

# Transformers, Tokenizers, and the GPT Architecture

Arda Tasci  
arda@nofi.nyc

April 2, 2026

# Contents

|          |                                                                         |           |
|----------|-------------------------------------------------------------------------|-----------|
| <b>1</b> | <b>A Brief History of Language Modeling</b>                             | <b>4</b>  |
| <b>2</b> | <b>Tokenizers</b>                                                       | <b>5</b>  |
| 2.1      | Motivation . . . . .                                                    | 5         |
| 2.2      | How Tokenizers Work . . . . .                                           | 5         |
| 2.3      | Choosing n-grams . . . . .                                              | 5         |
| 2.4      | Byte-Pair Encoding . . . . .                                            | 6         |
| 2.5      | Conclusion . . . . .                                                    | 6         |
| <b>3</b> | <b>Embeddings and Positional Encoding</b>                               | <b>7</b>  |
| 3.1      | Input Embeddings . . . . .                                              | 7         |
| 3.1.1    | Motivation . . . . .                                                    | 7         |
| 3.1.2    | Brief History . . . . .                                                 | 7         |
| 3.1.3    | How Embedding Layers Work . . . . .                                     | 8         |
| 3.2      | Positional Encoding . . . . .                                           | 8         |
| 3.2.1    | Motivation . . . . .                                                    | 8         |
| 3.2.2    | Absolute Positional Encoding . . . . .                                  | 9         |
| 3.2.3    | Relative Positional Encoding . . . . .                                  | 10        |
| <b>4</b> | <b>Attention</b>                                                        | <b>11</b> |
| 4.1      | Dot Product Attention . . . . .                                         | 11        |
| 4.2      | Scaled Dot-Product (Self) Attention . . . . .                           | 12        |
| 4.3      | Multi-Head (Self) Attention . . . . .                                   | 13        |
| 4.3.1    | Motivation . . . . .                                                    | 13        |
| 4.3.2    | Structure . . . . .                                                     | 13        |
| 4.4      | Computational Complexity of Attention . . . . .                         | 14        |
| 4.5      | Grouped Query Attention (GQA) and Multi-Query Attention (MQA) . . . . . | 14        |
| 4.5.1    | Motivation . . . . .                                                    | 14        |
| 4.5.2    | Structure . . . . .                                                     | 14        |
| 4.6      | Position-Wise Feed-Forward Networks (FFN) . . . . .                     | 16        |
| 4.6.1    | What's a FFN? . . . . .                                                 | 16        |
| 4.6.2    | Motivation . . . . .                                                    | 16        |
| 4.6.3    | Mathematical Definition . . . . .                                       | 16        |
| 4.7      | Layer Normalization . . . . .                                           | 16        |
| 4.7.1    | Motivation . . . . .                                                    | 16        |
| 4.7.2    | Mathematical Definition . . . . .                                       | 16        |
| 4.8      | Cross Attention . . . . .                                               | 17        |
| 4.8.1    | Motivation . . . . .                                                    | 17        |
| 4.8.2    | Mathematical Definition . . . . .                                       | 17        |
| <b>5</b> | <b>The Encoder Block</b>                                                | <b>19</b> |
| 5.1      | Residual Connections . . . . .                                          | 19        |
| 5.1.1    | Motivation . . . . .                                                    | 19        |

|          |                                              |           |
|----------|----------------------------------------------|-----------|
| 5.1.2    | Mathematical Definition . . . . .            | 19        |
| 5.2      | Full Block Structure . . . . .               | 19        |
| <b>6</b> | <b>The Decoder Block</b>                     | <b>20</b> |
| 6.1      | Structure . . . . .                          | 20        |
| 6.2      | How Encoder and Decoder Interact . . . . .   | 20        |
| <b>7</b> | <b>The Generative Pretrained Transformer</b> | <b>21</b> |
| 7.1      | Architecture Overview . . . . .              | 21        |
| 7.2      | Causal (Masked) Self-Attention . . . . .     | 21        |
| 7.2.1    | Motivation . . . . .                         | 21        |
| 7.2.2    | Causal Masking . . . . .                     | 21        |
| 7.3      | GPT Decoder Block . . . . .                  | 21        |
| 7.4      | Pre-Training . . . . .                       | 22        |
| 7.4.1    | Objective . . . . .                          | 22        |
| 7.4.2    | The Language Modeling Head . . . . .         | 22        |
| 7.5      | Autoregressive Inference . . . . .           | 22        |
| 7.5.1    | Sampling Strategies . . . . .                | 23        |
| 7.6      | KV Cache . . . . .                           | 23        |
| 7.6.1    | Motivation . . . . .                         | 23        |
| 7.6.2    | The Fix . . . . .                            | 23        |
| 7.7      | Fine-Tuning . . . . .                        | 24        |
| 7.8      | Conclusion . . . . .                         | 25        |

