从零实现带RLHF的类ChatGPT：逐行解析微软DeepSpeed Chat的源码

写在最前面

本文最早写于2023年4月的这篇文章中《从零实现带RLHF的类ChatGPT：从TRL/ChatLLaMA/ColossalChat到DeepSpeed Chat》，后因要在「大模型项目开发线下营」上讲DSC 的实现而不断扩写其中的DSC，为避免原文过长，故把该文最后的DSC部分抽取出来成本文

前言

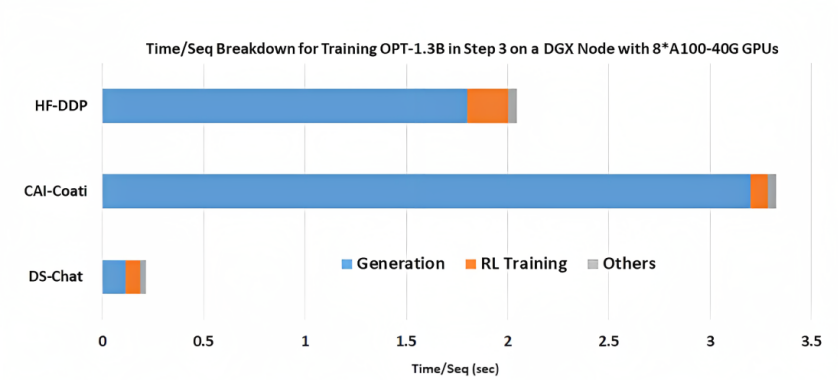
如此文所述，微软开源的DeepSpeed Chat(简称DSC)实现的不错，其类似ChatGPT的三阶段训练方式，会给你一个完整而通透的“PPO算法 /RLHF”的代码实现全流程(好的资料可以让你事半功)

抠完它的关键代码后，你会发现和之前本博客内另一篇写的原理部分都一一对应起来了(*如果你还没看过原理，建议先看此文：ChatGPT技术原理解析，只有懂原理才能更好的理解实现或实际实现，特别是该文的第三部分*)，而把论文、原理/算法、公式、代码一一对应，可以让你的理解有个质变

- 其论文地址：DeepSpeed-Chat: Easy, Fast and Affordable RLHF Training of ChatGPT-like Models at All Scales
  其官方的GitHub介绍页面：中文版、英文版
- 其GitHub代码地址：https://github.com/microsoft/DeepSpeedExamples/tree/master/applications/DeepSpeed-Chat

DSC的特点在于

1. 充分利用DeepSpeed项目的优化方案，考虑actor在RLHF阶段要在train (参数更新)和eval(经验采集)模式中反复切换，不做优化的话整体速度很慢，但是原本DeepSpeed的train加速和eval加速属于是解离的两种方案
2. DSC就设计了一种叫做DeepSpeedHybridEngine的引擎，使得actor在RLHF阶段能同时享有train和eval加速优化，整体提高RLHF速度
   一句话总结就是：DeepSpeed来给RLHF提速，遂成deepspeed chat



总的来说，DeepSpeed Chat和instructGPT的三阶段训练方式差不多，该三阶段分别用phase1、phase2、phase3表示

阶段	相关模型	赋能
0	具备基本生成能力的基座模型（通常为CausalLM）	-
1	有监督微调模型（SFT）	使用“prompt-response”数据（通俗含义上的“问答对”）对基座进行训练，基座将获得“根据指令生成出对应响应”的能力。
2	奖励模型（RM）	使用具有偏好评价的“prompt-response”数据（问答对）以及排序目标对预选模型进行训练，将得到具备“为指令数据做出人类偏好分值评估”能力的奖励模型。
3	SFT、Actor、RM、Critic	使用“prompt”数据（通俗含义上的“问句”），以第1、第2阶段训练得到的模型作为基线进行强化学习训练，最终得到具备“根据用户输入，生成符合人类偏好答复”能力的Actor模型。

下面简述这训练的三大阶段

> 注：七月在线ChatGPT课的一学员“吹牛班的春天”把这个DSC写的很细致了(年初至今的5个多月下来，除了本博客内的ChatGPT系列，春天这个deepspeed chat解析是我个人看到的唯一足够深入、细致的，主要真正写的深入、细致的文章实在是太少了，一方面 技术太新，二方面 涉及的细节太多)
>
> ---
>
> 故本文大部分的分析基于他的博客修改得到，当然，我为了让本文不断更加清晰易懂，在他的基础上做了大量反反复复的修改

第零部分 基本概念与数据集管理

0.1 数据的各个形式/概念

数据格式名称	说明
prompt	对当前情境的描述，为模型生成提供指令输入信息，可以理解为通俗含义上的“问句”，**适用于phase3**

response/answer	对上下文prompt的响应、回答、应答，可以理解为通俗含义上的“答句”
chosen	应答中的一种，人类所偏好的应答
rejected	应答中的一种，人类所排斥的应答
conversation	完整对话，由prompt衔接应答response得到
chosen_sentence	人类偏好的完整对话，由prompt衔接偏好应答chosen得到，**适用于phase1和phase2**
reject_sentence	人类排斥的完整对话，由prompt衔接排斥应答rejected得到，**适用于phase2**
unsup	无监督语料，符合自然语言要素的文本，适用于自回归语言模型的**无监督训练**

DeepSpeed-Chat设计的数据格式是直接服务于阶段训练的：

1. phase1：采用chosen_sentence作为训练数据，进行自回归语言建模训练
   chosen_sentence在通俗含义上代表“有效的问答数据”，有助于模型学习到理解指令并做出正确响应的能力
   而reject_sentence作为“相对无效的问答数据”，其响应部分往往是“反人类”的，并不利于模型进行学习

