TEXT ANALYTICS

Attention and Transformers

Francesco Spinnato

* francesco.spinnato@unipi.it
University of Pisa



Where We Are

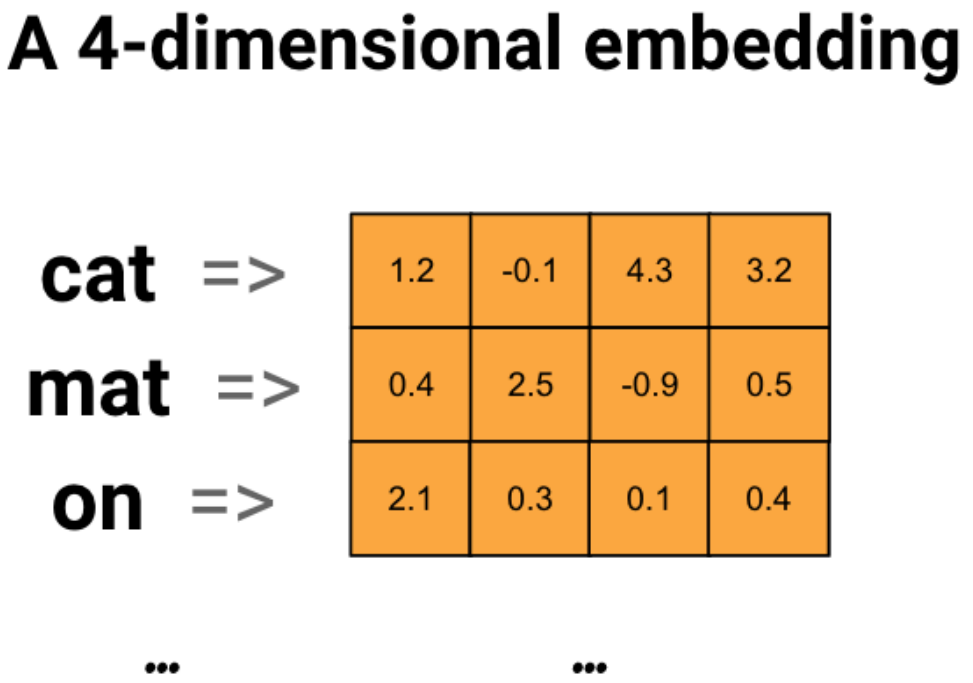
Previously, we introduced **embeddings**. A discrete object is mapped to a d-dimensional dense vector. For a sequence of tokens:

$$[w_1, w_2, \ldots, w_T]$$

we obtain a sequence of embeddings:

$$[\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T] \in \mathbb{R}^{T \times d}$$

How do we make each vector depend on the surrounding context?



From Static to Contextual Embeddings

Classical approaches like Word2Vec assign one vector, $\mathbf{e}$ to each token. These approaches learn one **static** embedding per word.

- in "cold weather", **cold** refers to temperature
- in "cold symptoms", **cold** refers to illness

The embedding, $\mathbf{e}_{\text{cold}}$, is the same, but the meaning is different.

Contextual models can produce different vectors for the same token depending on the surrounding words. A **contextual embedding** depends on the full sequence:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{x}_1, \ldots, \mathbf{x}_T)$$

So the representation of token t changes depending on its context.

Why Context Matters

In language, the meaning of one element almost always depends on the others.

Examples:

- *bank* means something different in "river **bank**" and "investment **bank**".
- *it* refers to a different noun depending on the sentence.
- *not* can flip the polarity of an entire review.
- A pronoun may refer to something mentioned many sentences earlier.

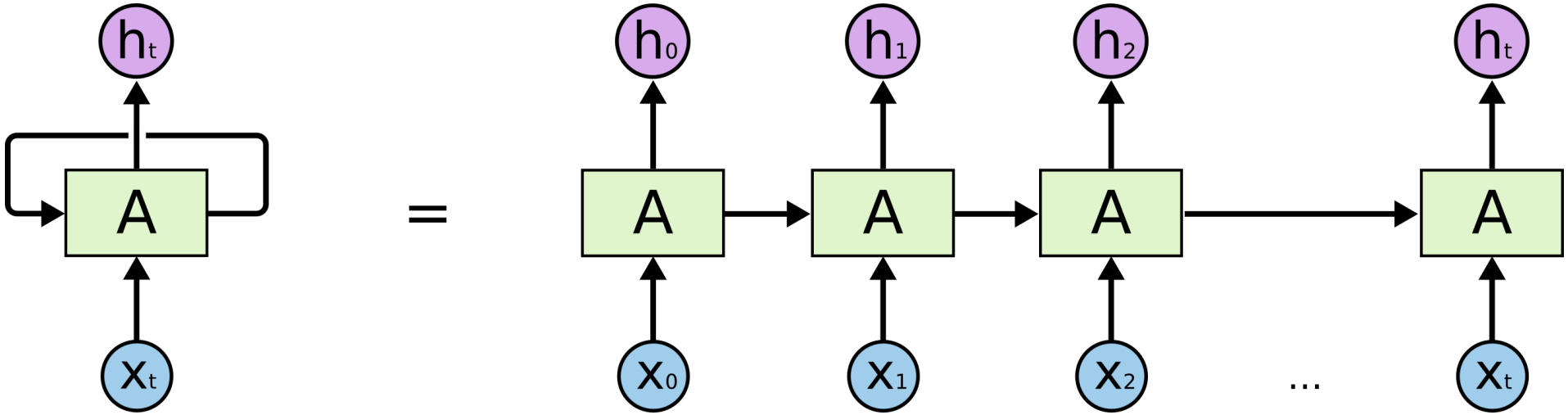
We need models that can capture **dependencies across a sequence**.

Recurrent Neural Networks

As we saw previously, an RNN reads a sequence one element at a time. At each step, to produce a new hidden state, it combines:

- the current input, $\mathbf{x}_t$
- the previous hidden state, $\mathbf{h}_{t-1}$

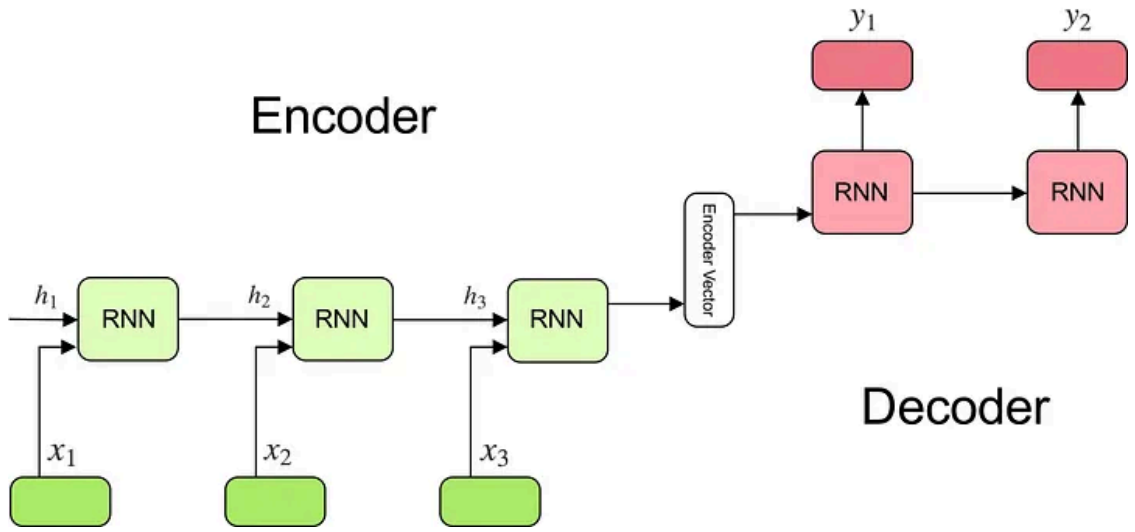
The hidden state is then passed to the next time step.



Sequence-to-Sequence Learning

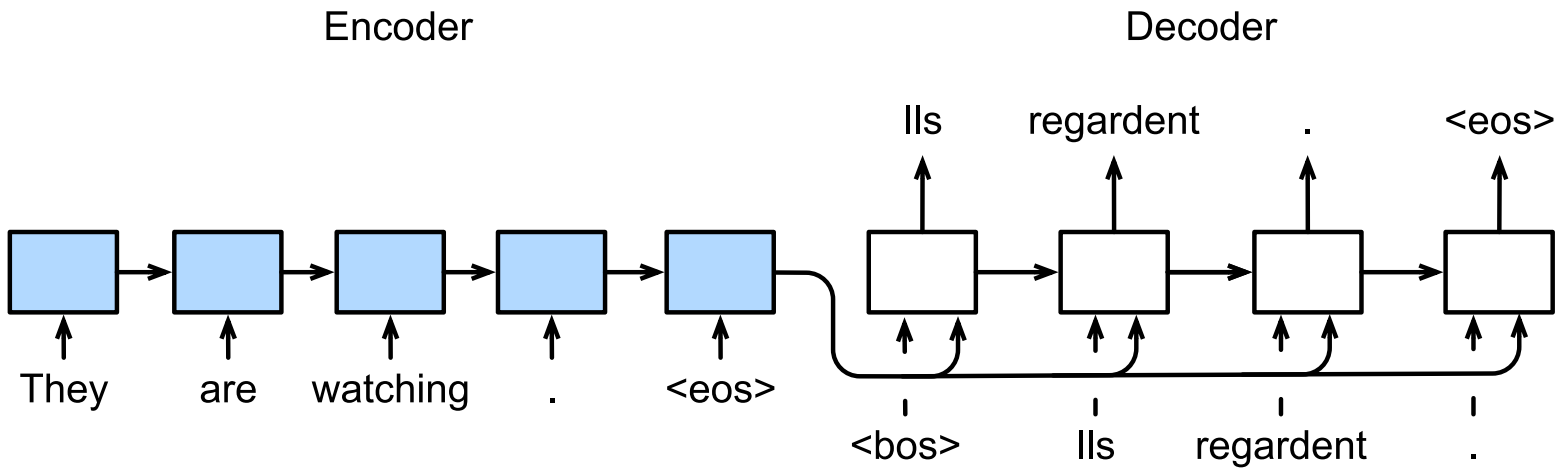
Consider machine translation as an example. When generating a translation of a source text, we first pass the source text through an **encoder** to obtain a sequence of encoder hidden states, and the last one is used by the **decoder**.

