分布式训练1——数据并行

参考：b站 你可是处女座啊（宝藏up）

分布式训练就是将一个模型训练任务拆分成多个子任务，并将子任务分发给多个**计算设备**（即，卡），从而解决资源瓶颈。总体目标就是提升总的训练速度，减少模型训练的总体时间。其中总训练速度可用以下公式简略估计：

$$总训练速度 \propto 单设备计算速度 \times 计算设备总量 \times 多设备加速比$$

单设备计算速度：

- 主要由单卡的运算速度和数据I/O能力决定，主要的优化手段有混合精度训练、算子融合、梯度累加等。

计算设备总量：

- 随着计算设备数量的增加，理论上峰值计算速度会增加，然而受通信效率的影响，计算设备增多会造成加速比急速降低。

多设备加速比：

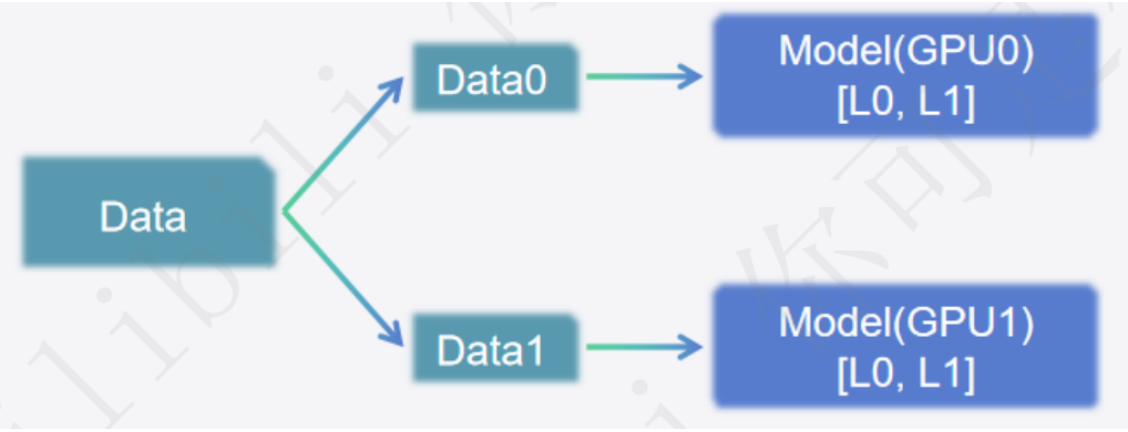
- 这指的是当使用多个计算设备并行处理时，相对于单设备计算速度的加速比例。理想情况下，如果你使用n个设备，理论上的加速比是n倍。但是，由于通信开销、负载不均衡等因素，实际的加速比往往低于理论值。需要结合算法和网络拓扑结构进行优化，例如分布式训练**并行策略。**