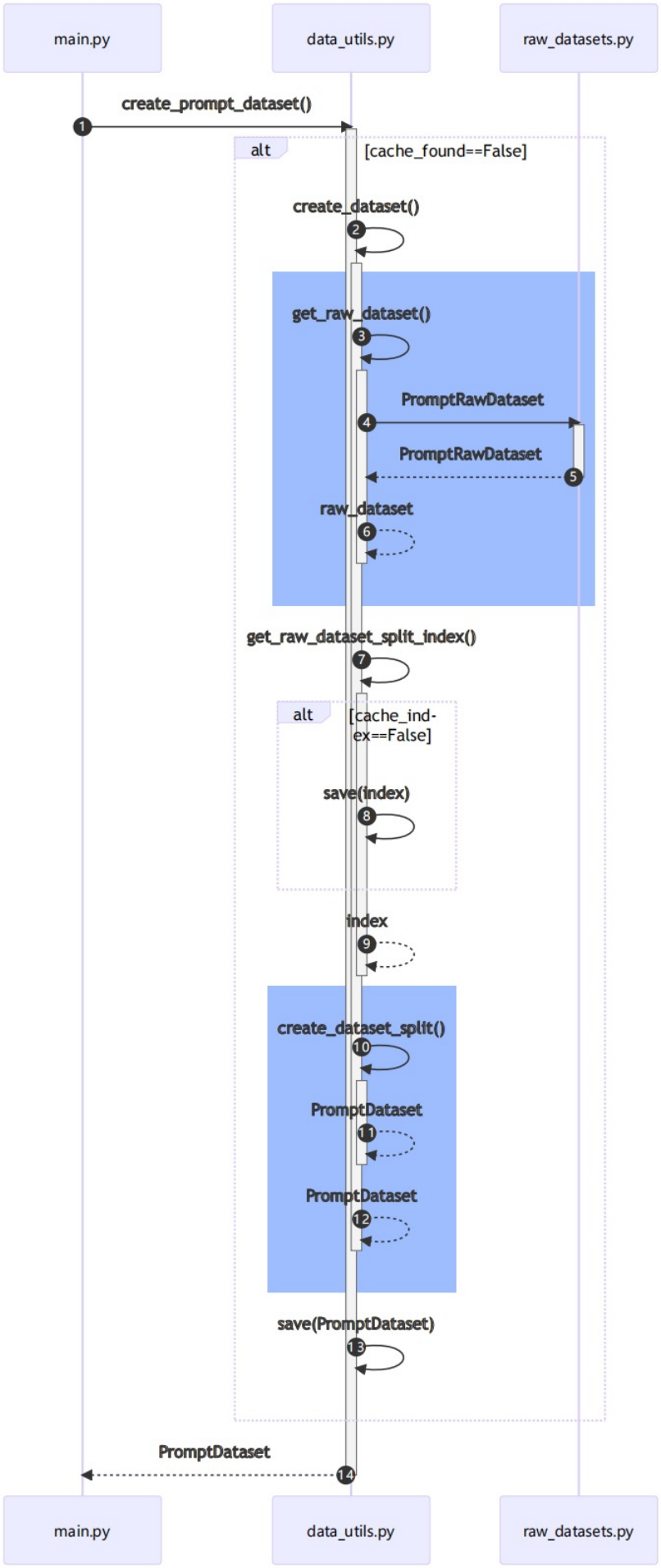
   因此在这个阶段采用了chosen_sentence作为训练数据
2. phase2：采用chosen_sentence和reject_sentence作为训练数据，进行成对排序训练(pairwise ranking loss)
   chosen_sentence和reject_sentence将分别作为成对数据中的较好者和较差者被送入模型中，模型将学习到其中的排序思路，从而给出更为合理的奖励评分

   这部分与InstructGPT中所述有些细微差别，InstructGPT是模型针对同个prompt构造了更多的conversations(如4至9个)，然后人类通过排列组合的方式，这些conversations将两两组成更多的成对数据被送入模型中进行训练
   不过，总的来说，DeepSpeed-Chat与InstructGPT的训练思想在本质上是一致的
3. phase3：采用prompt作为基本数据，调用中间模型(Actor、SFT、Critic、RM)根据基本数据构造出经验数据，使用强化学习中的PPO算法进行训练
4. 无监督训练：采用无监督语料数据，进行无监督的自回归语言建模训练
   InstructGPT提出，进行phase3的RLHF训练时，为使得模型在学习人类偏好的过程中仍能保有预训练模型解决任务的性能，引入了传统的自回归语言建模进行联合训练

0.2 DeepSpeed-Chat的数据读取流

首先，明确UML时序图中的各元素的含义：

1. 箭头表示信息传递：实线表示调用，虚线表示返回；
2. alt表示假设分支，其后方“[]”中的内容表示“条件”；
3. loop表示循环；
4. 淡蓝色区域即为高亮部分



总的来说，在训练的主流程代码main.py中，供DataLoader调用的Dataset（PromptDataset）将通过函数“create_prompt_dataset()”进行获取，其中将涉及到预存机制：

- 如果已经存在预存数据，则直接通过torch.load()进行载入得到Dataset；

- 如果未存在预存数据，则需要进行一系列预处理和预存的操作

其中，获取Dataset的过程大致为(“括号序号”与UML时序图的“圈序号”对应)：

查看是否存在数据集缓存(1)：如果存在则直接读取返回(14)；如果不存在则构建缓存(2-13)：

1. 读取全量数据集PromptRawDataset(3-6)；
   源码中预定义了几个开源数据集的PromptRawDataset类

   说明：当用户想要使用自定义数据集进行训练时，就必然需要自行编写与数据集情况相关的PromptRawDataset类，从而实现自定义数据的读取，详见下文0.2.1节
2. 查看是否缓存有切分后的index（该index可作为索引，从原始数据中取出对应数据构成子数据集），如果有则直接读取返回，如果没有则进行构建(此处并不十分重要，故不再加以叙述) (7-9)；
3. 根据index从全量数据集中取出子数据集，将子数据集进一步处理成对应phase所需的格式（如chosen_sentence、reject_sentence等），并且使用tokenizer提前进行encode，将encode后的内容使用Dataset类进行维护，得到最终所需的Dataset实例PromptDataset(10-12)；

   说明：这部分是子数据集的处理过程，理解子数据集的处理过程及其形式，将能更加透彻地理解各个阶段模型所需的输入形式，详见下文0.2.2节
4. 将Dataset实例进行存储(13)

0.2.1 自定义PromptRawDataset类：utils/data下的——raw_datasets.py与data_utils.py

具体代码如下所示

```
# applications/DeepSpeed-Chat/training/utils/data/raw_datasets.py
class PromptRawDataset(object):

    def __init__(self, output_path, seed, local_rank, dataset_name):
        """
        初始化
        :param output_path: 输出缓存路径。
        :param seed: 随机种子。
        :param local_rank: 当前进程序号。
        :param dataset_name: 数据集名称，后续指定所需读取的数据集时将以名称为准。
        """
        self.dataset_name = dataset_name
        self.dataset_clean_name = dataset_clean_name
        self.output_path = output_path
        self.seed = seed
        self.local_rank = local_rank
        # load_dataset源自datasets库，该方法支持读取csv/json/text等多种文件格式的数据
        self.raw_datasets = load_dataset(dataset_name)

    def get_train_data(self):
        """
        获取训练集
        :return: dataset数据格式
        """
        return

    def get_eval_data(self):
        """
        获取验证集
        :return: dataset数据格式
        """
        return

    # The prompt should be in the format of: " Human: " + actual_prompt_sentence + " Assistant:"
    def get_prompt(self, sample):
        """
        从dataset的sample（单个样本）中获取prompt。
        :param sample: dataset的元素
        :return: prompt。prompt的格式必须为 "Human: {} Assistant:".format(actual_prompt_sentence)
        """
        return

    # The chosen response should be in the format of: " " + actual_response_sentence
    def get_chosen(self, sample):
        """
        从dataset的sample（单个样本）中获取chosen。chosen实际上是“chosen response”，指的是“精选的回复”，即人类所偏好的、高分的回复。
        :param sample: dataset的元素
        :return: chosen。chosen的格式必须为" {}".format(actual_response_sentence)
        """
        return

    # The rejected response should be in the format of: " " + actual_response_sentence
    # If the dataset does not have rejected response, return None
    def get_rejected(self, sample):
```

```
        """
        从dataset的sample（单个样本）中获取rejected。rejected实际上是“rejected response”，指的是“排斥的回复”，即人类所厌恶的、低分的回复。
        :param sample: dataset的元素
        :return: rejected。如果数据集中不存在则返回为None；如果存在，则其格式必须为 " {}".format(actual_response_sentence)
        """
        return

    def get_prompt_and_chosen(self, sample):
        """
        从dataset的sample（单个样本）中获取prompt与chosen。
        :param sample: dataset的元素
        :return: prompt与chosen的衔接。同样需要满足上述格式要求，即衔接结果为
        "Human: {} Assistant: {}".format(actual_prompt_sentence, actual_response_sentence)
        """
        return

    def get_prompt_and_rejected(self, sample):
        """
        从dataset的sample（单个样本）中获取prompt与rejected。
        :param sample: dataset的元素
        :return: prompt与rejected的衔接。同样需要满足上述格式要求，即衔接结果为
        "Human: {} Assistant: {}".format(actual_prompt_sentence, actual_response_sentence)
        """
        return
```

