图解分布式训练（六）—— Pytorch的 DeepSpeed 详细解析

来自： AiGC面试宝典

宁静致远

2023年12月24日 00:39

动机

最常见的深度学习框架应该是TensorFlow、Pytorch、Keras，但是这些框架在面向大规模模型的时候都不是很方便。
比如Pytorch的分布式并行计算框架（Distributed Data Parallel，简称DDP），它也仅仅是能将数据并行，放到各个GPU的模型上进行训练。
也就是说，DDP的应用场景在你的模型大小大于显卡显存大小时，它就很难继续使用了，除非你自己再将模型参数拆散分散到各个GPU上。
今天要给大家介绍的DeepSpeed，它就能实现这个拆散功能，它通过将模型参数拆散分布到各个GPU上，以实现大型模型的计算，弥补了DDP的缺点，非常方便，这也就意味着我们能用更少的GPU训练更大的模型，而且不受限于显存。

一、为什么需要 Deepspeed?

大模型（LLM）在训练时往往需要大量内存来存储中间激活、权重等参数，百亿模型甚至无法在单个 GPU上进行训练，使得模型训练在某些情况下非常低效和不可能。这就需要进行多卡，或者多节点分布式训练。
在大规模深度学习模型训练中有个主要范式：

- 数据并行
- 模型并行

目前训练超大规模语言模型技术路线：GPU + PyTorch + Megatron-LM + DeepSpeed
DeepSpeed是由Microsoft提供的分布式训练工具，旨在支持更大规模的模型和提供更多的优化策略和工具。**与其他框架相比，DeepSpeed支持更大规模的模型和提供更多的优化策略和工具。其中，主要优势在于支持更大规模的模型、提供了更多的优化策略和工具（例如 ZeRO 和 Offload 等）**

1. 用 3D 并行化实现万亿参数模型训练。**DeepSpeed 实现了三种并行方法的灵活组合：ZeRO 支持的数据并行，流水线并行和张量切片模型并行。3D 并行性适应了不同工作负载的需求，以支持具有万亿参数的超大型模型**，同时**实现了近乎完美的显存扩展性和吞吐量扩展效率。**此外，**其提高的通信效率使用户可以在网络带宽有限的常规群集上以 2-7 倍的速度训练有数十亿参数的模型。**
2. ZeRO-Offload 使 GPU 单卡能够训练 10 倍大的模型： 为了同时利用 CPU 和 GPU 内存来训练大型模型，我们扩展了 ZeRO-2。我们的用户在使用带有单张英伟达 V100 GPU 的机器时，可以在不耗尽显存的情况下运行多达 130 亿个参数的模型，模型规模扩展至现有方法的10倍，并保持有竞争力的吞吐量。此功能使数十亿参数的模型训练更加大众化，并为许多深度学习从业人员打开了一扇探索更大更好的模型的窗户。
3. 通过 DeepSpeed Sparse Attention 用6倍速度执行10倍长的序列： DeepSpeed提供了稀疏 attention kernel ——一种工具性技术，可支持长序列的模型输入，包括文本输入，图像输入和语音输入。与经典的稠密 Transformer 相比，它支持的输入序列长一个数量级，并在保持相当的精度下获得最高 6 倍的执行速度提升。它还比最新的稀疏实现快 1.5–3 倍。此外，我们的稀疏 kernel 灵活支持稀疏格式，使用户能够通过自定义稀疏结构进行创新。
4. 1 比特 Adam 减少 5 倍通信量： Adam 是一个在大规模深度学习模型训练场景下的有效的（也许是最广为应用的）优化器。然而，它与通信效率优化算法往往不兼容。因此，在跨设备进行分布式扩展时，通信开销可能成为瓶颈。我们推出了一种 1 比特 Adam 新算法，以及其高效实现。该算法最多可减少 5 倍通信量，同时实现了与Adam相似的收敛率。在通信受限的场景

下，我们观察到分布式训练速度提升了 3.5 倍，这使得该算法可以扩展到不同类型的 GPU 群集和网络环境。

二、DeepSpeed 基本概念 介绍一下?

2.1 DeepSpeed 介绍

- 在分布式计算环境中，需要理解几个非常基础的概念：节点编号、全局进程编号、局部进程编号、全局总进程数和主节点。其中，主节点负责协调所有其他节点和进程的工作，因此是整个系统的关键部分。
- DeepSpeed 还提供了 mpi、gloo 和 nccl 等通信策略，可以根据具体情况进行选择和配置。在使用 DeepSpeed 进行分布式训练时，可以根据具体情况选择合适的通信库，例如在 CPU 集群上进行分布式训练，可以选择 mpi 和 gloo；如果是在 GPU 上进行分布式训练，可以选择 nccl。
- ZeRO（Zero Redundancy Optimizer）是一种用于大规模训练优化的技术，主要是用来减少内存占用。ZeRO 将模型参数分成了三个部分：Optimizer States、Gradient 和 Model Parameter。在使用 ZeRO 进行分布式训练时，可以选择 ZeRO-Offload 和 ZeRO-Stage3 等不同的优化技术。
- 混合精度训练是指在训练过程中同时使用FP16（半精度浮点数）和FP32（单精度浮点数）两种精度的技术。使用FP16可以大大减少内存占用，从而可以训练更大规模的模型。在使用混合精度训练时，需要使用一些技术来解决可能出现的梯度消失和模型不稳定的问题，例如动态精度缩放和混合精度优化器等。
- 结合使用huggingface和deepspeed

2.2 DeepSpeed 基础的概念

在分布式计算环境中，有几个非常基础的概念需要理解：

- 节点编号（node_rank:）：分配给系统中每个节点的唯一标识符，用于区分不同计算机之间的通信。
- 全局进程编号（rank）：分配给整个系统中的每个进程的唯一标识符，用于区分不同进程之间的通信。
- 局部进程编号（local_rank）：分配给单个节点内的每个进程的唯一标识符，用于区分同一节点内的不同进程之间的通信。
- 全局总进程数（word_size）：在整个系统中运行的所有进程的总数，用于确定可以并行完成多少工作以及需要完成任务所需的资源数量。
- 主节点（master_ip+master_port）：在分布式计算环境中，主节点负责协调所有其他节点和进程的工作，为了确定主节点，我们需要知道它的IP地址和端口号。主节点还负责监控系统状态、处理任务分配和结果汇总等任务，因此是整个系统的关键部分。

