

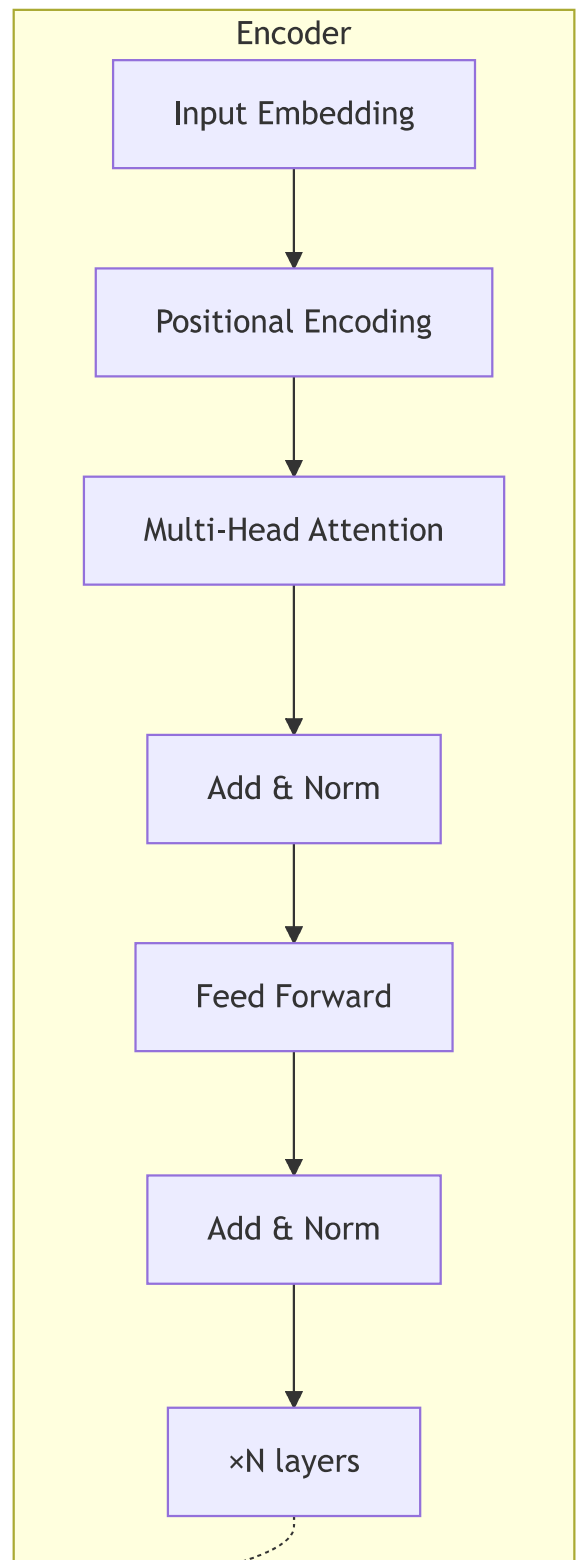
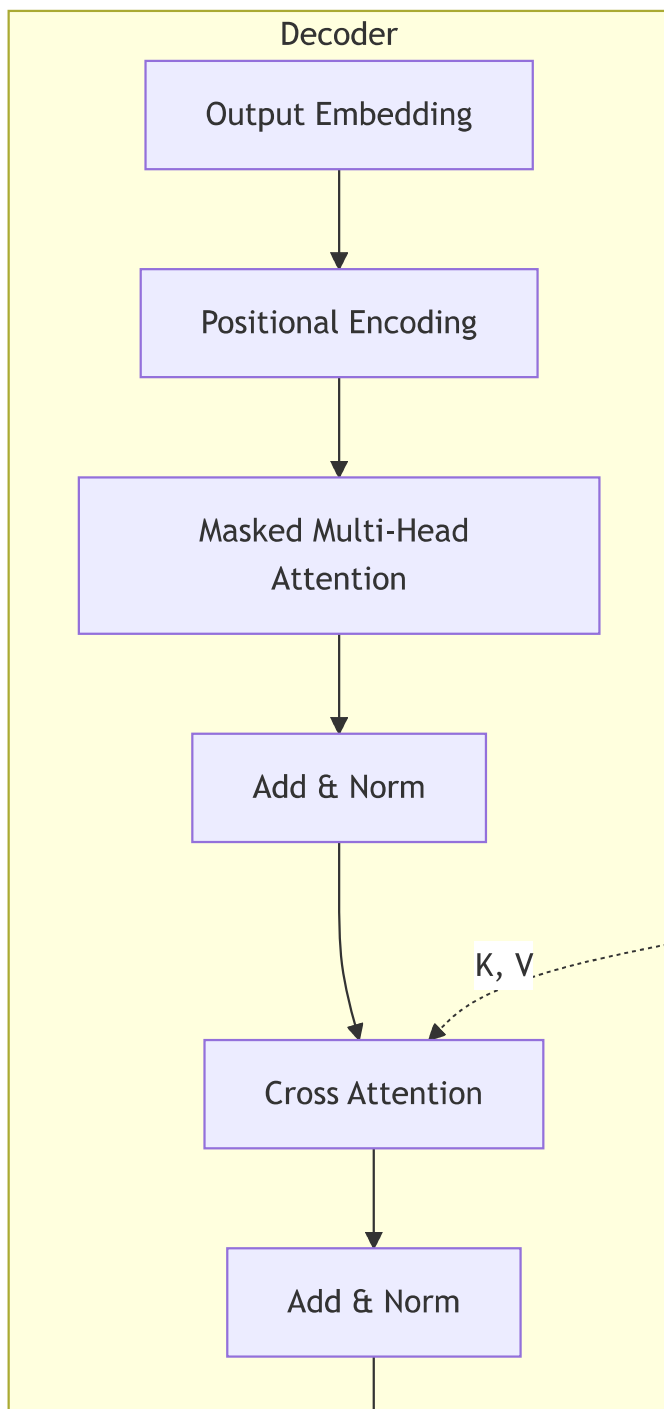
第2章 Transformer架构

