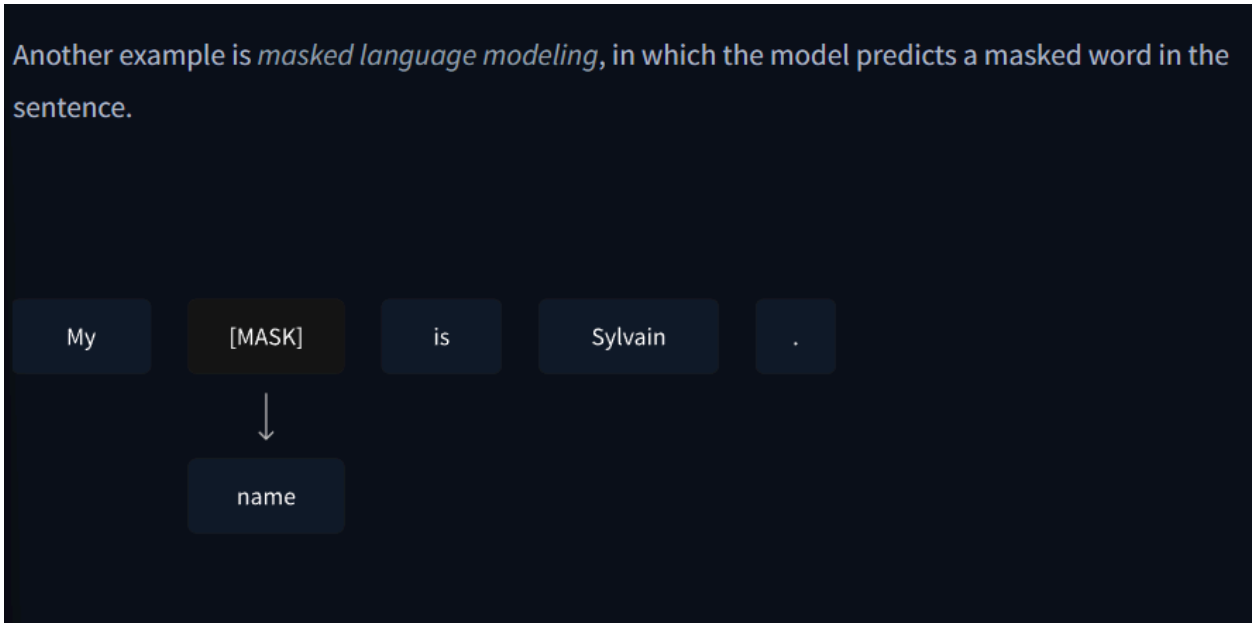
Transformers are language models

All the Transformer models have been trained as language models. This means they have been trained on large amounts of raw text in a self-supervised fashion.

Self-supervised learning is a type of training in which the objective is automatically computed from the inputs of the model. That means that humans are not needed to label the data!

This type of model develops a statistical understanding of the language it has been trained on, but it's less useful for specific practical tasks. Because of this, the general pretrained model then goes through a process called transfer learning or fine-tuning. During this process, the model is fine-tuned in a supervised way — that is, using human-annotated labels — on a given task.

An example of a task is predicting the next word in a sentence having read the n previous words. This is called causal language modeling because the output depends on the past and present inputs, but not the future ones.



Transformers are big models

Apart from a few outliers (like DistilBERT), the general strategy to achieve better performance is by increasing the models' sizes as well as the amount of data they are pre trained on.

Transformers

The transformer architecture is composed of an encoder and a decoder, each of which is made up of multiple layers of self-attention and feedforward neural networks.

Self Attention Mechanism

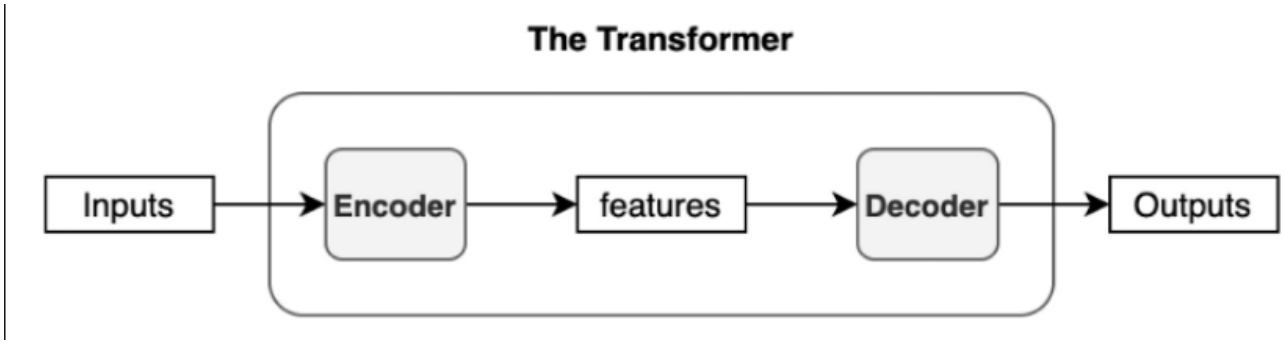
The self-attention mechanism is the heart of the transformer, allowing the model to weigh the importance of different words in a sentence based on their affinity with each other.