- all relevant information about a source sequence is translated into some internal **fixed-dimensional** state representation by the encoder.
- this state is used by the decoder as the **complete and exclusive source of information** for generating the translated sequence.



While this is reasonable for short sequences, it is infeasible for long ones

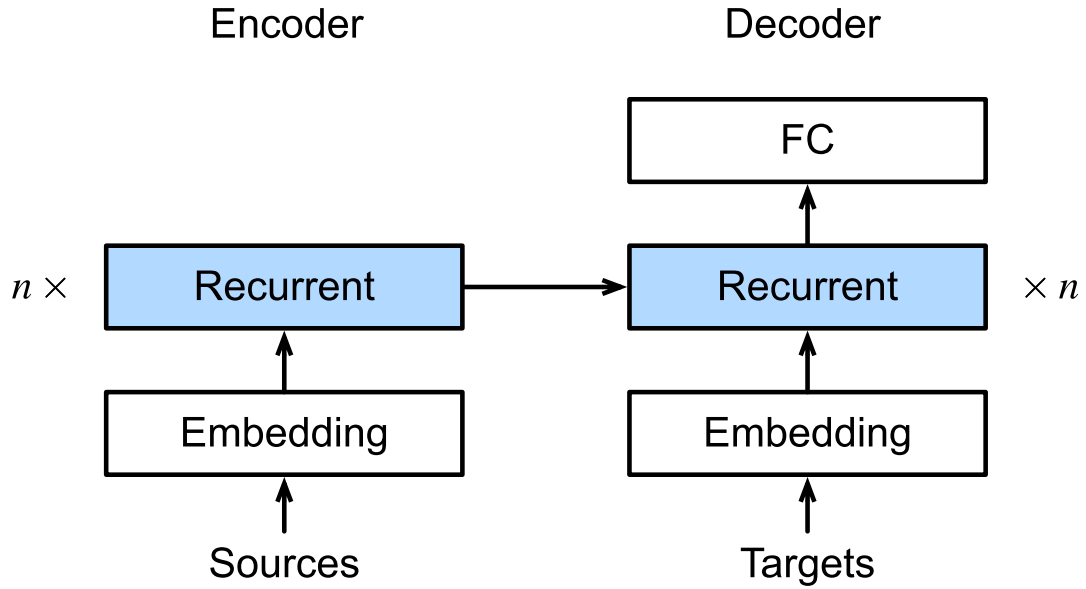
Seq2Seq: The Context Vector



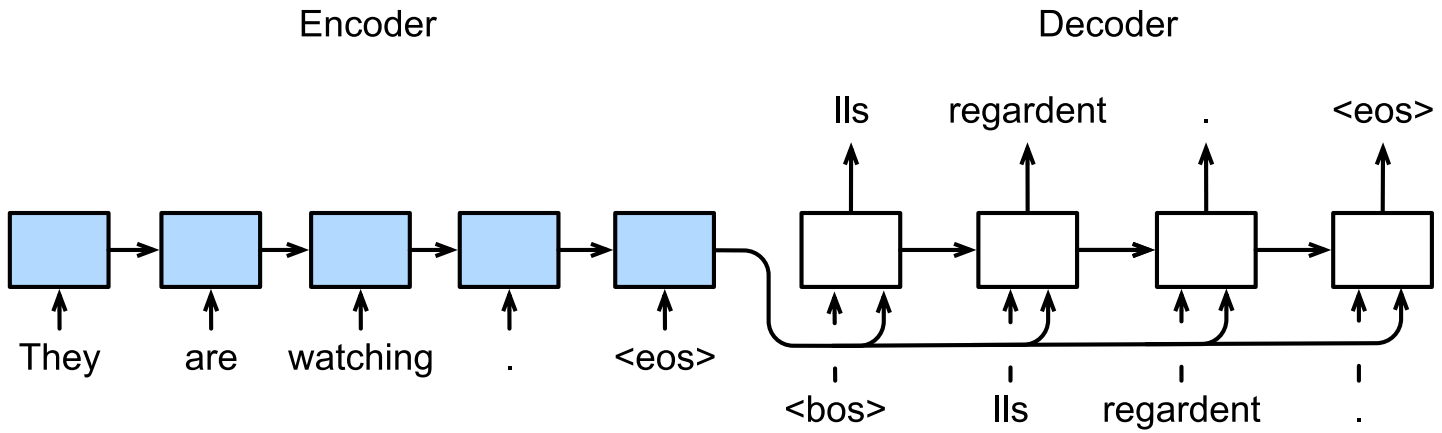
The encoder transforms the hidden states at all time steps into a context vector through a customized function:

$$\mathbf{c} = f(\mathbf{h}_1, \ldots, \mathbf{h}_T)$$

which is often just the last hidden state, i.e., $\mathbf{c} = \mathbf{h}_T$.



Seq2Seq: The Decoder



Given a target output sequence, $y_1, \ldots, y_{T'}$, for each decoder time step t' the decoder assigns a predicted probability to each possible token $y_{t'}$, based on the current state $\mathbf{s}_{t'}$, computed from:

- the previous step's target token ($y_{t'-1}$)
- the hidden RNN state from the previous time step ($\mathbf{s}_{t'-1}$)
- and the context vector ($\mathbf{c}$)

Seq2Seq: Limitations

Sequence-to-Sequence Learning gives the model memory, but also creates limitations:

- Information must pass through **many intermediate states**.
- Long-range dependencies can be difficult to preserve.
- Computation is **sequential**.
- Parallelization is limited.

Attention was introduced to reduce this bottleneck.

Attention

Queries, Keys, and Values

Attention in neural networks can be seen as a soft lookup operation in a database (or a dictionary), $\mathcal{D}$, using a query, q.

Key (last name)	Value (first name)
Zhang	Aston
Lipton	Zachary
Li	Mu
Smola	Alex
Hu	Rachel
Werness	Brent

We can search on $\mathcal{D}$ using queries, q:

- an exact query for “Li” which would return the value “Mu”.
- if (“Li”, “Mu”) was not a record in $\mathcal{D}$, there would be no valid answer.
- if we also allowed for approximate matches, we would also retrieve (“Lipton”, “Zachary”)

Attention Pooling

Given $\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \ldots (\mathbf{k}_m, \mathbf{v}_m)\}$, a database of m key-value pairs, and a query, $\mathbf{q}$, we define the **attention** over $\mathcal{D}$ as:

$$\text{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i,$$

where $\alpha(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}$ are scalar attention weights and (usually):

- the weights are positive: $\alpha(\mathbf{q}, \mathbf{k}_i) \geq 0$
- the weights sum to one: $\sum_i \alpha(\mathbf{q}, \mathbf{k}_i) = 1$

This is just a weighted average of all the values in the database, depending on how well the query matches each key.

The name attention derives from the fact that the operation pays particular attention to the values for which the weight is significant (i.e., large)

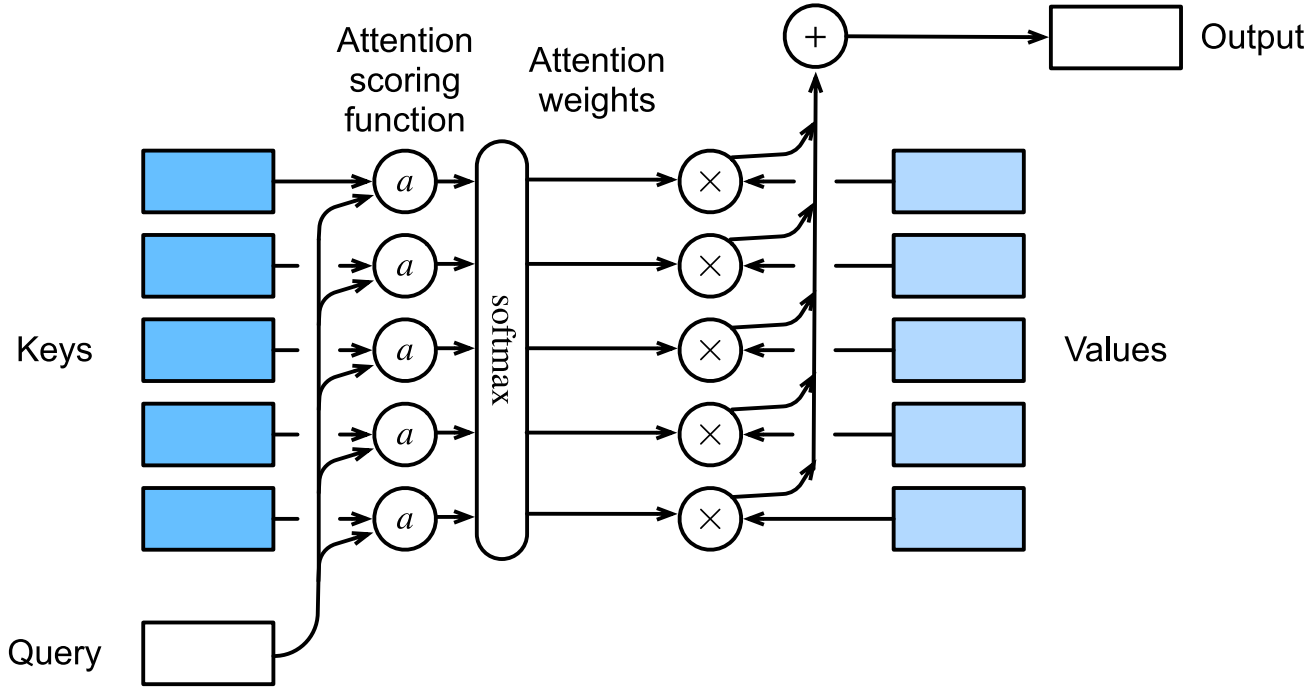
Dot Product Attention

The most common scoring function α is just the dot product between the query $\mathbf{q} \in \mathbb{R}^d$ and the key, $\mathbf{k} \in \mathbb{R}^d$, rescaled by $1/\sqrt{d}$, i.e.,

$$\mathbf{q}^\top \mathbf{k}_i / \sqrt{d}$$

To ensure the weights sum to one, we apply a softmax:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d})}{\sum_{j=1}^{m} \exp(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d})}.$$



Normalization avoids large dot products pushing softmax into saturation/vanishing gradient territory

Dot Product Attention Example

$$\text{Query:} \quad \mathbf{q} = [0.95, 0.35]$$

Key $\mathbf{k}_i$	Value $\mathbf{v}_i$	$\mathbf{q}^\top \mathbf{k}_i$	$\mathbf{q}^\top \mathbf{k}_i / \sqrt{d}$	Weight α_i
$[0.9, 0.1]$	$[10, 2]$	0.89	0.63	0.463
$[0.4, 0.6]$	$[4, 8]$	0.59	0.42	0.375
$[-0.7, 0.2]$	$[1, 9]$	-0.60	-0.42	0.162

Here $d = 2$, so we divide by $\sqrt{2} \approx 1.41$ and then apply the softmax.
Attention output:

$$0.463 \begin{bmatrix} 10 \\ 2 \end{bmatrix} + 0.375 \begin{bmatrix} 4 \\ 8 \end{bmatrix} + 0.162 \begin{bmatrix} 1 \\ 9 \end{bmatrix} \approx \begin{bmatrix} 6.29 \\ 5.38 \end{bmatrix}$$

Attention for Translation

Neural Machine Translation by Jointly Learning to Align and Translate

Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio

Jacobs University Bremen, Université de Montréal · ICLR 2015

Bahdanau, Cho and Bengio (2015): the decoder looks back at all the encoder states, and learns where to attend.

