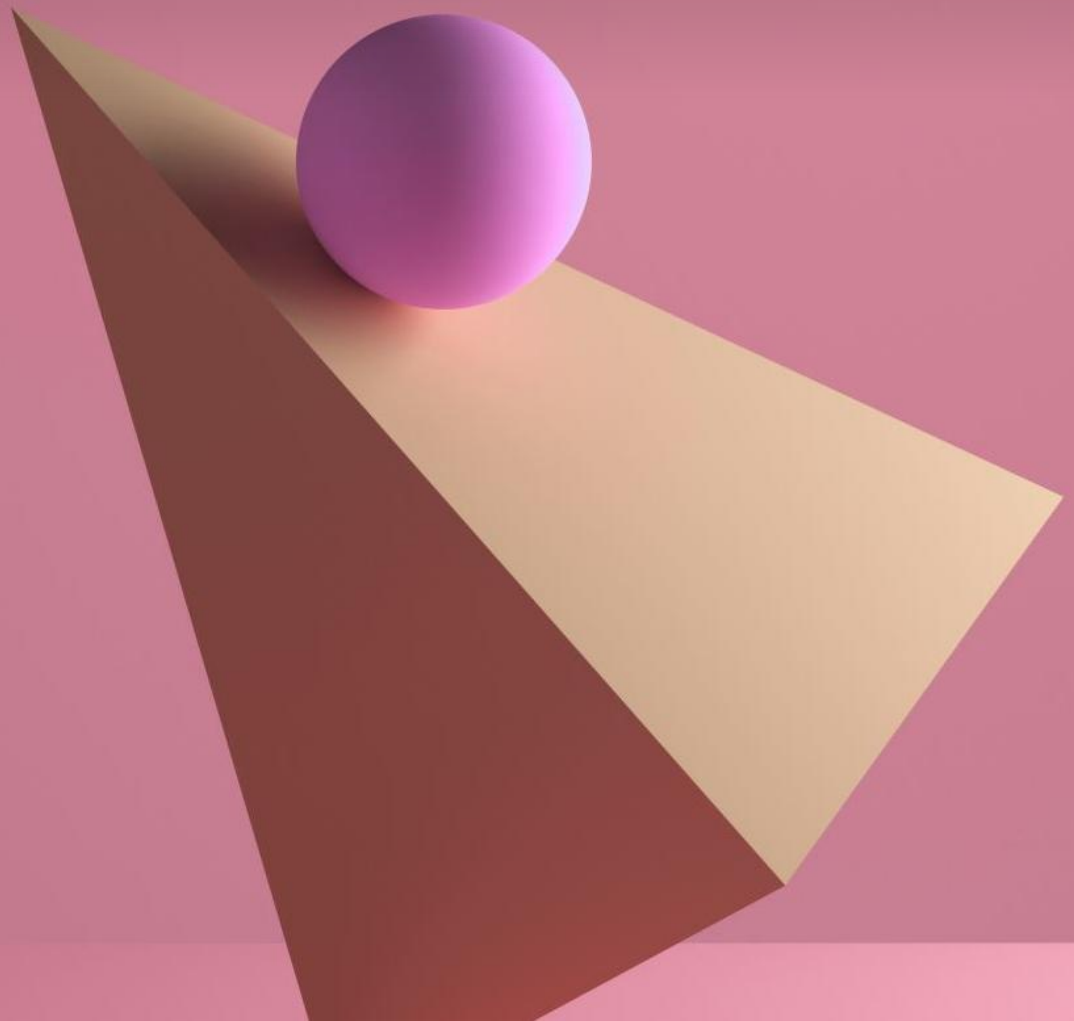
MINERVA

Minerva AI Winter School

Gianluca Carlini
Cineca

Bologna, 24/03/2026

Introduction to Large Language Models



In the beginning was the matrices

Matrix multiplication is essentially the main process powering Deep Learning models but, **what is a matrix?**

In the beginning was the matrices

Matrix multiplication is essentially the main process powering Deep Learning models but, **what is a matrix?**

A matrix is a representation of a linear map between two vector spaces:

$$T: V \to W$$

Here T is our transformation from a vector space V to a vector space W. We can represent our transformation as an array of scalars A, that we can apply to our vector $\boldsymbol{v}$.

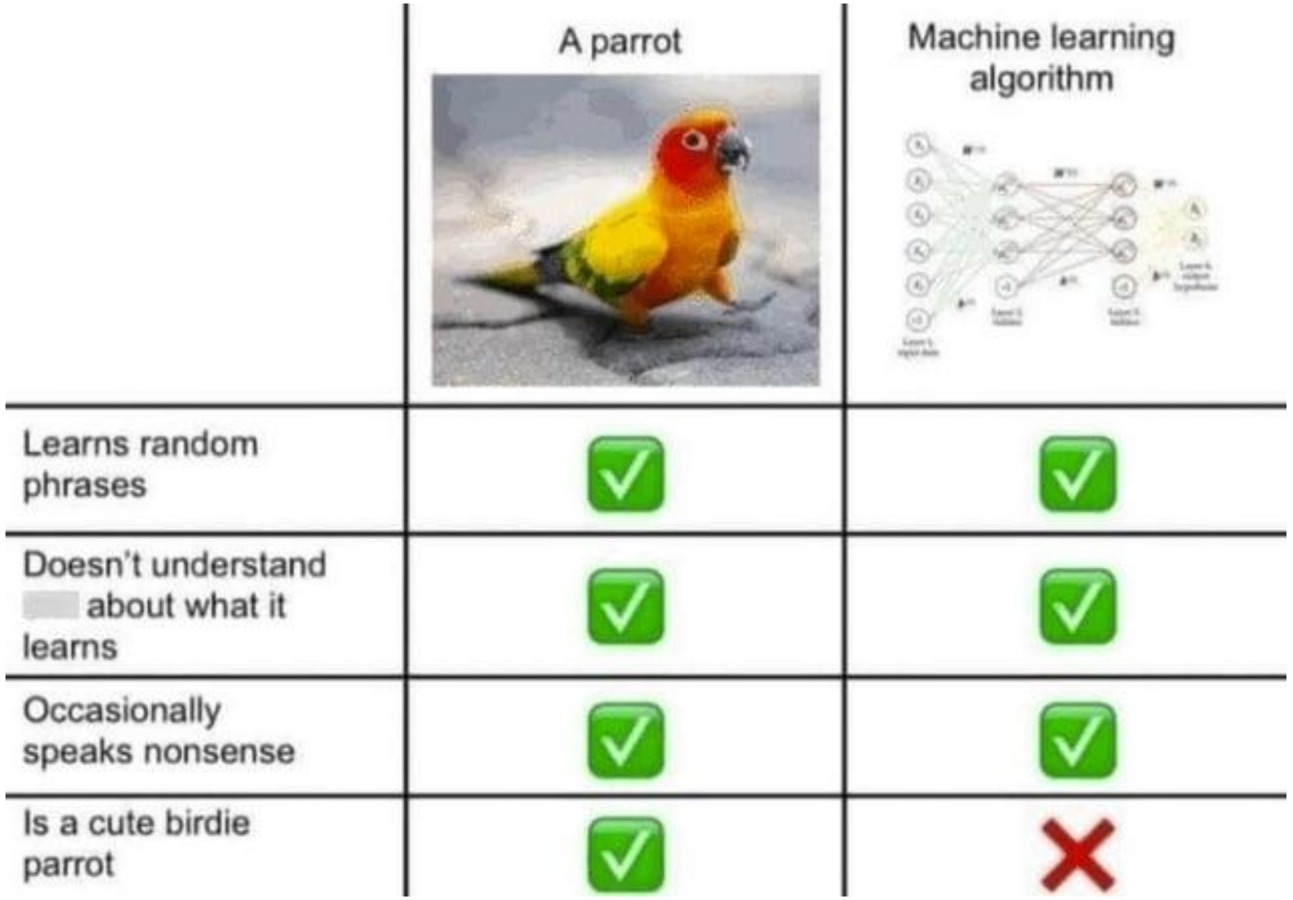
$$T(\boldsymbol{v}) = A\boldsymbol{v}$$

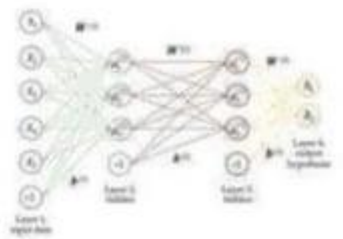
Large Language Models

Large Language Models (LLMs) are a special kind of NNs specialized in the analysis of text.

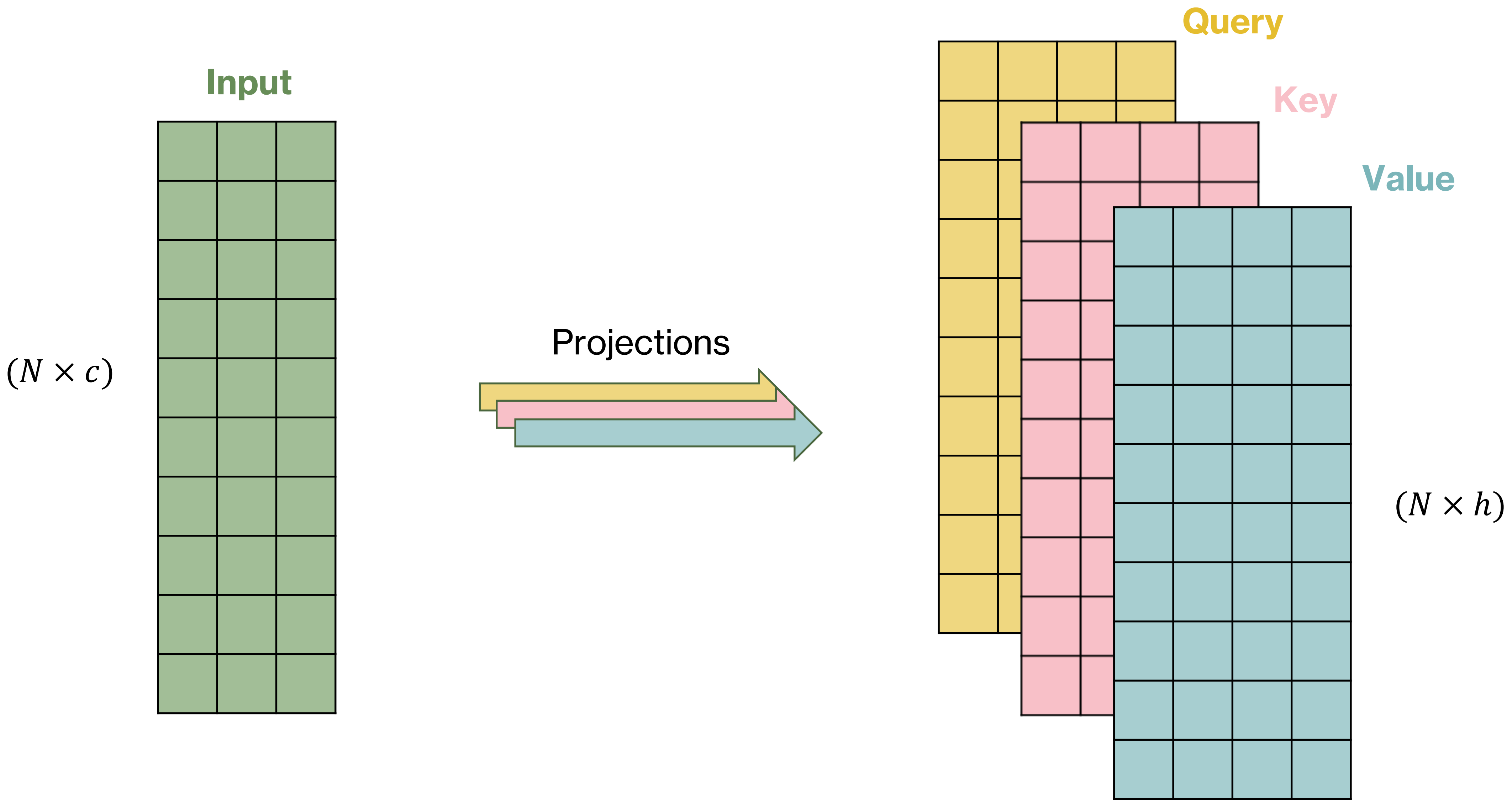
Their underlying mechanics are identical to NNs we already know.

They are no magic, just matrix multiplication and backpropagation

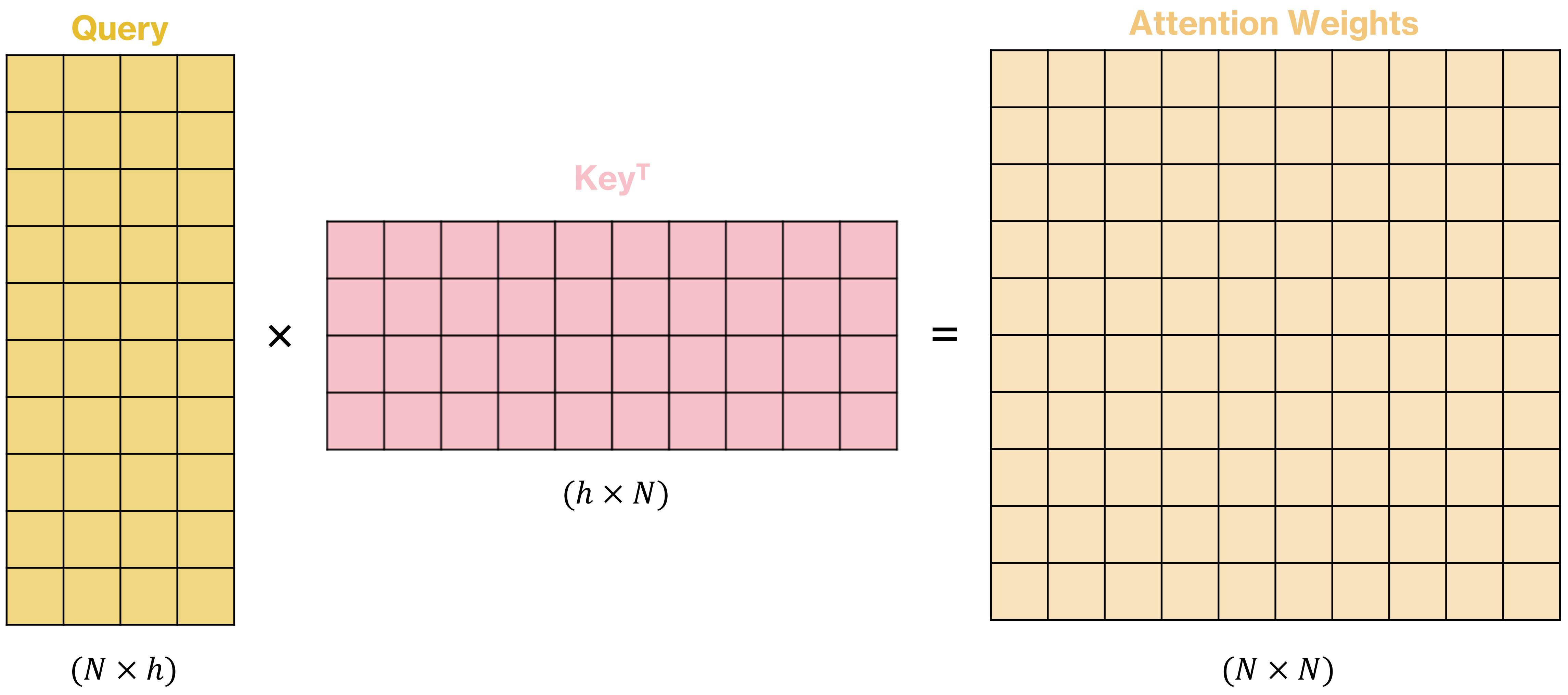


	A parrot	Machine learning algorithm
		
Learns random phrases	✅	✅
Doesn't understand [redacted] about what it learns	✅	✅
Occasionally speaks nonsense	✅	✅
Is a cute birdie parrot	✅	❌