1 分布式训练基础

（1）如何进行分布式模型训练

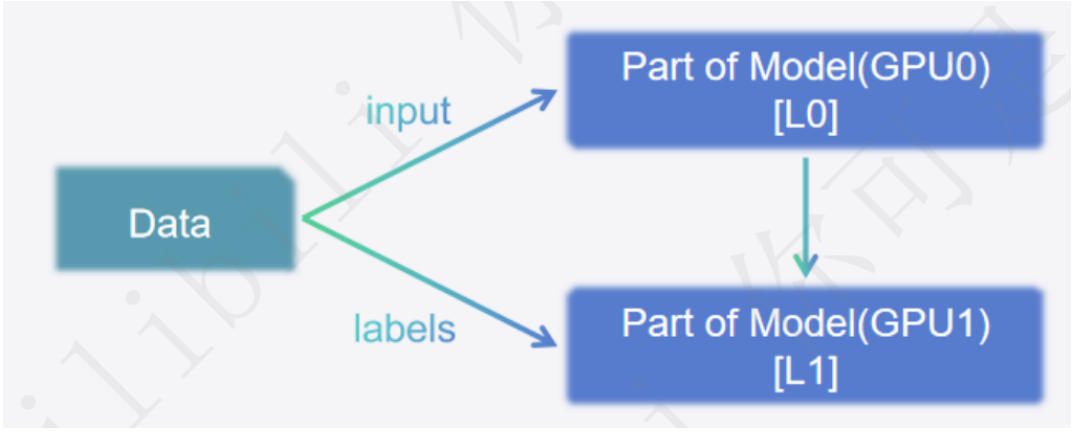
数据并行，Data Parallel，DP

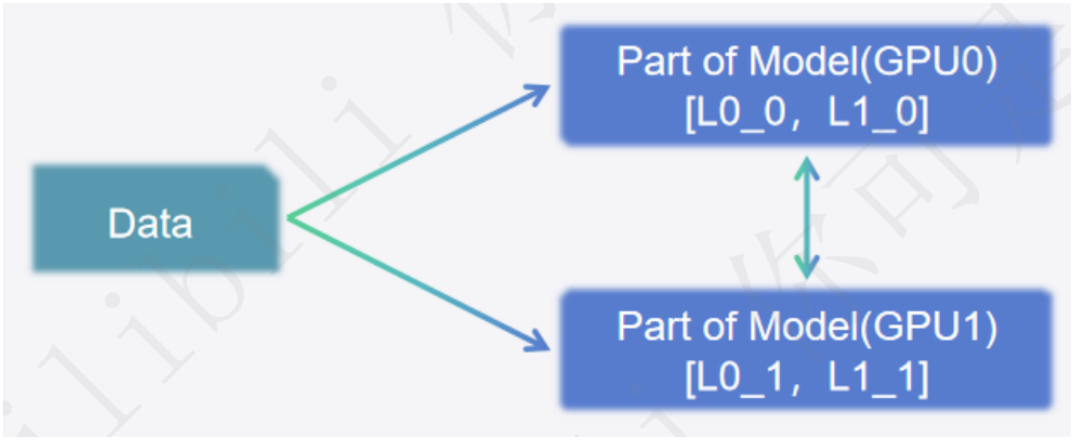
- 对数据进行切分（partition），并将同一个模型复制到多个GPU上，并执行不同的数据分片
- 要求每张卡内都可以**完整执行训练过程**
-



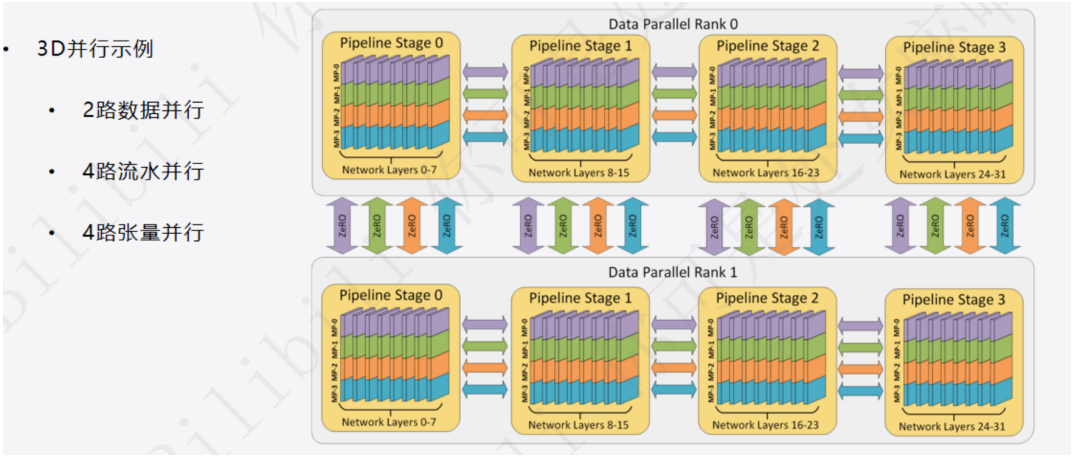
模型并行，Model Parallelism， MP

- 流水并行，Pipeline Parallel，PP
  - 将模型的层切分到不同GPU，每个GPU上包含部分层，也叫层间并行或算子间并行（Inter-operator Parallel）
  - 不要求每张卡内都可以完整执行训练过程

- 

- 张量并行，Tensor Parallel，TP
  - 将模型层内的参数切分到不同设备，也叫层内并行或算子内并行（Intra-operator Parallel）
  - 不要求每张卡内都可以完整执行训练过程
  - 

