轻量级大语言模型MiniMind源码解读（七）：像搭积木一样，构建一个大语言模型

Original 凡希 南极Python 2026年1月24日 14:56

前文回顾：

轻量级大语言模型MiniMind源码解读（一）：如何从头训练tokenizer？

轻量级大语言模型MiniMind源码解读（二）：为什么RMSNorm更适合大模型推理?

轻量级大语言模型MiniMind源码解读（三）：原始Transformer的位置编码及其缺陷

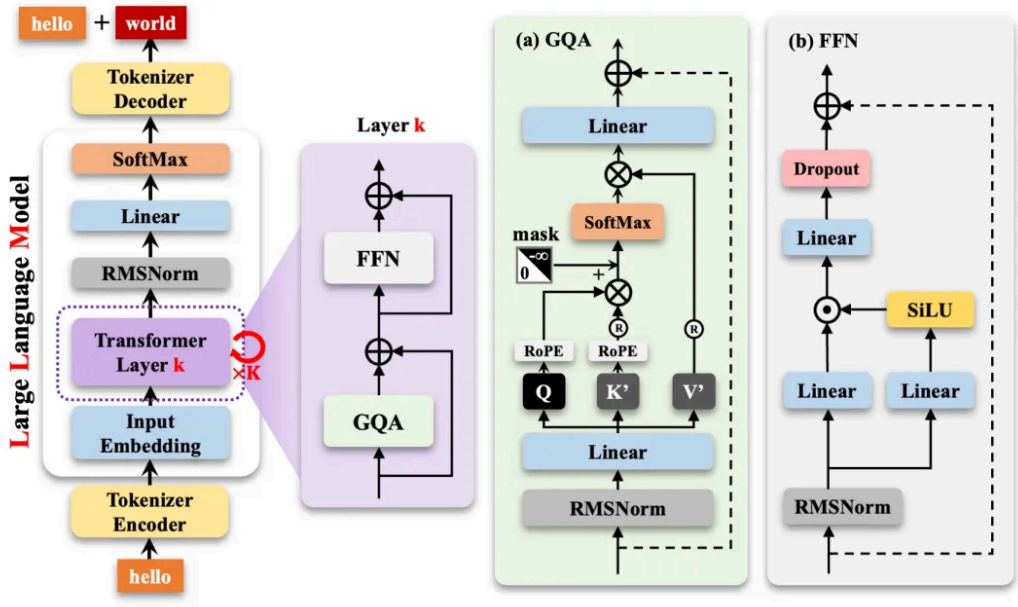
轻量级大语言模型MiniMind源码解读（四）：旋转位置编码原理与应用全解析

轻量级大语言模型MiniMind源码解读（五）：魔改的注意力机制，细数当代LLM的效率优化手段

轻量级大语言模型MiniMind源码解读（六）：从稠密到稀疏，详解专家混合模型MoE

通过前面六篇文章，我们已经完成了MiniMind中所有小组件的代码实现，将它们组合起来，就可以搭建MiniMind了。

这里展示的是MiniMind(Dense/MoE)的架构图



The Structure of MiniMind (Dense Model)

首先来搭建一个用于构建MiniMind的基础模块 `MiniMindBlock` ，对应MiniMind架构图中的 `Transformer Layer k` ：

```
class MiniMindBlock(nn.Module):
    def __init__(self, layer_id: int, config):
        super().__init__()
        # 基础配置
        self.num_attention_heads = config.num_attention_heads
        self.hidden_size = config.hidden_size
        self.head_dim = config.hidden_size // config.num_attention_heads

        # 自注意力模块，内部实现了 RoPE 相对位置编码
        self.self_attn = Attention(config)

        # 当前 Block 的层编号（可用于层内权重共享、分层控制等）
        self.layer_id = layer_id

        # Attention 前的 RMSNorm
        self.input_layernorm = RMSNorm(config.hidden_size, eps=config.rms_norm_eps)

        # Feed Forward 前的 RMSNorm
        self.post_attention_layernorm = RMSNorm(config.hidden_size, eps=config.rms_norm_e

        # 前馈网络模块，可配置是否使用专家混合（MoE）
        self.mlp = FeedForward(config) ifnot config.use_moe else MOEFeedForward(config)

    def forward(
        self,
        hidden_states,          # 输入的隐藏状态 [batch_size, seq_len, hidden_dim]
        position_embeddings,    # RoPE 位置编码 [seq_len, head_dim]
        past_key_value=None,    # KV 缓存 (key, value)，用于加速推理
        use_cache=False,        # 是否缓存当前层的 KV
        attention_mask=None     # attention 掩码
    ):
        # ---------------------- Self-Attention 层 ----------------------
        residual = hidden_states  # 保存残差连接

        # 对输入做 RMSNorm，再送入自注意力层
        hidden_states, present_key_value = self.self_attn(
```

```
            self.input_layernorm(hidden_states),  # LayerNorm 后输入 attention
            position_embeddings,                  # Rotary PE 传入 Attention
            past_key_value,                       # 过往 KV 缓存（一般在推理阶段使用）
            use_cache,                            # 是否缓存当前层 KV（一般在推理阶段使用）
            attention_mask                        # 注意力掩码（padding token不计算注意力矩阵）
        )

        # 残差连接：原始输入 + attention 输出
        hidden_states += residual

        # ---------------------- MLP 层 ----------------------
        # MLP 前再做一次 RMSNorm
        normed_hidden = self.post_attention_layernorm(hidden_states)

        # 残差连接：再加上 MLP 的输出
        hidden_states = hidden_states + self.mlp(normed_hidden)

        # 返回新的 hidden_states 和 当前层的 KV 缓存
        return hidden_states, present_key_value
```