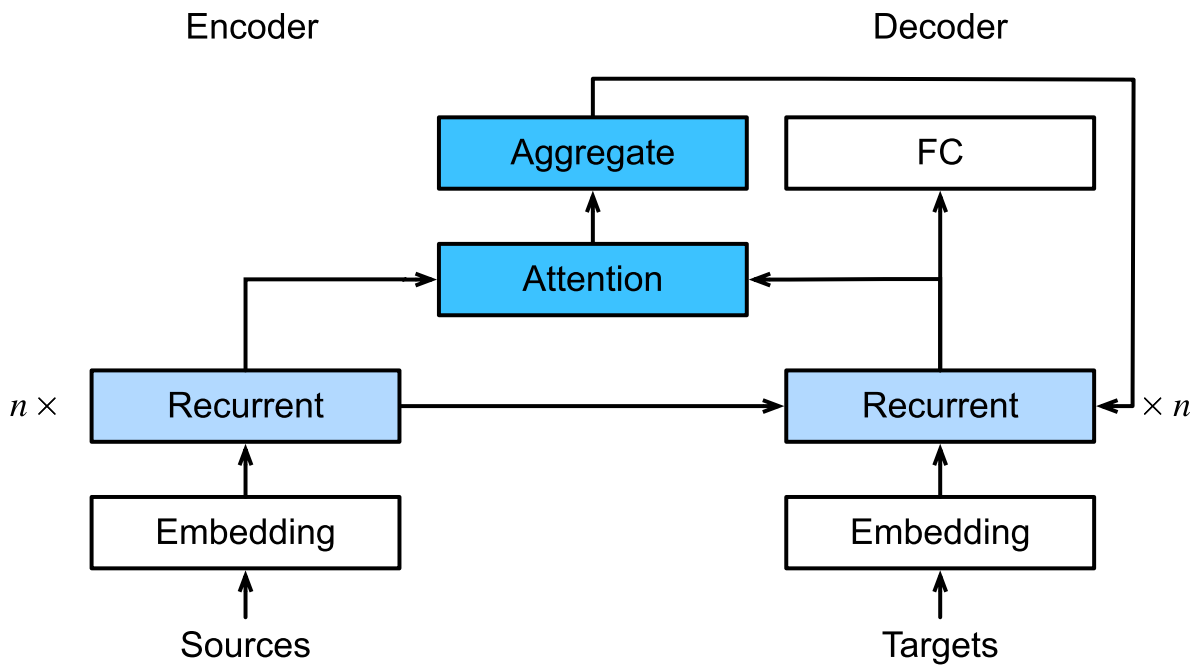
Bahdanau Attention Mechanism

The attention mechanism can be used in a sequence-to-sequence RNN architecture. Instead of keeping the context variable $\mathbf{c}$ fixed, we dynamically update it, as a function of:

- the original text (the encoder hidden state $\mathbf{h}_t$)
- the text that was already generated (the decoder hidden state $\mathbf{s}_{t'-1}$)

This yields $\mathbf{c}_{t'}$, which is updated after any decoding time step:

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



α are computed as in the previous slides, from a score between $\mathbf{s}_{t'-1}$ and each $\mathbf{h}_t$ followed by a softmax.

Bahdanau Attention: Query, Keys, Values

In Bahdanau attention:

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

- The query $\mathbf{q}$ is the previous decoder state, $\mathbf{s}_{t'-1}$.
- The keys and values are the encoder hidden states, $\mathbf{h}_t$.
- The current context vector, $\mathbf{c}_{t'}$, is a weighted average of the encoder hidden states, based on how *relevant* each encoder state is to the previous decoder state.

But do we really need the recurrent structure? Turns out that:

Attention is all you need!

The Transformer

Attention is all you need

Without Recurrence

Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones,
Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin

Google Brain, Google Research, University of Toronto · NeurIPS 2017

Vaswani et al. (2017) proposes the **TRANSFORMER**: a model built only on attention, with no recurrence.

Self-Attention: Queries, Keys, Values

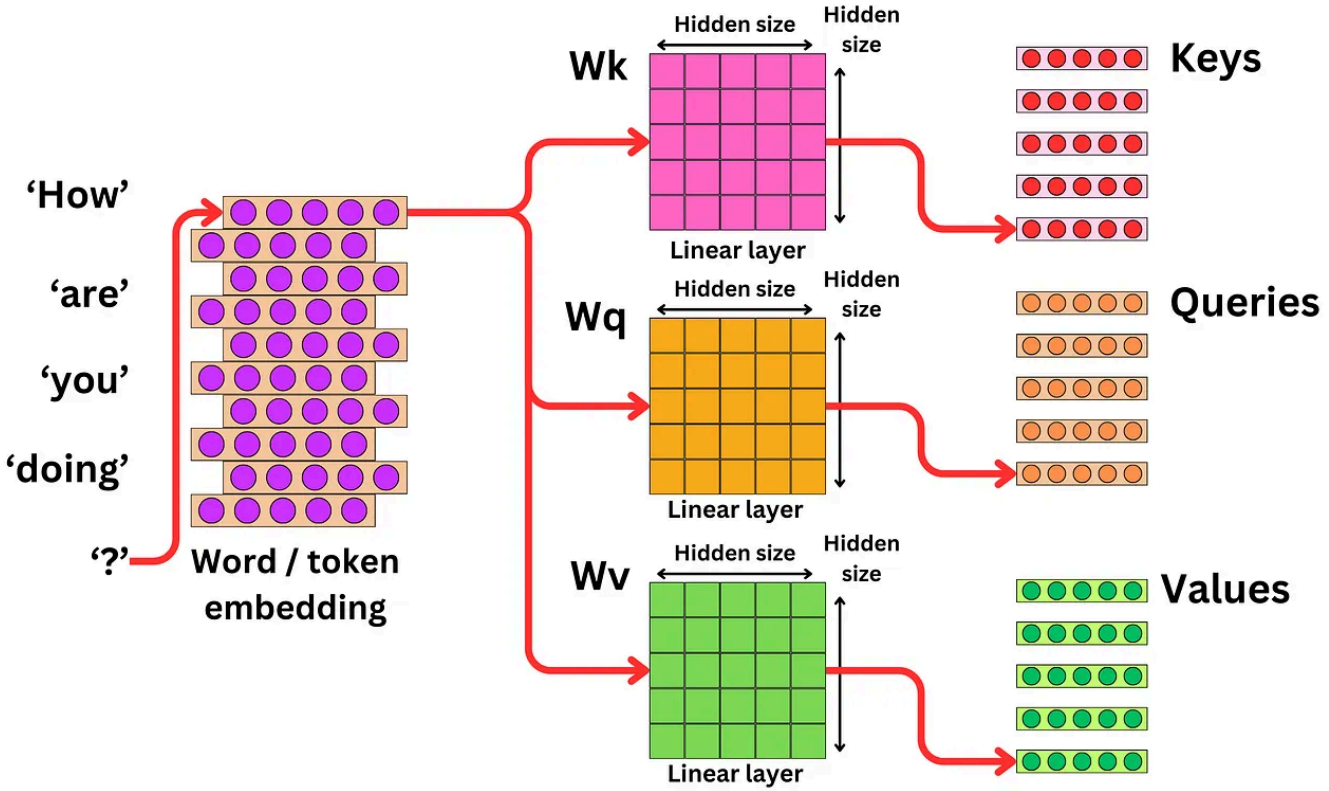
Instead of using the attention mechanism in a RNN, imagine feeding a sequence of tokens, $\mathbf{x}_1, \ldots, \mathbf{x}_T$ into an attention mechanism such that at every step, each token, $\mathbf{x}_t$ has its own query, keys, and values.

