

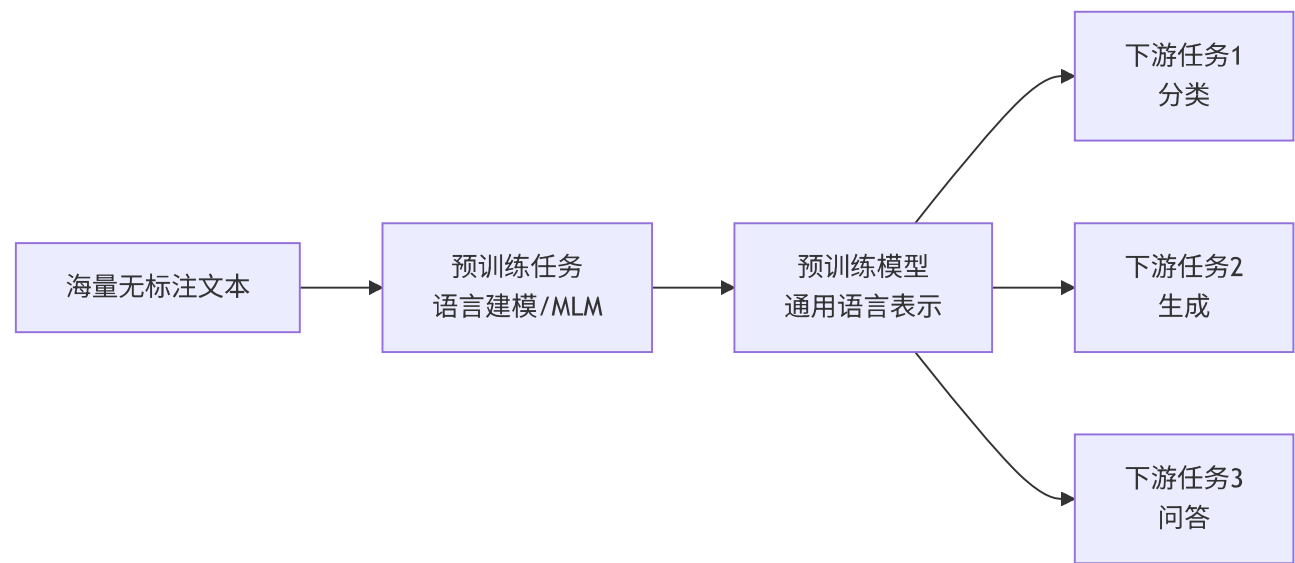
# 第4章 预训练技术

预训练是大模型能力的基础，数据、任务、策略缺一不可。

## 4.1 预训练概述

### 4.1.1 什么是预训练？

**定义：** 在大规模无标注数据上，通过自监督学习任务训练模型，使其学习通用的语言表示。

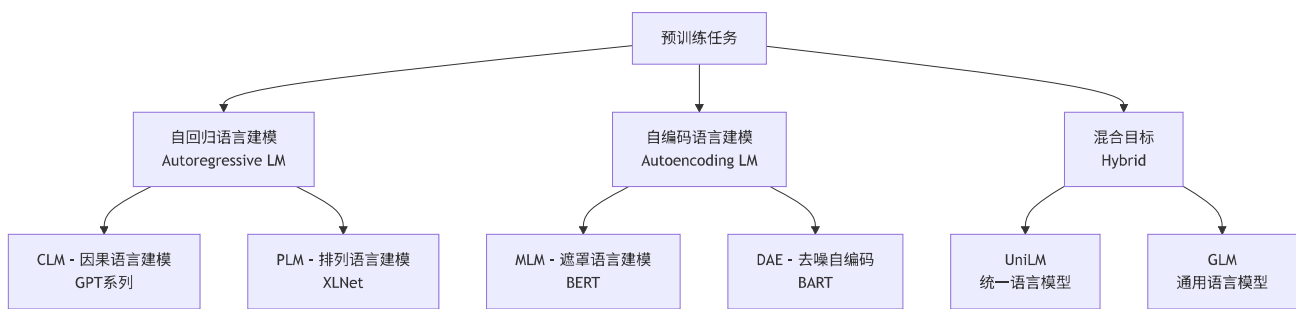


### 4.1.2 为什么需要预训练？

传统方法	预训练方法
随机初始化参数	从预训练权重开始
需要大量标注数据	少量标注数据即可
任务特定训练	迁移学习
训练时间长	微调快速
泛化能力有限	泛化能力强

