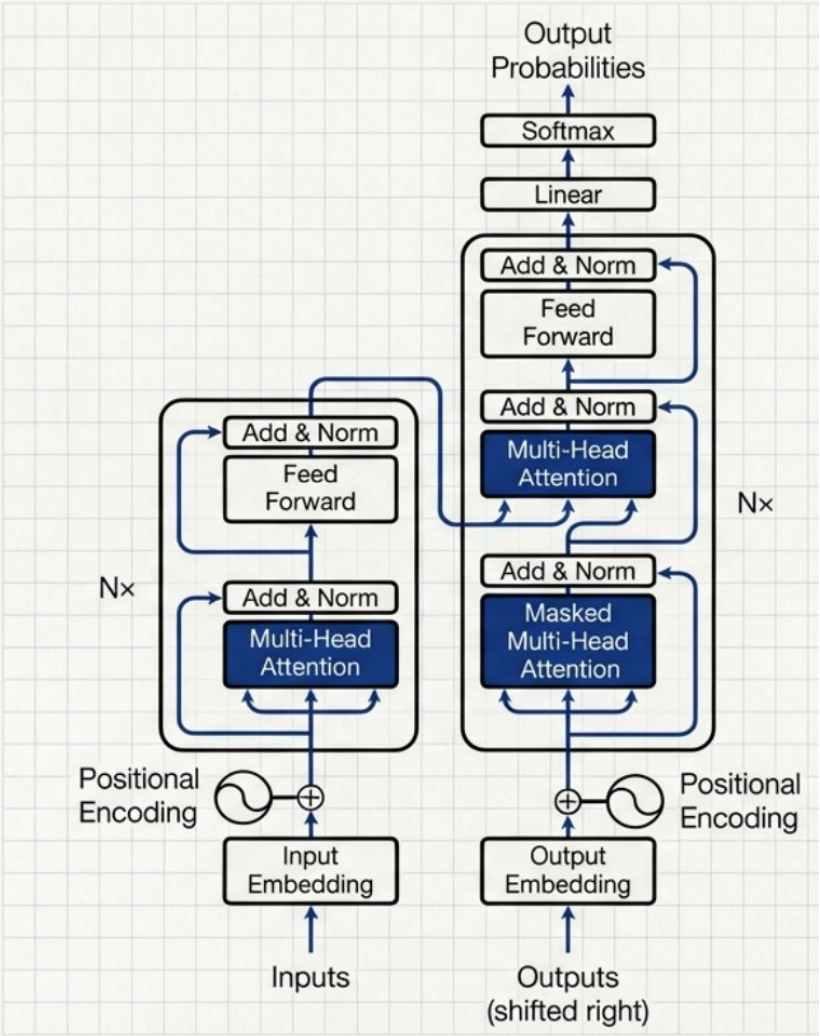
Transformers

Chapter 6

The Blueprint That Changed Everything

In 2017, the paper “Attention Is All You Need” introduced a new architecture that dispensed with recurrence and convolutions entirely. The **Transformer's** design, based solely on attention mechanisms, enabled unprecedented parallelization and set a new state-of-the-art in sequence modeling.

This diagram is our map. We will deconstruct it piece by piece, from its foundational layers to the modern innovations that have since redefined its performance.



The Problem with Sequence: Why RNNs Were a Bottleneck

The Problem
Recurrent models (RNNs, LSTMs) process data sequentially. This creates a bottleneck, limiting parallelization. Crucially, the path length for signals to travel between distant positions is long ($O(n)$), making it difficult to learn long-range dependencies.

The Solution
Self-attention connects all positions with a constant number of operations ($O(1)$), allowing signals to traverse the network instantly regardless of distance. This fundamentally solves the long-range dependency problem.

Computational Properties of Different Layer Types

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$

Teaching an Order-Blind Model About Sequence

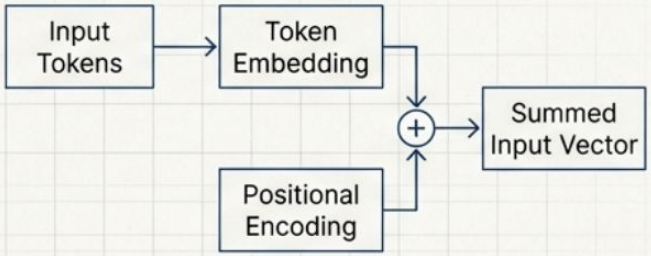
Because Transformers process all tokens simultaneously, they have no inherent sense of word order. The sentences “The cat sat on the mat” and “The mat sat on the cat” would appear identical.

Solution: Positional Encoding.

A vector representing each token’s position is added to its word embedding. This injects sequence information directly into the input.

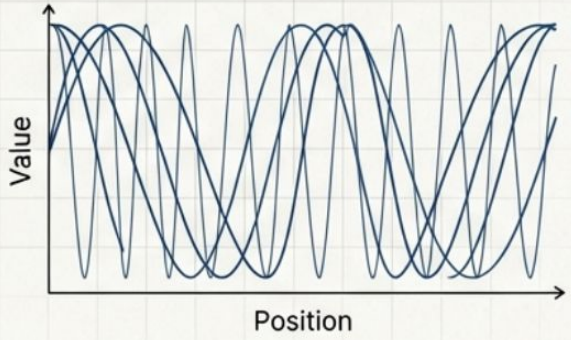
The 2017 Foundation: Sinusoidal Encoding.

The original paper used sine and cosine functions of different frequencies. This formula-based approach required no learning and could theoretically extrapolate to sequences longer than those seen during training.



$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

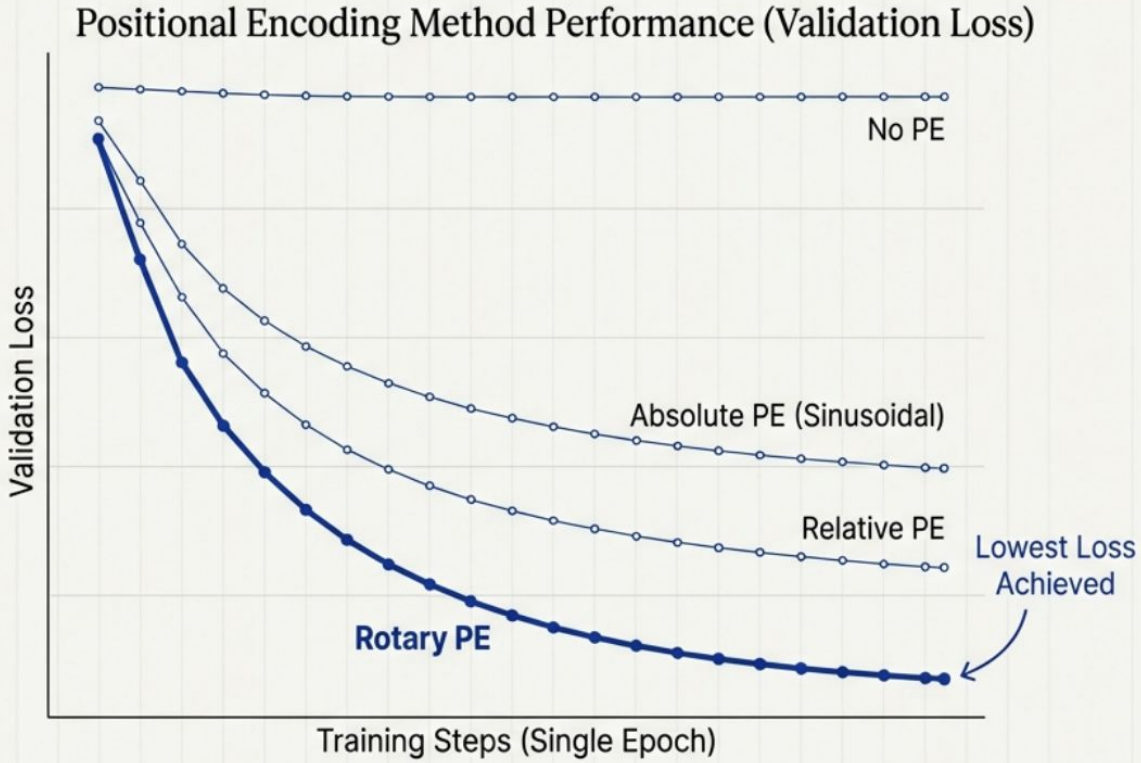


On the Frontier: The Evolution to Rotary Positional Encoding (RoPE)

While sinusoidal encoding worked, research revealed more effective methods. The key innovation was shifting from absolute position vectors to encoding relative positions within the attention mechanism itself.

Rotary Positional Encoding (RoPE) combines the best of both. Instead of adding a vector, RoPE rotates the query and key vectors based on their absolute position. This elegant method naturally preserves the relative angle (and thus, the relationship) between tokens, regardless of where they appear in a sentence.

Key Insight
RoPE is faster to train and generalizes better, as demonstrated by lower validation loss.



The Engine Room: Self-Attention

An attention function maps a query and a set of key-value pairs to an output. Self-attention allows every token in a sequence to act as a query and 'look at' every other token (the keys and values) to calculate a new representation.

The Q, K, V Analogy

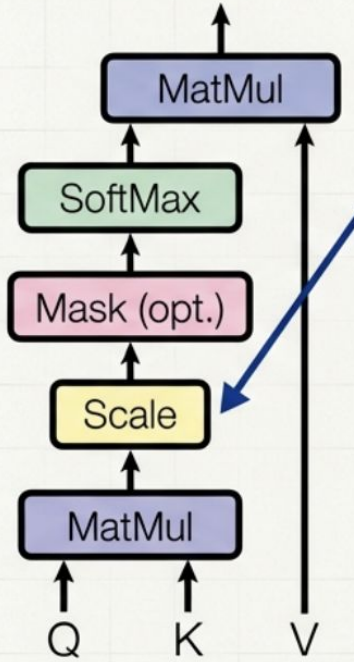
Query (Q)
My current focus. "What am I looking for?"

Key (K)
A label for other tokens. "What information do you hold?"

Value (V)
The actual representation of other tokens. "Here is the information I have."

The model calculates a compatibility score between the Query and each Key. These scores are converted into weights (via softmax) which are then applied to the Values to produce the final output.

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



Scaling by $\sqrt{d_k}$ is crucial to prevent the dot products from growing too large and pushing the softmax function into regions with extremely small gradients.

Self-attention processes text by allowing each token in a sequence, like words in a sentence, to attend to all others simultaneously for context-aware representations. A classic example uses the sentence "Life is short, eat dessert first," where embeddings for each word are projected into query (Q), key (K), and value (V) matrices.