有几点值得注意下

1. 自定义的数据集可以继承自上述的“PromptRawDataset”类，例如class CustomDataset(PromptRawDataset)
2. 然后重写其中的self.dataset_name及self.dataset_clean_name，此处的“dataset_name”即为传参指定数据集时所要填写的名称，例如self.dataset_name=custom
3. 在设置传参--data_path=‘custom’时，将会读取到CustomDataset的数据用于进行训练
4. 另外其中的get_train_data()等实例函数也需要进行重写，主要是实现将原始数据处理成注释所提及格式

定义好自定义PromptRawDataset后，还需要对其进行“注册”，具体可见下述代码块

```
# applications/DeepSpeed-Chat/training/utils/data/data_utils.py
def get_raw_dataset(dataset_name, output_path, seed, local_rank):

    if "Dahoas/rm-static" in dataset_name:
        return raw_datasets.DahoasRmstaticDataset(output_path, seed,
                                                  local_rank, dataset_name)
    elif "Dahoas/full-hh-rlhf" in dataset_name:
        return raw_datasets.DahoasFullhhrlhfDataset(output_path, seed,
                                                    local_rank, dataset_name)
    ...
    """
    将自定义的PromptRawDataset在此处进行注册
    届时在传参“--data_path”中赋值“custom”即可读取到相应的数据集
    """
    elif "custom" in dataset_name:
        return raw_datasets.CustomDataset(output_path, seed,
                                          local_rank, dataset_name)
    else:
      raise RuntimeError(
          f"We do not have configs for dataset {dataset_name}, but you can add it by yourself in raw_datasets.py."
      )
```

0.2.2 阶段数据集处理过程：utils/data/data_utils.py包含三个阶段的数据处理

直接看代码

```
# applications/DeepSpeed-Chat/training/utils/data/data_utils.py
def create_dataset_split(current_dataset, raw_dataset, train_phase, tokenizer,
                         end_of_conversation_token, max_seq_len):
    """
    将根据不同的阶段（train_phase）对数据集进行处理，主要是调用原先在PromptRawDataset类中定义的实例函数来实现。
    """
    prompt_dataset = []
    chosen_dataset = []
    reject_dataset = []
    if train_phase == 1:
        # 因为phase1只需要用到chosen数据，所以只取chosen进行处理
        for i, tmp_data in enumerate(current_dataset):
            # 获取chosen_sentence，即是将prompt和chosen拼接起来形成完整对话
```

```
            chosen_sentence = raw_dataset.get_prompt_and_chosen(
                                                                    tmp_data)
            if chosen_sentence is not None:
                # 在对话末尾加入对话终止符
                chosen_sentence += end_of_conversation_token

                # 使用tokenizer处理chosen_sentence，采取截断truncation
                chosen_token = tokenizer(chosen_sentence,
                                         max_length=max_seq_len,
                                         padding="max_length",
                                         truncation=True,
                                         return_tensors="pt")

                # 去掉batch维度
                chosen_token["input_ids"] = chosen_token["input_ids"].squeeze(
                    0)
                chosen_token["attention_mask"] = chosen_token[
                    "attention_mask"].squeeze(0)

                # 存储tokenize结果至列表chosen_dataset
                chosen_dataset.append(chosen_token)


    elif train_phase == 2:
        # phase2需要用到chosen_sentence和reject_sentence
        # 所以需要对两者都进行处理
        for i, tmp_data in enumerate(current_dataset):
            # 获取chosen_sentence，即是将prompt和chosen拼接起来形成完整对话
            chosen_sentence = raw_dataset.get_prompt_and_chosen(
                tmp_data)  # the accept response

            # 获取reject_sentence，即是将prompt和rejeced拼接起来形成完整对话
            reject_sentence = raw_dataset.get_prompt_and_rejected(
                tmp_data)

            if chosen_sentence is not None and reject_sentence is not None:
                # 在对话末尾加入对话终止符
                chosen_sentence += end_of_conversation_token  # the accept response
                reject_sentence += end_of_conversation_token

                # 使用tokenizer处理，采取截断truncation
                chosen_token = tokenizer(chosen_sentence,
                                         max_length=max_seq_len,
                                         padding="max_length",
                                         truncation=True,
                                         return_tensors="pt")
                reject_token = tokenizer(reject_sentence,
                                         max_length=max_seq_len,
                                         padding="max_length",
                                         truncation=True,
                                         return_tensors="pt")
                chosen_token["input_ids"] = chosen_token["input_ids"]
                chosen_token["attention_mask"] = chosen_token["attention_mask"]

                # 存储tokenize结果至列表chosen_dataset
                chosen_dataset.append(chosen_token)

                reject_token["input_ids"] = reject_token["input_ids"]
                reject_token["attention_mask"] = reject_token["attention_mask"]
                # 存储tokenize结果至列表reject_dataset
                reject_dataset.append(reject_token)

    elif train_phase == 3:
        # phase3用到prompt，prompt将被用来生成经验数据
        for i, tmp_data in enumerate(current_dataset):
            # 直接获取prompt
            prompt = raw_dataset.get_prompt(tmp_data)
            if prompt is not None:
                prompt_token = tokenizer(prompt, return_tensors="pt")
                prompt_token["input_ids"] = prompt_token["input_ids"]
                prompt_token["attention_mask"] = prompt_token["attention_mask"]

                for key_word in ["input_ids", "attention_mask"]:
                    # 获取当前文本token的实际长度
                    length = prompt_token[key_word].size()[-1]

                    # phase3此处的max_seq_len其实是max_prompt_len，默认只有256
                    if length > max_seq_len:
                        # 如果当前文本token长度比max_prompt_len还长
```

```
                        # 那么就截断文本前面的部分，保留后面max_prompt_len长度的部分文本
                    # 然后将token进行flip（翻转/倒序），之后在data_collator中再将其flip回来
                    y = prompt_token[key_word].squeeze(0)[length -
                                                          (max_seq_len -
                                                           1):].flip(0)
                else:
                    # 将token进行flip（翻转/倒序），之后在data_collator中再将其flip回来
                    y = prompt_token[key_word].squeeze(0).flip(0)
                prompt_token[key_word] = y
            prompt_dataset.append(prompt_token)

    # 返回PromptDataset实例，该实例相当于torch中的Dataset，可供DataLoader调用
    return PromptDataset(prompt_dataset, chosen_dataset, reject_dataset,
                         tokenizer.pad_token_id, train_phase)
```

