大型语言模型（LLM）的文本生成

LLM 的推理方式

LLM 的推理方式通俗易懂版

大型语言模型(LLM)就像一个聪明的写作助手, 它能根据你给的开头或提示, 自动生成后面的内容。

但它并不是一口气就写出整段文字, 而是一个词一个词地"打"出来的, 就像我们用键盘打字一样。

LLM生成文字的过程分两步:

1. **预填充**: 你给模型一个开头, 可以是几个词或一个句子,这就是"预填充"的内容。模型会先读懂这个开头, 就像你给了它一个写作的题目。
2. **续写**: 模型开始根据你给的开头, 一个接一个地生成新的词, 每生成一个新词, 它都会把前面已经生成的内容重新读一遍, 保证新词能紧接上文, 不至于写跑题。这个过程就叫"续写"。

举个例子, 假设你给模型的开头是"从前有座山,山里有座", 模型的续写过程可能是这样的:

- 输入:"从前有座山, 山里有座" >>> 输出:"庙"
- 输入:"从前有座山, 山里有座庙" >>> 输出:"，"
- 输入:"从前有座山, 山里有座庙，" >>> 输出:"庙"
- 输入:"从前有座山, 山里有座庙，庙" >>> 输出:"里"
- 输入:"从前有座山, 山里有座庙，庙里" >>> 输出:"有"
- ……

你看, 模型就像一个小学生在练习文字接龙游戏, 每次只接一个词, 接完后再把新的句子重复一遍, 再接下一个词。

这就是为什么当你使用ChatGPT等LLM工具时, 它的回复常常是一个字一个字蹦出来的, 而不是一下子就显示一大段。

这实际上就是模型"续写"的过程, 而不完全是为了模仿人打字的效果。

所以下次你看到LLM一个字一个字地"打"出文字时, 就知道它正在使出浑身解数, 根据你的提示, 认真地生成回复呢!

这就是LLM的"思考"方式。

大型语言模型（LLM）基于 Transformer 架构，通过两个主要步骤生成文本：

1. **预填充（Prefill）**：将输入提示中的所有Token转换为模型可以处理的格式。
2. **生成（Completion）**：模型采用自回归方式，一次生成一个新Token。

每次生成新Token时，模型都会将之前的所有Token作为输入。

注：LLM（大型语言模型）中的 Token 是模型在处理文本时使用的基本单位。它可以是单词、子词甚至字符，具体取决于模型的分词策略。

以下是一个生成示例：

- 输入：你 >>> 输出：好
- 输入：你好 >>> 输出：，
- 输入：你好， >>> 输出：我
- 输入：你好，我 >>> 输出：是
- 输入：你好，我是 >>> 输出：A
- 输入：你好，我是A >>> 输出：I

- 输入：你好，我是AI >>> .....

每个生成步骤都需要考虑之前的上下文，因此模型能保持一致的语境，并在流式输出界面上呈现类似打字的效果。

虽然这种效果看起来更拟人化，但本质上是由于模型的自回归生成机制所致。几乎所有 LLM 都采用这种方式生成文本。

在对话过程中，LLM 的生成方式也是如此。

模型实际上并没有对话和上下文记忆功能，而是将之前的所有对话作为输入，以生成新的Token。

例如，以下可能是一次推理的输入：

```
Human：你好，你是谁？
Assistant：你好，我是 AI 助手。
Human：你能做什么？
Assistant：
```

LLM 获取到这个输入后，会在最后一个冒号「：」后生成第一个Token，然后将其添加到输入中，再生成下一个Token，直到生成终止符Token。

当我们调用 API 而不使用流式输出时，通常会将第一个输出的Token到最后一个终止符Token的全部内容一次性返回给调用者。

这种行为与大多数人的直觉不符。

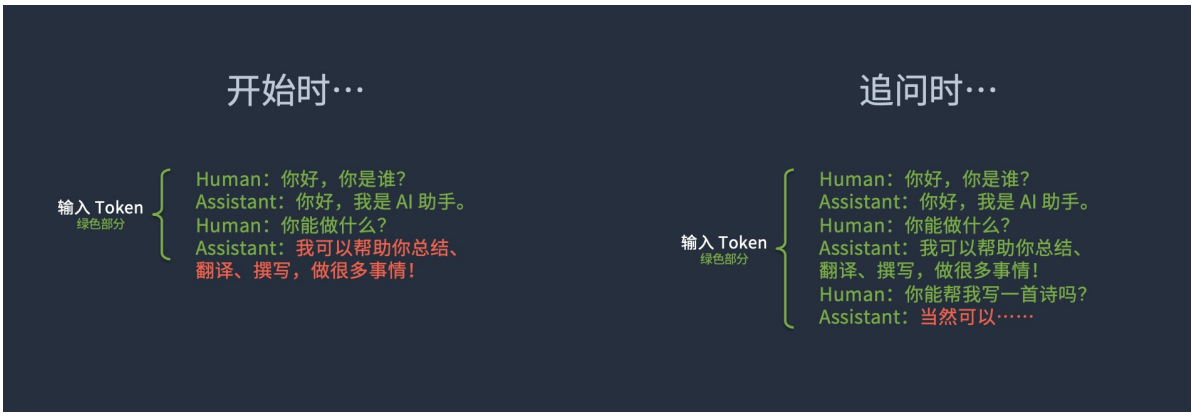
例如，在一个聊天机器人与 LLM 的对话中：

- 第一个问题输入了 8 个Token
- 第二个问题输入了 10 个Token

那么第二个问题的「输入Token数」是多少？

一般用户可能会认为是 10 个，但实际上消耗的Token数是：8 + 第一次生成的输出Token数 + 10。

再例如，如果我们将一篇包含 1000 个Token的文档发送给 LLM 并要求它总结，然后继续提问 10 个与文档相关的问题，那么每次提问的输入Token数都需要包括文档的 1000 个Token，加上之前所有问题和答案的Token数。



LLM在对话时，并没有真正意义上的"记忆"。它只是把之前的对话内容作为输入，然后基于这个输入生成下一个回复。

这个过程跟它生成普通文本非常类似。那些我们看到的完整回复，其实是模型一个词一个词生成的结果。

理解这个行为模式很重要，因为它会影响推理的成本。

虽然有的大模型支持 10+ 万 Token 的上下文窗口，但如果真的用满这个窗口，带着全部上下文来问问题，那么每问一个问题，输入的 Token 数也会达到 10 万。

这就是为什么, 尽管一些大模型支持很长的上下文(比如10万个Token), 我们在实际使用时还是要注意上下文的长度。

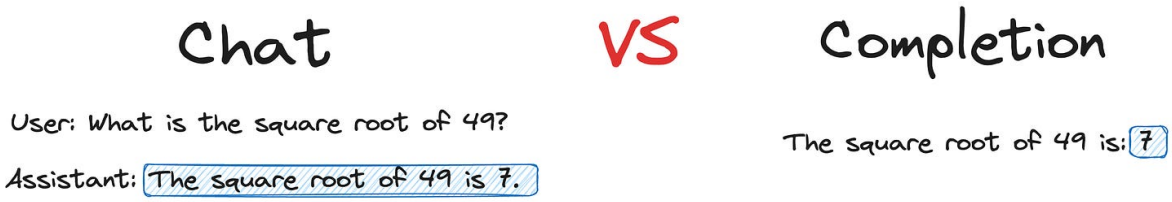
不然的话, 随着对话的进行, 每一轮消耗的Token数会快速增长, 推理的成本会变得很高, 效率也会受影响。

LLM的文本生成模式

LLM采用自回归生成方式，通过逐步预测每个Token来生成连贯的文本。

自回归生成方式意味着模型会一次生成一个 Token，并将其作为输入用于生成下一个 Token，直到生成结束。

下面详细介绍LLM的文本生成模式，包括Completion模式和Chat模式。



文本生成模式概述

1. Completion模式

Completion模式是LLM模型最基础的文本生成模式，适用于从给定提示（prompt）开始自动补全文本段落或回答问题。

工作流程

1. **输入提示：** 用户提供一个完整的输入提示。例如：

```
prompt = "The history of artificial intelligence is"
```