2.3 DeepSpeed 支持的功能

- DeepSpeed目前支持的功能
    - Optimizer state partitioning (ZeRO stage 1)
    - Gradient partitioning (ZeRO stage 2)
    - Parameter partitioning (ZeRO stage 3)
    - Custom mixed precision training handling
    - A range of fast CUDA-extension-based optimizers
    - ZeRO-Offload to CPU and NVMe

三、DeepSpeed 通信策略 介绍一下?

deepspeed 还提供了 mpi、gloo 和 nccl 等通信策略，可以根据具体情况进行选择和配置。

- mpi 是一种跨节点通信库，常用于 CPU 集群上的分布式训练；
- gloo 是一种高性能的分布式训练框架，支持 CPU 和 GPU 上的分布式训练；
- nccl 是 NVIDIA 提供的 GPU 专用通信库，被广泛应用于 GPU 上的分布式训练。

在使用 DeepSpeed 进行分布式训练时，可以根据具体情况选择合适的通信库。通常情况下，如果是在 CPU 集群上进行分布式训练，可以选择 mpi 和 gloo；如果是在 GPU 上进行分布式训练，可以选择 nccl。

```
    export CUDA_LAUNCH_BLOCKING=1
```

四、DeepSpeed 如何使用?

4.1 DeepSpeed 安装

```
    $ pip install deepspeed==0.8.1
    $ sudo apt-get update
    $ sudo apt-get install openmpi-bin libopenmpi-dev
    $ pip install mpi4py
```

4.2 DeepSpeed 使用

注：使用DeepSpeed其实和写一个pytorch模型只有部分区别，一开始的流程是一样的。

1. 导包

```
    import json, time, random
    import torch
    from sklearn.metrics import classification_report
    from torch.utils.data import DataLoader
    from collections import Counter
    from transformers import BertForMaskedLM, BertTokenizer,
BertForSequenceClassification, BertConfig, AdamW
    import torch.nn as nn
    import numpy as np

    # 导入 deepspeed 分布式训练包
    import deepspeed
    import torch.distributed as dist

    # 定义 设置随机种子
    def set_seed(seed=123):
        """
        设置随机数种子，保证实验可重现
        :param seed:
        :return:
        """
```

```
        random.seed(seed)
        torch.manual_seed(seed)
        np.random.seed(seed)
        torch.cuda.manual_seed_all(seed)
```

2. 定义 获取 和 加载 训练集 函数

```
    def get_data():
        with open("data/train.json", "r", encoding="utf-8") as fp:
            data = fp.read()
        data = json.loads(data)
        return data

    def load_data():
        data = get_data()
        return_data = []
        # [(文本,  标签id)]
        for d in data:
            text = d[0]
            label = d[1]
            return_data.append(("".join(text.split(" ")).strip(), label))
        return return_data
```

3. 定义 训练集编码类

```
class Collate:
    def __init__(
            self,
            tokenizer,
            max_seq_len,
        ):
        self.tokenizer = tokenizer
        self.max_seq_len = max_seq_len

    def collate_fn(self, batch):
        input_ids_all = []
        token_type_ids_all = []
        attention_mask_all = []
        label_all = []
        for data in batch:
            text = data[0]
            label = data[1]
            inputs = self.tokenizer.encode_plus(
                text=text,
                max_length=self.max_seq_len,
                padding="max_length",
                truncation="longest_first",
                return_attention_mask=True,
                return_token_type_ids=True
```

```
            )
            input_ids = inputs["input_ids"]
            token_type_ids = inputs["token_type_ids"]
            attention_mask = inputs["attention_mask"]
            input_ids_all.append(input_ids)
            token_type_ids_all.append(token_type_ids)
            attention_mask_all.append(attention_mask)
            label_all.append(label)

        input_ids_all = torch.tensor(input_ids_all, dtype=torch.long)
        token_type_ids_all = torch.tensor(token_type_ids_all, dtype=torch.long)
        attention_mask_all = torch.tensor(attention_mask_all, dtype=torch.long)
        label_all = torch.tensor(label_all, dtype=torch.long)
        return_data = {
            "input_ids": input_ids_all,
            "attention_mask": attention_mask_all,
            "token_type_ids": token_type_ids_all,
            "label": label_all
        }
        return return_data
```

