第15章 算法面试题精选

> 手写代码是大模型面试的必考环节，考察对核心算法的理解程度

15.1 Attention机制实现

15.1.1 Scaled Dot-Product Attention

题目： 手写实现Scaled Dot-Product Attention，包含mask支持。

解答：

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

def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Scaled Dot-Product Attention

    Args:
        Q: Query矩阵 (batch_size, seq_len_q, d_k)
        K: Key矩阵 (batch_size, seq_len_k, d_k)
        V: Value矩阵 (batch_size, seq_len_v, d_v)
        mask: 可选的mask矩阵 (batch_size, seq_len_q, seq_len_k)

    Returns:
        output: (batch_size, seq_len_q, d_v)
        attention_weights: (batch_size, seq_len_q, seq_len_k)
    """
    # 1. 计算注意力分数
    d_k = Q.size(-1)
    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)
    # scores shape: (batch_size, seq_len_q, seq_len_k)

    # 2. 应用mask（如果有）
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))

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

    # 4. 加权求和
    output = torch.matmul(attention_weights, V)
    # output shape: (batch_size, seq_len_q, d_v)

    return output, attention_weights
```

```python
# 测试
batch_size = 2
seq_len = 4
d_k = 64

Q = torch.randn(batch_size, seq_len, d_k)
K = torch.randn(batch_size, seq_len, d_k)
V = torch.randn(batch_size, seq_len, d_k)

# 因果mask (下三角矩阵)
causal_mask = torch.tril(torch.ones(seq_len, seq_len)).unsqueeze(0)

output, attn = scaled_dot_product_attention(Q, K, V, mask=causal_mask)
print(f"Output shape: {output.shape}")  # (2, 4, 64)
print(f"Attention shape: {attn.shape}")  # (2, 4, 4)
```

追问：为什么要除以√d_k?

```python
# 数值稳定性分析
import numpy as np

d_k_values = [64, 128, 256, 512]

for d_k in d_k_values:
    Q = np.random.randn(100, d_k)
    K = np.random.randn(100, d_k)

    scores_no_scale = np.dot(Q, K.T)
    scores_scaled = np.dot(Q, K.T) / np.sqrt(d_k)

    print(f"d_k={d_k}:")
    print(f"  不缩放方差: {np.var(scores_no_scale):.2f}")
    print(f"  缩放后方差: {np.var(scores_scaled):.2f}")

# 输出示例:
# d_k=64:   不缩放方差: 64.23, 缩放后方差: 1.00
# d_k=512:  不缩放方差: 512.45, 缩放后方差: 1.00
```

答案： 保持方差为1，防止softmax梯度消失。

15.1.2 Multi-Head Attention

题目： 实现完整的Multi-Head Attention层。

```python
import torch
import torch.nn as nn
```

```python
import math

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: 头数 (如8)
            dropout: dropout比例
        """
        super(MultiHeadAttention, self).__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  # 每个头的维度

        # 定义W_Q, W_K, W_V, W_O
        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)
        self.W_O = nn.Linear(d_model, d_model)

        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, 1, seq_len, seq_len) or None

        Returns:
            output: (batch_size, seq_len, d_model)
            attention_weights: (batch_size, num_heads, seq_len, seq_len)
        """
        batch_size = Q.size(0)

        # 1. 线性变换并分割成多个头
        # (batch_size, seq_len, d_model) -> (batch_size, seq_len, num_heads, d_k)
        # -> (batch_size, num_heads, seq_len, d_k)
        Q = self.W_Q(Q).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        K = self.W_K(K).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        V = self.W_V(V).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)

        # 2. 计算Scaled Dot-Product Attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

        if mask is not None:
```

```python
            scores = scores.masked_fill(mask == 0, float('-inf'))

        attention_weights = F.softmax(scores, dim=-1)
        attention_weights = self.dropout(attention_weights)

        context = torch.matmul(attention_weights, V)
        # context shape: (batch_size, num_heads, seq_len, d_k)

        # 3. 拼接多个头
        # (batch_size, num_heads, seq_len, d_k) -> (batch_size, seq_len, num_heads, d_k)
        # -> (batch_size, seq_len, d_model)
        context = context.transpose(1, 2).contiguous().view(batch_size, -1, self.d_model

        # 4. 最后的线性变换
        output = self.W_O(context)

        return output, attention_weights


# 测试
mha = MultiHeadAttention(d_model=512, num_heads=8)
x = torch.randn(2, 10, 512)  # (batch, seq_len, d_model)

output, attn = mha(x, x, x)
print(f"Output shape: {output.shape}")  # (2, 10, 512)
print(f"Attention shape: {attn.shape}")  # (2, 8, 10, 10)
```

追问1：如何创建因果mask?