Query, keys, and values for $\mathbf{x}_t$, are learned via three different linear projections:

$$\mathbf{q}_t = \mathbf{W}^Q \mathbf{x}_t,$$

$$\mathbf{k}_t = \mathbf{W}^K \mathbf{x}_t,$$

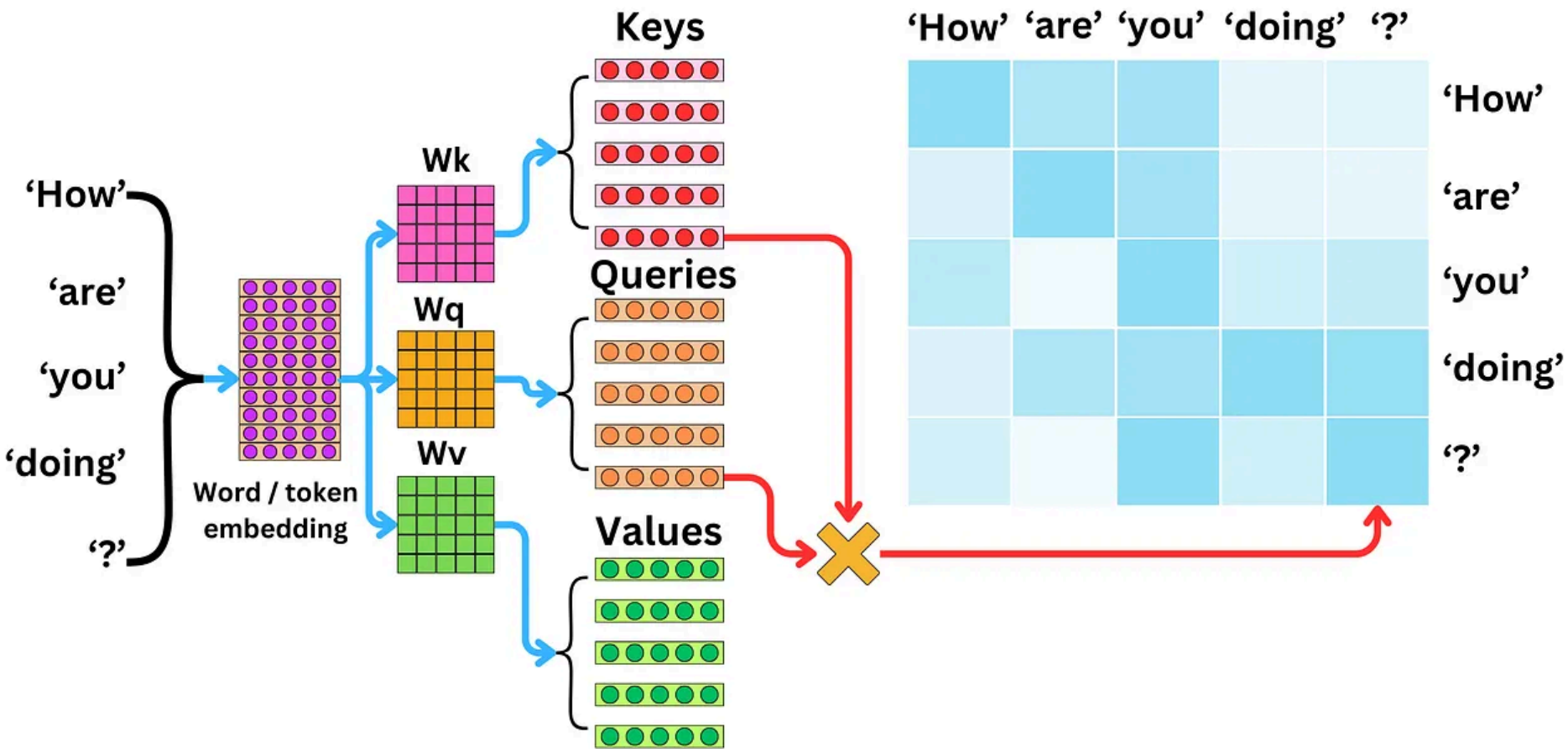
$$\mathbf{v}_t = \mathbf{W}^V \mathbf{x}_t$$



You can find an interactive visualization here: https://poloclub.github.io/transformer-explainer/

Self-Attention: Scores

We then compute the dot product between all the keys and the queries.

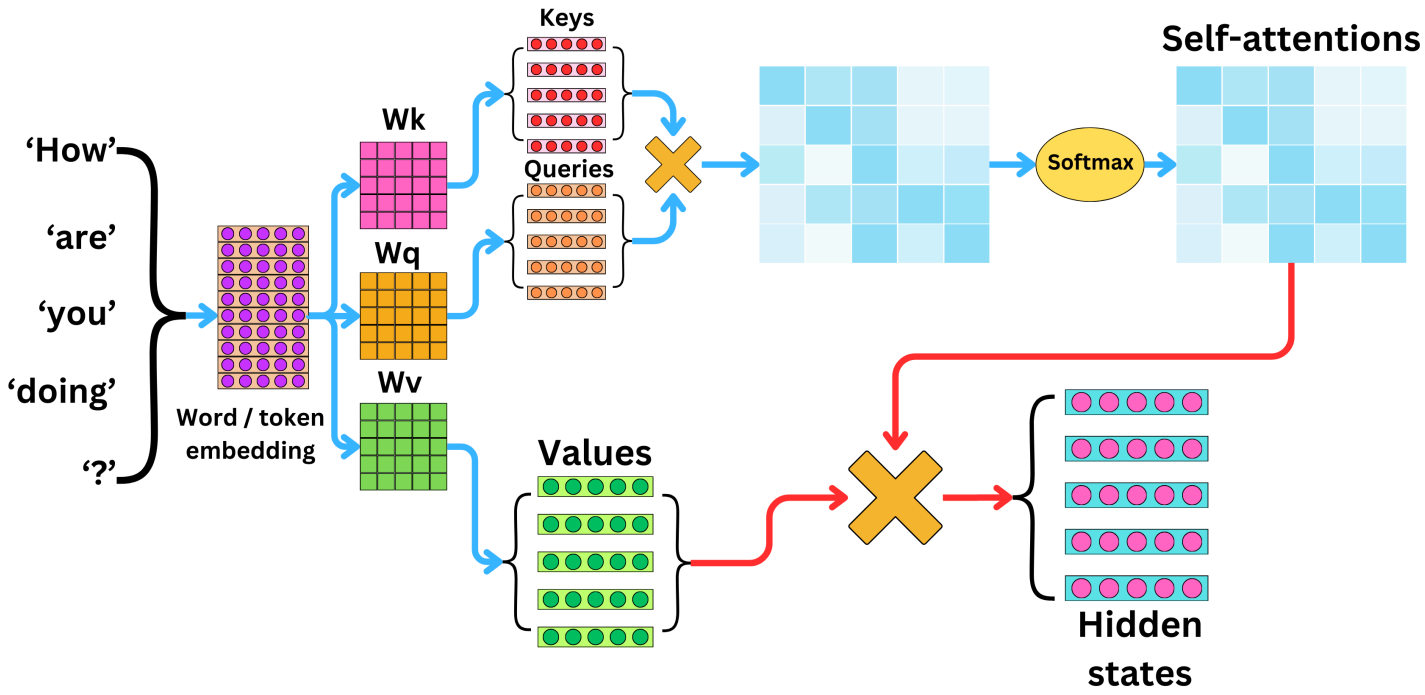


Self-Attention: The Attention Matrix

After a softmax transformation, this matrix of interactions is called the **attention matrix**. The resulting output hidden states of the attention layer are the matrix multiplication of the attention matrix and the values vectors.

In matrix form, with the queries, keys and values of the T tokens as the rows of $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$:

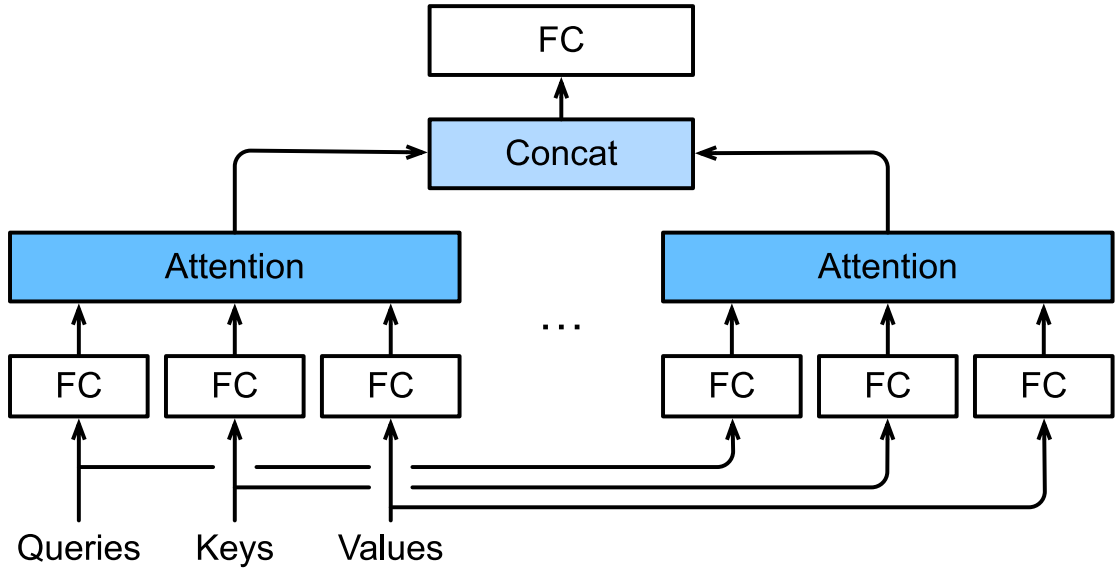
$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V}$$



Multi-Head Attention

In practice, instead of performing a single attention pooling:

- in self-attention, $\mathbf{x}_t$ goes through a **different set** of linear projections for each head ($\mathbf{W}_i^Q$, $\mathbf{W}_i^K$, $\mathbf{W}_i^V$), learned **independently**;
- every head runs attention pooling on its own $\mathbf{q}_i$, $\mathbf{k}_i$, and $\mathbf{v}_i$, **in parallel**;
- **the outputs of the heads are concatenated** and transformed with one more learned projection to produce the final output.



Each of the attention pooling outputs is a *head*.

Why Multiple Heads?

Different relationships may be useful at the same time.
For a sentence:

- one head may track syntactic dependencies (subject to verb)
- one head may link a pronoun to its antecedent
- one head may attend to the immediately preceding words
- one head may capture long-range dependencies

Multi-head attention gives the model multiple representation subspaces.

Attention Has No Order by Itself

Self-attention **compares every token with every other token.** However, if we only provide token embeddings, the model does not know their positions in the sequence.