## 4.2 预训练任务设计

### 4.2.1 主流预训练任务



## 4.2.2 Causal Language Modeling (CLM)

**任务定义：** 给定前面的token，预测下一个token。

输入："我 爱"

目标：预测 "中国"

输入："我 爱 中国"

目标：预测 <EOS>

**数学表达：**

$$\mathcal{L}_{CLM} = - \sum_{t=1}^T \log P(x_t | x_{<t}; \theta)$$

**代码实现：**

```

class CausalLanguageModelingLoss(nn.Module):
    def __init__(self):
        super().__init__()
        self.loss_fn = nn.CrossEntropyLoss(ignore_index=-100)

    def forward(self, logits, labels):
        """
        Args:
            logits: (batch_size, seq_len, vocab_size)
            labels: (batch_size, seq_len)
        """
        # Shift logits and labels for next token prediction
        shift_logits = logits[..., :-1, :].contiguous()
        shift_labels = labels[..., 1:].contiguous()

        # Flatten the tokens
        loss = self.loss_fn(
            shift_logits.view(-1, shift_logits.size(-1)),
            shift_labels.view(-1)
        )




```

```
return loss
```


# 使用示例

```
def train_step(model, batch):  
    input_ids = batch['input_ids'] # (B, L)  
  
    # 前向传播  
    logits = model(input_ids) # (B, L, V)  
  
    # 计算损失  
    loss_fn = CausalLanguageModelingLoss()  
    loss = loss_fn(logits, input_ids)  
  
    # 反向传播  
    loss.backward()  
  
    return loss.item()
```

**优点:**

-  简单高效
-  训练和推理一致
-  适合生成任务

**缺点:**

-  只能看到单向上下文

#### 4.2.3 Masked Language Modeling (MLM)

**任务定义:** 随机遮罩部分token, 预测被遮罩的token。

原始: "我 爱 吃 苹 果"

遮罩: "我 [MASK] 吃 [MASK] 果"

目标: 预测 "爱" 和 "苹"

**遮罩策略:**

```
def create_mlm_mask(tokens, mask_prob=0.15, vocab_size=50000):  
    """  
    BERT的MLM遮罩策略  
  
    Args:  
        tokens: 输入token序列  
        mask_prob: 遮罩概率 (默认15%)
```

Returns:

masked\_tokens: 遮罩后的序列

labels: 预测目标

"""

```
tokens = tokens.clone()
```

```
labels = tokens.clone()
```

# 创建遮罩概率矩阵

```
probability_matrix = torch.full(tokens.shape, mask_prob)
```

# 特殊token不遮罩

```
special_tokens_mask = torch.isin(tokens, torch.tensor([0, 101, 102])) # [PAD],  
probability_matrix.masked_fill_(special_tokens_mask, value=0.0)
```

# 决定哪些位置要遮罩

```
masked_indices = torch.bernoulli(probability_matrix).bool()
```

# 未被遮罩的位置, label设为-100 (不计算损失)

```
labels[~masked_indices] = -100
```

# BERT的三种处理方式

```
indices_replaced = torch.bernoulli(torch.full(tokens.shape, 0.8)).bool() & mask  
tokens[indices_replaced] = 103 # [MASK] token id
```

```
indices_random = torch.bernoulli(torch.full(tokens.shape, 0.5)).bool() & masked  
random_words = torch.randint(vocab_size, tokens.shape, dtype=torch.long)  
tokens[indices_random] = random_words[indices_random]
```

# 剩余10%保持不变

```
return tokens, labels
```

## BERT的MLM策略总结:

被选中的15%token	处理方式	比例	目的
80%	替换为[MASK]	12%	主要训练目标
10%	替换为随机token	1.5%	提高鲁棒性
10%	保持不变	1.5%	弥补预训练和微调的gap

## 优点:

- ✓ 双向上下文
- ✓ 适合理解任务

## 缺点:

- ❌ 预训练和微调不一致 ([MASK]只在预训练出现)
- ❌ 假设被遮罩token之间独立 (实际可能有依赖)

#### 4.2.4 Span Masking

**改进MLM：** 遮罩连续的span而不是单个token。

原始: "我 爱 吃 苹 果"

Span Mask: "我 [MASK] [MASK] 果"

目标: 预测 "爱 吃 苹"

