

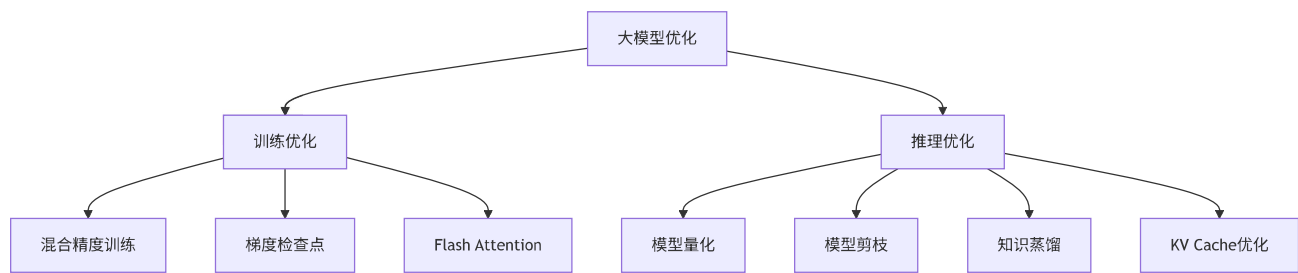
第6章 高效训练与推理

让大模型训练更快、推理更省、部署更易

6.1 概述

大模型面临的挑战：

- 训练慢：GPT-3需要数千GPU天
- 显存大：70B模型需要140GB+显存
- 推理慢：生成速度是产品体验的关键
- 成本高：API调用成本高昂



6.2 量化技术

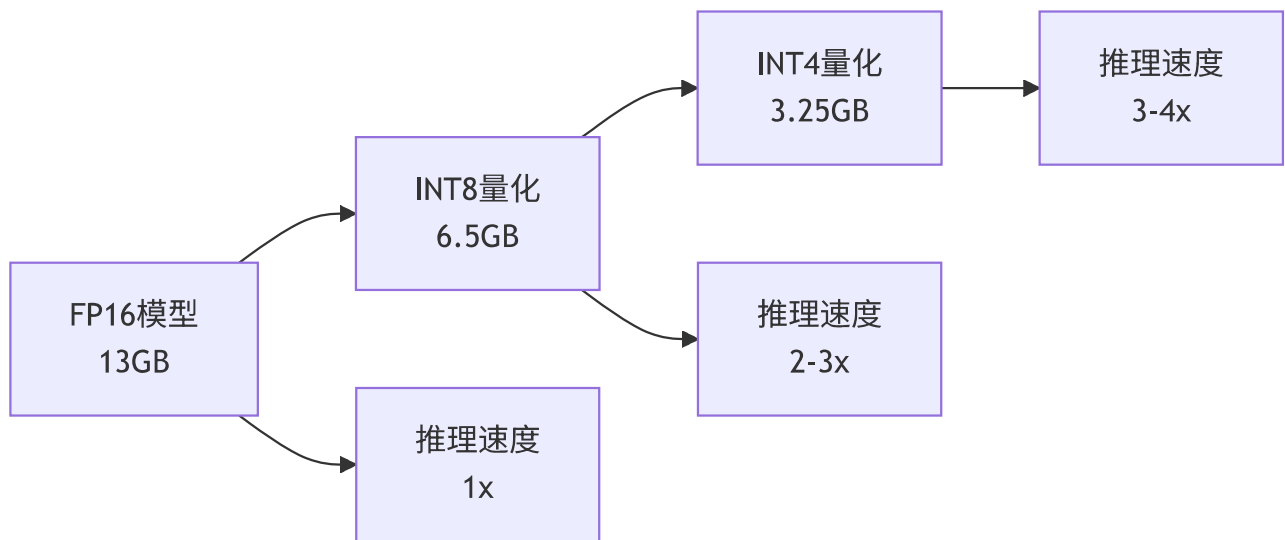
6.2.1 量化基础

定义： 将高精度（FP32/FP16）的模型参数和激活值转换为低精度（INT8/INT4）。

精度对比：

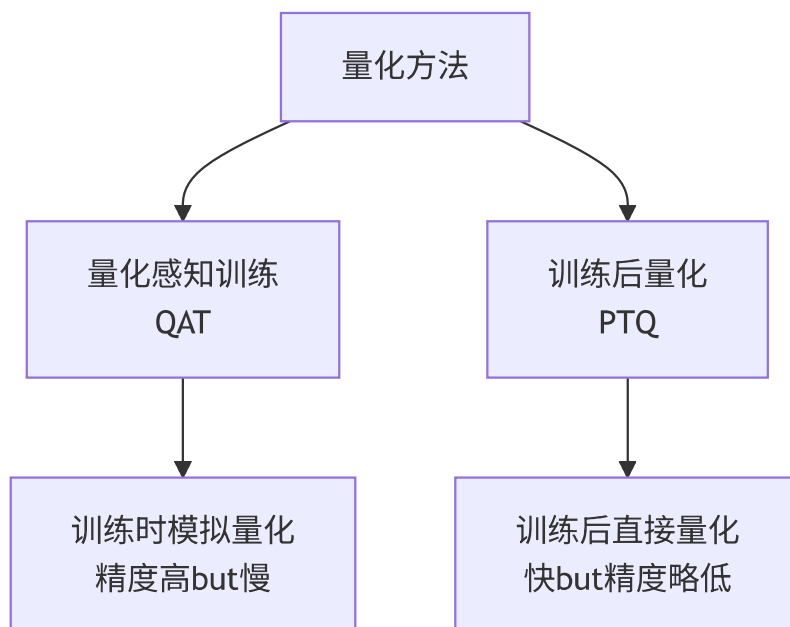
数据类型	位数	范围	精度	用途
FP32	32	$\pm 3.4 \times 10^{38}$	7位小数	训练
FP16	16	± 65504	3位小数	训练/推理
BF16	16	$\pm 3.4 \times 10^{38}$	2位小数	训练
INT8	8	-128~127	整数	推理
INT4	4	-8~7	整数	推理

量化收益：



6.2.2 量化方法分类

按时机分类：



按粒度分类：

- **Per-Tensor**：整个张量用同一个缩放因子
- **Per-Channel**：每个输出通道独立缩放
- **Per-Group**：每组用独立缩放因子（更精细）

6.2.3 对称量化 vs 非对称量化

对称量化：

$$X_{int8} = \text{round}\left(\frac{X_{fp32}}{scale}\right)$$

$$scale = \frac{\max(|X|)}{127}$$

```

def symmetric_quantize(tensor, bits=8):
    """
    对称量化
    """
    qmax = 2 ** (bits - 1) - 1 # INT8: 127

    # 计算scale
    scale = tensor.abs().max() / qmax

    # 量化
    quantized = torch.round(tensor / scale).clamp(-qmax-1, qmax)

    return quantized.to(torch.int8), scale

def symmetric_dequantize(quantized, scale):
    """
    反量化
    """
    return quantized.float() * scale

```

非对称量化 (使用zero-point) :

$$X_{int8} = \text{round}\left(\frac{X_{fp32}}{scale} + zero_point\right)$$

$$scale = \frac{\max(X) - \min(X)}{255}$$

$$zero_point = \text{round}\left(-\frac{\min(X)}{scale}\right)$$