# 1 A Brief History of Language Modeling

Understanding the previous research on sequence prediction/transduction allows us to ultimately understand modern methods and models better, that being said I can only afford to talk about a general overview of the field before Transformers. If you want a deep dive please read up on Prof. Graham Neubig’s tutorial: <https://arxiv.org/pdf/1703.01619>

- The main motivation of research in this field was sequence-to-sequence modeling.
- Formally, starting from some source sequence  $\mathbf{F} = [f_0, f_1, \dots, f_n]$  where each  $f_i$  represents some word from a language.
- Our goal is to translate  $\mathbf{F}$  into sequence  $\mathbf{E} = [e_0, e_1, \dots, e_k]$  where each  $e_i$  represents some word from another language. Probabilistically:

$$Pr(\mathbf{E}|\mathbf{F}) = Pr(e_0|\mathbf{F}) \cdot Pr(e_1|e_0, \mathbf{F}) \cdot \dots \cdot Pr(e_k|e_0, \dots, e_{k-1}, \mathbf{F})$$

- We end generation when our end-of-sentence token has the highest probability.

The first men used to employ Recurrent Neural Networks (RNNs) to make machines talk. Over the years, the RNN architecture was realized to (probably) its full potential through techniques such as:

- multiple RNNs making backwards and forwards passes on every input (bi-directional encoding)
- feeding it ground-truths during training (teacher forcing)
- an early version of attention (Additive/Bahdanau attention)

Despite these improvements, RNNs still struggled due to how they could only read sequentially. Fast-forward to 2017, and the groundbreaking ‘Attention is All You Need’ paper introduced a brand-new architecture: the Transformer, built solely on attention mechanisms. Attention really was all we needed!

## 2 Tokenizers

### 2.1 Motivation

- All ML models function by minimizing loss between predictions and ground-truths. The best and only way we know how is through numbers; namely via vectors, matrices, and floating-point values.
- Since words are not numbers, we need some sort of way to convert them into things we can operate on.

### 2.2 How Tokenizers Work

In essence, a tokenizer is just a function that maps n-grams (text sequence of length  $n$ ) to tokens, and returns their respective token IDs. Here's a simple example of word-level tokenization:

"I am happy"  $\rightarrow$  ["I", "am", "happy"]  $\rightarrow$  [42, 12, 94]

**Note:** training requires padding every input to a fixed length s.t. every input in a batch is equivalent in length.

### 2.3 Choosing n-grams

Choosing the granularity for tokenization is mostly up to preference; that being said, there are certain caveats and benefits to each.

Older models often used word-level tokenization, but modern models employ subword-level approaches.

- If you go too coarse (e.g. word level), you end up with huge token vocabularies and potential out-of-vocabulary errors. Additionally, you have to account for different forms of the same word (i.e. "run", "running", "runner") or different prefixes/suffixes.
- If you go for character-level tokenization, the length of your input gets too long, and you lose performance (Attention calculations scale roughly  $O(n^2)$ ).

## 2.4 Byte-Pair Encoding

Byte-Pair Encoding (BPE) is the weapon-of-choice for most models today. BPE is the golden mean between both ends of the granularity spectrum. BPE works by tokenizing subwords based on frequency rather than using a constant token length. Here's how the tokenizer learns token vocabularies:

1. Initialize base vocabulary of all UTF-8 bytes (256 total). This allows us to tokenize special characters and emojis.
2. Convert your own corpus into UTF-8 bytes and split at byte-level.  
e.g. ['cat', 'hat'] → [('c','a','t'), ('h','a','t')]
3. Afterwards, count all contiguous token pairs.
4. Merge most common token pair into one.  
e.g. [('c', 'a', 't'), ('h', 'a', 't')] → ['c', 'at', 'h', 'at']
5. Repeat merging until desired vocabulary size.

**Note:** Once trained, BPE is deterministic. Also, newer tokenizers opt to merge whitespace (e.g. '\_c', 'at\_').

## 2.5 Conclusion

Understanding how tokenizers work is essential to understanding weird LM behaviour. Having an intimate knowledge of how transformers work end-to-end is crucial to developing and building better models. Hopefully this chapter has helped you understand how LMs can parse language input.

## 3 Embeddings and Positional Encoding

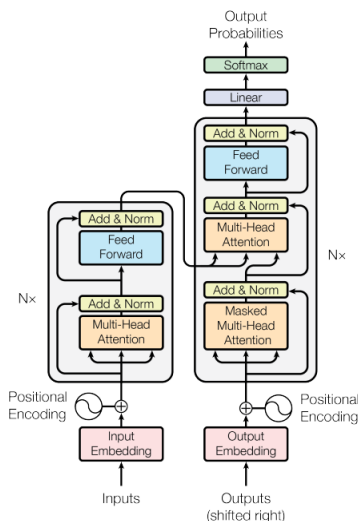


Figure 1: The basic transformer architecture.