Large Language Models: Attention



Large Language Models: Attention



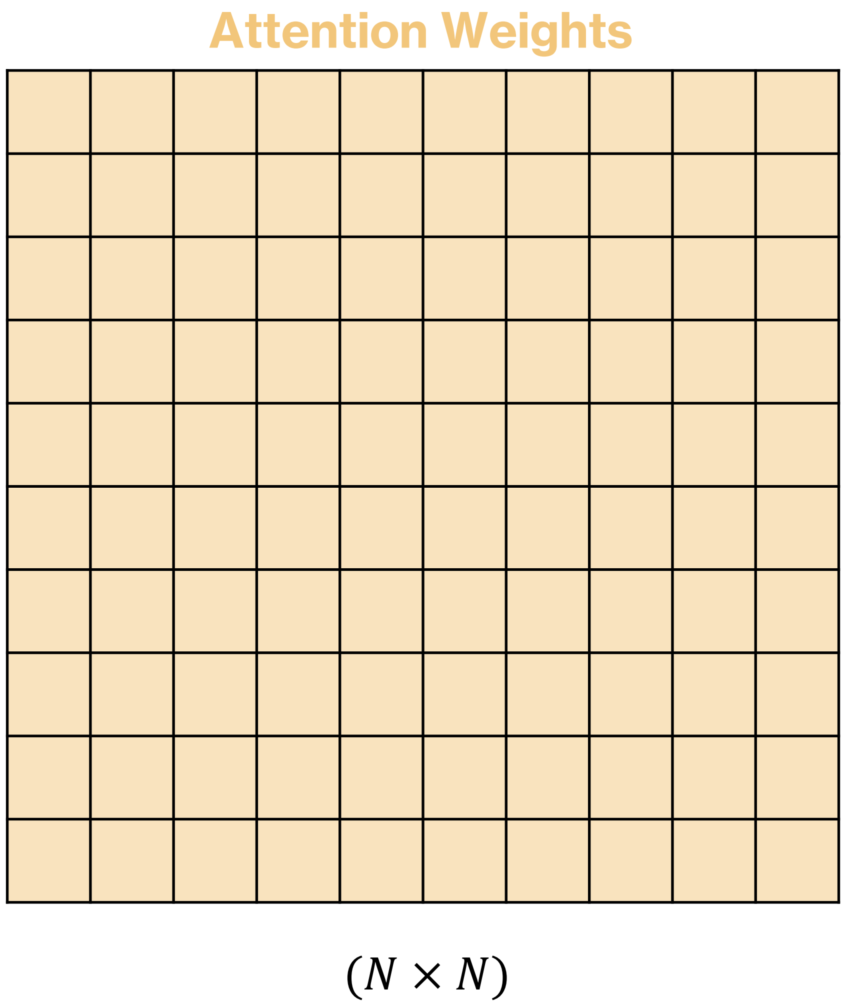
Large Language Models: Attention

Once we get our **attention weights** matrix, we apply the **softmax** function to it, *row-wise.*

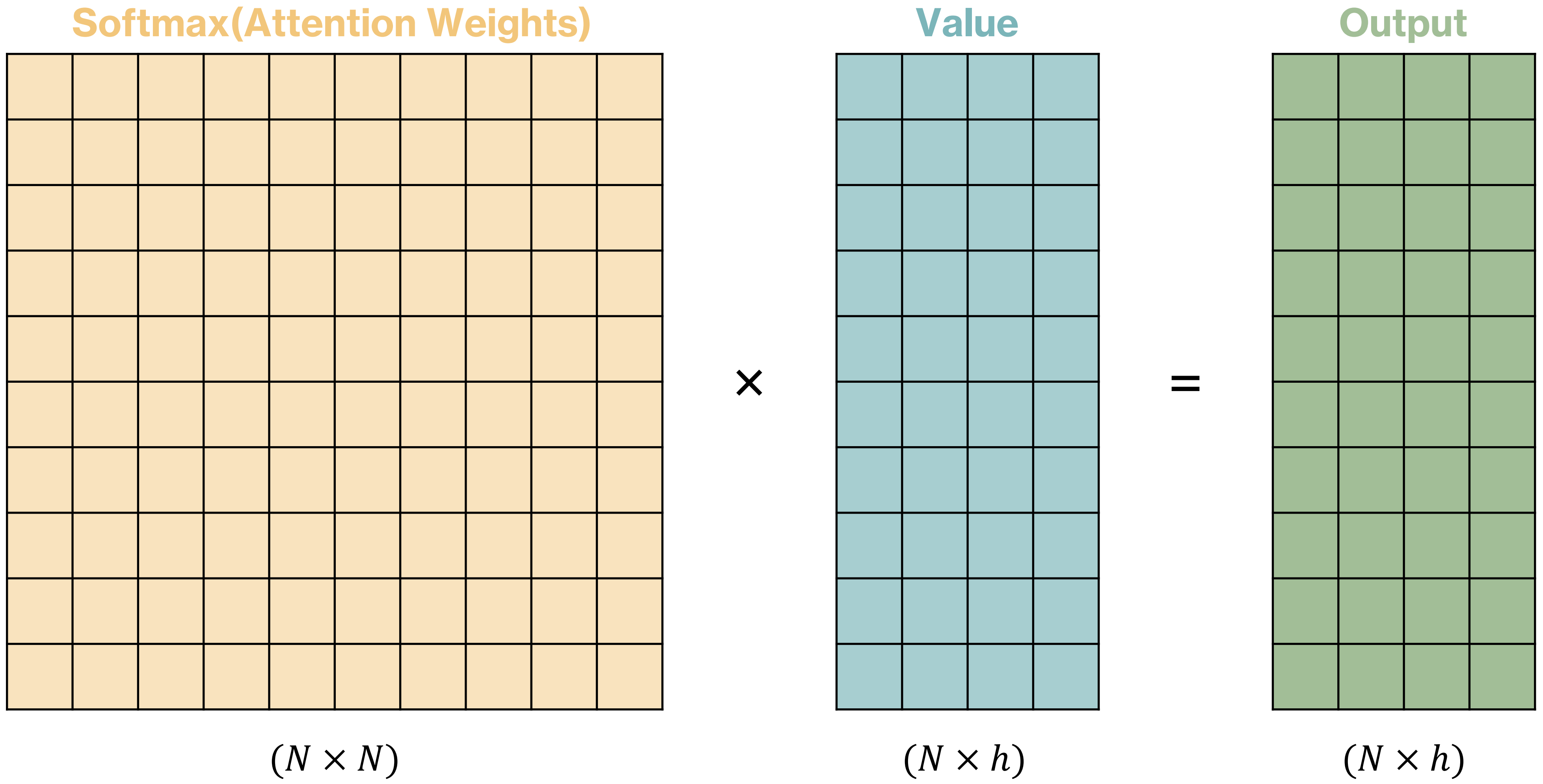
$$Softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

We need the softmax function to give some sort of *normalized importance* to each query-key pair in the matrix.

In this way we scale the values between 0 and 1, we ensure that the *cumulative importance* is 1, and we push larger values far apart from smaller ones.



Large Language Models: Attention

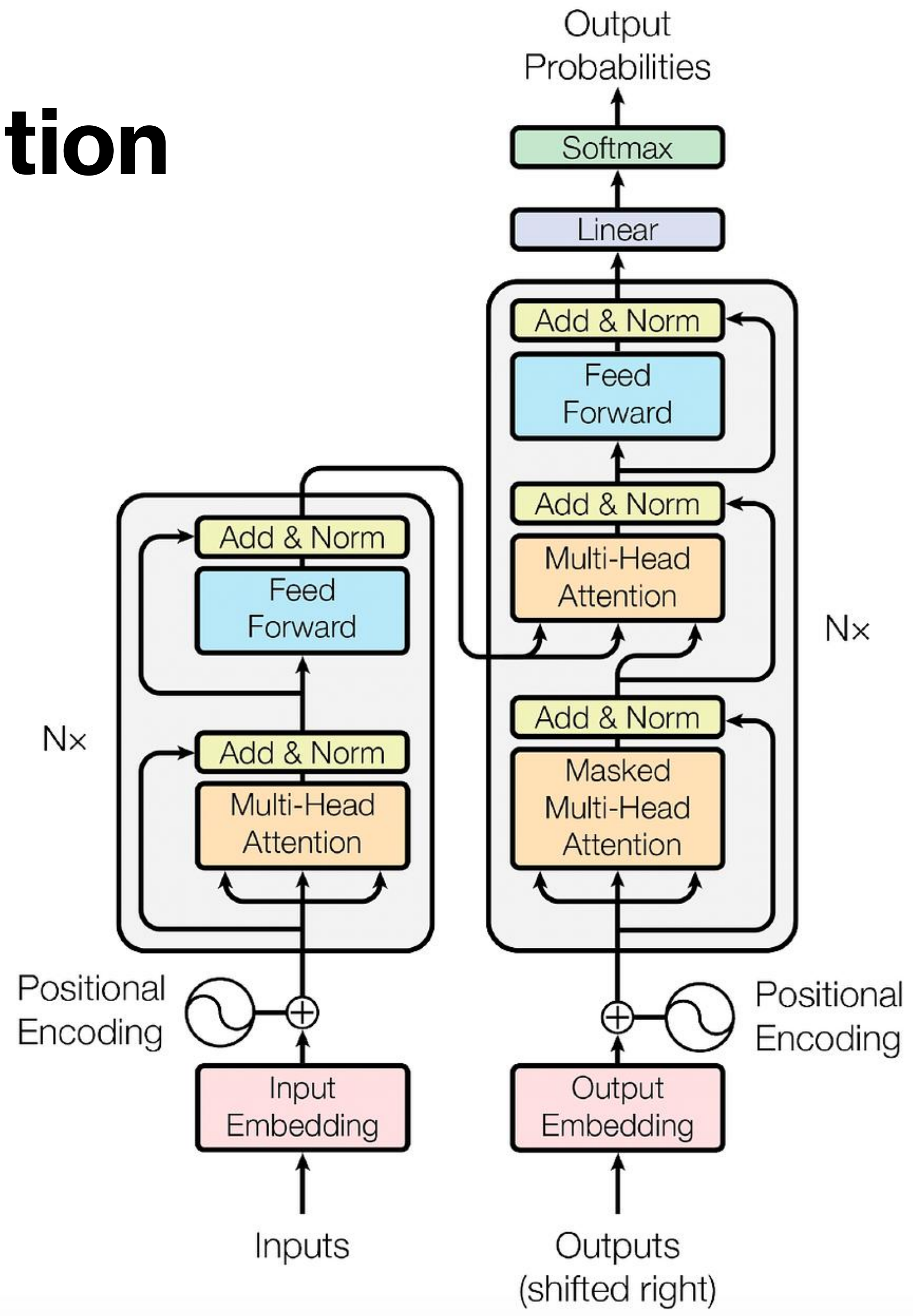


Large Language Models: Attention

From a mathematical point of view, the attention operation is the following:

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the projection dimension of the keys, queries and values.



Large Language Models: Attention

Attention was a breakthrough in the field of Deep Learning. It completely replaced other architectures such as Recurrent Layers.

It allows the network to grasp complex relations among the inputs and focus on the most important aspects of them.

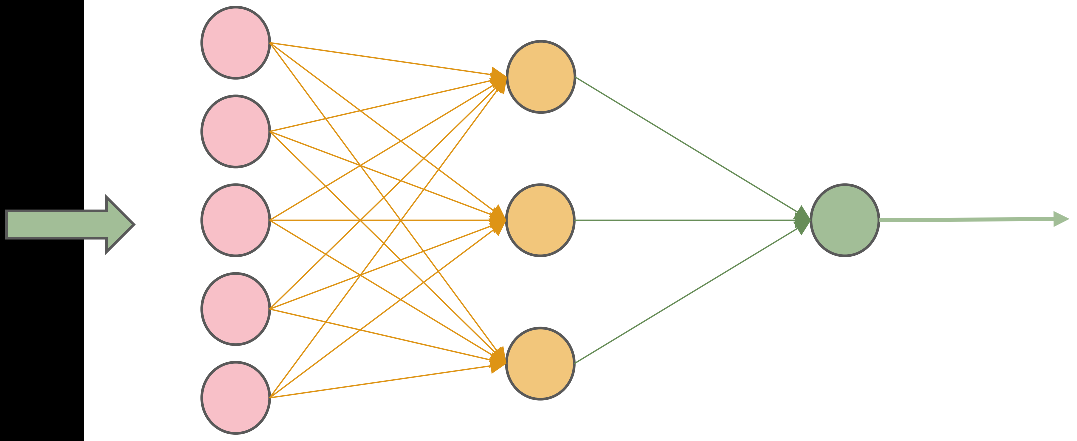
Models based on the attention mechanism are known as **Transformers**

A glimpse on the architecture

LLMs such as GPT are Transformers. It means they are stacks of subsequent **Transformer blocks**. A Transformer block in essentially an attention layer followed by an MLP, and we know how they both work, right?

Attention Block

MLP



Positional embedding

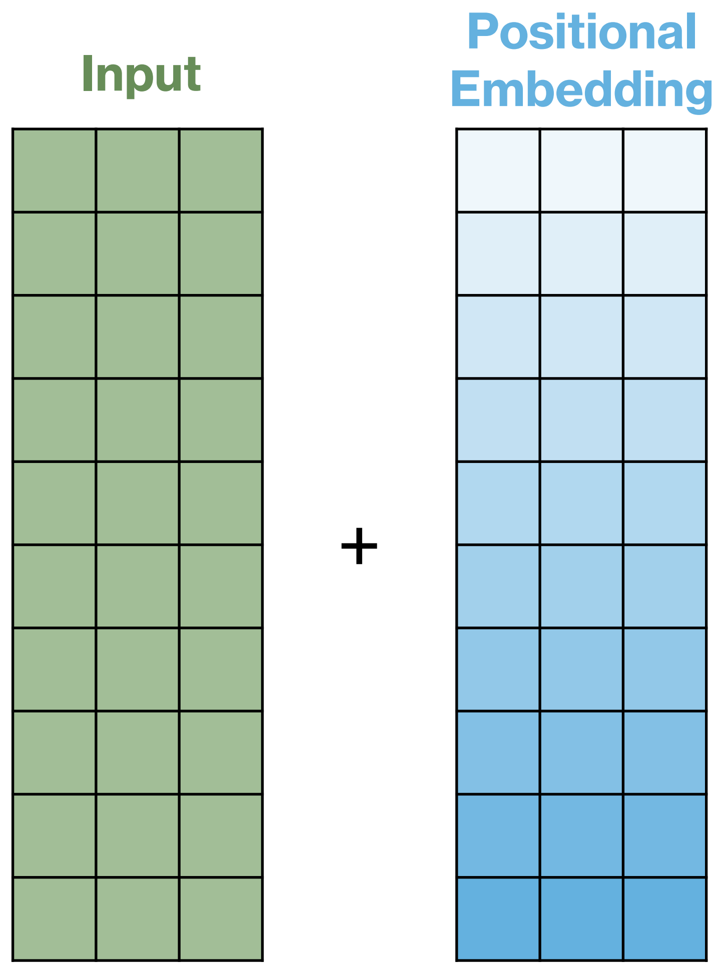
Attention calculation has a catch: it is **permutation invariant**.