Token embeddings form input matrix X of shape (sequence_length, embedding_dim), e.g., 8 words × 512 dims. Compute

$\mathrm{Q}=\mathrm{XW}^{\mathrm{Q}}$, $\mathrm{K}=\mathrm{XW}^{\mathrm{K}}$, $\mathrm{V}=\mathrm{XW}^{\mathrm{V}}$ with learned weights W. Scores are $\frac{QK^T}{\sqrt{d_k}}$, softmaxed to weights, then output $=\mathrm{softmax}(\mathrm{scores})\mathrm{V}$

This yields attention weights showing "Life" strongly attends to "short" and "eat," capturing dependencies like subject-verb relations.

Practical Text Demo

For "The animal didn't cross the street because it was too tired," self-attention links "it" highly to "animal" (not "street"), resolving pronouns via contextual scores.

Why One Perspective Isn't Enough: Multi-Head Attention

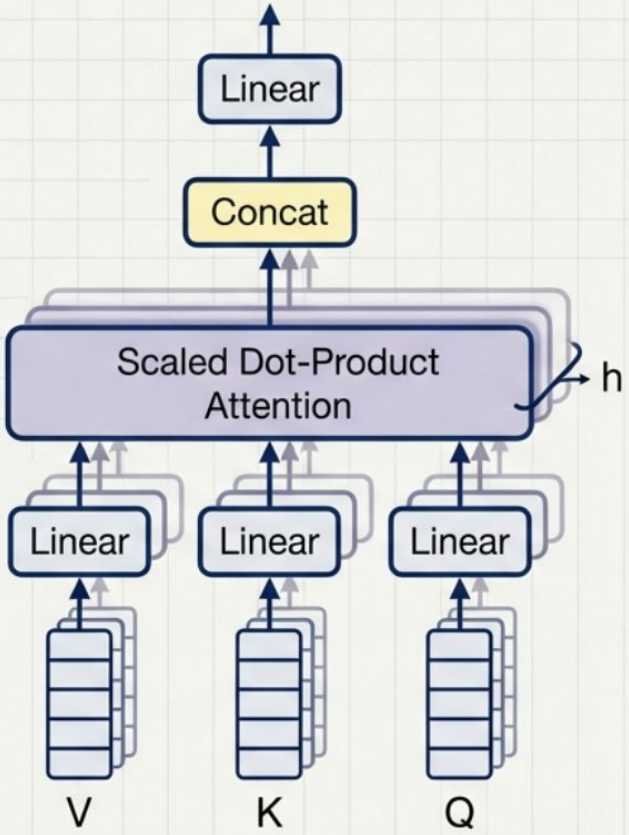
Instead of performing a single attention calculation, the Transformer projects the Queries, Keys, and Values into multiple lower-dimensional subspaces, or "heads".

Each head performs attention in parallel, allowing the model to jointly attend to information from different representational subspaces. For example, one head might learn syntactic relationships while another learns semantic ones.

The outputs of the h heads (the original paper used h=8) are then concatenated and projected back to the model's dimension, creating a richer final representation.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_o$$

where $\text{head}_i = \text{Attention}(QW_q^i, KW_k^i, VW_v^i)$



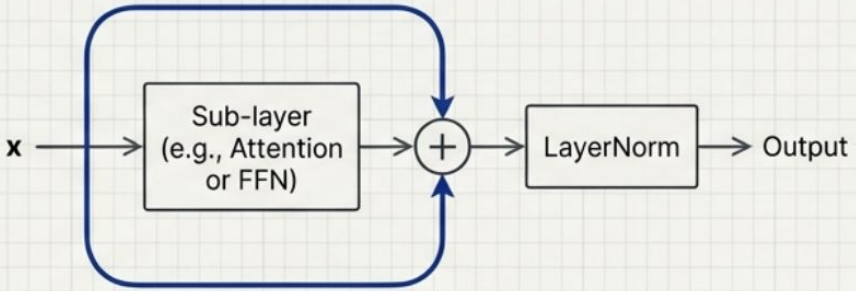
The Supporting Cast: Processing and Stabilizing the Signal

1. Position-wise Feed-Forward Network (FFN)

After attention, the output for each position is passed through a simple two-layer fully connected network. This processes the attention-infused representation and adds model capacity.

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Output = LayerNorm(**x** + Sublayer(**x**))



2. Residual Connections & Layer Normalization

Each sub-layer (Attention and FFN) is wrapped with a residual "skip" connection and a layer normalization step.

Residuals: Add the original input (x) to the sub-layer's output (Sublayer(x)). This helps prevent vanishing gradients, allowing for much deeper networks (e.g., N=6, 12, or even 24 layers).

Layer Norm: Stabilizes training by normalizing the inputs to each sub-layer.

The Encoder Stack: The Art of Understanding

The encoder's job is to map an input sequence to a sequence of continuous, contextualized representations.

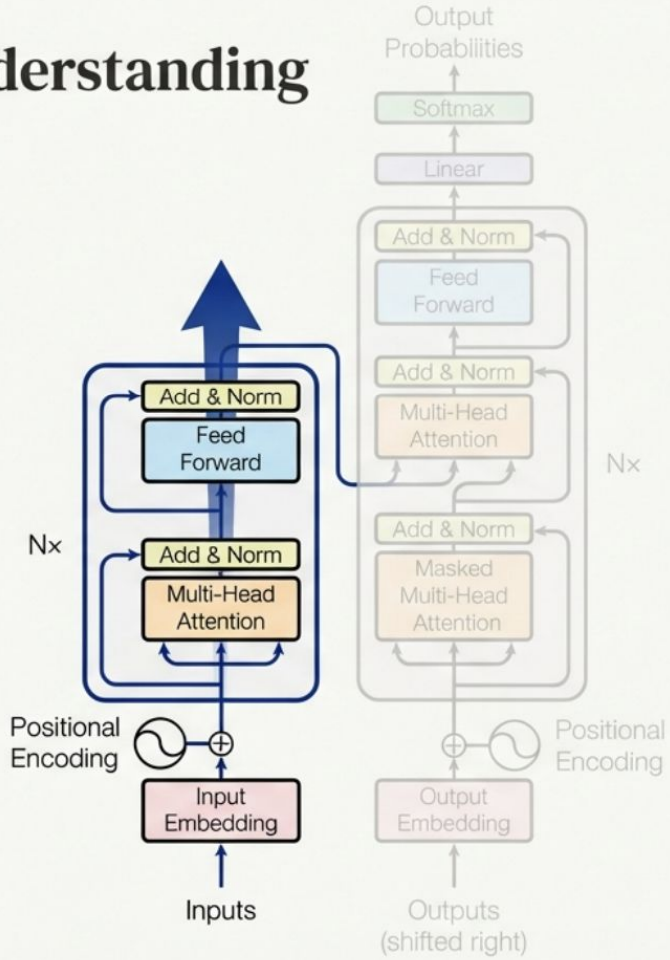
It consists of a stack of N identical layers (N=6 in the original paper).

Each layer has two sub-layers:

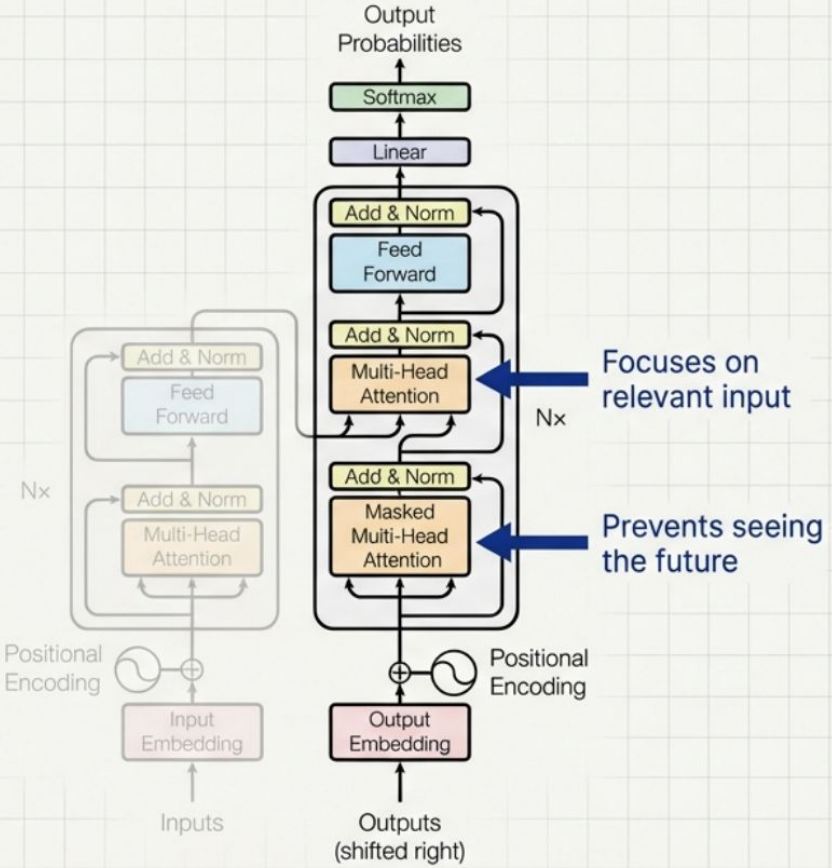
1. **Multi-Head Self-Attention**: In the encoder, attention is bidirectional. Each position can attend to all positions in the previous layer (both left and right). This allows the model to build a deep, contextual understanding of the entire input.

2. **Position-wise Feed-Forward Network**.

Both sub-layers are wrapped in residual connections and layer normalization.



The Decoder Stack: The Art of Generation



The decoder's job is to generate an output sequence one token at a time (auto-regressively).

It also consists of a stack of N identical layers.

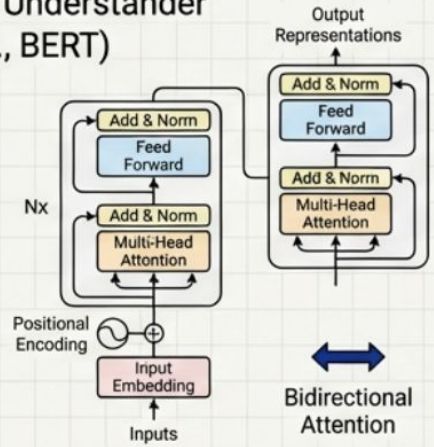
Each layer has three sub-layers:

1. **Masked Multi-Head Self-Attention**: To preserve the auto-regressive property, self-attention is masked to prevent positions from attending to subsequent ("future") positions. This ensures the prediction for position i only depends on known outputs up to i-1.
2. **Encoder-Decoder Attention**: This is the crucial link. The queries come from the decoder's previous layer, while the keys and values come from the encoder's final output. This allows the decoder to focus on the most relevant parts of the input sequence while generating the output.
3. **Position-wise Feed-Forward Network**.

The Synthesis: Assembling the Transformer Family

The most influential models in NLP are not entirely new architectures, but specific configurations of the Transformer blocks we've just explored.

