pretrain_sft

pretrain 理论

初始将 `input_ids` 复制一份作为 `labels`

input_ids	bos	t1	t2	t3	t4	t5
labels	bos	t1	t2	t3	t4	t5

取 `logits[:-1]` 和 `labels[1:]` 计算 loss

logits	bos	t1	t2	t3	t4
labels	t2	t3	t4	t5	eos

预测关系如图所示

sft 理论

初始将 `input_ids` 复制一份作为 `labels`， `p1-p2` 代表 prompt， `r1-r3` 代表 response， `labels` 只保留 response 以后的部分，其他使用 `-100` 进行 mask，期望模型从 prompt 之后开始学习

为什么：因为 sft 的 prompt 同质化严重

input_ids	bos	p1	p2	r1	r2	r3	eos
labels	–100	–100	–100	r1	r2	r3	eos

因此，真实计算 loss 时的对应关系为

logits	p2	r1	r2	r3
labels	r1	r2	r3	eos

pretrain 代码

```python
from tqdm import tqdm
import copy
import logging
from dataclasses import dataclass, field
from typing import Dict, Optional, Sequence

import torch
import transformers
from torch.utils.data import Dataset
from transformers import Trainer
from datasets import load_dataset
from typing import List
import os
import logging
from transformers import DataCollatorForSeq2Seq, default_data_collator, DataCollatorForLanguageModeling
from functools import partial
import os
os.environ['CUDA_VISIBLE_DEVICES'] = '0'
logger = logging.getLogger(__name__)


def get_all_datapath(dir_name: str) -> List[str]:
    all_file_list = []
    # all_file_size = []

    for (root, dir, file_name) in os.walk(dir_name):
        for temp_file in file_name:
            standard_path = f"{root}/{temp_file}"

            all_file_list.append(standard_path)

    return all_file_list


def load_dataset_from_path(data_path: Optional[str] = None,
                           cache_dir: Optional[str] = "cache_data",
                           data_file_number: Optional[int] = 2,
                           use_streaming: bool = False) -> Dataset:
    all_file_list = get_all_datapath(data_path)[:data_file_number]
    data_files = {'train': all_file_list}
    extension = all_file_list[0].split(".")[-1]

    logger.info("load files %d number", len(all_file_list))

```

```
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
```

```
    raw_datasets = load_dataset(
        extension,
        data_files=data_files,
        cache_dir=cache_dir,
        streaming=use_streaming
    )['train']
    return raw_datasets


IGNORE_INDEX = -100


@dataclass
class ModelArguments:
    model_name_or_path: Optional[str] = field(default="facebook/opt-125m"
)


@dataclass
class DataArguments:
    data_path: str = field(default=None, metadata={
        "help": "Path to the training data."})
    data_num_limit: int = field(default=None, metadata={
        "help": "the numbers of data file"
    })
    data_proc_num: int = field(default=None, metadata={
        "help": "the numbers of process"
    })
    use_streaming: bool = field(default=False, metadata={
                                "help": "use stream mode to process big dat
a"})


@dataclass
class TrainingArguments(transformers.TrainingArguments):
    cache_dir: Optional[str] = field(default=None)
    optim: str = field(default="adamw_torch")
    model_max_length: int = field(
        default=512,
        metadata={
            "help": "Maximum sequence length. Sequences will be right pad
ded (and possibly truncated)."},
    )
```

```
def make_train_dataset(tokenizer: transformers.PreTrainedTokenizer, data_path: str, data_file_number: int, data_proc_num: int, use_streaming: bool) -> Dataset:
    logging.warning("Loading data...")

    dataset = load_dataset_from_path(
        data_path=data_path,
        data_file_number=data_file_number,
        use_streaming=use_streaming
    )
    logging.warning("Formatting inputs...")

    def generate_sources_targets(examples: Dict, tokenizer: transformers.PreTrainedTokenizer):
        ins_data = examples['content']

        input_output = tokenizer(ins_data,
                                 return_tensors="pt",
                                 padding="longest",
                                 max_length=tokenizer.model_max_length-1,
                                 truncation=True)
        examples['input_ids'] = input_output['input_ids']
        return examples

    generate_sources_targets_p = partial(
        generate_sources_targets, tokenizer=tokenizer)

    if use_streaming:
        dataset = dataset.map(
            function=generate_sources_targets_p,
            batched=True
        ).shuffle(42, buffer_size=50000)
    else:
        dataset = dataset.map(
            function=generate_sources_targets_p,
            batched=True,
            desc="Running tokenizer on train dataset",
            num_proc=data_proc_num
        ).shuffle()

    return dataset


def train():
    parser = transformers.HfArgumentParser(
        (ModelArguments, DataArguments, TrainingArguments))
    model_args, data_args, training_args = parser.parse_args_into_dataclasses()
```

```
    model = transformers.AutoModelForCausalLM.from_pretrained(
        model_args.model_name_or_path,
        cache_dir=training_args.cache_dir,
        device_map='auto',
        torch_dtype=torch.bfloat16

    )

    # model.is_parallelizable = True
    # model.model_parallel = True
    torch.cuda.empty_cache()

    tokenizer = transformers.LlamaTokenizer.from_pretrained(
        model_args.model_name_or_path,
        cache_dir=training_args.cache_dir,
        model_max_length=training_args.model_max_length,
        padding_side="right",
        use_fast=True,
    )

    train_dataset = make_train_dataset(
        tokenizer=tokenizer,
        data_path=data_args.data_path,
        data_file_number=data_args.data_num_limit,
        data_proc_num=data_args.data_proc_num,
        use_streaming=data_args.use_streaming)
    train_dataset = train_dataset.remove_columns(
        ['uniqueKey', 'title', 'titleUkey', 'dataType', 'id', 'content'])

    data_collator = DataCollatorForLanguageModeling(
        tokenizer=tokenizer, mlm=False, pad_to_multiple_of=8
    )
    if not data_args.use_streaming:
        training_args.max_steps = -1



    trainer = Trainer(model=model,
                      tokenizer=tokenizer,
                      args=training_args,
                      train_dataset=train_dataset,
                      eval_dataset=None,
                      data_collator=data_collator,
                      )
    trainer.train()
    trainer.save_state()
    trainer.save_model(output_dir=training_args.output_dir)
```

```
182
183
184
185
186
187
```

```
if __name__ == "__main__":
    logging.basicConfig(
        format="%(asctime)s %(levelname)s [%(name)s] %(message)s", level=
logging.INFO, datefmt="%Y-%m-%d %H:%M:%S"
    )
    train()
```