Positional Bias

In addition to self-attention, the transformer also introduces positional bias, which allows the model to keep track of the relative positions of words in a sentence. This is important because the order of words in a sentence can significantly impact its meaning.

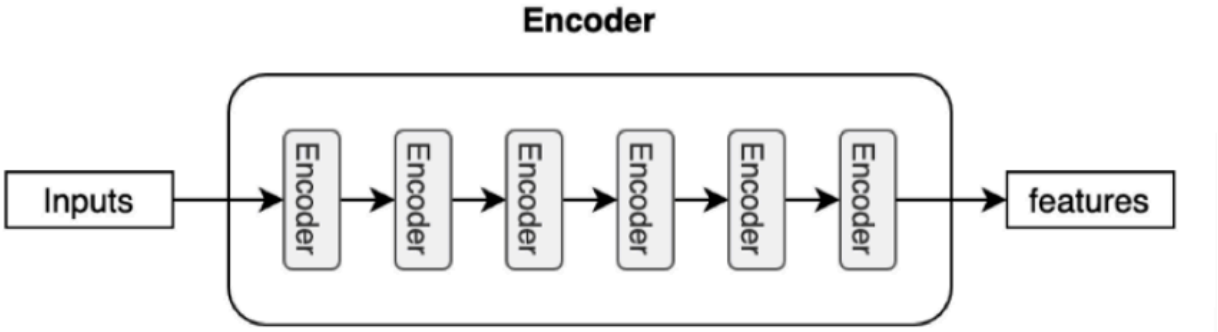
Transformer Encoder and Decoder Architecture

The transformer encoder-decoder architecture is used for tasks like language translation, where the model must take in a sentence in one language and output a sentence in another language. The encoder takes in the input sentence and produces a fixed-size vector representation of it, which is then fed into the decoder to generate the output sentence. The decoder uses both self-attention and cross-attention, where the attention mechanism is applied to the output of the encoder and the input of the decoder.



The encoder in the transformer consists of multiple encoder blocks. An input sentence goes through the encoder blocks, and the output of the last encoder block becomes the input features to the decoder.

Transformer: Encoder



The transformer encoder architecture is used for tasks like text classification, where the model must classify a piece of text into one of several predefined categories, such as sentiment analysis, topic classification, or spam detection.

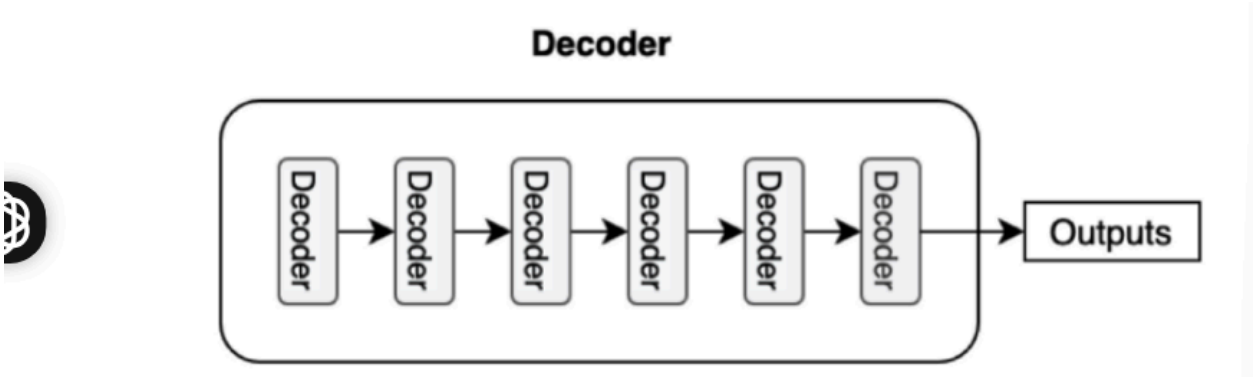
The encoder takes in a sequence of tokens and produces a fixed-size vector representation of the entire sequence, which can then be used for classification.

