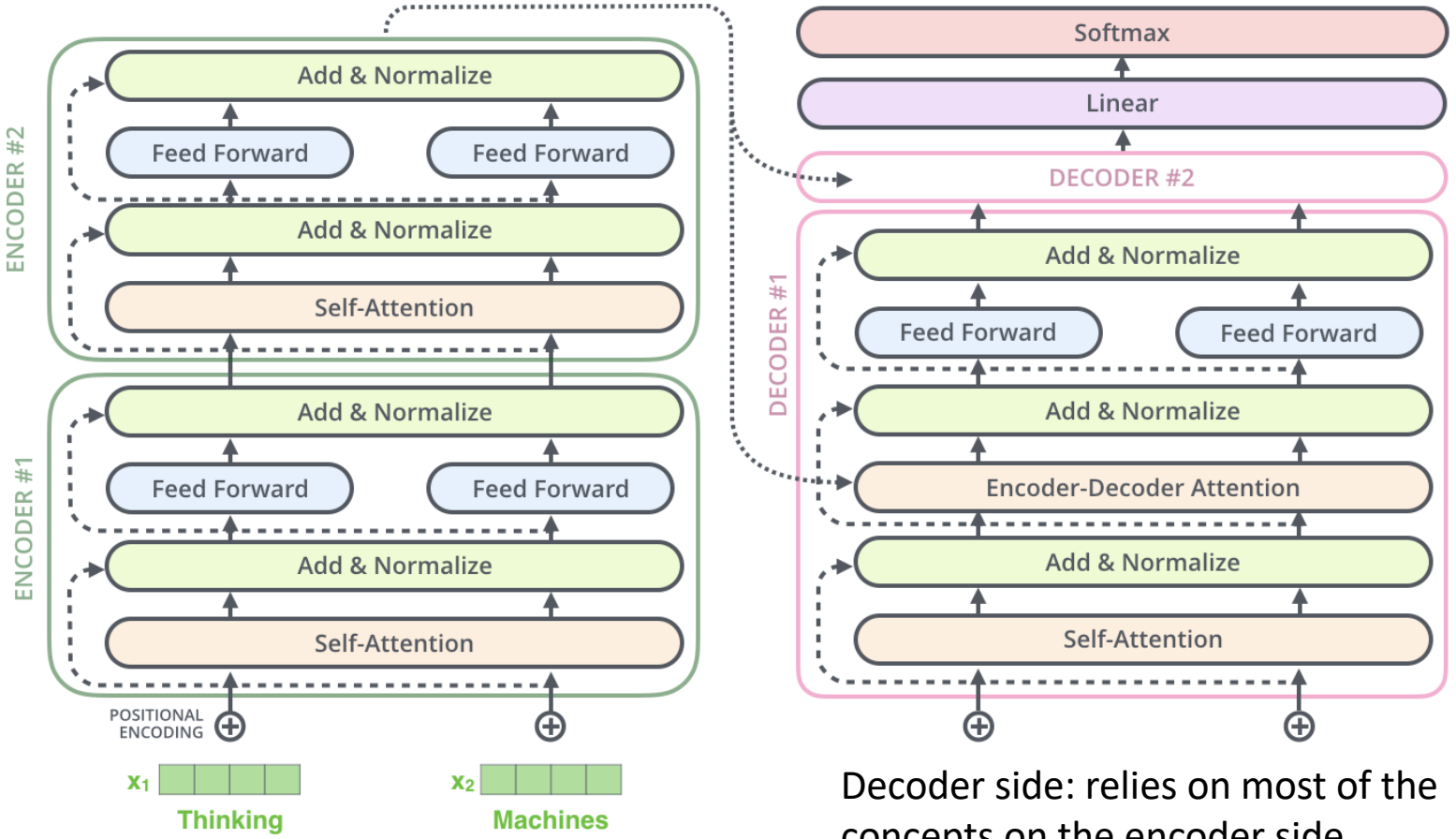
https://jalammar.github.io/illustrated-transformer/