For example, *"dog bites man"* and *"man bites dog"*: contain the same vectors, but in a different order.

$$[\mathbf{x}_{\text{dog}}, \mathbf{x}_{\text{bites}}, \mathbf{x}_{\text{man}}] \quad \text{and} \quad [\mathbf{x}_{\text{man}}, \mathbf{x}_{\text{bites}}, \mathbf{x}_{\text{dog}}]$$

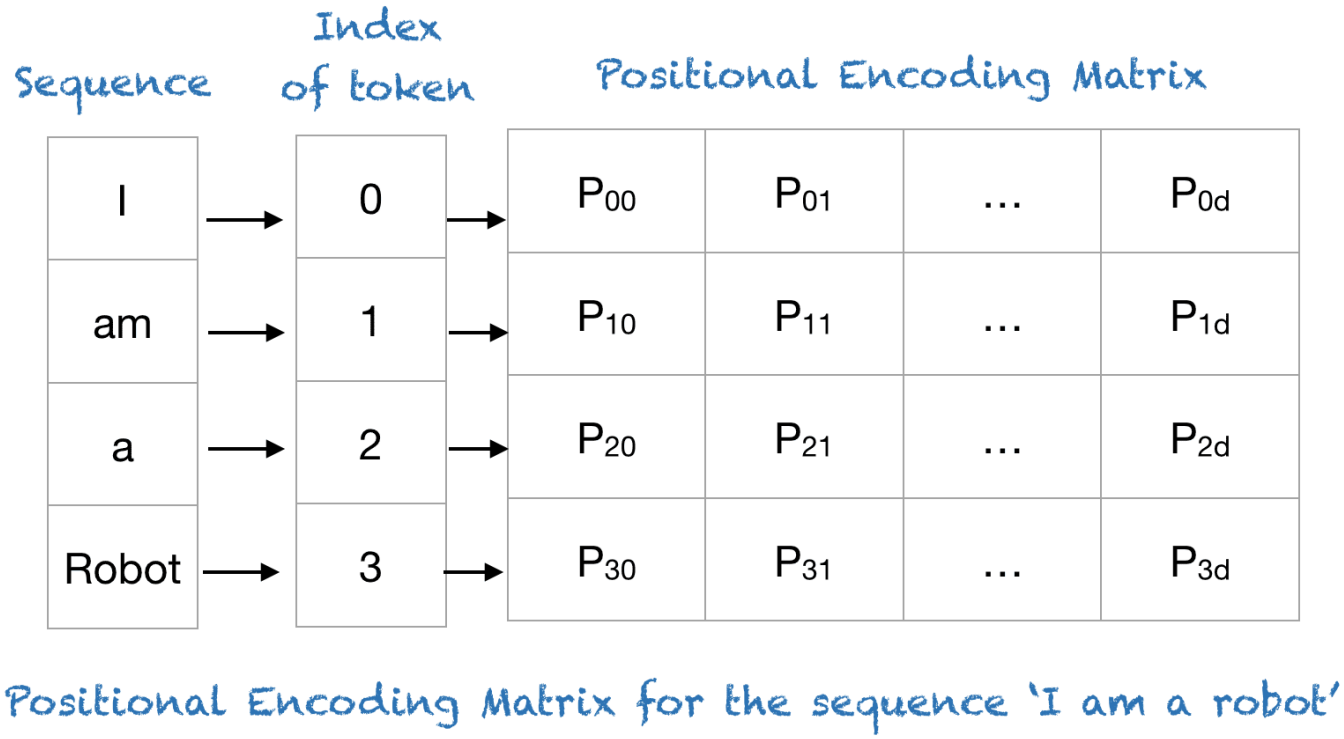
- Without positional information, self-attention sees only the vectors, not whether a token came first, second, or third.
- The word *dog* gets the same output in both sentences: the model cannot tell who bites whom.
- Therefore, **transformers add positional encodings** to the token embeddings.

Positional Encodings

We add a positional vector to each token embedding:

$$\tilde{\mathbf{x}}_t = \mathbf{x}_t + \mathbf{p}_t$$

- $\mathbf{x}_t$ is the token embedding
- $\mathbf{p}_t$ is the positional encoding
- $\tilde{\mathbf{x}}_t$ is the input to the transformer



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

The model can distinguish the same token appearing in different positions:

- **what it is:** token embedding
- **where it is:** positional encoding

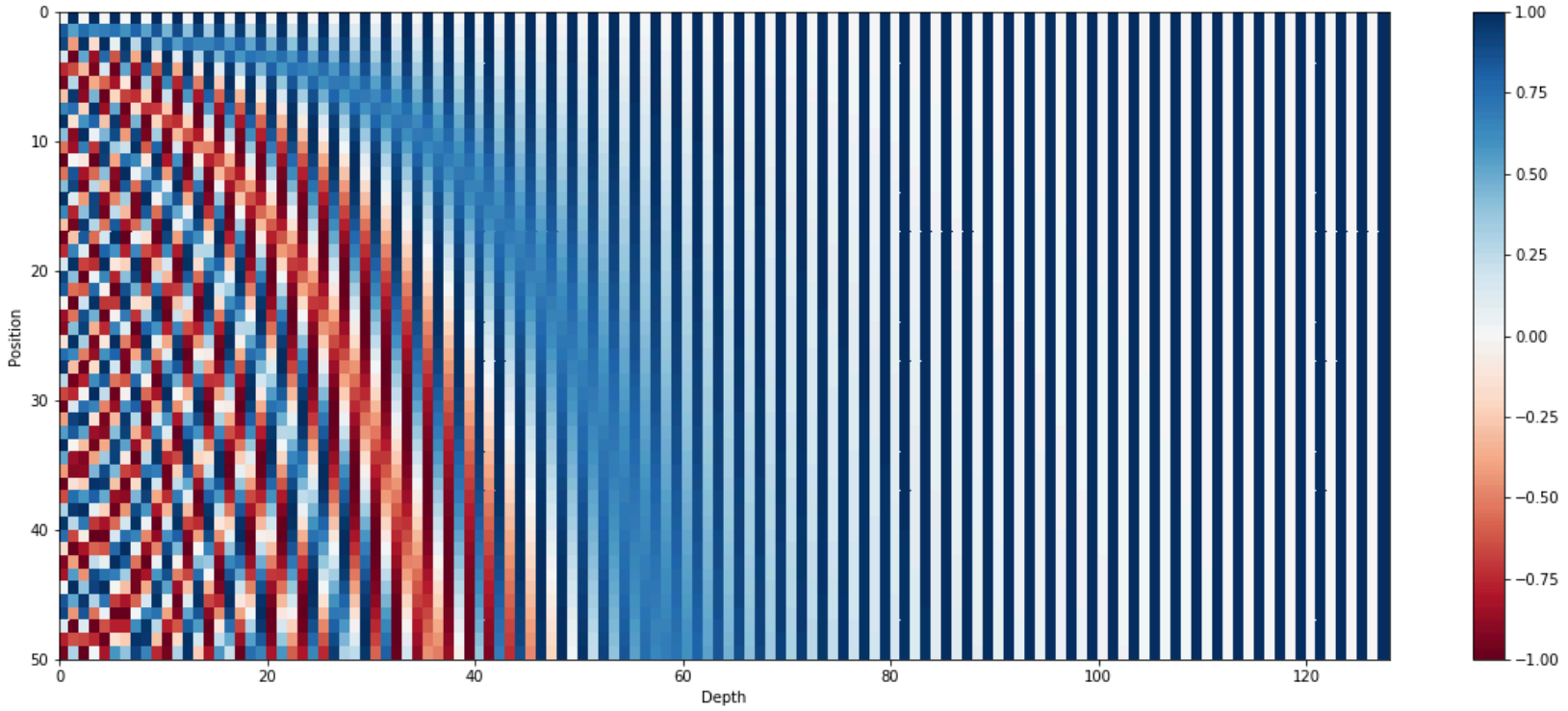
Positional Encodings

The original transformer uses **fixed** positional encodings (not learned). For position t, in a vector of size d, entry $2i$ is a sine and entry $2i+1$ is a cosine of the same angle, with $i = 0, \ldots, d/2 - 1$:

$$p_{t,2i} = \sin\left(\frac{t}{10000^{2i/d}}\right) \qquad p_{t,2i+1} = \cos\left(\frac{t}{10000^{2i/d}}\right)$$

Positional Encodings

Every value of i gives a sinusoid with its own wavelength, so each position (a row) gets a different vector, and the formula works for **any length**.



Positional Encoding Example: The Vectors

A toy case with $d = 4$, for *"dog bites man"*, with the token embeddings:

$$\mathbf{x}_1 = [1.0,\ 0.5,\ -0.5,\ 0.2], \quad \mathbf{x}_2 = [-0.5,\ 1.0,\ 0.3,\ -0.8], \quad \mathbf{x}_3 = [0.2,\ -1.0,\ 0.6,\ 0.4]$$

With $d = 4$ there are two values of i. For $i = 0$ the denominator is $10000^0 = 1$, for $i = 1$ it is $10000^{2/4} = 100$, so

$$\mathbf{p}_t = [\sin t,\ \cos t,\ \sin(t/100),\ \cos(t/100)]$$

$$\begin{aligned}
\mathbf{p}_1 &= [\sin 1,\ \cos 1,\ \sin 0.01,\ \cos 0.01] = [0.84,\ 0.54,\ 0.01,\ 1.00] \\
\mathbf{p}_2 &= [\sin 2,\ \cos 2,\ \sin 0.02,\ \cos 0.02] = [0.91,\ -0.42,\ 0.02,\ 1.00] \\
\mathbf{p}_3 &= [\sin 3,\ \cos 3,\ \sin 0.03,\ \cos 0.03] = [0.14,\ -0.99,\ 0.03,\ 1.00]
\end{aligned}$$

The angles are in radians. The first two entries change a lot from one position to the next, the last two change slowly. Real models have hundreds of dimensions.