现在来搭建MiniMind：

```
class MiniMindModel(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.vocab_size, self.num_hidden_layers = config.vocab_size, config.num_hidden_la

        # [vocab_size, hidden_size] -> 用于将 token id 映射为向量
        self.embed_tokens = nn.Embedding(config.vocab_size, config.hidden_size)

        self.dropout = nn.Dropout(config.dropout)

        # 构建多个 Transformer Block 层
        self.layers = nn.ModuleList([
            MiniMindBlock(l, config) for l in range(self.num_hidden_layers)
        ])

        # 输出前的 LayerNorm 层
        self.norm = RMSNorm(config.hidden_size, eps=config.rms_norm_eps)

        # 预计算 RoPE 所需的位置频率向量（cos/sin）
        freqs_cos, freqs_sin = precompute_freqs_cis(
            dim=config.hidden_size // config.num_attention_heads,
            end=config.max_position_embeddings,
            omiga=config.rope_theta
        )

        # 注册为 buffer（模型中持久存储但不参与优化）
        self.register_buffer("freqs_cos", freqs_cos, persistent=False)  # [max_pos, head_
        self.register_buffer("freqs_sin", freqs_sin, persistent=False)  # [max_pos, head_

    def forward(self,
                input_ids: Optional[torch.Tensor] = None,           # [B, T]
```

```
        attention_mask: Optional[torch.Tensor] = None,       # [B, 1, 1, T]（可选）
        past_key_values: Optional[List[Tuple[torch.Tensor, torch.Tensor]]] = None
        use_cache: bool = False,
        **kwargs):

    # B: batch_size, T: 当前输入 token 数量( T 在训练时就是seq_len，推理时就是当前已经生成的to
    batch_size, seq_length = input_ids.shape  # [B, T]

    # 如果没传入缓存，初始化为空（推理时才使用KV缓存）
    past_key_values = past_key_values or [None] * len(self.layers)

    # 获取历史缓存长度（增量生成时使用）
    start_pos = past_key_values[0][0].shape[1] if past_key_values[0] isnotNoneelse0

    # 输入 ids -> 嵌入向量：[B, T] → [B, T, hidden_size]
    hidden_states = self.dropout(self.embed_tokens(input_ids))  # [B, T, H]

    # 截取当前位置使用的旋转位置编码（RoPE）
    # freqs_cos: [max_pos, head_dim] → [T, head_dim]
    position_embeddings = (
        self.freqs_cos[start_pos:start_pos + seq_length],  # [T, D_head]
        self.freqs_sin[start_pos:start_pos + seq_length]   # [T, D_head]
    )

    presents = []  # 存储 KV cache：每层一个 (key, value)

    # 遍历每层 Transformer Block
    for layer_idx, (layer, past_key_value) in enumerate(zip(self.layers, past_key_val
        # 输入 hidden_states: [B, T, H]
        # 输出 hidden_states: [B, T, H]
        # 输出 present: (key, value)，用于缓存：[(B, T_total, H_kv, D), (B, T_total, H_k
        hidden_states, present = layer(
            hidden_states,                # [B, T, H]
            position_embeddings,          # ([T, D_head], [T, D_head])
            past_key_value=past_key_value,
            use_cache=use_cache,
            attention_mask=attention_mask
        )
        presents.append(present)

    # 最后输出 RMSNorm
    hidden_states = self.norm(hidden_states)  # [B, T, H]

    # 如果使用了 MOE（稀疏专家），则合并辅助损失
    aux_loss = sum(
        layer.mlp.aux_loss
        for layer in self.layers
        if isinstance(layer.mlp, MOEFeedForward)
    ) # 如果没有使用MOE，则 aux_loss = sum([]) = 0

    return hidden_states, presents, aux_loss
```

