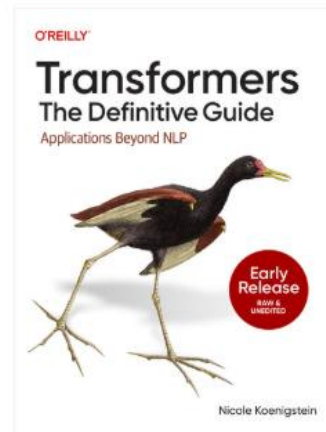


# Transformers: The Definitive Guide

[Provide feedback](#)

By [Nicole Koenigstein](#)



TIME TO COMPLETE:  
6h 44m

LEVEL:  
Intermediate to advanced

SKILLS:  
[Transformers](#)

PUBLISHED BY:  
[O'Reilly Media, Inc.](#)

PUBLICATION DATE:  
April 2026

PRINT LENGTH:  
400 pages

[+ Add to playlist](#)

## Chapter 1. From First Principles to State-of-the-Art Transformers

### A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo is available at <https://github.com/Nicolepcx/transformers-the-definitive-guide>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [sgrey@oreilly.com](mailto:sgrey@oreilly.com).

Since its introduction in 2017, the transformer architecture<sup>1</sup> has revolutionized the field of natural language processing (NLP), marking a paradigm shift towards models capable of natural language understanding (NLU). This shift was possible because transformers process sequential data in parallel, enabling a deeper and more contextual understanding of language than was achievable with previous sequential models, like long short-term memory (LSTM) networks.

In recent years, transformers have evolved to impact a wide array of domains, including computer vision, speech recognition, reinforcement learning, and mathematical operations, moving beyond their initial usage within NLP. Their adaptability has led to significant advancements in machine translation, allowing for context-aware translations, and in scientific research, notably in predicting protein structures with remarkable accuracy.

Among the most exciting developments are reasoning models, which are advanced LLMs trained with reinforcement learning to perform complex, multi-step reasoning. They generate internal chains of thought before answering, which is inspired by the human

thought process, this technique first solves intermediate steps before getting to the final answer.

I assume in this book that you have at least some familiarity with the transformer architecture. Perhaps you've read the book [Natural Language Processing with Transformers](#), or a similar work. Moreover, I take it you're not just curious about transformers. You're here because you want to build real applications with transformers, and you want to do it right.

This chapter provides a focused review of the transformer architecture to set the stage for the more advanced and complex models beyond NLP that I will cover in later chapters.

I will begin with the basic transformer architecture, then explain how longer context becomes possible, and finish with a tour of various attention mechanisms. Throughout this chapter, and the ones that follow, I will share practical insights from real deployments so that you can benefit from my experience and learn the patterns, pitfalls, and principles that matter when theory meets the hard surface of production.

## Transformer basics

This section explains the main architectural components of the original transformer model, such as encoder and decoder, positional embeddings, and attention mechanism.

The transformer architecture was originally developed for machine translation, a challenging sequence-to-sequence task in which the concept of tokenization plays a critical role. *Tokenization* breaks down sequences-like sentences into manageable units, or tokens, that the transformer can effectively process. For example, in the sentence:

The Transformer has revolutionized NLP.

the word *the* represents a single word-level token.

Before we dive into the architectural components, understanding tokenization is crucial, as it facilitates the transformer's ability to interpret text. And sets the foundation for its application to other sequences.

### Tokenizer: Text representation in the transformer

A *tokenizer* is used to tokenize the text. This is the first step to make natural language digestible for the model, before applying token embeddings and finally positional embeddings. The different types of tokenization are:

#### *Character-level tokenization*

*Character-level tokenization* splits the underlying alphabet into each existing character in the sequence. If you used character-level tokenization for

"The Transformer has revolutionized NLP."

it would yield:

[T, h, e, ' , ... 'N, L, P, .]

This will lead to very long sequences, which can increase computational complexity. It can also be challenging for the model to learn long-term dependencies. Nonetheless, this can be helpful if your task requires a fine-grained understanding.

### Word-level tokenization

Word-level tokenization would split the example sentence as follows:

[The, Transformer, ... NLP, ., ]

that is, the sequence will be split into its words, plus punctuation. The downside is that this requires a large vocabulary, and if the language changes, this tokenization will not be able to understand new words.

### Subword tokenization

Most modern LLMs use *subword tokenization*, in which the word is split into smaller parts. For instance, a subword tokenizer would split the word *hiking*, the tokenizer into:

[h, ik, ing]

and the word *cooking* into:

[cook, ing]

So subword tokenization splits a word (or sequence) into smaller, commonly occurring chunks, like

[ing]

Single-character words are also included.

Now that you understand the basics of tokenization, let's move on to token and positional embeddings.

### Token and positional embeddings

A part of the transformer architecture that contains learnable parameters is the token and positional embeddings (PE). The token embedding is tasked with encoding each vocabulary element into a  $V$ -dimensional vector in the space of  $V$ . The token embedding can mathematically be presented as follows:

Let  $V$  be the vocabulary with  $|V|$ , where each word  $w$  in  $V$  is assigned a unique token ID,  $id(w)$ . The token embedding is a function that maps each token ID to a  $V$ -dimensional vector. This is achieved through a token embedding matrix  $E$ , where  $V$  is the dimensionality of the embeddings. Here's how to do it using bidirectional encoder representations from transformers (BERT):

```
from transformers import AutoTokenizer, AutoModel
```

```
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased') ❶
```

```
model = AutoModel.from_pretrained('bert-base-uncased')
```

```
sentence = "The Transformer has revolutionized NLP."
```

```
inputs = tokenizer(sentence, return_tensors='pt') ❷
```

```
input_ids = inputs['input_ids'] ❸
```

```
print(input_ids)
```

```
outputs = model(input_ids)
```

```
embeddings = outputs.last_hidden_state ❷
```

```
print(embeddings)
```

❶

Load tokenizer and model from Hugging Face

❷

Tokenize the sentence

❸

Get the input IDs and pass them through the model to get the embeddings

❹

Get the last hidden state, to access the embeddings of the tokens

This will result in the following output for the input IDs of the sentence:

```
tensor([[ 101, 1996, 10938, 2121, 2038, 4329, 3550, 17953, 2361, 1012, 102]])
```

