

一. Transformer 模型综述

这一章的主要内容是深入学习 transformer 模型的工作原理。重点放在 generation models 上。

首先，我们可以使用 pipeline() 来调用模型，pipeline() 是 Hugging Face Transformers 库提供的高级 API，它简化了使用预训练模型进行推理的过程。你可以把它想象成一个“一站式”的模型调用工具，它自动处理了从文本预处理到后处理的整个流程。pipeline() 将复杂的模型推理过程封装成简单的函数调用。

```
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=False,
)

# Create a pipeline
generator = pipeline(
    "text-generation", # 任务类型：文本生成
    model=model, # 使用的模型
    tokenizer=tokenizer, # 使用的分词器
    return_full_text=False, # 不返回完整文本（只返回新生成的部分）
    max_new_tokens=50, # 最多生成 50 个新 token
    do_sample=False, # 不使用采样（使用贪婪解码，确定性输出）
)
```

1. 经过训练的 transformer 大语言模型的输入和输出

理解 Transformer 大型语言模型（LLM）行为最常见的方式是将其视为一个接收文本并生成响应文本的软件系统。当“文本输入-文本输出”大模型在大规模的高质量数据集上完成训练后，它就能够生成令人印象深刻且实用的输出。图 3-1 是使用此类模型撰写电子邮件的示例。

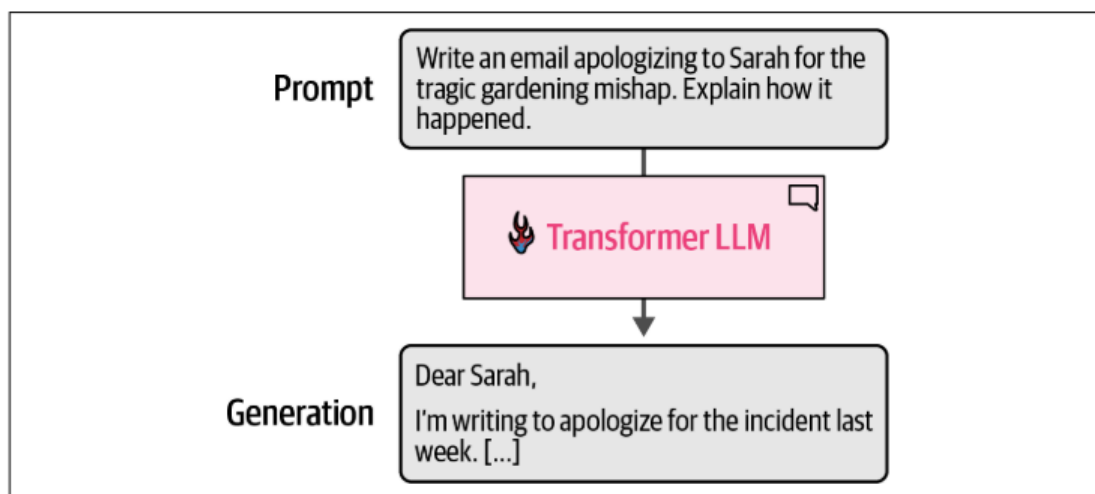


Figure 3-1. At a high level of abstraction, Transformer LLMs take a text prompt and output generated text.

在输出时，并非一次性生成所有文本，而是每次只生成一个 Token，我们在使用 ChatGPT 和 Deepseek 等模型时也能观察到这种生成模式。图 3-2 展示了针对输入提示 prompt 进行 token 生成的四个步骤。每个 token 生成步骤都是模型的一次前向传播（指输入进入神经网络，流经计算图所需的所有计算，最终在计算图的另一端产生输出的过程）。

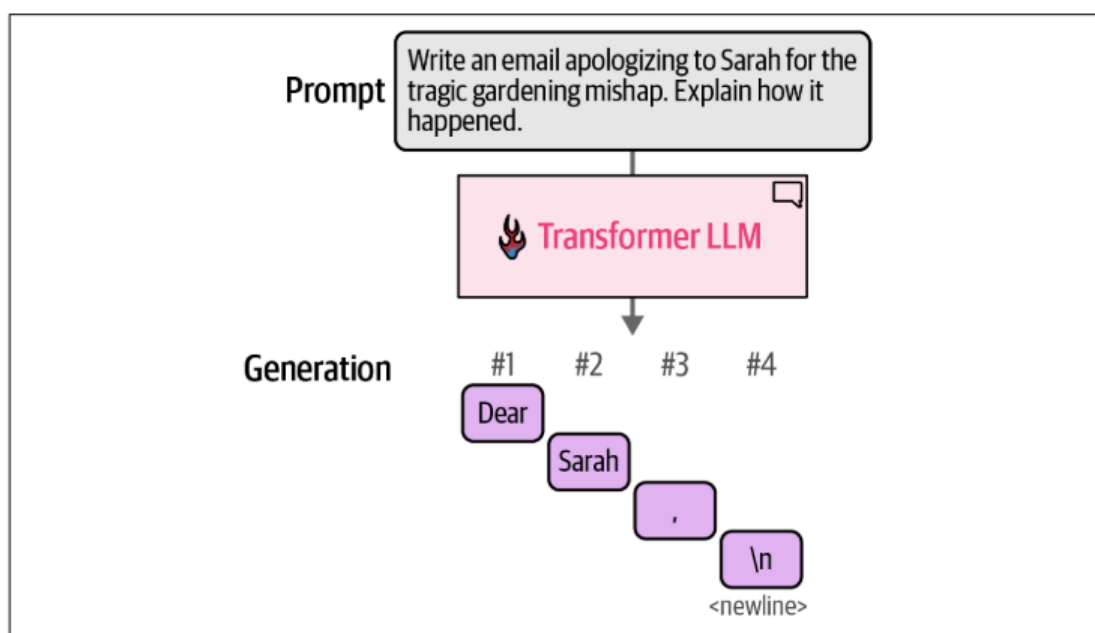


Figure 3-2. Transformer LLMs generate one token at a time, not the entire text at once.

每次生成 Token 后，需要将 token 附加在 prompt 后面，组成 input + output 的输入提示，再将新的 prompt 输入到模型中进行新一轮的前向传播，生成下一个 token，循环往复。也就是说，每一次生成 Token 时模型都会考虑已经生成好的 Token，每一次的生成都是一次完整的模型推理过程。

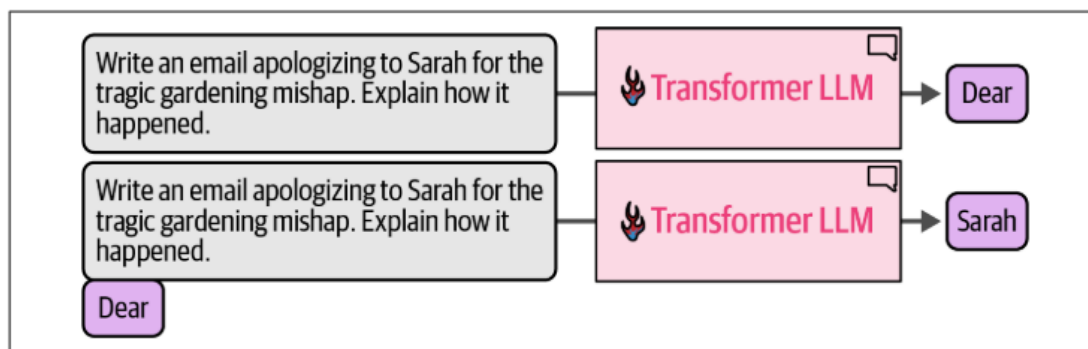


Figure 3-3. An output token is appended to the prompt, then this new text is presented to the model again for another forward pass to generate the next token.

在机器学习中，有一个特定术语用来描述那些消耗自身先前的预测来做出后续预测的模型（例如，模型生成的第一个 token 被用来生成第二个 token）。它们被称为自回归模型（autoregressive models）。

这就是为什么你会听到文本生成的 LLMs 被称为自回归模型。这个术语通常用于区分文本生成模型和文本表示模型（如 BERT），后者不是自回归的。

但是在 Colab 中使用模型生成文本时，这种自回归、逐个生成 token 的过程是在底层发生的：

```

prompt = "Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened."
output = generator(prompt)
print(output[0]['generated_text'])

```

Mention the steps you're taking to prevent it in the future.

Dear Sarah,

I hope this message finds you well. I am writing to express my sincerest apologies for the unfortunate incident that occurred

生成突然停止是因为 token 达到了 50 个，我们在 Pipeline() 中通过 max_new_token 设置成 50 来限制 token 的生成数量。

2. Forward Pass 前向传播的组成部分

Transformer 两个关键的内部组件是 tokenizer 和 language model head (LM head)。图 3-4 展示了这些组件在系统中的位置。分词器之后是**神经网络**：由一系列 Transformer 块组成的堆栈，负责所有的处理工作。这个堆栈之后接着 LM head，它将堆栈的输出转换为概率分数，用于预测最可能的下一个 token。

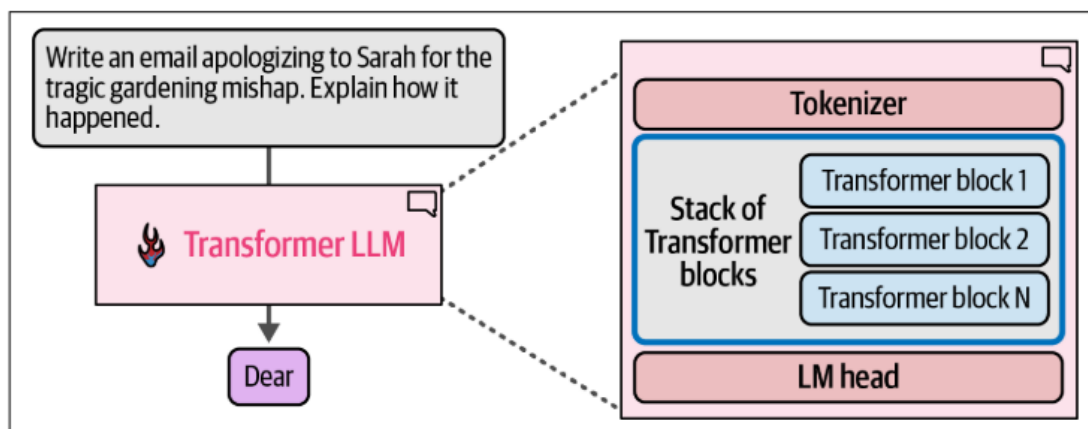


Figure 3-4. A Transformer LLM is made up of a tokenizer, a stack of Transformer blocks, and a language modeling head.

分词器包含一个词表，里面包含了所有 token 和它们的 ID，词表中的每个 token 在模型中都有一个相关联的**向量表示**（**token embeddings**）。图 3-5 展示了一个拥有 50,000 个 token 的词表及其相关联的 token 嵌入。

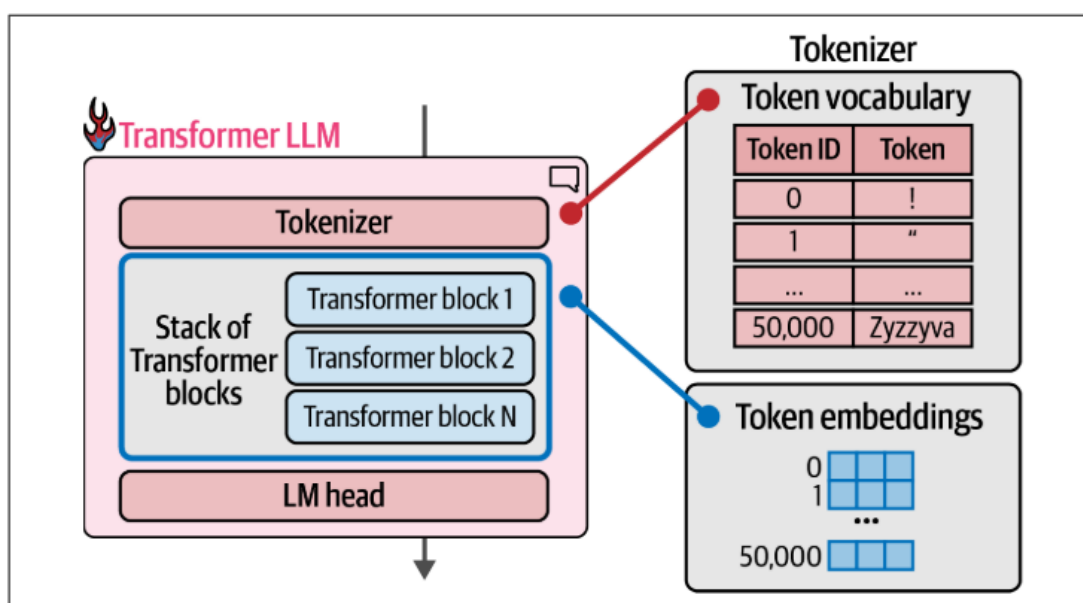


Figure 3-5. The tokenizer has a vocabulary of 50,000 tokens. The model has token embeddings associated with those embeddings.