2. **生成文本：** 基于输入提示，LLM模型使用逐个Token生成的方式生成后续文本。

2. Chat模式

Chat模式旨在模拟多轮对话，使得LLM能够与用户进行连贯的互动。

工作流程

1. **多轮对话输入：** 用户提供多轮对话的上下文。例如：

```
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Can you tell me about the history of AI?"}
]
```

2. **生成回复：** 基于对话上下文，LLM模型逐个Token生成助手的回复。

小结

LLM的Completion模式和Chat模式确实各有其独特的应用场景, 但在技术实现上也有一些共同点和区别。

Completion模式主要用于根据给定的提示(Prompt)生成连续的文本。

提示可以是一个或几个句子, 模型根据提示的上下文, 自回归地生成后续的文本内容。

这种模式适用于开放域的文本生成任务, 如故事创作、文章写作、代码补全等。

Chat模式则专门用于模拟多轮对话。在这种模式下, 模型根据之前的对话上下文, 生成当前轮次的回复。

Chat模式需要维护一个对话历史, 以便模型能够理解对话的上下文并生成连贯的回复。这种模式适用于对话系统、聊天机器人、虚拟助手等应用场景。

在生成策略方面, 两种模式都采用了逐个Token生成的策略, 即根据前面已生成的Token预测下一个Token。

但在生成过程中, 它们可能会使用不同的技术来控制生成过程, 以满足不同的应用需求:

1. Completion模式通常使用诸如Beam Search、Top-k采样、Top-p(nucleus)采样等策略, 以在生成的多样性和相关性之间取得平衡。这有助于生成更丰富、更连贯的文本内容。
2. Chat模式除了使用这些通用的生成策略外, 还可能引入一些专门的技术, 如角色指令(Role Instruction)、基于检索的生成(Retrieval-based Generation)等, 以更好地控制对话的风格、主题和逻辑一致性。

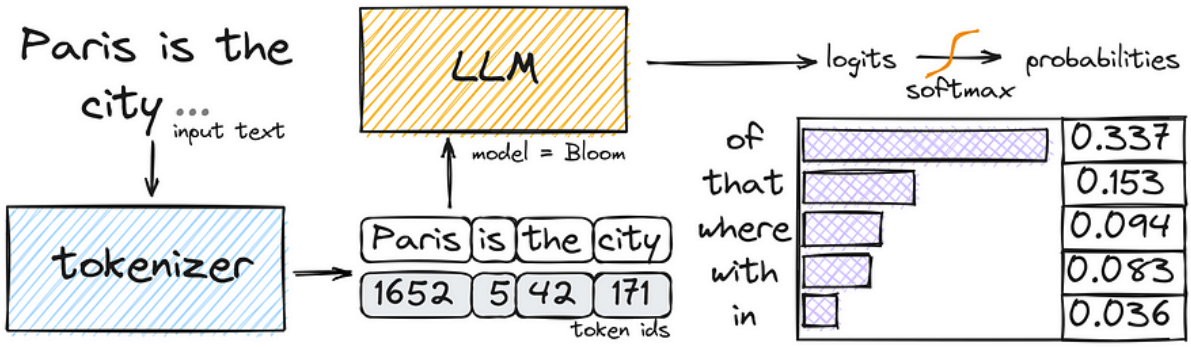
LLM的文本生成策略

大规模语言模型（LLM）的文本生成策略指的是从模型中生成文本的具体方法或技术。

这些策略用于控制模型生成的文本内容、质量和连贯性。

假设我们想继续生成短语“Paris is the city ...”的后续内容。

我们将使用LLM模型，它会为所有Token输出logits（可以简单理解为得分），通过softmax函数可以将这些得分转换成每个Token在生成时被选中的概率。



如果查看前五个输出Token，它们都很合理。我们可以生成以下听起来合适的短语：

- **Paris is the city** *of* love.
- **Paris is the city** *that* never sleeps.
- **Paris is the city** *where* art and culture flourish.
- **Paris is the city** *with* iconic landmarks.
- **Paris is the city** *in* which history has a unique charm.

现在的挑战是选择合适的Token。为此，有几种策略。

Greedy sampling (贪婪抽样)

在贪婪策略中，模型总是选择它认为在每一步最有可能的Token，不考虑其他可能性或探索不同选项。

模型选择概率最高的Token，并根据所选的Token继续生成文本。

使用贪婪策略计算上高效且直接，但有时会导致生成重复或过于确定的输出。

由于模型在每一步只考虑最有可能的Token，它可能无法充分捕捉上下文和语言的多样性，或生成最具创意的回应。

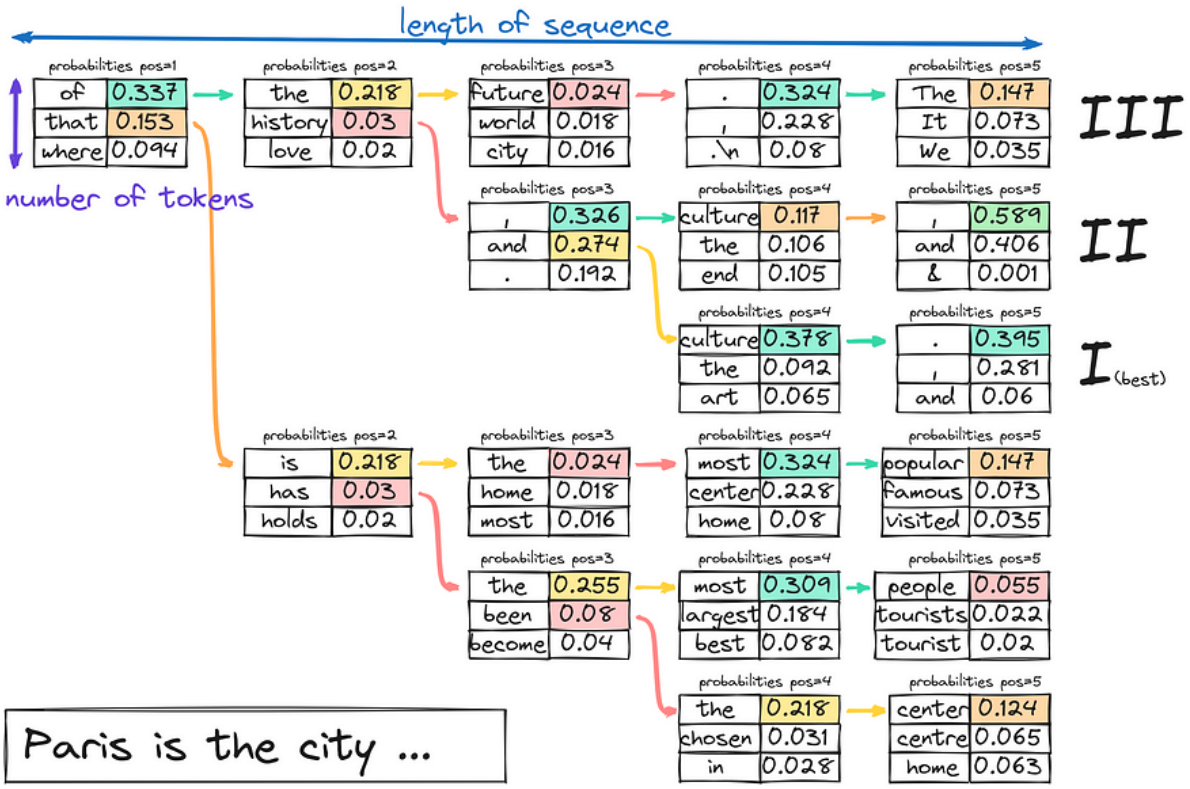
模型的短视性使其只关注每一步中最有可能的Token，而忽略了对整个序列的整体影响。

Generated output: *Paris is the city of the future. The*

Beam search (束搜索)