For the corresponding embeddings the output is:

```
tensor([[[[-0.5249, -0.2210,  0.2696, ..., -0.4204,  0.2605,  0.6457],
          [-0.6665, -0.4994,  0.4651, ..., -0.2517,  0.2334,  0.0176],
          [ 0.8416, -2.0561,  0.8323, ..., -0.2709, -0.1999, -0.1918],
          ...,
          [-0.4018, -0.6402,  0.7791, ..., -0.0290, -0.4070,  0.2974],
          [-0.3327, -0.8091, -0.0304, ...,  0.4745,  0.3230, -0.5991],
          [ 0.4928, -0.0878, -0.0971, ...,  0.1629, -0.7012, -0.3848]]]],
        grad_fn=<NativeLayerNormBackward0>)
```

This representation lacks the position of the word in the sequence. And since the transformer does not have *recurrence*, meaning that it doesn't need to process the data sequentially as it was originally represented, you need a function to represent the position. This is why you need to add positional embeddings: without them the model treats sequences as unordered collections of words.

The positional-embedding function learns to encode a token's location within a sequence into a vector in the space. The original Transformer uses for position :

Here is the element of the  $d$ -dimensional vector. This means that the position of the first token is captured by a vector,  $e_1$ , while the position of the second token is captured by a different learned vector  $e_2$ , and so on.

This technique enables transformer models to understand the order of words. In the next section you'll see how the transformer uses this vector representation to understand and learn from the text.

### Attention mechanism

The attention mechanism is at the core of the transformer's ability to understand and interpret text. It gives the model the ability to analyze the relevance of a word in a sequence on a token-to-token basis.

In that context, you will often hear the term *attribution matrix*, which is computed from the input embeddings. Here the term *attribution* refers to the significance between different parts of the input. The attribution matrix is computed with the (query) and the (key) matrices. The resulting scores form the and interaction to determine the attention weights, which are then applied to the (Value) matrix to produce the output of the attention mechanism:

This attribution matrix is crucial for understanding how the model interprets and processes the corresponding input sequences. For instance, by analyzing these scores, you can gain insights into the model's decision-making process, such as which tokens it sees as more relevant than others when generating the output token. Libraries such as [Captum](#) help make this decision-making process visible.

However, despite the specific roles of , and , the initial computation for each of these matrices follows a similar process: a *linear projection* of the input embeddings. This means that for each of these matrices, the input embeddings are multiplied by a weight matrix. This process can be mathematically described as follows:

- Query matrix Q:
- Key matrix K:
- Value matrix V:

Here represents the input embeddings, and , and are the weight matrices for the query, key and value projections, respectively. Take the dot product of the query and key matrices, followed by the softmax function and the scaling factor (for scaled dot attention). The result will be a matrix of scores representing self-attention, or how much focus each token should put on each other token by considering its relationship with every other element in the sequence. These scores are then used to weight the values in the matrix, producing the final weighted-sum output of the attention mechanism:

This dynamic process allows the model to focus on different parts of the input sequences for each input token, making it possible to understand each token's contextual relevance and information.

### Multi-head attention

The attention mechanism you've seen so far represents the computation performed by a single attention head, which is the component responsible for calculating attention in the transformer. However, the original transformer, as well as state-of-the-art (SOTA) models apply multiple attention heads simultaneously. Each individual attention head has its own learnable parameters, which are then combined into a single output. This allows the model

to integrate information from the same sequence and capture a variety of relationships between its words or elements. This approach enhances the model's ability to understand and represent complex dependencies in the data.

In technical terms, given input sequences  $X$ ,  $Y$ , and  $Z$ , the multi-head attention mechanism computes new representations for the elements in  $X$  by considering information from  $Y$ ,  $Z$ , and so on. This process involves several steps: each head computes its own attention scores and output vectors based on the input, then concatenates and linearly transforms these outputs to produce the final output vector  $X'$ .

This process consolidates the contextual information captured by the individual attention heads into one unified output that encapsulates all critical information across the entire input sequence. Since each attention head might focus on different relationships within the input sequence, this is crucial to the model gaining a better language understanding.

### **Bidirectional and unidirectional attention**

As I mentioned, the first transformer model was used for machine translation. That's why it uses two distinct types of attention mechanism within the architecture: one for the encoder, and another for the decoder.

First, the encoder applies bidirectional self-attention, not just left-to-right processing, as traditional sequence processing methods do. This means it treats all tokens as context, applying attention to each token in the sequence. This gives the model a full understanding of the entire input sequence when it generates representations for each token.

The decoder's attention is masked (also called *causal attention*) to prevent the model from attending to future tokens (subsequent positions). In practice, this means that for the prediction  $y_t$ , the model can only attend to the position  $t$ . With that method in place, the model generates each token based only on the tokens previously created, from left to right, thus preventing it from using future tokens in the sequence. This is important for all task where the model must generate one token at a time as, for instance, for translation.

Now that you understand the two distinct variations of attention used with the first transformer, let's look at the encoder and decoder.

### **Encoder and decoder parts**

The first transformer model's architecture ([Figure 1-1](#)) was characterized by its encoder-decoder structure. Some subsequent models leverage a decoder-only framework, such as GPT, LLaMA, Mistral, and Falcon.

The encoder itself is composed of six identical layers, each containing two principal components: a multi-head self-attention mechanism and a point-wise fully connected feed-forward network. The term *point-wise* refers to applying the same linear transformation to each sequence element. These components are further refined with residual connections and layer normalization.

The decoder interprets the encoded information, mirroring the encoder's layered structure but introduces an essential feature: *masked multi-head self-attention*. This added feature in the decoder prevents the model from accessing subsequent positions in the sequence.

