2 Transformer 训练与推理

2.1 Transformer 预训练流程

基于 Transformer 架构，我们可以设计多种预训练任务来训练不同类型的语言模型:

A. Encoder-Only 模型(如 BERT):

- 仅使用 Transformer 的编码器部分
- 主要采用"掩词补全"(Masked Language Modeling)等自监督任务
- 例如:
    - 输入: "今天天气很好,我准备去公园散步。"
    - 掩词: "今天[MASK]很好,我准备去公园散步。"
    - 模型需要独立预测: "天气"

B. Encoder-Decoder 模型(如 T5):

- 同时使用编码器和解码器
- 结合"文本截断补全(Text Truncation)"、"句子顺序恢复(Sentence Ordering)"等多个监督和自监督任务
- 例如
    - 文本截断补全(Text Truncation):
        - 输入: "今天天气很好,我准备去公园散步。"
        - 截断之后: "今天天气很好,我准"
        - 模型需要补全的目标: "备去公园散步。"
    - 句子顺序恢复(Sentence Ordering):
        - 打乱顺序的输入:
            - "他开心地笑了。"
            - "小明收到了一份礼物。"
            - "今天是小明的生日。"
        - 模型需要恢复正确顺序:
            1. "今天是小明的生日。"
            2. "小明收到了一份礼物。"
            3. "他开心地笑了。"

C. Decoder-Only 模型(如 GPT-3):

- 仅使用解码器部分
- 采用"下一词预测(Next Word Prediction)"任务

Next Word Prediction

在目前主流的LLM模型中，最常用的预训练任务是"下一词预测"任务。本文也以"下一词预测"任务为例，介绍 Transformer 语言模型的基本训练流程。

Next Word Prediction 任务，是对词序列 $\{w_1, w_2, w_3, \ldots, w_N\}$，基于 Transformer 的语言模型通过已知词序列 $\{w_1, w_2, \ldots, w_i\}$ 来预测下一个词 w_{i+1} 的概率。

模型的输出是一个概率向量 o_i，其维度等于词典大小 $|D|$:

- 词典 D 包含 $|D|$ 个词: $\{\hat{w}_1, \hat{w}_2, \ldots, \hat{w}_{|D|}\}$
- 对于词典中的每个词 $\hat{w}_d$，向量 o_i 的第 d 维 $o_i\left[\hat{w}_d\right]$ 表示在位置 i 处预测该词的概率
- 即 $o_i = \{o_i\left[\hat{w}_1\right], o_i\left[\hat{w}_2\right], \ldots, o_i\left[\hat{w}_{|D|}\right]\}$

因此，模型对整个词序列的概率预测为：

$$P\left(w_{1:N}\right) = \prod_{i=1}^{N-1} P\left(w_{i+1} \mid w_{1:i}\right) = \prod_{i=1}^{N} o_i\left[w_{i+1}\right] \tag{1}$$

与 RNN 语言模型类似，Transformer 使用交叉熵(Cross Entropy)作为损失函数：

$$l_{CE}\left(o_i\right) = -\sum_{d=1}^{|D|} I\left(\hat{w}_d = w_{i+1}\right) \log o_i\left[w_{i+1}\right] = -\log o_i\left[w_{i+1}\right] \tag{2}$$

其中指示函数 $I(\cdot)$ 在 $\hat{w}_d = w_{i+1}$ 时为1，否则为0。

> 交叉熵损失函数(Cross Entropy Loss)，又称对数损失函数(Logarithmic Loss)，是衡量两个概率分布之间差异的常用指标。
> 对于两个概率分布 P 和 Q，它们的交叉熵定义为：
> $H(P, Q) = -\sum_{x \in X} P(x) \log Q(x)$
> 其中，X 是所有可能的事件集合，$P(x)$ 是事件 x 的真实概率，$Q(x)$ 是事件 x 的预测概率。
>
> 这里把交叉熵公式简化了，因为：
>
> 1. 真实分布 P 是一个 one-hot 向量（只有正确词的位置是1，其他都是0）
> 2. 预测分布 Q 就是模型输出的概率向量 o_i

结合交叉熵损失函数，对训练集 S，总体损失函数为：

$$L(S, W) = \frac{1}{N|S|} \sum_{s=1}^{|S|} \sum_{i=1}^{N} l_{CE}\left(o_{i,s}\right) \tag{3}$$