This means if you shuffle the order of the input vectors, the set of output vectors remains exactly the same, they just move to new positions.

To the Attention mechanism, a sentence is just a "**bag of words**" until we give it a sense of time and order.

Without positional encoding, the model wouldn't know the difference between *“The dog bit the man”* and *“The man bit the dog”.*

We need to inject the concept of position to the model.



Positional embedding: Absolute

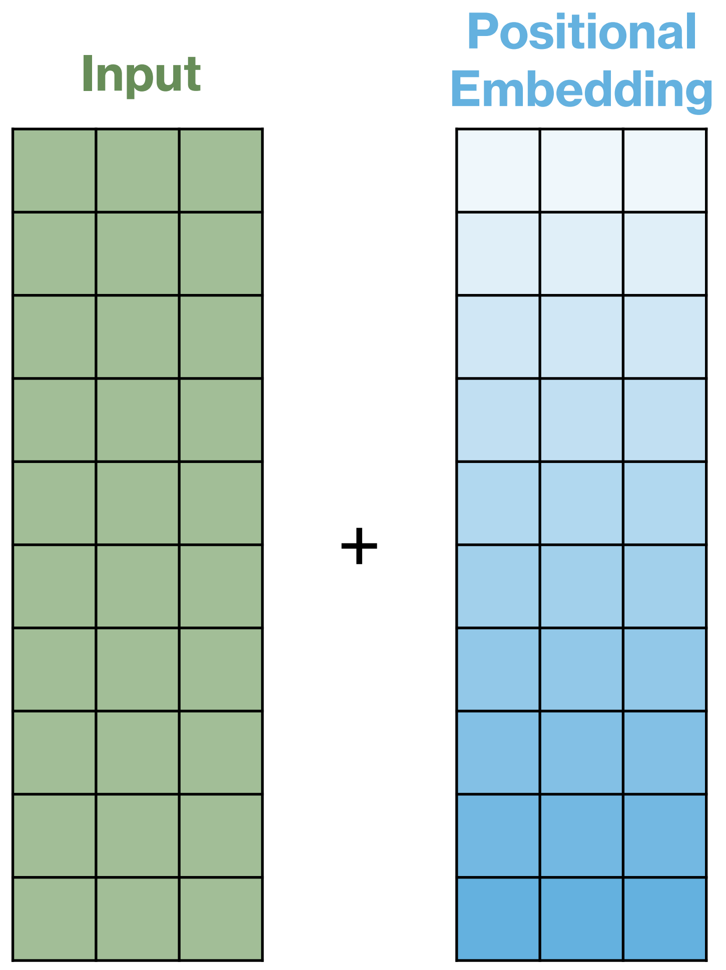
In **absolute positional embedding** we create a unique vector for every position (1, 2, 3, ...) and add it directly to the token's embeddings.

It is usually implemented with Sine and Cosine functions of different frequencies so the model can learn relative distances easily, like:

$$PE_{(pos,\,2i)} = \sin\left(\frac{pos}{K^{2i/d_{model}}}\right)$$

$$PE_{(pos,\,2i+1)} = cos\left(\frac{pos}{K^{2i/d_{model}}}\right)$$

Where i is the dimension index, i.e., the column, and pos is the position of the token, i.e., the row



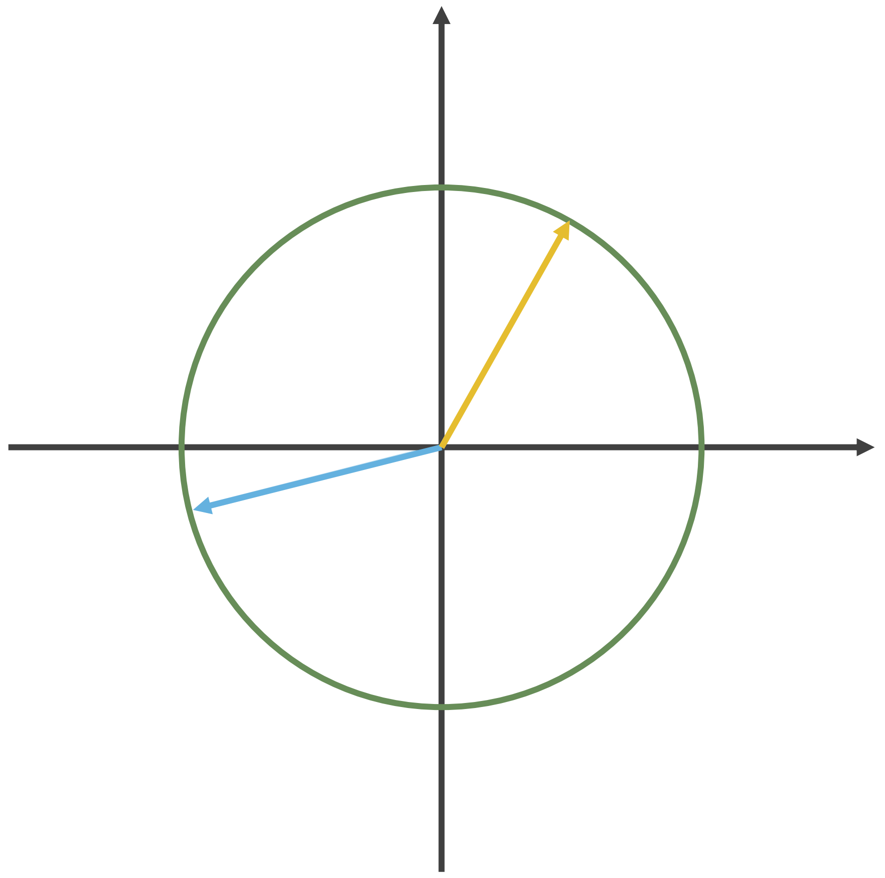
Positional embedding: RoPE

RoPE is the acronym for **Rotary Positional Embeddings**; it is the state-of-the-art technique used in modern LLMs such as GPT.

It applies a rotation matrix directly to the Query and Key vectors during the attention calculation, so for two positions m and n we get:

$$q_m = R_m \cdot q$$

$$k_n = R_n \cdot k$$



Positional embedding: RoPE

$$q_m = R_m \cdot q$$

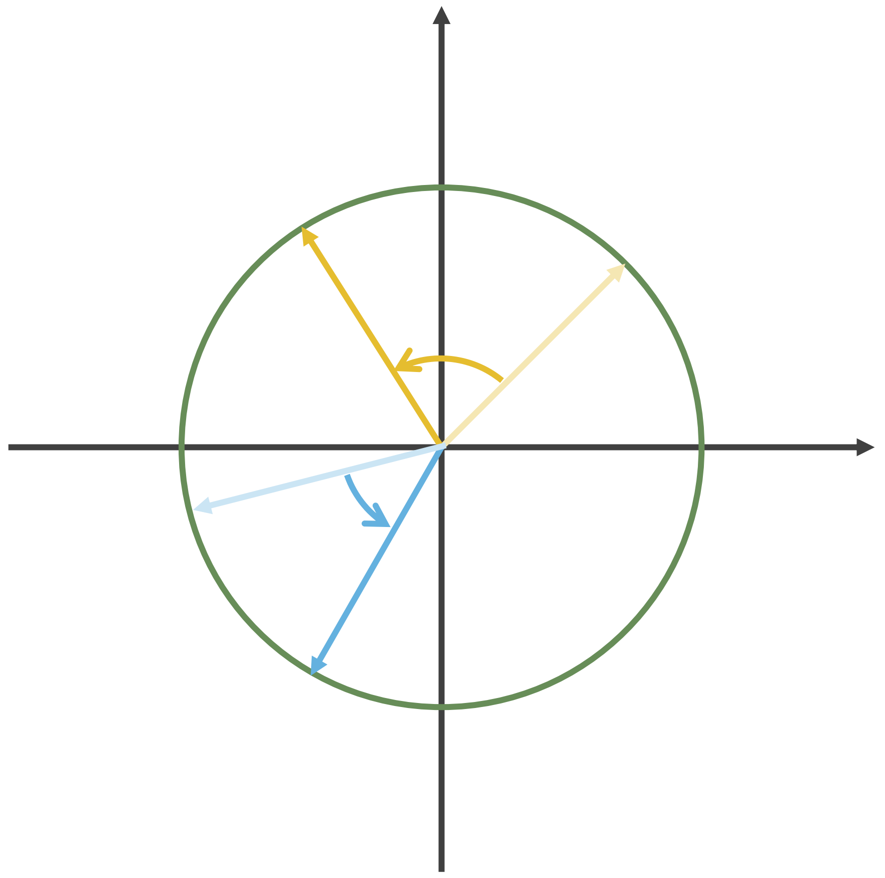
$$k_n = R_n \cdot k$$

Because of the properties of rotation matrices, the dot product between a Query at position m and a Key at position n only cares about the **difference** between their angles:

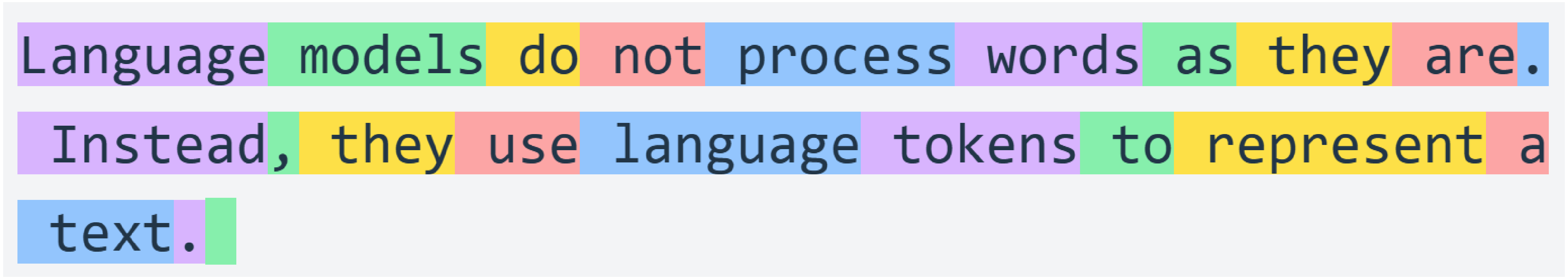
$$\langle R_m q, R_n k \rangle = q^T R_{n-m} k$$

The attention score naturally depends on **relative distance** $(n - m)$.

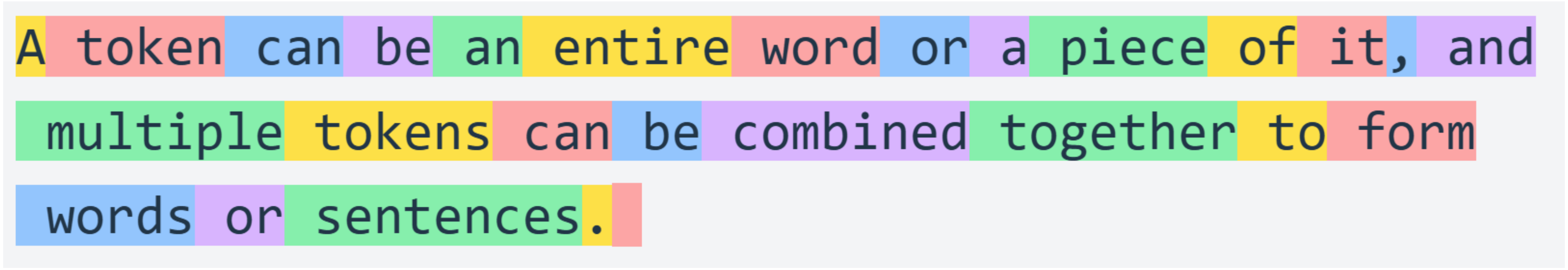
If two words are close together, their vectors are "rotated" to a similar degree, resulting in a higher dot product.



Large language models: Tokenization

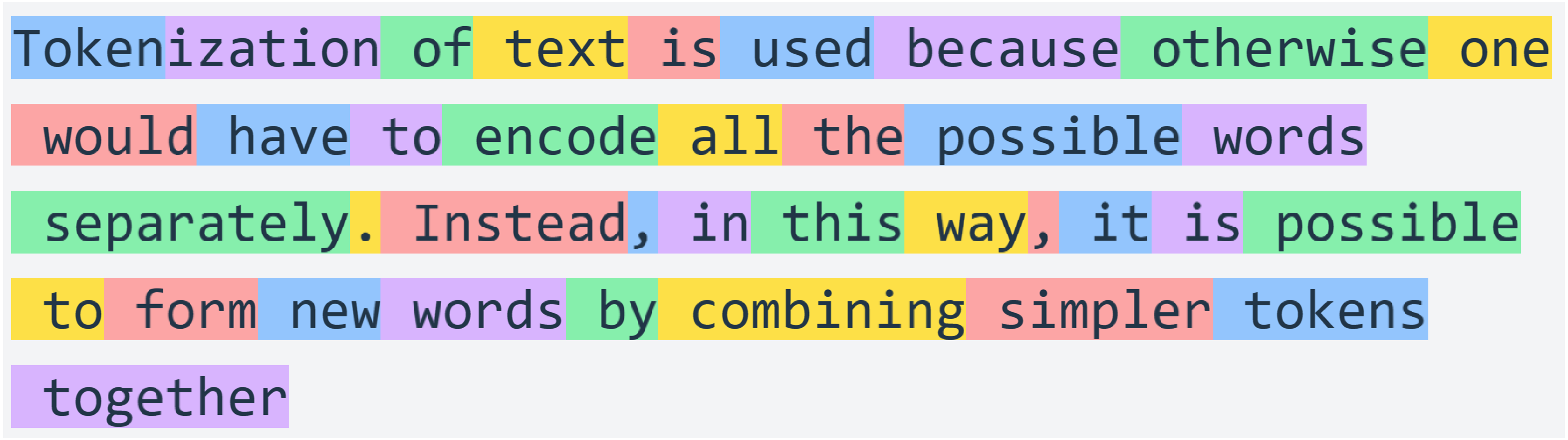


Language models do not process words as they are. Instead, they use language tokens to represent a text.

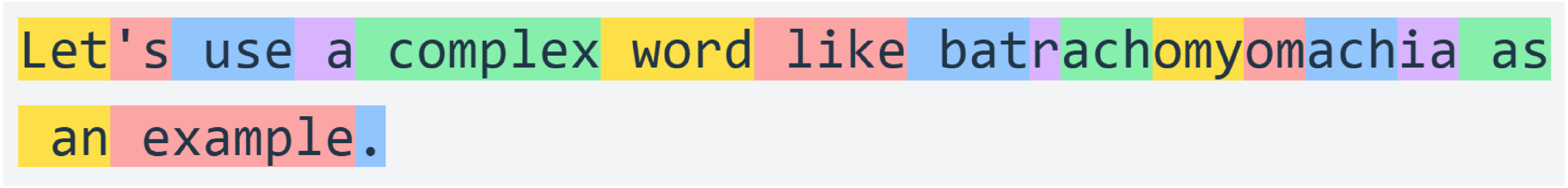


A token can be an entire word or a piece of it, and multiple tokens can be combined together to form words or sentences.

Large language models: Tokenization



Tokenization of text is used because otherwise one would have to encode all the possible words separately. Instead, in this way, it is possible to form new words by combining simpler tokens together



Let's use a complex word like batrachomyomachia as an example.