Decoder layer

- Same architecture as encoder
- Same approach as Seq2Seq architecture



Encoder-decoder architecture



Decoder side: relies on most of the concepts on the encoder side

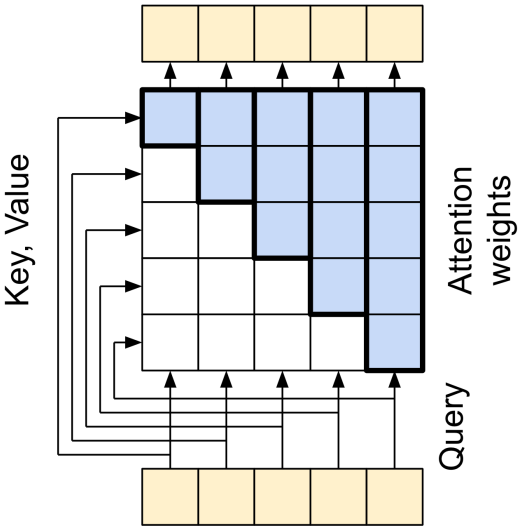
Masking in decoder layer

- A mask indicates which inputs are not to be used.
- A mask is usually used to indicate that padding should not be taken into account.

```
def create_padding_mask(seq):
  seq = tf.cast(tf.math.equal(seq, 0), tf.float32)
  # add extra dimensions to add the padding to the attention logits.
  # (batch_size, 1, 1, seq_len)
  return seq[:, tf.newaxis, tf.newaxis, :]
```

- But in the decoder architecture, it is necessary to mask the elements in the input sequence that correspond to the future sequence to be decoded.
  - We have removes the sequentiality present in RNNs in order to allows the code to be parallelized
  - Causal Masking is Important at the training stage

```
def create_look_ahead_mask(size):
  mask = 1-tf.linalg.band_part(tf.ones((size, size)),
                               -1, 0)
  return mask  # (seq_len, seq_len)
```



https://www.tensorflow.org/text/tutorials/transformer

How to train / use transformer model ?

- Apart from the necessary mask preparation, the Seq2Seq architecture can be used in the same way.
- Nevertheless, most of the models available today use very large volumes of data and fairly complex strategies to improve the performance of the final model.
- It is therefore generally preferable to reuse pre-trained models and proceed by transfer learning with or without fine tuning.
- Let's take a look at BERT, the model that popularized transformers.

BERT (Bidirectional Encoder Representations from Transformers)

- Mainly the encoder part of the transformer
- Main properties
  - **1: BERT is pre-trained on an absurd amount of data**
    - **Several variation: bert, biobert, flaubert, camembert, …**
  - **2: BERT is able to account for a word's context**
    - BERT returns different vectors for the same word depending on the words around it
  - **3: BERT is open-source.**
    - BERT's code is published on GitHub (https://github.com/google-research/bert)
- BERT involves two stages:
  - **Unsupervised pre-training (**few days on many Cloud TPUs)
  - followed by **supervised task-specific fine-tuning** (30 minutes on a single Cloud TPU)
    - For fine-tuning, one or more output layers are typically added to BERT.

BERT pre-training

- In the first stage, BERT is pre-trained on two tasks:
  - Masked Language Model (MLM):
    - Given a sequence of tokens, some of them are masked. The objective is then to predict the masked tokens.
  - Next Sentence Prediction (NSP):
    - Given two sentences, the model predicts if the second one logically follows the first one.
- In the second stage, BERT is trained on:
  1. Sequence Classification
  2. Named Entity Recognition (NER)
  3. Natural Language Inference (NLI) - is the task of determining whether the given "hypothesis" logically follows from the "premise".
  4. Question Answering (Q&A)

How the BERT tokenizer works

- !pip install transformers
- from transformers import AutoTokenizer
- tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")

- sequence="don't be so judgmental »
- sent = tokenizer.tokenize(sequence)
  - ['don', "'", 't', 'be', 'so', 'judgment', '##al']
- ids = tokenizer.convert_tokens_to_ids(sent)
  - [2123, 1005, 1056, 2022, 2061, 8689, 2389]
- new_sent = tokenizer.decode(model_ ids)
  - don't be so judgmental

- model_inputs = tokenizer(sequence)
  - [101, 2123, 1005, 1056, 2022, 2061, 8689, 2389, 102]
- new_sent = tokenizer.decode(model_inputs)
  - [CLS] don't be so judgmental [SEP]

How to use transformer model– Token embedding

- from transformers import AutoTokenizer , TFAutoModel

- tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
- model = TFAutoModel.from_pretrained("distilbert-base-uncased")

- raw_inputs = ["I've been waiting for a Deep Learning course my whole life.",
  "I hate this so much!"]