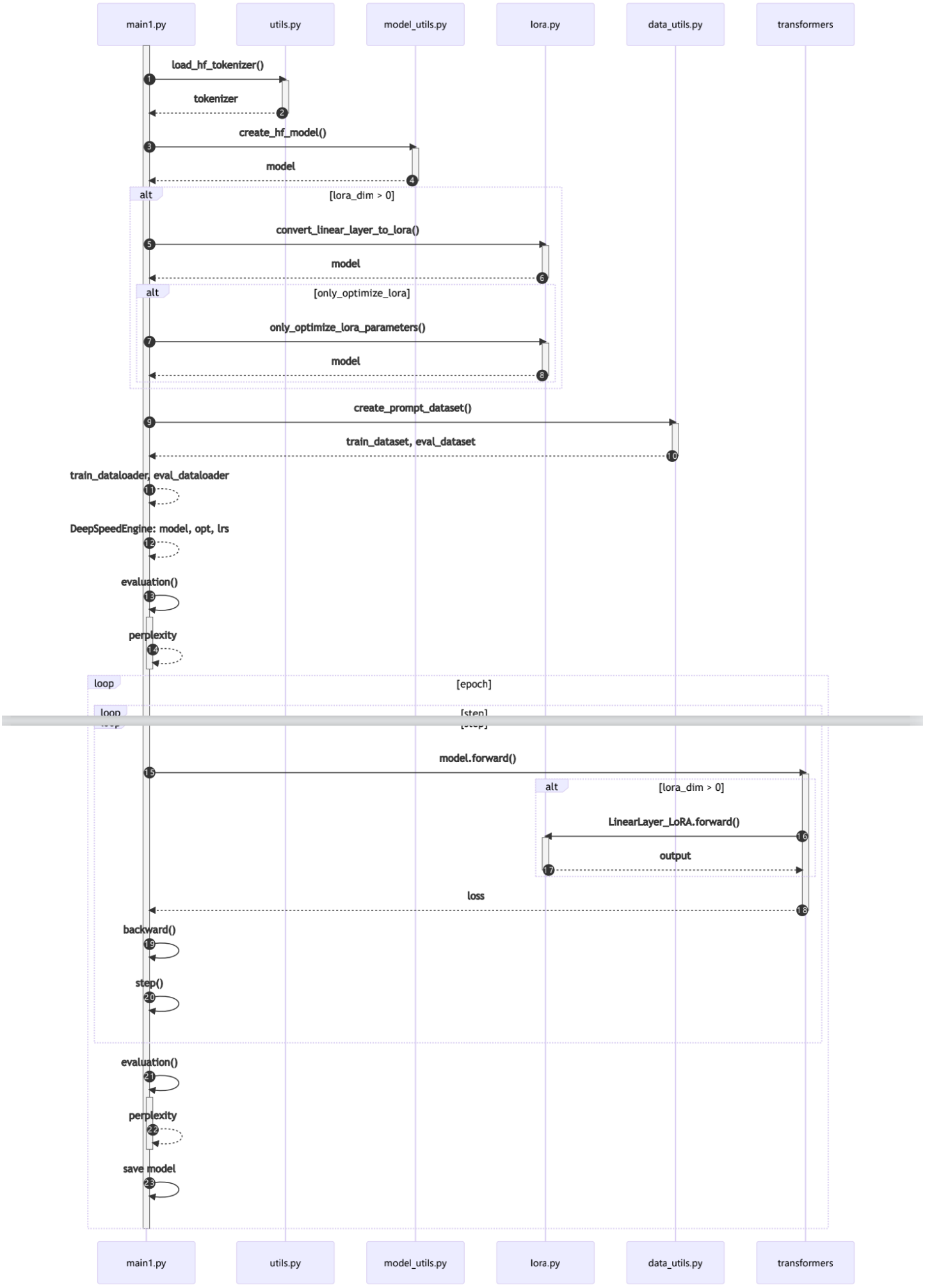
此处的处理部分很大程度依赖于原先所定义的PromptRawDataset实例函数，由此可见，只要正确编写实例函数，后续过程基本也不会出现什么问题。流程大致就是取出对应阶段所需的格式数据，然后使用tokenizer进行处理，综上所述：

- phase1模型所需的输入数据为chosen_sentence的input_ids及attention_mask；
- phase2模型所需的输入数据为chosen_sentence和reject_sentence的input_ids及attention_mask；
- phase3模型所需的输入数据为promt的input_ids及attention_mask

第一部分 DSC之phase-1: Supervised Finetuning

1.1 SFT的训练流程

phase1的核心代码见：applications/DeepSpeed-Chat/training/step1_supervised_finetuning/main.py，至于其训练过程如下图所示(*鼠标右键点击图片：在新标签页中打开图片，可以查看高清大图*)


main1.py
utils.py
model_utils.py
lora.py
data_utils.py
transformers
load_hf_tokenizer()
tokenizer
create_hf_model()
model
alt
[lora_dim > 0]
convert_linear_layer_to_lora()
model
alt
[only_optimize_lora]
only_optimize_lora_parameters()
model
create_prompt_dataset()
train_dataset, eval_dataset
train_dataloader, eval_dataloader
DeepSpeedEngine: model, opt, lrs
evaluation()
perplexity
loop
[epoch]
loop
[step]
model.forward()
alt
[lora_dim > 0]
LinearLayer_LoRA.forward()
output
loss
backward()
step()
evaluation()
perplexity
save model
main1.py
utils.py
model_utils.py
lora.py
data_utils.py
transformers

1. 载入tokenizer(1-2)
2. 载入基座模型(目前仅支持部分CausalLM模型) (3-4)
3. 根据是否设置lora_dim（LoRA的低秩维度）判断是否启用LoRA技术， 如果启用，则将基座模型结构进行LoRA改造，并返回改造后的模型(5-6)
4. 判断是否启用“仅更新LoRA参数”，如果启用，则对其余结构参数进行冻结处理，并返回冻结处理后的模型(7-8)
5. 获取Dataset (9-10)
6. 实例化DataLoader(11)
7. 使用DeepSpeed的优化技术DeepSpeedEngine包裹模型等对象(12)
8. 开始正式训练前首先进行指标评估，选用的指标为困惑度perplexity(13-14)
9. 开始训练，epoch循环：
    1. step循环：
        1. **正向传播得到loss(15-18)，如果模型启用了LoRA技术，则正向传播还需要经过LoRA结构(16-17)；**
        2. 反向传播计算梯度(19)；
        3. 更新模型参数（其中所涉及的梯度累计gradient_accumulation_steps将由DeepSpeedEngine自动进行管理，无需过度关注）(20)；
    2. 经过1个epoch的训练后进行指标评估(21-22)；
    3. 保存模型(23)。

