怎么让英文大语言模型支持中文？（三） —— 对预训练模型进行指令微调

来自： AiGC面试宝典

宁静致远

2023年09月29日 12:32

一、为什么需要对预训练模型进行指令微调？

在之前讲过的继续预训练之后，我们应该对数据处理到训练、预测的整个流程有所了解，其实，基本上过程是差不多的。我们在选择好一个大语言模型之后。比如chatglm、llama、bloom等，要想使用它，得了解三个方面：输入数据的格式、tokenization、模型的使用方式。

二、对预训练模型进行指令微调 数据 如何处理？

数据的输入的话，一般情况下我们要在模型的官方代码上找到数据输入的那部分，或者说找到其它的一些开源的项目里面关于数据预处理的部分。找一份小的数据集，将这部分单独拿出来运行一下，看一下输出是什么。返回的结果是什么。比如一般看一下input_ids里面的特殊标记，labels是怎么构造的。举个例子，cpm-bee在forward里面需要额外传入span和length，与一般的不同只需要传入input_ids和labels。

这里我们看下chatglm的数据格式是怎么样的，在test_dataset.py里面：

```
import logging
import os
from dataclasses import dataclass
from typing import Optional, Dict, Sequence, Union, List
import datasets
import torch
import logging
from datasets import load_dataset, concatenate_datasets
import copy
import transformers
import random

IGNORE_INDEX = -100

logger = logging.getLogger('__name__')

PROMPT_TEMPLATE = (
    "Below is an instruction that describes a task. "
    "Write a response that appropriately completes the request.\n\n"
    "### Instruction:\n{instruction}\n\n### Response: "
)

def buid_instruction_dataset(data_path: Union[List[str],str],
                tokenizer: transformers.PreTrainedTokenizer,
                max_seq_length: int, data_cache_dir = None,
                preprocessing_num_workers = None,
                ):

    def tokenization(examples):
```

```
        sources = []
        targets = []
        # prompt = PROMPT_TEMPLATE
        for instruction, input, output in
zip(examples['instruct'],examples['query'],examples['answer']):
            if input is not None and input !="":
                instruction = instruction+'\n'+input
            # source = prompt.format_map({'instruction': instruction})
            source = instruction
            target = f"{tokenizer.bos_token}{output}{tokenizer.eos_token}"

            sources.append(source)
            targets.append(target)

        tokenized_sources = tokenizer(sources,return_attention_mask=False,
add_special_tokens=False)
        tokenized_targets = tokenizer(targets,return_attention_mask=False,
add_special_tokens=False)

        print(tokenized_targets)

        all_input_ids = []
        all_labels = []
        for s,t in
zip(tokenized_sources['input_ids'],tokenized_targets['input_ids']):
            s = s + [tokenizer.gmask_token_id]
            input_ids = torch.LongTensor(s + t)[:max_seq_length]
            labels = torch.LongTensor([IGNORE_INDEX] * len(s) + t)[:max_seq_length]
            assert len(input_ids) == len(labels)
            all_input_ids.append(input_ids)
            all_labels.append(labels)

        results = {'input_ids':all_input_ids, 'labels': all_labels}
        return results


    logging.warning("building dataset...")
    all_datasets = []

    if not isinstance(data_path,(list,tuple)):
        data_path = [data_path]
    for file in data_path:

        if data_cache_dir is None:
            data_cache_dir = str(os.path.dirname(file))
        cache_path = os.path.join(data_cache_dir,os.path.basename(file).split('.')
[0])
```

```
        os.makedirs(cache_path, exist_ok=True)
        try:
            processed_dataset = datasets.load_from_disk(cache_path)
            logger.info(f'training datasets-{file} has been loaded from disk')
        except Exception:
            print(file)
            raw_dataset = load_dataset("json", data_files=file,
cache_dir=cache_path)
            print(raw_dataset)
            tokenization_func = tokenization
            tokenized_dataset = raw_dataset.map(
                tokenization_func,
                batched=True,
                num_proc=preprocessing_num_workers,
                remove_columns=["instruct","query","answer"],
                keep_in_memory=False,
                desc="preprocessing on dataset",
            )
            processed_dataset = tokenized_dataset
            processed_dataset.save_to_disk(cache_path)
        processed_dataset.set_format('torch')
        all_datasets.append(processed_dataset['train'])
    all_datasets = concatenate_datasets(all_datasets)
    return all_datasets

@dataclass
class DataCollatorForSupervisedDataset(object):
    """Collate examples for supervised fine-tuning."""

    tokenizer: transformers.PreTrainedTokenizer

    def __call__(self, instances: Sequence[Dict]) -> Dict[str, torch.Tensor]:
        input_ids = instances["input_ids"]
        labels = instances["labels"]

        input_ids = torch.nn.utils.rnn.pad_sequence(
            input_ids, batch_first=True, padding_value=self.tokenizer.pad_token_id
        )
        labels = torch.nn.utils.rnn.pad_sequence(labels, batch_first=True,
padding_value=-100)
        return dict(
            input_ids=input_ids,
            labels=labels,
        )

if __name__ == "__main__":
  from transformers import AutoModelForCausalLM, AutoTokenizer
```

```
  tokenizer = AutoTokenizer.from_pretrained("model_hub/chatglm-6b",
trust_remote_code=True)
  all_datasets = buid_instruction_dataset(["data/msra/train.txt"], tokenizer,
max_seq_length=256)
  print(all_datasets[0])
  data_collator = DataCollatorForSupervisedDataset(tokenizer=tokenizer)
  data = data_collator(all_datasets[:2])
  print(data)
```