#### T5的Span Corruption:

```
def span_corruption(tokens, corruption_rate=0.15, mean_span_length=3):
    """
    T5的Span Corruption策略

    Args:
        tokens: 输入序列
        corruption_rate: 被破坏的token比例
        mean_span_length: 平均span长度
    """
    seq_len = len(tokens)
    num_corrupted = int(seq_len * corruption_rate)

    # 计算span数量
    num_spans = max(1, num_corrupted // mean_span_length)

    # 随机选择span起始位置
    span_starts = random.sample(range(seq_len - mean_span_length), num_spans)

    masked_tokens = []
    sentinel_id = 32000 # <extra_id_0>

    for i, token in enumerate(tokens):
        if any(start <= i < start + mean_span_length for start in span_starts):
            if i in span_starts:
                masked_tokens.append(sentinel_id)
                sentinel_id += 1
            else:
                masked_tokens.append(token)

    return masked_tokens
```

**示例：**

输入: "Thank you for inviting me to your party last week."

T5 Span Corruption:

输入: "Thank you <X> inviting me to your party <Y> week."

目标: "<X> for <Y> last <Z>"

## 4.2.5 其他预训练任务

### Next Sentence Prediction (NSP):

句子A: "我喜欢吃苹果。"

句子B: "它们很好吃。"

标签: IsNext (1)

句子A: "我喜欢吃苹果。"

句子C: "今天天气不错。"

标签: NotNext (0)

**问题:** 研究发现NSP过于简单, RoBERTa移除后性能提升。

### Sentence Order Prediction (SOP):

ALBERT提出, 判断两个句子的顺序是否正确。

正例: [句子A] [句子B] # 原始顺序

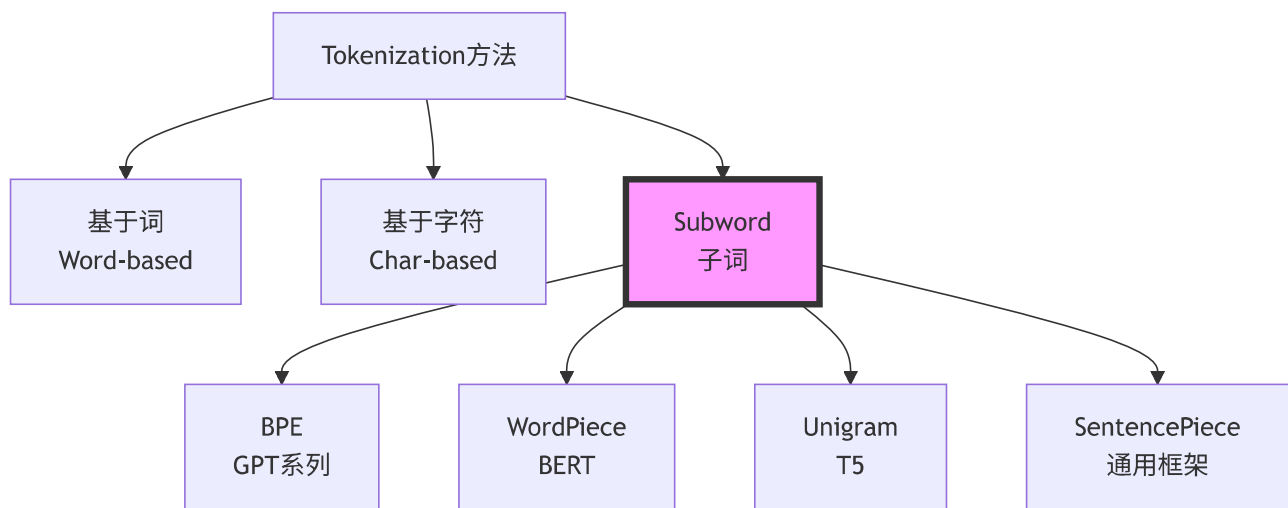
负例: [句子B] [句子A] # 交换顺序

比NSP更难, 效果更好。

## 4.3 Tokenization

Tokenization决定了模型如何"看"文本, 是预训练的第一步。

### 4.3.1 Tokenization方法对比



方法	优点	缺点	代表模型
Word-based	语义完整	词表巨大、OOV问题	传统NLP
Char-based	词表小、无OOV	序列长、语义弱	CharCNN
Subword	平衡词表和语义	需要训练	所有现代LLM