1.2 关于LoRA与困惑度的说明

上述过程有2个细节，值得一提

1. 关于**LoRA的详解**，可看此文《Alpaca-LoRA：通过PEFT库在消费级GPU上微调「基于LLaMA的Alpaca」》的2.2.3节
2. DeepSpeed-Chat选择了困惑度perplexity作为phase1训练期间的评估指标
   困惑度perplexity是一种度量语言模型性能的指标，它衡量了训练好的模型对测试数据的拟合程度，对于输出句子的每个token，都可以得到其输出的置信概率值，将这些值相乘并取其几何平均数的倒数即可计算得到困惑度perplexity，使用公式表达更为简洁：

   $$\text{perplexity} = \left(\prod_{t=1}^{T} p_t\right)^{-\frac{1}{T}}$$

   其中，输出的句子共有T个token，第t 个token的置信概率值为p_t

   而CausalLM模型的训练过程通常采用对数似然损失来进行优化，其输出的损失公式如下：

   $$\text{loss} = -\frac{1}{T}\sum_{t=1}^{T}\log p_t$$

   其中，输出的句子共有T个token，第t tt个token的置信概率值为p_t

   因此perplexity与CausalLM的loss之间实际存在如下关系：

   $$\text{perplexity} = \exp(\text{ loss })$$

   相关源码的perplexity计算也是基于上述公式得到的：先是将验证数据输入至模型，得到模型loss输出，然后通过perplexity与loss之间的指数关系计算得到perplexity

```
    def evaluation(model, eval_dataloader):
        """
        以困惑度perplexity为评估指标进行验证
        """
        model.eval()
        losses = 0
        for step, batch in enumerate(eval_dataloader):
            """
            batch: 由input_ids、attention_mask、labels共3个部分组成的dict。
            其中每个部分的shape均为(bs, max_seq_len)
            """
            batch = to_device(batch, device)
            with torch.no_grad():
                outputs = model(**batch)

            """Causal LM 的损失函数为交叉熵损失"""
            loss = outputs.loss
            losses += loss.float()
        losses = losses / (step + 1)

        try:
            """困惑度perplexity通常可以通过exp(CELoss)计算得到"""
            perplexity = torch.exp(losses)
        except OverflowError:
            perplexity = float("inf")

        try:
            """
            - get_all_reduce_mean中调用了torch.distributed.all_reduce(perplexity, op=torch.distributed.ReduceOp.SUM)
            - 对所有进程、或者说GPU（因为通常情况下就是单个进程控制单个GPU）中的perplexity进行求和
            - 然后再除以全局进程数torch.distributed.get_world_size()得到平均的perplexity结果
            """
            perplexity = get_all_reduce_mean(perplexity).item()
```

```
        except:
            pass
    return perplexity
```

第二部分 DSC之phase-2: Reward Model Finetuning

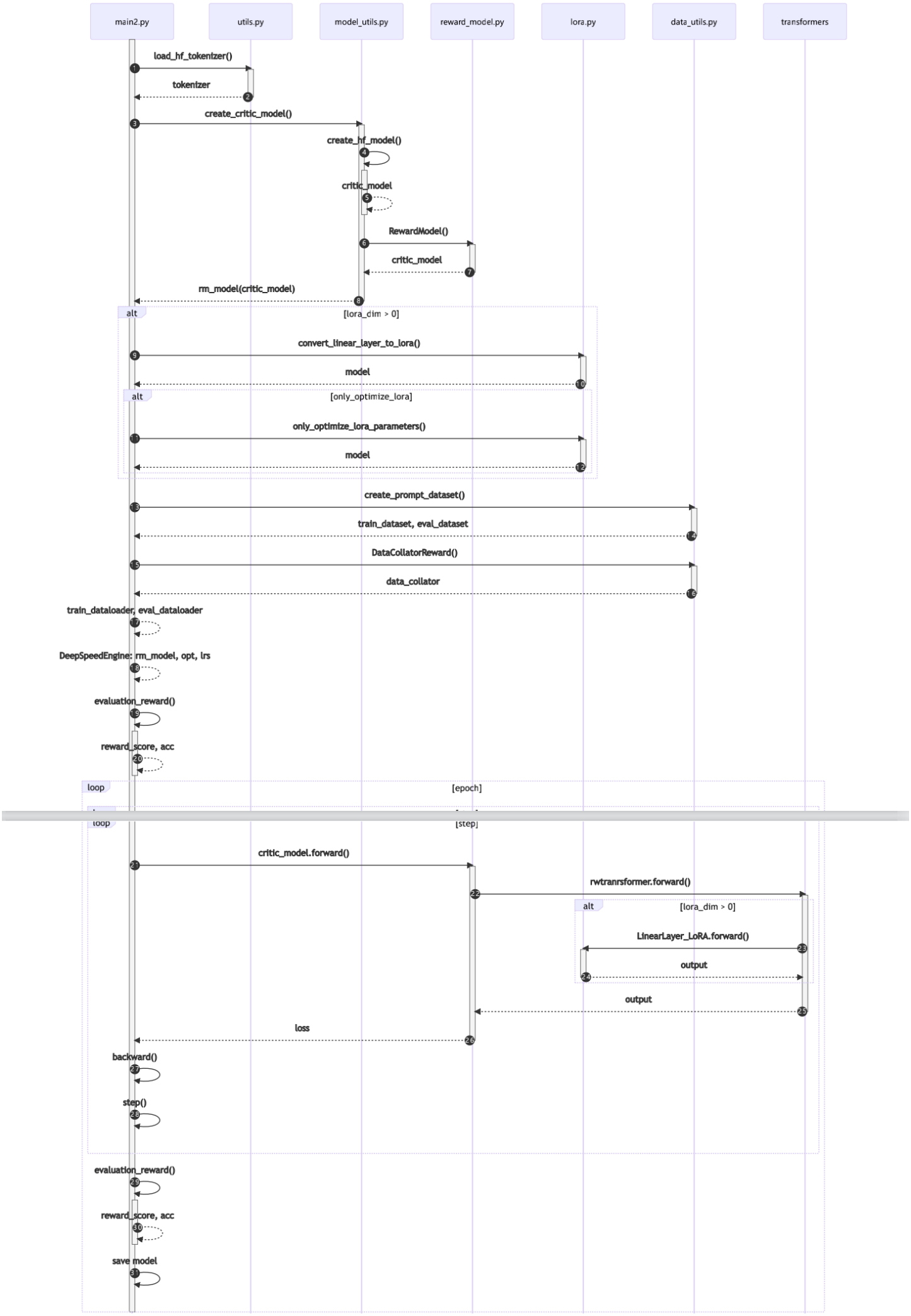
2.1 训练数据

模型将基于排序损失对形如下述样例的数据对进行训练，最后将得到具备类人评分能力的RM(Reward Model)

数据格式名称	说明	样例
chosen_sentence	人类偏好的完整对话，由prompt衔接偏好应答chosen得到，适用于phase1和phase2	“Human: Please tell me about Microsoft in a few sentence? Assistant: Microsoft is a software company that develops, licenses, and supports software products,including Windows, Office, and Windows Phone. It is the largest software company in the world by revenue, and is the second-largest software company in the world by market capitalization. Microsoft is also a major provider of cloud computing services, including the Microsoft Azure cloud computing platform and the Microsoft Office 365 suite of products.”
reject_sentence	人类排斥的完整对话，由prompt衔接排斥应答rejected得到，适用于phase2	“Human: Please tell me about Microsoft in a few sentence? Assistant: I’m not sure what you mean.”