```python
def create_causal_mask(seq_len, device='cpu'):
    """
    创建因果mask（下三角矩阵）

    Returns:
        mask: (1, 1, seq_len, seq_len)
    """
    mask = torch.tril(torch.ones(seq_len, seq_len, device=device))
    return mask.unsqueeze(0).unsqueeze(0)  # 添加batch和head维度

# 使用
seq_len = 5
mask = create_causal_mask(seq_len)
print(mask.squeeze())
# tensor([[1., 0., 0., 0., 0.],
#         [1., 1., 0., 0., 0.],
#         [1., 1., 1., 0., 0.],
```

```
#         [1., 1., 1., 1., 0.],
#         [1., 1., 1., 1., 1.]])
```

追问2：参数量计算?

```python
def count_mha_parameters(d_model, num_heads):
    """
    计算Multi-Head Attention的参数量
    """
    # W_Q, W_K, W_V: 各 d_model × d_model
    # W_O: d_model × d_model
    # 偏置: 4 × d_model (如果有)

    params = 4 * d_model * d_model + 4 * d_model  # 包含bias

    return params

# 示例
d_model = 512
num_heads = 8
params = count_mha_parameters(d_model, num_heads)
print(f"参数量: {params:,}")  # 1,052,672
```

15.2 位置编码实现

15.2.1 Sinusoidal Position Encoding

题目： 实现Transformer原始论文的位置编码。

```python
import torch
import math

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        """
        Sinusoidal位置编码

        PE(pos, 2i)   = sin(pos / 10000^(2i/d_model))
        PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

        Args:
            d_model: 模型维度
            max_len: 最大序列长度
        """
        super(PositionalEncoding, self).__init__()

        # 创建位置编码矩阵
```

```python
        pe = torch.zeros(max_len, d_model)

        # 位置索引: (max_len, 1)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)

        # 计算div_term: 10000^(2i/d_model)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
                            (-math.log(10000.0) / d_model))

        # 偶数位置: sin
        pe[:, 0::2] = torch.sin(position * div_term)

        # 奇数位置: cos
        pe[:, 1::2] = torch.cos(position * div_term)

        # 添加batch维度: (1, max_len, d_model)
        pe = pe.unsqueeze(0)

        # 注册为buffer（不会被视为模型参数）
        self.register_buffer('pe', pe)

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

        Returns:
            x + PE: (batch_size, seq_len, d_model)
        """
        seq_len = x.size(1)
        x = x + self.pe[:, :seq_len, :]
        return x


# 测试
pe = PositionalEncoding(d_model=512, max_len=100)
x = torch.randn(2, 50, 512)
output = pe(x)
print(f"Output shape: {output.shape}")  # (2, 50, 512)

# 可视化位置编码
import matplotlib.pyplot as plt

pos_encoding = pe.pe.squeeze().numpy()  # (max_len, d_model)

plt.figure(figsize=(15, 5))
plt.imshow(pos_encoding[:50, :], cmap='RdBu', aspect='auto')
plt.xlabel('Dimension')
plt.ylabel('Position')
```

```python
plt.colorbar()
plt.title('Positional Encoding')
plt.show()
```

15.2.2 RoPE (Rotary Position Embedding)

题目： 实现LLaMA使用的RoPE位置编码。

```python
class RotaryPositionalEmbedding(nn.Module):
    def __init__(self, dim, max_seq_len=2048, base=10000):
        """
        RoPE位置编码

        Args:
            dim: 每个头的维度 (通常是d_model // num_heads)
            max_seq_len: 最大序列长度
            base: 频率基数
        """
        super().__init__()