■ "Attention is All You Need" —— 这篇论文开启了大模型时代

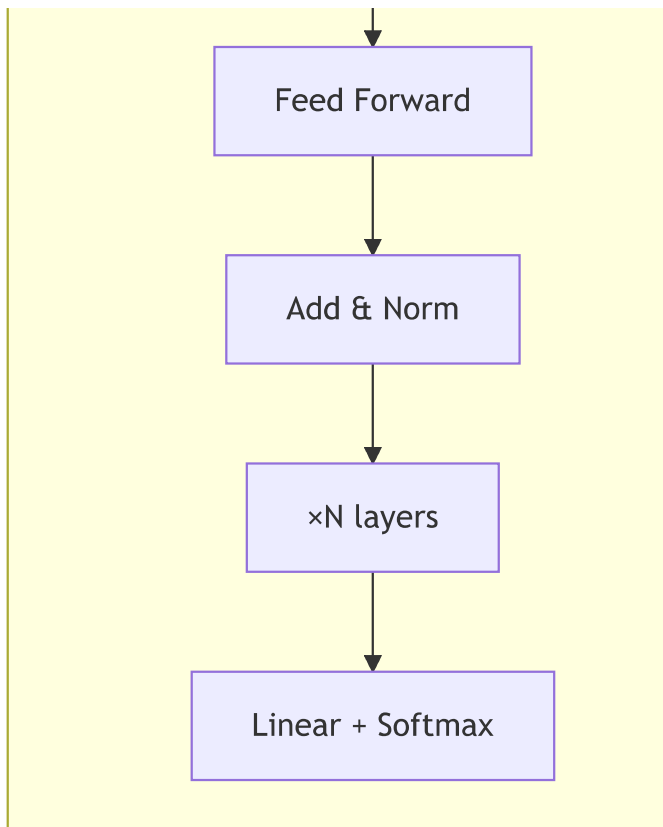
2.1 Transformer整体架构

2.1.1 经典Transformer结构

原始Transformer采用Encoder-Decoder架构，用于序列到序列任务（如机器翻译）。



K, V



2.1.2 三大主流架构对比

| 架构类型 | 代表模型 | 特点 | 适用场景 |
|------------------------|-------------------|------------|--------------------|
| Encoder-only | BERT, RoBERTa | 双向注意力，擅长理解 | 分类、信息抽取、embedding |
| Decoder-only | GPT系列, LLaMA, GLM | 单向注意力，擅长生成 | 文本生成、对话（主流） |
| Encoder-Decoder | T5, BART | 编码+解码，灵活 | 翻译、摘要、结构化生成 |

现代大模型趋势：

- ☒ GPT-3/4: Decoder-only
- ☒ LLaMA系列: Decoder-only
- ☒ Claude: Decoder-only
- ☒ ChatGLM: Decoder-only (UniLM结构)