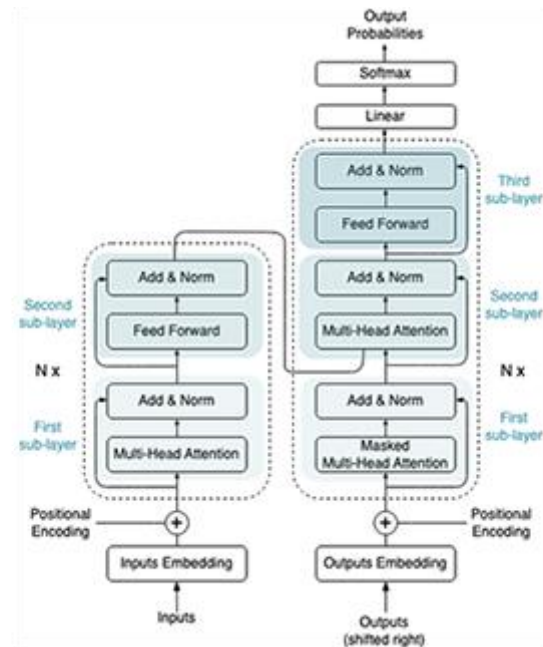


Figure 1-1. Encoder and decoder part of the Transformer architecture.

The model maintains a consistent output dimension of 512 across all sub-layers, including the embedding layers, meaning its maximum sequence length is 512 tokens. This limitation comes mostly from the specific architectural setup of the first transformer model, which made it hard to process longer sequences on the available hardware efficiently.

Enhancements in transformer design: Longer context and attention variations

Now it's time to look into methods by which modern transformer models, like GPT-4.5 and Qwen3, achieve higher levels of performance and flexibility. In particular the ability to process more information at once, through longer context windows. Attention-mechanism variations such as multi-query and flash attention also increase the efficiency and accuracy of SOTA transformer models.

### Longer context windows with better performance

A model's *context window* refers to the portion of text it can process when making predictions or generating text. A longer context window allows the model to understand more complex narratives and capture nuances better than it could using a chunked version of a text with a small context window.

However, simply extending the context length results in quadratic increases in time complexity and memory usage, which can constrain improvements. Therefore, recent enhancements, such as *rotary positional embedding* (RoPE)<sup>2</sup>, *position interpolation* (PI)<sup>3</sup> and *Yet another RoPE extension method* (YaRN)<sup>4</sup>, are designed to more effectively manage longer contexts during inference.

RoPE brings *absolute* and *relative* PEs together. But before I dive deeper into how RoPE works, let's first look at the key differences between absolute and relative PEs.

- With absolute PEs, for each token embedding, the model adds information about the absolute position of the token. Absolute PEs are simpler and faster to compute.

- Relative PEs consider distances between sequence elements and can be shared across sequences, which helps the model to understand and interpret the relationships and distances between different tokens within a sequence. Relative PEs result in an increase in performance, but are computationally more complex.

RoPE combines absolute and relative positional embeddings, representing a significant advancement in the design of transformer models. These models process longer sequences of text more naturally and accurately, while maintaining efficiency.

Specifically, RoPE integrates a rotation matrix,  $R$ , to encode the absolute positions of tokens, incorporating the explicit dependency of relative positions into the self-attention mechanism. To illustrate RoPE's implementation more concretely, consider a model with dimension  $d$ , which then can be computed as follows:

Higher dimensions are divided into subspaces, so the dimension number has to be even. Let's put the math into code to make the theoretical concept more clear:

```
def simple_rotary_matrix(d, m, max_len):
    assert d % 2 == 0, "Embedding dimension must be even."❶

    theta = 10000 ** (-2 * torch.arange(d // 2).float() / d)❷
    theta *= m

    cos_theta = torch.cos(theta)❸
    sin_theta = torch.sin(theta)

    R = torch.zeros((d, d))❹

    R[torch.arange(0, d, 2), torch.arange(0, d, 2)] = cos_theta❺
    R[torch.arange(0, d, 2), torch.arange(1, d, 2)] = -sin_theta
    R[torch.arange(1, d, 2), torch.arange(0, d, 2)] = sin_theta
    R[torch.arange(1, d, 2), torch.arange(1, d, 2)] = cos_theta

    return R
```

❶

To ensure the dimension  $d$  is even, since this is required.

❷

Compute thetas

③

Compute sin and cos for rotation

④

Initialize the rotation matrix

⑤

Compute the rotation matrix

To use the function, you can simply do the following:

d = 6 ①

max\_len = 10 ②

R\_matrix = simple\_rotary\_matrix(d, m=1, max\_len=max\_len) ③

print(R\_matrix)

①

Embedding dimension

②

Sequence length

③

Creates the rotation matrix

This creates the following rotary matrix:

```
tensor([[ 0.5403, -0.8415,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.8415,  0.5403,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.9989, -0.0464,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0464,  0.9989,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  1.0000, -0.0022],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0022,  1.0000]])
```

[Figure 1-2](#) shows an illustrates the RoPE process.

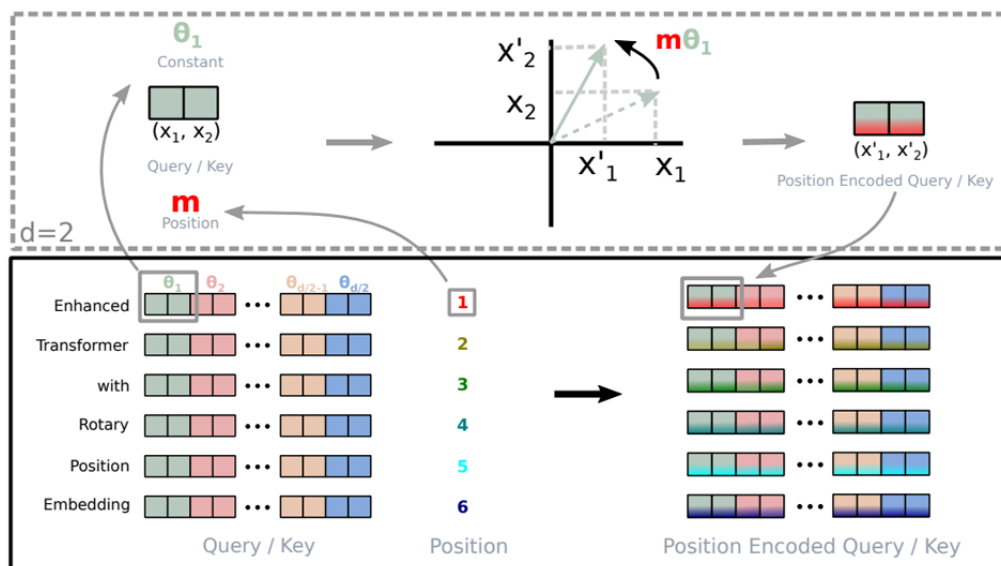


Figure 1-2. Illustration of Rotary Position Embedding(RoPE). Image adapted from: Jianlin Su et al.