束搜索（Beam search）是文本生成中使用的另一种策略。

在束搜索中，模型假设了一组“K”个最有可能的Token，而不是在每一步只考虑最有可能的Token。这组K个Token被称为“束”（beam）。



模型为每个Token生成可能的序列，并在文本生成的每一步通过扩展每个束的可能线路来跟踪它们的概率。

这个过程持续进行，直到生成的文本达到所需长度，或者每个束都遇到一个“结束”Token。

模型从所有束中选择整体概率最高的序列作为最终输出。

从算法角度来看，创建束相当于扩展一个K叉树。

创建束后，会选择整体概率最高的分支。

Generated output: *Paris is the city of history and culture.*

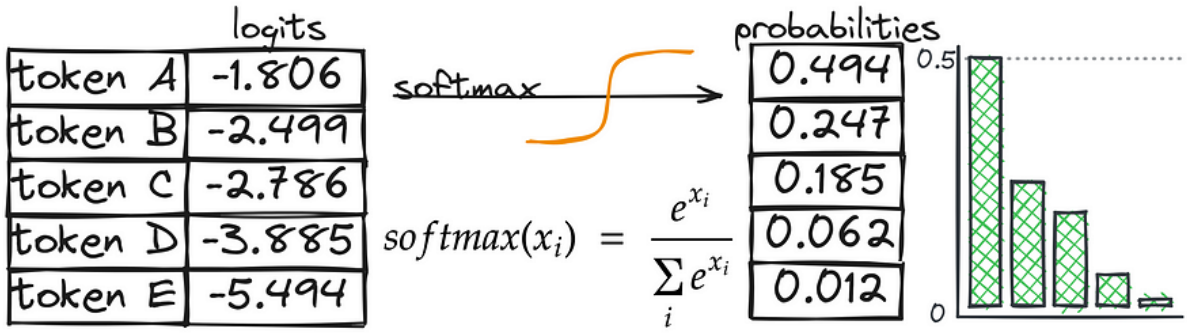
Normal random sampling (正常随机抽样)

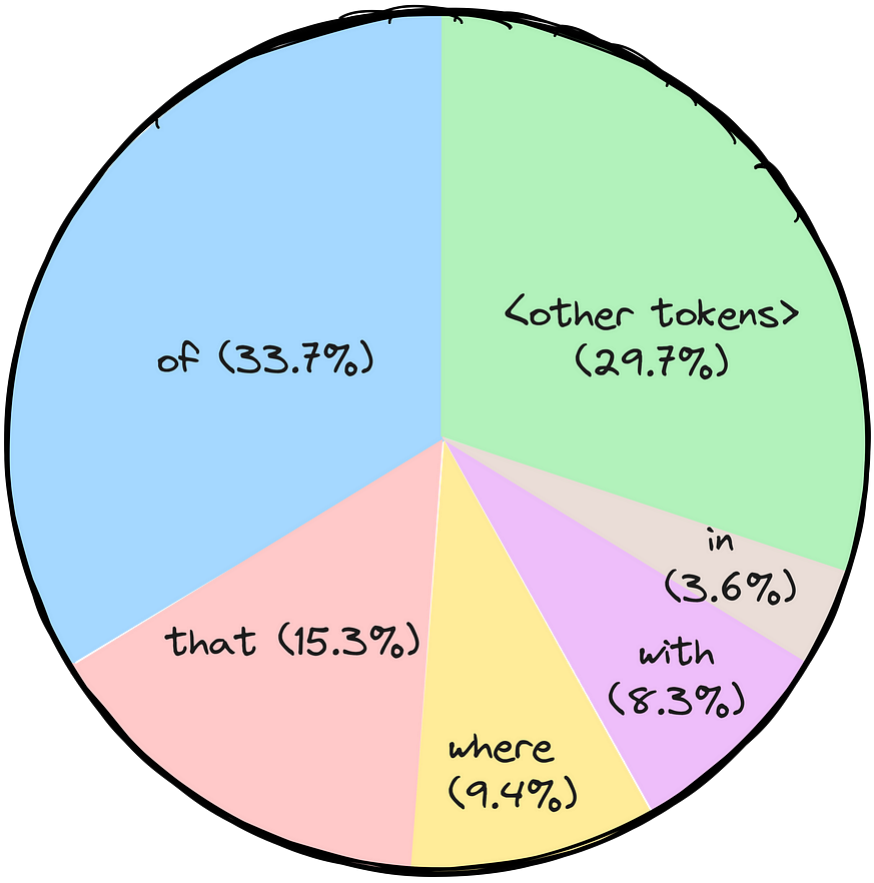
这个想法很简单——通过选择一个随机值并将其映射到选定的Token上来选择下一个词。

可以将其想象成转动一个轮盘，每个Token的区域由它的概率决定。

概率越高，Token被选中的机会就越大。

这是一种相对低成本的计算解决方案，由于相对随机性较高，句子（或Token序列）每次可能都会有所不同。

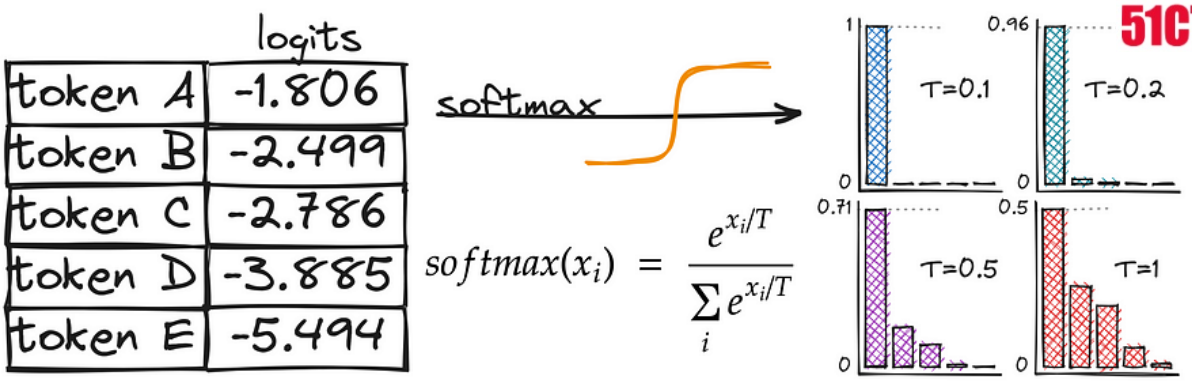




Random sampling with Temperature(随机温度抽样)

LLM使用softmax函数将logits转换为概率。

在这里，引入温度（temperature）这个超参数，它影响文本生成的随机性。



在标准的Softmax函数中，我们根据模型输出的logits计算每个Token的概率：

其中，x_i表示第i个Token的logit值，N为词表大小。

而在温度Softmax函数中，引入了一个温度参数T：

可以看到，温度T作为分母出现在指数项中。这个参数可以控制概率分布的形状：

1. 当$T = 1$时，温度Softmax函数退化为标准的Softmax函数。此时，概率分布与原始logits保持一致。
2. 当$T > 1$时，温度较高，概率分布变得更加平缓。这意味着原本概率较低的Token现在有更大的机会被选中，生成的文本将更加多样化和富有创造性。但同时，生成的文本也可能偏离原始主题或失去连贯性。
3. 当$0 < T < 1$时，温度较低，概率分布变得更加尖锐。原本概率较高的Token将占据主导地位，生成的文本将更加保守和确定性强。这有助于生成与提示高度相关的文本，但可能缺乏多样性。

