Laurea Magistrale in Informatica

Dipartimento di Informatica

UNIVERSITÀ DI TORINO

Transformers

Course

Neural Networks and Deep Learning

Professor

Roberto Esposito (roberto.esposito@unito.it)

These slides are based on Chapter 12 of: C. Bishop, H. Bishop, *Springer*, Deep Learning - Foundations and Concepts

Image dreamed by stable diffusion

Prompt: "head shot of optimus prime transformer robot"

The main idea behind Large Language Models is to have a **large** model, trained on a **large** dataset, able to **predict the next token in a sequence**.

Until the introduction of **transformers**, the most common approach to process sequences using neural networks was to use recurrent neural networks (RNNs) or long short-term memory (LSTM) networks.

However, these models have some limitations:

- they are **sequential** and cannot be easily parallelized;
- they have difficulties in capturing **long-range dependencies**;

Transformers represent one of the most important developments in deep learning.

They have been introduced in the seminal paper **Attention is all you need** by Vaswani et al. in 2017, and have since then become the de-facto standard for many deep learning tasks.

They are at the core of **Large Language Models**, such as GPT-4o, and have been used in many other applications, such as image generation, reinforcement learning, and more.

They are suitable for building large general-purpose models (**foundation models**) that can be fine-tuned for various tasks.

Foundation models are large models that are trained on a wide range of tasks, and can be fine-tuned for specific tasks. Transformers make them possible because of the following reasons:

- they are **scalable** and can be trained on very large datasets, exploiting parallelism and distributed computing;
- they can be trained in a **self-supervised** way, i.e., without the need for labeled data;
- the **scaling hypothesis** asserts that simply by increasing the scale of the model, as measured by the number of parameters, and training on a commensurately large dataset, significant improvements can be achieved, even with no architectural changes.

Scaling Hypothesis

The scaling hypothesis

performance improves smoothly as we increase the ***model*** *size,* ***datasetset*** *size, and amount of* ***compute*** *used for training. For optimal performance* ***all three factors must be scaled up in tandem****.*

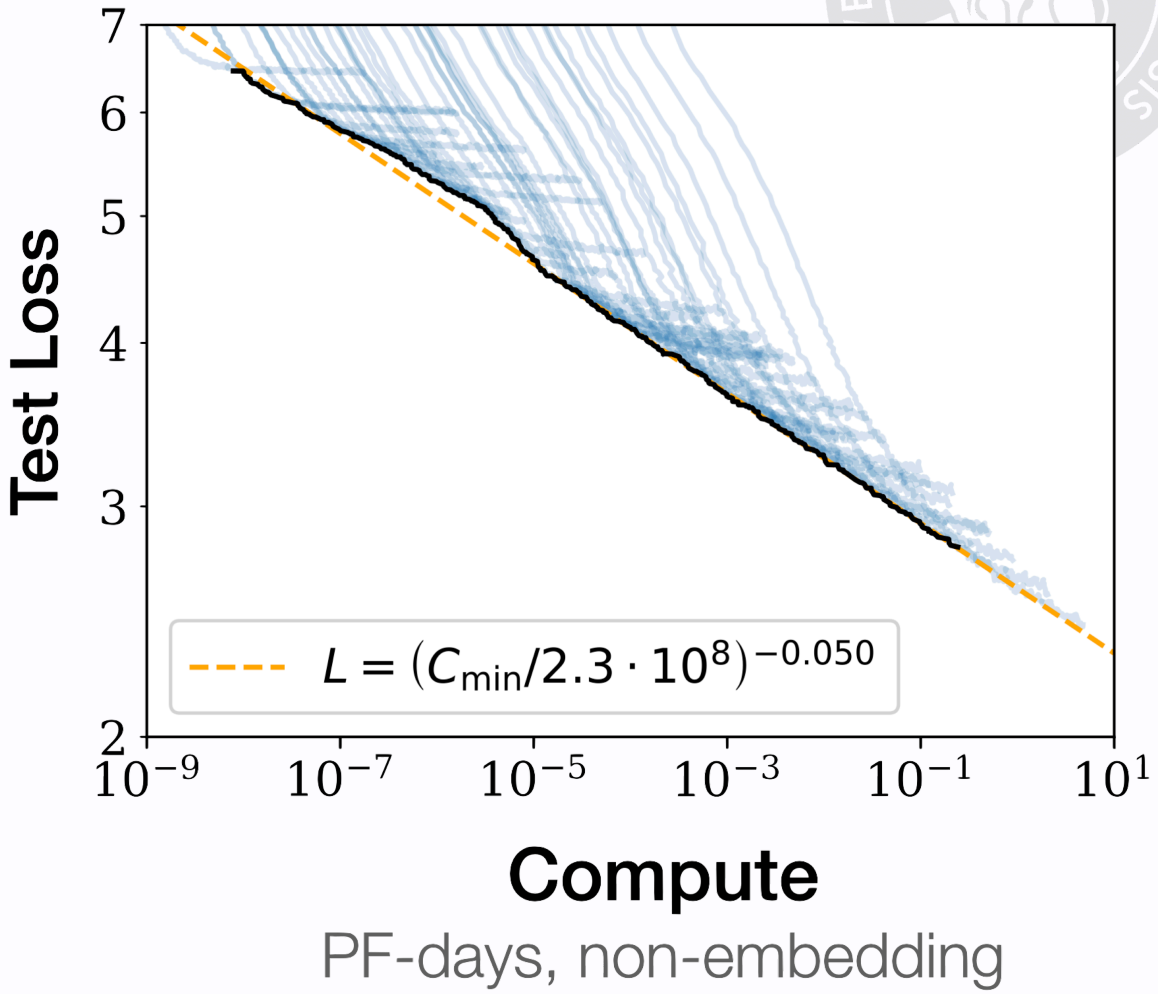


Image from: Scaling Laws for Neural Language Models

PF-day: is the number of floating-point operations that a machine capable of 1PF/s would perform in a day.

Attention

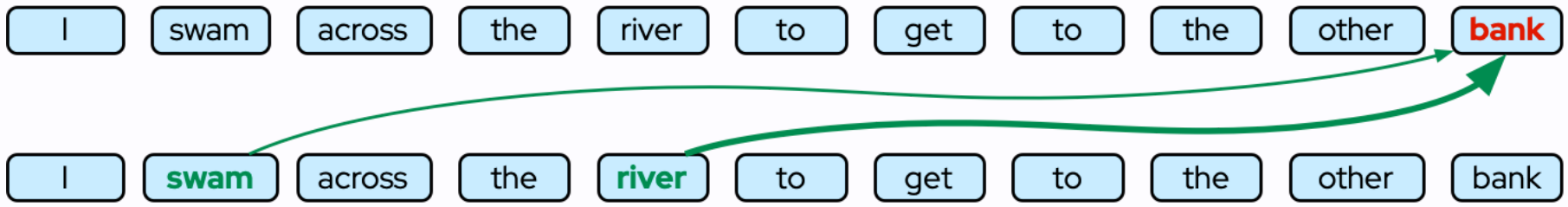
Attention is a mechanism that allows a model to focus on different parts of the input when making predictions.

It was originally introduced as an enhancement to RNNs for machine translation; it was later showed that **significantly better results can be achieved using attention mechanisms alone** (i.e., eliminating the recurrence mechanism).

Consider the following two sentences:

- "I swam across the river to get to the other **bank**."
- "I walked across the road to get cash from the **bank**."

and notice how the word "bank" has different meanings in the two sentences.



A neural network processing this sentence should **attend** to the words "swam" and "river" more than to the other words when trying to give a meaning to the word "bank".

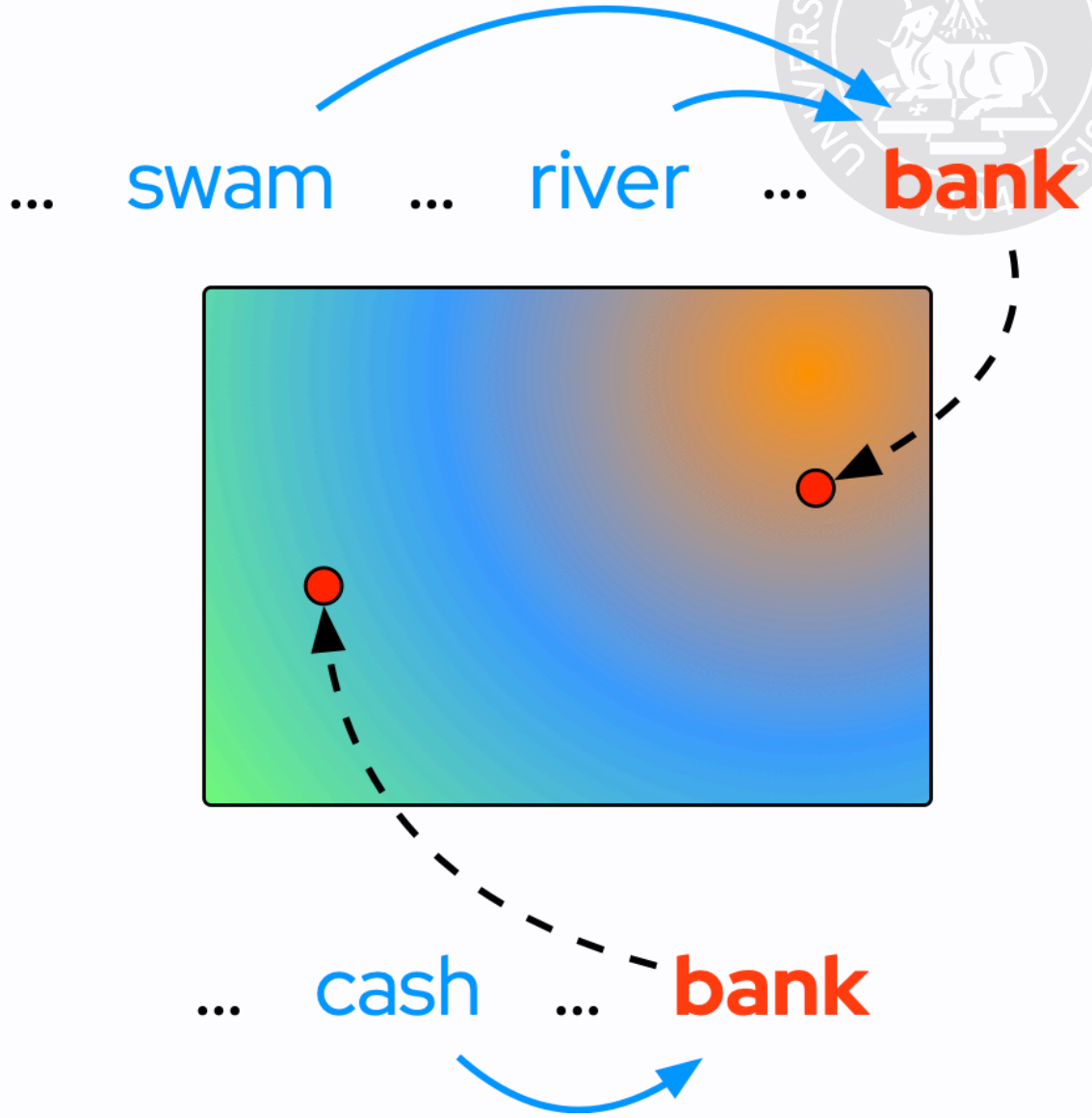
In **a standard neural network**, different inputs will influence the output to different extents according to the values of the weights that multiply those inputs. Once the network is trained, the weights are fixed and **the network will always give the same importance to the same locations**.

In the given example, however, the locations to attend **depend on the sequence itself**:

- I **swam** across the **river** to get to the other **bank**.
- I walked across the road to get **cash** from the **bank**.


The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>
The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>

A transformer can be viewed as a way to build a **richer form of embedding** in which a given vector is mapped to a location that depends on the other vectors in the sequence.

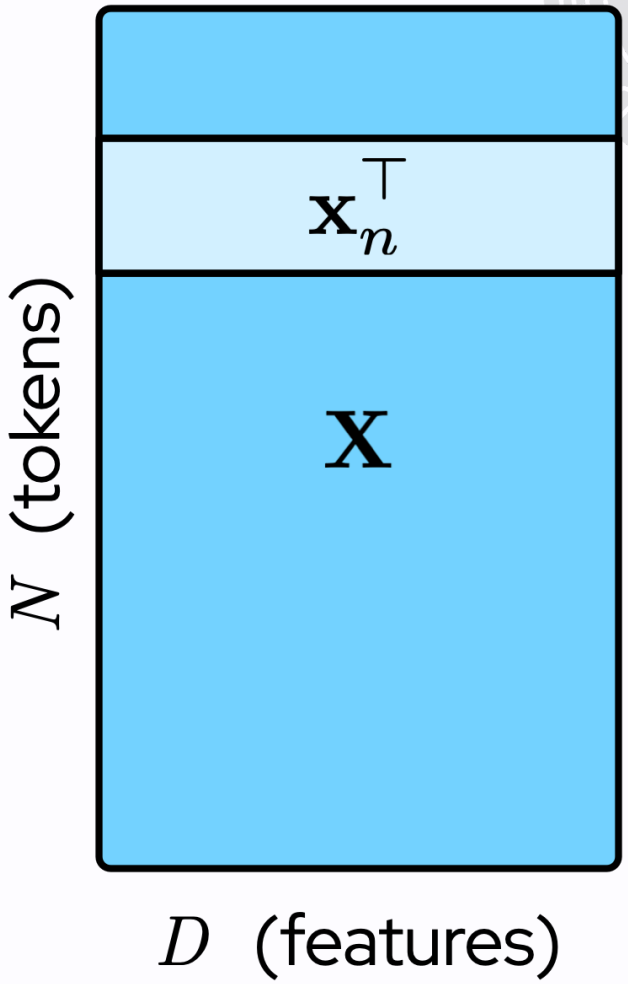


Transformer processing

Input data to a transformer is a sequence of vectors $[x_n^\top]_{n\in[N]}$ of dimensionality D.

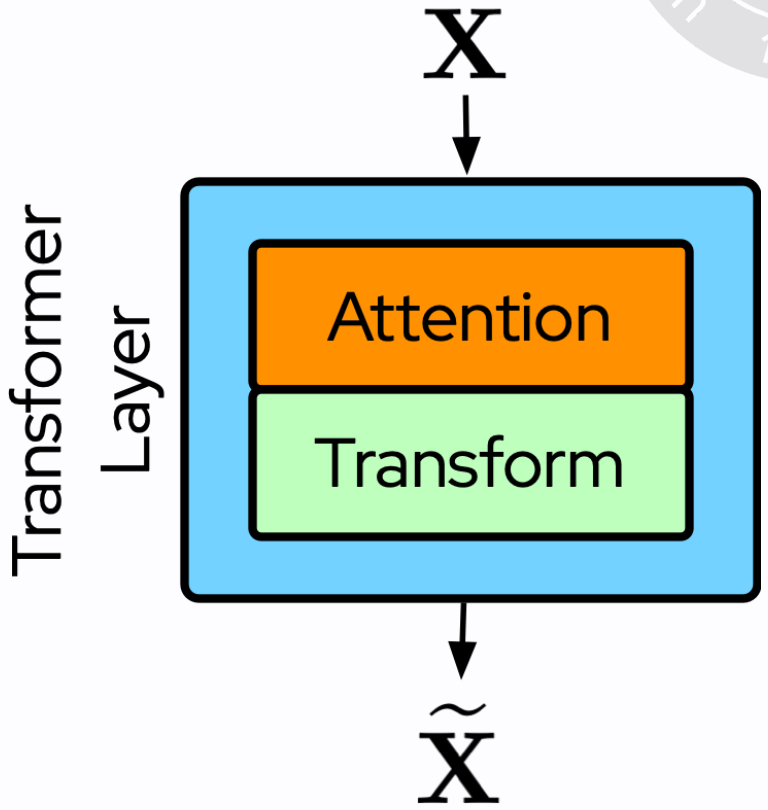
Each vector is called a **token** and represents a word in a sentence, a patch within an image, an amino acid in a protein, etc..