4. 定义 Trainer 训练类 【注：这里有部分改动】

```
class Trainer:
    def __init__(
                self,
                args,
                config,
                model_engine,
                criterion,
                optimizer
            ):
        self.args = args
        self.config = config
        self.model_engine = model_engine
        self.criterion = criterion
        self.optimizer = optimizer

    def on_step(self, batch_data):
        label = batch_data["label"].cuda()
        input_ids = batch_data["input_ids"].cuda()
        token_type_ids = batch_data["token_type_ids"].cuda()
        attention_mask = batch_data["attention_mask"].cuda()
        output = self.model_engine.forward(
            input_ids=input_ids,
            token_type_ids=token_type_ids,
            attention_mask=attention_mask,
            labels=label
```

```
        )
        logits = output[1]
        return logits, label

    # loss 聚合 计算
    def loss_reduce(self, loss):
        rt = loss.clone()
        dist.all_reduce(rt, op=dist.ReduceOp.SUM)
        rt /= torch.cuda.device_count()
        return rt

    # output 聚合
    def output_reduce(self, outputs, targets):
        output_gather_list = [torch.zeros_like(outputs) for _ in
range(torch.cuda.device_count())]
        # 把每一个GPU的输出聚合起来
        dist.all_gather(output_gather_list, outputs)

        outputs = torch.cat(output_gather_list, dim=0)
        target_gather_list = [torch.zeros_like(targets) for _ in
range(torch.cuda.device_count())]
        # 把每一个GPU的输出聚合起来
        dist.all_gather(target_gather_list, targets)
        targets = torch.cat(target_gather_list, dim=0)
        return outputs, targets

    def train(self, train_loader, dev_loader=None):
        gloabl_step = 1
        best_acc = 0.
        if self.args.local_rank == 0:
            start = time.time()
        for epoch in range(1, self.args.epochs + 1):
            for step, batch_data in enumerate(train_loader):
                self.model_engine.train()
                logits, label = self.on_step(batch_data)
                loss = self.criterion(logits, label)
                self.model_engine.backward(loss)
                self.model_engine.step()
                # loss 聚合 计算
                loss = self.loss_reduce(loss)
                if self.args.local_rank == 0:
                    print("【train】 epoch：{}/{} step：{}/{} loss：{:.6f}".format(
                        epoch, self.args.epochs, gloabl_step, self.args.total_step,
loss
                    ))
                gloabl_step += 1
                if self.args.dev:
```

```
                    if gloabl_step % self.args.eval_step == 0:
                        loss, accuracy = self.dev(dev_loader)
                        if self.args.local_rank == 0:
                            print("【dev】 loss：{:.6f} accuracy：
{:.4f}".format(loss, accuracy))
                        if accuracy > best_acc:
                            best_acc = accuracy
                            self.model_engine.save_checkpoint(self.args.ckpt_path,
save_latest=True)
                            if self.args.local_rank == 0:
                                print("【best accuracy】 {:.4f}".format(best_acc))


        if self.args.local_rank == 0:
            end = time.time()
            print("耗时：{}分钟".format((end - start) / 60))
        if not self.args.dev:
            self.model_engine.save_checkpoint(self.args.ckpt_path,
save_latest=True)

    def dev(self, dev_loader):
        self.model_engine.eval()
        correct_total = 0
        num_total = 0
        loss_total = 0.
        with torch.no_grad():
            for step, batch_data in enumerate(dev_loader):
                logits, label = self.on_step(batch_data)
                loss = self.criterion(logits, label)
                # loss 聚合 计算
                loss = self.loss_reduce(loss)
                # output 聚合
                logits, label = self.output_reduce(logits, label)
                loss_total += loss
                logits = logits.detach().cpu().numpy()
                label = label.view(-1).detach().cpu().numpy()
                num_total += len(label)
                preds = np.argmax(logits, axis=1).flatten()
                correct_num = (preds == label).sum()
                correct_total += correct_num

        return loss_total, correct_total / num_total

    def test(self, model, test_loader, labels):
        self.model_engine = model
        self.model_engine.eval()
        preds = []
```

```
        trues = []
        with torch.no_grad():
            for step, batch_data in enumerate(test_loader):
                logits, label = self.on_step(batch_data)
                # output 聚合
                logits, label = self.output_reduce(logits, label)
                label = label.view(-1).detach().cpu().numpy().tolist()
                logits = logits.detach().cpu().numpy()
                pred = np.argmax(logits, axis=1).flatten().tolist()
                trues.extend(label)
                preds.extend(pred)
        # print(trues, preds, labels)
        print(np.array(trues).shape, np.array(preds).shape)
        report = classification_report(trues, preds, target_names=labels)
        return report
```

5. Args 参数类定义

```
class Args:
    model_path = "/mnt/kaimo/data/pretrain/bert-base-chinese"
    ckpt_path = "output/deepspeed/"
    max_seq_len = 128
    ratio = 0.92
    epochs = 5
    eval_step = 50
    dev = False
    local_rank = None
```

6. 初始化DeepSpeed引擎

```
deepspeed_config = {
    "train_micro_batch_size_per_gpu": 32,
    "gradient_accumulation_steps": 1,
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": 3e-5
        }
    },
    "fp16": {
        "enabled": True
    },
    "zero_optimization": {
        "stage": 3,                    # ZeRO第3阶段
        "allgather_partitions": True,
        "allgather_bucket_size": 2e8,
        "overlap_comm": True,
        "reduce_scatter": True,
        "reduce_bucket_size": 2e8
```

```
    },
    "activation_checkpointing": {
        "partition_activations": True,
        "cpu_checkpointing": True,
        "contiguous_memory_optimization": True
    },
    "wall_clock_breakdown": True,
    "log_dist": False,
}
```

注：需要注意的是在ZeRO第3阶段，模型被划分到不同的GPU了

1. 主函数 main()

```
def main():
    # ======================================
    # 定义相关参数
    set_seed()
    label2id = {
        "其他": 0,
        "喜好": 1,
        "悲伤": 2,
        "厌恶": 3,
        "愤怒": 4,
        "高兴": 5,
    }
    args = Args()
    tokenizer = BertTokenizer.from_pretrained(args.model_path)
    # ======================================

    # ======================================
    # 加载数据集
    data = load_data()
    # 取1万条数据出来
    data = data[:10000]
    random.shuffle(data)
    train_num = int(len(data) * args.ratio)
    train_data = data[:train_num]
    dev_data = data[train_num:]

    collate = Collate(tokenizer, args.max_seq_len)
    train_loader = DataLoader(
        train_data,
        batch_size=deepspeed_config["train_micro_batch_size_per_gpu"],
        shuffle=True,
        num_workers=2,
        collate_fn=collate.collate_fn
    )
    total_step = len(train_loader) * args.epochs
```

```
    args.total_step = total_step
    dev_loader = DataLoader(
        dev_data,
        batch_size=deepspeed_config["train_micro_batch_size_per_gpu"],
        shuffle=False,
        num_workers=2,
        collate_fn=collate.collate_fn
    )
    test_loader = dev_loader
    # =====================================

    # =====================================
    # 定义模型、优化器、损失函数
    config = BertConfig.from_pretrained(args.model_path, num_labels=6)
    model = BertForSequenceClassification.from_pretrained(args.model_path,
config=config)
    model.cuda()
    criterion = torch.nn.CrossEntropyLoss()
    '''注： 这里需要用 deepspeed 初始化 model 和 optimizer'''
    model_engine, optimizer, _, _ = deepspeed.initialize(
        config=deepspeed_config,
        model=model,
        model_parameters=model.parameters()
    )

    args.local_rank = model_engine.local_rank
    # =====================================

    # =====================================
    # 定义训练器
    trainer = Trainer(
        args,
        config,
        model_engine,
        criterion,
        optimizer
    )

    # 训练和验证
    trainer.train(train_loader, dev_loader)
    # =====================================

    # =====================================
    # 测试
    labels = list(label2id.keys())
    config = BertConfig.from_pretrained(args.model_path, num_labels=6)
```

```
    model = BertForSequenceClassification.from_pretrained(args.model_path,
config=config)
    model.cuda()

    # 需要重新初始化引擎
    model_engine, optimizer, _, _ = deepspeed.initialize(
        config=deepspeed_config,
        model=model,
        model_parameters=model.parameters())
    model_engine.load_checkpoint(args.ckpt_path, load_module_only=True)
    report = trainer.test(model_engine, test_loader, labels)
    if args.local_rank == 0:
        print(report)
    # ====================================
```