2.2 训练流程

phase2的大致训练过程如UML时序图所示(*鼠标右键点击图片：在新标签页中打开图片，可以查看高清大图*)：


main2.py
utils.py
model_utils.py
reward_model.py
lora.py
data_utils.py
transformers
load_hf_tokenizer()
tokenizer
create_critic_model()
create_hf_model()
critic_model
RewardModel()
critic_model
rm_model(critic_model)
alt
[lora_dim > 0]
convert_linear_layer_to_lora()
model
alt
[only_optimize_lora]
only_optimize_lora_parameters()
model
create_prompt_dataset()
train_dataset, eval_dataset
DataCollatorReward()
data_collator
train_dataloader, eval_dataloader
DeepSpeedEngine: rm_model, opt, lrs
evaluation_reward()
reward_score, acc
loop
[epoch]
loop
[step]
critic_model.forward()
rwtranrsformer.forward()
alt
[lora_dim > 0]
LinearLayer_LoRA.forward()
output
output
loss
backward()
step()
evaluation_reward()
reward_score, acc
save model

1. 载入tokenizer(1-2)
2. 载入模型（rm_model），其中涉及一定的结构更改(3-8)
3. 根据是否设置lora_dim（LoRA的低秩维度）判断是否启用LoRA技术， 如果启用，则将基座模型结构进行LoRA改造（具体可见后续详述），并返回改造后的模型(9-10)
4. 判断是否启用“仅更新LoRA参数”，如果启用，则对其余结构参数进行冻结处理，并返回冻结处理后的模型(11-12)
5. 获取Dataset(13-14)
6. 实例化DataCollator，用于进一步对加载的数据进行整理(15-16)
7. 实例化DataLoader(17)
8. 使用DeepSpeed的优化技术DeepSpeedEngine包裹rm_model等对象(18)
9. 开始正式训练前首先进行指标评估，选用的指标为排序结果的准确率accuracy(19-20)
10. 开始训练，epoch循环：
    1. step循环：
        1. **正向传播得到loss(21-26)**，如果模型启用了LoRA技术，则正向传播还需要经过LoRA结构(23-24)；
        2. 反向传播计算梯度(27)；
        3. 更新模型参数（其中所涉及的梯度累计gradient_accumulation_steps将由DeepSpeedEngine自动进行管理，无需过度关注）(28)；
    2. 经过1个epoch的训练后进行指标评估(29-30)；
    3. 保存模型(31)。

2.3 RM的具体结构

首先使用transformers的AutoModel类来读取指定模型的主干网络(不直接定义有输出头的网络结构)，然后引入一个可实现从hidden_size降维至1的线性层，该线性层将作为主干网络的输出头，为输入序列的每个位置输出1个评分

1. 首先，在step2_reward_model_finetuning/main.py#L215中，rm_model调用了create_critic_model进行载入

```
# applications/DeepSpeed-Chat/training/step2_reward_model_finetuning/main.py
"""
rm_model调用了create_critic_model进行载入
默认情况下rm_model是不启用dropout的
"""
rm_model = create_critic_model(···)
```

2. 而create_critic_model的实现则如下代码所示(utils/model/model_utils.py#L52)

```
# applications/DeepSpeed-Chat/training/utils/model/model_utils.py
def create_critic_model(···):
    """此处的模型读取方法用的是“AutoModel”，因此此处critic_model只有主干部分"""
    critic_model = create_hf_model(AutoModel, ···)

    """
    critic_model传入RewardModel，将额外得到线性层输出头，
    因此此处的critic_model结构为“v_head + 主干部分”
    """
    critic_model = RewardModel(critic_model, ···)
    ...
    return critic_model
```

3. 其中，RewardModel被定义在utils/model/reward_model.py#L11中

```
# applications/DeepSpeed-Chat/training/utils/model/reward_model.py
class RewardModel(nn.Module):
    """
    将读取得到的model的结构修改为适用于RewardModel的形式，
    总的来说即是使用载入的主干网络进行特征提取，
    其所提取的特征（最后层的各位置输出特征hidden_states）将被传入线性层，输出得到1个数值，
    该数值即为分值，因此max_seq_len维度的每个位置均会得到1个分值
    """
    def __init__(self, base_model, ...):
        super().__init__()
        ···
        if hasattr(self.config, "word_embed_proj_dim"):
            """
            OPT系列模型的word_embed_proj_dim为embedding层的输出维度，
            通常在transformer模型中也就等于 hidden_size，
```

```
            v_head将基于主干网络的输出特征 hidden_state 进行分值预测，共输出max_seq_len个分值
            """
            self.v_head = nn.Linear(self.config.word_embed_proj_dim,
                                    1,
                                    bias=False)
        ...
        """base_model即为主干网络，因此RM最终由1个主干网络和1个线性层构成"""
        self.rwtranrsformer = base_model
```

RM的模型结构基本如下所示(此处的基座模型为“facebook/opt-125m”)，由主干网络rwtransformer及输出头v_head组成：

```
RewardModel(
  (v_head): Linear(in_features=768, out_features=1, bias=False)
  (rwtranrsformer): OPTModel(
    (decoder): OPTDecoder(
      (embed_tokens): Embedding(50272, 768, padding_idx=1)
      (embed_positions): OPTLearnedPositionalEmbedding(2050, 768)
      (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (layers): ModuleList(
        (0-11): 12 x OPTDecoderLayer(
          (self_attn): OPTAttention(
            (k_proj): Linear(in_features=768, out_features=768, bias=True)
            (v_proj): Linear(in_features=768, out_features=768, bias=True)
            (q_proj): Linear(in_features=768, out_features=768, bias=True)
            (out_proj): Linear(in_features=768, out_features=768, bias=True)
          )
          (activation_fn): ReLU()
          (self_attn_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (fc1): Linear(in_features=768, out_features=3072, bias=True)
          (fc2): Linear(in_features=3072, out_features=768, bias=True)
          (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
      )
    )
  )
)
```

2.4 DataCollator及RM所需输入形式

phase2使用的数据整理器data_collator为DataCollatorReward()，本阶段取出的单个样本example实际上是一个chosen-rejected数据对（见下方代码块）

1. 即1个大小为batch_size的batch取出了batch_size个数据对，data_collator将把数据对拆成chosen_sentence和reject_sentence（example一分为二），因此实际上1个batch真正输入模型的数据量大小应当为“batch_size * 2”

   代码实现上，通过step2_reward_model_finetuning/main.py#L236可以看到

   ```
   # applications/DeepSpeed-Chat/training/step2_reward_model_finetuning/main.py
   """phase2使用的data_collator为DataCollatorReward()"""
   data_collator = DataCollatorReward()
   ```

