模型架构设计

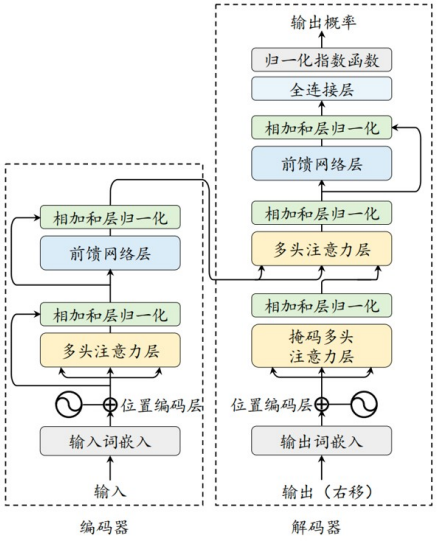
桂宁馨

Wuhan University

March 15, 2025

目录

1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构



Encoder-Decoder 结构

- 将输入的文本分词之后转换成词嵌入并和位置编码相加

$$x_t = v_t + p_t$$

- （掩码）多头自注意力 + 残差连接 + 层归一化

$$Q = XW^Q,\ K = XW^K,\ V = XW^V$$

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{D}})V$$

- 前馈网络层 + 残差连接 + 层归一化

$$\text{FFN}(X) = \sigma(XW^U + b_1)W^D + b_2$$

- 全连接层 + 归一化指数函数

$$O = \text{softmax}(W^L Y_L)$$

目录

1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构

- LayerNorm：

$$\text{LayerNorm}(x) = \frac{x-\mu}{\sigma} \cdot \gamma + \beta$$

$$\mu = \frac{1}{H}\sum_{i=1}^{H} x_i,\ \sigma = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(x_i-\mu)^2}$$

- RMSNorm：

$$\frac{x}{\text{RMS}(x)} \cdot \gamma$$

$$\text{RMS}(x) = \sqrt{\frac{1}{H}\sum_{i=1}^{H} x_i^2}$$

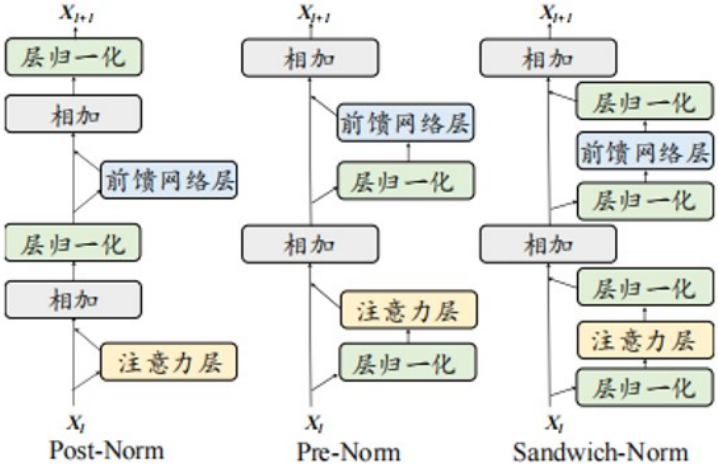
- DeepNorm：

$$\text{DeepNorm}(x) = \text{LayerNorm}(\alpha \cdot x + \text{Sublayer}(x))$$

```python
class LlamaRMSNorm(nn.Module):
    def __init__(self, hidden_size, eps=1e-6):
        super().__init__()
        self.weight = nn.Parameter(torch.ones(hidden_size))
        self.variance_epsilon = eps

    def forward(self, hidden_states):
        input_dtype = hidden_states.dtype
        hidden_states = hidden_states.to(torch.float32)
        variance = hidden_states.pow(2).mean(-1, keepdim=True)
        # 计算隐状态的均方根
        hidden_states = hidden_states * torch.rsqrt(variance +
        ↪  self.variance_epsilon)
        # 将隐状态除以其均方根后重新缩放
        return self.weight * hidden_states.to(input_dtype)
```