2. 运行代码

开始运行代码：

```
    $ deepspeed test.py --deepspeed_config config.json
```

测试的时候发现每块GPU对每批数据都进行计算了一次，这里可能需要做些修改，暂时还没找到相关的方法。

```
[2023-06-28 00:58:12,759] [INFO] [engine.py:2824:_get_all_zero_checkpointe_dicts]
successfully read 2 ZeRO state_dicts for rank 0
[2023-06-28 00:58:13,005] [INFO] [engine.py:2774:_load_zero_checkpoint] lg 2 zero
partition checkpoints for rank 1
[2023-06-28 00:58:13,005] [INFO] [engine.py:2774:_load_zero_checkpoint] lg 2 zero
partition checkpoints for rank 0
(1600,) (1600,)
              precision    recall  f1-score   support

          其他       0.62      0.67      0.64       546
          喜好       0.50      0.57      0.53       224
          悲伤       0.49      0.39      0.44       228
          厌恶       0.42      0.42      0.42       240
          愤怒       0.58      0.48      0.53       124
          高兴       0.64      0.62      0.63       238

    accuracy                           0.56      1600
   macro avg       0.54      0.53      0.53      1600
weighted avg       0.55      0.56      0.55      1600

(1600,) (1600,)
```

1. 训练模型转化

在./output/deepspeed 生成 多GPU的模型：

```
    $ ls output/deepspeed/global_step1440/
    >>>
```

```
    zero_pp_rank_0_mp_rank_00_model_states.pt
    zero_pp_rank_0_mp_rank_00_optim_states.pt
    zero_pp_rank_1_mp_rank_00_model_states.pt
    zero_pp_rank_1_mp_rank_00_optim_states.pt
```

需要利用./output/deepspeed下有一个zero_to_fp32.py文件，我们可以利用将多GPU的模型转换为完整的：

```
    $ python zero_to_fp32.py pytorch-distributed/output/deepspeed/
./pytorch_model.bin
```