Python

```
python train.py \
    --model_name_or_path /data/yh/bigscience_bloom-1b1 \
    --data_path /data/yh/WuDaoCorpus2.0_base_200G \
    --data_num_limit 60 \
    --data_proc_num 40 \
    --bf16 False \
    --output_dir output_dir \
    --num_train_epochs 3 \
    --per_device_train_batch_size 2 \
    --per_device_eval_batch_size 1 \
    --gradient_accumulation_steps 8 \
    --evaluation_strategy "no" \
    --save_strategy "steps" \
    --save_steps 1000 \
    --save_total_limit 10 \
    --learning_rate 2e-5 \
    --weight_decay 0. \
    --warmup_ratio 0.03 \
    --logging_steps 10 \
    --tf32 False \
    --model_max_length 1024 \
    --use_streaming True\
    --max_steps 10000
```

sft 代码

Python

```python
from peft.tuners.lora import LoraLayer
import copy
import logging
import logging
import os
import torch
import transformers
from dataclasses import dataclass, field
from datasets import load_dataset
from functools import partial
from torch.utils.data import Dataset
from tqdm import tqdm
from transformers import DataCollatorForSeq2Seq, Trainer
from typing import Dict, Optional, Sequence, List

logger = logging.getLogger(__name__)


@dataclass
class ModelArguments:
    model_name_or_path: Optional[str] = field(default="facebook/opt-125m"
)
    use_lora: Optional[bool] = field(default=False)


@dataclass
class DataArguments:
    data_path: str = field(default=None, metadata={
        "help": "Path to the training data."})
    source_length: int = field(default=512)
    target_length: int = field(default=512)


@dataclass
class TrainingArguments(transformers.TrainingArguments):
    cache_dir: Optional[str] = field(default=None)
    optim: str = field(default="adamw_torch")
    model_max_length: int = field(
        default=512,
        metadata={
            "help": "Maximum sequence length. Sequences will be right pad
ded (and possibly truncated)."},
    )
    use_deepspeed: bool = field(default=False)

