

Introduction of Large Language Models



Li Chen

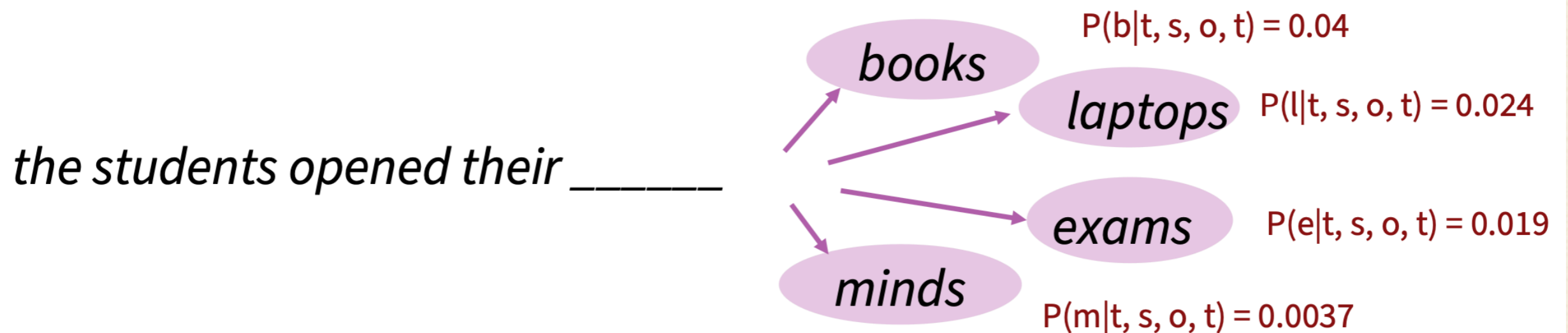
University of Louisiana at Lafayette

What is a language model?

- ❖ A language model is a **probability distribution** over **sequences of words**
- ❖ More formally, given a sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$, it computes the probability distribution of the next word $x^{(t+1)}$:
$$P(x^{(t+1)} | x^{(t)}, \dots, x^{(1)})$$
- ❖ Joint distribution of a sequence by **chain rule**:
$$P(x^{(1)}, \dots, x^{(t)}) = P(x^{(1)})P(x^{(2)} | x^{(1)}) \dots P(x^{(t)} | x^{(t-1)}, \dots, x^{(1)})$$

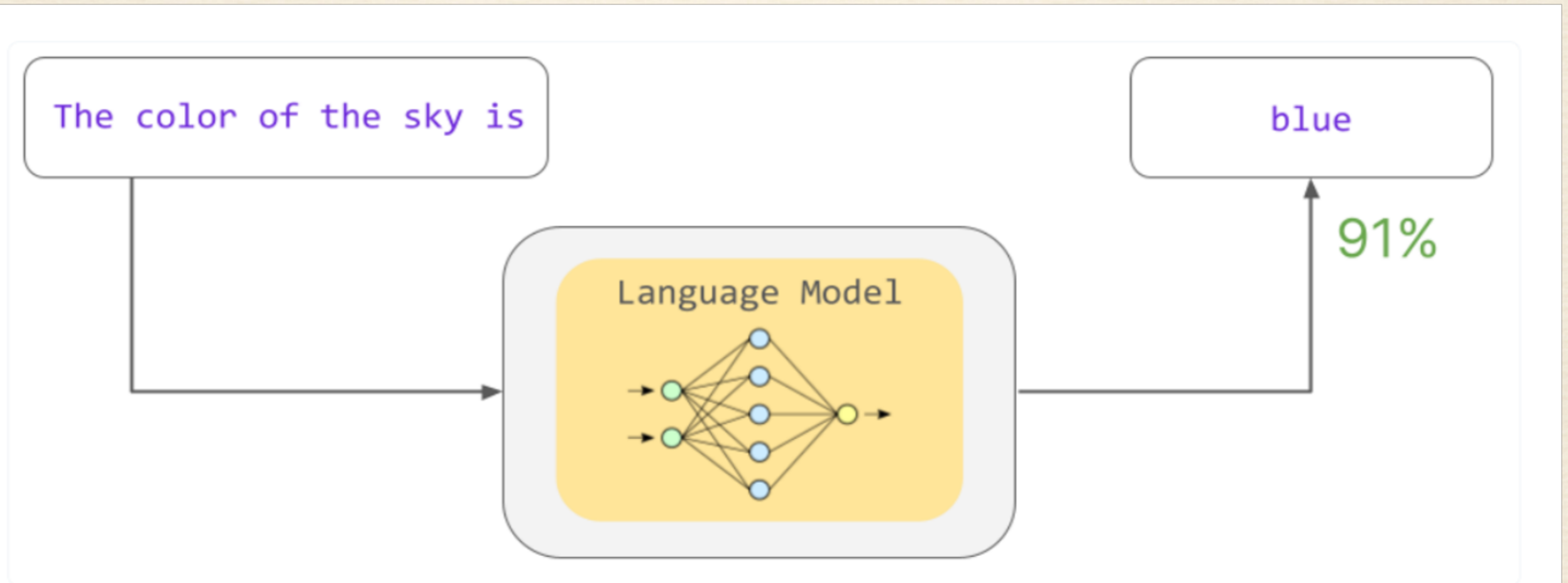
What is a language model?

- ❖ A language model predicts a word in a context:



- ❖ A key part of decoding tasks: *speech recognition, spelling correction*, and language generation tasks: *machine translation, summarization, story generation*

What is a language model?



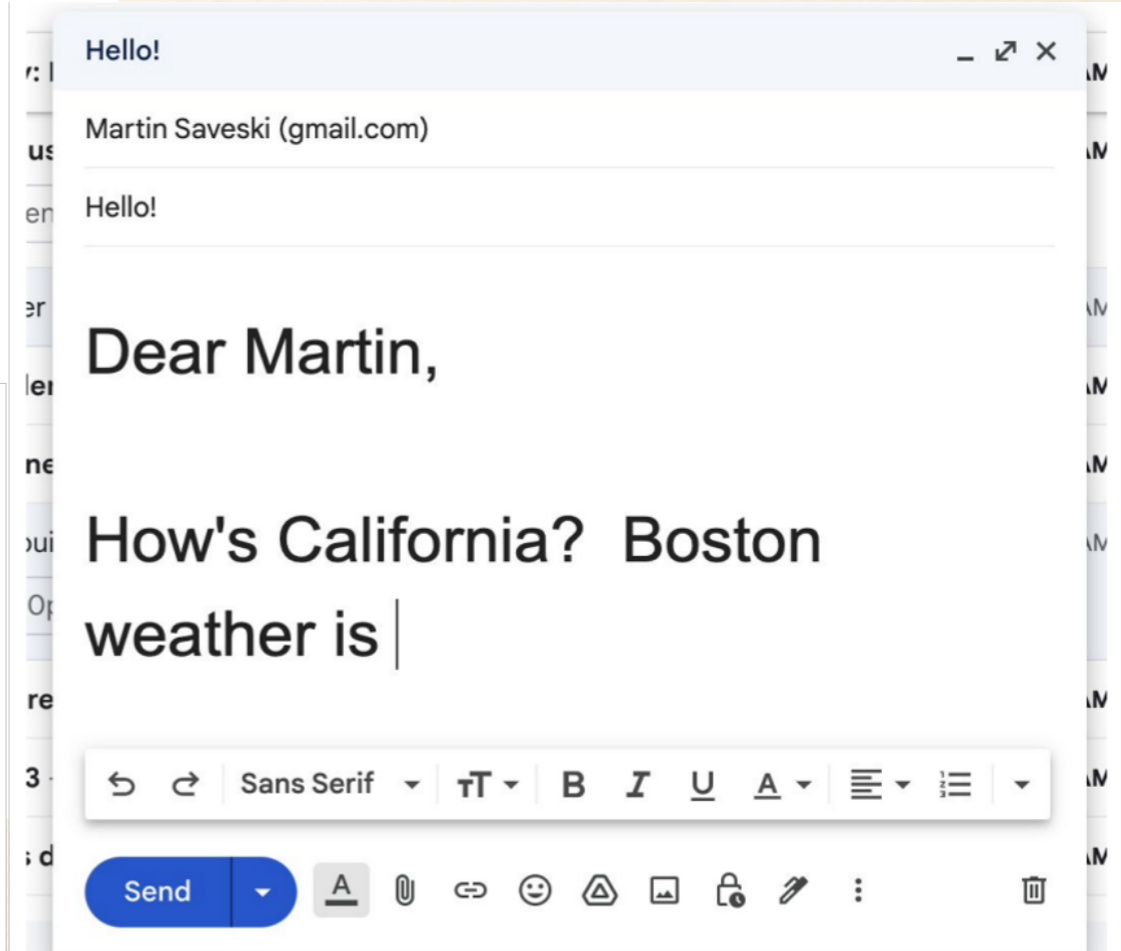
A language model can predict the most probable word (or words) to follow this phrase, based on the statistical patterns it has learned during training. In the figure, a Language Model may estimate a 91% probability that the word *blue* follows the sequence of words *The color of the sky is*.

What is a language model?

w	$P(w \mid \text{Boston weather is})$
cold	0.4
frigid	0.23
terrible	0.12
great	0.02
fantastic	0.01
miserable	0.008
warm	0.005
hot	0.002

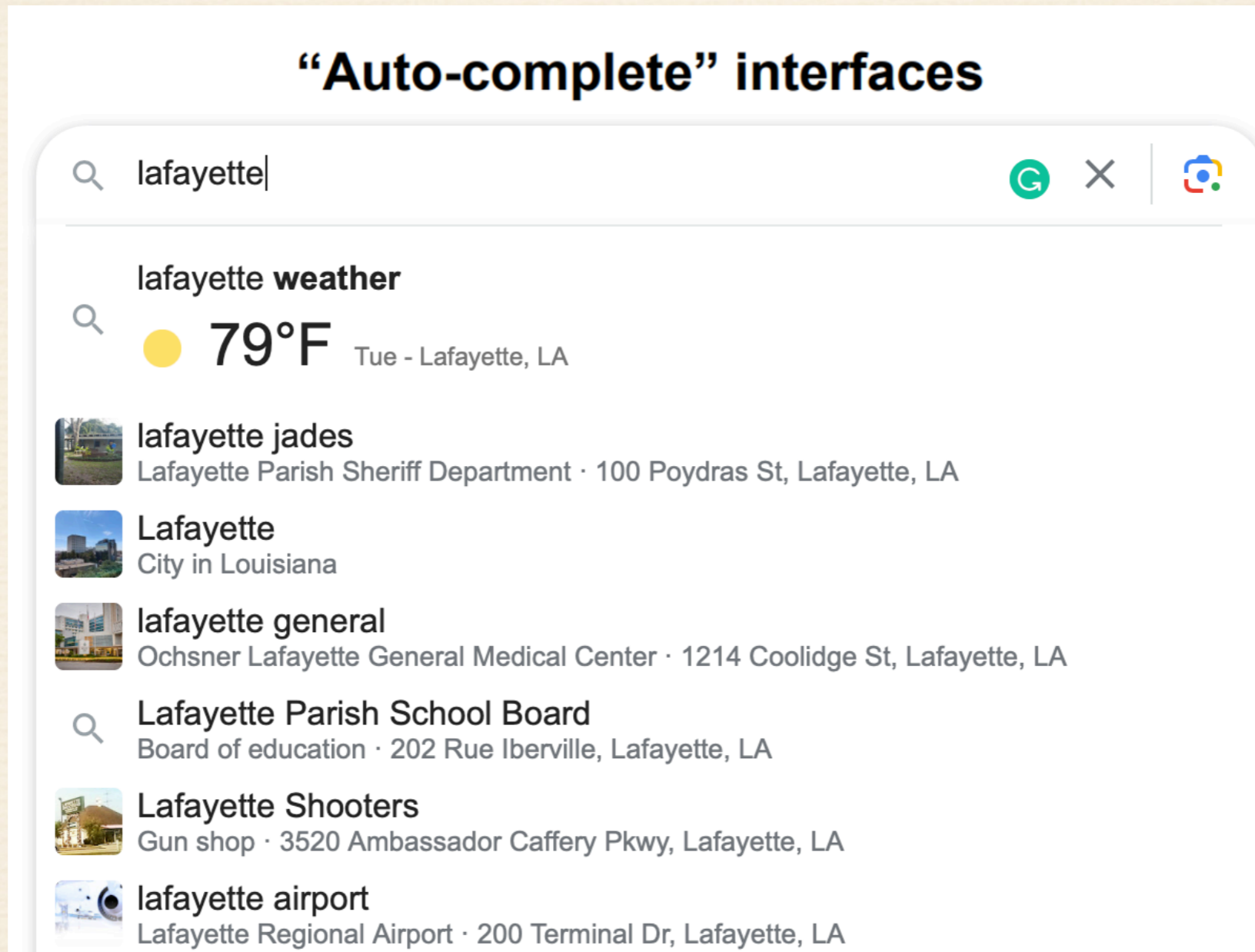
$P(\text{"Boston weather is cold"}) / P(\text{"Boston weather is *"})$

Sentence	Probability
Aardvarks ate apples	0.00000000241
...	...
Boston weather is callous	0.0000000121
Boston weather is cold	0.0000234
Boston weather is cork	0.00000000291
Boston weather is crane	0.00000000185
Boston weather is crazy	0.00000322
Boston weather is furious	0.00000000112
Boston weather is frigid	0.0000321
...	...
Zyzyyx zork zaphod	0.00000000112



What language models can do?

“Auto-complete” interfaces

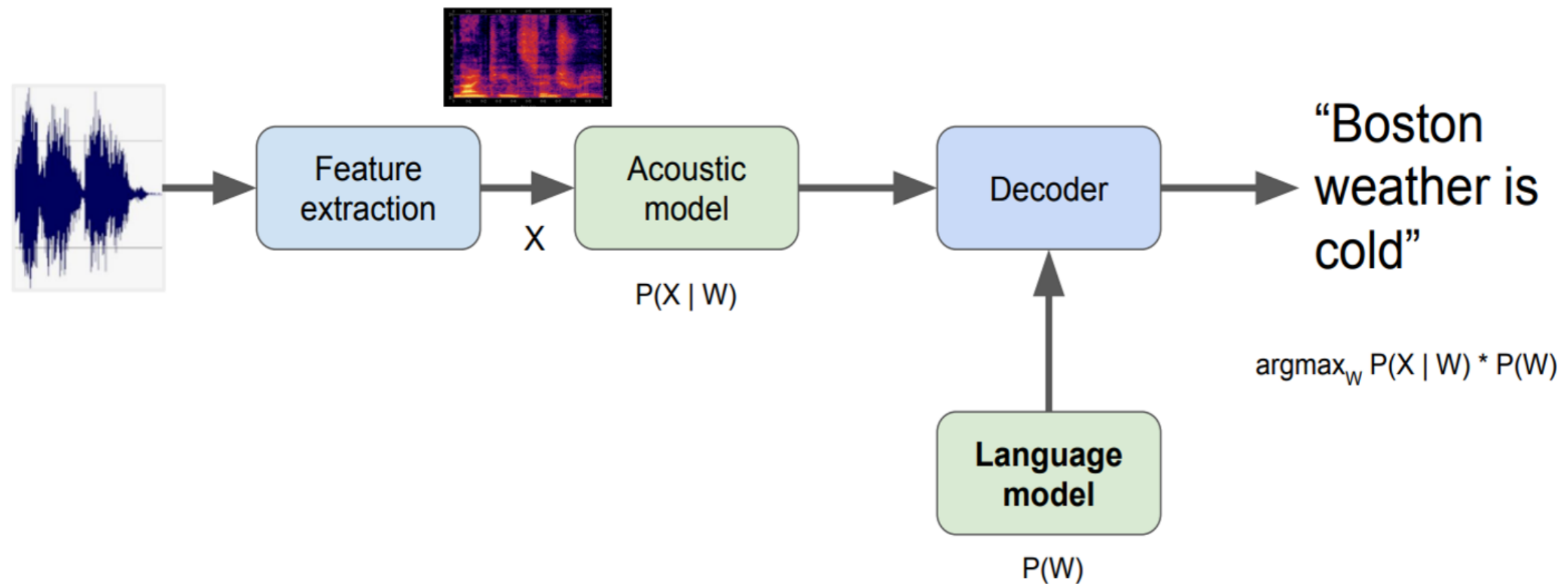


A screenshot of a search interface showing auto-complete suggestions for the query "lafayette". The search bar contains "lafayette" and has icons for search, refresh, close, and image search. Below the search bar, several suggestions are listed, each with a small image icon, a title, and a description.

- lafayette weather**
79°F Tue - Lafayette, LA
- lafayette jades**
Lafayette Parish Sheriff Department · 100 Poydras St, Lafayette, LA
- Lafayette**
City in Louisiana
- lafayette general**
Ochsner Lafayette General Medical Center · 1214 Coolidge St, Lafayette, LA
- Lafayette Parish School Board**
Board of education · 202 Rue Iberville, Lafayette, LA
- Lafayette Shooters**
Gun shop · 3520 Ambassador Caffery Pkwy, Lafayette, LA
- lafayette airport**
Lafayette Regional Airport · 200 Terminal Dr, Lafayette, LA

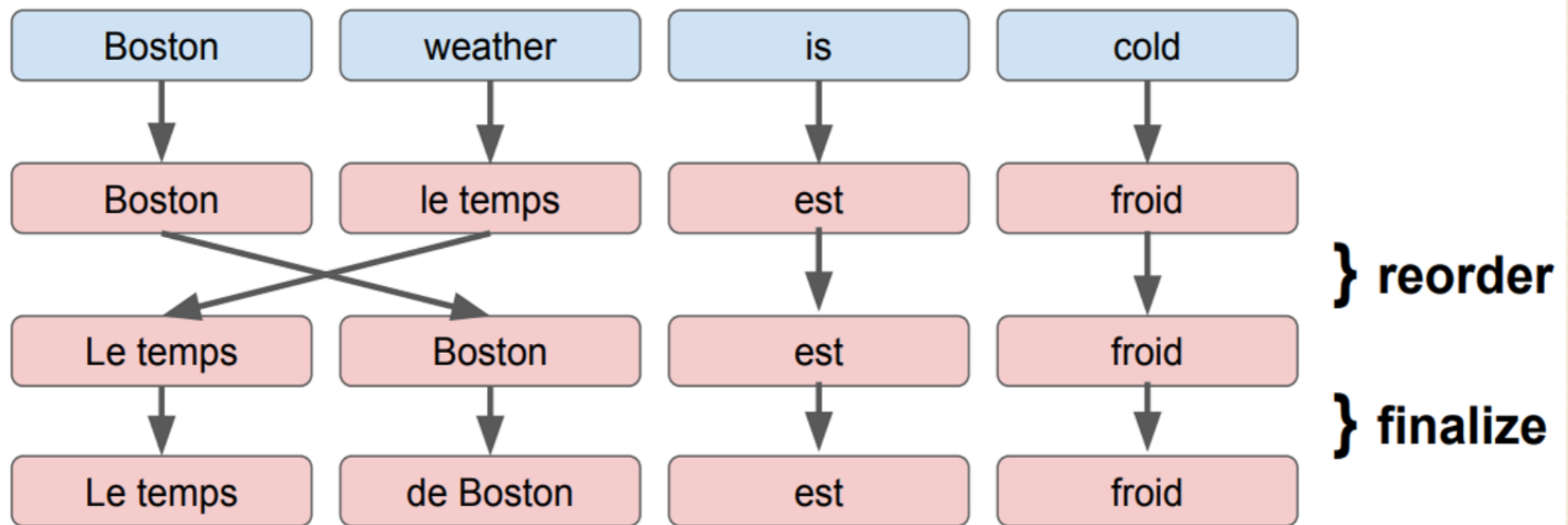
What language models can do?

Speech recognition



What language models can do?

Language Translation



What language models can do?



Q&A

Answer questions based on existing knowledge.



Grammar correction

Corrects sentences into standard English.



Summarize for a 2nd grader

Translates difficult text into simpler concepts.



Natural language to OpenAI API

Create code to call to the OpenAI API using a natural language instruction.



Text to command

Translate text into programmatic commands.