五、DeepSpeed 全部代码

```
import json
import time
import random
import torch
import deepspeed
import torch.nn as nn
import numpy as np
import torch.distributed as dist

from sklearn.metrics import classification_report
from torch.utils.data import DataLoader
from collections import Counter
from transformers import BertForMaskedLM, BertTokenizer,
BertForSequenceClassification, BertConfig, AdamW

# 定义 设置随机种子
def set_seed(seed=123):
    """
    设置随机数种子，保证实验可重现
    :param seed:
    :return:
    """
    random.seed(seed)
    torch.manual_seed(seed)
    np.random.seed(seed)
    torch.cuda.manual_seed_all(seed)


def get_data():
    with open("data/train.json", "r", encoding="utf-8") as fp:
        data = fp.read()
    data = json.loads(data)
    return data
```

```
def load_data():
    data = get_data()
    return_data = []
    # [(文本,  标签id)]
    for d in data:
        text = d[0]
        label = d[1]
        return_data.append(("".join(text.split(" ")).strip(), label))
    return return_data

class Collate:
    def __init__(
            self,
            tokenizer,
            max_seq_len,
        ):
        self.tokenizer = tokenizer
        self.max_seq_len = max_seq_len

    def collate_fn(self, batch):
        input_ids_all = []
        token_type_ids_all = []
        attention_mask_all = []
        label_all = []
        for data in batch:
            text = data[0]
            label = data[1]
            inputs = self.tokenizer.encode_plus(
                text=text,
                max_length=self.max_seq_len,
                padding="max_length",
                truncation="longest_first",
                return_attention_mask=True,
                return_token_type_ids=True
            )
            input_ids = inputs["input_ids"]
            token_type_ids = inputs["token_type_ids"]
            attention_mask = inputs["attention_mask"]
            input_ids_all.append(input_ids)
            token_type_ids_all.append(token_type_ids)
            attention_mask_all.append(attention_mask)
            label_all.append(label)

        input_ids_all = torch.tensor(input_ids_all, dtype=torch.long)
        token_type_ids_all = torch.tensor(token_type_ids_all, dtype=torch.long)
        attention_mask_all = torch.tensor(attention_mask_all, dtype=torch.long)
        label_all = torch.tensor(label_all, dtype=torch.long)
```

```
        return_data = {
            "input_ids": input_ids_all,
            "attention_mask": attention_mask_all,
            "token_type_ids": token_type_ids_all,
            "label": label_all
        }
        return return_data


class Trainer:
    def __init__(
                self,
                args,
                config,
                model_engine,
                criterion,
                optimizer
            ):
        self.args = args
        self.config = config
        self.model_engine = model_engine
        self.criterion = criterion
        self.optimizer = optimizer

    def on_step(self, batch_data):
        label = batch_data["label"].cuda()
        input_ids = batch_data["input_ids"].cuda()
        token_type_ids = batch_data["token_type_ids"].cuda()
        attention_mask = batch_data["attention_mask"].cuda()
        output = self.model_engine.forward(
            input_ids=input_ids,
            token_type_ids=token_type_ids,
            attention_mask=attention_mask,
            labels=label
        )
        logits = output[1]
        return logits, label

    # loss 聚合 计算
    def loss_reduce(self, loss):
        rt = loss.clone()
        dist.all_reduce(rt, op=dist.ReduceOp.SUM)
        rt /= torch.cuda.device_count()
        return rt

    # output 聚合
    def output_reduce(self, outputs, targets):
```

```
        output_gather_list = [torch.zeros_like(outputs) for _ in
range(torch.cuda.device_count())]
        # 把每一个GPU的输出聚合起来
        dist.all_gather(output_gather_list, outputs)

        outputs = torch.cat(output_gather_list, dim=0)
        target_gather_list = [torch.zeros_like(targets) for _ in
range(torch.cuda.device_count())]
        # 把每一个GPU的输出聚合起来
        dist.all_gather(target_gather_list, targets)
        targets = torch.cat(target_gather_list, dim=0)
        return outputs, targets

    def train(self, train_loader, dev_loader=None):
        gloabl_step = 1
        best_acc = 0.
        if self.args.local_rank == 0:
            start = time.time()
        for epoch in range(1, self.args.epochs + 1):
            for step, batch_data in enumerate(train_loader):
                self.model_engine.train()
                logits, label = self.on_step(batch_data)
                loss = self.criterion(logits, label)
                self.model_engine.backward(loss)
                self.model_engine.step()
                # loss 聚合 计算
                loss = self.loss_reduce(loss)
                if self.args.local_rank == 0:
                    print("【train】 epoch：{}/{} step：{}/{} loss：{:.6f}".format(
                        epoch, self.args.epochs, gloabl_step, self.args.total_step,
loss
                    ))
                gloabl_step += 1
                if self.args.dev:
                    if gloabl_step % self.args.eval_step == 0:
                        loss, accuracy = self.dev(dev_loader)
                        if self.args.local_rank == 0:
                            print("【dev】 loss：{:.6f} accuracy：
{:.4f}".format(loss, accuracy))
                        if accuracy > best_acc:
                            best_acc = accuracy
                            self.model_engine.save_checkpoint(self.args.ckpt_path,
save_latest=True)
                            if self.args.local_rank == 0:
                                print("【best accuracy】 {:.4f}".format(best_acc))
```

```
        if self.args.local_rank == 0:
            end = time.time()
            print("耗时：{}分钟".format((end - start) / 60))
        if not self.args.dev:
            self.model_engine.save_checkpoint(self.args.ckpt_path,
save_latest=True)

    def dev(self, dev_loader):
        self.model_engine.eval()
        correct_total = 0
        num_total = 0
        loss_total = 0.
        with torch.no_grad():
            for step, batch_data in enumerate(dev_loader):
                logits, label = self.on_step(batch_data)
                loss = self.criterion(logits, label)
                # loss 聚合 计算
                loss = self.loss_reduce(loss)
                # output 聚合
                logits, label = self.output_reduce(logits, label)
                loss_total += loss
                logits = logits.detach().cpu().numpy()
                label = label.view(-1).detach().cpu().numpy()
                num_total += len(label)
                preds = np.argmax(logits, axis=1).flatten()
                correct_num = (preds == label).sum()
                correct_total += correct_num

        return loss_total, correct_total / num_total

    def test(self, model, test_loader, labels):
        self.model_engine = model
        self.model_engine.eval()
        preds = []
        trues = []
        with torch.no_grad():
            for step, batch_data in enumerate(test_loader):
                logits, label = self.on_step(batch_data)
                # output 聚合
                logits, label = self.output_reduce(logits, label)
                label = label.view(-1).detach().cpu().numpy().tolist()
                logits = logits.detach().cpu().numpy()
                pred = np.argmax(logits, axis=1).flatten().tolist()
                trues.extend(label)
                preds.extend(pred)
        # print(trues, preds, labels)
        print(np.array(trues).shape, np.array(preds).shape)
```

```
        report = classification_report(trues, preds, target_names=labels)
        return report

class Args:
    model_path = "/mnt/kaimo/data/pretrain/bert-base-chinese"
    ckpt_path = "output/deepspeed/"
    max_seq_len = 128
    ratio = 0.92
    epochs = 5
    eval_step = 50
    dev = False
    local_rank = None

deepspeed_config = {
    "train_micro_batch_size_per_gpu": 32,
    "gradient_accumulation_steps": 1,
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": 3e-5
        }
    },
    "fp16": {
        "enabled": True
    },
    "zero_optimization": {
        "stage": 3,
        "allgather_partitions": True,
        "allgather_bucket_size": 2e8,
        "overlap_comm": True,
        "reduce_scatter": True,
        "reduce_bucket_size": 2e8
    },
    "activation_checkpointing": {
        "partition_activations": True,
        "cpu_checkpointing": True,
        "contiguous_memory_optimization": True
    },
    "wall_clock_breakdown": True,
    "log_dist": False,
}

def main():
    # ====================================
    # 定义相关参数
    set_seed()
    label2id = {
```

```
        "其他": 0,
        "喜好": 1,
        "悲伤": 2,
        "厌恶": 3,
        "愤怒": 4,
        "高兴": 5,
    }
    args = Args()
    tokenizer = BertTokenizer.from_pretrained(args.model_path)
    # ========================================

    # ========================================
    # 加载数据集
    data = load_data()
    # 取1万条数据出来
    data = data[:10000]
    random.shuffle(data)
    train_num = int(len(data) * args.ratio)
    train_data = data[:train_num]
    dev_data = data[train_num:]

    collate = Collate(tokenizer, args.max_seq_len)
    train_loader = DataLoader(
        train_data,
        batch_size=deepspeed_config["train_micro_batch_size_per_gpu"],
        shuffle=True,
        num_workers=2,
        collate_fn=collate.collate_fn
    )
    total_step = len(train_loader) * args.epochs
    args.total_step = total_step
    dev_loader = DataLoader(
        dev_data,
        batch_size=deepspeed_config["train_micro_batch_size_per_gpu"],
        shuffle=False,
        num_workers=2,
        collate_fn=collate.collate_fn
    )
    test_loader = dev_loader
    # ========================================

    # ========================================
    # 定义模型、优化器、损失函数
    config = BertConfig.from_pretrained(args.model_path, num_labels=6)
    model = BertForSequenceClassification.from_pretrained(args.model_path,
config=config)
    model.cuda()
```

```
    criterion = torch.nn.CrossEntropyLoss()
    '''注： 这里需要用 deepspeed 初始化 model 和 optimizer'''
    model_engine, optimizer, _, _ = deepspeed.initialize(
        config=deepspeed_config,
        model=model,
        model_parameters=model.parameters()
    )

    args.local_rank = model_engine.local_rank
    # ========================================

    # ========================================
    # 定义训练器
    trainer = Trainer(
        args,
        config,
        model_engine,
        criterion,
        optimizer
    )

    # 训练和验证
    trainer.train(train_loader, dev_loader)
    # ========================================

    # ========================================
    # 测试
    labels = list(label2id.keys())
    config = BertConfig.from_pretrained(args.model_path, num_labels=6)
    model = BertForSequenceClassification.from_pretrained(args.model_path,
config=config)
    model.cuda()

    # 需要重新初始化引擎
    model_engine, optimizer, _, _ = deepspeed.initialize(
        config=deepspeed_config,
        model=model,
        model_parameters=model.parameters())
    model_engine.load_checkpoint(args.ckpt_path, load_module_only=True)
    report = trainer.test(model_engine, test_loader, labels)
    if args.local_rank == 0:
        print(report)
    # ========================================

if __name__ == '__main__':
    main()
```