Positional Encoding Example: The Sum

The input of the transformer is $\tilde{\mathbf{x}}_t = \mathbf{x}_t + \mathbf{p}_t$:

$$\begin{aligned}
\text{dog:} \quad & \tilde{\mathbf{x}}_1 = [1.0,\ 0.5,\ -0.5,\ 0.2] + [0.84,\ 0.54,\ 0.01,\ 1.00] = [1.84,\ 1.04,\ -0.49,\ 1.20] \\
\text{bites:} \quad & \tilde{\mathbf{x}}_2 = [-0.5,\ 1.0,\ 0.3,\ -0.8] + [0.91,\ -0.42,\ 0.02,\ 1.00] = [0.41,\ 0.58,\ 0.32,\ 0.20] \\
\text{man:} \quad & \tilde{\mathbf{x}}_3 = [0.2,\ -1.0,\ 0.6,\ 0.4] + [0.14,\ -0.99,\ 0.03,\ 1.00] = [0.34,\ -1.99,\ 0.63,\ 1.40]
\end{aligned}$$

The word *dog* always has the same embedding, $\mathbf{x}_{\text{dog}} = [1.0,\ 0.5,\ -0.5,\ 0.2]$.
Its input depends on the position:

$$\begin{aligned}
\text{in "dog bites man"}, t = 1: \quad & \mathbf{x}_{\text{dog}} + \mathbf{p}_1 = [1.84,\ 1.04,\ -0.49,\ 1.20] \\
\text{in "man bites dog"}, t = 3: \quad & \mathbf{x}_{\text{dog}} + \mathbf{p}_3 = [1.14,\ -0.49,\ -0.47,\ 1.20]
\end{aligned}$$

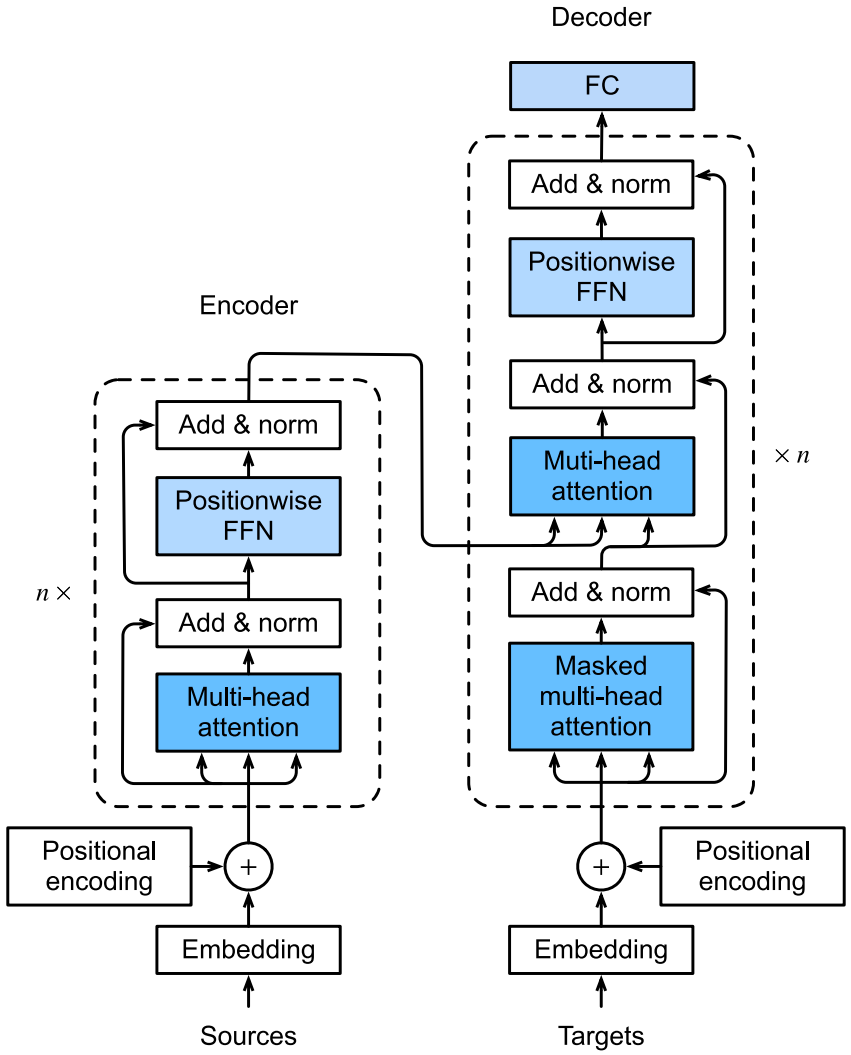
The same word gets a different input depending on **where it is**.

The Transformer

The Transformer **removes recurrence** entirely. It processes the whole sequence in parallel using attention mechanisms, without RNN or convolutional layers.

It has two main parts:

- **encoder**: reads the source sequence
- **decoder**: generates the target sequence

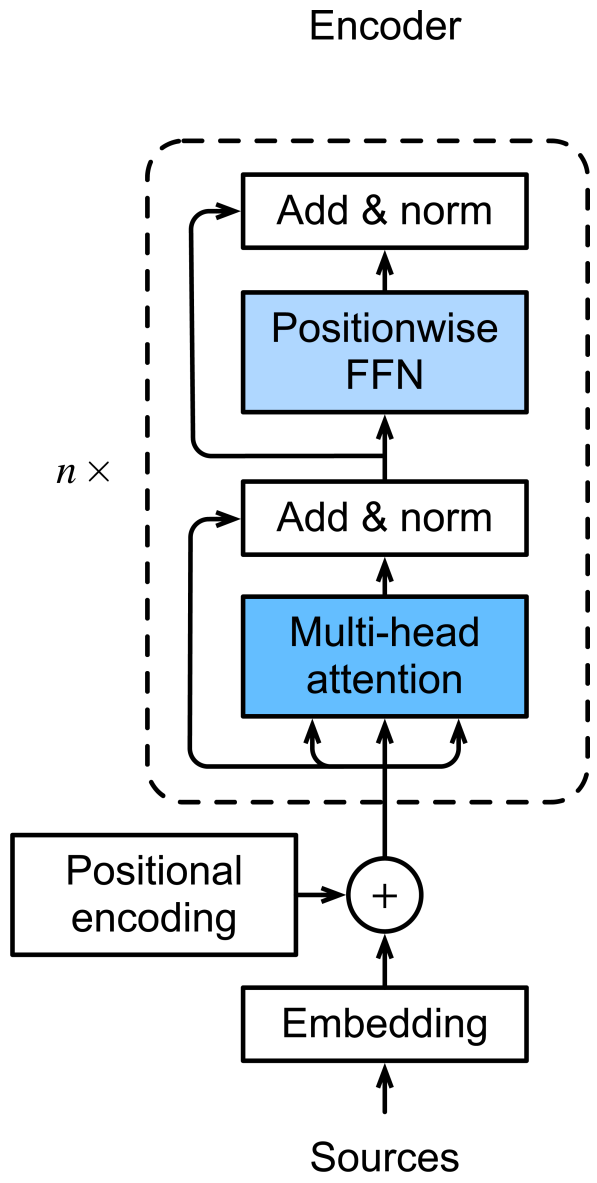


Transformer Encoder Block

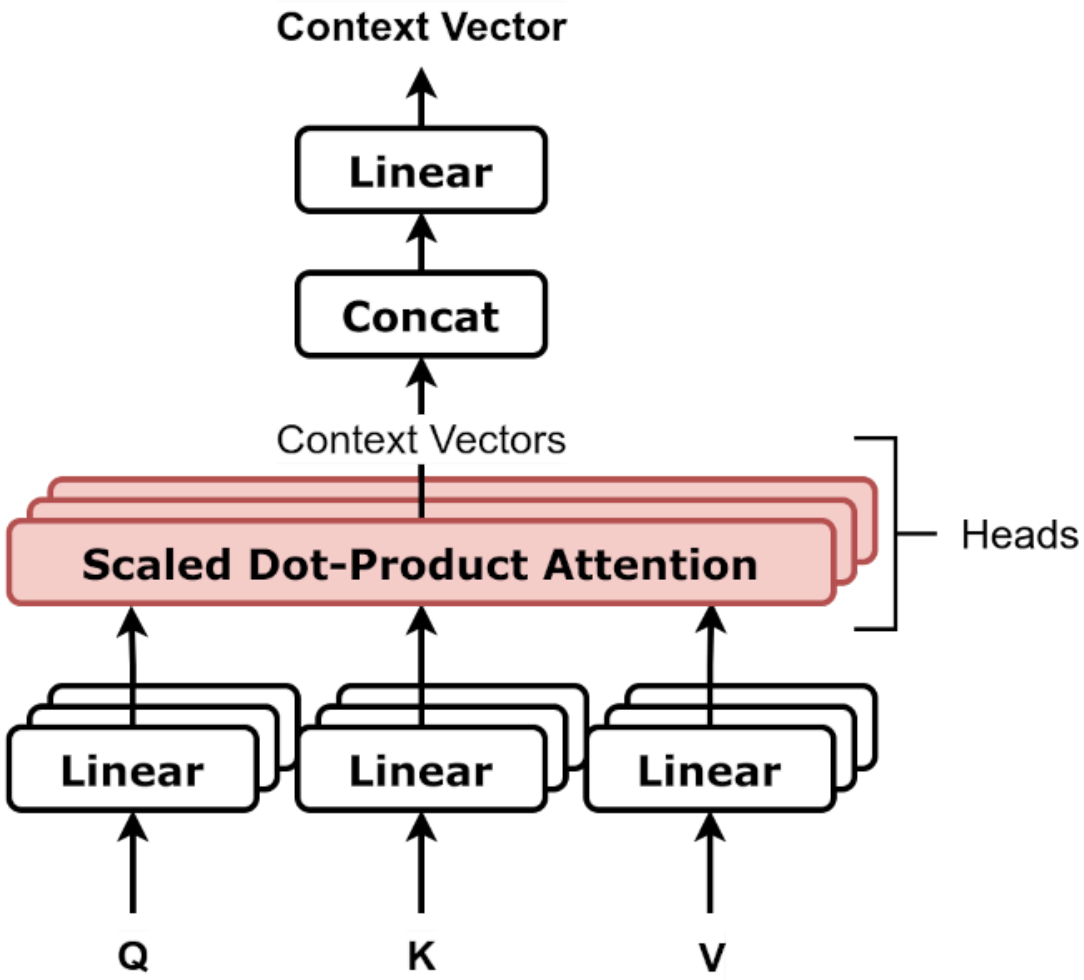
Before entering the model, **source embeddings are combined with positional encodings**, so the model knows both token identity and token position.