因此，温度参数T提供了一种调节生成文本特性的方法。

我们可以根据具体的应用场景和需求，灵活选择合适的温度值。

较高的温度适用于创意写作、故事生成等任务，而较低的温度则适用于事实回答、摘要生成等任务。

除了温度Softmax，还有其他一些变体函数，如Top-k采样和Top-p（nucleus）采样等。

它们通过在采样前截断概率分布，以不同的方式平衡生成文本的多样性和相关性。

这些函数与温度参数可以结合使用，提供更精细的生成控制。

Top-K sampling （Top-K抽样）

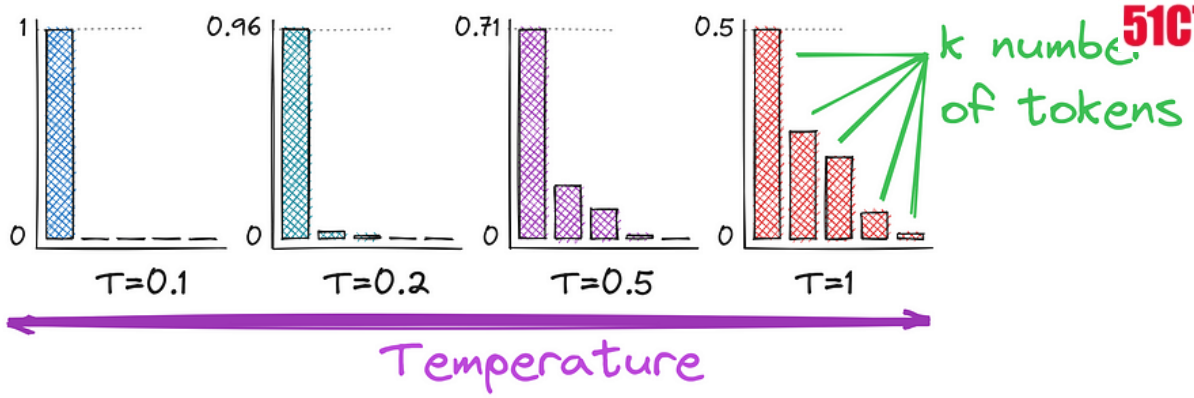
现在可以通过温度采样来调整概率。

另一个改进方法是只使用top-K Tokens，而不是全部Token。

这将提高文本生成的稳定性，并不会过多地降低创造力。

这种方法是对仅top-KToken进行随机采样并结合温度。

唯一可能的问题是选择K的数量。



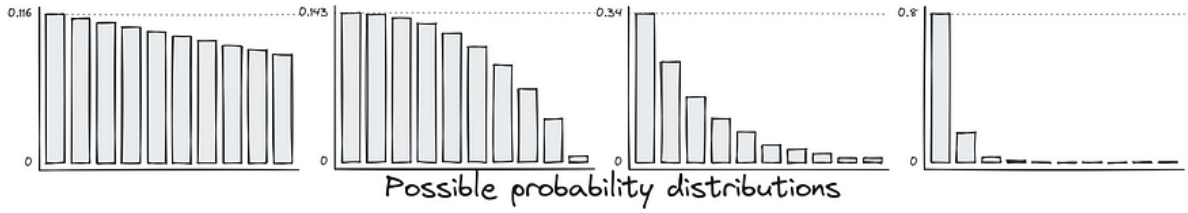
Nucleus sampling or top-p sampling （Top-p抽样）

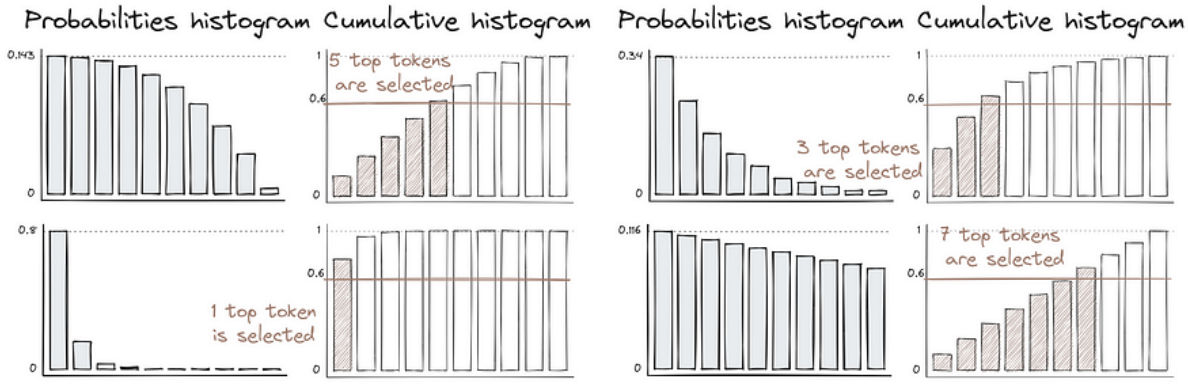
Token概率的分布可能会有很大差异，这在文本生成过程中可能会带来意想不到的结果。

核采样（Nucleus Sampling）旨在解决不同采样技术的某些局限性。

它不是指定要考虑的固定数量的“K”个Token，而是使用一个概率阈值“p”。这个阈值代表您希望在采样中包含的累积概率。

模型在每一步计算所有可能Token的概率，然后按降序排列。





模型继续向生成的文本中添加Token，直到它们的总概率超过指定的阈值。

核采样的优势在于它允许基于上下文进行更动态和自适应的Token选择。

在每一步选择的Token数量可以根据该上下文中Token的概率而变化，这可以导致更多样化和更高质量的输出。

LLM生成策略比较

主要生成策略的比较：

- **贪婪搜索：** 确定性强，但可能生成重复或缺乏多样性的内容。
- **束搜索：** 生成质量高，但计算复杂度较高。
- **温度采样：** 可调节生成内容的多样性和随机性。

- **Top-k和Top-p采样：** 提高生成质量，同时保持灵活性。

生成策略选择建议

- **贪婪搜索：** 用于生成确定性较高的内容。
- **束搜索：** 用于生成质量要求较高的内容。
- **温度采样：** 用于调整生成的随机性。
- **Top-k和Top-p采样：** 用于确保生成内容的多样性。

LLM的Token与分词器

LLM（大型语言模型）中的 **Token** 是模型在处理文本时使用的基本单位。

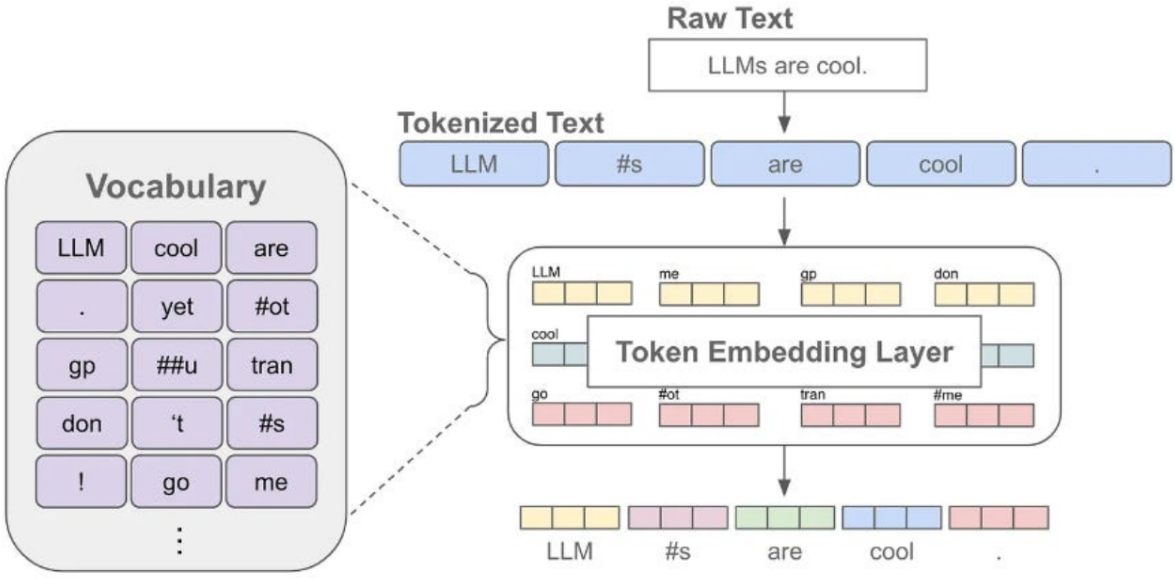
它可以是单词、子词甚至字符，具体取决于模型的分词策略。不同模型和库对 Token 的定义和分词方式可能有所不同。

以下是关于 LLM 中 Token 的一些关键点：

Token 的概念

- **单词级别的 Token**：完整的单词作为一个 Token，例如 "apple"、"banana"。
- **子词级别的 Token**：单词被分成更小的单元。例如，“unhappiness” 可以被分成 "un", "happiness" 或 "un", "hap", "pi", "ness" 等子词。
- **字符级别的 Token**：每个字符都是一个 Token，例如 "a", "p", "p", "l", "e"。

分词器（Tokenizer）



分词器用于将文本转换为模型可以理解的 Token 序列。常见的分词方式包括：

- **字典分词**：使用预定义的词汇表将输入文本与词汇表中的词匹配。
- **BPE（Byte-Pair Encoding）**：一种子词分词方法，将常见的字符或字符组合编码成 Token。
- **SentencePiece**：类似 BPE 的子词分词方法，通常用于处理多种语言。
- **WordPiece**：BERT 模型使用的分词方法，通过子词组合生成更大的词汇。

