了解 LLaMA-2 模型结构(5)

发表评论 / ChatGPT, GPT, OpenAI

9. 转换 tokenizer.model 并保存

前面的章节转换模型的所有权重后，还需要转换 tokenizer.model 为自己需要的格式。

把 meta-llama/Llama-2-7b-chat-hf/ 目录下的 tokenizer.model 拷贝到 newsrc 目录下。

参照 https://github.com/karpathy/llama2.c 项目下的 tokenizer.py 文件，命名为 test09.py，文件保存到 newsrc 目录下：

```python
import os
import struct
import argparse
from typing import List

from sentencepiece import SentencePieceProcessor

class Tokenizer:
    def __init__(self, tokenizer_model=None):
        model_path = tokenizer_model
        assert os.path.isfile(model_path), model_path
        self.sp_model = SentencePieceProcessor(model_file=model_path)
        self.model_path = model_path

        # BOS / EOS token IDs
        self.n_words: int = self.sp_model.vocab_size()
        self.bos_id: int = self.sp_model.bos_id()
        self.eos_id: int = self.sp_model.eos_id()
        self.pad_id: int = self.sp_model.pad_id()
        #print(f"#words: {self.n_words} - BOS ID: {self.bos_id} - EOS ID: {self.eos_id}")
        assert self.sp_model.vocab_size() == self.sp_model.get_piece_size()

    def encode(self, s: str, bos: bool, eos: bool) -> List[int]:
        assert type(s) is str
        t = self.sp_model.encode(s)
        if bos:
            t = [self.bos_id] + t
        if eos:
            t = t + [self.eos_id]
        return t

    def decode(self, t: List[int]) -> str:
        return self.sp_model.decode(t)

    def export(self):

        # get all the tokens (postprocessed) and their scores as floats
        tokens, scores = [], []
        for i in range(self.n_words):

            # decode the token and light postprocessing
            t = self.sp_model.id_to_piece(i)
            s = self.sp_model.get_score(i)
            if i == self.bos_id:
                t = '\n<s>\n'
            elif i == self.eos_id:
                t = '\n</s>\n'
            t = t.replace('▁', ' ') # sentencepiece uses this character as whitespace
            b = t.encode('utf-8') # bytes of this token, utf-8 encoded

            tokens.append(b)
            scores.append(s)

        # record the max token length
        max_token_length = max(len(t) for t in tokens)

        # write to a binary file
        # the tokenizer.bin file is the same as .model file, but .bin
        tokenizer_bin = self.model_path.replace('.model', '.bin')
        with open(tokenizer_bin, 'wb') as f:
            f.write(struct.pack("I", max_token_length))
            for bytes, score in zip(tokens, scores):
                f.write(struct.pack("fI", score, len(bytes)))
                f.write(bytes)

t = Tokenizer("newsrc/tokenizer.model")
t.export()
```

运行 test09.py，查看newsrc 的文件目录

```
ls -l newsrc/tokenizer.*
-rwxrwxrwx 1 tony tony 433869 Mar 11 14:24 newsrc/tokenizer.bin
-rwxrwxrwx 1 tony tony 499723 Mar 10 23:51 newsrc/tokenizer.model
```