One of the most popular transformer encoder models is BERT (Bidirectional Encoder Representations from Transformers), which was introduced by Google in 2018. BERT is pre-trained on large amounts of text data and can be fine-tuned for a wide range of NLP tasks.

Unlike the encoder-decoder architecture, the transformer encoder is only concerned with the input sequence and does not generate any output sequence. It applies self-attention mechanism to the input tokens, allowing it to focus on the most relevant parts of the input for the given task.

Real-world examples of the transformer encoder architecture include sentiment analysis, where the model must classify a given review as positive or negative, and email spam detection, where the model must classify a given email as spam or not spam.

Transformer: Decoder

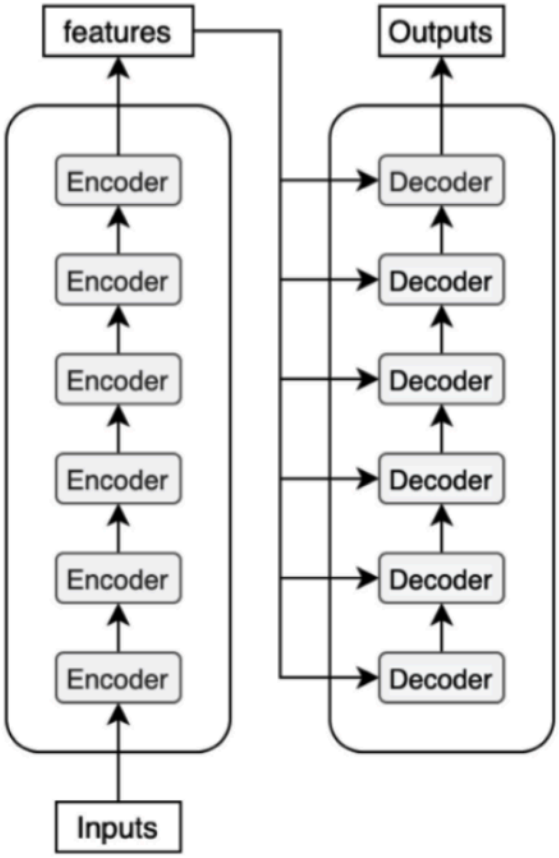


The transformer decoder architecture is used for tasks like language generation, where the model must generate a sequence of words based on an input prompt or context. The decoder takes in a fixed-size vector representation of the context and uses it to generate a sequence of words one at a time, with each word being conditioned on the previously generated words.

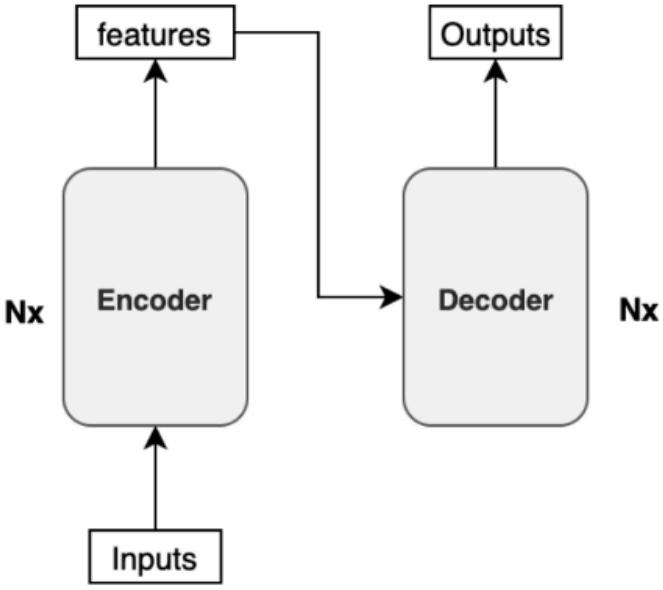
One of the most popular transformer decoder models is the GPT-3 (Generative Pre-trained Transformer 3), which was introduced by OpenAI in 2020. The GPT-3 is a massive language model that can generate human-like text in a wide range of styles and genres.

The transformer decoder architecture introduces a technique called triangle masking for attention, which ensures that the attention mechanism only looks at tokens to the left of the current token being generated. This prevents the model from "cheating" by looking at tokens that it hasn't generated yet.

Real-world examples of the transformer decoder architecture include text generation, where the model must generate a story or article based on a given prompt or topic, and chatbots, where the model must generate responses to user inputs in a natural and engaging way.