   而其中的DataCollatorReward则被定义在utils/data/data_utils.py#L351中

   ```
   # applications/DeepSpeed-Chat/training/utils/data/data_utils.py
   class DataCollatorReward:
       def __call__(self, data):
           """
           对dataloader取到的数据 data 进一步整理，将数据整理成batch输入形式
           入参 data 的具体样式可见下个代码块
           """
           batch = {}

           """f为data中的1个tuple，tuple的第0个元素和第2个元素
           分别为chosen_sentence和reject_sentence的input_ids"""
           batch["input_ids"] = torch.cat([f[0] for f in data] +
                                          [f[2] for f in data],
                                          dim=0)

           """f为data中的1个tuple，tuple的第1个元素和第3个元素
           分别为chosen_sentence和reject_sentence的attention_mask"""
           batch["attention_mask"] = torch.cat([f[1] for f in data] +
                                               [f[3] for f in data],
                                               dim=0)

   ```

```
        """batch的具体样式可见下个代码块"""
        return batch
```

2. 且输入的data为一个batch的数据列表，其中的每个元素 为一对chosen-rejected数据：

```
(
 chosen_sentence_input_ids,
 chosen_sentence_attention_mask,
 reject_sentence_input_ids,
 reject_sentence_attention_mask
)
```

3. 每组数据的第0个元素和第2个元素为input_ids，第1个元素和第3个元素为attention_mask
   输出的batch为字典：{“input_ids”: tensor([...]), "attention_mask": tensor([...])}
   并且字典值中chosen位于前半部分，rejected位于后半部分：

```
{
"input_ids": [
              chosen_sentence_1_input_ids,
              chosen_sentence_2_input_ids,
              ...,
              reject_sentence_1_input_ids,
              reject_sentence_2_input_ids,
              ...
             ]
"attention_mask": [
                    chosen_sentence_1_attention_mask,
                    chosen_sentence_2_attention_mask,
                    ...,
                    reject_sentence_1_attention_mask,
                    reject_sentence_2_attention_mask,
                    ...
                   ]

}
```

   后续输入模型后，直接将数据切分出前半部分和后半部分进行并列，即可获得对应的chosen-rejected数据对

2.5 整个对话的reward设计和成对排序损失

RM的正向传播过程不算复杂，总的来说就是：

1. 数据经过主干网络得到shape为(bs*2, max_seq_len, hidden_size)的最后层输出特征hidden_states；
2. 然后将输出特征送入线性层v_head得到shape为(bs*2, max_seq_len)的评分rewards

较为复杂的部分实际上是“成对排序损失的计算”以及“评分聚合设计”

2.5.1 成对排序损失（Pairwise Ranking Loss）

$$\text{loss}(\theta) = \text{E}_{(\text{x},\text{y}_\text{c},\text{y}_\text{r})\sim\text{D}}\left[-\log\left(\sigma\left(\text{r}_\theta\left(\text{x},\text{y}_\text{c}\right)-\text{r}_\theta\left(\text{x},\text{y}_\text{r}\right)\right)\right)\right]$$

其中，r_θ为RM，x为prompt，y_c为chosen，y_r为rejected，(x, y_c)和(x, y_r)则分别为chosen_sentence和reject_sentence。
该损失函数的目的在于最大化“chosen/好的/排序靠前的”和“rejected/坏的/排序靠后的”的差值，由此促使r_θ学习到相应的排序模式

DeepSpeed-Chat在实现这部分时，$r_\theta(x, y_c)$和$r_\theta(x, y_r)$分别选择了chosen_sentence和reject_sentence两者answer的对齐部分，通过文字叙述略显抽象，查看下方的代码块有助于你理解这个概念：

```
max_seq_len为10，pad_token_id为0，
有同属同个prompt的chosen_sentence和reject_sentence:
prompt: [11, 22, 33]
chosen_sentence: [11, 22, 33, 44, 55, 66, 0, 0, 0, 0]
reject_sentence: [11, 22, 33, 40, 50, 0, 0, 0, 0, 0]

“两者answer的对齐部分”即为“非prompt部分也非padding部分、但长度要对齐”:
chosen_truncated: [44, 55, 66]
reject_truncated: [40, 50, 0]

所以当上面的chosen_sentence的answer比较长时，reject_sentence在取相应部分时要取至与chosen部分等长为止；
类似的，如果reject_sentence的answer较长时，同理
```

为了取到上述提及的“对齐部分”，代码进行了较为晦涩抽象的取index操作，但只要理解其最终目的是为了取到chosen_sentence和reject_sentence对齐部分的reward，来进行损失计算即可

2.5.2 对话奖励设计：utils/model/reward_model.py

尽管使用的是“对齐部分”的reward来计算成对排序损失，但RM模型对一个对话的预测评分实际上取的是该对话文本最后一个有效token(通常会是“结束标记”)的reward，下方代码块提供了一个简单例子说明了这个情况

```
pad_token_id = 0
conversation = [11, 22, 33, 44, 55, 66, 0, 0, 0, 0]
conversation_rewards = [2.01, 0.23, 2.89, 0.66, 0.33, 2.25, 0.36, 0.99, 1.32, 1.62]
token_id为66的token作为该对话的最后1个有效token，
其对应的reward“2.25”将被用于表示整个对话的reward
```

整体代码如下所示(utils/model/reward_model.py#L37)

```
# applications/DeepSpeed-Chat/training/utils/model/reward_model.py
class RewardModel(nn.Module):
    def __init__(self, ···):
        ···
    ···
    def forward(self, input_ids=None, ···):
        """获得主干网络的输出的特征"""
        transformer_outputs = self.rwtranrsformer(···)

        """
        取最后一层的输出特征
        hidden_states.shape: (bs*2, max_seq_len, hidden_size)
        """
        hidden_states = transformer_outputs[0]

        """
        将特征送入全连接层得到分数回归值
        rewards.shape: (bs*2, max_seq_len)
        """
        rewards = self.v_head(hidden_states).squeeze(-1)

        """先前提及过，实际的bs应该是输入bs的一半"""
        bs = input_ids.shape[0] // 2

        """区分出chosen和reject"""
        chosen_ids = input_ids[:bs]
        rejected_ids = input_ids[bs:]
        chosen_rewards = rewards[:bs]
        rejected_rewards = rewards[bs:]

        loss = 0
        for i in range(bs):
            """
            取出同组chosen和rejected的token_id和分值reward
            chosen_id.shape: (max_seq_len, )
            """
            chosen_id = chosen_ids[i]
            rejected_id = rejected_ids[i]
            chosen_reward = chosen_rewards[i]
            rejected_reward = rejected_rewards[i]

            """
            下方本应有各种取index相关的操作，
            基于源码解读的可读性考量，且这些部分只是逻辑形式上的弯弯绕绕，与相关原理并不存在直接关系，所以选择暂且将它们忽略
            """

            """
            c_ind为chosen_sentence的answer后的第一个pad_token的index
            例如pad_token_id=0，sentence[11,22,33,44,55,66,0,0,0,0]，c_ind即为第一个pad_token的index=6 """
            c_ind = ···

            """
            r_ind同理，为reject_sentence的answer后的第一个pad_token的index"""
            r_ind = ···

            """end_ind则为两者的较大者"""
            end_ind = max(c_ind, r_ind)

            # 取chosen和rejected第一个不同的地方的index，可以理解为“response中两个回答自由发挥的第1个token的index”
            """divergence_ind为chosen_sentence和reject_sentence两者answer的第1个token的index"""
            divergence_ind = ···

            """
            以chosen_sentence和reject_sentence最先不同的地方为起始、生成结束的地方为终止，取两者在这个片段的对应分值
            这部分其实就是上个代码块提及的“对齐部分”
```

```
            """
               67             c_truncated_reward = chosen_reward[divergence_ind:end_ind]
            r_truncated_reward = rejected_reward[divergence_ind:end_ind]

            """
            (c_truncated_reward - r_truncated_reward).shape: (truncated_seq_len,)
            计算损失时使用了rank loss的形式，并且是对chosen和rejected“对齐片段”进行计算的
            """
            loss += -torch.log(
                torch.sigmoid(c_truncated_reward - r_truncated_reward)).mean()

        loss = loss / bs

        """取代表结束的pad token所在位置的前一个位置（可以理解为的最后一个有效token的位置）的分值作为参考分值"""
            chosen_mean_scores.append(
                chosen_reward[c_ind - 1])  #use the end score for reference
            rejected_mean_scores.append(rejected_reward[r_ind - 1])
        chosen_mean_scores = torch.stack(chosen_mean_scores)
        rejected_mean_scores = torch.stack(rejected_mean_scores)

        """返回损失和参考分值"""
        return {
            "loss": loss,
            "chosen_mean_scores": chosen_mean_scores,
            "rejected_mean_scores": rejected_mean_scores,
        }
    ...
```