Large language models: Tokenization

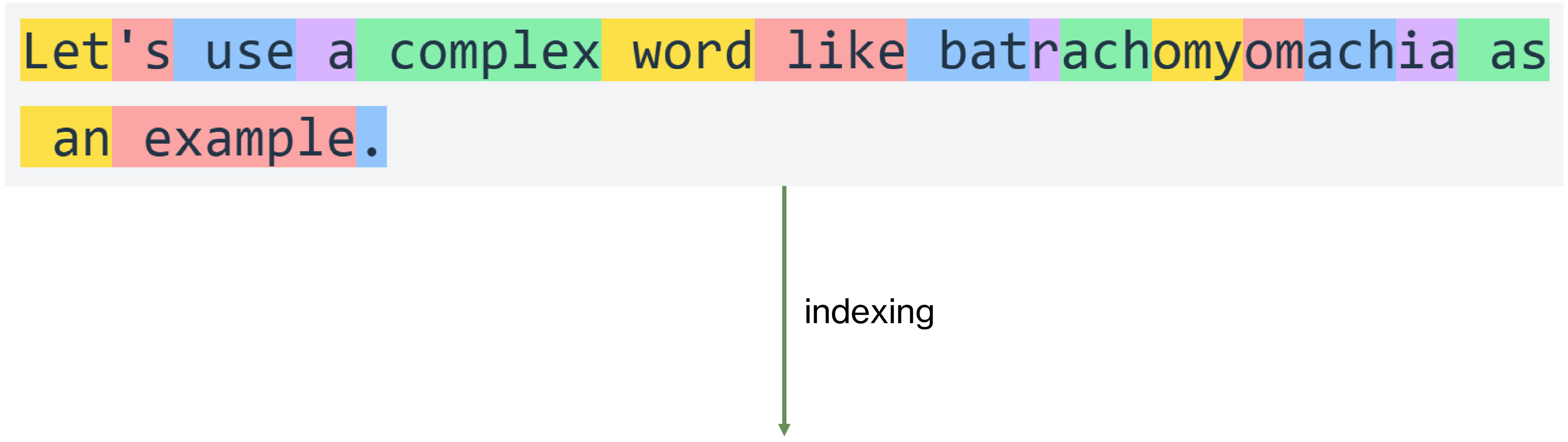
The **tokenization** strategy depends on the specific algorithm, so different models use different tokenizers. Typically, *the tokenizer learns the optimal tokenization* based on some heuristics.

The complete set of tokens that a model can access is known as the **vocabulary**.

The tokens are then **indexed**, i. e., they are assigned a unique numerical identifier.

Tokenization: text to math

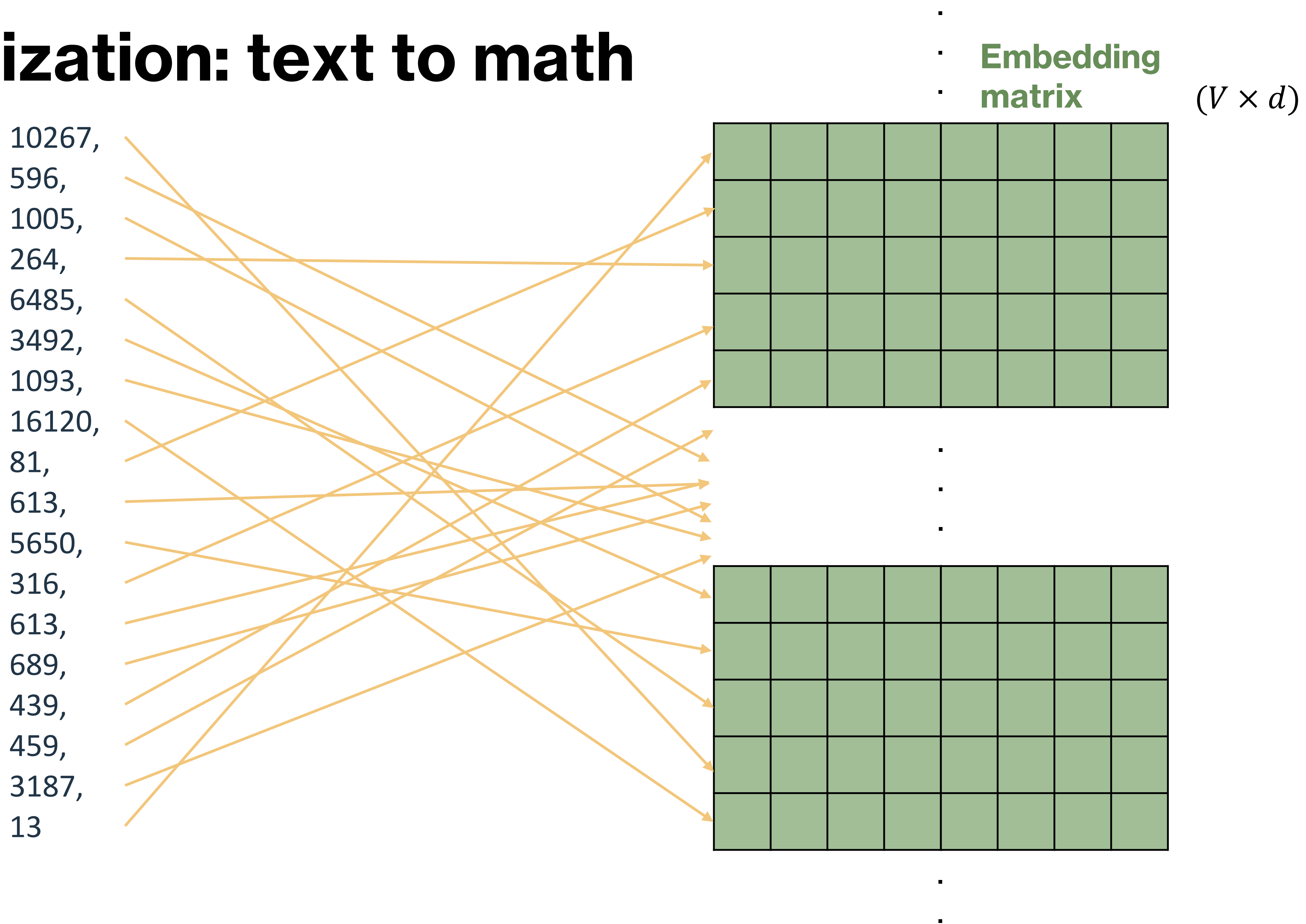
Let’s walk through the process transforming pure text into something the model can manage: numbers!



Let's use a complex word like batrachomyomachia as an example.

[10267, 596, 1005, 264, 6485, 3492, 1093, 16120, 81, 613, 5650, 316, 613, 689, 439, 459, 3187, 13]

Tokenization: text to math



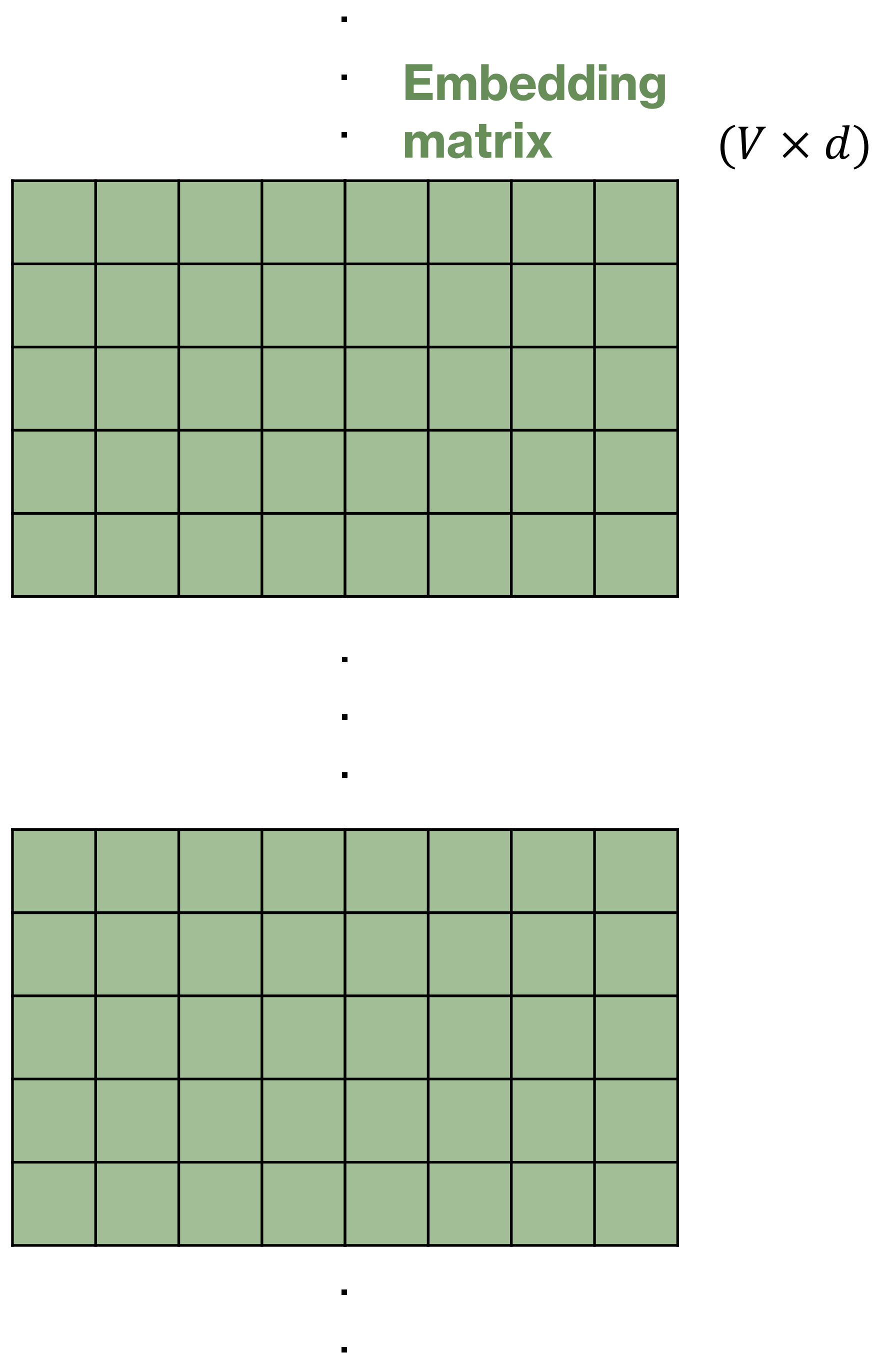
Tokenization: text to math

LLMs cannot read text as it is, so, after indexing, tokens are **embedded** into vectors using a **learnable embedding matrix**.

It is like a dictionary translating words into math.

Embedding vectors all have the same shape, like (1 × 784), so all the tokens, no matter the length, are mapped to the same dimensional space.

The embedding matrix is huge! It has $n_{tokens} \times n_{dimensions}$ weights, usually tens or hundreds of millions.



Modeling language with attention: BERT

BERT, acronym for Bidirectional Encoder Representations from Transformers was one of the first and most famous architectures leveraging attention to solve language tasks.

Contrarily to the original Transformer, BERT was an **encoder-only** model.

The objective of BERT was not to generate text, but to understand the context of every token in the sentence in order to learn meaningful representations of provided text

BERT: the training strategy

BERT was trained using a `[MASK]` called **masked language modeling**.

It is `[MASK]` simple in principle. Essentially, we give a sentence to the `[MASK]` and mask out some tokens, then we ask the model `[MASK]` predict them.

In the `[MASK]` work, 15% of the `[MASK]` in a sentence were masked and used to calculate `[MASK]` model prediction loss.

They were either masked with a special `[MASK]` token or replaced with a random word.

BERT: the training strategy

BERT was trained using a technique called **masked language modeling**.

It is quite simple in principle. Essentially, we give a sentence to the model and mask out some tokens, then we ask the model to predict them.

In the original work, 15% of the tokens in a sentence were masked and used to calculate the model prediction loss.

They were either masked with a special `[MASK]` token or replaced with a random word.

BERT

The general-purpose training strategy made BERT very versatile for downstream tasks like question answering or classification.

The bi-directional nature makes BERT quite powerful but also makes it not suitable for generative purposes.

BERT sequence length was limited to 512 tokens, which is quite short for today's generative models, that can read entire books.

Generative language models

Language models can be trained to solve a wide variety of tasks, such as summarization, classification, translation, and so on.

By far, the most popular kind of language models are the **generative language models**.

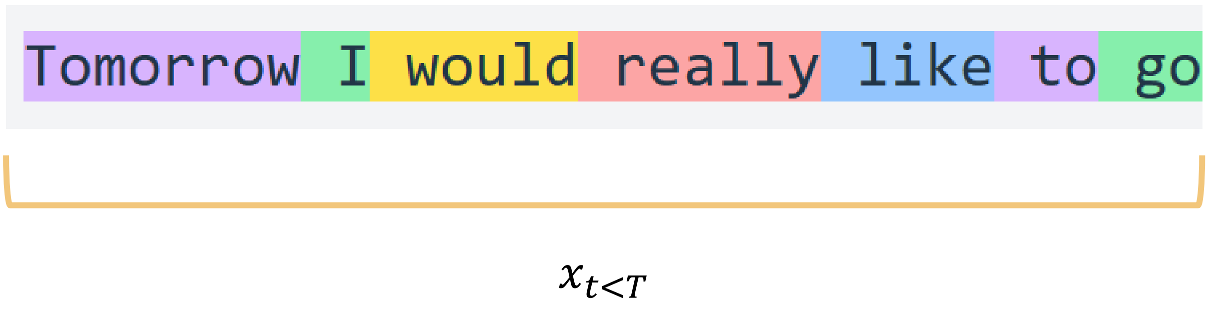
They are trained to predict *the next most probable* **token** given a list of previous tokens.

Mathematically speaking, ChatGPT is:

$$P(x_T | x_{t<T})$$

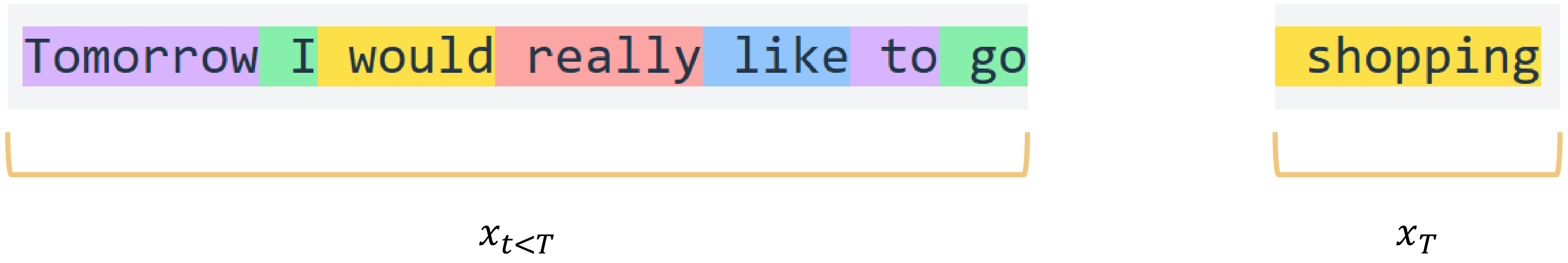
Next token prediction: Training

We are finally ready to unveil how LLMs do their magic. Let’s first define the training objective. We said a generative LLMs has to predict token x_T given all the tokens $x_{t<T}$, for instance:



Next token prediction: Training

We are finally ready to unveil how LLMs do their magic. Let's first define the training objective. We said a generative LLMs has to predict token x_T given all the tokens $x_{t<T}$, for instance:



Next token prediction: Training

Mathematically, what the model does is predicting a **probability distribution over the whole vocabulary**.

It means that, given the input sequence, the model associates a probability score to every possible token in its vocabulary, then, the token with the highest probability is selected as the *next one*.

We know which token is the correct answer, so we can use it to train the model.
If model error is big, its weights are strongly modified, otherwise they are not.

