Chapter 3. Looking Inside Large Language Models

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 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

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 *mcronin@oreilly.com*.

Now that we have a sense of tokenization and embeddings, we're ready to dive deeper into the language model and see how it works. In this chapter, we'll look at some of the main intuitions of how Transformer language models work. Our focus here will be directly on text generation models so we get a deeper sense for generative LLMs in particular.

We'll be looking at both the concepts and some code examples that demonstrate them. Let's start by loading a language model and getting it ready for generation by declaring a pipeline. In your first read, feel free to skip the code and focus on grasping the concepts involved. Then in a second read, the code will get you to start applying these concepts.

```
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,
)
```

An Overview of Transformer Models

Let's begin our exploration with a high-level overview of the model, and then we'll see how later work has improved upon the Transformer model since its introduction in 2017.

The Inputs and Outputs of a Trained Transformer LLM

The most common picture of understanding the behavior of a Transformer LLM is to think of it as a software system that takes in text and generates text in response. Once a large enough text-in-text-out model is trained on a large enough high-quality dataset, it becomes able to generate impressive and useful outputs. Figure 3-1 shows one such model used to author an email.

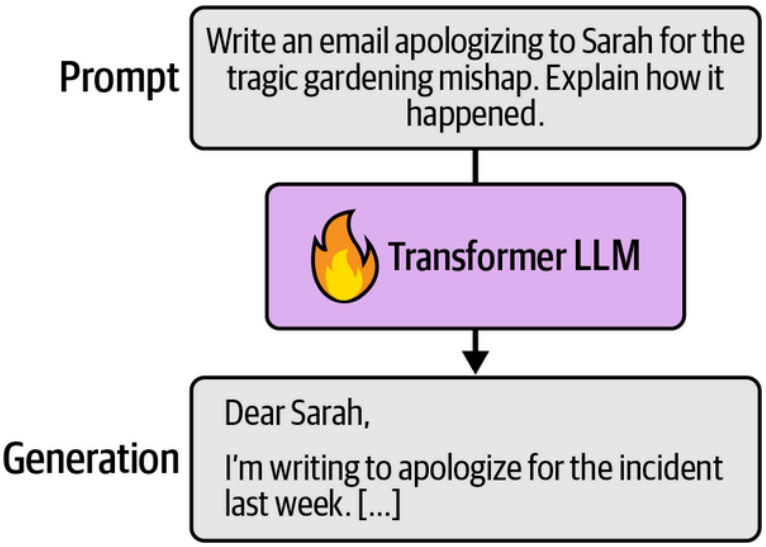

Figure 3-1. At a high level of abstraction, Transformer LLMs take a text prompt and output generated text.

The model does not generate the text all in one operation. The model actually generates one token at a time. Figure 3-2 shows four steps of token generation in response to the input prompt. Each token generation step is one forward pass through the model (that's machine-learning speak for the inputs going into the neural network and flowing through the computations it needs to produce an output on the other end of the computation graph).

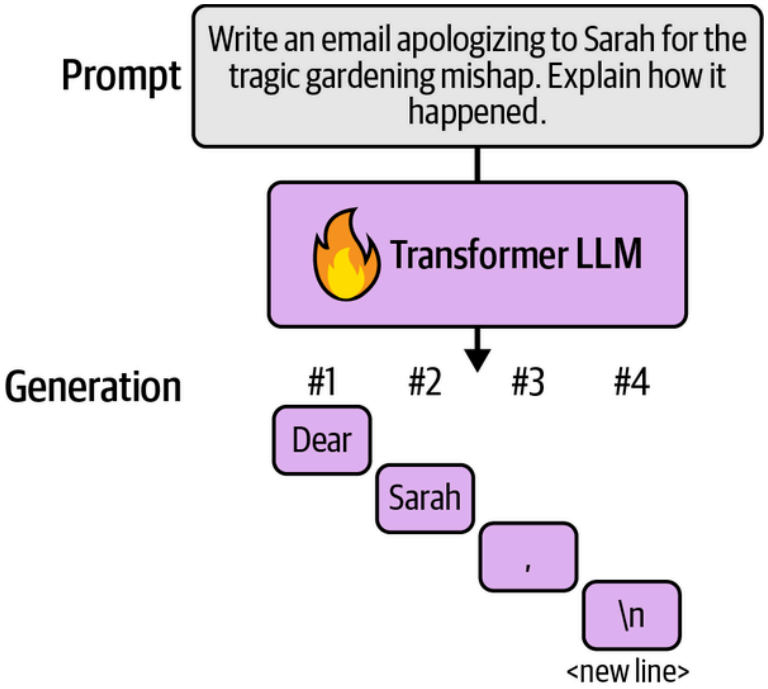

Figure 3-2. Transformer LLMs generate one token at a time, not the entire text at once.

After each token generation, we tweak the input prompt for the next generation step by appending the output token to the end of the input prompt. We can see this in Figure 3-3.

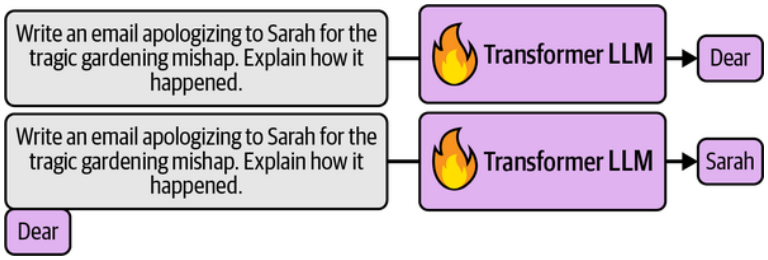

Figure 3-3. An output token is appended to the prompt, then this new text is presented to the model again for another forward pass to generate the next token.

This gives us a more accurate picture of the model as simply predicting the next token based on an input prompt. Software around the neural network basically runs it in a loop to sequentially expand the generated text until completion.

There's a specific word used in machine learning to describe models that consume their earlier predictions to make later predictions (e.g., the model's first generated token is used to generate the second token). They're called *autoregressive* models. That is why you'll hear text generation LLMs being called autoregressive models. This is often used to differentiate text generation models from text representation models like BERT, which are not autoregressive.

This autoregressive, token-by-token generation is what happens under the hood when we generate text with the LLM like we see here:

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

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
```

We can see the model starting to write the email starting with the subject. It stopped abruptly because it reached the token limit we established by setting `max_new_tokens` to 50 tokens. If we increase that, it will continue until concluding the email.

The Components of the Forward Pass

In addition to the loop, two key internal components are the tokenizer and the language modeling head (LM head). Figure 3-4 shows where these components lie in the system. We saw in the previous chapter how tokenizers break down the text into a sequence of token IDs that then become the input to the model.

The tokenizer is followed by the neural network: a stack of Transformer blocks that do all of the processing. That stack is then followed by the LM head, which translates the output of the stack into probability scores for what the most likely next token is.

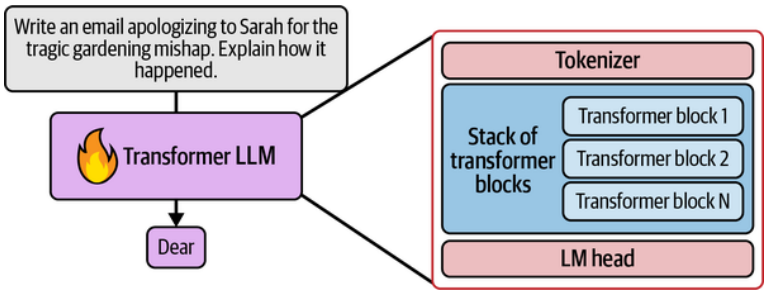

Figure 3-4. A Transformer LLM is made up of a tokenizer, a stack of transformer blocks, and a language modeling head.

Recall from Chapter 2 that the tokenizer contains a table of tokens—the tokenizer's *vocabulary*. The model has a vector representation associated with each of these tokens in the vocabulary (token embeddings). Figure 3-5 shows both the vocabulary and associated token embeddings for a model with a vocabulary of 50,000 tokens.

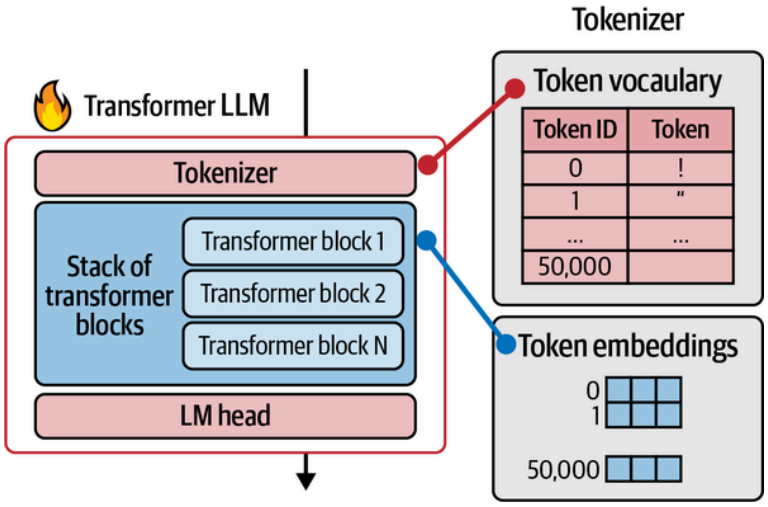

Figure 3-5. The tokenizer has a vocabulary of 50,000 tokens. The model has token embeddings associated with those embeddings.

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 seen in Figure 3-6.

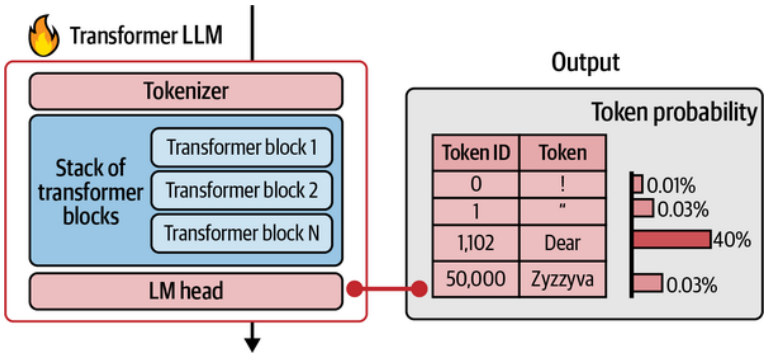

Figure 3-6. At the end of the forward pass, the model predicts a probability score for each token in the vocabulary.

The LM head is a simple neural network layer itself. It is one of multiple possible "heads" to attach to a stack of Transformer blocks to build different kinds of systems. Other kinds of Transformer heads include sequence classification heads and token classification heads.

We can display the order of the layers by simply printing out the model variable. For this model, we have:

```
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:

- This shows us the various nested layers of the model. The majority of the model is labeled `model`, followed by `lm_head`.
- Inside the `Phi3Model` model, we see the embeddings matrix `embed_tokens` and its dimensions. It has 32,064 tokens each with a vector size of 3,072.
- Skipping the dropout layer for now, we can see the next major component is the stack of Transformer decoder layers. It contains 32 blocks of type `Phi3DecoderLayer`.
- Each of these Transformer blocks includes an attention layer and a feedforward neural network (also known as an `mlp` or multilevel perceptron). We'll cover these in more detail later in the chapter.
- Finally, we see the `lm_head` taking a vector of size 3,072 and outputting a vector equivalent to the number of tokens the model knows. That output is the probability score for each token that helps us select the output token.

Choosing a Single Token from the Probability Distribution (Sampling/Decoding)

At the end of processing, the output of the model is a probability score for each token in the vocabulary, as we saw previously in Figure 3-6. The method of choosing a single token from the probability distribution is called the *decoding strategy*. Figure 3-7 shows how this leads to picking the token "Dear" in one example.

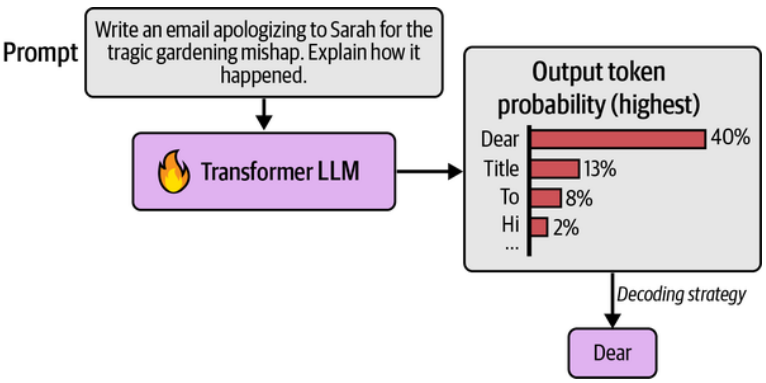

Figure 3-7. The tokens with the highest probability after the model's forward pass. Our decoding strategy decides which of the tokens to output by sampling based on the probabilities.

The easiest decoding strategy would be to always pick the token with the highest probability score. In practice, this doesn't tend to lead to the best outputs for most use cases. A better approach is to add some randomness and sometimes choose the second or third highest probability token. The

idea here is to basically *sample* from the probability distribution based on the probability score, as the statisticians would say.

What this means for the example in Figure 3-7 is that if the token "Dear" has a 40% probability of being the next token, then it has a 40% chance of being picked (instead of greedy search, which would pick it directly for having the highest score). So with this method, all the other tokens have a chance of being picked according to their score.

Choosing the highest scoring token every time is called *greedy decoding*. It's what happens if you set the temperature parameter to zero in an LLM. We cover the concept of temperature in Chapter 6.

Let's look more closely at the code that demonstrates this process. In this code block, we pass the input tokens through the model, and then `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])
```

Now, `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]`, which uses the index 0 across the batch dimension; the index -1 gets us the last token in the sequence. This is now a list of probability scores for all 32,064 tokens. We can get the top scoring token ID, and then decode it to arrive at the text of the generated output token:

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

In this case this turns out to be:

```
Paris
```

Parallel Token Processing and Context Size

One of the most compelling features of Transformers is that they lend themselves better to parallel computing than previous neural network architectures in language processing. In text generation, we get a first glance at this when looking at how each token is processed. We know from the previous chapter that the tokenizer will break the text down into tokens. Each of these input tokens then flows through its own computation path (that's a good first intuition, at least). We can see these individual processing tracks or streams in Figure 3-8.

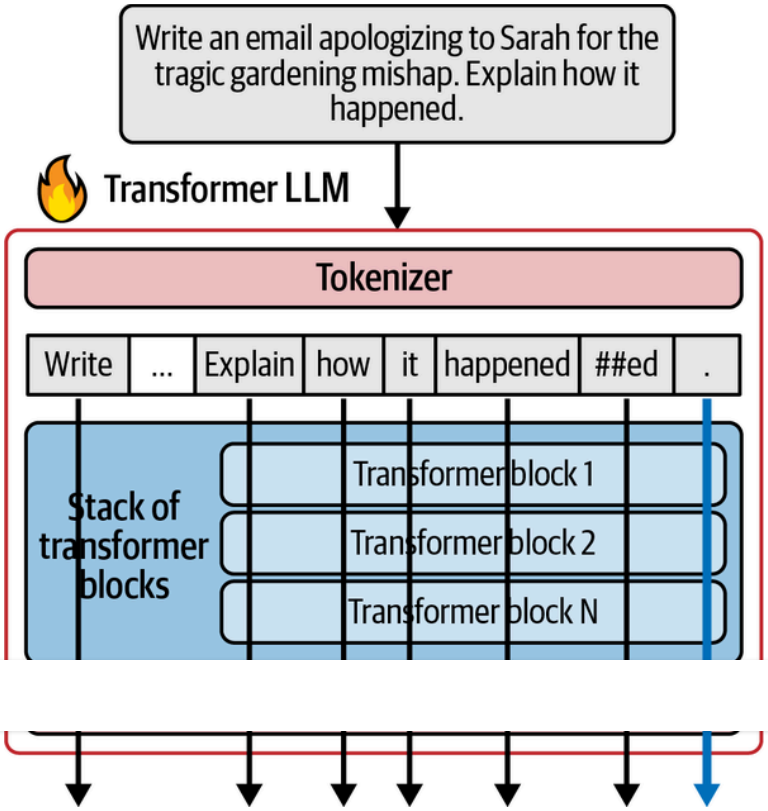

Figure 3-8. Each token is processed through its own stream of computation (with some interaction between them in attention steps, as we'll later see).

Current Transformer models have a limit for how many tokens they can process at once. That limit is 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.

Each of the token streams starts with an input vector (the embedding vector and some positional information; we'll discuss positional embeddings later in the chapter). At the end of the stream, another vector emerges as the result of the model's processing, as shown in Figure 3-9.

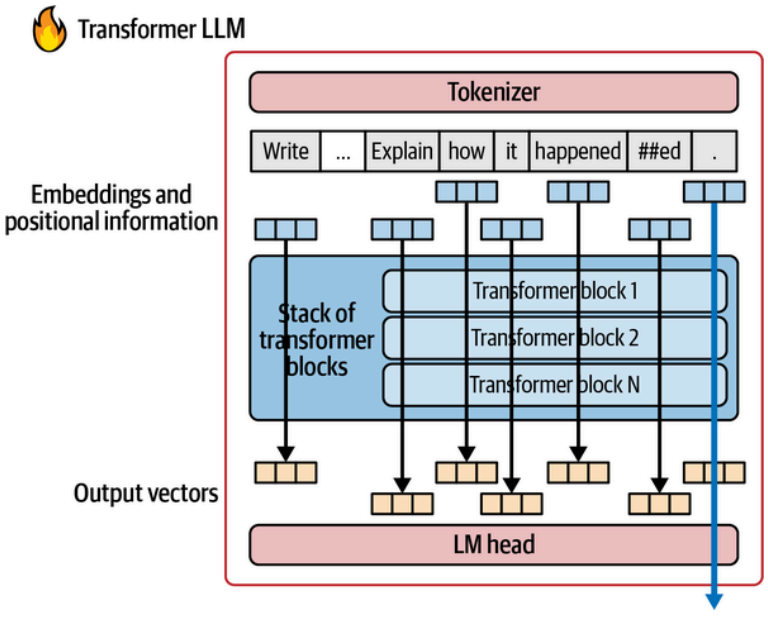

Figure 3-9. Each processing stream takes a vector as input and produces a final resulting vector of the same size (often referred to as the model dimension).

For text generation, only the output result of the last stream is used to predict the next token. That output vector is the only input into the LM

head as it calculates the probabilities of the next token.

You may wonder why we go through the trouble of calculating all the token streams if we're discarding the outputs of all but the last token. The answer is that the calculations of the previous streams are required and used in calculating the final stream. Yes, we're not using their final output vector, but we use earlier outputs (in each Transformer block) in the Transformer block's attention mechanism.

If you're following along the code examples, recall that the output of `lm_head` was of the 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.

We 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])
```

