大模型(LLM)部署框架对比篇

来自： AiGC面试宝典

宁静致远 2024年03月31日 20:27

一、为什么需要对大模型推理加速？

大模型推理加速的目标是高吞吐量、低延迟。

- 吞吐量：一个系统可以并行处理的任务量。
- 延时：指一个系统串行处理一个任务时所花费的时间。

二、大模型(LLM)部署框架对比总览

框架	llama.cpp	rtp-llm	vllm	TensorRT-LLM	LMDeploy	fastllm
语言	C++ ggerganov/llama.cpp	Python alibaba/rtp-llm	Python vllm-project/vllm	TensorRT-LLM NVIDIA/TensorRT-LLM	LMDeploy InternLM/lmdeploy	fastllm zbxz16/fastllm
特点	1、纯C++推理加速， 无任何额外依赖 2、F16 和 F32混合精度 3、 支持4bit量化 4、 无需GPU，可只用CPU运行	1、高性能的 cuda kernel 2、支持 paged attention 、flash attention2、和 kv cache 量化	1、PagedAttention 2、高性能的 cuda kernel	1、高性能CUDA Kernel 2、量化	Continuous Batch，Blocked K/V Cache，动态拆分和融合，张量并行，高性能 kernel等重要特性。推理性能是 vLLM 的 1.8 倍可靠的量化支持权重量化和 k/v 量化。	1、纯c++实现 2、浮点模型（FP32），半精度模型(FP16)，量化模型(INT8, INT4) 加速 3、多卡部署，支持GPU + CPU混合部署支持并发计算时动态拼Batch全平台支持
多模态支持	√(llava、mobileVLM、Obsidian等)	√(llava、Qwen-VL)	×	×	√ (InternLM、Qwen-VL)	×
多平台支持	√	×	×	×	×	√
KV Cache	√	√	√	√	√	√
量化方法	int8、int4等各种精度的量化	kv cache 量化支持 weight only INT8 量化，支持加载时自动量化	GPTQ、AWQ、SqueezeLLM、FP8 KV 缓存	INT4/INT8 Weight-Only Quantization (W4A16 & W8A16)SmoothQuantGPTQAWQFP8	INT4 权重量化K/V 量化W8A8 量化	INT8, INT4
高性能Cuda Kernal	√	√	√	√	√	√
Flash Attention	√	√	√	√	√	√
Paged Attention	√	√	√	√	√	×
Continuous Batching	√	√	√	√	√	√
Speculative Decoding （投机采样）	√	√	√	√	×	×
Medusa	√	√	×	√	×	×

三、大模型(LLM)部署优化策略

3.1 大模型(LLM)部署优化策略——量化

3.1.1 介绍一下 大模型(LLM)量化方法?

大模型量化将 16 位、32 位浮点数的模型参数或激活量化为 4 位或 8 位整数能够有效降低模型存储空间和计算资源需求，同时加速推理速度。

3.1.2 一般会对大模型(LLM)中哪些模块进行量化?

量化对象包含**权重、激活、KV Cache 量化**

- **仅权重量化**，如：W4A16、AWQ 及 GPTQ 中的 W4A16，W8A16（权重量化为 INT8，激活仍为 BF16 或 FP16）
- **权重、激活量化**，如：SmoothQuant 中的 W8A8
- **KV Cache INT8 量化**，LLM 推理时，为了避免冗余计算，设计了 KV Cache 缓存机制，本质上是空间换时间，由于 KV Cache 的存在，对于支持越长的文本长度的 LLM， KV Cache 的显存占用越高。 KV Cache 的量化也是有很必要的。