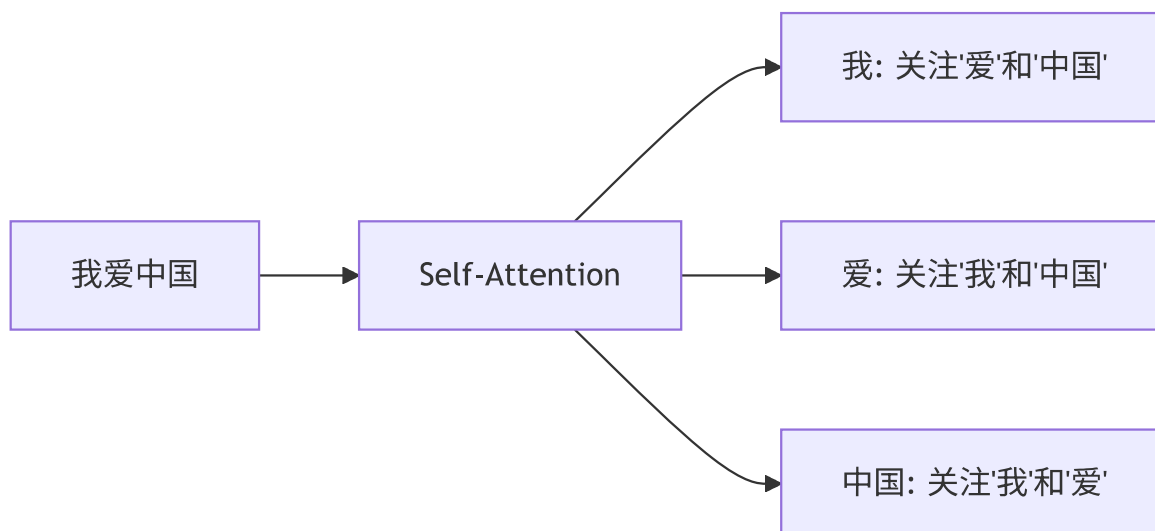
为什么Decoder-only成为主流？

1. **统一范式**：所有任务都可以表达为文本生成
2. **扩展性好**：架构简单，易于扩展到超大规模
3. **零样本能力**：预训练和下游任务形式一致

2.2 Self-Attention机制

Self-Attention是Transformer的核心，让模型能够"关注"输入序列的不同位置。

2.2.1 Attention核心思想



2.2.2 Scaled Dot-Product Attention

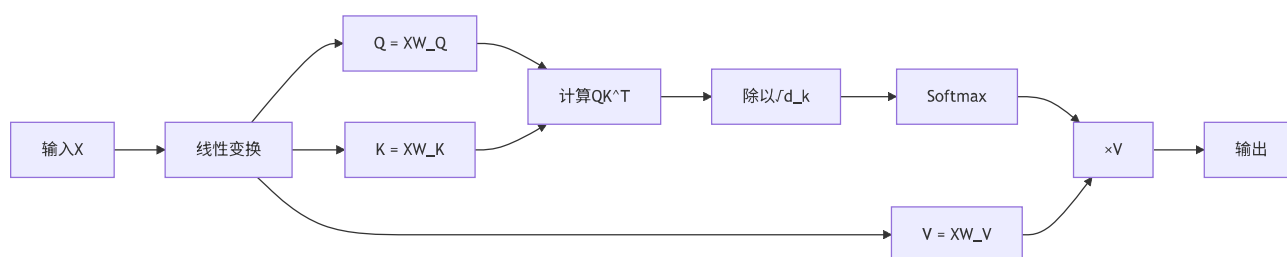
数学公式:

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

其中:

- Q (Query) : 查询矩阵, "我想找什么"
- K (Key) : 键矩阵, "我是什么"
- V (Value) : 值矩阵, "我的内容是什么"
- d_k : 键向量的维度

计算流程:



代码实现:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class ScaledDotProductAttention(nn.Module):
    def __init__(self, dropout=0.1):
        super().__init__()
```

```

self.dropout = nn.Dropout(dropout)

def forward(self, Q, K, V, mask=None):
    """
    Args:
        Q: (batch_size, num_heads, seq_len, d_k)
        K: (batch_size, num_heads, seq_len, d_k)
        V: (batch_size, num_heads, seq_len, d_v)
        mask: (batch_size, 1, seq_len, seq_len) or (batch_size, 1, 1, seq_len)

    Returns:
        output: (batch_size, num_heads, seq_len, d_v)
        attention_weights: (batch_size, num_heads, seq_len, seq_len)
    """
    d_k = Q.size(-1)

    # 1. 计算注意力分数:  $Q @ K^T / \sqrt{d_k}$ 
    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)
    # scores shape: (batch_size, num_heads, seq_len_q, seq_len_k)

    # 2. 应用mask (可选)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    # 3. Softmax归一化
    attention_weights = F.softmax(scores, dim=-1)
    attention_weights = self.dropout(attention_weights)

    # 4. 加权求和
    output = torch.matmul(attention_weights, V)

    return output, attention_weights

```

为什么要除以 $\sqrt{d_k}$?

```

# 数值稳定性实验
import numpy as np

d_k = 64
Q = np.random.randn(10, d_k)
K = np.random.randn(10, d_k)