```

```python

def get_all_datapath(dir_name: str) -> List[str]:
    all_file_list = []
    # all_file_size = []

    for (root, dir, file_name) in os.walk(dir_name):
        for temp_file in file_name:
            standard_path = f"{root}/{temp_file}"

            all_file_list.append(standard_path)

    return all_file_list


def load_dataset_from_path(data_path: Optional[str] = None,
                           cache_dir: Optional[str] = "cache_data") -> Dataset:
    all_file_list = get_all_datapath(data_path)
    data_files = {'train': all_file_list}
    extension = all_file_list[0].split(".")[-1]

    logger.info("load files %d number", len(all_file_list))

    raw_datasets = load_dataset(
        extension,
        data_files=data_files,
        cache_dir=cache_dir,
    )['train']
    return raw_datasets


IGNORE_INDEX = -100
PROMPT_DICT = {
    "prompt_input": (
        "Below is an instruction that describes a task, paired with an input that provides further context. "
        "Write a response that appropriately completes the request.\n\n"
        "### Instruction:\n{instruction}\n\n### Input:\n{input}\n\n### Response:"
    ),
    "prompt_no_input": (
        "Below is an instruction that describes a task. "
        "Write a response that appropriately completes the request.\n\n"
        "### Instruction:\n{instruction}\n\n### Response:"
    ),
}