10. 查看 tokenizer.bin

下面给出的例子开始的都是C/C++代码，这样更好理解文件里面的内容

参照 https://github.com/karpathy/llama2.c 项目下的 run.c 文件，命名为 test01.c，文件保存到 newsrc 目录下：

```c
#include <stdio.h>
#include <stdlib.h>
// The Byte Pair Encoding (BPE) Tokenizer that translates strings <-> tokens

typedef struct {
    // 一个字符指针，指向与ID关联的字符串。
    char *str;
    // 一个整数，表示与字符串关联的ID。
    int id;
} TokenIndex;

typedef struct {
    // 一个指针数组，存储词汇表中每个单词的字符串表示。
    // 例如，如果 vocab[0] 是 "apple"，那么 vocab[0][0] 就是字符 'a'。
    char** vocab;
    // 一个浮点数数组，可能存储与词汇表中的每个单词相关联的分数或权重。
    float* vocab_scores;
    // 结构体，存储一个字符串和与其相关联的整数ID
    TokenIndex *sorted_vocab;
    // 一个整数，表示词汇表的大小，即vocab和vocab_scores数组中的元素数量。
    int vocab_size;
    // 一个无符号整数，表示词汇表中的最大token长度。
    unsigned int max_token_length;
    // 一个无符号字符数组，存储所有单字节字符串
    // 这个数组的大小被设定为512，可能是为了存储ASCII字符表中的所有可能的单字节字符串。
    unsigned char byte_pieces[512]; // stores all single-byte strings
} Tokenizer;
```

```

void build_tokenizer(Tokenizer* t, char* tokenizer_path, int vocab_size) {
    // i should have written the vocab_size into the tokenizer file... sigh
    // 设置了Tokenizer结构体中vocab_size的值。
    t->vocab_size = vocab_size;
    // malloc space to hold the scores and the strings
    // 为vocab，vocab_scores数组分配了内存，而sorted_vocab被初始化为NULL，表示它将在后续被"懒惰地"初始化
    t->vocab = (char**)malloc(vocab_size * sizeof(char*));
    t->vocab_scores = (float*)malloc(vocab_size * sizeof(float));
    t->sorted_vocab = NULL; // initialized lazily
    // 初始化byte_pieces数组，该数组存储所有单字节字符串
    for (int i = 0; i < 256; i++) {
        t->byte_pieces[i * 2] = (unsigned char)i;
        t->byte_pieces[i * 2 + 1] = '\0';
    }
    // read in the file
    FILE *file = fopen(tokenizer_path, "rb");
    if (!file) { fprintf(stderr, "couldn't load %s\n", tokenizer_path); exit(EXIT_FAILURE); 
    // 从文件中读取max_token_length的值
    if (fread(&t->max_token_length, sizeof(int), 1, file) != 1) { fprintf(stderr, "failed rea
    int len;
    // 这个循环块读取每个vocab字符串及其对应的vocab_scores值。
    for (int i = 0; i < vocab_size; i++) {
        if (fread(t->vocab_scores + i, sizeof(float), 1, file) != 1) { fprintf(stderr, "faile
        if (fread(&len, sizeof(int), 1, file) != 1) { fprintf(stderr, "failed read\n"); exit
        t->vocab[i] = (char *)malloc(len + 1);
        if (fread(t->vocab[i], len, 1, file) != 1) { fprintf(stderr, "failed read\n"); exit(E
        t->vocab[i][len] = '\0'; // add the string terminating token
    }
    fclose(file);
}

void free_tokenizer(Tokenizer* t) {
    for (int i = 0; i < t->vocab_size; i++)
    {
        free(t->vocab[i]);
    }
    if(t->vocab) free(t->vocab);
    if(t->vocab_scores) free(t->vocab_scores);
    if(t->sorted_vocab) free(t->sorted_vocab);
}

void print_tokenizer(Tokenizer* t) {
        printf("vocab = %d\n", t->vocab_size);
        printf("max_token_length = %d\n", t->max_token_length);
    for (int i = 0; i < t->vocab_size; i++)
    {
        printf("%5d, %12.6lf, (%s)\n", i, t->vocab_scores[i], t->vocab[i]);
    }
}

int main(int argc, char *argv[]) {

    char *tokenizer_path = "tokenizer.bin";

    // build the Tokenizer via the tokenizer .bin file
    Tokenizer tokenizer;
    int vocab_size = 32000; // 从模型文件的 config.json 获取
    build_tokenizer(&tokenizer, tokenizer_path, vocab_size);

    print_tokenizer(&tokenizer);
    free_tokenizer(&tokenizer);

    return 0;
}
```

编译 `test01.c`

```
make test01
cc     test01.c   -o test01
```

运行 `test01`

```
./test01 > 1.txt
```

由于输出的内容很多，所以我们把输出重定向到 `1.txt` 文件中，下面是 `1.txt` 文件的开头和结尾部分内容

```
vocab = 32000
max_token_length = 27
    0,      0.000000, (<unk>)
    1,      0.000000, (
<s>
)
    2,      0.000000, (
</s>
)
    3,      0.000000, (<0x00>)
    4,      0.000000, (<0x01>)
    5,      0.000000, (<0x02>)
    6,      0.000000, (<0x03>)
    7,      0.000000, (<0x04>)
    8,      0.000000, (<0x05>)
    9,      0.000000, (<0x06>)
   10,      0.000000, (<0x07>)
   11,      0.000000, (<0x08>)
   12,      0.000000, (<0x09>)
...
  259, -1000000000.000000, (   )
  260,     -1.000000, ( t)
  261,     -2.000000, (er)
  262,     -3.000000, (in)
  263,     -4.000000, ( a)
  264,     -5.000000, (en)
  265,     -6.000000, (on)
  266,     -7.000000, ( th)
  267,     -8.000000, (es)
  268, -1000000000.000000, (     )
  269,    -10.000000, ( s)
  270,    -11.000000, ( d)
  271,    -12.000000, (at)
...
31985, -31726.000000, (怪)
31986, -31727.000000, (联)
31987, -31728.000000, (역)
31988, -31729.000000, (泰)
31989, -31730.000000, (백)
31990, -31731.000000, (ọ)
31991, -31732.000000, (げ)
31992, -31733.000000, (べ)
31993, -31734.000000, (边)
31994, -31735.000000, (还)
31995, -31736.000000, (黃)
31996, -31737.000000, (왕)
31997, -31738.000000, (收)
31998, -31739.000000, (弘)
31999, -31740.000000, (给)
```

可以看到，`token` 的最大长度为27