A transformer encoder block applies:

1. Multi-head self-attention
2. Residual connection and normalization
3. Feed-forward network
4. Residual connection and normalization



Multi-Head Attention: Block Diagram

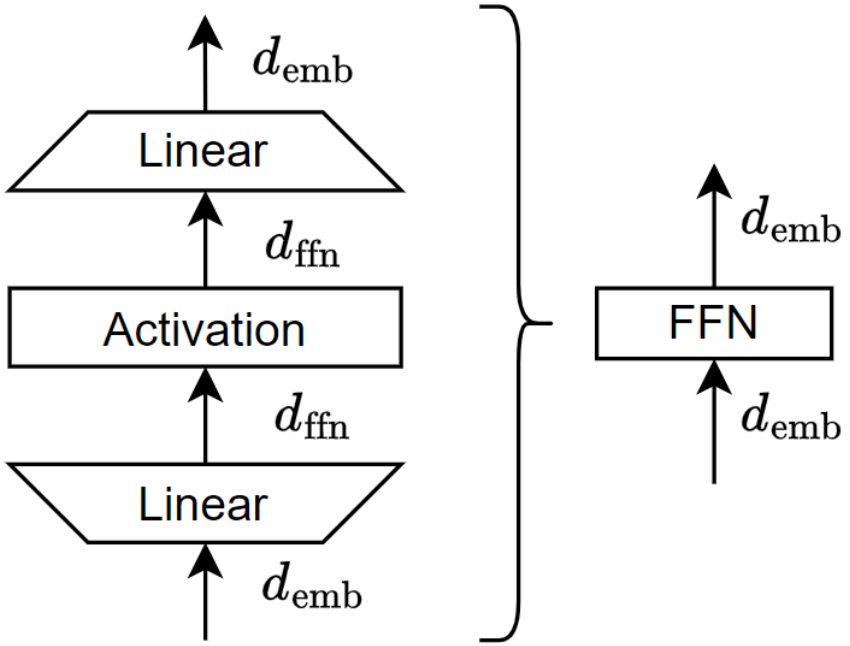


Feed-Forward Layer

After multi-head attention, each token position has an updated representation $\mathbf{h}_t$. The same feed-forward network is then applied independently to every $\mathbf{h}_t$:

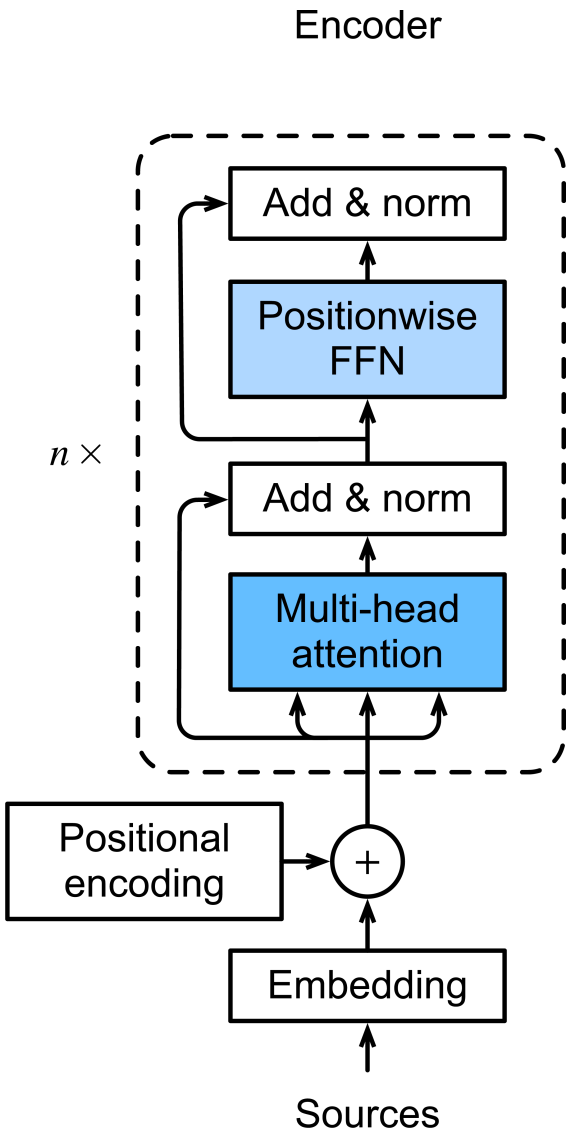
$$\mathrm{FFN}(\mathbf{h}) = \mathbf{W}_2\,\sigma(\mathbf{W}_1\mathbf{h} + \mathbf{b}_1) + \mathbf{b}_2$$

- This adds nonlinear transformation after information exchange.
- Attention mixes information across positions.
- **The feed-forward layer transforms each position independently.**



It is called position-wise because the same FFN is applied independently to each token position. It transforms each token representation separately and does not mix information across different positions.

Residual Connections and Normalization

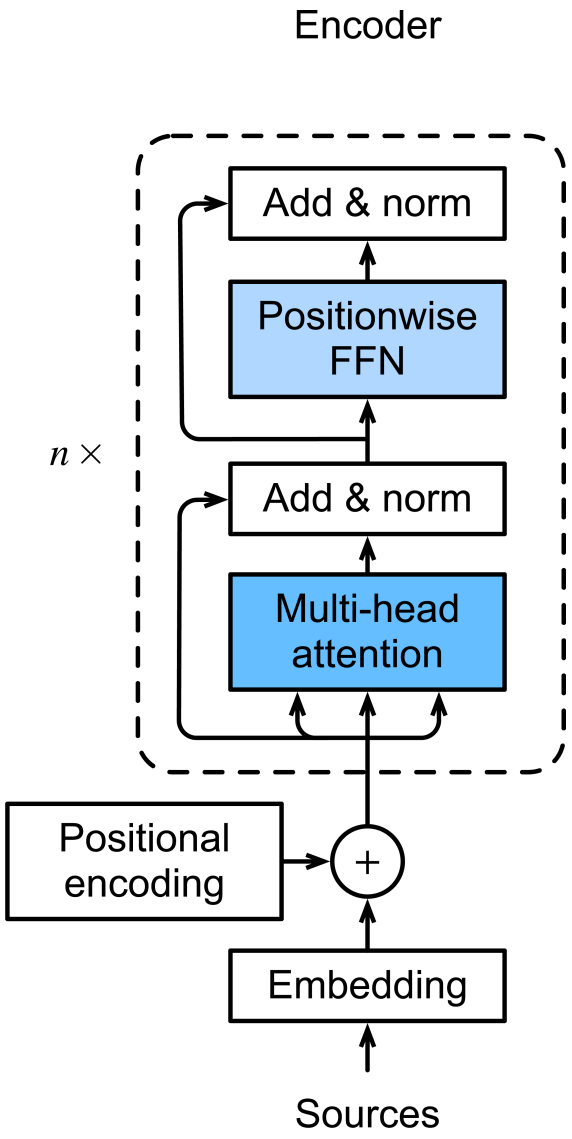


An encoder block also uses residual connections and normalizations.

- Residual connections help optimization by **allowing information and gradients to flow** through deep networks.
- Instead of forcing each layer to learn a complete transformation, **the layer learns a correction to the input representation**.
- Layer normalization stabilizes training.

In practice, residual connections and normalization are essential for training deep transformers.

Stacking Encoder Blocks



A transformer is built by **stacking several blocks**:

- Each layer produces more contextualized representations.
- Early layers may capture local or simple patterns.
- Deeper layers may capture more abstract relationships.

The Role of the Encoder

The encoder **reads** the whole input and turns it into a **contextual representation**: one vector per token,

$$\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_T] \in \mathbb{R}^{T \times d}$$

- every position sees **all the others**, to the left and to the right
- it does not produce text: the representation is used by something else, such as a classifier or a decoder

Role: understanding the input.

Decoder

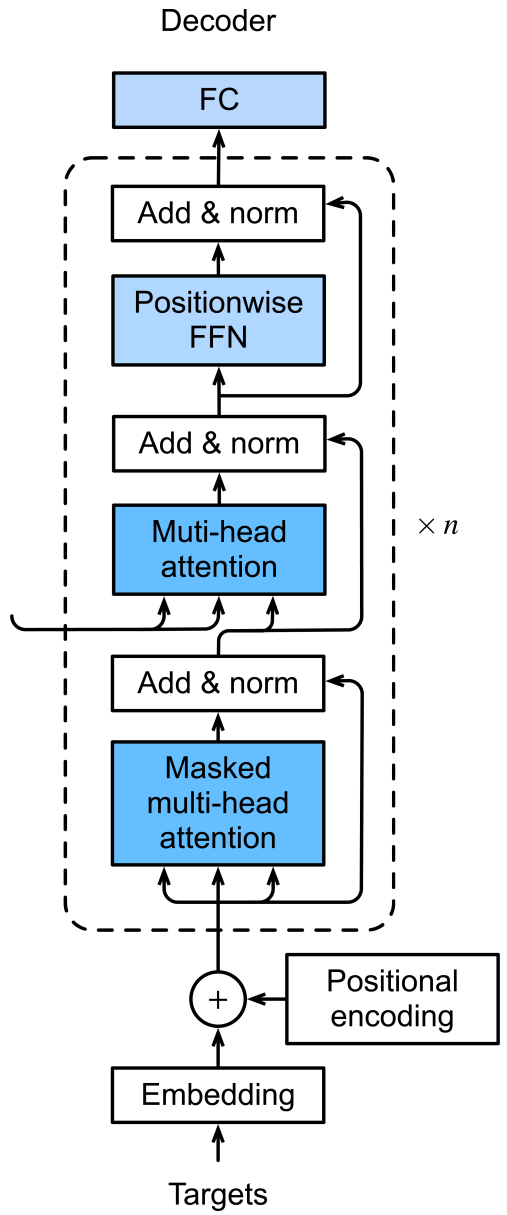
The decoder is used to **generate an output sequence one token at a time**. It receives:

- the tokens generated so far
- positional encodings
- information from the encoder output

In particular:

- At training time, the decoder sees the true previous target tokens.
- At inference time, it uses its own previous predictions.

The decoder must not look at future target tokens.
This is why it uses **masked self-attention**.