RMSNorm在LayerNorm的基础上，减少了需要训练的参数，并且没有影响模型的表现。

DeepNorm使大模型的训练更加稳定。


X_{l+1}
层归一化
相加
前馈网络层
层归一化
相加
注意力层
X_l
Post-Norm
X_{l+1}
相加
前馈网络层
层归一化
相加
注意力层
层归一化
X_l
Pre-Norm
X_{l+1}
相加
层归一化
前馈网络层
层归一化
相加
层归一化
注意力层
层归一化
X_l
Sandwich-Norm

归一化模块的位置	归一化模块的计算	优缺点
Post-Norm	Post-Norm(x) = Norm(x + Sublayer(x))	有助于加速收敛，理论性能更好，但是训练不稳定；
Pre-Norm	Pre-Norm(x) = x + Sublayer(Norm(x))	防止梯度消失或梯度爆炸，训练更加稳定，性能比post-norm模型差；
Sandwich-Norm	Sandwich-Norm(x) = x + Norm(Sublayer(Norm(x)))	防止出现数值爆炸，有时训练会不稳定；

目录

1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构

- 原始的 Transformer 使用 ReLU 激活函数可能产生神经元失效的问题

$$\text{ReLU}(x) = \max(x, 0)$$

- 针对 ReLU 存在的问题，Swish 和 GELU 被提出

$$\text{Swish}(x) = x \cdot \text{Sigmoid}(x)$$

$$\text{GELU}(x) = 0.5x \cdot [1 + \text{erf}(x/\sqrt{2})],\ \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_1^x e^{-t^2} dt$$

- 近来，大模型也常使用 GLU 和 GLU 的变种

$$\text{SwiGLU}(x) = \text{Swish}(W^G X) \odot (W^U x)$$

$$\text{GeGLU}(x) = \text{GELU}(W^G x) \odot (W^U x)$$

目录

1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构

- 绝对位置编码：
  - 三角函数编码

$$p_{t,j} = \begin{cases} \sin(t/10000^{(i-2)/H}) & i \bmod 2 = 0 \\ \cos(t/10000^{(i-1)/H}) & i \bmod 2 = 1 \end{cases}$$

  - 可学习的嵌入表示（早期 BERT 模型使用较多）
- 相对位置编码：
  T5 的相对位置编码，引入了可学习的标量

$$A_{ij} = x_i W^Q W^{K^T} x_j^T + r_{i-j}$$

- 旋转位置编码（RoPE）：基于绝对位置信息的旋转矩阵来表示注意力中的相对位置信息

$$\boldsymbol{R}_{\theta,t} = \begin{bmatrix} \cos t\theta_1 & -\sin t\theta_1 & 0 & 0 & \dots & 0 & 0 \\ \sin t\theta_1 & \cos t\theta_1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \cos t\theta_2 & -\sin t\theta_2 & \dots & 0 & 0 \\ 0 & 0 & \sin t\theta_2 & \cos t\theta_2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \cos t\theta_{H/2} & -\sin t\theta_{H/2} \\ 0 & 0 & 0 & 0 & \dots & \sin t\theta_{H/2} & \cos t\theta_{H/2} \end{bmatrix}$$

$$\Theta = \{\theta_i = b^{-2(i-1)/H} | i \in \{1, 2, \dots, H/2\}\}$$

$$\boldsymbol{q}_i = \boldsymbol{x}_i \boldsymbol{W}^Q \boldsymbol{R}_{\theta,i}, \ \boldsymbol{k}_j = \boldsymbol{x}_j \boldsymbol{W}^K \boldsymbol{R}_{\theta,j},$$

$$A_{ij} = (\boldsymbol{x}_i \boldsymbol{W}^Q \boldsymbol{R}_{\theta,i})(\boldsymbol{x}_j \boldsymbol{W}^K \boldsymbol{R}_{\theta,j})^\top = \boldsymbol{x}_i \boldsymbol{W}^Q \boldsymbol{R}_{\theta,i-j} \boldsymbol{W}^{K\top} \boldsymbol{x}_j^\top$$

```python
def rotate_half(x):
    x1 = x[..., : x.shape[-1] // 2]
    x2 = x[..., x.shape[-1] // 2 :]
    # 将向量每两个元素视为一个子空间
    return torch.cat((-x2, x1), dim=-1)

def apply_rotary_pos_emb(q, k, cos, sin, position_ids):
    cos = cos[position_ids].unsqueeze(1)
    sin = sin[position_ids].unsqueeze(1)
    # 获得各个子空间旋转的正余弦值
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    # 将每个子空间按照特定角度进行旋转
    return q_embed, k_embed
```