### 4.3.2 BPE (Byte Pair Encoding)

**核心思想：** 从字符开始，迭代合并最频繁连续字符对。

**算法流程：**

```
def train_bpe(corpus, num_merges):  
    """  
    训练BPE tokenizer  
  
    Args:  
        corpus: 训练文本列表  
        num_merges: 合并次数（决定词表大小）  
    """  
    # 1. 初始化: 每个字符是一个token  
    vocab = set()  
    word_freqs = {}  
  
    for text in corpus:  
        words = text.split()  
        for word in words:  
            word_freqs[word] = word_freqs.get(word, 0) + 1  
            vocab.update(word + '</w>') # 添加词尾标记  
  
    # 2. 迭代合并  
    for i in range(num_merges):  
        # 统计所有相邻pair的频率  
        pairs = {}
```

```

for word, freq in word_freqs.items():
    symbols = word.split()
    for j in range(len(symbols) - 1):
        pair = (symbols[j], symbols[j+1])
        pairs[pair] = pairs.get(pair, 0) + freq

# 找到最频繁的pair
if not pairs:
    break
best_pair = max(pairs, key=pairs.get)

# 合并这个pair
vocab.add(''.join(best_pair))

# 更新word_freqs
new_word_freqs = {}
for word, freq in word_freqs.items():
    new_word = word.replace(' '.join(best_pair), ''.join(best_pair))
    new_word_freqs[new_word] = freq
word_freqs = new_word_freqs

return vocab

```

```

def bpe_tokenize(text, bpe_vocab):
    """
    使用BPE词表进行tokenization
    """
    words = text.split()
    tokens = []

    for word in words:
        word = ' '.join(word) + ' </w>'

        while True:
            # 找到词表中最长的subword
            pairs = [(word[i:i+2]) for i in range(len(word)-1)]
            if not pairs:
                break

            # 选择在词表中的pair进行合并
            bigram = min(pairs, key=lambda x: bpe_vocab.get(x, float('inf'))))

            if bigram not in bpe_vocab:
                break

```



```

# 合并
first, second = bigram
new_word = []
i = 0
while i < len(word):
    if i < len(word) - 1 and word[i:i+2] == bigram:
        new_word.append(first + second)
        i += 2
    else:
        new_word.append(word[i])
        i += 1
word = ' '.join(new_word)

tokens.extend(word.split())

return tokens

```

### 示例:

原始文本: "lower lowest"

步骤1: l o w e r </w> | l o w e s t </w>

步骤2: 统计pair频率 -> "lo" 最频繁

步骤3: l o w e r </w> | l o w e s t </w>

步骤4: "low" 最频繁

步骤5: l o w e r </w> | l o w e s t </w>

...

最终: ["low", "er</w>", "low", "est</w>"]

### GPT-2/GPT-3使用的改进BPE:

```

# 使用tiktoken (OpenAI的tokenizer)
import tiktoken

encoding = tiktoken.get_encoding("cl100k_base") # GPT-4

text = "Hello, world!"
tokens = encoding.encode(text)
print(tokens) # [9906, 11, 1917, 0]

decoded = encoding.decode(tokens)
print(decoded) # "Hello, world!"

```

### 4.3.3 WordPiece

**与BPE的区别：** 选择合并pair时，不是选频率最高的，而是选择能最大化语言模型likelihood的。

**得分公式：**

$$\text{score}(\text{pair}) = \frac{\text{freq}(\text{pair})}{\text{freq}(\text{first}) \times \text{freq}(\text{second})}$$

**BERT使用WordPiece：**

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

text = "unbelievable"
tokens = tokenizer.tokenize(text)
print(tokens)  # ['un', '##believable'] 或 ['un', '##bel', '##iev', '##able']

# ##表示这个subword不是词的开始
```

### 4.3.4 SentencePiece

**特点：**

- ☒ 语言无关（不依赖空格分词）
- ☒ 直接处理raw text
- ☒ 支持BPE和Unigram两种算法

**使用示例：**

```
import sentencepiece as spm

# 训练
spm.SentencePieceTrainer.train(
    input='corpus.txt',
    model_prefix='m',
    vocab_size=32000,
    model_type='bpe', # 或 'unigram'
    character_coverage=0.9995,
    pad_id=0,
    unk_id=1,
    bos_id=2,
    eos_id=3
)

# 加载和使用
```

```

sp = spm.SentencePieceProcessor()
sp.load('m.model')

text = "我爱自然语言处理"
tokens = sp.encode_as_pieces(text)
print(tokens)  # ['__我', '__爱', '__自然', '__语言', '__处理']

ids = sp.encode_as_ids(text)
print(ids)  # [234, 567, 890, 1234, 5678]

```

## LLaMA使用SentencePiece:

```

# LLaMA的tokenizer配置
vocab_size = 32000
model_type = 'bpe'
character_coverage = 0.9995

# 中文处理
text_zh = "今天天气很好"
tokens = tokenizer.encode(text_zh)
# 每个汉字通常被编码为独立token

```

### 4.3.5 Tokenization最佳实践

#### 词表大小选择:

模型	词表大小	考虑因素
BERT	30K	英文为主
GPT-2	50K	多语言
GPT-3	50K	同GPT-2
LLaMA	32K	平衡性能和效率
GPT-4	100K	支持更多语言和符号

#### 权衡:

- 词表过小: 序列过长, 效率低
- 词表过大: embedding层参数多, 训练慢

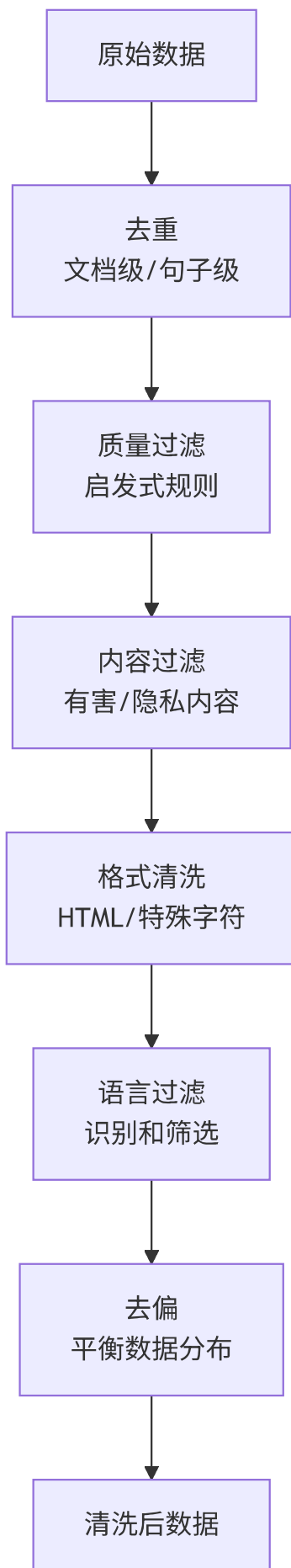
## 4.4 数据预处理与清洗

### 4.4.1 数据来源

#### 常见预训练数据集:

数据集	规模	语言	质量	使用模型
<b>Common Crawl</b>	~PB级	多语言	混杂	GPT-3, LLaMA
<b>WebText</b>	40GB	英文	较高	GPT-2
<b>C4</b>	750GB	英文	清洗后	T5
<b>Wikipedia</b>	~50GB	多语言	高	几乎所有模型
<b>BookCorpus</b>	5GB	英文	高	BERT, GPT
<b>The Pile</b>	825GB	英文	精选	GPT-Neo, GPT-J
<b>RedPajama</b>	1.2T tokens	多语言	开源复现	开源模型

#### 4.4.2 数据清洗Pipeline



详细步骤:

### 1. 去重 (Deduplication)

```

def deduplicate_corpus(documents):
    """
    文档级去重
    """
    seen_hashes = set()
    unique_docs = []

    for doc in documents:
        # 使用MinHash或SimHash进行近似去重
        doc_hash = hash_document(doc)
        if doc_hash not in seen_hashes:
            seen_hashes.add(doc_hash)
            unique_docs.append(doc)

    return unique_docs

def hash_document(doc, num_perm=128):
    """
    使用MinHash计算文档指纹
    """
    from datasketch import MinHash

    m = MinHash(num_perm=num_perm)
    for word in doc.split():
        m.update(word.encode('utf8'))

    return m.digest()

```

## 2. 质量过滤

```

def quality_filter(text):
    """
    启发式质量过滤
    """
    # 检查1: 长度
    if len(text) < 100 or len(text) > 100000:
        return False

    # 检查2: 平均词长 (过滤乱码)
    words = text.split()
    avg_word_len = sum(len(w) for w in words) / len(words)
    if avg_word_len < 3 or avg_word_len > 15:
        return False

    # 检查3: 符号比例

```

```

symbol_ratio = sum(c in '!@#$%^&*()' for c in text) / len(text)
if symbol_ratio > 0.1:
    return False