特殊 Token

在 LLM 中，通常还会使用一些特殊的 Token 来标识不同的功能：

- **[CLS]**：用于表示句子或文本的开始。
- **[SEP]**：分隔不同句子。
- **[PAD]**：用于填充对齐不同长度的输入。
- **[UNK]**：表示未知的 Token，通常是词汇表中未包含的单词或符号。
- **[MASK]**：用于掩码语言模型中的填补位置（如 BERT 的预训练任务）。

Token 计数

- 在 LLM 的推理过程中，输入和输出的 Token 数量对计算资源和费用具有重要影响。

  例如，GPT-3 使用的分词策略通常是基于字节对编码（BPE），平均每个单词可能会被分成 1.3 个 Token。

示例

以下是将一段文本转换为 Token 的示例：

使用 `tikToken` 库对文本进行分词，可以按照以下方式实现：

```
import tikToken

# 使用 GPT-3 模型的编码器
encoder = tikToken.get_encoding('gpt2')

text = "Hello, I'm an AI assistant."

# 对文本进行分词并获取 Token ID
Token_ids = encoder.encode(text)
Tokens = [encoder.decode([tid]) for tid in Token_ids]

print(f"Tokens: {Tokens}")
print(f"Token IDs: {Token_ids}")
```

这段代码的输出将会是：

```
Tokens: ['Hello', ',', ' I', "'m", ' an', ' AI', ' assistant', '.']
Token IDs: [15496, 11, 314, 39, 346, 616, 3265, 13]
```

输出：

在这个例子中，“Hello, I'm an AI assistant.” 被转换为 8 个 Token。

Llama 2 和 Llama 3 的分词器

Llama 2 的分词器：BPE 和 SentencePiece

Llama 2 使用了基于字节对编码（Byte Pair Encoding,BPE）算法的分词器。

BPE 是一种无监督的分词算法, 通过迭代地合并语料库中最频繁出现的字符对来构建词汇表。

具体来说,Llama 2 的分词器从字符级别开始, 找出语料库中出现频率最高的字符对并合并, 将合并后的字符对加入词汇表, 然后重复这个过程, 直到达到预设的词汇表大小或无法继续合并为止。

BPE 算法的优点在于, 它能够灵活地处理未知词（out-of-vocabulary, OOV）, 同时可以控制词汇表小。

通过将词分解为更小的子词单元, BPE 可以用有限的词汇表对词汇进行编码, 即使是未在训练数据中出现过的新词。

这使得模型能够更好地泛化到新的语言数据。

除了 BPE, Llama 2 的分词器还使用了 SentencePiece。

SentencePiece 是一个无监督的文本编码器, 提供了统一的接口, 支持 BPE、WordPiece 和 Unigram 等多种分词算法。

SentencePiece 的优势在于其灵活性和一致性。

它直接在原始文本上操作, 不依赖于预处理或语言特定的特征（如空格）, 因此可以轻松地适应不同的语言和领域。

Llama 3 的分词器：更大的词汇量和 Tiktoken

在 Llama 3 中,分词器进行了显著的升级。

首先,词汇量从 Llama 2 的 32,000 个令牌大幅增加到了 128,256 个令牌。

更大的词汇量使得模型能够更精细地编码输入和输出文本,提高了编码效率和下游任务性能。

其次, Llama 3 从 SentencePiece 转向了 Tiktoken 分词器。

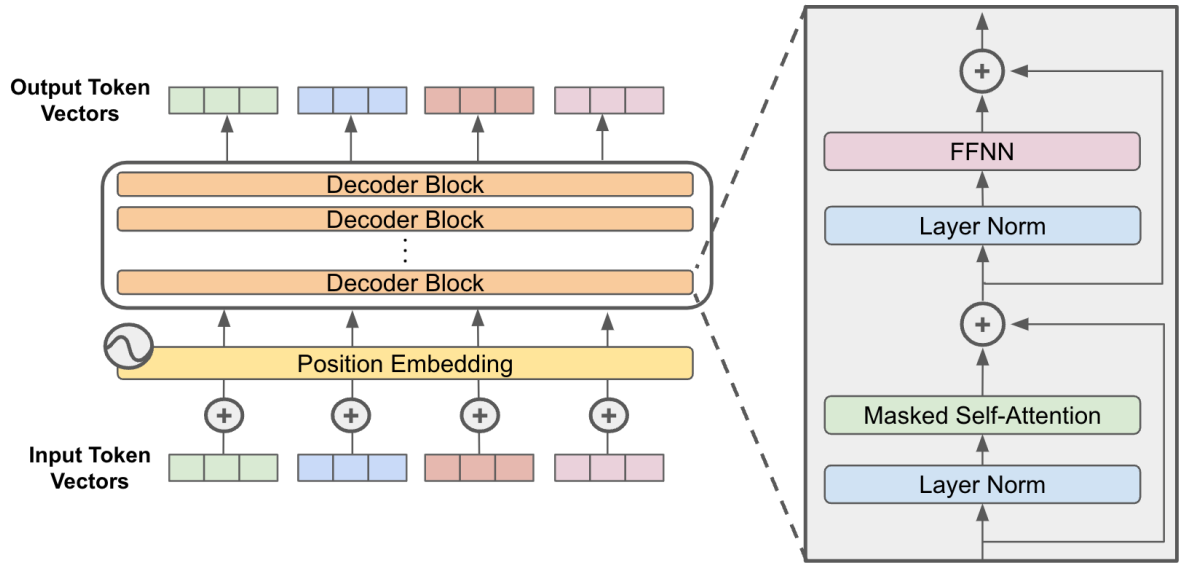
Tiktoken 是由 OpenAI 开发的现代分词和编码工具, 旨在更高效、更灵活地处理各种语言和文本数据。

Llama 3 选择 Tiktoken 可能基于其在编码效率和多语言支持方面的优势, 以及与 GPT 系列模型保持一致的考虑。

此外, Llama 3 的分词器可能还包含了一些特定的优化, 以适应其更大规模的模型和训练数据。

虽然具体细节尚未公开, 但我们可以合理地推测, 这些改进可能与提高分词效率、降低编码噪音以及更好地处理多语言数据有关。

LLM的文本生成过程



这个过程通常包括以下几个步骤：

1. **输入阶段（Input）**：
    - 用户提供一个初始输入，这可以是一个提示、问题或部分文本，模型将基于此生成文本。

2. **分词（Tokenization）**：
   - 输入文本被转换成一系列的Token（Tokens）。分词器会将输入文本分割成更小的单元，这些单元可以是单词、子词或字符，这取决于模型的分词策略。
3. **嵌入（Embedding）**：
   - 每个Token被转换成一个固定长度的向量，这个过程称为嵌入。这些向量通常通过预训练的词嵌入模型生成，它们能够捕捉词汇的语义信息。
4. **位置编码（Positional Encoding）**：
   - 为了保持词序信息，因为Transformer模型本身不具备捕捉顺序的能力，通常会给每个嵌入向量添加一个位置编码。
5. **Transformer处理**：
   - 经过嵌入和位置编码的输入被送入Transformer模型。Transformer模型由多个相同的层组成，每层都包括自注意力（Self-Attention）机制和前馈神经网络（Feed-Forward Neural Network）。
   - 自注意力机制允许模型在处理当前词时考虑到句子中的其他词，这有助于捕捉文本中的长距离依赖关系。
6. **输出转换（Output Transformation）**：
   - Transformer模型的最后一层输出被送入一个线性层，将输出向量映射到模型词汇表大小的维度空间。
7. **Softmax函数**：
   - 应用Softmax函数将线性层的输出转换为概率分布，每个Token对应的概率表示模型预测该Token是下一个词的可能性。
8. **采样（Sampling）**：
   - 基于Softmax函数得到的概率分布，模型进行采样以选择下一个生成的Token。采样策略可以是贪心采样、随机采样、核采样（Nucleus Sampling）或温度采样（Temperature Sampling）等。
9. **生成文本**：
   - 重复采样过程，直到生成了足够长度的文本或直到遇到句子结束的Token。
10. **后处理（Post-processing）**：
    - 生成的文本可能需要一些后处理，如去除多余的特殊Token，或者进行语法和语义的修正。