其中:

- $o_{i,s}$ 是模型对训练样本 s 中前 i 个词的输出概率向量
- N 是每个训练样本的词序列长度(在实际训练中，通常采用批处理的方式，每个批次包含多个训练样本，每个样本的词序列长度可能不同，所以会使用填充(Padding)的方式，将所有样本的词序列长度统一为最大长度N)
- $|S|$ 是训练集中样本的总数量

基于此损失函数构建计算图并反向传播来训练模型。

Next Word Prediction 简化版示例代码

下面是一个使用 Next Word Prediction 训练 Transformer 模型的示例代码。

其中，train_epoch函数是训练一个epoch的函数，而train_step函数是训练一步的函数。

```
def train_step(model, optimizer, input_ids, target_ids, criterion):
    """训练一步

    Args:
        model: Transformer模型
        optimizer: 优化器
        input_ids: 输入序列的token ids, shape=[batch_size, seq_len]
        target_ids: 目标序列的token ids, shape=[batch_size, seq_len]
        criterion: 损失函数(通常是CrossEntropyLoss)

    Returns:
        loss: 当前步的损失值
    """
    # 将模型设为训练模式
    model.train()

    # 清空梯度
    optimizer.zero_grad()

    # 前向传播,得到模型输出
    # outputs shape: [batch_size, seq_len, vocab_size]
    outputs = model(input_ids)

    # 计算损失
    # 将outputs重塑为[batch_size * seq_len, vocab_size]
    # target_ids重塑为[batch_size * seq_len]
    loss = criterion(
        outputs.view(-1, outputs.size(-1)),
        target_ids.view(-1)
    )

    # 反向传播
    loss.backward()

    # 更新参数
    optimizer.step()

    return loss.item()

def train_epoch(model, train_dataloader, optimizer, criterion, device):
    """训练一个epoch

    Args:
```

```
        model: Transformer模型
        train_dataloader: 训练数据加载器
        optimizer: 优化器
        criterion: 损失函数
        device: 训练设备(CPU/GPU)

    Returns:
        epoch_loss: 当前epoch的平均损失
    """
    model.train()
    total_loss = 0

    # 遍历训练数据
    for batch in train_dataloader:
        # 将数据移到指定设备
        input_ids = batch['input_ids'].to(device)
        target_ids = batch['target_ids'].to(device)

        # 训练一步
        loss = train_step(
            model=model,
            optimizer=optimizer,
            input_ids=input_ids,
            target_ids=target_ids,
            criterion=criterion
        )

        total_loss += loss

    # 计算平均损失
    avg_loss = total_loss / len(train_dataloader)

    return avg_loss
```

Teacher Forcing

Next Word Prediction训练时，使用"自回归"的方式来完成生成。模型会逐步生成文本：

1. 首先将第一个词`input_1`输入给模型，经过解码得到输出词`output_1`
2. 然后将`input_1`和`output_1`拼接作为下一轮输入
3. 继续解码得到`output_2`
4. 如此迭代直到生成完整文本

但如果使用自回归生成方式来训练，会存在两个主要问题：

1. 错误累积：如果模型在某一步生成了错误的词，这个错误会作为下一步的输入，导致后续生成的内容质量持续下降
2. 计算效率低：由于每一步的预测都依赖于前一步的输出，无法并行计算，降低了训练和推理速度

3. 训练不稳定：在自回归生成过程中，如果模型在早期预测错误，模型下一步的预测是基于错误的上下文，导致训练偏离原本的目标

为了解决这些问题，在训练过程中普遍采用"Teacher Forcing"技术，通过并行计算来提高训练效率。以"今天天气很好"这句话为例：
在 Teacher Forcing 中，每一步训练时都使用真实的目标序列作为输入,而不是使用模型生成的输出:

1. 第一步输入"今天",预测下一个词"天"
2. 第二步输入"今天天",预测下一个词"气"
3. 第三步输入"今天天气",预测下一个词"很"
4. 第四步输入"今天天气很",预测下一个词"好"

这4步可以并行计算，而不是像自回归那样一步一步生成。

这种方式可以:

- 避免错误累积,因为每一步都使用正确的上下文
- 支持并行计算,提高训练效率
- 保持训练的稳定性

当然，Teacher Forcing 也不是万能的，它有曝光偏差（Exposure Bias）问题。这个问题源于训练和推理过程的不一致性：

- 训练时：模型在每一步都能获得真实的上文（标准答案）作为输入
- 推理时：模型只能使用自己生成的内容作为上文，无法获得标准答案

这种训练-推理的不一致性会导致模型在实际应用中表现不佳。为了缓解这个问题，Bengio等人提出了 Scheduled Sampling 方法。该方法的核心思想是在训练过程中逐步引入模型自己生成的内容作为输入，通过调度策略，gradually增加使用模型生成内容的比例，帮助模型在训练阶段就开始适应推理时的工作模式，从而提升最终的生成质量。

Teacher Forcing 简化版示例代码

下面是一个使用 Teacher Forcing 前处理数据的示例代码。

```
def prepare_training_data(text, tokenizer, max_length):
    """准备训练数据,将文本处理成teacher forcing格式

    Args:
        text: 原始文本,如"Ming had a new laptop"
        tokenizer: 分词器
        max_length: 最大序列长度

    Returns:
        input_ids: 输入序列列表,如:
                   ["Ming"]
                   ["Ming", "had"]
                   ["Ming", "had", "a"]
                   ["Ming", "had", "a", "new"]
                   ...
```

```
        target_ids: 对应的目标token
    """
    # 对文本分词
    tokens = tokenizer.encode(text)

    # 如果文本长度超过最大长度,截断
    if len(tokens) > max_length:
        tokens = tokens[:max_length]
    # 初始化输入序列和目标序列
    input_sequences = []
    target_sequences = []

    # 生成teacher forcing训练数据，从1开始，因为第一个词是输入，不需要预测
    for i in range(1, len(tokens)):
        # 获取输入序列和目标序列
        input_seq = tokens[:i]
        target = tokens[i]

        # padding
        padding_length = max_length - len(input_seq)
        # 如果padding长度大于0，则进行padding
        if padding_length > 0:
            input_seq = input_seq + [tokenizer.pad_token_id] * padding_length

        # 添加到输入序列和目标序列中
        input_sequences.append(input_seq)
        target_sequences.append(target)

    return input_sequences, target_sequences

# 示例：TextDataset在返回数据时，会调用prepare_training_data函数，将文本处理成teacher forcing格
式
class TextDataset(Dataset):
    """文本数据集"""
    def __init__(self, texts, tokenizer, max_length):
        self.texts = texts
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        input_ids, target_ids = prepare_training_data(text, self.tokenizer,
self.max_length)
        return {
            'input_ids': torch.tensor(input_ids),
            'target_ids': torch.tensor(target_ids)
```

```
        }
```

2.2 Transformer 推理流程

在推理过程中，Transformer 模型采用自回归的方式生成文本。模型会将第一个词输入并解码得到输出词，然后将输入和输出拼接作为下一轮输入，如此迭代生成完整文本。

前面内容提到过，LLM的输出是一个向量，每一维代表词典中对应词的概率。在自回归文本生成任务中，LLM会依次生成一组向量并将其解码为文本。这个解码过程显著影响生成文本的质量。当前主流的解码方法可分为概率最大化方法(Greedy Search和Beam Search)和随机采样方法(Top-K Sampling和Top-P Sampling)。

概率最大化方法

设：

- 词典为 D
- 输入文本为 $\{w_1, w_2, w_3, \ldots, w_N\}$
- 第 i 轮自回归中输出的向量为 $o_i = \{o_i[w_d]\}_{d=1}^{|D|}$
- 模型在 M 轮自回归后生成的文本为 $\{w_{N+1}, w_{N+2}, w_{N+3}, \ldots, w_{N+M}\}$

生成文档的出现的概率可由下式进行计算。

$$P\left(w_{N+1:N+M}\right) = \prod_{i=N}^{N+M-1} P\left(w_{i+1} \mid w_{1:i}\right) = \prod_{i=N}^{N+M-1} o_i\left[w_{i+1}\right] \tag{4}$$

基于概率最大化的解码方法旨在最大化 $P\left(w_{N+1:N+M}\right)$ 以生成可能性最高的文本。这是一个搜索空间为 M^D 的 NP-Hard 问题，通常采用启发式搜索方法求解。