# 不缩放
scores_no_scale = np.dot(Q, K.T)
print(f"不缩放的方差: {np.var(scores_no_scale):.2f}")
# 输出: 不缩放的方差: 64.23

```

```
# 缩放
scores_scaled = np.dot(Q, K.T) / np.sqrt(d_k)
print(f"缩放后的方差: {np.var(scores_scaled):.2f}")
# 输出: 缩放后的方差: 1.00
```

原因:

1. **数值稳定**: 防止点积过大, 导致softmax梯度消失
2. **方差控制**: 保持方差为1, 训练更稳定
3. **梯度友好**: softmax在合理区间内, 梯度更好

2.2.3 Attention可视化示例

假设输入句子: "The cat sat on the mat"

Attention权重矩阵 (示例):

| | The | cat | sat | on | the | mat |
|-----|------|-----|-----|-----|-----|------|
| The | [0.1 | 0.2 | 0.1 | 0.1 | 0.2 | 0.3] |
| cat | [0.2 | 0.4 | 0.1 | 0.1 | 0.1 | 0.1] |
| sat | [0.1 | 0.3 | 0.2 | 0.2 | 0.1 | 0.1] |
| on | [0.1 | 0.1 | 0.2 | 0.1 | 0.3 | 0.2] |
| the | [0.1 | 0.1 | 0.1 | 0.2 | 0.2 | 0.3] |
| mat | [0.2 | 0.1 | 0.1 | 0.1 | 0.2 | 0.3] |

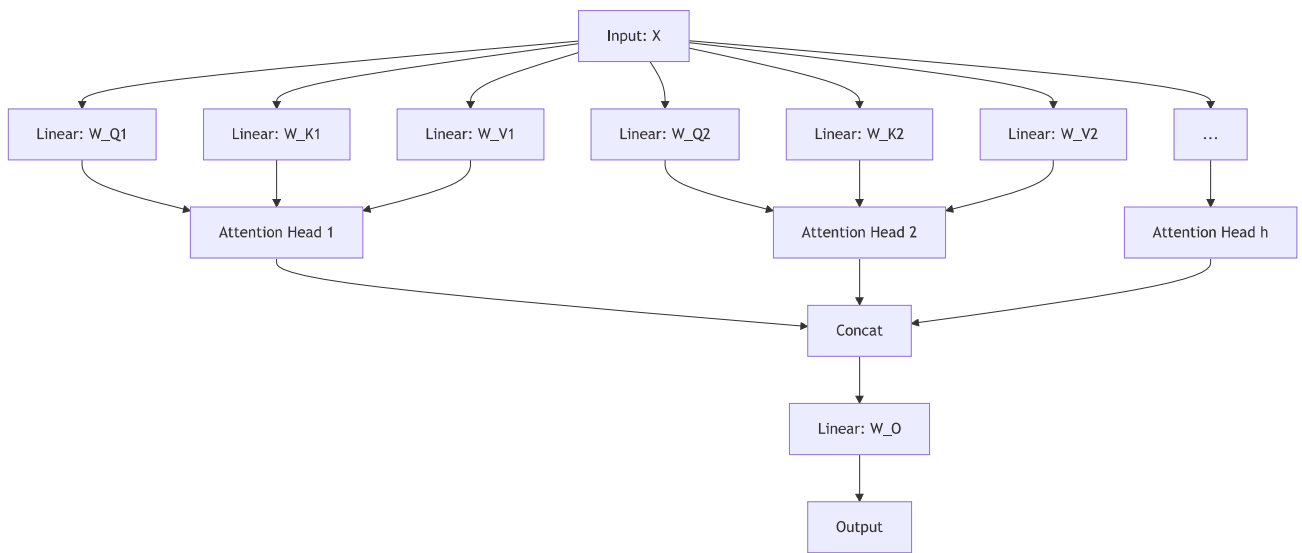
解读:

- "cat"高度关注自己 (0.4) 和"The" (0.2)
- "mat"关注"the" (0.3) 和自己 (0.3)

2.3 Multi-Head Attention

单个attention只能学习一种关注模式, Multi-Head让模型学习多种不同的关注模式。

2.3.1 Multi-Head结构



数学表达:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

代码实现:

```

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.1):
        """
        Args:
            d_model: 模型维度 (如512)
            num_heads: 注意力头数 (如8)
        """
        super().__init__()
        assert d_model % num_heads == 0, "d_model必须能被num_heads整除"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads # 每个head的维度

        # 定义Q、K、V的线性变换
        self.W_Q = nn.Linear(d_model, d_model)
        self.W_K = nn.Linear(d_model, d_model)
        self.W_V = nn.Linear(d_model, d_model)

        # 输出线性变换
        self.W_O = nn.Linear(d_model, d_model)

        self.attention = ScaledDotProductAttention(dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, Q, K, V, mask=None):

```

```

"""
Args:
    Q, K, V: (batch_size, seq_len, d_model)
    mask: (batch_size, seq_len, seq_len)
"""
batch_size = Q.size(0)

# 1. 线性变换
Q = self.W_Q(Q) # (batch_size, seq_len, d_model)
K = self.W_K(K)
V = self.W_V(V)

# 2. 拆分成多个头
# (batch_size, seq_len, d_model) -> (batch_size, seq_len, num_heads, d_k)
# -> (batch_size, num_heads, seq_len, d_k)
Q = Q.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
K = K.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
V = V.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)

# 3. 应用attention
if mask is not None:
    mask = mask.unsqueeze(1) # 广播到所有头

x, attention_weights = self.attention(Q, K, V, mask)
# x shape: (batch_size, num_heads, seq_len, d_k)

# 4. 合并多个头
x = x.transpose(1, 2).contiguous() # (batch_size, seq_len, num_heads, d_k)
x = x.view(batch_size, -1, self.d_model) # (batch_size, seq_len, d_model)

