Tokens, embeddings and transformers

Danilo Comminiello and Simone Scardapane

Neural Networks 2026/2027

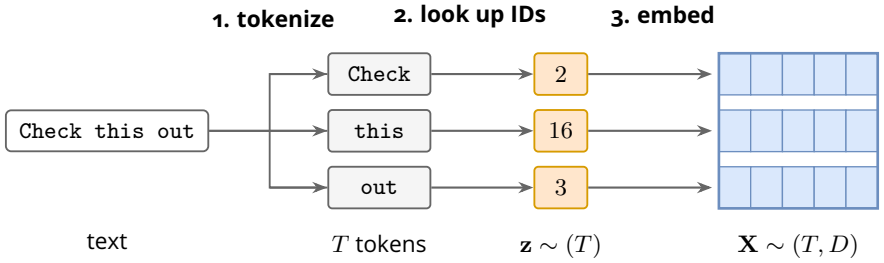


SAPIENZA
UNIVERSITÀ DI ROMA

1 From text to embeddings

Handling text data

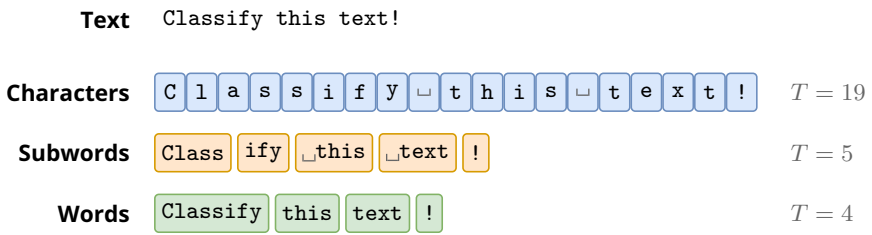
Text must be preprocessed to be handled by a neural network. We turn it into a matrix in three steps:



By the end of this procedure we have a matrix $\mathbf{X} \sim (T, D)$ of T tokens, each represented by D features (we write $\mathbf{X} \sim (T, D)$ for a matrix with T rows and D columns).

Step 1: choosing the tokens

We first choose how to split (**tokenize**) the text, e.g., by character, by byte, by **subword** or by word:



Larger **tokens** give shorter sequences, but we need many more of them to cover all possible texts. Byte tokenization works like character tokenization on the 256 byte values of UTF-8 text.

Subwords from the GPT-4 tokenizer (`cl100k_base`), via tiktoken.

Step 1: building the vocabulary

The set of tokens we can handle is the **vocabulary** or **dictionary**. Its size V depends on the type of token:

Characters or bytes
A small fixed set, e.g., the 256 byte values.

Words
The V most frequent words of a training corpus, plus a special *unknown* token.

Subwords
Reusable pieces learned from a corpus, e.g., with **byte pair encoding** (BPE).

Larger tokens need a larger vocabulary: a few hundred characters, but tens of thousands of words or subwords.

Subword embeddings

Word-level vocabularies can miss unseen words. Character tokenization produces longer sequences. With **subword tokenization**, we reuse pieces across words, for example (illustrative):

$$\begin{aligned} \texttt{playing} &\longmapsto (\texttt{play}, \texttt{ing}) \\ \texttt{played} &\longmapsto (\texttt{play}, \texttt{ed}) \end{aligned}$$

Each piece has an ID and a row in $\mathbf{E} \sim (V, D)$. Both words share the embedding for `play`.

Unseen words can split into known pieces, and frequent words can stay whole. Learned pieces need not match linguistic word parts.

Sennrich, Haddow and Birch (2016), *Neural Machine Translation of Rare Words with Subword Units*, ACL.

Byte pair encoding

We can learn the vocabulary with **byte pair encoding** (BPE):

1. Start the vocabulary with the single characters.
2. Merge the most frequent adjacent pair into a new token, **added to the vocabulary**.
3. Repeat until the vocabulary reaches the chosen size V.

Toy corpus: $\texttt{hug} \times 5$, $\texttt{hugs} \times 3$, $\texttt{pug} \times 2$. We count pairs within words:

$$\begin{aligned} (\texttt{u}, \texttt{g}) &\longmapsto \texttt{ug} && 10 \text{ occurrences} \\ (\texttt{h}, \texttt{ug}) &\longmapsto \texttt{hug} && 8 \text{ occurrences} \end{aligned}$$

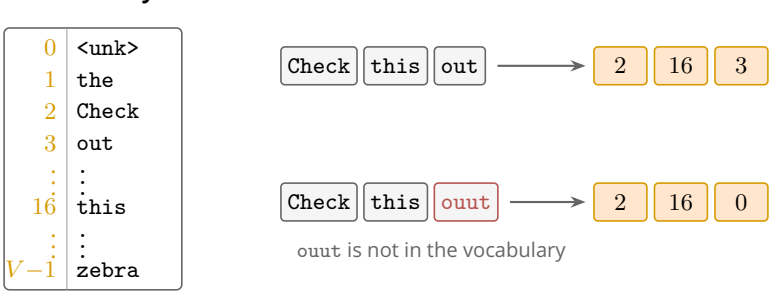
The vocabulary grows from $\{\texttt{g}, \texttt{h}, \texttt{p}, \texttt{s}, \texttt{u}\}$ to $\{\ldots, \texttt{ug}\}$ and then $\{\ldots, \texttt{ug}, \texttt{hug}\}$.

On new text, we apply the fixed merge rules in their learned order. **Byte-level BPE** starts from 256 byte values, so it can represent any UTF-8 text.

Hugging Face: Byte-Pair Encoding tokenization.

Step 2: from tokens to IDs

Each token in the text is converted into a discrete ID $z_t \in \{0, \dots, V-1\}$, its position in the vocabulary, giving us an integer sequence $\mathbf{z} \sim (T)$:



A word outside the vocabulary, such as a typo, becomes the unknown token.

Step 3: dense embeddings

For embedding we learn an **embedding matrix** $\mathbf{E} \sim (V, D)$, with one row for each of the V tokens in the vocabulary. Embedding element z_t of the sequence is just a row lookup:

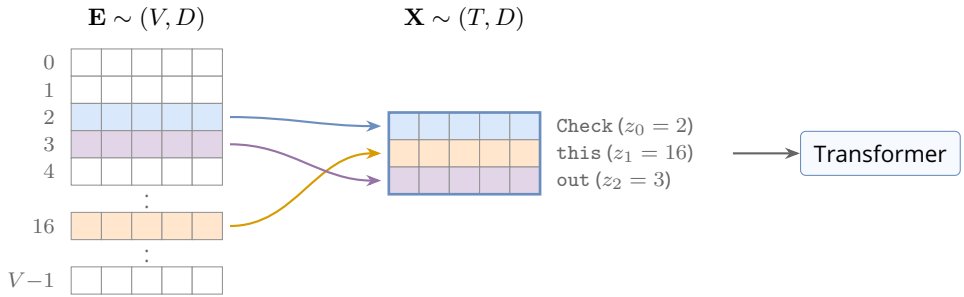
$$\mathbf{X}_{t,:} = \mathbf{E}_{z_t,:}$$

We initialize the embeddings randomly and train them with the network, or take them from some pre-trained model or algorithm. For example, GloVe and Word2Vec were two classic algorithms for this purpose.

Pre-trained embeddings also inherit the biases of their training text: for example, word analogies can reproduce gender stereotypes about professions.

Bolukbasi et al. (2016), *Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings*, NeurIPS.

Visualizing the embedding procedure



For $\mathbf{z} = (2, 16, 3)$, each ID selects one row of $\mathbf{E}$. With custom embeddings, we train $\mathbf{E}$ together with the rest of the network.

Sequence padding

Two texts may have different lengths. Their embedded matrices $\mathbf{X}_1 \sim (T_1, D)$ and $\mathbf{X}_2 \sim (T_2, D)$ share the same feature dimension D.

For $T_1 > T_2$, we can build a padded mini-batch:

$$X = \mathrm{stack}\left(\mathbf{X}_1, \begin{bmatrix}\mathbf{X}_2 \\ \mathbf{0}\end{bmatrix}\right), \qquad X \sim (2, T_1, D).$$

We pad to the maximum sequence length in the mini-batch. The added zero rows are placeholders.

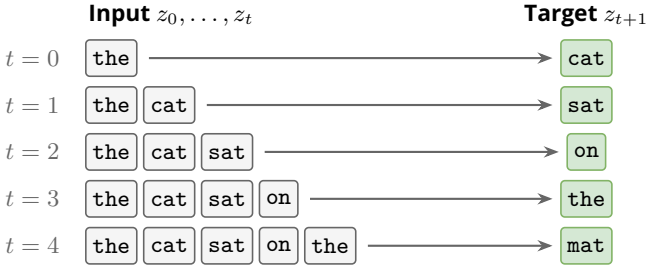
We keep a **validity mask** of real positions, so padded positions do not contribute to pooling or to the loss.

2 Language models

Next-token prediction

The recurrent lectures predicted the next value of a time series from its history. We now repeat this for text (**next-token prediction**): a sequence model (e.g., an RNN) maps $\mathbf{X}$ to $\mathbf{Y} \sim (T, C')$, and at each position a vocabulary classifier $\mathbf{p}_t = \mathrm{softmax}(g(\mathbf{Y}_{t,:})) \sim (V)$ predicts the following token z_{t+1}.