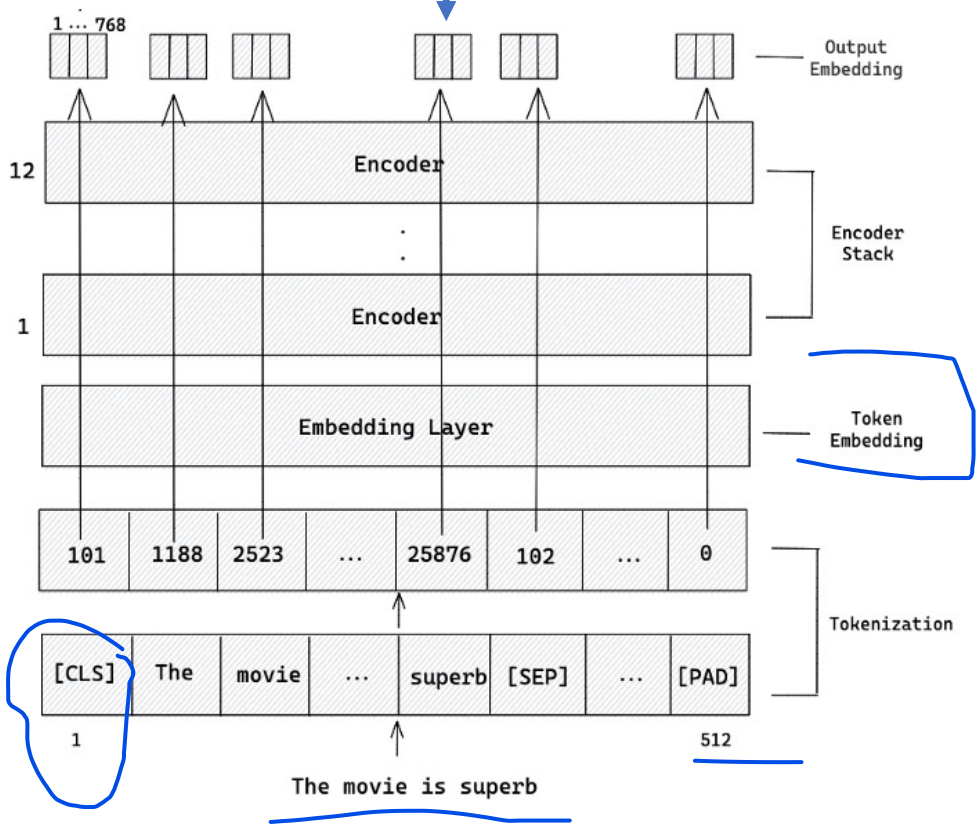
- inputs = tokenizer(raw_inputs, padding=True, truncation=True, return_tensors="tf")
- outputs = model(**inputs)

- print(outputs.last_hidden_state.shape)
  - (2, 16, 768) # 2 sentences of 16 tokens and embedding size is 768