# 5. 最后的线性变换
output = self.W_O(x)
output = self.dropout(output)

return output, attention_weights

```

2.3.2 为什么使用Multi-Head?

单头vs多头的类比:

单头注意力 = 一个专家看所有问题
 多头注意力 = 多个专家各看不同角度

例如翻译"bank":

- Head 1: 关注金融语境 → "银行"
- Head 2: 关注地理语境 → "河岸"

- Head 3: 关注句法依赖
- Head 4: 关注语义相似度

实际效果：

| 头数 | 参数量 | 性能 | 计算量 |
|----|----------|-------------|-----|
| 1 | 基准 | 较差 | 最低 |
| 8 | 相同（维度减小） | 最佳平衡 | 适中 |
| 16 | 相同 | 略有提升 | 较高 |
| 32 | 相同 | 提升有限 | 很高 |

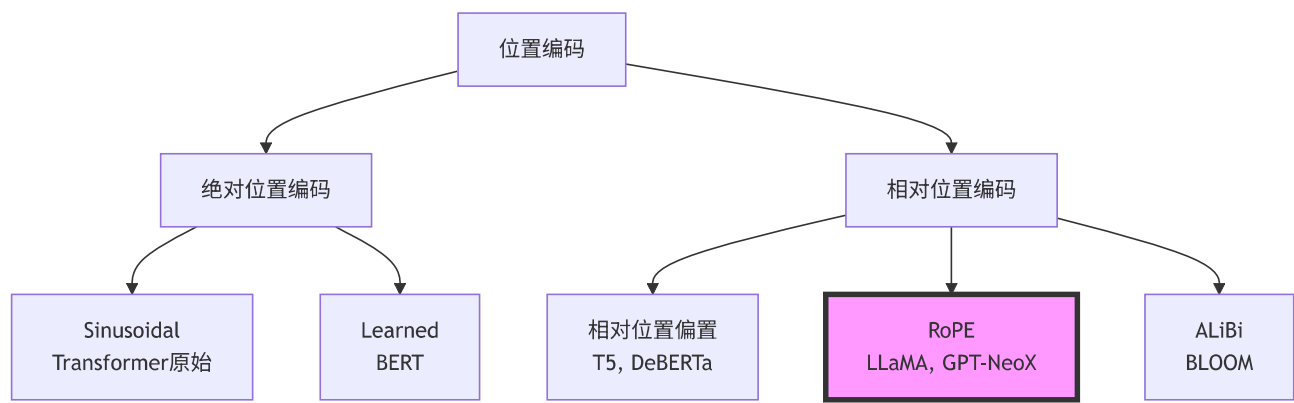
主流大模型配置：

- GPT-3: 96头 (d_model=12288, d_k=128)
- LLaMA-7B: 32头 (d_model=4096, d_k=128)
- BERT-base: 12头 (d_model=768, d_k=64)

2.4 Position Encoding

Transformer本身没有位置信息，需要通过Position Encoding引入。

2.4.1 位置编码类型对比



2.4.2 Sinusoidal Position Encoding

公式：

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

实现：

```

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()

```

```

# 创建位置编码矩阵
pe = torch.zeros(max_len, d_model)
position = torch.arange(0, max_len, 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)

pe = pe.unsqueeze(0) # (1, max_len, d_model)
self.register_buffer('pe', pe)

def forward(self, x):
    """
    Args:
        x: (batch_size, seq_len, d_model)
    """
    seq_len = x.size(1)
    x = x + self.pe[:, :seq_len, :]
    return x

```

特点:





-  不需要训练
-  理论上可以外推到任意长度
-  实际外推效果有限

2.4.3 RoPE (Rotary Position Embedding)

RoPE是现代大模型的主流选择，被LLaMA、GPT-NeoX等采用。

核心思想： 通过旋转矩阵在复数空间编码相对位置信息。

优势:

1.  相对位置编码（更合理）
2.  长度外推性好
3.  实现高效
4.  性能优秀

简化实现:

```

class RotaryPositionalEmbedding(nn.Module):
    def __init__(self, dim, max_seq_len=2048, base=10000):
        super().__init__()
        inv_freq = 1.0 / (base ** (torch.arange(0, dim, 2).float() / dim))

```

```

self.register_buffer('inv_freq', inv_freq)
self.max_seq_len = max_seq_len

def forward(self, x, seq_len):
    """
    Args:
        x: (batch, heads, seq_len, dim)
    """
    # 生成位置序列
    t = torch.arange(seq_len, device=x.device).type_as(self.inv_freq)

    # 计算频率
    freqs = torch.einsum('i,j->ij', t, self.inv_freq)
    emb = torch.cat((freqs, freqs), dim=-1)

    # 应用旋转
    return self.apply_rotary_emb(x, emb)

def apply_rotary_emb(self, x, emb):
    cos = emb.cos()
    sin = emb.sin()