# 检查4: 重复n-gram比例
def has_repetition(text, n=3):
    ngrams = [text[i:i+n] for i in range(len(text)-n)]
    return len(set(ngrams)) / len(ngrams) < 0.5

if has_repetition(text):
    return False

# 检查5: 语言模型困惑度 (需要预训练的小模型)
# perplexity = calculate_perplexity(text, language_model)
# if perplexity > threshold:
#     return False

return True

```

### 3. 有害内容过滤

```

def filter_toxic_content(text):
    """
    过滤有害内容
    """
    # 方法1: 关键词黑名单
    toxic_keywords = load_toxic_keywords()
    if any(keyword in text.lower() for keyword in toxic_keywords):
        return False

    # 方法2: 使用分类器
    # toxicity_score = toxicity_classifier(text)
    # if toxicity_score > 0.7:
    #     return False

    # 方法3: 检测PII (个人身份信息)
    import re

    # 检测email
    if re.search(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text):
        return False

    # 检测电话号码
    if re.search(r'\b\d{3}[-.]?\d{3}[-.]?\d{4}\b', text):
        return False

```

```
return True
```

### 4.4.3 数据配比策略

不同来源的数据需要合理配比。

**LLaMA的数据配比：**

数据源	比例	Tokens	采样Epochs
CommonCrawl	67%	930B	1.0
C4	15%	200B	1.0
Github	4.5%	60B	1.0
Wikipedia	4.5%	60B	2.0
Books	4.5%	60B	2.0
ArXiv	2.5%	35B	1.5
StackExchange	2%	28B	1.0

**策略考虑：**

1. **质量vs数量**：高质量数据可以多采样
2. **任务相关性**：根据目标能力调整配比
3. **时效性**：新数据权重可以更高

### 4.4.4 数据增强

```
def data_augmentation(text):  
    """  
    预训练数据增强  
    """  
  
    augmented = []  
  
    # 1. 原始文本  
    augmented.append(text)  
  
    # 2. 回译 (Back-translation)  
    # translated = translate(text, src='en', tgt='de')  
    # back_translated = translate(translated, src='de', tgt='en')  
    # augmented.append(back_translated)  
  
    # 3. 同义词替换 (需谨慎，可能改变语义)  
    # augmented.append(synonym_replacement(text))
```



```
# 4. 文档拼接（模拟长文档）
# augmented.append(text + " " + random_document())

return augmented
```

## 4.5 训练策略与技巧

### 4.5.1 学习率调度

#### 大模型常用：Warmup + Cosine Decay

```
def get_lr_schedule(
    optimizer,
    num_warmup_steps,
    num_training_steps,
    min_lr_ratio=0.1
):
    """
    Warmup + Cosine Decay学习率调度

    Args:
        num_warmup_steps: warmup步数（通常是总步数的1-5%）
        num_training_steps: 总训练步数
        min_lr_ratio: 最小学习率占初始学习率的比例
    """
    def lr_lambda(current_step):
        if current_step < num_warmup_steps:
            # Warmup: 线性增长
            return float(current_step) / float(max(1, num_warmup_steps))

            # Cosine Decay
            progress = float(current_step - num_warmup_steps) / \
                float(max(1, num_training_steps - num_warmup_steps))
            cosine_decay = 0.5 * (1.0 + math.cos(math.pi * progress))

            # 确保不低于min_lr
            return max(min_lr_ratio, cosine_decay)

    from torch.optim.lr_scheduler import LambdaLR
    return LambdaLR(optimizer, lr_lambda)

# 使用示例
optimizer = torch.optim.AdamW(model.parameters(), lr=6e-4)
total_steps = 100000
```

```

warmup_steps = 2000

scheduler = get_lr_schedule(optimizer, warmup_steps, total_steps)

for epoch in range(num_epochs):
    for batch in dataloader:
        loss = train_step(model, batch)
        optimizer.step()
        scheduler.step() # 每步更新学习率
        optimizer.zero_grad()

```

#### 4.5.2 梯度裁剪

防止梯度爆炸，大模型训练必备。

```

# 全局梯度裁剪
max_grad_norm = 1.0
torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)

# 在优化器step之前
optimizer.step()

```

#### 4.5.3 混合精度训练

使用FP16或BF16加速训练。

```

from torch.cuda.amp import autocast, GradScaler

# 初始化
scaler = GradScaler()

for batch in dataloader:
    optimizer.zero_grad()

    # 前向传播使用autocast
    with autocast():
        outputs = model(batch['input_ids'])
        loss = loss_fn(outputs, batch['labels'])