Tokens are collected in a matrix $\mathbf{X} \in \mathbb{R}^{N\times D}$, which is the input to the transformer.



The fundamental building block of a transformer network is the **transformer layer**, which is a function that takes a matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$ as input and produces a matrix $\widetilde{\mathbf{X}} \in \mathbb{R}^{N \times D}$ as output..

$$\widetilde{\mathbf{X}} = \mathrm{TransformerLayer}[\mathbf{X}]$$

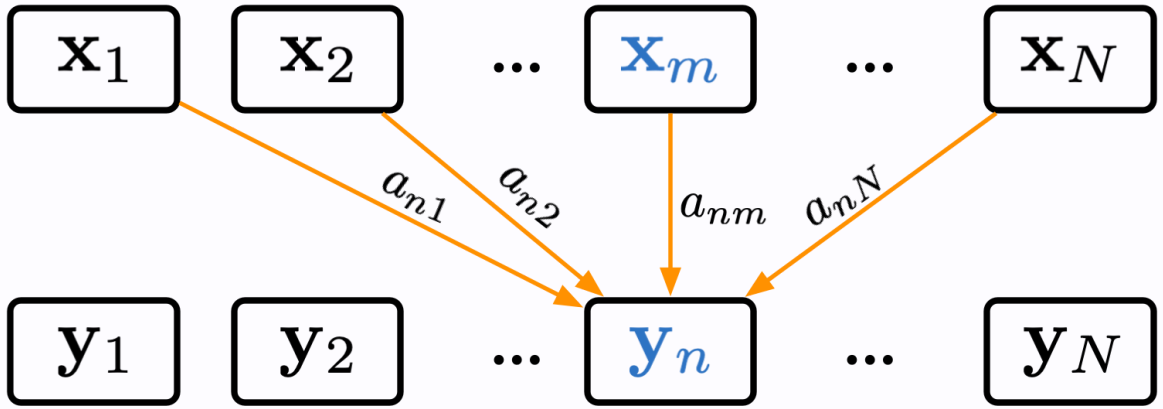


Attention weights

Assume to want to compute new embeddings $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N$ for tokens $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ in such a way that the embedding for $\mathbf{y}_n$ depends on the embeddings of all other tokens.

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm} \mathbf{x}_m$$

where the coefficients a_{nm} are the **attention weights**:



We would require attention weights a_{nm} to:

- capture the similarity between the tokens $\mathbf{x}_n$ and $\mathbf{x}_m$;
- $a_{nm} \geq 0$
- $\sum_{m=1}^{N} a_{nm} = 1$

$$\Rightarrow \quad a_{nm} = \frac{\exp(\mathbf{x}_n^\top \mathbf{x}_m)}{\sum_{m'=1}^{N} \exp(\mathbf{x}_n^\top \mathbf{x}_{m'})}$$

Self-attention

*This mechanism is called **dot-product self-attention** because it computes the attention weights **between the tokens in the same sequence** using the **dot-product** as a similarity measure.*

Question: assume that $\mathbf{x}_n$ and $\mathbf{x}_m$ are unrelated/independent, i.e., orthogonal, what can you say about the attention weights?

In matrix form

$$\mathbf{X}\mathbf{X}^\top = \begin{bmatrix} — & \mathbf{x}_1^\top & — \\ — & \mathbf{x}_2^\top & — \\ & \vdots & \\ — & \mathbf{x}_N^\top & — \end{bmatrix} \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_N \\ | & | & & | \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^\top \mathbf{x}_1 & \dots & \mathbf{x}_1^\top \mathbf{x}_N \\ \vdots & \ddots & \vdots \\ \mathbf{x}_N^\top \mathbf{x}_1 & \dots & \mathbf{x}_N^\top \mathbf{x}_N \end{bmatrix}$$

Let $\mathrm{Softmax}$ to be an operator that take the exponential of each element of a matrix and then normalize the rows of the resulting matrix to sum to one.

We can compute the new embeddings $\mathbf{Y}$ as:

$$\mathbf{Y} = \mathrm{Softmax}[\mathbf{X}\mathbf{X}^\top]\mathbf{X},$$

Terminology

In the original paper, the authors borrow terminology from the field of information retrieval. Let us assume to want to build a model that can retrieve movies from a movie database based on a query from the user.

In this context:

- the **query** is the user's request;
- the **key** is the information that the system uses to accompany the movie data and that should be matched with the query at retrieval time;
- the **value** is the information that the system should return to the user when the query matches the key (i.e., the movie data).

We say that the user will **attend** to the particular movie whose key is most similar to the query. This would be a form of **hard attention**, for transformers we use **soft attention**, where we use continuous variables to measure the degree of match between the query and the keys and use these to weight the influence of the value vectors on the output.

Using this terminology, we can rewrite the formula for the new embeddings as:

$$\mathbf{Y} = \mathrm{Softmax}[\mathbf{X}\mathbf{X}^\top]\mathbf{X} = \mathrm{Softmax}[\mathbf{Q}\mathbf{K}^\top]\mathbf{V}$$

and following the analogy:

- token $\mathbf{x}_n$ is the **value** vector that will be used to compute output tokens;
- we also use $\mathbf{x}_n$ as the **key** for value $\mathbf{x}_n$;
- finally, we use $\mathbf{x}_n$ as the **query** to compute the attention weights for output $\mathbf{y}_n$.

We have **self-attention** because we are using the same sequence to determine the **queries**, **key**, and **values**.

Trainable attention weights

The current formulation of the attention mechanism:

- is **deterministic** and does not depend on the parameters of the model, i.e., cannot be trained;
- each feature within token $\mathbf{x}_n$ contributes equally to the attention weights, whereas we would like have the flexibility to focus more on some features than on others.

To solve these issues, we can introduce **trainable parameters** to compute the attention weights and let:

$$\tilde{\mathbf{X}} = \mathbf{X}\mathbf{U}$$

where $\mathbf{U} \in \mathbb{R}^{D \times D}$ is a matrix of trainable parameters (analogous to a 'layer' in a standard neural network).

Substituting this expression in the formula for the new embeddings, we obtain:

$$\mathbf{Y} = \mathrm{Softmax}[\mathbf{X}\mathbf{U}\mathbf{U}^\top\mathbf{X}^\top]\mathbf{X}\mathbf{U}$$

which has the **problem** that **the matrix $\mathbf{X}\mathbf{U}\mathbf{U}^\top\mathbf{X}^\top$ is symmetric**, while we want to support significant asymmetries.

Example

For example, we might expect that 'chisel' should be strongly associated with 'tool' since every chisel is a tool, whereas 'tool' should only be weakly associated with 'chisel' because there are many other kinds of tools besides chisels.

Also, **we are using the same matrix $\mathbf{U}$ to define both the value vectors and the attention coefficients**, which is not ideal.

We can overcome these limitation defining separate matrices for the **queries**, **keys**, and **values**:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(q)}$$
$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(k)}$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}^{(v)}$$

where matrices $\mathbf{W}^{(q)}$, $\mathbf{W}^{(k)}$, $\mathbf{W}^{(v)}$ collect **trainable parameters**.

- $\mathbf{W}^{(k)}$ has dimensionality $D \times D_k$, where D_k is the dimensionality of the **key** vectors;
- $\mathbf{W}^{(q)}$ must also have dimensionality $D \times D_k$, so that the dot product with **query** vectors $\mathbf{Q}\mathbf{K}^\top$ is well defined;
- $\mathbf{W}^{(v)}$ has dimensionality $D \times D_v$, where D_v is the dimensionality of the **value** (output) vectors.

Usually, to simplify things and to facilitate the inclusion of residual connections $D_k = D_v = D$. Also this choice allows one to stack multiple attention layers on top of each other.

The final expression for the output of the self-attention mechanism is:

$$\mathbf{Y} = \mathrm{Softmax}\left[\mathbf{Q}\mathbf{K}^\top\right]\mathbf{V}$$

Scaled self-attention

The gradient of the softmax function can become exponentially small for inputs of high magnitude, which can lead to **vanishing gradients** during training. To mitigate this issue, we can scale the values of the dot product by a factor $\sqrt{D}$, where D is the dimensionality of the key vectors:

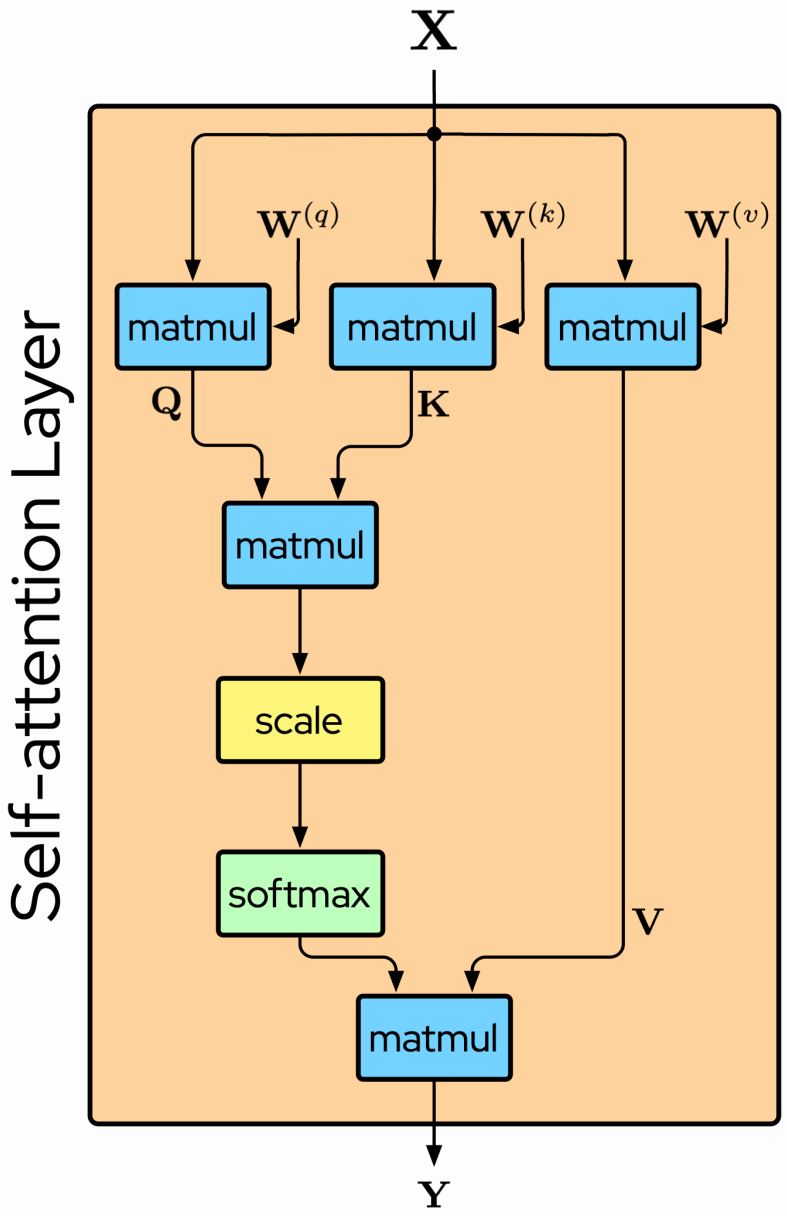
$$\mathbf{Y} = \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{D_k}}\right]\mathbf{V}$$

The scaling factor is chosen because it is the variance of the dot product between two independent vectors of dimensionality D_k. Specifically, let us assume that $\mathbf{q}$ and $\mathbf{k}$ are rows from $\mathbf{Q}$ and $\mathbf{K}$ respectively, that they are independent, and that they have zero mean and unit variance, we have:

$$\begin{aligned}\mathrm{Var}[\mathbf{q}^\top \mathbf{k}] &= \sum_i \mathrm{Var}[q_i k_i] \quad \textit{(by independence)} \\ &= \sum_i \left(\mathbb{E}[q_i^2 k_i^2] - \mathbb{E}[q_i k_i]^2\right) \\ &= \sum_i \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] = \sum_i 1 = D_k,\end{aligned}$$

where we used the fact that $\mathbb{E}[q_i^2] = 1$ and $\mathbb{E}[k_i^2] = 1$, since, e.g.,

$$\mathbb{E}[q_i^2] = \mathbb{E}[(q_i - 0)^2] = \mathbb{E}[(q_i - \mathbb{E}[q_i])^2] = \mathrm{Var}[q_i] = 1.$$


X
Self-attention Layer
$\mathbf{W}^{(q)}$
$\mathbf{W}^{(k)}$
$\mathbf{W}^{(v)}$
matmul
matmul
matmul
Q
K
matmul
scale
softmax
V
matmul
Y

Multi-head self-attention

The **multi-head attention** mechanism is a way to cope with the fact there might be multiple patterns of attention that are relevant at the same time.

Example

In natural language, for example, some patterns might be relevant to tense whereas others might be relevant to subject-verb agreement. Using a single attention head can lead to averaging over these effects.