六、优化器和调度器

当不使用offload_optimizer 时，可以按照下表，混合使用HF和DS的优化器和迭代器，除了HF Scheduler和DS Optimizer这一种情况。

Combos	HF Scheduler	DS Scheduler
HF Optimizer	Yes	Yes
DS Optimizer	No	Yes

6.1 优化器

- 启用 offload_optimizer 时可以使用非 DeepSpeed 的优化器，只要它同时具有 CPU 和 GPU 的实现（LAMB 除外）。
- DeepSpeed 的主要优化器是 Adam、AdamW、OneBitAdam 和 Lamb。 这些已通过 ZeRO 进行了彻底测试，建议使用。
- 如果没有在配置文件中配置优化器参数，Trainer 将自动将其设置为 AdamW，并将使用命令行参数的默认值：--learning_rate、--adam_beta1、--adam_beta2、 --adam_epsilon 和 --weight_decay。
- 与 AdamW 类似，可以配置其他官方支持的优化器。 请记住，它们可能具有不同的配置值。 例如 对于 Adam，需要将 weight_decay 设置为 0.01 左右。
- 此外，offload在与 Deepspeed 的 CPU Adam 优化器一起使用时效果最佳。 如果想对offload使用不同的优化器，deepspeed==0.8.3 以后的版本，还需要添加：

```
{
   "zero_force_ds_cpu_optimizer": false
}
```

6.2 调度器

- DeepSpeed 支持 LRRangeTest、OneCycle、WarmupLR 和 WarmupDecayLR 学习率调度器。
- Transformers和DeepSpeed中调度器的overlap

```
   WarmupLR 使用 --lr_scheduler_type constant_with_warmup
   WarmupDecayLR 使用 --lr_scheduler_type linear
```

七、训练精度

- 由于 fp16 混合精度大大减少了内存需求，并可以实现更快的速度，因此只有在在此训练模式下表现不佳时，才考虑不使用混合精度训练。 通常，当模型未在 fp16 混合精度中进行预训练时，会出现这种情况（例如，使用 bf16 预训练的模型）。 这样的模型可能会溢出，导致loss为NaN。 如果是这种情况，使用完整的 fp32 模式。
- 如果是基于 Ampere 架构的 GPU，pytorch 1.7 及更高版本将自动切换为使用更高效的 tf32 格式进行某些操作，但结果仍将采用 fp32。
- 使用 Trainer，可以使用 --tf32 启用它，或使用 --tf32 0 或 --no_tf32 禁用它。 PyTorch 默认值是使用tf32。

7.1 自动混合精度

- fp16
    - 可以使用 pytorch-like AMP 方式或者 apex-like 方式
    - 使用 --fp16--fp16_backend amp 或 --fp16_full_eval 命令行参数时启用此模式
- bf16
    - 使用--bf16 or --bf16_full_eval 命令行参数时启用此模式

7.2 NCCL

- 通讯会采用一种单独的数据类型
- 默认情况下，半精度训练使用 fp16 作为reduction操作的默认值
- 可以增加一个小的开销并确保reduction将使用 fp32 作为累积数据类型

```
{
    "communication_data_type": "fp32"
}
```

7.3 apex

- Apex 是一个在 PyTorch 深度学习框架下用于加速训练和提高性能的库。Apex 提供了混合精度训练、分布式训练和内存优化等功能，帮助用户提高训练速度、扩展训练规模以及优化 GPU 资源利用率。
- 使用--fp16、 --fp16_backend apex、 --fp16_opt_level 01 命令行参数时启用此模式

```
"amp": {
     "enabled": "auto",
     "opt_level": "auto"
}
```

八、获取模型参数

- deepspeed会在优化器参数中存储模型的主参数，存储在global_step*/*optim_states.pt 文件中，数据类型为fp32。因此，想要从checkpoint中恢复训练，则保持默认即可
- 如果模型是在ZeRO-2模式下保存的，模型参数会以fp16的形式存储在pytorch_model.bin中
- 如果模型是在ZeRO-3模式下保存的，需要如下所示设置参数，否则pytorch_model.bin将不会被创建

```
{
  "zero_optimization": {
        "stage3_gather_16bit_weights_on_model_save": true
    }
}
```

- 在线fp32权重恢复（需要很多的RAM）略
- 离线获取fp32权重

```
    $ python zero_to_fp32.py . pytorch_model.bin
```

8.1 ZeRO-3 and Infinity Nuances

- 构造超大模型（略）
- 搜集参数（略）

填坑笔记

1. ModuleNotFoundError: No module named 'torch._six

- 报错内容

```
$ ModuleNotFoundError: No module named 'torch._six：找到报错的文件，
```

- 解决方法

```
注释掉：from torch._six import string_classes
加入：
int_classes = int
string_classes = str
如果还报错：NameError: name 'inf' is not defined
找到文件中的那一行，
前面加入：
import math
inf = math.inf
```

2. 为什么单卡的情况，也可以使用deepspeed?