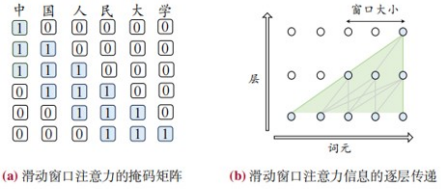
- ALiBi 位置编码位置编码：一种特殊的相对位置编码，主要用于增强 Transformer 模型的外推能力

$$A_{ij} = \boldsymbol{x}_i \boldsymbol{W}^{Q} \boldsymbol{W}^{K^{\mathsf{T}}} \boldsymbol{x}_j^{\mathsf{T}} - m(i-j),$$

```
def build_alibi_tensor(attention_mask: torch.Tensor, num_heads: int, dtype:
↪  torch.dtype) -> torch.Tensor:
    batch_size, seq_length = attention_mask.shape
    closest_power_of_2 = 2 ** math.floor(math.log2(num_heads))
    base = torch.tensor(
        2 ** (-(2 ** -(math.log2(closest_power_of_2) - 3))),
        ↪  device=attention_mask.device, dtype=torch.float32
    )
    powers = torch.arange(1, 1 + closest_power_of_2,
    ↪  device=attention_mask.device, dtype=torch.int32)
    slopes = torch.pow(base, powers)
    # 计算各个头的惩罚系数

    if closest_power_of_2 != num_heads:
        # 如果头数不是 2 的幂次方，修改惩罚系数
        extra_base = torch.tensor(
            2 ** (-(2 ** -(math.log2(2 * closest_power_of_2) - 3))),
            ↪  device=attention_mask.device, dtype=torch.float32
        )
        num_remaining_heads = min(closest_power_of_2, num_heads -
        ↪  closest_power_of_2)
        extra_powers = torch.arange(1, 1 + 2 * num_remaining_heads, 2,
        ↪  device=attention_mask.device, dtype=torch.int32)
        slopes = torch.cat([slopes, torch.pow(extra_base, extra_powers)],
        ↪  dim=0)

    arange_tensor = ((attention_mask.cumsum(dim=-1) - 1) *
    ↪  attention_mask)[:, None, :]
    # 计算相对距离
    alibi = slopes[..., None] * arange_tensor
    # 计算 ALiBi 施加的注意力偏置
    return alibi.reshape(batch_size * num_heads, 1, seq_length).to(dtype)
```

1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构

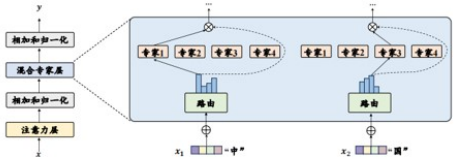
- 完整的自注意力机制
- 稀疏注意力机制：
  针对长序列计算时较大的计算和存储开销，减小注意力计算窗口，降低计算复杂度。
  代表工作：
  滑动窗口注意力
- 多查询/分组查询注意力：
  通过减少 KV 头的数量来减小访存量，实现推理加速，并且对模型性能影响较小。



(a) 滑动窗口注意力的掩码矩阵　(b) 滑动窗口注意力信息的逐层传递

1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构

拓展模型规模的同时不显著提升计算成本。




```python
class MoeLayer(nn.Module):
    def __init__(self, experts: List[nn.Module], gate: nn.Module,
        num_experts_per_tok: int):
        super().__init__()
        assert len(experts) > 0
        self.experts = nn.ModuleList(experts) # 所有专家的列表
        self.gate = gate # 路由网络
        self.num_experts_per_tok = num_experts_per_tok # 每个词元选择的专家数
        ↪ 目

    def forward(self, inputs: torch.Tensor):
        gate_logits = self.gate(inputs)
        weights, selected_experts = torch.topk(gate_logits,
                                               self.num_experts_per_tok)
        # 使用路由网络选择出 top-k 个专家
        weights = F.softmax(weights, dim=1,
        ↪  dtype=torch.float).to(inputs.dtype)
        # 计算出选择的专家的权重
        results = torch.zeros_like(inputs)
        for i, expert in enumerate(self.experts):
            batch_idx, nth_expert = torch.where(selected_experts == i)
            results[batch_idx] += weights[batch_idx, nth_expert, None] *
            ↪  expert(
                inputs[batch_idx]
            )
        # 将每个专家的输出加权相加作为最终的输出
        return results
```

