Chapter 3 - Looking Inside Large Language Models

An Overview of Transformer Models

The Inputs and Outputs of a Trained

> **Transformer**
>
> The common (and simplified) understanding of a Transformer LLM is that it is a software system that takes in text and generates text in response. Once a large enough text-in-text model is trained on a large enough high-quality dataset, it becomes to able to generate impressive and useful outputs.
>
> For the rest of this chapter, we will get to see the inner workings of an LLM, and how it actually works.

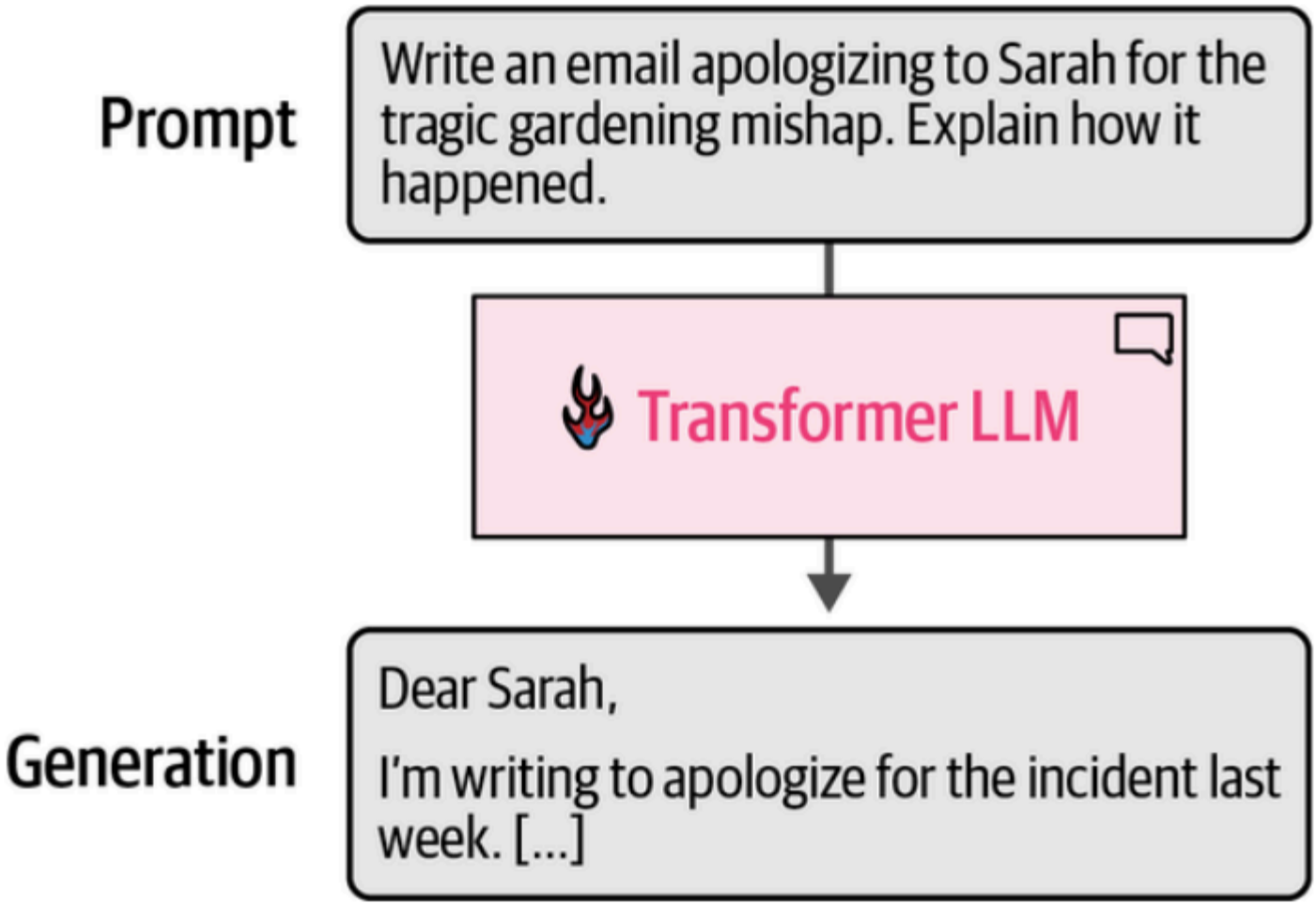

Figure 3-1. At a high level of abstraction, Transformer LLMs take a text prompt and output generated text.

In contrast to what it may look like and how it is illustrated above, an LLM generates one output at a time (not in batches).

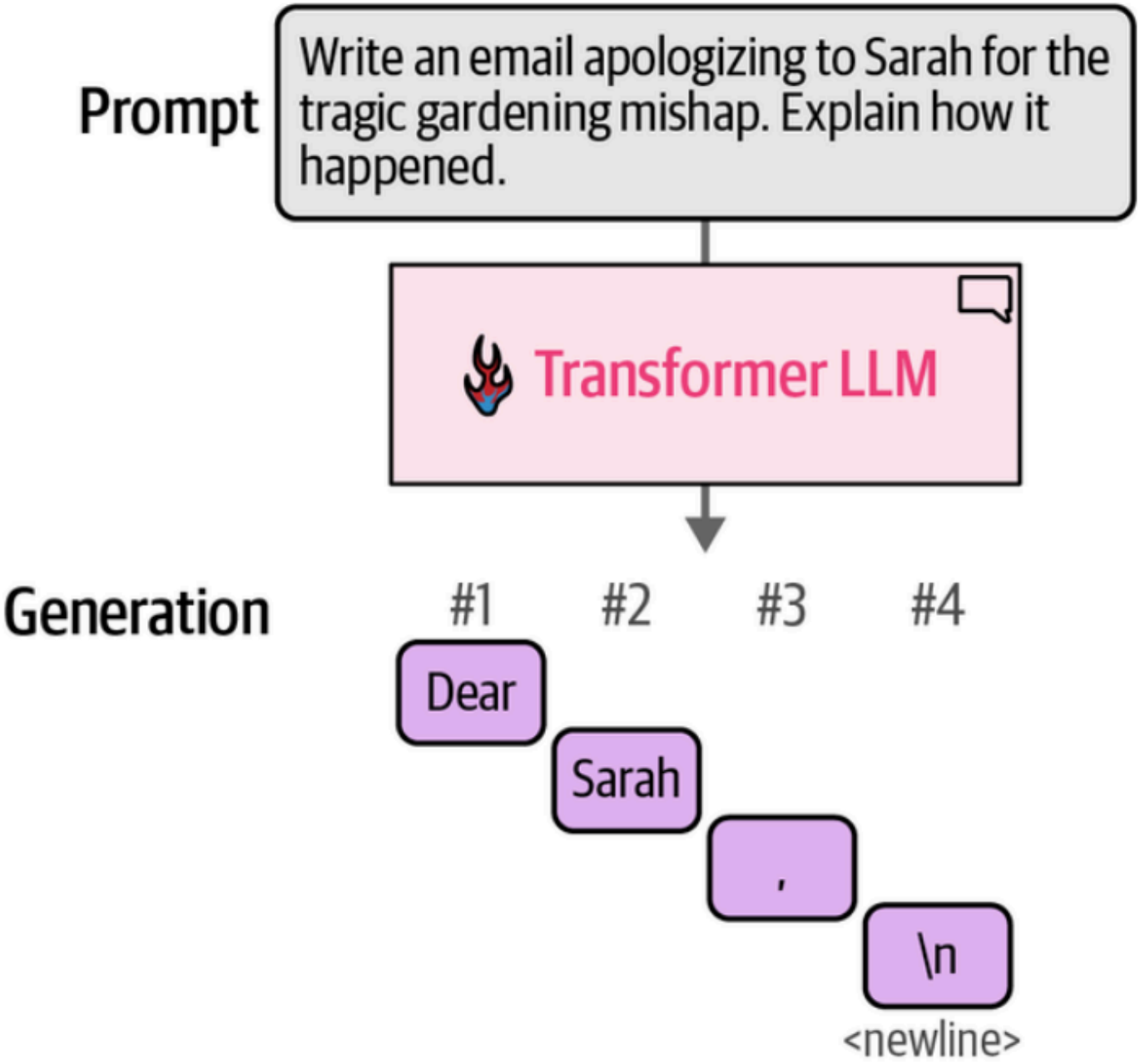

Figure 3-2. Transformer LLMs generate one token at a time, not the entire text at once.

> As discussed in Chapter 1, we know that an LLM is ***autoregressive***, meaning it takes the input prompt and append the previous LLM output. This is illustrated in the image below.

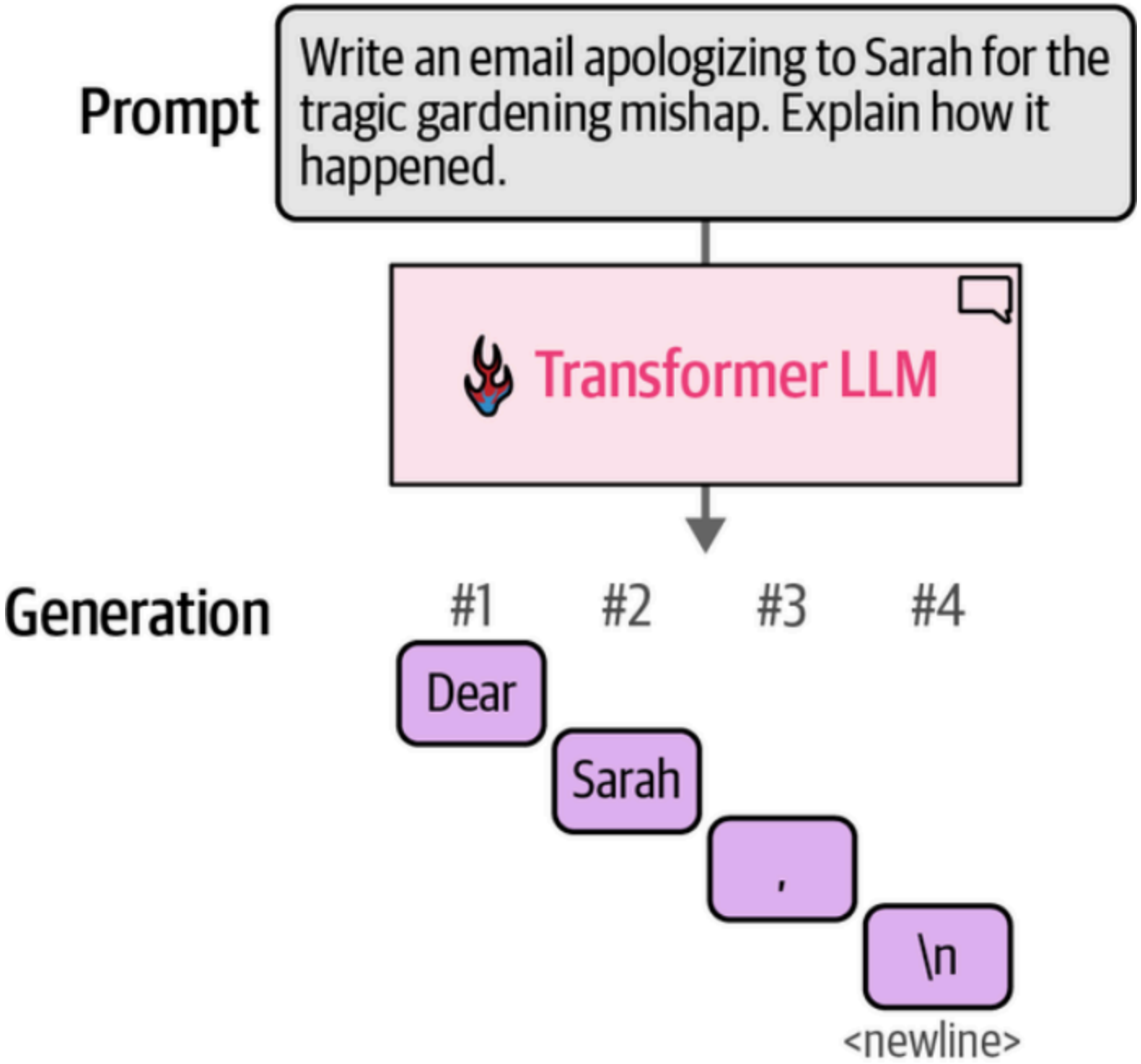

Figure 3-2. Transformer LLMs generate one token at a time, not the entire text at once.

BERT and BERT-like models, on the other hand, are not autoregressive.