Suppose we have H heads indexed by $h \in [H]$. For each head h:

$$\mathbf{H}_h = \text{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h),$$

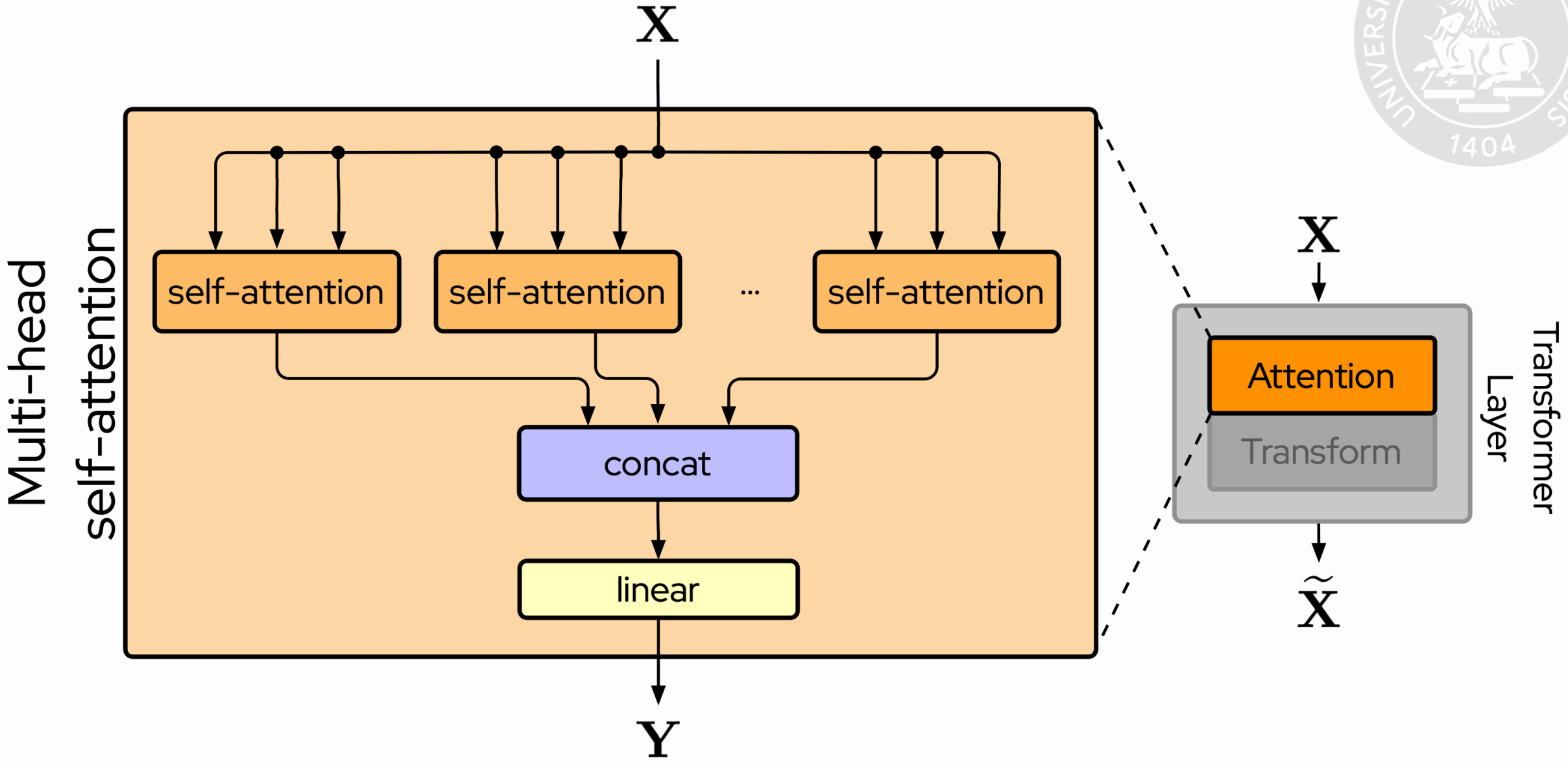
where

$$\begin{aligned}\mathbf{Q}_h &= \mathbf{X}\mathbf{W}_h^{(q)} \\ \mathbf{K}_h &= \mathbf{X}\mathbf{W}_h^{(k)} \\ \mathbf{V}_h &= \mathbf{X}\mathbf{W}_h^{(v)}\end{aligned}$$

The heads are first concatenated and then multiplied by a matrix $\mathbf{W}^{(o)}$ to obtain the final output:

$$\overbrace{\mathbf{Y}(\mathbf{X})}^{N \times D} = \overbrace{\text{Concat}[\mathbf{H}_1, \ldots, \mathbf{H}_H]}^{N \times HD_v} \overbrace{\mathbf{W}^{(o)}}^{HD_v \times D}$$

Typically $D_v = D/H$ so that the concatenated matrix has the same dimensionality as the input matrix.


X
Multi-head
self-attention
self-attention
self-attention
...
self-attention
concat
linear
Y
X
Attention
Transform
Transformer
Layer
X̃

Transformers Layer

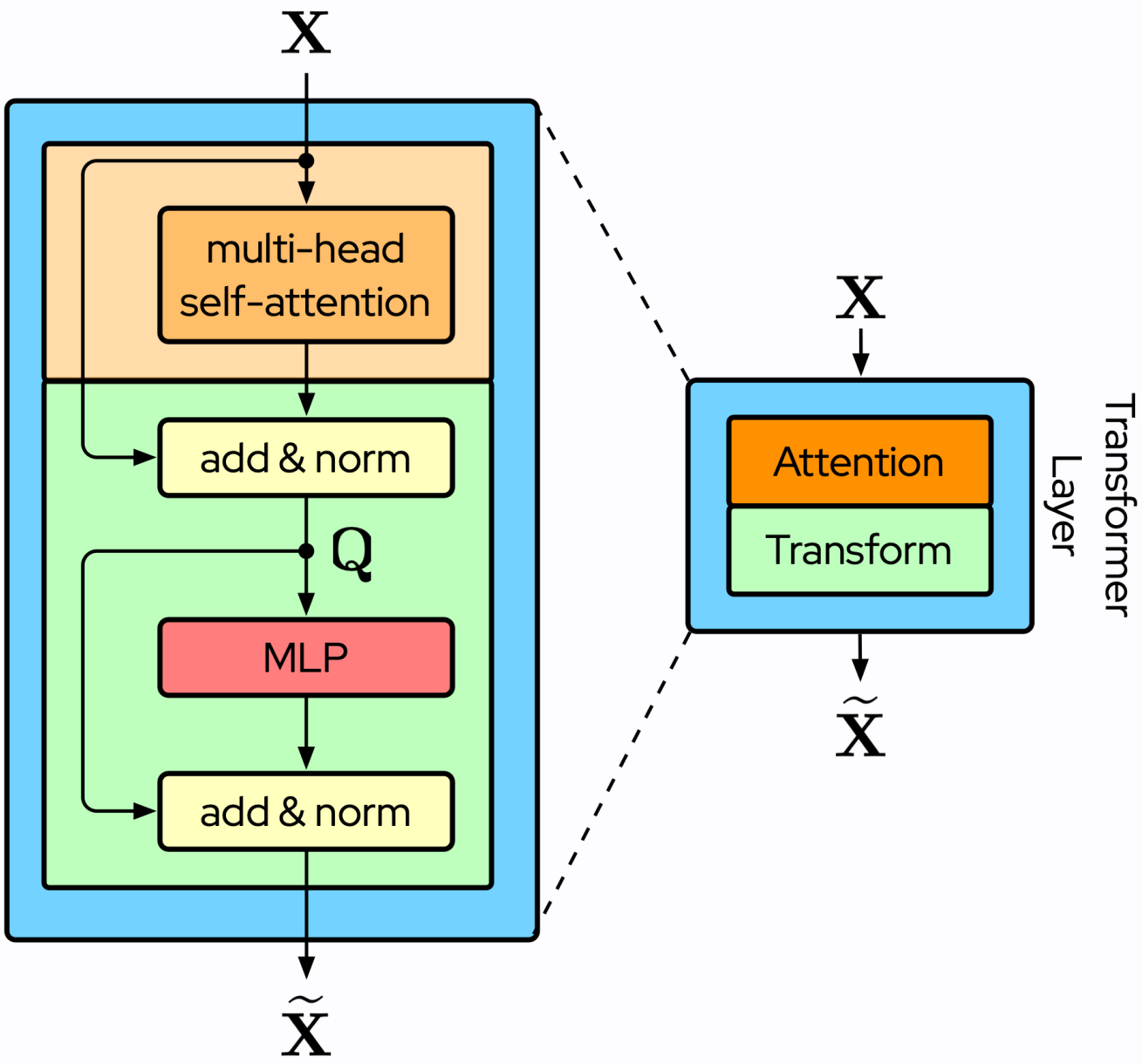
A few additional improvements can be made to the self-attention mechanism to make it more expressive and easier to train:

- to improve training efficiency, we can add a **residual connection** around the self-attention mechanism, followed (or preceeded) by **layer normalization**:

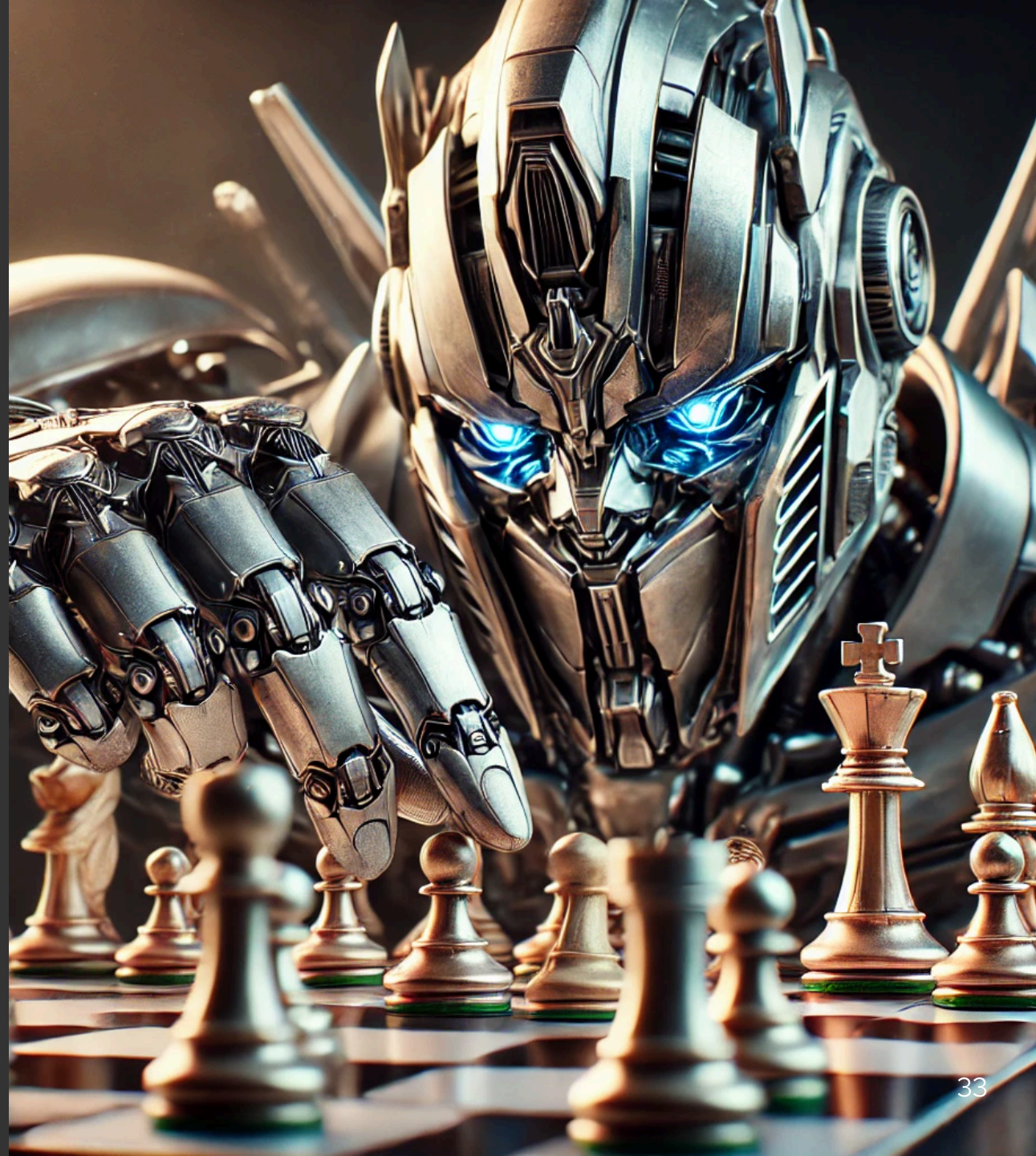
$$\mathbf{Z} = \mathrm{LayerNorm}[\mathbf{Y}(\mathbf{X}) + \mathbf{X}] \quad \text{or} \quad \mathbf{Z} = \mathbf{Y}(\mathrm{LayerNorm}[\mathbf{X}]) + \mathbf{X};$$

- the output is then passsed through a non-linear neural network with D input units and D output units (MLP), for instance this can be a two-layer feedforward neural network with ReLU activations:

$$\tilde{\mathbf{X}} = \mathrm{LayerNorm}[\mathrm{MLP}[\mathbf{Z}] + \mathbf{Z}] \quad \text{or} \quad \tilde{\mathbf{X}} = \mathrm{MLP}[\mathrm{LayerNorm}[\mathbf{Z}]] + \mathbf{Z}.$$


X
multi-head self-attention
add & norm
Q
MLP
add & norm
$\tilde{X}$
X
Attention
Transform
Transformer Layer
$\tilde{X}$

Positional encoding



Consider a permutation $\mathbf{P}$ of the rows of matrix $\mathbf{X}$.

The main matrix operations in the transformer layer are: $\mathbf{X}\mathbf{X}^\top\mathbf{X}$. Note that:

$$\mathbf{P}\mathbf{X}(\mathbf{P}\mathbf{X})^\top\mathbf{P}\mathbf{X} = \mathbf{P}\mathbf{X}\mathbf{X}^\top\mathbf{P}^\top\mathbf{P}\mathbf{X} = \mathbf{P}(\mathbf{X}\mathbf{X}^\top\mathbf{X}).$$

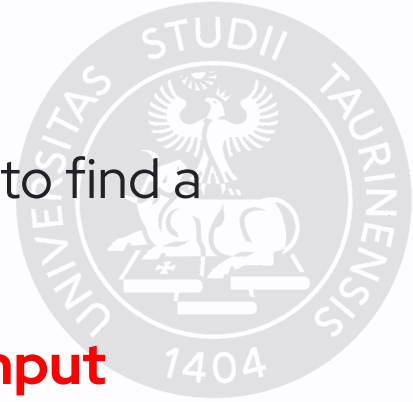
Implying that the output of the **transformer layer is equivariant** to the permutation of the input tokens.

This is not a complete proof, since there are many other operations in the transformer layer, but what shown is the mean reason behind the problem.

Problem

For sequential tasks the order of the tokens is very often crucial to determine the meaning of the sentence. "The food was ***bad****, not* ***good*** *at all" has a* ***very*** *different meaning than "The food was* ***good****, not* ***bad*** *at all".*

The equivariance property *of the transformer layer make these distinctions hard to model.*



Approach: We *do not want* to change the transformer architecture, then we need to find a way to **encode the position of the tokens in the input sequence**.

The idea is then **to encode the position n of the n-th tokens as an additional input vector $\mathbf{r}_n$** and combine it with the token vector $\mathbf{x}_n$ before passing it to the transformer.

Example

Let us assume to encode:

- *position* $n = 1$ *as* $\mathbf{r}_1 = [1, 0, 0, 0, \ldots]$;
- *position* $n = 2$ *as* $\mathbf{r}_2 = [0, 1, 0, 0, \ldots]$;
- *position* $n = 3$ *as* $\mathbf{r}_3 = [0, 0, 1, 0, \ldots]$;
- ...