3.1.3 大模型(LLM)量化方法

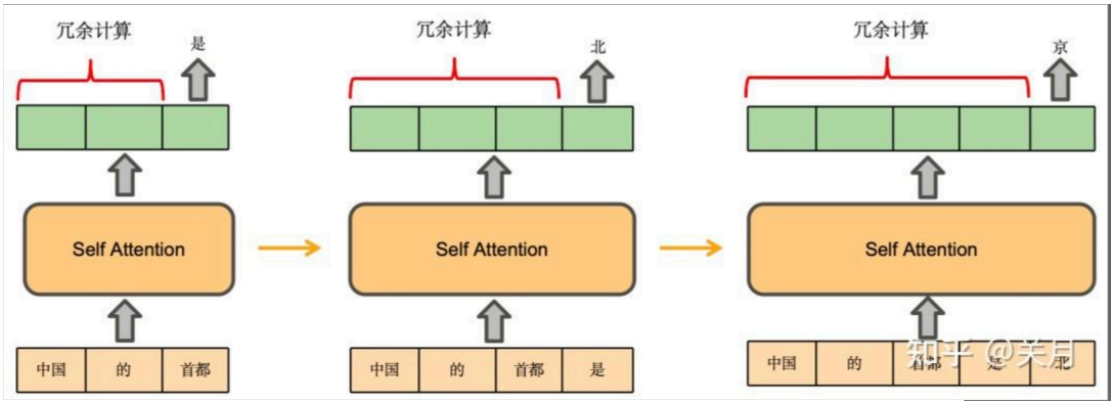
weight only INT8、KV Cache 量化、int4 量化、int8 量化

3.2 大模型(LLM)部署优化策略——KV Cache

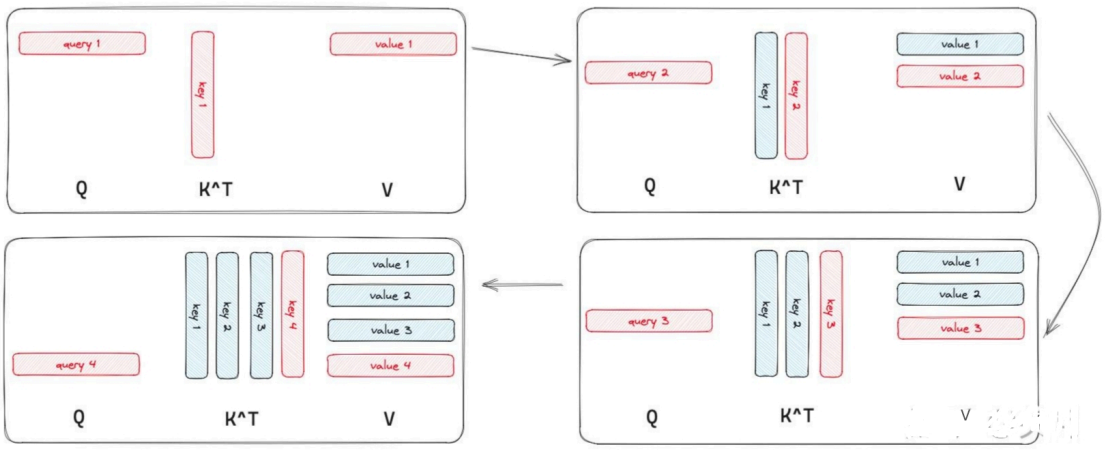
3.2.1 介绍一下 大模型(LLM) KV Cache 方法?

KV Cache 采用**以空间换时间的思想**，复用上次推理的 KV 缓存，可以极大降低内存压力、提高推理性能，而且不会影响任何计算精度。

3.2.2 为什么需要 大模型(LLM) KV Cache 方法?



以 GPT 为代表的一个 token 一个 token 往外蹦的 AIGC 大模型为例，里面最主要的结构就是 transformer 中的 self-attention 结构的堆叠，**实质是将之前计算过的 key-value 以及当前的 query 来生成下一个 token，LM 用于推理的时候就是不断基于前面的所有 token 生成下一个 token，每一轮用上一轮的输出当成新的输入让 LLM 预测，一般这个过程会持续到输出达到提前设定的最大长度或者是 LLM 自己生成了特殊的结束 token。其中有存在很多的重复计算，比如 token 的 Key Value 的计算，一个方法就是将过去 token 得到的 Key Value 存起来，Query 不需要存，Q 矩阵是依赖于输入的，因此每次都不同，无法进行缓存，因此 Q 矩阵通常不被缓存。**



3.3 大模型(LLM)部署优化策略——Flash Attention

3.3.1 介绍一下 大模型(LLM) Flash Attention 方法?