    # 反向传播
    scaler.scale(loss).backward()

    # 梯度裁剪
    scaler.unscale_(optimizer)
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)

```

```
# 优化器step
scaler.step(optimizer)
scaler.update()
```

## FP16 vs BF16:

特性	FP16	BF16
动态范围	小（易溢出）	大（与FP32相同）
精度	高	较低
硬件支持	广泛（V100+）	较新（A100+）
训练稳定性	需要loss scaling	<b>更稳定</b>
大模型推荐		<b>BF16</b>

### 4.5.4 梯度累积

当GPU显存不足时，模拟大batch size。

```
accumulation_steps = 4 # 累积4步
effective_batch_size = batch_size * accumulation_steps

optimizer.zero_grad()

for i, batch in enumerate(dataloader):
    # 前向传播
    outputs = model(batch['input_ids'])
    loss = loss_fn(outputs, batch['labels'])

    # 损失缩放
    loss = loss / accumulation_steps

    # 反向传播
    loss.backward()

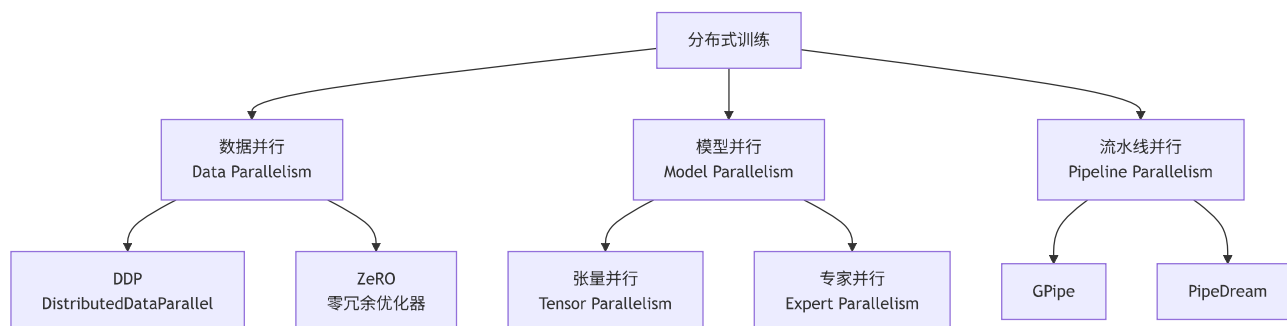
    # 每accumulation_steps步更新一次
    if (i + 1) % accumulation_steps == 0:
        # 梯度裁剪
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)

        # 优化器更新
        optimizer.step()
        scheduler.step()
        optimizer.zero_grad()
```

## 4.6 分布式训练

大模型训练必须使用分布式。

### 4.6.1 三种并行策略



### 4.6.2 数据并行 (DDP)

**原理：** 每个GPU持有完整模型，处理不同数据。

```
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

def setup(rank, world_size):
    """
    初始化进程组
    """
    dist.init_process_group(
        backend='nccl', # GPU使用nccl
        init_method='env://',
        rank=rank,
        world_size=world_size
    )

def train_ddp(rank, world_size):
    setup(rank, world_size)

    # 创建模型并移到对应GPU
    model = GPTModel(...).to(rank)
    model = DDP(model, device_ids=[rank])

    # 创建分布式采样器
    from torch.utils.data.distributed import DistributedSampler
    sampler = DistributedSampler(
        dataset,
        num_replicas=world_size,
        rank=rank
    )
```

```

dataloader = DataLoader(
    dataset,
    batch_size=batch_size,
    sampler=sampler
)

# 训练循环
for epoch in range(num_epochs):
    sampler.set_epoch(epoch) # 重要: 确保每个epoch数据不同

    for batch in dataloader:
        batch = {k: v.to(rank) for k, v in batch.items()}

        outputs = model(batch['input_ids'])
        loss = loss_fn(outputs, batch['labels'])

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

# 启动
import torch.multiprocessing as mp

world_size = torch.cuda.device_count()
mp.spawn(train_ddp, args=(world_size,), nprocs=world_size)

```

#### 4.6.3 ZeRO (零冗余优化器)

**DeepSpeed ZeRO** 是大模型训练的关键技术。

**ZeRO三个阶段:**

阶段	分片内容	内存节省	通信开销
ZeRO-1	优化器状态	4×	低
ZeRO-2	+梯度	8×	中
ZeRO-3	+模型参数	N×	高