1. 使用ZeRO-offload，将部分数据offload到CPU，降低对显存的需求
2. 提供了对显存的管理，减少显存中的碎片

3. 不同 ZeRO 如何配置

3.1 ZeRO-2

配置示例

```
{
    "fp16": {
        "enabled": "auto",
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },

    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": "auto",
```

```
            "betas": "auto",
            "eps": "auto",
            "weight_decay": "auto"
        }
    },

    "scheduler": {
        "type": "WarmupLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto"
        }
    },

    "zero_optimization": {
        "stage": 2,
        "offload_optimizer": {
            "device": "cpu",
            "pin_memory": true
        },
        "allgather_partitions": true,
        "allgather_bucket_size": 2e8,
        "overlap_comm": true,
        "reduce_scatter": true,
        "reduce_bucket_size": 2e8,
        "contiguous_gradients": true
    },

    "gradient_accumulation_steps": "auto",
    "gradient_clipping": "auto",
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

- 重要参数介绍：
  - **overlap_comm**：**控制是否使用通信与计算的重叠。**当设置为True时，DeepSpeed将在梯度计算时尝试并行执行梯度通信。可以有效地减少通信时间，从而加速整个训练过程。
  - **allgather_bucket_size**：**用于控制Allgather操作的分桶大小。**Allgather操作是指在分布式训练中，每个进程收集其他所有进程的张量，并将这些张量按顺序拼接起来。通过将张量划分为较小的桶（buckets），可以在通信过程中更高效地传输数据。allgather_bucket_size值越大，每个桶的大小越大，通信操作可能会变得更快，但也需要更多的内存来存储中间结果。合适的桶大小要根据实际情况调整。
  - **reduce_bucket_size**：**类似于allgather_bucket_size，用于控制Allreduce操作的分桶大小。**Allreduce操作是将所有进程的某个张量进行规约（例如求和），并将结果广播回

所有进程。通过将张量划分为较小的桶，可以更高效地传输数据。reduce_bucket_size值越大，每个桶的大小越大，通信操作可能会变得更快，但同时也需要更多的内存来存储中间结果。合适的桶大小需要根据实际情况进行调整。

- **overlap_comm使用的是allgather_bucket_size和reduce_bucket_size值的4.5倍**。如果它们被设置为5e8，需要9GB显存（5e8 x 2Bytes x 2 x 4.5）。如果内存大小是8GB或更小，需要将这些参数减少到约2e8，从而避免OOM，这需要3.6GB显存。如果在大容量GPU上也出现OOM，也需要做同样的调整。
- 在deepspeed==0.4.4中新增了**round_robin_gradients**选项，可以**并行化CPU的offload**。当梯度累积的步数增加，或者GPU数量增加时，会有更好的性能优势。

3.2 ZeRO-3

配置示例

```
{
    "fp16": {
        "enabled": "auto",
        "loss_scale": 0,
        "loss_scale_window": 1000,
        "initial_scale_power": 16,
        "hysteresis": 2,
        "min_loss_scale": 1
    },

    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": "auto",
            "betas": "auto",
            "eps": "auto",
            "weight_decay": "auto"
        }
    },

    "scheduler": {
        "type": "WarmupLR",
        "params": {
            "warmup_min_lr": "auto",
            "warmup_max_lr": "auto",
            "warmup_num_steps": "auto"
        }
    },

    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu",
            "pin_memory": true
```

```
        },
        "offload_param": {
            "device": "cpu",
            "pin_memory": true
        },
        "overlap_comm": true,
        "contiguous_gradients": true,
        "sub_group_size": 1e9,
        "reduce_bucket_size": "auto",
        "stage3_prefetch_bucket_size": "auto",
        "stage3_param_persistence_threshold": "auto",
        "stage3_max_live_parameters": 1e9,
        "stage3_max_reuse_distance": 1e9,
        "stage3_gather_16bit_weights_on_model_save": true
    },

    "gradient_accumulation_steps": "auto",
    "gradient_clipping": "auto",
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

- 重要参数介绍：
  - **stage3_max_live_parameters是保留在 GPU 上的完整参数数量的上限。**
  - **stage3_max_reuse_distance是指将来何时再次使用参数的指标，从而决定是丢弃参数还是保留参数。** 如果一个参数在不久的将来要再次使用（小于stage3_max_reuse_distance），可以保留以减少通信开销。 使用activation checkpointing时，这一点非常有用。
  - **如果遇到 OOM，可以减少 stage3_max_live_parameters 和 stage3_max_reuse_distance。** 除非正在使用activation checkpointing，否则它们对性能的影响应该很小。 1e9 会消耗 ~2GB。 内存由 stage3_max_live_parameters 和 stage3_max_reuse_distance 共享，所以不是相加的，一共 2GB。
  - **stage3_gather_16bit_weights_on_model_save 在保存模型时启用模型 fp16 权重合并。** 对大型模型和多GPU，在内存和速度方面都是一项昂贵的操作。 如果打算恢复训练，目前需要使用它。 未来的更新将消除此限制。
  - **sub_group_size 控制在optimizer steps中更新参数的粒度。** 参数被分组到sub_group_size 的桶中，每个桶一次更新一个。 当与 ZeRO-Infinity 中的 NVMe offload一起使用时，sub_group_size 控制模型状态在optimizer steps期间从 NVMe 移入和移出CPU 内存的粒度。 防止超大模型耗尽 CPU 内存。不使用NVMe offload时，使其保持默认值。出现OOM时，减小sub_group_size。当优化器迭代很慢时，可以增大sub_group_size 。
  - **ZeRO-3 中未使用 allgather_partitions、allgather_bucket_size 和 reduce_scatter 配置参数**