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 we give the model the ability to cache the results of the previous calculation (especially some of the specific vectors in the attention mechanism), we no longer need to repeat the calculations of the previous streams. This time the only needed calculation is for the last stream. This is an optimization technique called the keys and values (kv) cache and it provides a significant speedup of the generation process. Keys and values are some of the central components of the attention mechanism as we'll see later in this chapter.

Figure 3-10 shows how when generating the second token, only one processing stream is active as we cache the results of the previous streams.

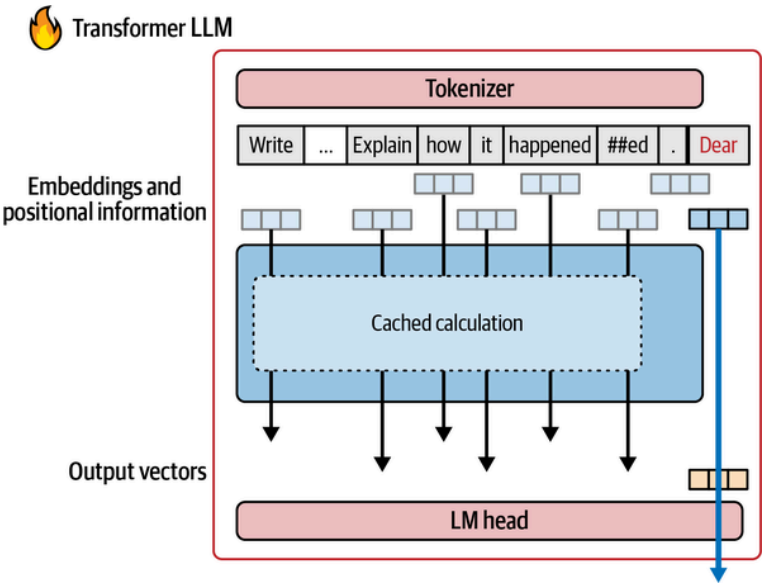

Figure 3-10. When generating text, it's important to cache the computation results of previous tokens instead of repeating the same calculation over and over again.

In Hugging Face Transformers, cache is enabled by default. We can disable it by setting `use_cache` to `False`. We can see the difference in speed by asking for a long generation, and timing the generation with and without caching:

```
prompt = "Write a very long email apologizing to Sarah for the tragic gardening mishap. Explai
# Tokenize the input prompt
input_ids = tokenizer(prompt, return_tensors="pt").input_ids
input_ids = input_ids.to("cuda")
```

Then we time how long it takes to generate 100 tokens with caching. We can use the `%%timeit` magic command in Jupyter or Colab to time how long the execution takes (it runs the command several times and gets the average):

```
%%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. How long would that take if we disable the cache, however?

```
%%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. A dramatic difference. In fact, from a user experience standpoint, even the four-second generation time tends to be a long time to wait for a user that's staring at a screen and waiting for an

output from the model. This is one reason why LLM APIs stream the output tokens as the model generates them instead of waiting for the entire generation to be completed.

Inside the Transformer Block

We can now talk about where the vast majority of processing happens: the Transformer blocks. As Figure 3-11 shows, Transformer LLMs are composed of a series Transformer blocks (often in the range of six in the original Transformer paper, to over a hundred in many large LLMs). Each block processes its inputs, then passes the results of its processing to the next block.

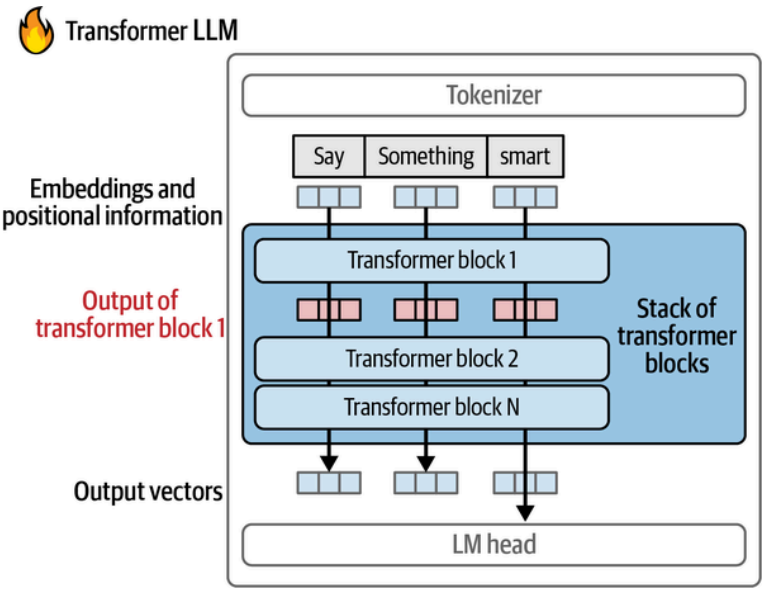

Figure 3-11. The bulk of the Transformer LLM processing happens inside a series of Transformer blocks, each handing the result of its processing as input to the subsequent block.

A Transformer block, as shown in Figure 3-12, is made up of two successive components:

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

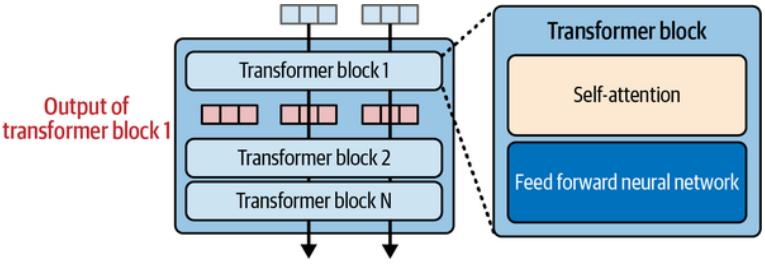

Figure 3-12. A Transformer block is made up of a self-attention layer and a feedforward neural network.

The feedforward neural network at a glance

A simple example giving the intuition of the feedforward neural network would be if we pass the simple input "The Shawshank" to a language model, with the expectation that it will generate "Redemption" as the most probable next word (in reference to the film from 1994).

The feedforward neural network (collectively in all the model layers) is the source of this information, as Figure 3-13 shows. When the model was successfully trained to model a massive text archive (which included many mentions of "The Shawshank Redemption"), it learned and stored the information (and behaviors) that make it succeed at this task.