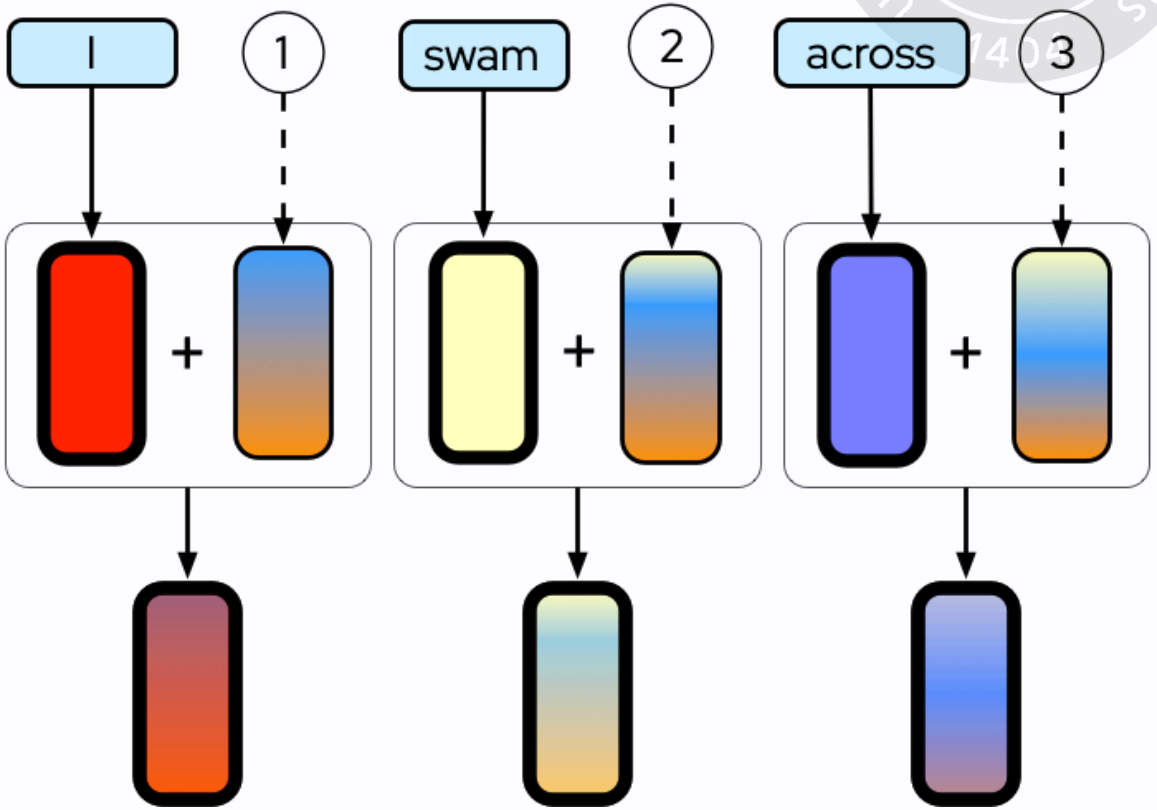
then we can combine the positional encoding to the token vector as $\widetilde{\mathbf{x}}_n = \mathbf{x}_n \circ \mathbf{r}_n$, *where* $\circ$ *denotes a suitable combination operation.*

How do we combine the positional encoding with the token vector?

Appending the positional encoding to the token vector is not desirable because this would increase the dimensionality of the input space and hence of all the subsequent layers, increasing significantly the computational cost.

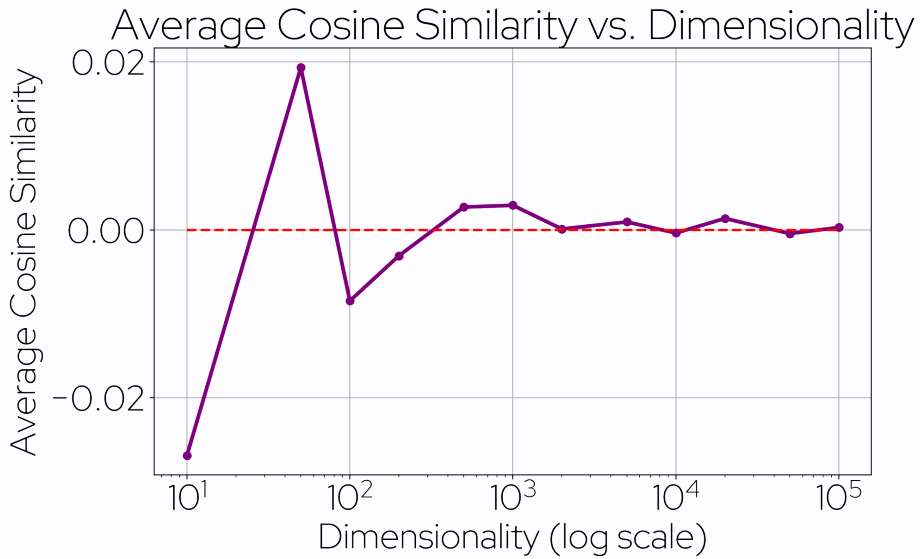
Instead we can simply add the positional encoding to the token vector:

$$\widetilde{\mathbf{x}}_n = \mathbf{x}_n + \mathbf{r}_n.$$



It might seem that adding the position information onto the token vector would corrupt the input vectors and make the task more difficult, but in practice this is not the case: **two randomly chosen uncorrelated vectors tend to be nearly orthogonal in high-dimensional spaces**, allowing the network to process them separately.

Also, thanks to residual connections, the positional information does not get lost in going from one layer to the next.



Due to the linear processing in the transformer, **a concatenated representation exhibits properties similar to an additive one**.

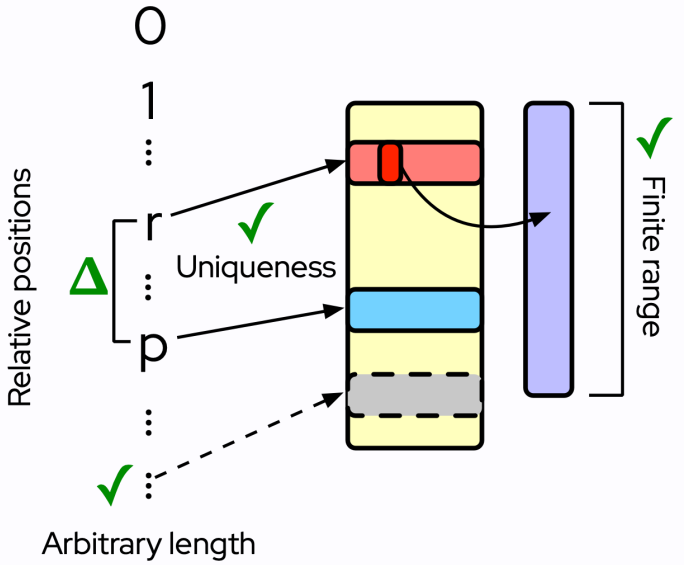
Consider representation $\mathbf{x}|\mathbf{r}$, where $|$ denotes concatenation and let us apply a linear transformation $\mathbf{w_x}|\mathbf{w_r}$ to it:

$$\begin{bmatrix}\mathbf{w_x} & \mathbf{w_r}\end{bmatrix}\begin{bmatrix}\mathbf{x} \\ \mathbf{r}\end{bmatrix} = \mathbf{w_x x} + \mathbf{w_r r} = \mathbf{w}(\mathbf{x}+\mathbf{r})$$

where in the last equality, we assumed to want to apply the same transformation to the token representations and the positional encodings.

An **ideal encoding** should:

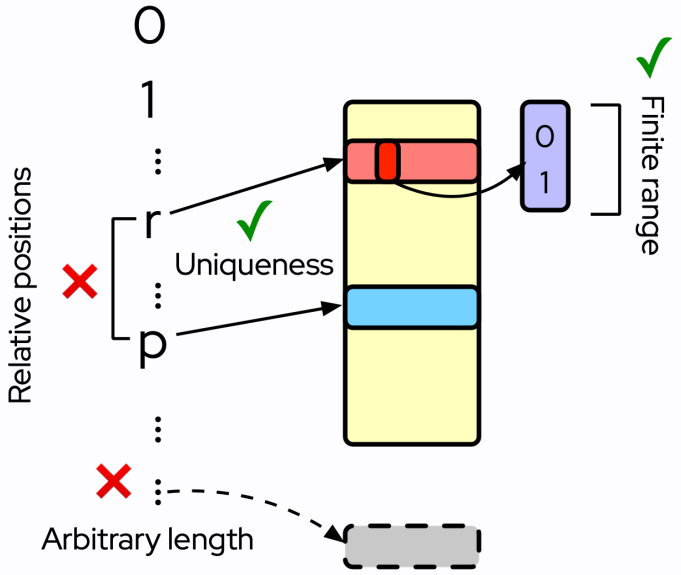
- be **unique** for each position;
- be bounded, i.e., each element of the encoding representation should have a **finite range**;
- generalize to **sequences of arbitrary length**;
- have a consistent way to express **relative positions**.

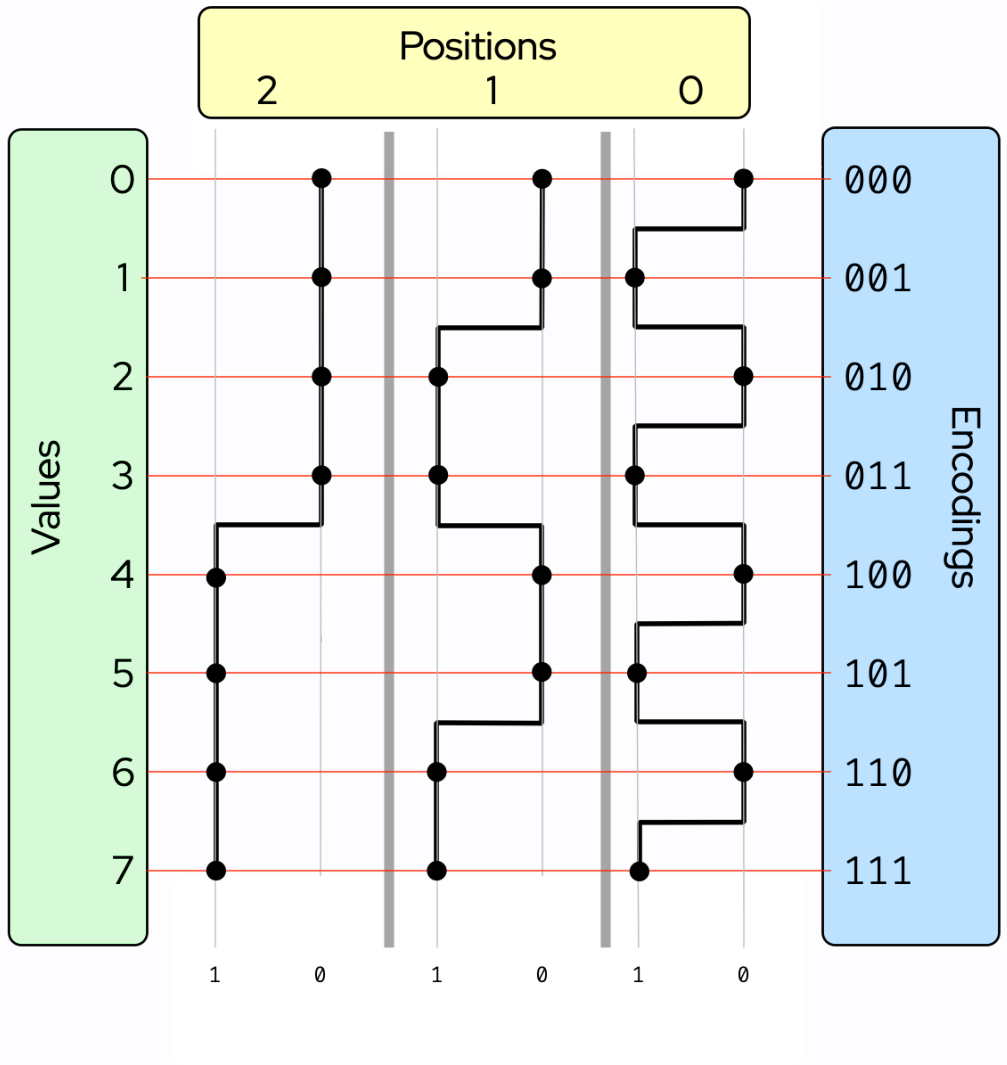


Examples of bad positional encodings

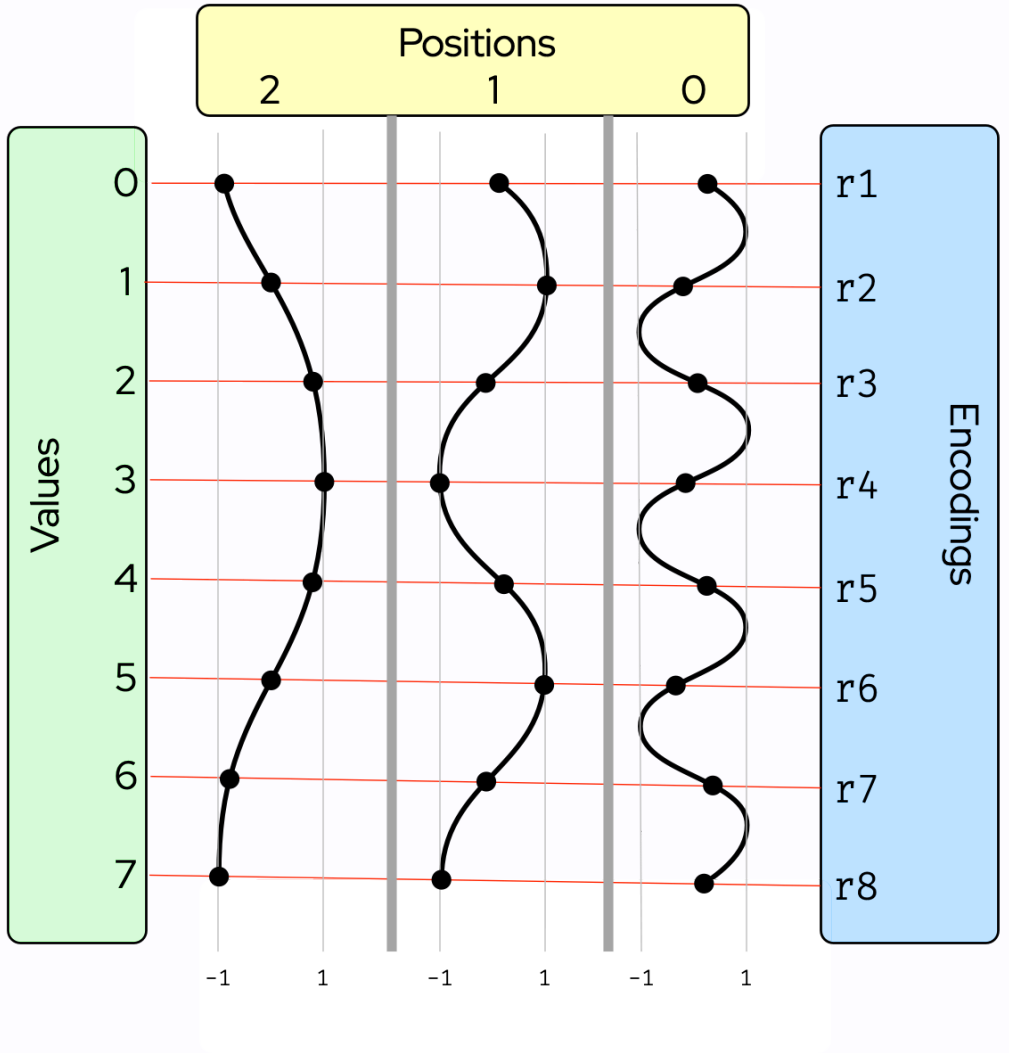
- **One-hot encoding**: it is unique and bounded, but it does not generalize to sequences of arbitrary length and does not make it easy to reason about relative positions.
- **Assigning an integer to each position**: is unique, but not bounded, it may start to corrupt the vector significantly as the sequence length increases.
- **Assigning a real number in $[0, 1]$ to each position**: is bounded, but it is not unique since it depends on the length of the sequence

One hot encoding





Positional encoding using square waves.