        # 计算频率
        inv_freq = 1.0 / (base ** (torch.arange(0, dim, 2).float() / dim))
        self.register_buffer('inv_freq', inv_freq)

        # 预计算cos和sin
        t = torch.arange(max_seq_len, dtype=torch.float)
        freqs = torch.outer(t, inv_freq)  # (max_seq_len, dim//2)
        emb = torch.cat((freqs, freqs), dim=-1)  # (max_seq_len, dim)

        self.register_buffer('cos_cached', emb.cos())
        self.register_buffer('sin_cached', emb.sin())

    def forward(self, x, seq_len):
        """
        Args:
            x: (batch, heads, seq_len, dim)
            seq_len: 当前序列长度

        Returns:
            rotated_x: (batch, heads, seq_len, dim)
        """
        cos = self.cos_cached[:seq_len, :]
        sin = self.sin_cached[:seq_len, :]

        return self.apply_rotary_emb(x, cos, sin)

    def apply_rotary_emb(self, x, cos, sin):
        """
```

```
        应用旋转
        """
        # 分离奇偶维度
        x1 = x[..., ::2]   # 偶数维度
        x2 = x[..., 1::2]  # 奇数维度

        # 旋转
        rotated = torch.cat([
            x1 * cos - x2 * sin,
            x1 * sin + x2 * cos
        ], dim=-1)

        return rotated


# 测试
rope = RotaryPositionalEmbedding(dim=64, max_seq_len=128)
x = torch.randn(2, 8, 10, 64)  # (batch, heads, seq_len, dim)
output = rope(x, seq_len=10)
print(f"Output shape: {output.shape}")  # (2, 8, 10, 64)
```

15.3 Layer Normalization实现

题目： 手写Layer Normalization，并解释与Batch Norm的区别。

```
class LayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        """
        Layer Normalization

        Args:
            normalized_shape: 归一化的维度（通常是d_model）
            eps: 数值稳定性常数
        """
        super(LayerNorm, self).__init__()

        # 可学习的缩放和平移参数
        self.gamma = nn.Parameter(torch.ones(normalized_shape))
        self.beta = nn.Parameter(torch.zeros(normalized_shape))
        self.eps = eps

    def forward(self, x):
        """
        Args:
            x: (batch_size, seq_len, d_model) 或 (batch_size, d_model)

        Returns:
            normalized_x: 与x形状相同
```

```python
        """
        # 在最后一个维度（特征维度）计算均值和方差
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)

        # 归一化
        x_norm = (x - mean) / torch.sqrt(var + self.eps)

        # 缩放和平移
        out = self.gamma * x_norm + self.beta

        return out


# 测试
ln = LayerNorm(normalized_shape=512)
x = torch.randn(2, 10, 512)
output = ln(x)
print(f"Output shape: {output.shape}")  # (2, 10, 512)

# 验证归一化效果
print(f"输入均值: {x.mean():.4f}, 方差: {x.var():.4f}")
print(f"输出均值: {output.mean():.4f}, 方差: {output.var():.4f}")
```

对比Batch Norm：

```python
def compare_normalizations():
    """
    对比Layer Norm和Batch Norm
    """
    batch_size = 32
    seq_len = 10
    d_model = 512

    x = torch.randn(batch_size, seq_len, d_model)

    # Layer Norm: 在特征维度归一化
    ln = nn.LayerNorm(d_model)
    x_ln = ln(x)

    print("Layer Norm:")
    print(f"  每个样本的均值: {x_ln[0].mean():.4f}")
    print(f"  每个样本的方差: {x_ln[0].var():.4f}")

    # Batch Norm: 在batch维度归一化（需要调整维度）
    bn = nn.BatchNorm1d(d_model)
    x_bn = bn(x.transpose(1, 2)).transpose(1, 2)  # (B, D, L) -> (B, L, D)
```

```python
    print("\nBatch Norm:")
    print(f"  每个特征的均值: {x_bn[:, :, 0].mean():.4f}")
    print(f"  每个特征的方差: {x_bn[:, :, 0].var():.4f}")
```

维度	Layer Norm	Batch Norm
归一化维度	特征维度（每个样本独立）	Batch维度（跨样本）
依赖batch大小	✅ 否	❌ 是
训练/测试差异	小	大（需要运行均值/方差）
适用场景	**Transformer、RNN**	CNN

15.4 前馈网络实现

题目： 实现Transformer的Feed-Forward Network，包含GELU激活函数。