### 3.1 Input Embeddings

#### 3.1.1 Motivation

- The encoder picks up where the tokenizer left off, starting from integer arrays of token IDs.
- Integers by themselves cannot represent any semantic value (that isn't their job anyway; they are just for vocab lookup).
- So, we need some way of capturing the 'meaning' and intricacies of language.

#### 3.1.2 Brief History

Early approaches were often as simple as Principle Component Analysis (PCA) or Singular Value Decomposition (SVD). Later, ML algorithms such as Word2Vec were employed. Since 2018, with the introduction of Google's BERT model, transformers have been learning their own embeddings during training.

### 3.1.3 How Embedding Layers Work

- Our goal is to turn every input of token IDs into a vector of floating-point values.
- Initially, we one-hot encode our vocabulary and choose a dimension for our embeddings.
- The embedding layer involves learning a matrix  $E \in \mathbb{R}^{V \times d}$  (vocab size, embed dim) that is learned alongside everything else in backpropagation.
- Here, ‘embed dim’ is an arbitrary factor of 2 (optimal for binary computers). Bigger  $d$  allows us more depth and richness in our embeddings at the cost of latency and compute.
- The weights of  $E$  are usually learned through what is essentially a linear layer with no biases or activations. One key difference is that only rows corresponding to the training pass’ token IDs are affected by backpropagation. More formally:  $\frac{\partial L}{\partial E[i]}$  for some loss  $L$  and embedding  $E[i]$ .

Embeddings are deterministic at every training step. For example, at epoch  $X$ , the token ID ‘42’ will map to some vector  $E[42]$  every time.

You can think of embedding matrices as dictionaries that we can retrieve embeddings from according to some index token ID. These vectors then get passed along in an array.

## 3.2 Positional Encoding

### 3.2.1 Motivation

Here’s what our input looks like from end-to-end,

$$\begin{array}{c} \text{“the cat in the hat”} \\ \downarrow \\ [\text{‘the’, ‘c’, ‘at’, ‘i’, ‘n’, ‘the’, ‘h’at}] \\ \downarrow \\ [1, 2, 3, 4, 5, 1, 6, 3] \\ \downarrow \\ E = \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \mathbf{e}_3 \\ \vdots \\ \mathbf{e}_8 \end{bmatrix} \text{ where } \mathbf{e}_i = [0.163..., 0.425..., 0.952..., \dots, 0.658...] \end{array}$$

This is great so far, but there is one crucial oversight (not here but in attention itself); the way transformers fundamentally work does not pay attention to the positions of our inputs.

This means that the model:

- has no way of explicitly ‘knowing’ that tokens ‘c’, ‘at’ form the word ‘cat’.
- also has trouble ‘understanding’ the difference between ‘the cat in the hat’ and ‘the hat in the cat’ (or any permutation of those tokens for that matter).

To remedy this, we bake positional data into our embeddings before passing them into attention blocks.

### 3.2.2 Absolute Positional Encoding

APE involves finding some way to represent an embedding’s position versus the entire sequence. The most well-known method is Sinusoidal Positional Encoding (Vaswani et al. 2017):

$$\begin{aligned} PE_{(pos, 2i)} &= \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \text{ for even dimension indices,} \\ PE_{(pos, 2i+1)} &= \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \text{ for odd dimension indices.} \end{aligned}$$

where

- $pos$ : position index in sequence
- $d_{model}$ : embedding dimension
- $i$ : dimension index ( $0 \dots d_{model} - 1$ )

These formulae create unique positional signatures for each dimension. PE gets calculated for every index in the sequence, then we add it to each embedding.

$$\mathbf{x}_i = \mathbf{e}_i + \mathbf{PE}_i$$

**Why add, not concat?** Intuitively, it may seem wrong to add positional data into our embeddings, but empirically the Transformer team found that it works. (**It seems** that in high dimensional embedding spaces, word vectors and positional vectors exist in different smaller dimensional subspaces, allowing us to manipulate both with a single transformation. Performance would also be hurt if we concatenated.)

There are a bunch of other APEs but this is the only one you really have to know.

Also, note that there’s no learning going on here.

### 3.2.3 Relative Positional Encoding

RPE sets out to mend the main weaknesses of APE: max token limits. We calculate **pairwise** positional encodings between every possible  $i$  and  $j$  token pair. These embeddings can differ in shape and form, but they historically *typically* involve some sort of learning. The most promising one right now, however, does not; **RoPE** (Rotary Positional Embedding) is used in LLaMA (and maybe some closed source models as well).