Positional encoding using sinusoidal waves.

Sinusoidal positional encoding

There are many approaches to define positional encodings, one of the most popular is the **sinusoidal positional encoding** (Vaswani et al., 2017).

D=dimension of the reprentation

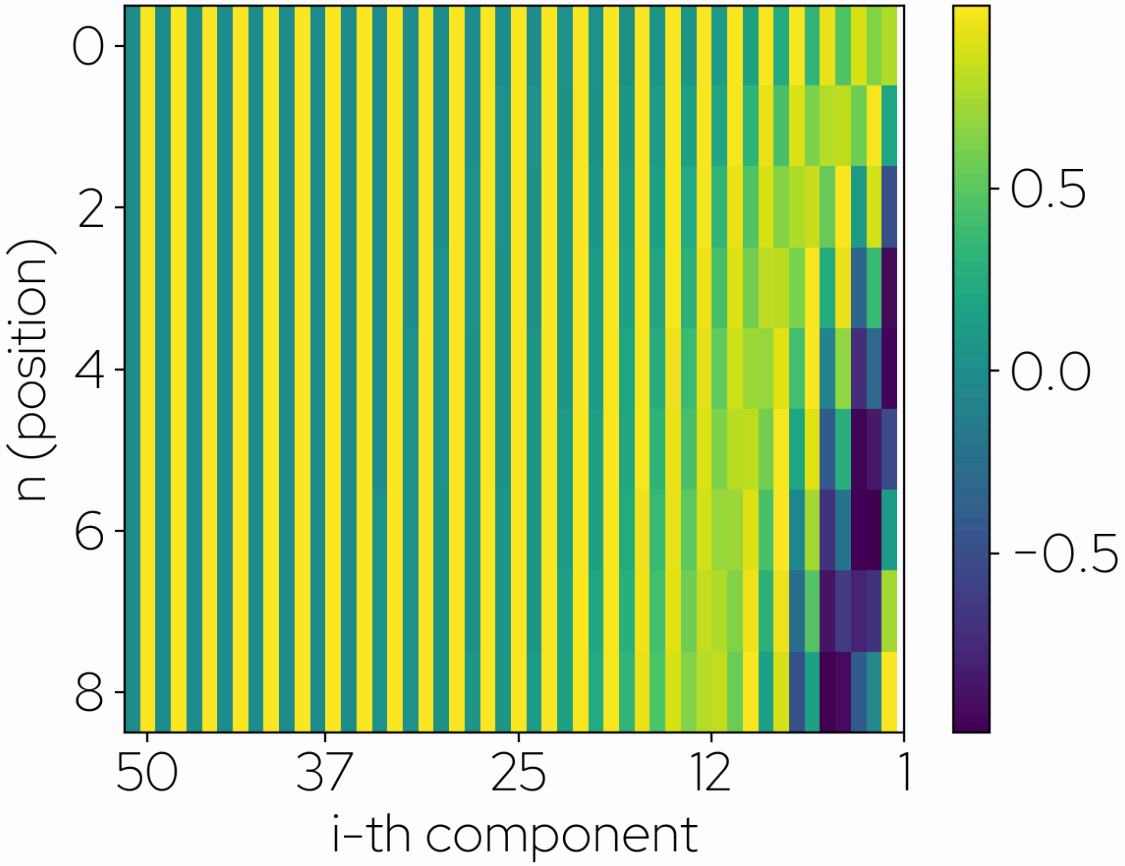
$$\mathbf{r}_n = \begin{bmatrix} \sin\left(w_1 \cdot n\right) \\ \cos\left(w_1 \cdot n\right) \\ \sin\left(w_2 \cdot n\right) \\ \cos\left(w_2 \cdot n\right) \\ \vdots \\ \sin\left(w_{D/2} \cdot n\right) \\ \cos\left(w_{D/2} \cdot n\right) \end{bmatrix}, \quad \text{where } w_i = \frac{1}{10000^{2i/D}}$$

w_1 ecc sono frequenze, ognuna moltiplicata per il numero n che rappresenta la posizione

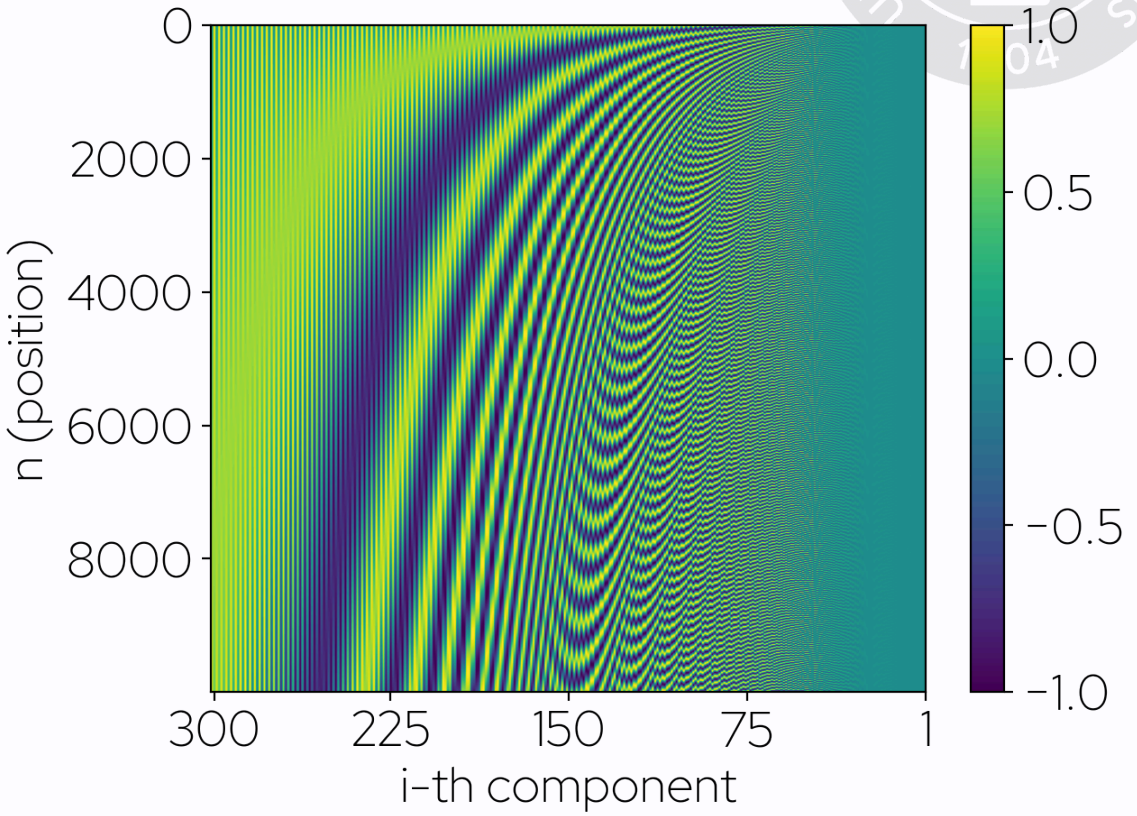
anche questo viene imparato dalla rete neurale

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

for D=50



for D=300



A nice property of the sinusoidal positional encoding is that it makes it easy to reason about **relative positions**.

Here are two reasoning that sustain this assertion:

1. the dot product between two positional encodings $\mathbf{r}_n$ and $\mathbf{r}_m$ depends only on the difference $n - m$ and not on the absolute positions n and m:

$$\begin{aligned}\mathbf{r}_n^\top \mathbf{r}_m &= \sum_{i=1}^{D/2} \sin\left(w_i \cdot n\right) \sin\left(w_i \cdot m\right) + \cos\left(w_i \cdot n\right) \cos\left(w_i \cdot m\right) \\ &= \sum_{i=1}^{D/2} \cos\left(w_i \cdot (n - m)\right).\end{aligned}$$

conta solo la distanza tra le due rappresentazioni

2. the encoding of $n + m$ can be expressed as **a linear combination of the encodings** of n and m. In fact it is always possible to find a matrix $\mathbf{M}$ that depends only on k, such that $\mathbf{r}_{n+k} = \mathbf{M}\mathbf{r}_n$.

Using: $\sin(a)\sin(b) + \cos(a)\cos(b) = \cos(a - b)$.

Proof of the second property

Let us consider a frequency w_i, we look for a matrix $\mathbf{M}$ such that:

$$\overbrace{\begin{bmatrix} v_1 & v_2 \\ v_3 & v_4 \end{bmatrix}}^{\mathbf{M}} \cdot \begin{bmatrix} \sin(w_i \cdot n) \\ \cos(w_i \cdot n) \end{bmatrix} = \begin{bmatrix} \sin(w_i \cdot (n+k)) \\ \cos(w_i \cdot (n+k)) \end{bmatrix}$$

esiste una matrice per cui se moltiplico per essa il vettore del positional encoding ottengo la rappresentazione di r_{n+k}

By matrix multiplication we have:

$$\begin{bmatrix} v_1 & v_2 \\ v_3 & v_4 \end{bmatrix} \cdot \begin{bmatrix} \sin(w_i \cdot n) \\ \cos(w_i \cdot n) \end{bmatrix} = \begin{bmatrix} v_1 \sin(w_i \cdot n) + v_2 \cos(w_i \cdot n) \\ v_3 \sin(w_i \cdot n) + v_4 \cos(w_i \cdot n) \end{bmatrix}$$

Due to the summation formulas for sine and cosine, we can write:

$$\begin{bmatrix} \sin(w_i \cdot (n+k)) \\ \cos(w_i \cdot (n+k)) \end{bmatrix} = \begin{bmatrix} \sin(w_i \cdot n)\cos(w_i \cdot k) + \cos(w_i \cdot n)\sin(w_i \cdot k) \\ \cos(w_i \cdot n)\cos(w_i \cdot k) - \sin(w_i \cdot n)\sin(w_i \cdot k) \end{bmatrix}$$

Using: $\sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b)$
and: $\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$.

$$\begin{bmatrix} v_1 \sin(w_i \cdot n) + v_2 \cos(w_i \cdot n) \\ v_3 \sin(w_i \cdot n) + v_4 \cos(w_i \cdot n) \end{bmatrix} = \begin{bmatrix} \sin(w_i \cdot n)\cos(w_i \cdot k) + \cos(w_i \cdot n)\sin(w_i \cdot k) \\ \cos(w_i \cdot n)\cos(w_i \cdot k) - \sin(w_i \cdot n)\sin(w_i \cdot k) \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} v_1 \sin(w_i \cdot n) + v_2 \cos(w_i \cdot n) \\ v_3 \sin(w_i \cdot n) + v_4 \cos(w_i \cdot n) \end{bmatrix} = \begin{bmatrix} \sin(w_i \cdot n)\cos(w_i \cdot k) + \cos(w_i \cdot n)\sin(w_i \cdot k) \\ \cos(w_i \cdot n)\cos(w_i \cdot k) - \sin(w_i \cdot n)\sin(w_i \cdot k) \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} v_1 & v_2 \\ v_3 & v_4 \end{bmatrix} = \begin{bmatrix} \cos(w_i \cdot k) & \sin(w_i \cdot k) \\ -\sin(w_i \cdot k) & \cos(w_i \cdot k) \end{bmatrix}$$

this is cool :)

and very simple!

Note: this property would not hold anymore if the positional encoding was purely sinusoidal or purely cosinusoidal.

yooo

Applications of the transformers architecture

this is le ebic

The **transformer architecture** has been used to build models that have achieved state-of-the-art performance on a wide range of tasks that deal with processing sequences.

In general, when processing a sequence, three tasks are possible:

1. **prediction**: given a sequence make a prediction about its nature (e.g., classify it, or output a numerical value). Transformers are used as encoders of the sequence. *Example:* sentiment analysis.
   see Bishop et al., §12.3.3
2. **generation**: given a single input, generate a sequence of tokens. Transformers are used as decoders of the sequence. *Example:* generate a caption for an image.
   see Bishop et al., §12.3.1
3. **translation** a.k.a. **seq2seq**: given a sequence, generate a new sequence. Transformers are used for both encoding and decoding the sequences. *Example:* machine translation.
   see Bishop et al., §12.3.4

Tokenization

Modern natural language models are trained on large datasets that contain text in the form of sequences of **tokens**.

Tokens are created in a pre-processing step that convert a string of words and punctuation into a string of tokens, which are generally small groups of characters and might include common words in their entirety, along with fragments of longer words as well as individual characters that can be assembled in less common words.

we have a corpus of text that we want to organize in n tokens... questa tokenizzazione può anche essere applicata a immagini

Many approaches to tokenization exist, one of the most common is the **byte pair encoding** algorithm.

```
def byte_pair_encoding(corpus, n_tokens):
    tokens = initialize_tokens_w_chars(corpus)

    for _ in range(n_tokens):
        pairs = count_adjacent_pairs(tokens)

        if not pairs:
            break

        best_pair = find_most_frequent_pair(pairs)
        tokens = replace_pair_in_tokens(tokens, best_pair)

    return tokens
```

inizializziamo con singoli caratteri

replace this new toaken

quindi partiamo con le singole lettere, cerchiamo le coppie più frequenti e alla fine del loop avremo di nuovo le singole lettere ma in più anche questa coppia più frequente.. e così via

Example

capitalization counts!

Peter Piper picked a peck of pickled peppers

Peter Piper picked a peck of pickled peppers — nuovo token: pe

Peter Piper picked a peck of pickled peppers — nuovo token: ck

Peter Piper picked a peck of pickled peppers — nuovo token: pi

Peter Piper picked a peck of pickled peppers — nuovo token: ed

Peter Piper picked a peck of pickled peppers

a questo punto la nuova coppia più frequente è pe+r, quindi il mio nuovo token è per

The example sentence would be tokenized as:

Try yourself on the OpenAI tokenizer: https://platform.openai.com/tokenizer

From the OpenAI documentation:

> *A helpful rule of thumb is that one token generally corresponds to ~4 characters of text for common English text. This translates to roughly ¾ of a word (so 100 tokens ~= 75 words).*

Decoder transformers

The GPT (Generative Pre-trained Transformer) models are examples of **decoder transformers**.