To apply RoPE in the context of self-attention, define the relationship between the in position and key in position as:

Here  $m$  represents the rotary matrix adapting the relative positions.

RoPE enhances efficiency and accuracy, so it's used in SOTA models like Qwen3. Even SOTA LLMs have a maximum number of tokens they can process at once. For instance, the Qwen3 models can handle up to 32,768 tokens in a single input.

This limitation becomes a problem in use cases that involve long prompts or extensive document summaries, where LLMs capable of managing more extensive contexts are desirable. However, it would take substantial computational resources to create a new LLM with an expanded context capability from the ground up. This raises an important question: Is it possible to increase the context window size of an already pre-trained LLM? The good news is: yes! PI and YaRN can extend these pre-trained LLMs with minimal fine-tuning. [Figure 1-3](#) demonstrates the PI technique for a LLaMA model with a 2048 context window.

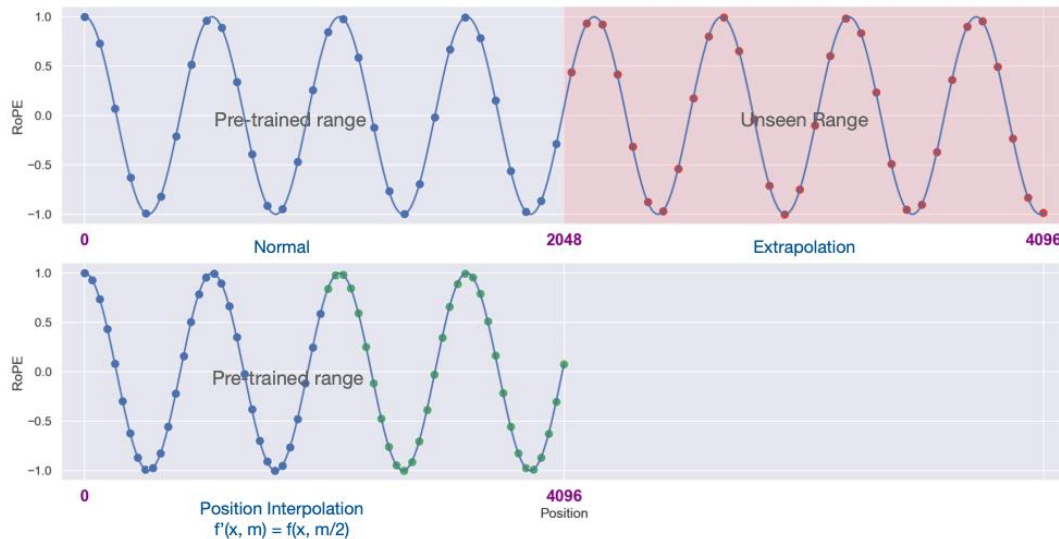


Figure 1-3. How the position interpolation (PI) method works for a LLaMA model with a 2048 context window. The blue dots stand for the training limit of LLMs; the red squares illustrate how models adapt to new positions. The blue dots and the green triangles demonstrate how PI scales down from [0, 4096] to [0, 2048] to keep them within the trained range.

Normally, LLM models use input positions (blue dots) within their trained range. For length extrapolation, models handle new positions (red squares) up to 4096. Position interpolation downscales these indices (blue dots and the green triangles) from [0, 4096] to [0, 2048], ensuring they stay within the pretrained range.

To extend the context window, PI interpolates the position indices within the pre-trained limit, with a small set of fine-tuning applied.

That is, PI extends RoPEs function by as follows:

Here is a new context window beyond the pre-trained one.

Let me take a short step back and explain an important way to evaluate the performance of a model - *perplexity* (PPL). This is a measure how “surprised” or “perplexed” a model is about context. That is, perplexity measures on how well a probability model predicts a sample, with lower values indicating better predictive accuracy. Let me illustrate this with a concrete coding example:

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained("tiiuae/falcon-7b")

tokenizer = AutoTokenizer.from_pretrained("tiiuae/falcon-7b")

wiki_text = tokenizer("Apple Inc. is an American multinational " +
    "corporation and technology company headquartered " +
    "in Cupertino, California, in Silicon Valley. ",
    return_tensors = "pt")
```

```
loss = model(input_ids = wiki_text["input_ids"],
              labels = wiki_text["input_ids"]).loss
ppl = torch.exp(loss)
print(ppl)
```

```
input_text = tokenizer("A Falcon is a generative transformer "+
                       "model and it can't fly.", return_tensors = "pt")
```