```python
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        """
        Position-wise Feed-Forward Network

        FFN(x) = max(0, xW1 + b1)W2 + b2  (使用ReLU)
        或
        FFN(x) = GELU(xW1 + b1)W2 + b2  (现代大模型常用)

        Args:
            d_model: 输入/输出维度 (如512)
            d_ff: 中间层维度 (通常是4*d_model, 如2048)
            dropout: dropout比例
        """
        super(FeedForward, self).__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()  # 或nn.ReLU()

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

        Returns:
            output: (batch_size, seq_len, d_model)
        """
        x = self.linear1(x)
        x = self.activation(x)
        x = self.dropout(x)
```

```python
        x = self.linear2(x)
        x = self.dropout(x)
        return x


# 测试
ffn = FeedForward(d_model=512, d_ff=2048)
x = torch.randn(2, 10, 512)
output = ffn(x)
print(f"Output shape: {output.shape}")  # (2, 10, 512)
```

GELU激活函数实现：

```python
class GELU(nn.Module):
    """
    Gaussian Error Linear Unit

    GELU(x) = x * Φ(x)
    其中Φ(x)是标准正态分布的累积分布函数

    近似版本：
    GELU(x) ≈ 0.5 * x * (1 + tanh(√(2/π) * (x + 0.044715 * x^3)))
    """
    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x, 3))
        ))

# 对比PyTorch内置的GELU
x = torch.linspace(-3, 3, 100)
gelu_custom = GELU()(x)
gelu_pytorch = F.gelu(x)

print(f"差异: {(gelu_custom - gelu_pytorch).abs().max():.6f}")  # 很小
```

15.5 完整Transformer Block

题目： 实现一个完整的Transformer Decoder Block（GPT风格）。

```python
class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        """
        Transformer Decoder Block (Pre-LN)

        x -> LayerNorm -> MultiHeadAttention -> Add ->
             LayerNorm -> FeedForward -> Add -> output
        """
```

```python
        super(TransformerBlock, self).__init__()

        self.attention = MultiHeadAttention(d_model, num_heads, dropout)
        self.ffn = FeedForward(d_model, d_ff, dropout)

        self.ln1 = LayerNorm(d_model)
        self.ln2 = LayerNorm(d_model)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        """
        Args:
            x: (batch_size, seq_len, d_model)
            mask: (batch_size, 1, seq_len, seq_len) or None

        Returns:
            output: (batch_size, seq_len, d_model)
        """
        # 1. Multi-Head Self-Attention + Residual
        attn_input = self.ln1(x)
        attn_output, _ = self.attention(attn_input, attn_input, attn_input, mask)
        x = x + self.dropout(attn_output)

        # 2. Feed-Forward + Residual
        ffn_input = self.ln2(x)
        ffn_output = self.ffn(ffn_input)
        x = x + self.dropout(ffn_output)

        return x


# 测试
block = TransformerBlock(d_model=512, num_heads=8, d_ff=2048)
x = torch.randn(2, 10, 512)
mask = create_causal_mask(10)

output = block(x, mask)
print(f"Output shape: {output.shape}")  # (2, 10, 512)
```

15.6 LoRA实现

题目： 实现LoRA（Low-Rank Adaptation）。

```python
class LoRALayer(nn.Module):
    def __init__(self, in_features, out_features, rank=8, alpha=16):
        """
        LoRA层
```

```
        输出 = Wx + (B@A)x * (alpha/rank)

        Args:
            in_features: 输入维度
            out_features: 输出维度
            rank: LoRA的秩
            alpha: 缩放因子
        """
        super(LoRALayer, self).__init__()

        self.rank = rank
        self.alpha = alpha
        self.scaling = alpha / rank

        # LoRA矩阵
        self.lora_A = nn.Parameter(torch.randn(in_features, rank) * 0.01)
        self.lora_B = nn.Parameter(torch.zeros(rank, out_features))

    def forward(self, x):
        """
        Args:
            x: (batch_size, ..., in_features)

