# 轻量级大语言模型MiniMind源码解读（一）：如何从头训练tokenizer？

Original 凡希 南极Python 2025年7月5日 07:12

在前面更新的《大模型炼丹术》系列的八篇文章中，我们已经一步步打好了大语言模型的基础，比如：

- Tokenizer怎么把文字变成模型能理解的数字?
- Embedding和位置编码怎么让模型“知道”词和词的位置关系?
- Attention机制怎么让模型学会“注意”重要的信息?
- GPT2的架构是怎么搭起来的?
- 大模型是怎么进行预训练的?
- 模型生成文本时用的各种策略（比如贪心、采样等）？
- 模型训练完之后，还有哪两种常见的微调方法?

从这篇开始，我们要进入“炼丹术”的下一阶段——进阶实战。

在这一新的阶段里，我们将结合开源的轻量级大语言模型MiniMind，针对《大模型炼丹术》系列中没有涉及或者浅浅带过的进阶知识点进行详细讲解。

具体来说，我们将对MiniMind的源码进行逐行解读，并结合张量（也就是数据）在模型中shape的变化，一步步看清楚它是怎么运作的。

正文马上开始～

## 一、导入训练tokenizer所需库

`tokenizers` 是Hugging Face出的一个高性能、可定制的子词分词器库，主要用于训练和使用像BPE、WordPiece、Unigram等子词模型，是训练 LLM（如 GPT/BERT）时常用的工具。

```
import random
import json
from tokenizers import (
    Tokenizer,
    decoders,
    models,
    pre_tokenizers,
    trainers,
)
import os
```

`Tokenizer` 是核心分词器对象，控制整个分词、编码、解码过程，可以与不同的模型、预处理器和解码器配合使用。

`models` 包含各种子词分词模型（如 BPE、WordPiece、Unigram），定义了如何对文本进行分割与映射成token IDs。

`pre_tokenizers` 定义了文本的预处理方式，负责在真正的分词前对文本进行初步分割，如按空格、字节或其他规则分割。

`trainers` 用于训练分词模型的工具，包括设置词表大小、特殊符号等的参数配置，常用的有 `BpeTrainer` 、 `WordPieceTrainer` 等。

`decoders` 用于将分词后的 token IDs 转回原始文本（解码），支持不同的解码策略，如 `ByteLevel` 和 `Metaspace` 。

## 二、初始化tokenizer

```
tokenizer = Tokenizer(models.BPE())
tokenizer.pre_tokenizer = pre_tokenizers.ByteLevel(add_prefix_space=False)
```

代码首先定义了一个使用BPE模型的分词器，该分词器使用BPE算法将文本拆分成子词单元，以增强模型对未登录词和低频词的处理能力。

然后将预处理器设置为ByteLevel，这一步将文本转换为字节级别的单位，允许更细粒度的文本处理，且add_prefix_space=False控制是否在每个单词前加空格，由于处理的是中文，因此将其设置为False。

## 三、设置训练器trainer并添加特殊token

```
# 定义特殊token
special_tokens = ["<|endoftext|>", "<|im_start|>", "<|im_end|>"]

# 设置训练器并添加特殊token
trainer = trainers.BpeTrainer(
    vocab_size=6400,
    special_tokens=special_tokens,  # 确保这三个token被包含
    show_progress=True,
    initial_alphabet=pre_tokenizers.ByteLevel.alphabet()
)
```

首先来看这些特殊tokens。

它们是在训练过程中专门定义的，用于标识特定的文本模式或结构，通常用于控制生成文本的结构。

- "<|endoftext|>"用于表示文本的结束，通常在模型生成任务中用来指示生成文本的终止。
- "<|im_start|>"标识对话或任务的开始，可以用于标记输入文本的起始。
- "<|im_end|>"标识对话或任务的结束，用来标记输入文本的结束。