The goal is to use the transformer architecture to construct an **autoregressive model** of the form:

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n-1}).$$

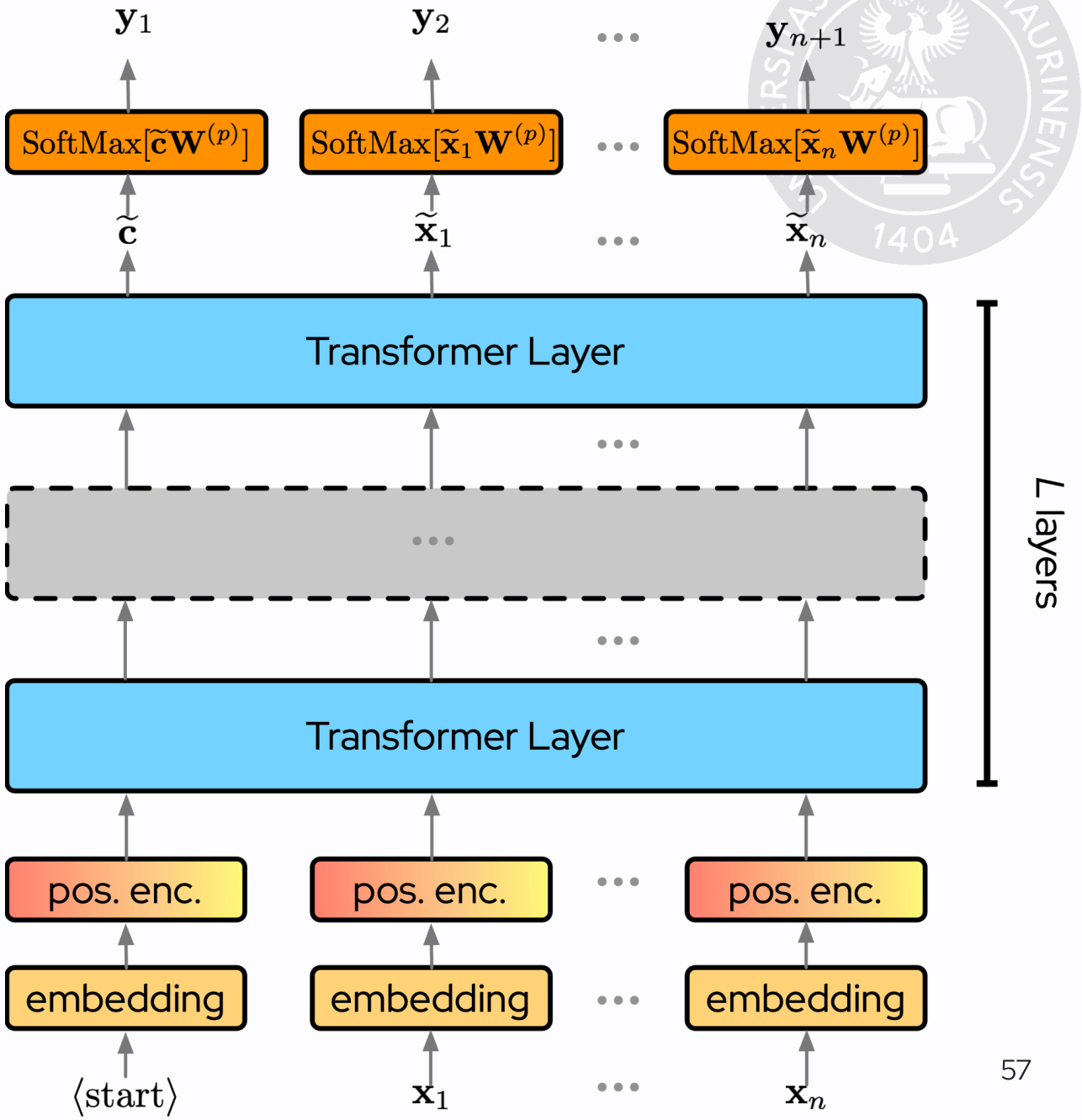
where the conditional distributions $p(\mathbf{x}_n | \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n-1})$ are expressed by a transformer network learned from data.

The architecture consists of a **stack of transformers** that take a sequence of tokens and produce a sequence $\widetilde{\mathbf{x}}_1, \widetilde{\mathbf{x}}_2, \ldots, \widetilde{\mathbf{x}}_n$ of dimensionality D as output.

A linear transformation followed by a softmax to compute the distribution over the K output tokens is then applied:

$$\mathbf{Y} = \text{Softmax}[\widetilde{\mathbf{X}}\mathbf{W}^{(p)}]$$

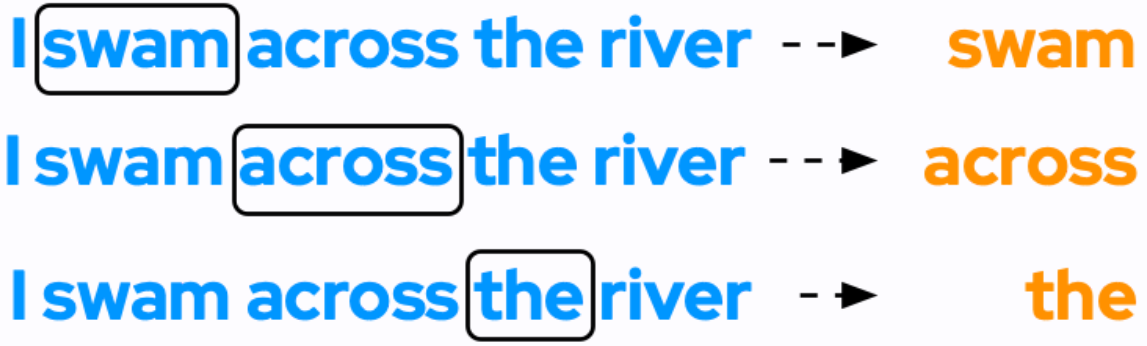
nei decoder ci serve un starting token



quello che vogliamo avere è: per la parola numero n la probabilità di un certo token è x

GPT models generate text by sampling from the distribution $p(\mathbf{x}_n|\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n-1})$ at each step n, **older predictions are used as input to the model to generate the next token**.

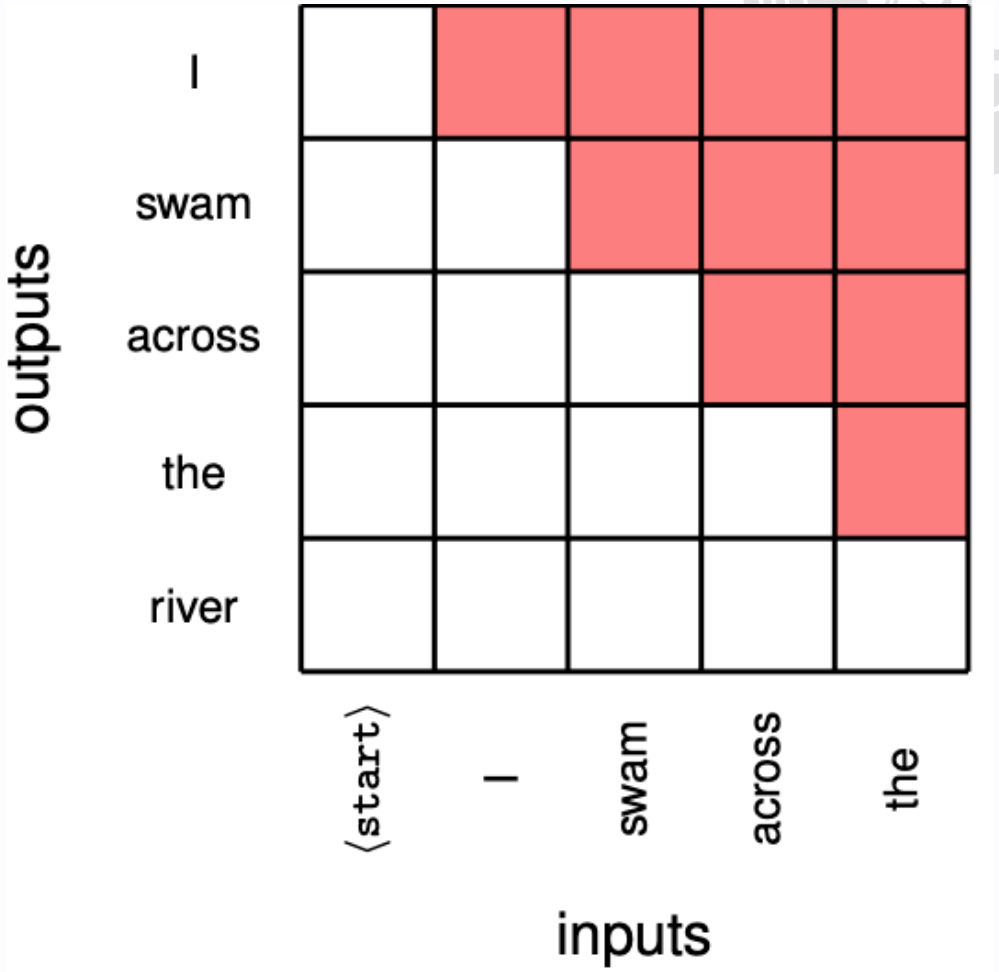
A problem with RNN and other classical methods for processing sequences is that, while **it would be nice to process entire sequences at once**, the model would be able to see the future tokens, which would make the task trivial.



posso "vedere" solo le parole che arrivano dopo

The attention mechanism gives a way to **mask out the future tokens** when computing the attention weights, thus allowing parallel processing of whole sequence. The idea consists of two steps:

1. add a special token to the input sequence that represents the start of the sequence;
2. mask out (set to zero) the attention weights for the future tokens.



Specifically, we can compute the attention weights using $\mathbf{QK}^\top$, as before, but then we can set the attention weights to zero for all future tokens.

Recall that the attention weights are computed as:

$$(\mathbf{QK}^\top)_{nm} = \text{attention weight between tokens } n \text{ and } m$$

if we multiply (elementwise) this matrix by a mask matrix $\mathbf{M}$ that has $-\infty$ in the upper triangular part, we obtain the masked attention weights:

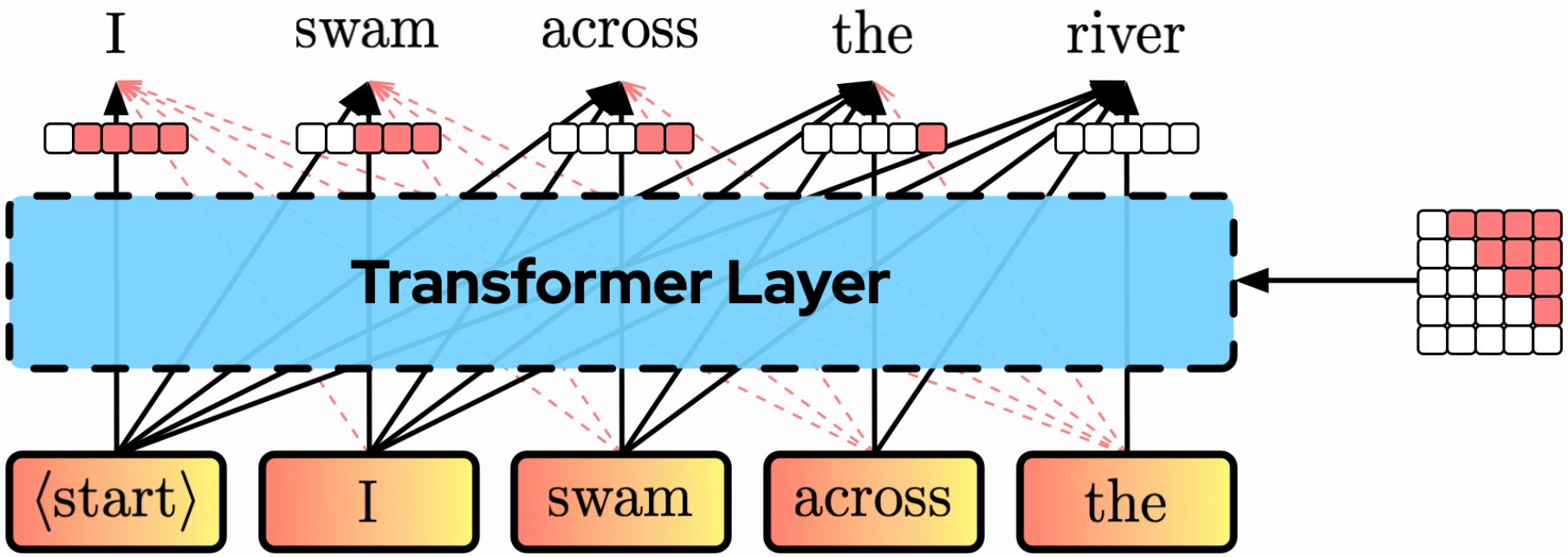
$$\mathbf{Y} = \text{Softmax}\left[\frac{\mathbf{QK}^\top}{\sqrt{D_k}} \circ \mathbf{M}\right]\mathbf{V}$$

we obtain that $\mathbf{y}_1$ depends only on $\mathbf{x}_1$, $\mathbf{y}_2$ depends on $\mathbf{x}_1$ and $\mathbf{x}_2$, and so on.

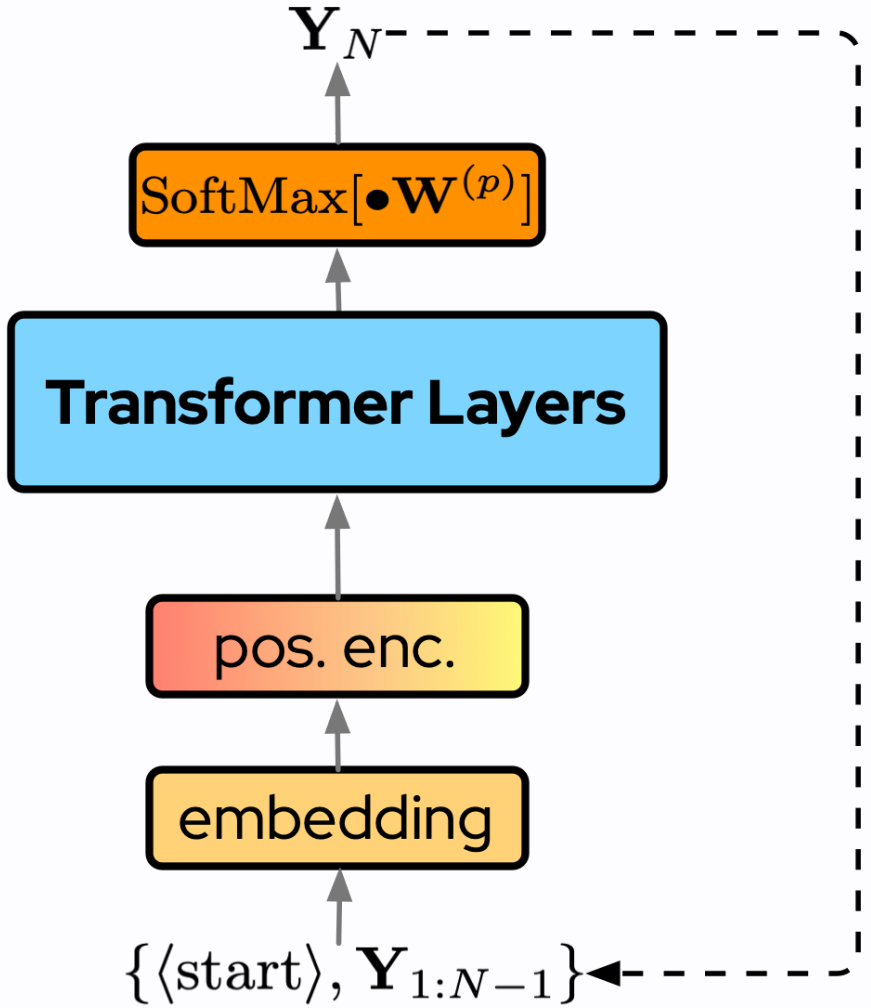
To allow processing multiple sequences at once, we would like to collect them into a single tensor, but this would require that all sequences have the same length.

To solve this issue, we can **pad the sequences** to the same length, using a special token to represent the padding.