The targets come from the text itself, without human labels: the task is **self-supervised**, and a text of $T + 1$ tokens gives T examples:



Training a next-token predictor

We train on a **shifted target**: the input is $z_0, \ldots, z_{T-1}$ and the target is the same text moved by one position, $z_1, \ldots, z_T$. Each prediction $\mathbf{p}_t$ is compared with the next ID, averaging the cross-entropy over the T input positions:

$$\mathcal{L} = -\frac{1}{T} \sum_{t=0}^{T-1} \log \mathbf{p}_{t\, z_{t+1}}.$$

During training we feed the true tokens (**teacher forcing**), as in the open-loop setting of the recurrent lectures. The model must be causal, so that $\mathbf{p}_t$ never sees its own target z_{t+1}.

A probabilistic view

Consider the probability $p(\mathbf{z})$ of a whole sequence. By the chain rule of probability, as for time series in the recurrent lectures, we can always write it as

$$p(\mathbf{z}) = \prod_{t=0}^{T-1} p(z_{t+1} \mid z_0, \ldots, z_t),$$

where z_0 is a fixed start token, so it needs no factor. A causal model estimates every factor with the same network, $p(z_{t+1} \mid z_0, \ldots, z_t) \approx \mathbf{p}_{t\, z_{t+1}}$.

Our loss is then $\mathcal{L} = -\frac{1}{T} \log p(\mathbf{z})$: training maximizes the likelihood of the training texts. A model of $p(\mathbf{z})$ over text is a **language model**. Since we can sample new sequences from it, it is also a **generative model**.

Section 8.4 of *Alice's Adventures in a Differentiable Wonderland*.

Autoregressive models

The same next-token predictor can generate text in an **autoregressive** way, as in the free-running forecasts of the recurrent lectures.

We prompt it with the beginning of the sequence and sample the next token from the predicted distribution. We concatenate its ID to the input and call the model again, continuing one token at a time. For example:

`Check` $\longrightarrow$ `Check this` $\longrightarrow$ `Check this out` ...

The input now includes the model's own sampled tokens. As in the linear models lecture, we can sample with a **temperature** τ: a small τ gives predictable text, a large τ more varied text.

During training, all the input tokens are known in advance. Generation must wait for each sampled token before predicting the following one.

Autoregressive generation

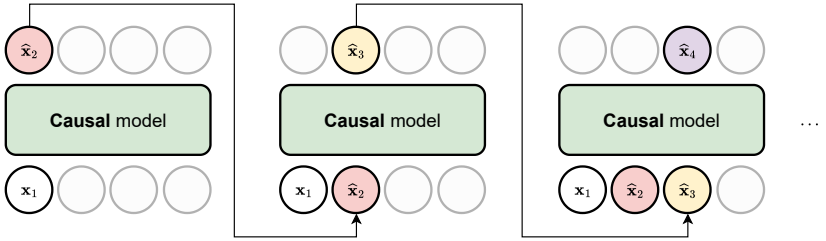


Figure 1: The hat marks a generated value: $\hat{x}_{t+1}$ is the model's prediction for position $t+1$, which is fed back as the next input. For text, we feed back sampled token IDs. Gray circles mark unused positions.

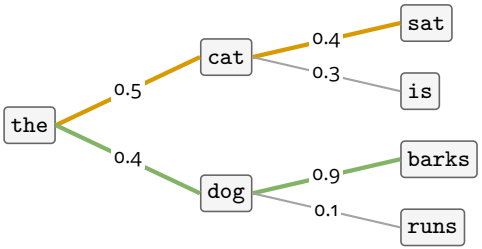
Decoding strategies

The rule we use to pick each token from $\mathbf{p}_t$ is called **decoding**:

- **Greedy decoding** always takes the most probable token. Since $p(\mathbf{z})$ is a product of many factors, a likely token now can lead to an unlikely sequence later.
- **Sampling** draws the token from $\mathbf{p}_t$, with a temperature τ ($\tau \to 0$ recovers greedy). Top-k and top-p sampling first discard the unlikely tail.
- **Beam search** keeps the k partial sequences with the highest $\log p$, extends each of them, and keeps the best k again.

Holtzman et al. (2020), *The Curious Case of Neural Text Degeneration*, ICLR.

Greedy decoding and beam search



$p(\textbf{sequence})$

0.20 greedy

0.15

0.36 beam, $k = 2$

0.04

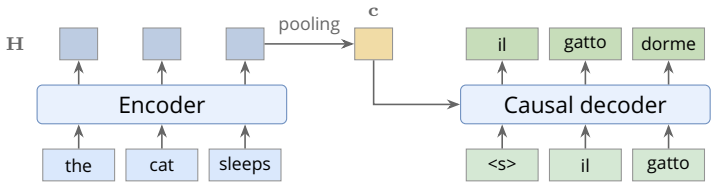
Keeping `dog` lets beam search find a more likely sequence.

3 Designing the transformer

- Sequence-to-sequence models
- From recurrence to attention
- Self-attention
- Multi-head attention
- The transformer block
- Positional embeddings
- The complete transformer model
- Computational considerations

Sequence to sequence

In **machine translation**, the input and the output are two sequences of different lengths, and the t-th output token has no fixed relation to the t-th input token. Around 2014, this was called a **sequence-to-sequence** (seq2seq) task. We can already solve it with the models we have seen:



An **encoder** (e.g., an RNN or a 1D CNN) processes the source sentence, and **pooling** summarizes it in a single vector $\mathbf{c}$. A causal **decoder** generates the translation autoregressively, receiving $\mathbf{c}$ at every step (e.g., concatenated to each input embedding).

Sutskever et al. (2014), *Sequence to sequence learning with neural networks*, NeurIPS; Cho et al. (2014), *Learning phrase representations using RNN encoder-decoder for statistical machine translation*, EMNLP.

From pooling to attention

The whole source sentence must fit in a vector $\mathbf{c}$ of fixed size, and the translations got worse on long sentences. Bahdanau et al. (2015) let the decoder compute a different summary at every step t:

$$\mathbf{c}_t = \sum_i \alpha_{t,i}\, \mathbf{h}_i\,, \qquad \alpha_{t,i} = \mathrm{softmax}_i\big(a(\mathbf{s}_t, \mathbf{h}_i)\big)\,,$$

weight of source token i at step t

where $\mathbf{s}_t$ is the decoder's hidden vector at step t and a is a small MLP. With $\alpha_{t,i} = 1/T'$ we recover average pooling, but in general the weights depend on what the decoder is generating and on the content of each source token. This was the first **attention** layer: the model learned to align the words of the two languages without supervision.

Bahdanau et al. (2015), *Neural machine translation by jointly learning to align and translate*, ICLR.

In 2017, Vaswani et al. kept this encoder-decoder design but built it only from attention layers, hence the title *Attention is all you need*: this is the **transformer**. Today the two halves are also used on their own, with a task-specific head:

Sequence classification
An **encoder**: each output depends on the whole sequence (e.g. BERT, ViT).

Next-token prediction
A **decoder**: causal, as in a recurrent model (e.g. GPT).

Vaswani et al. (2017), *Attention is all you need*, NeurIPS.

Sequence to sequence today

GPT-3 showed that a large decoder alone can solve many sequence-to-sequence tasks: we write the input in the **prompt**, and the model continues it autoregressively.

```
Translate English to Italian:
the cat sleeps => il gatto dorme
```

Translation, summarization or question answering all become next-token prediction on a single sequence, with no separate encoder.

Something of the encoder remains: the prompt is known in advance, so the model processes all its tokens in a single parallel pass (the **prefill** stage), and only the output is generated one token at a time.

Brown et al. (2020), *Language models are few-shot learners*, NeurIPS.

Beyond recurrence

A recurrent model reads the tokens one at a time and summarizes the past in a state of fixed size, $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)$. This has two drawbacks:

- **Long-range dependencies:** information from token $t-k$ reaches token t through k state updates, and the gradients through many updates tend to vanish or explode.
- **Sequential computation:** $\mathbf{h}_t$ needs $\mathbf{h}_{t-1}$, so even with teacher forcing we cannot process the positions of a sequence in parallel.

Compare:

- *the **cat** is on the **table***;
- *the **cat**, which belonged to my mother, is on the **table***.

The relation between the two words is the same, but their distance is not. We want a layer where every output reads every token directly, and all outputs are computed in parallel.

Convolutions as a sum over offsets

Take a sequence $\mathbf{X} \sim (T, D)$ with rows $\mathbf{x}_t = \mathbf{X}_{t,:}$. Because transformers come from NLP, we call each row a **token** and D the **embedding dimension**.

A 1D convolution, the sequence version of the image convolutions of the CNN lecture, processes all tokens in parallel. Ignoring padding and the activation, with kernel size $s = 2k+1$:

$$\mathbf{h}_t = \sum_{j=-k}^{k} \mathbf{W}_{j+k,:,:}\, \mathbf{x}_{t+j}\,, \qquad \mathbf{W} \sim (s, D, D).$$

one matrix per offset

The kernel has one $D \times D$ matrix for each offset j, so its parameters grow linearly with s. Each output sees only a window of s tokens (**locality**), and the weight of a neighbor depends only on its offset, never on what the two tokens contain. We want to remove both limits while keeping the number of parameters independent of the length T.