        Returns:
            delta: (batch_size, ..., out_features)
        """
        # ΔW = B @ A
        delta = x @ self.lora_A @ self.lora_B
        return delta * self.scaling


class LinearWithLoRA(nn.Module):
    def __init__(self, linear_layer, rank=8, alpha=16):
        """
        为Linear层添加LoRA
        """
        super(LinearWithLoRA, self).__init__()

        # 原始层（冻结）
        self.linear = linear_layer
        for param in self.linear.parameters():
            param.requires_grad = False

        # LoRA层
        self.lora = LoRALayer(
            linear_layer.in_features,
            linear_layer.out_features,
            rank, alpha
```

```python
        )

    def forward(self, x):
        # 原始输出 + LoRA输出
        return self.linear(x) + self.lora(x)


# 测试
original_linear = nn.Linear(512, 512)
lora_linear = LinearWithLoRA(original_linear, rank=8)

# 统计参数量
original_params = sum(p.numel() for p in original_linear.parameters())
lora_params = sum(p.numel() for p in lora_linear.lora.parameters())

print(f"原始参数: {original_params:,}")  # 262,656
print(f"LoRA参数: {lora_params:,}")      # 8,192
print(f"参数减少: {original_params/lora_params:.1f}倍")  # 32倍

# 前向传播
x = torch.randn(2, 10, 512)
output = lora_linear(x)
print(f"Output shape: {output.shape}")  # (2, 10, 512)
```

15.7 Softmax变体

15.7.1 数值稳定的Softmax

题目： 实现数值稳定的Softmax函数。

```python
def softmax_naive(x):
    """
    朴素实现（可能溢出）
    """
    exp_x = np.exp(x)
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)


def softmax_stable(x):
    """
    数值稳定版本

    技巧：减去最大值不改变softmax结果，但防止exp溢出
    softmax(x) = softmax(x - max(x))
    """
    x_max = np.max(x, axis=-1, keepdims=True)
    exp_x = np.exp(x - x_max)
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)
```

```python
# 测试
x_large = np.array([1000, 1001, 1002])

print("朴素版本:")
try:
    result = softmax_naive(x_large)
    print(result)
except:
    print("溢出！")  # 会溢出

print("\n稳定版本:")
result = softmax_stable(x_large)
print(result)  # [0.09, 0.24, 0.67]
```

15.7.2 LogSumExp技巧

```python
def log_softmax(x):
    """
    计算log(softmax(x))，数值稳定版本

    log(softmax(x_i)) = x_i - log(∑ exp(x_j))
                      = x_i - LSE(x)
    """
    x_max = np.max(x, axis=-1, keepdims=True)

    # LogSumExp技巧
    lse = x_max + np.log(np.sum(np.exp(x - x_max), axis=-1, keepdims=True))

    return x - lse


def cross_entropy_loss(logits, labels):
    """
    使用log_softmax计算交叉熵损失
    """
    log_probs = log_softmax(logits)

    # 选择正确类别的log概率
    nll = -log_probs[np.arange(len(labels)), labels]

    return nll.mean()


# 测试
logits = np.random.randn(10, 5)  # (batch_size=10, num_classes=5)
labels = np.array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
```

```python
loss = cross_entropy_loss(logits, labels)
print(f"Loss: {loss:.4f}")
```

15.8 Embedding层实现

题目： 实现Token Embedding + Position Embedding。

```python
class Embeddings(nn.Module):
    def __init__(self, vocab_size, d_model, max_len=512):
        """
        Token Embedding + Positional Embedding
        """
        super(Embeddings, self).__init__()

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

        # Positional Embedding (learned)
        self.position_embedding = nn.Embedding(max_len, d_model)

        self.d_model = d_model

    def forward(self, x):
        """
        Args:
            x: (batch_size, seq_len) - token ids

        Returns:
            embeddings: (batch_size, seq_len, d_model)
        """
        batch_size, seq_len = x.size()