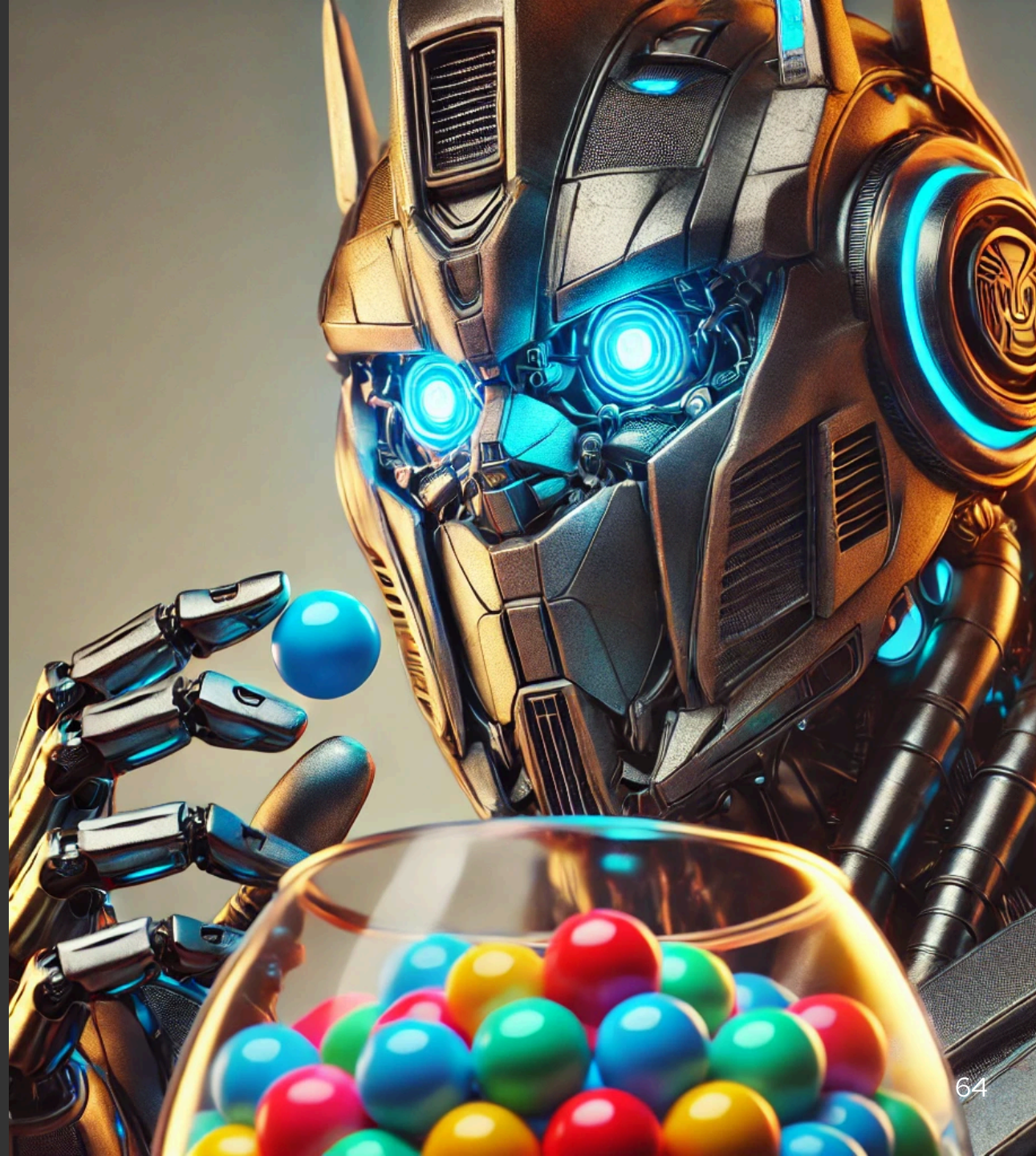
We can then use a mask matrix $\mathbf{M}$ that has $-\infty$ in the positions of the padding tokens, so that the attention weights for the padding tokens are zero.


I
swam
across
the
river
Transformer Layer
⟨start⟩
I
swam
across
the

Generation


$\mathbf{Y}_N$
$\mathrm{SoftMax}[\bullet \mathbf{W}^{(p)}]$
Transformer Layers
pos. enc.
embedding
$\{\langle \mathrm{start} \rangle, \mathbf{Y}_{1:N-1}\}$

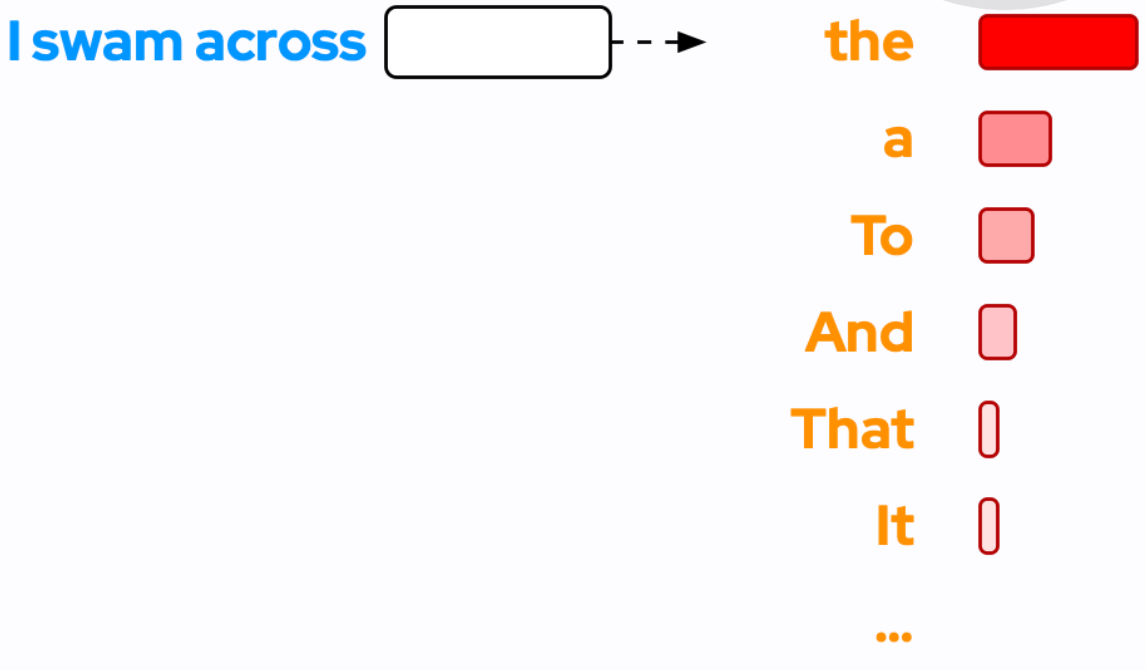
Sampling strategies



Once we have computed the distribution $p(\mathbf{x}_n|\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n-1})$, we can sample the next token from it.

The **greedy strategy**, simply chooses the token with the highest probability.

Question: is the produced sequence optimal? Meaning: is it the most probable sequence?

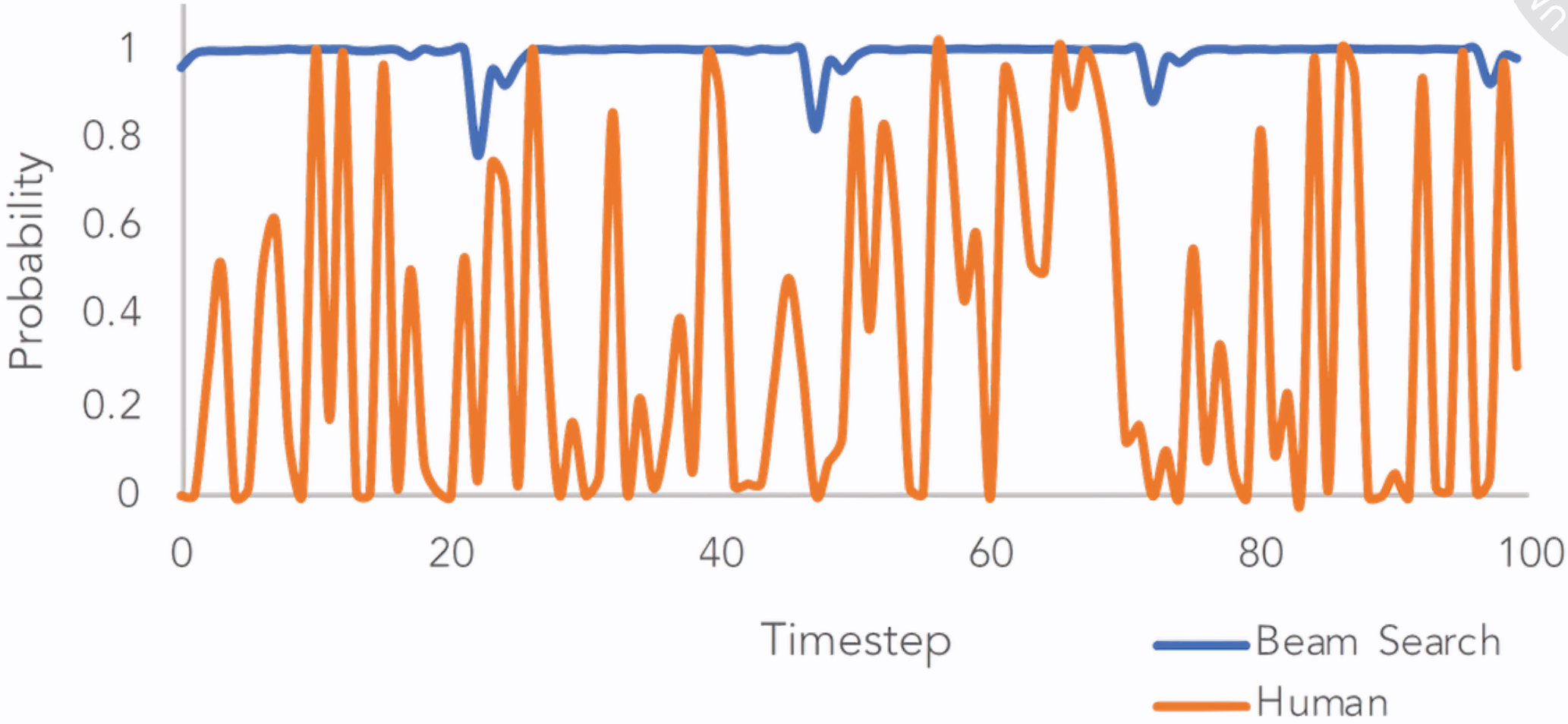


To find the most probable sequence, we would like to maximize the joint distribution over all tokens:

$$p(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \prod_{n=1}^{N} p(\mathbf{y}_n | \mathbf{y}_1, \ldots, \mathbf{y}_{n-1}).$$

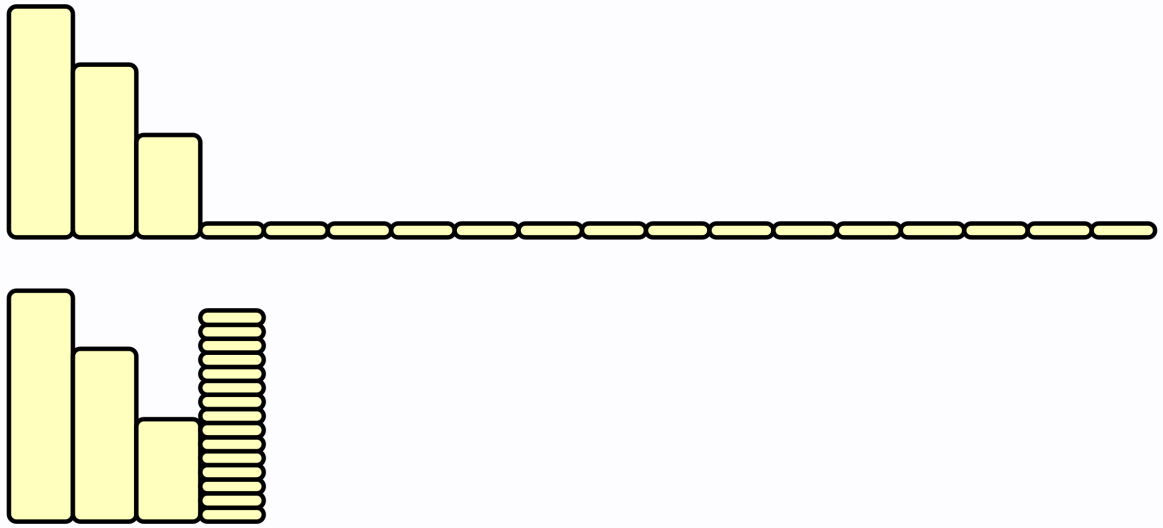
Higher probability sequences can be generated using a **beam search** strategy, which keeps track of the k most probable sequences at each step.

However: Most likely sequences are not necessarily the most human-like sequences.



If, on the other hand, we sample from the full distribution, we can generate sequences that are nonsensical or that are not grammatically correct.

This arises from the typically very large size of the token dictionary, which has a **long tail of tokens that have very low probability**.



Top-K sampling mitigates this issue by sampling from a **truncated distribution** that only includes the top-K most probable tokens.

Similarly, **top-p sampling** (or **nucleus sampling**) samples from a truncated distribution that includes the smallest set of tokens whose cumulative probability exceeds a threshold p.

A *soft* version of top-K sampling is **temperature scaling**, which scales the pre-activation values by a *temperature* parameter T before applying the softmax function:

$$y_i = \frac{\exp(a_i/T)}{\sum_j \exp(a_j/T)}$$

when:

- $T \to 0$ the distribution becomes more peaked around the most probable tokens;
- $T = 1$ the distribution is the same as the original softmax distribution;
- $T \to \infty$ the distribution becomes uniform across all states.

The **transformer architecture** has been used to build models that have achieved state-of-the-art performance on a wide range of tasks that deal with processing sequences.