Continuous convolutions

Instead of storing one matrix per offset, we *compute* it with a small trainable network $g : \mathbb{Z} \to \mathbb{R}^{D \times D}$ that takes the offset as input:

$$\mathbf{h}_t = \sum_{t'=0}^{T-1} g(t' - t)\, \mathbf{x}_{t'} \,.$$

$g(t' - t)$: weight computed from the offset

The kernel now covers the whole sequence with a fixed number of parameters. These **continuous convolutions** also handle inputs of variable size and resolution.

Romero et al. (2022), *Towards a General Purpose CNN for Long Range Dependencies in ND*, arXiv.

Non-local networks

A continuous convolution handles long-range dependencies, but they are still *regular*: the weight depends only on the offset $t' - t$. We can let it depend on the *values* of the tokens instead:

$$\mathbf{h}_t = \sum_{t'=0}^{T-1} \alpha(\mathbf{x}_t, \mathbf{x}_{t'})\ \mathbf{x}_{t'} \ .$$

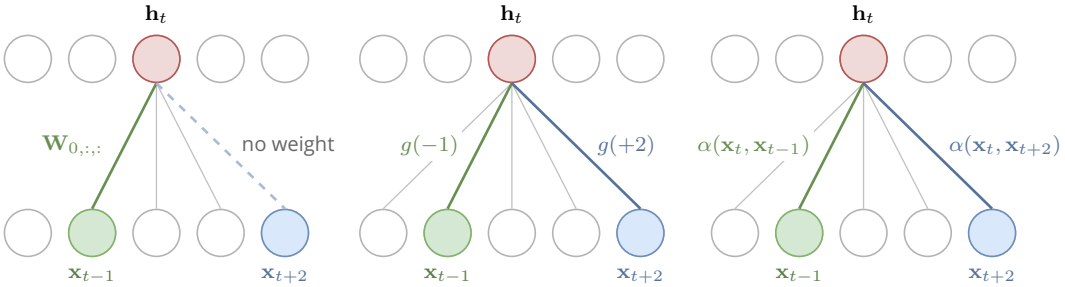
weight computed from the two tokens

This is a **non-local neural network**. With a suitable choice of the weighting function α, we will obtain the attention layer.[1]

[1] We could also combine the two ideas, with a weight $\alpha(\mathbf{x}_t, \mathbf{x}_{t'}, t' - t)$ that depends on both the tokens and their offset. We will return to this with *relative positional embeddings*.

Wang et al. (2018), *Non-local Neural Networks*, CVPR.

Three ways to weight a neighbor



(a) Convolution, $s = 3$ (b) Continuous convolution (c) Non-local network

Each edge weights an input token for the output $\mathbf{h}_t$: (a) one matrix per offset, inside the kernel only; (b) computed from the offset; (c) computed from the two tokens.

Choosing the weighting function

We now design α. Two simplifications make it cheap and stable:

- **Scalar weights:** α returns a number, not a $D \times D$ matrix, so $\mathbf{h}_t$ is a weighted average of the tokens.
- **Normalized weights:** for each t, the weights are positive and sum to one over t'. Each token has a *budget* of attention: raising one weight lowers the others.

Both follow from a **scoring function** s and a softmax over t':

$$\alpha(\mathbf{x}_t, \mathbf{x}_{t'}) = \frac{\exp\left(s(\mathbf{x}_t, \mathbf{x}_{t'})\right)}{\sum_{r=0}^{T-1} \exp\left(s(\mathbf{x}_t, \mathbf{x}_r)\right)}\,.$$

We call $\alpha(\mathbf{x}_t, \mathbf{x}_{t'})$ the **attention weight** of token t on token t'.

Dot-product attention

The most common score is the scaled dot product, $s(\mathbf{x}_t, \mathbf{x}_{t'}) = \mathbf{x}_t^\top \mathbf{x}_{t'} / \sqrt{D}$. All pairs come from one matrix product, which is fast on a GPU. Putting everything together:

$$\mathbf{h}_t = \sum_{t'=0}^{T-1} \underbrace{\mathrm{softmax}_{t'}}_{\text{normalized over } t'} \left(\frac{\mathbf{x}_t^\top \mathbf{x}_{t'}}{\sqrt{D}} \right) \mathbf{x}_{t'} \,.$$

This is already an attention layer, without any parameters.[1]

[1] Why $\sqrt{D}$? If the elements of $\mathbf{x}_t$ and $\mathbf{x}_{t'}$ ($t' \neq t$) are independent with mean 0 and variance σ^2, then $\mathrm{Var}(\mathbf{x}_t^\top \mathbf{x}_{t'}) = D\sigma^4$ (check!). Without the scaling, the scores grow with D and a single large score saturates the softmax.

Queries, keys and values

So far the same vector plays three roles. We give each role its own trainable projection:

$$\mathbf{q}_t = \mathbf{W}_q^\top \mathbf{x}_t, \qquad \mathbf{k}_t = \mathbf{W}_k^\top \mathbf{x}_t, \qquad \mathbf{v}_t = \mathbf{W}_v^\top \mathbf{x}_t,$$

with $\mathbf{W}_q, \mathbf{W}_k \sim (D, D_k)$ and $\mathbf{W}_v \sim (D, D_v)$. The complete **self-attention** (SA) layer is:

$$\mathbf{h}_t = \sum_{t'=0}^{T-1} \mathrm{softmax}_{t'} \left(\frac{\mathbf{q}_t^\top \mathbf{k}_{t'}}{\sqrt{D_k}} \right) \mathbf{v}_{t'} .$$

The scores are no longer necessarily symmetric. Value projection by $\mathbf{W}_v$ allows for feature mixing.

Self-attention in matrix form

Stacking the tokens in $\mathbf{X} \sim (T, D)$, the projections are three matrix products:

$$\underset{(T,D_k)}{\mathbf{Q}} = \mathbf{X}\mathbf{W}_q, \qquad \underset{(T,D_k)}{\mathbf{K}} = \mathbf{X}\mathbf{W}_k, \qquad \underset{(T,D_v)}{\mathbf{V}} = \mathbf{X}\mathbf{W}_v.$$

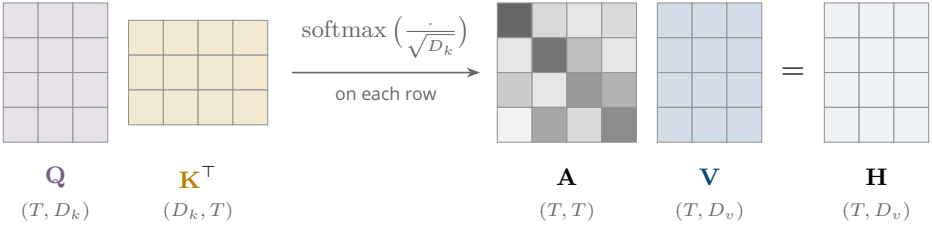
The whole layer is then:

$$\underset{(T,D_v)}{\mathbf{H}} = \operatorname{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{D_k}}\right)\mathbf{V}.$$

attention matrix $\mathbf{A} \sim (T, T)$, softmax on each row

Row t of $\mathbf{A}$ holds the attention weights of token t. In a batch $\mathbf{X} \sim (B, T, D)$, tokens attend only within their own sequence. The need to compute T^2 entries for $\mathbf{A}$ is a potential bottleneck for long sequences (more on this later on).

Visualizing self-attention



Each row of $\mathbf{A}$ (darker means larger weight) mixes the rows of $\mathbf{V}$ into one row of $\mathbf{H}$. The projections from $\mathbf{X}$ are not shown.

The dictionary analogy

The names come from (programming) dictionaries. A Python dictionary stores *(key, value)* pairs, and `d[q]` returns the value whose key is exactly `q`, or fails.

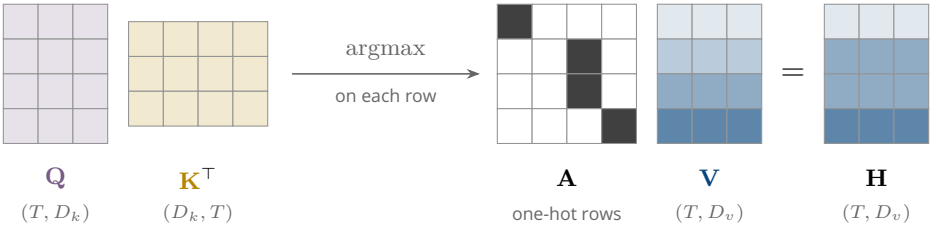
A simple variant (often called **hard attention**) instead compares the query with every key, and returns the value of the most similar one. With vectors and dot-product similarity, this is:

$$\mathbf{h}_t = \mathbf{v}_{t^*}, \qquad t^* = \arg\max_{t'} \mathbf{q}_t^\top \mathbf{k}_{t'} .$$

This has a gradient of zero almost everywhere. Replacing the argmax with a softmax returns a blend of all values, weighted by similarity: that is self-attention.

