ARTIFICIAL INTELLIGENCE 2

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 dense vector:

$$\text{token} \longrightarrow \mathbf{e} \in \mathbb{R}^d$$

For a sequence of tokens:

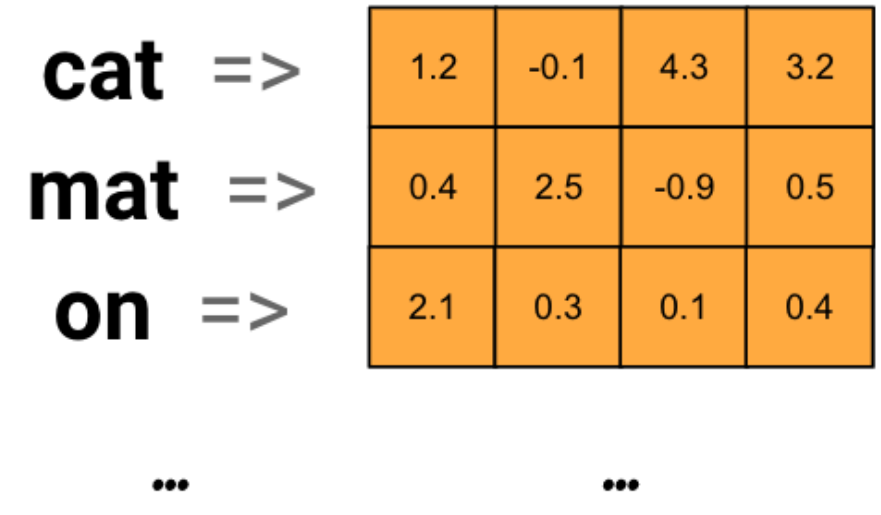
$$[w_1, w_2, \ldots, w_T]$$

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?

A 4-dimensional embedding



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

Besides the natural language processing domain, in healthcare and biology, the meaning of one element often depends on other elements.

Examples:

- A medication is meaningful given previous diagnoses.
- A clinical word depends on the sentence.
- A protein residue depends on nearby and distant residues.
- A DNA variant depends on its sequence context.
- A patient visit depends on previous visits.

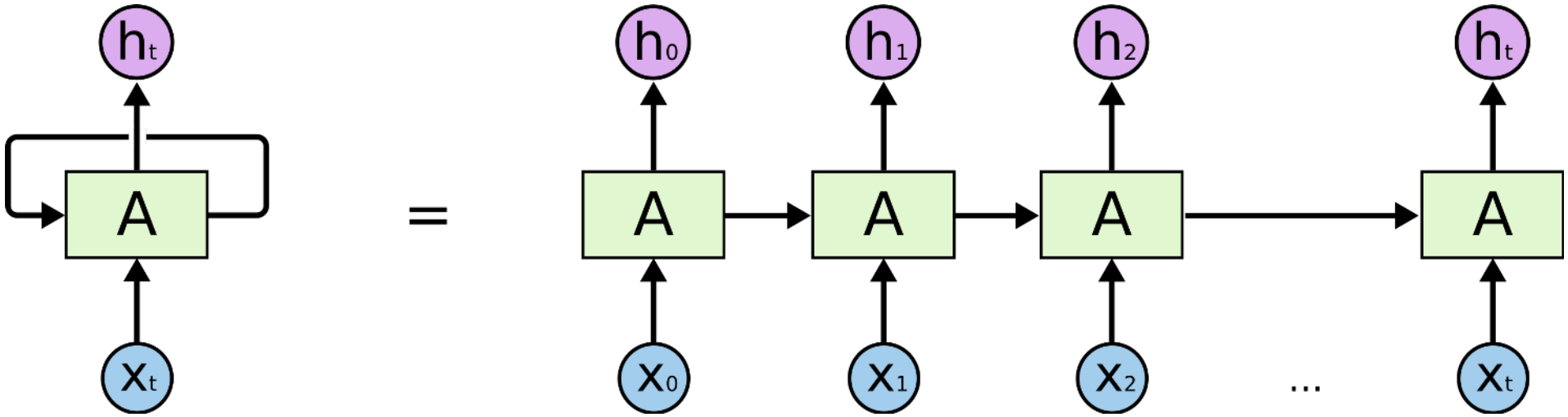
We need models that can capture **dependencies across a sequence**.

Recurrent Neural Networks

As we saw in 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, $\mathbf{h}_{t-1}$

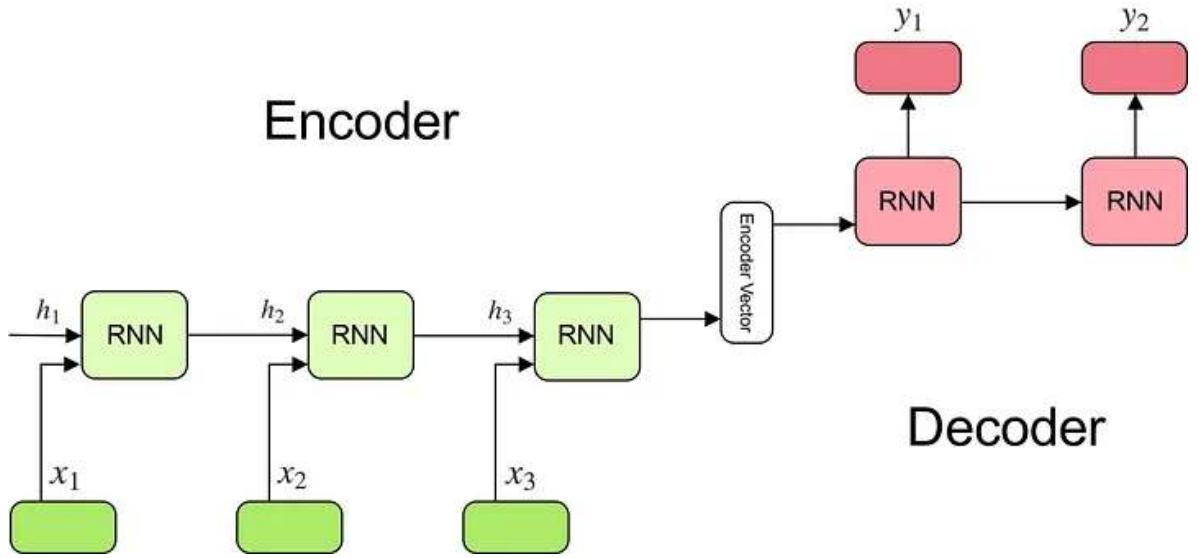
The hidden state is then passed to the next time step.



Sequence-to-Sequence Learning

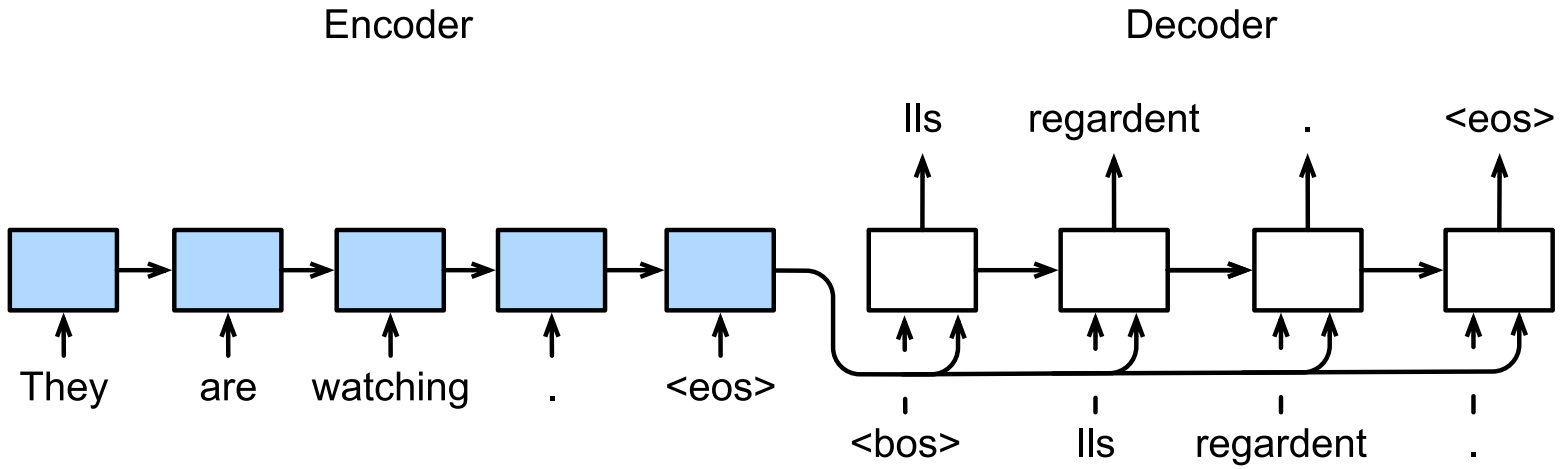
Let us consider machine translation as an example. When generating a translation of a source text, we first pass the source text through an encoder to obtain a sequence of encoder hidden states, and the last one is used by the decoder.

- 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 quite reasonable for short sequences, it is clear that it is infeasible for long ones

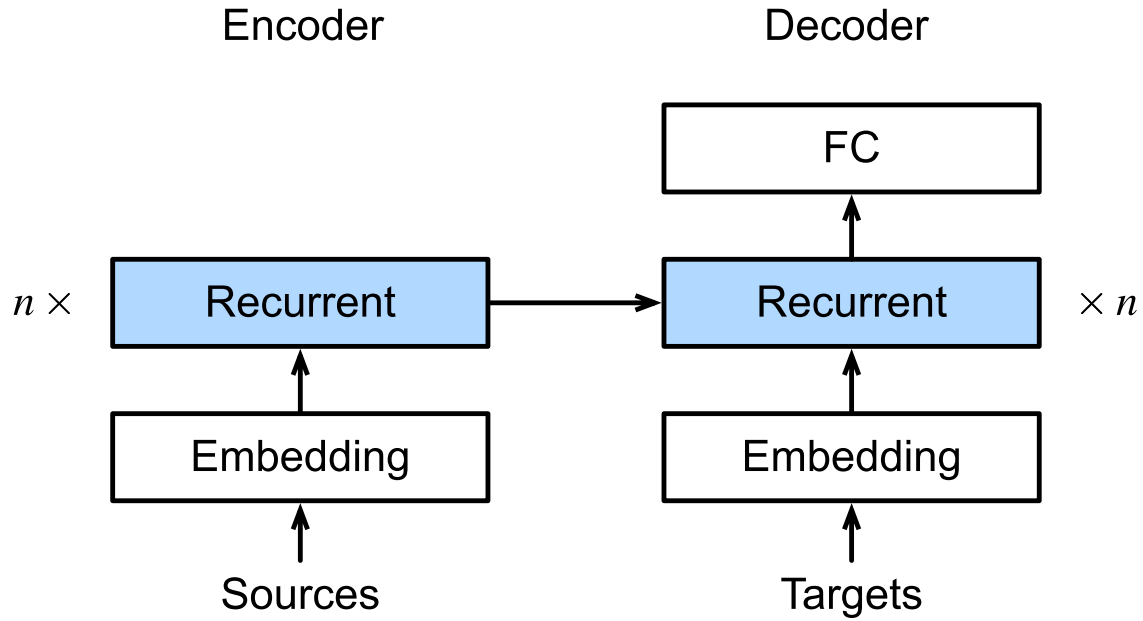
Sequence-to-Sequence Learning



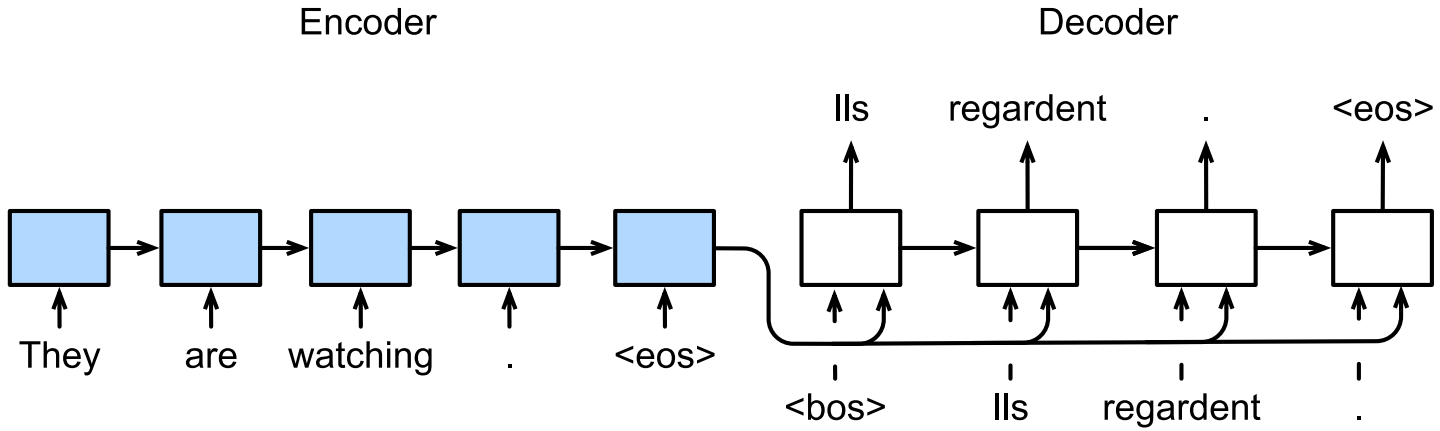
The encoder transforms the hidden states at all time steps into a context vector through a customized function:

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

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



Sequence-to-Sequence Learning



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 occurring at step $t' + 1$, based on the current state, $\mathbf{s}_{t'}$, which contains information about:

- the previous step's target token
- the hidden RNN state from the previous time step
- and the context variable

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}).$$

Sequence-to-Sequence Learning

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 and Transformers

The Attention Mechanism

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 operate 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 dabase 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)

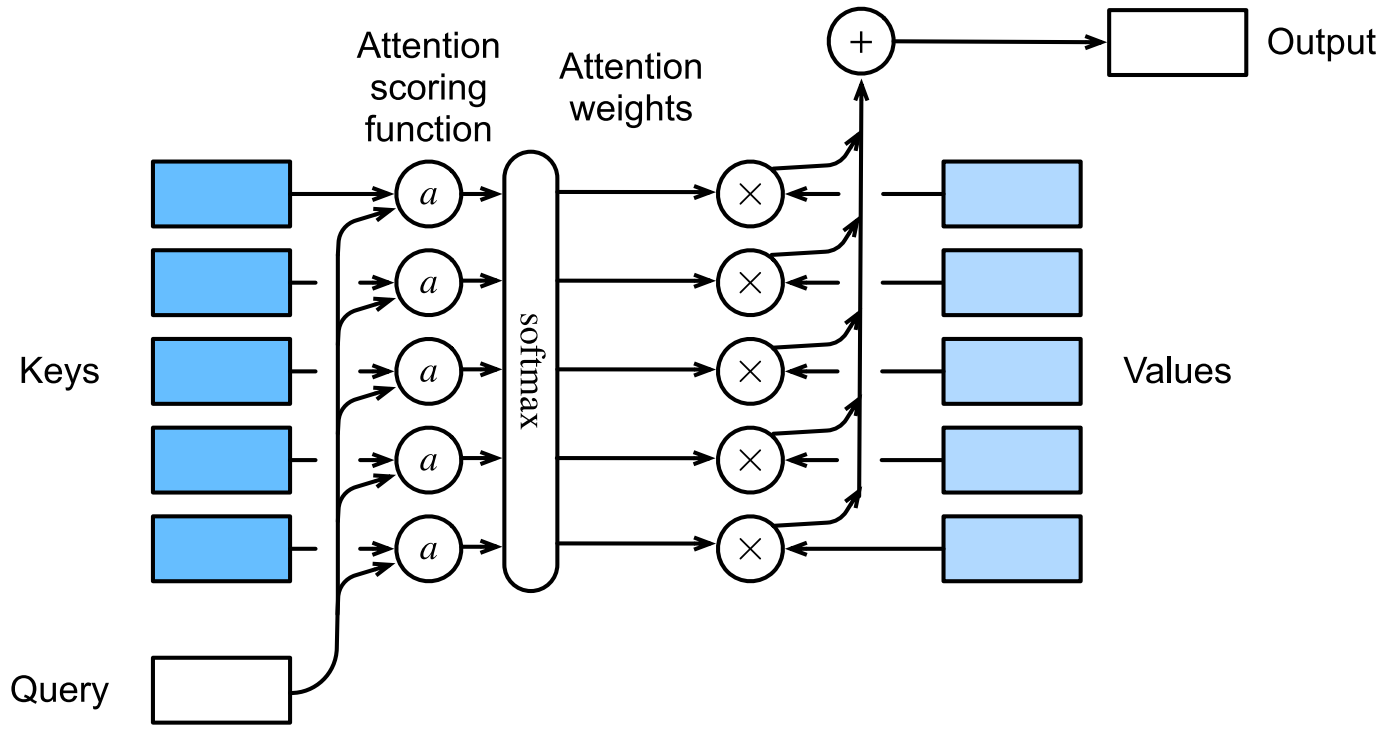
Dot Product Attention

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:

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



*The reason is that large dot products push softmax into saturation/vanishing gradient territory

Dot Product Attention Example

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

Key $\mathbf{k}_i$	Value $\mathbf{v}_i$	$\mathbf{q}^\top \mathbf{k}_i$	Weight α_i
$[0.9, 0.1]$	$[10, 2]$	0.89	0.51
$[0.4, 0.6]$	$[4, 8]$	0.59	0.38
$[-0.7, 0.2]$	$[1, 9]$	-0.60	0.11

Attention output:

$$0.51 \begin{bmatrix} 10 \\ 2 \end{bmatrix} + 0.38 \begin{bmatrix} 4 \\ 8 \end{bmatrix} + 0.11 \begin{bmatrix} 1 \\ 9 \end{bmatrix} \approx \begin{bmatrix} 6.73 \\ 5.05 \end{bmatrix}$$

Attention is Flexible

If the query and key dimension do not match, i.e., $\mathbf{q} \in \mathbb{R}^{d_q}$ and $\mathbf{k} \in \mathbb{R}^{d_k}$, with $d_k \neq d_q$, instead of $\mathbf{q}^\top \mathbf{k}$ you can use:

$$\text{Bilinear Attention:} \quad \mathbf{q}^\top \mathbf{M} \mathbf{k}$$

where $\mathbf{M} \in \mathbb{R}^{d_q \times d_k}$. $\mathbf{M}$ is a learnable matrix that make the dimensions of query and key compatible: $(1 \times d_q)(d_q \times d_k)(d_k \times 1) = 1 \times 1$

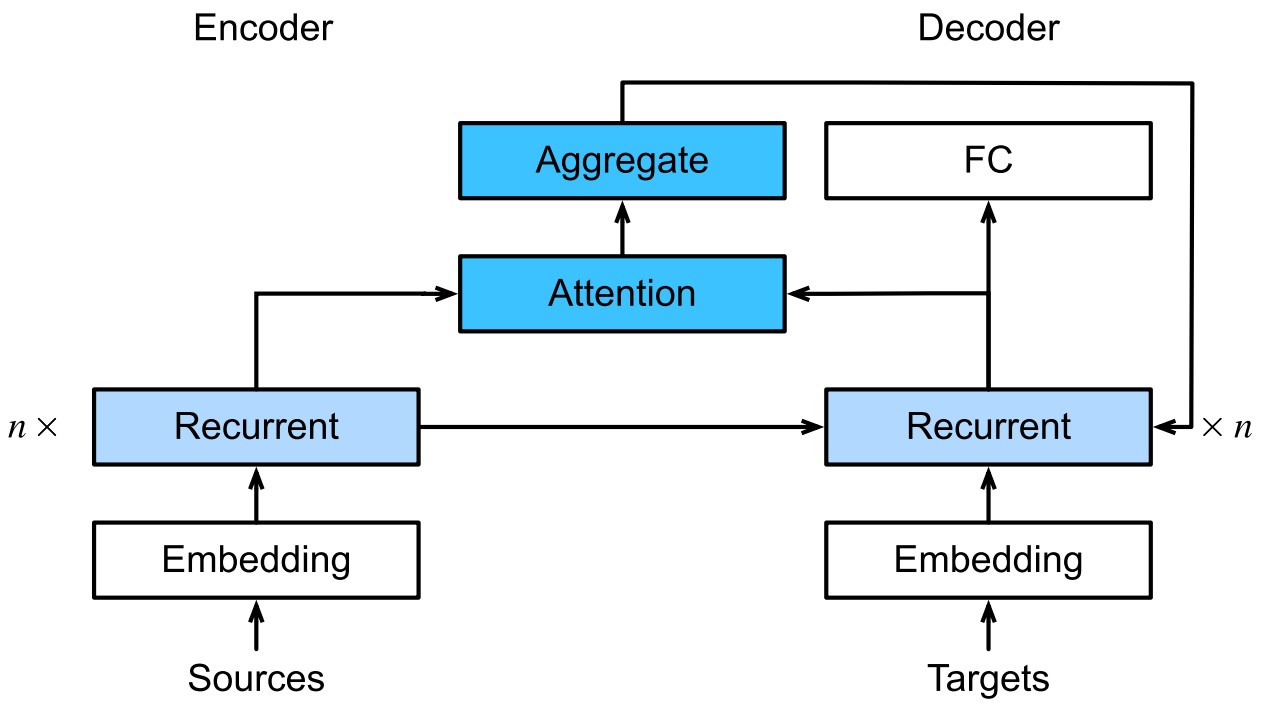
Bahdanau Attention Mechanism

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:

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



Bahdanau Attention Mechanism

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!

Self-Attention

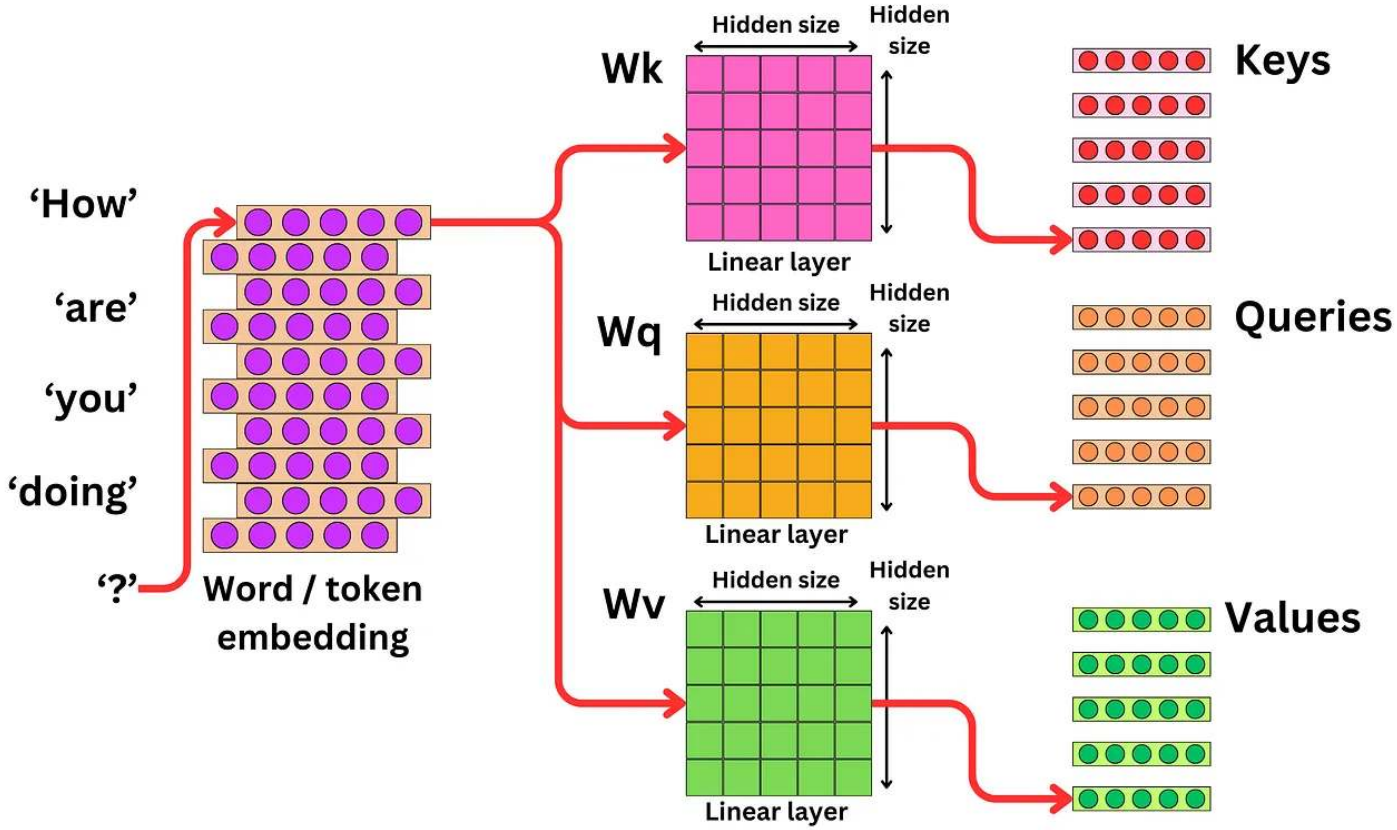
Instead of using the attention mechanism in a recurrent neural network, imagine feeding a sequence of tokens, $\mathbf{x}_1, \ldots, \mathbf{x}_T$ into an attention mechanism such that at every step, each token, $\mathbf{x}_t$ has its own query, keys, and values.