计算流程按照箭头从上到下进行。对于每个生成的 token 处理过程按顺序依次流过堆栈中的每个 Transformer 块，然后到达 LM head，最终输出下一个 token 的概率分布，如图 3-6 所示。

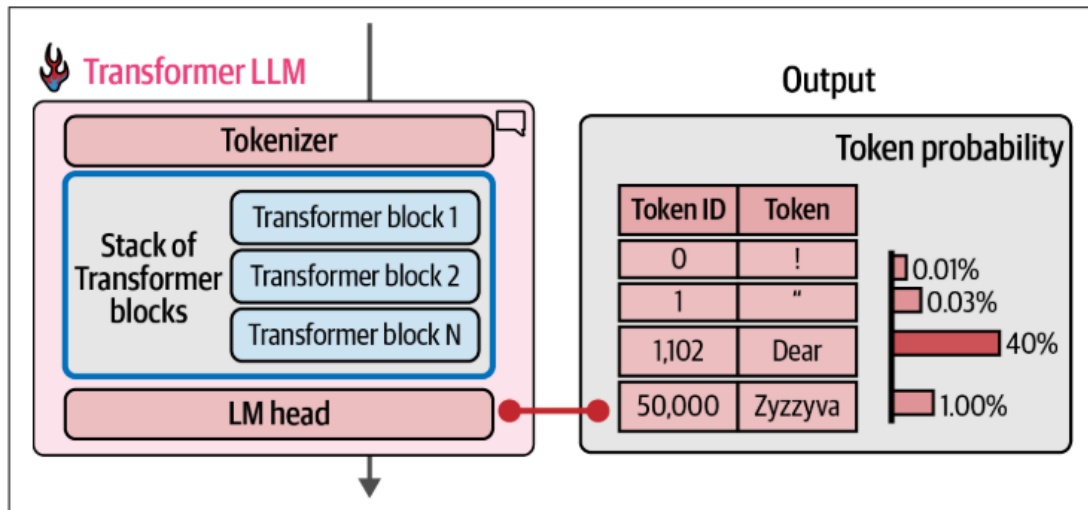


Figure 3-6. At the end of the forward pass, the model predicts a probability score for each token in the vocabulary.

LM head 本身是一个简单的神经网络层。它是可以附加到 Transformer 块堆栈上的多种可能“头”之一，用于构建不同类型的系统。其他类型的 Transformer 头包括序列分类头和 token 分类头。

可以通过打印模型来显示内部层级的顺序：

```
print(model)

Phi3ForCausalLM(
  (model): Phi3Model(
    (embed_tokens): Embedding(32064, 3072, padding_idx=32000)
    (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)
        )
        (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((3072,), eps=1e-05)
        (post_attention_layernorm): Phi3RMSNorm((3072,), eps=1e-05)
        (resid_attn_dropout): Dropout(p=0.0, inplace=False)
        (resid_mlp_dropout): Dropout(p=0.0, inplace=False)
      )
    )
    (norm): Phi3RMSNorm((3072,), eps=1e-05)
    (rotary_emb): Phi3RotaryEmbedding()
  )
  (lm_head): Linear(in_features=3072, out_features=32064, bias=False)
)
```

通过观察这个结构，我们可以注意到以下重点：

- ① 模型有各个嵌套层次。两个主要部分为 `model` 和 `lm_head`。
- ② 在 `Phi3Model` 模型内部，我们可以看到嵌入矩阵 `embed_tokens` 及其维度。它包含 32,064 个 token，每个 token 的向量大小为 3,072 维。
- ③ 下一个主要组件是 Transformer 解码器层的堆栈。它包含 32 个 `Phi3DecoderLayer` 类型的块。
- ④ 每个 Transformer 块都包含一个注意力层和一个前馈神经网络（也称为 `mlp` 或多层感知机）。
- ⑤ 最后，我们看到 `lm_head` 接收一个维度为 3,072 的向量，并输出维度等于词表大小（32064 个）的向量。该输出有 32064 维，每一维对应每个 token 的概率分数。

`embed_tokens`：将 token ID 转换为高维向量；

注意力层：处理 token 间的依赖关系；

MLP 层：进行非线性变换和特征提取；

`lm_head`：将 3,072 维特征映射到 32,064 维概率分布；

3. 从概率分布中选择单个 Token（采样/解码）

输出 token 的概率分布后，从概率分布中选择单个 token 的方法被称为解码策略（`decoding strategy`）。

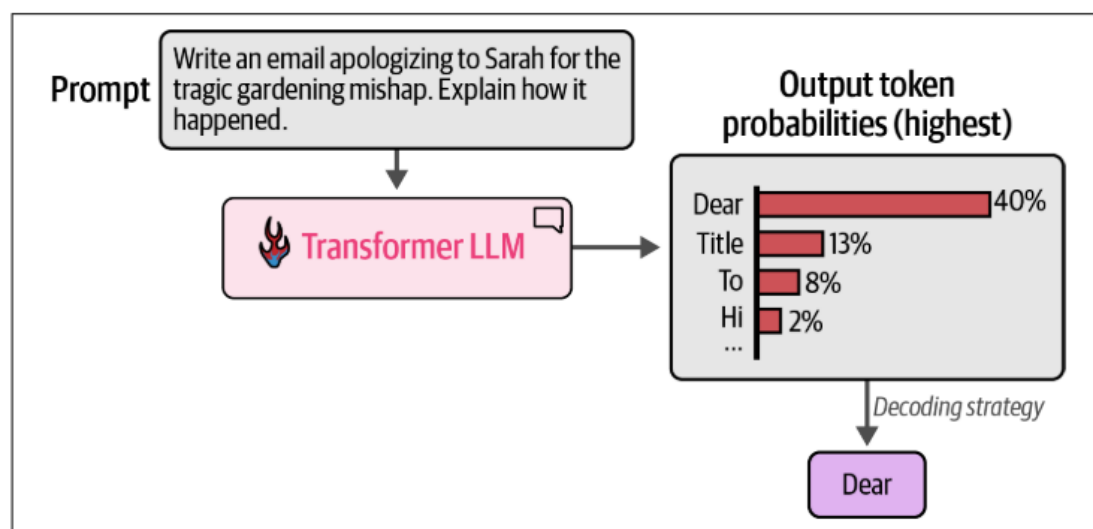


Figure 3-7. The tokens with the highest probability after the model's forward pass. Our decoding strategy decides which of the tokens to output by sampling based on the probabilities.

最简单的解码策略是选择概率分数最高的 token。但在实践中往往不能产生最佳输出。更好的方法是添加一些随机性，有时选择第二或第三高概率的 token，因

为这种做法符合基于概率分数从概率分布中进行采样的思想。

对于图 3-7 中的示例来说，这意味着如果 token“Dear”有 40%的概率成为下一个 token，那么它就有 40%的机会被选中（而不是 greedy decoding 贪婪解码，后者会因为它的分数最高而直接选择它）。通过这种方法，所有其他 token 也都有机会根据它们的分数被选中。这种随机性可以避免每次生成重复的文本，并且增加了创造性。

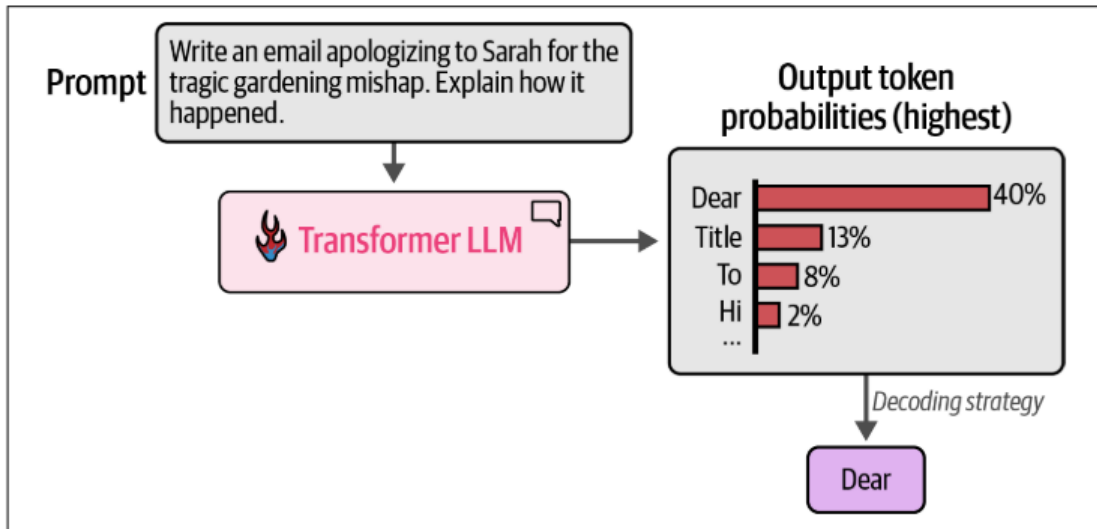



Figure 3-7. The tokens with the highest probability after the model's forward pass. Our decoding strategy decides which of the tokens to output by sampling based on the probabilities.

如果在 LLM 中将参数 temperature 设置为 0 会发生贪婪解码的情况。我们将在第 6 章介绍 temperature。

代码演示过程：

```
 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])
```

lm_head_output 的形状是 [1, 6, 32064]。1 是批处理大小 (batch size)，表示

同时处理 1 个序列；6 是序列长度，当前已生成 6 个 token；32064 是词表大小，每个位置对应所有可能 token 的概率分数。

```
token_id = lm_head_output[0, -1].argmax(-1) # 访问最后生成的token的概率分布
                                                # 这是32064个token的概率得分列表
                                                # 取概率最高的token

tokenizer.decode(token_id) # 解码

'Paris'
```

4. 并行 token 处理与上下文长度

Transformer 最引人注目的特性之一是，与语言处理中先前的神经网络架构相比，它们更适合并行计算，能更好地发挥 GPU 的架构优势。在文本生成中，当我们观察每个 token 的处理方式时，就能初步看到这一点。

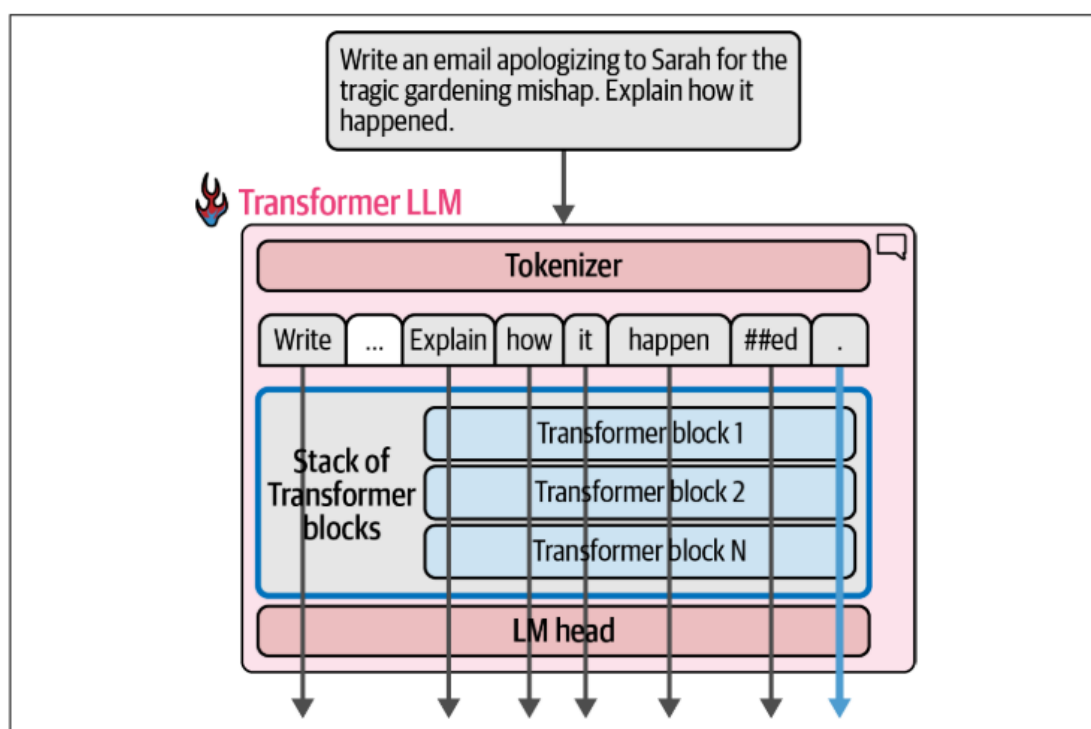


Figure 3-8. Each token is processed through its own stream of computation (with some interaction between them in attention steps, as we'll later see).