The dictionary is built from the sequence itself: every token writes one *(key, value)* pair, and every token reads with its query.

Hard attention as a dictionary



Each query retrieves the value of its best-matching key, like `d[q]`: every row of $\mathbf{H}$ is a copy of one row of $\mathbf{V}$. Small changes to the scores leave the argmax unchanged, so the gradient is zero almost everywhere.

Trainable keys and values

Consider a variant in which we parameterize explicitly the keys and values as trainable matrices $\mathbf{K} \sim (M, D)$ and $\mathbf{V} \sim (M, D_v)$ (ignoring $\mathbf{W}_q$ and the scaling):

$$\mathbf{H} = \text{softmax}\left(\mathbf{X}\mathbf{K}^\top\right)\mathbf{V}\,.$$

The rows of $\mathbf{K}$ and $\mathbf{V}$ store patterns retrieved by an attention-like operation (a **memory layer**). But this is also a **two-layer MLP** with a softmax activation function: MLPs can be seen as attention over trainable keys and values.

If we remove the softmax, we get a **linear associative memory** $\mathbf{H} = \mathbf{X}\mathbf{S}$, with $\mathbf{S} = \sum_i \mathbf{k}_i \mathbf{v}_i{}^\top$. Each read returns the matching value plus **interference** from all other pairs, which degrades quickly once M exceeds D.

Geva et al. (2021), *Transformer feed-forward layers are key-value memories*, EMNLP; A. Wang (2026), *You could have invented transformers from memory*, blog post.

Causal self-attention

For next-token prediction, $\mathbf{h}_t$ must not depend on tokens after t, as in a recurrent model. We add a **mask** to the scores before the softmax:

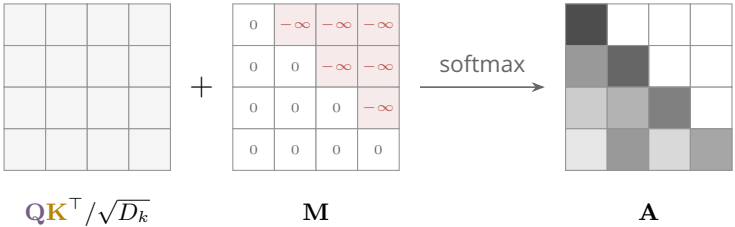
$$\mathbf{H} = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{D_k}} + \mathbf{M}\right)\mathbf{V}, \qquad \mathbf{M}_{t,t'} = \begin{cases} 0 & t' \le t \\ -\infty & t' > t \end{cases}$$

causal mask

Since $\exp(-\infty) = 0$, future tokens get zero weight and each row still sums to one. In code, $-\infty$ is a large negative number, or the scores are filled in place.

This is the single difference between our decoder and an encoder: without $\mathbf{M}$, every token sees the whole sequence.

Visualizing the causal mask



Adding $-\infty$ above the diagonal gives future tokens exactly zero weight. Multiplying the scores by a $0/1$ mask would not work, since $\exp(0) = 1$; zeroing $\mathbf{A}$ after the softmax would break the sum to one.

Multi-head attention

The SA layer can handle quasi-sparse dependencies (because of the softmax), and also dynamic ones (because of the attention function). However, what happens when a token depends on multiple subsets of tokens?

A common generalization in this case is **multi-head attention** (MHA). It works by computing $i = 1, \dots, h$ separate sets of queries, keys, and values:

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_{q,i}, \qquad \mathbf{K}_i = \mathbf{X}\mathbf{W}_{k,i}, \qquad \mathbf{V}_i = \mathbf{X}\mathbf{W}_{v,i},$$

$$\mathbf{H}_i = \mathrm{softmax}\left(\frac{\mathbf{Q}_i {\mathbf{K}_i}^\top}{\sqrt{D_k}}\right)\mathbf{V}_i\,.$$

Each set is a **head**, with its own attention matrix $\mathbf{A}_i$, so different heads can attend to different tokens. We now have $3h$ trainable matrices.

Combining the heads

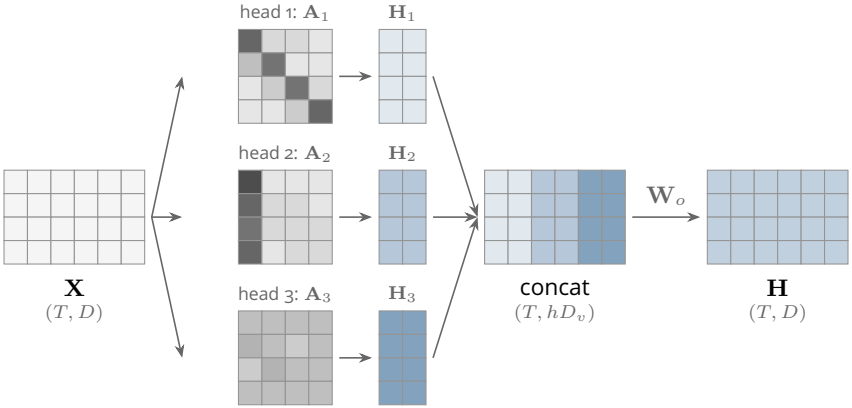
The previous operation has h separate outputs; we combine them by concatenation over the embedding dimension, and a final reprojection with a trainable output matrix $\mathbf{W}_o$:

$$\underset{(T,D)}{\mathbf{H}} = \underbrace{\begin{bmatrix}\mathbf{H}_1 & \cdots & \mathbf{H}_h\end{bmatrix}}_{(T,\,hD_v)} \mathbf{W}_o\,, \qquad \mathbf{W}_o \sim (hD_v, D)\,.$$

As hyperparameters, we typically choose the embedding dimension D and the number of heads h, and we set $D_k = D_v = D/h$ for all heads. The output has the same shape as the input, so MHA layers can be stacked.

With $D_k = D_v = D/h$, the four projections hold $4D^2$ **parameters for any** h: more heads split the same budget into smaller subspaces.

Visualizing multi-head attention



Three heads with $D_v = D/3$: each has its own attention pattern (here local, on the first token, and almost uniform). $\mathbf{W}_o$ mixes the concatenated outputs back into D features.

Attention in code

```
def mha(x, Wq, Wk, Wv, Wo, h): # x: (B, T, D)
    B, T, D = x.shape
    def heads(y): # (B, h, T, D/h)
        return y.view(B, T, h, D // h).transpose(1, 2)
    q, k, v = heads(x @ Wq), heads(x @ Wk), heads(x @ Wv)
    s = q @ k.transpose(-2, -1) / (D // h) ** 0.5 # (B, h, T, T)
    mask = torch.ones(T, T, dtype=torch.bool).triu(1)
    a = s.masked_fill(mask, float('-inf')).softmax(-1)
    out = (a @ v).transpose(1, 2).reshape(B, T, D) # concat heads
    return out @ Wo
```

This **naive** implementation of causal MHA stores all the attention matrices, (B, h, T, T) numbers. PyTorch's `F.scaled_dot_product_attention` (with `is_causal=True`) gives the same result computing $\mathbf{A}$ block by block, without storing it.

The transformer block

In transformers, the MHA layer is always used inside a more complex block, called the **transformer block**. Originally, this was composed of an MHA layer, two layer normalization operations, two residual connections, and a so-called **position-wise** network as follows:

1. Start with an MHA layer: $\mathbf{H} = \mathrm{MHA}(\mathbf{X})$.
2. Add a residual connection and a layer normalization operation: $\mathbf{H} = \mathrm{LayerNorm}(\mathbf{H} + \mathbf{X})$.
3. Apply an MLP g on each row: $\mathbf{F} = g(\mathbf{H})$.
4. Do again step 2: $\mathbf{H} = \mathrm{LayerNorm}(\mathbf{F} + \mathbf{H})$.

The output has shape (T, D) like the input: a transformer is a stack of these blocks.

Vaswani et al. (2017), *Attention is all you need*, NeurIPS.

Design of the block

The design of the block was mostly based on empirical considerations. Roughly speaking, steps (1)–(2) correspond to a **token mixing** operation, while steps (3)–(4) are a per-token update, with the same MLP applied to every token.

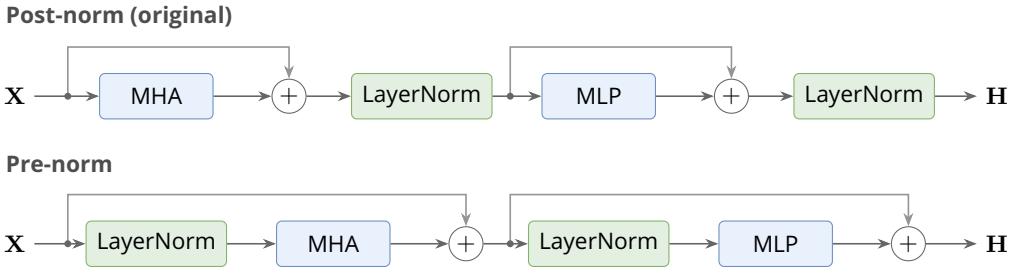
Token mixing (MHA)
Combines rows of $\mathbf{X}$, with weights that depend on the input.