Let's see this work in real time:

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

# Load model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")

model = AutoModelForCausalLm.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)
```

```
# Create a pipeline
generator = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=50,
    do_sample=False,
```

```
prompt = "Write an email apologizing to Sarah for the tragic gardening mishap. Explain
how it happened."

output = generator(prompt)
print(output[0]['generated_text'])
```

This generates the text:

```
Solution 1:
Subject: My Sincere Apologies for the Gardening Mishap

Dear Sarah,
I hope this message finds you well. I am writing to express my deep
```

You can see that the message is cut in the middle. This is because of the configuration `max_new_tokens = 50` . If you increase it, you will see more of the response.

The Components of a Forward Pass

What is Forward Pass?

The forward pass in LLMs/transformers is simply the process of the model reading and processing your input text to generate an output. Think of it like data flowing through a pipeline: your text goes in one end as numbers (each word becomes a token), flows through many layers that help the model understand the context and relationships between words, and comes out the other end as the model's response. It's called "forward" because data only **moves in one direction (input → output)**. When you use ChatGPT or any LLM, every response is generated through a forward pass - it's the model's way of "thinking" through your prompt to produce an answer.

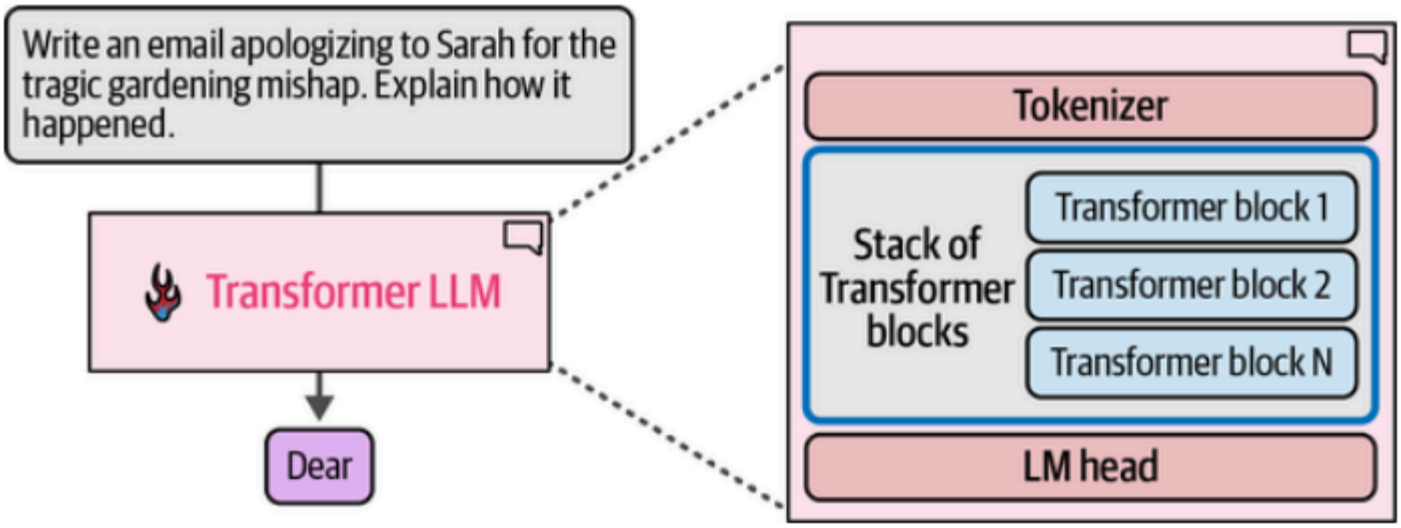

Figure 3-4. A Transformer LLM is made up of a tokenizer, a stack of Transformer blocks, and a language modeling head.

The flow of the computation follows the direction of the arrow from top to bottom. For each generated token, the process flows once through each of the Transformer blocks in the stack in order, then to the LM head, which finally outputs the probability distribution for the next token.

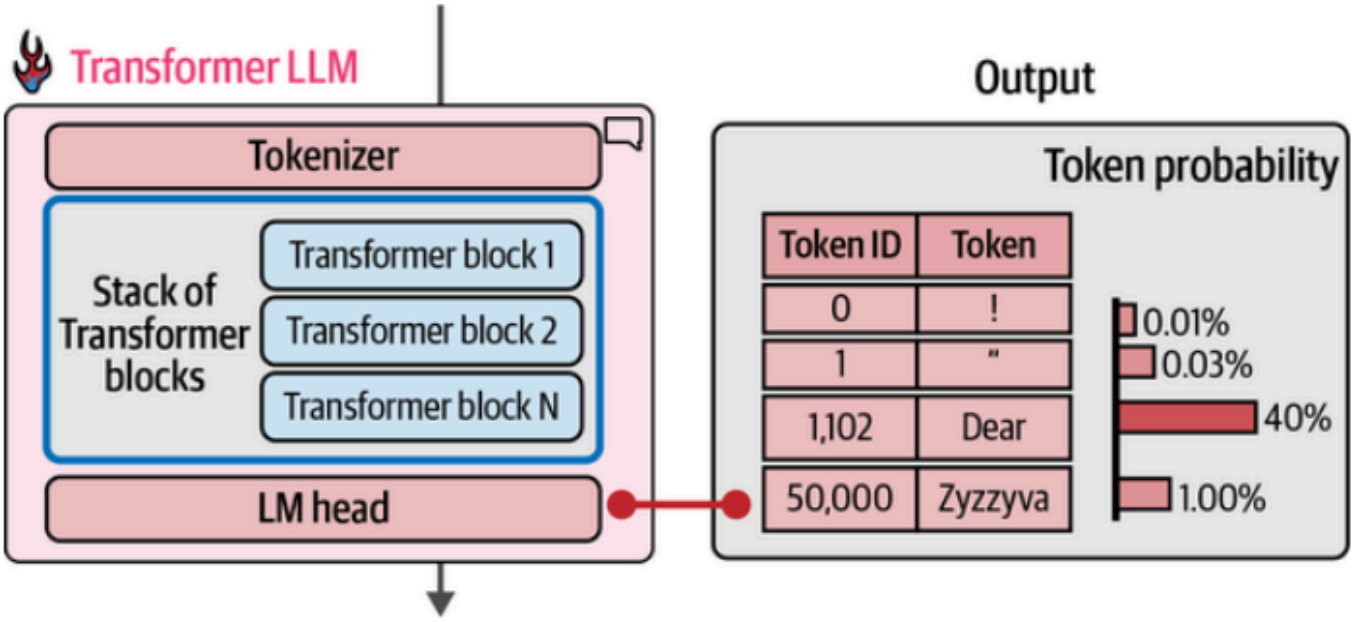

Figure 3-6. At the end of the forward pass, the model predicts a probability score for each token in the vocabulary.

> The LM head is a simple neural network layer itself. It is one of the multiple possible "heads" to attach to a stack of Transformer blocks to build different kinds of systems.

You can see the order of the layers by printing th model variable:

```
Phi3ForCausalLM(
    (model): Phi3Model(
        (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
        (embed_dropout): Dropout(p=0.0, inplace=False)
        (layers): ModuleList(
```

```
            (0-31): 32 x Phi3DecoderLayer(
                (self_attn): Phi3Attention(
                    (o_proj): Linear(in_features=3072, out_features=3072, bias=False)
                    (qkv_proj): Linear(in_features=3072, out_features=9216, bias=False)
                    (rotary_emb): Phi3RotaryEmbedding())
                (mlp): Phi3MLP(
                    (gate_up_proj): Linear(in_features=3072, out_features=16384,
bias=False)
                    (down_proj): Linear(in_features=8192, out_features=3072,
bias=False)
                    (activation_fn): SiLU()
                )
                (input_layernorm): Phi3RMSNorm()
                (resid_attn_dropout): Dropout(p=0.0, inplace=False)
                (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
                (post_attention_layernorm): Phi3RMSNorm()
            )
        )
        (norm): Phi3RMSNorm()
    )
    (lm_head): Linear(in_features=3072, out_features=32064, bias=False)
)
```

Looking at this structure, we can notice the following highlights:

- The model has two main parts: `model` (main body) and `lm_head` (output layer)
- Inside `model`: embeddings matrix `embed_tokens` with 32,064 tokens × 3,072 dimensions
- Contains 32 Transformer decoder blocks (`Phi3DecoderLayer`)
- Each block has: attention layer + feedforward network (MLP)
- Final `lm_head`: transforms 3,072-dim vector → vocabulary-sized output (probability per token)

Choosing a Single Token from the Probability

Distribution (Sampling/Decoding)

The output of each forward pass is a probability score for every token in the library. The method of choosing a single token from the probability distribution is called the ***decoding strategy.***

While choosing the one with the highest library seems to be the most intuitive step, it doesn't always give the best outputs for many use cases.

A better approach is to add some randomness and choose the token with the second or third highest probability.

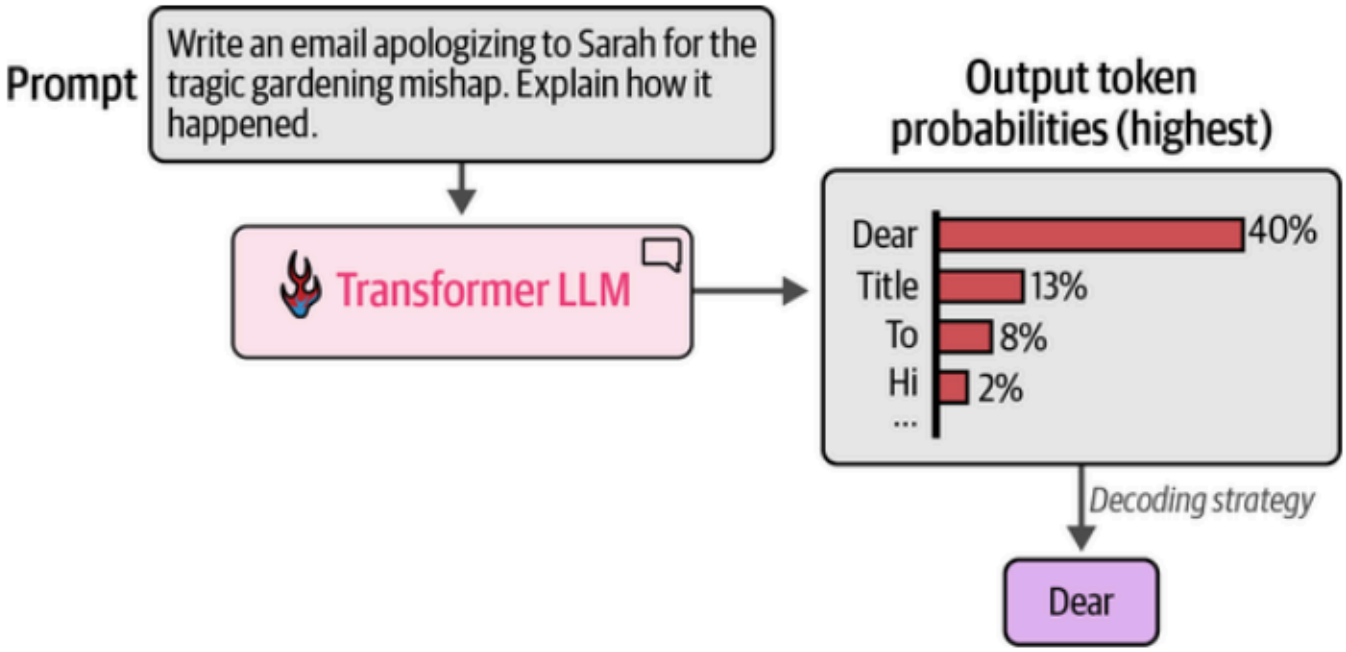

Figure 3-7. The tokens with the highest probability after the model's forward pass. Our decoding strategy decides which of the tokens to output by sampling based on the probabilities.

Choosing the highest scoring token every time is called ***greedy decoding***. This is what happens when ***temperature*** is set to 0.

> **What is temperature?**
>
> Temperature dictates the randomness of picking the token for the next word generation. A **high temperature** means there is **more randomness** and it takes into account lower probability tokens from the distribution, while **temperature close to 0** means the **highest probability tokens** will be picked, and a **temperature of 0** means the token with the **highest probability** will be picked.

More about temperature in Chapter 6.

In the following code, we pass the input tokens through the model, and then to `lm_head`:

```
prompt = "The capital of France is"

# Tokenize the input prompt
input_ids = tokenizer(prompt, return_tensors="pt").input_ids

# Tokenize the input prompt
input_ids = input_ids.to("cuda")

# Get the output of the model before the lm_head
model_output = model.model(input_ids)

# Get the output of the lm_head
lm_head_output = model.lm_head(model_output[0])
```

From the previous code blocks, we know that the model and its tokenizer has a vocabulary size of 32,064.

Now, the `lm_head_output` is of the shape `[1, 6, 32064]`. We can access the token probability scores for the last generated token using `lm_head_output[0, -1]`.

We can get the top scoring token ID, and then decode it to get the actual text:

```
token_id = lm_head_out[0, -1].argmax(-1)
tokenizer.decode(token_id)
```

This prints out:

```
Paris
```

Parallel Token Processing and Context Size

Compared to previous neural networks, transformers handle parallel computing better.

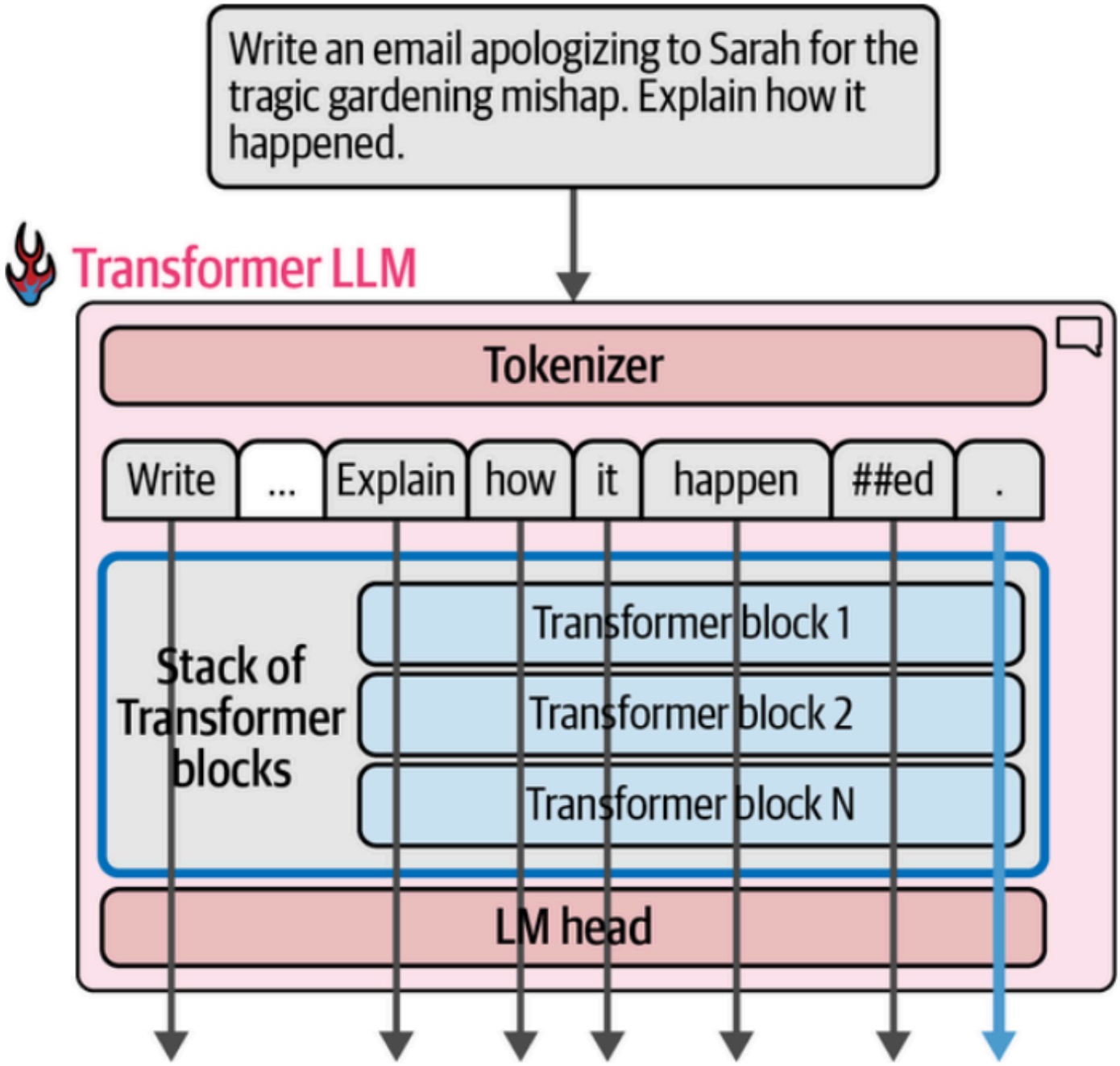

Figure 3-8. Each token is processed through its own stream of computation (with some interaction between them in attention steps, as we'll later see).

Current Transformer models have a limit for how many tokens they can process at once, called the model's ***context length***.

A model with 4K context length can only process 4K tokens and would only have 4K of these streams.

How Token Prediction Works in a Single Forward Pass:

When processing "Write an email... Explain how it happened.", the model tokenizes it into `[Write, ..., Explain, how, it, happen, ##ed, .]` and processes all 8 tokens all at the same time through the transformer blocks. Each token gets its own "stream" of computation, producing 8 output vectors.

For generation: We only use the **last position's output** (the "." token at position 8) to predict what comes next (likely "Dear" or "Subject"). The LM head takes this final vector and outputs probabilities for all

possible next tokens.

Why compute all 8 positions if we only use the last? The attention mechanism requires them. When computing the representation for "." (position 8), attention looks back at ALL previous positions' representations - it needs to know this is an email about a gardening mishap to predict appropriate next words. Without computing the representations for "Write", "Explain", "gardening", etc., the final position couldn't understand the full context.

The output:

- LM head produces: `[8 × vocab_size]` matrix (probabilities for all positions)
- For generation: only position 8's probabilities matter → sample "Dear"
- Next forward pass: append "Dear", run all 9 tokens through, use position 9 to predict token 10

This single forward pass produces *ONE* next token. The ***autoregressive*** loop repeats this process to generate the complete email.

If you're following along with the code examples, you can remember that the `lm_head` output was of shape `[1, 6, 32064]`.

That was because the input to it was of the shape `[1, 6, 3072]` which is a batch of **one** input string, containing **six** tokens, each of them represented by a vector of size 3,072 corresponding to the output vectors after the stack of Transformer blocks.

You can access these matrices and view their dimensions by printing:

```
model_output[0].shape
```

This outputs:

```
torch.Size([1, 6, 3072])
```

Similarly, we can print the output of the LM head:

```
lm_head_output.shape
```

This outputs:

```
torch.Size([1, 6, 32064])
```

For clarity:

```
Input:  [1, 6, 3,072]    (batch=1, positions=6, hidden_dim=3,072)
           ↓
    Matrix multiply each position independently
```

```
                ↓
Output: [1, 6, 32,064]   (batch=1, positions=6, vocab_size=32,064)
```

Speeding Up Generation by Caching Keys and Values

Recall that when generating the second token, we simply append the output token to the input and do another forward pass through the model.

If the model could cache the results of the previous calculation, there would be no longer a need to repeat the calculations of the previous streams.

> This is an optimization called ***keys and values (kv) cache*** and it significantly speeds up the generation process.

Keys and values are some of the central components of the attention mechanism. They will be discussed in detail later in this chapter.

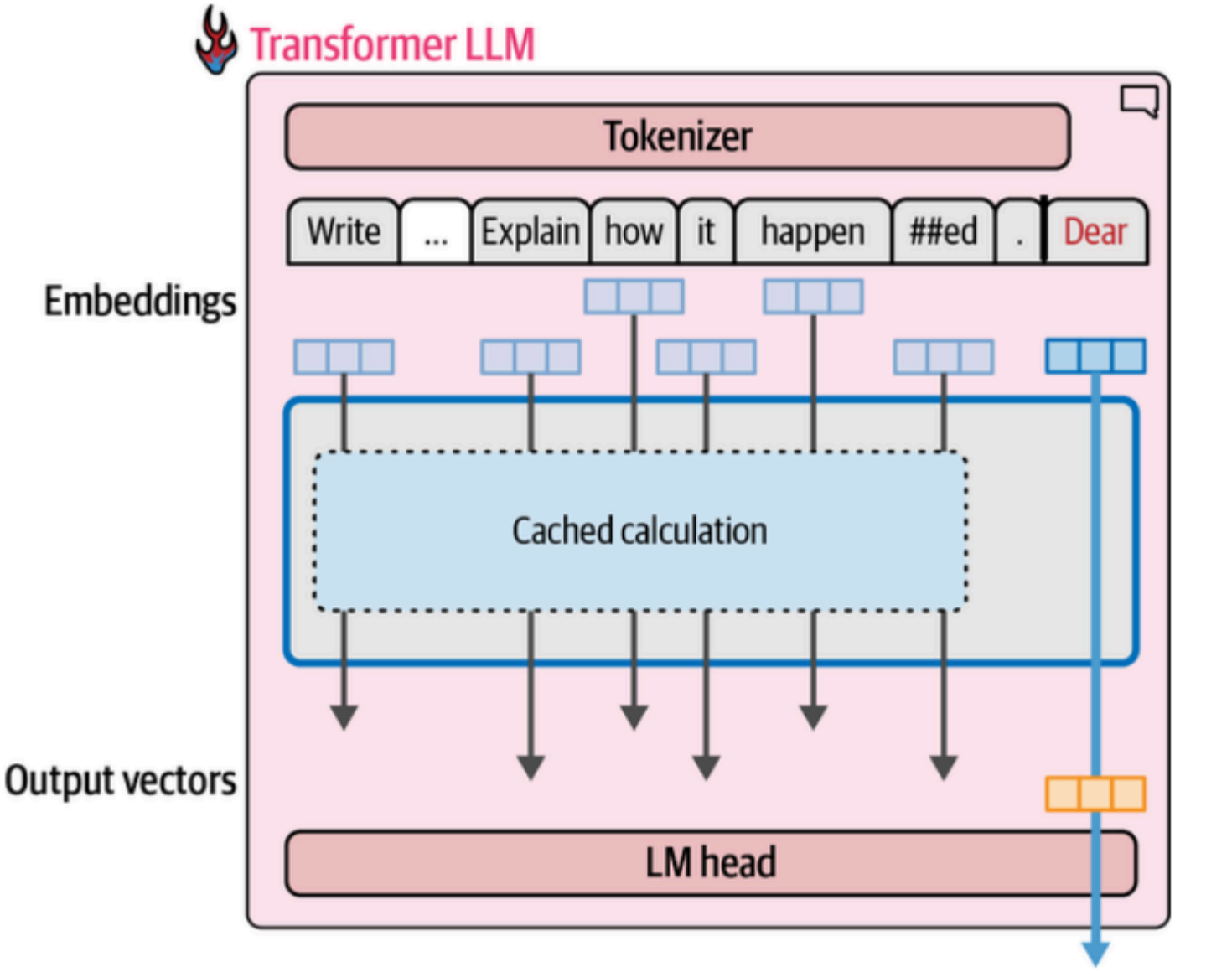

Figure 3-10. When generating text, it's important to cache the computation results of previous tokens

In HuggingFace Transformers, cache is enabled by default, but can be disabled by setting `use_cache` to `False`.

In the following lines of codes (which you can run) and their inputs, you the difference in speed with and without using cache:

```
prompt = "Write a very long email apologizing to Sarah for
the tragic gardening mishap. Explain how it happened."

# Tokenize the input prompt
input_ids = tokenizer(prompt, return_tensors="pt").input_ids
input_ids = input_ids.to("cuda")
```

To see the difference, we time it using `%%timeit` :

```
%%timeit -n 1
# Generate the text
generation_output = model.generate(
input_ids=input_ids,
max_new_tokens=100,
use_cache=True
)
```

On a Colab with a T4 GPU, this comes to 4.5 seconds.