RoPE doesn't mess with our embeddings, but instead directly injects pairwise information with a rotation operation applied on the Key and Query vectors. We'll discuss this in depth in the attention chapter, but here's a run down:

Intuitively, we split our input vector  $x = [x_1, x_2, \dots, x_{dim}]$  into two parts. (Assuming  $dim$  is even [which it should be.])

$$x = [x_1, x_2, \dots, x_{dim}] \rightarrow \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \\ \vdots & \vdots \\ x_{dim-1} & x_{dim} \end{bmatrix}$$

Then, we rotate every row of  $x$  with this rotation matrix:

$$R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

We choose  $\theta$  as a position and dimension-dependent frequency:

$$\theta_{p,i} = p \cdot \omega_i = \frac{p}{10000^{2i/dim}}$$

where  $i$  is the row index and  $p$  is the sequence position index. So, the product of our rotation can be expressed as

$$x_i^{\text{rot}}(p) = R(\theta_{p,i}) \cdot x_i$$

and the full vector as:

$$\text{RoPE}(x, p) = [x_0^{\text{rot}}(p), x_1^{\text{rot}}(p), \dots, x_{dim-1}^{\text{rot}}(p),]$$

We apply this to  $Q$  and  $K$  when calculating attention between tokens  $i$  and  $j$ ,

$$\begin{aligned} Q_i^{\text{rot}} &= \text{RoPE}(Q_i, i) \\ K_j^{\text{rot}} &= \text{RoPE}(K_j, j) \\ \therefore \text{Score}(i, j) &= Q_i^{\text{rot}} \cdot K_j^{\text{rot}} \end{aligned}$$

In practice we can reduce this dot product to:

$$Q_i^{\text{rot}} \cdot K_j^{\text{rot}} = Q^{T_i} \cdot R(\theta_{j-i}) K_j$$

## 4 Attention

‘Attention’ is the internal mechanism under all transformer encoder/decoder blocks. It can take different shapes and forms but the elementary operations are the same. We’ll discuss those and then move on to different flavors of attention.

### 4.1 Dot Product Attention

This was an early version of attention from the 2015 Luong et al. paper. Not really used nowadays; regardless it’s important to understand how something developed to motivate the choices and improvements the researchers made with each iteration.

Here’s the minimum amount of context you need to understand how this works:

- Recall that RNNs receive input sequences token-by-token.
- At every step of the way, the RNN updates a fixed-length hidden vector  $h_i$  where  $i$  is the position in the sequence. ( $h_0$  is randomly initialized)
- Every  $h_i$  gets stored in matrix  $H$  because it contains new information about the sequence (initially researchers would only keep the last  $h$ , which was worse). So,  $H$  looks like

$$\begin{bmatrix} h_1^1 & h_1^2 & \dots & h_1^n \\ h_2^1 & h_2^2 & \dots & h_2^n \\ \vdots & \vdots & \ddots & \vdots \\ h_m^1 & h_m^2 & \dots & h_m^n \end{bmatrix}$$

(note that  $n$  is a hyperparameter.)

Once we finish going through our input, we’re ready to decode (generate sequence). But we have too much data on our hands: **How do we know what tokens to care about?** Furthermore, note that  $H$  **gets longer with every token, and we need a fixed-size matrix** for our learning to actually be useful across inputs of different length.

This is the power of attention. Formally, at each step of generating our target sequence  $e_i$ , we want to give our decoder RNN a fixed-size context vector  $\mathbf{c}_i$  that tells it what’s most relevant.

We do this by taking the weighted average of  $H$  and collapsing it into one vector. We get our weights from taking the dot product between  $h_i^{\text{decoder}}$  and every  $h_i^{\text{encoder}}$  and converting those scores into weights (for weighted avg) with softmax.

Let  $h_i^{\text{decoder}} := q_i$  and  $h_i^{\text{encoder}} := v_i$ . Then,

$$e_i = q^\top v_i \quad (1)$$

$$\alpha_i = \frac{\exp(e_i)}{\sum_j \exp(e_j)} = \text{softmax}(e_i) \quad (2)$$

$$c_i = \sum_j \alpha_j v_j \quad (3)$$

$c_i$  looks like

$$c_i = [h_1^*, h_2^*, \dots, h_n^*]$$

Note how we renamed our hidden-state vectors to  $q$  (query) and  $v$  (value). This is because we use each decoder hidden state as our query in a similarity search wrt each key.

**PS:** You might be wondering where the  $k$  (key) vector is. In RNN DP attention,  $k \equiv v$  since we don't learn any specific way to attend to each value and use the values themselves. This changes in Transformers.

## 4.2 Scaled Dot-Product (Self) Attention

This is the function used in all layers of the original Transformer. Let's understand how it works mathematically.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{\dim_k}}\right) V = \text{score}(Q, K, V) \cdot V$$

The math here is pretty simple. If you don't understand, note these key differences from RNN dot product attention:

- Since  $k$  no longer equals  $v$ , we separate them in the equation.
- We're working with matrices instead of vectors. This is because we're batching the vector calculations from earlier into one. This is the fundamental improvement of the Transformer over the RNN.
- We divide our similarity scores by a scaling factor  $\sqrt{d_k}$  to ensure large inputs don't result in diminishing gradients.