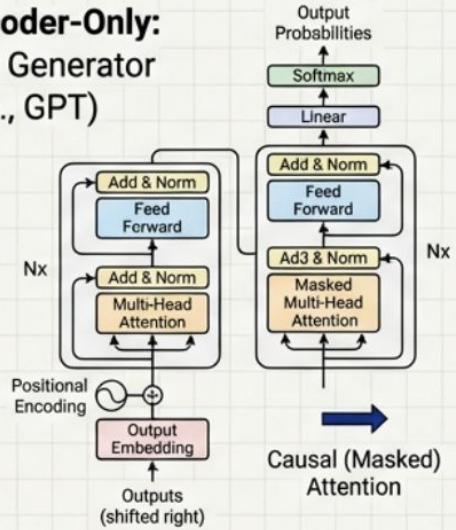
Encoder-Only:
The Understander
(e.g., BERT)



Use Cases

Natural Language Understanding (NLU), text classification, named entity recognition (NER).

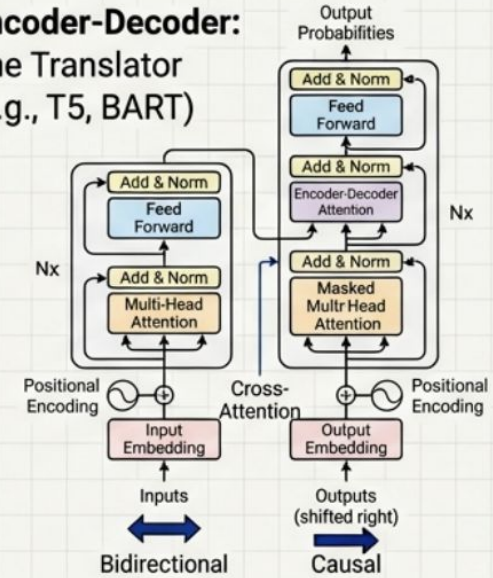
Decoder-Only:
The Generator
(e.g., GPT)



Use Cases

Text generation, Large Language Models (LLMs).

Encoder-Decoder:
The Translator
(e.g., T5, BART)



Use Cases

Translation, summarization, sequence-to-sequence tasks.

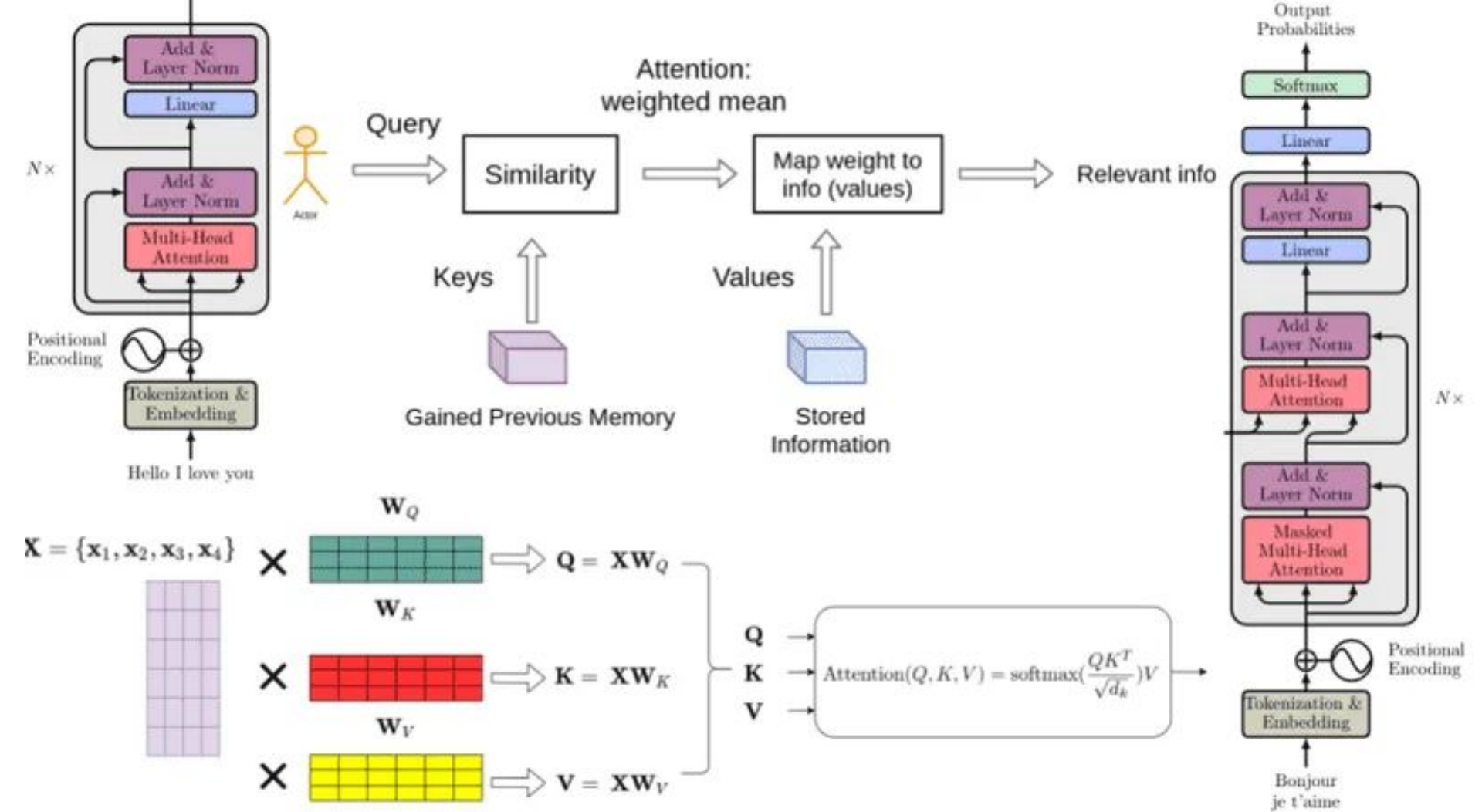
Transformer Models

Transformer models are one of the most exciting new developments in machine learning.

They were introduced in the paper Attention is All You Need.

Transformers can be used to write stories, essays, poems, answer questions, translate between languages, chat with humans, and they can even pass exams that are hard for humans! But what are they?

The architecture of transformer models is not that complex, it simply is a concatenation of some very useful components, each of which has its own function


Add & Layer Norm
Linear
Add & Layer Norm
Multi-Head Attention
N×
Positional Encoding
Tokenization & Embedding
Hello I love you
Actor
Query
Attention: weighted mean
Similarity
Map weight to info (values)
Relevant info
Keys
Values
Gained Previous Memory
Stored Information
Output Probabilities
Softmax
Linear
Add & Layer Norm
Linear
Add & Layer Norm
Multi-Head Attention
N×
Add & Layer Norm
Masked Multi-Head Attention
Positional Encoding
Tokenization & Embedding
Bonjour je t'aime
$\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$
$\mathbf{W}_Q$
$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$
$\mathbf{W}_K$
$\mathbf{K} = \mathbf{X}\mathbf{W}_K$
$\mathbf{W}_V$
$\mathbf{V} = \mathbf{X}\mathbf{W}_V$
$\mathbf{Q}$
$\mathbf{K}$
$\mathbf{V}$
$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$

Imagine that you're writing a text message on your phone.

After each word, you may get three words suggested to you.

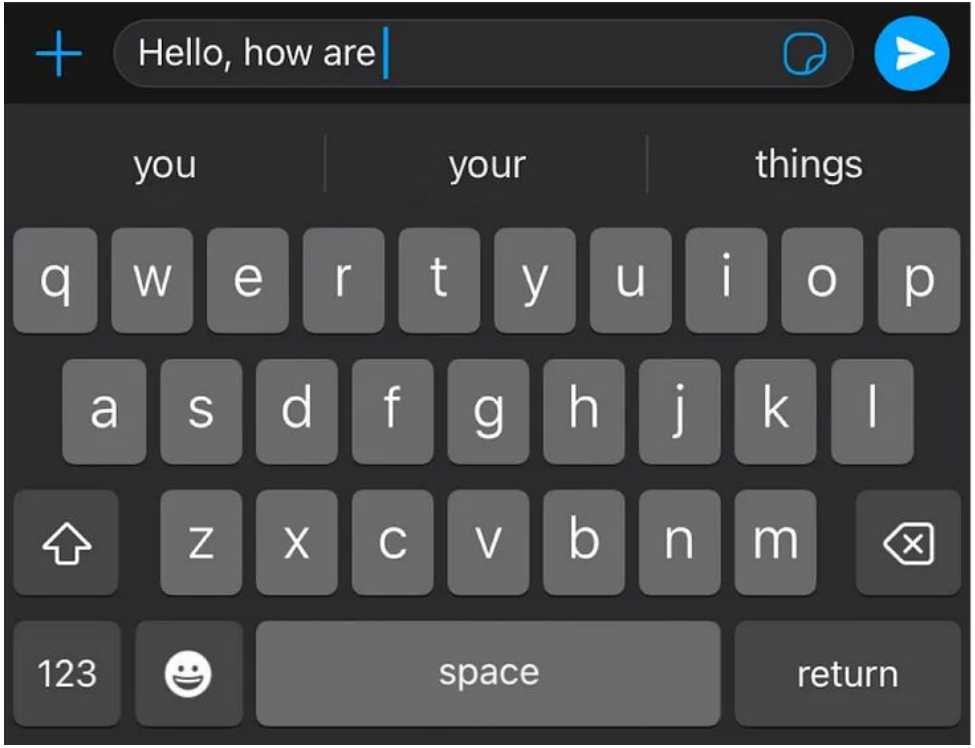
For example, if you type "Hello, how are", the phone may suggest words such as "you", or "your" as the next word.

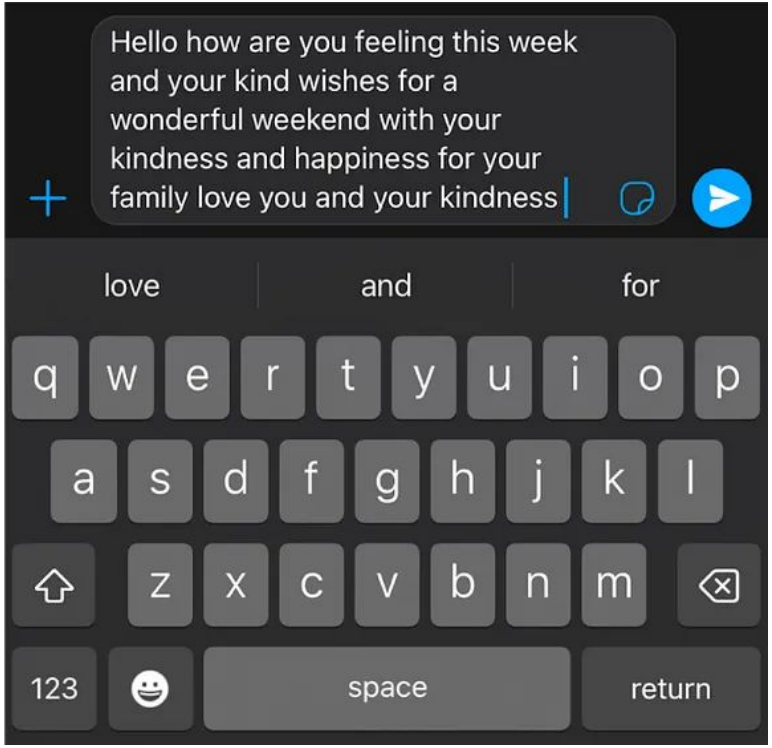
Of course, if you continue selecting the suggested word in your phone, you'll quickly find that the message formed by these words makes no sense.