BERT's main outputs

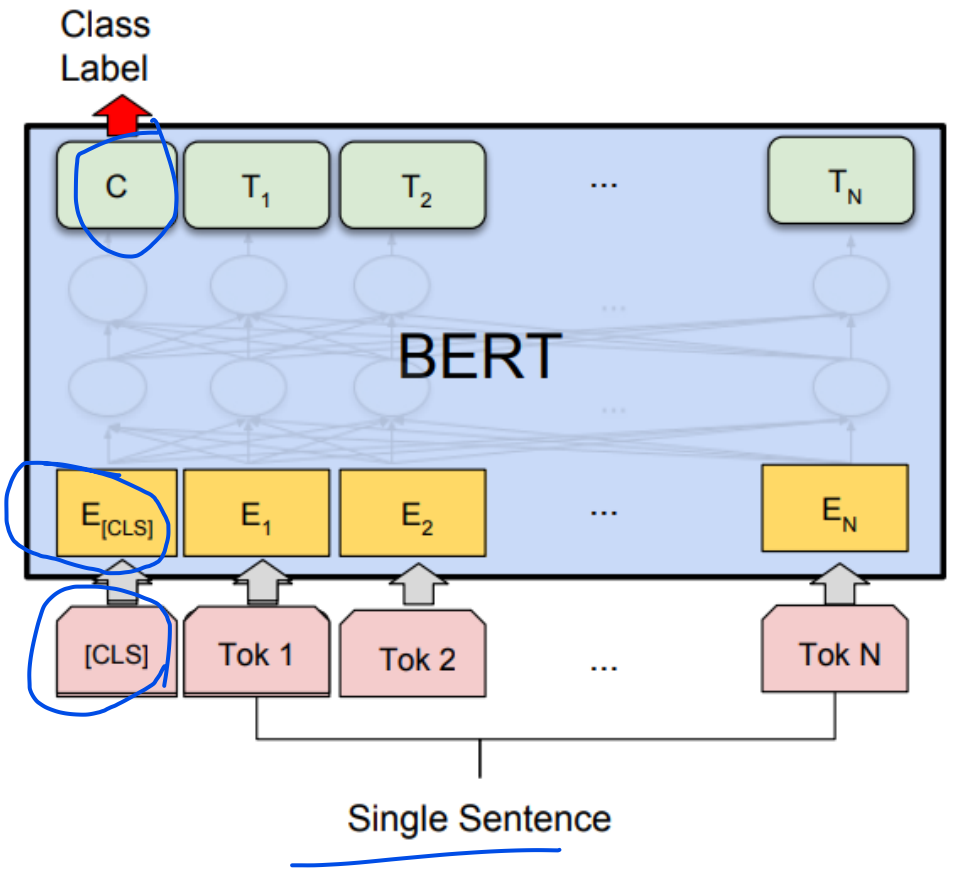
Sentence embedding

embedding of each token in the sentence



How to use BERT for sentiment analysis/document classification task

It is of course possible to finetune BERT to suit your own labels.



BERT for sentiment analysis

- Using a pre-trained model
  - from transformers import pipeline

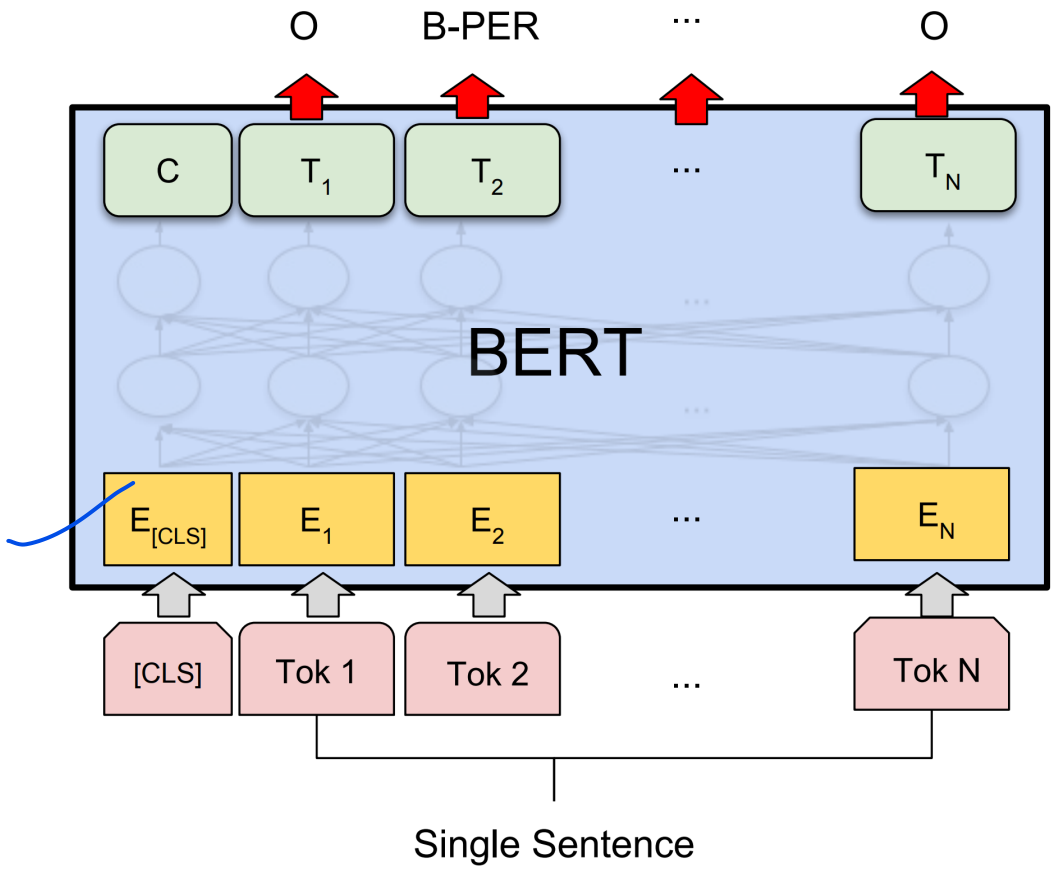
  - classifier = pipeline("sentiment-analysis")
  - classifier("I've been waiting for a HuggingFace course my whole life.")
    - [{'label': 'POSITIVE', 'score': 0.9598050713539124}]