```
loss = model(input_ids = input_text["input_ids"],
              labels = input_text["input_ids"]).loss ❶
ppl = torch.exp(loss)
print(ppl)
```

❶

Compute loss

The wiki\_text

input yields a score of 5.08, while the

input\_text

yields 121.19. This significantly higher perplexity score indicates that the model finds this sentence quite surprising or unlikely. This is because the model was most likely just trained on data indicating that a falcon is a bird known for its remarkable flying abilities, not a transformer model.

For evaluating LLM performance with longer context windows, you will use *sliding window perplexity*. This metric calculates perplexity over a fixed-size window of tokens, moving across the text, to better handle and evaluate large texts and datasets.

One downside of RoPE is that it expands token positional information into a multidimensional complex vector. It struggles with encoding high-frequency components, because its one-dimensional input limits its ability to distinguish between very similar and proximate tokens.

### Softmax and the haystack problem

The attention distributions in transformers are computed using the Softmax function. As the context window grows, Softmax tends to produce flatter distributions. This happens because the denominator (the sum of exponentials across all tokens) increases with context size, while each numerator (the exponential of a token's score) remains fixed. As a result, the output probabilities shrink, and the model struggles to focus on important tokens.

This is often referred to as the haystack problem: relevant signals get diluted among many irrelevant ones. Even with advanced techniques like RoPE, the model’s ability to prioritize key elements across long contexts weakens. To address this, SOTA models like LLaMA 4 apply post-training optimization on long contexts, use inference-time temperature scaling of attention<sup>5</sup>, and introduce architectural changes such as interleaved attention layers without positional embeddings (iRoPE)<sup>6</sup>. The usage of these methods increases the supported context to up to 10 million tokens, while still performing well on “retrieval the needle in the haystack”.

To address this, practitioners of *neural tangent kernel* (NTK)<sup>7</sup> theory developed, *NTK-aware interpolation*, adjusting the scaling of frequencies differently across dimensions to preserve high-frequency information. One of the applications of NTK theory is identifying and mitigating issues related to training neural networks, such as difficulties in learning high-frequency components or patterns in data with low *intrinsic dimensionality*, as is the case with RoPE. Intrinsic dimensionality refers to the minimum number of parameters needed to accurately describe a dataset without losing significant information, representing the dataset’s inherent complexity.

However, NTK-aware interpolation can stretch some dimensions beyond their bounds, potentially degrading the model’s performance. Additionally, *NTK-by-parts interpolation* and *dynamic NTK interpolation* were introduced as refined strategies, focusing on preserving relative local distances and adapting scale factors dynamically for varying sequence lengths, respectively.

Building upon these NTK-techniques, YaRN introduces a temperature to the attention scores before the attention softmax, uniformly affecting perplexity across different data samples and token positions. This approach modifies attention weight computation and utilizes a length-scaling technique that adjusts both  $\log$  and  $\exp$  by a constant factor, enhancing the attention mechanism without altering its underlying code. RoPE embeddings, pre-generated and reused, facilitate this process with no additional computational cost during inference or training. When combined with *NTK-by-parts interpolation*, YaRN performs effectively in models like LLaMA and LLaMA 2 see [Figure 1-4](#).

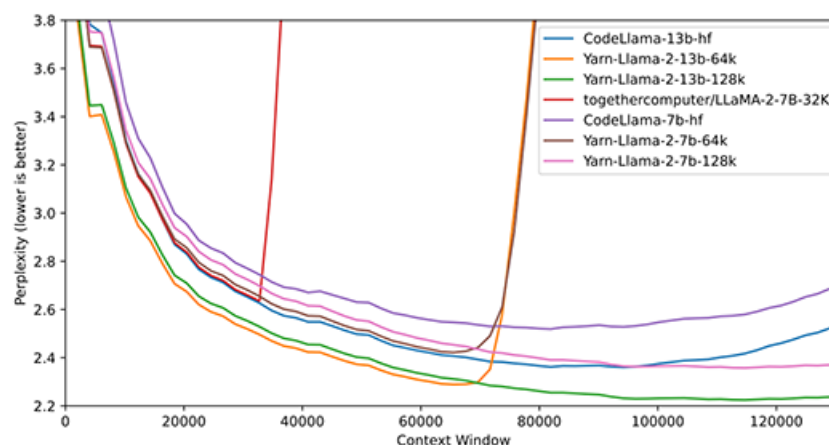


Figure 1-4. How the context window can affect the perplexity.

As you have seen the lower the perplexity score, the better the model performs. For instance, LLaMA 7b with YaRN and 128k extrapolation performs well in comparison to LLaMA 7b without YaRN.

I'm sure you now would love to know how you can actually apply techniques like RoPE or YaRN to enhance the context length, to ensure optimal performance on lengthy texts. The great news is that most frameworks allow for easy activation of longer context windows, for instance, vLLM supports YaRN, which can be configured as:

```
vllm serve Qwen3/Qwen3-8B --rope-scaling {"rope_type": "yarn", "factor": 4.0, "original_max_position_embeddings": 32768} --max-model-len 131072
```

Next, let's move to different attention variations and how they improve the performance.

### Attention mechanism variations

Today's transformers are more efficient than previous models, like LSTMs. That is, the first transformer model achieved a similar high BLEU score as LSTMs, which needed to be trained for months, after only 3.5 days of training. However, transformers can still be considered memory-hungry, since the time and memory complexity of self-attention grows quadratically with the sequence length. This section explores various improvements on the attention mechanisms used in high-performing SOTA LLMs including:

- Cross attention<sup>[8](#)</sup>
- Multi-query attention (MQA)<sup>[9](#)</sup>
- Grouped-query attention (GQA)<sup>[10](#)</sup>
- FlashAttention<sup>[11](#)</sup>
- FlashAttention-2<sup>[12](#)</sup>
- FlashAttention-3<sup>[13](#)</sup>

It is common for models to combine different attention variations: for instance, Falcon uses multi-query attention and FlashAttention.

### Cross-Attention

In *cross-attention* the inputs from two different sequences are combined. Usually this means that the queries come from the decoder and the keys and the values come from the encoder. So, in essence, cross-attention enables the interaction between a set of embeddings. This is important for applications where you want to attend to a source sequence while generating a target sequence, such as translation or question-answering tasks. Let me explain the concept further with code.

```
def CrossAttention(x_1, x_2, W_query, W_key, W_value):
```

```
    scaling_factor = W_query.shape[1]**0.5
```

```

Q = torch.einsum('bd,dk->bk', x_1, W_query)
K = torch.einsum('bd,dk->bk', x_2, W_key)
V = torch.einsum('bd,dv->bv', x_2, W_value)

attn_scores = torch.einsum('bk,mk->bm', Q, K)
attn_weights = F.softmax(attn_scores / scaling_factor, dim=-1)

Y = torch.einsum('bm,mv->bv', attn_weights, V)

return Y

```

In this code, you can see that the input for `Q` comes from `x_1` and for `K` and `V` from `x_2`, demonstrating the information flow between sequences. Using multiple information sources the LLM gets a more sophisticated understanding and better generation results.

### Multi-Query Attention

*Multi-query attention* (MQA) uses only a single key-value head, whereas multi-head attention (MHA) uses -number of heads for query, key and value heads, respectively. Thus, MQA significantly speeds up the decoder's inference time. [Figure 1-5](#) compares the two.

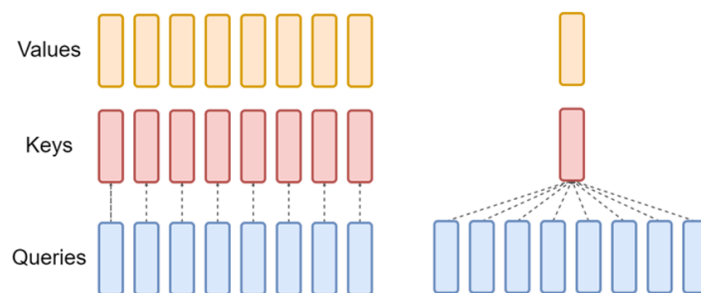


Figure 1-5. Comparison of multi-head attention (left) and multi-query attention (right). Where multi-head attention has number of query, key and value heads, multi-query shares a single key and value head across all query heads.

To make this difference more tangible, read the following code which computes MHA. Note that there is a letter `h` for each `h`, and `h` to represent the head's dimension.