Channel mixing (MLP)
Combines the features of each row, with the same weights for every token.

The intermediate MLP has two layers, with hidden dimension an integer multiple of D ($4D$ in the original model).

Vaswani et al. (2017), *Attention is all you need*, NeurIPS.

Pre- and post-normalized blocks



The original block is called **post-normalized**. A **pre-normalized** variant is also common due to it being simpler to train in most cases: $\mathbf{H} = \mathbf{X} + \mathrm{MHA}(\mathrm{LayerNorm}(\mathbf{X}))$.

The skip path carries $\mathbf{X}$, and the gradient, unchanged through all blocks. In post-norm, the gradient crosses every LayerNorm, and training needs a small learning rate in the first steps (**warmup**).

Xiong et al. (2020), *On layer normalization in the transformer architecture*, ICML.

Permutation matrices

In text, the t-th row of $\mathbf{X}$ represents a single token. To reason about reordering the tokens, we use permutation matrices.

A **permutation matrix** $\mathbf{P} \sim (T, T)$ has exactly one 1 in each row and in each column, and zeros elsewhere. Multiplying $\mathbf{PX}$ reorders the rows of $\mathbf{X}$.

For example:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \qquad \mathbf{P} \begin{bmatrix} \mathbf{x}_0^\top \\ \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \end{bmatrix} = \begin{bmatrix} \mathbf{x}_0^\top \\ \mathbf{x}_2^\top \\ \mathbf{x}_1^\top \end{bmatrix}.$$

Multiplying on the right, $\mathbf{AP}^\top$, reorders the columns in the same way. Since $\mathbf{P}^\top$ undoes the permutation, $\mathbf{P}^\top \mathbf{P} = \mathbf{I}$.

The MHA layer is equivariant to permutations

Without the causal mask, the MHA layer is *equivariant* to the ordering of the tokens:

$$\mathrm{MHA}(\mathbf{PX}) = \mathbf{P} \cdot \mathrm{MHA}(\mathbf{X})\,.$$

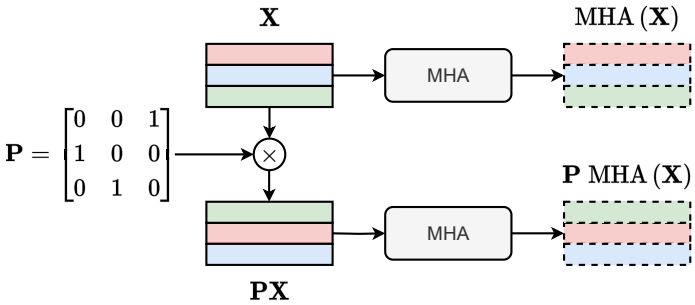
Follow one head (scaling omitted). The projections become $\mathbf{PQ}$, $\mathbf{PK}$, $\mathbf{PV}$, and the scores $\mathbf{PQK}^\top\mathbf{P}^\top$ have rows and columns reordered in the same way. A row-wise softmax does not care about this reordering, so:

$$\mathrm{softmax}(\mathbf{PQK}^\top\mathbf{P}^\top)\,\mathbf{PV} = \mathbf{P}\,\mathrm{softmax}(\mathbf{QK}^\top)\,\underbrace{\mathbf{P}^\top\mathbf{P}}_{=\mathbf{I}}\,\mathbf{V} = \mathbf{PH}\,.$$

Concatenation and $\mathbf{W}_o$, like the MLP and LayerNorm, act on each row, so the same holds trivially for the entire block. This is not a good property to have for sequences.

The causal mask breaks the symmetry: token t sees exactly $t+1$ tokens.

Visualizing permutation equivariance



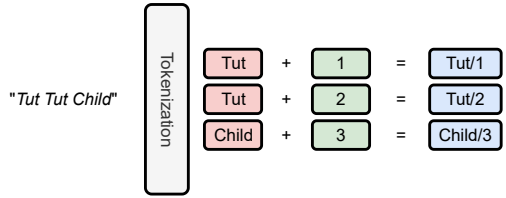
Permuting the input rows permutes the output rows in the same way. Each output depends on which tokens are present, not on their order: without positions, “dog” gets the same vector in “the dog bit the man” and “the man bit the dog”.

Positional embeddings

Before the first block, we sum (or concatenate) to the input $\mathbf{X}$ a matrix of **positional embeddings** $\mathbf{E} \sim (T, D)$:

$$\mathbf{X}' = \mathbf{X} + \mathbf{E}\,,$$

where each row $\mathbf{E}_{t,:}$ should *uniquely* encode the position t. Using this strategy, we 'break' the equivariance.



The same token at two different positions is now represented by different vectors.

Trainable positional embeddings

The simplest choice is to *learn* the positional embeddings with an embedding layer (`nn.Embedding`), as for the tokens:

- To each position t we associate an embedding vector of dimension D.
- The embeddings are trained with the rest of the network.

This is the strategy of BERT, GPT-2 and the Vision Transformer.

However, we need to fix the maximum length $T_{\max}$, and positions never seen in training have no embedding. To change the image resolution, ViT interpolates the trained embeddings.

Sinusoidal embeddings

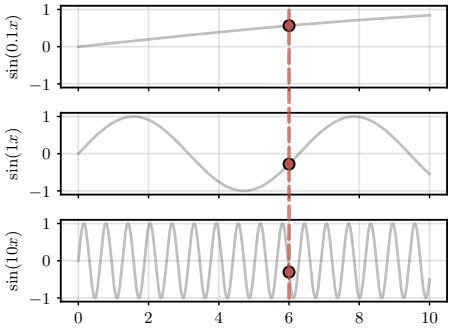
A sinusoid $\sin(\omega t)$ is a clock with frequency ω: it gives the position only modulo its period.

With multiple frequencies, we get a clock with D hands, each rotating at its own speed:

$$\mathbf{E}_{t,j} = \sin(\omega_j t), \qquad \omega_j = \frac{1}{b^{j/D}}\,.$$

The frequencies decrease geometrically, from a period of 2π tokens ($j = 0$) to about $b \cdot 2\pi$. The **base** b is a hyperparameter, larger than the longest sequence.

No parameters, and no maximum length fixed *a priori*.



Three hands at one position (red line).

Sines and cosines

The original transformer alternates sines and cosines of the same frequency, with $\omega_j = b^{-2j/D}$ and $b = 10000$:

$$\mathbf{E}_{t,2j} = \sin(\omega_j t), \qquad \mathbf{E}_{t,2j+1} = \cos(\omega_j t)\,.$$

Each pair is a point on a circle. Moving by p positions rotates it by the angle $\omega_j p$, whatever t is:

$$\mathbf{E}_{t+p,:} = \mathbf{E}_{t,:}\,\mathbf{T}(p), \qquad \mathbf{T}(p) = \operatorname{diag}\big(\mathbf{R}(\omega_0 p), \dots\big), \quad \mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Hence $\mathbf{E}_{t,:}\mathbf{E}_{t+p,:}^{\top} = \sum_j \cos(\omega_j p)$ depends only on p. Rotating queries and keys gives **rotary embeddings** (RoPE), shortly.

Vaswani et al. (2017), *Attention is all you need*, NeurIPS.

Relative positional embeddings

Another possibility is **relative positional embeddings**: we modify the attention function to make it depend on the distance $t - t'$ between tokens, as in the weight $\alpha(\mathbf{x}_t, \mathbf{x}_{t'}, t' - t)$ of non-local networks. Nothing is added to $\mathbf{X}$, and position enters every attention layer.

Rotate queries and keys

Rotary embeddings (RoPE): Llama, Qwen, Gemma, Mistral, and most current language models.

Add a bias to the scores

Linear biases (ALiBi, used in BLOOM and MPT), or the learned biases of T5.

Rotary positional embeddings (RoPE)

RoPE splits each query and key into $D_k/2$ pairs of components, and treats every pair as a point in the plane. Pair j of the query (and of the key) at position t is rotated by an angle proportional to t:

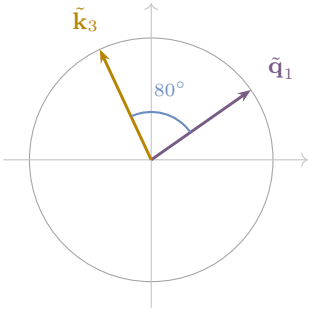
$$\begin{bmatrix} \tilde{q}_{t,2j} \\ \tilde{q}_{t,2j+1} \end{bmatrix} = \mathbf{R}(\omega_j t) \begin{bmatrix} q_{t,2j} \\ q_{t,2j+1} \end{bmatrix}, \qquad \mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Why pairs?
A rotation needs two dimensions. It keeps the norm, and two rotations add their angles.

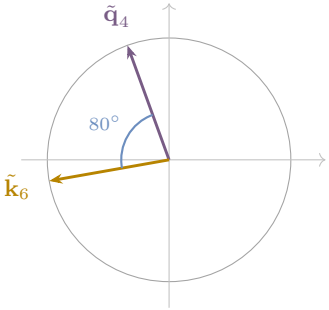
Why many frequencies?
One pair is a clock with one hand. With $\omega_j = b^{-2j/D_k}$, fast pairs separate nearby tokens, slow pairs distant ones.