**Why is  $Q \neq K$ ?** This is because we're **learning** how to attend to each token through a linear projection for  $K$ , and learning what to pass as context to our decoder. Formally, for some input matrix  $X$ :

$$\begin{aligned} Q &= XW^Q, \\ K &= XW^K, \\ V &= XW^V \end{aligned}$$

Where each  $W$  is our specific learned weight matrix. If really tickled to, you could write out the matrix operations but you should already have a solid understanding as to what is going on from the previous chapter. However the final operation of this expression is very important to grasp. For brevity's sake, let

$$\text{softmax}\left(\frac{QK^\top}{\sqrt{\dim_k}}\right) = \begin{bmatrix} s_0^0 & s_0^1 & \dots & s_0^n \\ s_1^0 & s_1^1 & \dots & s_1^n \\ \vdots & \vdots & \ddots & \vdots \\ s_m^0 & s_m^1 & \dots & s_m^n \end{bmatrix}$$

Note that each row sums to 1. When we multiply  $V$  by this matrix,

$$\begin{bmatrix} s_0^0 & s_0^1 & \dots & s_0^n \\ s_1^0 & s_1^1 & \dots & s_1^n \\ \vdots & \vdots & \ddots & \vdots \\ s_m^0 & s_m^1 & \dots & s_m^n \end{bmatrix} \begin{bmatrix} v_0^0 & v_0^1 & \dots & v_0^n \\ v_1^0 & v_1^1 & \dots & v_1^n \\ \vdots & \vdots & \ddots & \vdots \\ v_m^0 & v_m^1 & \dots & v_m^n \end{bmatrix} = \begin{bmatrix} \sum_i s_0^i v_i^0 & \sum_i s_0^i v_i^1 & \dots & \sum_i s_0^i v_i^n \\ \sum_i s_1^i v_i^0 & \sum_i s_1^i v_i^1 & \dots & \sum_i s_1^i v_i^n \\ \vdots & \vdots & \ddots & \vdots \\ \sum_i s_m^i v_i^0 & \sum_i s_m^i v_i^1 & \dots & \sum_i s_m^i v_i^n \end{bmatrix}$$

Basically, at the end of these matmuls, we're still doing the same thing we did in RNN attention: creating weighted averages from the rows of  $V$  using scaled similarity scores as weights.

## 4.3 Multi-Head (Self) Attention

### 4.3.1 Motivation

We've talked about the building blocks of attention. Now, we want to scale this mechanism to actually perform meaningfully. The most natural next step is to duplicate SDPA to capture more patterns and depth. That is what Multi-Head Attention is.

Mathematically, a single Attention mechanism can only learn one type of relationship since the dot product projects queries and keys into a single subspace. MHA means we can be in multiple subspaces simultaneously.

### 4.3.2 Structure

Not much changes except for the fact that we now have  $h$  'heads' indexed as:

$$Q_i = XW_i^Q, K_i = XW_i^K, V_i = XW_i^V$$

Where  $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ , and usually  $d_k = d_v = d_{\text{model}}/h$ .

After applying SDPA independently for each head, we bring the results together by concatenating heads:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) \cdot W^O$$

Where  $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$  is a learned projection back to the model's dimension.

## 4.4 Computational Complexity of Attention

One thing worth internalizing is why attention is expensive. For a sequence of length  $n$  and embedding dimension  $d$ , computing  $QK^\top$  requires comparing every token against every other token, which is  $n^2$  dot products, each of dimension  $d$ .

So the full cost of one attention layer is  $O(n^2 \cdot d)$ . The  $n^2$  term is the painful one. Double your sequence length and your attention cost quadruples. This is why long-context models are hard and why a lot of research has gone into approximating or replacing the full attention matrix.

**Note:** Memory scales with  $O(n^2)$  as well, since we have to store the full attention matrix. For very long sequences this becomes the bottleneck before compute does.

## 4.5 Grouped Query Attention (GQA) and Multi-Query Attention (MQA)

### 4.5.1 Motivation

Standard MHA has  $h$  heads, each with its own  $K$  and  $V$  matrices. At inference time, we need to store those  $K$  and  $V$  tensors in memory for every layer and every token generated so far (more on this in the KV Cache section). With many heads, this gets expensive fast.

**Multi-Query Attention** (Shazeer 2019) is the aggressive fix: share a single  $K$  and  $V$  across all heads. Only  $Q$  stays per-head. This shrinks the KV memory footprint by a factor of  $h$ , at some cost to quality.

**Grouped Query Attention** (Ainslie et al. 2023) is the middle ground: split the  $h$  query heads into  $g$  groups, and give each group its own  $K$  and  $V$ . MHA is GQA with  $g = h$ ; MQA is GQA with  $g = 1$ .