目前 Transformer 架构的模型对一次可以并行处理的 token 数量有限，这一限制称为上下文长度。

每个 token 处理流从对应嵌入向量开始，最后输出另一个维度相同的向量作为处理流的结果输出，如图 3-9 所示。

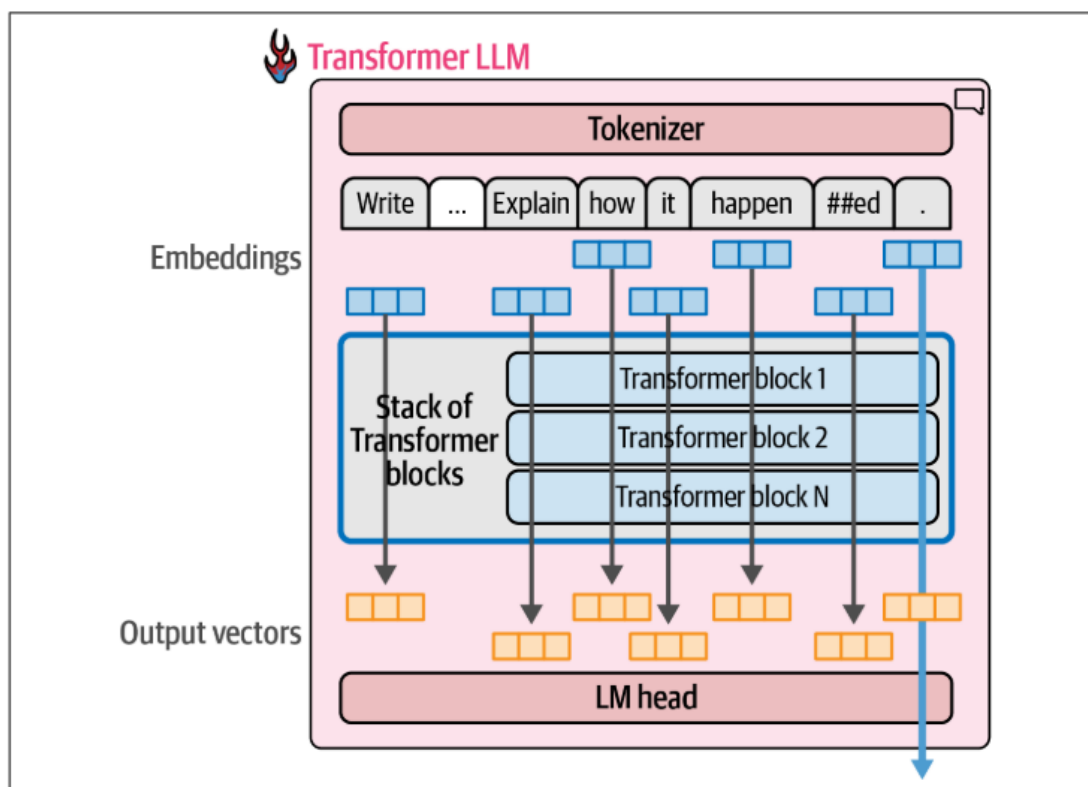


Figure 3-9. Each processing stream takes a vector as input and produces a final resulting vector of the same size (often referred to as the model dimension).

对于文本生成，只有最后一个处理流的输出结果被用于预测下一个 token。该输出向量是 LM head 计算下一个 token 概率时的唯一输入。虽然只使用最终输出，但中间所有计算都是必要的。每个 token 的计算都依赖于序列中所有先前 token 的信息，每一次处理流的输出结果都包含了目前为止前面所有输入的语义信息。对于模型生成的最后一个向量，它包含了输入提示和之前所有生成文本的语义信息。

```

▶ model_output[0].shape # model部分输出的是更新后的嵌入向量
⇨ torch.Size([1, 5, 3072])

lm_head_output.shape # lm_head部分输出的是概率列表
⇨ torch.Size([1, 5, 32064])

```

5. 通过缓存 Keys 和 Values 来加快生成速度

在生成第二个 token 时，我们只是将输出 token 附加到输入中，并再次通过模型进行前向传播。如果我们让模型能够缓存先前计算的结果，模型就不用再次重复

计算，只需要计算最后一个处理流。这种优化技术被称为键值缓存 (kv cache)，它能显著加速生成过程，不用重复计算已经处理过的 token，计算量不会随序列长度线性增加，是用内存换时间的合理权衡。键 (key) 和值 (value) 是注意力机制的核心组成部分，后续会介绍。

图 3-10 展示了在生成第二个 token 时，由于我们缓存了之前处理流的结果，只有一个处理流处于活动状态。

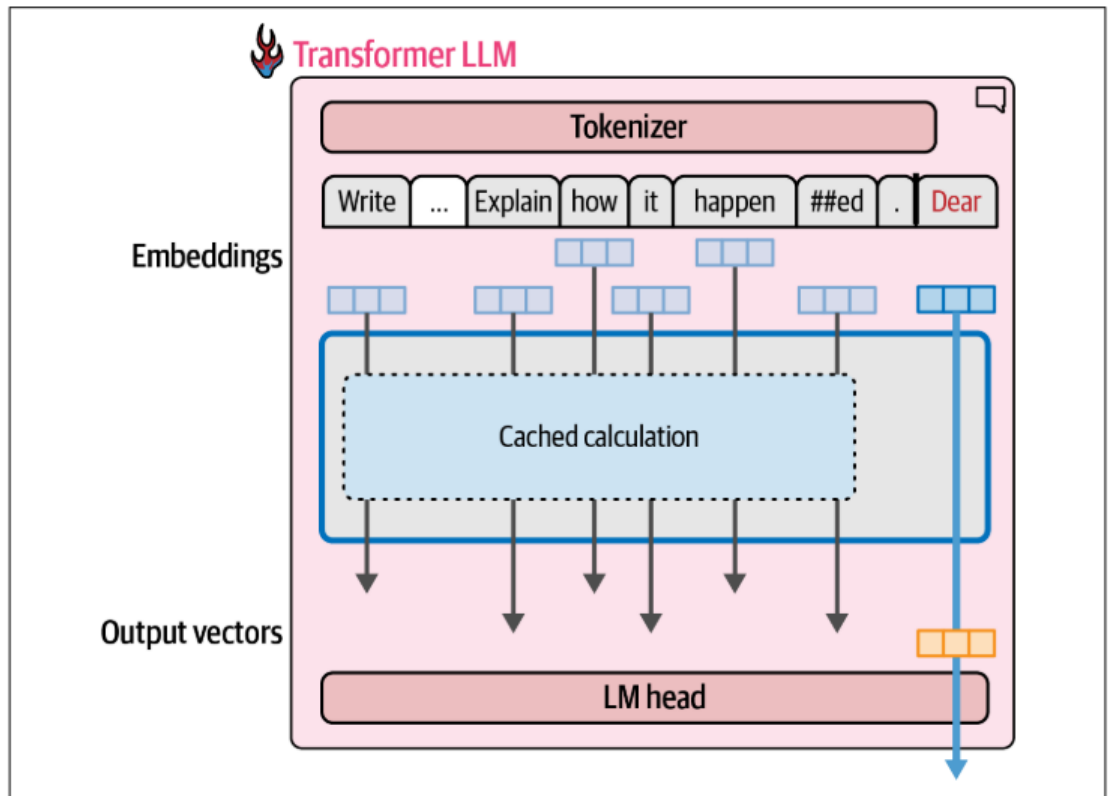


Figure 3-10. When generating text, it's important to cache the computation results of previous tokens instead of repeating the same calculation over and over again.

在 Hugging Face Transformers 中，缓存 (cache) 默认是启用的。我们可以通过将 `use_cache` 设置为 `False` 来禁用它。我们可以通过请求一个长文本生成，并计时比较启用和禁用缓存时的生成速度，来看到速度上的差异：

```
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") # 将数据移动到GPU
```

然后，我们计算使用缓存生成 100 个令牌所需的时间。我们可以在 Jupyter 或 Colab 中使用 `%timeit` 魔术命令来计算执行所需的时间，该命令会多次运行并计算平均时间。

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

The attention mask is not set and cannot be inferred from input because it is not a valid prefix. 4.45 s ± 214 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

在配备 T4 图形处理器的 Colab 上，这一时间为 4.5 秒。然后我们禁用缓存，查看需要多少时间：

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

33.4 s ± 119 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

结果是 33.4 秒。从用户体验的角度来看，对于盯着屏幕等待模型输出的用户来说，即使是四秒钟的时间也往往是很长的时间。这就是为什么 LLM API 在模型生成时流式输出 token，而不是等待整个生成完成的原因之一。

6. Transformer 组块的内部

在 Transformer 内部，处理嵌入向量由 transformer blocks 完成。如图 3-11 所示，Transformer 由一系列 block（块）组成，在提出 Transformer 的论文《Attention is all you need》中，编码器和解码器分别包含 6 个 Transformer blocks，而现在许多大模型通常在 100 blocks 以上。每块处理其输入向量，然后将其处理结果传递给下一块。

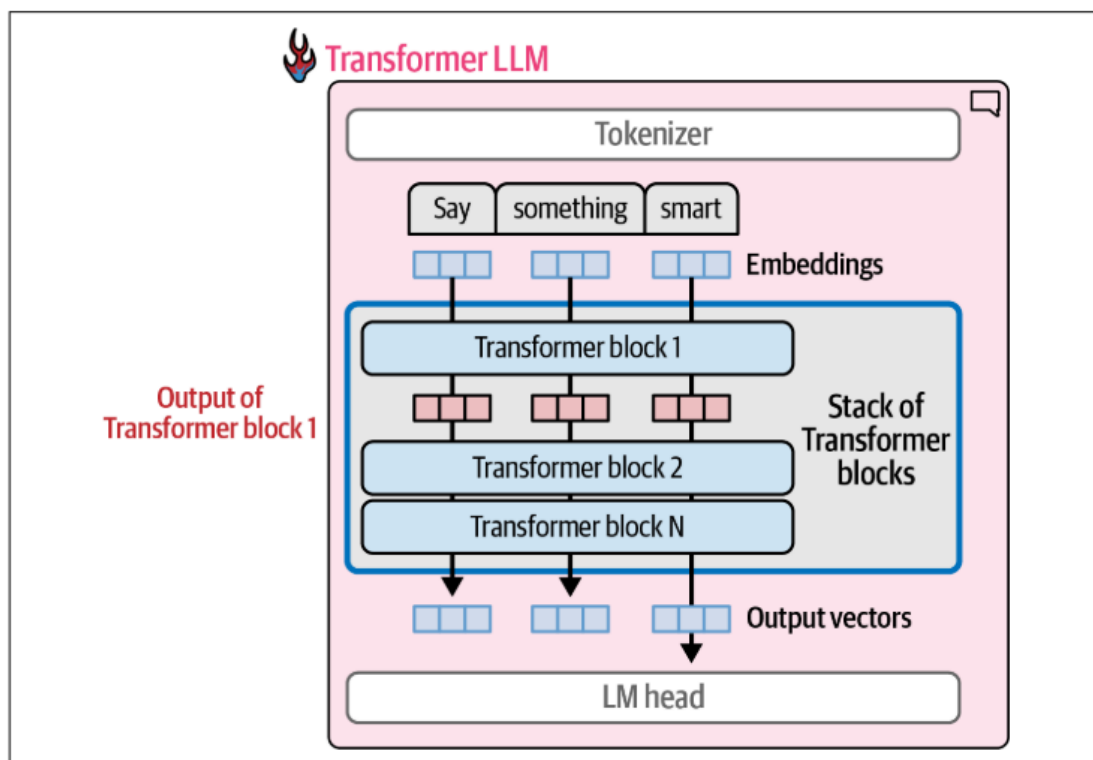


Figure 3-11. The bulk of the Transformer LLM processing happens inside a series of Transformer blocks, each handing the result of its processing as input to the subsequent block.