这些特殊tokens会在训练过程中作为词表的一部分，确保它们在分词和生成过程中能被正确处理。

接下来解释trainer的参数。

- `vocab_size=6400` 表示模型训练过程中会生成最多 6400 个子词（包括特殊tokens）。一般情况下，词汇表的大小会在模型训练之前根据训练数据的大小和复杂度来决定。
- `special_tokens=special_tokens` 用于标记特殊tokens，使得它们不会在BPE训练过程中被拆分或合并，因为这些特殊标记一般用于标记对话的开始或者结束等特殊意义，而没有明显的对话语义信息。具体来说，在 BPE 训练过程中，算法会找到文本中最频繁的字节对（或字符对），并将它们合并成新的词汇项。例如，BPE会将两个常见的字符组合，如'a'和'b'，合并成一个新的词汇项'ab'。然而，特殊tokens是不参与这一过程的。它们是已经定义好的、固定的标记，不会被拆分或进行BPE 合并。所以，当训练过程中遇到 "<|im_start|>" 这样的特殊token时，它会被保留为一个整体，BPE不会再进一步拆分它。
- `show_progress=True` 用于在训练过程中会展示当前的训练进度，以便跟踪训练的进度和耗时。
- `initial_alphabet=pre_tokenizers.ByteLevel.alphabet()` 指定了BPE模型的初始字母表。在这里使用的是ByteLevel的字母表，ByteLevel是一个字符级的预处理方法，它将文本拆解为字节级的子单元（包括字母、标点符号、空格等）。这一设置的作用是让训练器初始化时使用ByteLevel分词器的默认字母表，从而使得ByteLevel分词器能够适应不同的字符和符号类型，特别是在处理包含非标准字符的文本时非常有用。

## 四、读取文本数据

使用预训练数据集训练tokenizer，为了便于演示，这里只读取前100条数据。

```
# 读取JSONL文件并提取文本数据
def read_texts_from_jsonl(file_path, max_samples=100):
    with open(file_path, 'r', encoding='utf-8') as f:
        for i, line in enumerate(f):
            if i >= max_samples:
                break
            data = json.loads(line)
            yield data['text']
```

```
data_path = r'D:\MyFile\github\minimind-master\minimind_dataset\pretrain_hq.jsonl'
texts = read_texts_from_jsonl(data_path)
```

查看数据示例：

```
print(list(texts)[1])
```

```
<|im_start|>根据输入的内容，编写一个类别标签。
这是一篇介绍如何阅读心电图的文章类别标签: 医学/心电图阅读指南<|im_end|> <|im_start|>帮我搜索一下最近的
```

## 五、开始训练tokenizer

直接调用 `tokenizer` 的 `train_from_iterator` 方法即可开始训练

```
# 训练tokenizer
tokenizer.train_from_iterator(texts, trainer=trainer)
```

## 六、设置解码器

为分词器设置一个 ByteLevel 解码器，让其在将 token ID 序列转换回原始文本时，能够正确还原被分词器按字节切分的内容。

```
tokenizer.decoder = decoders.ByteLevel()
```

## 七、保存训练好的tokenizer

```
tokenizer_dir = r"./model"
os.makedirs(tokenizer_dir, exist_ok=True)
tokenizer.save(os.path.join(tokenizer_dir, "tokenizer.json"))
tokenizer.model.save(tokenizer_dir)
```

## 八、手动创建并保存配置文件

```
# 手动创建配置文件
config = {
    "add_bos_token": False,
    "add_eos_token": False,
    "add_prefix_space": False,
    "added_tokens_decoder": {
        "0": {
```

```
            "content": "<|endoftext|>",
            "lstrip": False,
            "normalized": False,
            "rstrip": False,
            "single_word": False,
            "special": True
        },
        "1": {
            "content": "<|im_start|>",
            "lstrip": False,
            "normalized": False,
            "rstrip": False,
            "single_word": False,
            "special": True
        },
        "2": {
            "content": "<|im_end|>",
            "lstrip": False,
            "normalized": False,
            "rstrip": False,
            "single_word": False,
            "special": True
        }
    },
    "additional_special_tokens": [],
    "bos_token": "<|im_start|>",
    "clean_up_tokenization_spaces": False,
    "eos_token": "<|im_end|>",
    "legacy": True,
    "model_max_length": 32768,
    "pad_token": "<|endoftext|>",
    "sp_model_kwargs": {},
    "spaces_between_special_tokens": False,
    "tokenizer_class": "PreTrainedTokenizerFast",
    "unk_token": "<|endoftext|>",
    "chat_template": "{% if messages[0]['role'] == 'system' %}{% set system_message = mes
}

# 保存配置文件
with open(os.path.join(tokenizer_dir, "tokenizer_config.json"), "w", encoding="utf-8") as
    json.dump(config, config_file, ensure_ascii=False, indent=4)
```

