分布式训练框架 Megatron —— 2. 数据预处理

Megatron-LM git源码

数据预处理整体流程

前处理阶段主要负责的是根据输入的json格式的文本文件，分别进行分句，分词，基于已有的词典对token进行id化等一系列操作。下面以 bert 模型的预训练数据为例。

git上提供大数据太大了，有20个G，这是某个知乎大佬提供的小的。

> 【为了方便大家测试，这个eight.files3.json，我放下面了，其实任何纯文本，都可以，只要格式正确，json的，就好了；我这里是用了1000行的一个Json文件来做示范而已。】
>
> eight.files3.json

```
jsonfile="/workspace/dataset/eight.files3.json"
vocabfile="/workspace/dataset/bert-large-uncased-vocab.txt"
prefix="my_bert"

python tools/preprocess_data.py \
       --input $jsonfile \
       --output-prefix $prefix \
       --vocab-file $vocabfile \
       --dataset-impl mmap \
       --tokenizer-type BertWordPieceLowerCase \
       --split-sentences \
       --workers 10
```

mmap代表，数据保存的时候，是memory-map file。

预计最后产出的是如下两个文件：

my_bert_text_sentence.bin

my_bert_text_sentence.idx

代码里定义了一个 Encoder，这个 encoder 有两个重要属性和方法，分别是 splitter/split，tokenizer/encode。

其中 spliter 就是划分句子，也就是说一段话可能很长，需要把每一句话切分出来。然后 tokenizer 再对切出来的句子进行切词。可以猛戳这里了解一下 nltk.tokenize.punkt module。

（说句题外话，代码里 .pickle, 这个 pickle 本意是腌制，应该就是理解为可以持久保存的对象，但是这个 punkt 该怎么翻译理解好呢）

为了直观的感受一下 spliter 的功能，这里写了一段测试代码测试一下：

```
import nltk
library = "tokenizers/punkt/{}.pickle".format("english")
nltk.download('punkt')
splitter = nltk.data.load(library)

text = "this is first sequence. this is second sequence. this is third sequence."
output = {}
tokens_list = splitter.tokenize(text)
output["text"] = [partial for partial in tokens_list]
print(output)
```

输出为：

```
{'text': ['this is first sequence.', 'this is second sequence.', 'this is third sequence.']}
```

Tokenize 就是对句子进行切词，这里不再赘述。

整体代码的流程为：

1. partiton：将一个文件划分为很多小的文件。
2. 在每个小文件中，多线程进行 split / token 操作
3. 最后将结果合并。

这个脚本充分考虑了数据集很大，cpu 主存不够时的场景，比如将大模型进行 partition，以及在 partition 文件中使用多进程 imap 的调用方式，都是基于文件很大，主存不够的考虑。

Partition

这部分核心逻辑，就是打开文件后，一行一行的读入，然后每行写入到不同的文件中，比如设置 partition = 2， 就会把原来的 eight.files3.json 文件分为 eight.files3_0.json, eight.files3_1.json 两个文件。

```
fin = open(in_file_name, 'r', encoding='utf-8')
for line in fin:
    partitioned_input_files[index].write(line)
```

```
    index = (index + 1)%args.partitions
```

Split

在每个 partition 文件中，split sentences。

```
# split sentences in partition files
processes = []
for name in in_ss_out_names:
    p = multiprocessing.Process(target=partition.split_sentences,
                                args=((name['partition'],
name['sentence_split']),))
    p.start()
    processes.append(p)

for p in processes:
    p.join()

```

这里相当于启动了2个进程，分别去调用 partition.split_sentences， 每个进程负责处理一个 partition 文件。同时每个 partition.split_sentences 又开启了 n 个进程，来同时做 split 工作。所以一共有 partition * n 个进程。partition.split_sentences 如下:

```
class Partition(object):
    ...
    def split_sentences(self, file_name):
        input_file_name, output_file_name = file_name
        print("Opening", input_file_name)
        fin = open(input_file_name, 'r', encoding='utf-8')
        fout = open(output_file_name, 'w')

        encoder = Encoder(self.args)
        pool = multiprocessing.Pool(self.workers,
initializer=encoder.initializer)
        split_docs = pool.imap(encoder.split, fin, 32)

        proc_start = time.time()
        total_bytes_processed = 0
        for i, (doc, bytes_processed) in enumerate(split_docs, start=1):
            total_bytes_processed += bytes_processed
            fout.write(doc + "\n")
            self.print_processing_stats(i, proc_start, total_bytes_processed)