一个 Transformer block 包含两个主要的子层：多头自注意力层（Multi-Head Self-Attention）和前馈神经网络层（Feed-Forward Network）。

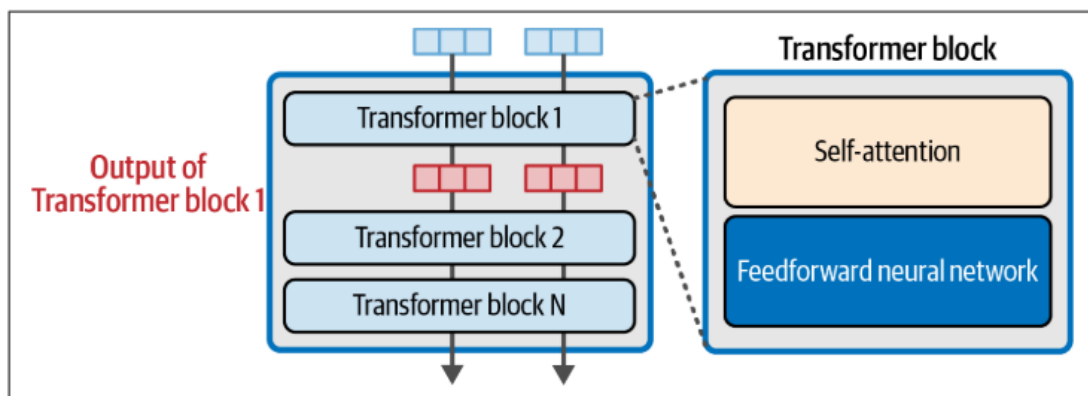


Figure 3-12. A Transformer block is made up of a self-attention layer and a feedforward neural network.

- ① 多头自注意力层的作用是计算输入序列中所有位置之间的关系，通过计算所有 token 的三个参数 Key、Value 和 Query 之间的关系来捕获文本全局关系。
- ② 前馈层容纳了模型的大部分处理能力。通过公式 $\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$ 对注意力输出进行非线性变换（两个线性变换+一个非线性变换）。 \mathbf{W}_1 和 \mathbf{W}_2 是权重矩阵， \mathbf{b}_1 和 \mathbf{b}_2 是偏置向量。权重矩阵用来扩展和压缩结构，引入 ReLU

激活使模型能够学习复杂非线性模式。

什么是前馈神经网络？

前馈神经网络的作用是存储关于世界的事实知识，学习词汇间的统计关联，并基于上下文预测最可能的下一个词。用大规模文本数据训练的过程中，知识会编码到前馈神经网络的权重矩阵中。当向模型输入简单的文本“The Shawshank”，我们期望它生成“Redemption”作为最可能的下一个词，前馈网络的作用就是从训练中学到的知识检索“The Shawshank”的关联信息。前馈网络就是 LLM 的“知识心脏”，它存储了模型在训练中学到的所有事实和模式。

什么是注意力层？

注意力是一种机制，帮助模型在处理特定 token 时融入上下文。例如下面这句话：“The dog chased the squirrel because it”为了让模型预测“it”之后的内容，它需要知道“it”指代什么。它指的是狗还是松鼠？训练好的 LLM 会做出决定，注意力机制将上下文中的信息添加到“it”这个 token 的表示中，如图 3-14 所示。

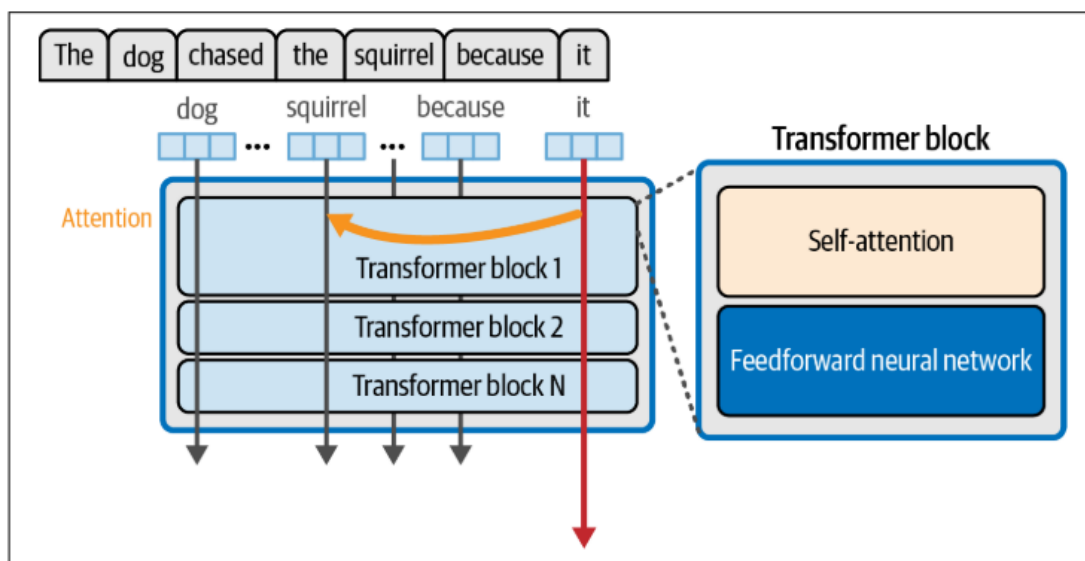


Figure 3-14. The self-attention layer incorporates relevant information from previous positions that help process the current token.

简单来说，注意力机制就像 3-15 所示的那样，进入关注层的有多个 token，粉色箭头是当前正在处理的位置，注意力机制对该位置的 token 起作用时，会将上下文中的相关信息合并到该位置输出的生成向量中。

注意力机制主要有两个步骤：1. 对不同位置的 token 和当前正在处理的 token 进行相关性评估（隐藏在粉色箭头中，后续会说到）；2. 使用这些评估结果，将来自不同位置的信息合并成单个输出向量。

Figure 3-16 shows these two steps.

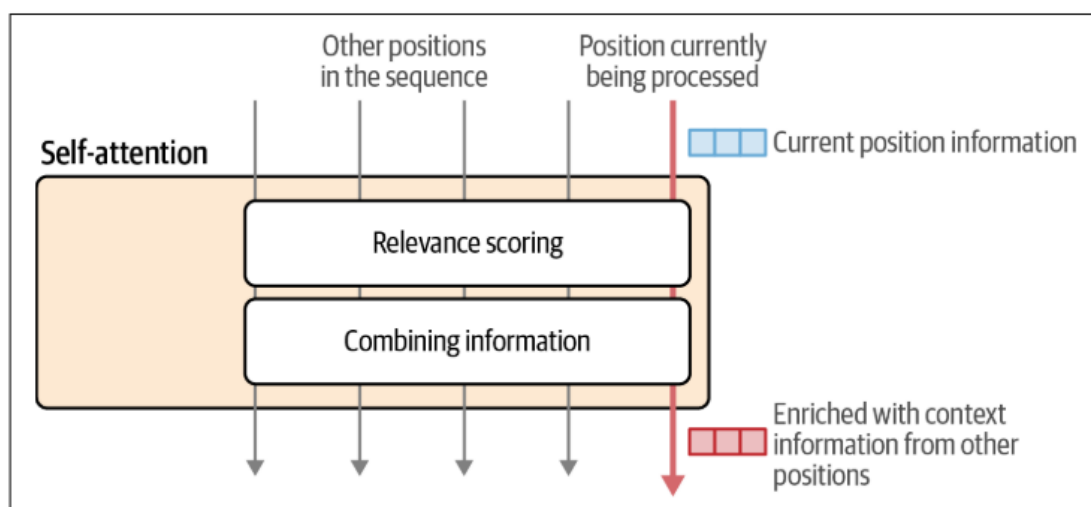


Figure 3-16. Attention is made up of two major steps: relevance scoring for each position, then a step where we combine the information based on those scores.

为了使 Transformer 的注意力能力更强大，模型中设置了多个注意力头，这些注意力头可以并行执行，每个头有自己的一套权重参数，不同的头专注于捕获不同类型的模式，例如头 1 捕获文本语法关系，头 2 捕获语义关系，头 3 捕获指代关系。

让我们来看看注意力是如何在单一注意力头中计算出来的。首先我们需要弄清楚以下几点：①注意力层处理的是单一位置的注意力；②层的输入有：当前位置或 token 的表示向量，之前的 token 的表示向量。③目标是产生当前位置的新的表示向量，包含了来自先前 token 的相关信息。④训练过程产生了三个投影矩阵：query（查询）投影矩阵 W_q ，key（键）投影矩阵 W_k ，value（值）投影矩阵 W_v 。query，key 和 value 是注意力机制中三个重要的参数，图 3-18 显示了在注意力计算开始前所有这些组件的初始位置。为简单起见，我们只关注一个注意力头，因为其他头的计算过程相同，只是使用各自独立的投影矩阵。

注意力计算的步骤：始于将输入向量分别乘以三个投影矩阵，得到查询向量 Q 、键向量 K 和值向量 V 。输入的每个 token 都有各自的查询向量 Q 、键向量 K 和值向量 V 。我们可以将这三个参数理解成：查询（Query）表示“我想要什么”，键（Key）：表示“我有什么可提供”，值（Value）表示“我实际提供的内容”（这里的“我”指 token）。由于处理过程是逐个 token 输入的，在处理单个 token 时，对应的嵌入向量生成的查询向量会与自己以及其他 token 的键向量的转置作点积，得到一个分数，再经过 `softmax()` 函数归一化得到权重，最后将得到的注意力权

重乘以对应 token 的值向量 \mathbf{WV} ，加权求和得到嵌入向量的更新向量 ΔE 。将更新向量与原始的嵌入向量求和就能得到更新后的嵌入向量。整个流程有两个主要步骤：token 的相关性评分和组合上下文信息。

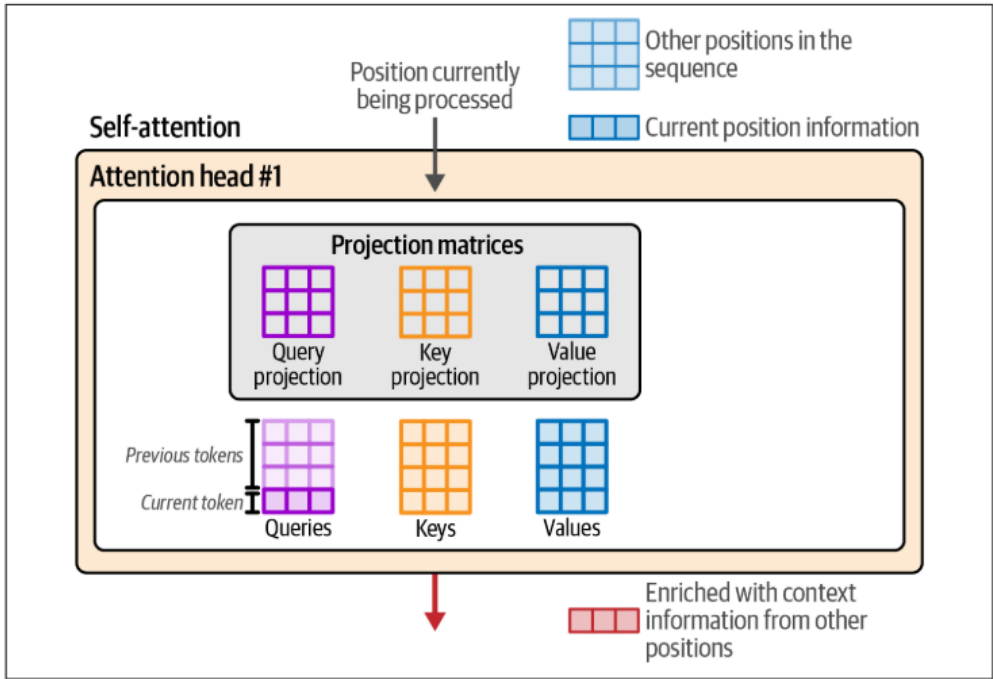


Figure 3-19. Attention is carried out by the interaction of the queries, keys, and values matrices. Those are produced by multiplying the layer's inputs with the projection matrices.