Next token prediction: an ill posed problem

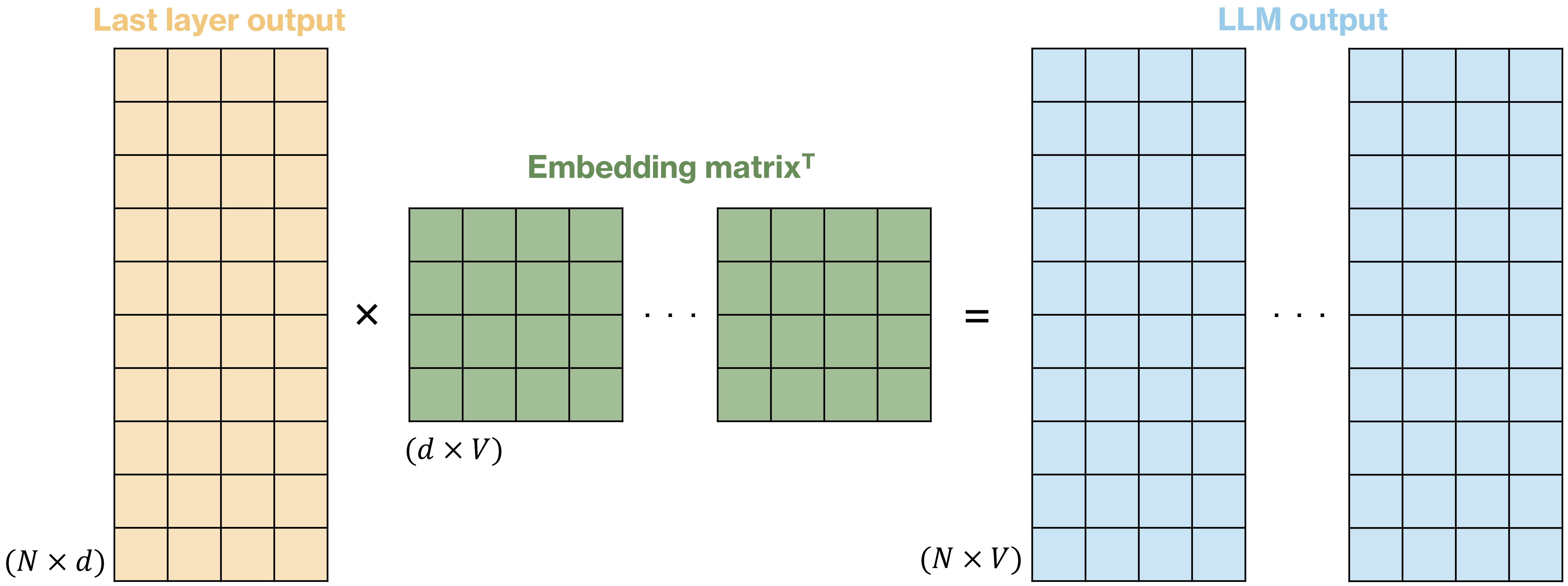
There are **a lot** of possible correct *next tokens.* In our example, *shopping*, could be substituted with *dancing*, *skiing, out…* and they would all be correct continuations for the initial phrase.

However, there are tokens more correct than others. By reading billions of phrases, the model learns a probability distribution that **mimics** incredibly well the concept of *talking*

The model implicitly learns relations between words, grammar, semantics. It *indirectly learns a language*.

The output layer

A special mention goes to the output layer. It uses *the same embedding matrix* we saw before

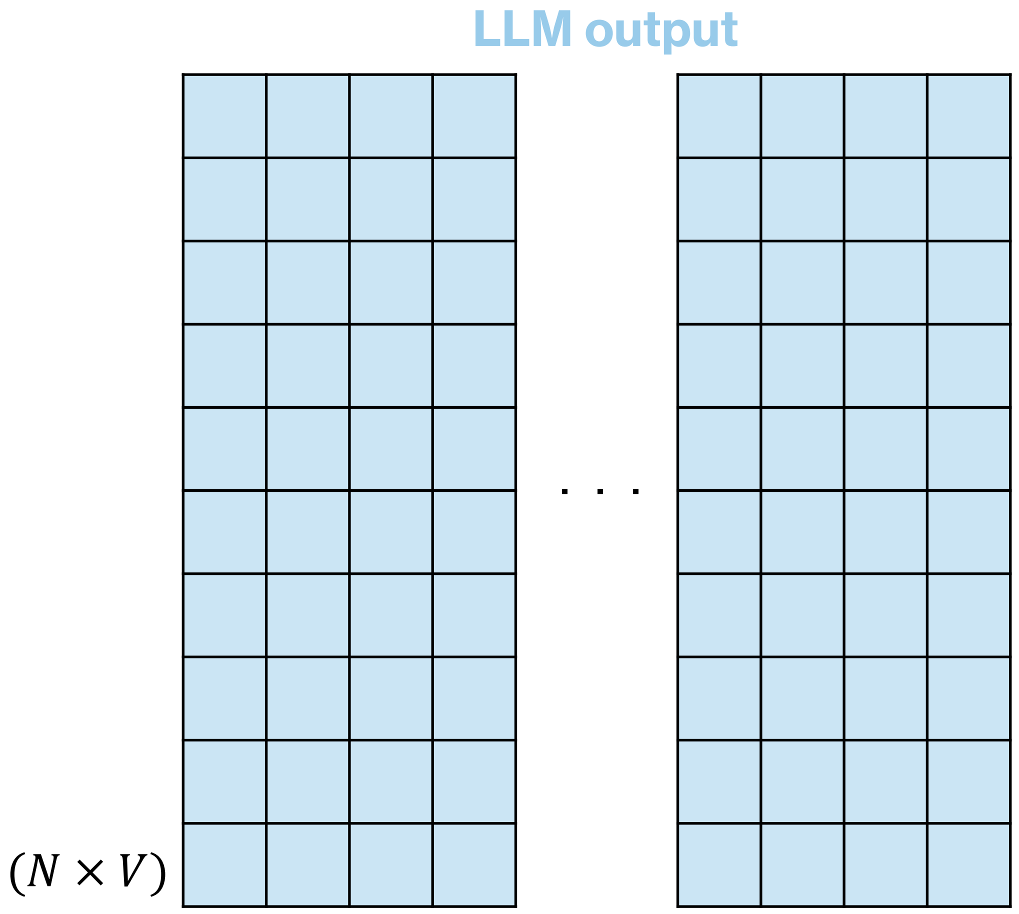


The output layer: Training

The network predicts a probability distribution of the next token *for each token in the sequence*!

This is useful in training because each token of the sequence becomes a learning example.

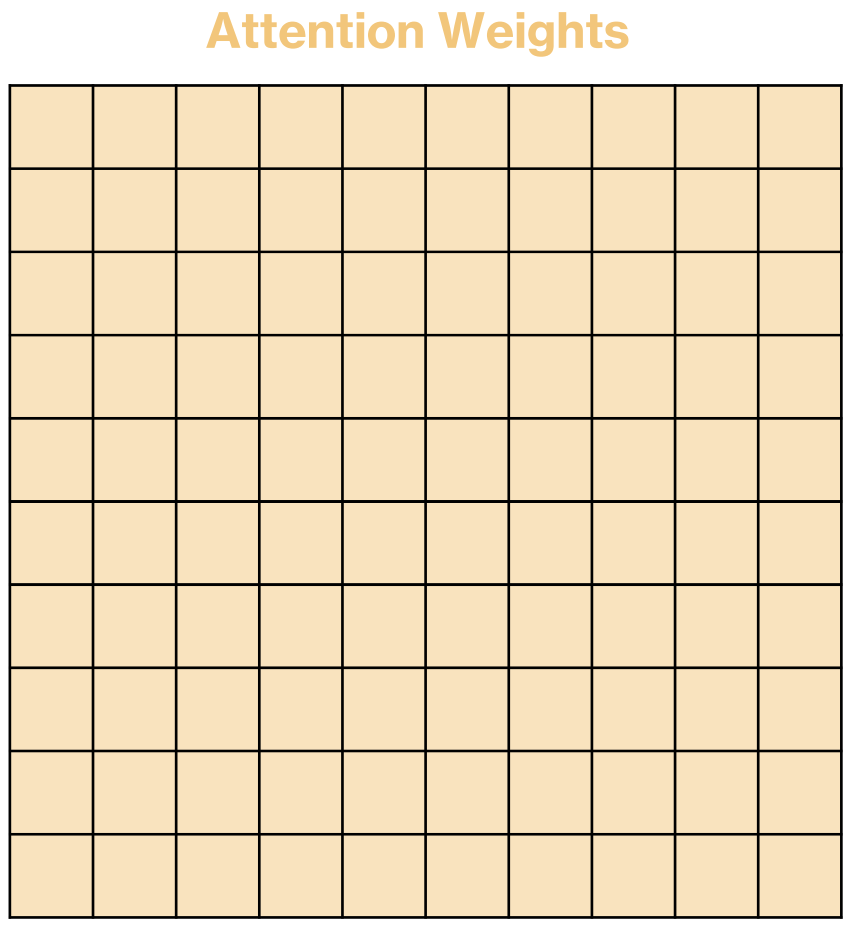
Here N is the number of tokens in the input sequence, while V is the number of tokens in the vocabulary.



Training: causal attention

The main difference between encoder-only and decoder-only models is in the way they use attention.

BERT for instance can attend to the entire sequence, since it is used as an encoder model.



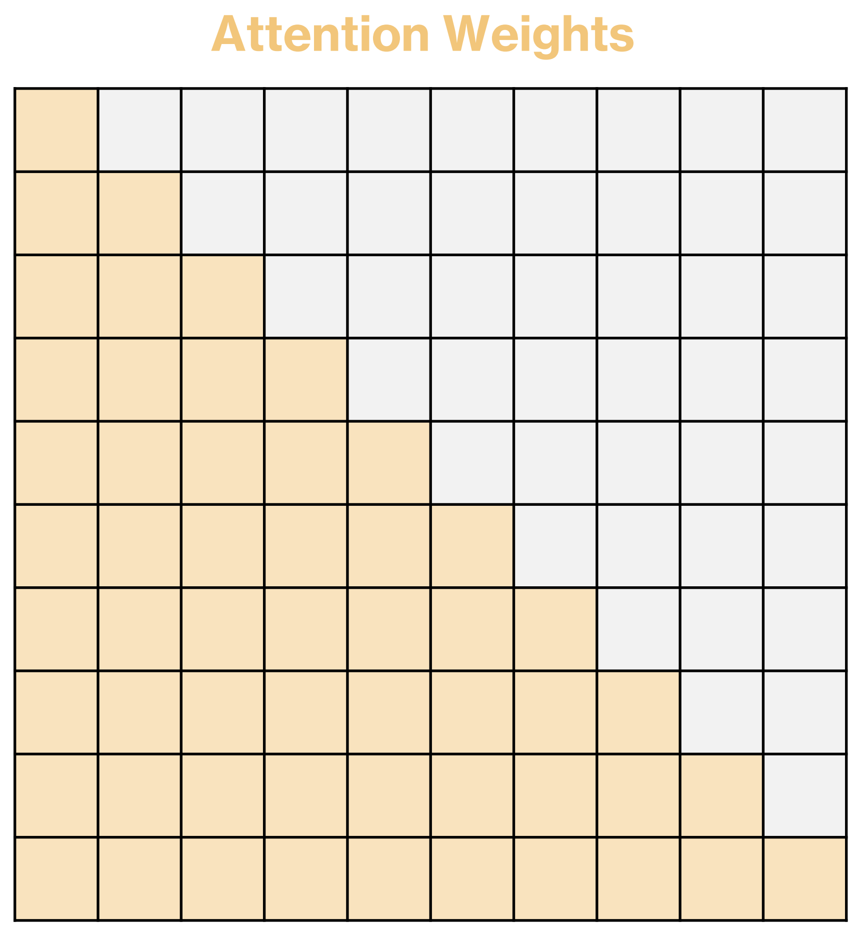
Training: causal attention

The main difference between encoder-only and decoder-only models is in the way they use attention.

BERT for instance can attend to the entire sequence, since it is used as an encoder model.

On the other end, generative models like GPT must use causal attention, which masks out the attention calculation for the tokens after the current token x_T.

This is necessary, otherwise the network would have information about the tokens it should generate.



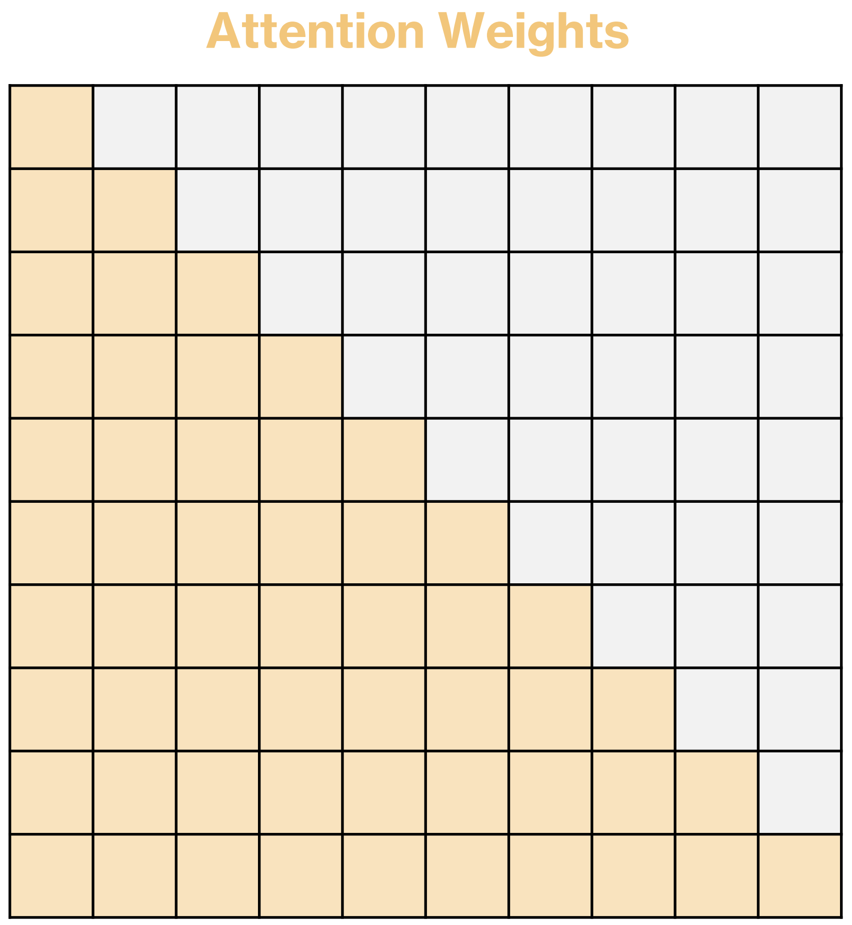
Training: causal attention

The main difference between encoder-only and decoder-only models is in the way they use attention.

BERT for instance can attend to the entire sequence, since it is used as an encoder model.

On the other end, generative models like GPT must use causal attention, which masks out the attention calculation for the tokens after the current token x_T.

This is necessary, otherwise the network would have information about the tokens it should generate.