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

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

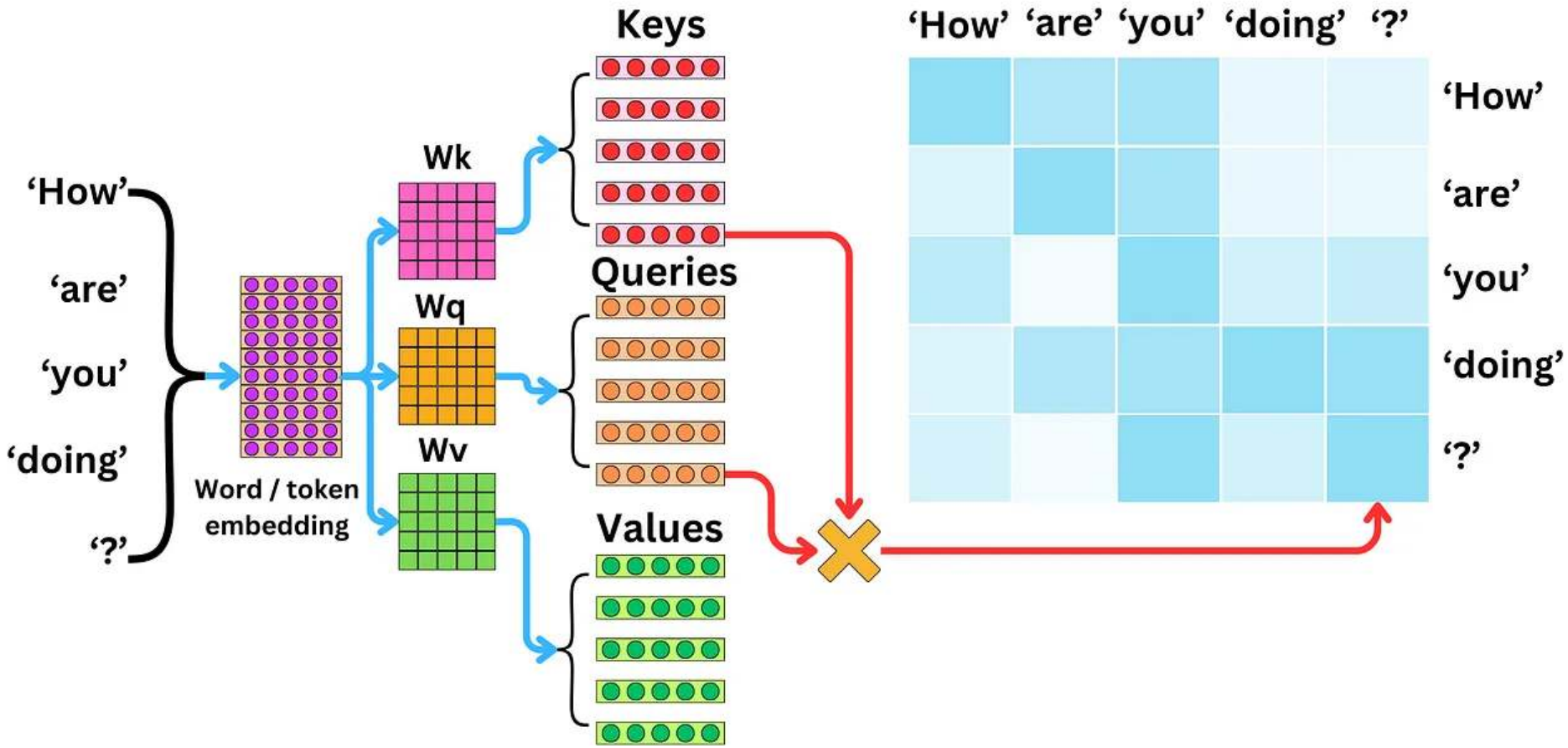
$$\mathbf{k}_t = \mathbf{W}^K \mathbf{x}_t,$$

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



Self-Attention

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

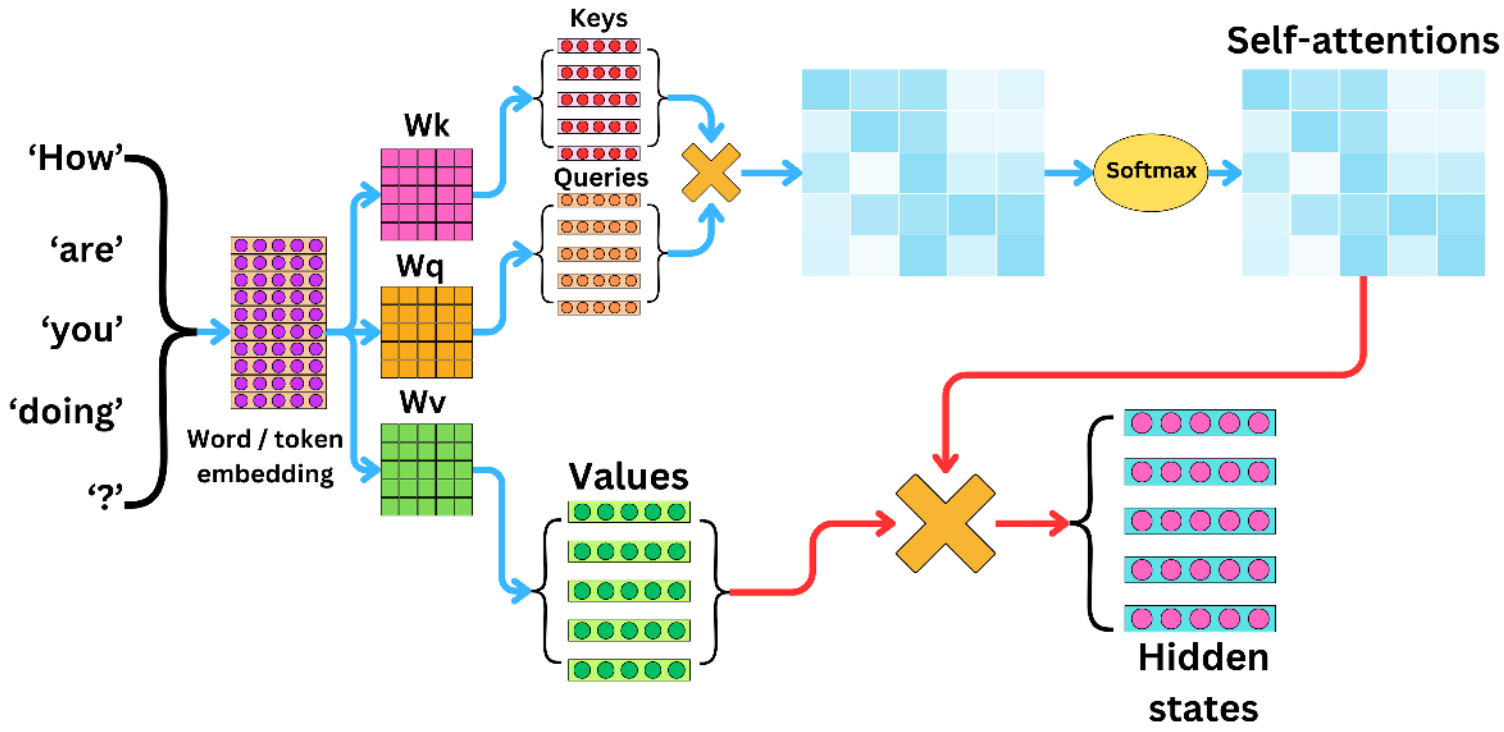


Self-Attention

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

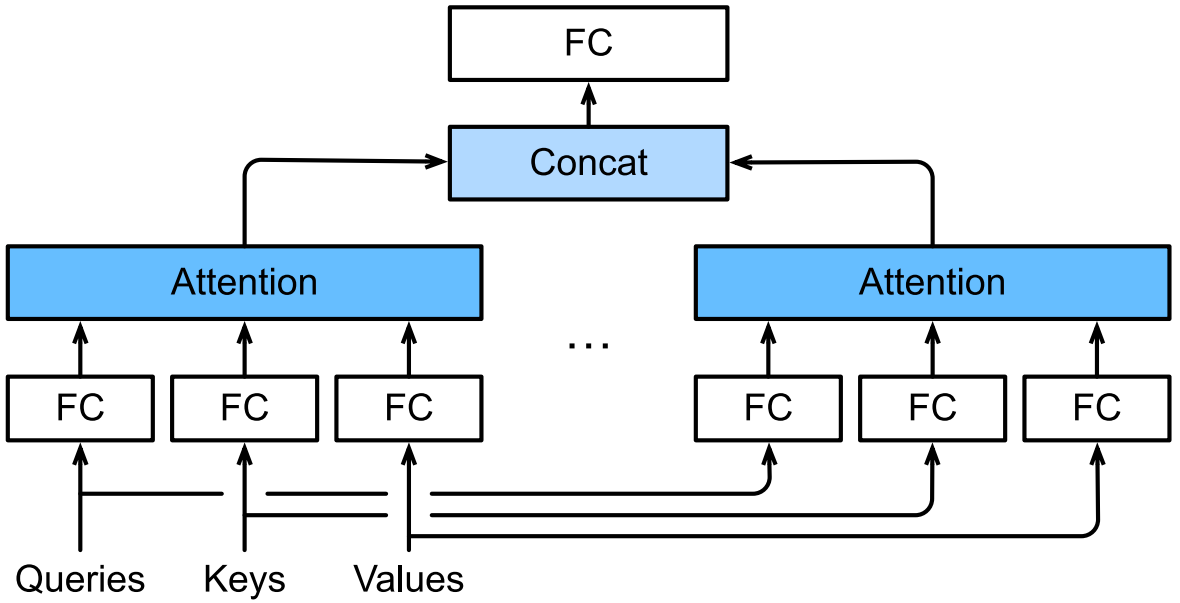
- the attention matrix
- and the values vectors.



Multi-Head Attention

In practice, given the same set of queries, keys, and values we may want our model to **combine knowledge from different behaviors of the same attention mechanism**. Instead of performing a single attention pooling:

- queries, keys, and values are transformed with **independently** learned linear projections;
- projected queries, keys, and values are fed into attention pooling **in parallel**;
- **attention-pooling outputs are concatenated** and transformed with another learned linear 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 clinical sequences:

- one head may focus on recent visits
- one head may focus on medications
- one head may focus on diagnoses
- one head may capture long-range dependencies

For biological sequences:

- one head may focus on local motifs
- one head may capture long-range residue interactions
- one head may focus on repeated patterns

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, these two sequences contain the same vectors, but in a different order:

$$[\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3] \quad \text{and} \quad [\mathbf{x}_3, \mathbf{x}_1, \mathbf{x}_2]$$

- Without positional information, self-attention sees only the vectors, not whether a token came first, second, or third.
- Therefore, transformers add positional encodings to the token embeddings.

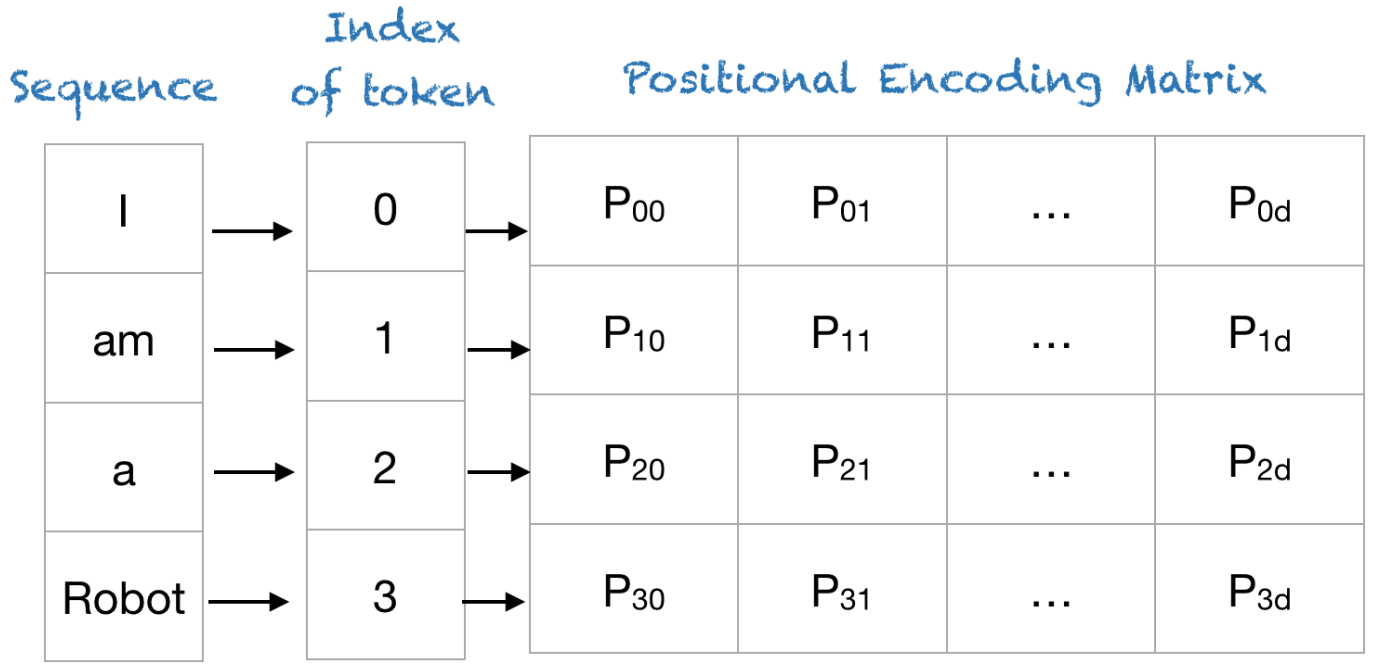
Positional Encodings

We add a positional vector to each token embedding:

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

where:

- $\mathbf{x}_t$ 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 then distinguish the same token appearing in different positions:

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

Attention and Transformers

The Transformer Architecture

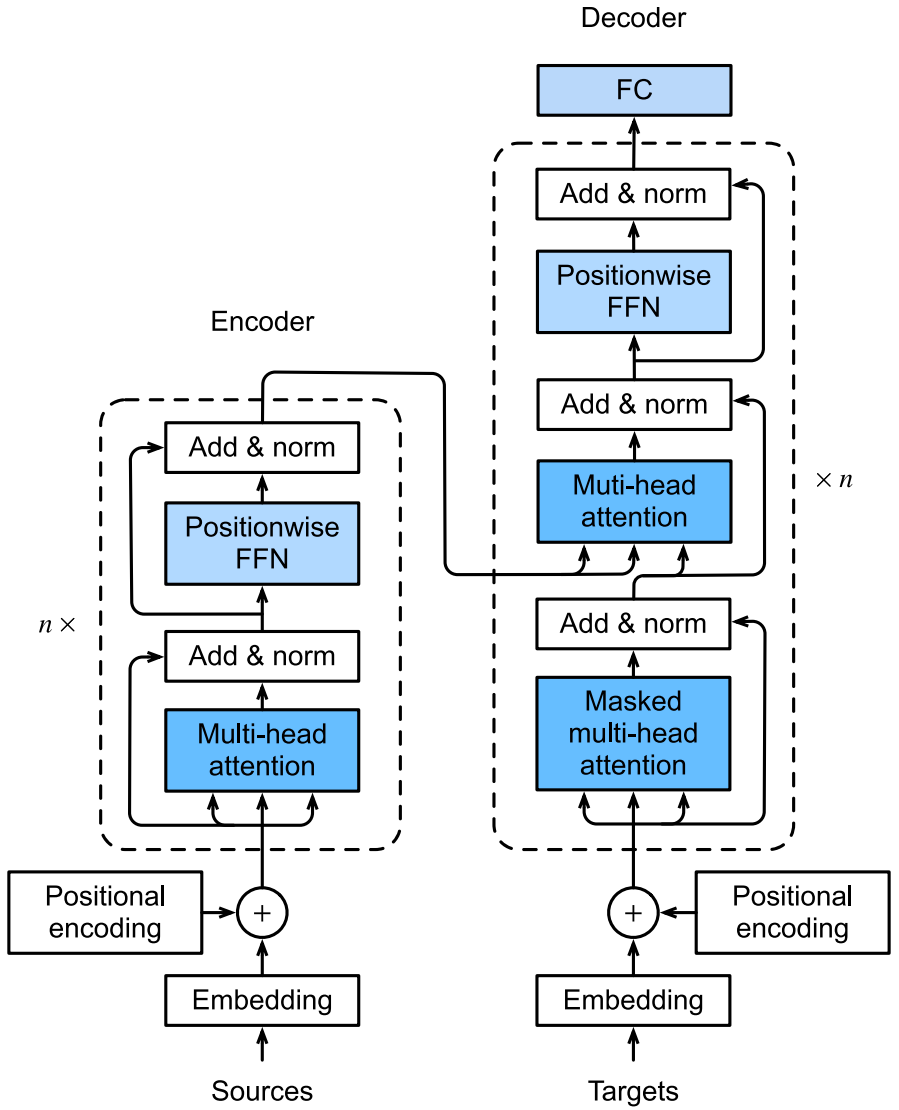
The Transformer

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

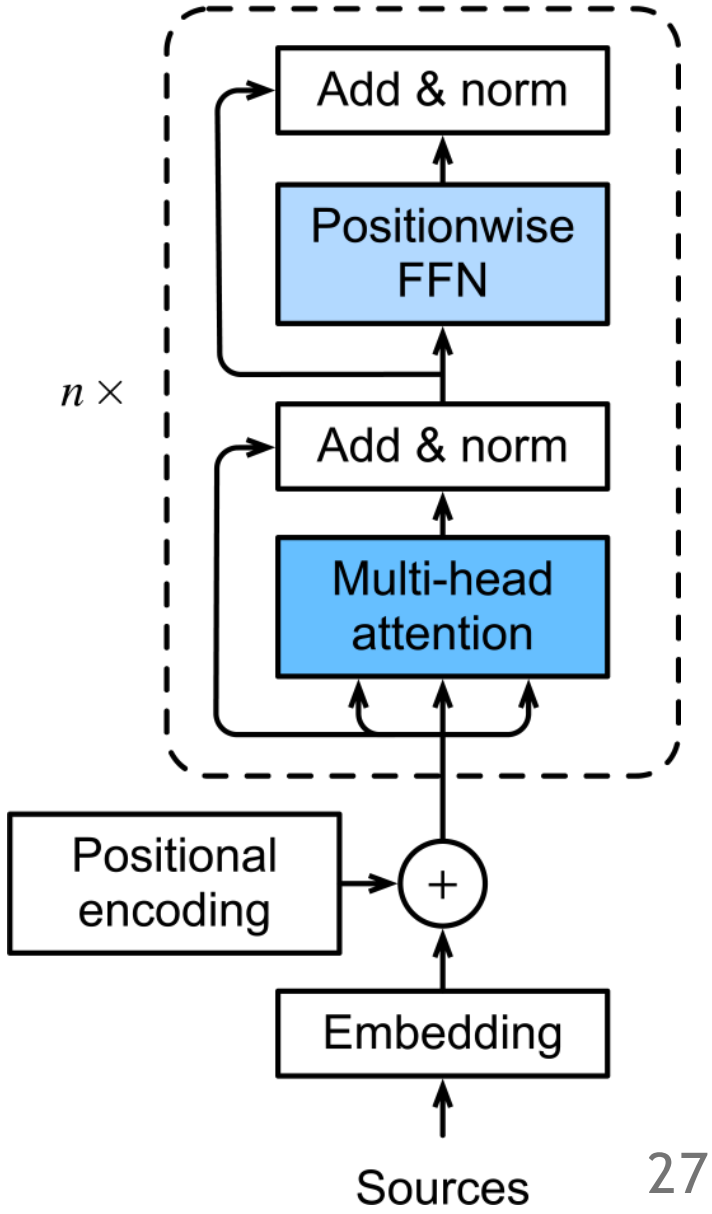
Before entering the model, source and target embeddings are combined with positional encodings, so the model knows both token identity and token position.



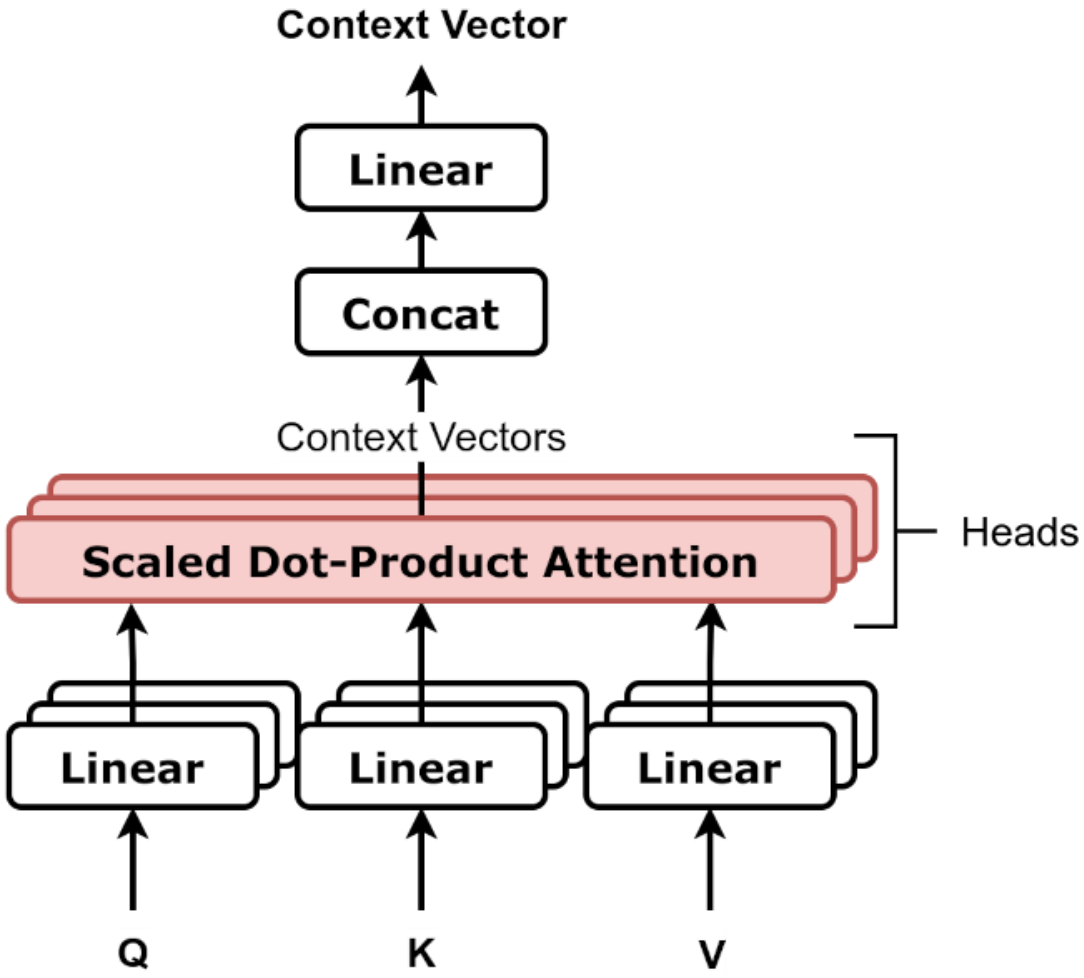
Transformer Encoder Block

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

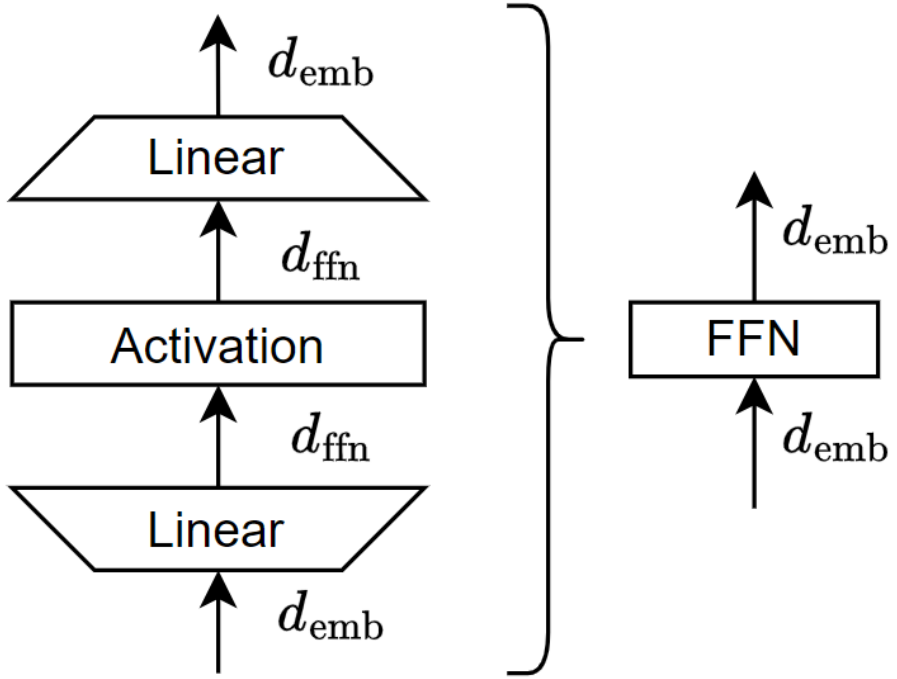


Feed-Forward Layer

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 $\mathbf{h}_t$:

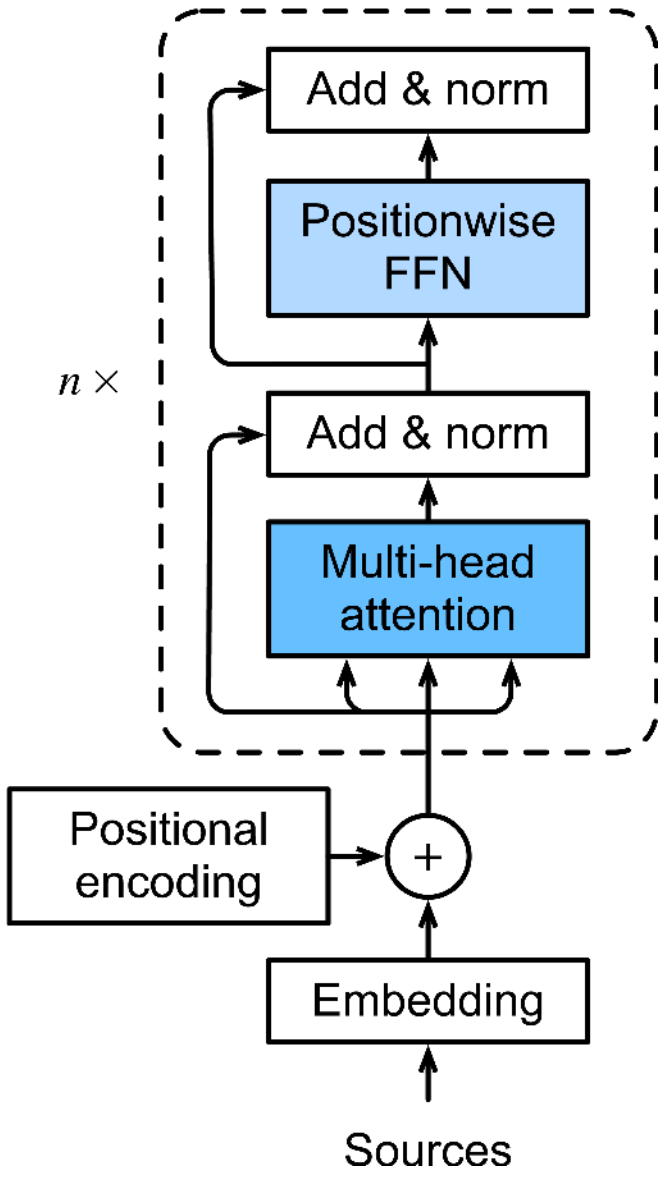
$$\mathrm{FFN}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

- 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

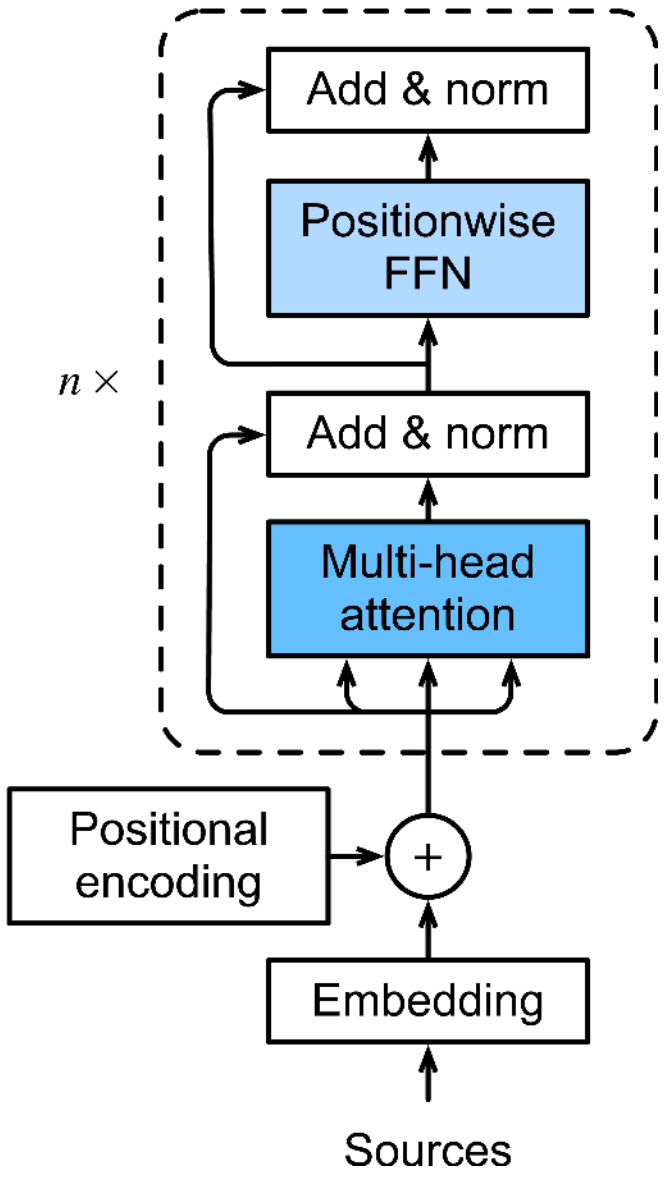


An encoder block also uses residual connections and normalizations.

- 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.

Transformer for Classification

The encoder is enough for many tasks. For classification, a transformer encoder produces contextual representations:

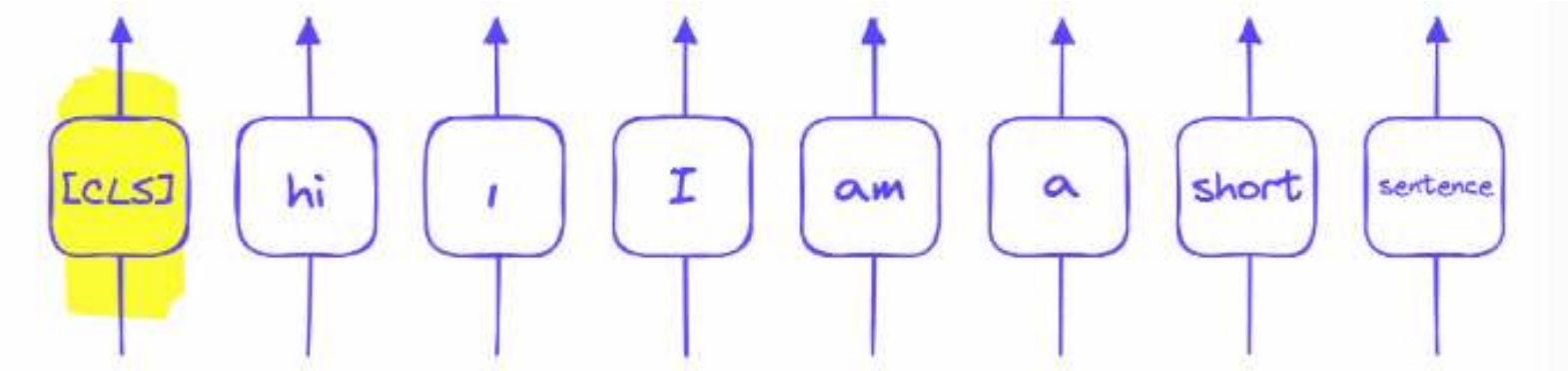
$$\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_T]$$

Then we need one vector for the whole sequence. Common strategies:

- Use the encoder output corresponding to a special classification token, $\mathbf{h}_{[\text{CLS}]}$
- average the token representations
- use the last representation for ordered sequences

Then a classifier predicts:

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{h}_{\text{seq}}\mathbf{W} + \mathbf{b})$$



Decoder

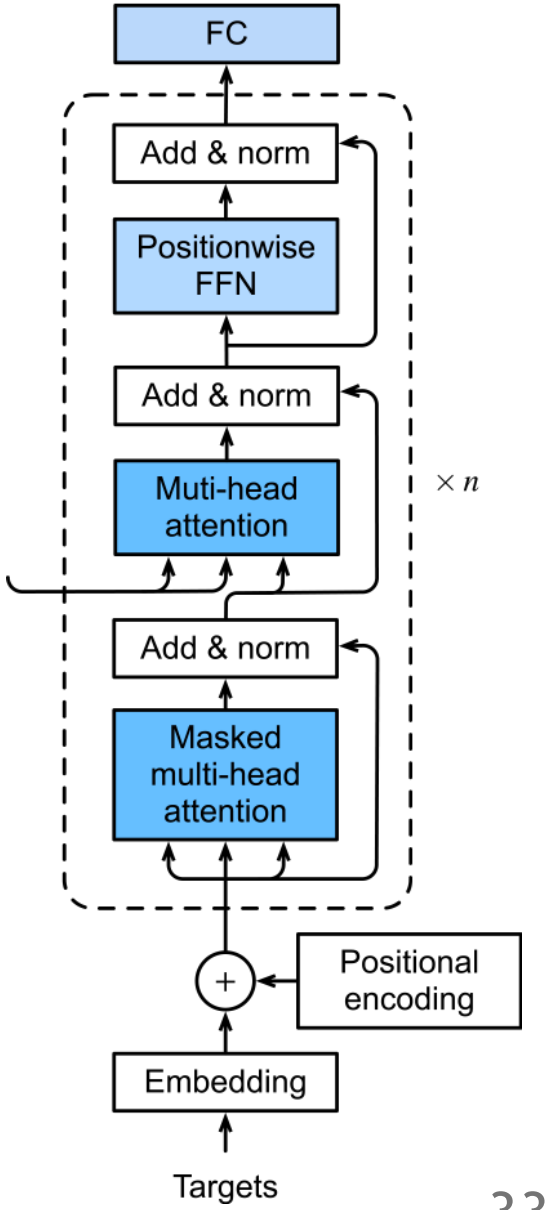
The decoder is used to generate an output sequence one token at a time. It receives:

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

Then:

- 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**.



Decoder Block

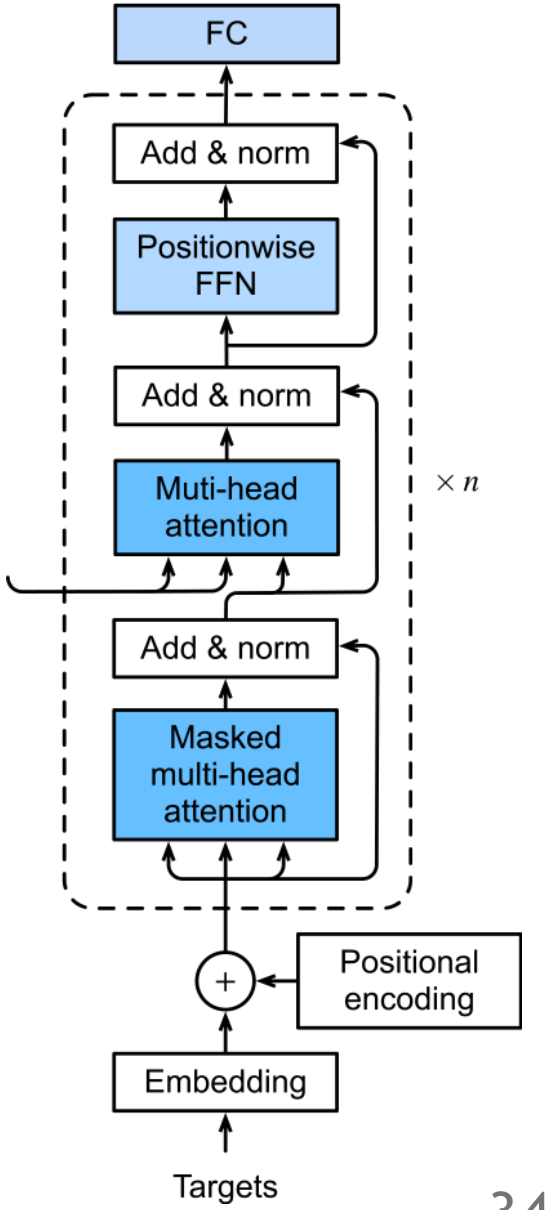
A transformer decoder block contains three main sublayers:

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

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.