If you look at each set of 3 or 4 consecutive words, it may make sense, but these words don't concatenate to anything with a meaning.

This is because the model used in the phone doesn't carry the overall context of the message, it simply predicts which word is more likely to come up after the last few.

Transformers, on the other hand, keep track of the context of what is being written, and this is why the text that they write makes sense.





The phone can suggest the next word to use in a text message, but does not have the power to generate coherent text.

Transformers build text one word at a time

This is not how humans form sentences and thoughts.

We first form a basic thought, and then start refining it and adding words to it. This is also not how ML models do other things.

For example, images are not built this way.

Most neural network based graphical models form a rough version of the image, and slowly refine it or add detail until it is perfect.

So why would a transformer model build text word by word?

One answer is, because that works really well.

A more satisfying one is that because transformers are so incredibly good at keeping track of the context, that the next word they pick is exactly what it needs to keep going with an idea.

And how are transformers trained?

With a lot of data, all the data on the internet, in fact.

So when you input the sentence “Hello, how are” into the transformer, it simply knows that, based on all the text in the internet, the best next word is “you”.

If you were to give it a more complicated command, say, “Write a story.”, it may figure out that a good next word to use is “Once”.

Then it adds this word to the command, and figures out that a good next word is “upon”, and so on.

And word by word, it will continue until it writes a story.

Command: Write a story.

Response: Once

Next command: Write a story. Once

Response: upon

Next command: Write a story. Once upon

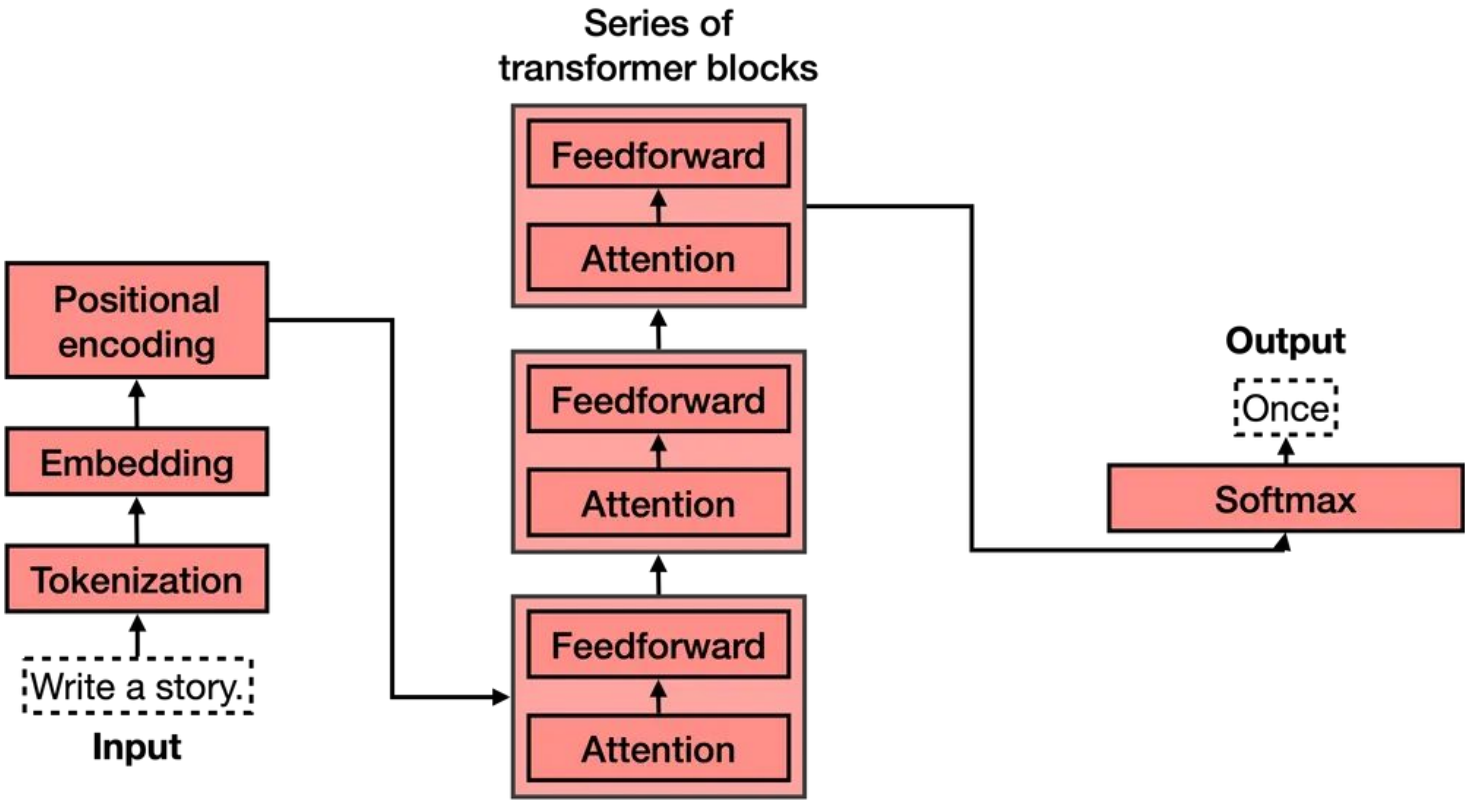
Response: a

Next command: Write a story. Once upon a

Response: time

The transformer has 4 main parts:

- Tokenization
- Embedding
- Positional encoding
- Transformer block (several of these)
- Softmax

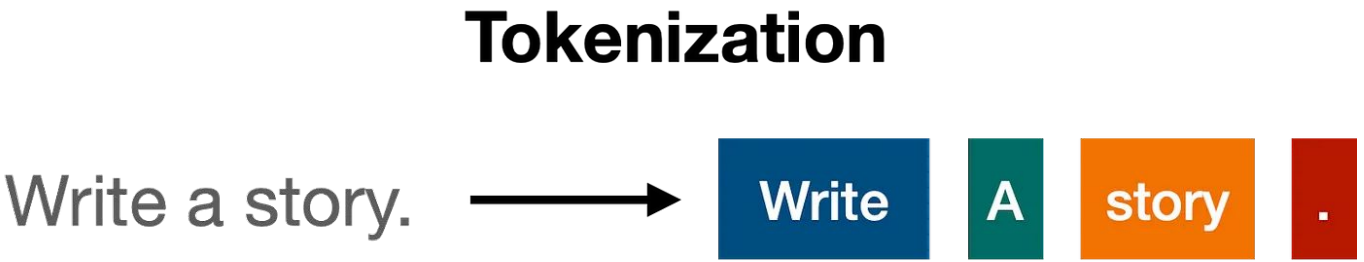


Tokenization

Tokenization is the most basic step.

It consists of a large dataset of tokens, including all the words, punctuation signs, etc.

The tokenization step takes every word, prefix, suffix, and punctuation signs, and sends them to a known token from the library.

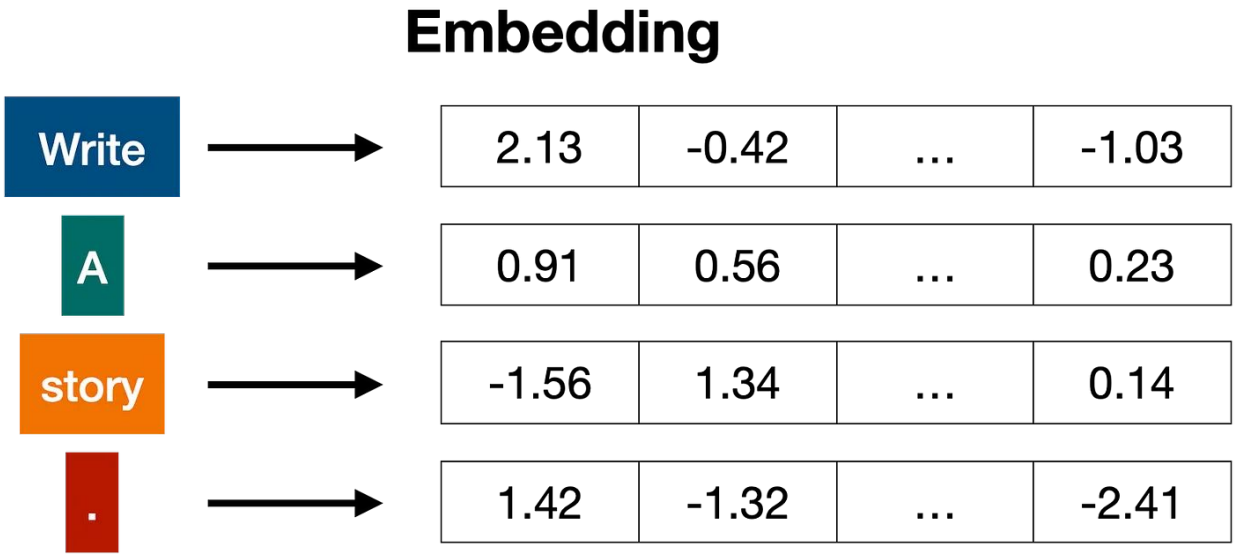


Embedding

Once the input has been tokenized, it's time to turn words into numbers. For this, we use an embedding.

If two pieces of text are similar, then the numbers in their corresponding vectors are similar to each other (componentwise, meaning each pair of numbers in the same position are similar).

Otherwise, if two pieces of text are different, then the numbers in their corresponding vectors are different.



Positional encoding

Once we have the vectors corresponding to each of the tokens in the sentence, the next step is to turn all these into one vector to process.

The most common way to turn a bunch of vectors into one vector is to add them, componentwise.

That means, we add each coordinate separately.

For example, if the vectors (of length 2) are [1,2], and [3,4], their corresponding sum is [1+3, 2+4], which equals [4, 6]. This can work, but there's a small caveat.

Addition is commutative, meaning that if you add the same numbers in a different order, you get the same result.

In that case, the sentence "I'm not sad, I'm happy" and the sentence "I'm not happy, I'm sad", will result in the same vector, given that they have the same words, except in different order.

This is not good.