1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构

```python
class LlamaModel(LlamaPreTrainedModel):
    def __init__(self, config: LlamaConfig):
        super().__init__(config)
        self.vocab_size = config.vocab_size
        # LLaMA 的词表大小
        self.embed_tokens = nn.Embedding(config.vocab_size,
        ↪  config.hidden_size, self.padding_idx)
        # LLaMA 的词嵌入矩阵，将输入的 id 序列转化为词向量序列
        self.layers = nn.ModuleList(
            [LlamaDecoderLayer(config, layer_idx) for layer_idx in
            ↪  range(config.num_hidden_layers)]
        )
        # 所有的 Transformer 解码器层
        self.norm = LlamaRMSNorm(config.hidden_size,
        ↪  eps=config.rms_norm_eps)
        causal_mask = torch.full(
            (config.max_position_embeddings,
            ↪  config.max_position_embeddings), fill_value=True,
            ↪  dtype=torch.bool
        )
```

```python
@add_start_docstrings_to_model_forward(Llama_INPUTS_DOCSTRING)
    def forward(
        self,
        input_ids: torch.LongTensor = None,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        **kwargs,

    ) -> Union[Tuple, BaseModelOutputWithPast]:
        if inputs_embeds is None:
            inputs_embeds = self.embed_tokens(input_ids)
            # 将输入的 input id 序列转化为词向量序列
        causal_mask = self._update_causal_mask(attention_mask,
        ↪  inputs_embeds)
        # 创建单向注意力的注意力掩盖矩阵

        hidden_states = inputs_embeds

        for decoder_layer in self.layers:
            hidden_states = decoder_layer(
                hidden_states,
                attention_mask=causal_mask,
                position_ids=position_ids,
            )[0]
            # 用每个 LLaMA 解码器层对词元的隐状态进行映射
        hidden_states = self.norm(hidden_states)
        # 对每个词元的隐状态使用 RMSNorm 归一化
        return BaseModelOutputWithPast(
            last_hidden_state=hidden_states,
        )
```

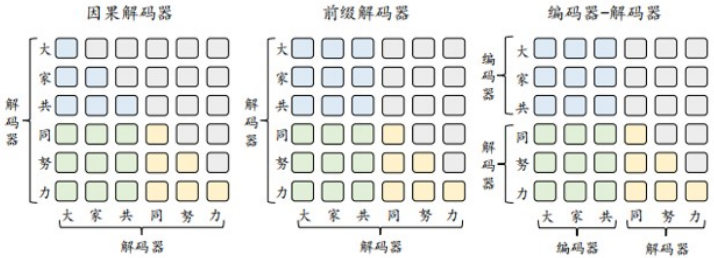
```python
class LlamaDecoderLayer(nn.Module):
    def __init__(self, config: LlamaConfig, layer_idx: int):
        super().__init__()

        self.hidden_size = config.hidden_size
        self.self_attn = LlamaAttention(config=config, layer_idx=layer_idx)
        ↪  # 注意力层
        self.mlp = LlamaMLP(config) # 前馈网络层

        self.input_layernorm = LlamaRMSNorm(config.hidden_size,
        ↪  eps=config.rms_norm_eps)
        self.post_attention_layernorm = LlamaRMSNorm(config.hidden_size,
        ↪  eps=config.rms_norm_eps)
        # 注意力层和前馈网络层前的 RMSNorm
```

```python
def forward(
    self,
    hidden_states: torch.Tensor,
    attention_mask: Optional[torch.Tensor] = None,
    position_ids: Optional[torch.LongTensor] = None,
    **kwargs,
) -> Tuple[torch.FloatTensor, Optional[Tuple[torch.FloatTensor,
↪  torch.FloatTensor]]]:

    residual = hidden_states

    hidden_states = self.input_layernorm(hidden_states)
    # 注意力层前使用 RMSNorm 进行归一化
    hidden_states, self_attn_weights, present_key_value =
    ↪  self.self_attn(
        hidden_states=hidden_states,
        attention_mask=attention_mask,
        position_ids=position_ids,
        **kwargs,
    )
    # 进行注意力模块的计算
    hidden_states = residual + hidden_states
    # 残差连接

    residual = hidden_states
    hidden_states = self.post_attention_layernorm(hidden_states)
    # 前馈网络层前使用 RMSNorm 进行归一化
    hidden_states = self.mlp(hidden_states)
    # 进行前馈网络层的计算
    hidden_states = residual + hidden_states
    # 残差连接
    outputs = (hidden_states,)
    return outputs
```