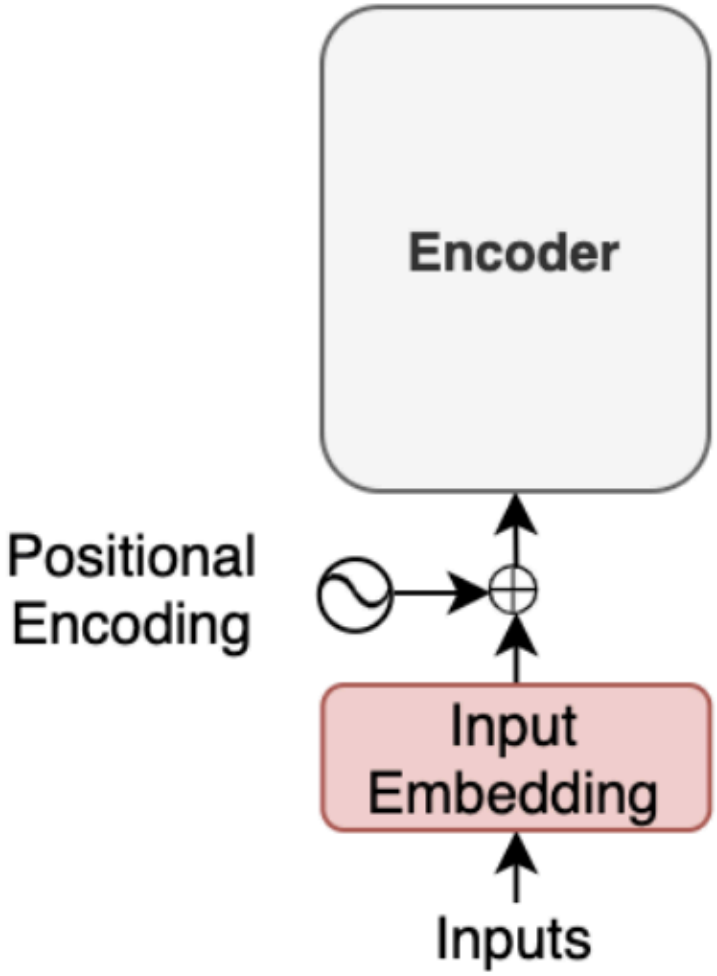
Each decoder block receives the features from the encoder. If we draw the encoder and the decoder vertically, the whole picture looks like the diagram from the paper.



The paper uses “Nx” (N-times) to indicate multiple blocks. So we can draw the same diagram in a concise format.

Input Embedding and Positional Encoding

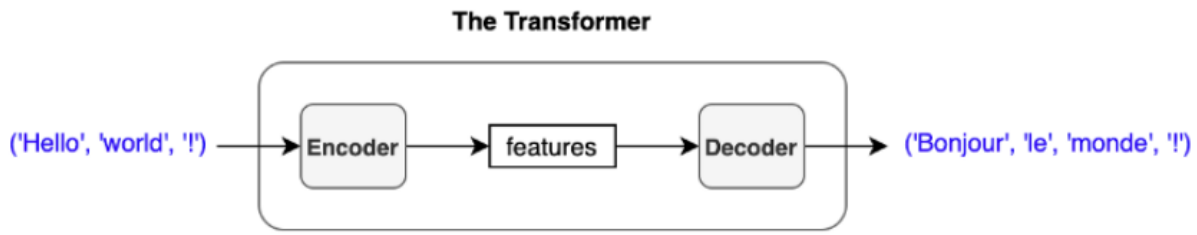
Like any neural translation model, we often tokenize an input sentence into distinct elements (tokens). A tokenized sentence is a fixed-length sequence.
To feed those tokens into the neural network, we convert each token into an embedding vector, a common practice in neural machine translation and other natural language models. In the paper, they use a 512-dimensional vector for such embedding. So, if the maximum length of a sentence is 200, the shape of every sentence will be (200, 512). The transformer learns those embeddings from scratch during training.
For embedding section look this Word Embeddings



Positional encoding is a technique used in models like Transformers to provide information about the position of elements (like words) in a sequence, enabling the model to understand the order of input data. It achieves this by adding position-dependent signals to word embeddings, which allows the model to capture the sequential nature of the input.