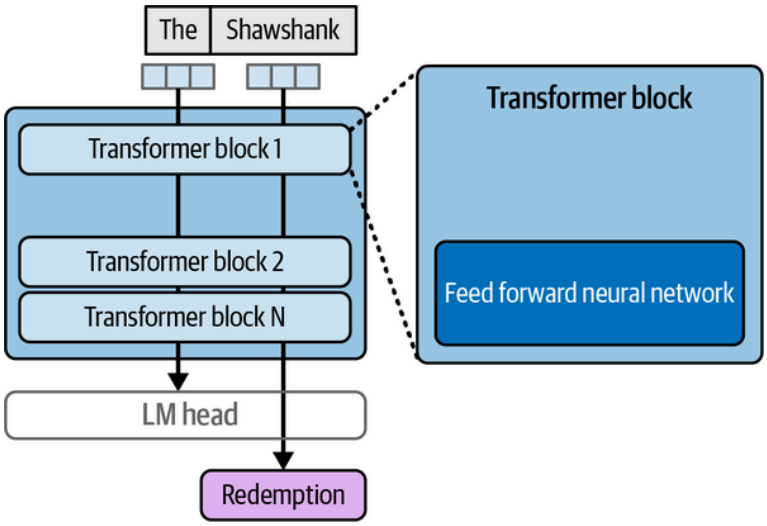

Figure 3-13. The feed-forward neural network component of a Transformer block likely does the majority of the model's memorization and interpolation.

For an LLM to be successfully trained, it needs to memorize a lot of information. But it is not simply a large database. Memorization is only one ingredient in the recipe of impressive text generation. The model is able to use this same machinery to interpolate between data points and more complex patterns to be able to generalize—which means doing well on inputs it hadn't seen in the past and were not in its training dataset.

NOTE

When you use a modern commercial LLM, the outputs you get are not the ones mentioned above in the strict meaning of a "language model." Passing "The Shawshank" to a chat LLM like GPT-4 produces an output:

NOTE

"The Shawshank Redemption" is a 1994 film directed by Frank Darabont and is based on the novella "Rita Hayworth and Shawshank Redemption" written by Stephen King. ...etc.

NOTE

This is because raw language models (like GPT-3) are difficult for people to properly utilize. This is why the language model is then trained on instruction-tuning and human preference and feedback fine-tuning to match people's expectations of what the model should output

---

The attention layer at a glance

Context is vital in order to properly model language. Simple memorization and interpolation based on the previous token can only take us so far. We know that because this was one of the leading approaches to build language models before neural networks (see Chapter 3,"N-gram

Language Models" of *Speech and Language Processing* by Daniel Jurafsky and James H. Martin).

Attention is a mechanism that helps the model incorporate context as it's processing a specific token. Think of the following prompt:

"The dog chased the squirrel because **it**"

For the model to predict what comes after "it," it needs to know what "it" refers to. Does it refer to the dog or the squirrel?

In a trained Transformer LLM, the attention mechanism makes that determination. Attention adds information from the context into the representation of the "it" token. We can see a simple version of that in Figure 3-14.

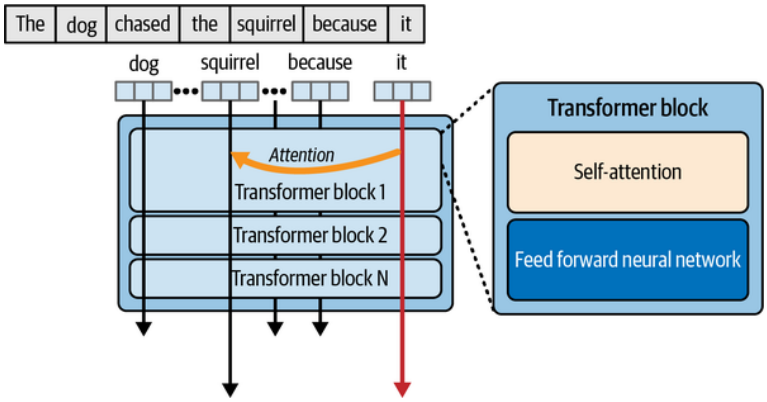

Figure 3-14. The self-attention layer incorporates relevant information from previous positions that help process the current token.

The model does that based on the patterns seen and learned from the training dataset. Perhaps previous sentences also give more clues, like for example referring to the dog as "she" thus making it clear that "it" refers to the squirrel.

Attention is all you need

It is worth diving deeper into the attention mechanism. The most stripped-down version of the mechanism is shown in Figure 3-15. It shows multiple token positions going into the attention layer; the final one is the one being currently processed (the pink arrow). The attention mechanism operates on the input vector at that position. It incorporates relevant information from the context into the vector it produces as the output for that position.

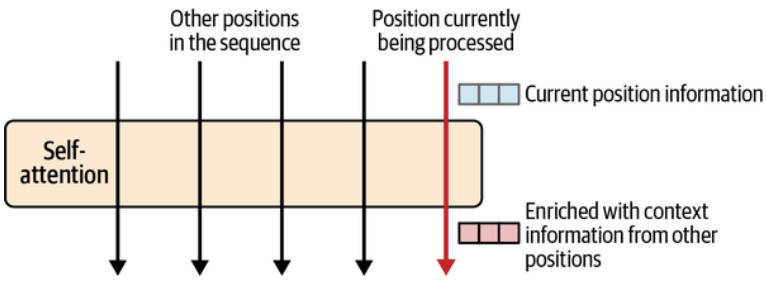

Figure 3-15. A simplified framing of attention: an input sequence and a current position being processed. As we're mainly concerned with this position, the figure shows an input vector and an output vector that incorporates information from the previous elements in the sequence according to the attention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

Figure 3-16 shows these two steps.

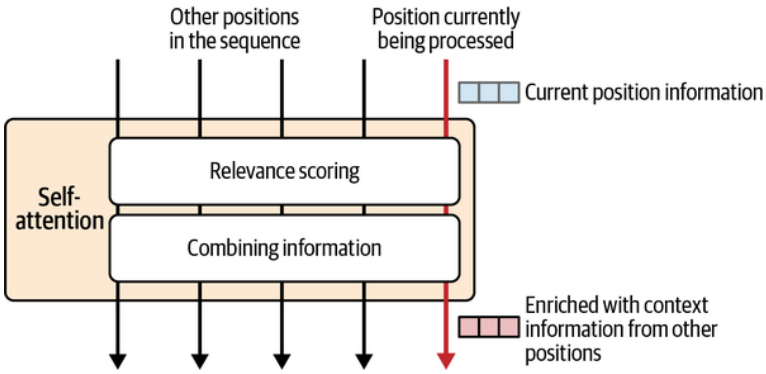

Figure 3-16. Attention is made up of two major steps: relevance scoring for each position, then a step where we combine the information based on those scores.

To give the Transformer more extensive attention capability, the attention mechanism is duplicated and executed multiple times in parallel. Each of these parallel applications of attention is conducted into an *attention head*. This increases the model's capacity to model complex patterns in the input sequence that require paying attention to different patterns at once.

Figure 3-17 shows the intuition of how attention heads run in parallel with a preceding step of splitting information and a later step of combining the results of all the heads.

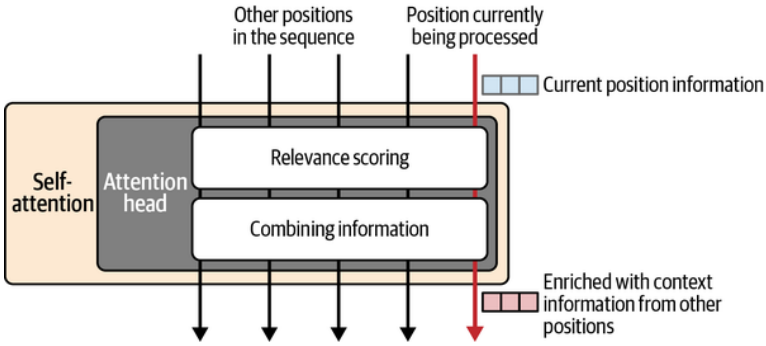

Figure 3-17. We get better LLMs by doing attention multiple times in parallel, increasing the model's capacity to attend to different types of information

How attention is calculated

Let's look at how attention is calculated inside a single attention head. Before we start the calculation, let's observe the following as the starting position:

- The attention layer (of a generative LLM) is processing attention for a single position.
- The inputs to the layer are:
  - The vector representation 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 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 the cat—so attention bakes in "cat information" from the cat token.
- The training process produces three projection matrices that produce the components that interact in this calculation:
  - A query projection matrix
  - A key projection matrix
  - A value projection matrix

Figure 3-18 shows the starting position for all of these components before the attention calculations start. For simplicity, let's look at only one attention head because the other heads have identical calculations but with their individual projection matrices.