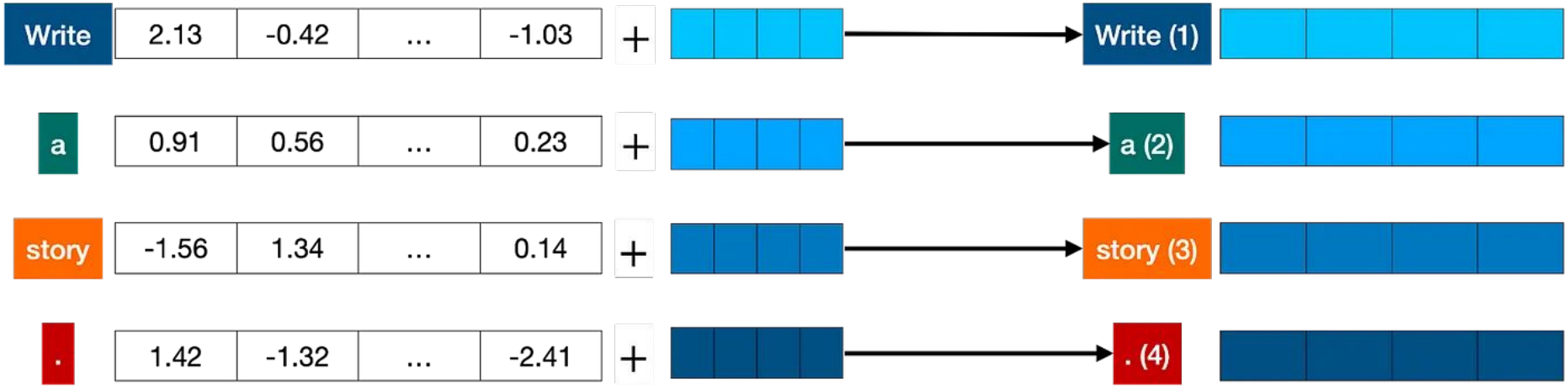
Positional encoding

Positional encoding consists of adding a sequence of predefined vectors to the embedding vectors of the words.

This ensures we get a unique vector for every sentence, and sentences with the same words in different order will be assigned different vectors.

In the example below, the vectors corresponding to the words “Write”, “a”, “story”, and “.” become the modified vectors that carry information about their position, labeled “Write (1)”, “a (2)”, “story (3)”, and “. (4)”.

Positional encoding



We can train such a large network, but we can vastly improve it by adding a key step: the attention component.

Introduced in the seminal paper Attention is All you Need, it is one of the key ingredients in transformer models, and one of the reasons they work so well

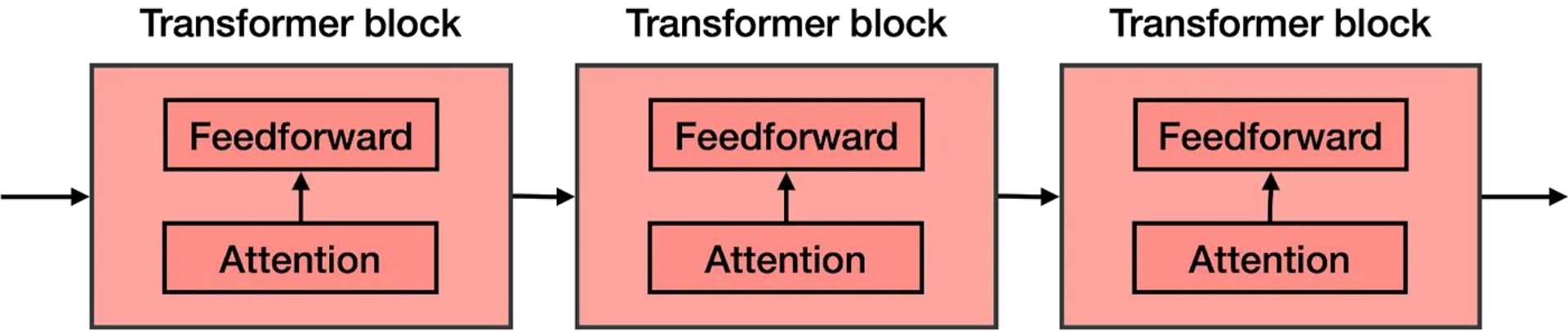
The attention component is added at every block of the feedforward network.

Therefore, if you imagine a large feedforward neural network whose goal is to predict the next word, formed by several blocks of smaller neural networks, an attention component is added to each one of these blocks.

Each component of the transformer, called a transformer block, is then formed by two main components:

- The attention component.
- The feedforward component.

Transformer



Attention

The next step is attention. Attention mechanism deals with a very important problem: the problem of context. Sometimes, as you know, the same word can be used with different meanings.

This tends to confuse language models, since an embedding simply sends words to vectors, without knowing which definition of the word they're using.

Attention is a very useful technique that helps language models understand the context. In order to understand how attention works, consider the following two sentences:

Sentence 1: The bank of the river.

Sentence 2: Money in the bank.

Attention

As you can see, the word ‘bank’ appears in both, but with different definitions. In sentence 1, we are referring to the land at the side of the river, and in the second one to the institution that holds money.

The computer has no idea of this, so we need to somehow inject that knowledge into it.

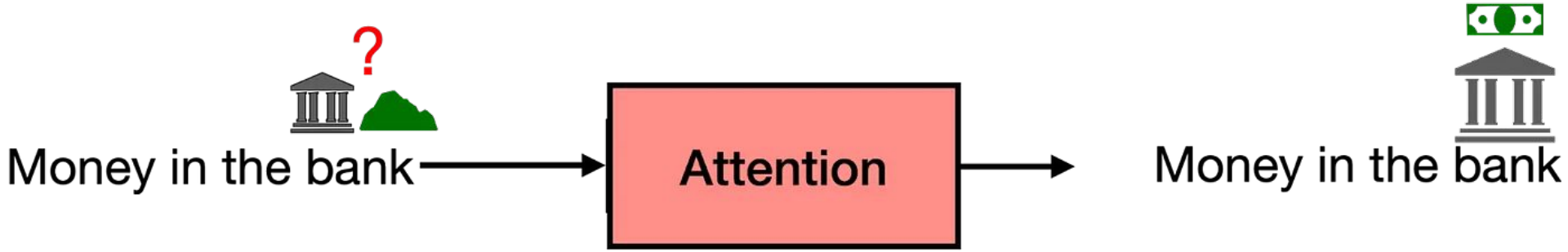
What can help us? Well, it seems that the other words in the sentence can come to our rescue.

For the first sentence, the words ‘the’, and ‘of’ do us no good.

But the word ‘river’ is the one that is letting us know that we’re talking about the land at the side of the river.

Similarly, in sentence 2, the word ‘money’ is the one that is helping us understand that the word ‘bank’ is now referring to the institution that holds money.

Attention



In short, what attention does is it moves the words in a sentence (or piece of text) closer in the word embedding.

In that way, the word “bank” in the sentence “Money in the bank” will be moved closer to the word “money”.

Equivalently, in the sentence “The bank of the river”, the word “bank” will be moved closer to the word “river”.

That way, the modified word “bank” in each of the two sentences will carry some of the information of the neighboring words, adding context to it.

Attention

The attention step used in transformer models is actually much more powerful, and it's called multi-head attention. In multi-head attention, several different embeddings are used to modify the vectors and add context to them.

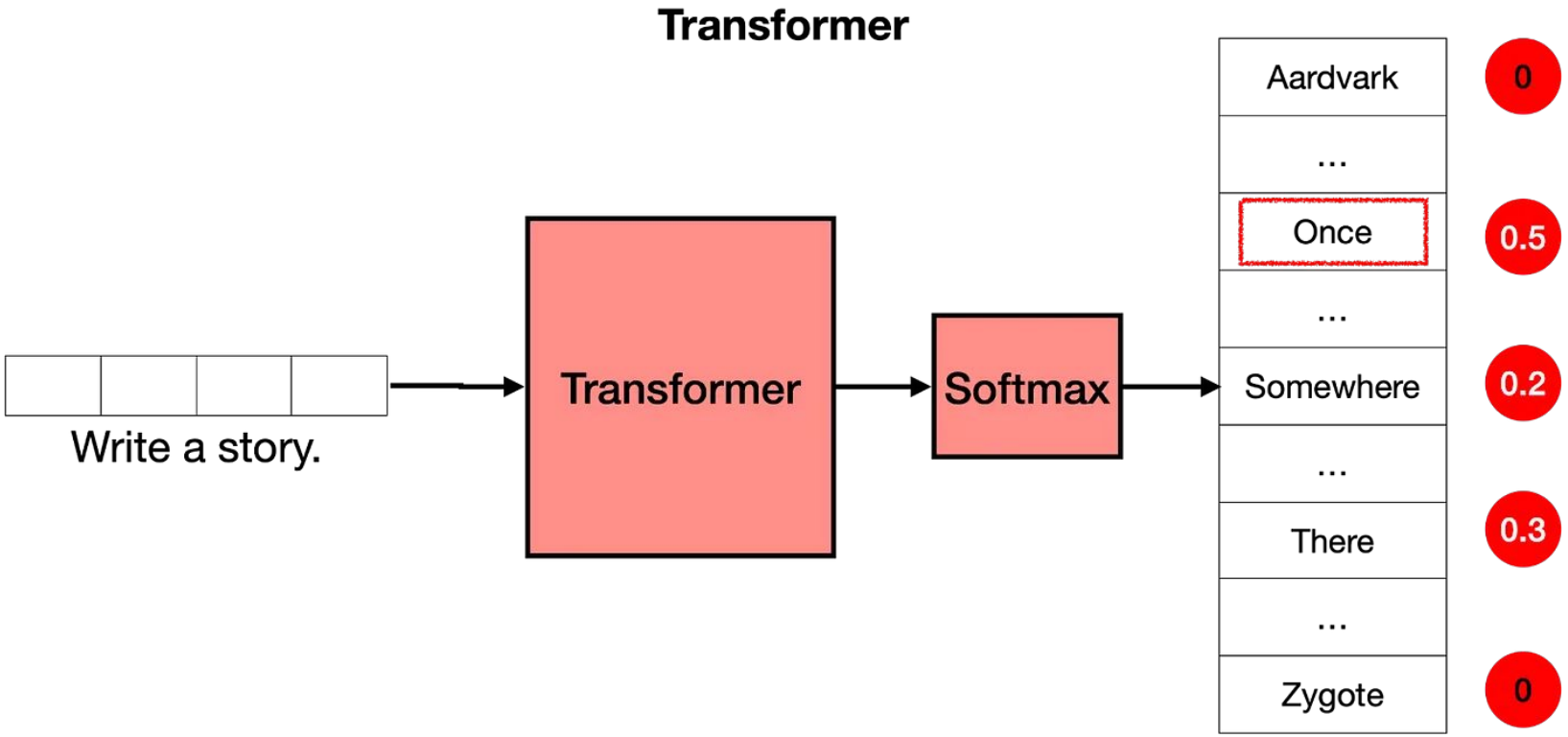
Multi-head attention has helped language models reach much higher levels of efficacy when processing and generating text.