```

```
20            fin.close()
21            fout.close()
```

里面调用了 pool.imap(encoder.split, fin, 32) 函数启动了 self.workers 个进程，每个进程调用 encoder.split， 一行一行处理 fin 文件中的文本。encoder.split 调用了 nltk 库中的 splitter （splitter= nltk.load(library)。

```
1   class Encoder(object):
2       ...
3       def split(self, json_line):
4       data = json.loads(json_line)
5       output = {}
6       for key in self.args.json_keys:
7           text = data[key]
8           max_len = 1000000
9           tokens_list = [Encoder.splitter.tokenize(text[i:i+max_len]) for i in
    range(0, len(text), max_len)]
10          output[key] = [tokens for partial in tokens_list for tokens in
    partial]
11      return json.dumps(output), len(json_line)
```

这步完成之后生成 eight.files3_ss_0.json eight.files3_ss_1.json 两个 split sentence 之后的文件。

Encode (tokenize)

这一步对上面的 split sentence 进行切词并转换为 token id。并创建 .bin, .idx 文件，将encode 之后的 token id 写入 bin 文件，这块和 Split 类似，也是多进程处理。最后生成了 my_bert_0_text_sentence.bin my_bert_0_text_sentence.idx my_bert_1_text_sentence.bin my_bert_1_text_sentence.idx 四个文件。

Merge

最后是把每个 patition 文件生成的 .bin, .idx 文件 merge 为一个。生成了 my_bert_text_sentence.bin my_bert_text_sentence.idx 两个文件。

MMAP 到底是啥

前面提到

> mmap代表，数据保存的时候，是memory-map file。

这个数据格式到底是啥意思呢。简单创建一个测试数据， my_bert_test.json， 内容如下：

```
{"text" : "I am Iron Man. I am the savior."}
{"text" : "You are more than what you have become. You must take your place in the circle of life."}
```

然后使用 preprocess_data.py 做预处理：

```
python tools/preprocess_data.py \
       --input /home/shengying.wei/dataset/my_bert_test.json \
       --output-prefix my_bert_test \
       --vocab-file /home/shengying.wei/dataset/bert-large-uncased-vocab.txt \
       --dataset-impl mmap \
       --tokenizer-type BertWordPieceLowerCase \
       --split-sentences \
       --workers 1
```

因为 my_bert_test.json 有两句话，所以 split 后，my_bert_test_ss.json 中的内容如下：

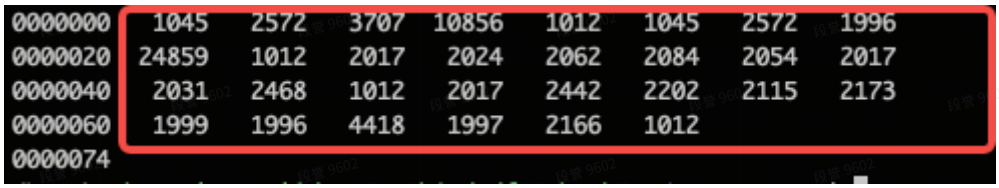
```
{"text": ["I am Iron Mann.", "I am the savior."]}
{"text": ["You are more than what you have become.", "You must take your place in the circle of life."]}
```

Tokenize 处理结果如下：

```
doc:  {'text': [1045, 2572, 3707, 10856, 1012, 1045, 2572, 1996, 24859, 1012]}
sentence_lens:  {'text': [5, 5]}
doc:  {'text': [2017, 2024, 2062, 2084, 2054, 2017, 2031, 2468, 1012, 2017, 2442, 2202, 2115, 2173, 1999, 1996, 4418, 1997, 2166, 1012]}
sentence_lens:  {'text': [9, 11]}
```

然后把 token id 写入 .bin 文件中。通过代码可以知道，token id 的类型是 uint16， 所以我们可以直接下面命令行来看看 .bin 文件中到底有哪些东西。

```
od -s my_bert_test_text_sentence.bin
```



可以看出，.bin 文件，就是把 token id 在内存中连续的存进去了。

接下来就是 .idx 文件，可想而知，要从一块连续的二进制内存中读出一句话的 token，至少要知道这句话的长度，要读取多少个byte吧，以及读取的起始地址等，这些辅助信息就储存在 .idx 中。这里不再赘述。

有了.idx 和 .bin 之后，读取数据是，先把 .bin 和 .idx 加入内存中，然后根据上述信息从 bin 文件中读取 token。

> 实际上 megatron 的数据读取上有很多丰富的细节，后续再逐一深入。