English to other languages

Translates English text into French, Spanish and Japanese.



Natural language to Stripe API

Create code to call the Stripe API using natural language.



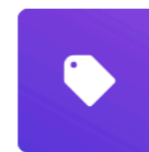
SQL translate

Translate natural language to SQL queries.



Parse unstructured data

Create tables from long form text.



Classification

Classify items into categories via example.



Python to natural language

Explain a piece of Python code in human understandable language.

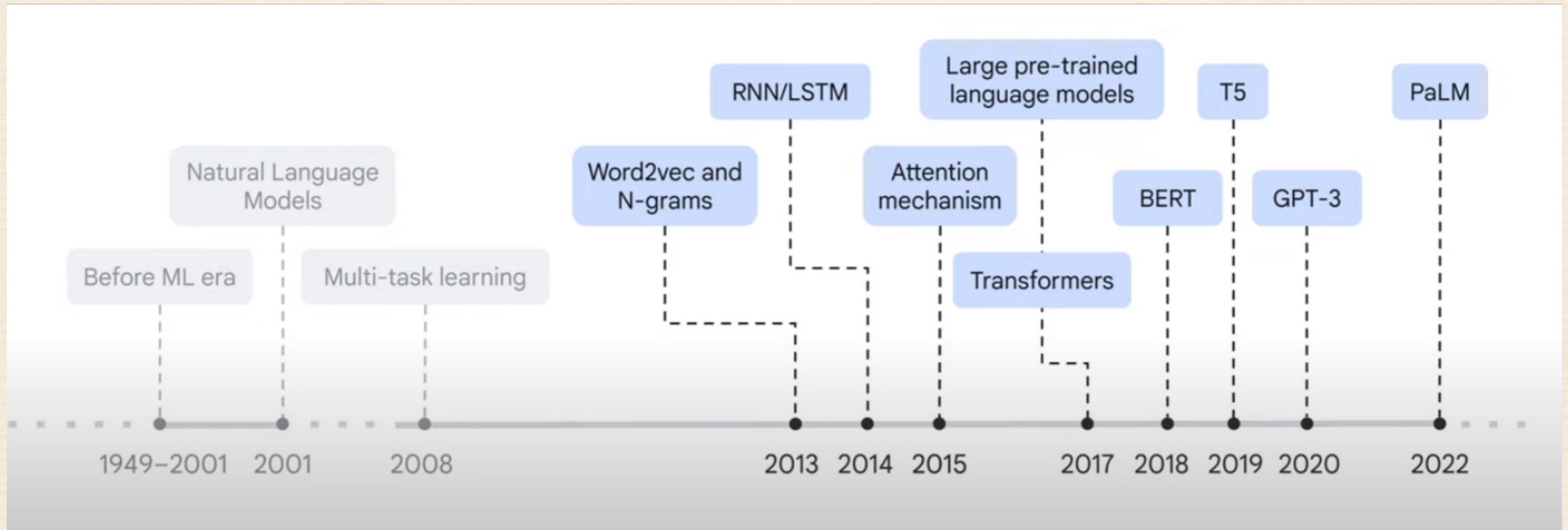


Movie to Emoji

Convert movie titles into emoji.

<https://platform.openai.com/examples>

Language modeling history



N-gram models

- ❖ An n-gram is a sequence of n words in a sentence
- ❖ Use the previous n-1 words in a sequence to predict the next word
- ❖ How? Count how often words follow word sequences in a training corpus; divide to get cond. prob.

The big red **dog**

- ❖ Unigrams: $P(\text{dog})$ Bigrams: $P(\text{dog} | \text{red})$
- ❖ Trigrams: $P(\text{dog} | \text{big red})$ Four-grams: $P(\text{dog} | \text{the big red})$

A Bigram Grammar Fragment

eat on	.16	eat Thai	.03
eat some	.06	eat breakfast	.03
eat lunch	.06	eat in	.02
eat dinner	.05	eat Chinese	.02
eat at	.04	eat Mexican	.02
eat a	.04	eat tomorrow	.01
eat Indian	.04	eat dessert	.007
eat today	.03	eat British	.001

A Bigram Grammar Fragment

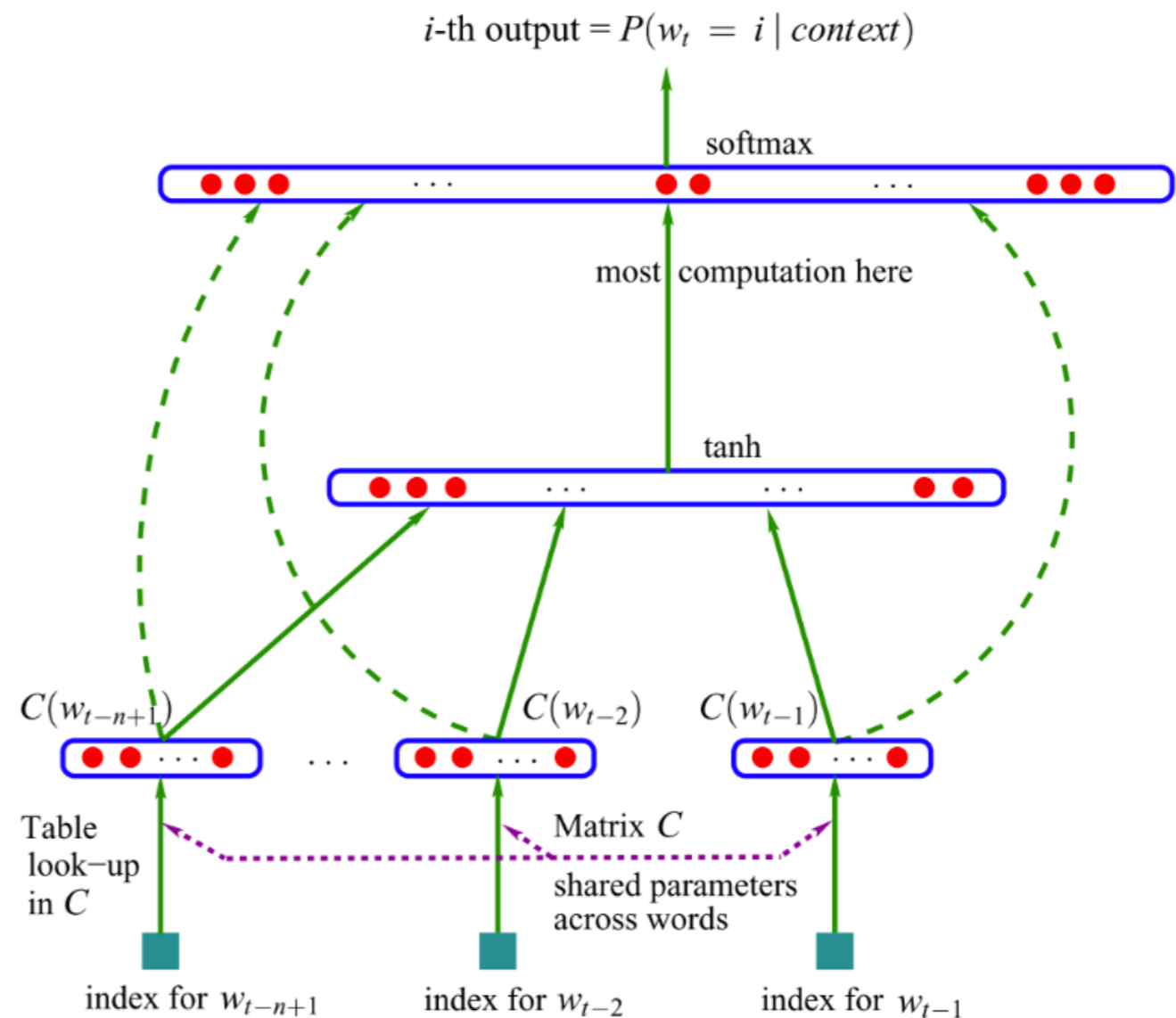
<start> I	.25	want some	.04
<start> I'd	.06	want Thai	.01
<start> Tell	.04	to eat	.26
<start> I'm	.02	to have	.14
I want	.32	to spend	.09
I would	.29	to be	.02
I don't	.08	British food	.60
I have	.04	British restaurant	.15
want to	.65	British cuisine	.01
want a	.05	British lunch	.01

A Bigram Grammar Fragment

- ❖ $P(\text{I want to eat British food}) = P(\text{I} | \langle \text{start} \rangle) P(\text{want} | \text{I}) P(\text{to} | \text{want}) P(\text{eat} | \text{to}) P(\text{British} | \text{eat}) P(\text{food} | \text{British}) = .25 * .32 * .65 * .26 * .001 * .60 = .000080$
- ❖ $P(\text{I want to eat Chinese food}) = .00015$
- ❖ Probabilities seem to capture “syntactic” facts, “world knowledge”
 - ❖ *eat* is often followed by a noun phrase
 - ❖ British food is not too popular
- ❖ **Limitations:** curse of dimensionality: zillions of parameters; limiting context

Neural language models

- ❖ A feed-forward neural network was proposed in 2003 by Bengio et al.
- ❖ Solve the curse of dimensionality: dense, low-dimensionality real-number word vectors



Recurrent NN-based LMs

- Recurrent neural networks RNNs (Mikolov et al., 2010) can memorize the previous outputs when receiving the next inputs

output distribution

$$\hat{y}^{(t)} = \text{softmax}(U\mathbf{h}^{(t)} + \mathbf{b}_2) \in \mathbb{R}^{|V|}$$

hidden states

$$\mathbf{h}^{(t)} = \sigma(\mathbf{W}_h\mathbf{h}^{(t-1)} + \mathbf{W}_e\mathbf{e}^{(t)} + \mathbf{b}_1)$$

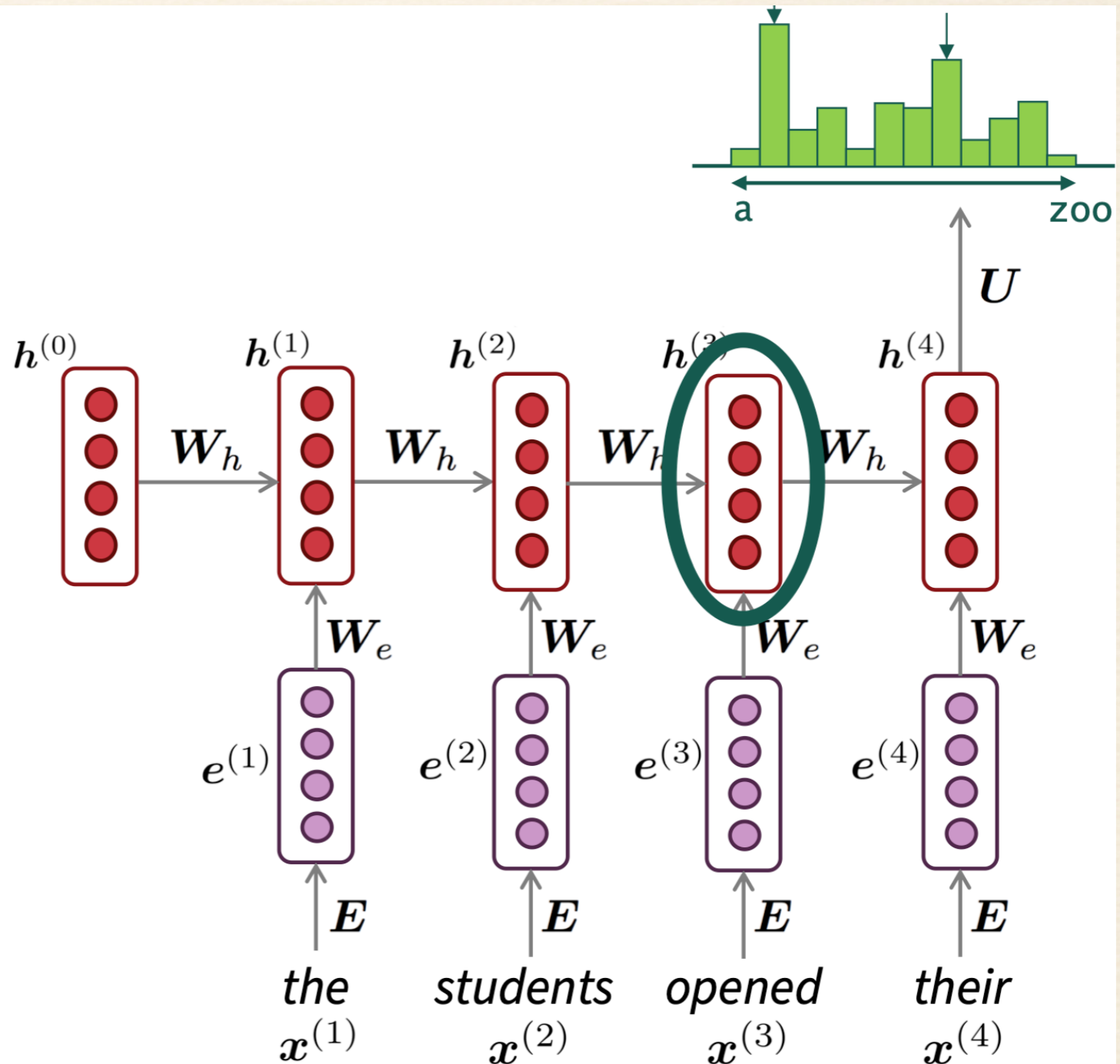
$\mathbf{h}^{(0)}$ is the initial hidden state

word embeddings

$$\mathbf{e}^{(t)} = \mathbf{E}\mathbf{x}^{(t)}$$

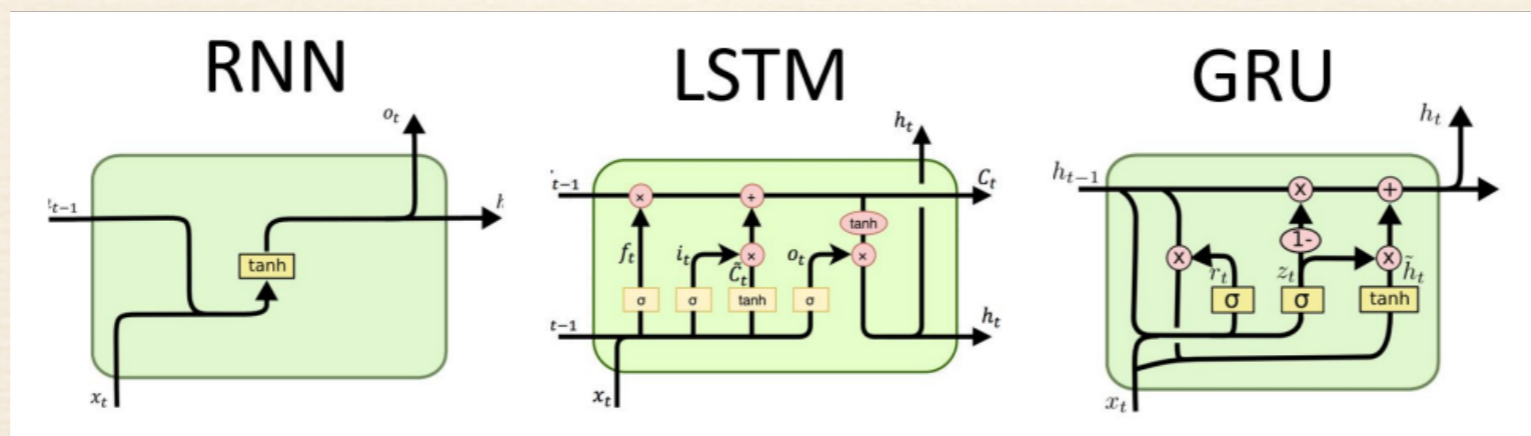
words / one-hot vectors

$$\mathbf{x}^{(t)} \in \mathbb{R}^{|V|}$$



Long Short-Term Memory

- ❖ Recurrent Neural Networks suffer from short-term memory
 - ❖ vanishing gradient problem: the gradient shrinks as it back propagates through time
- ❖ If a sequence is long enough, it'll have a hard time carrying information from earlier time steps to later ones
- ❖ Solution: LSTMs [Zaremba et al. 2014] and Gated Recurrent Units (GRUs)
 - ❖ they have internal mechanisms called gates to regulate the flow of information
 - ❖ these gates can learn which data in a sequence is important to keep or throw away
- ❖ GRU is quicker to compute and has fewer parameters than LSTM



Large language models

- ❖ Large, general-purpose language models, can be pre-trained and then fine-tuned for specific purposes

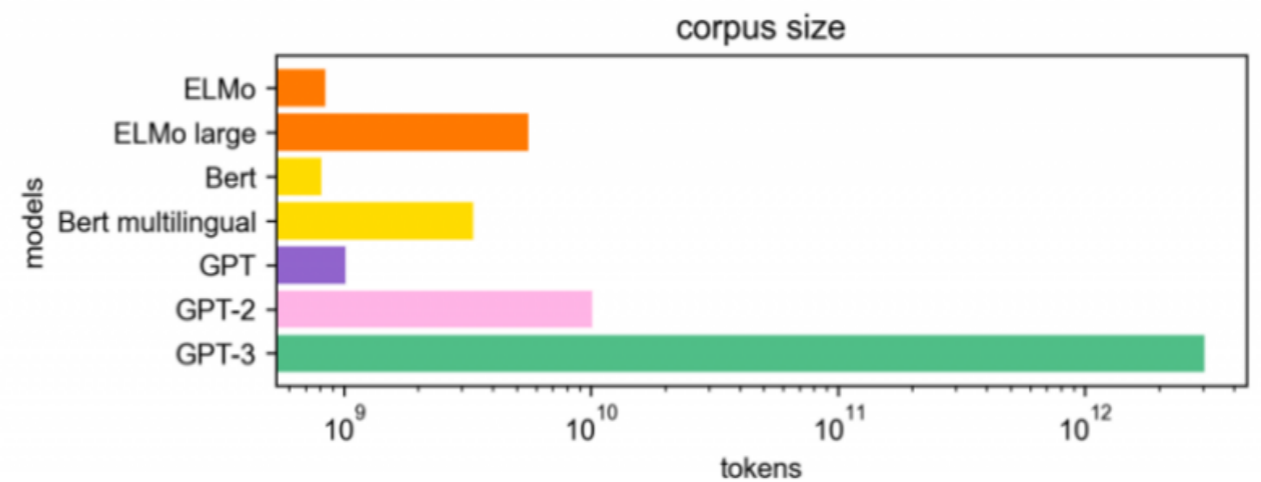
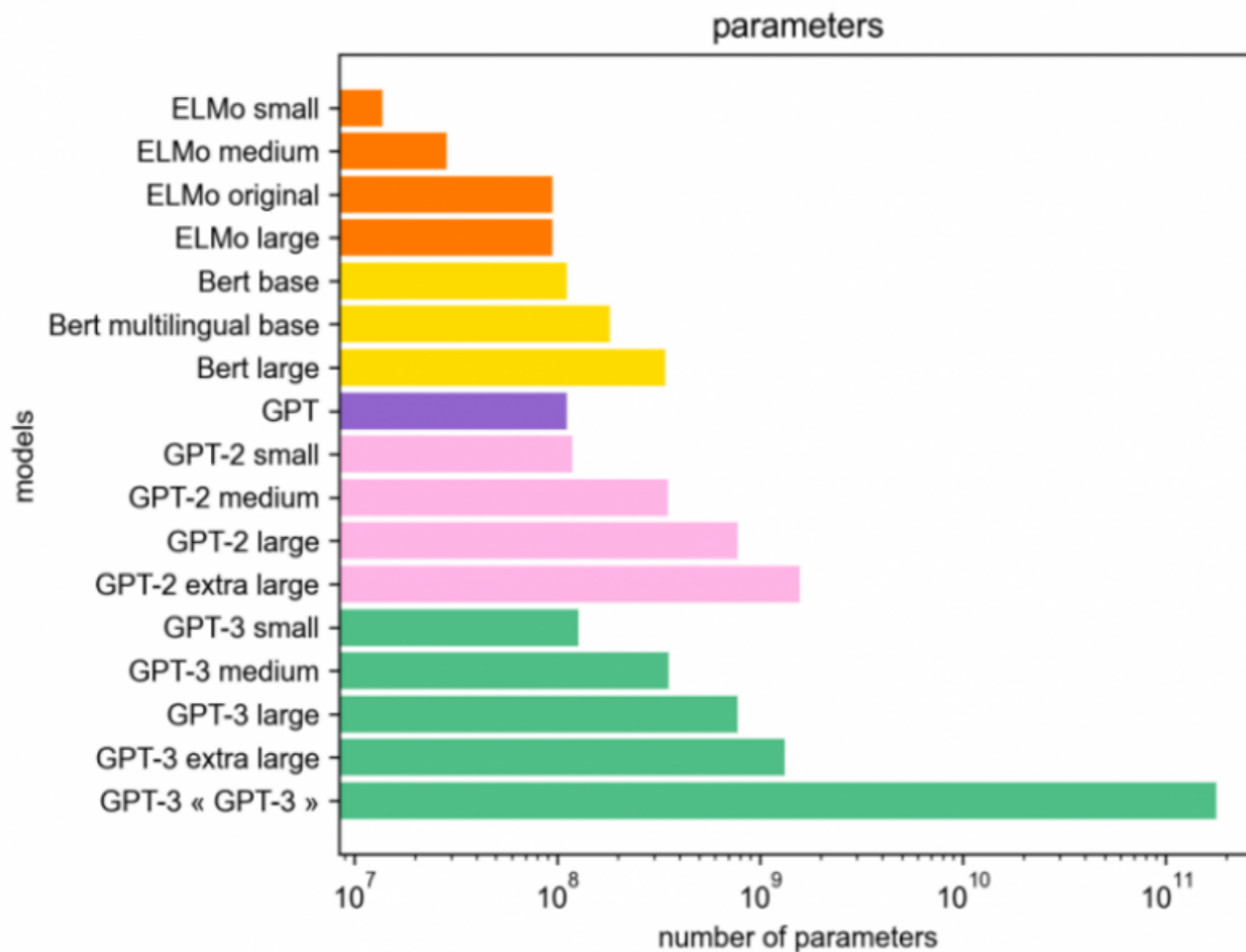
LLMs are a type of deep learning model designed to process and understand natural language data

They are built on neural network architectures, particularly the transformer architecture

How large?