混合并行

- 数据并行+流水并行+张量并行（3D并行）
  - 示例：
  - 

注：本文主要讲解数据并行，对大多数6B、13B的模型来说，数据并行已经可以满足需求了

（3）环境配置

使用transformers库里的trainer.train()自动就使用了多张卡

2 数据并行 Data Parallel

注1：这里特指Pytorch框架中的nn.DataParallel所实现的数据并行方法

注2：基本不用DP做训练，只用DDP，只是作为DDP的前置知识进行学习

（1）原理

训练流程：



（2）训练实战

看源码

```
class DataParallel(Module, Generic[T]):
    def forward(self, *inputs: Any, **kwargs: Any) -> Any:
        with torch.autograd.profiler.record_function("DataParallel.forward"):
            if not self.device_ids:
                return self.module(*inputs, **kwargs)

            for t in chain(self.module.parameters(), self.module.buffers()):
                if t.device != self.src_device_obj:
                    raise RuntimeError("module must have its parameters and buffers "
                                       f"on device {self.src_device_obj} (device_ids[0]) but found one of "
                                       f"them on device: {t.device}")

            inputs, module_kwargs = self.scatter(inputs, kwargs, self.device_ids)
            # for forward function without any inputs, empty list and dict will be created
            # so the module can be executed on one device which is the first one in device_ids
            if not inputs and not module_kwargs:
                inputs = ((),)
                module_kwargs = ({},)

            if len(self.device_ids) == 1:
                return self.module(*inputs[0], **module_kwargs[0])
            replicas = self.replicate(self.module, self.device_ids[:len(inputs)])
            outputs = self.parallel_apply(replicas, inputs, module_kwargs)
            return self.gather(outputs, self.output_device)
```

分发数据

复制模型

多线程启动

（3）推理对比

需要修改的部分代码

```
model = torch.nn.DataParallel(model, device_ids=None)
# 如果不指定device_ids参数，PyTorch会自动使用所有可用的GPU。

output.loss.mean().backward()
# 否则output.loss会变成多个GPU上的loss值，不是标量，无法.backward()
```

注：要把Batc_size拉大才会有效果

trainer.train()中的是怎么使用DataParallel的：

通过TrainingArguments得到n_gpu>1

```python
class Trainer:
    def _wrap_model(self, model, training=True, dataloader=None):
        model, self.optimizer = amp.initialize(model, self.optimizer, opt_level=self.args.fp16_opt_level)

        # Multi-gpu training (should be after apex fp16 initialization) / 8bit models does not support DDP
        if self.args.n_gpu > 1 and not getattr(model, "is_loaded_in_8bit", False):
            model = nn.DataParallel(model)
```

实际效果：

- 调用了多GPU进行训练，但训练速度没有多大的提升

Data Parallel的问题：

- 单进程，多线程，由于GIL锁的问题，不能充分发挥多卡的优势
- 由于Data Parallel的训练策略问题，会导致一个主节点占用比其他节点高很多
- 效率低，尤其是模型很大batch_size很小的情况，每次训练开始时都要重新同步模型
- 只适用于单机训练，无法支持真正的分布式多节点训练

真正的分布式数据并行

- Distributed Data Parallel

然而，DataParallel还是可以在**并行推理**上发挥作用！（不过还是只能是单节点内的卡）

- DataParallel.module.forward()
- DataParallel.forward()
- DataParallel.forward()改进版——把replicate放到前面，只复制一次模型

使用场景：

- 当你有多张GPU卡时，可以使用DP进行前向传播。DP会将输入数据分割成多份，分别分配到各个GPU上，并行计算前向传播，最后收集各GPU上的输出结果。
- 例如，在RAG模型中，对大量数据进行向量编码时，可以通过DP并行化这个过程，提高效率。