请注意，这个过程是非常简化的描述，实际的大型语言模型可能包含更多的细节和优化步骤。

而且，不同的模型和框架可能会在具体实现上有所差异。

举例：

使用 `tikToken` 库对输入文本进行分词并获取嵌入表示，可以按照以下方式实现。

我们将使用 OpenAI 的 GPT 模型以及 `torch` 库来加载预训练模型并计算嵌入。

```
import torch
import tikToken
from transformers import GPT2Model

# 使用 tikToken 初始化 GPT-2 编码器
encoder = tikToken.get_encoding("gpt2")

# 加载预训练的 GPT-2 模型
model = GPT2Model.from_pretrained("gpt2")

# 输入文本
text = "Hello, I'm an AI assistant."
```

```
# 使用 tikToken 对输入文本进行分词
Token_ids = encoder.encode(text)
Tokens = [encoder.decode([tid]) for tid in Token_ids]

# 将 Token IDs 转换为 PyTorch 张量并添加批次维度
inputs = torch.tensor([Token_ids])

# 获取模型的输出
with torch.no_grad():
    outputs = model(inputs)

# 获取嵌入层的输出表示（最后一层的隐状态）
last_hidden_states = outputs.last_hidden_state

# 打印每个 Token 的嵌入向量
for Token, emb in zip(Tokens, last_hidden_states[0]):
    print(f"Token: {Token}\nEmbedding: {emb}\n")
```

输出示例

```
Token: Hello
Embedding: tensor([ 0.0168, -0.0136, ..., -0.0234,  0.0216])

Token: ,
Embedding: tensor([-0.0008,  0.0053, ..., -0.0189,  0.0013])

Token:  I
Embedding: tensor([ 0.0238, -0.0272, ...,  0.0128, -0.0042])

Token: 'm
Embedding: tensor([ 0.0086,  0.0103, ...,  0.0175, -0.0069])

Token:  an
Embedding: tensor([ 0.0123, -0.0198, ..., -0.0028,  0.0084])

Token:  AI
Embedding: tensor([ 0.0038, -0.0097, ...,  0.0152, -0.0141])

Token:  assistant
Embedding: tensor([-0.0102, -0.0124, ...,  0.0216,  0.0001])

Token: .
Embedding: tensor([ 0.0033, -0.0131, ...,  0.0006,  0.0120])
```

解释

- **Token IDs**：输入文本被编码为 Token 序列的索引。
- **嵌入表示**：每个 Token 被转换为一个固定长度的向量表示，捕捉到 Token 的语义信息。

这样，通过 `tikToken` 库分词并结合预训练模型获取嵌入表示，模型能够在后续处理阶段使用这些丰富的语义表示。

在这个例子中，输入给模型的 `inputs` 是一个 PyTorch 张量，维度为 `(batch_size, seq_length)`，其中：

- `batch_size`：表示一次输入的样本数量。
- `seq_length`：表示输入序列的长度，即 Token 的数量。

以下通过举例来说明后续步骤在LLM的文本生成过程中的作用：

1. 位置编码（Positional Encoding）

Transformer 模型本身不具备顺序信息，所以需要为每个嵌入向量添加位置编码，以确保模型能够识别出词语的顺序。

```
import torch
import math

def positional_encoding(seq_length, d_model):
    """为序列添加位置编码"""
    pe = torch.zeros(seq_length, d_model)
    position = torch.arange(0, seq_length, dtype=torch.float).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-
math.log(10000.0) / d_model))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    return pe

# 假设我们有一个序列长度为 10，嵌入维度为 16
pos_encoding = positional_encoding(10, 16)
print(pos_encoding)
```

输出示例：

```
tensor([[ 0.0000,  1.0000,  0.0000,  1.0000, ...,  0.0000,  1.0000,  0.0000,
1.0000],
        [ 0.8415,  0.5403,  0.0707,  0.9975, ...,  0.0002,  1.0000,  0.0001,
1.0000],
        ...])
```

2. Transformer 处理

将嵌入和位置编码后的输入送入 Transformer 模型。以下代码展示一个简单的 Transformer 模型结构：

```
import torch.nn as nn

class SimpleTransformer(nn.Module):
    def __init__(self, input_dim, model_dim, num_heads, num_layers):
        super(SimpleTransformer, self).__init__()
        self.embedding = nn.Embedding(input_dim, model_dim)
        self.position_encoding = positional_encoding(5000, model_dim)
        self.transformer_layers = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(model_dim, num_heads, model_dim*4),
num_layers
        )
        self.fc_out = nn.Linear(model_dim, input_dim)

    def forward(self, x):
```

```
        x = self.embedding(x) + self.position_encoding[:x.size(1), :]
        x = self.transformer_layers(x)
        return self.fc_out(x)

# 示例
model = SimpleTransformer(input_dim=10000, model_dim=512, num_heads=8,
num_layers=6)
input_seq = torch.randint(0, 10000, (10, 20))  # (batch_size, seq_length)
output_seq = model(input_seq)
```

3. 输出转换（Output Transformation）

Transformer 模型的最后一层输出通过线性层将其映射到词汇表的大小。

```
output_layer = nn.Linear(512, 10000)
output_seq_transformed = output_layer(output_seq)
print(output_seq_transformed.shape)
```

输出示例：

```
torch.Size([10, 20, 10000])  # (batch_size, seq_length, vocab_size)
```

4. Softmax 函数

应用 Softmax 函数将线性层的输出转换为每个 Token 对应的概率分布。

```
softmax = nn.Softmax(dim=-1)
probabilities = softmax(output_seq_transformed)
print(probabilities.shape)
```

输出示例：

```
torch.Size([10, 20, 10000])  # (batch_size, seq_length, vocab_size)
```

5. 采样（Sampling）

根据 Softmax 输出的概率分布，使用不同策略选择下一个生成的 Token。

- **贪心采样（Greedy Sampling）**：选择概率最高的标记。

```
predicted_Tokens = torch.argmax(probabilities, dim=-1)
print(predicted_Tokens)
```

- **随机采样（Random Sampling）**：基于概率分布随机选择标记。

```
predicted_Tokens_random = torch.multinomial(probabilities.view(-1,
probabilities.size(-1)), 1).view(probabilities.size()[:-1])
print(predicted_Tokens_random)
```

6. 生成文本

通过不断采样和更新输入序列，生成完整的文本。此过程可使用循环来实现。

```
# 示例：生成文本的循环流程
def generate_text(model, Tokenizer, start_text, max_length):
    Tokens = Tokenizer.encode(start_text)
    generated = Tokens.copy()

    for _ in range(max_length - len(Tokens)):
        input_tensor = torch.tensor([generated])
        with torch.no_grad():
            output = model(input_tensor)
            probabilities = softmax(output[:, -1, :])
            next_Token = torch.argmax(probabilities, dim=-1).item()
        generated.append(next_Token)
        if next_Token == Tokenizer.eos_Token_id:  # 终止符
            break

    return Tokenizer.decode(generated)

# 使用示例
from transformers import GPT2Tokenizer, GPT2Model

# 加载 GPT-2 模型
Tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2Model.from_pretrained("gpt2")

start_text = "Hello, I'm an AI assistant"
generated_text = generate_text(model, Tokenizer, start_text, max_length=50)
print(generated_text)
```