❖ Large number of parameters

Large training dataset

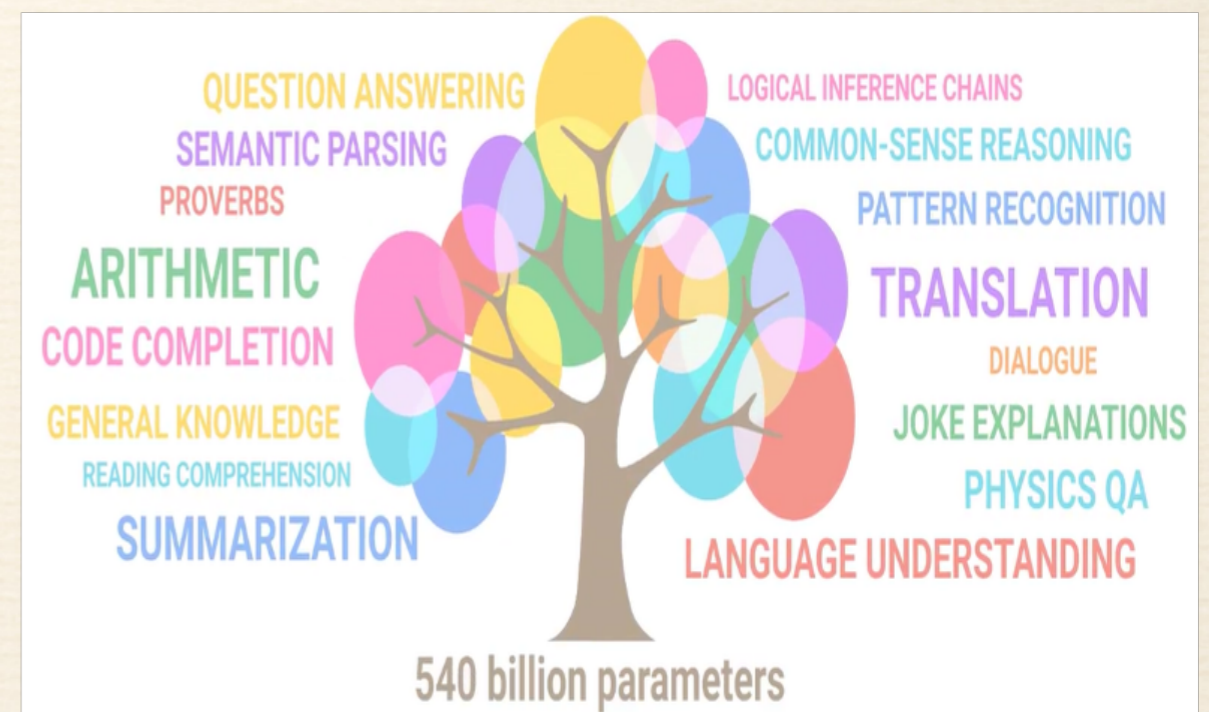
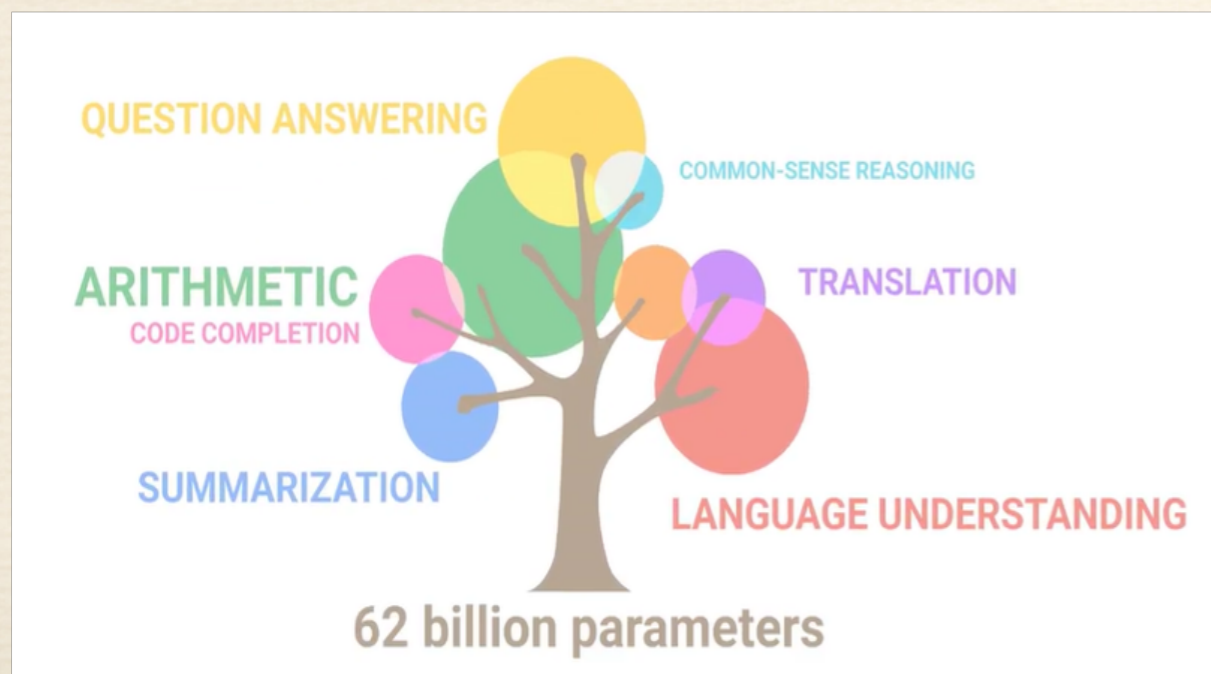


More recent models: PaLM (540B), OPT (175B), BLOOM (176B)...

Image source: <https://hellofuture.orange.com/en/the-gpt-3-language-model-revolution-or-evolution/>

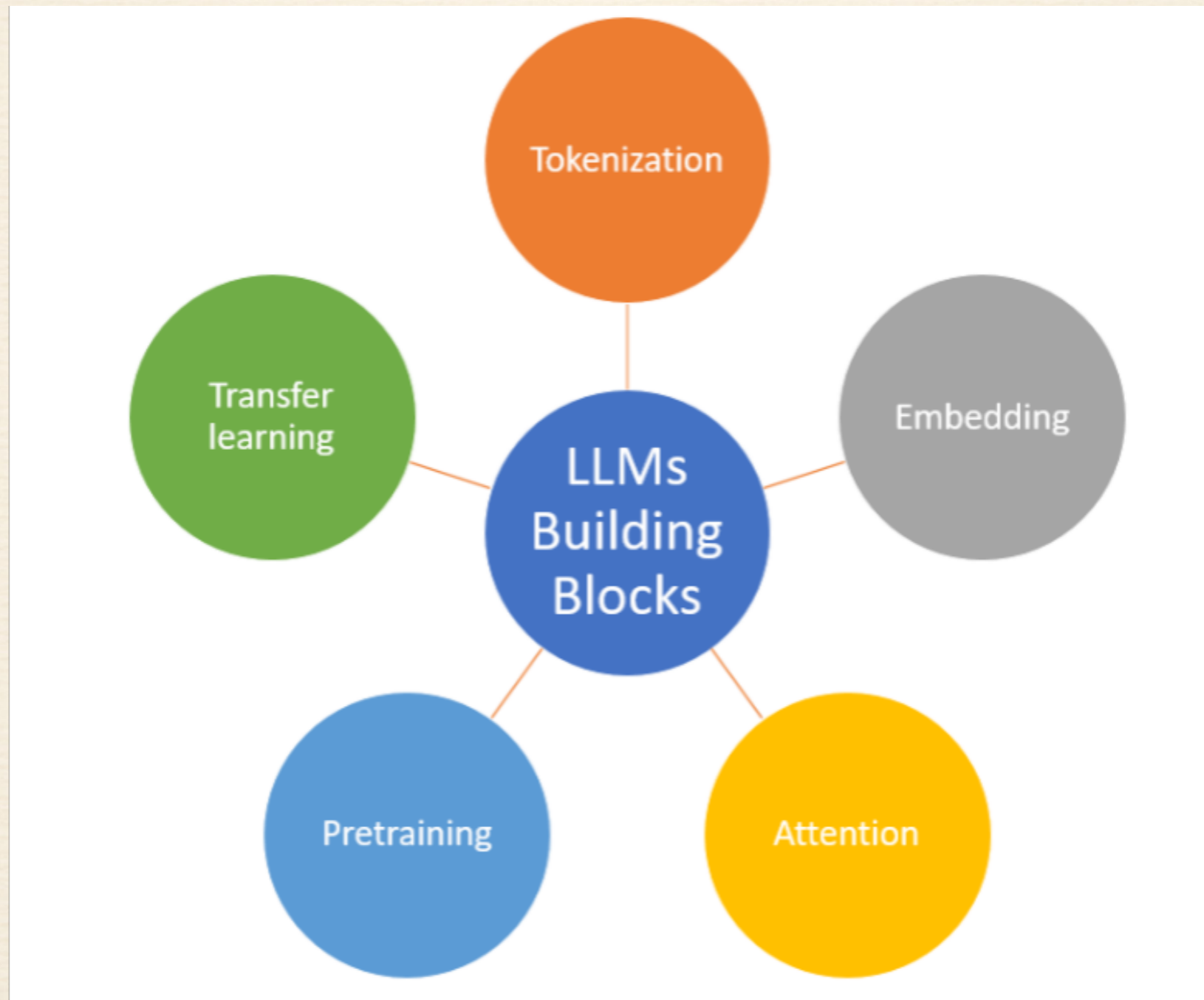
How promising?

- ❖ A single model can be used for different tasks
- ❖ The fine-tune process requires minimal field data
- ❖ The performance is continuously growing with more data and parameters



What's the secret sauce?

- ❖ LLMs are composed of several key building blocks that enable them to efficiently process and understand natural language data



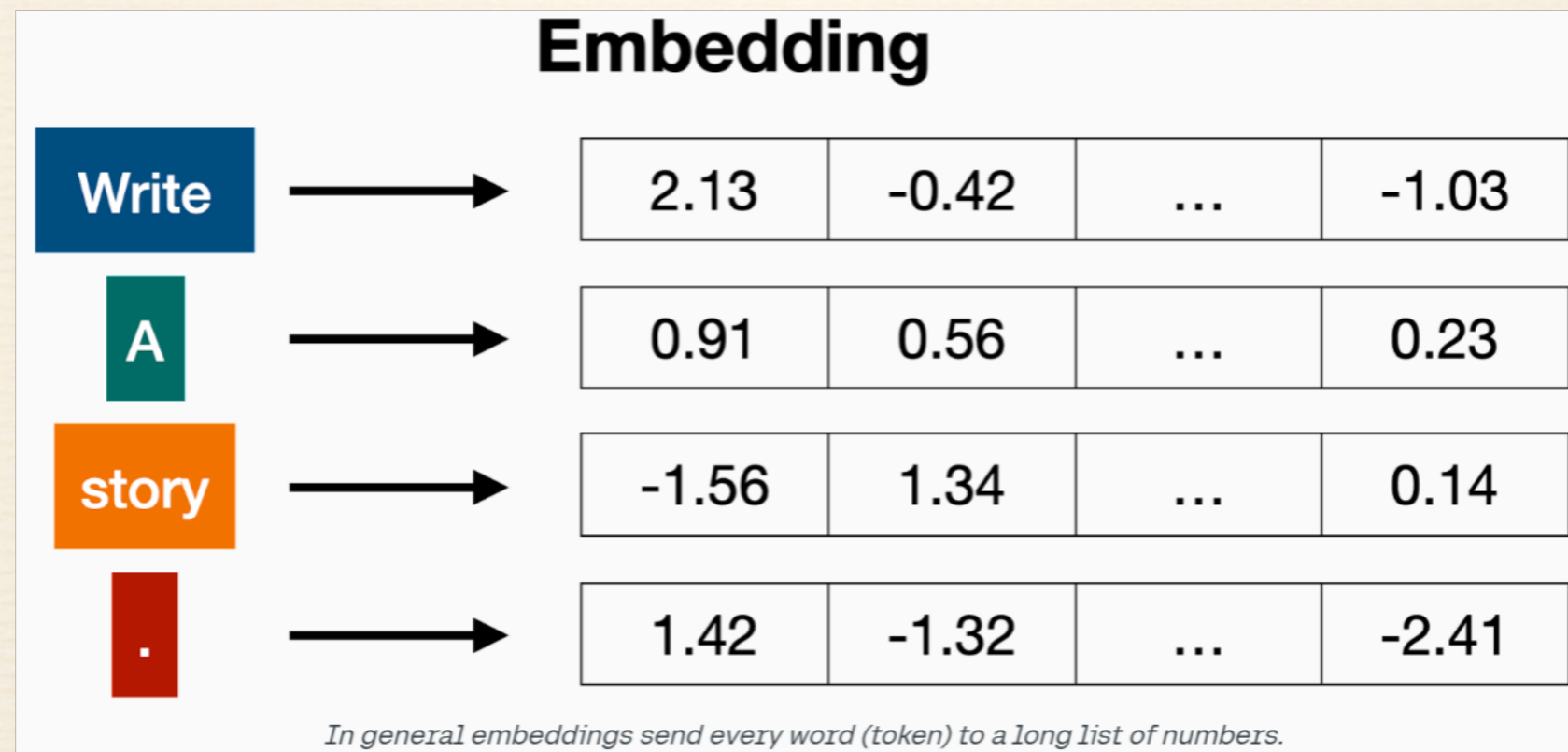
Key components: Tokenization

- ❖ In order to get our computer to understand any text, we need to break that word down in a way that our machine can understand
- ❖ Tokenization is a way of separating a piece of text into smaller units called tokens
- ❖ Tokens can be either words, characters, or subwords



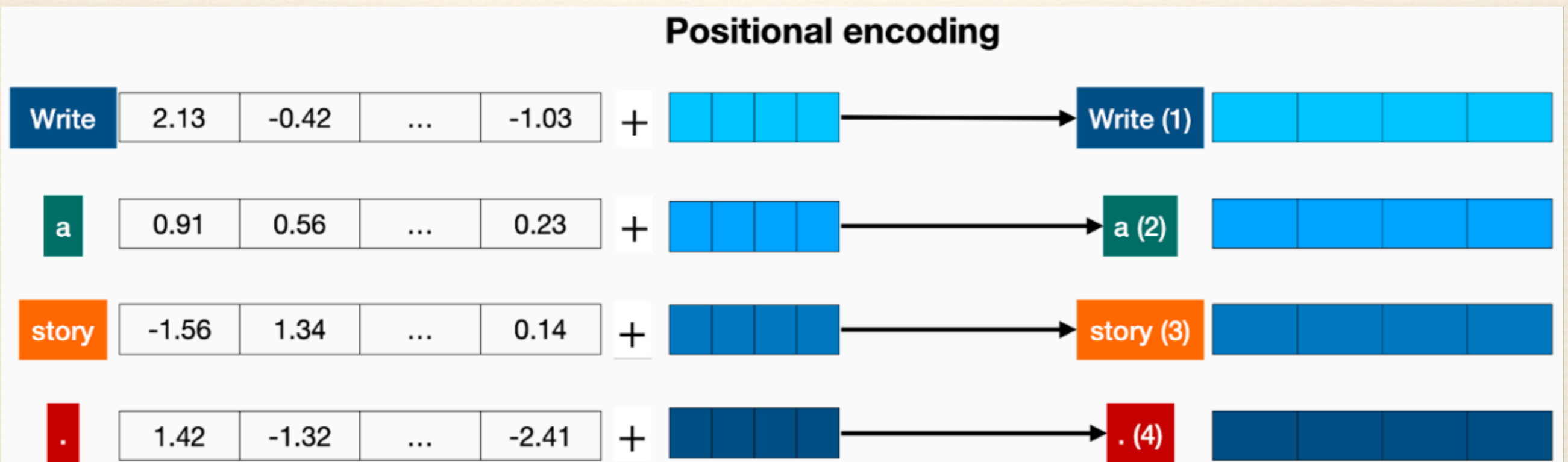
Key components: Embedding

- ❖ Machine learning or deep learning models cannot process text, so we need to figure out a way to convert these textual data into numerical data
- ❖ Every piece of text turns into a vector (a list) of numbers



Key components: Positional encoding

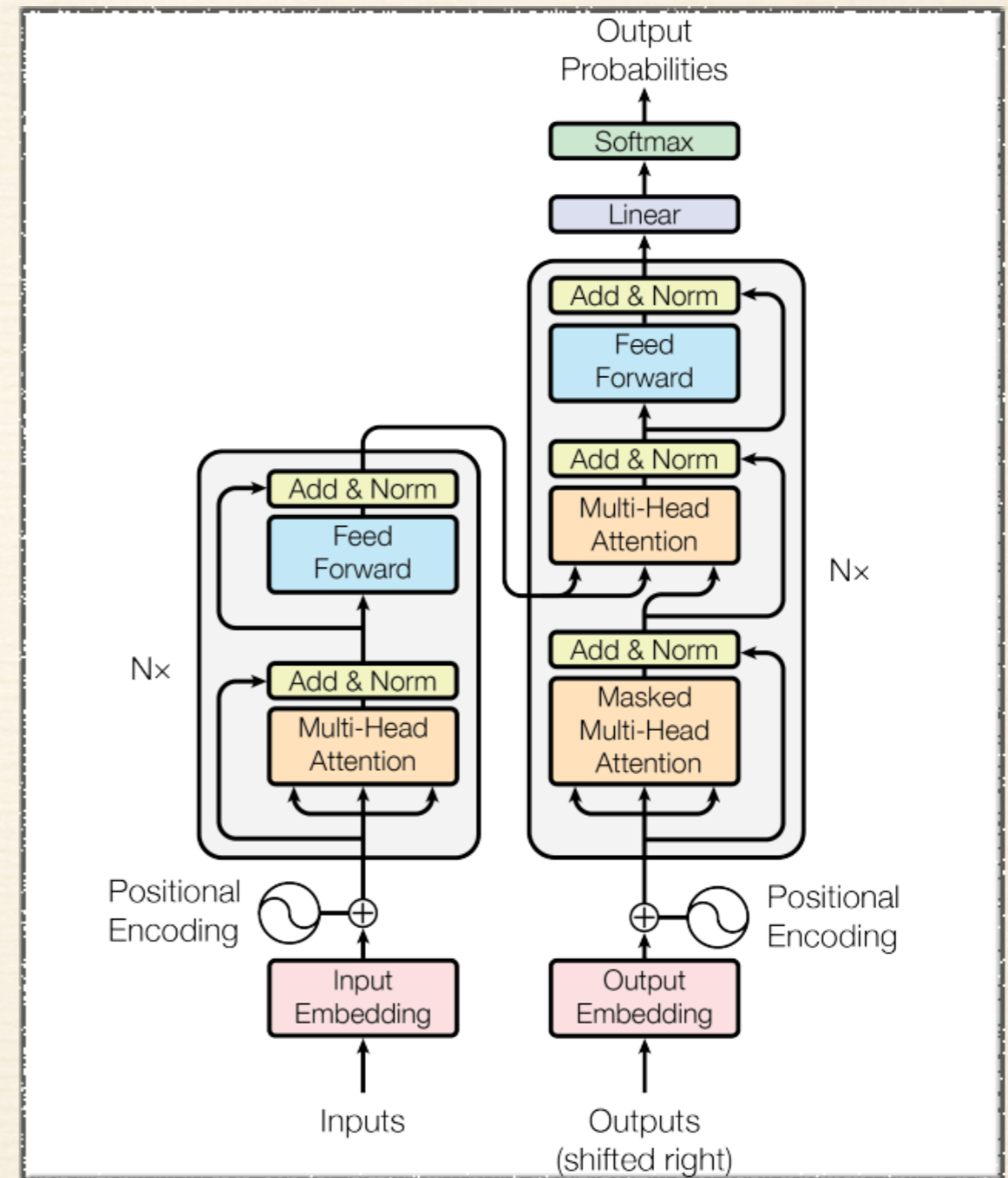
- ❖ The vectors corresponding to the words “Write”, “a”, “story”, and “.” become the modified vectors that carry information about their position, labeled “Write (1)”, “a (2)”, “story (3)”, and “.(4)”.