Su et al. (2024), *RoFormer: Enhanced transformer with rotary position embedding*, Neurocomputing.

Visualizing RoPE



$t = 1, t' = 3$



$t = 4, t' = 6$

One pair of components, rotating by $25°$ per position. Moving both tokens by three positions rotates both vectors: the angle between them, hence the score, does not change.

RoPE for all pairs

Collecting the rotations of all pairs in a block-diagonal matrix:

$$\tilde{\mathbf{q}}_t = \mathbf{R}_t \mathbf{q}_t, \qquad \tilde{\mathbf{k}}_{t'} = \mathbf{R}_{t'} \mathbf{k}_{t'}, \qquad \mathbf{R}_t = \operatorname{diag}\left(\mathbf{R}(\omega_0 t), \ldots, \mathbf{R}(\omega_{D_k/2-1} t)\right).$$

Rotations compose, $\mathbf{R}_t^\top \mathbf{R}_{t'} = \mathbf{R}_{t'-t}$, so the score becomes:

position enters only through $t' - t$

$$s_{t,t'} = \frac{\tilde{\mathbf{q}}_t^\top \tilde{\mathbf{k}}_{t'}}{\sqrt{D_k}} = \frac{\mathbf{q}_t^\top \mathbf{R}_{t'-t} \mathbf{k}_{t'}}{\sqrt{D_k}}.$$

The values are not rotated, nothing is added to $\mathbf{X}$, and there are no parameters.

RoPE in code

```
def rope(x, base=10000.0): # x: (..., T, D_k)
    T, D = x.shape[-2:]
    w = base ** (-torch.arange(0, D, 2) / D) # (D_k/2,) frequencies
    ang = torch.arange(T)[:, None] * w # (T, D_k/2) angles
    cos, sin = ang.cos(), ang.sin()
    x1, x2 = x[..., 0::2], x[..., 1::2] # the pairs
    return torch.stack([x1 * cos - x2 * sin,
           x1 * sin + x2 * cos], -1).flatten(-2)

scores = rope(q) @ rope(k).transpose(-2, -1) / D_k ** 0.5
```

RoPE is applied in every attention layer, after the projections and before the scores. The base b sets the slowest rotation: recent models raise it (Llama 3 uses $b = 500{,}000$) to cover longer contexts.

Attention with linear biases (ALiBi)

ALiBi adds no embedding to $\mathbf{X}$; it subtracts from the scores a penalty proportional to the distance:

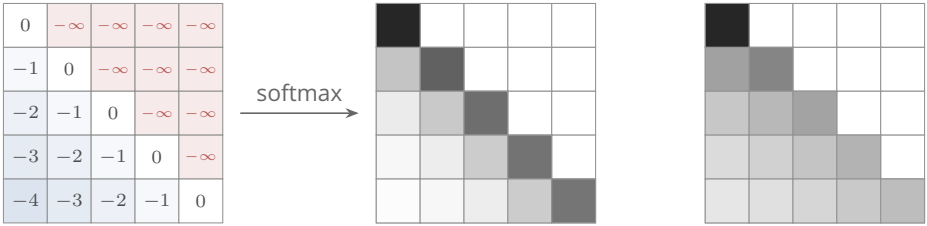
$$s_{t,t'} = \frac{\mathbf{q}_t^\top \mathbf{k}_{t'}}{\sqrt{D_k}} - m_i\,(t - t'), \qquad t' \le t\,.$$

m_i: fixed slope of head i (e.g., $1/2, 1/4, \ldots$)

The penalty is not trained. Heads with a large slope look at nearby tokens, heads with a small one at the whole past. Models trained on short sequences still work on longer ones.

Press et al. (2022), *Train short, test long: Attention with linear biases enables input length extrapolation*, ICLR.

Visualizing ALiBi



bias / m_i $\mathbf{A}$, $m_i = 1$ $\mathbf{A}$, $m_i = 1/4$

Attention weights when all the scores $\mathbf{q}_t^\top \mathbf{k}_{t'}$ are equal. A large slope gives a soft local window, a small one an almost uniform average over the past.

No positional embeddings (NoPE)

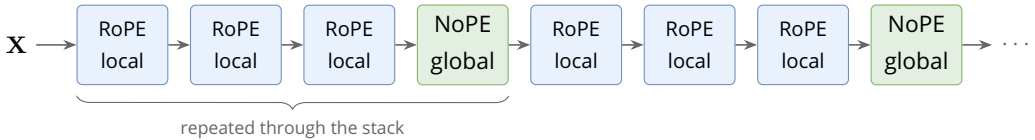
In a decoder, the causal mask already breaks the equivariance: token t attends to exactly $t+1$ tokens. For example, a head with equal scores outputs $\mathbf{h}_t = \frac{1}{t+1}\sum_{t' \le t} \mathbf{v}_{t'}$, where the contribution of the first token shrinks as $1/(t+1)$: later layers can read the position from it.

Causal language models with **no positional embeddings** (NoPE) train almost as well as with them, and degrade less on sequences longer than those seen in training.

This only works for decoders: without the mask, an encoder with no positional embeddings is still equivariant.

Haviv et al. (2022), *Transformer language models without positional encodings still learn positional information*, Findings of EMNLP; Kazemnejad et al. (2023), *The impact of positional encoding on length generalization in transformers*, NeurIPS.

Interleaving RoPE and NoPE



Recent models combine both, for example three RoPE layers, then one NoPE layer, repeated through the stack.

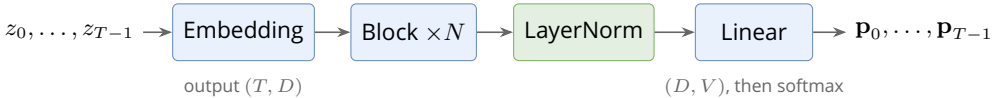
The RoPE layers handle word order among nearby tokens, often restricted to a window of recent tokens. The NoPE layers retrieve from the whole context, with no rotation angle that was never seen in training. Examples: Llama 4, Command A, SmolLM3.

Yang et al. (2025), *Rope to Nope and back again: A new hybrid attention strategy*.

Positional embeddings at a glance

Method	Position enters	Trained	Longer sequences	Used in
Learned	added to $\mathbf{X}$	yes	no	BERT, GPT-2, ViT
Sinusoidal	added to $\mathbf{X}$	no	only in theory	original transformer
ALiBi	bias on the scores	no	yes	BLOOM, MPT
RoPE	rotation of $\mathbf{q}$, $\mathbf{k}$	no	degrades	most current LLMs
NoPE	causal mask only	no	better than RoPE	interleaved with RoPE

Putting everything together



A **decoder-only** transformer, as in GPT:

1. Embed the tokens (plus positional embeddings, if any).
2. Apply a stack of N causal transformer blocks.
3. Apply a final LayerNorm: with pre-norm blocks, the residual path is never normalized otherwise.
4. Apply the same vocabulary classifier at every position.

The classifier weights are sometimes shared with the embedding matrix (**weight tying**).

Training and generation

We train as at the start of this lecture, on shifted targets with teacher forcing:

$$\mathcal{L} = -\frac{1}{T}\sum_{t=0}^{T-1} \log \mathbf{p}_{t\, z_{t+1}} .$$

Training is parallel

One forward pass on T tokens gives all T predictions. The causal mask guarantees that $\mathbf{p}_t$ never sees z_{t+1}.

Generation is sequential

Each new token needs a forward pass over all the previous ones. Caching their keys and values helps.

Cheap training and expensive generation drive much of the design of modern models.

The residual stream

In a pre-norm model, the vector of each token is only ever added to. Writing $\mathbf{x}_t$ for its embedding, the final vector is:

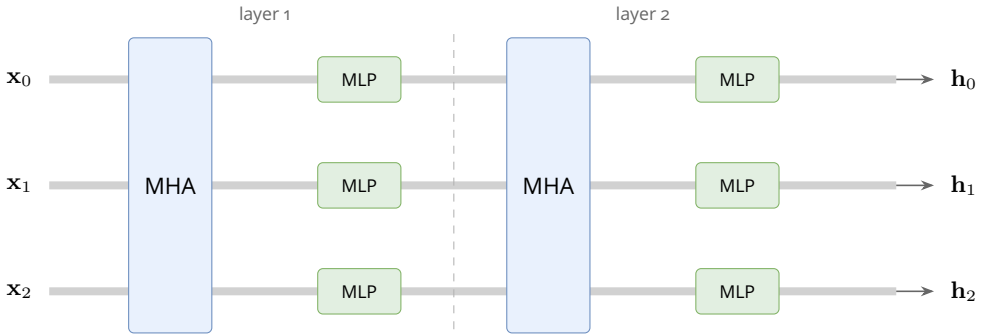
$$\mathbf{h}_t = \mathbf{x}_t + \sum_{\ell=1}^{N} \Big(\underbrace{\mathrm{MHA}_\ell}_{\text{reads all tokens}} + \underbrace{\mathrm{MLP}_\ell}_{\text{reads token } t \text{ only}} \Big)_t ,$$

We can read the model as one **residual stream** per token: every layer reads the stream through a LayerNorm and writes an update back. The MHA layers are the only ones that move information between streams.