```

def asymmetric_quantize(tensor, bits=8):
    """
    非对称量化
    """
    qmin = 0
    qmax = 2 ** bits - 1 # INT8: 255

    # 计算scale和zero_point
    min_val = tensor.min()
    max_val = tensor.max()

    scale = (max_val - min_val) / (qmax - qmin)
    zero_point = qmin - torch.round(min_val / scale)

    # 量化

```

```

quantized = torch.round(tensor / scale + zero_point).clamp(qmin, qmax)

return quantized.to(torch.uint8), scale, zero_point

def asymmetric_dequantize(quantized, scale, zero_point):
    """
    反量化
    """
    return (quantized.float() - zero_point) * scale

```

6.2.4 GPTQ (Post-Training Quantization for GPT)

核心思想： 最小化量化后的重构误差。

算法流程：

```

def gptq_quantize_layer(weight, bits=4, group_size=128):
    """
    GPTQ量化单层权重

    Args:
        weight: (out_features, in_features)
        bits: 量化位数
        group_size: 分组大小
    """
    out_features, in_features = weight.shape

    # 计算Hessian矩阵（用于确定量化顺序）
    H = compute_hessian(weight)

    # 按列量化
    quantized = torch.zeros_like(weight)




    for i in range(in_features):
        # 量化当前列
        w_col = weight[:, i]
        q_col = quantize_column(w_col, bits, group_size)
        quantized[:, i] = q_col

        # 补偿误差到未量化的列（关键步骤）
        error = w_col - q_col
        weight[:, i+1:] -= error.unsqueeze(1) @ H[i, i+1:].unsqueeze(0) / H[i, i]

```

```
return quantized
```

特点:

-  无需训练数据
-  4-bit量化精度损失小 (<1%)
-  适合大模型

6.2.5 AWQ (Activation-aware Weight Quantization)

观察: 不是所有权重都同等重要, 某些权重对应的激活值更大。

策略: 基于激活值的大小调整量化scale。

```
def awq_quantize(weight, activations, bits=4):
    """
    AWQ量化




    Args:
        weight: 权重矩阵
        activations: 对应的激活值 (从校准数据获得)
    """
    # 计算每个通道的激活重要性
    importance = activations.abs().mean(dim=0)  # (in_features,)

    # 根据重要性调整scale
    # 重要的通道使用更大的scale (更精细的量化)
    scale = compute_scale(weight, importance)

    # 量化
    quantized = torch.round(weight / scale).clamp(-2**(bits-1), 2**(bits-1)-1)

    return quantized, scale
```

优势:

-  保护重要权重
-  4-bit性能优于GPTQ
-  需要少量校准数据

6.2.6 实际使用

使用GPTQ量化:

```

from transformers import AutoModelForCausalLM, GPTQConfig

# GPTQ配置
gptq_config = GPTQConfig(
    bits=4,
    group_size=128,
    desc_act=False, # 是否量化激活值
)

# 加载量化模型
model = AutoModelForCausalLM.from_pretrained(
    "TheBloke/Llama-2-7B-GPTQ",
    device_map="auto",
    quantization_config=gptq_config
)

# 推理
outputs = model.generate(input_ids, max_length=100)

```

使用AWQ量化:

```

from awq import AutoAWQForCausalLM

# 加载模型
model = AutoAWQForCausalLM.from_pretrained("llama-2-7b")

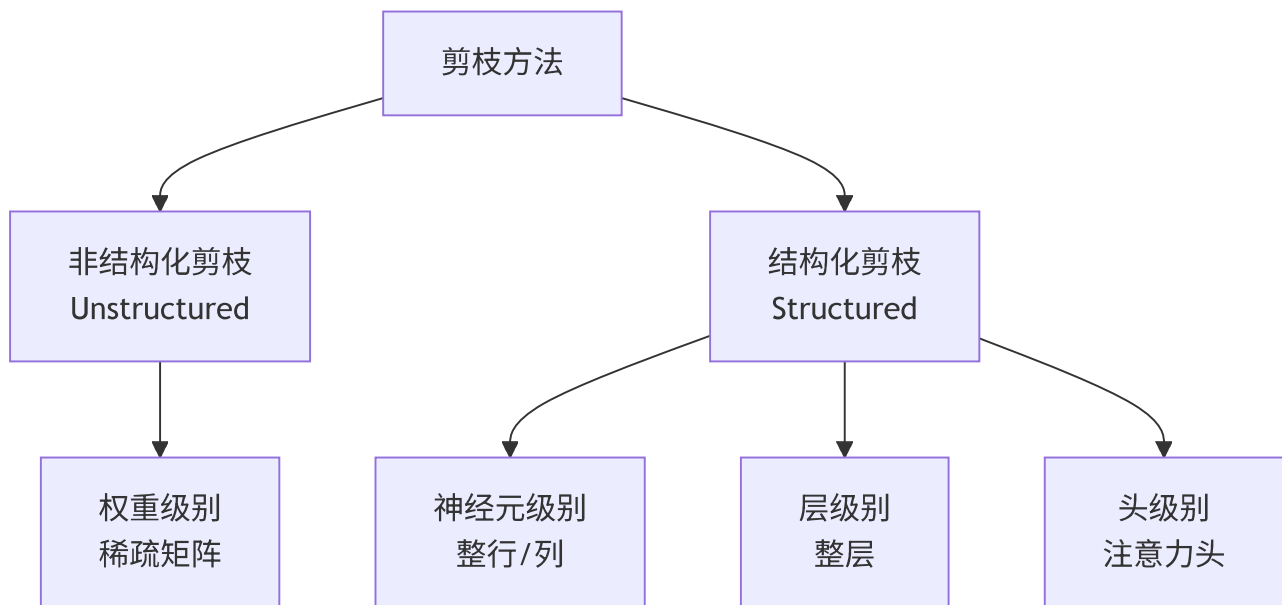
# 量化（需要校准数据）
model.quantize(
    tokenizer,
    quant_config={"bits": 4, "group_size": 128},
    calib_data="wikitext" # 校准数据集
)

# 保存
model.save_quantized("llama-2-7b-awq")