Positional encoding adds a positional vector to each word, in order to keep track of the positions of the words.

Transformer architecture

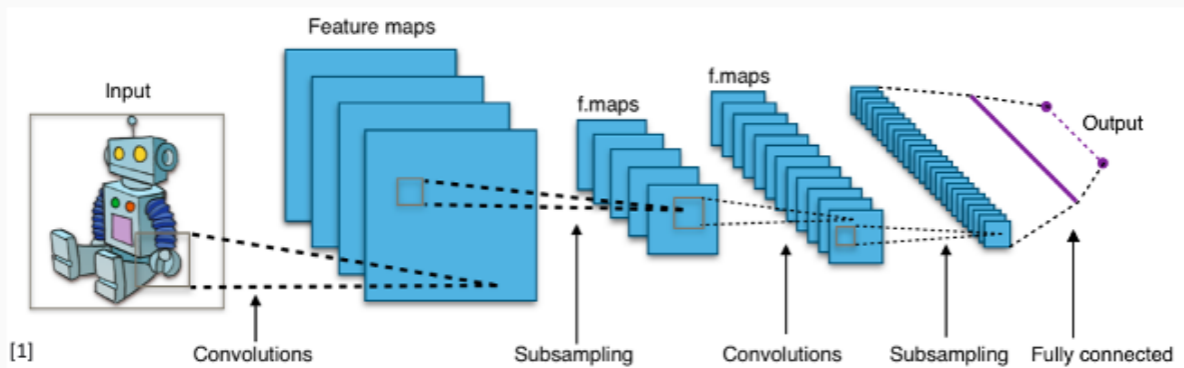
- ❖ Introduced in the paper “Attention Is All You Need” by Vaswani et al. in 2017
- ❖ Ground-breaking architecture that set SOTA on translation and later all other NLP tasks
- ❖ The key component is the **self-attention** mechanism, which enables the model to attend to different parts of the input sequence to compute representation for each position



Before ~2020

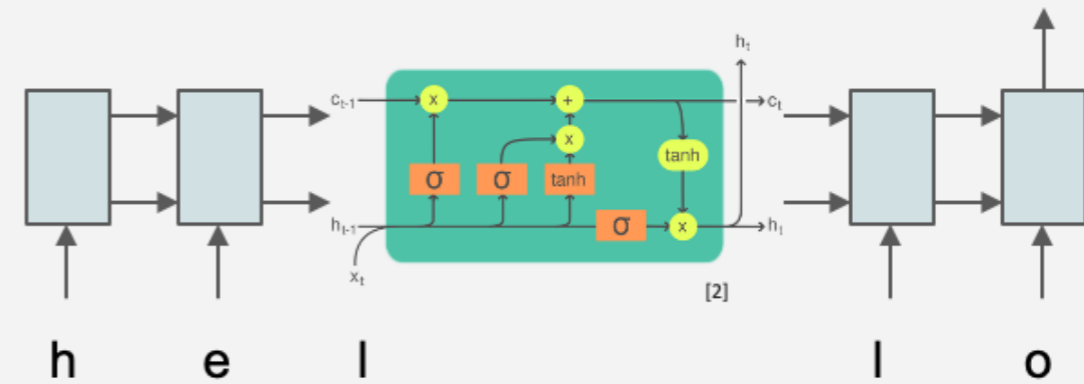
Computer Vision

Convolutional NNs (+ResNets)



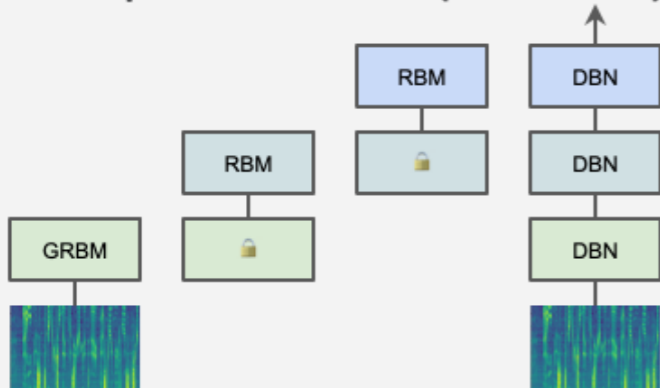
Natural Lang. Proc.

Recurrent NNs (+LSTMs)



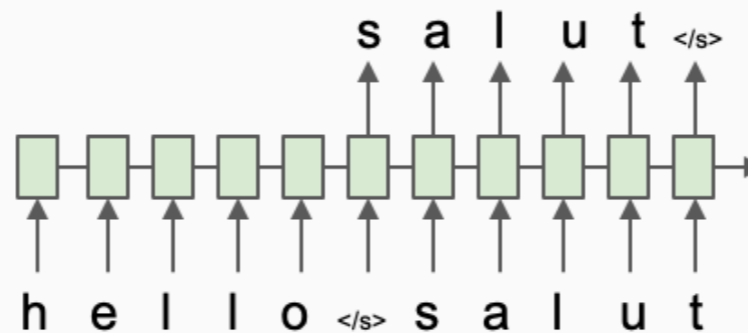
Speech

Deep Belief Nets (+non-DL)



Translation

Seq2Seq



RL

BC/GAIL

Algorithm 1 Generative adversarial imitation learning

- 1: **Input:** Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters θ_0, w_0
- 2: **for** $i = 0, 1, 2, \dots$ **do**
- 3: Sample trajectories $\tau_i \sim \pi_{\theta_i}$
- 4: Update the discriminator parameters from w_i to w_{i+1} with the gradient

$$\hat{\mathbb{E}}_{\tau_i}[\nabla_w \log(D_w(s, a))] + \hat{\mathbb{E}}_{\tau_E}[\nabla_w \log(1 - D_w(s, a))] \quad (17)$$
- 5: Take a policy step from θ_i to θ_{i+1} , using the TRPO rule with cost function $\log(D_{w_{i+1}}(s, a))$. Specifically, take a KL-constrained natural gradient step with

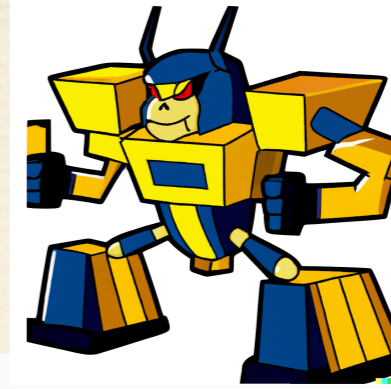
$$\hat{\mathbb{E}}_{\tau_i}[\nabla_{\theta} \log \pi_{\theta}(a|s)Q(s, a)] - \lambda \nabla_{\theta} H(\pi_{\theta}), \quad (18)$$
 where $Q(\bar{s}, \bar{a}) = \hat{\mathbb{E}}_{\tau_i}[\log(D_{w_{i+1}}(s, a)) | s_0 = \bar{s}, a_0 = \bar{a}]$
- 6: **end for**

[1] CNN image CC-BY-SA by Aphex34 for Wikipedia https://commons.wikimedia.org/wiki/File:Typical_cnn.png

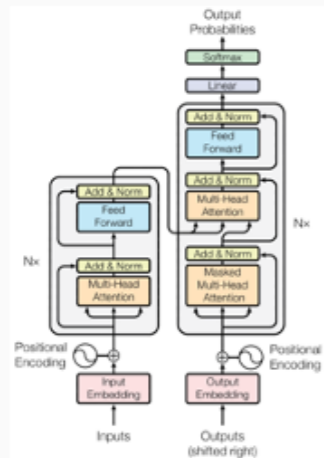
[2] RNN image CC-BY-SA by GChe for Wikipedia https://commons.wikimedia.org/wiki/File:The_LSTM_Cell.svg

<http://lucasb.eyer.be/transformer>

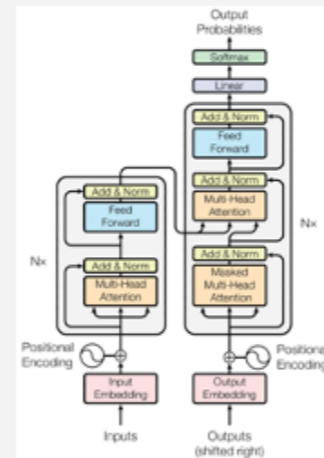
Now



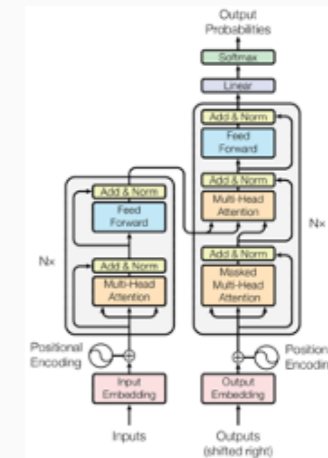
Computer Vision



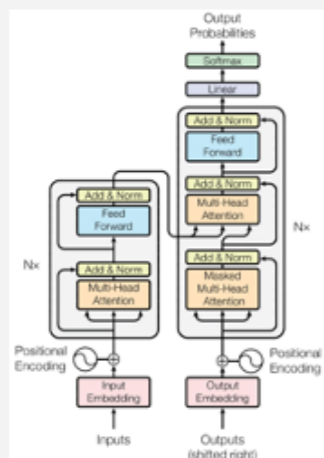
Natural Lang. Proc.



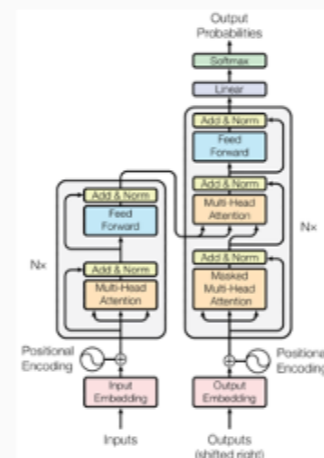
Reinf. Learning



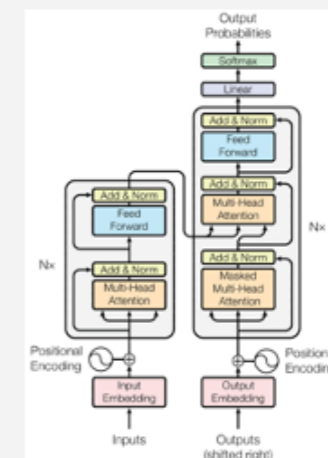
Speech



Translation



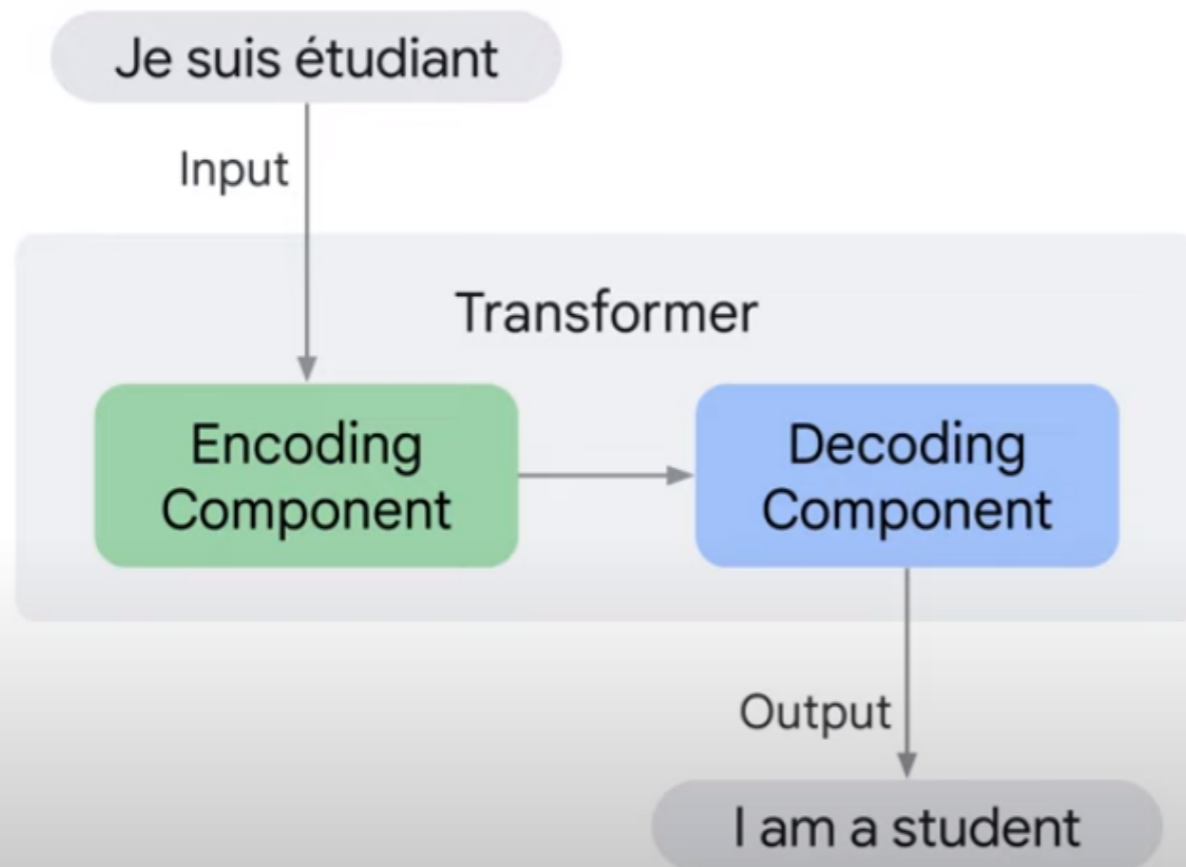
Graphs/Science



<http://lucasb.eyer.be/transformer>

Transformer

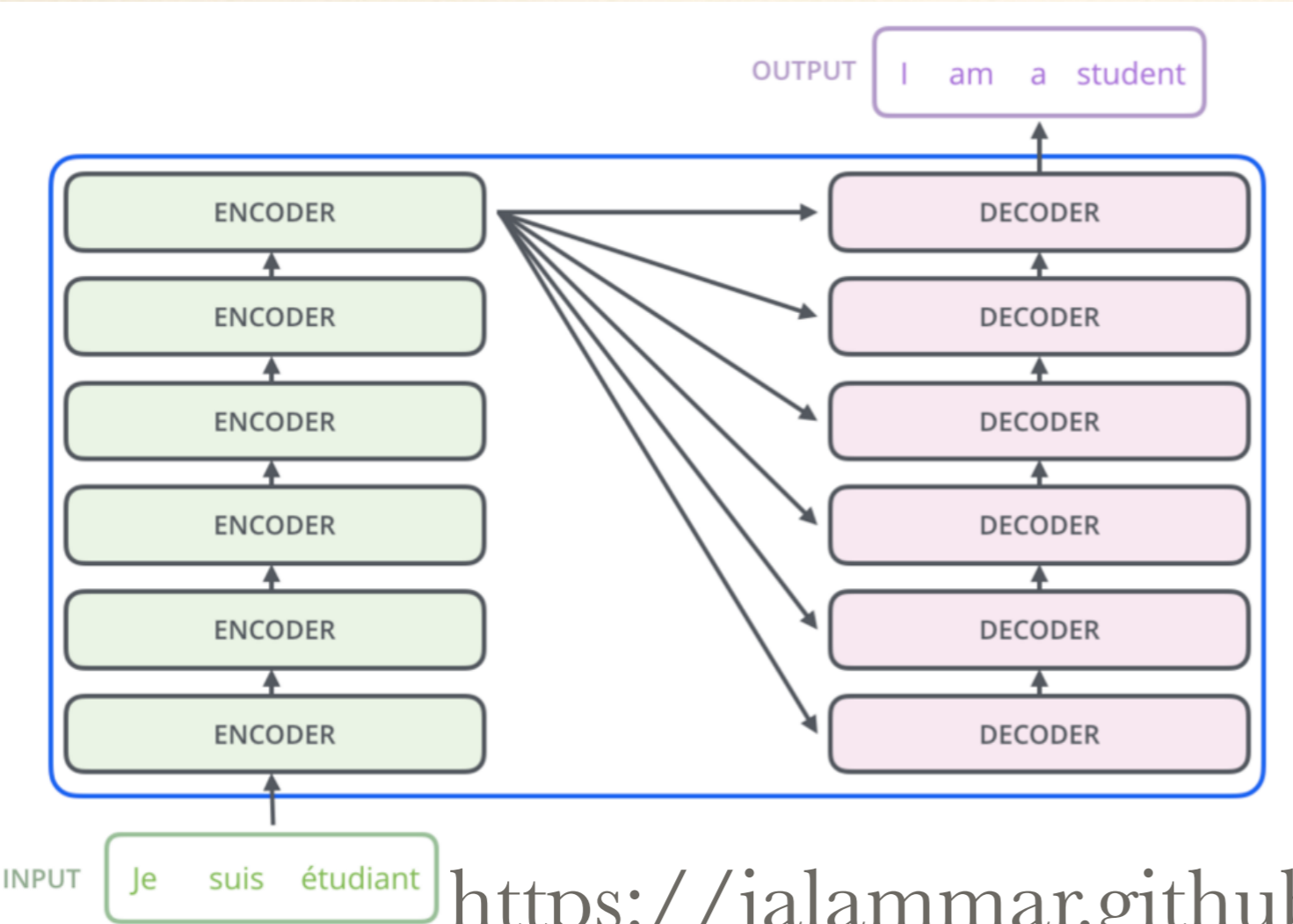
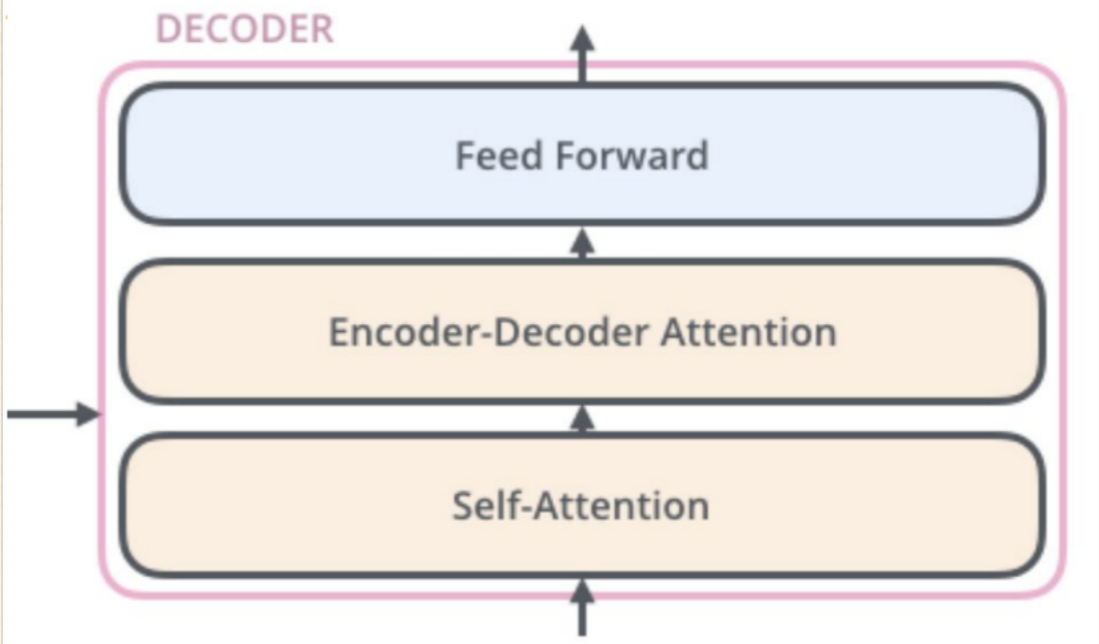
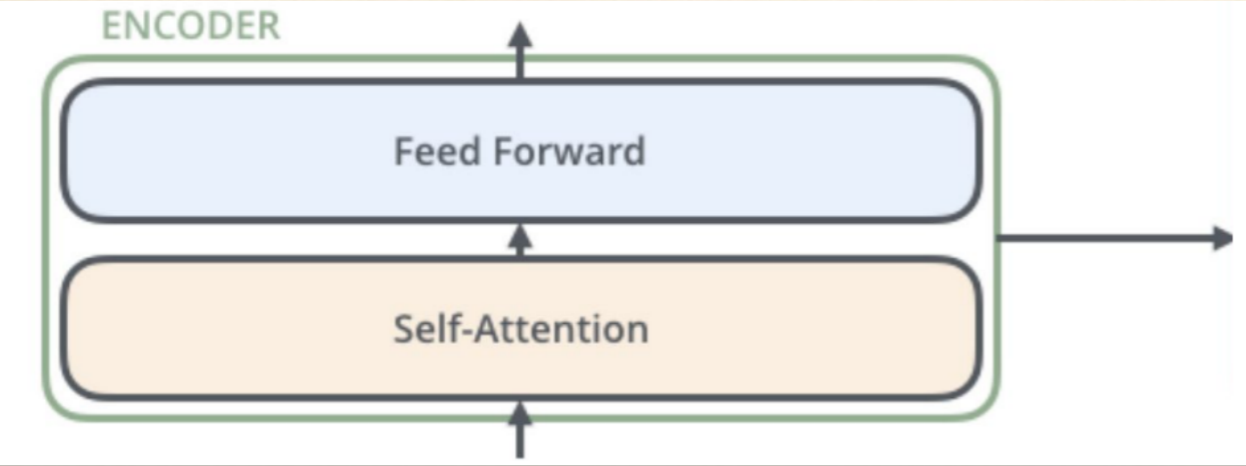
- ❖ A transformer is an encoder-decoder model that uses the attention mechanism
- ❖ Encoder encodes the input sequence and passes it to the decoder
- ❖ The decoder decodes a representation for a relevant task