### 4.5.2 Structure

For GQA with  $g$  groups and  $h$  query heads (so  $h/g$  query heads per group):

$$Q_i = XW_i^Q, \quad K_j = XW_j^K, \quad V_j = XW_j^V$$

where  $i \in \{1, \dots, h\}$  and  $j \in \{1, \dots, g\}$ . Each group of  $h/g$  query heads shares the same  $K_j$  and  $V_j$ . The rest of the attention computation is identical to MHA.

In practice, GQA with  $g = h/4$  or similar gets most of the memory savings of

MQA while barely sacrificing quality. LLaMA 2/3, Mistral, and most modern open-source models use GQA.

## 4.6 Position-Wise Feed-Forward Networks (FFN)

### 4.6.1 What's a FFN?

A feedforward network is simply a neural network with an input, output, and hidden layer.

### 4.6.2 Motivation

So far, notice that everything we've done is (mostly) linear projections (aside from the softmax we use to score attention). This is not ideal for a few reasons, but the most important one to grasp is that linear combinations alone are limited in their expressiveness. For example, consider the tokens 'not' and 'good'. When put together, the semantic meaning of 'not good' cannot be equated to the sum of 'not' + 'good' (though not technically correct, a good way of internalizing this intuition is that the negative value of 'not' combined with the positive value of 'good' would land in between the two manifolds of negative and positive. This wouldn't be correct because 'not good' is an objectively negative sentiment).

### 4.6.3 Mathematical Definition

The FFN used in the Transformer is defined as such:

$$\text{FFN}(x) = \sigma(xW_1 + b_1)W_2 + b_2$$

Where  $\sigma$  is a non-linear activation function (ReLU in the Transformer paper). This allows the model to learn non-linear transformations, such as the fact that the word 'not' inverts whatever word comes after it.

## 4.7 Layer Normalization

### 4.7.1 Motivation

LayerNorm is a housekeeping function that aims to stabilize training.

**Why is training unstable?** Think about a deep neural network: every time a layer's weights are updated, the input it passes to the subsequent layer also changes. This means that networks are prone to **Internal Covariate Shift (ICS)**, the phenomenon where the input distribution to a layer drifts in a non-meaningful way (due to weight changes rather than new patterns in data), slowing down convergence. To mend this issue, we need some way of fixing our input distributions after each layer.

### 4.7.2 Mathematical Definition

LayerNorm fixes the mean  $\approx 0$  and the variance  $\approx 1$  for each token's embedding (scaling is done per-token because sequence lengths vary). This makes gradients more predictable and allows for higher learning rates.

The first step is to compute mean and variance across the feature dimension.

$$\mu = \frac{1}{d} \sum_{j=1}^d x_j, \quad \sigma^2 = \frac{1}{d} \sum_{j=1}^d (x_j - \mu)^2$$

Normalize:

$$\hat{x}_j = \frac{x_j - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Where  $\epsilon$  is a small constant for stability. This is all well and good, but fixing mean and variance too aggressively might result in loss of expressivity/flexibility for the network. For that reason, we introduce learnable parameters to let the network decide what's best. Introducing learnable scaling and bias parameters  $(\gamma, \beta)$ :

$$y_j = \gamma_j \hat{x}_j + \beta_j$$

Where  $\gamma, \beta \in \mathbb{R}^d$  so that the network can undo normalization or rescale features if needed.

## 4.8 Cross Attention

### 4.8.1 Motivation

So far, we've only talked about self-attention, which obtains the QKV matrices all from the same input sequences. This is enough for the encoder block of the Transformer, which only concerns itself with turning the input into matrices. However, within the decoder block, we have two different sequences we're working with: generated output and our encoded input. For example, consider the task of translation: translating the phrase 'I am a male' to Italian.

"I am a man"  
 $\downarrow$   
 "Io ... " — Model: 'Given what I've generated so far, what do I need to generate next?'

Concretely, we need to query our Italian output, and decide what (encoded) English input we must attend to. Another motivating example would be multimodal inputs, where we'd need to attend to parts of an image given a text input.

### 4.8.2 Mathematical Definition

Cross Attention is when Q is obtained from one source, and K, V are obtained from another source. Formally, given:

$$\begin{aligned}
Q &\in \mathbb{R}^{n_q \times d} \\
K &\in \mathbb{R}^{n_k \times d} \\
V &\in \mathbb{R}^{n_k \times d_v}
\end{aligned}$$

Note:  $n_k$  is not necessarily equal to  $n_q$ . The formula for cross attention is equivalent to self-attention. The only difference is that we obtain  $Q$  from elsewhere.

## 5 The Encoder Block

### 5.1 Residual Connections

#### 5.1.1 Motivation

One other reason training deep networks is hard (aside from ICS) is the **vanishing gradient problem**. As you backpropagate through many layers, gradients get multiplied by numbers smaller than 1 over and over, eventually shrinking to near zero. Early layers stop learning.