需要注意的是，同一个注意力头中的所有 token 使用相同的投影矩阵，但由于 token 不同，嵌入向量不同，导致查询矩阵、键矩阵和值矩阵各不相同。如图 3-19 所示，将所有 token 的嵌入向量组成的矩阵与查询投影矩阵相乘，就能得到每一行都对应一个 token 的查询矩阵。

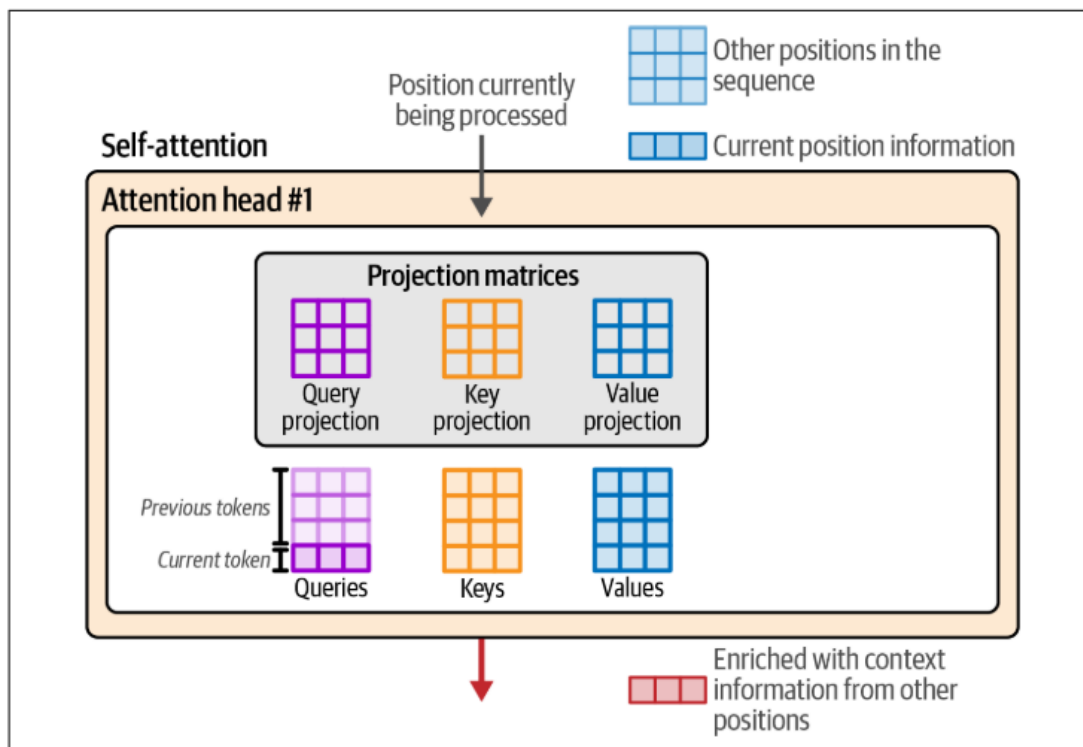


Figure 3-19. Attention is carried out by the interaction of the queries, keys, and values matrices. Those are produced by multiplying the layer's inputs with the projection matrices.

Token 的相关性评分：生成式 Transformer 时逐个生成 token 的，这意味着我们每次只处理一个位置。因此这里的注意力机制只关注这一个位置，以及如何从其他位置提取信息来丰富这个位置。注意力的相关性评分步骤是通过将当前位置的查询向量与键矩阵相乘来进行的。这会产生一个分数，表明每个先前 token 的相关性程度。通过 softmax 操作传递这些分数可以将它们归一化，使其总和为 1。图 3-20 显示了这一计算得到的相关性分数。

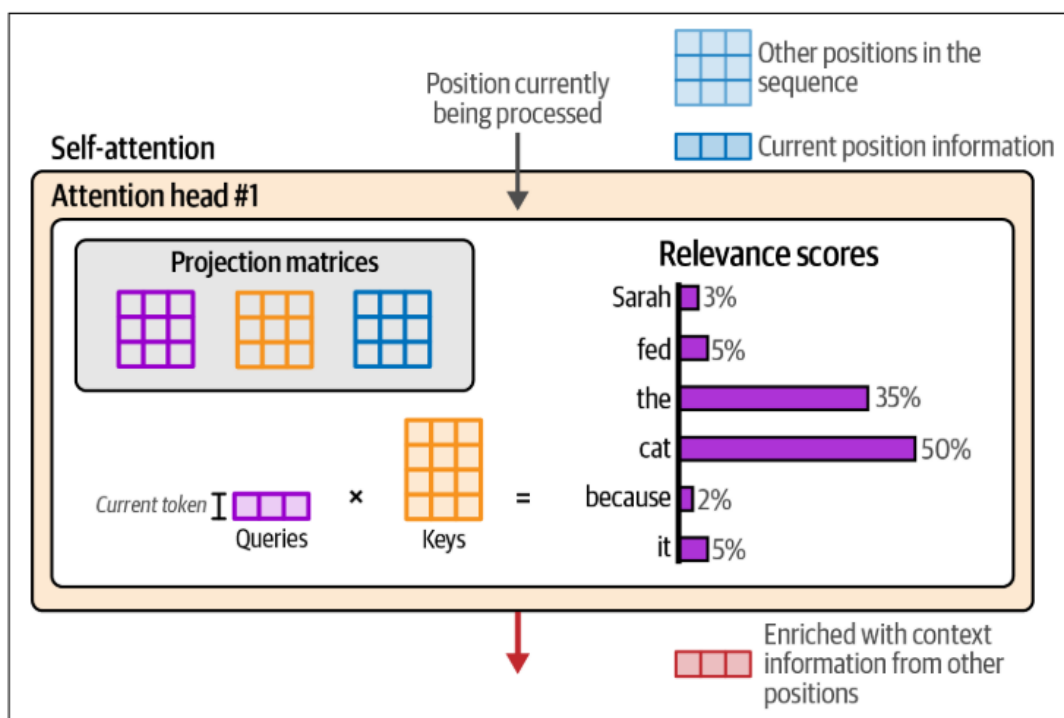


Figure 3-20. Scoring the relevance of previous tokens is accomplished by multiplying the query associated with the current position with the keys matrix.

组合上下文信息：现在我们有相关性分数，我们将每个 token 对应的值向量乘以该 token 的分数。将这些结果向量加权求和，就产生了这个注意力步骤的输出，即嵌入向量的更新向量 ΔE 。正如我们在图 3-21 中看到的那样。

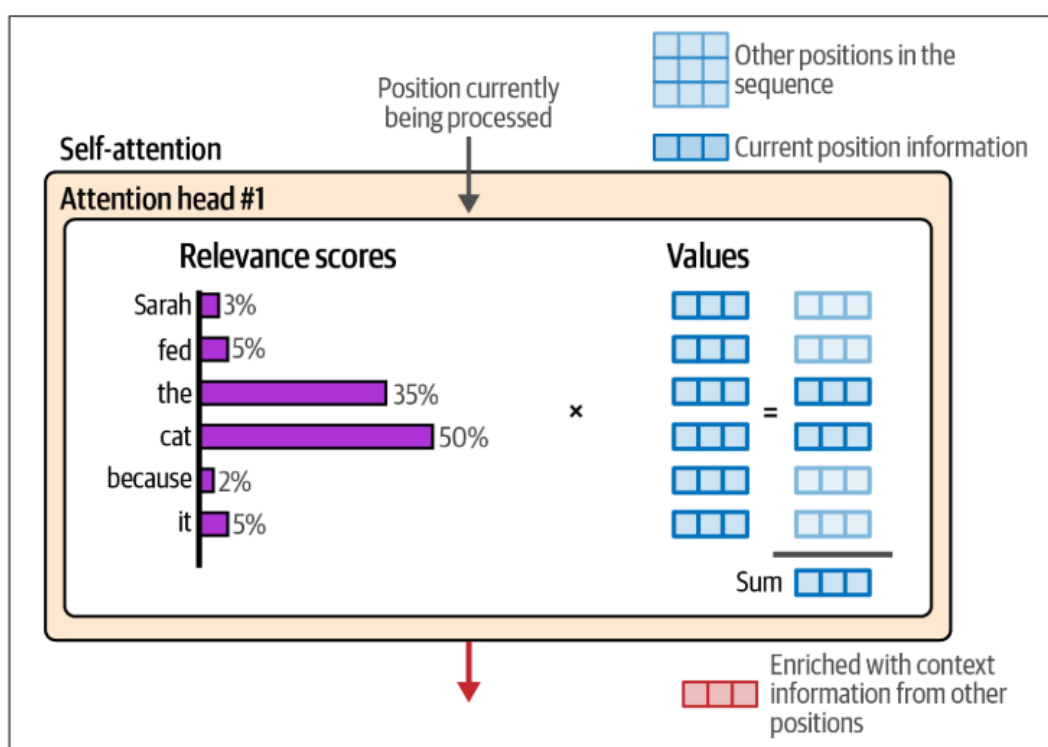


Figure 3-21. Attention combines the relevant information of previous positions by multiplying their relevance scores by their respective value vectors.

二. Transformer 架构的最新改进

自 Transformer 架构发布以来，研究者们致力于从多个维度提升其性能并构建更先进的模型。这些努力不仅体现在采用更大规模数据集进行训练、优化训练过程和学习率策略等方面，也延伸至架构本身的革新。这一节我们将详细探讨 Transformer 架构近期若干重要进展。

1. 更高效的注意力

最受学术界关注的是 Transformer 的注意力层，因为它是整个计算过程中计算成本最高的部分。主要有以下改进方法：

① Local/sparse attention 局部/稀疏注意力

随着 Transformer 模型越来越大，sparse attention 和 local attention 等方法改进了注意力计算的效率。需要注意的是，Sparse Attention 是一个更广义的概念。它泛指任何打破“全连接”模式，让每个 token 只关注某个稀疏子集（可自定义）的方法。Local attention 其实是 Sparse attention 的一种特例。

Local attention 限制每个 token 只能关注一个以它为中心的固定大小的局部窗口内的 token。对于位置 i 的 token，它只能关注位置在 $[i - w, i + w]$ 区间内的 token，而不是所有 N 个 token。典型代表是 Sliding Window Attention 滑动窗口注意力（论文：《Longformer: The long-document transformer》<https://arxiv.org/pdf/2004.05150>）。局部注意力的优点是效率高，计算和内存需求线性增长，可以处理极长序列。对于语言建模、局部图像处理等任务，邻近信息的确是重要性很高。缺点是无法建模长期依赖，对于窗口之外的 token，模型就无法直接捕获它。

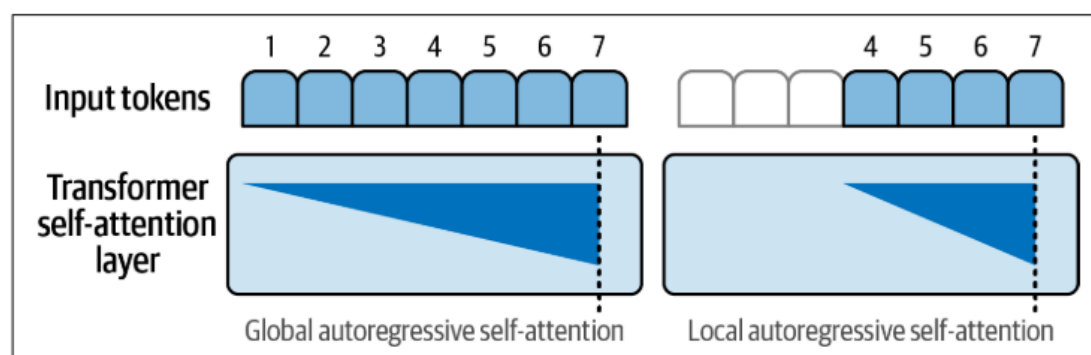


Figure 3-22. Local attention boosts performance by only paying attention to a small number of previous positions.