1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构

模型	类别	大小	归一化	位置编码	激活函数	L	N	H
GPT-3	因果	175B	Pre Layer	Learned	GELU	96	96	12288
PanGU-α	因果	207B	Pre Layer	Learned	GELU	64	128	16384
OPT	因果	175B	Pre Layer	Learned	ReLU	96	96	12288
PaLM	因果	540B	Pre Layer	RoPE	SwiGLU	118	48	18432
BLOOM	因果	176B	Pre Layer	ALiBi	GELU	70	112	14336
MT-NLG	因果	530B	-	-	-	105	128	20480
Gopher	因果	280B	Pre RMS	Relative	-	80	128	16384
Chinchilla	因果	70B	Pre RMS	Relative	-	80	64	8192
Galactica	因果	120B	Pre Layer	Learned	GELU	96	80	10240
LaMDA	因果	137B	-	Relative	GeGLU	64	128	8192
Jurassic-1	因果	178B	Pre Layer	Learned	GELU	76	96	13824
LLaMA-2	因果	70B	Pre RMS	RoPE	SwiGLU	80	64	8192
Pythia	因果	12B	Pre Layer	RoPE	GELU	36	40	5120
Baichuan-2	因果	13B	Pre RMS	ALiBi	SwiGLU	40	40	5120
Qwen-1.5	因果	72B	Pre RMS	RoPE	SwiGLU	80	64	8192
InternLM-2	因果	20B	Pre RMS	RoPE	SwiGLU	48	48	6144
Falcon	因果	180B	Pre Layer	RoPE	GELU	80	232	14848
MPT	因果	30B	Pre Layer	ALiBi	GELU	48	64	7168
Mistral	因果	7B	Pre RMS	RoPE	SwiGLU	32	32	4096
Gemma	因果	7B	Pre RMS	RoPE	GELU	28	16	3072
DeepSeek	因果	67B	Pre RMS	RoPE	SwiGLU	95	64	8192
Yi	因果	34B	Pre RMS	RoPE	SwiGLU	60	56	7168
YuLan	因果	12B	Pre RMS	RoPE	SwiGLU	40	38	4864
GLM-130B	前缀	130B	Post Deep	RoPE	GeGLU	70	96	12288
T5	编-解	11B	Pre RMS	Relative	ReLU	24	128	1024

- encoder-架构（BERT）
- encoder-decoder 架构（T5）
- decoder-架构
  - 前缀解码器（GLM 系列模型）
  - 因果解码器（GPT 系列，LLaMA 系列.....）



1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构

在长文档分析，多轮对话以及故事创作等背景下模型处理的文本长度会超过预定义的上下文窗口大小。如何有效拓展模型的上下文窗口是当前的研究焦点。

当前的解决方案：

- 拓展位置编码
- 调整上下文窗口

某些特定的位置编码在超出原始上下文窗口的文本上，也能够表现出较好的建模能力，这种能力通常被称为**外推（Extrapolation）**能力。

当前 T5 偏置和 ALiBI 等方法展示了不同程度的外推能力，可以确保在长文本上生成流畅文本，但并不能保证和短文本一样的理解能力，通常需要在长文本上继续训练。

对当前主流的 RoPE 在未修改的情况下不具备良好的外推能力，很多工作在 RoPE 上进行了改进提升模型的长文本能力。

对 RoPE 改进方法的形式化定义：

将上下文窗口从 T_{max} 提升到 T'_{max}

对于子空间 i 上的相对位置 t，旋转角度为 $f(t,i) = t \cdot \theta_i$
定义新的旋转角度为：$f(t,i) = g(t) \cdot h(i)$