```

6.3 模型剪枝

6.3.1 剪枝类型



非结构化剪枝:

```
def magnitude_pruning(weight, sparsity=0.5):  
    """  
    幅度剪枝: 剪掉绝对值最小的权重  
  
    Args:  
        weight: 权重矩阵  
        sparsity: 稀疏度 (剪掉的比例)  
    """  
    # 计算阈值  
    threshold = torch.quantile(weight.abs(), sparsity)  
  
    # 创建mask  
    mask = (weight.abs() > threshold).float()  
  
    # 应用mask  
    pruned_weight = weight * mask  
  
    return pruned_weight, mask
```

优点:

- ☒ 灵活, 可达到高稀疏度

缺点:

- ☒ 需要专门的稀疏矩阵库支持
- ☒ 实际加速有限

结构化剪枝 (更实用) :

```

def structured_pruning_attention_heads(model, importance_scores, prune_ratio=0.2):
    """
    剪枝注意力头

    Args:
        importance_scores: 每个头的重要性分数
        prune_ratio: 剪掉的头的比例
    """
    num_heads = len(importance_scores)
    num_to_prune = int(num_heads * prune_ratio)

    # 选择最不重要的头
    heads_to_prune = torch.argsort(importance_scores)[:num_to_prune]

    # 剪枝
    for layer_idx, head_idx in heads_to_prune:
        model.layers[layer_idx].attention.prune_head(head_idx)

    return model


def compute_head_importance(model, dataloader):
    """
    计算每个注意力头的重要性
    """
    importance = {}

    for batch in dataloader:
        outputs = model(batch, output_attentions=True)

        # 统计每个头的注意力权重
        for layer_idx, attn in enumerate(outputs.attentions):
            for head_idx in range(attn.size(1)):
                head_attn = attn[:, head_idx, :, :]
                importance[(layer_idx, head_idx)] = \
                    importance.get((layer_idx, head_idx), 0) + head_attn.sum().item()

    return importance

```

6.3.2 LLM-Pruner

针对大模型的结构化剪枝方法。

步骤：

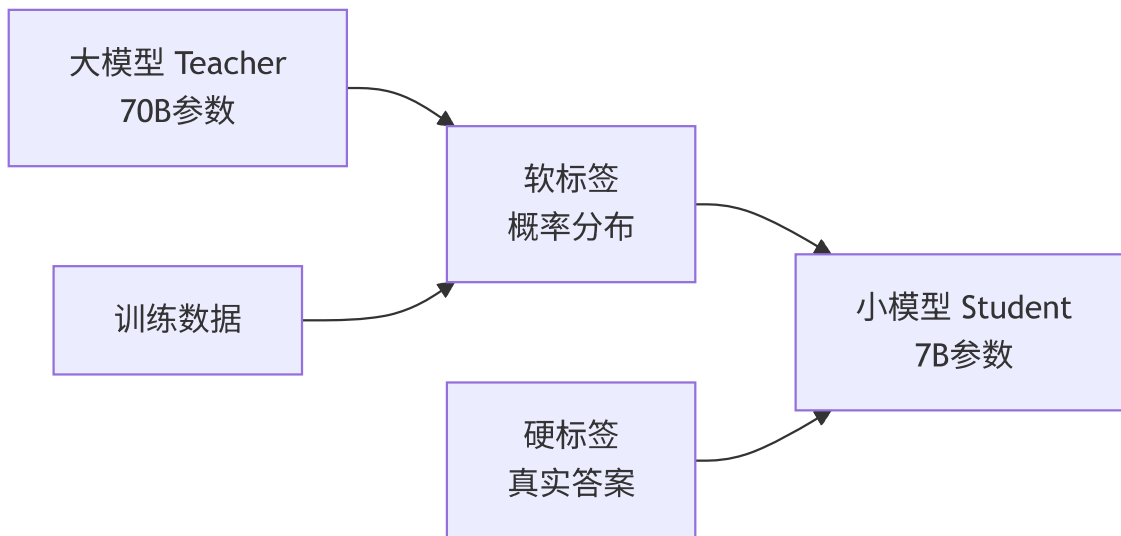
1. 识别依赖关系
2. 计算重要性分数
3. 分组剪枝
4. 快速恢复训练

```
def llm_pruner(model, target_sparsity=0.2):  
    """  
    LLM-Pruner简化实现  
    """  
  
    # 1. 计算每个神经元的重要性  
    importance_scores = compute_neuron_importance(model)  
  
    # 2. 识别依赖组  
    groups = identify_dependency_groups(model)  
  
    # 3. 组级别剪枝  
    for group in groups:  
        group_importance = sum(importance_scores[neuron] for neuron in group)  
  
        if group_importance < threshold:  
            # 剪掉整个组  
            prune_group(model, group)  
  
    # 4. 快速恢复训练 (LoRA)  
    apply_lora_to_pruned_model(model)  
  
    return model
```