使用全局注意力机制的有 GPT-3，但是它没有对所有的 Transformer blocks 应

用这一机制，否则模型的生成质量会大大降低。GPT-3 的结构交织了 full-attention 和 efficient-attention Transformer 块。

Sparse attention 稀疏注意力(论文:《Generating long sequences with sparse transformers》<https://arxiv.org/pdf/1904.10509>) 提供了更多样化的、非局部的模型来选择要关注的 token。常见的模式包括：全局注意力 (Global Attention)、随机注意力 (Random Attention)、带状注意力 (Band Attention)。全局注意力是指定少数特定的 token (例如，序列开头的 [CLS]，或者句号分隔的句子首词) 拥有全局视野，可以关注所有 token，同时也被所有 token 关注。这相当于在稀疏图中添加了一些“枢纽”节点，让信息可以通过这些枢纽进行远程交换；随机注意力让每个 token 随机关注一定数量的其他 token；带状注意力主要针对图像等二维数据，关注一个像素的同行和同列的像素。稀疏注意力的优点是灵活，允许设计各种模式来适配特定任务或数据结构。通过结合“局部”和“全局”等模式，可以在高效的同时保留捕获远程信息的能力。缺点在于模式设计复杂，需要先验知识来设计有效的稀疏模式。并且高效的稀疏矩阵运算在硬件上不如稠密矩阵优化得好，可能难以充分发挥 GPU 等硬件的性能。

图 3-23 显示了不同的注意力机制是如何工作的，深蓝色是当前正在处理的 token，浅蓝色的是允许关注的 token。

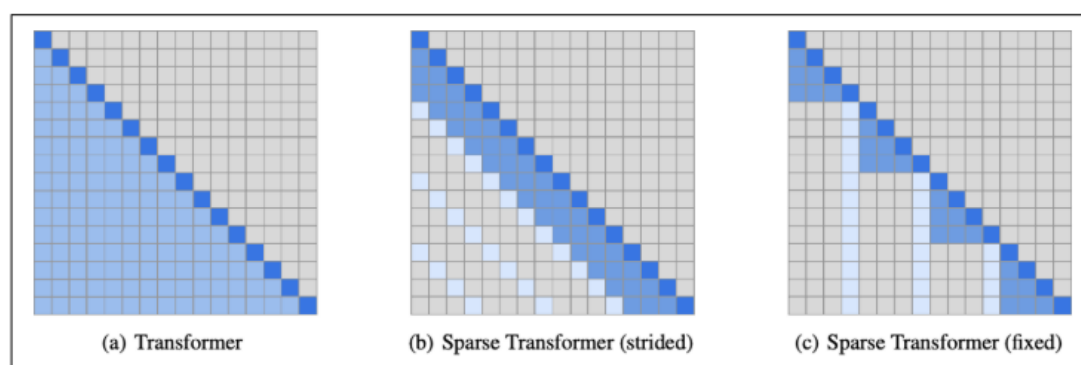


Figure 3-23. Full attention versus sparse attention. *Figure 3-24 explains the coloring.* (Source: “Generating long sequences with sparse transformers”.)

该图也展示了解码器 Transformer 模块（构成大多数文本生成模型的主体）的自回归特性；它们只能关注之前的词元。这与 BERT 形成鲜明对比——BERT 能够同时关注两侧的上下文（因此 BERT 中的 B 代表“双向的”）。

② Multi-query and grouped-query attention

近两年对 Transformer 模型的一个更有效的注意力优化是 Multi-query

attention (MQA) 多查询注意力 (论文:《Fast Transformer Decoding: One Write-Head is All You Need》[1911.02150] Fast Transformer Decoding: One Write-Head is All You Need) 和 grouped-query attention (GQA) 分组查询注意力 (论文:《GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints》[2305.13245] GQA: 从多头检查点训练广义多查询转换器模型)。

要理解多查询注意力 (MQA) 和分组查询注意力 (GQA), 首先要回顾一下原始 Transformer 中的多头注意力 (Multi-Head Attention, MHA)。在 MHA 中, 假设有 h 个头, 对于每个头, 都会创建独立的三个矩阵: 查询 (Query, Q)、键 (Key, K) 和值 (Value, V)。这意味着, 对于相同的输入序列, 需要存储 h 套不同的 K 和 V 矩阵。优点是每个头可以专注于捕获输入中不同方面的关系, 但缺点是在生成输出时, 需要缓存所有注意力头的 K 和 V 矩阵以供后续计算。当模型很大 (h 很多)、序列很长、批量很大时, 这会产生巨大的内存开销和内存带宽压力, 成为推理 (特别是自回归生成) 的主要瓶颈。那么解决方案就是**减少矩阵的数量**。

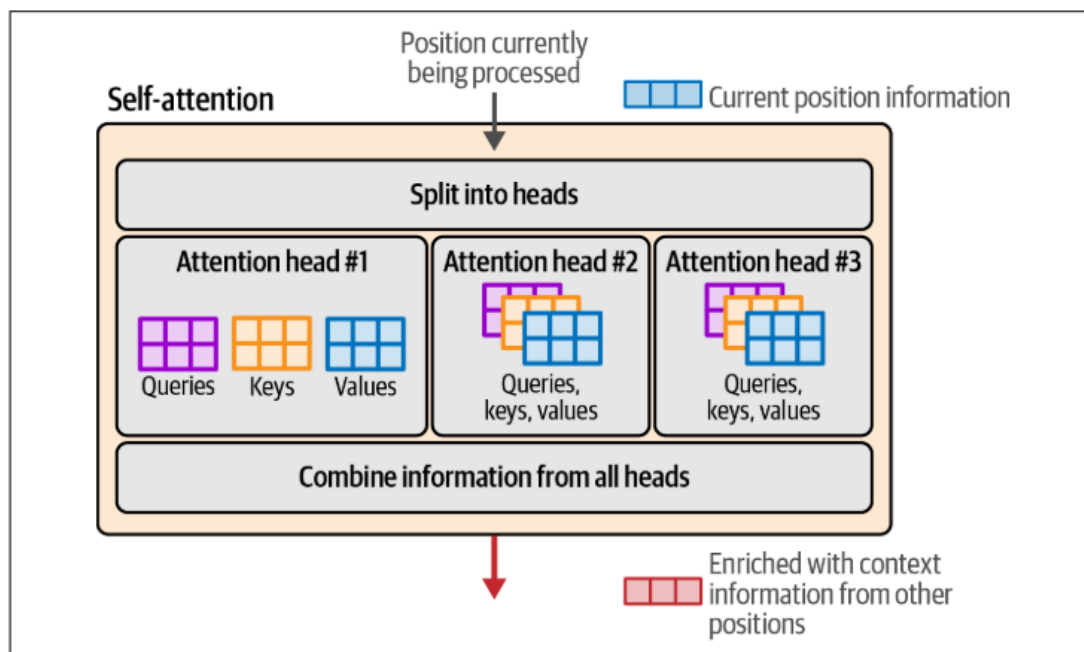


Figure 3-26. Attention is conducted using matrices of queries, keys, and values. In multi-head attention, each head has a distinct version of each of these matrices.

多查询注意力 (MQA) 是解决上述问题的一个极端但非常有效的方案, 核心思想是让所有注意力头共享同一套 K 矩阵和 V 矩阵。假设仍然有 h 个注意力头, 有 h 个 Q 查询矩阵, 从而产生 h 个 Q 查询向量, 但是只有 1 个 K 矩阵和 V 矩阵, 然后

h 个不同的 Q 查询向量和同一套 K 和 V 向量进行注意力计算。这极大减少了内存占用，并且由于 K 和 V 数据量大幅减少，降低了内存带宽压力，计算更快。但缺点是由于所有头都共享相同的 K 和 V ，可能导致模型的**表达能力下降**，理论上可能影响性能。但在许多实践中，这种性能损失很小，尤其是在大模型上。

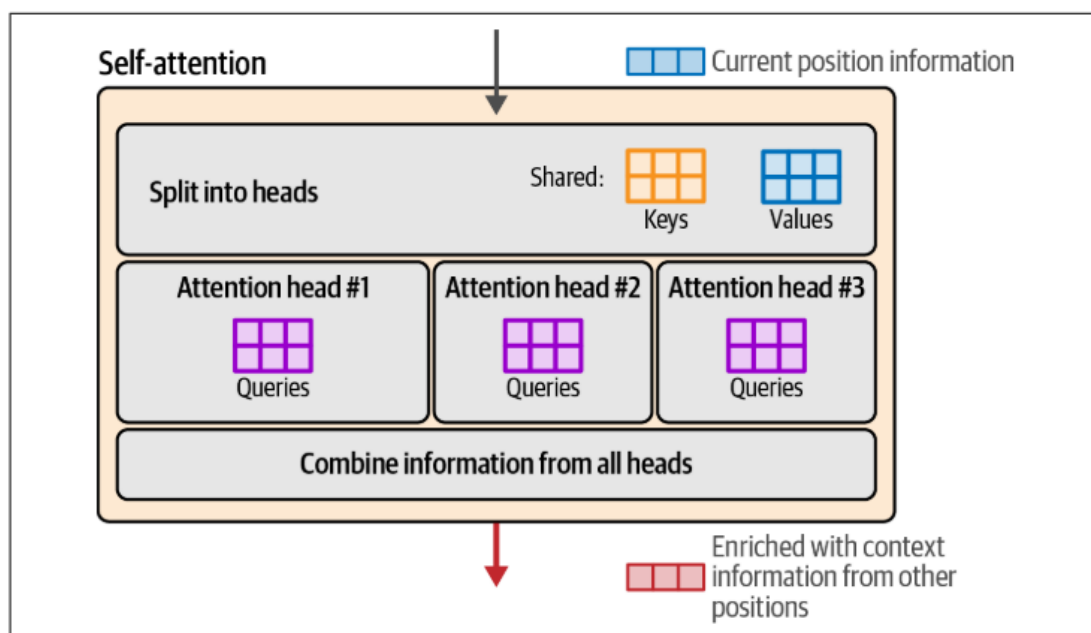


Figure 3-27. Multi-query attention presents a more efficient attention mechanism by sharing the keys and values matrices across all the attention heads.