3.3 ZeRO-stage-0

配置示例

```
{
    "zero_optimization": {
        "stage": 0
    }
}
```

stage 0会禁用所有的分片，然后把DeepSpeed当作时DDP来使用。

3.4 ZeRO-stage-1

配置示例

```
{
    "zero_optimization": {
        "stage": 1
    }
}
```

只对优化器参数进行分片，可以加速一丢丢

4. ZeRO-3 会比 ZeRO-2 慢很多 如何优化?

- ZeRO-Infinity 需要使用 ZeRO-3
- ZeRO-3 会比 ZeRO-2 慢很多。使用以下策略，可以使得ZeRO-3 的速度更接近ZeRO-2
    - 将stage3_param_persistence_threshold参数设置的很大，比如6 * hidden_size * hidden_size
    - 将offload_params参数关闭（可以极大改善性能）

5. 如何选择不同的Zero stage和offload

从左到右，越来越慢

```
    Stage 0 (DDP) > Stage 1 > Stage 2 > Stage 2 + offload > Stage 3 > Stage 3 + offloads
```

从左到右，所需GPU显存越来越少

```
    Stage 0 (DDP) < Stage 1 < Stage 2 < Stage 2 + offload < Stage 3 < Stage 3 + offloads
```

6. DeepSpeed 遇到问题，如何 确定 调参步骤?

1. 将batch_size设置为1，通过梯度累积实现任意的有效batch_size
2. 如果OOM则，设置--gradient_checkpointing 1 (HF Trainer)，或者 model.gradient_checkpointing_enable()
3. 如果OOM则，尝试ZeRO stage 2
4. 如果OOM则，尝试ZeRO stage 2 + offload_optimizer
5. 如果OOM则，尝试ZeRO stage 3
6. 如果OOM则，尝试offload_param到CPU

7. 如果OOM则，尝试offload_optimizer到CPU
8. 如果OOM则，尝试降低一些默认参数。比如使用generate时，减小beam search的搜索范围
9. 如果OOM则，使用混合精度训练，在Ampere的GPU上使用bf16，在旧版本GPU上使用fp16
10. 如果仍然OOM，则使用ZeRO-Infinity ，使用offload_param和offload_optimizer到NVME
11. 一旦使用batch_size=1时，没有导致OOM，测量此时的有效吞吐量，然后尽可能增大batch_size
12. 开始优化参数，可以关闭offload参数，或者降低ZeRO stage，然后调整batch_size，然后继续测量吞吐量，直到性能比较满意（调参可以增加66%的性能）

7. 如何估算需要的显存?

可以通过下面的代码，先估算不同配置需要的显存数量，从而决定开始尝试的ZeRO stage。

```
    python -c 'from transformers import AutoModel; \
    from deepspeed.runtime.zero.stage3 import
estimate_zero3_model_states_mem_needs_all_live; \
    model = AutoModel.from_pretrained("bigscience/T0_3B"); \
    estimate_zero3_model_states_mem_needs_all_live(model, num_gpus_per_node=2,
num_nodes=1)'
    [...]
    Estimated memory needed for params, optim states and gradients for a:
    HW: Setup with 1 node, 2 GPUs per node.
    SW: Model with 2783M total params, 65M largest layer params.
    per CPU  |  per GPU |   Options
    70.00GB |   0.25GB | offload_param=cpu , offload_optimizer=cpu , zero_init=1
    70.00GB |   0.25GB | offload_param=cpu , offload_optimizer=cpu , zero_init=0
    62.23GB |   2.84GB | offload_param=none, offload_optimizer=cpu , zero_init=1
    62.23GB |   2.84GB | offload_param=none, offload_optimizer=cpu , zero_init=0
    0.74GB |  23.58GB | offload_param=none, offload_optimizer=none, zero_init=1
    31.11GB |  23.58GB | offload_param=none, offload_optimizer=none, zero_init=0
```

8. 启动时，进程被杀死，并且没有打印出traceback

- 问题描述：启动时，进程被杀死，并且没有打印出traceback
- 问题定位：GPU显存不够
- 解决方法：加卡

9. loss是NaN

- 问题描述：loss是NaN
- 问题定位：训练时用的是bf16，使用时是fp16。常常发生于google在TPU上train的模型，如T5。此时需要使用fp32或者bf16。

10. 确保一致性

1. **代码路径 执行训练的代码、模型、数据集等相关文件、路径要一致**（如果不一致，考虑建立软链接）

2. **环境配置 虚拟环境路径也要一样**;
3. **conda各个节点安装路径也要一样**（否则会报exits with return code = 127）；
4. **各种安装的库版本要保持高度一致**（否则会报exits with return code = -6）；
5. 环境路径加载 主节点上默认虚拟环境路径系统加载可能不正确,python环境无法正常加载，所以需要在入口python文件里加入：（这个问题目前没有在部署中遇到，记录一下别人的坑）

```
local_env = os.environ.copy()
local_env["PATH"]="/home/centos/.conda/envs/pretrain6/bin:" + local_env["PATH"]
os.environ.update(local_env)
```

11. 如何配置 配置ssh?

1. hosts中假如节点名称 同时在相关的机器上，vim /etc/hosts，输入：

```
10.58.253.27 model1
10.58.253.26 model2
```

2. 生成sshkey

```
ssh-keygen -t rsa
```

3. 互相拷贝sshkey 自己的sshkey拷贝到对方以及自己的authorized_keys

```
ssh-copy-id ccwork@10.58.253.27
ssh-copy-id ccwork@10.58.253.26
```

4. 测试一下,比如：

```
ssh model2
```

如果不需要输入密码且连成了，则配置成功（服务器需要使用相同的用户名）

12. 如何配置 安装pdsh?

pdsh是一个并行分布式运维工具，它的优点是只需要在一台机上运行脚本就可以，pdsh会自动帮你把命令和环境变量推送到其他节点上，然后汇总所有节点的日志到主节点。

```
sudo apt-get install pdsh
export PDSH_RCMD_TYPE=ssh
```

12. 如何配置 配置deepspeed文件?

1. hostfile 在你的工程下

```
vim hostfile
```

2. 编辑 hostfile

```
model1 slots=8
model2 slots=8
```

3. 运行

```
deepspeed --hostfile=src2/hostfile --include="model2:1,2@model1:3"
src2/train_bash.py  --stage sft  --blabla bla 除了头上这部分，其他都跟单机一样
```