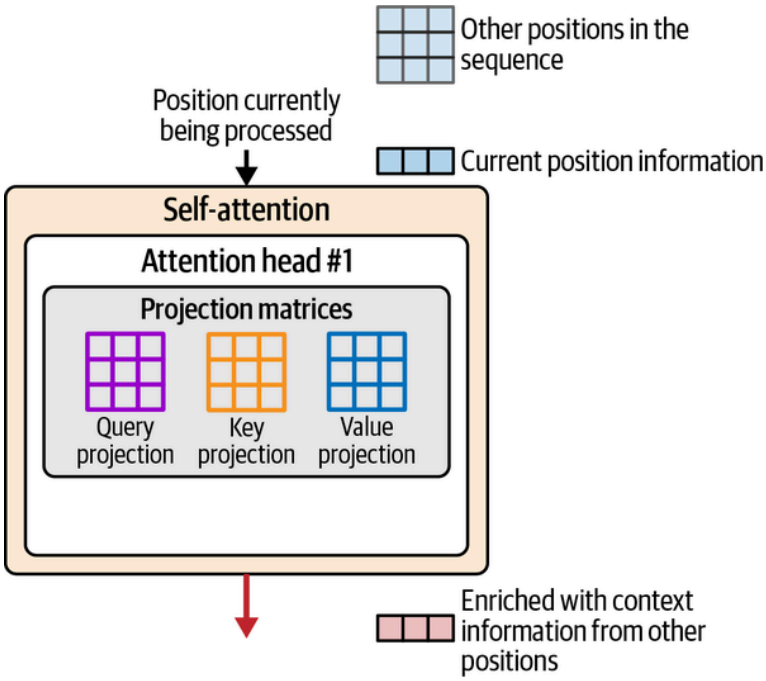

Figure 3-18. Before starting the self-attention calculation, we have the inputs to the layer and projection matrices for queries, keys, and values.

Attention starts by multiplying the inputs by the projection matrices to create three new matrices. These are called the queries, keys, and values matrices. These matrices contain the information of the input tokens projected to three different spaces that help carry out the two steps of attention:

1. Relevance scoring
2. Combining information

Figure 3-19 shows these three new matrices, and how the bottom row of all three matrices is associated with the current position while the rows above it are associated with the previous positions.

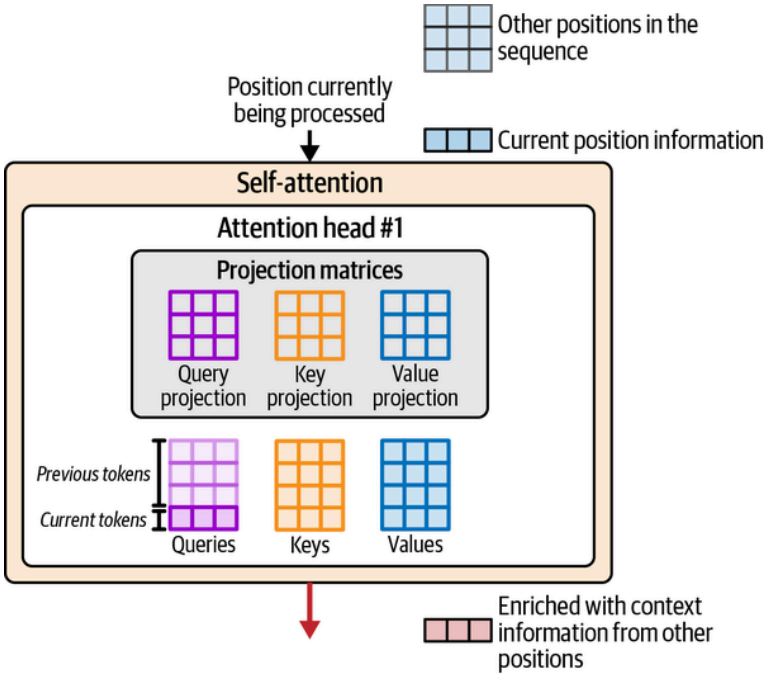

Figure 3-19. Attention is carried out by the interaction of the queries, keys, and values matrices. Those are produced by multiplying the layer's inputs with the projection matrices.

Self-attention: Relevance scoring

In a generative Transformer, we're generating one token at a time. This means we're processing one position at a time. So the attention mechanism here is only concerned with this one position, and how information from other positions can be pulled in to inform this position.

The relevance scoring step of attention is conducted by multiplying the query vector of the current position with the keys matrix. This produces a score stating how relevant each previous token is. Passing that by a softmax operation normalizes these scores so they sum up to 1. Figure 3-20 shows the relevance score resulting from this calculation.

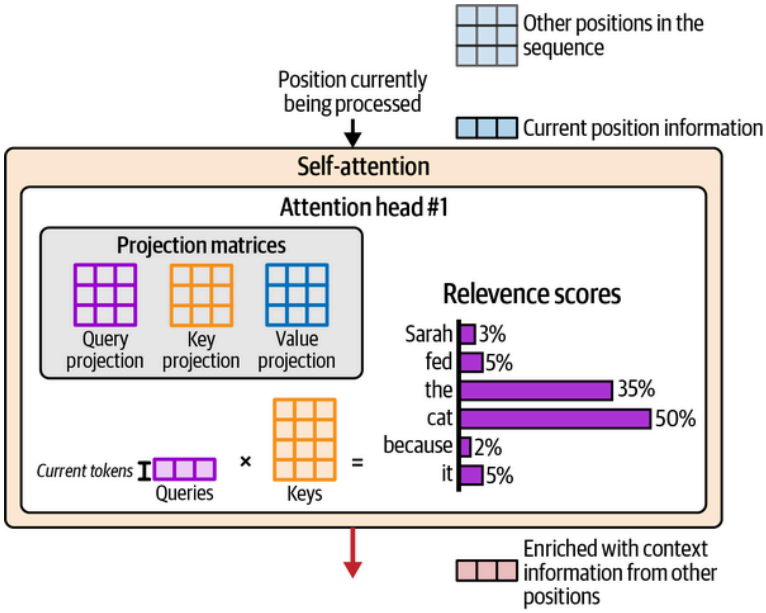

Figure 3-20. Scoring the relevance of previous tokens is accomplished by multiplying the query associated with the current position with the keys matrix.

Self-attention: Combining information

Now that we have the relevance scores, we multiply the value vector associated with each token by that token's score. Summing up those resulting vectors produces the output of this attention step, as we see in Figure 3-21.

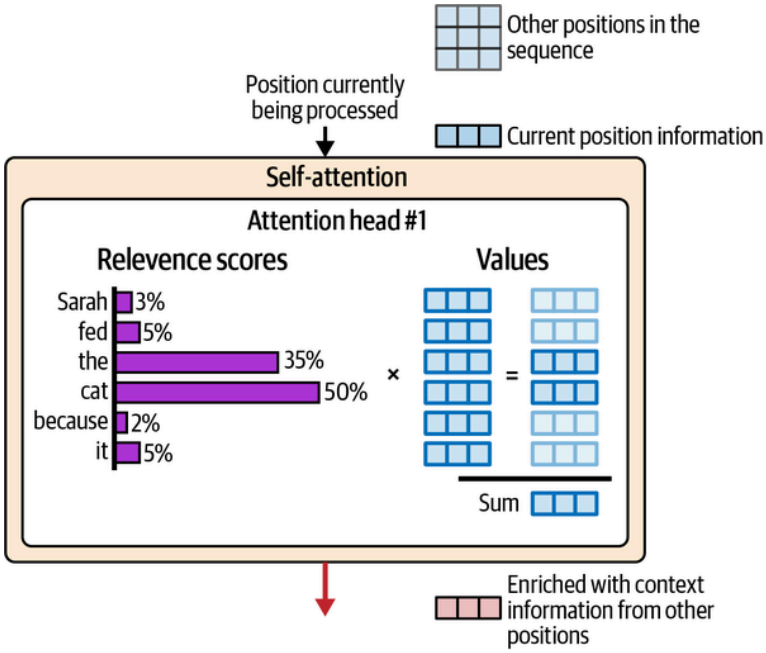

Figure 3-21. Attention combines the relevant information of previous positions by multiplying their relevance scores by their respective value vectors.

Recent Improvements to the Transformer Architecture

Since the release of the Transformer architecture, much work has been done to improve it and create better models. This spans training on larger datasets and optimizations for the training process and learning rates to use, but it also extends to the architecture itself. At the time of writing, a lot of the ideas of the original Transformer stand unchanged. There are a few architectural ideas that have proved to be valuable. They contribute to the performance of more recent Transformer models like Llama 2. In this final section of the chapter, we go over a number of the important recent developments of the Transformer architecture.

More Efficient Attention

The area that gets the most focus from the research community is the attention layer of the Transformer. This is because the attention calculation is the most computationally expensive part of the process.

Local/sparse attention

As Transformers started getting larger, ideas like sparse attention ("Generating long sequences with sparse transformers") and sliding window attention ("Longformer: The long-document transformer") provided improvements for the efficiency of the attention calculation. Sparse attention limits the context of previous tokens that the model can attend to, as we can see in Figure 3-22.