- BERT fine tuning for sentiment analysis
  - tokenizer = AutoTokenizer.from_pretrained(("distilbert-base-uncased")
  - model = TFAutoModelForTokenClassification.from_pretrained("distilbert-base-uncased")
  - batch = dict(tokenizer(corpus, padding=True, truncation=True, return_tensors="tf"))

  - model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
  - model.fit(batch, labels, epochs=5)

How to use BERT for NER

It is of course possible to finetune BERT to suit your own labels.



BERT for NER

- Using a pre-trained model
  - from transformers import pipeline

  - ner = pipeline("ner", grouped_entities=True)
  - ner("My name is Sylvain and I work at Hugging Face in Brooklyn.")
  - [{'entity_group': 'PER', 'score': 0.9981694, 'word': 'Sylvain', 'start': 11, 'end': 18},
    {'entity_group': 'ORG', 'score': 0.97960204, 'word': 'Hugging Face', 'start': 33, 'end': 45},
    {'entity_group': 'LOC', 'score': 0.9932106, 'word': 'Brooklyn', 'start': 49, 'end': 57}]

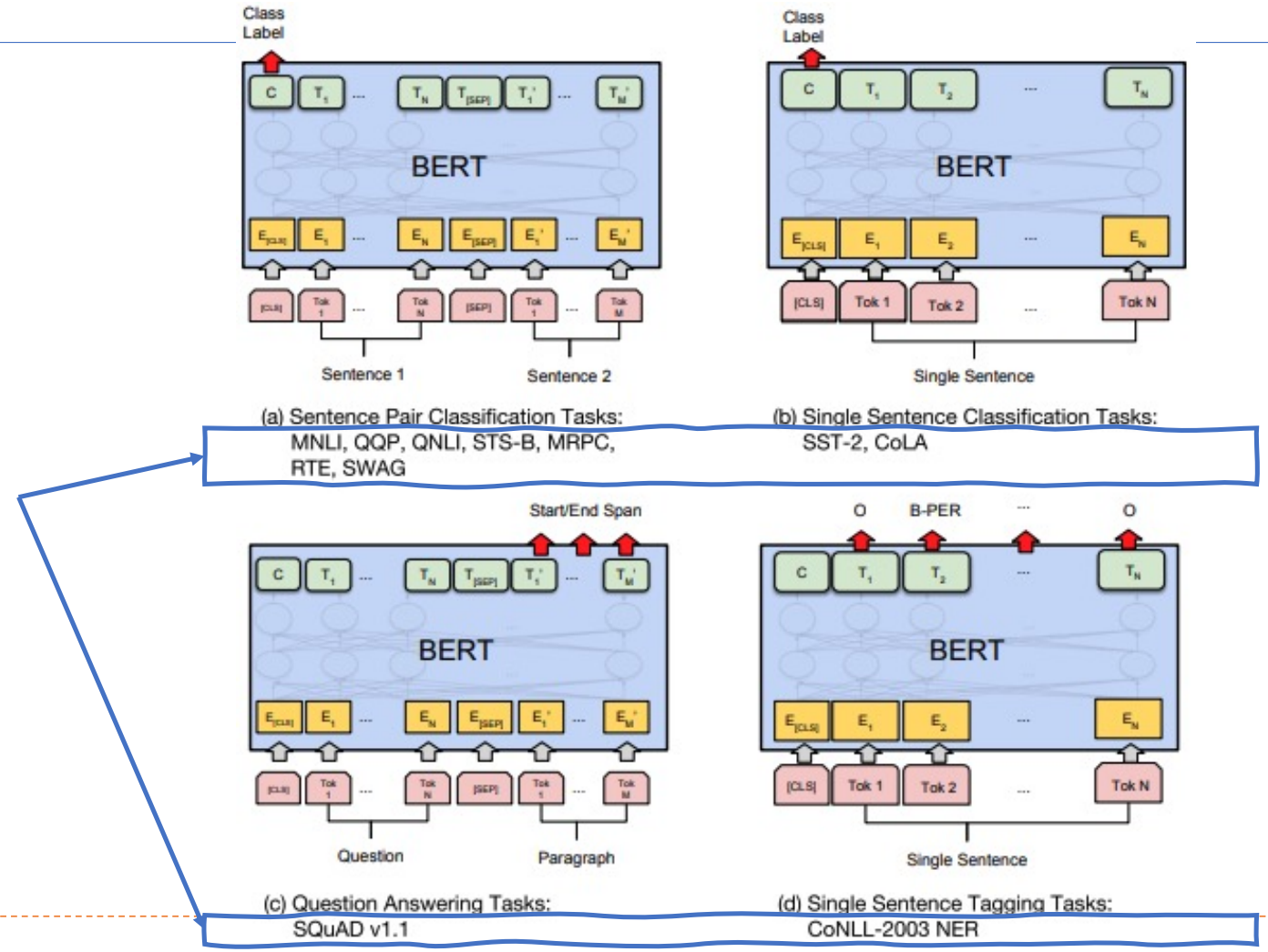
- BERT fine tuning
  - tokenizer = AutoTokenizer.from_pretrained(("distilbert-base-uncased")
  - model = TFAutoModelForSequenceClassification.from_pretrained(("distilbert-base-uncased",
    id2label=id2label, label2id=label2id))