```
%%timeit -n 1
# Generate the text
generation_output = model.generate(
input_ids=input_ids,
max_new_tokens=100,
use_cache=False
)
```

This comes out to 21.8 seconds.

ⓘ Caching In Action

Recall that the **model** is ***autoregressive***, meaning the output token from the previous generation step gets appended to the input for the next generation step. Now, where caching comes in is instead of recomputing the Key and Value vectors for all previous tokens at every layer, they are cached and ready to use. Without caching, every time we generate a new token, we'd have to recalculate the K and V matrices for the entire sequence from scratch through all transformer layers - but with caching, we only compute K and V for the newest token while reusing the cached K,V vectors from all previous tokens.

ⓘ What is KV

Key and Value (K,V) are two of the three matrices computed in the attention mechanism, the other is Query (Q). During attention, each token's embedding gets processed into these three vectors through learned linear projections. The Query represents "what information am I looking for", the Key represents "what information do I contain", and the Value represents "the actual information to pass along if selected". Attention computes similarity between one token's Query and all tokens' Keys to determine attention weights, then uses these weights to create a weighted sum of the Values. This produces a new representation that mixes information from relevant positions. Every transformer layer computes its own K,V,Q matrices with different learned weights, allowing each layer to look for different types of relationships - early layers might track syntax while later layers capture semantics.

Inside the Transformer

Transformer LLMs are composed of a series Transformer blocks (6 in the original transformer paper, over a hundred in many large LLMs).

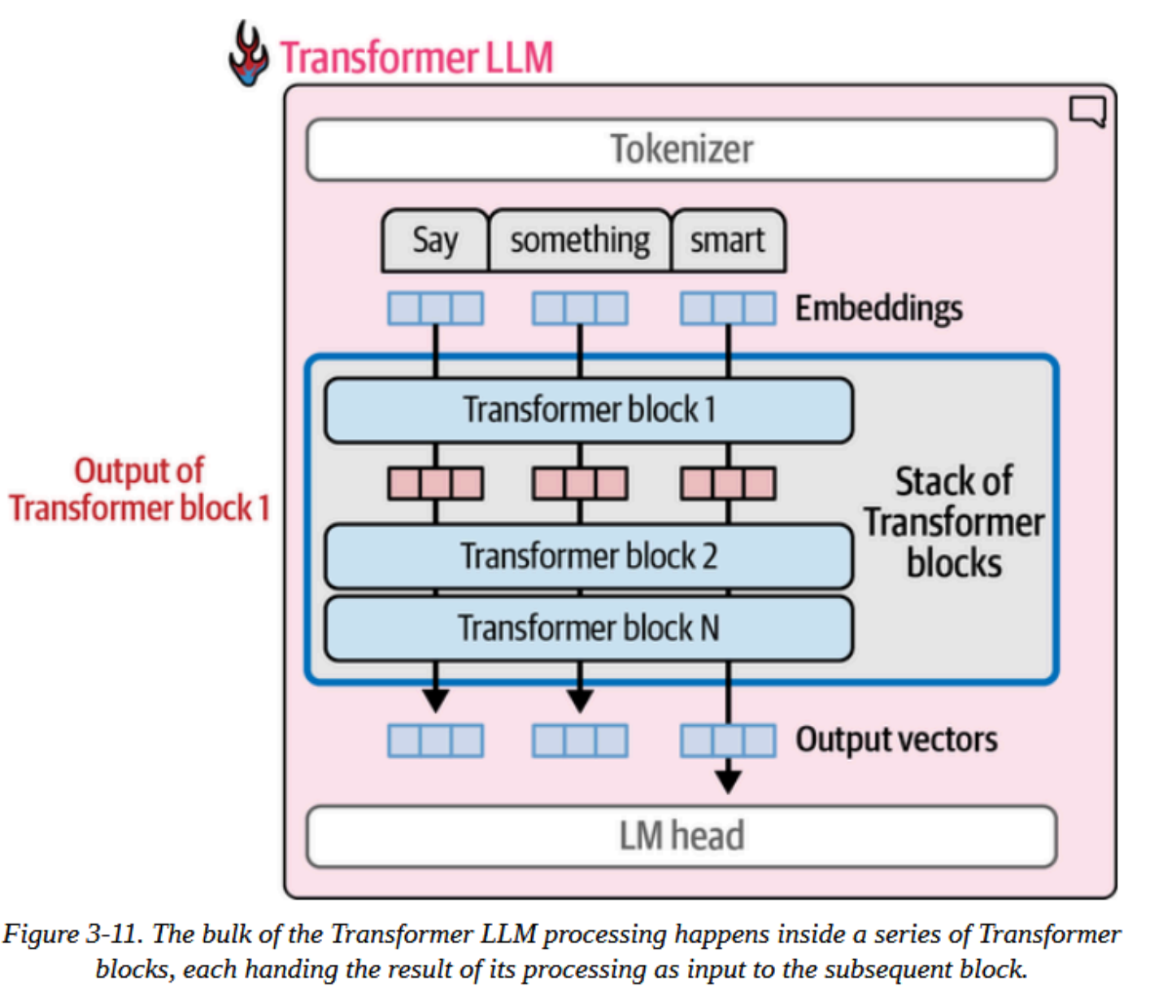

Figure 3-11. The bulk of the Transformer LLM processing happens inside a series of Transformer blocks, each handing the result of its processing as input to the subsequent block.

2 components of the Transformer block:

1. The ***attention layer*** is mainly concerned with incorporating relevant information from other input tokens and positions
2. The ***feedforward layer*** houses the majority of the model's processing capacity

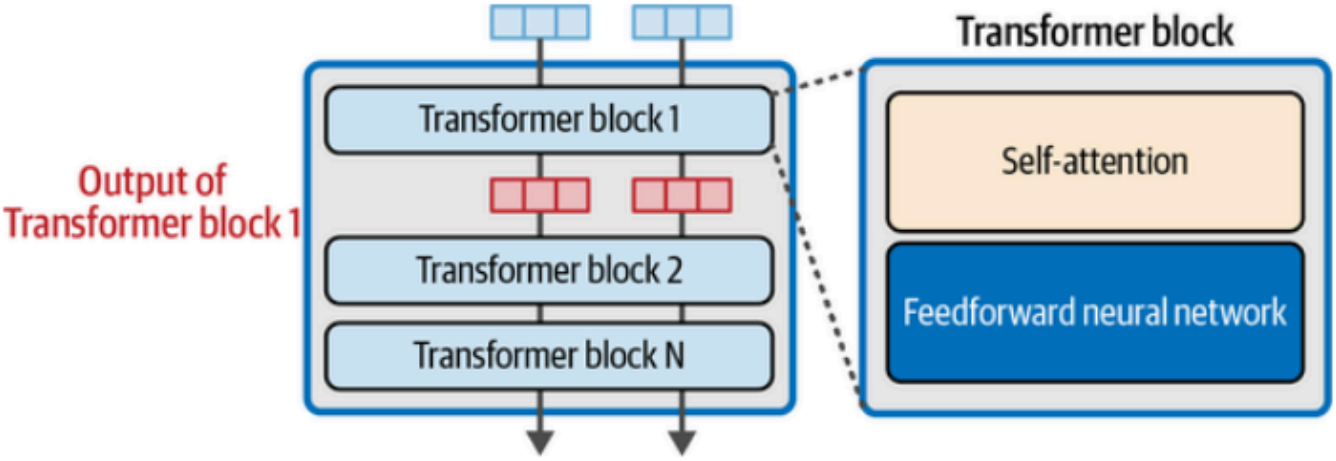

Figure 3-12. A Transformer block is made up of a self-attention layer and a feedforward neural network.

The feedforward neural network at a glance

A simple example of the intuition of the feedforward neural network would be if we pass the input "The Shawshank" to an LM, it would generate "Redemption" as the next most probable word (a 1994 film).

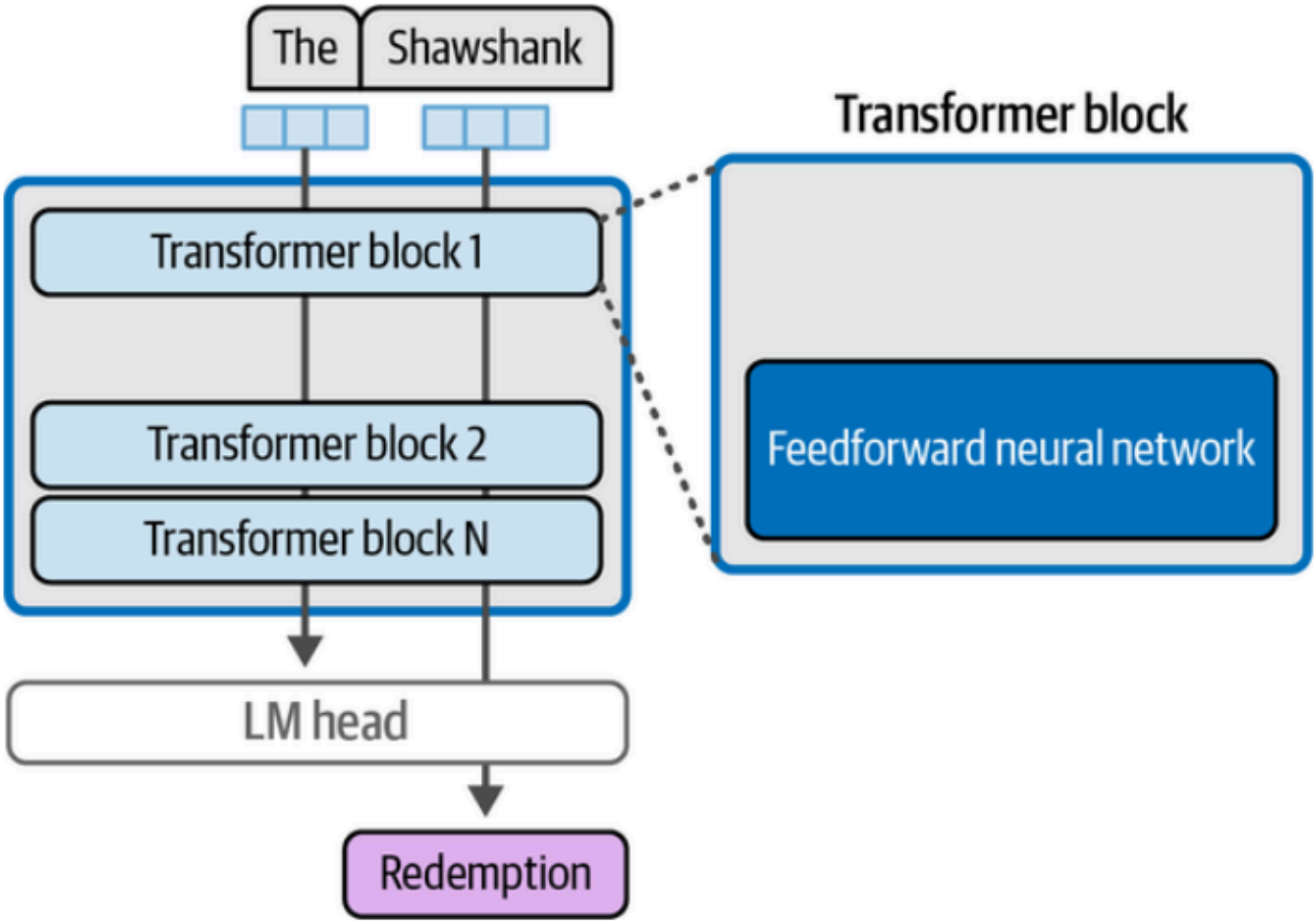

Figure 3-13. The feedforward neural network component of a Transformer block likely does the majority of the model's memorization and interpolation.

This knowledge comes primarily from the feedforward neural networks across all transformer layers. During training on large text datasets (which included many mentions of "The Shawshank Redemption"), the feedforward networks learned and stored this association. However, LLMs don't just memorize like databases - they also interpolate between learned patterns to generalize to new inputs not seen during training. The feedforward network handles both this memorization of specific information and the interpolation between data that allows the model to handle new information.