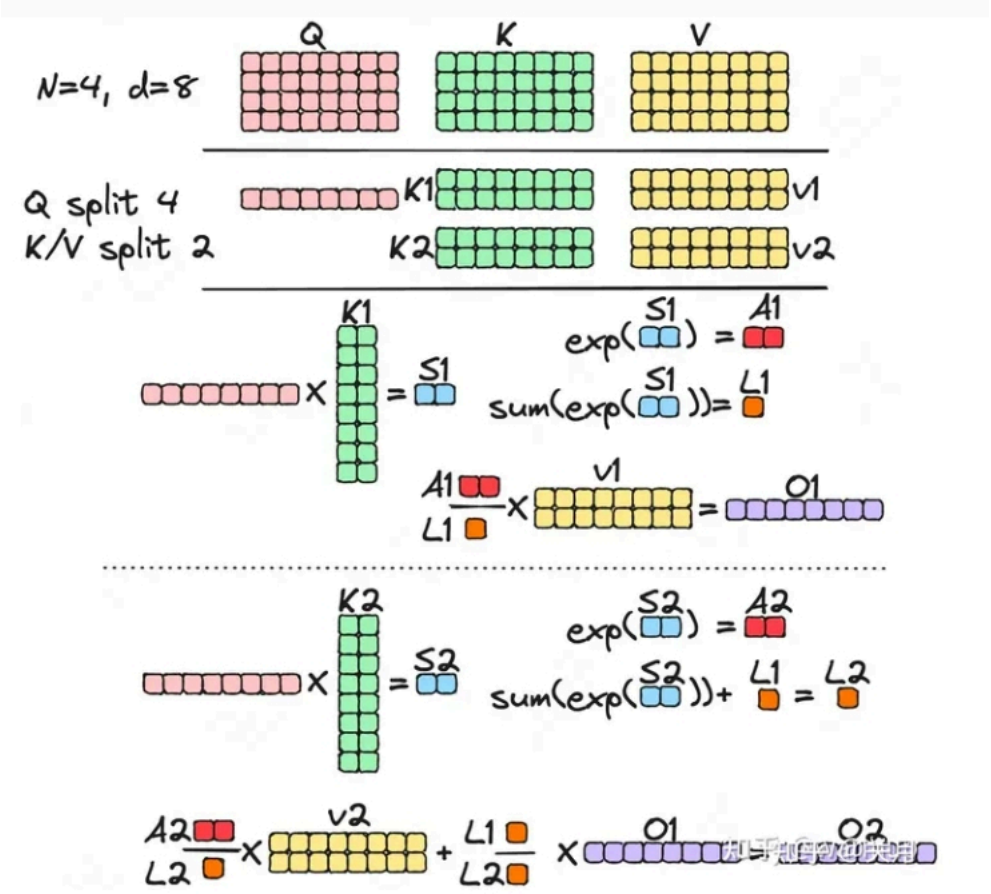
当输入序列（sequence length）较长时，Transformer 的计算过程缓慢且耗费内存，这是因为 self-attention 的 time 和 memory complexity 会随着 sequence length 的增加成二次增长。GPU 中存储单元主要有 HBM 和 SRAM，GPU 将数据从显存（HBM）加载至 on-chip 的 SRAM 中，然后由 SM 读取并进行计算，计算结果再通过 SRAM 返回给显存，HBM 容量大但是访问速度慢，SRAM 容量小却有着较高的访问速度。普通的 Attention 的计算过程如下，需要多次访问 HBM，Flash Attention 的目的就是通过分片+算子融合（矩阵乘法和 Softmax）减少对 HBM 的访问。

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

将矩阵分片运算，矩阵乘法这些简单，直接拆分计算就可以，但是 softmax 算子分片计算出来的不是最终的，Flash Attention 的通过另外的变量，将 softmax 优化成了一个可以分片运算的算子。



3.4 大模型(LLM)部署优化策略——Paged Attention

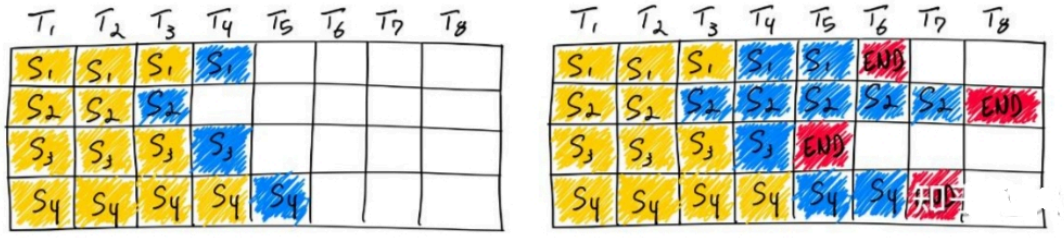
3.4.1 介绍一下 大模型(LLM) Paged Attention 方法?