```

```python
def _tokenize_fn(strings: Sequence[str], tokenizer: transformers.PreTrainedTokenizer) -> Dict:
    """Tokenize a list of strings."""
    tokenized_list = [
        tokenizer(
            text,
            return_tensors="pt",
            padding="longest",
            max_length=tokenizer.model_max_length,
            truncation=True,
        )
        for text in strings
    ]
    input_ids = labels = [tokenized.input_ids[0]
                          for tokenized in tokenized_list]
    ne_pad_token_id = IGNORE_INDEX if tokenizer.pad_token_id is None else tokenizer.pad_token_id
    input_ids_lens = labels_lens = [
        tokenized.input_ids.ne(ne_pad_token_id).sum().item() for tokenized in tokenized_list
    ]
    return dict(
        input_ids=input_ids,
        labels=labels,
        input_ids_lens=input_ids_lens,
        labels_lens=labels_lens,
    )


def preprocess(
        sources: Sequence[str],
        targets: Sequence[str],
        tokenizer: transformers.PreTrainedTokenizer,
) -> Dict:
    """Preprocess the data by tokenizing."""
    examples = [s + t for s, t in zip(sources, targets)]
    examples_tokenized, sources_tokenized = [_tokenize_fn(
        strings, tokenizer) for strings in (examples, sources)]
    input_ids = examples_tokenized["input_ids"]
    labels = copy.deepcopy(input_ids)
    for label, source_len in zip(labels, sources_tokenized["input_ids_lens"]):
        label[:source_len] = IGNORE_INDEX
    return dict(input_ids=input_ids, labels=labels)


def make_train_dataset(tokenizer: transformers.PreTrainedTokenizer, data_path: str, data_args: DataArguments) -> Dataset:
```

```
    logging.warning("Loading data...")

    dataset = load_dataset_from_path(
        data_path=data_path,
    )
    logging.warning("Formatting inputs...")
    prompt_input, prompt_no_input = PROMPT_DICT["prompt_input"], PROMPT_DICT["prompt_no_input"]

    def generate_sources_targets(examples: Dict, tokenizer: transformers.PreTrainedTokenizer):
        ins_data = examples['instruction']
        if 'input' not in examples.keys():
            input_data = [""] * len(ins_data)
        else:
            input_data = examples['input']
        output = examples['output']

        len_ = len(ins_data)

        # sources = []
        # targets = []

        # for i in range(len_):
        #     s_t = prompt_input.format_map({'instruction':ins_data[i],
        #                                    'input':input_data[i]}) if input_data[i] != "" else prompt_input.format_map({'instruction':ins_data[i]})
        #     sources.append(s_t)

        sources = [prompt_input.format_map({'instruction': ins_data[i], 'input': input_data[i]}) if input_data[
            i] != "" else prompt_no_input.format_map(
            {'instruction': ins_data[i]})
            for i in range(len_)]
        sources = [i[:data_args.source_length] for i in sources]
        targets = [
            f"{example[:data_args.target_length-1]}{tokenizer.eos_token}" for example in output]

        # sources = [prompt_input.format_map(example) if example.get("input", "") != "" else prompt_no_input.format_map(example)
        #            for example in examples]
        # targets = [
        #     f"{example['output']}{tokenizer.eos_token}" for example in examples]

        input_output = preprocess(
```

```
            sources=sources, targets=targets, tokenizer=tokenizer)
        examples['input_ids'] = input_output['input_ids']
        examples['labels'] = input_output['labels']
        return examples

    generate_sources_targets_p = partial(
        generate_sources_targets, tokenizer=tokenizer)

    dataset = dataset.map(
        function=generate_sources_targets_p,
        batched=True,
        desc="Running tokenizer on train dataset",
        num_proc=20
    ).shuffle()
    return dataset



def load_model_and_tokenizer(model_args: ModelArguments, training_args: TrainingArguments, data_args: DataArguments) -> tuple:

    if training_args.use_deepspeed:

        model = transformers.AutoModelForCausalLM.from_pretrained(
            model_args.model_name_or_path,
            cache_dir=training_args.cache_dir,
            torch_dtype='auto',
            # if model_args.model_name_or_path.find("falcon") != -1 else False
            trust_remote_code=True

        )
    else:
        model = transformers.AutoModelForCausalLM.from_pretrained(
            model_args.model_name_or_path,
            cache_dir=training_args.cache_dir,
            device_map='auto',
            torch_dtype='auto',
            # if model_args.model_name_or_path.find("falcon") != -1 else False
            trust_remote_code=True

        )

    if model_args.use_lora:

        logging.warning("Loading model to Lora")
```

```
        from peft import LoraConfig, get_peft_model
        LORA_R = 32
        # LORA_ALPHA = 16
        LORA_DROPOUT = 0.05
        TARGET_MODULES = [
            "o_proj","gate_proj", "down_proj", "up_proj"
        ]

        config = LoraConfig(
            r=LORA_R,
            # lora_alpha=LORA_ALPHA,
            target_modules=TARGET_MODULES,
            lora_dropout=LORA_DROPOUT,
            bias="none",
            task_type="CAUSAL_LM",
        )
        # model = model.to(torch.bfloat16)
        model = get_peft_model(model, config)
        # peft_module_casting_to_bf16(model)
        model.print_trainable_parameters()

    # model.is_parallelizable = True
    # model.model_parallel = True
    # torch.cuda.empty_cache()

    tokenizer = transformers.AutoTokenizer.from_pretrained(
        model_args.model_name_or_path, trust_remote_code=True)

    return model, tokenizer


def train():
    parser = transformers.HfArgumentParser(
        (ModelArguments, DataArguments, TrainingArguments))
    model_args, data_args, training_args = parser.parse_args_into_datacla
sses()

    model, tokenizer = load_model_and_tokenizer(
        model_args, training_args, data_args)

    with training_args.main_process_first(desc="loading and tokenization"
):

        train_dataset = make_train_dataset(
            tokenizer=tokenizer, data_path=data_args.data_path, data_args
=data_args)
```

```
    data_collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model=model,
                                           label_pad_token_id=IGNORE_INDEX
                                           )

    trainer = Trainer(model=model,
                      tokenizer=tokenizer,
                      args=training_args,
                      train_dataset=train_dataset,
                      eval_dataset=None,
                      data_collator=data_collator)
    trainer.train()
    trainer.save_state()
    trainer.save_model(output_dir=training_args.output_dir)


if __name__ == "__main__":
    logging.basicConfig(
        format="%(asctime)s %(levelname)s [%(name)s] %(message)s", level=logging.INFO, datefmt="%Y-%m-%d %H:%M:%S"
    )
```

```python
# --nnodes 1 --nproc_per_node 4 --master_port 25641

deepspeed --include localhost:0,1,2,3 train_sft.py \
    --deepspeed ds_zero2_no_offload.json \
    --model_name_or_path internlm-7b \
    --use_lora true \
    --use_deepspeed true \
    --data_path hz_sft_datav2 \
    --bf16 true \
    --fp16 false \
    --output_dir output_refusev2 \
    --num_train_epochs 5 \
    --per_device_train_batch_size 3 \
    --per_device_eval_batch_size 1 \
    --gradient_accumulation_steps 8 \
    --evaluation_strategy "no" \
    --save_strategy "epoch" \
    --save_total_limit 3 \
    --learning_rate 4e-4 \
    --logging_steps 10 \
    --tf32 False \
    --model_max_length 2048

# --save_steps 1000 \
```