最常用的概率最大化方法有：

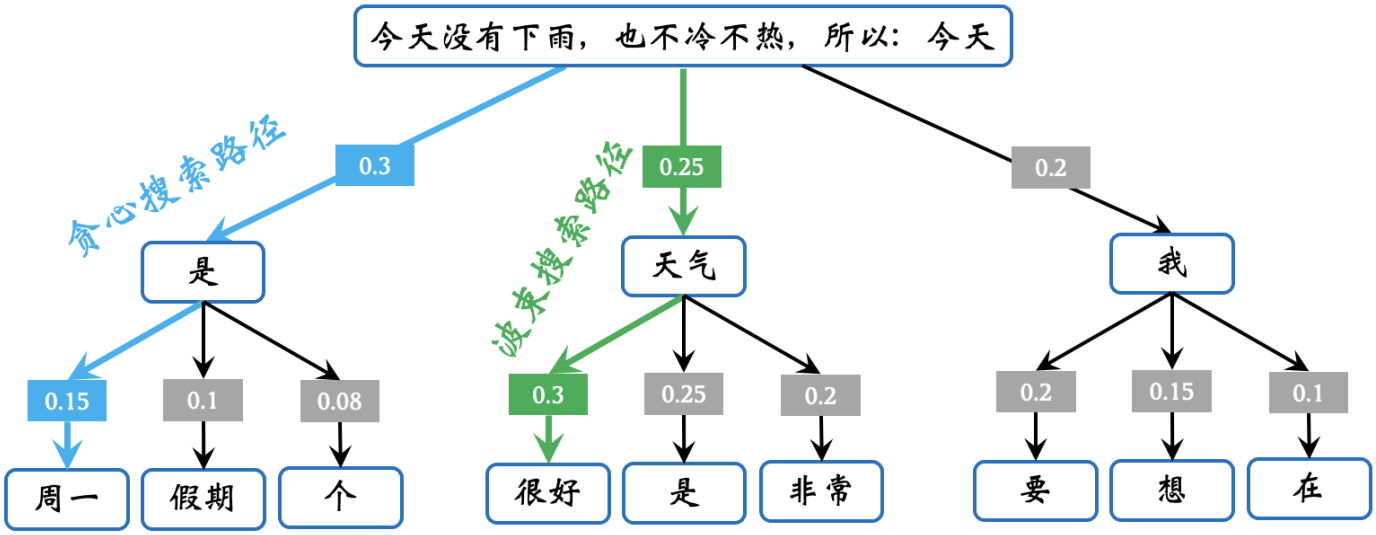
- 贪心搜索
- 波束搜索

贪心搜索

贪心搜索是最简单的方法，每轮都选择概率最大的词：

$$w_{i+1} = \arg\max_{w \in D} o_i[w] \tag{5}$$

然而贪心搜索容易陷入局部最优，例如下图中，输入"今天没有下雨，也不冷不热，所以：今天"，贪心搜索会陷入局部最优，生成"今天是周一"，而更优的结果"今天天气真好"，需要在第一步选择次优概率的"天气"，要实现这种效果，需要模型尝试更多路径，这就涉及到另一种解码方法，波束搜索。



波束搜索

波束搜索通过保留多个候选来缓解这个问题。每轮保留 b 个最可能的词 $B_i = \left\{w_{i+1}^1, w_{i+1}^2, \ldots, w_{i+1}^b\right\}$，满足：

$$\min\left\{o_i[w] \text{ for } w \in B_i\right\} > \max\left\{o_i[w] \text{ for } w \in D - B_i\right\}。 \tag{6}$$

搜索结束时从 M 个候选集 $\{B_i\}_{i=1}^M$ 中找出最优组合：

$$\left\{w_{N+1}, \ldots, w_{N+M}\right\} = \arg\max_{\left\{w^i \in B_i\right.} \text{ for } 1 \le i \le M \Bigg\} \prod_{i=1}^{M} o_{N+i}\left[w^i\right] \tag{7}$$

这个公式表示波束搜索中每轮保留的候选集 B_i 中的词的概率都要大于非候选集中词的概率。

在上图的例子中，使用波束搜索方法，可以搜索到

通过比较整体概率，波束搜索可以选择到更优的"今天天气真好"这个结果。

但概率最大化方法往往生成常见但平庸的文本。在开放式生成中容易产生重复且缺乏新意的"废话文学"。为提升生成文本的新颖度，可以在解码过程中引入随机性，这就是随机采样方法。