PagedAttention 是对 kv cache 所占空间的分页管理，是一个典型的以内存空间换计算开销的手段，虽然 kv cache 很重要，但是 kv cache 所占的空间也确实是大且有浪费的，所以出现了 pagedattention 来解决浪费问题。kv cache 大小取决于 seqlen，然而这个东西对于每个 batch 里面的 seq 来说是变化的，毕竟不同的人输入不同长度的问题，模型有不同长度的答案回答，kv cache 统一按照 max seq len 来申请，造成现有 decoder 推理系统浪费了很多显存。

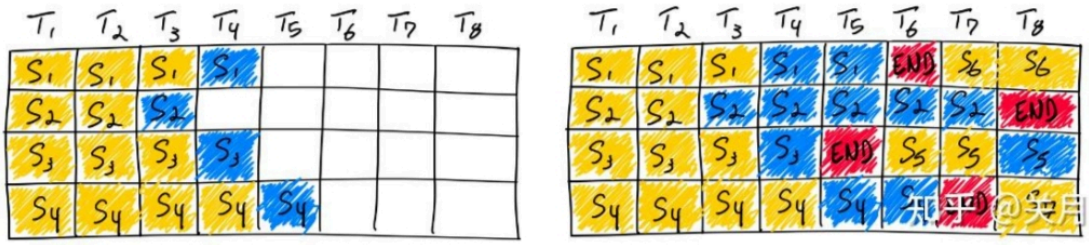
PagedAttention 将每个序列的 KV 缓存分成多个块，每个块包含固定数量的标记的键和值。在注意力计算过程中，PagedAttention Kernel 高效地识别和获取这些块，采用并行的方式加速计算。（和 ByteTransformer 的思想有点像）PagedAttention 的核心是一张表，类似于 OS 的 page table，这里叫 block table，记录每个 seq 的 kv 分布在哪个 physical block 上，通过把每个 seq 的 kv cache 划分为固定大小的 physical block，每个 block 包含了每个句子某几个 tokens 的一部分 kv，允许连续的 kv 可以不连续分布。在 attention compute 的时候，pagedattention CUDA kernel 就通过 block table 拿到对应的 physical block 序号，然后 CUDA 线程 ID 计算每个 seq 每个 token 的 offset 从而 fetch 相应的 block，拿到 kv，继续做 attention 的计算。

3.5 大模型(LLM)部署优化策略——Continuous batching

3.5.1 介绍一下 大模型(LLM) Continuous batching 方法?



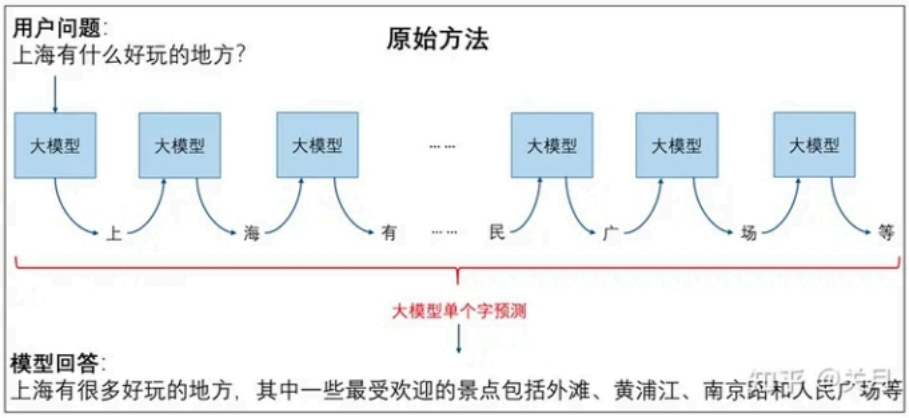
静态批处理：在第一遍迭代（左）中，每个序列从提示词（黄）中生成一个标记（蓝色）。经过几轮迭代（右）后，完成的序列具有不同的尺寸，因为每个序列在不同的迭代结束时产生不同的结束序列标记（红色）。尽管序列 3 在两次迭代后完成，但静态批处理意味着 GPU 将在批处理中的最后一个序列完成。