        # Token embeddings
        token_emb = self.token_embedding(x)  # (B, L, D)

        # Position ids
        positions = torch.arange(seq_len, device=x.device).unsqueeze(0).expand(batch_si

        # Position embeddings
        pos_emb = self.position_embedding(positions)  # (B, L, D)

        # 相加（有些模型会乘以sqrt(d_model)）
        embeddings = token_emb + pos_emb

        return embeddings
```

```python
# 测试
vocab_size = 50000
d_model = 512
max_len = 512

emb = Embeddings(vocab_size, d_model, max_len)

# 输入token ids
input_ids = torch.randint(0, vocab_size, (2, 20))  # (batch=2, seq_len=20)

embeddings = emb(input_ids)
print(f"Embeddings shape: {embeddings.shape}")  # (2, 20, 512)
```

15.9 Beam Search实现

题目： 实现文本生成的Beam Search算法。

```python
def beam_search(model, input_ids, beam_size=5, max_length=50,
                eos_token_id=2, temperature=1.0):
    """
    Beam Search解码

    Args:
        model: 语言模型
        input_ids: 输入token ids (batch_size=1, seq_len)
        beam_size: beam大小
        max_length: 最大生成长度
        eos_token_id: 结束符id
        temperature: 温度参数

    Returns:
        best_sequence: 最佳序列
        best_score: 最佳分数
    """
    device = input_ids.device
    batch_size = input_ids.size(0)
    assert batch_size == 1, "Beam search目前只支持batch_size=1"

    # 初始化beams: [(序列, 分数)]
    beams = [(input_ids[0].tolist(), 0.0)]
    completed_beams = []

    for step in range(max_length):
        all_candidates = []

        for seq, score in beams:
            # 如果已经生成EOS，跳过
            if seq[-1] == eos_token_id:
```

```python
                completed_beams.append((seq, score))
                continue

            # 获取下一个token的概率
            input_tensor = torch.tensor([seq], device=device)
            with torch.no_grad():
                logits = model(input_tensor)[:, -1, :]  # (1, vocab_size)

            # 应用温度
            logits = logits / temperature

            # 计算log概率
            log_probs = F.log_softmax(logits, dim=-1)[0]  # (vocab_size,)

            # 获取top-k候选
            topk_log_probs, topk_ids = torch.topk(log_probs, beam_size)

            # 扩展beam
            for i in range(beam_size):
                new_seq = seq + [topk_ids[i].item()]
                new_score = score + topk_log_probs[i].item()
                all_candidates.append((new_seq, new_score))

        # 选择top beam_size个候选
        # 按平均log概率排序（归一化序列长度）
        all_candidates.sort(key=lambda x: x[1] / len(x[0]), reverse=True)
        beams = all_candidates[:beam_size]

        # 如果所有beams都完成，提前结束
        if len(completed_beams) >= beam_size:
            break

    # 合并completed和未完成的beams
    all_beams = completed_beams + beams
    all_beams.sort(key=lambda x: x[1] / len(x[0]), reverse=True)

    best_sequence, best_score = all_beams[0]

    return best_sequence, best_score


# 简化的测试示例
class SimpleModel(nn.Module):
    def __init__(self, vocab_size, d_model):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.lm_head = nn.Linear(d_model, vocab_size)

    def forward(self, x):
```

```python
        x = self.embedding(x)
        logits = self.lm_head(x)
        return logits

model = SimpleModel(vocab_size=1000, d_model=128)
input_ids = torch.tensor([[1]])  # 起始token

best_seq, score = beam_search(model, input_ids, beam_size=3, max_length=10)
print(f"最佳序列: {best_seq}")
print(f"分数: {score:.4f}")
```

15.10 梯度裁剪

题目： 实现梯度裁剪（Gradient Clipping）。

```python
def clip_grad_norm(parameters, max_norm, norm_type=2):
    """
    梯度裁剪

    Args:
        parameters: 模型参数（可迭代）
        max_norm: 梯度的最大范数
        norm_type: 范数类型（1, 2, inf）

    Returns:
        total_norm: 裁剪前的总梯度范数
    """
    parameters = list(filter(lambda p: p.grad is not None, parameters))

    if len(parameters) == 0:
        return torch.tensor(0.)