随着模型大小的增长，这种优化可能过于苛刻，我们可以使用更多的内存来提高模型的质量。GQA 是 MHA 和 MQA 之间的一个折中方案，为了在 MHA 的表达能力和 MQA 的推理效率之间取得一个更好的平衡。

GQA (Group-query attention) 的核心思想是将所有的头分成 g 个组 (group)。在一个组内，所有头共享同一套键 (K) 矩阵和值 (V) 矩阵。假设有 h 个查询矩阵，并创建 g 套 ($1 < g < h$) K 和 V 矩阵 (g 是一个可调的超参数)，如果 $g=1$ ，那就退化成了 MQA (共享一套 KV 矩阵)，如果 $g=h$ ，那就退化成了 MHA (所有头有独立的 KV 矩阵)。与 MHA 相比，显著减少了内存占用和带宽压力 (需要缓存的 K, V 矩阵从 h 套减少到 g 套)。与 MQA 相比，通过使用多组 ($g > 1$) K, V ，保留了模型的一部分表达能力，通常比纯 MQA 能达到更好的模型性能。

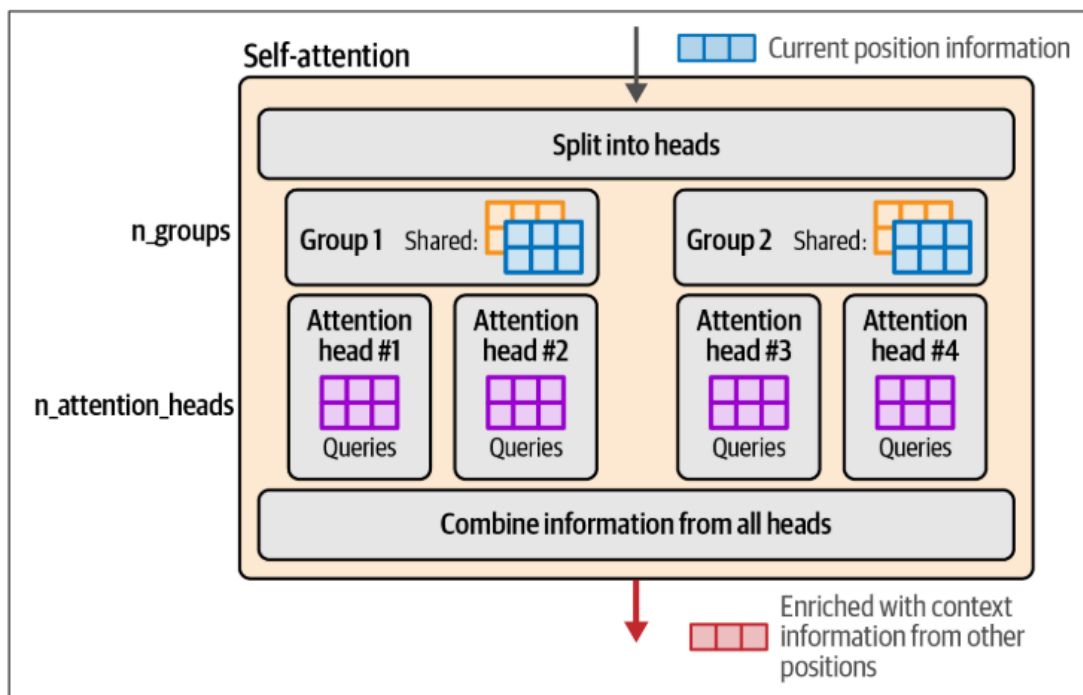


Figure 3-28. Grouped-query attention sacrifices a little bit of the efficiency of multi-query attention in return for a large improvement in quality by allowing multiple groups of shared key/value matrices; each group has its respective set of attention heads.

③ Flash Attention

Flash Attention 极大地优化了 Transformer 模型中自注意力(Self-Attention)的计算方式，主要目标是显著减少内存占用并提升计算速度。在标准的自注意力计算中，最耗内存的一步是计算并存储一个巨大的中间矩阵——注意力矩阵(Attention Matrix)。给定一个序列长度为 N ，首先会计算一个 $N \times N$ 的矩阵（即注意力矩阵，其中每个元素代表一个 token 对另一个 token 的注意力分数）。这个矩阵在内存中的大小是 $O(N^2)$ 。现代 GPU 的计算速度(FLOPs)非常快，但从高带宽内存(GPU 显存)中读写数据相对很慢。传统算法需要将整个巨大的 $N \times N$ 矩阵从 HBM 中读出来，在 SRAM 上（带宽是 HBM 的几十倍，速度极快，但容量为几十 MB）进行计算，然后再写回去。这个过程受到 HBM 带宽的限制，计算核心大部分时间在“等待”数据，效率低下，成为了整个计算的瓶颈。

Flash Attention 的算法设计正是精准地利用了 GPU 这种固有的硬件层次结构：Flash Attention 的计算方法是：先分块：将大的输入矩阵(Q, K, V)分成许多适合 SRAM 容量的小块。然后 HBM→SRAM：将一个小块从慢速的 HBM 加载到高速的 SRAM 中。然后在 SRAM 中计算：在 SRAM 这个“高速工作区”内完成这个小块所有的计算（矩阵乘法、Softmax、dropout 等）。最后 SRAM→HBM：只将最终的

输出结果（通常是降维后的结果）写回 HBM，丢弃那些巨大的中间结果（如注意力矩阵）。

相关论文：《FlashAttention: Fast and memory-efficient exact attention with IO-awareness》[\[2205.14135\]](#) [FlashAttention: 使用 IO 感知实现快速且节省内存的精确注意力](#)

《FlashAttention-2: Faster attention with better parallelism and work partitioning》<https://tridao.me/publications/flash2/flash2.pdf>

2. The Transformer block

Transformer block 的两个主要组件是注意力层和前馈神经网络。图 3-29 显示了更详细结构，除了注意力层和前馈神经网络，还有残差连接与层归一化（Add & Normalize）

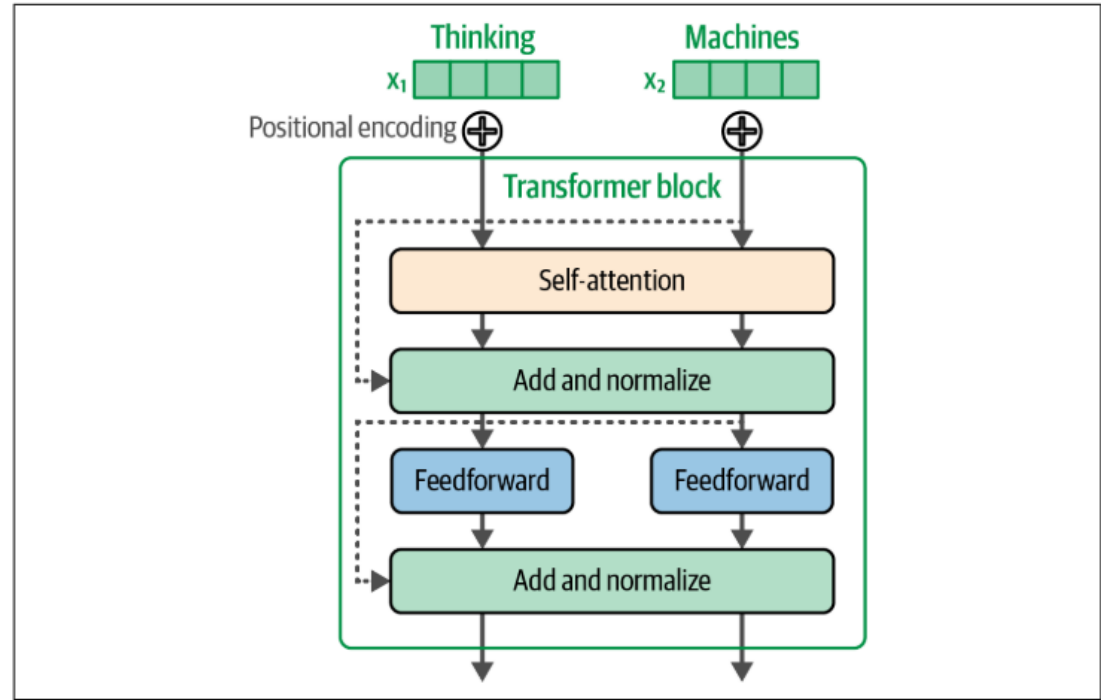


Figure 3-29. A Transformer block from the original Transformer paper.

到目前为止，最新的 Transformer 模型仍保留了其主要组件，但如图 3-30 所示，它们已进行了若干优化调整。

该版本 Transformer 块的一个显著区别是：归一化操作被提前至注意力层和前馈网络层之前执行。这种调整被证明能有效减少模型所需的训练时间（论文《On layer normalization in the Transformer architecture》[\[2002.04745\]](#) [关于 Transformer 架构中的层归一化](#)）。另一项归一化改进是采用 RMSNorm 方法——

相比原始 Transformer 使用的 LayerNorm，这种归一化方式更简洁高效（论文《Root mean square layer normalization》[\[1910.07467\]](#) 均方根层归一化）。最后，新一代模型普遍采用 SwiGLU 等激活函数（详见《GLU Variants Improve Transformer》[2002.05202](#)）取代了原始架构中的 ReLU 激活函数。

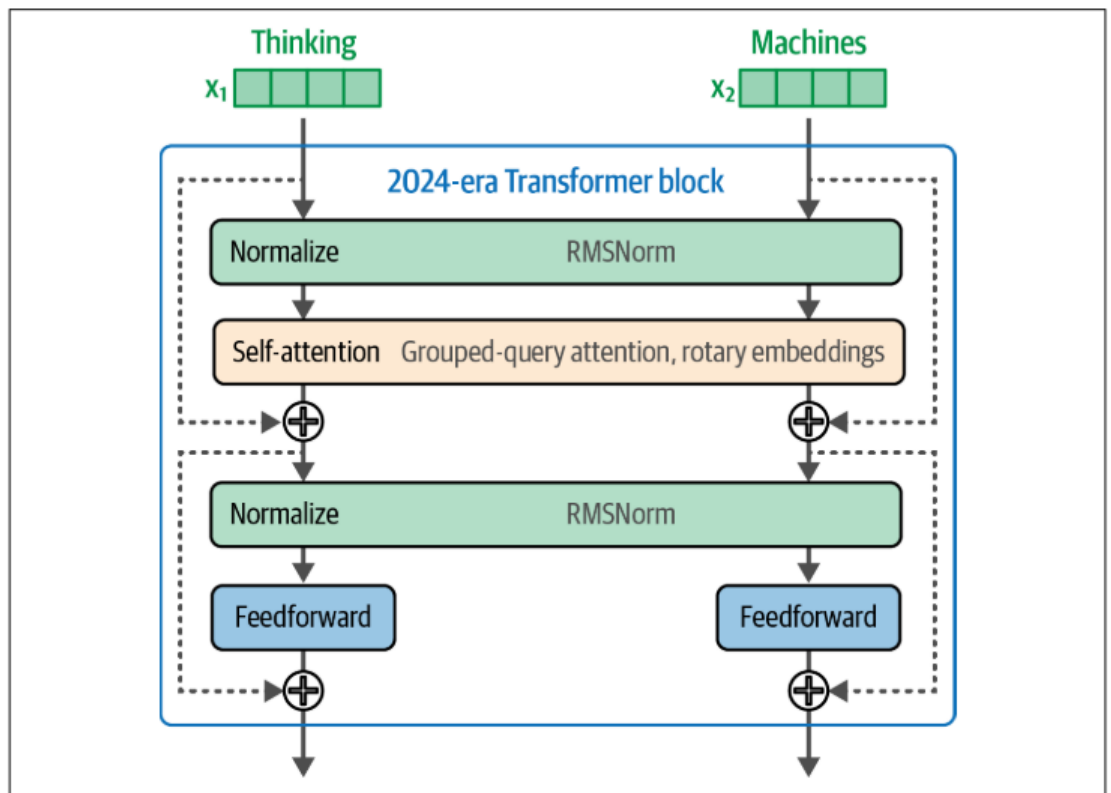


Figure 3-30. The Transformer block of a 2024-era Transformer like Llama 3 features some tweaks like pre-normalization and an attention optimized with grouped-query attention and rotary embeddings.