    # 旋转操作（简化版）
    x1, x2 = x[..., ::2], x[..., 1::2]
    return torch.cat([
        x1 * cos - x2 * sin,
        x1 * sin + x2 * cos
    ], dim=-1)

```

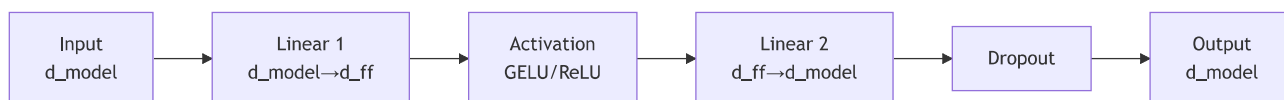
2.4.4 位置编码对比

| 类型 | 代表模型 | 外推能力 | 性能 | 复杂度 |
|------------|--------------|-------|------|-----|
| Sinusoidal | Transformer | ☆☆ | ☆☆☆ | 低 |
| Learned | BERT | ☆ | ☆☆☆ | 低 |
| RoPE | LLaMA | ☆☆☆☆ | ☆☆☆☆ | 中 |
| ALiBi | BLOOM | ☆☆☆☆☆ | ☆☆☆ | 低 |

2.5 Feed-Forward Network

每个Transformer层都包含一个Position-wise Feed-Forward Network。

2.5.1 FFN结构



数学表达:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

或使用GELU:

$$\text{FFN}(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2$$

代码实现:

```

class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        """
        Args:
            d_model: 输入输出维度 (如512)
            d_ff: 中间层维度 (通常是d_model的4倍, 如2048)
        """
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
        self.activation = nn.GELU() # 大模型常用GELU

    def forward(self, x):
        """
        Args:
            x: (batch_size, seq_len, d_model)
        """
        x = self.linear1(x)
        x = self.activation(x)
        x = self.dropout(x)
        x = self.linear2(x)
        x = self.dropout(x)
        return x
  
```

2.5.2 FFN的作用

为什么需要FFN?

1. **增加非线性**: Attention是线性操作 (加权求和), FFN引入非线性
2. **特征转换**: 在更高维空间进行特征转换
3. **参数主体**: FFN占Transformer参数量的2/3

参数量分析:

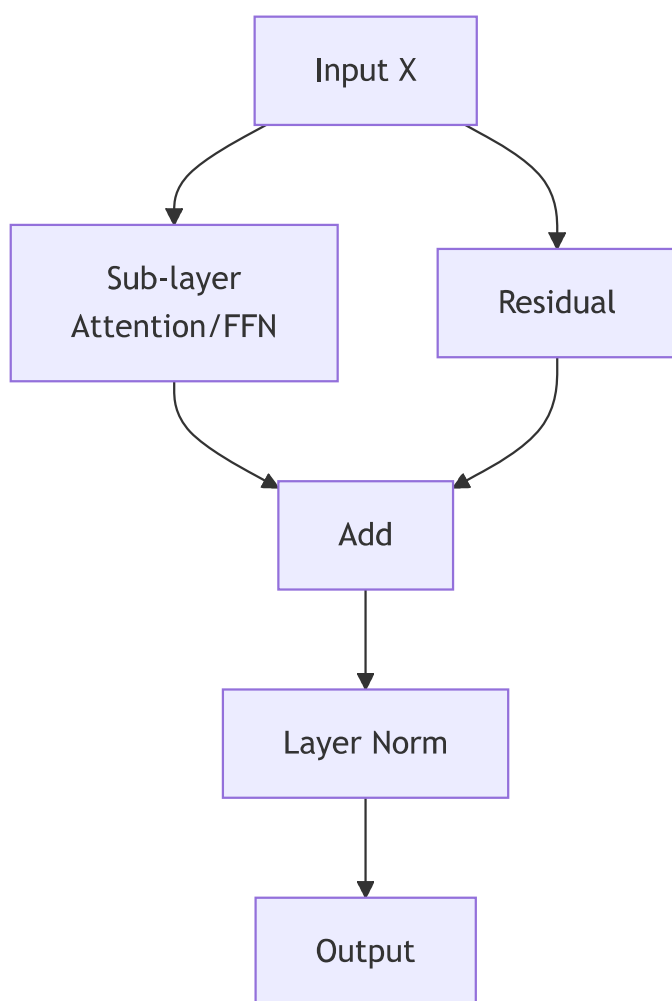
假设 $d_{\text{model}}=512$, $d_{\text{ff}}=2048$, $\text{num_heads}=8$

| 组件 | 参数量计算 | 参数量 |
|----------------------|-------------------------------------|-------|
| Multi-Head Attention | $4 \times 512 \times 512$ | 1.05M |
| Feed-Forward | $512 \times 2048 + 2048 \times 512$ | 2.10M |
| 总计 | | 3.15M |

FFN占比: $2.10\text{M} / 3.15\text{M} \approx 67\%$

2.6 残差连接与层归一化

2.6.1 Add & Norm



两种主流顺序:

```
# Post-LN (原始Transformer)
class TransformerBlockPostLN(nn.Module):
    def forward(self, x):
        # Attention
        x = self.norm1(x + self.attention(x))
        # FFN
```