Softmax layer

The last step of a transformer is a softmax layer, which turns these scores into probabilities (that add to 1), where the highest scores correspond to the highest probabilities.

Then, we can sample out of these probabilities for the next word. In the example below, the transformer gives the highest probability of 0.5 to “Once”, and probabilities of 0.3 and 0.2 to “Somewhere” and “There”.

Once we sample, the word “once” is selected, and that’s the output of the transformer.


Transformer
Write a story.
Transformer
Softmax
Aardvark
0
...
Once
0.5
...
Somewhere
0.2
...
There
0.3
...
Zygote
0

BERT (Bidirectional Encoder Representations from Transformer)

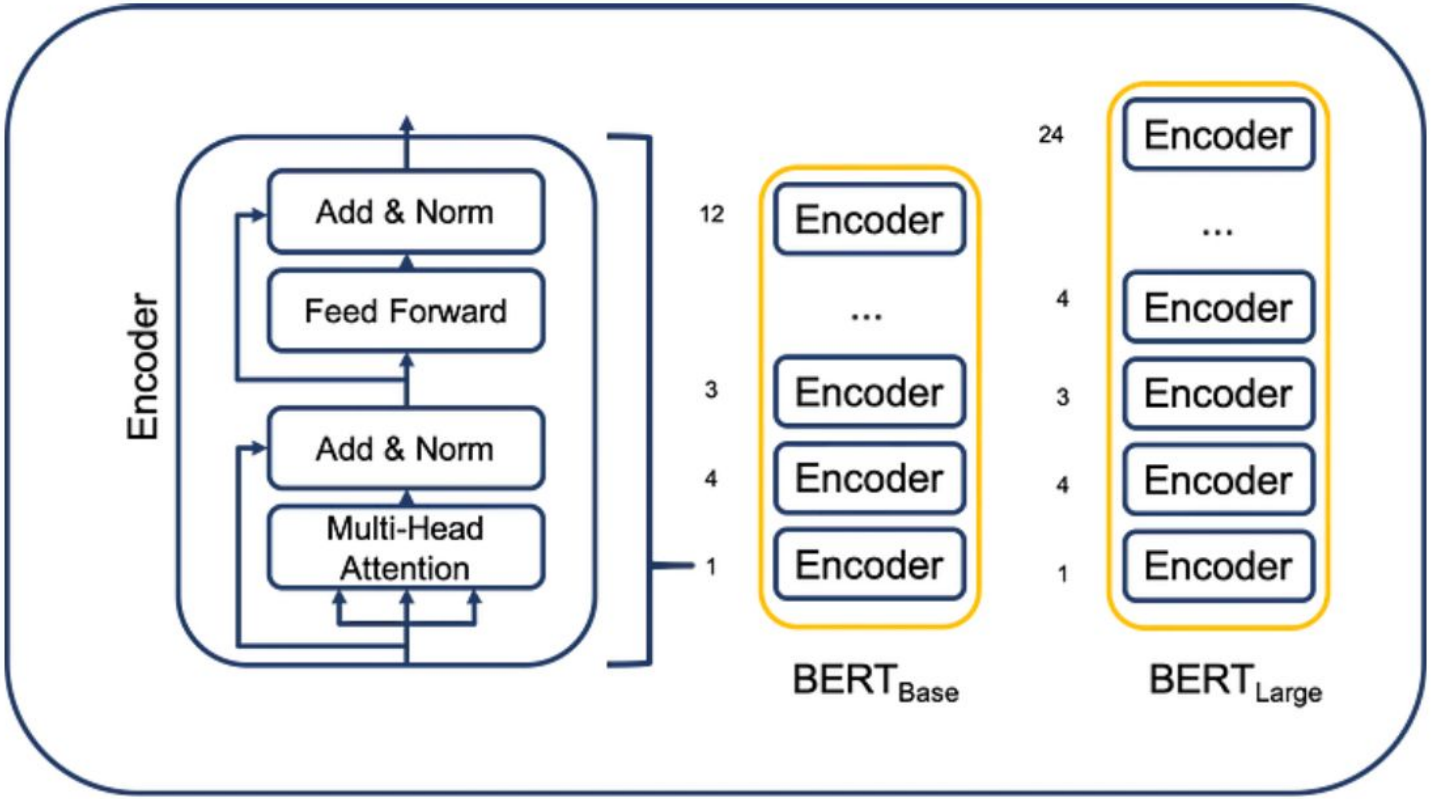
- Transformer-based language model developed by Google in 2018.

- Reads text bidirectionally - it looks at both left and right context when understanding a word

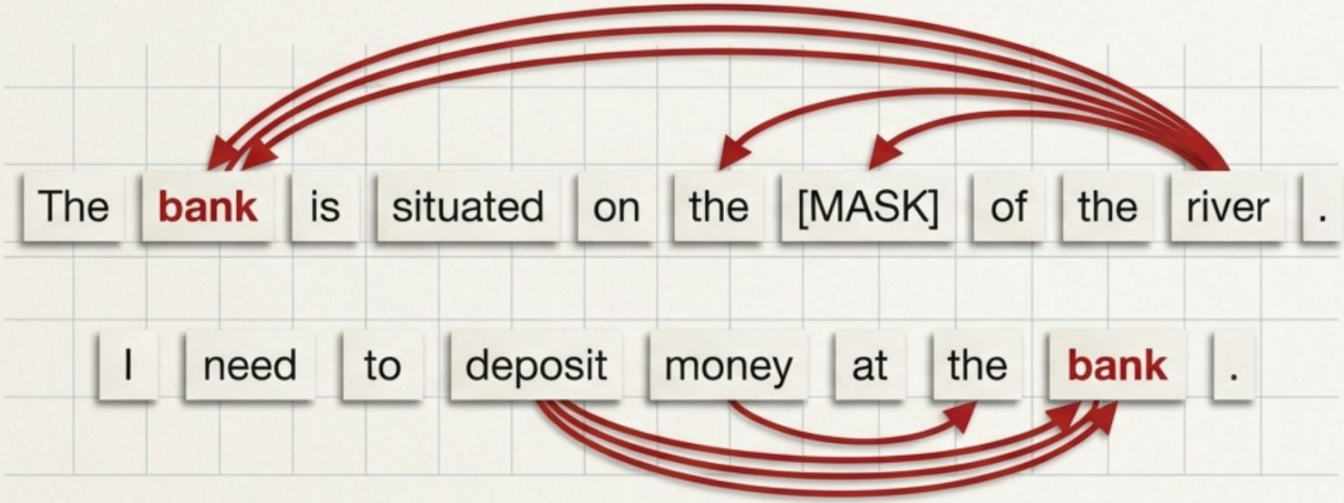
BERT (Bidirectional Encoder Representations from Transformer)

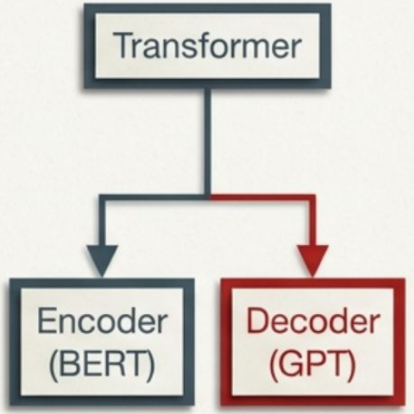
- Trained on large corpus: Wikipedia + BookCorpus (over 3.3 billion words)
  - Masked Language Modeling (MLM)
  - Next Sentence Prediction (NSP)

- BERT uses only the encoder part of the original transformer architecture, stacking multiple identical encoder layers to process input text


Encoder
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
12
3
4
1
Encoder
...
Encoder
Encoder
Encoder
BERT Base
24
4
3
4
1
Encoder
...
Encoder
Encoder
Encoder
Encoder
BERT Large

The Bi-Directional Advantage

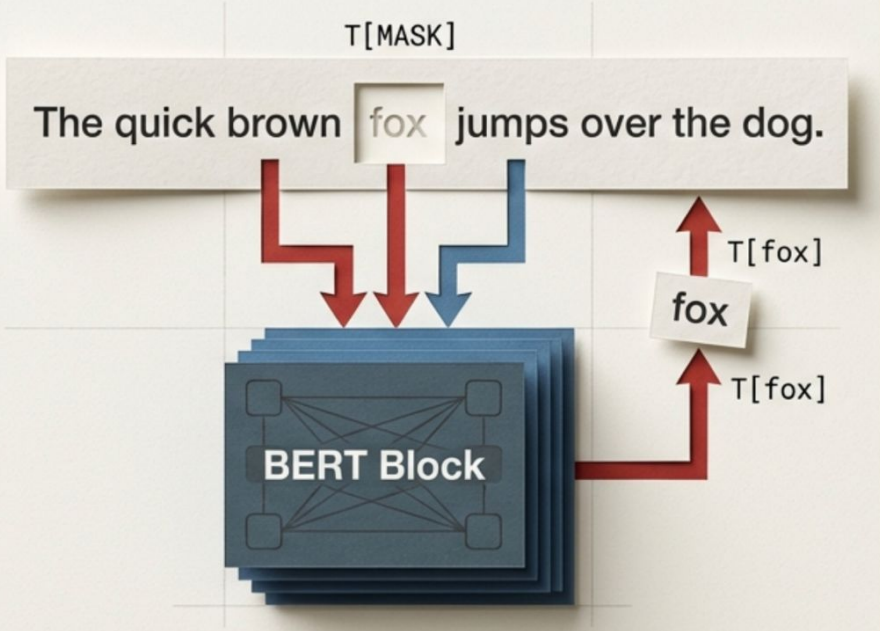




Model Specs			
BERT-Base:	12 Transformer Blocks	768 Hidden Size	110M Parameters
BERT-Large:	24 Transformer Blocks	1024 Hidden Size	340M Parameters

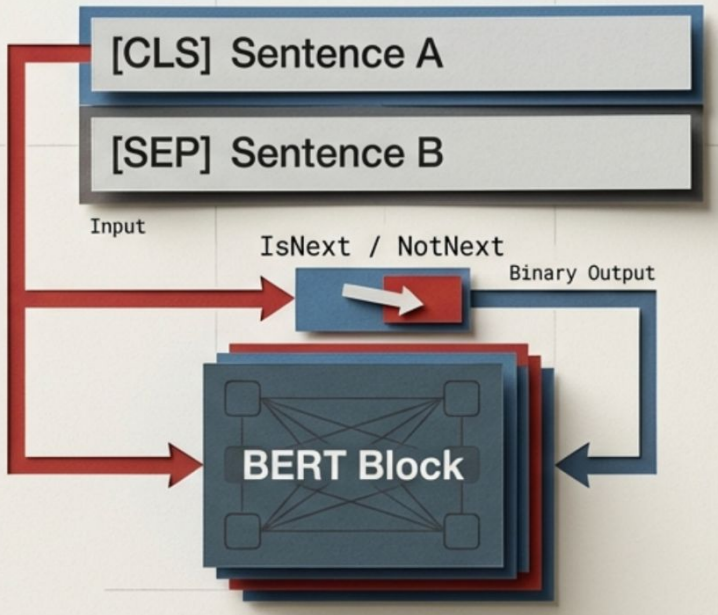
Pre-Training Objectives: The Engine of Context

Objective 1: Masked Language Model (MLM)



Randomly masks words. The model predicts the original value based *only* on context. Slower convergence than directional models, but higher context awareness.