Massive advantage over RNN based encoder-decoder architecture since it allows us to:

- Take advantage of parallelization GPU/TPU.
- Process much more data in the same amount of time.
- **Process all tokens at once!**

Transformer



<https://jalammarmar.github.io/illustrated-transformer>

Inner workings of the Encoder

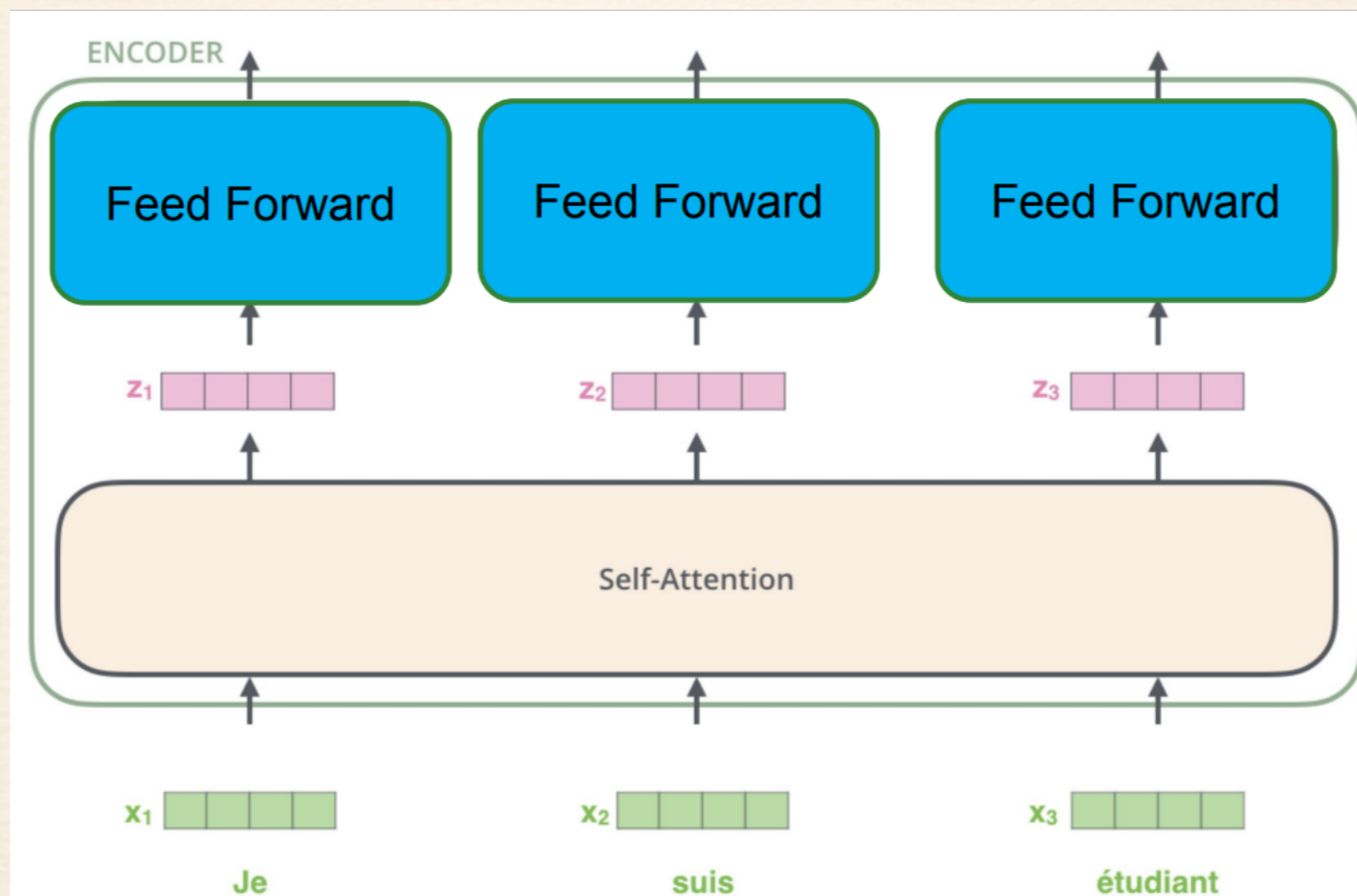
- ❖ Begin by taking a word embedding for each input word



- ❖ The embedding only enters in the bottom-most encoder
- ❖ Each encoder receives a list of vectors
- ❖ In the bottom encoder these are the word embeddings
- ❖ But in other encoders, it is the output of the encoder that's directly below
- ❖ The size of this list is a hyperparameter we can set – you can think of this as the length of the longest sentence in our training dataset

Inner workings of the Encoder

- ❖ The word in each position flows through its own path in the encoder
- ❖ There are dependencies between these paths in the self-attention layer



Key components: Attention

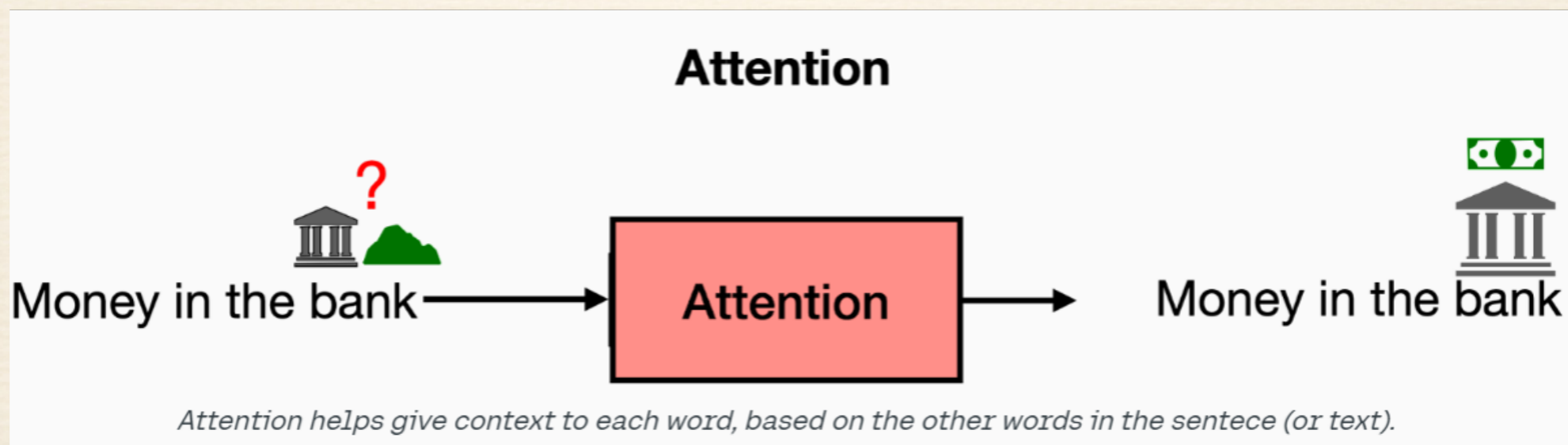
- ❖ The same word can be used with different meanings
- ❖ Attention is a very useful technique that help LMs understand the context. Consider the following two sentences:
 - ❖ Sentence 1: *The bank of the river*
 - ❖ Sentence 2: *Money in the bank*
- ❖ The word '*bank*' appears in both, but with different definitions.
- ❖ In sentence 1, we are referring to the land at the side of the river, and in the second one to the institution that holds money.
- ❖ The computer has no idea of this, so we need to somehow inject that knowledge into it

Key components: Attention

- ❖ What can help us? Well, it seems that the other words in the sentence can come to our rescue.
- ❖ For the first sentence, the words '*the*', and '*of*' do us no good.
- ❖ But the word '*river*' is the one that is letting us know that we're talking about the land at the side of the river.
- ❖ Similarly, in sentence 2, the word '*money*' is the one that is helping us understand that the word '*bank*' is now referring to the institution that holds money.

Key components: Attention

- ❖ Attention moves the words in a sentence closer in the word embeddings
- ❖ The word “*bank*” in the sentence “*Money in the bank*” will be moved closer to the word “*money*”.
- ❖ Equivalently, in the sentence “*The bank of the river*”, the word “*bank*” will be moved closer to the word “*river*”.
- ❖ That way, the modified word “*bank*” vector in each of the two sentences will carry some of the information of the neighboring words, adding context to it.

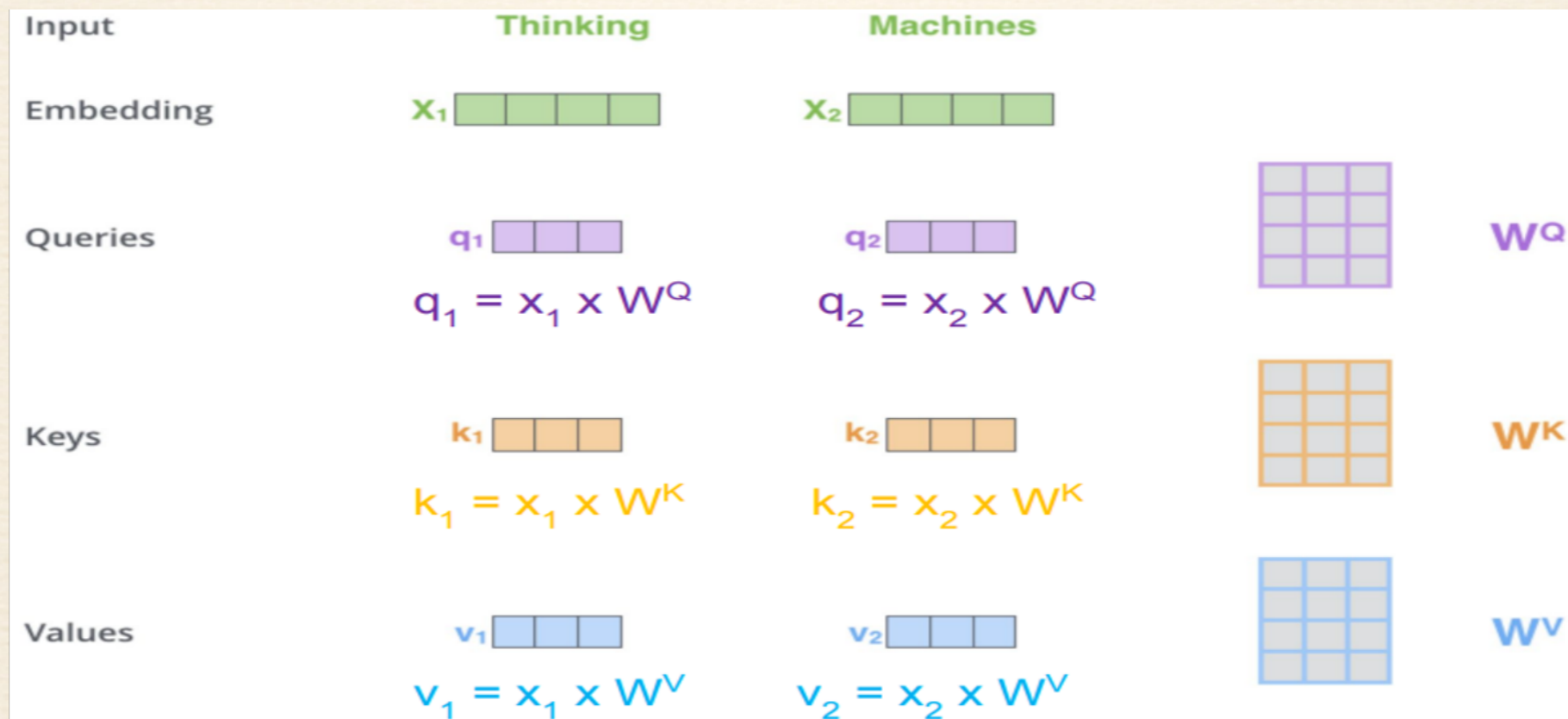


Key components: Attention

- ❖ The attention step used in transformer models is actually much more powerful
- ❖ Self attention: allows the model to look at other words in the input sequence to obtain a better **contextualized encoding** for each word
 - ❖ This essentially bakes in the “understanding” of other relevant words into the one we’re currently processing
- ❖ Multi-head attention: several different embeddings are used to modify the vectors and add context to them

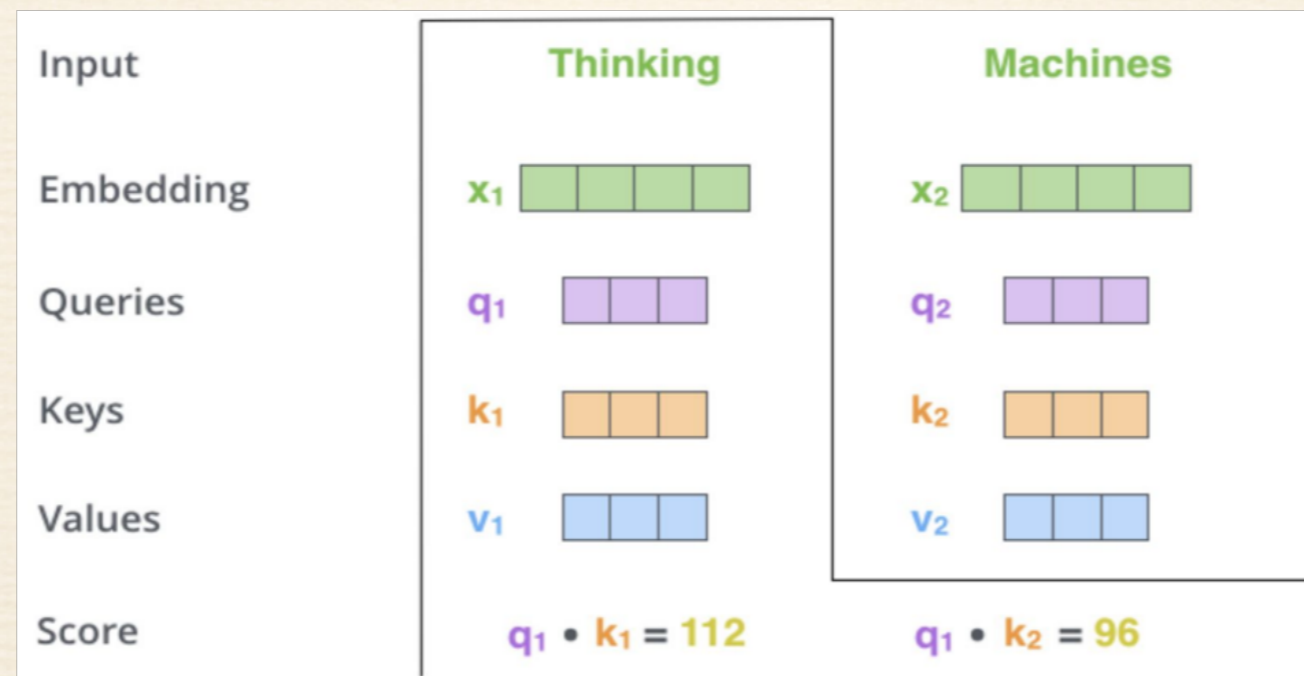
Self attention

- ❖ 1. Create three vectors corresponding to each of the encoder's input vectors
- ❖ For each word, we create a **Query**, a **Key**, and a **Value** vector, by multiplying the embedding by **three matrices** that we train during this training process