7. 后处理（Post-processing）

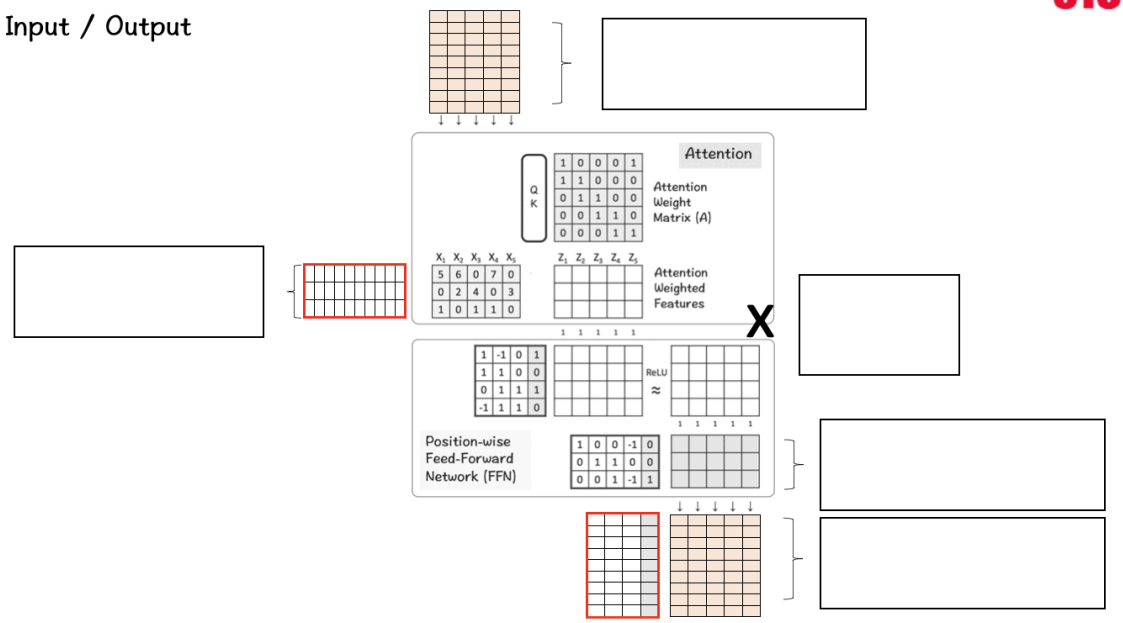
对生成的文本进行必要的后处理，例如去除特殊 Token 或修正语法和语义错误。

```
# 示例：后处理生成的文本
def post_process(text):
    text = text.replace("<pad>", "").replace("<s>", "").replace("</s>",
"").strip()
    return text

cleaned_text = post_process(generated_text)
print(cleaned_text)
```

通过这些步骤可以看到，LLM 是如何通过自注意力机制和自回归生成方式，逐步生成连贯文本的。

Llama3的文本生成



步骤 1：首先，输入文本经过分词器，会产生一个大小为 seq_len (8K上下文窗口大小）的输入Token序列。

实际上，分词器会将 Tokens 映射到词汇表（词汇表大小为128K词汇量）中的对应 Token IDs。

步骤 2：通过嵌入矩阵，将 Token IDs 映射为大小为 seq_len x 4096 的嵌入表示矩阵，4096 是 LLaMA 模型中指定的特征维度。

步骤 3：这个特征矩阵经过 Transformer 块，首先由注意力层进行处理，然后经过前馈网络（FFN）层。

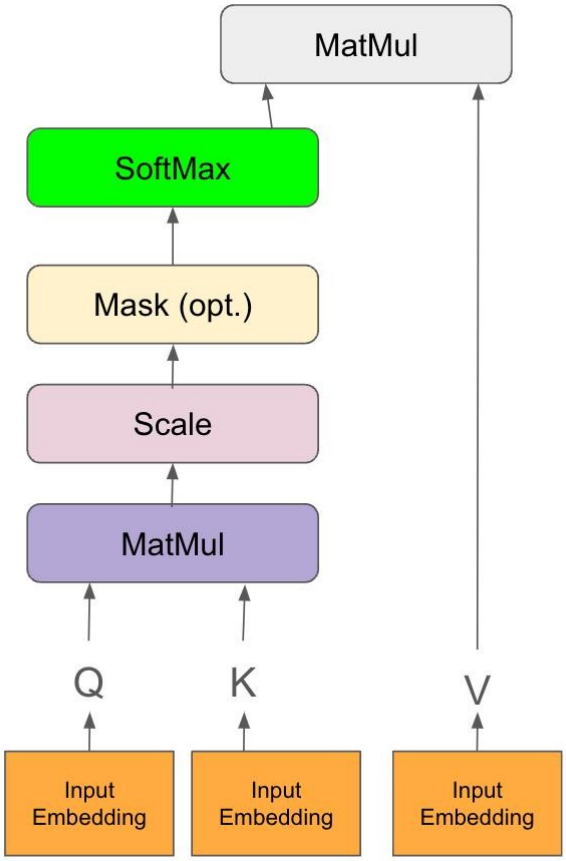
注意力层在特征上水平处理，而 FFN 层在维度上垂直处理。

步骤 4：步骤 3 在 Transformer 块中重复 32 层。最终，得到的矩阵与特征维度保持相同大小，即 seq_len x 4096。

步骤 5：最后，将这个矩阵通过分词器的解码转换回词汇表的大小 128K，以便模型可以选择并映射词汇表中的单词，生成最终输出。

LLM文本生成的Q、K、V

在大语言模型(LLM)的文本生成任务中,Q、K、V分别代表Query(查询)、Key(键)和Value(值)。



这三个概念源于注意力机制(Attention Mechanism), 特别是自注意力(Self-Attention)机制, 它们在Transformer架构中扮演着关键角色。

1. Query(Q): 表示当前正在处理的词或词组。在自注意力机制中, 每个词都被视为一个Query。

   对于生成任务, Query通常是前面已生成的词或词组。

   我们希望了解当前Query与其他词(Key-Value对)之间的相关性, 以决定如何生成下一个词。

2. Key(K): 表示一组候选词或词组, 用于与Query进行相关性计算。在自注意力机制中, 每个词也同时扮演Key的角色。

   通过计算Query与不同Key之间的相似度(通常使用点积或其他相似度函数), 我们可以得到一个注意力分布, 表示当前Query与每个Key的相关程度。

3. Value(V): 与Key相对应, 表示每个候选词或词组的实际表示或含义。

   一旦我们根据Query和Key的相关性计算出注意力分布, 就可以使用这个分布对Value进行加权求和, 得到当前Query的上下文表示。

   这个上下文表示融合了与当前Query相关的所有词的信息, 用于指导下一个词的生成。

在Transformer的自注意力层中, 输入序列首先被映射到三个不同的矩阵: Q矩阵、K矩阵和V矩阵。

这三个矩阵的维度通常为(序列长度, 隐藏层大小)。然后, 通过将Q矩阵与K矩阵转置相乘, 并除以一个缩放因子(通常为隐藏层大小的平方根), 得到注意力分布。最后, 将注意力分布与V矩阵相乘, 得到输出表示。

对于生成任务, Transformer的解码器部分会根据前面生成的词(作为Query)与编码器输出或解码器自身的隐藏状态(作为Key-Value对)进行交互, 逐步生成后续的词。这个过程反复进行, 直到生成结束Token或达到最大长度限制。

总之, Q、K、V机制使得大语言模型能够动态地关注与当前生成相关的上下文信息, 从而生成更加连贯、自然的文本。

这种机制也使得模型能够处理长距离依赖关系, 捕捉词与词之间的复杂交互。

理解Q、K、V的作用和计算过程, 对于深入认识大语言模型的文本生成能力至关重要。

让我们以"Paris is the city"作为提示(Prompt), 看看大语言模型如何利用Q、K、V机制生成后续文本

假设我们要生成的下一个词是"of"。

在我们的例子中,"of"可以被视为一种"预填充"(prefill)或"预测"(prediction)阶段得到的Token。

在实际的文本生成过程中, 大语言模型通常会使用一种称为"自回归"(autoregressive)的方法, 即根据前面已生成的词来预测下一个词。

这个过程可以分为两个阶段:

1. 预填充(Prefill)阶段: 在这个阶段, 模型根据前面的上下文(如"Paris is the city")和其内部的语言知识, 预测出最可能的下一个词。

   在我们的例子中, 模型预测"of"是一个高概率的候选词。这个预测过程本质上就是通过Q、K、V机制, 利用注意力权重和上下文表示来估计每个词的概率分布。

2. 采样(Sampling)阶段: 一旦模型得到了下一个词的概率分布 ,它就可以从这个分布中采样出一个实际的词。

   常见的采样策略包括贪婪采样(选择概率最高的词)、随机采样(根据概率随机选择)、束搜索(保留多个高概率候选)等。采样得到的词会被添加到已生成的序列中,然后模型再次进入预填充阶段,预测下一个词。