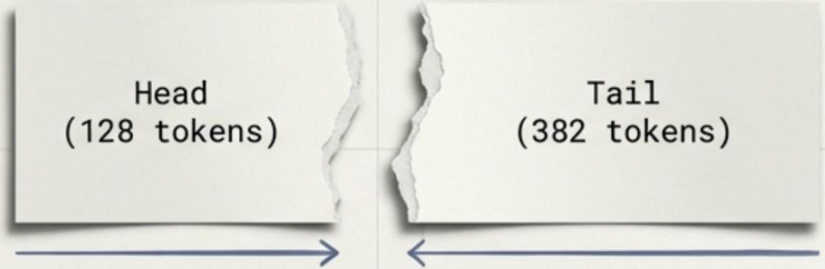
Objective 2: Next Sentence Prediction (NSP)



Predicts if Sentence B logically follows A. Critical for QA and Inference tasks.

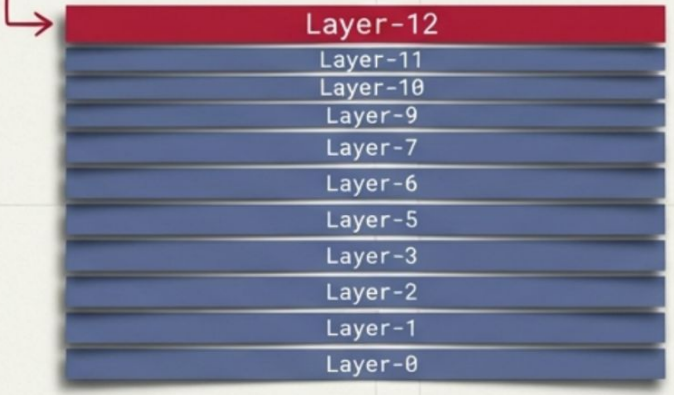
Optimization Manual: Structural Strategies

Handling Long Text: The Head+Tail Strategy



Key information usually resides at the start and end. Truncating the middle yields the lowest Error Rate: 5.42%.

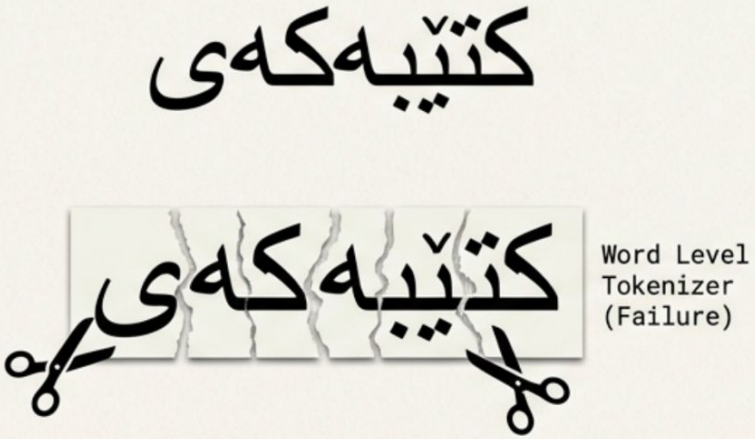
Feature Selection: The Power of Layer 12



Layer-12 contains the most effective features for classification. Using lower layers increases error rates significantly (Layer-0 Error: 11.07% vs Rate Layer-0 Error: 11.07% vs Layer-12: 5.42%).

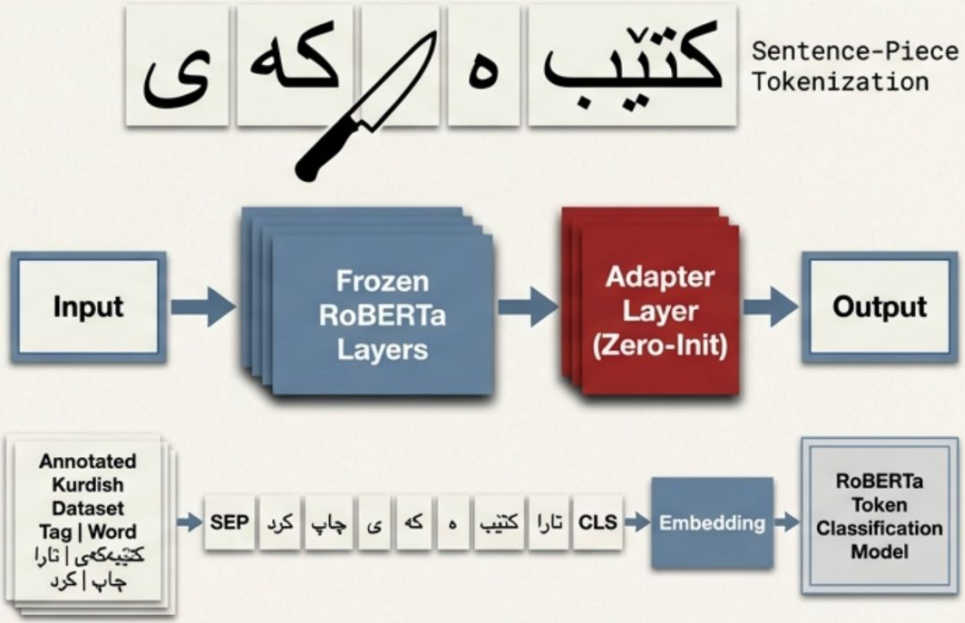
Case Study: Low-Resource NER (Kurdish)

The Challenge: Morphology



Complex morphology and scarce data cause standard models to fail.

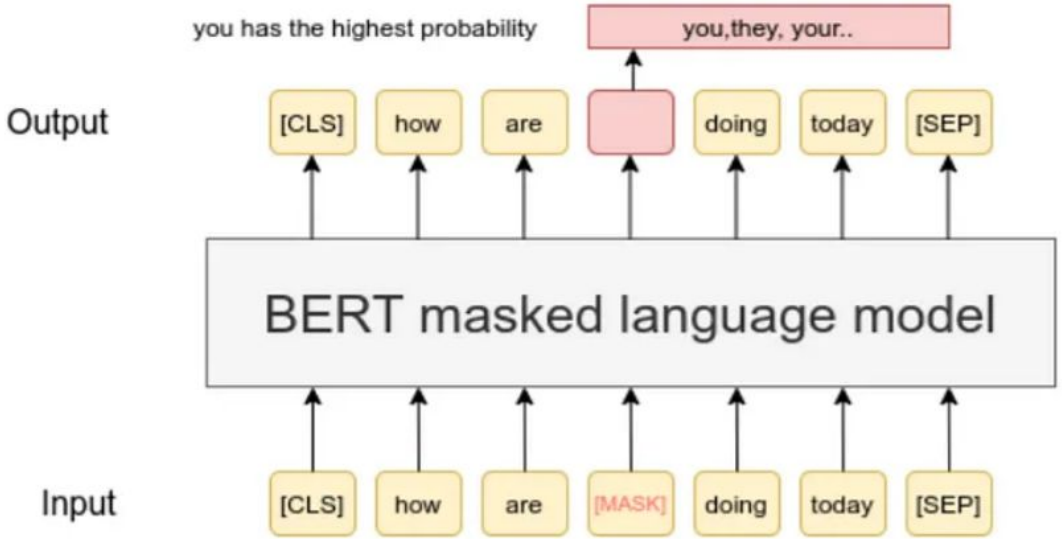
The Fix: RoBERTa + Sentence-Piece



Masked Language Modelling (MLM)

- Randomly masks 15% of input tokens
- Model predicts the masked tokens them using context from both sides
- teaches the model to deeply understand how words relate to one another
- Input: "The cat sat on the [MASK]."
- Target: "mat"

Masked Language Modelling (MLM)



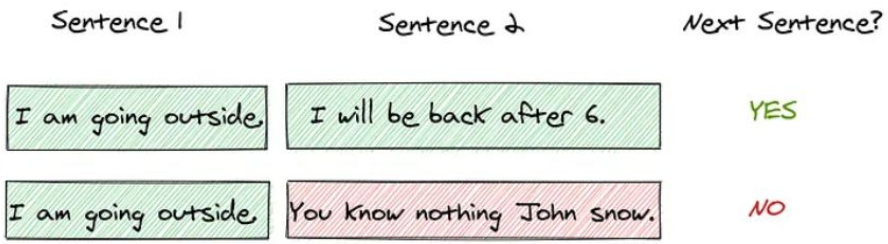
Next Sentence Prediction (NSP)

- Predicts whether sentence B follows sentence A
- helps with tasks like question answering and natural language inference

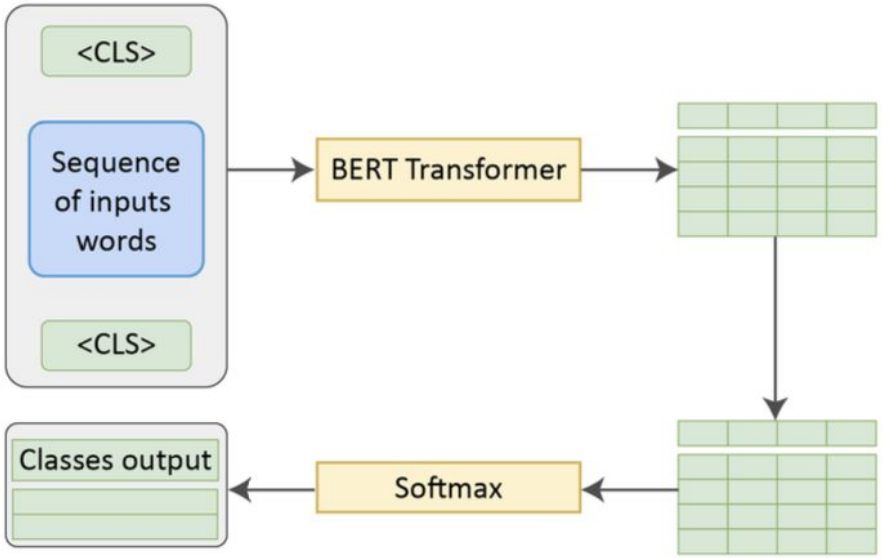
Training Pairs:

Positive: "I went to the library. I read a book"

Negative: "I went to the library. Penguins can not fly"



Architecture



- BERT-Base: 12 layers, 768 hidden units, 12 attention heads, 110M parameters.