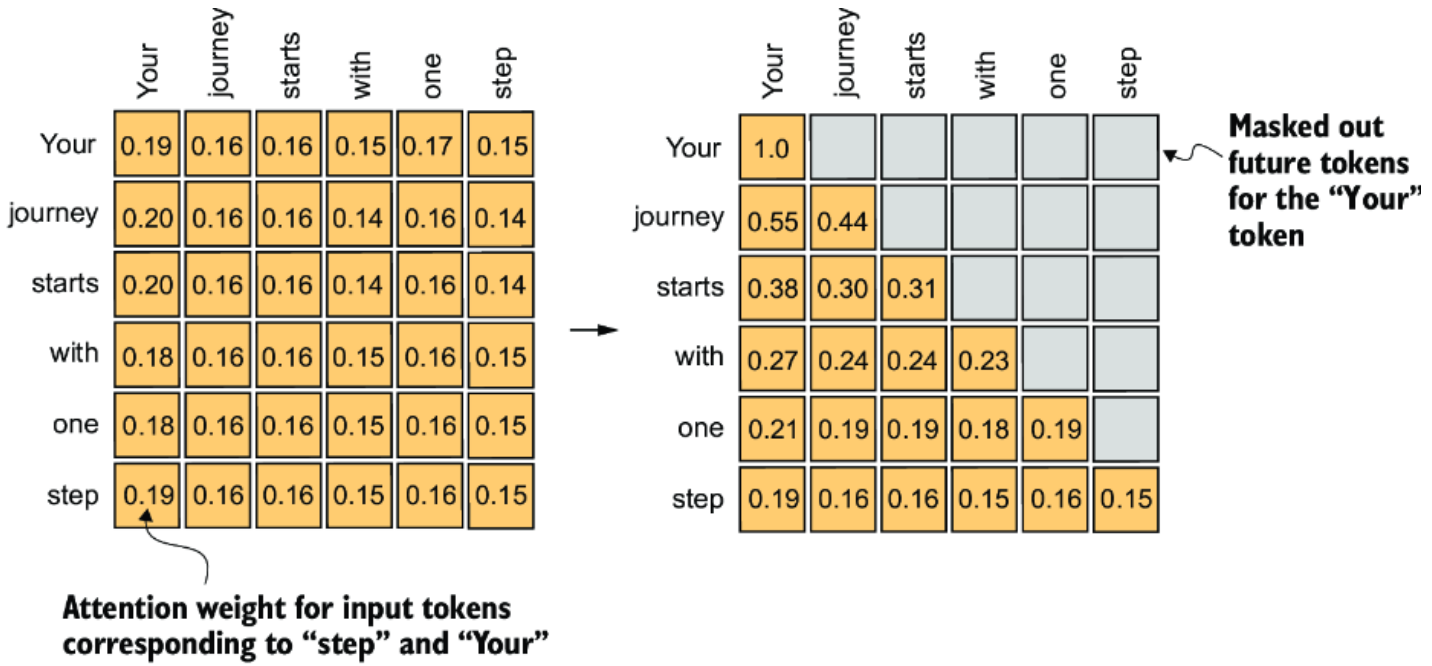
Masked Self-Attention

In the decoder, each target position can attend only to previous target positions.

For example, when predicting token y_4, the decoder can use:

$y_1,\ y_2,\ y_3$ but not: $y_5,\ y_6, \ldots$

This prevents information leakage during training.
This is called a **causal mask**.



Encoder-Decoder Attention

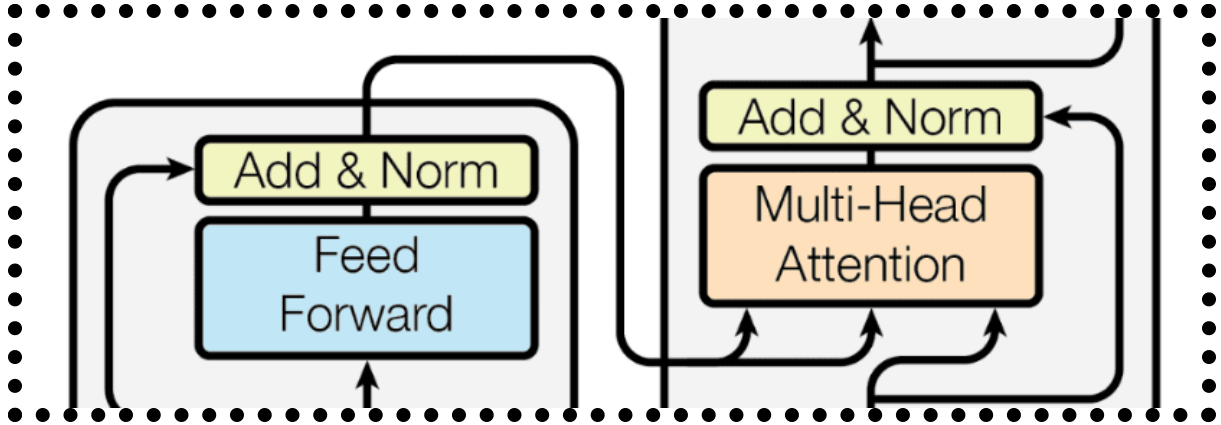
After masked self-attention, the decoder attends to the encoder outputs. The decoder asks:

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

Here:

- queries come from the decoder
- keys and values come from the encoder

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



Autoregressive Generation

So the decoder is **autoregressive**: each prediction depends on previous predictions. During inference, the decoder generates tokens sequentially.

1. Start with a special beginning token.
2. Predict the next token.
3. Append the predicted token to the input.
4. Repeat until an end token is generated.

Example:

$$<\text{bos}> \rightarrow y_1$$

$$<\text{bos}>, y_1 \rightarrow y_2$$

$$<\text{bos}>, y_1, y_2 \rightarrow y_3$$

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 many healthcare and biological prediction tasks, encoder-only transformers are often the most 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

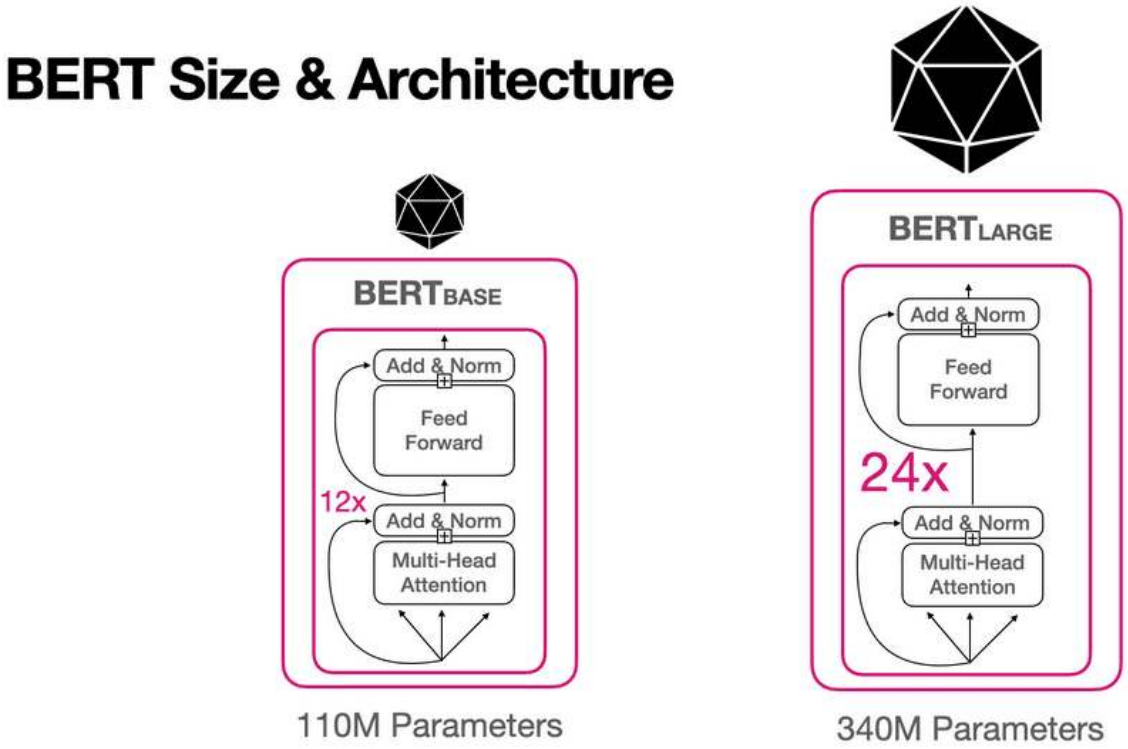
BERT: Encoder-Only Transformer

BERT uses only the transformer encoder. It reads the whole input sequence at once and produces contextual representations for all tokens.

- BERT is not naturally a text generator.
- It is mainly useful when the model must understand or classify an input sequence.

Typical uses:

- text classification
- named entity recognition
- clinical-note classification
- medical code prediction
- extracting sentence or patient representations



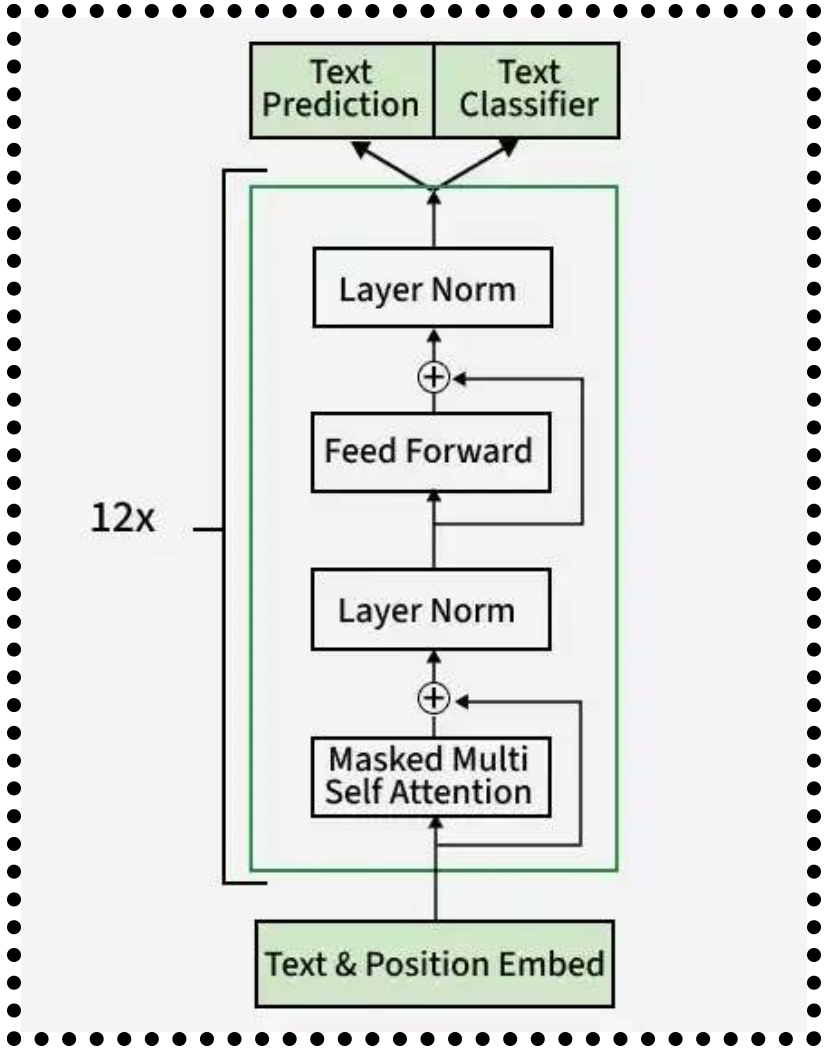
GPT: Decoder-Only Transformer

Generative Pre-trained Transformers (GPT) use only the transformer decoder.

They are trained to predict the next token. During generation, the model repeatedly predicts one token at a time. Typical uses:

- text generation
- question answering
- dialogue
- summarization
- code generation

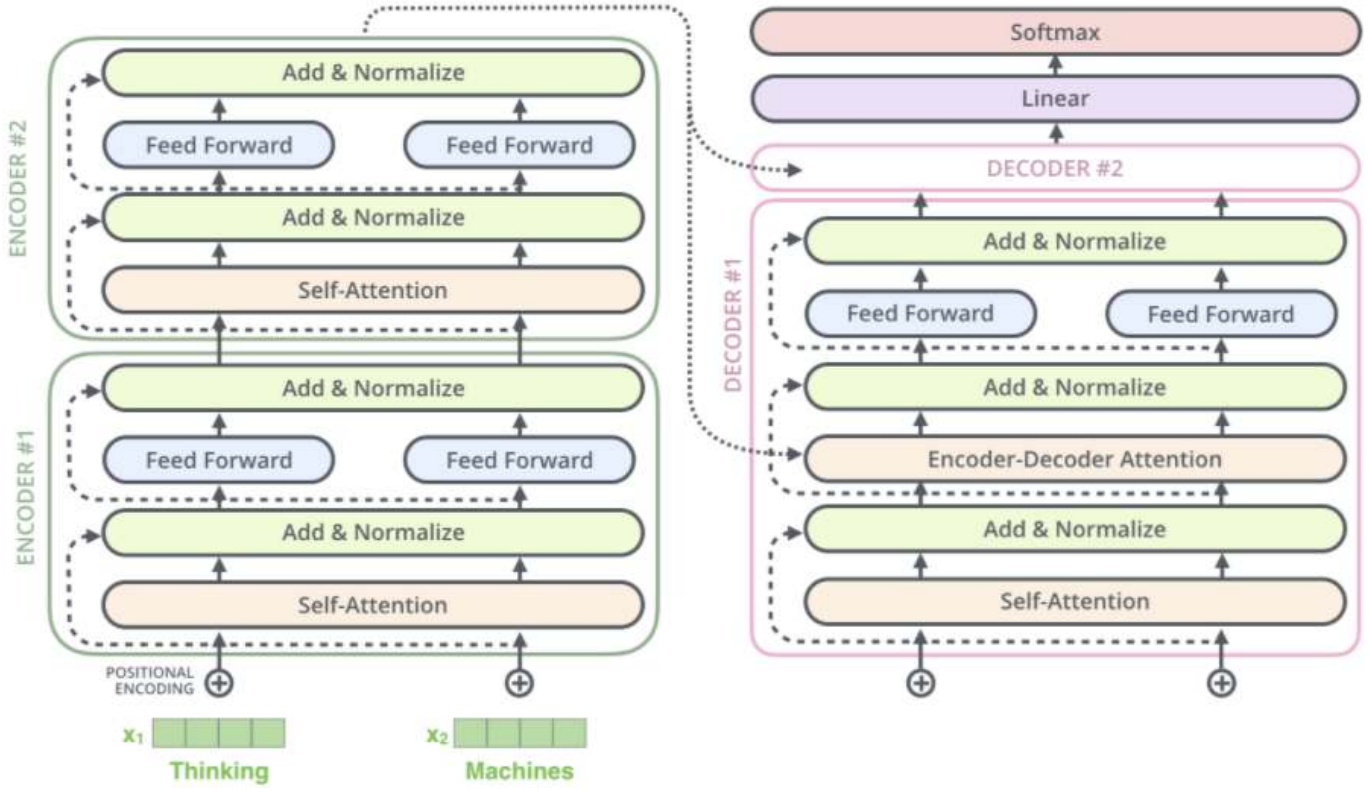
GPT-style models are autoregressive language models.