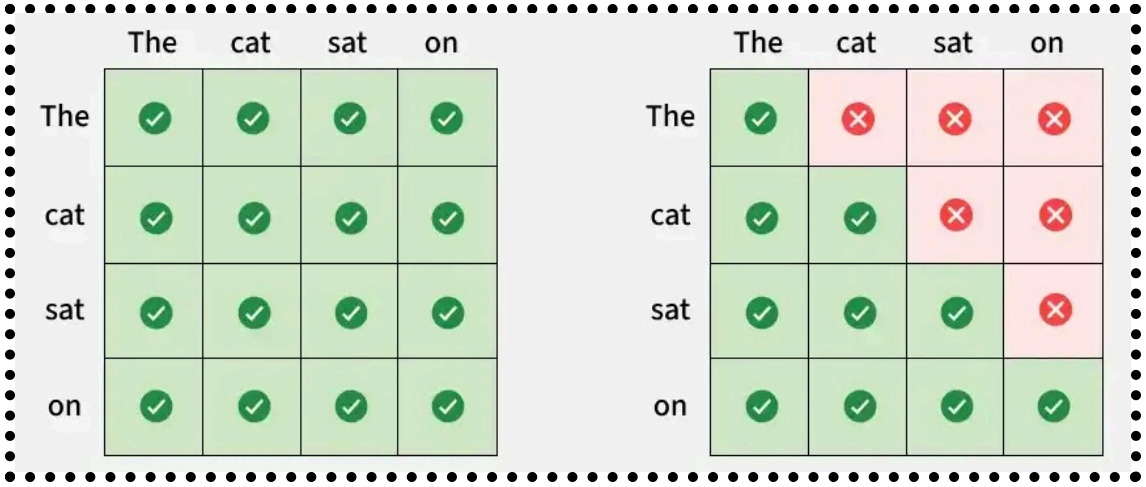
Masked Self-Attention

The decoder generates the output **one token at a time**. At inference time the future tokens **do not exist yet**, so each position can only use the tokens before it.

At training time we do know the whole target sequence, but the decoder must work as it will at inference.

Masked self-attention hides the future positions: each token attends only to previous ones.

Without the mask the model could just copy the next token.

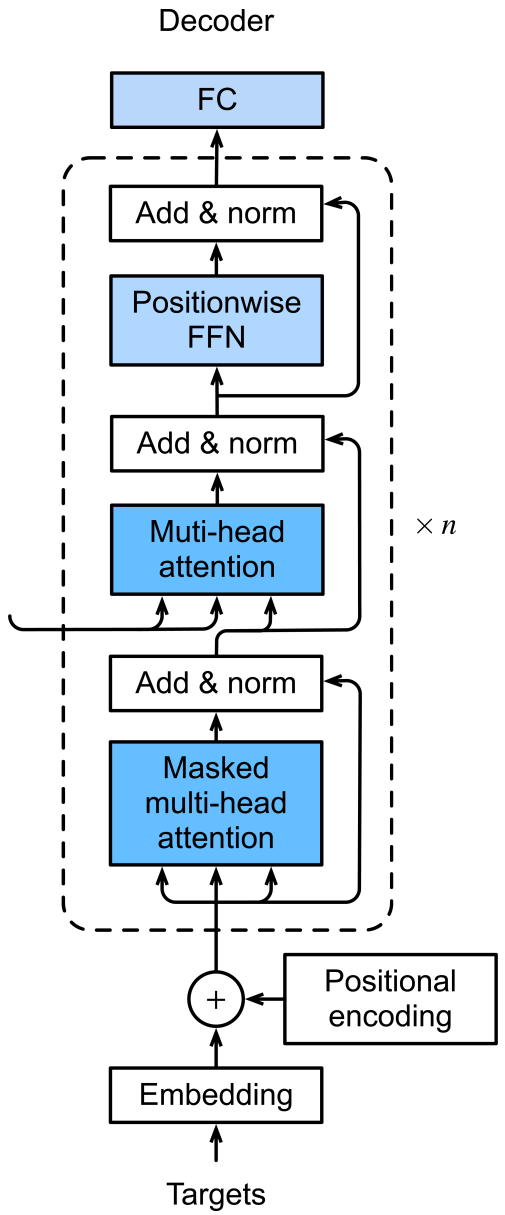


Decoder Block

Before entering the model, **target embeddings are combined with positional encodings.**

A transformer decoder block contains three main sublayers:

1. **Masked multi-head self-attention**: looks at previous target tokens
2. **Encoder-decoder attention** (cross-attention): looks at the encoder output
3. **Feed-forward network**: transforms each target position independently



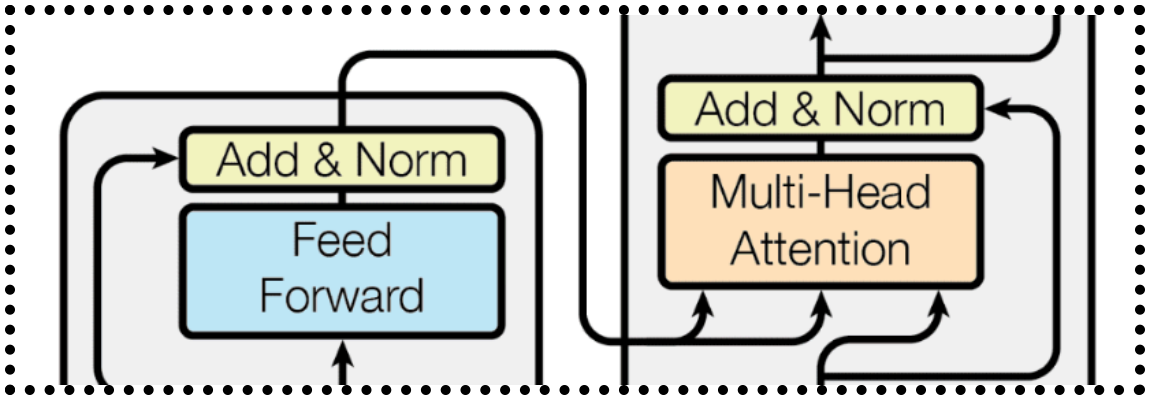
Encoder-Decoder Attention

After masked self-attention, the decoder attends to the encoder outputs with **encoder-decoder attention**, also called **cross-attention**. The decoder asks:

> Which source tokens are useful for generating the next target token?

Here:

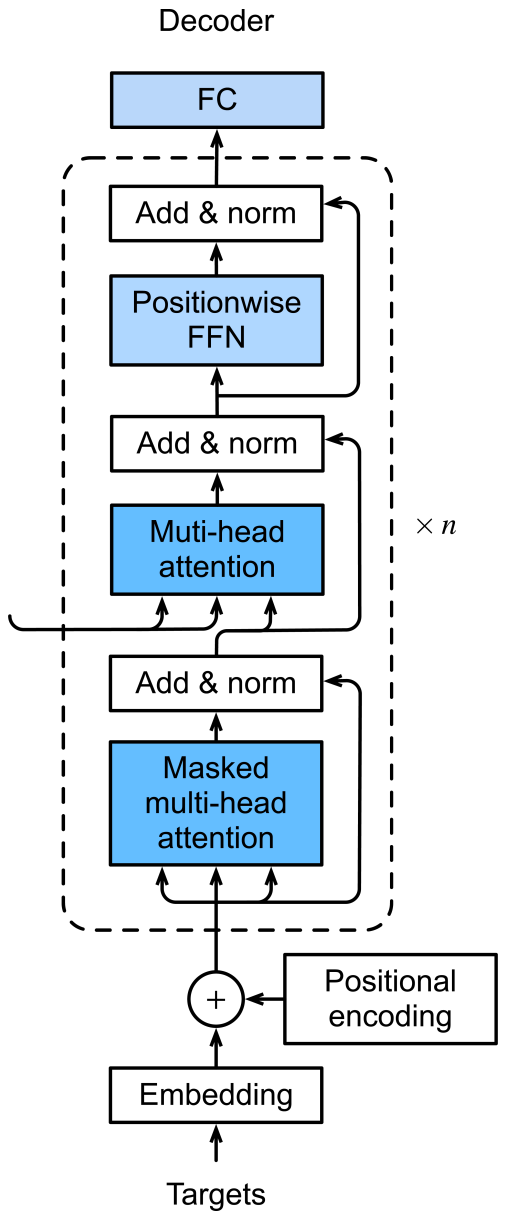
- queries come from the decoder state $\mathbf{s}$: $\mathbf{q} = \mathbf{W}^Q \mathbf{s}$
- keys and values come from the encoder outputs $\mathbf{h}_t$:
  - $\mathbf{k}_t = \mathbf{W}^K \mathbf{h}_t$,
  - $\mathbf{v}_t = \mathbf{W}^V \mathbf{h}_t$



This connects the generated target sequence to the original input sequence.

Decoder Block

- Each sublayer is followed by residual connection and normalization.
- The decoder output is a contextual representation for each target position.
- The final linear layer and softmax convert it into probabilities over the vocabulary.



The Role of the Decoder

The decoder **writes** the output sequence **one token at a time**. Each token depends on the previous ones and, in the original transformer, on the encoder's representation of the input:

$$P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{H})$$

- it is **autoregressive**: every prediction is fed back as the next input, until an end token is generated
- the **mask** lets it see only the past

Role: generating the output.

Encoder, Decoder, Encoder-Decoder

There are three common transformer families.

Architecture	Main idea	Typical use
Encoder-only	reads the full input sequence	classification, representation learning
Decoder-only	predicts the next token	generation
Encoder-decoder	maps input sequence to output sequence	translation, summarization

Many modern language models are transformer-based, but they do not all use the same architecture.

For most text classification and labeling tasks, encoder-only transformers are the natural starting point.

Famous Text Transformers

Different transformer families use different parts of the architecture.

Model family	Architecture	Main idea	Typical tasks
BERT	Encoder-only	read the full sequence bidirectionally	classification, tagging, representation learning
GPT	Decoder-only	predict the next token autoregressively	text generation, chat, reasoning
T5	Encoder-decoder	convert every task into text-to-text	translation, summarization, QA
BART	Encoder-decoder	reconstruct corrupted text	summarization, generation

Next: how these models are pre-trained, adapted and used.

Sources

Textbooks

- Zhang, Lipton, Li & Smola, *Dive into Deep Learning*, chapters on attention mechanisms and transformers: queries, keys and values, attention pooling, bilinear and dot-product scoring, Bahdanau attention, multi-head attention
- Jurafsky & Martin, *Speech and Language Processing*, 3rd ed., ch. 7 (transformers)

Papers

- Sutskever, Vinyals & Le (2014), *Sequence to Sequence Learning with Neural Networks*
- Bahdanau, Cho & Bengio (2014), *Neural Machine Translation by Jointly Learning to Align and Translate*
- Vaswani et al. (2017), *Attention Is All You Need*