  - tf_train_dataset = << prepare the dataset >> # See hugginface tutorial

  - model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
  - model.fit(tf_train_dataset, epochs=5)

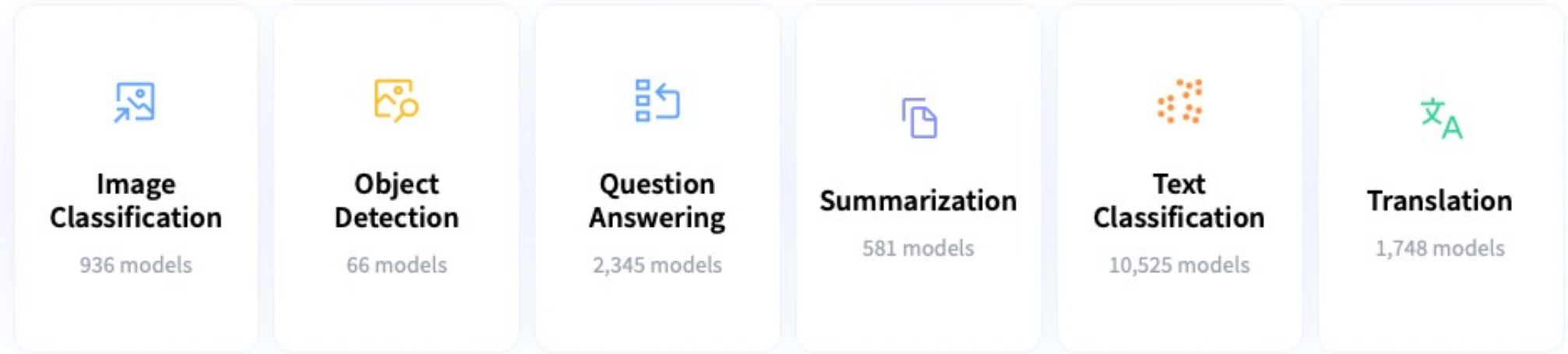
Main BERT Task

Usual dataset



BERT for translation

- Not directly usable (see how BERT was trained).
- However, there are models with a similar architecture for this
- https://huggingface.co
  - a library that allows existing models to be fine-tuned to different tasks