As for ResNets, this view explains why removing a single layer often does little damage.

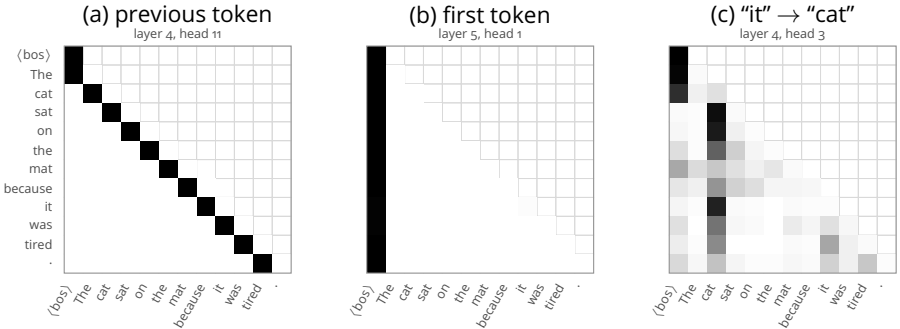
Elhage et al. (2021), *A mathematical framework for transformer circuits*, Transformer Circuits Thread.

Visualizing the residual stream



Each token has its own stream (gray), from its embedding to the output. An MHA layer reads from all the streams and writes to each of them; an MLP reads and writes a single stream.

Attention in a trained model



Three of the 144 heads of GPT-2 (small) on the same sentence (rows are queries, darker is larger). Some heads have simple, readable roles: (a) attends to the previous token, (b) puts almost all its weight on the first token, a way to “do nothing” (more on this shortly); in (c), “it” attends mostly to “cat”. Many heads have no clear pattern.

Classification with an encoder

To perform classification or regression with an encoder, we can apply a final global pooling on the T tokens and one or more fully-connected layers.

An alternative that is empirically found to work well is the **class token**, an additional *trainable* token $\mathbf{c} \in \mathbb{R}^D$ added to the input matrix:

$$\underset{(T+1,D)}{\mathbf{X}'} = \begin{bmatrix} \mathbf{c}^\top \\ \mathbf{X} \end{bmatrix}, \qquad \mathbf{H} = \mathrm{Transformer}(\mathbf{X}'), \qquad \mathbf{y} = \mathrm{softmax}\left(\mathbf{W}^\top \mathbf{H}_{0,:}\right).$$

The class token attends to all the others, and its output summarizes the sequence for the classifier. In a decoder, the last token is the only one that has seen the whole sequence, and plays the same role.

Attention sinks and registers

The attention weights sum to one, so a head with nothing to retrieve still has to look somewhere. Trained transformers put this weight on one uninformative token, typically the first one (as in GPT-2 above): an **attention sink**, often with very large activations.

In vision transformers, a few background patches become sinks and scratch space for global information, creating artifacts in the attention maps.

One fix is to add dedicated tokens, called **registers**: extra trainable tokens added like the class token and discarded at the output. Another fix adds an extra score to each head's softmax, fixed at zero (a $+1$ in the denominator) or trainable as in gpt-oss, so the weights on the tokens can sum to less than one. Sinks matter for long contexts.

Xiao et al. (2024), *Efficient streaming language models with attention sinks*, ICLR; Darcet et al. (2024), *Vision transformers need registers*, ICLR; E. Miller (2023), *Attention is off by one*; OpenAI (2025), *gpt-oss model card*.

Cross-attention

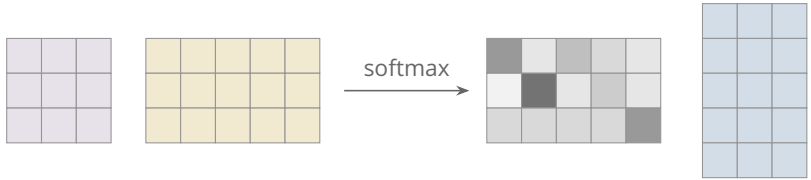
Given *two* sequences $\mathbf{X} \sim (T, D)$ and $\mathbf{Z} \sim (T', D')$, **cross-attention** takes the queries from the first and the keys and values from the second:

$$\underset{(T, D_v)}{\mathrm{CA}(\mathbf{X}, \mathbf{Z})} = \mathrm{softmax}\left(\frac{(\mathbf{X}\mathbf{W}_q)(\mathbf{Z}\mathbf{W}_k)^\top}{\sqrt{D_k}}\right)\mathbf{Z}\mathbf{W}_v\,.$$

The two sequences can have different lengths, and no mask is needed on $\mathbf{Z}$.

This is a useful operation that can combine information coming from different streams of information, e.g., a text and an image, or a text and its translation.

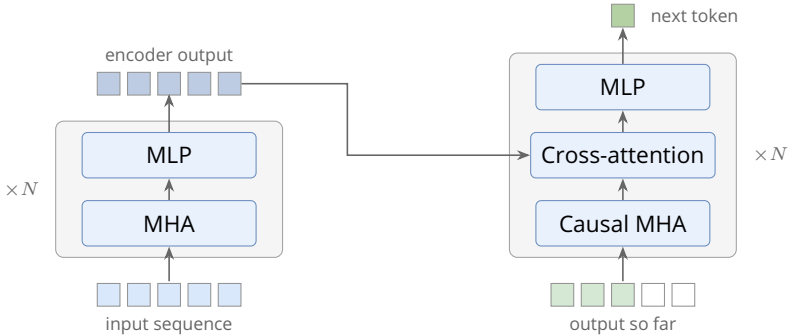
Visualizing cross-attention



Each of the T tokens of $\mathbf{X}$ gets a row of weights over the T' tokens of $\mathbf{Z}$, and collects their values.

The attention matrix is no longer square.

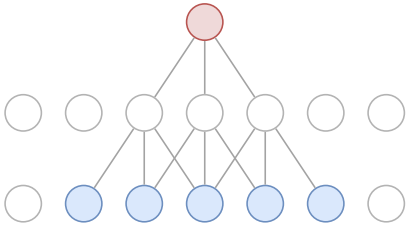
Encoder-decoder models



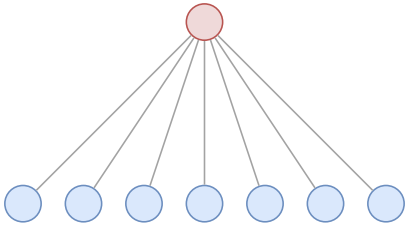
This is the original transformer, a **sequence-to-sequence** model as in the start of the lecture: every decoder block reads the encoder output with cross-attention, in place of the attention of Bahdanau et al. Today: T5, Whisper, image generators.

Vaswani et al. (2017), *Attention is all you need*, NeurIPS.

Convolutions and attention



(a) Two convolutions, $s = 3$: 5 tokens



(b) One attention layer: all T tokens

After L convolutions with kernel size s, an output sees $L(s-1)+1$ tokens, and two distant tokens interact only after $\mathcal{O}(T/s)$ layers. A single attention layer connects every pair of tokens directly, at the price of computing all T^2 weights. In a recurrent model, two tokens at distance k interact through k sequential state updates.

Computational cost

Consider a single layer on $\mathbf{X} \sim (T, D)$, counting multiply-adds. The recurrent layer is $\mathbf{h}_t = \phi(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t)$, the convolution has kernel size s:

	Recurrent	**Convolution**	**Multi-head attention**
Parameters	$2D^2$	sD^2	$4D^2$
Time	$\mathcal{O}(TD^2)$	$\mathcal{O}(TsD^2)$	$\mathcal{O}(TD^2) + \mathcal{O}(T^2D)$
Sequential steps	T	1	1
Receptive field	all past tokens	s tokens	T tokens
Extra memory	none	none	$\mathbf{A} \sim (T, T)$ per head

The $\mathcal{O}(TD^2)$ term comes from the projections, the $\mathcal{O}(T^2D)$ term from $\mathbf{QK}^\top$ and $\mathbf{AV}$. The quadratic term takes over once T is larger than (a small multiple of) D, which limited early transformers to a few hundred tokens.

Where do the parameters go?

Ignoring biases and normalization, a block with hidden MLP size $4D$ has:

$$\underbrace{4D^2}_{\text{MHA}} + \underbrace{8D^2}_{\text{MLP}} = 12D^2 \text{ parameters.}$$

GPT-2 (small) has $N = 12$ blocks, $D = 768$ and $V = 50{,}257$ tokens:

- blocks: $12 \cdot 12 \cdot 768^2 \approx 85$M parameters;
- token embeddings (tied with the classifier): $VD \approx 39$M;
- positional embeddings: $1024 \cdot 768 \approx 0.8$M.

The total is about 124M. A forward pass costs about 2 FLOPs per parameter and token (one multiply-add), training about three times more.

Radford et al. (2019), *Language models are unsupervised multitask learners*; Kaplan et al. (2020), *Scaling laws for neural language models*, arXiv.