分布式训练2——DDP

参考：b站 你可是处女座啊（宝藏up）

3 分布式数据并行（Distributed Data Parallel，DDP）

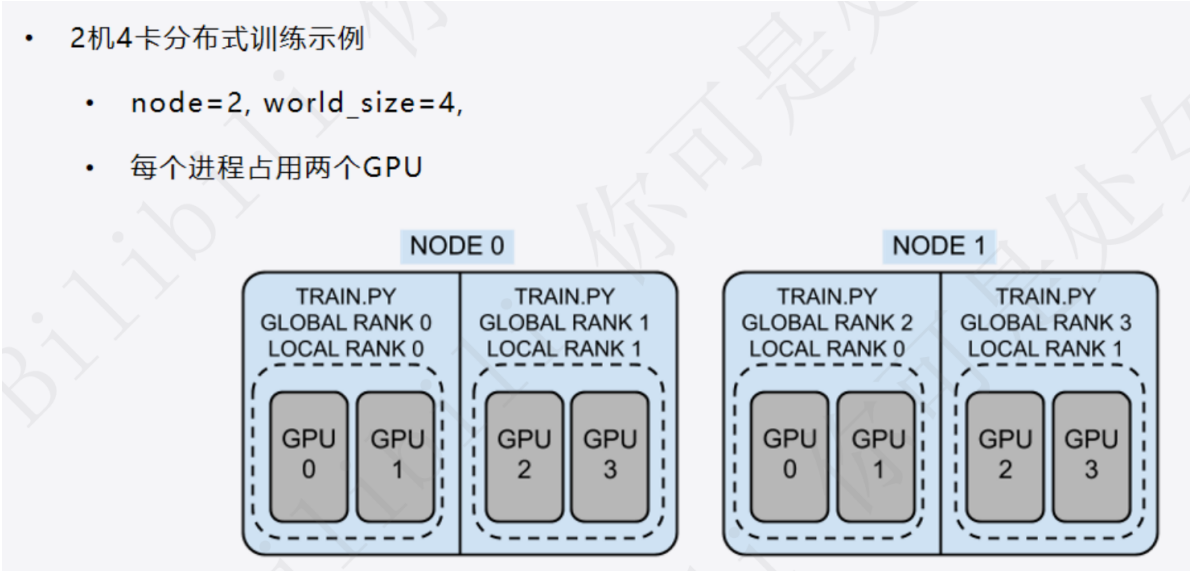
（1）原理



（2）基本概念

- **进程组（group）**：进程组是一个逻辑上的分组，包含参与同一个分布式训练任务的所有进程。对于一个分布式训练任务，所有GPU上的进程通常会被包含在一个进程组里。

- **全局并行数（world_size）**：整个分布式训练任务中参与训练的总进程数。通常等于总的GPU数量，因为每个GPU通常运行一个进程。（但在DP中就是多个GPU运行一个进程）
- **节点（node）**：节点可以是一台机器或一个容器，节点内部通常包含多个GPU。
- **全局序号（rank或global_rank）**：在整个分布式训练任务中，每个进程的唯一标识号。
- **本地序号（local_rank）**：在每个节点内部，每个进程的相对序号。