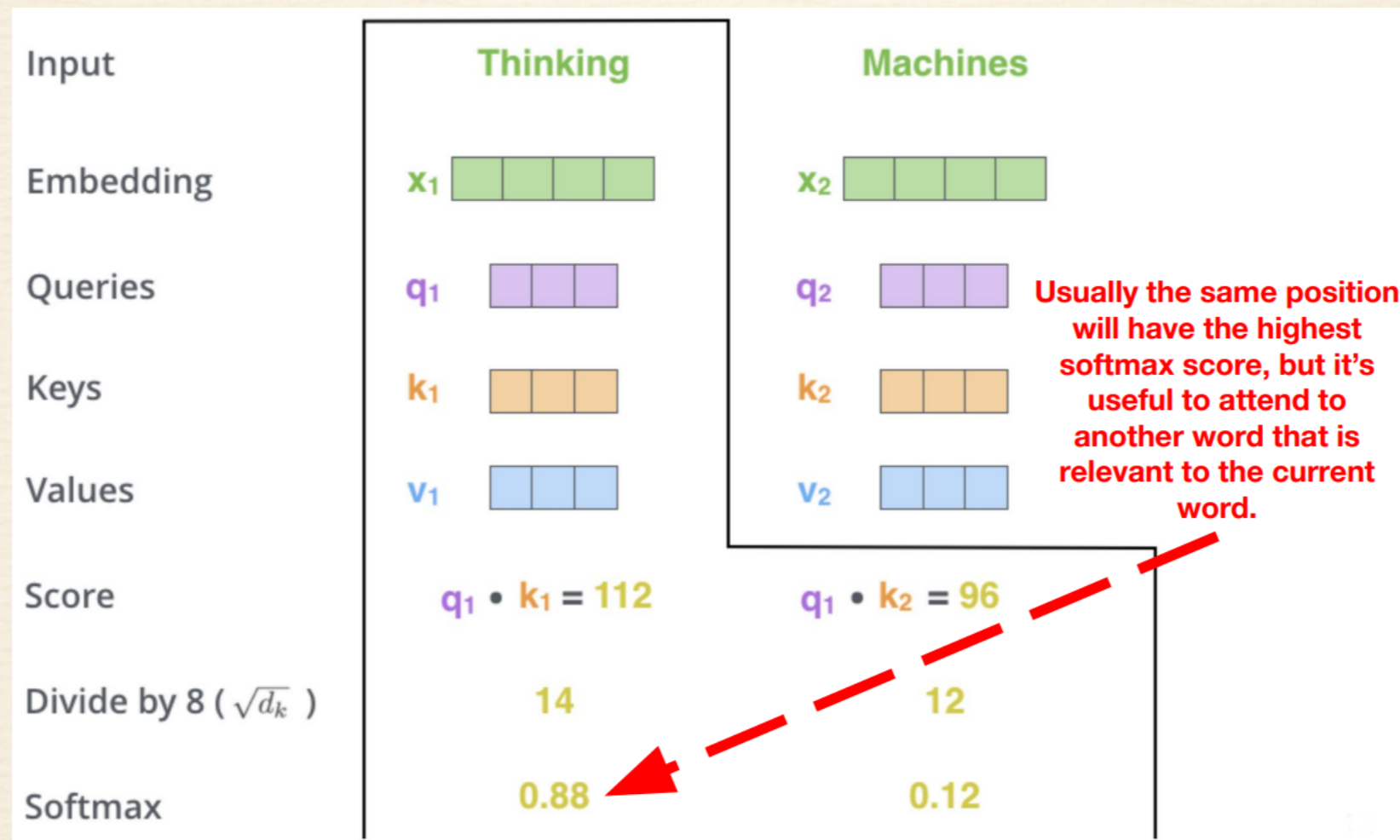
Self attention

- ❖ 2. Calculate the **attention score**, which determines how much we should focus on the other words in the input sentence as we encode this word
- ❖ scores each word in the input sentence against all other words
- ❖ calculated by taking the dot product of the **query** vector with the **key** vector of the respective words we're scoring



Self attention

- ❖ 3. Divide the scores by 8 (sqrt of dimension, this is just a hyperparameter), then pass them to the Softmax



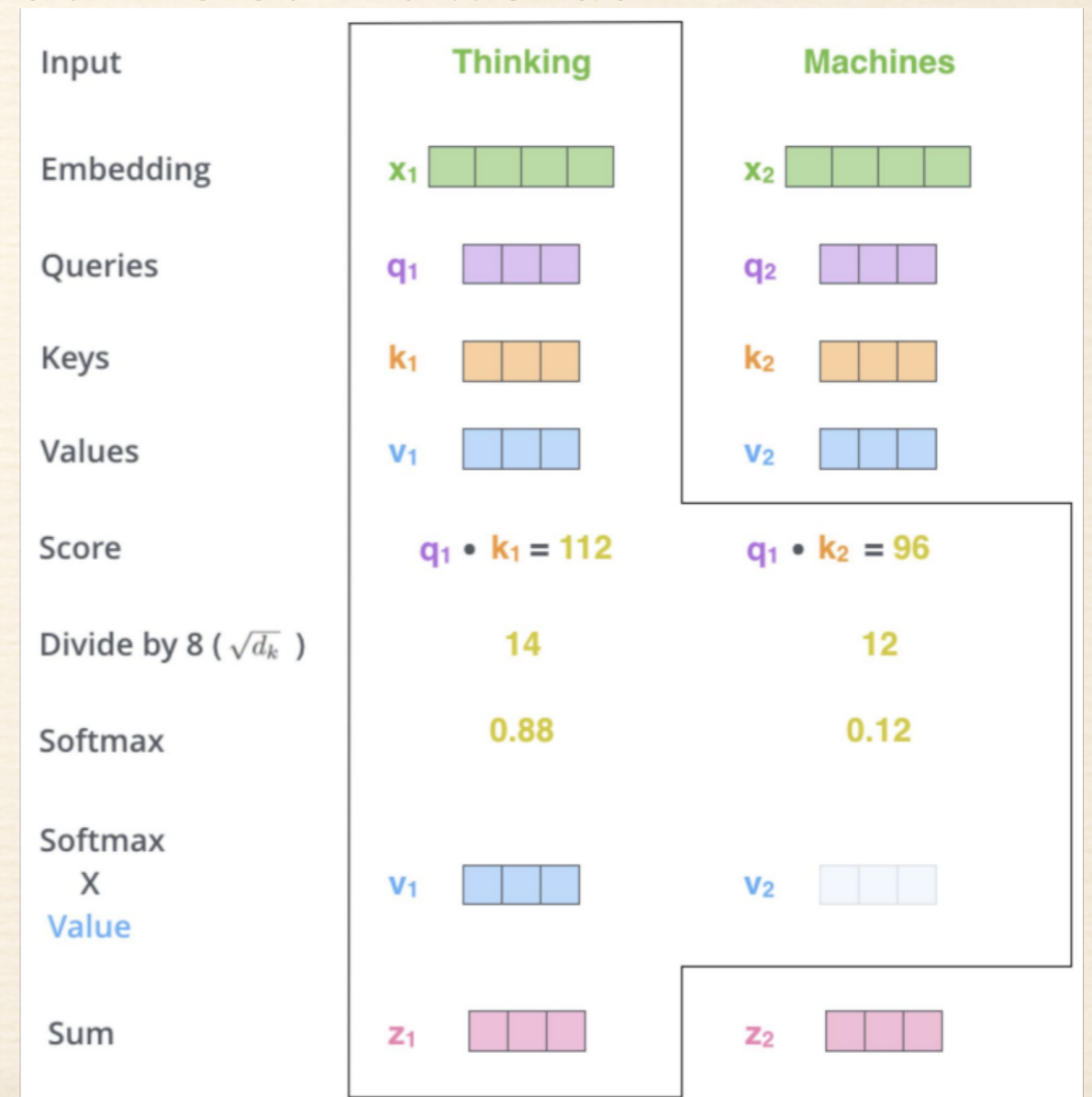
Self attention

- ❖ 4. Multiply **value** vectors of all words by the Softmax score
- ❖ The intuition is to keep intact the values of the word(s) we want to focus on and drown-out irrelevant words

- ❖ Sum up the weighted value vectors

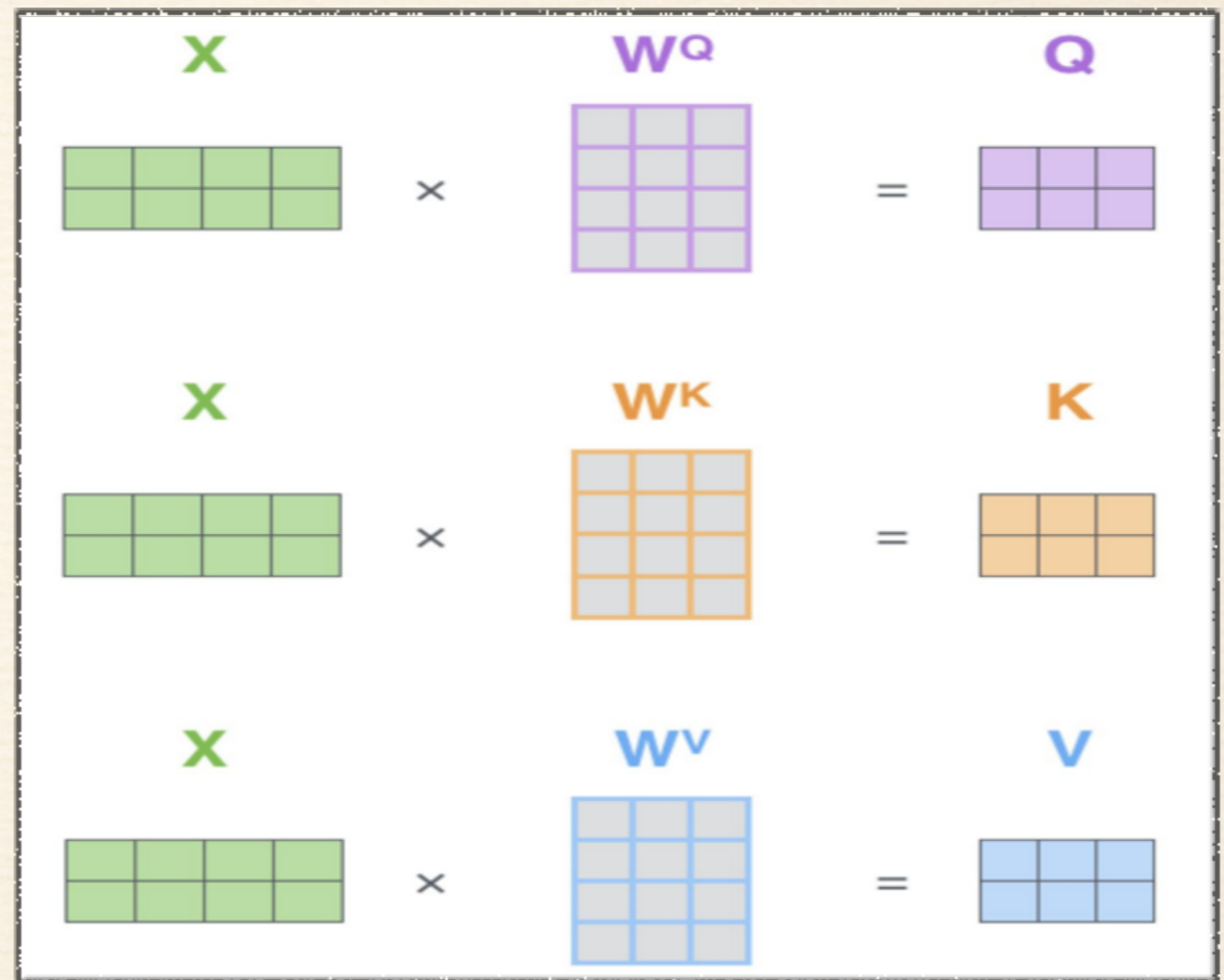
- ❖ Obtain the output of the self-attention layer at this position (for the first word)

- ❖ The resulting vector is sent to the feed-forward neural network.



Efficient computation in matrix terms

- ❖ Calculate the **Query**, **Key**, and **Value** “matrices” for the entire context, by packing our embeddings into a matrix **X** and multiplying it by the weight matrices



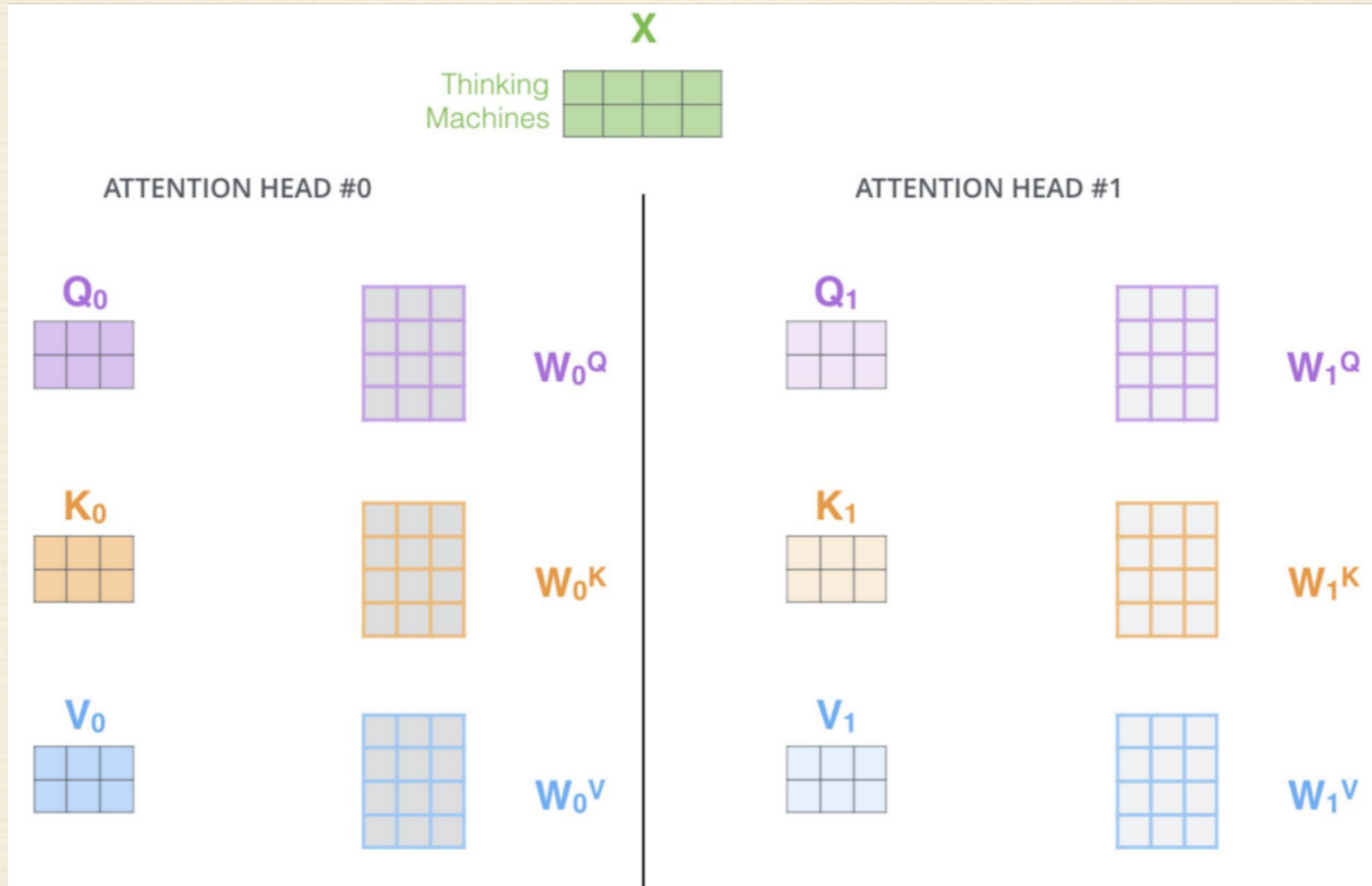
Efficient computation in matrix terms

❖ Outputs of the self-attention layer:

$$Z = \text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) V$$

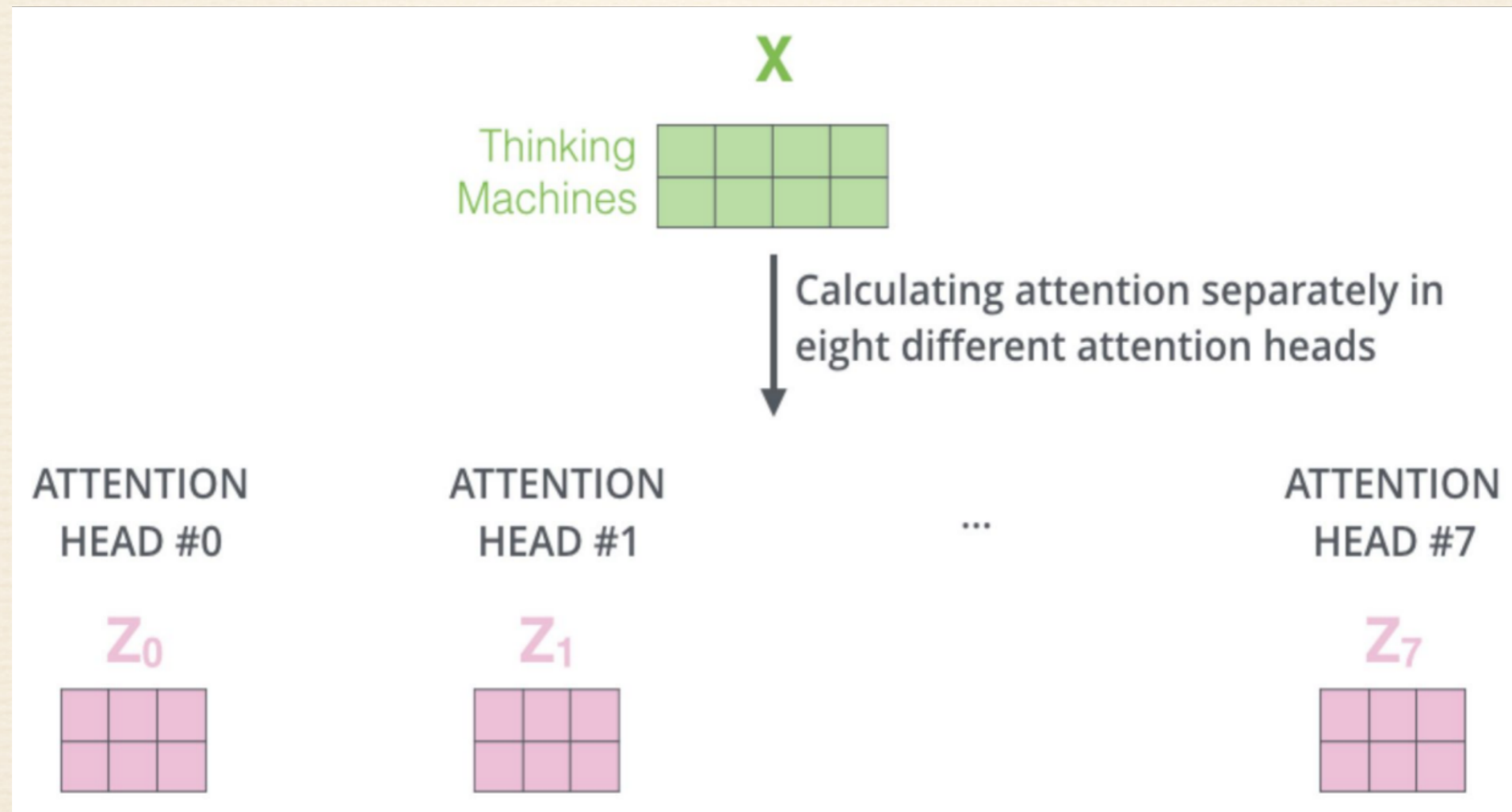
The diagram illustrates the computation of the output Z of a self-attention layer. It is shown as the product of a softmax function and a value matrix V . The softmax function takes the product of the query matrix Q and the key matrix K^T as input, scaled by the square root of the key dimension d_k . The matrices are represented as grids: Q is a 2x3 purple grid, K^T is a 3x2 orange grid, V is a 2x3 blue grid, and Z is a 2x3 pink grid.

A few more ornaments: (a) Multiple heads



A few more ornaments: (a) Multiple heads

- ❖ We end up with multiple different Z matrices



A few more ornaments: (a) Multiple heads

- ❖ Concatenate Z matrices and then multiply with an additional weight matrix

1) Concatenate all the attention heads



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

\times



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



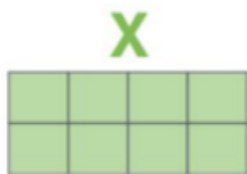
A few more ornaments: (a) Multiple heads

❖ Putting all together:

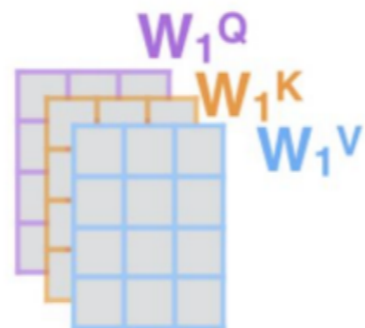
1) This is our input sentence*

Thinking Machines

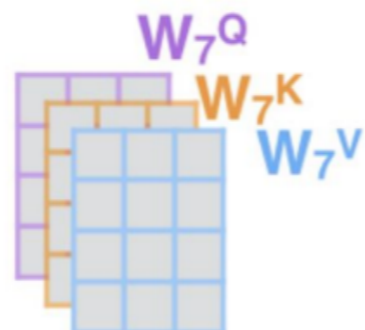
2) We embed each word*



3) Split into 8 heads. We multiply X or R with weight matrices



...