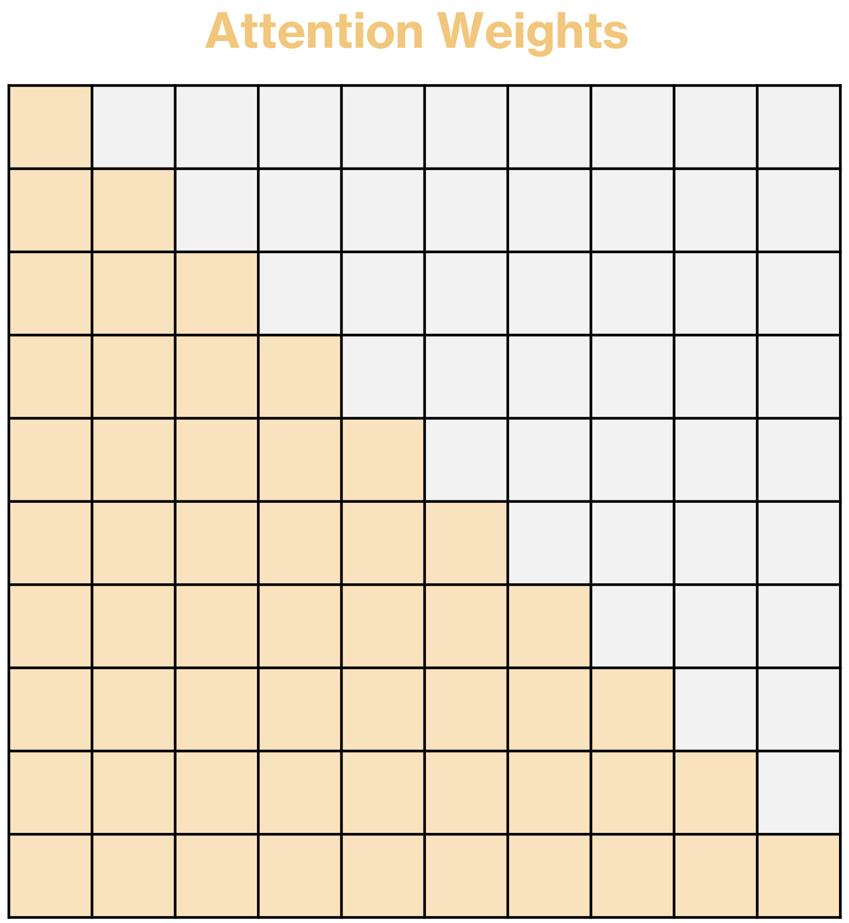
Encoder or decoder: this is the question

We said BERT is an encoder-only model while GPT is a decoder only.

It could be confusing, but the difference is not in the architecture.

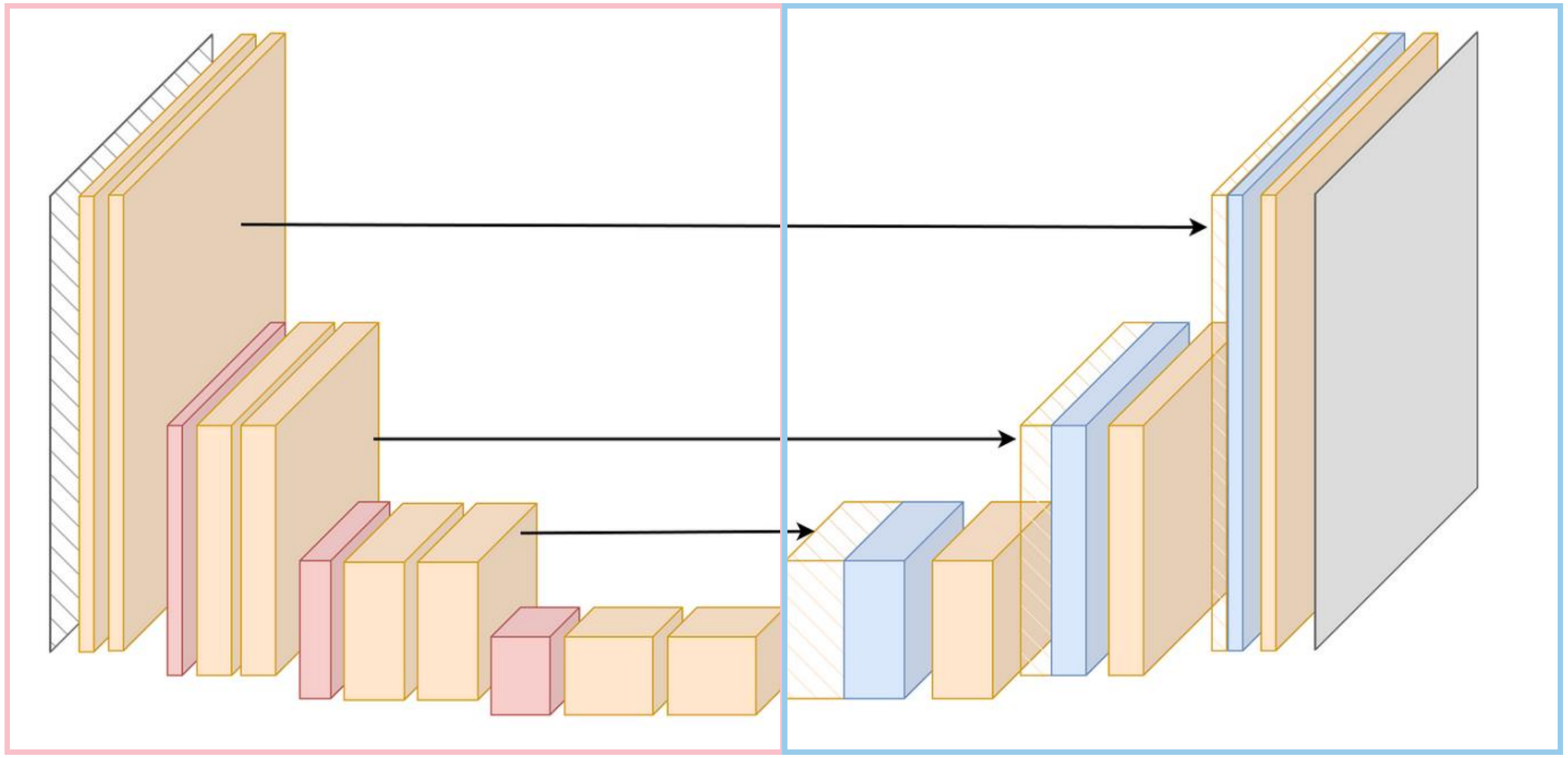
At the end of the day, they use the same transformer blocks, so what's the difference?

The difference is just in the training objective and in the use of a causal attention matrix for decoder-only models



Encoder or decoder: this is the question

This is in contrast with the common definition of encoder and decoder we find in computer vision, where it indicates an actual architectural difference.

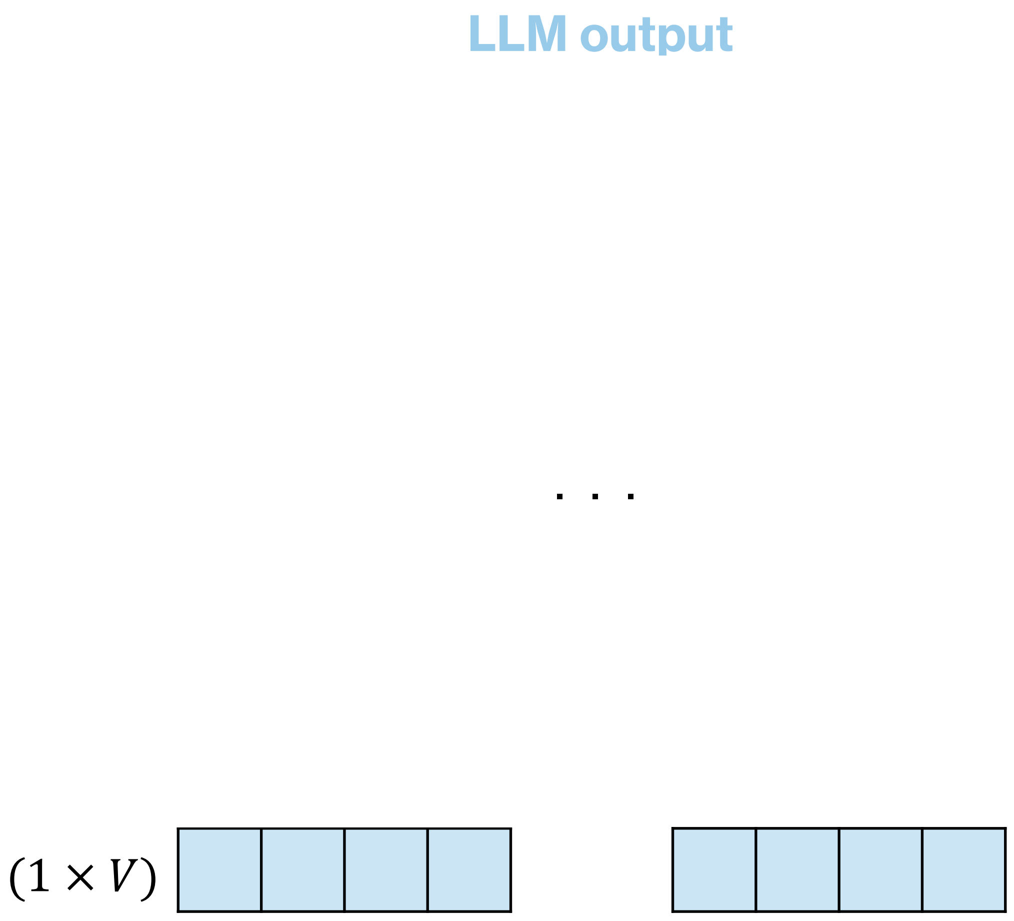


The output layer: Inference

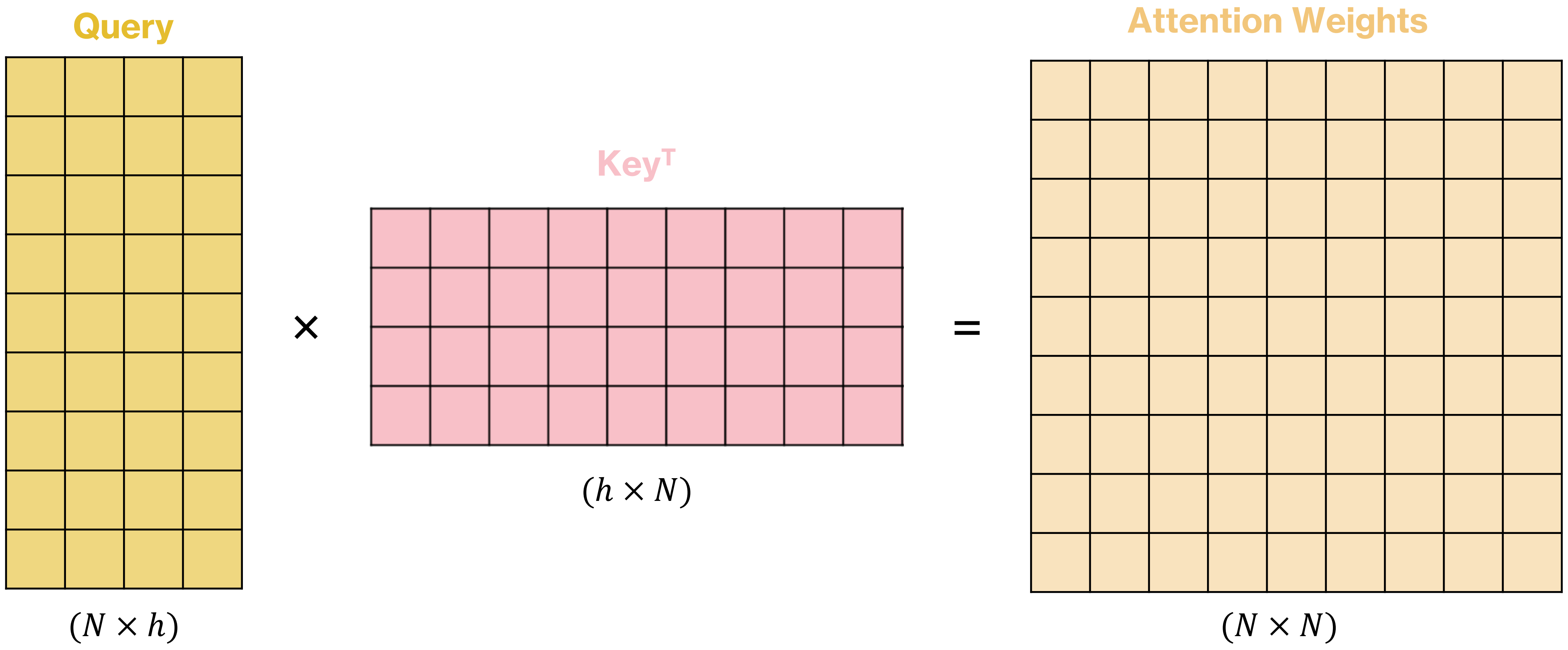
During inference, we are not interested in the halfway predictions, so we just take the last row of the output matrix to select the token.

To continue the prediction, we append the predicted token to the initial sequence, and give it back in input to the model, so it can keep *talking*.

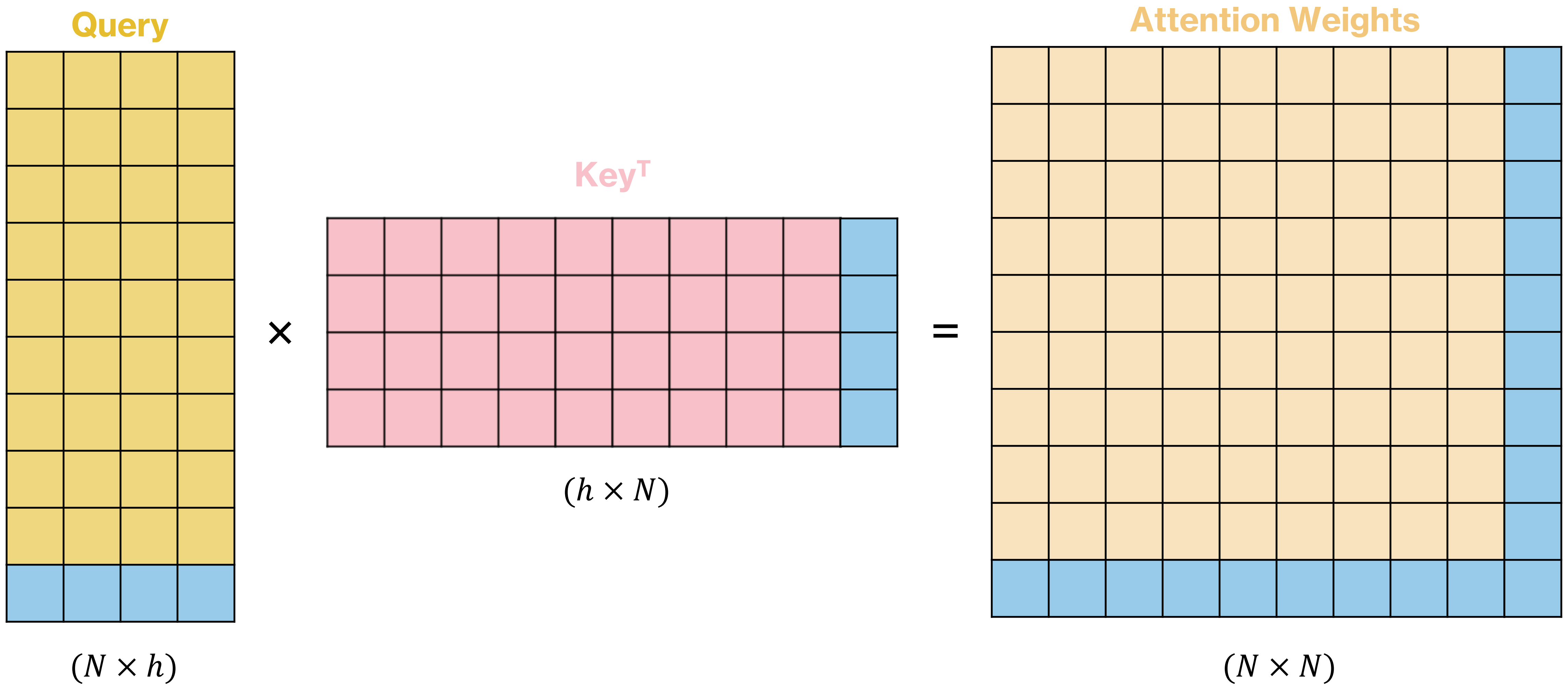
This prediction method is known as **auto-regressive** generation, since each subsequent output is re-used as input to generate the next one.