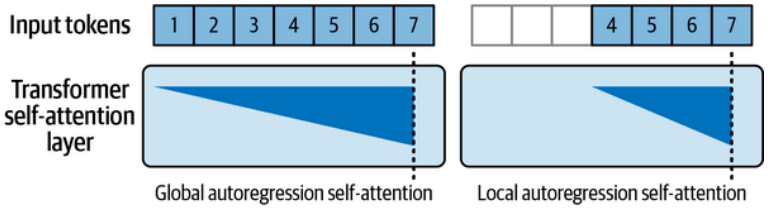

Figure 3-22. Local attention boosts performance by only paying attention to a small number of previous positions.

One model that incorporates such a mechanism is GPT-3. But it does not use that for all the Transformer blocks—the quality of the generation would vastly degrade if the model could only see a small number of previous tokens. The GPT-3 architecture interweaved full-attention and efficient-attention Transformer blocks. So the Transformer blocks alternate between full attention (e.g., blocks 1 and 3) and sparse attention (e.g., blocks 2 and 4).

To demonstrate different kinds of attention, review Figure 3-23, which shows how different attention mechanisms work. Each figure shows which previous tokens (light blue) can be attended to when processing the current token (in dark blue).

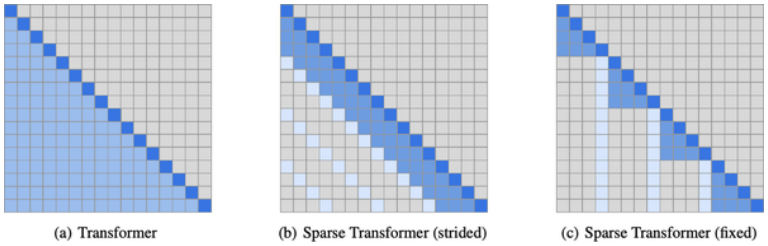

Figure 3-23. Full attention versus sparse attention. Figure 3-26 explains the coloring. (Source: "Generating long sequences with sparse transformers.")

Each row corresponds to a token being processed. The color coding indicates which tokens the model is able to pay attention to while it's processing the token in the dark blue cell. Figure 3-24 describes this in more clarity.

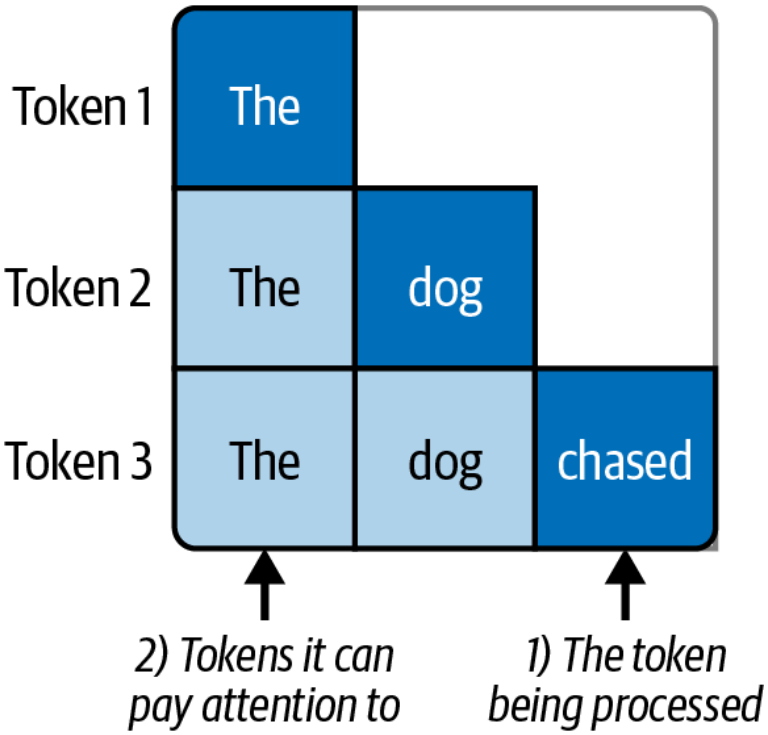

Figure 3-24. Attention figures show which token is being processed, and which previous tokens an attention mechanism allows it to attend to.

This figure also shows the autoregressive nature of decoder Transformer blocks (which make up most text generation models); they can only pay attention to previous tokens. Contrast this to BERT, which can pay attention to both sides (hence the B in BERT stands for bidirectional).

Multi-query and grouped-query attention

A more recent efficient attention tweak to the Transformer is grouped-query attention ("GQA: Training generalized multi-query transformer models from multi-head checkpoints"), which is used by models like Llama 2. Figure 3-25 shows these different types of attention, and the next section continues to explain them.

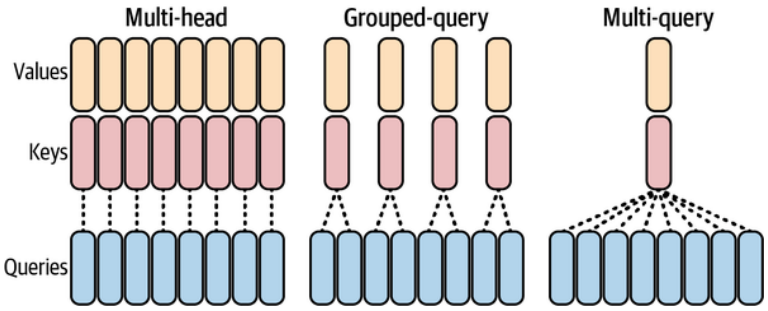

Figure 3-25. A comparison of different kinds of attention: the original multi-head, grouped-query attention, and multi-query attention (source: "Fast transformer decoding: One write-head is all you need").

Grouped-query attention builds on multi-query attention ("Fast transformer decoding: One write-head is all you need"). These methods improve inference scalability of larger models by reducing the size of the matrices involved.

Optimizing attention: From multi-head to multi-query to grouped query

Earlier in the chapter we showed how the Transformer paper described multi-headed attention. The Illustrated Transformer discusses in detail how the queries, keys, and values matrices are used to conduct the attention operation. Figure 3-26 shows how each attention has its own distinct query, key, and value matrices calculated for a given input.

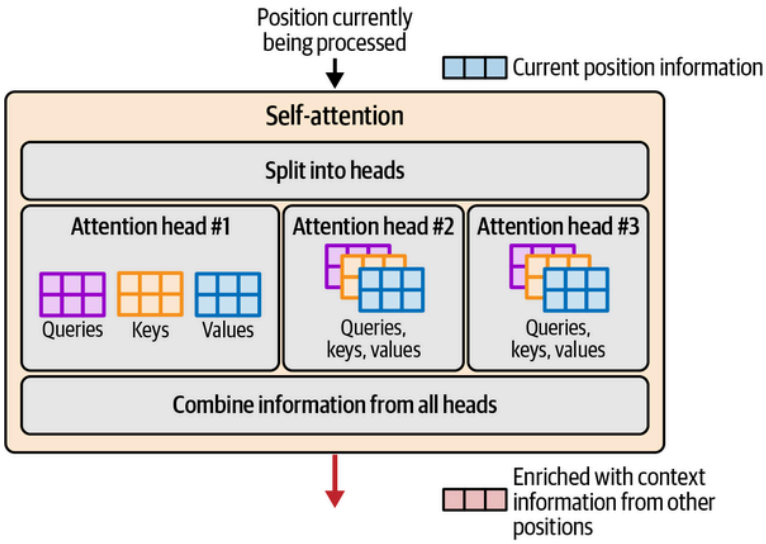

Figure 3-26. Attention is conducted using matrices of queries, keys, and values. In multi-head attention, each head has a distinct version of each of these matrices.

The way that multi-query attention optimizes this is to share the keys and values matrices between all the heads. So the only unique matrices for each head would be the queries matrices, as we can see in Figure 3-27.

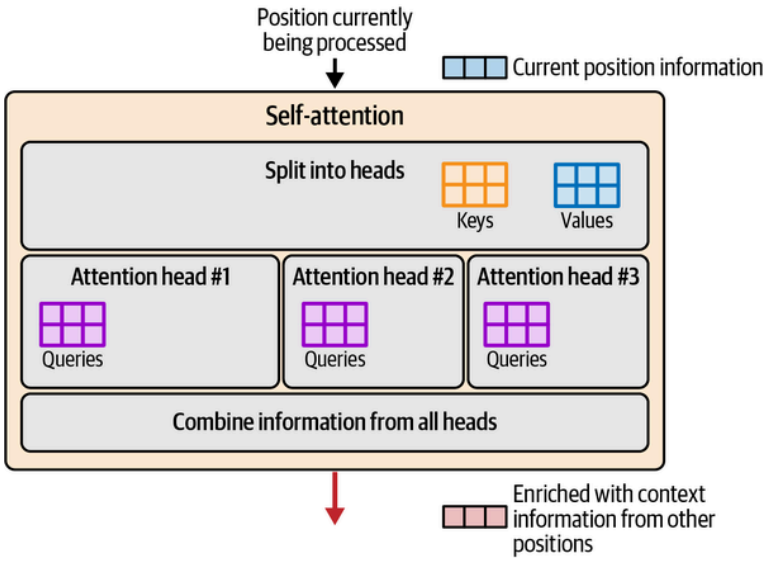

Figure 3-27. Multi-query attention presents a more efficient attention mechanism by sharing the keys and values matrices across all the attention heads.