```

import torch

import torch.nn.functional as F

def MultiheadAttention(x, M, W_query, W_key, W_value, P_o):

    scaling_factor = W_key.shape[1]**0.5

```

❶

```
Q = torch.einsum('d,hdk->hk', x, W_query)
K = torch.einsum('md,hdk->hmk', M, W_key)
V = torch.einsum('md,hdv->hmv', M, W_value)
```

❷

```
attn_scores = torch.einsum('hk,hmk->hm', Q, K) / scaling_factor
```

❸

```
attn_weights = F.softmax(attn_scores, dim=-1)
```

❹

```
o = torch.einsum('hm,hmv->hv', attn_weights, V)
y = torch.einsum('hv,hdv->d', o, P_o)
```

```
return y
```

❶

Weight matrices

❷

Compute attribution matrices using the scaling factor for scaled dot-product attention

❸

Apply softmax to attention scores

❹

Compute final attention weights (context vectors)

With MQA, the letter `i` is omitted from the `Q` and `K` matrices:

```
def MultiqueryAttention(X, M, mask, W_query, W_key, W_value, P_o):
```

```
    scaling_factor = W_key.shape[1]**0.5
```

```
    Q = torch.einsum('bnd,hdk->bhnk', X, W_query)
```

```
    K = torch.einsum('bmd,dk->bmk', M, W_key)
```

```
V = torch.einsum('bmd,dv->bm', M, W_value)
```

```
attn_scores = torch.einsum('bhnk,bmk->bhn', Q, K) / scaling_factor
```

```
attn_weights = F.softmax(attn_scores + mask, dim=-1)
```

```
O = torch.einsum('bhn,bmv->bhn', attn_weights, V)
```

```
Y = torch.einsum('bhn,hdv->bnd', O, P_o)
```

```
return Y
```

These two examples make it clear that MQA is identical to MHA, except that in MQA the different heads share a single set of keys and values. This modification speeds up computation in the decoder, but can lead to loss of quality, though still more performant than MHA. GQA was developed to address this.

### Grouped-Query Attention

*Grouped-query attention* (GQA) organizes query heads into number of groups, with each group sharing one key and one value head. [Figure 1-6](#) compares multi-head attention (left) and grouped-query attention (right).

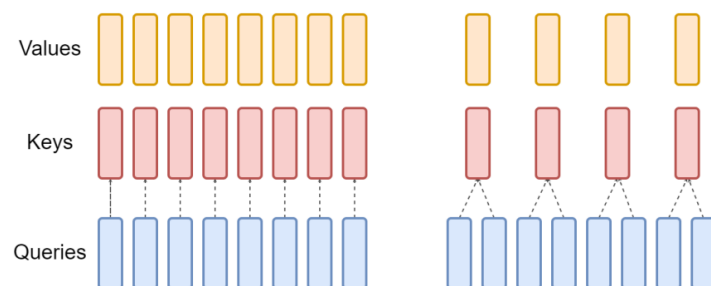


Figure 1-6. Comparison of multi-head attention (left) and grouped-query attention (right). Where multi-head attention has number of query, key and value heads grouped-query attention instead shares one key and value head for each group of query heads, interpolating between multi-head and multi-query attention.

Comparing MHA to GQA, you can see that GQA consolidates multiple key and value heads into a single key and value head, effectively reducing the key-value (KV) size.

### KV caching

*KV caching* optimizes *inference latency* by storing the computed key and value tensors for previously generated tokens during autoregressive decoding. That is, instead of recalculating the full attention context at every step, the model appends only the new keys and values, which significantly reduces the computational cost of the attention mechanism. However, although KV caching provides substantial improvements in inference speed, it increases memory usage proportionally with the sequence length and the number of layers. In scenarios where memory is a limiting factor, you may have to

reduce the model size or limit the context window, which can lead to a drop in model accuracy. Deploying KV caching in large-scale production systems also introduces complexity in managing the cache lifecycle. This includes implementing strategies for cache eviction, dynamic memory allocation, and evaluating strategies for cache reuse across requests or sessions.

This means significantly lesser data to load into memory during computation, decreasing the required bandwidth and capacity by a factor of . The following code illustrates this setup:

```
def GroupedQueryAttention(Q, K, V, num_heads, group_size):

    batch_size, seq_len, embed_dim = Q.shape
    scaling_factor = (embed_dim // num_heads) ** 0.5

    Q = rearrange(Q, 'b s (h d) -> (b h) s d', h=num_heads)
    K = rearrange(K, 'b s (h d) -> (b h) s d', h=num_heads)
    V = rearrange(V, 'b s (h d) -> (b h) s d', h=num_heads)

    attn_scores = torch.einsum('bid,bjd->bij', Q, K) / scaling_factor
    attn_weights = F.softmax(attn_scores, dim=-1)
    attn_output = torch.einsum('bij,bjd->bid', attn_weights, V)