Memory considerations

Materializing the attention matrices also requires $\mathcal{O}(T^2)$ memory, for every head and every layer. For example, with 32 heads, $T = 32{,}768$ tokens and 16-bit numbers:

$$32 \cdot 32{,}768^2 \cdot 2 \text{ bytes} \approx 69 \text{ GB},$$

for a single layer and a single sequence.

In practice, $\mathbf{A}$ is never stored as a whole. Modern implementations (e.g., **FlashAttention**) process the keys in chunks, keeping a running maximum and a running denominator of the softmax, and normalize only at the end. During generation, the keys and values of past tokens are stored in a **KV cache** instead of being recomputed.

4 Practical transformer models

Text, images and beyond

Transformers everywhere

Once data is turned into a sequence (or a set) of tokens, the same architecture applies almost unchanged:

Audio

wav2vec 2.0 tokenizes raw audio with a 1D CNN and is pre-trained self-supervised; Whisper is an encoder-decoder on spectrograms.

Graphs

Nodes become tokens, and the connectivity enters through positional embeddings or biases on the scores, as in AliBi.

Mixer models

Replacing MHA with an MLP across tokens still works well on images: the block structure matters as much as attention.

Baevski et al. (2020), *wav2vec 2.0*, NeurIPS; Radford et al. (2023), *Whisper*, ICML; Ying et al. (2021), *Do transformers really perform badly for graph representation?*, NeurIPS; Tolstikhin et al. (2021), *MLP-Mixer*, NeurIPS.

Transformers for text

Pre-trained text models differ mostly in the mask and in the objective:

GPT-like (decoders)
Causal mask, next-token prediction: the model of this lecture. Generates one token at a time.

BERT-like (encoders)
No mask; 15% of the tokens are hidden and predicted from both sides. For classification and embeddings.

Diffusion LMs
An encoder starts from a fully hidden sequence and fills it in over a few steps, many tokens at once.

All are **self-supervised** (the targets come from the text itself) and give **contextual** embeddings: "bank" differs in "river bank" and "bank account". Turning them into assistants is called *post-training*.

Devlin et al. (2019), *BERT*, NAACL; Qiu et al. (2020), *Pre-trained models for natural language processing: A survey*; Nie et al. (2025), *Large language diffusion models*, arXiv.

Applying transformers to images

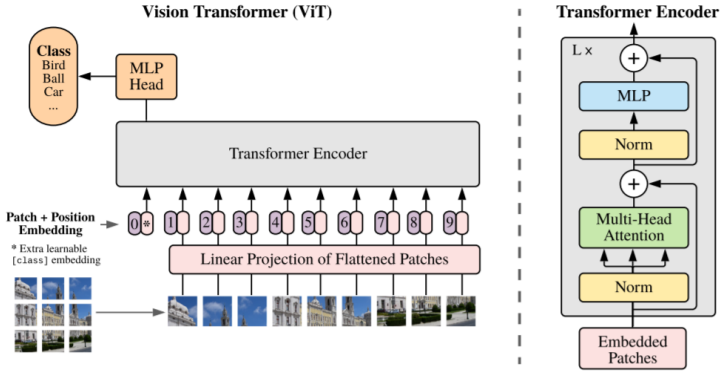
Transformers can also be applied in computer vision, especially when trained on huge datasets. However, this requires to convert the image (a 2D grid) into a sequence of tokens.

Using every pixel as a token is too expensive: a 224×224 image has about $50{,}000$ pixels, and $50{,}000^2$ attention weights per head. The common solution is to work on non-overlapping **patches**: with patches of size $P \times P$, an image $H \times W$ becomes

$$T = \frac{HW}{P^2} \text{ tokens,}$$

each obtained by flattening a patch and projecting it to D features. With $P = 16$, a 224×224 image becomes $14 \times 14 = 196$ tokens.

Vision transformer (ViT)



The Vision Transformer (ViT) is a standard transformer encoder applied on top of image patches, with a class token and trainable positional embeddings.

Dosovitskiy et al. (2021), *An image is worth 16x16 words: Transformers for image recognition at scale*, ICLR.

Patches as a strided convolution

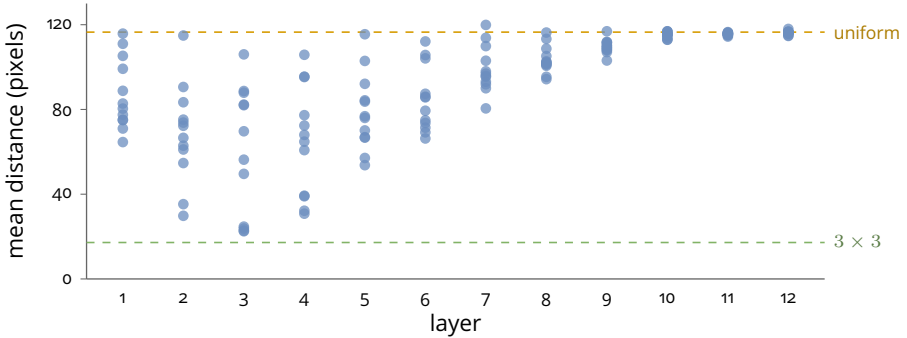
Flattening each patch and projecting it with a shared matrix is exactly a convolution with kernel size equal to the stride, as in the CNN lecture:

```
patchify = nn.Conv2d(3, D, kernel_size=P, stride=P)
x = patchify(img) # (B, D, H/P, W/P)
x = x.flatten(2).transpose(1, 2) # (B, T, D), T = HW/P^2
```

The patch size trades accuracy for cost. Halving P gives $4\times$ more tokens and $16\times$ more attention weights: ViT-B/16 and ViT-B/8 have the same parameters, but very different costs.

Dosovitskiy et al. (2021), *An image is worth 16x16 words: Transformers for image recognition at scale*, ICLR.

What does a ViT attend to?



Mean distance between a patch and the patches it attends to, one dot per head (ViT-B/16 pre-trained on 3.6B images, 260 ImageNet images). Middle layers mix local heads with global ones; later layers are close to a uniform average. Attention learns part of the locality that a convolution hard-codes.

As in Dosovitskiy et al. (2021); weights from Singh et al. (2022), CVPR.

Inductive biases versus data

Trained only on ImageNet-1k (1.3M images), the original ViT is worse than a ResNet of similar size. Pre-trained on ImageNet-21k (14M) it matches it, and on JFT-300M (300M) it is better.

Better training (DeiT)
Strong data augmentation, regularization and distillation from a CNN teacher make ViTs competitive on ImageNet-1k alone.

Better CNNs (ConvNeXt)
A ResNet updated with the transformer's design choices and training recipe matches vision transformers of similar cost.

Locality and translation equivariance are useful priors when data is scarce; with enough data, attention can learn what it needs.

Dosovitskiy et al. (2021), ICLR; Touvron et al. (2021), *Training data-efficient image transformers & distillation through attention*, ICML; Liu et al. (2022), *A ConvNet for the 2020s*, CVPR.

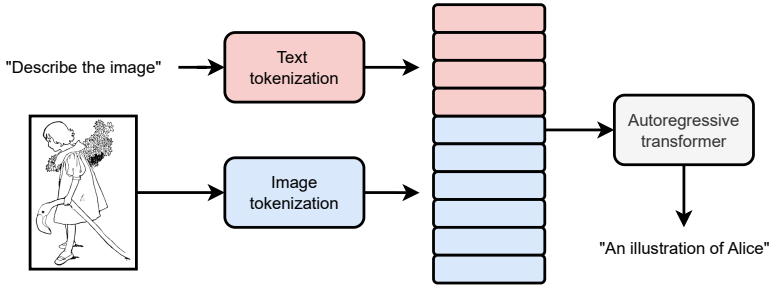
Pre-trained vision backbones

Pre-trained ViTs are today the default image encoders. A name like *ViT-B/16* may refer to the architecture or to one of many trained versions, shared through hubs (Hugging Face, `timm`, `torchvision`). They differ mostly in how they were trained:

- **Self-supervised on images**: masked autoencoders (MAE) hide 75% of the patches and reconstruct them; DINOv2 and DINOv3 train a model to match the features of a slowly updated copy of itself. Their patch features work well for dense tasks.
- **Image and text**: CLIP and SigLIP train an image encoder and a text encoder so that matching image-caption pairs have similar embeddings, enabling zero-shot classification and retrieval.

He et al. (2022), *Masked autoencoders are scalable vision learners*, CVPR; Oquab et al. (2024), *DINOv2*, TMLR; Siméoni et al. (2025), *DINOv3*, arXiv; Radford et al. (2021), *CLIP*, ICML; Zhai et al. (2023), *SigLIP*, ICCV.

Multimodal models



Transformers also simplify **multimodal** models by projecting tokens of different modalities in a single embedding space. In most vision-language models, the image tokens come from a pre-trained ViT (e.g., SigLIP), followed by a small projection.

Further readings

- **Chapters 10 and 11** of *Alice's Adventures in a Differentiable Wonderland*.
- The Illustrated Transformer.
- Thinking Like Transformers, for intuitions about how transformers can implement algorithms.
- A Mathematical Framework for Transformer Circuits, for the residual stream view.