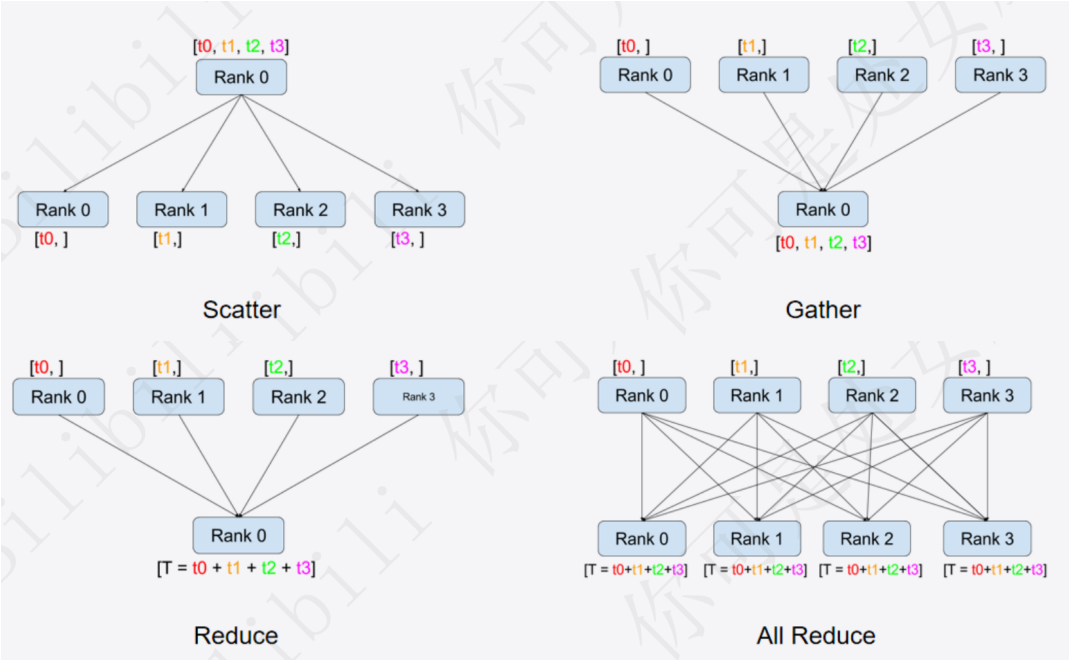


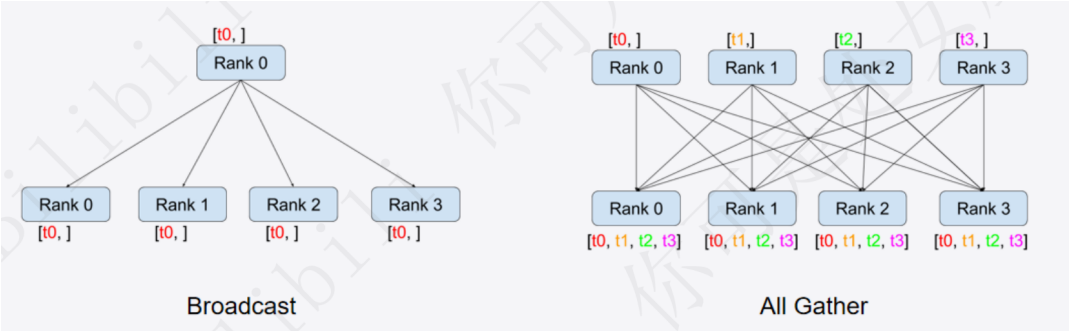
（3）**通信**基本概念（原理中的step4）

什么是通信

- 指的是不同计算节点之间进行信息交换以协调训练任务

通信类型

- 点对点通信：将数据从一个进程传输到另一个进程
- 集合通信：一个分组中**所有进程**的通信模式
  - 6种通信类型：Scatter、Gather、Reduce、All Reduce、Broadcast、All Gather
  - 

- 

（4）实战

初始化group：

```
dist.init_process_group(backend="nccl")
```

Dataloader的部分改造：去除shuffle，加上sampler

```
trainloader = DataLoader(trainset, batch_size=32, collate_fn=collate_func,
sampler=DistributedSampler(trainset))

validloader = DataLoader(validset, batch_size=64, collate_fn=collate_func,
sampler=DistributedSampler(validset))
```

在使用 `DistributedSampler` 时，通常会禁用 `shuffle`，因为 `DistributedSampler` 本身会根据 `epoch` 进行数据的打乱和分配。 `DistributedSampler` 的主要作用是在分布式训练中确保每个进程（GPU）获取不同的数据子集，以避免数据重复使用和数据不平衡问题。

调整模型：

```
model = DDP(model)
```

该进程当前应该用哪个GPU：

```
if torch.cuda.is_available():

  model = model.to(int(os.environ["LOCAL_RANK"]))
```

启动多进程训练：要设置每个节点上有几个进程

```
torchrun --nproc_per_node=2 ddp.py
```

loss的通信部分：原本是在每个GPU上都打印，打印各自的结果，而不是总的平均值

```
dist.all_reduce(output.loss, op = dist.ReduceOp.AVG)
```

只打印一次loss：

```
def print_rank_0(info):
  if int(os.environ["RANK"]) == 0:
        print(info)
```

acc原本是除以整个验证集的长度，需要把多个GPU上的acc汇总

```
dist.all_reduce(acc_num) # op默认就是sum
```

acc有些虚高，是因为数据集的划分导致的，多个进程可能会造成训练集污染验证集的情况

```
trainset, validset = random_split(dataset, lengths = [0.9, 0.1], generator =
torch.Generator().manul_seed(42))
```

手动调用`set_epoch(epoch)`：通知`DistributedSampler`这是新的一个`epoch`，以便它在内部打乱数据。这样做是因为`DistributedSampler`需要知道当前的`epoch`，以便生成不同的随机种子，从而在每个`epoch`中打乱数据。

```
trainloader.sampler.set_epoch(ep)
```

假设验证集只有99条，但是验证集的batch_size是64，怎么平均分在两台GPU上呢？在DistributedSampler里做了填充

```python
class DistributedSampler(Sampler[T_co]):
    def __iter__(self) -> Iterator[T_co]:
        if self.shuffle:
            # deterministically shuffle based on epoch and seed
            g = torch.Generator()
            g.manual_seed(self.seed + self.epoch)
            indices = torch.randperm(len(self.dataset), generator=g).tolist()  # type: ignore[arg-type]
        else:
            indices = list(range(len(self.dataset)))  # type: ignore[arg-type]

        if not self.drop_last:
            # add extra samples to make it evenly divisible
            padding_size = self.total_size - len(indices)
            if padding_size <= len(indices):
                indices += indices[:padding_size]
            else:
                indices += (indices * math.ceil(padding_size / len(indices)))[:padding_size]
        else:
            # remove tail of data to make it evenly divisible.
            indices = indices[:self.total_size]
        assert len(indices) == self.total_size

        # subsample
        indices = indices[self.rank:self.total_size:self.num_replicas]
        assert len(indices) == self.num_samples

        return iter(indices)
```