    Y = rearrange(attn_output, '(b h) s d -> b s (h d)', b=batch_size, h=num_heads)

    return Y
```

GQA is specifically beneficial for larger models as they usually expand the number of heads. That said, employing GQA substantially reduces both memory bandwidth and capacity, while maintaining performance as models scale up.

Thus, memory bandwidth overhead from attention has less impact in larger models. This is because the key-value cache size increases linearly with the model dimension, whereas the model's floating-point operations per second (FLOPs) and parameters increase quadratically with the model dimension.

Even given these improvements, there is still room to optimize how attention leverages the GPU memory. This is where FlashAttention and FlashAttention-2 come in.

## FlashAttention

*FlashAttention* uses *tiling* to rearrange how attention calculations are performed. By doing so, it avoids creating a attention matrix. Tiling involves transferring chunks of input data

from GPU high bandwidth memory (HBM) and GPU on-chip SRAM (speedy cache). FlashAttention iterates over sections of the  $Q$  and  $V$  matrices, transferring them to the “speedy cache”. Within each section, it cycles through portions of the  $K$  matrix, moving them to SRAM, then saves the results of the attention process back to the HBM (illustrated in Figure 1-7).

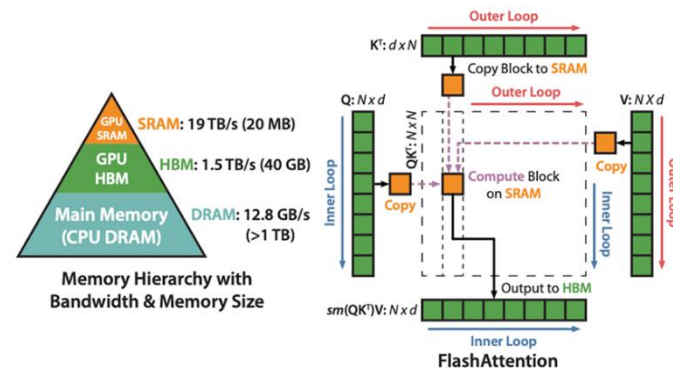


Figure 1-7. FlashAttention uses tiling to eliminate the large attention matrix. It works by cycling through segments of the  $Q$  and  $V$  matrices in its outer loop (indicated with red arrows), loading these segments into the fast on-chip SRAM. For each segment, FlashAttention also processes chunks of the  $K$  matrix (denoted by blue arrows), loading them into SRAM, then saving the attention output back to HBM.

This enhances computation speed while decreasing memory consumption from quadratic to linear, relative to the sequence length. FlashAttention avoids saving the large intermediate attention matrices in HBM, minimizing memory operations and doubling or even quadrupling processing speed. In addition, FlashAttention enables longer context windows in transformers, resulting in better perplexity scores and therefore higher quality models.

This is impressive, but there is still room for more improvement. The number of non-matmul FLOPs operations can be further reduced, as you will see in the next section.

## FlashAttention-2

I mentioned earlier that it’s difficult to increase context window size in transformers. The core attention layer’s runtime and memory demands grow quadratically with the input sequence length. RoPE, PI, and YaRN help improve efficiency and lower the perplexity, as you saw.

FlashAttention-2 reduces the amount of non-matmul FLOPs while not changing the output. Although these non-matrix multiplication FLOPs amount to only a minor portion of the total FLOPs, they are slower to execute. GPUs have specialized units that make matrix multiplication operations run up to 16 times faster than non-matrix multiplication operations. Therefore, minimizing non-matrix multiplication FLOPs and maximizing the time spent on matrix multiplication FLOPs is crucial for speeding up your computations.

FlashAttention-2 achieves this by optimizing GPU resource utilization. It minimizes shared memory access through parallel computation across different thread blocks and work partitioning among warps within a single thread block. A *warp* is a group of threads that execute computations. These adjustments contribute to a 2-3 times speedup.

This approach involves inverting the *loop hierarchy*, focusing first on row segments in the outer loop and column segments in the inner loop. This reverses the original method presented in the FlashAttention and introduces parallel processing along the sequence length dimension. [Figure 1-8](#) illustrates this.

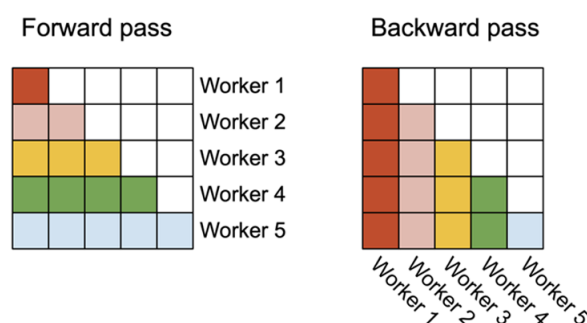


Figure 1-8. In the forward pass (left), the tasks (thread blocks) are distributed in parallel, with each task handling a segment of rows from the attention matrix. In the backward pass (right), each task is responsible for a segment of columns within the attention matrix.

[Figure 1-9](#) compares the work partitioning between different warps in the forward pass in FlashAttention and FlashAttention-2. Efficiently dividing work among warps can significantly impact the performance of parallel computing tasks, including those in deep learning models like transformers.

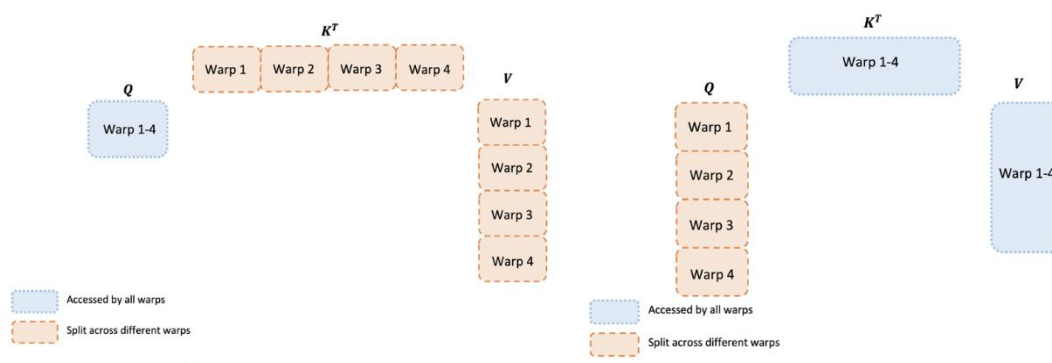


Figure 1-9. Comparison of work partitioning between different warps in the forward pass in FlashAttention (left) and FlashAttention-2 (right).

### FlashAttention-3

*FlashAttention-3* introduces new programming techniques that take full advantage of the Hopper<sup>14</sup> GPU architecture, specifically the NVIDIA H100, to accelerate attention computation beyond the limits of previous methods. While FlashAttention-2 performs well for most GPUs, on newer architectures such as H100, FlashAttention-2 achieves only 35% GPU utilization.

While FlashAttention and FlashAttention-2 focused on reducing memory bandwidth usage and optimizing compute schedules, FlashAttention-3 advances performance by leveraging hardware asynchrony and low-precision formats such as FP8. One of its key innovations is the use of producer-consumer asynchrony, where separate GPU warps are assigned distinct roles: some act as producers loading data (Q, K, V) via the Tensor Memory Accelerator (TMA), while others act as consumers performing matrix multiplications on

Tensor Cores. This strategy, often referred to as “pingpong scheduling,” allows data transfer and computation to run concurrently, effectively hiding latency and maximizing throughput.