6.4 知识蒸馏

6.4.1 知识蒸馏原理

目标： 让小模型（Student）学习大模型（Teacher）的知识。



损失函数:

$$\mathcal{L} = \alpha \mathcal{L}_{CE}(y, \hat{y}) + (1 - \alpha) \mathcal{L}_{KL}(p_T, p_S)$$

其中:

- \mathcal{L}_{CE} : 交叉熵损失 (硬标签)
- \mathcal{L}_{KL} : KL散度 (软标签)
- α : 权衡系数

6.4.2 标准知识蒸馏

```

def distillation_loss(student_logits, teacher_logits, labels,
                      temperature=2.0, alpha=0.5):
    """
    知识蒸馏损失

    Args:
        student_logits: 学生模型的logits
        teacher_logits: 教师模型的logits
        labels: 真实标签
        temperature: 温度参数 (平滑概率分布)
        alpha: 硬标签损失的权重
    """
    # 硬标签损失
    hard_loss = F.cross_entropy(student_logits, labels)

    # 软标签损失
    student_soft = F.log_softmax(student_logits / temperature, dim=-1)
    teacher_soft = F.softmax(teacher_logits / temperature, dim=-1)

    soft_loss = F.kl_div(student_soft, teacher_soft, reduction='batchmean')
    soft_loss = soft_loss * (temperature ** 2) # 温度平方校正
  
```

```

# 总损失
loss = alpha * hard_loss + (1 - alpha) * soft_loss

return loss

def train_distillation(student_model, teacher_model, dataloader):
    """
    知识蒸馏训练
    """
    teacher_model.eval() # 教师模型不更新

    for batch in dataloader:
        input_ids = batch['input_ids']
        labels = batch['labels']

        # 教师模型推理
        with torch.no_grad():
            teacher_logits = teacher_model(input_ids).logits

        # 学生模型推理
        student_logits = student_model(input_ids).logits

        # 计算蒸馏损失
        loss = distillation_loss(student_logits, teacher_logits, labels)

        # 反向传播
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

```

6.4.3 序列级蒸馏

对于生成任务，可以让学生模仿教师的生成结果。

```

def sequence_level_distillation(teacher_model, student_model, prompts):
    """
    序列级蒸馏：学生学习教师生成的文本
    """
    # 1. 教师模型生成
    with torch.no_grad():
        teacher_outputs = teacher_model.generate(
            prompts,
            max_length=512,
            do_sample=False # 使用greedy decoding
        )

```

2. 学生模型学习教师的输出

```
for batch in create_batches(prompts, teacher_outputs):
    loss = student_model(
        input_ids=batch['input_ids'],
        labels=batch['labels']
    ).loss

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

6.4.4 在上下文蒸馏 (In-Context Distillation)

让学生学习教师的in-context learning能力。

```
def in_context_distillation(teacher_model, student_model, tasks):
    """
    上下文蒸馏
    """
    for task in tasks:
        # 构造few-shot prompt
        prompt = create_few_shot_prompt(task)

        # 教师模型few-shot推理
        teacher_output = teacher_model.generate(prompt)

        # 学生模型直接学习（无需few-shot）
        # 学生的输入：只包含最后的query
        student_input = task['query']
        student_loss = student_model(
            input_ids=student_input,
            labels=teacher_output
        ).loss

        student_loss.backward()
```

6.5 KV Cache优化

6.5.1 KV Cache原理

在自回归生成中，避免重复计算已生成token的K和V。

无KV Cache:

```
# 每步都要计算所有token的K和V
# Step 1: 计算 token_1 的 K, V
# Step 2: 计算 token_1, token_2 的 K, V (重复计算token_1)
# Step 3: 计算 token_1, token_2, token_3 的 K, V (重复计算token_1, token_2)
```

有KV Cache:

```
# 只计算新token的K和V, 之前的缓存起来
# Step 1: 计算 token_1 的 K, V → 缓存
# Step 2: 计算 token_2 的 K, V → 拼接到缓存
# Step 3: 计算 token_3 的 K, V → 拼接到缓存
```

实现:

```
class AttentionWithKVCache(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        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)

    def forward(self, x, kv_cache=None):
        """
        Args:
            x: 当前step的输入 (batch, 1, d_model) 生成时
            kv_cache: 缓存的 (K, V)
        """
        batch_size = x.size(0)

        # 计算Q, K, V
        Q = self.W_Q(x).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        K = self.W_K(x).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        V = self.W_V(x).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)

        # 如果有缓存, 拼接
        if kv_cache is not None:
            K_cache, V_cache = kv_cache
            K = torch.cat([K_cache, K], dim=2)
            V = torch.cat([V_cache, V], dim=2)
```

```

# Attention计算
scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
attn = F.softmax(scores, dim=-1)
context = torch.matmul(attn, V)

# 输出
context = context.transpose(1, 2).contiguous().view(batch_size, -1, self.d_
output = self.W_O(context)

# 返回输出和新的cache
return output, (K, V)

def generate_with_kv_cache(model, input_ids, max_length=100):
    """
    使用KV Cache生成
    """
    kv_caches = [None] * len(model.layers)

    for step in range(max_length):
        # 如果是第一步，输入全部tokens；否则只输入最新的token
        if step == 0:
            current_input = input_ids
        else:
            current_input = next_token_id.unsqueeze(0)

        # 逐层前向传播，更新KV cache
        hidden_states = model.embed(current_input)

        for i, layer in enumerate(model.layers):
            hidden_states, kv_caches[i] = layer(hidden_states, kv_caches[i])

        # 预测下一个token
        logits = model.lm_head(hidden_states[:, -1, :])
        next_token_id = torch.argmax(logits, dim=-1)

        input_ids = torch.cat([input_ids, next_token_id.unsqueeze(0)], dim=1)

        if next_token_id == eos_token_id:
            break

    return input_ids

```

KV Cache带来的收益：

指标	无KV Cache	有KV Cache
计算量	$O(n^2)$	$O(n)$
生成速度	慢	快2-3倍
显存占用	低	高 (需要存储cache)

6.5.2 MQA (Multi-Query Attention)

问题： KV Cache占用显存大（尤其是大batch size时）。

MQA方案： 所有头共享K和V，只有Q是多头的。

```
class MultiQueryAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_Q = nn.Linear(d_model, d_model) # 多头
        self.W_K = nn.Linear(d_model, self.d_k) # 单头!
        self.W_V = nn.Linear(d_model, self.d_k) # 单头!
        self.W_O = nn.Linear(d_model, d_model)

    def forward(self, x):
        batch_size, seq_len, _ = x.size()

        # Q: (batch, heads, seq_len, d_k)
        Q = self.W_Q(x).view(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)

        # K, V: (batch, 1, seq_len, d_k) - 单头!
        K = self.W_K(x).view(batch_size, seq_len, 1, self.d_k).transpose(1, 2)
        V = self.W_V(x).view(batch_size, seq_len, 1, self.d_k).transpose(1, 2)

        # K, V广播到所有头
        K = K.expand(-1, self.num_heads, -1, -1)
        V = V.expand(-1, self.num_heads, -1, -1)

        # 标准attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        attn = F.softmax(scores, dim=-1)
        output = torch.matmul(attn, V)

        output = output.transpose(1, 2).contiguous().view(batch_size, seq_len, -1)
        return self.W_O(output)
```

KV Cache减少:

```
Multi-Head Attention:
K cache: (batch, heads, seq_len, d_k) = batch × 32 × seq_len × 128
V cache: (batch, heads, seq_len, d_k) = batch × 32 × seq_len × 128

Multi-Query Attention:
K cache: (batch, 1, seq_len, d_k) = batch × 1 × seq_len × 128  (减少32倍!)
V cache: (batch, 1, seq_len, d_k) = batch × 1 × seq_len × 128
```

代表模型: PaLM, Falcon

6.5.3 GQA (Grouped-Query Attention)

折中方案: 将头分组, 组内共享K和V。

```
# 示例: 32个头分成8组, 每组4个头
num_heads = 32
num_kv_heads = 8 # K和V的头数
heads_per_group = num_heads // num_kv_heads # 4

# K, V: (batch, 8, seq_len, d_k)
# Q: (batch, 32, seq_len, d_k)

# 每组的4个Q共享同一组K和V
```

对比:

方法	K/V头数	KV Cache大小	性能	代表模型
MHA	32	1×	最好	GPT-3, LLaMA-1
GQA	8	0.25×	接近MHA	LLaMA-2
MQA	1	0.03×	略低	PaLM

6.6 Flash Attention

6.6.1 标准Attention的瓶颈

问题: 标准attention需要具化整个注意力矩阵 ($n \times n$)。