- Trainer代码的修改

切分数据集时要加入随机种子：

```
datasets = dataset.train_test_split(test_size=0.1, seed=42)
```

- DDP与DP效率对比

- chinese-roberta-wwm-base，单机2卡

Training Strategy	Num GPUs	Batch Size	Spend Time
None	1	64	97.0s
None	1	128	94.0s
None	1	256	OOM
DP	2	64	62.1s
DP	2	128	55.2s
DP	2	256	54.4s
DDP	2	64	63.8s
DDP	2	128	53.6s
DDP	2	256	50.7s

- chinese-roberta-wwm-large，单机2卡

Training Strategy	Num GPUs	Batch Size	Spend Time
None	1	64	331.3s
None	1	128	OOM
None	1	256	OOM
DP	2	64	228.3s
DP	2	128	187.8s
DP	2	256	OOM
DDP	2	64	179.3s
DDP	2	128	165.6s
DDP	2	256	OOM

Accelerate库

参考：b站 你可是处女座啊

1 Accelerate基础入门

（1）Accelerate基本介绍

什么是Accelerate

- Accelerate是Huggingface生态中针对分布式训练推理提供的库，目标是简化分布式训练流程
- Accelerate库本身不提供分布式训练的内容，但其内部集成了多种分布式训练框架DDP、FSDP、Deepspeed等
- Accelerate库提供了统一的接口，简单的几行代码（4行），就可以让单机训练的程序变为分布式训练程序
- Transformers库中也是通过Accelerate集成的分布式训练

```
+ from accelerate import Accelerator
+ accelerator = Accelerator()

+ model, optimizer, training_dataloader, scheduler = accelerator.prepare(
+     model, optimizer, training_dataloader, scheduler
+ )

  for batch in training_dataloader:
      optimizer.zero_grad()
      inputs, targets = batch
      inputs = inputs.to(device)
      targets = targets.to(device)
      outputs = model(inputs)
      loss = loss_function(outputs, targets)
+     accelerator.backward(loss)
      optimizer.step()
      scheduler.step()
```

（2）基于Accelerate DDP代码实现

对ddp代码的初步改造：

```
  accelerator = Accelerator()

  trainloader, validloader = prepare_dataloader()

  model, optimizer = prepare_model_and_optimizer()

  model, optimizer, trainloader, validloader = accelerator.prepare(model,
optimizer, trainloader, validloader)

#改造loss.backward()
accelerator.backward(loss)
```

dataloader部分不需要DistributedSampler，加上shuffle即可：

```
    trainloader = DataLoader(trainset, batch_size=32, collate_fn=collate_func,
shuffle=True)
    validloader = DataLoader(validset, batch_size=64, collate_fn=collate_func,
shuffle=False)
```

删除DDP相关的代码，不需要再放到指定设备上。

暂时用torchrun --nproc_per_node=2启动。但此时acc>1了，因为用pytorch训练时为了让数据平分到两张卡上会做自动填充，怎么去掉多的这部分呢？用gather_for_metrics方法，此时也不再需要all_reduce。修改评估部分的代码：

```
def evaluate(model, validloader, accelerator: Accelerator):
    model.eval()
    acc_num = 0
    with torch.inference_mode():
        for batch in validloader:
            output = model(**batch)
            pred = torch.argmax(output.logits, dim=-1)
            pred, refs = accelerator.gather_for_metrics((pred, batch["labels"]))
            acc_num += (pred.long() == refs.long()).float().sum()
    return acc_num / len(validloader.dataset)
```

修改训练部分的代码，去除all_reduce、修改print：

```
def train(model, optimizer, trainloader, validloader, accelerator: Accelerator,
epoch=3, log_step=10):
    global_step = 0
    for ep in range(epoch):
        model.train()
        for batch in trainloader:
            optimizer.zero_grad()
            output = model(**batch)
            loss = output.loss
            accelerator.backward(loss)
            optimizer.step()
            if global_step % log_step == 0:
                loss = accelerator.reduce(loss, "mean")
                accelerator.print(f"ep: {ep}, global_step: {global_step}, loss:
{loss.item()}")
            global_step += 1
        acc = evaluate(model, validloader, accelerator)
        accelerator.print(f"ep: {ep}, acc: {acc}")
```