3. 位置编码 Positional Embeddings (RoPE)

自原始 Transformer 模型问世以来，位置嵌入一直是其核心组件。它使模型能够追踪序列/句子中词元的顺序信息——这是语言中不可或缺的关键信息。在过往提出的多种位置编码方案中，旋转位置嵌入（Rotary Positional Embeddings, RoPE；论文《RoFormer: Enhanced Transformer with rotary position embedding》[\[2104.09864v4\]](#) RoFormer: 带旋转位置嵌入的增强型变压器）尤为值得关注。

原始 Transformer 论文及早期变体模型采用的都是绝对位置嵌入，其本质是将首个词元标记为位置 1，第二个标记为位置 2，依此类推。这类编码既可采用静态方法（通过几何函数生成位置向量），也可采用学习式（在模型训练过程中通过

学习获得位置向量值)。但随着模型规模扩大，此类方法逐渐暴露出弊端。例如，当训练集中大量文档的长度远小于上下文窗口时（将完整的 4K 上下文窗口分配给仅包含 10 个单词的短句），会造成显著的计算资源浪费。因此在模型训练过程中，如图 3-31 所示，会将多个文档拼接组合后填入训练批次的上下文窗口中。

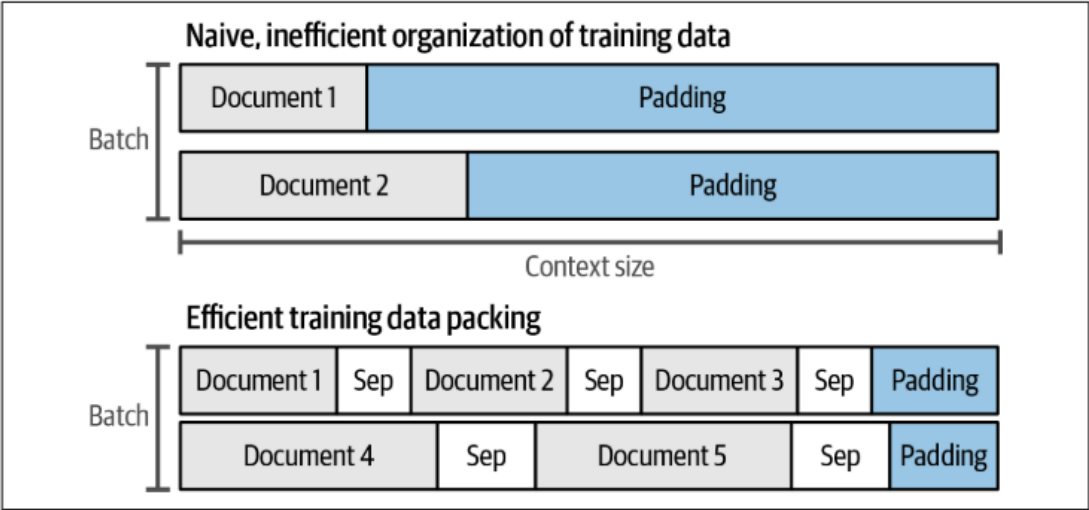


Figure 3-31. Packing is the process of efficiently organizing short training documents into the context. It includes grouping multiple documents in a single context while minimizing the padding at the end of the context.

但是当把多个短文档打包到一个长序列中时，绝对位置编号会出错。例如，第二个文档的第一个词元位置可能是“50”，模型会错误地认为前面有 49 个属于同一文档的词元。若想深入了解序列打包技术，可以阅读《Efficient Sequence Packing without Cross-contamination: Accelerating Large Language Models without Impacting Performance》[\[2107.02027\] Efficient Sequence Packing without Cross-contamination: Accelerating Large Language Models without Impacting Performance](#)，并观看 [Introducing packed BERT for 2X training speed-up in natural language processing](#) [推出打包 BERT，将自然语言处理的训练速度提高 2 倍](#) 的视觉演示。

RoPE 是一种巧妙的位置编码技术，它通过在向量空间中旋转特征向量的方式，来为 Transformer 模型提供绝对和相对位置信息。RoPE 的核心思想是不要将位置信息作为一个独立的数字“加”上去，而是将它表示为一种“旋转”操作，直接作用于计算注意力之前的查询（Query）和键（Key）向量上。如图 3-32 所示，它作用在自注意力层上。

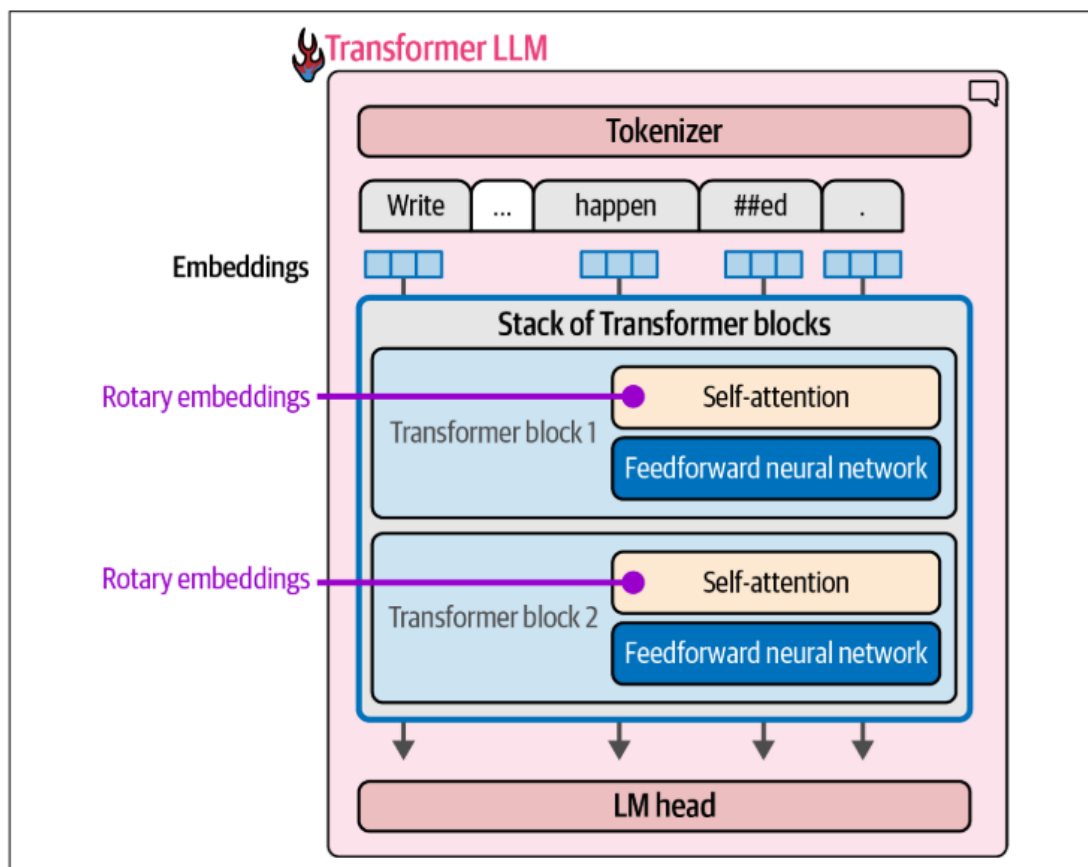


Figure 3-32. Rotary embeddings are applied in the attention step, not at the start of the forward pass.

工作原理：①将位置表示为旋转：对于序列中的第 m 个位置，RoPE 定义了一个独特的旋转矩阵 R_m 。这个矩阵的设计使得它能够对向量进行一个特定角度的旋转。②作用于 Query 和 Key：在计算注意力分数 $(Q \cdot K^T)$ 之前，RoPE 使用这个旋转矩阵来“加持” Query 和 Key 向量，如图 3-33 所示。

旋转后的 Query 向量： $q'_m = q_m \cdot R_m$

旋转后的 Key 向量： $k'_n = k_n \cdot R_n$

此时，注意力分数变为： $(q_m \cdot R_m) (k_n \cdot R_n)^T = q_m \cdot R_m \cdot R_n^T \cdot k_n^T$

由于旋转矩阵 R 是正交矩阵 ($R^T = R^{-1}$)， $R_m \cdot R_n^T$ 实际上的效果是从 θ_m 旋转到角度 θ_n ，这等价于一个只依赖于位置差 $(n-m)$ 的旋转矩阵 $R_{\{n-m\}}$ 。

旋转位置嵌入的优点在于：它天然地包含了相对位置信息，能够处理长度更长的序列，解决了打包文档的混淆问题。

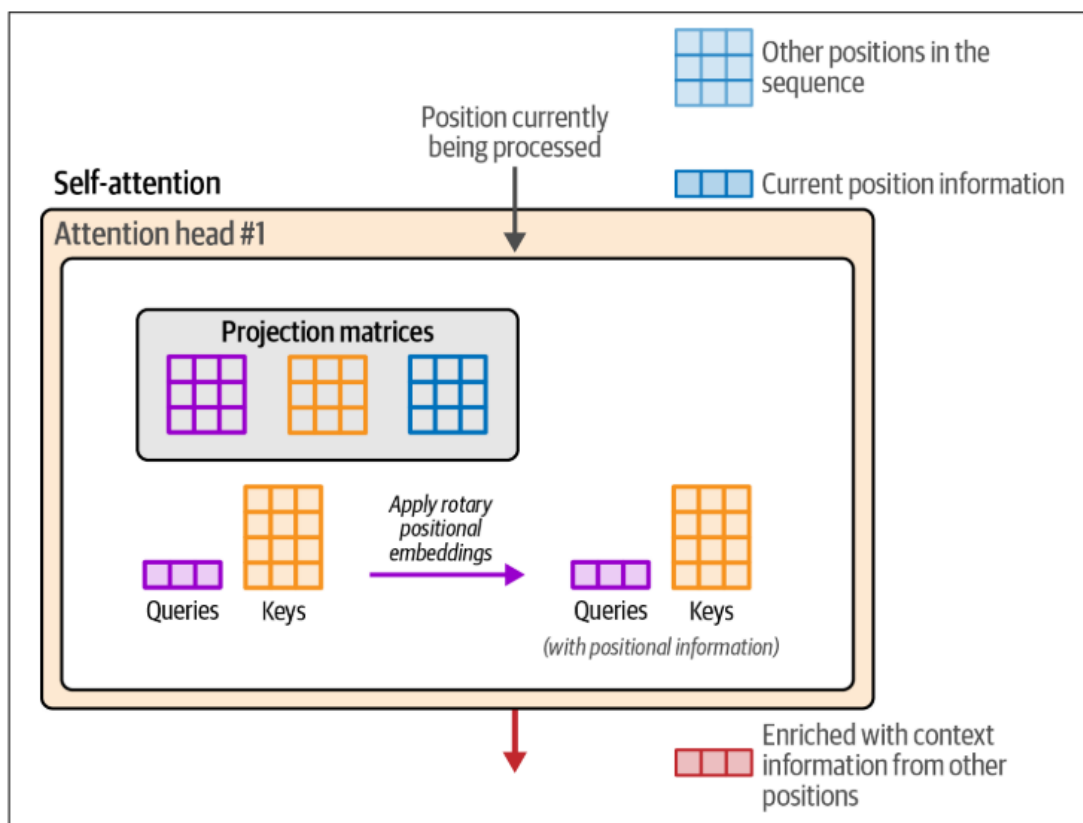


Figure 3-33. Rotary positional embeddings are added to the representation of tokens just before the relevance scoring step in self-attention.

4. 其他架构的探索与改进

针对 Transformer 的各种优化调整持续被提出和研究。《A Survey of Transformers》<https://arxiv.org/abs/2106.04554> 重点梳理了若干主要发展方向。除了 LLM 领域，Transformer 架构不断适配其他领域，如计算机视觉 Computer vision、机器人技术 Robotics 和时间序列分析 Time series。

计算机视觉：A Survey on Vision Transformer [A Survey on Vision Transformer | IEEE Journals & Magazine | IEEE Xplore](#)

Transformers in Vision: A Survey [Transformers in Vision: A Survey | ACM Computing Surveys](#)

机器人技术：“Open X-Embodiment: Robotic learning datasets and RT-X models [开放式 X 实施例：机器人学习数据集和 RT-X 模型](#)

时间序列分析：Transformers in Time Series: A Survey [\[2202.07125\] Transformers in Time Series: A Survey](#)