随机采样方法

随机采样通过在预测时增加随机性来提升文本多样性。每轮预测先选出高概率的候选词，然后按概率分布进行采样。主流的 Top-K 采样和 Top-P 采样分别通过指定候选词数量和概率阈值来选择候选词。这两个方法和 Temperature 机制结合，可以调整候选词的概率分布。

Top-K 采样

Top-K 采样每轮选取 K 个最高概率的词作为候选，用 softmax 归一化后得到分布：

$$p\left(w_{i+1}^1, \ldots, w_{i+1}^K\right) = \left\{ \frac{\exp\left(o_i\left[w_{i+1}^1\right]\right)}{\sum_{j=1}^K \exp\left(o_i\left[w_{i+1}^j\right]\right)}, \ldots, \frac{\exp\left(o_i\left[w_{i+1}^K\right]\right)}{\sum_{j=1}^K \exp\left(o_i\left[w_{i+1}^j\right]\right)} \right\} \tag{8}$$

然后从该分布中采样：

$$w_{i+1} \sim p\left(w_{i+1}^1, \ldots, w_{i+1}^K\right) \tag{9}$$

Top-K 采样简化版示例代码：

```
def top_k_sampling(probs, k):
    """Top-K 采样

    Args:
        probs: 概率分布 (torch.Tensor)
        k: 候选词数量
    """
    # 获取前k个最大概率的值和索引
    top_k_probs, top_k_idx = torch.topk(probs, k)

    # 归一化概率
    top_k_probs = torch.softmax(top_k_probs, dim=-1)

    # 从top k中采样
    next_word_idx = torch.multinomial(top_k_probs, num_samples=1)

    # 获取实际的词表索引
    next_word_idx = top_k_idx[next_word_idx]

    return next_word_idx.item()
```

Top-K 采样虽然可以增加生成文本的新颖度，但由于使用固定大小的候选集，在不同预测轮次中可能会出现一些问题。当候选词的概率分布方差较大时，固定大小的候选集可能会包含一些概率很低的词，这些低概率词如果被选中，会导致生成不符合语境、不通顺的文本。而当候选词的概率分布方差较小，多个词的概率比较接近时，固定大小的候选集可能会排除掉一些概率相近的合理候选词，使得生成的文本缺乏变化，显得单调重复。为了解决上述问题，我们需要一种能够根据概率分布动态调整候选集大小的方法，这就是 Top-P 采样(又称 Nucleus 采样)。

Top-P 采样

Top-P 采样(Nucleus采样)通过设定阈值 p 来动态选取候选集 $S_p = \left\{w_{i+1}^1, w_{i+1}^2, \ldots, w_{i+1}^{|S_p|}\right\}$，满足 $\sum_{w \in S_p} o_i[w] \geq p$。

候选集分布为：

$$p\left(w_{i+1}^1, \ldots, w_{i+1}^{|S_p|}\right) = \left\{ \frac{\exp\left(o_i\left[w_{i+1}^1\right]\right)}{\sum_{j=1}^{|S_p|} \exp\left(o_i\left[w_{i+1}^j\right]\right)}, \ldots, \frac{\exp\left(o_i\left[w_{i+1}^{|S_p|}\right]\right)}{\sum_{j=1}^{|S_p|} \exp\left(o_i\left[w_{i+1}^j\right]\right)} \right\} \tag{10}$$

采样方式为：

$$w_{i+1} \sim p\left(w_{i+1}^1, \ldots, w_{i+1}^{|S_p|}\right) \tag{11}$$

这种方法可以避免选到低概率的不合理词，同时保持足够的候选词丰富度。
**
具体的筛选流程如下:**

Step 1. 首先对词表中所有词按概率从大到小排序:
$w_1, w_2, \ldots, w_n$ 其中 $p(w_1) \geq p(w_2) \geq \ldots \geq p(w_n)$

Step 2. 从高到低累加概率,直到超过阈值p:
$\sum_{i=1}^{k} p(w_i) \geq p$
此时得到候选集 $S_p = \{w_1, w_2, \ldots, w_k\}$

Step 3. 对候选集中的概率重新归一化:
$p'(w_i) = \frac{p(w_i)}{\sum_{w \in S_p} p(w)}$

Step 4. 从归一化后的分布中采样下一个词:
$w_{next} \sim p'(w)$

例如,假设词表概率分布为:

词	概率	累积概率
的	0.3	0.3
是	0.2	0.5
在	0.15	0.65
了	0.1	0.75
...	...	...