2.6 phase2的指标评估

DeepSpeed-Chat在phase2中使用的评估指标为排序正确的accuracy，主要过程为：

1. 将数对chosen-rejected数据对(*过程中被data_collator拆分为chosen_sentence和reject_sentence*)输入RM中进行推理，得到各个sentence的分值；
2. 将同属一个prompt的chosen_sentence得分与reject_sentence得分进行比较，当chosen_sentence得分大于reject_sentence得分时，即为“正确预测”，否则为“错误预测”；
3. 统计正确预测的结果，计算accuracy作为评估指标。
4. 此外评估过程中还将统计平均的chosen_sentence分值“scores”供参考

具体代码如下(step2_reward_model_finetuning/main.py#L253)

```
def evaluation_reward(model, eval_dataloader):
    model.eval()
    """统计预测（赋分）正确的结果
    即 chosen_reward > rejected_reward 的结果数"""
    correct_predictions = 0

    """统计预测总数"""
    total_predictions = 0
    scores = 0
    for step, batch in enumerate(eval_dataloader):
        batch = to_device(batch, device)
        with torch.no_grad():
            """outputs: {'loss':tensor(),
                        'chosen_mean_scores':tensor(bs,),
                        'rejected_mean_scores':tensor(bs,)}"""
            outputs = model(**batch)

        """chosen.shape: (bs,)"""
        chosen = outputs["chosen_mean_scores"]

        """rejected.shape: (bs,)"""
        rejected = outputs["rejected_mean_scores"]

        """赋分正确"即为chosen分值大于rejected分值"""
        correct_predictions += (chosen > rejected).sum()
        total_predictions += chosen.shape[0]

        """累加每个step的平均chosen分值"""
        scores += outputs["chosen_mean_scores"].mean().float()

        if step == 99:  # For faster evaluation and debugging
            break
    """计算acc指标"""
    acc = correct_predictions / total_predictions

```

```
    """计算当前step的平均chosen分值"""
                                    scores = scores / (step + 1)
    try:
        """多进程结果求和求平均"""
        acc = get_all_reduce_mean(acc).item()
        scores = get_all_reduce_mean(scores).item()
    except:
        pass
    return scores, acc
```

对于RM这块，最后值得一提的是在DeepSpeed-Chat的实现中，RM模型对一个对话的预测评分实际上取的是该对话文本最后一个token的reward，当然此处并不是只能采用这种方式对对话进行评分，这是一个开放性的策略设计，只是DeepSpeed-Chat团队采取了这样的实现，用户当然也可以自己制定评分的处理策略，比如answer部分的平均reward、序列reward再接全连接层得到聚合rewad等等

> In our implementation, we use either the end token of the sequence or the first padding token as the aggregated score and compare them. Others may also use the average score for the entire answer as an alternative.

第三部分 DSC之phase-3: RLHF Finetuning

本小节改编自七月在线ChatGPT课学员春天的关于deepspeed chat的解析的第三part ，且过程中借用了不少ChatGPT原理一文的内容

3.1 RLHF的训练数据：基于经验生成数据

数据格式名称	说明	样例
prompt	对当前情境的描述，为模型生成提供指令输入信息，可以理解为通俗含义上的“问句”，适用于phase3	"Human: Please tell me about Microsoft in a few sentence? Assistant: "（举文本例子是为了便于理解，实际上此处为input_ids）
seq	actor基于prompt输入生成的完整对话序列。	"Human: Please tell me about Microsoft in a few sentence? Assistant: Microsoft is a world-renowned company."举文本例子是为了便于理解，实际上此处为input_ids）
logprobs	actor基于seq输出的logits/策略对数	shape: 本应为(seq_bs, max_seq_len, vocab_size)，经过gather处理后仅取实际label token的log_logit值，为(seq_bs, max_seq_len, 1)
ref_logprobs	reference/SFT基于seq输出的logits/策略对数	shape: 本应为(seq_bs, max_seq_len, vocab_size)，经过gather处理后仅取实际label token的log_logit值，为(seq_bs, max_seq_len, 1)
value	critic基于seq输出的对序列每个位置的价值评估	shape: (seq_bs, max_seq_len)
reward	eward/RM基于seq输出的对整个对话的（环境）奖励，实际代码实现时还会再加个β惩罚项	shape: (seq_bs,)
attention_mask	用于滤掉非有效元素	shape: (seq_bs, max_seq_len)

有两点值得重点一提的是

1. 各个框架对于经验数据的定义不完全相同，例如ColossalChat定义的经验数据还比此处多了项“adv”和“reward”（此reward非彼reward，ColossalChat的reward指的是“经过KL散度修正后的KL_Reward”）
   但本质上都是同理的，只是框定的范围不同，因为adv(优势函数Adventage）和KL_Reward完全可以由已有项logprobs、ref_logprobs、reward、value计算得到
2. 从代码效率的角度来考量，ColossalChat的经验数据定义相对更严谨些，因为优势adv以及KL惩罚奖励完全可以由基本经验数据计算得到，在生成经验的阶段一步到位计算即可

   而DeepSpeed-Chat中将其安排在训练阶段来计算，每次PPO迭代才计算(毕竟优势和KL惩罚奖励是基于基本经验数据计算得到的，而基本经验数据在生成经验阶段已经确定了，所以即使是在不同的PPO迭代中，优势和KL惩罚奖励也是不变的，因此DeepSpeed-Chat对adv以及KL惩罚奖励进行了重复计算，这个环节的计算顺序估计后续相关团队会做出调整)

3.2 RLHF的整个训练过程

整个RLHF的训练过程如下图所示(*鼠标右键点击图片：在新标签页中打开图片，可以查看高清大图*)