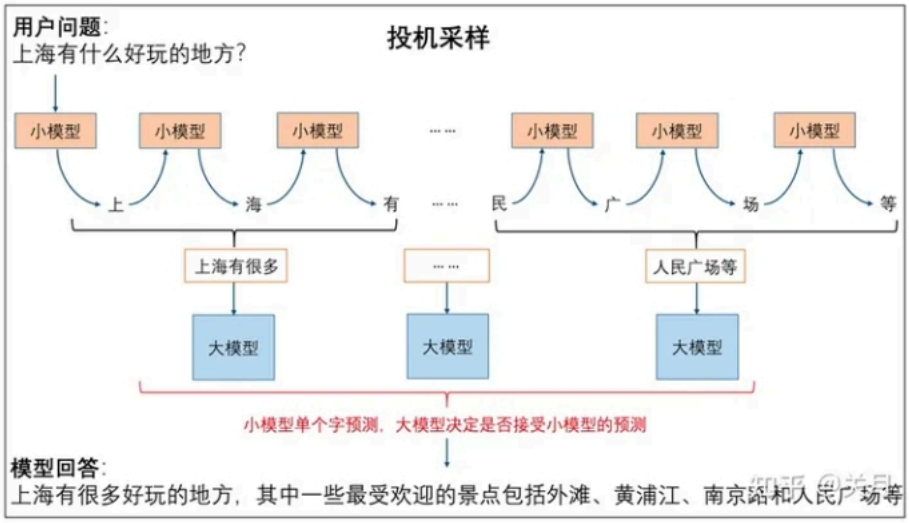


动态批处理：一旦批中的一个序列完成生成，就可以在其位置插入一个新的序列，从而实现比静态批处理更高的 GPU 利用率。

3.6 大模型(LLM)部署优化策略——Speculative Decoding

3.6.1 介绍一下 大模型(LLM) Speculative Decoding 方法?





投机采样的关键在于利用小模型多次推理单个字，让大模型进行多字预测，从而提升整体推理效率。每次小模型的单字推理耗时远远小于大模型，因此投机采样能够有效地提高推理效率。这种方法的优 势在于，通过蒸馏学习和投机采样，可以在减小模型规模的同时，保持较高的预测效果和推理速度，从而在实际部署中获得更好的性能优化。

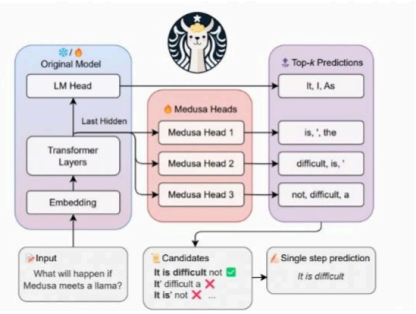
```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

prompt = "Alice and Bob"
checkpoint = "EleutherAI/pythia-1.4b-deduped"
assistant_checkpoint = "EleutherAI/pythia-160m-deduped"
device = "cuda" if torch.cuda.is_available() else "cpu"
```

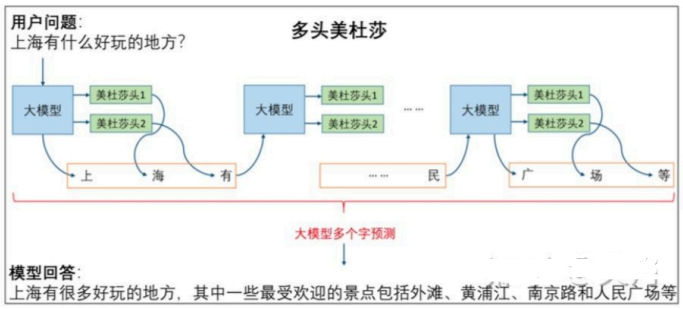
```
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
inputs = tokenizer(prompt, return_tensors="pt").to(device)
# ----------------------------------------------------------------------------------
model = AutoModelForCausalLM.from_pretrained(checkpoint).to(device)
assistant_model = AutoModelForCausalLM.from_pretrained(assistant_checkpoint).to(device)
# ----------------------------------------------------------------------------------
outputs = model.generate(**inputs, assistant_model=assistant_model)
print(tokenizer.batch_decode(outputs, skip_special_tokens=True))
# ['Alice and Bob are sitting in a bar. Alice is drinking a beer and Bob is drinking a']
```