```

        x = self.norm2(x + self.ffn(x))
    return x

# Pre-LN (现代大模型主流)
class TransformerBlockPreLN(nn.Module):
    def forward(self, x):
        # Attention
        x = x + self.attention(self.norm1(x))
        # FFN
        x = x + self.ffn(self.norm2(x))
    return x
```

Pre-LN vs Post-LN对比:

| 特性 | Post-LN | Pre-LN |
|-------|-------------|----------------------|
| 训练稳定性 | 较差，需要warmup | 更稳定 |
| 梯度流动 | 可能梯度爆炸 | 更平滑 |
| 性能 | 理论上稍好 | 实际差异小 |
| 大模型使用 | GPT-2 | GPT-3, LLaMA等 |

为什么Pre-LN更稳定?

Pre-LN将归一化放在残差分支内，保证主路径的梯度流畅通。

2.7 Masked Attention

在语言模型中，需要防止“偷看未来”，使用Causal Mask。

2.7.1 Causal Mask可视化

输入序列: ["I", "love", "AI"]

注意力矩阵 (Causal Mask):

| | | | |
|------|----|------|-----|
| | I | love | AI |
| I | [✓ | X | X] |
| love | [✓ | ✓ | X] |
| AI | [✓ | ✓ | ✓] |

✓ 可以看到

X 被mask掉 (设为-inf)

实现:

```

def create_causal_mask(seq_len):
    """
    创建因果mask矩阵
    返回下三角矩阵（包含对角线）
    """
    mask = torch.tril(torch.ones(seq_len, seq_len))
    # mask shape: (seq_len, seq_len)
    # [[1, 0, 0],
    #  [1, 1, 0],
    #  [1, 1, 1]]
    return mask

def apply_causal_mask(scores, mask):
    """
    应用mask到attention scores
    """
    scores = scores.masked_fill(mask == 0, -1e9)
    return scores

```

完整的Masked Multi-Head Attention:

```

class MaskedMultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.1):
        super().__init__()
        self.attention = MultiHeadAttention(d_model, num_heads, dropout)

    def forward(self, x):
        """
        Args:
            x: (batch_size, seq_len, d_model)
        """
        batch_size, seq_len, d_model = x.size()

        # 创建因果mask
        causal_mask = create_causal_mask(seq_len).to(x.device)
        causal_mask = causal_mask.unsqueeze(0).expand(batch_size, -1, -1)

        # Self-attention with mask
        output, attn_weights = self.attention(x, x, x, mask=causal_mask)

        return output, attn_weights

```

2.8 完整Transformer Block实现

```

class TransformerBlock(nn.Module):
    """
    一个完整的Transformer decoder层 (Pre-LN)
    """
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()

        # Multi-Head Attention
        self.attention = MaskedMultiHeadAttention(d_model, num_heads, dropout)

        # Feed-Forward Network
        self.ffn = FeedForward(d_model, d_ff, dropout)

        # Layer Normalization
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

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

    def forward(self, x):
        """
        Args:
            x: (batch_size, seq_len, d_model)
        Returns:
            output: (batch_size, seq_len, d_model)
        """
        # 1. Masked Multi-Head Attention + Residual + Norm
        attn_output, _ = self.attention(self.norm1(x))
        x = x + self.dropout(attn_output)

        # 2. Feed-Forward + Residual + Norm
        ffn_output = self.ffn(self.norm2(x))
        x = x + self.dropout(ffn_output)

        return x

```

```

class GPTModel(nn.Module):
    """
    简化的GPT模型
    """
    def __init__(self, vocab_size, d_model=768, num_heads=12,
                  num_layers=12, d_ff=3072, max_len=1024, dropout=0.1):
        super().__init__()

```



```

# Token Embedding
self.token_embedding = nn.Embedding(vocab_size, d_model)

# Position Encoding
self.pos_encoding = PositionalEncoding(d_model, max_len)

# Transformer Blocks
self.blocks = nn.ModuleList([
    TransformerBlock(d_model, num_heads, d_ff, dropout)
    for _ in range(num_layers)
])

# Final Layer Norm
self.ln_f = nn.LayerNorm(d_model)

# Output Head
self.lm_head = nn.Linear(d_model, vocab_size, bias=False)

# 权重共享 (Embedding和输出层共享权重)
self.lm_head.weight = self.token_embedding.weight

self.dropout = nn.Dropout(dropout)

def forward(self, input_ids):
    """
    Args:
        input_ids: (batch_size, seq_len)
    Returns:
        logits: (batch_size, seq_len, vocab_size)
    """
    # Embedding
    x = self.token_embedding(input_ids) # (B, L, D)
    x = self.pos_encoding(x)
    x = self.dropout(x)