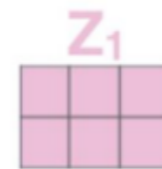
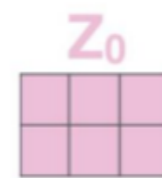
4) Calculate attention using the resulting $Q/K/V$ matrices



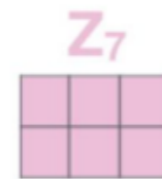
...



5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer



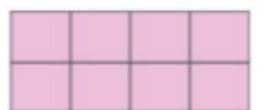
...



W^O

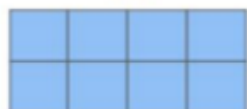


Z



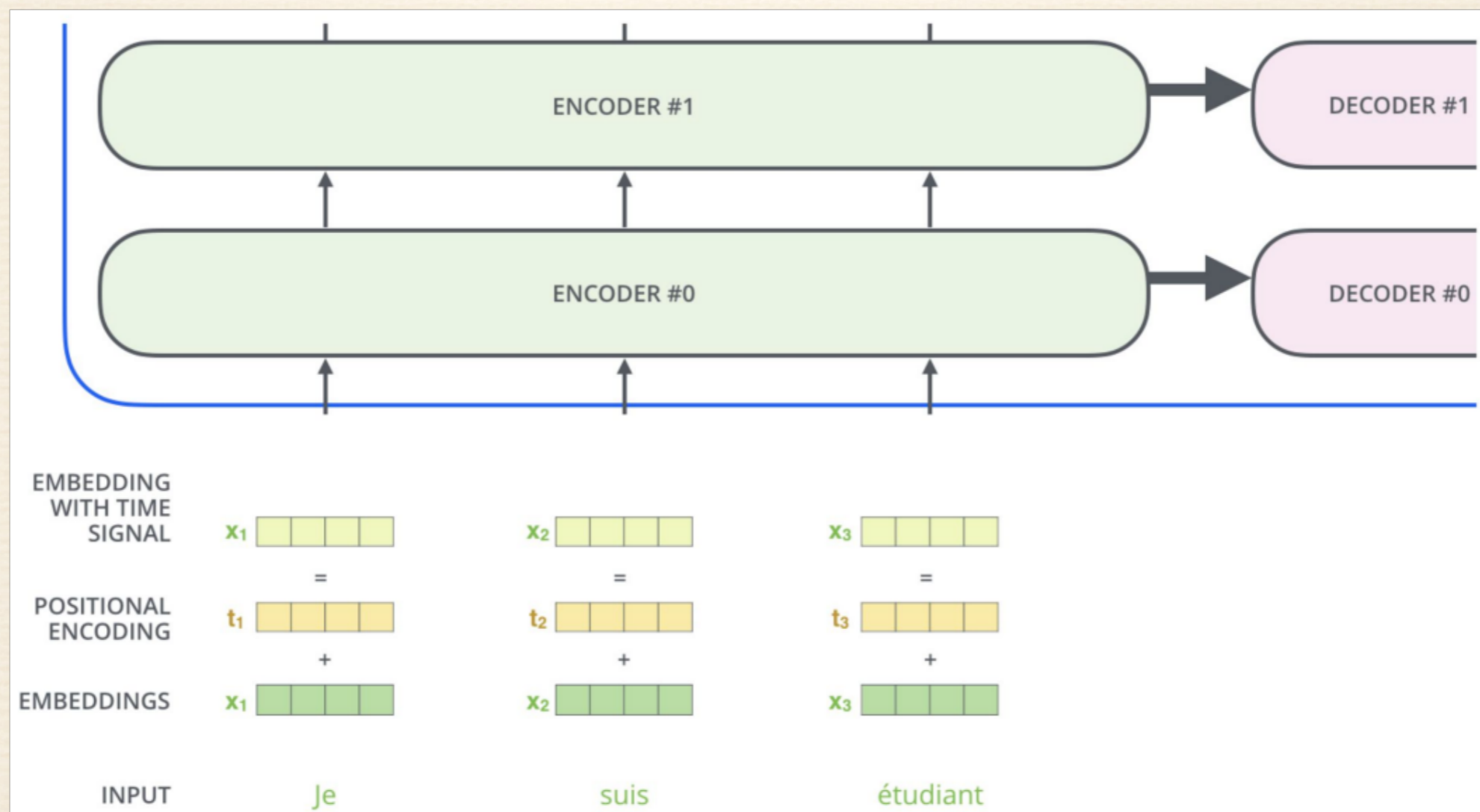
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

R



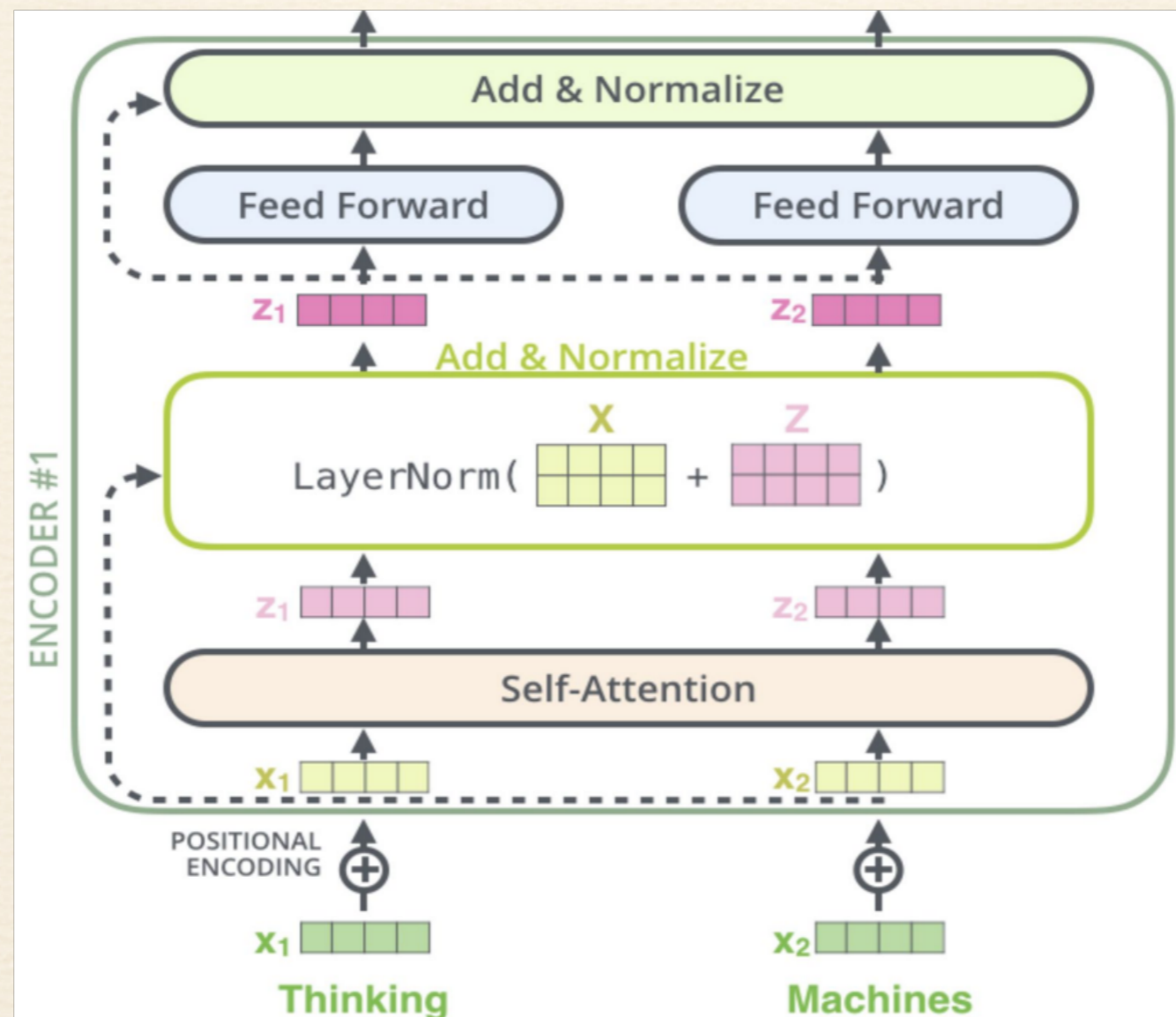
A few more ornaments: (b) Positional encodings

- ❖ So far, we have no way to account for the order of the words in the input sequence.
- ❖ To address this, we add a vector corresponding to each input position which helps it determine the position of each word.
- ❖ Positional encodings can be fixed (following a specific pattern) or may be learnt



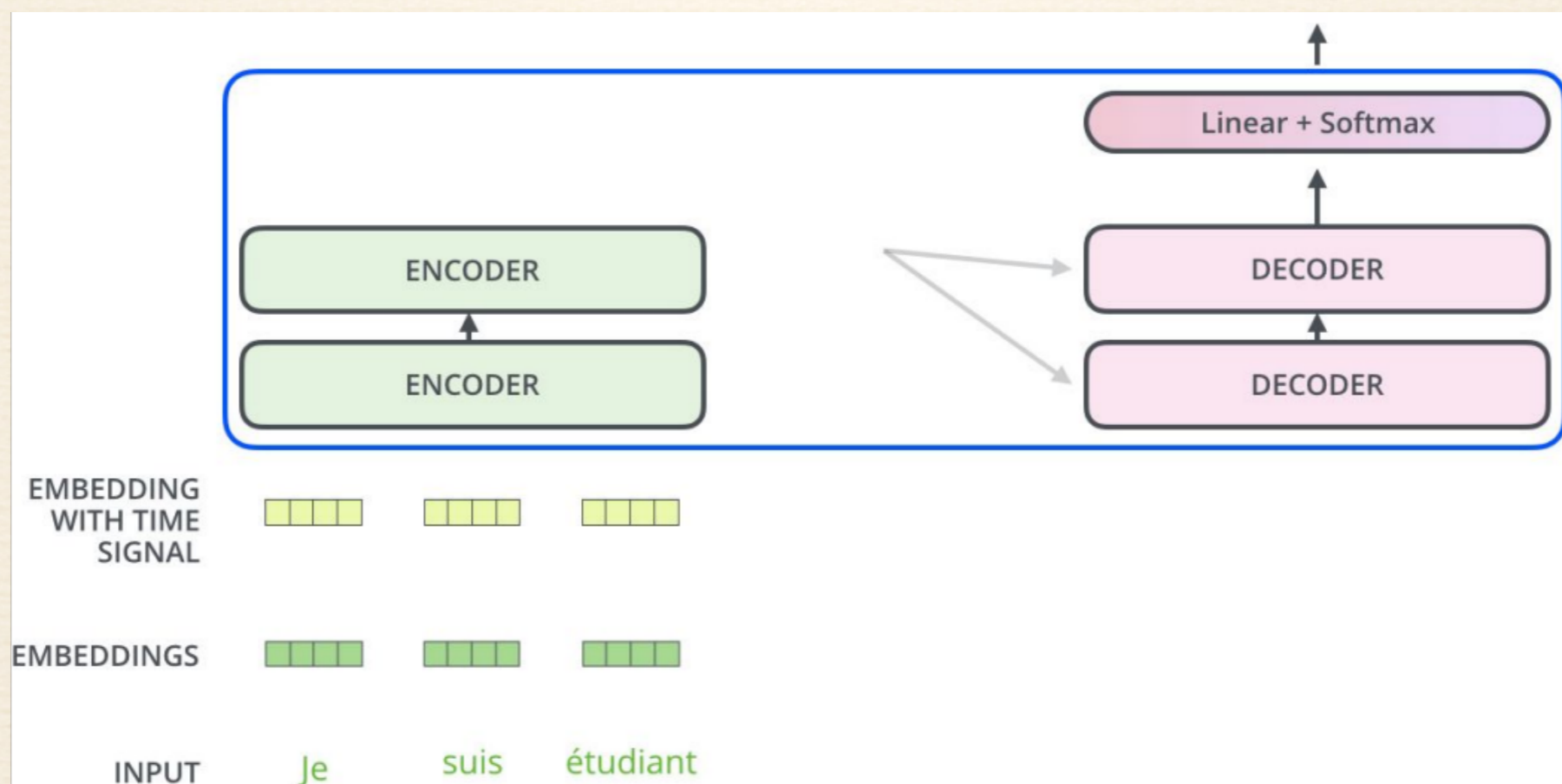
A few more ornaments: (c) Residuals

- ❖ A few add-ons to stabilize training of deeper networks: Each sub-layer (self-attention, ffn) in each encoder has a residual connection around it, and is followed by a layer-normalization step



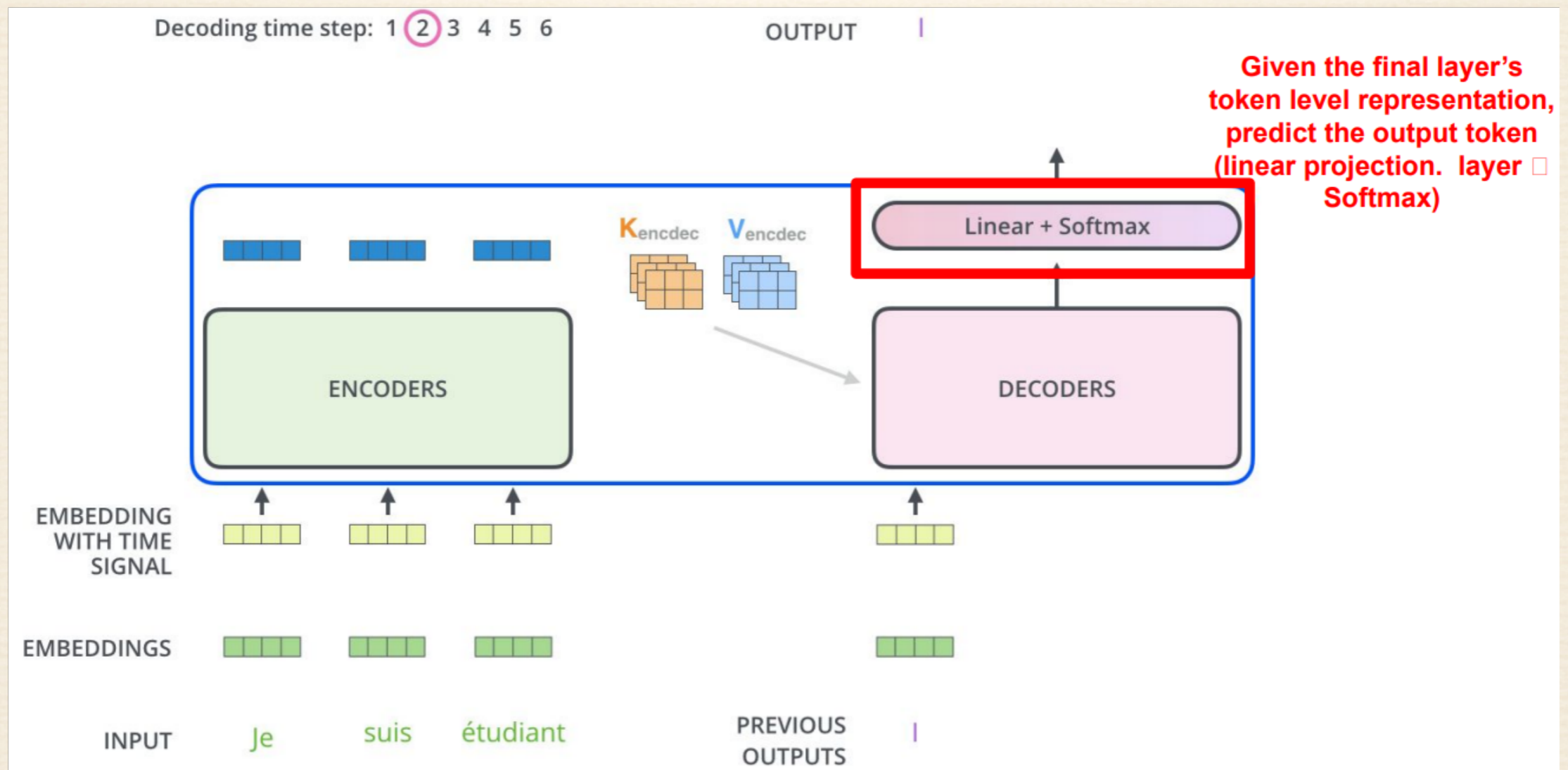
The Encoder-Decoder in synch

- ❖ The decoder is very similar. But it must look at the input representations
- ❖ Output of the top encoder is transformed into a set of key and value attention vectors.
- ❖ These are used by each decoder in a “encoder-decoder attention” layer (also called cross-attention) which helps the decoder focus on appropriate places in the input sequence



The Encoder-Decoder in synch

- ❖ In the decoder, at test time, the output is generated token by token as the output of each step is fed to the bottom decoder in the next time step.



Masking

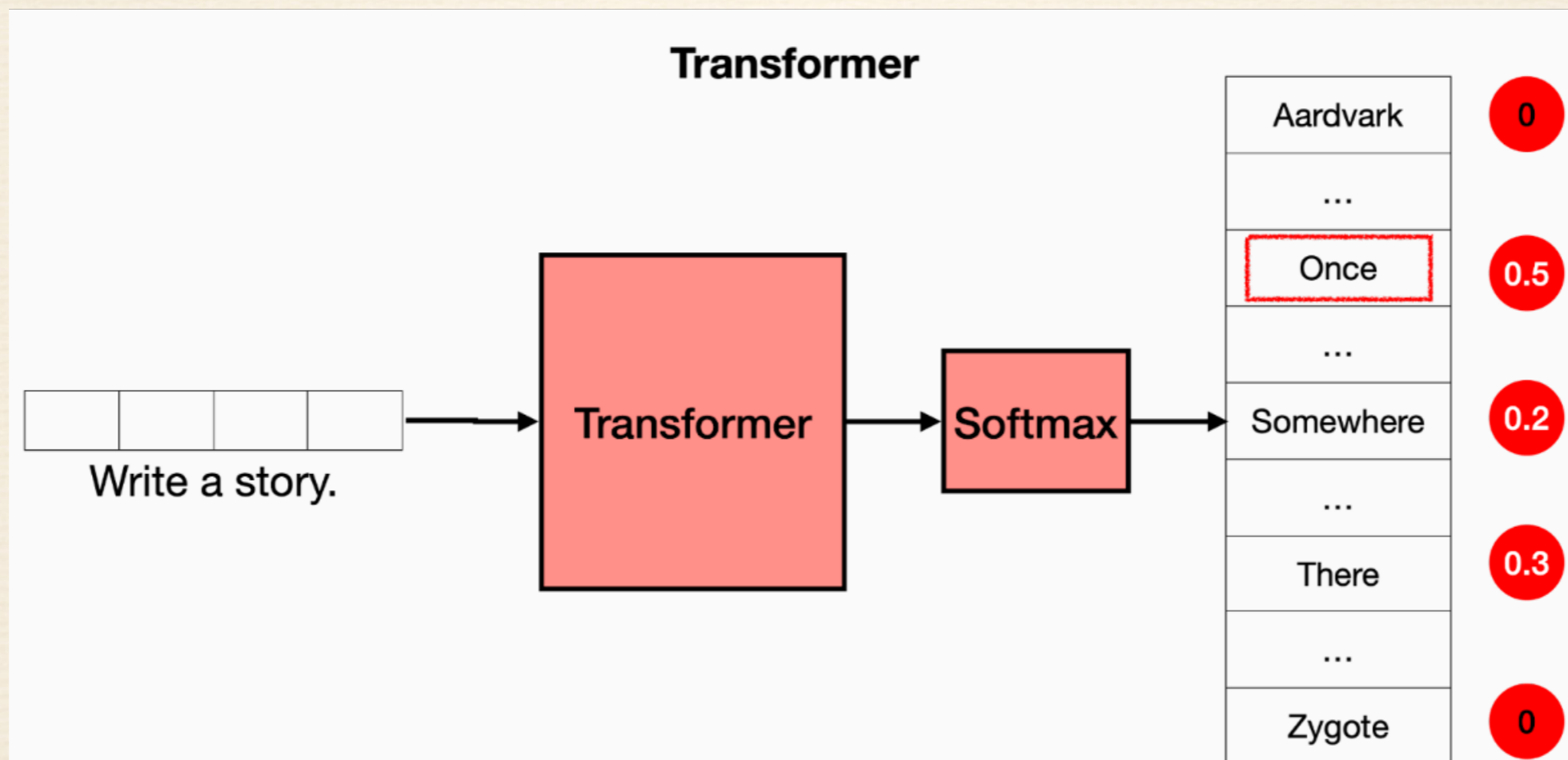
- ❖ In the decoder, the self-attention layer is only allowed to attend to **earlier positions** in the output sequence. Otherwise, we'd be cheating!
- ❖ This is done by **masking future positions** (setting the dot product score to $-\infty$) before the Softmax step in the self-attention calculation.

$$Z = \text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) \times V$$

The diagram shows the calculation of the output Z (pink 2x3 matrix) as the product of the softmax of the scaled dot product of Q (purple 2x3 matrix) and K^T (orange 3x2 matrix) and V (blue 2x3 matrix). A red arrow labeled '+ M' points to a yellow 2x2 matrix M, which represents the mask added to the attention scores. The top-right cell of M contains '-inf', indicating that the score for attending to a future position is set to negative infinity.