```
# 标准实现
Q = ... # (batch, heads, seq_len, d_k)
K = ... # (batch, heads, seq_len, d_k)
V = ... # (batch, heads, seq_len, d_k)
```



```
# 计算注意力矩阵 (seq_len × seq_len)
scores = Q @ K.T # 需要 $O(n^2)$ 显存!
attn = softmax(scores)
output = attn @ V
```

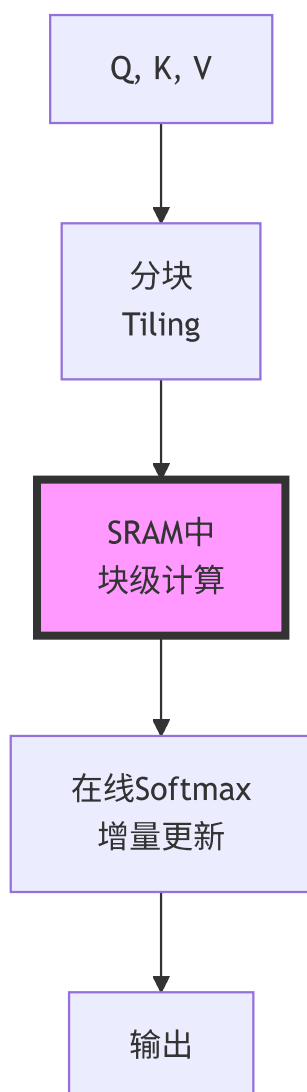
显存占用:

序列长度 $n=2048$, batch=1, heads=32, $d_k=128$

Attention矩阵: $32 \times 2048 \times 2048 \times 4 \text{ bytes} = 512 \text{ MB}$ (单个样本!)

6.6.2 Flash Attention原理

核心思想: 分块计算, 不存储完整的attention矩阵。



算法流程:

1. 将Q, K, V分成多个块
2. 外层循环K, V的块

3. 内层循环Q的块
4. 在SRAM中计算小块的attention
5. 使用在线softmax增量更新结果

简化伪代码：

```
def flash_attention(Q, K, V, block_size=64):
    """
    Flash Attention简化版本
    """
    seq_len, d_k = Q.shape
    num_blocks = seq_len // block_size

    O = torch.zeros_like(Q) # 输出
    l = torch.zeros(seq_len) # softmax归一化因子
    m = torch.full((seq_len,), -float('inf')) # softmax最大值

    # 外层循环: K, V的块
    for j in range(num_blocks):
        K_j = K[j*block_size:(j+1)*block_size]
        V_j = V[j*block_size:(j+1)*block_size]

        # 内层循环: Q的块
        for i in range(num_blocks):
            Q_i = Q[i*block_size:(i+1)*block_size]

            # 计算当前块的attention
            S_ij = Q_i @ K_j.T / math.sqrt(d_k)

            # 在线softmax更新
            m_new = torch.maximum(m[i*block_size:(i+1)*block_size], S_ij.max(dim=1))
            l_new = torch.exp(m[i*block_size:(i+1)*block_size] - m_new) * \
                l[i*block_size:(i+1)*block_size] + \
                torch.sum(torch.exp(S_ij - m_new.unsqueeze(1)), dim=1)

            # 更新输出
            O[i*block_size:(i+1)*block_size] = \
                (l[i*block_size:(i+1)*block_size].unsqueeze(1) * \
                 torch.exp(m[i*block_size:(i+1)*block_size].unsqueeze(1) - m_new) + \
                 O[i*block_size:(i+1)*block_size] + \
                 torch.exp(S_ij - m_new.unsqueeze(1)) @ V_j) / l_new.unsqueeze(1)

        m[i*block_size:(i+1)*block_size] = m_new
        l[i*block_size:(i+1)*block_size] = l_new
```

```
return 0
```

使用Flash Attention:

```
# 安装
# pip install flash-attn

from flash_attn import flash_attn_qkvpacked_func

# Q, K, V: (batch, seq_len, num_heads, d_k)
qkv = torch.stack([Q, K, V], dim=2) # (batch, seq_len, 3, num_heads, d_k)

output = flash_attn_qkvpacked_func(
    qkv,
    dropout_p=0.0,
    causal=True # 因果mask
)
```

Flash Attention收益:

指标	标准Attention	Flash Attention
显存复杂度	$O(n^2)$	$O(n)$
时间复杂度	$O(n^2)$	$O(n^2)$ (但更快)
实际加速	1×	2-4×
支持序列长度	2K-4K	8K-32K+

6.6.3 Flash Attention 2

改进:

1. 更好的并行化
2. 减少非矩阵乘法操作
3. 更优的分块策略

性能提升:

- 比Flash Attention 1快 2×
- 比标准attention快 5-9×

6.7 推理框架对比

6.7.1 主流推理框架

框架	开发者	特点	适用场景
vLLM	UC Berkeley	PagedAttention, 高吞吐	生产环境首选
TensorRT-LLM	NVIDIA	高度优化, NVIDIA专用	NVIDIA GPU
Text Generation Inference	HuggingFace	易用, 生态好	快速部署
llama.cpp	社区	CPU推理, 量化	CPU/边缘设备
DeepSpeed-Inference	Microsoft	ZeRO-Inference	超大模型

6.7.2 vLLM详解

核心创新：PagedAttention

类比操作系统的虚拟内存，将KV Cache分页管理。

```
# vLLM使用示例
from vllm import LLM, SamplingParams

# 初始化
llm = LLM(
    model="meta-llama/Llama-2-7b-hf",
    tensor_parallel_size=2, # 使用2个GPU
    dtype="float16"
)

# 推理
prompts = ["Hello, my name is", "The future of AI is"]
sampling_params = SamplingParams(temperature=0.8, top_p=0.95, max_tokens=100)

outputs = llm.generate(prompts, sampling_params)

for output in outputs:
    print(output.outputs[0].text)
```

性能对比：

吞吐量测试 (Llama-7B, A100):
HuggingFace: 100 tokens/s
Text Generation Inference: 300 tokens/s
vLLM: 800+ tokens/s (8倍提升!)

6.8 面试高频问题

Q1: 为什么量化可以工作?

答案要点：

- 1. **冗余性**：神经网络参数有冗余
- 2. **容错性**：小的精度损失不影响整体性能
- 3. **值分布**：大部分权重集中在小范围内

实验证据：

- INT8量化：精度损失<1%
- INT4量化：精度损失1-3%
- INT2：精度显著下降（不实用）

Q2: KV Cache和计算量的权衡？

生成第n个token时：

方法	计算量	显存占用
无Cache	$O(n^2)$	$O(1)$
有Cache	$O(n)$	$O(n)$

结论：

- **训练时**：不用KV Cache（batch内并行）
- **推理时**：必须用KV Cache（加速关键）

Q3: Flash Attention为什么快？

三个关键：

- 1. **IO优化**：减少HBM访问，利用SRAM
- 2. **分块计算**：避免存储巨大的attention矩阵
- 3. **融合操作**：将多个kernel融合

具体数据：

标准Attention HBM访问： $O(n^2 \times d)$
Flash Attention HBM访问： $O(n \times d)$

Q4: 如何选择量化方法？

场景	推荐方法	原因
推理加速	GPTQ/AWQ (INT4)	平衡精度和速度
显存受限	QLoRA (INT4)	极致压缩
追求精度	INT8 PTQ	精度损失最小
CPU推理	GGUF/Q4_K_M	llama.cpp优化

Q5: 如何评估压缩后的模型?

量化指标:

1. 精度:

- Perplexity (越低越好)
- 下游任务准确率
- 人工评估

2. 效率:

- 推理速度 (tokens/s)
- 显存占用 (GB)
- 延迟 (ms)

示例:

LLaMA-7B:

- FP16: 13GB显存, 100 tokens/s, PPL: 5.68
- INT8: 7GB显存, 180 tokens/s, PPL: 5.72 (+0.7%)
- INT4: 4GB显存, 250 tokens/s, PPL: 5.89 (+3.7%)

6.9 本章小结

本章深入讲解了大模型的高效训练与推理技术:

✅ **量化**: GPTQ和AWQ是主流INT4量化方案 ✅ **剪枝**: 结构化剪枝更实用 ✅ **知识蒸馏**: 让小模型学习大模型知识 ✅ **KV Cache**: 推理加速的关键, MQA/GQA减少显存 ✅ **Flash Attention**: 突破长序列的显存瓶颈 ✅ **推理框架**: vLLM是生产环境首选

实践建议:

1. 推理优化优先级: KV Cache > 量化 > Flash Attention
2. 量化首选GPTQ或AWQ的INT4
3. 长文本任务必须使用Flash Attention
4. 生产部署推荐vLLM

下一章预告: 第7章将进入工程实践篇, 讲解Prompt Engineering。