The attention layer at a glance

Although memorization and interpolation enable the model to handle many situations, it can only do so much. The nature of human language is that it is ambiguous in a lot of cases. This is where the importance of context comes into play.

What attention does

Attention mechanism that helps the model incorporate the context as it's processing a specific token.

For example: *"The dog chased the squirrel because **it** bit its tail"*

Humans would know ***it*** refers to the squirrel from context. However, models don't understand context.

Attention makes it possible for Transformers to understand that ***it*** is the squirrel.

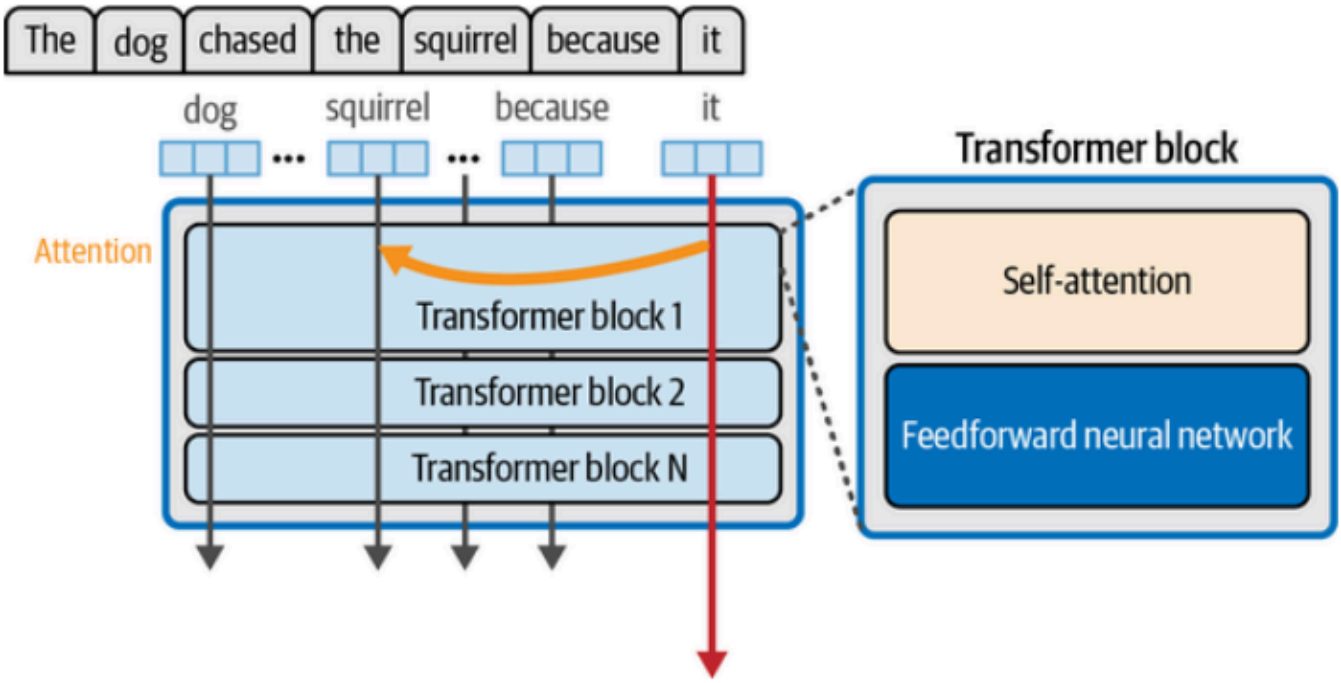

Figure 3-14. The self-attention layer incorporates relevant information from previous positions that help process the current token.

The model does that based on the patterns seen and learned from the training dataset.

Attention is all you need

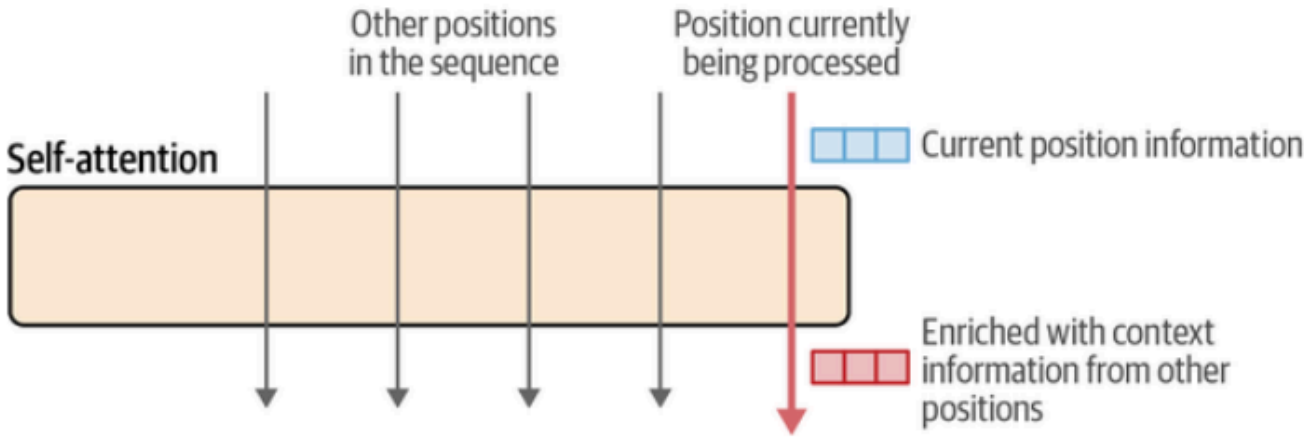

Figure 3-15. A simplified framing of attention: an input sequence and a current position being processed. As we're mainly concerned with this position, the figure shows an input vector and an output vector that incorporates information from the previous elements in the sequence according to the attention mechanism.

Two main steps are involved in the attention mechanism:

1. A way to score how relevant each of the previous input tokens are to the current token being processed (in the pink arrow)
2. Using those scores, we combine th information from the various positions into a single output vector.

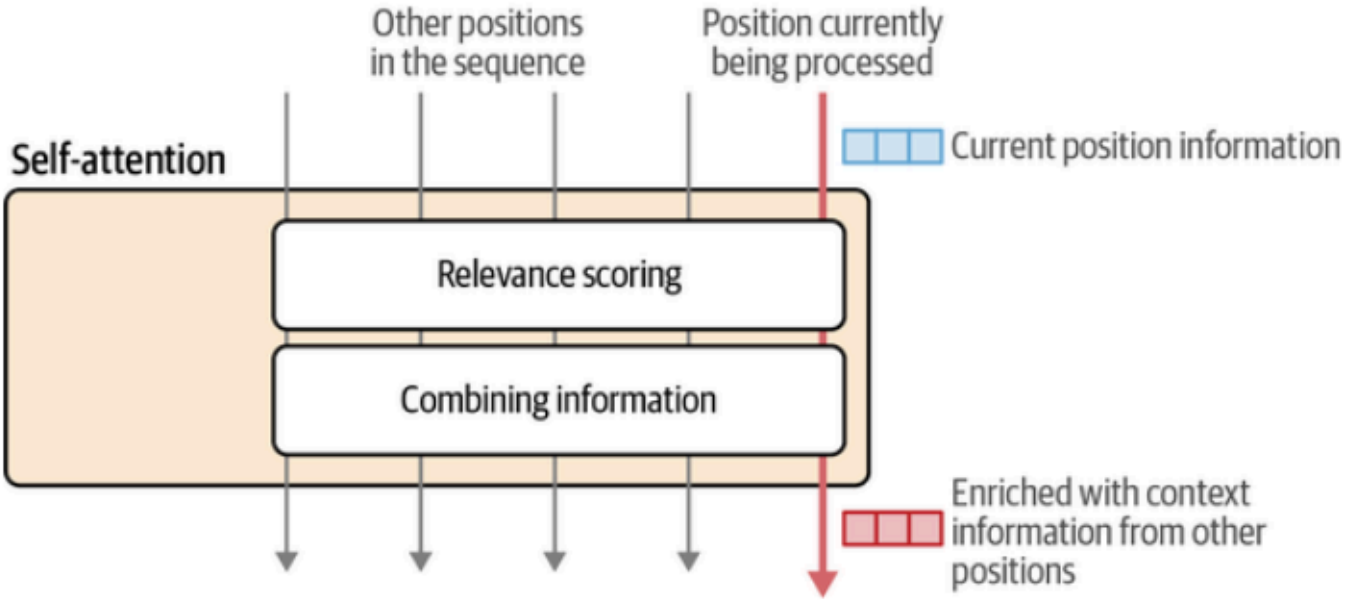

Figure 3-16. Attention is made up of two major steps: relevance scoring for each position, then a step where we combine the information based on those scores.

To improve the Transformer's attention capabilities, the model runs multiple attention mechanisms in parallel, each called an **attention head**. Each attention head analyzes and learns from the input from different perspectives. One might learn grammar, the other meaning. The process works by concatenating all the learned embeddings from each attention head to create a rich, multi-faceted understanding of the text.

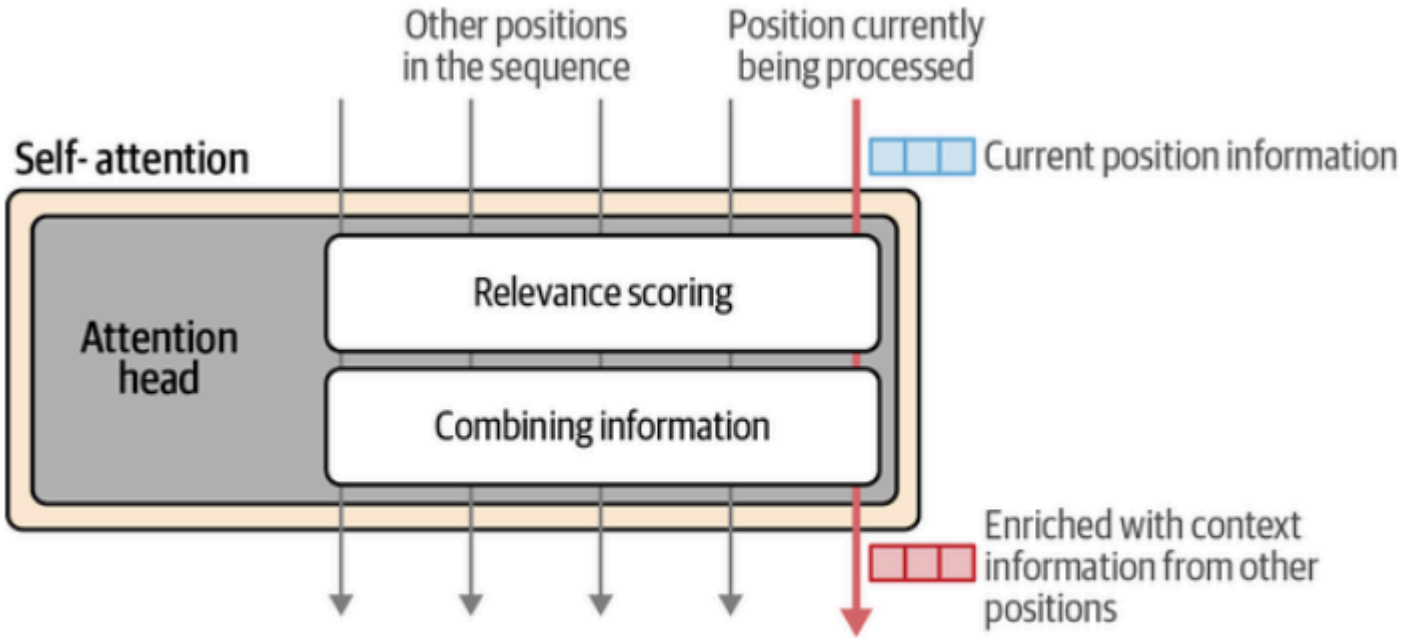

Figure 3-17. We get better LLMs by doing attention multiple times in parallel, increasing the model's capacity to attend to different types of information.

How attention is calculated

Let's see how attention is calculated inside a single attention head. First, let's observe the following first:

- The attention layer (of a generative LLM) is processing attention for a single position - *the next token*.
- As we know, the inputs to the attention layer are:
  - the vector representation (e.g., `[0.961, ..., 0.231]`) of the **current position or token**
  - the vector representations of the **previous tokens**
- The goal is to produce a new representation of the current position (the word to be predicted) that incorporates relevant information from the previous tokens:
  - For example, if we're processing the last position in the sentence `Sarah fed the cat because it`, we want `it` to represent that cat. Attention enables the model, using the context embeddings from previous tokens, to know this.
- As we've touched upon before, the training process produces three projection matrices:
  - A query **(Q)** projection matrix
  - A key **(K)** projection matrix
  - A value **(V)** projection matrix

For simplicity, let's focus on **one attention head**, as all of the attention heads use the same calculations anyway.

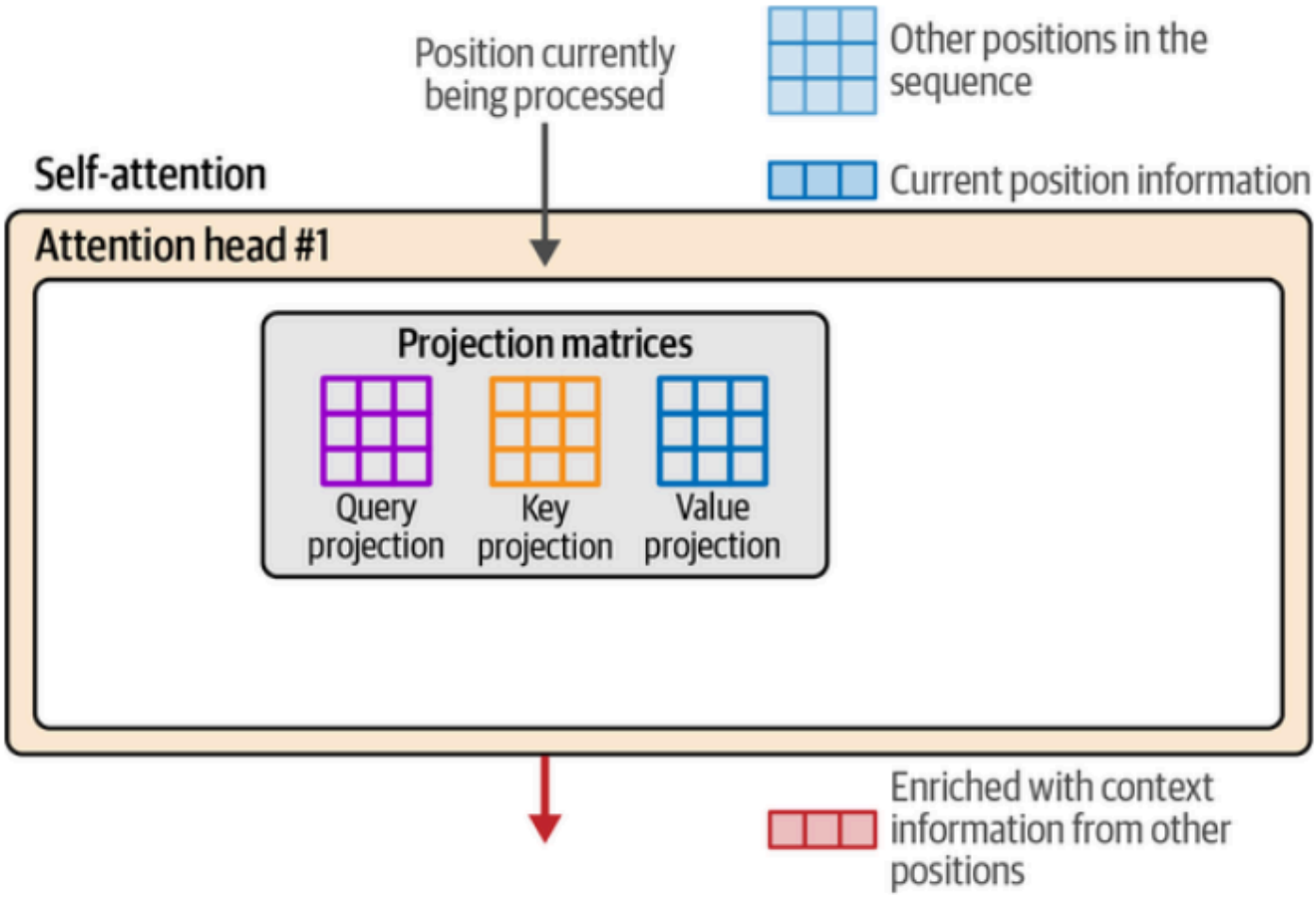

Figure 3-18. Before starting the self-attention calculation, we have the inputs to the layer and projection matrices for queries, keys, and values.

Attention starts by multiplying the inputs by the projection matrices to create three new matrices.

These resulting three matrices contain the information of the input tokens projected to three different spaces that help carry out **2 steps** of attention:

1. **Relevance Scoring**
2. **Combining Information**

A Simplified Explanation

The **key, query, and value projection matrices (W_K, W_Q, W_V)** are learned parameters established during training and are part of the model's architecture. The attention head receives the token embedding (not the token ID) combined with positional information. This input then gets multiplied by the projection matrices **W_K, W_Q, W_V** to produce **K, Q, and V matrices** - three new representations projected into different spaces.

This is well visualized in the image below.

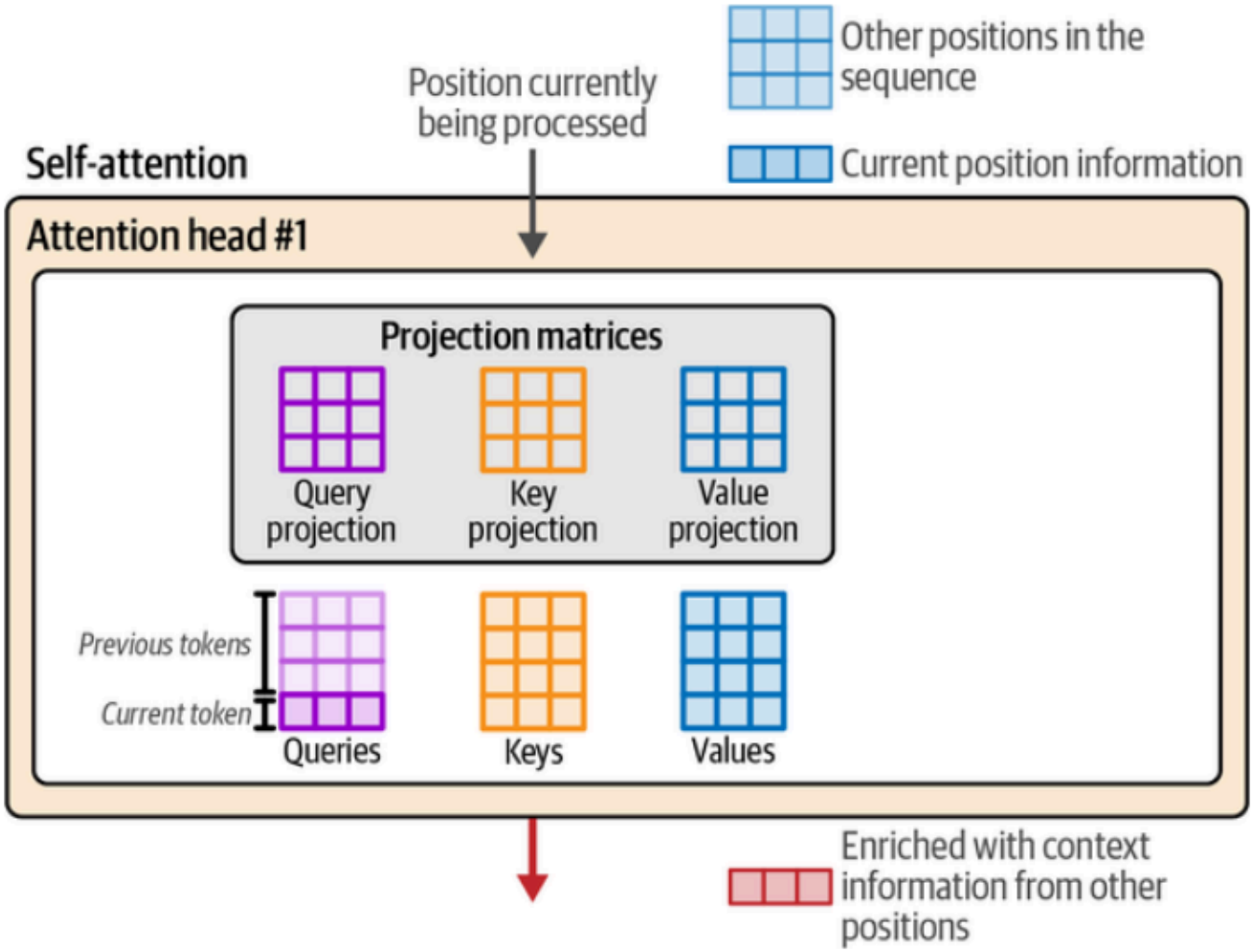

Figure 3-19. Attention is carried out by the interaction of the queries, keys, and values matrices. Those are produced by multiplying the layer's inputs with the projection matrices.

Self-attention: Relevance scoring

The ***relevance scoring*** step of attention is conducted by multiplying the **query vector** (the query matrix is a *stack* of multiple query vectors from previous positions and the current position) of the current position with the keys (K) matrix.

> This produces a score for each of the previous tokens.

Using a softmax operation, these scores are normalized so they sum up to 1.

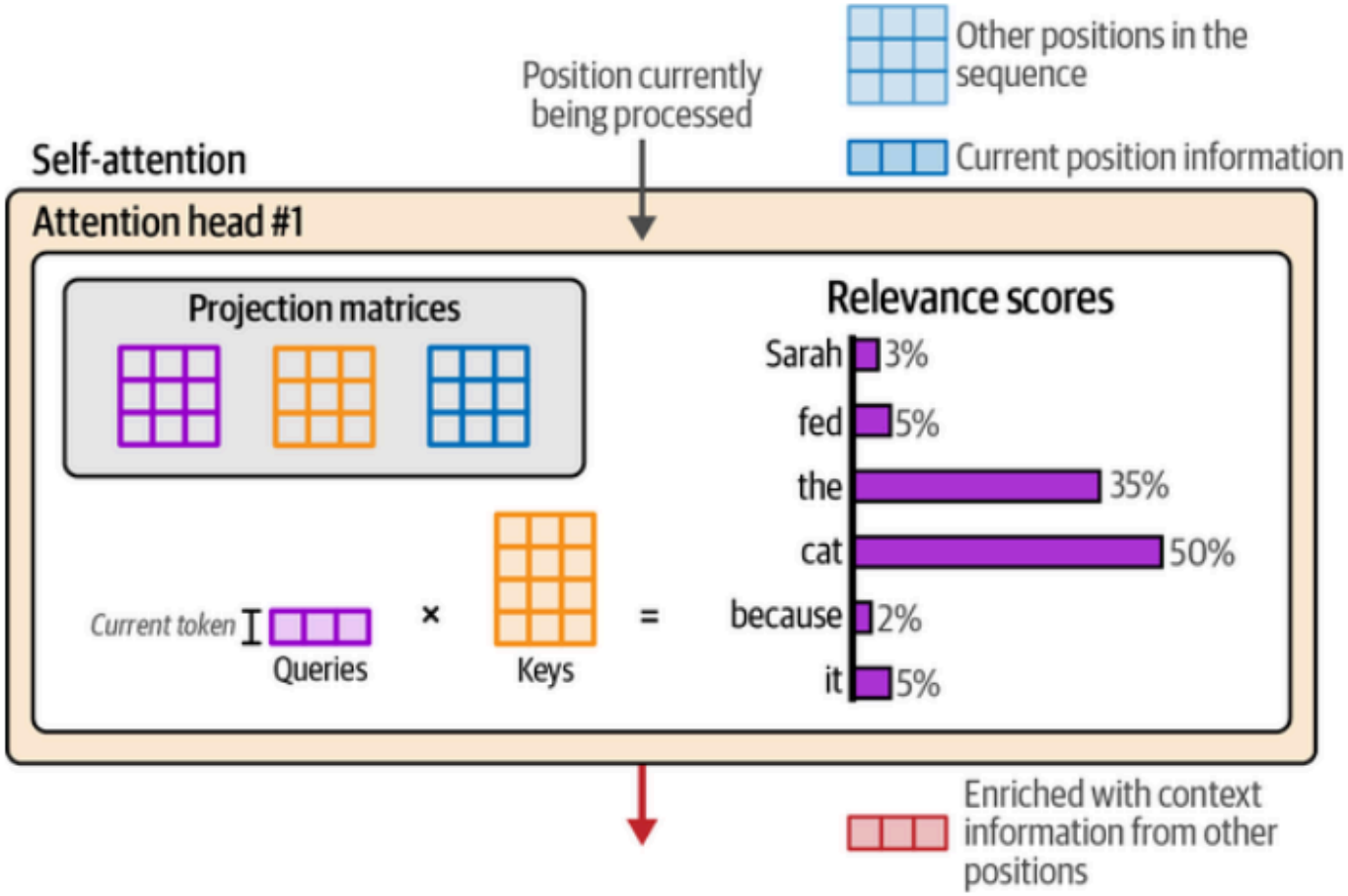

Figure 3-20. Scoring the relevance of previous tokens is accomplished by multiplying the query associated with the current position with the keys matrix.

Self-attention: Combining Information

After obtaining the relevance scores from the previous step, we multiply **we multiply the value (V) vector associated with each token by that token's score.**

The resulting vectors are then summed, which is the result of this attention step. This is well illustrated in the image below.

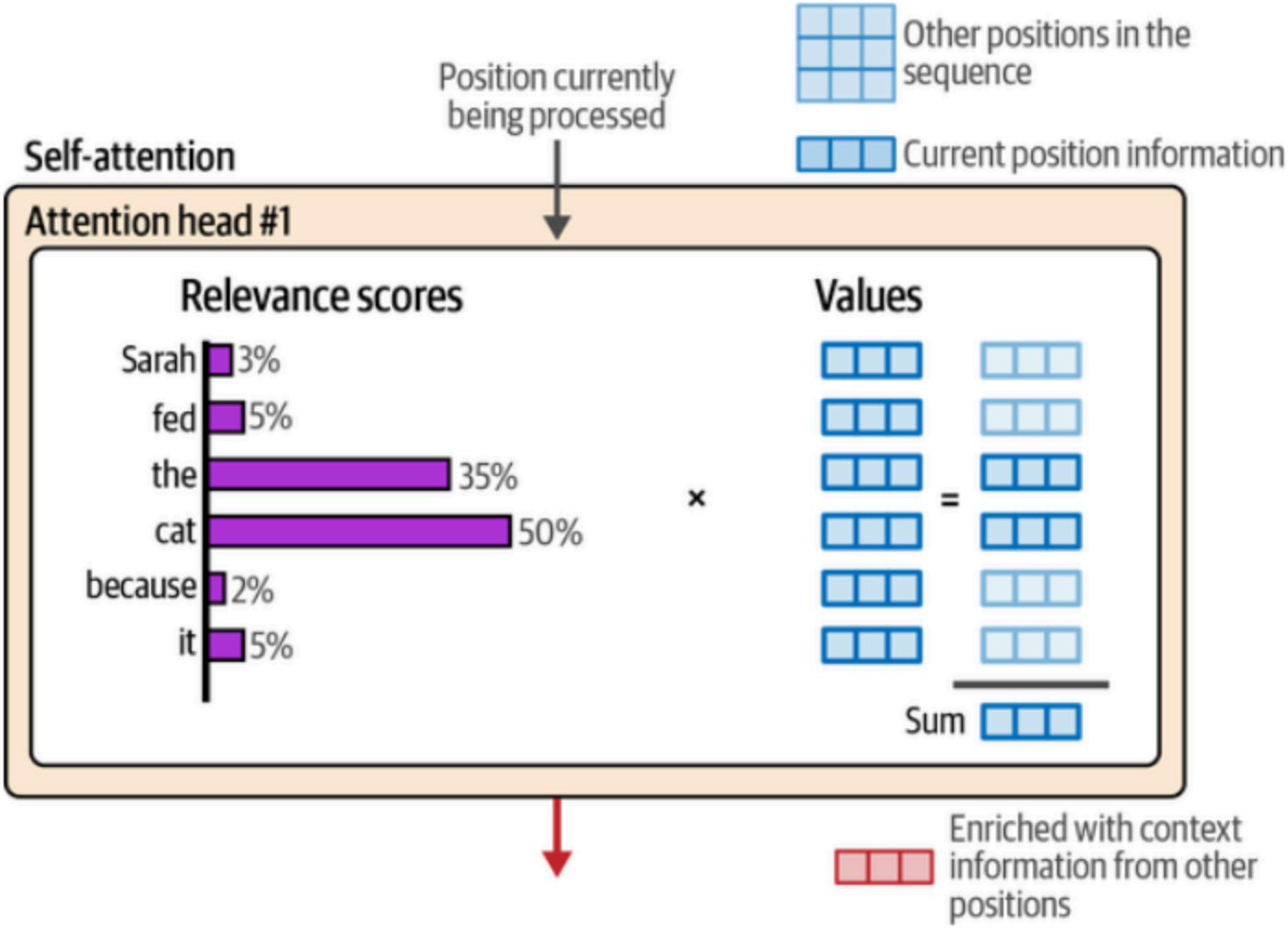

Figure 3-21. Attention combines the relevant information of previous positions by multiplying their relevance scores by their respective value vectors.

Recent Improvements to the Transformer Architecture

More Efficient Attention

Due to the computational resources that attention calculation takes, it has been the most researched part of the transformer architecture.

Local/sparse attention

This mechanism limits how many previous tokens the model can attend to - instead of looking at all previous tokens (full attention), sparse attention restricts the model to attending only to a small subset of recent positions. This significantly reduces computational cost.