Softmax and Output Probabilities

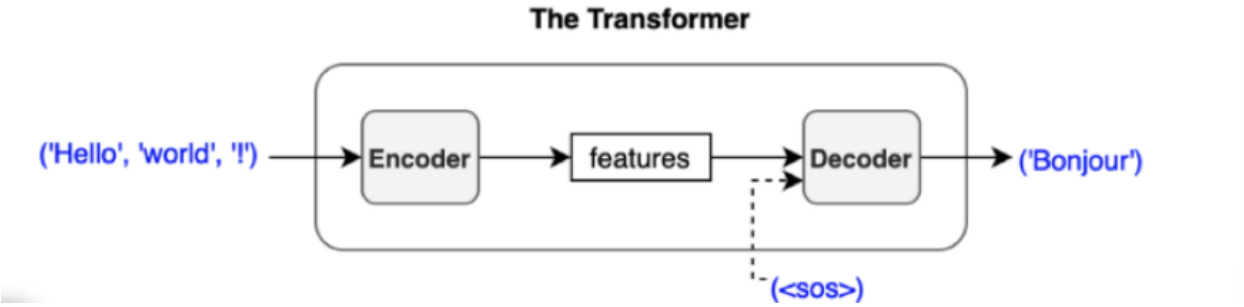
The decoder uses input features from the encoder to generate an output sentence. The input features are nothing but enriched embedding vectors.



For simplicity, I express a sentence like ('Hello', 'world', '!'), but the actual inputs to the encoder are input embeddings with positional encodings.

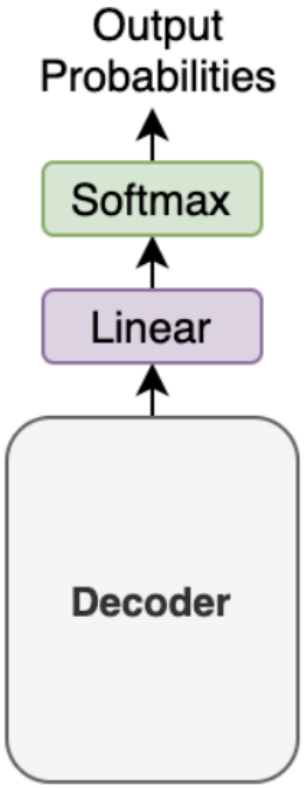
The decoder outputs one token at a time. An output token becomes the subsequent input to the decoder.
In other words, a previous output from the decoder becomes the last part of the next input to the decoder. This kind of processing is called **"auto-regressive"**—a typical pattern for generating sequential outputs and not specific to the transformer. It allows a model to generate an output sentence of different lengths than the input.

However, we have no previous output at the beginning of a translation. So, we pass the start-of-sentence marker <SOS> (as known as the beginning-of-sentence marker <BOS>) to the decoder to initiate the translation.



The decoder uses multiple decoder blocks to enrich <SOS> with the contextual information from the input features. In other words, the decoder transforms the embedding vector <SOS> into a vector containing information helpful to generating the first translated word (token).

Then, the output vector from the decoder goes through a linear transformation that changes the dimension of the vector from the embedding vector size (512) into the size of vocabulary (say, 10,000). The softmax layer further converts the vector into 10,000 probabilities.

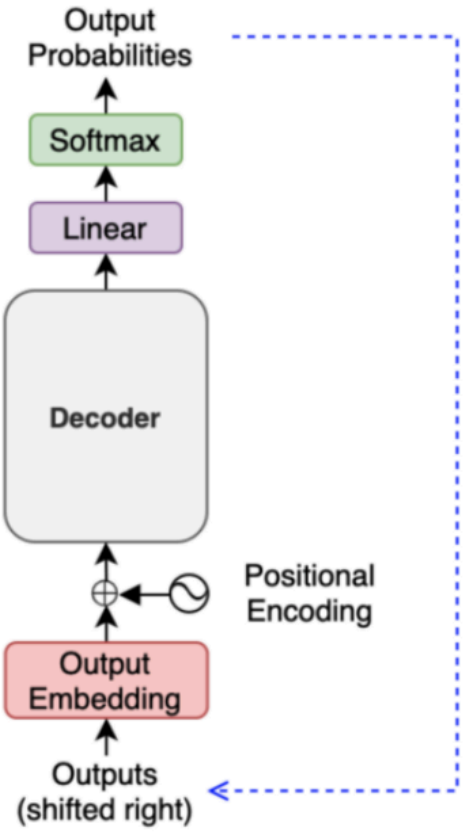


Let’s use a simple example with a tiny vocabulary dataset. It only has three words: 'like', 'cat', 'I'. The model predicts the probabilities for the first output token as [0.01, 0.01, 0.98]. In this case, the word 'I' is most probable. So, we should choose the word 'I'. Or should we?