BERT for translation

- Not directly usable (see how BERT was trained).
- However, there are models with a similar architecture for this
- https://huggingface.co
  - Choose the right model for the right job

    → For translation use for example: "Helsinki-NLP/opus-mt-fr-en"

  - from transformers import pipeline

  - translator = pipeline("translation", model="Helsinki-NLP/opus-mt-fr-en")
  - translator("Ce cours est produit par Hugging Face.")
    - [{'translation_text': 'This course is produced by Hugging Face.'}]

Transformer for text generation

- Like translation → BERT model is not adequate for text generation
  - Choose the right model for the right job
    - Just change model

    → For translation use for example: ="distilgpt2"

  - from transformers import pipeline

  - generator = pipeline("text-generation", model="distilgpt2")
  - generator("Michel RIVEILL is", max_length=30, num_return_sequences=2)
    - [{'generated_text': 'Michel RIVEILL is a professor of psychology at Yale University. She writes about the psychology of the public.'},
      {'generated_text': 'Michel RIVEILL is a PhD student based on research on health and well-being at the University of California Riverside. He is an author,'}]

Summary

- Transformer is an architecture
  - A model is an instance of this architecture trained for one or more tasks
  - Some final models use only encoder part
    - Token embedding, Sentence embedding, Sentiment analysis, NER, POS,
  - Some others, use mainly decoder part
    - Q&A, Translator, Text generation, Text summarization
  - The model can be used as long as the task to be solved is identical to the one used to train it.
  - We can fine-tune a model (a few epochs) to adapt it to the specific characteristics of our data (labels, for example).

- Input points
  - https://huggingface.co/docs/transformers/task_summary
    - a French company that allows you to share models
    - The most important thing is the choice of models:
      - Some models use only one of the two platforms: Tensorflow or Pytorch

Some reference

- Attention is all you need - https://arxiv.org/abs/1706.03762
  - Original transformer paper
- Elmo - https://arxiv.org/abs/1802.05365
  - Contextualised embedding
- BERT - https://arxiv.org/abs/1810.04805
  - Mainly for contextual embedding (encoder part of transformer)
  - https://blog.paperspace.com/bert-pre-training-of-deep-bidirectional-transformers-for-language-understanding/
  - https://keras.io/examples/nlp/pretraining_BERT/
- GPT - https://arxiv.org/abs/2005.14165
  - Mainly for text summarization (decoder part of transformer)

- Transformer from scratch
  - https://pyimagesearch.com/2022/09/05/a-deep-dive-into-transformers-with-tensorflow-and-keras-part-1/
  - https://pyimagesearch.com/2022/09/26/a-deep-dive-into-transformers-with-tensorflow-and-keras-part-2/

Some global reference on NLP

- https://blog.paperspace.com/tag/natural-language-processing/

- https://keras.io/examples/nlp/
- https://huggingface.co/docs/datasets/main/en/quickstart#nlp

MSc. DSAI lab

- Add a new step on your notebook
  - Preprocessing
  - Seq2Seq architecture
  - Improved seq2seq architecture
    - Stacked bi-LSTM (optional)
    - Context available to each decoder step (optional)
  - Attentional Seq2Seq architecture
    - Build your own Seq2Seq class
    - Use Attention tensorflow.keras class (optional)
    - Use MultiHeadAttention tensorflow.keras class (optional)
  - → Transformer architecture
    - Build your own transformer (you can use Attention tensorflow.keras class)
      - Piece of code: https://www.tensorflow.org/text/tutorials/transformer?hl=fr
    - Use pre-trained model: https://huggingface.co/ (optional)

Submit your notebook - written and executed (and a pdf copy of it on lms.univ-cotedazur.fr)

Positional encoding

```
def positional_encoding(position, d_model):
    def get_angles(pos, i, d_model):
        angle_rates = 1 / np.power(10000, (2 * (i//2)) /
np.float32(d_model))
        return pos * angle_rates

    angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                            np.arange(d_model)[np.newaxis, :],
                            d_model)

    # apply sin to even indices in the array; 2i
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

    # apply cos to odd indices in the array; 2i+1
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])

    pos_encoding = angle_rads[np.newaxis, ...]

    return tf.cast(pos_encoding, dtype=tf.float32)
```

https://www.tensorflow.org/text/tutorials/transformer