如果设置p=0.7,则候选集为{"的","是","在","了"},归一化后的概率分布为:
$p'(\text{的}) = 0.3/0.7, p'(\text{是}) = 0.2/0.7, p'(\text{在}) = 0.15/0.7, p'(\text{了}) = 0.1/0.7$

Top-P 采样简化版示例代码:

```
def top_p_sampling(probs, p):
    """Top-P 采样

    Args:
        probs: 概率分布 (torch.Tensor)
        p: 阈值
    """
    # 对概率分布进行降序排序
    sorted_probs, sorted_indices = torch.sort(probs, descending=True)

    # 计算累积概率
    cumulative_probs = torch.cumsum(sorted_probs, dim=0)

    # 找到累积概率大于等于p的词的索引
    # nonzero返回满足条件的索引
    # squeeze去掉维度为1的维度
    p_idx = torch.nonzero(cumulative_probs >= p)[0].item()

    # 获取前p_idx+1个词作为候选集
```

```
    candidate_probs = sorted_probs[:p_idx+1]

    # 归一化概率
    candidate_probs = candidate_probs / candidate_probs.sum()

    # 从候选集中采样
    # multinomial进行多项分布采样
    next_word_idx = torch.multinomial(candidate_probs, num_samples=1).item()

    # 返回原始词表中的索引
    return sorted_indices[next_word_idx].item()
```

Temperature 机制

Temperature 机制是一种通过温度参数 T 来调节模型输出概率分布的技术。它主要应用在模型最后输出层的 softmax 之前，通过调节 T 的大小来控制生成文本的随机性和创造性。

在训练的时候，Temperature被恒定为1，而在推理的时候，Temperature可以被设置为任何大于0的值。

对 Top-K 采样，引入 Temperature 后的分布为：

$$p\left(w_{i+1}^{1},\ldots,w_{i+1}^{K}\right)=\left\{\frac{\exp\left(\frac{o_i\left[w_{i+1}^{1}\right]}{T}\right)}{\sum_{j=1}^{K}\exp\left(\frac{o_i\left[w_{i+1}^{j}\right]}{T}\right)},\ldots,\frac{\exp\left(\frac{o_i\left[w_{i+1}^{K}\right]}{T}\right)}{\sum_{j=1}^{K}\exp\left(\frac{o_i\left[w_{i+1}^{j}\right]}{T}\right)}\right\} \tag{12}$$

对 Top-P 采样，引入 Temperature 后的分布为：

$$p\left(w_{i+1}^{1},\ldots,w_{i+1}^{|S_p|}\right)=\left\{\frac{\exp\left(\frac{o_i\left[w_{i+1}^{1}\right]}{T}\right)}{\sum_{j=1}^{|S_p|}\exp\left(\frac{o_i\left[w_{i+1}^{j}\right]}{T}\right)},\ldots,\frac{\exp\left(\frac{o_i\left[w_{i+1}^{|S_p|}\right]}{T}\right)}{\sum_{j=1}^{|S_p|}\exp\left(\frac{o_i\left[w_{i+1}^{j}\right]}{T}\right)}\right\} \tag{13}$$

当 $T>1$ 时分布更平坦，随机性增加，概率低的词也有更大机会被选中；当 $0<T<1$ 时分布更陡峭，随机性减弱，概率低的词被选中的概率更低。这使得我们可以根据不同场景需求灵活调节随机性。

Temperature 机制简化版示例代码：

```
def apply_temperature(probs, temperature):
    """Temperature 机制

    Args:
        probs: 概率分布
        temperature: 温度
    """
    # 对概率分布进行归一化
    probs = probs / torch.sum(probs)

    # 对概率分布进行温度调节
```

```
    probs = torch.pow(probs, 1/temperature)

    # 归一化概率
    probs = probs / torch.sum(probs)

    return probs
```