Encoder-Decoder Transformers

Encoder-decoder models use both parts of the transformer.

- The encoder reads the input sequence.
- The decoder generates the output sequence.



Examples include T5 and BART.
Typical uses:

- translation
- summarization
- text rewriting
- question answering
- report generation

Why Transformers Became Important

Transformers became dominant because they combine:

- contextual representations
- long-range dependency modeling
- parallelizable training
- scalability to large datasets
- flexible use across text, images, signals, and biological sequences

The same basic mechanism can be reused across many data modalities.

This is why transformers are central to modern foundation models.

Large Language Models

Large Language Models are transformer-based language models trained on very large text collections.

- Most modern LLMs are decoder-only transformers for next-token prediction
- Their main capability is given a context, predict a plausible continuation.

After large-scale pretraining, they can be adapted or prompted for many tasks:

- answering questions
- summarizing documents
- generating text
- extracting information...

The core mechanism is still the transformer: embeddings, self-attention, feed-forward layers, and autoregressive generation.

Attention and Transformers

Healthcare Applications

Healthcare Examples

Transformers can model many healthcare data types.

Data	Tokens
Clinical notes	words or subwords
Patient history	visits, diagnoses, procedures, medications
ECG or signals	time windows or patches
Medical images	image patches
Multimodal records	text, codes, signals, images

The common structure is:

$$\text{objects} \rightarrow \text{embeddings} \rightarrow \text{contextual representations} \rightarrow \text{prediction}$$

Biology Examples

Transformers are also useful for biological sequences.

- DNA sequence modeling
- protein sequence modeling
- variant effect prediction
- gene expression modeling
- single-cell representation learning

A protein sequence, for example, can be treated as a sequence of amino acid tokens:

$$[a_1, a_2, \ldots, a_T]$$

Self-attention allows distant residues to influence each other.

Attention Models in Healthcare

The book presents attention models for four healthcare tasks:

Case study	Data	Main task
RETAIN	longitudinal EHR	heart failure prediction
GRAM	medical ontology + EHR codes	heart failure prediction
CAML	clinical notes	ICD code assignment
MINA	ECG signals	heart disease detection

The common idea is: use attention to decide which visits, codes, words, or signal segments are most relevant.

RETAIN: Attention over Longitudinal EHR

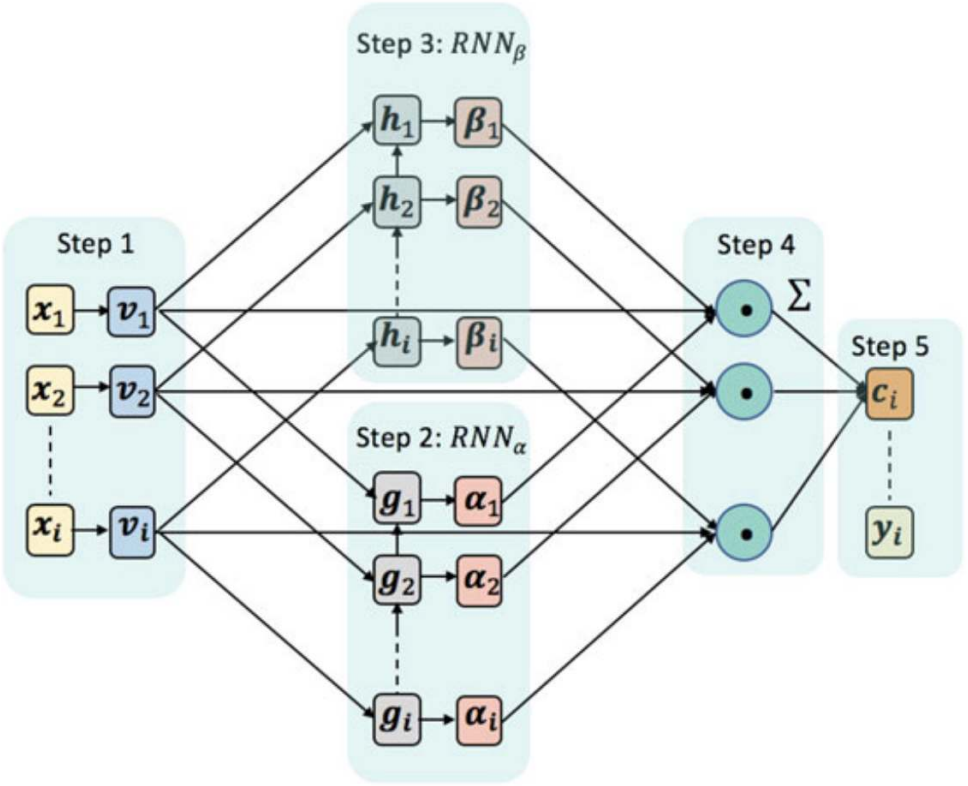
Problem: predict future heart failure from a patient's EHR history.

Input data:

- sequence of patient visits
- each visit represented by medical codes
- diagnosis, medication, and procedure groups

Method:

- embed each visit
- compute attention over previous visits
- compute attention over variables within visits
- predict future heart failure risk

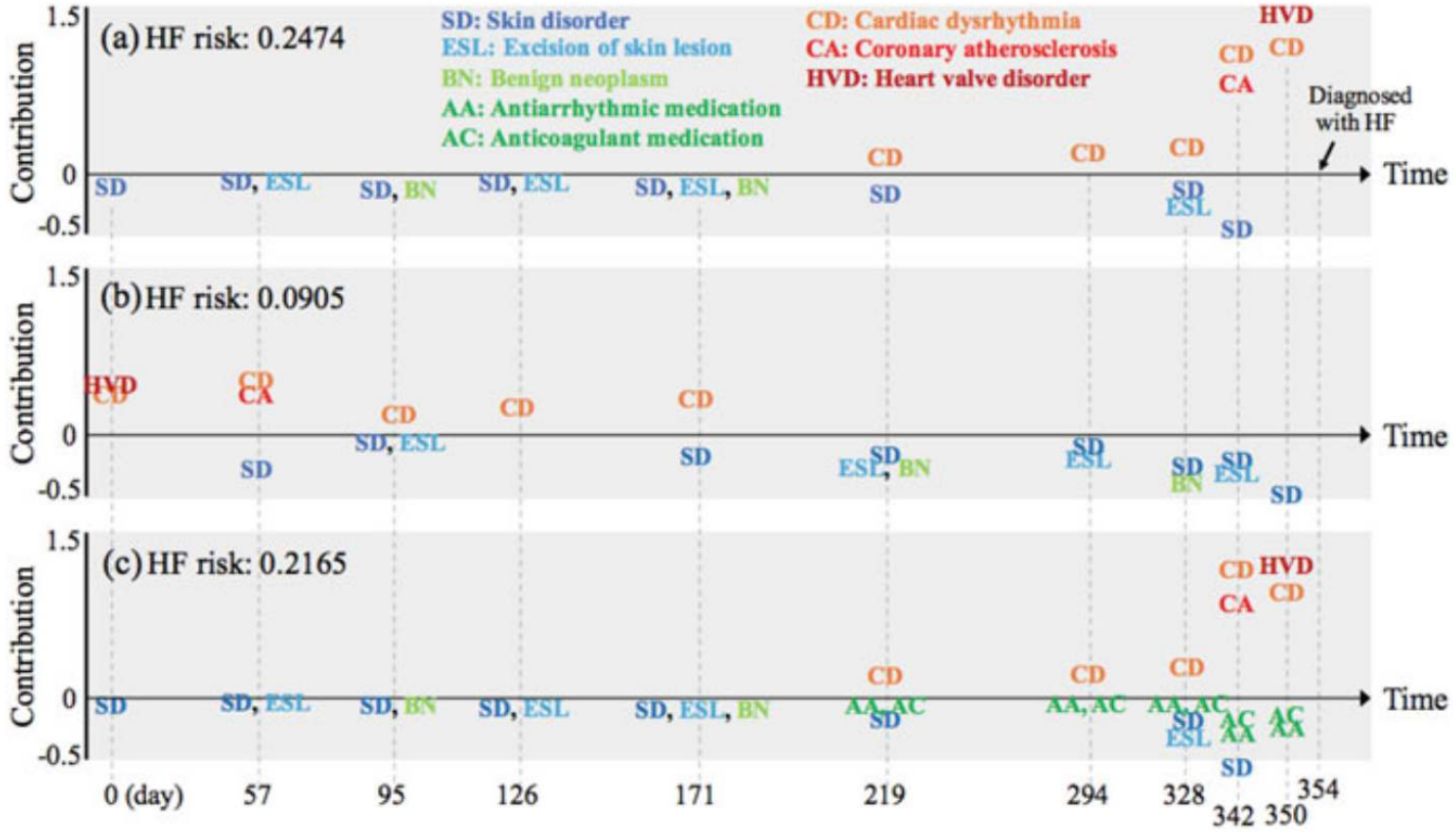


RETAIN: Why Attention Helps

RETAIN is designed not only for prediction, but also for interpretation.

The model can highlight:

- which previous visits influenced the prediction
- which medical codes contributed within those visits
- whether recent cardiac-related codes matter more than older unrelated events



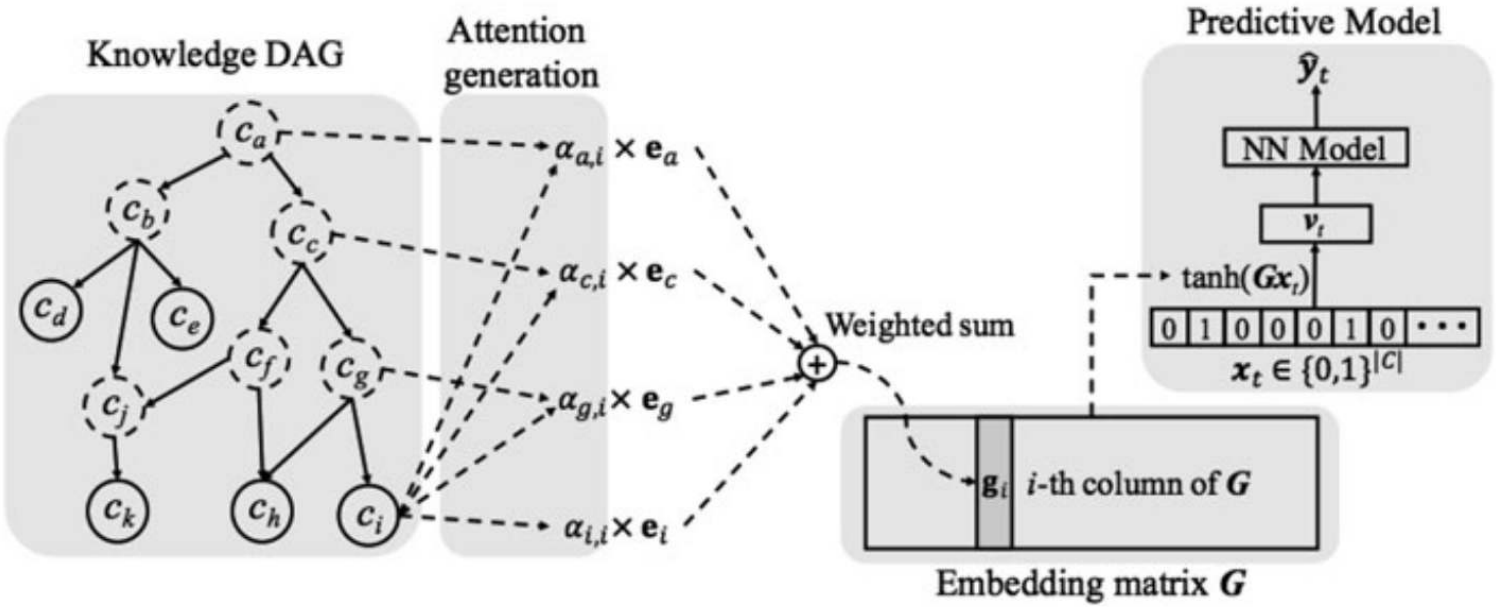
GRAM: Attention over Medical Ontologies

Problem: medical codes are sparse and hierarchical. Example:

- a specific diagnosis code may be rare
- its parent category may have more examples
- ontology ancestors can provide useful generalization

Method:

- represent each medical code
- also represent its ancestors in the ontology
- use attention to combine the code and ancestor embeddings

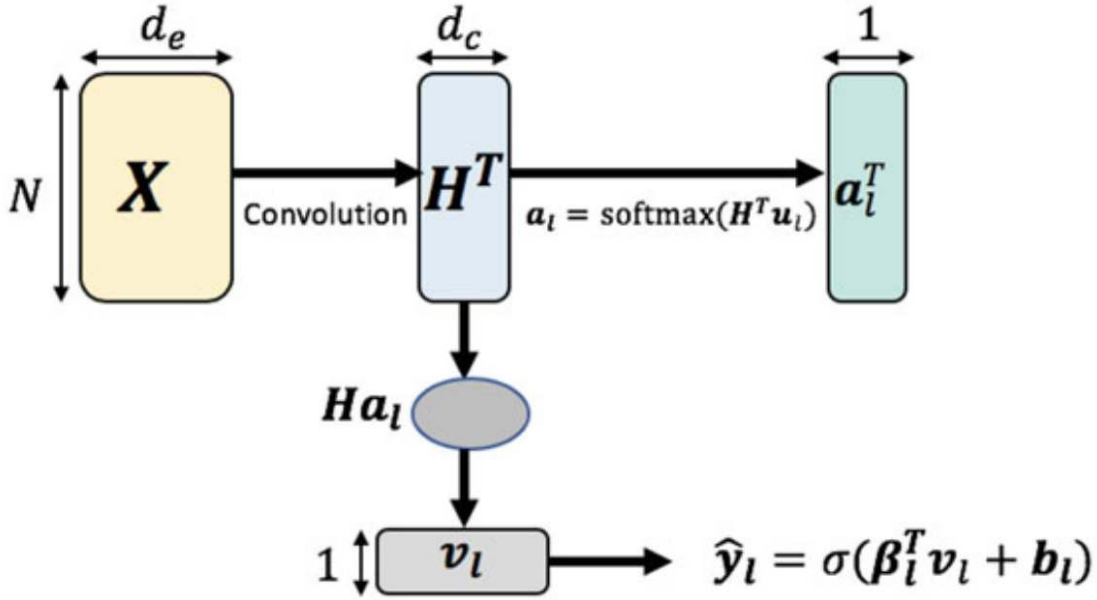


A medical ontology is a structured hierarchy or graph of medical concepts and their relationships, such as diagnosis codes, procedures, medications, and broader disease categories.

CAML: ICD Classification from Clinical Notes

Problem: assign International Classification of Diseases (ICD) codes from clinical notes.

- Clinical notes are long text documents written during patient encounters.
- ICD codes are usually assigned manually by human coders, which is labor-intensive and error-prone.



Method:

- use a convolutional layer over the note
- use attention for each ICD label
- predict multiple ICD codes

CAML: Label-Specific Attention

CAML uses a different attention distribution for each ICD code.

This means:

- the model can look at different words for different diagnoses
- one note can support many labels
- each label attends to the text span most relevant for that label

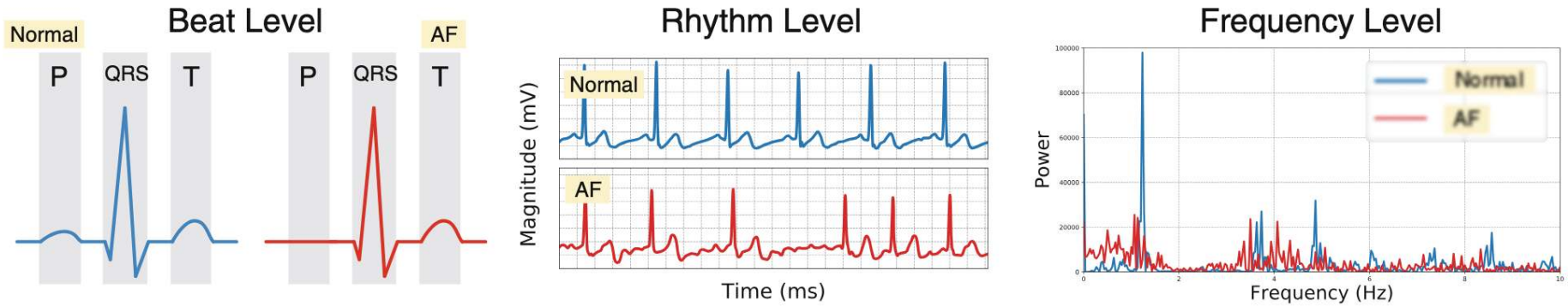
CAML is evaluated on MIMIC-III discharge summaries for ICD-9 code prediction.

MINA: Attention for ECG Classification

Problem: classify heart disease from ECG signals.

ECG interpretation traditionally relies on expert-defined patterns such as:

- P-wave
- QRS complex
- RR interval

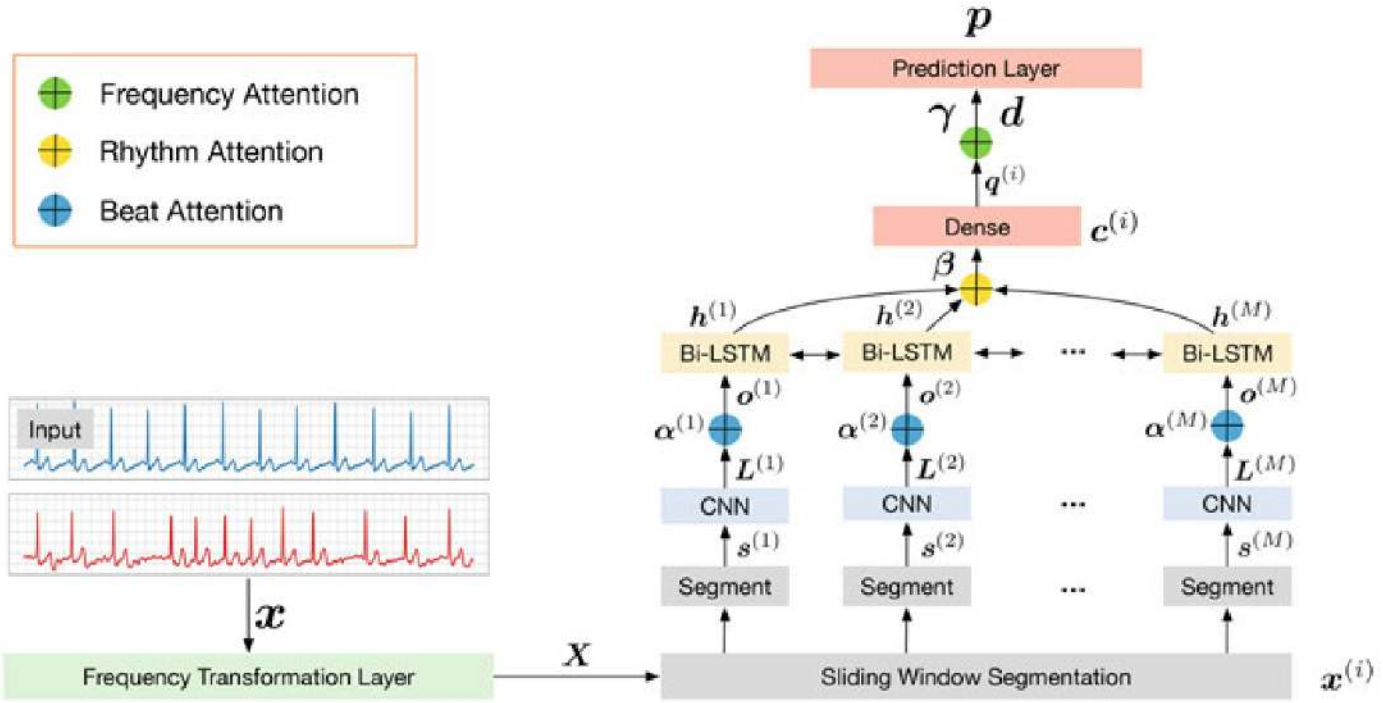


Method:

- use deep learning on ECG signals
- use attention to focus on informative parts of the signal
- classify heart rhythm or heart disease patterns

MINA: Why Attention Helps

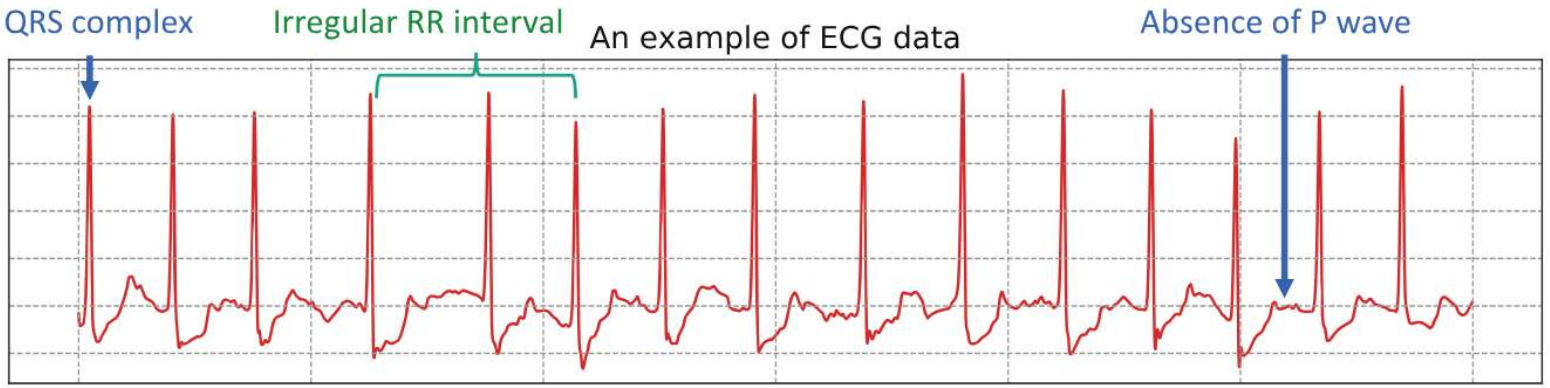
- ECG signals are long and noisy.
- Attention helps the model focus on signal regions that are more informative for classification.



This is useful because:

- not every time segment is equally diagnostic
- clinically meaningful patterns may occur only in parts of the signal
- attention can provide a partial explanation of the prediction

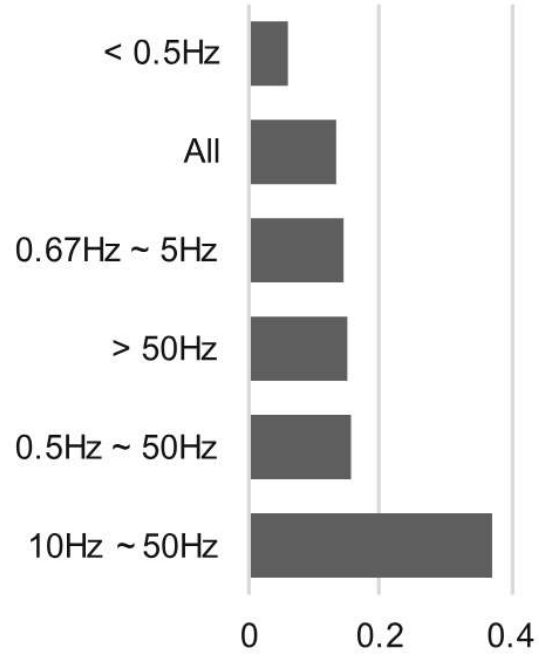
MINA: Why Attention Helps



Beat level attention points the location of QRS complex and absent P waves.

Rhythm level attention shows the location of abnormal RR interval.

Frequency level attention
shows QRS complex is dominant between 10Hz ~ 50 Hz.



References

- Introduction to Deep Learning for Healthcare, Chapter 9
- https://d2l.ai/chapter_attention-mechanisms-and-transformers/index.html
- https://jalammar.github.io/illustrated-transformer/
- https://poloclub.github.io/transformer-explainer/