```

# 使用DeepSpeed ZeRO
import deepspeed

# deepspeed配置
ds_config = {
    "train_batch_size": 32,

```

```

    "gradient_accumulation_steps": 1,
    "fp16": {
        "enabled": True
    },
    "zero_optimization": {
        "stage": 2, # ZeRO-2
        "offload_optimizer": {
            "device": "cpu" # 将优化器状态offLoad到CPU
        }
    }
}

```

```

model_engine, optimizer, _, _ = deepspeed.initialize(
    model=model,
    config=ds_config,
    model_parameters=model.parameters()
)

```

# 训练

```

for batch in dataloader:
    loss = model_engine(batch)
    model_engine.backward(loss)
    model_engine.step()

```

#### 4.6.4 模型并行

##### 张量并行 (Megatron-LM) :

将单个层的参数分割到多个GPU。

# 简化示例: 将`Linear`层分割

```

class ColumnParallelLinear(nn.Module):
    """
    将输出维度切分到多个GPU
    """
    def __init__(self, in_features, out_features, world_size):
        super().__init__()
        assert out_features % world_size == 0

        self.out_features_per_partition = out_features // world_size
        self.weight = nn.Parameter(
            torch.randn(self.out_features_per_partition, in_features)
        )

    def forward(self, x):
        # 本地计算

```

```

output_parallel = F.linear(x, self.weight)

# ALLGather收集所有GPU的输出
output = all_gather(output_parallel)

return output

```

## 4.7 面试高频问题

Q1: 为什么需要Warmup?

答案要点:

1. **初始阶段不稳定**: 参数随机初始化, 梯度方差大
2. **Adam的问题**: Adam的二阶矩估计在初期不准确
3. **大学习率风险**: 可能导致梯度爆炸或损失发散

数学解释:

Adam的偏差校正:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

初期 $\beta_1^t$ 和 $\beta_2^t$ 接近1, 校正后的值偏大。

Q2: BPE和WordPiece的区别?

维度	BPE	WordPiece
合并策略	选择频率最高的pair	选择最大化likelihood的pair
得分函数	$\text{freq}(\text{pair})$	$\text{freq}(\text{pair}) / (\text{freq}(a) * \text{freq}(b))$
训练速度	快	较慢
理论基础	贪心压缩	语言模型概率
使用模型	GPT系列	BERT

Q3: 如何选择预训练任务?

任务	适用场景	模型
CLM	生成任务、通用模型	GPT系列
MLM	理解任务、需要双向上下文	BERT
Span Corruption	Seq2seq任务	T5
混合	希望同时具备理解和生成能力	GLM, UniLM

现代趋势: CLM占主导, 因为:

- 统一的训练和推理范式
- 更好的扩展性
- In-context learning能力

Q4: 数据去重为什么重要?

**原因:**

1. **防止过拟合**: 重复数据导致模型记忆而非理解
2. **提高效率**: 避免浪费计算资源
3. **减少偏见**: 某些内容重复出现会放大偏见

**实际案例:**

- GPT-3训练数据中发现大量重复
- LLaMA专门进行了文档级和句子级去重
- 去重后模型泛化性能提升

Q5: 如何估算预训练所需的计算量?

**公式 (Kaplan et al.) :**

$$C \approx 6 \times N \times D$$

**其中:**

- C: 计算量 (FLOPs)
- N: 模型参数量
- D: 训练token数

**示例:**

LLaMA-7B:

- $N = 7B$
- $D = 1.4T \text{ tokens}$
- $C \approx 6 \times 7B \times 1.4T \approx 60E21 \text{ FLOPs} = 60 \text{ ZFLOPs}$

使用A100 (312 TFLOPS): 时间  $\approx 60E21 / 312E12 / 3600 / 24 \approx 22,000 \text{ GPU天}$

## 4.8 本章小结

本章深入讲解了预训练技术的各个方面:

✅ **预训练任务**: CLM是现代主流, MLM适理解任务 ✅ **Tokenization**: BPE/WordPiece/SentencePiece各有特点  
 ✅ **数据处理**: 清洗、去重、过滤是关键 ✅ **训练策略**: Warmup、混合精度、梯度裁剪  
 ✅ **分布式训练**: ZeRO是大模型训练的必备技术

**实践要点:**

1. 数据质量 > 数量
2. 合理的学习率调度至关重要



3. 监控训练指标 (loss、梯度norm、学习率)

4. 定期checkpoint和评估

---

**下一章预告：** 第5章将讲解微调与对齐技术，包括RLHF、DPO等。