    device = parameters[0].grad.device

    if norm_type == float('inf'):
        # L-inf范数
        total_norm = max(p.grad.data.abs().max() for p in parameters)
    else:
        # L-p范数
        total_norm = torch.norm(
            torch.stack([torch.norm(p.grad.data, norm_type) for p in parameters]),
            norm_type
        )

    # 计算裁剪系数
    clip_coef = max_norm / (total_norm + 1e-6)

    # 如果总范数超过max_norm，则裁剪
```

```python
        if clip_coef < 1:
            for p in parameters:
                p.grad.data.mul_(clip_coef)

        return total_norm


    # 使用示例
    model = nn.Linear(10, 5)
    optimizer = torch.optim.Adam(model.parameters())

    # 训练步骤
    x = torch.randn(32, 10)
    y = torch.randn(32, 5)

    output = model(x)
    loss = F.mse_loss(output, y)
    loss.backward()

    # 梯度裁剪
    total_norm = clip_grad_norm(model.parameters(), max_norm=1.0)
    print(f"Total gradient norm: {total_norm:.4f}")

    optimizer.step()
    optimizer.zero_grad()
```

15.11 学习率调度器

题目： 实现Warmup + Cosine Decay学习率调度。

```python
class WarmupCosineScheduler:
    def __init__(self, optimizer, warmup_steps, total_steps, min_lr=0):
        """
        Warmup + Cosine Decay学习率调度

        Args:
            optimizer: 优化器
            warmup_steps: warmup步数
            total_steps: 总训练步数
            min_lr: 最小学习率
        """
        self.optimizer = optimizer
        self.warmup_steps = warmup_steps
        self.total_steps = total_steps
        self.min_lr = min_lr

        # 保存初始学习率
        self.base_lrs = [group['lr'] for group in optimizer.param_groups]
```

```python
        self.current_step = 0

    def step(self):
        """
        更新学习率
        """
        self.current_step += 1

        if self.current_step < self.warmup_steps:
            # Warmup阶段：线性增长
            lr_scale = self.current_step / self.warmup_steps
        else:
            # Cosine Decay阶段
            progress = (self.current_step - self.warmup_steps) / \
                      (self.total_steps - self.warmup_steps)
            lr_scale = 0.5 * (1 + math.cos(math.pi * progress))

            # 确保不低于min_lr
            lr_scale = max(lr_scale, self.min_lr / self.base_lrs[0])

        # 更新所有参数组的学习率
        for param_group, base_lr in zip(self.optimizer.param_groups, self.base_lrs):
            param_group['lr'] = base_lr * lr_scale

    def get_lr(self):
        """
        获取当前学习率
        """
        return [group['lr'] for group in self.optimizer.param_groups]


# 测试
model = nn.Linear(10, 5)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)

total_steps = 10000
warmup_steps = 1000

scheduler = WarmupCosineScheduler(optimizer, warmup_steps, total_steps, min_lr=1e-5)

# 记录学习率变化
lrs = []
for step in range(total_steps):
    scheduler.step()
    lrs.append(scheduler.get_lr()[0])

# 可视化
import matplotlib.pyplot as plt
```

```python
plt.figure(figsize=(10, 5))
plt.plot(lrs)
plt.xlabel('Step')
plt.ylabel('Learning Rate')
plt.title('Warmup + Cosine Decay')
plt.grid(True)
plt.show()
```

15.12 本章小结

本章精选了大模型面试中的高频算法题：

✅ **Attention机制**：Scaled Dot-Product + Multi-Head ✅ **位置编码**：Sinusoidal + RoPE ✅ **归一化**：Layer Norm的实现和原理 ✅ **LoRA**：参数高效微调的核心算法 ✅ **训练技巧**：梯度裁剪、学习率调度 ✅ **生成算法**：Beam Search

备考建议：

1. 手写代码时注意维度变换，多用注释标注shape
2. 理解算法原理，能解释为什么这样设计
3. 准备好常见的追问（参数量计算、复杂度分析）
4. 实际运行代码，确保能work

刷题策略：

- 核心算法每周手写1-2遍
- 不看代码默写，然后对比修正
- 准备每个算法的3-5个变体题

下一章预告：第16章将讲解系统设计题，如何设计一个完整的对话系统。