- 方法一：直接微调：
  不对 RoPE 做修改，直接在更长文本上进行训练，收敛缓慢，需要大量数据进行继续预训练。
- 方法二：对位置索引进行修改：
  对旋转角度进行限制，确保拓展后的上下文窗口的旋转角度得到充分且有效的训练

位置内插：$g(t) = \frac{T_{\max}}{T'_{\max}} \cdot t$

位置裁剪：$g(t) = \begin{cases} t & t <= w, \\ w & t > w \text{且使用 ReRoPE}, \\ w + \frac{(T_{\max}-w)(t-w)}{T_{\max}-w} & t > w \text{且使用 LeakyReRoPE} \end{cases}$

- 方法三：对基进行修改
  - 针对基进行缩放，保证旋转角度在训练阶段出现过

$$f(T'_{max}, i) = T'_{max} \cdot h(i) \leq T_{max} \cdot \theta_i$$

对角度公式的底数进行调整：

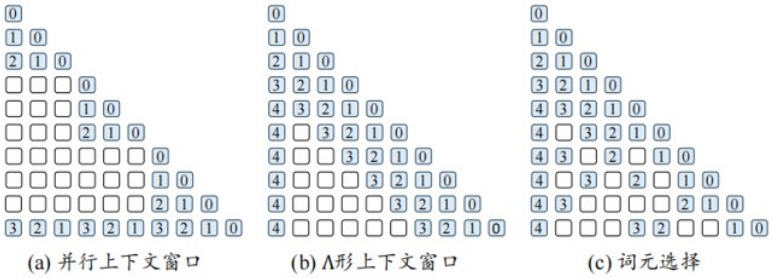
$$h(i) = (\alpha \cdot b)^{-(i-1)/H}$$

NTK-RoPE:$(T'_{max}/T_{max})^{H/(H-2)}$

Dynamic-NTK-RoPE 根据输入文本长度动态设置：$\alpha = \max(1, T/T_{max})$

  - 基截断

$$h(i) = \begin{cases} \theta_i & \theta_i \geq c \quad \text{防止出现超过预期分布的旋转角度} \\ \beta & c \geq \theta_i \geq a \quad \text{一定程度上削弱了某些子空间对不同位置索引的区分能力} \\ 0 & \theta_i \leq a \end{cases}$$



- 并行上下文窗口：采用了一种分而治之的策略来处理输入文本，该方法无法有效地区分不同段落之间的顺序。
- 基于对大模型倾向于对序列起始位置以及邻近的词元赋予更高的注意力权重的观察得到，缺陷是无法利用所有的上下文信息。
- 通过计算查询与词元或者分块的相似度来决定选取哪些词元进行计算。

为了有效拓展模型的长文本建模能力，通常需要使用特殊准备的数据对于模型进行**继续预训练**。

长文本数据的数据量：只需在约 1B 级别的词元上执行数百步的训练，就可以将 7B 或者 13B 大小的 LLaMA 系列模型的上下文窗口至 100K 词元以上的长度，并具有较好的长上下文利用能力。

长文本数据混合：长文本数据的领域应尽可能多样化，并且与预训练数据集的分布保持相似。同时需要整体型（完整的有意义的长文）、聚合型（多篇相关文本的聚合）的数据并去除杂乱型（杂乱无章的文本）。

1. Transformer 模型
   - 模型基本架构
2. 详细配置
   - 归一化方法
   - 激活函数
   - 位置编码
   - 注意力机制
   - 混合专家模型（MoE）
   - LLaMA 详细配置
3. 主流结构
   - 主流架构
4. 长上下文模型
   - 长上下文模型
5. 新型模型架构
   - 新型模型架构