Residual connections (He et al. 2016) are the standard fix. Instead of passing  $x$  through a sub-layer and only keeping the output, we add  $x$  back to the result as a shortcut. This way, gradients always have a direct path to flow backward through the network.

#### 5.1.2 Mathematical Definition

For any sub-layer (MHA or FFN):

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

Even if  $\text{Sublayer}(x)$  vanishes, gradients still flow through the  $x$  term directly.

**Note:** This is *post-norm*, the original paper’s formulation. Modern models (including GPT) use *pre-norm*, applying LayerNorm before the sub-layer:

$$\text{Output} = x + \text{Sublayer}(\text{LayerNorm}(x))$$

Pre-norm trains more stably at scale, so you will see it basically everywhere now.

### 5.2 Full Block Structure

An encoder block is MHA and FFN wired together with residual connections and LayerNorm. Formally, for input  $X$ :

$$X' = \text{LayerNorm}(X + \text{MHA}(X))$$

$$X'' = \text{LayerNorm}(X' + \text{FFN}(X'))$$

(post-norm notation, per the original paper)

The original Transformer stacks  $N = 6$  of these blocks. Each block refines the token representations further, building up richer contextual information with every pass. The output is a matrix of context-aware representations. In encoder-decoder models, these get passed to the decoder via cross attention. In GPT, there is no encoder at all.

## 6 The Decoder Block

This section covers the decoder block from the *original* encoder-decoder Transformer (Vaswani et al. 2017). If you only care about GPT, you can skip this, but it is worth understanding since cross attention lives here, and many non-GPT models (e.g. translation, speech, vision-language) still use the full encoder-decoder setup.

### 6.1 Structure

The decoder block has three sub-layers instead of two, each wrapped with a residual connection and LayerNorm:

1. **Masked Multi-Head Self-Attention:** the decoder attends to its own previously generated tokens. Causal masking is applied so it cannot peek at future positions.
2. **Cross Attention:** the decoder queries the encoder’s output.  $Q$  comes from the decoder,  $K$  and  $V$  come from the encoder.
3. **FFN:** same position-wise feedforward as in the encoder.

Formally, for decoder input  $Y$  and encoder output  $Z$ :

$$\begin{aligned} Y' &= \text{LayerNorm}(Y + \text{MaskedMHA}(Y)) \\ Y'' &= \text{LayerNorm}(Y' + \text{CrossAttn}(Q=Y', K=Z, V=Z)) \\ Y''' &= \text{LayerNorm}(Y'' + \text{FFN}(Y'')) \end{aligned}$$

The original paper stacks  $N = 6$  decoder blocks, matching the encoder depth.

### 6.2 How Encoder and Decoder Interact

At each decoder block, the cross attention step lets the decoder look at the full encoder output  $Z$  without any masking. The decoder can attend to any encoder position freely; the causal mask only applies to the decoder’s self-attention over its own generated output.

This is what allows the model to condition on the entire source sequence while generating the target sequence one token at a time.

## 7 The Generative Pretrained Transformer

### 7.1 Architecture Overview

The original Transformer was an encoder-decoder model designed for translation. GPT cuts the encoder out entirely, making it a **decoder-only** model. The key architectural change this introduces is **causal self-attention** instead of full bidirectional self-attention. Since there is no encoder, cross attention is dropped from the decoder block as well.

### 7.2 Causal (Masked) Self-Attention

#### 7.2.1 Motivation

GPT is trained to predict the next token given all previous tokens. For this to work, when predicting token at position  $i$ , the model must not be allowed to look at positions  $j > i$ . If it could, the task would be trivial; it would just copy the answer from future tokens.

#### 7.2.2 Causal Masking

We enforce this with a **causal mask**, an upper-triangular matrix of  $-\infty$  values added to the attention scores before softmax:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V$$

where

$$M_{ij} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

Since  $e^{-\infty} = 0$ , any future position gets zeroed out in the softmax. Each token can only attend to itself and tokens before it.

### 7.3 GPT Decoder Block

Since there is no encoder to cross-attend to, the GPT decoder block is just masked MHA and FFN with residual connections (pre-norm). Formally, for input  $X$ :

$$\begin{aligned} X' &= X + \text{MaskedMHA}(\text{LayerNorm}(X)) \\ X'' &= X' + \text{FFN}(\text{LayerNorm}(X')) \end{aligned}$$

GPT stacks  $N$  of these blocks. GPT-1 used  $N = 12$ ; models have only gotten deeper since.

## 7.4 Pre-Training

### 7.4.1 Objective

GPT is pre-trained on a massive text corpus using the **language modeling objective**: predict the next token given all preceding tokens. Given a sequence  $[t_1, t_2, \dots, t_n]$ , we maximize:

$$\mathcal{L} = \sum_{i=1}^n \log P(t_i \mid t_1, \dots, t_{i-1})$$

In practice, this means minimizing cross-entropy loss between predicted and ground-truth token distributions.