- BERT-Large: 24 layers, 1024 hidden units, 16 attention heads, 340M parameters.

BERT Embeddings

In BERT, every input sentence or text is transformed into a series of tokens

an entire sentence is about to be processed and interpreted, it uses two special tokens:

CLS at the start, signaling the beginning and “summary” position of the sentence.

SEP to indicate the end of a single sentence or to separate two sentences in a pair.

Example: “The sky is clear and blue.”

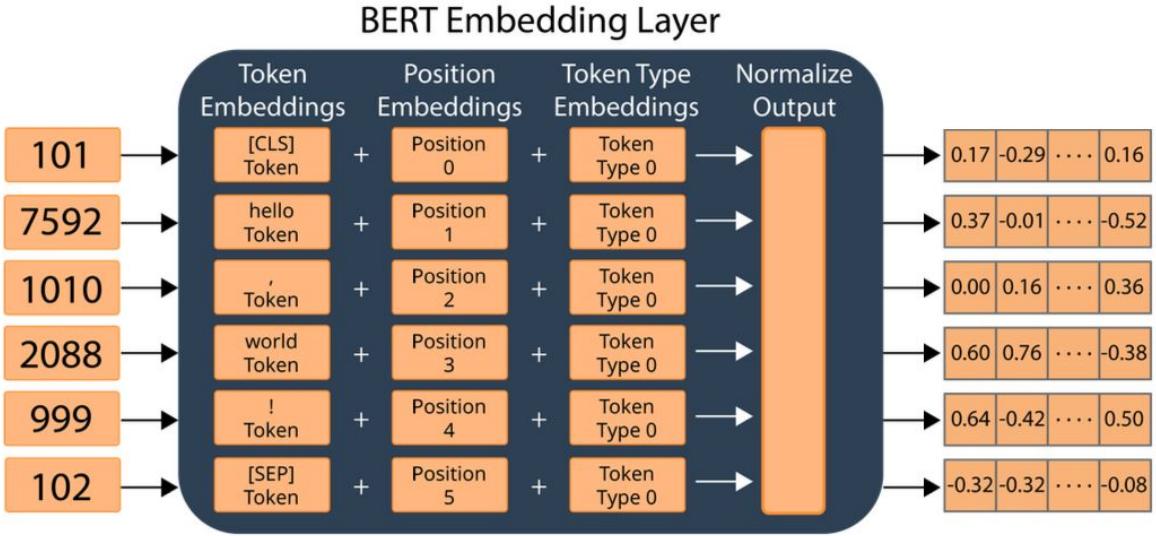
Before processing, BERT adds [CLS] and [SEP] to the sentence:

CLS The sky is clear and blue. [SEP]

BERT Embeddings

To represent textual input data, BERT relies on 3 distinct types of embeddings:

Token Embeddings, Position Embeddings, and Token Type Embeddings



Token Embeddings

used to convert the input from a string into a list of integer Token IDs

where each ID directly maps to a word or part of a word in the original string

Example: "hello, world!"

CLS hello, world! [SEP]

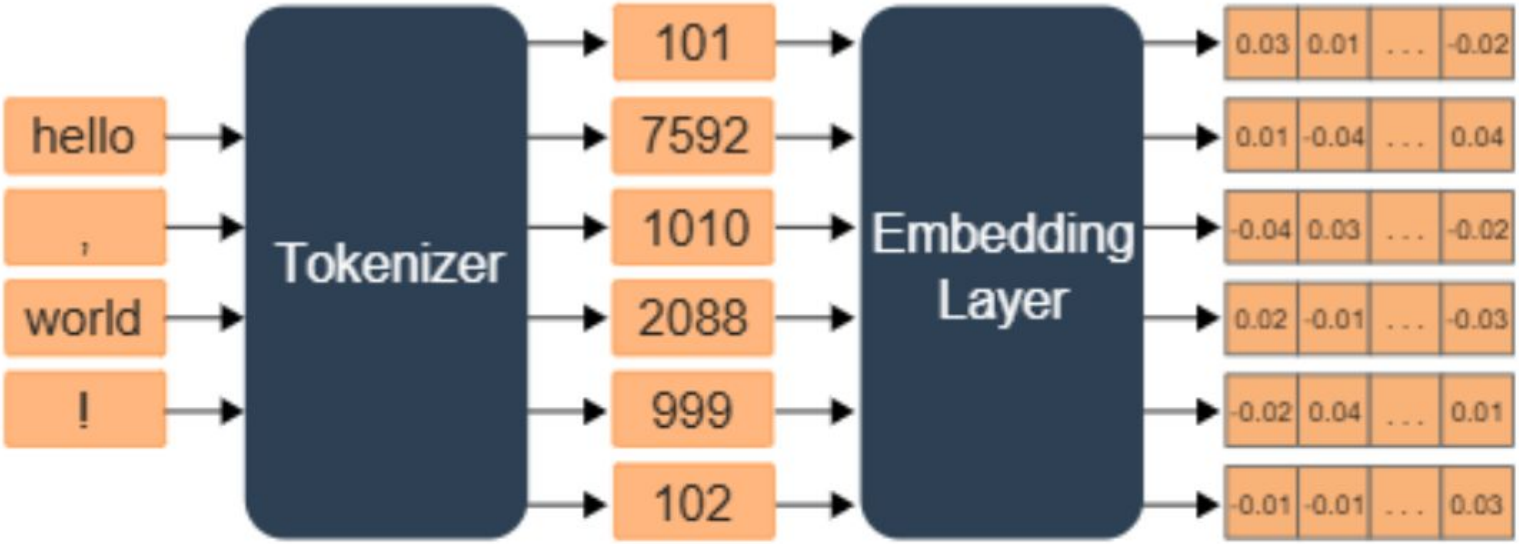
101 7592 1010 2088 999 102

For each unique Token ID (i.e. for each of the 30,522 words and subwords in the BERT Tokenizer's vocabulary), the BERT model contains an embedding that is trained to represent that specific token

The Embedding Layer within the model is responsible for mapping tokens to their corresponding embeddings

The original BERT model has a Hidden Size of 768

but other variations of BERT have been trained with smaller and larger values of the embedding (hidden) Size


hello
,
world
!
Tokenizer
101
7592
1010
2088
999
102
Embedding Layer
0.03 0.01 . . . -0.02
0.01 -0.04 . . . 0.04
-0.04 0.03 . . . -0.02
0.02 -0.01 . . . -0.03
-0.02 0.04 . . . 0.01
-0.01 -0.01 . . . 0.03

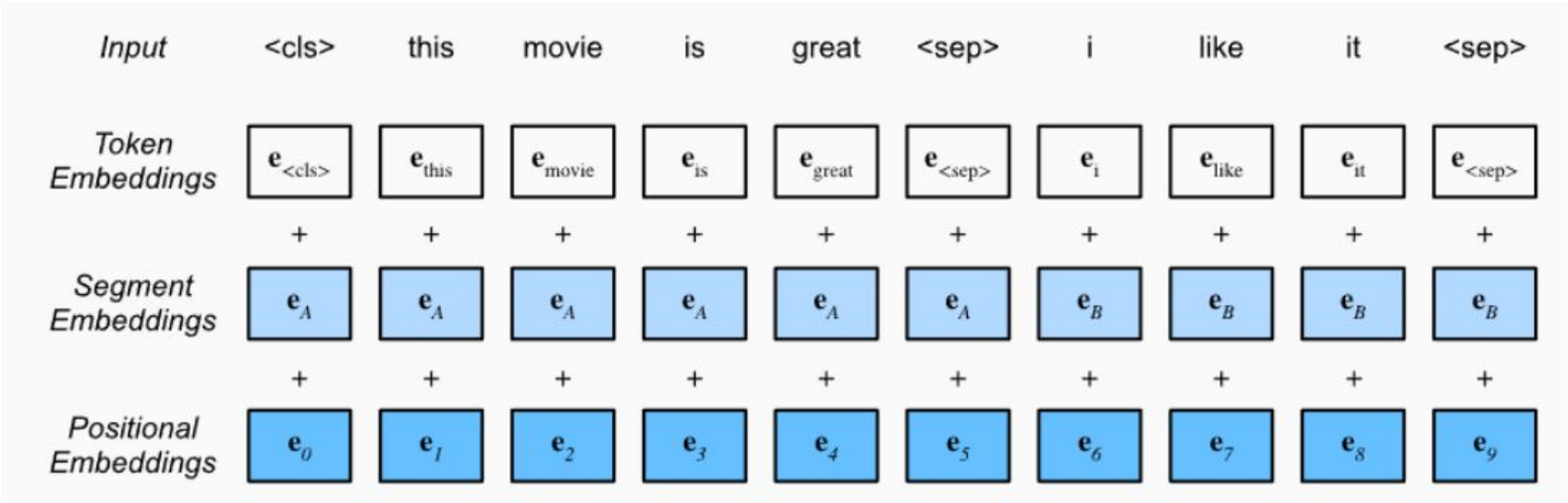
Position Embeddings

- While Token Embeddings are used to represent each possible word or subword that can be provided to the model
- Position Embeddings represent the position of each token in the input sequence
- While there are 30,522 different Token Embeddings, there are only 512 different Position Embeddings
    - This is because the largest input sequence accepted by the BERT model is 512 tokens long

Token Type Embeddings

also called the Segment Embedding in the original BERT Paper

help the model understand the structure of input sequences, especially when dealing with paired sentences.



BERT Embedding Layer

- combined embedding captures the token's identity, its segment, and its position in the sequence, providing a rich representation that the model uses for further processing
- enables BERT to capture semantic meaning, positional context, and sentence relationships within a unified vector space

  Final Embedding= LayerNorm(Token + Position + Token Type)

- where LayerNorm applies normalization to the summed vectors

BERT as a Contextual Embedding Model

- Contextual embedding is a word representation that changes depending on the surrounding words (i.e. its context)
- Unlike traditional embeddings like Word2Vec or GloVe, which assign a single fixed vector to each word regardless of usage
  - contextual embeddings adapt dynamically based on how the word is used in a sentence
- For example:

  1 “He went to the bank to withdraw money”

  - the word bank refers to a financial institution

  2 “She sat on the river bank”

  - bank refers to the edge of a river

BERT as a Contextual Embedding Model

- A fixed embedding would represent both with the same vector
  - contextual embedding model gives bank different vectors in each case
    - because it understands the context
- BERT is a contextual embedding model
  - processes sentences using a transformer-based architecture that reads the entire input bidirectionally (both left-to-right and right-to-left) to determine the meaning of each word in context
- So when a sentence through BERT, a vector for each token that reflects its meaning in that specific sentence, not some static notion of the word

Case Study (any one)

1. Sentiment Analysis (Multi-lingual analysis)
2. Fill - Masking in Nepali or other languages
3. Grammatical Error correction