指令数据一般由三部分组成：instruction(instruct)、input(query)、output(answer)，分别表示提示指令、文本、返回的结果。 构造的时候一般是instruction和input进行拼接，当然input可能是为空的，最终对output进行预测。需要注意的是，除了instruction之外，可能还有特殊的prompt，不同模型的prompt是不一样的，比如：

```
PROMPT_DICT = {
    "chatglm_input": ("{instruction}{input}"),
    "alpaca_input": (
        "Below is an instruction that describes a task. "
        "Write a response that appropriately completes the request.\n\n"
        "### Instruction:\n{instruction}{input}\n\n### Response: "
    ),
    "bloom_input": ("Human: \n{instruction}{input}\n\nAssistant: \n"),
}
```

我们在构造的时候最好想之前预训练模型那样构造样本。

接下来再讲讲input_ids和labels。假设我们现在有样本：我爱北京天安门，你喜欢什么？，分词之后得到["我", "爱", "北京", "天安门", "你", "喜欢", "什么", "？"]，之后转换为token_id，[12, 112, 122324, 22323, 23, 2346, 1233, 545]，我们有Output：我喜欢故宫，转换为token_id：[12, 2346, 654]，一般情况下，output前后会被标识，比如bos_token_id和eos_token_id，假设分别为1和2，那么我们样本的输入就是：[12, 112, 122324, 22323, 23, 2346, 1233, 545] + [1] + [12, 2346, 654] + [2]。至于labels的构建，直接说明为：[-100, -100, -100, -100, -100, -100, -100, -100, 1, 12, 2346, 654, 2]，长度和input_ids保持一致。有人可能会疑惑，不是说是根据上一个字预测下一个字吗? 怎么是自己预测自己。这是因为一般的模型内部在前向计算的时候已经帮我们处理了：input_ids = input_ids[-1] labels=labels[1:]。-100是表示在计算损失的时候不考虑标签为-100的位置。如果还设置了文本最大长度，则input_ids后面用pad_token_id进行填充，需要注意可能有的模型的tokenization中pad_token为None，需要自己去设置一个，可以和eos_token_id一样。而标签需要用-100进行填充。