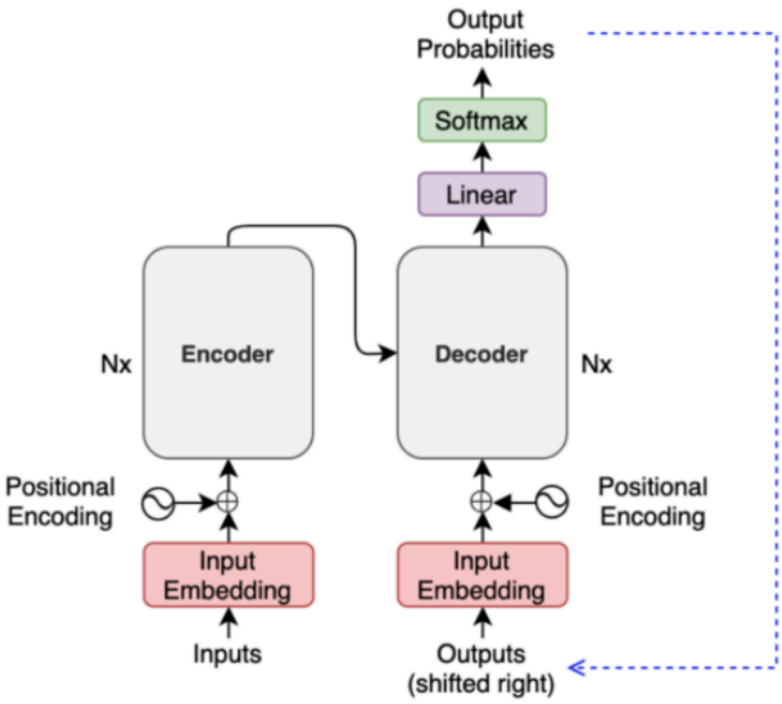
It depends. Granted, we must choose one word (token) from the calculated probabilities. This article assumes the greedy method, which selects the most probable word (i.e., the word with the highest probability).

However, another approach, called **beam search,** may produce better performance in the BLUE score(BLEU score measures the similarity between the generated text and a human-written reference text.).
The beam search looks for the best combination of the tokens rather than selecting the most probable token at a time. The paper also uses the beam search with a beam size of 4.

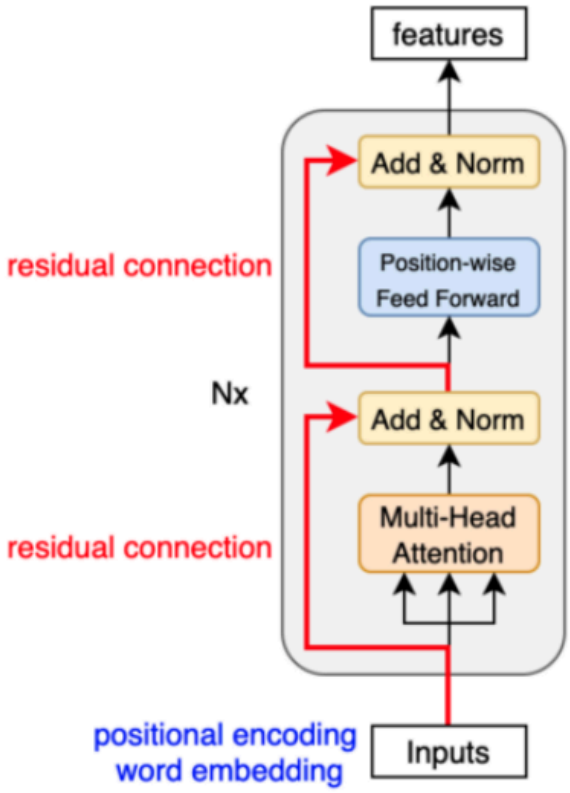
In any case, we feed the predicted word back to the decoder to produce the next token. As shown below, we provide the chosen token to the decoder as the last part of the next decoder input.



It may look like a slow process as we must generate one output at a time, especially for training. However, during training, we typically use the teacher-forcing method, Teacher forcing in Transformers refers to a training technique where the model is fed the correct target sequence as input at each step during training, rather than its own predicted output, making learning more stable. It also makes the training run faster as we can prepare an attention mask, allowing parallel processing.


Output Probabilities
Softmax
Linear
Nx
Encoder
Decoder
Nx
Positional Encoding
Positional Encoding
Input Embedding
Input Embedding
Inputs
Outputs (shifted right)

Encoder


features
Add & Norm
residual connection
Position-wise Feed Forward
Nx
Add & Norm
Multi-Head Attention
residual connection
positional encoding
word embedding
Inputs

MULTIHEAD ATTENTION

The encoder block uses the self-attention mechanism to enrich each token (embedding vector) with contextual information from the whole sentence.