（3）Accelerate启动命令介绍

```
accelerate lanch ddp_accelerate.py
```

如何在启动时指定默认信息：

```
accelerate config
```

配置完后的文件就存在/root/.cache/huggingface/accelerate/default_config.yaml

2 Accelerate 使用进阶

（1）混合精度

什么是混合精度训练

- 一种提高神经网络训练效率的技术，结合了FP32和FP16/BF16来进行模型训练。这种方法可以减少GPU内存的使用，同时加速训练。



混合精度训练一定会降低显存占用吗?

不一定:

- 假设模型参数量为M

	混合精度训练	单精度训练
模型	(4+2) Bytes * M	4 Bytes * M
优化器	8 Bytes * M	8 Bytes * M
梯度	(2 +) Bytes * M	4 Bytes * M
汇总	(16 +) Bytes * M	16 Bytes * M

当激活值比较大时，会有明显的显存占用降低!

	混合精度训练	单精度训练
模型	(4+2) Bytes * M	4 Bytes * M
优化器	8 Bytes * M	8 Bytes * M
梯度	(2 +) Bytes * M	4 Bytes * M
激活值	2 Bytes * A	4 Bytes * A
汇总	(16 +) Bytes * M + 2 Bytes * A	16 Bytes * M + 4 Bytes * A

方式一:

```
accelerator = Accelerator(mixed_precision="bf16")
```

方式二:

```
accelerator config &&choice bf16
```

方式三:

```
accelerator launch --mixed_precision bf16 {script.py}
```

(2) 梯度累积

什么是梯度累积

- 一种训练技术，允许模型在有限资源下模拟更大batch_size的训练效果

具体做法

- **分割批次**：将大的训练数据batch分割成多个较小的mini-batch。
- **计算梯度**：对每个mini-batch独立进行前向和反向传播，计算出对应的梯度。

- **累积梯度**：将每个mini-batch计算得到的梯度进行累积，而不是立即更新模型参数。
- **更新参数**：在累积了一定数量的梯度后，使用这些累积的梯度统一更新模型参数。

不使用accelerator的情况下实现梯度累积：

```
accumulation_steps = 4  # 设定累积步数
model.zero_grad()       # 清空梯度
for step, (inputs, targets) in enumerate(data_loader):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss = loss / accumulation_steps  # 缩放损失
    loss.backward()                   # 计算梯度
    if (step + 1) % accumulation_steps == 0:
        optimizer.step()              # 更新参数
        model.zero_grad()             # 清空梯度
```

使用accelerator进行梯度累积

步骤一：

创建Accelerator时指定梯度累积的步数：

```
accelerator = Accelerator(gradient_accumulation_steps = xx)
```

步骤二：

训练过程中，加入accelerator.accumulate(model)的上下文

```
with accelerator.accumulate(model):
        if accelerator.sync_gradients:
          global_step += 1

          if global_step % log_step == 0:
            loss = accelerator.reduce(loss, "mean")
            accelerator.print(f"ep: {ep}, global_step: {global_step}, loss:
{loss.item()}")
            accelerator.log({"loss": loss.item()}, global_step)
```

（3）实验记录

可视化及日志记录

步骤一：

创建Accelerator时指定project_dir

```
accelerator = Accelerator(log_with = "tensorboard", project_dir = "xx")
```

步骤二：

初始化tracker

```
accelerator.init_trackers(project_name = "xx")
```

步骤三：

训练结束，确保所有tracker结束

```
accelerator.end_training()
```

（4）模型保存

方式一：

```
accelerator.save_model(model, accelerator.project_dir + f"/step_{global_step}")
```

存在的问题：

不保存配置文件config.json，只保存模型参数，这样加载模型时会出错

对PEFT模型支持不好，会将完整模型保存

方式二：

```
accelerator.unwrap_model(model).save_pretrained(
    save_directory=accelerator.project_dir + f"/step_{global_step}/model",
    is_main_process=accelerator.is_main_process,
    state_dict=accelerator.get_state_dict(model),
    save_func=accelerator.save
)
```