    # Transformer Blocks
    for block in self.blocks:
        x = block(x)

    # Final Norm
    x = self.ln_f(x)

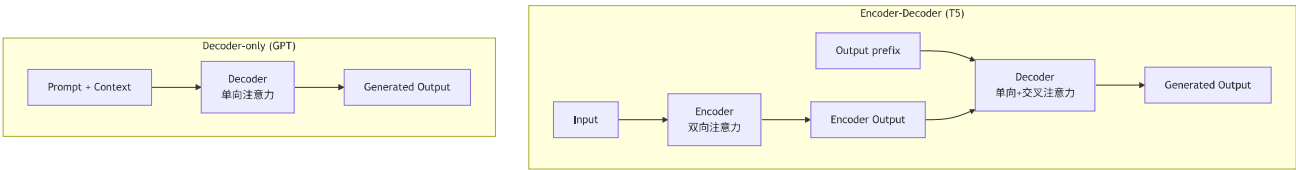
    # Output Projection
    logits = self.lm_head(x) # (B, L, V)

    return logits

```

2.9 Encoder-Decoder vs Decoder-only

2.9.1 架构对比



2.9.2 详细对比表

| 维度 | Encoder-Decoder | Decoder-only |
|-------|-----------------|----------------|
| 架构复杂度 | 复杂（两个模块） | 简单（单模块） |
| 参数效率 | 需要更多参数 | 相同性能参数更少 |
| 训练目标 | 不同模块不同目标 | 统一的自回归 |
| 推理效率 | 需要编码阶段 | 端到端生成 |
| 适用任务 | 结构化生成（翻译、摘要） | 通用生成 |
| 扩展性 | 复杂 | 易扩展到百亿参数 |
| 零样本能力 | 有限 | 强大 |
| 代表模型 | T5, BART | GPT-3/4, LLaMA |

2.10 面试高频问题

Q1: 手写Self-Attention的核心代码

```
def self_attention(Q, K, V, mask=None):
    """
    最简版本的Self-Attention
    """
    d_k = Q.shape[-1]

    # 计算注意力分数
    scores = np.dot(Q, K.T) / np.sqrt(d_k)

    # 应用mask
    if mask is not None:
        scores = np.where(mask == 0, -1e9, scores)

    # Softmax
    attention_weights = np.exp(scores) / np.sum(np.exp(scores), axis=-1, keepdims=1)
```

```
# 加权求和
output = np.dot(attention_weights, V)

return output, attention_weights
```

Q2: 为什么Transformer可以并行训练?

答案要点:

1. **RNN的问题**: $h_t = f(h_{t-1}, x_t)$, 必须顺序计算
2. **Transformer的优势**:
 - 所有位置的attention可以并行计算
 - 训练时, 整个序列的mask是固定的
 - 只有推理时需要顺序生成

```
# 训练时 (并行)
input_ids = ["我", "爱", "中国"]
# 一次性计算所有位置的表示

# 推理时 (顺序)
# Step 1: "我" → 生成"爱"
# Step 2: "我爱" → 生成"中国"
# Step 3: "我爱中国" → 生成<EOS>
```

Q3: Multi-Head Attention中, 为什么拆分后每个头的维度要减小?

答案要点:

1. **参数量保持**: 总参数量与单头相同
2. **计算量保持**: 避免计算量爆炸
3. **子空间学习**: 每个头在不同子空间学习不同模式

单头: $d_{\text{model}}=512$, 1个头, $d_k=512$
 多头: $d_{\text{model}}=512$, 8个头, $d_k=64$

参数量:
 单头: $512 \times 512 \times 4 = 1,048,576$
 多头: $512 \times 512 \times 4 = 1,048,576$ (相同)

Q4: Transformer的时间复杂度和空间复杂度?

时间复杂度:

| 操作 | 复杂度 | 说明 |
|----------------|------------------|--------------|
| Self-Attention | $O(n^2 \cdot d)$ | n=序列长度, d=维度 |

| 操作 | 复杂度 | 说明 |
|--------------|--------------------------------|-----------|
| Feed-Forward | $O(n \cdot d^2)$ | 通常d_ff=4d |
| 总计 | $O(n^2 \cdot d + n \cdot d^2)$ | 序列长度主导 |

空间复杂度：

- 存储attention矩阵： $O(n^2)$
- KV Cache（推理时）： $O(n \cdot d)$

长序列问题： 当n很大时， n^2 是瓶颈， 解决方案：

- Sparse Attention
- Sliding Window Attention
- Flash Attention（计算优化）

Q5: 如何理解Transformer中的"Attention is All You Need"?

答案要点：

1. **抛弃循环和卷积**： 纯基于attention
2. **直接建模长距离依赖**： 任意两个位置可以直接交互
3. **并行化**： 训练效率高
4. **可解释性**： attention权重可视化

但实际上还需要：

- ☒ Position Encoding（位置信息）
- ☒ Feed-Forward（非线性）
- ☒ Residual & Norm（训练稳定性）

2.11 本章小结

本章深入讲解了Transformer架构，这是现代大模型的基石：

☒ **Self-Attention机制**： 核心创新，建模序列关系 ☒ **Multi-Head Attention**： 多角度学习不同模式 ☒
Position Encoding： 引入位置信息，RoPE是现代主流 ☒ **Feed-Forward Network**： 增加非线性，占参数量2/3 ☒ **Decoder-only架构**： 现代大模型的主流选择

实践建议：

1. 手写实现一个简化的Transformer模型
2. 可视化attention权重，理解模型学到的模式
3. 对比不同位置编码的效果

下一章预告： 第3章将回顾预训练语言模型的发展历史，从Word2Vec到GPT-4。