Depending on the surrounding tokens, each token may have more than one semantic and/or function. Hence, the self-attention mechanism employs multiple heads (eight parallel attention calculations) hence building the Multi-head attention ,each self attention with different linear projections of the input data. This allows the model to capture diverse relationships within the input sequence.

The position-wise feed-forward network (FFN) has a linear layer, ReLU, and another linear layer, which processes each embedding vector independently with identical weights. So, each embedding vector (with contextual information from the multi-head attention) goes through the position-wise feed-forward layer for further transformation.

$$\mathbf{FFN}(\boldsymbol{x}) = \mathrm{ReLU}(\boldsymbol{x}W_1 + b_1)W_2 + b_2$$

RELU VS Leaky ReLU vs ELU

<ReLU> It mitigates Vanishing Gradient Problem.

Cons: It causes Dying ReLU Problem. It's non-differentiable at x=0.

<Leaky ReLU>

It mitigates the Vanishing Gradient Problem.

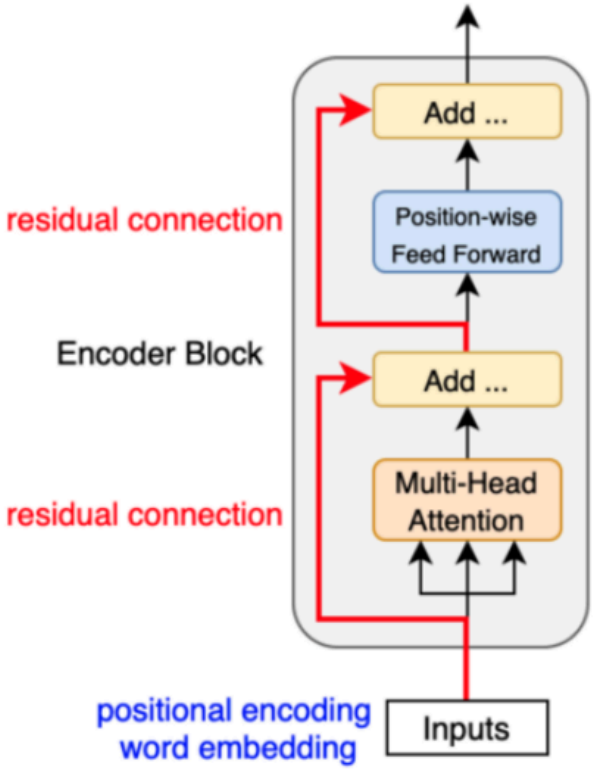
It mitigates the Dying ReLU Problem. *0 is still produced for the input value 0 so the Dying ReLU Problem is not completely avoided.

Cons: It's non-differentiable at x=0.

<ELU>

It normalizes negative input values so the convergence with negative input values is stable.
It mitigates the Vanishing Gradient Problem.
It mitigates Dying ReLU Problem. *0 is still produced for the input value 0 so Dying ReLU Problem is not completely avoided.
Cons:
It's computationally expensive because of exponential operation.
It's non-differentiable at x = 0 if a is not 1.

The point is Add....
Another point is that the encoder block uses residual connections, which is simply an element-wise addition:



Sublayer is either multi-head attention or point-wise feed-forward network.

Residual connections carry over the previous embeddings to the subsequent layers. As such, the encoder blocks enrich the embedding vectors with additional information obtained from the multi-head self-attention calculations and position-wise feed-forward networks.

After each residual connection, there is a layer normalization:

$$\text{LayerNorm}(\boldsymbol{x} + \text{Sublayer}(\boldsymbol{x}))$$

WHAT IS BATCH AND LAYER NORMALISATION

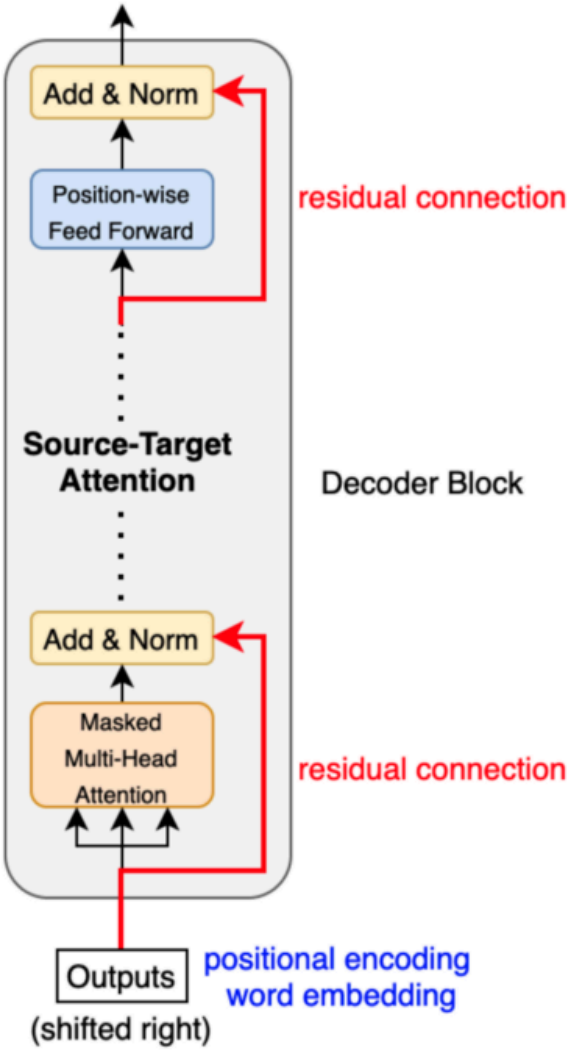
Like batch normalization, layer normalization aims to reduce the effect of covariant shift. In other words, it prevents the mean and standard deviation of embedding vector elements from moving around, which makes training unstable and slow (i.e., we can't make the learning rate big enough). Unlike batch normalization, layer normalization works at each embedding vector (not at the batch level).

Decoder Block Internals

The decoder block is similar to the encoder block, except it calculates the source-target attention.

Each **decoder block** has 3 sub-layers:

1. **Masked Multi-Head Self-Attention**
2. **Encoder-Decoder Attention**
3. **Feed-Forward Neural Network (FFN)**



As mentioned before, an input to the decoder is an output shifted right, which becomes a sequence of embeddings with positional encoding. So, we can think of the decoder block as another encoder generating enriched embeddings useful for translation outputs.

Masked multi-head attention means the multi-head attention receives inputs with masks so that the attention mechanism does not

use information from the hidden (masked) positions. The paper mentions that they used the mask inside the attention calculation by setting attention scores to negative infinity (or a very large negative number). The softmax within the attention mechanisms effectively assigns zero probability to masked positions.

Encoder-Decoder Cross-Attention

Uses information from the input sentence.

Here's where the decoder looks at the encoder's output (representing the input sentence like "The weather is great").
This layer answers:
"Based on the input, what should I generate next?"

This cross-attention connects the output being generated to the input sentence — bringing in relevant parts of the input.

🧠 So now the decoder knows:

What has been generated so far (from self-attention)
What parts of the input are relevant for the next word (from cross-attention)

Feed-Forward Layer

After attention, the combined embeddings go through a fully connected layer to add non-linearity and generate the next token's embedding.

CONCLUSION
✅ "The decoder looks at the already generated output (partial output) using **masked self-attention`,
✅ Then it pulls relevant parts of the input (using cross-attention),
✅ And finally uses this combined context to predict the next word."

SUMMARY OF TRANSFORMERS

🏗️ Architecture Overview:

The Transformer is composed of:
Encoder Stack (N layers) → understands the input
Decoder Stack (N layers) → generates the output

Each block contains multi-head attention, layer norm, and feed-forward layers

Step-by-Step Flow:

1. Input Embedding + Positional Encoding
Input words are converted to word embeddings (vectors)

Positional encodings are added to help the model understand word order
Because transformers don't have RNNs, they need position info explicitly.

2. Encoder: Understand the Input Sentence
Each encoder layer has:

Self-Attention → Every word looks at all other words in the sentence to capture context.

Feed-Forward Network (FFN) → Adds non-linearity, processes token meaning further.
Residual connections + LayerNorm

🧠 All encoder layers produce a set of contextualized embeddings — a deep representation of the input sentence.

3. Decoder: Generate the Output Sentence
Each decoder layer has:
Masked Self-Attention → Looks only at previous tokens (so future words aren't seen)
Encoder-Decoder Attention → Focuses on relevant parts of the input
Feed-Forward Network

At each time step, the decoder:
Uses already generated tokens ("partial output")
Attends to encoder output
Predicts the next token

4. Output Layer
Final output goes through a linear layer + softmax
This gives a probability distribution over the vocabulary
The word with the highest probability is chosen as the next output token.