（5）断点续训

如何进行断点续训：

- 保存检查点（checkpoint）
- 加载检查点（模型权重、优化器状态、学习率调度器、随机状态）
- 跳过已训练数据（epoch、batch）

实现步骤：

步骤一：

保存检查点

```
accelerator.save_state(accelerator.project_dir + f"/step_{global_step}") # 不能只存模型权重，学习率等也要存起来
```

步骤二：

加载检查点

```
accelerator.load_state(resume)
```

步骤三：

计算跳过的轮数和步数

```
resume_step = 0
resume_epoch = 0

if resume is not None:
    accelerator.load_state(resume)

    steps_per_epoch = math.ceil(len(trainloader) /
accelerator.gradient_accumulation_steps) # 向上取整，计算每个epoch需要多少步

    resume_step = global_step = int(resume.split("step_")[-1]) # 拿到当前是多少步
    resume_epoch = resume_step // steps_per_epoch # 当前训练了多少个epoch
    resume_step -= resume_epoch * steps_per_epoch
    accelerator.print(f"resume from checkpoint -> {resume}")
```

步骤四：

数据集跳过对应步数

```
if resume and ep == resume_epoch and resume_step != 0:
    active_dataloader = accelerator.skip_first_batches(trainloader, resume_step
* accelerator.gradient_accumulation_steps)
else:
    active_dataloader = trainloader
```

6 Accelerate集成Deepspeed

DDP存在的问题：

每个GPU上都要存一份完整的模型参数，即单卡至少需要16 * M * Bytes的资源，M为模型参数量

对于大模型全量微调，基本不可能

存在冗余，N张卡上进行DDP训练，内存中需要加载N份模型，N份梯度，N份优化器

Deepspeed介绍

一个由微软开发的深度学习优化库，目标是使分布式训练变得简单、高效。

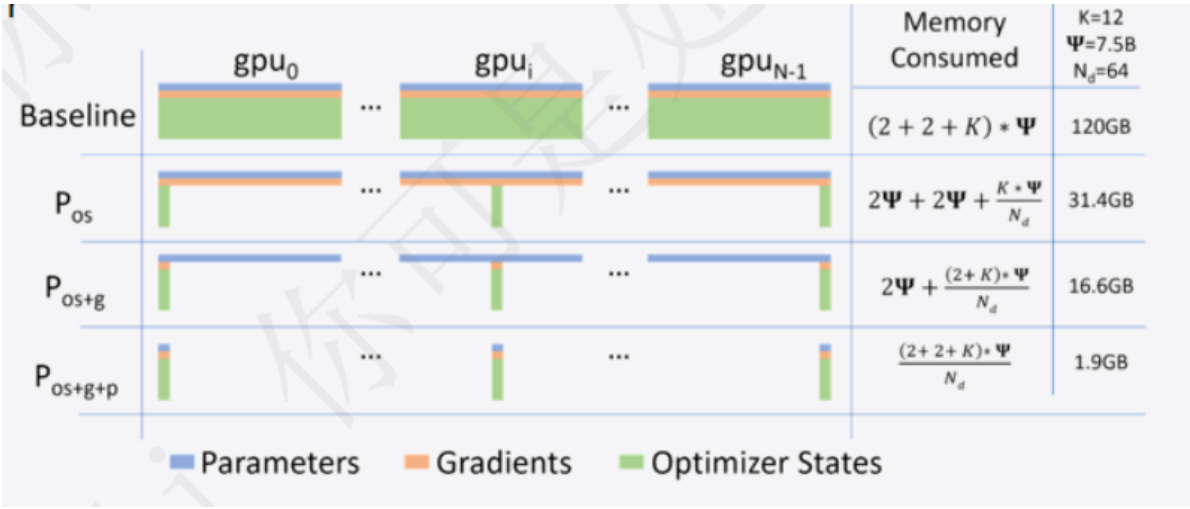
核心：零冗余优化

ZeRO，Zero Redundancy Optimizer

ZeRO1，Optimizer States

ZeRO2，OS + Gradient

ZeRO3， OS + G + Parameter


gpu0
gpui
gpuN-1
Memory Consumed
K=12
Ψ=7.5B
Nd=64
Baseline
(2 + 2 + K) * Ψ
120GB
Pos
2Ψ + 2Ψ + K * Ψ / Nd
31.4GB
Pos+g
2Ψ + (2+ K)* Ψ / Nd
16.6GB
Pos+g+p
(2+ 2+ K)* Ψ / Nd
1.9GB
Parameters
Gradients
Optimizer States

代价：