In general, when processing a sequence, three tasks are possible:

1. **prediction**: given a sequence make a prediction about its nature (e.g., classify it, or output a numerical value). Transformers are used as encoders of the sequence. *Example:* sentiment analysis. see Bishop et al., §12.3.3
2. **generation**: given a single input, generate a sequence of tokens. Transformers are used as decoders of the sequence. *Example:* generate a caption for an image.
3. **translation** a.k.a. **seq2seq**: given a sequence, generate a new sequence. Transformers are used for both encoding and decoding the sequences. *Example:* machine translation. see Bishop et al., §12.3.4

Encoder transformers

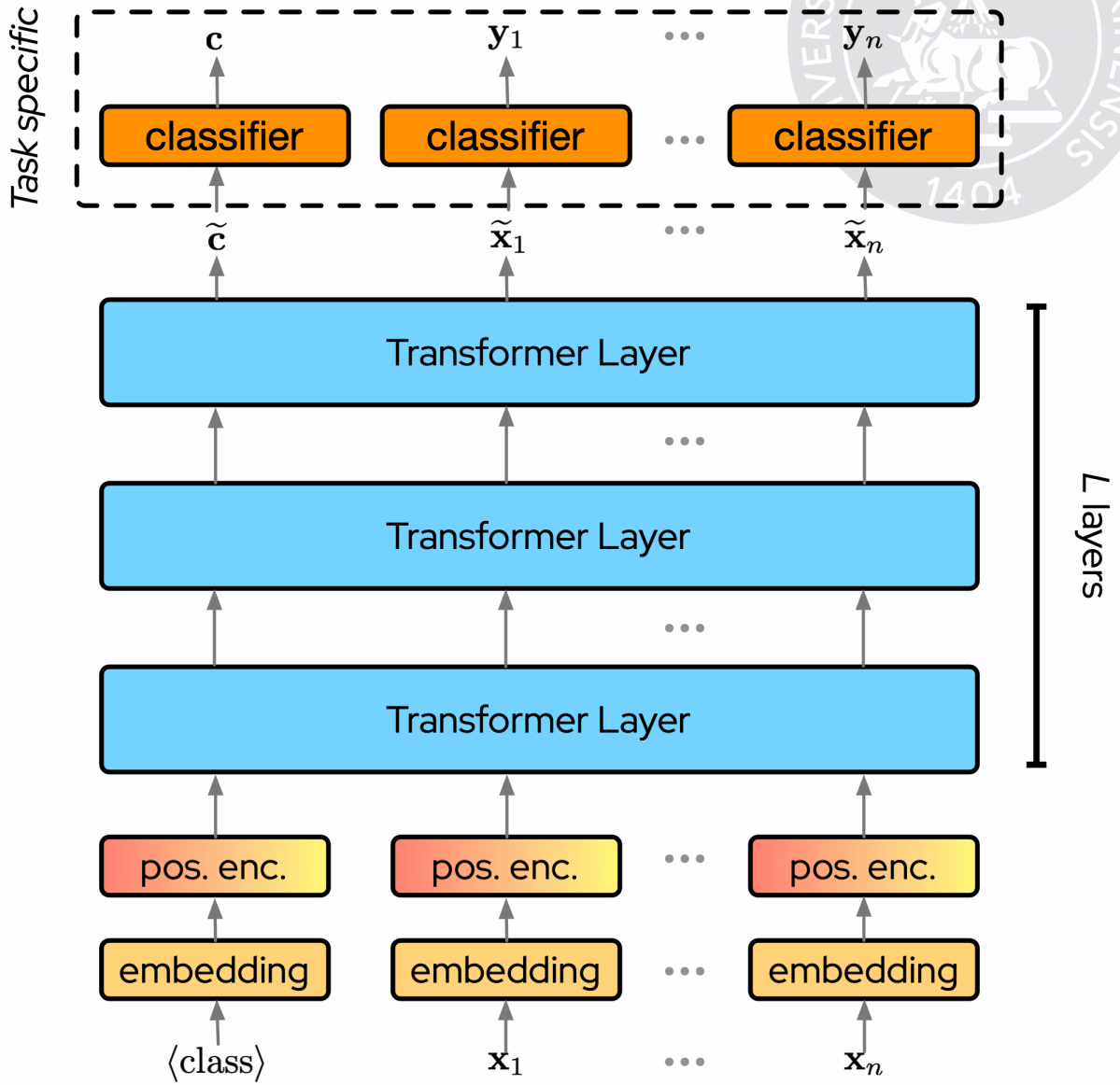
Encoder transformers are used to process sequences of tokens and produce contextual embeddings that can be used to produce a **fixed-size representation of the sequence**.

The class token is ignored during pre-training.

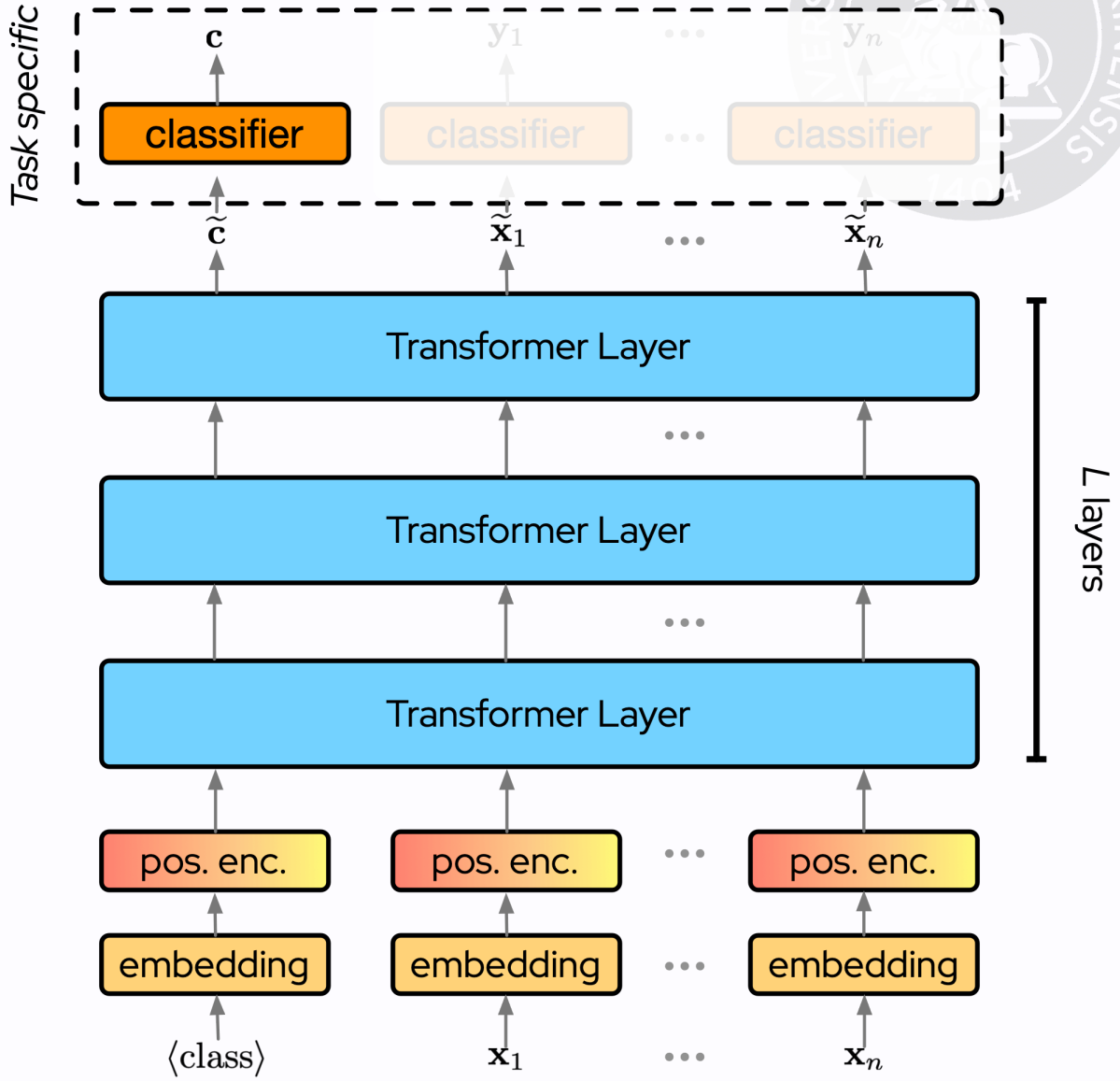
A randomly chosen subset (say 15%) of the tokens are replaced by a special token, the mask token, and the model is trained to predict the original tokens from the masked tokens.

Example

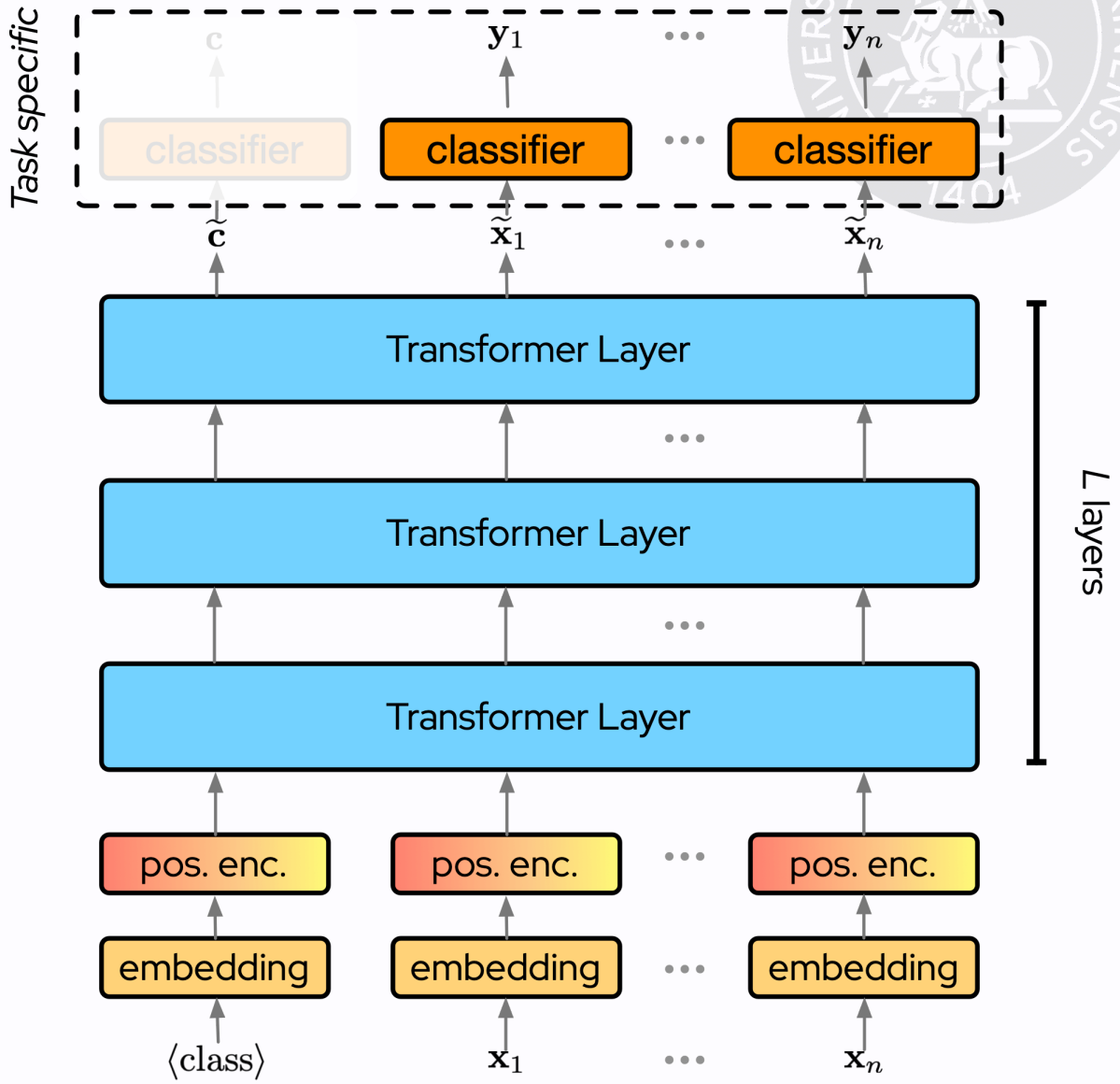
I $\langle \text{mask} \rangle$ *across the river to get to the* $\langle \text{mask} \rangle$ *bank.*



For sequence classification tasks, the class token is used to produce a fixed-size representation of the sequence, which is then passed through a linear layer to produce the output.



For token classification tasks, the output of the transformer is passed through a linear layer (or a more complex classifier) for each token to produce the output.



The **transformer architecture** has been used to build models that have achieved state-of-the-art performance on a wide range of tasks that deal with processing sequences.

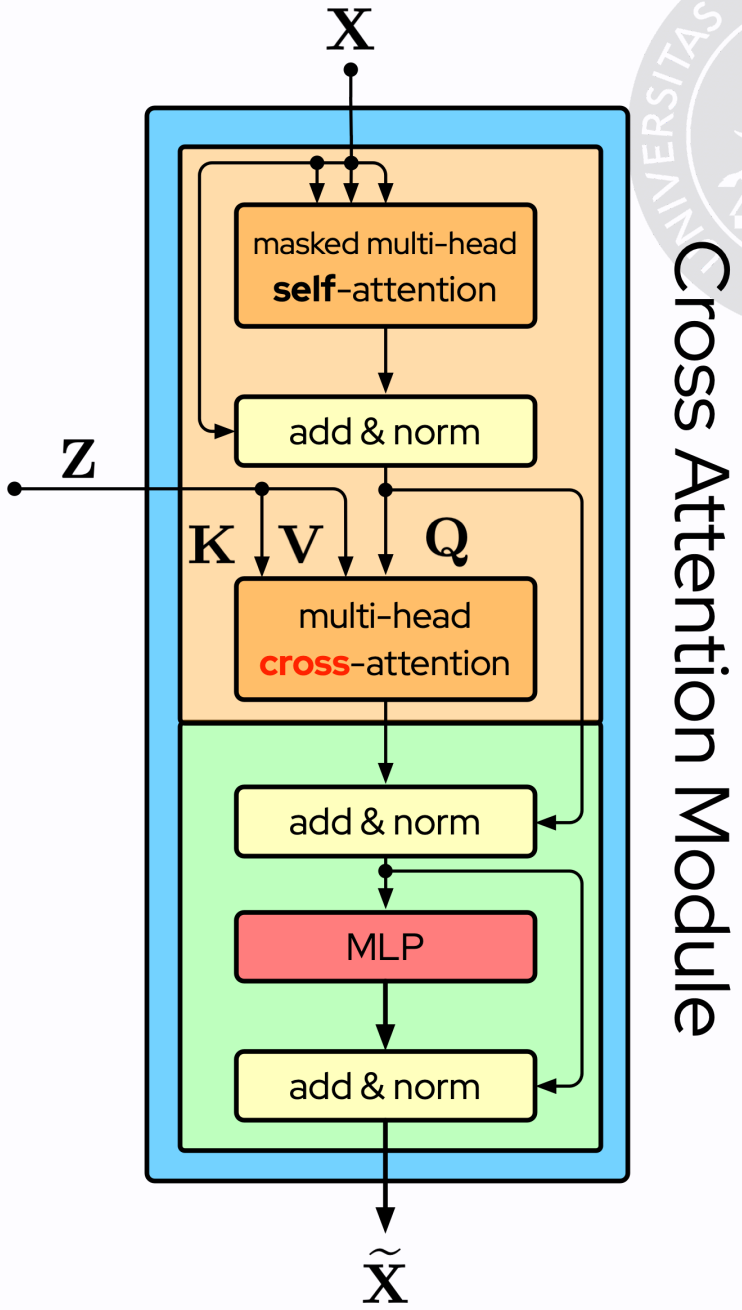
In general, when processing a sequence, three tasks are possible:

1. **prediction**: given a sequence make a prediction about its nature (e.g., classify it, or output a numerical value). Transformers are used as encoders of the sequence. *Example:* sentiment analysis. see Bishop et al., §12.3.3
2. **generation**: given a single input, generate a sequence of tokens. Transformers are used as decoders of the sequence. *Example:* generate a caption for an image.
3. **translation** a.k.a. **seq2seq**: given a sequence, generate a new sequence. Transformers are used for both encoding and decoding the sequences. *Example:* machine translation. see Bishop et al., §12.3.4

Cross-attention

To perform translation, the transformer architecture uses a **cross-attention mechanism** that allows the decoder to mix information from the encoder with the information it has generated so far.

In the picture $\mathbf{Z}$ is the output of the encoder, which is fed with the entire input sequence.



Encoder-decoder architecture