| 字段名 | 解释 |
| --- | --- |
| add_bos_token | 是否自动在文本开头添加 bos_token（如 `< |
| add_eos_token | 是否自动在文本末尾添加 eos_token（如 `< |
| add_prefix_space | Byte-level 分词时是否在文本前加空格。通常英文中启用（True）更好，中文中设为 False。 |

| 字段名 | 解释 |
| --- | --- |
| added_tokens_decoder | 特殊 token 的详细配置。包括 token 内容、是否为特殊 token、是否仅限单词等。key 是内部 token ID。 |
| additional_special_tokens | 除了 bos/eos/pad/unk 外，额外声明的特殊 token 列表。当前为空。 |
| bos_token | 起始 token，通常用于语言模型的开头控制符，这里设为 `< |
| clean_up_tokenization_spaces | 解码时是否清理 token 化带来的空格冗余。False 表示不清理。 |
| eos_token | 结束 token，通常用于语言模型输出结束的标记，这里设为 `< |
| legacy | 设置为 True 兼容旧版本 tokenizer 行为。推荐保持默认。 |
| model_max_length | 模型支持的最大 token 长度。超过将触发截断或报错。这里为 32768。 |
| pad_token | 用于对齐 padding 的特殊 token。此处为 `< |
| sp_model_kwargs | SentencePiece 模型的额外配置参数（当前为 BPE，未使用，故为空）。 |
| spaces_between_special_tokens | 是否在特殊 token 之间自动添加空格。设置为 False。 |
| tokenizer_class | 指定 tokenizer 类型。Hugging Face 使用 "PreTrainedTokenizerFast" 支持 Rust 实现加速。 |
| unk_token | 用于标记未知词（out-of-vocabulary）的 token，这里也设为 `< |
| chat_template | Jinja2 模板字符串，用于格式化对话数据为模型输入格式。适用于 Chat 模型（如 LLaMA2-Chat、ChatGPT）。 |

## 九、测试训练好的tokenizer

```python
def eval_tokenizer():
    from transformers import AutoTokenizer

    # 加载预训练的tokenizer
    tokenizer = AutoTokenizer.from_pretrained(r"D:\MyFile\github\minimind-master\mm")

    messages = [
        {"role": "system", "content": "你是一个优秀的聊天机器人，总是给我正确的回应！"},
        {"role": "user", "content": '你来自哪里？'},
        {"role": "assistant", "content": '我来自地球'}
    ]
    new_prompt = tokenizer.apply_chat_template(
        messages,
        tokenize=False
    )


    # 获取实际词汇表长度（包括特殊符号）
    actual_vocab_size = len(tokenizer)
```

```
    print('tokenizer实际词表长度：', actual_vocab_size)

    model_inputs = tokenizer(new_prompt)
    print('encoder长度：', len(model_inputs['input_ids']))

    input_ids = model_inputs['input_ids']
    response = tokenizer.decode(input_ids, skip_special_tokens=False)
    print('decoder和原始文本是否一致：', response == new_prompt)

    print('\n输入文本：\n',new_prompt,'\n')
    print('解码文本：\n',response,'\n')

eval_tokenizer()
```

```
tokenizer实际词表长度： 259
encoder长度： 133
decoder和原始文本是否一致： True

输入文本：
 <|im_start|>system
你是一个优秀的聊天机器人，总是给我正确的回应！<|im_end|>
<|im_start|>user
你来自哪里？<|im_end|>
<|im_start|>assistant
我来自地球<|im_end|>


解码文本：
 <|im_start|>system
你是一个优秀的聊天机器人，总是给我正确的回应！<|im_end|>
<|im_start|>user
你来自哪里？<|im_end|>
<|im_start|>assistant
我来自地球<|im_end|>
```

至此，关于 MiniMind 中的tokenizer训练部分就解读完成了。

---

（如果你也想从0到1看懂大语言模型的底层实现，欢迎关注我，系列文章将持续更新，我们一起把大模型“拆干净”）


**南极Python**

CV一砖，大模型一瓦，深度学习全靠码。这里是我的技术工地，欢迎围观。

152篇原创内容

公众号