基于 Transformer 的模型的优缺点：有很好的数据表示以及建模能力，但是处理长序列时计算和存储的复杂度会随输入序列长度的**平方级别**增长。

针对长文本建模效率的问题，一些基于参数化 (离散型) 状态空间模型（SSM）设计的新型模型被提出。

参数化状态空间模型可以看作 RNN 和 CNN 的结合体：

- 利用卷积计算对输入进行并行化编码
- 利用额外的状态变量存储序列信息，在进行预测时仅需要当前时刻的词元
- 提高了模型的训练效率和推理效率

模型的数学表示：

$$\boldsymbol{S}_t = \boldsymbol{A} \otimes \boldsymbol{S}_{t-1} + \boldsymbol{B} \otimes \boldsymbol{x}_t,$$

$$\boldsymbol{y}_t = \boldsymbol{C} \otimes \boldsymbol{S}_t,$$

$\boldsymbol{A} \in \mathbb{R}^{H \times N \times N}$、$\boldsymbol{B} \in \mathbb{R}^{H \times N \times 1}$ 和 $\boldsymbol{C} \in \mathbb{R}^{H \times 1 \times N}$ 是可训练参数

H表示输入表示的维度
N表示状态空间模型压缩后的维度

根据状态空间模型的数学表示：

$$S_t = A \otimes S_{t-1} + B \otimes x_t$$

$$y_t = C \otimes S_t$$

进一步循环计算：

$$y_t = C \otimes S_t = C \otimes A \otimes (A \otimes S_{t-2} + B \otimes x_{t-1}) + C \otimes B \otimes x_t$$

$$= C \otimes A^{t-1} \otimes Bx_1 + ... + C \otimes B \otimes x_t = \sum_{i=1}^{t} C \otimes A^{t-i} \otimes Bx_i$$

也可以表示为卷积计算（可以利用快速傅里叶变换加快计算效率）：

$$K = (C \otimes B, C \otimes A \otimes B, ..., C \otimes A^{t-1} \otimes B, ...)$$

$$y = x * K$$

尽管状态空间模型计算效率较高，但是在**文本任务**上的表现相比 Transformer 模型**仍有一定的差距**。

- Mamba：在状态空间模型的基础上引入**信息选择**机制，**将更新状态和输出的方程的参数矩阵（A，B，C）表示成输入的非线性函数**，基于当前时刻的输入对上一时刻的状态和当前时刻的输入做选择性过滤，实现更为有效的上下文表示，相比于状态空间模型，**有更好的文本建模性能**，但是由于引入非线性关系，**无法使用快速傅里叶变换，训练计算慢**。
- RWKV：尝试将 RNN 和 Transformer 的优点进行结合，继承 Transformer 的建模优势和 RNN 的计算效率，**训练中缺少并行计算的能力**。
  主要改进：
  - 在每层的计算中使用词元偏移代替词元表示。（词元偏移是当前时刻和上一时刻的词元表示的线性插值）**时间混合模块**和**频道混合模块**分别代替多头自注意力模块和前馈网络模块。RNN 的网络，并使用词元偏移对状态进行更新。。

- RetNet：使用**多尺度保留机制**提升序列建模能力。在标准状态空间模型基础上，计算状态变量时，类似于多头注意力机制，将输入词元映射为 q,k,v，通过 k，v 和前一时刻的状态变量 S 得到当前状态变量 S‘，使用 q 和当前时刻的 S‘计算输出。（**保留了循环计算和并行计算的优点**）
- Hyena：在 Transformer 中利用**长卷积模块代替注意力计算**，从而借助卷积的快速傅里叶变换来提高计算效率。即每一层的长卷积模块中，即每个相对位置有一个相应的滤波器，利用该卷积核和输入的序列做卷积得到每个位置的中间表示，并使用门控函数（基于当前输入词元）对中间表示加权得到最终的输出。

模型	可并行性	解码复杂度	训练复杂度
Transformer	✓	$O(TH+H^2)$	$O(T^2H+TH^2)$
标准 SSM	✓	$O(N^2H+H^2)$	$O(TH\log T+THN^2+TH^2)$
Mamba	×	$O(N^2H+H^2)$	$O(TN^2H+TH^2)$
RWKV	×	$O(H^2)$	$O(TH^2)$
RetNet	✓	$O(H^2)$	$O(TH^2)$
Hyena	✓	$O(TMH+MH^2)$	$O(TMH\log T+TMH^2)$

T 表示序列长度，H 表示输入表示的维度，N 表示状态空间模型压缩后的维度，M 表示 Hyena 每个模块的层数。