3.7 大模型(LLM)部署优化策略——Medusa

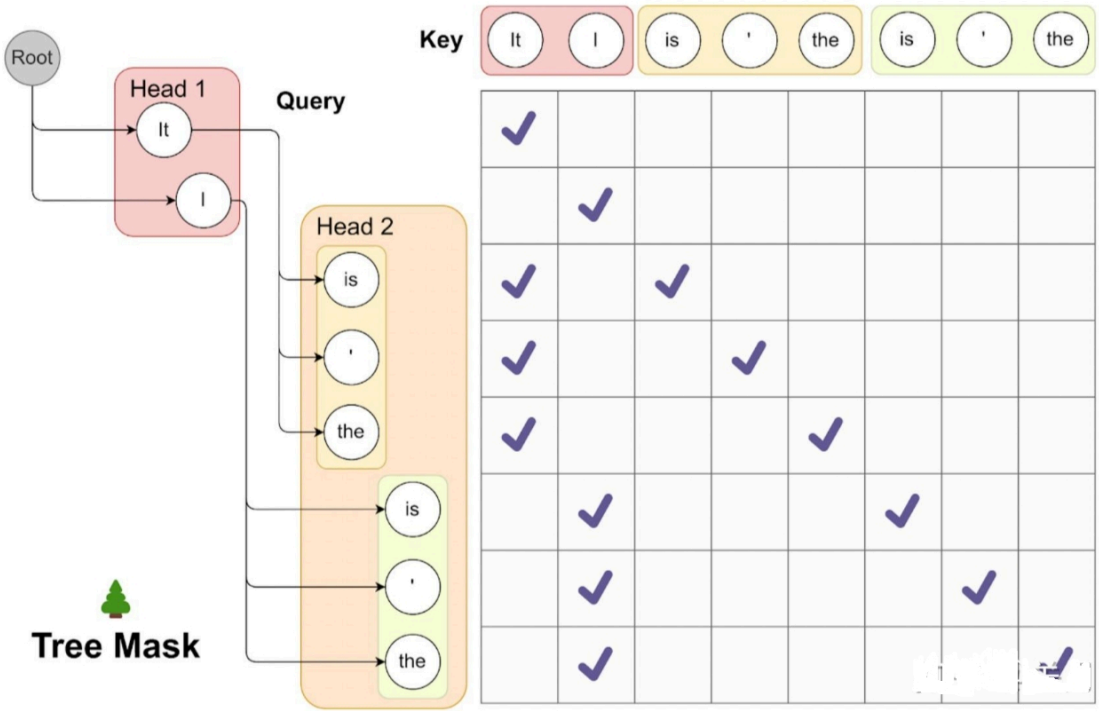
3.7.1 介绍一下 大模型(LLM) Medusa 方法?

一次生成多个词，相对于投机采样使用一个小模型一次生成多个词，主要思想是在正常的 LLM 的基础上，增加几个解码头，并且每个头预测的偏移量是不同的，比如原始的头预测第 i 个 token，而新增的 medusa heads 分别为预测第 i+1，i+2...个 token。如上图，并且每个头可以指定 topk 个结果，这样可以将所有的 topk 组装成一个一个的候选结果，最后选择最优的结果。





1. 多头美杜莎预测，记录 logits
2. 预测的 token 组合输入大模型，token 的组合太多了，每个分别输入大模型判断耗时太长，Madusa 提出了下图所示的树形注意力机制，预测的所有 token 可以同时输入进大模型，输出 logits 概率进行判别是否接受以及接收长度。
3. 选择最优，重复 1



多头美杜莎还会面临的一个问题是随着美杜莎头数量增加，top-k 的树状分支也将会以指数增长，造成庞大的计算开销。此外，许多基础和微调模型并没有开放其训练数据集，因此多头美杜莎面临的另一大问题是使用什么数据来训练美杜莎头。

知识星球