As model sizes grow, however, this optimization can be too punishing and we can afford to use a little more memory to improve the quality of the models. This is where grouped query comes in. Instead of cutting the number of keys and values matrices to one of each, it allows us to use more (but less than the number of heads). Figure 3-28 shows these groups and how each group of attention heads shares keys and values matrices.

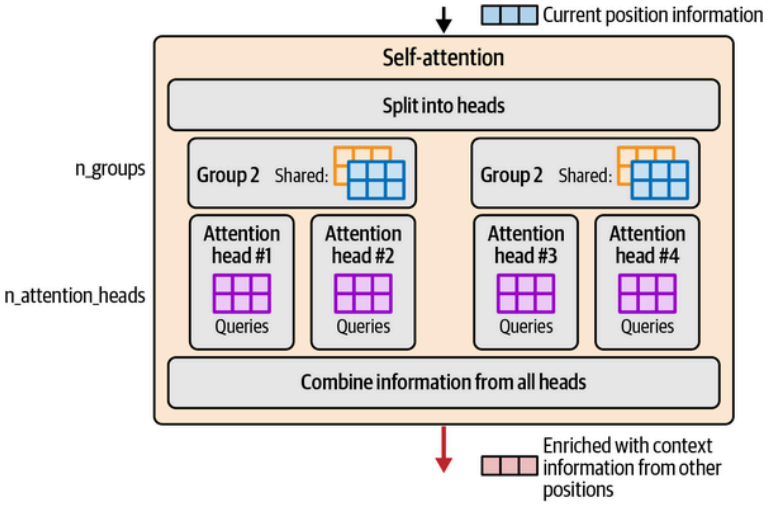

Figure 3-28. Grouped-query attention sacrifices a little bit of the efficiency of multi-query attention in return for a large improvement in quality by allowing multiple groups of shared key/value matrices; each group has its respective set of attention heads.

Flash Attention

Flash Attention is a popular method and implementation that provides significant speedups for both training and inference of Transformer LLMs on GPUs. It speeds up the attention calculation by optimizing what values are loaded and moved between a GPU's shared memory (SRAM) and high bandwidth memory (HBM). It is described in detail in the papers "FlashAttention: Fast and memory-efficient exact attention with IO-awareness" and the subsequent "FlashAttention-2: Faster attention with better parallelism and work partitioning."

The Transformer Block

Recall that the two major components of a Transformer block are an attention layer and a feedforward neural network. A more detailed view of the block would also reveal the residual connections and layer-normalization operations that we can see in Figure 3-29.

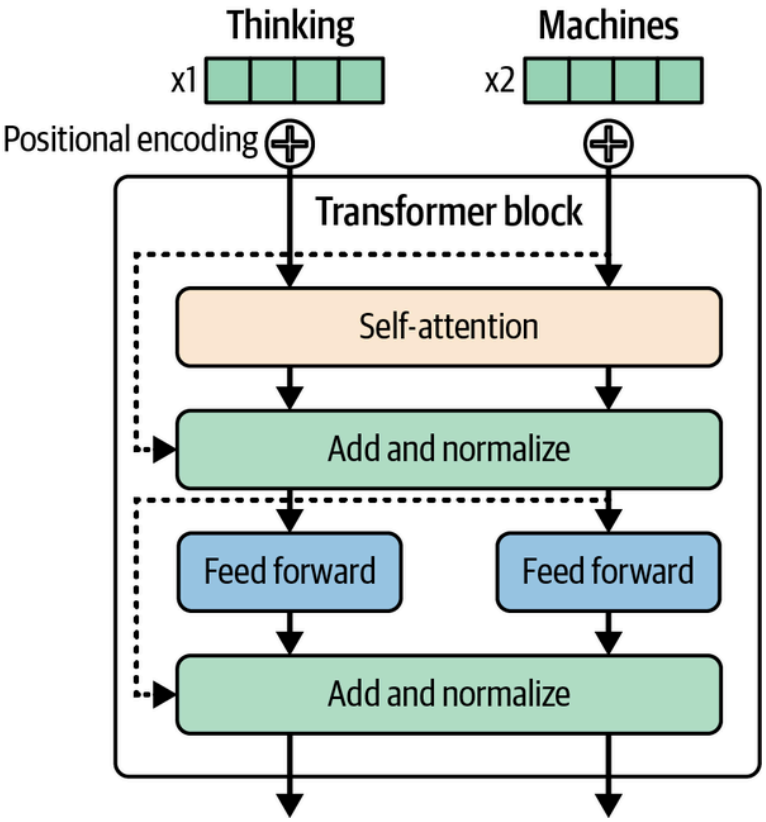

Figure 3-29. A Transformer block from the original Transformer paper.

The latest Transformer models at the time of this writing still retain the major components, yet make a number of tweaks as we can see in Figure 3-30.

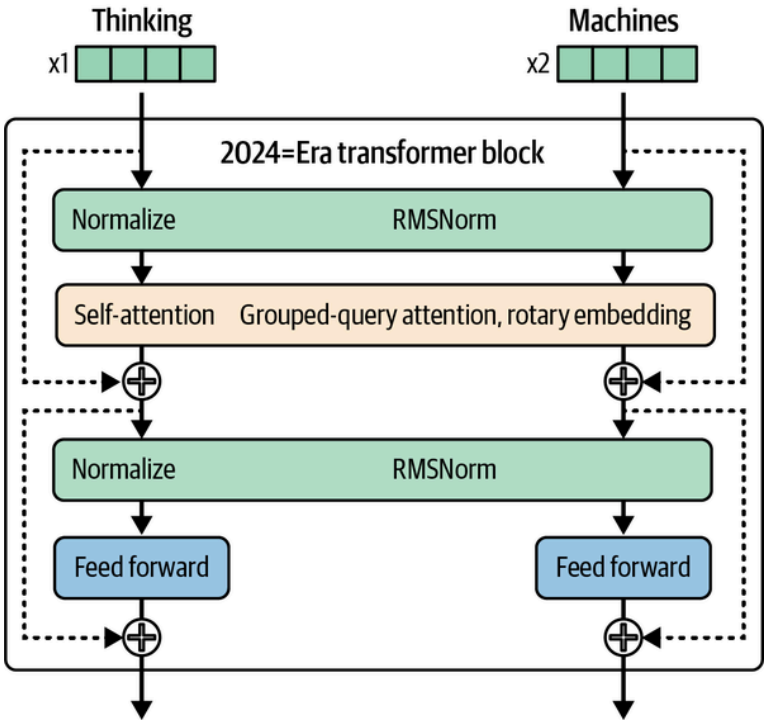

Figure 3-30. The Transformer block of a 2024-era Transformer like Llama 3 features some tweaks like pre-normalization and an attention optimized with grouped-query attention and rotary embeddings.

One of the differences we see in this version of the Transformer block is that normalization happens prior to attention and the feedforward layers. This has been reported to reduce the required training time (read: "On layer normalization in the Transformer architecture"). Another improvement in normalization here is using RMSNorm, which is simpler

and more efficient than the LayerNorm used in the original Transformer (read: "Root mean square layer normalization").

Positional Embeddings (RoPE)

Positional embeddings have been a key component since the original Transformer. They enable the model to keep track of the order of tokens/words in a sequence/sentence, which is an indispensable source of information in language. From the many positional encoding schemes proposed in the past years, rotary positional embeddings (or "RoPE," introduced in "RoFormer: Enhanced Transformer with rotary position embedding") are especially important to point out.

The original Transformer paper and some of the early variants had absolute positional embeddings that, in essence, marked the first token as position 1, the second as position 2...etc. These could either be static methods (where the positional vectors are generated using geometric functions) or learned (where the model training assigns them their values during the learning process). Some challenges arise from such methods when we scale up models, which requires us to find ways to improve their efficiency.

For example, one challenge in efficiently training models with large context is that a lot of documents in the training set are much shorter than that context. It would be inefficient to allocate the entire, say, 4K context to a short 10-word sentence. So during model training, documents are packed together into each context in the training batch, as Figure 3-31 shows.