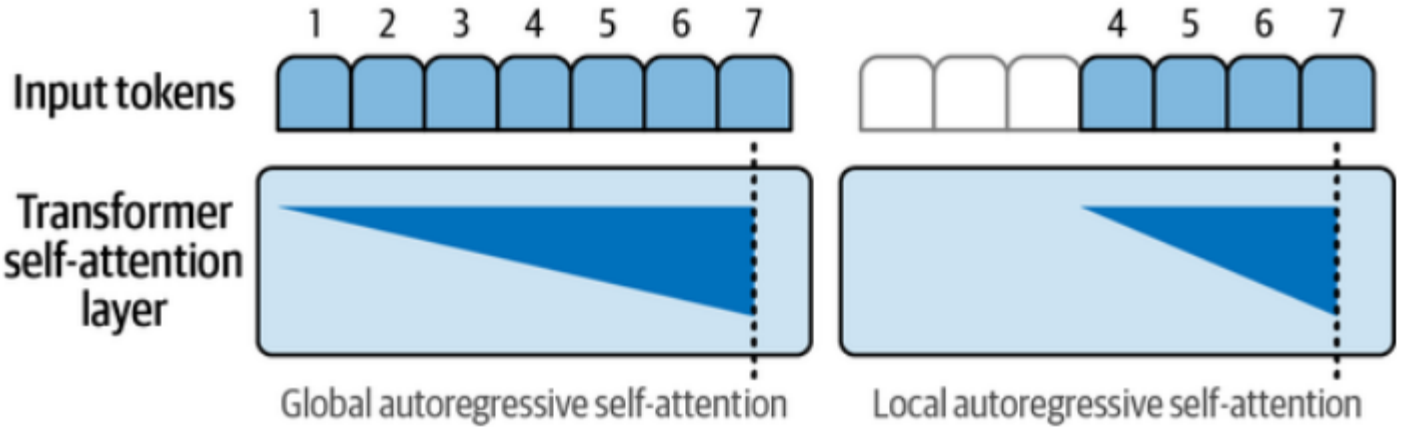

Figure 3-22. Local attention boosts performance by only paying attention to a small number of previous positions.

However, using only sparse attention would severely degrade generation quality, since models need broader context to generate text. GPT-3 addresses this by alternating between different attention types on its transformer blocks.

The architecture alternates between full-attention blocks (e.g., blocks 1 and 3) that can see all previous tokens, and sparse-attention blocks (e.g., blocks 2 and 4) that only see a limited window.

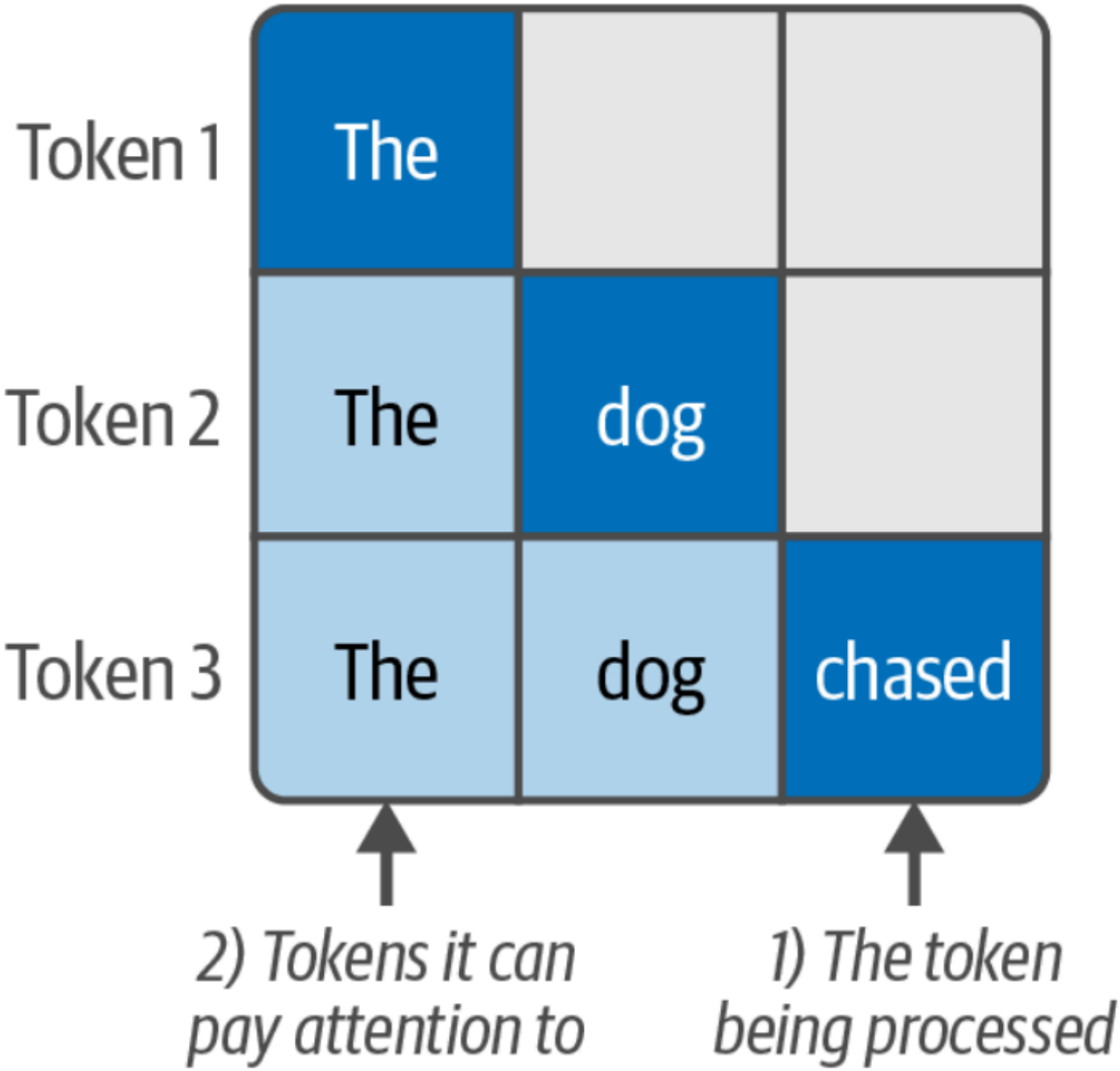

Figure 3-24. Attention figures show which token is being processed, and which previous tokens an attention mechanism allows it to attend to.

The attention patterns also highlight the fundamental difference between decoder and encoder models.

As discussed in Chapter 1, Decoder-based models (like GPT) are autoregressive, meaning they can only attend to previous tokens, never future ones. This enables text generation but limits context to one direction. This is illustrated in the image above.

In contrast, BERT's **bidirectional** attention can look at tokens both before and after the current position (hence the B in BERT), which is why it excels at understanding tasks but cannot generate text autoregressively.

Optimizing Attention: From Multi-head to Multi-query to Grouped-query

Multi-head Attention (Original)

In the original transformer architecture, each attention head maintains its own distinct set of Q, K, and V matrices.

This means if you have 16 heads, you have 16 different sets of projection matrices, allowing each head to learn completely different patterns and relationships.

While this provides rich, diverse perspectives on the input, **it requires substantial memory to store all these separate matrices.**

Multi-query Attention

Multi-query attention optimizes this by having all heads **share the same K and V matrices.**

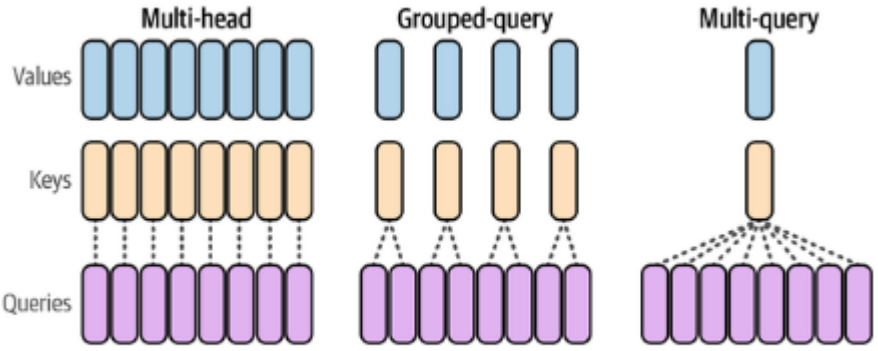

Figure 3-25. A comparison of different kinds of attention: the original multi-head, grouped-query attention, and multi-query attention (source: "Fast transformer decoding: One write-head is all you need").

Only the Query matrices remain unique to each head, dramatically reducing the memory footprint - instead of 16 sets of K,V matrices, you only need one shared set.

This makes inference much faster and more memory-efficient, though it can reduce model quality since heads have less flexibility to specialize.

Grouped-query Attention

Grouped-query attention strikes a balance between the two approaches.

Instead of all heads sharing one set of K,V matrices or each having their own, heads are divided into groups that share matrices within the group.

For example, *16 heads might be split into 4 groups, with each group of 4 heads sharing one set of K,V matrices.*

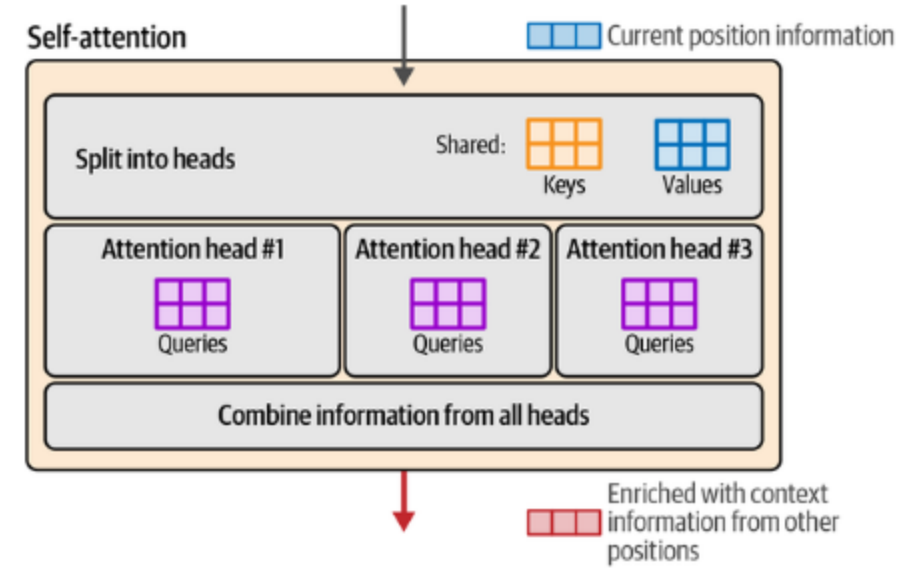

Figure 3-27. Multi-query attention presents a more efficient attention mechanism by sharing the keys and values matrices across all the attention heads.

This preserves much of the efficiency gain while maintaining better model quality than pure multi-query attention.

Flash Attention