KV caching


Query
Key^{T}
Attention Weights
$\times$
$=$
$(N \times h)$
$(h \times N)$
$(N \times N)$

KV caching

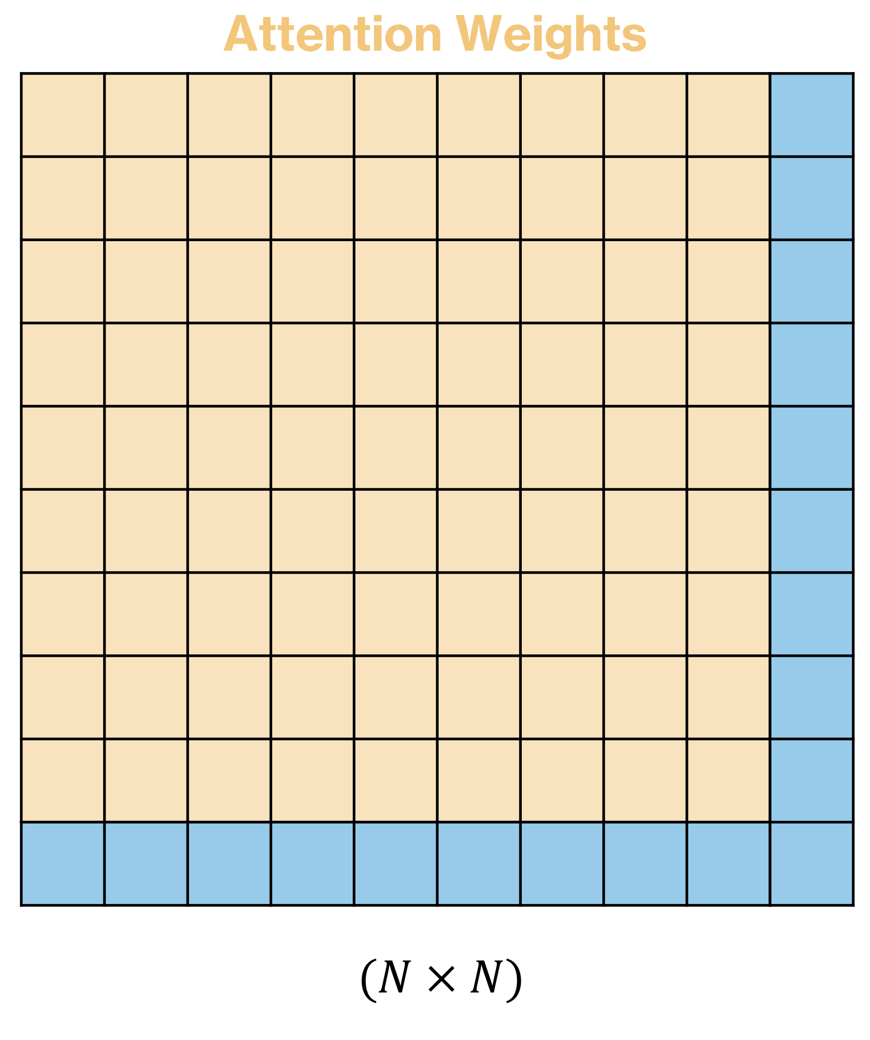

Query
$\times$
Key^T
=
Attention Weights
$(N \times h)$
$(h \times N)$
$(N \times N)$

KV caching

Since the attention weights stay the same for old tokens, we can cache them to avoid the expensive attention calculation.

We only have to calculate the attention for the new tokens we append during the auto-regressive process.

That's why a key metric is **time-to-first-token**; it represents the *Prefill* phase where we process the entire input prompt at once to initialize the cache. Afterward, we only compute attention for one new token at a time.



Some considerations

Given the nature of auto-regressive generation, the model could keep talking forever.
There is a **special token** the model can output to stop the generative process.

For an LLM it is not trivial at all to determine **how long is a word** in terms of characters, since it operates on the token level, and *in the mathematical space, each token has the same dimension*.

There's no real **memory**. The *illusion of memory* comes from the fact that the input prompt includes the whole conversation so far, and the model reads it all at each generation step.

Mixture of Experts (MoE)

Mixture of Experts are a class of LLMs in which only a subset of the total weights is used for each input token.

The concept itself is not new in the field of machine learning. Many successful models leverage ensembles of multiple predictors to produce more robust results. Among the most famous we have **random forests** and **gradient boosting** models.

Essentially, in an MoE layer, we have a list of MLPs, the experts, and a **router** that directs the input to the relative expert.

Mixture of Experts (MoE)

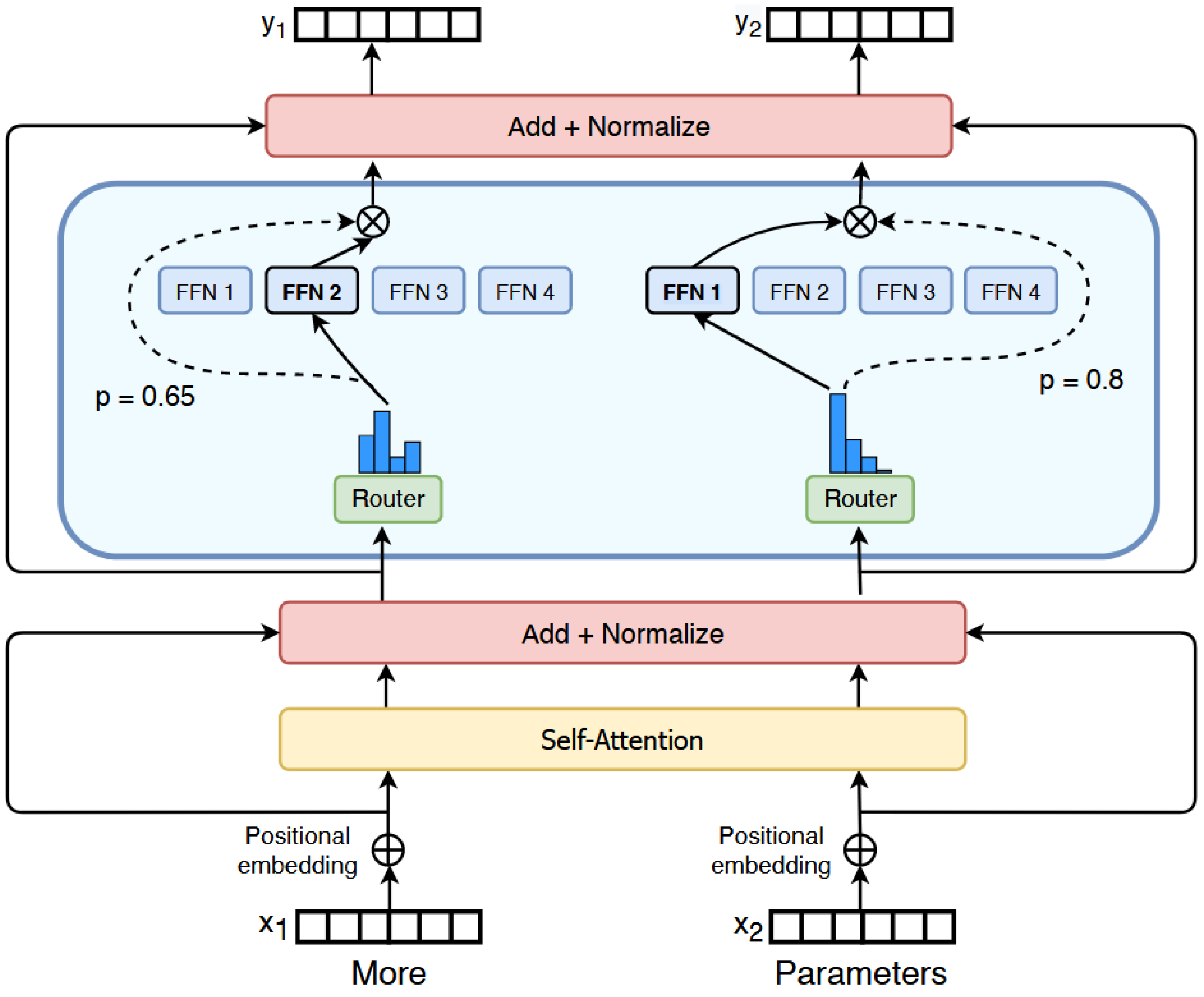
The router **independently** routes each token using a probability function, then the output of the selected MLP is multiplied by the router probability value.

$$h(x) = W_r x$$

$$p_i(x) = \frac{e^{h(x)_i}}{\sum_{j=1}^{K} e^{h(x)_j}}$$

$$y = \sum_{i \in \Gamma} p_i(x) E_i(x)$$

Where $h(x)$ is the router function, $E_i(x)$ is the expert function, and Γ is the list of selected experts.

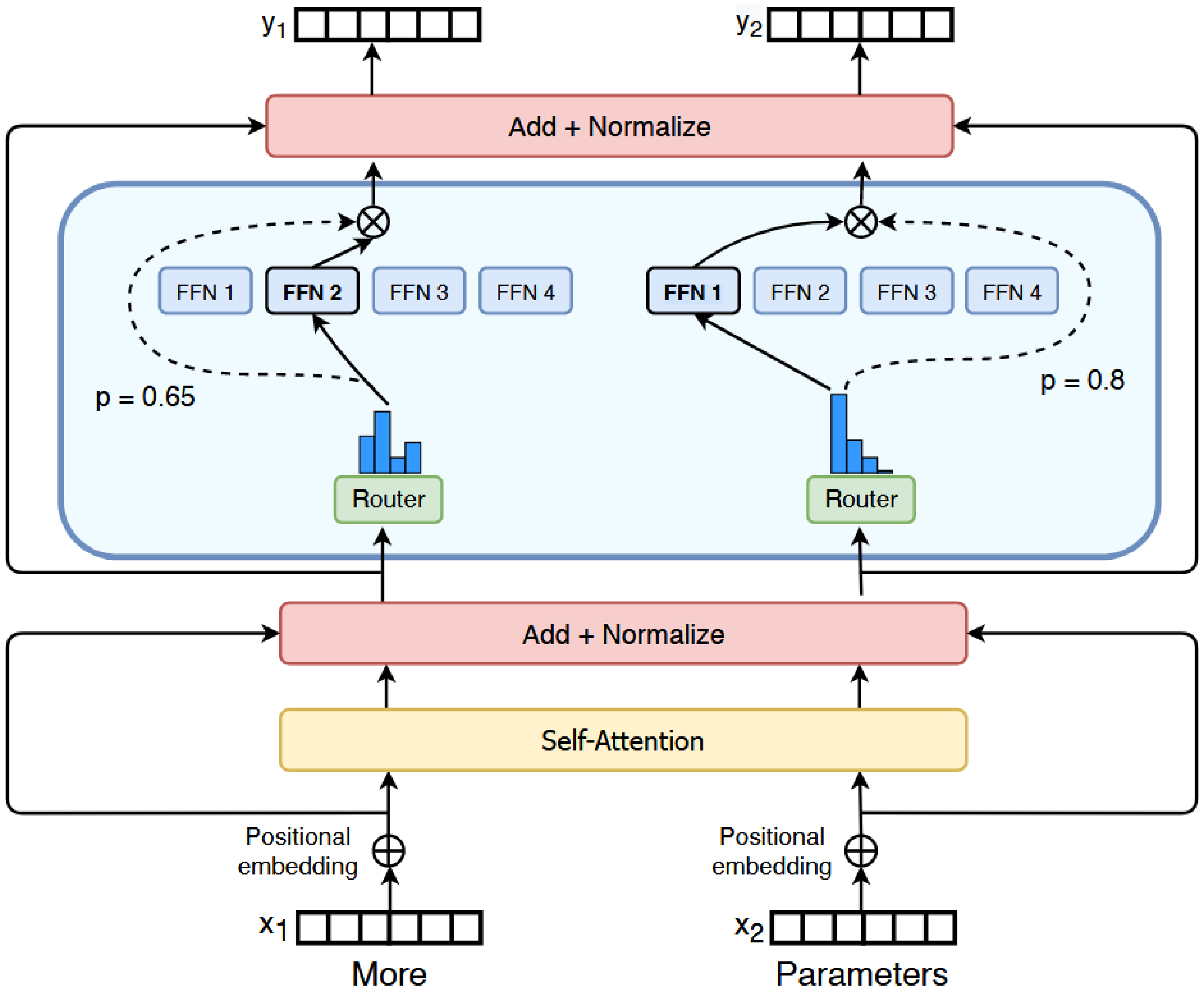


Mixture of Experts (MoE)

The top-k experts can be selected, based on the router probability, so if we decide to select more that one expert, the final output will be a weighted sum of their output.

The problem with routing is that NNs are lazy so, if unrestricted, *the router would probably converge to route each input to the same expert*.