在我们的演示中, 我们假设模型在预填充阶段预测"of"是下一个最可能的词, 然后直接将其作为生成的词, 以便更清晰地展示Q、K、V机制的计算过程。

假设我们要生成的下一个词是"of"。此时, "of"对应的Query向量(Q)会与前面已生成词("Paris", "is", "the", "city")的Key向量(K)进行交互。

1. 首先, 模型将"of"的嵌入向量作为Query(Q), 将"Paris","is","the","city"的嵌入向量分别作为Key(K)。
2. 然后, 模型计算Query与每个Key的相似度得分。在这个例子中,"of"可能与"city"的相似度最高, 因为"city of"是一个常见的短语。相似度得分可以表示为:

   ```
   Similarity("of", "Paris") = 0.1
   Similarity("of", "is") = 0.2
   Similarity("of", "the") = 0.3
   Similarity("of", "city") = 0.8
   ```

3. 接下来, 模型将这些相似度得分转化为注意力权重。这通常通过Softmax函数实现, 以确保所有权重的和为1:

   ```
   Attention_weights = Softmax([0.1, 0.2, 0.3, 0.8])
                     = [0.05, 0.07, 0.10, 0.78]
   ```

4. 最后, 模型使用注意力权重对每个词的Value向量(V)进行加权求和。Value向量表示每个词的实际语义表示。假设我们有:

   ```
   Value("Paris") = [0.2, 0.1, ..., 0.5]
   Value("is") = [0.3, 0.2, ..., 0.4]
   Value("the") = [0.1, 0.3, ..., 0.2]
   Value("city") = [0.5, 0.2, ..., 0.9]
   ```

   那么,"of"的上下文表示(Context)就是:

   ```
   Context("of") = 0.05 * Value("Paris") + 0.07 * Value("is")
                 + 0.10 * Value("the") + 0.78 * Value("city")
                 = [0.44, 0.20, ..., 0.79]
   ```

这个上下文表示融合了与"of"最相关的词("city")的语义信息, 同时也考虑了其他词的影响。

模型将使用这个上下文表示来预测下一个词, 比如"light", "love", "fashion"等。

通过这种方式, 大语言模型可以动态地关注与当前生成最相关的上下文, 并根据这些信息生成连贯、的后续文本。

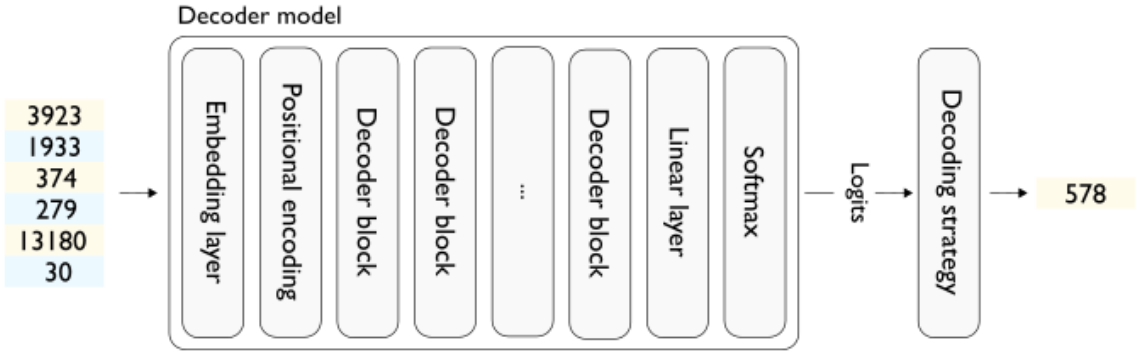
在实际的模型中, 这个过程会重复多次(多头注意力机制), 并通过多个Transformer层进行处理, 以捕捉更复杂、更抽象的语言模式和依赖关系。

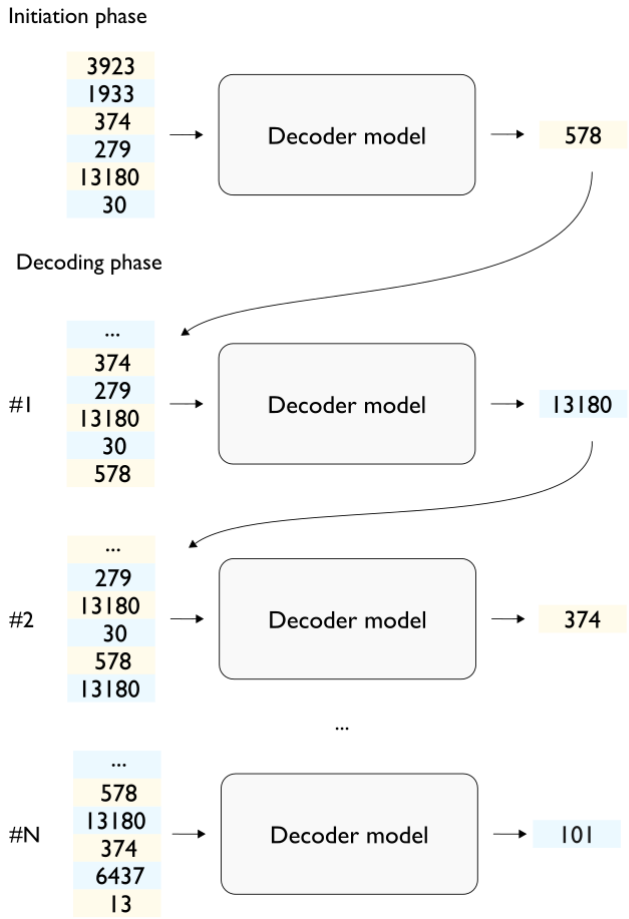
LLM文本生成的两个阶段：预填充（prefill）和解码（decode）

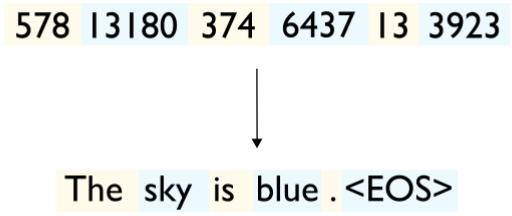
在LLM生成文本的过程中，通常会涉及两个阶段：预填充（prefill）和解码（decode）。

这两个阶段对于模型生成连贯、准确和高效的文本非常重要。

下面详细解释这两个阶段的原理及其在大型语言模型中的应用。







预填充（Prefill）阶段

预填充阶段主要用于准备初始上下文，为模型生成后续文本提供基础。

在这个阶段，模型会处理输入的初始文本（通常是用户提供的提示或上下文），并生成相应的内部状态（如KV缓存）。

具体过程

1. **初始上下文输入：**
   - 用户提供一个初始文本作为提示，例如问题、句子或段落。
   - 该文本通常被编码为一系列Token（Tokens）。
2. **初始上下文的注意力计算：**
   - 模型对输入的所有Token进行处理，生成键和值，并存储在KV缓存中。
3. **初始概率分布：**
   - 计算最后一个Token的概率分布，作为生成下一个Token的起始点。

解码（Decode）阶段

解码阶段是模型根据预填充阶段准备好的上下文，生成后续文本的过程。

在这个阶段，模型会逐步生成每个新Token，并在每次生成后更新KV缓存。

具体过程

1. **Token生成：**
   - 从预填充阶段的初始概率分布中采样一个新Token，作为下一个输入。
   - 根据用户选择的策略（如贪心、采样、温度）进行采样。
2. **KV缓存更新：**
   - 使用新Token进行前向传递，更新KV缓存，生成新的键和值。
   - 生成新的概率分布，用于采样下一个Token。
3. **循环迭代：**
   - 重复步骤1和步骤2，直到达到生成的最大长度或满足停止条件。

预填充和解码阶段的意义

1. **预填充阶段的意义：**
   - 为模型提供初始的上下文，使得生成的文本更连贯和准确。
   - 预先计算KV缓存，减少解码阶段的计算量。
2. **解码阶段的意义：**
   - 根据预填充阶段的初始上下文逐步生成连贯的文本。
   - 通过选择不同的采样策略（如温度、Top-k、Top-p）控制生成的文本质量。

预填充（prefill）和解码（decode）是大型语言模型文本生成中的两个关键阶段。

预填充阶段为模型提供初始上下文，而解码阶段逐步生成文本。

通过合理利用这两个阶段，可以生成高质量且连贯的文本。