chat template

```python
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("beomi/Llama-3-Open-Ko-8B")

messages = [
    {"role": "user", "content": "Hi there!"},
    {"role": "assistant", "content": "Nice to meet you!"},
    {"role": "user", "content": "Can I ask a question?"}
]

print(tokenizer.apply_chat_template(messages, tokenize=False))
```

```
<|begin_of_text|><|start_header_id|>user<|end_header_id|>

Hi there!<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Nice to meet you!<|eot_id|><|start_header_id|>user<|end_header_id|>

Can I ask a question?<|eot_id|>
```

```python
# 添加模型开始回复
print(tokenizer.apply_chat_template(messages, tokenize=False, add_generation_prompt=True))
```

```
<|begin_of_text|><|start_header_id|>user<|end_header_id|>

Hi there!<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Nice to meet you!<|eot_id|><|start_header_id|>user<|end_header_id|>

Can I ask a question?<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

sft 数据

数据任务

1.OpenAI 官网列出了 ChatGPT 擅长的所有任务项，诸如翻译、emoji 聊天……之类的。

2.NER、机器阅读理解、意图识别等传统的 NLP 任务。

3.参考业务需求。

数据量配比：难 task_type 数据多点，简单 task_type 数据少点

数据形式

1.prompt 表达方式需要多样性，不要千篇一律的“把中文句子 A 翻译成英文”，也要适当有一些“我在英国旅游，我现在需要向路人问路，我想表达 A 的意思，该怎么说”，“我是一个英文老师，我需要向我的学生讲解句子 A 用英文怎么写，请你用最正宗的表达方式帮我完成。”这么做的目的是防止模型只认识 prompt 中的几个关键 token，进而导致训练过拟合或者泛化性变差

2.prompt 长度均衡，既要有短数据，也要有长数据，避免模型的 attention 退化到无法聚焦长 prompt

3.answer 长度均衡，不能让模型没出输几个 token 就停止，适当的有一些语料让它学会输出尽量长的 answer，否则模型会很难 follow “不少于2000字”这种指令

4.多轮聊天的切换 topic 能力，有的数据当前 query 是和 session 有关系的，有的数据则是当前 query 和 session 毫无关系，要让模型自己学会判断 query 是否和 session 有关。类似的

数据还要有 system 是否生效，有些数据 system 是个摆设，有些数据的 answer 则和 system 直接相关

5.answer 分布的多样性，例如总共一万条训练数据，一千条数据的 answer 都说同一句话，太单一的话会严重让模型过拟合

数据生产

prompt：https://github.com/yizhongw/self-instruct

answer：GPT/Claude，部署选择 qwen 或者 deepseek-moe

筛选用户日志，例如有用/没用可以辅助 DPO/RLHF

sft 参数

注意的参数：

`epoch, gradient_accumulation_steps, global_batch_size, learning_rate, lr_scheduler_type, dropout`

影响速度的参数

`zero_stage, max_seq_len, offload, gradient_checkpointing, seq_parallel_size`

其他

`weight_decay, per_device_train_batch_size, num_warmup_steps`

sft 技巧

1.小模型大学习率，大模型小学习率，epoch 基本 1～3

2.模型的初始 loss，7B / 13B 可能在 2 左右，数据难了也有可能到 3，72B 则大多在 1 ～ 2 之间这个水平状态；最终 loss 则大概在 0.5 左右

拟合性

欠拟合

判断模型是真的连训练数据都没学会，还是学会了训练数据但无法进行泛化

测试方法：直接让模型回答训练集

1.没学会训练集

解决方法：多训 1 epoch，调整学习率（观察 loss 曲线和梯度，如果 loss 下降缓慢就增大学习率跳出局部最小值，如果 loss 比较震荡学习很困难就减小学习率提高训练稳定性）

2.学会了训练集

判断任务难度与模型 size 是否相符，判断 pretrain 是否学过相关知识（续写相关内容），抽样 answer 查看质量，重写 prompt

过拟合

暴露模型对什么任务类型过拟合，调整数据配比

评估

原则：Helpfulness、Honesty、Harmlessness

方法：GPT（prompt 参考Alignbench）、人工

Lora!

原理

将 ΔW 分解为 A*B

初始化

一个初始化为 0，一个使用某种随机初始化（保证训练的开始旁路矩阵依然是0矩阵）

超参

$$\mathbf{h} = (\mathbf{W}_0 + \frac{\alpha}{r}\Delta\mathbf{W})\mathbf{x}$$

r：秩 α：缩放因子

系数 $\frac{\alpha}{r}$ 越大，LoRA微调权重的影响就越大，在下游任务上越容易过拟合

系数 $\frac{\alpha}{r}$ 越小，LoRA微调权重的影响就越小（微调的效果不明显，原始模型参数受到的影响也较少）

注：

1. 简单任务所需的 r 不大，任务越难/多任务混合的情况，需要更大的 r
2. 越强的基座，所需的 r 应该更小
3. 数据规模越大，需要更大的 r