注意，回看MiniMind架构图，到现在为止，我们搭建的网络仅到RMSNorm层，后面的Linear和SoftMax还没有添加。

接下来需要进一步封装一个MiniMindForCausalLM类，这是为了更好地应用该模型于**因果语言建模任务（Causal Language Modeling）**，并增强其在推理、训练、生成等任务中的灵活性与兼容性。

因为虽然MiniMindModel已经实现了Transformer主干（包括嵌入层、注意力模块等核心组件），它只负责将输入的token ID编码为hidden states，属于“纯backbone”模块。

而MiniMindForCausalLM是一个“任务级封装”，它在主干模型基础上加上了输出层（language modeling head）和统一的输出结构，用于直接进行token-level的预测。

```
from transformers import PreTrainedModel, GenerationMixin, PretrainedConfig
from transformers.modeling_outputs import CausalLMOutputWithPast

# MiniMindConfig='' # 仅占位

class MiniMindForCausalLM(PreTrainedModel, GenerationMixin):
    # config_class = MiniMindConfig

    def __init__(self, config: MiniMindConfig = None):
        self.config = config or MiniMindConfig()
        super().__init__(self.config)

        # 模型主干：MiniMindModel，输出 hidden_states
        self.model = MiniMindModel(self.config)

        # 输出层：将 hidden_size 映射为 vocab_size（即每个 token 的 logits）
        self.lm_head = nn.Linear(self.config.hidden_size, self.config.vocab_size, bias=Fa

        # 权重绑定：embedding 权重与 lm_head 权重共享
        self.model.embed_tokens.weight = self.lm_head.weight

        # 输出容器（CausalLMOutputWithPast 结构体，方便 structured return）
        self.OUT = CausalLMOutputWithPast()

    def forward(self,
                input_ids: Optional[torch.Tensor] = None,                    # [batch_si
                attention_mask: Optional[torch.Tensor] = None,               # [batch_siz
                past_key_values: Optional[List[Tuple[torch.Tensor, torch.Tensor]]] = None
                use_cache: bool = True,
                logits_to_keep: Union[int, torch.Tensor] = 0,                # 控制 logits
                **args):

        # 调用主干模型，输出 hidden_states、presents（KV缓存）、aux_loss
        h, past_kvs, aux_loss = self.model(
            input_ids=input_ids,                                             # 输入 token 序
            attention_mask=attention_mask,                                   # 用于 mask pa
            past_key_values=past_key_values,                                 # 用于增量推理的
            use_cache=use_cache,                                             # 是否返回 KV c
            **args
        )
        # h.shape: [batch_size, total_seq_len, hidden_size]， 训练时，total_seq_len就是seq_l
```

```
        # past_kvs: List of (key, value), 每个层各一对
        # aux_loss: 用于 MOE 模型的辅助损失（如果使用MOE）

        # 根据 logits_to_keep 参数决定保留输出的哪些位置
        slice_indices = slice(-logits_to_keep, None) if isinstance(logits_to_keep, int) e

        # 从 h 中保留最后 logits_to_keep 个位置，送入 lm_head 做分类
        logits = self.lm_head(h[:, slice_indices, :]) # 训练时，slice_indices 是 0，logits
        # logits.shape: [batch_size, logits_to_keep, vocab_size]

        # 构建结构化输出字典
        self.OUT.__setitem__('last_hidden_state', h)          # [batch_size, seq_len, hid
        self.OUT.__setitem__('logits', logits)                # [batch_size, logits_to_ke
        self.OUT.__setitem__('aux_loss', aux_loss)            # scalar or tensor
        self.OUT.__setitem__('past_key_values', past_kvs)     # list of tuples: (key, val

        return self.OUT
```

为了能够无缝对接HuggingFace的训练、推理与生成框架，MiniMindForCausalLM 继承了 `PreTrainedModel` 和 `GenerationMixin`，并使用标准的输出结构 `CausalLMOutputWithPast`，从而实现了以下兼容性：

- 与 `transformers.Trainer` 配合训练时自动识别 `logits` 和 `loss`；
- 支持 `.generate()` 方法进行文本生成（增量推理、KV缓存、温度采样等）；
- 与 `LLaMA`、`GPT` 等结构保持一致，便于迁移预训练权重或微调脚本;
- 通过 `past_key_values` 的接口设计，MiniMindForCausalLM 支持增量推理场景下的KV缓存机制，显著提升生成速度.

也就是说，通过这些兼容性，后续的许多代码不需要再次手动实现，而是可以直接调用HuggingFace官方实现的接口，方便快捷且高效。

Read more