This is **self-supervised learning**: the labels come from the data itself, so you need no human annotation. Just throw text at it.

### 7.4.2 The Language Modeling Head

After the final transformer block, we turn our token representations back into a probability distribution over the vocabulary. We do this with an **unembedding layer**:

$$\text{logits} = X_{\text{final}} W_E^T \in \mathbb{R}^{n \times V}$$

where  $W_E \in \mathbb{R}^{V \times d}$  is our embedding matrix and  $V$  is the vocabulary size. We apply softmax to get the final distribution over the next token.

**Note: Weight tying**, which shares  $W_E$  between the input embedding layer and the output unembedding layer, is common practice. It reduces parameter count and empirically tends to improve performance.

## 7.5 Autoregressive Inference

At inference time, GPT generates text by repeatedly predicting the next token and appending it to the context. This loop is called **autoregressive generation**:

1. Feed in a prompt.
2. Get a probability distribution over the vocabulary for the next token position.
3. Sample a token from that distribution.
4. Append it to the input.
5. Repeat until hitting the end-of-sequence token or a max length.

### 7.5.1 Sampling Strategies

How you sample in step 3 has a big impact on output quality:

- **Greedy decoding:** always pick the highest-probability token. Fast, but tends to produce repetitive output.
- **Temperature sampling:** divide logits by a scalar  $T$  before softmax. Lower  $T$  makes the distribution sharper (more confident); higher  $T$  flattens it (more random).

$$P(t_i) = \frac{\exp(\text{logit}_i/T)}{\sum_j \exp(\text{logit}_j/T)}$$

- **Top-k sampling:** sample only from the  $k$  most probable tokens, re-normalizing the distribution over them.
- **Top-p (nucleus) sampling:** sample from the smallest set of tokens whose cumulative probability exceeds threshold  $p$ . More adaptive than top-k since the number of candidates shrinks when the model is confident.

In practice, temperature and top-p are usually combined.

## 7.6 KV Cache

### 7.6.1 Motivation

Look at the autoregressive loop again. At step  $t$ , we feed in the full sequence  $[t_1, \dots, t_t]$  to get the next token. At step  $t + 1$ , we feed in  $[t_1, \dots, t_t, t_{t+1}]$ . The problem: we are recomputing  $K$  and  $V$  for  $t_1, \dots, t_t$  from scratch every single step. That's a lot of wasted work.

### 7.6.2 The Fix

Since the causal mask means that token  $i$ 's  $K$  and  $V$  vectors never change once computed (past tokens don't get updated by future ones), we can just cache them.

Concretely, after processing position  $i$  for the first time, we store:

$$\text{cache}[i] = (K_i, V_i)$$

On the next step, we only compute  $K$  and  $V$  for the new token, retrieve the rest from cache, and concatenate before running attention:

$$K = [\text{cache}[1], \dots, \text{cache}[t], K_{t+1}], \quad V = [\text{cache}[1], \dots, \text{cache}[t], V_{t+1}]$$

This turns attention at each generation step from  $O(n^2)$  to  $O(n)$  in compute, at the cost of  $O(n \cdot d)$  memory per layer.

**Note:** KV cache memory grows with every token generated. For long contexts or large batch sizes, this is the main bottleneck. GQA and MQA (covered in the attention chapter) exist largely to shrink this cost.

## 7.7 Fine-Tuning

Pre-training gives you a model with broad knowledge of language and the world. Fine-tuning adapts that model to specific tasks or behaviors by continuing training on a smaller, curated dataset. The pre-trained weights serve as a strong initialization, so you need far less data and compute than training from scratch.

Modern fine-tuning usually means one of:

- **Supervised fine-tuning (SFT)**: train on (prompt, ideal response) pairs.
- **RLHF** (Reinforcement Learning from Human Feedback): use human preference data to train a reward model, then optimize the policy against it.
- **Parameter-efficient fine-tuning (PEFT)**: methods like LoRA that freeze most weights and only update a small fraction, making fine-tuning far cheaper without much loss in quality.

## 7.8 Conclusion

At this point you should have a solid end-to-end picture of how a modern language model works. To recap the full pipeline:

1. **Tokenizer** splits raw text into subword tokens and maps them to integer IDs.
2. **Embedding layer** looks up a learned vector for each token ID.
3. **Positional encoding** (sinusoidal or RoPE) bakes position information into those vectors.
4. **Transformer blocks** (encoder, decoder, or GPT-style decoder-only) repeatedly refine the representations through masked/full self-attention, optional cross attention, FFN layers, and residual connections with Layer-Norm.
5. **Language modeling head** unembeds the final representations back into a vocabulary distribution.
6. **Autoregressive generation** samples one token at a time, caching KV pairs to avoid redundant computation.

Everything else in the modern LLM landscape (scaling laws, instruction tuning, RLHF, mixture of experts, multimodality) builds on top of this foundation. Understanding it deeply is what separates people who use these models from people who build them.