Flash Attention addresses a different problem - not the computation itself, but how data moves through GPU memory.

It optimizes the transfer of values between the GPU's fast but small SRAM and its larger but slower HBM memory.

Flash Attention provides significant speedups for both training and inference without changing any of the mathematical operations of attention.

The Transformer Block

Original Transformer Block

Recall that the two major components of a Transformer block are an attention layer and a feedforward neural network.

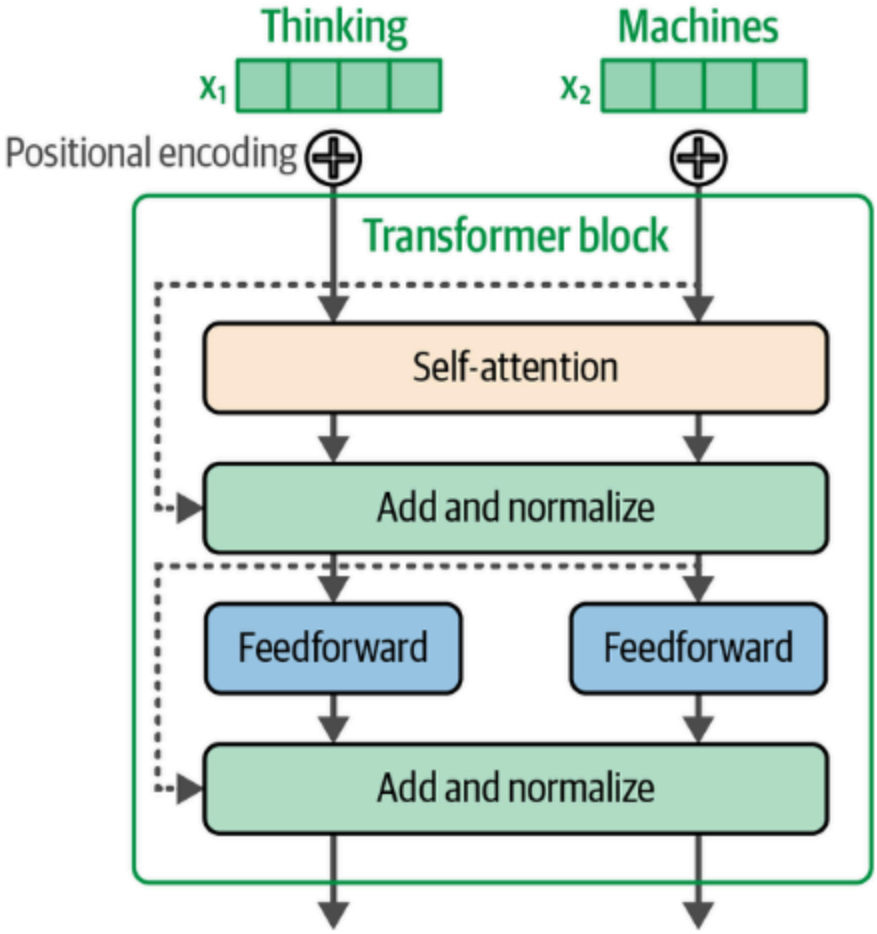

Figure 3-29. A Transformer block from the original Transformer paper.

Modern Transformer Block (2024-era)

Modern transformers have refined this design based on years of research.

The key change is pre-normalization - applying normalization before the attention and feedforward layers rather than after, which speeds up training convergence.

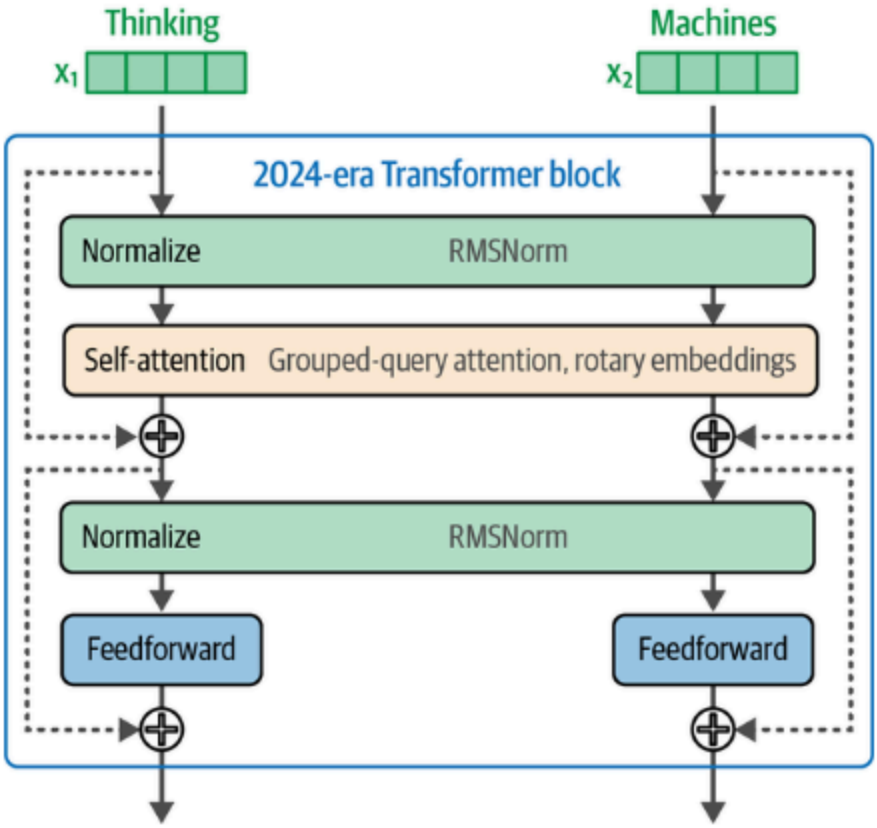

Figure 3-30. The Transformer block of a 2024-era Transformer like Llama 3 features some tweaks like pre-normalization and an attention optimized with grouped-query attention and rotary embeddings.

They've also replaced components with more efficient versions: RMSNorm instead of LayerNorm (simpler computation), SwiGLU instead of ReLU (better performance), and incorporated grouped-query attention and rotary positional embeddings for additional improvements.

Positional Embeddings (RoPE)

Problems with Absolute Position Embeddings

Traditional positional embeddings simply mark tokens as position 1, 2, 3, and so on.

This works fine when processing single documents, but breaks during training when multiple short documents are packed into one context window.

If document 2 starts at position 50 in the packed sequence, its first token thinks it has 49 tokens of prior context **when those tokens are actually from an unrelated document the model should ignore.**

Rotary Positional Embeddings (RoPE)

RoPE solves this by encoding position differently - instead of adding position at the input, it's incorporated during the attention computation itself.

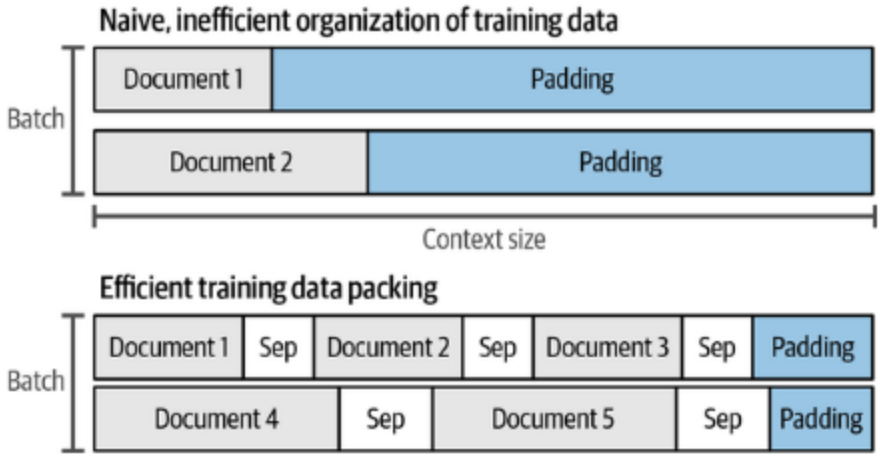

Figure 3-31. Packing is the process of efficiently organizing short training documents into the context. It includes grouping multiple documents in a single context while minimizing the padding at the end of the context.

The method rotates vectors in the embedding space based on their position, and this rotation is applied to the Q and K matrices just before calculating attention scores.

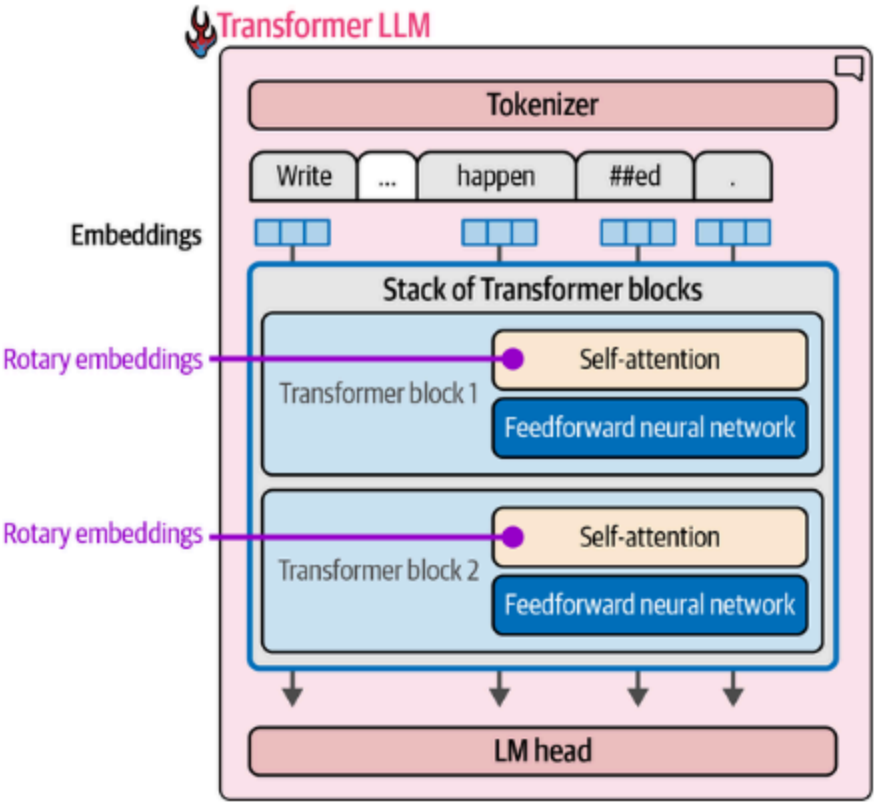

Figure 3-32. Rotary embeddings are applied in the attention step, not at the start of the forward pass.

This approach captures both absolute and relative position information while naturally handling the document packing problem.

Other Architectural Improvements

Transformer architectures continue to evolve fast with research focusing on improving efficiency and domain-specific applications.

Transformers have been successfully adapted for computer vision (Vision Transformers), robotics (RT-X models), and time series analysis.

Alternative architectures like Mamba and RWKV are also being developed, each offering specific advantages like longer context windows or faster inference for particular use cases.