An **auxiliary loss** in thus used to force the router to balance the load among experts



Mixture of Experts (MoE)

Auxiliary loss:

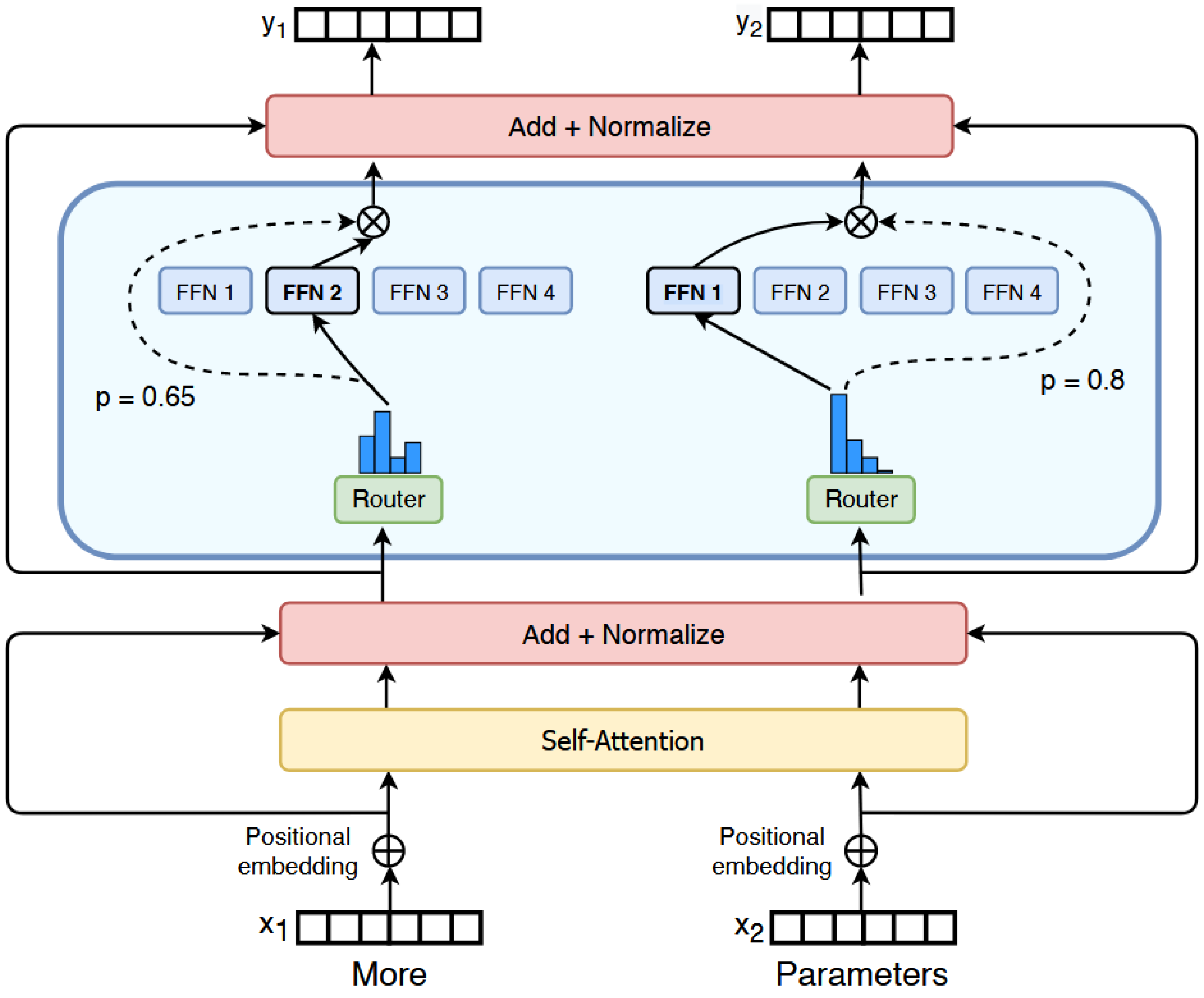
$$\mathcal{L} = \alpha \cdot N \cdot \sum_{i=1}^{N} f_i P_i$$

Where

$$f_i = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathbb{1}\{argmax\ p\_i\ (x) = i\}$$

And

$$P_i = \frac{1}{T} \sum_{x \in \mathcal{B}} p_i(x)$$



Mixture of Experts (MoE)

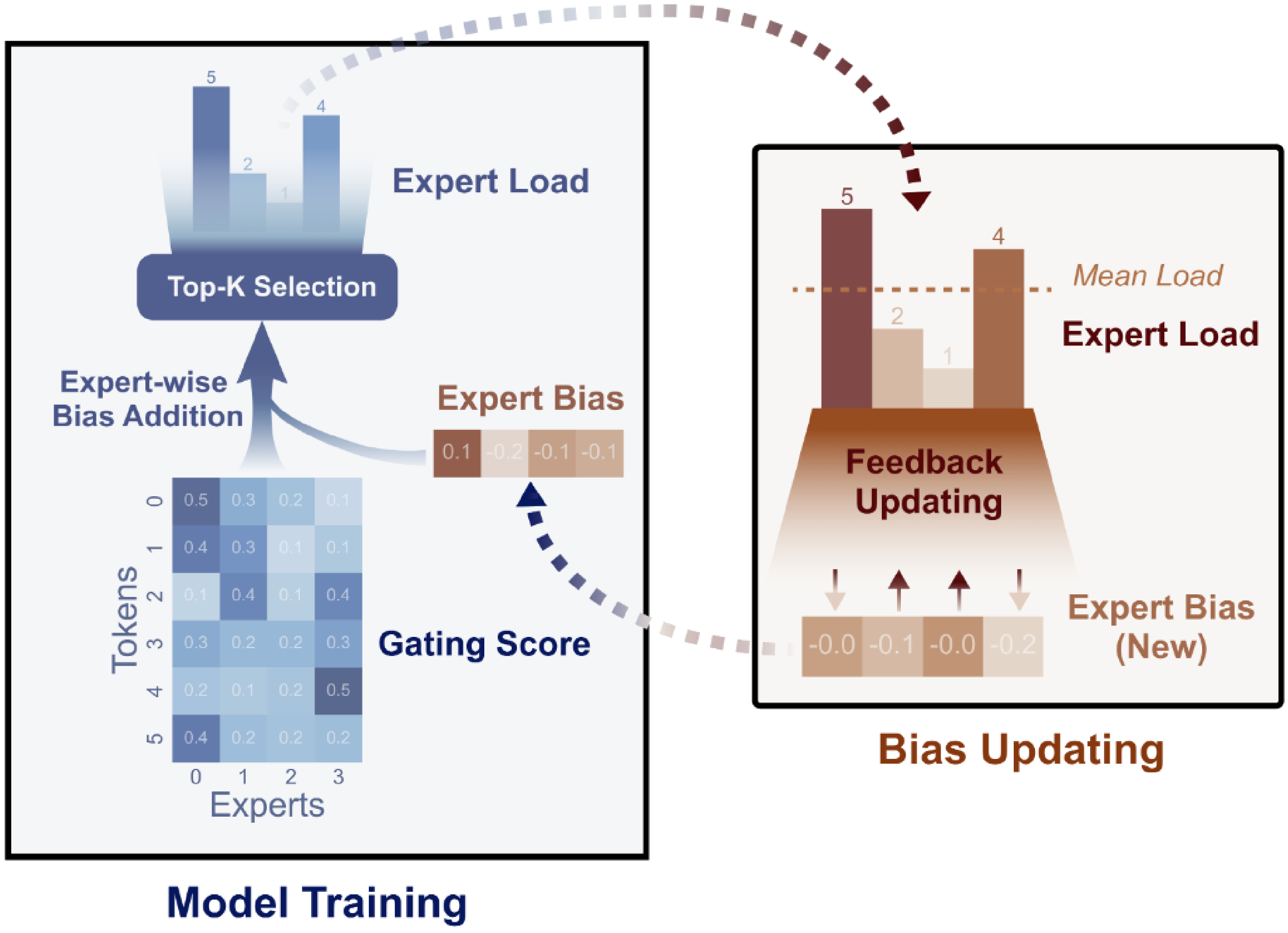
Loss-free routers have also been proposed, for example, by adding a bias to the gating score.
At each step, the bias is updated according to:

$$b_i = b_i + u \cdot sign(e_i)$$

Where

$$e_i = \overline{c_i} - c_i$$

Here c_i is the number of tokens assigned to each expert and $\overline{c_i}$ is the average number.



Mixture of Experts (MoE)

MoEs allow to build incredibly larger models while keeping the computational burden low, since the number of **active experts** is fixed

Experts can be **split** on different devices so model weights can grow with number of devices while maintaining manageable memory and computational footprint on the single device.

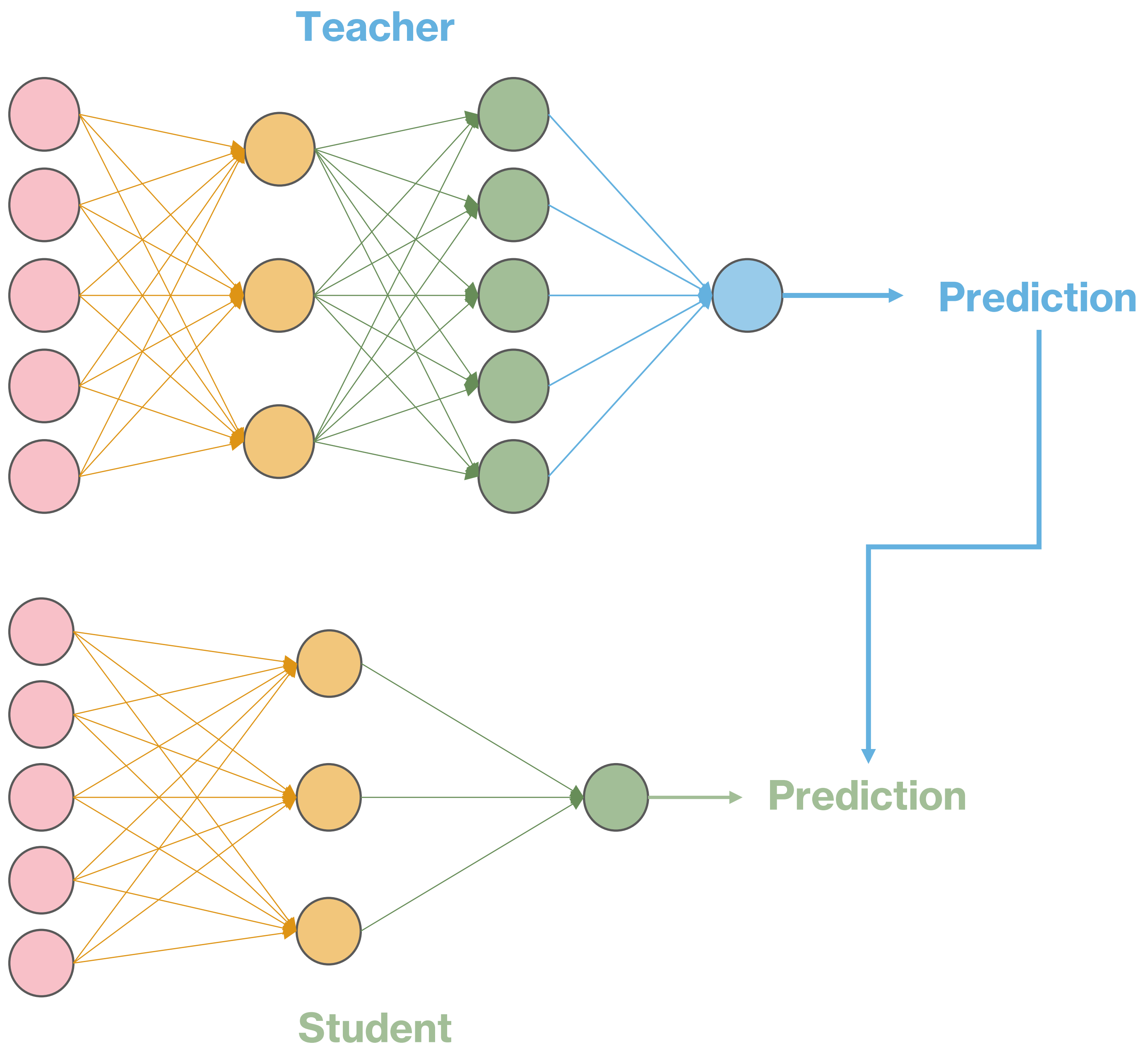
MoEs are significantly more **compute-efficient** during training and are also quite faster in inference compared to a dense model with the same amount of parameters

A note on model distillation

Distillation is the process of transferring knowledge from a large model to a much smaller model. The two models are called **Teacher** and **Student**

The Student learns the **Soft Targets** from the Teacher.

The goal is to achieve similar performance to the Teacher but with a fraction of the parameters, memory, and latency



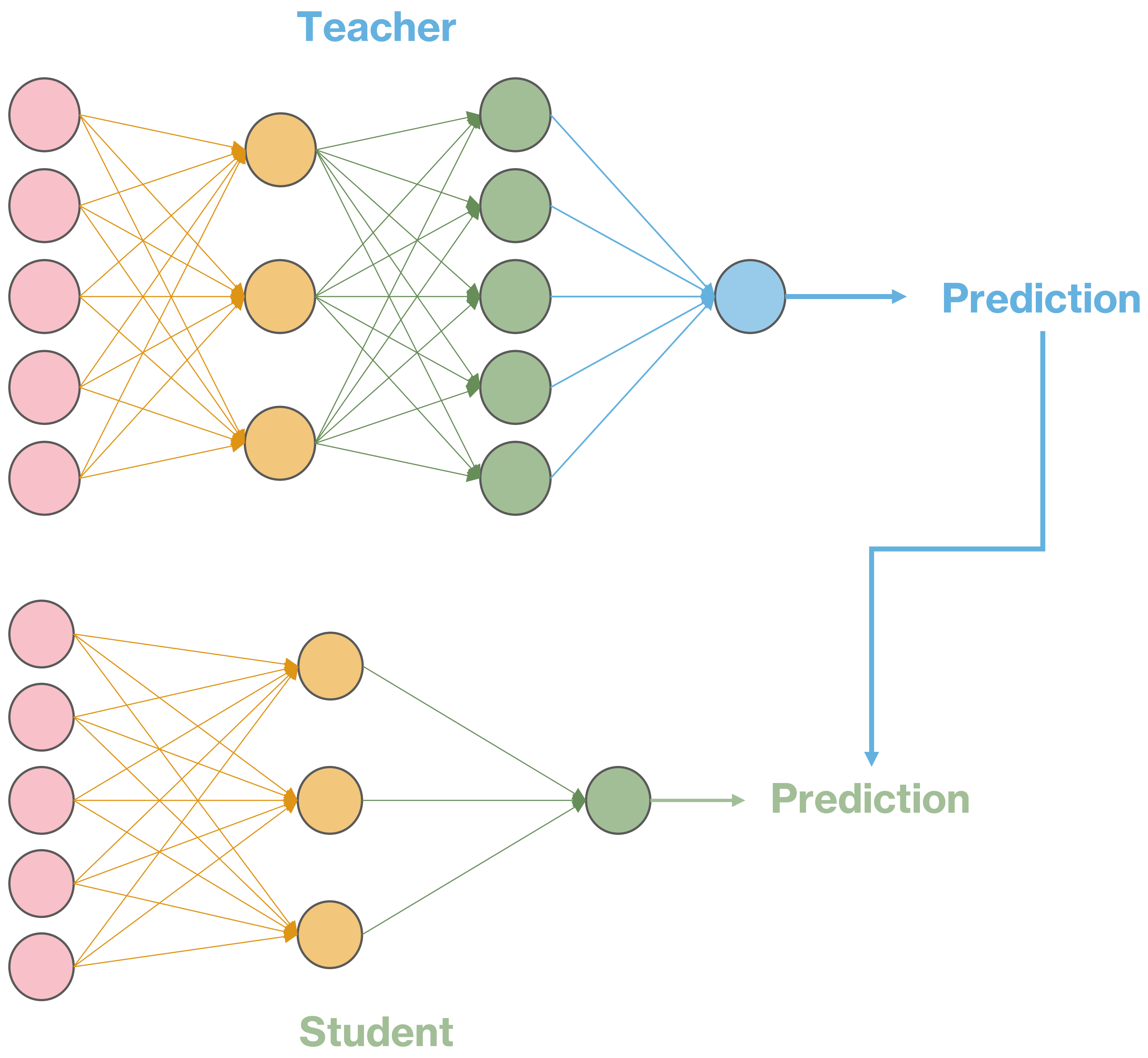
A note on model distillation

We usually have two loss components, a **distillation loss**, being the KL divergence between the (softened) probability outputs of the models:

$$q_i = \frac{\exp(z_i/T)}{\sum_j \exp\left(z_j/T\right)}$$

And a **student loss**, where the student tries to predict the actual ground truth. The final loss becomes:

$$\mathcal{L} = \alpha\mathcal{L}_{distill} + (1-\alpha)\mathcal{L}_{student}$$



Conclusions

LLMs share the same base mechanics of all the other NNs. They are essentially **matrix multiplication** and **backpropagation**.

Attention allows transformer models to focus on important relations among the input and recognize arbitrarily complex patterns.

The underlying similarities allow the **combination** of multiple modalities and architectures to build incredibly powerful models.

Conclusions

A famous paper by Apple was titled **the illusion of thinking**. Do you think LLMs decisional process can be considered *thinking*?

LLMs are trained in such a way that they **must answer** questions, it is very difficult for an LLM to say *I don't know*.

The statistical nature of LLMs favors the **average** against the **outlier**. Try asking GPT to generate an analog clock showing 15:37 for example.

MINERVA

Thank You for the attention

Gianluca Carlini
Cineca

Bologna, 24/03/2026