针对于chatglm，除了上述说明的外，它还有一个额外的[gMASK]标记。而它的输入为：

```
# instruction为instruction + input
# [gmask]等标记转换为id，这里直接展示
input_ids = instruction_ids + [gmask] + <sop> + output_ids + <eop>
# +1是[gmask]
-100 * len(instruction_ids + 1) + <sop> + output_ids + <eop>
```

所以说不同模型的输入构造可能不大一样，需要注意：

1. 特殊标记的使用。
2. 除了input_ids和labels，是否需要额外的输入。
3. 有的模型内部是帮你自动转换labels和input_ids计算损失，有的没有转换，可能需要自己手动转换，比如cpm-bee。

三、对预训练模型进行指令微调 tokenization 如何构建?

tokenization也很重要，我们一般可以先探索一下，在test_tokenizer.py中：

```
from transformers import AutoTokenizer, AutoModel
tokenizer = AutoTokenizer.from_pretrained("model_hub/chatglm-6b",
trust_remote_code=True)

text = "我爱北京天安门"
print(tokenizer(text))
print(tokenizer.convert_ids_to_tokens([18060, 12247, 14949]))
print(tokenizer.decode([18060, 12247, 14949]))

# 打印特殊 token
print("BOS token: ", tokenizer.bos_token)
print("EOS token: ", tokenizer.eos_token)
print("PAD token: ", tokenizer.pad_token)
print("UNK token: ", tokenizer.unk_token)

# 打印特殊 token_id
print("BOS token: ", tokenizer.bos_token_id)
print("EOS token: ", tokenizer.eos_token_id)
print("PAD token: ", tokenizer.pad_token_id)
print("UNK token: ", tokenizer.unk_token_id)

print(tokenizer.decode([130004,
          67470,     24,  83049,      4,  76699,     24,  83049,      4,  67357,
          65065,     24,  83049,      4,  64484,  68137,  63940,     24,  64539,
          63972,      4,  69670,  72232,  69023,     24,  83049,      4,  64372,
          64149,     24,  83049,      4,  63855,     24,  83049, 130005]))

# 这个是chatglm特有的。
input_ids = tokenizer.build_inputs_with_special_tokens([1], [2])

print(input_ids)
```

四、对预训练模型进行指令微调 模型 如何构建?

模型加载方式的话，一般使用的是AutoTenizer和AutoModelForCausalLM，但有的模型可能这么加载会报错。比如LLaMA的加载方式就是：LlamaForCausalLM和LlamaTokenizer,。针对于chatglm的话，加载方式为：AutoTenizer和AutoModel，但需要注意的是其加载的时候设置了trust_remote_code=True，该参数会根据映射找到真正使用的模型文件，比如modeling_chatglm.py。下载好模型权重后，我们可以根据情况先看看效果，在test_model.py里面:

```
from transformers import AutoTokenizer, AutoModel
tokenizer = AutoTokenizer.from_pretrained("model_hub/chatglm-6b",
trust_remote_code=True)
model = AutoModel.from_pretrained("model_hub/chatglm-6b",
trust_remote_code=True).half().cuda()
model = model.eval()
response, history = model.chat(tokenizer, "你好", history=[])
print(response)
response, history = model.chat(tokenizer, "晚上睡不着应该怎么办", history=history)
```

```
print(response)
```

五、是否可以结合 其他库 使用?

其它的一些就是结合一些库的使用了，比如：

- deepspeed
- transformers
- peft中使用的lora
- datasets加载数据

需要注意的是， 我们可以把数据拆分为很多小文件放在一个文件夹下，然后遍历文件夹里面的数据，用datasets加载数据并进行并行处理后保存到磁盘上。如果中间发现处理数据有问题的话要先删除掉保存的处理后的数据，再重新进行处理，否则的话就是直接加载保存的处理好的数据。

在SFT之后其实应该还有对齐这部分，就是对模型的输出进行规范，比如使用奖励模型+基于人类反馈的强化学习等，这里就不作展开了。

最后，接下来的话终于要开始去好好了解下langchain了，一直都在关注这个但没有好好地看下。