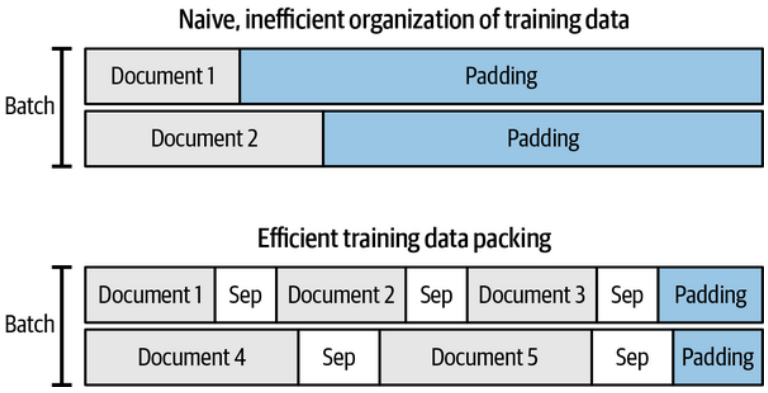

Figure 3-31. Packing is the process of efficiently organizing short training documents into the context. It includes grouping multiple documents in a single context while minimizing the padding at the end of the context.

Learn more about packing by reading "Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance" and watching the great visuals in "Introducing packed BERT for 2X training speed-up in natural language processing."

Positional embedding methods have to adapt to this and other practical considerations. If Document 50, for example, starts at position 50, then we'd be misinforming the model if we tell it that that first token is number 50 and that would affect its performance (because it would assume there's previous context while in reality the earlier tokens belong to a different and unrelated document the model should ignore).

Instead of the static, absolute embeddings that are added in the beginning of the forward pass, rotary embeddings are a method to encode positional information in a way that captures absolute and relative token position information. It is based on the idea of rotating vectors in their embeddings space. In the forward pass, they are added in the attention step, as Figure 3-32 shows.

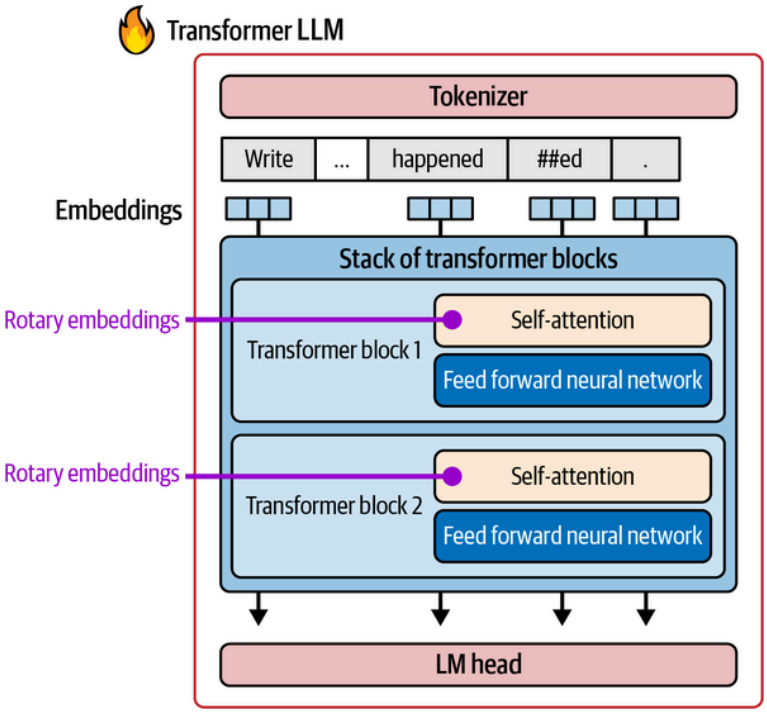

Figure 3-32. Rotary embeddings are applied in the attention step, not at the start of the forward pass.

During the attention process, the positional information is mixed in specifically to the queries and keys matrices just before we multiply them for relevance scoring, as we can see in Figure 3-33.

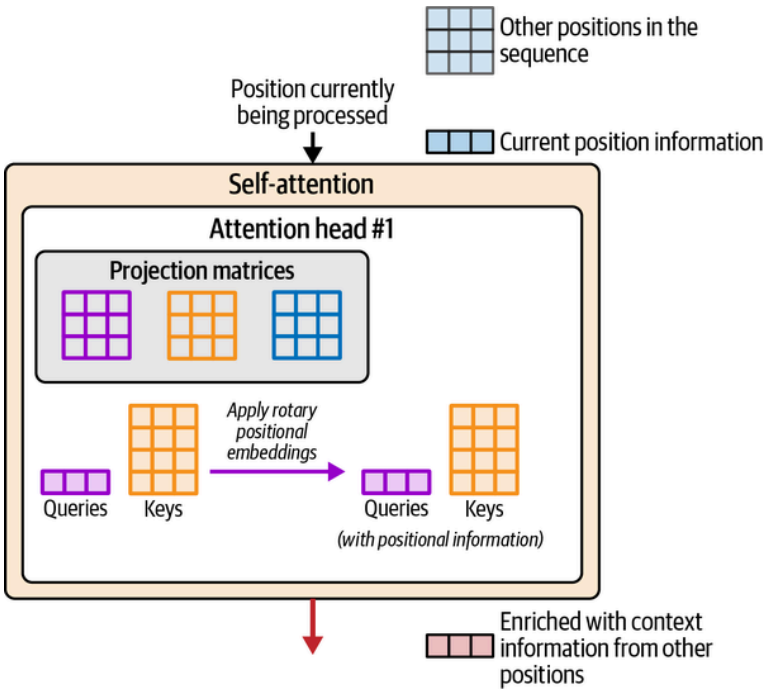

Figure 3-33. Rotary positional embeddings are added to the representation of tokens just before the relevance scoring step in self-attention.

Other Architectural Experiments and Improvements

Many tweaks of the Transformer are proposed and researched on a continuous basis. A survey of transformers highlights a few of the main directions. Transformer architectures are also constantly adapted to domains beyond LLMs. Computer vision is an area where a lot of Transformer architecture research is happening (see: "Transformers in vision: A survey" and "A survey on vision transformer"). Other domains include robots (see "Open X-Embodiment: Robotic learning datasets and RT-X models") and time series (see "Transformers in time series: A survey").

Summary

In this chapter we discussed the main intuitions of Transformers and recent developments that enable the latest Transformer LLMs. We went over many new concepts, so let's break down the key concepts that we discussed in this chapter:

- A Transformer LLM generates *one token at a time.*
- That output token is *appended to the prompt*, then this updated prompt is presented to the model again for another forward pass to generate the next token
- The *three major components* of the Transformer LLM are the tokenizer, a stack of Transformer blocks, and a language modeling head.
- The tokenizer contains the *token vocabulary* for the model. The model has *token embeddings* associated with those embeddings. Breaking the text into tokens and then using the embeddings of these tokens is the first step in the token generation process
- The forward pass flows through all the stages once, *one by one*.
- Near the end of the process, the LM head scores the *probabilities of the next possible token.* Decoding strategies inform which actual token to pick as the output for this generation step (sometimes it's the most probable next token, but not always).
- One reason the Transformer excels is its ability to process tokens in parallel. Each of the input tokens flow into their *individual tracks or streams of processing*. The number of streams is the model's "context size" and this represents the max number of tokens the model can operate on.
- Because Transformer LLMs loop to generate the text one token at a time, it's a good idea to *cache* the processing results of each step so we don't duplicate the processing effort (these results are various matrices within the layers)
- The majority of processing happens within *Transformer blocks*. These are made up of two components. One of them is the f*eedforward neural network*, which is able to store information and make predictions and interpolations from data it was trained on.
- The second major component of a Transformer block is the a*ttention* layer. Attention incorporates contextual information to allow the model to better capture the nuance of language.
- Attention happens in two major steps: (1) scoring relevance and (2) combining relevance.

- A Transformer attention layer conducts several attention operations in parallel, each occurring inside an *attention head*, and their outputs are aggregated to make up the output of the attention layer.
- Attention can be accelerated via sharing the keys and values matrices between all heads, or groups of heads (*grouped-query attention*).
- Methods like *Flash Attention* speed up the attention calculation by optimizing how the operation is done on the different memory systems of a GPU.

Transformers continue to see new developments and proposed tweaks to improve them in different scenarios, including language models and other domains and applications.

In Part II of the book, we will cover some of these practical applications of LLMs. In Chapter 4, we start with text classification, a common task in Language AI. This next chapter serves as an introduction to applying both generative and representation models.