### **PagedAttention for higher throughput**

While FlashAttention-3 introduces techniques that fully leverage the Hopper architecture and low-precision formats like FP8, it’s optimized for H100 GPUs only. But these GPUs can be expensive to run on cloud services. Therefore, for most teams and production environments, PagedAttention offers a more accessible and cost-effective solution to increase inference throughput without needing specialized hardware. *PagedAttention* is a memory-efficient attention variant designed to improve throughput during LLM inference. I’m sure you’ve read my note on KV caching earlier in this chapter and that you might have to evaluate strategies to optimize KV caching. This is exactly what PagedAttention<sup>15</sup> does. PagedAttention stores the KV cache in non-contiguous memory blocks, similar to virtual memory paging in operating systems. These blocks can be dynamically allocated, shared across sequences, and reused with copy-on-write semantics.

PagedAttention is built into the [vLLM serving system](#) and achieves up to 4× higher throughput by minimizing KV cache waste and enabling batching of more requests. PagedAttention is especially beneficial for workloads with long sequences, variable decoding lengths, and complex algorithms like beam search or parallel sampling. Note that PagedAttention is only available on vllm. Moreover, vllm can struggle if you have a lot of concurrent requests, so your throughput could still be better with [Hugging Face’s Text Generation Inference](#) (TGI), as it’s very reliable on many concurrent requests. I suggest that you use a [TGI benchmarking tool](#) to validate this for your application.

Another innovation is GEMM-softmax pipelining. *General matrix-matrix multiplication* (GEMM), is a fundamental operation in deep learning that multiplies two matrices to produce a third, and is heavily optimized on GPUs using specialized hardware like Tensor Cores. In transformers, the softmax operation depends on the output of GEMM, introducing a sequential dependency. FlashAttention-3 breaks this bottleneck by pipelining GEMM and softmax across iterations, so that while one block performs softmax, the next GEMM operation can already begin. This overlapping is essential to exploit Hopper’s asynchronous compute capabilities.

FlashAttention-3 also introduces low-precision attention with FP8, which nearly doubles throughput compared to FP16. To achieve this without sacrificing accuracy, it adapts the memory layout of Q, K, and V to meet Hopper’s FP8 GEMM constraints and applies two techniques to reduce quantization error: block quantization and incoherent processing. The latter involves multiplying Q and K with a random orthogonal matrix constructed from Hadamard transforms before quantization. *Hadamard transforms* refer to a mathematical operation that maps a vector into a new space using only additions and subtractions. It relies on the *Hadamard matrix*, which is made up entirely of +1 and -1 entries. This transformation is efficient to compute and helps spread information across dimensions, which is useful for reducing the impact of outliers in low-precision quantization.

### **Conclusion**

This chapter has taken you from the foundational ideas of the original transformer to some of the most powerful architectural and inference-time innovations that define today’s SOTA models. From tokenization and multi-head attention to rotary embeddings, longer context

windows, and advanced memory optimizations like PagedAttention and FlashAttention, you've seen how the architecture has evolved over time to meet the ever-growing demands of real-world applications.

This progression is a testament to the fact that the transformer is no longer a static blueprint confined to language tasks. It is a dynamic and extensible framework that continues to improve in both accuracy and efficiency. In the next chapters, we will move beyond language and explore how these models, along with the architectural advances introduced here, enable breakthroughs in domains such as vision, time series, reinforcement learning, and structured reasoning. You will learn how to apply these tools in practice and how to make architectural choices based on the specific demands of each problem space.

**1** Ashish Vaswani et al. "Attention Is All You Need.", <https://arxiv.org/abs/1706.03762> (2017).

**2** Jianlin Su et al. "RoFormer: Enhanced Transformer with Rotary Position Embedding", <https://arxiv.org/abs/2104.09864> (2021).

**3** Shouyuan Chen et al. "Extending Context Window of Large Language Models via Positional Interpolation", <https://arxiv.org/abs/2306.15595> (2023).

**4** Bowen Peng et al. "YaRN: Efficient Context Window Extension of Large Language Models", <https://arxiv.org/abs/2309.00071> (2023).

**5** Ken M. Nakanishi "[Scalable-Softmax Is Superior for Attention.](#)", (2025).

**6** Amirhossein Kazemnejad et al. "[The Impact of Positional Encoding on Length Generalization in Transformers.](#)", (2023).

**7** Arthur Jacot et al. "Neural Tangent Kernel: Convergence and Generalization in Neural Networks", <https://arxiv.org/abs/1806.07572> (2018).

**8** Mozhdeh Gheini et al. "Cross-Attention is All You Need: Adapting Pretrained Transformers for Machine Translation", <https://arxiv.org/abs/2104.08771> (2021).

**9** Noam Shazeer "Fast Transformer Decoding: One Write-Head is All You Need", <https://arxiv.org/abs/1911.02150> (2019).

**10** Joshua Ainslie et al. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints", <https://arxiv.org/abs/2305.13245> (2023).

**11** Tri Dao et al. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness", <https://arxiv.org/abs/2205.14135> (2022).

**12** Tri Dao "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning", <https://tridao.me/publications/flash2/flash2.pdf> (2023).

**13** Jay Shah et al. "[FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision.](#)", (2024).

**14** The architecture is named after Grace Hopper, a pioneer in computer programming who famously popularized the term "bug" in 1947. She was known for carrying a piece of wire to illustrate how far light travels in a nanosecond, which she used as a playful response when asked to make things faster. Now she gets her revenge, as we name GPUs after her that can perform nearly two quadrillion operations per second.

**15** Woosuk Kwon et al. [“Efficient Memory Management for Large Language Model Serving with PagedAttention.”](#), (2023).