Transformer: a summary

- ❖ The last step: a softmax layer turns scores into probabilities
- ❖ Repeat: input the text “Write a story. Once” into the model, and most likely, the output will be “upon”
- ❖ Repeating this step again and again, the transformer will end up writing a story, such as “Once upon a time, there was a ...”.



The softmax layer turns the scores into probabilities, and these are used to pick the next word in the text.

Transformer: a summary

- ❖ **Tokenizer:** Turns words into tokens
- ❖ **Embedding:** Turns tokens into numbers (vectors)
- ❖ **Positional encoding:** Adds order to the words in the text
- ❖ **Transformer block:** Guesses the next word. It is formed by an attention block and a feedforward block
- ❖ **Attention:** Adds context to the text
- ❖ **Feedforward:** Is a block in the transformer neural network, which guesses the next word
- ❖ **Softmax:** Turns the scores into probabilities in order to sample the next word

Key components: Transfer learning

- ❖ Transformers are not always better than RNNs by themselves
- ❖ However, what has made Transformers popular is that they can be combined with the idea of transfer learning
- ❖ Transformers have become the go-to model for building **large pretrained language models** which can be adapted for several tasks
- ❖ The idea of transfer learning is to use the knowledge gained while solving one problem and applying it to a different but related problem

Key components: Transfer learning

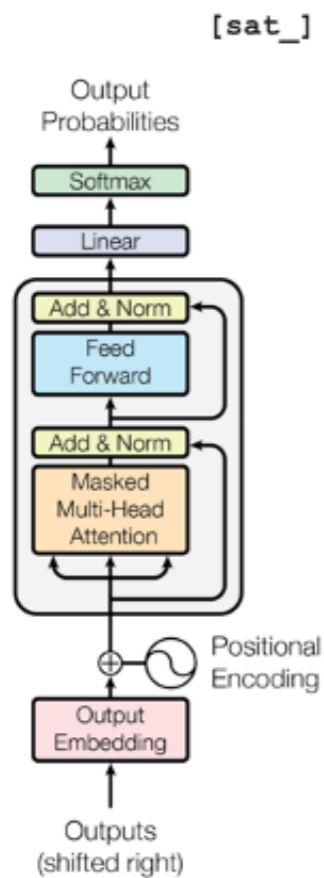
- ❖ For example, knowledge gained while learning to recognize cars could be applied when trying to recognize trucks



- ❖ In fact, if we have a large dataset of cars, we can pretrain a model on this dataset, and may be able to do well on recognizing trucks by **finetuning** the model on a small dataset of trucks
- ❖ In natural language processing (NLP), we can train a NN language model (on vast text corpora), and then use it to transfer that knowledge to any target task in NLP we care about

Notable LLMs

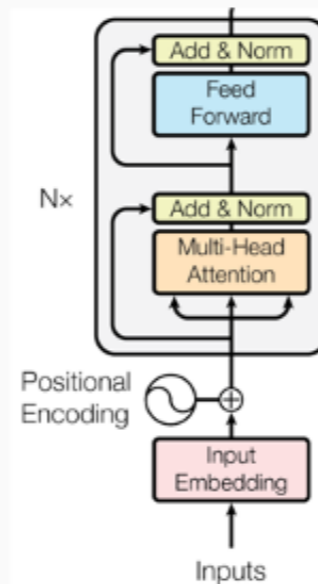
Decoder-only GPT



[START] [The_] [cat_] [sat_] [the_] [*]

Encoder-only BERT

[*] [*] [sat_] [*] [the_] [*]



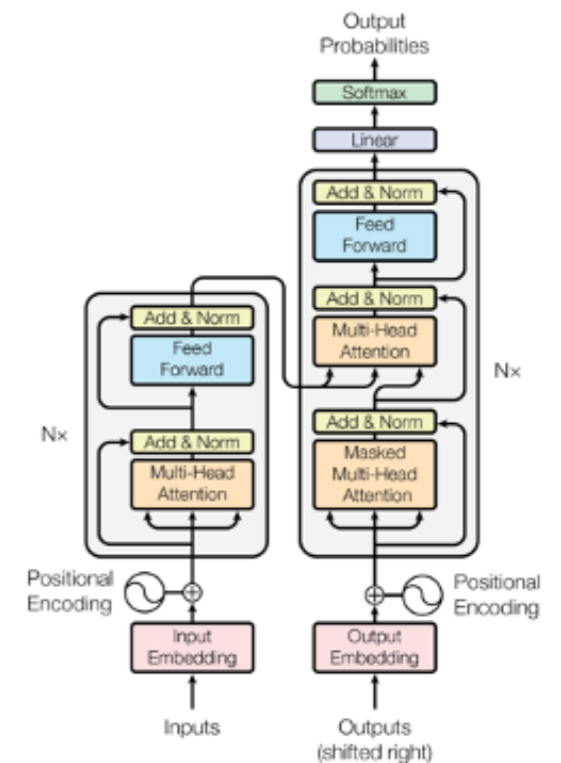
[The_] [cat_] [MASK] [on_] [MASK] [mat_] [sat_] [the_] [*]

Enc-Dec T5

Das ist gut.

A storm in Attala caused 6 victims.

This is not toxic.



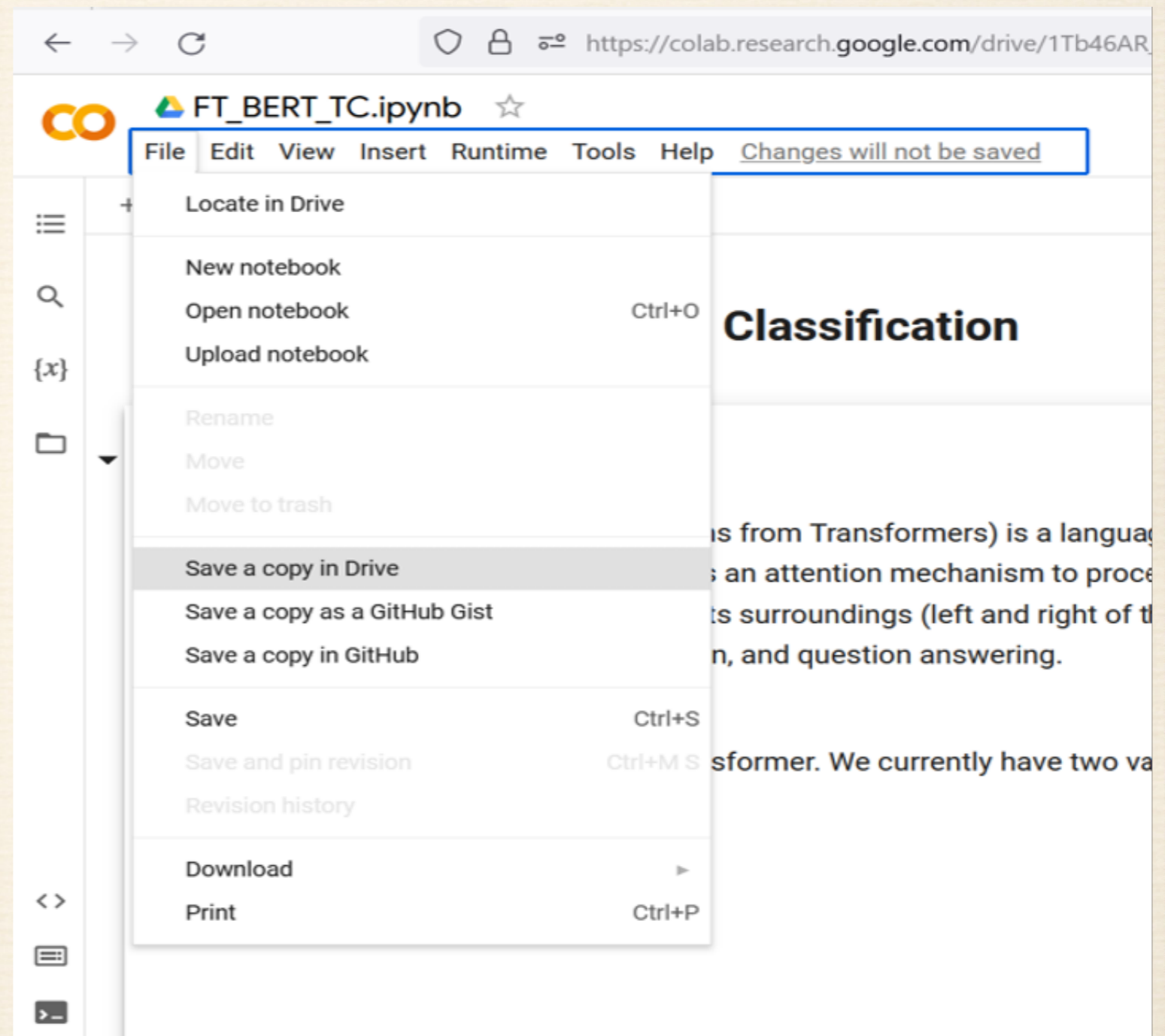
Translate EN-DE: This is good.

Summarize: state authorities dispatched...

Is this toxic: You look beautiful today!

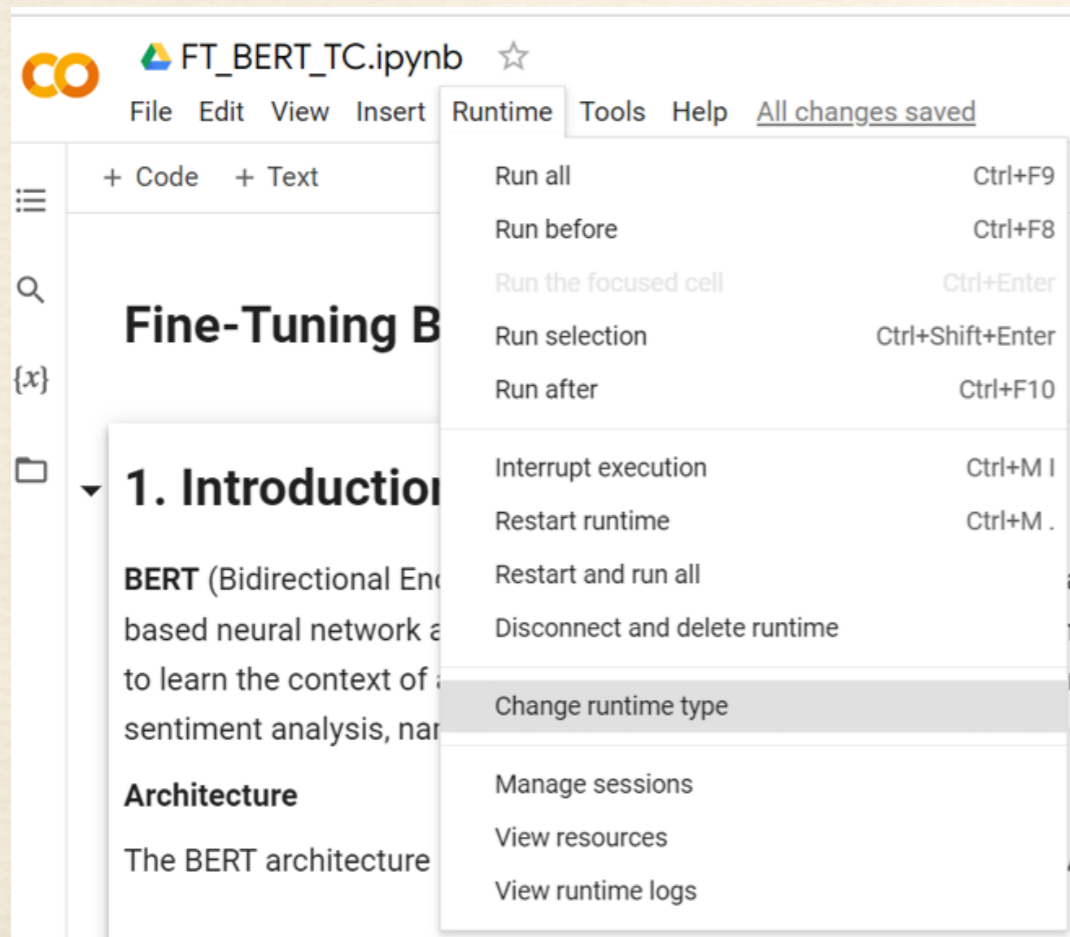
Lab 7 - preview

- ❖ Google account to use Google Colab
- ❖ Save a copy of this Colab notebook file into your google drive: File -> Save a copy in Drive
- ❖ Work on the saved copy



Lab 7 - preview

- ❖ Notebook setting: Runtime -> Change runtime type -> Hardware accelerator: GPU



The screenshot shows the JupyterLab interface for a notebook titled 'FT_BERT_TC.ipynb'. The 'Runtime' menu is open, and the 'Change runtime type' option is highlighted. The notebook content includes a section titled '1. Introduction' and a paragraph about BERT (Bidirectional Encoder Representations from Transformers) based neural networks used for sentiment analysis.

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

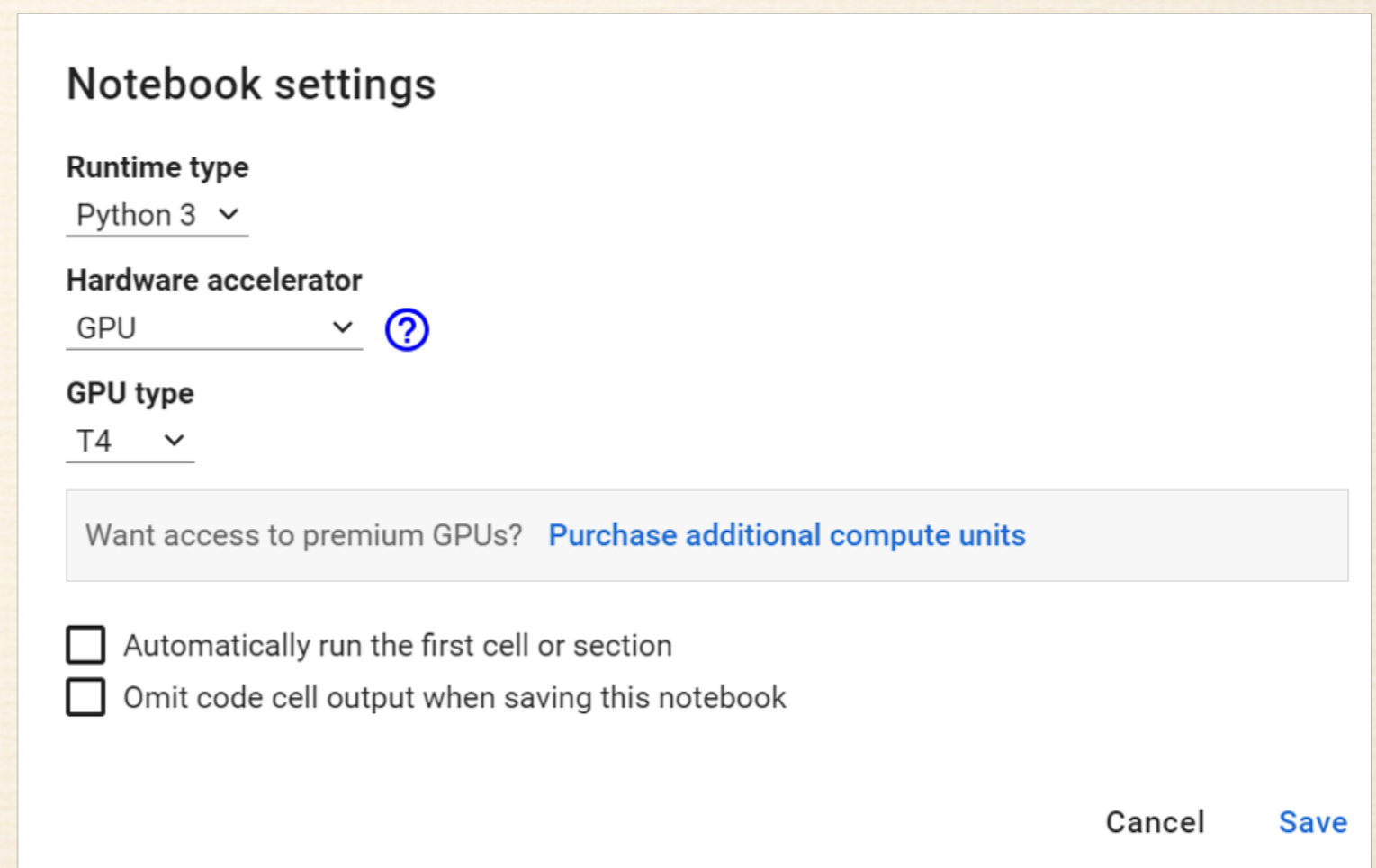
Fine-Tuning B

1. Introduction

BERT (Bidirectional Encoder Representations from Transformers) based neural network architecture is used to learn the context of words in a sentence for tasks like sentiment analysis, named entity recognition, and text classification.

Architecture

The BERT architecture



The 'Notebook settings' dialog box is shown, with the following options:

- Runtime type:** Python 3
- Hardware accelerator:** GPU
- GPU type:** T4

Want access to premium GPUs? [Purchase additional compute units](#)

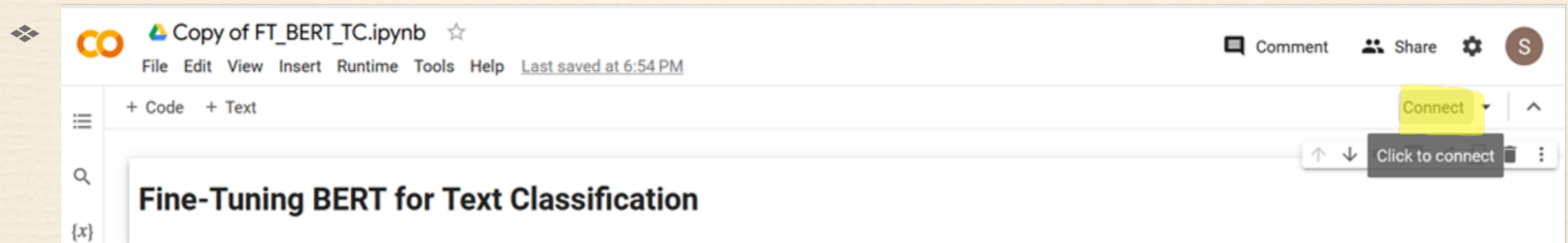
Automatically run the first cell or section

Omit code cell output when saving this notebook

Cancel Save

Lab 7 - preview

- ❖ Click on connect, it will connect to a hosted runtime on cloud by default



- ❖ Now, you are ready to execute each coding cell sequentially by clicking on the Run cell button

